Compilers and computer architecture

10. Considering code generation

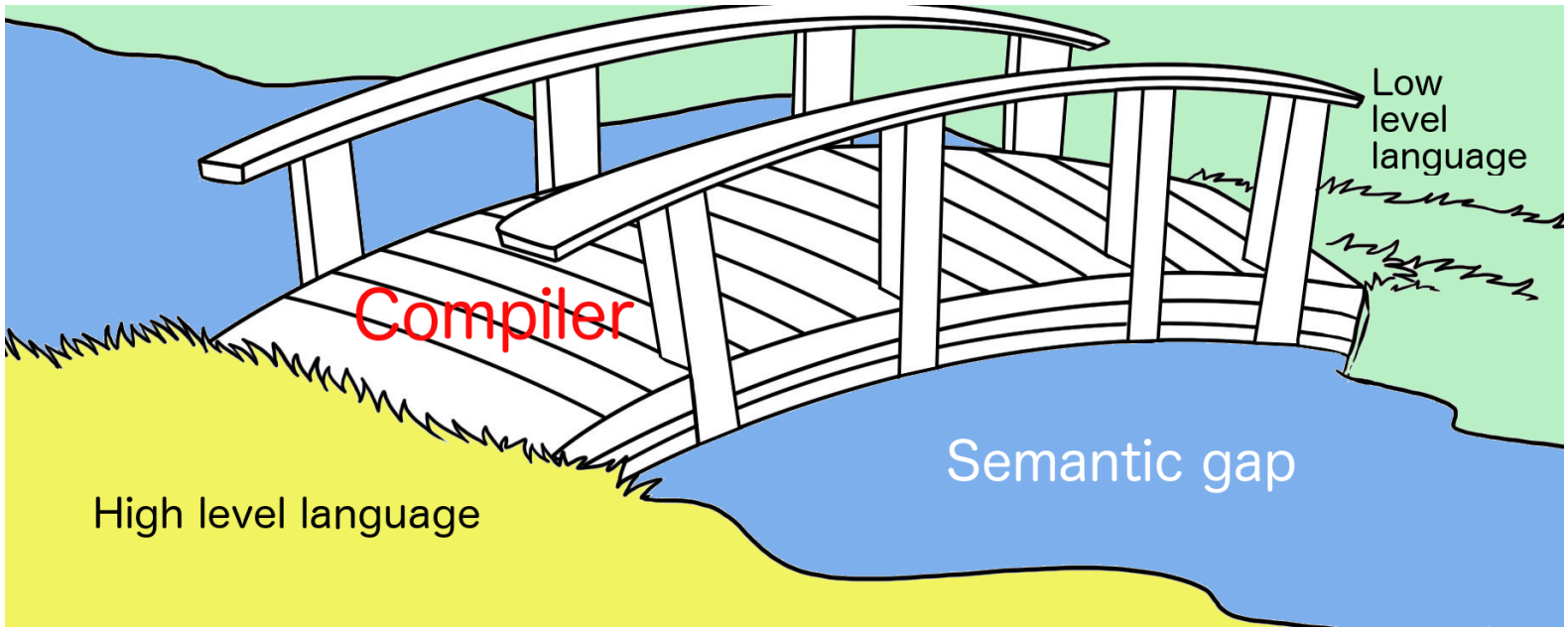
Hsi-Ming Ho

(slides by Niel de Beaudrap)

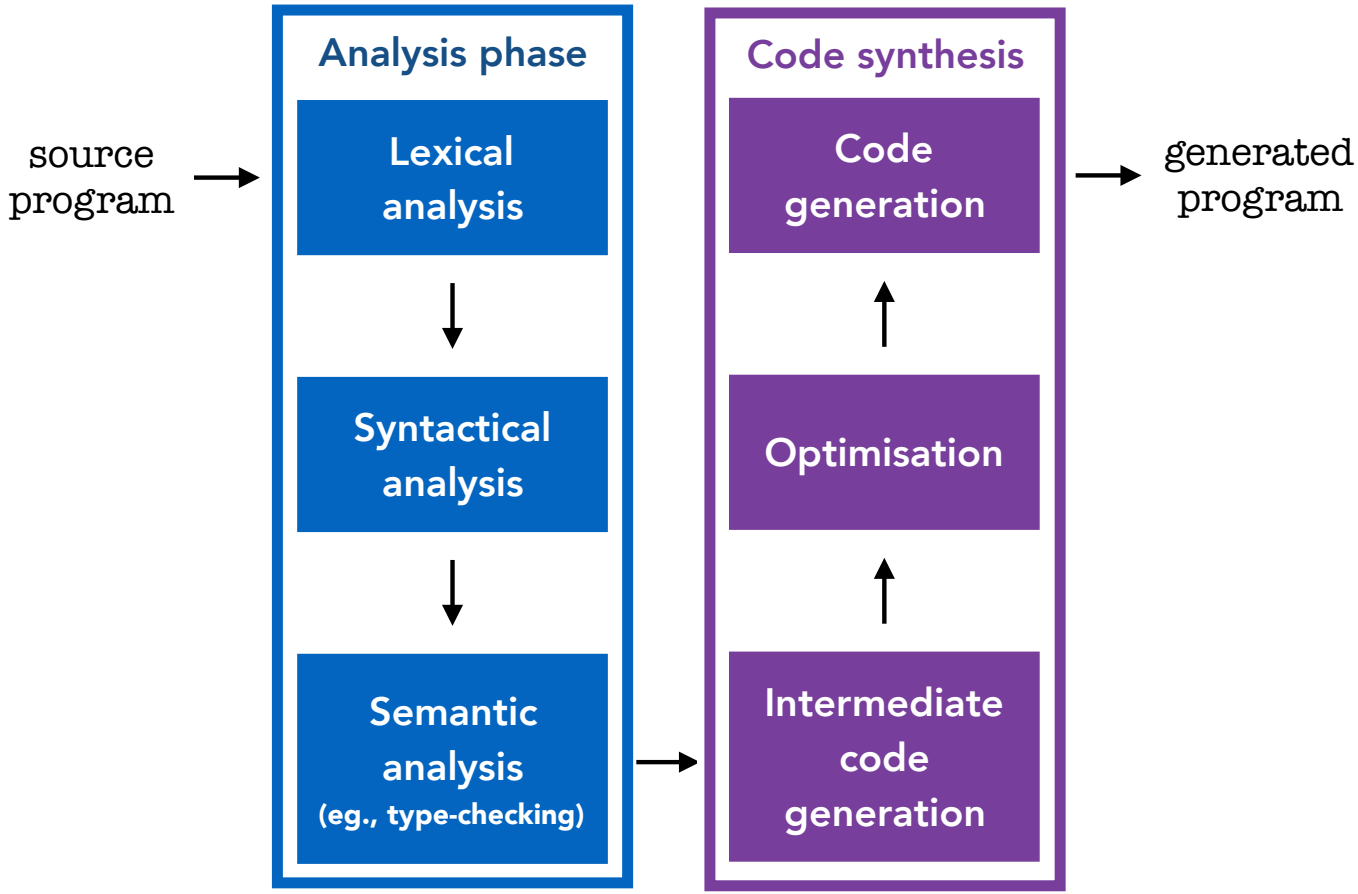
Email: **hsi-ming.ho@sussex.ac.uk**
Office hours: Online, by email appointment

Autumn 2022

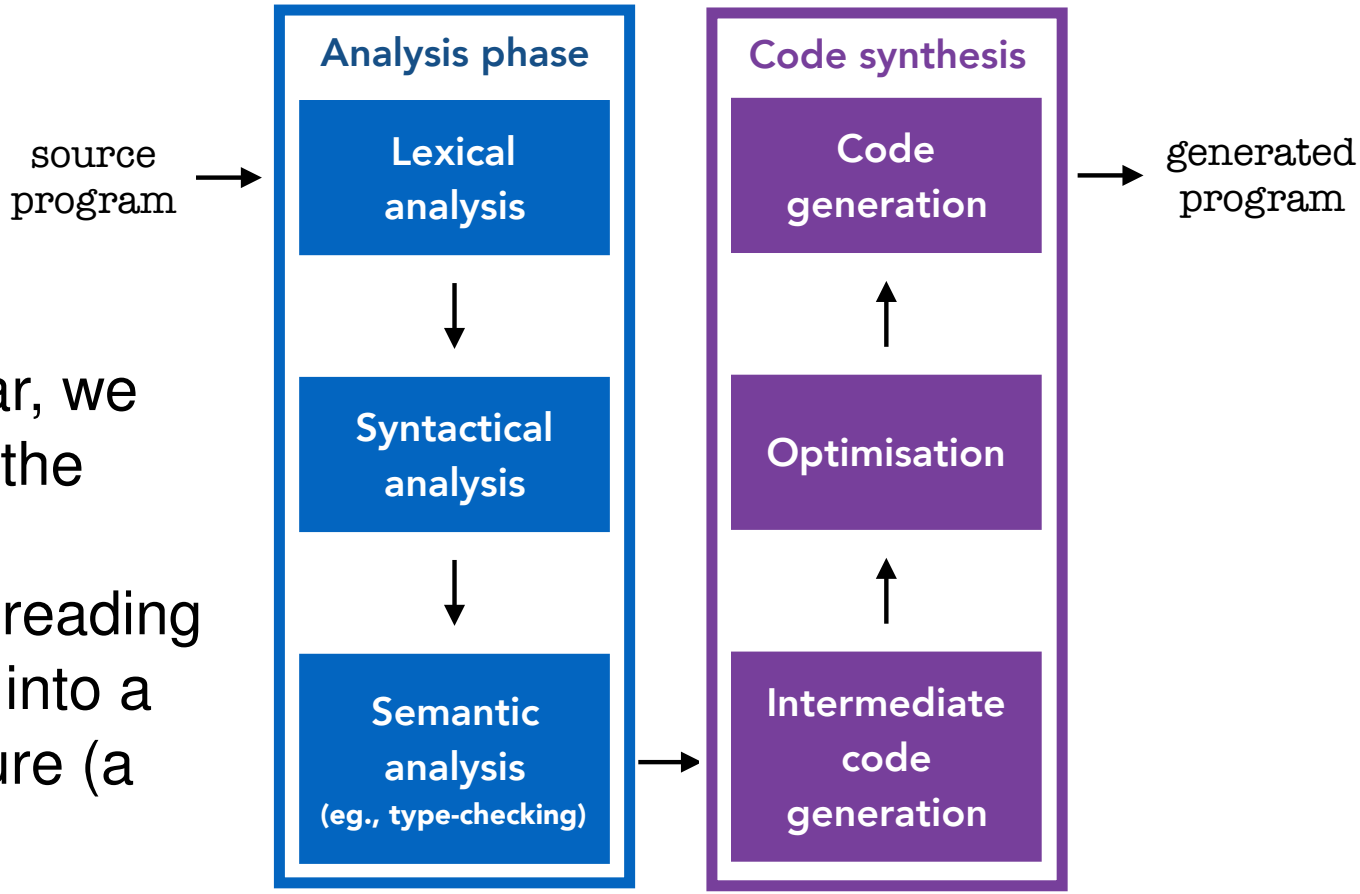
Recall the function of compilers



Where we find ourselves

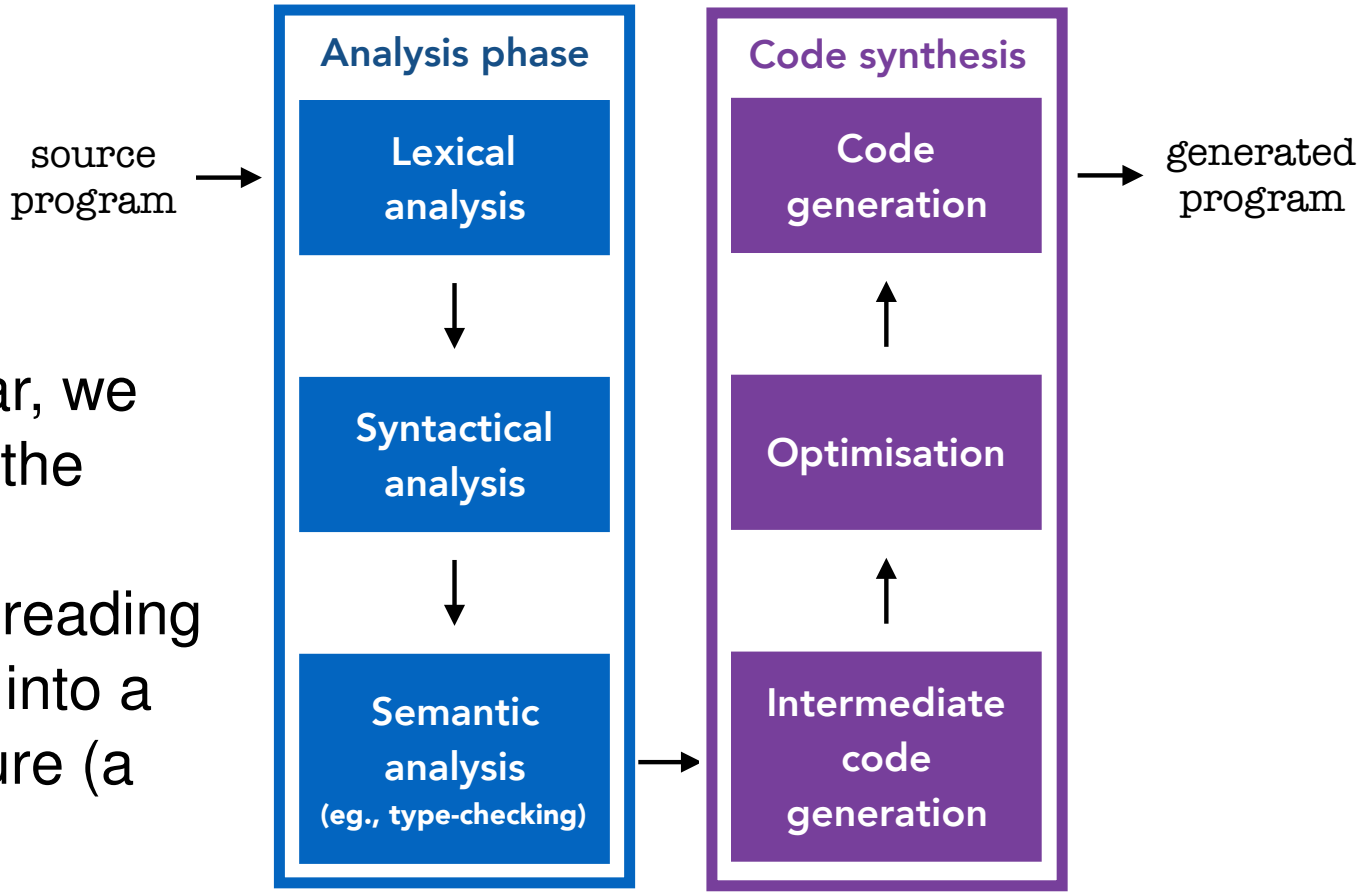


Where we find ourselves



In the course so far, we have studied only the analysis phase — the theory behind reading a source program into a useful data structure (a *syntax tree*).

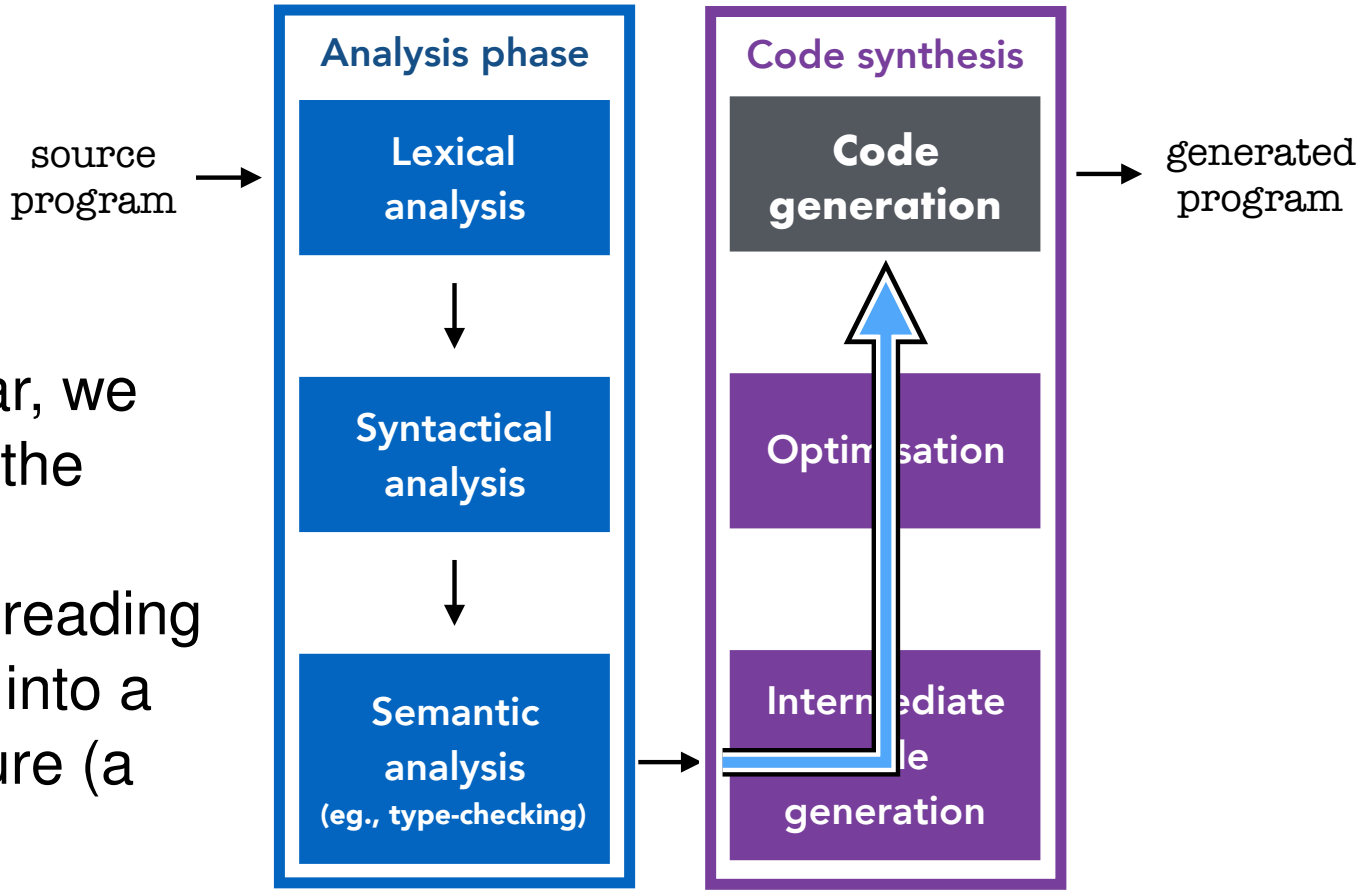
Where we find ourselves



In the course so far, we have studied only the analysis phase — the theory behind reading a source program into a useful data structure (a *syntax tree*).

We now **fast-forward** to the other end of the compiler —

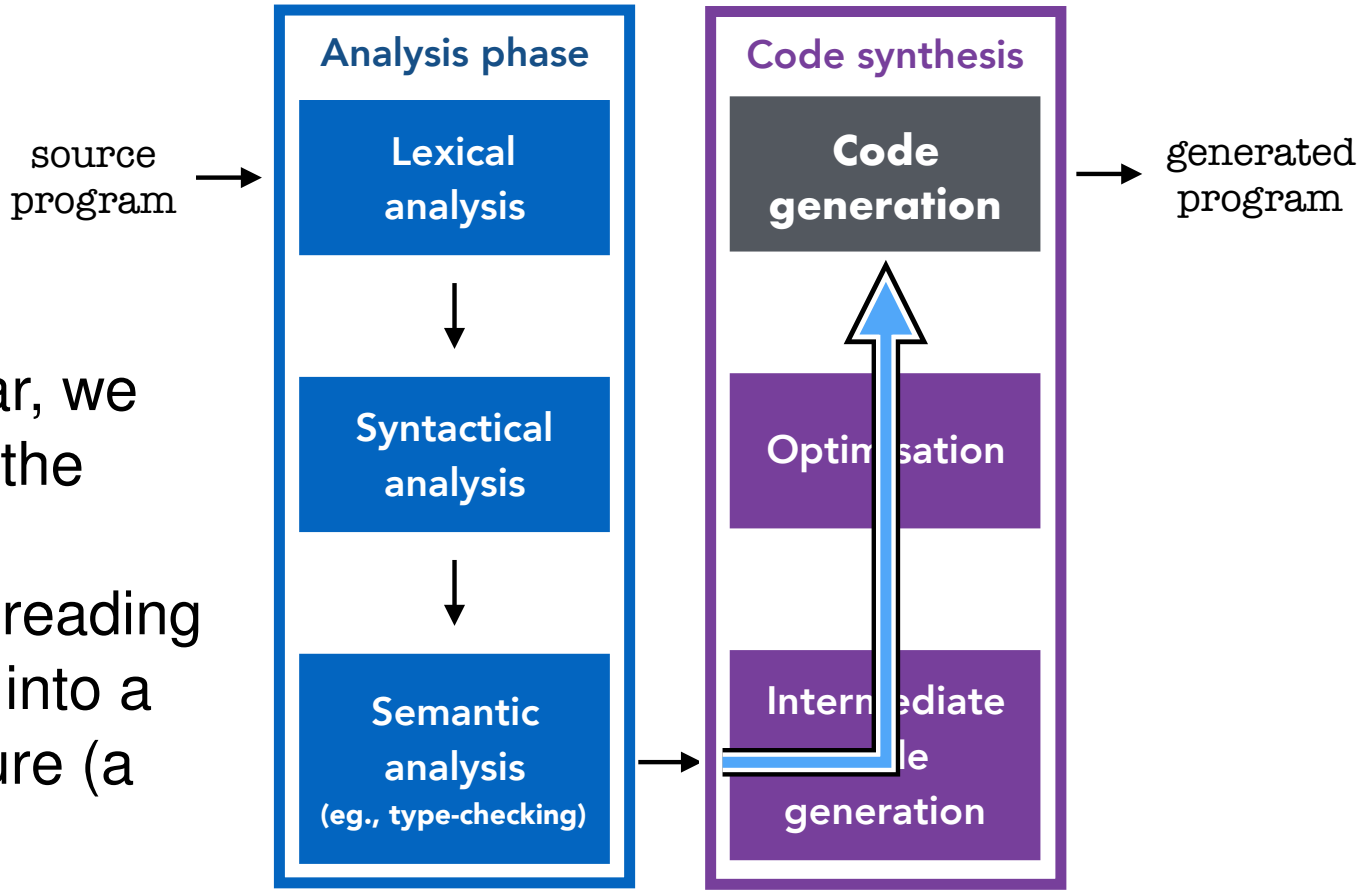
Where we find ourselves



In the course so far, we have studied only the analysis phase — the theory behind reading a source program into a useful data structure (a *syntax tree*).

We now **fast-forward** to the other end of the compiler — *code generation* — and consider how it must contend with the limitations of practical computer architectures.

Where we find ourselves



In the course so far, we have studied only the analysis phase — the theory behind reading a source program into a useful data structure (a *syntax tree*).

We now **fast-forward** to the other end of the compiler — *code generation* — and consider how it must contend with the limitations of practical computer architectures.

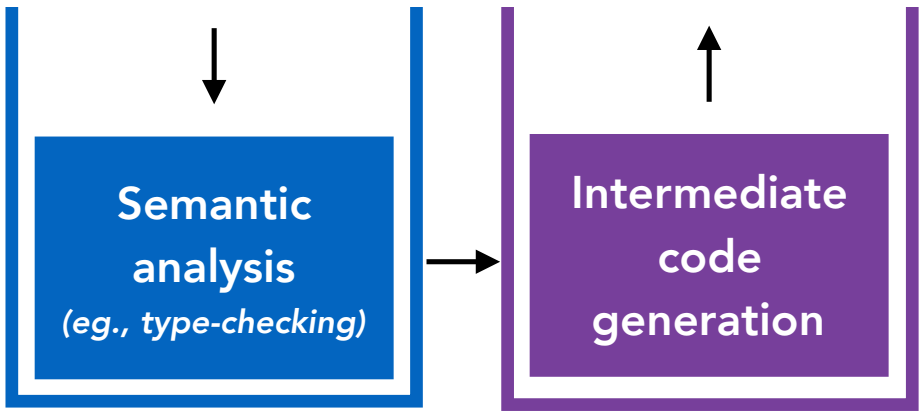
Today: an overview of the practicalities of code generation.

Wait wait wait

— what about intermediate code generation??

We spent all this time having the compiler workflow drummed into us — only to now skip several stages?

Question. What happened to this bit?

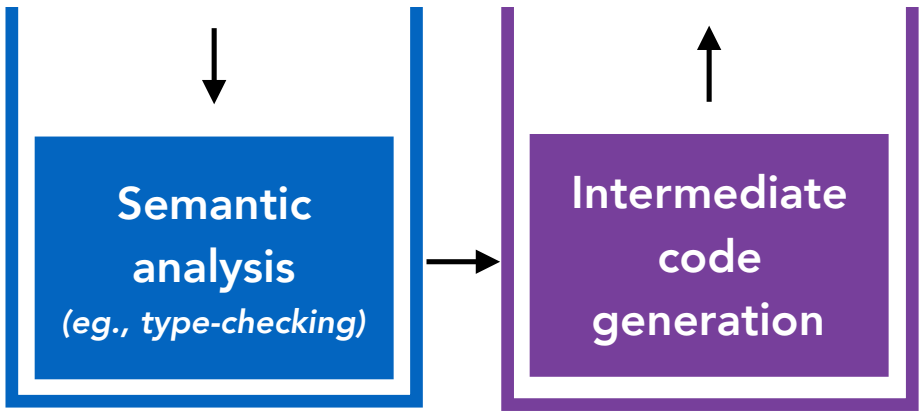


Wait wait wait

— what about intermediate code generation??

We spent all this time having the compiler workflow drummed into us — only to now skip several stages?

Question. What happened to this bit?



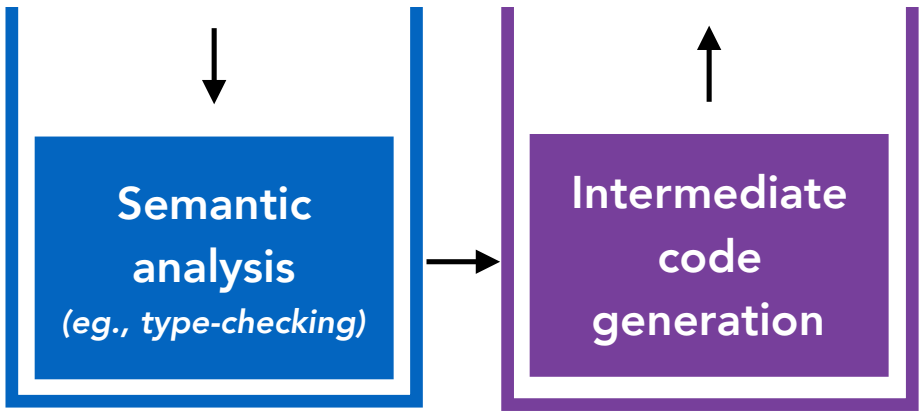
Answer: Intermediate code are often just assembly languages of hypothetical target architectures (e.g., stack machines). It is particularly useful when you target multiple (actual) architectures.

Wait wait wait

— what about intermediate code generation??

We spent all this time having the compiler workflow drummed into us — only to now skip several stages?

Question. What happened to this bit?



Answer: Intermediate code are often just assembly languages of hypothetical target architectures (e.g., stack machines). It is particularly useful when you target multiple (actual) architectures.

This will be a refinement, of the simpler story we will tell first.

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format



Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format
- **Machine code**
  — the format that a program takes,
  and which is **actually executed**
  on a machine

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format
- **Machine code**
  — the format that a program takes,
  and which is **actually executed**
  on a machine; produced by
  - an **assembler** — translates a program in assembly and produces a binary file

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format
- **Machine code**
  — the format that a program takes,
  and which is **actually executed**
  on a machine; produced by
  - an **assembler** — translates a program in assembly and produces a binary file
  - a **linker** — takes one or more binary files (including pre-compiled libraries) and produces an executable

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

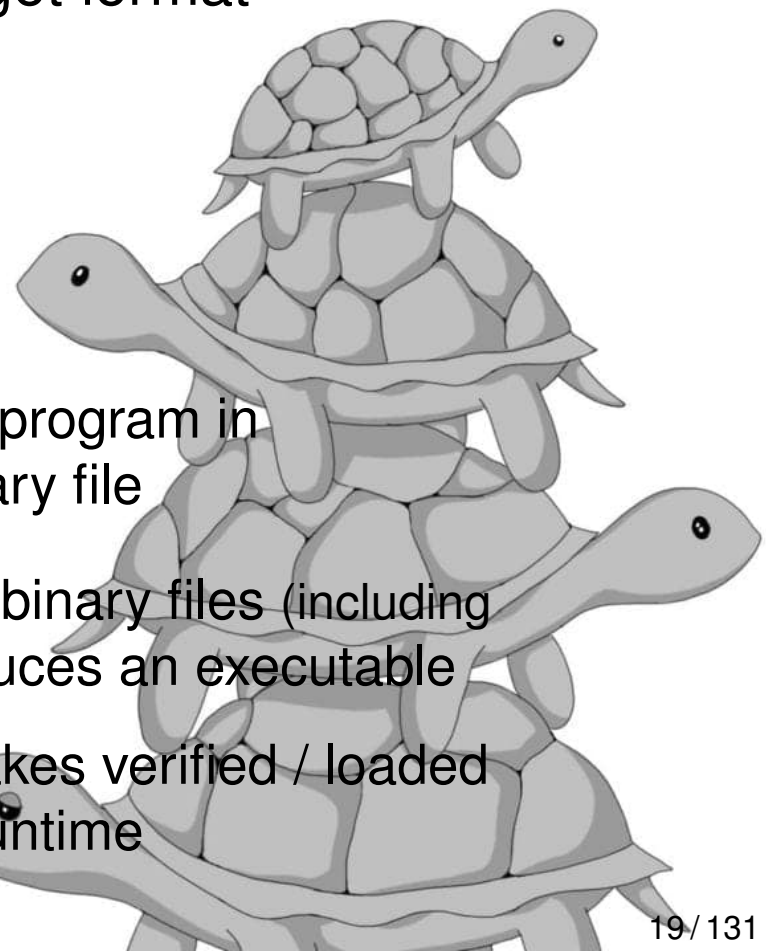
What does 'code generation' generate?

- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format
- **Machine code**
  — the format that a program takes,
  and which is **actually executed**
  on a machine; produced by
  - an **assembler** — translates a program in assembly and produces a binary file
  - a **linker** — takes one or more binary files (including pre-compiled libraries) and produces an executable
  - or: an **execution engine** — takes verified / loaded bytecode and interprets it at runtime

Well, I say that we're skipping intermediate code
— let's split hairs for a little while

What does 'code generation' generate?

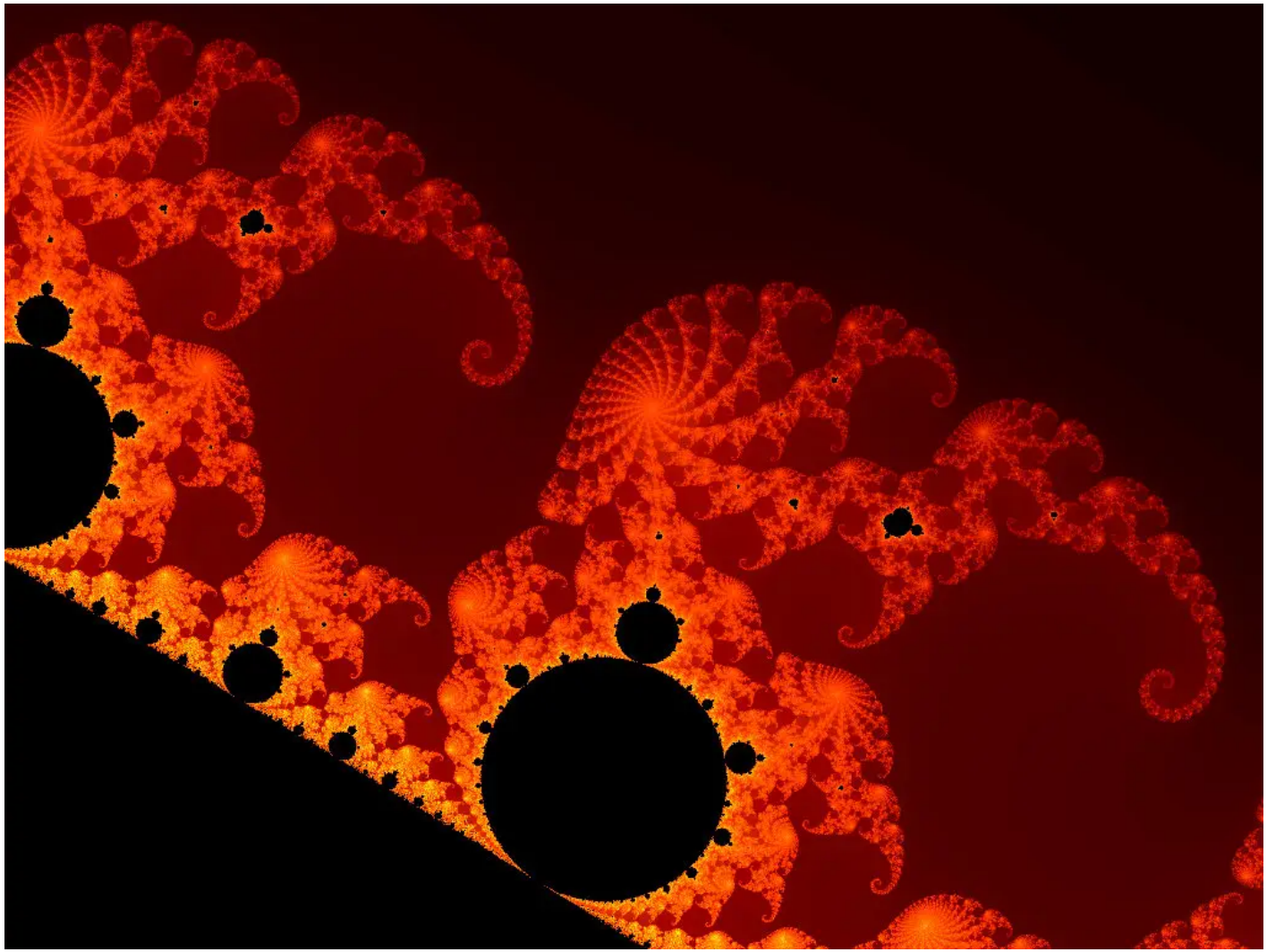
- **Assembly language** / **Byte code**
  — a low-level, *human readable* target format
- **Machine code**
  — the format that a program takes, and which is **actually executed** on a machine; produced by
  - an **assembler** — translates a program in assembly and produces a binary file
  - a **linker** — takes one or more binary files (including pre-compiled libraries) and produces an executable
  - or: an **execution engine** — takes verified / loaded bytecode and interprets it at runtime



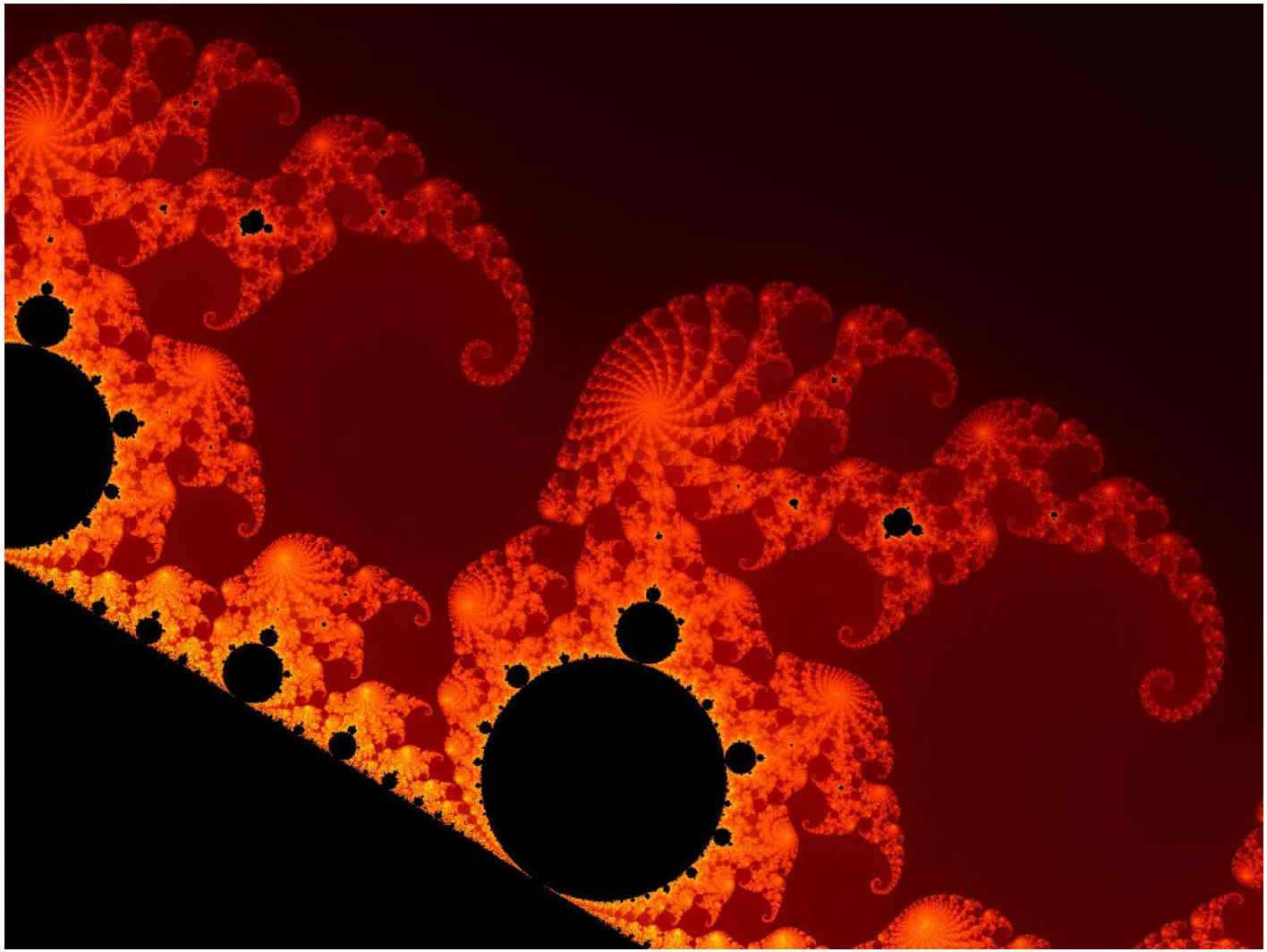
The theme of this course



The theme of this course



The theme of this course



When we set out to do programming about programming, this was always going to be the result.

Plan for the next two weeks

To get you up to speed with RISC-V for the coursework, we will look at:

- What the compiler is meant to produce (**assembly code**);

Plan for the next two weeks

To get you up to speed with RISC-V for the coursework, we will look at:

- What the compiler is meant to produce (**assembly code**);
- *Strategies* for realising computations with assembly (in particular, basic memory management);

Plan for the next two weeks

To get you up to speed with RISC-V for the coursework, we will look at:

- What the compiler is meant to produce (**assembly code**);
- *Strategies* for realising computations with assembly (in particular, basic memory management);
- How these strategies are affected by choice of target architecture:
  - **Stack machines** (unbounded stack);
  - **Register machines** (unbounded registers);
  - **Accumulator machines** (1 register, unbounded stack).

Why assembly language?

Even compiler writers stop at assembly language —

Why assembly language?

Even compiler writers stop at assembly language —

- The job of the **assembler** is pretty direct
  — **transliteration** rather than translation

Why assembly language?

Even compiler writers stop at assembly language —

- The job of the **assembler** is pretty direct

  — **transliteration** rather than translation

- The **linker** is an interesting program in its own right:
  takes modularised binaries and combines them
  into a self-contained executable
  - resolves memory addresses for library calls
  - in some cases, performs some final code generation
    (*e.g.*, some C++ linkers for ‘generics’)

Why assembly language?

Even compiler writers stop at assembly language —

- The job of the **assembler** is pretty direct

  — **transliteration** rather than translation

- The **linker** is an interesting program in its own right: takes modularised binaries and combines them into a self-contained executable
  - resolves memory addresses for library calls
  - in some cases, performs some final code generation (*e.g.*, some C++ linkers for ‘generics’)
- An **execution engine (virtual machine)**, on the other hand, takes assembly — well, **byte code**, actually — as input and generates machine code *on-the-fly*

Why assembly language?

Even compiler writers stop at assembly language —

- The job of the **assembler** is pretty direct

  — **transliteration** rather than translation

- The **linker** is an interesting program in its own right: takes modularised binaries and combines them into a self-contained executable
  - resolves memory addresses for library calls
  - in some cases, performs some final code generation (*e.g.*, some C++ linkers for ‘generics’)
- An **execution engine (virtual machine)**, on the other hand, takes assembly — well, **byte code**, actually — as input and generates machine code *on-the-fly*
  - **Question.** How is this different from interpreters?

You want to do machine code?

```
0010 0111 1011 1101 1111 1111 1110 0000 1010 1111
1011 1111 0000 0000 0001 0100 1010 1111 1010 0100
0000 0000 0010 0000 1010 1111 1010 0101 0000 0000
0010 0100 1010 1111 1010 0000 0000 0000 0001 1000
1010 1111 1010 0000 0000 0000 0001 1100 1000 1111
1010 1110 0000 0000 0001 1100 1000 1111 1011 1000
0000 0000 0001 1000 0000 0001 1100 1110 0000 0000
0001 1001 0010 0101 1100 1000 0000 0000 0000 0001
0010 1001 0000 0001 0000 0000 0110 0101 1010 1111
1010 1000 0000 0000 0001 1100 0000 0000 0000 0000
0111 1000 0001 0010 0000 0011 0000 1111 1100 1000
0010 0001 0001 0100 0010 0000 1111 1111 1111 0111
1010 1111 1011 1001 0000 0000 0001 1000 0011 1100
0000 0100 0001 0000 0000 0000 1000 1111 1010 0101
0000 0000 0001 1000 0000 1100 0001 0000 0000 0000
1110 1100 0010 0100 1000 0100 0000 0100 0011 0000
1000 1111 1011 1111 0000 0000 0001 0100 0010 0111
1011 1101 0000 0000 0010 0000 0000 0011 1110 0000
0000 0000 0000 1000 0000 0000 0000 0000 0001 0000
0010 0001
```

You want to do machine code?

```
0010 0111 1011 1101 1111 1111 1110 0000 1010 1111
1011 1111 0000 0000 0001 0100 1010 1111 1010 0100
0000 0000 0010 0000 1010 1111 1010 0101 0000 0000
0010 0100 1010 1111 1010 0000 0000 0000 0001 1000
1010 1111 1010 0000 0000 0000 0001 1100 1000 1111
1010 1110 0000 0000 0001 1100 1000 1111 1011 1000
0000 0000 0001 1000 0000 0001 1100 1110 0000 0000
0001 1001 0010 0101 1100 1000 0000 0000 0000 0001
0010 1001 0000 0001 0000 0000 0110 0101 1010 1111
1010 1000 0000 0000 0001 1100 0000 0000 0000 0000
0111 1000 0001 0010 0000 0011 0000 1111 1100 1000
0010 0001 0001 0100 0010 0000 1111 1111 1111 0111
1010 1111 1011 1001 0000 0000 0001 1000 0011 1100
0000 0100 0001 0000 0000 0000 1000 1111 1010 0101
0000 0000 0001 1000 0000 1100 0001 0000 0000 0000
1110 1100 0010 0100 1000 0100 0000 0100 0011 0000
1000 1111 1011 1111 0000 0000 0001 0100 0010 0111
1011 1101 0000 0000 0010 0000 0000 0011 1110 0000
0000 0000 0000 1000 0000 0000 0000 0000 0001 0000
0010 0001
```

— and this does . . . ?

Walking things back a little

Here is same code written in **MIPS** assembly language
(without symbolic labels for registers / memory locations)

```
addi $29, $29, -32
sw $31, 20($29)
sw $4,  32($29)
sw $5, 36($29)
sw $0,  24($29)
sw $0, 28($29)
lw $14, 28($29)
lw $24, 24($29)
multu $14, $14
addi $8, $14, 1
slti $1, $8, 101
sw $8,  28($29)
```

```
mflo $15
addu $25, $24, $15
bne $1, $0, -9
sw $25, 24($29)
lui $4, 4096
lw $5, 24($29)
jal 1048812
addi $4, $4, 1072
lw $31, 20($29)
addi $29, $29, 32
jr $31
move $2,  $0
```

Walking things back a little **more**

Here is same code again (in **MIPS** assembly language) with **symbolic** names for registers / memory locations.

```
        .text
        .align 2
        .globl main
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
loop:
        lw t6, 28(sp)
        mul t7, t6, t6
        lw t8, 24(sp)
        addu t9, t8, t7
        sw t9, 24(sp)
        addu t0, t6, 1
        sw t0, 28(sp)
        ble t0, 100, loop
        la a0, str
        lw a1, 24(sp)
        jal printf
        move v0, 0
        lw ra, 20(sp)
        addu sp, sp, 32
        jr ra
        .data
        .align 0
str:
        .asciz "The sum from
                0 .. 100 is %d\n"
```

Walking things back a little **more**

Here is same code again (in **MIPS** assembly language) with **symbolic** names for registers / memory locations.

```
        .text
        .align 2
        .globl main
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
loop:
        lw t6, 28(sp)
        mul t7, t6, t6
        lw t8, 24(sp)
        addu t9, t8, t7
        sw t9, 24(sp)
        addu t0, t6, 1
        sw t0, 28(sp)
        ble t0, 100, loop
        la a0, str
        lw a1, 24(sp)
        jal printf
        move v0, 0
        lw ra, 20(sp)
        addu sp, sp, 32
        jr ra
        .data
        .align 0
str:
        .asciz "The sum from
                0 .. 100 is %d\n"
```

As these translations are straightforward, code such as the above would be better for code generation (why?)

Walking things back a little **more**

Here is same code again (in **MIPS** assembly language) with **symbolic** names for **registers** / memory locations.

```
        .text
        .align 2
        .globl main
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
loop:
        lw t6, 28(sp)
        mul t7, t6, t6
        lw t8, 24(sp)
        addu t9, t8, t7
        sw t9, 24(sp)
        addu t0, t6, 1
        sw t0, 28(sp)
        ble t0, 100, loop
        la a0, str
        lw a1, 24(sp)
        jal printf
        move v0, 0
        lw ra, 20(sp)
        addu sp, sp, 32
        jr ra
        .data
        .align 0
str:
        .asciz "The sum from
                0 .. 100 is %d\n"
```

As these translations are straightforward, code such as the above would be better for code generation (why?)

Walking things back a little **more**

Here is same code again (in **MIPS** assembly language) with **symbolic** names for **registers** / **memory locations**.

```
        .text
        .align 2
        .globl main
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
loop:
        lw t6, 28(sp)
        mul t7, t6, t6
        lw t8, 24(sp)
        addu t9, t8, t7
        sw t9, 24(sp)
        addu t0, t6, 1
        sw t0, 28(sp)
        ble t0, 100, loop
        la a0, str
        lw a1, 24(sp)
        jal printf
        move v0, 0
        lw ra, 20(sp)
        addu sp, sp, 32
        jr ra
        .data
        .align 0
str:
        .asciz "The sum from
                0 .. 100 is %d\n"
```

As these translations are straightforward, code such as the above would be better for code generation (why?)

Memory access

An assembly snippet from the last slide:

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

Many of the arguments like `20(sp)` and `32(sp)` are references to locations in memory, relative to a particular memory pointer `sp` (a *stack pointer*) — offset by *multiples of 4*.

Memory access

An assembly snippet from the last slide:

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

Many of the arguments like `20(sp)` and `32(sp)` are references to locations in memory, relative to a particular memory pointer `sp` (a *stack pointer*) — offset by *multiples of 4*.

- The computer can only access data, whose location it is told about (!)

Memory access

An assembly snippet from the last slide:

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

Many of the arguments like `20(sp)` and `32(sp)` are references to locations in memory, relative to a particular memory pointer `sp` (a *stack pointer*) — offset by *multiples of 4*.

- The computer can only access data, whose location it is told about (!)
- But also, for reasons to do with computer architecture, it can only access data at memory locations which are multiples of some particular integer (such as 4 or 8).

Memory access

An assembly snippet from the last slide:

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

Many of the arguments like `20(sp)` and `32(sp)` are references to locations in memory, relative to a particular memory pointer `sp` (a *stack pointer*) — offset by *multiples of 4*.

- The computer can only access data, whose location it is told about (!)
- But also, for reasons to do with computer architecture, it can only access data at memory locations which are multiples of some particular integer (such as 4 or 8).

This impacts how easily and quickly data may be accessed.

Memory access

An assembly snippet from the last slide:

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

Many of the arguments like `20(sp)` and `32(sp)` are references to locations in memory, relative to a particular memory pointer `sp` (a *stack pointer*) — offset by *multiples of 4*.

- Computers can only use data, which are stored in **registers** (such as the memory pointer `sp`), or whose locations in memory *it knows about* (!)
- Furthermore, the locations in memory that it can easily access is restricted by issues of architecture — how the computer is built (!)

Memory alignment

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

Memory alignment

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

- Most modern CPUs have a 32-bit or 64-bit data bus (*i.e.*, the CPU reads 32 or 64 bits in one cycle) — we say that the **word size** is 32 or 64 bits.

Memory alignment

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

- Most modern CPUs have a 32-bit or 64-bit data bus (*i.e.*, the CPU reads 32 or 64 bits in one cycle) — we say that the **word size** is 32 or 64 bits.
  - Memory accesses are 32 or 64 bits at a time
  - Memory is *accessible* at intervals of 32 or 64 bits

Memory alignment

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

- Most modern CPUs have a 32-bit or 64-bit data bus (*i.e.*, the CPU reads 32 or 64 bits in one cycle) — we say that the **word size** is 32 or 64 bits.
  - Memory accesses are 32 or 64 bits at a time
  - Memory is *accessible* at intervals of 32 or 64 bits
- **However:** for historical reasons, these same computers address memory in bytes (8-bit units).

Memory alignment

```
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
```

- Most modern CPUs have a 32-bit or 64-bit data bus (*i.e.*, the CPU reads 32 or 64 bits in one cycle) — we say that the **word size** is 32 or 64 bits.
  - Memory accesses are 32 or 64 bits at a time
  - Memory is *accessible* at intervals of 32 or 64 bits
- **However:** for historical reasons, these same computers address memory in bytes (8-bit units).

How easily data may be accessed, and how quickly, depends on whether the address is **aligned** or not; and memory is generally not allocated in units of bytes.

What is memory alignment?

For an $8 \cdot N$-bit CPU (with words of N bytes:
$N = 4$ means 32 bit, $N = 8$ means 64 bit) —
addresses divisible by N are **word-boundaries**.

Memory access of N bytes starting at a word-boundary is **aligned**, access at all other addresses is **misaligned**.

What is memory alignment?

For an $8 \cdot N$-bit CPU (with words of N bytes:
$N = 4$ means 32 bit, $N = 8$ means 64 bit) —
addresses divisible by N are **word-boundaries**.

Memory access of N bytes starting at a word-boundary is **aligned**, access at all other addresses is **misaligned**.

- **32 bit CPUs:** $N = 4$, so word boundaries are at addresses that are multiples of 4 (*e.g.*, 0, 4, 8, 12, 16, 20, ...)

What is memory alignment?

For an $8 \cdot N$-bit CPU (with words of N bytes:
$N = 4$ means 32 bit, $N = 8$ means 64 bit) —
addresses divisible by N are **word-boundaries**.

Memory access of N bytes starting at a word-boundary is **aligned**, access at all other addresses is **misaligned**.

- **32 bit CPUs:** $N = 4$, so word boundaries are at addresses that are multiples of 4 (*e.g.*, 0, 4, 8, 12, 16, 20, ...)
- **64 bit CPUs:** $N = 8$, so word boundaries are at addresses that are multiples of 8 (*e.g.*, 0, 8, 16, 24, 32, 40, ...)

What is memory alignment?

For an $8 \cdot N$-bit CPU (with words of N bytes:
$N = 4$ means 32 bit, $N = 8$ means 64 bit) —
addresses divisible by N are **word-boundaries**.

Memory access of N bytes starting at a word-boundary is **aligned**, access at all other addresses is **misaligned**.

- **32 bit CPUs:** $N = 4$, so word boundaries are at addresses that are multiples of 4 (*e.g.*, 0, 4, 8, 12, 16, 20, ...)
- **64 bit CPUs:** $N = 8$, so word boundaries are at addresses that are multiples of 8 (*e.g.*, 0, 8, 16, 24, 32, 40, ...)
- *etc.*

What is memory alignment?

For an $8 \cdot N$-bit CPU (with words of N bytes:
$N = 4$ means 32 bit, $N = 8$ means 64 bit) —
addresses divisible by N are **word-boundaries**.

Memory access of N bytes starting at a word-boundary is **aligned**, access at all other addresses is **misaligned**.

- **32 bit CPUs:** $N = 4$, so word boundaries are at addresses that are multiples of 4 (*e.g.*, 0, 4, 8, 12, 16, 20, ...)
- **64 bit CPUs:** $N = 8$, so word boundaries are at addresses that are multiples of 8 (*e.g.*, 0, 8, 16, 24, 32, 40, ...)
- *etc.*

Note! Misaligned access is much slower (approx. 10 times) than aligned access for many (most? all?) CPUs — and in some cases produces a processor exception.

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).
- To read a word at address **3001**:
  - The address 3001 is not a multiple of 4, so a word at this address spans two words of memory.

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).
- To read a word at address **3001**:
  - The address 3001 is not a multiple of 4, so a word at this address spans two words of memory.
  - The CPU may read a word at address 3001 by:

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).
- To read a word at address **3001**:
  - The address 3001 is not a multiple of 4, so a word at this address spans two words of memory.
  - The CPU may read a word at address 3001 by:
    - reading the word at addresses 3000, **3001**, **3002**, **3003**;

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).
- To read a word at address **3001**:
  - The address 3001 is not a multiple of 4, so a word at this address spans two words of memory.
  - The CPU may read a word at address 3001 by:
    - reading the word at addresses 3000, **3001**, **3002**, **3003**;
    - reading the word at addresses **3004**, 3005, 3006, 3007;

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).
- To read a word at address **3001**:
  - The address 3001 is not a multiple of 4, so a word at this address spans two words of memory.
  - The CPU may read a word at address 3001 by:
    - reading the word at addresses 3000, **3001**, **3002**, **3003**;
    - reading the word at addresses **3004**, 3005, 3006, 3007;

    and keeping the contents of addresses 3001 – 3004.

Misaligned memory access: an example

Example: assume we have a CPU with 32-bit (= 4 byte) words.

- To read a word at address **3000**:
  - The address 3000 is a multiple of 4, so this address is at the beginning of a word of memory.
  - The CPU may then read a word at address 3000 by reading in a single word from memory (the contents of addresses **3000**, **3001**, **3002**, **3003**).
- To read a word at address **3001**:
  - The address 3001 is not a multiple of 4, so a word at this address spans two words of memory.
  - The CPU may read a word at address 3001 by:
    - reading the word at addresses 3000, **3001**, **3002**, **3003**;
    - reading the word at addresses **3004**, 3005, 3006, 3007;

    and keeping the contents of addresses 3001 – 3004.
  - Or maybe the CPU would produce an error, and needs its hand held to read these two words of memory.

Compiler alignment should be “lawful”

Because of the huge speed penalty for misaligned memory access (or the possibility that the CPU will produce an error), compilers should help to ensure that all memory access is aligned when possible.

Compiler alignment should be “lawful”

Because of the huge speed penalty for misaligned memory access (or the possibility that the CPU will produce an error), compilers should help to ensure that all memory access is aligned when possible.

- Compilers achieve this by locating data whose length is a word or more, at *word boundaries* (using padding where necessary — fill up unused space to a word boundary).

Compiler alignment should be “lawful”

Because of the huge speed penalty for misaligned memory access (or the possibility that the CPU will produce an error), compilers should help to ensure that all memory access is aligned when possible.

- Compilers achieve this by locating data whose length is a word or more, at *word boundaries* (using padding where necessary — fill up unused space to a word boundary).
- Data whose length is less than a word doesn’t necessarily have to be aligned to a word boundary — but we should still guarantee that it does not go across two aligned words

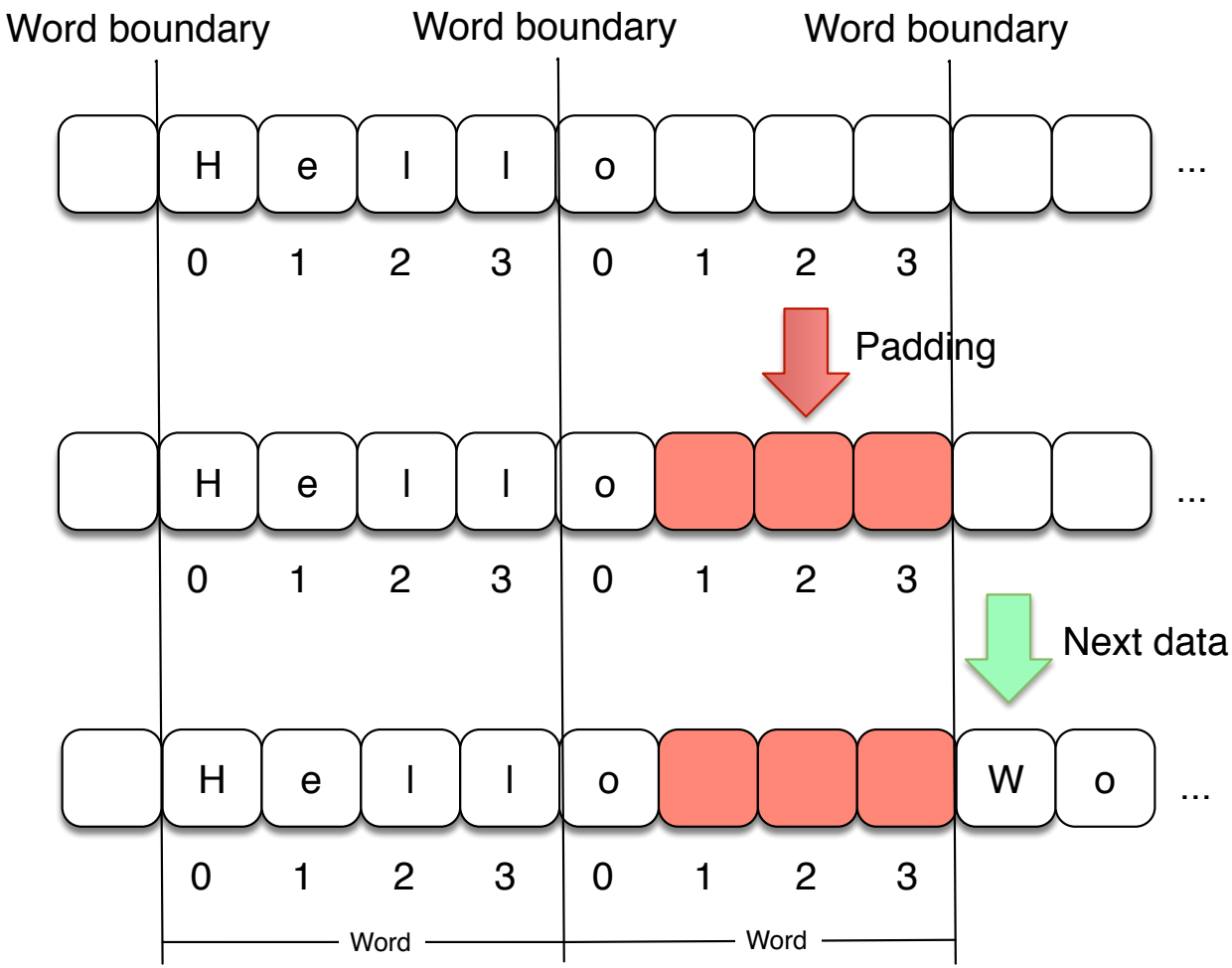
Compiler alignment should be “lawful”

Because of the huge speed penalty for misaligned memory access (or the possibility that the CPU will produce an error), compilers should help to ensure that all memory access is aligned when possible.

- Compilers achieve this by locating data whose length is a word or more, at *word boundaries* (using padding where necessary — fill up unused space to a word boundary).
- Data whose length is less than a word doesn’t necessarily have to be aligned to a word boundary — but we should still guarantee that it does not go across two aligned words
  - For single bytes: this is never an issue
  - For two-byte blocks (*e.g.*, `short` in C or Java): achieved by aligning to addresses which are multiples of 2 bytes
  - For four-byte blocks (in 64-bit architectures or higher): aligning to addresses which are multiples of 4 bytes

Storing aligned data

Here is an example of padding data for a 32 bit CPU. We want to store two distinct strings: `"Hello"` and `"World"`.



Aligning with assembly

Most assembler languages have build in support that aligns (some) data automatically.

- RISC-V assembler has the `.align` directive. Putting `.align n` in a line means that the following line is aligned on a 2^n-byte boundary.

Example:

```
.align 2
.asciiz "Hello world!"
```

Here, `.align 2` aligns the next value (the string `"Hello world!"`) on a word boundary on a 32-bit machine — that is, an address which is a multiple of $4 = 2^2$.

The assembler

The **assembler** — the program which translates assembly language to machine code — primarily does two things.

The assembler

The **assembler** — the program which translates assembly language to machine code — primarily does two things.

- Translate commands in assembly language like `addi t3 t6 t8` into machine code.

The assembler

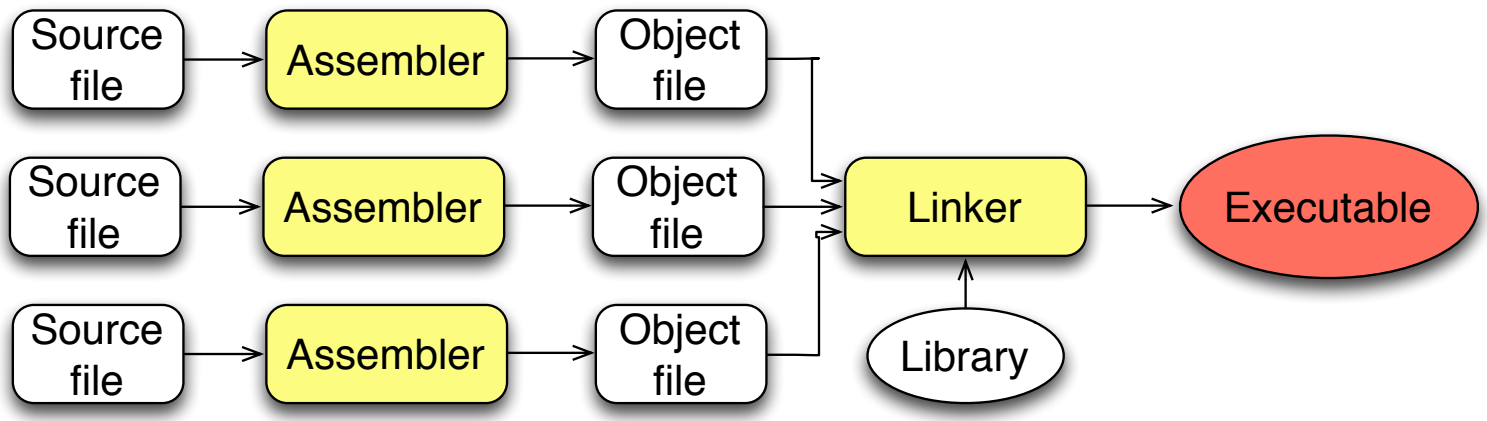
The **assembler** — the program which translates assembly language to machine code — primarily does two things.

- Translate commands in assembly language like `addi t3 t6 t8` into machine code.
- Convert symbolic addresses such as `main` or `loop` into machine addresses (sometimes deferring to the linker).

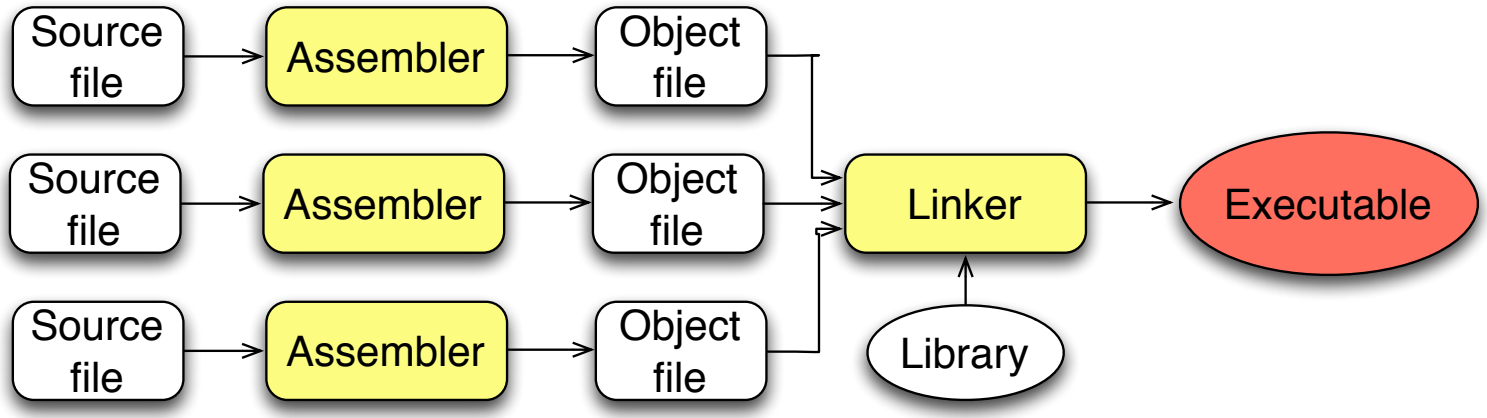
The assembler

The **assembler** — the program which translates assembly language to machine code — primarily does two things.

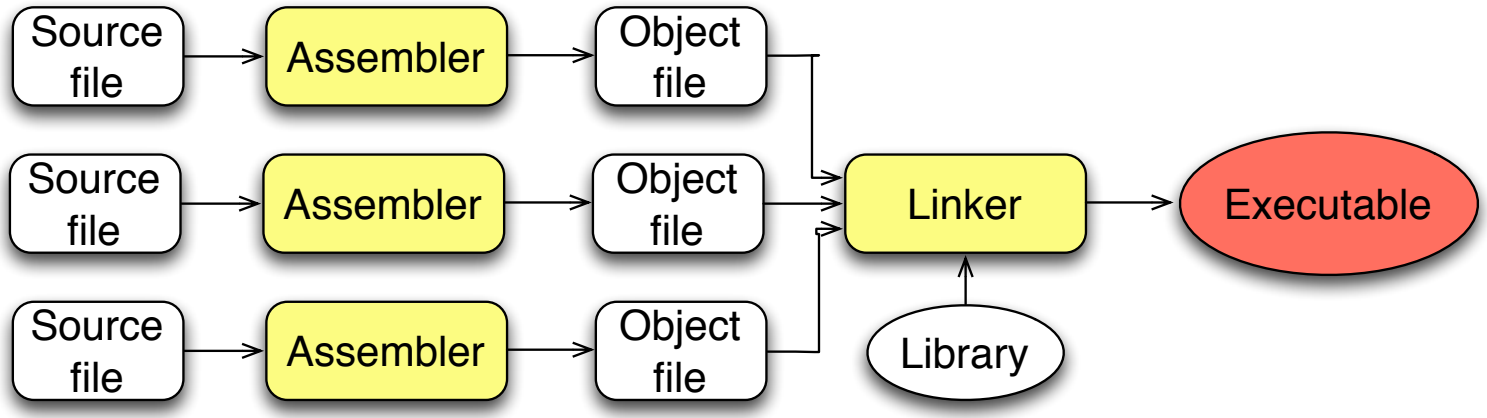
- Translate commands in assembly language like `addi t3 t6 t8` into machine code.
- Convert symbolic addresses such as `main` or `loop` into machine addresses (sometimes deferring to the linker).
  - Symbolic addresses are used to name commonly occurring register names / memory locations, for readability.



The assembler

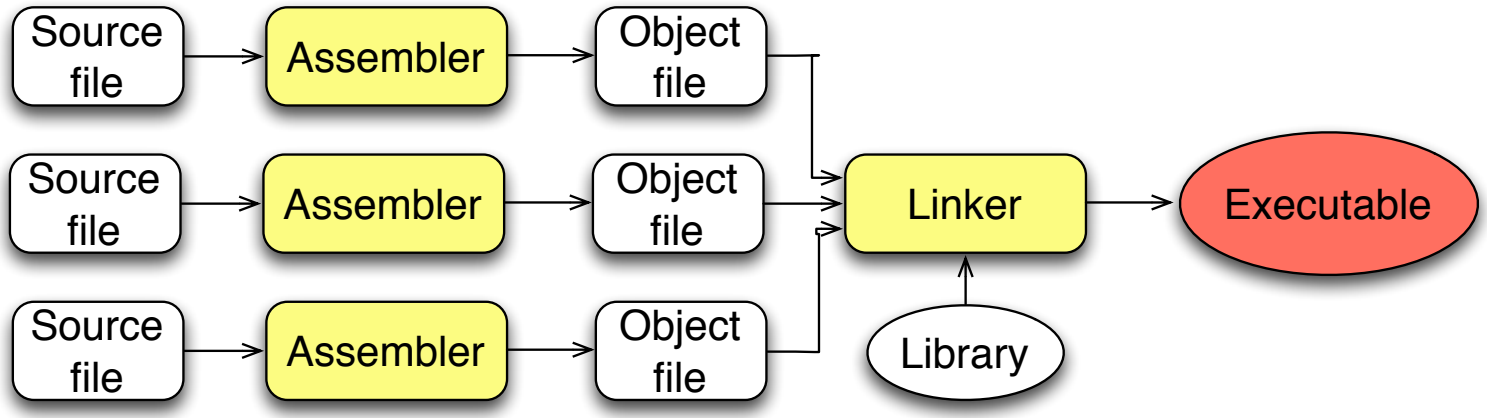


The assembler



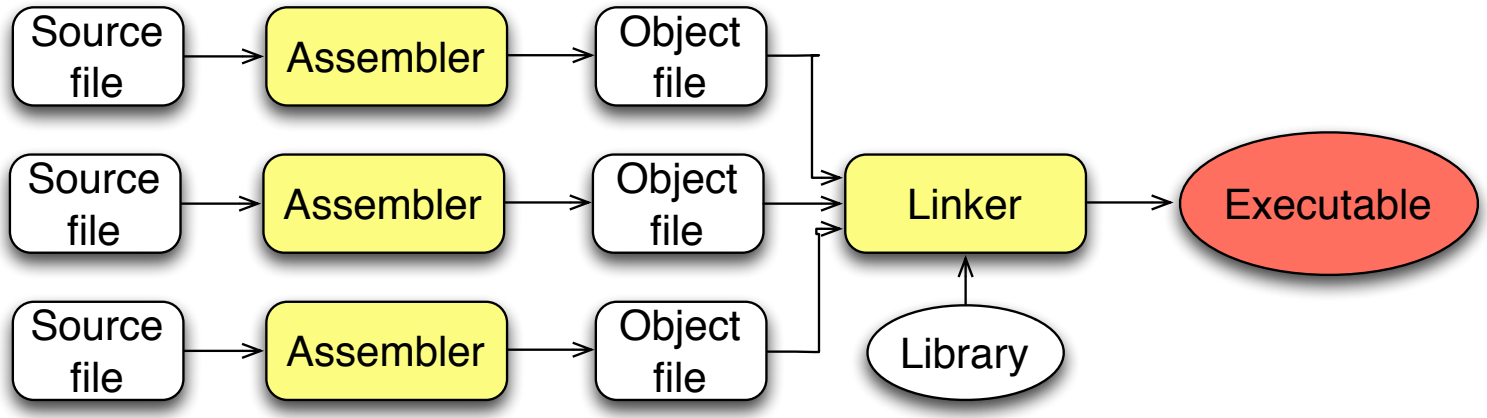
- **Source files:** produced by a compiler (often separately), potentially with references to subroutines in different source files / libraries.

The assembler



- **Source files:** produced by a compiler (often separately), potentially with references to subroutines in different source files / libraries.
- **Object files:** machine code which still contains references to external code — not directly executable.

The assembler



- **Source files:** produced by a compiler (often separately), potentially with references to subroutines in different source files / libraries.
- **Object files:** machine code which still contains references to external code — not directly executable.

The linker gets all object files and libraries and puts the right addresses into holes, yielding an executable.

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                    ; an address, for flow control purposes
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
  addu t0, t6, 1        ; t0 = t6 + 1
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
  addu t0, t6, 1        ; t0 = t6 + 1
  sw t0, 28(sp)         ; store an int from t0 into sp+28
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
  addu t0, t6, 1        ; t0 = t6 + 1
  sw t0, 28(sp)         ; store an int from t0 into sp+28
  ble t0, 100, loop     ; if t0 ⩽ 100, goto loop
```

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
  addu t0, t6, 1        ; t0 = t6 + 1
  sw t0, 28(sp)         ; store an int from t0 into sp+28
  ble t0, 100, loop     ; if t0 ⩽ 100, goto loop
```

. . . things like this, and not very much more.

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
  addu t0, t6, 1        ; t0 = t6 + 1
  sw t0, 28(sp)         ; store an int from t0 into sp+28
  ble t0, 100, loop     ; if t0 ⩽ 100, goto loop
```

. . . things like this, and not very much more.

- Essentially **everything**.
  (This is what programs are made from, after all!)

Assembly lines

What can we do with assembly?

- Not very much:

```
loop:                   ; an address, for flow control purposes
  lw t6, 28(sp)         ; load an int into register t6 from sp+28
  mul t7, t6, t6        ; t7 = t6 * t6
  lw t8, 24(sp)         ; load an int into register t8 from sp+24
  addu t9, t8, t7       ; t9 = t8 + t7
  sw t9, 24(sp)         ; store an int from register t9 into sp+24
  addu t0, t6, 1        ; t0 = t6 + 1
  sw t0, 28(sp)         ; store an int from t0 into sp+28
  ble t0, 100, loop     ; if t0 ⩽ 100, goto loop
```

  . . . things like this, and not very much more.

- Essentially **everything**.
  (This is what programs are made from, after all!)

The main question is:
how to do the things we want to do, with assembly.

What's the plan?

We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

What's the plan?

We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

We can do simple arithmetic and use `for` loops and print `"hello world"` easily enough. But as soon as we want to do something complicated, how do we do it?

What's the plan?

We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

We can do simple arithmetic and use `for` loops and print `"hello world"` easily enough. But as soon as we want to do something complicated, how do we do it?

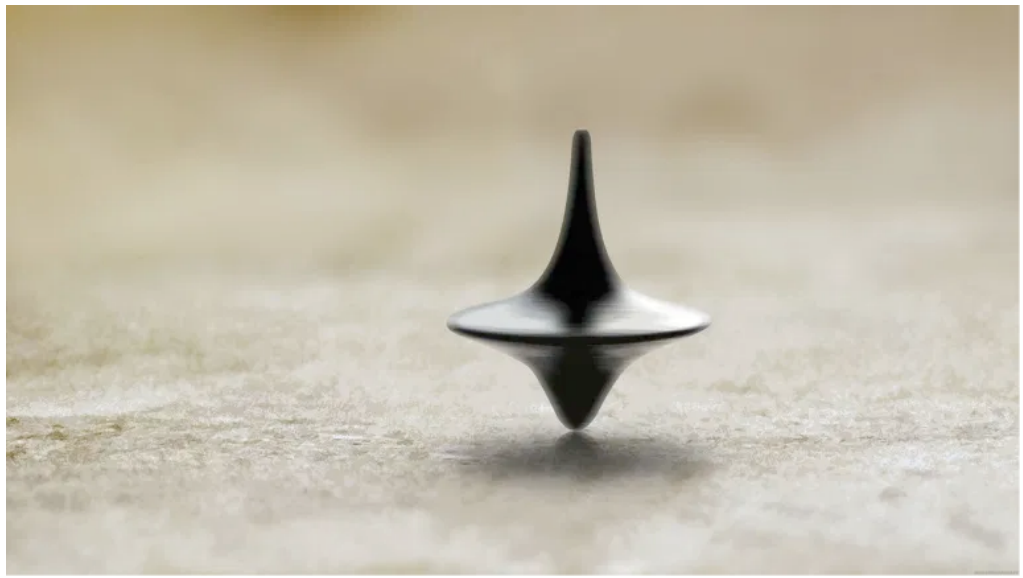
Something complicated. . . like writing a compiler, for instance.

What's the plan?

We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

We can do simple arithmetic and use `for` loops and print `"hello world"` easily enough. But as soon as we want to do something complicated, how do we do it?

Something complicated. . . like writing a compiler, for instance.



What's the plan?

We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

We can do simple arithmetic and use `for` loops and print `"hello world"` easily enough. But as soon as we want to do something complicated, how do we do it?

Something complicated. . . like writing a compiler, for instance.

- Subroutine calls, and



What's the plan?

We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

We can do simple arithmetic and use `for` loops and print `"hello world"` easily enough. But as soon as we want to do something complicated, how do we do it?

Something complicated. . . like writing a compiler, for instance.

- Subroutine calls, and
- Recursion.



What's the plan?

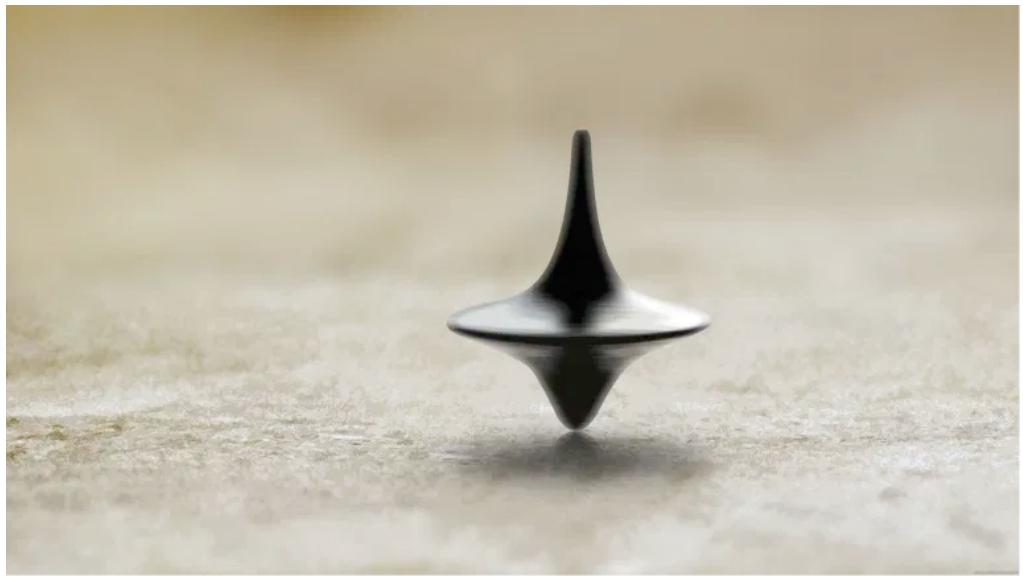
We need a strategy for code generation — a model for how to realise a complex program, ideally on different architectures.

We can do simple arithmetic and use `for` loops and print `"hello world"` easily enough. But as soon as we want to do something complicated, how do we do it?

Something complicated. . . like writing a compiler, for instance.

- Subroutine calls, and
- Recursion.

We need a way to represent different levels of solving problems and sub-problems — using partial solutions to carry on working on the main problem.



Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression)

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of “operations” to be carried out in a fixed order (no parallelism/concurrency).

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of "operations" to be carried out in a fixed order (no parallelism/concurrency).
- Each "operation" may be another procedure call (invoking some subroutine S)

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of "operations" to be carried out in a fixed order (no parallelism/concurrency).
- Each "operation" may be another procedure call (invoking some subroutine S)
  - each such a subroutine S is **also** a sequence of operations of this kind

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of "operations" to be carried out in a fixed order (no parallelism/concurrency).
- Each "operation" may be another procedure call (invoking some subroutine *S*)
  - each such a subroutine *S* is **also** a sequence of operations of this kind
  - each new invocation of *S* should be independent of previous ones, except through its input arguments

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of "operations" to be carried out in a fixed order (no parallelism/concurrency).
- Each "operation" may be another procedure call (invoking some subroutine *S*)
  - each such a subroutine *S* is **also** a sequence of operations of this kind
  - each new invocation of *S* should be independent of previous ones, except through its input arguments
  - subroutine calls are realised in such a way that they "terminate" in a well-defined way with a return value

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of "operations" to be carried out in a fixed order (no parallelism/concurrency).
- Each "operation" may be another procedure call (invoking some subroutine *S*)
  - each such a subroutine *S* is **also** a sequence of operations of this kind
  - each new invocation of *S* should be independent of previous ones, except through its input arguments
  - subroutine calls are realised in such a way that they "terminate" in a well-defined way with a return value
  - the operation performed after the subroutine terminates, is the next operation in the calling procedure

Sequential procedures

The simplest case of a subroutine call is a sequential procedure (e.g., evaluation of a mathematical expression):

- The procedure is given as a sequence of "operations" to be carried out in a fixed order (no parallelism/concurrency).
- Each "operation" may be another procedure call (invoking some subroutine *S*)
  - each such a subroutine *S* is **also** a sequence of operations of this kind
  - each new invocation of *S* should be independent of previous ones, except through its input arguments
  - subroutine calls are realised in such a way that they "terminate" in a well-defined way with a return value
  - the operation performed after the subroutine terminates, is the next operation in the calling procedure

We may consider how to realise this through managing the system resources.

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

In a procedure with recursive subroutines, we generally don't know what subroutine calls we will make

$\Longrightarrow$ managed dynamically

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

In a procedure with recursive subroutines, we generally don't know what subroutine calls we will make

$\Longrightarrow$ managed dynamically

In a more complex program, we also generally don't know what persistent objects we may need to create

$\Longrightarrow$ managed dynamically

Managing run-time resources

To realise a model of sequential procedure calls (or anything else), there is an important distinction in how resources are allocated:

- **Statically** — at compile-time;
- **Dynamically** — at run-time.

In a procedure with recursive subroutines, we generally don't know what subroutine calls we will make

$\Longrightarrow$ managed dynamically

In a more complex program, we also generally don't know what persistent objects we may need to create

$\Longrightarrow$ managed dynamically

The instructions for each subroutine (apart from dynamically loaded libraries) are part of the program, once compiled

$\Longrightarrow$ managed statically

A sketch: running a program

How memory is organised at run-time

When a program is invoked:

A sketch: running a program

How memory is organised at run-time

When a program is invoked:

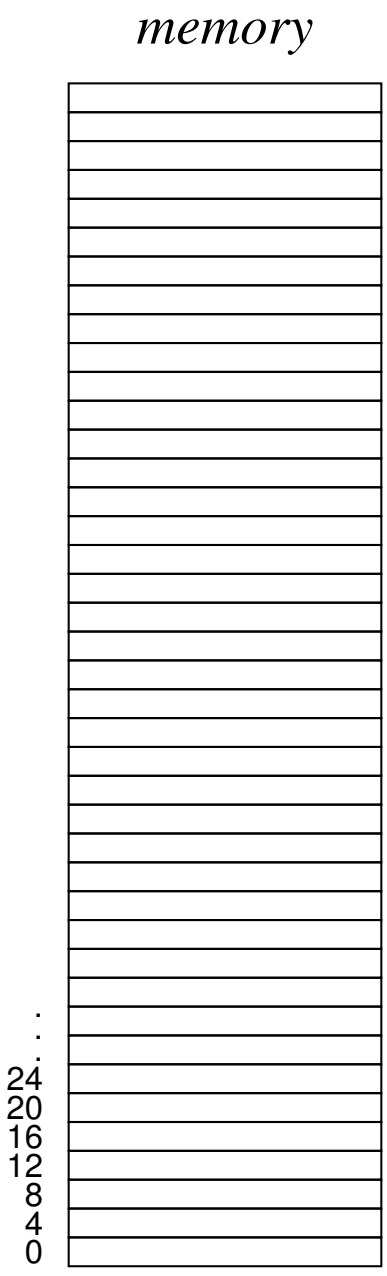
- The OS allocates space for the program
  — a block of *virtual memory*
  (both to store code and data).

A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).

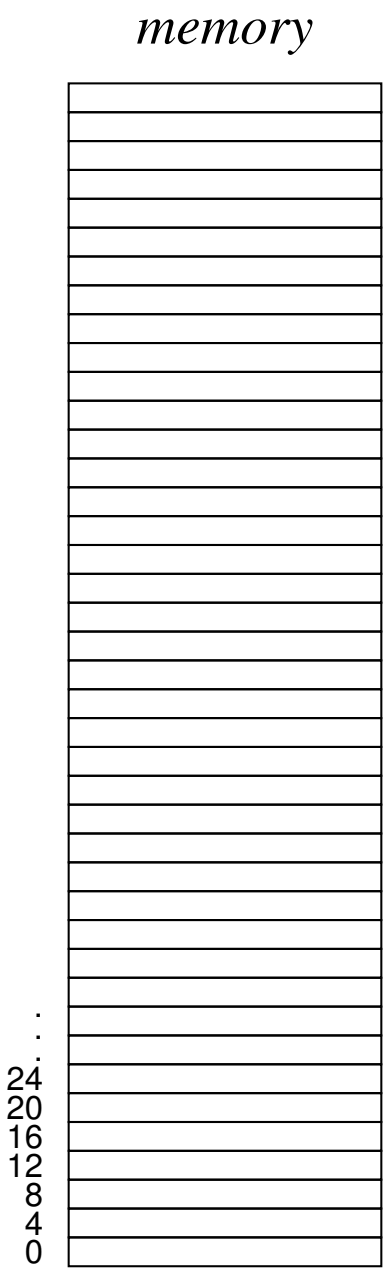


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.

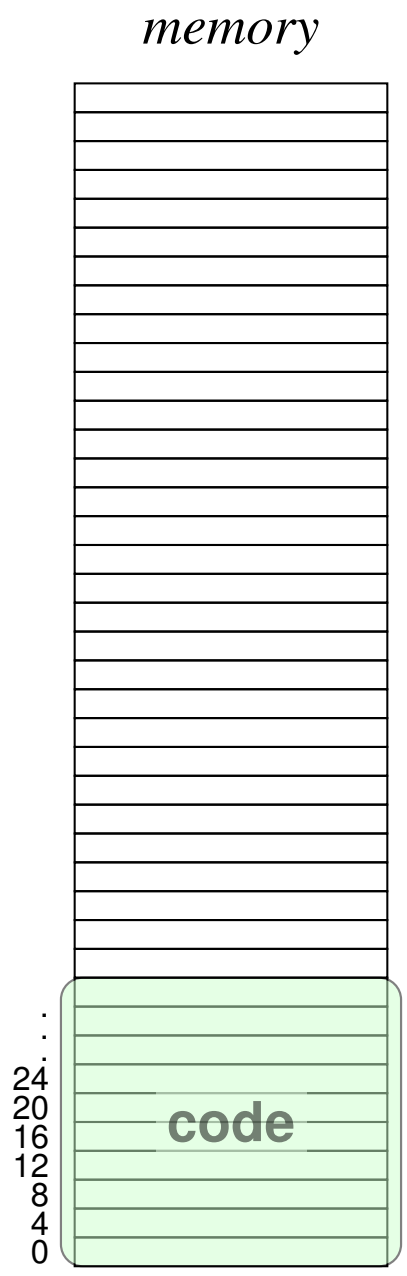


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.

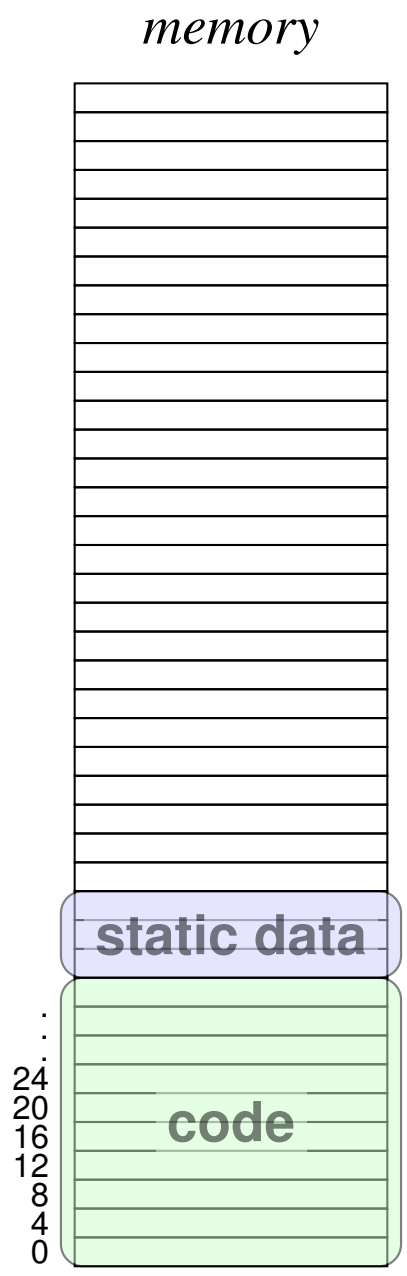


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.

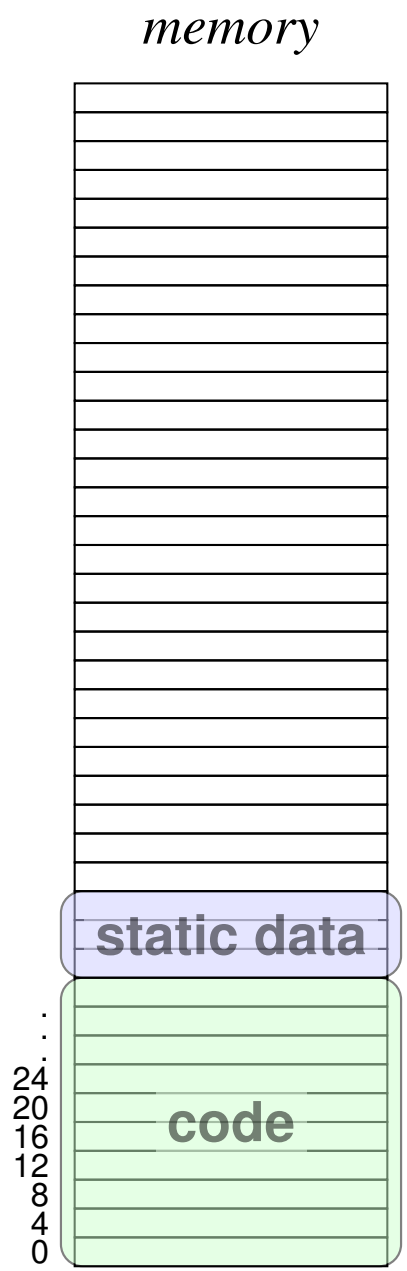


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables (‘static’ data)

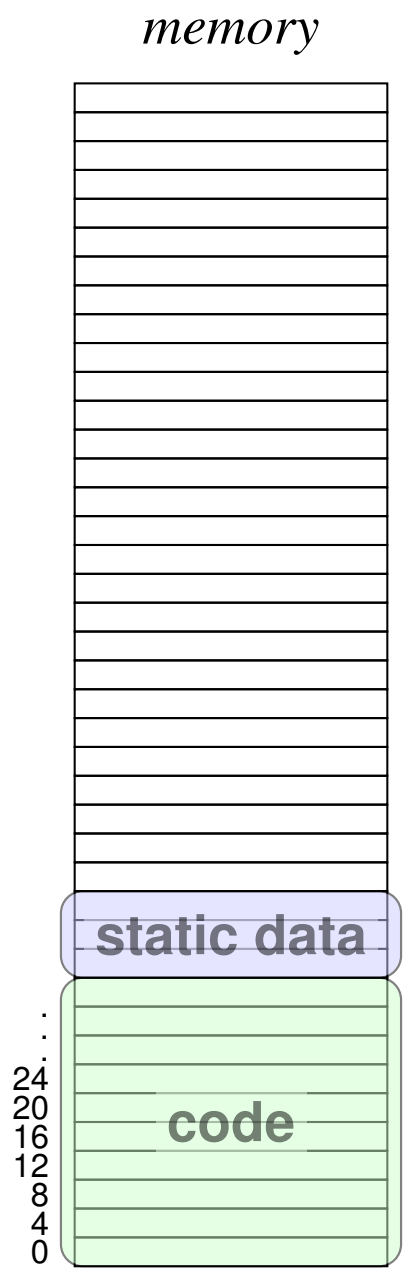


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables (‘static’ data)
  - . . . sometimes, may require linking libraries dynamically as well

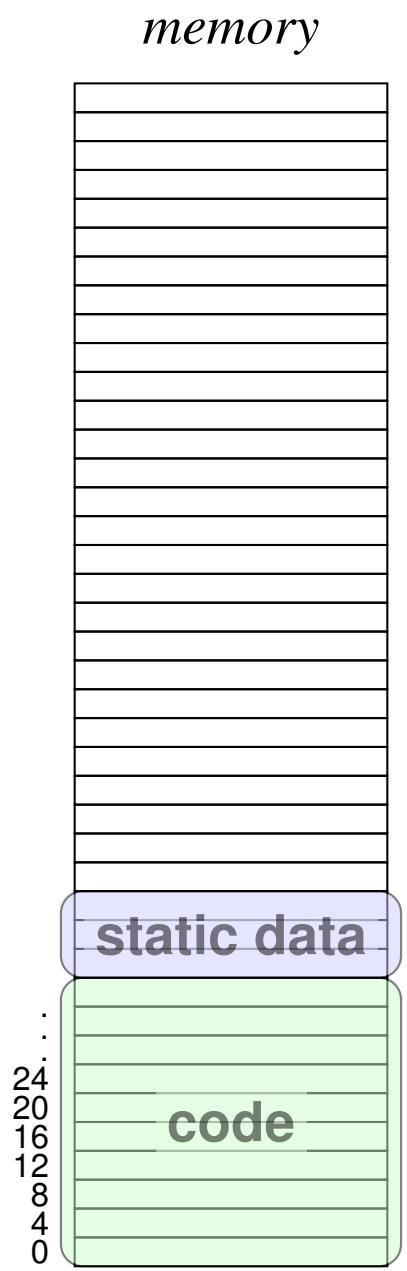


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables (‘static’ data)
  - . . . sometimes, may require linking libraries dynamically as well
  - . . . we generally expect not to modify the loaded code at run-time

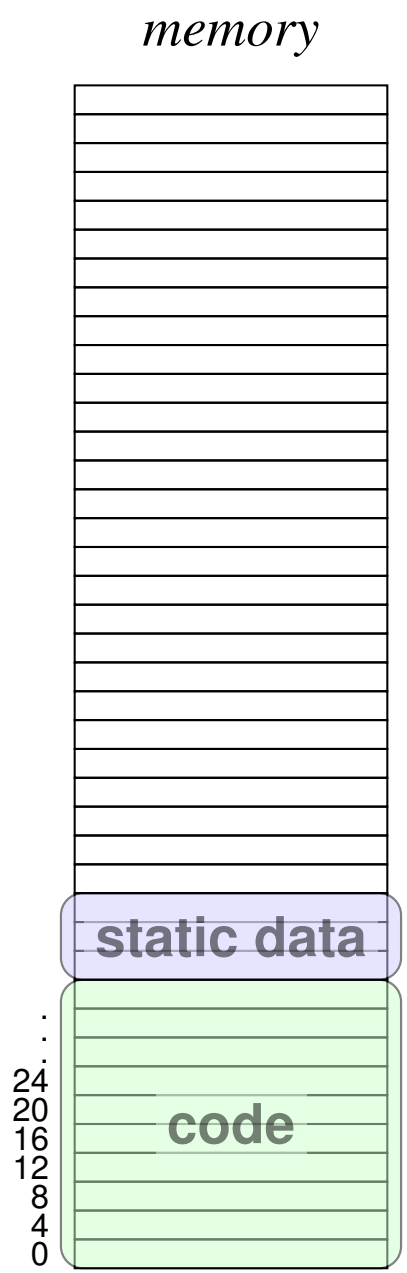


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables ('static' data)
  - . . . sometimes, may require linking libraries dynamically as well
  - . . . we generally expect not to modify the loaded code at run-time
- The OS jumps to the entry point of the program — *i.e.*, the program's entry point (*e.g.*, the `main` method in Java).

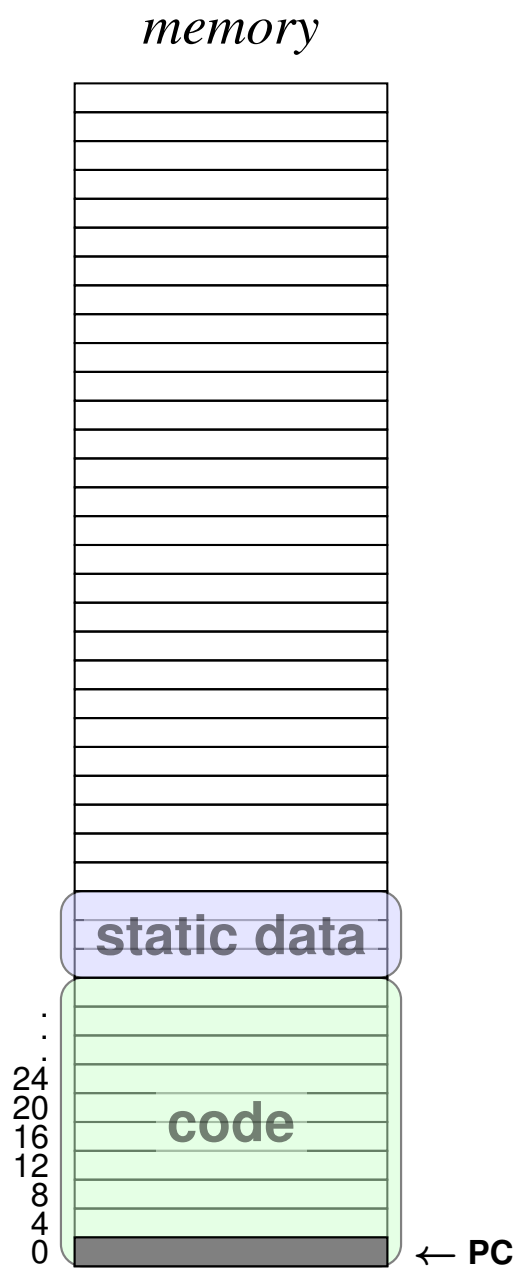


A sketch: running a program

How memory is organised at run-time

When a program is invoked:

- The OS allocates space for the program — a block of *virtual memory* (both to store code and data).
- The OS loads the code of the program into the allocated space.
  - . . . including all subroutines, libraries, and global variables ('static' data)
  - . . . sometimes, may require linking libraries dynamically as well
  - . . . we generally expect not to modify the loaded code at run-time
- The OS jumps to the entry point of the program — *i.e.*, the program's entry point (*e.g.*, the `main` method in Java).



Global data

Consider the following Java program.

```
public class Test {
   static int x = 1;
   public static void main ( String [] args ) {
      System.out.println ( x ); }}
```

Global data

Consider the following Java program.

```
public class Test {
   static int x = 1;
   public static void main ( String [] args ) {
      System.out.println ( x ); }}
```

As `x` is **static** — it is in-scope throughout the entire program (in any scope which doesn't declare a local variable `x`) — it may be stored at a fixed address in the program's memory.

Global data

Consider the following Java program.

```
public class Test {
   static int x = 1;
   public static void main ( String [] args ) {
      System.out.println ( x ); }}
```

As `x` is **static** — it is in-scope throughout the entire program (in any scope which doesn't declare a local variable `x`) — it may be stored at a fixed address in the program's memory.

- We say of the location of `x`, that it is **statically allocated**.
- The memory location of the code which defines the method `main()`, is also statically allocated
- Depending on the programming language, other static data may exist beyond global variables

Global data

Consider the following Java program.

```
public class Test {
   static int x = 1;
   public static void main ( String [] args ) {
      System.out.println ( x ); }}
```

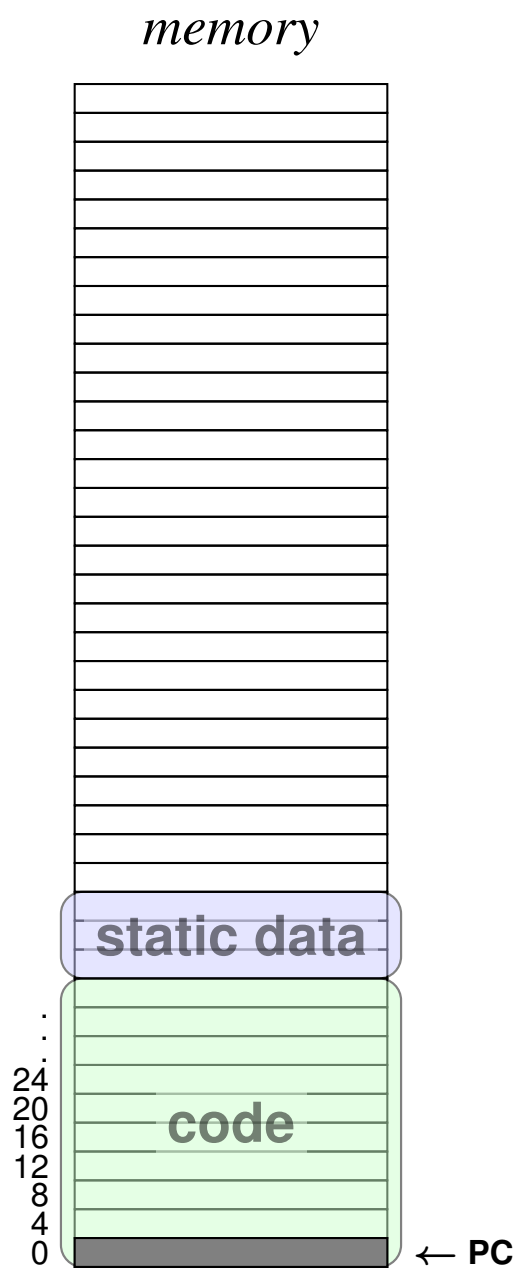
As `x` is **static** — it is in-scope throughout the entire program (in any scope which doesn't declare a local variable `x`) — it may be stored at a fixed address in the program's memory.

- We say of the location of `x`, that it is **statically allocated**.
- The memory location of the code which defines the method `main()`, is also statically allocated
- Depending on the programming language, other static data may exist beyond global variables

This won't do for local variables in a procedure call, however...

Dynamic memory allocation?

How is memory organised at run-time?

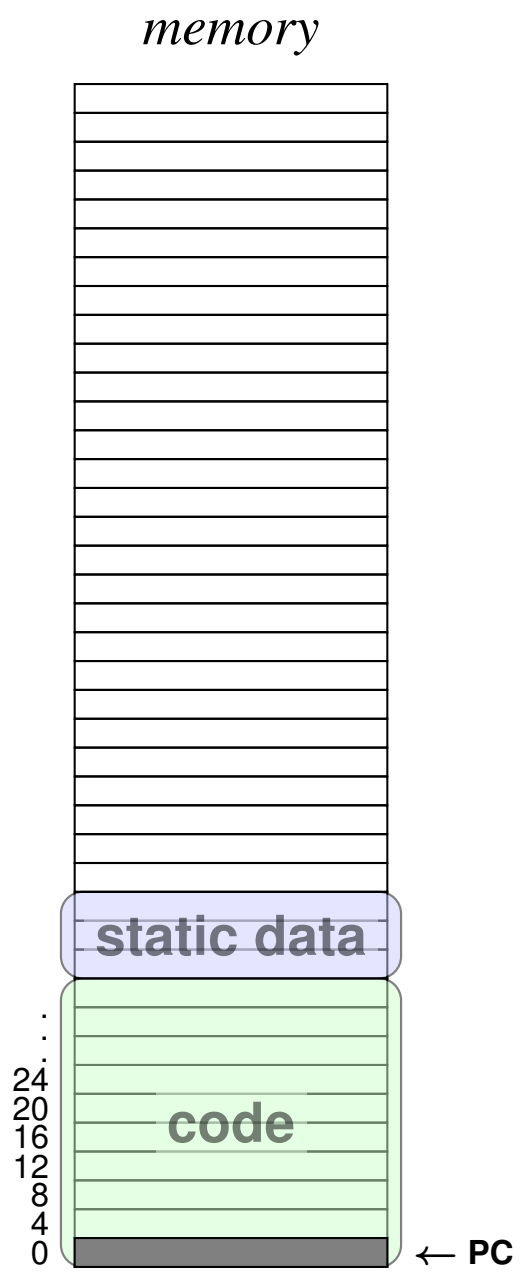


Dynamic memory allocation?

How is memory organised at run-time?

What about subroutine calls / persistent objects?

Anything which is generated at run-time
must be managed dynamically, by definition.



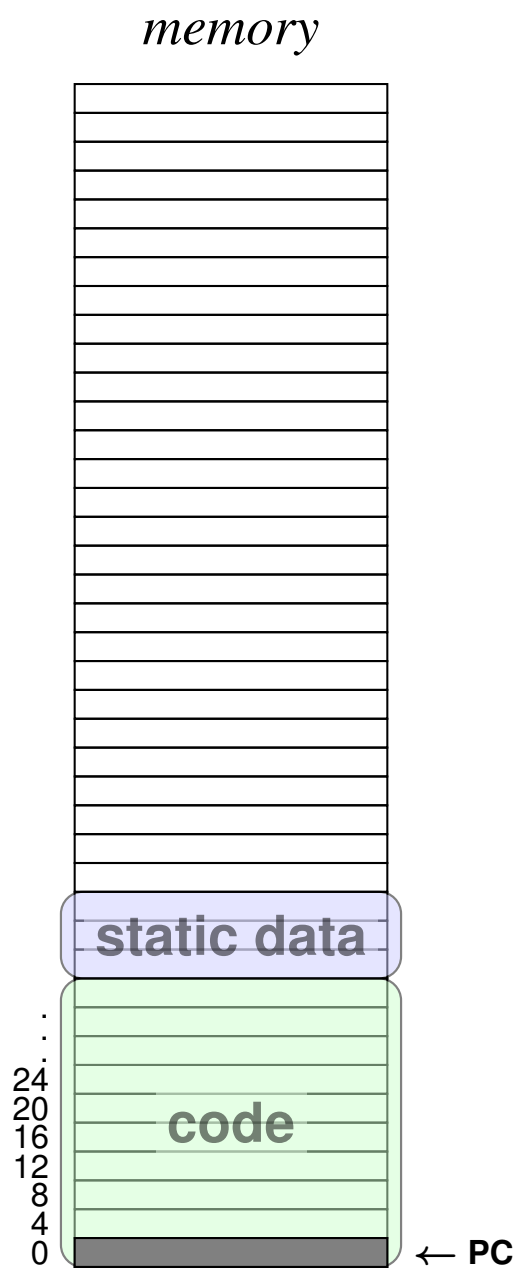
Dynamic memory allocation?

How is memory organised at run-time?

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their “lifetime” and scope



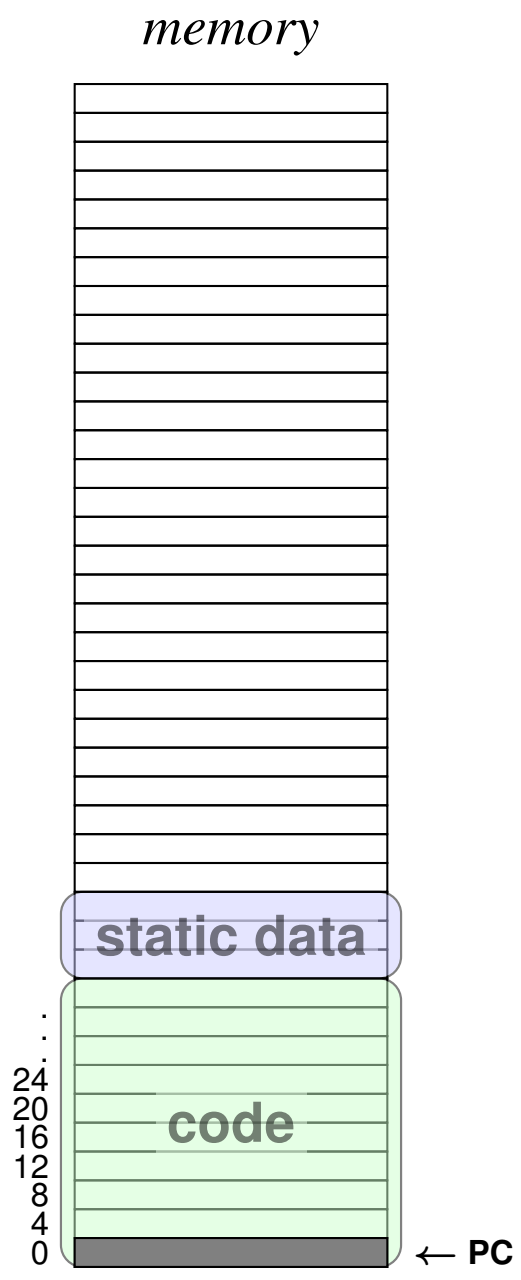
Dynamic memory allocation?

How is memory organised at run-time?

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their “lifetime” and scope
- Persistent objects: longer-lived (again by definition), more akin to statically initialised data



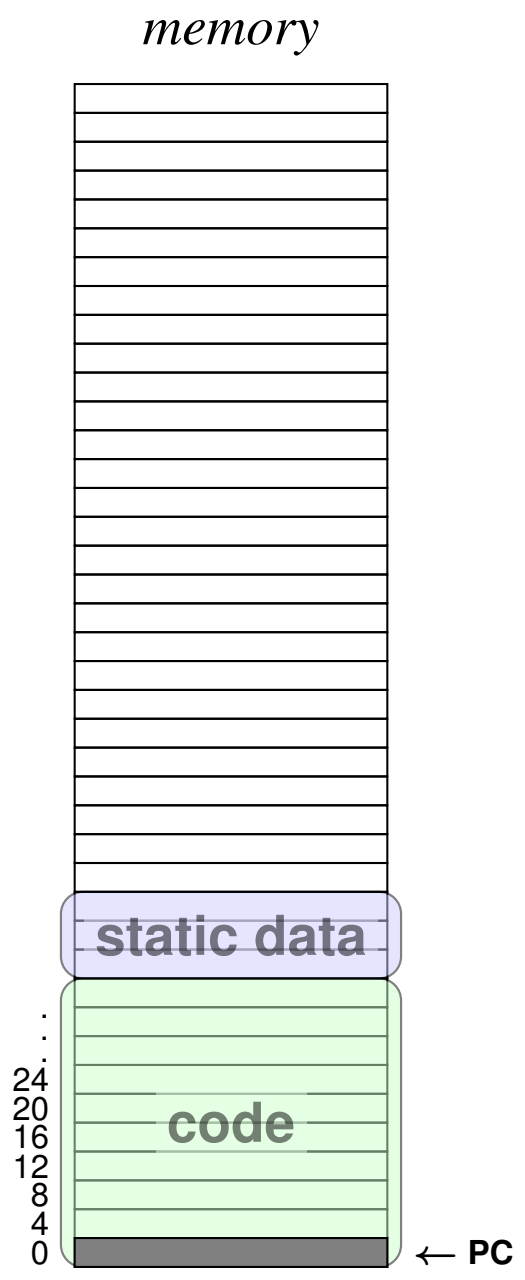
Dynamic memory allocation?

How is memory organised at run-time?

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their “lifetime” and scope
- Persistent objects: longer-lived (again by definition), more akin to statically initialised data
- Thanks to OS memory management, we may **pretend** that the *available memory* is allocated statically.
  — but how can we use it?



Dynamic memory allocation?

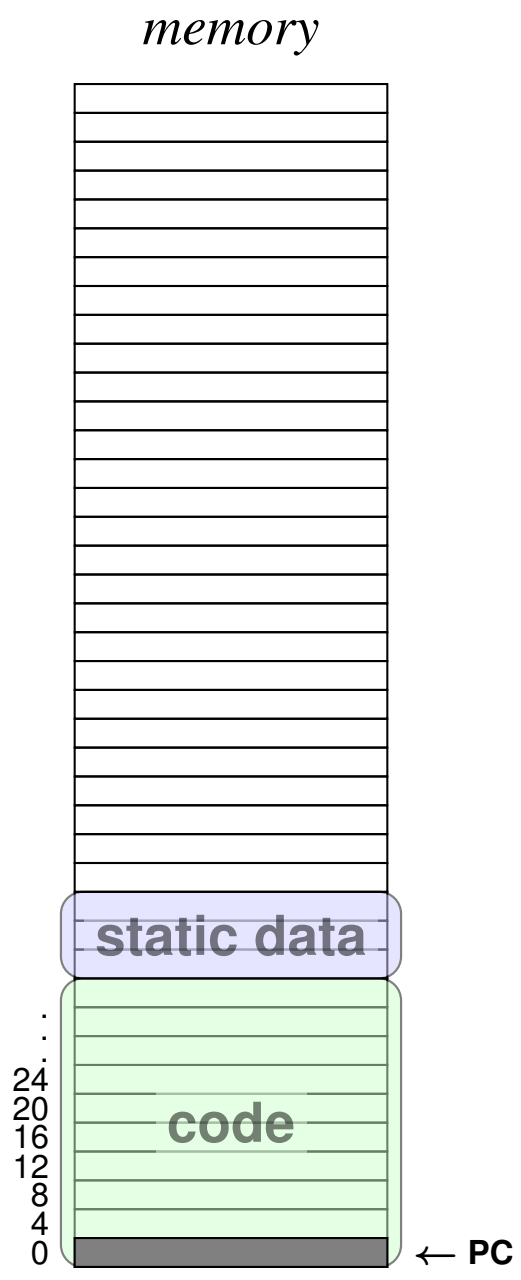
How is memory organised at run-time?

What about subroutine calls / persistent objects?

Anything which is generated at run-time must be managed dynamically, by definition.

- Subroutine calls (and their local variables): limited in their “lifetime” and scope
- Persistent objects: longer-lived (again by definition), more akin to statically initialised data
- Thanks to OS memory management, we may **pretend** that the *available memory* is allocated statically.
  — but how can we use it?

Code-generation must describe (at compile time) how these will be managed at run-time.



Accessing dynamically allocated memory

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

Accessing dynamically allocated memory

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

What memory addresses does the computer “know about”?

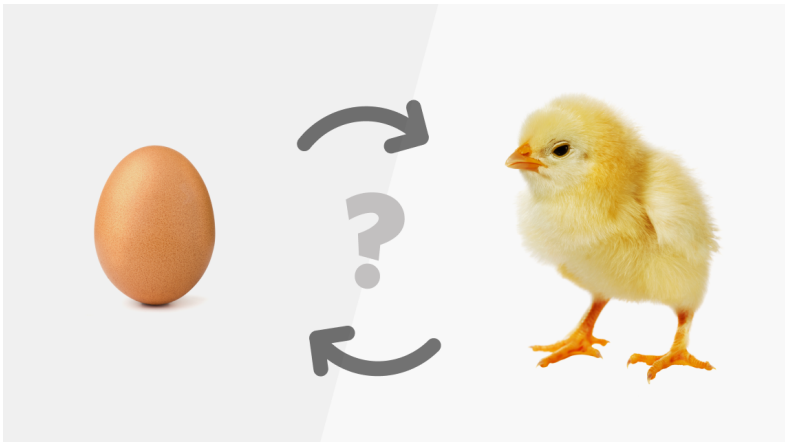
Accessing dynamically allocated memory

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

What memory addresses does the computer “know about”?

— apart from **static** addresses (known at compile time):
only those stored somewhere the computer knows about!



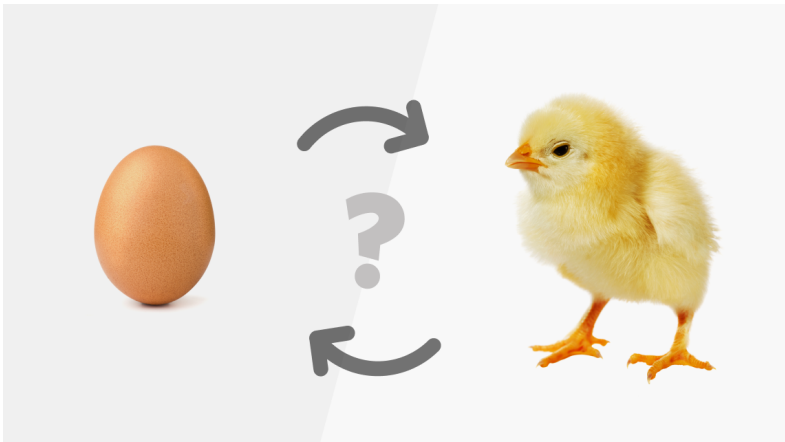
Accessing dynamically allocated memory

The computer can only work with

- data whose location in memory it knows about
- *instructions* whose location in memory it knows about
- in **some** cases: data stored in general-purpose registers

What memory addresses does the computer “know about”?

— apart from **static** addresses (known at compile time):
only those stored somewhere the computer knows about!



Next lecture: solving this problem using the stack pointer, and the memory stack!