Capítulo

2

Sistemas Computacionais Embarcados

Luigi Carro e Flávio Rech Wagner

Abstract

This chapter discusses the modeling and design of computational embedded systems, which are nowadays extensively used in products of our everyday's life, such as eletronic equipments and vehicles. Since they involve more complex restrictions than traditional computational systems, such as power limitations, cost, and very tight time-to-market, but without compromising performance, these systems create a new study subject. Architectures, models of computation and software support for the development of such systems are covered in this chapter, and open research subjects are identified.

Resumo

Este capítulo discute a modelagem e o projeto de sistemas computacionais embarcados, encontrados cada vez mais em produtos utilizados nas atividades cotidianas, como equipamentos eletrônicos e veículos. Por envolver restrições mais complexas que um sistema computacional tradicional, como limite de potência, custo e baixíssimo tempo de projeto, mas sem comprometer o desempenho final, estes sistemas são um objeto de estudo per se. Uma revisão das arquiteturas, modelos de computação e suporte de software necessários ao desenvolvimento de tais sistemas é apresentada neste capítulo, assim como são identificados pontos de pesquisa ainda em aberto.

2.1. Introdução

Os sistemas computacionais embarcados estão presentes em praticamente todas as atividades humanas e, com os baixos custos tecnológicos atuais, tendem a aumentar sua presença no cotidiano das pessoas. Exemplos de tais sistemas são os telefones celulares com máquina fotográfica e agenda, o sistema de controle dos carros e ônibus, os computadores portáteis palm-top, os fornos de microondas com controle de temperatura inteligente, as máquinas de lavar e outros eletrodomésticos.

O projeto deste tipo de sistema computacional é extremamente complexo, por envolver conceitos até agora pouco analisados pela computação de propósitos gerais. Por exemplo, as questões da portabilidade e do limite de consumo de potência sem perda de desempenho, a baixa disponibilidade de memória, a necessidade de segurança e confiabilidade, a possibilidade de funcionamento em uma rede maior, e o curto tempo de projeto tornam o desenvolvimento de sistemas computacionais embarcados uma área em si [Wolf 2001].

O projeto de sistemas eletrônicos embarcados enfrenta diversos grandes desafios, pois o espaço de projeto arquitetural a ser explorado é muito vasto. A arquitetura de hardware de um

SoC embarcado pode conter um ou mais processadores, memórias, interfaces para periféricos e blocos dedicados. Os componentes são interligados por uma estrutura de comunicação que pode variar de um barramento a uma rede complexa (NoC – *network-on-chip*) [De Micheli e Benini 2002]. Os processadores podem ser de tipos diversos (RISC, VLIW, DSP, até ASIPs – processadores integrados para aplicações específicas), conforme a aplicação. No caso de sistemas contendo componentes programáveis, o software de aplicação pode ser composto por múltiplos processos, distribuídos entre diferentes processadores e comunicando-se através de mecanismos variados. Um sistema operacional de tempo real (RTOS) [Burns e Wellings 1997], oferecendo serviços como comunicação e escalonamento de processos, pode ser necessário. Além do grande tempo que pode ser gasto com uma exploração sistemática deste espaço de projeto, deve-se considerar ainda o tempo necessário para o projeto e validação individual de todos os componentes dedicados do sistema – processadores, blocos de hardware, rotinas de software, RTOS – assim como o tempo de validação de sua agregação dentro de um mesmo sistema.

Por outro lado, a grande pressão mercadológica num mercado mundial globalizado, somada à contínua evolução tecnológica, impõe às empresas a necessidade de projetarem novos sistemas embarcados dentro de janelas de tempo cada vez mais estreitas, de poucos meses. Além disto, novos produtos têm uma vida cada vez mais curta, de modo que o retorno financeiro de seu projeto deve ser obtido também em poucos meses. Conforme mostrado na Figura 2.1, atrasos de poucas semanas no lançamento de um produto podem comprometer seriamente os ganhos esperados de um novo produto no mercado.

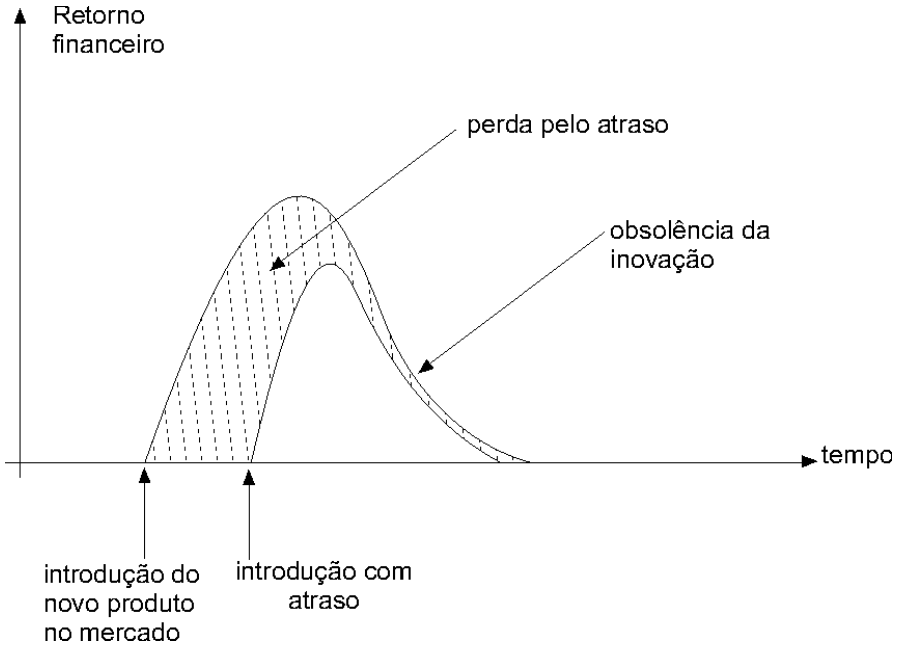


Figura 2.1. Retorno financeiro e janelas de tempo

Um terceiro problema diz respeito aos custos de engenharia não-recorrentes (NRE, em inglês). O projeto de um sistema embarcado de grande complexidade é bastante caro para uma empresa, envolvendo equipes multidisciplinares (hardware digital, hardware analógico, software, teste) e a utilização de ferramentas computacionais de elevado custo. São especialmente elevados os custos de fabricação de sistemas integrados numa pastilha. Um conjunto de máscaras de fabricação está alcançando o custo de um milhão de dólares, o que obriga as

empresas ao projeto de componentes que tenham garantidamente muito alto volume de produção, de forma a amortizar os custos de fabricação.

Em muitas aplicações, é adequada a integração do sistema em uma única pastilha (SoC – *system-on-a-chip*). Em situações onde requisitos de área, potência e desempenho sejam críticos, o projeto do SoC na forma de um ASIC (circuito integrado para aplicação específica) pode ser mandatório, elevando bastante os custos de projeto e fabricação. Em muitas outras situações, no entanto, é mais indicada a implementação do sistema em FPGA, alternativa de customização mais econômica para baixos volumes, ou através de sistemas baseados em famílias de microprocessadores, componentes que são fabricados em grandes volumes e integram milhões de transistores.

Na atual situação de competitividade industrial, seguindo-se a lei de Moore, tem-se à disposição o dobro de transistores a cada 18 meses [Moore 1965]. Consequentemente, sistemas dedicados com milhões de transistores devem ser projetados em poucos meses [Magarshack 2002]. Para isto, tem sido adotado o paradigma de projeto baseado em plataformas [Keutzer 2000]. Uma plataforma é uma arquitetura de hardware e software específica para um domínio de aplicação [Dutta 2001, Demmeler e Giusto 2001, Paulin 1997], mas altamente parametrizável (no número de componentes de cada tipo, na estrutura de comunicação, no tamanho da memória, nos tipos de dispositivos de E/S, etc.). Esta estratégia viabiliza o reuso [Keating e Bricaud 2002] de componentes (ou núcleos) [Bergamaschi 2001] previamente desenvolvidos e testados, o que reduz o tempo de projeto. O reuso pode ser ainda reforçado pela adoção de padrões [VSIA 2003] na arquitetura e projeto dos sistemas.

A Figura 2.2 apresenta um gráfico retirado do *roadmap* ITRS [ITRS 2001], que ilustra que as metodologias tradicionais de projeto têm um custo crescente que não consegue acompanhar a evolução tecnológica permitida pela lei de Moore. Como mostrado na figura, estes custos de projeto podem ser sensivelmente reduzidos pelo reuso de plataformas e componentes.

O projeto de um SoC embarcado consiste então em se encontrar um derivativo da plataforma que atenda aos requisitos da aplicação, como desempenho e consumo de potência. Partindo-se de uma especificação de alto nível da aplicação, é feita uma exploração das soluções arquiteturais possíveis, estimando-se o impacto de diferentes particionamentos de funções entre hardware e software. Feita a configuração da arquitetura, é necessária a síntese da estrutura de comunicação que integrará os componentes de hardware [Lyonnard 2001]. Também é vital a existência de uma metodologia adequada ao teste de sistemas complexos integrados numa única pastilha [Zorian e Marinissen 2000].

Neste estilo de projeto, cada vez mais a inovação de uma aplicação depende do software. Embora a plataforma de hardware de um celular possa ser similar à de um controle de freios ABS, definitivamente o software não é o mesmo. Com a automação do projeto de hardware encaminhada pelo reuso de plataformas e componentes, a automação do projeto do software se torna o principal objetivo a ser alcançado para a diminuição do tempo de projeto, sem sacrifício na qualidade da solução. Esta automação deve idealmente cobrir o software aplicativo, o RTOS [Gauthier 2001], as interfaces entre os processos [Böke 2000] e os acionadores dos periféricos. Também aqui é essencial o reuso de componentes de software [Shandle e Martin 2002] previamente desenvolvidos, de modo que o projeto do sistema embarcado concentre-se apenas na configuração e integração dos mesmos.

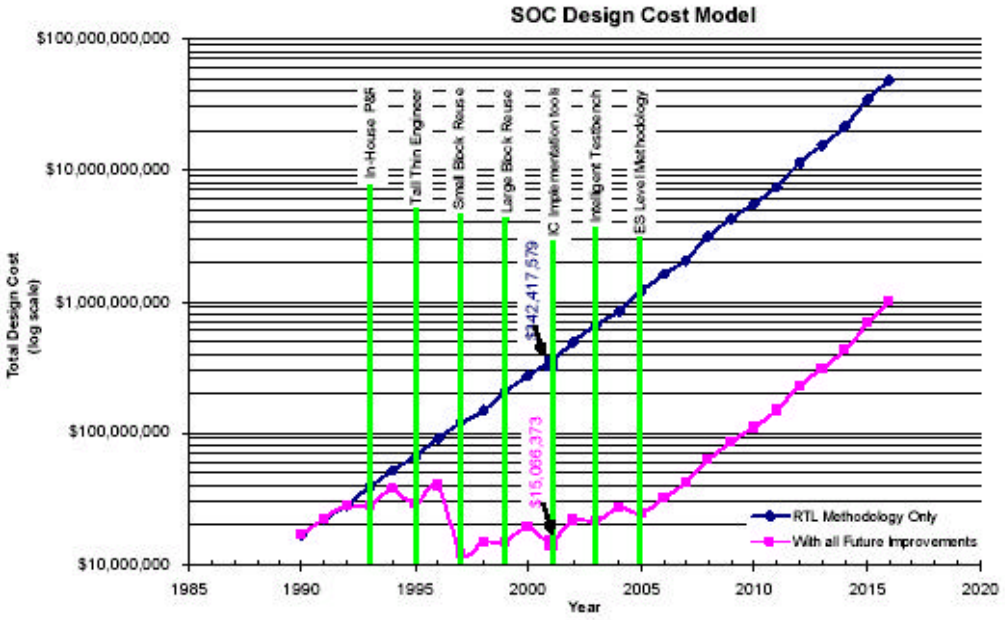


Figura 2.2. Custos no projeto tradicional e no projeto com reuso [ITRS 2001]

2.2. Arquitetura

Nesta seção são revisadas as arquiteturas clássicas de processadores e as tendências modernas, discutindo-se a adequação de cada estilo de projeto para um dado sistema alvo; discute-se também o impacto das memórias em sistemas embarcados, assim como as estruturas de comunicação hoje disponíveis, à luz de seu impacto em futuros sistemas constituídos de milhões de componentes heterogêneos.

2.2.1. Microprocessadores e seu espaço de projeto

O projeto de sistemas embarcados toma sempre como base um ou mais processadores. Embora esta solução pareça extremamente conservadora do ponto de vista de inovação, ela traz enormes vantagens do ponto de vista operacional. Primeiro, o fator de escala. Como os microprocessadores são encontrados em milhares de projetos, seu custo dilui-se entre muitos clientes, as vezes até competidores entre si. Mais ainda, uma vez que uma plataforma baseada em processador esteja disponível dentro de uma empresa, novas versões de produtos podem ser feitas pela alteração do software da plataforma. A personalização do sistema dá-se através do software de aplicação, que toma atualmente a maior parte do tempo de projeto. Além destas vantagens competitivas, há ainda o fator treinamento de engenheiros, já que estes geralmente se formam com conhecimentos de programação de microprocessadores.

É claro que o projeto usando microprocessadores não é vantajoso em todo aspecto. A questão da potência, cada vez mais valorizada nos tempos atuais, é crítica. Como são projetados para executar qualquer programa, existem estruturas de hardware dentro dos processadores que consomem muitos recursos, mas que são muitas vezes sub-utilizadas. Por exemplo, o sistema de predição de saltos de processadores complexos tem um custo de área e potência enorme para aplicações tipo processamento de sinais, onde um preditor mais simples mas eficiente pode ser feito em tempo de compilação. Também para este tipo de aplicações, caches tornam-se

extremamente ineficientes, já que os dados são consumidos muito rapidamente, sem obedecer ao princípo da localidade espacial ou temporal.

Para uma aplicação extremamente específica, um projeto usando lógica programável como FPGAs ou ASICs pode ter um desempenho muito melhor que usando um processador, ou mesmo uma potência mais baixa. O problema é que sistemas reais possuem diversos comportamentos (modelos de computação) e o atendimento simultâneo dos mesmos tende a diminuir o desempenho do hardware dedicado. Além disto, é preciso considerar que se encontram processadores nas mais diversas combinações de preço, desempenho e potência. Os processadores também contam com grupos de projeto imensos, que chegam às centenas de projetistas, e com tecnologias do estado-da-arte para sua fabricação. Tudo isto torna o uso de processadores extremamente interessante para o projeto de sistemas embarcados.

Há contudo uma série de questões a serem respondidas, mesmo que o projetista decida usar um microprocessador. Por exemplo, se o projeto tem limitações de potência, famílias de processadores que trabalham com freqüências mais baixas podem não possuir desempenho suficiente. Nestas situações, é preciso escolher processadores com controle de potência embutido, onde partes do processador possam ser desligadas, ou utilizar técnicas como processadores e multiprocessadores de múltipla voltagem [Zhang 2002]. Uma alternativa mais interessante é escolher a arquitetura adequada para o projeto em questão, pois os ganhos em potência e desempenho podem ser maiores, conforme será discutido a seguir.

Para um projeto embarcado, não só a CPU é importante. O quanto de memória estará à disposição impacta a potência do sistema (memórias rápidas são grande fonte de consumo) e a sua possibilidade de reuso, já que pouca memória limita expansões, ainda obrigando a codificação em assembler, sem muito espaço para o uso de compiladores. Isto, por sua vez, aumenta o tempo de projeto, variável sempre crítica.

2.2.2. Arquiteturas clássicas e tendências modernas

Explorar o paralelismo possível é a solução natural para se diminuir o tempo de execução de um certo programa. Contudo, devido a seus custos, as primeiras versões de paralelismo dentro de um processador foram limitadas ao uso de pipeline, ou seja, à execução simultânea de estágios distintos de instruções diversas. A Figura 2.3 ilustra esta situação, onde é executado um pipeline de 5 estágios em uma máquina de registradores, assim dividido: B - busca instrução; D - decodifica instrução e lê os operandos; O - opera sobre os registradores, M - escreve ou lê da memória; e finalmente E - escreve de volta no banco de registradores.

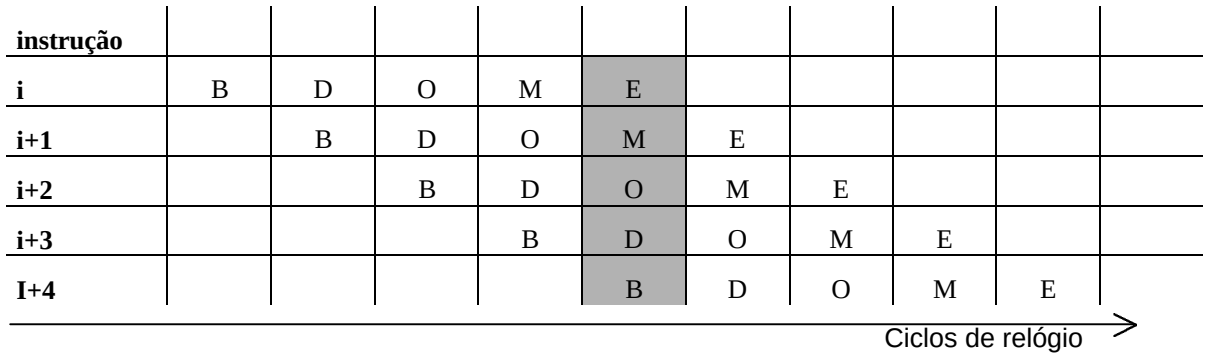


Figura 2.3. Pipeline clássico

A máquina da Figura 2.3 executa bem instruções do tipo *rd = rf1 op rf2*, onde rd é o registrador destino, rf1 e rf2 são os registradores fonte da operação, e op é uma operação lógica ou aritmética. As memórias de programa e de dados encontram-se separadas, sendo que esta última pode ser acessada em instruções do tipo carga (*rd = MEM [rf1 op rf2]*) ou armazenamento (*MEM [rd] = rf1 op rf2*).

O máximo desempenho da arquitetura proposta na Figura 2.3 corresponde à execução de 5 instruções ao mesmo tempo, e a cada ciclo uma instrução é completada. Infelizmente, nem todas as instruções necessárias ao funcionamento de um programa podem ser assim executadas. Existem as dependências de dados, como se pode observar na Figura 2.4.

Instrução	(a) código original	(b) código alterado
I	ld r6, 36(r2)	ld r6, 36(r2)
I+1	add r5, r6, r4	sub r9, r12, r8
I+2	sub r9, r12, r8	add r5, r6, r4
I+3	st r5, 200(r6)	add r3, r9, r9
I+4	add r3, r9, r9	st r5, 200(r6)
I+5	and r11, r5, r6	and r11, r5, r6

Figura 2.4. Dependências de dados

As dependências de dados podem ser resolvidas de diversas maneiras, por exemplo através do compilador, pela reodernação da sequência de instruções, ou através do uso de um mecanismo de detecção de dependência e antecipação do cálculo. Embora a reordenação através do compilador implique em custo zero de hardware, nem sempre o compilador encontra instruções suficientes para serem trocadas de lugar. Consequentemente, praticamente todas as máquinas possuem estruturas de *forwarding*, isto é, estruturas para antecipar os operandos em caso de dependência. Por exemplo, o trecho da Figura 2.4 (a) possui uma dependência de dados pois o operando que deve ser lido no estágio D da instrução i+3 (registrador r6), calculado pela instrução i e escrito nos respectivo estágio E, ainda não está disponível. Neste caso, uma troca de ordem de instruções pode resolver a dependência, conforme sugerido na Figura 2.4 (b), ou um multiplexador na entrada da unidade aritmética é necessário para redirecionar a saída da própria unidade à sua entrada, economizando-se os ciclos de relógio que seriam gastos à espera da leitura do dado escrito no registrador.

Instruções de salto condicional ou incondicional também apresentam problemas para a boa sequência do pipeline. Uma instrução do tipo *bne r3,r2,r1* deve carregar o contador de programa PC com o conteúdo de r3, se os registradores r2 e r1 não forem iguais. O problema advém do fato que somente no final do terceiro estágio a condição de salto estará disponível. Neste meio tempo, mais instruções já estão no pipeline, e portanto, no caso de um salto, deverão ser descartadas. Pode-se contornar o problema pela inserção de uma bolha, isto é, uma interrupção parcial no pipeline, até que as dependências estejam resolvidas, ou promover a limpeza do pipeline, isto é, a remoção de todas as instruções que haviam entrado no pipeline depois da instrução de salto. Ambas as soluções podem parecer pouco custosas, ainda mais em se tratando de um pipeline de poucos estágios como o apresentado. Contudo, para programas teste de aplicações escalares, a cada 5 instruções em média tem-se um salto [Hennessy 1996]. Consequentemente, o pipeline seria constantemente esvaziado, e tendo-se em conta o tempo

necessário para que de novo enchesse, a média de instruções executadas em paralelo cairia para 2 ou 3. O problema torna-se ainda maior para arquiteturas com vários estágios de pipeline, como 10 ou 12, o que é muito comum em processadores de alto desempenho.

A solução para o problema da dependência em saltos que esvaziam o pipeline é obtida através de um mecanismo de predição de saltos. Basicamente, pode-se apostar que, em um programa contendo muitos laços, um salto será sempre tomado, por exemplo. Então, ao invés de carregar no pipeline as instruções que seguem o salto, carrregam-se aquelas no destino do salto. Este mecanismo funcionaria bem para programas compostos apenas de grandes laços, mas é evidente que, para programas de propósito geral, a solução peca pela excessiva simplicidade. Uma máquina de predição de saltos mais refinada encontra-se na Figura 2.5. A predição somente muda sua saída caso cometa erros duas vezes. Desta maneira, os arquitetos de computadores estão tendo benefícios da estatística de execução de um programa, prevendo o seu comportamento futuro como uma reprodução do passado. Para que a máquina de estados da Figura 2.5 funcione, é preciso saber em que ponto o contador de programa se encontra, e necessita-se de uma memória extra para guardar o valor da predição e o endereço destino. Este hardware extra pode ser facilmente suportado pela enorme quantidade de transistores disponíveis para fabricação, mas o preço a pagar é a dissipação extra de potência e a área maior do processador.

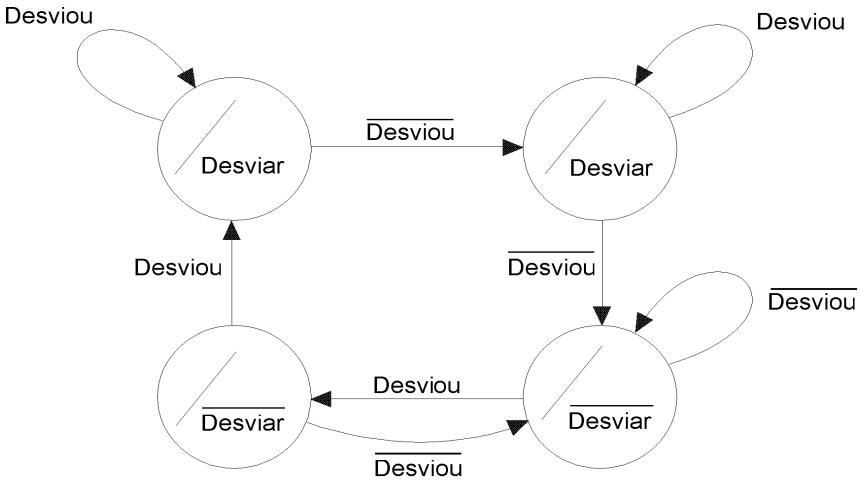


Figura 2.5. Predição de saltos através de máquina de estados

Pode-se aumentar ainda mais o desempenho do processador pela adoção do paralelismo explícito, por exemplo pela execução de mais instruções em paralelo através de arquiteturas superescalares, como se pode ver na Figura 2.6, onde uma máquina com pipeline de 5 estágios possui superescalaridade 2, isto é, duas máquinas iguais trabalham em paralelo.

Infelizmente, programas de vida real nem sempre possuem muito paralelismo a ser explorado. Na verdade, técnicas de compiladores como desenrolamento de laços (*loop-unrolling*) podem aumentar o paralelismo disponível, mas o preço a pagar é muitas vezes o aumento da dependência de dados. Os arquitetos de processadores rapidamente reagiram, através do desenvolvimento de mecanismos de resolução de dependências em hardware. Os principais são a execução fora de ordem de instruções, junto com a renomeação de registradores. O primeiro mecanismo busca, numa fila de instruções, todas aquelas cujas dependências de dados estejam resolvidas. Desta maneira, não é necessária a intervenção do

compilador. A segunda técnica procura esconder a dependência de dados, pois muitas vezes um registrador é escrito para logo depois ser re-escrito, de maneira que um registrador *alias* pode ser usado antes de seu conteúdo ser efetivamente destruído.

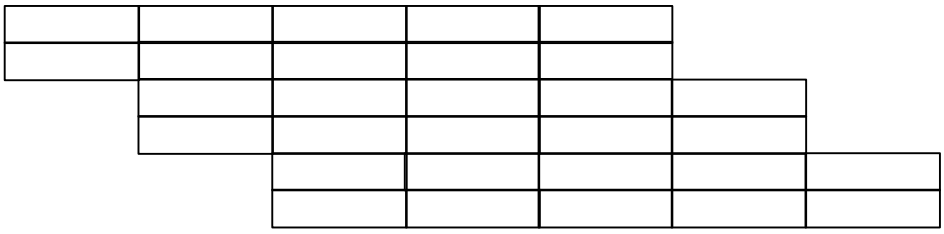


Figura 2.6. Arquiteturas superescalares com pipeline

As máquinas superescalares são as mais utilizadas hoje em dia em processadores de uso geral. A grande vantagem destas arquiteturas é a descoberta do máximo de paralelismo disponível sem intervenção do programador ou compilador. O maior problema é o excesso de potência, já que os mecanismos de predição de saltos, fila de instruções, execução fora de ordem e renomeação de registradores trabalham apenas para manter as unidades aritméticas do processador ocupadas a maior parte do tempo.

Uma alternativa às máquinas superescalares são as máquinas VLIW (Very Large Instruction Word). Nestas máquinas a palavra de instrução controla todos os bits da parte operativa, como exemplificado na Figura 2.7. As unidades operativas possuem registradores próprios, e o paralelismo máximo é decidido a priori, em tempo de compilação.

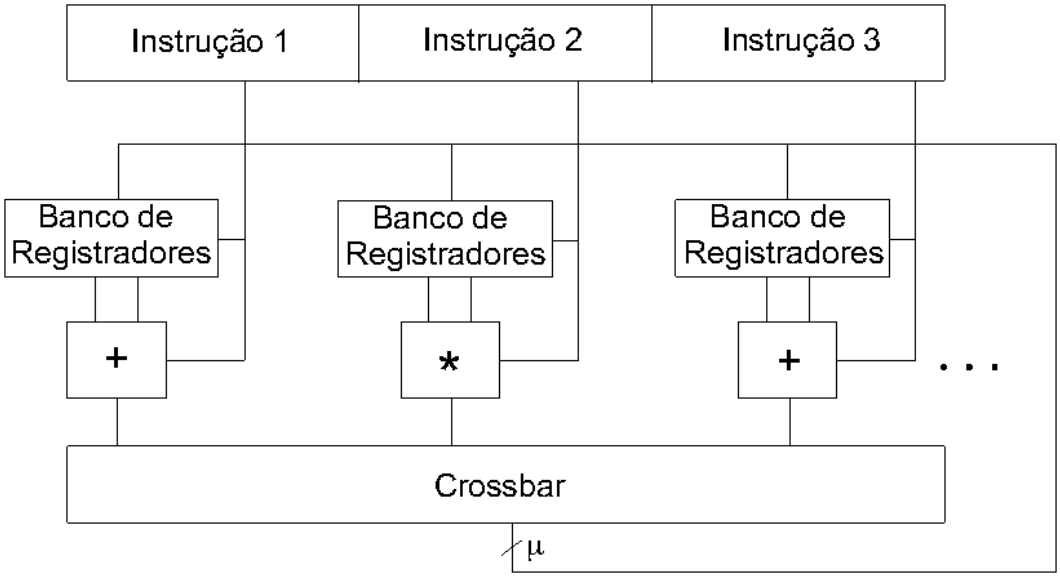


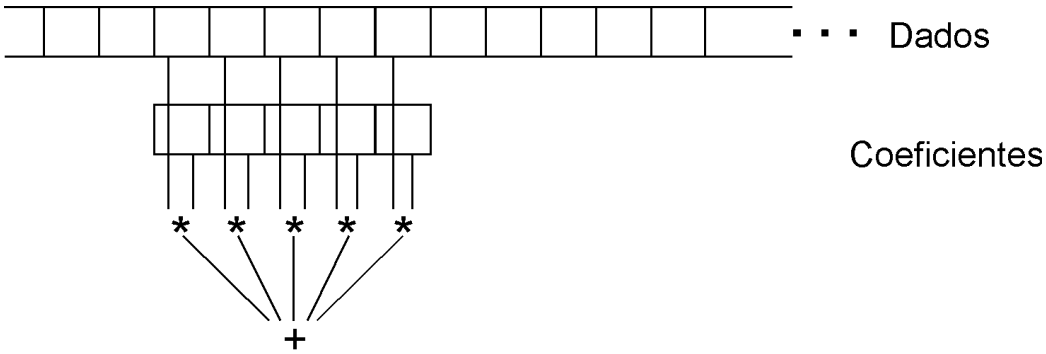
Figura 2.7. Máquina VLIW

Apesar de não possuir todo o sistema de predição de saltos e de detecção de paralelismo no fluxo corrente de instruções, e portanto ser potencialmente mais econômica que uma máquina superescalar em termos de consumo de potência, uma desvantagem da máquina VLIW é que, devido à dependência de dados, muitas unidades podem ficar esperando instruções, e portanto a memória extra de instruções está sendo desperdiçada junto com as unidades operativas. As máquinas VLIW, por outro lado, dependem drasticamente de um compilador eficiente, e são muito mais interessantes se a quantidade de paralelismo puder ser descoberta a priori.

2.2.3. Modificações arquiteturais para suporte ao processamento digital de sinais

Presentemente, muitos sistemas embarcados precisam possuir a capacidade de processar sinais digitalmente. Um exemplo clássico é o telefone celular, onde quase todo o processamento das informações que chegam e saem da antena é feito no domínio digital, para permitir maior repetitibilidade e facilidade de projeto em relação ao processamento analógico. Arquiteturas para processamento digital de sinais tornaram-se muito populares na última década e, impulsionadas pelo mercado de modems e outros equipamentos de comunicação, chegam ao mercado sob forma de processadores especializados para execução de algoritmos específicos de processamento de sinais, com alto desempenho e baixo custo de potência.

Para melhor discutir as otimizações arquiteturais, é importante entender a família de algoritmos para a qual os processadores DSP se adaptaram. Na Figura 2.8 tem-se a realização de um filtro digital. A memória de dados (barra longa) deve ser varrida, e para cada posição tem-se um conjunto de coeficientes que deve ser multiplicado a cada dado e acumulado para produção de uma única saída. É patente o grande número de operações aritméticas.



$$y(n) = \sum_{k=0}^{M-1} c_k\, x(n-k)$$

Figura 2.8. Filtro FIR

Como são feitas tantas tantas multiplicações e somas quanto o tamanho do filtro, ao invés de se utilizar duas instruções (multiplicação seguida de acumulação) e os vários registradores envolvidos, a estratégia mais interessante é o uso de uma única instrução, a instrução MAC (multiplica e acumula).

Contrariamente ao mostrado na Figura 2.8, muitas operações de processamento de sinais raramente são feitas em *batch*, mas sim tem de responder em tempo real. Isto significa que os dados a serem processados estão sempre chegando, a uma taxa fixa. A memória utilizada não será infinita, e em algum momento os dados que chegam deverão ocupar a posição daqueles já utilizados e não mais necessários. Isto pode ser visto na Figura 2.9, que mostra o código Java de um buffer circular.

É evidente que a gerência deste buffer consome muitas instruções, que servem apenas para otimizar o acesso à memória. Em hardware, um buffer circular é apenas um contador de módulo, isto é, um contador que, ao chegar ao seu limite, volta a seu endereço inicial e prossegue incrementando. Praticamente todos os processadores DSP possuem esta otimização em hardware. É interessante observar que esta otimização geralmente só está disponível em assembler, já que é difícil construir-se um compilador apto a reconhecer um buffer circular.

Pela própria natureza dos algoritmos de filtragem, o acesso à memória é um grande gargalo. Para um filtro básico, como o da Figura 2.8 (filtro FIR) são necessários, para cada multiplicação, um acesso aos coeficientes do filtro e um acesso aos dados. Consequentemente, muitos processadores possuem bancos de memórias separados, para que o acesso possa ser feito simultaneamente em cada banco. Além disto, muitos algoritmos, como a transformada rápida de Fourier (FFT) e a transformada discreta do cosseno (DCT), utilizam modos de endereçamento não ortodoxos, como endereçamento de bit-reverso, onde uma parcela dos bits menos significativos do endereço deve ter sua ordem revertida. Em software esta é uma operação custosa, pois são muitas as instruções necessárias em comparação com aquelas úteis ao laço. Já em hardware, o bit-reverso significa apenas uma inversão de fios a custo próximo de zero.

```
1  Public static void initSystem() {
2        Entry = 0;
3        size = coef.length;
4        for (int i=0;i<51;i++) coef[i] = coef1[i];
5        while(true){  // infinite loop through inputs
6              //write new input to the buffer
7              buffer[entry] = FemtoJavaIO.read(0);
8              //update buffer pointer
9              if (entry<(size-1)) entry++;
10             else entry = 0;
11             //reset sum to make a new output
12             sum = 0;
13             for (j=0;j<size; j++) {//coefficient control loop
14                   if (entry+j>(size-1)) {
15                         sum = sum + (coef[j]*buffer[entry+j-size]);}
16                   else {
17                         sum = sum + (coef[j]*buffer[entry+j]);}
18             }//end for
19             FemtoJavaIO.write( sum , 1 );
20             i++;
21       }//end while
22 }//end initSystem
```

Figura 2.9. Buffer circular

Como a maior parte dos algoritmos de processamento digital de sinais é baseada em laços curtos, com poucas instruções dentro do laço, o controle do laço em software significa instruções extras que pesam no total de instruções a serem executadas no laço. Ao prover-se controle de laços em hardware, estas instruções extras são eliminadas, com grande aceleração do algoritmo a um mínimo custo de hardware.

A cada vez que uma interrupção deve ser atendida, o status do processador deve ser salvo, o que significa um longo tempo de salvamento de registradores e status do pipeline do processador. Em aplicações DSP, muitas vezes a interrupção apenas sinaliza que um novo dado chegou, e que uma simples leitura de uma porta do processador e uma escrita do dado na memória serão suficientes. Processadores DSP possuem então a possibilidade do programador escolher interrupções mais simples, cujo tempo de atendimento é de realmente poucos ciclos, sem envolver pesado chaveamento de contexto.

Muitas das funções de processamento digital de sinais agora estão sendo migradas para arquiteturas convencionais, pois os benchmarks utilizados para projeto destas arquiteturas antes não traziam este tipo de problema. Nos sistemas embarcados, módulos com co-processadores DSP ou com processadores VLIW adaptados ao processamento digital de sinais são cada vez

mais disponíveis [Madisseti 1995, Lapsley 1997, Texas 2002]. Krapf (2002, 2003) sugere que o uso de algumas das otimizações acima mencionadas conseguiu acelerar o processamento de sinais em Java em até 50%, com um custo de área extra de apenas 5% no processador.

2.2.4. Hierarquia de memórias

Uma vez resolvido o problema do paralelismo em processadores, o gargalo passa a outro ponto. Neste caso, o sistema de memórias é crítico. Por problemas de fabricação, não podem existir memórias ao mesmo tempo rápidas e de grande capacidade. Memórias estáticas podem ser rápidas, funcionando no mesmo ciclo de relógio da CPU (e portanto, dentro do pipeline), mas apenas se seu tamanho for bem limitado. Além disto, por envolverem grandes capacitâncias parasitas de linhas de bit, memórias rápidas tendem a possuir um enorme consumo de potência, o que somente agrava o problema.

Para resolver o problema de velocidade de memória próxima à da CPU, há muitos anos os arquitetos inventaram as memórias cache. A idéia surgiu na época em que as memórias de núcleo de ferrite de grande capacidade eram lentas, enquanto que a tecnologia de fabricação de semicondutores não permitia a integração de um grande número de bits de memória. As caches eram memórias rápidas, fabricadas no mesmo processo da CPU, mas que obviamente possuiam uma capacidade limitada de armazenamento. Fortunadamente, para a maioria dos programas aplicam-se a localidade temporal (uma posição visitada será visitada novamente em pouco tempo), ótima para laços, e a localidade espacial (a posição visitada a seguir será próxima da atual), que permitem o reuso intenso dos dados e instruções armazenados na cache..

Com o avanço das tecnologias de memória, podem-se fabricar memórias de grande capacidade (as memórias dinâmicas atuais de muitos Mbits), mas infelizmente várias ordens de grandeza mais lentas que as CPUs. Consequentemente, quase todos os processadores utilizam caches hoje em dia.

No domínio de sistemas embarcados, as caches têm seu uso questionado. Embora efetivamente permitam ao processador aumentar seu desempenho, a gerência das caches e seu consumo são pontos muito desfavoráveis, já que a questão da portabilidade está sempre presente. Outro aspecto desfavorável ao uso de caches é que as memórias grandes são dinâmicas, o que torna seu custo baixo em termos de área, mas alto em termos de potência gasta para um acesso e recuperação do estado da cache. Existe ainda a dificuldade de predição do tempo de resposta em caso de sistemas de tempo real, já que a gerência da cache e das memórias dinâmicas pode tornar a previsão excessivamente pessimista, implicando em potência extra. Por outro lado, os avanços tecnológicos são rápidos, e usar uma memória lenta implica em não tomar proveito do estado da arte da tecnologia.

Nem sempre as caches auxiliam a execução rápida de um programa. Quando as regras de localidade não se aplicam, as caches estarão consumindo potência sem nada contribuir. Um exemplo típico é a filtragem de imagens, onde um dado carregado na cache pode ser usado em poucas computações e depois removido, pois somente será novamente instanciado na próxima varredura. Isto provoca um contínuo esvaziamento da cache, aumentando a latência do algoritmo e a potência consumida.

Por serem memórias mais rápidas, as caches consomem enorme quantidade de energia, chegando a 75% da energia de um processador [Kin 2000]. Diversos trabalhos tentam diminuir o impacto de potência das caches através, basicamente, da redução do tamanho da cache quanto

mais próxima ela esteja da CPU, confiando que a localidade é maior para aplicações modernas [Herbert 2000, Tomiyama 98, Kin 2000, Shiue e Chakrabarty 1999].

No domínio dos sistemas embarcados, alguns trabalhos pressupõem que o uso de caches imporá uma restrição séria de consumo. Assim, vários trabalhos otimizam técnicas para redução de consumo, como a compressão de instruções, na esperança que o número de *misses* na cache diminua [Lekatsas e Wolf 1999, Benini, 1999]. Outra alternativa é o agrupamento de instruções, transformando um processador RISC em um CISC, para diminuir a quantidade de acessos à memória, como proposto por Ishihara e Yasuura (2000).

Soluções alternativas às caches também têm sido propostas, por exemplo, através da colocação de código muito usado em uma memória dedicada junto ao processador [Benini 2000]. A questão da execução mais rápida do código crítico é resolvida, ao mesmo tempo em que o controle da cache e sua potência excessiva ficam minimizados. Uma pequena memória rápida de 1024 posições poderia ter uma taxa de acessos chegando a 75% do total de acessos necessários para executar um decodificador MP3. Para este mesmo tamanho de memória, a economia de potência chegou a 44% em comparação com uma cache de mesmo tamanho, com política *write-through* de escrita.

Presentemente, as aplicações embarcadas utilizam quantidades de memória impensáveis há pouco tempo, como por exemplo nas câmeras fotográficas digitais. Estas memórias são baseadas em tecnologia EEPROM, isto é, são memórias estáticas não voláteis. São contudo lentas, o que significa que não podem ser usadas para processamento puro, pois são ordens de grandeza mais lentas que as estáticas. O avanço tecnológico porém não está paralisado, e outras memórias como a FRAM (*ferromagnetic* RAM) já são realidade comercial [Inomata 2001]. Estas memórias possuem a velocidade e a densidade das memórias DRAM, mas são estáticas, o que significa muito menos potência gasta, pois não há necessidade de *refresh*. Ainda, elas possuem a enorme vantagem de poderem ser fabricadas no mesmo processo que as CPUs convencionais, como uma SRAM. Portanto, o uso de caches para manter a velocidade da CPU alta pode ser pensado quando estas novas memórias estiverem sendo usadas para memória de massa de sistemas embarcados.

2.2.5. Estruturas de comunicação: barramentos e redes em um chip

Até o final desta década os projetistas estarão trabalhando com sistemas em silício (SoCs) de até 4 bilhões de transistores, usando um processo com transistores de 50-nm de canal [De Micheli e Benini 2002]. É evidente que esta abundância de transistores permitirá a integração de centenas de núcleos de propriedade intelectual, cada um com sua centena de milhares de transistores per se. Contudo, pela quantidade de elementos envolvidos e pelo custo das máscaras de fabricação para estas tecnologias, é provável que o reuso de blocos seja mandatório.

Comunicações ponto-a-ponto são aquelas mais rápidas, pois o tamanho do fio e o número de bits usados na comunicação são projetados sob medida. Por outro lado, esta forma de comunicação é a menos reusável de todas, já que a cada projeto um novo conjunto de terminais deve ser analisado. Quando centenas de núcleos tem de se comunicar, o projeto da comunicação do sistema tende a ser problemático.

Barramentos são reusáveis, mas permitem apenas uma transação por ciclo, o que serializa completamente a comunicação. Como todos os núcleos do sistema estarão conectados ao mesmo barramento, e este deve passar por todo o circuito integrado, a capacitância de carga (devida aos núcleos e ao comprimento do fio) será elevada, limitando excessivamente a máxima

frequência que se poderia obter com a comunicação. Alguns destes problemas foram resolvidos pelo uso de barramentos hierárquicos como o Core Connect [IBM 2003] e o AMBA [ARM 1999]. Nestas soluções, porém, o máximo paralelismo é restrito, e uma comunicação entre núcleos mapeados para sub-barramentos diferentes provocará a paralisação de diversos recursos de comunicação ao mesmo tempo. Este modelo de comunicação é adequado a um processador central com vários periféricos, não a um modelo com vários processadores complexos executando diferentes tarefas.

Embora já existam hoje conceitos de reuso quanto à funcionalidade (processadores, memórias, processadores DSP, tocadores de MP3 e outros blocos), os recursos de comunicação somente recentemente mereceram atenção da comunidade internacional. Devido à complexidade das funções, à heterogeneidade dos blocos e à multitude de tarefas que estes SoCs realizarão, é provável que o modelo de computação seja distribuído, isto é, não mais centrado em um processador e periféricos. Exemplos de tais sistemas já existem hoje na área de entretenimento [Paulin 1997, Dutta 2001].

Nesta situação, a comunicação entre os diferentes blocos não pode ter o mesmo paradigma atual (barramentos com ou sem prioridade e multi-nível). Primeiro, pela escalabilidade. Ao agregar-se mais um núcleo de hardware a um conjunto já existente, o reuso do mesmo barramento implicará em:

- maior consumo, pelo aumento da capacitância parasita;
- menor frequência disponível para comunicação, também pelo aumento da capacitância parasita;
- menor taxa de comunicação, pelo efeito combinado da diminuição de banda (no barramento, quando um módulo fala os outros devem escutar) e pela diminuição de frequência.

Além de se manter a comunicação em altas taxas e escalável, deseja-se que os mecanismos de comunicação sejam também eles reusáveis [Zeferino 2002a]. Portanto, os mecanismos de comunicação no futuro deverão ser escaláveis, oferecer paralelismo para que vários núcleos conversem ao mesmo tempo, e reusáveis, para diminuição do custo de desenvolvimento.

Os mecanismos de comunicação baseados em comunicação ponto-a-ponto não são genéricos, tendo de ser redimensionados a cada nova versão do sistema. Barramentos, por outro lado, provêm um padrão, mas não o paralelismo e a escalabilidade necessários [Guerrier e Greiner 2000, Dally e Towles 2001]. Para atender aos requisitos dos SoCs feitos com centenas de núcleos, alguns trabalhos recentes propõem o uso de uma rede em um chip (NoC – *Network-on-Chip*) [De Micheli e Benini 2002]. As origens destes trabalhos encontram-se nas redes de comunicação de arquiteturas paralelas.

Aparentemente, Guerrier e Greiner (2000) foram dos primeiros a propor o uso de uma rede de chaveamento para comunicação dentro de um circuito integrado. A rede proposta (Spin – *Scalable programmable interconnection Network*) é chaveada a pacotes e tem arquitetura de árvore gorda.

Uma NoC é composta por um conjunto de canais e de roteadores, que podem ser vistos como núcleos do SoC dedicados à comunicação. Cada roteador está conectado a um ou mais núcleos, e possui um conjunto de canais para se comunicar com outros roteadores. A Figura 2.10 apresenta três topologias bastante utilizadas de NoCs, onde os parâmetros acima também

podem ser variáveis. A mensagem deverá passar por tantos roteadores quantos existirem no caminho entre o núcleo origem e o núcleo destino, pelo menos. Como cada roteador possui um tempo diferente de zero para processar a mensagem e decidir o caminho a ser usado, quanto mais longe um núcleo estiver de outro, maior o tempo de comunicação envolvido. Este aparente prejuízo é largamente compensado por outros fatores:

- a NoC pode transmitir uma mensagem numa freqüência maior, pois a capacitância parasita das conexões é menor que a do barramento. Como as conexões entre roteadores são conexões ponto-a-ponto, a inclusão de mais núcleos não aumenta a carga capacitiva do fio, e portanto a frequência de transmissão da informação em um fio é maior [Zeferino 2002b];
- a NoC permite um paralelismo não presente no barramento. Mesmo que um roteador esteja ocupado, outros roteadores podem estar transmitindo várias mensagens em paralelo;
- uma NoC é facilmente escalável, basta colocar-se mais roteadores.

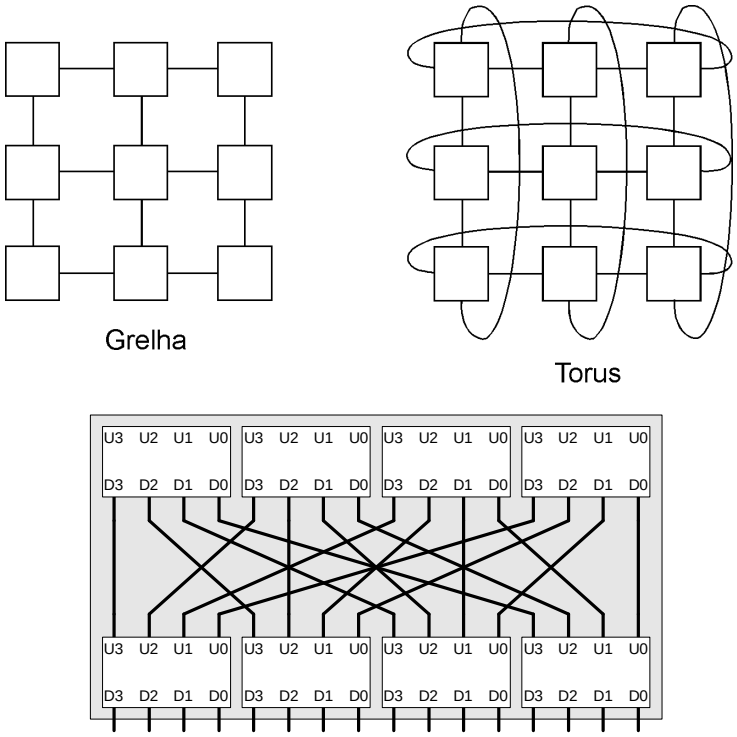


Figura 2.10. Diferentes topologias de NoC

A eficiência de uma NoC depende porém de vários fatores. O tamanho do canal de roteamento (ou seja, quantos fios fazem parte do mesmo), a política de prioridade de mensagens, a topologia da própria NoC (grelha, árvore gorda ou torus, por exemplo) e a estratégia de chaveamento devem ser escolhidas em função da aplicação [Zeferino 2002b, Karim 2002]. Além disto, a posição dos núcleos ao redor da NoC pode ser crítica, já que núcleos distantes terão latência de comunicação aumentada. Como centenas de núcleos devem ser colocados na NoC, o problema de posicionamento logo torna-se crítico. Por fim, é importante ressaltar que o custo de um roteador não é pequeno. Partindo-se de pelo menos 2000 portas [Zeferino 2002b], a área extra provocada pelo uso da NoC implica no uso de tecnologias onde os transistores realmente

tenham um baixo custo, embora o fator potência ainda não tenha sido levado em conta nas publicações recentes.

Uma NOC é uma rede de interconexão, e portanto pode ser descrita por sua topologia, política de roteamento e controle de fluxo. A topologia diz respeito ao ordenamento dos nós da rede no espaço, o roteamento determina como uma mensagem toma um certo caminho no grafo topológico. O controle de fluxo, por sua vez, trata da alocação de canais e buffers para que uma mensagem percorra o caminho necessário da fonte até o destino [Dally 1990]. Existem ainda outros parâmetros, como a arbitragem e implementação de hardware [Duato 1997].

2.2.6. Arquiteturas especializadas para sistemas embarcados

Das tendências modernas em arquiteturas de computadores, somente algumas das idéias acima discutidas serão efetivamente usadas para sistemas embarcados. O problema da potência dissipada parece definir que, com os recursos tecnológicos atuais, o uso de caches de alto desempenho, o sistema de predição de saltos e a execução fora de ordem das instruções parecem não ser viáveis. Máquinas VLIW, como o Crusoé [Klaiber 2000], parecem oferecer a combinação correta de desempenho e potência, com paralelismo descoberto em tempo de compilação, já que a maioria das aplicações embaracdas são estáticas, isto é, não são alteradas pelo usuário final.

A reconfiguração de hardware [Hartenstein 2001] estará cada vez mais presente nos sistemas embarcados. Embora os FPGAs clássicos tenham alto consumo de potência, arquiteturas de mais baixa potência estào sendo estudadas para serem embarcadas em plataformas. A enorme vantagem de se incluir a reconfiguração de hardware é a possibilidade extra de personalização de um chip, e a atenuação do enorme custo de máscaras que as tecnologias nano-métricas apresentam.

2.3. Modelagem e projeto de alto nível de sistemas embarcados

Devido à possível complexidade da arquitetura de um sistema embarcado, contendo múltiplos componentes de hardware e software em torno de uma estrutura de comunicação, e à grande variedade de soluções possíveis visando o atendimento de requisitos de projeto, como desempenho, consumo de potência e área ocupada, é essencial o projeto do sistema em níveis de abstração elevados e utilizando ferramentas que automatizem ao máximo as diversas etapas de uma metodologia consistente com os desafios existentes.

2.3.1. Metodologia de projeto

A Figura 2.11 mostra uma metodologia completa de projeto de um sistema eletrônico embarcado. Esta metodologia é ideal, segundo a perspectiva do estado-da-arte da pesquisa na área, embora na prática ainda não existam ambientes comerciais de software de projeto que a implementem inteiramente.

O projeto de um sistema embarcado é iniciado usualmente por uma especificação da funcionalidade desejada, feita através de uma linguagem ou formalismo adequado. Idealmente, esta especificação deve ter um alto nível de abstração, no qual ainda não tenham sido tomadas decisões em relação à implementação desta funcionalidade em termos da arquitetura-alvo a ser adotada, nem sobre os componentes de hardware ou software a serem selecionados. Esta especificação deve ser preferencialmente executável, para fins de validação.

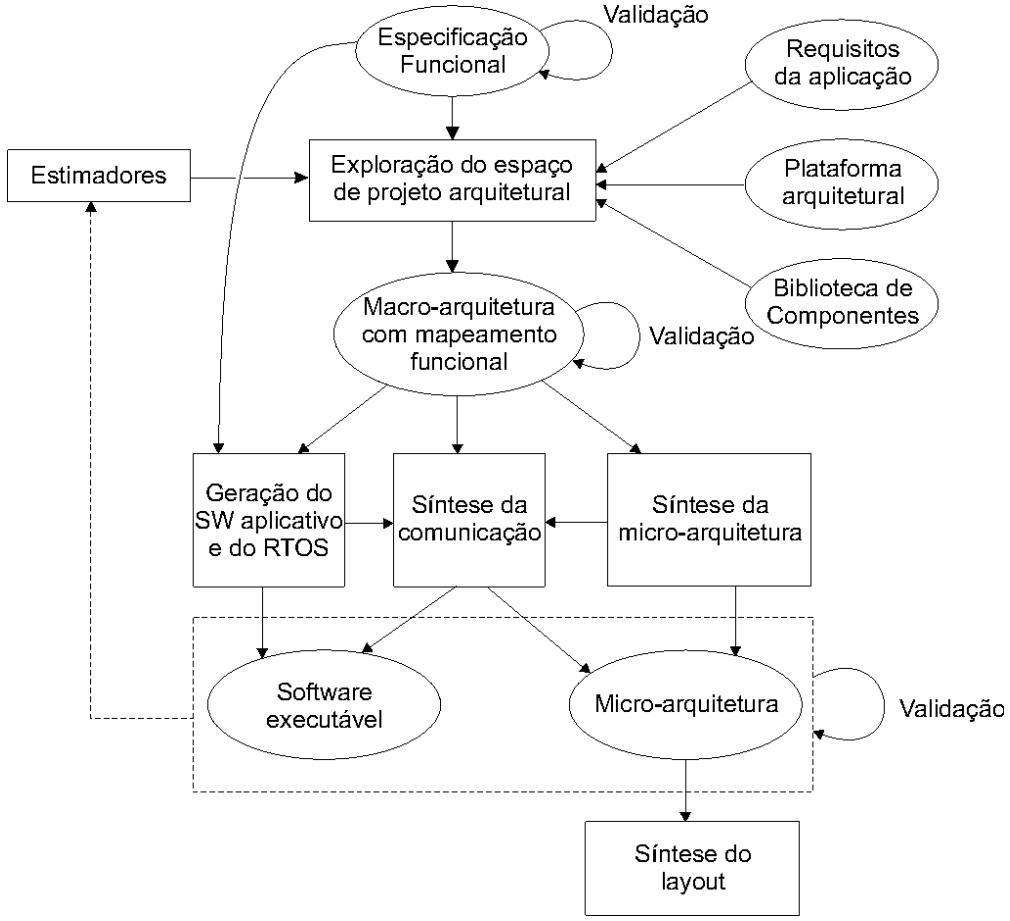


Figura 2.11. Metodologia de projeto

Deve a seguir ser feita uma exploração do espaço de projeto arquitetural, de modo a se encontrar uma arquitetura que implemente as funções contidas na especificação inicial e que atenda aos requisitos de projeto, em termos de custo, desempenho, consumo de potência, área, etc. O resultado final desta etapa é uma *macro-arquitetura* (ou *arquitetura abstrata*), contendo um ou mais processadores de determinados tipos (DSP, microcontroladores) e outros componentes necessários (memórias, interfaces, blocos dedicados de hardware), todos interconectados através de uma infra-estrutura de comunicação (um ou mais barramentos ou uma NoC). Entre a especificação funcional e a macro-arquitetura estabelece-se um *mapeamento*, através do qual cada função do sistema é atribuída a um processador ou a um bloco dedicado de hardware. Este mapeamento estabelece um determinado particionamento de funções entre hardware (blocos dedicados) e software (funções implementadas por um processador de instruções).

A exploração do espaço de projeto deve encontrar uma solução ótima para três questões básicas: 1) Quantos e quais são os processadores e blocos dedicados de hardware necessários? 2) Qual é o mapeamento ideal entre funções e componentes de hardware? 3) Qual é a estrutura de comunicação ideal para conectar os componentes entre si, tendo em vista as trocas de informações que devem ser realizadas entre as funções mapeadas para os componentes? Para que esta exploração seja efetuada rapidamente, é fundamental a existência de estimadores que, a partir da especificação funcional do sistema, sejam capazes de informar, com um grau de

precisão adequado, os valores de métricas importantes de projeto (desempenho, consumo de potência, área) que irão resultar de cada alternativa arquitetural (uma macro-arquitetura e um mapeamento de funções).

Tendo em vista um espaço quase infindável de soluções arquiteturais possíveis, com uma correspondente complexidade computacional para exploração do mesmo em busca de uma solução ótima ou mesmo sub-ótima, esta etapa é usualmente simplificada pela escolha prévia de uma plataforma arquitetural conhecida e adequada ao domínio da aplicação, contendo um ou mais processadores de tipos conhecidos, além de outros componentes necessários, todos interconectados através de uma estrutura de comunicação também pré-definida. Esta opção será discutida em detalhes na Seção 2.4.

Usualmente, diversas funções serão mapeadas para um mesmo processador, sendo então implementadas como tarefas concorrentes que precisarão ser escalonadas e gerenciadas por um sistema operacional, e caso a aplicação assim o requisite, este deverá ser um sistema operacional de tempo real (RTOS). Além da função de escalonamento de tarefas, o RTOS deve oferecer recursos para comunicação entre as tarefas, considerando que elas poderão estar distribuídas entre diversos processadores e mesmo blocos dedicados de hardware. Estes recursos devem oferecer uma abstração adequada ao software aplicativo, escondendo detalhes de mais baixo nível da infra-estrutura de comunicação. Também acionadores (*drivers*) dos periféricos devem ser oferecidos, igualmente escondendo detalhes das interfaces e da infra-estrutura de comunicação.

Uma vez definida a macro-arquitetura, é necessária a geração do software para a mesma, a partir da especificação funcional do sistema. Idealmente, seria desejável uma síntese automática do software, incluindo tanto o software aplicativo como o RTOS. Esta geração do software é bastante facilitada se a especificação funcional inicial tiver sido feita sobre uma interface de programação da aplicação (API – *application programming interface*) padronizada, que ofereça recursos para comunicação entre as tarefas e para a qual exista uma implementação sobre a plataforma arquitetural (processadores e RTOS) selecionada. É também necessário um compilador, que traduza a especificação funcional para uma linguagem de programação adequada a cada processador adotado (a menos que a especificação funcional já tenha sido feita em uma tal linguagem).

Componentes de hardware e software selecionados para a macro-arquitetura podem ter interfaces heterogêneas, implementando diferentes protocolos de comunicação. Neste caso, é necessária a síntese da comunicação entre os componentes. Esta síntese deve gerar adaptadores (*wrappers*) que fazem a conversão entre os diferentes protocolos. Adaptadores de software podem ser considerados como elementos de um RTOS dedicado gerado para a aplicação, enquanto que adaptadores de hardware são componentes dedicados que ajustam as interfaces dos componentes ao protocolo da infra-estrutura de comunicação (embora conexões ponto-a-ponto também sejam possíveis).

Uma vez definidos os componentes de hardware da macro-arquitetura, incluindo a infra-estrutura de comunicação e os eventuais adaptadores, pode ser feita a síntese do hardware. Numa primeira etapa, a macro-arquitetura pode ser expandida para uma *micro-arquitetura* (ou arquitetura RT), contendo o detalhamento de todos os componentes e suas interconexões, pino-a-pino e considerando o funcionamento do circuito com precisão de ciclo de relógio. Numa segunda etapa, podem ser usadas ferramentas convencionais de síntese de hardware, que a partir da micro-arquitetura irão gerar o layout final do circuito. Para tanto, é necessário que a micro-

arquitetura esteja descrita numa linguagem apropriada para estas ferramentas, como VHDL ou Verilog. A existência prévia de layouts para os componentes de hardware selecionados facilita bastante esta síntese, que se limita então ao posicionamento e roteamento de células.

Em todas as etapas da metodologia de projeto, é necessária uma validação das descrições funcionais e arquiteturais geradas. Normalmente, esta validação se dá por simulação, sendo portanto necessária a existência de simuladores adequados para o tratamento das linguagens utilizadas no processo de projeto. Embora ferramentas de verificação formal, que dispensam simulações exaustivas, sejam bastante atraentes, tais recursos ainda são incipientes no que se refere a sua utilização extensiva em determinados níveis de abstração.

2.3.2. Níveis de abstração

Como introduzido na seção anterior, o projeto de um sistema eletrônico embarcado passa por uma seqüência de níveis de abstração. Como não existe uma padronização destes níveis, a definição dos mesmos depende de metodologias e ferramentas particulares de projeto. Cada nível permite a validação de determinadas propriedades de projeto e serve de partida para o processo de síntese para um nível inferior subseqüente. Um esforço de padronização recente está sendo patrocinado por diversas empresas, divergindo parcialmente da apresentação a seguir [Haverinen 2002].

A especificação inicial de um sistema é usualmente feita de uma forma puramente funcional, na qual não há nenhuma informação estrutural ou dependente da arquitetura-alvo sobre a qual o sistema será implementado. Esta descrição deve ser neutra em relação a possíveis implementações das funções em software ou em hardware, não necessitando conter informações detalhadas de como implementar os requisitos temporais. O sistema é descrito como um conjunto de funções (tarefas ou objetos, dependendo da linguagem adotada), que se comunicam através de primitivas de comunicação de alto nível, por exemplo na forma de *mensagens* ou de requisições de *serviços* e respostas aos mesmos. Cada transferência pode transportar diversos itens de dados simultaneamente. Este nível de abstração permite a validação da especificação funcional do sistema e serve como entrada para o processo de exploração arquitetural.

Uma vez que são tomadas decisões em relação à arquitetura-alvo do sistema, este é descrito no nível de macro-arquitetura (ou arquitetura abstrata). Apesar da ausência de muitos detalhes, uma descrição neste nível de abstração já permite a obtenção de estimativas (de desempenho, potência, área) com um grau de precisão suficiente para que sejam tomadas decisões no processo de exploração arquitetural. Esta descrição já contém os componentes principais da arquitetura (processadores, blocos dedicados de hardware, memória, interfaces, estrutura de comunicação), assim como o mapeamento entre as funções do sistema e os processadores e blocos dedicados de hardware. No entanto, ainda não estão incluídos diversos componentes acessórios que serão necessários, tais como adaptadores de protocolos (*wrappers*), decodificadores de endereço, gerenciadores de interrupção e temporizadores, que dependem de decisões de projeto tomadas posteriormente. A descrição já é temporizada, mas não tem a precisão de ciclos de relógio.

Tarefas executadas pelos componentes no nível de macro-arquitetura comunicam-se através de mecanismos que já refletem recursos específicos incluídos na infra-estrutura de comunicação. Exemplos podem ser DMA, memória compartilhada, FIFO's de tamanho finito, escrita direta em registradores, etc., cuja diversidade reflete alternativas possíveis para tarefas mapeadas para componentes de hardware ou software. Embora em cada comunicação apenas

um item de dados seja transferido, ainda não há a escolha de um protocolo de comunicação particular.

Software já pode ser descrito através da linguagem de máquina de cada processador envolvido. No entanto, pela ausência de precisão temporal, o software é tipicamente simulado no nível de instruções (ISS - *instruction-set simulation*), e não no nível do ciclo de relógio. Com isto, os processadores podem estar descritos de forma apenas a refletir o efeito das instruções sobre conteúdos de memória e de registradores.

Uma descrição no nível de macro-arquitetura pode ser usada como entrada para os processos de síntese de comunicação, de software e de hardware. O resultado deste conjunto de sínteses será a descrição do sistema no nível de micro-arquitetura (ou nível RT), no qual o hardware estará inteiramente detalhado em termos de todos os blocos funcionais necessários, com informação a respeito de todos os seus pinos e das interconexões entre os mesmos. O software estará descrito na linguagem de máquina de cada processador contido na arquitetura, sendo simulado com precisão de ciclo de relógio, o que exige uma descrição detalhada dos processadores, em termos de pipelines e memória cache, por exemplo. A comunicação estará descrita através de sinais específicos de determinados protocolos, e cada transferência transportará um único item de dados. A descrição serve de entrada para a síntese do layout do circuito integrado, através do uso de ferramentas comercialmente disponíveis.

Outros níveis de abstração de hardware mais detalhados (portas lógicas, transistores) não são relevantes no contexto do projeto de sistemas embarcados. Usualmente, o sistema é projetado pela interconexão de componentes já previamente projetados e validados, e estes níveis de abstração são relevantes apenas no projeto interno destes componentes. O nível final de projeto, que é o layout do circuito, é relevante apenas do ponto de vista do posicionamento dos componentes e do roteamento de suas interconexões, e não do projeto do layout interno de cada componente.

2.3.3. Linguagens de especificação e projeto

O projeto de um sistema embarcado passa por diversos níveis de abstração e é orientado a um determinado domínio de aplicação (aplicações orientadas a dados ou controle, por exemplo). Para a representação de cada domínio, existe um modelo de computação [Edwards 1997] mais adequado. Diferentes linguagens de especificação e de projeto têm sido adotadas para tratamento destes níveis de abstração e modelos de computação.

Para a especificação funcional inicial, é importante que a linguagem seja executável, para fins de validação. Seria também desejável que a linguagem permitisse a descrição de funções de forma neutra em relação a sua implementação em software ou hardware e que pudesse ser utilizada como entrada para um processo de síntese automática do software sobre a plataforma arquitetural adotada. Por outro lado, poderia ser também interessante a adoção de uma linguagem neutra em relação aos domínios de aplicação, requisito que no entanto parece conflitante com a possibilidade de síntese automática do software, já que este processo é altamente dependente da plataforma arquitetural e do domínio de aplicação (modelo de computação).

As descrições da macro-arquitetura e da micro-arquitetura devem ser feitas em linguagens que possam ser utilizadas como entrada para processos de síntese automática de hardware, além de serem evidentemente simuláveis.

Linguagens de programação têm sido bastante utilizadas, aproveitando sua popularidade. Este é o caso evidente de C e C++. Se, por um lado, C não é uma linguagem ideal do ponto de vista do conjunto de requisitos anteriormente estabelecidos para linguagens de especificação, especialmente em relação ao grau de abstração e à generalidade dos domínios de aplicação, ela tem a vantagem de permitir a geração de software para um grande número de processadores utilizados no contexto de sistemas embarcados. A adoção da orientação a objetos, como em C++, se por um lado é vantajosa do ponto de vista de especificações de alto nível onde reuso e especialização de componentes são características muito interessantes, por outro lado causa problemas para a síntese de hardware ou tamanho do software.

Para a descrição de hardware, as linguagens mais populares são VHDL [VHDL 2002] e Verilog [Verilog 2003]. Sua grande vantagem é a possibilidade de serem utilizadas como entrada para simulação e síntese automática de circuitos descritos no nível da micro-arquitetura, através da utilização de ferramentas comerciais bastante difundidas. VHDL, no entanto, é uma linguagem mais orientada para simulação, de tal modo que algumas de suas construções não são sintetizáveis, o que força ferramentas de síntese a aceitarem apenas um determinado sub-conjunto da linguagem e/ou estilo de descrição. VHDL e Verilog têm uma semântica orientada apenas para a descrição de hardware, não sendo apropriadas para descrições de software nem para especificações funcionais de alto nível.

C e C++ apresentam um evidente prejuízo quando consideradas como linguagens de projeto no nível da macro ou da micro-arquitetura, tendo em vista sua inadequação semântica para a descrição de aspectos de hardware. A introdução de SystemC [SystemC 2003] visa justamente sanar este defeito, combinando as vantagens da popularidade de C++ e sua adequação ao processo de geração de software com uma semântica adicional (na forma de uma biblioteca de funções) apropriada para a descrição de hardware, através de construções como portas, sinais, relógios (que sincronizam processos), etc. SystemC 1.0 apresentava uma semântica de hardware muito próxima de VHDL, e portanto mais adequada para descrições no nível RT, exigindo por exemplo a descrição da comunicação entre processos através de sinais de um protocolo específico, o que já reflete uma determinada implementação do hardware. SystemC 2.0, por outro lado, introduziu construções de comunicação e sincronização de mais alto nível (*canais*, *interfaces* e *eventos*), permitindo a modelagem de mecanismos mais abstratos de comunicação e, portanto, o uso da linguagem em níveis de abstração superiores ao nível RT.

Na tentativa de aumentar as possibilidades de modelagem e abstração, Java também tem sido utilizada como ferramenta de descrição e simulação de sistemas [Mulchandani 1998, Mrva 1998, Ito 2000, Ito 2001]. Contudo, o nível de abstração oferecido é equivalente àquele utilizado em C++. Outras linguagens como Matlab [Mathworks 2003] podem ser usadas, devido à facilidade de modelagem de fenômenos físicos e pela possibilidade de descrições mistas analógico-digitais. Em especial, Matlab possui mecanismos de resolução internos que permitem a fácil descrição de aplicações *stream*, isto é, aquelas em que uma grande quantidade de dados chega a intervalos regulares e conhecidos para processamento.

Nenhuma destas linguagens (C, C++, Java, Matlab, VHDL, Verilog, SystemC), no entanto, atende simultaneamente os requisitos de cobertura de múltiplos domínios e níveis de abstração e de abstração em relação a implementações de software e hardware. Rosetta [Alexander e Kong 2001] é um esforço recente, ainda não implementado comercialmente, que procura oferecer esta cobertura completa através de uma abordagem distinta. Rosetta oferece

uma base sintática e semântica comum, a partir da qual podem ser descritos e combinados diferentes modelos de computação, cada um deles implementados por uma linguagem específica.

Finalmente, deve ser salientado que as linguagens aqui consideradas são essencialmente destinadas à *descrição* do comportamento e/ou estrutura de um sistema, seja em software e/ou em hardware, e não à *especificação* funcional e/ou de requisitos, o que poderia ser feito com uma linguagem como UML [Fowler 2000]. Esforços recentes têm procurado investigar a utilização de UML na especificação funcional e de requisitos de sistemas embarcados [Lavagno 2001].

2.3.4. Compiladores e suas amarras com o modelo arquitetural

Os compiladores têm um papel fundamental no desenvolvimento de sistemas embarcados. Como a complexidade dos sistemas cresce continuamente, suportada pela lei de Moore, a geração de código deve ser numa linguagem o mais abstrata possível, para diminuir o tempo de projeto. Por outro lado, muitas vezes deve-se programar diretamente em assembler, para melhor atingir os objetivos de projeto de menor consumo e maior desempenho. Esta contudo é uma solução muito custosa, pelo tempo de projeto envolvido. O custo de compiladores adequados para máquinas VLIW, por exemplo, é muitas ordens de grandeza maior que o custo de um compilador para uma arquitetura tradicional. Isto se explica não somente pelo volume de vendas, mas também pela complexidade do compilador em si, já que ele é o responsável por obter os dados de paralelismo que poderão gerar código eficiente.

Segundo Paulin (1997), nos domínios de aplicação de sistemas embarcados no mercado de comunicação, 75% dos aplicativos que executam em CPUs tradicionais e 90% daqueles que executam em arquiteturas DSP são escritos em assembler. Isto porque, até o compilador de uma nova arquitetura ficar pronto, já houve mudanças na arquitetura para suportar novas funções requeridas. Para atender a esta necessidade já existem muitas pesquisas em compiladores multi-alvo (*retargeting compilers*) [Bhattacharyya 2000, Halambi 1999], que na prática são geradores de compiladores. A entrada de tais programas é uma aquitetura de hardware, descrita em nível estrutural, relativa ao conjunto de instruções suportado. Ao gerador de compiladores cabe então re-escrever o compilador, de maneira a tirar o máximo proveito das novas instruções ou das variações arquiteturais propostas.

Infelizmente, o nível de abstração no qual os geradores de compiladores trabalham não permite o real reuso de funções. Por exemplo, para que um buffer circular esteja visível ao conjunto de funções, ele deve ter não apenas instruções equivalentes de assembler, mas definitivamente construções correspondentes de alto nível na própria linguagem a ser compilada, sob pena de não ser usado. Este problema repete-se continuamente, já que estratégias de alto nível são sempre mais eficientes que estratégias mais tardias. Consequentemente, é provável que somente quando o modelo de computação, e consequentemente a linguagem usada para descrever o comportamento do sistema, levar em conta as características da arquitetura de suporte é que será possível ter-se uma geração de código adequada.

Técnicas como pipeline de software, desenrolamento de laços e reordenamento de laços são constantemente utilizadas, já que se o hardware de suporte tem possibilidade de paralelismo, este deve ser explorado ao máximo.

2.3.5. Particionamento entre software e hardware

O particionamento de funções entre software e hardware é realizado como parte da exploração do espaço de projeto arquitetural. A entrada para este processo é a especificação funcional do

sistema, idealmente desenvolvida numa linguagem e num estilo de descrição tais que não privilegiem determinados particionamentos. A saída do processo é um mapeamento entre cada função da especificação e um componente da macro-arquitetura (um processador ou um bloco dedicado de hardware).

O processo de particionamento é evidentemente dependente da macro-arquitetura selecionada. Idealmente, um espaço de soluções muito mais amplo seria explorado se a própria definição da macro-arquitetura (número e tipos de componentes) fosse resultante de um processo automático de particionamento, conforme sugerido na Figura 2.12(a). Esta abordagem, no entanto, teria uma complexidade computacional bastante maior, motivo pelo qual usualmente uma macro-arquitetura é definida e então particionamentos diversos são explorados sobre a mesma, como mostrado na Figura 2.12(b). Esta abordagem é aceitável num grande número de situações, onde o projeto é na verdade apenas uma variação de um projeto anterior para o qual já foi encontrada uma macro-arquitetura aceitável.

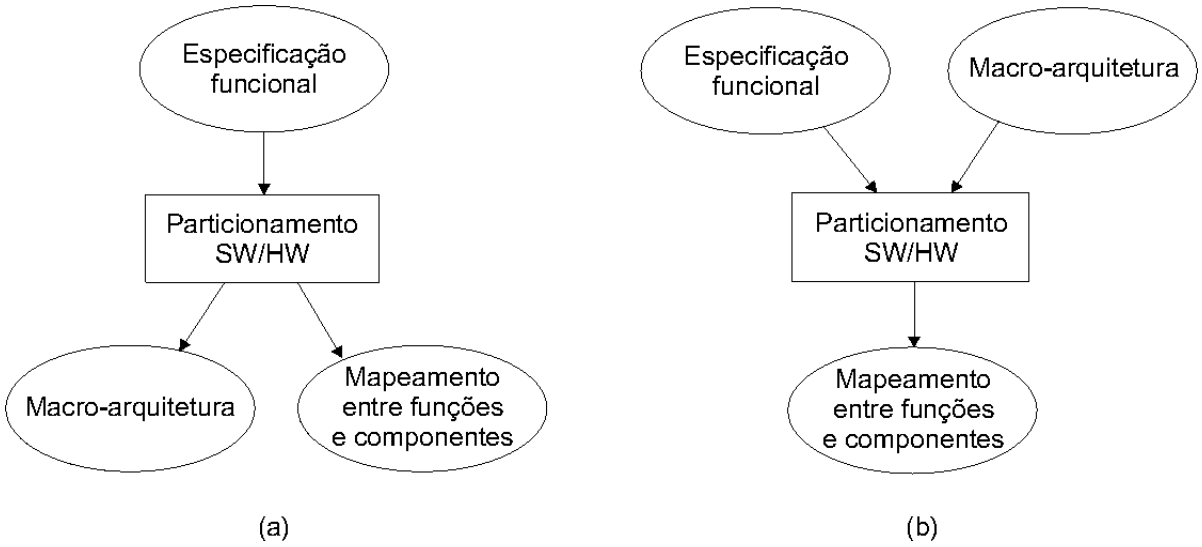


Figura 2.12. Relação entre particionamento e definição da macro-arquitetura

Há uma enorme literatura a respeito do particionamento automático entre hardware e software [Edwards 1997]. Uma abordagem clássica supõe inicialmente que o sistema será inteiramente desenvolvido em software, sobre um processador conhecido. Uma avaliação de desempenho, através de um estimador adequado ou de um simulador, permite identificar partes críticas da aplicação, que comprometem restrições temporais. Estas partes são movidas para blocos dedicados de hardware. O processo é repetido até que uma solução aceitável seja encontrada.

Um problema destas abordagens de particionamento automático, no atual contexto de sistemas embarcados, é que elas são geralmente direcionadas ao atendimento de restrições de desempenho, desconsiderando a potência. A maioria delas é restrita à exploração de plataformas arquiteturais contendo um único processador, de tipo previamente conhecido.

A metodologia pragmática atualmente adotada prevê uma exploração manual, como em ferramentas comerciais como CoCentric System Studio, da Synopsys [Synopsys 2003], onde o próprio projetista faz uma alocação de funções a um ou mais processadores e a blocos dedicados de hardware, todos previamente definidos, interconectados através de uma dada infra-estrutura de comunicação, sendo o resultado deste mapeamento avaliado através de estimadores (de

desempenho e de consumo de potência, por exemplo). Assim, a busca de uma solução aceitável é feita por tentativas sucessivas, processo que evidentemente não garante que uma solução ótima será encontrada. No entanto, caso os estimadores tenham suficiente precisão e sejam executados com bastante rapidez, o projetista pode encontrar uma solução sub-ótima num curto espaço de tempo. Caso a exploração não alcance nenhuma solução aceitável, uma nova macro-arquitetura deve ser definida e um novo processo de exploração manual deve ser iniciado.

O particionamento de funções entre hardware e software também pode ser aplicado na geração do RTOS, como proposto por Mooney III e Blough (2000), onde uma função crítica como o escalonamento de tarefas pode ser implementada em hardware em um co-processador dedicado.

2.3.6. Adaptação automática de código binário

O avanço da capacidade computacional dos processadores modernos é tal que se podem realizar aplicações onde se pode permitir uma certa perda de desempenho, para ganho em outros aspectos do projeto. Compatibilidade de software é um deles. Como uma larga parte do tempo de desenvolvimento dos sistemas embarcados é gasto em software, todo novo avanço arquitetural deve manter a herança do software já desenvolvido. Isto evidentemente impõe limites nos avanços que os projetistas de arquitetura conseguem alcançar. Contudo, com a lei de Moore seguindo válida, a quantidade de transistores é tamanha que algumas soluções comerciais baseadas na tradução binária automática começam a aparecer [Altman 01].

A idéia central da tradução binária é realizar uma arquitetura de CPU otimizada para uma certa função, capaz de executar um conjunto de instruções livre da necessidade de compatibilidade com o software mais antigo. Assim, pode-se aproveitar todos os avanços de arquitetura disponíveis. Resta o problema da herança de software. Para tal, as novas máquinas traduzem, durante a execução, um código binário em outro, do software original para a nova arquitetura. Evidentemente, haverá uma perda de desempenho, compensada pela enorme capacidade da nova máquina, e pelo fato que, em muitos domínios de aplicação, somente o desempenho bruto não é suficiente, devendo-se atentar também para a potência consumida. Este é, por exemplo, o caso do processador Crusoé [Klaiber 2000], que é capaz de executar código nativo para máquinas da família x86, mas internamente possui um outro conjunto de instruções, baseado numa arquitetura VLIW. Estes processadores são hoje utilizados em PDAs, por exemplo, pois seu consumo é muito mais adequado a aplicações portáteis que os modelos mais recentes da linha x86.

Central ao uso de tradução binária está o conceito de “escreva uma vez, execute em qualquer lugar”. Com a tradução binária, a arquitetura de uma máquina é puramente virtual, isto é, uma camada de software, e portanto, otimizações recentes de hardware podem ser compartilhadas por códigos mais antigos. Como a arquitetura é uma camada de software, pode-se imaginar um cenário onde se poderia obter via internet o código de uma aplicação, que originalmente somente executaria em estações de trabalho de uma certa marca, e seria possível executá-lo numa máquina local, após o carregamento nesta do modelo de arquitetura desejado.

Exemplos de máquinas comerciais que utilizam a tradução binária são a Daisy [Ebcioglu e Altman 1997] e o Crusoé [Klaiber 2000]. Daisy é uma máquina virtual capaz de executar de 3 a 4 instruções do PowerPC por ciclo. Ela foi projetada para atender ao mercado de servidores, possuindo uma arquitetura VLIW que é capaz de executar 8 a 16 instruções nativas por ciclo. A máquina é capaz de endereçar gigabytes de memória, mas ao menos 100 Mbytes são consumidos

pela máquina interpretadora. O Crusoé é um processador voltado a outro segmento de mercado, os PDAs. Executa 2 a 4 instruções por ciclo da família x86, é capaz de endereçar entre 64 e 128 Mbytes e consome 16 Mbytes para a máquina virtual. Um Crusoé a 667 MHz é equivalente a um Pentium III a 500 MHz, mas consome apenas uma fração da potência, utilizando apenas 7 milhões de transistores.

Evidentemente, uma máquina de tradução binária jamais será mais rápida que uma máquina de execução nativa, mas, do ponto de vista de sistemas embarcados, a herança de software e a facilidade de troca de plataforma (basta carregar uma nova máquina virtual na arquitetura real) tornam o conceito de tradução binária interessante. Existem no entanto outros problemas além do desempenho. O sistema de interrupção, para tratamento de sistemas de tempo real, pode ser muito diferente entre a máquina herdada e a máquina real que executa o código. A camada de tradução, por outro lado, introduz mais um fator complicador para a correta previsibilidade do tempo de resposta para um sistema de tempo real. Outro ponto é que todas as otimizações que o compilador possa ter feito para a máquina original serão perdidas, o que pode ter impacto no tempo total de execução, adicionando mais perda de desempenho à interpretação.

Aliada aos desenvolvimentos de hardware reconfigurável [Hartenstein 2001], a tradução binária transforma o software e o hardware em *commodities*, isto é, blocos básicos gerais que podem ser facilmente adaptados para novos domínios de aplicação. A personalização e o fato de haver reuso extensivo dos recursos de hardware mais modernos favorecem o projeto baseado em plataformas. Pode-se imaginar um cenário onde as inovações de hardware, como uso de FPGAs ou de tecnologias mais modernas com maior capacidade de integrar memórias, possam ser usadas para fazer o hardware mais eficiente, mas mantendo todo o conjunto de aplicações, sem que se tenha de adaptar o software à nova plataforma.

2.3.7. Síntese de comunicação

Numa situação ideal, componentes selecionados para uma dada macro-arquitetura teriam interfaces compatíveis e poderiam ser conectados diretamente uns aos outros, ou então através de uma estrutura de comunicação também consistente. Isto vale tanto para componentes de hardware, por exemplo conectados através de um barramento padronizado, como para componentes de software, comunicando-se através de funções padronizadas disponíveis numa API. Numa situação mais genérica, no entanto, o projetista pode desejar reusar componentes já disponíveis que foram desenvolvidos em projetos anteriores, dentro de contextos diferentes, e que podem portanto apresentar interfaces inconsistentes. Este é o caso de reuso de componentes de propriedade intelectual (IP), oferecidos por fornecedores externos à empresa dentro de um contexto de comércio eletrônico, ou mesmo desenvolvidos por outras equipes dentro da empresa.

Componentes IP de hardware podem ser oferecidos de duas maneiras distintas. Componentes *hard* já estão sintetizados para uma dada tecnologia-alvo e é seu layout que é comercializado, já caracterizado quanto aos resultados obtidos em termos de área, potência e desempenho. Sua restrição, no entanto, é a quase impossibilidade de adaptação para outra situação (outra arquitetura e/ou tecnologia). Componentes *soft*, por outro lado, são ofertados através de descrições de mais alto nível (tipicamente nível RT), podendo ter suas interfaces e comportamentos adaptados, se necessário, e sintetizados para diferentes tecnologias-alvo. No entanto, sua caracterização precisa em termos das métricas de projeto terá que ser feita pelo projetista do sistema. Componentes IP de software também podem ser divididos em

componentes *hard*, já compilados para um determinado processador e oferecidos através de seu código executável, e *soft*, descritos numa linguagem de alto nível.

A adaptação de componentes IP a um novo projeto pode ser feita de duas maneiras. No caso de componentes *soft*, descrições de alto nível estão disponíveis e podem ser modificadas de modo que as interfaces de componentes a serem conectados passem a ser consistentes entre si. A adaptação pode também afetar a própria funcionalidade do componente, caso isto seja necessário no contexto do novo projeto. No caso de componentes *hard*, no entanto, interfaces heterogêneas não podem ser adaptadas através de modificações nas descrições dos componentes. Neste caso, é imprescindível o desenvolvimento de adaptadores de comunicação (ou *wrappers*).

Soluções para a síntese automática de adaptadores entre componentes IP de hardware com interfaces heterogêneas dependem da existência de uma descrição formal destas interfaces, como *expressões regulares* [Passerone 1998] ou *máquinas de estados finitos* [Smith e DeMicheli 1998]. Mas estas abordagens não se aplicam a componentes de software. A ferramenta TEReCS [Böke 2000] sintetiza software de comunicação entre tarefas, dada uma especificação da estrutura de comunicação e uma alocação de tarefas a processadores. De modo geral, as abordagens de síntese de adaptadores para componentes de software estão associadas à geração de um sistema operacional mínimo e dedicado para uma aplicação, como em TEReCS, ou de pelo menos um escalonador de tarefas dedicado, como em IPChinook [Chou 1999].

Abordagens mais recentes propõem o desenvolvimento uniforme de adaptadores de software e hardware. No ambiente COSY [Brunel 2000], existe uma biblioteca de adaptadores de hardware e software previamente desenvolvida e caracterizada em termos de desempenho, oferecendo diferentes *esquemas* de comunicação (DMA, memória compartilhada, FIFO, escrita e leitura em registradores, etc.). A aplicação é descrita inicialmente através de um mecanismo abstrato de comunicação. Na medida em que funções são mapeadas para processadores ou blocos dedicados de hardware, selecionados numa macro-arquitetura, o projetista pode escolher manualmente os esquemas de comunicação que julga mais apropriados, realizando a seguir uma avaliação do desempenho global do sistema resultante desta escolha. No ambiente ROSES [Cesario 2002], adaptadores de hardware e software são sintetizados automaticamente a partir da composição de elementos básicos disponíveis em bibliotecas expansíveis, também a partir de uma seleção de mecanismos de comunicação feita manualmente pelo projetista. A abordagem ROSES também está associada à geração de um sistema operacional mínimo e dedicado [Gauthier 2001].

Note-se que, mesmo que um componente de hardware tenha sido desenvolvido com uma interface consistente com algum padrão de barramento, ainda assim a integração deste componente a um sistema construído em torno deste barramento exigirá a inclusão de alguma lógica adicional. A ferramenta Coral [Bergamaschi 2001], por exemplo, que permite a integração de componentes consistentes com o padrão de barramento CoreConnect da IBM, dá suporte a tarefas como as definições de mapas de memória acessados pelos diferentes componentes e de prioridades de componentes num sistema de interrupção e num esquema de arbitramento do barramento.

2.3.8. Estimadores

A exploração do espaço de projeto num alto nível de abstração depende da existência de estimativas suficientemente precisas dos resultados finais a serem obtidos na implementação do

sistema para cada alternativa de projeto considerada. São necessárias estimativas para métricas relevantes de cada projeto, como desempenho, consumo de potência e área no silício. A obtenção destas estimativas deve atender dois requisitos principais: precisão e velocidade. Se por um lado deseja-se alta precisão, esta por outro lado só pode ser obtida com estimativas feitas em níveis de abstração mais baixos, que usualmente dependem de uma síntese de software e hardware que pode ser demorada e exigir diversas outras decisões de projeto mais detalhadas. Compromete-se assim uma rápida exploração dos potenciais impactos causados por decisões arquiteturais de alto nível. É evidente que, em qualquer caso, as estimativas devem considerar a plataforma arquitetural sobre a qual o sistema embarcado será implementado.

Uma estimativa de desempenho pode ser usualmente obtida através de simulação num nível de abstração que considere aspectos temporais. O nível de abstração a ser escolhido, no entanto, depende da precisão desejada para a estimativa. Se a estimativa for feita no nível da micro-arquitetura, isto exigirá que o software aplicativo seja compilado para o processador adotado na plataforma e que exista uma descrição detalhada do hardware no nível RT. Isto exigirá uma síntese de software e hardware bastante demorada, o que inviabiliza a idéia da rápida exploração do espaço de projeto. Como alternativa num nível de abstração mais alto, pode-se anotar o código fonte da aplicação (por exemplo escrito em C) com atrasos estimados para cada comando ou grupo de comandos. Isto exige obviamente uma caracterização prévia do tempo consumido por estes comandos quando executados sobre o processador desejado. Não se pode desprezar nesta análise o efeito do compilador sobre a qualidade do código gerado. Também o efeito da implementação do sistema operacional pode ser considerado, desde que existam estimativas razoáveis sobre o custo de funções importantes como o escalonamento de tarefas e a comunicação entre tarefas. Bacivarov (2002), por exemplo, anota uma descrição funcional da aplicação, incluindo o RTOS, com estimativas de desempenho, e compila esta descrição para a linguagem nativa do computador hospedeiro da simulação, o que garante processamento muito rápido. Estimativas de desempenho são também necessárias para a análise de escalonabilidade de tarefas em sistemas operacionais, assunto que é abordado na Seção 2.4.

Uma estimação de alto nível da potência consumida por uma aplicação embarcada pode ser baseada na soma das potências consumidas nas instruções executadas pelo processador durante a aplicação [Tiwari 1994]. Cada instrução tem um consumo que é definido, a partir de experimentações prévias, pela média da corrente elétrica requerida na execução repetida daquele tipo de instrução, conforme ilustrado na Tabela 2.1 para diferentes instruções do processador 486DX2. Este método pode ser refinado com modelos específicos de consumo de potência para diferentes componentes arquiteturais, como unidades funcionais, bancos de registradores, barramentos, memórias e unidades de controle, que são afetados pelas instruções do programa [Dalal e Ravikuman 2001, Choi e Chaterjee 2001, Chen 2001, Givargis 2001]. Estes modelos consideram o tipo de atividade (variações de valores de sinais) ocorrido na entrada de cada componente, assim como propriedades físicas de cada componente.

Outras abordagens [Landman e Rabaey 1996, Simunié 1999] propõem uma estimativa de potência mais precisa a partir da simulação no nível dos ciclos de relógio, também considerando a potência consumida por cada bloco funcional. A implementação de simuladores de código compilado permite compensar o maior tempo de processamento exigido por este nível de abstração.

Embora simuladores de potência no nível de instrução tenham a velocidade como maior vantagem, eles produzem resultados menos acurados e eventualmente até incorretos. Deve-se

notar sua incapacidade de considerar bolhas em pipelines, causadas por dependências de dados e de controle, assim como falhas em caches, TLB's (*Translation Look-aside Buffers*) e BTB's (*Branch Table Buffers*). A desvantagem dos simuladores no nível de ciclo de relógio, além do maior tempo de processamento, é a necessidade de se dispor de uma descrição detalhada da arquitetura, usualmente no nível RT. Enquanto a maioria dos simuladores deste tipo está restrito à arquitetura de um processador em particular, Beck Filho et alii [2003] apresentam um simulador que aceita como entrada a descrição de uma arquitetura qualquer, a partir da interconexão de componentes arquiteturais quaisquer, para os quais no entanto devem ser providos modelos adequados de consumo de potência.

Tabela 2.1. Corrente exigida em instruções do processador 486DX2 [Tiwari 1994]

Instrução	Corrente (mA)	Ciclos
NOP	275.7	1
MOV DX, BX	302.4	1
MOV DX, [BX]	428.3	1
MOV DX, [BX][DI]	409.0	2
MOV [BX], DX	521.7	1
MOV [BX][DI], DX	451.7	2

2.3.9. Validação

Ao longo do processo de projeto, inúmeras descrições do sistema embarcado são geradas, em diferentes níveis de abstração (especificação funcional, macro-arquitetura, micro-arquitetura), cobrindo diferentes aspectos (software e hardware) e eventualmente com a utilização combinada de diferentes linguagens (p.ex. Java e VHDL). Estas descrições precisam ser validadas, o que exige a execução das descrições, no caso de linguagens de programação como Java e C, ou sua simulação, no caso de linguagens de descrição de hardware ou de sistemas como VHDL e SystemC.

Conforme já discutido na Seção 2.3.3, nenhuma das linguagens consegue cobrir simultaneamente todos os domínios de aplicação, níveis de abstração e aspectos de hardware e software. Assim, é bastante comum a necessidade de validação de descrições multi-linguagem em determinados passos do projeto. Um exemplo evidente é a validação combinada da micro-arquitetura, descrita por exemplo em VHDL, e do software que irá rodar sobre um processador. Outro exemplo é a validação de um sistema embarcado contendo partes de hardware digital, descritas em VHDL, hardware analógico, descrito por equações diferenciais em Matlab, e software, descrito em C.

Por outro lado, a validação de um sistema muito complexo pode ser bastante facilitada por um processo de refinamentos sucessivos, no qual apenas partes selecionadas do sistema são descritas num nível de abstração mais detalhado (p.ex. micro-arquitetura), enquanto o restante do sistema permanece descrito de forma mais abstrata (p.ex. como macro-arquitetura ou mesmo como componente puramente funcional). Esta abordagem, que permite melhor focalizar as decisões de projeto que precisam ser validadas a cada novo passo de refinamento, também pode resultar em descrições multi-linguagem.

Exemplos de co-simulação entre VHDL e C são encontrados nos simuladores VCI [Valderrama 1998] e SIMOO [Oyamada e Wagner 2000]. Ferramentas de co-simulação atuais,

como CoCentric System Studio, da Synopsys [Synopsys 2003], e Seamless CVE, da Mentor [Mentor 2003], permitem a integração de SystemC, C, simuladores de software no nível de instruções de máquina de um processador e HDL's diversas, como VHDL e Verilog. MCI [Hessel 1999] é um exemplo de solução genérica, que permite a geração de modelos de co-simulação a partir de uma especificação multi-linguagem do sistema. No entanto, a co-simulação dá-se através de um mecanismo de comunicação proprietário, como nas soluções comerciais. O ambiente Ptolemy [Davis II 2001] adota uma abordagem orientada a objetos na modelagem do sistema, e oferece um reportório de classes explicitamente orientado à co-simulação de diferentes modelos de computação.

No contexto de reuso de componentes IP, torna-se interessante o desenvolvimento de simulações distribuídas, nas quais componentes IP sejam simulados remotamente, no *site* de seus fornecedores, visando a proteção da propriedade intelectual. O ambiente JavaCAD [Dalpasso 1999] oferece este recurso, mas está restrito a componentes descritos em Java, assim como o ambiente proposto por Fin e Fummi (2000) está restrito a componentes VHDL e Verilog. Soluções mais genéricas, abertas a diversas linguagens, são propostas pelos ambientes WESE [Dhananjai 2000], baseado em CORBA, e DCB [Mello e Wagner 2002], inspirado no padrão HLA [HLA 2000] de simulação distribuída.

A validação por simulação apresenta duas grandes restrições. Em primeiro lugar, o número de casos de testes para uma validação exaustiva da descrição do sistema é grande demais, obrigando os projetistas na prática a limitarem-se a uma cobertura parcial dos mesmos. Em segundo lugar, quanto mais baixo o nível de abstração, maior o número de casos de teste e mais demorada é a simulação, pelo maior detalhamento da descrição. Técnicas de verificação formal [Edwards 1997], que realizam uma validação simbólica do sistema, e não numérica, prometem resolver este problema por cobrirem todos os possíveis casos de teste através de um único processamento. No entanto, tais técnicas ainda não atingiram um grau de maturidade suficiente que permita sua aplicação em larga escala em todo o processo de projeto, estando limitadas a determinadas combinações de linguagens, estilos de descrição, domínios de aplicação e níveis de abstração.

2.4. Projeto baseado em plataforma

O grande espaço de soluções arquiteturais possíveis para a implementação de uma determinada aplicação embarcada torna o processo de exploração arquitetural computacionalmente muito complexo. O reuso de plataformas de hardware e software, padronizadas e previamente validadas, orientadas para determinados domínios de aplicação, permite uma grande redução no espaço de soluções e portanto no tempo de projeto de um novo sistema.

2.4.1. Plataformas, derivativos e componentes IP

Uma plataforma é uma base comum de hardware e software que pode ser reutilizada em projetos de diferentes sistemas embarcados [Sangiovanni-Vincentelli e Martin 2001, Keutzer 2000]. Esta base comum pode ser composta, do lado de hardware, por uma micro-arquitetura praticamente fixa, com ou mais processadores e outros componentes complementares, interconectados através de uma estrutura de comunicação, e, do lado de software, por um RTOS (incluindo acionadores de periféricos) acessível através de uma API (*Application Programming Interface*). Esta base é padronizada, de tal modo que seus componentes principais não precisam ser novamente revalidados a cada novo projeto. No entanto, tendo em vista necessidades específicas de cada

nova aplicação, esta plataforma precisa oferecer um grau adequado de parametrização e configuração. Assim, como exemplos, a plataforma pode estar aberta à inclusão de novos blocos dedicados de hardware, a capacidade de sua memória pode ser definida para cada aplicação e o RTOS pode ser configurado apenas com os serviços indispensáveis à aplicação.

Chama-se de *derivativo* o projeto de um novo sistema a partir da configuração ou parametrização da plataforma básica. No projeto de um derivativo, o maior esforço reside no desenvolvimento (especificação, projeto, depuração) do software aplicativo, já que a configuração do hardware e do RTOS pode ser feita de forma quase automática, com uma razoável certeza quanto ao correto funcionamento dos mesmos. É a API, que oferece ao software aplicativo os serviços básicos implementados pela micro-arquitetura e pelo RTOS, que abstrai os detalhes de baixo nível da plataforma e permite o projeto de um derivativo baseado quase exclusivamente no desenvolvimento do software aplicativo. Esta API pode ser considerada como uma *plataforma de software*, que, junto com a *plataforma de hardware* (a micro-arquitetura), forma uma *plataforma de sistema*.

O projeto baseado em plataformas depende da existência de componentes de hardware e software reusáveis e compatíveis com os padrões estabelecidos pela plataforma. Uma plataforma pode estar por exemplo baseada em um microcontrolador ARM conectado a outros componentes através de um barramento AMBA [ARM 1999], incluindo um RTOS VxWorks [WindRiver 2003] configurável e dedicado a este processador. Para o projeto de um derivativo desta plataforma, novos componentes de hardware cujas interfaces sejam compatíveis com o barramento AMBA podem ser incluídos. Da mesma forma, componentes de software compatíveis com os serviços oferecidos pelo RTOS podem ser utilizados no desenvolvimento da aplicação. Este estilo de projeto incentiva o surgimento de fornecedores de componentes reusáveis, ditos de *propriedade intelectual* (IP) [Design&Reuse 2003].

Igualmente desejável no projeto baseado em plataformas e no reuso de componentes IP é a possibilidade de caracterização prévia, com alto grau de precisão, dos componentes arquiteturais básicos (hardware e RTOS), de modo que é possível a obtenção de estimativas precisas do impacto do derivativo de uma plataforma desenvolvido para uma dada aplicação.

2.4.2. Plataformas de software

Sistemas operacionais (SO) possuem quatro funções principais: gerência de processos (ou tarefas), comunicação entre processos, gerência de memória e gerência de E/S. A gerência de processos inclui aspectos como criação, carga e controle da execução de processos. Na função de comunicação entre processos devem ser consideradas questões como sincronização entre processos, deteção e tratamento de *deadlocks* e mecanismos de troca de dados. A gerência de memória inclui a criação, remoção e proteção de arquivos. Finalmente, a gerência de E/S é responsável pelo controle de comunicação com os periféricos, incluindo tratamento de interrupções.

Um sistema operacional de tempo real (RTOS) [Burns e Wellings 1997] é um SO cujo funcionamento adequado, além de cobrir as funções acima, depende do atendimento correto de requisitos temporais associados à execução dos processos, tais como *deadlines* (tempos máximos de execução) e períodos de processos periódicos (intervalos de tempo entre inícios sucessivos de execução de um processo). A principal conseqüência das restrições temporais incide sobre o escalonamento de tarefas, função associada à gerência de processos. Num RTOS, tarefas sempre têm uma prioridade associada, definida de acordo com critérios que podem

variar, como forma de garantir o atendimento das restrições temporais, e devem ser preemptivas, ou seja, devem poder ser interrompidas por tarefas de maior prioridade. Uma tarefa em um RTOS nunca tem sua execução iniciada apenas porque ela está pronta para executar, como ocorre em um SO convencional, mas sim porque tem prioridade máxima naquele instante de tempo. Um RTOS utiliza temporizadores que permitem programar interrupções para execução de funções como o escalonamento de tarefas.

Algoritmos de escalonamento de tarefas podem ser divididos em estáticos e dinâmicos. Em algoritmos estáticos, que apresentam menor *overhead* em tempo de execução, a prioridade das tarefas é definida estaticamente. O algoritmo estático mais conhecido é o RMS (*Rate Monotonic Scheduling*), que atribui a cada tarefa uma prioridade inversamente proporcional ao *deadline* da mesma. Este algoritmo garante a escalonabilidade das tarefas caso a utilização da CPU não ultrapasse 69%. A perda de 30% no tempo útil do processador pode levar à exigência de um processador operando a uma freqüência mais alta (e portanto consumindo mais potência) para atender os requisitos temporais do conjunto de tarefas da aplicação. Nos algoritmos dinâmicos, a prioridade das tarefas pode ser alterada dinamicamente pelo próprio RTOS. O algoritmo dinâmico mais conhecido é o EDF (*Earliest Deadline First*), no qual a tarefa que, num dado momento, deve concluir mais rapidamente sua execução recebe a máxima prioridade. Algoritmos dinâmicos garantem a escalonabilidade das tarefas até 100% de utilização da CPU, mas apresentam maior *overhead* em tempo de execução. A Figura 2.13 mostra um exemplo de escalonamento de tarefas segundo os algoritmos RMS e EDF.

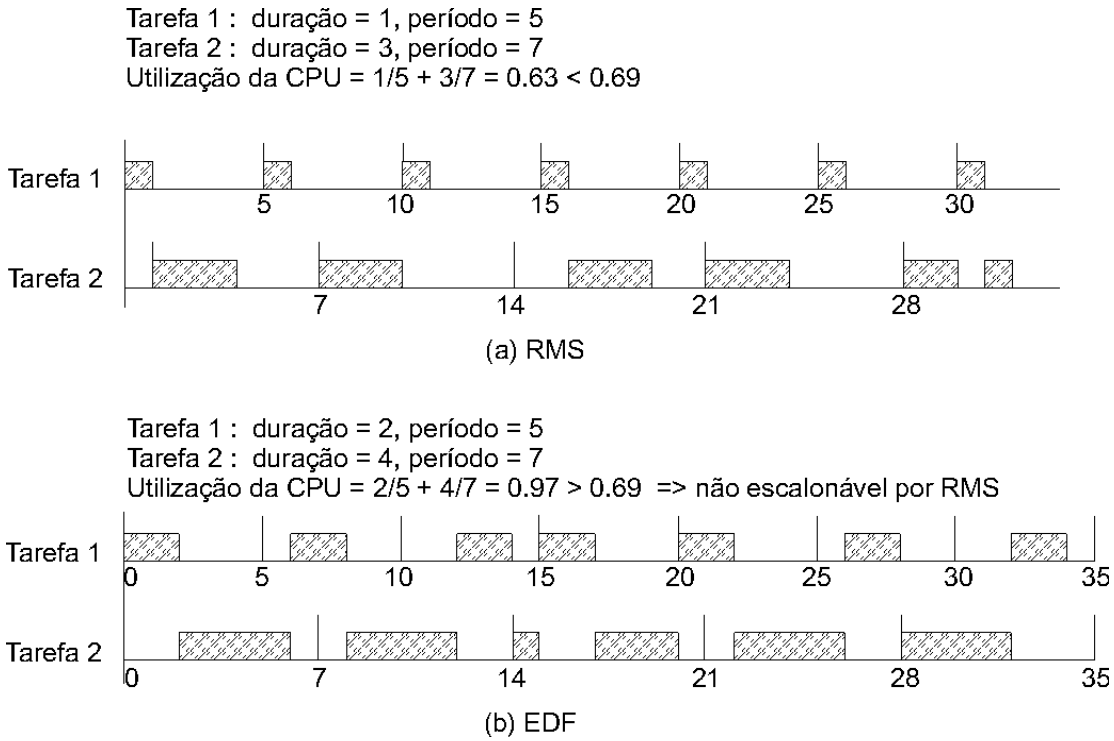


Figura 2.13. Algoritmos de escalonamento

Um problema associado ao escalonamento das tarefas é a necessidade de utilização de estimativas de tempo de execução das tarefas que correspondem ao pior caso (WCET – *Worst-Case Execution Time*), como forma de se garantir que os deadlines serão atendidos mesmo em situações extremas. Esta exigência é mais forte em sistemas de tempo real *hard*, onde a perda de *deadlines* pode levar a efeitos inaceitáveis (por exemplo em termos de segurança). Como conseqüência de um escalonamento baseado em WCET, pode-se também criar a necessidade de um processador operando a uma freqüência bem mais alta do que seria exigido considerando-se apenas os tempos típicos de execução da tarefa, que podem ser bastante inferiores ao WCET.

Métricas para a avaliação de RTOS incluem, entre outras, a latência das interrupções (tempo decorrido entre o pedido de interrupção e seu atendimento completo), o tempo de chaveamento de contexto, a resolução do relógio dos temporizadores e o tempo de execução das diversas chamadas do sistema.

RTOS dedicados a aplicações embarcadas [Li 1997, Stepner 1999] precisam atender outros requisitos, além daqueles naturalmente já exigidos por sistemas operacionais de tempo real. Em primeiro lugar, eles devem ser escaláveis, ou seja, não devem oferecer um conjunto completo de serviços de forma monolítica, mas sim como uma biblioteca de módulos, a serem facilmente selecionados no momento da geração da aplicação de acordo com as suas necessidades específicas. Em segundo lugar, eles devem atender restrições de projeto da aplicação. Uma destas restrições é justamente o tamanho da memória exigida, restrição esta já parcialmente atendida pela característica de escalabilidade. No entanto, adicionalmente o RTOS embarcado deve também considerar restrições de desempenho e consumo de potência.

Existe grande número de sistemas operacionais de tempo real comerciais, alguns deles desenvolvidos especificamente para sistemas embarcados, tais como VxWorks [WindRiver 2003], RTLinux [2003], Virtuoso [Transtech 2003] e eCos [Red Hat 2003]. A diversidade de produtos comerciais reflete claramente a grande diversidade de aplicações de tempo real e embarcadas, sem que exista uma grande padronização de requisitos que conduza à predominância de poucos produtos no mercado.

2.4.3. Reuso de software

Conforme já introduzido previamente, a geração do software aplicativo de uma aplicação embarcada seria extremamente facilitada caso fosse oferecida uma API (*Application Programming Interface*) padronizada, que abstraísse da aplicação todos os detalhes de mais baixo nível, não apenas do hardware, mas também do RTOS, conforme ilustrado na Figura 2.14. Esta API deve incluir todos os serviços oferecidos ao programa do usuário pelo RTOS, por exemplo para gerência de E/S e para comunicação entre processos. A existência desta API promove o reuso de todo o software aplicativo desenvolvido sobre a mesma, tornando-o independente das implementações possíveis desta API através de diferentes RTOS e plataformas de hardware. O software aplicativo torna-se assim automaticamente portável para diferentes plataformas. Além disto, esta API permite o projeto concorrente do software aplicativo e da plataforma do sistema, encurtando enormemente o tempo de projeto em relação à abordagem convencional, na qual o software aplicativo só pode ser desenvolvido e validado depois do projeto da plataforma.

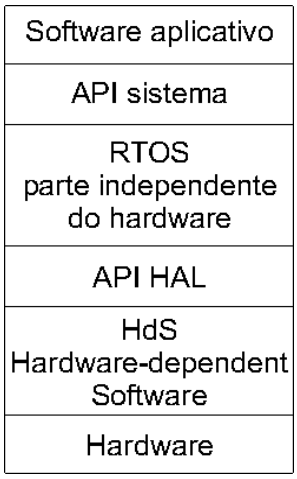


Figura 2.14. Interfaces padronizadas de software

A implementação do sistema embarcado, portanto, passa a corresponder ao projeto do RTOS e da plataforma de hardware de forma a implementar a API, atendendo simultaneamente aos requisitos da aplicação. Evidentemente, apenas os serviços disponibilizados pela API que são necessários para a aplicação específica precisam ser implementados.

Existem esforços de padronização da especificação dos serviços a serem oferecidos no contexto de RTOS para sistemas embarcados. Como a padronização é feita sobre uma especificação, isto permite o desenvolvimento de um SO dedicado para cada situação, já que ele não precisa implementar toda a especificação, atendendo-se assim a exigência de escalabilidade. Exemplos são as especificações OSEK [OSEK 2003], orientada ao domínio automotivo, e ITRON [Itron 2003], para produtos eletrônicos de pequeno porte, como câmeras digitais e aparelhos de fax.

No entanto, também na implementação do RTOS pode-se promover o reuso de software. O RTOS pode ser dividido em duas partes, uma delas dependente da plataforma de hardware (HdS – *Hardware-dependent Software*) e outra independente da mesma. Estas duas partes podem ser explicitamente separadas através de uma HAL (*Hardware Abstraction Layer*) [Yoo e Jerraya 2003], conforme mostrado na Figura 2.14. A HAL é uma API que oferece ao RTOS uma abstração do hardware, promovendo assim o reuso, para diferentes plataformas, do código do RTOS que é independente de hardware. Exemplos de HdS incluem código para inicialização (*boot*) do sistema, para chaveamento de contexto e para configuração e acesso a recursos de hardware, como MMU, barramentos, interfaces e temporizadores. Os acionadores (*drivers*) de dispositivos periféricos são em grande parte dependentes do hardware, sendo assim tipicamente oferecidos no nível da HAL. De forma similar à API no nível do software aplicativo, a HAL permite o projeto concorrente do hardware e de um RTOS mínimo e dedicado que atende aos requisitos da aplicação embarcada.

O conceito de HAL tem sido utilizado por diferentes RTOS, como WindowsCE (através dos BSP's – *board support packages*) [Microsoft 2003], eCos e RT-Linux. Mas não existem ainda padrões para a HAL. Recentemente, a VSIA [Shandle e Martin 2002] criou um grupo de trabalho visando o estudo de HdS e a padronização da HAL.

2.4.4. Síntese automática do RTOS

No projeto de um sistema embarcado, deve ser feita uma decisão entre o uso de um sistema operacional genérico ou sintetizado a partir das necessidades específicas de uma aplicação.

Conforme já mencionado, existe um grande número de sistemas operacionais de tempo real comerciais desenvolvidos especificamente para sistemas embarcados. Eles são de modo geral modulares, podendo portanto ser configurados de acordo com as necessidades de uma determinada aplicação. Sua flexibilidade permite que sejam portados para diferentes plataformas. Ainda assim, apesar desta configurabilidade, sua generalidade e a granularidade dos módulos resultam na geração de um SO com tamanho que pode ser excessivo em aplicações onde a área ocupada pela memória é um fator crítico. Além disto, o método de geração privilegia apenas uma certa diminuição no tamanho do código do SO, sem considerar a otimização de outros fatores críticos no projeto de sistemas embarcados, como desempenho e consumo de potência.

Outro problema crucial é a necessidade de adaptação do SO para cada nova arquitetura-alvo, além da adaptação do software aplicativo para esta combinação de SO e arquitetura (questão já discutida na Seção 2.3.6). No contexto de sistemas embarcados, cada sistema pode ter por exemplo diferentes organizações de memória (com diferentes mapas de endereço) e sistemas de E/S (diferentes estruturas de barramentos e tipos de periféricos, com diferentes endereços e níveis de interrupção), o que exige reconfiguração do SO. Usualmente, tanto a adaptação do SO como o mapeamento do software aplicativo são feitos manualmente, num processo demorado e propenso a erros.

Os motivos acima expostos justificam esforços para o desenvolvimento de métodos de síntese automática de um SO dedicado e que atenda requisitos da aplicação, em termos de desempenho e/ou consumo de potência e/ou área de memória.

A ferramenta TEReCS [Böke 2000] objetiva a síntese de um RTOS mínimo e dedicado, com ênfase nas estruturas de comunicação. O SO é construído a partir de uma biblioteca pré-definida de serviços denominada DREAMS. Os serviços disponíveis estão organizados nesta biblioteca através de relações de dependência. A partir de uma análise das necessidades de comunicação da aplicação, a ferramenta TEReCS seleciona os serviços imediatamente necessários para implementar esta comunicação, além dos demais serviços dos quais aqueles dependem, adicionando-os a um kernel mínimo (o zeroDREAMS). No ambiente ROSES [Gauthier 2001], está disponível uma biblioteca de elementos básicos, descritos numa linguagem de macros, que podem ser configurados e compostos para a geração automática de um RTOS. Esta abordagem também está fortemente voltada para a síntese automática da comunicação entre processos que podem estar alocados a diferentes processadores. Mas a ferramenta também gera automaticamente um escalonador de tarefas para cada processador. No projeto de um sistema de pequeno porte, é relatada a geração de SO com apenas 1,86 Kbytes de tamanho. As abordagens TEReCS e ROSES, no entanto, não consideram a otimização de desempenho e de potência.

Outra idéia que começa a ser explorada é a implementação em hardware de funções críticas de um RTOS [Mooney III e Blough 2002], visando aumento de desempenho ou atendimento de restrições temporais severas. Uma função que é boa candidata a uma implementação em hardware é o escalonador de tarefas, especialmente em aplicações críticas no tempo que apresentem um número grande de chaveamentos de contexto.

Na UFRGS, iniciou-se recentemente um trabalho de exploração do espaço de projeto de um RTOS dedicado, considerando restrições de desempenho, consumo de potência e ocupação de memória. Primeiros experimentos visaram a avaliação do impacto de diferentes mecanismos de comunicação [Gervini 2003] e escalonadores sobre estas métricas.

2.4.5. Reuso de componentes IP

Existe uma contradição no mercado atual de semicondutores, pois ao mesmo tempo em que a indústria de semicondutores possibilita a criação de sistemas completos em silício (SoC – *Systems-on-Chip*) através do avanço tecnológico, os projetistas não têm como lidar com a complexidade destes sistemas. Por exemplo, em 2001 a produtividade dos projetistas, medida em transistores por homem-mês, era de aproximadamente 2 mil. Os recursos tecnológicos, contudo, permitiam a integração de 100 milhões de transistores com a mesma tecnologia do período [Magarshack 2002]. Um circuito de um milhão de transistores levaria 500 homens-mês para ser implementado, ou aproximadamente 42 homens trabalhando durante um ano.

A solução para o problema de projeto passa pelo reuso de componentes pré-projetados, já testados em outro projetos. O reuso, além de possibilitar um menor tempo de projeto (já que a maioria dos módulos estaria pronta a ser usada), possibilitou um mercado de vendas de núcleos de hardware para serem utilizados por diferentes clientes, mantendo a propriedade intelectual do fornecedor do núcleo. Estes módulos são chamados de blocos IP (*Intellectual Property*).

Nos últimos tempos houve portanto um deslocamento do problema de projeto. Se antes a validação de um bloco e a automação do projeto do mesmo passavam por uma descrição no nível de transferência de registradores (RTL) e posterior síntese, hoje os grandes projetos são feitos através da escolha adequada dos componentes que deverão fazer parte do SoC.

A escolha adequada de quais componentes utilizar não é no entanto suficiente. A integração de diversas partes feitas por diferentes fabricantes é um problema conhecido da comunidade de engenharia, já que o projeto de placas vem sendo feito com este estilo há muito tempo. Contudo, ao se colocar um conjunto de blocos pré-projetados em uma mesma pastilha de silício, os problemas de projeto em nível sistema tornam-se mais prementes.

Em especial, a arquitetura final do sistema, o desempenho e a potência que se devem obter, os tipos de interface, a comunicação entre os módulos, o *floorplanning* global que permita a correta distribuição de sinais elétricos importantes como o relógio e ainda outras questões passam a ser fundamentais para o projeto de SoCs.

Por exemplo, a Figura 2.15 apresenta uma arquitetura de SoC possível de ser encontrada em várias aplicações, desde um telefone portátil até o controle de um sistema automotivo. Portanto, do ponto de vista do sistema, é muito importante não somente o reuso dos blocos, mas o que esperar de cada um, como colocá-lo no sistema de modo a manter o desempenho global, como garantir este desempenho global e a possibilidade de alterações a posteriori, já que quanto mais complexo um sistema, maiores as chances que ele tenha de ser adequado a diferentes contextos.

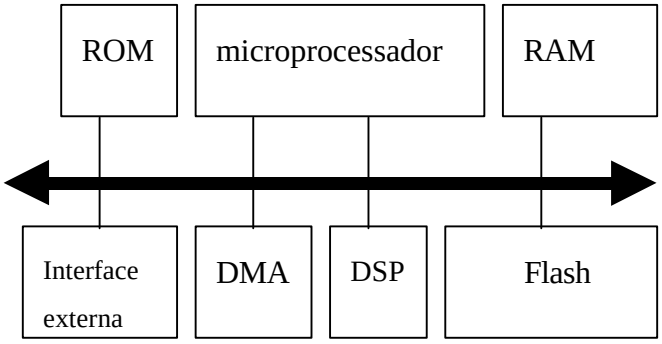


Figura 2.15. Arquitetura facilmente encontrável em múltiplos projetos

No sistema descrito através da Figura 2.15 pode-se observar a presença de ao menos dois microprocessadores, já que a maioria dos sistemas tem de executar comportamentos muito diferentes, dependendo do modelo de computação adequado. É certo também que sistemas em silício farão uso abundante de memórias, para que estas forneçam código e dados para os diferentes processadores presentes no sistema. Circuitos de acesso a periféricos como controladores de DMA, controladores de disco em memórias flash e interfaces com o mundo real (analógico) devem também estar presentes.

2.4.6. Projeto baseado em padrões de barramentos e de interfaces de núcleos

A integração de componentes IP em uma plataforma arquitetural é extremamente facilitada se esta plataforma for baseada em padrões para diversos de seus aspectos. Uma padronização importante diz respeito à infra-estrutura de comunicação, em torno da qual devem estar conectados os componentes de hardware.

Numa abordagem de projeto baseado em barramentos, os componentes estão interconectados por um ou mais barramentos, os quais estão por sua vez conectados entre si através de *pontes*. Como a especificação funcional, física e elétrica de um barramento pode ser padronizada, bibliotecas de componentes IP cujas interfaces se ajustam diretamente a este padrão podem ser desenvolvidas, permitindo que o projeto da micro-arquitetura do sistema seja feita facilmente através da conexão direta dos componentes ao barramento. Esta abordagem tem sido adotada por diversas empresas que oferecem soluções para o projeto de sistemas embarcados, tais como IBM, com o padrão CoreConnect [IBM 2003], ARM, com o padrão AMBA [ARM 1999], e Sonics, com o padrão Silicon Backplane MicroNetwork [Sonics 2003]. Cada uma destas empresas oferece, além do barramento padronizado, também ricas bibliotecas de componentes e ambientes de desenvolvimento de sistemas baseados nestes barramentos. Mesmo componentes cujas interfaces não sejam compatíveis com o padrão podem ser reutilizados no projeto, desde que sejam desenvolvidos adaptadores de interfaces, conforme já discutido na Seção 2.3.7.

A Figura 2.16 mostra a arquitetura típica de um sistema construído em torno do padrão AMBA (*Advanced Microcontroller Bus Architecture*). Três barramentos distintos são definidos no padrão. O barramento AHB (*Advanced High-performance Bus*) é o principal, ao qual estão conectados o processador central do sistema, memória *on-chip* e dispositivos de acesso direto à memória (DMA), e que sustenta a largura de banda exigida para transferências de dados de/para uma memória externa de grande capacidade. Usualmente, o processador principal do sistema é o microcontrolador ARM, bastante utilizado em aplicações embarcadas e consistente com o padrão AMBA. Um processador secundário também poderia ser conectado a este barramento. Para atingir alto desempenho, este barramento apresenta características como transferências de dados em rajada, transações *split* e larguras de dados de 64 ou 128 bits. Ele permite um ou vários mestres (processadores e dispositivos de DMA). Escravos típicos são as memórias e uma ponte para um barramento APB (*Advanced Peripheral Bus*). Um arbitrador garante que apenas um mestre poderá iniciar transferências de dados a cada momento. Um decodificador de endereços gera um sinal de seleção para o escravo endereçado por uma transferência. O APB é um barramento secundário destinado a conectar um número arbitrário de periféricos que exijam pequena largura de banda para transferência de dados. Aproveitando sua simplicidade, ele é também projetado para baixo consumo de potência. O terceiro padrão do barramento, ASB (*Advanced System Bus*), é uma versão anterior do AHB, com menor número de funcionalidades.

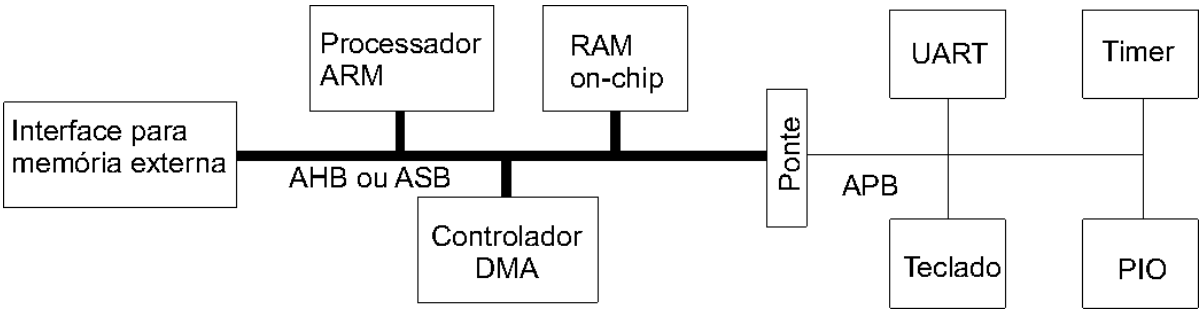


Figura 2.16. Arquitetura típica de sistema baseado no padrão de barramento AMBA

Numa abordagem distinta, interfaces de componentes podem seguir um padrão que é independente de barramentos e que permite a conexão direta de um componente a outro. Esta abordagem é seguida pelos padrões VCI (Virtual Component Interface), da VSIA (*Virtual Socket Interface Alliance*) [VSIA 2003], um consórcio de empresas cujo objetivo é a definição de padrões em diversas áreas do projeto de sistemas embarcados, e OCP (*Open Core Protocol*), originalmente desenvolvido pela empresa Sonics e agora promovido pela organização OCP-IP [OCP 2003]. Esta abordagem de projeto é baseada em *núcleos* (ou, do inglês, *cores*), por oposição à abordagem baseada em barramentos. Componentes consistentes com um destes padrões de interface também podem ser conectados através de um barramento, numa situação que exige a inclusão de adaptadores entre a interface do componente e o barramento. Esta é a abordagem adotada pela empresa Sonics, por exemplo, onde componentes com interfaces OCP são conectados a um barramento Silicon Backplane MicroNetwork através de adaptadores padronizados.

O padrão OCP, por exemplo, prevê um amplo espectro de funcionalidades, que podem ser agregadas seletivamente a um componente, de modo que este terá a interface com a complexidade estritamente necessária para a aplicação específica. Assim, na sua forma mais simples o padrão prevê unicamente os sinais de interface necessários para leituras e escritas isoladas de dados, através de um protocolo simples de tipo *handshake*. Em cada conexão um componente é considerado mestre e outro escravo. Um componente que opere como mestre e como escravo precisará sinais duplicados. Extensões neste protocolo básico permitem transferências em rajada, transferências em pipeline e até transferências concorrentes de múltiplas *threads*. Extensões complementares são definidas para interrupções, teste e reinicialização de componentes. Note-se que este tipo de padrão é orientado para conexões ponto-a-ponto entre componentes, não dispondo de funcionalidades para conexões múltiplas, como arbitramento e decodificação de endereços. Estes recursos precisam ser adicionados pelo projetista no caso de conexão de componentes OCP a um barramento.

2.4.7. Exemplo de plataforma de hardware

A Philips Semiconductors desenvolveu a plataforma Nexperia-DVP (Digital Video Processor) para o projeto de sistemas destinados ao mercado de *set-top boxes* e televisão digital. O SoC Viper (ou PNX8500) [Dutta 2001], desenvolvido sobre esta plataforma, é ilustrado na Figura 2.17. Contendo até 5 processadores, além de diversos blocos de hardware especializados, conectados em torno de dois barramentos, ele permite o controle de interações com o usuário e o processamento concorrente de fluxos de dados multi-mídia contendo áudio e vídeo.

Os dois processadores principais são um MIPS PR3940, de tipo RISC, e um TriMedia TM32, de tipo DSP-VLIW, ambos de 32 bits. O TM32 foi desenvolvido na própria Philips, tem sido utilizado desde 1996 em diversos sistemas multi-mídia e dispõe de um pequeno kernel de

RTOS. Seu conjunto de instruções é otimizado para um SoC que deva processar fluxos de áudio e vídeo. Até 5 operações podem ser executadas em paralelo pelo TM32, graças a sua filosofia VLIW. O PR3940 é um processador de baixo consumo de potência e alto desempenho, destinado a executar o sistema operacional e o software aplicativo do provedor de serviços, além de tratar diversas tarefas de controle do Viper e de realizar algum processamento gráfico. Cabe a ele solicitar tarefas de tratamento de imagens do processador TM32. Existem diversos sistemas operacionais portados para o processador MIPS, como VxWorks, Linux e WindowsCE. Ambos os processadores principais possuem caches internas de dados e de instruções.

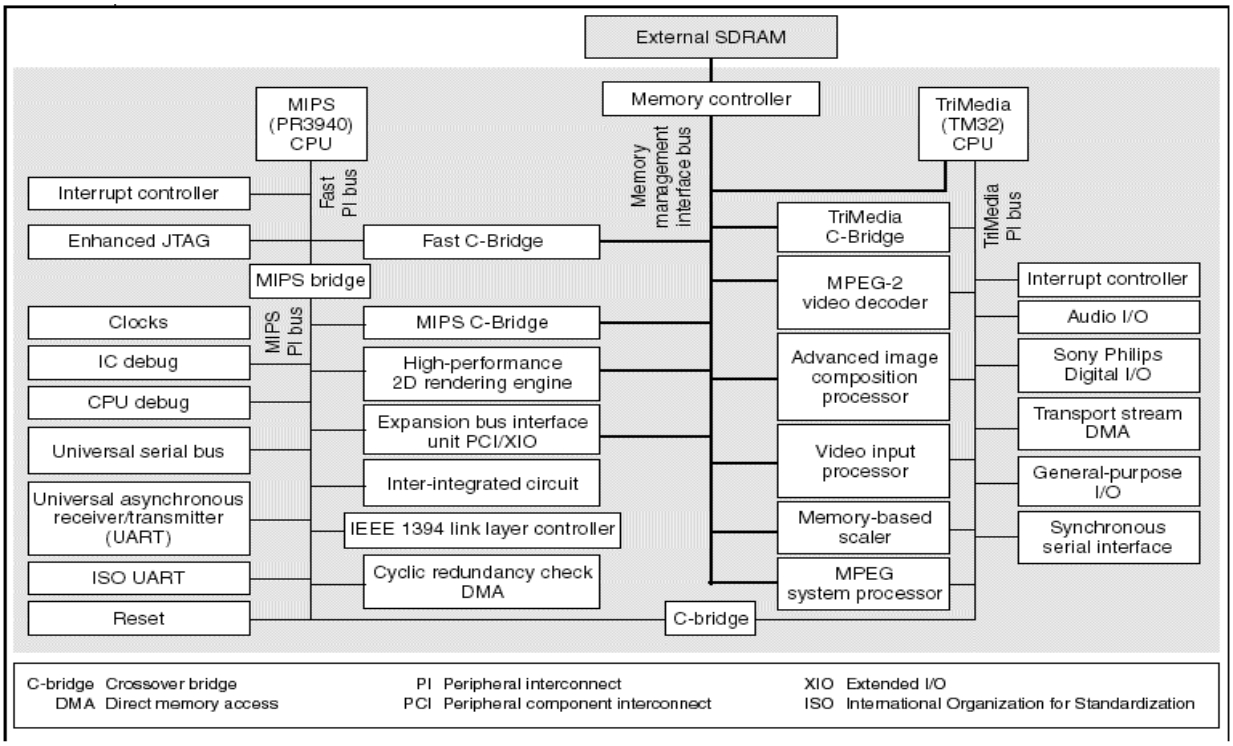


Figura 2.17. Arquitetura do Viper [Dutta 2001]

O Viper também contém até 3 processadores RISC de 16 bits, dedicados para filtragem, decodificação e demultiplexação de fluxos MPEG-2. Além dos processadores, o Viper possui diversos blocos de hardware para processamento dedicado, tais como um acelerador gráfico para *rendering* 2D, um processador capaz de combinar imagens, um decodificador de vídeo MPEG-2 de alta definição e dois processadores de entradas de vídeo em padrões NTSC e PAL. Afora estes processadores dedicados, o Viper inclui um amplo repertório de interfaces analógicas e digitais. Exemplos são um controlador USB, 3 interfaces UART, duas interfaces I^2C multi-mestre, uma interface serial síncrona, uma interface para recepção de comandos em infra-vermelho e 3 entradas e 3 saídas de áudio. Finalmente, a arquitetura ainda contém outros blocos necessários, como um controlador do barramento PI, um controlador de interrupção para cada processador principal e um módulo padrão EJTAG para oferecer portas de depuração do processador MIPS.

O sistema está organizado em torno de dois barramentos PI, associados aos dois processadores principais e destinados principalmente a operações de controle. Uma ponte conecta os dois barramentos e permite que diversos blocos dedicados e interfaces sejam acessados também pelo outro processador, mas em mais baixa prioridade. Além destes, existe

ainda um barramento MMI, destinado a conexões DMA entre periféricos de alta velocidade e a memória principal externa à pastilha do Viper.

Apesar do reuso de grande número de componentes IP internos à própria Philips, foi necessário o projeto de diversos novos componentes, para o que foi definido um padrão de desenvolvimento, visando facilitar o trabalho de validação. O circuito final contém 35 milhões de transistores, em tecnologia 0.18 µm, com 6 camadas de metal, e consome 4.5 W.

2.5. Estudo de caso

As propostas ad -hoc tradicionais utilizadas no desenvolvimento de sistemas embarcados não têm sido suficientes para tratar o crescimento da complexidade das novas aplicações. O desejável é realizar a descrição do sistema no mais alto nível de abstração possível, seguida de passos de validação e com posterior síntese automática da especificação em alto nível.

Algumas necessidades como área do circuito, desempenho, baixo consumo de potência e tamanho de software devem ser consideradas para se obter um bom resultado final. Por outro lado, em alguns casos o tempo necessário para o produto atingir o mercado torna-se mais importante que os itens tecnológicos. Assim, o projetista deseja obter um bom equilíbrio entre os fatores mencionados acima e também reduzir tempo de projeto.

Recentemente, a linguagem Java tornou-se uma alternativa às linguagens tradicionais no desenvolvimento de sistemas embarcados. O projetistas adotaram Java devido, principalmente, à portabilidade e reuso de código que a linguagem fornece às suas aplicações [Mulchandani 1998].

O desenvolvimento de aplicações Java para sistemas embarcados é uma questão ainda em aberto. A maioria das soluções propostas visa um ambiente no qual existem recursos suficientes para acomodar um sistema operacional de tempo real, uma implementação específica da máquina virtual Java (MVJ) [Lindholm e Yellin 1997], suporte a múltiplas linhas de execução e mecanismo de *garbage collection* [Barr 1998, Mrva 1998]. Porém, existe uma carência de metodologias que tornem a tecnologia disponível a ambientes com apenas algumas dezenas de Kbytes de memória e restrições de consumo de potência.

Processadores como o PicoJava [McGhan 1998] foram projetados para obter alto desempenho, não podendo competir no mercado de aplicações com restrições de recursos e potência, nas quais os microcontroladores são a melhor opção.

Sendo assim, para o desenvolvimento de aplicações embarcadas simples foi introduzida a metodologia baseada em um ambiente para geração de aplicações específicas baseadas em software e hardware para microcontroladores, denominado SASHIMI [Ito 2001] (Systems as Software and Hardware In Microcontrollers), que utiliza a linguagem Java como tecnologia fundamental para o projeto. Neste ambiente, o projetista fornece uma aplicação Java que será analisada e otimizada para execução em um microcontrolador denominado FemtoJava, capaz de executar instruções Java nativamente, exibindo ainda características de um ASIP (Java *Application Specific Instruction Set Processor*), adicionalmente podendo-se incluir um ASIC (*Application Specific Integrated Circuit*), onde ambos serão sintetizados em um único FPGA. Esta abordagem é caraterizada por uma alta integração, um ambiente simples de execução, nenhum desenvolvimento de compiladores, compatibilidade de software e um reduzido tempo de projeto.

2.5.1. O ambiente SASHIMI

SASHIMI é um ambiente destinado à síntese de sistemas microcontrolados especificados em linguagem Java. O ambiente SASHIMI utiliza as vantagens da tecnologia Java e fornece ao projetista um método simples, rápido e eficiente para obter soluções baseadas em hardware e software para microcontroladores. O conjunto de ferramentas disponível no SASHIMI também foi desenvolvido inteiramente em Java, tornando-o altamente portável para diversas plataformas.

O fluxo de projeto definido para o SASHIMI, desde o código fonte da aplicação até o chip do microcontrolador sintetizado [Ito 2000] é apresentado na Figura 2.18. No ambiente SASHIMI o usuário inicia o projeto através da modelagem da aplicação utilizando a linguagem Java. Durante esta fase do processo, o desenvolvimento segue o ciclo tradicional de implementação-compilação- execução em um sistema convencional (computador pessoal ou estação de trabalho) utilizando o ambiente JDK correspondente.

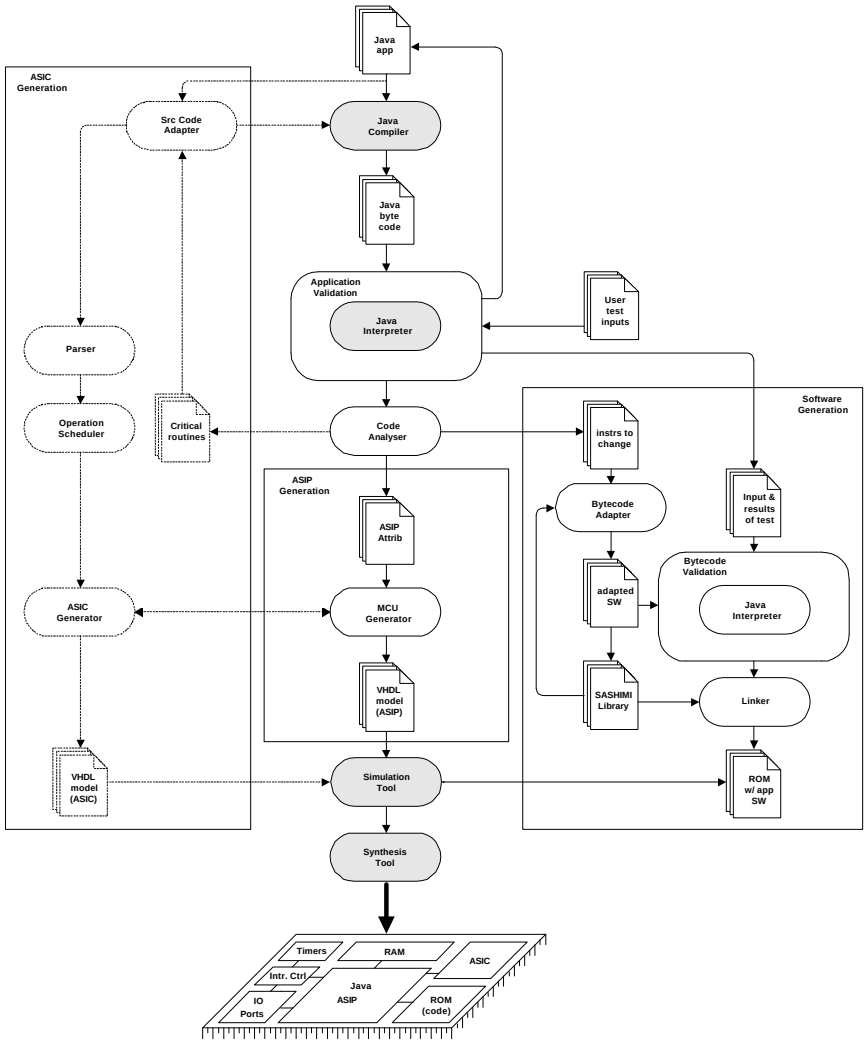


Figura 2.18. Fluxo de Projeto SASHIMI

Neste contexto, a execução é equivalente à simulação da aplicação no hardware ainda não disponível, emulado pelo interpretador Java e pelas classes adicionais do ambiente. Durante a fase de simulação, o projetista pode usar classes pré-definidas (na qual *threads* e outros recursos são permitidos) para modelar o comportamento de dispositivos periféricos necessários. Atualmente estão disponíveis classes que simulam o comportamento de dispositivos como LCDs, teclados, displays de 7 segmentos, conversores AD/DA, etc. Durante o processo de síntese, tais classes serão substituídas pelo código responsável por prover a interface entre o microcontrolador e os componentes reais especificados.

Quando o projetista considera que a aplicação atende às especificações funcionais, os vetores de teste fornecidos pelo usuário durante a simulação e os resultados obtidos durante este processo são armazenados para serem utilizados pela etapa de validação de bytecodes.

A ferramenta de análise de código tem como entrada o código binário executável (arquivos *class*) e realiza análises para extrair informações de desempenho e área ocupada pelo hardware. É necessário ter em mente que o conjunto de instruções Java não é suportado de forma completa, e a análise da aplicação deve prover informações necessárias à geração do ASIP e à adaptação de código.

A adaptação de bytecodes envolve transformações nos arquivos *class* da aplicação, mantendo as propriedades semânticas da mesma. Este processo deve transformar instruções complexas (e.g. *tableswitch*, *lookupswitch*) em uma seqüência de instruções suportadas pelo microcontrolador. Bytecodes desnecessários à aplicação (e.g. chamadas ao método *System.out.println* usados durante a simulação) podem ser descartados, de acordo com as informações fornecidas pelo analisador de código.

Para validar as modificações efetuadas uma nova fase de simulação é necessária. Entretanto, a disponibilidade do conjunto de dados armazenados durante a primeira execução da aplicação possibilita que este processo possa ser realizado de forma automatizada. Os critérios que guiam a modificação de bytecodes são as instruções suportadas pelo FemtoJava, os requisitos de área e desempenho, a taxa de utilização de cada instrução e o tamanho da memória disponível para armazenar o software da aplicação.

O produto obtido neste ponto do fluxo de projeto constitui-se apenas de um conjunto de arquivos *class* adaptados para serem compatíveis com um conjunto de instruções específico que será implementado no microcontrolador FemtoJava. A próxima etapa remove todas as informações desnecessárias (p.ex. estruturas que armazenam informações de depuração), realiza a resolução da hierarquia de classes, efetua a ligação e converte o código da aplicação e o código de sistema (bibliotecas para ter acesso aos recursos de hardware e para execução de instruções complexas) em uma única descrição de ROM.

2.5.2. Regras de modelagem

No ambiente SASHIMI a modelagem do sistema deve ser feita em linguagem Java, respeitando as restrições que serão apresentadas. Dentre as tarefas automatizadas disponíveis no ambiente SASHIMI está a verificação da aplicação com vistas às regras de modelagem.

O domínio das aplicações atualmente sintetizáveis pelo ambiente é restrito a aplicações embarcadas clássicas como controle de portas automáticas, sistemas de segurança e identificação, etc. A natureza das aplicações-alvo induz a algumas restrições no estilo de

codificação. Tais restrições servem a propósitos similares àqueles que tornam o subconjunto sintetizável de VHDL menor do que a linguagem completa, originalmente concebida com propósitos de simulação. Portanto, uma aplicação Java para ser sintetizável no ambiente SASHIMI deve respeitar as seguintes condições:

- o operador *new* não é permitido, pois seria necessário prover serviços de gerenciamento de memória virtual;
- apenas métodos e variáveis estáticas são permitidos, salvo os métodos das interfaces providas pela API do ambiente SASHIMI e implementadas pela classe sendo modelada;
- métodos recursivos não são permitidos, pois seriam necessários mecanismos para gerenciamento dinâmico de memória;
- interfaces não são suportadas, dado que associações dinâmicas (*binding*) representam um custo adicional em tempo de execução;
- dados do tipo ponto-flutuante não podem ser utilizados na versão atual. Contudo, o suporte a esse tipo de dados pode ser obtido através da disponibilidade de FPGAs de maior capacidade ou através de uma biblioteca adequada;
- múltiplas *threads* não são sintetizadas, visto que grande parte das aplicações microcontroladas podem ser implementadas em um único fluxo de execução, minimizando custos de projeto e hardware;
- o comportamento da aplicação é modelado a partir do método *initSystem*, sendo o método *main* e o construtor utilizados para efeito de inicialização de outros componentes do sistema.

O modelo de simulação obtido observando-se as regras descritas acima permite que o mapeamento para o modelo de implementação seja amplamente simplificado. As transformações realizadas na aplicação estarão restritas àquelas dependentes dos parâmetros de síntese, como a complexidade das instruções que podem determinar a área em FPGA e a freqüência de operação.

O modelo de simulação pode conter todo o comportamento do sistema, incluindo a comunicação através de portas de E/S, temporizadores e tratamento de interrupção, encapsulados em classes disponíveis no ambiente. Ao projetista não é necessário o conhecimento da estrutura desses mecanismos, da programação ou da linguagem de montagem necessária para construir o código de interface entre a aplicação e o restante do sistema.

2.5.3. Análise e geração de estimativas

O processo de análise da aplicação serve a múltiplos propósitos no presente trabalho, englobando a verificação da consistência do modelo do sistema em relação às regras estabelecidas para a modelagem, a identificação de instruções não suportadas pelo microcontrolador e estimação de resultados de síntese. Dentre as estimativas geradas se incluem a área ocupada em FPGA, freqüência máxima de operação do microcontrolador, tamanho do código da aplicação e quantidade de RAM utilizada pela aplicação. A análise dos bytecodes da aplicação pode se processar de forma estática ou dinâmica (*profiling*), servindo basicamente à verificação das regras de modelagem e obtenção de estimativas de resultados mais precisos, respectivamente.

O produto do processo de análise deve ser suficiente para dirigir o processo de adaptação e síntese do software da aplicação e do microcontrolador. Quando as estimativas de desempenho não satisfazem os requisitos determinados para a aplicação, partes críticas do código podem ser

identificadas pela ferramenta de análise de código, e o projetista pode optar por efetuar uma substituição de instruções mais ou menos agressiva. Alternativamente, o projetista pode prover a especificação de um ASIC, ou efetuar a sua geração diretamente a partir de Java utilizando as ferramentas do SASHIMI, para ser integrado ao hardware do sistema e obter os resultados desejados.

2.5.4. Adaptação e síntese do software da aplicação

A adaptação do código da aplicação reflete o conjunto de instruções suportado pelo microcontrolador FemtoJava e as opções de síntese especificadas pelo projetista. O produto da adaptação de código compõe-se de um conjunto de arquivos *class* Java válidos, que virtualmente devem oferecer a mesma funcionalidade do código original.

Basicamente, o processo de adaptação é bastante flexível, permitindo a substituição de seqüências de instruções, remoção e inserção de métodos e cópia de métodos de uma classe à outra. O ajuste das referências de código (instruções de desvio) é realizado de forma automática a cada modificação, e as informações de crescimento da pilha referentes a cada método são devidamente atualizadas.

A adaptação da aplicação mantendo sua consistência com a MVJ permite realizar a simulação do software adaptado validando a funcionalidade do novo modelo. O processo descrito somente é possível graças à disponibilidade de um pacote de classes especificamente desenvolvido para o ambiente SASHIMI (porém não limitadas à utilização neste projeto), cujas principais bibliotecas são *ClassFile*, *ClassFileInputStream* e *ClassFileOutputStream*.

A síntese do software da aplicação é a fase imediatamente posterior à validação do modelo modificado do sistema. O processo de síntese do software é relativamente simples. Ao final do processo, o código da aplicação é convertido pela ferramenta de síntese de software em um modelo de ROM descrito em linguagem VHDL ou outro formato de arquivo adequado à ferramenta de simulação e/ou síntese.

2.5.5. Geração do microcontrolador

O hardware resultante do sistema SASHIMI é composto essencialmente por um microcontrolador Java (FemtoJava) dedicado à aplicação modelada, cuja operação é compatível com a operação da Máquina Virtual Java. As informações extraídas na etapa de análise de código permitem determinar um conjunto de instruções, quantidade de memória de programa e de dados, tamanho da palavra de dados, e demais componentes adequados aos requisitos da aplicação alvo, podendo o projetista interferir nos resultados desse processo.

O modelo do microcontrolador resultante é descrito em linguagem VHDL e sintetizável por ferramentas como Maxplus-II da Altera Corporation [Altera 2003]. A principal adaptação da arquitetura do microcontrolador consiste em modificar o mecanismo de decodificação de forma a reconhecer apenas o subconjunto de instruções contidas na aplicação. Em função dos diferentes formatos e da complexidade de algumas instruções Java, o hardware de decodificação se torna proporcionalmente complexo, de forma que um menor número de instruções suportadas permite uma economia significativa de recursos (células lógicas em FPGA), possibilitando ainda a integração de outros componentes no mesmo chip.

A geração de ASICs é uma alternativa que fornece ainda mais flexibilidade ao método proposto, pois possibilita a inclusão de circuitos específicos que permitam acelerar a execução da aplicação. Neste processo, o próprio projetista pode fornecer o modelo VHDL do ASIC ou

optar pela geração automática, a partir da identificação dos métodos cuja execução contribua significativamente para o tempo total de execução da aplicação.

Os modelos do microcontrolador, do código da aplicação e dos ASICs gerados podem ser simulados conjuntamente e depurados utilizando ferramentas de simulação comercialmente disponíveis, e posteriormente sintetizados.

Entretanto, é importante ressaltar ainda que os componentes utilizados durante a simulação e modelados como parte do sistema externo ao microcontrolador são removidos pelas ferramentas do ambiente SASHIMI. Contudo, são incluídas todas as rotinas que implementam a interface com tais componentes, de acordo com o comportamento modelado. Portanto, o projetista pode facilmente dispor de componentes reais e realizar a montagem final do sistema.

2.5.6. O microcontrolador FemtoJava

O microcontrolador FemoJava [Ito 1999] é um microcontrolador baseado na arquitetura de pilha com capacidade de execução nativa de bytecodes Java. Suas principais características são um reduzido conjunto de bytecodes, arquitetura Harvard (memórias separadas de dados e instruções), pequeno tamanho e facilidade de inclusão e remoção de instruções.

A arquitetura do microcontrolador FemtoJava foi concebida mediante um detalhado estudo do conjunto de instruções Java e da arquitetura da MVJ [Lindholm e Yellin 1997, Venners 1998, Meyer e Downing 1997]. A Figura 2.19 apresenta a microarquitetura do FemtoJava.

Além do microcontrolador ser composto por uma unidade de processamento baseada em arquitetura de pilha, possui memórias RAM e ROM integradas, portas de entrada e saída mapeadas em memória e um mecanismo de tratamento de interrupções com dois níveis de prioridade. A arquitetura da unidade de processamento implementa um subconjunto de 66 instruções da Máquina Virtual Java, e seu funcionamento é consistente com a especificação da MVJ. A utilização de registradores para armazenar elementos da pilha possibilita ainda ganho de desempenho, redução na área ocupada em FPGA e o processamento realizado simultaneamente aos acessos à memória.

É importante ter-se em mente que alguns detalhes da micro-arquitetura foram projetados considerando justamente o FPGA como o componente alvo para síntese. Portanto, possivelmente os resultados obtidos no processo de síntese podem sofrer sensível impacto se forem realizadas otimizações na micro-arquitetura, visando outra tecnologia como *standard-cell* ou mesmo FPGAs de outros fabricantes.

A execução de apenas uma aplicação com uma única *thread* no microcontrolador permite que diversas simplificações sejam efetuadas. A carga dinâmica de classes pode ser substituída por ferramentas que realizam a resolução das referências contidas no código em tempo de projeto. Tais ferramentas identificam todas as classes da aplicação, armazenando-as na memória de programa juntamente com as bibliotecas específicas para o acesso ao hardware. Grande parte das áreas de memória definidas pela arquitetura da MVJ não são utilizadas, restando apenas a área de métodos (memória de programa) e uma única pilha Java para execução da aplicação.

A interface com métodos nativos utilizada pela tecnologia Java também é desnecessária, desde que tal função é realizada pelas bibliotecas armazenadas juntamente com a aplicação. O último e principal componente deste modelo é o microcontrolador Java como mecanismo de

execução de bytecodes em hardware, contendo instruções adicionais para realização de funções não definidas pela especificação da MVJ.

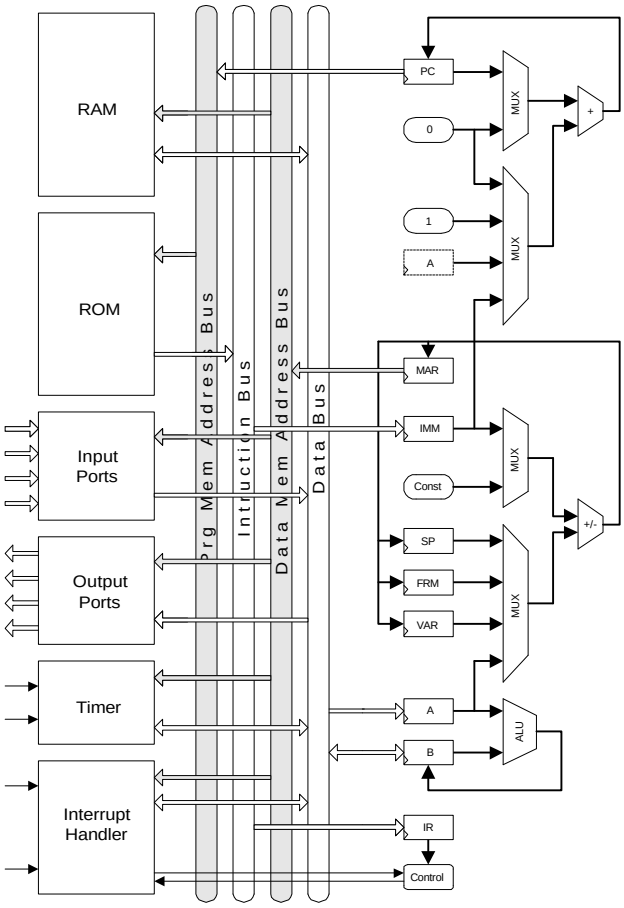


Figura 2.19. Microcontrolador FemtoJava

2.5.7. Resultados práticos

Nesta seção são apresentados os resultados de síntese para algumas aplicações especificadas em Java e sintetizadas utilizando a Ferramenta SASHIMI. As aplicações são as seguintes:

- *Biquad* é um filtro biquadrático;
- *ECS* corresponde a *Elevator Control System* e implementa um algoritmo para o controle de um elevador simples;
- *Podos* é um algoritmo utilizado em um dispositivo eletrônico capaz de medir a distância percorrida por um pessoa caminhando ou correndo;
- *Tradutor* é um algoritmo de busca implementado através de uma tabela *hash* que realiza traduções de palavras da língua inglesa para portuguesa;
- *Caneta* é um dispositivo portátil usado como uma caneta tradutora. É composto por um algoritmo que implementa uma rede neural que reconhece um conjunto de caracteres e um algoritmo de busca que executa traduções de palavras da língua inglesa para a portuguesa.

A Tabela 2.2 apresenta os resultados de área, em termos de células lógicas, e freqüência para os microcontroladores obtidos a partir das diferentes aplicações. Todos os processadores foram sintetizados no mesmo FPGA, o EPF10K30R240-4 (Flex 10K30) da Altera Corporation [Altera 2003]. A tabela apresenta os resultados, tanto para o microcontrolador FemtoJava completo (com todo o seu conjunto de 68 instruções implementado), como também para as diferentes aplicações nas versões de 8 bits e 16 bits.

Pela Tabela 2.2 pode-se observar, para cada aplicação, que o número necessário de diferentes instruções é sensivelmente menor que o número de instruções disponível no FemtoJava. Assim, eliminando-se as instruções desnecessárias para determinada aplicação, obtém-se uma significativa redução na área ocupada pelo microcontrolador e também um aumento de freqüência.

Tabela 2.2. Resultados de síntese dos microcontroladores para as diferentes aplicações.

μ controlador	Aplicação	Área FPGA (L.C.)	Freq (MHz)	Mem. Prg. (bytes)	Mem. Dados (bytes)	≠ Instr
8 bits	Completo	1481	4,85	-	-	68
	Biquad	991	7,97	49	30	22
16 bits	Completo	1979	3,93	-	-	69
	ECS	1556	5,65	612	74	31
	PODOS	1465	5,55	246	90	29
	Tradutor	1253	5,13	280	118	32
	Caneta	1698	6,19	815	310	35

A Tabela 2.2 também apresenta os resultados de software para as diversas aplicações, apresentados em termos de memória de dados e memória de programa necessárias para cada aplicação.

Atualmente, a ferramenta vêm recebendo diversas melhorias, com a inclusão de novas facilidades para modelagem, simulação e síntese, como também com relação à interação com o usuário. Trabalhos atuais comprendem a realização do microcontrolador FemtoJava para suportar *multithreading* e alocação dinâmica de objetos, como também a implementação de características de baixo consumo de potência no microcontrolador, entre outras modificações arquiteturais.

Referências bibliográficas

Alexander, P. e Kong, C. (2001). “Rosetta: Semantic Support for Model-Centered Systems-Level Design”. IEEE Computer, Vol. 34, No. 11, Nov. 2001. pp 64-70.

Altera (2003). "Maxplus-II". http://www.altera.com.

Altman, E., Ebcioglu, K., Gschwind, M. e Sathaye, S. (2001). "Advances and Future Challenges in Binary Translation and Optimization". IEEE Proceedings, Vol. 89, No. 11, Nov. 2001. pp 1710-22.

ARM (1999). “AMBA Specification Rev2.0”. ARM Limited, Mar. 1999. Disponível em: www.arm.com/armwww.nsf/

Bacivarov, I., Yoo, S. e Jerraya, A.A. (2002). "Timed HW-SW Cosimulation Using Native Execution of OS and Application SW". HLDVT'02 – 7th IEEE International High-Level Design Validation and Test Workshop, Cannes, França, Out. 2002. Proceedings, 2002.

Barr, M. (1998). "Free Java Virtual Machine for Embedded Systems". Embedded System Conference, San Francisco, 1998. Proceedings, Miller Freeman, pp 277-288.

Beck Filho, A.C., Wagner, F.R. e Carro, L. (2003). "CACO-PS: A General Purpose Cycle-Accurate Compiled-Code Power Simulator". 16th Symposium on Integrated Circuits and Systems Design. São Paulo, Brasil, Set. 2003. Proceedings, IEEE Computer Society Press, 2003.

Benini, L., Macii, E. e Poncino, M. (1999). "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems". International Symposium on Low Power Electronics and Design, San Diego, EUA, Ago. 1999. Proceedings, ACM, 1999. pp 206-211.

Benini, L., Macii, A., Macii, E. e Poncino, M. (2000). "Increasing Energy Efficiency of Embedded Systems by Application-specific Memory Hierarchy Generation". IEEE Design & Test of Computers, Vol. 17, No. 2, Abr/Jun. 2000. pp 74-85.

Bergamaschi, R.A. et alii (2001). "Automating the Design of SOCs Using Cores". IEEE Design & Test of Computers, Vol. 18, No. 5, Set/Out. 2001. pp 32-45.

Bhattacharya, S., Leupers, R. e Marwedel, P. (2000). "Software Synthesis and Code Generation for Signal Processing Systems". IEEE Transactions on Circuits & Systems, Vol. 47, No. 9, Set. 2000. pp 849-875.

Böke, C. (2000). "Combining Two Customization Approaches: Extending the Customization Tool TEReCS for Software Synthesis of Real-Time Execution Platforms". AES'2000 – Workshop on Architectures of Embedded Systems, Karlsruhe, Alemanha, Jan. 2000.

Brunel, J.-Y. et alii (2000). "COSY Communication IP's". DAC'2000 – Design Automation Conference, Los Angeles, EUA, Jun. 2000. Proceedings, ACM Press, 2000.

Burns, A. e Wellings, A. (1997). Real-Time Systems and Programming Languages. Addison-Wesley, 1997.

Cesario, W. et alii (2002). "Multiprocessor SoC Platforms: A Component-Based Design Approach". IEEE Design & Test of Computers. Vol. 19, No. 6., Nov/Dez. 2002. pp 44-51

Chen, R., Irwin, M.J. e Bajwa, R. (2001). "Architecture-Level Power Estimation and Design Experiments". ACM Transactions on Design Automation of Electronic Systems, Vol. 6, No. 1, Jan. 2001. pp 50-66.

Choi, K. e Chatterjee, A. (2001). "Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems". International Symposium on System Synthesis, Montreal, Canada, Out. 2001. Proceedings, ACM, 2001. pp 147-152.

Chou, P. et alii (1999). "IPChinook: An Integrated IP-Based Design Framework for Distributed Embedded Systems". DAC'99 – Design Automation Conference, New Orleans, EUA, Jun. 1999. Proceedings, ACM Press, 1999.

Dalal, V. e Ravikumar, C.P. (2001). "Software Power Optimizations in an Embedded System". VLSI Design Conference, Bangalore, India, Jan. 2001. Proceedings, IEEE Computer Science Press. 2001, pp 254-259

Dally, W.J. e Towles, B. (2001). "Route Packets, Not Wires: On-Chip Interconnection Networks", DAC'2001 – Design Automation Conference, New Orleans, EUA, Jun. 2001. Proceedings, ACM Press, 2001.

Dally, W. J. (1990). "Network and Processor Architecture for Message-Driven Computing". In: VLSI and Parallel Computation. Morgan Kaufmann, Los Altos, 1990.

Dalpasso, M., Bogliolo, A. e Benini. L. (1999). "Virtual Simulation of Distributed IP-Based Designs". DAC'99 – Design Automation Conference, New Orleans, EUA, Jun. 1999. Proceedings, ACM Press, 1999.

Davis II, J. et alii (2001). "Overview of the Ptolemy Project". Technical Memorandum UCB/ERL M01/11, University of California at Berkeley, Department of Electrical Engineering and Computer Science, Mar. 2001. Disponível em http://ptolemy.eecs.berkeley.edu

Design & Reuse (2003). http://www.us.design-reuse.com

Dhananjai, R., Chernyakhovsky, V. e Wilsey, P.A. (2000). "WESE: A Web-based Environment for Systems Engineering". WEBSIM'2000 - International Conference on Web-Based Modelling & Simulation. Jan. 2000. Proceedings, SCS, 2000

Demmeler, T. e Giusto, P. (2001) "A Universal Communication Model for an Automotive System Integration Platform". DATE'01 – Design, Automation and Test in Europe, Munich, Alemanha, Mar. 2001. Proceedings, IEEE Computer Society Press, 2001.

Duato, J. et alii. (1997). Interconnection Networks: an Engineering Approach. IEEE Computer Society Press, Los Alamitos, CA, 1997.

Dutta, S., Jensen, R. e Rieckmann, A. (2001). "Viper: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems". IEEE Design & Test of Computers, Vol. 18, No. 5, Set/Out. 2001. pp 21-31

Ebcioglu, K. e Altman, E. (1997). "DAISY: Dynamic Compilation for 100% Architectural Compatibility". 24th Annual Symposium on Computer Architecture, Denver, EUA, Jun. 1997. Proceedings, IEEE, 1997. pp 26-37.

Edwards, S., Lavagno, L., Lee, E.A. e Sangiovanni-Vincentelli, A. (1997). "Design of Embedded Systems: Formal Models, Validation, and Synthesis". Proceedings of the IEEE, Vol. 85, No. 3, Mar. 1997. pp 366-390

Fin, A. e Fummi, F. (2000). "A Web-CAD Methodology for IP-Core Analysis and Simulation". DAC'2000 – Design Automation Conference, Los Angeles, EUA, Jun. 2000. Proceedings, ACM Press, 2000.

Fowler, M.; Scott, K. (2000). "UML Distilled", Second Edition, Addison-Wesley, 2000.

Gauthier, L., Yoo, S. e Jerraya, A.A. (2001). "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software". DATE'01 – Design, Automation and Test in Europe, Munich, Alemanha, Mar. 2001. Proceedings, IEEE Computer Society Press, 2001.

Gervini, A.I., Corrêa, E.F., Carro, L. e Wagner, F.R. (2003). "Avaliação de Desempenho, Área e Potência de Mecanismos de Comunicação em Sistemas Embarcados". SEMISH'2003 – XXX Seminário Integrado de Software e Hardware. Campinas, Ago. 2003. Anais, SBC, 2003.

Givargis, T., Vahid, F. e Henkel, J. (2001). "System-Level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip". IEEE/ACM International Conference on Computer-Aided Design, San Jose, EUA, Nov. 2001. Proceedings, 2001.

Guerrier, P. e Greiner, A. (2000). "A Generic Architecture for On-Chip Packet-Switched Interconnections". DATE'00 – Design, Automation and Test in Europe, Paris, França, Mar. 2000. Proceedings, IEEE Computer Society Press, 2000.

Halambi, A. et alii (1999). "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability". DATE'99 – Design, Automation and Test in Europe, Munich, Alemanha, Mar. 1999. Proceedings, IEEE Computer Society Press, 1999.

Hartenstein, R. (2001). "A Decade of Reconfigurable Computing: a Visionary Retrospective". DATE'01 – Design, Automation and Test in Europe, Munich, Alemanha, Mar. 2001. Proceedings, IEEE Computer Society Press, 2001.

Haverinen, A. et alii (2002). "SystemC based SoC Communication Modeling for the OCP Protocol". White paper, Out. 2002. Disponível em http://www.synopsys.org

Hennessy, J.L., Patterson, D.A. e Goldberg, D. (1996). Computer Architecture: A Quantitative Approach. Second Edition. San Francisco, California, Morgan Kaufmann.

Herbert, O., Kraljic, I.C. e Savaria, Y. (2000). "A Method to Derive Application-Specific Embedded Processing Cores". CODES'2000 – 8th International Workshop on Hardware / Software Codesign, San Diego, EUA, Mai. 2000. Proceedings, ACM, 2000. pp 88-92.

Hessel, F. et alii (1999). "MCI: Multilanguage Distributed Cosimulation Tool". In: F.J.Rammig (ed.), Distributed and Parallel Embedded Systems. Kluwer Academic Publishers, 1999.

HLA (2000). "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - IEEE Standard No. 1516-2000". IEEE, 2000.

IBM (2003). "IBM CoreConnect Bus Architecture". Disponível em http://www3.ibm.com/chips/products/coreconnect/index.html

Inomata, K. (2001). "Present and Future of Magnetic RAM Technology". IEICE Transactions on Electronics. Vol. E84-C, No. 6 (2001). pp 740-746

Ishihara, T. e Yasuura, H. (2000). "A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors". DATE'00 – Design, Automation and Test in Europe, Paris, França, Mar. 2000. Proceedings, IEEE Computer Society Press, 2000.

Ito, S., Carro, L. e Jacobi, R. (2000). "System Design Based on Single Language and Single-Chip Java ASIP Microcontroller". DATE'00 – Design, Automation and Test in Europe, Paris, França, Mar. 2000. Proceedings, IEEE Computer Society Press, 2000.

Ito, S., Carro, L. e Jacobi, R. (2001). "Sashimi and FemtoJava: Making Java Work for Microcontroller Applications". IEEE Design & Test of Computers. Vol. 18, No. 5, Set-Out 2001. pp 100-110.

ITRON (2003). “ITRON”. Disponível em: http://www.itron.gr.jp

ITRS (2001). International Technology Roadmap for Semiconductors, versão 2001. Disponível em http://public.itrs.net/,

Karim, F., Nguyen, A. e Dey, S. (2002). "An Interconnect Architecture for Networking Systems on Chips". IEEE Micro, Vol. 22, No. 5, Set-Out. 2002. pp 36-45.

Keating, M. e Bricaud, P. (2002). Reuse Methodology Manual for System-on-a-Chip Designs. Kluwer Academic Publishers, 2002 (terceira edição).

Keutzer, K. et alii (2000). “System-Level Design: Orthogonalization of Concerns and Platform-Based Design”. IEEE Transactions on Computer-Aided Design of Integrated Circuits, Vol. 19, No. 12, Dez. 2000. pp 1523-1543.

Kin, J., Gupta, M. e Mangione-Smith, W. (2000). "Filtering Memory References to Increase Energy Efficiency". IEEE Transactions on Computers, Vol. 49, No. 1, Jan. 2000, pp 1-15.

Klaiber, A. (2000). "The Technology Behind Crusoe Processors". Technical Report Trasnmeta Corp., Santa Clara, CA.

Krapf, R., Mattos, J., Spellmeier, G. e Carro, L. (2002). "A Study on a Garbage Collector for Embedded Applications". SBCCI'02 – 15th Symposium on Integrated Circuits and System Design, Porto Alegre, Brasil, Set. 2002. Proceedings, IEEE Computer Society Press, 2002.

Krapf, R. e Carro, L. (2003). "Efficient Signal Processing in Embedded Java". ISCAS'2003 – IEEE International Symposium on Circuits & Systems, Bangkok, Tailândia, Mai. 2003. Proceedings, IEEE, 2003.

Landman, P. e Rabaey, J. (1996). “Activity-Sensistive Architectural Power Analysis”. IEEE Transactions on CAD of Integrated Circuits, Vol. 15, No. 6, Jun. 1996, pp 571-587

Lapsley, P., Bier, J., Shoham, A. e Lee, E. (1997). DSP Processor Fundamentals: Architectures and Features. IEEE Press, New York, 1997.

Lavagno, L., Martin, G. e Selic, B. (2001). UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Press, 2001.

Lekatsas, H. e Wolf, W. (1999). "SAMC: A Code Compression Algorithm for Embedded Processors". IEEE Transactions on CAD of Integrated Circuits, Vol. 18, No. 12, Dez. 1999. pp 1689-1701.

Li, Y., Potkonjak, M. e Wolf, W. (1997). “Real-time Operating Systems for Embedded Computing”. International Conference on Computer Design, Austin, EUA, Out. 1997. Proceedings, IEEE Computer Society Press, 1997.

Lindholm, T. e Yellin, F. (1997). The Java Virtual Machine Specification. The Java Series. Addison-Wesley, Reading, 1997.

Lyonnard, D. et alii (2001). “Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip”. DAC'2001 – Design Automation Conference. New Orleans, EUA, Jun. 2001. Proceedings, ACM Press, 2001.

Madisetti, V. K. (1995). VLSI Digital Signal Processors. IEEE Press, Butterworth-Heinemann, 1995.

Magarshack, P. (2002). "Improving SOC Design Quality through a Reproducible Design Flow". IEEE Design & Test of Computers, Vol. 19, No. 1, Jan-Fev. 2002, pp 76-83.

Mathworks (2003). Disponível em http://www.mathworks.com

McGhan, H. e O'Connor, M. (1998). "Picojava: A Direct Execution Engine For Java Bytecode". IEEE Computer, Vol. 31, No. 10, Out. 1998. pp 22–30.

Mello, B.A. e Wagner, F.R. (2002). “A Distributed Co-Simulation Backbone”. In: M.Robert et al. (eds), SOC Design Methodologies. Kluwer Academic Publishers, 2002.

Mentor (2003). “Seamless CVE”. Disponível em: http://www.mentor.com/seamless

Meyer, J. e Downing, T. (1997). Java Virtual Machine. O’Reilly & Associates, Inc., Sebastopol, EUA, 1997.

De Micheli, G. e Benini, L. (2002). "Networks-on-Chip: A New Paradigm for Systems-on-Chip Design". DATE’02 – Design, Automation and Test in Europe, Paris, França, Mar. 2002. Proceedings, IEEE Computer Society Press, 2002.

Microsoft (2003). “WindowsCE”. http://www.microsoft.com/windows/embedded

Mooney III, V. e Blough, D.M. (2002). "A Hardware-Software Real-Time Operating System Framework for SoCs". IEEE Design & Test of Computers. Vol. 19, No. 6., Nov/Dez. 2002. pp 44-51

Moore, G.E. (1965). "Cramming More Components Onto Integrated Circuits". Electronics Magazine, Vol. 38, Abr. 1965. pp 114-117.

Mrva, M., Buchenrieder, K. e Kress, R. (1998). "A Scalable Architecture for Multi-threaded JAVA Applications". DATE'98 – Design, Automation and Test in Europe, Paris, França, Mar. 1998. Proceedings, IEEE Computer Society Press, 1998.

Mulchandani, D. (1998). "Java for Embedded Systems". Internet Computing, Vol. 31, No. 10, Maio 1998. pp 30–39

OCP (2003). "Open Core Protocol". OCP-IP, 2003. Disponível em: http://www.ocpip.org

OSEK (2003). "OSEK". Disponível em: http://www.osek-vdx.org/osekvdx_OS.html

Oyamada, M. e Wagner, F.R. (2000). "Co-simulation of Embedded Electronic Systems". 12nd European Simulation Symposium. Hamburgo, Alemanha, Out. 2000. Proceedings, SCS, 2000.

Paulin, P. et alii (1997). "Embedded Software in Real-time Signal Processing Systems: Application and Architecture Trends". Proceedings of the IEEE, Vol. 85, No. 3, Mar. 1997. pp 419-435.

Passerone, R., Rowson, J.A. e Sangiovanni-Vincentelli, A. (1998). "Automatic Synthesis of Interfaces between Incompatible Protocols". DAC'98 – Design Automation Conference, San Francisco, EUA, Jun. 1998. Proceedings, ACM Press, 1998.

Red Hat (2003). "eCos". Disponível em: http://sources.redhat.com/ecos/

RTLinux (2003). "RTLinux". Disponível em: http://fsmlabs.com/community

Sangiovanni-Vincentelli, A. e Martin, G. (2001). "Platform-Based Design and Software Design Methodology for Embedded Systems". IEEE Design & Test of Computers, Vol. 18, No. 6, Nov/Dez. 2001. pp 23-33.

Shandle, J. e Martin, G. (2002). "Making Embedded Software Reusable for SoCs". EEDesign, Mar. 1, 2002.

Shiue, W.-T. e Chakrabarti, C. (1999). "Memory Exploration for Low Power, Embedded Systems". ISCAS'1999 – IEEE International Symposium on Circuits & Systems. Proceedings, Vol. 1, pp 250-253

Simunié, T., Benini, L. e De Micheli, G. (1999)."Cycle-Accurate Simulation of Energy Comsumption in Embedded Systems". DAC'99 – Design Automation Conference, New Orleans, EUA, Jun. 1999. Proceedings, ACM Press, 1999.

Smith, J. e De Micheli, G. (1998). "Automated Composition of Hardware Components". DAC'98 – Design Automation Conference, San Francisco, EUA, Jun. 1998. Proceedings, ACM Press, 1998.

Sonics (2003). "Sonics SiliconBackplane MicroNetwork". Sonics, 2003. Disponível em: www.sonicsinc.com

Stepner, D., Rajan, N. e Hui, D. (1999). "Embedded Application Design Using a Real-time OS". DAC'99 – Design Automation Conference, New Orleans, EUA, Jun. 1999. Proceedings, ACM Press, 1999.

Synopsys (2003). "CoCentric System Studio". Disponível em: http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html

SystemC (2003). "SystemC". Disponível em: http://www.systemc.org

Tomiyama, H., Ishihara, T., Inoue, A. e Yasuura, H. (1998). "Instruction Scheduling for Power Reduction in Processor-Based System Design". DATE'98 – Design, Automation and Test in Europe, Paris, França, Mar. 1998. Proceedings, IEEE Computer Society Press, 1998.

Texas Instruments Inc. (2002). "TMS320C6000 CPU and Instruction Set Reference Guide". Texas Instruments, 2000. Disponível em: http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf

Tiwari, V., Malik, S. e Wolfe, A. (1994). "Power Analysis of Embedded Software: a First Step Towards Software Power Minimization". IEEE Transactions on Very Large Scale Integration (VLSI) Systems. Vol. 2, No. 4, Abr. 1994, pp 437-445.

Transtech (2003). "Virtuoso 4.1 Real Time Operating System". Transtech DSP, 2003. Disponível em: http://www.transtech-dsp.com/software/virtuoso.htm

Valderrama, C. et alii (1998). "Automatic VHDL-C Interface Generation for Distributed Cosimulation: Application to Large Design Examples". Journal on Design Automation for Embedded Systems, Vol. 3, No. 2/3, Mar. 1998.

Verilog (2003). "Verilog". Disponível em: http://www.accellera.org

Venners, B. (1998). Inside the Java Virtual Machine. McGraw-Hill, New York, USA, 1998.

VHDL (2002). "IEEE Standard VHDL Language Reference Manual. IEEE Standard No. 1076-2002". IEEE, 2002.

VSIA (2003). "Virtual Socket Interface Alliance". Disponível em: http://www.vsi.org

WindRiver (2003). "VxWorks". Disponível em http://www.windriver.com/products/vxworks5.

Wolf, W. (2001). Computers as Componentes. McGraw-Hill, 2001.

Yoo, S. e Jerraya, A.A. (2003). "Introduction to Hardware Abstraction Layers for SoC". DATE'2003 – Design, Automation and Test in Europe. Munich, Alemanha, Mar. 2003. Proceedings, IEEE Computer Society Press, 2003.

Zeferino, C., Kreutz, M., Carro, L. e Suzim, A.A. (2002a). "A Study on Communication Issues for Systems-on-Chip". SBCCI'02 – 15th Symposium on Integrated Circuits and System Design, Porto Alegre, Brasil, Set. 2002. Proceedings, IEEE Computer Society Press, 2002.

Zeferino, C., Kreutz, M., Carro, L. e Suzim, A.A. (2002b). "Models for Comunication Tradeoffs on Systems-on-Chip". IFIP WG 10.5 International Workshop on IP-Based SOC Design, Grenoble, França, Out. 2002. Proceedings, 2002. pp 395-400.

Zhang, Y., Hu, X. e Chen, D. (2002). "Task Scheduling and Voltage Selection for Energy Minimization". DAC'2002 – Design Automation Conference, New Orleans, EUA, Jun. 2002. Proceedings, ACM Press, 2002.

Zorian, Y. e Marinissen, E. (2000). "System Chip Test - How will it Impact your Design". DAC'2000 – Design Automation Conference, Las Vegas, EUA, Jun. 2000. Proceedings, ACM Press, 2000.

Biografias

Luigi Carro é professor adjunto do Deptartamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul (UFRGS) e orientador dos Programas de Pós-graduação em Computação (PGCC) e Pós-graduação em Engenharia Elétrica (PGEE) da mesma universidade Em 2001 realizou seu pós-doutorado junto à University of California, San Diego, Computer Science Dept.. Participou de diversos projetos de pesquisa, é bolsista CNPq II-B e já publicou mais de 50 trabalhos em conferências internacionais, tendo orientado 9 dissertações de mestrado. Atualmente orienta 4 alunos de mestrado e 6 de doutorado (3 em co-orientação).

Flávio Rech Wagner é professor titular do Instituto de Informática da UFRGS e orientador do Programa de Pós-graduação em Computação (PGCC) da mesma universidade. Tem doutorado em Informática pela Universidade de Kaiserslautern, Alemanha (1980-1983) e pós-doutorados junto ao Laboratório TIMA de Grenoble, França (1992 e 2002). É bolsista pesquisador I-C do CNPq. Já publicou um total de 41 trabalhos internacionais e 45 trabalhos nacionais. Publicou ainda 2 livros-texto na Escola de Computação. Já orientou ou co-orientou 4 teses de doutorado e 21 dissertações de mestrado, e atualmente orienta outras 4 teses de doutorado (uma em co-orientação) e 5 dissertações de mestrado. Já participou de comitês de programa de 18 conferências internacionais, sendo duas vezes na qualidade de coordenador. É o atual coordenador do Grupo de Trabalho 10.5 da IFIP, que reúne especialistas internacionais na área de "Design and Engineering of Electronic Systems". Já foi membro do Comitê Assessor de Ciência da Computação do CNPq e ocupou diversas funções na diretoria da Sociedade Brasileira de Computação, da qual é o atual presidente, em segundo mandato.