The University for world-class professionals

Manchester Metropolitan University

Hardware Architectures and the Implications for Compilers

Dr Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- ~~Semantic analysis~~
- ~~Machine-independent optimisation~~
- ~~Code generation~~
- **Hardware architectures**
- Machine-dependent optimisation
- Review

Outline

- Discuss *why* it is important to understand hardware architectures
- Explain the principles of *pipelining*
- Contrast *out-of-order*, *superscalar*, and *very long instruction word* processor architectures
- Reflect upon implications of memory hierarchy for compiler implementation

Motivation

- Two weeks ago, we looked at machine-independent optimisation
  - These techniques involve manipulating the intermediate representation levels in an attempt to achieve the smallest and/or fastest correct program.
  - These techniques are platform-independent, and pay little attention to the details of the target architecture.
- Last week we looked at code generation, where we produced code to run on a particular machine
  - We can improve target code further if we consider particular architectural characteristics of the target hardware.

The Basic Instruction Cycle

- Although we think of an assembly instruction as an atomic action, there are actually several functional units at the hardware level:
- consider: **add $1, $2, $3**

Instruction Fetch (IF)	Register Fetch (RF)	Execute (EX)	Memory Access (MEM)	Register write-back (WB)

- Not all instructions will use all five, but some do

Instruction execution

```
                  -----> time ----->
      1    2    3    4    5    6    7    8    9    10

Inst1 IF   ID   EX   MEM WB
```

Instruction execution

```
                  -----> time ----->
      1   2   3   4   5   6   7   8   9   10

Inst1 IF  ID  EX  MEM WB

Inst2                     IF  ID  EX  MEM WB
```

Pipelined Implementation

- In a pipelined processor, each functional unit works independently
  - Different functional units can be handling different instructions simultaneously
- Result from each unit is passed in a pipeline to the next at the start of each clock cycle
  - Clock cycle should be as long as the time for the slowest functional unit

Pipelined Example

```
                  -----> time ----->
      1   2   3   4   5   6   7   8   9   10  11  12
Inst1 IF  ID  EX  MEM WB
Inst2     IF  ID  EX  MEM WB
Inst3         IF  ID  EX  MEM WB
Inst4             IF  ID  EX  MEM WB
Inst5                 IF  ID  EX  MEM WB
Inst6                     IF  ID  EX  MEM WB
Inst7                         IF  ID  EX  MEM WB
```

- Pipelining allows several instruction steps to execute simultaneously
- In this example, up to 5 x faster execution
- Requires that instructions are ***independent***

Data Hazards

- Works fine if we can start a new instruction every clock cycle – up to five times speedup
- Instructions must be independent
- If the pipeline detects a hazard it initiates a *stall* (aka “bubble”) to wait for dependencies

Waiting for Data

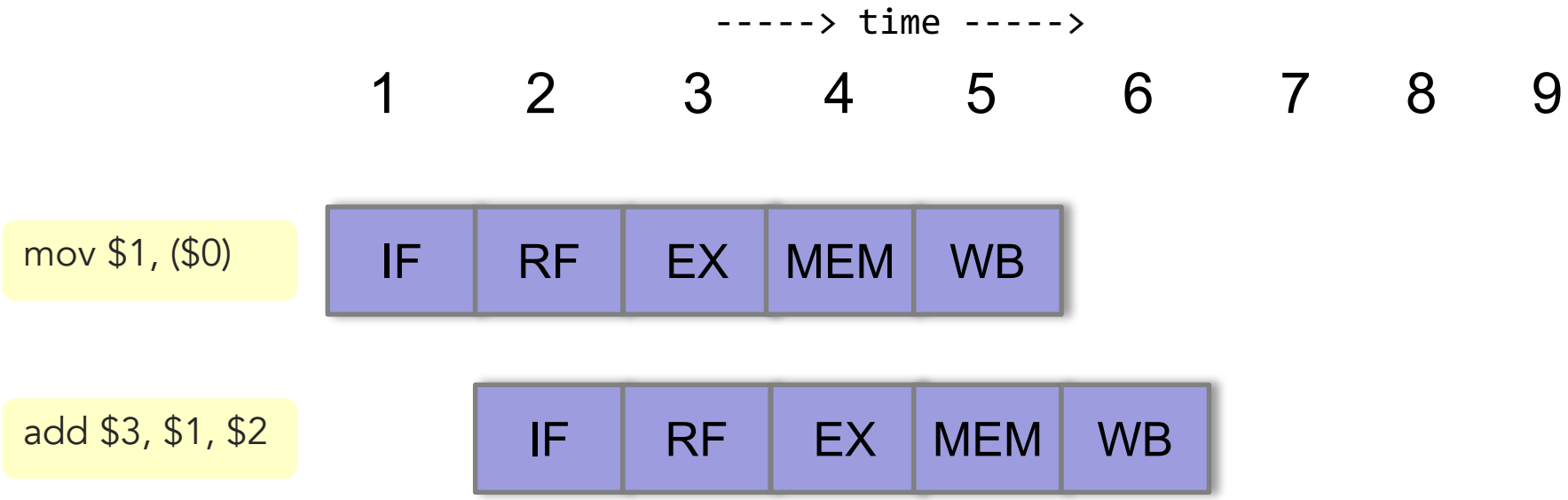
```
mov $1, ($0)

add $3, $1, $2
```

Waiting for Data

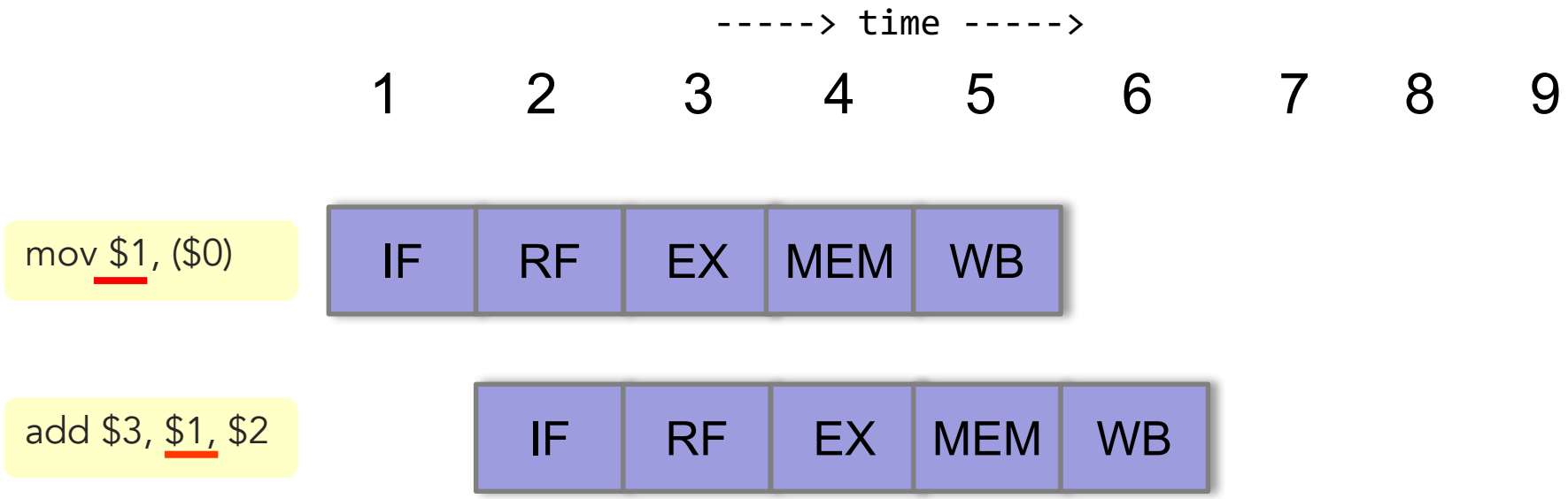
```
mov $1, ($0)

add $3, $1, $2
```

Notation:
$r – value in register r
($r) – value in memory, memory address is in register r
r – absolute value r

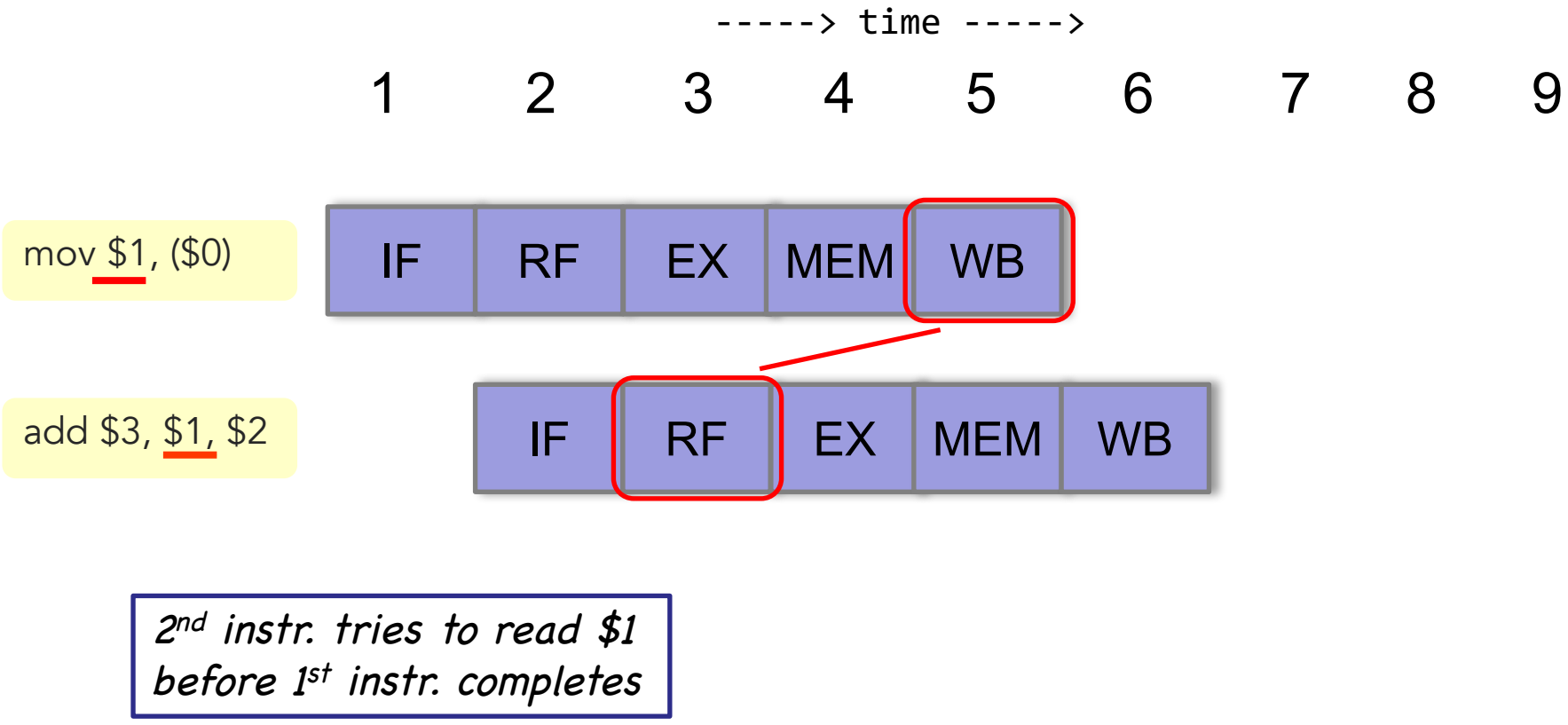
Waiting for Data



Waiting for Data

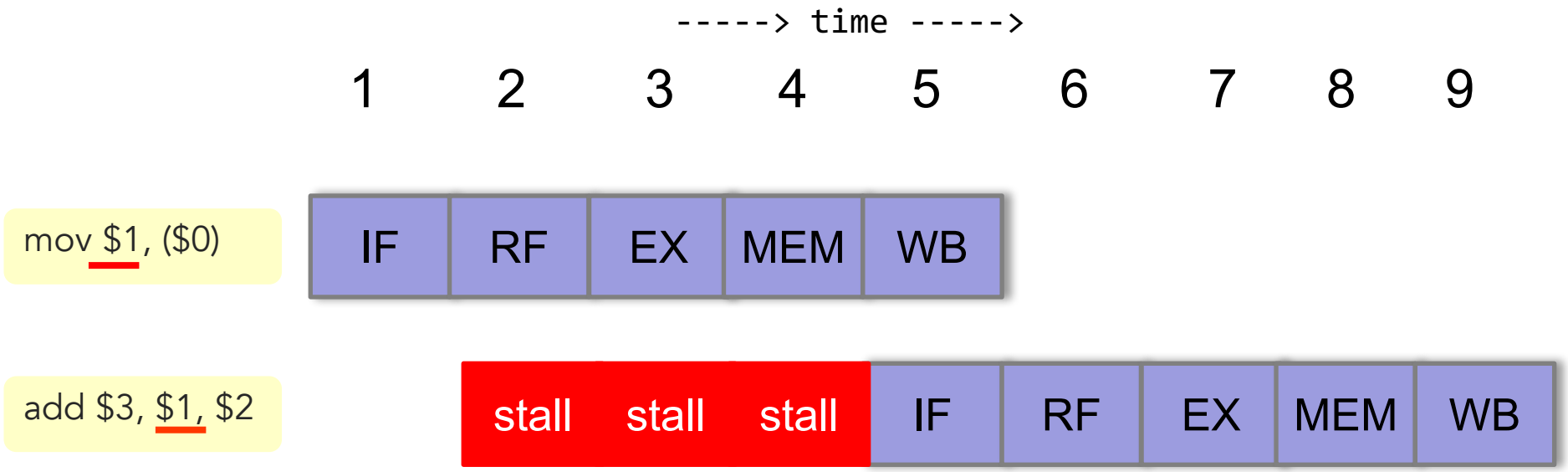


Waiting for Data

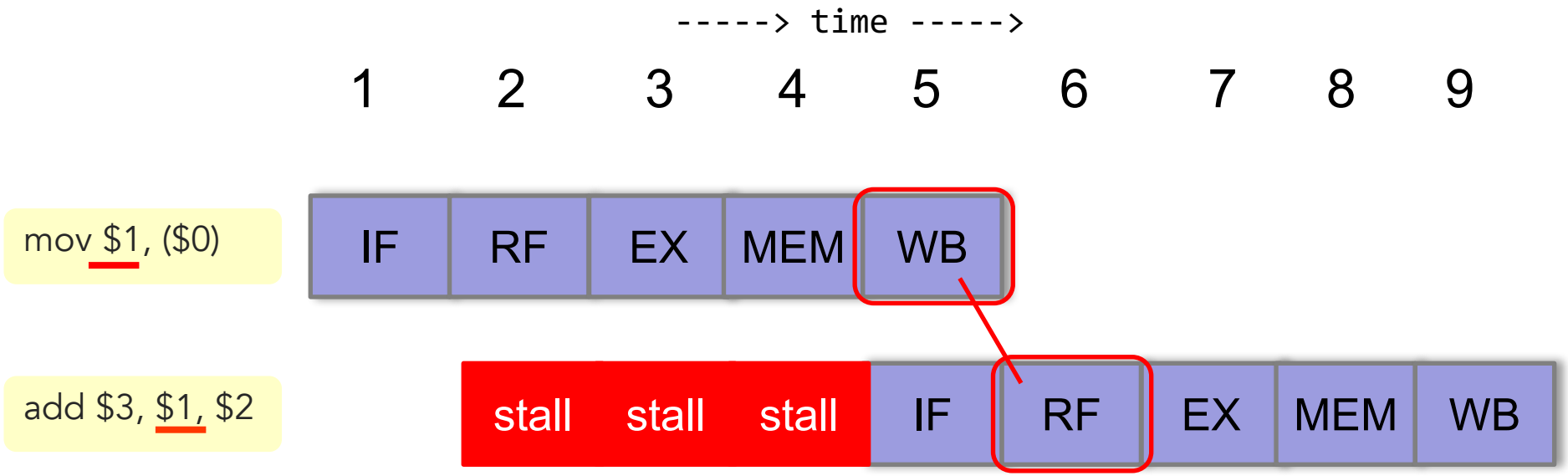


2nd instr. tries to read $1 before 1st instr. completes

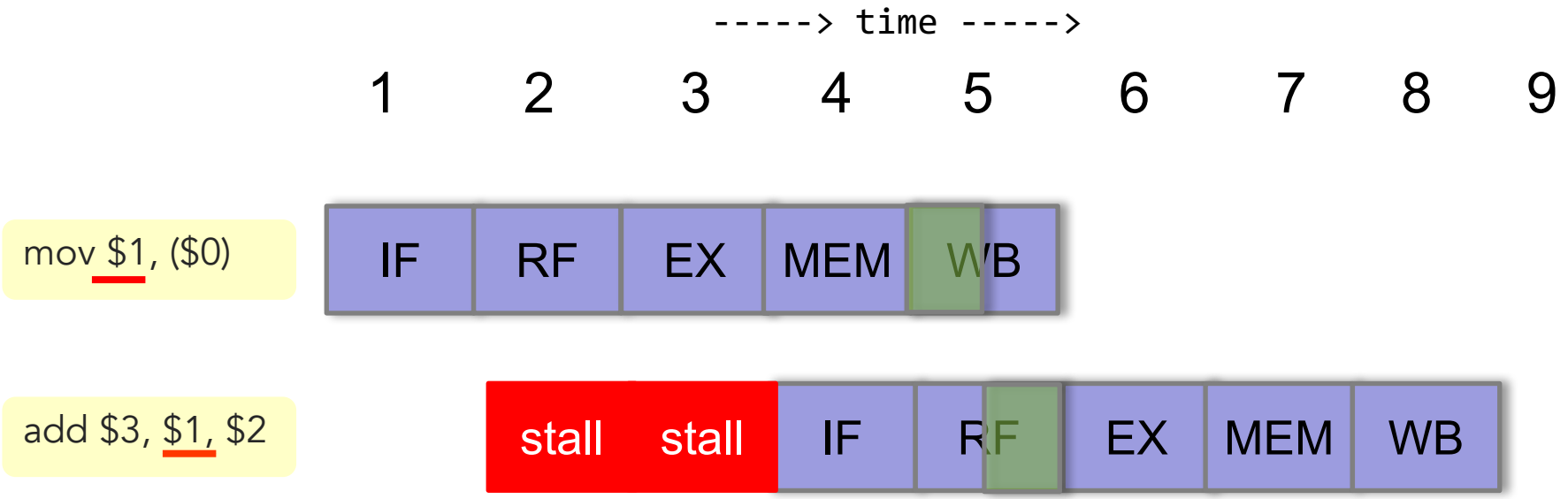
Waiting for Data



Waiting for Data

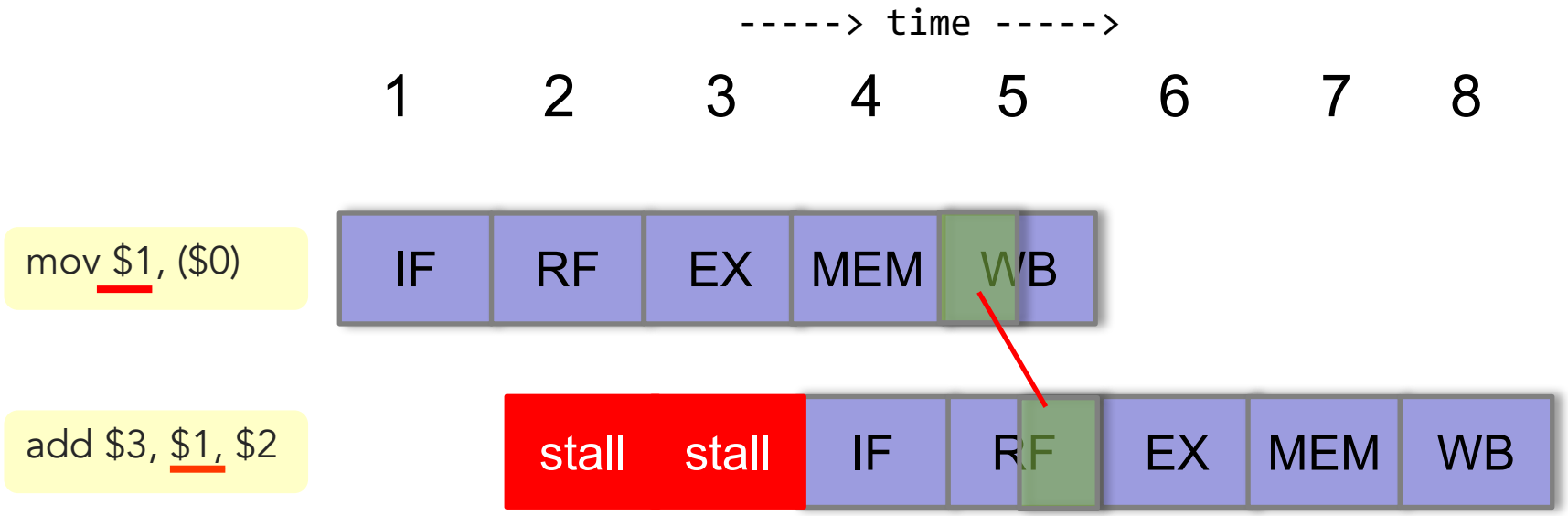


Waiting for Data



Re-arranging the register file so that WB writes in the first half of the cycle and the RF reads in the second half allows to save one stall

Waiting for Data



Re-arranging the register file so that WB writes in the first half of the cycle and the RF reads in the second half allows to save one stall

Handout

Look at your handout

- First (completed) example is the one from the previous slide
- Have a go at the remaining examples
- **When do you need to stall?** When any of the operands of an instruction depend upon the result of an operation preceding it.

Manchester Metropolitan University

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
add $1 $2 $3	I	R	E	M	W													
add $2 $1 $4		stall	stall	I	R	E	M	W										

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
add $1 $2 $3	I	R	E	M	W													
add $2 $3 $4		I	R	E	M	W												

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
add $1 $2 $3	I	R	E	M	W													
add $2 $1 $4		stall	stall	I	R	E	M	W										
sub $5 $4 $3					I	R	E	M	W									

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
add $1 $2 $3	I	R	E	M	W													
sub $5 $4 $3		I	R	E	M	W												
add $2 $1 $4			stall	I	R	E	M	W										

Last example technique is known as *instruction percolation* – the independent instruction "percolates" to an earlier execution point → helps minimize stalls

What to do with Dependencies

- Some hardware can detect dependencies
  - Default action is to *stall* – instruction is paused until previous instruction completes
- Some types of dependencies can be handled more gracefully than this at the hardware level
  - Register renaming: renaming of registers to remove certain type of dependencies
  - Data forwarding: data forwarded to appropriate unit without storing in register

Hazards

- There are three kinds of **data dependencies**:
  - Read after write
  - Write after read
  - Write after write
- These become particularly important when considering optimisation (next week).

Read After Write (RAW)

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

```
add $3,$1,$2
…
add $4,$4,$3
```

- Second instruction here cannot proceed until first instruction has stored result in register 3.

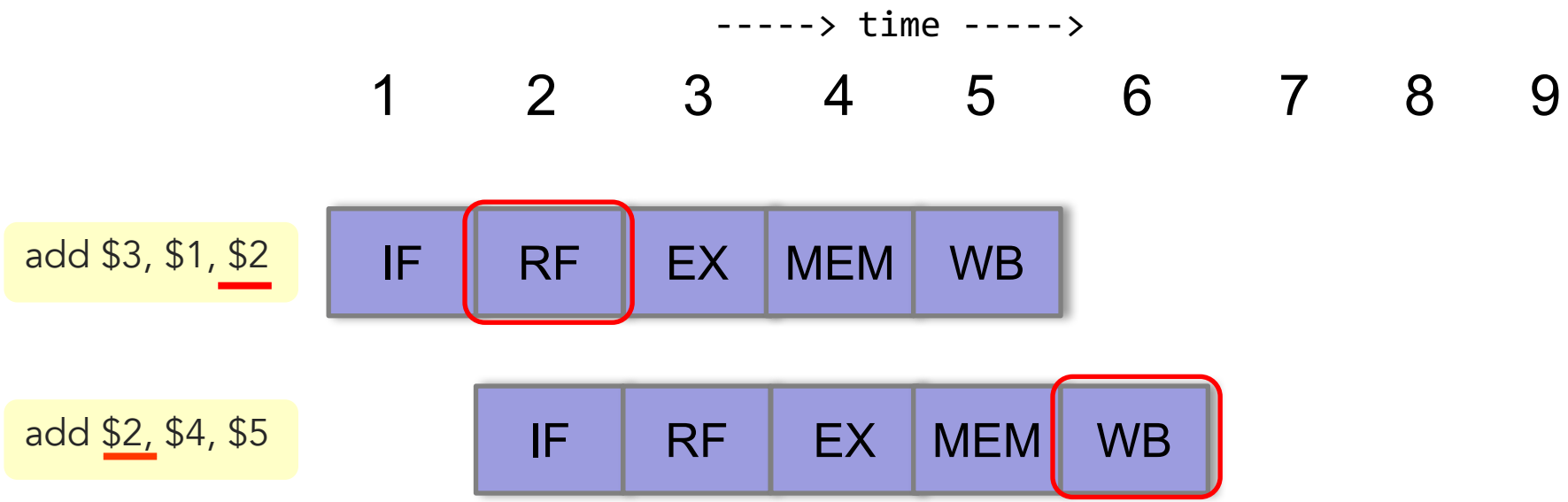
Write After Read (WAR)

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

```
add $3,$1,$2
…
add $2,$4,$5
```

- This isn't really a problem here: register write-back occurs last in the pipeline, so the store to register 2 should always occur after the read from register 2 *unless the instructions are reordered.*

Write After Read (WAR)



Write After Write (WAW)

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

```
add $3,$1,$2
…
add $3,$4,$5
```

- Again, this isn't really a problem. *If this ordering is maintained*, the value in register 3 should be the result of the second instruction.

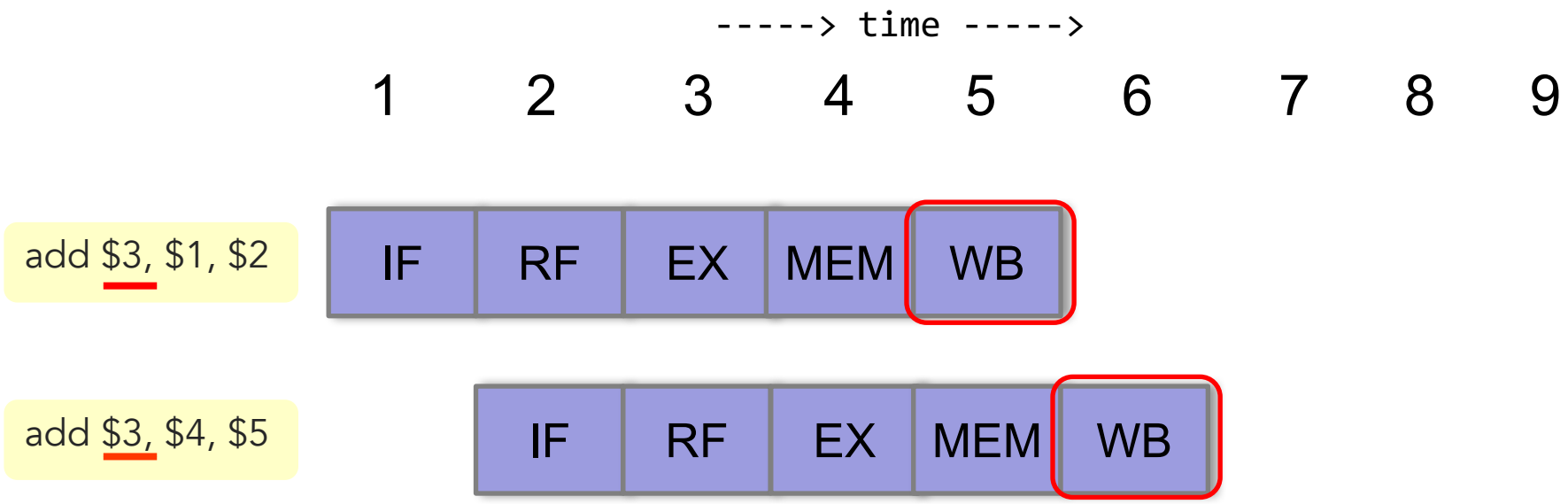
Write After Write (WAW)

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

```
add $3,$1,$2
…
add $3,$4,$5
```

- Again, this isn't really a problem. *If this ordering is maintained*, the value in register 3 should be the result of the second instruction.

Write After Write (WAW)



Specialist Processor Architectures

- Out-of-order execution (OOE) processors
- Superscalar processors
- Very long instruction word (VLIW) processors

OOE Processors

- Instruction order governed by availability of data rather than original order in program:
  - Instruction fetch
  - Dispatch to queue
  - **When operands are available, instruction is executed**
  - Result stored

OOE Processor Compilers

- OOE processors can avoid idleness while waiting for data
- Implementation is at the hardware level
  - *Is it worth worrying about data dependencies at the compiler level?*

Superscalar Processors

- Superscalar processors contain two (or more) copies of the functional units in the pipeline
- This allows independent operations to execute with more parallelism:

```
                    -----> time ----->
       1   2   3   4   5   6   7   8   9   10  11  12
Inst1  IF  ID  EX  MEM WB
Inst2  IF  ID  EX  MEM WB
Inst3      IF  ID  EX  MEM WB
Inst4      IF  ID  EX  MEM WB  MEM WB
Inst5          IF  ID  EX  MEM WB  MEM WB
Inst6          IF  ID  EX  MEM WB  EX  MEM WB
Inst7              IF  ID  EX  MEM WB  EX  MEM WB
```

Superscalar (cont)

- Exploits the potential of *instruction level parallelism* (ILP)
  - Only possible for data-independent instructions
- Instructions are initiated for execution in parallel based on the availability of operand data, rather than the original program sequence
  - Upon completion, instruction results are re-sequenced in the original order

VLIW Processors

- Fixed number of operations are formatted as one instruction in a *bundle*
- Unlike OOE or superscalar processors, decision as to what operations are performed in parallel left to the compiler, rather than hardware
- Simplifies hardware
  - no dependence checking
  - no out-of-order execution

VLIW Processor Compilers

- Must provide support to increase ILP
  - data hazards
    - no data hazards among instructions
  - structural hazards
    - no 2 ops to same functional unit
    - no 2 ops to same memory bank
  - control hazards
    - static branch prediction
  - hiding latencies
    - data prefetching

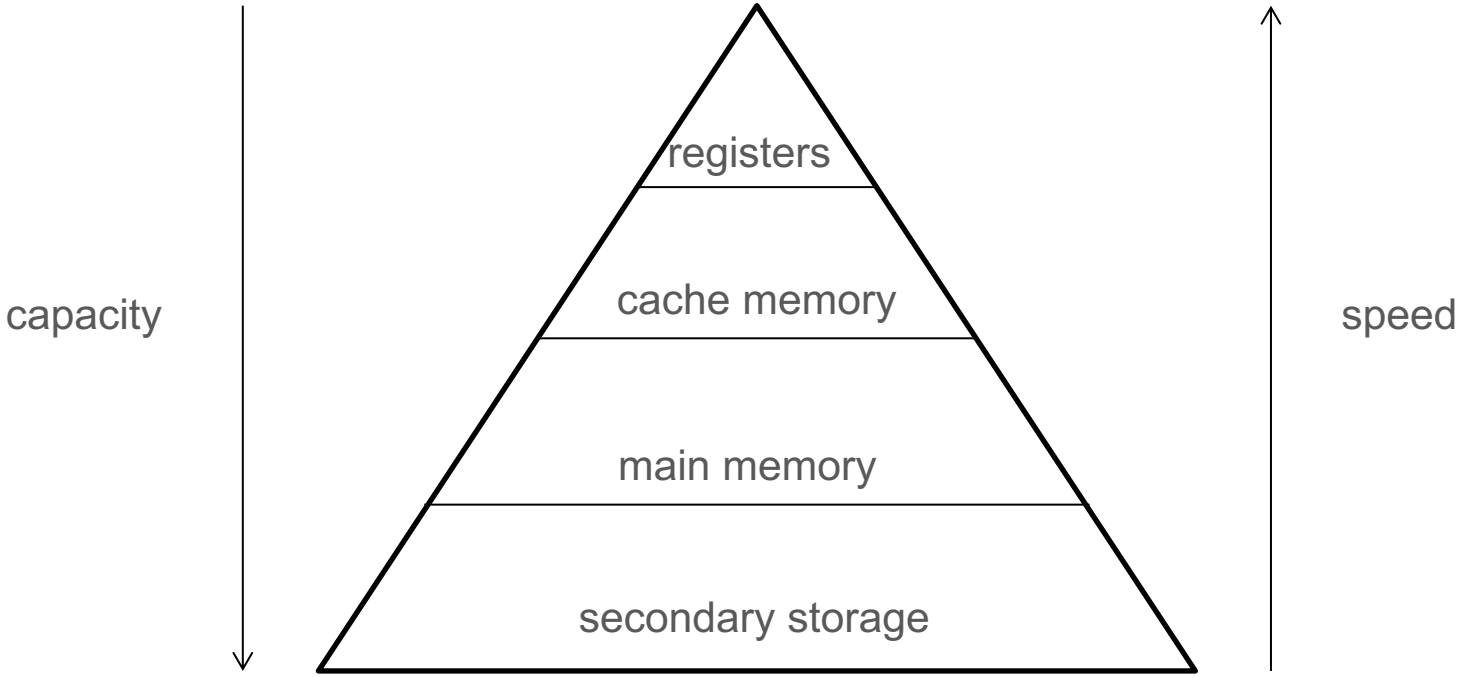
Assembly Code

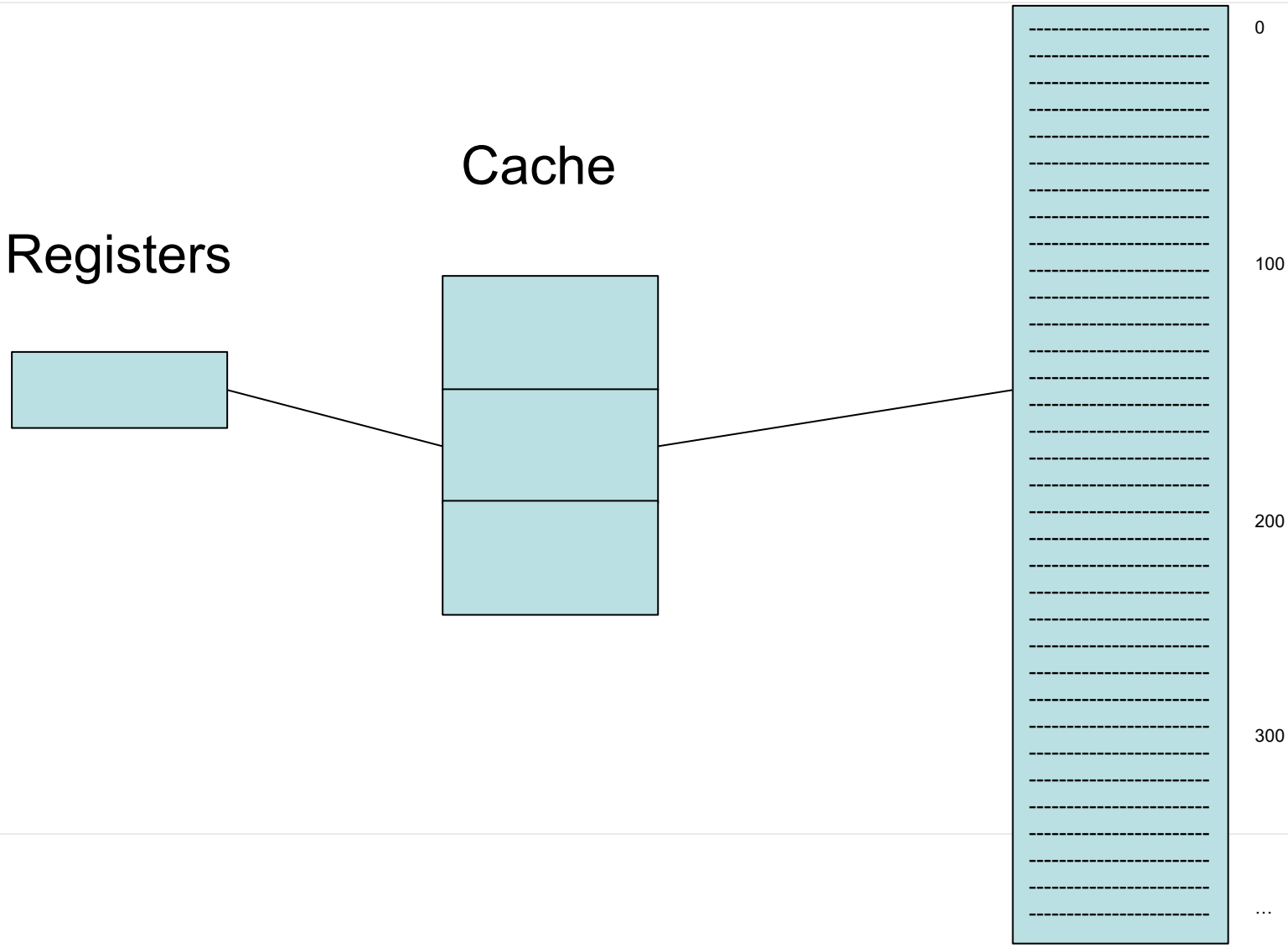
```
load r0 mem[7]
loop:
 r1 = r0 - 2
 j_zero r1 done
 r0 = r0 + 1
 jump loop
done:
```

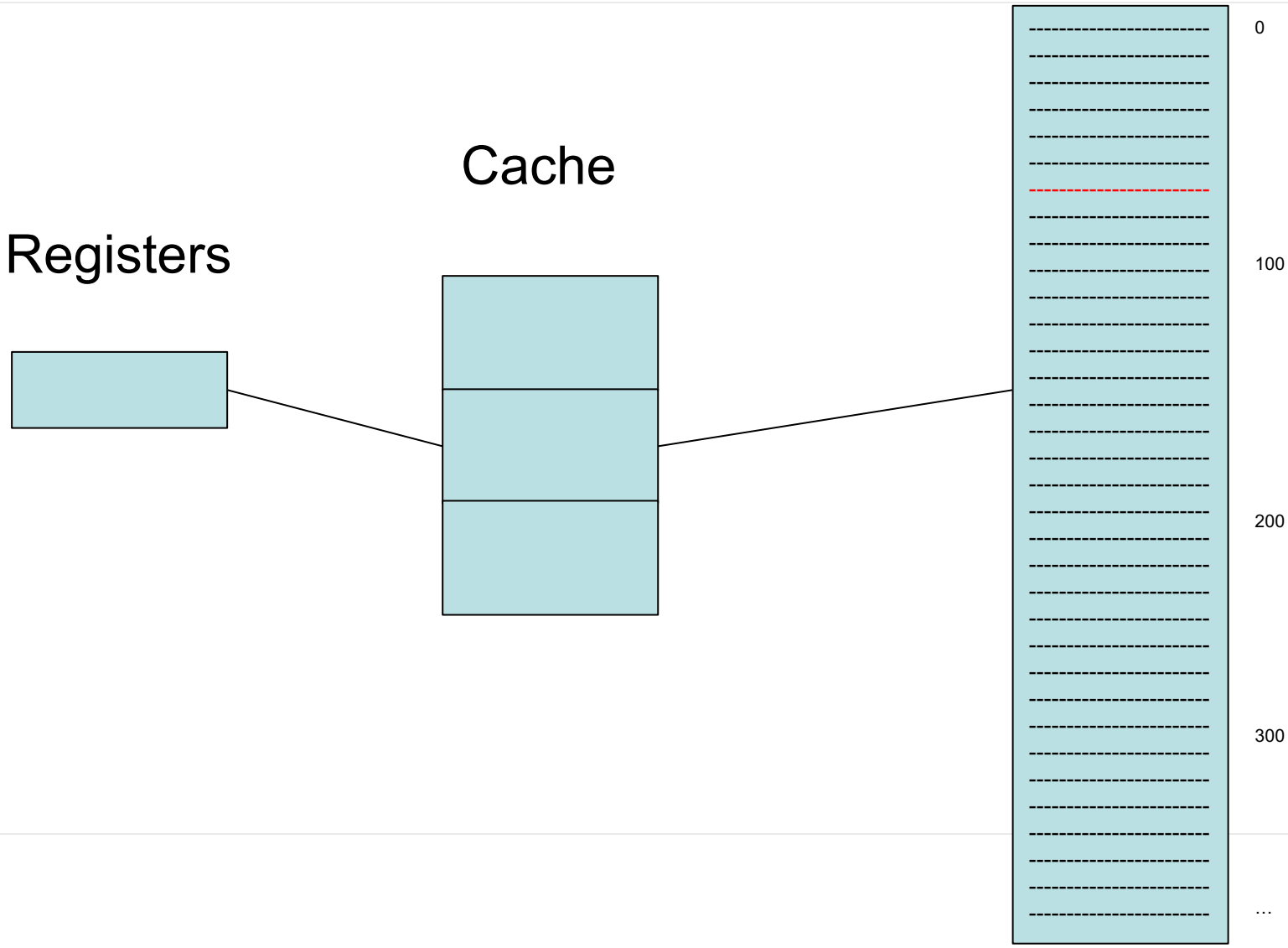
Further Hardware Considerations

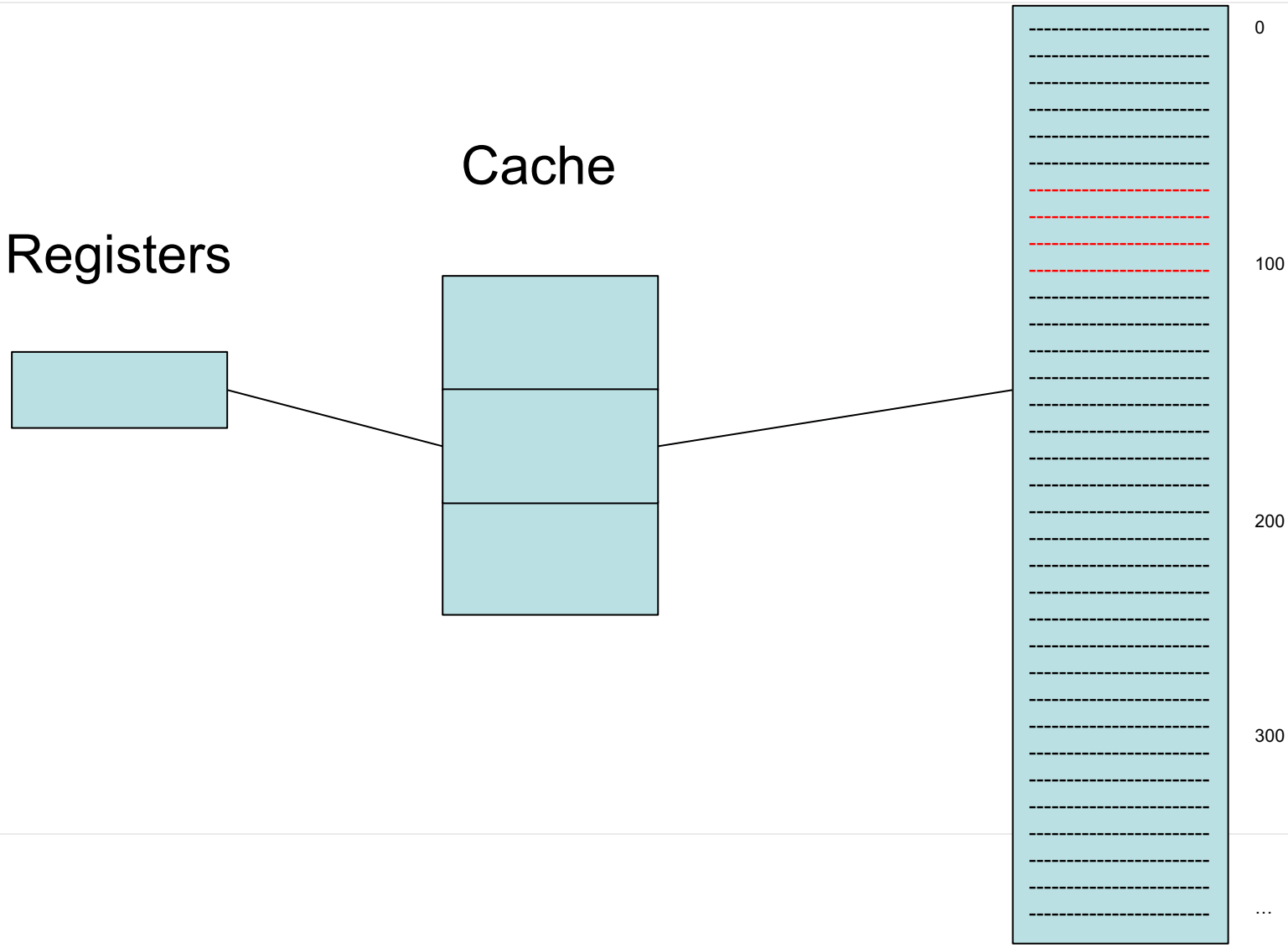
- Last week discussed the problems of *register allocation* and *register assignment*
- If there are insufficient registers, need to temporarily store results back into memory, then retrieve them again
  - "Register spill"
  - Degrades performance – memory access is SLOW
- Use cache to avoid long delays to main memory

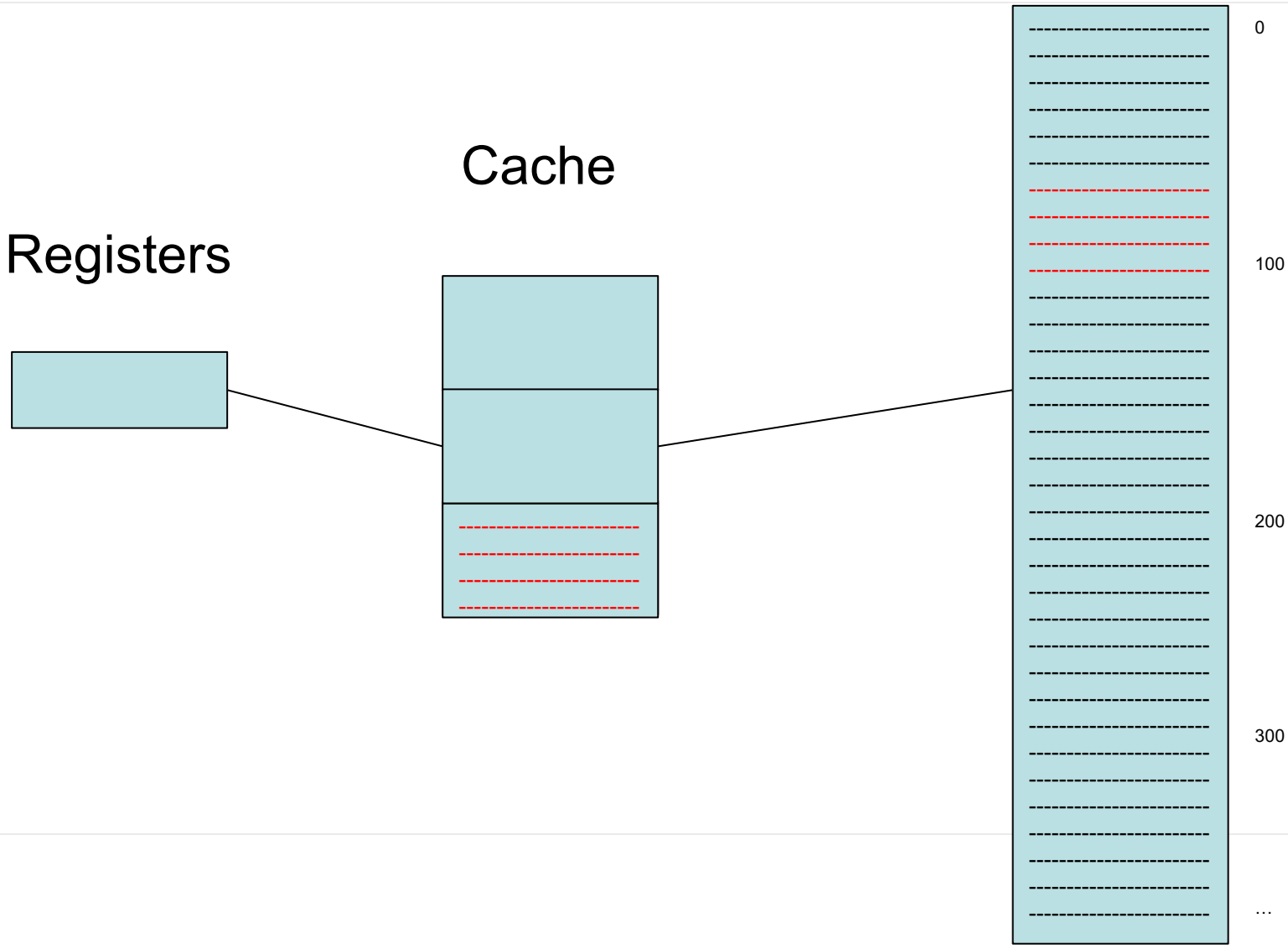
Recall: Memory Hierarchy




Main
Memory
Cache
Registers
0
100
200
300
…


Main
Memory
Cache
Registers
0
100
200
300
…


Main
Memory
Cache
Registers
0
100
200
300
…


Main
Memory
Cache
Registers
0
100
200
300
…

Spatial and Temporal Locality

- Temporal locality
  - If a memory location has been accessed (read/written) recently, it's likely to be accessed again in the near future
- Spatial locality
  - Cache is divided into "lines"
  - When memory is read, not just a single variable is read, but a whole block (to fill a line)
  - When a memory location has been accessed recently, then it is likely that nearby memory locations will be accessed in the near future

Memory Structure: Compiler Implications

- Use registers as much as possible
- If more space needed, use cache
- If still more memory is needed, try to minimize number of main memory accesses

We will look at *how* to do this in next week's lecture

What Compilers *Can* Do

- Compilers know which resources are available and how long instructions take
- Compilers schedule the instructions
- Compilers try to minimize **stalls**
- Compilers try to keep all resources busy (increase parallelism)
- Compilers can perform branch prediction
- Compilers can implement out-of-order execution

Summary

- Why consider processor architectures when building compilers?
- Pipelining:
  - Definition
  - Data dependencies
- *Out-of-order, superscalar*, and *VLIW* processor architectures
- Memory hierarchy implications for compilers

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Context-free grammars~~
- ~~Top-down and bottom-up parsing~~
- ~~Context-sensitive analysis~~
- ~~Intermediate representation~~
- ~~Machine-independent optimisation~~
- ~~Code generation~~
- ~~Hardware architectures~~
- **Machine-dependent optimisation**
- Review