Comparação entre as arquitecturas de processadores RISC e CISC

Luís Filipe Silva[1], Vítor José Marques Antunes[2]

[1] *Email:* ee91163@fe.up.pt
[2] *Email:* ee95070@fe.up.pt

Faculdade de Engenharia da Universidade do Porto

Sumário: Este documento visa ilustrar a relação histórica e técnica entre as arquitecturas de processadores RISC (*Redudec Instruction Set Computer*) e CISC (*Complex Instruction Set Computer*) ao longo do tempo, cobrindo o debate entre as mesmas pelo domínio do mercado desde o seu nascimento até à actualidade. Iremos especificar as arquitecturas, procedendo à sua comparação para chegarmos a conclusões que podem não ser as esperadas, pois cada vez mais o importante é o desempenho "a qualquer preço". Concluí-se que a diferença entre processadores RISC e CISC já não reside no tamanho nem no tipo do conjunto de instruções, mas sim na arquitectura em si, e as nomenclaturas RISC e CISC já não descrevem a realidade das arquitecturas actuais.

1. Introdução

Talvez a abordagem mais comum para a comparação entre RISC e CISC seja a de listar as características de ambas e colocá-las "lado-a-lado" para comparação, discutindo o modo como cada característica ajuda ou não o desempenho. Esta abordagem é correcta se estivermos a comparar duas peças de tecnologia contemporâneas, como os sistemas operativos, placas de vídeo, *CPU*'s específicos, etc., mas ela falha quando aplicada ao nosso debate. Falha porque RISC e CISC não são exactamente tecnologias, são antes estratégias de projecto de *CPU*'s – abordagens para atingir um certo número de objectivos definidos em relação a um certo conjunto de problemas. Ou, para ser um pouco mais abstracto, poderíamos chamar-lhes filosofias de projecto de *CPU*'s, ou maneiras de pensar acerca de um determinado conjunto de problemas e das suas soluções.

É importante olhar para estas duas estratégias como tendo evoluído a partir de um conjunto de condições tecnológicas que existiram num dado momento. Cada uma delas foi uma abordagem ao projecto de máquinas que os projectistas sentiram ser a mais eficiente no uso dos recursos tecnológicos existentes na época. Na formulação e aplicação destas estratégias, os projectistas tomaram em consideração as limitações da tecnologia da altura – limitações essas que não são exactamente as mesmas de hoje. Compreender estas limitações e a forma como os projectistas trabalharam com elas é a chave para perceber os dois tipos de arquitectura. Assim sendo, uma comparação entre as arquitecturas RISC e CISC requer mais do que apenas uma listagem das características, *benchmarks*, etc. de cada uma – requer um contexto histórico.

Para entender o contexto histórico e tecnológico de onde evoluíram as arquitecturas RISC e CISC é necessário, em primeiro lugar, entender o estado das coisas em relação a VLSI, memória/armazenamento e compiladores nos anos 70 e inicio dos anos 80. Estas três tecnologias definiram o ambiente tecnológico no qual os projectistas e investigadores trabalharam para construir as máquinas mais rápidas.

Memória e armazenamento

É difícil subestimar os efeitos que a tecnologia de armazenamento tinha no projecto de um CPU nos anos 70. Nessa altura, os computadores usavam memória de cariz magnético para armazenar o código dos programas, memória que era, não só, cara como também bastante lenta. Depois da introdução da RAM as coisas melhoraram em termos de velocidade, no entanto o seu preço era ainda proibitivo. Apenas a título ilustrativo, em finais dos anos 70, 1 MB de memória RAM podia custar centenas de contos. Em meados dos anos 90, essa mesma quantidade de memória custaria apenas poucos (1-2) milhares de escudos [1]. Adicionado ao preço da memória, o armazenamento secundário era caro e lento, por isso, colocar grandes volumes de código na memória desde o armazenamento secundário era, por si só, um grande impedimento ao desempenho.

O grande custo da memória e a lentidão do armazenamento secundário "conspiraram" para fazer com que a escrita de código fosse um assunto muito sério. O bom código era o compacto já que era necessário colocá-lo todo num pequeno espaço de memória. Como a memória constituía uma parte significativa do preço total do sistema, uma redução no tamanho do código era traduzida directamente numa redução do custo total do sistema.

Compiladores

O trabalho de um compilador era relativamente simples nesta altura: traduzir código escrito numa linguagem de alto nível, como C ou Pascal, em *assembly*. O *assembly* era depois convertido para código máquina por um assemblador. A compilação demorava bastante tempo e o resultado dificilmente se poderia dizer óptimo. O melhor que se poderia esperar era que a tradução da linguagem de alto nível para o *assembly* fosse correcta. Se realmente se quisesse código compacto e optimizado, a única solução era programar em *assembly*.

VLSI

Em termos de VLSI (*Very Large Scale Integration*) a tecnologia da altura apenas permitia densidades de transístores que seriam muito baixas quando comparadas com os *standards* de hoje. Era simplesmente impossível colocar muitas funcionalidades num único *chip*. No início dos anos 80, quando se começou a desenvolver a arquitectura RISC, um milhão de transístores num único *chip* era já bastante [2]. Devido à falta de recursos (transístores) as máquinas CISC da altura tinham as suas unidades funcionais espalhadas por vários *chips*. Isto era um problema por causa do alto tempo de espera nas transferências de dados entre os mesmos, o que desde logo era um óbice ao desempenho. Uma implementação num único *chip* seria o ideal.

2. CISC

No inicio dos anos 70, quer porque os compiladores eram muito pobres e pouco robustos, quer porque a memória era lenta e cara causando sérias limitações no tamanho do código, levou a que uma certa corrente previsse uma crise no *software*. O *hardware* era cada vez mais barato e o *software* cada vez mais caro. Um grande número de investigadores e projectistas defendiam que a única maneira de contornar os grandes problemas que se avizinhavam era mudar a complexidade do (cada vez mais caro) *software* e transportá-la para o (cada vez mais barato) *hardware*. Se houvesse uma função mais comum, que o programador tivesse de escrever vezes sem conta num programa, porque não implementar essa função em *hardware*? Afinal de contas o *hardware* era barato e o tempo do programador não. Esta ideia de mover o fardo da complexidade do *software* para o *hardware* foi a ideia impulsionadora por trás da filosofia CISC, e quase tudo o que um verdadeiro CISC faz tem este objectivo. Alguns investigadores sugeriram que uma maneira de tornar o trabalho dos programadores mais fácil seria fazer com que o código *assembly* se parecesse mais com o código das linguagens de alto nível (C ou Pascal).

Os mais extremistas falavam já de uma arquitectura de computação baseada numa linguagem de alto nível. Este tipo de arquitectura era CISC levado ao extremo. A sua motivação primária era reduzir o custo global do sistema fazendo computadores para os quais fosse mais fácil de programar. Ao simplificar o trabalho dos programadores, pensava-se que os custos seriam mantidos num nível razoável.

Aqui está uma listagem das principais razões para se promover este tipo de arquitectura [3]:

- Reduzir as dificuldades de escrita de compiladores;
- Reduzir o custo global do sistema;
- Reduzir os custos de desenvolvimento de *software*;
- Reduzir drasticamente o *software* do sistema;
- Reduzir a diferença semântica entre linguagens de programação e máquina;
- Fazer com que os programas escritos em linguagens de alto nível corressem mais eficientemente;
- Melhorar a compactação do código;
- Facilitar a detecção e correcção de erros.

Sumariando, se uma instrução complexa escrita numa linguagem de alto nível fosse traduzida em, exactamente, uma instrução *assembly*, então:

- Os compiladores seriam mais fáceis de escrever. Isto pouparia tempo e esforço para os programadores, reduzindo, assim, os custos de desenvolvimento de *software*;
- O código seria mais compacto, o que permitiria poupar em memória, reduzindo o custo global do *hardware* do sistema;
- Seria mais fácil fazer a detecção e correcção de erros o que, de novo, permitiria baixar os custos de desenvolvimento de *software* e de manutenção.

Até este momento, centramos a atenção nas vantagens económicas da arquitectura CISC, ignorando a questão do desempenho. A abordagem utilizada neste tipo de arquitectura para melhorar o desempenho das máquinas CISC foi, conforme já foi referido, transferir a complexidade do *software* para o *hardware*. Para melhor se compreender como é que este tipo de abordagem afecta o desempenho vamos analisar um pouco a seguinte equação – a equação do desempenho de um processador – uma métrica vulgarmente utilizada para avaliar o desempenho de um sistema de computação:

$$\frac{\textbf{Tempo}}{\textbf{Programa}} = \left[\left(\frac{\textbf{Instruções}}{\textbf{Programa}}\right) \times \left(\frac{\textbf{Ciclos}}{\textbf{Instrução}}\right) \times \left(\frac{\textbf{Tempo}}{\textbf{Ciclo}}\right)\right]$$

Melhorar o desempenho significa reduzir o termo à esquerda do sinal de igual (=), porque quanto menor for o tempo que um programa demora a ser executado, melhor será o desempenho do sistema. As máquina CISC tentam atingir este objectivo reduzindo o primeiro termo à direita do sinal de igual, isto é, o número de instruções por programa. Os investigadores pensaram que ao reduzir o número de instruções que a máquina executa para completar uma determinada tarefa, poder-

se-ia reduzir o tempo que ela necessita para completar essa mesma tarefa, aumentando, assim, o seu desempenho.

Assim, ao reduzir o tamanho dos programas conseguiam-se dois propósitos: por um lado era necessária uma menor quantidade de memória para armazenar o código; e por outro o tempo de execução era, também, diminuído pois havia menos linhas de código para executar.

Além de implementar todo o tipo de instruções que faziam um variado número de tarefas como copiar *strings* ou converter valores para BCD entre muitas outras, havia outra táctica que os projectistas utilizavam para reduzir o tamanho do código e a sua complexidade: os modos de endereçamento complexos.

Vejamos o seguinte exemplo, meramente ilustrativo, da multiplicação de dois números armazenados em memória:

A figura seguinte ilustra um esquema de armazenamento para um computador genérico. Se quiséssemos multiplicar dois números, teríamos primeiro que carregar cada um dos operandos de uma localização na memória para um dos registos.

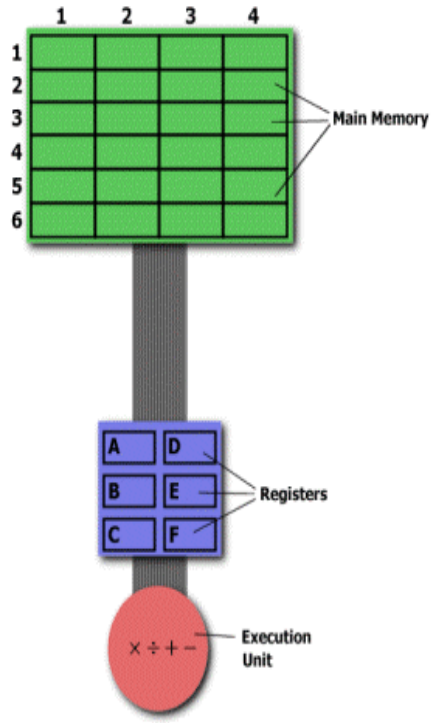

Figura 1. Esquema de armazenamento para um computador genérico.

Uma vez carregados nos registos, os operandos poderiam ser multiplicados pela unidade de execução (ALU – *Aritmetric Logic Unit*). Utilizaríamos as seguintes linhas de código para multiplicar o conteúdo das posições de memória [2:3] e [5:2] e armazenar o resultado na posição [2:3]:

MOV [A, 2:3]

MOV [B, 5:2]

MUL [A, B]

MOV [2:3, A]

Este exemplo de código explicita os passos que têm de ser dados pelo processador para multiplicar os conteúdos de duas posições de memória. Há que carregar os dois registos com o conteúdo da memória principal, multiplicar os dois números e guardar o resultado de novo na memória principal. Se quiséssemos tornar o *assembly* menos complicado e mais compacto, poderíamos modificar a arquitectura por forma a realizar a operação descrita com uma instrução apenas. Para atingir esse objectivo, a instrução MUL teria que ser alterada por forma a aceitar como parâmetros duas posições de memória:

MUL [2:3, 5:2]

Evoluir de quatro instruções para apenas uma é uma grande poupança. Apesar de a "nova arquitectura" ainda ter que carregar o conteúdo das duas posições de memória para os registos, multiplicá-los e voltar a armazenar na memória o resultado – não há como contornar isso – todas essas operações de baixo nível são feitas em *hardware* e são invisíveis para o programador. Este é um exemplo de endereçamento complexo. Uma instrução *assembly*, na realidade leva a cabo uma série complexa de operações. Uma vez mais, isto é um exemplo da filosofia CISC de transferir a complexidade do *software* para o *hardware*.

Outra das características das máquinas CISC era a utilização de micro-código. A micro-programação era mesmo uma das características primordiais que permitia aos projectistas a implementação de instruções complexas em *hardware* [4]. Para melhor compreender o que é a micro-programação, vamos considerar, sumariamente, a sua alternativa: a execução directa. Usando execução directa, a máquina carrega a instrução da memória, descodifica-a e envia-a para a unidade de execução. Esta unidade de execução "pega" na instrução à sua entrada e activa alguns circuitos que levam a cabo a tarefa. Se, por exemplo, a máquina carrega a instrução de ADD com operadores em vírgula flutuante e a fornece à unidade de execução, existe, algures, um circuito que a carrega e direcciona as unidades de execução para garantir que todas as operações de deslocamento, adição e normalização são executadas correctamente. A execução directa é, na realidade, o que se poderia esperar que acontecesse dentro de um computador se não o houvesse conhecimento do micro-código.

A principal vantagem da execução directa é que ela é rápida. Não existe qualquer tipo de abstracção ou tradução extras; a máquina apenas descodifica e executa as instruções em *hardware*. O seu maior problema é que

pode ocupar algum espaço. De facto, se todas as instruções têm que ter um circuito que as execute, então quanto maior for o número de instruções, maior vai ser o espaço ocupado pela unidade de execução. Por isso, executar directamente as instruções não era uma boa abordagem para o projecto de uma máquina CISC. Até porque os recursos (transístores) disponíveis na altura eram bastante escassos.

Chegamos assim à micro-programação. Com a micro-programação, é quase como termos um mini-processador dentro do processador. A unidade de execução é um processador de microcódigo que executa micro-instruções. Os projectistas usam estas micro-instruções para escrever micro-programas que são armazenados numa memória de controlo especial. Quando uma instrução normal de um programa é carregada da memória, descodificada e entregue ao processador de micro-código, este último executa a subrotina de micro-código adequada. Esta subrotina "diz" às várias unidades funcionais o que fazer e como fazer.

No início, o micro-código era lento. Mas ainda assim, a ROM utilizada para a memória de controlo era cerca de 10 vezes mais rápida que a memória magnética da altura, por isso o processador de micro-código ainda conseguia estar suficientemente "avançado" para oferecer um desempenho razoável [4].

Com a evolução da tecnologia, o micro-código estava cada vez mais rápido – os processadores de micro-código nos processadores modernos conseguem velocidades da ordem dos 95% em relação à execução directa. Como o micro-código era cada vez melhor, fazia cada vez mais sentido transferir funcionalidades do *software* para o *hardware*. Assim, os conjuntos de instruções cresceram rapidamente e o número médio de instruções por programa decresceu.

Contudo, à medida que os microprogramas cresceram para fazer face ao crescente número de instruções, alguns problemas começaram a surgir. Para manter um bom desempenho, o micro-código tinha que ser altamente optimizado, eficiente e bastante compacto para que os custos de memória não começassem a crescer desmesuradamente. Como os micro-programas eram, agora, tão grandes, era bastante mais difícil testá-los e detectar e corrigir erros. Como resultado, o micro-código incluído nas máquinas que vinham para o mercado tinha, por vezes, erros e tinha que ser corrigido inúmeras vezes no terreno. Foram estas dificuldades de implementação do micro-código que levaram a que os investigadores questionassem se a implementação de todas estas instruções complexas e elaboradas em micro-código seria, realmente, o melhor caminho para fazer uso dos limitados recursos (transístores) existentes [4].

3. RISC

Como já foi referido, muitas das implementações da arquitectura CISC eram tão complexas que eram distribuídas por vários *chips*. Esta situação não era, por razões óbvias, ideal. Era necessário uma solução num único *chip*, uma solução que fizesse melhor uso dos escassos recurso disponibilizados (transístores). No entanto, para que todo um processador coubesse num só *chip*, algumas das suas funcionalidades teriam que ser deixadas de fora. Nesse intuito, realizaram-se estudos destinados a descobrir que tipos de situações ocorrem mais frequentemente na execução de aplicações. A ideia era descobrir em que tipo de tarefas o processador passava mais tempo e optimizar essas mesmas tarefas. Se tivessem que ser feitos compromissos, estes deviam ser feitos em favor da velocidade de execução das tarefas nas quais o processador passa mais tempo a trabalhar, ainda que isso pudesse atrasar outras tarefas não tão frequentes.

Esta abordagem quantitativa, de fazer mais rápidas as tarefas mais comuns, provocou a inversão da filosofia iniciada pelos CISC e a complexidade teve que ser retirada do *hardware* e ser passada para o *software*. A memória estava a ficar mais barata e os compiladores eram cada vez mais eficientes, por isso muitas das razões que conduziram os projectistas a "complicar" o conjunto de instruções deixaram de existir. Os investigadores diziam que o suporte para as linguagens de alto nível poderia ser mais eficiente se fosse implementada em *software*, gastar recursos (transístores) preciosos para suportar as linguagens de alto nível em *hardware* era um desperdício. Esses recursos poderiam ser utilizados noutras tecnologias para melhorar o desempenho.

Quando os investigadores tiveram que decidir quais as funcionalidades que teriam que ser retiradas, o suporte para o micro-código foi o primeiro a sair, e com ele saíram também um grupo de instruções complexas que, alegadamente, tornava o trabalho dos compiladores e dos programadores mais fácil. A ideia era que quase ninguém utilizava aquelas instruções tão complexas. Os programadores ao escreverem compiladores, com certeza que não as utilizavam pois estas eram difíceis de implementar. Ao compilar o código, os compiladores preteriam este tipo de instruções em favor da geração de um conjunto de instruções mais simples que realizassem a mesma tarefa.

O que os investigadores concluíram dos estudos realizados foi que um pequeno conjunto de instruções estava a fazer a maioria do trabalho. Aquelas instruções que raramente eram usadas poderiam ser eliminadas sem que houvesse perda de qualquer funcionalidade. Esta ideia da redução do conjunto de instruções, deixando de for a todas as instruções que não fossem absolutamente necessárias, substituindo as instruções mais complexas por conjuntos de instruções mais simples, foi o que esteve na origem do termo *Reduced Instruction Set Computer*. Ao incluir apenas um pequeno e criteriosamente escolhido grupo de instruções numa máquina, poder-se-ia deixar de fora o suporte do micro-código e passar a usar a execução directa.

Não só o número de instruções foi reduzido, mas também o tamanho das mesmas. Foi decidido que todas as instruções RISC deveriam, sempre que possível, demorar apenas um ciclo de relógio a terminar a sua execução. A razão por trás desta decisão foi baseada em algumas observações feitas pelos investigadores. Em primeiro lugar, aperceberam-se que tudo o que poderia ser feito usando as instruções de micro-código, também poderia ser feito com pequenas e rápidas instruções de *assembly*. A memória que estava a ser usada para armazenar o micro-código, poderia simplesmente ser usada para armazenar o *assembler*, assim a necessidade de micro-código seria pura e simplesmente eliminada. É por esta razão que muitas das instruções de uma máquina RISC correspondem a micro-instruções numa máquina CISC [4].

De seguida é apresentada uma tabela ilustrativa das diferenças que caracterizaram os primeiros processadores das arquitecturas RISC e CISC ao longo do tempo relativamente ao numero de instruções e a sua relação com o tamanho do micro-código e o tamanho das instruções:

	CISC			**RISC**	
	IBM370	VAX 11/780	8086	SPARC I	MIPS I
Ano	1973	1973	1978	1981	1983
Nº Instr.	208	303	100	39	55
Microc.	54Kb	400Kb	11Kb	0	0
Instr. (bytes)	2-6	2-57	1-17	4	4

Tabela 1 . diferenças que caracterizaram os primeiros processadores das arquitecturas RISC e CISC

A segunda razão que levou a que o formato fosse uniformizado e demorasse apenas um ciclo de relógio foi a observação de que a implementação do *pipelining* só é realmente viável se não tiver que lidar com instruções de diferentes graus de complexidade. Como o *pipelining* permite a execução de várias instruções em paralelo, uma máquina que o suporte consegue reduzir drasticamente o número médio de ciclos por instrução (CPI – *Cycles Per Instruction*). Baixar o número médio de ciclos que as instruções necessitam para terminar a sua execução é uma maneira muito efectiva de baixar o tempo total que é necessário à execução de um programa.

Relembre-mos, de novo a equação do desempenho. Os projectistas deste tipo de arquitectura tentaram reduzir o tempo por programa reduzindo o segundo termo à direita do sinal de igual (ciclos/instrução), permitindo que o primeiro termo (instruções/programa) aumentasse ligeiramente. Pensava-se que uma redução no número de ciclos por instrução, alcançada à custa da redução do conjunto de instruções, a introdução da técnica de *pipelining* e outras funcionalidades (das quais já falaremos) compensariam largamente o aumento do número de instruções por programa. Acabou por se verificar que esta filosofia estava correcta.

Além da técnica de *pipelining*, houve duas inovações importantes que permitiram o decréscimo do número de ciclos por instrução mantendo o aumento do tamanho do código num nível mínimo: a eliminação dos modos de endereçamento complexos e o aumento do número de registos internos do processador. Nas arquitecturas RISC existem apenas operações registo-registo e apenas as instruções LOAD e STORE podem aceder à memória. Poder-se-ia pensar que o uso de LOAD's e STORE's em vez de uma única instrução que operasse na memória iria aumentar o número de instruções de tal modo que o espaço necessário em memória e o desempenho do sistema viriam afectados. Como posteriormente se verificou, existem algumas razões que fazem com que este pensamento não seja correcto. Verificou-se que mais de 80% dos operandos que apareciam num programa eram variáveis escalares locais [2]. Significa isto que, se fossem adicionados múltiplos bancos de registos à arquitectura, estas variáveis locais poderiam ficar armazenadas nos registos, evitando ter que ir à memória de todas as vezes que fosse necessária alguma delas. Assim, sempre que uma subrotina é chamada, todas as variáveis locais são carregadas para um banco de registos, sendo aí mantidas conforme as necessidades.

Esta separação das instruções LOAD e STORE de todas as outras, permite ao compilador "programar" uma operação imediatamente a seguir ao LOAD (por exemplo). Assim, enquanto o processador espera alguns ciclos para os dados serem carregados para o(s) registo(s), pode executar outra tarefa, em vez de ficar parado à espera. Algumas máquinas CISC também tiram partido desta demora nos acessos à memória mas esta funcionalidade tem que ser implementada em micro-código.

Como se pode ver da discussão acima, o papel do compilador no controlo dos acessos à memória é bastante diferente nas máquinas RISC em relação às máquinas CISC. Na arquitectura RISC, o papel do compilador é muito mais proeminente. O sucesso deste tipo de arquitectura depende fortemente da "inteligência" e nível de optimização dos compiladores que se "aproveitam" da maior responsabilidade que lhes é concedida para poderem debitar código mais optimizado. Este acto de transferir o fardo da optimização do código do *hardware* para o compilador foi um dos mais importantes avanços da arquitectura RISC. Como o *hardware* era, agora, mais simples, isto significava que o *software* tinha que absorver alguma da complexidade examinando agressivamente o código e fazendo um uso prudente do pequeno conjunto de instruções e grande número de registos típicos desta arquitectura. Assim, as máquinas RISC dedicavam os seus limitados recursos (transístores) a providenciar um ambiente em que o código poderia ser executado tão

depressa quanto o possível, confiando no compilador para fazer o código compacto e optimizado.

Este tipo de arquitectura é caracterizado, a este nível, por ter um número bastante grande, por comparação com as máquinas CISC, de registos de uso geral.

4. RISC vs CISC

Vamos agora tecer uma breve consideração acerca do estado actual dos três parâmetros que definiram o ambiente tecnológico do qual partiu estudo em questão:

Armazenamento e memória

A memória, hoje em dia, é rápida e barata; qualquer pessoa que tenha instalado recentemente um programa da Microsoft sabe que muitas das companhias que desenvolvem *software* já não têm em consideração as limitações de memória. Assim, as preocupações com o tamanho do código que deram origem ao vasto conjunto de instruções da arquitectura CISC já não existem. De facto, os processadores da era pós-RISC têm conjuntos de instruções cada vez maiores de um tamanho e diversidade sem precedentes, e ninguém pensa duas vezes no efeito que isso provoca no uso da memória.

Compiladores

O desenvolvimento dos compiladores sofreu um tremendo avanço nos últimos anos. De facto, chegou a um ponto tal que a próxima geração de arquitecturas (como o IA-64 ou Merced da Intel) dependem apenas do compilador para ordenar as instruções tendo em vista a máxima taxa de instruções executadas.

Os compiladores RISC tentam manter os operandos em registos por forma a poderem usar simples instruções registo-registo. Os compiladores tradicionais, por outro lado, tentam descobrir o modo de endereçamento ideal e o menor formato de instrução para fazerem os acessos à memória. Em geral, os programadores de compiladores RISC preferem o modelo de execução registo-registo de forma que os compiladores possam manter os operandos que vão ser reutilizados em registos, em vez de repetirem os acessos à memória de cada vez que é necessário um operando. Usam, por isso, LOAD's e STORE's para aceder à memória para que os operandos não sejam, implicitamente, rejeitados após terminada a execução de uma determinada instrução, como acontece nas arquitecturas que utilizam um modelo de execução memória-memória.

VLSI

O número de transístores que "cabem" numa placa de silício é extremamente elevado e com tendência a crescer ainda mais. O problema agora já não é a falta de espaço para armazenar as funcionalidades necessárias, mas o que fazer com todos os transístores disponibilizados. Retirar às arquitecturas as funcionalidades que só raramente são utilizadas já não é uma estratégia moderna de projecto de processadores. De facto, os projectistas procuram afincadamente mais funcionalidades para integrarem nos seus processadores para fazerem uso dos vastos recursos (transístores) disponíveis. Eles procuram não o que podem tirar, mas o que podem incluir. A maioria das funcionalidades pós-RISC são uma consequência directa do aumento do número de transístores disponíveis e da estratégia "incluir se aumentar o desempenho".

Conjunto de Instruções

Uma instrução é um comando codificado em 0's e 1's que leva o processador a fazer algo. Quando um programador escreve um programa em C, por exemplo, o compilador traduz cada linha de código C em uma ou mais instruções do processador. Para que os programadores possam (se quiserem) "ver" estas instruções não tendo que lidar com 0's e 1's as instruções são representadas por mnemónicas – por exemplo MOV, que copia um valor de uma localização para outra ou ADD, que adiciona dois valores. A seguinte linha de código adiciona dois valores (b e c) e coloca o resultado em a:

a=b+c

Um compilador de C poderia traduzir isto na seguinte sequência de instruções:

```
mov     ax, b
add     ax, c
mov     a, ax
```

A primeira instrução copia o conteúdo da localização de memória que contém o valor **b** para o registo **ax** do processador (um registo é uma localização de armazenamento dentro do processador que pode conter uma certa quantidade de dados, normalmente 16 ou 32 bits. Sendo uma parte integrante do processador, os acessos aos registos são muitíssimo mais rápidos que os acessos à memória). A segunda instrução adiciona o valor **c** ao conteúdo de **ax** e a terceira copia o resultado, que está em **ax**, para a localização onde a variável **a** está armazenada. Qualquer programa, por mais complexo que seja, é traduzido, em última análise, em séries de instruções do género da anterior.

Os programas de aplicação modernos contêm, frequentemente, centenas de milhar de linhas de código. Os sistemas operativos são ainda mais complexos: o Microsoft Windows 95 contém cerca de 10 milhões de linhas de código, a maior parte dele escrito em C, e o Windows NT tem mais de 5 milhões de linhas de código escritas em C e C++. Imagine-se o que seria ter de traduzir 1 milhão de linhas de código C num conjunto de instruções com uma a vinte ou trinta instruções por linha de código e é fácil de perceber o porquê de o *software* de hoje em dia ser tão complicado – e tão difícil de corrigir.

Quando um programa corre, o processador carrega as instruções uma a uma e executa-as. Leva tempo a carregar uma instrução e mais tempo ainda a

descodificar essa instrução para determinar o que representam os 0's e 1's. E quando começa a execução, são necessários um determinado número de ciclos de relógio (daqui em diante apenas designados por ciclos) para a completar. Um ciclo é um "batimento" do oscilador que "alimenta" o processador. Num 386 a 25 MHz um ciclo de relógio é igual a 40 ns, num PENTIUM a 120 MHz um ciclo é igual a menos de 9 ns. Uma maneira de fazer com que um processador corra o *software* mais rapidamente é aumentar a velocidade do relógio. Outra é diminuir o número de ciclos que uma instrução requer para completar a execução. Se tudo o resto fosse igual, um processador que funcionasse a 100 MHz teria apenas metade do desempenho de um outro que funcionasse a 50 MHz, se o primeiro requeresse 4 ciclos de relógio por instrução e o segundo apenas um ciclo.

Desde o início da era dos microprocessadores, o grande objectivo dos projectistas de *chips* é desenvolver um *CPU* que requeira apenas 1 ciclo de relógio por instrução – não apenas certas instruções, mas TODAS as instruções. O objectivo original dos projectistas de *chips* RISC era limitar o número de instruções suportadas pelo *chip* de modo a que fosse possível alocar um número suficiente de transístores a cada uma delas, para que a sua execução precisasse apenas de um ciclo de relógio para se completar.

Em vez de disponibilizar uma instrução MUL, por exemplo, o projectista faria com que a instrução ADD executasse em 1 ciclo de relógio. Então o compilador poderia fazer a multiplicação de **a** e **b** somando **a** a ele próprio **b** vezes ou vice versa.

Um CPU CISC poderia multiplicar **10** e **5** da seguinte forma:

```
        mov     ax, 10
        mov     bx, 5
        mul     bx
```

enquanto um CPU RISC faria o mesmo de outro modo:

```
        mov     ax, 0
        mov     bx, 10
        mov     cx, 5
begin:
        add     ax, bx
        loop    begin; loop cx vezes
```

é claro que este é apenas um exemplo ilustrativo, pois os *chips* RISC actuais têm, de facto, instruções de multiplicação.

Comparação de dois tipos de arquitecturas comerciais

O que é que faz do PowerPC um processador RISC e do PENTIUM um processador CISC? Ironicamente, a resposta não tem nada a haver com o tamanho do conjunto de instruções.

Se repararmos nos manuais técnicos dos dois processadores vamos descobrir que os processadores RISC de hoje oferecem um conjunto de instruções mais rico e complexo do que os processadores CISC. Por exemplo, o PowerPC 601 oferece um conjunto de instruções mais alargado do que o PENTIUM, mesmo assim o PowerPC é considerado um processador RISC e o PENTIUM não deixa de ser um processador CISC.

Hoje em dia os processadores de arquitectura RISC já não apresentam as características que eram consideradas como sendo exclusivamente tecnologia RISC, nos primórdios do seu desenvolvimento (início dos anos 70), ou seja o conjunto de instruções na realidade já não é reduzido como era antigamente, o que nos leva a concluir que já não faz sentido este tipo de arquitectura ser denominada RISC (*Reduced instruction Set Computer*).

O que realmente distingue os processadores RISC dos CISC actualmente, está relacionado com a arquitectura em si e não tanto com o conjunto de instruções. Podem ser encontrados alguns pontos chave que caracterizam as diferenças entre um PowerPC e um PENTIUM, das quais são referidas as seguintes[5]:

- Os processadores RISC têm um maior número de registos de uso geral. A melhor maneira de escrever código rápido é maximizar o número de operações executadas directamente no processador e minimizar o número de acessos aos dados guardados na memória RAM. Um maior número de registos de uso geral pode assim facilitar a concretização do objectivo atrás referido, pois os acessos aos registos internos são praticamente instantâneos, enquanto que os acessos à memória RAM demoram algum tempo. Enquanto que um processador PENTIUM apenas possuí 8 registos internos, o PowerPC tem 32.

  Além disso, os processadores RISC têm, também, um número significativo de registos dedicados apenas a operações de vírgula flutuante.

- As instruções que operam directamente na memória requerem inúmeros ciclos de relógio para completarem a sua execução. Este tipo de instruções têm ainda uma implementação a nível de *hardware* que requisita um número excessivamente elevado de transístores. Os processadores RISC têm arquitecturas que minimizam o número de instruções que manipulam dados directamente na memória. Realizam as operações lógicas directamente nos registos internos, e quando chegam ao valor final este é finalmente guardado na memória.

- Uma das características do PowerPC é ter instruções de comprimento fixo, enquanto que no PENTIUM podem variar desde 1 byte até 7 *bytes*, ou mais caso o código de 32 bits esteja a ser

executado num segmento de 16 bits. Os projectistas das arquitecturas RISC preferem projectar todas as instruções do mesmo comprimento, pois simplificam assim a busca de instruções e a lógica de descodificação das mesmas, permitindo que uma instrução inteira possa ser buscada com apenas um acesso à memória de 32 bits.

- Os microprocessadores RISC apresentam uma melhor performance a nível de vírgula flutuante, o que justifica a sua preferência pela comunidade científica. Esta preferência advém da necessidade de desempenho a nível de vírgula flutuante exigida por aplicações de índole científica, que fazem mais cálculos em vírgula flutuante do que aplicações de processamento de texto. Só recentemente, no início dos anos 90, a Intel começou a introduzir a unidade de co-processamento matemático interna em alguns dos seus processadores da família 486. Só posteriormente, com a introdução no mercado dos processadores PENTIUM é que a unidade de co-processamento matemático começou a fazer parte de todos os processadores.

Em geral, os projectistas de arquitecturas RISC tiveram em vista a adopção de tecnologias de ponta como *caches* directamente integradas no processador, *pipelining* das instruções e lógica de predição de modo a obter processadores que conseguissem um melhor desempenho. Mas a Intel incorporou as mesmas tecnologias nos seus processadores, logo é difícil distinguir processadores RISC e CISC tendo como base estas tecnologias.

Na tabela abaixo podem ser observadas as diferenças entre os técnicas de *pipelining* adoptadas pelos projectistas de ambas as arquitecturas:

***Pipelines* para RISC e CISC**	
RISC	CISC
Como possuí mais instruções ortogonais, apresenta uma menor variação da estrutura *pipeline*.	Como possuí instruções mais variadas, apresenta também uma *pipeline* com uma estrutura mais complexa.
A maior parte das instruções RISC são baseadas em operações nos registos internos.	As instruções podem aceder os registos internos ou a memória.
A descodificação das instruções é "trivial".	A descodificação das instruções pode demorar mais do que um ciclo de relógio.

Tabela 2

De qualquer modo ainda existem alguns processadores RISC que se mantêm próximos da linha original dos processadores RISC, por exemplo os processadores Alpha da Digital que combinam conjuntos de instruções reduzidas com enormes caches internas e elevadas velocidades de relógio.

Por outro lado fabricantes de processadores como a AMD, Cyrix, e a NexGen ajudaram ainda mais à fusão das arquitecturas RISC e CISC incorporando nos seus processadores *clones* dos PENTIUM tecnologias adoptadas pelas arquitecturas RISC. Por exemplo o K6 da AMD apresenta um conjunto instruções todas do mesmo comprimento, assim como o PENTIUM II da Intel.

RISC & CISC, lado a lado

Já deve ser aparente que os acrónimos "RISC" e "CISC" apoiam o facto de que ambas as filosofias arquitectónicas têm que ter em conta muito mais do que simplesmente a complexidade ou simplicidade do conjunto de instruções. Qualquer decisão que afecte o custo irá afectar também o desempenho e vice versa. Na tabela abaixo são sumariados os factos apresentados até ao momento ajudando a entender o porquê das decisões tomadas pelos projectistas de processadores.

CISC	**RISC**
Preço/Desempenho	
Preço: mudança da complexidade do *software* para o hardware. **Desempenho**: diminuição do tamanho do código, em troca de um maior CPI*.	**Preço**: mudança da complexidade do hardware para o software. **Desempenho**: diminuição do tamanho do CPI*, em troca de um maior tamanho do código.
Decisões Arquitectónicas	
• Um grande e variado conjunto de instruções que incluí instruções rápidas e simples para executar tarefas básicas assim como complexas e multi-ciclo que correspondem a declarações em HLL**. • Suporte para HLL** é feito em hardware. • Modos de endereçamento memória-memória. • Uma unidade de controlo em micro-código. • Gastar menos	• Instruções simples e de um só ciclo que executam somente funções básicas. Instruções em *assembly* correspondem a instruções em micro-código numa arquitectura CISC. • Todo o suporte HLL** é feito em software. • Modos de endereçamento simples permitem somente que as funções LOAD e STORE acedam à memória. Todas as operações são do tipo registo-registo.

transístores no fabrico dos registos internos.	• Unidade de controlo de execução directa. • Gastar mais transístores em vários bancos de registos. • Uso de execução em *pipeline* para baixar CPI*.

*CPI – *Cycles Per Instruction*

**HLL – *High Level Language*

Tabela 3 . Comparação entre as arquitecturas RISC e CISC.

5. O FUTURO…

A maior ameaça para as arquitecturas RISC e CISC pode não ser nenhuma delas (por oposição à outra), mas uma nova arquitectura denominada EPIC (*Explicit Parallel Instruction Computer*). Como se pode depreender da palavra "paralelo" a arquitectura EPIC pode executar várias instruções em paralelo umas com as outras. Esta filosofia foi criada pela Intel e é, de certa forma, a combinação das arquitecturas RISC e CISC.

A Intel e a Hewlet Packard estão a desenvolver um processador usando esta filosofia sob o nome MERCED (IA-64) e a Microsoft já está a desenvolver uma plataforma (WIN64) para o mesmo. O processador MERCED será um processador de 64 bits.

Se esta arquitectura for bem sucedida poderá tornar-se na maior ameaça à arquitectura RISC. Todas as grandes marcas de fabricantes de processadores, exceptuando a Sun e a Motorola, estão neste momento a comercializar produtos baseados no x86 e alguns estão apenas à espera que o MERCED venha para o mercado. Por causa do mercado dos x86, não é provável que a arquitectura CISC desapareça num futuro próximo, mas a arquitectura RISC poderá vir a ser uma arquitectura em extinção. O futuro poderá trazer-nos processadores baseados na arquitectura EPIC bem como mais famílias de processadores CISC, enquanto que os processadores baseados em arquitecturas RISC poderão tender a desaparecer do mercado.

6. CONCLUSÃO

A diferença entre processadores RISC e CISC já não reside no tamanho nem no tipo do conjunto de instruções, mas sim na arquitectura em si.

As nomenclaturas RISC e CISC já não descrevem a realidade das arquitecturas actuais. O que conta actualmente é a velocidade com que o processador consegue executar as instruções que lhe são passadas e a fiabilidade com que consegue correr o *software*.

Hoje em dia os fabricantes de processadores, sejam eles RISC ou CISC, estão a utilizar todos os truques de modo a melhorarem o desempenho e permitir algum avanço em relação aos seus concorrentes.

Ambas as arquitecturas têm sobrevivido no mercado por razões diferentes, a arquitectura RISC pela sua performance e a arquitectura CISC pela compatibilidade de *software*.

O futuro poderá não trazer a vitória a nenhum deles, mas sim a sua provável extinção, já que a Intel, que sempre foi a empresa líder no fabrico da arquitectura x86 (arquitectura CISC), a vai abandonar em favor da arquitectura RISC depois de ter assinado com a HP para o projecto do Merced. A arquitectura EPIC pode então fazer com que as arquitecturas RISC e CISC se tornem obsoletas.

Referências:

[1] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach, Second Edition" Morgan Kaufmann Publishers, 1996.

[2] David A. Patterson, Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer. 25 years of the international symposia on Computer architecture (selected papers)", 1998.

[3] David R. Ditzel, David A. Patterson, "Retrospective on HLLCA. 25 years of the international symposia on Computer architecture (selected papers)", 1998.

[4] David A. Patterson, "Reduced Instruction Set Computers", Commun. ACM 28, Jan. 1985.

[5]Jeff Duntemann, Ron Pronk, "Inside the PowerPC Revolution", Coriolis Group, 1994.