COMP110: Principles of Computing

9: Compilers and interpreters

Learning outcomes

- Outcome 1
- Outcome 2
- Outcome 3

FALMOUTH UNIVERSITY

How programs are executed

Executing programs

- CPUs execute **machine code**

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution
- There are three main ways of doing this:

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

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

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

Examples

Interpreted:

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

Compiled:

- C
- C++
- Swift

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

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

Interpreter vs compiler

- Run-time efficiency: compiler $>$ interpreter
  - The compiler translates the program **in advance**, on the developer's machine

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

Interpreter vs compiler

- Portability: compiler $<$ interpreter
  - A compiled program can only run on the operating system and CPU architecture it was compiled for

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

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

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

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

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

Interpreter vs compiler

- JIT compilers have similar pros/cons to interpreters

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)

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

Virtual machines

- Many modern interpreters and JIT compilers translate programs into **bytecode**

Virtual machines

- Many modern interpreters and JIT compilers translate programs into **bytecode**
- Bytecode is essentially machine code for a **virtual machine (VM)**

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

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

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

Machine code

MIPS

MIPS

- An example of a **Reduced Instruction Set Computer (RISC)** 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

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

- 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

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

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)

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

Registers

- Memory locations inside the CPU

Registers

- Memory locations inside the CPU
- Faster to access than main memory

Registers

- Memory locations inside the CPU
- Faster to access than main memory
- Registers in MIPS architecture include:

Registers

- Memory locations inside the CPU
- Faster to access than main memory
- Registers in MIPS architecture include:
  - `$zero`: constant 0

Registers

- Memory locations inside the CPU
- Faster to access than main memory
- Registers in MIPS architecture include:
    - `$zero`: constant 0
    - `$t0`–`$t9`: temporary storage

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

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

Adding register values

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

Adding register values

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

- `$d`, `$s` and `$t` are register names

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`

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
```

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

Adding a constant

```
ADDI $d, $s, C
```

Adding a constant

```
ADDI $d, $s, C
```

- `$d` and `$s` are register names, `C` is an integer constant

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`

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

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

More fun with ADDI

- Socrative `FALCOMPED`

More fun with ADDI

- Socrative FALCOMPED
- What does this code do?

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

More fun with ADDI

- Socrative `FALCOMPED`
- What does this code do?

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

- What does this code do?

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

More fun with ADDI

- Socrative `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`

Labels and jumping

Labels and jumping

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

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

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

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

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

Branching

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

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

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`

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
```

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`

Loops

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: ADD $s1, $s1, $s0
ADDI $s0, $s0, 1
BNE $s0, $s2, Loop
```