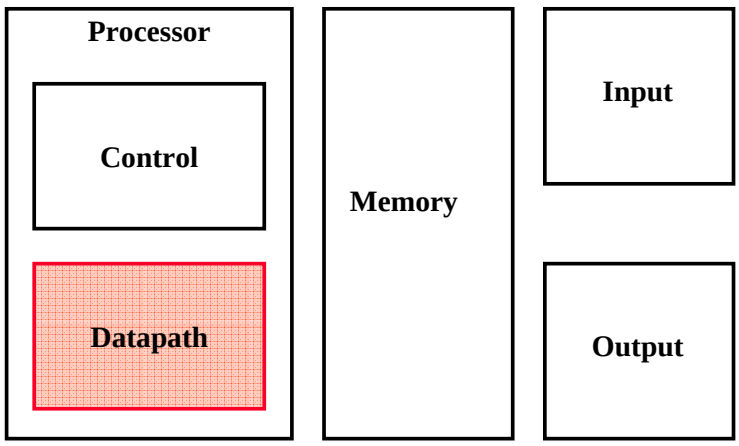
Aula 06: Introdução ao Pipelining, Hazards Estruturais e *Forwarding*

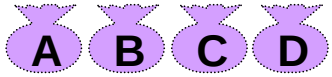
The Big Picture: Where are We Now?

° **The Five Classic Components of a Computer**



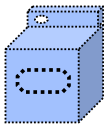
Pipelining: É Natural!

- Lavanderia
- 4 pessoas A, B, C e D possuem 4 sacolas de roupa para lavar, secar e dobrar



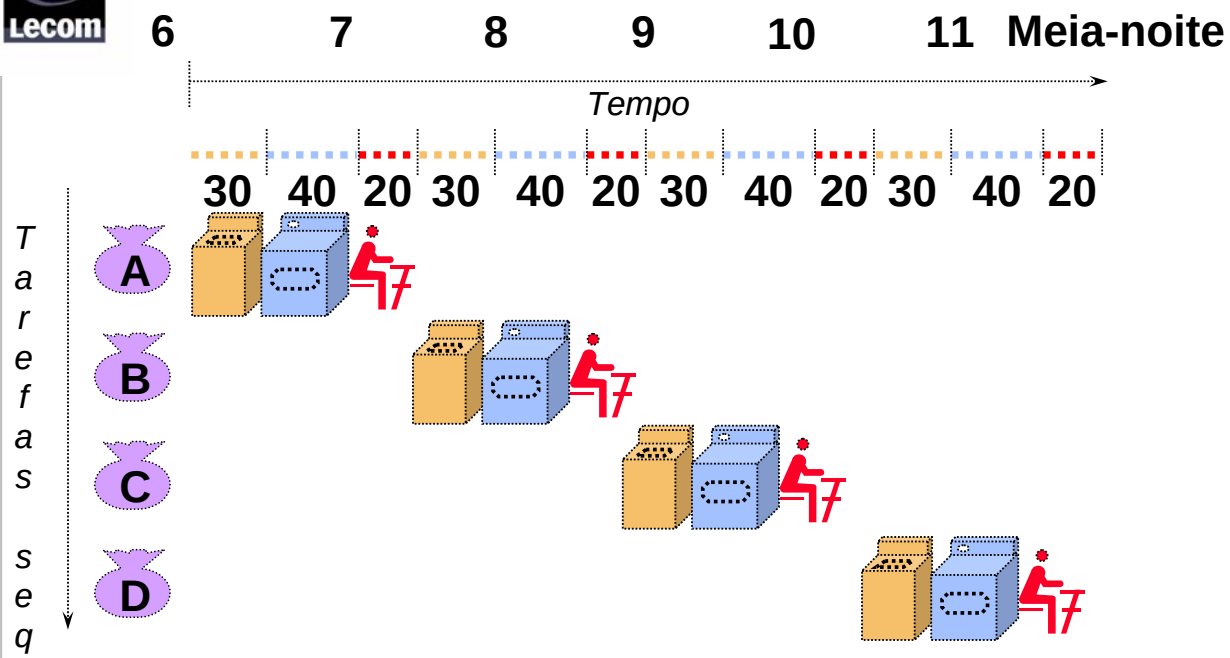
- Lavar leva 30 minutos

- Secar leva 40 minutos



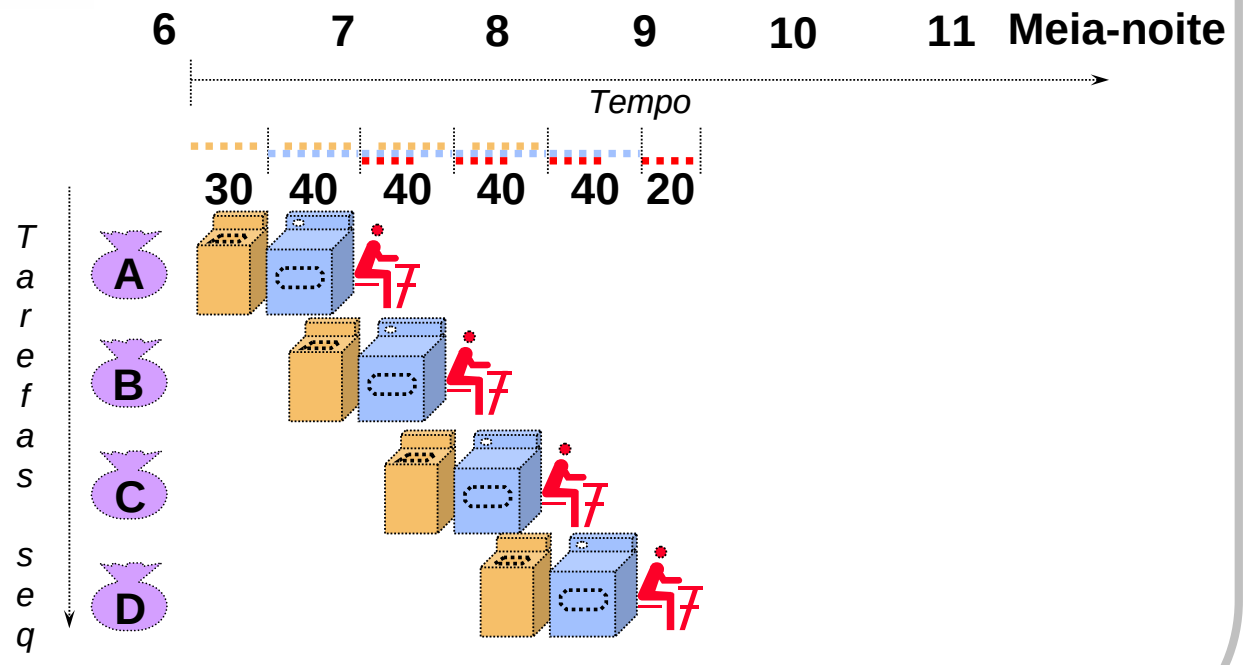
- Dobrar leva 20 minutos

Lavanderia Sequencial



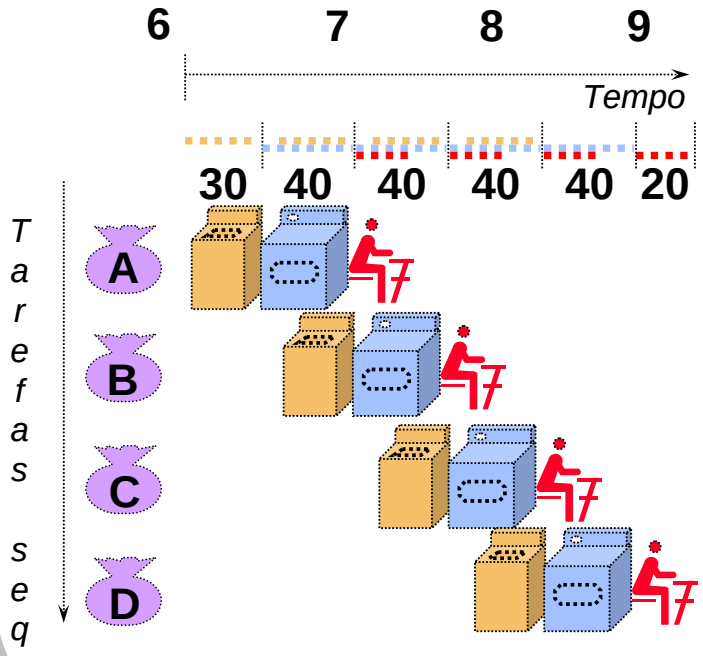
- Lavanderia sequencial leva 6 horas para terminar
- Se eles conhecessem *computação*, quanto tempo levaria?

Lavanderia com Pipelining: Início da Tarefa ASAP



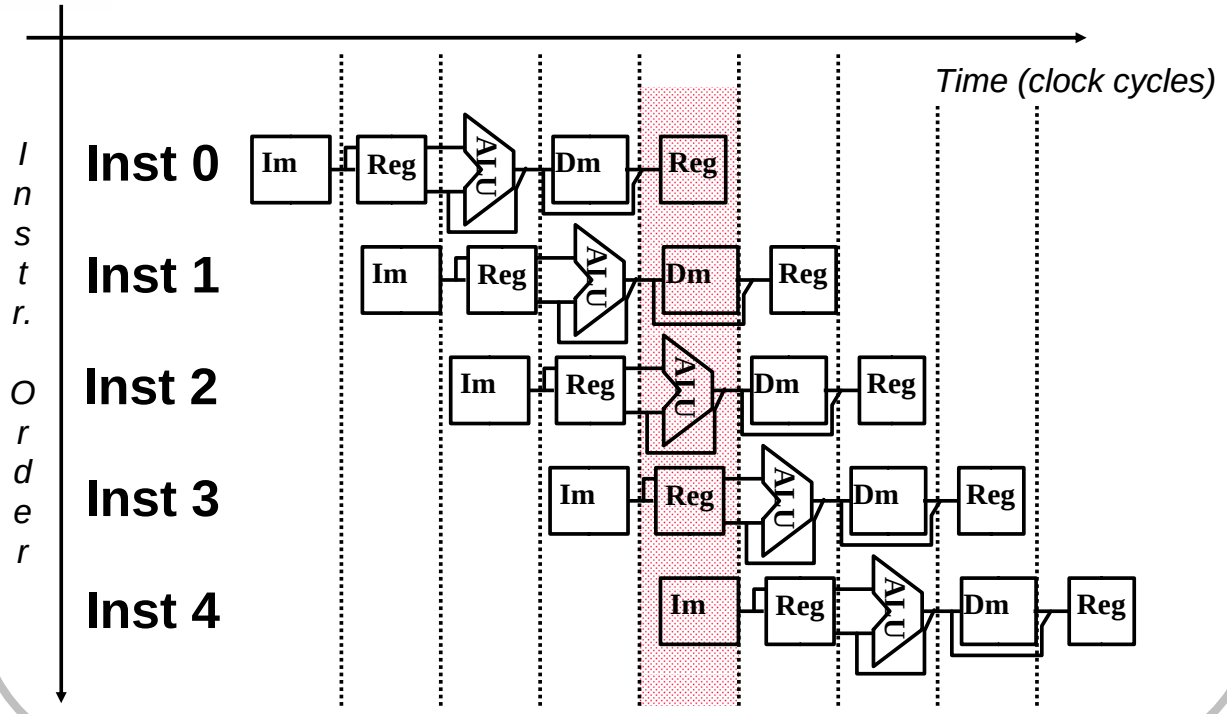
- Lavanderia com *pipelining* leva 3.5 horas !!!

Lições Aprendidas

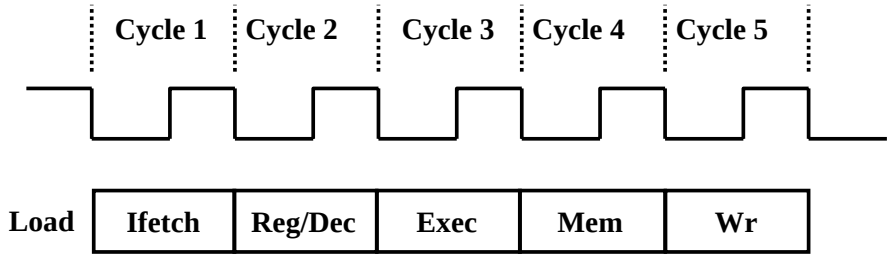


- *Pipelining* não melhora a latência de uma única tarefa, mas melhora o throughput do trabalho todo
- Taxa de inserção de tarefas é limitada pela tarefa mais lenta
- Existem múltiplas tarefas sendo executadas em um dado instante
- SpeedUp potencial = número de estágios
- Tempo para encher o pipeline e tempo de dreno reduzem o speedup

Why Pipeline?
Because the resources are there!

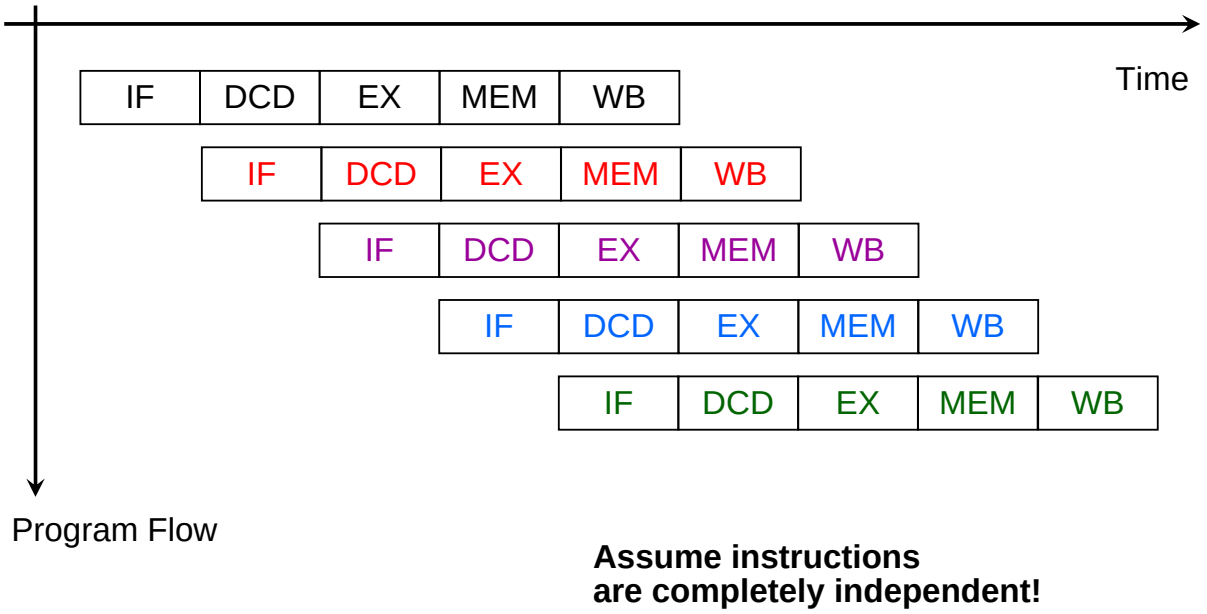


The Five Stages of Load



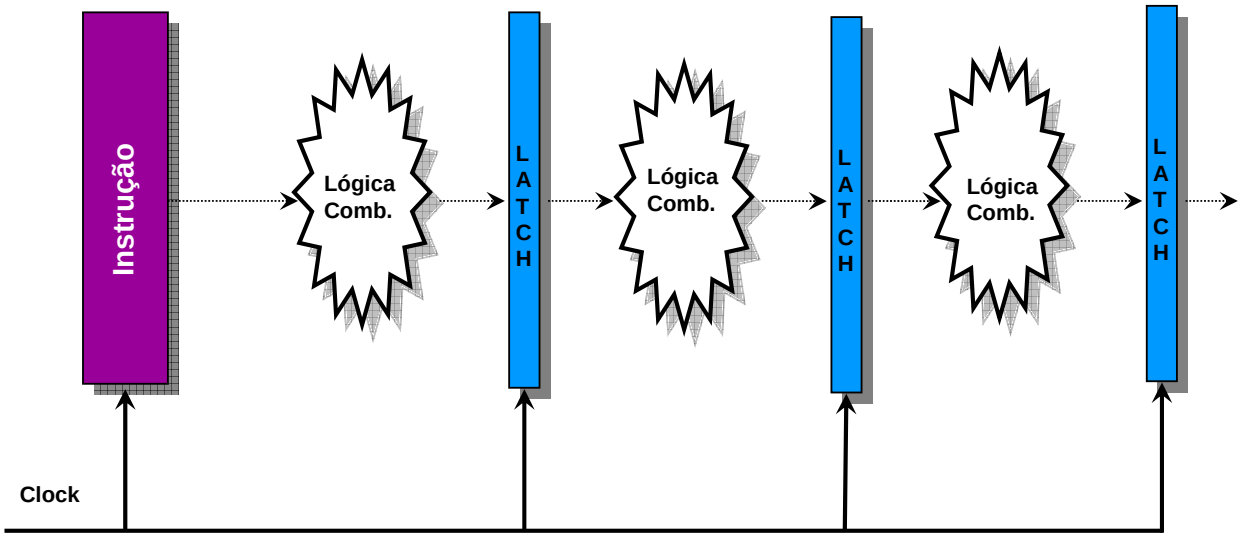
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Ideal Pipelining



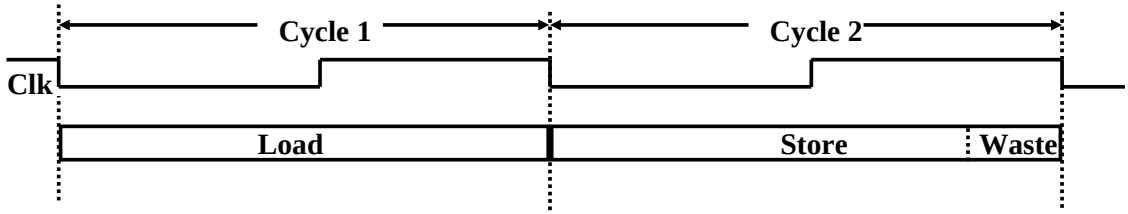
Assume instructions are completely independent!

Implementação de *Pipeline*

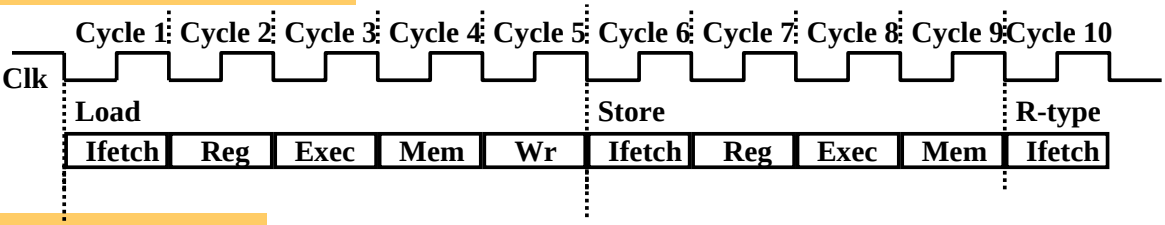


Single Cycle, Multiple Cycle, vs. Pipeline

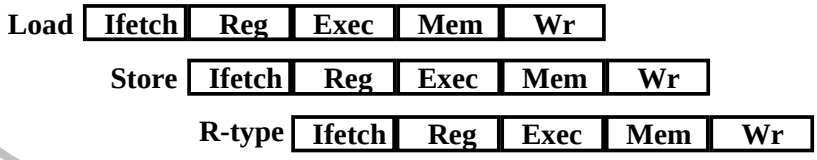
Single Cycle Implementation:



Multiple Cycle Implementation:



Pipeline Implementation:



Ideal Pipelining

$$Maximum\ Speedup \leq Number\ of\ Stages$$

$$Speedup \leq \frac{Time\ for\ unpipelined\ operation}{Time\ for\ longest\ stage}$$

Example:

- 40ns data path,
- 5 stages,
- Longest stage is 10 ns,

Speedup ≤ 4

Why Pipeline?

Suppose we execute 100 instructions

- **Single Cycle Machine**
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- **Multicycle Machine**
  - 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns
- **Ideal Pipelined Machine**
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

Equação de Speedup com *Pipelining*

$$Pipeline\ Speedup = \frac{Tempo\ Medio\ de\ Instrucao\ sem\ pipeline}{Tempo\ Medio\ de\ Instrucao\ com\ pipeline}$$

$$= \frac{CPI_{unpipelined} \times Clock\ Cycle_{unpipelined}}{CPI_{pipelined} \times Clock\ Cycle_{pipelined}}$$

$$= \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{Clock\ Cycle_{unpipelined}}{Clock_{C}ycle_{pipelined}}$$

$$CPI\ Ideal = \frac{CPI_{unpipelined}}{Pipeline\ Depth}$$

$$Speedup = \frac{CPI\ Ideal \times Pipeline\ Depth}{CPI_{pipelined}} \times \frac{Clock\ Cycle_{unpipelined}}{Clock\ Cycle_{pipelined}}$$

Equação de Speedup com *Pipelining*

$$CPI_{pipelined} = CPI\ Ideal + Ciclos\ de\ stalls$$

$$Speedup = \frac{CPI\ Ideal \times Pipeline\ Depth}{CPI\ Ideal + CPI\ stalls} \times \frac{Clock\ Cycle_{unpipelined}}{Clock\ Cycle_{pipelined}}$$

$$Speedup = \frac{Pipeline\ Depth}{1 + CPI\ stalls} \times \frac{Clock\ Cycle_{unpipelined}}{Clock\ Cycle_{pipelined}}$$

Datapath do DLX

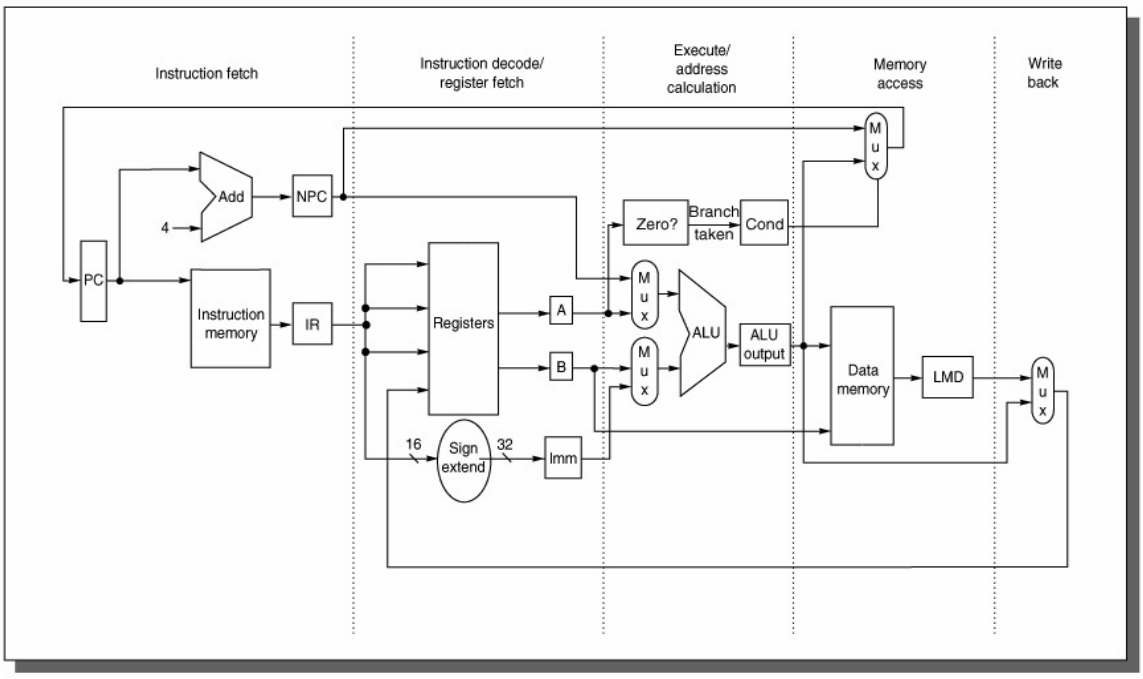


Figura 3.1

Datapath do DLX com *Pipeline*

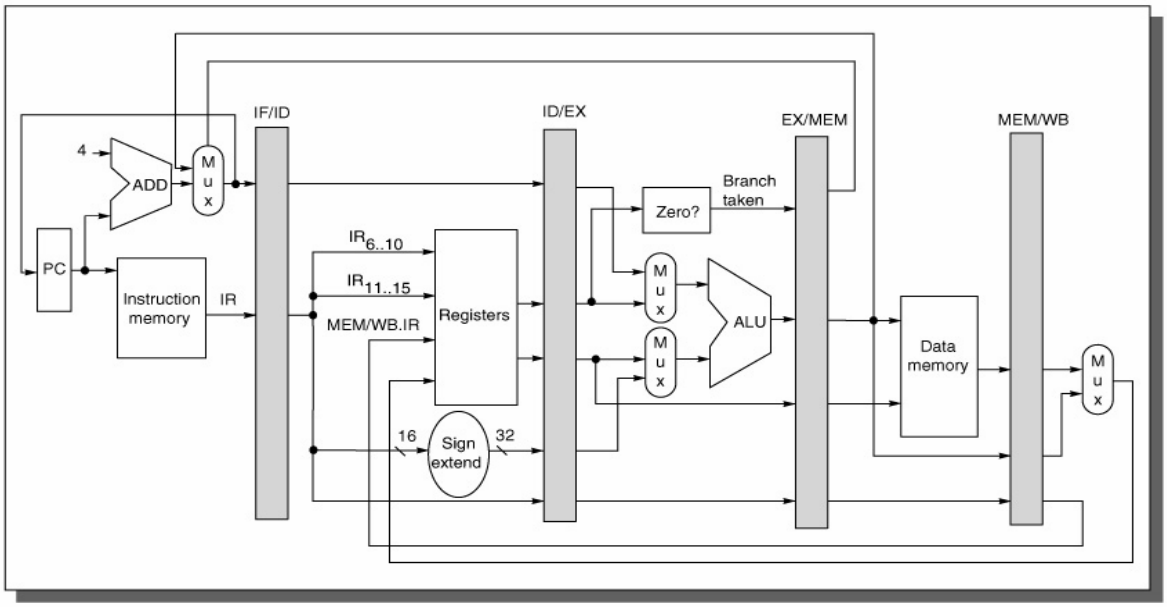


Figura 3.4

- **Decodificação local para uma instrução em cada fase do pipeline**

Visualização do Pipeline

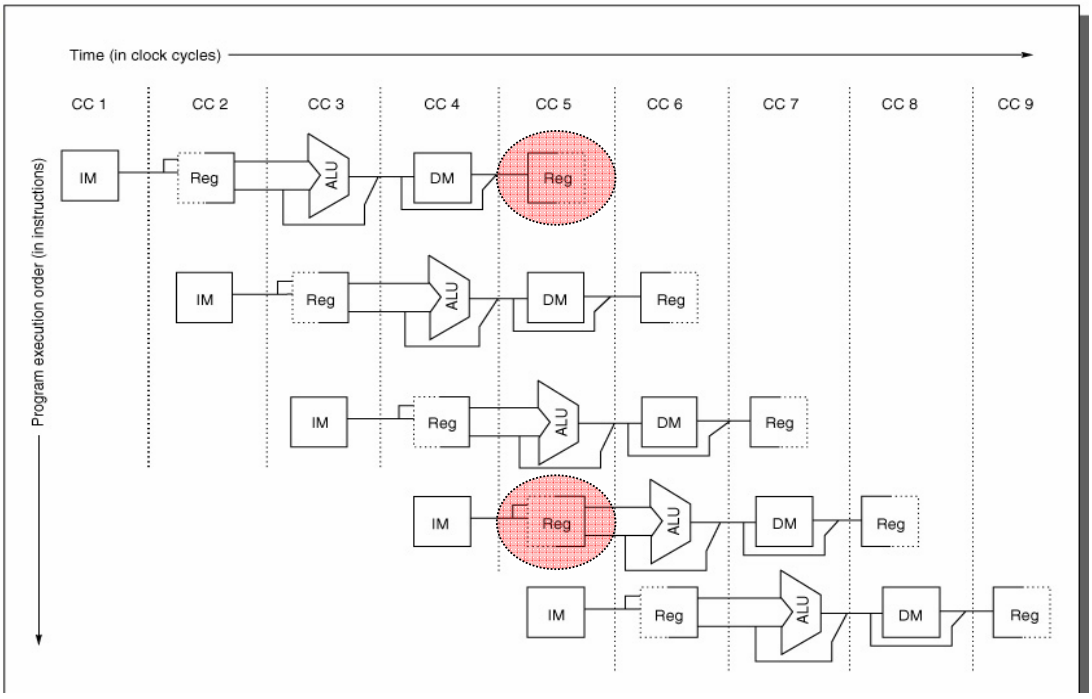


Figura 3.3

Representação Esquemática do *Pipeline*

Instrução	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

Entretanto, *pipeline* ainda não funciona

Lavar roupas ou construir carros é diferente de arquitetura!!!

Diagrama Esquemático do *Pipeline*

Instrução	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr. 1		IF	ID	EX	MEM	WB			
Instr. 2			IF	ID	EX	MEM	WB		
Stall				-	-	-	-	-	
Instr. 3					IF	ID	EX	MEM	WB

Representação alternativa (preferível)

Instrução	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr. 1		IF	ID	EX	MEM	WB			
Instr. 2			IF	ID	EX	MEM	WB		
Instr. 3				-	IF	ID	EX	MEM	WB

VERLab
Visão e Robótica

Can pipelining get us into trouble?

Yes: Pipeline Hazards

- **Structural hazards**
- **Data hazards**
- **Control hazards**

VERLab
Visão e Robótica

Pipeline Hazards

Structural hazards

attempt to use the same resource two different ways at the same time

- **e.g. combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)**

Pipeline Hazards

VERLab
Visão e Robótica

Data hazards

attempt to use item before it is ready

- e.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
- instruction depends on result of prior instruction still in the pipeline

Pipeline Hazards

VERLab
Visão e Robótica

Control hazards

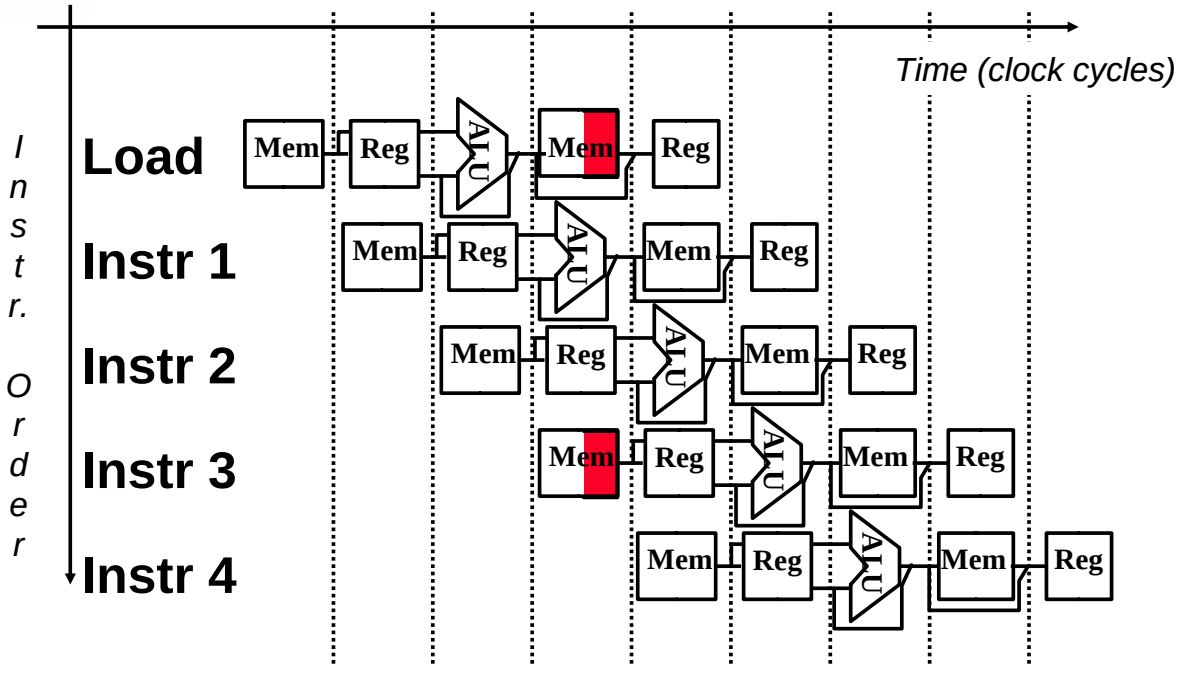
attempt to make a decision before condition is evaluated

- e.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
- branch instructions

Pipeline Hazards

- Can always resolve hazards by *stalling* the pipeline
- Pipeline control must detect the hazard
- Take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards
Porto de Memória Único

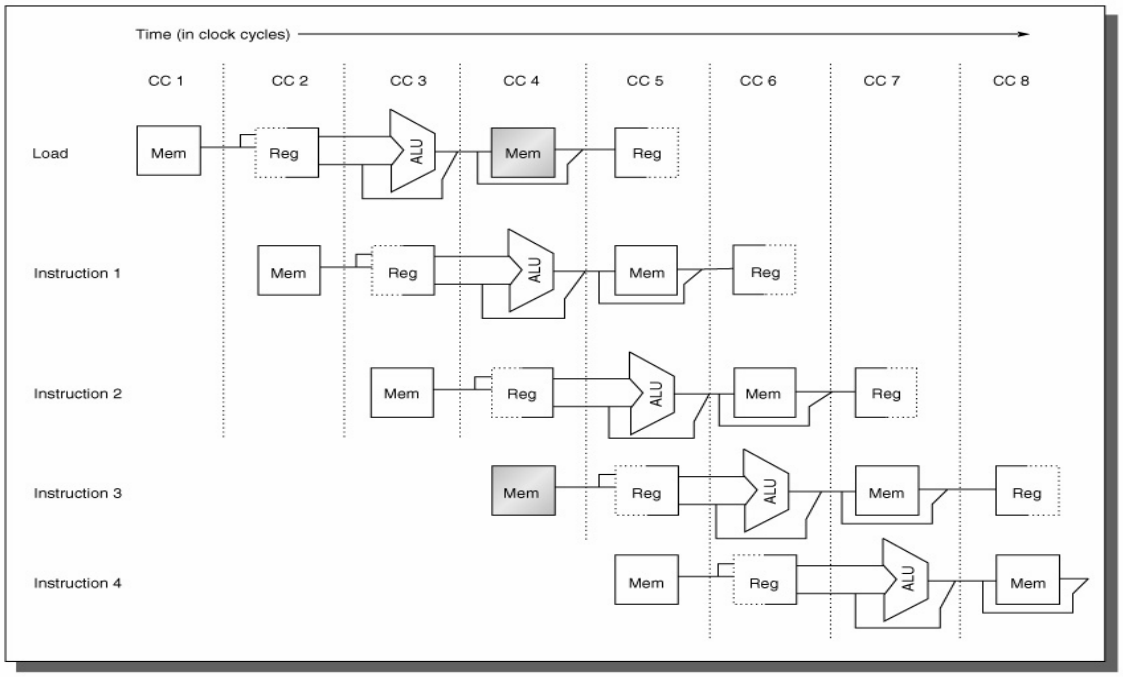


Figura 3.6

Structural Hazards:
Porto de Memória Único

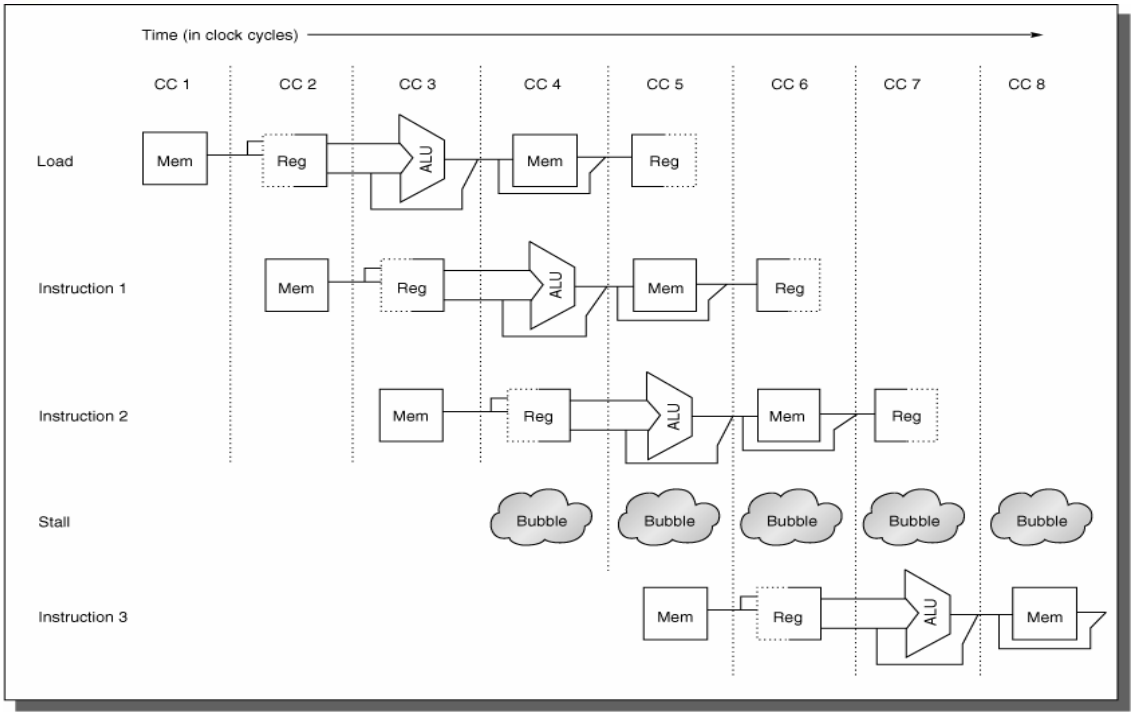


Figura 3.7

Structural Hazards limit performance

Example: if 1.3 memory accesses per instruction and only one memory access per cycle then

- **average CPI = 1.3**
- **otherwise resource is more than 100% utilized**

Exemplo *Control Hazard*: Dois Portos de Memória vs. Porto Único

- Máquina A: Dois portos de memória
- Máquina B: Um porto de memória, mas a implementação do *pipeline* é 1.05 vezes rápida do que A
- CPI Ideal = 1 para A e B
- *Loads* ocorrem com freq. de 40%

$$SpeedUp_A = Pipeline\ Depth/(1 + 0) \times (clock_{unpipe}/clock_{pipe})$$
$$= Pipeline\ Depth$$
$$SpeedUp_B = Pipeline\ Depth/(1 + 0.4 \times 1) \times (clock_{unpipe}/(clock_{unpipe} / 1.05)$$
$$= (Pipeline\ Depth/1.4) \times 1.05$$
$$= 0.75 \times Pipeline\ Depth$$

$SpeedUp_A / SpeedUp_B$ = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

- Máquina A é 1.33 vezes mais rápida

Data Hazard no DLX

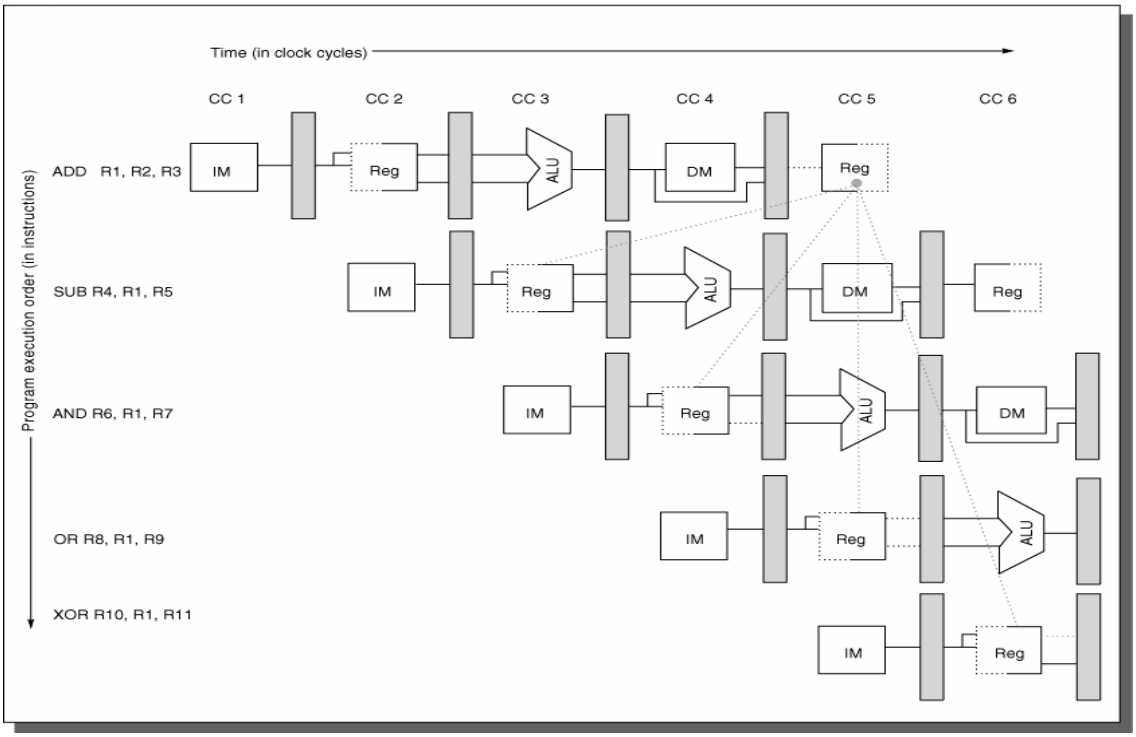


Figura 3.9

Tipos de *Data Hazards*

Instr$_i$ precede Instr$_j$

- *Read After Write* (RAW)
  Instr$_j$ tenta ler operando antes que Instr$_i$ escreva resultado

Tipos de *Data Hazards*

$Instr_i$ precede $Instr_j$

- *Write After Read* (WAR)
  $Instr_j$ tenta escrever resultado antes que $Instr_i$ leia operando
- Não ocorre no *pipeline* do DLX porque:
  - Todas as instruções levam 5 ciclos,
  - Leitura de operandos ocorrem no estágio 2,
  - Escrita de resultados ocorre no estágio 5

Tipos de *Data Hazards*

$Instr_i$ precede $Instr_j$

- *Write After Write* (WAW)
  $Instr_j$ tenta escrever resultado antes que $Instr_i$ escreva
  - Deixa resultado errado
- Não ocorre no *pipeline* do DLX porque :
  - Todas as instruções levam 5 ciclos,
  - Escrita de resultados ocorre no estágio 5
- WAR e WAW aparecerão em variações posteriores do DLX

Tipos de *Data Hazards*

$Instr_i$ precede $Instr_j$

- *E Read After Read* (RAR) ???
  $Instr_j$ tenta ler resultado antes que $Instr_i$ leia operando
  - NÃO CAUSA PROBLEMA ! ! !
- Só existem três tipos: RAW, WAW, WAR
  (pelo menos uma escrita tem que existir)

Data Hazard on r1:

add r1 ,r2,r3

sub r4, r1 ,r3

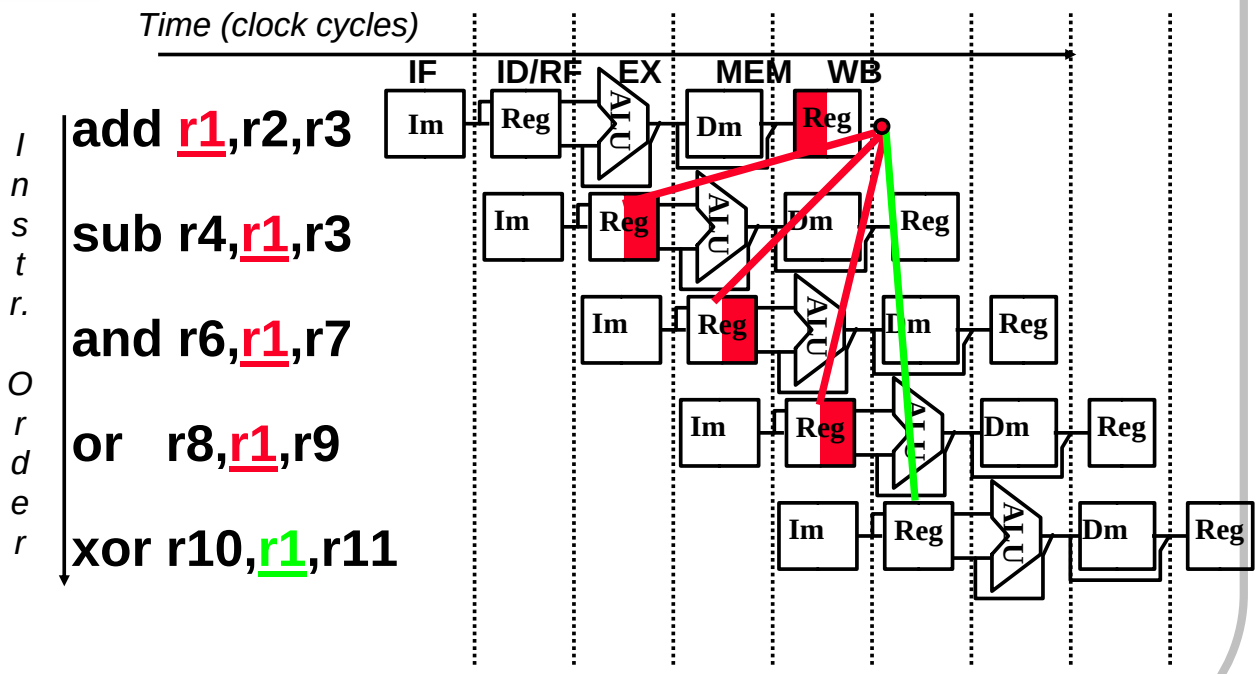
and r6, r1 ,r7

or r8, r1 ,r9

xor r10, r1 ,r11

Data Hazard on r1:

- **Dependencies backwards in time are hazards**



Forwarding para Evitar *Data Hazards*

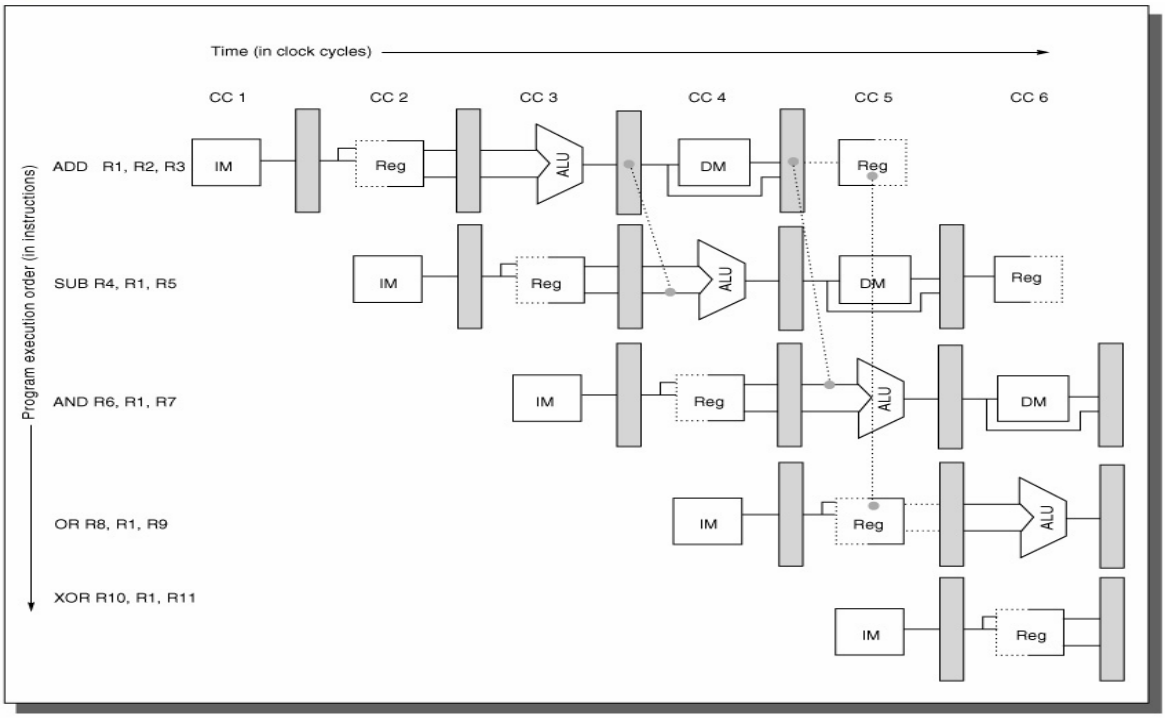


Figura 3.10

Tipos de *Forwarding*

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR_{16..20} = ID/EX.IR_{6..10}$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR_{16..20} = ID/EX.IR_{11..15}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{16..20} = ID/EX.IR_{6..10}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{16..20} = ID/EX.IR_{11..15}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR_{11..15} = ID/EX.IR_{6..10}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR_{11..15} = ID/EX.IR_{11..15}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{11..15} = ID/EX.IR_{6..10}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{11..15} = ID/EX.IR_{11..15}$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{11..15} = ID/EX.IR_{6..10}$
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{11..15} = ID/EX.IR_{11..15}$

Figura 3.19

Mudança de HW para Permitir *Forwarding*

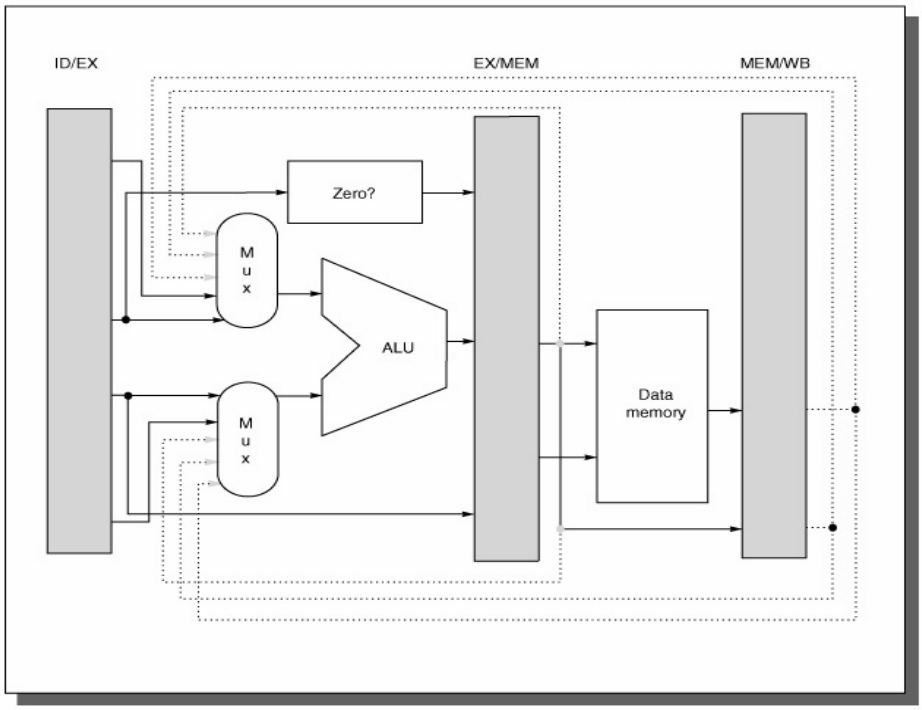


Figura 3.20

Data Hazard com *Forwarding*

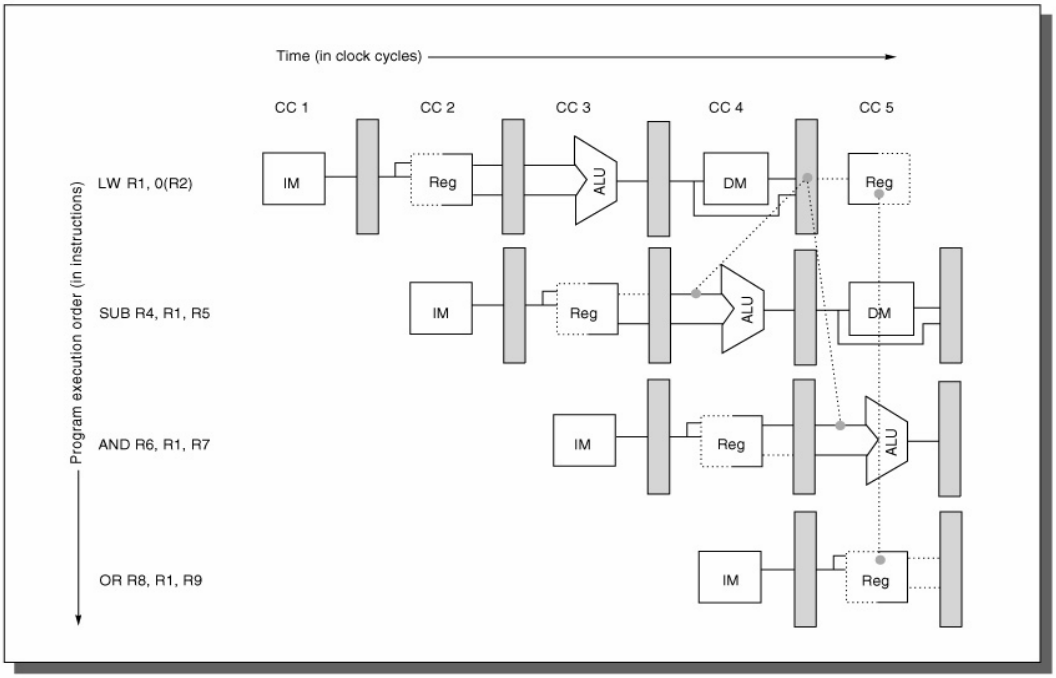


Figura 3.12

Data Hazard com *Forwarding*

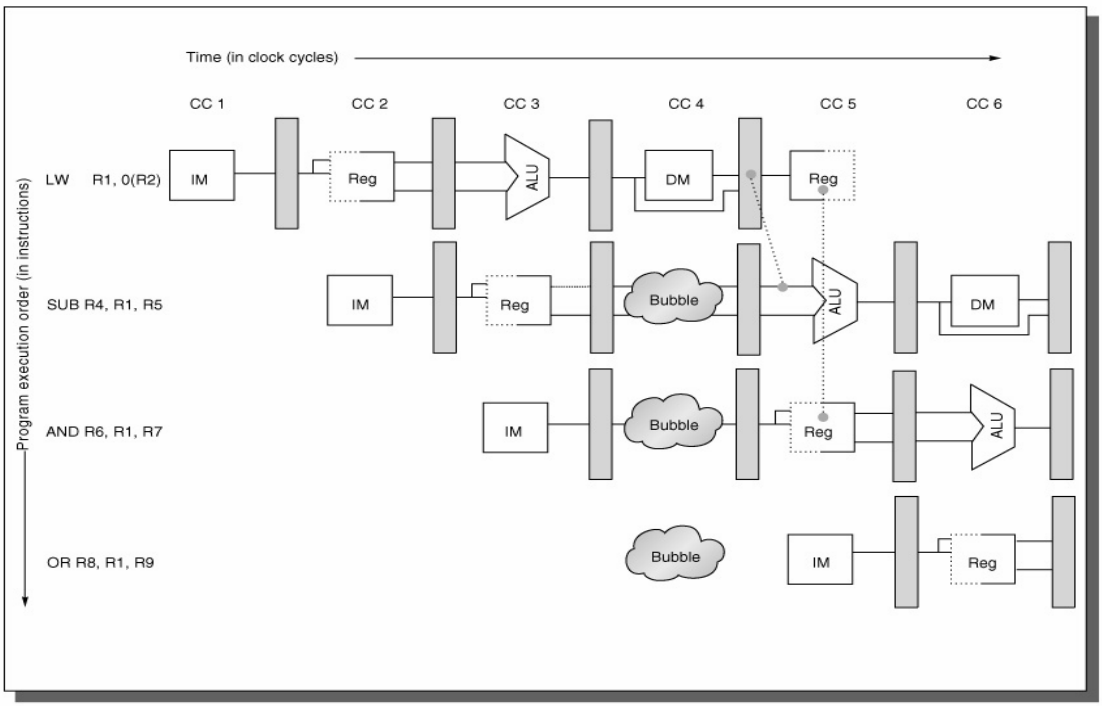
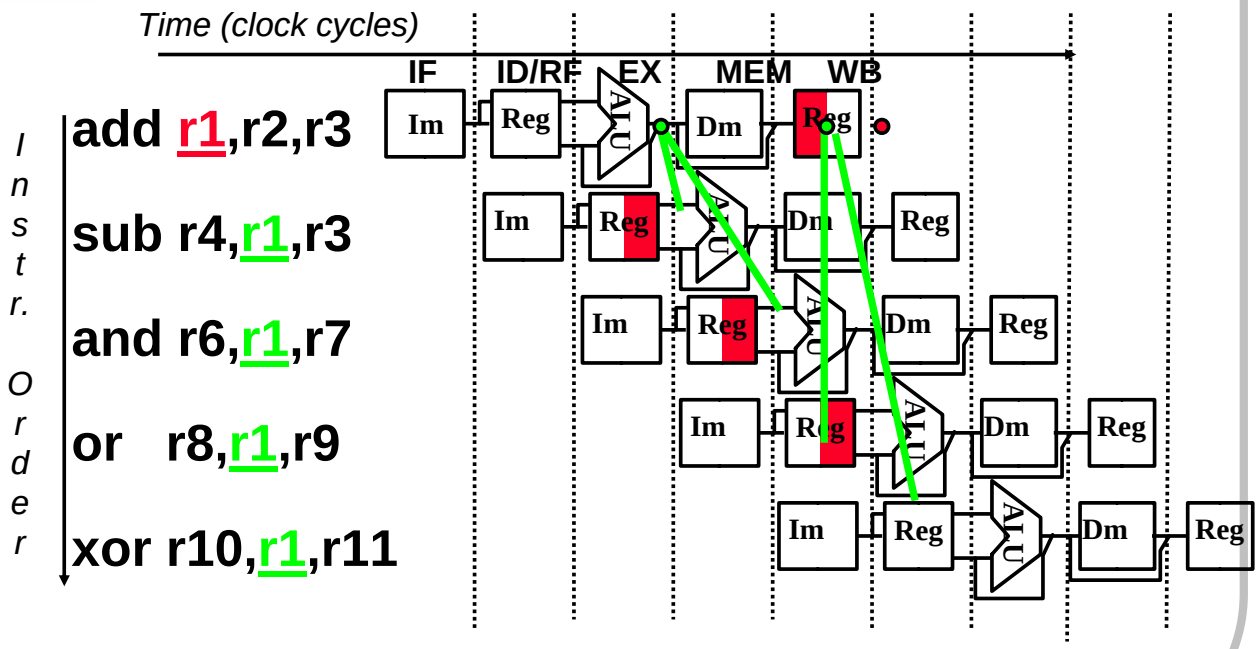


Figura 3.13

Data Hazard Solution:

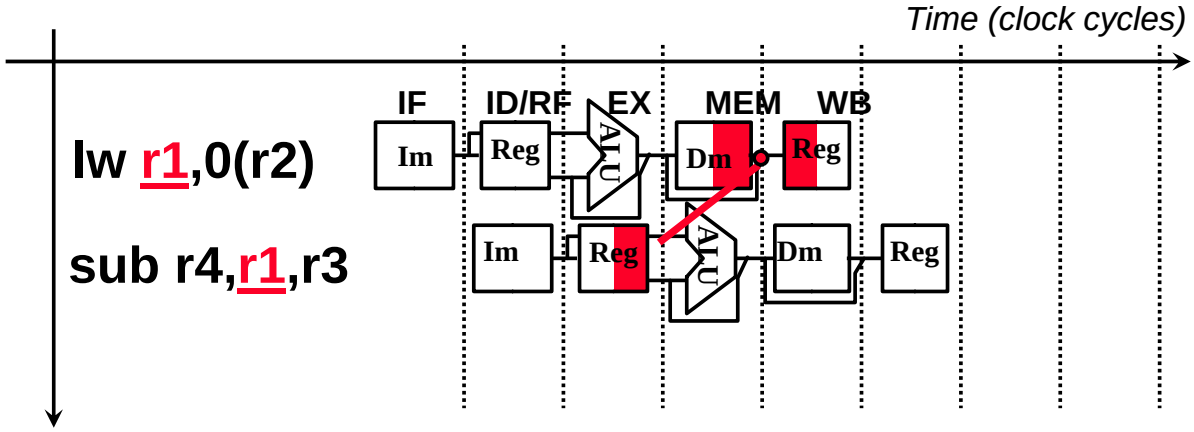
- **"Forward" result from one stage to another**



- **"or" OK if define read/write properly**

Forwarding (or Bypassing): What about Loads?

- **Dependencies backwards in time are hazards**



- **Can't solve with forwarding:**
- **Must delay/stall instruction dependent on loads**

Escalonamento de SW para Evitar *Load Hazards*

Produza o código mais rápido para

a = b + c;

d = e – f;

assumindo que a, b, c, d ,e, f estão na memória.

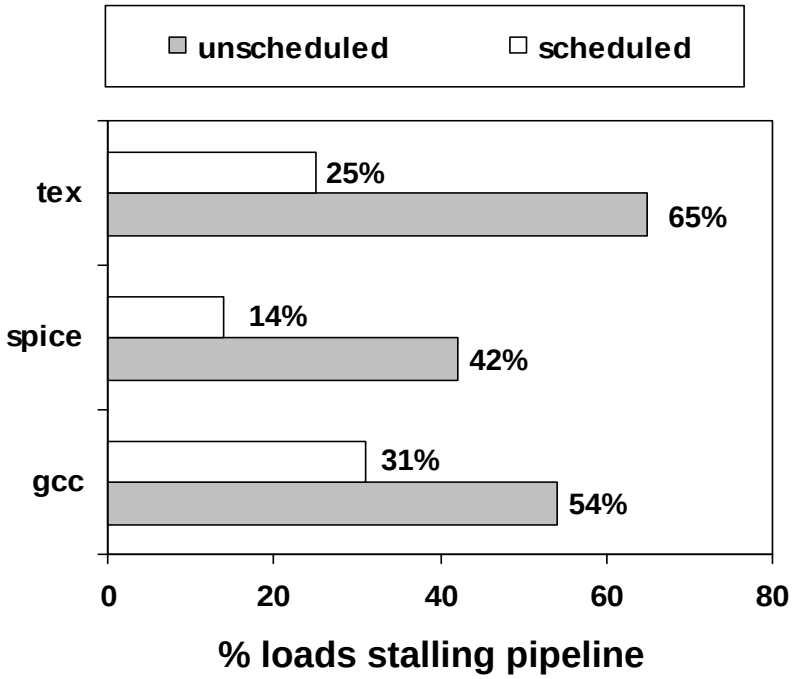
Código lento:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Código rápido:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

Sucesso dos Compiladores para Evitar *Load Stalls*



Resumo: *Pipelining*

- Se tarefas são independentes, é só executar as sub-tarefas concorrentemente (*overlap*)
- Speed Up e Pipeline Depth; se CPI Ideal é 1, então:

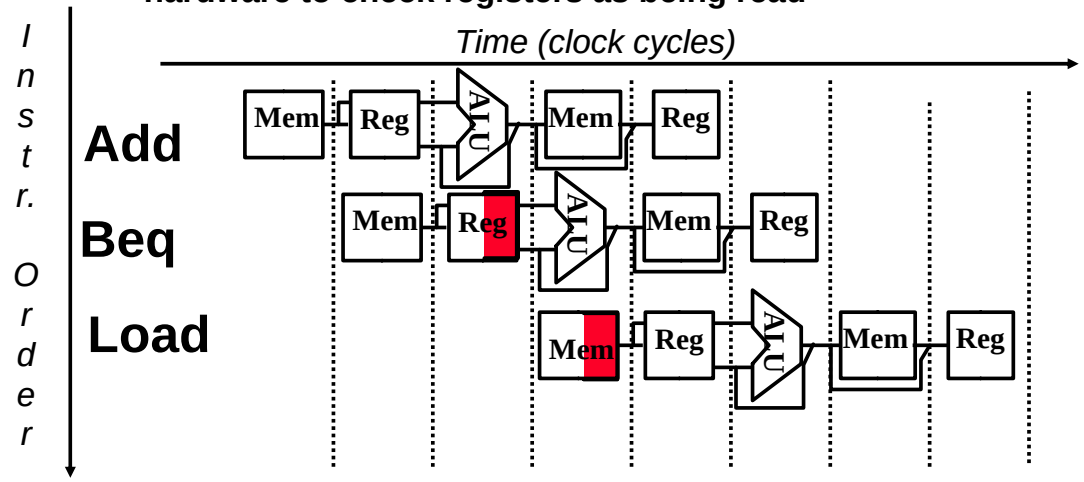
$$Speedup = \frac{Pipeline\ Depth}{1 + CPI\ stalls} \times \frac{Clock\ Cycle\ Unpipelined}{Clock\ Cycle\ Pipelined}$$

- *Hazards* limitam a performance:
  - *Structural*: é necessário alocar mais recursos de HW
  - *Data*: necessita de *forwarding*, e escalonamento de instruções pelo compilador
  - *Control*: a ser discutido

Control Hazard Solutions

Stall: wait until decision is clear

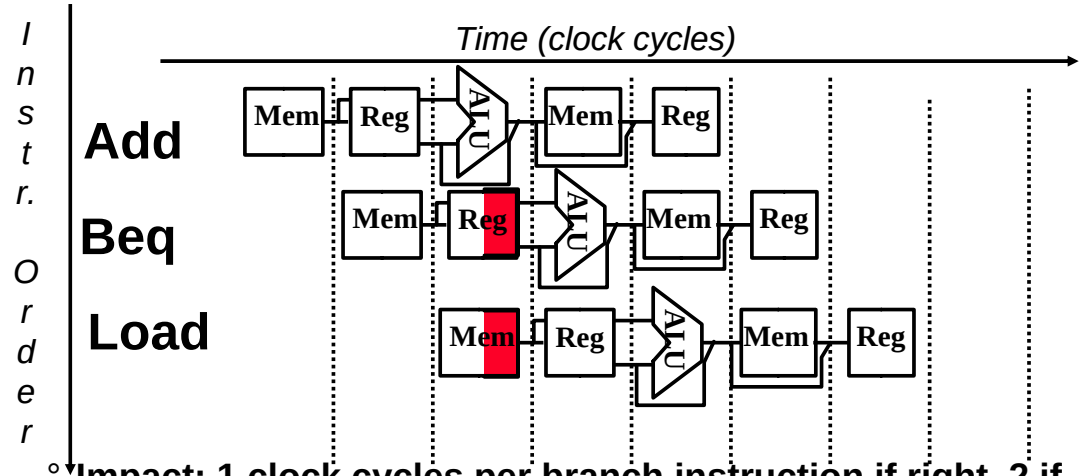
- **Its possible to move up decision to 2nd stage by adding hardware to check registers as being read**



° **Impact: 2 clock cycles per branch instruction => slow**

Control Hazard Solutions

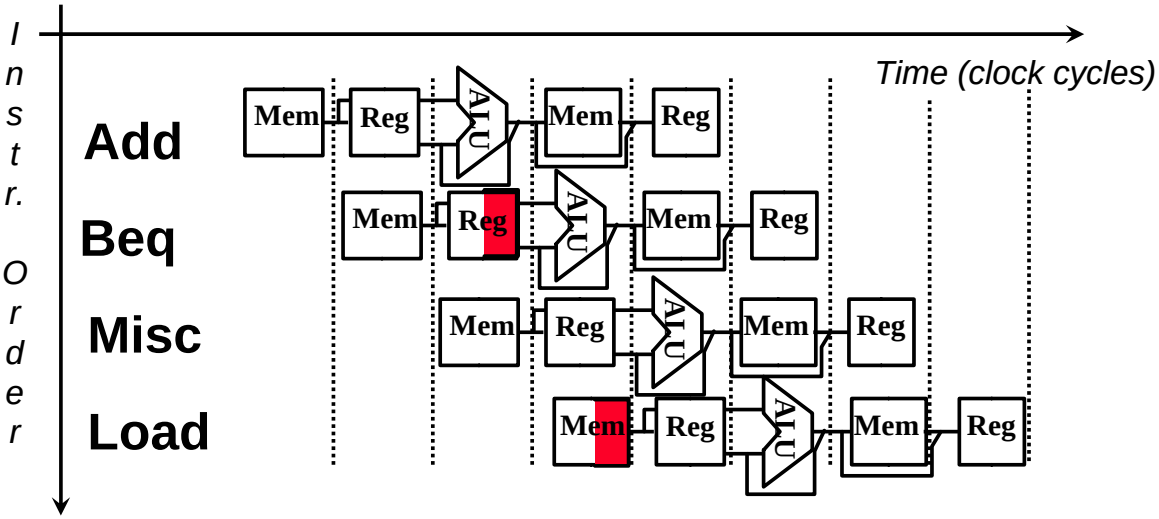
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
- More dynamic scheme: history of 1 branch (- 90%)

Control Hazard Solutions

- Redefine branch behavior (takes place after next instruction) "delayed branch"

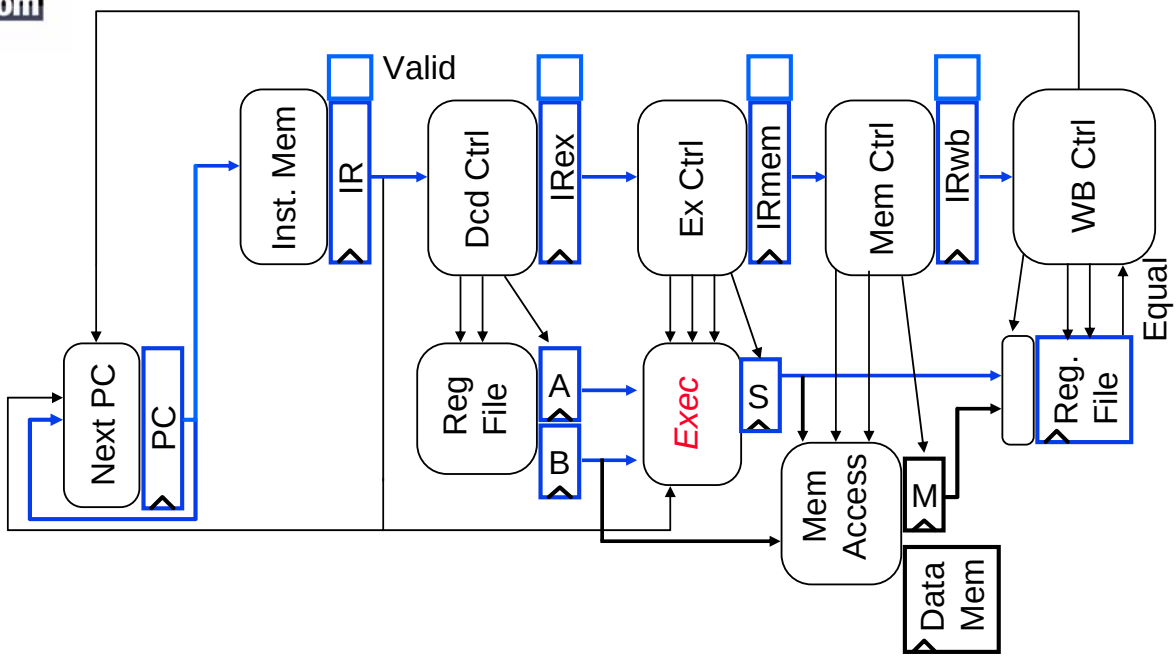


- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)
- As launch more instruction per clock cycle, less useful

Designing a Pipelined Processor

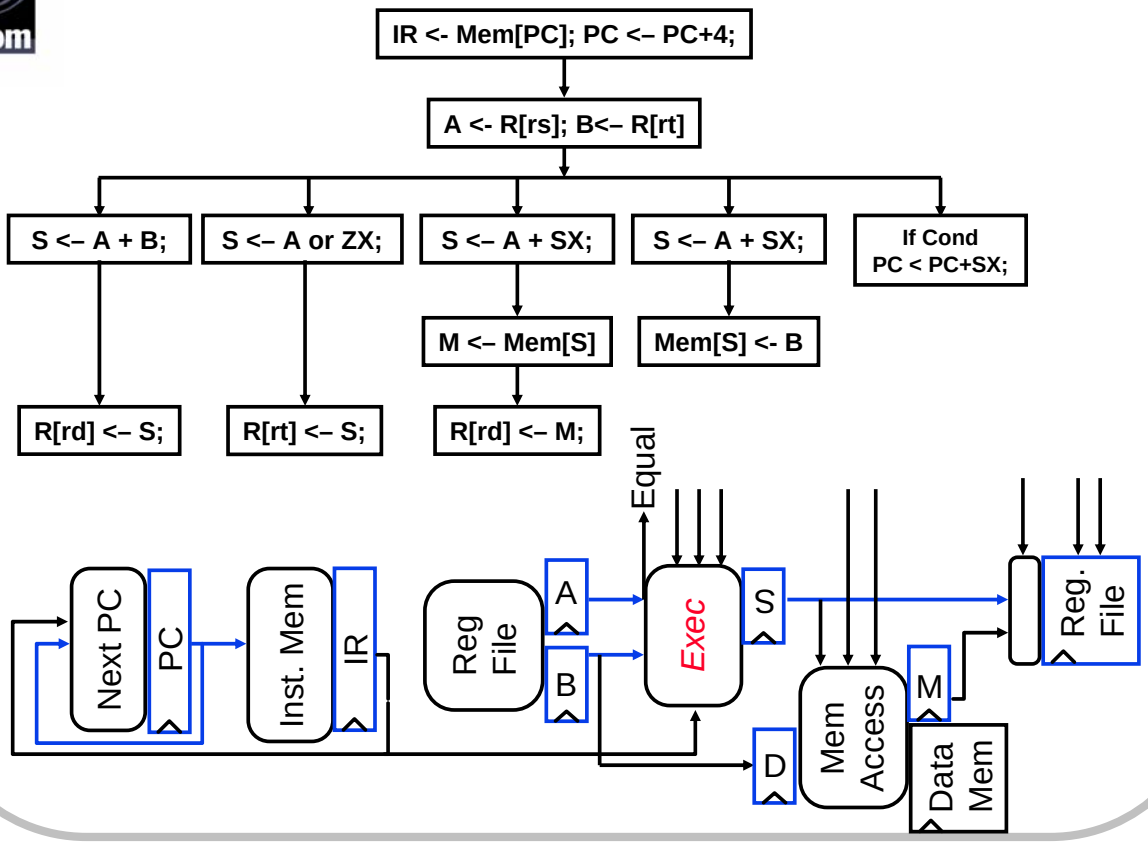
- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve
- assert control in appropriate stage

Pipelined Processor (almost)

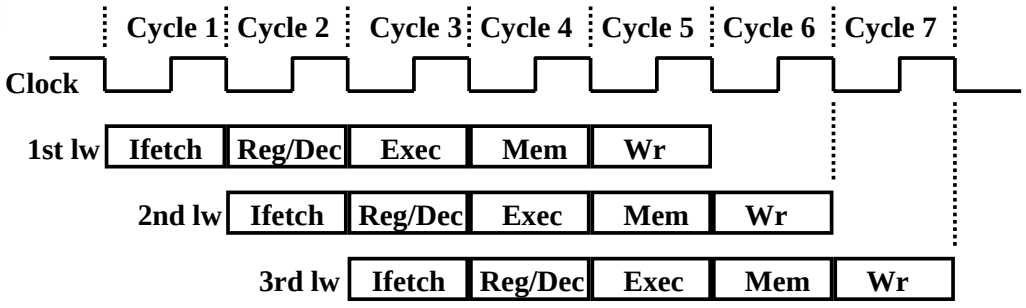


What happens if we start a new instruction every cycle?

Control and Datapath

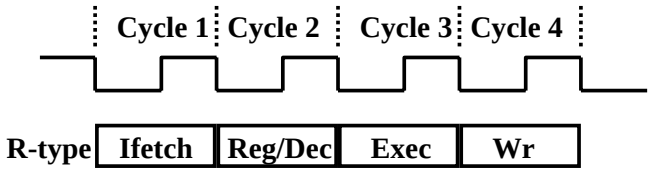


Pipelining the Load Instruction



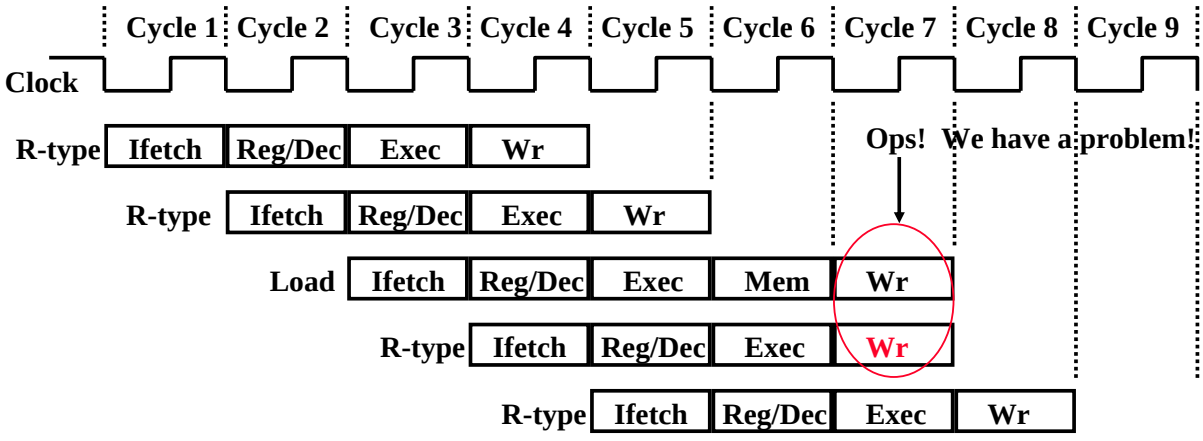
- **The five independent functional units in the pipeline datapath are:**
  - **Instruction Memory for the Ifetch stage**
  - **Register File's Read ports (bus A and bus B) for the Reg/Dec stage**
  - **ALU for the Exec stage**
  - **Data Memory for the Mem stage**
  - **Register File's Write port (bus W) for the Wr stage**

The Four Stages of R-type



- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
  - ALU operates on the two register operands
  - Update PC
- **Wr: Write the ALU output back to the register file**

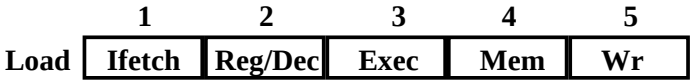
Pipelining the R-type and Load Instruction



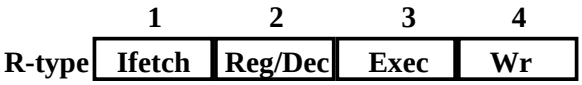
- **We have pipeline conflict or structural hazard:**
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Important Observation

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage

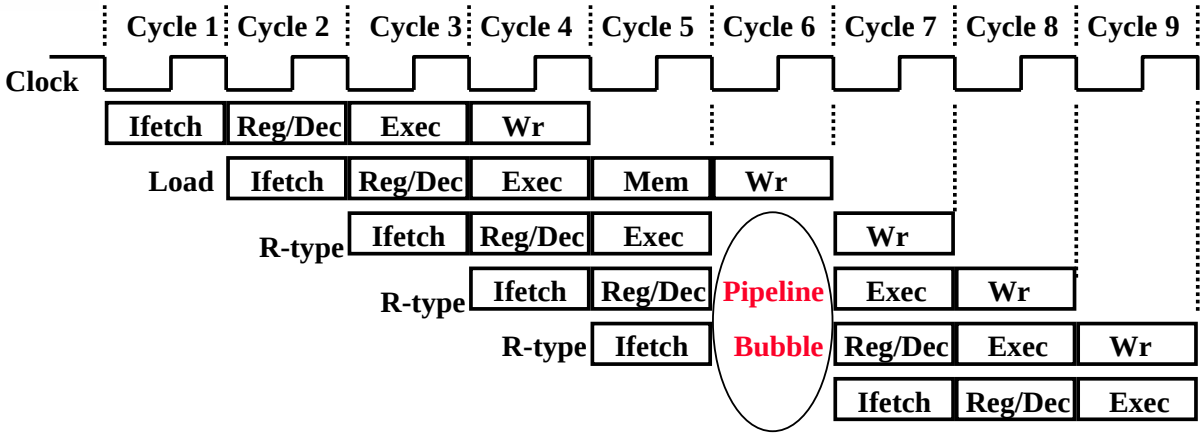


  - R-type uses Register File's Write Port during its 4th stage



- 2 ways to solve this pipeline hazard.

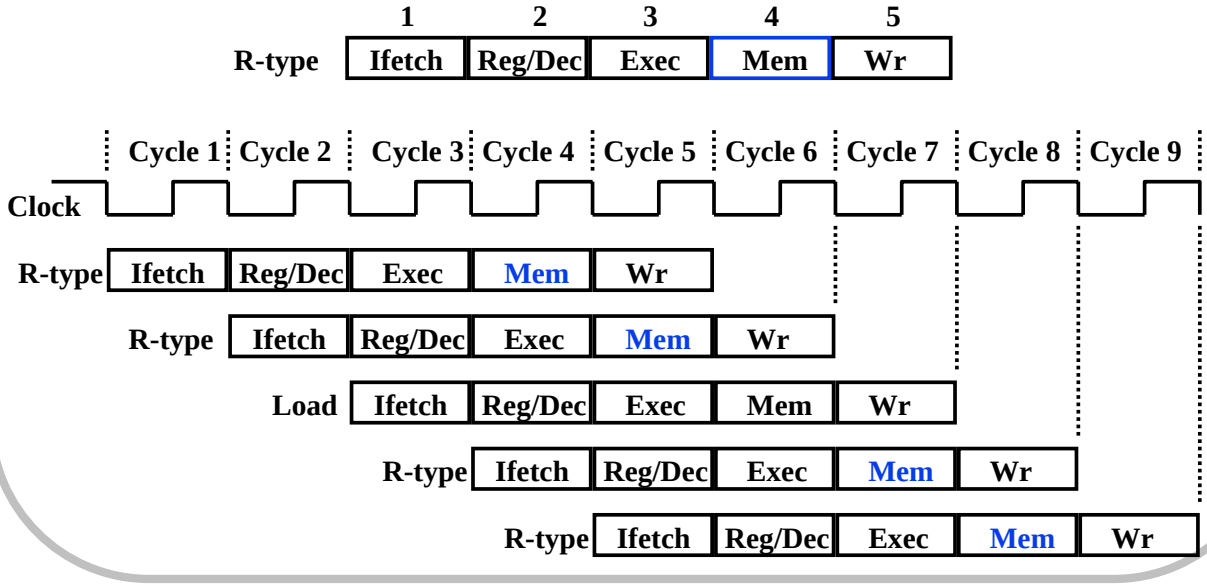
Solution 1: Insert "Bubble" into the Pipeline



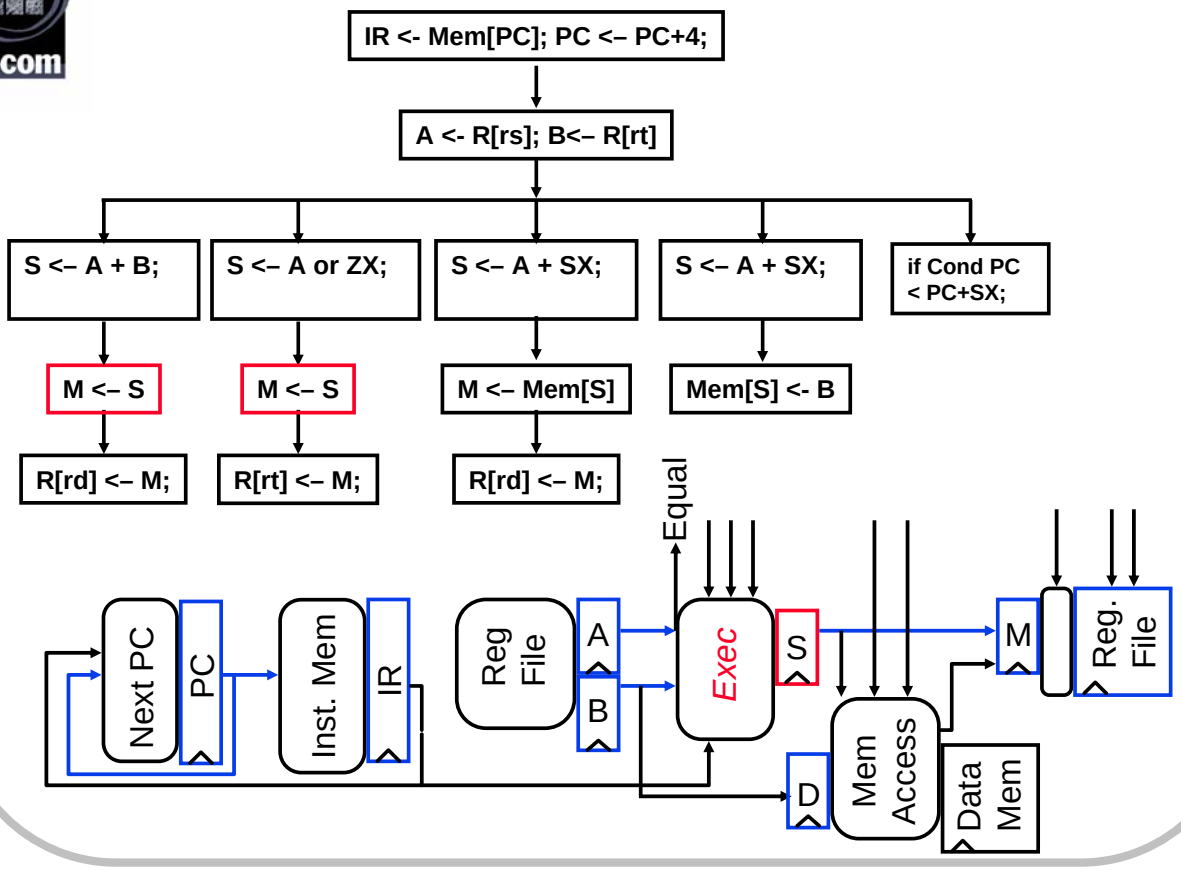
- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write by One Cycle

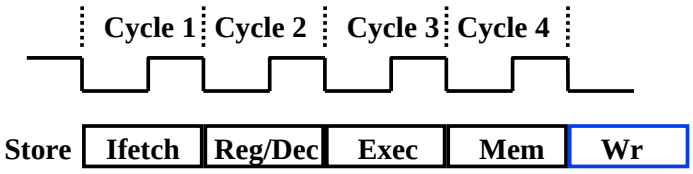
- **Delay R-type's register write by one cycle:**
  - **Now R-type instructions also use Reg File's write port at Stage 5**
  - **Mem stage is a NOOP stage: nothing is being done.**



Modified Control & Datapath



The Four Stages of Store



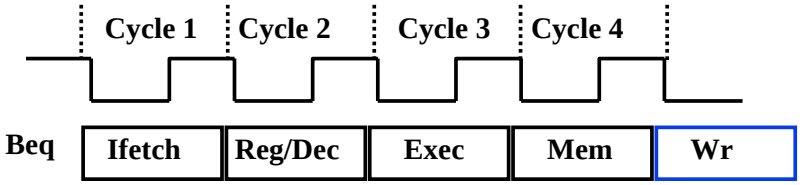
° **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory

° **Reg/Dec: Registers Fetch and Instruction Decode**

° **Exec: Calculate the memory address**

° **Mem: Write the data into the Data Memory**

The Three Stages of Beq

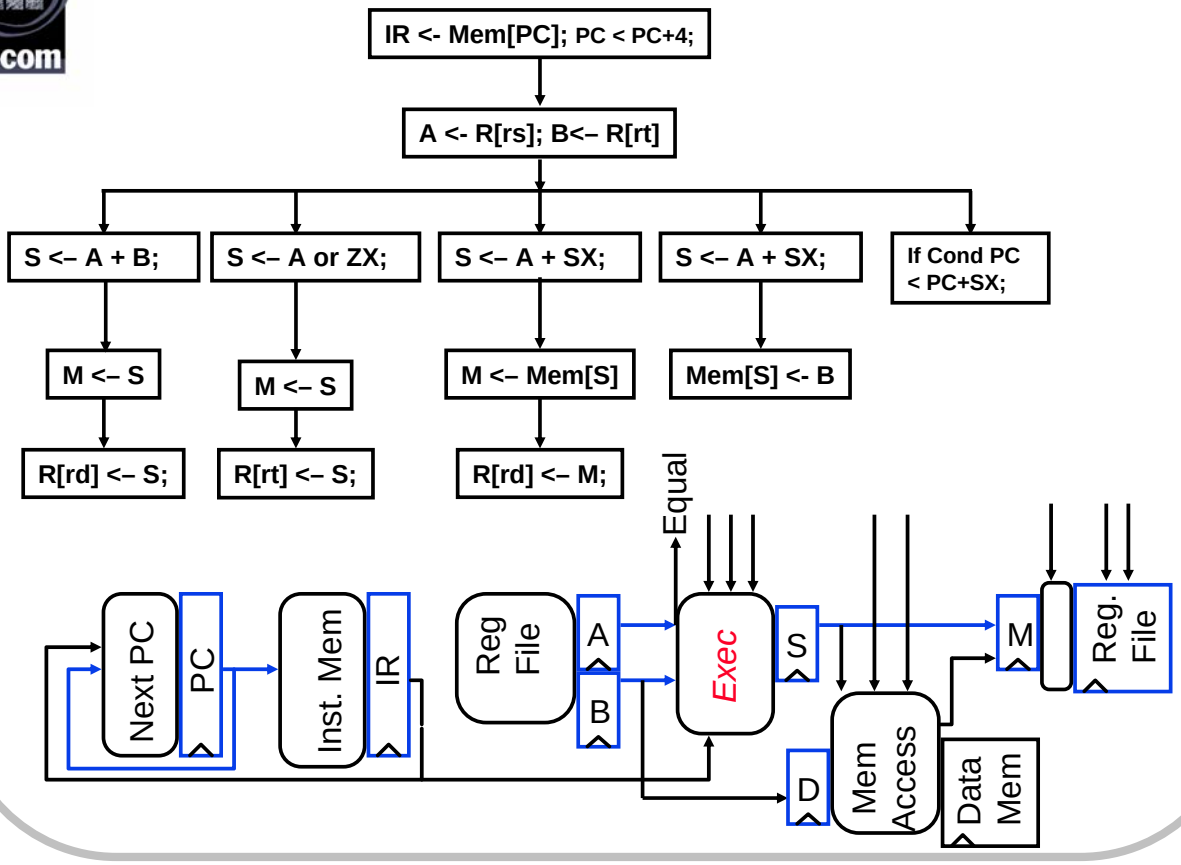


° **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory

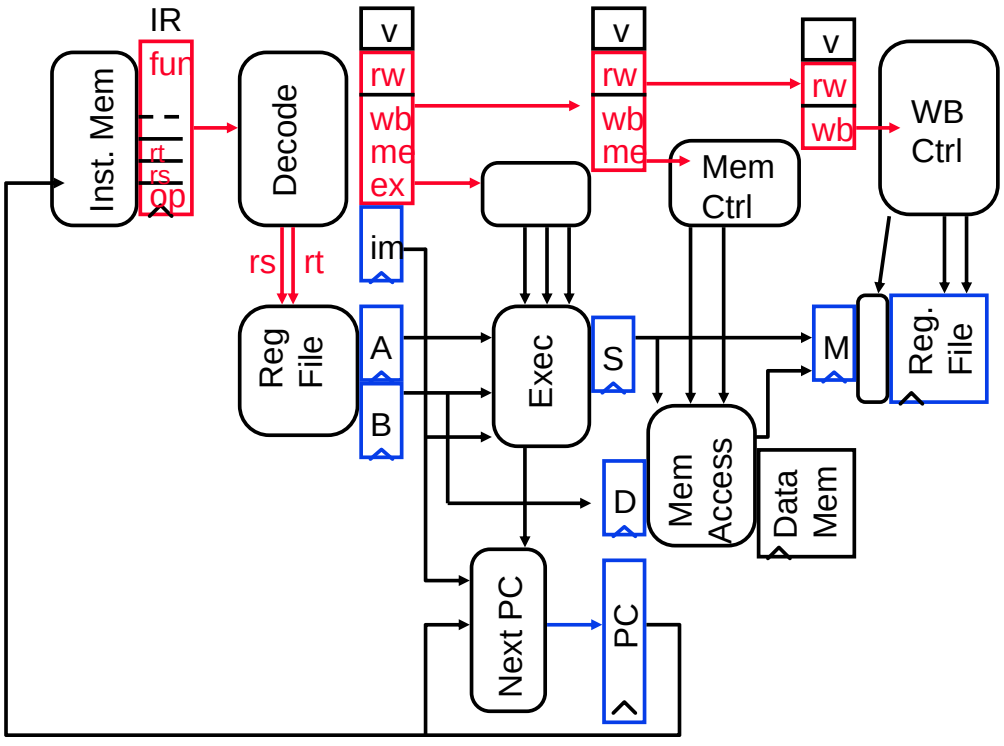
° **Reg/Dec:**
  - Registers Fetch and Instruction Decode

° **Exec:**
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

Control Diagram



Datapath + Data Stationary Control

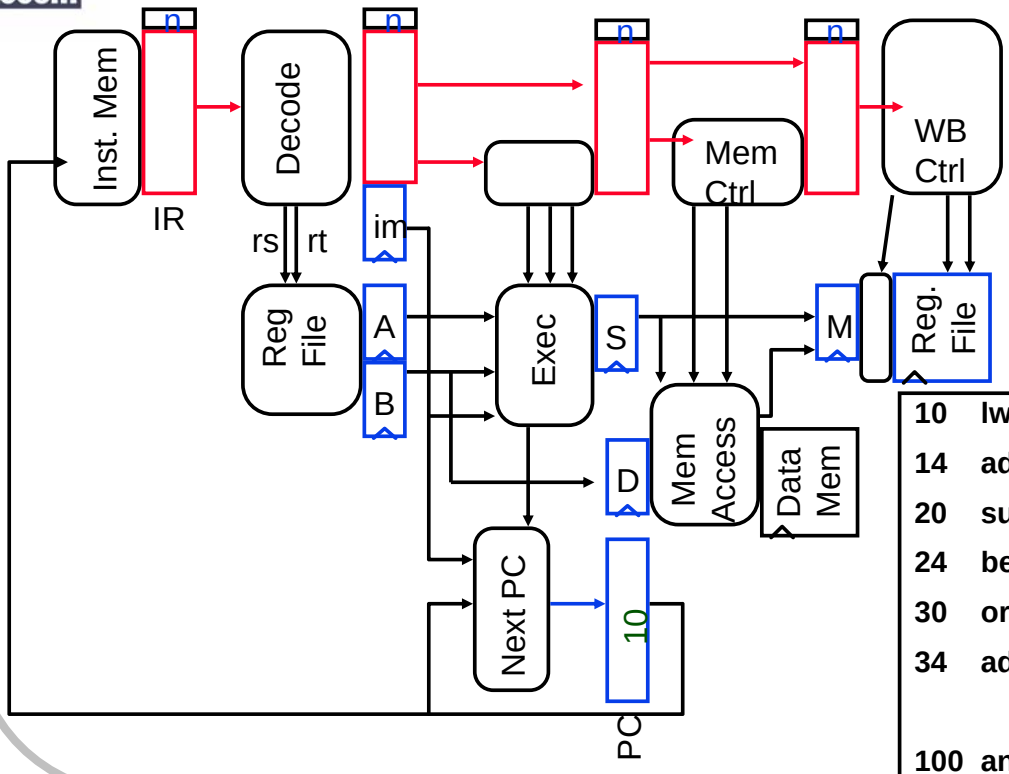


Let's Try it Out

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

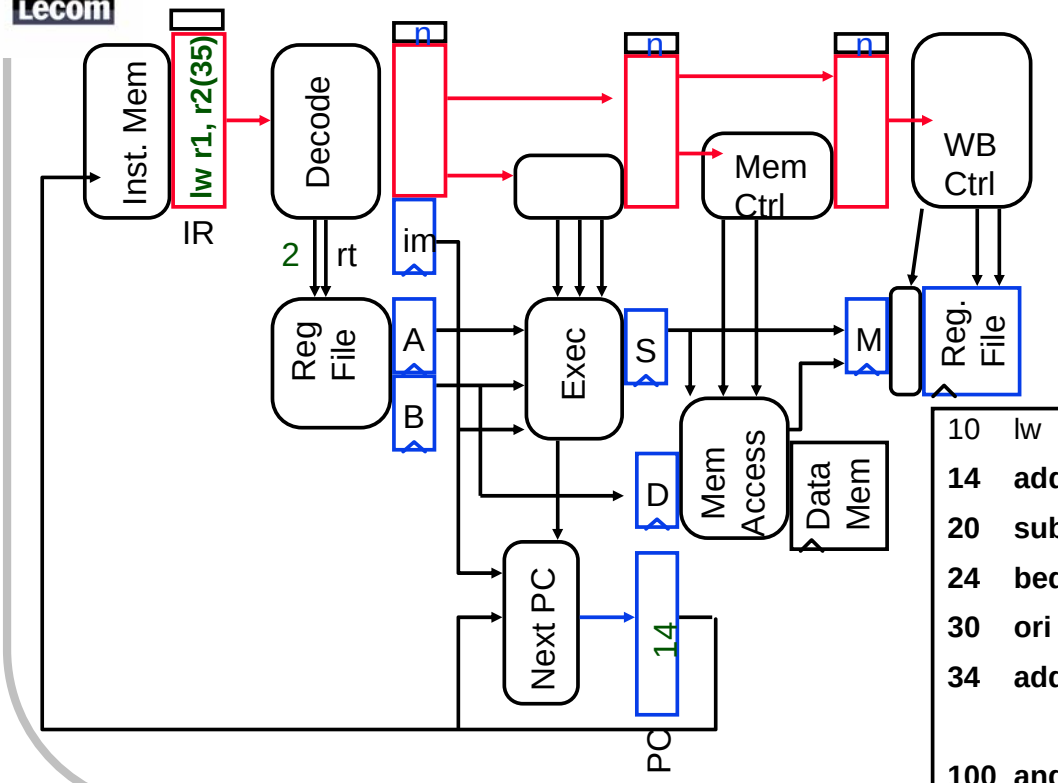
these addresses are octal

Start: Fetch 10



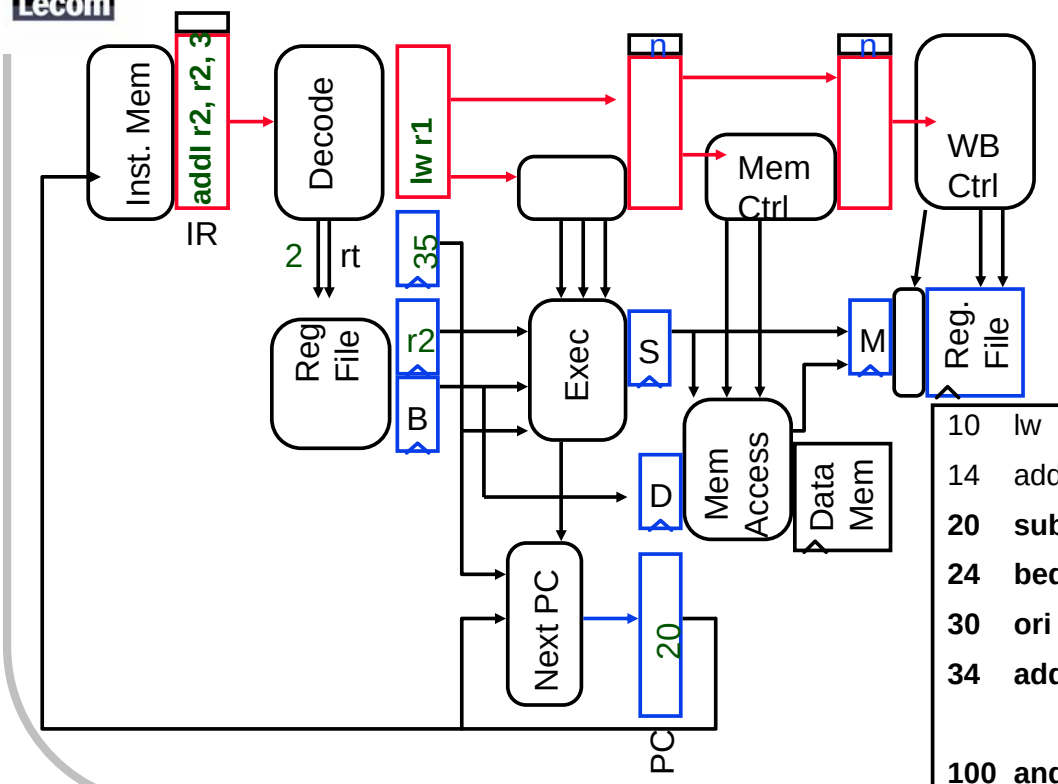
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Fetch 14, Decode 10



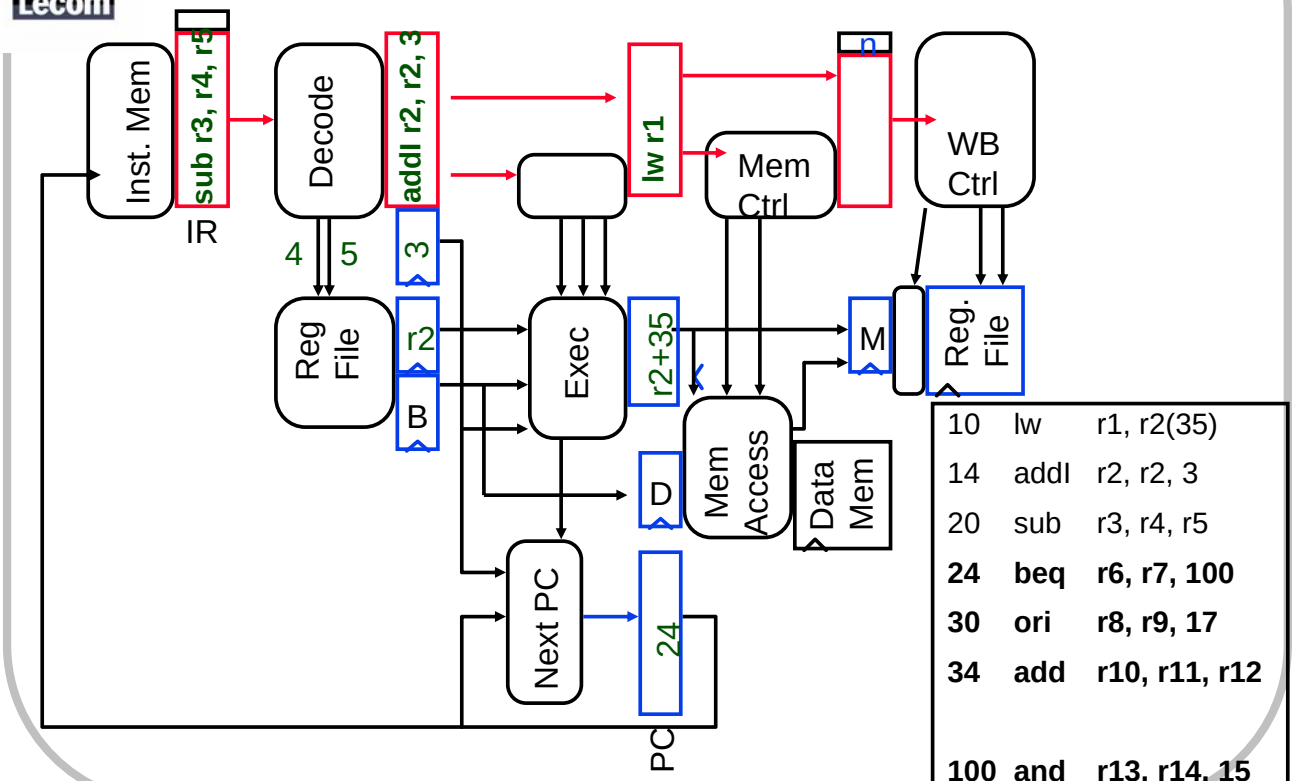
10	lw	r1, r2(35)
14	**addI**	**r2, r2, 3**
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 20, Decode 14, Exec 10

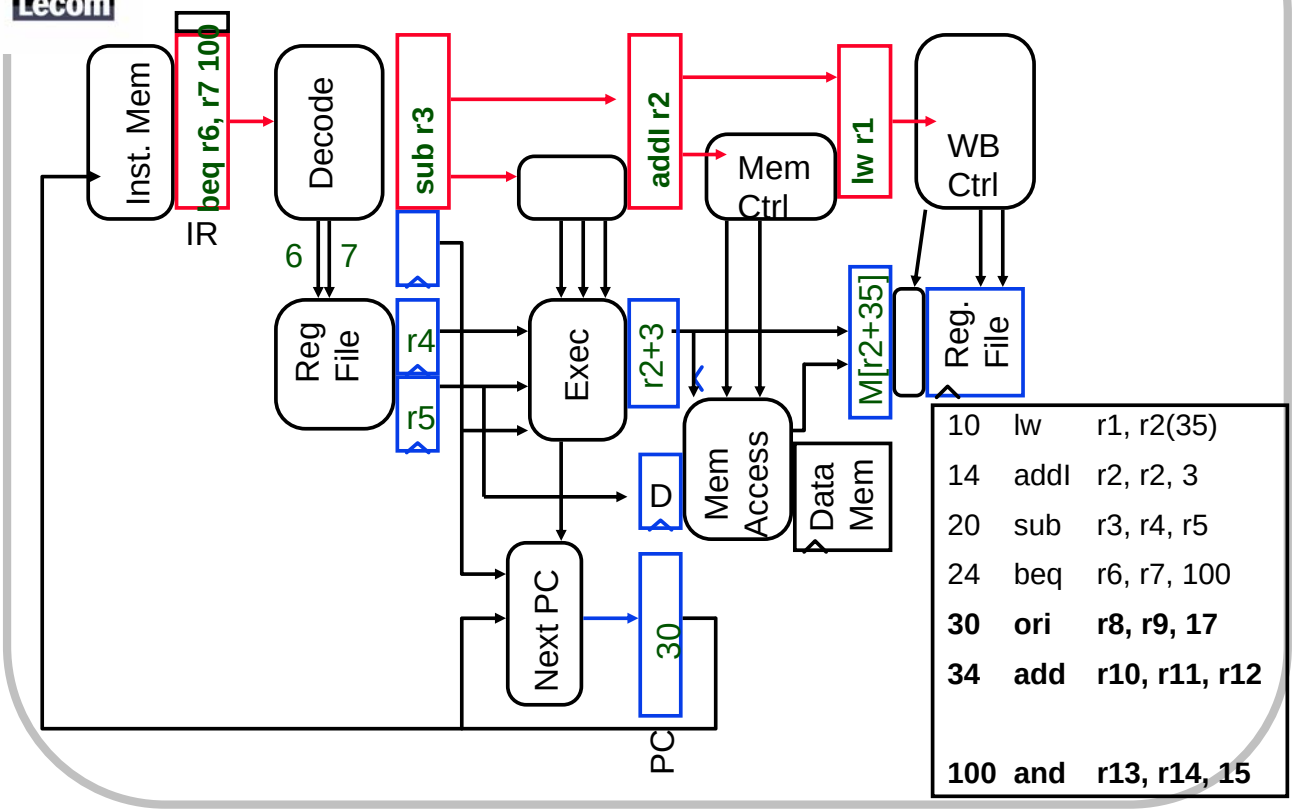


10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

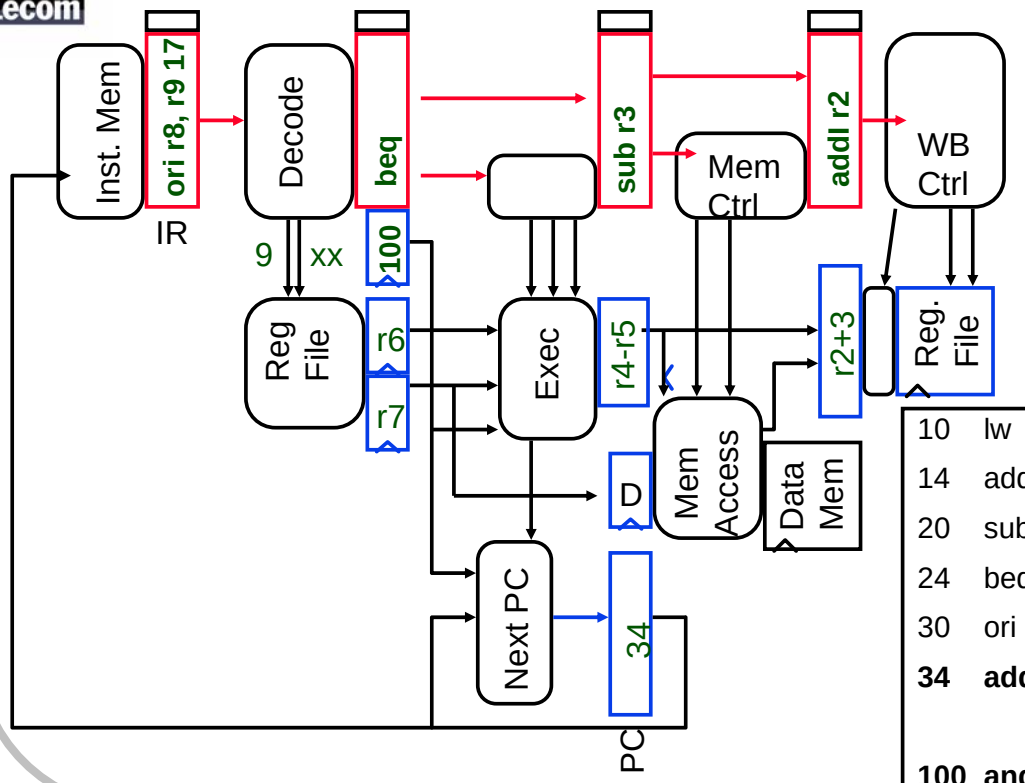
Fetch 24, Decode 20, Exec 14, Mem 10



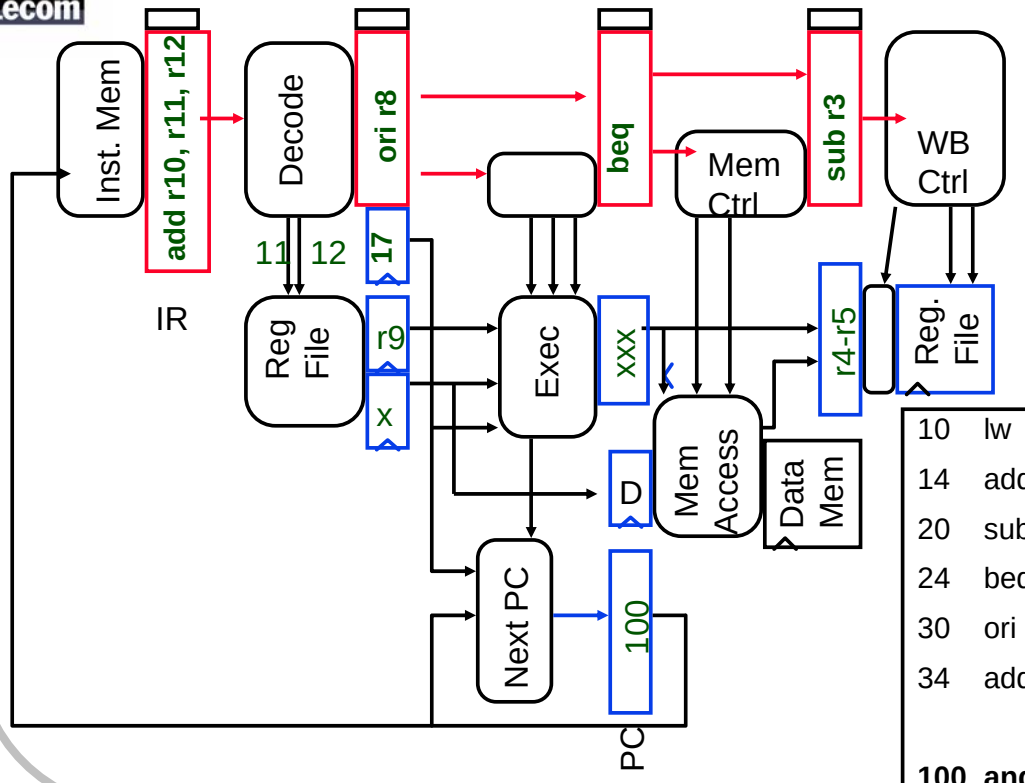
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14

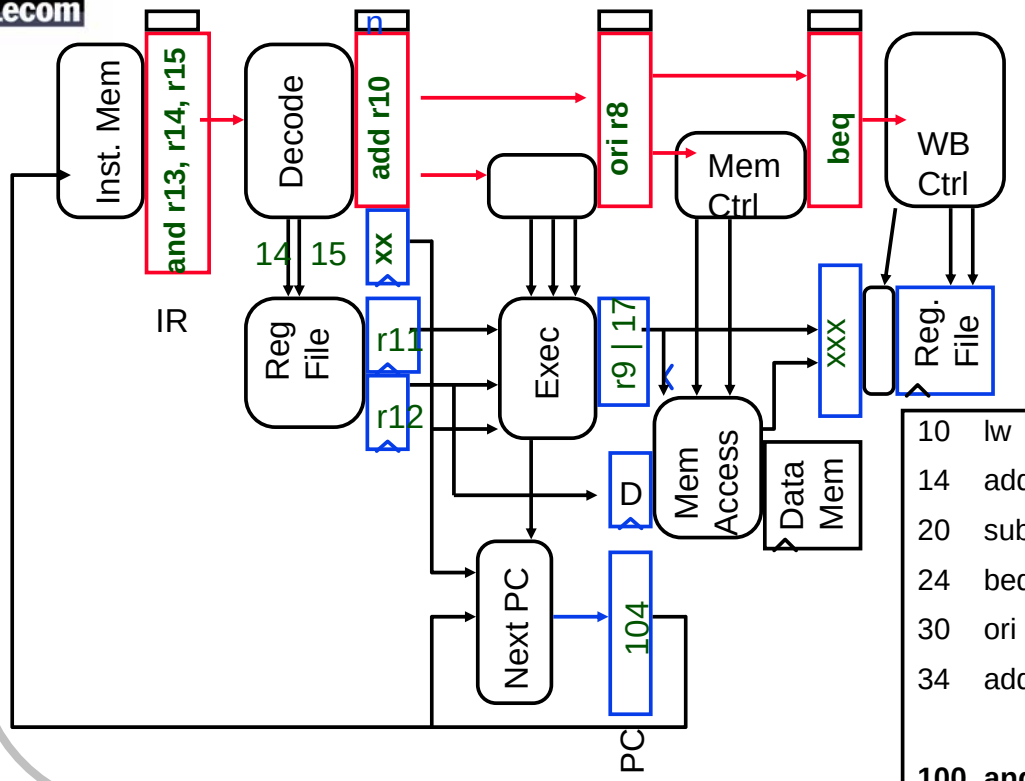


Fetch 100, Dcd 34, Ex 30, Mem 24, WB 20



ooops, we should have only one delayed instruction

Fetch 104, Dcd 100, Ex 34, Mem 30, WB 24

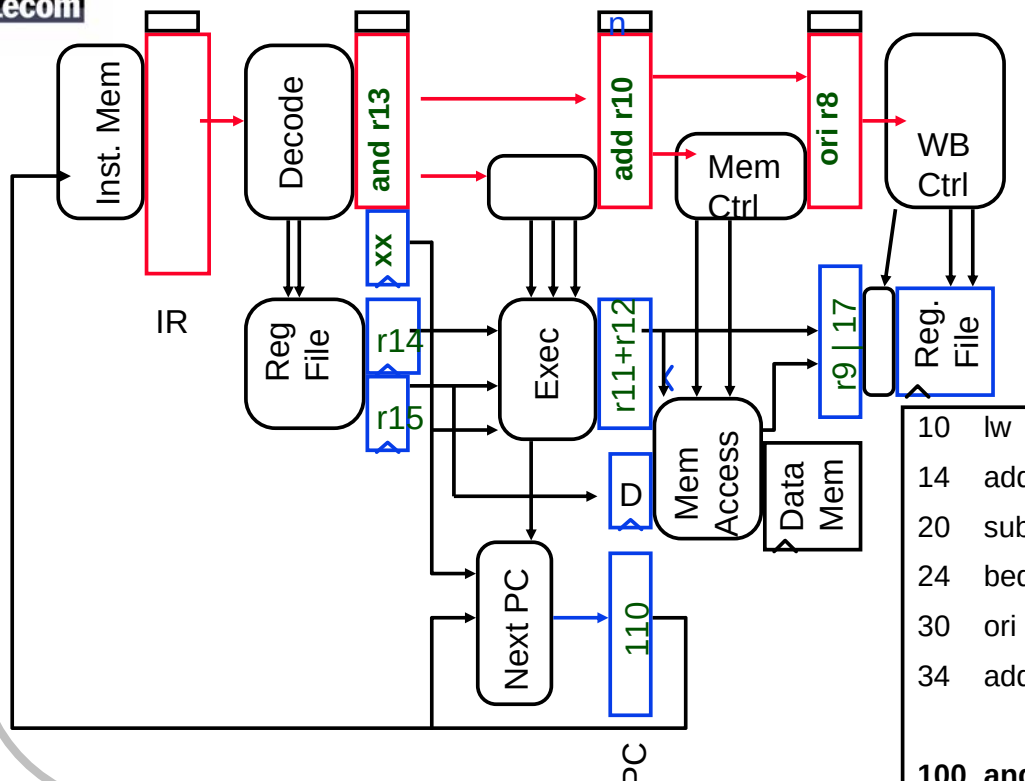


r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, 15**

Squash the extra instruction in the branch shadow!

Fetch 108, Dcd 104, Ex 100, Mem 34, WB 30

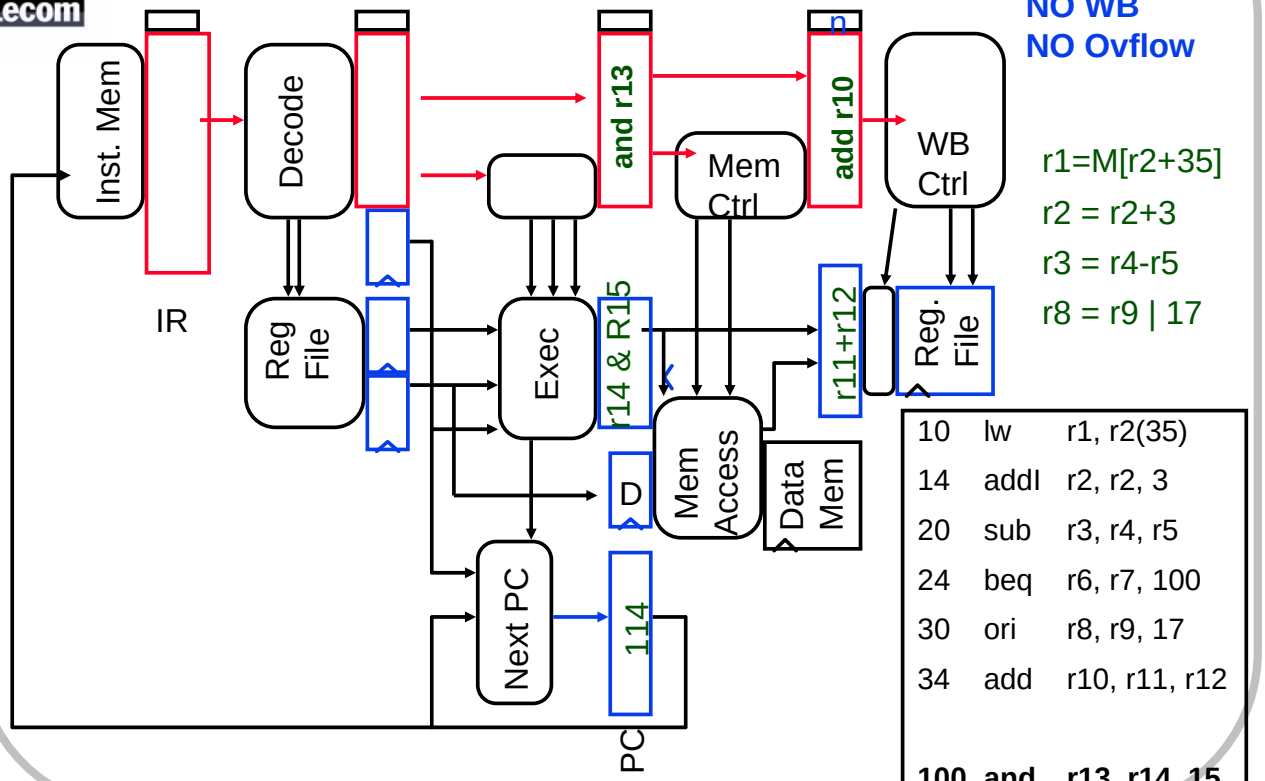


r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, 15**

Squash the extra instruction in the branch shadow!

Fetch 114, Dcd 110, Ex 104, Mem 100, WB 34



Squash the extra instruction in the branch shadow!

Summary: Pipelining

- **What makes it easy**
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- **What makes it hard?**
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- **We'll build a simple pipeline and look at these issues**
- **We'll talk about modern processors and what really makes it hard:**
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Summary

- **Pipelining is a fundamental concept**
  - **multiple steps using distinct resources**
- **Utilize capabilities of the Datapath by pipelined instruction processing**
  - **start next instruction while working on the current one**
  - **limited by length of longest stage (plus fill/flush)**
  - **detect and resolve hazards**