*"**Fallacy:** Pipelining is easy"*
-Patterson, 2005

Welcome to Hell.

Resumo do Cap. 6 do Livro “Computer Organization and Design”, com adicionais de Superescalar dados em aula pela professora Edna Barros.

A abordagem pipeline pode ser vista como uma modificação da abordagem “mono-ciclo”, só que muito mais eficiente. Isso se deve ao fato de que ela explora os estágios de uma instrução genérica do MIPS:

1. Busca instrução da memória
2. Decodifica instrução e lê registradores
3. Executa instrução
4. Acessa operando na memória
5. Escreve o resultado em um registrador

Por isso o pipeline do MIPS tem 5 estágios. Um estágio para cada fase de instrução. Com isso, podemos ter até 5 instruções executando simultaneamente, desde que em estágios diferentes do pipeline.

O ganho de velocidade de uma abordagem pipeline, em situações ideais, pode ser visto como igual ao número de estágios que o pipeline possui. Assim, um pipeline de 5 estágios pode chegar a ser 5x mais rápido do que uma abordagem convencional.

Vale lembrar que o pipeline aumenta a performance do processador aumentando o *throughput* das instruções, ou seja, a taxa com que elas são processadas, ao invés de *diminuir o tempo de execução de uma instrução individual.* Mesmo assim, o pipeline funciona com muita eficiência porque programas na vida real executam bilhões de instruções.

Porque o conjunto de instruções MIPS é ideal para pipeline?

1. Todas as instruções do MIPS possuem o mesmo tamanho.
2. As instruções do mips contam com os registradores dos quais os dados são obtidos localizados sempre no mesmo lugar (o livro chama isso de “simetria” nas instruções). Isso quer dizer que o segundo estágio (decodificação) pode ler o conteúdo do registrador *ao mesmo tempo* que está determinando qual o tipo de instrução que será executada.
3. Operandos de memória aparecem somente em duas instruções: Loads e Stores. Se aproveitando desse padrão, podemos usar o estágio de execução para calcular o endereço, e acessá-lo diretamente no estágio seguinte. Se o mips permitisse que fizéssemos operações diretamente na memória, precisamos de estágios adicionais no pipeline.
4. Os dados no mips podem ser transferidos do processador para a memória em um único estágio do pipeline.

Conflitos de Pipeline

Existem três tipos diferentes de conflitos que podem ocorrer no pipeline:

Conflitos Estruturais

No pipeline do MIPS, não há a necessidade de se preocupar com esse tipo de conflito. Como mencionado anteriormente, as instruções do MIPS foram "feitas" para serem implementadas em Pipeline, tendo os conflitos estruturais evitados desde o princípio. Por exemplo: se a arquitetura do MIPS possuíssem apenas uma memória ao invés de duas, poderia existir um ciclo onde uma instrução estaria tentando acessar um dado da memória enquanto o processador tentaria buscar a próxima instrução dessa mesma memória. Isso se qualificaria como conflito estrutural.

Conflitos de Dados

Conflitos de dados acontecem quando o pipeline necessita "travar" sua execução, pois necessita que uma instrução termine antes que a próxima possa ser executada. Isso é chamado de *stalling.* Por exemplo, uma instrução ***add $1, $3, $5*** seguida de uma instrução ***sub $2, $1, $4*** causaria um conflito de dados. Algumas vezes, estes conflitos podem ser removidos pelo compilador via reorganização de código. Mas eles são tão comuns que a abordagem mais utilizada é a inserção de um hardware extra no pipeline, chamado de **forwarding unit** (unidade de adiantamento). Essa unidade avalia a dependência entre instruções subsequentes e "curto-circuita" a saída da ALU (que contém o valor atualizado do registrador, porém ainda não foi escrito) com a entrada da ALU que será usada pela instrução com dependência. Assim, ao invés de acessar o registrador desatualizado, ela terá acesso "em primeira mão" da informação mais recente.

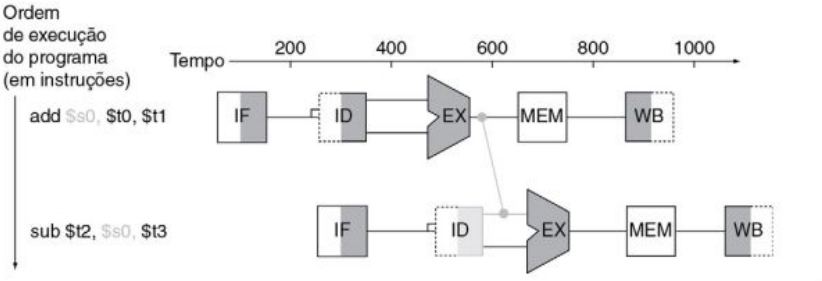


FIGURA 4.29 Representação gráfica do forwarding. A conexão mostra o caminho do forwarding desde a saída do estágio EX de `add` até a entrada do estágio EX para `sub`, substituindo o valor do registrador `$s0` lido no segundo estágio de `sub`.

Note que no exemplo gráfico acima, não seria possível conectar a saída de EX com a entrada de ID, por exemplo, pois isso significaria "voltar no tempo". Técnicas de adiantamento podem apenas ser feitas para estágios a frente.

Importante: Mesmo com forwarding, ainda é necessário *um stall (bolha)* para instruções tipo load. Isso porque o adiantamento de um dado obtido numa instrução tipo LOAD é obtido apenas após o estágio MEM, e como foi dito, não podemos adiantar coisas "para trás".

Conflitos de Controle

O terceiro tipo de conflito são os conflitos de controle. Ele vem da necessidade de fazer uma decisão baseada nos resultados de uma instrução, enquanto outras instruções também estão executando.

Em outras palavras: Como podemos fazer um beq ou um j num pipeline, sendo que ele só será efetivado no fim do estágio EX? (o que quer dizer que já temos 3 instruções "enfileiradas" atrás do BEQ. E se elas não devessem ser executadas?). A solução mais simples é: inserir bolhas no pipeline para cada instrução de decisão. Assim, a próxima instrução só irá iniciar após avaliação da condição.

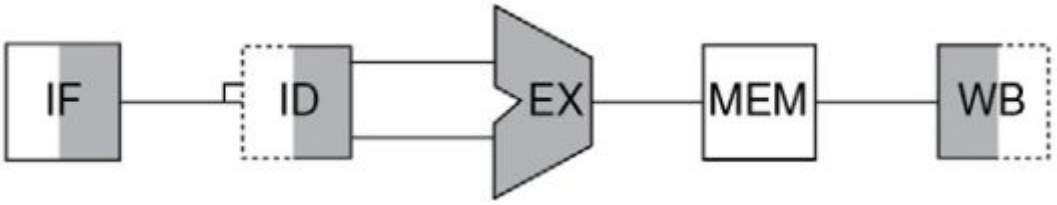
Uma solução melhor para resolver os conflitos de controle é a predição: Ou seja, fazemos um "chute" de que o desvio nunca será tomado. Se acertarmos, ótimo. Caso contrário, temos que corrigir nosso erro, pois contaminamos o programa com três instruções indevidas. Desse modo, o hardware cuida de "reverter" essas operações zerando os dados de controle e dos registradores, transformando as instruções em instruções do tipo NOP (também chamado de fazer o *flush* das instruções erradas).

Uma abordagem mais dinâmica de predição consiste em manter um histórico de cada decisão feita para cada desvio, e usar esse histórico para melhor prever o futuro.

Por fim, a abordagem mais utilizada é a combinação da **unidade de adiantamento** com a **unidade detectora de conflitos** (responsável por fazer o stall do pipeline/flush das instruções). Com isso, o cálculo do desvio pode ser adiantado de forma a já ser determinado no estágio ID. Com isso, caso seja feita uma predição errada de desvio, teremos que fazer o *flush* de apenas uma instrução, e não três. O hardware extra para adiantamento compensa, visto que a penalidade em caso de erro diminui drasticamente.

O caminho de dados do Pipeline

O caminho de dados do pipeline é separado em cinco estágios de execução de uma instrução:



1. **IF:** Instruction Fetch - Busca da instrução
2. **ID:** Instruction Decode - Decodificação da instrução e leitura dos registradores
3. **EX:** Execução da instrução *ou* cálculo do endereço
4. **MEM:** Acesso de dados na memória (load/store)
5. **WB:** Write back - escrita do resultado da execução em um registrador

De modo geral, o pipeline **nunca anda para trás.** O fluxo é sempre da esquerda para a direita, exceto:

- Ao incrementar o valor de PC e atualizar o registrador
- O estágio de Write Back, onde o resultado é escrito num registrador que se encontra no meio do caminho de dados.

Dados que fluem da esquerda para a direita nunca afetam a instrução atual. Note que justamente as exceções a essa regra mencionados acima (PC e WB) são as causas de conflitos de controle e de dados, respectivamente.

Essencialmente, como o pipeline funciona? Bom, primeiramente, como as instruções vão sendo “carregadas” da esquerda para a direita no caminho de dados, a instrução que entrou primeiro precisa levar consigo todos os dados necessários para que continue executando de forma independente. Se ela não fizesse isso, seria sobrescrita pela próxima instrução que passasse pelo fetch, perderia os valores dos seus registradores, etc, e não chegaríamos a lugar algum. Para isso, inserimos um registrador na “divisa” entre cada estágio, para segurar e encaminhar a frente as informações referentes a instrução que está passando por ele. Usualmente, são registradores de grande porte, de mais de 100 bits, para que carreguem consigo informações úteis para execução da instrução, como valores do registrador, cálculo de endereço, opcode, funct, shamt, etc.

Lembre-se: cada componente do pipeline só pode ser usado dentro de um único estágio. Caso contrário, ocorreria um conflito estrutural.

Colocando uma unidade de controle no pipeline

Podemos colocar também uma unidade de controle (como aquela feita no projeto Multi-ciclo) no pipeline, responsável por processar o opCode, gerenciar leitura e escrita nos registradores/memória, e controlar o fluxo de dados nos diferentes multiplexadores do caminho de dados. Porém, como o comando feito pela unidade de controle está associado apenas uma instrução (ou seja, a apenas um estágio), todos os dados desse estado

atribuídos pela unidade de controle necessitam de serem passados adiante, do mesmo modo como foi explicado com os registradores de pipeline.

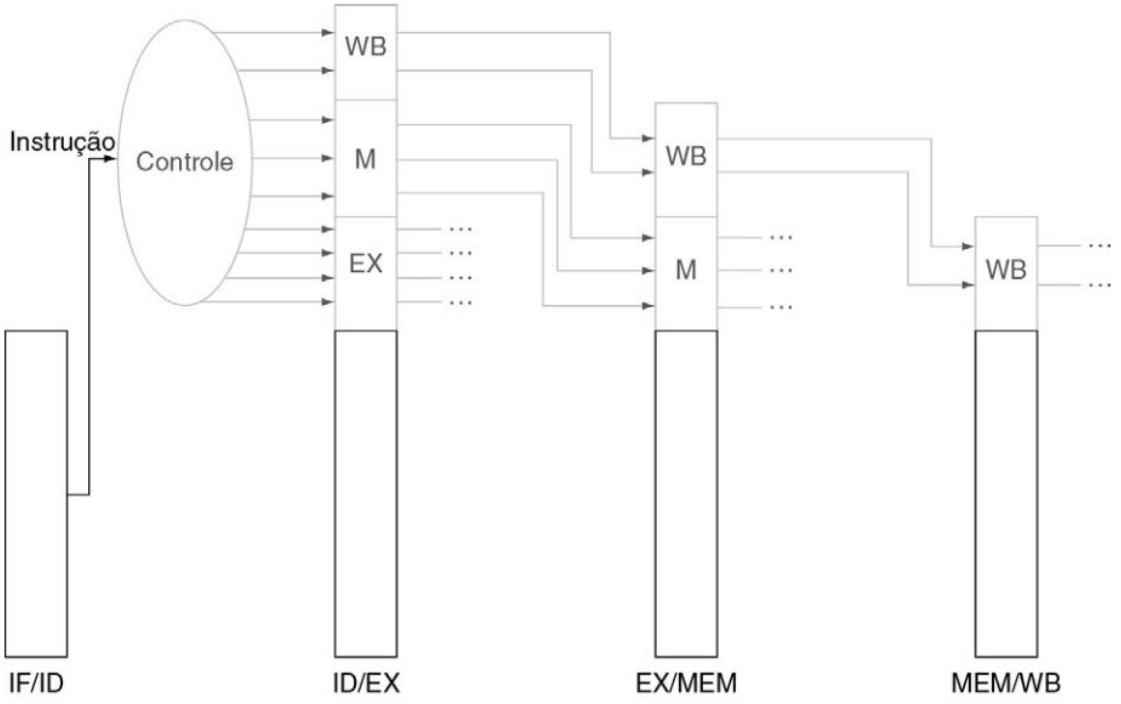
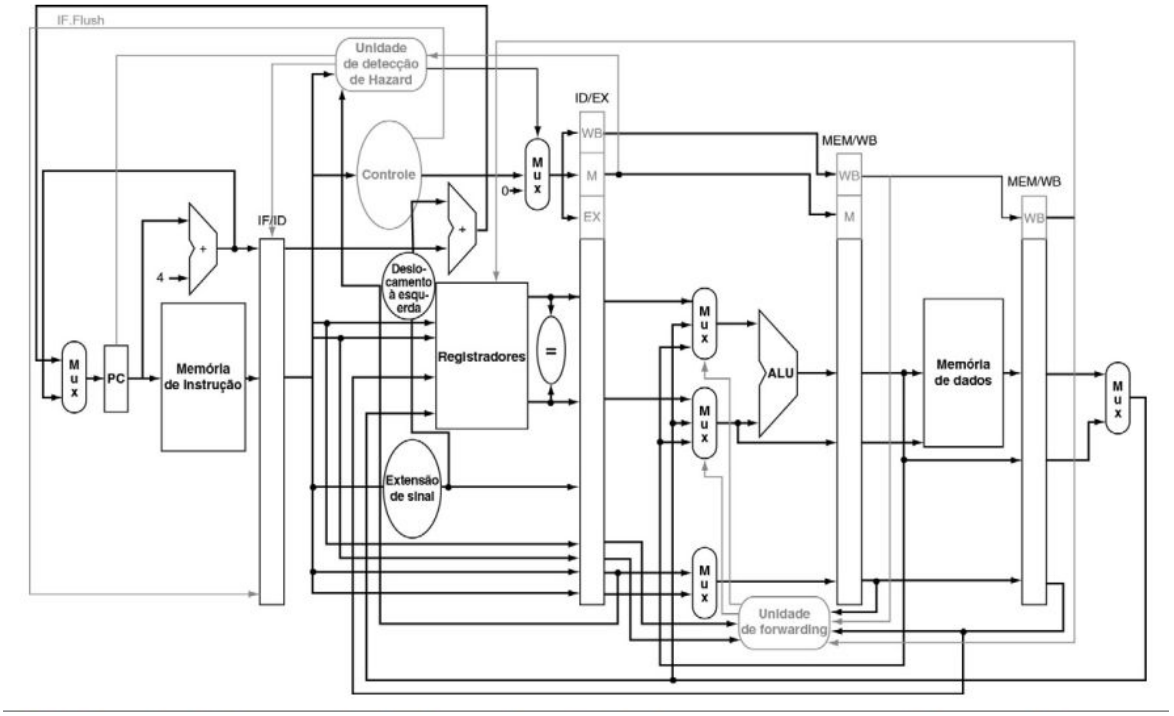


FIGURA 4.50 As linhas de controle para os três estágios finais. Observe que quatro das nove linhas de controle são usadas na fase EX, com as cinco linhas de controle restantes passadas adiante para o registrador de pipeline EX/MEM, para manter as linhas de controle; três são usadas durante o estágio MEM, e as duas últimas são passadas a MEM/WB, para uso no estágio WB.

Circuito Pipeline Completo:



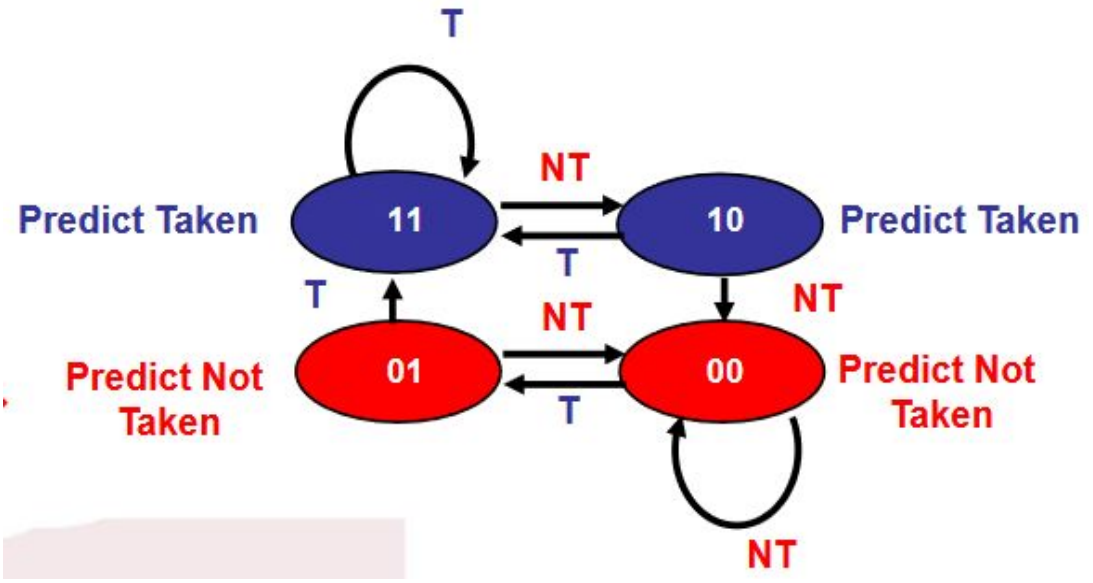
Previsão Dinâmica de Desvios

Assumir que desvios nunca serão tomados é a forma mais simples de técnica de previsão. Como já dito antes, assumimos que desvios não são tomados e fazemos um *flush* no pipeline caso estejamos errados.

Podemos então fazer uso de uma BHT (branch history table), que é uma memória pequena de buffer que armazena os bits menos significativos da instrução de desvio. Essa tabela servirá para mantermos registro dos desvios tomados e não tomados e com base nela, fazer previsões acerca das próximas decisões a serem feitas pelo pipeline.

Combinando a BHT com um esquema de previsão de 2 bits, temos um resultado excepcional. Em um programa com um laço que se repete 500 vezes, a taxa de acerto é de 99%. Isso porque nesse sistema, a previsão é feita dinamicamente com base nos erros. A cada 2 erros, muda-se a próxima previsão. Ou seja, se começamos prevendo que desvios não serão tomados, mas nos dois próximos desvios isso se mostrar uma previsão errada (significando que tomamos o desvio), o sistema agora passa a prever que iremos tomar o próximo desvio.

Fazendo uso dessa previsão e mantendo os bits menos significativos do endereço do desvio na BHT, temos um desempenho bastante razoável na primeira execução do programa, e excelente nas execuções seguintes, agora que temos o histórico dos desvios tomados ou não.



Exceções no Pipeline

Quando uma exceção é encontrada no processador MIPS, sabemos que ele deve parar a execução, iniciar o tratador de exceção e depois retomar o programa. Com pipeline, temos o mesmo comportamento, porém devemos dar *flush* nas instruções que entraram *depois* da exceção, para que não sejam executadas antes que a exceção seja tratada. Para fazer essa limpeza no pipeline, o mesmo mecanismo usado para o *flush* de desvios condicionais pode ser utilizado.

Um input adicional ao registrador PC é adicionado com o endereço da rotina de tratamento da exceção, e os sinais do controle são setados para provocar um stall no pipeline até que a exceção seja tratada devidamente. Como algumas exceções requerem que a instrução causadora execute até o fim, o modo mais comum de proceder é interromper a instrução, e executá-la novamente assim que a instrução seja tratada. Assim, o endereço + 4 da instrução que causou a exceção é salvo em EPC, e ao retomar o controle após o tratamento, o controlador subtrai 4 deste endereço.

Um desafio do tratamento de exceções é saber, num pipeline com 5 instruções simultâneas, qual delas causou a instrução, e caso seja mais de uma, qual deve ser tratada primeiro. Para isso, o hardware organiza as instruções para que a que chegou mais cedo seja tratada primeiro, mas todas as que causaram são salvas no registrador CAUSE, para que o tratamento de exceções posteriores ainda possam ser realizadas, assim que a exceção preferencial tiver sido tratada.

Temos dois tipos principais de exceções: ***precisas*** e ***imprecisas***.

- **Precisas:** a exceção está associada com uma instrução específica, e o valor salvo em EPC é o endereço desta instrução
- **Imprecisas:** a exceção não está associada exatamente com uma instrução, e o valor salvo em EPC é o valor atual de PC.

--It's all downhill from here--

Melhorando a Performance do Pipeline - Superpipeline e Superescalar

Essencialmente, superpipeline e superescalar são duas técnicas (onde o superpipeline é um pouco pior que o superescalar) que fazem uso de uma mesma abordagem: **ILP** (instruction level parallelism - Paralelismo ao Nível de Instrução). Ou seja, múltiplas instruções são executadas em paralelo para aumentar a performance.

Leve diferença:

- **Super Pipeline:** Pipeline com mais estágios: menos trabalho por estágio, maior período do clock, aumenta o paralelismo.
- **Multiple Issue:** Subdividido em Static Multiple Issue e Dynamic Multiple Issue, consiste em enviar mais de uma instrução por vez para execução.

Static Multiple Issue

Nessa abordagem, geralmente o pipeline é capaz de executar duas instruções por vez, agrupada no que chamamos de **VLIW** (Very Long Instruction Word), sendo possível executar uma instrução de desvio/aritmética e outra instrução de load/store. O Compilador faz o trabalho de agrupar essas instruções em pares, e despachar os pacotes de forma a evitar conflitos de dados e controle, reorganizando o código mas respeitando as dependências.

Vale lembrar que não podemos parear instruções dependentes entre si, e não podemos colocar uma instrução dependente de um load no par imediatamente subsequente (lembrando do stall necessário de 1 nop para instruções tipo load). Quando não houver um par disponível para aquela instrução, devemos inserir um nop para ser seu par.

Como muitas vezes o número de conflitos é muito grande em um código, técnicas de escalonamento são necessárias para realizar um paralelismo eficiente. Uma dessas técnicas é o **Loop Unrolling** que consiste em replicar o corpo de um loop para conseguir maior número de paralelismo, usando registradores diferentes na hora de replicar.

Por exemplo, num loop de 1000 iterações, podemos "desenrolar" as 4 primeiras, renomear seus registradores (os registradores dentro de um loop geralmente carregam algo chamado *antidependência:* similar a uma dependência de dados, mas na realidade as instruções são dependentes apenas pelo fato de usarem o mesmo registrador - não se trata de uma dependência *real* de dados -) e agrupar instruções parecidas. Assim, o ILP fica mais eficiente, e podemos parear mais instruções, aumentando nosso **IPC** (Instruction per Cycle). Obviamente agora o loop irá executar apenas 250 vezes. O "preço" de um aumento no IPC é a quantidade extensa de código e o uso de muito mais registradores do que era necessário inicialmente.

Importante lembrar também do hardware extra necessário para essa implementação: mais entradas para acesso aos registradores, um somador extra para cálculo de endereços, e portas de escrita para ALU e para LOAD. Sem esses novos componentes, esse pipeline estaria fadado a conflitos estruturais.

Dynamic Multiple Issue (Superscalar Processors)

Diferente da abordagem anterior, essa encarrega o processador para escolher quantas e quais instruções se deve executar, em tempo real. O compilador pode ajudar reorganizando as instruções antes da execução, mas não é necessário o escalonamento completo do programa.

Conflitos antes tratados pelo compilador agora serão tratados em tempo real pela CPU usando algumas técnicas.

Na abordagem mais simples de superescalar, as instruções são *despachadas em ordem* e o processador decide quantas podem ser executadas simultaneamente naquele ciclo de clock, respeitando as dependências. A vantagem aqui é que não dependemos de um compilador. Na abordagem anterior, ao mudarmos de modelo de processador, uma recompilação seria necessária para extrair a melhor performance.

A abordagem mais geral de superescalar conta com uma fila de instruções, que são despachadas em ordem, mas após o despacho, não vão diretamente para execução, mas sim para *reservation stations,* que mantém a instrução e seus operandos em "standby" até que seja o momento apropriado da execução (ou seja, quando não existem mais dependências). Note que o despacho é em ordem, mas a execução não. Por isso, precisamos de componentes adicionais: uma **commit unit** e um **reorder buffer** (que fica na commit unit). A commit unit cuida de guardar, dentro do reorder buffer, os valores que estão sendo processados atualmente, até que seja seguro escrevê-los nos registradores.

Quando uma instrução vai ser executada, ela pode buscar seus operandos no reorder buffer ou nos registradores. Assim que termina sua execução, seu valor é transmitido para todas as *reservation stations* que possam estar precisando desse valor para executarem, através de broadcast por um barramento compartilhado. A commit unit cuida de escrever nos registradores na ordem original do programa, evitando conflitos.

Importante! Podemos juntar o que já sabemos sobre exceções e especulação dinâmica de desvios no superescalar: Como temos a commit unit, que guarda os valores gerados pelas instruções até que seja seguro escrevê-los em definitivo nos registradores, caso necessitemos anular instruções executadas erroneamente por uma previsão errada ou uma exceção gerada, agora podemos simplesmente dar um *flush* no reorder buffer, limpando os valores errados obtidos (antes fazíamos um *flush* nas instruções).

***OBS*:** Note que lançar várias instruções por estágio permite que a taxa de execução de instruções, ou seja, o CPI, seja menor que 1. Por isso, é comum invertermos a métrica para IPC: Instruções por ciclo de clock. Assim, numa abordagem *multiple issue*, a otimização ideal deve tender ao número de instruções despachadas por vez. Por exemplo, se despachamos 2 instruções por vez, o IPC desejado deve ser bem próximo de 2.

Conflitos em Superescalar

Dependência verdadeira(RAW ou Read-after-Write): É o conflito de dados do MIPS pipeline que já conhecíamos, onde é necessária fazer uma leitura, porém o valor atualizado ainda não foi escrito. Isso era, e ainda é, resolvido por meio de forwarding.

Antidependência(WAR ou Write-after-Read): Um novo tipo de conflito que se assemelha com o inverso do RAW, isso é, temos uma instrução que irá ler um valor

atualizado do registrador que acabou de ser escrito, quando na verdade deveria ter lido o valor antigo, antes da escrita ocorrer. Mesmo se a instrução vier depois no código, as duas podem acabar sendo executadas juntas e acabar dando o mesmo problema.

Dependência de saída(WAW ou Write-after-Write): Outro novo tipo de conflito, acontece quando duas instruções que escrevem no mesmo registrador são executadas juntas. É necessário saber qual o valor final certo de registrador, ou qual instrução se encontra DEPOIS no código, para que a ordem de escrita seja preservada.

Como tratar esses novos conflitos?

1) Utilizando-se de NOPs.
2) Escolhendo instruções que não apresentam o conflito.
3) *Usar armazenamento temporário.*

Utilizaremos a número 3, e sua implementação será feita pelo **Algoritmo de Tomasulo.**

--The end is near--

Algoritmo de Tomasulo

O algoritmo de tomasulo funciona da seguinte maneira:

1) As instruções do programa são armazenadas em uma fila. Assim que possível, elas são encaminhadas para a *reservation station* correspondente, onde são cadastradas junto com os operandos que desejam utilizar
2) Execução: Aqui as instruções na *reservation station* são executadas (não necessariamente na mesma ordem que foram cadastradas) após as dependências de dados serem verificadas (através da verificação dos outros operandos em uso em todas as *reservation stations* através da tabela de reservas*).* Se houver alguma dependência, a instrução é "segurada" até que o dado necessário esteja disponível. Quando isso acontece, a tabela de reservas é atualizada e a instrução pode utilizar o operando. **("segurar" a instrução resolve o conflito RAW).**
3) Write result: Os resultados calculados nas *reservation stations* são enviados para todas as outras e para o *reorder buffer* através de barramento compartilhado, para que sejam utilizados pelas instruções que estão esperando. A *station* é marcada como disponível para receber outra instrução e o registrador correspondente ao resultado é escrito.
4) Commit: atualiza os registradores com o valor do reorder buffer, assim que for seguro (ou seja, houver a certeza de que aquela instrução deveria ter sido executada). Caso seja uma especulação falha, o reorder buffer é limpo (flushed).

O **WAW** será evitado utilizando o REORDER BUFFER, descrito na questão seguinte, independente da ordem de execução, o buffer escreverá os resultados na ordem certa.

O **WAR** será resolvido lendo dados antes que sejam sobrescritos por instruções seguintes. O algoritmo de Tomasulo lida com isso, porque quando uma instrução é despachada em ordem, a *reservation station* armazena de onde os operandos veem, assim, se uma instrução de leitura for enviada para execução antes de outra de escrita, terá seu operando antes dele ser atualizado por outra instrução de escrita.

Conclusão

Pipeline melhora o tempo médio de execução por instrução, mas não melhora o tempo inerente de execução de cada instrução. Porém como usualmente temos bilhões de instruções sendo executadas, o pipeline se mostra ûma alternativa muito mais eficiente.

Na abordagem multiple-issue, o foco é claramente o aumento do IPC - instruções executadas num mesmo ciclo, colocando-as em pares respeitando as dependências de dados, e despachando de uma só vez. Na abordagem estática, essa reorganização é feita pelo compilador. Na abordagem dinâmica, temos o hardware gerenciando em tempo real o que irá ser ou não executado no momento.

Fórmulas Úteis (Ou nem tanto)

Tempo de Execução Monociclo:

T = NumInstruções x Período

Tempo de Execução Multiciclo:

T = NumInstr1 x CPI1 + NumInstr2 x CPI2 + … + NumInstr[i] x CPI2[i]

Tempo de Execução Pipeline:

T = NumInstruções + 4

CPI:

CPI = numCiclos/numInstruções

CPI Pipeline:

CPI = TempoPipe/numInstruções

IPC:

IPC = 1 / CPI

Quantidade de Stall (NOP/bolhas) necessária para cada instrução MIPS do projeto 2017.2
(assumindo conflito de dados/controle)

Instrução	Stall (Sem fwd unit)	Stall (Com fwd unit)
TIPO R		
add	2	0
addu	2	0
and	2	0
jr	3	1
sll	2	0
sllv	2	0
slt	2	0
sra	2	0
srav	2	0
srl	2	0
sub	2	0
subu	2	0
xor	2	0
break	N/A	N/A
nop	N/A	N/A
rte	N/A	N/A
TIPO I		
addi	2	0
addu	2	0
andi	2	0
beq	3	1
bne	3	1
lbu	3	1
lhu	3	1

lui	2	0
lw	3	1
sb	?	?
sh	?	?
slti	2	0
sw	2	0
sxori	2	0
TIPO J		
j	3	1
jal	3	1

--Good bye then. Be safe, friend. Don't you dare go hollow--