Simulating a Pipelined RISC Processor

Amit Pandey

2016

Organização da apresentação

Organização da apresentação

A busca constante por maior desempenho em processadores levou ao desenvolvimento da arquitetura pipeline, que permite a execução simultânea de múltiplas instruções. A arquitetura RISC, com sua simplicidade e instruções de tamanho fixo, se mostra ideal para implementação de pipelines. Nesta apresentação, exploraremos a simulação de um processador RISC pipelined, com foco na escolha estratégica da arquitetura RISC.

Arquitetura RISC

A arquitetura RISC (Reduced Instruction Set Computer) se caracteriza por um conjunto reduzido de instruções simples, de tamanho fixo. Utiliza um grande número de registradores de propósito geral, simplificando o acesso aos dados. Privilegia operações com registradores, com poucos modos de endereçamento. Essa simplicidade resulta em um hardware mais enxuto e eficiente.

Pipeline

Um pipeline RISC típico possui cinco estágios: Busca da Instrução (**IF**), Decodificação da Instrução (**ID**), Execução (**EX**), Acesso à Memória (**MEM**) e Escrita de Volta (**WB**). Em cada ciclo de clock, uma nova instrução entra no pipeline, enquanto as instruções anteriores avançam para o próximo estágio.

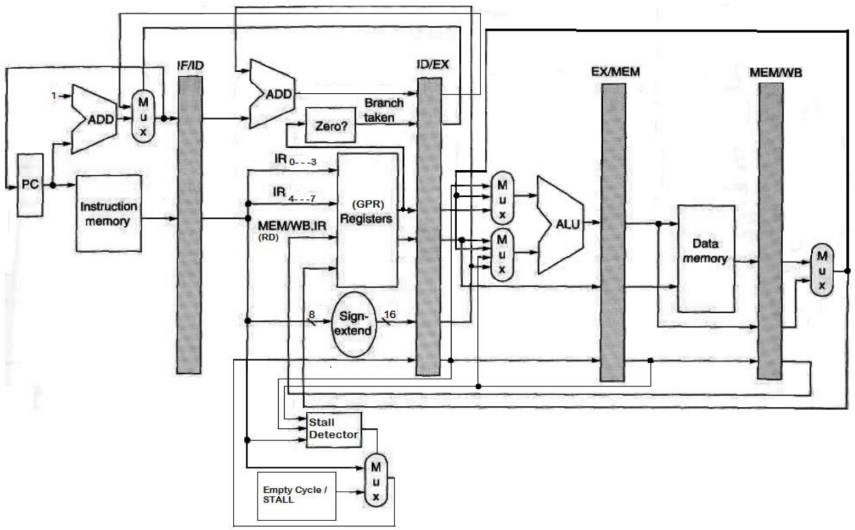


Fig. 1. Conceptual Data path design for a five stage pipelined RISC processor.

Pipeline em Processadores RISC

A arquitetura RISC é ideal para pipelines devido às suas instruções de tamanho fixo e à simplicidade do hardware. Isso facilita o projeto e o controle do pipeline, resultando em um processador com maior desempenho.

Os Hazards

A arquitetura apresentada parece sólida e perfeita. No entanto, existem alguns perigos que podem estar presentes no projeto. É sobre isso que Amit aborda em seu artigo, os chamados **hazards** que podem ocorrer durante a execução das instruções em um processador pipelined.

Os Hazards

A arquitetura apresentada parece sólida e perfeita. No entanto, existem alguns perigos que podem estar presentes no projeto. É sobre isso que Amit aborda em seu artigo, os chamados **hazards** que podem ocorrer durante a execução das instruções em um processador pipelined.

Existem três tipos principais:

- Hazards de Dados
- Hazards de Controle
- Hazards Estruturais

Os tipos hazards

Hazards de Dados

Ocorrem quando uma instrução precisa de um dado que ainda não foi calculado por uma instrução anterior no pipeline. Essa dependência pode gerar resultados incorretos se não for tratada adequadamente.

Os tipos hazards

Hazards de Dados

Ocorrem quando uma instrução precisa de um dado que ainda não foi calculado por uma instrução anterior no pipeline. Essa dependência pode gerar resultados incorretos se não for tratada adequadamente.

Hazards de Controle

Surgem de qualquer instrução que altere o fluxo sequencial de execução do programa. Isso pode levar à busca de instruções erradas.

Os tipos hazards

Hazards de Dados

Ocorrem quando uma instrução precisa de um dado que ainda não foi calculado por uma instrução anterior no pipeline. Essa dependência pode gerar resultados incorretos se não for tratada adequadamente.

Hazards de Controle

Surgem de qualquer instrução que altere o fluxo sequencial de execução do programa. Isso pode levar à busca de instruções erradas.

Hazards Estruturais

Acontecem quando duas ou mais instruções no pipeline tentam acessar o mesmo recurso de hardware simultaneamente, como a memória ou a unidade de execução.

Resolução dos hazards

A técnica mais simples para tratamento de hazards é a geração de bolhas no pipeline. Uma bolha consiste em gerar uma instrução vazia. Veremos com mais detalhes a seguir!

Resolução dos hazards

A técnica mais simples para tratamento de hazards é a geração de bolhas no pipeline. Uma bolha consiste em gerar uma instrução vazia. Veremos com mais detalhes a seguir!

Há outras técnicas, como:

- **Técnica de forwarding:** onde os resultados das operações anteriores são encaminhados diretamente para as instruções subsequentes sem esperar pela conclusão total da operação anterior.
- **Branch prediction:** previsão de desvios, relacionando indiretamente como o Princípio da Localidade.

Do conceito à prática

O artigo escolhido traz como objetivo principal desenvolver uma simulação funcional de um processador RISC com pipeline e que consiga lidar com esses hazards.

Vejamos como o autor realizou essa simulação utilizando o software "Logic Works", entendendo a implementação prática dos conceito teóricos discutidos anteriormente.

Organização da apresentação

Conjunto de Instruções no Processador RISC

A base para a execução eficiente em arquitetura *pipeline*.

O que é um **Conjunto de Instruções?**

É a coleção de todas as operações que um processador pode executar. Serve como interface entre o *hardware* e o *software* e define como os dados são processados no *pipeline*.

Por que é importante?

Ele organiza as operações e otimiza a execução no processador.

Categorias de Instruções

ALU Instructions	
OP.Code	Instruction
0000	No Operation
0001	Subtraction
0010	Addition
0011	Add Immediate
0100	Shift Right
0101	Shift Left
0110	Logical And
0111	Logical OR
1000	Logical NOR
1001	Logical NAND
1010	Logical XOR
Load - Store Instructions	
OP.Code	Instruction
1011	Load
1100	Store
Branch - Jump Instructions	
OP.Code	Instruction
1101	Branch Equal
1110	Branch Not Equal
1111	Jump

- **Instruções ALU (aritméticas e lógicas):** Operações como soma, subtração, *AND*, *OR*, etc. **Exemplo:** somar dois valores armazenados em registradores.
- **Instruções Load-Store:** Transferem dados entre registradores e memória. **Exemplo:** *LOAD* (carrega um valor da memória).
- **Instruções de Controle (Branch-Jump):** Controlam o fluxo do programa **(ex.: BEQ, JUMP)**. **Exemplo:** Saltar para uma instrução específica se uma condição for atendida.

Formatos de Instruções

R-Type: Usado para operações ALU.

Opcode	RD	RS1	RS2

Exemplo: **ADD RS3, RS1, RS2**;

I-Type: Usado para operações imediatas, *load* e *branch*.

Opcode	RD	RS1	Immediate

Exemplo: **ADD IMM RS2, RS1, #5**.

J-Type: Usado para instruções de salto.

Opcode	Offset Value

Exemplo: **JUMP #100**.

Por que esses formatos são eficientes?

Simplificam a decodificação e agilizam o *pipeline*.

Organização da apresentação

Mapeando operações do *chip* 74181

Proposed Opcode				**Operations in 74181**						**Operation Performed**
F3	*F2*	*F1*	*F0*	*M*	*S3*	*S2*	*S1*	*S0*	C_{-1}'	
0	0	0	0	0	0	0	0	0	1	No Operation
0	0	0	1	0	0	1	1	0	0	A - B
0	0	1	0	0	1	0	0	1	1	A + B
0	0	1	1	0	1	1	1	1	1	Add Immediate
0	1	0	0	0	0	0	0	0	1	Circular Right Shift
0	1	0	1	0	0	0	0	0	1	Circular Left Shift
0	1	1	0	1	1	0	1	1	x	A AND B
0	1	1	1	1	1	1	1	0	x	A OR B
1	0	0	0	1	0	0	0	1	x	A NOR B
1	0	0	1	1	0	1	0	0	x	A NAND B
1	0	1	0	1	0	1	1	0	x	A XOR B
1	0	1	1	0	1	0	0	1	1	Load
1	1	0	0	0	1	0	0	1	1	Store
1	1	0	1	0	0	0	0	0	1	Branch equal
1	1	1	0	0	0	0	0	0	1	Branch Not Equal
1	1	1	1	0	0	0	0	0	1	Jump

Mapas de *Karnaugh*

F3,F2 \ F1,F0	00	01	11	10
00			1	1
01			1	1
11	1			
10			1	

S3 = F3'.F1 + F2'.F1.F0 + F3.F2.F1'.F0'

F3,F2 \ F1,F0	00	01	11	10
00		1	1	
01			1	1
11				
10				1

S1 = F3'.F2'.F0 + F3'.F2.F1 + F3.F2'.F1.F0'

F3,F2 \ F1,F0	00	01	11	10
00	1		1	1
01	1	1	x	x
11	1	1	1	1
10	x	x	1	x

$(C_{-1})'$ = F0' + F1 + F2 + F3

F3,F2 \ F1,F0	00	01	11	10
00		1	1	
01			1	
11				
10		1		1

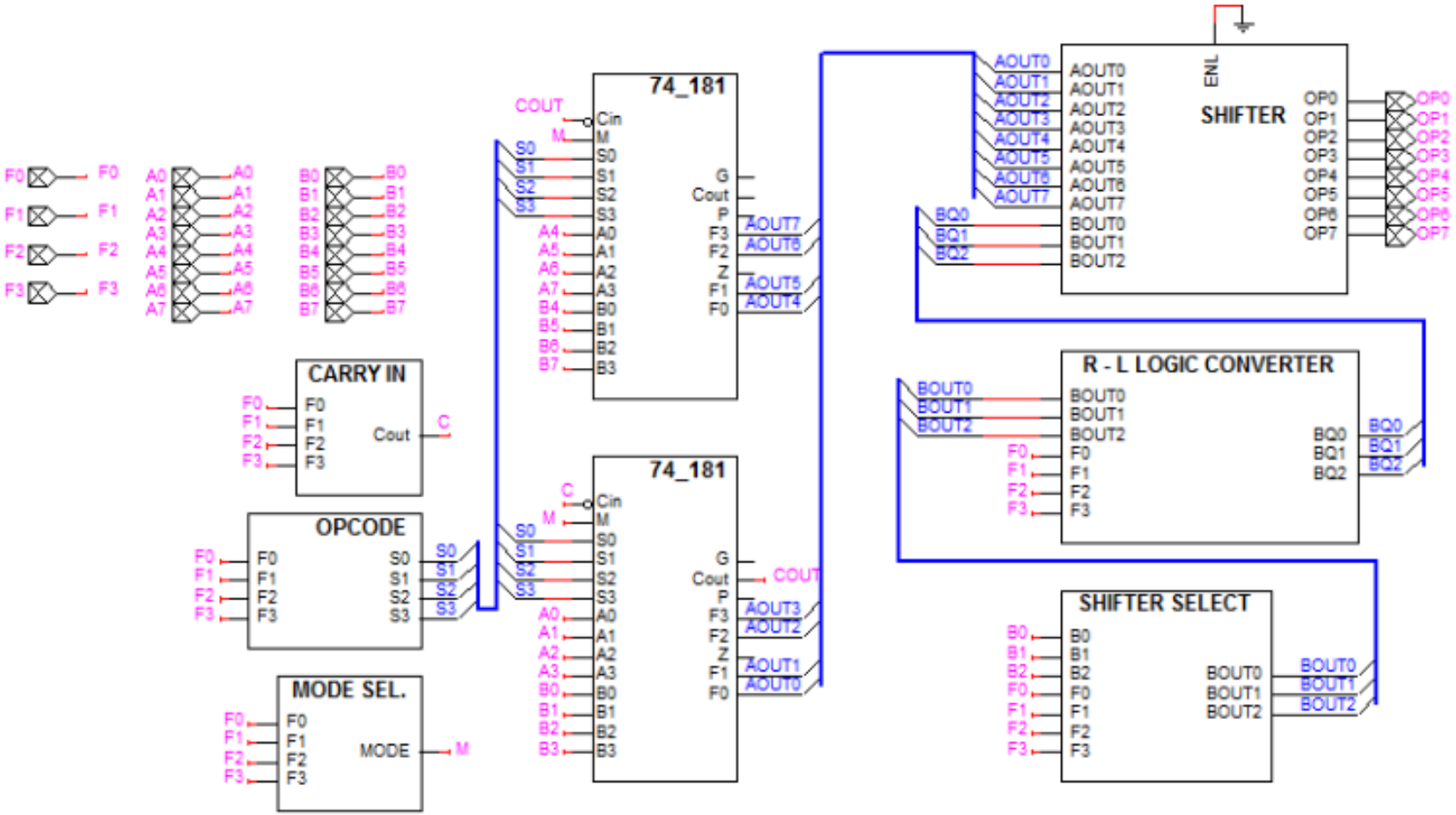
S2 = F2'.F1'.F0 + F3'.F1.F0 + F3.F2'.F1.F0'

F3,F2 \ F1,F0	00	01	11	10
00			1	1
01				1
11	1			
10	1		1	

S0 = F3.F1'.F0' + F2'.F1.F0 + F3'.F1.F0'

F3,F2 \ F1,F0	00	01	11	10
00				
01			1	1
11				
10	1	1		1

Unidade lógica-aritmética



Organização da apresentação

Hazards estruturais

- Os *hazards* estruturais foram resolvidos da forma mais simples possível: não fornecendo as instruções cujas unidades funcionais poderiam causá-los.
- Os algoritmos para **multiplicação** e **divisão**, quando implementados em *hardware*, exigem mais de um ciclo de *clock* para serem executados.
- Este processador não faz operações em **ponto flutuante**, visto que estas também exigiriam unidades funcionais passíveis de causar *hazards* estruturais.

Indução de *bolhas*

A técnica mais simples para tratamento de *hazards* é a geração de *bolhas* no pipeline.
Uma *bolha* consiste em gerar uma instrução vazia, isto é, que não faça nada em nenhum estágio do *pipeline*, de forma a atrasar a execução da instrução causadora do *hazard*.
Todavia, utilizar unicamente esse método prejudica o desempenho do processador, uma vez que isso diminui a taxa de vazão de instruções do *pipeline*.
Técnicas mais inteligentes foram utilizadas pelo autor do artigo para resolver os *hazards* evitando o uso exagerado de *bolhas*.

Hazards de dados

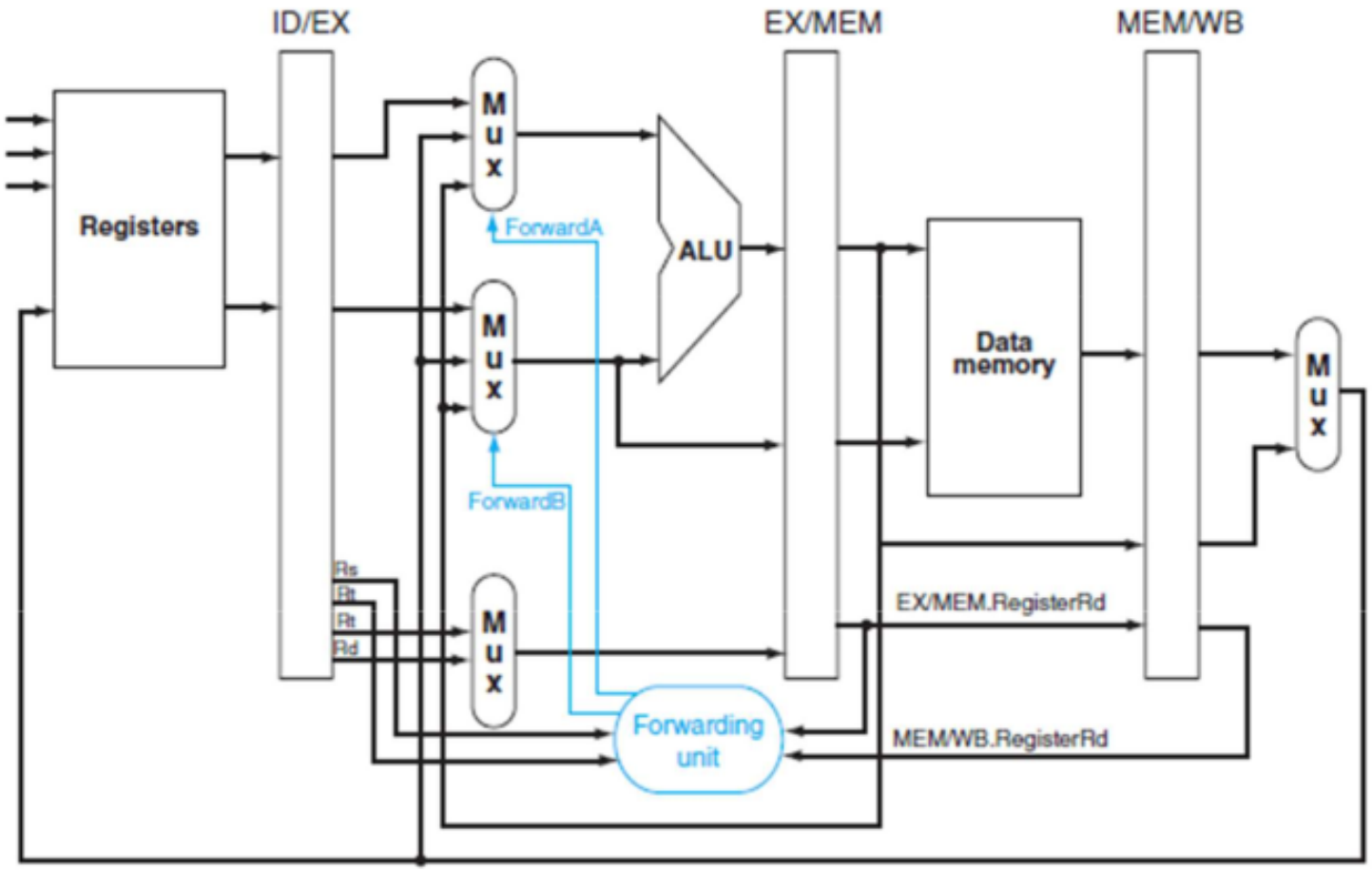
Considere o fragmento de código abaixo:

Exemplo

1. **ADD** RS3, RS1, RS2
2. **SUB** RS5, RS3, RS4

A instrução **SUB** precisa que o valor do registrador **RS3** esteja atualizado ao chegar ao estágio **ID (instruction decode)**, o que cria a necessidade de que a instrução **ADD** esteja no estágio **WB (write back)**.
Caso o tratamento por indução de *bolhas* fosse aplicado aqui, teríamos que gerar 2 delas até que a segunda instrução pudesse ler o valor correto do banco de registradores.
Contudo, outra técnica mais eficiente pode ser aplicada.

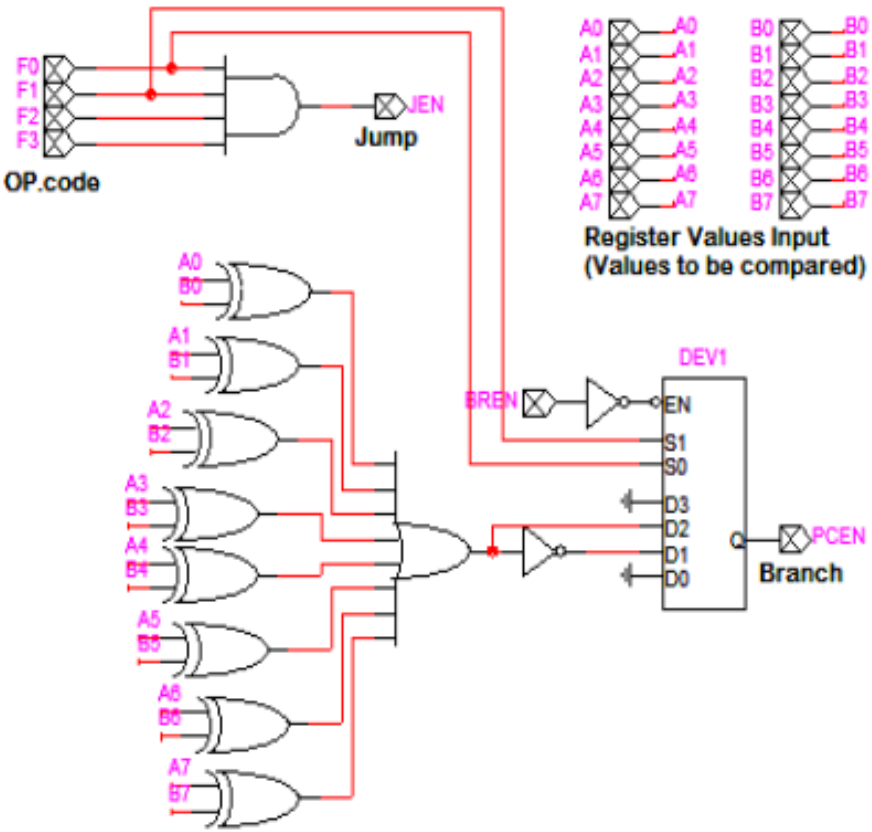
Forwarding de dados



Hazards de controle

O tratamento dado na simulação a *hazards* de controle causados por instruções **BRANCH** e **JUMP** é induzir uma única *bolha* após a emissão dessa instrução no estágio **ID**.
Essa *bolha* é necessária uma vez que um desvio no fluxo de execução do código descoberto na decodificação implica que a instrução recém-buscada no estágio **IF (instruction fetch)** não deve ser executada.
Assim, qualquer ação no estágio **IF** deve ser bloqueada até que a instrução de **JUMP** ou **BRANCH** seja executada e o **PC** seja atualizado com o endereço correto da próxima instrução.

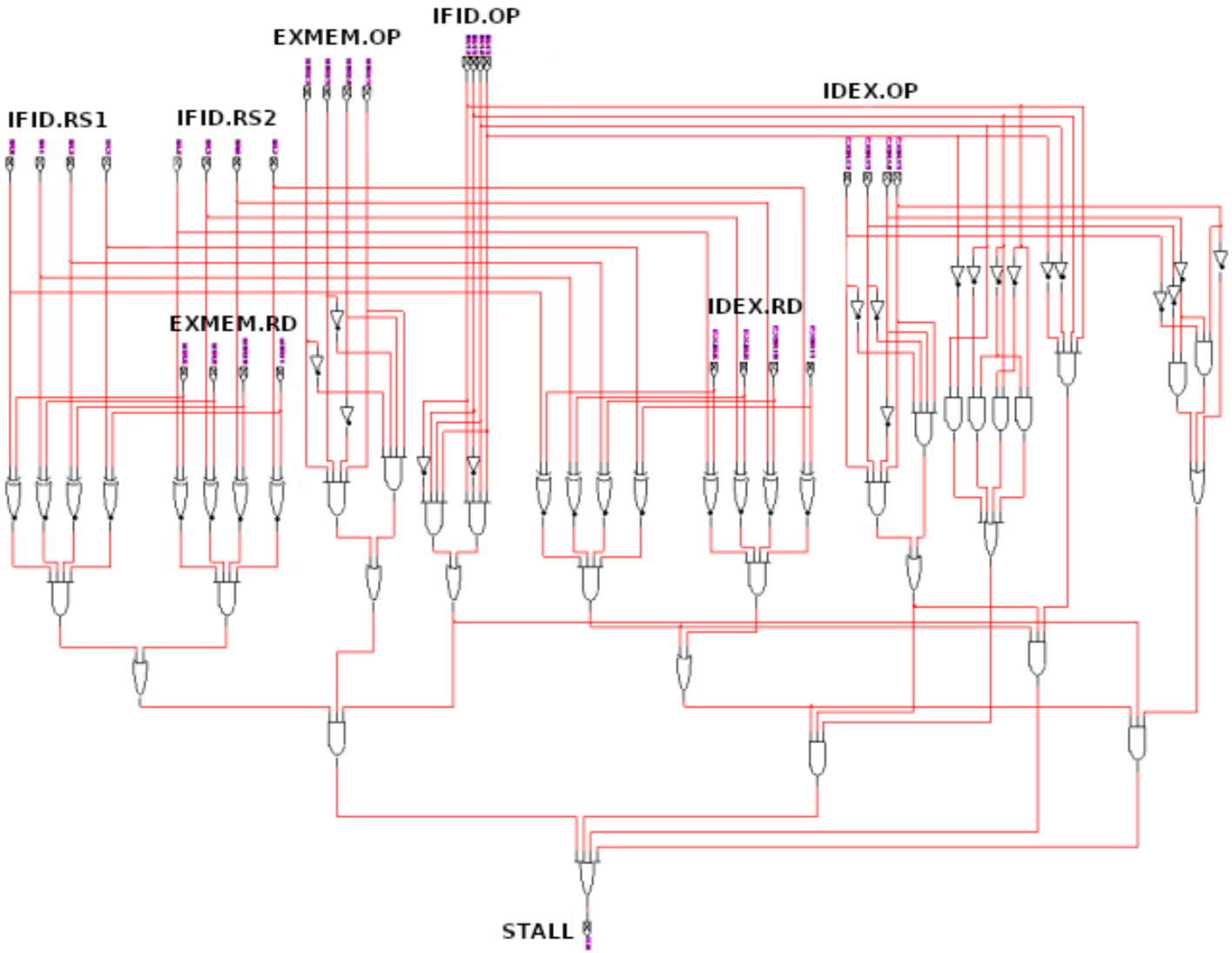
Saltos condicionais



D2	D1	Conclusão
1	0	$A \neq B$
0	1	$A = B$

OPCODE	PCEN
BREQ	D1
BRNEQ	D2

Circuito para indução de *bolhas*



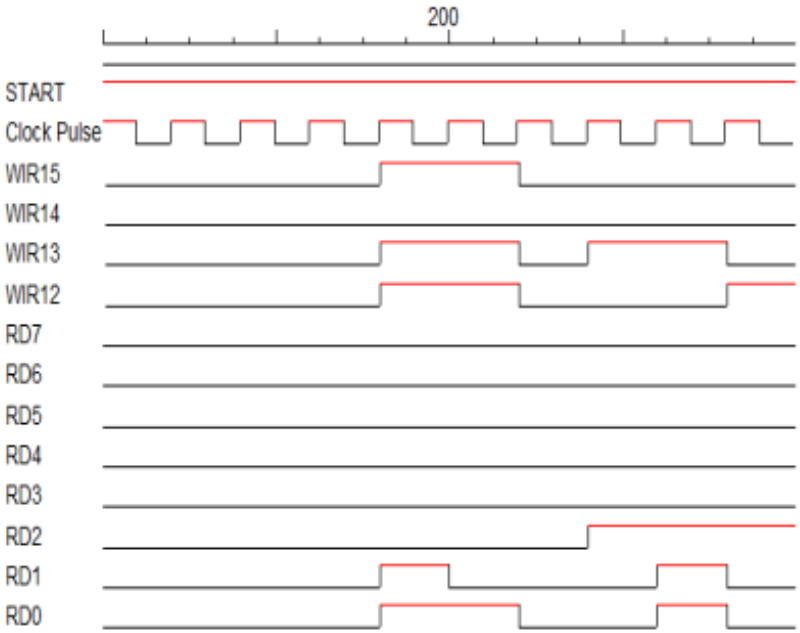
Interpretação do circuito

Casos em que a geração de *bolhas* é inevitável:

- Quando uma instrução lógico-aritmética ou um **BRANCH** é precedido por um **LOAD** cujo registrador-destino é um dos registradores-fonte da operação a ser realizada.
- Quando um **BRANCH** está dois estágios atrás de um **LOAD** cujo registrador-destino é um dos registradores-fonte da comparação a ser realizada.
- Quando um **BRANCH** é precedido por uma instrução de **ULA** cujo registrador-destino é um dos registradores-fonte da comparação a ser realizada. Neste caso, a *bolha* pode ser seguida por um *forwarding* do dado necessário, para evitar que seja gerada uma segunda *bolha*.

Execução de códigos-teste

```
LOAD RS1, #3
LOAD RS2, #1
ADD RS3, RS1, RS2
ADD RS4, RS1, RS3
SUB RS5, RS4, RS1
```




```
LOAD RS1, #3
LOAD RS2, #1
ADD RS3, RS1, RS2
BREQ RS1, RS2, #3
```