AUGUSTO KEN MORITA

Projeto e desenvolvimento de uma arquitetura de baixo consumo de potência para microprocessadores

São Paulo
2015

AUGUSTO KEN MORITA

Projeto e desenvolvimento de uma arquitetura de baixo consumo de potência para microprocessadores

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Ciências

São Paulo
2015

AUGUSTO KEN MORITA

Projeto e desenvolvimento de uma arquitetura de baixo consumo de potência para microprocessadores

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Ciências

Área de Concentração: Microeletrônica

Orientador: Prof. Livre-Docente Wilhelmus A. M. Van Noije

São Paulo
2015

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Morita, Augusto Ken

Projeto e desenvolvimento de uma arquitetura de baixo consumo de potência para microprocessadores / A. K. Morita -- versão corr. -- São Paulo, 2015.

173 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1.Engenharia elétrica 2.Engenharia eletrônica 3.Microprocessadores 4.Consumo de energia elétrica I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II.t.

AGRADECIMENTOS

Ao professor Wilhelmus A. M. Van Noije, pela orientação e, em especial, pela liberdade que proporcionou nesta pesquisa.

À Freescale Semicondutores, por possibilitar o uso de dados obtidos durante uma investigação de processadores com menor área e consumo de potência.

A todos que, de alguma forma, colaboraram com este trabalho.

À minha família, pela paciência e pelo incentivo.

RESUMO

O trabalho trata do projeto e do desenvolvimento de um processador de baixo consumo de potência, de forma simplificada, explorando técnicas de microarquitetura, para atingir menor consumo de potência. É apresentada uma sequência lógica de desenvolvimento, a partir de conceitos e estruturas básicas, até chegar a estruturas mais complexas e, por fim, mostrar a microarquitetura completa do processador. Esse novo modelo de processador é comparado com estudos prévios de três processadores, sendo o primeiro modelo síncrono, o segundo assíncrono e o terceiro uma versão melhorada do primeiro modelo, que inclui minimizações de registradores e circuitos. Uma nova metodologia de criação de *padring* de microcontroladores, baseada em reuso de informações de projetos anteriores, é apresentada. Essa nova metodologia foi criada para a rápida prototipagem e para diminuir possíveis erros na geração do código do *padring*. Comparações de resultados de consumo de potência e área são apresentadas para o processador desenvolvido e resultados obtidos com a nova metodologia de geração de padring também são apresentados. Para o processador, um modelo, no qual se utilizam múltiplos barramentos para minimizar o número de ciclos de máquina por instrução, é apresentado. Também foram ressaltadas estruturas que podem ser otimizadas e circuitos que podem ser reaproveitados para diminuir a quantidade de circuito necessário na implementação. Por fim, a nova implementação é comparada com os três modelos anteriores; os ganhos obtidos de desempenho com a implementação dessas estruturas foram de 18% – que, convertidos em consumo de potência, representam economia de 13% em relação ao melhor caso dos processadores comparados. A tecnologia utilizada no desenvolvimento dos processadores foi CMOS 250nm da TSMC.

Palavras-chave: processador, microprocessador, potência, baixo consumo, arquitetura

ABSTRACT

This work is a development and implementation of a low power processor in a simplified way, exploring microarchitecture techniques to achieve low power consumption. A logic sequence of design flow is presented, starting from basic concepts and circuit structures incrementing these concepts and structures to achieve a complex microarchitecture of a processor. A new methodology for microcontroller padring creations based in reuse of previous project information is presented. This new methodology was developed for fast prototyping and decreases the possible error in generation of microcontroler padring code creation. This new microarchitecture is compared with three previous processors, one is an original synchronous version, the second is an asynchronous version, and the third is based on the first model with register and circuit minimizations. Results of area and power consumption are compared with this new proposed architecture. The new model uses multiple buses with access timing tuned for different internal blocks. This timing tuning decrease the number of machine cycle necessary per instruction. In addition, it presents some macro block circuit partition and circuit reuse to minimize the circuit necessary for implementation. The gain obtained in performance with these new structures was 18%, converting to power consumption, it represent a decrease in 13% in relation with the best of three processor compared. The technology used in the development of these processors was CMOS 250nm from TSMC.

Keywords: processor; microcontroller, power, low power; architecture

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

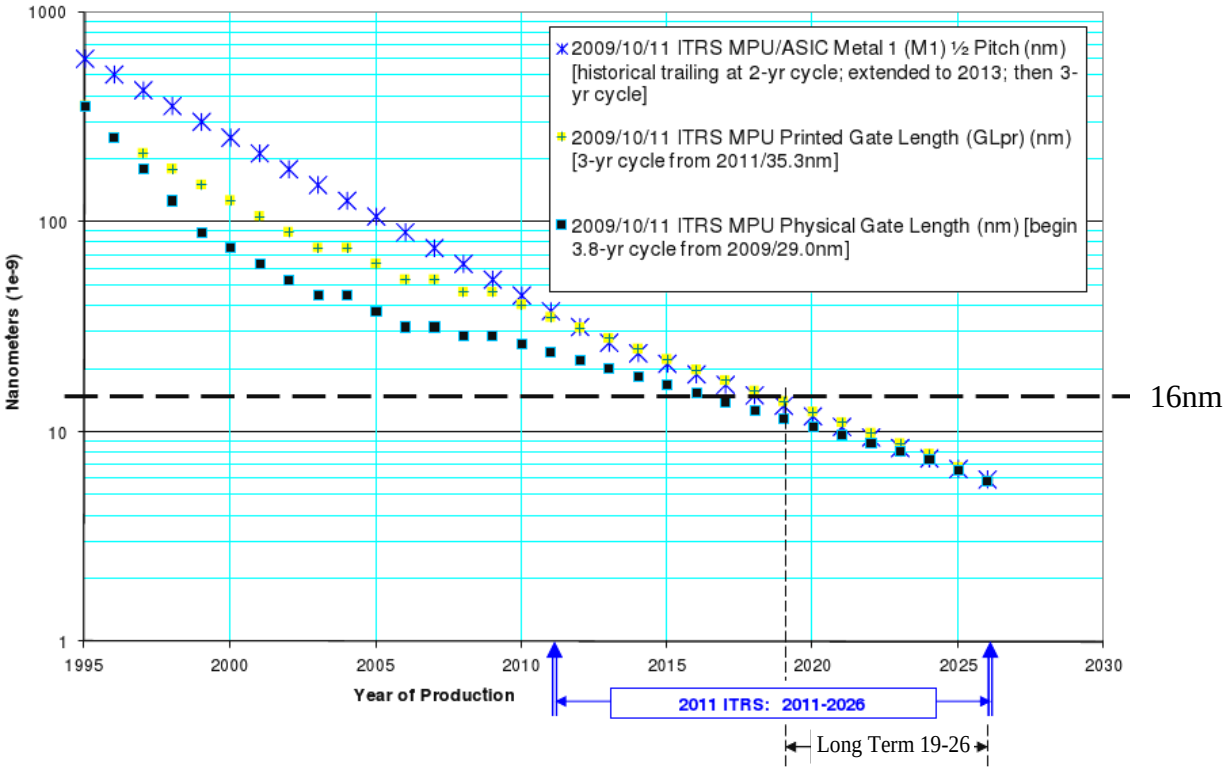
Capítulo - 1. Introdução

1.1 Motivação do Trabalho

Nos dias atuais, o baixo consumo de potência é um dos pontos mais importantes no projeto de SOC (*System On a Chip*). Os sistemas estão mais complexos, demandando cada vez mais transistores integrados para a realização de suas tarefas, fazendo-se necessário o aumento da energia para alimentar esses circuitos [1]. Apesar de a tecnologia de fabricação ter evoluído bastante com a redução do comprimento de canal dos transistores (Figura 1), reduzindo o consumo desses transistores, ainda existe grande necessidade de tornar os dispositivos mais eficientes, no que diz respeito ao consumo de potência [2][3].

Em 2015, a tecnologia com menor comprimento de canal utilizada em microcontroladores está entre 40nm e 28nm. Para processadores, o comprimento está na ordem de 14nm. Muito próximo da projeção apresentada na tendência do comprimento de canal ilustrada na Figura 1.

Figura 1 - Tendência do comprimento do canal e espessura do metal M1

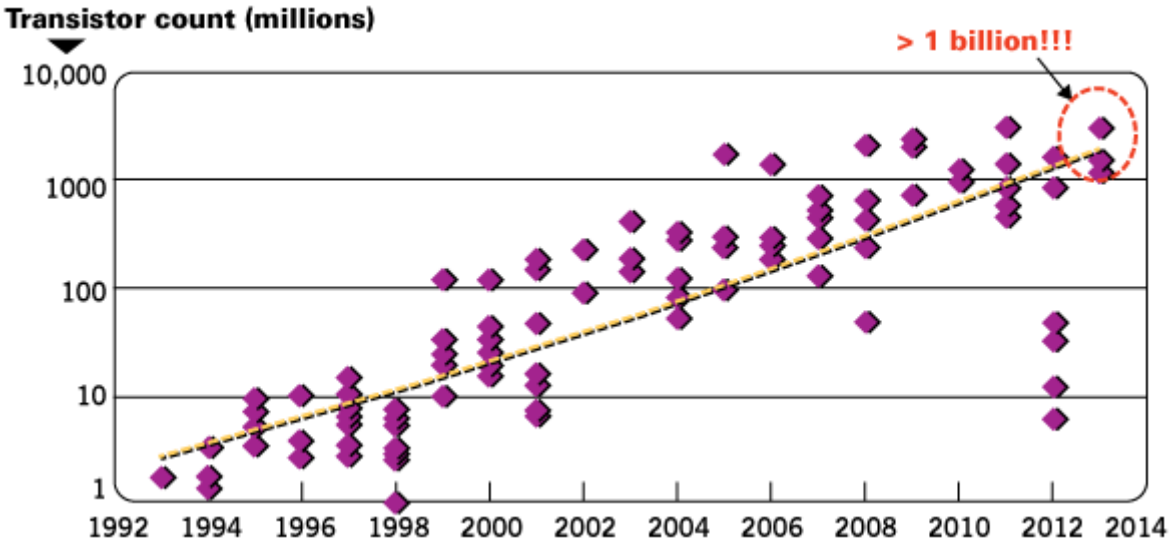


A Figura 2 mostra o crescimento da complexidade do circuito na integração dos processadores durante os últimos anos [3]. Na proporção que os integrados estão evoluindo,

em breve, será inviável a confecção de SOC mais complexos, devido a problemas térmicos causados pelo consumo de potência.

Hoje, podemos notar forte tendência de uso de processadores de vários núcleos, devido à falta de tecnologia adequada para dissipar o calor gerado pelo consumo de potência dos circuitos – no caso de um núcleo de processamento com frequência maior. Com múltiplos núcleos, podemos distribuir o processamento para diversas regiões do *chip*, aproveitando o paralelismo dos programas para diminuir a frequência de operação; assim, sem necessidade de concentrar a dissipação de potência em uma determinada região e podendo trabalhar em frequências menores devido ao processamento em paralelo.

Figura 2 - Evolução da integração de transistores por *die*

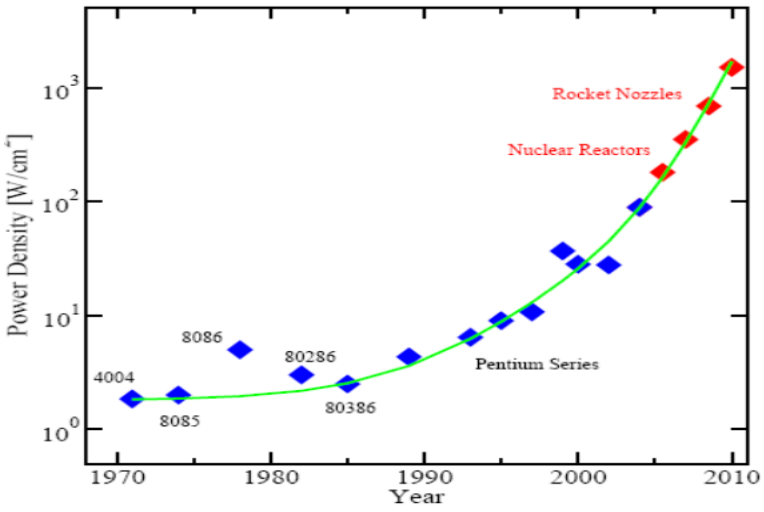


Fonte: ISSCC 2013: High-performance digital trends

Com a tendência de termos, cada vez mais, circuitos complexos para aumentar o desempenho e o processamento dos processadores para várias aplicações. A solução de múltiplos núcleos não será suficiente para processamento de aplicações que não tenham paralelismo inerente da aplicação, assim precisamos diminuir o consumo de potência desses circuitos de uma outra forma. Para aplicações como de imagem e gráficos onde existe um forte paralelismo intrínseco da aplicação, existem atualmente soluções com centenas de núcleos de processamento.

Interessante notarmos que, em 2004, tínhamos uma previsão muito ruim a respeito do consumo de potência – que pode ser observada na Figura 3 [4]. Em 2009, esta previsão foi revisada, devido à introdução dos processadores de múltiplos núcleos. Na Figura 4, podemos observar expressiva mudança na evolução no consumo de potência.

Figura 3 - Tendência da evolução do consumo de potência dos processadores em 2004 [4]



Mesmo em sistemas de alto desempenho, que não tenham restrições com o fornecimento de energia, o escalonamento de frequências e o desempenho dos processadores têm previsão de evolução em ritmo menor, devido à limitação da tecnologia de dissipação de calor – como pode ser observado na Figura 4 [5].

Novamente, em 2011, foi revisada para baixo a previsão de evolução do consumo de potência, devido às limitações de dissipação térmica que não apresentaram melhora.

A Figura 4 mostra a tendência do consumo de potência para os processadores de alto desempenho nos próximos anos, considerando quatro possíveis cenários:

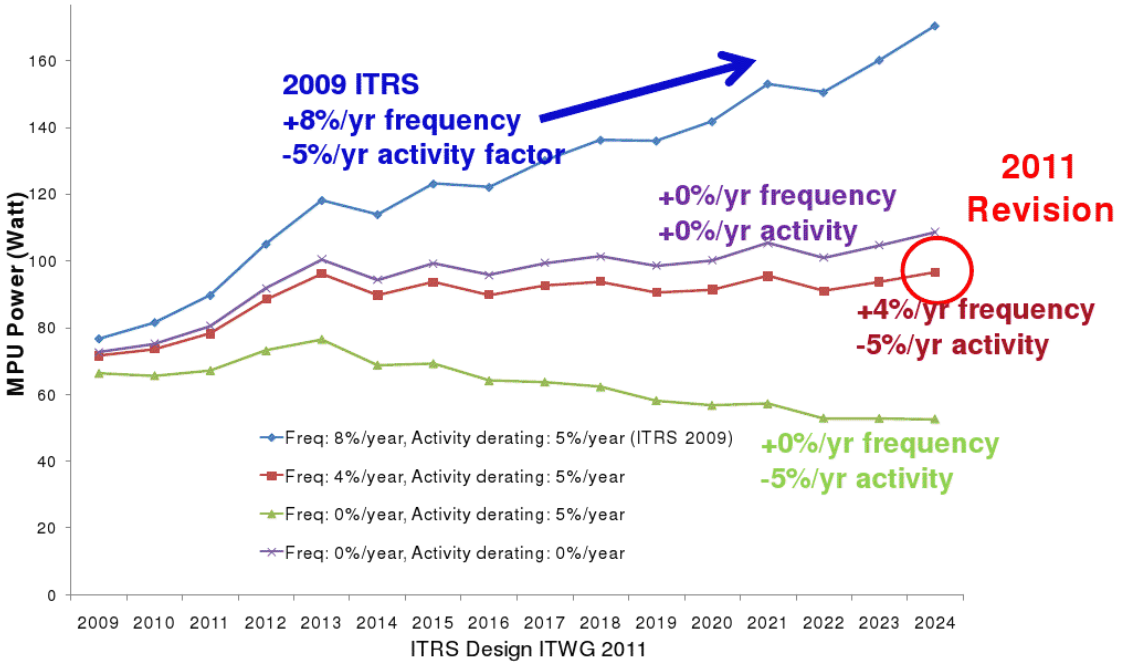
- Frequência de operação crescendo na proporção de 8% ao ano e atividade do circuito diminuindo à taxa de 5% ao ano.
- Frequência de operação fixa, isto é, 0% de crescimento ano e atividade do circuito fixa.
- Frequência de operação crescendo na proporção de 4% ao ano e atividade do circuito diminuindo à taxa de 5% ao ano.
- Frequência de operação fixa e atividade do circuito diminuindo à taxa de 5% ao ano.

Além da limitação da frequência devido à tecnologia, outro fator importante para serem desenvolvidos dispositivos de menor consumo de potência é o elevado custo dos encapsulamentos para dissipação de calor. A necessidade de maior desempenho e a limitação devido a esses dois fatores têm levado à crescente demanda por utilizar técnicas de baixo consumo de potência e de distribuição térmica. Nesse sentido, podemos dizer que a dissipação

térmica está relacionada ao consumo de potência e ao desempenho dos dispositivos eletrônicos.

Outro motivo dessa demanda de componentes de baixo consumo de potência é a explosão de dispositivos eletrônicos portáteis, que utilizam baterias para seu funcionamento – tocadores de MP3, tabuletas eletrônicas (tablet), microcomputadores portáteis (notebook), telefones celulares, entre outros.

Figura 4 - Tendência da evolução do consumo de potência dos processadores

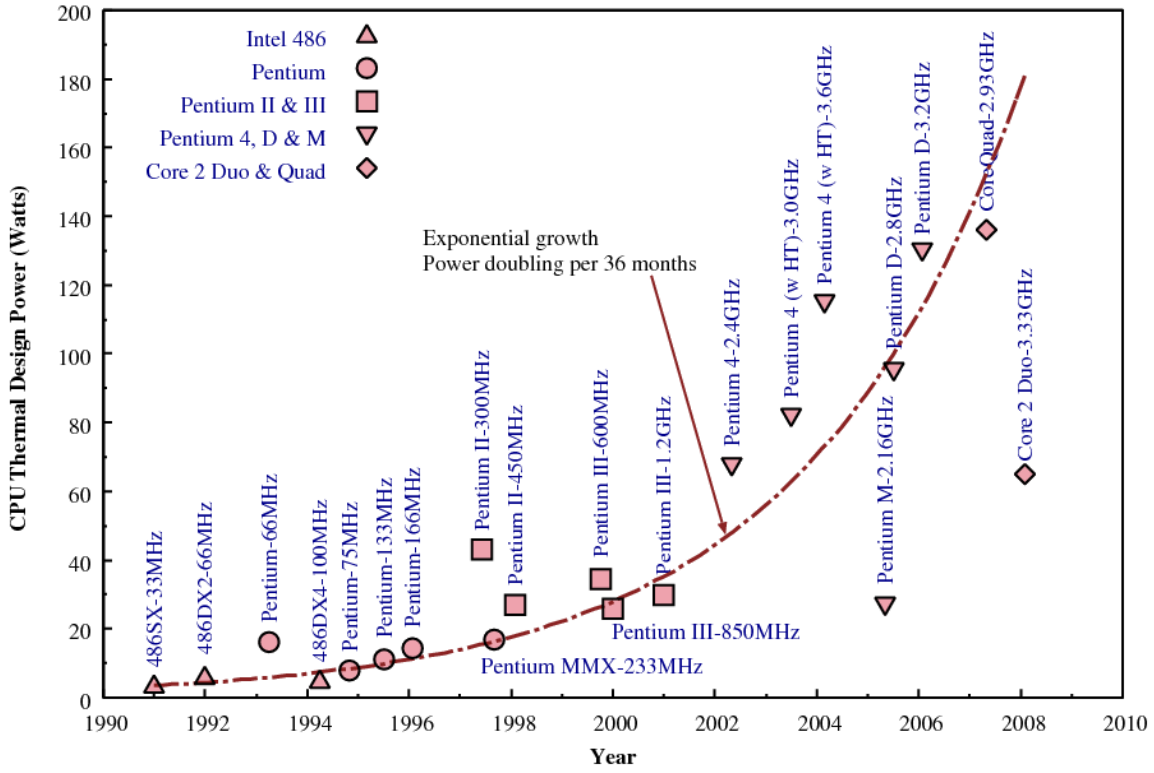


Para dispositivos portáteis, um dos itens mais importantes é o tempo que ele pode ficar em operação sem a necessidade de uma fonte de energia para carregar suas baterias. A evolução da tecnologia de baterias não acompanha, com a mesma velocidade, o aumento da complexidade dos circuitos; assim, faz-se necessário produzir circuitos que tenham menor demanda da quantidade de energia para o funcionamento.

Apesar de a tecnologia de fabricação ter evoluído e de ter havido diminuição exponencial no consumo de potência de cada transistor, o consumo médio de potência dos processadores tem dobrado a cada 3 anos (Figura 5) [4]. Após 2009, devido à introdução de processadores com vários núcleos e técnicas de controle de frequências de operação, o consumo de potência não tem subido de forma exponencial.

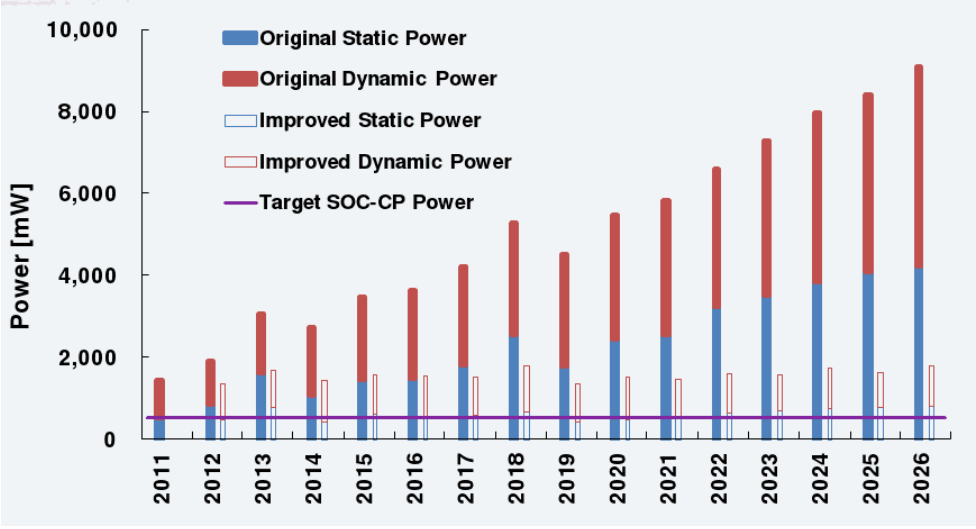
Dispositivos de baixo consumo de potência são um fator importante para desenvolvimento de produtos de menor custo, tamanhos menores dos equipamentos, baterias menores e redução nos meios de resfriamento, os quais resultam em produtos mais baratos.

Figura 5 - Evolução do consumo de potência dos processadores da Intel [4]



A Figura 6 mostra a tendência do consumo de potência dos dispositivos portáteis para os próximos anos [5]. A linha horizontal roxa representa o valor médio ideal do consumo de potência para os dispositivos portáteis. Podemos observar que aplicar melhorias tecnológicas existentes e descobrir novas melhorias no consumo de potência terão fator significativo no desenvolvimento de novos dispositivos, motivados pela limitação da utilização de baterias e tecnologia de dissipação de calor.

Figura 6 - Tendência do consumo de potência dos dispositivos portáteis e melhorias

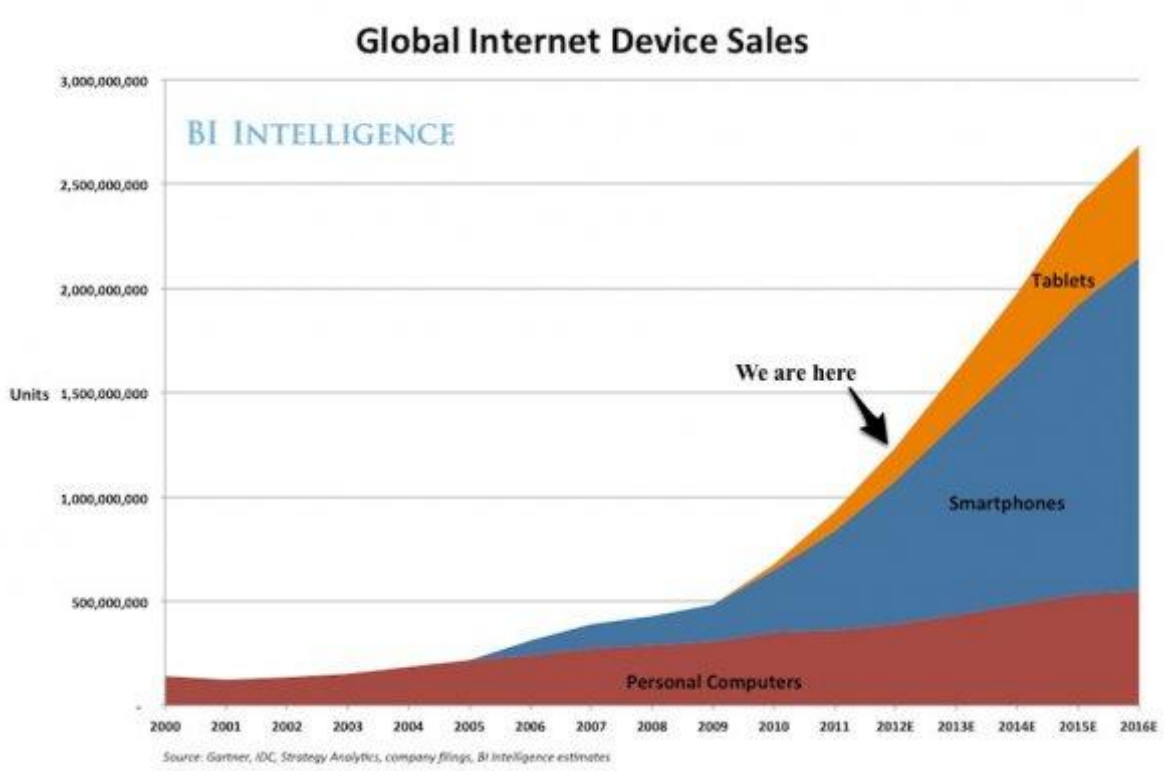


A explosão no uso da internet e de serviços de telecomunicação tem aumentado o uso de dispositivos eletrônicos no cotidiano. A internet possibilitou às pessoas estarem sempre conectadas na rede, em muitos casos, simplificando as tarefas mais simples do dia a dia. Essa demanda de conexão tem trazido o aumento exponencial de acesso de alta velocidade e, consequentemente, o aumento de consumo de potência dos dispositivos em relação aos sistemas anteriores.

A quantidade de computadores e dispositivos de rede tem aumentado em proporções geométricas, como pode ser observado na Figura 7. Consequentemente, o consumo de energia conjunto desses dispositivos tem crescido no decorrer do tempo e possivelmente continuará crescendo.

De acordo com o relatório "Gartner Dataquest's Statistics", em abril de 2012, 1 bilhão de computadores pessoais tinha sido confeccionado. Em 2007, atingiu-se a marca de 2 bilhões de computadores produzidos. Pelos dados da "Forrester Research", havia, em 2008, cerca de 1 bilhão de PC em uso no mundo, possivelmente atingindo 2 bilhões em 2015. Importante notar que foram 27 anos para se atingir a marca de 1 bilhão de computadores pessoais (PC) em uso e se previam somente 7 anos para o segundo bilhão. Esses dados indicam que número de dispositivos eletrônicos está crescendo de forma geométrica.

Figura 7 - Vendas de dispositivos com acesso à internet



Para termos dimensão do que representa o consumo de potência desses dispositivos, no nível global, podemos fazer uma conta simples, na qual consideraremos 400 Wh por cada PC ligado, incluindo-se o monitor.

Conforme dados da “Forrester Research”, temos mais que 1 bilhão de PC em funcionamento, neste momento; isto significa, se todos estiverem ligados ao mesmo tempo, o consumo de potência de:

$$1.000.000.000 * 400 = 400\ GWh$$

Esse valor representa 0,2% de toda energia gerada no mundo ou quase 1% de toda energia gerada no Brasil. Lembrando que, além dos PC, existe uma gama completa de dispositivos eletrônicos que suportam o estilo de vida atual, como, por exemplo, telefones celulares, tablets, roteadores, telefones comuns, impressoras, televisores, etc. O consumo de energia desses dispositivos semicondutores tem impacto direto no consumo total da energia elétrica de uma residência, representando, atualmente, cerca de 10% da energia doméstica consumida [8]. Como a tecnologia de geração de energia também não tem evoluído no mesmo ritmo do aumento dos circuitos eletrônicos, resolver a questão de dispositivos de baixo consumo de potência tem se mostrado cada vez mais necessário.

Nesse sentido, a oportunidade de melhoria de projeto, enfatizando baixo consumo de potência, está presente em todos os níveis do desenvolvimento de um produto, desde algoritmos e arquitetura, até a tecnologia utilizada.

1.2 Objetivos

Esse trabalho teve por objetivos desenvolver e implementar um processador de baixo consumo de potência. Também pretendeu-se fazer comparação de 3 diferentes implementações do mesmo processador para a validação do método utilizado nesse trabalho.

Foi desenvolvido uma metodologia de criação de padring para microcontroladores para automatizar e facilitar a criação de prototipos e diminuir possível erros de geração de código que levem a um acrescimo de consumo de potência e possiveis erros de implementação dos microcontroladores.

A avaliação do consumo de potência e da área se limitaram ao processador; um conjunto básico de periféricos foi introduzido, compondo um microntrolador completo para avaliação.

Um programa de uma aplicação, fornecido por um cliente, foi utilizado na avaliação do consumo de potência; essa metodologia diminui as chances de escolhermos uma aplicação/programa que esteja privilegiando uma determinada característica de um dos processadores.

A tecnologia utilizada no desenvolvimento dos processadores para comparação e do processador com arquitetura melhorada foi CMOS 250nm da TSMC.

O programa de controle rodou em simulação, gerando um arquivo de transição de estados dos transistores; as informações deste arquivo foram processadas pelo programa de cálculo de potência consumida e traduzidas em consumo de potência.

A área foi avaliada com os resultados fornecidos nos relatórios do programa de *backend* Encounter da Cadence. Para facilitar a integração dos processadores e dos periféricos, uma metodologia de geração de partes do código RTL de integração do microcontrolador foi desenvolvida durante o trabalho.

1.3 Descrição sucinta dos capítulos da tese

A seguir, apresentamos descrição sucinta de cada capítulo desta tese.

No capítulo 2, analisaremos o consumo de potência de circuitos, comentaremos as várias dimensões que estão envolvidas quando consideramos esses consumos de potências. Apresentaremos as principais técnicas existentes para diminuição de consumo de potência de circuitos, como diminuição do número de chaveamentos dos transistores, planejamento físico e tecnológico, arquitetura do sistema e domínio de alimentações. Será feita uma breve

introdução do direcionamento desse trabalho, definindo as diretrizes básicas de desenvolvimento e implementação. Também será apresentado o processador que iremos utilizar nesse trabalho.

No capítulo 3, discutiremos o desempenho de processadores, apresentando técnicas de otimização de desempenho. Será feita a abordagem do significado de desempenho, olhando-se o lado do consumo de potência do circuito. Uma abordagem de conceitos básicos de microcontroladores será apresentada, seguindo para estruturas básicas que compõe o processador. Continuaremos com estruturas mais complexas, até compor todas as estruturas de um processador. Detalhamento de algumas estruturas específicas e particularidades do microcontrolador serão apresentados.

O capítulo 4 apresentará os resultados obtidos nesse trabalho e faremos a comparação com trabalhos anteriores, tanto de potência como de área. Explicaremos os parâmetros utilizados nas medidas de potência e área. Apresentaremos e detalharemos uma metodologia de geração automática de código de interligação e multiplexação de *pads*, desenvolvida e utilizada na implementação dos microcontroladores para comparação.

Por fim, no capítulo 5, apresentaremos as conclusões desse trabalho. Abordaremos o motivo dos valores obtidos e explicaremos onde houve ganhos e perdas devido à abordagem utilizada nesse trabalho. Faremos comentários sobre as dificuldades encontradas e possíveis melhorias que podem ser implementadas no futuro.

Capítulo - 2. Conceitos de consumo de potência e direcionamento do trabalho

2.1 Consumo de potência de circuitos digitais

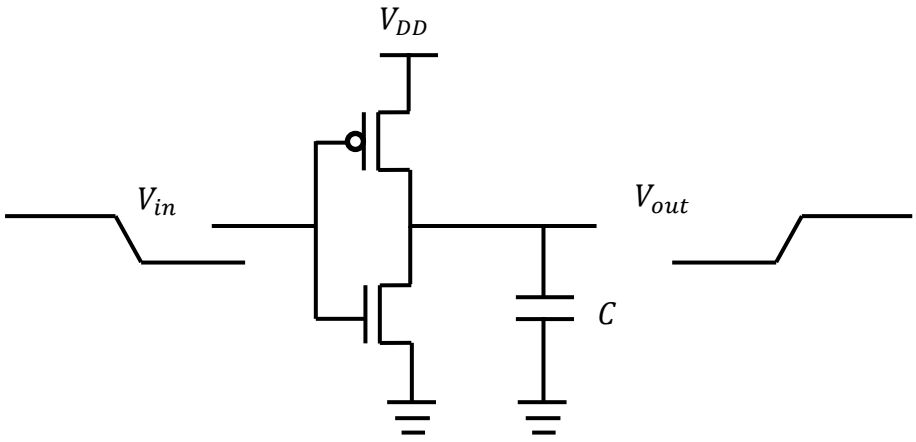
Para dissertamos sobre baixo consumo de potência, precisamos definir o que é consumo de potência nos CI (Circuitos Integrados). Basicamente, podemos dizer que existem, no CI, dois tipos de consumos de potência: i) a dissipação dinâmica devido à operação do circuito e ii) o consumo de potência estático, que é o consumo de potência do circuito mesmo que não esteja em operação.

Nos dias atuais, a tecnologia predominante é o CMOS (*Complementary metal-oxide semiconductor*). O consumo de potência para esses dispositivos é composto de três componentes: i) potência dinâmica necessária para o chaveamento das capacitâncias P_{dyn}; ii) potência dinâmica dissipada em curto circuito (*short circuit*) P_{sc}; iii) e a potência estática dissipada por fuga (*leakage)* P_{leak}.

$$P = P_{dyn-c} + P_{sc} + P_{leak} \quad (1)$$

Dentro desses componentes, a potência dinâmica de chaveamento P_{dyn} é a de maior grandeza. Este consumo de potência é associado ao chaveamento dos valores lógicos do circuito e está ilustrado na Figura 8.

Figura 8 - Porta inversora CMOS



$$Energia/Transição = CV_{DD} \quad (2)$$

$$P_{dyn} = n\frac{Energia}{Transição}f = nCV_{DD}{}^2 f \quad (3)$$

Onde:

C é a capacitância total de chaveamento, V_{DD} é a tensão de alimentação, f a frequência de operação do circuito e n representa a probabilidade de transição esperada do circuito [9].

Esse consumo de potência está diretamente ligado à complexidade do circuito desenvolvido. O valor do consumo de potência pode diminuir, caso o circuito desenvolvido seja simplificado ou otimizado de alguma forma.

O consumo de potência por curto circuito P_{sc} ocorre quando há superposição da condução dos transistores PMOS e NMOS que formam a porta lógica CMOS, no momento da mudança das entradas da porta lógica – por exemplo: durante a transição das tensões de saída dos inversores CMOS, quando a corrente flui diretamente da fonte para o terra.

O consumo de potência por curto circuito tem dedução complexa, mas, em sua forma simplificada, pode ser escrito como [10]:

$$P_{sc} = \left(nV_{DD}\frac{I_{sat}t_r}{2} + nV_{DD}\frac{I_{sat}t_f}{2}\right)f = nV_{DD}fI_{sat}\frac{t_r + t_f}{2} \quad (4)$$

Onde:

I_{sat} é a corrente de saturação e está ligado à largura dos transistores e os tempos t_r e t_f referem-se ao tempo de subida e descida do sinal de entrada do circuito, respectivamente.

Esse consumo de potência pode ser representativo se as entradas do circuito tiverem tempos de subida e descida longos. Podemos concluir que há relação direta com o tamanho dos transistores, devido ao componente I_{sat}. Para transistores largos, como *buffers* ou *drivers* de relógio, é necessário tomar precauções para diminuir a corrente de curto circuito.

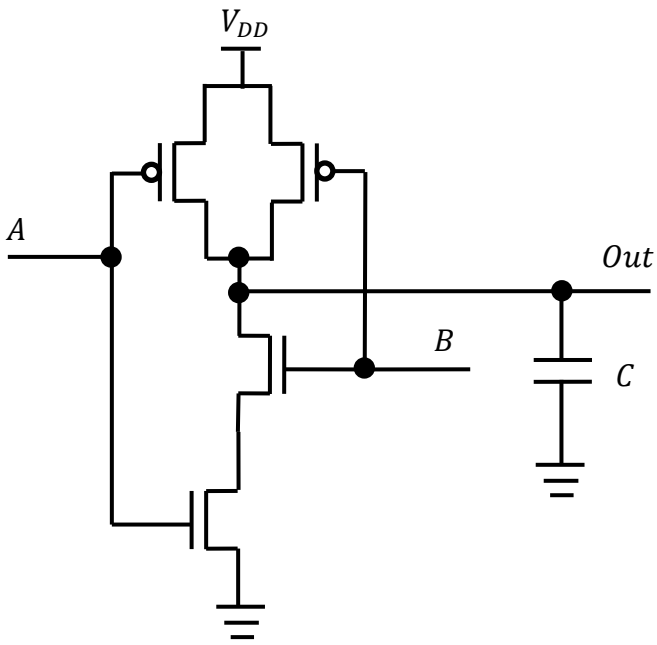
Com certos cuidados, a potência de curto circuito pode ser menor do que 15% da potência de chaveamento [11].

Um ponto importante a considerar, quando medirmos o consumo de potência dinâmica, é a necessidade de definir um vetor de testes ou um algoritmo de programa comum, para se fazer a medida, pois a atividade de chaveamento é dependente dos dados de entrada. Um exemplo é o cálculo de probabilidade de transição para Porta NAND duas entradas, ilustrado na Figura 9. Assumindo a probabilidade das entradas assumirem valores 0 ou 1 iguais a ½, podemos chegar ao valor de probabiliade de ¾ para a saída assumir o valor 1 e ¼ a probabiliade de a saída assumir 0. A energia envolvida na transição pode ser calculada quando se faz a análise da probabiliade de a saída transicionar de um valor a outro.

Não adianta fazer medida de consumo de potência para dois processadores rodando programas ou algoritmos distintos; isto porque a quantidade de mudanças de estado dos transistores será diferente para cada algoritmo. A análise simplificada de consumo de potência é feita através da taxa de atividade de chaveamento do circuito, sem considerar algoritmo ou aplicação específico; nesse caso, os valores obtidos serão somente uma aproximação da ordem de grandeza do valor real a ser obtido.

Nas tecnologias acima de 0,5µm, o consumo de potência predominante é o consumo de potência dinâmico, enquanto o consumo de potência estático P_{Leak} pode ser ignorado. Idealmente falando, o consumo de potência estático deveria ser zero; isto é, não existe corrente entre fonte e dreno ou junção para substrato quando o transistor CMOS está alimentado.

Figura 9 - Porta NAND 2 CMOS



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

$$Assumindo$$

$$P(A=1)=\frac{1}{2} \qquad P(B=1)=\frac{1}{2}$$

$$P(Out=1)=\frac{3}{4}$$

$$P(0\rightarrow 1)=P(1\rightarrow 0)$$

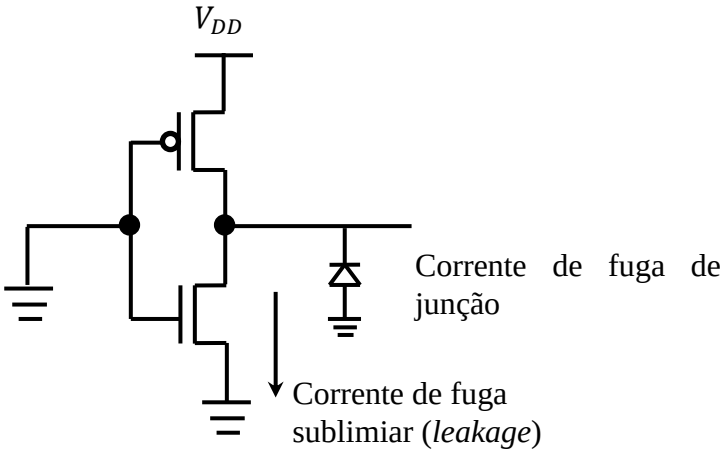
$$=P(Out=0).P(Out=1)$$

$$=\frac{1}{4}.\frac{3}{4}=\frac{3}{16}$$

$$nC=\frac{3}{16}C$$

Na prática, existe pequena corrente quando um transistor CMOS é alimentado, devido à polarização reversa das junções PN associadas com os transistores MOS e também devido às correntes de fuga sublimiares (Figura 10). O componente de *leakage* é correlacionado à área dos dispositivos, à temperatura e à tensão sublimiar (V_T).

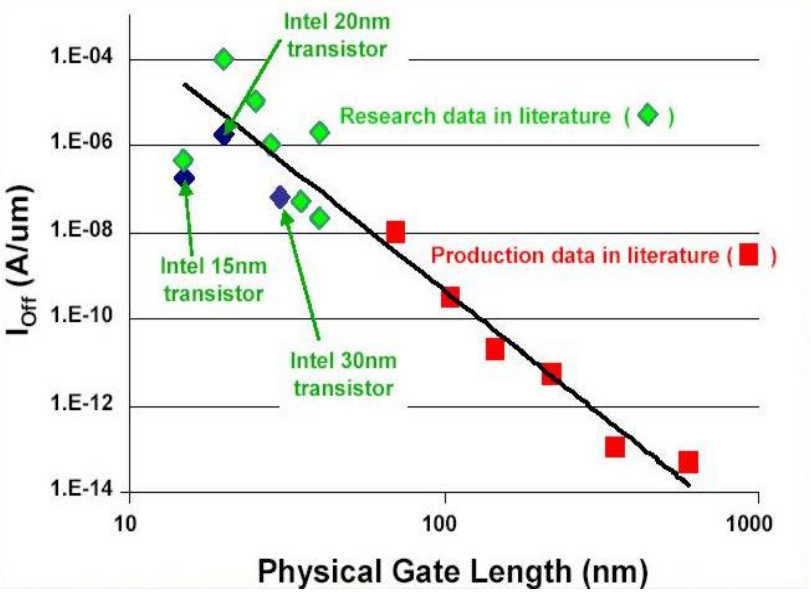
Figura 10 - Representação da corrente de fuga (*leakage*)



Selecionando tecnologias com dimensões submicrométricas ainda menores e as tensões de operação menores, o consumo de potência estático P_{leak} dos transistores aumenta e toma proporções que não podem mais ser ignoradas. A Figura 11 mostra a evolução da corrente de fuga com a diminuição do comprimento do canal do transistor. Ou seja, dependendo da tecnologia escolhida para o projeto, esses consumos de potência estático podem ou não ser considerados durante a avaliação de consumo de potência.

$$P_{leak} = I_{fuga} V_{dd} \tag{5}$$

Figura 11 - Evolução da corrente de fuga nos transistores CMOS



Em blocos de memória que ocupam bastante área de silício, a corrente de fuga deve ser calculada com cuidado, pois essas correntes chegam a representar a ordem de grandeza de 25% do total de corrente de fuga de todo o dispositivo.

2.2 Pontos a se considerar no planejamento de consumo de potência

Em relação aos consumos de potências dinâmicos e estáticos, nos dias atuais, existem muitos outros pontos a considerar em projetos de circuitos integrados:

- Tensão de operação
- Frequência de operação
- Custo de encapsulamento
- Engenharia de teste
- Aplicação do dispositivo
- Desempenho do dispositivo

No consumo de potência dinâmico de chaveamento P_{dyn-c}, podemos observar que a tensão de operação impacta significativamente no seu valor – e de forma quadrática. Assim, colocar um regulador de tensão para que esses transistores utilizem a menor tensão necessária para o funcionamento adequado pode significar boa economia em consumo de potência.

Se a frequência de operação for diminuída de alguma forma também implica em redução do consumo de potência. Nos dispositivos atuais existem várias técnicas pelas quais a frequência de operação pode ser diminuída dinamicamente, dependendo da funcionalidade do dispositivo naquele momento; isto contribui para diminuição do consumo de potência médio do circuito. Muitos dispositivos, como telefones celulares, utilizam sensores para definir o uso do dispositivo ajustando dinamicamente a frequência de operação do circuito.

A potência máxima de consumo define o tipo de encapsulamento e de resfriamento necessário para o dispositivo desenvolvido. Devemos considerar a necessidade de diminuir os custos de encapsulamento. Consumo baixo de potência significa que a necessidade de dissipação térmica também é menor e diminui a área/tecnologia necessária para esse fim, permitindo produtos finais mais compactos e simples de confeccionar.

Hoje, condições de consumo de potência são também levadas em conta nos circuitos de BIST (*Built In System Test*) [10] e de teste em fábrica [13], apesar de, no ambiente controlado de fábrica, não se estar preso à quantidade de energia necessária para a realização de testes. Os circuitos internos específicos para teste têm determinado consumo de potência e,

se o consumo de potência for alto, isso pode acarretar em encapsulamento mais caro para suportar essa demanda de dissipação, mesmo que seja somente utilizado em teste.

Em muitos casos, no modo funcional, esse consumo de potência nunca seria atingido e não haveria necessidade de encapsulamento mais caro; assim, planejamento para a realização de testes também deve ser levado em conta como parte de projeto.

A possibilidade de vários modos de operação também é muito importante na integração dos SOC [14], um dispositivo de baixo consumo de potência "*low power*" não está somente atrelado ao consumo de potência médio ou instantâneo de potência, mas também ao ambiente e à aplicação que terá esse dispositivo no mercado. Por exemplo, nada adianta um chip para telefone celular com o menor consumo médio de potência se não houver um modo de *standby* no qual, em aplicação real, possa estar 95% do tempo.

Outro ponto importante a abordar é o desempenho do dispositivo. Hoje, os dispositivos estão limitados em desempenho, devido à tecnologia de dissipação térmica do encapsulamento. Assim, diminuir o consumo de potência ou distribuir a dissipação térmica pode significar o aumento do desempenho do dispositivo. Nesse sentido, atualmente, o mais comum é a utilização de vários núcleos de processamento, para distribuir a potência dentro do silício e, assim, aumentar o desempenho dos dispositivos – não na forma de processamento linear, mas na de processamento paralelo.

Pelo ponto de vista de aplicação de dispositivos portáteis, o tempo de bateria é usado para se medir o consumo médio de potência, apesar de ser muito difícil de se definir esse consumo de potência, devido à falta de padrões bem determinados no mercado.

2.3 Técnicas para redução de consumode potência

Existem muitas técnicas aplicadas para a diminuição de consumo de potência nos SoC atuais. Vamos abordar as seis técnicas mais comuns aplicadas a CMOS nos dias atuais:

- Redução do chaveamento de estados dos transistores
- Controle de memória e interfaces de blocos
- Circuitos assíncronos
- Planejamento físico e de tecnologia
- Arquitetura do sistema
- Domínios de alimentação

2.3.1 Redução do chaveamento de estados dos transistores

Nos circuitos atuais digitais CMOS, a parte sequencial é normalmente o que mais contribui no total da potência consumida. O consumo de potência dinâmico de potência está ligado diretamente à frequência de operação; caso se consiga realizar a mesma aplicação com frequência de operação menor, isto significa que a consumo de potência é menor.

Circuitos específicos podem realizar tarefas em menor número de ciclos de relógio, isto é, realizar a tarefa em menos tempo, diminuindo a frequência de operação e acarretando na diminuição do consumo de potência. Esses circuitos específicos poderiam conter simplificações de circuito que minimizem a quantidade de chaveamento necessário para realizar a tarefa, também levando à redução no consumo de potência. Incluir dispositivo tipo DSP pode ser interessante na redução de consumo de potência [15], principalmente para aplicações específicas, como criptografia, decodificadores de vídeo e processamento de sinais.

O melhoramento de arquitetura/algoritmo, para que haja menos transições, também reduz o consumo de potência, isto é, fazer funções minimizadas ou com controles que diminuam o chaveamento dos transistores diminui o consumo de potência.

Nos processadores, o conjunto de instruções pode ser otimizado, de forma que haja menor quantidade de chaveamentos. Se for possível diminuir a mudança de estados espúrios de partes de circuitos que não estão sendo utilizados, pode-se conseguir considerável economia no consumo de potência. Essa diminuição pode ser atingida pela arquitetura de sistema previamente planejada, utilizando sinais de controle para disparar o próximo ciclo da função. Assim, se a função não é requerida num dado ciclo, basta garantir que os sinais de controle e entradas se mantenham estáveis, eliminando toda cadeia de chaveamentos de estado da função.

Esse tipo de economia de consumo de potência também pode ser atingido utilizando-se *gated clocks*, isto é, não fornecendo sinal de relógio quando não necessário. Os benefícios na diminuição do consumo de potência são devido aos seguintes fatores:

- Os registradores que recebem esse sinal de relógio controlado (*clock gated*) não são acionados em ciclos desnecessários.
- A carga na árvore de relógio é diminuída, diminuindo o número de *buffers* necessários para alimentar a árvore de relógio.
- As funções lógicas das entradas podem ser simplificadas, uma vez que existe a condição na qual o relógio nos registradores não transiciona; assim, mesmo que

suas entradas desses registradores variem, não há perigo que esses valores se propaguem para suas saídas, causando o chaveamento de estados desnecessários em sua cadeia.

Atualmente, o *gated clock* é um dos principais métodos utilizados para reduzir o número de chaveamentos. A otimização da arquitetura e o bom projeto da rede de *clock gating* podem contribuir bastante na redução do consumo de potência. Essa técnica pode ser feita automaticamente, pelas ferramentas de síntese de circuito, ou manualmente, durante o desenvolvimento do circuito, criando sinais de controle específicos para essa função. Criar esses controles durante o desenvolvimento gera circuitos melhores no ponto de vista de consumo de potência, devido à preocupação com consumo de potência nos estágios iniciais do projeto. Somando-se a inserção automática de gated clock na síntese do circuito, atinge-se um circuito ótimo dessa técnica.

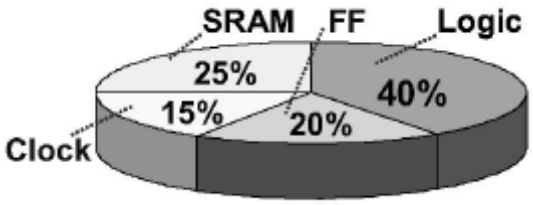
Outro ponto interessante a abordar é o planejamento cuidadoso dos multiplexadores de barramento no caminho de dados. Esses multiplexadores selecionam um dos possíveis dados para um barramento único. Se não planejados com cuidado, podem acarretar em transições indesejadas na entrada dos multiplexadores, propagando para a saída do barramento único e, consequentemente, para o restante dos circuitos ligados ao barramento [16].

A solução mais comum para os multiplexadores é manter o valor atual até que a próxima mudança seja necessária ou chavear para um valor constante, quando não utilizado. Quando esses multiplexadores não são utilizados por muitos ciclos de relógio, redução no consumo de potência pode ser observada, mas, se o multiplexador é utilizado com grande frequência, essa técnica não proporciona grande economia de consumo de potência.

2.3.2 Controle de memória e interfaces de blocos

Uma das fontes de consumo de potência é o acesso a bloco de memória: quanto maior for o tamanho do bloco de memória, maior é o consumo de potência devido ao acionamento das linhas de multiplexação dos endereços e dados [17]. Um exemplo da divisão do consumo de potência de um processador está ilustrado na Figura 12. O bloco de memória representa 25% do consumo de potência total, a segmentação em blocos menores de memória e a redução no acesso a elas podem representar diminuição no consumo de potência.

Figura 12 - Divisão do consumo de potência em um processador



Outro ponto a considerar é que, no lado externo ao integrado, outros blocos que fazem interface para compor o sistema funcionam com IO de 5V, 3.3V e 2.5V. A integração em conjunto desses blocos (SOC) externos pode representar menor consumo de potência, devido à diminuição do consumo de potência de IO desnecessários. Essa integração também diminui os custos de placa de circuito impresso, montagem e testes em relação a ter vários componentes para um determinado sistema [18]. A integração de dispositivos em um só também pode ser vantajosa no que diz respeito a compartilhamento de otimizações de circuitos digitais internos, redução de registradores entre blocos de circuitos e reaproveitamento de circuitos de alimentação e controles, tanto digitais como analógicos.

Hoje, podemos observar que existe forte tendência de criação de SOC para qualquer tipo de aplicação; um dos grandes motivadores para essa integração é que diminui muito o consumo de potência e o tamanho dos dispositivos finais – características fundamentais dos dispositivos portáteis.

2.3.3 Circuitos assíncronos

A abordagem dos circuitos assíncronos é totalmente diferente, quando comparada com a dos circuitos síncronos, e tem grande potencial na diminuição do consumo de potência [19]. Nos circuitos assíncronos, são utilizados sinais para indicar o término das operações, diferentemente dos circuitos síncronos, que utilizam, de forma cadenciada, o sinal de relógio.

Isso não quer dizer que não existam elementos de memória (registradores) nos circuitos assíncronos; significa que o armazenamento, nos estágios de armazenamento, não é cadenciado por um sinal de relógio.

As vantagens, em relação aos circuitos síncronos, são:

- Não há necessidade de árvore de relógio.
- Não existem transições espúrias nos registradores.
- Devido ao relógio de elementos de memória ser controlado diretamente por um sinal de controle gerado combinacionalmente, o *clock gating* é automático nas regiões não utilizadas.

- Diminuição das correntes de pico, uma vez que as transições ocorrem em tempos diferentes.
- Adaptação automática à variação de temperatura e tensão.
- Arquitetura modular e flexível.
- Pode atingir desempenhos melhores em condições normais de operação.

Assim, a tecnologia de circuitos assíncronos é mais eficiente no que diz respeito a consumo de energia por um fator significativo, conforme Figura 13, Figura 14 e Figura 15, nas quais há comparação de um processador 8051 que utiliza o mesmo método de projeto de circuitos assíncronos utilizado no processador assíncrono de comparação deste trabalho. Nas citadas figuras, o modelo assíncrono está representado pelos gráficos à esquerda e o modelo síncrono está representado pelos gráficos à direita.

Figura 13 - Gráfico do consumo de potência e corrente: (a) assíncrono e (b) síncrono

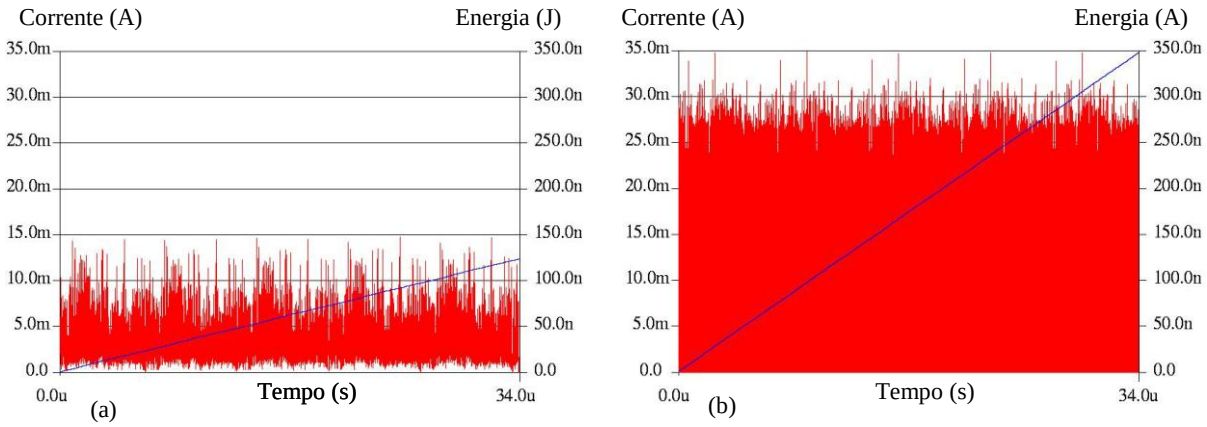


Figura 14 - Gráfico da corrente de pico: (a) assíncrono e (b) síncrono

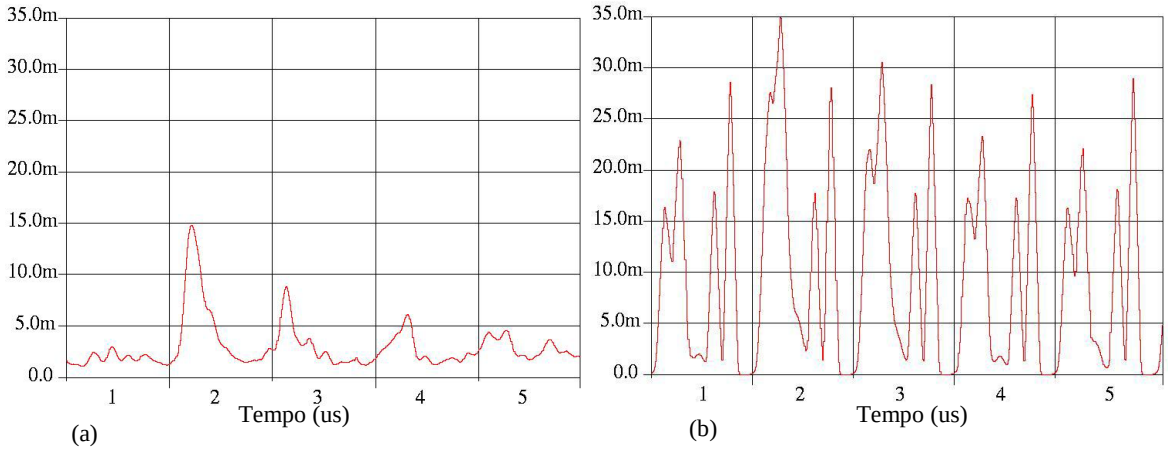
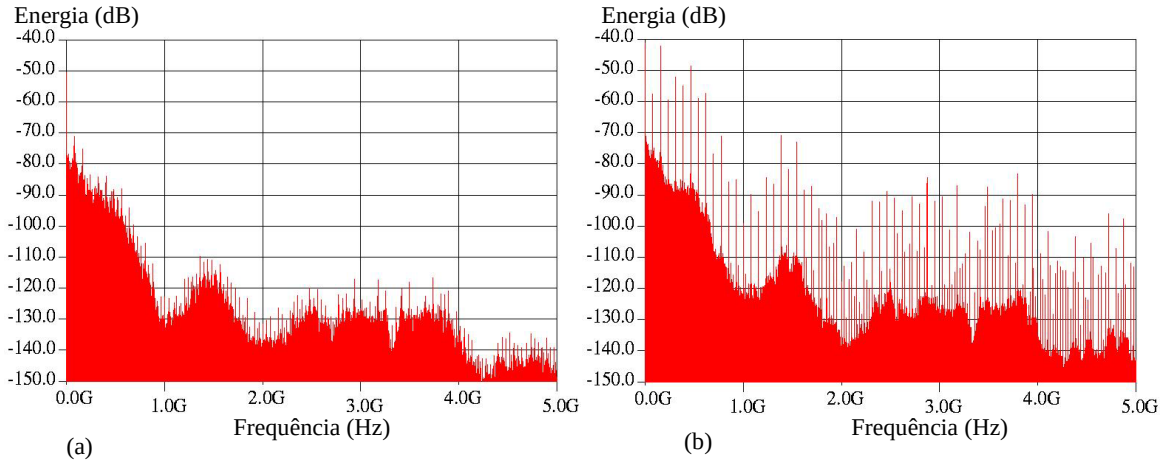


Figura 15 - Gráfico da emissão eletromagnética: (a) assíncrono e (b) síncrono



Uma importante vantagem que podemos ressaltar dos circuitos assíncronos é a possibilidade de se variar a tensão de operação dentro de certos níveis sem causar o mal funcionamento do circuito. Assim, podemos otimizar o consumo de potência somente ajustando a tensão de operação.

Muitos tipos de abordagens para criação de circuitos assíncronos já foram propostos [17][20]. Um dos problemas para se trabalhar com circuitos assíncronos é que as ferramentas e as metodologias de projetos atuais não estão adaptadas para se utilizar essa tecnologia. Quanto mais complexo o dispositivo desenvolvido, mais difícil é realizá-lo em tecnologia assíncrona.

O método utilizado no processador assíncrono, descrito neste trabalho, foi da empresa Handshake. Esse método era composto de um conjunto de scripts e uma linguagem de programação proprietária que gerava o circuito assíncrono através do código descrito nessa linguagem. Os *scripts* eram adaptações feitas utilizando as ferramentas convencionais do fluxo de projeto, fazendo ajustes nos caminhos de controle dos circuitos assíncronos.

2.3.4 Planejamento físico e de tecnologia

Planejar qual será a tecnologia utilizada para realização do dispositivo faz parte do planejamento para redução de consumo de potência. Dependendo da aplicação e do mercado final do dispositivo desenvolvido, existe a relação frequência de operação e tamanho dos transistores. Levando-se em conta esses dois parâmetros, existe a possibilidade de otimizar o tamanho dos transistores das células da tecnologia escolhida [21].

A técnica de dimensionamento dos transistores ataca a potência consumida por curto circuito P_{cc}, alargando ou estreitando o comprimento do canal do transistor [22]. O efeito da

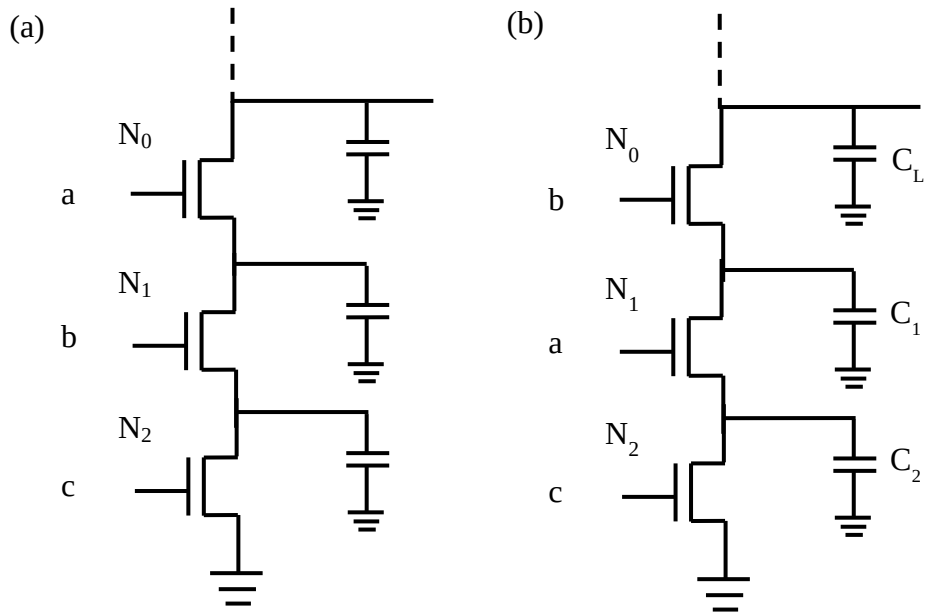
mudança do canal pode ser observado como troca de desempenho por baixo consumo de potência.

No âmbito do consumo de potência estático, existem bibliotecas que atuam em células de baixa corrente sublimiar (*leakage*). Células de bibliotecas com V_T normal e com largura de canal L "grande" são somente 30% mais lentas do que a mesma biblioteca com V_T normal e com comprimento de canal L mínimo; contudo, o *leakage* desta célula é 20 vezes menor do que o segundo. Também esta célula é 30% mais rápida do que a célula utilizando V_T mais alto e com comprimento de canal L mínimo, o qual tem *leakage* somente 2 vezes menor do que o segundo.

A escolha correta das células a serem utilizadas pode acarretar na diminuição do consumo de potência e atingir o desempenho desejado. Assim, se o circuito opera em frequência bem baixa, a escolha por utilizar celulas com baixo *leakage* é a certa. Se o circuito funciona em frequência muito alta, durante curtos períodos de tempo, a melhor escolha são células de baixo V_T [23].

Outra técnica é o reordenamento dos transistores, que tenta diminuir a potência consumida por chaveamento dos transistores P_{sc}. A ideia básica dessa técnica é atingir potência consumida menor, ajustando a ordem dos transistores conectados numa cadeia em série, baseando-se no comportamento das várias entradas. A Figura 16 ilustra essa situação e mostra implementações diferentes na rede N da cadeia de transistores (NMOS) de uma porta NAND de três entradas.

Figura 16 - Ilustração de ordenação de entradas na rede de transistores



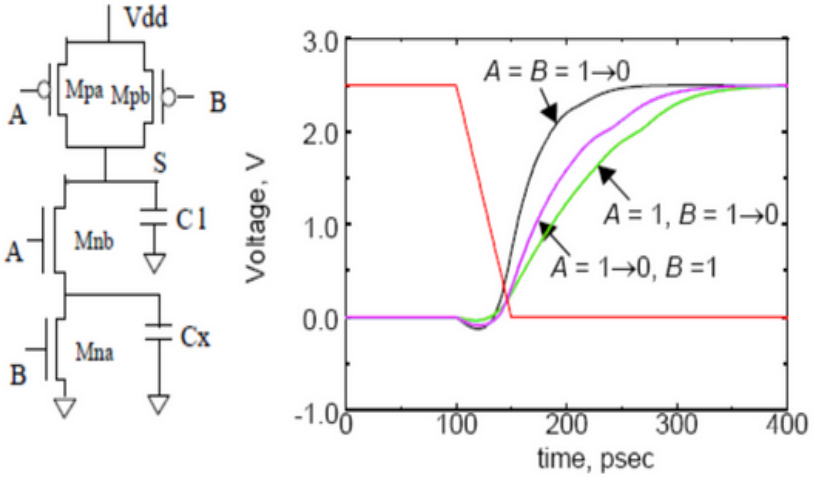
Na Figura 16, três entradas (**a**, **b** e **c**) são interligadas à entrada desses transistores. Assume-se o vetor “111”, seguido do vetor “011”, aplicados às entradas em ordem alfabética – e considerando que todas as entradas cheguem simultaneamente. Na configuração (a), esta sequência causa o descarregamento das capacitâncias C_L, C_1 e C_2, no primeiro vetor, e, depois, das capacitâncias C_1 e C_2, no segundo vetor. Na configuração (b), o descarregamento é somente da capacitância C2. Assim, a mesma sequência de entrada pode causar diferentes consumos de potência, quando interligados em ordens diferentes.

Para determinar a ordem apropriada dos transistores, deve-se ter informação de como as entradas se comportam. Essa informação pode ser levantada estatisticamente. Esse levantamento estatístico pode ser trabalhoso, no caso de circuitos maiores e mais complexos. Detalhes referentes à reordenação de transistores podem ser obtidos na literatura [21][24].

Uma regra básica para configuração NAND seria colocar a entrada com a menor probabilidade de transição no transistor NMOS perto do sinal terra. No caso, para NOR, seria colocar o sinal com menor probabilidade de transição no transistor PMOS perto da fonte.

As reordenações desses sinais também podem afetar o desempenho do circuito, dependendo dos atrasos dos sinais envolvidos e de qual dos sinais chegue primeiro a cada transistor.

Figura 17 - Atraso do chaveamento de portas



Input Data Pattern	Delay (psec)
A = B= 0→1	69
A = 1, B= 0→1	62
A= 0→1, B = 1	50
A=B=1→0	35
A=1, B = 1→0	76
A= 1→0, B = 1	57

Observando a porta NAND2 da Figura 17, o atraso no chaveamento da porta é menor quando o sinal B chega primeiro (A=0→1 e B=1) e maior quando o sinal A chega primeiro (A=1 e B=0→1). Assim, como regra geral para se ter o maior desempenho, deve-se colocar o sinal que tem maior atraso mais perto da saída da porta [25].

Técnicas de *layout* e circuito podem contribuir para a redução da capacitância entre as células da biblioteca. A utilização de *drivers* de tamanhos menores e distribuídos ao longo dos

caminhos com comprimentos longos pode ser melhor opção em termos de consumo de potência que a utilização de *drivers* de tamanhos maiores em menor quantidade. Já a utilização de tensões de operação menores pode reduzir bastante o consumo de potência, uma vez que a tensão tem relação quadrática com a potência consumida. Um ponto importante no desenvolvimento de SOC são os reguladores de tensão internos ao circuito integrado. Esses reguladores mantêm a tensão fornecida ao circuito em ponto mínimo para o bom funcionamento do circuito desenvolvido, mantendo o consumo de potência em níveis mínimos para dada tecnologia.

$$P_{dyn-c} = nCV_{DD}{}^2 f \tag{6}$$

A redução da tensão de alimentação acaba aumentando o atraso do circuito. O atraso de propagação é inversamente proporcional a $V_{DD} - V_T$, como é mostrado na equação abaixo [26].

$$T_g = K \frac{V_{DD}}{(V_{DD} - V_T)^{\alpha}} \tag{7}$$

Onde:

K é uma constante proporcional à tecnologia, α é o fator de potência da equação de corrente do dreno.

Podemos observar que a diminuição do V_T diminui o atraso do circuito. Contudo, essa diminuição acarreta no aumento da corrente de *leakage*.

$$I_l = W_{eff} I_s e^{\frac{V_T}{V_0}} \tag{8}$$

Onde:

W_{eff} é a largura efetiva do canal do transistor, I_s é a corrente de *leakage* zero e V_0 é a variação da tensão V_{GS} por década de aumento da corrente de dreno I_D na região $V_{GS} < V_T$.

O benefício ganho pela diminuição da tensão de alimentação V_{DD} acaba sendo reduzido devido ao aumento da dissipação de potência pela corrente de *leakage* [27].

Existem vários métodos que tentam otimizar a corrente de *leakage*. Algumas técnicas se baseiam em vários tipos de células, com V_T maiores e menores em sua base de células básicas, sendo que as células de V_T maiores são utilizadas em circuitos que necessitam de menor desempenho e menor corrente de *leakage*. Existem outras técnicas, como tensão de alimentação e tensão sublimiar variável [23], que também utilizam V_{DD} e V_T variáveis para diminuir a potência consumida dentro de parâmetros aceitáveis de atraso do circuito.

Utilizar tecnologias como *Silicon On Insulator* (SOI), que tem alto desempenho em tensões baixas e com baixas capacitâncias parasitas, pode ser uma boa solução. O controle da tensão de substrato pode reduzir a corrente de *leakage*. Assim, dependendo da aplicação que o dispositivo terá, podemos fazer escolhas que melhor atendam ao perfil de consumo de potência da aplicação. Por exemplo: se o dispositivo será utilizado em um sistema que funciona por curto período de tempo e depois fica grande parte em espera, podemos entender que o consumo de potência pela corrente de *leakage* será predominante no total da potência consumida. Esse dado nos levará a utilizar uma solução tecnológica onde a corrente de *leakage* seja baixa, contribuindo para menor consumo de potência.

Até mesmo ferramentas de desenvolvimento desempenham papéis fundamentais na redução de consumo de potência. Como a linguagem HDL não possui nenhum tipo de representação para projeto de baixo consumo de potência, os próprios fabricantes têm criado padrões para a realização dessas tarefas, como é o caso do UPF (*universal power format*), da Mentor Graphics [26], e o CPF (*commun power format*), da Cadence, entre outros formatos no mercado.

2.3.5 Arquitetura do sistema

A redução de consumo de potência através de técnicas de algoritmo foca em minimizar o número de operações realizadas para determinada tarefa. Normalmente, se leva em consideração custo da energia contra tamanho da implementação, mas, de forma geral, reduzir o número de operações é o objetivo. Em alguns casos, podem aparecer situações nas quais fazer operações aritméticas seja mais eficiente do que várias releituras de memória.

Na forma de arquitetura do sistema, pontos como a utilização de representação em sistema, ponto flutuante ou ponto fixo, levam a diferentes resultados no consumo de potência, quando se realizam operações aritméticas.

A utilização de arquitetura que se beneficia de paralelismo e *pipeline* é importante na minimização de potência. O aumento de níveis de *pipeline* pode acarretar no aumento da latência e no aumento de circuito para controle dos vários níveis de dependência. Na procura por maximizar o desempenho e diminuir o consumo de potência, os sistemas complicados de predição e pré-carga são utilizados nos processadores para tentar evitar o acesso à memória desnecessariamente. Muito cuidado deve ser tomado quando se utilizam técnicas desse tipo, pois existe aumento no circuito de controle para realizar essas tarefas. Em muitos casos, o consumo de potência extra do circuito pode ser maior do que a economia resultante pela

mudança da arquitetura. Como regra geral, quando os níveis de *pipeline* são grandes, costuma-se perder toda a vantagem no que diz respeito ao consumo de potência do circuito.

Um bom planejamento da arquitetura dos blocos que compõe o sistema é fundamental para se obter um sistema com consumo de potência reduzido, não criando circuitos desnecessários – que não serão utilizados na aplicação final, acarretando no aumento do consumo de potência e área. Muitas vezes, são criados circuitos que atendem a várias necessidades, mas, no final da aplicação, o circuito é utilizado de uma única forma – todo o resto do circuito fica sem utilização.

2.3.6 Regiões de alimentação (*power domain*)

Atualmente, existe forte demanda para integração de muitos blocos funcionais em um único chip. Um exemplo clássico seria um chip para telefone celular, no qual, internamente, temos blocos de memória, GPS (*Global Position System*), codificadores e decodificadores de vídeo, aceleradores gráficos, circuitos de RF, câmera, sensores e seus circuitos relacionados, entre outros blocos. São necessárias técnicas de *power down* parciais, nos quais blocos inteiros sejam desligados até o momento que haja a necessidade de utilização dos mesmos.

Do ponto de vista dinâmico, esses blocos podem estar em estado de espera, sem o fornecimento de relógio; contudo, do lado estático, ainda existe consumo de potência. Em tecnologias mais avançadas, esse consumo de potência estático já não é desprezível em relação ao consumo de potência dinâmico, como mostrado anteriormente.

O melhor jeito de diminuir esse consumo de potência estático é utilizar a técnica de alimentação parcial, na qual blocos funcionais inteiros não estariam sendo alimentados. Se não há alimentação, não há consumo de potência estático, muito menos consumo de potência dinâmico. Nessa técnica, devemos ter grande planejamento entre os sinais em torno dos blocos, para que não haja correntes diretas entre transistores PMOS e NMOS, uma vez que o estado dos sinais em torno pode estar instável devido ao desligamento da alimentação dessas células. Nesses sinais, é necessária a inserção de isoladores entre os sinais que cruzam os domínios de alimentação. Também deve-se tomar muito cuidado no planejamento de árvores de relógio e sinais que precisam cruzar esses vários domínios de alimentação: não seria adequado que um *driver* da árvore de relógio, que alimenta uma região ativa, esteja desligado devido ao desligamento de outra região. As questões levantadas acima tornam muito difíceis o planejamento do *floorplan* para esse tipo de técnica.

Nos *chips* atuais, existem integrações com regiões de alimentação distintas, da ordem de 20 [15]. As ferramentas já implementam vários procedimentos para que o tratamento dos

domínios seja feito de forma mais simplificada e automática. Essa técnica de alimentação parcial normalmente é utilizada com aplicação de mais alto nível de *power management* feita por software no qual este comanda e controla as várias regiões de alimentação do chip.

2.4 Direcionamento desse trabalho

Para se fazer um processador que execute um conjunto de instruções, é crítica a definição do *Instruction Set Architecture* (ISA), o qual especifica a funcionalidade que o processador deve executar. Assim, a definição do ISA é crucial no desenvolvimento do processador.

Qualquer projeto de engenharia inicia-se com a especificação do comportamento que estamos desejando (o que faz o projeto). A implementação é a descrição estrutural do projeto de como é construído.

Nesse trabalho, vamos dividir o projeto em duas partes: síntese e análise. A síntese compreende a parte na qual vamos tentar achar uma implementação baseada na especificação e a análise examinará a implementação para determinar se atende à especificação.

A síntese é a parte na qual podemos definir as soluções; é onde podemos fazer as escolhas necessárias, baseadas em trocas de desempenho, área e otimizações, e escolher a melhor solução. Na análise, poderemos utilizar as ferramentas de mercado e determinar se a implementação está correta e se atende as especificações determinadas.

Foi feita a escolha de se desenvolver um processador sicrono com microarquitetura melhorada. Existem alguns motivos que levaram a essa abordagem, um dos quais é a frequência de operação desses processadores. O processador assíncrono desenvolvido anteriormente indicava que não poderia cumprir os requisitos de frequência dos dispositivos comerciais atuais. Assim, a abordagem de um processador assíncrono seria para uma linha de produtos totalmente nova. Outro ponto foi que a economia de consumo de potência não era significativamente maior em relação a abordagem sicrona. Os circuitos, extras para atender os requisitos de circuito assíncrono, consumiam muita área, acarretando em um processador com área de silício maior do que o processador original.

Os estudos anteriores com os processadores síncronos mostravam a possibilidade de melhorias que ainda não tinham sido abordadas e que poderiam trazer alguns ganhos em consumo de potência ou desempenho do processador.

O desenvolvimento de um processador começa com a definição do ISA, o qual especifica o conjunto de instruções que o processador deve executar. O desenvolvimento de processadores envolve basicamente dois passos: a criação da microarquitetura do processador

e o projeto lógico desta. Neste trabalho, o projeto lógico será implementado usando-se a HDL (*hardware description language*) Verilog e cada arquivo conterá desde primitivas básicas, como NAND e OR, até mesmo módulos funcionais completos, como somadores, multiplicadores, multiplexadores.

Daremos ênfase maior na microarquitetura, pois acreditamos que algumas escolhas, como caminhos de dados (*data path*), estruturas de barramento e circuitos de controle, podem influenciar significativamente na criação de um processador que atenda o objetivo de ter consumo de potência mais eficiente. O resultado final da microarquitetura é uma descrição em alto nível da organização do processador.

Existem muitos ISA bem conhecidos no mercado, como Motolora 68K, Power PC, IBM 360 e Intel IA32. Cada um destes ISA tem atributos associados à linguagem assembly, formato da instrução, modos de endereçamento e modelos de programação que compõem, sendo importante ressaltar que cada ISA pode ter várias implementações que são determinadas na sua microarquitetura. Contudo, todas as implementações executam o mesmo programa codificado num determinado ISA. Cada implementação de um determinado ISA tem associado atributos como projeto de memória *cache*, projeto de *pipeline*, *multicores*, entre outros. Normalmente, os detalhes da microarquitetura são implementados em hardware e transparentes para o software.

Os atributos relacionados para cada implementação estão diretamente ligados à necessidade específica do propósito que o chip será utilizado. Dependendo do objetivo, cada implementação pode diferir em termos de requisitos de frequência de relógio, capacidade de memória, interface com periféricos, tecnologia, encapsulamento, etc.

2.4.1 Escolha do *Instruction Set Architecture* (ISA)

Desde os primeiros processadores, muitos ISA têm sido desenvolvidos. Eles diferenciam, entre si, como os operadores e operandos são especificados. Tipicamente, um determinado ISA define um conjunto de instruções, chamados de instruções *assembly*. Cada instrução define um operador e um ou mais operandos. Cada ISA define uma linguagem *assembly* distinta.

Um programa numa determinada linguagem *assembly* é uma sequência das instruções assembly definida num determinado ISA. Estes ISA costumam evoluir lentamente, com o tempo, devido à inércia da necessidade de recompilar o código previamente escrito. Geralmente, essa recompilação só acontece quando existe grande vantagem ou necessidade na execução do código escrito.

Importante ressaltar que, durante a vida de um determinado ISA, acontecem adições de instruções de tempos em tempos, para acomodar novas utilizações de aplicações emergentes da sociedade. Vale lembrar, também, que a adoção de um ISA totalmente novo vai muito além de uma atualização de um ISA preexistente e que isso ocorre de forma muito mais rara. Um dos motivos de ser tão rara a adoção de novo ISA é devido ao longo tempo de desenvolvimento de sistemas operacionais, compiladores e *software* de apoio – em média, demoram mais de 10 anos.

Quanto mais antigo um determinado ISA, maior a base de *software* de compilação, sistema operacional e *software* de apoio existente, tornando, assim, a reposição deste ISA cada vez mais difícil.

Quanto à implementação de ISA, novas implementações são desenvolvidas, em média, a cada 3 a 5 anos. Na década de 1980, houve grande interesse da comunidade de se formular o melhor ISA. Nos dias atuais, o foco mudou mais para a implementação de microarquiteturas mais inovadoras que se aplicam à maioria dos ISA existentes.

Nesse trabalho, poderíamos escolher muitos tipos de ISA presentes no mercado, como ARM, ISA 32 da Intel, MIPS, mas optamos pelo ISA da Freescale usado nos processadores S08, o qual é um processador de 8bits. A motivação para essa escolha está baseada na simplicidade de ser um processador de 8bits para o desenvolvimento do trabalho; além disso, há o fato de que já foram realizados alguns trabalhos para diminuição de consumo de potência e área dentro da Freescale, dentre os quais o autor desta tese foi membro participante.

Outros pontos que fundamentam a escolha é a familiaridade de trabalhar com esse processador e se ter acesso aos dados obtidos nos trabalhos anteriores. Por fim, há indicações que apontam a possibilidade de melhoria no consumo de potência. Possuir esses resultados dos trabalhos anteriores facilita no momento de fazer a comparação com os projetos de microarquitetura anteriores, identificando se houve ou não ganho na proposta de diminuir o consumo de potência do processador.

Apesar de estarmos usando um processador simples, os conceitos discutidos nesse trabalho poderiam ser estendidos para outros ISA mais complexos.

Nos trabalhos já realizados, temos:

1- Versão original do S08

2- Versão assíncrona

3- Versão com diminuição de registradores (reprojeto do mesmo processador com melhorias para área e consumo de potência)

O processador proposto é de configuração CISC (*Complex Instruction Set Computer*). Nesse trabalho não entraremos muito em detalhes sobre se é melhor utilizar a arquitetura CISC ou RISC (*Reduced Instruction Set Computer*), pois já existem muito trabalhos nessa área [28] [30]. Nos dias atuais, o mais predominante é um híbrido entre CISC e RISC, ou seja, não existe mais uma clara definição dentre essas duas arquiteturas.

2.4.2 Conceitos básicos do processador S08

O S08 é um processador de 8bits usando o ISA do 68HC08. Assim, o conjunto de instruções do HCS08 é totalmente compatível com o do 68HC08; por conta disso, o *assembly* gerado pode ser intercambiável entre eles. A diferença entre esses dois processadores é que o 68HC08 era uma versão não sintetizável e a versão HCS08 já é uma versão em linguagem de descrição de circuito.

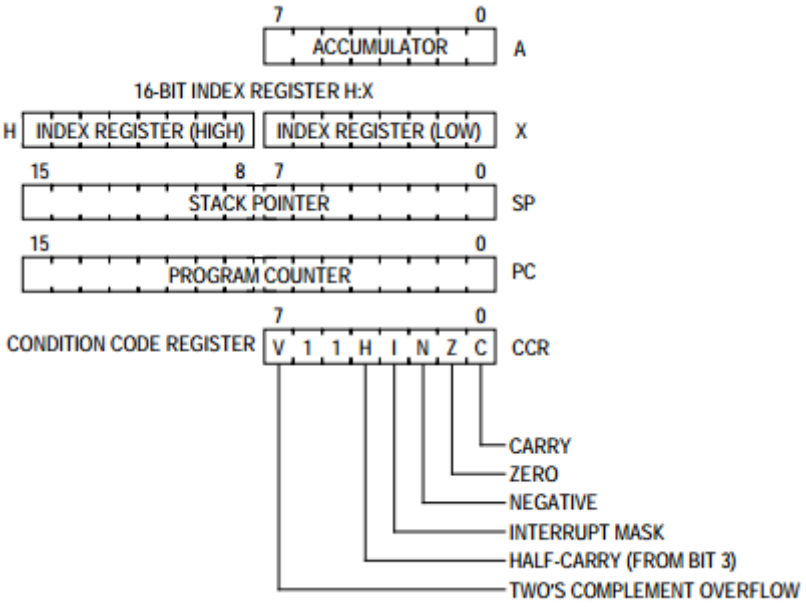
Características do processador HCS08:

- Código de objeto totalmente compatível com as famílias M68HC05 e M68HC08.
- Espaço de endereçamento de 64KB, com possibilidade de ser maior do que 64KB.
- *Stack pointer* (SP) de 16 bits.
- Registrador de indexação de 16 bits, com vários modos de endereçamentos.
- Acumulador de 8 bits.
- Sete modos de endereçamento:
    1. Inerente
    2. Relativo
    3. Imediato
    4. Direto
    5. Estendido
    6. Indexado relativo a H:X
    7. Indexado relativo ao SP (*stack pointer*)
- Instrução de movimentação de memória para memória com quatro tipos de combinação de modos.
- Sinalização de *overflow*, *half-carry*, negativo, zero, condição de *carry* e suporte para *branch*.
- Instruções eficientes para manipulação de bits.
- Multiplicador de 8x8 bits e divisor 16x8 bits.

- Instrução de STOP e WAIT para modos de baixo consumo de potência.

Existem 8 registradores, definidos e separados conforme Figura 18.

Figura 18 - Registradores do processador S08



2.4.2.1 Acumulador (A)

O acumulador é um registrador de propósito geral de 8bits. Um dos operadores da ULA (Unidade Lógica Aritmética) é conectado ao acumulador e o resultado normalmente é registrado no acumulador após a realização da operação aritmética.

O acumulador pode ser carregado da memória utilizando-se vários modos de endereçamento para especificar o endereço de onde vem o dado. Da mesma forma, o dado registrado em A pode ser gravado na memória, utilizando-se vários modos de endereçamento para definir o endereço a ser gravado.

2.4.2.2 Registrador de indexação (H:X)

O registrador de indexação tem 16 bits, está dividido em dois registradores de 8 bits (H e X), que quase sempre trabalham juntos, compondo um endereço de 16 bits, onde H contém os bits mais significativos do endereço e X contém os bits menos significativos do endereço. Por questões de compatibilidade com M68HC05 (modelo anterior ao S08), algumas instruções somente operam nos bits menos significativos (X). Muitas instruções tratam X como um registrador de uso geral de 8 bits, que pode ser usado para manter valores de dados

de 8 bits. O registrador X pode ser apagado, incrementado, decrementado, complementado, negado, deslocado ou rotacionado. Instruções de transferência permitem a transferência de dados entre A, onde as instruções aritméticas e lógicas são realizadas, e o registrador X.

2.4.2.3 *Stack Pointer* (SP)

Stack Pointer é um registrador de 16 bits que aponta para a próxima localização disponível e utiliza configuração *last-in-first-out* (LIFO). O *stack* pode se localizar em qualquer parte do endereçamento de 64KB onde está localizada a RAM. O *stack* é utilizado para salvar automaticamente o endereço de retorno das chamadas de sub-rotinas, o endereço de retorno e os registradores da CPU durante as interrupções e para guardar valores de variáveis locais.

O SP tem o valor 0x00FF após o reset, para manter compatibilidade com a família M68HC05. No HCS08, o programa normalmente muda o valor de SP para a última posição do endereço de memória RAM durante a inicialização do *reset*.

2.4.2.4 *Program Counter* (PC)

O contador de programa é um registrador de 16 bits que contém o endereço da próxima instrução válida para leitura.

Durante a execução normal do programa, o contador de programa automaticamente incrementa para o próximo endereço sequencial toda vez que uma instrução (ou operação) é lida. O carregamento de endereços de *jump, branch*, interrupções e retornos, que tem valores diferentes ao próximo endereço sequencial, é chamado de mudança de fluxo.

Durante a inicialização, o PC é carregado com o valor do endereço do vetor de *reset* 0xFFFE e 0xFFFF. O valor guardado nesses endereços é o endereço da primeira instrução que será executada após a inicialização.

2.4.2.5 Registro de condições

Um registro de 8 bits contém uma sinalização de interrupção e cinco sinalizações sobre a operação da instrução que foi imediatamente executada. Duas sinalizações não utilizadas estão sempre setadas e reservadas para futuro uso.

V	1	1	H	I	N	Z	C

V - Sinalização de *overflow*

A CPU seta está sinalizada quando há overflow de operações com complemento de dois, caso contrário, este bit é zerado. Este bit é utilizado por instruções de mudança de fluxo BGT, BGE, BLE e BLT. Este bit também é setado nas instruções, ASL, ASR, LSL, ROL e ROR; contudo, não existe significado para o valor. A tabela completa das instruções pode ser encontrada no Anexo A.

H - Sinalização de *Half-Carrry*

A CPU seta esta sinalização quando há *carry* entre os bits 3 e 4 da ULA durante instruções ADD e ADC.

I - Sinalização de interrupção

Quando a máscara de interrupção é setada, todas as interrupções são desabilitadas. As interrupções são ativadas quando a máscara de interrupção é zerada. Quando a interrupção ocorre, a máscara de interrupção é automaticamente setada, logo após a CPU salvar os registros no *stack*, antes de o vetor de interrupção ser carregado.

N - Sinalização de negativo

A CPU seta esta sinalização quando uma operação aritmética, operação lógica ou manipulação de dados produzir resultado negativo.

Z - Sinalização de zero

A CPU seta esta sinalização quando uma operação aritmética, operação lógica ou manipulação de dados produzir resultado zero.

C - Sinalização *Carry/Borrow*

A CPU seta esta sinalização em operações de adição que produzem um "vai um" do bit 7 do acumulador ou quando uma subtração resultar no empresta um. Algumas outras instruções de manipulação também podem setar ou zerar esta sinalização.

Capítulo - 3. Princípios de desempenho de processadores

O desempenho de um processador está definido na equação abaixo [29]:

$$\frac{1}{Desempenho} = \frac{Tempo}{Programa} = \sum \frac{Instruções}{Programa} x \frac{Ciclos}{Instruções} x \frac{Tempo}{Ciclo} \quad (9)$$

Onde:

o programa é o conjunto de instruções já compilados para a ISA específica – esse conjunto de instruções realiza uma determinada tarefa;

o desempenho é definido como o inverso da quantidade de tempo necessária para execução de um determinado programa – este tempo de execução pode ser formulado de maneira simplificada em três subprodutos, instruções/programa, ciclos/instruções e tempo/ciclo;

o primeiro termo indica o número de instruções necessárias para determinado programa; ou seja, o número total de instruções para a realização da tarefa;

o segundo termo indica o número médio de ciclos de relógio necessários para realização de cada instrução, que é a média aritmética do número de ciclos de relógio necessária de todas as instruções desse programa;

o tempo de cada ciclo de máquina, basicamente, é o período do relógio utilizado no processador.

Quanto menor o tempo de execução, melhor o desempenho. Podemos aumentar o desempenho reduzindo qualquer 1 dos 3 fatores da equação. No caso de o número de instruções ser reduzido, como existem menos instruções, o tempo necessário para execução será reduzido. Se o número de ciclos por instrução for reduzido, o resultado do produto também será reduzido, levando, consequentemente, à redução do tempo necessário para execução. Se o tempo de ciclo for reduzido, que nada mais é do que aumentar frequência do relógio, isso também levaria à diminuição do tempo de execução.

Importante perceber que os três termos não são totalmente independentes e que existe uma complexa correlação entre eles. A redução de um dos termos pode acarretar na necessidade de aumento de um ou dos dois outros fatores da equação. As relações dos termos dessa equação não são de fácil caracterização. Podemos dizer que aumentar o desempenho

pode se tornar um grande desafio, pois envolve grandes trocas, sendo necessário um delicado balanceamento dessas trocas.

Também é importante, nesse momento, ter a percepção de que o desempenho está diretamente correlacionado ao consumo de potência do circuito – como foi apresentado anteriormente. Circuitos mais rápidos necessitam de células maiores para poder carregar e descarregar as capacitâncias de forma mais eficiente. Circuitos mais complexos, para serem mais rápidos, também acarretam em aumento de número de células e, assim, também aumentam o consumo de potência total.

3.1 Técnicas de otimização de desempenho

Toda técnica de otimização de desempenho acaba reduzindo um ou mais termos da equação de desempenho. Algumas otimizações podem reduzir um dos termos e deixar os outros intactos. Um exemplo é quando o compilador realiza otimização no código [31], eliminando redundâncias ou usando instruções que sejam mais eficientes, resultando em código objeto menor [32]. Assim, o número de instruções necessárias para a execução do programa deve ser menor, sem o impacto nos outros dois termos da equação – lembrando que a otimização não pode ser o reaproveitamento de código, pois estruturas, como loop, não necessariamente diminuem o número total de instruções executadas durante a realização de uma tarefa. Outro exemplo seria a migração para uma tecnologia mais avançada, que diminuísse os atrasos de propagação de circuitos; assim, o tempo de ciclo de máquina poderia ser diminuído através do aumento da frequência, reduzindo, com isso, o tempo de execução do programa.

Outras técnicas reduzem um dos termos, mas podem, ao mesmo tempo, levar ao crescimento dos outros dois termos. Para essas técnicas, haverá aumento do desempenho desde que os crescimentos dos outros dois termos não ultrapassem o ganho obtido na redução do primeiro termo.

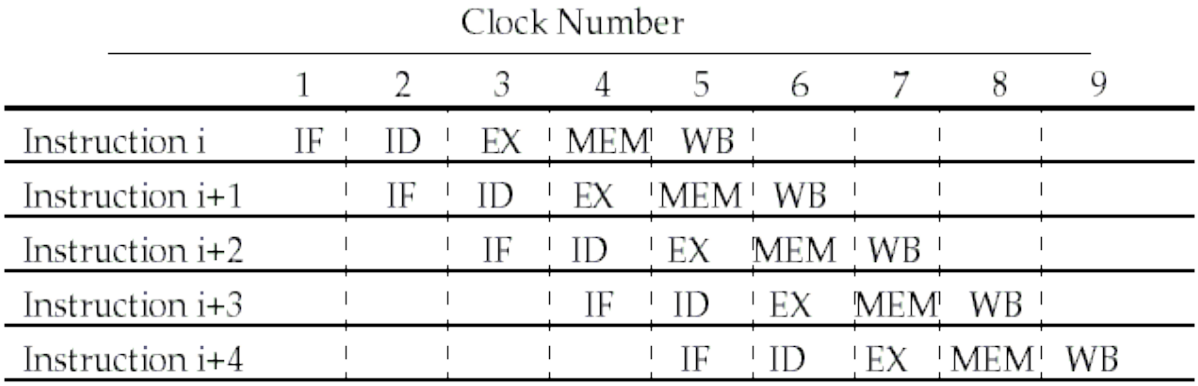
Existem muitas maneiras de reduzir o número de instruções do programa. Uma delas é que o conjunto de instruções pode conter instruções complexas para realizar mais trabalho de uma só vez. Por exemplo: em alguns casos, *CISC* ISA pode ter instruções complexas que podem resultar em programas com a metade do tamanho de programas com *RISC* ISA [33]. Essas instruções complexas, por outro lado, podem causar o aumento dos atrasos de propagação do circuito, devido ao aumento da complexidade do circuito. Esse aumento pode levar ao aumento do tempo de ciclo de máquina, ou seja, diminuição da frequência de operação.

Também existe a possibilidade de as instruções complexas necessitarem de maior número de ciclos de máquina, aumentando, assim, a média do número de ciclos de máquina por instrução.

O desejo de diminuir o número de ciclos de máquina por instrução inspirou muitas técnicas de arquitetura e microarquitetura. Uma das motivações para utilizar *RISC* ISA é reduzir a complexidade de cada instrução, para reduzir o número de ciclos de máquina necessário para processar cada instrução. Essa redução acarreta na elevação do número de instruções necessárias para realizar o programa, uma vez que as instruções, na arquitetura *RISC*, realizam menos trabalho. Contudo, com circuitos menos complexos e mais rápidos, a frequência de operação pode ser aumentada.

Outra técnica muito utilizada é o *pipeline*. Um processador usando *pipeline* pode executar várias instruções simultaneamente em cada estágio do *pipeline*. Tendo como base um processador sem *pipeline* e assumindo que cada instrução tem o mesmo número de ciclos necessários para execução, essa técnica pode, de fato, reduzir significativamente o número de ciclos por instrução, uma vez que várias instruções estariam sendo executadas concorrentemente, adicionando, somente, uma latência no começo para preencher o *pipeline* com a sequência de instruções que serão executadas, conforme ilustrado na Figura 19.

Figura 19 - Exemplo de *pipeline*



Contudo, a técnica *pipeline* tem seus problemas, relacionados a instruções que não têm número igual de ciclos de máquina. Para essas instruções, que necessitam de menor número de ciclos, será necessário acrescentar ciclos não utilizados nas instruções (*stall*), aumentando o número de ciclos necessários na execução do programa. A Figura 20 mostra um exemplo onde existem ciclos não utilizados quando a instrução anterior tem a necessidade de número maior de ciclos de máquina para ser realizada; neste caso, da instrução de multiplicação, seguida de uma instrução que poderia ser realizada em menor número de ciclos de máquina, instrução de soma, pode-se observar que durante a execução da instrução de multiplicação, a

instrução de soma aguarda o término dos vários estágios para continuar a execução da instrução.

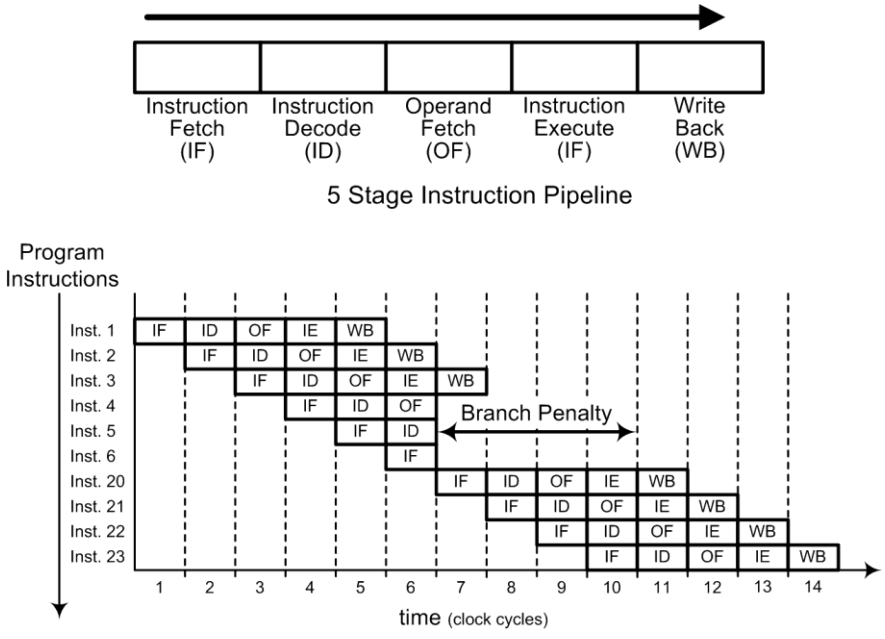
Figura 20 - Exemplo de ciclos não utilizados em *pipeline*

	Clock cycle number																
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Também devido à instrução de desvios no programa, como *jump* e *branch*, pode ser necessário o descarte dos dados presentes no *pipeline* (*branch penalty*) e uma nova sequência de instruções deve ser carregada, aumentando o número de acessos à memória e o número de ciclos necessários para execução do código. O número de acessos à memória é maior devido à leitura de instruções sequenciais que não serão executadas – lembrando que o acesso à memória pode ter alto consumo de potência associado.

A Figura 24 ilustra a condição de mudança de caminho do programa; exatamente na instrução 4, há mudança e essa mudança fez com que houvesse 3 acessos desnecessários à memória (inst. 4, inst. 5 e inst. 6). Também existem 3 ciclos adicionais (ciclos 7, 8 e 9) para carregar o *pipeline*, para que a execução se normalize novamente.

Figura 21 - Exemplo de ciclos não utilizados em *pipeline*



Para tentar resolver esse problema de descarte dos dados do *pipeline,* é possível adicionar técnicas de previsão de mudança de caminho (*branch prediction*) [34] [35]. Essa técnica aumenta o circuito necessário e, possivelmente, pode levar ao aumento dos atrasos de circuito, acarretando no aumento do tempo de ciclo de máquina (diminuição da frequência de relógio).

Muitas outras técnicas existem no que diz respeito a *pipeline* [36] – uma das técnicas mais utilizadas, pois existem duas vantagens em utilizá-la. Uma delas é diminuir o tempo de ciclo de máquina; como as instruções são realizadas em estágios com circuitos menos complexos, pode-se aumentar a frequência do relógio; assim, o tempo do ciclo de máquina pode ser diminuído drasticamente (aumento da frequência de relógio). A segunda vantagem é pela possibilidade de poder processar instruções em paralelo, decorrente de o número de ciclos de máquina necessários por instrução ser diminuído no mesmo fator da profundidade desse *pipeline*. Mas, para que isso funcione corretamente, existe grande esforço para equilibrar as vantagens em relação às desvantagens [37].

Existem outras técnicas, além do pipeline, como os processadores em paralelo na execução de instruções que aproveitam os paralelismos inerentes ao programa, podendo resultar em significativos aumentos de desempenho. Técnicas de processadores em paralelo normalmente são utilizadas em aplicações como processamento de imagens e algoritmos aritméticos, onde existe grande repetição e necessidade de grande poder de processamento. Nessas aplicações, devido ao paralelismo ser inerente da aplicação, os processadores com vários núcleos podem trazer grandes vantagens no desempenho final.

Como podemos ver, o aumento do desempenho não é uma tarefa fácil. Muitos fatores influenciam, sem contar que esse aumento de desempenho está correlacionado, em alguma dimensão, com o consumo de potência e que esse aumento de consumo de potência deve ser sempre levado em consideração.

3.2 Do ponto de vista de consumo de potência

Vale lembrar que queremos limitar nosso escopo aos aspectos que dizem respeito ao consumo de potência. Podemos dizer que a mudança de tecnologia para uma mais avançada pode, de fato, diminuir os atrasos de propagação, levando a um circuito mais rápido e de menor consumo de potência, devido à tecnologia; contudo, não iremos abordar essa ideia de diminuir o consumo de potência através da mudança de tecnologia, uma vez que queremos que o projeto seja portável para outras tecnologias.

Os resultados e as simulações apresentados nesse trabalho consideram células da tecnologia 0.25µm TSMC. O motivo da escolha desta tecnologia é permitir comparações mais reais com os trabalhos anteriormente realizados, nos quais essa tecnologia foi utilizada. Ressaltando que este trabalho foca mais na diminuição da potência dinâmica do circuito, através da escolha cuidadosa da microarquitetura que possa levar ao consumo de potência menor do processador.

Um ponto interessante a observar, relativo ao desempenho de processadores, é que, se conseguirmos fazer um processador que tenha desempenho melhor com o mesmo consumo de potência, significa termos um processador que, para o mesmo desempenho, terá consumo de potência menor.

3.3 Conceitos básicos de microcontroladores

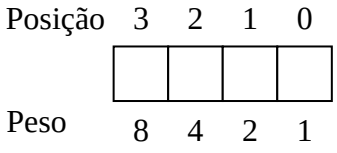
3.3.1 Representação de números

Aqui apresentamos como vamos tratar o número dentro do sistema. A representação será utilizando um sistema binário onde os dois possíveis estados seriam 0 e 1. A partir dessa definição básica, temos que considerar como iremos entender os números, utilizando células que só podem assumir esses dois estados. A composição dessas células será utilizada para representar internamente números que têm mais do que dois estados.

3.3.1.1 Inteiros binários sem sinal

A primeira representação numérica binária é o vetor de bits. Esta representação em vetor de bits ainda é normalmente usada em quase todos os estágios computacionais e também contém variáveis lógicas da maioria das implementações de linguagem existentes.

Figura 22 - Número binário de 4 digitos



Considere a Figura 22, os números acima dos quadrados representam as posições dos bits. Os números abaixo dos quadrados representam os pesos que cada posição tem relativamente à posição unitária do bit 0. Como, no exemplo da Figura 22, esta notação tem 4 bits que pode representar 16 estados, sendo que cada quadrado representa um bit do número

binário. Com esta notação, podemos adicionar mais bits, de acordo com a necessidade do resultado esperado dos nossos cálculos.

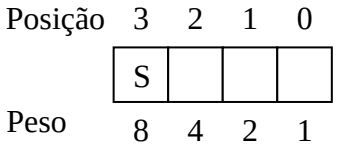
Na Tabela 1, é apresentado para cada possível estado de um número binário de 4 bits – o valor correspondente para representação de números sem sinal.

Em uma arquitetura que será implementada em circuito, devemos definir previamente o maior número representável. Essa necessidade de definir o maior número representável dá origem ao conceito de número de bits dos circuitos digitais, isto é, processadores de n bits, circuitos aritméticos de n bits, entre outros. Quanto maior o número a ser representado, maior será a complexidade do circuito resultante dessa arquitetura.

3.3.1.2 Inteiros binários com sinal e complemento de dois

Para representar números negativos, utilizamos uma das posições para a representação de sinal. Este, então, pode representar positivo ou negativo. Normalmente, o bit escolhido é o mais à esquerda, Figura 23.

Figura 23 - Número binário de 4 dígitos com sinal



O restante dos bits será tratado como um número sem sinal e chamado de magnitude (Tabela 1).

Arranjando os números no sistema de números de complemento de dois evita-se o aumento de complexidade do circuito, quando se utilizam números com sinais em operações aritméticas. Nessa representação, todos os bits podem utilizar a mesma regra aritmética. Para isso, basta que os números estejam centrados no zero, estando os positivos acima e os negativos abaixo, como podemos observar na Tabela 1.

A próxima equação demonstra como poderíamos gerar o número negativo correspondente a partir do número positivo. Observando-se a Tabela 1, chegamos à equação a seguir para determinar –A:

$$Negative(A) = NOT(A) + 1 \quad (10)$$

Onde:

NOT(A) é a operação que inverte todos os bits de A.

As regras aritméticas serão demonstradas quando discutirmos somadores e subtradores.

Tabela 1 - Representação de números binários complemento de dois

Número sem sinal	Número com sinal	Binário
7	7	0111
6	6	0110
5	5	0101
4	4	0100
3	3	0011
2	2	0010
1	1	0001
0	0	0000
15	-1	1111
14	-2	1110
13	-3	1101
12	-4	1100
11	-5	1011
10	-6	1010
9	-7	1001
8	-8	1000

3.3.2 Soma de números binários

Considerando a soma de números binários, analisemos os somadores de 1 bit. Nele, temos as entradas A, B e *carry in* e as saídas S e *carry out*. A função de soma, para números positivos, está representada na Tabela 2 e a função soma, para números negativos, está representada na Tabela 3.

Tabela 2 - Tabela verdade da função soma (números positivos)

entrada			saída	
carry in	A	B	S	**carry out**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1

Tabela 3 - Tabela verdade da função soma (números negativos)

entrada			saída	
carry in	A	B	S	**carry out**
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Para ilustrar a operação de soma, apresentamos alguns exemplos:

	Decimal	Binário 4bits
Carry in		0
A	+3	0011
B	+2	0010
S	+5	0101

Cada bit de coluna é somado da direita para esquerda, obedecendo a Tabela 2 da verdade, como se fosse uma soma normal de decimais. O *carry out* de cada coluna é o *carry in* da próxima coluna. Para números negativos, é utilizada a Tabela 3, seguindo o mesmo princípio dos números positivos.

	Decimal	Binário 4 bits
Carry in		0
A	-3	1101
B	-2	1110
S	-5	1 1011

Podemos observar que, usando a notação em complemento de dois, podemos realizar as somas de todos os bits obedecendo-se a mesma regra aritmética. Agora, mudando para a soma de números positivos com negativos:

	Decimal	Binário 4bits
Carry in		0
A	-3	1101
B	+2	0010
S	-1	1111

Novamente aplicando a Tabela 2 e a Tabela 3, obtemos o resultado esperado. Nota-se que temos que desconsiderar o *carry out* do bit mais à esquerda, isto é, o bit mais significativo. Nessa nova representação, existe agora a necessidade de entendermos a possibilidade de a operação aritmética gerar um número que não está especificado na nossa representação de 4 bits.

No caso de *overflow*, começaremos a análise com a soma de dois números de sinais diferentes. Nesse caso, podemos concluir que nunca daria *overflow*, porque o resultado estaria mais perto do zero. Contudo, quando os números a serem somados têm sinais iguais, existe a possibilidade de dar o *overflow*.

Inicialmente, analisaremos os números de mesmo sinal que não dão *overflow*. Os números no exemplo foram selecionados para que o valor resultante fique próximo ao número máximo representável. Todas as somas inferiores terão o mesmo comportamento.

	Decimal	Binário 4 bits	Decimal	Binário 4 bits
Carry in		0		0
A	+3	0011	-3	1101
B	+4	0100	-4	1100
S	+7	0111	-7	1001
Carry out		0		1
Carry in		0		1

Analisando exemplos de número que dão *overflow* na soma, concluímos que o *overflow* pode ser detectado observando-se o *carry in* e o *carry out* do bit mais significativo, isto é, da coluna mais à esquerda.

	Decimal	Binário 4 bits	Decimal	Binário 4 bits
Carry in		0		0
A	+7	0111	-7	1001
B	+2	0010	-2	1110
S	+9	1001	-9	0111
Carry out		0		1
Carry in		1		0

O *overflow* acontece quando um dos *carry* é 0 e o outro 1; nesse momento, podemos equacionar o *overflow* como:

$$Overflow = Carry\ In_{bit\ mais\ significativo}\ \boldsymbol{XOR}\ Carry\ out_{\ bit\ mais\ significativo} \qquad (11)$$

Esta equação pode ser construída combinacionamente utilizando-se uma porta XOR. O resultado estará presente no mesmo instante que o resultado da soma for produzido. Devido a esta característica e também pelo fato do bit de sinal poder ser tratado com a mesma aritmética, o sistema de complemento de dois torna-se o mais utilizado.

Importante notar que o mesmo circuito pode ser utilizado para se fazer a soma de números sem sinal.

3.3.2.1 Operações básicas (NOT, AND, OR, XOR)

Algumas operações básicas são importantes para os processadores. Essas operações aparecem em todos os ISA existentes e compõem instruções básicas de todos os processadores. A Tabela 5 e a

Tabela 4 ilustram a tabela verdade de algumas funções básicas que serão utilizadas no decorrer do desenvolvimento.

Tabela 4 - Tabela verdade das funções AND, OR e XOR

Entrada		Saída XOR	Saída OR	Saída AND
A	B	S	S	S
0	0	0	0	0
0	1	1	1	0
1	0	1	1	0
1	1	0	1	1

Tabela 5 - Tabela verdade da função NOT

Entrada	Saída
A	S
0	1
1	0

3.3.3 Estruturas aritméticas

3.3.3.1 Somadores

Começaremos uma análise simples e evoluiremos até uma estrutura mais complexa de somadores. A tabela verdade do meio somador é mostrada na Tabela 6.

Tabela 6 - Tabela verdade do meio somador

Entrada		Saída	
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Este elemento não inclui o *carry in* previamente descrito; pela rápida análise da tabela verdade, podemos identificar que a operação de soma pode ser descrita:

$$S = A\ XOR\ B$$

A saída de *carry out* (C_{out}) pode, também, da mesma forma, ser identificada como uma simples função. Observando atentamente a tabela verdade do meio somador podemos identificar que a saída C_{out} é a função AND.

$$C_{out} = A\ AND\ B \tag{12}$$

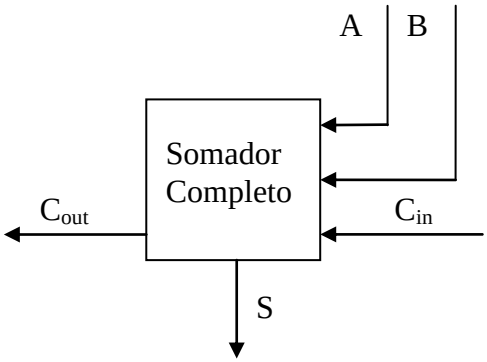
A tabela verdade do meio somador nada mais é do que um somador aplicando-se no *carry in* (C_{in}) com o valor 0. Podemos evoluir para um somador completo adicionando-se o elemento *carry in* – como se pode observar pelas equações a seguir.

$$S = A\ XOR\ B\ XOR\ C_{in} \tag{13}$$

$$C_{out} = (A\ AND\ B)\ OR\ (A\ AND\ C_{in}) OR\ (B\ AND\ C_{in}) \tag{14}$$

A implementação dessas duas equações dá o resultado final ao somador completo de 1 bit. A Figura 24 mostra o diagrama de bloco do somador completo de 1 bit.

Figura 24 - Diagrama de somador completo



Estendendo a análise para somadores com maior número de bits, começaremos nossa análise com a determinação do C_{out} dessa estrutura de 4 bits (vide Figura 25). Para efeito de simplificação, iremos utilizar o sinal de ponto como a operação AND, o sinal + como a operação de OR e o símbolo ^ como a operação XOR.

$$C_{out1} = A_0\,.B_0 + A_0\,.\,C_{in0} + B_0.C_{in0} \tag{15}$$

$$C_{out2} = A_1\,.B_1 + A_1\,.\,C_{in1} + B_1.C_{in1} \tag{16}$$

$$C_{out3} = A_2\,.B_2 + A_2\,.\,C_{in2} + B_2.C_{in2} \tag{17}$$

$$C_{out4} = A_3\,.B_3 + A_3\,.\,C_{in3} + B_3.C_{in3}$$

$$C_{out4} = A_3\,.B_3 + (A_3 + B_3).C_{in3} \tag{18}$$

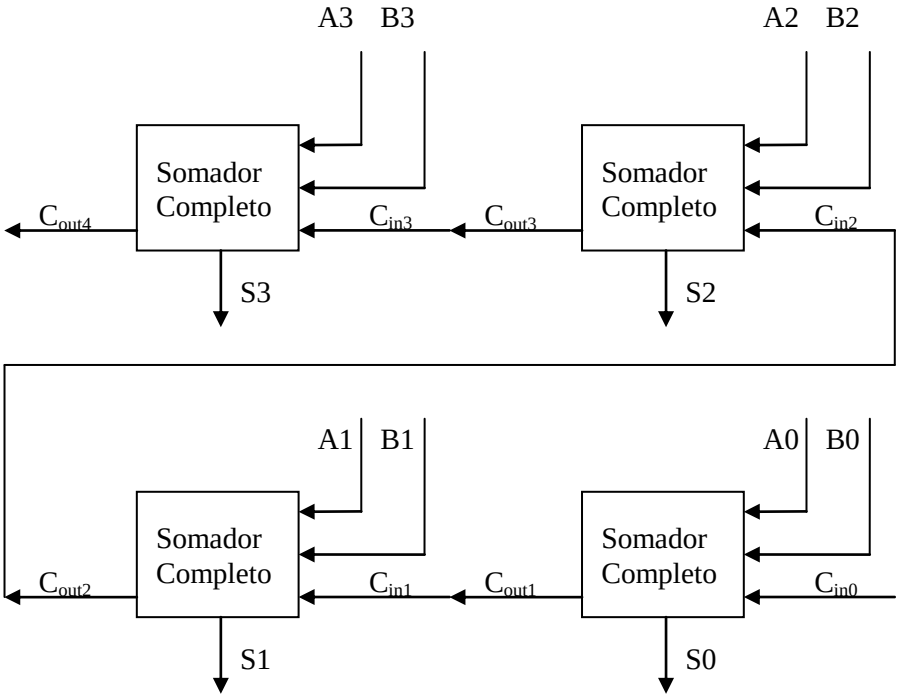
$$C_{out4} = A_3\,.B_3 + (A_3 + B_3).[(A_2.B_2) + (A_2 + B_2).C_{in2}]$$

$$C_{out4} = A_3 \,.\, B_3 + (A_3 + B_3).[(A_2.B_2) + (A_2 + B_2).\{(A_1.B_1) + (A_1 + B_1).\langle(A_0.B_0) + (A_0 + B_0).C_{in0}\rangle\}]$$

Observando as equações correspondentes, podemos observar que, na implementação, utilizando-se portas lógicas básicas, leva-se a atraso de propagação muito grande. Este é o grande motivo das limitações existentes na implementação de sistemas aritméticos, principalmente quando se implementa utilizando a composição de somadores completos de 1 bit como estrutura básica.

Para contornar esse problema, podemos descrever a tabela verdade dessas nove entradas (as 4 entradas A, mais as 4 entradas B e o C_{in}), gerando uma tabela verdade de 2^9; ou seja, 512 linhas. Essa tabela pode ser desmembrada para cada saída Sn e fazer a minimização booleana para cada equação de saída.

Figura 25 - Diagrama de somador completo de 4 bits



Para simplificar essa abordagem, iremos descrever rapidamente uma possível solução para esse problema (existem muitas outras abordagens [38] que não serão descritas nesse trabalho). Iremos adicionar 2 novos termos ao somador de 1 bit apresentado.

$$G_i = A_i \,.\, B_i \qquad \textit{Carry} \text{ gerado} \tag{19}$$

$$P_i = A_i + B_i \qquad \textit{Carry} \text{ propagado} \tag{20}$$

Esses 2 termos serão gerados em cada somador de 1 bit; contudo, os sinais serão utilizados não pelo próximo somador e sim no final do processo de soma.

O termo gerado G_i representa o enésimo termo do somador, indicando se gerou ou não o *carry*. De forma similar, o termo propagado P_i representa se o próximo estágio do somador necessita que o *carry* seja propagado caso o *carry in* do estágio seja 1.

Voltando um pouco às equações apresentadas, podemos observar que esses termos já aparecem na equação do *carry out* do somador completo de 1 bit. A próxima equação mostra esses termos identificados.

$$\begin{gathered} C_{out\,(i+1)} = A_i \,.\, B_i + \, A_i \,.\, C_{in(i)} + \, B_i . C_{in(i)} \\ C_{out\,(i+1)} = A_i \,.\, B_i + \, (A_i \, + \, B_i) . C_{in(i)} \\ C_{out\,(i+1)} = G_i \, + \, P_i \,.\, C_{in(i)} \end{gathered} \tag{21}$$

Para ilustrar um pouco mais, as equações a seguir mostram o *carry out* dos dois primeiros estágios.

$$C_{out\,(1)} = G_0 \, + \, P_0 \,.\, C_{in(0)} \tag{22}$$

$$\begin{gathered} C_{out\,(2)} = G_1 \, + \, P_1 . (G_0 \, + \, P_0 \,.\, C_{in(0)}) \\ C_{out\,(2)} = G_1 \, + \, P_1 . G_0 + P_1 . P_0 . C_{in(0)} \end{gathered} \tag{23}$$

Podemos observar que, se o somador fosse somente de dois estágios, o tempo de propagação para essa equação seria do atraso de duas portas lógicas. Também interessante notar que a lógica ainda é simples o suficiente para se implementar.

Seguindo com essa implementação, temos:

$$\begin{gathered} C_{out\,(3)} = G_2 \, + \, P_2 . \left(G_1 \, + \, P_1 . G_0 + P_1 . P_0 . C_{in(0)}\right) \\ C_{out\,(3)} = G_2 \, + \, P_2 . G_1 + P_2 . P_1 . G_0 + P_2 . P_1 . P_0 . C_{in(0)} \end{gathered} \tag{24}$$

$$C_{out\,(4)} = G_3 + P_3 . G_2 \, + \, P_3 . P_2 . G_1 + P_3 . P_2 . P_1 . G_0 + P_3 . P_2 . P_1 . P_0 . C_{in(0)} \tag{25}$$

O tempo de propagação necessário para o $C_{out\,(3)}$ não aumentou em relação ao estágio anterior. Da mesma forma, o tempo de propagação das portas lógicas, para implementação em

circuito do $C_{out\ (4)}$, também é o mesmo (dois níveis de lógica). Dessa forma, podemos fazer um somador de n bits de uma forma que os atrasos de propagação dos sinais de *carry* não se tornem proibitivos na implementação em circuito.

A equação de soma de n bits pode ser representada pela fórmula a seguir.

$$S_i = A_i \text{ ^ } B_i \text{^} C_{in(i)} \tag{26}$$

Para um somador de 4 bits, temos:

$$S_0 = A_0 \text{ ^ } B_0 \text{^} C_{in(0)} \tag{27}$$

$$S_1 = A_1 \text{ ^ } B_1 \text{^} \left(G_0 \ + \ P_0 \ . C_{in(0)}\right) \tag{28}$$

$$S_2 = A_1 \text{ ^ } B_1 \text{^} \left(G_1 \ + \ P_1 . G_0 + P_1 . P_0 . C_{in(0)}\right) \tag{29}$$

$$S_3 = A_2 \text{ ^ } B_2 \text{^} \left(G_2 \ + \ P_2 . G_1 + P_2 . P_1 . G_0 + P_2 . P_1 . P_0 . C_{in(0)}\right) \tag{30}$$

Deve-se notar que, nestas equações, os atrasos das portas lógicas necessárias para a implementação em circuito foram minimizados em relação à utilização das primeiras equações apresentadas com o componente $C_{in(i)}$. Mesmo utilizando esse algoritmo, com o crescimento do número de bits do somador, a complexidade da geração de $C_{in(i)}$ cresce exponencialmente até um ponto que será inviável a construção do somador. Assim, é interessante definir quantidade de bits razoável para implementação, utilizando-se G e P, e depois realizar o cascateamento desses somadores, como proposto inicialmente, podendo, assim, obter somadores com número de bits maiores.

Se chamarmos J o número de bits de estruturas que utilizam esse algoritmo G, P e K, o número de estruturas necessárias para o somador final, podemos ter economia de tempo de atraso de uma relação J/K do somador implementado.

3.3.3.2 Subtradores

Os subtradores binários podem ser derivados dos somadores anteriormente apresentados, se considerarmos que um dos operandos esteja em complemento de dois, como apresentado quando abordamos a representação de números binários com complemento de dois.

$$\begin{gathered} R = A - B \\ R = A + (-B) \end{gathered} \tag{31}$$

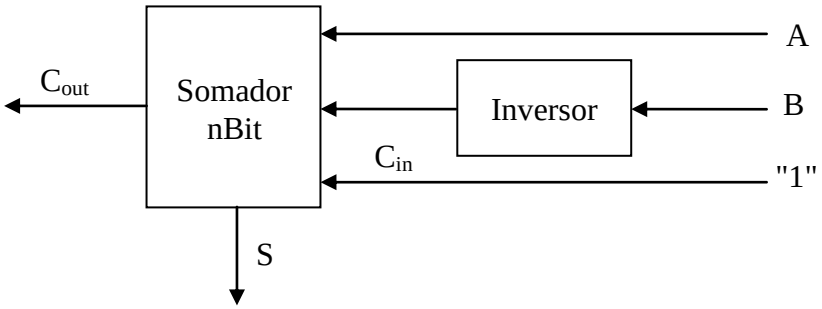
$$-B = inv(B) + 1 \quad (32)$$

$$R = A + inv(B) + 1 \quad (33)$$

Para se realizar isso, basta inverter bit a bit um dos operandos (B), isto é, implementando o complemento de um. Para se alcançar o complemento de dois, podemos utilizar a entrada de *carry in* (C_{in}), a qual servirá para somar 1; assim, o operando (B) estaria em complemento de dois.

Para ilustrar um subtrador, a Figura 26 mostra como estariam ligados para que uma estrutura de somador seja utilizada como subtrador.

Figura 26 - Diagrama de somador convertido para subtrador



3.3.3.3 Multiplicadores

A Tabela 7 mostra a tabela verdade da função multiplicadora.

Para estruturas de multiplicadores, primeiramente poderíamos imaginar que se pode fazer n operações de adição. O número de operações necessárias para fazer uma determinada multiplicação seria somar o operando A por número de vezes representado pelo operando B.

Tabela 7 - Tabela verdade da função multiplicação

Saída	Entrada	
Prod	A	B
0	0	0
0	0	1
0	1	0
1	1	1

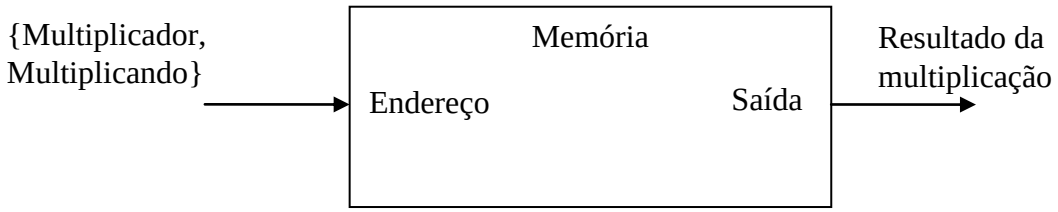
Considerando um multiplicador de 4 bits, poderíamos, no limite, ter 16 operações de soma. Se considerarmos que cada operação seja realizada a um ciclo de relógio, então seriam necessários 16 ciclos de relógio. Se considerarmos uma operação de multiplicação de 8 bits, o

número de ciclos de relógio para as sucessivas somas seria de $2^8 = 256$. Podemos perceber que o número de operações cresce exponencialmente, o que torna essa abordagem proibitiva para multiplicadores com operandos de número de bits maiores.

Outra abordagem seria o armazenamento do resultado da multiplicação em uma tabela (memória), assim o endereço dessa tabela seria a composição do multiplicador e do multiplicando e o resultado dessa multiplicação estaria armazenado nesse endereço da tabela. Essa abordagem está ilustrada na Figura 27. Esse tipo de abordagem é utilizado não somente para multiplicação, como também para funções aritméticas mais complexas (seno, cosseno, funções não lineares de criptografia).

Uma das vantagens de se usar tabela é que a operação estaria imediatamente pronta após a composição do endereço, não dependendo de nenhum outro circuito; isto é, o tempo de atraso é somente o tempo de acesso à memória.

Figura 27 - Operação aritmética por tabela



Contudo, para números que necessitam de representação com quantidade de bits elevados, também existe a necessidade de essa tabela crescer de forma exponencial, acarretando, cada vez, a necessidade de memória de maior capacidade.

Observando a tabela verdade do somador de um bit, apresentada na Tabela 7, verifica-se que a porta lógica AND tem a mesma tabela verdade (C_{out}). Esta característica permite expressar a operação de multiplicação como uma série de operações de função AND, deslocamentos e soma. Essa abordagem pode ser criada com uma sequência de passos, como também é possível a criação em circuitos combinacionais.

A criação de multiplicadores de n bits aparentemente necessita de muitas portas AND, de muitas entradas e muitas portas OR. Uma particular abordagem seria fazer o algoritmo de multiplicação muito parecido como se faz manualmente. Essa implementação pode ser vista na Figura 28. Cada coluna seria somada com somadores completos de 1 bit, criando o produto final P(i). O *carry out* de cada somador seria adicionado no próximo estágio: essa característica determina o nome desse somador "*carry save adder*".

Para esses multiplicadores, o maior tempo de propagação, no caso de totalmente combinacional, é o caminho que cruza o somador através dos *carry out* – muito parecido com os atrasos encontrados nos somadores completos que cascateiam os *carries*. Importante notar que, com o aumento do número de bits do multiplicador, esse tempo também cresce; contudo, esse crescimento é menor do que o dos somadores cascateados (*ripple carry adder*).

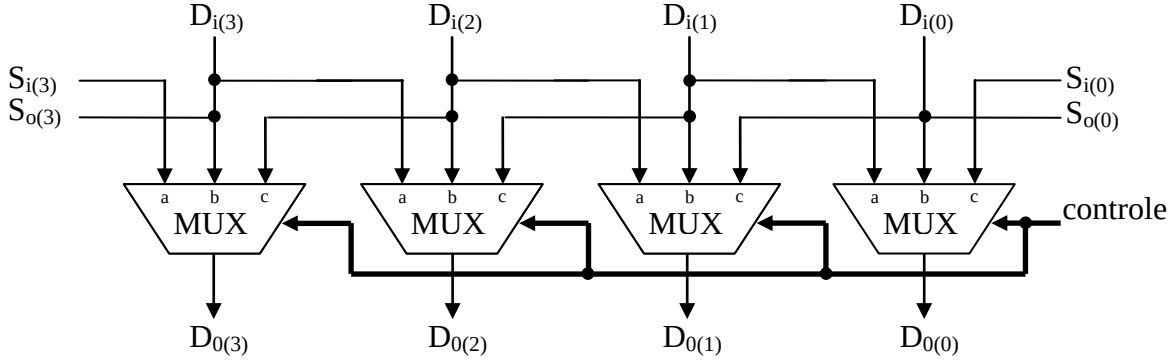
Figura 28 - Ilustração da operação de multiplicação

Multiplicando A				A_3	A_2	A_1	A_0
Multiplicando B				B_3	B_2	B_1	B_0
				$A_3.B_0$	$A_2.B_0$	$A_1.B_0$	$A_0.B_0$
			$A_3.B_1$	$A_2.B_1$	$A_1.B_1$	$A_0.B_1$	
		$A_3.B_2$	$A_2.B_2$	$A_1.B_2$	$A_0.B_2$		
	$A_3.B_3$	$A_2.B_3$	$A_1.B_3$	$A_0.B_3$			
P_7	P_7	P_5	P_4	P_3	P_2	P_1	P_0

3.3.3.4 Deslocadores

Nesta parte, iremos mostrar estruturas de deslocamento "*shifters*". Basicamente, deslocadores são estruturas simples que deslocam os bits para direita ou esquerda de 1 bit. O sinal de controle é composto por dois bits, que seleciona uma das entradas *a*, b ou *c* de cada multiplexador. Dependendo do valor selecionado para o controle, podemos deslocar os bits de entrada para direita ou esquerda.

Figura 29 - Diagrama de deslocador de bits



Se o controle selecionar a entrada "a", teremos um deslocamento para direita, onde $D_{o(3)}=S_{i(3)}$, $D_{o(2)}=D_{i(3)}$, $D_{o(1)}=D_{i(2)}$, $D_{o(0)}=D_{i(1)}$ e $S_{o(0)}=D_{i(0)}$. No caso do controle selecionar as

entradas "b", teremos as saídas $D_{o(i)}$ iguais à entrada $D_{i(i)}$, caracterizando o modo transparente onde os dados de entrada passam diretamente para saída. Por fim, se o controle selecionar as entradas "c" dos multiplexadores, teremos o deslocamento para esquerda da mesma forma que ocorre o deslocamento para direita.

O comprimento do deslocador pode ser aumentado de acordo com o número de multiplexadores na estrutura e conectando-se junto o controle de cada multiplexador. Se quisermos criar estruturas que deslocam dois bits, basta reconectar os multiplexadores, de forma que reflita o deslocamento desejado.

Nessa linha de raciocínio, é possível criar um tipo de deslocador que possa selecionar o número de deslocamentos desejados (*barrel shifter*). Existem muitas operações aritméticas que necessitam de deslocamento maior do que 1 bit; se essas operações puderem ser feitas somente em um passo, isso tornaria a realização muito mais rápida.

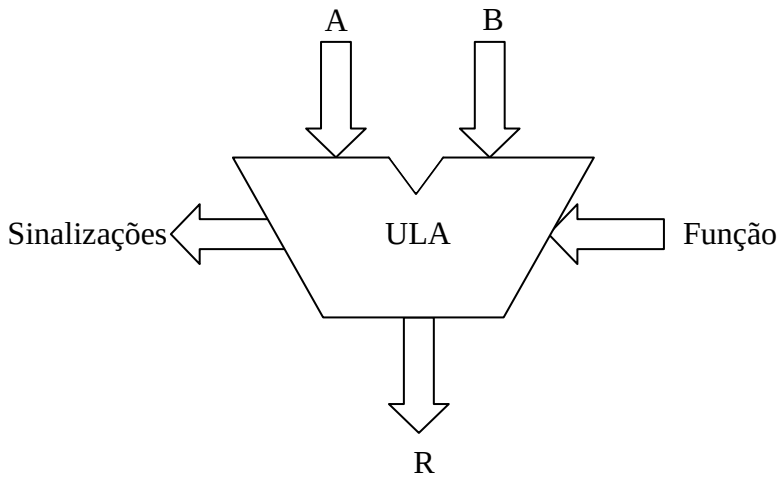
A construção desses deslocadores é o cascateamento de vários multiplexadores; a profundidade desses multiplexadores define o número de bits de deslocamentos possível por essa estrutura. Algumas operações que utilizam deslocamento são normalizações de números, alinhamento do ponto em números binários e identificações de bit.

3.4 Macroblocos de processadores

3.4.1 Unidade Lógica Aritmética - ULA

Um bloco fundamental para criação de processadores é a unidade lógica aritmética.

Figura 30 - Símbolo de Unidade Lógica Aritmética (ULA)



As entradas A e B serão chamadas de operadores, a função indica a operação que a ULA realizará e R representa o resultado da operação aritmética. Sinalizações são

informações que são geradas durante a realização da operação. Importante ressaltar, nesse momento, que a ULA é totalmente implementada em circuito combinacional e não necessita de elementos de memória, tipo registradores, para a realização da operação aritmética.

Para esse trabalho, vamos criar uma ULA com as operações listadas na Tabela 8:

Tabela 8 - Funções implementadas na ULA

Função	
0	Complemento de um
1	AND bit a bit
2	Or bit a bit
3	XOR bit a bit
4	Soma de complemento de dois
5	Subtração de complemento de dois (A-B)
6	Subtração de complemento de dois (B-A)

As sinalizações que iremos implementar estão ilustradas na Tabela 9.

Tabela 9 - Sinais implementado na ULA

V	Overflow de complemento de dois
H	*Carry out* do bit 4
N	Negativo
Z	Zero
C	*Carry out*

A representação numérica considerada pela ULA, nesse trabalho, será a de complemento de dois; existem muitas outras representações, mas, atualmente, pela simplicidade, a maioria das ULA tem essa representação. É possível criar uma ULA tão complexa quanto se queira, com a decorrência de que o circuito gerado será maior, com, consequentemente, consumo de potência maior e área maior.

A ULA pode ser entendida como o conjunto de estruturas e o caminho dos dados que realizam as funções definidas na Tabela 8. Assim, para nossa ULA, necessitaremos de portas AND, OR, XOR, somador e multiplexadores para realizar as interconexões dos dados e seleções para essas estruturas.

Vamos, inicialmente, analisar a operação de adição e subtração, observando a estrutura de subtração, apresentada anteriormente na Figura 26, e estendendo para uma estrutura que faça uma operação de soma ou subtração dependendo de um sinal de entrada. Para isso, criaremos o sinal SubAdd, que representa se a função desejada é uma subtração (1) ou uma adição (0). Facilmente, podemos usar o sinal SubAdd diretamente ligado ao carry in para poder representar a soma de 1 na transformação de um dos operadores em número negativo.

Olhando para o operador B, temos um inversor, no caso de subtração, e, no caso de soma, o sinal B passa diretamente ao somador.

Analisando a Tabela 10, onde o operando B' considera a possibilidade de soma ou subtração, chegamos à função XOR, que poderia ser colocada entre o sinal B e em conjunto com o sinal SubAdd e gerar o segundo operando na entrada do somador. Lembrando que existirá uma célula de XOR para cada bit do operador B.

Tabela 10 - Sinal B considerando soma ou subtração

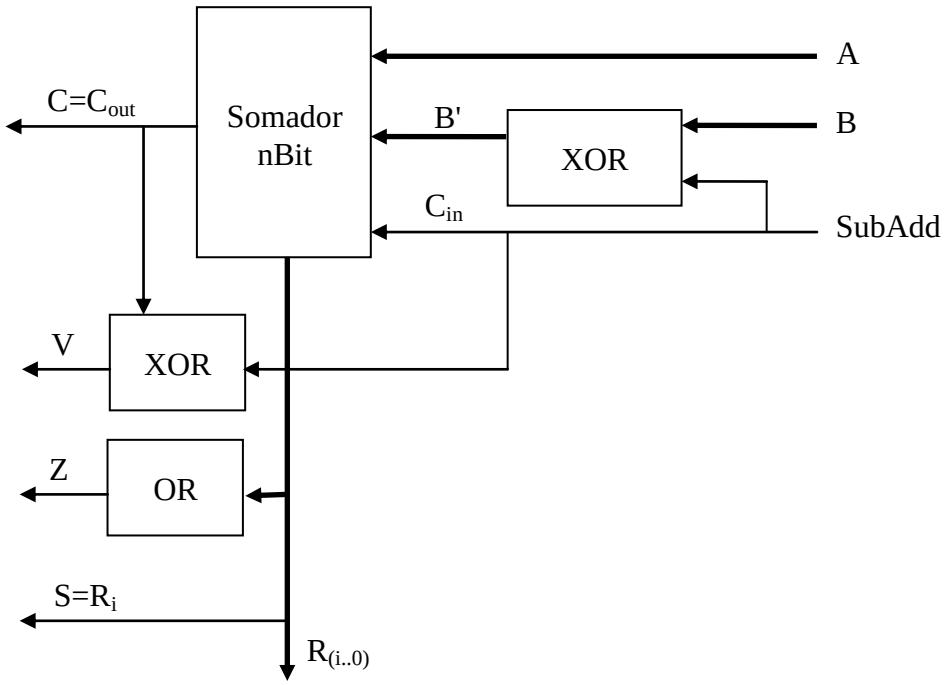
B'	SubADD	B
0	0 (soma)	0
1	0 (soma)	1
1	1 (sub)	0
0	1(sub)	1

Podemos derivar a criação de B' através da função:

$$B' = B \text{ ^ } SubAdd \tag{34}$$

A Figura 31, mostra uma estrutura simples, que pode fazer a soma ou a subtração de operandos. Essa estrutura acaba economizando circuito, uma vez que o somador pode ser utilizado para realizar as operações de soma como também de subtração.

Figura 31 - Diagrama de somador/subtrador convertido



Para se implementar as sinalizações, precisamos combinar, de alguma forma, os operandos A e B e o resultado R da operação aritmética, para que gere a sinalização correta.

A sinalização Z significa que todos os bits resultantes da operação aritmética é 0. Podemos gerar esse sinal facilmente, com a utilização de portas OR e um inversor, como podemos observar na equação a seguir.

$$Z = \text{NOT}\ (R_i + R_{i-1} + \ldots R_0) \tag{35}$$

A sinalização S significa o sinal do número binário resultante da operação realizada na ULA. Considerando que estamos trabalhando com sinais em complemento de dois, a sinalização pode facilmente ser entendida como o bit mais significativo do resultado da operação:

$$S = R_i \tag{36}$$

A sinalização V (Overflow) é uma sinalização de operação aritmética e pode ser obtida através da combinação dos sinais de *carry* out e *carry* in. Esta equação já foi discutida quando abordados os números binários de complemento de dois.

$$V = C_{out}\ (\text{do bit mais significativo})\ \hat{}\ C_{in}\ (\text{do bit mais significativo}) \tag{37}$$

A sinalização C é o próprio *carry out* do somador, onde:

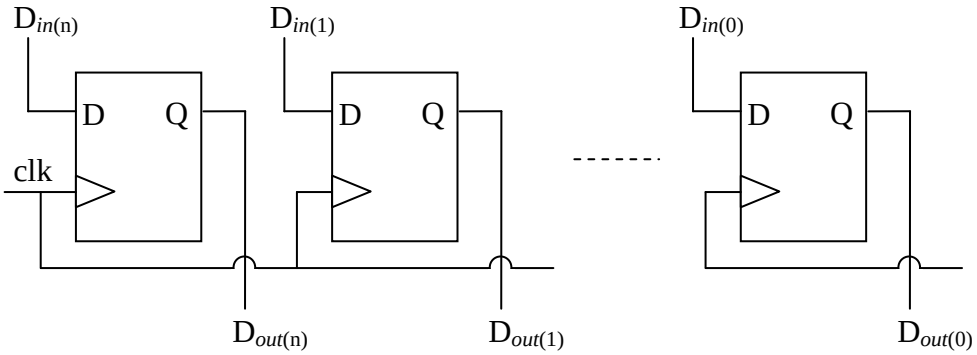
$$C = C_{out}\ (\text{do bit mais significativo do somador}) \tag{38}$$

Todas as sinalizações aqui descritas podem ser geradas totalmente combinatorialmente; assim, o resultado estará presente no mesmo momento que a operação aritmética estiver pronta.

3.4.2 Arquivo de registros

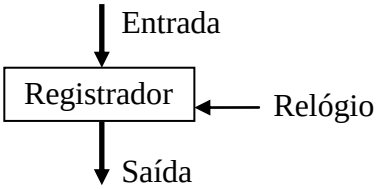
Registrador é um conjunto de flip-flops ligados em paralelo. Os relógios são ligados todos juntos e os dados de entradas dos flip-flops são a entrada, e a saída é o valor registrado nos flip-flops. A cada ciclo de relógio, os dados são atualizados com os valores da entrada. A Figura 32 mostra a configuração das ligações dos flip-flops para se obter o registrador.

Figura 32 - Registrador n+1 bits



AFigura 33 ilustra o símbolo do bloco de registrador utilizado nesse trabalho.

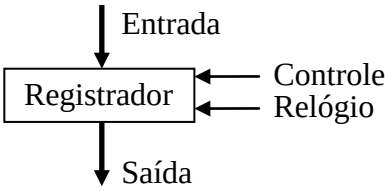
Figura 33 - Diagrama de registrador básico



Uma configuração mais interessante é dos registradores nos quais exista controle de carregamento dos dados de entrada.

Se adicionarmos um circuito de controle, podemos ter controle quando os dados forem armazenados nos *flip-flops* – o diagrama desse registrador está ilustrado na Figura 34.

Figura 34 - Diagrama de registrador com controle de carregamento



Expandindo um pouco mais o conceito de registradores, podemos, através da combinação de registradores, criar uma matriz de registros nas quais se pode escrever em um registro enquanto se faz a leitura de outro registro (Figura 35).

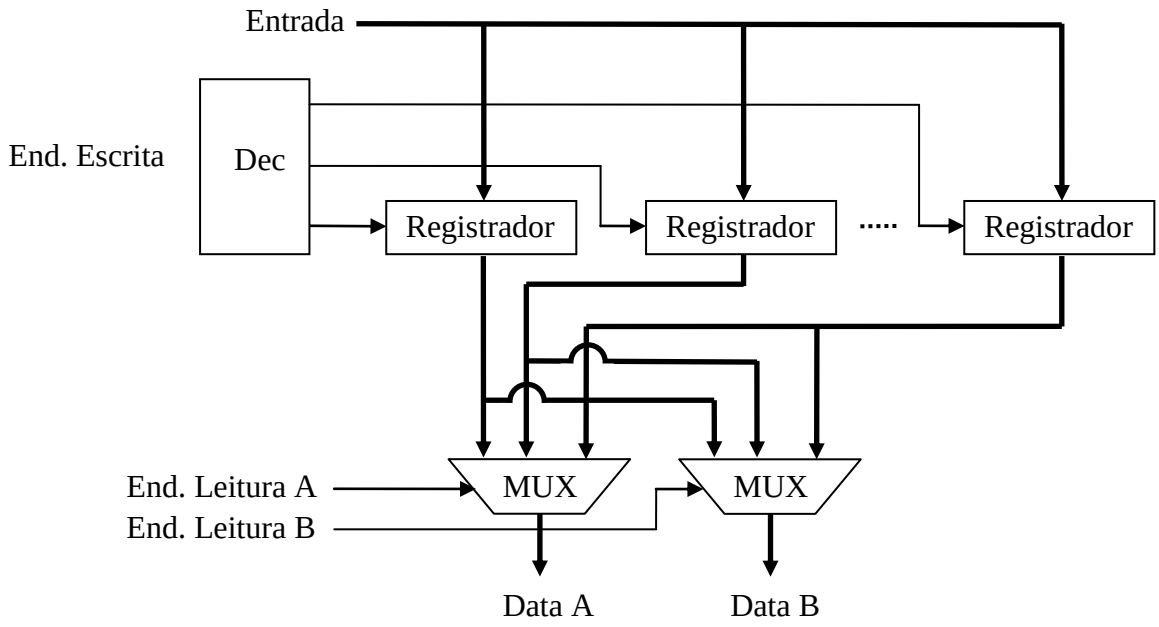
Com a adição de um segundo multiplexador, podemos ter uma estrutura na qual podemos escrever a partir do endereço de escrita em qualquer registrador e, ao mesmo tempo, podemos ter a leitura de dois registradores – tudo ao mesmo tempo. Assim, criamos um

registro *multi-port* – estrutura que, dentro de processadores, tem o nome de arquivo de registro (*register file*).

Interessante notar que, nessa configuração, a leitura não está relacionada com o relógio. Somente os dados de escrita são capturados no registrador selecionado pelo endereço de escrita na borda do relógio.

Esse tipo de estrutura, em conjunto com a unidade lógica aritmética, se torna uma parte importante da microarquitetura do processador, podendo ter mais portas de leitura e número de registradores internos, dependendo somente da arquitetura utilizada.

Figura 35 - Matriz de registradores

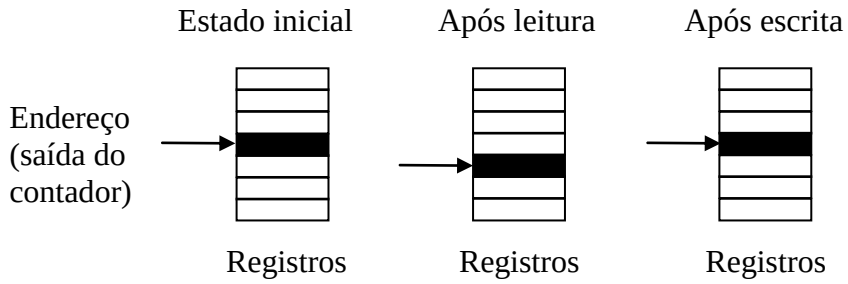


3.4.2.1 Stack pointer

O *stack pointer* é muito parecido com uma estrutura de arquivo de registro, somente que o endereço de escrita é controlado por um contador que pode ser incrementado ou decrementado. O valor desse contador é decodificado e gera o sinal que habilita a escrita para o banco de registradores. O mesmo valor do contador é também utilizado como endereço de leitura, assim, na saída dessa estrutura, temos os dados do registro apontado pelo contador. Na entrada do contador, existe um sinal que indica a leitura ou a escrita no banco de registro e um sinal que a habilita a leitura ou a escrita dessa estrutura. No caso de o sinal de habilitar não estar ativo, pode se considerar que não existe a inclusão de dado novo nem a retirada de um dado.

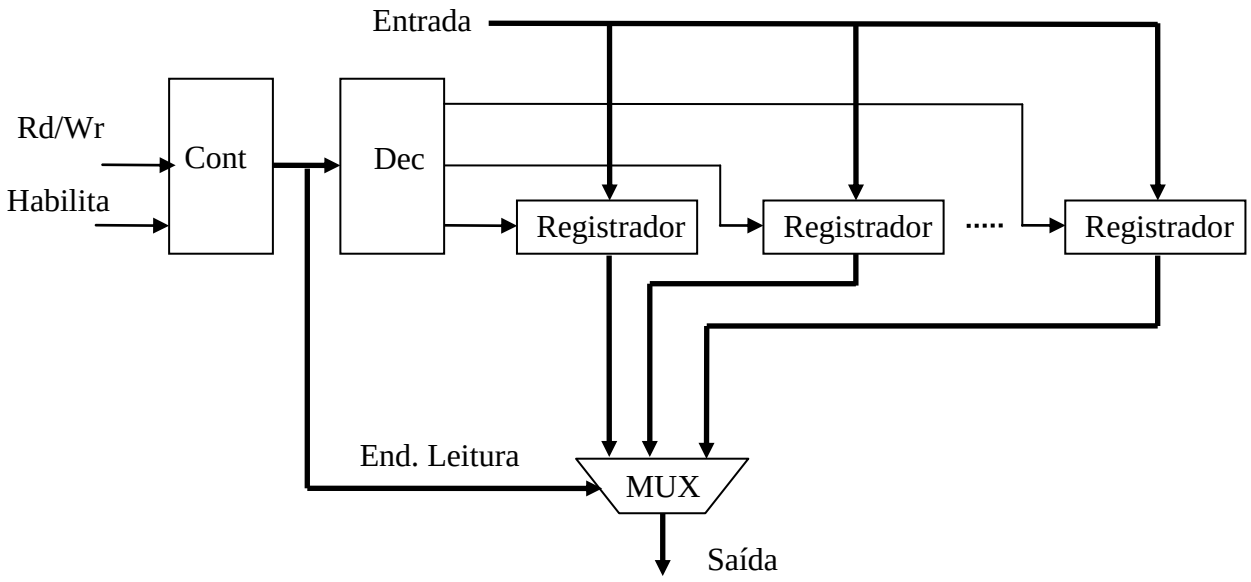
O dado de saída permanece o mesmo, apontando para o último dado escrito nos registros. Por exemplo, se quisermos fazer uma leitura – e considerando que o contador decrescesse a cada leitura –, o registro apontado após uma leitura seria o registro de endereço menor; o mesmo acontece com a escrita, fazendo que, após a escrita, o endereço apontado pelo contador seja incrementado (Figura 36).

Figura 36 - Memória de *stack pointer*



Um diagrama simplificado da estrutura de *stack pointer* é apresentado na Figura 37.

Figura 37 - Diagrama de *stack pointer*



Normalmente, a estrutura de *stack pointer*, apesar de se comportar como um banco de registros, é implementada utilizando-se memória RAM do próprio sistema. O processador faz um acesso a uma determinada região da memória RAM onde previamente se define um seguimento de endereço para esta finalidade.

3.4.3 Contador de programa

O contador de programa é um registrador que será utilizado como base do endereço de programa do processador. Este registrador armazena o endereço da instrução corrente do programa.

Este bloco é composto somente por um registrador de nbit e é controlado por um circuito complexo, onde é feita a multiplexação do próximo valor a ser armazenado nesse registrador.

3.5 Arquitetura final

Nesta parte, iremos combinar, de modo sequencial, as estruturas apresentadas nas seções anteriores, para se chegar à estrutura mais complexa.

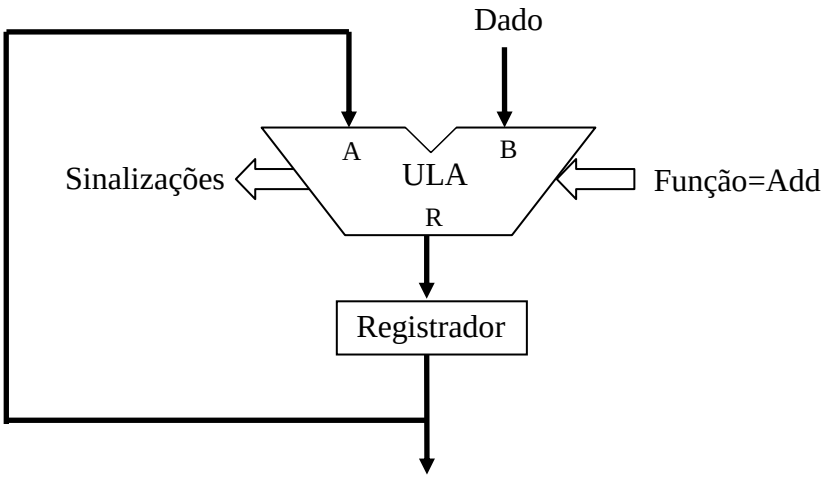
3.5.1 Acumulador

Essas estruturas complexas são dirigidas para realizar as operações mais comuns dos processadores. Neste trabalho, não serão abordados todos os detalhes da arquitetura, devido a algumas estruturas finais serem de propriedade da Freescale. Contudo, os passos e as ideias básicas para a criação do processador serão abordados.

Uma das primeiras abordagens será a criação da operação de adição e guardar o resultado em um registrador. Essa operação aparece frequentemente em muitas aplicações, sendo utilizada em muitos algoritmos; seu nome é acumulação.

$$RegA = RegA + Dado \tag{39}$$

Figura 38 - Diagrama do acumulador



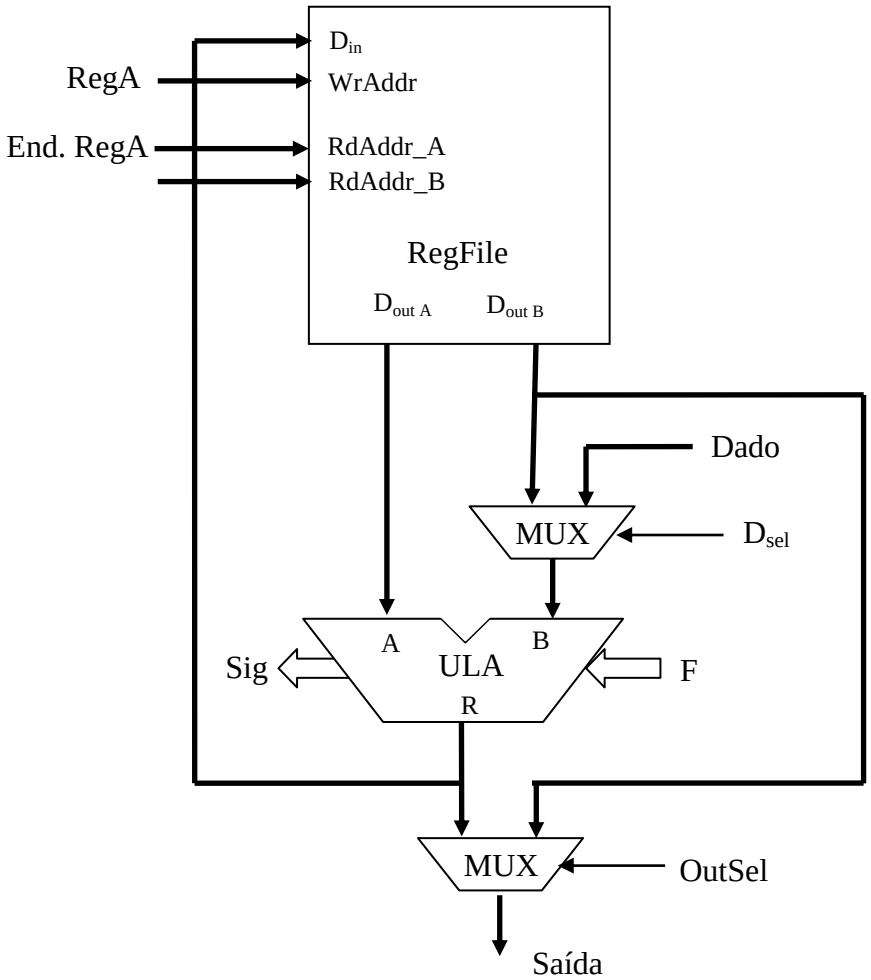
Utilizaremos a ULA, ao invés de uma estrutura de somador completo; assim, podemos extrapolar para qualquer outra operação que a ULA possa realizar.

O sinal de relógio e alguns sinais de controle não estão descritos na Figura 38, para manter a simplicidade no entendimento. Contudo, vale ressaltar que o registrador descrito tem sinais de controle que habilitam a escrita e também sinal de relógio para armazenamento. Assim, se a operação de acumulação for realizada sequencialmente, a cada ciclo de relógio o dado de saída do registrador será somado com os dados de entrada B da ULA e esse novo resultado da ULA será armazenado no registrador. Nesse ponto, é importante notar que o caminho que o dado faz é de importância extrema para a criação dessas estruturas complexas.

3.5.2 Acumulador com arquivo de registro

Modificando a estrutura anteriormente apresentada do acumulador, substituindo o registro acumulador pelo arquivo de registros (*Register File*) e reordenando os blocos na nova estrutura da Figura 39.

Figura 39 - Diagrama do registro de arquivo com ULA



Nessa nova estrutura, o registro acumulador será um dos registros internos do arquivo de registro; ainda temos o caminho de dados para que possamos implementar o acumulador apresentado anteriormente. No caso do acumulador, basta selecionar para que o multiplexador de entrada do operando B da ULA seja o dado de entrada, o endereço de escrita (*WrAddr*) seja o registro A e o endereço de leitura (*RdAddr_A*) seja o registro A.

A estrutura na Figura 39 é um pouco melhor, pois, além de implementar o acumulador, podemos utilizar, na entrada B da ULA, o dado armazenado em outro registrador, que estaria dentro do arquivo de registro, ou seja, podemos implementar qualquer função F que a ULA possa fazer entre dois registros.

O resultado dessa operação pode ser armazenado novamente em um registro do arquivo de registro e/ou ser externado através do multiplexador de saída (*OutSel*). Outro item que foi adicionado nessa estrutura é a possibilidade de selecionar um dado do registro de arquivo diretamente para a saída. Interessante notar que a escolha dos dados de saída é independente do armazenamento dos dados de resultado da ULA no arquivo de registro, podendo selecionar o resultado da operação aritmética ou um dado armazenado no registro de arquivo.

Com essa nova estrutura, podemos, de acordo com a seleção do registro de escrita, realizar uma sequência de operação ULA e armazenar os dados obtidos no arquivo de registro. Como exemplo, poderíamos fazer uma sequência de somas.

Saída= RegA + Dado * 4

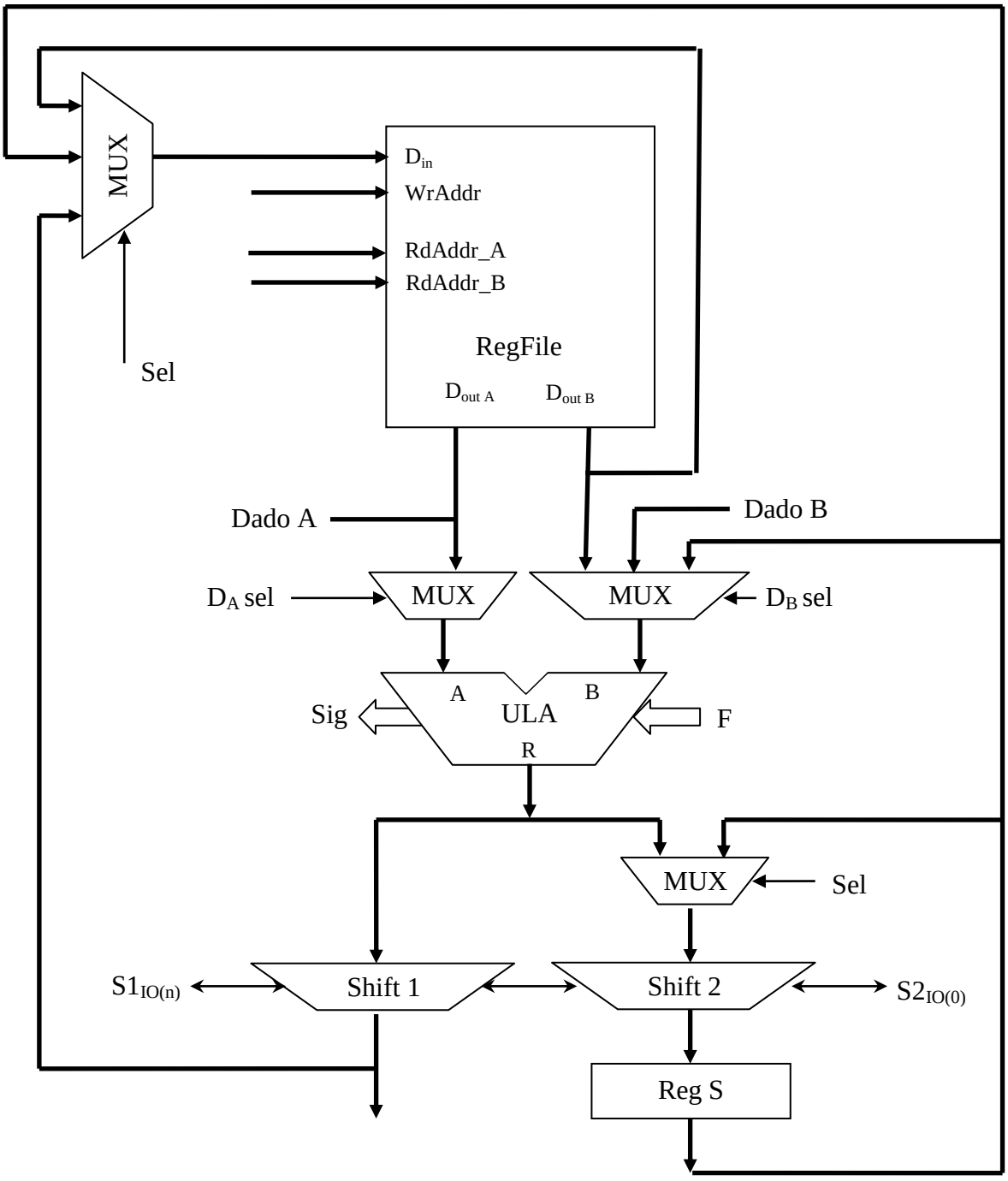
Essa sequência de somas poderia ser realizada da seguinte forma:

```
                RegB=4
LOOP:           RegA= RegA+Dado
                RegB=RegB-1
                Test (Z) RegB, se nao: LOOP
END:
```

Podemos, através do exemplo descrito, perceber que temos uma estrutura onde é possível, através de uma sequência de sinais de controle, realizar tarefas complexas. Neste exemplo, utilizamos a sinalização da ULA Z para fazer o teste se o número de operações foi alcançado. Podem existir várias formas de estruturar a mesma sequência. Poderia se usar, nesse mesmo exemplo, outra sinalização para implementar a mesma sequência; por exemplo, utilizar a sinalização de negativo (N), se tivéssemos colocado, no registro B, o valor inicial 3.

Continuando nossa sequência de melhoramentos da estrutura, poderíamos utilizar a estrutura apresentada para fazer operações mais complexas, como uma multiplicação, da mesma forma apresentada nesse exemplo. Contudo, não seria a forma mais eficiente de implementação dessa operação, como descrito anteriormente. Assim, para implementar a multiplicação e as operações de deslocamento, iremos adicionar o deslocador de bits. A operação será realizada como foi descrito quando discutimos a multiplicação, com uma sequência de deslocamentos e somas.

Figura 40 - Diagrama do registro de arquivo com ULA melhorado



A operação consiste em gerar produtos parciais e cada produto é deslocado de um bit para esquerda. Todos os produtos parciais são somados para se obter o resultado final. No caso de números binários, cada bit só pode ter valor 1 ou 0. Assim, cada produto parcial assume o valor zero ou o próprio valor do multiplicando. Essa característica simplifica a operação de multiplicação de números binários.

Observando a Figura 40, colocamos os deslocadores para poder realizar a multiplicação, como descrito anteriormente. Inicialmente, o multiplicador, que está no arquivo de registro (MPL), é carregado no registro S que está localizado depois do deslocador 2 (*Shift* 2). O registrador S tem o valor do multiplicador deslocado de 1 bit.

Nesse primeiro deslocamento, é utilizado o $S2_{IO(0)}$ para a condição onde é necessário somar o multiplicando (dado do registro de arquivo $D_{out\ A}$). O resultado é armazenado dentro do registro (PROD) interno ao registrador de arquivo.

A cada deslocamento, o bit menos significativo do resultado do deslocador 1 é deslocado para o bit mais significativo do deslocador 2; assim, esses bits são registrados no registro S.

É realizado um loop onde o $S2_{IO(0)}$ determina se haverá a soma entre os operadores A (PROD) e B (MPL) da ULA. Assim, após n interações, temos o resultado do produto na composição do registro PROD e S, sendo que o registro PROD seria os bits mais significativos e o registro S teria os valores menos significativos da multiplicação.

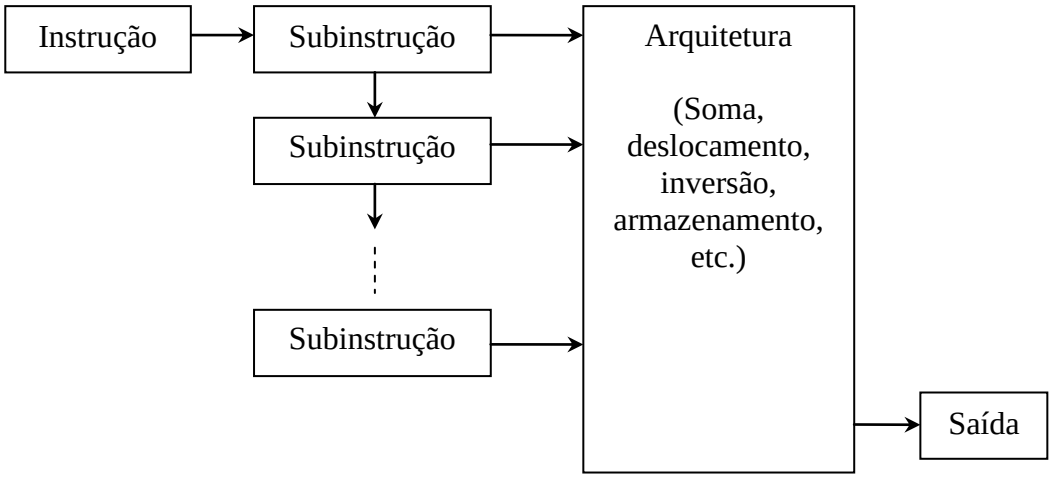
Caso seja necessário um multiplicador com menor número de passos, podemos incorporar, na estrutura aqui apresentada, um multiplicador combinacional ou um multiplicador de tabela. Este multiplicador pode ser composto por somente uma estrutura ou fazer a composição de pequenos blocos e, assim, compor uma unidade maior, dependendo da necessidade de projeto.

3.5.3 Sequenciador

Sequenciador é uma estrutura especializada na conexão dos dados entre as diversas estruturas que compõem o processador. Inicialmente, vamos considerar que qualquer instrução pode ser realizada com uma sequência de subinstruções, como ilustrado na Figura 41.

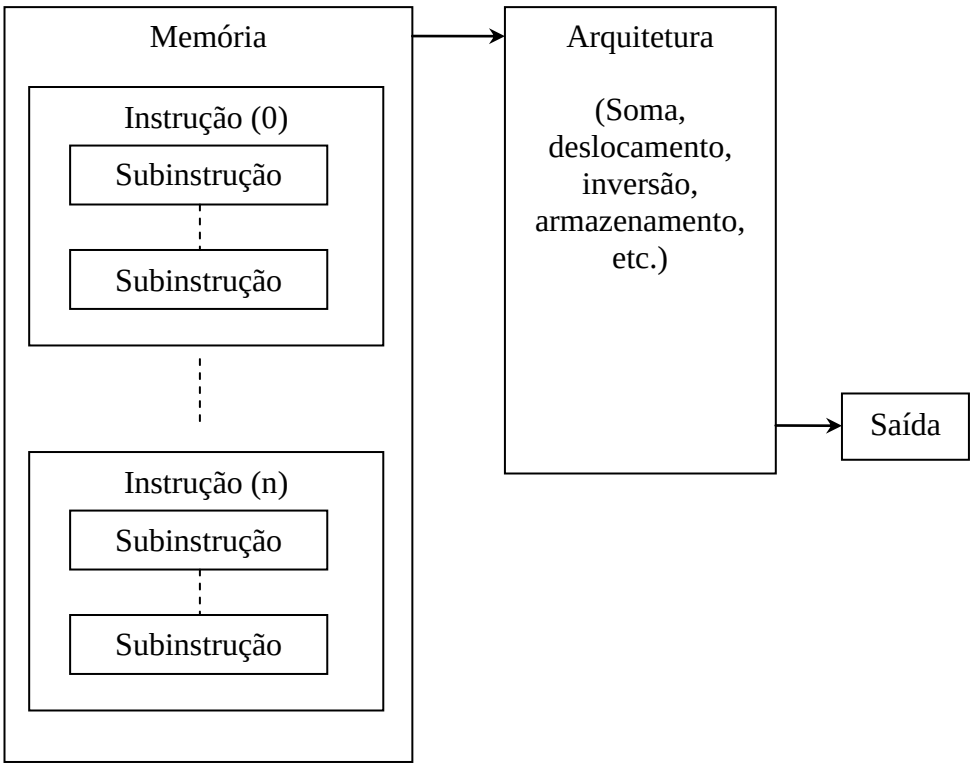
Essas subinstruções podem ser consideradas como caminhos para uma memória (ROM, PROM, RAM, FLASH, Máquina de estado) contendo as informações necessárias para a realização de uma determinada instrução.

Figura 41 - Composição da instrução



Na Figura 42, podemos observar que, dentro dessa memória, podem existir vários conjuntos de subinstruções. Cada conjunto de subinstrução representa uma instrução complexa executável pela arquitetura.

Figura 42 - Múltiplas instruções e subinstruções

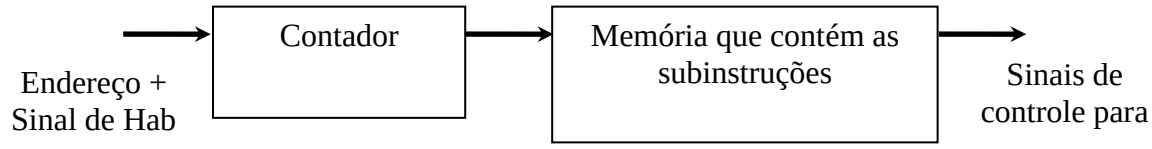


Iniciaremos nosso modelo a partir de uma composição simples de um contador com a memória contendo as subinstruções (Figura 43). Essa configuração permite compor uma instrução e suas subinstruções. Contudo, nessa configuração, somente é possível realizar uma

instrução. Gostaríamos que a memória fosse contemplada com todas as instruções possíveis do ISA.

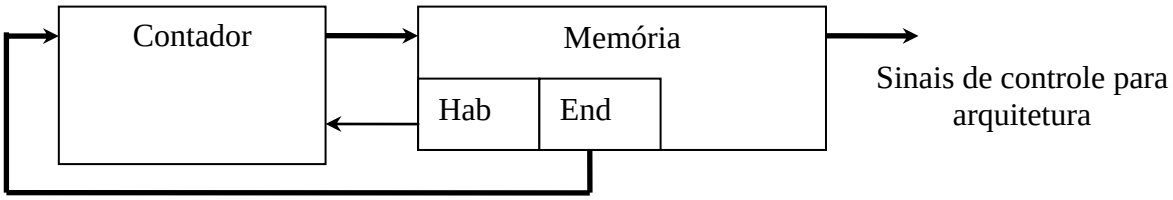
Para que possamos criar uma estrutura que possa executar mais do que uma instrução, necessitamos da possibilidade de trocar o valor de contador dependendo da instrução. Se essa estrutura puder trocar o valor do contador, também será possível a criação de instruções que possam ser compostas de pequenas sub-rotinas.

Figura 43 - Estrutura inicial de sequenciador



Primeiramente, precisamos de algum sinal de controle, para que o contador possa mudar o endereço da memória que será acessado de acordo com a necessidade das subinstruções armazenadas na memória.

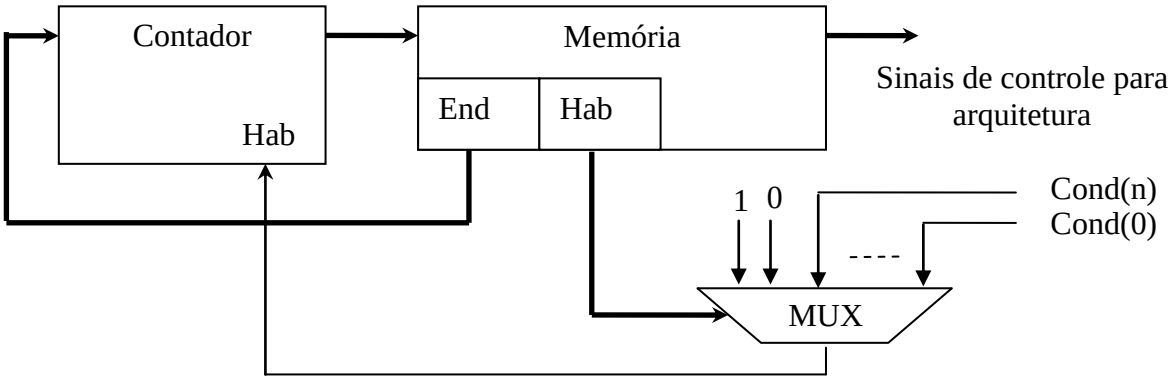
Figura 44 - Estrutura que pode fazer desvios na sequência de dados



A estrutura na Figura 44 se assemelha muito com a instrução *jump* (JMP) e pode ser implementada dessa forma.

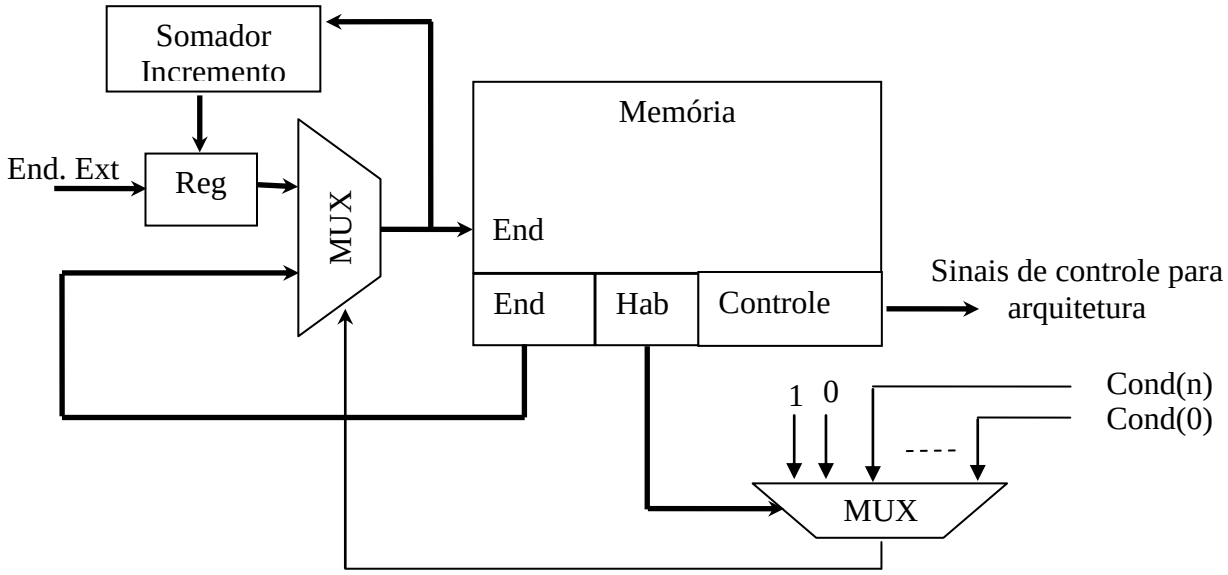
O próximo passo seria a possibilidade de pular de endereço de acordo com uma condição determinada por um evento externo. A instrução relacionada seria "*jump if*"ou "*branch if*".

Figura 45 - Estrutura para fazer desvios na sequência de dados dependendo de condições externas



A seleção do *mux* será feita pelos bits no campo "*Hab*", o número de bits desse sinal é definido pelo número de condições que possam existir para causar a mudança do valor do endereço armazenado no contador.

Figura 46 - Sequenciador melhorado



Nesse momento, podemos perceber que o endereço de desvio está passando pelo contador e sendo armazenado; assim, perdemos um ciclo de relógio para que esse endereço esteja disponível na entrada da memória. Para melhorarmos essa estrutura, iremos modificar o contador e adicionar um multiplexador na entrada de endereço da memória.

Na estrutura de sequenciador, apresentada até agora, podem ser adicionados estruturas de *stack pointer*, registros e multiplexadores para mudar o fluxo de dados para que atenda nossas necessidades. Assim, o fluxo de dados e os sinais de controle ficam cada vez mais complexos na estrutura apresentada até este momento.

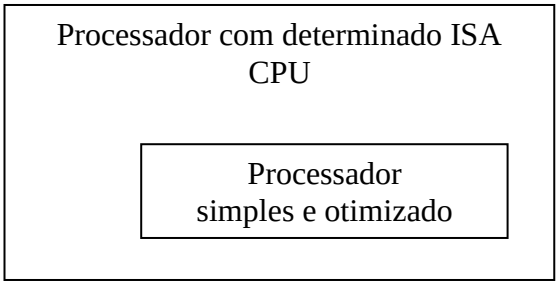
3.5.4 CPU

Dando sequência ao processador, já temos uma estrutura que pode, através de sinais de controle, realizar tarefas básicas. O sequenciamento desses sinais de controle poderia realizar tarefas mais complexas. Com a soma dos caminhos possíveis com o sequenciador, podemos executar tarefas complexas.

A Unidade Central de Processamento (CPU) é um circuito especializado que está acima da estrutura apresentada até agora e trata do carregamento do programa, decidindo como esses dados são interpretados pela estrutura interna e traduzindo as instruções do ISA

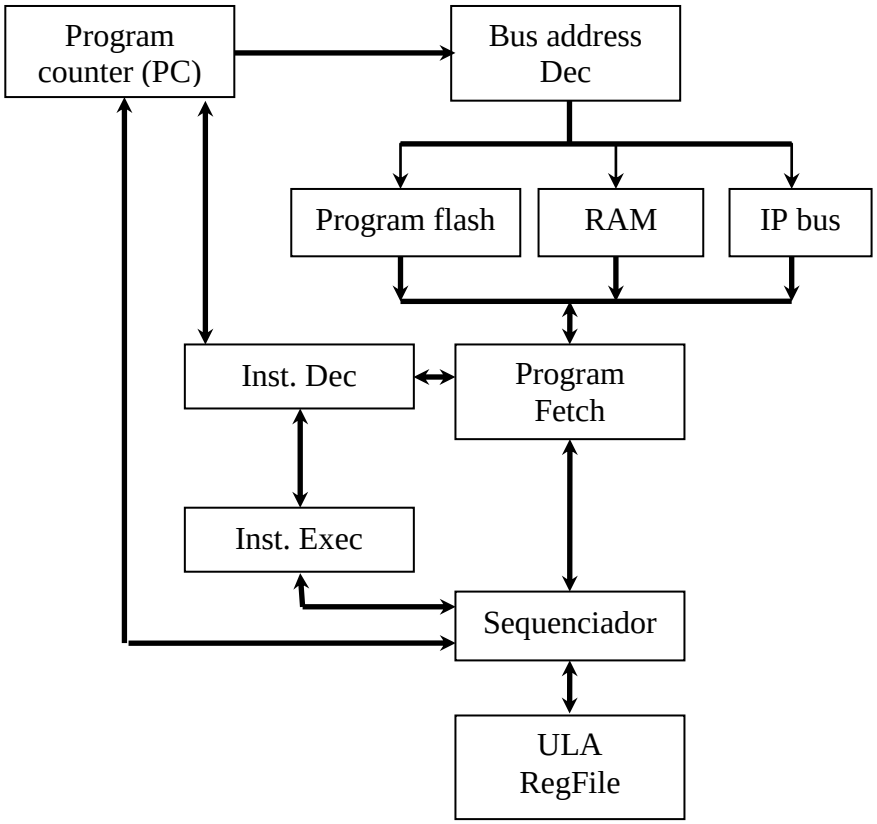
em uma sequência de sinais para o sequenciador. Por sua vez, o sequenciador executa microinstruções que, em conjunto, formam a execução da instrução do ISA. A Figura 47 ilustra o papel da CPU nesse trabalho.

Figura 47 - Ideia básica da CPU



Essa abordagem, de criar essa divisão, difere das implementações anteriores da arquitetura, sendo uma forma mais moderna de como são criados os processadores atuais. A vantagem desse tipo de estrutura é que podemos incluir novas instruções sem a necessidade de mexer no sequenciador. Basicamente, o sequenciador seria o responsável pelo microcódigo e a CPU pelo carregamento e pela decodificação das instruções de um determinado ISA, sendo também responsável pela geração dos sinais de controle para o sequenciador.

Figura 48 - Diagrama de blocos simplificados da CPU

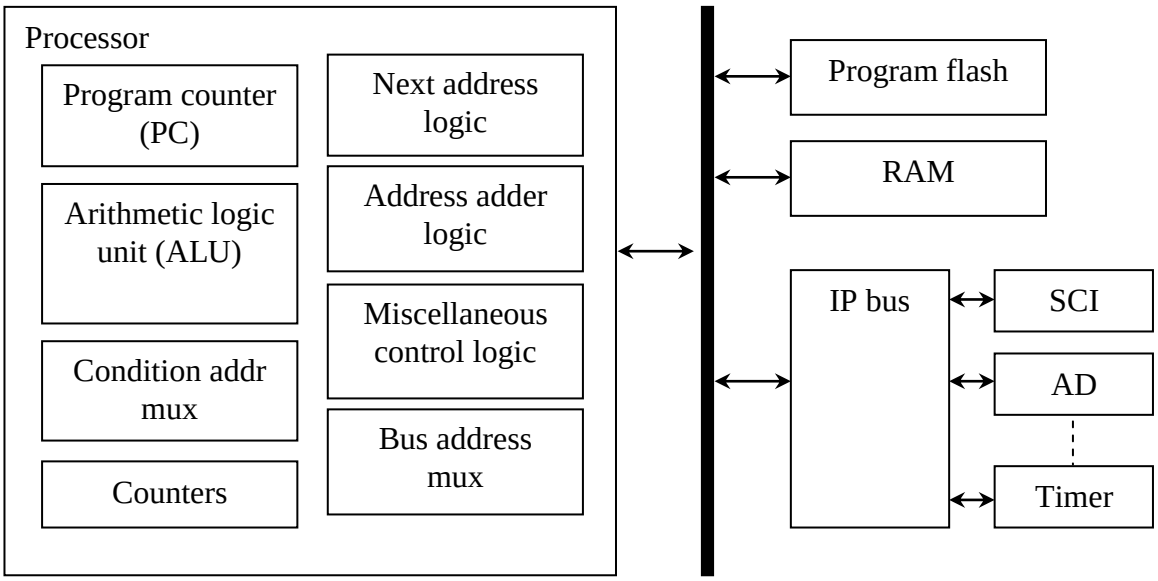


Esse tipo de abordagem é feito em muitos processadores, onde microinstruções são implementadas e a instrução propriamente dita da ISA é decodificada nessas subinstruções, para ser executada pela arquitetura. Muitos optam por pequenos processadores RISC, onde as poucas instruções podem ser realizadas – todas levando ao mesmo número de ciclos de máquina para poder aproveitar essas estruturas e adicionar estruturas de *pipeline* nas microinstruções.

3.5.5 Particularidade dos barramentos

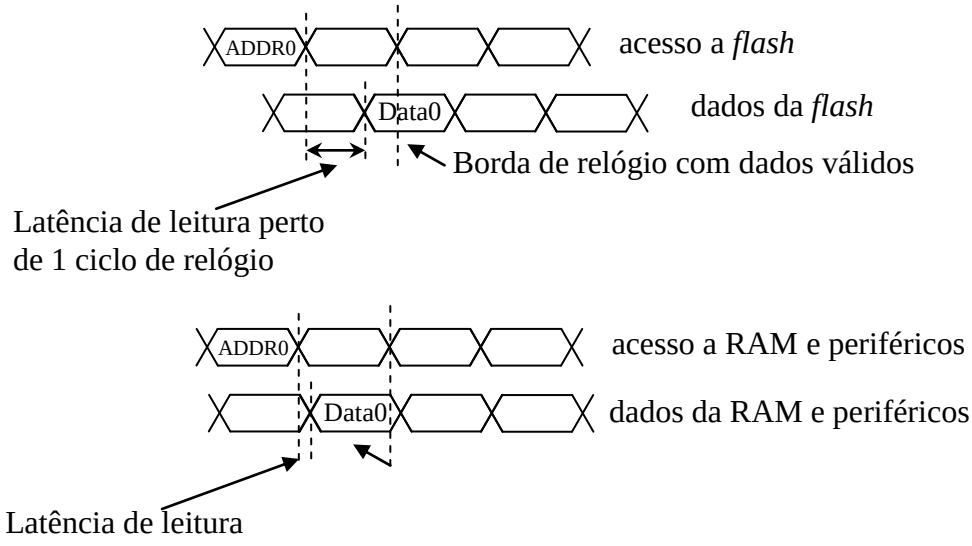
Um diagrama simplificado de como o microcontrolador escolhido está organizado pode ser visto na Figura 49. Podemos perceber que o microcontrolador escolhido é composto somente de um barramento para acesso a todos os blocos do sistema (*Flash*, RAM e periféricos). Nesse projeto, percebemos que o tempo de acesso de cada macrobloco, através do processar, é diferente. Normalmente, a flash tem a pior latência na entrega dos dados. Por outro lado, RAM e registros de periférico são mais rápidos.

Figura 49 - Diagrama de bloco do microcontrolador



Para aumentar a desempenho, através da diminuição do tempo necessário para realização de instruções que necessitem acesso a esses módulos (RAM, *Flash*, registro de periféricos), resolvemos mudar a arquitetura do processador, para que tenha barramento de acesso independente para cada um desses módulos. Assim, o tempo de acesso para cada módulo será diferente, consequentemente, uma instrução que faça acesso ao modulo flash demora mais ciclos de máquina do que a mesma instrução para acessar um registro de periférico ou RAM.

Figura 50 - Tempos de acesso a *flash*, RAM e periféricos



Como uma aplicação normalmente tem muitos acessos à memória e aos registros de periférico, o ganho desse tipo de abordagem será razoável. Contudo, mais circuito é necessário para a implementação do controle desses barramentos independentes.

Essas estruturas de barramentos independentes são utilizadas nesse trabalho. Acreditamos que, apesar do acréscimo de circuito, se obtém ganho de desempenho e este pode ser convertido para a criação de um processador que tenha menor consumo de potência do que o atual.

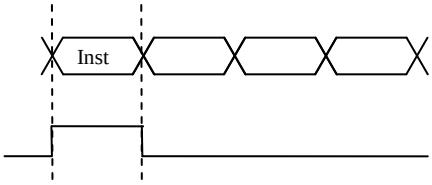
3.5.6 Carregamento dos dados

Um ponto importante do desenvolvimento de processadores é o tratamento de como os dados serão lidos do barramento para dentro do processador. Inicialmente, a primeira abordagem seria registrar os dados de leitura da memória (*flash*, ROM, RAM, etc.). Como estamos usando um ISA que é CISC, uma característica dessa arquitetura é que o tamanho das instruções varia de acordo com a instrução, pois os modos de endereçamento influenciam bastante nessa característica.

A leitura das instruções não é uma tarefa tão simples, se implementada de qualquer forma, acarreta na necessidade de muitos registros para se poder fazer a decodificação das instruções. Para a criação da estrutura de leitura de dados, iniciaremos com o número de registradores do maior número de bytes necessários das instruções que compõem esse ISA.

Nesse ISA, são possíveis instruções de 1, 2 e 3 bytes. A Figura 51, a Figura 52 e a Figura 53 ilustram os possíveis acesso desse ISA. A ideia é criar uma estrutura de registro na qual a informação da instrução corrente fique armazenada até o momento da execução.

Figura 51 - Instrução de um byte



A importância da estrutura de carregamento de dados é permitir o correto armazenamento dos dados e posicionar a instrução para que haja decodificação correta da instrução. Se essa estrutura de carregamento de dados "confundir" a ordem dos dados, pode acarretar na perda de sincronismo da execução do programa.

Figura 52 - Instrução de dois bytes

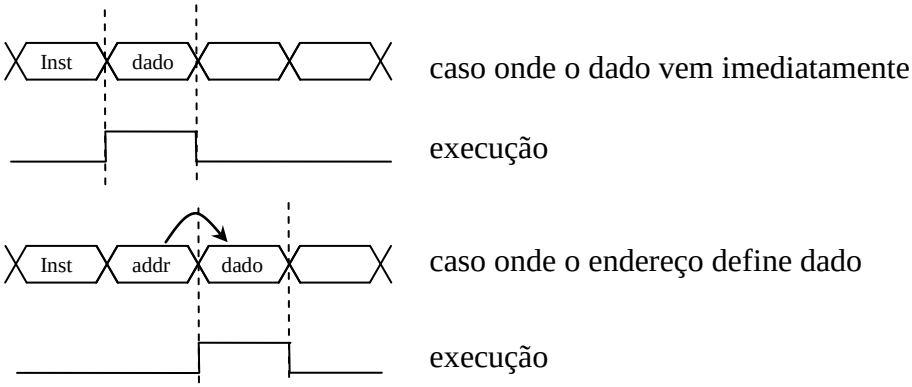
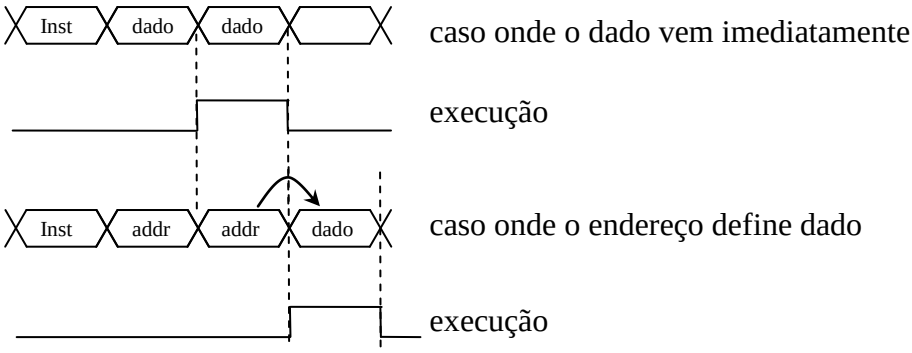


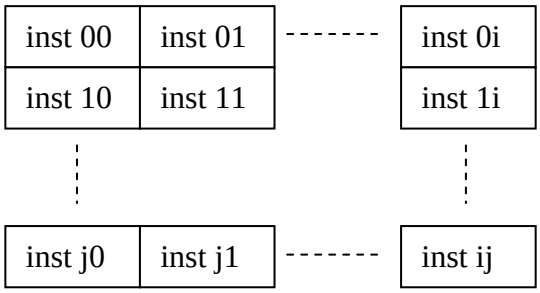
Figura 53 – Instrução de 3 bytes.



3.5.7 Decodificador de instruções

O decodificador de instrução é muito importante para interpretação da instrução propriamente dita, enviando sinais importantes para o controle do fluxo de dados para as estruturas corretas dentro do sistema.

Figura 54 - Mapa de instruções



Normalmente, as instruções são agrupadas em linhas e colunas dentro do mapa de instruções do ISA. O motivo para esse agrupamento é tentar juntar as instruções que têm uma determinada característica em comum. Esses agrupamentos acontecem por linha e ou por coluna. O motivo desse agrupamento é tentar facilitar o circuito necessário para decodificação de instruções. O mapa de instrução do ISA escolhido pode ser visto no Anexo A desse trabalho.

Abaixo, apresentamos alguns trechos de código que exemplificam a decodificação das instruções para esse ISA.

```
// ------------------------------------------------------------------------------
// addressing mode signal generation
// ------------------------------------------------------------------------------
assign addrmode_dir_1 = (pipe_reg0[7:4]==`ADDR_DIR_1) && !page2;
assign addrmode_dir_2 = (pipe_reg0[7:4]==`ADDR_DIR_2) && !page2;
assign addrmode_dir_3 = (pipe_reg0[7:4]==`ADDR_DIR_3) && !page2;
assign addrmode_dir_4 = (pipe_reg0[7:4]==`ADDR_DIR_4) && !page2;
assign addrmode_rel = (pipe_reg0[7:4]==`ADDR_REL) && !page2;
assign addrmode_inh_1 = (pipe_reg0[7:4]==`ADDR_INH_1) && !page2;
assign addrmode_inh_2 = (pipe_reg0[7:4]==`ADDR_INH_2) && !page2;
assign addrmode_inh_3 = (pipe_reg0[7:4]==`ADDR_INH_3) && !page2;
assign addrmode_inh_4 = (pipe_reg0[7:4]==`ADDR_INH_4) && !page2;
assign addrmode_ix1_1 = (pipe_reg0[7:4]==`ADDR_IX1_1) && !page2;
assign addrmode_ix1_2 = (pipe_reg0[7:4]==`ADDR_IX1_2) && !page2;
assign addrmode_ix_1 = (pipe_reg0[7:4]==`ADDR_IX_1) && !page2;
assign addrmode_ix_2 = (pipe_reg0[7:4]==`ADDR_IX_2) && !page2;
assign addrmode_imm = (pipe_reg0[7:4]==`ADDR_IMM) && !page2;
assign addrmode_ext = (pipe_reg0[7:4]==`ADDR_EXT) && !page2;
assign addrmode_ix2 = (pipe_reg0[7:4]==`ADDR_IX2) && !page2;
assign addrmode_sp2 = (pipe_reg0[7:4]==`ADDR_SP2) && page2;
assign addrmode_sp1_1 = (pipe_reg0[7:4]==`ADDR_SP1_1) && page2;
assign addrmode_sp1_2 = (pipe_reg0[7:4]==`ADDR_SP1_2) && page2;
assign addrmode_sp1_3 = (pipe_reg0[7:4]==`ADDR_SP1_3) && page2;

// ------------------------------------------------------------------------------
assign addrmode_dir = addrmode_dir_1 || addrmode_dir_2 || addrmode_dir_3
||addrmode_dir_4;
assign addrmode_inh = addrmode_inh_1 || addrmode_inh_2 || addrmode_inh_3 ||
addrmode_inh_4;
assign addrmode_ix = addrmode_ix_1 || addrmode_ix_2;
assign addrmode_ix1 = addrmode_ix1_1 || addrmode_ix1_2;
assign addrmode_sp1 = addrmode_sp1_1 || addrmode_sp1_2 || addrmode_sp1_3;

// ------------------------------------------------------------------------------
assign type_reg_mem =  (addrmode_imm || addrmode_dir_4 || addrmode_ext ||
addrmode_ix2 || addrmode_ix_2);
```

```
assign type_rd_mod_wr = (addrmode_dir_3 || addrmode_inh_1 || addrmode_inh_2 ||
addrmode_ix1_1 || addrmode_ix_1);

// ------------------------------------------------------------------------------
// auxiliar instuction low nible row decoding signal
// ------------------------------------------------------------------------------
assign row_0 = (pipe_reg0[3:0]==4'h0);
assign row_1 = (pipe_reg0[3:0]==4'h1);
assign row_2 = (pipe_reg0[3:0]==4'h2);
assign row_3 = (pipe_reg0[3:0]==4'h3);
assign row_4 = (pipe_reg0[3:0]==4'h4);
assign row_5 = (pipe_reg0[3:0]==4'h5);
assign row_6 = (pipe_reg0[3:0]==4'h6);
assign row_7 = (pipe_reg0[3:0]==4'h7);
assign row_8 = (pipe_reg0[3:0]==4'h8);
assign row_9 = (pipe_reg0[3:0]==4'h9);
assign row_a = (pipe_reg0[3:0]==4'ha);
assign row_b = (pipe_reg0[3:0]==4'hb);
assign row_c = (pipe_reg0[3:0]==4'hc);
assign row_d = (pipe_reg0[3:0]==4'hd);
assign row_e = (pipe_reg0[3:0]==4'he);
assign row_f = (pipe_reg0[3:0]==4'hf);

// ------------------------------------------------------------------------------
// instruction decoding
// ------------------------------------------------------------------------------
assign inst_sub = row_0 && type_reg_mem;
assign inst_cmp = row_1 && type_reg_mem;
assign inst_sbc = row_2 && type_reg_mem;
assign inst_cpx = row_3 && type_reg_mem;
assign inst_and = row_4 && type_reg_mem;
assign inst_bit = row_5 && type_reg_mem;
assign inst_lda = row_6 && type_reg_mem;
assign inst_sta = row_7 && type_reg_mem && !addrmode_imm;
assign inst_eor = row_8 && type_reg_mem;
assign inst_adc = row_9 && type_reg_mem;
assign inst_ora = row_a && type_reg_mem;
assign inst_add = row_b && type_reg_mem;
assign inst_jmp = row_c && type_reg_mem && !addrmode_imm;
assign inst_jsr = row_d && type_reg_mem && !addrmode_imm;
assign inst_ldx = row_e && type_reg_mem;
assign inst_stx = row_f && type_reg_mem && !addrmode_imm;
assign inst_ais = row_7 && addrmode_imm;
assign inst_aix = row_f && addrmode_imm;

assign inst_asr = row_7 && type_rd_mod_wr;
assign inst_asl = row_8 && type_rd_mod_wr;

assign inst_neg = row_0 && (addrmode_dir_3 || addrmode_inh_1 || addrmode_inh_2 ||
addrmode_ix1_1 || addrmode_ix_1);


// ------------------------------------------------------------------------------
// alu operation select
// ------------------------------------------------------------------------------
assign alu_oper_add_ab = inst_adc | inst_add | inst_ais | inst_aix;
assign alu_oper_sub_ba = inst_sub | inst_sbc;
assign alu_oper_and = inst_and;
assign alu_oper_or = inst_ora;
assign alu_oper_xor = inst_eor;
assign alu_oper_neg = inst_neg;


// ------------------------------------------------------------------------------
// To ALU
// ------------------------------------------------------------------------------
assign ctrl_alu_oper_add_ab = alu_oper_add_ab | alu_oper_pc_lo_add_1;
assign ctrl_alu_oper_sub_ba = alu_oper_sub_ba;
```

```
assign ctrl_alu_oper_and = alu_oper_and;
assign ctrl_alu_oper_or = alu_oper_or;
assign ctrl_alu_oper_xor = alu_oper_xor;
assign ctrl_alu_oper_neg = alu_oper_neg
```

Como podemos observar, essa decodificação é feita totalmente combinacional a partir do código da instrução armazenado em um registrador. Esse registrador, que armazena a instrução, é mantido estável até o fim da execução do comando, onde novamente é carregado com o código da nova instrução.

O motivo para termos tanto cuidado em armazenar a instrução é porque todas as gerações de sinais de controle, como dito anteriormente, são feitas de forma combinacional. Essa ordem de armazenar a instrução para depois gerar os sinais de controle economiza a quantidade de registradores necessários para armazenar informações da decodificação das instruções.

Uma máquina de estado pode ser utilizada para gerar os sinais de controle dos registradores que contêm o código da instrução corrente e dos dados dessa instrução, selecionando, de forma seletiva, o carregamento desses registradores com os dados em uma ordem correta para a fase de execução da instrução. Um pequeno trecho dessa máquina está ilustrado no código abaixo.

```
`STATE_0:
  begin // 1
    if (addrmode_imm)
        wr_pipe_reg_tmp_en = 1'b1;
    else if (addrmode_dir | addrmode_ix1 | addrmode_sp1)
      begin
        wr_pipe_reg_tmp_en = 1'b1;
        core_ab_sel = SEL_00_PIPE1;
        pc_reg_ld_b = 1'b1;
      end
    else if (addrmode_ix)
      begin
        ctrl_pipe_reg0_mux_sel_reg1 = 1'b1;
        wr_pipe_reg0_en = 1'b1;
        wr_pipe_reg1_en = 1'b1;
        ctrl_last_cycle = 1'b1;
      end
    else if (inst_page2)
      begin
        ctrl_pipe_reg0_mux_sel_reg1 = 1'b1;
        wr_pipe_reg0_en = 1'b1;
        wr_pipe_reg1_en = 1'b1;
        ctrl_last_cycle = 1'b1;
      end

    else
      begin
        wr_pipe_reg2_en = 1'b1;
      end
  end
`STATE_1:
  begin // 1
    if (addrmode_imm)
      begin
```

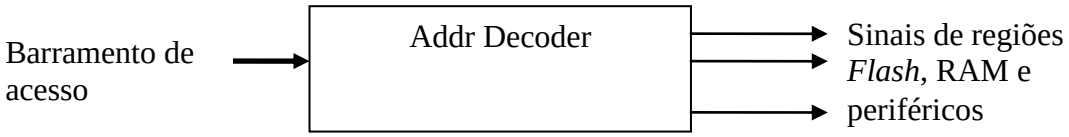
```
          wr_pipe_reg0_en = 1'b1;
          wr_pipe_reg1_en = 1'b1;
          ctrl_last_cycle = 1'b1;
        end
      else if (addrmode_dir | addrmode_ix1 | addrmode_sp1)
        begin
          if (direct_access_region_reg)
            begin
              ctrl_last_cycle = 1'b1;
              wr_pipe_reg0_en = 1'b1;
              wr_pipe_reg1_en = 1'b1;
            end
          else
            begin
              wr_pipe_reg1_en = 1'b1;
              if (core_ipbi_sel_reg)
                rdata_sel_ipbi = 1'b1; // remember rdata is flash in other case
            end

        end
      else
        begin
          wr_pipe_reg_tmp_en = 1'b1;
          ctrl_flash_ab_pc_sel = 1'b1;
          core_ab_sel = SEL_PIPE12;
          pc_reg_ld_b = 1'b1;
        end

    end
  `STATE_2:
    begin // 1
      if (addrmode_dir | addrmode_ix1 | addrmode_sp1)
        begin
          ctrl_last_cycle = 1'b1;
          wr_pipe_reg0_en = 1'b1;
          wr_pipe_reg1_en = 1'b1;
        end
      else if (direct_access_region_reg)
        begin
          ctrl_last_cycle = 1'b1;
          wr_pipe_reg0_en = 1'b1;
          wr_pipe_reg1_en = 1'b1;
        end
      else
        begin
          wr_pipe_reg1_en = 1'b1;
          if (core_ipbi_sel_reg)
            rdata_sel_ipbi = 1'b1; // remember rdata is flash in other case
        end
    end
```

3.6 Decodificador de endereço

O decodificador de endereço é uma peça fundamental nesse trabalho, uma vez que essa estrutura determina qual região (*flash*, RAM ou periférico) o processador está acessando. Ele é o responsável pela determinação da região de mapeamento que será usada, determinando o tipo de periférico (*flash*, memória, registro, etc.). O decodificador de endereço gera os sinais que informam o acesso do processador a essas regiões; os sinais serão utilizados por outros blocos para cadenciar o fluxo de dados nos ciclos de máquina corretos para o acesso (Figura 55).

Este bloco também é responsável por gerar alguns outros sinais, como acesso inválido para regiões não mapeadas da memória. Os parâmetros desse decodificador de endereço são dependentes de cada microcontrolador, uma vez que os endereços de periféricos, RAM e Flash diferem entre projetos.

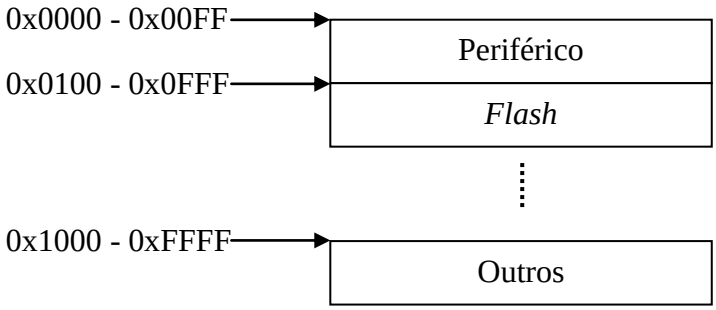
Figura 55 - Ilustração do decodificador de endereço



O barramento de acesso é um barramento interno, sendo que o valor nesse barramento define onde o processador deve acessar para obter o próximo dado. O valor desse barramento varia de acordo com o endereçamento definido para o microcontrolador; isto é, existe um mapeamento de endereços para cada bloco do sistema, ilustrado de uma forma simplificada na Figura 56.

Como o mapa varia de um microcontrolador para outro, é importante criar esse bloco de forma parametrizada, para facilitar mudanças nos endereços dos blocos e periféricos.

Figura 56 - Exemplo de mapa de memória do microcontrolador



O controle do barramento de acesso pode ser realizado por uma máquina de estado que direciona os registros que contêm o próximo endereço a ser acessado pelo processador. Essa máquina de estado utiliza-se dos sinais de controle gerados pelo decodificador de instrução para fazer o controle do fluxo de dados para o decodificador de endereço.

Pequenos trechos de código desse decodificador estão ilustrados abaixo. Pode-se notar que os endereços de comparação são parâmetros que são dependentes do microcontrolador. Existem circuitos de controles mais complexos para gerar os sinais propriamente ditos, onde esses controles impedem e arbitram os sinais de seleção de cada módulo (RAM, *flash*, periféricos, etc.).

```
always @ ( core_ab or mmu_paged_access or mmu_regs_access)
begin
  if ( core_ab >= 16'h0000 && core_ab <= DP_UPPER && ~mmu_paged_access &&
       ~mmu_regs_access)
    core_dp_sel = 1'b1;
  else
    core_dp_sel = 1'b0;
end // PERIFERICO
```

.....

```
always @ ( core_ab or mmc_block_hp_flseep or mmu_paged_access or
           mmu_regs_access)
  begin
    if ( core_ab >= 16'h1800  &&  core_ab <= HP_UPPER &&
         ~mmc_block_hp_flseep && ~mmu_paged_access && ~mmu_regs_access
       )
      core_hp_sel = 1'b1;
    else
      core_hp_sel = 1'b0;
  end // HIGH_PAGE_MUX
```

.....

```
always @( core_ab or ram_lower_cfg or ram_upper_cfg)
  begin
    if ( core_ab >= ram_lower_cfg && core_ab <= ram_upper_cfg )
      mmc_ram_primary_sel = 1'b1 ;
    else
      mmc_ram_primary_sel = 1'b0 ;
  end // RAM
```

.....

Capítulo - 4. Método comparativo de resultados

Durante o desenvolvimento dos modelos para comparação, foi necessário o desenvolvimento de uma metodologia de criação de *padring* para a rápida prototipagem. Apresentaremos como isto foi desenvolvido e os principais resultados obtidos. Em seguida, serão apresentados os principais resultados dos processadores discutidos e comparados com consumo de potência e a área necessária da nova microarquitetura proposta neste trabalho.

4.1 Criação de método de geração de *padring* para microcontroladores

A definição do *padring* e suas interconexões para SoC de microcontrolador é um processo trabalhoso e que consome muito tempo no início da criação do RTL de integração [39][40]. Como era necessária a criação de SoC completos para a realização de teste de consumo de potência – e também pela possibilidade de utilizar uma nova metodologia em futuros dispositivos –, foi feito um trabalho no qual a geração do *padring* fosse automatizada.

Em um microcontrolador, a quantidade de pinos é limitada, contudo, a funcionalidade associada a cada pino tem crescido a cada novo projeto. Uma confiável e rápida criação do RTL com as células de *pad* e suas interconexões diminui o tempo necessário para criação do dispositivo [41] e também diminui os custos do desenvolvimento desses dispositivos [39].

Existem dois pontos relevantes na criação de *padring*, um é o posicionamento e a organização física das células de *pad* [39][40][42], o outro é o aspecto lógico, que inclui o tipo de *pad*, a multiplexação das funcionalidades e os pinos de teste [43].

Pad é uma célula física que faz a interface com o mundo exterior e o circuito interno. Existem muitos tipos de *pads*, como: alimentação, quinas (*corner*), espaçadores (*spacer*), alta tensão, entrada/saída digitais ou analógica, isolação e assim por diante.

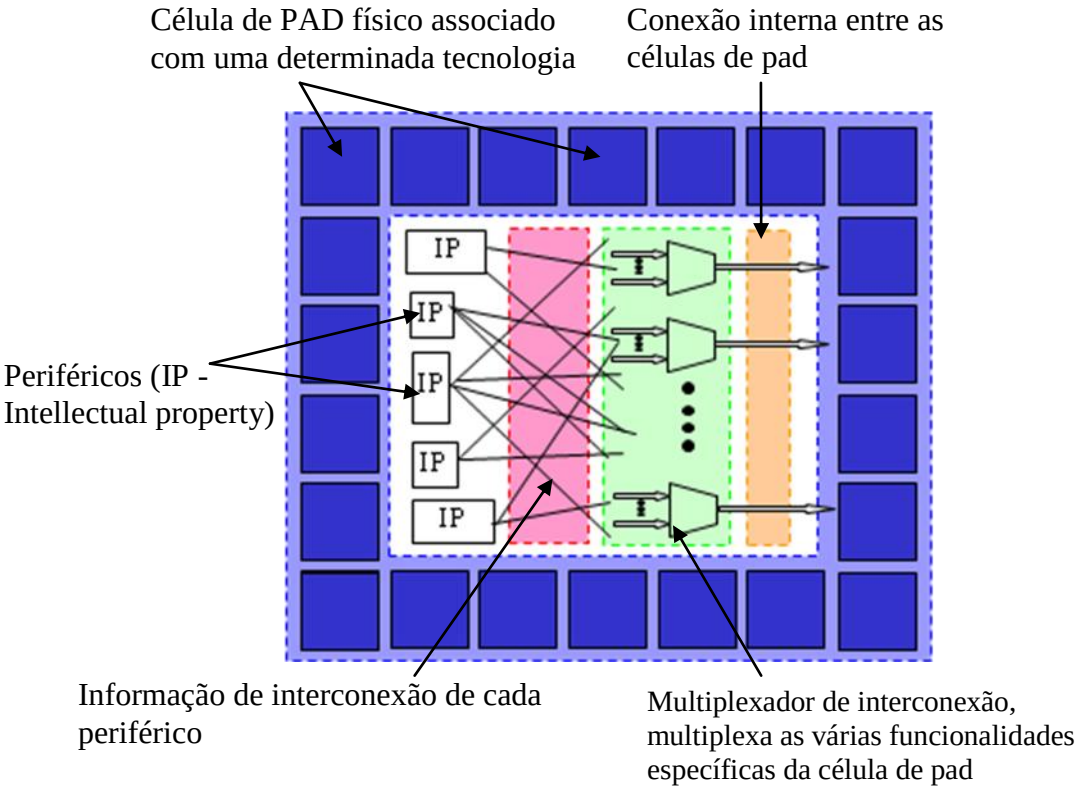
Cada *pad* tem muitos sinais para controlar as características associadas com cada célula física de *pad*, como direção do sinal, *pull up*, *pull down*; se o sinal vai ser digital ou analógico, *buffers* de entrada, *drivers* de saída e muitas outras características específicas de cada célula. Cada tipo de célula de *pad* tem seu próprio conjunto de sinais de controle. É importante a escolha correta da célula para se obter a função desejada no projeto.

Os *pads* de entrada/saída (IO – *input/output*) são a maioria nos projetos de microcontroladores. Devido à limitação na quantidade de pinos que um dispositivo pode ter, estes *pads* compartilham muitas funcionalidades [44], incluindo função de teste [45], e precisam ser conectados a muitos periféricos e circuitos que compõem o SoC de microcontrolador.

Por exemplo, uma célula de *pad* pode ser utilizada para entrada de um sinal analógico para um IP específico como um conversor AD (analógico digital); ou como entrada/saída de uma porta genérica; ou como entrada de relógio para periférico de *timers*; ou também para funcionalidade de teste.

Para cada função que a célula de *pad* pode assumir, existe um conjunto de sinais de controle que precisam ser configurados. Também existem sinais que precisam ser conectados para o correto módulo de circuito ou a um sinal interno. Convertendo em números, essas conexões variam de 10.000, em pequenos microcontroladores, até mais que 100.000 conexões, em dispositivos mais complexos; estes números são baseados em projetos de outros microcontroladores já implementados. Na Figura 57, é mostrada a ilustração de um *padring* de microcontrolador.

Figura 57 - *Padring* de microcontroladores



Na criação de *padring* tradicional, são utilizadas planilhas e pequenos *scripts* para ajudar a criação do código RTL. Mas esse método tradicional não é independente da tecnologia dos *pads* e não tem forte ligação com reuso de dados de projetos anteriores [40][46]. Existem alguns *softwares* no mercado que ajudam na geração de *padring*, mas estão mais restritos à parte física de posicionamento e organização das células.

O método aqui desenvolvido está diretamente associado às ligações lógicas dos sinais e às escolhas das células de *pad* a serem utilizadas.

4.1.1 Blocos básicos de *padring*

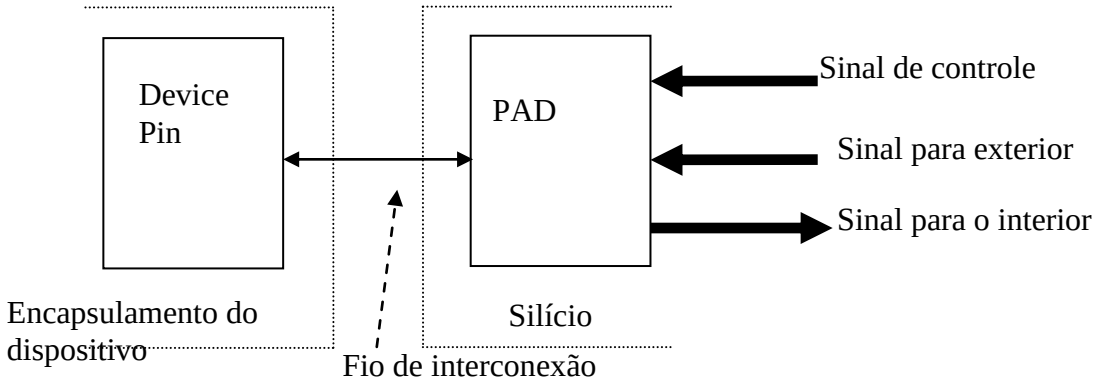
Basicamente, o *padring* é composto de alguns blocos e suas interconexões. Para entendermos melhor como essa metodologia funciona, iremos detalhar os blocos que compõem o *padring* de microcontroladores.

4.1.1.1 Célula de *PAD*

Cada célula *pad* tem seu conjunto de entradas e saídas. As entradas são sinais de controle do comportamento que o *pad* deve assumir; estes configuram e selecionam a funcionalidade embutida na célula de *pad*. Também incluem sinal de entrada, que deve ser interconectado com o circuito interno.

As saídas da célula de *pad* são o resultado do sinal que vem do mundo exterior e que deve ser interconectado com o circuito interno de um periférico ou outra célula. A Figura 58 mostra o bloco *pad* e como os sinais se interconectam dentro do dispositivo.

Figura 58 - Célula de *pad*



Para cada tipo de célula de *pad*, o número de entradas/saídas varia. Há casos que não existem sinais de entradas ou saída, como, por exemplo, células de *corner* e espaçadores. Também existem outros com muitas entradas e saídas, como células de IO.

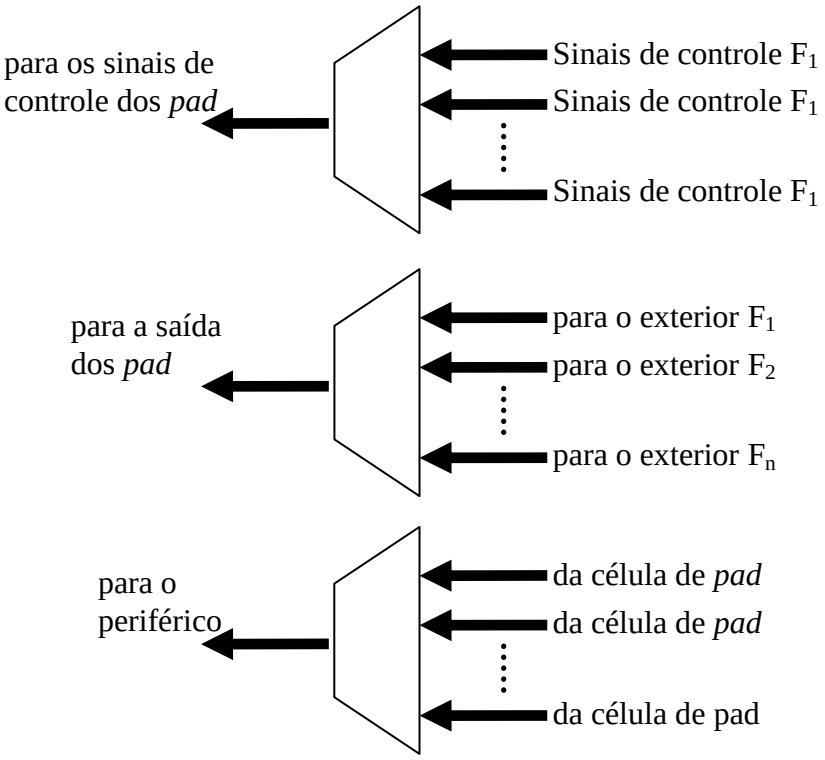
4.1.1.2 Bloco de multiplexação

O bloco de multiplexação é um módulo composto com muitos multiplexadores que tem a função de encaminhar o sinal correto para a célula de *pad*. Normalmente, as

interconexões do bloco de multiplexação são únicas para cada projeto de SoC, devido aos requisitos específicos de cada microcontrolador.

Células de IO normalmente dividem muitas funcionalidades e necessitam ser conectadas a blocos específicos internamente, dependendo de como queremos usar esse IO. Para cada IO, existem muitas possibilidades de configuração de suas entradas e saídas. Para atender essa flexibilidade, todos os sinais relacionados à configuração e às entradas e saídas da célula de *pad* devem ser multiplexados e estar disponíveis para a célula de *pad* quando o pino for configurado para uma determinada funcionalidade. A Figura 59 ilustra o diagrama de bloco do multiplexador.

Figura 59 - Bloco multiplexador

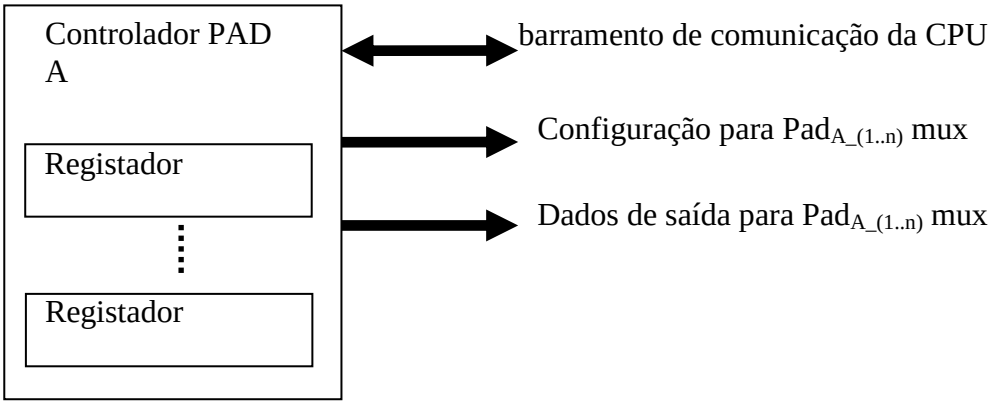


4.1.1.3 Controlador de porta

O bloco controlador de porta é composto de registradores que guardam os valores de como a célula de *pad* será configurada e está ilustrado na Figura 60. Neste bloco controlador de porta, são armazenadas as informações de qual funcionalidade o *pad* deve atender em um momento específico, gerando os sinais de controle corretos para o bloco multiplexador. Este é o responsável por selecionar a direção do sinal do pino e como *buffers*, *drivers* e multiplexadores internos à célula de *pad* devem ser configurados para atender a funcionalidade requerida.

A comunicação com esse bloco é feita usando-se o processador, que tem o controle de escrita de informação necessário para cada registrador interno a esse bloco. Normalmente, esse tipo de configuração é feito na inicialização do microcontrolador. Contudo, caso necessite a reconfiguração da função do *pad* durante a execução do programa, estas configurações podem ser modificadas dinamicamente.

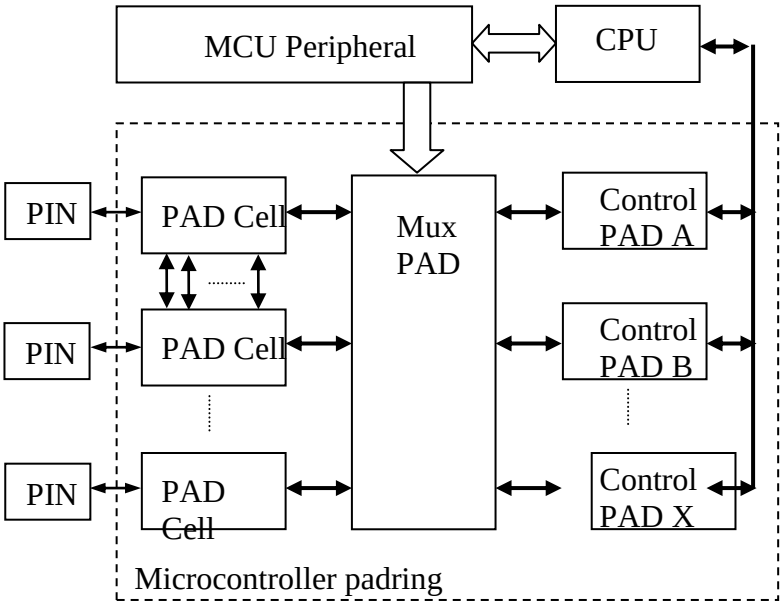
Figura 60 - Controlador de porta



Este bloco também é responsável pelos sinais de IO genéricos, armazenando dados para serem transferidos para o mundo exterior, através das células de *pad*, ou capturando dados externos e entregando-os ao processador. Cada instância desse bloco normalmente referencia um conjunto de portas do microcontrolador (PTA[n:0], PTB[i:0] e assim por diante).

4.1.1.4 *Padring* de microcontroladores

Figura 61 - *Padring* de microcontrolador



Todos os blocos apresentados até o momento compõem o *padring* de um microcontrolador quando interconectados entre eles. A Figura 61 mostra uma ilustração básica do *padirng* de microcontroladores.

Além das conexões entre os circuitos internos e entre os blocos, existem também as conexões entre as células de *pad*, isto é, referentes à ligação dos sinais de alimentação que compõem o anel de alimentação.

4.1.2 Metadado e banco de dados

Todas as informações relacionadas a conexões, células de *pad* e blocos precisam ser uniformizadas e guardadas em um banco de dados para serem utilizadas na geração do código RTL.

Cada SoC tem um código RTL único que precisa ser regerado com parâmetros específicos para cada novo projeto. Para criar esse banco de dados, as informações foram divididas em seguimentos menores (metadado). O primeiro metadado "pad" é composto de vários parâmetros. Um exemplo de metadado de "pad" é ilustrado na Tabela 11. Para facilitar a utilização entre várias tecnologias independentemente, foi criado um pseudônimo para cada tipo de célula de pad. Normalmente, esse nome é escolhido de forma a refletir a funcionalidade primária que cada célula de pad pode realizar (port, power, corner, etc.) e está armazenado no campo "Connection Name".

O nome da tecnologia define um conjunto de células de *pad* que têm características elétricas e tecnológicas comum, como *pads* com a mesma tensão de alimentação e tecnologia específica (250nm, 180nm, etc.).

O campo "*Mux Possible*" é uma sinalização (verdadeiro/falso) que define se o *pad* pode ser utilizado para múltiplas funções. Geralmente, essa sinalização é verdadeira para as células de IO.

O campo "Pad Name" armazena o nome da célula dentro da tecnologia definida. Assim, cada tecnologia tem um nome específico de célula, ainda que essa célula tenha a mesma funcionalidade, costuma-se ter nomes que reflitam a tecnologia usada. No caso do exemplo da Tabela 11, indica uma célula para IO genérico com tensão de 3V e 4 níveis de metais.

A última informação adicionada a esse metadado é o campo "Connection". Ele contém a informação dos sinais de entrada/saída que devem ser conectados a cada célula de pad. Neste ponto, devemos notar que os nomes dos sinais de conexão do periférico devem ser

todos uniformizados. Com a uniformização desses sinais, podemos utilizar variáveis para compor o nome do sinal, facilitando sinais que aparecem mais que uma vez.

Tabela 11 - Metadado de "*pad*"

Technology				
io90u3v_V2				
Conection Name				
PT				
PAD NAME				
p_io_3v_4m				
MUX POSSIBLE				
TRUE				
Conection				
Pin Name	**Pin Type**	**Connect To**	**Connect To Type**	**Description**
IPP_INA_3V	output	ipp_ina_$NAME	output	
IPP_INA_MUX_3V	output	ipp_ina_mux_3v_$NAME_nc	wire	
IPP_OUTA_MUX_3V	inout	ipp_outa_mux_3v_$NAME_nc	wire	
IPP_OUTA_3V	inout	ipp_outa_$NAME	inout	
PAD	inout	pad_$NAME	inout	
IPP_IND	output	ipp_ind_$NAME	output	
IPP_IND_3V	output	ipp_ind_3v_$NAME_nc	wire	
IPP_IBE	input	ipp_ibe_$NAME	input	Input Buffer Enable
IPP_IFE	input	ipp_ife_$NAME	input	Input Filter Enable
IPP_DO	input	ipp_do_$NAME	input	Data Out
IPP_OBE	input	ipp_obe_$NAME	input	Output Buffer Enable
IPP_ODE	input	ipp_ode_$NAME	input	Open Drain Enable
IPP_DSE	input	ipp_dse_$NAME	input	Drive Strenght Enable
IPP_PUE	input	ipp_pue_$NAME	input	PullUp Enable
IPP_PUS	input	ipp_pus_$NAME	input	PullUp polarity Select
IPP_SRE	input	ipp_sre_$NAME	input	Slew Rate Enable
VBUF_3V	inout	VBUF_3V	wire	
VDD_3V	inout	VDD_3V	wire	
VSS_3V	inout	VSS_3V	wire	
VDD_LV	inout	VDD_LV	wire	

Os campos "*Pin Name"* e "*Pin Type*" identificam o pino do *pad* e sua direção. No lado da conexão, temos o nome ("*Connect to*") e o tipo ("*Connect to Type*") do sinal que será conectado à porta do *pad* na geração do código RTL. O tipo "*wire*" normalmente é usado em sinais da célula de *pad* para outra célula de *pad* (conexão do anel de alimentação entre *pads*). Um campo de descrição é adicionado, o qual será propagado no código RTL gerado para auxiliar na leitura do código.

O segundo metadado é relacionado com a função que o pino pode suportar. Cada função tem um nome único ("*Function Name*") definido. Este nome será usado em uma tabela de abstração maior, para selecionar a correta funcionalidade desejada para cada pino do projeto do microcontrolador (Tabela 12).

Se o projeto necessita de uma conexão diferente de projetos anteriores, um novo metadado precisa ser adicionado ao banco de dados de metadado, refletindo a conexão modificada. Um novo nome único deve ser criado para essa nova função.

O valor para a conexão pode ser um valor fixo (0/1) ou o nome de um sinal interno. Nota-se que alguns nomes são compostos com variáveis e serão substituídos dinamicamente, utilizando-se informações de metadado com abstração maior.

Cada tecnologia tem seu próprio conjunto de funções e os nomes dessas funções correlacionam a mesma funcionalidade entre as tecnologias. O campo de tecnologia não é mostrado na Tabela 12, mas pode-se entender que cada função é um subconjunto do metadado de uma tecnologia específica.

Tabela 12 - Metadado de possível função do *pad*

Function Name	
PORT	
PinName	**PinVal**
ipp_ind_$MOD_$PIN_NAME	ipp_ind_port_PIN_NAMEPIN_NUMBER
ipp_obe_$MOD_$PIN_NAME	ipp_obe_port_PIN_NAMEPIN_NUMBER
ipp_do_$MOD_$PIN_NAME	ipp_do_port_PIN_NAMEPIN_NUMBER
ipp_ode_$MOD_$PIN_NAME	0
ipp_dse_$MOD_$PIN_NAME	ipp_dse_pctl_PIN_NAMEPIN_NUMBER
ipp_pue_$MOD_$PIN_NAME	ipp_pue_pctl_PIN_NAMEPIN_NUMBER
ipp_pus_$MOD_$PIN_NAME	1
ipp_ibe_$MOD_$PIN_NAME	ipp_ibe_port_PIN_NAMEPIN_NUMBER
ipp_hys_$MOD_$PIN_NAME	1
ipp_sre_$MOD_$PIN_NAME	ipp_sre_pctl_PIN_NAMEPIN_NUMBER
ipp_ife_$MOD_$PIN_NAME	1
Function Name	
TPM-CH	
PinName	**PinVal**
ipp_ana_en_$MOD_$PIN_NAME	0
ipp_port_en_$MOD_$PIN_NAME	ipp_port_en_$FUNC_NAME$FUNC_NAME_NUMBER$FUNC_PIN_NAME$FUNC_PIN_NUMBER
ipp_ind_$MOD_$PIN_NAME	ipp_ind_$FUNC_NAME$FUNC_NAME_NUMBER$FUNC_PIN_NAME$FUNC_PIN_NUMBER
ipp_obe_$MOD_$PIN_NAME	ipp_obe_$FUNC_NAME$FUNC_NAME_NUMBER$FUNC_PIN_NAME$FUNC_PIN_NUMBER
ipp_do_$MOD_$PIN_NAME	ipp_do_$FUNC_NAME$FUNC_NAME_NUMBER$FUNC_PIN_NAME$FUNC_PIN_NUMBER
ipp_ode_$MOD_$PIN_NAME	0
ipp_dse_$MOD_$PIN_NAME	ipp_dse_pctl_PIN_NAMEPIN_NUMBER
ipp_pue_$MOD_$PIN_NAME	ipp_pue_pctl_PIN_NAMEPIN_NUMBER
ipp_pus_$MOD_$PIN_NAME	1
ipp_ibe_$MOD_$PIN_NAME	ipp_ibe_$FUNC_NAME$FUNC_NAME_NUMBER$FUNC_PIN_NAME$FUNC_PIN_NUMBER
ipp_hys_$MOD_$PIN_NAME	1
ipp_sre_$MOD_$PIN_NAME	ipp_sre_pctl_PIN_NAMEPIN_NUMBER
ipp_ife_$MOD_$PIN_NAME	1
ipp_offval_$MOD_$PIN_NAME	0

Para se adicionar uma nova função para um novo periférico (funcionalidade), é necessário incluir uma tabela indicando como essa nova funcionalidade configura os sinais de controle do *pad*. Para se exportar uma determinada função para outra tecnologia, basta incluir essa nova tabela de conexão da função na outra tecnologia, mudando o nome dos sinais que devem ser configurados para se obter a funcionalidade desejada.

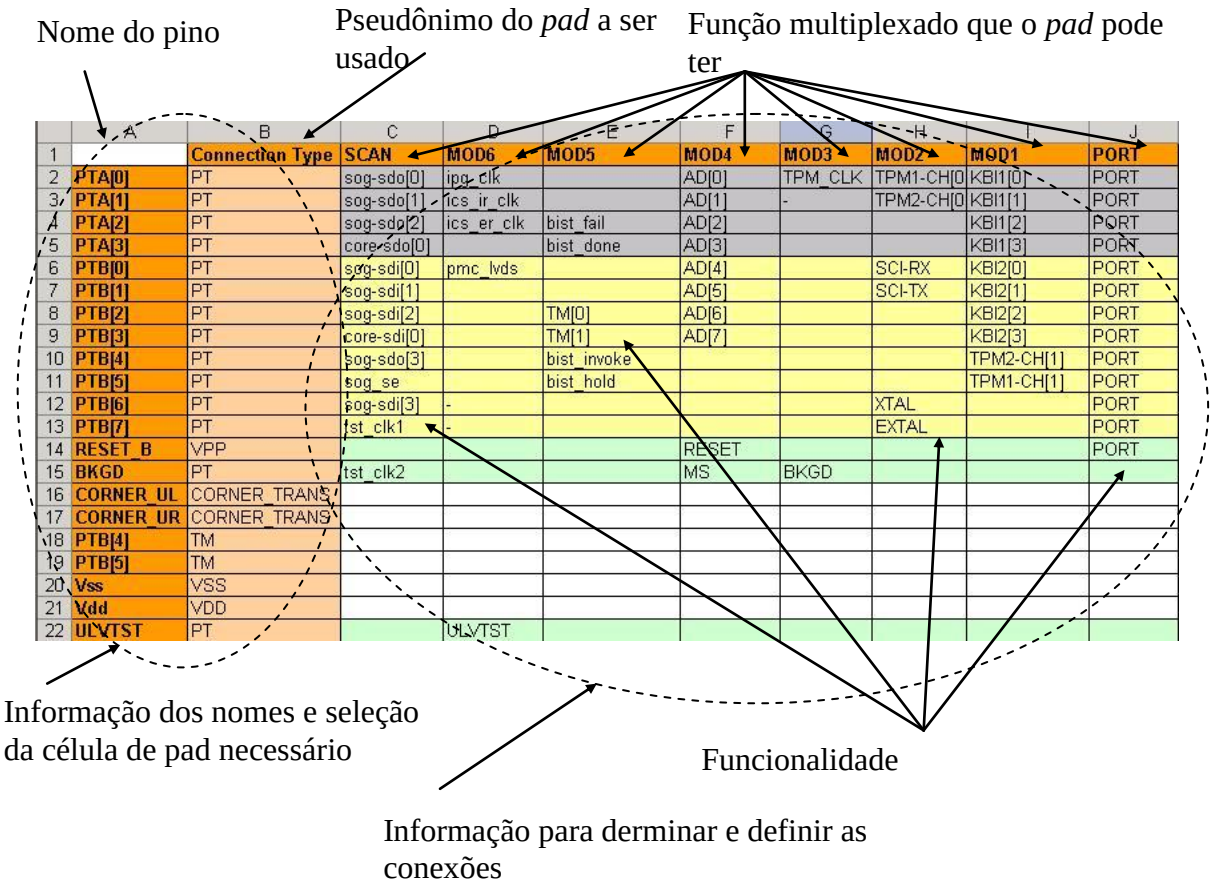
O terceiro e último metadado é o metadado de projeto, que armazena as informações específicas do projeto propriamente dito e pode ser visto na Figura 62.

Na coluna A, adicionamos o nome que cada *pad* terá. Este é o nome que vai ser utilizado na instanciação da célula de *pad* no código RTL. Esse nome é único e pode ser composto com colchetes.

Na coluna B, é selecionado o tipo de conexão e está relacionado com o pseudônimo utilizado no metadado da célula de *pad*. A seleção do nome só pode ser escolhida a partir dos nomes existentes no banco de dados de metadado, isto inibe a possibilidade de tentar utilizar uma célula de *pad* não existente em uma tecnologia específica.

Colunas C a J estão relacionadas com o bloco de multiplexação; pode-se selecionar a função que cada *pad* terá em um específico projeto. Novamente, o nome da função é somente selecionável das possibilidades armazenadas no banco de dados de metadado e também pode ser composto com colchetes.

Figura 62 - Metadado de projeto



	A	B	C	D	E	F	G	H	I	J
1		Connection Type	SCAN	MOD6	MOD5	MOD4	MOD3	MOD2	MOD1	PORT
2	PTA[0]	PT	sog-sdo[0]	ipg_clk		AD[0]	TPM_CLK	TPM1-CH[0	KBI1[0]	PORT
3	PTA[1]	PT	sog-sdo[1]	ics_ir_clk		AD[1]	-	TPM2-CH[0	KBI1[1]	PORT
4	PTA[2]	PT	sog-sdo[2]	ics_er_clk	bist_fail	AD[2]			KBI1[2]	PORT
5	PTA[3]	PT	core-sdo[0]		bist_done	AD[3]			KBI1[3]	PORT
6	PTB[0]	PT	sog-sdi[0]	pmc_lvds		AD[4]		SCI-RX	KBI2[0]	PORT
7	PTB[1]	PT	sog-sdi[1]			AD[5]		SCI-TX	KBI2[1]	PORT
8	PTB[2]	PT	sog-sdi[2]		TM[0]	AD[6]			KBI2[2]	PORT
9	PTB[3]	PT	core-sdi[0]		TM[1]	AD[7]			KBI2[3]	PORT
10	PTB[4]	PT	sog-sdo[3]		bist_invoke				TPM2-CH[1]	PORT
11	PTB[5]	PT	sog_se		bist_hold				TPM1-CH[1]	PORT
12	PTB[6]	PT	sog-sdi[3]	-				XTAL		PORT
13	PTB[7]	PT	tst_clk1	-				EXTAL		PORT
14	RESET_B	VPP				RESET				PORT
15	BKGD	PT	tst_clk2			MS	BKGD			
16	CORNER_UL	CORNER_TRANS								
17	CORNER_UR	CORNER_TRANS								
18	PTB[4]	TM								
19	PTB[5]	TM								
20	Vss	VSS								
21	Vdd	VDD								
22	ULVTST	PT		ULVTST						

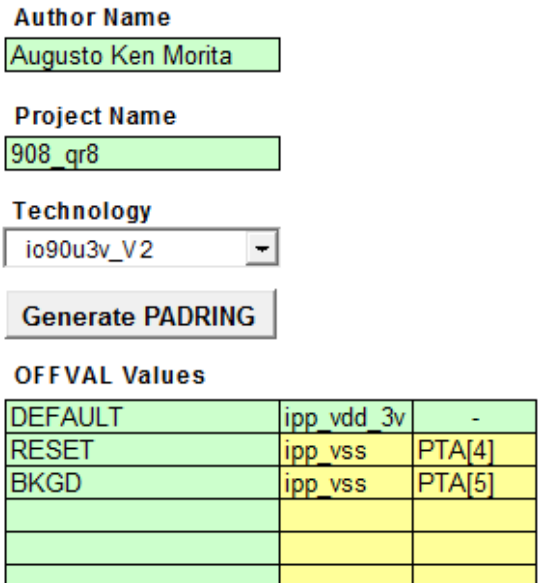
Existem algumas outras informações que devem ser adicionadas para a correta geração do código RTL – estão ilustradas na Figura 63.

O campo "*Author Name*" será utilizado no header do código gerado. O campo "*Technology*" seleciona a tecnologia do conjunto de *pad* a ser usado. O nome do projeto será utilizado na geração dos nomes das instâncias e dos módulos do código RTL gerado.

Usando essa metodologia, podemos facilmente trocar o conjunto de células de *pad* de uma tecnologia para outra. Caso se selecione uma tecnologia na qual uma determinada funcionalidade não tem o correspondente, o campo dessa funcionalidade faltante estará

pintado de vermelho, indicando a necessidade de incluir a funcionalidade faltante no banco de dados de metadado dessa tecnologia.

Figura 63 - Informação de projeto



Outra informação que precisamos incluir é o valor do sinal do *pad* quando este não for utilizado; assim, o bloco de multiplexação pode fornecer o correto valor nos sinais de controle quando o *pad* não estiver sendo utilizado.

4.1.3 Uso e resultados da utilização da metodologia de geração de *padring*

A metodologia descrita nesse trabalho foi utilizada, além de nos projetos realizados na comparação dos diversos processadores, em alguns dispositivos comerciais da Freescale (MC9S08MP16 [47], MC9S08QB8 [48] e MC9S08SC4 [49]). Em todos os projetos utilizados, houve diminuição significativa no tempo gasto para a criação do *padring*. Notamos que, utilizando essa metodologia, a pessoa que gerou o *padring* não necessitou de conhecimentos específicos de como os blocos internos são conectados, aumentando, assim, a abstração da tarefa de criação do código.

A utilização de dados de outros projetos que ficaram armazenados ou importados na base de dados de metadado ajuda bastante na criação de novos projetos. Se a funcionalidade requerida já está dentro da base de dados, não há mais necessidade de criar a funcionalidade; bastando somente selecioná-la na planilha e a conexão será realizada conforme dados do banco de dados.

Durante a importação de projetos anteriores na base de dados, identificamos algumas inconsistências entre projetos que possibilitaram a criação da base de dados mais consistente e de qualidade superior que puderam ser incorporados em novos projetos. E, durante a execução dos projetos, percebemos que a utilização da metodologia facilitava bastante as modificações de especificações relacionadas aos pinos do microcontrolador, acelerando a realização do trabalho.

O metadado de projeto facilita bastante uma visão global de como os pinos estarão organizados em funcionalidade dentro do microcontrolador, possibilitando identificação de inconsistências no nível de sistema antecipadamente.

Apesar de esse trabalho utilizar bastante planilhas em Excel, toda a inteligência e a metodologia foi desenvolvida em programação *Visual Basic* e as planilhas são basicamente um método mais gráfico de interação com o programa propriamente dito. As tabelas de metadado são convertidas em formato XML (*Extensible Markup Language*) e esses dados XML compõem o banco de dados de metadado, o qual é utilizado pelo programa na geração do código RTL.

Um exemplo genérico do código do *padring* gerado por essa metodologia pode ser observado no apêndice B.

4.2 Comparação dos resultados obtidos para os processadores

Neste item, apresentaremos os principais resultados dos processadores utilizados nas comparações com o da microarquitetura nova proposta (S08N).

Temos três modelos para comparar com a abordagem do processador proposto nesse trabalho, que são:

- S08 - síncrono original.
- AS08 - modelo assíncrono baseado na microarquitetura do S08.
- S08L - modelo síncrono com melhorias de microarquitetura.

Todos esses três modelos foram avaliados e comparados, tanto em simulação como também foi realizada a implementação em silício (*test vehicle*), com todas as condições para que pudéssemos ter valores reais do consumo de potência e do desempenho desses processadores.

A Figura 64, Figura 65, Figura 66 e Figura 67 ilustram o layout final de cada um dos processadores. A área do silício foi de 5mm x 5 mm, tamanho padrão de um *slot* do *test vehicle*. Foram criados dois domínios de alimentação, para auxiliar nas medidas de bancada

do consumo de potência do processador; assim, as células do processador ficaram totalmente isoladas. A Figura 64 indica a descrição das regiões do silício.

Figura 64 - Layout final do silício – S08

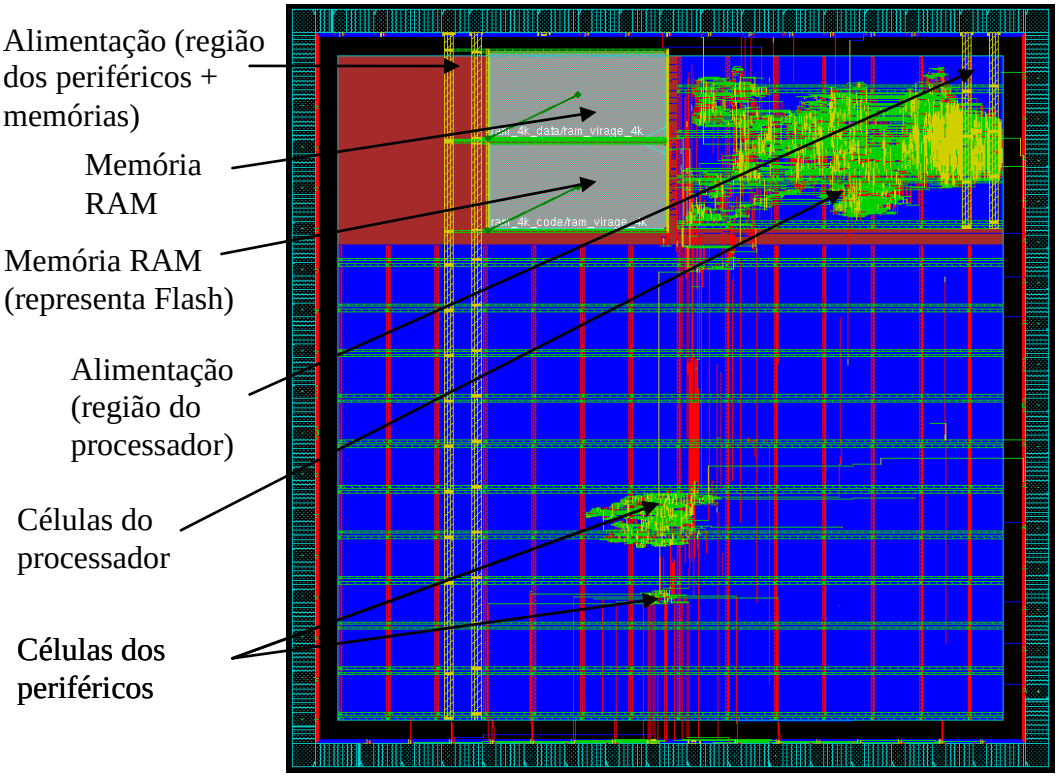


Figura 65 - Layout final do silício – AS08

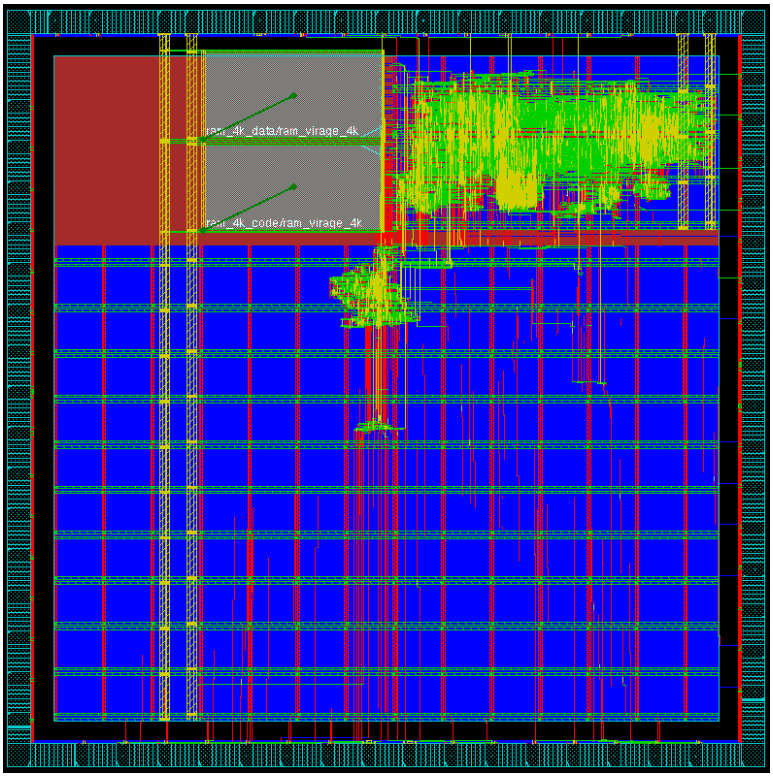


Figura 66 - Layout final do silício – S08L

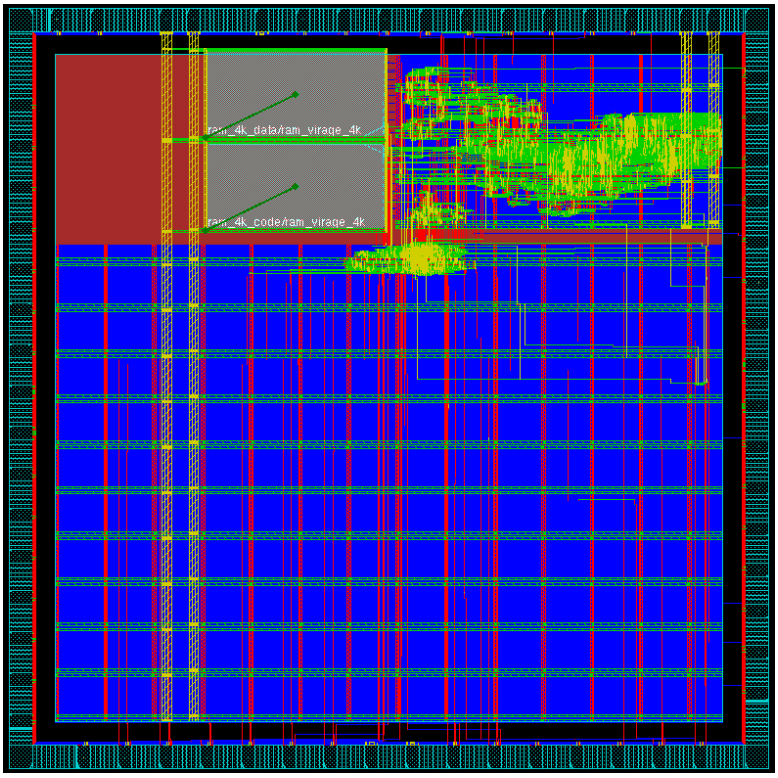
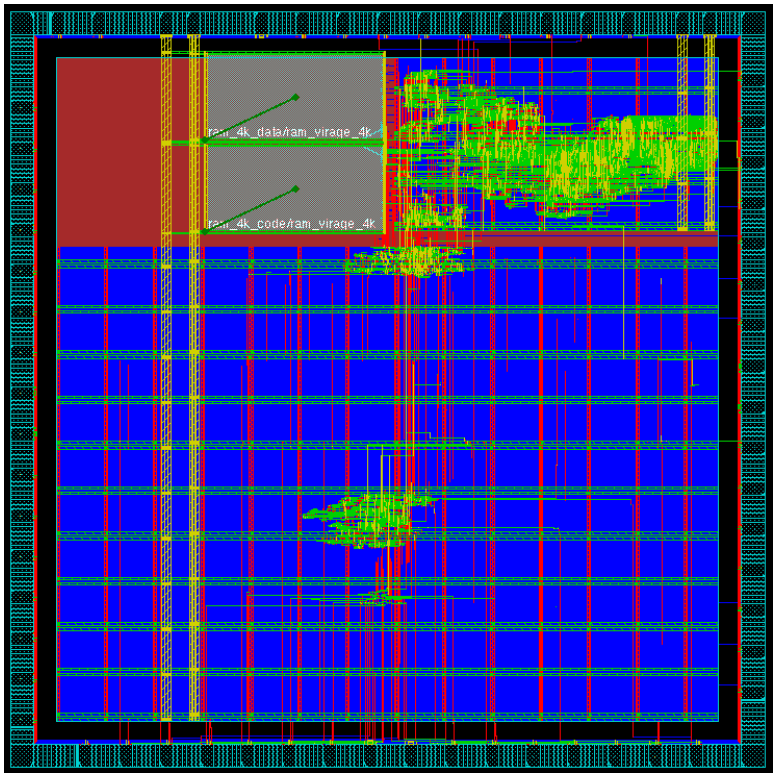


Figura 67 - Layout final do silício – S08N (este trabalho)



A tecnologia utilizada nesses microcontroladores de avaliação foi o CMOS 250nm da TSMC, o tamanho do silício de teste foi 5 mm x 5 mm. Foram utilizadas duas memórias RAM de 4KBytes, uma sendo mapeada como memória RAM e a outra emulando a memória flash para o código do programa. A área do *sea of gates* foi deixada em um tamanho maior para que não houvesse necessidade de o sintetizador fazer minimizações complexas, facilitando a comparação do circuito final mapeado na tecnologia.

Para se medir a potência, foi criado um conjunto de periféricos mínimos para que não interferissem nas medições do processador propriamente dito. Após análise, a Tabela 13 representa os dados de potência obtidos tanto de simulação como medidos em bancada.

A tensão de alimentação (Vdd) considerada nas medidas de potência é de 2,5V, que é o valor da alimentação das células na tecnologia utilizada.

Tabela 13 - Comparação do consumo de potência entre implementações de processadores

Processador	Consumo de potência (simulação)	%	Consumo de potência (medido)	%
S08	2,40 mW	100	1,68 mW	100
AS08	1,56 mW	65	1,14 mW	68
S08L	1,0 mW	40	0,69 mW	41
Este Trabalho S08N	0,87mW	36		

Para as simulações de potência, foi utilizado o "*PowerTheater*", com frequência de barramento de 10MHz. Essa frequência foi utilizada devido à limitação do modelo assíncrono, que não atingia a velocidade de 20MHz esperada no início do projeto.

Os resultados obtidos de medidas do processador em silício mantêm a relação de queda obtida na simulação, como pode ser visto na coluna de porcentagem da Tabela 13. A diferença dos valores obtidos entre simulação e silício está relacionada com a biblioteca utilizada na simulação (*corner case*). A condição de temperatura utilizada na simulação foi de 125° Celsius e as medidas de bancada foi feita em 25° Celsius, acarretando um desvio no resultado.

Para o processador S08N ser comparável com os dados obtidos anteriormente, a frequência foi ajustada para que a tarefa fosse executada no mesmo tempo que os processadores com barramento simples. O valor do aumento de desempenho obtido, devido à modificação das instruções, foi de 18%. Assim, a frequência utilizada na avalição foi de 10MHz – menos 18 %, ou seja, 8,2 MHz.

Não tivemos a oportunidade de fazer o silício do modelo proposto neste trabalho. Contudo, observando os valores obtidos em simulação e comparando com os obtidos em bancada, acreditamos que o valor que iremos obter o caso proposto nesta tese seria próximo em relação ao obtido em simulação.

Em comparação com o processador original S08, o consumo de potência do processador descrito aqui foi 63,75% menor. Em comparação com o melhor processador, ainda temos ganho de 13%.

Tabela 14 - Detalhamento do consumo de potência do processador S08 , com a tensão Vdd de 2,5V

POWER S08					
				Internal Power (Watts)	
Component		**Model**	**Static**	**Dynamic**	**Total**
S08		user	85uW	1,34mW	1,42mW
core_cpu		user	84,3uW	880uW	964uW
	core_soft_st8	user	392nW	482uW	482uW
	sim	user	158nW	164uW	164uW
	flsr	user	1,72uW	132uW	134uW
	ram_512b_hd_st8	user	82uW	43,6uW	126uW
	sim_gated_clk24	clkinv_12	167pW	9,6uW	9,6uW
	sim_gated_clk24	clkbuf_12	199pW	8,1uW	8,1uW
	sim_gated_clk24	clkdly_1	214pW	5,39uW	5,39uW
	sim_gated_clk24	clkbuf_12	199pW	5,11uW	5,11uW
	sim_ipg_clk_gated_flash	clkbuf_12	199pW	4,12uW	4,12uW
	ipbi_908qb8	user	13,7nW	2,7uW	2,71uW
	sim_gated_clk24	clkinv_16	224pW	2,64uW	2,64uW
	sim_gated_clk24	clkinv_12	167pW	2,34uW	2,34uW
	sim_ipg_clk_gated_flash	clkbuf_4	83,5pW	1,87uW	1,87uW
	sim_ipg_clk_gated_flash	clkbuf_4	83,5pW	1,85uW	1,85uW
	sim_ipg_clk_gated_flash	clkbuf_4	83,5pW	1,82uW	1,82uW
ics_ip8		user	48,9nW	66uW	66,1uW
Sci		user	105nW	50,4uW	50,5uW
adc10lv_ip8		user	98,3nW	46,1uW	46,2uW
pmc_ip8		user	31,4nW	28,6uW	28,7uW
Tpm		user	111nW	17,3uW	17,4uW
rtc_ip8		user	49,9nW	16,9uW	17uW
mtim_ip8		user	23,5nW	12uW	12,1uW
port_ptb		user	26nW	8,36uW	8,39uW
irq_ip8		user	8,59nW	8,35uW	8,36uW
port_pta		user	14,8nW	8,14uW	8,15uW
port_ptc		user	18nW	7,97uW	7,99uW
sim_core_clk__L4_I0		clkbuf_12	199pW	7,06uW	7,06uW
ipg_clk__L6_I0		clkinv_16	224pW	6,37uW	6,37uW
Kbi		user	14nW	6,19uW	6,2uW

Sabemos que, em um microcontrolador real, a economia de potência deve ser maior, uma vez que esse processador necessita de frequência menor para realizar a tarefa. Assim, todos os periféricos e circuitos que complementam o SOC devem ter algum tipo de ganho, devido à redução da frequência do relógio. Nas simulações, como consideramos um conjunto

mínimo de periféricos, os dados não são completos, podendo, assim, ter variação no ganho total de potência de consumo, de acordo com a especificação de cada microcontrolador.

Outro ponto a ressaltar é que, com esse novo processador rodando na mesma frequência que outro, haverá também ganho de desempenho do processador mais ou menos na mesma ordem. Isso significa que, com o mesmo consumo de potência, poderíamos ter um processador com desempenho maior. No estudo feito nesse trabalho, obtivemos ganho de desempenho de 18%.

Esse ganho de desempenho e potência deve estar muito relacionado à aplicação propriamente dita, uma vez que, dependendo da aplicação, o desempenho deve mudar; consequentemente, a potência para realizar a tarefa também deve mudar.

Tabela 15 - Detalhamento do consumo de potência do processador S08L, com a tensão Vdd de 2,5V

POWER S08L						
				Internal Power (Watts)		
Component			**Model**	**Static**	**Dynamic**	**Total**
S08L			user	84,8uW	969uW	1,05mW
	core_cpu		user	84,1uW	569uW	653uW
		fs08	user	336nW	198uW	198uW
		flsr	user	1,63uW	141uW	143uW
		ram_512b_hd_st8	user	82uW	48,3uW	130uW
		sim	user	97,6nW	121uW	121uW
		cpu_l08_wrap	user	3,66nW	6,51uW	6,51uW
		ipbi_9l08qb8	user	11,2nW	4,49uW	4,5uW
		sim_ipg_clk_gated_flash	clkbuf_12	199pW	4,15uW	4,15uW
		FE_OFC136_core_ab	buf_s_8	135pW	3,76uW	3,76uW
		sim_gated_clk24	clkbuf_12	199pW	3,64uW	3,64uW
		sim_gated_clk24	clkbuf_12	199pW	3,04uW	3,04uW
		FE_OFC139_core_ab	buf_s_8	135pW	3,02uW	3,02uW
		FE_OFC137_core_ab	buf_12	151pW	2,45uW	2,45uW
		FE_OFC135_core_ab	buf_s_8	135pW	2,09uW	2,09uW
		FE_OFC140_core_ab	buf_8	108pW	1,82uW	1,82uW
		sim_ipg_clk_gated_flash	clkbuf_4	83,5pW	1,73uW	1,73uW
	sci		user	105nW	50,4uW	50,5uW
	adc10lv_ip8		user	97,6nW	47uW	47,1uW
	ics_ip8		user	42,8nW	32,4uW	32,5uW
	pmc_ip8		user	31,8nW	29,5uW	29,5uW
	rtc_ip8		user	49,7nW	20,6uW	20,7uW
	tpm		user	119nW	17,2uW	17,3uW
	mtim_ip8		user	23nW	12,1uW	12,2uW
	irq_ip8		user	8,65nW	8,49uW	8,5uW
	port_ptc		user	12,8nW	8,47uW	8,48uW
	port_ptb		user	12,7nW	8,28uW	8,29uW
	port_pta		user	11,8nW	8,05uW	8,06uW
	kbi		user	14nW	6,3uW	6,32uW
	acmp		user	9,02nW	4,47uW	4,48uW
	ipg_clk		clkbuf_12	199pW	4,17uW	4,17uW

Existem alguns trabalhos que abordam consumos de potência relacionados ao comportamento de barramento [50]. Cada um desses trabalhos aborda uma determinada

característica, como transições das linhas, tamanho da carga das linhas do barramento e como eles afetam no consumo de potência, *performance* e outros.

A Tabela 14, Tabela 15 e a Tabela 16 mostram os valores de consumo de potência de cada um dos processadores avaliados e obtidos pela simulação de forma mais detalhada.

Tabela 16 Detalhamento do consumo de potência do processador S08N, com a tensão Vdd de 2,5V

POWER NEW					
				Internal Power (Watts)	
Component		**Model**	**Static**	**Dynamic**	**Total**
S08N		user	82,26uW	794,58uW	876,84uW
core_cpu		user	81,58uW	472,27uW	553,85uW
	cpu	user	329,28nW	168,3uW	168,63uW
	flsr	user	1,6uW	122,67uW	124,27uW
	ram_512b_hd_st8	user	78,72uW	38,64uW	117,36uW
	sim	user	92,72nW	114,95uW	115,04uW
	cpu_new	user	3,55nW	5,21uW	5,21uW
	ipbi_9l08qb8	user	11,09nW	3,59uW	3,6uW
	sim_ipg_clk_gated_flash	clkbuf_12	189,05pW	3,94uW	3,94uW
	FE_OFC136_core_ab	buf_s_8	108pW	3,01uW	3,01uW
	sim_gated_clk24	clkbuf_12	167,16pW	3,09uW	3,09uW
	sim_gated_clk24	clkbuf_12	165,17pW	2,61uW	2,61uW
	FE_OFC127_core_ab	buf_4	129,6pW	2,51uW	2,51uW
	FE_OFC134_core_ab	buf_12	147,98pW	2,23uW	2,23uW
	FE_OFC111_core_ab	buf_s_8	130,95pW	1,84uW	1,84uW
	FE_OFC138_core_ab	buf_8	105,84pW	1,78uW	1,78uW
	sim_ipg_clk_gated_flash	clkbuf_4	82,67pW	1,56uW	1,56uW
Sci		user	100,8nW	43,85uW	43,95uW
adc10lv_ip8		user	87,84nW	41,83uW	41,92uW
ics_ip8		user	36,38nW	25,92uW	25,96uW
pmc_ip8		user	31,16nW	27,14uW	27,17uW
rtc_ip8		user	46,22nW	17,51uW	17,56uW
Tpm		user	105,91nW	15,14uW	15,24uW
mtim_ip8		user	22,54nW	11,5uW	11,52uW
irq_ip8		user	8,3nW	8,32uW	8,33uW
port_ptc		user	12,67nW	8,39uW	8,4uW
port_ptb		user	12,45nW	8,11uW	8,13uW
port_pta		user	11,45nW	7,81uW	7,82uW
Kbi		user	11,2nW	5,04uW	5,05uW
Acmp		user	7,58nW	3,75uW	3,76uW
ipg_clk		clkbuf_12	195,02pW	3,59uW	3,59uW

Quanto à área dos vários modelos analisados, a Tabela **17**, a Tabela 18 e a Tabela 19 apresentam os valores obtidos. Na versão deste trabalho, a área total teve pequeno aumento em relação ao S08L, que deve-se ao aumento da complexidade para lidar com os novos barramentos incluídos nessa abordagem. Esse aumento de área deve ser o responsável pela diminuição somente de 13% na potência – quando comparado com o aumento de desempenho de 18%. O importante desses dados é que a inclusão de circuitos de controle e novos

barramentos não acarretou em significativo aumento de área. Essa é a principal razão para termos ganho na diminuição do consumo de potência.

Tabela 17 - Análise da área do modelo original S08

S08		Structural				
	Cells	Cell Area Total	SOG Cell Area	Hardblock Area	Net Area	Total Area
mcu_9s08	12004	2572224	588573,44	1983650,93	291648	2863872
core_soft_st8	3644	148095			78537	226632
core_soft_st8 %	30.36%	5,76%	25,16%		26,93%	7,91%

S08	QB8 Postlayout		
	Cell Area Total	SOG Cell Area	Hardblock Area
mcu_9s08	2895355,27	785458,24	2109897,03
core_soft_st8	164241.30		
	5,67%	20,91%	

Tabela 18 - Análise da área do modelo S08L

S08L		Structural				
	Cells	Cell Area Total	SOG Cell Area	Hardblock Area	Net Area	Total Area
mcu_9l08	11656	2537016	561939,84	1975075,73	292103	2829119
fs08	3144	122192			72099	194291
Fs08 %	26,97%	4,82%	21,74%		24,68%	6,87%

S08L	QB8L Postlayout		
	Cell Area Total	SOG Cell Area	Hardblock Area
mcu_9l08	2833150,47	723253,44	2109897,03
fs08	135434.9		
	4,78%	18,73%	

Tabela 19 - Análise da área do modelo proposto nesse trabalho

Arquitetura proposta		Structural				
	Cells	Cell Area Total	SOG Cell Area	Hardblock Area	Net Area	Total Area
mcu_9n08	11656	2541097	566020,84	1976575,63	292241	2833338
Cpu	3231	126273			73032	199305
NEW %	27,72%	4,97%	22,31%		24,99%	7,03%

Capítulo - 5. Conclusões e considerações finais

Dispositivos de baixo consumo de potência têm estado em evidência nos dias atuais, devido à grande demanda de dispositivos portáteis com alta integração de circuitos que utilizam baterias – as quais têm limitações severas na capacidade. Ademais, devido às limitações das tecnologias de resfriamento, faz se necessário um trabalho cada vez maior no planejamento do consumo de potência destes novos dispositivos.

Existe a possibilidade de se atuar em todos os níveis do processo de desenvolvimento de circuitos, desde níveis físicos, arquitetura, algoritmo, de sistema e otimizações, sendo que a atuação em cada região tem suas vantagens e desvantagens. A escolha de atuar no nível de microarquitetura foi baseada em um estudo prévio de outras arquiteturas, como a de tecnologia assíncrona e síncrona, com redução de registradores, que indicaram a possibilidade de melhorias maiores – ainda atuando no nível de arquitetura.

Os resultados obtidos por simulação atingiram aumento no desempenho de 18% no tempo; assim, podemos operar o sistema para que a frequência de relógio fosse menor; ou seja, executando o mesmo padrão de teste no mesmo tempo dos outros modelos comparados e, com isso, obtendo economia de consumo de potência de 13%.

O maior aumento de desempenho foi devido à diminuição do número de ciclos de máquina de algumas instruções que acessam periféricos, como memória e registros internos conjugados ao processador. Outra melhora foi a criação de controlador, que foi minimizado desde a concepção, controlando, assim, estruturas básicas que normalmente são deixadas a resolver pelo sintetizador – o qual não atinge a melhor solução. Um dos problemas dessa solução é que o tempo de execução do programa acaba não sendo mais determinístico, ficando dependente de como a aplicação acessa os barramentos internos.

Existe o ganho devido à diminuição da energia dissipada nos acessos do barramento, uma vez que os caminhos das linhas que compõem o barramento são menores e também de menor capacitância. Contudo, como foi utilizado um conjunto mínimo de periféricos, os valores obtidos com esse ganho são mínimos.

Este trabalho foi feito em um processador muito simples, para fácil entendimento e que fosse suficiente para obter dados de ganho ou não para esse tipo de abordagem. Não adiantaria ter escolhido um processador moderno de múltiplos núcleos, com acessos a memória *cache* complexos e muito outras características. O interessante desse trabalho foi demonstrar que há ganho quando se utilizam múltiplos barramentos para acesso, ajustando o *timing* de leitura e execução de instruções.

Apesar de muitos acharem que não existe mais espaço para melhoria de arquitetura, este trabalho demonstra existir características do sistema que podem ser exploradas, de forma a obter ganhos de desempenho que acarretem na diminuição de consumo de potência.

Durante o trabalho, percebemos que um dos grandes motivos para se explorar pouco as melhorias de microarquitetura é devido à grande complexidade envolvida no desenvolvimento de microprocessador. Uma extensa pesquisa de estruturas e a grande experiência em desenvolvimento de circuitos são necessárias para realizar análise e escolher as melhores estruturas. Não haver muitos dados para comparação é o outro fator que dificulta pesquisas dessas novas soluções e melhorias na arquitetura.

A criação de metodologia de geração de código RTL para *padring* de microprocessador foi interessante e economizou muito tempo na geração e modificação dos códigos usados na integração de todos os processadores protótipos desenvolvidos para este trabalho. A padronização e a geração automática de código para *padring* contribuem para diminuir erros que acarretam consumo de potência desnecessário, devido à configuração errada das interconexões dos sinais com os periféricos.

Esse mesmo método também foi utilizado em outros microcontroladores desenvolvidos na empresa colaboradora desse trabalho, que se tornaram comerciais e que fazem parte dos dispositivos à venda atualmente.

A possibilidade de utilizar informações de projetos anteriores para a geração do código de *padring* facilita bastante a abstração da complexidade das interconexões que envolvem a criação de um microcontrolador.

Considerações finais

A microarquitetura final desse trabalho foi desenvolvida de forma a deixar espaço para um futuro estudo de *pipeline* e outras técnicas relacionadas que possam levar a aumento de desempenho – o que talvez possa se traduzir como diminuição do consumo de potência ainda maior.

Outro aspecto que poderia ser abordado é a tentativa de aplicar esse tipo de abordagens em projetos com processadores de 32bits e tentar fazer correlação da diminuição do consumo de potência. Múltiplos núcleos podem ser explorados nessa técnica, permitindo acesso a múltiplos barramentos independentes e possibilitando aumento no desempenho que possivelmente possa se refletir na diminuição do consumo de potência.

Quanto à metodologia de geração automática de código para *padring*, esta poderia ser estendida para microcontroladores de 32 bits, os quais têm, atualmente, metodologia diferente da utilizada nos microcontroladores de 8 bits.

REFERÊNCIAS

[1] Pop E. Energy dissipation and transport in nanoscale devices. Nano, Res, 2010, 3: 147–169

[2] Blair, G.M., "Designing low-power digital CMOS," Electronics & Communication Engineering Journal , vol.6, no.5, pp.229,236, Oct 1994

[3] Intel Corporation, ``Microprocessor Hall of Fame", 2004, http://www.intel.com.

[4] Semiconductor Industry Association, ``International Technology Roadmap for Semiconductors - 2004 Update", 2004, http://public.itrs.net.

[5] Semiconductor Industry Association, ``International Technology Roadmap for Semiconductors - 2011 Update", 2011, http://public.itrs.net.

[6] CIA World Factbook 2009 (https://www.cia.gov/library/publications/download/)

[7] IEA Statistics and Balance (http://www.iea.org/stats/index.asp)

[8] U.S Energy Information Administration, 2010

[9] M. Pedram and J. Rabaey, "Power Aware Design Methodologies". Norwell, MA: Kluwer, 2002

[10] Bellaur A. and Elmasry M. I., "Low Power Digital CMOS Design: Circuits and Systems. Norwell, MA: Kluwer," ch. 4, pp. 135–137, 1996

[11] Vemuru, Srinivasa R; Scheinberg, N., "Short-circuit power dissipation estimation for CMOS logic gates," Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on , vol.41, no.11, pp.762,765, Nov 1994

[12] Kim, C.H.; Roy, K., "Dynamic V_{TH} scaling scheme for active leakage power reduction," *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings* , vol., no., pp.163,167, 2002

[13] Hamann, H.F.; Weger, A.; Lacey, J.A.; Zhigang Hu; Bose, P.; Cohen, E.; Wakil, J., "Hotspot-Limited Microprocessors: Direct Temperature and Power Distribution Measurements," Solid-State Circuits, IEEE Journal of , vol.42, no.1, pp.56,65, Jan. 2007

[14] Khoa Dac Tran; Van Nguyen, P.; Hoa Tan Lu; Cuong Phuc Phan; Quang Hai Phan; Kudo, H.; Masuda, H.; Negishi, S.; Yamamoto, M.; Hirose, K.; Okamoto, Y., "A low-power processor for portable navigation devices: 456 mW at 400 MHz and 24 mW in software standby mode," Solid-State Circuits Conference, 2008. A-SSCC '08. IEEE Asian , vol., no., pp.197,200, 3-5 Nov. 2008

[15] Hattori, T., "Challenges for Low-power Embedded SOC's," VLSI Design, Automation and Test, 2007. VLSI-DAT 2007. International Symposium on , vol., no., pp.1,4, 25-27 April 2007

[16] Osborne, S.; Erdogan, A.T.; Arslan, T.; Robinson, D., "Bus encoding architecture for low-power implementation of an AMBA-based SoC platform," Computers and Digital Techniques, IEE Proceedings - , vol.149, no.4, pp.152,156, Jul 2002

[17] Yamada, Tetsuy.; Abe, M.; Nitta, Y.; Ogura, K.; Kusaoke, M.; Ishikawa, M.; Ozawa, M.; Takada, K.; Arakawa, F.; Nishii, O.; Hattori, T., "Low-power design of 90-nm SuperH™ processor core," Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on , vol., no., pp.258,263, 2-5 Oct. 2005

[18] Lanuzza, M.; Margala, M.; Corsonello, P., "Cost-effective low-power processor-in-memory-based reconfigurable datapath for multimedia applications," Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on , vol., no., pp.161,166, 8-10 Aug. 2005

[19] Moyer, B., "Low-power design for embedded processors," Proceedings of the IEEE , vol.89, no.11, pp.1576,1587, Nov 2001

[20] Furber, S.B.; Day, P.; Garside, J.D.; Paver, N.C.; Temple, S.; Woods, J.V., "The design and evaluation of an asynchronous microprocessor," Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on , vol., no., pp.217,220, 10-12 Oct 1994

[21] Hossain, R.; Menghui Zheng; Albicki, A., "Reducing power dissipation in CMOS circuits by signal probability based transistor reordering," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.15, no.3, pp.361,368, Mar 1996

[22] Borah, Manjit; Owens, R.M.; Irwin, M.J., "Transistor sizing for low power CMOS circuits," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.15, no.6, pp.665,671, Jun 1996

[23] Schoellkopf, J.-P.; Magarshack, P., "Low-Power Design Solutions for Wireless Multimedia SoCs," Design & Test of Computers, IEEE , vol.26, no.2, pp.20,29, March-April 2009

[24] Wen-Zen Shen,; Jiing-Yuan Lin; Fong-Wen Wang, "Transistor reordering rules for power reduction in CMOS gates," Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95, IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal , vol., no., pp.1,6, 29 Aug-1 Sep 1995

[25] Rabaey Jan M., "Digital Integrated Circuits A Design Perspective," Upper Saddle River, N.J - Pearson Education - 2003

[26] Suzuki, K.; Mita, S.; Fujita, T.; Yamane, F.; Sano, F.; Chiba, A.; Watanabe, Y.; Matsuda, K.; Maeda, T.; Kuroda, T., "A 300 MIPS/W RISC core processor with variable supply-voltage scheme in variable threshold-voltage CMOS," Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997 , vol., no., pp.587,590, 5-8 May 1997

[27] Gonzalez, R.; Gordon, B.M.; Horowitz, M.A., "Supply and threshold voltage scaling for low power CMOS," Solid-State Circuits, IEEE Journal of , vol.32, no.8, pp.1210,1216, Aug 1997

[28] Blem, E.; Menon, J.; Sankaralingam, K., "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures," High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on , vol., no., pp.1,12, 23-27 Feb. 2013

[29] John L. Hennessy, David A. Patterson, "Computer Architecture A QuantitativeApproach", Fifth Edition, 2011

[30] Tolley, D.B., "Analysis of CISC versus RISC microprocessors for FDDI network interfaces," Local Computer Networks, 1991. Proceedings., 16th Conference on , vol., no., pp.485,493, 14-17 Oct 1991

[31] Finlayson, I.; Gang-Ryung Uh; Whalley, D.; Tyson, G., "Improving Low Power Processor Efficiency with Static Pipelining," Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on , vol., no., pp.17,24, 12-12 Feb. 2011

[32] Yoshida, Y.; Bao-Yu Song; Okuhata, H.; Onoye, T.; Shirakawa, I., "Low-power consumption architecture for embedded processor," ASIC, 1996., 2nd International Conference on , vol., no., pp.77,80, 21-24 Oct 1996

[33] Yamada T., et al., "A Low-Power Embedded RISC special bus or power off other modules. Microprocessor with an Integrated DSP for Mobile Applications." IEICE Transactions on Electronics, Vol.E85-C,pp 253-262, Feb 2002

[34] Bate, I.; Reutemann, R., "Efficient integration of bimodal branch prediction and pipeline analysis," Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on , vol., no., pp.39,44, 17-19 Aug. 2005

[35] Hongkyun Jung; Hyoungjun Kim; Kwangmyoung Kang; Kwangki Ryoo, "Performance Improvement and Low Power Design of Embedded Processor," Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on , vol.2, no., pp.140,145, 11-13 Nov. 2008

[36] Hiraki, M.; Bajwa, R.S.; Kojima, H.; Gorny, D.J.; Nitta, K.; Shri, A., "Stage-skip pipeline: a low power processor architecture using a decoded instruction buffer," Low Power Electronics and Design, 1996., International Symposium on , vol., no., pp.353,358, 12-14 Aug 1996

[37] Tse-Yu Yeh, "Low-Power, High-Performance Architecture of the PWRficient Processor Family," Micro, IEEE , vol.27, no.2, pp.69,78, March-April 2007

[38] Maroju SaiKumar1, Dr. Samundiswary P., "Design and Performance Analysis of Various Adders using Verilog", IJCSMC, Vol. 2, Issue. 9, pg.128 – 138, Sep. 2013

[39] ng Boon Chong; Ho Kah Chun, "Unified Padring Design Flow," Computational Intelligence, Communication Systems and Networks (CICSyN), 2013 Fifth International Conference on , vol., no., pp.418,423, 5-7 June 2013

[40] Ang Boon Chong; Ho Kah Chun, "Unified Padring Design Flow - New Developments and Results," Artificial Intelligence, Modelling and Simulation (AIMS), 2013 1st International Conference on , vol., no., pp.462,466, 3-5 Dec. 2013

[41] http://www.defactotech.com/star-rtl-build-signoff/star-padring

[42] Ming-Fang Lai; Hung-Ming Chen, "An Implementation of Performance-Driven Block and I/O Placement for Chip-Package Codesign," Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on , vol., no., pp.604,607, 17-19 March 2008

[43] Alcom C., Dworak D., Haddad N., Henley W., and Nixon P., "Kerf Test Structure Designs for Process and Device Characterisation," Solid State Technology, pp. 229-235, May 1985

[44] Ward, D.; Walton, A.J.; Gammie, W.G.; Holwill, R.J., "The use of a digital multiplexer to reduce process control chip pad count," Microelectronic Test Structures, 1992. ICMTS 1992. Proceedings of the 1992 International Conference on , vol., no., pp.129,133, 16-19 Mar 1992

[45] Nishimura A. et al., "Multiplexed test structure: A novel VLSI technology development tool," in Proc. IEEE VLSI Workshop Test Structures, pp. 17–18, Feb. 1986

[46] http://www.ispd.cc/slides/slides10/2_04.pdf

[47] http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08MP16DS.pdf

[48] http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08QB8.pdf

[49] http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08SC4.pdf

[50] Ozer, E.; Sendag, R.; Gregg, D., "Multiple-valued logic buses for reducing bus energy in low-power systems," Computers and Digital Techniques, IEE Proceedings - , vol.153, no.4, pp.270,282, 3 July 2006

[51] Morita A. K.; Noije W. V., " Multiple bus low power processor design," XXI Iberchip Workshop – IWS'2015, - , Feb 2015

APÊNDICE A - Algumas estruturas de código HDL

```
// ------------------------------------------------------------------------------
// AUTHOR         : Augusto Ken Morita
// AUTHOR'S EMAIL : augustkm@gmail.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// Version  Date         Author  Description
// 1.0     2014-01-28 Augusto Morita Initial version
// ------------------------------------------------------------------------------

module cla_adder4 (
    // input
    a,
    b,
    c_in,
    // output
    c_out,
    over,
    s
);
// ------------------------------------------------------------------------------
// INPUT DECLARATION
// ------------------------------------------------------------------------------
input [3:0] a;    // adder input a(x)
input [3:0] b;    // adder input b(x)
input       c_in; // Carry In input

// ------------------------------------------------------------------------------
// OUTPUT DECLARATION
// ------------------------------------------------------------------------------
output       c_out; // carry out
output       over;  // overflow
output [3:0] s;     // sum result

// ------------------------------------------------------------------------------
// WIRE DECLARATION (INPUT)
// ------------------------------------------------------------------------------
wire [3:0] a;
wire [3:0] b;
wire       c_in;

// ------------------------------------------------------------------------------
// WIRE/REG DECLARATION (OUTPUT)
// ------------------------------------------------------------------------------
wire       c_out;
wire       over;
wire [3:0] s;

// ------------------------------------------------------------------------------
// INTERNAL WIRE DECLARATION
// ------------------------------------------------------------------------------
wire [4:0] carry; // carry signal
wire [3:0] g;     // generate signal a && B
wire [3:0] p;     // propagate signal a || B

// ------------------------------------------------------------------------------
// 4 x full adder
// ------------------------------------------------------------------------------
full_adder add0 (
   //input
   .a     (a[0]),     // adder input a
   .b     (b[0]),     // adder input b
   .c_in  (carry[0]), // Carry In input
    //output
   .c_out (),         // carry out
```

```
   .g     (g[0]),     // generate signal a && B
   .p     (p[0]),     // propagate signal a ^ B
   .s     (s[0])      // sum
);

full_adder add1 (
   //input
   .a     (a[1]),     // adder input a
   .b     (b[1]),     // adder input b
   .c_in  (carry[1]), // Carry In input
    //output
   .c_out (),         // carry out
   .g     (g[1]),     // generate signal a && B
   .p     (p[1]),     // propagate signal a ^ B
   .s     (s[1])      // sum
);

full_adder add2 (
   //input
   .a     (a[2]),     // adder input a
   .b     (b[2]),     // adder input b
   .c_in  (carry[2]), // Carry In input
    //output
   .c_out (),         // carry out
   .g     (g[2]),     // generate signal a && B
   .p     (p[2]),     // propagate signal a ^ B
   .s     (s[2])      // sum
);

full_adder add3 (
   //input
   .a     (a[3]),     // adder input a
   .b     (b[3]),     // adder input b
   .c_in  (carry[3]), // Carry In input
    //output
   .c_out (),         // carry out
   .g     (g[3]),     // generate signal a && B
   .p     (p[3]),     // propagate signal a ^ B
   .s     (s[3])      // sum
);
// ------------------------------------------------------------------------------
// carry generation logic
// ------------------------------------------------------------------------------
   assign carry [0] = c_in;

   assign carry [1] = g[0] | (p[0] & carry[0]);

   assign carry [2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & carry[0]);

   assign carry [3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0])
                           | (p[2] & p[1] & p[0] & carry[0]);

   assign carry [4] = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1])
                     | (p[3] & p[2] & p[1] & g[0])
                     | (p[3] & p[2] & p[1] & p[0] & carry[0]);

// ------------------------------------------------------------------------------
// cla adder carry out assign
// ------------------------------------------------------------------------------
   assign c_out = carry[4];

// ------------------------------------------------------------------------------
// cla adder carry out assign
// ------------------------------------------------------------------------------
   assign over = carry[4] ^ carry[3];
endmodule // cla_adder4
// ------------------------------------------------------------------------------

// AUTHOR         : Augusto Ken Morita
```

```
// AUTHOR'S EMAIL : augustkm@gmail.com
// -------------------------------------------------------------------------
// RELEASE HISTORY
// Version  Date         Author  Description
// 1.0     2014-01-28 Augusto Morita Initial version
// -------------------------------------------------------------------------

module alu (
    // input
    alu_a_in,
    alu_b_in,
    alu_c_in,
    alu_oper,
    // output
    alu_flag_carry,
    alu_flag_half_carry,
    alu_flag_over,
    alu_result
);

// -------------------------------------------------------------------------
// DEFINE DECLARATION
// -------------------------------------------------------------------------
`define SUB_AB 3'b101
`define SUB_BA 3'b110

// -------------------------------------------------------------------------
// INPUT DECLARATION
// -------------------------------------------------------------------------
input [7:0] alu_a_in; // input data a
input [7:0] alu_b_in; // input data b
input       alu_c_in; // Carry In input
input [2:0] alu_oper; // alu operation select

// -------------------------------------------------------------------------
// OUTPUT DECLARATION
// -------------------------------------------------------------------------
output       alu_flag_carry;      // carry out
output       alu_flag_half_carry; // contitional flag half carry
output       alu_flag_over;       // conditional flag overflow
output [7:0] alu_result;          // result

// -------------------------------------------------------------------------
// WIRE DECLARATION (INPUT)
// -------------------------------------------------------------------------
wire [7:0] alu_a_in;
```

```
wire [7:0] alu_b_in;
wire       alu_c_in;
wire [2:0] alu_oper;

// ----------------------------------------------------------------------------
// WIRE/REG DECLARATION (OUTPUT)
// ----------------------------------------------------------------------------
wire       alu_flag_carry;
wire       alu_flag_half_carry;
wire       alu_flag_over;
reg  [7:0] alu_result;

// ----------------------------------------------------------------------------
// INTERNAL WIRE DECLARATION
// ----------------------------------------------------------------------------
wire       a_minus_b;  // signal indication a minus b operation
wire       add_c_out0; // carry out of carry lookahead adder 0
wire [7:0] add_out;    // carry lookahead adder output
wire [7:0] adder_a;    // adder input signal after inverter used for sub operation
wire [7:0] adder_b;    // adder input signal after inverter used for sub operation
wire       adder_c_in; // adder input signal after inverter used for sub operation
wire       b_minus_a;  // signal indication b minus a operation
wire [7:0] zero_8bit;  // internal 8 bit zero signal {0000_0000}


// ----------------------------------------------------------------------------
// internal signal asign
// ----------------------------------------------------------------------------
assign zero_8bit = 8'b0;
assign b_minus_a = (alu_oper == `SUB_BA);
assign a_minus_b = (alu_oper == `SUB_AB);

// ----------------------------------------------------------------------------
// adder input logic
// ----------------------------------------------------------------------------
assign adder_a = alu_a_in ^ {8{b_minus_a}};
assign adder_b = alu_b_in ^ {8{a_minus_b}};
assign adder_c_in = (a_minus_b || b_minus_a) ?  1'b1 : alu_c_in;

// ----------------------------------------------------------------------------
// 4x2 bit lookahead adder
// ----------------------------------------------------------------------------
cla_adder4 cla_adder4_0 (
   //input
   .a     (adder_a[3:0]), // adder input a(x)
   .b     (adder_b[3:0]), // adder input b(x)
```

```
   .c_in  (adder_c_in),   // Carry In input
    //output
   .c_out (add_c_out0),   // carry out
   .over  (),             // overflow
   .s     (add_out[3:0])  // sum result
);

cla_adder4 cla_adder4_1 (
   //input
   .a     (adder_a[7:4]),   // adder input a(x)
   .b     (adder_b[7:4]),   // adder input b(x)
   .c_in  (add_c_out0),     // Carry In input
    //output
   .c_out (alu_flag_carry), // carry out
   .over  (alu_flag_over),  // overflow
   .s     (add_out[7:4])    // sum result
);

// ----------------------------------------------------------------------------
// conditiona flag generation
// ----------------------------------------------------------------------------
assign alu_flag_half_carry = add_c_out0;
//assign alu_flag_zero = !(|alu_result);
//assign alu_flag_sign = alu_result[7];

// ----------------------------------------------------------------------------
// output mux funcion select (alu_oper)
// ----------------------------------------------------------------------------
always @(alu_a_in or alu_b_in or add_out or alu_oper)
begin
  case(alu_oper)
    3'b000 : alu_result = ~alu_b_in;      // invert b
    3'b001 : alu_result = alu_a_in & alu_b_in;  // and
    3'b010 : alu_result = alu_a_in | alu_b_in;  // or
    3'b011 : alu_result = alu_a_in ^ alu_b_in;  //xor
    3'b100 : alu_result = add_out; // add a+b
    3'b101 : alu_result = add_out; // sub a-b
    3'b110 : alu_result = add_out; // sub b-a
    3'b111 : alu_result = alu_a_in;
    default : alu_result = zero_8bit;
  endcase
end

// ----------------------------------------------------------------------------
// UNDEFINE DECLARATION
// ----------------------------------------------------------------------------
```

```
`undef SUB_AB
`undef SUB_BA

endmodule // alu
```

```
// ------------------------------------------------------------------------------
// AUTHOR         : Augusto Ken Morita
// AUTHOR'S EMAIL : augustkm@gmail.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// Version  Date         Author  Description
// 1.0     2014-01-28 Augusto Morita Initial version
// ------------------------------------------------------------------------------

module ralu_ccr (
    // input
    alu_flag_carry,
    alu_flag_half_carry,
    alu_flag_over,
    clk,
    ctrl_alu_flag_over_mask0,
    ctrl_ralu_ccr_carry_tmp_wr_en,
    ctrl_ralu_ccr_carry_wr_en,
    ctrl_ralu_ccr_half_carry_wr_en,
    ctrl_ralu_ccr_over_wr_en,
    ctrl_ralu_ccr_sign_wr_en,
    ctrl_ralu_ccr_zero_wr_en,
    ralu_out,
    reset_b,
    ralu_shifter_a_lsb_out,
    ralu_shifter_a_msb_out,
    ctrl_ralu_ccr_carry_sel,
    // output
    ralu_ccr_carry_reg,
    ralu_ccr_carry_tmp_reg,
    ralu_ccr_half_carry_reg,
    ralu_ccr_over_reg,
    ralu_ccr_sign_reg,
    ralu_ccr_zero_reg
);

// ------------------------------------------------------------------------------
// INPUT DECLARATION
// ------------------------------------------------------------------------------
input       alu_flag_carry;
input       alu_flag_half_carry;
input       alu_flag_over;
input       clk;                            // module clock
input       ctrl_alu_flag_over_mask0;
input       ctrl_ralu_ccr_carry_tmp_wr_en;
input       ctrl_ralu_ccr_carry_wr_en;
```

```
input       ctrl_ralu_ccr_half_carry_wr_en;
input       ctrl_ralu_ccr_over_wr_en;
input       ctrl_ralu_ccr_sign_wr_en;
input       ctrl_ralu_ccr_zero_wr_en;
input [7:0] ralu_out;
input       reset_b;                        // module reset
input ralu_shifter_a_lsb_out;
input ralu_shifter_a_msb_out;
input [1:0] ctrl_ralu_ccr_carry_sel;

// ------------------------------------------------------------------------
// OUTPUT DECLARATION
// ------------------------------------------------------------------------
output  ralu_ccr_carry_reg;
output  ralu_ccr_carry_tmp_reg;
output  ralu_ccr_half_carry_reg;
output  ralu_ccr_over_reg;
output  ralu_ccr_sign_reg;
output  ralu_ccr_zero_reg;

// ------------------------------------------------------------------------
// WIRE DECLARATION (INPUT)
// ------------------------------------------------------------------------
wire       alu_flag_carry;
wire       alu_flag_half_carry;
wire       alu_flag_over;
wire       clk;
wire       ctrl_alu_flag_over_mask0;
wire       ctrl_ralu_ccr_carry_tmp_wr_en;
wire       ctrl_ralu_ccr_carry_wr_en;
wire       ctrl_ralu_ccr_half_carry_wr_en;
wire       ctrl_ralu_ccr_over_wr_en;
wire       ctrl_ralu_ccr_sign_wr_en;
wire       ctrl_ralu_ccr_zero_wr_en;
wire [7:0] ralu_out;
wire       reset_b;
wire ralu_shifter_a_lsb_out;
wire ralu_shifter_a_msb_out;
wire [1:0] ctrl_ralu_ccr_carry_sel;

// ------------------------------------------------------------------------
// WIRE/REG DECLARATION (OUTPUT)
// ------------------------------------------------------------------------
reg  ralu_ccr_carry_reg;
reg  ralu_ccr_carry_tmp_reg;
reg  ralu_ccr_half_carry_reg;
```

```
reg  ralu_ccr_over_reg;
reg  ralu_ccr_sign_reg;
reg  ralu_ccr_zero_reg;

// ----------------------------------------------------------------------------
// INTERNAL WIRE DECLARATION
// ----------------------------------------------------------------------------
wire  ralu_ccr_sign;
wire  ralu_ccr_zero;
reg ralu_ccr_carry;

// ----------------------------------------------------------------------------
// conditiona flag generation
// ----------------------------------------------------------------------------
assign ralu_ccr_zero = !(|ralu_out);
assign ralu_ccr_sign = ralu_out[7];

// ----------------------------------------------------------------------------
// Conditional register
// ----------------------------------------------------------------------------
always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
    begin
      ralu_ccr_carry_tmp_reg <= 1'b0;
    end
  else if (ctrl_ralu_ccr_carry_tmp_wr_en)
      ralu_ccr_carry_tmp_reg <= alu_flag_carry;
  else
      ralu_ccr_carry_tmp_reg <= ralu_ccr_carry_tmp_reg;
end

// ----------------------------------------------------------------------------
// mux shifter a lsb input
// ----------------------------------------------------------------------------
always  @(ctrl_ralu_ccr_carry_sel  or  alu_flag_carry  or  ralu_shifter_a_lsb_out  or
ralu_shifter_a_msb_out)
begin
  case(ctrl_ralu_ccr_carry_sel)
    2'b00 : ralu_ccr_carry = alu_flag_carry;
    2'b01 : ralu_ccr_carry = ralu_shifter_a_lsb_out;
    2'b10 : ralu_ccr_carry = ralu_shifter_a_msb_out;
    default : ralu_ccr_carry = 1'b0;
  endcase
end
```

```
always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
    begin
      ralu_ccr_carry_reg <= 1'b0;
    end
  else if (ctrl_ralu_ccr_carry_wr_en)
      ralu_ccr_carry_reg <= ralu_ccr_carry;
  else
      ralu_ccr_carry_reg <= ralu_ccr_carry_reg;
end

always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
    begin
      ralu_ccr_over_reg <= 1'b0;
    end
  else if (ctrl_ralu_ccr_over_wr_en)
      ralu_ccr_over_reg <= alu_flag_over & (~ctrl_alu_flag_over_mask0);
  else
      ralu_ccr_over_reg <= ralu_ccr_over_reg;
end

always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
    begin
      ralu_ccr_sign_reg <= 1'b0;
    end
  else if (ctrl_ralu_ccr_sign_wr_en)
      ralu_ccr_sign_reg <= ralu_ccr_sign;
  else
      ralu_ccr_sign_reg <= ralu_ccr_sign_reg;
end

always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
    begin
      ralu_ccr_zero_reg <= 1'b0;
    end
  else if (ctrl_ralu_ccr_zero_wr_en)
      ralu_ccr_zero_reg <= ralu_ccr_zero;
  else
      ralu_ccr_zero_reg <= ralu_ccr_zero_reg;
```

```
end

always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
    begin
      ralu_ccr_half_carry_reg <= 1'b0;
    end
  else if (ctrl_ralu_ccr_half_carry_wr_en)
      ralu_ccr_half_carry_reg <= alu_flag_half_carry;
  else
      ralu_ccr_half_carry_reg <= ralu_ccr_half_carry_reg;
end

endmodule // ralu_ccr
```

```
// ------------------------------------------------------------------------------
// AUTHOR         : Augusto Ken Morita
// AUTHOR'S EMAIL : augustkm@gmail.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// Version  Date         Author  Description
// 1.0     2014-01-28 Augusto Morita Initial version
// ------------------------------------------------------------------------------

module ralu_shifter (
    // input
    clk,
    ralu_shifter_a_in,
    ralu_shifter_a_lsb_in,
    ralu_shifter_a_msb_in,
    ralu_shifter_b_in,
    ralu_shifter_b_lsb_in,
    ralu_shifter_b_msb_in,
    ralu_shifter_ctl_lr,
    reset_b,
    // output
    ralu_shifter_a_lsb_out,
    ralu_shifter_a_msb_out,
    ralu_shifter_a_out,
    ralu_shifter_b_lsb_out,
    ralu_shifter_b_msb_out,
    ralu_shifter_b_out_reg
);

// ------------------------------------------------------------------------------
// INPUT DECLARATION
// ------------------------------------------------------------------------------
input       clk;                   // module clock
input [7:0] ralu_shifter_a_in;     // shifter a input
input       ralu_shifter_a_lsb_in; // (a) least significant bit in for left shift
input       ralu_shifter_a_msb_in; // (a) most significant bit in for right shift
input [7:0] ralu_shifter_b_in;     // shifter b input
input       ralu_shifter_b_lsb_in; // (b) least significant bit in for left shift
input       ralu_shifter_b_msb_in; // (b) most significant bit in for right shift
input       ralu_shifter_ctl_lr;   // shifter control hi=left shift low=right shift
input       reset_b;               // module reset

// ------------------------------------------------------------------------------
// OUTPUT DECLARATION
// ------------------------------------------------------------------------------
```

```
output         ralu_shifter_a_lsb_out; // (a) least significant bit out from right
shift
output          ralu_shifter_a_msb_out; // (a) most significant bit out from left
shift
output [7:0] ralu_shifter_a_out;     // shifter a output
output         ralu_shifter_b_lsb_out; // (b) least significant bit out from right
shift
output          ralu_shifter_b_msb_out; // (b) most significant bit out from left
shift
output       ralu_shifter_b_out_reg; // shifter b output

// ----------------------------------------------------------------------------
// WIRE DECLARATION (INPUT)
// ----------------------------------------------------------------------------
wire       clk;
wire [7:0] ralu_shifter_a_in;
wire       ralu_shifter_a_lsb_in;
wire       ralu_shifter_a_msb_in;
wire [7:0] ralu_shifter_b_in;
wire       ralu_shifter_b_lsb_in;
wire       ralu_shifter_b_msb_in;
wire       ralu_shifter_ctl_lr;
wire       reset_b;

// ----------------------------------------------------------------------------
// WIRE/REG DECLARATION (OUTPUT)
// ----------------------------------------------------------------------------
wire       ralu_shifter_a_lsb_out;
wire       ralu_shifter_a_msb_out;
wire [7:0] ralu_shifter_a_out;
wire       ralu_shifter_b_lsb_out;
wire       ralu_shifter_b_msb_out;
wire       ralu_shifter_b_out_reg;

// ----------------------------------------------------------------------------
// INTERNAL REG DECLARATION
// ----------------------------------------------------------------------------
reg [7:0] ralu_shifter_b_reg; // register to capture shifter b output

// ----------------------------------------------------------------------------
// shifter logic
// ----------------------------------------------------------------------------
assign   ralu_shifter_a_out   =   ralu_shifter_ctl_lr   ?   {ralu_shifter_a_in[6:0],
ralu_shifter_a_lsb_in} :  {ralu_shifter_a_msb_in, ralu_shifter_a_in[7:1]};
assign  ralu_shifter_b_out_int  =  ralu_shifter_ctl_lr  ?  {ralu_shifter_b_in[6:0],
ralu_shifter_b_lsb_in} :  {ralu_shifter_b_msb_in, ralu_shifter_b_in[7:1]};
```

```
// ------------------------------------------------------------------------
// shifter msb and lsb output logic
// ------------------------------------------------------------------------
assign ralu_shifter_a_lsb_out = ralu_shifter_a_in[0];
assign ralu_shifter_a_msb_out = ralu_shifter_a_in[7];
assign ralu_shifter_b_lsb_out = ralu_shifter_b_in[0];
assign ralu_shifter_b_msb_out = ralu_shifter_b_in[7];

// ------------------------------------------------------------------------
// register to caputer shifter b output
// ------------------------------------------------------------------------
always @(posedge clk or negedge reset_b)
begin
  if (!reset_b)
      ralu_shifter_b_reg <= 8'b0;
  else
      ralu_shifter_b_reg <= ralu_shifter_b_out_int;
end

// ------------------------------------------------------------------------
// shifter b registered output
// ------------------------------------------------------------------------
assign ralu_shifter_b_out_reg = ralu_shifter_b_reg;

endmodule // ralu_shifter
```

```
// ------------------------------------------------------------------------------
// AUTHOR         : Augusto Ken Morita
// AUTHOR'S EMAIL : augustkm@gmail.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// Version  Date         Author  Description
// 1.0     2014-01-28 Augusto Morita Initial version
// ------------------------------------------------------------------------------

module ralu (
    // input
    alu_a_in_sel,
    alu_b_in_sel,
    alu_c_in_sel,
    alu_oper,
    bus_data_a,
    bus_data_b,
    clk,
    ctrl_alu_flag_carry_tmp_wr_en,
    ctrl_alu_flag_carry_wr_en,
    ctrl_alu_flag_half_carry_wr_en,
    ctrl_alu_flag_over_mask0,
    ctrl_alu_flag_over_wr_en,
    ctrl_alu_flag_sign_wr_en,
    ctrl_alu_flag_zero_wr_en,
    ctrl_ralu_shifter_a_lsb_sel,
    ctrl_ralu_shifter_a_msb_sel,
    ipbi_rdata,
    pc_reg_7_0,
    pipe_reg1,
    ralu_out_shift_sel,
    ralu_regfile_rdaddr_a,
    ralu_regfile_rdaddr_b,
    ralu_regfile_wdata_sel,
    ralu_regfile_wr_en,
    ralu_regfile_wraddr,
    ralu_shifter_ctl_lr,
    reset_b,
    ctrl_ralu_ccr_carry_sel,
    // output
    alu_flag_carry,
    alu_flag_half_carry,
    alu_flag_over,
    alu_flag_sign,
    alu_flag_zero,
    ralu_out,
```

```
    rdata_reg_h,
    rdata_reg_sph,
    rdata_reg_spl,
    rdata_reg_x
);

// ----------------------------------------------------------------------------
// INPUT DECLARATION
// ----------------------------------------------------------------------------
input [2:0] alu_a_in_sel;
input [1:0] alu_b_in_sel;
input [1:0] alu_c_in_sel;
input [2:0] alu_oper;                       // alu alu_operation select
input [7:0] bus_data_a;
input [7:0] bus_data_b;                     // register file data b
input       clk;                            // module clock
input       ctrl_alu_flag_carry_tmp_wr_en;
input       ctrl_alu_flag_carry_wr_en;
input       ctrl_alu_flag_half_carry_wr_en;
input       ctrl_alu_flag_over_mask0;
input       ctrl_alu_flag_over_wr_en;
input       ctrl_alu_flag_sign_wr_en;
input       ctrl_alu_flag_zero_wr_en;
input       ctrl_ralu_shifter_a_lsb_sel;
input       ctrl_ralu_shifter_a_msb_sel;
input [7:0] ipbi_rdata;
input [7:0] pc_reg_7_0;
input [7:0] pipe_reg1;
input           ralu_out_shift_sel;                    // control ralu ouput from
(alu/shifter)
input [2:0] ralu_regfile_rdaddr_a;
input [2:0] ralu_regfile_rdaddr_b;
input [1:0] ralu_regfile_wdata_sel;
input       ralu_regfile_wr_en;
input [2:0] ralu_regfile_wraddr;
input       ralu_shifter_ctl_lr;
input       reset_b;                        // module reset
input [1:0] ctrl_ralu_ccr_carry_sel;

// ----------------------------------------------------------------------------
// OUTPUT DECLARATION
// ----------------------------------------------------------------------------
output       alu_flag_carry;      // conditional flag carry
output       alu_flag_half_carry;
output       alu_flag_over;       // conditional flag overflow
output       alu_flag_sign;       // conditional flag sign
```

```
output       alu_flag_zero;       // conditional flag zero
output [7:0] ralu_out;            // ralu result
output [7:0] rdata_reg_h;
output [7:0] rdata_reg_sph;
output [7:0] rdata_reg_spl;
output [7:0] rdata_reg_x;

// ------------------------------------------------------------------------------
// WIRE DECLARATION (INPUT)
// ------------------------------------------------------------------------------
wire [2:0] alu_a_in_sel;
wire [1:0] alu_b_in_sel;
wire [1:0] alu_c_in_sel;
wire [2:0] alu_oper;
wire [7:0] bus_data_a;
wire [7:0] bus_data_b;
wire       clk;
wire       ctrl_alu_flag_carry_tmp_wr_en;
wire       ctrl_alu_flag_carry_wr_en;
wire       ctrl_alu_flag_half_carry_wr_en;
wire       ctrl_alu_flag_over_mask0;
wire       ctrl_alu_flag_over_wr_en;
wire       ctrl_alu_flag_sign_wr_en;
wire       ctrl_alu_flag_zero_wr_en;
wire       ctrl_ralu_shifter_a_lsb_sel;
wire       ctrl_ralu_shifter_a_msb_sel;
wire [7:0] ipbi_rdata;
wire [7:0] pc_reg_7_0;
wire [7:0] pipe_reg1;
wire       ralu_out_shift_sel;
wire [2:0] ralu_regfile_rdaddr_a;
wire [2:0] ralu_regfile_rdaddr_b;
wire [1:0] ralu_regfile_wdata_sel;
wire       ralu_regfile_wr_en;
wire [2:0] ralu_regfile_wraddr;
wire       ralu_shifter_ctl_lr;
wire       reset_b;
input [1:0] ctrl_ralu_ccr_carry_sel;

// ------------------------------------------------------------------------------
// WIRE/REG DECLARATION (OUTPUT)
// ------------------------------------------------------------------------------
wire       alu_flag_carry;
wire       alu_flag_half_carry;
wire       alu_flag_over;
wire       alu_flag_sign;
```

```
wire       alu_flag_zero;
reg  [7:0] ralu_out;
wire [7:0] rdata_reg_h;
wire [7:0] rdata_reg_sph;
wire [7:0] rdata_reg_spl;
wire [7:0] rdata_reg_x;

// ----------------------------------------------------------------------------
// INTERNAL WIRE DECLARATION
// ----------------------------------------------------------------------------
wire [7:0] alu_result;
wire [7:0] ralu_regfile_rddata_a;
wire [7:0] ralu_regfile_rddata_b;
wire       ralu_shifter_a_lsb_out;
wire       ralu_shifter_a_msb_out;
wire [7:0] ralu_shifter_a_out;
wire [7:0] ralu_shifter_b_msb_out;
wire       ralu_shifter_b_out_reg;
wire [7:0] zero_8bit;              // internal 8 bit zero signal {0000_0000}

// ----------------------------------------------------------------------------
// INTERNAL REG DECLARATION
// ----------------------------------------------------------------------------
reg [7:0] alu_a_in;
reg [7:0] alu_b_in;
reg       alu_c_in;
wire      ralu_ccr_carry_reg;
wire       ralu_ccr_carry_tmp_reg;
reg [7:0] ralu_regfile_wrdata;
reg       ralu_shifter_a_lsb_in;
reg       ralu_shifter_a_msb_in;
reg [7:0] ralu_shifter_b_in;

// ----------------------------------------------------------------------------
// internal signal asign
// ----------------------------------------------------------------------------
assign zero_8bit = 8'b0;

// ----------------------------------------------------------------------------
// register file write data
// ----------------------------------------------------------------------------
always @(bus_data_a or ralu_shifter_a_out or bus_data_b or alu_result or zero_8bit)
begin
  case(ralu_regfile_wdata_sel)
    2'b00 : ralu_regfile_wrdata = alu_result;
    2'b01 : ralu_regfile_wrdata = bus_data_a;
```

```
    2'b10 : ralu_regfile_wrdata = ralu_shifter_a_out;
    2'b11 : ralu_regfile_wrdata = bus_data_b;
    default : ralu_regfile_wrdata = zero_8bit;
  endcase
end

// ----------------------------------------------------------------------------
// instnce regfile
// ----------------------------------------------------------------------------

ralu_regfile regfile (
   //input
   .clk                   (clk),                   // module clock
   .ralu_regfile_rdaddr_a (ralu_regfile_rdaddr_a), // port a read address
   .ralu_regfile_rdaddr_b (ralu_regfile_rdaddr_b), // port b read address
   .ralu_regfile_wr_en    (ralu_regfile_wr_en),    // write enable signal
   .ralu_regfile_wraddr   (ralu_regfile_wraddr),   // regfile write address
   .ralu_regfile_wrdata   (ralu_regfile_wrdata),   // write data
   .reset_b               (reset_b),               // module reset
    //output
   .ralu_regfile_rddata_a (ralu_regfile_rddata_a), // port a read data
   .ralu_regfile_rddata_b (ralu_regfile_rddata_b), // port a read data
   .rdata_reg_h           (rdata_reg_h),
   .rdata_reg_sph         (rdata_reg_sph),
   .rdata_reg_spl         (rdata_reg_spl),
   .rdata_reg_x           (rdata_reg_x)
);


// ----------------------------------------------------------------------------
// mux alu a input
// ----------------------------------------------------------------------------

always @(alu_a_in_sel or pipe_reg1 or ipbi_rdata or ralu_regfile_rddata_a or
pc_reg_7_0 or zero_8bit)
begin
  case(alu_a_in_sel)
    3'b001 : alu_a_in = pipe_reg1;
    3'b010 : alu_a_in = ipbi_rdata;
    3'b011 : alu_a_in = ralu_regfile_rddata_a;      // register file data a
    3'b100 : alu_a_in = pc_reg_7_0;      // register file data a
    3'b110 : alu_a_in = 8'hFF;
    default : alu_a_in = zero_8bit;
  endcase
end
```

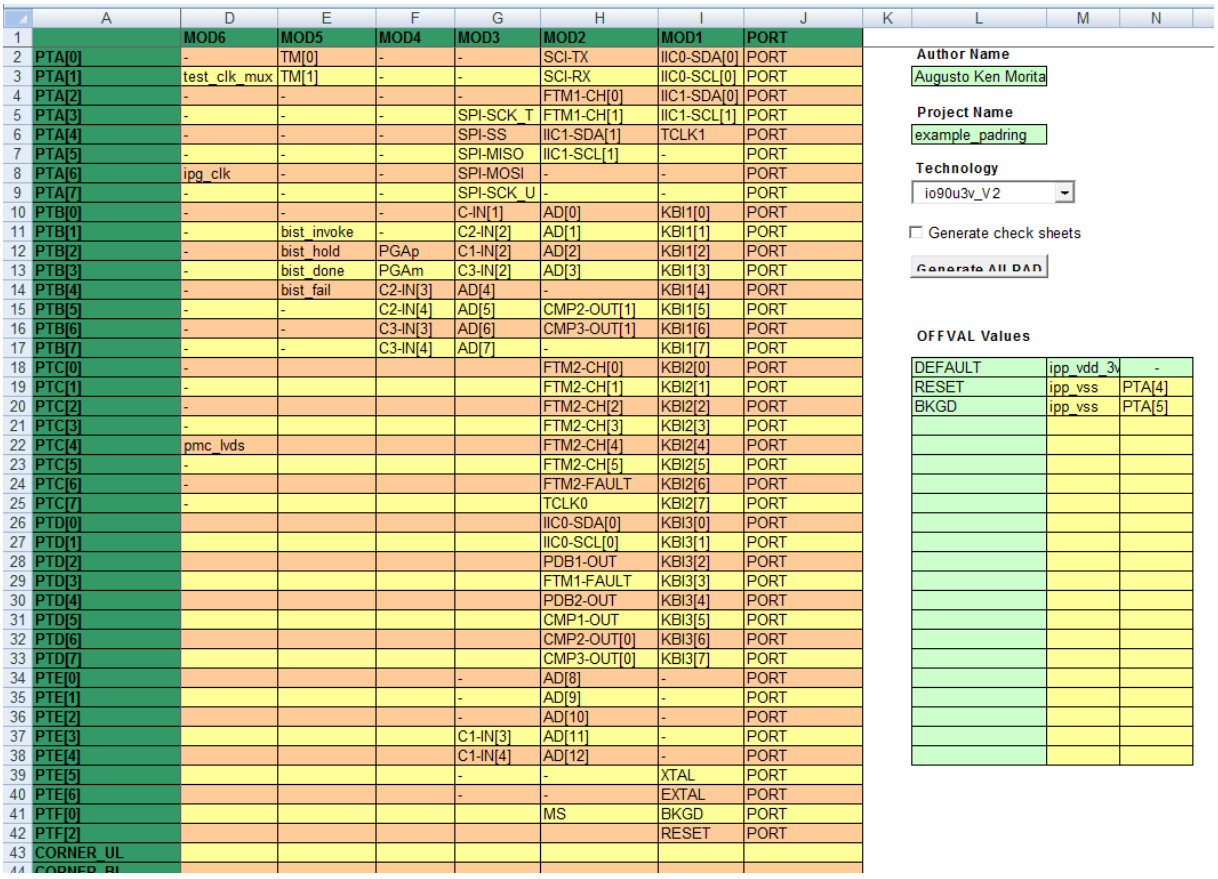
```
// ------------------------------------------------------------------------
// mux alu b input
// ------------------------------------------------------------------------
always    @(alu_b_in_sel    or    ralu_regfile_rddata_b    or    bus_data_b    or
ralu_shifter_b_out_reg or zero_8bit)
begin
  case(alu_b_in_sel)
    2'b00 : alu_b_in = ralu_regfile_rddata_b;
    2'b01 : alu_b_in = bus_data_b;
    2'b10 : alu_b_in = ralu_shifter_b_out_reg;
    2'b11 : alu_b_in = zero_8bit;
    default : alu_b_in = zero_8bit;
  endcase
end


always @(alu_c_in_sel or ralu_ccr_carry_reg)
begin
  case(alu_c_in_sel)
    2'b00 : alu_c_in = 1'b0;
    2'b11 : alu_c_in = 1'b1;
    2'b10 : alu_c_in = ralu_ccr_carry_reg;
    2'b01 : alu_c_in = ralu_ccr_carry_tmp_reg;
    default : alu_c_in = 1'b0;
  endcase
end


// ------------------------------------------------------------------------
// instance alu
// ------------------------------------------------------------------------
alu alu (
   //input
   .alu_a_in            (alu_a_in),            // input data a
   .alu_b_in            (alu_b_in),            // input data b
   .alu_c_in            (alu_c_in),            // Carry In input
   .alu_oper            (alu_oper),            // alu operation select
    //output
   .alu_flag_carry      (alu_flag_carry),      // carry out
   .alu_flag_half_carry (alu_flag_half_carry), // contitional flag half carry
   .alu_flag_over       (alu_flag_over),       // conditional flag overflow
   .alu_result          (alu_result)           // result
);

// ------------------------------------------------------------------------
// Conditional register
```

```
// ------------------------------------------------------------------------
ralu_ccr ralu_ccr (
   //input
   .alu_flag_carry                (alu_flag_carry),
   .alu_flag_half_carry           (alu_flag_half_carry),
   .alu_flag_over                 (alu_flag_over),
   .clk                              (clk),                              // module
clock
   .ctrl_alu_flag_over_mask0      (ctrl_alu_flag_over_mask0),
   .ctrl_ralu_ccr_carry_tmp_wr_en (ctrl_alu_flag_carry_tmp_wr_en),
   .ctrl_ralu_ccr_carry_wr_en     (ctrl_alu_flag_carry_wr_en),
   .ctrl_ralu_ccr_half_carry_wr_en (ctrl_alu_flag_half_carry_wr_en),
   .ctrl_ralu_ccr_over_wr_en      (ctrl_alu_flag_over_wr_en),
   .ctrl_ralu_ccr_sign_wr_en      (ctrl_alu_flag_sign_wr_en),
   .ctrl_ralu_ccr_zero_wr_en      (ctrl_alu_flag_zero_wr_en),
   .ralu_out                      (ralu_out),
   .reset_b                          (reset_b),                          // module
reset
   .ralu_shifter_a_lsb_out (ralu_shifter_a_lsb_out),
   .ralu_shifter_a_msb_out(ralu_shifter_a_msb_out),

   .ctrl_ralu_ccr_carry_sel (ctrl_ralu_ccr_carry_sel),
    //output
   .ralu_ccr_carry_reg            (ralu_ccr_carry_reg),
   .ralu_ccr_carry_tmp_reg        (ralu_ccr_carry_tmp_reg),
   .ralu_ccr_half_carry_reg       (),
   .ralu_ccr_over_reg             (),
   .ralu_ccr_sign_reg             (),
   .ralu_ccr_zero_reg             ()
);

// ------------------------------------------------------------------------
// mux shifter b input
// ------------------------------------------------------------------------
always @(alu_a_in_sel or alu_result or ralu_shifter_b_out_reg or zero_8bit)
begin
  case(alu_a_in_sel)
    1'b0 : ralu_shifter_b_in = alu_result;
    1'b1 : ralu_shifter_b_in = ralu_shifter_b_out_reg;
    default : ralu_shifter_b_in = zero_8bit;
  endcase
end

// ------------------------------------------------------------------------
// mux shifter a lsb input
// ------------------------------------------------------------------------
```

```
always @(ctrl_ralu_shifter_a_lsb_sel or ralu_shifter_b_msb_out)
begin
  case(ctrl_ralu_shifter_a_lsb_sel)
    1'b0 : ralu_shifter_a_lsb_in = 1'b0;
    1'b1 : ralu_shifter_a_lsb_in = ralu_shifter_b_msb_out;
    default : ralu_shifter_a_lsb_in = 1'b0;
  endcase
end

// ------------------------------------------------------------------------------
// mux shifter a msb input
// ------------------------------------------------------------------------------
always @(ctrl_ralu_shifter_a_msb_sel or alu_result)
begin
  case(ctrl_ralu_shifter_a_msb_sel)
    1'b0 : ralu_shifter_a_msb_in = 1'b0;
    1'b1 : ralu_shifter_a_msb_in = alu_result[7];
    default : ralu_shifter_a_msb_in = 1'b0;
  endcase
end

// ------------------------------------------------------------------------------
// instance shifter
// ------------------------------------------------------------------------------
ralu_shifter shifter (
   //input
   .clk                    (clk),                    // module clock
   .ralu_shifter_a_in      (alu_result),             // shifter a input
   .ralu_shifter_a_lsb_in   (ralu_shifter_a_lsb_in),   // (a) least significant bit
in for left shift
   .ralu_shifter_a_msb_in  (ralu_shifter_a_msb_in),  // (a) most significant bit in
for right shift
   .ralu_shifter_b_in      (ralu_shifter_b_in),      // shifter b input
   .ralu_shifter_b_lsb_in  (),                         // (b) least significant bit
in for left shift
   .ralu_shifter_b_msb_in  (ralu_shifter_a_lsb_out), // (b) most significant bit in
for right shift
   .ralu_shifter_ctl_lr      (ralu_shifter_ctl_lr),      // shifter control hi=left
shift low=right shift
   .reset_b                (reset_b),                // module reset
    //output
   .ralu_shifter_a_lsb_out  (ralu_shifter_a_lsb_out), // (a) least significant bit
out from right shift
   .ralu_shifter_a_msb_out (ralu_shifter_a_msb_out),                          // (a)
most significant bit out from left shift
   .ralu_shifter_a_out     (ralu_shifter_a_out),     // shifter a output
```

```
   .ralu_shifter_b_lsb_out (),                          // (b) least significant bit
out from right shift
   .ralu_shifter_b_msb_out  (ralu_shifter_b_msb_out), //  (b)  most  significant  bit
out from left shift
   .ralu_shifter_b_out_reg (ralu_shifter_b_out_reg)  // shifter b output
);

// ----------------------------------------------------------------------------
// mux shifter b input
// ----------------------------------------------------------------------------
always @(ralu_out_shift_sel or alu_result or ralu_shifter_a_out or zero_8bit)
begin
  case(ralu_out_shift_sel)
    1'b0 : ralu_out = alu_result;
    1'b1 : ralu_out = ralu_shifter_a_out;
    default : ralu_out = zero_8bit;
  endcase
end

endmodule // ralu
```

APÊNDICE B - Exemplo de geração de *padring*



	A	D	E	F	G	H	I	J
1		MOD6	MOD5	MOD4	MOD3	MOD2	MOD1	PORT
2	PTA[0]	-	TM[0]	-	-	SCI-TX	IIC0-SDA[0]	PORT
3	PTA[1]	test_clk_mux	TM[1]	-	-	SCI-RX	IIC0-SCL[0]	PORT
4	PTA[2]	-	-	-	-	FTM1-CH[0]	IIC1-SDA[0]	PORT
5	PTA[3]	-	-	-	SPI-SCK_T	FTM1-CH[1]	IIC1-SCL[1]	PORT
6	PTA[4]	-	-	-	SPI-SS	IIC1-SDA[1]	TCLK1	PORT
7	PTA[5]	-	-	-	SPI-MISO	IIC1-SCL[1]	-	PORT
8	PTA[6]	ipg_clk	-	-	SPI-MOSI	-	-	PORT
9	PTA[7]	-	-	-	SPI-SCK_U	-	-	PORT
10	PTB[0]	-	-	-	C-IN[1]	AD[0]	KBI1[0]	PORT
11	PTB[1]	-	bist_invoke	-	C2-IN[2]	AD[1]	KBI1[1]	PORT
12	PTB[2]	-	bist_hold	PGAp	C1-IN[2]	AD[2]	KBI1[2]	PORT
13	PTB[3]	-	bist_done	PGAm	C3-IN[2]	AD[3]	KBI1[3]	PORT
14	PTB[4]	-	bist_fail	C2-IN[3]	AD[4]	-	KBI1[4]	PORT
15	PTB[5]	-	-	C2-IN[4]	AD[5]	CMP2-OUT[1]	KBI1[5]	PORT
16	PTB[6]	-	-	C3-IN[3]	AD[6]	CMP3-OUT[1]	KBI1[6]	PORT
17	PTB[7]	-	-	C3-IN[4]	AD[7]	-	KBI1[7]	PORT
18	PTC[0]	-				FTM2-CH[0]	KBI2[0]	PORT
19	PTC[1]	-				FTM2-CH[1]	KBI2[1]	PORT
20	PTC[2]	-				FTM2-CH[2]	KBI2[2]	PORT
21	PTC[3]	-				FTM2-CH[3]	KBI2[3]	PORT
22	PTC[4]	pmc_lvds				FTM2-CH[4]	KBI2[4]	PORT
23	PTC[5]	-				FTM2-CH[5]	KBI2[5]	PORT
24	PTC[6]	-				FTM2-FAULT	KBI2[6]	PORT
25	PTC[7]	-				TCLK0	KBI2[7]	PORT
26	PTD[0]					IIC0-SDA[0]	KBI3[0]	PORT
27	PTD[1]					IIC0-SCL[0]	KBI3[1]	PORT
28	PTD[2]					PDB1-OUT	KBI3[2]	PORT
29	PTD[3]					FTM1-FAULT	KBI3[3]	PORT
30	PTD[4]					PDB2-OUT	KBI3[4]	PORT
31	PTD[5]					CMP1-OUT	KBI3[5]	PORT
32	PTD[6]					CMP2-OUT[0]	KBI3[6]	PORT
33	PTD[7]					CMP3-OUT[0]	KBI3[7]	PORT
34	PTE[0]				-	AD[8]	-	PORT
35	PTE[1]				-	AD[9]	-	PORT
36	PTE[2]				-	AD[10]	-	PORT
37	PTE[3]				C1-IN[3]	AD[11]	-	PORT
38	PTE[4]				C1-IN[4]	AD[12]	-	PORT
39	PTE[5]				-	-	XTAL	PORT
40	PTE[6]				-	-	EXTAL	PORT
41	PTF[0]					MS	BKGD	PORT
42	PTF[2]						RESET	PORT
43	CORNER_UL							

Códigos gerados:

pad_multi_example_padring.v -> 7184 linhas
padring_example_padring.v - > 2412 linhas
padi_multi_example_padring_stub.v -> 1183 linhas
padring_example_padring_stub.v -> 215 linhas

A. Trecho do código (pad_multi_example_padring.v -> 7184 linhas)

```
// ------------------------------------------------------------------------------
// FILE NAME       : pad_multi_example_padring.v
// AUTHOR          : Augusto Ken Morita
// AUTHOR'S EMAIL  : augusto.morita@freescale.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// VER. DATE                    AUTHOR                    DESCRIPTION
// ------------------------------------------------------------------------------
// KEYWORDS  :
// ------------------------------------------------------------------------------
// PURPOSE  : Top level padring module for 9S08DZ128 MCU.
//          : Instantiate all I/O pads used in the MCU.
// ------------------------------------------------------------------------------
// PARAMETERS
//     Name              : Description    : Default  : Unit
// ------------------------------------------------------------------------------
// REUSE ISSUES
```

```
//     Reset Strategy       : N/A
//     Clock Domains        : No clocked logic
//     Critical Timing      : N/A
//     Test Features        :
//     Asynchronous I/F     : None
//     Scan Methodology     : N/A
//     Instantiations       :
//                          : Module Name    Instance Name
//                          : pad_ip      -> PTA0 -- PTA7
//                          : pad_ip      -> PTB0 -- PTB7
//                          : pad_ip      -> PTC0 -- PTC7
//                          : pad_ip      -> PTD0 -- PTD7
//                          : pad_ip      -> PTE0 -- PTE6
//                          : pad_ip      -> PTF0, 2
//                          : pad_ip      -> PTF1
//     Synthesizable (y/n)  : yes
//     Other                :
// -----------------------------------------------------------------------------

module pad_multi_example_padring (
        // output pins
        ipp_do_pta,
        ipp_do_ptb,
        ipp_do_ptc,
        ipp_do_ptd,
        ipp_do_pte,
        ipp_do_ptf,
        ipp_dse_pta,
        ipp_dse_ptb,
        ipp_dse_ptc,
        ipp_dse_ptd,
        ipp_dse_pte,
        ipp_dse_ptf,
        ipp_hys_pta,
        ipp_hys_ptb,
        ipp_hys_ptc,
        ipp_hys_ptd,
        ipp_hys_pte,
        ipp_hys_ptf,
.......
        ipp_pus_pte,
        ipp_pus_ptf,
        ipp_sre_pta,
        ipp_sre_ptb,
        ipp_sre_ptc,
        ipp_sre_ptd,
        ipp_sre_pte,
        ipp_sre_ptf,

        // input pins
        ipp_ana_en_mod1_pta,
        ipp_ana_en_mod1_ptb,
        ipp_ana_en_mod1_ptc,
        ipp_ana_en_mod1_ptd,
        ipp_ana_en_mod1_pte,
        ipp_ana_en_mod1_ptf,
        ipp_ana_en_mod2_pta,
        ipp_ana_en_mod2_ptb,
        ipp_ana_en_mod2_ptc,
..............
        scan_pin_sel_ptc,
        scan_pin_sel_ptd,
        scan_pin_sel_pte,
        scan_pin_sel_ptf,
        scan_sre_override_pta,
        scan_sre_override_ptb,
        scan_sre_override_ptc,
        scan_sre_override_ptd,
```

```
        scan_sre_override_pte,
        scan_sre_override_ptf,
        scan_tri_pta,
        scan_tri_ptb,
        scan_tri_ptc,
        scan_tri_ptd,
        scan_tri_pte,
        scan_tri_ptf,
        test_ife_override_pta,
        test_ife_override_ptb,
        test_ife_override_ptc,
        test_ife_override_ptd,
        test_ife_override_pte,
        test_ife_override_ptf
);

// output
    output [7:0] ipp_do_pta;
    output [7:0] ipp_do_ptb;
    output [7:0] ipp_do_ptc;
..............
// input
    input [7:0] ipp_ana_en_mod1_pta;
    input [7:0] ipp_ana_en_mod1_ptb;
    input [7:0] ipp_ana_en_mod1_ptc;
    input [7:0] ipp_ana_en_mod1_ptd;
    input [6:0] ipp_ana_en_mod1_pte;
    input [2:0] ipp_ana_en_mod1_ptf;
    input [7:0] ipp_ana_en_mod2_pta;
    input [7:0] ipp_ana_en_mod2_ptb;
..............

// output wire
    wire [7:0]     ipp_do_pta;
    wire [7:0]     ipp_do_ptb;
    wire [7:0]     ipp_do_ptc;
    wire [7:0]     ipp_do_ptd;
    wire [6:0]     ipp_do_pte;

..............

// input wire
    wire [7:0]    ipp_ana_en_mod1_pta;
    wire [7:0]    ipp_ana_en_mod1_ptb;
    wire [7:0]    ipp_ana_en_mod1_ptc;
.................


// ------------------------------------------------------------------------------
// instance PTA0 -> Pad cell = pad_ip
// ------------------------------------------------------------------------------
    pad_ip PTA0 (
            .ipp_obe           (ipp_obe_pta[0]),
            .ipp_do            (ipp_do_pta[0]),
            .ipp_ode           (ipp_ode_pta[0]),
            .ipp_dse           (ipp_dse_pta[0]),
            .ipp_pue           (ipp_pue_pta[0]),
            .ipp_pus           (ipp_pus_pta[0]),
            .ipp_ibe           (ipp_ibe_pta[0]),
            .ipp_hys           (ipp_hys_pta[0]),
            .ipp_sre           (ipp_sre_pta[0]),
            .ipp_ife           (ipp_ife_pta[0]),
            .ipp_offval        (ipp_offval_pta[0]),
            .ipp_ind_scan      (ipp_ind_scan_pta[0]),
            .ipp_ind_mod6      (ipp_ind_mod6_pta[0]),
            .ipp_ind_mod5      (ipp_ind_mod5_pta[0]),
            .ipp_ind_mod4      (ipp_ind_mod4_pta[0]),
            .ipp_ind_mod3      (ipp_ind_mod3_pta[0]),
```

```
.ipp_ind_mod2       (ipp_ind_mod2_pta[0]),
.ipp_ind_mod1       (ipp_ind_mod1_pta[0]),
.ipp_ind_port       (ipp_ind_port_pta[0]),
.ipp_ind            (ipp_ind_pta[0]),
.offval_pad         (offval_pad_pta[0]),
.ipp_port_en_scan   (ipp_port_en_scan_pta[0]),
.ipp_do_scan        (ipp_do_scan_pta[0]),
.scan_pin_sel       (scan_pin_sel_pta[0]),
.scan_sre_override (scan_sre_override_pta[0]),
.scan_tri           (scan_tri_pta[0]),
.test_ife_override (test_ife_override_pta[0]),
.ipp_port_en_mod6   (ipp_port_en_mod6_pta[0]),
.ipp_obe_mod6       (ipp_obe_mod6_pta[0]),
.ipp_do_mod6        (ipp_do_mod6_pta[0]),
.ipp_ode_mod6       (ipp_ode_mod6_pta[0]),
.ipp_dse_mod6       (ipp_dse_mod6_pta[0]),
.ipp_pue_mod6       (ipp_pue_mod6_pta[0]),
.ipp_pus_mod6       (ipp_pus_mod6_pta[0]),
.ipp_ibe_mod6       (ipp_ibe_mod6_pta[0]),
.ipp_hys_mod6       (ipp_hys_mod6_pta[0]),
.ipp_sre_mod6       (ipp_sre_mod6_pta[0]),
.ipp_ife_mod6       (ipp_ife_mod6_pta[0]),
.ipp_offval_mod6    (ipp_offval_mod6_pta[0]),
.ipp_ana_en_mod6    (ipp_ana_en_mod6_pta[0]),
.ipp_port_en_mod5   (ipp_port_en_mod5_pta[0]),
.ipp_obe_mod5       (ipp_obe_mod5_pta[0]),
.ipp_do_mod5        (ipp_do_mod5_pta[0]),
.ipp_ode_mod5       (ipp_ode_mod5_pta[0]),
.ipp_dse_mod5       (ipp_dse_mod5_pta[0]),
.ipp_pue_mod5       (ipp_pue_mod5_pta[0]),
.ipp_pus_mod5       (ipp_pus_mod5_pta[0]),
.ipp_ibe_mod5       (ipp_ibe_mod5_pta[0]),
.ipp_hys_mod5       (ipp_hys_mod5_pta[0]),
.ipp_sre_mod5       (ipp_sre_mod5_pta[0]),
.ipp_ife_mod5       (ipp_ife_mod5_pta[0]),
.ipp_offval_mod5    (ipp_offval_mod5_pta[0]),
.ipp_ana_en_mod5    (ipp_ana_en_mod5_pta[0]),
.ipp_port_en_mod4   (ipp_port_en_mod4_pta[0]),
.ipp_obe_mod4       (ipp_obe_mod4_pta[0]),
.ipp_do_mod4        (ipp_do_mod4_pta[0]),
.ipp_ode_mod4       (ipp_ode_mod4_pta[0]),
.ipp_dse_mod4       (ipp_dse_mod4_pta[0]),
.ipp_pue_mod4       (ipp_pue_mod4_pta[0]),
.ipp_pus_mod4       (ipp_pus_mod4_pta[0]),
.ipp_ibe_mod4       (ipp_ibe_mod4_pta[0]),
.ipp_hys_mod4       (ipp_hys_mod4_pta[0]),
.ipp_sre_mod4       (ipp_sre_mod4_pta[0]),
.ipp_ife_mod4       (ipp_ife_mod4_pta[0]),
.ipp_offval_mod4    (ipp_offval_mod4_pta[0]),
.ipp_ana_en_mod4    (ipp_ana_en_mod4_pta[0]),
.ipp_port_en_mod3   (ipp_port_en_mod3_pta[0]),
.ipp_obe_mod3       (ipp_obe_mod3_pta[0]),
.ipp_do_mod3        (ipp_do_mod3_pta[0]),
.ipp_ode_mod3       (ipp_ode_mod3_pta[0]),
.ipp_dse_mod3       (ipp_dse_mod3_pta[0]),
.ipp_pue_mod3       (ipp_pue_mod3_pta[0]),
.ipp_pus_mod3       (ipp_pus_mod3_pta[0]),
.ipp_ibe_mod3       (ipp_ibe_mod3_pta[0]),
.ipp_hys_mod3       (ipp_hys_mod3_pta[0]),
.ipp_sre_mod3       (ipp_sre_mod3_pta[0]),
.ipp_ife_mod3       (ipp_ife_mod3_pta[0]),
.ipp_offval_mod3    (ipp_offval_mod3_pta[0]),
.ipp_ana_en_mod3    (ipp_ana_en_mod3_pta[0]),
.ipp_port_en_mod2   (ipp_port_en_mod2_pta[0]),
.ipp_obe_mod2       (ipp_obe_mod2_pta[0]),
.ipp_do_mod2        (ipp_do_mod2_pta[0]),
.ipp_ode_mod2       (ipp_ode_mod2_pta[0]),
.ipp_dse_mod2       (ipp_dse_mod2_pta[0]),
```

```
            .ipp_pue_mod2       (ipp_pue_mod2_pta[0]),
            .ipp_pus_mod2       (ipp_pus_mod2_pta[0]),
            .ipp_ibe_mod2       (ipp_ibe_mod2_pta[0]),
            .ipp_hys_mod2       (ipp_hys_mod2_pta[0]),
            .ipp_sre_mod2       (ipp_sre_mod2_pta[0]),
            .ipp_ife_mod2       (ipp_ife_mod2_pta[0]),
            .ipp_offval_mod2    (ipp_offval_mod2_pta[0]),
            .ipp_ana_en_mod2    (ipp_ana_en_mod2_pta[0]),
            .ipp_port_en_mod1   (ipp_port_en_mod1_pta[0]),
            .ipp_obe_mod1       (ipp_obe_mod1_pta[0]),
            .ipp_do_mod1        (ipp_do_mod1_pta[0]),
            .ipp_ode_mod1       (ipp_ode_mod1_pta[0]),
            .ipp_dse_mod1       (ipp_dse_mod1_pta[0]),
            .ipp_pue_mod1       (ipp_pue_mod1_pta[0]),
            .ipp_pus_mod1       (ipp_pus_mod1_pta[0]),
            .ipp_ibe_mod1       (ipp_ibe_mod1_pta[0]),
            .ipp_hys_mod1       (ipp_hys_mod1_pta[0]),
            .ipp_sre_mod1       (ipp_sre_mod1_pta[0]),
            .ipp_ife_mod1       (ipp_ife_mod1_pta[0]),
            .ipp_offval_mod1    (ipp_offval_mod1_pta[0]),
            .ipp_ana_en_mod1    (ipp_ana_en_mod1_pta[0]),
            .ipp_obe_port       (ipp_obe_port_pta[0]),
            .ipp_do_port        (ipp_do_port_pta[0]),
            .ipp_ode_port       (ipp_ode_port_pta[0]),
            .ipp_dse_port       (ipp_dse_port_pta[0]),
            .ipp_pue_port       (ipp_pue_port_pta[0]),
            .ipp_pus_port       (ipp_pus_port_pta[0]),
            .ipp_ibe_port       (ipp_ibe_port_pta[0]),
            .ipp_hys_port       (ipp_hys_port_pta[0]),
            .ipp_sre_port       (ipp_sre_port_pta[0]),
            .ipp_ife_port       (ipp_ife_port_pta[0])
            ); // end PTA0

// ------------------------------------------------------------------------------
// instance PTA1 -> Pad cell = pad_ip
// ------------------------------------------------------------------------------
    pad_ip PTA1 (
            .ipp_obe            (ipp_obe_pta[1]),
            .ipp_do             (ipp_do_pta[1]),
            .ipp_ode            (ipp_ode_pta[1]),
            .ipp_dse            (ipp_dse_pta[1]),
            .ipp_pue            (ipp_pue_pta[1]),
            .ipp_pus            (ipp_pus_pta[1]),
            .ipp_ibe            (ipp_ibe_pta[1]),
            .ipp_hys            (ipp_hys_pta[1]),
            .ipp_sre            (ipp_sre_pta[1]),
            .ipp_ife            (ipp_ife_pta[1]),
            .ipp_offval         (ipp_offval_pta[1]),
            .ipp_ind_scan       (ipp_ind_scan_pta[1]),
            .ipp_ind_mod6       (ipp_ind_mod6_pta[1]),
            .ipp_ind_mod5       (ipp_ind_mod5_pta[1]),
            .ipp_ind_mod4       (ipp_ind_mod4_pta[1]),
            .ipp_ind_mod3       (ipp_ind_mod3_pta[1]),
            .ipp_ind_mod2       (ipp_ind_mod2_pta[1]),
            .ipp_ind_mod1       (ipp_ind_mod1_pta[1]),
            .ipp_ind_port       (ipp_ind_port_pta[1]),
            .ipp_ind            (ipp_ind_pta[1]),
            .offval_pad         (offval_pad_pta[1]),
            .ipp_port_en_scan   (ipp_port_en_scan_pta[1]),
            .ipp_do_scan        (ipp_do_scan_pta[1]),
            .scan_pin_sel       (scan_pin_sel_pta[1]),
            .scan_sre_override  (scan_sre_override_pta[1]),
            .scan_tri           (scan_tri_pta[1]),
            .test_ife_override  (test_ife_override_pta[1]),
            .ipp_port_en_mod6   (ipp_port_en_mod6_pta[1]),
            .ipp_obe_mod6       (ipp_obe_mod6_pta[1]),
            .ipp_do_mod6        (ipp_do_mod6_pta[1]),
            .ipp_ode_mod6       (ipp_ode_mod6_pta[1]),
```

```
.ipp_dse_mod6      (ipp_dse_mod6_pta[1]),
.ipp_pue_mod6      (ipp_pue_mod6_pta[1]),
.ipp_pus_mod6      (ipp_pus_mod6_pta[1]),
.ipp_ibe_mod6      (ipp_ibe_mod6_pta[1]),
.ipp_hys_mod6      (ipp_hys_mod6_pta[1]),
.ipp_sre_mod6      (ipp_sre_mod6_pta[1]),
.ipp_ife_mod6      (ipp_ife_mod6_pta[1]),
.ipp_offval_mod6   (ipp_offval_mod6_pta[1]),
.ipp_ana_en_mod6   (ipp_ana_en_mod6_pta[1]),
.ipp_port_en_mod5  (ipp_port_en_mod5_pta[1]),
.ipp_obe_mod5      (ipp_obe_mod5_pta[1]),
.ipp_do_mod5       (ipp_do_mod5_pta[1]),
.ipp_ode_mod5      (ipp_ode_mod5_pta[1]),
.ipp_dse_mod5      (ipp_dse_mod5_pta[1]),
.ipp_pue_mod5      (ipp_pue_mod5_pta[1]),
.ipp_pus_mod5      (ipp_pus_mod5_pta[1]),
.ipp_ibe_mod5      (ipp_ibe_mod5_pta[1]),
.ipp_hys_mod5      (ipp_hys_mod5_pta[1]),
.ipp_sre_mod5      (ipp_sre_mod5_pta[1]),
.ipp_ife_mod5      (ipp_ife_mod5_pta[1]),
.ipp_offval_mod5   (ipp_offval_mod5_pta[1]),
.ipp_ana_en_mod5   (ipp_ana_en_mod5_pta[1]),
.ipp_port_en_mod4  (ipp_port_en_mod4_pta[1]),
.ipp_obe_mod4      (ipp_obe_mod4_pta[1]),
.ipp_do_mod4       (ipp_do_mod4_pta[1]),
.ipp_ode_mod4      (ipp_ode_mod4_pta[1]),
.ipp_dse_mod4      (ipp_dse_mod4_pta[1]),
.ipp_pue_mod4      (ipp_pue_mod4_pta[1]),
.ipp_pus_mod4      (ipp_pus_mod4_pta[1]),
.ipp_ibe_mod4      (ipp_ibe_mod4_pta[1]),
.ipp_hys_mod4      (ipp_hys_mod4_pta[1]),
.ipp_sre_mod4      (ipp_sre_mod4_pta[1]),
.ipp_ife_mod4      (ipp_ife_mod4_pta[1]),
.ipp_offval_mod4   (ipp_offval_mod4_pta[1]),
.ipp_ana_en_mod4   (ipp_ana_en_mod4_pta[1]),
.ipp_port_en_mod3  (ipp_port_en_mod3_pta[1]),
.ipp_obe_mod3      (ipp_obe_mod3_pta[1]),
.ipp_do_mod3       (ipp_do_mod3_pta[1]),
.ipp_ode_mod3      (ipp_ode_mod3_pta[1]),
.ipp_dse_mod3      (ipp_dse_mod3_pta[1]),
.ipp_pue_mod3      (ipp_pue_mod3_pta[1]),
.ipp_pus_mod3      (ipp_pus_mod3_pta[1]),
.ipp_ibe_mod3      (ipp_ibe_mod3_pta[1]),
.ipp_hys_mod3      (ipp_hys_mod3_pta[1]),
.ipp_sre_mod3      (ipp_sre_mod3_pta[1]),
.ipp_ife_mod3      (ipp_ife_mod3_pta[1]),
.ipp_offval_mod3   (ipp_offval_mod3_pta[1]),
.ipp_ana_en_mod3   (ipp_ana_en_mod3_pta[1]),
.ipp_port_en_mod2  (ipp_port_en_mod2_pta[1]),
.ipp_obe_mod2      (ipp_obe_mod2_pta[1]),
.ipp_do_mod2       (ipp_do_mod2_pta[1]),
.ipp_ode_mod2      (ipp_ode_mod2_pta[1]),
.ipp_dse_mod2      (ipp_dse_mod2_pta[1]),
.ipp_pue_mod2      (ipp_pue_mod2_pta[1]),
.ipp_pus_mod2      (ipp_pus_mod2_pta[1]),
.ipp_ibe_mod2      (ipp_ibe_mod2_pta[1]),
.ipp_hys_mod2      (ipp_hys_mod2_pta[1]),
.ipp_sre_mod2      (ipp_sre_mod2_pta[1]),
.ipp_ife_mod2      (ipp_ife_mod2_pta[1]),
.ipp_offval_mod2   (ipp_offval_mod2_pta[1]),
.ipp_ana_en_mod2   (ipp_ana_en_mod2_pta[1]),
.ipp_port_en_mod1  (ipp_port_en_mod1_pta[1]),
.ipp_obe_mod1      (ipp_obe_mod1_pta[1]),
.ipp_do_mod1       (ipp_do_mod1_pta[1]),
.ipp_ode_mod1      (ipp_ode_mod1_pta[1]),
.ipp_dse_mod1      (ipp_dse_mod1_pta[1]),
.ipp_pue_mod1      (ipp_pue_mod1_pta[1]),
.ipp_pus_mod1      (ipp_pus_mod1_pta[1]),
```

```
            .ipp_ibe_mod1      (ipp_ibe_mod1_pta[1]),
            .ipp_hys_mod1      (ipp_hys_mod1_pta[1]),
            .ipp_sre_mod1      (ipp_sre_mod1_pta[1]),
            .ipp_ife_mod1      (ipp_ife_mod1_pta[1]),
            .ipp_offval_mod1   (ipp_offval_mod1_pta[1]),
            .ipp_ana_en_mod1   (ipp_ana_en_mod1_pta[1]),
            .ipp_obe_port      (ipp_obe_port_pta[1]),
            .ipp_do_port       (ipp_do_port_pta[1]),
            .ipp_ode_port      (ipp_ode_port_pta[1]),
            .ipp_dse_port      (ipp_dse_port_pta[1]),
            .ipp_pue_port      (ipp_pue_port_pta[1]),
            .ipp_pus_port      (ipp_pus_port_pta[1]),
            .ipp_ibe_port      (ipp_ibe_port_pta[1]),
            .ipp_hys_port      (ipp_hys_port_pta[1]),
            .ipp_sre_port      (ipp_sre_port_pta[1]),
            .ipp_ife_port      (ipp_ife_port_pta[1])
            ); // end PTA1

.................

endmodule // pad_multi_example_padring
```

B. Trecho do código (padring_example_padring.-> 2412 linhas)

```
// ------------------------------------------------------------------------------
// FILE NAME       : padring_example_padring.v
// AUTHOR          : Augusto Ken Morita
// AUTHOR'S EMAIL  : augusto.morita@freescale.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// VER. DATE                    AUTHOR                  DESCRIPTION
// ------------------------------------------------------------------------------
// KEYWORDS  :
// ------------------------------------------------------------------------------
// PURPOSE  : Top level padring module for 9S08DZ128 MCU.
//          : Instantiate all I/O pads used in the MCU.
// ------------------------------------------------------------------------------
// PARAMETERS
//     Name              : Description    : Default  : Unit
//     teste 1           : sdafsdf        : 1        : ssshh
//     teste parametter  : asdf sadfsdaf  : 2        :
// ------------------------------------------------------------------------------
// REUSE ISSUES
//     Reset Strategy       : N/A
//     Clock Domains        : No clocked logic
//     Critical Timing      : N/A
//     Test Features        :
//     Asynchronous I/F     : None
//     Scan Methodology     : N/A
//     Instantiations       :
//                          : Module Name          Instance Name
//                          : p_io_3v_4m        -> PTA0 -- PTA7
//                          : p_io_3v_4m        -> PTB0 -- PTB7
//                          : p_io_3v_4m        -> PTC0 -- PTC7
//                          : p_io_3v_4m        -> PTD0 -- PTD7
//                          : p_io_3v_4m        -> PTE0 -- PTE6
//                          : p_io_3v_4m        -> PTF0, 2
//                          : pad_spc_crn       -> CORNER_UL
//                          : pad_spc_crn       -> CORNER_BL
//                          : pad_spc_crn       -> CORNER_BR
//                          : pad_spc_crn       -> CORNER_UR
//                          : p_vss_3v_4m       -> Vss1
//                          : p_vss_3v_nobuf_4m -> Vss1_2NDB
//                          : p_vss_3v_nobuf_4m -> Vss2
//                          : p_vdd_3v_4m       -> Vdd1
//                          : p_vdd_3v_nobuf_4m -> Vdd1_2NDB
//                          : p_vdd_3v_nobuf_4m -> Vdd2
//                          : p_vdda_3v_4m      -> Vddad
//                          : p_vssa_3v_4m      -> Vssad
//                          : p_vssa_3v_4m      -> Vrefl
//                          : p_vdda_3v_4m      -> Vrefh
//                          : p_vpp_3v_4m       -> PTF1
//                          : pad_tm            -> TM0
//                          : pad_tm            -> TM1
//                          : pad_trans         -> TRANS0 -- TRANS8
//     Synthesizable (y/n)  : yes
//     Other                :
// ------------------------------------------------------------------------------

module padring_example_padring (
        // output pins
        ipp_fls_tm,
        ipp_fls_vpp0,
        ipp_ina_pta,
        ipp_ina_ptb,
        ipp_ina_ptc,
        ipp_ina_ptd,
        ipp_ina_pte,
        ipp_ina_ptf,
```

```
ipp_ind_pta,
ipp_ind_ptb,
ipp_ind_ptc,
ipp_ind_ptd,
ipp_ind_pte,
ipp_ind_ptf,
ipp_vdd_3v,
ipp_vdda,
ipp_vss,
ipp_vssa,

// input pins
ipp_do_pta,
ipp_do_ptb,
ipp_do_ptc,
ipp_do_ptd,
ipp_do_pte,
ipp_do_ptf,
ipp_dse_pta,
ipp_dse_ptb,
ipp_dse_ptc,
ipp_dse_ptd,
ipp_dse_pte,
ipp_dse_ptf,
ipp_ibe_pta,
ipp_ibe_ptb,
ipp_ibe_ptc,
ipp_ibe_ptd,
ipp_ibe_pte,
ipp_ibe_ptf,
ipp_ife_pta,
ipp_ife_ptb,
ipp_ife_ptc,
ipp_ife_ptd,
ipp_ife_pte,
ipp_ife_ptf,
ipp_obe_pta,
ipp_obe_ptb,
ipp_obe_ptc,
ipp_obe_ptd,
ipp_obe_pte,
ipp_obe_ptf,
ipp_ode_pta,
ipp_ode_ptb,
ipp_ode_ptc,
ipp_ode_ptd,
ipp_ode_pte,
ipp_ode_ptf,
ipp_pue_pta,
ipp_pue_ptb,
ipp_pue_ptc,
ipp_pue_ptd,
ipp_pue_pte,
ipp_pue_ptf,
ipp_pus_pta,
ipp_pus_ptb,
ipp_pus_ptc,
ipp_pus_ptd,
ipp_pus_pte,
ipp_pus_ptf,
ipp_sre_pta,
ipp_sre_ptb,
ipp_sre_ptc,
ipp_sre_ptd,
ipp_sre_pte,
ipp_sre_ptf,
pmc_close_3p3,
pmc_driver_vdd_sw,
```

```
        pmc_por_3v,
        sim_ipp_port_en_fls_tm,

        // inout pins
        ipp_outa_pta,
        ipp_outa_ptb,
        ipp_outa_ptc,
        ipp_outa_ptd,
        ipp_outa_pte,
        ipp_outa_ptf,
        pad_pta,
        pad_ptb,
        pad_ptc,
        pad_ptd,
        pad_pte,
        pad_ptf,
        pad_vdd1,
        pad_vdd1_2ndb,
        pad_vdd2,
        pad_vddad,
        pad_vrefh,
        pad_vrefl,
        pad_vss1,
        pad_vss1_2ndb,
        pad_vss2,
        pad_vssad
);

// output
    output [1:0] ipp_fls_tm;
    output       ipp_fls_vpp0;
    output [7:0] ipp_ina_pta;
    output [7:0] ipp_ina_ptb;
    output [7:0] ipp_ina_ptc;
..........
// input
    input [7:0] ipp_do_pta;             // Data Out
    input [7:0] ipp_do_ptb;             // Data Out
    input [7:0] ipp_do_ptc;             // Data Out
    input [7:0] ipp_do_ptd;             // Data Out
    input [6:0] ipp_do_pte;             // Data Out
.............

// inout
    inout [7:0] ipp_outa_pta;
    inout [7:0] ipp_outa_ptb;
    inout [7:0] ipp_outa_ptc;
.............
    inout       pad_vrefh;
    inout       pad_vrefl;
    inout       pad_vss1;
    inout       pad_vss1_2ndb;
    inout       pad_vss2;
    inout       pad_vssad;

// internal wire
    wire             CLOSE_3V;
    wire             ESD_BOOST;
..............

// inout wire
    wire [7:0]    ipp_outa_pta;
    wire [7:0]    ipp_outa_ptb;
..........


// ------------------------------------------------------------------------------
// instance PTA0 -> Pad cell = p_io_3v_4m
```

```
// ------------------------------------------------------------------------------
    p_io_3v_4m PTA0 (
            .IPP_INA_3V      (ipp_ina_pta[0]),
            .IPP_INA_MUX_3V  (ipp_ina_mux_3v_pta_nc[0]),
            .IPP_OUTA_MUX_3V (ipp_outa_mux_3v_pta_nc[0]),
            .IPP_OUTA_3V     (ipp_outa_pta[0]),
            .PAD             (pad_pta[0]),
            .IPP_IND         (ipp_ind_pta[0]),
            .IPP_IND_3V      (ipp_ind_3v_pta_nc[0]),
            .IPP_IBE         (ipp_ibe_pta[0]),
            .IPP_IFE         (ipp_ife_pta[0]),
            .IPP_DO          (ipp_do_pta[0]),
            .IPP_OBE         (ipp_obe_pta[0]),
            .IPP_ODE         (ipp_ode_pta[0]),
            .IPP_DSE         (ipp_dse_pta[0]),
            .IPP_PUE         (ipp_pue_pta[0]),
            .IPP_PUS         (ipp_pus_pta[0]),
            .IPP_SRE         (ipp_sre_pta[0]),
            .VBUF_3V         (VBUF_3V),
            .VDD_3V          (VDD_3V),
            .VSS_3V          (VSS_3V),
            .VDD_LV          (VDD_LV),
            .VSS_LV          (VSS_LV),
            .TIE_3V          (TIE_3V),
            .TIE_HI          (TIE_HI),
            .TIE_LO          (TIE_LO),
            .ESD_TRIGGER     (ESD_TRIGGER),
            .ESD_BOOST       (ESD_BOOST),
            .PADRST_3V       (PADRST_3V),
            .CLOSE_3V        (CLOSE_3V),
            .IPP_AME         (ipp_vss),
            .IPP_OFFVALB_3V  (ipp_vdd_3v)
            ); // end PTA0

// ------------------------------------------------------------------------------
// instance PTA1 -> Pad cell = p_io_3v_4m
// ------------------------------------------------------------------------------
    p_io_3v_4m PTA1 (
            .IPP_INA_3V      (ipp_ina_pta[1]),
            .IPP_INA_MUX_3V  (ipp_ina_mux_3v_pta_nc[1]),
            .IPP_OUTA_MUX_3V (ipp_outa_mux_3v_pta_nc[1]),
            .IPP_OUTA_3V     (ipp_outa_pta[1]),
            .PAD             (pad_pta[1]),
            .IPP_IND         (ipp_ind_pta[1]),
            .IPP_IND_3V      (ipp_ind_3v_pta_nc[1]),
            .IPP_IBE         (ipp_ibe_pta[1]),
            .IPP_IFE         (ipp_ife_pta[1]),
            .IPP_DO          (ipp_do_pta[1]),
            .IPP_OBE         (ipp_obe_pta[1]),
            .IPP_ODE         (ipp_ode_pta[1]),
            .IPP_DSE         (ipp_dse_pta[1]),
            .IPP_PUE         (ipp_pue_pta[1]),
            .IPP_PUS         (ipp_pus_pta[1]),
            .IPP_SRE         (ipp_sre_pta[1]),
            .VBUF_3V         (VBUF_3V),
            .VDD_3V          (VDD_3V),
            .VSS_3V          (VSS_3V),
            .VDD_LV          (VDD_LV),
            .VSS_LV          (VSS_LV),
            .TIE_3V          (TIE_3V),
            .TIE_HI          (TIE_HI),
            .TIE_LO          (TIE_LO),
            .ESD_TRIGGER     (ESD_TRIGGER),
            .ESD_BOOST       (ESD_BOOST),
            .PADRST_3V       (PADRST_3V),
            .CLOSE_3V        (CLOSE_3V),
            .IPP_AME         (ipp_vss),
            .IPP_OFFVALB_3V  (ipp_vdd_3v)
```

```
            ); // end PTA1

// ------------------------------------------------------------------------------
// instance VDD1 -> Pad cell = p_vdd_3v_4m
// ------------------------------------------------------------------------------
    p_vdd_3v_4m VDD1 (
            .PAD            (pad_vdd1),
            .VDD_CORE_3V    (ipp_vdd_3v),
            .VDD_CORE_LV    (pmc_driver_vdd_sw),
            .VSS_CORE_LV    (ipp_vss),
            .IPP_PADRST_3V (pmc_por_3v),
            .VBUF_3V        (VBUF_3V),
            .VDD_3V         (VDD_3V),
            .VSS_3V         (VSS_3V),
            .VDD_LV         (VDD_LV),
            .VSS_LV         (VSS_LV),
            .TIE_3V         (TIE_3V),
            .TIE_HI         (TIE_HI),
            .TIE_LO         (TIE_LO),
            .ESD_TRIGGER    (ESD_TRIGGER),
            .ESD_BOOST      (ESD_BOOST),
            .PADRST_3V      (PADRST_3V),
            .CLOSE_3V       (CLOSE_3V)
            ); // end VDD1
// ------------------------------------------------------------------------------
// instance VDDAD -> Pad cell = p_vdda_3v_4m
// ------------------------------------------------------------------------------
    p_vdda_3v_4m VDDAD (
            .PAD           (pad_vddad),
            .IPP_INA_3V    (ipp_ina_3v_vddad_nc),
            .IPP_OUTA_3V   (ipp_outa_3v_vddad_nc),
            .VDDA_CORE_3V (ipp_vdda),
            .VDD_CORE_LV   (pmc_driver_vdd_sw),
            .VSS_CORE_LV   (ipp_vss),
            .VBUF_3V       (VBUF_3V),
            .VDD_3V        (VDD_3V),
            .VSS_3V        (VSS_3V),
            .VDD_LV        (VDD_LV),
            .VSS_LV        (VSS_LV),
            .TIE_3V        (TIE_3V),
            .TIE_HI        (TIE_HI),
            .TIE_LO        (TIE_LO),
            .ESD_TRIGGER   (ESD_TRIGGER),
            .ESD_BOOST     (ESD_BOOST),
            .PADRST_3V     (PADRST_3V),
            .CLOSE_3V      (CLOSE_3V)
            ); // end VDDAD

// ------------------------------------------------------------------------------
// instance VSSAD -> Pad cell = p_vssa_3v_4m
// ------------------------------------------------------------------------------
    p_vssa_3v_4m VSSAD (
            .PAD           (pad_vssad),
            .IPP_INA_3V    (ipp_ina_3v_vssad_nc),
            .IPP_OUTA_3V   (ipp_outa_3v_vssad_nc),
            .VSSA_CORE_3V (ipp_vssa),
            .VDD_CORE_LV   (pmc_driver_vdd_sw),
            .VSS_CORE_LV   (ipp_vss),
            .VBUF_3V       (VBUF_3V),
            .VDD_3V        (VDD_3V),
            .VSS_3V        (VSS_3V),
            .VDD_LV        (VDD_LV),
            .VSS_LV        (VSS_LV),
            .TIE_3V        (TIE_3V),
            .TIE_HI        (TIE_HI),
            .TIE_LO        (TIE_LO),
            .ESD_TRIGGER   (ESD_TRIGGER),
            .ESD_BOOST     (ESD_BOOST),
```

```
            .PADRST_3V    (PADRST_3V),
            .CLOSE_3V     (CLOSE_3V)
            ); // end VSSAD

// ------------------------------------------------------------------------------
// instance VREFL -> Pad cell = p_vssa_3v_4m
// ------------------------------------------------------------------------------
    p_vssa_3v_4m VREFL (
            .PAD          (pad_vrefl),
            .IPP_INA_3V   (ipp_ina_3v_vrefl_nc),
            .IPP_OUTA_3V  (ipp_outa_3v_vrefl_nc),
            .VSSA_CORE_3V (ipp_vssa),
            .VDD_CORE_LV  (pmc_driver_vdd_sw),
            .VSS_CORE_LV  (ipp_vss),
            .VBUF_3V      (VBUF_3V),
            .VDD_3V       (VDD_3V),
            .VSS_3V       (VSS_3V),
            .VDD_LV       (VDD_LV),
            .VSS_LV       (VSS_LV),
            .TIE_3V       (TIE_3V),
            .TIE_HI       (TIE_HI),
            .TIE_LO       (TIE_LO),
            .ESD_TRIGGER  (ESD_TRIGGER),
            .ESD_BOOST    (ESD_BOOST),
            .PADRST_3V    (PADRST_3V),
            .CLOSE_3V     (CLOSE_3V)
            ); // end VREFL

// ------------------------------------------------------------------------------
// instance PTF1 -> Pad cell = p_vpp_3v_4m
// ------------------------------------------------------------------------------
    p_vpp_3v_4m PTF1 (
            .IPP_INA_3V      (ipp_ina_ptf[1]),
            .IPP_INA_MUX_3V  (ipp_ina_mux_3v_ptf_nc[1]),
            .IPP_OUTA_MUX_3V (ipp_outa_mux_3v_ptf_nc[1]),
            .IPP_OUTA_3V     (ipp_outa_ptf[1]),
            .VPP1            (ipp_fls_vpp0),
            .VPP2            (vpp2_nc),
            .PAD             (pad_ptf[1]),
            .IPP_IND         (ipp_ind_ptf[1]),
            .IPP_IND_3V      (ipp_ind_3v_ptf_nc[1]),
            .IPP_IBE         (ipp_ibe_ptf[1]),
            .IPP_IFE         (ipp_ife_ptf[1]),
            .IPP_DO          (1'b0),
            .IPP_OBE         (ipp_obe_ptf[1]),
            .IPP_PUE         (ipp_pue_ptf[1]),
            .IPP_PUS         (ipp_pus_ptf[1]),
            .VBUF_3V         (VBUF_3V),
            .VDD_3V          (VDD_3V),
            .VSS_3V          (VSS_3V),
            .VDD_LV          (VDD_LV),
            .VSS_LV          (VSS_LV),
            .TIE_3V          (TIE_3V),
            .TIE_HI          (TIE_HI),
            .TIE_LO          (TIE_LO),
            .ESD_TRIGGER     (ESD_TRIGGER),
            .ESD_BOOST       (ESD_BOOST),
            .PADRST_3V       (PADRST_3V),
            .CLOSE_3V        (CLOSE_3V),
            .IPP_AME         (ipp_vss),
            .IPP_OFFVALB_3V  (ipp_vdd_3v)
            ); // end PTF1
..............
// ------------------------------------------------------------------------------
// instance TM0 -> Pad cell = pad_tm
// ------------------------------------------------------------------------------
    pad_tm TM0 (
            .INA_TM_3V   (ipp_fls_tm[0]),
```

```
            .IPP_TME     (sim_ipp_port_en_fls_tm),
            .MUXIN_3V    (ipp_ina_ptb[0]),
            .VBUF_3V     (VBUF_3V),
            .VDD_3V      (VDD_3V),
            .VSS_3V      (VSS_3V),
            .VDD_LV      (VDD_LV),
            .VSS_LV      (VSS_LV),
            .TIE_3V      (TIE_3V),
            .TIE_HI      (TIE_HI),
            .TIE_LO      (TIE_LO),
            .ESD_TRIGGER (ESD_TRIGGER),
            .ESD_BOOST   (ESD_BOOST),
            .PADRST_3V   (PADRST_3V),
            .CLOSE_3V    (CLOSE_3V)
            ); // end TM0
..............
// ------------------------------------------------------------------------------
// instance TRANS0 -> Pad cell = pad_trans
// ------------------------------------------------------------------------------
    pad_trans TRANS0 (
            .CLOSE_3V    (CLOSE_3V),
            .ESD_BOOST   (ESD_BOOST),
            .ESD_TRIGGER (ESD_TRIGGER),
            .PADRST_3V   (PADRST_3V),
            .TIE_3V      (TIE_3V),
            .TIE_HI      (TIE_HI),
            .TIE_LO      (TIE_LO),
            .VBUF_3V     (VBUF_3V),
            .VDD_3V      (VDD_3V),
            .VDD_LV      (VDD_LV),
            .VSS_3V      (VSS_3V),
            .VSS_LV      (VSS_LV)
            ); // end TRANS0
.................
endmodule // padring_example_padring
```

C. Trecho do código (padi_multi_example_padring_stub.v -> 1183 linhas) utilizado na instância do bloco.

```
// ******************************************************
// padmulti instance wires
// ******************************************************
wire          sim_ipp_port_en_tst_clk1
wire          ipp_ind_tst_clk1
wire          sim_test_ife_override
wire [7:0]    ipp_ind_mod6_pta_nc
wire [7:0]    ipp_ind_mod5_pta_nc
wire [7:0]    ipp_ind_mod4_pta_nc
wire          ipp_port_en_spi_sck_u
wire          ipp_ind_spi_sck_u
...........
// ******************************************************
// Instantiate the module: padring_example_padring
// ******************************************************
padring_example_padring    padring_example_padring   (
// ******************************************************
// **** PTA pin section ****
// ******************************************************
    // ******************************************************
    // SCAN
    // ******************************************************
    .offval_pad_pta        (8'b00000000),
    .ipp_port_en_scan_pta  ({sim_ipp_port_en_tst_clk1, sim_ipp_port_en_tst_clk2,
4'b0000, sim_bus_se_not_single_chain, sim_scan_not_single_chain}),
    .ipp_ind_scan_pta      ({ipp_ind_tst_clk1, ipp_ind_tst_clk2,
ipp_ind_scan_pta_nc[5..1], ipp_ind_bus_se}),
    .ipp_do_scan_pta       ({6'b000000, ipt_test_scan_out_sog[2], 1'b0}),
    .scan_sre_override_pta ({6'b000000, 2{sim_scan_mode}}),
    .scan_tri_pta          ({2'b00, 6{sim_iddq_test_enable}}),
    .scan_pin_sel_pta      (8'b00000010),
    .test_ife_override_pta (8{sim_test_ife_override})
    // ******************************************************
    // MOD6
    // ******************************************************
    .ipp_ana_en_mod6_pta   (8'b00000000),
    .ipp_port_en_mod6_pta  ({1'b0, sim_ipp_port_en_pcout, 4'b0000,
ipp_port_en_test_clk_mux, 1'b0}),
    .ipp_ind_mod6_pta      (ipp_ind_mod6_pta_nc[7..0]),
    .ipp_obe_mod6_pta      (8'b01000010),
    .ipp_do_mod6_pta       ({1'b0, ipg_clk, 4'b0000, extclk_tpm, 1'b0}),
    .ipp_ode_mod6_pta      (8'b00000000),
    .ipp_dse_mod6_pta      ({1'b0, ipp_dse_pctl_pta[6], 4'b0000,
ipp_dse_pctl_pta[1], 1'b0}),
    .ipp_pue_mod6_pta      (8'b00000000),
    .ipp_pus_mod6_pta      (8'b00000000),
    .ipp_ibe_mod6_pta      (8'b00000000),
    .ipp_hys_mod6_pta      (8'b00000000),
    .ipp_sre_mod6_pta      ({1'b0, ipp_sre_pctl_pta[6], 4'b0000,
ipp_sre_pctl_pta[1], 1'b0}),
    .ipp_ife_mod6_pta      (8'b00000000),
    .ipp_offval_mod6_pta   (8'b00000000),
    // ******************************************************
    // MOD5
    // ******************************************************
    .ipp_ana_en_mod5_pta   (8'b00000011),
    .ipp_port_en_mod5_pta  ({6'b000000, ipp_port_en_fls_tm[1..0]}),
    .ipp_ind_mod5_pta      (ipp_ind_mod5_pta_nc[7..0]),
    .ipp_obe_mod5_pta      (8'b00000000),
    .ipp_do_mod5_pta       (8'b00000000),
    .ipp_ode_mod5_pta      (8'b00000000),
    .ipp_dse_mod5_pta      (8'b00000000),
```

```
    .ipp_pue_mod5_pta      (8'b00000000),
    .ipp_pus_mod5_pta      (8'b00000000),
    .ipp_ibe_mod5_pta      (8'b00000000),
    .ipp_hys_mod5_pta      (8'b00000000),
    .ipp_sre_mod5_pta      (8'b00000000),
    .ipp_ife_mod5_pta      (8'b00000000),
    .ipp_offval_mod5_pta   (8'b00000000),
    // *********************************************************
    // MOD4
    // *********************************************************
    .ipp_ana_en_mod4_pta   (8'b00000000),
    .ipp_port_en_mod4_pta  (8'b00000000),
    .ipp_ind_mod4_pta      (ipp_ind_mod4_pta_nc[7..0]),
    .ipp_obe_mod4_pta      (8'b00000000),
    .ipp_do_mod4_pta       (8'b00000000),
    .ipp_ode_mod4_pta      (8'b00000000),
    .ipp_dse_mod4_pta      (8'b00000000),
    .ipp_pue_mod4_pta      (8'b00000000),
    .ipp_pus_mod4_pta      (8'b00000000),
    .ipp_ibe_mod4_pta      (8'b00000000),
    .ipp_hys_mod4_pta      (8'b00000000),
    .ipp_sre_mod4_pta      (8'b00000000),
    .ipp_ife_mod4_pta      (8'b00000000),
    .ipp_offval_mod4_pta   (8'b00000000),
    // *********************************************************
    // MOD3
    // *********************************************************
    .ipp_ana_en_mod3_pta   (8'b00000000),
    .ipp_port_en_mod3_pta  ({ipp_port_en_spi_sck_u, ipp_port_en_spi_mosi,
ipp_port_en_spi_miso, ipp_port_en_spi_ss, ipp_port_en_spi_sck_t, 3'b000}),
    .ipp_ind_mod3_pta      ({ipp_ind_spi_sck_u, ipp_ind_spi_mosi, ipp_ind_spi_miso,
ipp_ind_spi_ss, ipp_ind_spi_sck_t, ipp_ind_mod3_pta_nc[2..0]}),
    .ipp_obe_mod3_pta      ({ipp_obe_spi_sck, ipp_obe_spi_mosi, ipp_obe_spi_miso,
ipp_obe_spi_ss, ipp_obe_spi_sck, 3'b000}),
    .ipp_do_mod3_pta       ({ipp_do_spi_sck, ipp_do_spi_mosi, ipp_do_spi_miso,
ipp_do_spi_ss, ipp_do_spi_sck, 3'b000}),
    .ipp_ode_mod3_pta      (8'b00000000),
    .ipp_dse_mod3_pta      ({ipp_dse_pctl_pta[7..3], 3'b000}),
    .ipp_pue_mod3_pta      ({ipp_pue_pctl_pta[7..3], 3'b000}),
    .ipp_pus_mod3_pta      (8'b11111000),
    .ipp_ibe_mod3_pta      (8'b11111000),
    .ipp_hys_mod3_pta      (8'b11111000),
    .ipp_sre_mod3_pta      ({ipp_sre_pctl_pta[7..3], 3'b000}),
    .ipp_ife_mod3_pta      ({5{sim_spife}, 3'b000}),
    .ipp_offval_mod3_pta   (8'b11111000),
    // *********************************************************
    // MOD2
    // *********************************************************
    .ipp_ana_en_mod2_pta   (8'b00000000),
    .ipp_port_en_mod2_pta  ({2'b00, sim_ipp_port_en_iic1_scl[1],
sim_ipp_port_en_iic1_sda[1], ipp_port_en_ftm1_ch[1..0], ipp_port_en_sci_rx,
ipp_port_en_sci_tx}),
    .ipp_ind_mod2_pta      ({ipp_ind_mod2_pta_nc[7..6], ipp_ind_iic1_scl[1],
ipp_ind_iic1_sda[1], ipp_ind_ftm1_ch[1..0], ipp_ind_sci_rx, ipp_ind_sci_tx}),
    .ipp_obe_mod2_pta      ({2'b00, ipp_obe_iic1_scl, ipp_obe_iic1_sda,
ipp_obe_ftm1_ch[1..0], 1'b0, ipp_obe_sci_tx}),
    .ipp_do_mod2_pta       ({2'b00, ipp_do_iic1_scl, ipp_do_iic1_sda,
ipp_do_ftm1_ch[1..0], 1'b0, ipp_do_sci_tx}),
    .ipp_ode_mod2_pta      (8'b00110000),
    .ipp_dse_mod2_pta      ({2'b00, ipp_dse_pctl_pta[5..2], 1'b0,
ipp_dse_pctl_pta[0]}),
    .ipp_pue_mod2_pta      ({2'b00, ipp_pue_pctl_pta[5..0]}),
    .ipp_pus_mod2_pta      (8'b00111111),
    .ipp_ibe_mod2_pta      ({4'b0011, ipp_ibe_ftm1_ch[1..0], 1'b1,
ipp_ibe_sci_tx}),
    .ipp_hys_mod2_pta      (8'b00111111),
    .ipp_sre_mod2_pta      ({2'b00, ipp_sre_pctl_pta[5..2], 1'b0,
ipp_sre_pctl_pta[0]}),
```

```
    .ipp_ife_mod2_pta      (8'b00111111),
    .ipp_offval_mod2_pta   (8'b00110011),
    // *********************************************************
    // MOD1
    // *********************************************************
    .ipp_ana_en_mod1_pta   (8'b00000000),
    .ipp_port_en_mod1_pta  ({3'b000, ipp_port_en_tclk1,
sim_ipp_port_en_iic1_scl[1], sim_ipp_port_en_iic1_sda[0],
sim_ipp_port_en_iic0_scl[0], sim_ipp_port_en_iic0_sda[0]}),
    .ipp_ind_mod1_pta      ({ipp_ind_mod1_pta_nc[7..5], ipp_ind_tclk1,
ipp_ind_iic1_scl[1], ipp_ind_iic1_sda[0], ipp_ind_iic0_scl[0],
ipp_ind_iic0_sda[0]}),
    .ipp_obe_mod1_pta      ({4'b0000, ipp_obe_iic1_scl, ipp_obe_iic1_sda,
ipp_obe_iic0_scl, ipp_obe_iic0_sda}),
    .ipp_do_mod1_pta       ({4'b0000, ipp_do_iic1_scl, ipp_do_iic1_sda,
ipp_do_iic0_scl, ipp_do_iic0_sda}),
    .ipp_ode_mod1_pta      (8'b00001111),
    .ipp_dse_mod1_pta      ({4'b0000, ipp_dse_pctl_pta[3..0]}),
    .ipp_pue_mod1_pta      ({3'b000, ipp_pue_pctl_pta[4..0]}),
    .ipp_pus_mod1_pta      (8'b00011111),
    .ipp_ibe_mod1_pta      (8'b00011111),
    .ipp_hys_mod1_pta      (8'b00011111),
    .ipp_sre_mod1_pta      ({4'b0000, ipp_sre_pctl_pta[3..0]}),
    .ipp_ife_mod1_pta      (8'b00011111),
    .ipp_offval_mod1_pta   (8'b00001111),
    // *********************************************************
    // PORT
    // *********************************************************
    .ipp_ind_port_pta      (ipp_ind_port_pta[7..0]),
    .ipp_obe_port_pta      (ipp_obe_port_pta[7..0]),
    .ipp_do_port_pta       (ipp_do_port_pta[7..0]),
    .ipp_ode_port_pta      (8'b00000000),
    .ipp_dse_port_pta      (ipp_dse_pctl_pta[7..0]),
    .ipp_pue_port_pta      (ipp_pue_pctl_pta[7..0]),
    .ipp_pus_port_pta      (8'b11111111),
    .ipp_ibe_port_pta      (ipp_ibe_port_pta[7..0]),
    .ipp_hys_port_pta      (8'b11111111),
    .ipp_sre_port_pta      (ipp_sre_pctl_pta[7..0]),
    .ipp_ife_port_pta      (8'b11111111),
    // *********************************************************
    // MUX_OUT
    // *********************************************************
    .ipp_ana_en_pta        (ipp_ana_en_pta[7..0]),
    .ipp_port_en_pta       (ipp_port_en_pta[7..0]),
    .ipp_ind_pta           (ipp_ind_pta[7..0]),
    .ipp_obe_pta           (ipp_obe_pta[7..0]),
    .ipp_do_pta            (ipp_do_pta[7..0]),
    .ipp_ode_pta           (ipp_ode_pta[7..0]),
    .ipp_dse_pta           (ipp_dse_pta[7..0]),
    .ipp_pue_pta           (ipp_pue_pta[7..0]),
    .ipp_pus_pta           (ipp_pus_pta[7..0]),
    .ipp_ibe_pta           (ipp_ibe_pta[7..0]),
    .ipp_hys_pta           (ipp_hys_pta[7..0]),
    .ipp_sre_pta           (ipp_sre_pta[7..0]),
    .ipp_ife_pta           (ipp_ife_pta[7..0]),
    .ipp_offval_pta        (ipp_offval_pta[7..0]),

.................

// *******************************************************
// **** PTF pin section ****
// *******************************************************
    // *********************************************************
    // SCAN
    // *********************************************************
    .offval_pad_ptf        (3'b000),
    .ipp_port_en_scan_ptf  (3'b000),
    .ipp_ind_scan_ptf      (ipp_ind_scan_ptf_nc[2..0])
```

```
.ipp_do_scan_ptf       (3'b000),
.scan_sre_override_ptf (3'b000),
.scan_tri_ptf          (3{sim_iddq_test_enable}),
.scan_pin_sel_ptf      (3'b000),
.test_ife_override_ptf (3{sim_test_ife_override}),
// ********************************************************
// MOD6
// ********************************************************
.ipp_ana_en_mod6_ptf   (3'b000),
.ipp_port_en_mod6_ptf  (3'b000),
.ipp_ind_mod6_ptf      (ipp_ind_mod6_ptf_nc[2..0]),
.ipp_obe_mod6_ptf      (3'b000),
.ipp_do_mod6_ptf       (3'b000),
.ipp_ode_mod6_ptf      (3'b000),
.ipp_dse_mod6_ptf      (3'b000),
.ipp_pue_mod6_ptf      (3'b000),
.ipp_pus_mod6_ptf      (3'b000),
.ipp_ibe_mod6_ptf      (3'b000),
.ipp_hys_mod6_ptf      (3'b000),
.ipp_sre_mod6_ptf      (3'b000),
.ipp_ife_mod6_ptf      (3'b000),
.ipp_offval_mod6_ptf   (3'b000),
// ********************************************************
// MOD5
// ********************************************************
.ipp_ana_en_mod5_ptf   (3'b000),
.ipp_port_en_mod5_ptf  (3'b000),
.ipp_ind_mod5_ptf      (ipp_ind_mod5_ptf_nc[2..0]),
.ipp_obe_mod5_ptf      (3'b000),
.ipp_do_mod5_ptf       (3'b000),
.ipp_ode_mod5_ptf      (3'b000),
.ipp_dse_mod5_ptf      (3'b000),
.ipp_pue_mod5_ptf      (3'b000),
.ipp_pus_mod5_ptf      (3'b000),
.ipp_ibe_mod5_ptf      (3'b000),
.ipp_hys_mod5_ptf      (3'b000),
.ipp_sre_mod5_ptf      (3'b000),
.ipp_ife_mod5_ptf      (3'b000),
.ipp_offval_mod5_ptf   (3'b000),
// ********************************************************
// MOD4
// ********************************************************
.ipp_ana_en_mod4_ptf   (3'b000),
.ipp_port_en_mod4_ptf  (3'b000),
.ipp_ind_mod4_ptf      (ipp_ind_mod4_ptf_nc[2..0]),
.ipp_obe_mod4_ptf      (3'b000),
.ipp_do_mod4_ptf       (3'b000),
.ipp_ode_mod4_ptf      (3'b000),
.ipp_dse_mod4_ptf      (3'b000),
.ipp_pue_mod4_ptf      (3'b000),
.ipp_pus_mod4_ptf      (3'b000),
.ipp_ibe_mod4_ptf      (3'b000),
.ipp_hys_mod4_ptf      (3'b000),
.ipp_sre_mod4_ptf      (3'b000),
.ipp_ife_mod4_ptf      (3'b000),
.ipp_offval_mod4_ptf   (3'b000),
// ********************************************************
// MOD3
// ********************************************************
.ipp_ana_en_mod3_ptf   (3'b000),
.ipp_port_en_mod3_ptf  (3'b000),
.ipp_ind_mod3_ptf      (ipp_ind_mod3_ptf_nc[2..0]),
.ipp_obe_mod3_ptf      (3'b000),
.ipp_do_mod3_ptf       (3'b000),
.ipp_ode_mod3_ptf      (3'b000),
.ipp_dse_mod3_ptf      (3'b000),
.ipp_pue_mod3_ptf      (3'b000),
.ipp_pus_mod3_ptf      (3'b000),
```

```
    .ipp_ibe_mod3_ptf      (3'b000),
    .ipp_hys_mod3_ptf      (3'b000),
    .ipp_sre_mod3_ptf      (3'b000),
    .ipp_ife_mod3_ptf      (3'b000),
    .ipp_offval_mod3_ptf   (3'b000),
    // *********************************************************
    // MOD2
    // *********************************************************
    .ipp_ana_en_mod2_ptf   (3'b000),
    .ipp_port_en_mod2_ptf  ({2'b00, sim_ipp_port_en_ms}),
    .ipp_ind_mod2_ptf      ({ipp_ind_mod2_ptf_nc[2..1], ipp_ind_ms}),
    .ipp_obe_mod2_ptf      (3'b000),
    .ipp_do_mod2_ptf       (3'b000),
    .ipp_ode_mod2_ptf      (3'b000),
    .ipp_dse_mod2_ptf      (3'b000),
    .ipp_pue_mod2_ptf      ({2'b00, sim_iddq_test_enable_b}),
    .ipp_pus_mod2_ptf      (3'b001),
    .ipp_ibe_mod2_ptf      (3'b001),
    .ipp_hys_mod2_ptf      (3'b001),
    .ipp_sre_mod2_ptf      (3'b000),
    .ipp_ife_mod2_ptf      (3'b001),
    .ipp_offval_mod2_ptf   (3'b001),
    // *********************************************************
    // MOD1
    // *********************************************************
    .ipp_ana_en_mod1_ptf   (3'b000),
    .ipp_port_en_mod1_ptf  ({sim_ipp_port_en_reset, 1'b0, sim_ipp_port_en_bkgd}),
    .ipp_ind_mod1_ptf      ({ipp_ind_reset_b, ipp_ind_mod1_ptf_nc[1],
ipp_ind_bkgd}),
    .ipp_obe_mod1_ptf      ({sim_ipp_obe_reset, 1'b0, core_bkgd_ipp_obe}),
    .ipp_do_mod1_ptf       ({2'b00, core_bkgd_ipp_do}),
    .ipp_ode_mod1_ptf      (3'b000),
    .ipp_dse_mod1_ptf      (3'b101),
    .ipp_pue_mod1_ptf      ({sim_iddq_test_enable_b, 2'b01}),
    .ipp_pus_mod1_ptf      (3'b101),
    .ipp_ibe_mod1_ptf      (3'b101),
    .ipp_hys_mod1_ptf      (3'b101),
    .ipp_sre_mod1_ptf      (3'b000),
    .ipp_ife_mod1_ptf      (3'b101),
    .ipp_offval_mod1_ptf   (3'b101),
    // *********************************************************
    // PORT
    // *********************************************************
    .ipp_ind_port_ptf      (ipp_ind_port_ptf[2..0]),
    .ipp_obe_port_ptf      (ipp_obe_port_ptf[2..0]),
    .ipp_do_port_ptf       (ipp_do_port_ptf[2..0]),
    .ipp_ode_port_ptf      (3'b000),
    .ipp_dse_port_ptf      (ipp_dse_pctl_ptf[2..0]),
    .ipp_pue_port_ptf      (ipp_pue_pctl_ptf[2..0]),
    .ipp_pus_port_ptf      (3'b111),
    .ipp_ibe_port_ptf      (ipp_ibe_port_ptf[2..0]),
    .ipp_hys_port_ptf      (3'b111),
    .ipp_sre_port_ptf      (ipp_sre_pctl_ptf[2..0]),
    .ipp_ife_port_ptf      (3'b111),
    // *********************************************************
    // MUX_OUT
    // *********************************************************
    .ipp_ana_en_ptf        (ipp_ana_en_ptf[2..0]),
    .ipp_port_en_ptf       (ipp_port_en_ptf[2..0]),
    .ipp_ind_ptf           (ipp_ind_ptf[2..0]),
    .ipp_obe_ptf           (ipp_obe_ptf[2..0]),
    .ipp_do_ptf            (ipp_do_ptf[2..0]),
    .ipp_ode_ptf           (ipp_ode_ptf[2..0]),
    .ipp_dse_ptf           (ipp_dse_ptf[2..0]),
    .ipp_pue_ptf           (ipp_pue_ptf[2..0]),
    .ipp_pus_ptf           (ipp_pus_ptf[2..0]),
    .ipp_ibe_ptf           (ipp_ibe_ptf[2..0]),
    .ipp_hys_ptf           (ipp_hys_ptf[2..0]),
```

```
    .ipp_sre_ptf           (ipp_sre_ptf[2..0]),
    .ipp_ife_ptf           (ipp_ife_ptf[2..0]),
    .ipp_offval_ptf        (ipp_offval_ptf[2..0])
);
```

D. Trecho código (padring_example_padring_stub.v -> 215 linhas) utilizado na instância do bloco.

```
// ----------------------------------------------------------------------------
// Module padring_example_padring wire declarations
// ----------------------------------------------------------------------------
// output wire
    wire [1:0]     ipp_fls_tm;
    wire           ipp_fls_vpp0;
    wire [7:0]     ipp_ina_pta;
    wire [7:0]     ipp_ina_ptb;
..............
// input wire
    wire [7:0]    ipp_do_pta;             // Data Out
    wire [7:0]    ipp_do_ptb;             // Data Out
    wire [7:0]    ipp_do_ptc;             // Data Out
.................
// inout wire
    wire [7:0]    ipp_outa_pta;
    wire [7:0]    ipp_outa_ptb;
    wire [7:0]    ipp_outa_ptc;
............

// ----------------------------------------------------------------------------
// Module padring_example_padring instance declarations
// ----------------------------------------------------------------------------
padring_example_padring    padring_example_padring (
        // output pins
        .ipp_fls_tm    (ipp_fls_tm),
        .ipp_fls_vpp0  (ipp_fls_vpp0),
        .ipp_ina_pta   (ipp_ina_pta),
        .ipp_ina_ptb   (ipp_ina_ptb),
        .ipp_ina_ptc   (ipp_ina_ptc),
        .ipp_ina_ptd   (ipp_ina_ptd),
...............

        // inout pins
        .ipp_outa_pta  (ipp_outa_pta),
        .ipp_outa_ptb  (ipp_outa_ptb),
        .ipp_outa_ptc  (ipp_outa_ptc),
        .ipp_outa_ptd  (ipp_outa_ptd),
        .ipp_outa_pte  (ipp_outa_pte),
        .ipp_outa_ptf  (ipp_outa_ptf),
        .pad_pta       (pad_pta),
.............
        .pad_vssad     (pad_vssad)
    );
```

ANEXO A - *Data sheet* do microcontrolador 8 bit

O *datasheet* completo está no link abaixo. Uma pequena parte, referente ao processador, está anexa a este trabalho.

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=S08QE

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=S08QE&tab=Documentation_Tab&pspll=1&SelectedAsset=Documentation&ProdMetaId=PID/DC/S08QE&fromPSP=true&assetLockedForNavigation=true&componentId=2&leftNavCode=1&pageSize=25&Documentation=Documentation/00610Ksd1nd``Data%20Sheets&fpsp=1&linkline=Data%20Sheets

Chapter 8
Central Processor Unit (S08CPUV4)

8.1 Introduction

This section provides summary information about the registers, addressing modes, and instruction set of the CPU of the HCS08 Family. For a more detailed discussion, refer to the *HCS08 Family Reference Manual, volume 1,* Freescale Semiconductor document order number HCS08RMV1/D.

The HCS08 CPU is fully source- and object-code-compatible with the M68HC08 CPU. Several instructions and enhanced addressing modes were added to improve C compiler efficiency and to support a new background debug system which replaces the monitor mode of earlier M68HC08 microcontrollers (MCU).

8.1.1 Features

Features of the HCS08 CPU include:

- Object code fully upward-compatible with M68HC05 and M68HC08 Families
- 64-KB CPU address space with banked memory management unit for greater than 64 KB
- 16-bit stack pointer (any size stack anywhere in 64-KB CPU address space)
- 16-bit index register (H:X) with powerful indexed addressing modes
- 8-bit accumulator (A)
- Many instructions treat X as a second general-purpose 8-bit register
- Seven addressing modes:
  - — Inherent — Operands in internal registers
  - — Relative — 8-bit signed offset to branch destination
  - — Immediate — Operand in next object code byte(s)
  - — Direct — Operand in memory at 0x0000–0x00FF
  - — Extended — Operand anywhere in 64-Kbyte address space
  - — Indexed relative to H:X — Five submodes including auto increment
  - — Indexed relative to SP — Improves C efficiency dramatically
- Memory-to-memory data move instructions with four address mode combinations
- Overflow, half-carry, negative, zero, and carry condition codes support conditional branching on the results of signed, unsigned, and binary-coded decimal (BCD) operations
- Efficient bit manipulation instructions
- Fast 8-bit by 8-bit multiply and 16-bit by 8-bit divide instructions
- STOP and WAIT instructions to invoke low-power operating modes

8.2 Programmer's Model and CPU Registers

Figure 8-1 shows the five CPU registers. CPU registers are not part of the memory map.

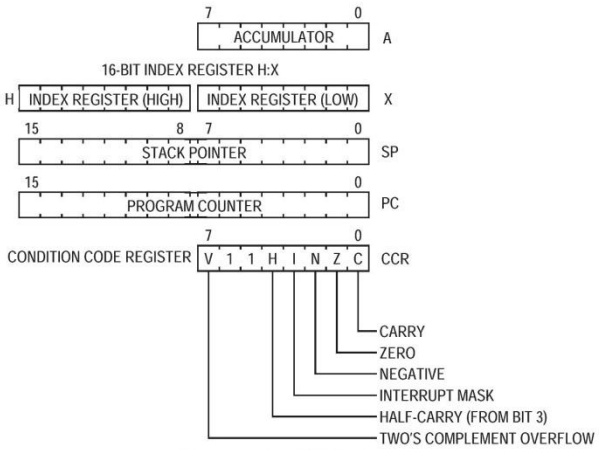


Figure 8-1. CPU Registers

8.2.1 Accumulator (A)

The A accumulator is a general-purpose 8-bit register. One operand input to the arithmetic logic unit (ALU) is connected to the accumulator and the ALU results are often stored into the A accumulator after arithmetic and logical operations. The accumulator can be loaded from memory using various addressing modes to specify the address where the loaded data comes from, or the contents of A can be stored to memory using various addressing modes to specify the address where data from A will be stored.

Reset has no effect on the contents of the A accumulator.

8.2.2 Index Register (H:X)

This 16-bit register is actually two separate 8-bit registers (H and X), which often work together as a 16-bit address pointer where H holds the upper byte of an address and X holds the lower byte of the address. All indexed addressing mode instructions use the full 16-bit value in H:X as an index reference pointer; however, for compatibility with the earlier M68HC05 Family, some instructions operate only on the low-order 8-bit half (X).

Many instructions treat X as a second general-purpose 8-bit register that can be used to hold 8-bit data values. X can be cleared, incremented, decremented, complemented, negated, shifted, or rotated. Transfer instructions allow data to be transferred from A or transferred to A where arithmetic and logical operations can then be performed.

For compatibility with the earlier M68HC05 Family, H is forced to 0x00 during reset. Reset has no effect on the contents of X.

8.2.3 Stack Pointer (SP)

This 16-bit address pointer register points at the next available location on the automatic last-in-first-out (LIFO) stack. The stack may be located anywhere in the 64-Kbyte address space that has RAM and can be any size up to the amount of available RAM. The stack is used to automatically save the return address for subroutine calls, the return address and CPU registers during interrupts, and for local variables. The AIS (add immediate to stack pointer) instruction adds an 8-bit signed immediate value to SP. This is most often used to allocate or deallocate space for local variables on the stack.

SP is forced to 0x00FF at reset for compatibility with the earlier M68HC05 Family. HCS08 programs normally change the value in SP to the address of the last location (highest address) in on-chip RAM during reset initialization to free up direct page RAM (from the end of the on-chip registers to 0x00FF).

The RSP (reset stack pointer) instruction was included for compatibility with the M68HC05 Family and is seldom used in new HCS08 programs because it only affects the low-order half of the stack pointer.

8.2.4 Program Counter (PC)

The program counter is a 16-bit register that contains the address of the next instruction or operand to be fetched.

During normal program execution, the program counter automatically increments to the next sequential memory location every time an instruction or operand is fetched. Jump, branch, interrupt, and return operations load the program counter with an address other than that of the next sequential location. This is called a change-of-flow.

During reset, the program counter is loaded with the reset vector that is located at 0xFFFE and 0xFFFF. The vector stored there is the address of the first instruction that will be executed after exiting the reset state.

8.2.5 Condition Code Register (CCR)

The 8-bit condition code register contains the interrupt mask (I) and five flags that indicate the results of the instruction just executed. Bits 6 and 5 are set permanently to 1. The following paragraphs describe the functions of the condition code bits in general terms. For a more detailed explanation of how each instruction sets the CCR bits, refer to the *HCS08 Family Reference Manual, volume 1,* Freescale Semiconductor document order number HCS08RMv1.

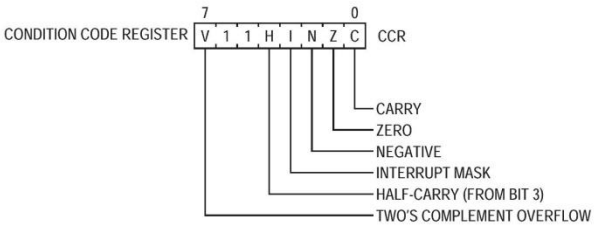


Figure 8-2. Condition Code Register

Table 8-1. CCR Register Field Descriptions

Field	Description
7 V	**Two's Complement Overflow Flag —** The CPU sets the overflow flag when a two's complement overflow occurs. The signed branch instructions BGT, BGE, BLE, and BLT use the overflow flag. 0 No overflow 1 Overflow
4 H	**Half-Carry Flag —** The CPU sets the half-carry flag when a carry occurs between accumulator bits 3 and 4 during an add-without-carry (ADD) or add-with-carry (ADC) operation. The half-carry flag is required for binary-coded decimal (BCD) arithmetic operations. The DAA instruction uses the states of the H and C condition code bits to automatically add a correction value to the result from a previous ADD or ADC on BCD operands to correct the result to a valid BCD value. 0 No carry between bits 3 and 4 1 Carry between bits 3 and 4
3 I	**Interrupt Mask Bit —** When the interrupt mask is set, all maskable CPU interrupts are disabled. CPU interrupts are enabled when the interrupt mask is cleared. When a CPU interrupt occurs, the interrupt mask is set automatically after the CPU registers are saved on the stack, but before the first instruction of the interrupt service routine is executed. Interrupts are not recognized at the instruction boundary after any instruction that clears I (CLI or TAP). This ensures that the next instruction after a CLI or TAP will always be executed without the possibility of an intervening interrupt, provided I was set. 0 Interrupts enabled 1 Interrupts disabled
2 N	**Negative Flag —** The CPU sets the negative flag when an arithmetic operation, logic operation, or data manipulation produces a negative result, setting bit 7 of the result. Simply loading or storing an 8-bit or 16-bit value causes N to be set if the most significant bit of the loaded or stored value was 1. 0 Non-negative result 1 Negative result
1 Z	**Zero Flag —** The CPU sets the zero flag when an arithmetic operation, logic operation, or data manipulation produces a result of 0x00 or 0x0000. Simply loading or storing an 8-bit or 16-bit value causes Z to be set if the loaded or stored value was all 0s. 0 Non-zero result 1 Zero result
0 C	**Carry/Borrow Flag —** The CPU sets the carry/borrow flag when an addition operation produces a carry out of bit 7 of the accumulator or when a subtraction operation requires a borrow. Some instructions — such as bit test and branch, shift, and rotate — also clear or set the carry/borrow flag. 0 No carry out of bit 7 1 Carry out of bit 7

8.3 Addressing Modes

Addressing modes define the way the CPU accesses operands and data. In the HCS08, memory, status and control registers, and input/output (I/O) ports share a single 64-Kbyte CPU address space. This arrangement means that the same instructions that access variables in RAM can also be used to access I/O and control registers or nonvolatile program space.

MCU derivatives with more than 64-Kbytes of memory also include a memory management unit (MMU) to support extended memory space. A PPAGE register is used to manage 16-Kbyte pages of memory which can be accessed by the CPU through a 16-Kbyte window from 0x8000 through 0xBFFF. The CPU includes two special instructions (CALL and RTC). CALL operates like the JSR instruction except that CALL saves the current PPAGE value on the stack and provides a new PPAGE value for the destination. RTC works like the RTS instruction except RTC restores the old PPAGE value in addition to the PC during the return from the called routine. The MMU also includes a linear address pointer register and data access registers so that the extended memory space operates as if it was a single linear block of memory. For additional information about the MMU, refer to the Memory chapter of this data sheet.

Some instructions use more than one addressing mode. For instance, move instructions use one addressing mode to specify the source operand and a second addressing mode to specify the destination address. Instructions such as BRCLR, BRSET, CBEQ, and DBNZ use one addressing mode to specify the location of an operand for a test and then use relative addressing mode to specify the branch destination address when the tested condition is true. For BRCLR, BRSET, CBEQ, and DBNZ, the addressing mode listed in the instruction set tables is the addressing mode needed to access the operand to be tested, and relative addressing mode is implied for the branch destination.

8.3.1 Inherent Addressing Mode (INH)

In this addressing mode, operands needed to complete the instruction (if any) are located within CPU registers so the CPU does not need to access memory to get any operands.

8.3.2 Relative Addressing Mode (REL)

Relative addressing mode is used to specify the destination location for branch instructions. A signed 8-bit offset value is located in the memory location immediately following the opcode. During execution, if the branch condition is true, the signed offset is sign-extended to a 16-bit value and is added to the current contents of the program counter, which causes program execution to continue at the branch destination address.

8.3.3 Immediate Addressing Mode (IMM)

In immediate addressing mode, the operand needed to complete the instruction is included in the object code immediately following the instruction opcode in memory. In the case of a 16-bit immediate operand, the high-order byte is located in the next memory location after the opcode, and the low-order byte is located in the next memory location after that.

8.3.4 Direct Addressing Mode (DIR)

In direct addressing mode, the instruction includes the low-order eight bits of an address in the direct page (0x0000–0x00FF). During execution a 16-bit address is formed by concatenating an implied 0x00 for the high-order half of the address and the direct address from the instruction to get the 16-bit address where the desired operand is located. This is faster and more memory efficient than specifying a complete 16-bit address for the operand.

8.3.5 Extended Addressing Mode (EXT)

In extended addressing mode, the full 16-bit address of the operand is located in the next two bytes of program memory after the opcode (high byte first).

8.3.6 Indexed Addressing Mode

Indexed addressing mode has seven variations including five that use the 16-bit H:X index register pair and two that use the stack pointer as the base reference.

8.3.6.1 Indexed, No Offset (IX)

This variation of indexed addressing uses the 16-bit value in the H:X index register pair as the address of the operand needed to complete the instruction.

8.3.6.2 Indexed, No Offset with Post Increment (IX+)

This variation of indexed addressing uses the 16-bit value in the H:X index register pair as the address of the operand needed to complete the instruction. The index register pair is then incremented (H:X = H:X + 0x0001) after the operand has been fetched. This addressing mode is only used for MOV and CBEQ instructions.

8.3.6.3 Indexed, 8-Bit Offset (IX1)

This variation of indexed addressing uses the 16-bit value in the H:X index register pair plus an unsigned 8-bit offset included in the instruction as the address of the operand needed to complete the instruction.

8.3.6.4 Indexed, 8-Bit Offset with Post Increment (IX1+)

This variation of indexed addressing uses the 16-bit value in the H:X index register pair plus an unsigned 8-bit offset included in the instruction as the address of the operand needed to complete the instruction. The index register pair is then incremented (H:X = H:X + 0x0001) after the operand has been fetched. This addressing mode is used only for the CBEQ instruction.

8.3.6.5 Indexed, 16-Bit Offset (IX2)

This variation of indexed addressing uses the 16-bit value in the H:X index register pair plus a 16-bit offset included in the instruction as the address of the operand needed to complete the instruction.

8.3.6.6 SP-Relative, 8-Bit Offset (SP1)

This variation of indexed addressing uses the 16-bit value in the stack pointer (SP) plus an unsigned 8-bit offset included in the instruction as the address of the operand needed to complete the instruction.

8.3.6.7 SP-Relative, 16-Bit Offset (SP2)

This variation of indexed addressing uses the 16-bit value in the stack pointer (SP) plus a 16-bit offset included in the instruction as the address of the operand needed to complete the instruction.

8.4 Special Operations

The CPU performs a few special operations that are similar to instructions but do not have opcodes like other CPU instructions. In addition, a few instructions such as STOP and WAIT directly affect other MCU circuitry. This section provides additional information about these operations.

8.4.1 Reset Sequence

Reset can be caused by a power-on-reset (POR) event, internal conditions such as the COP (computer operating properly) watchdog, or by assertion of an external active-low reset pin. When a reset event occurs, the CPU immediately stops whatever it is doing (the MCU does not wait for an instruction boundary before responding to a reset event). For a more detailed discussion about how the MCU recognizes resets and determines the source, refer to the Resets, Interrupts, and System Configuration chapter.

The reset event is considered concluded when the sequence to determine whether the reset came from an internal source is done and when the reset pin is no longer asserted. At the conclusion of a reset event, the CPU performs a 6-cycle sequence to fetch the reset vector from 0xFFFE and 0xFFFF and to fill the instruction queue in preparation for execution of the first program instruction.

8.4.2 Interrupt Sequence

When an interrupt is requested, the CPU completes the current instruction before responding to the interrupt. At this point, the program counter is pointing at the start of the next instruction, which is where the CPU should return after servicing the interrupt. The CPU responds to an interrupt by performing the same sequence of operations as for a software interrupt (SWI) instruction, except the address used for the vector fetch is determined by the highest priority interrupt that is pending when the interrupt sequence started.

The CPU sequence for an interrupt is:

1. Store the contents of PCL, PCH, X, A, and CCR on the stack, in that order.
2. Set the I bit in the CCR.
3. Fetch the high-order half of the interrupt vector.
4. Fetch the low-order half of the interrupt vector.
5. Delay for one free bus cycle.

6. Fetch three bytes of program information starting at the address indicated by the interrupt vector to fill the instruction queue in preparation for execution of the first instruction in the interrupt service routine.

After the CCR contents are pushed onto the stack, the I bit in the CCR is set to prevent other interrupts while in the interrupt service routine. Although it is possible to clear the I bit with an instruction in the interrupt service routine, this would allow nesting of interrupts (which is not recommended because it leads to programs that are difficult to debug and maintain).

For compatibility with the earlier M68HC05 MCUs, the high-order half of the H:X index register pair (H) is not saved on the stack as part of the interrupt sequence. The user must use a PSHH instruction at the beginning of the service routine to save H and then use a PULH instruction just before the RTI that ends the interrupt service routine. It is not necessary to save H if you are certain that the interrupt service routine does not use any instructions or auto-increment addressing modes that might change the value of H.

The software interrupt (SWI) instruction is like a hardware interrupt except that it is not masked by the global I bit in the CCR and it is associated with an instruction opcode within the program so it is not asynchronous to program execution.

8.4.3 Wait Mode Operation

The WAIT instruction enables interrupts by clearing the I bit in the CCR. It then halts the clocks to the CPU to reduce overall power consumption while the CPU is waiting for the interrupt or reset event that will wake the CPU from wait mode. When an interrupt or reset event occurs, the CPU clocks will resume and the interrupt or reset event will be processed normally.

If a serial BACKGROUND command is issued to the MCU through the background debug interface while the CPU is in wait mode, CPU clocks will resume and the CPU will enter active background mode where other serial background commands can be processed. This ensures that a host development system can still gain access to a target MCU even if it is in wait mode.

8.4.4 Stop Mode Operation

Usually, all system clocks, including the crystal oscillator (when used), are halted during stop mode to minimize power consumption. In such systems, external circuitry is needed to control the time spent in stop mode and to issue a signal to wake up the target MCU when it is time to resume processing. Unlike the earlier M68HC05 and M68HC08 MCUs, the HCS08 can be configured to keep a minimum set of clocks running in stop mode. This optionally allows an internal periodic signal to wake the target MCU from stop mode.

When a host debug system is connected to the background debug pin (BKGD) and the ENBDM control bit has been set by a serial command through the background interface (or because the MCU was reset into active background mode), the oscillator is forced to remain active when the MCU enters stop mode. In this case, if a serial BACKGROUND command is issued to the MCU through the background debug interface while the CPU is in stop mode, CPU clocks will resume and the CPU will enter active background mode where other serial background commands can be processed. This ensures that a host development system can still gain access to a target MCU even if it is in stop mode.

Recovery from stop mode depends on the particular HCS08 and whether the oscillator was stopped in stop mode. Refer to the Modes of Operation chapter for more details.

8.4.5 BGND Instruction

The BGND instruction is new to the HCS08 compared to the M68HC08. BGND would not be used in normal user programs because it forces the CPU to stop processing user instructions and enter the active background mode. The only way to resume execution of the user program is through reset or by a host debug system issuing a GO, TRACE1, or TAGGO serial command through the background debug interface.

Software-based breakpoints can be set by replacing an opcode at the desired breakpoint address with the BGND opcode. When the program reaches this breakpoint address, the CPU is forced to active background mode rather than continuing the user program.

The CALL is similar to a jump-to-subroutine (JSR) instruction, but the subroutine that is called can be located anywhere in the normal 64-Kbyte address space or on any page of program expansion memory. When CALL is executed, a return address is calculated, then it and the current program page register value are stacked, and a new instruction-supplied value is written to PPAGE. The PPAGE value controls which of the possible 16-Kbyte pages is visible through the window in the 64-Kbyte memory map. Execution continues at the address of the called subroutine.

The actual sequence of operations that occur during execution of CALL is:

1. CPU calculates the address of the next instruction after the CALL instruction (the return address) and pushes this 16-bit value onto the stack, low byte first.
2. CPU reads the old PPAGE value and pushes it onto the stack.
3. CPU writes the new instruction-supplied page select value to PPAGE. This switches the destination page into the program overlay window in the CPU address range 0x8000 0xBFFF.
4. Instruction queue is refilled starting from the destination address, and execution begins at the new address.

This sequence of operations is an uninterruptable CPU instruction. There is no need to inhibit interrupts during CALL execution. In addition, a CALL can be performed from any address in memory to any other address. This is a big improvement over other bank-switching schemes, where the page switch operation can be performed only by a program outside the overlay window.

For all practical purposes, the PPAGE value supplied by the instruction can be considered to be part of the effective address. The new page value is provided by an immediate operand in the instruction.

The RTC instruction is used to terminate subroutines invoked by a CALL instruction. RTC unstacks the PPAGE value and the return address, the queue is refilled, and execution resumes with the next instruction after the corresponding CALL.

The actual sequence of operations that occur during execution of RTC is:

1. The return value of the 8-bit PPAGE register is pulled from the stack.
2. The 16-bit return address is pulled from the stack and loaded into the PC.
3. The return PPAGE value is written to the PPAGE register.

4. The queue is refilled and execution begins at the new address.

Since the return operation is implemented as a single uninterruptable CPU instruction, the RTC can be executed from anywhere in memory, including from a different page of extended memory in the overlay window.

The CALL and RTC instructions behave like JSR and RTS, except they have slightly longer execution times. Since extra execution cycles are required, routinely substituting CALL/RTC for JSR/RTS is not recommended. JSR and RTS can be used to access subroutines that are located outside the program overlay window or on the same memory page. However, if a subroutine can be called from other pages, it must be terminated with an RTC. In this case, since RTC unstacks the PPAGE value as well as the return address, all accesses to the subroutine, even those made from the same page, must use CALL instructions.

8.5 HCS08 Instruction Set Summary

Instruction Set Summary Nomenclature

The nomenclature listed here is used in the instruction descriptions in Table 8-2.

Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: "gets")
&	=	Boolean AND
|	=	Boolean OR
⊕	=	Boolean exclusive-OR
×	=	Multiply
÷	=	Divide
:	=	Concatenate
+	=	Add
–	=	Negate (two's complement)

CPU registers

A	=	Accumulator
CCR	=	Condition code register
H	=	Index register, higher order (most significant) 8 bits
X	=	Index register, lower order (least significant) 8 bits
PC	=	Program counter
PCH	=	Program counter, higher order (most significant) 8 bits
PCL	=	Program counter, lower order (least significant) 8 bits
SP	=	Stack pointer

Memory and addressing

M	=	A memory location or absolute data, depending on addressing mode
M:M + 0x0001	=	A 16-bit value in two consecutive memory locations. The higher-order (most significant) 8 bits are located at the address of M, and the lower-order (least significant) 8 bits are located at the next higher sequential address.

Condition code register (CCR) bits

V	=	Two's complement overflow indicator, bit 7
H	=	Half carry, bit 4
I	=	Interrupt mask, bit 3
N	=	Negative indicator, bit 2
Z	=	Zero indicator, bit 1
C	=	Carry/borrow, bit 0 (carry out of bit 7)

CCR activity notation

–	=	Bit not affected

0	=	Bit forced to 0
1	=	Bit forced to 1
↕	=	Bit set or cleared according to results of operation
U	=	Undefined after the operation

Machine coding notation

dd	=	Low-order 8 bits of a direct address 0x0000–0x00FF (high byte assumed to be 0x00)
ee	=	Upper 8 bits of 16-bit offset
ff	=	Lower 8 bits of 16-bit offset or 8-bit offset
ii	=	One byte of immediate data
jj	=	High-order byte of a 16-bit immediate data value
kk	=	Low-order byte of a 16-bit immediate data value
hh	=	High-order byte of 16-bit extended address
ll	=	Low-order byte of 16-bit extended address
pg	=	Page
rr	=	Relative offset

Source form

Everything in the source forms columns, *except expressions in italic characters,* is literal information that must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, and plus signs (+) are literal characters.

n	—	Any label or expression that evaluates to a single integer in the range 0–7
opr8i	—	Any label or expression that evaluates to an 8-bit immediate value
opr16i	—	Any label or expression that evaluates to a 16-bit immediate value
opr8a	—	Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low order 8 bits of an address in the direct page of the 64-Kbyte address space (0x00xx).
opr16a	—	Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64-Kbyte address space.
oprx8	—	Any label or expression that evaluates to an unsigned 8-bit value, used for indexed addressing
oprx16	—	Any label or expression that evaluates to a 16-bit value. Because the HCS08 has a 16-bit address bus, this can be either a signed or an unsigned value.
page	—	Any label or expression that evaluates to a valid bank number for the PPAGE register. For a 128-Kbyte deravative, any value between 0 and 7 is valid.
rel	—	Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

Address modes

INH	=	Inherent (no operands)

IMM = 8-bit or 16-bit immediate
DIR = 8-bit direct
EXT = 16-bit extended
IX = 16-bit indexed no offset
IX+ = 16-bit indexed no offset, post increment (CBEQ and MOV only)
IX1 = 16-bit indexed with 8-bit offset from H:X
IX1+ = 16-bit indexed with 8-bit offset, post increment (CBEQ only)
IX2 = 16-bit indexed with 16-bit offset from H:X
REL = 8-bit relative offset
SP1 = Stack pointer with 8-bit offset
SP2 = Stack pointer with 16-bit offset

Table 8-2. HCS08 Instruction Set Summary (Sheet 1 of 7)

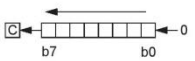
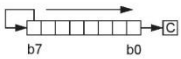
Source Form	Operation	Description	Effect on CCR					Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
ADC #*opr8i* ADC *opr8a* ADC *opr16a* ADC *oprx16*,X ADC *oprx8*,X ADC ,X ADC *oprx16*,SP ADC *oprx8*,SP	Add with Carry	A ← (A) + (M) + (C)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A9 B9 C9 D9 E9 F9 9ED9 9EE9	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
ADD #*opr8i* ADD *opr8a* ADD *opr16a* ADD *oprx16*,X ADD *oprx8*,X ADD ,X ADD *oprx16*,SP ADD *oprx8*,SP	Add without Carry	A ← (A) + (M)	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	AB BB CB DB EB FB 9EDB 9EEB	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
AIS #*opr8i*	Add Immediate Value (Signed) to Stack Pointer	SP ← (SP) + (M) M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	A7	ii	2
AIX #*opr8i*	Add Immediate Value (Signed) to Index Register (H:X)	H:X ← (H:X) + (M) M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	AF	ii	2
AND #*opr8i* AND *opr8a* AND *opr16a* AND *oprx16*,X AND *oprx8*,X AND ,X AND *oprx16*,SP AND *oprx8*,SP	Logical AND	A ← (A) & (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A4 B4 C4 D4 E4 F4 9ED4 9EE4	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
ASL *opr8a* ASLA ASLX ASL *oprx8*,X ASL ,X ASL *oprx8*,SP	Arithmetic Shift Left (Same as LSL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	5 1 1 5 4 6
ASR *opr8a* ASRA ASRX ASR *oprx8*,X ASR ,X ASR *oprx8*,SP	Arithmetic Shift Right		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	37 47 57 67 77 9E67	dd ff ff	5 1 1 5 4 6
BCC *rel*	Branch if Carry Bit Clear	Branch if (C) = 0	–	–	–	–	–	–	REL	24	rr	3

Table 8-2. HCS08 Instruction Set Summary (Sheet 2 of 7)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
BCLR *n,opr8a*	Clear Bit n in Memory	Mn ← 0	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5
BCS *rel*	Branch if Carry Bit Set (Same as BLO)	Branch if (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BEQ *rel*	Branch if Equal	Branch if (Z) = 1	–	–	–	–	–	–	REL	27	rr	3
BGE *rel*	Branch if Greater Than or Equal To (Signed Operands)	Branch if (N ⊕ V) = 0	–	–	–	–	–	–	REL	90	rr	3
BGND	Enter Active Background if ENBDM = 1	Waits For and Processes BDM Commands Until GO, TRACE1, or TAGGO	–	–	–	–	–	–	INH	82		5+
BGT *rel*	Branch if Greater Than (Signed Operands)	Branch if (Z) | (N ⊕ V) = 0	–	–	–	–	–	–	REL	92	rr	3
BHCC *rel*	Branch if Half Carry Bit Clear	Branch if (H) = 0	–	–	–	–	–	–	REL	28	rr	3
BHCS *rel*	Branch if Half Carry Bit Set	Branch if (H) = 1	–	–	–	–	–	–	REL	29	rr	3
BHI *rel*	Branch if Higher	Branch if (C) | (Z) = 0	–	–	–	–	–	–	REL	22	rr	3
BHS *rel*	Branch if Higher or Same (Same as BCC)	Branch if (C) = 0	–	–	–	–	–	–	REL	24	rr	3
BIH *rel*	Branch if IRQ Pin High	Branch if IRQ pin = 1	–	–	–	–	–	–	REL	2F	rr	3
BIL *rel*	Branch if IRQ Pin Low	Branch if IRQ pin = 0	–	–	–	–	–	–	REL	2E	rr	3
BIT #*opr8i* BIT *opr8a* BIT *opr16a* BIT *oprx16*,X BIT *oprx8*,X BIT ,X BIT *oprx16*,SP BIT *oprx8*,SP	Bit Test	(A) & (M) (CCR Updated but Operands Not Changed)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A5 B5 C5 D5 E5 F5 9ED5 9EE5	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
BLE *rel*	Branch if Less Than or Equal To (Signed Operands)	Branch if (Z) | (N ⊕ V) = 1	–	–	–	–	–	–	REL	93	rr	3
BLO *rel*	Branch if Lower (Same as BCS)	Branch if (C) = 1	–	–	–	–	–	–	REL	25	rr	3
BLS *rel*	Branch if Lower or Same	Branch if (C) | (Z) = 1	–	–	–	–	–	–	REL	23	rr	3
BLT *rel*	Branch if Less Than (Signed Operands)	Branch if (N ⊕ V) = 1	–	–	–	–	–	–	REL	91	rr	3
BMC *rel*	Branch if Interrupt Mask Clear	Branch if (I) = 0	–	–	–	–	–	–	REL	2C	rr	3
BMI *rel*	Branch if Minus	Branch if (N) = 1	–	–	–	–	–	–	REL	2B	rr	3
BMS *rel*	Branch if Interrupt Mask Set	Branch if (I) = 1	–	–	–	–	–	–	REL	2D	rr	3
BNE *rel*	Branch if Not Equal	Branch if (Z) = 0	–	–	–	–	–	–	REL	26	rr	3
BPL *rel*	Branch if Plus	Branch if (N) = 0	–	–	–	–	–	–	REL	2A	rr	3
BRA *rel*	Branch Always	No Test	–	–	–	–	–	–	REL	20	rr	3

Table 8-2. HCS08 Instruction Set Summary (Sheet 3 of 7)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
BRCLR *n,opr8a,rel*	Branch if Bit *n* in Memory Clear	Branch if (Mn) = 0	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	01 03 05 07 09 0B 0D 0F	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BRN *rel*	Branch Never	Uses 3 Bus Cycles	–	–	–	–	–	–	REL	21	rr	3
BRSET *n,opr8a,rel*	Branch if Bit *n* in Memory Set	Branch if (Mn) = 1	–	–	–	–	–	↕	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	00 02 04 06 08 0A 0C 0E	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	5 5 5 5 5 5 5 5
BSET *n,opr8a*	Set Bit *n* in Memory	Mn ← 1	–	–	–	–	–	–	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	10 12 14 16 18 1A 1C 1E	dd dd dd dd dd dd dd dd	5 5 5 5 5 5 5 5
BSR *rel*	Branch to Subroutine	PC ← (PC) + 0x0002 push (PCL); SP ← (SP) – 0x0001 push (PCH); SP ← (SP) – 0x0001 PC ← (PC) + *rel*	–	–	–	–	–	–	REL	AD	rr	5
CALL page, opr16a	Call Subroutine	PC ← PC + 4 Push (PCL); SP ← (SP) – 0x0001 Push (PCH); SP ← (SP) – 0x0001 Push (PPAGE); SP ← (SP) – 0x0001 PPAGE ← page PC ← Unconditional Address	–	–	–	–	–	–	EXT	AC	pghll	8
CBEQ *opr8a,rel* CBEQA *#opr8i,rel* CBEQX *#opr8i,rel* CBEQ *oprx8*,X+,*rel* CBEQ ,X+,*rel* CBEQ *oprx8*,SP,*rel*	Compare and Branch if Equal	Branch if (A) = (M) Branch if (A) = (M) Branch if (X) = (M) Branch if (A) = (M) Branch if (A) = (M) Branch if (A) = (M)	–	–	–	–	–	–	DIR IMM IMM IX1+ IX+ SP1	31 41 51 61 71 9E61	dd rr ii rr ii rr ff rr rr ff rr	5 4 4 5 5 6
CLC	Clear Carry Bit	C ← 0	–	–	–	–	–	0	INH	98		1
CLI	Clear Interrupt Mask Bit	I ← 0	–	–	0	–	–	–	INH	9A		1
CLR *opr8a* CLRA CLRX CLRH CLR *oprx8*,X CLR ,X CLR *oprx8*,SP	Clear	M ← 0x00 A ← 0x00 X ← 0x00 H ← 0x00 M ← 0x00 M ← 0x00 M ← 0x00	0	–	–	0	1	–	DIR INH INH INH IX1 IX SP1	3F 4F 5F 8C 6F 7F 9E6F	dd ff ff	5 1 1 1 5 4 6
CMP *#opr8i* CMP *opr8a* CMP *opr16a* CMP *oprx16*,X CMP *oprx8*,X CMP ,X CMP *oprx16*,SP CMP *oprx8*,SP	Compare Accumulator with Memory	(A) – (M) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A1 B1 C1 D1 E1 F1 9ED1 9EE1	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4

Table 8-2. HCS08 Instruction Set Summary (Sheet 4 of 7)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
COM *opr8a* COMA COMX COM *oprx8*,X COM ,X COM *oprx8*,SP	Complement (One's Complement)	M ← (M̄)= 0xFF – (M) A ← (Ā) = 0xFF – (A) X ← (X̄) = 0xFF – (X) M ← (M̄) = 0xFF – (M) M ← (M̄) = 0xFF – (M) M ← (M̄) = 0xFF – (M)	0	–	–	↕	↕	1	DIR INH INH IX1 IX SP1	33 43 53 63 73 9E63	dd ff ff	5 1 1 5 4 6
CPHX *opr16a* CPHX #*opr16i* CPHX *opr8a* CPHX *oprx8*,SP	Compare Index Register (H:X) with Memory	(H:X) – (M:M + 0x0001) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	EXT IMM DIR SP1	3E 65 75 9EF3	hh ll jj kk dd ff	6 3 5 6
CPX #*opr8i* CPX *opr8a* CPX *opr16a* CPX *oprx16*,X CPX *oprx8*,X CPX ,X CPX *oprx16*,SP CPX *oprx8*,SP	Compare X (Index Register Low) with Memory	(X) – (M) (CCR Updated But Operands Not Changed)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A3 B3 C3 D3 E3 F3 9ED3 9EE3	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
DAA	Decimal Adjust Accumulator After ADD or ADC of BCD Values	$(A)_{10}$	U	–	–	↕	↕	↕	INH	72		1
DBNZ *opr8a,rel* DBNZA *rel* DBNZX *rel* DBNZ *oprx8*,X,*rel* DBNZ ,X,*rel* DBNZ *oprx8*,SP,*rel*	Decrement and Branch if Not Zero	Decrement A, X, or M Branch if (result) ≠ 0 DBNZX Affects X Not H	–	–	–	–	–	–	DIR INH INH IX1 IX SP1	3B 4B 5B 6B 7B 9E6B	dd rr rr rr ff rr rr ff rr	7 4 4 7 6 8
DEC *opr8a* DECA DECX DEC *oprx8*,X DEC ,X DEC *oprx8*,SP	Decrement	M ← (M) – 0x01 A ← (A) – 0x01 X ← (X) – 0x01 M ← (M) – 0x01 M ← (M) – 0x01 M ← (M) – 0x01	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3A 4A 5A 6A 7A 9E6A	dd ff ff	5 1 1 5 4 6
DIV	Divide	A ← (H:A)÷(X) H ← Remainder	–	–	–	–	↕	↕	INH	52		6
EOR #*opr8i* EOR *opr8a* EOR *opr16a* EOR *oprx16*,X EOR *oprx8*,X EOR ,X EOR *oprx16*,SP EOR *oprx8*,SP	Exclusive OR Memory with Accumulator	A ← (A ⊕ M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A8 B8 C8 D8 E8 F8 9ED8 9EE8	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
INC *opr8a* INCA INCX INC *oprx8*,X INC ,X INC *oprx8*,SP	Increment	M ← (M) + 0x01 A ← (A) + 0x01 X ← (X) + 0x01 M ← (M) + 0x01 M ← (M) + 0x01 M ← (M) + 0x01	↕	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3C 4C 5C 6C 7C 9E6C	dd ff ff	5 1 1 5 4 6
JMP *opr8a* JMP *opr16a* JMP *oprx16*,X JMP *oprx8*,X JMP ,X	Jump	PC ← Jump Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BC CC DC EC FC	dd hh ll ee ff ff 	3 4 4 3 3
JSR *opr8a* JSR *opr16a* JSR *oprx16*,X JSR *oprx8*,X JSR ,X	Jump to Subroutine	PC ← (PC) + *n* (*n* = 1, 2, or 3) Push (PCL); SP ← (SP) – 0x0001 Push (PCH); SP ← (SP) – 0x0001 PC ← Unconditional Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BD CD DD ED FD	dd hh ll ee ff ff 	5 6 6 5 5

Table 8-2. HCS08 Instruction Set Summary (Sheet 5 of 7)

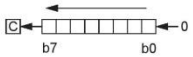
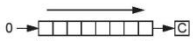
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
LDA #*opr8i* LDA *opr8a* LDA *opr16a* LDA *oprx16*,X LDA *oprx8*,X LDA ,X LDA *oprx16*,SP LDA *oprx8*,SP	Load Accumulator from Memory	A ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A6 B6 C6 D6 E6 F6 9ED6 9EE6	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
LDHX #*opr16i* LDHX *opr8a* LDHX *opr16a* LDHX ,X LDHX *oprx16*,X LDHX *oprx8*,X LDHX *oprx8*,SP	Load Index Register (H:X) from Memory	H:X ← (M:M + 0x0001)	0	–	–	↕	↕	–	IMM DIR EXT IX IX2 IX1 SP1	45 55 32 9EAE 9EBE 9ECE 9EFE	jj kk dd hh ll ee ff ff ff	3 4 5 5 6 5 5
LDX #*opr8i* LDX *opr8a* LDX *opr16a* LDX *oprx16*,X LDX *oprx8*,X LDX ,X LDX *oprx16*,SP LDX *oprx8*,SP	Load X (Index Register Low) from Memory	X ← (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	AE BE CE DE EE FE 9EDE 9EEE	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
LSL *opr8a* LSLA LSLX LSL *oprx8*,X LSL ,X LSL *oprx8*,SP	Logical Shift Left (Same as ASL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff	5 1 1 5 4 6
LSR *opr8a* LSRA LSRX LSR *oprx8*,X LSR ,X LSR *oprx8*,SP	Logical Shift Right		↕	–	–	0	↕	↕	DIR INH INH IX1 IX SP1	34 44 54 64 74 9E64	dd ff ff	5 1 1 5 4 6
MOV *opr8a*,*opr8a* MOV *opr8a*,X+ MOV #*opr8i*,*opr8a* MOV ,X+,*opr8a*	Move	$(M)_{destination} \leftarrow (M)_{source}$ H:X ← (H:X) + 0x0001 in IX+/DIR and DIR/IX+ Modes	0	–	–	↕	↕	–	DIR/DIR DIR/IX+ IMM/DIR IX+/DIR	4E 5E 6E 7E	dd dd dd ii dd dd	5 5 4 5
MUL	Unsigned multiply	X:A ← (X) × (A)	–	0	–	–	–	0	INH	42		5
NEG *opr8a* NEGA NEGX NEG *oprx8*,X NEG ,X NEG *oprx8*,SP	Negate (Two's Complement)	M ← – (M) = 0x00 – (M) A ← – (A) = 0x00 – (A) X ← – (X) = 0x00 – (X) M ← – (M) = 0x00 – (M) M ← – (M) = 0x00 – (M) M ← – (M) = 0x00 – (M)		–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	30 40 50 60 70 9E60	dd ff ff	5 1 1 5 4 6
NOP	No Operation	Uses 1 Bus Cycle	–	–	–	–	–	–	INH	9D		1
NSA	Nibble Swap Accumulator	A ← (A[3:0]:A[7:4])	–	–	–	–	–	–	INH	62		1
ORA #*opr8i* ORA *opr8a* ORA *opr16a* ORA *oprx16*,X ORA *oprx8*,X ORA ,X ORA *oprx16*,SP ORA *oprx8*,SP	Inclusive OR Accumulator and Memory	A ← (A) | (M)	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	AA BA CA DA EA FA 9EDA 9EEA	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
PSHA	Push Accumulator onto Stack	Push (A); SP ← (SP) – 0x0001	–	–	–	–	–	–	INH	87		2
PSHH	Push H (Index Register High) onto Stack	Push (H); SP ← (SP) – 0x0001	–	–	–	–	–	–	INH	8B		2
PSHX	Push X (Index Register Low) onto Stack	Push (X); SP ← (SP) – 0x0001	–	–	–	–	–	–	INH	89		2

Table 8-2. HCS08 Instruction Set Summary (Sheet 6 of 7)

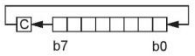
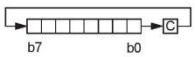
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
PULA	Pull Accumulator from Stack	SP ← (SP + 0x0001); Pull (A)	–	–	–	–	–	–	INH	86		3
PULH	Pull H (Index Register High) from Stack	SP ← (SP + 0x0001); Pull (H)	–	–	–	–	–	–	INH	8A		3
PULX	Pull X (Index Register Low) from Stack	SP ← (SP + 0x0001); Pull (X)	–	–	–	–	–	–	INH	88		3
ROL *oprx8*,X ROL ,X ROL *oprx8*,SP	Rotate Left through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	39 49 59 69 79 9E69	dd ff ff	5 1 1 5 4 6
ROR *oprx8*,X ROR ,X ROR *oprx8*,SP	Rotate Right through Carry		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	36 46 56 66 76 9E66	dd ff ff	5 1 1 5 4 6
RSP	Reset Stack Pointer	SP ← 0xFF (High Byte Not Affected)	–	–	–	–	–	–	INH	9C		1
RTC	Return fom CALL	SP ← (SP) + 0x0001; Pull (PPAGE) SP ← (SP) + 0x0001; Pull (PCH) SP ← (SP) + 0x0001; Pull (PCL)	–	–	–	–	–	–	INH	8D		7
RTI	Return from Interrupt	SP ← (SP) + 0x0001; Pull (CCR) SP ← (SP) + 0x0001; Pull (A) SP ← (SP) + 0x0001; Pull (X) SP ← (SP) + 0x0001; Pull (PCH) SP ← (SP) + 0x0001; Pull (PCL)	↕	↕	↕	↕	↕	↕	INH	80		9
RTS	Return from Subroutine	SP ← SP + 0x0001; Pull (PCH) SP ← SP + 0x0001; Pull (PCL)	–	–	–	–	–	–	INH	81		6
SBC #*opr8i* SBC *opr8a* SBC *opr16a* SBC *oprx16*,X SBC *oprx8*,X SBC ,X SBC *oprx16*,SP SBC *oprx8*,SP	Subtract with Carry	A ← (A) – (M) – (C)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A2 B2 C2 D2 E2 F2 9ED2 9EE2	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
SEC	Set Carry Bit	C ← 1	–	–	–	–	–	1	INH	99		1
SEI	Set Interrupt Mask Bit	I ← 1	–	–	1	–	–	–	INH	9B		1
STA *opr8a* STA *opr16a* STA *oprx16*,X STA *oprx8*,X STA ,X STA *oprx16*,SP STA *oprx8*,SP	Store Accumulator in Memory	M ← (A)	0	–	–	↕	↕	–	DIR EXT IX2 IX1 IX SP2 SP1	B7 C7 D7 E7 F7 9ED7 9EE7	dd hh ll ee ff ff ee ff ff	3 4 4 3 2 5 4
STHX *opr8a* STHX *opr16a* STHX *oprx8*,SP	Store H:X (Index Reg.)	(M:M + 0x0001) ← (H:X)	0	–	–	↕	↕	–	DIR EXT SP1	35 96 9EFF	dd hh ll ff	4 5 5
STOP	Enable Interrupts: Stop Processing Refer to MCU Documentation	I bit ← 0; Stop Processing	–	–	0	–	–	–	INH	8E		2+
STX *opr8a* STX *opr16a* STX *oprx16*,X STX *oprx8*,X STX ,X STX *oprx16*,SP STX *oprx8*,SP	Store X (Low 8 Bits of Index Register) in Memory	M ← (X)	0	–	–	↕	↕	–	DIR EXT IX2 IX1 IX SP2 SP1	BF CF DF EF FF 9EDF 9EEF	dd hh ll ee ff ff ee ff ff	3 4 4 3 2 5 4

Table 8-2. HCS08 Instruction Set Summary (Sheet 7 of 7)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Bus Cycles[1]
			V	H	I	N	Z	C				
SUB *#opr8i* SUB *opr8a* SUB *opr16a* SUB *oprx16*,X SUB *oprx8*,X SUB ,X SUB *oprx16*,SP SUB *oprx8*,SP	Subtract	A ← (A) – (M)	↕	–	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A0 B0 C0 D0 E0 F0 9ED0 9EE0	ii dd hh ll ee ff ff ee ff ff	2 3 4 4 3 3 5 4
SWI	Software Interrupt	PC ← (PC) + 0x0001 Push (PCL); SP ← (SP) – 0x0001 Push (PCH); SP ← (SP) – 0x0001 Push (X); SP ← (SP) – 0x0001 Push (A); SP ← (SP) – 0x0001 Push (CCR); SP ← (SP) – 0x0001 I ← 1; PCH ← Interrupt Vector High Byte PCL ← Interrupt Vector Low Byte	–	–	1	–	–	–	INH	83		11
TAP	Transfer Accumulator to CCR	CCR ← (A)	↕	↕	↕	↕	↕	↕	INH	84		1
TAX	Transfer Accumulator to X (Index Register Low)	X ← (A)	–	–	–	–	–	–	INH	97		1
TPA	Transfer CCR to Accumulator	A ← (CCR)	–	–	–	–	–	–	INH	85		1
TST *opr8a* TSTA TSTX TST *oprx8*,X TST ,X TST *oprx8*,SP	Test for Negative or Zero	(M) – 0x00 (A) – 0x00 (X) – 0x00 (M) – 0x00 (M) – 0x00 (M) – 0x00	0	–	–	↕	↕	–	DIR INH INH IX1 IX SP1	3D 4D 5D 6D 7D 9E6D	dd ff ff	4 1 1 4 3 5
TSX	Transfer SP to Index Reg.	H:X ← (SP) + 0x0001	–	–	–	–	–	–	INH	95		2
TXA	Transfer X (Index Reg. Low) to Accumulator	A ← (X)	–	–	–	–	–	–	INH	9F		1
TXS	Transfer Index Reg. to SP	SP ← (H:X) – 0x0001	–	–	–	–	–	–	INH	94		2
WAIT	Enable Interrupts; Wait for Interrupt	I bit ← 0; Halt CPU	–	–	0	–	–	–	INH	8F		2+

[1] Bus clock frequency is one-half of the CPU clock frequency.

Table 8-3. Opcode Map (Sheet 1 of 2)

Bit-Manipulation		Branch	Read-Modify-Write					Control		Register/Memory					
00 5 BRSET0 3 DIR	10 5 BSET0 2 DIR	20 3 BRA 2 REL	30 5 NEG 2 DIR	40 1 NEGA 1 INH	50 1 NEGX 1 INH	60 5 NEG 2 IX1	70 4 NEG 1 IX	80 9 RTI 1 INH	90 3 BGE 2 REL	A0 2 SUB 2 IMM	B0 3 SUB 2 DIR	C0 4 SUB 3 EXT	D0 4 SUB 3 IX2	E0 3 SUB 2 IX1	F0 3 SUB 1 IX
01 5 BRCLR0 3 DIR	11 5 BCLR0 2 DIR	21 3 BRN 2 REL	31 5 CBEQ 3 DIR	41 4 CBEQA 3 IMM	51 4 CBEQX 3 IMM	61 5 CBEQ 3 IX1+	71 5 CBEQ 2 IX+	81 6 RTS 1 INH	91 3 BLT 2 REL	A1 2 CMP 2 IMM	B1 3 CMP 2 DIR	C1 4 CMP 3 EXT	D1 4 CMP 3 IX2	E1 3 CMP 2 IX1	F1 3 CMP 1 IX
02 5 BRSET1 3 DIR	12 5 BSET1 2 DIR	22 3 BHI 2 REL	32 5 LDHX 3 EXT	42 5 MUL 1 INH	52 6 DIV 1 INH	62 1 NSA 1 INH	72 1 DAA 1 INH	82 5+ BGND 1 INH	92 3 BGT 2 REL	A2 2 SBC 2 IMM	B2 3 SBC 2 DIR	C2 4 SBC 3 EXT	D2 4 SBC 3 IX2	E2 3 SBC 2 IX1	F2 3 SBC 1 IX
03 5 BRCLR1 3 DIR	13 5 BCLR1 2 DIR	23 3 BLS 2 REL	33 5 COM 2 DIR	43 1 COMA 1 INH	53 1 COMX 1 INH	63 5 COM 2 IX1	73 4 COM 1 IX	83 11 SWI 1 INH	93 3 BLE 2 REL	A3 2 CPX 2 IMM	B3 3 CPX 2 DIR	C3 4 CPX 3 EXT	D3 4 CPX 3 IX2	E3 3 CPX 2 IX1	F3 3 CPX 1 IX
04 5 BRSET2 3 DIR	14 5 BSET2 2 DIR	24 3 BCC 2 REL	34 5 LSR 2 DIR	44 1 LSRA 1 INH	54 1 LSRX 1 INH	64 5 LSR 2 IX1	74 4 LSR 1 IX	84 1 TAP 1 INH	94 2 TXS 1 INH	A4 2 AND 2 IMM	B4 3 AND 2 DIR	C4 4 AND 3 EXT	D4 4 AND 3 IX2	E4 3 AND 2 IX1	F4 3 AND 1 IX
05 5 BRCLR2 3 DIR	15 5 BCLR2 2 DIR	25 3 BCS 2 REL	35 4 STHX 2 DIR	45 3 LDHX 3 IMM	55 4 LDHX 2 DIR	65 3 CPHX 3 IMM	75 5 CPHX 2 DIR	85 1 TPA 1 INH	95 2 TSX 1 INH	A5 2 BIT 2 IMM	B5 3 BIT 2 DIR	C5 4 BIT 3 EXT	D5 4 BIT 3 IX2	E5 3 BIT 2 IX1	F5 3 BIT 1 IX
06 5 BRSET3 3 DIR	16 5 BSET3 2 DIR	26 3 BNE 2 REL	36 5 ROR 2 DIR	46 1 RORA 1 INH	56 1 RORX 1 INH	66 5 ROR 2 IX1	76 4 ROR 1 IX	86 3 PULA 1 INH	96 5 STHX 3 EXT	A6 2 LDA 2 IMM	B6 3 LDA 2 DIR	C6 4 LDA 3 EXT	D6 4 LDA 3 IX2	E6 3 LDA 2 IX1	F6 3 LDA 1 IX
07 5 BRCLR3 3 DIR	17 5 BCLR3 2 DIR	27 3 BEQ 2 REL	37 5 ASR 2 DIR	47 1 ASRA 1 INH	57 1 ASRX 1 INH	67 5 ASR 2 IX1	77 4 ASR 1 IX	87 2 PSHA 1 INH	97 1 TAX 1 INH	A7 2 AIS 2 IMM	B7 3 STA 2 DIR	C7 4 STA 3 EXT	D7 4 STA 3 IX2	E7 3 STA 2 IX1	F7 2 STA 1 IX
08 5 BRSET4 3 DIR	18 5 BSET4 2 DIR	28 3 BHCC 2 REL	38 5 LSL 2 DIR	48 1 LSLA 1 INH	58 1 LSLX 1 INH	68 5 LSL 2 IX1	78 4 LSL 1 IX	88 3 PULX 1 INH	98 1 CLC 1 INH	A8 2 EOR 2 IMM	B8 3 EOR 2 DIR	C8 4 EOR 3 EXT	D8 4 EOR 3 IX2	E8 3 EOR 2 IX1	F8 3 EOR 1 IX
09 5 BRCLR4 3 DIR	19 5 BCLR4 2 DIR	29 3 BHCS 2 REL	39 5 ROL 2 DIR	49 1 ROLA 1 INH	59 1 ROLX 1 INH	69 5 ROL 2 IX1	79 4 ROL 1 IX	89 2 PSHX 1 INH	99 1 SEC 1 INH	A9 2 ADC 2 IMM	B9 3 ADC 2 DIR	C9 4 ADC 3 EXT	D9 4 ADC 3 IX2	E9 3 ADC 2 IX1	F9 3 ADC 1 IX
0A 5 BRSET5 3 DIR	1A 5 BSET5 2 DIR	2A 3 BPL 2 REL	3A 5 DEC 2 DIR	4A 1 DECA 1 INH	5A 1 DECX 1 INH	6A 5 DEC 2 IX1	7A 4 DEC 1 IX	8A 3 PULH 1 INH	9A 1 CLI 1 INH	AA 2 ORA 2 IMM	BA 3 ORA 2 DIR	CA 4 ORA 3 EXT	DA 4 ORA 3 IX2	EA 3 ORA 2 IX1	FA 3 ORA 1 IX
0B 5 BRCLR5 3 DIR	1B 5 BCLR5 2 DIR	2B 3 BMI 2 REL	3B 7 DBNZ 3 DIR	4B 4 DBNZA 2 INH	5B 4 DBNZX 2 INH	6B 7 DBNZ 3 IX1	7B 6 DBNZ 2 IX	8B 2 PSHH 1 INH	9B 1 SEI 1 INH	AB 2 ADD 2 IMM	BB 3 ADD 2 DIR	CB 4 ADD 3 EXT	DB 4 ADD 3 IX2	EB 3 ADD 2 IX1	FB 3 ADD 1 IX
0C 5 BRSET6 3 DIR	1C 5 BSET6 2 DIR	2C 3 BMC 2 REL	3C 5 INC 2 DIR	4C 1 INCA 1 INH	5C 1 INCX 1 INH	6C 5 INC 2 IX1	7C 4 INC 1 IX	8C 1 CLRH 1 INH	9C 1 RSP 1 INH	AC 8 CALL 4 EXT	BC 3 JMP 2 DIR	CC 4 JMP 3 EXT	DC 4 JMP 3 IX2	EC 3 JMP 2 IX1	FC 3 JMP 1 IX
0D 5 BRCLR6 3 DIR	1D 5 BCLR6 2 DIR	2D 3 BMS 2 REL	3D 4 TST 2 DIR	4D 1 TSTA 1 INH	5D 1 TSTX 1 INH	6D 4 TST 2 IX1	7D 3 TST 1 IX	8D 7 RTC 1 INH	9D 1 NOP 1 INH	AD 5 BSR 2 REL	BD 5 JSR 2 DIR	CD 6 JSR 3 EXT	DD 6 JSR 3 IX2	ED 5 JSR 2 IX1	FD 5 JSR 1 IX
0E 5 BRSET7 3 DIR	1E 5 BSET7 2 DIR	2E 3 BIL 2 REL	3E 6 CPHX 3 EXT	4E 5 MOV 3 DD	5E 5 MOV 2 DIX+	6E 4 MOV 3 IMD	7E 5 MOV 2 IX+D	8E 2+ STOP 1 INH	9E Page 2	AE 2 LDX 2 IMM	BE 3 LDX 2 DIR	CE 4 LDX 3 EXT	DE 4 LDX 3 IX2	EE 3 LDX 2 IX1	FE 3 LDX 1 IX
0F 5 BRCLR7 3 DIR	1F 5 BCLR7 2 DIR	2F 3 BIH 2 REL	3F 5 CLR 2 DIR	4F 1 CLRA 1 INH	5F 1 CLRX 1 INH	6F 5 CLR 2 IX1	7F 4 CLR 1 IX	8F 2+ WAIT 1 INH	9F 1 TXA 1 INH	AF 2 AIX 2 IMM	BF 3 STX 2 DIR	CF 4 STX 3 EXT	DF 4 STX 3 IX2	EF 3 STX 2 IX1	FF 2 STX 1 IX

INH	Inherent	REL	Relative	SP1	Stack Pointer, 8-Bit Offset
IMM	Immediate	IX	Indexed, No Offset	SP2	Stack Pointer, 16-Bit Offset
DIR	Direct	IX1	Indexed, 8-Bit Offset	IX+	Indexed, No Offset with Post Increment
EXT	Extended	IX2	Indexed, 16-Bit Offset	IX1+	Indexed, 1-Byte Offset with Post Increment
DD	DIR to DIR	IMD	IMM to DIR		
IX+D	IX+ to DIR	DIX+	DIR to IX+		

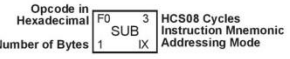


Table 8-3. Opcode Map (Sheet 2 of 2)

Bit-Manipulation		Branch	Read-Modify-Write					Control		Register/Memory					
						9E60 6 NEG 3 SP1							9ED0 5 SUB 4 SP2	9EE0 4 SUB 3 SP1	
						9E61 6 CBEQ 4 SP1							9ED1 5 CMP 4 SP2	9EE1 4 CMP 3 SP1	
													9ED2 5 SBC 4 SP2	9EE2 4 SBC 3 SP1	
						9E63 6 COM 3 SP1							9ED3 5 CPX 4 SP2	9EE3 4 CPX 3 SP1	9EF3 6 CPHX 3 SP1
						9E64 6 LSR 3 SP1							9ED4 5 AND 4 SP2	9EE4 4 AND 3 SP1	
													9ED5 5 BIT 4 SP2	9EE5 4 BIT 3 SP1	
						9E66 6 ROR 3 SP1							9ED6 5 LDA 4 SP2	9EE6 4 LDA 3 SP1	
						9E67 6 ASR 3 SP1							9ED7 5 STA 4 SP2	9EE7 4 STA 3 SP1	
						9E68 6 LSL 3 SP1							9ED8 5 EOR 4 SP2	9EE8 4 EOR 3 SP1	
						9E69 6 ROL 3 SP1							9ED9 5 ADC 4 SP2	9EE9 4 ADC 3 SP1	
						9E6A 6 DEC 3 SP1							9EDA 5 ORA 4 SP2	9EEA 4 ORA 3 SP1	
						9E6B 8 DBNZ 4 SP1							9EDB 5 ADD 4 SP2	9EEB 4 ADD 3 SP1	
						9E6C 6 INC 3 SP1									
						9E6D 5 TST 3 SP1									
										9EAE 5 LDHX 2 IX	9EBE 6 LDHX 4 IX2	9ECE 5 LDHX 3 IX1	9EDE 5 LDX 4 SP2	9EEE 4 LDX 3 SP1	9EFE 5 LDHX 3 SP1
						9E6F 6 CLR 3 SP1							9EDF 5 STX 4 SP2	9EEF 4 STX 3 SP1	9EFF 5 STHX 3 SP1

INH	Inherent	REL	Relative	SP1	Stack Pointer, 8-Bit Offset
IMM	Immediate	IX	Indexed, No Offset	SP2	Stack Pointer, 16-Bit Offset
DIR	Direct	IX1	Indexed, 8-Bit Offset	IX+	Indexed, No Offset with Post Increment
EXT	Extended	IX2	Indexed, 16-Bit Offset	IX1+	Indexed, 1-Byte Offset with Post Increment
DD	DIR to DIR	IMD	IMM to DIR		
IX+D	IX+ to DIR	DIX+	DIR to IX+		

Note: All Sheet 2 Opcodes are Preceded by the Page 2 Prebyte (9E)