
$$E = \frac{mc^2}{\sqrt{1 - \frac{v^2}{c^2}}}$$

# GENEROVÁNÍ KÓDU

## 10. PARALELISMUS NA ÚROVNI INSTRUKCÍ (PIPELINING, MULTIPLE INSTRUCTION PARALLELISM)



2011 Jan Janoušek  
MI-GEN



Evropský sociální fond  
Praha & EU:  
Investujeme do vaší budoucnosti

# Instruction-level parallelism

- Obecně NP-úplná úloha, zrychluje kód o 30-50%
- Pipeline
  - Instrukce se zpracovává v několika fázích (stages)
    - fetch – read – execute – write
  - V jednom okamžiku se provádějí různé fáze různých instrukcí
  - Závislosti po sobě jdoucích instrukcí způsobují:
    - Pipeline stall – zdržení (i486)
    - Nekorektní provedení kódu (Sparc)
- Superskalární procesory (MIMD)
  - Několik výkonných jednotek (Pentium: 2)
  - V jednom okamžiku se provádějí stejné fáze několika instrukcí
- Vektorové procesory (SIMD)
  - Mnoho výkonných jednotek (Cray: 64)
    - Slabší varianty: Pentium MMX/SSE
  - V jednom okamžiku se provádí tatáž instrukce mnohokrát
  - Využití nepatří pod pojem scheduling
    - Automatická vektorizace používá podobné algoritmy jako některé metody schedulingu

# Motivation

We have seen optimisation techniques which involve removing and reordering code at both the source- and intermediate-language levels in an attempt to achieve the smallest and fastest correct program.

These techniques are platform-independent, and pay little attention to the details of the target architecture.

We can improve target code if we consider the architectural characteristics of the target processor.

# Single-cycle implementation

In *single-cycle* processor designs, an entire instruction is executed in a single clock cycle.

Each instruction will use some of the processor's *functional units*:

Instruction fetch (IF)	Register fetch (RF)	Execute (EX)	Memory access (MEM)	Register write-back (WB)
------------------------------	---------------------------	-----------------	---------------------------	--------------------------------

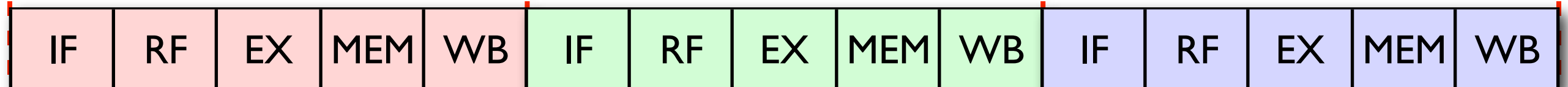
For example, a load instruction uses all five.

# Single-cycle implementation

`lw $1, 0($0)`

`lw $2, 4($0)`

`lw $3, 8($0)`



# Single-cycle implementation

On these processors, the order of instructions doesn't make any difference to execution time: each instruction takes one clock cycle, so  $n$  instructions will take  $n$  cycles and can be executed in any (correct) order.

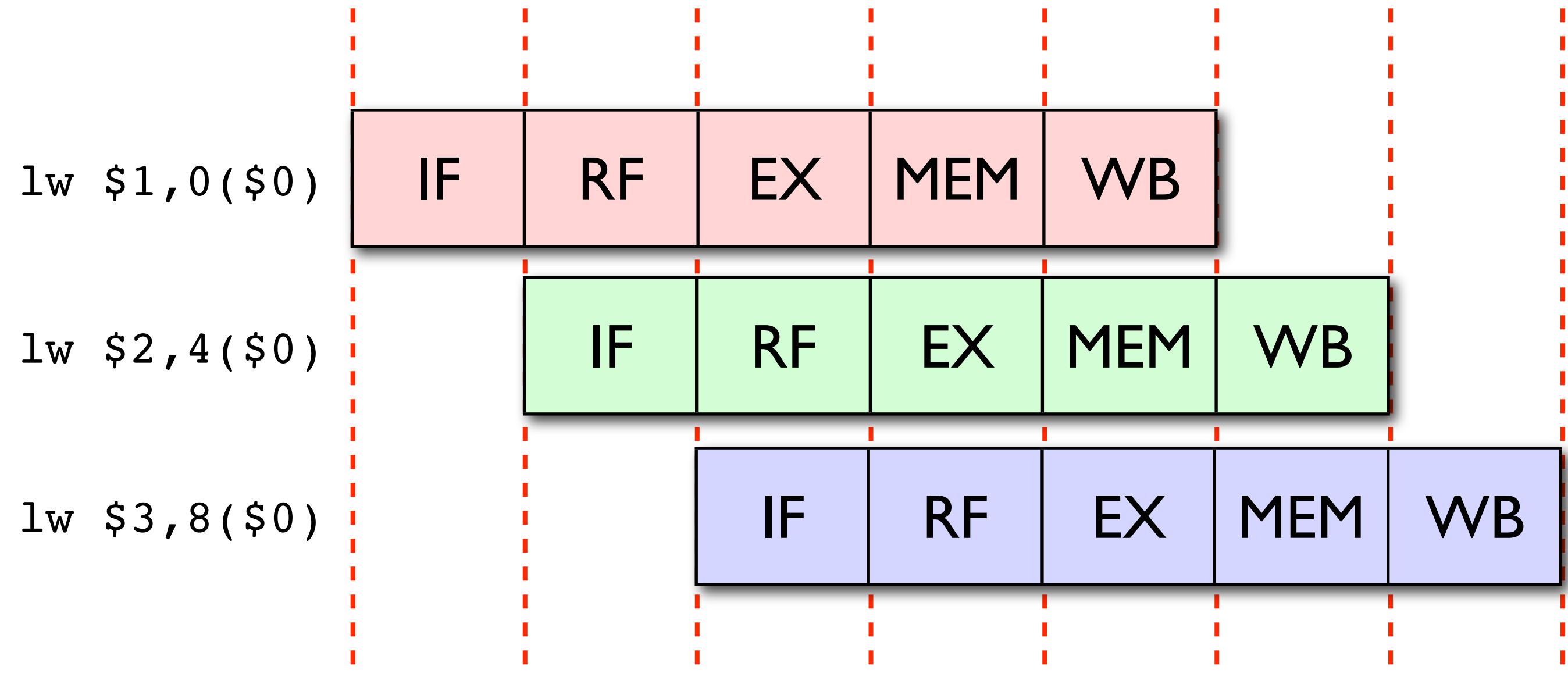
In this case we can naïvely translate our optimised 3-address code by expanding each intermediate instruction into the appropriate sequence of target instructions; clever reordering is unlikely to yield any benefits.

# Pipelined implementation

In *pipelined* processor designs (e.g. MIPS R2000), each functional unit works independently and does its job in a single clock cycle, so different functional units can be handling different instructions simultaneously.

These functional units are arranged in a pipeline, and the result from each unit is passed to the next one via a pipeline register before the next clock cycle.

# Pipelined implementation





# Pipelined implementation

In this *multicycle* design the clock cycle is much shorter (one functional unit vs. one complete instruction) and ideally we can still execute one instruction per cycle when the pipeline is full.

Programs will therefore execute more quickly.

# Pipeline hazards

However, it is not always possible to run the pipeline at full capacity.

Some situations prevent the next instruction from executing in the next clock cycle: this is a *pipeline hazard*.

On interlocked hardware (e.g. SPARC) a hazard will cause a *pipeline stall*; on non-interlocked hardware (e.g. MIPS) the compiler must generate explicit NOPs to avoid errors.

# Pipeline hazards

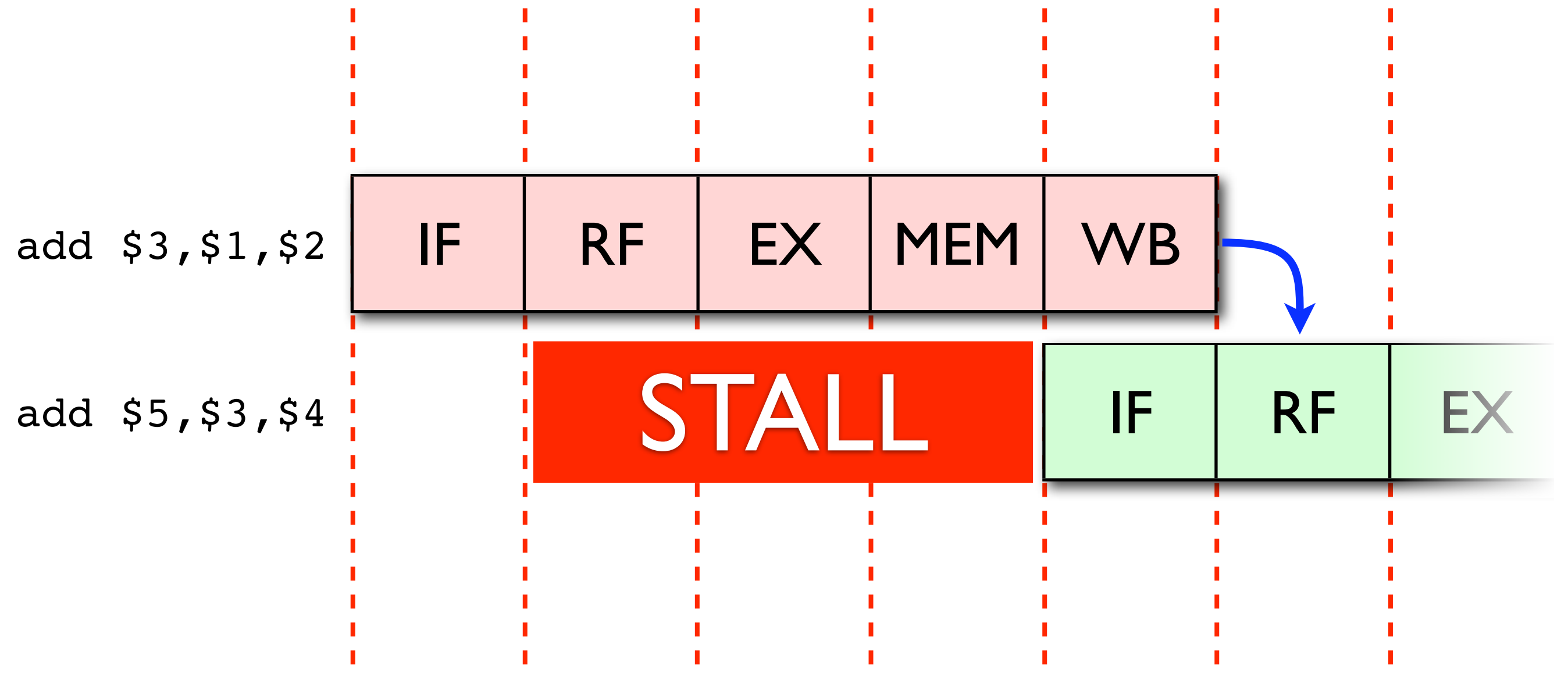
Consider *data hazards*: these occur when an instruction depends upon the result of an earlier one.

add \$3, \$1, \$2

add \$5, \$3, \$4

The pipeline must stall until the result of the first add has been written back into register \$3.

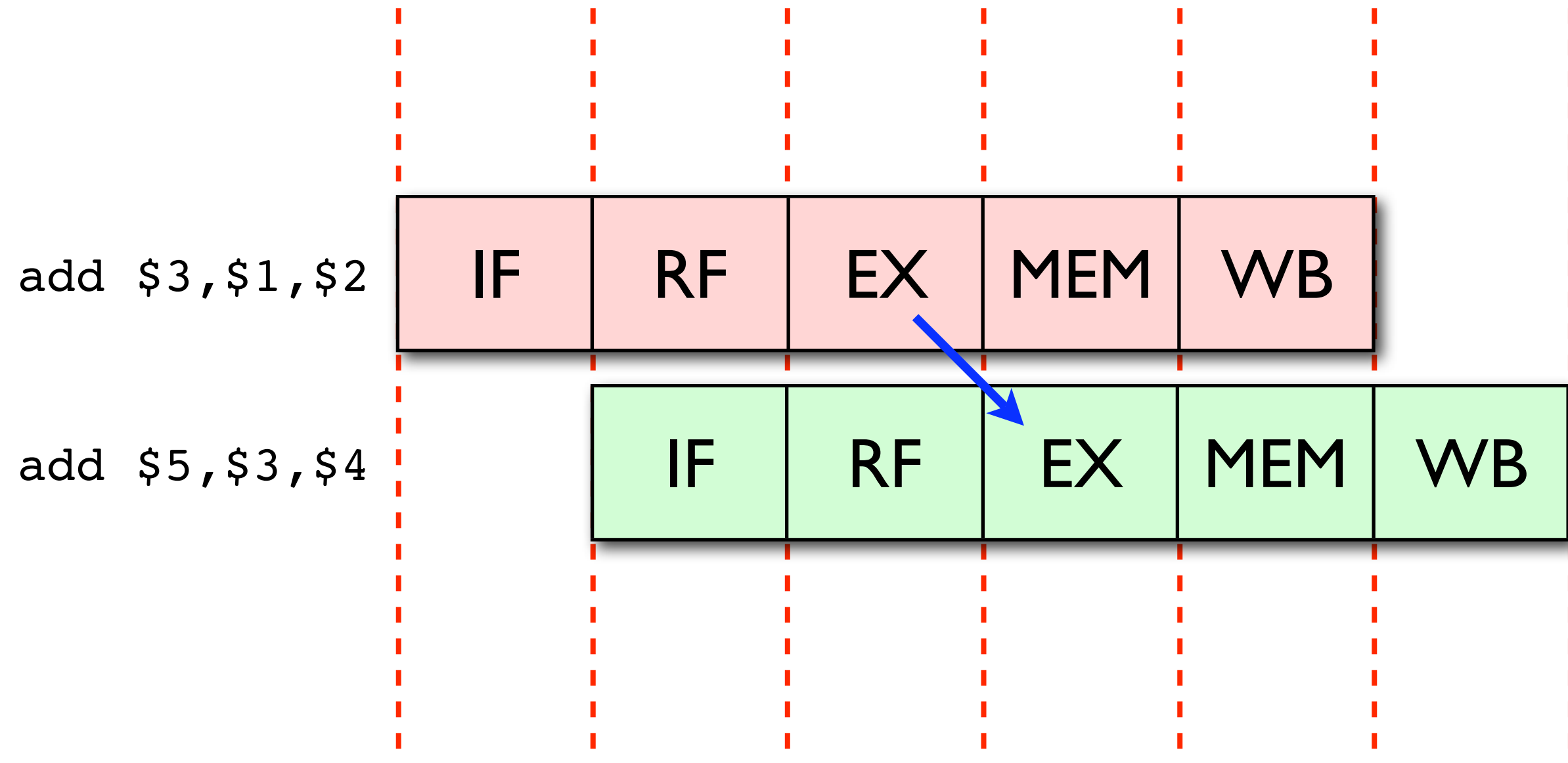
# Pipeline hazards



# Pipeline hazards

The severity of this effect can be reduced by using *feed-forwarding*: extra paths are added between functional units, allowing data to be used before it has been written back into registers.

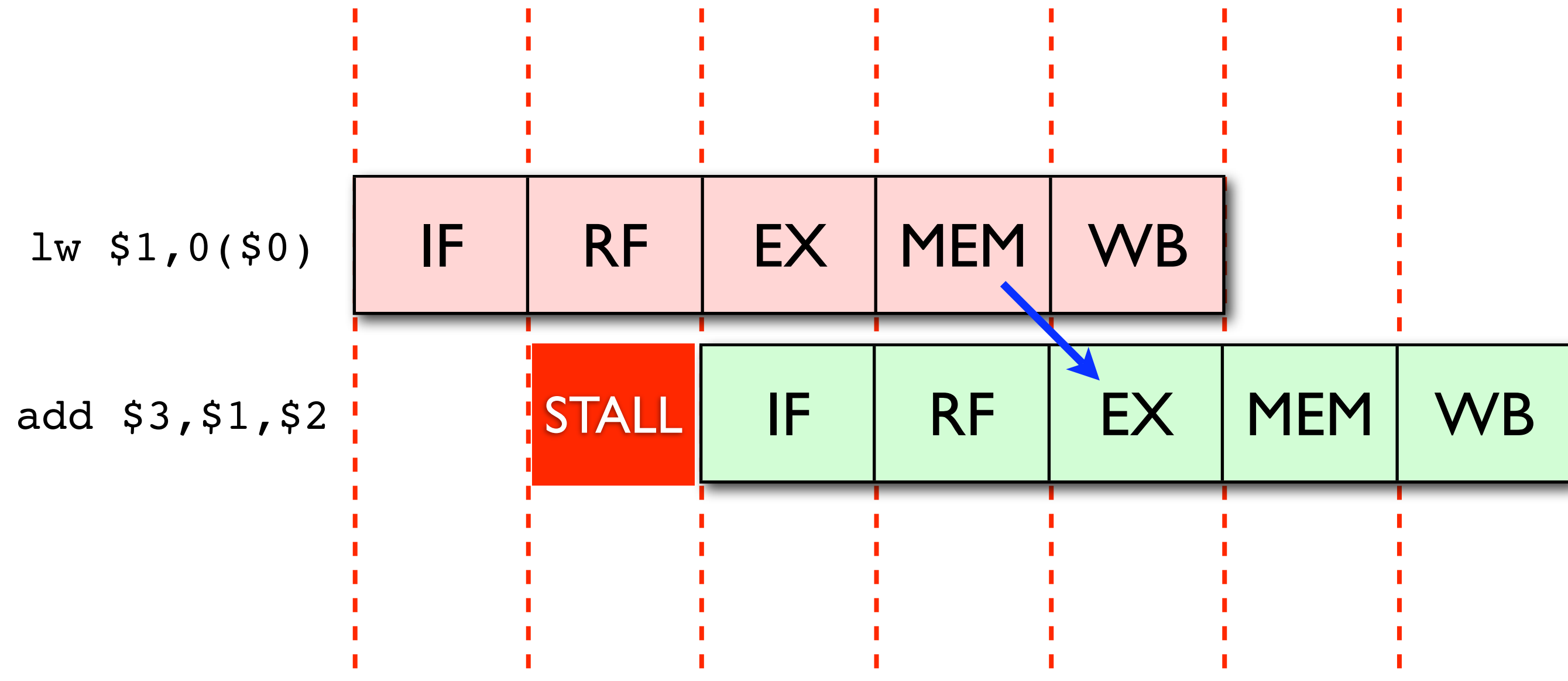
# Pipeline hazards



# Pipeline hazards

But even when feed-forwarding is used,  
some combinations of instructions will  
always result in a stall.

# Pipeline hazards



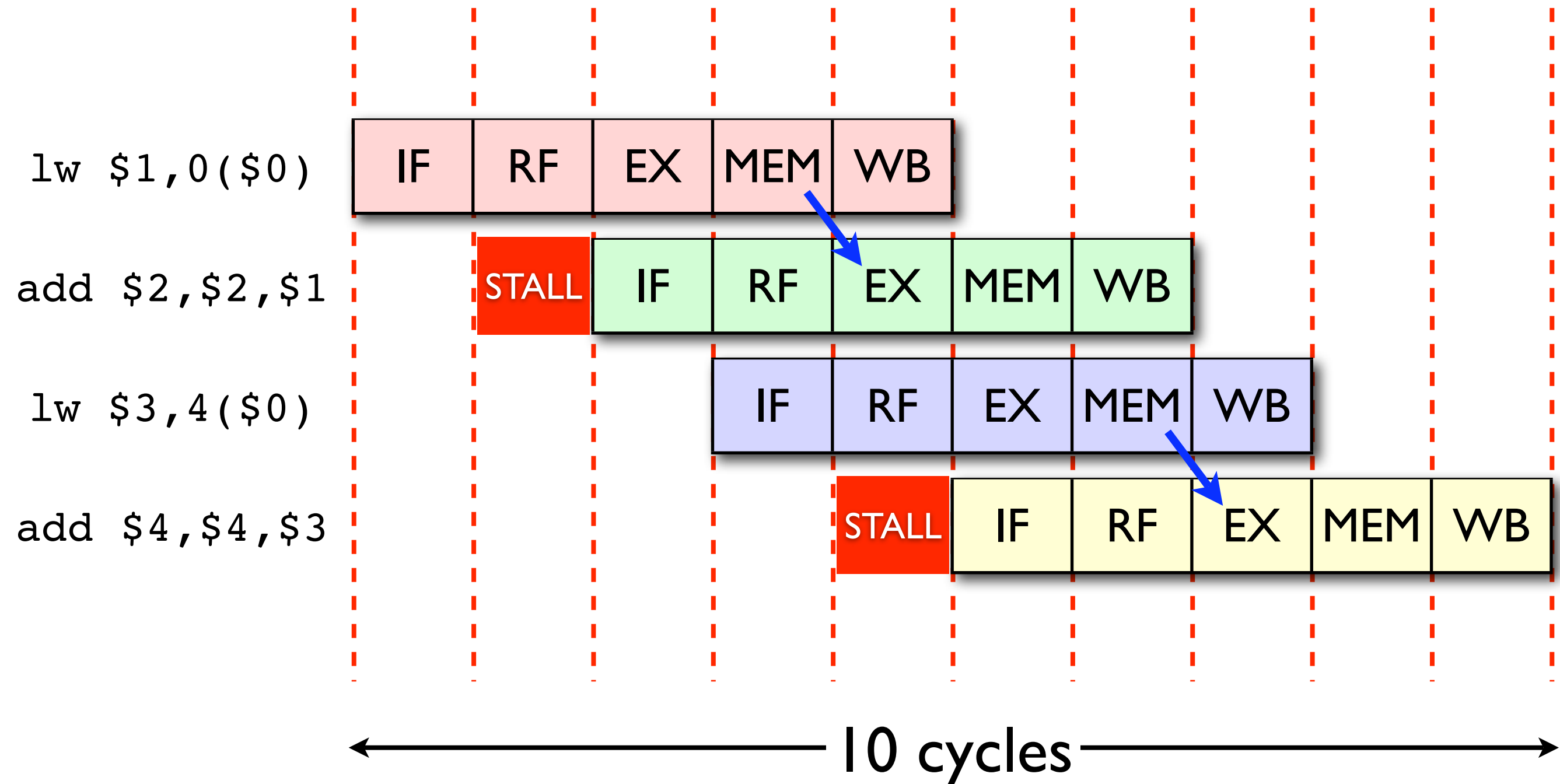


# Instruction order

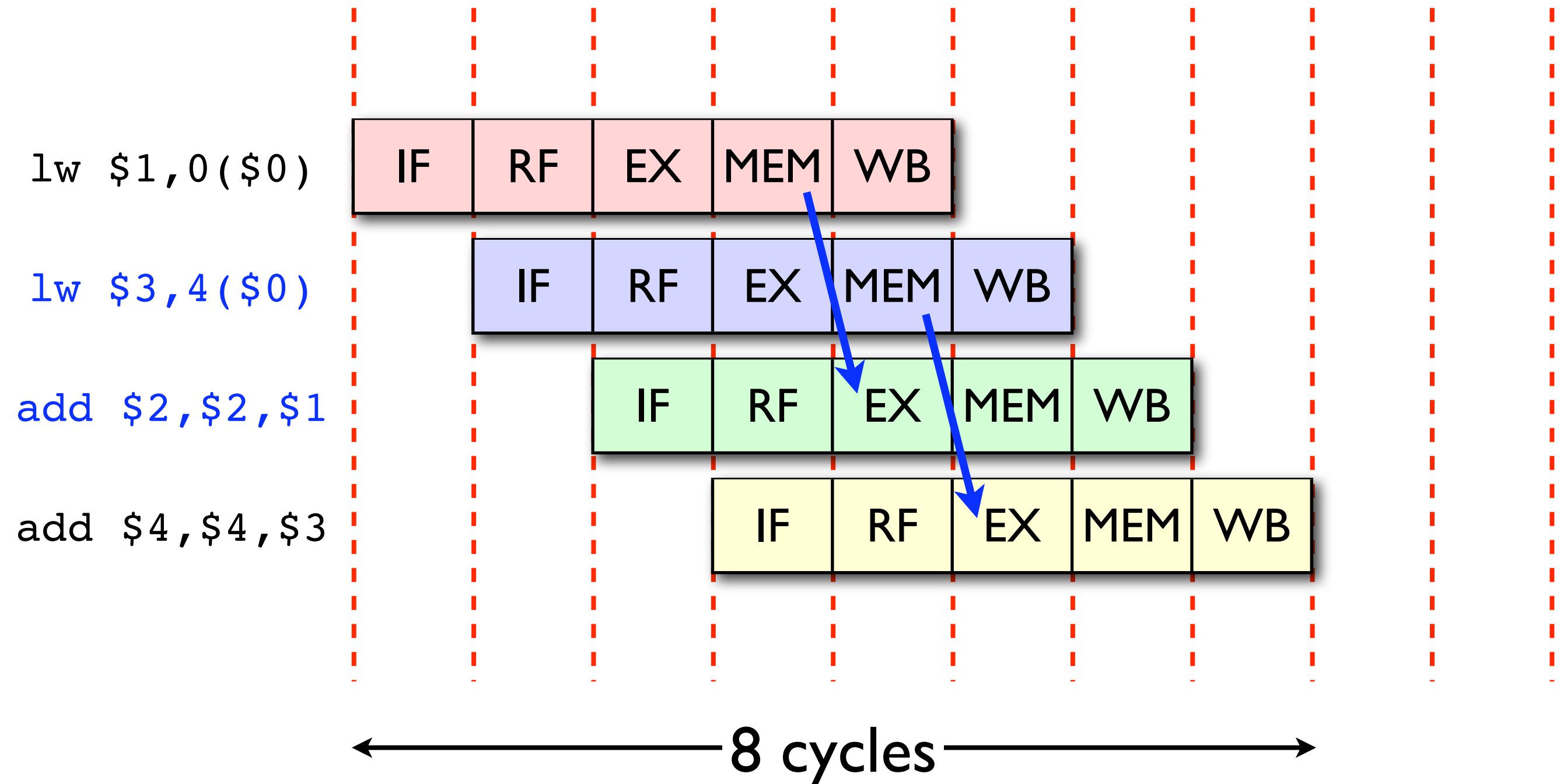
Since particular combinations of instructions cause this problem on pipelined architectures, we can achieve better performance by reordering instructions where possible.

```
lw  $1, 0($0)
add $2, $2, $1
lw  $3, 4($0)
add $4, $4, $3
```

# Instruction order



# Instruction order



# Instruction dependencies

We can only reorder target-code instructions if the meaning of the program is preserved.

We must therefore identify and respect the *data dependencies* which exist between instructions.

In particular, whenever an instruction is dependent upon an earlier one, the order of these two instructions must not be reversed.

# Instruction dependencies

There are three kinds of data dependency:

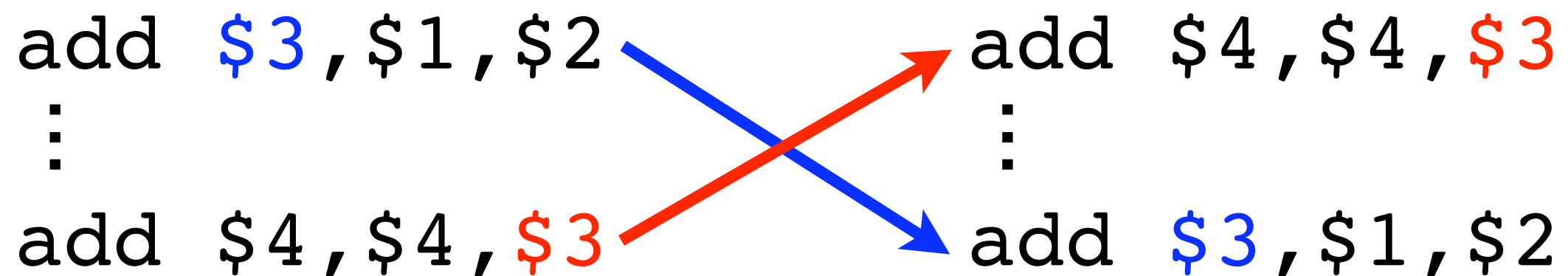
- Read after write
- Write after read
- Write after write

Whenever one of these dependencies exists between two instructions, we cannot safely permute them.

# Instruction dependencies

Read after write:

An instruction **reads** from a location  
after an earlier instruction has **written** to it.



Reads old value

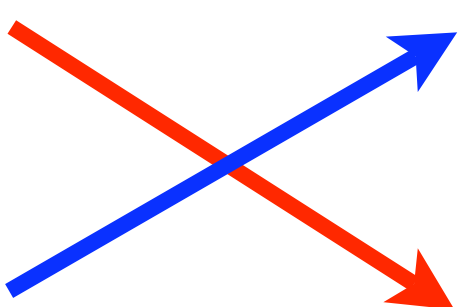


# Instruction dependencies

Write after read:

An instruction **writes** to a location  
after an earlier instruction has **read** from it.

add \$4, \$4, \$3		add \$3, \$1, \$2
⋮		⋮
add \$3, \$1, \$2		add \$4, \$4, \$3



Reads new value

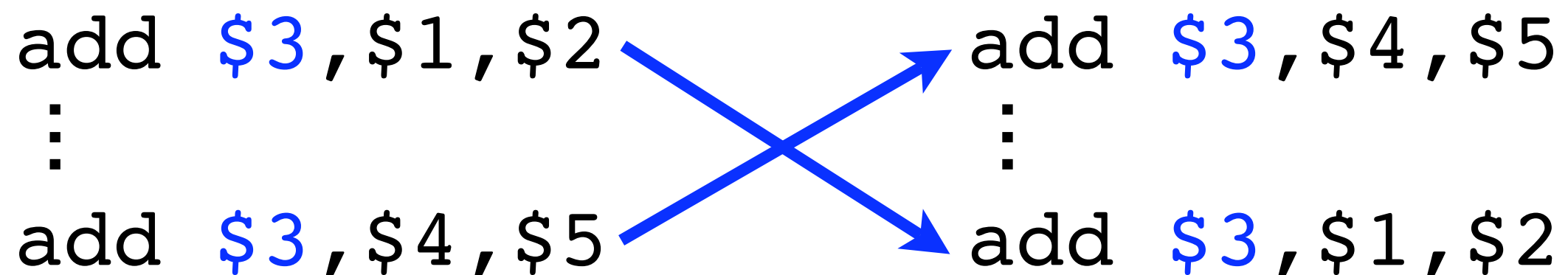


# Instruction dependencies

Write after write:

An instruction **writes** to a location  
after an earlier instruction has **written** to it.

add \$3, \$1, \$2		add \$3, \$4, \$5
⋮		⋮
add \$3, \$4, \$5		add \$3, \$1, \$2



Writes old value





# Instruction scheduling

We would like to reorder the instructions within each basic block in a way which

- preserves the dependencies between those instructions (and hence the correctness of the program), and
- achieves the minimum possible number of pipeline stalls.

We can address these two goals separately.

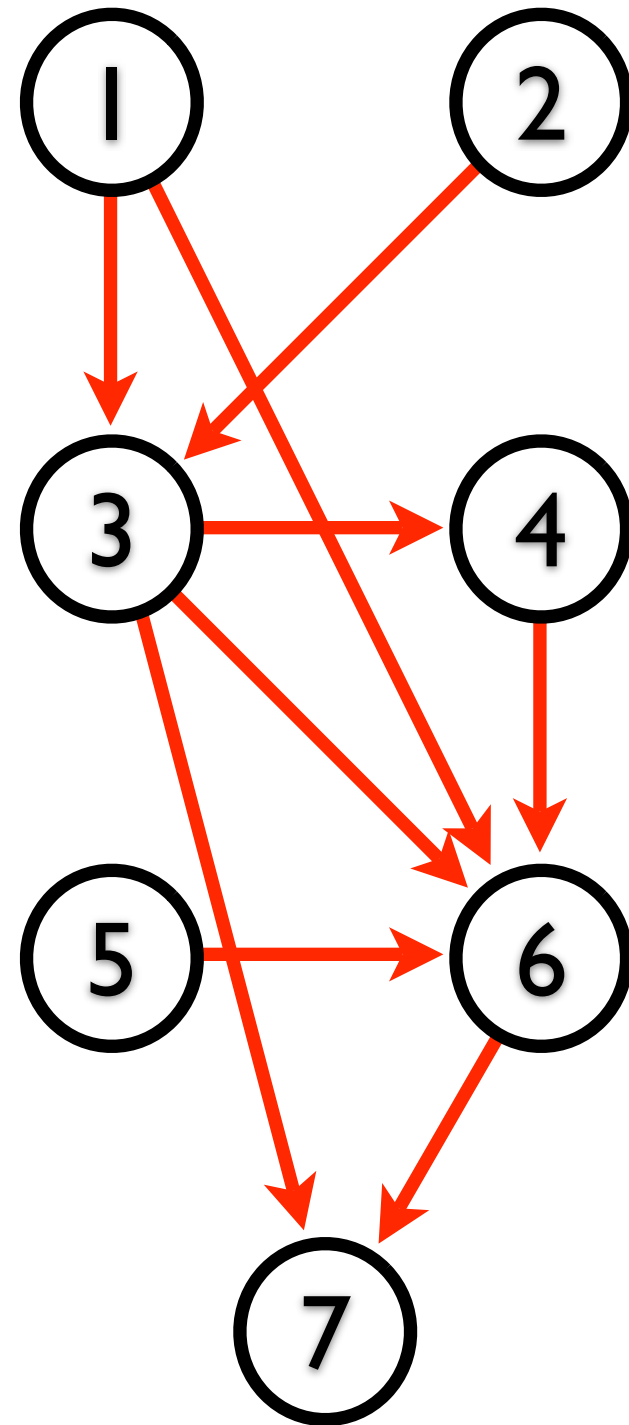
# Preserving dependencies

Firstly, we can construct a *directed acyclic graph* (DAG) to represent the dependencies between instructions:

- For each instruction in the basic block, create a corresponding vertex in the graph.
- For each dependency between two instructions, create a corresponding edge in the graph.
  - ▶ This edge is *directed*: it goes from the earlier instruction to the later one.

# Preserving dependencies

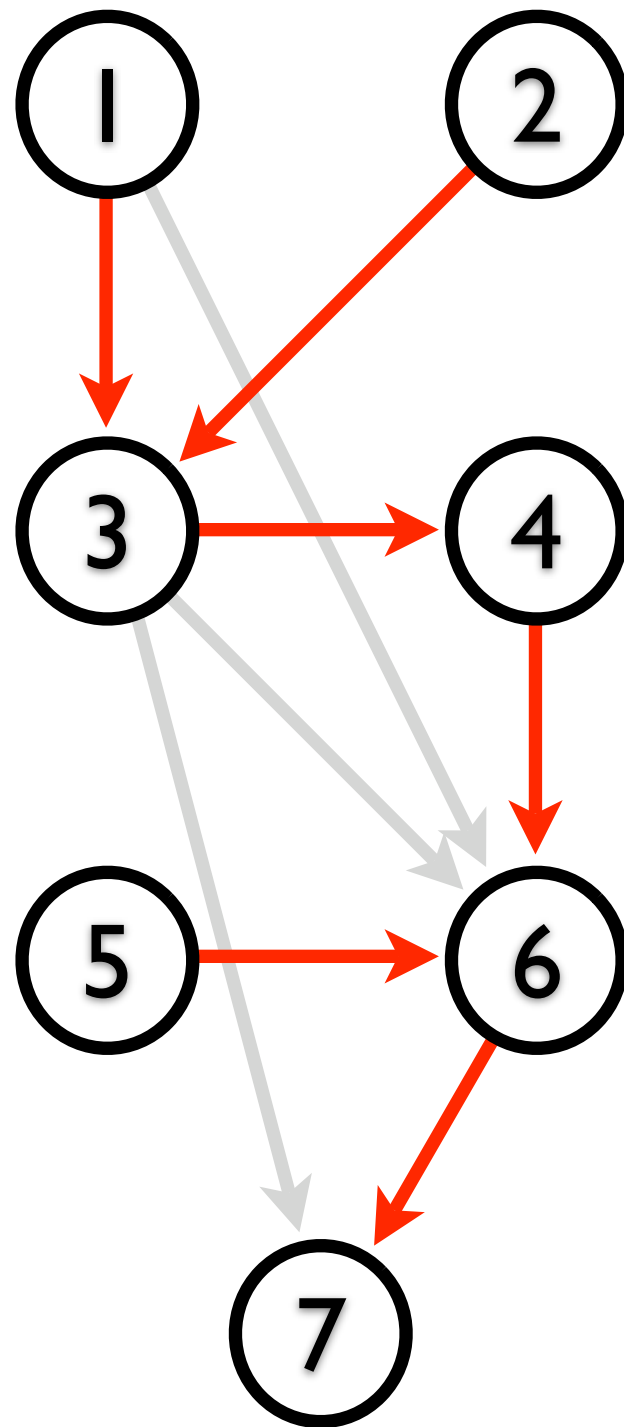
```
1 lw $1, 0($0)
2 lw $2, 4($0)
3 add $3, $1, $2
4 sw $3, 12($0)
5 lw $4, 8($0)
6 add $3, $1, $4
7 sw $3, 16($0)
```



# Preserving dependencies

Any topological sort of this DAG (i.e. any linear ordering of the vertices which keeps all the edges “pointing forwards”) will maintain the dependencies and hence preserve the correctness of the program.

# Preserving dependencies



1, 2, 3, 4, 5, 6, 7

2, 1, 3, 4, 5, 6, 7

1, 2, 3, 5, 4, 6, 7

1, 2, 5, 3, 4, 6, 7

1, 5, 2, 3, 4, 6, 7

5, 1, 2, 3, 4, 6, 7

2, 1, 3, 5, 4, 6, 7

2, 1, 5, 3, 4, 6, 7

2, 5, 1, 3, 4, 6, 7

5, 2, 1, 3, 4, 6, 7

# Minimising stalls

Secondly, we want to choose an instruction order which causes the fewest possible pipeline stalls.

Unfortunately, this problem is (as usual) NP-complete and hence difficult to solve in a reasonable amount of time for realistic quantities of instructions.

However, we can devise some *static scheduling heuristics* to help guide us; we will hence choose a sensible and reasonably optimal instruction order, if not necessarily the absolute best one possible.

# Minimising stalls

Each time we're emitting the next instruction, we should try to choose one which:

- does not conflict with the previous emitted instruction
- is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)
- is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

# Algorithm

Armed with the scheduling DAG and the static scheduling heuristics, we can now devise an algorithm to perform instruction scheduling.



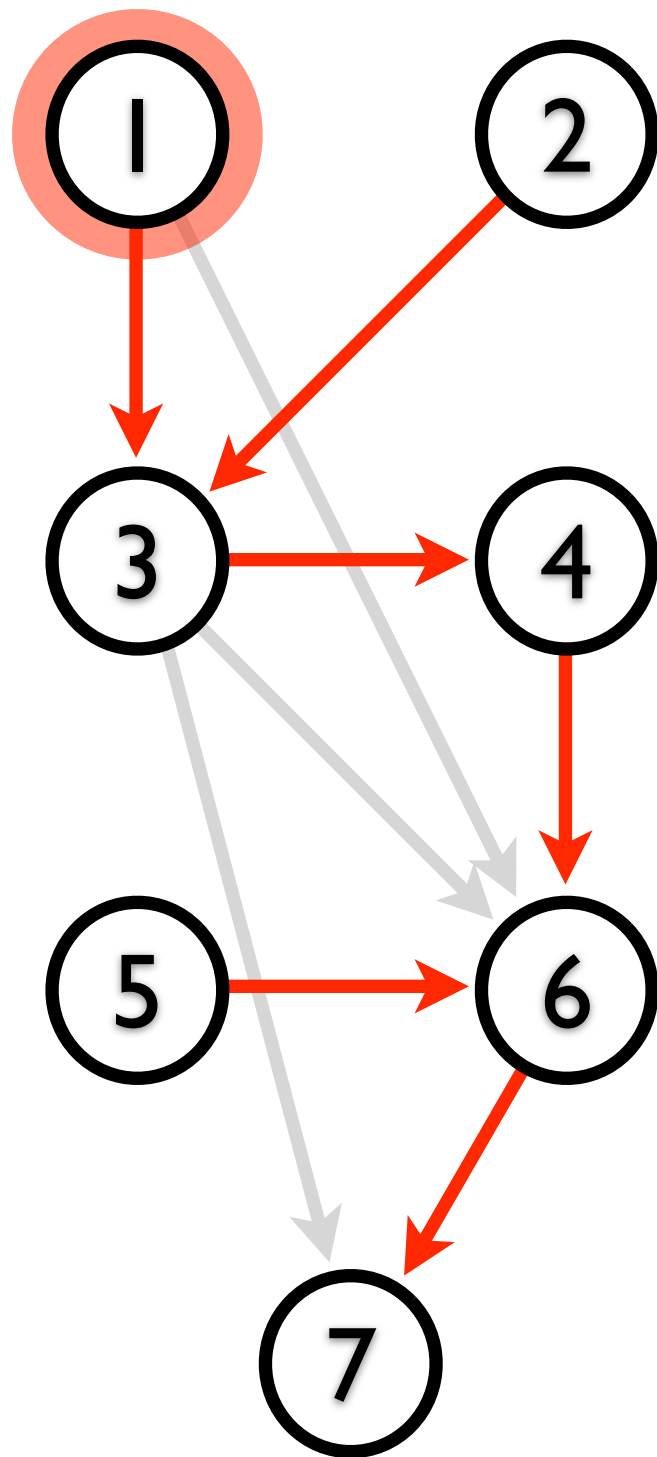
# Algorithm

- Construct the scheduling DAG.
  - ▶ We can do this in  $O(n^2)$  by scanning backwards through the basic block and adding edges as dependencies arise.
- Initialise the *candidate list* to contain the minimal elements of the DAG.

# Algorithm

- While the candidate list is non-empty:
  - If possible, emit a candidate instruction satisfying all three of the static scheduling heuristics;
  - if no instruction satisfies all the heuristics, either emit NOP (on MIPS) or an instruction satisfying only the last two heuristics (on SPARC).
  - Remove the instruction from the DAG and insert the newly minimal elements into the candidate list.

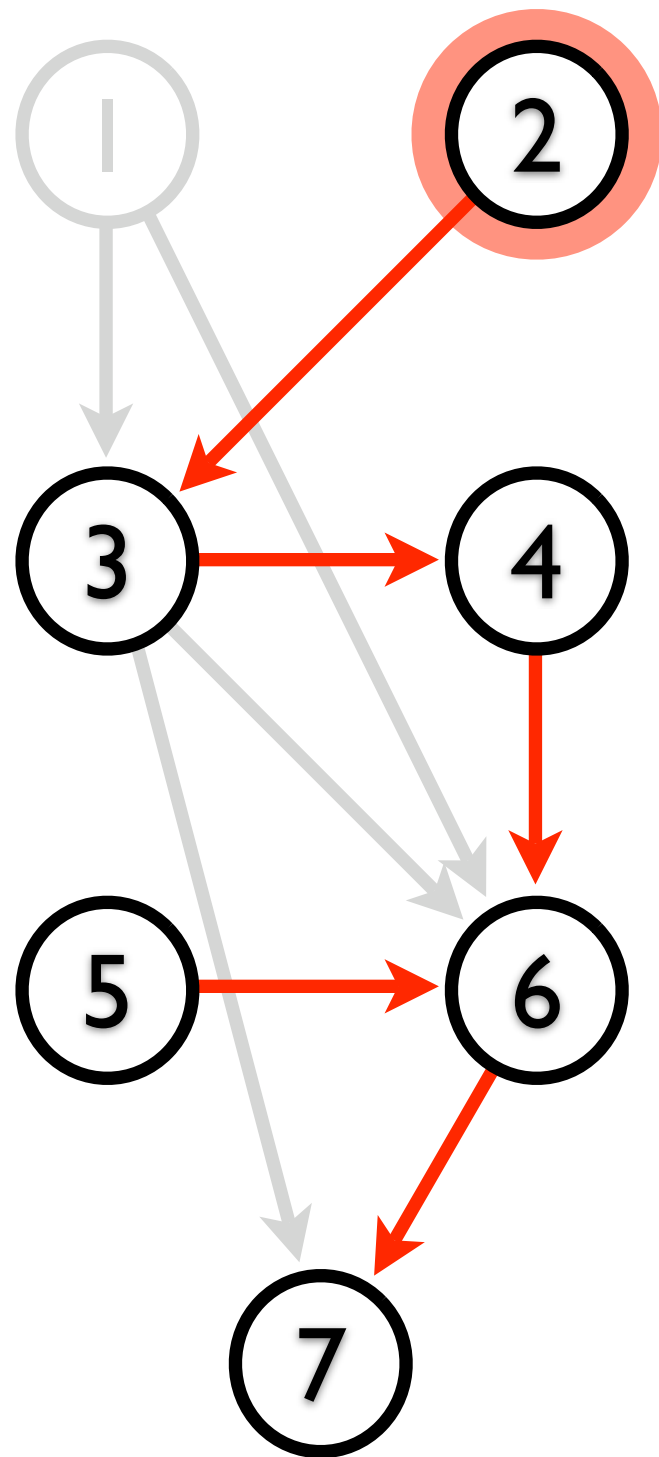
# Algorithm



Candidates:  
 $\{ 1, 2, 5 \}$

| lw \$1, 0 (\$0)

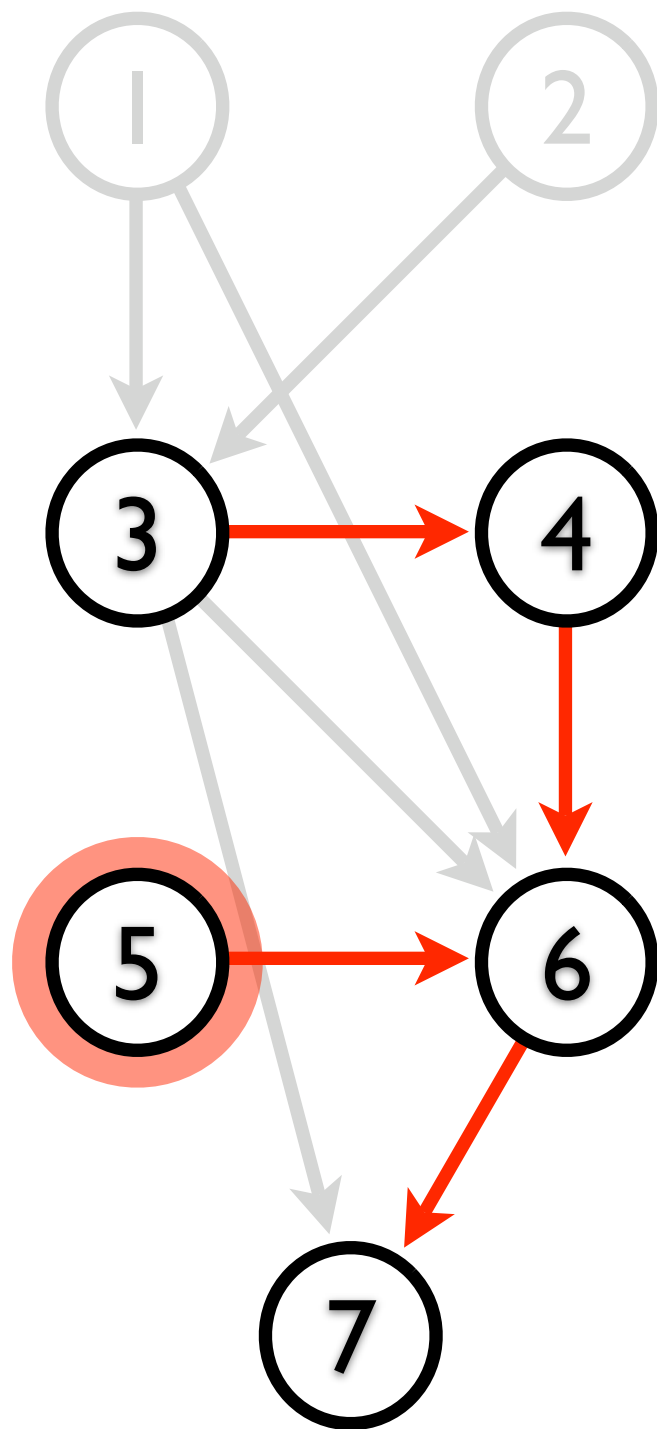
# Algorithm



Candidates:  
 $\{ 2, 5 \}$

<b>1</b>	1w	\$1, 0 ( \$0 )
<b>2</b>	1w	\$2, 4 ( \$0 )

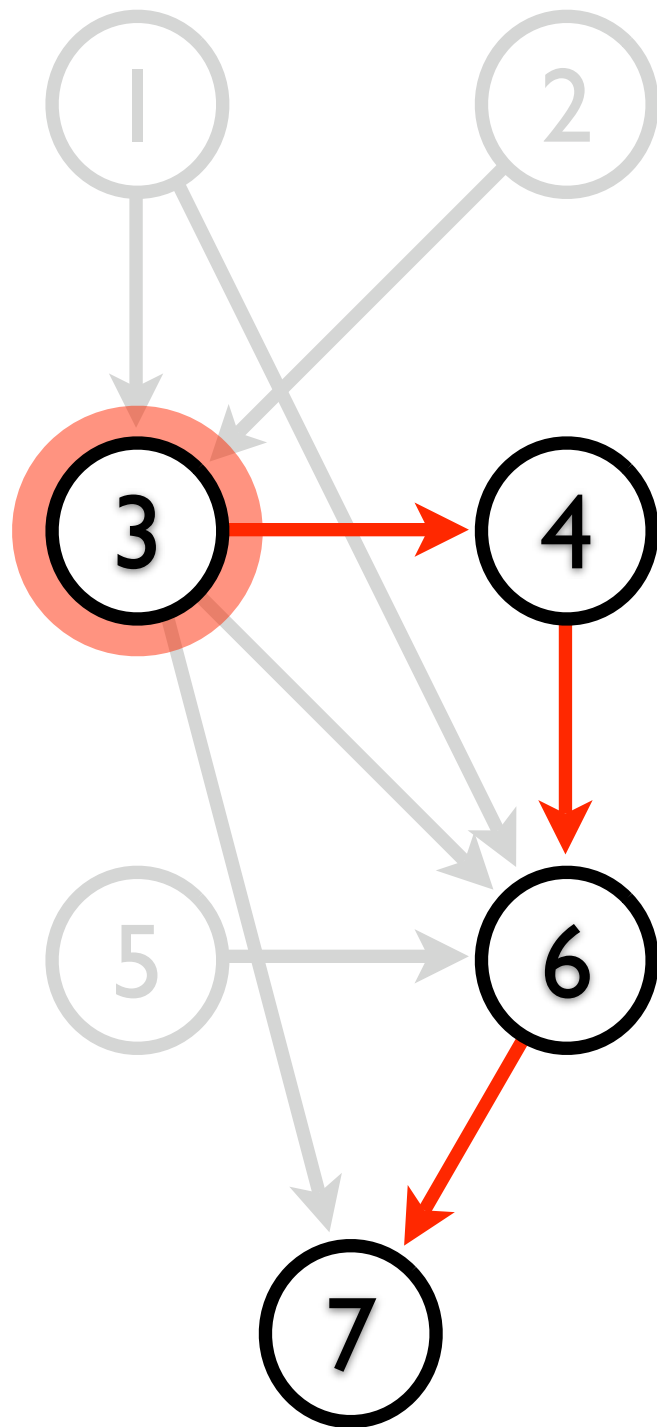
# Algorithm



Candidates:  
 $\{ 3, 5 \}$

1	1w	\$1, 0 (\$0)
2	1w	\$2, 4 (\$0)
5	1w	\$4, 8 (\$0)

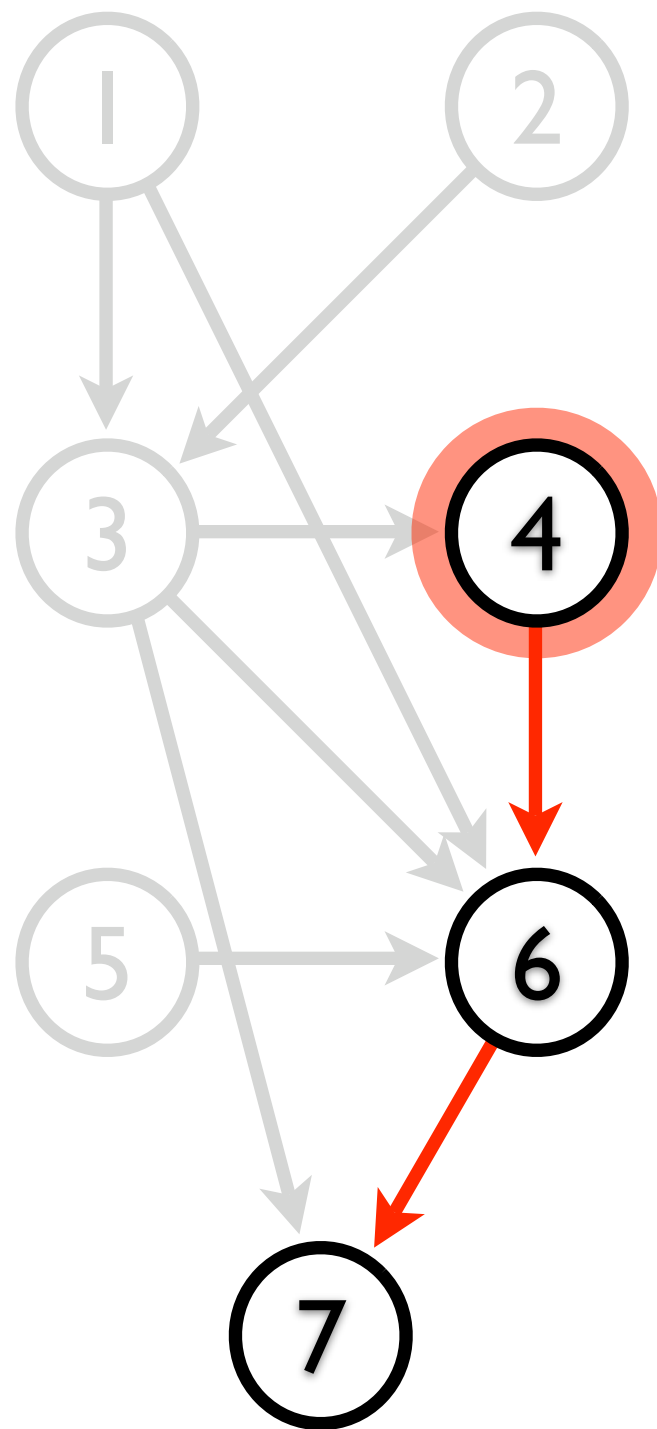
# Algorithm



Candidates:  
 $\{ 3 \}$

1	lw	\$1, 0 (\$0)
2	lw	\$2, 4 (\$0)
5	lw	\$4, 8 (\$0)
3	add	\$3, \$1, \$2

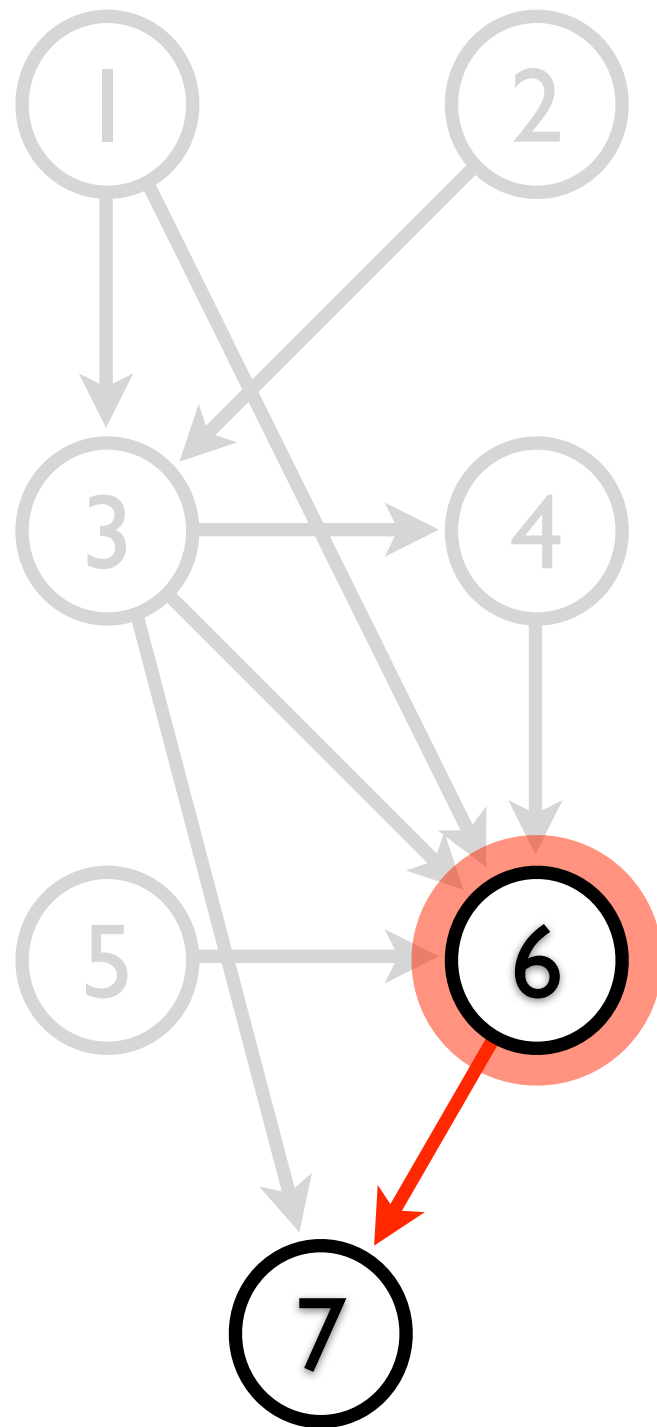
# Algorithm



Candidates:  
 $\{ 4 \}$

1	lw	\$1, 0 (\$0)
2	lw	\$2, 4 (\$0)
5	lw	\$4, 8 (\$0)
3	add	\$3, \$1, \$2
4	sw	\$3, 12 (\$0)

# Algorithm

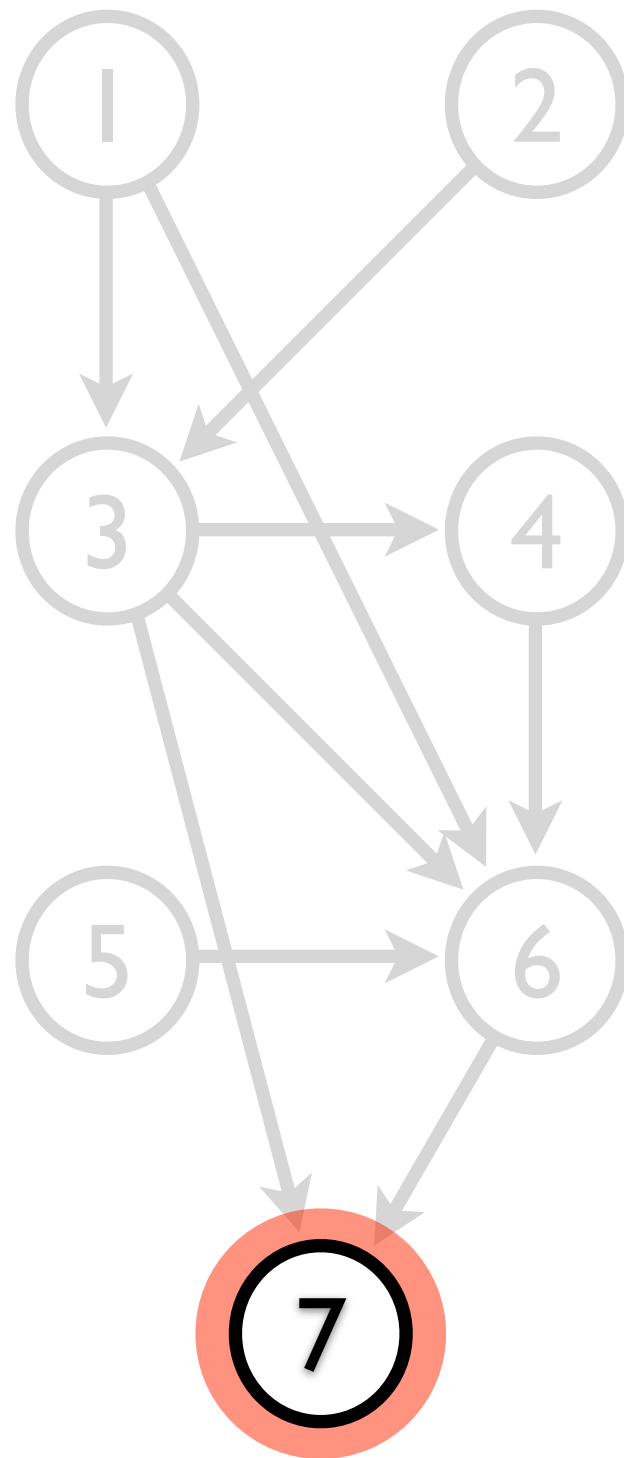


Candidates:  
{ 6 }

1	lw	\$1, 0 (\$0)
2	lw	\$2, 4 (\$0)
5	lw	\$4, 8 (\$0)
3	add	\$3, \$1, \$2
4	sw	\$3, 12 (\$0)
6	add	\$3, \$1, \$4



# Algorithm



Candidates:  
 $\{ 7 \}$

**1** lw \$1, 0 (\$0)  
**2** lw \$2, 4 (\$0)  
**5** lw \$4, 8 (\$0)  
**3** add \$3, \$1, \$2  
**4** sw \$3, 12 (\$0)  
**6** add \$3, \$1, \$4  
**7** sw \$3, 16 (\$0)

# Algorithm

Original code:

```
1 lw $1, 0($0)
2 lw $2, 4($0)
3 add $3, $1, $2
4 sw $3, 12($0)
5 lw $4, 8($0)
6 add $3, $1, $4
7 sw $3, 16($0)
```

2 stalls

13 cycles

Scheduled code:

```
1 lw $1, 0($0)
2 lw $2, 4($0)
5 lw $4, 8($0)
3 add $3, $1, $2
4 sw $3, 12($0)
6 add $3, $1, $4
7 sw $3, 16($0)
```

no stalls

11 cycles

## Another type of optimization – loop unrolling

### Example 1

```
for (i=1; i<=1000; i= i+1)  
    x[i] = x[i] + y[i];
```

```
// parallel loop
```

## Example 2

```
for (i=1; i<=100; i= i+1){  
    a[i] = a[i] + b[i];    //s1  
    b[i+1] = c[i] + d[i];  //s2  
}
```

Is this loop parallel?

Statement s1 uses the value assigned in the previous iteration by statement s2, so there is a loop-carried dependency between s1 and s2. Despite this dependency, this loop can be made parallel because the dependency is not circular:

- neither statement depends on itself;
- while s1 depends on s2, s2 does not depend on s1.

## Example 2

```
for (i=1; i<=100; i= i+1){  
    a[i] = a[i] + b[i];    //s1  
    b[i+1] = c[i] + d[i];  //s2  
}
```

A loop can be **parallel** unless there is a cycle in the dependencies, since the absence of a cycle means that the dependencies give a partial ordering on the statements.

- There is no dependency from s1 to s2. Then, interchanging the two statements will not affect the execution of s2.
- On the first iteration of the loop, statement s1 depends on the value of b[1] computed prior to initiating the loop.

## Example 2, contd

```
a[1] = a[1] + b[1];  
for (i=1; i<=99; i= i+1){  
    b[i+1] = c[i] + d[i];  
    a[i+1] = a[i+1] + b[i+1];  
}  
b[101] = c[100] + d[100];
```

### Example 3

```
for (i=1; i<=100; i= i+1){  
    a[i+1] = a[i] + c[i];    //S1  
    b[i+1] = b[i] + a[i+1];  //S2  
}
```

This loop is not parallel because it has cycles in the dependencies, namely the statements S1 and S2 depend on themselves!

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.

To avoid stalls, **a dependent instruction must be separated from the source instruction by a distance** in clock cycles equal to the pipeline latency of that source instruction.



Our example uses a simple loop that adds a scalar value to an array in memory:

```
for (i=1; i<=1000; i++)  
    x[i] = x[i] + s;
```

1) The first step is to translate the above segment to DLX assembly language:

Loop: LD F0, 0(R1) ;F0 - array element

ADDD F4, F0, F2 ;add scalar in F2

SD 0(R1), F4 ;store result

SUBI R1, R1, #8 ;decrement pointer

;8 bytes (per double)

BENZ R1, Loop ;branch R1 != zero

Our example uses a simple loop that adds a scalar value to an array in memory:

```
for (i=1; i<=1000; i++)  
    x[i] = x[i] + s;
```

1) The first step is to translate the above segment to DLX assembly language:

Loop: LD F0, 0(R1) ;F0 - array element

ADDD F4, F0, F2 ;add scalar in F2

SD 0(R1), F4 ;store result

SUBI R1, R1, #8 ;decrement pointer

;8 bytes (per double)

BENZ R1, Loop ;branch R1 != zero

9 Cycles

Loop: LD F0, 0(R1) 1

stall 2

ADDD F4, F0,F2 3

stall 4

stall 5

SD 0(R1), F4 6

SUBI R1, R1,#8 7

BENZ R1, Loop 8

stall 9

After the loop is unrolled so that there are 4 copies of the loop body, assuming R1 is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations, and do not reuse any of the registers.

```
Loop: LD F0, 0(R1) 1
      stall 2
      ADDD F4, F0, F2 3
      stall 4
      stall 5
      SD 0(R1), F4 6 ;drop SUBI &BNEZ
      LD F6, -8(R1) 7
      stall 8
      ADDD F8, F6, F2 9
      stall 10
      stall 11
```

SD -8(R1), F8 12 ;drop SUBI &BNEZ

LD F10,-16(R1) 13

stall 14

ADDD F12,F10,F2 15

stall 16

stall 17

SD -16(R1), F12 18 ;drop SUBI &BNEZ

LD F14,-24(R1) 19

stall 20

ADDD F16,F14,F2 21

stall 22

stall 23

SD -24(R1),F16 24

SUBI R1, R1, #32 25

BENZ R1, Loop 26

stall 27

**; 27 clock cycles per iteration;  
;27/4 = 6.8 clock cycles per element**

# Dynamic scheduling

Instruction scheduling is important for getting the best performance out of a processor; if the compiler does a bad job (or doesn't even try), performance will suffer.

As a result, modern processors (e.g. Intel Pentium) have dedicated hardware for performing instruction scheduling dynamically as the code is executing.

This may appear to render compile-time scheduling rather redundant.

# Dynamic scheduling

But:

- This is still compiler technology, just increasingly being implemented in hardware.
- Somebody — now hardware designers — must still understand the principles.
- Embedded processors may not do dynamic scheduling, or may have the option to turn the feature off completely to save power, so it's still worth doing at compile-time.

# Scheduling

- Výstup
- Přiřazení času každému uzlu dagu
- Aplikace výstupu schedulingu
- Běžné procesory (Intel IA-32, včetně x86\_64)
  - Seřazení instrukcí podle schedulovaného času
  - Procesor nemusí dodržet předpokládaný čas
- Procesory se sekvenčními body (Intel IA-64)
  - V kódu jsou mezi instrukcemi označeny sekvenční body (stops)
  - Procesor má právo přeházet pořadí instrukcí mezi sekvenčními body
    - Ignorují se antidependence i některé dependence
  - Výstupem scheduleru jsou i sekvenční body
- VLIW procesory
  - Very Large Instruction Word
    - Instrukce řídí paralelní činnost jednotek procesoru
  - Jeden schedulovaný čas = jedna instrukce



# Scheduling

- Scheduling pouze odhaduje skutečné časování
- Skutečné časování je ovlivněno nepředvídatelnými jevy
  - Zbytky rozpracovaných instrukcí z předchozího BB
    - Řešení: Trace-scheduling, řízení profilem
  - Paměťová hierarchie
    - Doba přístupu k paměti závisí na přítomnosti v cache
    - Obvykle se předpokládá nejlepší možný průběh
    - Speciální problém: Multithreaded aplikace na multiprocесorech
  - Fetch bandwidth
    - Instrukce nemusí být načteny a dekodovány včas
    - Zdržují skoky a soupeření o přístup do paměti
    - Přesné simulování fetch jednotky by neúměrně komplikovalo scheduler
- Scheduler nezná skutečné závislosti přístupů do paměti
  - Musí postupovat opatrně a zohledňuje i nejisté závislosti
  - Procesor zná skutečné adresy přístupů a detekuje pouze skutečné závislosti
    - Agresivně optimalizující procesor může zvolit zcela jiné pořadí instrukcí

# Scheduling

- Model procesoru
- Latence – časování závislých dvojic instrukcí
  - Počet cyklů procesoru, který musí proběhnout mezi referenčními body závislých instrukcí
  - U antidependencí a ve speciálních případech může být nulová
  - U procesorů se sekvenčními body může být záporná
  - Latence se obvykle zapisuje ke hranám dagu
    - Přiřazena na základě klasifikace závislosti podle tabulek latencí

## Example of a Machine Model

- Each machine cycle can execute 2 operations
- 1 ALU operation or branch operation

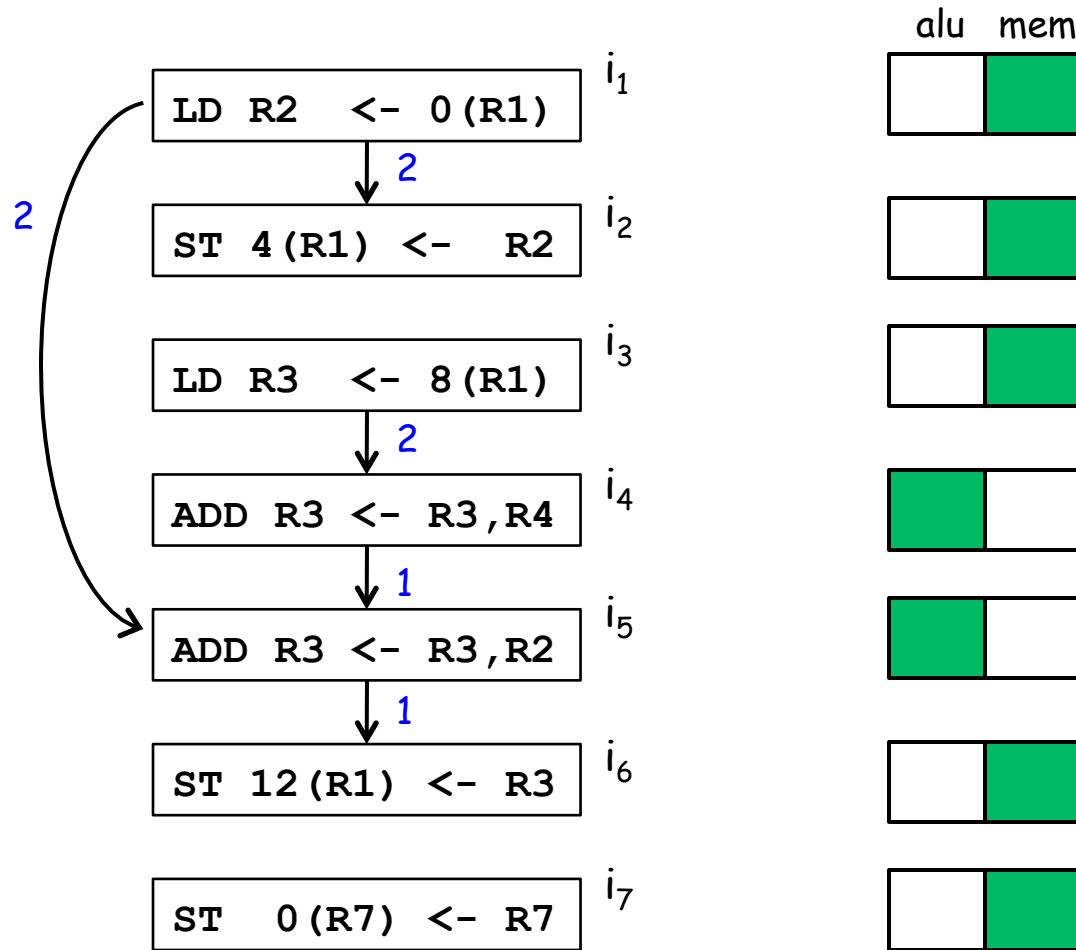
Op dst,src1,src2      executes in 1 clock

- 1 load or store operation

LD dst, addr      result is available in 2 clocks  
pipelined: can issue LD next clock

ST src, addr      executes in 1 clock cycle

## Basic Block Scheduling



## With Resource Constraints

- NP-complete in general → Heuristics time!

- **List Scheduling:**

READY = nodes with 0 predecessors

Loop until READY is empty {

Let **n** be the node in READY with **highest priority**

Schedule **n** in the **earliest** slot

that **satisfies precedence + resource constraints**

Update predecessor count of **n**'s successor nodes

Update READY

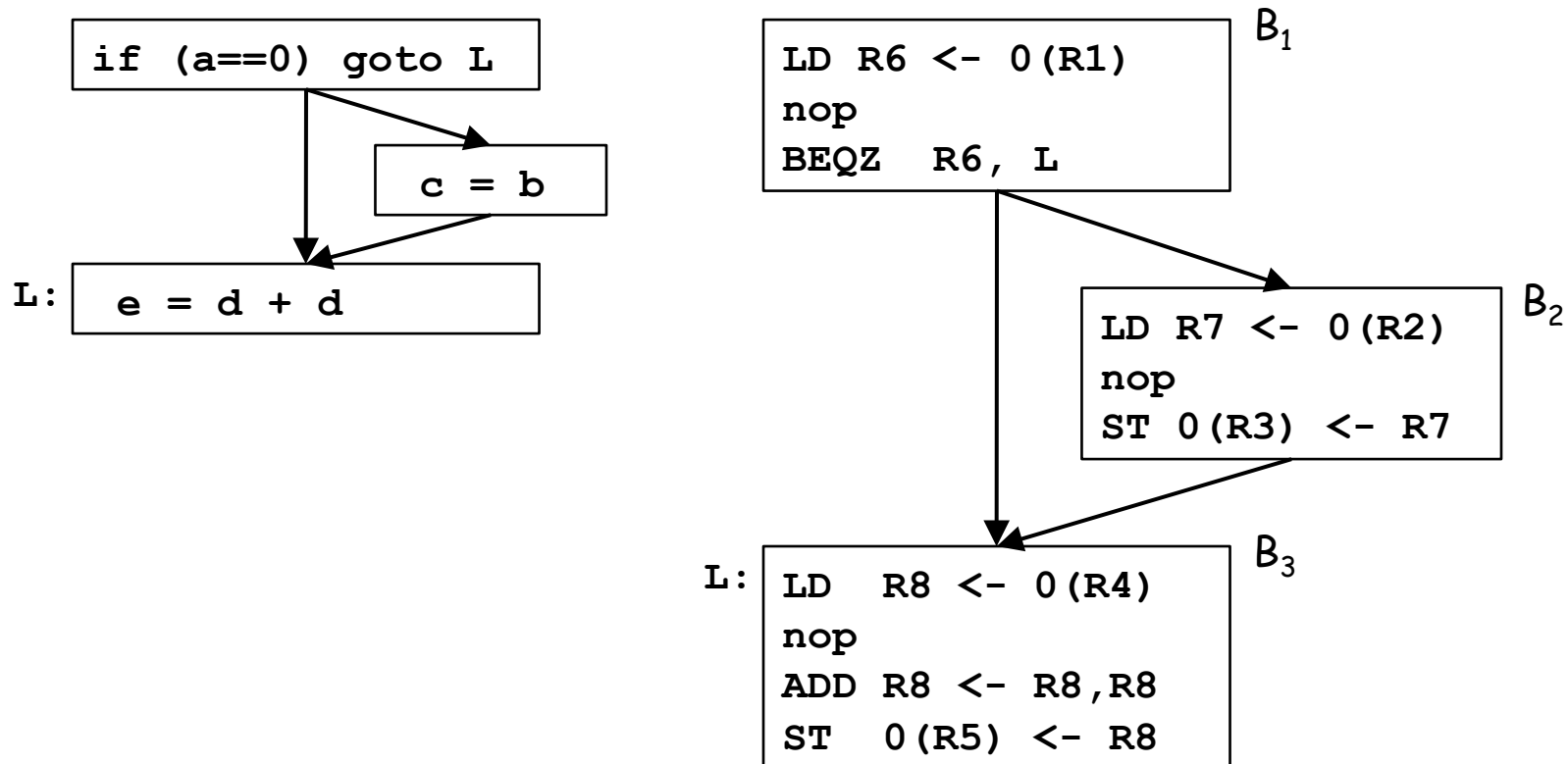
}

## List Scheduling

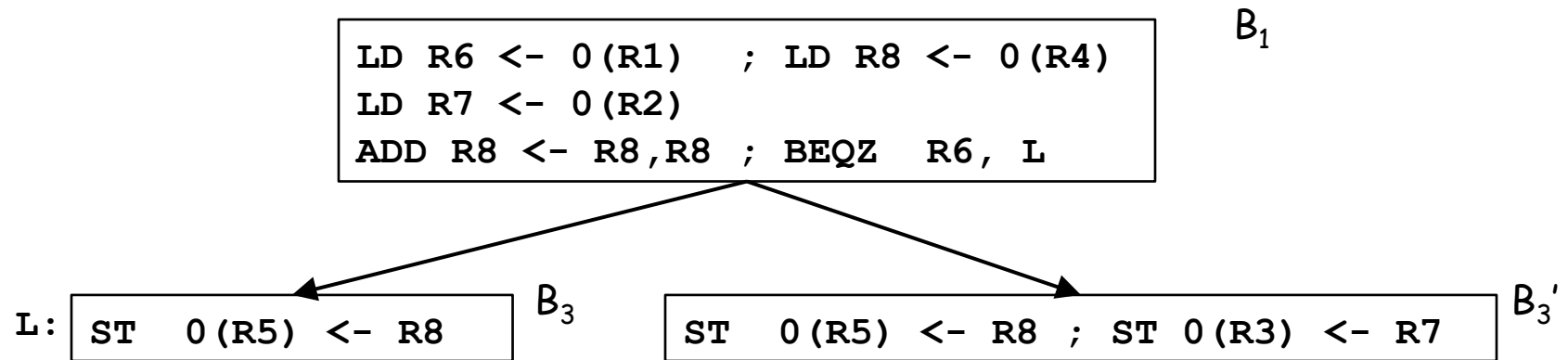
- **Scope: DAGs**
  - Schedules operations in **topological** order
  - Never backtracks
- **Variations:**
  - **Priority function** for node **n**
    - **critical path**: max clocks from **n** to any node
    - resource requirements
    - source order

## II. Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind.



## Result of Code Scheduling





# Terminology

## Control equivalence:

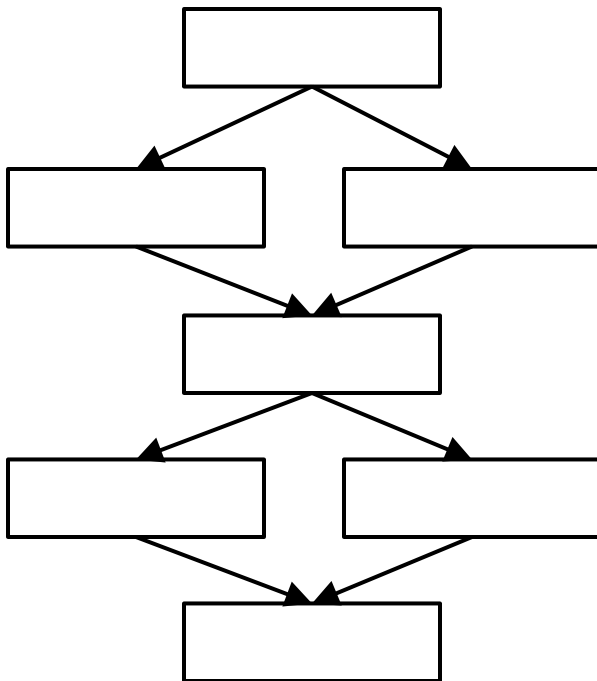
- Two operations  $o_1$  and  $o_2$  are *control equivalent* if  $o_1$  is executed if and only if  $o_2$  is executed.

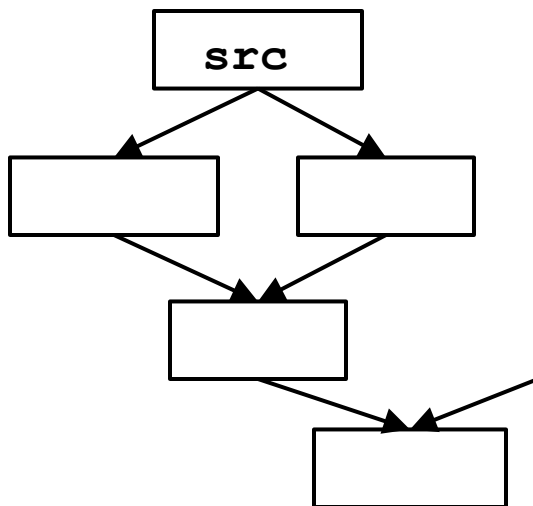
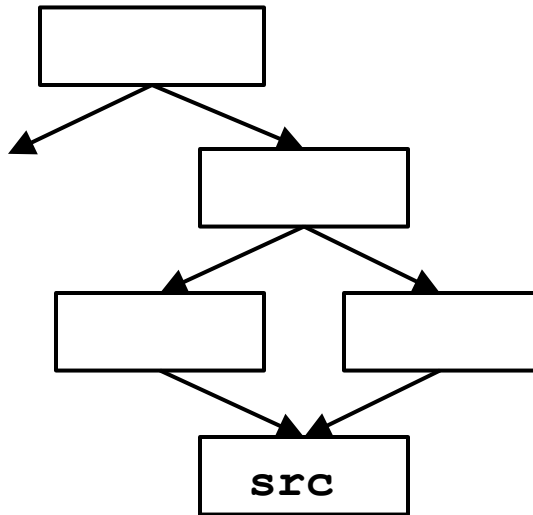
## Control dependence:

- An op  $o_2$  is *control dependent* on op  $o_1$  if the execution of  $o_2$  depends on the outcome of  $o_1$ .

## Speculation:

- An operation  $o$  is *speculatively* executed if it is executed before all the operations it depends on (control-wise) have been executed.
- Requirement: Raises no exception,  
Satisfies data dependences





## Code Motions

Goal: Shorten execution time **probabilistically**

Moving instructions **up**:

- Move instruction to a cut set (from entry)
- Speculation: even when not anticipated.

Moving instructions **down**:

- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code