3

Projeto e Implementação de Arquiteturas Superescalares

Rafael Ramos dos Santos[1] *(Universidade de Santa Cruz do Sul, rsantos@unisc.br)*
Tatiana G. S. dos Santos[2] *(Universidade de Santa Cruz do Sul, tatianas@unisc.br)*

Resumo:

As Arquiteturas Superescalares dominam quase que totalmente o mercado de microprocessadores de propósito geral. O que torna essa abordagem tão atrativa são as múltiplas unidades funcionais capazes de executar e completar diversas instruções em um único ciclo de relógio. Tal abordagem, contudo, sofre com problemas que surgem, pricipalmente, em conseqüência dessa capacidade de execução concorrente.

A dificuldade encontra-se, sobretudo, na implemetação de mecanismos para solucionar problemas de dependências entre as instruções. É comum que duas ou mais instruções dependam entre si, produzindo uma seqüência difícil de ser executada concorrentemente. Assim sendo, a arquitetura deve prover alternativas para que o *pipeline* não fique parado a espera de uma instrução específica. Porém, cada um desses mecanimos apresenta um alto custo lógico, fazendo com que o projeto desses processadores torne-se muito complexo e sofisticado. O ciclo completo de desenvolvimento de uma arquitetura superescalar pode durar anos e esse é um grande desafio ainda a ser vencido. Desenvolver uma arquitetura para suprir efetivamente a necessidade de usuários e aplicações ainda não conhecidos não é uma tarefa trivial.

O principal objetivo desse curso é apresentar a estrutura básica de um processador superescalar, suas características e problemas estruturais, bem como alternativas para solucionar esses problemas. Além disso, as etapas de desenvolvimento de um projeto típico também serão abordadas, assim como as novas tendências de mercado.

[1]Professor/UNISC, Depto. de Informática, Av. Independência, 2293 – Santa Cruz do Sul, RS
[2]Professora/UNISC, Depto. de Informática, Av. Independência, 2293 – Santa Cruz do Sul, RS

3.1. Introdução

Para satisfazer consumidores cada vez mais exigentes, a indústria de microprocessadores enfrenta uma constante busca por aumento de desempenho. Aplicações cada vez maiores e mais complexas forçam o rápido desenvolvimento de técnicas e alternativas que explorem eficientemente o paralelismo ao nível de instrução (ILP – *Instruction Level Parallelism*) e produzam resultados satisfatórios, com o melhor aproveitamento possível.

Já há alguns anos o mercado está tomado pelas conhecidas arquiteturas superescalares [JOH 91, HEN 2003]. Esse tipo de arquitetura tem o potencial de executar várias instruções por ciclo de relógio, através do emprego de várias unidades funcionais (UFs). Microprocessadores como o Alpha 21264 [KES 99], o UltraSparc III [HOR 99] e o Pentium 4 [INT 2001], utilizam o modelo superescalar nas suas microarquiteturas.

3.2. O *pipeline* superescalar de instruções

Mesmo sendo preferência na maioria dos projetos do estado da arte, as arquiteturas superescalares possuem alguns problemas que diminuem seu desempenho potencial causado por uma sub-utilização dos recursos. A maioria desses problemas estão relacionados com as dependências de dados e de controle existentes nos programas executados.

Essa seção concentra-se no estudo da estrutura básica de um *pipeline* superescalar. Com isso, pretende-se introduzir não apenas a funcionalidade de cada estágio, mas também discutir-se os problemas que limitam o paralelismo e, conseqüentemente, reduzem o número de instruções executadas por ciclo.

Tipicamente um *pipeline* superescalar compreende os estágios de busca/previsão, decodificação, despacho, delegação, execução, escrita e compleção. No entanto, cada microprocessador possui peculiaridades que vão desde a variação no número de estágios até o emprego de mecanismos inovadores e sofisticados de previsão [KES 99], pré-busca [SAN 2000], *trace cache* [INT 2001, ROT 97], entre outros. Nessa seção discute-se o modelo e problemas ligados ao *pipeline* básico de instruções.

A Figura 3.1 apresenta um esquema básico de um *pipeline* superescalar. Nota-se que o estágio de execução apresenta várias unidades funcionais que processam instruções paralelamente. Idealmente, após N ciclos, onde N é o número de estágios, M instruções serão concluídas a cada ciclo, sendo M o número de unidades funcionais. No entanto, em conseqüência dos problemas discutidos em breve, o número de instruções executadas em cada ciclo fica em geral bastante abaixo do número máximo. Microprocessadores atuais, que possuem oito, ou mais unidades funcionais não conseguem executar, em média, mais do que duas instruções por ciclo.

3.3. Estágio de busca/previsão

O primeiro estágio é conhecido como estágio de busca de instruções e tem como principal objetivo alimentar o *pipeline* com instruções trazidas do sistema de memória. Nesse mesmo estágio, a previsão de desvios é realizada com o intuito de evitar que a busca pare para esperar os resultados produzidos por tais instruções.

Dessa forma, as atividades realizadas nessa primeira etapa resumem-se em:

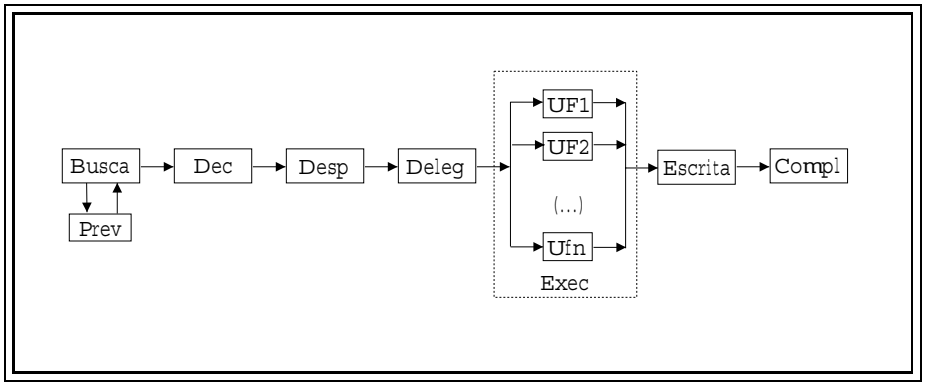

Figura 3.1: Esquema de um *pipeline* superescalar.

- Buscar um bloco de instruções do sub-sistema de memória;
- Reconhecer a existência de um desvio na linha buscada;
- Prever o desvio, se encontrado;
- Calcular o endereço previsto e, finalmente;
- Redirecionar a busca no caso de desvio previsto como tomado.

É possível observar que o trabalho mais significativo e crítico é a detecção bem como a previsão de instruções de desvio. Sendo assim, as tarefas dos itens (3) e (4) são totalmente executadas pela unidade de previsão da arquitetura.

O problema das dependências de controle é antigo e não existe solução definitiva. A previsão de desvios tenta acertar o endereço da instrução alvo de um desvio, baseado geralmente, em execuções anteriores desse mesmo desvio. Contudo, sabe-se que esse recurso nem sempre é suficiente e a ocorrência de uma pequena percentagem de falha no mecanismo de previsão é suficiente para reduzir drasticamente o desempenho global da arquitetura.

O mecanismo de busca acessa a memória *cache* nível 1 de instruções e traz um número de instruções equivalente à largura de busca (tamanho da linha da cache). Caso haja coincidência (*hit*), as instruções requisitadas pela busca são disponibilizadas pela *I-cache* no ciclo seguinte. Caso haja falha na *I-cache*, as instruções só são disponibilizadas após a sua resolução. Assim, o estágio de busca fica parado até que as instruções requisitadas sejam trazidas dos níveis superiores da hierarquia de memória. A ocorrência de falhas na *I-cache* é também um problema grave nessas arquiteturas que necessitam de um alto fluxo de instruções para manter as unidades funcionais ocupadas. A saída adotada pelos microprocessadores superescalares atuais é quase sempre o mesmo: aumentar o tamanho da *cache* nível 1. Assim, o número de falhas é reduzido e o desempenho melhorado. É possível notar que esse solução pode não representar a melhor alternativa, sobretudo nas próximas gerações. O desafio de buscar uma linha dentro de uma *cache* excessivamente grande em uma fração de segundo pode ser impraticável. Alternativas relacionadas a técnicas de redução de falhas e mascaramento de latências de acesso devem ser adotadas [SAN 2000]. Além disso, a integração de mais de um nível da hierarquia *on-chip* é uma tendência que já vem sendo empregada em microarquiteturas mais modernas.

Com a disponibilização das instruções, a unidade de busca procura eventuais desvios dentro da linha de *cache* buscada. Caso não hajam desvios, todas as instruções são encaminhadas para o próximo estágio do *pipeline*. No entanto, se um desvio é detectado, o mecanismo de previsão é acionado. Se o mecanismo de previsão prevê que o desvio é não tomado, as instruções buscadas anteriormente continuam sendo encaminhadas especulativamente para os próximos estágios do *pipeline*. Caso o desvio seja previsto como tomado, as instruções que se localizam após o desvio são descartadas, e uma nova linha, baseada no endereço previsto, é buscada no próximo ciclo e executada especulativamente. Observa-se que em ambos os casos, as instruções subseqüentes a um desvio são especulativas, ou seja, o *pipeline* continua buscando, decodificando, despachando, delegando e executando instruções que podem não ser corretas. Esse é o grande problema da ocorrência de desvios. Para cada desvio previsto incorretamente, existe uma grande penalidade a ser cumprida pelo *pipeline*. Quando acontecem previsões incorretas, os resultados computados especulativamente devem ser desfeitos, a busca deve ser redirecionada e as instruções após o desvio previsto incorretamente devem ser descartadas. É importante observar também que são necessários ciclos adicionais para que o estágio de busca consiga preencher o *pipeline* novamente.

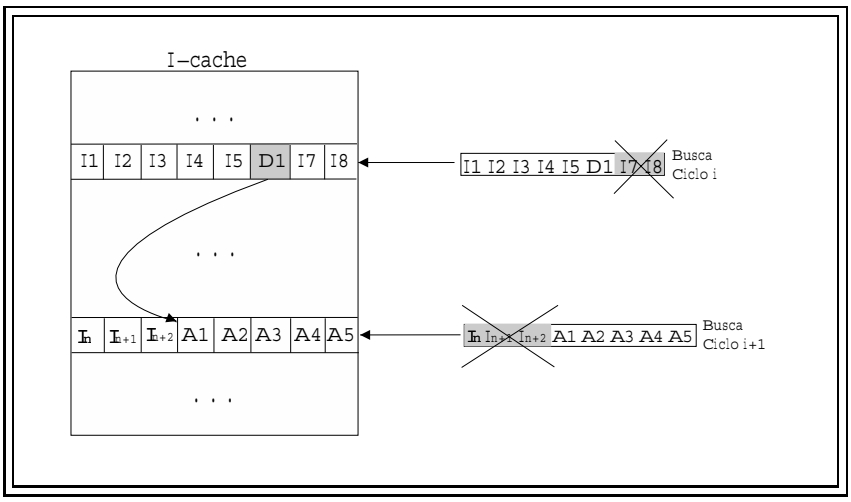

Figura 3.2: Problema do alinhamento da *I-cache*.

Além da penalidade, os desvios também contribuem para outro tipo de problema relacionado ao alinhamento da *cache* nível 1. Na ocorrência de um desvio tomado, a instrução alvo não encontra-se obrigatoriamente na primeira posição de uma linha de *cache*. Nesse caso, a unidade de busca aproveita apenas parte da linha e a largura de busca disponível não é completamente utilizada. Mesmo que o desvio tenha sido previsto corretamente, a sua simples ocorrência pode comprometer o andamento da execução de algum modo.

A Figura 3.2 ilustra o problema apresentado. A arquitetura tem largura de busca igual a oito instruções e a linha trazida da *cache* em determinado ciclo tem uma instrução *D1* de desvio. Quando a instrução *D1* é detectada como um desvio condicional e tem previsão de tomado, todas as instruções trazidas após esse desvio são descartadas, pois o alvo de *D1* é a instrução *A1*, encontrada em outra linha da *cache*. Assim, a busca redireciona seu alvo para *A1* e a *I-cache* é acessada novamente. Todas as instruções anteriores a ela são descartadas. O caso extremo é verificado se *A1* encontra-se na última

posição da linha. A largura de busca, que tem o potencial de buscar oito instruções, reduz-se à busca de apenas uma. Nota-se que o problema é ainda mais grave se o caminho for previsto incorretamente. Nesse caso, a linha que foi descartada anteriormente, com as instruções subseqüentes a *D1*, deve ser acessada novamente. Nesse segundo acesso, entretanto, as instruções anteriores a *D1* serão descartadas, sub-utilizando mais uma vez a largura disponível para a busca.

Um esforço conjunto da indústria e pesquisadores é efetuado continuamente com a finalidade de atenuar as penalidades sofridas pelo estágio de busca. O principal problema é que cada parada do estágio de busca traz como conseqüência o esvaziamento total ou parcial do *pipeline*, como discutido previamente. Busca de múltiplos fluxos [SAN 97, SKA 99], *trace cache* [ROT 96, ROT 97], execução simultânea de *threads* [NEM 90, GON 2000], são exemplos de alternativas para alguns desses problemas.

3.4. Estágio de decodificação

Basicamente, o estágio de decodificação tem como objetivo definir três elementos necessários à execução. São eles:

- A operação a ser executada;
- O endereço/local dos operandos necessários, e;
- O endereço/local de armazenamento do resultado da operação.

Esse estágio não apresenta grandes problemas e não é gargalo na execução do *pipeline*. As instruções são retiradas do *buffer* de busca, decodificadas e encaminhadas ao estágio de despacho.

3.5. Estágio de despacho

Tipicamente, é no estágio de despacho que as dependências de dados são detectadas e tratadas.

Dependência de dados são observadas quando duas ou mais instruções subseqüentes lêem ou modificam o valor de um mesmo registrador ou posição de memória. Existem, basicamente, três tipos de dependências de dados – as dependência verdadeiras, as anti-dependências e as dependências de saída.

As dependências verdadeiras são observadas quando uma determinada instrução necessita de um resultado produzido por uma instrução executada anteriormente, ou seja, na ocorrência de uma leitura após uma escrita (*Read After Write* ou *RAW*). A Figura 3.3 (a) apresenta um exemplo desse tipo de dependência. Nota-se que a terceira instrução é dependente da primeira e, caso as duas fossem executadas em paralelo, a instrução dependente usaria o valor antigo de *r1*, gerando um resultado incorreto. Logo, a terceira instrução deve esperar o término da primeira instrução não podendo ser despachada antes que isso aconteça.

Embora sendo dependências verdadeiras o tipo mais óbvio de dependência de dados, existem ainda outros dois tipos, as anti-dependências e as dependências de saída. Esses dois tipos de dependência não causam o bloqueio do despacho porém podem causar

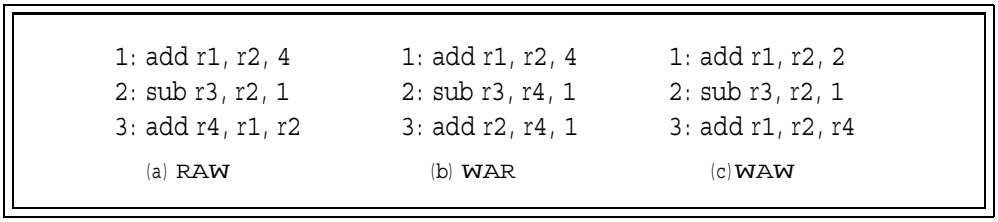

Figura 3.3: Exemplo de (a) dependência verdadeira (b) anti-dependência e (c) dependência de saída.

resultados incorretos quando mais de uma instrução é despachada por ciclo ou executada fora de ordem. Nas arquiteturas superescalares acontecem as duas coisas.

Um exemplo de anti-dependência é apresentado na Figura 3.3 (b). Nesse caso, a terceira instrução modifica um valor que é lido pela primeira. Dessa forma, essas duas instruções não poderiam ser executadas fora de ordem ou em paralelo, pois a primeira instrução poderia ler o novo valor de *r2*, produzindo como conseqüência um resultado incorreto. Esse tipo de dependência caracteriza-se pela existência de uma escrita após uma leitura (*Write After Read* ou *WAR*).

O terceiro tipo de dependência é chamada dependência de saída. Esse tipo de dependência é observada na ocorrência de duas escritas subseqüentes a um mesmo registrador. É possível observar um exemplo de uma dependência de saída na Figura 3.3 (c). Nota-se que a primeira e a terceira instruções escrevem no mesmo registrador *r1*. Novamente, se essa dependência não for tratada, as instruções devem ser executas em ordem, garantindo o valor correto de *r1* (o resultado produzido pela terceira instrução deve ser mantido). As dependência de saída caracterizam-se por apresentarem uma escrita após outra escrita (*Write After Write* ou *WAW*).

É possível observar que a ocorrência de qualquer tipo de dependência de dados pode afetar o desempenho da arquitetura. Tipicamente, o estágio de despacho é responsável pela detecção das dependências verdadeiras e da detecção/resolução das anti-dependências e dependências de saída. Além disso, esse estágio encaminha as instruções para os *buffers* associados às unidades funcionais.

A detecção de uma dependência de dados é dada pela observação da lista de operandos de cada instrução, fornecidas pelo estágio de decodificação. Tão logo as anti-dependências ou dependências de saída sejam verificadas, o estágio de despacho encarrega-se de acionar o mecanismo responsável pelas suas respectivas resoluções.

Algumas arquiteturas, no entanto, optam por não tratarem as dependências de dados no estágio de despacho. Normalmente, esses processadores utilizam um mecanismo como o *scoreboard* [THO 64] que podem gerenciar todo o tratamento de dependências já no estágio de delegação. Por outro lado, é bastante viável o processamento dessa tarefa durante o despacho e, mesmo utilizando o *scoreboard*, vários processadores resolvem as dependências antes da delegação [KES 99]. A renomeação de registradores é reconhecidamente a técnica mais utilizada para esse fim.

A renomeação de registradores emprega o conceito de registradores físicos e lógicos. O conjunto de registradores físicos é formado pelos registradores disponíveis em hardware, invisíveis aos olhos do usuário enquanto que, os registradores lógicos, por sua vez, constituem um conjunto tipicamente menor de registradores utilizado pelo programador [SMI 95].

A renomeação consiste em mapear os registradores lógicos em registradores fisi-

cos. Na Figura 3.4 é apresentado um esquema onde observa-se o método de renomeação de registradores.

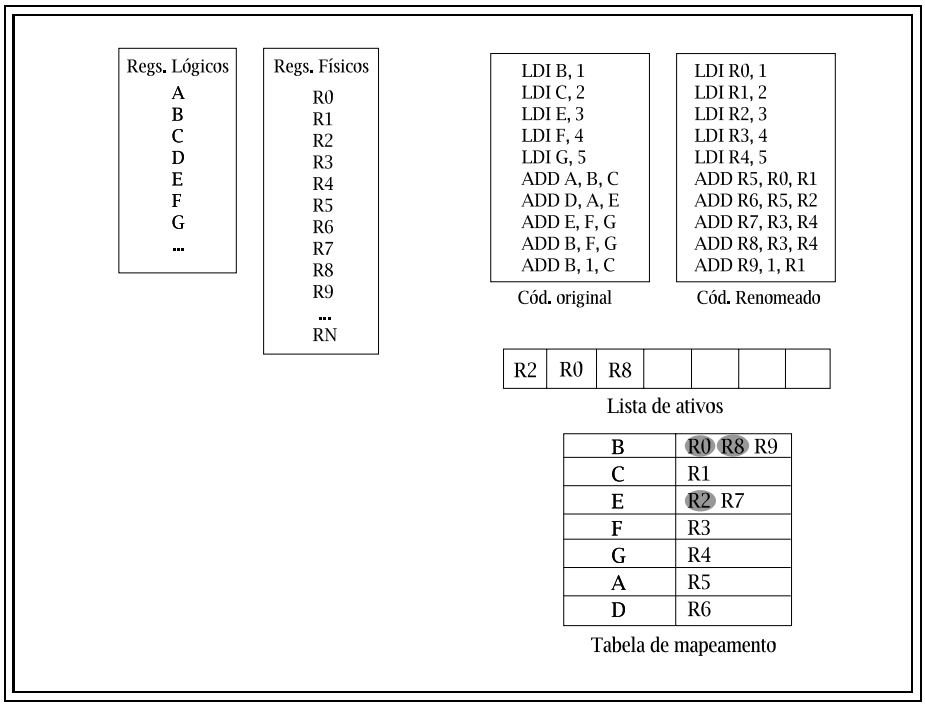

Figura 3.4: Esquema de renomeação de registradores.

O código original é concebido utilizando-se o conjunto lógico de registradores, pois esses são conhecidos pelo programador e definidos pela arquitetura do processador. O mecanismo de renomeação verifica esse código e faz um mapeamento dos registradores lógicos para registradores físicos. Um caso de anti-dependência ou dependência de saída é caracterizado se existe uma nova escrita em um registrador que foi mapeado anteriormente. Sendo assim, uma nova renomeação do registrador que causa o conflito é realizada. O registrador para o qual estava mapeado anteriormente é inserido em uma lista de registradores ativos, pois está sendo utilizado por instruções que ainda não terminaram sua execução. Os registradores físicos são mantidos na lista de ativos até que a execução da instrução dependente seja completada, garantindo a coerência dos resultados. Por esse motivo, o estágio de despacho deve estabelecer algum tipo de comunicação com o estágio de escrita de resultados. Essa comunicação entre estágios será descrita nas próximas seções.

É possível observar que as dependências verdadeiras não são tratadas com a renomeação de registradores. Esse tipo de dependências, apesar de detectadas ainda nesse estágio, só são resolvidas em estágios mais avançados do *pipeline*. Por outro lado, a renomeação pode resolver todas as dependências de saída e anti-dependência dependendo somente da dispoinibilidade de registradores físicos para tanto.

Após a renomeação de registradores, o despacho propriamente dito é efetuado. As instruções são encaminhadas para o próximo estágio, para uma posterior execução.

3.6. Estágio de delegação

Nesse estágio, as instruções, dispostas previamente em um ou vários *buffers*, são delegadas às unidades funcionais para que sejam executadas. Assim, o estágio de delegação deve observar quando uma instrução está pronta para a execução e liberá-la para tal. Diz-se que uma instrução está pronta quando todos os seus operandos, bem como os recursos necessários para executá-la, estão disponíveis.

É possível observar que as instruções independentes podem ser delegadas fora da sua ordem original. As instruções que apresentam dependência verdadeira devem aguardar por seus operandos. Contudo, a delegação de instruções renomeadas e independentes faz com que o fluxo entregue às unidades funcionais em cada ciclo seja maior.

Obviamente, a execução fora de ordem obriga um aparato maior no estágio de escrita de resultados. As instruções executadas devem retornar a sua ordem original, de forma a garantir a consistência do código em execução e respeitar a semântica do programa em execução. Para isso, são adotados os *buffers* de reordenação ou *Re-Order Buffers* (ROB). A estrutura desses, é apresentada na próxima seção desse trabalho.

O estágio de delegação pode ser implementado utilizando como base de gerenciamento dois mecanismos diferentes. O primeiro utiliza um único *buffer* centralizado, sendo conhecido como mecanismo de *scoreboard* [THO 64]. A segunda opção de implementação usa diversos *buffers* de instruções, um para cada unidade funcional. Essa solução é adotada pelo conhecido algoritmo de Tomasulo [TOM 67].

No entanto, o mecanismo de *scoreboard* e o algoritmo de Tomasulo gerenciam tarefas que vão além do estágio de delegação. Esses mecanismos também são responsáveis pela sincronia dos estágios de execução e escrita de resultados. Isso faz-se necessário porque ainda existem problemas relacionados à reordenação e ao adiantamento de dados para as instruções vítimas de dependências verdadeiras.

Na delegação, o *scoreboard* verifica se existe uma unidade funcional disponível para determinada instrução, além de observar os operandos de cada instrução a ser delegada. Apesar de tratar eficientemente dependências verdadeiras e anti-dependências, o estágio pára caso dependências de saída existam. Esse é o motivo pelo qual utiliza-se renomeação de registradores, mesmo com a adoção desse mecanismo.

Um operando está disponível se nenhuma instrução delegada recentemente o teve como registrador destino, ou ainda, se o registrador que contém esse operando estiver sendo escrito no estágio corrente. Isso é verificado para que não haja nenhum conflito na escrita de resultados nos registradores. Assim, se dependências de saída existem, o estágio de delegação pára até que essas sejam solucionadas. As anti-dependências e dependências verdadeiras são tratadas através de cópias mantidas dos registradores em uso.

O *scoreboard* autoriza a execução da instrução quando é determinado que os seus operandos estão disponíveis. Nesse passo, ocorre o escalonamento propriamente dito, ou seja, as instruções são enviadas fora da sua ordem original, de acordo com a disponibilidade de operandos.

O algoritmo de Tomasulo, por sua vez, faz uso de um esquema baseado em *buffers* específicos, conhecidos como estações de reserva. As estações de reserva armazenam operandos de instruções que estão aguardando para serem executadas. Assim, através desses recursos, é possível armazenar operandos tão logo estes estejam disponíveis, eliminando a necessidade de acessar um registrador. Tipicamente, utiliza-se uma estação de reserva para cada unidade funcional. O algoritmo de Tomasulo é usado associado à

renomeação de registradores, pois não existe controle sobre as anti-dependências e dependências de saída. As dependências verdadeiras são, normalmente, tratadas através da técnica de adiantamento de dados como será visto na próxima seção.

Para definir se os operandos estão disponíveis, o algoritmo de Tomasulo segue um esquema descrito a seguir. Para cada instrução existe um campo para armazenar cada operação e operando, com o seu respectivo valor e um *bit* de validade. Similarmente, existe um campo que indica o registrador destino da operação. Os operandos estão prontos quando os seus respectivos *bits* de validade estão *setados*. Uma instrução é encaminhada para execução quando todos os seus operandos estão prontos e existe uma unidade funcional disponível.

3.7. Estágio de execução

O estágio de execução de um *pipeline* superescalar é composto, basicamente, de unidades funcionais. Essas unidades funcionais são especializadas de acordo com o tipo de operação executada. Assim, existem unidades funcionais para operações com inteiros, operações com ponto-flutuante, além de multiplicadores/divisores de inteiros e ponto flutuante. Algumas arquiteturas tratam as instruções de desvio com um quinto tipo especializado de unidade funcional.

As instruções já chegam na unidade funcional sem nenhuma questão pendente e após a execução elas são encaminhadas para o estágio de escrita de resultados.

Contudo, as instruções de acesso a memória não comportam-se como as demais e devem ser tratadas diferentemente a partir desse estágio. Nos processadores RISC superescalares usuais, algumas instruções enquadram-se nessa categoria. Essas instruções são conhecidas como carga (*load*) e armazenamento (*store*).

A razão para essas instruções receberem tratamento específico é simples. Os endereços de memória que serão acessados por instruções de carga e armazenamento não podem ser identificados no estágio de decodificação, como ocorre nos demais tipos de instrução. Para a determinação dos endereços, faz-se necessário um cálculo, geralmente uma adição de inteiros. Assim, as instruções de acesso à memória são delegadas às unidades funcionais responsáveis e apenas após essa fase o endereço de acesso é conhecido.

As unidades funcionais, entretanto, não produzem um endereço físico e assim mais uma etapa, destinada a conversão desse endereço, deve ser executada. Usualmente, uma *cache* especializada, chamada de *Translation Lookaside Buffer* ou TLB é utilizada de forma a gerar mais rapidamente o endereço requerido. A TLB armazena as páginas mais recentemente utilizadas, e o endereço físico necessário pode ser obtido mais facilmente. No caso de falha na TLB, o endereço gerado deve ser convertido através de um mecanismo de mapeamento. Esse processo, extremamente penoso, é realizado apenas em último caso, pois é necessário acessar os mais altos níveis da hierarquia de memória.

Uma vez que o endereço a ser acessado é conhecido, a instrução de carga e/ou armazenamento é encaminhada para um *buffer* especial, de onde o acesso à memória propriamente dito será efetuado.

Observa-se ainda que armazenamentos na memória não podem ser realizados especulativamente, pois podem gerar incoerência. Isso pode ocorrer quando uma instrução está sendo executada após uma previsão de desvios. Desse modo, se o desvio foi previsto incorretamente, o armazenamento de um dado também incorreto teria sido realizado. Para

evitar esse tipo de problema, as instruções de armazenamento devem ser mantidas em um *buffer* e a atualização da memória, somente quando a instrução é a mais antiga no *buffer* de reordenação, ou seja, não existem desvios pendentes que possam descartar essas instruções.

As instruções de carga, por sua vez, utilizam os valores que estão no *buffer* de armazenamento e só acessam a memória se o endereço requerido não é encontrado no *buffer*. Processadores mais agressivos, já propõem a execução fora de ordem das instruções de acesso à memória. Nesse caso, esquemas especiais para solucionar o problema supracitado devem ser implementados.

Com o fim da execução é possível determinar quais desvios foram previstos incorretamente e quais obtiveram sucesso na previsão. Quando o resultado de uma instrução de desvio é conhecido, algumas tarefas são executadas pelo *pipeline*. Primeiramente, o resultado correto é comparado com o previsto no estágio de busca. Caso sejam iguais, nenhum problema é observado, pois a previsão foi realizada corretamente. Contudo, se o resultado da comparação é negativo, uma previsão incorreta foi realizada. O mecanismo de previsão deve então ser atualizado, o *pipeline* esvaziado e a busca redirecionada. Porém, dentro do estágio de execução é impossível identificar as instruções que vieram após o desvio e que, conseqüentemente, devem ser descartadas. Assim, o *pipeline* só pode ser esvaziado e a busca redirecionada no próximo estágio, onde as instruções são reordenadas.

Também ao fim do estágio de execução o mecanismo de gerenciamento utilizado, seja ele *scoreboard* ou Tomasulo, atualiza a fila de instruções pendentes. No caso do algoritmo de Tomasulo as estações de reserva são atualizadas e operandos que possuem alguma instrução pendente nas estações de reserva são encaminhados (ou adiantados), reduzindo a penalidade no caso de dependências verdadeiras. No caso do *scoreboard*, o *status* de cada operando/instrução é reavaliado.

3.8. Estágio de escrita de resultados

É no estágio de escrita que as instruções executadas fora de ordem, são reordenadas. Essa é uma atividade realizada necessariamente por qualquer arquitetura que utilize execução fora de ordem, garantindo a semântica dos resultados.

Para reordenação de instruções é comum o uso de estruturas conhecidas como *Buffers* de Reordenação (*ReOrder Buffers*) ou ROB.

Os *buffers* de reordenação são implementados como simples filas do tipo FIFO (*First In First Out*) que armazenam a ordem original de instruções. Dessa forma, o estágio de despacho é o responsável pelo preenchimento dessa fila, fazendo-se necessário um caminho de dados ligando esses dois estágios. Várias instruções podem ser inseridas no ROB, pois várias instruções podem ser despachadas de uma só vez. Quando a execução de uma instrução é finalizada, o ROB é sinalizado. Contudo, a instrução só será liberada para a compleção quando todas as instruções anteriores a ela sejam liberadas, garantindo assim, a ordem inicial do código.

Quando um desvio chega ao topo do *buffer* de reordenação, uma comparação entre o endereço previsto e o produzido pela execução é realizada. Se os dois endereços são iguais, a previsão foi bem sucedida. Nesse caso, o *buffer* de reordenação segue liberando as instruções subseqüentes ao desvio para compleção. Se for observada alguma diferença na comparação entre os dois endereços, houve erro de previsão. Nesse caso, todas as

instruções que aparecem depois do desvio são descartadas, o *pipeline* também é esvaziado e a busca de instruções redirecionada com o valor correto fornecido pelo estágio de execução. Nota-se que a penalidade de um desvio previsto incorretamente é altíssimo. Todas as instruções buscadas, decodificadas, despachadas, delegadas e executadas após a busca de um desvio previsto incorretamente são inutilizadas.

As instruções de armazenamento também são colocadas no *buffer* de reordenação, mas o procedimento de liberação é um tanto diferente das outras instruções. Quando uma instrução de armazenamento chega ao topo do ROB, o *buffer* que carrega essas instruções é investigado, e a escrita na memória finalmente é liberada. As instruções de carga por sua vez, obedecem ao mesmo procedimento das instruções comuns.

Caso a arquitetura não utilize renomeação de registradores, o mecanismo de *scoreboard* realiza uma tarefa adicional nesse estágio. A existência de anti-dependências são identificadas e caso essas sejam detectadas, o *scoreboard* não libera a instrução para compleção até que o valor antigo do registrador seja lido pela instrução dependente. Caso não seja identificada nenhuma anti-dependência, a compleção é realizada sem atraso.

3.9. Estágio de compleção

O estágio de compleção é responsável por completar as instruções liberadas, já em ordem, pelo estágio anterior. Completar as instruções significa modificar registradores e memória de acordo com cada instrução pronta para compleção.

Nota-se que o número de instruções completadas é menor que o número de instruções executadas, em virtude de desvios previstos incorretamente. Assim, pode-se afirmar que o número de instruções completadas depende de bons resultados na previsão de desvios. Quanto melhor a previsão, mais instruções entregues para compleção. Idealmente, se fosse utilizado um previsor perfeito, o número de instruções completadas e executadas seria o mesmo.

3.10. Projeto e Implementação de Microprocessadores

Como apresentado nas seções anteriores, muitas técnicas são empregadas hoje em dia para exploração do paralelismo ao nível de instruções (ILP - *Instruction Level Parallelism*) nas arquiteturas superescalares. O emprego destas técnicas tem permitido aumentar o número de instruções executadas por ciclo, o que conseqüentemente aumenta o desempenho dos processadores. Por outro lado, o emprego de técnicas sofisticadas como previsão de desvios, múltiplos níveis de memória, execução fora de ordem, entre outras, aumenta também a complexidade do projeto e da implementação destes dispositivos.

O projeto de um microprocessador superescalar do estado da arte é uma tarefa que compreende não somente a definição da arquitetura mas também a busca pelo equilíbrio entre diversos aspectos altamente dependentes. Desempenho não é o único aspecto observado. Potência, temperatura, área, preço, desempenho e o que a concorrência está planejando são alguns dos requisitos considerados pelos *designers* durante o projeto e implementação. Estes requisitos possuem um grau de dependência elevado, de forma que melhorias em um podem causar problemas em outros, muitas vezes complicando o projeto e criando restrições difíceis de serem compensadas.

Em outras palavras, a arquitetura de computadores está centrada na arte de balancear uma coleção de aspectos inter-dependentes para atender às exigências de um mercado em evolução rápida.

A razão pela qual os processadores continuam a evoluir rapidamente desde a integração dos primeiros transistores advém do impacto duplo da tecnologia e das aplicações [SHR 98].

A tecnologia tem proporcionado um aumento constante na densidade dos circuitos que podem ser integrados em um *chip* e ao mesmo tempo tem reduzido o custo dos mesmos. Isto tem forçado os projetistas a continuamente repensar a forma com que podem melhor explorar estes fatores de maneira que mais funcionalidade e desempenho possam ser oferecidos à preços significativamente menores.

Similarmente, o crescimento rápido do emprego da computação em diversos setores, para os mais diversos fins, tem também gerado uma demanda maior por funcionalidade. Os engenheiros de software desenvolvem aplicações mais pesadas para atender às expectativas dos usuários e para tirar vantagem dos processadores mais rápidos criando uma demanda nova para processadores ainda mais eficientes.

Dentro desse contexto, os projetistas precisam implementar cuidadosamente cada arquitetura, garantindo os requisitos necessários à demanda. O projeto de um microprocessador consiste em transformar a representação em alto nível da arquitetura do conjunto de instruções, ou modelo da arquitetura do processador, em uma pastilha de silício de alto desempenho que representa fielmente o modelo da arquitetura e que terá uma vida longa sem falhas [SHR 98]. O projeto de circuitos com um grande número de transistores é tipicamente chamado de Projeto VLSI (*Very Large Scale Integration*).

3.11. Etapas do Ciclo de Projeto

Projetar um circuito VLSI significa implementar a especificação do comportamento ou função de um circuito ainda não existente obedecendo à restrições de velocidade, área, consumo de potência, custo, confiabilidade, etc.

Para definir o comportamento, um circuito (ou parte dele) é considerado como uma caixa preta (*black box*) onde somente as entradas e saídas são conhecidas, enquanto a estrutura interna não precisa necessariamente estar definida [GOL 96].

O ciclo de projeto então é na verdade uma decomposição do problema inicial em vários níveis hierárquicos de abstração podendo seguir uma abordagem *top-down*, *bottom-up* ou uma combinação de ambas num estilo conhecido como *yo-yo*.

A partir desta decomposição hierárquica, vários níveis de abstração podem ser visualizados como apresentado na Tabela 3.1.

Os níveis de abstração apresentados na Tabela 3.1 não necessitam ser criteriosamente seguidos conforme a especificação apresentada. Muitas vezes, algumas fases podem ser combinadas, sobrepostas ou até mesmo eliminadas dependendo do nível de detalhamento e dos requisitos do projeto. Por exemplo, as etapas de projeto físico e *layout* podem ser eliminadas quando o projeto compreende a definição lógica de um processador ou circuito que será implementado em FPGA (*Field Programable Gate Array*). Neste caso, somente a arquitetura, microarquitetura e projeto lógico são necessários. Abaixo são apresentadas algumas definições típicas para cada uma destas etapas.

- Arquitetura: Define o conjunto de instruções, recursos e características visíveis ao

Tabela 3.1: Níveis de abstração.

Nível de Abstração	Exemplo das entidades modeladas	Nível equivalente no domínio de projeto
Sistema	pipelines, decodificadores de instrução, TLBs, tamanho de caches, etc	arquitetura e microarquitetura
Register Transfer Level (RTL)	registradores, barramentos, multiplexadores, lógica combinacional	microarquitetura e projeto lógico
nível lógico ou de portas lógicas (gate level)	biblioteca de células AND, OR, etc	projeto lógico
nível de transistores	transistores em diferentes tecnologias	projeto físico
nível de *layout*	geometria e localização dos transistores	projeto físico e *chip*

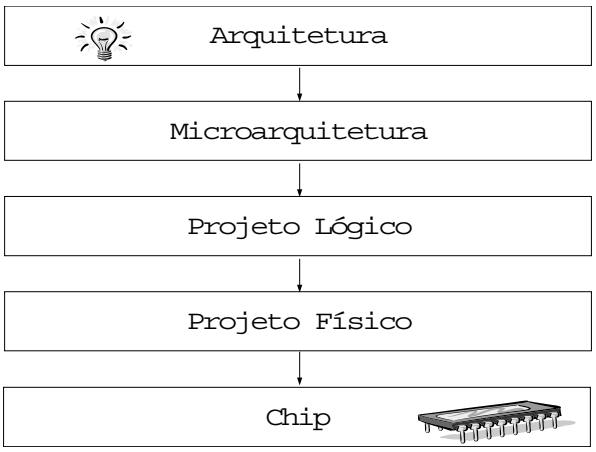

Figura 3.5: Hierarquia dos níveis de abstração de projeto.

software que está executando no processador. Pode também ser definida como a interface entre hardware e software provendo a especificação para a microarquitetura.

- Microarquitetura: Define o conjunto de recursos e métodos usados para implementar o conjunto de instruções da arquitetura (ISA - *Instruction Set Architecture*). Consiste na definição da maneira como os recursos estão organizados e das técnicas de projeto empregadas para alcançar os objetivos em termos de custo, desempenho, etc. Por fim, provê a especificação para o projeto lógico.

- Projeto Lógico: Especificação comportamental (ou estrutural) no nível de portas lógicas e registradores (Register Transfer Level - RTL). Serve de modelo para a simulação e verificação funcional. Provê também a especificação para o projeto físico podendo ser sintetizado automaticamente, dependendo do nível de detalhamento e dos requisitos de controle do projeto.

- Projeto Físico: Compreende a caracterização elétrica do circuito que implementa a lógica do processador (*Transistor level*), provendo também informações sobre o tempo de atraso das portas lógicas (*timing*), consumo de potência, área ocupada,

etc. Na etapa de projeto físico são gerados os *layouts* de máscaras que serão usados na fabricação do *chip*.

- *Chip*: implementação física do projeto lógico utilizando a tecnologia da fotolitografia para processar o material semicondutor (tipicamente o silicio) que será posteriormente empacotado formando o *chip*.

O projeto de circuitos VLSI pode também ser dividio em 3 domínios, cada um com suas características e complexidade. A Figura 3.6 apresenta o fluxo simplificado do projeto de circuitos VLSI segundo a visão do Professor Kenneth R. Laker da University of Pennsylvania. Observe que para cada uma das fases existe também uma fase de verificação necessária para garantir que os níveis resultantes não herdarão problemas existentes nos níveis acima, os quais fornecem a especificação para a sua implementação. Existe portanto uma interação entre cada uma das fases de modo que erros descobertos numa fase posterior, ou melhorias inseridas no projeto em fases avançadas, requerem modificações ou atualizações nas fases preliminares. O custo se torna mais alto quando modificações se fazem necessárias num estágio avançado do projeto.

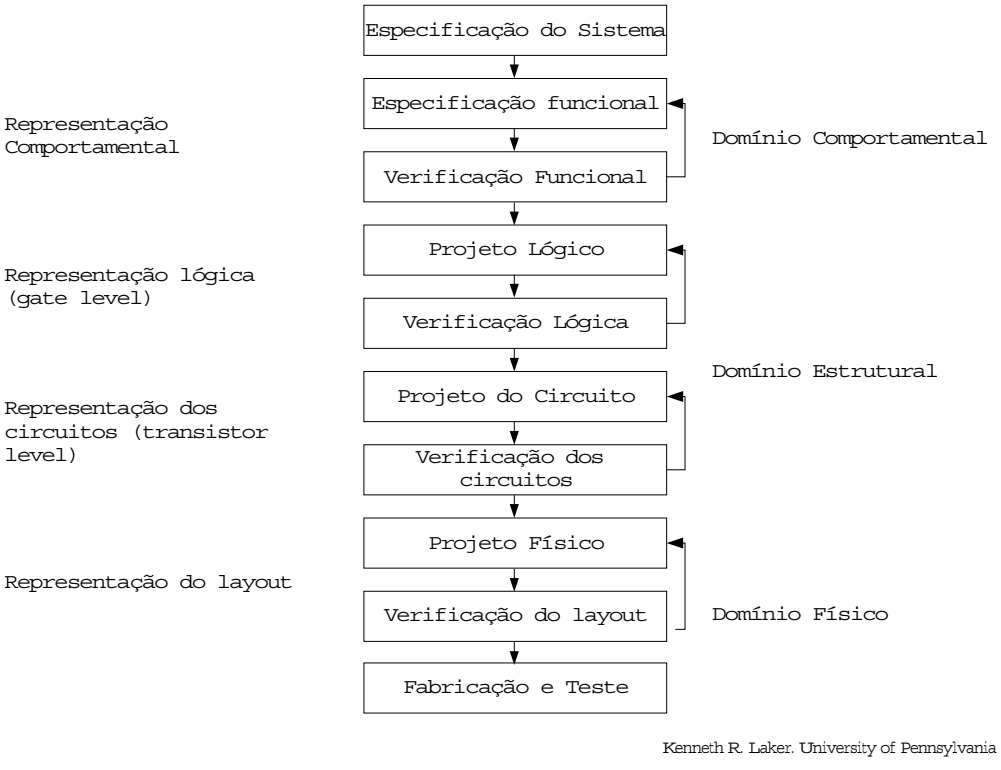

Figura 3.6: Fluxo Simplificado do Projeto VLSI.

A Figura 3.7 apresenta os 3 domínios de projeto (comportamental, estrutural e físico) no diagrama Y de Gajski. Neste diagrama, as etapas de projeto são apresentadas de acordo com o domínio onde estão incluídas. O domínio comportamental engloba principalmente a funcionalidade do projeto enquanto que o domínio estrutural determina os recursos e a forma com que estes operam. No domínio físico estão especificadas as etapas que correspondem a implementação dos domínios comportamental e estrutural.

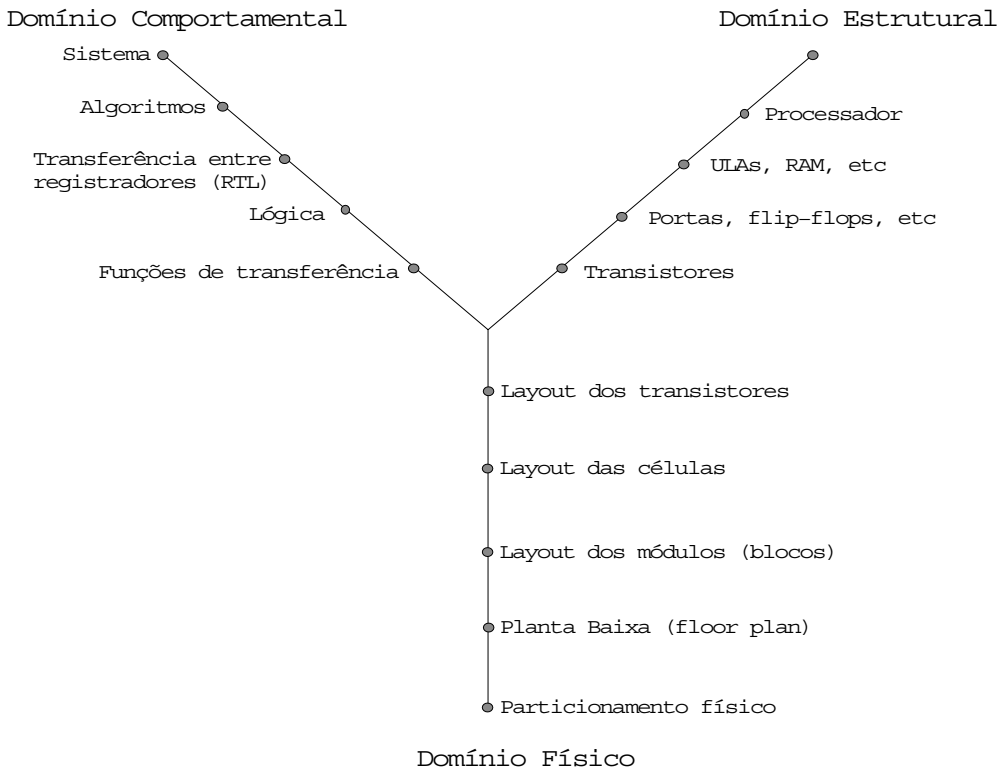

Figura 3.7: Diagrama Y de Gajski.

Os modelos empregados exercem uma influência extremamente importante neste processo de transformação. Esses modelos são usados para explorar e analisar as várias alternativas de projeto e para verificar as especificações comportamentais (funcionais), requisitos de compatibilidade e conformidade com os padrões exigidos. Um modelo - representação do microprocessador expressa em alguma linguagem - é tipicamente implementada em um simulador, emulador ou uma combinação de ambos.

Por fim, o projeto de circuitos complexos pode seguir também diretrizes similares às do projeto estruturado de software. Algumas estratégias comuns ao projeto de software e hardware são descritas se seguir:

- Hierarquia: o projeto é dividido em vários níveis (sub-módulos) similares em todos os domínios do projeto (comportamental, estrutural e físico);

- Modularidade: sub-módulos não devem ser ambíguos e devem possuir interfaces bem definidas promovendo o projeto e teste individual de cada sub-módulo;

- Regularidade: emprego de estruturas similares em todo o projeto. Por exemplo, no nível de circuito, deve-se empregar um tamanho uniforme de transistores ao invés de utilizar dispositivos manualmente otimizados para cada situação a não ser que isso se faça extremamente necessário;

- Localidade: caminhos críticos devem ser mantidos dentro dos sub-módulos não atravessando as fronteiras entre diferentes sub-módulos, sempre que possível.

3.12. Definição da Arquitetura Alvo

O termo arquitetura é referenciado freqüentemente como conjunto de instruções (ISA - *Instruction Set Architecture*) devido ao fato da arquitetura ser, na verdade, a especificação completa da interface (especificação externa) entre o hardware (processador) e o software (programas).

O conjunto de instruções portanto define o comportamento e a funcionalidade do processador como visto pelo usuário (programador). Nesta fase do projeto, um simulador (ISS - *Instruction Set Simulator*) do ISA é produzido como um meio de testar possíveis programas(*benchmarks*) a fim de verificar a compatibilidade com outros sistemas, ou versões anteriores do mesmo ISA, e proporcionar as primeiras noções de como a arquitetura será implementada nos níveis mais detalhados do projeto (microarquitetura).

Este simulador serve então como modelo de referência (*golden model*) para as demais etapas. Como o comportamento externo não está conectado necessariamente com uma estrutura interna em particular, este pode ser desenvolvido de uma maneira simplificada, não precisando apresentar um alto nível de detalhamento.

Existem, no entanto, uma série de fatores que podem influenciar a natureza do conjunto de instruções e as microarquiteturas candidatas para implementar o ISA resultante, como por exemplo [SHR 98]:

1. aplicações alvo e sistema operacional;
2. plataforma alvo (desktop, laptop, servidor, etc);
3. custo;
4. desempenho, e;
5. compatibilidade.

Segundo Michael Flynn em [JON 95], um conjunto de instruções bem definido deve permitir diferentes tecnologias de implementação e ser pouco sensível às evoluções destas tecnologias. Contudo, mesmo um ISA bem definido deve sofrer atualizações com o passar do tempo. Exemplos de melhorias podem compreender a inclusão de novas instruções para suportar novas funcionalidades e/ou a remoção de instruções que implementam aspectos antigos não mais empregados. Logo, a qualquer momento, uma arquitetura define um conjunto de instruções que consiste em um conjunto de instruções usadas freqüentemente, algumas melhorias para estender ou corrigir limitações do projeto inicial e algumas instruções que não são mais necessárias a não ser por razões de compatibilidade.

O ISA define ainda a especificação das operações o computador pode executar e que dados são necessários para cada uma delas [PAT 2003]. O termo *operando* é usado para descrever valores de dados individuais que servem como fonte e/ou destino da *operação* gerada a partir de uma instrução específica. Os operandos podem assumir tipos de dados que são igualmente definidos pelo ISA. Um tipo de dados é uma representação definitiva (inteiro, ponto flutuante, complemento de dois, etc) para um operando de tal forma que o computador possa executar operações sob aquela representação.

Além dos tipos de dados (operandos) e instruções o ISA define ainda os modos de endereçamento e a forma de acesso à memória onde são armazenados os dados e instruções dos programas que executam no computador.

O número de operações, tipos de dados, modos de endereçamento e formas de acesso à memória variam entre ISAs diferentes. Alguns ISAs implementam um conjunto reduzido de instruções para simplificar a implementação considerando apenas as instruções mais freqüentemente executas e emulando as demais (RISC - *Reduced Instruction Set Computer*) ou implementam um número elevado de instruções complexas que dão maior funcionalidade sob o ponto de vista programação mas tornam a microarquitetura mais complexa (CISC - *Complex Instruction Set Computer*).

Vários ISAs são usados hoje em dia. Os exemplos mais comuns são o IA-32 introduzido pela Intel Corporation em 1979 atualmente usado também pela AMD e outras empresas. Outros ISAs são o PowerPC (IBM e Motorola), Alpha (Compaq Computer Corporation), PA-RISC (Hewlett-Packard) and SPARC (Sun Microsystems, HAL Computer Systems e Fujitsu).

A tradução a partir de uma linguagem de alto nível como C, Pascal e outras se dá através de um compilador que traduz os comandos da linguagem em instruções *assembly* (linguagem de baixo nível) que são posteriormente traduzidas em linguagem de máquina por um montador (*Assembler*).

3.13. Definição da Microarquitetura e Projeto Lógico

A microarquitetura é a implementação da arquitetura (conjunto de instruções, modos de endereçamento, tipos de dados, etc) sob uma perspectiva de mais baixo nível. A microarquitetura é comumente definida em função do desempenho e custo desejados para a implementação de uma arquitetura.

Yale Patt e Sanjay Patel em [PAT 2003] apresentam uma analogia interessante que facilita a compreensão do fato de que uma arquitetura pode ser implementada através de diferentes microarquiteturas. Cada implementação é uma oportunidade para o projetista escolher aspectos que contemplam com maior ou menor ênfase o desempenho ou o custo do microprocessador.

O exemplo mostrado por eles é o de um automóvel. O ISA seria a interface, ou seja, o que o motorista vê quando senta ao volante e prepara-se para dirigir o automóvel. Todos os automóveis possuem a mesma interface (ISA) que é diferente da interface de um barco ou avião, por exemplo. A interface compreende o conjunto de pedais (acelerador, freio, embreagem), a direção, o cojunto de botões que comandam os faróis, etc, enquanto a microarquitetura que implementa esta interface (ISA) é o que faz o automóvel andar de verdade: o motor e a parte mecânica, que estão diretamente relacionados com o desempenho e custo do automóvel (sistema).

A microarquitetura se refere ao conjunto de recursos e métodos usados para implementar a especificação da arquitetura. Este nível de abstração tipicamente define a forma com que os recursos estão organizados bem como as técnicas de projeto empregadas para alcançar os objetivos em termos de desempenho e custo. A microarquitetura, por fim, provê a especificação para o projeto lógico [SHR 98].

A partir desta especificação pode-se conceber o modelo comportamental a partir do emprego de uma linguagem de descrição de hardware (HDL - *Hardware Description Language*) como VHDL ou Verilog, ou até mesmo SystemC que permite uma descrição em mais alto nível mas que pode igualmente ser sintetizada em niveis mais detalhados.

O próximo passo é implementar cada elemento da microarquitetura a partir de

circuitos lógicos. Neste momento também é necessário escolher como melhor transformar o nível de abstração anterior em termos de desempenho e custo.

Nesta fase o nível de detalhamento do projeto aumenta, dando uma maior ênfase aos circuitos lógicos que implementam a funcionalidade do processador. O projeto lógico visa então a especificação da lógica combinacional e seqüencial que implementa a micro-arquitetura aproximando-se mais do nível fisico.

A representação lógica (Figura 3.8) pode também ser chamada de *Gate level* pois descreve a função de um bloco ou sub-bloco em particular. Uma porta (*gate*) é, de um modo geral, um bloco básico a partir do qual implementa-se uma função *booleana*.

As portas (*gates*) são então conectadas para criar uma unidade funcional mais complexa a partir da qual são gerados os modelos de circuitos (ex.: nível de transistor - projeto físico).

A microarquitetura e o projeto lógico exercem uma grande influência sobre a freqûencia de operação alcançada pelo processador já que é nesta fase que são definidos os elementos básicos empregados na implementação. A Tabela 3.2 mostra a relação do aumento de freqüência entre duas implementações do ISA IA-32 (x86) através do Celeron e Pentium4. Observe, no entanto, que o aumento de 83% na freqüência, resultado de uma microarquitetura mais complexa implementada com um tecnologia mais avançada, permitiu um aumento de apenas 20% no desempenho para o *benchmark Business Winstone 2001*.

Tabela 3.2: Freqüência x Desempenho.

Processador	Celeron	Pentium4	Diferença (%)
Freqüência	1.2GHz	2.2GHz	83%
Desempenho	44.7	54.8	20%
# de transistores	44M	42M	-5%
Potência	29.9W	55.1W	84%
Custo	US $103	US $562	446%

Source: PC Magazine (February 2002)

Business Winstone 2001 measures system performance on several office-based applications, including Lotus Notes, Microsoft Office and Norton Antivirus.

Cost source: CNET (January 4, 2002). Yahoo Finance News (Jan 8, 2002)

3.13.1. Verificação Funcional

Assim como a arquitetura é a especificação externa do processador, a microarquitetura e o projeto lógico podem ser considerados como a especificação interna. Esta especificação deve ser detalhada o suficiente para guiar e definir claramente a implementação física podendo contemplar aspectos bem definidos como: parte operativa e de controle, pipeline, registradores e caches, barramentos principais e auxiliares, interrupções e aspectos de temporização.

O modelo comportamental ou funcional possui um comportamento externo idêntico ou bastante similar ao modelo de referência elaborado a partir da arquitetura. Para verificar a similaridade com o modelo de referência e garantir que ambos são funcionalmente equivalentes, os programas de teste (*benchmarks*, *test suites* e *testbenches*) são aplicados novamente no novo modelo, agora mais detalhado, e as saídas são então comparadas.

Os programas de teste podem ser concebidos para testar aspectos de desempenho, compatibilidade, ou podem ser escritos para testar aspectos específicos de cada bloco ou sub-bloco do processador.

Nesta fase faz-se necessário o uso de ferramentas de apoio ao projeto (CAD - *Computer Aided Design Tools*) para facilitar a descrição dos níveis mais detalhados e permitir também a comparação com o modelo de referência e a monitoração do grau de cobertura dos testes.

Linguagens para descrição de hardware, simuladores, ferramentas de verificação e análise de cobertura são as mais empregadas nesta fase. As ferramentas para análise de cobertura como HDLScore, por exemplo, são de extrema valia na determinação da cobertura dos testes realizados. Esta análise compreende a avaliação do acesso a cada um dos blocos do modelo (*BC - Bloco Coverage*), de cada um dos caminhos quando existem testes de condição (*PC - Path Coverage*) e de todas as expressões mapeadas (*EP - Expression Coverage*). A partir desta análise, a verificação funcional pode determinar se são necessários mais testes ou até mesmo avaliar a qualidade dos testes.

Os programas de teste podem ser escritos manualmente para verificar aspectos individuais de blocos específicos (*Directed Focused testing*) ou podem ser gerados automaticamente por ferramentas de geração randômica de testes (*Random Code Generators*). Outros testes podem fazer parte de *suites* existentes quando o objetivo é a verificação de compatibilidade e/ou desempenho.

No entanto, o aspecto mais importante da verificação funcional é garantir que o modelo comportamental é uma representação fiel e compatível com o modelo de referência já que este, na verdade, se tornará o modelo de referência para as fases posteriores do projeto.

A Figura 3.8 apresenta o fluxo de projeto agora de uma forma mais detalhada. Observe a interação entre as diversas etapas do projeto [SHR 98].

3.14. Projeto Físico

As portas lógicas que descrevem o nível comportamental ou estrutural são selecionadas a partir de uma biblioteca de células que é elaborada com base em uma especificação otimizada para um determinado processo de fabricação. Estas células consistem de NORs, NANDs, inversores, flip-flops, latches, multiplexadores e células especializadas. O primeiro objetivo do projeto físico é definir as características físicas e elétricas destas células.

A dimensão e localização dos pinos de cada célula é expressada em múltiplos do *metal pitch* que é definido pelo processo de fabricação. Outros fatores são também levados em consideração, como por exemplo, os pontos de fornecimento de potência e aterramento (*power/ground supplies*) de maneira a evitar a criação de densidades de corrente elevadas que podem causar falhas posteriores devido ao efeito da migração de elétrons. De forma similar, o tamanho dos canais *p* e *n* dos transistores são definidos.

Além disso, características inerentes aos flip-flops são otimizadas para reduzir os tempos de *hold* e *setup* permitindo maior velocidade de operação. Linhas de *scan* são inseridas nos flip-flops para facilitar o teste e depuração do circuito em fases posteriores, porém isto implica em lógica adicional que pode aumentar o custo e complexidade. Entretanto, devido a alta complexidade dos circuitos de hoje, estes aspectos podem reduzir o tempo de teste compensando o custo mais elevado através da redução do tempo de projeto

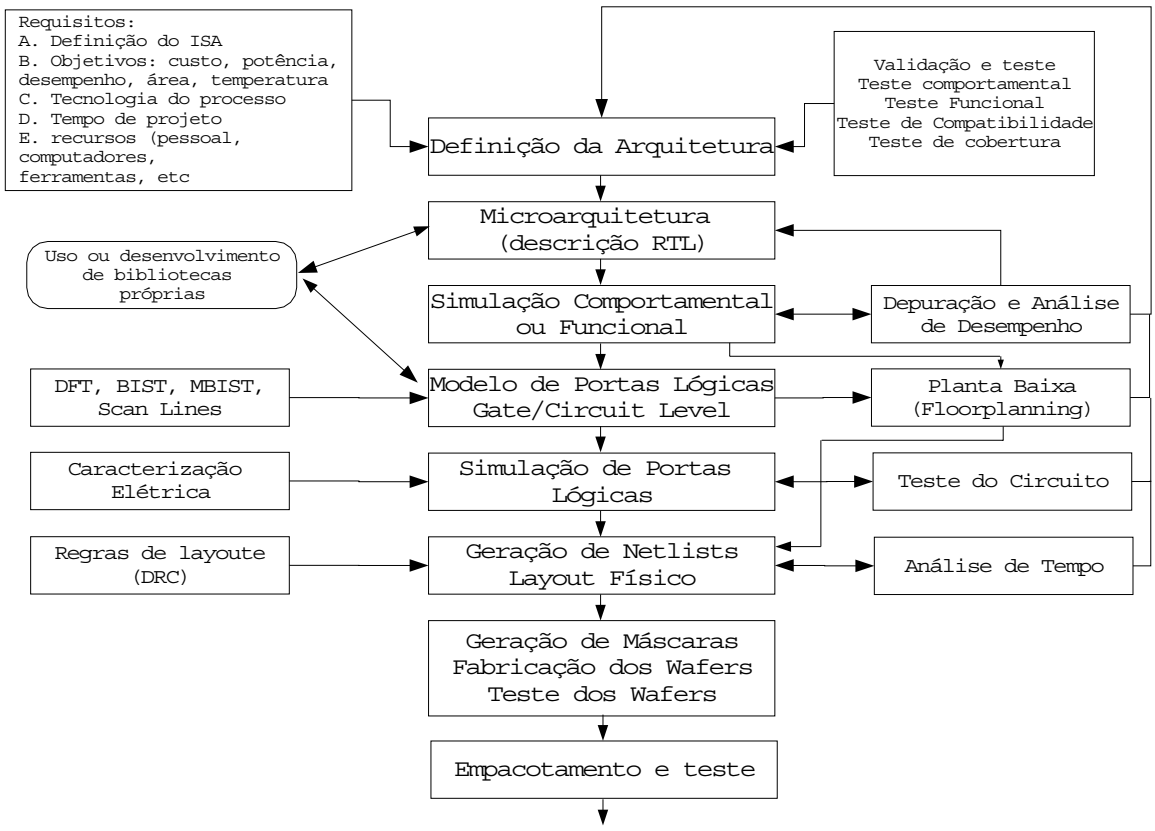

Figura 3.8: Fluxo de projeto mais detalhado.

e implementação levando o *chip* até o mercado mais rapidamente (*time-to-market*).

Depois de construir a biblioteca de células, torna-se importante caracterizá-la quanto a sua temporização. O primeiro requisito é determinar os atrasos máximos de cada célula. Isto é geralmente feito através da simulação de cada célula sob circunstâncias não ótimas (voltagem nominal e temperaturas elevadas). Os atrasos de propagação são então definidos para cada caminho (entrada até saída). Outras características que devem ser definidas neste momento são os requisitos de *min-path* e *hold time*.

A conexão das células lógicas se dá através da elaboração de esquemáticos que obedecem as regras definidas na bibliteca de células e que o projeto lógico como especificação. Este novo modelo, num nível de abstração mais baixo, é usado para análise do consumo de potência, temporização, área, planejamento e verificação da planta baixa (*floor planning*), etc.

Nesta fase do projeto podem ser inseridos também elementos redundantes, especialmente nas áreas que implementam memórias, já que estas áreas são mais sucetíveis a ocorrência de falhas de fabricação. Outra razão para inserir elementos redundantes como *spare flops*, *spare latches* e *spare gates* é facilitar a correção de erros lógicos. Sem elementos redudantes, um erro lógico que só é descoberto após a fabricação do *chip* remeteria o processo todo de volta à fase do projeto lógico e todas as etapas subseqüentes precisam ser refeitas.

Por outro lado, na existência de elementos redundantes, o mesmo problema en-

contrado após a fabricação, exige apenas modificações no projeto lógico para a obtenção de um novo modelo de referência, sem o erro encontrado. Nesse caso, a maior parte do projeto físico não precisa ser refeito, sendo necessário apenas o re-roteamento dos metais para interconectar as células redundantes ao circuito já existente (veja seção 3.16.).

Outro aspecto importante avaliado nesta fase está relacionado com a árvore de distribuição de *clocks*. O sinal de *clock* deve chegar à todos os elementos síncronos do circuito ao mesmo tempo. Considerando a complexidade dos circuitos atuais e que muitas vezes o sinal deve atravessar o *chip* para chegar aos elementos mais distantes, é preciso inserir *buffers* intermediários para gerar atrasos propositais de modo que o sinal de *clock* não chegue primeiro aos pontos mais próximos do ponto de geração do sinal. O número de *buffers* inseridos ao longo da distribuição do *clock* deve ser igual em todas as ramificações da árvore. Desta forma, o sinal passa pelo mesmo número de *buffers* para chegar à qualquer região do *chip* de forma que o atraso do caminho mais longo é também refletido nos caminhos mais curtos tornando a distribuição homogênea.

Como resultado desta etapa, tem-se a geração de esquemáticos e *netlists* a partir do modelo de portas lógicas que descrevem a topologia e conectividade dos circuitos. Em geral, as *netlists* são criadas em conformidade com um padrão (por exemplo EDIF) de forma que podem ser lidas e interpretadas por ferramentas de diferentes fornecedores e em diferentes plataformas.

3.14.1. Análise de Temporização

Cada bloco do processador é tido como uma entidade gerada a partir de uma *netlist*. A análise inicial de temporização é feita por um analisador estático onde as células são conectadas de acordo com a especificação fornecida através da *netlist*. Cada módulo é analisado individualmente e as células são dispostas manualmente ou com a ajuda de um programa de posicionamento (*placement*). Usando os dados de posicionamento uma ferramenta de análise de *minpath* é executada usando algumas estimativas de distorções do *clock* para garantir que os tempos de *hold* estão sendo obedecidos.

A análise de tempo do *chip* inteiro é feita posteriormente depois que cada bloco tenha sido analisado individualmente. Nas fases iniciais do projeto, a análise de tempo é feita no nível de bloco e sub-blocos para verificar a consistência dos requisitos de tempo para a sintese ou *layout* manual. Numa fase intermediária, *pré-layout*, a análise de tempo é executada sob os principais blocos do processador e na fase *pós-layout* a análise é realizada sobre o *chip* como um todo. Esta análise gradativa, iniciando no nível de sub-bloco, evita surpresas ao final do projeto.

3.14.2. Verificação Formal

Verificação formal é o processo de verificar matematicamente que o projeto está funcionando corretamente. Uma das formas de realizar verificação formal é chamada de *Equivalence Checking*.

Este processo consiste em contar todos os elementos de estado (*state elements: flops and latches*) do circuito e comparar o número resultante e a localização de cada um com o projeto lógico. Isso garante que as fronteiras de cada sub-bloco são iguais tanto no projeto lógico quanto no físico e que cada sub-bloco contém o mesmo número de elementos de estado. Além da contagem do número e localização de elementos de estado, este processo pode verificar também o nome de cada pino de entrada e saída do circuito

físico, sempre comparando com o projeto lógico. Adicionalmente a polaridade do sinal de saída de cada elemento lógico em cada uma das etapas é observado.

Esta fase não garante necessariamente uma equivalência funcional mas garante que o circuito físico representa fielmente o circuito lógico. A verificação funcional dá-se numa etapa anterior. Diferenças encontradas por este processo são resultantes, em geral, de otimizações aplicadas pelos projetistas do circuito físico e devem ser refletidas no projeto lógico, sempre que possível, para manter a equivalência.

A verificação de equivalência é necessária para garantir que a funcionalidade não é alterada por nenhum processo de otimização. Se alterações são detectadas, este processo pode apontar exatamente as partes do projeto que não estão de acordo com a especificação lógica. Apesar do projeto como um todo geralmente seguir uma abordagem *top-down*, falhas ou problemas detectados nos níveis mais baixos da hierarquia do projeto devem ser corrigidos e refletidos nas camadas superiores.

3.15. *Layout* de Máscaras

O processo de fabricação de circuitos integrados (*ICs - Integrated Circuits*) na tecnologia CMOS (*Complementary Metal Oxide Silicon*) deve contar com a elaboração de máscaras que servem de guias para o processo de litografia aplicado ao sílicio (matéria prima).

As máscaras podem ser geradas automaticamente a partir do RTL (projeto lógico) quando não se faz necessário otimizar partes específicas do circuito. No entanto, em alguns casos, os projetistas precisam ter total controle sobre o conjunto de máscaras geradas para otimizar tamanho e velocidade. Neste caso, o processo é chamado de *full-custom*.

Projeto de máscaras *full-custom* é o nome dado para a técnica onde a função e *layout* de praticamente cada transistor são otimizadas. O projeto envolve então a manipulação pontual da geometria e a simulação detalhada de cada estrutura do circuito físico.

O projetista do *layout* do circuito deve obedecer regras específicas (*design rules*) fornecidas pelo fabricante do IC para elaborar as máscaras que serão usadas no processo de litografia. O objetivo principal destas regras é obter um circuito com um *yield* ótimo (circuitos funcionais *versus* circuitos não-funcionais) na menor área possível, sem comprometer a confiabilidade do circuito [WES 92].

De uma maneira geral, as regras de projeto representam a melhor relação entre desempenho e *yield*. Quanto mais conservativas as regras são, maior é a probabilidade de que o circutio irá funcionar corretamente. Entretanto, quanto mais agressivas, maior será a probabilidade de que o circuito irá proporcionar bom desempenho. Observe que estas duas condições podem ser inversamente proporcionais. Para obter a melhor relação *yield* x desempenho faz-se necessário otimizar o *layout* manualmente, em alguns casos.

As regras de projeto especificam os limites de geometria que devem ser obedecidos pelo projetista das máscaras para que os padrões usados na fabricação conservem as características do projeto lógico/físico.

Dois conjuntos de limites determinados pelas regras se referem a largura das linhas e das interconexões entre diferentes níveis de metais. Se a largura das linhas é muito pequena é possível que as linhas se tornem descontínuas, o que significa que algumas conexões do circuito podem ficar abertas. Por outro lado, se os fios são colocados muito próximos uns dos outros, é possível que estes fios venham a tocar um ao outro causando

curto-circuitos.

As regras de projeto de *layout* definem basicamente duas coisas: (*i*) a reprodução geométrica das características que podem ser reproduzidas nas máscaras e no processo de litografia e (*ii*) a interação entre os diferentes níveis de metais.

Os processos atuais de integração de circuitos CMOS permitem vários níveis de metais para interconectar as células básicas (mais de 13 níveis). Em geral, os 3 primeiros níveis são usados para interconectar as células básicas formando as peças de lógica básica que compõem os sub-blocos do *chip*. Os sub-blocos são conectados entre si através de metais nos níveis superiores.

Como discutido na seção 3.14., durante o projeto físico são inseridos elementos redundantes que servem entre outras coisas para corrigir erros lógicos que só são descobertos após a fabricação do *chip*. Na verdade, as primeiras versões do *chip* só são produzidas até o terceiro nível de metal, permitindo o teste isolado de cada um dos sub-blocos. Caso sejam detectados problemas no nível de sub-blocos, é possível corrigir-se alguns destes problemas, se o número de elementos redundantes (*flops, latches* e *gates*) for suficiente para corrigir a lógica responsável pela falha. Se for possível, os elementos redundantes são re-conectados à lógica incorreta formando a solução para o problema encontrado. Assim, as máscaras dos níveis que interconectam os sub-blocos só precisam ser confeccionadas uma vez, caso não sejam encontrados outros problemas.

Observe que o custo da correção de problemas aumenta a medida que o processo avança. Logo, é extremamente importante encontrar erros lógicos nas fases iniciais do projeto antes que o projeto físico tenha sido terminado. Por isso, as etapas de verificação funcional e formal são essenciais para a produção de *chips* funcionalmente corretos.

No projeto de *chips* atuais com mais de 200 milhões de transistores, as fases de verificação (elaboração de testes, ferramentas de teste específicas e depuração) podem consumir mais do que 40% do tempo total do projeto e exigir a dedicação de muitos engenheiros, sendo atualmente uma das fases mais importantes do projeto.

As máscaras são elaboradas usando ferramentas gráficas que permitem verificar o corte de linhas, curto-circuitos, e verificar as regras de projeto (DRC - *Design Rule Checking*), etc. Após projetadas, as máscaras podem ser enviadas para o fabricante. Este procedimento é comumente chamado de *tape-out*.

A Figura 3.9 apresenta o layout de um inversor usando a tecnologia CMOS.

3.16. Processo de Fabricação

O processo de fabricação de circuitos integrados é tão ou mais complexo que o projeto lógico/físico. Ao receber as máscaras, o fabricante as processa para fabricar o circuito integrado em grandes *wafers* de silício.

Um circuito integrado é construído com diferentes camadas de materiais com formas e características elétricas distintas. Cada uma das camadas pode se transformar em uma forma diferente através da aplicação do processo de fotolitografia.

Inicialmente o *wafer* é coberto com uma camada de dióxido de silício e logo após com uma camada uniforme de material metálico. Para aplicar o processo de litografia é necessário ainda depositar uma fina camada de um material fotoresistente sobre as camadas já depositadas [UYE 2002].

No próximo passo, o *wafer* passa por um processo chamado de exposição onde

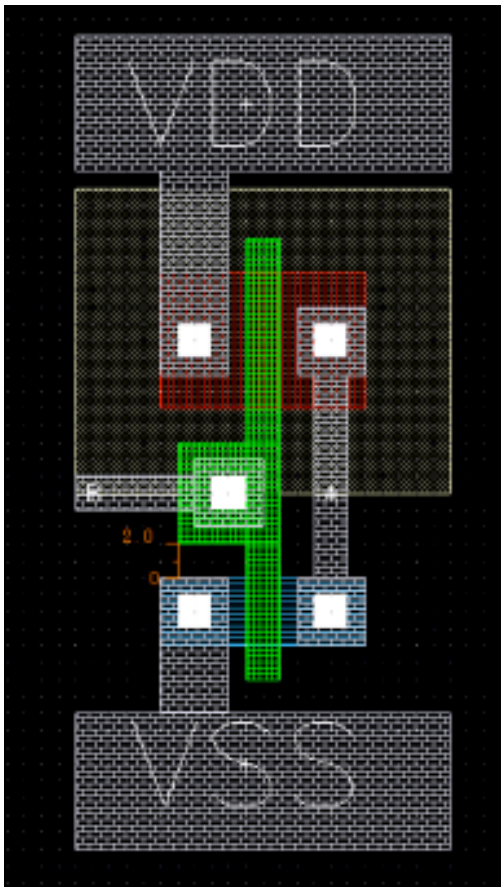

Figura 3.9: Layout CMOS de um Invesor.

uma forma é transferida para a superfícia das camadas de material aplicadas anteriormente.

O padrão transferido é determinado pelas máscaras elaboradas na última etapa do projeto físico. As máscaras servem de guias para a criação de uma placa chamada de retícula que consiste em uma placa de vidro produzido com uma material reflexívo (tipicamente o cromo). A retícula contém o desenho das formas que serão transferidas para o silício.

A transferência se dá quando uma luz ultra-violeta incide sobre a superfície do *wafer* tendo a retícula posicionada em cima do mesmo. A sombra da retícula é focalizada na superfície do material fotoresistente o que resulta em regiões sendo expostas à luz enquanto outras permanecem na sombra das formas desenhadas na retícula.

Após a exposição, as regiões da superfície fotoresistente que foram expostas à luz são removidas enquanto as que permaneceram sob a sombra são mantidas transferindo a forma definida na retícula para a superfície do *wafer*.

O processo tem continuidade quando o *wafer* é então submetido à um processo de corrosão no qual compostos químicos são colocados em contato com a superfície de metal removendo aquelas regiões que não estão cobertas pela resina fotoresistente sobrando então apenas as regiões que formam os padrões definidos pela máscara. O passo final é a remoção da camada fotoresistente que faz com que sobre apenas o metal sobre o silício inicial representando a forma desejada. A seqüência de impressão litográfica é executada

para cada camada de metal que deve cobrir o *wafer*.

Um *wafer* após passar pelo processo completo de litografia contém vários circuitos integrados. O *wafer* é então cortado resultando em vários circuitos integrados individuais. Cada circuito integrado é posteriormente empacotado em um módulo onde cada ponto de entrada e saída é conectado à um pino formando o *chip* que pode então ser acoplado à uma placa que deve conter um *socket* compatível com o número de pinos resultantes.

3.17. Conclusões e Novas Tendências

É indiscutível a importância das arquiteturas superescalares. Atualmente, a maioria dos microprocessadores de propósito geral comerciais utilizam esse tipo de arquitetura. Além disso, mecanismos cada vez mais complexos surgem com o intuito de suprir as deficiências e atingir o potencial máximo de desempenho existente. Execução especulativa fora de ordem, previsão de desvios, *cache* de traços e execução de *threads* simultâneas são apenas exemplos entre tantas opções disponíveis.

No entanto, a capacidade de expansão das arquiteturas superescalares está chegando a seu limite. Para aumentar efetivamente o desempenho é necessário mecanismos muito complexos e sofisticados que requerem alto custo e área de implementação. É possível que em poucos anos seja encontrado um novo paradigma que gerencie mais eficientemente os problemas encontrados em arquiteturas superescalares.

Uma das linhas mais discutidas diz respeito a migração de mecanismos, que hoje são implementados por hardware, para o software. Arquiteturas desse tipo são normalmente implementadas através de técnicas especiais, como VLIW (Very Long Instruction Word). Além disso, com um processador mais enxuto, é mais fácil e viável o desenvolvimento de mecanismos mais eficientes no controle de potência, muito importante para aplicações móveis, por exemplo. Um exemplo já real desse tipo de processador é o Crusoe da Transmeta. O grande problema dessa abordagem está relacionado com a sua principal vantagem: muitos arquitetos e projetistas continuam relutantes em concentrar tanto poder no software.

O avanço da tecnologia de concepção de circuitos tem permitido integrar cada vez mais transistores num mesmo *chip*. Por vários anos a tendência foi tornar os processadores mais sofisticados através da adoção de inúmeras técnicas para permitir explorar o paralelismo ao nível de instrução. Isso tornou estes processadores extremamente complexos e sem muito valor agregado. A adoção dessas técnicas permitiu aumento de desempenho mas, por outro lado, causou um aumento muito maior na complexidade do projeto.

O tempo de projeto e a dificuldade de testar processadores muito complexos se tornou um fator de risco. Os fabricantes de processadores não podem mais dedicar tanto tempo e assumir riscos tão grandes, já que o tempo de vida de um processador no mercado é muito curto devido a rápida evolução da tecnologia e da exigência dos compradores.

A tendência atual é simplificar o *pipeline* eliminando mecanismos muito complexos de forma a diminuir o tempo de projeto. A eliminação destes mecanismos torna o processador mais lento em termos de desempenho mas permite um ciclo de projeto mais rápido e mais garantido, já que por ser mais simples, o processador pode ser mais facilmente verificado e a propabilidade de erros diminui.

Ao invés de usar os transistores disponíveis para implementar um processador mais complexo, os projetistas estão optando por integrar em um mesmo *chip* vários pro-

cessadores mais simples. A replicação dos processadores permite um aumento do *throughput* total quando existem várias aplicações (*threads*). No entanto, o desempenho pode diminuir em uma aplicação analisada de forma isolada, já que o processador não explora o paralelismo inerente à aplicação em si, mas sim o paralelismo possível entre aplicações. Processadores como estes são hoje chamados de CMP (*Chip MultiProcessor*) e já estão disponíveis no mercado, como é o caso do IBM Power 4.

3.18. Bibliografia

[GOL 96] GOLZE, U. **Vlsi chip design with the hardware description language verilog**. 1st.ed. [S.l.]: Berlin, Springer-Verlag, 1996.

[GON 2000] GONÇALVES, R. A. L. **Simultaneous multithreaded architectures**: sempre: a SMT architecture with capacity of execution and scheduling of processes. 2000. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HEN 2003] HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture**: a quantitative approach. 3rd ed..ed. San Francisco: Morgan Kaufmann, 2003.

[HOR 99] HOREL, T.; LAUTHERBACH, G. UltraSparc-III: designing third generation 64-bit performance. **IEEE Micro**, Los Alamitos, v.19, n.13, p.73–85, May/June 1999.

[INT 2001] INTEL. **Inside the NetBurst micro-architecture of the Intel Pentium 4 processor**. Disponível em: <http://download.intel.com/pentium4/download/netburst.pdf>. Acesso em: janeiro 2001.

[JOH 91] JOHNSON, M. **Superscalar microprocessor design**. Englewood Cliffs: Prentince Hall, 1991.

[JON 95] JONES; BARTLETT. **Computer architecture, pipelined and parallel processor design**. [S.l.: s.n.], 1995.

[KES 99] KESSLER, R. E. The Alpha 21264 microprocessor. **IEEE Micro**, Los Alamitos, v.19, n.2, March/April 1999.

[NEM 90] NEMIROVISKY, M. **DISC**: a dynamic insturction stream computer. 1990. PhD Thesis — University of California, Santa Barbara, USA.

[PAT 2003] PATT, Y.; PATEL, S. **Introduction to computing systems**: from bits and gates to c and beyond. 3rd.ed. [S.l.]: McGraw-Hill, 2003.

[ROT 96] ROTENBERG, E.; BENNET, S.; SMITH, J. **Trace Cache**: a low latency approach to high bandwidth instruction fetching. Madison: University of Wisconsin-Madison/Computer Science Departament, 1996.

[ROT 97] ROTENBERG, E. et al. Trace processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.138–148.

[SAN 97] SANTOS, R. R. dos. **Um mecanismo de busca especulativa de múltiplos fluxos de instruções**. 1997. Dissertação (Mestrado em Ciência da Computação) — CPGCC/UFRGS, Porto Alegre.

[SAN 2000] SANTOS, T. G. S. dos. **Análise do comportamento de mecanismos de pré-busca em memórias hierárquicas de microprocessadores RISC superescalares**. 2000. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[SHR 98] SHRIVER, B.; SMITH, B. **The anatomy of a high-performance microprocessor**: a systems perspective. 1st.ed. New Jersey: IEEE Computer Society, 1998.

[SKA 99] SKADRON, K. **Characterizing and removing branch mispredictions**. 1999. PhD Dissertation — Princeton University, Princeton.

[SMI 95] SMITH, J. E.; SOHI, G. S. The microarchitecture of superscalar processors. **Proceeding of the IEEE**, New York, v.83, p.1609–1624, Dec. 1995.

[THO 64] THORNTON, J. E. Parallel operation in the Control Data 6600. **AFIPS Fall Joint Computer Conference**, v.26, n.2, 1964.

[TOM 67] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. **IBM Journal of Research and Development**, Armonk, v.11, n.1, Jan. 1967.

[UYE 2002] UYEMURA, J. P. **Sistemas digitais**: uma abordagem integrada. São Paulo: Thomson, 2002.

[WES 92] WESTE, N. H. E.; ESHRAGHIAN, K. **Principles of cmos vlsi design**: a systems perspective. 2nd.ed. [S.l.]: Addison Wesley, 1992.