UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RISCO - MICROPROCESSADOR RISC CMOS DE 32 BITS

por

ALEXANDRE AMBROZI JUNQUEIRA

Dissertação submetida como requisito parcial
para a obtenção de grau de
Mestre em Ciência da Computação

Prof. Altamiro Amadeu Suzim
Orientador

Porto Alegre, setembro de 1993.

CIP - CATALOGAÇÃO NA FONTE

Junqueira, Alexandre Ambrozi

Risco - Microprocessador RISC CMOS de 32 bits / Alexandre Ambrozi Junqueira. - Porto Alegre: CPGCC da UFRGS, 1993.

256p.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1993. Orien-dor: Suzim, Altamiro Amadeu.

Dissertação: Microprocessadores: RISC
VLSI: Circuitos Integrados
Arquitetura: Processadores

Universidade Federal do Rio Grande do Sul
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Cláudio Scherer
Diretor do Instituto de Informática: Prof. Clésio S. dos Santos
Coordenador do CPGCC: Prof. Ricardo A. da Luz Reis
Bibliotecária do Instituto de Informática: Celina Leite Miranda

"Some Books are to be tasted,
others to be swallowed,
and some few to be chewed and digested."

Francis Bacon, *Essays*, 1625;
in *Elements of Linear*
Circuits, R. E. Scott.

Aos amigos, aos familiares e aos mestres, por nos ensinarem dia a dia que vale a pena correr **Riscos...**

AGRADECIMENTOS

Há tanta gente por agradecer. Uma extensa lista seria injusta, pois alguém poderia ser esquecido sem merecer. Assim, aos meus amigos, a todos aqueles que me acompanharam, meu muito obrigado.

Entretanto, há algumas pessoas que tiveram uma importância decisiva na elaboração deste trabalho e merecem citação especial. Na fase inicial, em Campinas, agradeço aos amigos Paulo Motta e Wagner Sanz, e suas famílias, por todo apoio recebido. Os estudos desenvolvidos na UNICAMP e a convivência com seus professores também tiveram muito valor. Em Porto Alegre, agradeço especialmente aos amigos Luigi Carro e Prof. Altamiro A. Suzim pelo incentivo e paciência no desenvolvimento deste trabalho. Agradeço também aos auxiliares de pesquisa André Hentz, Fernando Soto e Daniela Cunha Lima pela colaboração no desenvolvimento do CAD para o Risco e a Márcio Gil Faccin e Paulo Godoy pelos layouts e simulações elétricas do processador. Por fim, agradeço aos funcionários e professores do CPGCC-UFRGS, pelo apoio que sempre prestaram.

Devo gratidão especial também a meus pais, Arno e Déa, e aos meus avós, Alexandre e Marieta, pelo carinho e constante incentivo que sempre me proporcionaram. Da mesma forma, agradeço meus irmãos e suas famílias.

Finalmente, agradeço à Carmen, companheira em todos os sentidos da palavra.

SUMÁRIO

LISTA DE ABREVIATURAS

ADD	instr. soma
ADDC	instr. soma com carry
ADxx	pino de dado e endereço xx
ALE	pino de sinal de amostragem de endereço
AND	instr. e-lógico
APS	campo de instr.: atualização da PSW
BA	Barramento A
BB	Barramento B
BOUT	Barramento de saída
BR	Banco de Registradores
BSIS	Barramento de sistema
C	bit de carry na PSW
C4 a C0	campos de instr.: operação ou teste
CAD	Computer Aided Design
chip	circuito integrado
CI	Circuito Integrado
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
COND	condição de teste
const.	constante
CONST	registrador temporário da constante
CONSTEXT	registrador temporário da constante estendida
CPGCC	Curso de Pós-Graduação em Ciência da Computação
D	estágio de decodificação
dest	destino da operação da instr.
DST	campo de instr.: endereço do registrador destino
E	estágio de escrita de endereço de instr.
E/S	entrada/saída
ECL	Emitter-Coupled Logic
Em	estágio de escrita de endereço de dado de memória
f1	fase 1 de relógio
F1, F0	campos de instr.: formato e operandos
f2	fase 2 de relógio
f3	fase 3 de relógio
fa	primeiro operando da instr.
fb	segundo operando da instr.
fetch	busca de instr.
fig.	figura
FSM	Finite State Machine
FT1	campo de instr. com endereço do 1º registrador fonte
FT2	campo de instr. com endereço do 2º registrador fonte

Gbytes	2^{30} bytes
GGMOD	Gerador de Gerador de Módulos
Giga	2^{30}
GME	Grupo de Microeletrônica do CPGCC
GND	pino de terra elétrico do circuito
GNDP	pino de terra elétrico dos pads
HDC	Hardware Description in C
HDL	Hardware Description Language
I1	estágio 1 de leitura de instr.
I2	estágio 2 de leitura de instr.
INC	unidade de incremento
instr.	instrução
INT	pino de sinal de pedido de interrupção
INTACK	pino de sinal de aceite de pedido de interrupção
JMP	instr. salto
Kbytes	2^{10} bytes
Kg	campo de instr.: constante de 17 bits
Kgh	Kg estendida e rotacionada
Kgl	Kg estendida
Kp	campo de instr.: constante de 11 bits
Kpe	Kp estendida
LAN	Linguagem de Alto Nível
LD	instr. carga de registrador
LDPOD	instr. carga de registrador com pós-decremento
LDPOI	instr. carga de registrador com pós-incremento
LDPRI	instr. carga de registrador com pré-incremento
LSI	Large Scale Integration
m	metro
M1	estágio 1 de acesso de dado de memória
M2	estágio 2 de acesso de dado de memória
Mbytes	2^{20} bytes
MHz	10^{6} Hertz
micra	plural de micron
micron	10^{-6}
MIMD	Multiple Instruction Multiple Data
MIPS	Microprocessor without Interlocked Pipe Stages
MSI	Medium Scale Integration
mux	multiplexador
MW	estágio de escrita de dado lido da memória
n	nano
N	bit de negativo na PSW
nano	10^{-9}
NMOS	N-channel Metal-Oxide-Semiconductor
NOOP	instr. não-opera

NOT	operação lógica complemento de 1
nº	número
O	estágio de operação
op	operação
OR	instr. ou-lógico
Ov	bit de overflow na PSW
overflow	operação aritmética incorreta
pads	circuitos de entrada/saída do CI
pág.	página
PC	registrador contador de programa
PCINC	registrador temporário do PC incrementado
PLA	Programmable Logic Array
PO	Parte Operativa
PSW	registrador palavra de status do processador
P.C.	Parte de Controle
R	estágio de leitura de operandos
RAM	Random Access Memory
RD	pino de sinal de leitura de memória
RDA	registrador temporário da entrada A da UD
RDB	registrador temporário da entrada B da UD
RES	pino de sinal de reset
RISC	Reduced Instruction Set Computer
RLA	instr. rotação à esquerda aritmético
RLAC	instr. rotação à esquerda aritmético com carry
RLL	instr. rotação à esquerda lógico
RLLC	instr. rotação à esquerda lógico com carry
RMEM	registrador temporário para carga de dado de memória
ROM	Read Only Memory
RRA	instr. rotação à direita aritmético
RRAC	instr. rotação à direita aritmético com carry
RRL	instr. rotação à direita lógico
RRLC	instr. rotação à direita lógico com carry
RUA	registrador temporário da entrada A da ULA
RUB	registrador temporário da entrada B da ULA
seg	segundo
Silex	Framework para o CAD do GME, para estações SUN
SLA	instr. deslocamento à esquerda aritmético
SLAC	instr. deslocamento à esquerda aritmético com carry
SLL	instr. deslocamento à esquerda lógico
SLLC	instr. deslocamento à esquerda lógico com carry
SP	registrador apontador de pilha
SR	instr. sub-rotina
SRA	instr. deslocamento à direita aritmético
SRAC	instr. deslocamento à direita aritmético com carry

SRL	instr. deslocamento à direita lógico
SRLC	instr. deslocamento à direita lógico com carry
SS2	campo de instr.: existência de Kp ou FT2
SSI	Small Scale Integration
ST	instr. armazenamento de registrador
STPOD	instr. armazenamento de registrador com pós-decremento
STPOI	instr. armazenamento de registrador com pós-incremento
STPRI	instr. armazenamento de registrador com pré-incremento
SUB	instr. subtração
SUBC	instr. subtração com carry
SUBR	instr. subtração reversa
SUBRC	instr. subtração reversa com carry
T1, T0	campos de instr.: tipo de instr.
u	micron
UC	Unidade da Constante
UCP	Unidade Central de Processamento
UD	Unidade de Deslocamento
UFRGS	Universidade Federal do Rio Grande do Sul
ULA	Unidade Lógico-Aritmética
UPC	unidade do contador de programa
VDD	pino de alimentação elétrica do circuito
VDDP	pino de alimentação elétrica dos pads
VHDL	Very high speed integrated circuit HDL
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
W	estágio de escrita do resultado da operação
WR	pino de sinal de escrita de memória
XOR	instr. ou-exclusivo-lógico
Z	bit de zero na PSW

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta o estudo, a definição e a simulação elétrica e lógica de um microprocessador CMOS de 32 bits, com arquitetura tipo RISC - o Risco. Dentre as principais características do Risco destacam-se: dados, instruções e endereços são palavras de 32 bits; a unidade de endereçamento é a palavra, permitindo um acesso a 4 Giga palavras (16 Gbytes); a comunição com a memória é feita por um barramento multiplexado de 32 bits para dados e endereços; possui 32 registradores de 32 bits, incluídos nestes o contador de programa, o apontador de pilha, a palavra de status do processador e um registrador constante zero; possui um pipeline de instruções de 3 estágios, atingindo no pico de execução uma instrução por ciclo de máquina; e as instruções de salto têm sua execução retardada de uma instrução.

A Arquitetura de Computadores é analisada, em especial as Arquiteturas RISC (*Reduced Instruction Set Computer* - Processador com Conjunto de Instruções Reduzido) e CISC (*Complex...*), mostrando suas características e comparando-as. Algumas máquinas RISC importantes são vistas e o tema de Arquiteturas VLSI e suas implicações tecnológicas no projeto também é abordado.

A arquitetura do Risco é descrita dando-se ênfase aos objetivos do projeto e construindo uma visão geral do processador. O tratamento de exceções é apresentado e o conjunto de instruções é analisado quanto ao formato, aos tipos e ao processamento no pipeline. A organização interna do Risco é tratada em detalhes, descrevendo-se a Parte Operativa (barramentos, o banco de registradores, a unidade

de tratamento da constante, o contador de programa e o incrementador associado, a unidade lógico-aritmética, a unidade de deslocamento/rotação) e a Parte de Controle (o funcionamento do pipeline de instruções, a decodificação, o autômato de controle, a geração e a validação dos comandos). A simulação funcional do Risco, feita em HDC, também é reportada, incluindo o modelamento, os vetores de teste e os resultados.

A implementação do Risco é discutida enfatizando-se alguns blocos críticos quanto à área e ao desempenho. Os barramentos e o banco de registradores, a ULA e a unidade de deslocamento/rotação são estudados em detalhes pela sua importância no desempenho da máquina. Um test chip contendo a maior parte dos blocos funcionais da parte operativa foi construído, tendo sido aprovado nos testes funcionais.

Por fim, faz-se comentários sobre os resultados obtidos, os problemas encontrados e as etapas futuras no desenvolvimento do Risco, além de serem expostas as conclusões finais.

PALAVRAS-CHAVE: Arquitetura, Circuitos Integrados, Concepção, Microprocessadores, Processadores, RISC, VLSI.

TITLE: "RISCO – A 32-BIT CMOS RISC MICROPROCESSOR"

ABSTRACT

This work presents the study, the definition, the electric and logic simulation, and the implementation of some blocks of a 32-bit CMOS microprocessor, with RISC architecture - the Risco. Among Risco's main characteristics it is highlighted that data, instructions and addresses are 32-bit words; the address unit is the word, allowing an access to 4-Giga words (16 GBytes); communication with memory is made through a data and address bus of 32 bits; it has 32 registers of 32 bits, including program counter, stack pointer, processor status word, and a zero constant register; it also has an instruction pipeline of three stages, fully capable of issuing one instruction at the execution peak per every machine cycle; and control flow instructions are implemnted as delayed branches.

A study on computer architecture is carried out, and special attention is given to the RISC (Reduced Instruction Set Computer) and CISC (Complex...) architectures by means of making comparisons between them, showing their main characteristics and listing some important RISC machines. The VLSI architectures are also discussed, giving emphasis to their technological importance for the Risco's project.

Risco's architecture is described, bringing into prominence the aims of the project and an overview of the processor. Exception handling is presented and the instruction set is analysed with regard to format, type and

pipeline processing. Risco's internal organization is dealt with in detail, providing descriptions of the data path (buses, register bank, constant unit, program counter and associated incrementer, barrel shifter) and of the control part (operation of pipeline instruction, as well as decodification, control automaton, generation and validation of commands). Risco's functional simulation, through HDC, is mentioned, including modeling, test vectors, and results.

Risco's implementation is also discussed giving emphasis to some critical blocks in regard to area and performance. Buses, register bank, arithmetic-logic unit, and barrel shifter are dealt with in detail because of their importance concerning the machine performance. A test-chip, containing most of the functional blocks of the data path, was made and successfully passed the functional tests.

Finally, some comments are made with regard to results, main problems, and next stages in the development of Risco.

KEYWORDS: Architecture, Design, Integrated Circuits, Microprocessors, Processors, RISC, VLSI.

1 INTRODUÇÃO

1.1 Motivação

Este trabalho de pesquisa procura alcançar dois objetivos. O primeiro diz respeito à disposição do grupo de microeletrônica do CPGCC-UFRGS no sentido de projetar e implementar um processador integrado simples, visando a obtenção de experiências em projetos VLSI e o desenvolvimento de metodologias e de ferramentas de CAD.

O segundo objetivo, de ordem pessoal, refere-se ao desejo de aprofundar os conhecimentos adquiridos em computação, em especial as arquiteturas VLSI.

1.2 Organização da dissertação

Além deste capítulo inicial, a dissertação está organizada em outros cinco, os quais são descritos resumidamente a seguir.

No capítulo 2, elabora-se um estudo sobre as arquiteturas RISC e CISC, enumerando suas características e aplicações.

A arquitetura do Risco é relatada no capítulo 3, onde se listam as decisões tomadas para a escolha das características do processador.

No capítulo 4, realiza-se um estudo sobre as possibilidades de pipeline em processadores RISC e, a partir das conclusões, a escolha do pipeline do Risco é demonstrada.

A implementação do Risco é discutida no capítulo 5, enfatizando-se o estudo da parte de controle e da parte operativa. Neste capítulo também são relatadas as ferramentas de CAD utilizadas e o circuito de teste implementado.

Finalmente, no capítulo 6, são apresentadas as conclusões do trabalho, dando-se ênfase às perspectivas do projeto no ambiente do CPGCC/UFRGS.

2 ARQUITETURA DE COMPUTADORES

2.1 Introdução

Este capítulo é dedicado a uma análise relativamente detalhada das arquiteturas RISC e CISC. Com base nesta análise são lançados os alicerces para o desenvolvimento do Risco.

2.2 Arquiteturas CISC

2.2.1 Conceituação e Características

A denominação CISC, do inglês *Complex Instruction Set Computer* (/PAT 80/), é usada para designar computadores com conjunto de instruções complexo. Em geral, estas máquinas possuem um grande número de instruções e de tipos de dados, diversos modos de endereçamento, além de formatos variados de instruções e dados. Como exemplo, tome-se o computador Digital VAX 11/780 (/TAB 87/): possui 304 instruções, sendo que em algumas pode-se especificar até 6 operandos, e cada especificador de operando pode variar de 1 a 10 bytes; contém, ainda, 16 modos de endereçamento e 7 tipos de dados, tais como *string* de caracteres de até 64 Kbytes ou ponto flutuante de 128 bits.

As arquiteturas CISC eram o enfoque tradicional no projeto de processadores comerciais até a década de 80, onde cada novo modelo caracterizava-se como uma extensão da geração anterior, principalmente pela inclusão na nova arquitetura (no hardware ou no microcódigo) de primitivas antes realizadas em software (/FER 85/).

Segundo /TOD 86/, o estilo CISC determina duas características intrínsecas nestas máquinas: uma alta codificação das instruções e uma implementação microprogramada da arquitetura. O grande número de instruções, de tipos de dados e de modos de endereçamento torna necessário uma codificação eficiente para que o

tamanho das instruções seja pequeno. Quanto à implementação microprogramada, é decorrente da própria complexidade da arquitetura, permitindo que o projeto e eventuais alterações sejam mais facilitados do que em outros tipos de implementações.

2.2.2 Histórico - O porquê do aumento da complexidade

Ao longo do tempo, computadores evoluíram de máquinas simples, com poucas instruções, para máquinas mais sofisticadas e complexas que seus antecessores. As principais razões para este aumento de complexidade são comentadas a seguir.

1) *Velocidade da memória versus velocidade da UCP.* A memória principal (memória de núcleos magnéticos) era aproximadamente 10 vezes mais lenta do que a velocidade da UCP. Esta restrição tecnológica da época (década de 60 e primeira metade dos anos 70) fazia com que a UCP permanecesse desocupada entre o fim de uma instrução e a próxima, e que primitivas realizadas como sub-rotinas de software fossem muito mais lentas do que primitivas implementadas como instruções. Estes fatos incentivaram a migração de funções antes realizadas em software para microcódigo, e de microcódigo para hardware. Por exemplo, tome-se uma sub-rotina qualquer anteriormente realizada com 10 instruções de um ciclo de máquina cada: sua execução necessitava de 10 ciclos de memória e a UCP permanecia desocupada 9 décimos de cada um destes ciclos; caso esta sub-rotina fosse implementada como uma instrução de 10 ciclos de máquina, a execução se daria em 1 ciclo de memória, com a UCP ocupada todo tempo.

2) *Microcódigo e tecnologia LSI.* A microprogramação tem sido o meio mais usual de implementação para o controle de máquinas algoritmamente muito complexas (/ANC 86/, /OBR 82/, /ZYS 83/). Com o avanço da tecnologia LSI, em especial memórias maiores e menos dispendiosas, a escolha por controle microprogramado

mostrou-se mais atrativa do que outras implementações. Nestas máquinas, o custo para se aumentar o conjunto de instruções é pequeno, pois significa aumentar o número de microinstruções em uma memória de controle já implementada, sem modificar o hardware. Como o tamanho de memórias LSI é fixo e proporcional à potência de 2, geralmente estas não são totalmente ocupadas pelo microprograma, permitindo facilmente sua expansão. Esta característica da implementação LSI permitiu que funções tradicionalmente executadas em software fossem colocadas em microcódigo sem maiores custos adicionais. Por exemplo, tome-se um processador hipotético cuja implementação do controle seja feita por microcódigo com memórias LSI e que, para implementar todas suas funções básicas, tenham sido gastas 600 palavras de micromemória. Haveria, então, espaço para pelo menos mais 424 ($=2^{10}-600$) microinstruções, sem que praticamente nada fosse preciso modificar na implementação física (hardware) atual do processador.

3) *Densidade de código.* Um dos objetivos que se queria alcançar no projeto de computadores era a produção de programas compactos, ou seja, que utilizassem poucas instruções para realizar a tarefa desejada e que necessitassem a menor quantidade possível de memória para serem armazenados. As razões eram a concepção de que um programa com menor número de instruções era executado mais rapidamente (por ter menos acessos à memória) e que o custo das memórias era muito elevado.

Deste modo, o aumento da complexidade permitiu que menos instruções fossem necessárias para realizar uma mesma tarefa. O uso de formatos variados de instruções também contribuiu para uma maior densidade de código. No exemplo dado no item 1, a sub-rotina de software que foi implementada em microcódigo reduziu de 10 instruções para uma cada chamada no programa que a utiliza e, conseqüentemente, diminuiu o gasto de memória deste programa.

4) *Estratégias de marketing.* A inclusão de instruções "poderosas" foi uma importante estratégia de vendas, servindo mais para promover uma arquitetura do que para uso efetivo (/PAT 80/).

5) *Compatibilidade entre máquinas.* A cada novo modelo de uma família de máquinas de um fabricante, novas características e aperfeiçoamentos eram acrescentados.

Uma vez que os custos do software eram cada vez maiores em relação aos do hardware, seria por demais dispendioso refazer todos os programas em utilização para a nova máquina, ainda mais por que programas em Assembly tinham uma parcela significativa do total do software instalado. Deste modo, tornou-se necessário que os novos modelos executassem os programas já desenvolvidos anteriormente, ou seja, que fossem compatíveis.

Isto levou a conjuntos de instruções sempre ampliados em número, já que o antigo conjunto não era reduzido devido à necessidade de compatibilidade, e em complexidade, pois as instruções mais simples e essenciais já estavam presentes nos modelos anteriores.

6) *Suporte para Linguagens de Alto Nível (LAN).* À medida que a programação era cada vez mais feita em LAN, novas instruções foram introduzidas para melhor executar construções típicas destas linguagens, tais como *for*, *while*, comparação de cadeias de caracteres, etc. Estas instruções tendiam a ser mais complexas devido ao nível semântico mais alto.

2.2.3 Críticas

Com os avanços tecnológicos, várias das motivações para o aumento da complexidade perderam sua razão de ser: máquinas construídas com circuitos LSI, memória de núcleos magnéticos e programação em Linguagem Assembly evoluíram para processadores e memórias VLSI com programação predominantemente em LAN. Por outro lado,

estudos feitos sobre o uso de LAN e sobre as implicações do aumento da complexidade mostraram alguns pontos negativos neste tipo de arquitetura. Discute-se a seguir alguns destes pontos.

1) *Memórias semicondutoras e densidade de código.* Com a evolução da tecnologia de memórias semicondutoras, tornando-se estas mais rápidas e menos dispendiosas, deixaram de ser primordiais vários fatores. A velocidade da memória principal (semicondutora) tornou-se semelhante à da UCP, não sendo mais necessário a migração de funções do software para microcódigo por razões de desempenho. O baixo custo das memórias permitiu que a densidade de código também deixasse de ser um fator relevante, além de permitir o uso de entrelaçamento de memórias para acelerar o acesso destas pelo processador.

2) *Microcódigo e processadores integrados.* Ao contrário de implementações LSI, o aumento da memória de controle em processadores integrados - para inclusão de novas instruções ou para aperfeiçoamentos - traz como conseqüência direta um aumento de área interna do CI, com uma provável degradação no desempenho (/ANC 86/,/MAC 92a/).

3) *Compatibilidade de máquinas.* Devido ao sempre crescente custo relativo do software em relação ao hardware, a programação passou a ser feita predominante em LAN e não mais em Assembly. Deste modo, o esforço para se transportar o antigo software para o novo modelo está basicamente na recompilação dos programas fontes de LAN. Até mesmo sistemas operacionais são escritos em LAN (em C, por exemplo), necessitando-se apenas reescrever os compiladores para a nova máquina, já que estes são pouco portáveis devido à direta interação que possuem com a implementação da arquitetura e por continuarem sendo escritos parte em Assembly devido a razões de velocidade e otimização.

4) *Suporte para LAN.* O estudo de programas escritos em LAN utilizados em máquinas CISC demonstraram alguns fatos interessantes.

Em primeiro lugar, os programas gerados por compiladores (programa objeto) utilizavam apenas uma fração do conjunto de instruções de máquina. A principal razão é a dificuldade do compilador (e do programador que o escreveu) em gerar seqüências de código utilizando instruções de máquina com alto nivel semântico (complexas) para as diversas variações encontradas em cada comando da LAN (/PAT 82/, /PAT 85/). Outra razão é o fato de que certas instruções complexas têm aplicação para uma LAN, mas são inúteis para outras linguagens (compare-se COBOL x C x FORTRAN, por exemplo). A tendência foi, então, padronizar certas seqüências para cada comando de cada LAN e desprezar o uso das instruções complexas, pois apenas em poucas construções estas podiam ser sempre aplicadas. Em /PAT 80/ é relatado um estudo de um compilador do IBM 360: 10 instruções representam 80% de todas as instruções executadas, 16 representam 90%, 21 representam 95%, e 30 representam 99%. Em outro compilador, no IBM 370, apenas 84 das 183 instruções são utilizadas (46% do total). Entre as 84, 26 representam 90% (14% do total) e 48 representam 99% (26% do total).

Outro aspecto importante foi a constatação de que apenas um pequeno número de comandos de LAN e de instruções de máquina eram responsáveis pela imensa maioria das operações realizadas. Em /KAT 85/ vários estudos sobre a freqüência de operações em LAN são reunidos e analisados, donde se reproduzem alguns dados:

- Nas instruções de máquina geradas por compiladores de LAN, 33% são *load*s, 10% *store*s, 14% *branch*s e 6% *compare*s (total de 63%) (/ALE 75/).

- Em programas fonte de LAN, 42% dos comandos são atribuições, 13% são *if*s, e 13% são chamadas de sub-rotina (total de 68%) (/ALE 75/).

- Nos comandos de LAN executados, 42% são atribuições, 36% são *ifs*, 14% são chamadas e retornos de sub-rotinas, e 4% são laços (total de 96%) (/PAT 82/).

- Nas instruções de máquina executadas, 13% são devidas a comandos LAN de atribuição, 16% devido a *ifs*, 32% devido a chamadas e retornos de sub-rotinas, e 37% devido a laços (total de 98%) (/PAT 82/).

Apesar dos dados acima não serem absolutos (as medições foram feitas em diferentes programas, com diferentes compiladores e em diferentes arquiteturas), eles mostram que somente um reduzido número de comandos de LAN e de instruções de máquina tem grande predominância na execução dos programas em LAN. A conseqüência direta deste fato é que a parte de controle, que tem grande parcela de seu código microprogramado pouco utilizado, degrada o desempenho de todas as instruções.

Do exposto acima, observa-se que a inclusão de instruções complexas para diminuir a distância semântica entre a Linguagem de Máquina e as Linguagens de Alto Nível não se revelou uma boa alternativa para melhor executar programas em LAN.

2.3 Arquiteturas RISC

2.3.1 Conceituação e características

A sigla RISC, do inglês *Reduced Instruction Set Computer* (/PAT 80/), tem sido usada para designar máquinas com conjunto reduzido de instruções - e com instruções simples. Pelos motivos expostos em 2.2.3, a arquitetura RISC surgiu como uma alternativa de projeto às máquinas CISC, na tentativa de melhor utilizar os recursos de silício no projeto de processadores e de executar mais eficientemente programas escritos em LAN.

Encontram-se na literatura específica várias definições de quais seriam as características essenciais para uma máquina ser classificada como RISC. Como RISC é

muitas vezes mais uma filosofia de projeto do que uma arquitetura caracterizada por um número fixo de atributos, procurou-se enumerar a seguir quais são os principais fatores de projeto e quais suas conseqüências para se obter uma máquina RISC (/TAB 88/,/STA 88/,/MIL 88/,/MIL 89/).

1) *Arquitetura de Carga/Armazenamento.* Os dados são operados somente entre registradores internos, e apenas as instruções de carga (registrador recebe conteúdo de memória) e armazenamento (memória recebe conteúdo de registrador) acessam a memória. Esta característica possibilita que a execução seja mais rápida, pois o endereço dos operandos não precisa ser calculado (já está contido na palavra de instrução) e o acesso aos operandos não precisa ser feito na memória (já estão nos registradores internos da UCP).

Para que a arquitetura de carga/armazenamento permita um efetivo aumento de desempenho, é necessário também que a UCP possua muitos registradores de uso geral para manter os cálculos intermediários e operandos que ainda serão reutilizados durante a execução do programa, diminuindo o fluxo de dados com a memória.

A arquitetura de carga/armazenamento é a chave para se obter o objetivo principal: a execução em um único ciclo das instruções simples (como já foi exposto, as mais predominantes), possibilitando um alto desempenho. O ciclo único fornece também um baixo custo interpretativo (/TOD 86/).

2) *Simplificação e redução do número de formatos de instruções (≤4).* Sem a existência de instruções de tamanhos e formatos variáveis, bem como múltiplos operandos, fica facilitada a decodificação das instruções, permitindo a possível eliminação (ou redução drástica de tempo) deste estágio no pipeline de instruções e a realização em paralelo da decodificação de cada campo na palavra de instrução.

3) *Simplificação e redução do número de instruções (≤100) e dos modos de endereçamento (≤3).* Facilitam a interpretação das instruções pela parte de controle devido à regularidade e às poucas operações existentes.

Estes três fatores, em conjunto, permitem o surgimento de outros quatro atributos igualmente importantes em máquinas RISC, os quais são citados a seguir.

4) *Parte de controle simples.* Uma das grandes vantagens de arquiteturas RISC está na simplicidade da parte de controle, possibilitando que esta seja implementada em lógica fixa (*hardwired*), sem microcódigo. Decorrem desta simplicidade algumas características, particularmente importantes se a implementação for em VLSI:

- Menor área da parte de controle (P.C.). Arquiteturas RISC apresentam a área da P.C. drasticamente menor do que em máquinas CISC. O percentual da P.C. em relação à área total é de aproximadamente 53% no Zilog Z8000, 62% no Motorola 68000 e 65% no Intel iAPX-43201, enquanto que em máquinas RISC encontra-se 8% no RISC II e 18% no MIPS (/FIT 82/). Isto possibilita que a implementação da arquitetura seja feita em tecnologias que não permitem grande escala de integração, por exemplo Arsenieto de Gálio (/FOX 86/). Também possibilita a inclusão de diversos recursos, tais como grandes bancos de registradores, memória Cache interna, unidade aritmética de ponto flutuante, unidade de gerenciamento de memória, etc. Com a evolução da tecnologia, estes recursos estão sendo incorporados em máquinas CISC, mas esta mesma evolução permitirá a implementação de vários RISC operando de forma multi-processada em uma única pastilha de CI.

- Menor tempo de projeto. A simplicidade e o tamanho reduzido da P.C. permitem que esta seja rapidamente projetada, depurada e eventualmente modificada, tornando as chances de erro de projeto menores. Pela mesma razão,

permite que os custos sejam menores, tanto pelo menor tempo de projeto em si, quanto pelo fator de regularidade maior. Como geralmente a parte operativa (PO) é *bit-slice* (as células são repetidas) e a P.C. é gerada automaticamente, apenas uma pequena porção de todo o circuito terá de ser efetivamente desenhada. Finalmente, a implementação da arquitetura tem menos chances de se tornar obsoleta devido ao tempo de projeto, já que a tecnologia de integração tem atualmente tempo de vida em torno de dois anos.

- Menor tempo de propagação dos sinais. O caminho crítico dos dados não reside mais na P.C. e sim na PO, permitindo um ciclo de máquina menor. Deste modo, a otimização de velocidade do processador é feita principalmente nas unidades funcionais da PO, como na ULA e banco de registradores. Por serem, em geral, circuitos combinacionais e gerados principalmente pela duplicação de células de 1 bit, a otimização fica facilitada se comparada, por exemplo, a uma lógica randômica da parte de controle.

5) *Pipeline eficiente.* A implementação de pipeline de instruções é facilitada pela uniformidade dos formatos de instrução (o estágio de decodificação deixa de ser um ponto crítico de tempo) e pela arquitetura de carga/armazenamento (o ciclo único e a eliminação de múltiplos acessos à memória em uma mesma instrução facilitam a redução de estágios do pipeline e a regularidade no funcionamento deste).

6) *Uso de compilador otimizado.* O número reduzido de instruções permite que o compilador seja projetado para otimizar o código, reorganizando a seqüência de instruções para evitar falhas no pipeline, movendo as complexidades de tempo de execução para tempo de compilação e organizando a utilização dos registradores para evitar cargas, armazenamentos e cálculos de endereços redundantes. Entre as otimizações mais comuns estão a avaliação de constantes em tempo de compilação, movimento de expressões constantes

para fora de laços e salvamento de resultados intermediários comuns a várias expressões ou blocos.

7) *Hierarquia de memória elaborada.* Devido ao menor tempo de ciclo de máquina, à execução em ciclo único e ao uso de pipeline, torna-se necessário o uso de vários níveis de memória (muitos registradores, memória Cache, entrelaçamento de memória, etc) para manter a UCP ocupada e evitar falhas no pipeline.

2.3.2 Algumas máquinas RISC importantes

2.3.2.1 O minicomputador IBM 801

O minicomputador 801 foi desenvolvido a partir de 1975 no IBM Thomas J. Watson Research Center, New York, EUA (/RAD 83/). Apesar de não ter sido produzido comercialmente, foi um dos primeiros projetos de pesquisa a procurar um desvio do estilo das arquiteturas tradicionais. O 801 é considerado uma máquina RISC, ainda que esta conceituação só fosse criada em 1980 (/PAT 80/).

Três conceitos básicos direcionaram o projeto do 801. Primeiro, construir uma UCP que pudesse executar suas instruções rapidamente, isto é, em um único e pequeno ciclo de máquina. Deste modo, o conjunto de instruções deveria ser primitivo o suficiente para poder ser implementado em lógica fixa (sem microprogramação). Segundo, determinar uma hierarquia de armazenamento e organização de E/S para permitir que a UCP executasse uma instrução em quase todos os ciclos, impedindo que esta tivesse de esperar freqüentemente por acessos ao armazenamento. Finalmente, o projeto e utilização de todo o sistema deveria ser orientado à programação em LAN, com o uso profundo do sofisticado compilador 801 (/RAD 83/).

A partir destes conceitos, a arquitetura foi elaborada sobre três princípios de projeto. De acordo com Radin (/RAD 83/), "a arquitetura do 801 foi definida como

aquele conjunto de operações em tempo de execução que: não podiam ser movidas para tempo de compilação; não podiam ser mais eficientemente executadas por código objeto produzido por um compilador que compreendeu as intenções de alto nível do programa; ou eram para ser implementadas em lógica randômica mais efetivamente do que a seqüência equivalente de instruções de software".

O 801 possui 120 instruções, com arquitetura de 32 bits (instruções, endereços, operações e dados de 32 bits). Possui 32 registradores de 32 bits de uso geral, sendo uma máquina de 3 endereços de operandos: um registrador destino recebe o resultado da operação entre dois registradores fonte. É uma máquina de carga/armazenamento, onde apenas as instruções de carga e de armazenamento acessam a memória, com somente dois modos de endereçamento: base mais índice e base mais imediato. Para as instruções de desvio, três modos de endereçamento são fornecidos: absoluto, contador de programa mais imediato e registrador mais registrador.

Um protótipo foi concluído em 1979 com CIs SSI/MSI comerciais de tecnologia ECL, permitindo um ciclo de relógio de 66 nseg e taxa de processamento de 10 MIPS. Com o mesmo compilador do IBM-370/168, o 801 executa programas 1,5 vezes mais rapidamente devido, basicamente, ao menor número de referências à memória, ao menor código compilado e ao menor número de instruções executadas.

Sendo um dos primeiros projetos RISC, o 801 apresentou algumas soluções de implementação que são seguidas até hoje. Entre estas estão o salto retardado (*delayed branch*), um compilador otimizador de código e, inserido neste, um algoritmo para alocação de registradores. Estas otimizações são explicadas resumidamente a seguir.

Uma das dificuldades na implementação de pipeline de instruções ocorre quando da execução de uma instrução

de salto, pois as instruções seguintes a esta, já no pipeline, terão de ser desconsideradas caso o salto ocorra (na condição verdadeira). Para reduzir esta perda de desempenho, técnicas de predição do salto podem ser utilizadas, tanto em hardware como em software.

Uma técnica de predição estática em software para o salto é o salto retardado de instruções, isto é, a execução do salto ocorrerá somente após a instrução seguinte, no caso de salto retardado de uma instrução. Em outras palavras, se o salto é retardado de *n* instruções, as *n* instruções buscadas após a instrução de salto serão executadas antes que o fluxo do programa seja desviado. Isto permite que o pipeline não seja esvaziado, cabendo ao compilador inserir *NOOP*s após o salto ou, então, trocar a ordem de instruções quando possível. Por exemplo, troca-se a instrução de salto por uma instrução imediatamente anterior, caso esta não manipule dados necessários ao salto (/GRO 81/). Um exemplo está na figura 2.1.

Endereço	Normal	Retardado	Retardado Otimizado
100	ADD C,D	ADD C,D	JUMP 500
101	JUMP 500	JUMP 500	ADD C,D
102	ADD A,B	NOOP	ADD A,B
103	...	ADD A,B	...
104		...	
...			
500	...	...	...

ADD A,B no Salto normal já está no pipeline e será descartada, nos outros dois casos não estará no pipe.
ADD C,D no salto retardado otimizado é executada.

Figura 2.1 Salto retardado

O salto retardado facilita a implementação, pois não é necessário incluir mecanismos em hardware que esvaziem e reinicializem o pipeline a cada salto, permitindo que este se mantenha mais próximo da sua taxa máxima possível. No 801, 60% das instruções de desvio são do tipo salto retardado, sem a inserção de *NOOP* (o 801 também

possui instruções de salto sem retardo). A penalidade no uso deste esquema é o aumento de código compilado, em torno de 50% no 801 (/RAD 83/). No RISC II, de Berkeley, o aumento é de 6% aproximadamente, conforme /KAT 85/.

Outra característica importante do 801 foi o projeto do seu compilador, que realizava diversas otimizações de código. Entre estas estão a reorganização da seqüência de instruções para evitar falhas no pipeline, a avaliação de constantes em tempo de compilação, o movimento de expressões constantes para fora de laços e o salvamento de resultados intermediários comuns a várias expressões ou blocos. Um fator marcante no compilador do 801 era a incorporação de um algoritmo para alocação global de registradores, como explicado abaixo (/CHA 82/).

O método é baseado no algoritmo de coloração de grafos. Inicialmente um número arbitrário de registradores é assumido. O compilador coloca, então, cada variável e o resultado em registradores separados. Após determinar o tempo de utilização de cada variável durante a execução do programa, estas são mapeadas para um conjunto finito de registradores de modo a minimizar o número de acessos à memória. Variáveis com tempo de utilização não coincidentes são designadas para um mesmo registrador físico. Se o número de registradores é menor que o número de variáveis ativas ao mesmo tempo, algumas destas são armazenadas em memória.

No 801, com 32 registradores, estudos demonstraram que operandos estão disponíveis nos registradores durante 95% do tempo. Este dado demonstra que um maior esforço, em tempo de compilação, na alocação dos registradores traz vantagens apreciáveis, principalmente na redução dos acessos à memória para busca de operandos.

Um exemplo simplificado da aplicação do algoritmo para 4 registradores físicos está na figura 2.2, onde o compilador deverá armazenar a variável G em memória por falta de registradores.

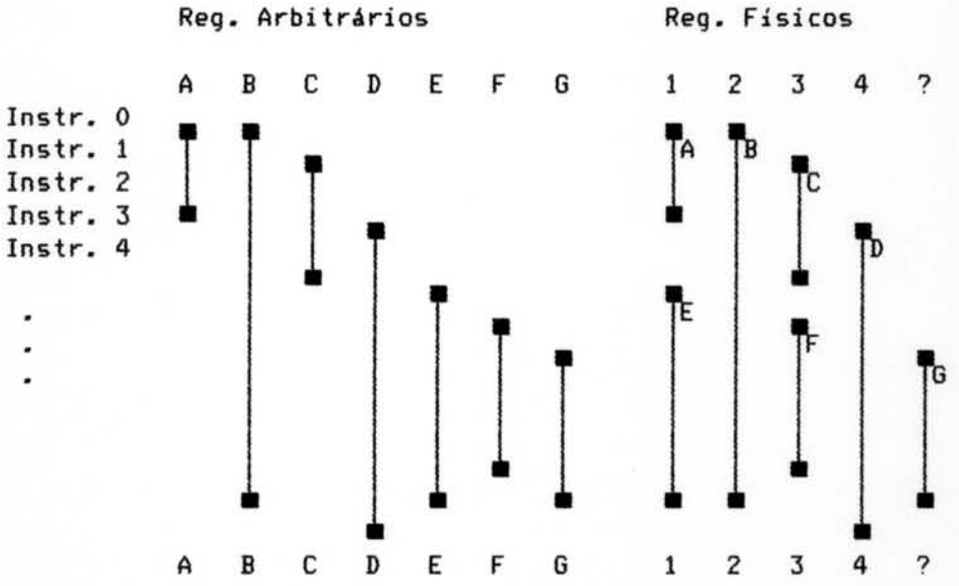


Figura 2.2 Alocação de registradores

Como se infere do exposto acima, o 801 foi uma mudança profunda em relação aos modelos de arquitetura de até então, pois implicava instruções simples e primitivas, execução em ciclo único e implementação em lógica fixa, além de ser voltado ao uso de código compilado. Por isso é considerado uma das primeiras máquinas RISC.

2.3.2.2 O Processador RISC da UCB

Os primeiros microprocessadores a explorar o conceito de instruções simplificadas foram o RISC I e RISC II da Universidade de Berkeley (UCB), Califórnia, sendo também os primeiros implementados em CI único. A filosofia era usar hardware simples para prover suporte eficiente para a execução de LAN. O RISC I foi projetado e fabricado entre 1981 e 1982, o projeto do RISC II foi iniciado em 1981 e concluído um ano após o RISC I. O RISC II, sob o qual se concentra este texto, foi uma evolução do seu antecessor,

com um número maior de registradores, um pipeline de 3 estágios e 8 instruções a mais.

O processador RISC II possui arquitetura de 32 bits do tipo carga/armazenamento orientada a registradores. Dois formatos de instrução são utilizados. Em ambos, os primeiros 7 bits indicam o código da instrução (do máximo de 128 instruções possíveis, apenas 39 foram codificadas), 1 bit indica quando o resultado da operação deve alterar os códigos de condição e os demais 24 bits indicam os operandos.

Para a maioria das instruções, o primeiro formato é utilizado, sendo as operações realizadas registrador a registrador, com 3 endereços de operandos contidos na palavra de instrução: $R_d \leftarrow R_{s1}$ *op* $S2$, onde R_d e R_{s1} são respectivamente os registradores destino e primeiro operando, e $S2$ é o segundo operando, podendo ser o registrador R_{s2} ou a constante *imm13* de 13 bits em complemento de 2. As operações (*op*) são simples e reduzidas: adição, subtração e subtração reversa ($-R_{s1} + S2$) de inteiros, com ou sem *carry*; *e*, *ou* e *ou-exclusivo* lógicos bit-a-bit; e deslocamentos de qualquer valor do tipo *esquerda-lógico*, *direita-lógico* e *direita-aritmético*.

Para as instruções de salto, a operação é $PC \leftarrow R_{s1} + S2$, sendo que o campo da palavra de instrução antes ocupado por R_d contém agora a condição a ser testada (o registrador destino, o contador de programa *PC*, é implícito).

As únicas instruções que acessam a memória são as de carga, $R_d \leftarrow M[R_{s1} + S2]$, e as de armazenamento, $M[R_{s1} + S2] \leftarrow R_d$, havendo instruções separadas para dados tipo byte, meia-palavra (16 bits) e palavra (32 bits).

Para as instruções com endereço relativo ao *PC*, existe um segundo formato de instrução, permitindo saltos, $PC \leftarrow PC + imm19$; carga, $R_d \leftarrow M[PC + imm19]$; e

armazenamento, $M[PC + imm19] \leftarrow R_d$; sendo *imm19* uma constante de 19 bits em complemento de 2.

As demais instruções controlam chamadas e retornos de sub-rotinas, interrupções e palavra de status do processador.

O RISC II executa suas instruções em três ciclos de máquina: o primeiro busca e decodifica a instrução, o segundo faz a leitura de dois operandos, a operação entre estes e o armazenamento temporário do resultado e, no último, é feita a escrita do resultado no banco de registradores. Como é implementado um pipeline de instruções de 3 estágios, uma nova instrução é iniciada a cada ciclo de máquina. A exceção são as instruções de carga e armazenamento, onde mais um ciclo é gasto, parando-se o pipeline para fazer o acesso à memória.

A implementação do RISC II foi feita em tecnologia NMOS de 4 micra e uma camada de metal, atingindo 8 MHz de relógio e taxa de processamento de até 4 MIPS no pico (as instruções de acesso à memória ocupam mais um ciclo, diminuindo a taxa máxima). Uma segunda implementação, com processo de 3 micra, obteve valores de 12 MHz e 6 MIPS, respectivamente.

O RISC II buscou outra alternativa para diminuir o tráfego de dados com a memória. O 801 e o MIPS baseavam-se no compilador (software) para otimizar o uso dos registradores. O RISC II implementou fisicamente um grande banco de registradores (138) divididos logicamente em vários conjuntos (janelas) parcialmente sobrepostos e gerenciados segundo uma estrutura de pilha, tendo como objetivo principal diminuir o custo da execução das chamadas de sub-rotinas nas LAN.

O conjunto de registradores é organizado em múltiplos bancos de 32 registradores parcialmente sobrepostos (janelas). Cada procedimento criado tem somente

32 registradores a sua disposição (uma janela). Cada janela é dividida em duas partes:

a) Registradores Globais, de 0 a 9, podendo ser utilizados por qualquer procedimento do programa em execução. São utilizados para armazenar variáveis globais.

b) Registradores de Janela, de 10 a 31 (total de 22), particulares a um procedimento. Estes registradores são novamente subdivididos: de 10 a 15 são denominados grupo baixo (total de 6), de 16 a 25 de grupo local (10), e de 26 a 31 de grupo alto (6). São utilizados para armazenar variáveis locais e parâmetros de procedimentos.

Existem oito janelas e cada procedimento está associado a uma janela. Quando ocorre a chamada de uma sub-rotina, uma nova janela de registradores é alocada (na estrutura de pilha), sobrepondo-se em parte sobre a janela do procedimento anterior. Deste modo, os registradores de 26 a 31 (grupo alto) do procedimento chamador anterior passam a ser os registradores de 10 a 15 (grupo baixo) do novo procedimento (chamado), permitindo assim a passagem de parâmetros sem a necessidade de acessos à memória nem de cálculos de endereços. Quando da conclusão da rotina atual, retorna-se à janela anterior, passando os resultados do procedimento pelos mesmos registradores. Um esquema da pilha de conjunto de registradores está na figura 2.3.

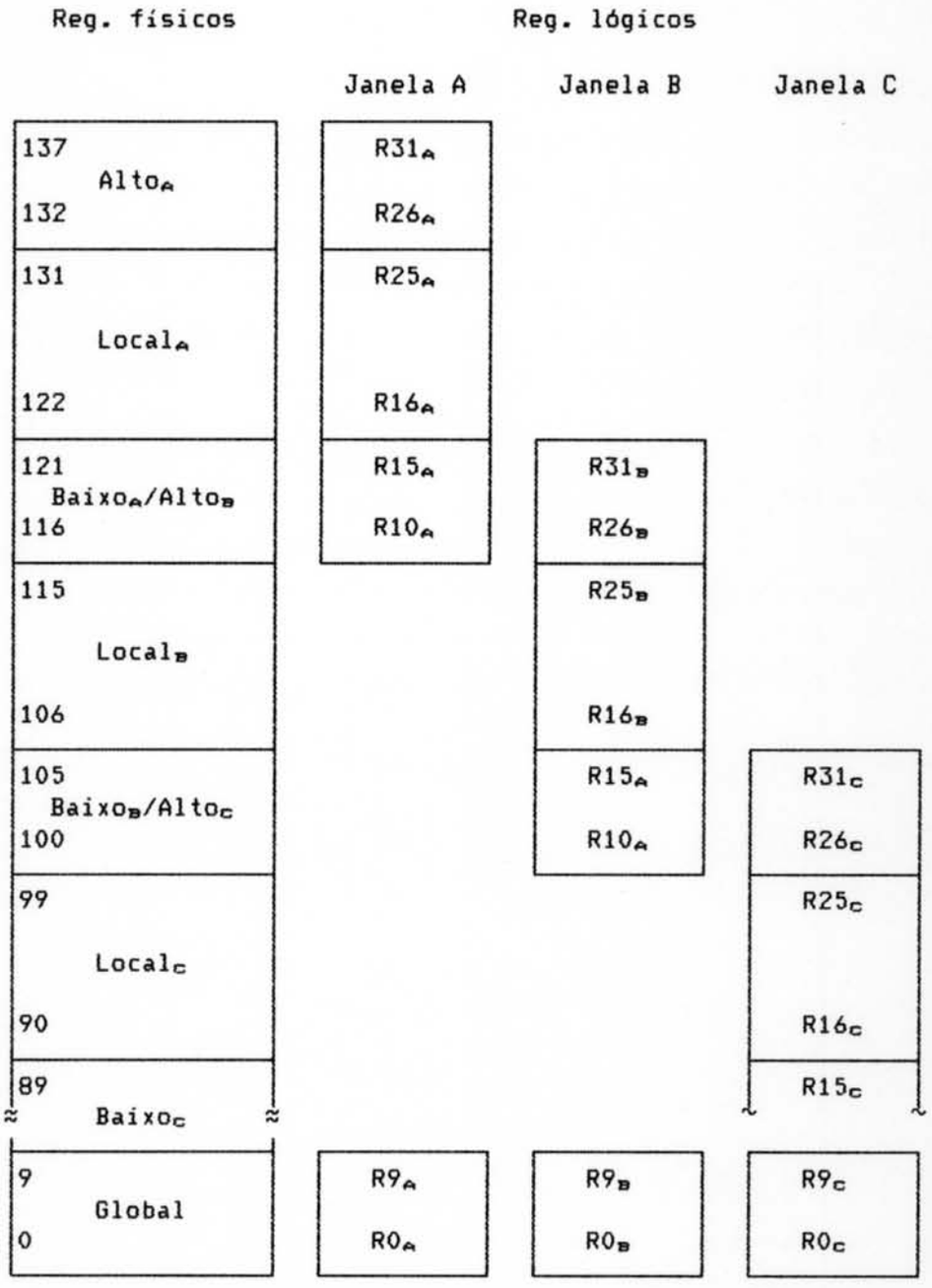


Figura 2.3 Pilha de registradores no RISC II

Apesar desta técnica já ser bastante utilizada comercialmente (/CYP 90/, /SUN 87/), algumas desvantagens são apontadas a seguir.

A ocupação de uma grande área em silício com apenas uma fração (uma janela) ativa a um determinado tempo é fator negativo pois, caso o número de bits seja grande, o uso de uma memória interna tipo cache associativa para os

dados torna-se mais atrativo, visto a memória estar ativa todo o tempo.

Outra desvantagem ocorre quando da troca de contexto durante um chaveamento de processos, pois o estado atual do processador deve ser salvo. Apesar da pouca freqüência em relação às chamadas de procedimentos, o tempo de chaveamento aumenta consideravelmente pois todas as janelas (o estado) devem ser salvas em memória.

Por fim, com o aumento do número de registradores o tempo de acesso a estes também aumenta, tornando-se necessário um compromisso entre área ocupada (devido ao número de bits ou à arquitetura do banco utilizada para acelerar o acesso) e velocidade (mais registradores aumentam a carga capacitiva do barramento no qual estão conectados).

O RISC II demonstrou que, efetivamente, um processador com conjunto de instruções simples e reduzido pode ser uma alternativa melhor para a execução de LAN. A comparação da execução de programas em C com dados inteiros entre o RISC II @ 8MHz e outros processadores (iAPX-286 @ 8MHz, NS 16032 @ 10MHz, M68000 @ 12MHz) mostraram um desempenho maior do RISC II (/TOD 86/).

2.3.2.3 O Processador MIPS da SU

O processador MIPS (*Microprocessor without Interlocked Pipe Stages*), da Universidade de Stanford (SU), teve seu projeto baseado na idéia-chave de expor no conjunto de instruções todas as atividades do processador que pudessem afetar o desempenho. Para tal, utilizou-se um conjunto de instruções extremamente simples e eficiente, semelhante a um microcódigo, bem como uma tecnologia avançada de compilador otimizador. Como o nome do processador indica, no MIPS não há hardware para bloquear os estágios do pipeline devido a conflitos entre suas dependências. Este gerenciamento é feito em software em tempo de compilação (/HEN 82/).

A arquitetura do MIPS é do tipo carga/armazenamento, com endereçamento somente à palavra (32 bits). A escolha por endereçamento à palavra deveu-se a sua predominância (em relação à referência a byte) em programas de LAN e pela simplificação na implementação. Para tratamento de bytes existem instruções para manipular ponteiros de bytes e para extrair/inserir bytes específicos em palavras.

Existem 16 registradores de propósito geral de 32 bits. A ULA suporta instruções para obter dois bits da seqüência de multiplicação de Booth e um bit da seqüência de divisão. Não há suporte para operações de ponto flutuante. Todas as instruções aritmético-lógicas são entre registradores, com formatos de dois ou três operandos, podendo um deles ser substituído por uma pequena constante.

Em relação às operações de desvio não há códigos de condição, mas sim a instrução de compara-e-salta [PC←dst+PC se Cond(src1,src2)]. A operação é realizada em um único ciclo pois, no pipeline do MIPS, duas operações na ULA podem ser realizadas em uma instrução (no caso, dst+PC para cálculo de endereço e src1-src2 para cálculo da condição). A não utilização de códigos de condição simplifica o projeto, pois não há necessidade de decodificar quais instruções afetam (e de que maneira) os códigos de condição, nem de transmitir ao longo do processador sinais localizados em pontos diferentes para o registrador de condição, tampouco de salvar a condição quando da troca de contexto durante a execução. Também o compilador fica simplificado, pois não tem de gerar código com o fim específico de setar os bits de condição necessários a uma posterior instrução de salto condicional. As instruções de salto têm atraso de uma instrução, exceto a de salto indireto, onde o atraso é de duas instruções pois se exige um acesso à memória para buscar o endereço de desvio.

Um pipeline de 5 estágios, com duas fases de relógio por estágio, é implementado no MIPS: busca de instrução (IF), decodificação de instrução (ID), decodificação de operando (OD), armazenamento de operando e execução (SX), e busca de operando (OF). Tal configuração permite 3 instruções ativas no pipeline e duas operações na ULA (em OD e SX) durante uma instrução. Com o uso dos dois barramentos e de uma memória cache interna para instruções, o MIPS consegue, durante um ciclo de instrução, buscar a próxima instrução (endereço no início de IF, acesso na primeira metade de ID) e realizar uma carga de operando da memória (endereço ao fim de OD, acesso ao fim de OF).

O conjunto de instruções do MIPS é dividido em dois níveis. O primeiro nível é visível ao compilador e ao programador de linguagem de montagem e apresenta um conjunto de 31 instruções simples, tipo RISC. Neste nível, o código apresenta execução seqüencial das instruções, sem atrasos no salto ou nas instruções de acesso à memória, sem dependência de dados entre instruções e com apenas uma operação na ULA por instrução.

O segundo nível, instruções de linguagem de máquina, é fortemente dependente do pipeline da máquina e é gerado por um reorganizador, cuja entrada é o código de montagem gerado pelo compilador. Neste nível, o reorganizador modifica este código para acomodar as restrições do pipeline, eliminando dependências entre dados, além de otimizar a utilização dos registradores. Também otimiza o código nas instruções de salto e de carga/armazenamento retardadas, eliminando quando possível instruções de *NOOP* desnecessárias após aquelas. Caso seja possível, o reorganizador junta instruções de montagem, pois nem todas necessitam de 32 bits para codificação, em uma única instrução de máquina onde duas operações na ULA possam ser realizadas.

A realização em software do gerenciamento das dependências do pipeline apresenta três vantagens. Primeiro, nenhuma instrução é atrasada no pipeline devido a conflito entre registradores. Segundo, o ciclo básico do processador pode ser diminuído pois não há hardware para detectar o conflito entre dados. Finalmente, a ausência deste hardware permite utilizar esta área adicional para outros fins.

O MIPS foi implementado em 1983, usando tecnologia NMOS de 2μ com uma camada de metal, contendo 24000 transistores e atingindo 2 MIPS com relógio de 4 MHz. Programas de benchmark mostraram um desempenho de 5 a 6 vezes mais rápido que o Motorola 68000. Grande parte deste desempenho é devido à estrutura simplificada do processador e às otimizações realizadas pelo reorganizador de código, sendo estimado um aumento da ordem de 2 devido a estes fatores.

2.3.3 Controvérsia CISC x RISC

Apesar das máquinas RISC já terem chegado com sucesso ao mercado, existem controvérsias a respeito da real validade deste tipo de arquitetura, em especial quando da comparação com as máquinas CISC. Neste item discutem-se algumas destas críticas (/COL 85/,/COL 85a/,/PAT 85a/).

As máquinas RISC, por terem instruções mais simples, necessitam mais instruções do que máquinas CISC para executar uma mesma tarefa. Isto resulta em um aumento do código objeto e em maior tempo de execução (pois existem mais instruções). Este aumento de código, no entanto, não é significativo. Em máquinas RISC o tamanho estático de programas objeto é de 40% a 50% maior do que em CISC, enquanto que tamanho dinâmico (número de bytes de instruções buscados na memória) é de 10% a 30% maior (/HEN 84/). O pequeno aumento no código é devido ao fato de que um pequeno número de instruções simples é responsável por aproximadamente 90% das instruções executadas, e estas instruções estão nas máquinas RISC. A

premissa de que o tempo de execução será maior não é válida, pois a não inclusão das instruções infreqüentes permitiu que as instruções restantes fossem executadas muito mais rapidamente.

Outra crítica diz respeito ao fato de que a programação manual em linguagem de montagem em máquinas RISC é mais difícil, pois estas máquinas expõem no conjunto de instruções o pipeline de execução, além de exigirem mais instruções para realizar uma mesma tarefa em máquinas CISC.

Esta afirmação é correta, mas deve ser considerado o fato de que arquiteturas RISC são especialmente voltadas para a execução de LAN, possuindo compiladores que otimizam o código e tratam das eventuais dependências do pipeline. Programas objeto, gerados por máquinas RISC, demonstraram ser tão otimizados quanto a codificação manual de instruções em linguagem de montagem. Assim sendo, a dificuldade maior em se programar em linguagem de montagem somente surge quando a LAN não permite a completa expressão da tarefa que se deseja ou quando a interação com a implementação é essencial.

É de importância também que uma das maiores dificuldades em máquinas RISC é a realização de operações aritméticas em ponto flutuante. A inclusão nestas máquinas de uma unidade funcional para efetuar operações em ponto flutuante traz dificuldades maiores de implementação do que nas arquiteturas CISC. Cálculos em ponto flutuante geralmente são executados em múltiplos ciclos, tendo uma implementação seqüencial para reduzir a área da unidade funcional. A natureza multiciclo destas operações não se compatibiliza com duas características RISC: ciclo único e lógica de controle sem microcódigo.

As operações multiciclo ocasionarão problemas no pipeline, pois este deixará de ter um número fixo de estágios e/ou os estágios terão variações muito grandes no

tempo de duração, implicando que o controle do processador agora tenha mais uma tarefa - a de administrar estas características do pipeline. A operação com ciclos variados também provoca dificuldades maiores no projeto do controle em lógica fixa, o que não ocorre quando os múltiplos ciclos são acomodados em uma parte de controle microprogramada.

Mesmo resolvidos estes problemas, surge a questão da colocação da unidade funcional para ponto flutuante. Caso esta esteja ligada diretamente aos barramentos da parte operativa, o aumento da carga capacitiva e do tamanho da parte operativa irá causar uma degradação na velocidade de operação, afetando a execução de todas as instruções do processador. Outra opção é colocar a unidade de ponto flutuante externa ao processador, na forma de um coprocessador.

O uso de um coprocessador aritmético externo operando em conjunto com uma máquina RISC acarreta outro problema de incompatibilidade: a arquitetura de carga/armazenamento, com os dados mantidos internamente em um grande número de registradores. Deste modo, uma operação entre dois dados de 128 bits em ponto flutuante implica mover para a memória o conteúdo de diversos registradores internos e, após a operação, retornar da memória o resultado. Claramente se percebe que haverá um grande dispêndio de tempo apenas em comunicação com a memória. A utilização de um barramento externo extra para agilizar esta transferência também não é uma solução ótima, pois este recurso poderia ser melhor utilizado para, por exemplo, comunicação com memórias cache externas separadas para dados e instruções ou, ainda, para se fazer pré-busca de várias instruções para a execução de saltos condicionais.

Finalmente, a solução mais apropriada, permitida atualmente pela tecnologia, é integrar no mesmo CI o processador e o coprocessador. Deste modo, fica eliminada a necessidade de comunicação via memória externa, podendo-se

realizar a transferência de dados mais rapidamente via barramento interno especifico. Com as duas máquinas operando separadamente e comunicando-se de acordo com um protocolo particular, não há os problemas de múltiplos ciclos nem degradação de desempenho das instruções simples do processador RISC (/DOB 92/,/GRO 85/).

Há, também, críticas quanto ao desempenho de máquinas RISC, causado principalmente pelo grande número de registradores internos que estas máquinas possuem.

Um contra-argumento é que, justamente por serem máquinas RISC, este grande número de registradores pôde ser implementado, pois há uma drástica redução da área da parte de controle. Também por este mesmo motivo, o compilador pode otimizar mais facilmente o uso destes registradores, diminuindo acessos à memória e eliminando cálculos desnecessários.

Outro aspecto criticado é que as exigências da banda de passagem da memória em máquinas RISC são muito maiores. Para se verificar isso, primeiro deve-se analisar as exigências para acesso de dados e acesso de instruções. A banda de passagem para dados em máquinas com arquitetura de carga/armazenamento, como é o caso das máquinas RISC, é significativamente menor. Isto se deve ao grande banco de registradores e/ou à alocação dos registradores feita pelo compilador, permitindo menos acessos à memória para buscas de dados.

Para o acesso a instruções, a banda de passagem necessária é realmente maior em máquinas RISC, pois são necessárias mais instruções para executar uma mesma tarefa do que em CISC. No entanto, deve-se considerar que o uso de uma memória cache interna para instruções é uma boa solução para diminuir a banda de passagem externa ao chip, pois a cache de instruções tem altas taxas de sucesso devido à grande localidade no código. Além disto, o controle desta cache é simplificado pois ela se comporta como uma

ROM (o código não é alterado). Outro fator importante quanto ao acesso de instruções em RISC, comparativamente a CISC, é o número de instruções executadas em relação às instruções buscadas na memória. Por exemplo, no VAX, 25% das instruções são saltos executados. Isto implica que 25% das instruções buscadas imediatamente após o salto são desconsideradas e, portanto, há um aumento desnecessário na banda de memória. Em máquinas RISC, com o uso de salto retardado e código otimizado pelo compilador, grande parte destas instruções seguintes ao salto não são desperdiçadas. No MIPS, 21% das instruções executadas ocorrem durante o atraso do salto.

Para concluir, deve-se ter em mente que a escolha entre RISC e CISC não pode ser absoluta, mas sim que estas duas arquiteturas são os extremos entre as opções de projeto para se construir um processador, devendo-se adotar aqueles atributos em cada arquitetura necessários para conseguir a melhor implementação.

2.3.4 Perspectivas RISC

Processadores RISC, devido ao alto desempenho e à área em silício consideravelmente menor que em máquinas CISC, abriram algumas fronteiras promissoras no projeto de sistemas. Discute-se neste item algumas destas possibilidades.

Em processadores RISC sem o uso de janelas de registradores, a contagem de transistores é da ordem de 20 a 30 mil e, portanto, a área em silício é bastante reduzida em relação às máquinas CISC. Esta característica permite que se possa implementar uma arquitetura RISC em tecnologias que não permitam ainda grande escala de integração, como o Arsenieto de Gálio ou bipolar ECL, obtendo-se a vantagem de executar operações a taxas 5 ou 6 vezes mais rápidas do que em CMOS (/AGR 88/,/KAR 89/).

Pela mesma razão, a de pequeno número relativo de transistores, máquinas RISC podem ser implementadas utilizando-se uma metodologia de standard-cell em tecnologias convencionais. Apesar da provável degradação em desempenho, este método permite que uma arquitetura possa ser projetada e testada funcionalmente em poucos meses, possibilitando a rápida entrada no mercado de um novo produto enquanto aperfeiçoamentos são realizados nesta arquitetura para implementação final em full-custom (/NAM 88/, /NAM 88a/).

A pequena área de máquinas RISC, aliado ao seu alto desempenho, permite que vários processadores sejam integrados em uma mesma pastilha de silício ou, então, que um mesmo processador possua diversas unidades funcionais repetidas. Este fato permitiu o surgimento de arquiteturas VLIW (*Very Long Instruction Word*), onde uma instrução da ordem de centenas de bits especifica diversas operações em paralelo; de arquiteturas Super-Escalares, onde diversas instruções são buscadas e executadas em paralelo; e da integração em um mesmo chip de vários processadores operando paralelamente (MIMD). O núcleo destas máquinas está baseado em RISC, seja pela menor área, seja pela simplicidade de projeto.

O alto desempenho do processador tipo RISC permite que este emule em software um outro processador, sem degradação insuportável do desempenho da máquina emulada. Por exemplo, o processador RISC do IBM *System/6000* pode emular o software do MS-DOS tão rapidamente quanto um IBM-PC com processador Intel 80286 /HUN 89/. Esta possibilidade permite que a base de software já instalado continue a ser utilizada.

Finalmente, o uso de RISC em sistemas integrados (*embeded systems*), como em aviônica e impressoras lasers, tem sido promissor. Em aplicações embarcadas as necessidades são diferentes das que são encontradas em outros ambientes, como as estações de trabalho, por

exemplo. Não é preciso grandes bancos de registradores, memórias cache ou coprocessadores, e muitas vezes máquinas de 16 bits são suficientes. Deste modo, as características de desempenho podem ser ainda maiores quando se elimina do processador aqueles atributos não requeridos pela aplicação. Por outro lado, os atributos de rápido chaveamento de contexto e pequena latência na resposta de interrupção são, em geral, bastante necessários nestes sistemas.

O tempo de chaveamento está principalmente ligado ao tempo de salvamento e recuperação de registradores. Como o número de registradores pode ser otimizado para a aplicação (em geral um pequeno número é suficiente), a troca de contexto pode ser otimizada.

Já a escolha de RISC devido ao tempo de latência curto para atender interrupções é decorrência natural da própria arquitetura. Sendo as instruções executadas em ciclo único e a interrupção atendida somente entre as instruções (e não no meio de uma instrução), o tempo de resposta é sempre determinável à priori, com duração igual e curta. Estas características facilitam a implementação dos sistemas de tempo real.

2.4 Arquiteturas VLSI

Apesar dos avanços tecnológicos, a implementação da arquitetura de processadores integrados VLSI é limitada por diversos fatores, entre estes o número de dispositivos, a área total disponível, a potência dissipada e a velocidade da operação. Deste modo, uma arquitetura em projeto deve ter em vista as possibilidades de implementação na tecnologia escolhida ou disponível.

Em VLSI também é importante a densidade da tecnologia e o tipo de encapsulamento. A densidade é afetada pela quantidade de dispositivos nas portas lógicas e pelos níveis de interconexões disponíveis. O encapsulamento, por

sua vez, leva a limitações na potência dissipada e no número de pinos disponíveis.

Nos próximos itens, discute-se a influência da tecnologia na arquitetura e um método para tratar da complexidade da arquitetura.

2.4.1 Implicações tecnológicas no projeto de arquiteturas

A Arquitetura de um computador impõe exigências no nível organizacional desta arquitetura e este, por sua vez, as impõe sobre a tecnologia. Por exemplo, uma arquitetura possui uma instrução que exige duas operações aritméticas, forçando a organização interna a possuir duas ULAs ou a serializar no tempo as operações com uma única ULA. Por sua vez, a implementação da(s) ULA(s) poderá exigir alto grau de paralelismo na operação, caso a tecnologia não permita alto desempenho. É possível até que a arquitetura não seja realizável, caso o alto grau de paralelismo necessário exija uma área maior do que é possível integrar na tecnologia disponível.

As implementações paralelas são, geralmente, utilizadas para superar a limitação de velocidade da tecnologia. Muitos componentes lentos operando em paralelo são preferencialmente utilizados em lugar de poucos componentes rápidos, desde que pesados os custos para gerenciar estes componentes operando paralelamente.

Já a complexidade dos circuitos que podem ser implementados na tecnologia disponível limita o que pode ser feito pela arquitetura. Por exemplo, não é possível implementar em processador genérico uma instrução de multiplicação de dois números em ponto flutuante, em um ciclo único e em um tempo aceitável.

A comunicação tem maiores custos do que a computação em si. Deste modo, uma arquitetura que exige maior comunicação global apresentará maiores problemas na

implementação. Em VLSI, os custos para comunicação estão no maior consumo de potência para diminuir o tempo de retardo da propagação de um sinal, no gasto em área e na limitação das ligações devido ao número reduzido de níveis de interconexões disponíveis (usualmente 2 ou 3 níveis).

Por fim, em VLSI as fronteiras do chip são uma grande influência no projeto, impondo um limite físico na banda de passagem do chip com o exterior e criando uma disparidade substancial nos atrasos de comunicação dentro e fora do chip.

2.4.2 Modelo P.C./PO

Processadores são máquinas programáveis e, portanto, podem ser vistos como um circuito seqüencial de alta complexidade, onde complexidade se entende por uma grande quantidade de estados e uma função próximo estado não elementar. Para projetar tal sistema, de alta complexidade, é conveniente a utilização do modelo P.C./PO (/SUZ 85/,/TOD 86/).

Neste modelo, o sistema digital é dividido em dois subsistemas: o operacional (PO - parte operativa) e o de comando (P.C. - parte de controle). A PO contém as ações a serem realizadas e a P.C. realiza o seqüenciamento destas ações, determinando quando as operações devem ser executadas.

Este modelo se aplica bem ao projeto de processadores. A PO contém os elementos necessários para operar sobre os dados indicados pela instrução, sendo geralmente do tamanho da palavra do processador e composta de registradores de dados, unidades funcionais e barramentos. A PO recebe sinais de comando da P.C. para realizar ações e devolve informações a esta sobre o estado das operações em curso ou realizadas.

A P.C. contém os elementos necessários para identificar a instrução e ativar na PO aqueles componentes exigidos para realizar a operação, além de tratar do seqüenciamento das instruções e da interação com o exterior do processador.

Esta divisão facilita o projeto, na medida que o sistema se reduz a dois grandes blocos. A PO é realizada segundo uma estrutura *bit-slice* distribuída ao longo de barramentos. O projeto, na maioria das vezes, se reduz a um bit lógico projetado sobre uma estrutura de barramentos previamente escolhida, bastando replicar a estrutura no número de bits necessário, já estando os problemas de interconexões resolvidos antecipadamente.

A divisão também facilita o projeto da P.C., podendo em sua forma mais simples ser reduzido a um circuito composto de um registrador e um circuito combinacional do tipo PLA ou ROM, compondo uma máquina seqüencial do tipo Moore ou Mealy.

Deve ser observado que a divisão do sistema digital em P.C./PO é, na realidade, a divisão do autômato em outros dois - um voltado a realizar as operações e outro a seqüenciar o funcionamento. Como cada subsistema resultante é ainda um autômato, o processo de divisão pode continuar, podendo-se, por exemplo, dividir a P.C. em dois subsistemas do tipo P.C./PO caso a complexidade seja ainda muito grande.

3 ARQUITETURA DO MICROPROCESSADOR RISCO

3.1 Introdução

Neste capítulo, diversos aspectos do Risco são apresentados. Mostra-se o conjunto de instruções, seu formato, a interface com o mundo exterior e o tratamento de exceções.

3.2 Descrição Geral

O Risco é um microprocessador CMOS de 32 bits com arquitetura tipo RISC. Suas principais características são:

a) Dados, instruções e endereços são palavras de 32 bits.

b) A unidade de endereçamento é a palavra, permitindo um acesso a 4 Giga palavras (16 Gbytes). Não existe endereçamento a byte ou meia-palavra (2 bytes).

c) A comunicação com a memória é feita por um barramento único de 32 bits, multiplexado de dados e endereços.

d) Possui 32 registradores de 32 bits, incluídos nestes o PC (contador de programa = R31), SP (apontador de pilha = Rxx, definido pelo compilador), PSW (palavra de status do processador = R01) e R0 (constante 0 = R00).

e) Possui um pipeline de instruções de 3 estágios, atingindo o pico de uma instrução executada por ciclo de máquina.

f) As instruções de salto têm sua execução retardada de uma instrução.

Um diagrama da PO (Parte Operativa) encontra-se na figura 3.1.

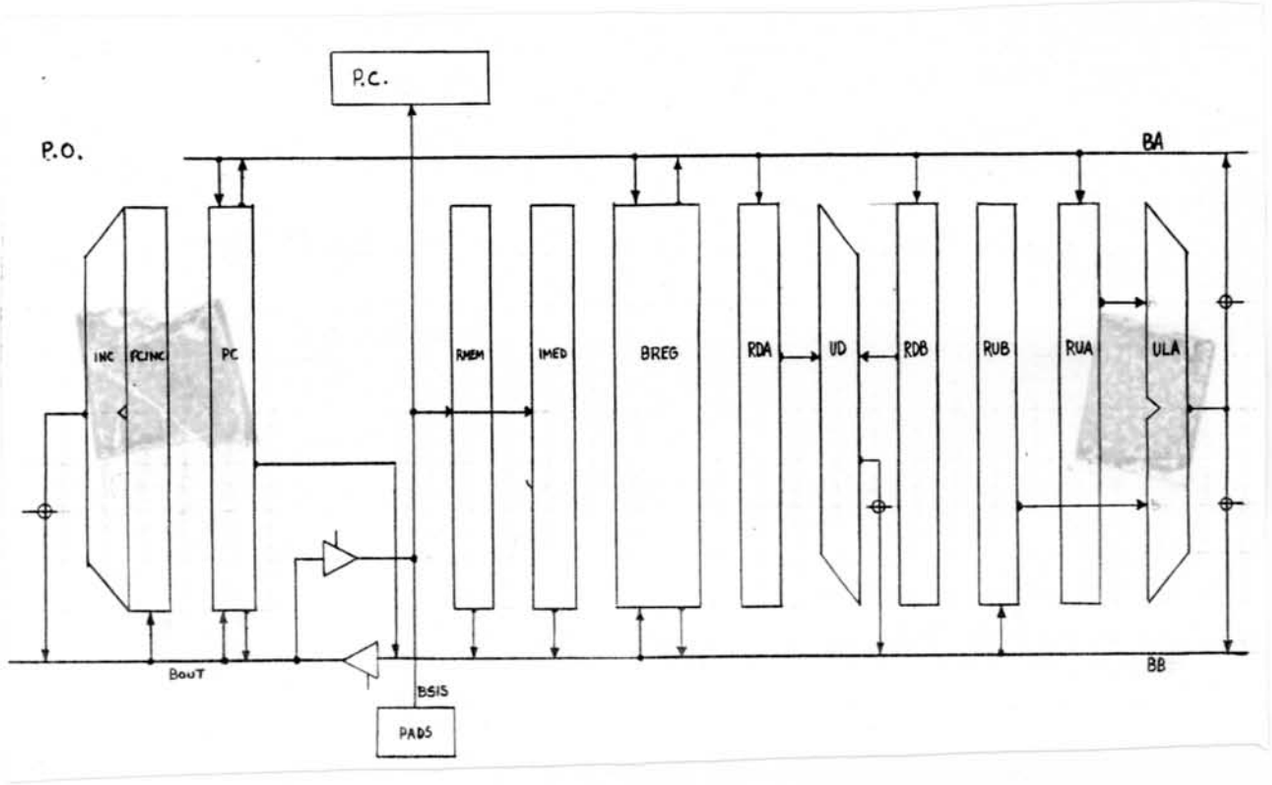


Figura 3.1 Diagrama de Blocos da PO

3.3 Formatos de Instruções

O Risco é uma máquina de três endereços (de operandos), usualmente com um operando destino e dois operandos fontes (os operandos são sempre registradores da UCP ou uma constante presente na palavra de instrução). Existem também instruções com um destino, uma fonte e uma constante de 11 bits, bem como com um destino/fonte e uma constante de 17 bits. Não há palavra com valor imediato após as instruções.

Os três formatos possíveis de instrução estão esquematizados na figura 3.2.

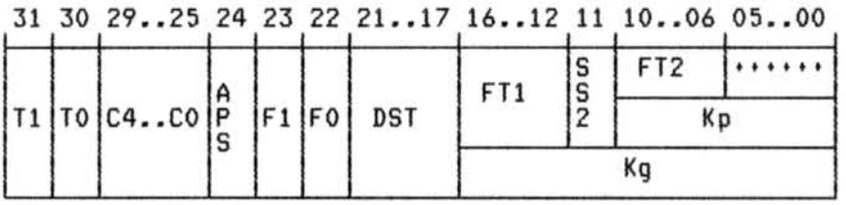


Figura 3.2 Formatos de Instrução

Os campos da palavra de instrução e respectivas funções são descritos a seguir:

a) T1, T0: definem o tipo de instrução a ser executada: aritmético-lógica, salto, acesso à memória ou sub-rotina. T1 indica se a instrução terá um ciclo extra para acesso à memória (instruções de carga/armazenamento ou de sub-rotina) e T0 define se o campo C4..C0 contém um código de operação (aritmético-lógica ou de carga/armazenamento) ou o código de teste (salto ou sub-rotina).

b) C4, C3, C2, C1, C0: determinam a operação a ser executada nas instruções aritmético-lógicas e de acesso à memória, ou a condição de teste nas instruções de salto e de sub-rotina.

c) APS: indica se a PSW (palavra de status do processador) deve ser atualizada ao fim da instrução corrente, ou seja, se APS=1 então os bits N (negativo), Ov (overflow), Z (zero) e C (carry) da PSW serão atualizados de acordo com o resultado da operação.

d) F1, F0, Kg: os bits F1 e F0 indicam se os bits de 16 a 0 contêm uma constante (Kg) de 17 bits como segundo operando da operação, ou o campo FT1/SS2/[FT2 ou Kp]; F1, F0 e SS2 indicam também os operandos da instrução, conforme apresentado na tabela 3.1.

e) DST, FT1: são os endereços de 2 dos 32 registradores para destino (DST) e primeiro operando (FT1) da operação.

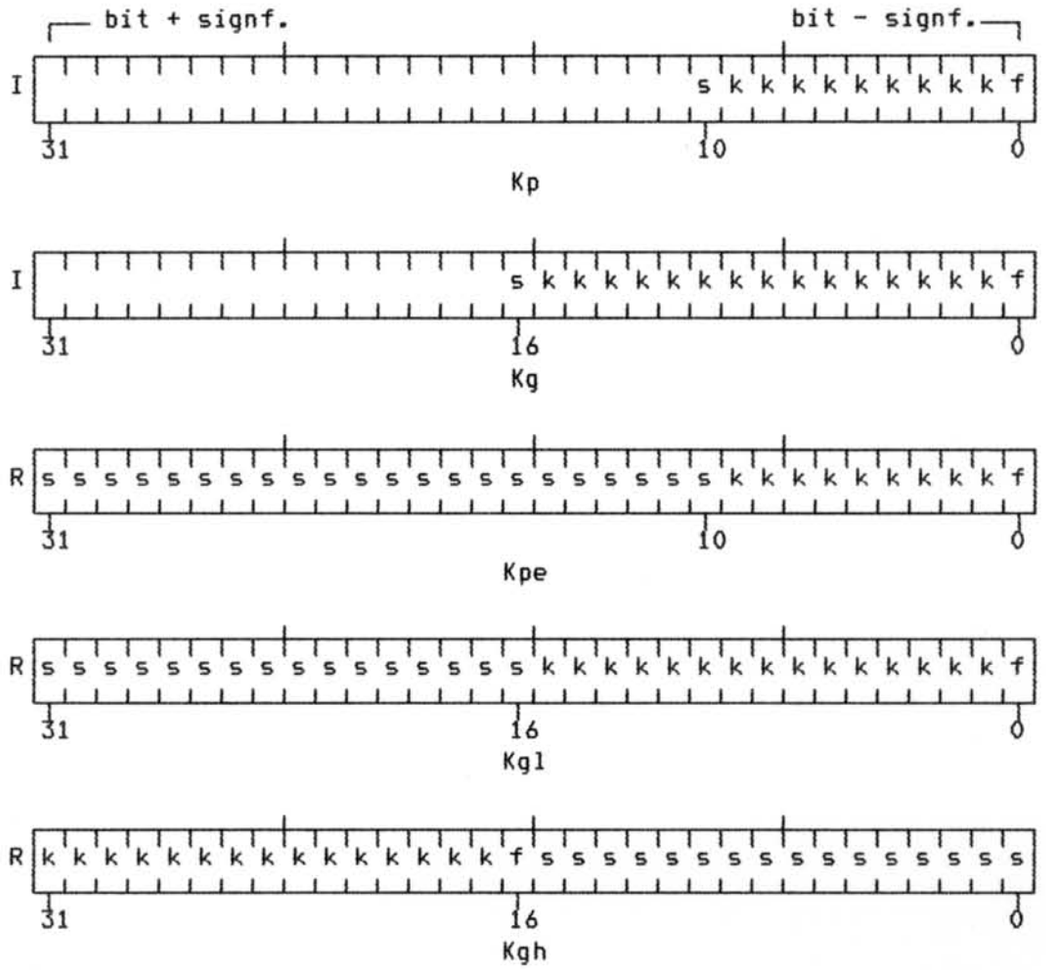
f) SS2, FT2, Kp: SS2 indica se os bits de 10 a 0 significam uma constante (Kp) de 11 bits, ou se os bits de 10 a 6 significam o endereço de 1 dos 32 registradores (FT2) para segundo operando da operação.

Tabela 3.1 Formatos de Instrução e Operandos

F1	F0	SS2	Formato	Operação (dest <-- fa op fb)
0	0	0	DST/FT1/FT2	DST ← FT1 op FT2
0	0	1	DST/FT1/Kp	DST ← FT1 op Kpe
0	1	x	DST/Kg	DST ← R0 op Kgl
1	0	x	DST/Kg	DST ← DST op Kgh
1	1	x	DST/Kg	DST ← DST op Kgl

g) Constantes: de acordo com F1 e F0, a constante presente na palavra de instrução é carregada em um registrador interno temporário para ser o segundo operando, assumindo a forma descrita na figura 3.3. A constante Kp, de 11 bits, tem seu bit mais significativo (nº10) estendido, permitindo constantes de valor entre +1023 (=+1K-1) a -1024 (=-1K).

A constante Kg, de 17 bits, é tratada de duas maneiras distintas. O primeiro modo é semelhante ao de Kp, ou seja, o bit mais significativo (nº16) é estendido, permitindo constantes de +65535 (=+64K-1) a -65536 (=-64K). No segundo modo, os 16 bits menos significativos da instrução são carregados nos 16 bits mais significativos do registrador interno temporário, e o 17º bit (nº16) é estendido para os 16 bits menos significativos do registrador interno.



Onde:
I - palavra de instrução, R - registrador interno da const. estendida,
Kp - constante de 11 bits em I, Kg - constante de 17 bits em I,
Kpe - constante Kp estendida em R, Kgl - constante Kg estendida em R,
Kgh - constante Kg estendida e rotacionada em R,
f - bit menos significativo (bit 0), k - demais bits da constante.
s - bit mais significativo (bit 10 em Kp, bit 16 em Kg),

Figura 3.3 Carregamento da Constante no Registrador Interno

O formato das instruções do Risco procurou separar a codificação da operação a ser realizada da codificação dos operandos a serem utilizados, permitindo assim facilitar a interpretação das instruções. De uma maneira simplificada, pode-se analisar uma instrução no Risco da seguinte forma: dois bits indicam o tipo de instrução e outros cinco determinam as variações dentro deste tipo, um bit indica se a PSW deverá ser alterada ao fim da instrução, três bits indicam o formato da instrução

e os operandos e os bits restantes indicam os endereços dos operandos (registradores ou constante, se houver).

A separação dos campos facilita a implementação da seguinte maneira: o campo contendo a operação é utilizado diretamente pela parte de controle para determinar a tarefa a ser executada, sem que seja necessário determinar os operandos. Assim, uma operação aritmético-lógica é gerada em termos de A ⟵ B op C, cabendo à parte de controle determinar somente a operação op e seqüenciar a leitura de B e C e a escrita de A. Para a parte de controle não importa o endereço de A, B e C, mas sim a geração no tempo correto da leitura ou escrita destes.

Por outro lado, o campo contendo a codificação dos operandos é utilizado somente pela parte operativa. Este campo contém o endereço dos operandos (registradores) ou eventualmente um dos operandos (constante). Esta informação vai diretamente para os decodificadores do banco de registradores ou para a unidade funcional que realiza a extensão da constante. Como no Risco existem três formatos de instrução (note-se que a variação está somente no campo dos operandos) e estes formatos são definidos por três bits apenas (F1, F0 e SS2), uma pequena lógica separada é utilizada sobre estes bits para determinar quais são os operandos (DST, FT1, FT2, Kp, e Kg) e direcionar o campo destes para o lugar apropriado na parte operativa (decodificadores ou unidade da constante).

No Risco, os bits de 31 a 24 (total de 8) da instrução vão direto para a parte de controle; os bits 23, 22 e 11 vão para uma lógica separada para determinar os operandos da instrução; e os bits de 21 a 0 (total de 22) vão direto para a parte operativa. A parte de controle determina uma operação do tipo A ⟵ B op C, gerando no tempo sinais do tipo "lê B","lê C" e "escreve A". A parte operativa recebe estes sinais e busca no endereço correto as variáveis A, B e C.

3.4 Conjunto de Instruções e Modos de Endereçamento

As instruções no Risco foram planejadas segundo as características das máquinas RISC, principalmente a busca da execução em ciclo único e da execução eficiente de LAN. Buscou-se, também, a maior ortogonalidade e simetria possível no conjunto de instruções. Entende-se por ortogonalidade todas as instruções utilizarem todos os tipos de dados e, por simetria, oferecer todos os modos de endereçamento de operandos em todas as instruções. Estas características auxiliam no projeto, pois pode-se separar, durante a interpretação da instrução, as tarefas de execução da operação da de cálculo dos endereços de operandos e do seu tipo.

Uma característica da arquitetura do Risco é a presença de uma constante de valor 0 endereçável como registrador R00 (pode-se escrever no registrador R00, mas seu valor não se altera). Com o uso de R00 e da constante na palavra de instrução, pode-se sintetizar outras operações não explicitadas no seu conjunto de instruções. Algumas operações são listadas a seguir:

```
MOVE:          DST ⟵ FT1+R00
INCREMENTA:    DST ⟵ FT1+Kp; Kp=+1
DECREMENTA:    DST ⟵ FT1+Kp; Kp=-1
COMPLEMENTA:   DST ⟵ R00-FT2
NEGAÇÃO:       DST ⟵ FT1 xor Kp; Kp=-1
CLEAR:         DST ⟵ R00+R00
COMPARA:       R00 ⟵ FT1-FT2; APS=1
NOOP:          R00 ⟵ R00+R00; APS=0
```

A seguir, são descritas as operações para os quatro tipos de instrução: aritmético-lógica, salto, acesso à memória (carga/armazenamento) e sub-rotina, conforme determinado pelos bits T1 e T0 da palavra de instrução. Existem implementadas 25 instruções aritmético-lógicas, 8 de carga/armazenamento, 1 de salto e 1 de sub-rotina (sob 16 condições cada uma), totalizando 35 instruções. No item 3.4.2 são apresentadas as possibilidades de endereçamento para as instruções de salto e de carga/armazenamento.

3.4.1 Instruções aritmético-lógicas

Nas instruções aritmético-lógicas (T1=T0=0), o Risco implementa 25 instruções, listadas na tabela 3.2, do total das 32 possíveis definidas por C4..C0.

Tabela 3.2 Instruções Aritmético-Lógicas

Instrução	Operação	Comentário
ADD	dest ⟵ fa + fb	Soma
ADDC	dest ⟵ fa + fb + C	Soma c/ C
SUB	dest ⟵ fa - fb	Subtração
SUBC	dest ⟵ fa - fb - C	Subtração c/ C
SUBR	dest ⟵ fb - fa	Subtração Reversa
SUBRC	dest ⟵ fb - fa - C	Subtração Reversa c/ C
AND	dest ⟵ fa & fb	E Lógico
OR	dest ⟵ fa ¦ fb	OU Lógico
XOR	dest ⟵ fa ^ fb	OU EXCLUSIVO Lógico
RRL	dest ⟵ fa rot fb	Rot. Dir. Lóg.
RRLC	dest ⟵ fa rot fb c/ C	Rot. Dir. Lóg. c/ C
RRA	dest ⟵ fa rot fb	Rot. Dir. Arit.
RRAC	dest ⟵ fa rot fb c/ C	Rot. Dir. Arit. c/ C
RLL	dest ⟵ fa rot fb	Rot. Esq. Lóg.
RLLC	dest ⟵ fa rot fb c/ C	Rot. Esq. Lóg. c/ C
RLA	dest ⟵ fa rot fb	Rot. Esq. Arit.
RLAC	dest ⟵ fa rot fb c/ C	Rot. Esq. Arit. c/ C
SRL	dest ⟵ fa dlc fb	Desl. Dir. Lóg.
SRLC	dest ⟵ fa dlc fb c/ C	Desl. Dir. Lóg. c/ C
SRA	dest ⟵ fa dlc fb	Desl. Dir. Arit.
SRAC	dest ⟵ fa dlc fb c/ C	Desl. Dir. Arit. c/ C
SLL	dest ⟵ fa dlc fb	Desl. Esq. Lóg.
SLLC	dest ⟵ fa dlc fb c/ C	Desl. Esq. Lóg. c/ C
SLA	dest ⟵ fa dlc fb	Desl. Esq. Arit.
SLAC	dest ⟵ fa dlc fb c/ C	Desl. Esq. Arit. c/ C

dest,fa,fb: Operadores referenciados na Tabela 3.1,
c/ C: com Carry, Rot.: Rotação, Desl.: Deslocamento,
Dir.: Direita, Esq.: Esquerda,
Lóg.: Lógico, Arit.: Aritmético.

Algumas observações sobre as instruções aritmético-lógicas são feitas a seguir:

a) A operação SUB (fa-fb) é realizada internamente como fa+NOT(fb)+1.

b) A operação SUBR é necessária para permitir a subtração de uma constante pelo valor do conteúdo de um registrador (const-Rxx).

c) Arquiteturas RISC são usualmente projetadas para execução de Linguagens de Alto Nível (LAN), nas quais as operações de deslocamento e rotação não são freqüentes (apenas deslocamentos de 1 ou 2 bits são suficientes). Mesmo assim, no Risco foi implementado um conjunto relativamente extenso destas instruções, visando permitir um tratamento mais eficiente de caracteres (bytes), pois o acesso à memória no Risco é sempre a palavra (4 bytes). Além disto, desejava-se facilitar a execução de aritmética de múltipla precisão, permitindo que o Risco opere sem muita degradação de desempenho na ausência de um coprocessador.

d) Nas instruções de deslocamento e rotação, as operações executadas seguem a seguinte norma geral:

- desl./rot. arit.: S (sinal=bit 31) e C (carry) permanecem inalterados; exceto em SRA, onde S é copiado para os bits menos significativos;

- desl./rot. lógico: C mantém-se inalterado, S entra na operação;

- desl./rot. lógico c/ C: S e C entram na operação (C é interpretado como se fosse o bit 32);

- desl./rot. aritmético c/ C: S fica inalterado, C entra na operação (C passa a ser o bit 31).

Para qualquer uma das instruções da tabela 3.2 é possível utilizar todos os formatos e operandos definidos na

tabela 3.1. Cabe ao compilador, e não ao hardware, decidir quais operações não têm sentido ou utilidade. Esta decisão de projeto visa diminuir a complexidade do autômato de controle, resultando em uma P.C. (Parte de Controle) mais simples e rápida, de acordo com os critérios da Arquitetura RISC. Por exemplo, a operação SLL: R23 ⟵ R00 dlc Kgl faz o registrador R23 receber o registrador R00 (contém zero constante) deslocado logicamente à esquerda de n bits, sendo n o valor contido nos 5 bits menos significativos (b4..b0) da constante Kgl. Uma vez que são inseridos zeros nos bits menos significativos do operando deslocado, R23 receberá zero. Tal operação não realiza um deslocamento de conteúdo de registrador, mas sim um "clear" (R23 ⟵ 0) no registrador.

A carga de um registrador com uma constante de 32 bits pode ser executada com duas instruções:

```
ADD:  Rxx ⟵ R00 +  Kgl  /* bit 16 de Kg é 0 */
OR:   Rxx ⟵ Rxx or Kgh  /* bit 16 de Kg é 0 */
```

Na primeira instrução, o registrador destino Rxx recebe uma constante nos seus 16 bits menos significativos. O bit 16 (17º bit) da constante, valor 0, é estendido para os bits mais significativos de Rxx. Na instrução seguinte, Rxx recebe um OU lógico entre ele mesmo e outra constante. Como a nova constante foi carregada nos dois bytes mais significativos (e o bit 16, valor 0, estendido para os bits menos significativos), Rxx é carregado com uma constante de 32 bits.

Este método para carga de um valor de 32 bits em registrador é utilizado no Risco por apresentar vantagens sobre a utilização de um valor imediato após a instrução, pois este último implica mais estados no autômato de controle (decisão entre busca de instrução ou busca de imediato). Em ambos os casos, o número de acessos à memória, a quantidade de bytes utilizados em memória e o tempo de execução é o mesmo.

3.4.2 Instruções de acesso à memória (carga/armazenamento)

São implementadas as operações de carga e armazenamento de registrador na memória, com possibilidade de haver pré-incremento, pós-incremento e pós-decremento do segundo operando quando este é um registrador. As operações estão listadas na tabela 3.3 .

Tabela 3.3 Instruções de Acesso à Memória

Inst.	Operação	Comentário
LD	dest ⟵ M[fa+fb]	Carga
LDPRI	FT2 ⟵ FT2+1; DST ⟵ M[FT1+FT2]	Carga c/ pré-inc.
LDPOI	DST ⟵ M[FT1+FT2]; FT2 ⟵ FT2+1	Carga c/ pós-inc.
LDPOD	DST ⟵ M[FT1+FT2]; FT2 ⟵ FT2-1	Carga c/ pós-dec.
ST	M[fa+fb] ⟵ dest	Armazenamento
STPRI	FT2 ⟵ FT2+1; M[FT1+FT2] ⟵ DST	Armaz. c/ pré-inc.
STPOI	M[FT1+FT2] ⟵ DST; FT2 ⟵ FT2+1	Armaz. c/ pós-inc.
STPOD	M[FT1+FT2] ⟵ DST; FT2 ⟵ FT2-1	Armaz. c/ pós-dec.

dest,fa,fb,DST,FT1,FT2: Operadores referenciados na Tabela 3.1,
M[ ]: Conteúdo de Memória, fa+fb,FT1+FT2: Endereço de Memória E,
Armaz.: Armazenamento, inc.: incremento, dec.: decremento.

O endereço E é obtido pela soma de dois operandos quaisquer (todos os modos listados na tabela 3.1), obtendo-se de modo prático as seguintes possibilidades de endereço:

```
E=FT1          /*  FT1+FT2,   FT2=R00   */
E=FT2          /*  FT1+FT2,   FT1=R00   */
E=FT1+FT2      /*  FT1+FT2              */
E=Kp           /*  FT1+Kp,    FT1=R00   */
E=FT1+Kp       /*  FT1+Kp               */
E=Kg           /*  R00+Kgl              */
```

3.4.3 Instruções de salto

As instruções de salto são implementadas como uma soma condicional:

Tabela 3.4 Instruções de Salto

Instrução	Operação	Comentário
JMP	(se COND verdadeira então dest ⟵ fa + fb)	salto

COND: Condição do teste,
dest,fa,fb: Operadores referenciados na Tabela 3.1,
fa+fb=E: Endereço de Salto E.

Para se obter o salto deve ser especificado dest=DST=PC(=R31) na palavra de instrução. COND é o teste (especificado no campo C4..C0 da instrução) a ser feito sobre a PSW(=R01). O endereço E é obtido do mesmo modo que nas instruções de carga/armazenamento, podendo, para o salto, assumir os seguintes endereços na prática (observe-se que, por exemplo, E=FT1+FT2 significa PC ⟵ FT1+FT2):

```
E=FT1          /*  FT1+FT2,  FT2=R00  */
E=FT2          /*  FT1+FT2,  FT1=R00  */
E=FT1+FT2      /*  FT1+FT2            */
E=Kp           /*  FT1+Kp,   FT1=R00  */
E=FT1+Kp       /*  FT1+Kp             */
E=Kg           /*  R00+Kg1            */
E=PC+FT2       /*  FT1+FT2,  FT1=R31  */
E=PC+Kp        /*  FT1+Kp,   FT1=R31  */
E=PC+Kg1       /*  DST+Kg1,  DST=R31  */
```

O Risco utiliza uma extensa possibilidade de condições de teste (16), visando diminuir operações intermediárias para a obtenção deste mesmo teste. São implementadas as condições sempre_verdadeiro (para salto incondicional), carry, overflow, negativo, zero (ou igual), menor_ou_igual para números sem sinal, menor e menor_ou_igual para números com sinal. Pelo complemento destas oito condições, tem-se respectivamente sempre_falso, sem_carry, sem_overflow, positivo, não_zero (ou diferente), maior para números sem sinal, maior_ou_igual e maior para números com sinal.

Para melhorar o desempenho do pipeline, o Risco utiliza a característica de salto retardado, ou seja, a efetivação do salto somente ocorre após a execução da instrução seguinte à de salto. Compete ao compilador inserir um NOOP (não opera: R00 ⟵ R00+R00) após o salto, ou trocar o salto com a instrução anterior se possível.

3.4.4 Instruções de sub-rotina

A operação encontra-se na tabela 3.5:

Tabela 3.5 Instruções de Sub-rotina

Instrução	Operação	Comentário
SR	{ se COND verdadeiro então (dest ⟵ dest - 1 ; M[dest] ⟵ PC ; PC ⟵ fa + fb) }	Sub-rotina

COND: Condição de Teste,
dest,fa,fb: Operadores referenciados na Tabela 3.1,
fa+fb=E: Endereço de Salto E,
M[ ]: Conteúdo de Memória,
dest=DST=Rxx: Endereço de Pilha,
PC=R31: Contador de Programa.

O registrador DST é qualquer um dos 32 registradores escolhidos pelo compilador para ser o SP. O endereço E é obtido como na instrução de salto, assim como a condição COND. Para o retorno da Sub-rotina, realiza-se uma instrução de carga de registrador, PC, da memória com pós-incremento (LDPOI):

```
LDPOI: ( PC ⟵ M[R00+SP] ;
         SP ⟵ SP+1 ;      )
```

3.5 Tratamento de Exceções

Só um nível de interrupção é permitido no Risco. Na verdade, buscou-se implementar um hardware capaz de ser simples e, ao mesmo tempo, de não inibir futuros desenvolvimentos.

A interrupção só é atendida ao final de uma instrução. O pipeline do Risco sendo de 3 estágios, o atendimento de uma interrupção provoca a necessidade de esvaziamento do mesmo. Para tal, as duas instruções mais antigas são mantidas, sendo cancelada a escrita no banco de registradores do resultado. A instrução mais recente (que estava sendo buscada quando da interrupção) é simplesmente descartada, sendo trocada internamente por um salto à posição fixa 400h.

Pelo estratagema acima, um usuário poderá impedir o acesso à posição 400h inibindo o sinal de RD na memória e colocando no barramento a sua instrução (normalmente um novo JMP para endereço onde está a sub-rotina de tratamento dos vários tipos de interrupção).

Pelo esquema acima proposto, pode-se colocar um processador de interrupções externo ao Risco, de modo que o fato de dispor de apenas uma via de interrupção não seja uma limitação ao uso do processador.

É importante ressaltar que, se uma instrução no pipeline é de acesso à memória (carga ou armazenamento), esta instrução não pode ser simplesmente descartada, pois a memória deve estar sempre consistente. Neste caso, a instrução pendente é concluída e o JMP para a posição 400h é atrasado de um ciclo de máquina, o que equivaleria a um NOOP. Assim, o INTACK da interrupção é obviamente atrasado.

3.6 Interface Externa

O Risco foi projetado tendo como uma das condições de contorno o menor custo possível para o encapsulamento. Deste modo, a arquitetura foi especificada tendo por base um encapsulamento simples de 48 pinos. Os pinos não utilizados servirão para aumentar a testabilidade e a observabilidade do protótipo. Também para facilitar os testes, as fases de relógio são geradas externamente. A tabela 3.6 lista os pinos necessários.

O protocolo de comunicação com a memória é o mais simples possível, para a primeira versão do Risco. Um diagrama do funcionamento está na figura 3.4.

Tabela 3.6 Pinos do Risco

Função	Nome	Quant.	Direção
Dados e endereços	AD00..AD31	32	ent/sai
Amostragem de endereço	ALE	1	sai
Leitura de dado de memória	RD	1	sai
Escrita de dado de memória	WR	1	sai
Pedido de interrupção	INT	1	ent
Atendimento de interrupção	INTACK	1	sai
Reset	RES	1	ent
Alimentação Pads	VDDP, GNDP	2	ent
Alimentação Circuito	VDD, GND	2	ent
Relógio	f1,f2,f3	3	ent
Total		= 45	

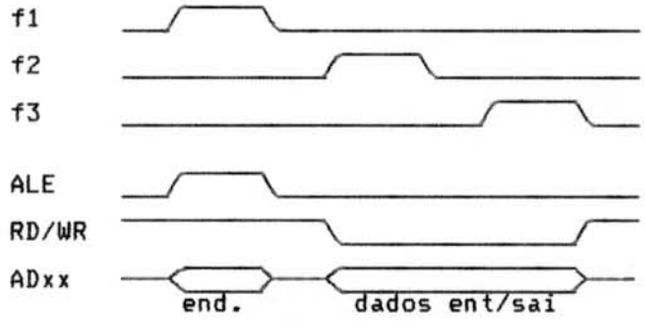


Figura 3.4 Protocolo externo para o ciclo de leitura/escrita

3.7 Pipeline de Instruções

O Risco possui um pipeline de instruções de 3 estágios: busca de instrução, operação e escrita do resultado. Cada estágio é dividido em 3 etapas por um relógio de 3 fases. O diagrama de funcionamento do pipeline está na figura 3.5 . Detalhes do pipeline serão descritos no capítulo quatro.

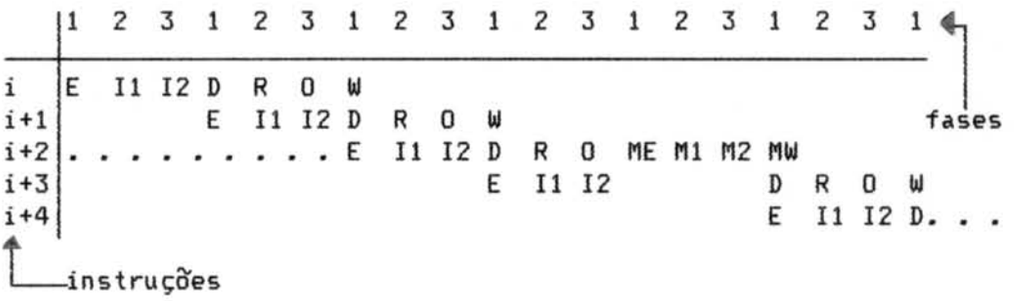


Figura 3.5 Diagrama do Pipeline de Instruções

Uma nova instrução é buscada a cada ciclo (a cada 3 fases), atingindo o processador a taxa de uma instrução por ciclo de máquina. Uma vez que só é possível um único acesso à memória ao mesmo tempo, as instruções de acesso à memória (carga/armazenamento e sub-rotina) ocupam mais um ciclo, suspendendo o pipeline (observe-se no diagrama a instrução i+2).

As etapas do pipeline são:

- E: Escrita do endereço de memória da próxima instrução.

- I1: Início da leitura da instrução da memória e cálculo do endereço da próxima instrução.

- I2: Conclusão da leitura da instrução.

- D: Decodificação.

- R: Leitura dos operandos, sendo estes dois registradores ou, então, um registrador e uma constante.

- O: Operação na unidade funcional, na ULA ou na UD.

- Em: Escrita do endereço de memória do dado a ser carregado ou armazenado.

- M1: Início da leitura ou escrita do dado de memória.

- M2: Conclusão da leitura ou escrita do dado de memória.

- W ou MW: Escrita do resultado da operação no destino, um registrador. Em W, o resultado é a saída da unidade funcional; em MW, o resultado é o dado carregado da memória.

4 DISCUSSÃO SOBRE O PIPELINE

Neste item, estudam-se as possibilidades para realizar um pipeline de instruções em um processador do tipo RISC e, a partir destas conclusões, justificar a escolha do pipeline do Risco. Várias decisões posteriores relativas à implementação do Risco foram feitas baseadas nos resultados expostos neste capítulo.

A fim de restringir a amplitude do estudo, assume-se que: (a) a arquitetura do processador é do tipo carga/armazenamento (/PAT 80/), ou seja, as únicas instruções que acessam a memória são a de carga de dado de memória para registrador e a de armazenamento do conteúdo de registradores na memória, as demais instruções realizando as operações somente com dados entre os registradores; (b) nas instruções de carga e armazenamento o pipeline é suspenso para que aconteça um ciclo extra para acesso à memória; (c) o pipeline não possui mecanismos de interlock, isto é, conflitos ocasionais entre os estágios do pipeline não devem existir ou devem ser resolvidos em tempo de compilação, e não por hardware em tempo de execução.

Um processador do tipo von Neumann, com as restrições acima mencionadas, requer um mínimo de etapas distintas e em seqüência para a execução de instruções armazenadas em memória (/MAN 79/). Estas etapas são:

E - Cálculo e fornecimento à memória do endereço onde está a próxima instrução a ser executada pelo processador.

I - Leitura da instrução na memória.

D - Interpretação (Decodificação) da instrução recebida, determinando a operação a ser efetuada, o endereço dos operandos a serem utilizados e o endereço destino onde será guardado o resultado da operação.

R - Leitura dos operandos e escrita destes nos armazenadores temporários (registradores) na entrada da unidade funcional (ULA, deslocador, etc).

O - Operação na unidade funcional (ULA, Deslocador, etc) sobre os operandos lidos.

W - Escrita do resultado da operação no local de destino.

Um processador do tipo von Neumann é, por natureza, um sistema digital seqüencial. Para aumentar seu desempenho pode-se:

a) Diminuir o número de eventos: implica unir dois eventos consecutivos pela eliminação de etapas intermediárias eventualmente desnecessárias (por exemplo, eliminar a barreira temporal entre leitura de operandos e a operação destes na unidade funcional).

b) Diminuir o tempo de cada evento: implica paralelizar um evento (somador seqüencial $\longrightarrow$ somador paralelo), ou aumentar o paralelismo do evento (somador carry-propagate $\longrightarrow$ somador carry-lock-ahead), ou realizar o evento com mais eficiência (somador standard-cell $\longrightarrow$ somador full-custom).

c) Realizar mais eventos ao mesmo tempo: implica paralelizar os eventos no tempo pela superposição da execução dos mesmos (introduzir pipeline de instruções).

A adoção das opções acima deve ser criteriosamente balanceada. Unir eventos pela eliminação da barreira temporal significa que o evento final será provavelmente de duração maior. Além disto, a inclusão de pipeline implica que eventos diferentes serão realizados em paralelo, mas a duração será obrigatoriamente a do evento maior, pensando-se em estágios de pipeline com duração fixa.

Deseja-se ressaltar aqui o estudo do pipeline, o qual se dá em nível da arquitetura do processador. Assim sendo, não será discutido neste capítulo a otimização quanto aos dispositivos conforme relatado acima: otimização da ULA, bufferização dos registradores, etc. O desenvolvimento total do Risco, contudo, levou em conta estes dados para implementação de um processador com melhor desempenho possível dentro da tecnologia disponível. As otimizações não arquiteturais do Risco são descritas no capitulo 5.

Raciocinando-se em termos de arquitetura do processador, para um aumento do desempenho pode-se agrupar as etapas definidas anteriormente, bem como introduzir um pipeline de instruções.

Agrupando-se as etapas, têm-se as seguintes possibilidades de implementação, conforme a figura 4.1.

No diagrama da figura 4.1, DRO (caso nº 7), por exemplo, significa que a decodificação (D), a leitura dos operandos (R) e a operação (O) acontecem na mesma fase de relógio.

1)	E	I	D	R	O	W
2)	E	I	D	R	OW	
3)	E	I	D	RO	W	
4)	E	I	D	ROW		
5)	E	I	DR	O	W	
6)	E	I	DR	OW		
7)	E	I	DRO	W		
8)	E	I	DROW			
9)	E	ID	R	O	W	
10)	E	ID	R	OW		
11)	E	ID	RO	W		
12)	E	ID	ROW			
13)	E	IDR	O	W		
14)	E	IDR	OW			
15)	E	IDRO	W			
16)	E	IDROW				
17)	EI	D	R	O	W	
18)	EI	D	R	OW		
19)	EI	D	RO	W		
20)	EI	D	ROW			
21)	EI	DR	O	W		
22)	EI	DR	OW			
23)	EI	DRO	W			
24)	EI	DROW				
25)	EID	R	O	W		
26)	EID	R	OW			
27)	EID	RO	W			
28)	EID	ROW				
29)	EIDR	O	W			
30)	EIDR	OW				
31)	EIDRO	W				
32)	EIDROW					

Figura 4.1 Possibilidades de pipeline

Para cada uma das 32 possibilidades elencadas, pode-se fazer um pipeline de instruções. Por exemplo, o caso 1 admite ao menos dois pipelines onde a memória é ocupada em todo os ciclos (figuras 4.2 e 4.3) e um terceiro pipeline onde a memória é subutilizada.

Figura 4.2 Duas instruções a cada ciclo de memória.

Figura 4.3 Uma instrução a cada ciclo de memória.

Notar a memória fica desocupada durante 1/3 de tempo de seu ciclo mínimo.

Figura 4.4 Duas instruções a cada três ciclos possíveis de memória.

Pela análise da figura 4.2, pode-se observar que:

1º) E e I ocorrem simultaneamente, tornando necessários dois barramentos externos para comunicação com a memória, além de exigir uma organização de memória que permita escrita do endereço em um banco simultaneamente ao acesso dos dados de um outro banco de memória. Este pipeline permite o acesso de duas instruções a cada ciclo de memória.

2º) R e W ocorrem na mesma fase, tornando necessário que o banco de registradores permita a leitura de dois operandos com a escrita do resultado da operação (3 acessos simultâneos). Além disto, são necessários 3 barramentos internos para ligação entre o banco de registradores e a unidade funcional, implicando maior custo do CI, já que em silício as conexões possuem custo elevado.

3º) O resultado de uma instrução é escrito no banco de registradores ao fim da etapa de W. Devido ao pipeline, este resultado não estará disponível para a instrução seguinte, pois a leitura dos operandos (R) desta foi realizada uma fase antes. Deste modo, define-se que este pipeline tem atraso de uma instrução para uso dos dados resultantes de uma instrução. Além disto, para que o atraso seja de apenas uma instrução, é necessário que a saída da

unidade funcional seja não só direcionada para o destino, mas também realimentada para sua entrada, no caso em que a instrução seguinte use os dados da anterior. Por exemplo, R6=R5+R1 seguido de R9=R6+4.

4º) Uma instrução de salto somente tem o resultado, isto é, o endereço destino do salto, ao fim da etapa de W, e este resultado deve ser o endereço de busca (E) da nova instrução. Como se observa no pipeline, quatro instruções já haviam sido buscadas e continuam em execução antes que a instrução de salto tivesse sido concluída. Define-se, então, que este pipeline tem atraso de quatro instruções para a instrução de salto.

O segundo pipeline (figura 4.3) tem metade do desempenho do primeiro (figura 4.2), pois uma nova instrução só é realizada a cada ciclo de memória. No entanto, esta implementação tem qualidades que devem ser analisadas. Os recursos de hardware necessários são substancialmente menores: um único barramento externo pois E e I estão em fases diferentes, organização da memória mais simples, atraso para o salto de duas instruções e sem atraso para uso de dados (não necessita da realimentação da saída da unidade funcional para a sua entrada).

No projeto do Risco assumiram-se diferentes condições de contorno, tendo em vista as limitações tecnológicas de fabricação do processador e de sua placa. Por questões de custo quanto ao encapsulamento, largura menor do barramento de memória e maior simplicidade para a organização desta, decidiu-se que haveria apenas um barramento externo multiplexado de dados e endereços. Assim sendo, E e I não podem estar na mesma fase no pipeline do Risco.

Nem só as razões de custo e desempenho influenciaram a arquitetura do Risco. Em /HEN 84/ demonstra-se ser difícil preencher o pipeline com instruções úteis (sem inserir NOOP) após o salto

postecipado quando o atraso deste é de duas ou mais instruções. Deste modo, decidiu-se que, no Risco, o atraso nas instruções de salto seria de, no máximo, uma instrução. Portanto, a escrita do resultado de uma instrução de salto (o endereço de salto) deve ocorrer antes da busca da segunda instrução seguinte ao salto. Esta decisão também tem influência na implementação da parte de controle, pela sua simplificação. Como mencionado anteriormente, os conflitos entre os estágios do pipeline devido a dependências de dados são resolvidos em software através do compilador.

Por simplicidade na implementação, decidiu-se que o pipeline deveria ser tal que tornasse desnecessário implementar o circuito para realimentação da saída da ULA para sua entrada, permitindo uma PO de menor área e com controle mais simples. Assim sendo, o atraso para uso dos dados de uma instrução deveria ser zero. Portanto, a escrita de um resultado de uma instrução deve ocorrer antes da leitura dos operandos da instrução seguinte a esta.

Finalmente, a leitura de dois operandos simultaneamente à escrita do resultado exige três barramentos internos na parte operativa e registradores tipo mestre-escravo no banco de registradores. Já duas leituras simultâneas ou uma escrita necessitam de dois barramentos e registradores mais simples, até do tipo latch (ou seja, quando da escrita, a saída é igual à entrada). Em /KAT 85/, relata-se que a primeira solução mostrou-se mais lenta devido à maior área consumida durante a implementação. Por estas razões, decidiu-se que no Risco haveria dois barramentos internos para comunicação entre as unidades funcionais e o banco de registradores, sendo este formado por registradores mais simples. Conseqüentemente, no pipeline do Risco, R e W não podem estar na mesma fase.

Analisando-se as possibilidades de etapas na execução de uma instrução, listadas na figura 4.1, e as

restrições do pipeline do Risco, verifica-se que, nos tipos de 17 a 32, E e I sempre ocorrem na mesma fase e que, em 4, 8, 12 e 16, R e W acontecem na mesma fase. Eliminando-se estas possibilidades, resta o diagrama da figura 4.5.

1)	E	I	D	R	O	W
2)	E	I	D	R	OW	
3)	E	I	D	RO	W	
5)	E	I	DR	O	W	
6)	E	I	DR	OW		
7)	E	I	DRO	W		
9)	E	ID	R	O	W	
10)	E	ID	R	OW		
11)	E	ID	RO	W		
13)	E	IDR	O	W		
14)	E	IDR	OW			
15)	E	IDRO	W			

Figura 4.5 Possibilidades restantes de pipeline

Para verificar o atendimento das demais restrições do pipeline do Risco, é necessário esboçar os pipelines para cada um dos tipos da figura 4.5. O pipeline resultante deverá ser deslocado de duas fases pois, com deslocamento de apenas uma fase, E e I estarão na mesma fase e, com três ou mais, a memória será subutilizada. Os pipelines resultantes são:

```
1)
E   I   D   R   O   W
        E   I   D   R   O   W
                E   I   D   R   O   W
                        E   I   D   R   O   W

2)
E   I   D   R   OW
        E   I   D   R   OW
                E   I   D   R   OW
                        E   I   D   R   OW

3)
E   I   D   RO  W
        E   I   D   RO  W
                E   I   D   RO  W
                        E   I   D   RO  W
```

5)

E	I	DR	0	W					
		E	I	DR	0	W			
				E	I	DR	0	W	
						E	I	DR	0	W

6)

E	I	DR	OW					
		E	I	DR	OW			
				E	I	DR	OW	
						E	I	DR	OW

7)

E	I	DRO	W					
		E	I	DRO	W			
				E	I	DRO	W	
						E	I	DRO	W

9)

E	ID	R	0	W					
		E	ID	R	0	W			
				E	ID	R	0	W	
						E	ID	R	0	W

10)

E	ID	R	OW					
		E	ID	R	OW			
				E	ID	R	OW	
						E	ID	R	OW

11)

E	ID	RO	W					
		E	ID	RO	W			
				E	ID	RO	W	
						E	ID	RO	W

13)

E	IDR	0	W					
		E	IDR	0	W			
				E	IDR	0	W	
						E	IDR	0	W

14)

E	IDR	OW						
		E	IDR	OW				
				E	IDR	OW		
						E	IDR	OW

15)

E	IDRO	W						
		E	IDRO	W				
				E	IDRO	W		
						E	IDRO	W

Por inspeção, verifica-se que os pipelines 1, 5, 9 e 13 tem R e W na mesma fase, o que vai contra a restrição proposta em parágrafo anterior. O pipeline 1 também não atende a restrição de atraso do salto igual ou menor a um.

Eliminando-se, portanto, os pipelines que não atenderam as restrições, tem-se o diagrama da figura 4.6.

```
2)
E    I    D    R    OW
          E    I    D    R    OW
                    E    I    D    R    OW

3)
E    I    D    RO   W
          E    I    D    RO   W
                    E    I    D    RO   W

6)
E    I    DR   OW
          E    I    DR   OW
                    E    I    DR   OW

7)
E    I    DRO  W
          E    I    DRO  W
                    E    I    DRO  W

10)
E    ID   R    OW
          E    ID   R    OW
                    E    ID   R    OW

11)
E    ID   RO   W
          E    ID   RO   W
                    E    ID   RO   W

14)
E    IDR  OW
          E    IDR  OW
                    E    IDR  OW

15)
E    IDRO W
          E    IDRO W
                    E    IDRO W
```

Figura 4.6 Pipelines finais

Os pipelines restantes atendem as restrições impostas para o Risco. A escolha final dar-se-á pelo estudo preliminar sobre a implementação destes pipelines, mais especificamente sobre o tempo necessário em cada estágio para conclusão da etapa. Inicia-se discutindo os estágios de escrita do endereço E e leitura da instrução I.

Tendo o Risco apenas um barramento externo, a busca de uma instrução é feita, numa primeira fase, fornecendo o endereço no barramento junto com o sinal para um latch externo armazenar este endereço. Na fase seguinte, o processador lê a instrução no barramento.

Havendo apenas um latch externo para armazenagem do endereço, não há problemas de custo ou implementação em se usar um latch de alta velocidade. Neste caso, o tempo necessário nesta primeira fase depende somente do tempo de propagação do latch e da rapidez com que o processador consegue colocar o endereço no barramento (uma questão de dimensionamento dos buffers de saída).

Já na segunda fase da busca de instrução, o atraso depende das memórias e da organização destas, não havendo como o processador acelerar esta resposta. O uso de uma organização de memórias de alto desempenho tende a aumentar os custos do sistema, caso a memória real seja muito grande, e, mesmo neste caso, existem as limitações da tecnologia quanto à velocidade.

Em resumo, a escrita do endereço pelo processador pode ser feita em um tempo substancialmente menor do que a leitura da instrução. Isto implica que no pipeline o estágio de escrita de endereço tem duração menor que o estágio de leitura da instrução.

Sendo o Risco um processador do tipo RISC, a decodificação e interpretação das instruções é um processo rápido, pois estas são de número reduzido e simples, com formato e tamanho fixos. A implementação pode ser feita sem microcódigo. Sendo a parte de controle de

processadores RISC em torno de 10% a 20% da área total do circuito, esta pode ser projetada para máxima velocidade sem prejuízo substancial na área consumida. Portanto, o estágio de decodificação no pipeline tem uma pequena duração, comparada, por exemplo, à soma em uma ULA.

O estágio de leitura de operandos é mais crítico. O tempo, neste caso, é determinado pela capacidade de corrente dos buffers de saída do banco de registradores. Como este banco geralmente possui muitos registradores, a diminuição do tempo de leitura devido ao aumento dos buffers irá provocar um aumento considerável em área. Além disto, o barramento sofrerá um aumento da carga capacitiva associada a ele, tendendo a se tornar mais lento, contrariando o objetivo desejado.

O estágio de operação nas unidades funcionais também exige cuidados. A princípio, pode-se adotar uma organização tal nestas unidades para que o tempo de operação seja o desejado (por exemplo, através do uso de uma ULA carry look ahead). No entanto, isto implica diretamente aumento de área na parte operativa.

Finalmente, o estágio de escrita do resultado segue o raciocínio semelhante feito quanto ao tempo de leitura dos operandos: o tempo é determinado pelo buffer de saída da unidade funcional. Como aqui há apenas um buffer por unidade funcional, e sendo estas de número reduzido, o uso de grandes buffers não irá provocar demasiado consumo de área.

Para recapitular, concluiu-se que o tempo para escrita de endereços é bem menor do que a leitura da instrução. Além disto, a decodificação e a escrita são de tempo relativamente pequeno, enquanto a leitura dos operandos para a ULA e a operação nas unidades funcionais são de tamanho proporcionalmente maior.

Assim, no projeto do pipeline, já estando E e I em fases diferentes (e, no caso, E bem menor do que I), é uma decisão natural, decorrente do acima exposto, que D e W estejam na mesma fase de E, e R e O na mesma fase de I.

Retornando-se aos pipelines restantes depois de eliminados os que não atendiam as restrições iniciais listados na figura 4.6, foram selecionados aqueles que possuem E, D e W em uma fase, e I, R e O na outra. Ou, caso não seja encontrado, que ao menos I, R e O estejam na mesma fase, pois são os maiores.

Dois pipelines atendem a nova restrição: 3 e 15, conforme diagrama abaixo.

```
3)
E   I   D   RO  W
        E   I   D   RO  W
                E   I   D   RO  W
                        E   I   D   RO  W

15)
E   IDRO  W
        E   IDRO  W
                E   IDRO  W
                        E   IDRO  W
```

O pipeline 15 apresenta as etapas de leitura da instrução, decodificação, leitura dos operandos e operação em um único estágio. Isto fará com que a fase onde ocorre IDRO seja muito maior que a fase onde ocorre W/E, gerando relógios com tempos ativos muito distintos e, portanto, mais difíceis de serem implementados e distribuídos para todo o circuito.

Outro problema ocorre quando do projeto em si. Uma das fases do pipeline deveria fazer a leitura da instrução, decodificação da mesma, leitura dos operandos e operação na unidade funcional. Isto implicaria um banco de registradores com barramento de decodificadores estáticos, sem possibilidade de pré-carga. A ULA também não poderia ser dinâmica. Portanto, ter-se-ia um alto custo em área e

tempo devido à soma de atrasos combinacionais deste caminho crítico.

Mesmo não se levando em consideração os dois parágrafos anteriores, o pipeline 15 tem a duração do ciclo de instrução obrigatoriamente maior que o pipeline 3. Observe-se o porquê. Nos dois pipelines, W e E ocorrem na mesma fase. No pipeline 3, D também ocorre junto com W e E. Como estes 3 estágios são de tempos proporcionalmente pequenos, esta fase ocorre provavelmente com a mesma duração (pequena) nos dois pipelines. A diferença fundamental ocorre na outra fase. Enquanto em 15 os estágios I, R e O ocorrem em seqüência na mesma fase, no pipeline 3 o estágio I ocorre em paralelo com os estágios R e O. Conclui-se que o pipeline 3 executa suas instruções mais rapidamente devido ao maior paralelismo.

O pipeline 3 é o pipeline escolhido para o Risco, com uma variação feita durante a fase RO/I. Sendo RO um estágio, acontecendo seqüencialmente durante uma fase, ocorre que os barramentos internos ficam ocupados durante toda esta etapa. Modificando o pipeline como na figura 4.7, pela separação de RO em duas fases consecutivas, obtém-se as seguintes vantagens:

a) Durante a fase O, o barramento interno fica livre. Isto permite que seja feita uma pré-carga nos barramentos antes da fase de W, possibilitando que os buffers de saída das unidades funcionais sejam menores (possuindo somente os transistores NMOS) e/ou mais rápidos (os transistores podem ser maiores com a mesma área do caso sem pré-carga). Isto é particularmente importante nas instruções de salto, pois o resultado desta, o endereço, é escrito não só no registrador destino (o contador de programa, PC) mas também no barramento externo para a busca da instrução no endereço do salto.

b) O barramento livre durante a fase de O, quando executando uma instrução de acesso à memória, permite que seja implementada a operação de carga de memória com pré ou pós-incremento de um dos registradores fonte (útil em instruções do tipo Rn=M[Rm+Rk]), sem que seja necessário introduzir mais ciclos extras no pipeline. Neste caso, a ausência da pré-carga não é problemática, pois o registrador pós/pré-incrementado a ser alterado é interno ao processador e o tempo necessário é suficiente.

c) Para que RO seja dividido e o pipeline ainda mantenha a simetria (ou seja, cada fase com a mesma duração temporal para facilidade de geração e distribuição), é necessário que o estágio I seja dividido em dois. O estágio I é o tempo que o processador espera pela chegada da instrução da memória. A divisão de I em I1 e I2 não afeta em nada o funcionamento interno, pois a instrução é guardada em um latch, não importando se isto acontece ao fim de I (sem divisão) ou de I2 (subdividido).

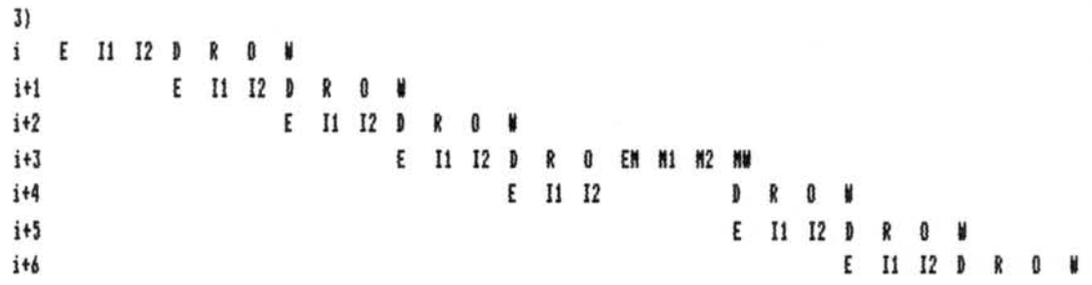


Figura 4.7 Pipeline do Risco

Na figura 4.7, pode-se observar o comportamento do pipeline com uma instrução de carga/armazenamento (i+3), causando a suspensão do pipeline para inserir mais um ciclo de memória. Este exato pipeline foi aquele implementado no Risco pelo conjunto de razões expostas neste capítulo.

5 IMPLEMENTAÇÃO

Neste capítulo, vários detalhes relativos à implementação do Risco serão discutidos, inclusive em relação ao test-chip.

5.1 Parte operativa

A parte operativa (PO) do Risco é organizada de forma a permitir a execução de uma instrução em paralelo com a busca da próxima instrução. A PO tem número de bits igual a palavra de dados e de endereço da arquitetura, e seus elementos estão distribuídos ao longo dos barramentos. A PO é projetada na forma de *bit-slice*, sendo estes repetidos até formar elementos de 32 bits.

Um diagrama da PO encontra-se na figura 5.1. Seus componentes são os seguintes:

a) Elementos de armazenamento. São os diversos registradores da PO. Dividem-se entre aqueles pertencentes à arquitetura, visíveis ao usuário (dispostos em um banco de 32 registradores que incluem o PC, SP, PSW e R00), e aqueles utilizados internamente (RUA, RUB, RDA, RDB, RMEM, CONST, CONSTEXT e PCINC) para armazenar dados temporários nas unidades funcionais.

b) Elementos de transformação. São as unidades funcionais utilizadas para operar sobre os dados da instrução. As unidades são a unidade lógico-aritmética (ULA), a unidade de deslocamento (UD), a unidade da constante (UC) e a unidade do contador de programa (UPC).

c) Elementos de comunicação. Compõe-se dos barramentos distribuídos ao longo da PO: BA, BB, BOUT e BSIS, além dos buffers para amplificação e ligação entre alguns destes barramentos. São utilizados para transferir dados entre os registradores e as unidades funcionais.

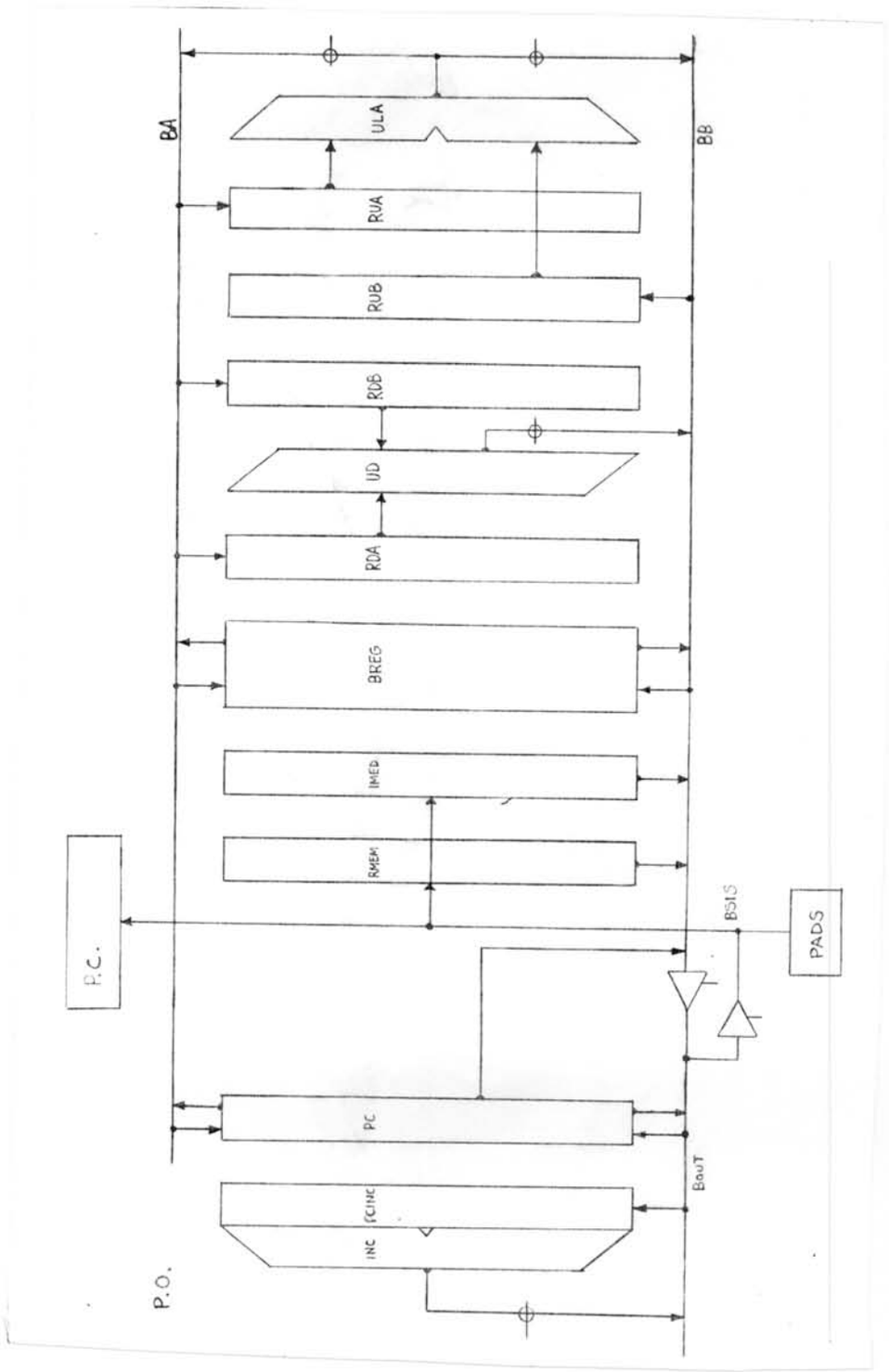


Figura 5.1 Diagrama de blocos da parte operativa

5.1.1 Barramentos

A PO possui quatro barramentos: BA, BB, BOUT e BSIS. Os barramentos BA e BB são barramentos complementares bifilares, ou seja, um dado é transmitido por dois condutores, com informação complementar. Deste modo, existem fisicamente BAd e BAdn (BAdn = ~BAd), e BBd e BBdn (BBdn = ~BBd). BA e BB são utilizados basicamente para realizar a comunicação entre o banco de registradores e unidades funcionais. Ambos os barramentos utilizam pré-carga antes de serem utilizados.

A escolha por barramentos bifilares complementados pré-carregados foi adotada para permitir maior imunidade a ruídos e para diminuir o tamanho do banco de registradores. Sendo um sinal complementado, um *sense-amplifier* diferencial pode detectar com segurança os níveis lógicos e mantê-los fixos, diminuindo os efeitos de ruído. A pré-carga permite eliminar no buffer de saída dos registradores os transistores P, diminuindo a área necessária à implementação destes.

O barramento BOUT é unifilar, sem pré-carga, e é utilizado pela unidade do contador de programa para realizar o incremento do PC. BSIS, também unifilar e sem pré-carga, realiza a comunicação entre o processador e o exterior do chip (via pads), e é disposto perpendicularmente aos demais barramentos. A decisão de adotar barramentos simples em BOUT e BSIS deveu-se à extensão de comprimento muito menor (em relação a BA e BB) e por haver poucos elementos conectados a estes - ruído e excesso de área consumida não eram problemas potenciais neste caso.

Na escrita de dados na memória, BSIS recebe o conteúdo de BOUT. BOUT, por sua vez, pode receber a saída da unidade do contador de programa ou o conteúdo da barramento BB. No primeiro caso, BSIS escreve o endereço da instrução seguinte ao PC atual. No outro caso (BOUT recebe BB), BSIS escreve o endereço resultante de uma instrução de salto ou

escreve o dado a ser armazenado por uma instrução de armazenamento. Observe-se que, no primeiro caso, os barramentos BOUT e BB não estão conectados, permitindo, assim, que a busca da instrução seguinte seja feita em paralelo à execução da instrução.

Na leitura de dados da memória, o dado que chega via BSIS é direcionado para a parte de controle e para a unidade da constante, no caso de busca de instrução, ou é direcionado para o registrador temporário RMEM, caso seja um dado buscado por uma instrução de carga de registrador.

5.1.2 Banco de registradores (BR) e registradores temporários

Os elementos armazenadores da PO podem ser classificados entre aqueles registradores visíveis ao usuário, especificados pela arquitetura, e aqueles registradores temporários, utilizados nas unidades funcionais.

Os registradores da arquitetura estão dispostos em um banco de 32 registradores, sendo o registrador 0 uma constante de valor 0 (R00). Apesar de implementados separadamente, o contador de programa (PC=R31) e a palavra de status do processador (PSW=R01) são endereçados logicamente neste banco de registradores.

Os registradores do banco de registradores são implementados com a célula de memória da figura 5.2. Esta é composta de dois inversores realimentados para manter a informação e de quatro transistores para conexão aos dois barramentos bifilares. Para uma leitura, o barramento é pré-carregado ($Bxdn_i$ e Bxd_i são conectados a VDD) previamente ao acionamento da linha de seleção sel_Bx_j. Quando a seleção é ativada, os transistores conectam o par de inversores ao barramento, onde a célula descarregará um dos fios complementados para 0 e manterá o outro em VDD. Duas leituras simultâneas são possíveis. Para a escrita, o

dado é colocado complementado no barramento. Quando a linha de seleção ativar, a célula recebe o valor do barramento. Também são permitidas duas escritas simultâneas, obviamente em células diferentes.

A célula de memória é uma variação da célula clássica a seis transistores, tendo seu funcionamento descrito em (/SUZ 81/). O principal motivo desta escolha é a necessidade de uma dupla escrita simultânea no banco de registradores, exigido na instrução de carga com incremento/decremento do segundo registrador fonte.

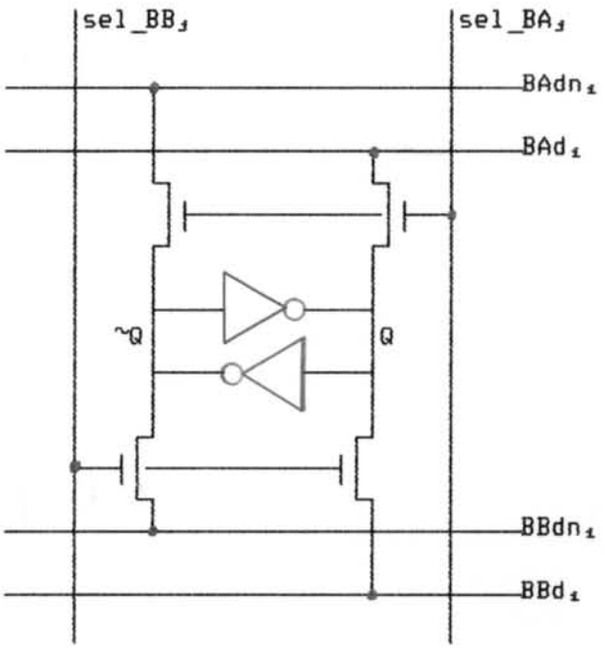


Figura 5.2 Célula de memória

A PO contém, ainda, outros registradores não visíveis no conjunto de instruções, utilizados para armazenar dados temporários: RUA e RUB na unidade lógico-aritmética, RDA e RDB na unidade de deslocamento, RMEM para armazenamento temporário durante a instrução de carga, CONST e CONSTEXT na unidade da constante, e PCINC na unidade do contador de programa.

Os registradores temporários foram implementados conforme a figura 5.3. A ativação do sinal ~W abre a

realimentação dos dois inversores e a de W conecta a entrada ao primeiro inversor, permitindo o armazenamento do novo dado de entrada no registrador. O sinal R ativa o buffer de saída, conectando o segundo inversor à saída. Nos registradores RUA, RUB, RDA, RDB, CONST e PCINC, foi eliminado o buffer de saída acionado por R, pois a barreira temporal é realizada na saída das unidades funcionais onde estes registradores estão conectados.

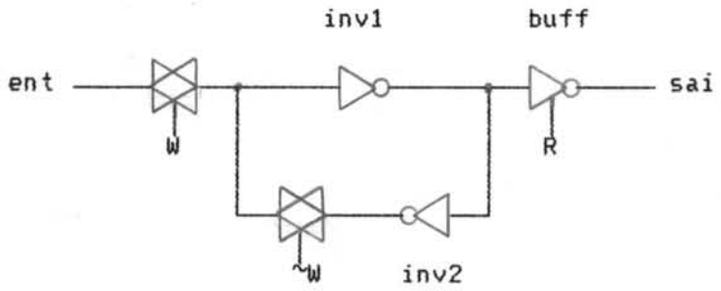


Figura 5.3 Registradores temporários

A escolha por este tipo de registrador deu-se pela facilidade de projeto e segurança de utilização, pois o funcionamento é completamente estático, não inversor, e os únicos transistores a serem dimensionados em função da carga são os do buffer de saída. Além disto, a operação é bastante flexível, podendo ser usado como latch transparente (sinal R sempre ativo, ou com um inversor não controlado na saída) ou como latch com saída tri-state (W e R nunca ativos simultaneamente).

Outra razão na escolha deste registrador é a facilidade para se implementar um seletor na entrada e/ou um demultiplexador na saída. Na figura 5.4 está um exemplo com seleção para duas entradas e demultiplexação para duas saídas.

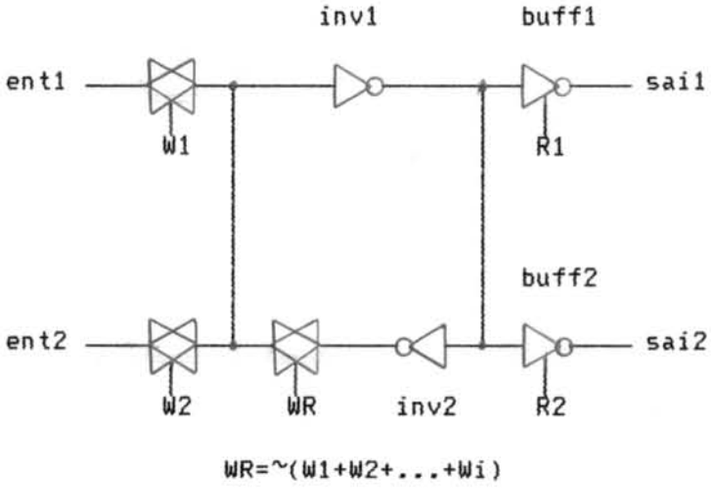


Figura 5.4 Mux de entrada e saída

5.1.3 Unidade da constante (UC)

A unidade da constante compõe-se do registrador CONST para armazenar a constante presente na palavra de instrução, de um subcircuito composto basicamente por seletores para realizar a extensão da constante e do registrador CONSTEXT para guardar o resultado estendido e servir como fonte do segundo operando das instruções que utilizam a constante. A figura 5.5 mostra a unidade da constante.

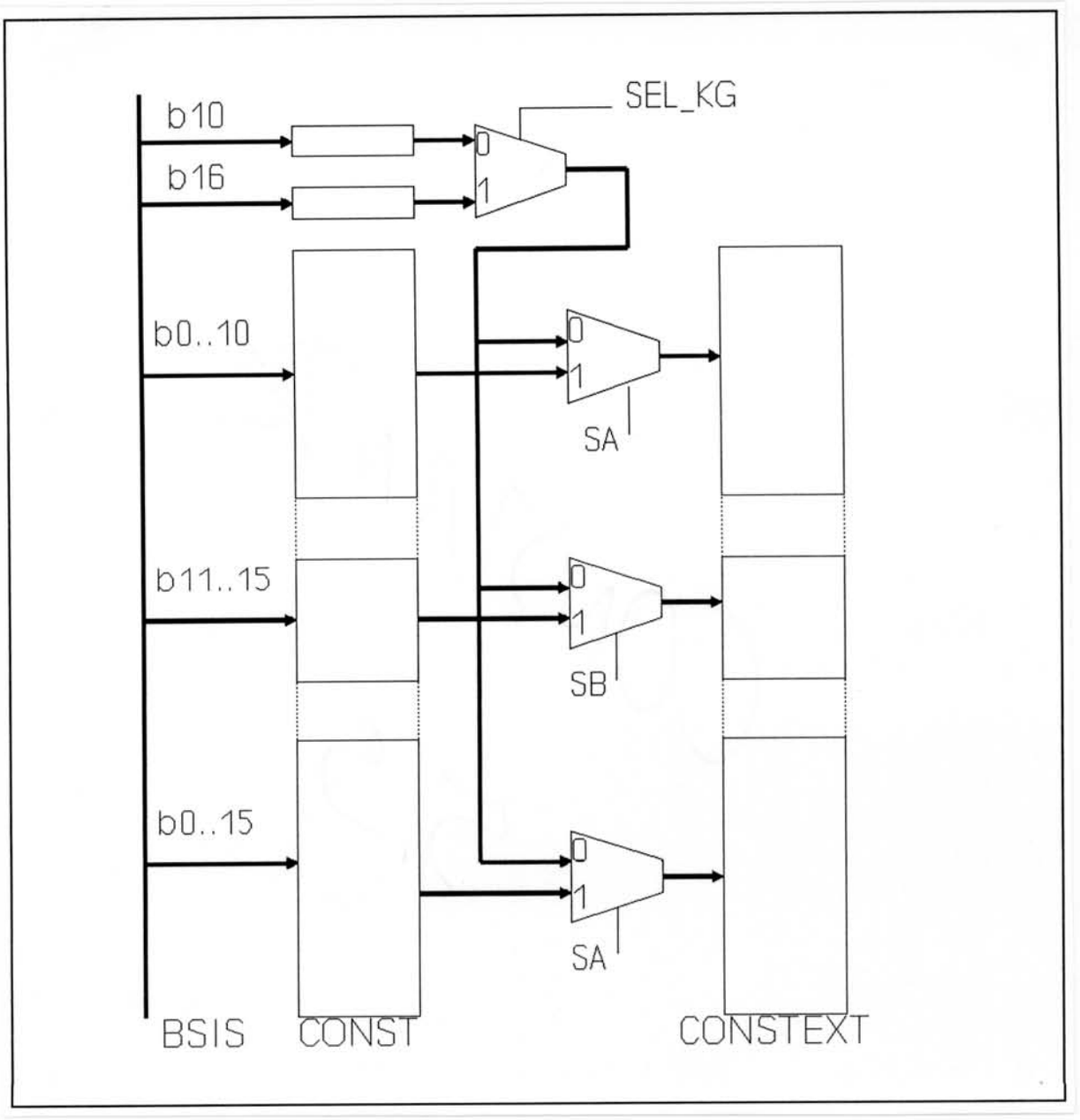
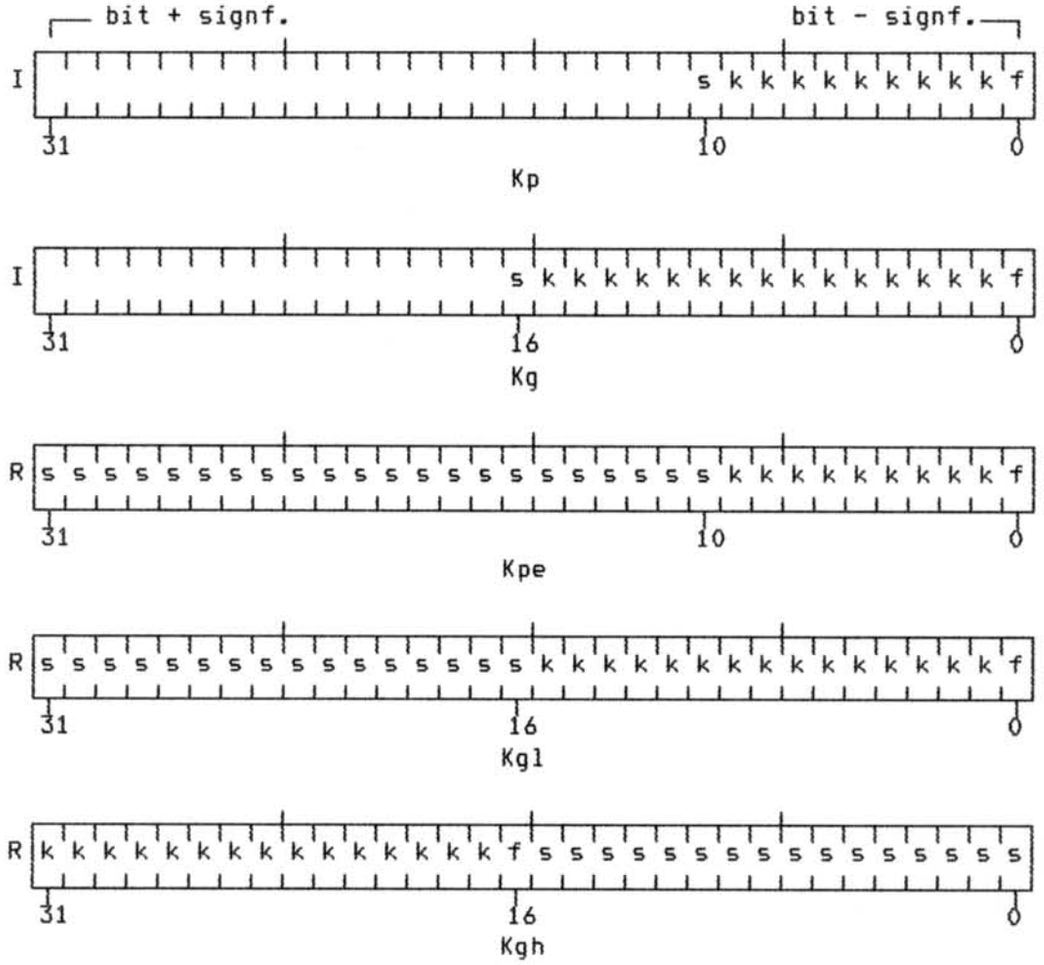


Figura 5.5 Unidade da constante

De acordo com a especificação da arquitetura, uma constante na palavra de instrução é utilizada internamente como o segundo operando da instrução. Esta constante, de dois tamanhos possíveis (Kp e Kg), tem seu bit mais significativo estendido de três formas distintas (Kpe, Kgl e Kgh). Para maior clareza, reproduz-se na figura 5.6 o diagrama já apresentado no capítulo 3, exemplificando-se a extensão da constante.



Onde:
I - palavra de instrução, R - registrador interno da const. estendida,
Kp - constante de 11 bits em I, Kg - constante de 17 bits em I,
Kpe - constante Kp estendida em R, Kgl - constante Kg estendida em R,
Kgh - constante Kg estendida e rotacionada em R,
f - bit menos significativo (bit 0), k - demais bits da constante.
s - bit mais significativo (bit 10 em Kp, bit 16 em Kg),

Figura 5.6 Carregamento da Constante no Registrador Interno

Devido a esta exigência da arquitetura, o registrador interno CONSTEXT, que contém a constante já estendida (Kpe, Kgl ou Kgh), deverá apresentar uma das configurações dispostas na figura 5.7. Nesta figura um * indica o bit mais significativo de Kp ou de Kg na palavra de instrução e sua posição quando estendido.

	Bits de:		
inst	Kpe	Kgl	Kgh
00	00	00	16*
01	01	01	16*
02	02	02	16*
03	03	03	16*
04	04	04	16*
05	05	05	16*
06	06	06	16*
07	07	07	16*
08	08	08	16*
09	09	09	16*
10*	10*	10*	16*
11	10*	11	16*
12	10*	12	16*
13	10*	13	16*
14	10*	14	16*
15	10*	15	16*
16*	10*	16*	00
17	10*	16*	01
18	10*	16*	02
19	10*	16*	03
20	10*	16*	04
21	10*	16*	05
22	10*	16*	06
23	10*	16*	07
24	10*	16*	08
25	10*	16*	09
26	10*	16*	10*
27	10*	16*	11
28	10*	16*	12
29	10*	16*	13
30	10*	16*	14
31	10*	16*	15

Figura 5.7 Configurações possíveis de constante

CONST está ligado ao barramento BSIS, por onde chega a instrução buscada. Os 16 bits menos significativos de BSIS são carregados nos 16 bits menos significativos de CONST e, também, nos 16 mais significativos de CONST. Com o uso apropriado dos seletores, a partir da informação decodificada da instrução - se a constante é kp (11 bits) ou kg (17 bits) e a forma como esta deve ser estendida (Kpe, Kgl ou Kgh) - a saída do registrador CONST é direcionada

para diferentes bits do registrador CONSTEXT, obtendo-se a apropriada extensão da constante.

5.1.4 Unidade lógico-aritmética (ULA)

A unidade lógico-aritmética (ULA) realiza as operações básicas de soma, e-lógico, ou-lógico e ou-exclusivo-lógico, bem como gera os bits N (negativo), Ov (overflow), Z (zero) e C (carry) para a PSW. Controlando-se as entradas A, B (normal ou complementadas de 1) e Carry (0, 1, C e C negado da PSW) da ULA, obtém-se no Risco 15 operações, listadas na figura 5.8. Estas operações são utilizadas para realizar as instruções lógico-aritméticas da arquitetura (ADD, ADDC, SUB, SUBC, SUBR, SUBRC, AND, OR e XOR), bem como nas demais instruções onde cálculos na ULA são necessários.

A ULA recebe seus dados dos barramento BA e BB em uma fase de relógio (estágio R do pipeline), opera na fase seguinte (O) e coloca o resultado no barramento BA ou BB durante a última fase (W).

Função	Operação
and	A & B
or	A ¦ B
xor	A ^ B
subrc	~A + B + ~Carry
subrcnot	~A + B + Carry
subr	~A + B + 1
subrnc	~A + B + 0
subc	A + ~B + ~Carry
subcnot	A + ~B + Carry
sub	A + ~B + 1
subnc	A + ~B + 0
addcnot	A + B + ~Carry
addc	A + B + Carry
add1	A + B + 1
add	A + B + 0

Figura 5.8 Operações da ULA

A implementação da ULA requer maiores cuidados em arquiteturas RISC, pois seu desempenho é fundamental na velocidade de processamento da máquina. No Risco, procurou-se como objetivo inicial que as instruções fossem executadas no pico a uma taxa de pelo menos 10 MIPS (10 milhões de instruções por segundo). Como uma instrução é executada em um ciclo de máquina e uma operação na ULA em uma fase de relógio (1/3 de ciclo de máquina), a ULA deverá obter um resultado em torno de 30 nseg.

Procurou-se uma estimativa de primeira ordem para o desempenho da ULA. Caso a ULA fosse implementada com somadores *bit-slice* com propagação do carry por todos os 32 bits da PO, o atraso seria a soma do tempo de propagação de 32 portas. Com a estimativa de que o tempo de atraso de um gate na tecnologia disponível é de 2 nseg, o atraso da ULA seria da ordem de 60 a 70 nseg, o que não atendia as especificações.

Uma maneira de aumentar o desempenho seria pelo uso de somadores carry-lock-ahead de 4 bits. Neste caso, o desempenho seria da ordem de 15 a 20 nseg (8 blocos x atraso em torno de 2 nseg por bloco). Apesar de atender as especificações, este método não foi utilizado devido à maior complexidade no projeto e implementação física do bloco de carry-lock-ahead, além do desempenho estar super-dimensionado em relação aos demais elementos da PO.

Finalmente, a organização escolhida para a ULA foi a de carry-select (/WES 85/), com o cálculo do carry feito em blocos de 2 bits. A divisão da ULA em dois blocos de 16 bits permite um tempo de operação estimado entre 35 e 40 nseg (16 bits x 2 nseg por bit, sem contar o tempo no seletor), ainda acima do desejado. Com o cálculo do carry em blocos de 2 bits, este tempo diminui pela metade, em torno de 15 a 20 nseg. A escolha deste método, em detrimento do carry-lock-ahead de 4 bits, deveu-se à facilidade de projeto, pois este implica uma célula de dois bits apenas. O desempenho acima do necessário, em relação aos demais

elementos da PO, permite que o esforço de projeto se concentre na economia de área e potência, e não no aumento de velocidade. Um diagrama da ULA encontra-se na figura 5.9.

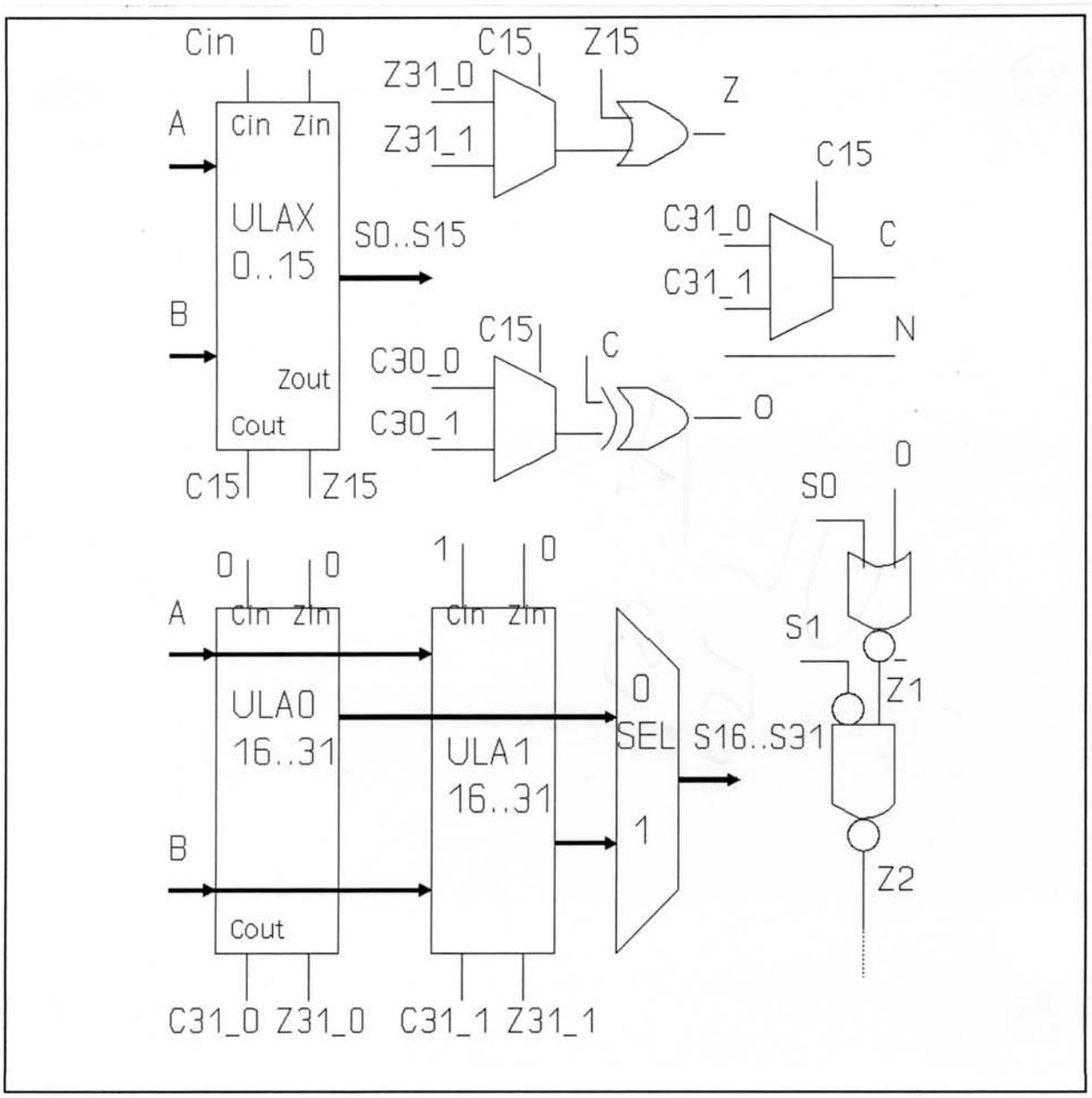


Figura 5.9 Diagrama da ULA do Risco

5.1.5 Unidade do contador de programa (UPC)

O registrador PC, o incrementador INC e o registrador temporário PCINC compõem a unidade do contador de programa. A função desta unidade é realizar o incremento do registrador PC e a busca de nova instrução.

O funcionamento básico desta unidade é descrito a seguir. Durante a fase 1, o conteúdo de INC, o próximo endereço de instrução, é colocado no barramento BOUT, sendo de BOUT direcionado para o exterior do processador através do barramento BSIS, bem como armazenado no PC. Na fase 2 o conteúdo de PC, o endereço da instrução que está sendo buscada, é colocado em BOUT e carregado em PCINC. Durante o restante da fase 2 e toda a fase 3, o incrementador INC, cuja entrada está conectada diretamente à saída de PCINC, calcula o endereço da instrução seguinte. Novamente, retorna-se à fase 1. Caso a instrução que estava sendo executada em paralelo não altere o PC, o processo se repete. Do contrário, a saída de INC é inibida, o barramento BOUT recebe o conteúdo do barramento BB, e PC e BSIS recebem BOUT.

A implementação do incrementador INC não é critica para o desempenho. O tempo para se executar o incremento é maior que uma fase de relógio: toda a fase 3 e grande parte da fase 2, já que a escrita do conteúdo do PC em PCINC é bastante rápida - o barramento BOUT é bastante pequeno e possui pouca carga capacitiva associada, e os buffers de saída de PC podem ser super-dimensionados para diminuir o tempo de escrita. Deste modo, estima-se um tempo de pelo menos meio ciclo de máquina, em torno de 50 nseg.

Com o uso de somadores com cálculo do carry realizado em blocos de 2 bits, o atraso da propagação total seria da ordem de 35 a 40 nseg (16 blocos x 2 nseg por bloco), o que atende as necessidades do projeto. O incrementador INC tem seu projeto facilitado, pois pode ser

visto como uma cadeia de somadores onde uma das entradas é zero (falso) e o carry in do primeiro bit é um (verdadeiro).

5.1.6 Unidade de deslocamento (UD)

O Risco possui uma unidade de deslocamento composta dos registradores temporários RDA e RDB e de um cross-bar (/SHE 82/) para realizar as instruções de deslocamento e rotação de até 32 bits, definidas pela arquitetura do Risco. Na figura 5.10, encontra-se um diagrama da disposição dos elementos de um barrel shifter de 3 bits.

Pela figura 5.10, explica-se o funcionamento básico do deslocador/rotacionador. Para deslocamentos à esquerda, DR recebe 0 e DL recebe o dado; para deslocamentos à direita, DR recebe o dado e DL recebe 0; e, para rotações, coloca-se o mesmo dado em DR e DL. Pela seleção da linha de controle apropriada, o dado deslocado ou rotacionado é colocado no barramento de saída DS.

No Risco, o dado a ser deslocado é sempre o operando fonte 1, que é transferido na fase 2 via barramento BA para o registrador RDA (e RDB recebe internamente 0), ou para RDB (e RDA recebe 0) ou, simultaneamente, para RDA e RDB. O valor do deslocamento ou rotação é fornecido pelos 5 bits menos significativos do operando fonte 2, sendo transferido via barramento BB para o decodificador que seleciona uma das linhas de comando. Durante a fase seguinte a UD está em operação, concluindo na fase 1 com o resultado colocado no barramento BB para ser escrito no destino.

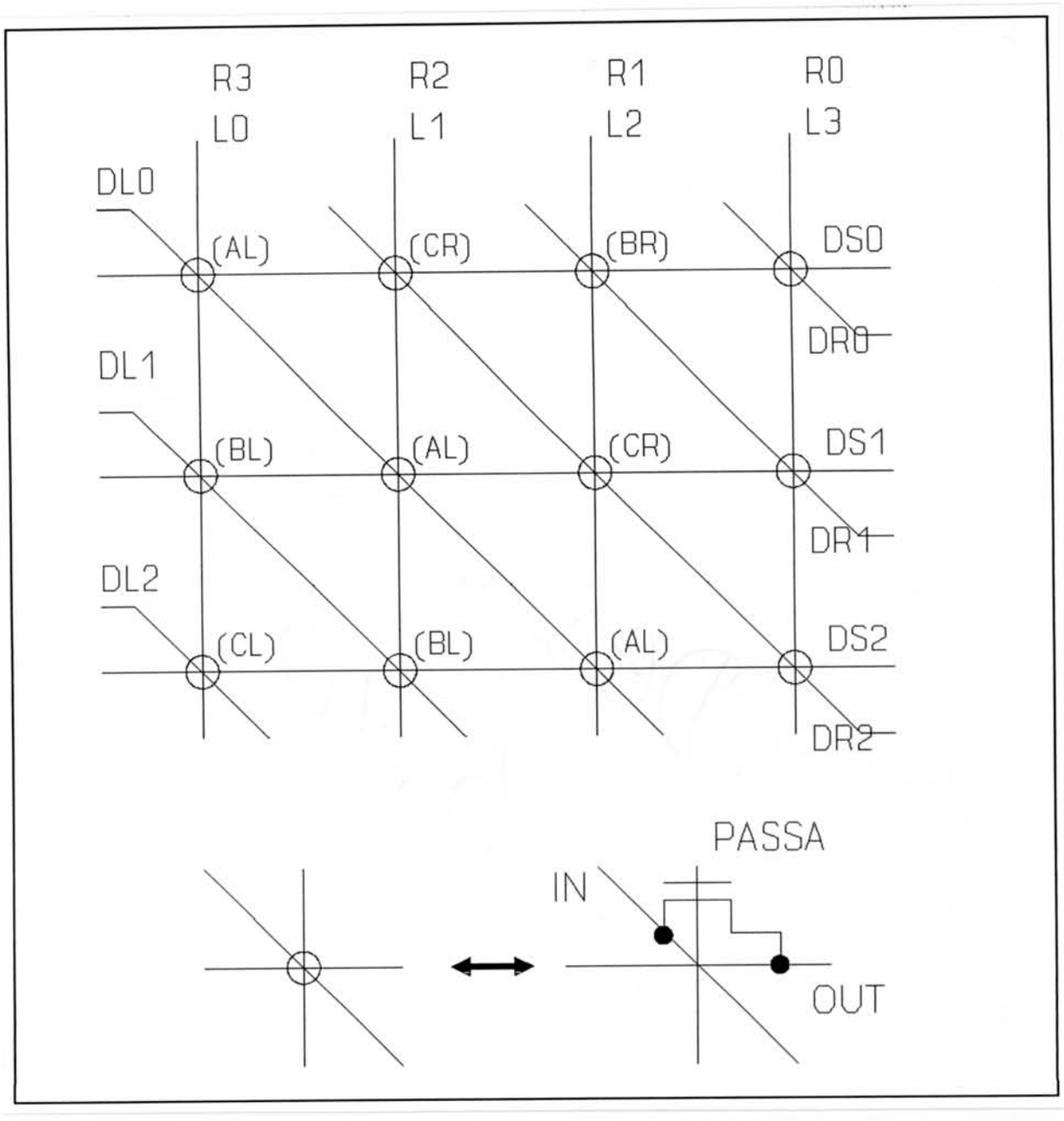


Figura 5.10 Barrel shifter

5.2 Parte de Controle

Neste item, descreve-se a concepção da parte de controle do Risco em função da arquitetura, da parte operativa e do pipeline escolhidos.

Uma parte operativa apresenta elementos de armazenamento, de transformação e de comunicação. No

Risco, estes elementos são basicamente os registradores, as unidades funcionais (ULA, UD e incrementador do PC) e os barramentos, respectivamente. A função do controle é ordenar no tempo a atuação destes elementos (a parte operativa) em função das instruções (a arquitetura) a serem executadas no pipeline. Assim sendo, na realização da parte de controle é necessário estabelecer para cada estágio do pipeline quais os elementos da parte operativa que são utilizados para cada uma das instruções do Risco.

Para determinar estes elementos, é necessário antes definir quais tarefas devem ocorrer ao longo de cada instrução para que cada uma destas seja executada. Deste modo, para cada uma das classes de instrução do Risco estabelece-se o que é necessário realizar.

Na figura 5.11 está uma primeira aproximação das tarefas necessárias para as instruções do Risco. Neste quadro, o destino é o registrador especificado no campo DST da palavra de instrução e os operandos fonte são os registradores FT1 e FT2 ou a constante, especificados também nos campos da palavra de instrução. Para maior concisão na figura não foram listadas as tarefas de busca de instrução e decodificação, comuns e iguais para todas as instruções.

Aritmético-Lógica

R Lê operandos fonte.
O Operação na ULA ou UD.
W Escreve resultado da operação no destino.

Salto

R Lê operandos fonte.
O Operação soma na ULA (cálculo do endereço de salto).
W Se condição verdadeira então escreve resultado no destino (normalmente o PC).

Carga/armazenamento

R Lê operandos fonte.
O Operação soma na ULA (cálculo do endereço do dado de memória).
Em Escreve resultado no barramento de memória (endereço do dado).
M1 Se armazenamento então barramento de memória recebe destino,
se altera operando fonte 2 então lê operando 2.
M2 Se armazenamento então barramento de memória recebe destino,
se altera operando fonte 2 então incremento ou decremento na ULA,
se carga então registrador de memória recebe barramento de memória.
MW Se altera operando fonte 2 então operando 2 recebe novo valor,
se carga então destino recebe registrador de memória.

Sub-rotina

R Lê destino (registrador apontador de pilha).
O Operação decremento na ULA (cálculo do endereço de pilha).
Em Se condição verdadeira então escreve resultado no barramento de memória (endereço de pilha),
se condição não verdadeira então fim da instrução.
M1 Barramento de memória recebe PC,
lê operandos fonte.
M2 Barramento de memória recebe PC,
operação soma na ULA (cálculo do endereço de sub-rotina).
MW PC recebe resultado.

Figura 5.11 Tarefas para instrução do Risco

Estas tarefas, em cada estágio, foram sendo sucessivamente refinadas até se chegar a um detalhamento no nível de transferência de registradores, ainda sem a inclusão de pipeline. Estas transferências entre registradores para cada instrução foram sobrepostas no tempo para se obter o pipeline, verificando-se, então, por inspeção, se não havia conflitos no uso de recursos da parte operativa. Nestes casos, a operação teve de ser deslocada, quando possível, para outro estágio do pipeline, caso contrário um novo recurso físico teve de ser incorporado na parte operativa. A verificação final foi feita por simulação em HDC.

No anexo 2, encontra-se a versão final deste detalhamento. Na figura 5.12 está a definição para as instruções aritmético-lógicas. Nesta figura, pode-se observar que, na estrutura do pipeline, as tarefas em cada estágio já estão ocorrendo em paralelo. Assim, a escrita de endereço da próxima instrução (E), a decodificação da instrução atual (D) e a escrita do resultado da instrução anterior (W) ocorrem paralelamente. O mesmo acontece para o início da leitura da próxima instrução (I1) e leitura dos operandos da instrução atual (R), bem como para a conclusão da leitura da próxima instrução (I2) e operação da instrução atual (O).

Uma condição a ser verificada pelo controle é observar se a instrução aritmético-lógica que está sendo concluída, pela escrita do resultado (W), tem como registrador destino o PC (o PC é um registrador lógico do banco de registradores, podendo ser utilizado por qualquer instrução). Caso isto ocorra, o controle evita que o dado para escrita do endereço da próxima instrução (E) venha do incrementador do PC, fazendo com que venha da ULA ou UD, como resultado da instrução que está sendo concluída.

```
#define  falso       0
#define  verdadeiro  ~falso

/**** FASE 1 - E ****/
se instrução_aritmético_lógica
então ( se dst=pc
        então bout := bb
        senão bout := inc;
        pc := bout
      );
bsis := bout;
ale := verdadeiro;

/**** FASE 2 - I1 (leitura da instrução) ****/
bout := pc;
pcinc := bout;
inc := pcinc + 1;
rd := verdadeiro;

/**** FASE 3 - I2 (leitura da instrução) ****/
proxri := bsis;
ri := proxri;
const := bsis;
rd := verdadeiro;

/**** FASE 1 - D  (decodificação) ****/
decodificação();

/**** FASE 2 - R (leitura dos operandos) ****/
ba := fa; /* fa = ft1 ou r00 ou dst          */
bb := fb; /* fb = ft2 ou constante estendida */
se usa_ula
então ( rua := ba;
        rub := bb;
        rula := codula
      )
senão ( rda := ba;
        rdb := bb;
        rud := codud
      );

/**** FASE 3 - O (operação) ****/
bb := verdadeiro;

/**** FASE 1 - W (escrita do resultado) ****/
se instrução_aritmético_lógica
então ( se usa_ula
        então bb := sai_ula
        senão bb := sai_ud;
        se (dst ~= pc)
        então dst := bb
      );
```

Figura 5.12 Parcela da descrição algorítmica

Estando definido para cada estágio do pipeline quais operações devem ocorrer, torna-se necessário estabelecer o mecanismo de seqüenciamento das instruções, bem como a geração dos comandos. O seqüenciamento deve prover mecanismos para buscar as instruções e tratar das exceções ao processamento, além de tratar das dependências do pipeline para as instruções que estão presentes neste. A geração de comandos deve fornecer os sinais de comando para cada ciclo de máquina em cada tipo de instrução.

Para a geração do seqüenciamento, realizado por uma máquina de estados finita (FSM), foi analisado o tipo de instruções que o Risco possui e o pipeline quando estas instruções estão presentes.

O pipeline de instruções do Risco encontra-se presente na figura 5.13. Neste diagrama, observa-se que uma instrução é decodificada sempre na fase 1 de relógio. Os comandos que resultam da interpretação serão utilizados durante o restante da instrução, nas fases 2, 3 e 1 seguintes. No caso de instruções onde há acesso à memória, os comandos são utilizados nas fases 2, 3 e 1 e, depois, novamente em outro ciclo de fases 2, 3 e 1. O padrão que se pode observar é que comandos são gerados para as seguintes configurações do pipeline:

fases: 2	3	1	
R	O	W	ciclo único das
I1	I2	-	instr. sem acesso à memória
-	-	E	
R	O	ME	1º ciclo das
I1	I2	-	instr. com acesso à memória
-	-	-	
M1	M2	MW	2º ciclo das
-	-	-	instr. com acesso à memória
-	-	E	

Todos os detalhes da implementação da P.C. do Risco encontram-se na descrição HDC do Risco, no anexo 1. A simulação foi feita em nível de gates, correspondendo a uma

implementação física real. Detalhes como tratamento de exceções (interrupção e reset) e geração e validação de comandos podem ser identificados na listagem do anexo.

No anexo 9, encontram-se telas com resultados de simulações HDC para diferentes programas: soma de 2 números, salto e interrupção.

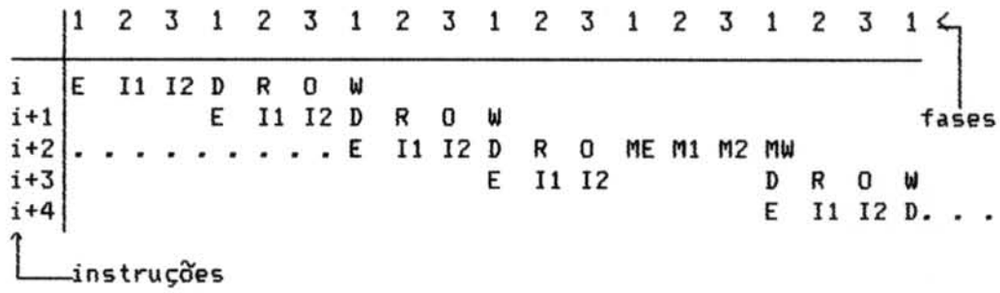


Figura 5.13 Diagrama do Pipeline de Instruções

5.3 Auxílio CAD

O Risco foi desenvolvido com extenso apoio de CAD. Em sua maioria, as ferramentas utilizadas foram desenvolvidas no CPGCC-UFRGS.

Cada célula-folha foi desenhada através de um editor simbólico (/MAR 89/). Deste modo, o projeto do Risco tornou-se maleável em termos de *foundry*, podendo ser fabricado em qualquer tecnologia CMOS com 2 níveis de metal. Isto foi feito inclusive para evitar que a tecnologia torne-se obsoleta devido ao tempo de projeto.

As células foram, então, montadas simbolicamente através do GGMOD (/CAR 92/) no ambiente Silex (/MAR 92/). No anexo 4, encontram-se layouts simbólicos e reais do Risco.

A simulação do Risco, do alto nível até o nível lógico, foi feita utilizando-se o HDC (/MAC 92/). A simulação elétrica foi feita utilizando-se o simulador Spice. No anexo 3 encontram-se descrições Spice e resultados de simulação para cada bloco, enquanto que no anexo 9 apresentam-se algumas telas com simulações HDC.

5.4 Test-Chip

Foi realizado um test-chip do Risco em tecnologia ES2, com transistor de canal 1.2 micra. O test-chip abrange a ULA, o contador de programa, vários registradores e multiplexadores. No anexo 4, uma planta baixa do test-chip esta presente. No anexo 10, fotos do test-chip são apresentadas. Não foi possível fotografar o chip completo devido à falta de uma lente apropriada. A área final do test-chip é de 3600x1500 micra quadradas e os testes encontram-se em sua fase inicial no momento em que se escreve este texto.

Para o teste do circuito, diversos padrões foram desenvolvidos em HDC, os quais encontram-se no anexo 5.

6 CONCLUSÃO

Este trabalho se propôs a conceber e implementar um processador integrado CMOS com arquitetura do tipo RISC. Na fase em que se encontra o desenvolvimento do projeto, já obteve-se a concepção de um circuito integrado do tipo processador, a sua simulação completa até o nível de portas lógicas e a implementação dos blocos da parte operativa em um circuito de teste.

Durante a realização desta tarefa, as deficiências e facilidades da cultura de projeto do GME e do CAD disponível tornaram-se patentes.

A utilização de metodologia durante o processo de concepção demonstrou-se fundamental devido à complexidade inerente aos circuitos VLSI do tipo processador. Através deste fato percebeu-se a necessidade de documentar as etapas concluídas, visando construir um histórico do desenvolvimento para que posteriores correções ou melhoramentos no projeto pudessem ser realizados. Esta documentação não se refere a manual de usuário ou similar, mas a um processo de projeto definido por etapas não formais, conforme colocado na metodologia do grupo.

Uma deficiência sentida durante o projeto foi a falta de ferramentas adequadas para especificação e simulação de sistemas em níveis de descrição mais elevados do que o funcional. Um ambiente de CAD com facilidades para descrições VHDL e comportamental, além de um compilador configurável à arquitetura, teriam sido ferramentas úteis para se obter mais segurança quando da especificação da arquitetura, já se prevendo o desempenho da máquina final.

Quanto aos níveis mais baixos de descrição, o ferramental de CAD disponível no CPGCC mostrou-se adequado. Espera-se que com a colocação das ferramentas em um ambiente integrado de projeto - o Silex, por exemplo - a produtividade aumente consideravelmente.

Em relação ao futuro do desenvolvimento do Risco, as próximas etapas são a implementação em silício da parte de controle e a construção de uma placa para acomodar o processador, as memórias e a interface de comunicação com um computador hospedeiro, permitindo que o mesmo seja testado em um ambiente real. Em um futuro mais distante, espera-se a construção de um compilador de LAN para o Risco, já estando disponível um montador de linguagem de máquina.

O Risco criou uma base sólida que permitirá a realização de outros projetos. Uma perspectiva que se desvenda em decorrência, é a utilização do Risco como uma macrocélula em outros sistemas VLSI, havendo a possibilidade para a construção de arquiteturas superpipeline onde ele seria o ponto de partida. Do mesmo modo, a elaboração de uma família de processadores Risco (8 e 16 bits, com unidades funcionais específicas para cada aplicação) é uma alternativa atraente.

Finalmente, cabe citar que a concepção de um CAD para geração automática de processadores com a arquitetura-base do Risco já se encontra em elaboração no GME/CPGCC.