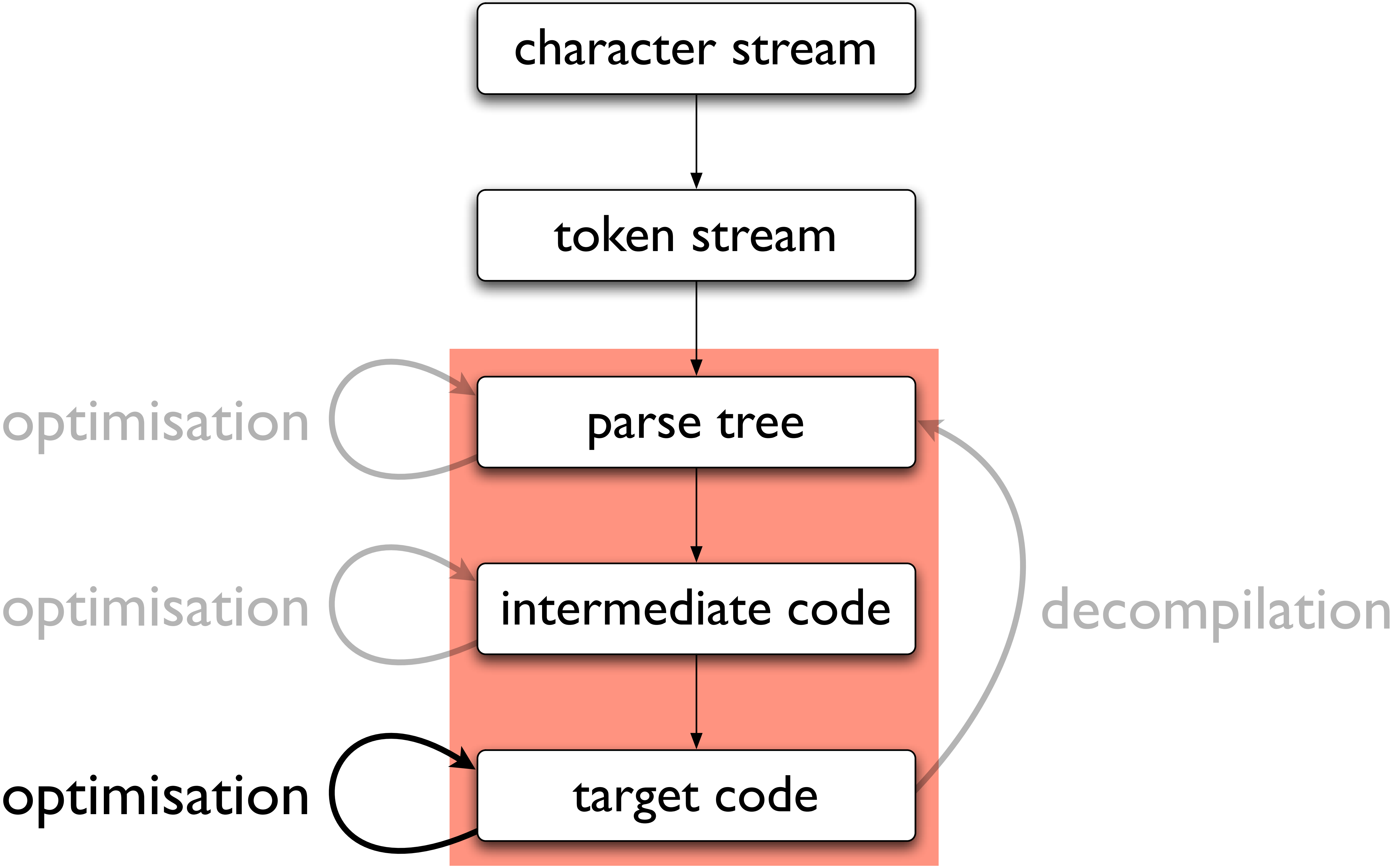
Part C

Instruction scheduling

Instruction scheduling



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

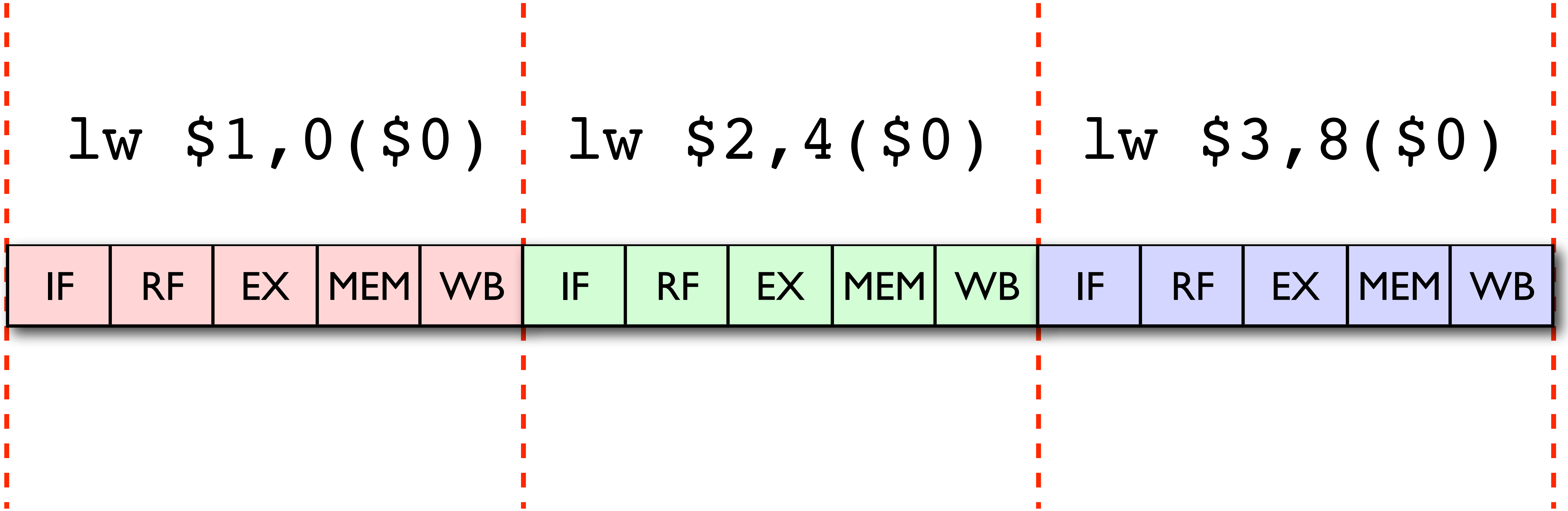
In *single-cycle* processor designs, an entire instruction is executed in a single clock cycle.

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)
```

IF RF EX MEM WB IF RF EX MEM WB IF RF EX MEM WB

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.

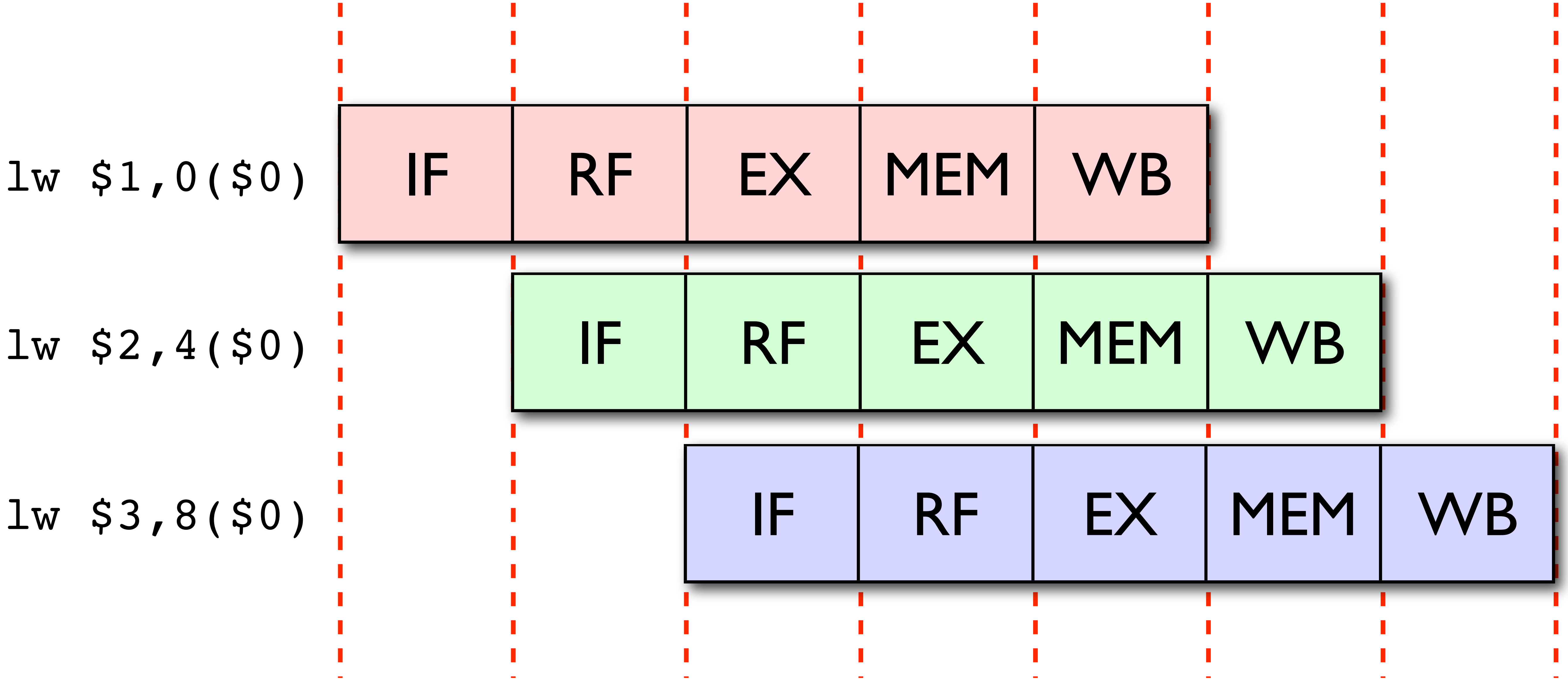
In this case we can naïvely translate our optimised 3-address code by expanding each intermediate instruction into the appropriate sequence of target instructions; clever reordering is unlikely to yield any benefits.

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



lw $1,0($0) IF RF EX MEM WB

lw $2,4($0) IF RF EX MEM WB

lw $3,8($0) IF RF EX MEM WB

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.

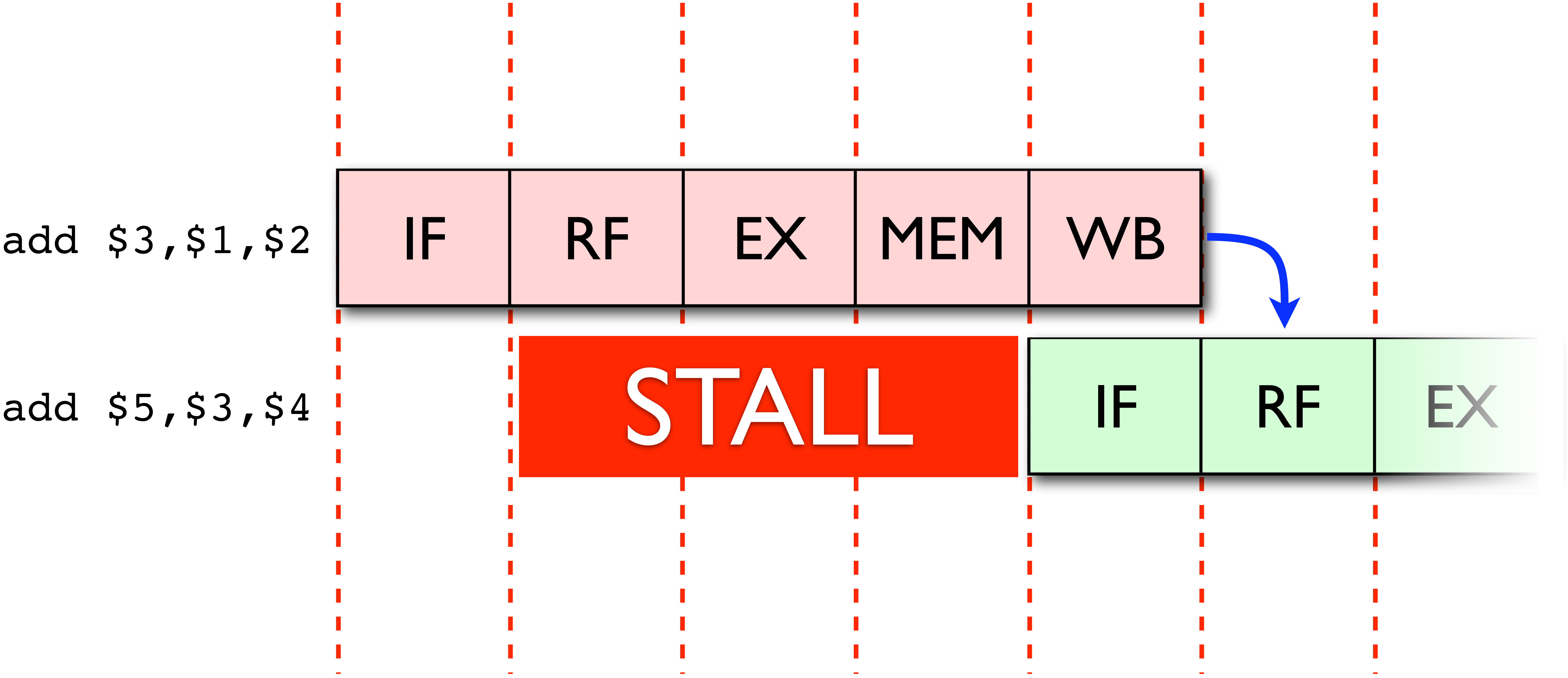
Pipeline hazards

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`.

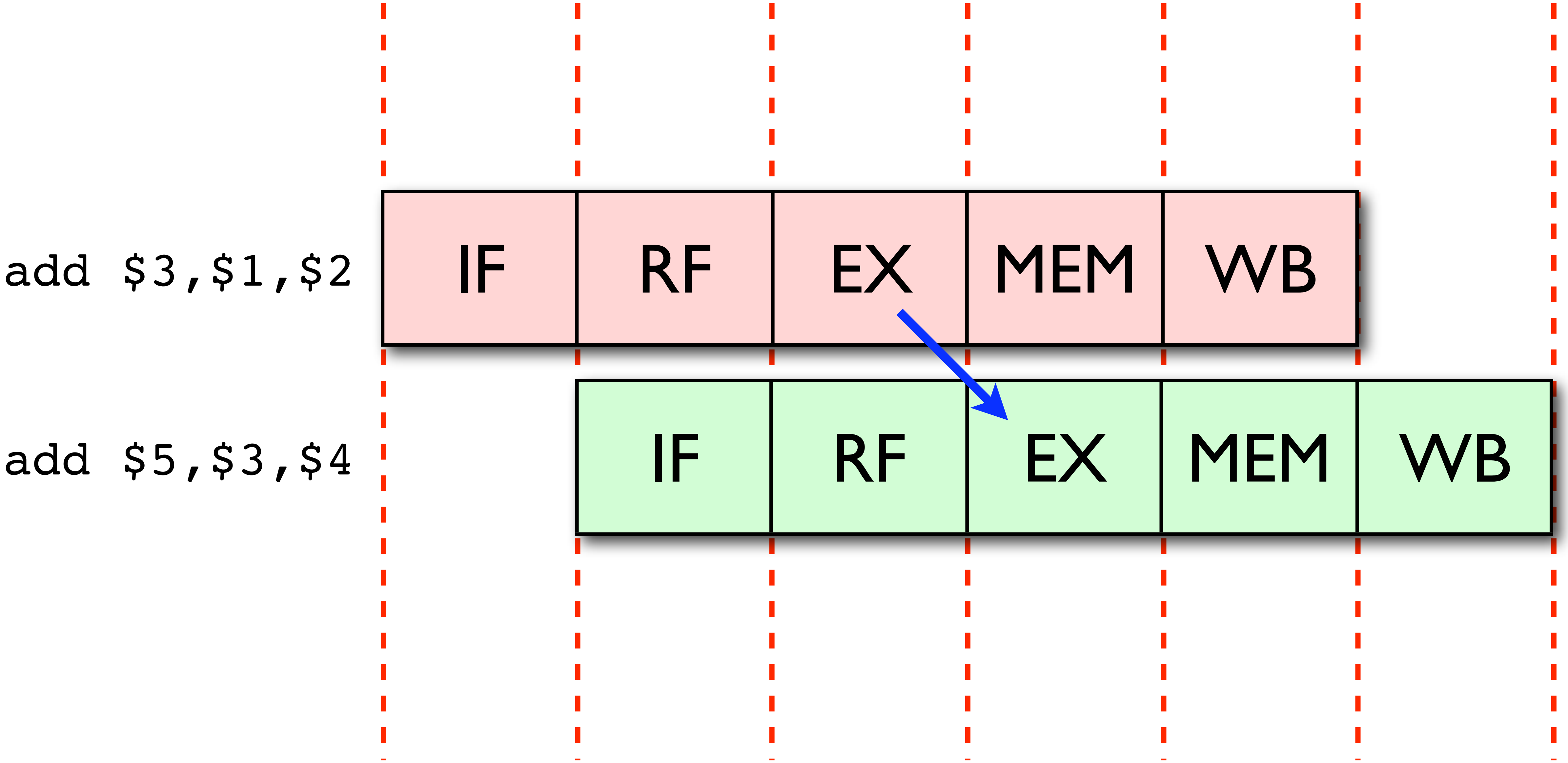
Pipeline hazards



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



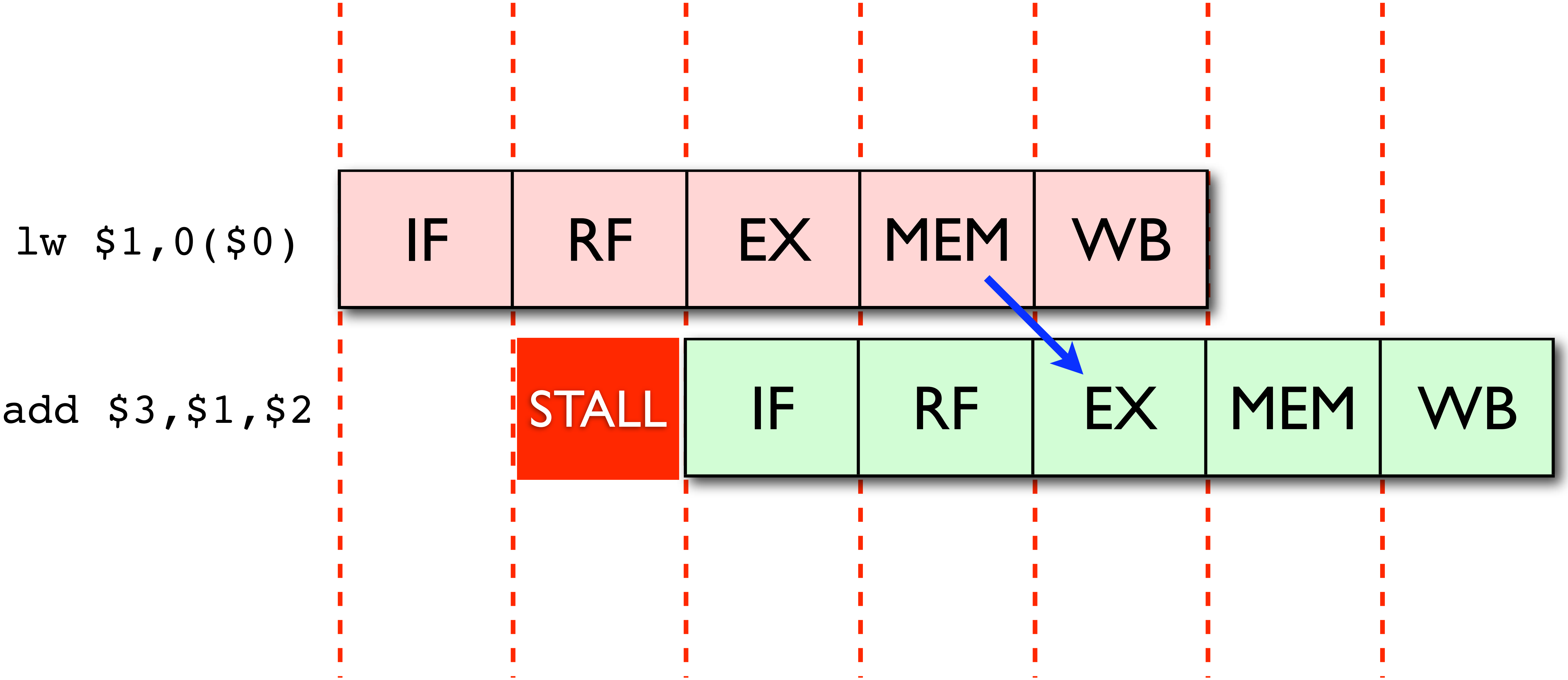
add $3,$1,$2 IF RF EX MEM WB

add $5,$3,$4 IF RF EX MEM WB

Pipeline hazards

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

Pipeline hazards

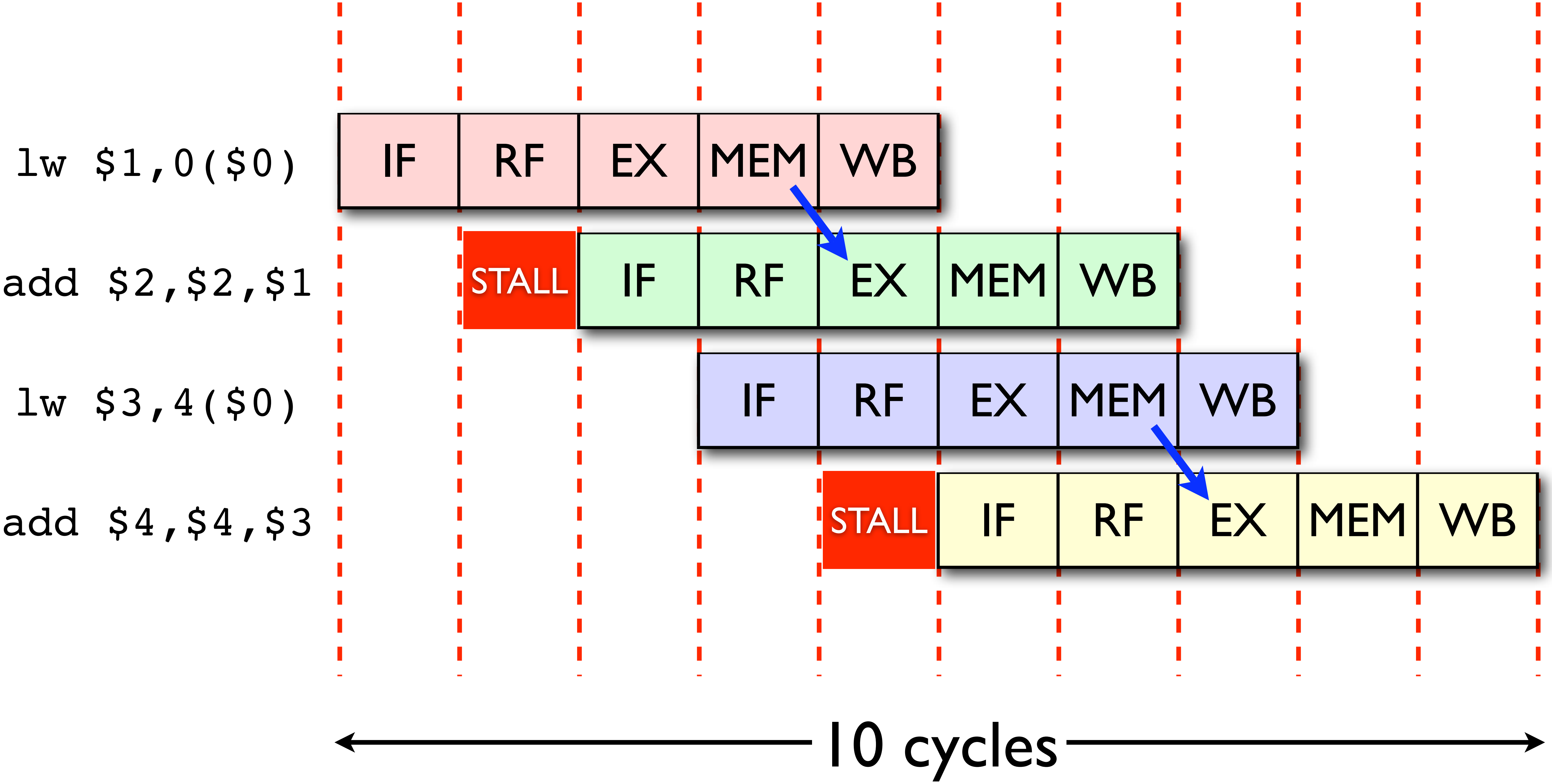


Instruction order

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
```

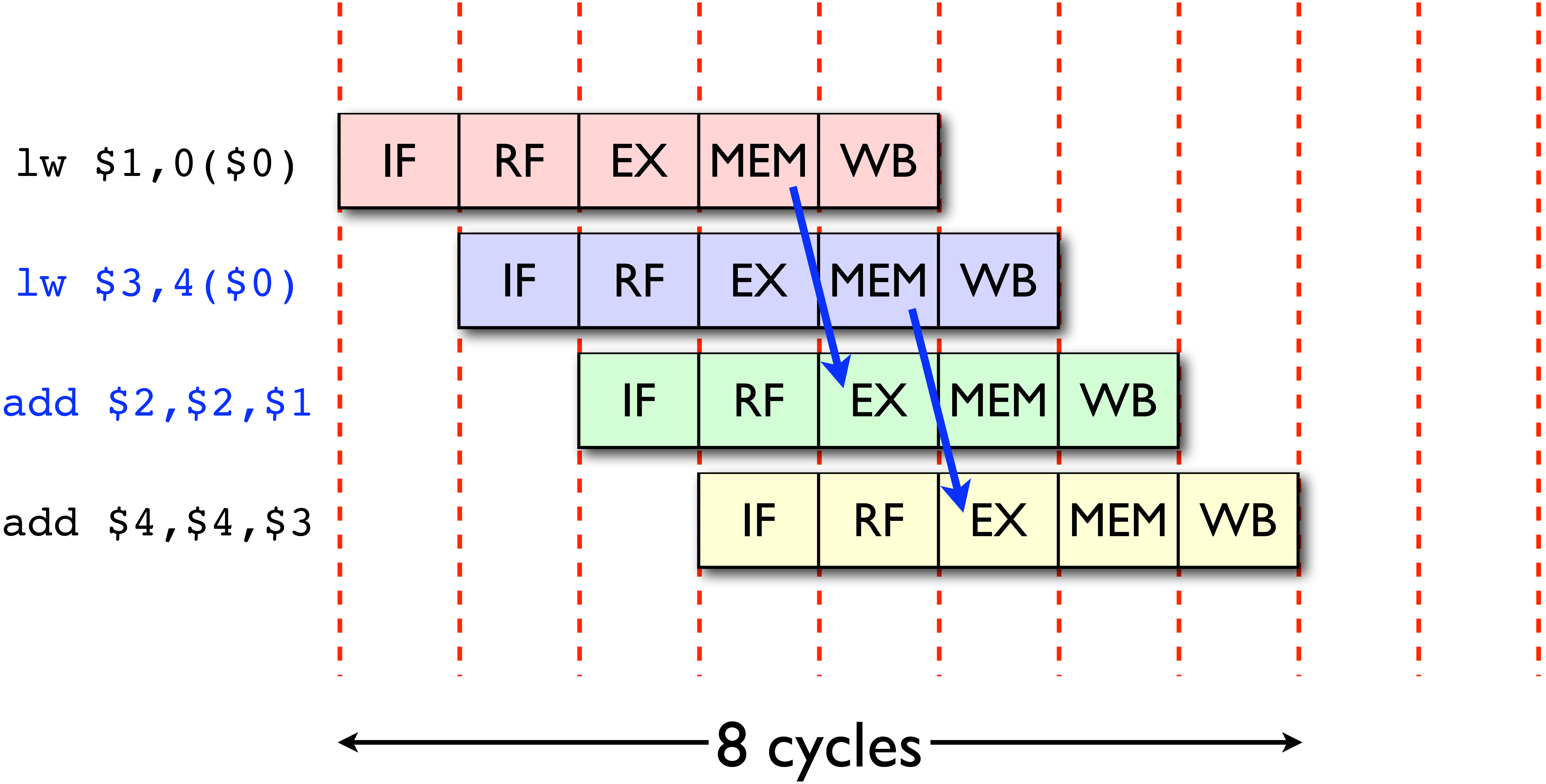
Instruction order



lw $1,0($0)	IF	RF	EX	MEM	WB					
add $2,$2,$1		STALL	IF	RF	EX	MEM	WB			
lw $3,4($0)				IF	RF	EX	MEM	WB		
add $4,$4,$3					STALL	IF	RF	EX	MEM	WB

10 cycles

Instruction order



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

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

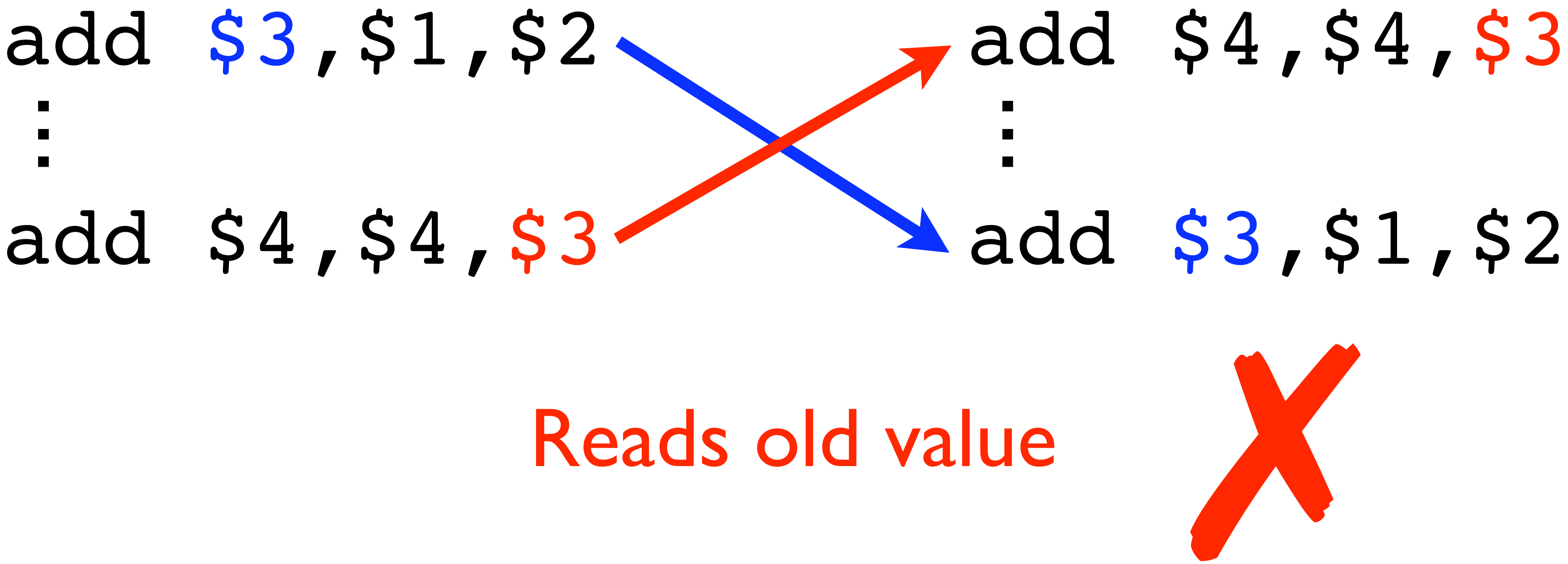
There are three kinds of data dependency:

- 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.



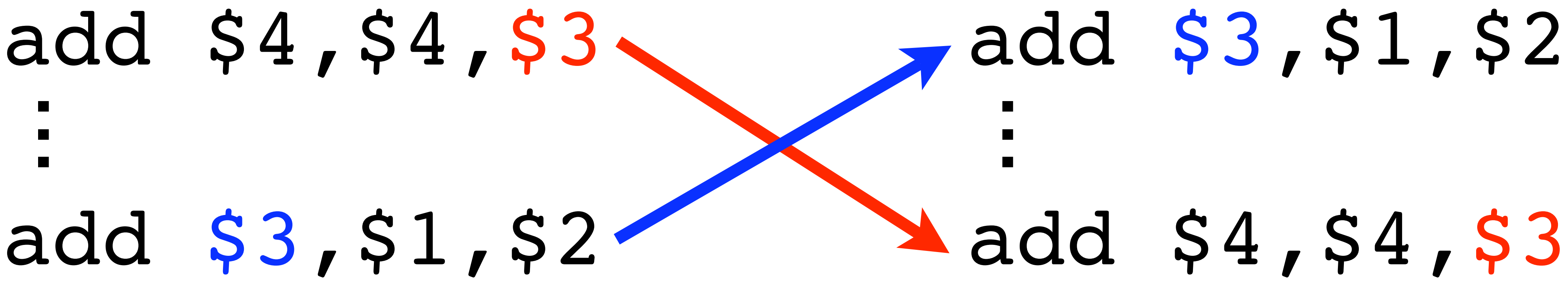
```
add $3,$1,$2
 ⋮
add $4,$4,$3
```

```
add $4,$4,$3
 ⋮
add $3,$1,$2
```

Reads old value ✗

Instruction dependencies

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

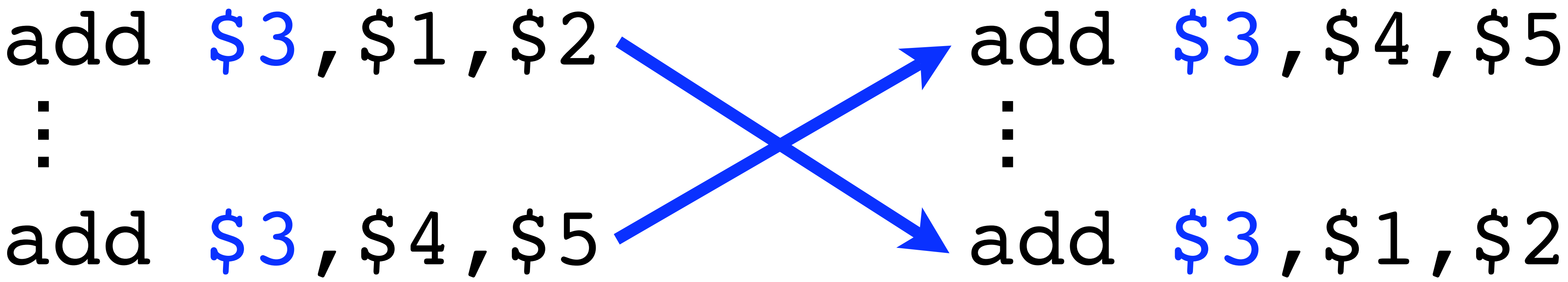


```
add $4,$4,$3          add $3,$1,$2
 ⋮                     ⋮
add $3,$1,$2          add $4,$4,$3
```

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.

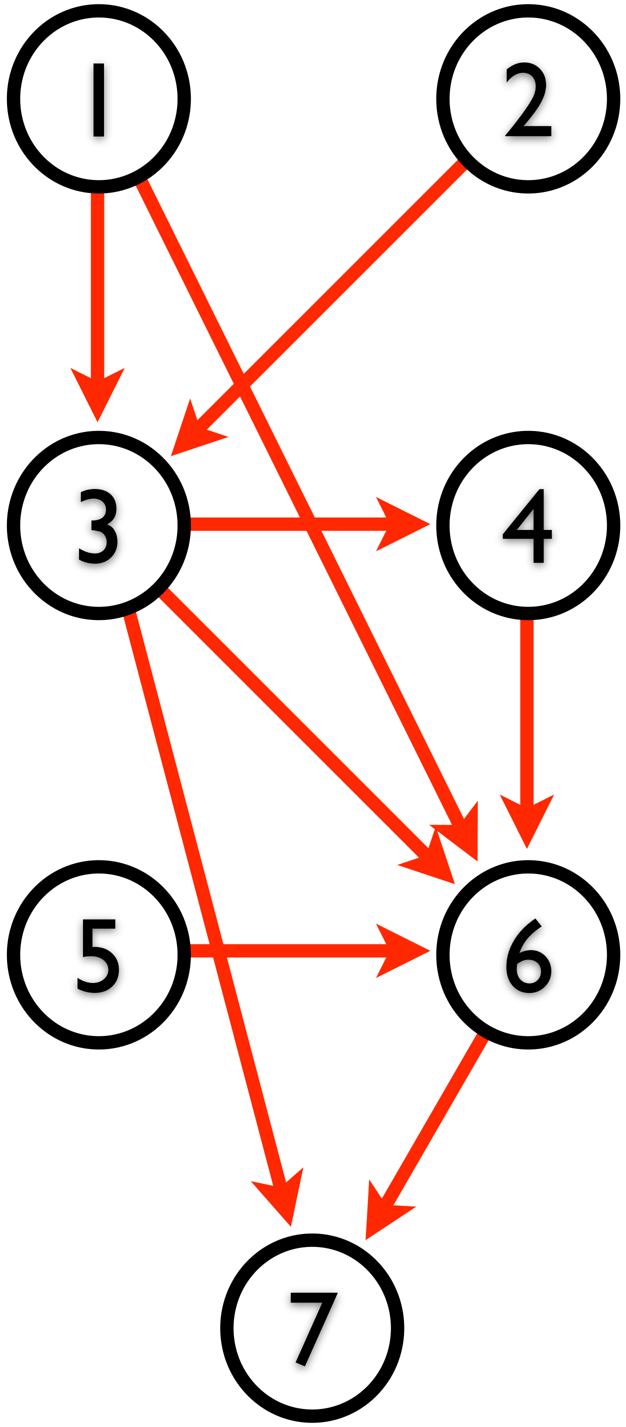
Preserving dependencies

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

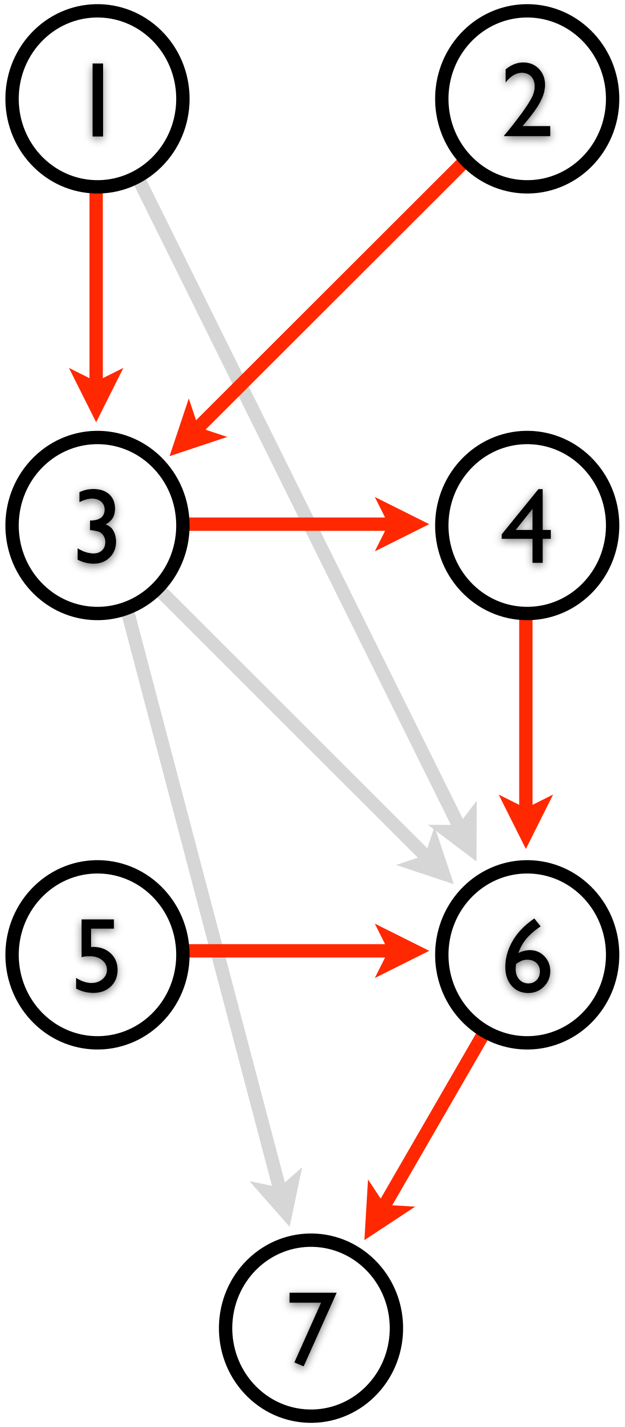
```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```



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.

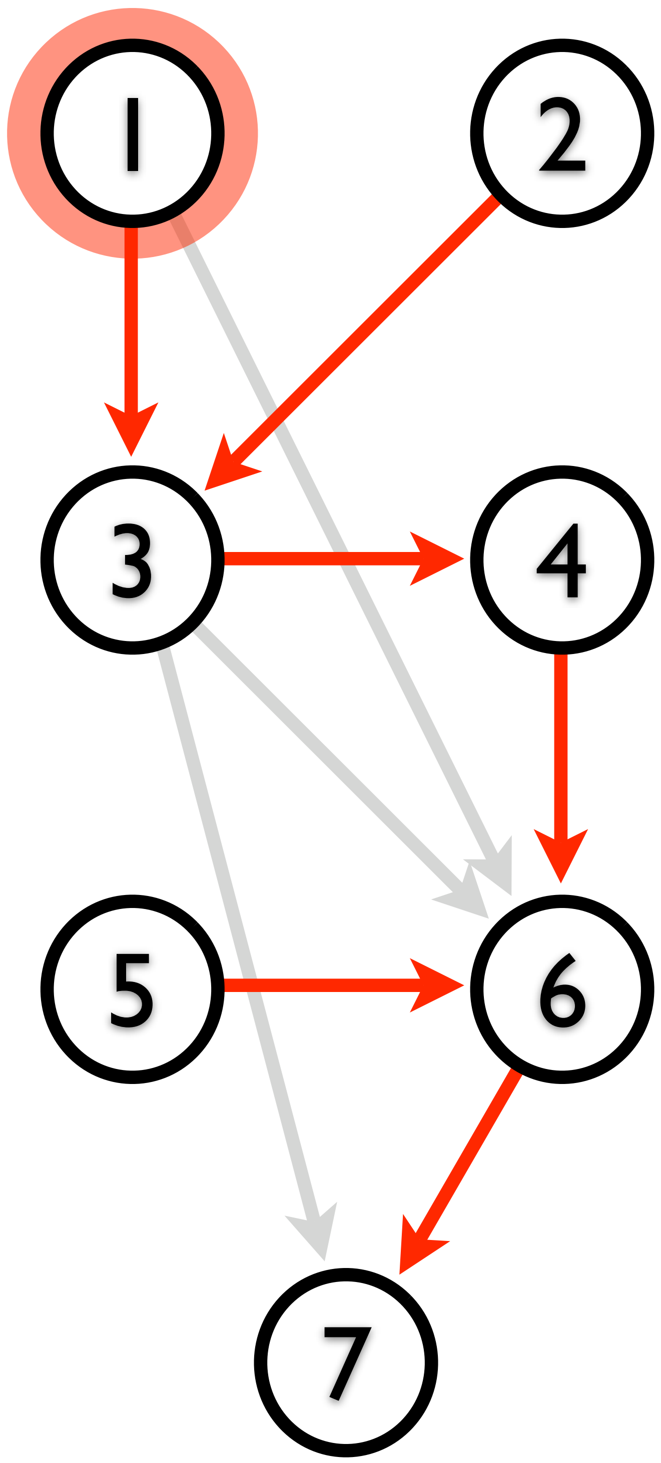
Algorithm

- 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.

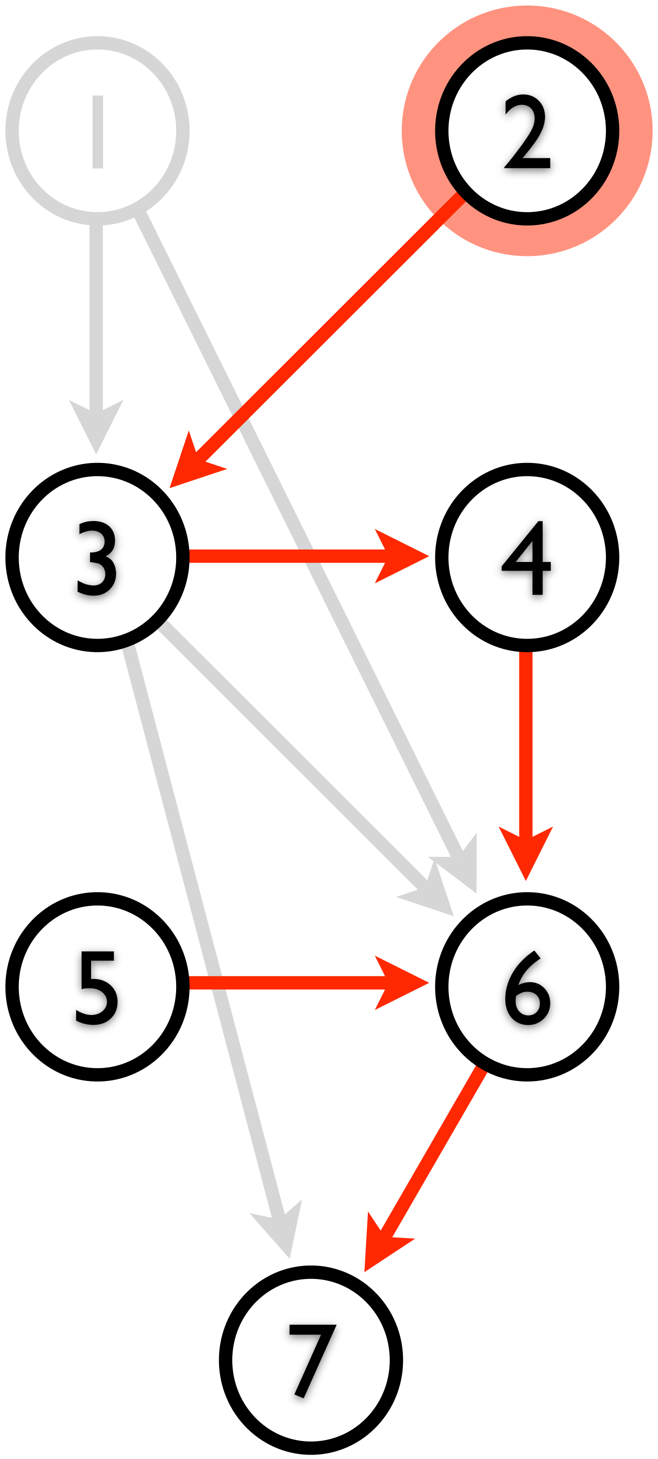
Algorithm



Candidates:
{ 1, 2, 5 }

```
1  lw $1,0($0)
```

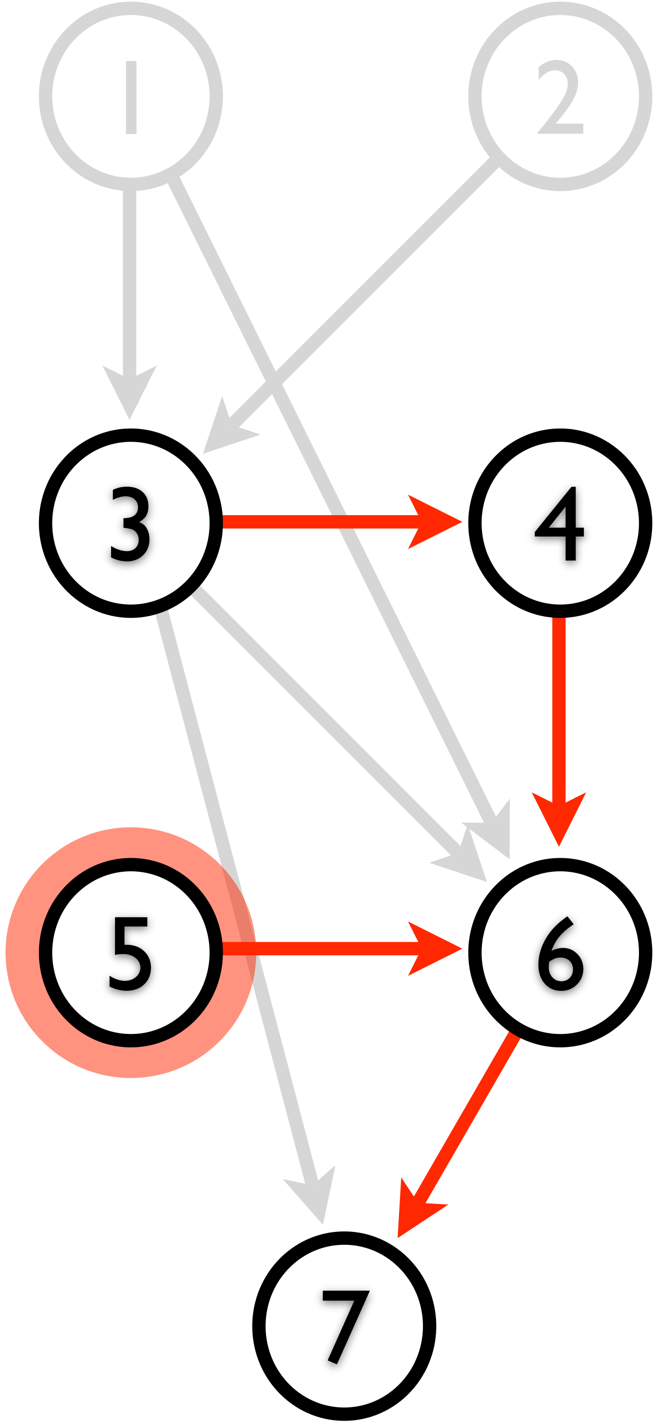
Algorithm



Candidates:
{ 2, 5 }

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

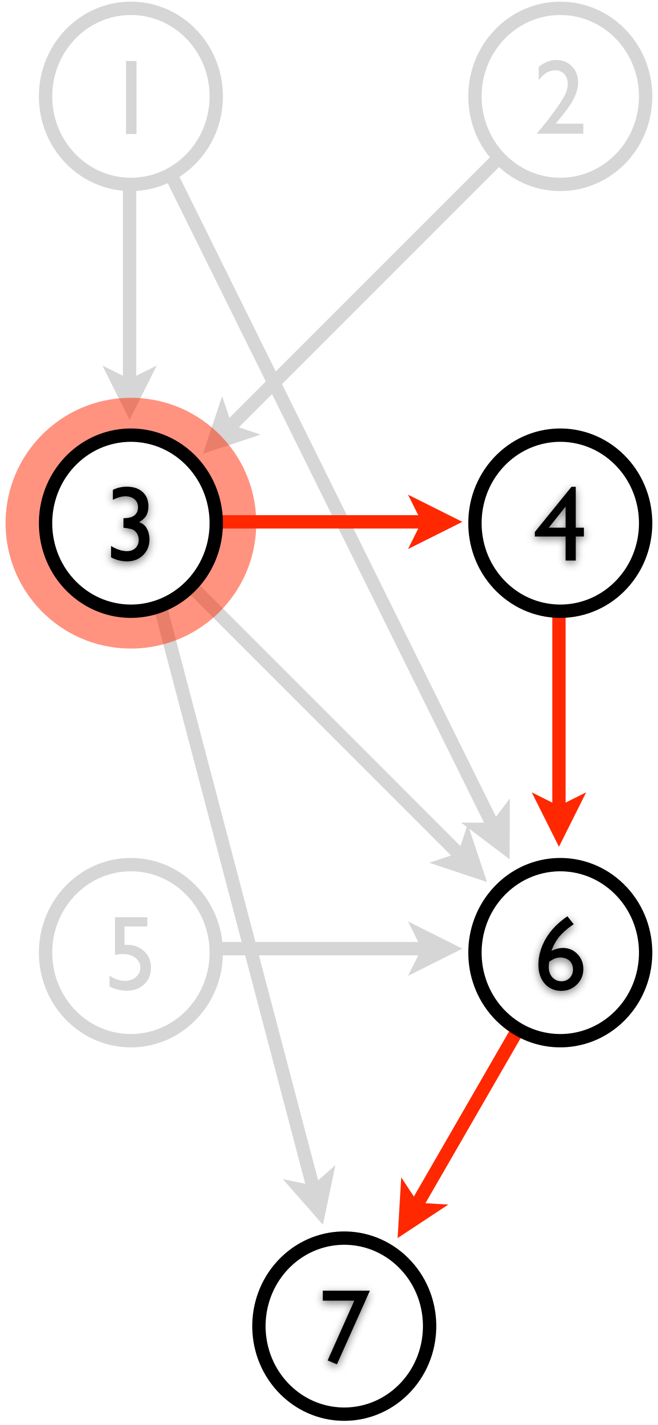
Algorithm



Candidates:
{ 3, 5 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
```

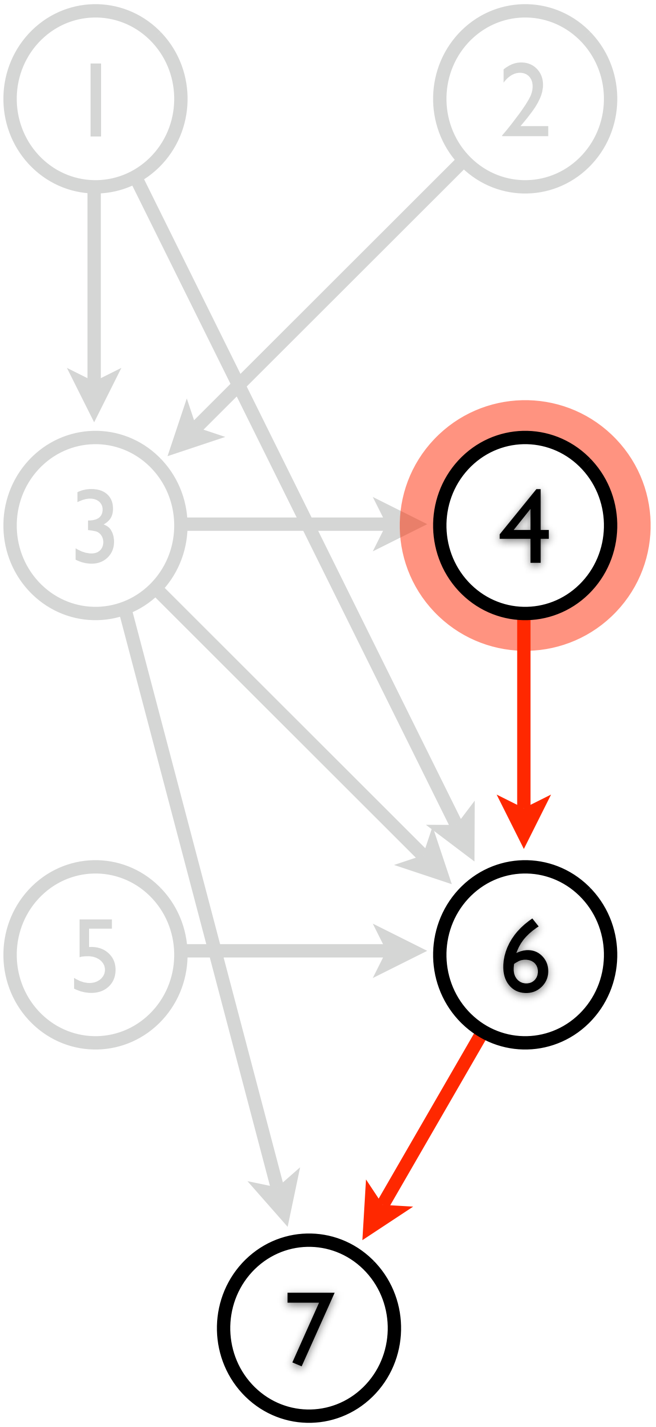
Algorithm



Candidates:
{ 3 }

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

Algorithm

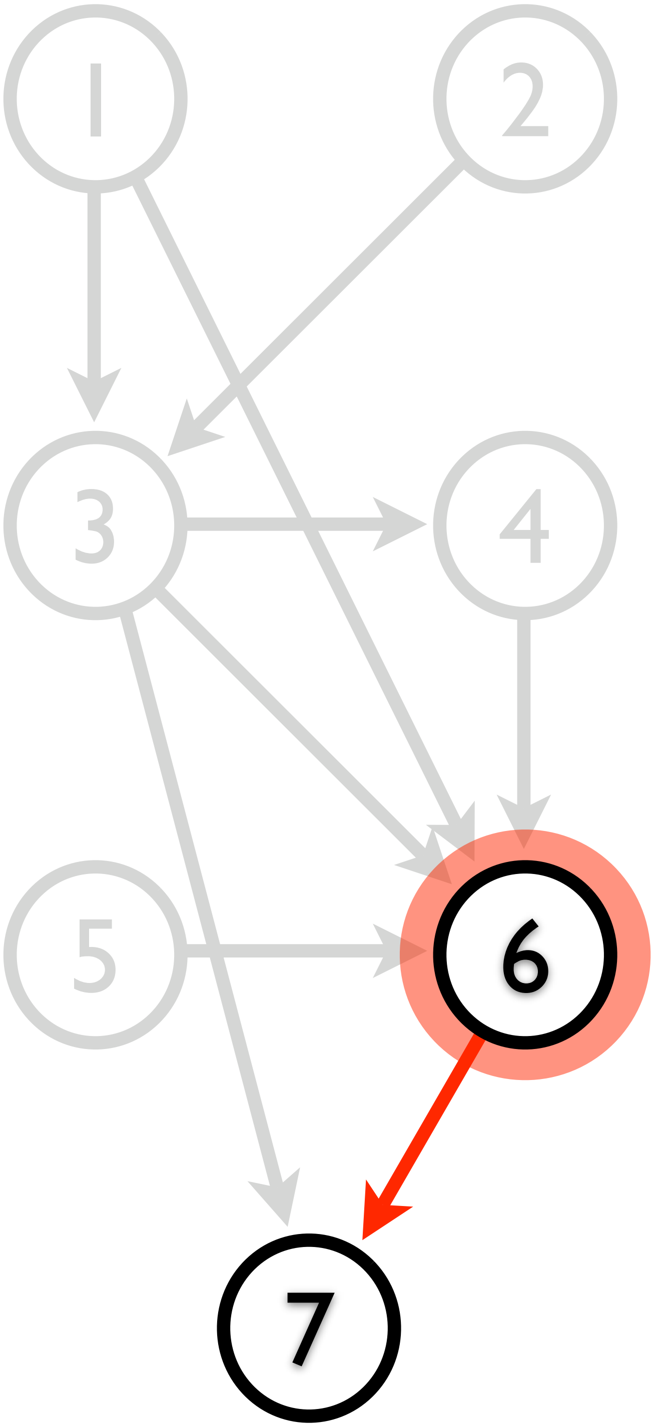


Candidates:

{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)
```

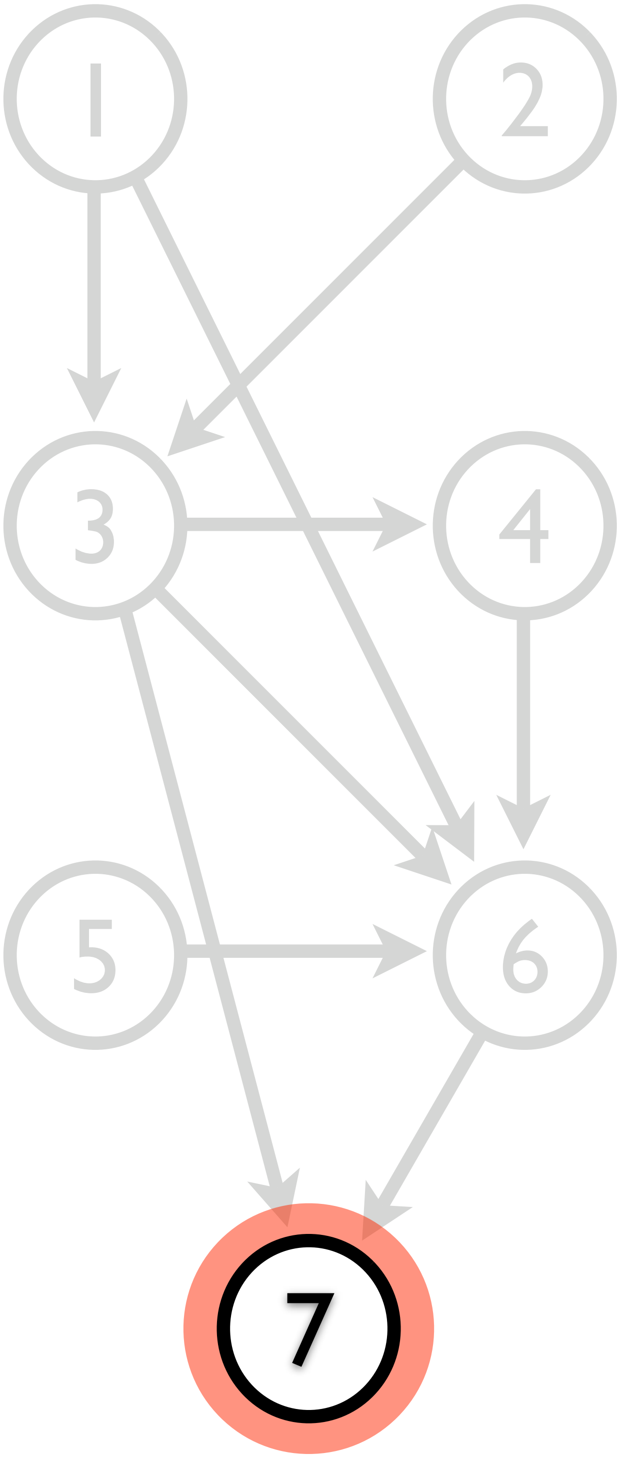
Algorithm



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

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.

Summary

- Instruction pipelines allow a processor to work on executing several instructions at once
- Pipeline hazards cause stalls and impede optimal throughput, even when feed-forwarding is used
- Instructions may be reordered to avoid stalls
- Dependencies between instructions limit reordering
- Static scheduling heuristics may be used to achieve near-optimal scheduling with an $O(n^2)$ algorithm