COMS12200 Introduction to Computer Architecture

Dr. Cian O'Donnell

cian.odonnell@bristol.ac.uk

Topic 2: Intro to memory

1. Data, control and instructions

Summary from last lecture

- How data and control information get processed.
- What control information looks like.
- An introduction to instructions.
- How instructions are encoded as op-codes.
- How op-codes can be decoded.

Topics

1. Data, Control and Instructions
2. Memory
3. Execution cycle
4. Processor control flow
5. Machine types
6. State machines and decoding
7. Memory paradigms

Topics

1. Data, Control and Instructions
2. Memory
3. Execution cycle
4. Processor control flow
5. Machine types
6. State machines and decoding
7. Memory paradigms

2. Intro to memory

Overview

- Memory as a place to store data and instructions
- What is the memory hierarchy?
- Memory addressing

Memory

- Memory is simply a place to store information.
- Memories allow two basic operations:
  `Write`, which allows information to be inserted
  `Read`, which allows information to be extracted

Memory organisation

- Each piece of information in memory is assigned to a unique address.
- To access or update information, we need to specify this address to a memory, then our information can be returned or changed.
- Addresses are specified as indexes.

Memory addresses

Example of addresses and stored values

Address	Value
0	1
1	82
2	291
3	271
4	22
5	89
6	427

Which values are data and which are instructions?

Instructions in memory

Values can be op-codes, for example

Address	Value (op-code)	
0	1	'ADD'
1	2	'SUB'
2	1	'ADD'
3	4	'MUL'
4	1	'ADD'
5	3	'DIV'
6	427	?

Instructions and data

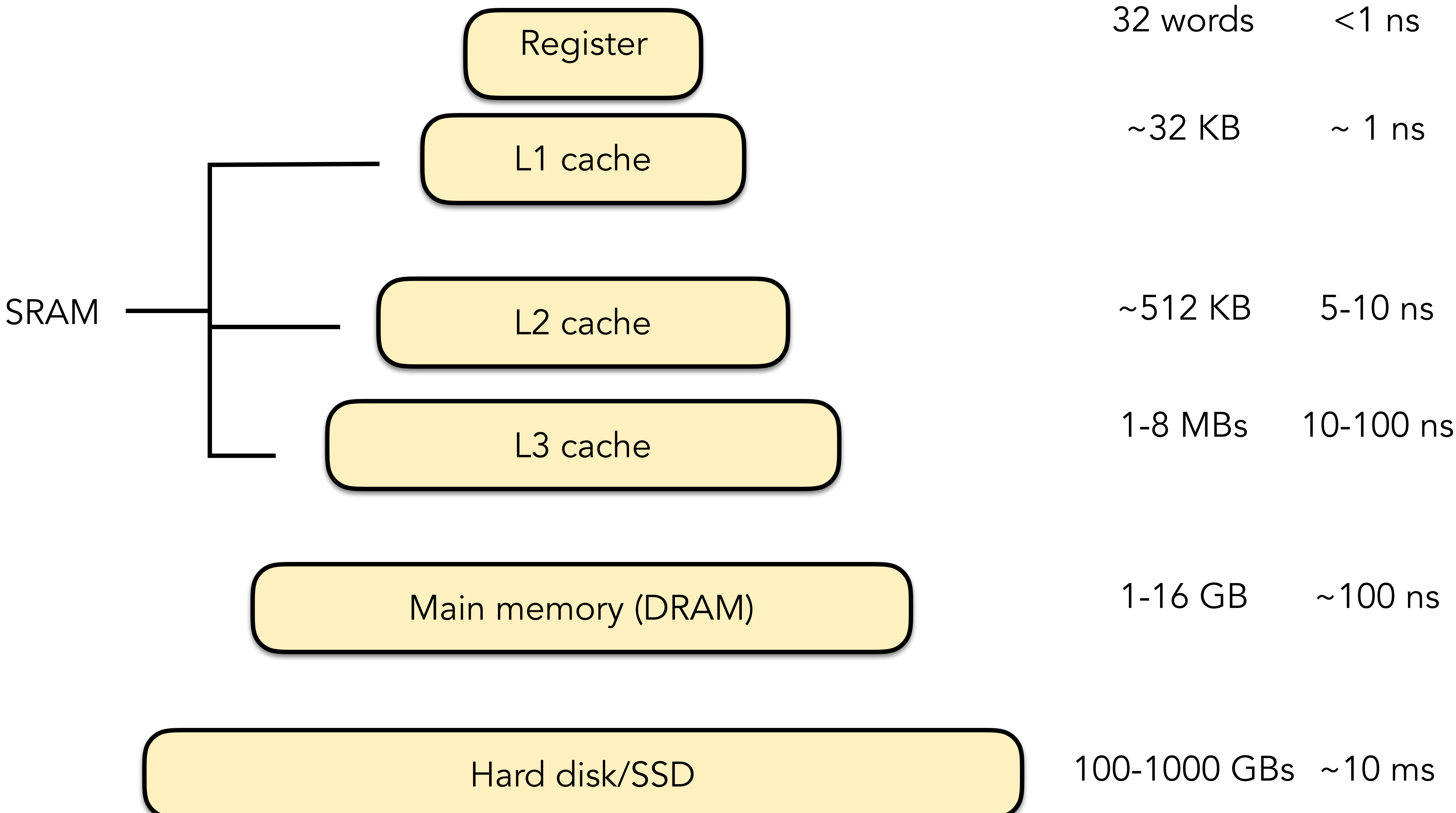
- A single memory location can combine both an instruction and some data.
- This can be useful for e.g. constant-based operations
- Consider `ADD1`
- Or something that loads a constant e.g. `MOVE2`

Instructions and data

- Example: `ADD1`
- `ADD` → '1'
- 1 → '1'
- So `ADD1` could be expressed as '11'

…it all depends how it is interpreted.

Memory hierarchy



Memory addressing

a.k.a. Addressing modes

1. Immediate addressing
2. Direct addressing
3. Memory-indirect addressing
4. Register-indirect addressing
5. Indexed addressing

What is addressing?

- When we want to access memory (as opposed to registers), we need to specify which memory address to use
  - e.g. `MEM[10]` — access memory address 10
- Ideally, we could directly specify a memory address every time, but this is not always possible.
- Sometimes, we would actually like to specify a sequence of addresses.
- Therefore, we have invented many different ways to specify a memory address.

1.Immediate addressing

- Immediate addressing is when data is supplied in an instruction — there is no real memory address, and all information is embedded in the instruction and data is immediately available.
- e.g. `r1 ← 42`
- Very fast and simple — the simplest.

Example:

MOVI	42
0 1 0 1	1 0 1 0 1 0

1.Immediate addressing

Pros	Cons
All information embedded in instruction (predictable)	Lack of flexibility
Makes it very fast	Must be inserted statically
Easy to understand	Limited range (limited by permitted number of operand bits in opcode)
Good for optimisers to analyse	

2.Direct addressing

- What about an instruction like

  `MEM[10] ← 42`

- How is this instruction actually formulated?
  - Operation | Operand1 | Operand 2
  - e.g. 6 | 10 | 42
- This is called Direct addressing
- The exact memory address used is embedded in the instruction

Example:

MOVM	MEM[10]	42
0 1 1 0	0 0 1 0 1 0	1 0 1 0 1 0

2.Direct addressing

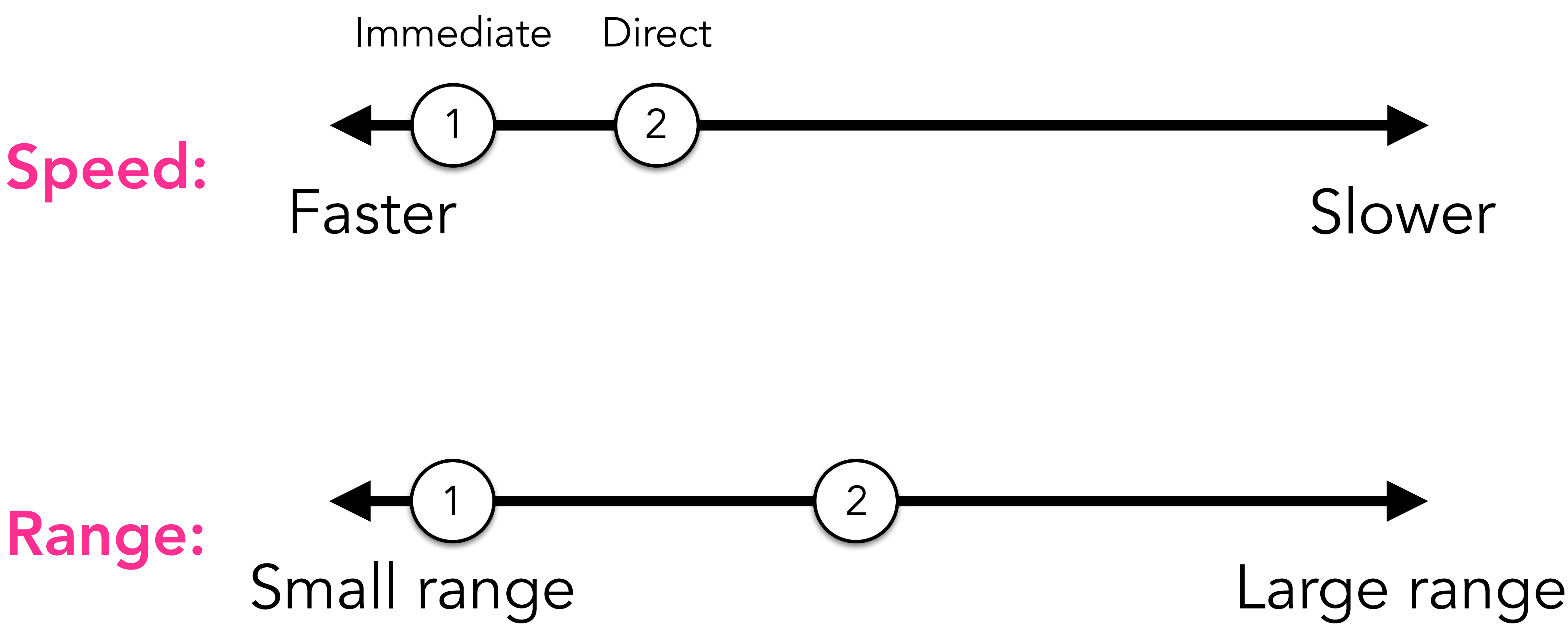
- Direct (a.k.a. absolute) addressing has the same pros and cons as immediate addressing.

- Pros:
    - All information embedded in instruction
    - Easy to understand
    - Good for optimisers to analyse

2.Direct addressing

- Cons:
  - Lack of flexibility
  - Must be inserted statically
  - Slower than immediate addressing
  - Limited range
    e.g. 16 bits can specify only 2^{16}=65,536 unique addresses (corresponds to 64KB of memory)

2.Direct addressing



3.Memory-indirect addressing

- *Memory-indirect addressing* solves the problem of limited range by storing the address to be accessed in memory itself.

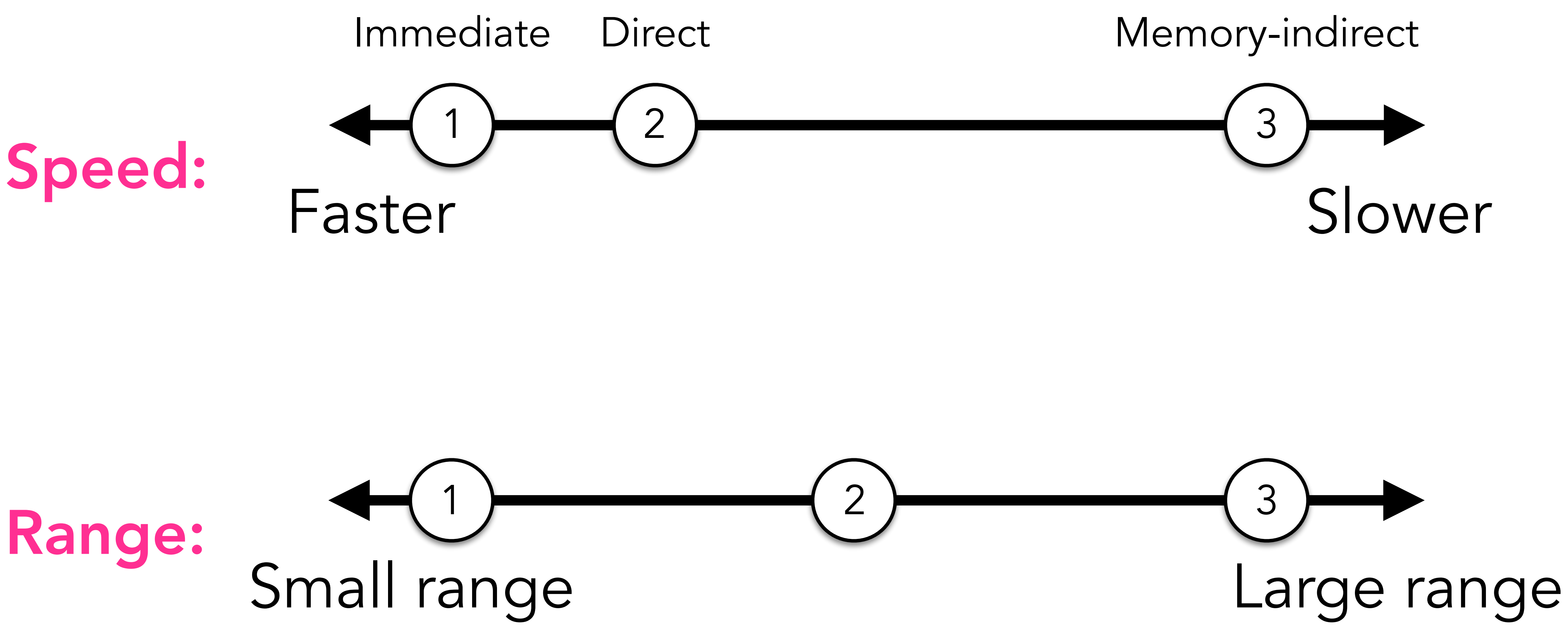
- e.g. `MEM[ MEM[ 42 ] ] ← r1`
  - Meaning: go and look at memory address 42 and fetch the value there.
  - That value is the address to *write* the value in `r1` to.

3.Memory-indirect addressing

- Plus points:
  - Larger range, e.g. 32 bits (corresponds to ~4GBs)
  - The source memory location for the address may be dynamically changed.

- Still has some drawbacks:
  - The first memory address is still statically compiled.
  - The range restriction now applies to the initial memory range.
  - Slower than direct addressing

3.Memory-indirect addressing



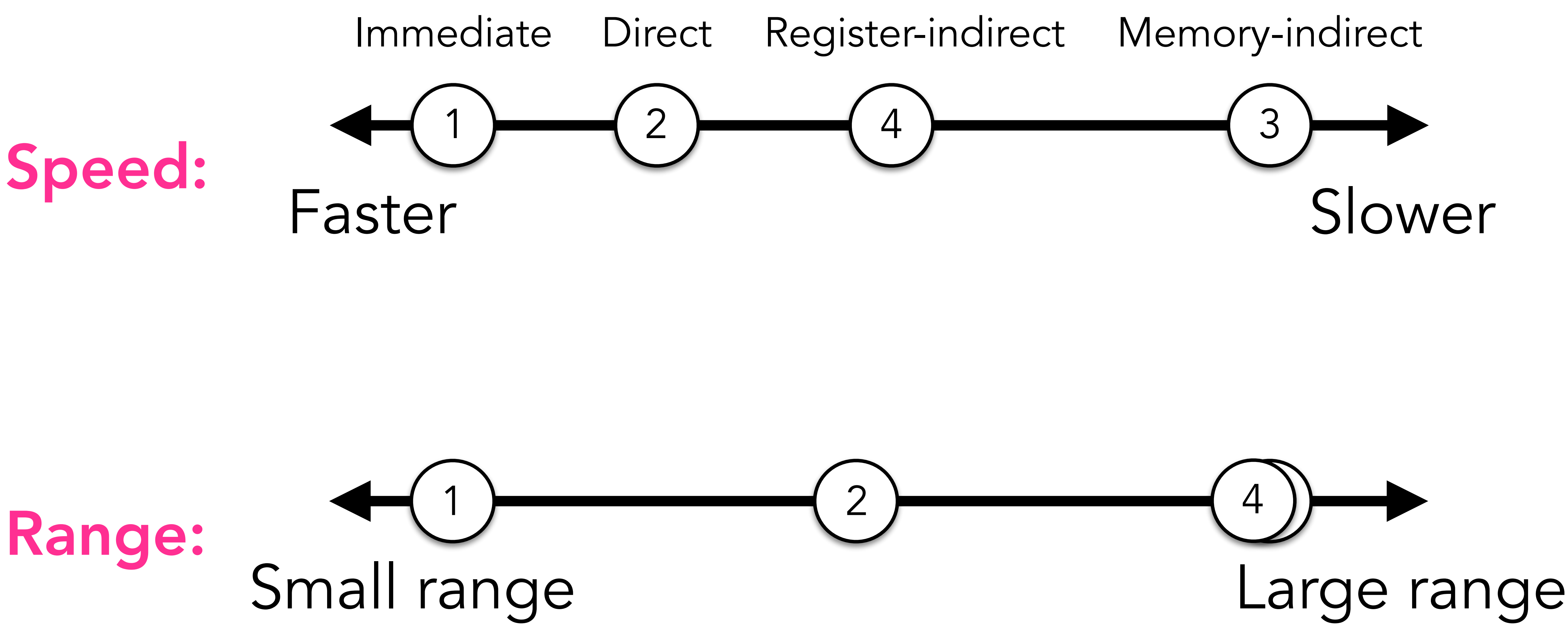
4.Register-indirect addressing

- Register-indirect provides more flexibility.
- Idea: use a register's value as the memory address
  e.g. `MEM[ r1 ] ← r1`

4.Register-indirect

- There are lots of advantages to register-indirect addressing:
  - The memory address can be dynamically computed.
  - The value does not need to be stored in the instruction, reducing code size
  - The register is internal to the processor — faster, more energy efficient.

4.Register-indirect



Pointers

- Indirect addressing allows native support of pointers, a key programming primitive.
- Accessing indirectly is equivalent to a de-referencing operation (e.g. `*p` in C)

5.Indexed addressing

- Sometimes it makes sense to define a base address and access memory based on this.
  - Useful for stacks, arrays, caches…
  - Indexed addressing extends indirect addressing to support this.
  - We have a base address and an offset.

5.Indexed addressing

- Normally the base and offset are both stored in registers, although this need not be the case.

- We gain instructions like
  `MEM [ r1 + r2 ] ← r3`

- `r1` is the base, `r2` is the offset

- Base and offset can be varied independently.

5.Indexed addressing

- Many implementations support the base + offset construct natively.

- Architectures often have a dedicated register to help, normally called either:
  - The stack pointer
  - Or the base register

5.Indexed addressing

- The stack / base registers may or may not be general purpose, depending on the architecture.
- The offset usually comes from an additional general purpose register.
- Example of indexed addressing based on an array

Example of indexed addressing

Example:

```
for i = 0:10
  x = some computation…
  a[i] = x
end
```

Registers / Memory

Before:

rb	r3	r5
17526	x=?	i=?

MEM[17533]
0

Get to i = 7: MEM [rb + r5] ← r3

Before inst:

rb	r3	r5
17526	x=?	7

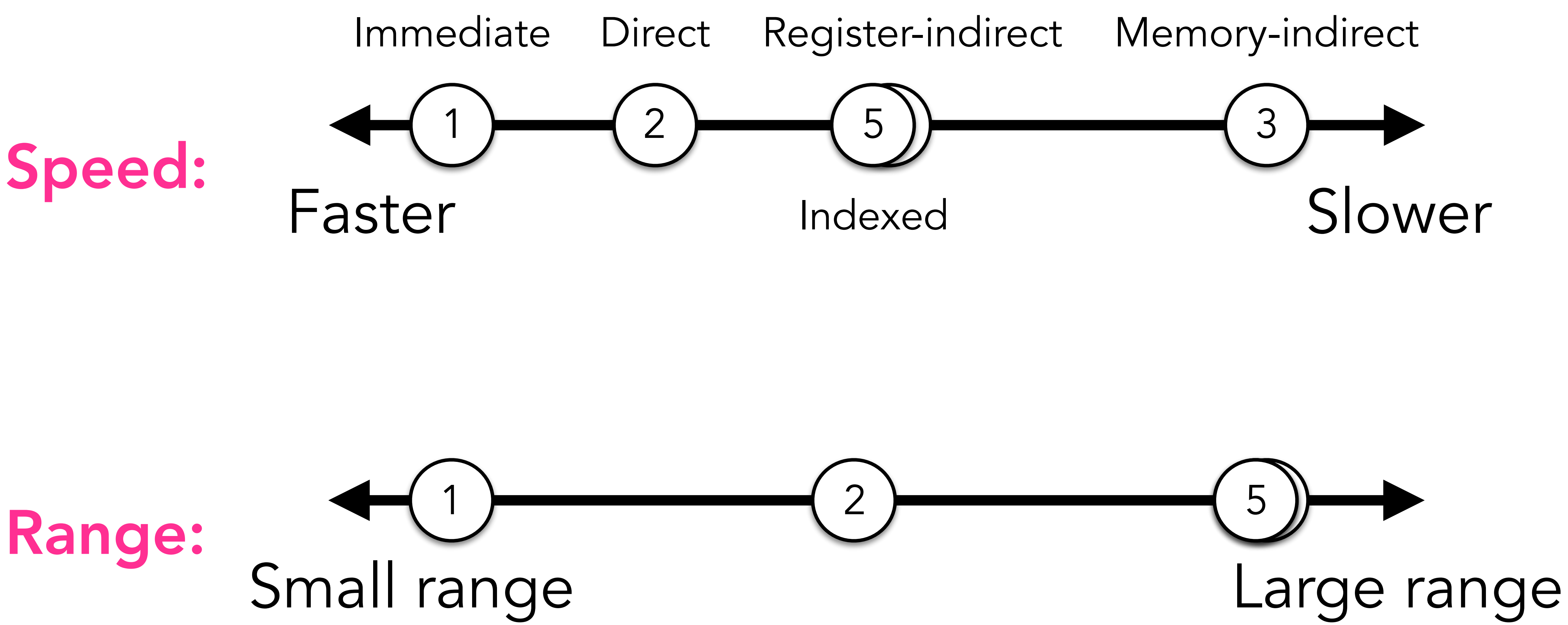
MEM[17533]
0

After:

rb	r3	r5
17526	25	8

MEM[17533]
25

5.Indexed addressing



Summary

- We have seen 5 methods of memory addressing.
- We have evaluated their use and efficiency.
- Some directly relate to programming primitives.
- There are more methods with varying complexity, but these are the most commonly used ones.