COMP110: Principles of Computing

12: Machine Code

Worksheets

Worksheet 8

Due **today**!

Worksheet 9

Due **Monday 6th January 2020**!

Final hand-in

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**
- See MyFalmouth for the deadline

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**
- See MyFalmouth for the deadline
- Upload a zip containing all your worksheet submissions to LearningSpace

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**
- See MyFalmouth for the deadline
- Upload a zip containing all your worksheet submissions to LearningSpace
  - Worksheets 2 and 8: save the "review" of your quiz attempt as a PDF

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**
- See MyFalmouth for the deadline
- Upload a zip containing all your worksheet submissions to LearningSpace
  - Worksheets 2 and 8: save the "review" of your quiz attempt as a PDF
  - All other worksheets: put the files from your GitHub submission into a folder

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**
- See MyFalmouth for the deadline
- Upload a zip containing all your worksheet submissions to LearningSpace
  - Worksheets 2 and 8: save the "review" of your quiz attempt as a PDF
  - All other worksheets: put the files from your GitHub submission into a folder
- **If** you have changed your work since opening a pull request, please include a README file detailing which worksheets you have changed!

Final hand-in

- You are required to make a **final submission** of all your worksheets by the **summative deadline**
- See MyFalmouth for the deadline
- Upload a zip containing all your worksheet submissions to LearningSpace
  - Worksheets 2 and 8: save the "review" of your quiz attempt as a PDF
  - All other worksheets: put the files from your GitHub submission into a folder
- **If** you have changed your work since opening a pull request, please include a README file detailing which worksheets you have changed!
- Viva sessions in January

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
- Rust

Examples

Interpreted:

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

Compiled:

- C
- C++
- Swift
- Rust

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
- Rust

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

Assemblers

Assemblers

- **Assembly language** is designed to translate directly into machine code

Assemblers

- **Assembly language** is designed to translate directly into machine code
- An ahead-of-time compile for assembly language is called an **assembler**

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 6502 CPU architecture

MOS Technology 6502

MOS Technology 6502

- Designed by Chuck Peddle and team at **MOS Technology**

MOS Technology 6502

- Designed by Chuck Peddle and team at **MOS Technology**
- **8-bit** CPU, 16-bit addressing

MOS Technology 6502

- Designed by Chuck Peddle and team at **MOS Technology**
- **8-bit** CPU, 16-bit addressing
- Clocked between **1MHz** and **3MHz**

MOS Technology 6502

- Designed by Chuck Peddle and team at **MOS Technology**
- **8-bit** CPU, 16-bit addressing
- Clocked between **1MHz** and **3MHz**
- First produced in **1975**

MOS Technology 6502

- Designed by Chuck Peddle and team at **MOS Technology**
- **8-bit** CPU, 16-bit addressing
- Clocked between **1MHz** and **3MHz**
- First produced in **1975**
- Still in production **today**

Uses of the 6502

Uses of the 6502

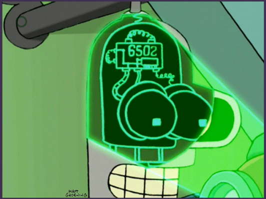
Uses of the 6502

Uses of the 6502

Uses of the 6502

Uses of the 6502

Uses of the 6502



Recap: hexadecimal notation

Recap: hexadecimal notation

- We usually write numbers in **decimal** i.e. **base 10**

Recap: hexadecimal notation

- We usually write numbers in **decimal** i.e. **base 10**
- Hexadecimal is **base 16**

Recap: hexadecimal notation

- We usually write numbers in **decimal** i.e. **base 10**
- Hexadecimal is **base 16**
- Uses extra digits:
  - A=10, B=11, ..., F=15

Recap: hexadecimal notation

- We usually write numbers in **decimal** i.e. **base 10**
- Hexadecimal is **base 16**
- Uses extra digits:
  - A=10, B=11, ..., F=15

Hex	Dec	Hex	Dec	Hex	Dec
00	0	10	16	F0	240
01	1	11	17	F1	241
⋮	⋮	⋮	⋮	⋮	⋮
09	9	19	25	F9	249
0A	10	1A	26	FA	250
0B	11	1B	27	FB	251
0C	12	1C	28	FC	252
0D	13	1D	29	FD	253
0E	14	1E	30	FE	254
0F	15	1F	31	FF	255

How a CPU works

How a CPU works

- Executes a series of **instructions** stored in **memory**

How a CPU works

- Executes a series of **instructions** stored in **memory**
- An instruction is stored as an **opcode** followed by 0 or more **arguments**

How a CPU works

- Executes a series of **instructions** stored in **memory**
- An instruction is stored as an **opcode** followed by 0 or more **arguments**
- CPU has several **registers**, each storing a single value

How a CPU works

- Executes a series of **instructions** stored in **memory**
- An instruction is stored as an **opcode** followed by 0 or more **arguments**
- CPU has several **registers**, each storing a single value
  - Memory locations inside the CPU

How a CPU works

- Executes a series of **instructions** stored in **memory**
- An instruction is stored as an **opcode** followed by 0 or more **arguments**
- CPU has several **registers**, each storing a single value
  - Memory locations inside the CPU
  - Faster to access than main memory

How a CPU works

- Executes a series of **instructions** stored in **memory**
- An instruction is stored as an **opcode** followed by 0 or more **arguments**
- CPU has several **registers**, each storing a single value
  - Memory locations inside the CPU
  - Faster to access than main memory
- Instructions can read and write values in **registers** and in **memory**, and can perform **arithmetic and logical** operations on them

How a CPU works

- Executes a series of **instructions** stored in **memory**
- An instruction is stored as an **opcode** followed by 0 or more **arguments**
- CPU has several **registers**, each storing a single value
  - Memory locations inside the CPU
  - Faster to access than main memory
- Instructions can read and write values in **registers** and in **memory**, and can perform **arithmetic and logical** operations on them
- The **program counter (PC)** register stores the address of the next instruction to execute

Registers on the 6502

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte
  - A number between 0 and 255

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte
  - A number between 0 and 255
- A = **accumulator**

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte
  - A number between 0 and 255
- A = **accumulator**
- X, Y = **index** registers

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte
  - A number between 0 and 255
- A = **accumulator**
- X, Y = **index** registers
- SP = **stack pointer** register

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte
  - A number between 0 and 255
- A = **accumulator**
- X, Y = **index** registers
- SP = **stack pointer** register
- PC = **program counter** register (16 bits)

Registers on the 6502

- 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
  - 1 byte
  - A number between 0 and 255
- A = **accumulator**
- X, Y = **index** registers
- SP = **stack pointer** register
- PC = **program counter** register (16 bits)
- **Status** register, composed of seven 1-bit **flags**

Assembly language

Assembly language

- Translates **directly** to machine code

Assembly language

- Translates **directly** to machine code
- I.e. 1 line of assembly = 1 CPU instruction

Assembly language

- Translates **directly** to machine code
- I.e. 1 line of assembly = 1 CPU instruction
- An **assembler** translates assembly to machine code

Assembly language

- Translates **directly** to machine code
- I.e. 1 line of assembly = 1 CPU instruction
- An **assembler** translates assembly to machine code
  - I.e. an assembler is a “compiler” for assembly language

Assembly language

- Translates **directly** to machine code
- I.e. 1 line of assembly = 1 CPU instruction
- An **assembler** translates assembly to machine code
  - I.e. an assembler is a "compiler" for assembly language
- Each CPU architecture has its own instruction set therefore its own assembly language

Our first assembly program

Our first assembly program

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

Our first assembly program

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

Try it out! `http://skilldrick.github.io/easy6502/`

Our first assembly program

Our first assembly program

```
LDA  #$01
```

Our first assembly program

```
LDA  #$01
```

- Store the value `01` (hexadecimal) into register A

Our first assembly program

```
LDA #$01
```

- Store the value `01` (hexadecimal) into register A
- `LDA` ("load accumulator") stores a value in register A

Our first assembly program

```
LDA #$01
```

- Store the value `01` (hexadecimal) into register A
- `LDA` ("load accumulator") stores a value in register A
- `#` denotes a literal number (as opposed to a memory address)

Our first assembly program

```
LDA #$01
```

- Store the value `01` (hexadecimal) into register A
- `LDA` ("load accumulator") stores a value in register A
- `#` denotes a literal number (as opposed to a memory address)
- `$` denotes hexadecimal notation

Our first assembly program

Our first assembly program

```
STA  $0200
```

Our first assembly program

```
STA $0200
```

- Write the value of register A into memory address `0200` (hex)

Our first assembly program

```
STA  $0200
```

- Write the value of register A into memory address `0200` (hex)
- `STA` ("store accumulator") copies the value of register A into main memory

Our first assembly program

```
STA $0200
```

- Write the value of register A into memory address `0200` (hex)
- `STA` ("store accumulator") copies the value of register A into main memory
- Note that address is a **16-bit** number (2 bytes, 4 hex digits)

Our first assembly program

```
STA $0200
```

- Write the value of register A into memory address `0200` (hex)
- `STA` ("store accumulator") copies the value of register A into main memory
- Note that address is a **16-bit** number (2 bytes, 4 hex digits)
- In this emulator the display is "memory mapped", with 1 byte per pixel, starting from address `0200`
  - Real systems are usually more complicated than this!

Assembly to machine code

Assembly to machine code

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

Assembly to machine code

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

```
A9 01
8D 00 02
A9 05
8D 01 02
A9 08
8D 02 02
```

Assembly to machine code

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

```
A9 01
8D 00 02
A9 05
8D 01 02
A9 08
8D 02 02
```

Note that the 6502 is **little endian**

Assembly to machine code

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

```
A9 01
8D 00 02
A9 05
8D 01 02
A9 08
8D 02 02
```

Note that the 6502 is **little endian**

- In 16-bit values, the “low” byte comes before the “high” byte

Assembly to machine code

```
LDA  #$01
STA  $0200
LDA  #$05
STA  $0201
LDA  #$08
STA  $0202
```

```
A9 01
8D 00 02
A9 05
8D 01 02
A9 08
8D 02 02
```

Note that the 6502 is **little endian**

- In 16-bit values, the "low" byte comes before the "high" byte
- Intel x86 is also little endian

Looping

Looping

- PC normally **advances** to the next instruction

Looping

- PC normally **advances** to the next instruction
- Some instructions **modify** the PC

Looping

- PC normally **advances** to the next instruction
- Some instructions **modify** the PC
- E.g. `JMP` (jump) sets the PC to the specified address

Looping

- PC normally **advances** to the next instruction
- Some instructions **modify** the PC
- E.g. `JMP` (jump) sets the PC to the specified address

```
INC $0200 ; add 1 to the value at address 0200
JMP $0600 ; jump back to beginning of program
```

Looping

- PC normally **advances** to the next instruction
- Some instructions **modify** the PC
- E.g. `JMP` (jump) sets the PC to the specified address

```
INC $0200 ; add 1 to the value at address 0200
JMP $0600 ; jump back to beginning of program
```

- In this emulator the program always starts at address `0600`
  - This may **not** be the case on other 6502-based systems!

Labels

Labels

- Don't use explicit jump locations in your code, it's not maintainable!

Labels

- Don't use explicit jump locations in your code, it's not maintainable!
- Can add a **label** to a line of code, by giving a name followed by a colon

Labels

- Don't use explicit jump locations in your code, it's not maintainable!
- Can add a **label** to a line of code, by giving a name followed by a colon
- Labels can then be used in instructions

Labels

- Don’t use explicit jump locations in your code, it’s not maintainable!
- Can add a **label** to a line of code, by giving a name followed by a colon
- Labels can then be used in instructions

```
start:
    INC $0200
    JMP start
```

Labels

- Don't use explicit jump locations in your code, it's not maintainable!
- Can add a **label** to a line of code, by giving a name followed by a colon
- Labels can then be used in instructions

```
start:
    INC  $0200
    JMP  start
```

- `start` is essentially a constant with value `$0600`

Labels

- Don't use explicit jump locations in your code, it's not maintainable!
- Can add a **label** to a line of code, by giving a name followed by a colon
- Labels can then be used in instructions

```
start:
    INC  $0200
    JMP  start
```

- `start` is essentially a constant with value `$0600`
- The assembled code is exactly the same as for the previous slide

Conditional branching

Conditional branching

```
   LDX #$08       ; set X=8
decrement:
   DEX            ; subtract 1 from X
   STX $0200      ; store X in top left pixel
   CPX #$03       ; compare X to 3
   BNE decrement  ; if not equal, jump
   STX $0201      ; store X in next pixel
   BRK            ; halt execution
```

Conditional branching

```
  LDX #$08        ; set X=8
decrement:
  DEX             ; subtract 1 from X
  STX $0200       ; store X in top left pixel
  CPX #$03        ; compare X to 3
  BNE decrement   ; if not equal, jump
  STX $0201       ; store X in next pixel
  BRK             ; halt execution
```

$X = 8$
do
  $X = X - 1$
  $\text{memory}[0200] = X$
while $X \neq 3$
$\text{memory}[0201] = X$

Conditional branching

Conditional branching

- Assembly language does not have **structured programming** constructs such as if/else, switch/case, for, while, etc.

Conditional branching

- Assembly language does not have **structured programming** constructs such as if/else, switch/case, for, while, etc.
- However all of these can be implemented using branch instructions

Conditional branching

- Assembly language does not have **structured programming** constructs such as if/else, switch/case, for, while, etc.
- However all of these can be implemented using branch instructions
- ... which is exactly how compilers implement them

Conditionals

Conditionals

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

Conditionals

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

```
    CPX #$01        ; compare X to 1
    BMI else        ; if X < 1, jump
    DEY             ; Y = Y - 1
    JMP end         ; skip over else block
else:
    INY             ; Y = Y + 1
end:
```

Conditionals

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

```
  CPX #$01        ; compare X to 1
  BMI else        ; if X < 1, jump
  DEY             ; Y = Y - 1
  JMP end         ; skip over else block
else:
  INY             ; Y = Y + 1
end:
```

if $X \geq 1$ **then**

$Y = Y - 1$

else

$Y = Y + 1$

end if

Subroutines

Subroutines

- `JSR` (jump to subroutine) works like `JMP`, but stores the current PC

Subroutines

- `JSR` (jump to subroutine) works like `JMP`, but stores the current PC
- `RTS` (return from subroutine) jumps back to the instruction after the `JSR`

Subroutines

- `JSR` (jump to subroutine) works like `JMP`, but stores the current PC
- `RTS` (return from subroutine) jumps back to the instruction after the `JSR`
- These are used to implement **function calls**

Subroutines

- `JSR` (jump to subroutine) works like `JMP`, but stores the current PC
- `RTS` (return from subroutine) jumps back to the instruction after the `JSR`
- These are used to implement **function calls**
- Return addresses are stored on a **stack**

Addressing modes

Addressing modes

- Immediate: `LDA #$42`

Addressing modes

- Immediate: `LDA #$42`
  - Load the literal value 42 (hex) into register A

Addressing modes

- Immediate: `LDA #$42`
  - Load the literal value 42 (hex) into register A
- Absolute: `LDA $42`

Addressing modes

- Immediate: `LDA #$42`
  - Load the literal value 42 (hex) into register A
- Absolute: `LDA $42`
  - Load the value stored at memory address 42 (hex) into register A

Addressing modes

- Immediate: `LDA #$42`
  - Load the literal value 42 (hex) into register A
- Absolute: `LDA $42`
  - Load the value stored at memory address 42 (hex) into register A
- That `#` makes a big difference!

Addressing modes

- Immediate: `LDA #$42`
  - Load the literal value 42 (hex) into register A
- Absolute: `LDA $42`
  - Load the value stored at memory address 42 (hex) into register A
- That `#` makes a big difference!
- Note that these actually assemble to **different** CPU instructions

Indexed addressing

```
LDA $0200,X
```

Indexed addressing

```
LDA $0200,X
```

- Look up the value stored at memory address

$$\texttt{0200} + (\text{value of X register})$$

and store it in A

Indexed addressing

```
LDA $0200,X
```

- Look up the value stored at memory address

  $$\texttt{0200} + (\text{value of X register})$$

  and store it in A
- Can also do `LDA $0200,Y`

Indexed addressing

```
LDA $0200,X
```

- Look up the value stored at memory address

  $$\texttt{0200} + (\text{value of X register})$$

  and store it in A
- Can also do `LDA $0200,Y`
- ... but **only** `X` and `Y` registers can be used for indexed addressing

Indexed addressing

```
  LDX #0        ; X=0
loop:
  TXA           ; A=X
  STA $0200,X   ; store A to 0200+X
  INX           ; X++
  JMP loop      ; loop forever
```

Indexed addressing

```
  LDX #0        ; X=0
loop:
  TXA           ; A=X
  STA $0200,X ; store A to 0200+X
  INX           ; X++
  JMP loop      ; loop forever
```

- Why does it stop $\frac{1}{4}$ of the way down?

Indexed addressing

```
  LDX #0        ; X=0
loop:
  TXA           ; A=X
  STA $0200,X   ; store A to 0200+X
  INX           ; X++
  JMP loop      ; loop forever
```

- Why does it stop $\frac{1}{4}$ of the way down?
- Hint: it stops after filling 256 pixels...

FALMOUTH UNIVERSITY

Workshop