

COMP110: Principles of Computing

9: Compilers and interpreters

Learning outcomes

- ▶ **Explain** the difference between interpretation, just-in-time compilation and ahead-of-time compilation
- ▶ **Distinguish** the basic parts of a compilation pipeline and recall how they operate
- ▶ **Describe** how common high-level code structures translate to machine code

Worksheet E

- ▶ Compilers and machine code
- ▶ Due in class on **Monday 21st November** (next week)

Final worksheet submission

- ▶ Soon!!! (see MyFalmouth)
- ▶ Should be easy: "Download Zip" from GitHub, rename, upload to LearningSpace
- ▶ If any of your work is **not** on GitHub (e.g. images on imgur), be sure to **add it to the zip!**
 - ▶ Exception: SpaceChem **videos** can stay as YouTube links
- ▶ Even if you have submitted everything on time via GitHub, **late submission** to LearningSpace will have the standard penalty for late submission (40% cap)!

How programs are executed

Executing programs

- ▶ CPUs execute **machine code**
- ▶ Programs must be **translated** into machine code for execution
- ▶ There are three main ways of doing this:
 - ▶ An **interpreter** is an application which reads the program source code and executes it directly
 - ▶ An **ahead-of-time (AOT) compiler**, often just called a **compiler**, is an application which converts the program source code into executable machine code
 - ▶ A **just-in-time (JIT) compiler** is halfway between the two — it compiles the program on-the-fly at runtime

Examples

Interpreted:

- ▶ Python
- ▶ Lua
- ▶ JavaScript
(in old web
browsers)
- ▶ Bespoke
scripting
languages

Compiled:

- ▶ C
- ▶ C++
- ▶ Swift

JIT compiled:

- ▶ Java
- ▶ C#
- ▶ JavaScript
(in modern
web
browsers)
- ▶ Jython

NB: technically any language could appear in any column here, but this is where they typically are

Interpreter vs compiler

- ▶ Run-time efficiency: compiler > interpreter
 - ▶ The compiler translates the program **in advance**, on the developer's machine
 - ▶ The interpreter translates the program **at runtime**, on the user's machine — this takes extra time

Interpreter vs compiler

- ▶ Portability: compiler < interpreter
 - ▶ A compiled program can only run on the operating system and CPU architecture it was compiled for
 - ▶ An interpreted program can run on any machine, as long as a suitable interpreter is available

Interpreter vs compiler

- ▶ Ease of development: compiler < interpreter
 - ▶ Writing an AOT or JIT compiler (especially a good one) is hard, and required in-depth knowledge of the target machine
 - ▶ Writing an interpreter is easy in comparison

Interpreter vs compiler

- ▶ Dynamic language features: compiler < interpreter
 - ▶ The interpreter is already on the end user's machine, so programs can use it e.g. to dynamically generate and execute new code
 - ▶ The AOT compiler is not generally on the end user's machine, so this is more difficult

Interpreter vs compiler

- ▶ JIT compilers have similar pros/cons to interpreters
 - ▶ Runtime efficiency: JIT > interpreter (e.g. code inside a loop only needs to be translated once, then can be executed many times)
 - ▶ Ease of development: JIT < interpreter

Virtual machines

- ▶ Many modern interpreters and JIT compilers translate programs into **bytecode**
- ▶ Bytecode is essentially machine code for a **virtual machine (VM)**
- ▶ Translation from source code to bytecode can be done ahead of time
- ▶ At runtime, translate the bytecode (by interpretation or JIT compilation) into machine code for the physical machine
- ▶ E.g. a Java JAR file, a .NET executable, a Python .pyc or .pyo file all contain bytecode for their respective VMs

Assemblers

- ▶ **Assembly language** is designed to translate directly into machine code
- ▶ An ahead-of-time compile for assembly language is called an **assembler**
- ▶ Generally much simpler than an AOT compiler for a higher-level language

The C++ build process

Preprocessor

- ▶ Handles **header inclusion**, **macro expansion** and **conditional compilation**
- ▶ Outputs modified C++ source code

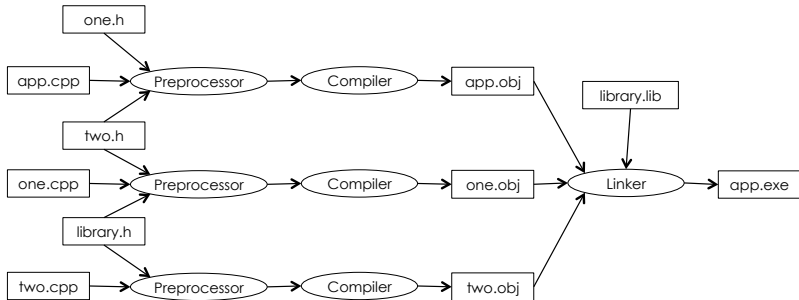
Compiler

- ▶ Translates each source file into an **object file** containing machine code
- ▶ May be configured to **optimise** code (e.g. calculate constant values, expand inline functions, re-order instructions to take advantage of CPU architecture, etc.)

Linker

- ▶ Combines the object files together with any external libraries to produce an **executable** (on Windows, a .exe file)

The C++ build process



The MIPS architecture

MIPS

- ▶ An example of a **Reduced Instruction Set Computer (RISC)** architecture
 - ▶ Small number of simple instructions — computational power comes from executing many instructions per second
 - ▶ Compare with **Complex Instruction Set Computer (CISC)** architecture (e.g. Intel x86) — large number of complex instructions — fewer instructions per second, but shorter programs
- ▶ MIPS was popular in 1980s – 2000s
 - ▶ Embedded systems
 - ▶ Consoles (Nintendo 64, PlayStation 1 and 2)
- ▶ Easier to understand than most CPU instruction sets in common use today

Online MIPS simulator

<http://rivoire.cs.sonoma.edu/cs351/wemips/>

Registers

- ▶ Memory locations inside the CPU
- ▶ Faster to access than main memory
- ▶ Registers in MIPS architecture include:
 - ▶ **\$zero**: constant 0
 - ▶ **\$t0–\$t9**: temporary storage
 - ▶ **\$s0–\$s7**: saved temporary storage
- ▶ Each register holds a single 32-bit value

Adding register values

```
ADD $d, $s, $t
```

- ▶ `$d`, `$s` and `$t` are register names
- ▶ This adds the value of `$s` to the value of `$t`, and stores the result in `$d`

```
SUB $d, $s, $t
```

- ▶ Subtracts the value of `$t` from the value of `$s`, and stores the result in `$d`

Adding a constant

```
ADDI $d, $s, c
```

- ▶ `$d` and `$s` are register names, `c` is an integer constant
- ▶ This adds the value of `$s` to `c`, and stores the result in `$d`
- ▶ `ADDI` = “add immediate” — as in `c` is specified immediately in the code, not looked up from a register
- ▶ There is no `SUBI` instruction — to subtract `c`, add `-c`

More fun with ADDI

- ▶ Socratic `FALCOMPED`
- ▶ What does this code do?

```
ADDI $s0, $s1, 0
```

- ▶ What does this code do?

```
ADDI $s0, $zero, 12
```

- ▶ MIPS does not have dedicated instructions for setting a register value to a constant or to the value of another register — it has to be done with `ADDI`

Control flow in MIPS

Labels and jumping

- ▶ In assembly code, can set a **label** on any line:

```
MyLabel: ADD $s0, $s1, 1
```

- ▶ Some instructions use labels to refer to a location in the code
- ▶ E.g. the `j` instruction simply jumps (backwards or forwards) to the specified line:

```
j MyLabel
```

Branching

- **Branching** is **conditional jumping**

```
BEQ $s, $t, Label
```

- This jumps to `Label` **if and only if** the value of `$s` equals the value of `$t`

```
BNE $s, $t, Label
```

- This jumps to `Label` **if and only if** the value of `$s` does not equal the value of `$t`

Conditionals

- Branching allows us to implement **if statements**

```
if s0 != 0:  
    s1 += 1  
else:  
    s2 += 1
```

```
beq $s0, $zero, Else  
addi $s1, $s1, 1  
j End  
Else: addi $s2, $s2, 1  
End:
```

Loops

- Branching allows us to implement **while loops**

```
i = 0
total = 0
limit = 10

while i != limit:
    total += i
    i += 1
#end while
```

```
ADDI $s0, $zero, 0
ADDI $s1, $zero, 0
ADDI $s2, $zero, 10

Loop: BEQ $s0, $s2, LoopEnd
ADD $s1, $s1, $s0
ADDI $s0, $s0, 1
J Loop
LoopEnd:
```

Exercise

Write a piece of Python code equivalent to the following:

```
ADDI $s0, $zero, 10
ADDI $s1, $zero, 0

Loop: BEQ $s0, $zero, LoopEnd
ADD $s1, $s1, $s0
ADDI $s0, $s0, -1
J Loop
LoopEnd:
```

Not quite a while loop

- ▶ Socratic FALCOMPED
- ▶ What is the difference between these two programs?
- ▶ (NB: assume $\$s0 \geq 0$)

```
ADDI $s1, $zero, 0
```

```
Loop: BEQ $s0, $zero, ←  
      LoopEnd
```

```
ADD $s1, $s1, $s0
```

```
ADDI $s0, $s0, -1
```

```
J Loop
```

```
LoopEnd:
```

```
ADDI $s1, $zero, 0
```

```
Loop: ADD $s1, $s1, $s0
```

```
ADDI $s0, $s0, -1
```

```
BNE $s0, $zero, Loop
```

The code on the right implements a **do-while** loop (which Python doesn't have, but other languages do)

Function calls

- ▶ Function calls can be implemented using the jump instruction:
 - ▶ To call the function: save the address of the instruction after the current one, then jump to the function
 - ▶ To return from the function: jump to the previously saved address
- ▶ MIPS has `jal` and `jr` instructions and `$ra` register for this purpose
- ▶ Socratic `FALCOMPED`: why save the return address? Why not just hard-code it into the program?
- ▶ Nested function calls require a **stack** of return addresses

MIPS machine code

MIPS instructions

- ▶ Each line of MIPS assembly code can be translated into a **machine code instruction**
- ▶ 1 line of assembly = 1 instruction
- ▶ Each instruction is a **32 bit** value
- ▶ First 6 bits specify the **opcode**; how the remaining 26 bits are interpreted depends on which opcode it is

Anatomy of an instruction

R-type instruction:

[illegible]

- ▶ **opcode** and **function** together specify the operation to execute
 - ▶ E.g. `ADD` has opcode `000000` and function `100000`
 - ▶ E.g. `SUB` has opcode `000000` and function `100010`
- ▶ Some instructions specify a **shift** amount
 - ▶ For `ADD` `SUB` etc these 5 bits are ignored
- ▶ Registers are identified by a 5-bit number
 - ▶ E.g. `$zero` → `00000`, `$s0` → `01000`, `$s1` → `01001`
 - ▶ There are 32 registers

Example

```
ADD $s0, $s0, $s1
```



opcode	s	t	d	shift	function
000000	01000	01001	01001	00000	100000

Anatomy of an instruction

l-type instruction:

opcode 6 bits						\$s 5 bits		\$t 5 bits		c 16 bits													
------------------	--	--	--	--	--	---------------	--	---------------	--	--------------	--	--	--	--	--	--	--	--	--	--	--	--	--

- ▶ **opcode** specifies the operation to execute
 - ▶ E.g. `ADDI` has opcode `001000`
- ▶ `c` is specified as a 16-bit number

Example

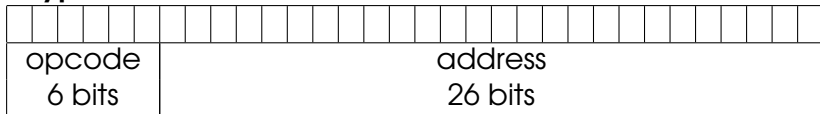
```
ADDI $s0, $s1, 123
```



opcode	s	t	C
001000	01001	01000	0000000001111011

Anatomy of an instruction

J-type instruction:



- ▶ **opcode** specifies the operation to execute
 - ▶ E.g. `J` has opcode `000010`
- ▶ **address** is specified as a 26-bit number