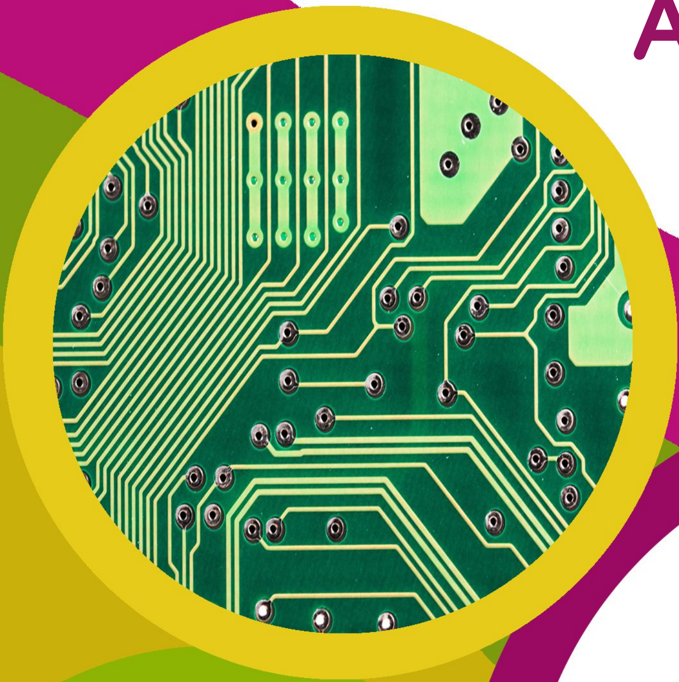
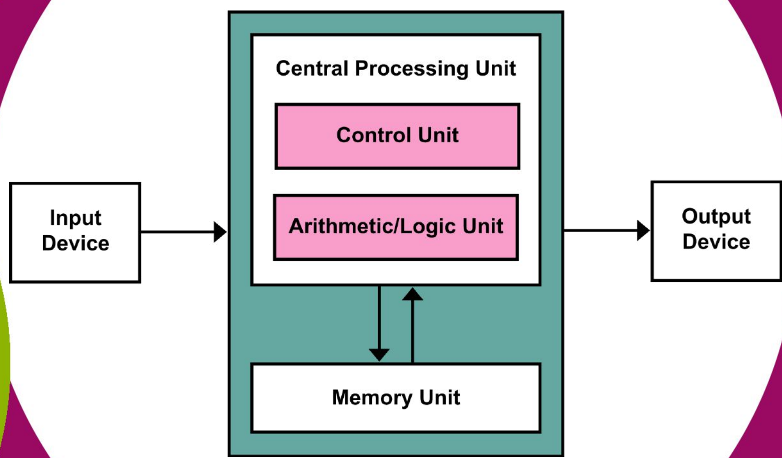
technocamps

Assembly Language Session Plan





Undeb Ewropeaidd / European Union
Llywodraeth Cymru / Welsh Government
Cronfa Gymdeithasol Ewrop
European Social Fund
Prifysgol Abertawe / Swansea University
Cardiff University / Prifysgol Caerdydd
Prifysgol Bangor University
Cardiff Metropolitan University / Prifysgol Metropolitan Caerdydd
i.t.wales
Prifysgol Aberystwyth University
Prifysgol Glyndŵr Wrecsam / Wrexham Glyndŵr University
University of South Wales / Prifysgol De Cymru

Introduction - 10 minutes

Hardware - 1 hour 30 minutes

Architecture - 30 minutes

What is Assembly Language? - 30 minutes

Little Man Computer Tasks - 1 hour 20 minutes

Sequences (Maths GCSE) - 30 minutes

Post-Day Questionnaires - 10 minutes

Note: These are estimated times, these will vary between classes, schools etc. so times will need to be adjusted accordingly.

Total: 4 hours 40 minutes

Preparation

1. Ensure all computers have access to https://peterhigginson.co.uk/LMC/
2. A small handheld whiteboard, a few blank A4 sheets, whiteboard markers, printouts for visualising activity.
3. Print out assembly language workbook, one for each student attending workshop.

Learning Outcomes

1. To gain a better understanding of the fetch-decode-execute cycle.
2. To develop a strong understanding of programming in assembly language.
3. To develop understanding of sequences (mathematics GCSE topic) through assembly language programming.

Attendee Prerequisites

1. No previous knowledge of Assembly Language programming required.

Session Plan Key

In this session plan we use the following colours to differentiate the types of activities:

- **Yellow - Explain.** Teachers should explain the slide/example to the class.
- **Green - Discuss.** Teachers should start an open discussion with the class to get them to feedback some answers/ideas.
- **Purple - Activity.** Students are expected to complete an activity whether it be in their workbooks or on the computer, followed by a discussion of their solutions.
- **Green - Introduction/Conclusion.** The introduction/conclusion is also colour coded green. Teachers should hand out materials in the introduction and conclude the day and collect materials at the end.

Introduction

Begin with introductions, and a brief explanation of the Technocamps programme, before handing out pre-day questionnaires to be filled out by the students and teacher.

Explain: Surface Pro 5 vs. MacBook Pro 2017

Play the marketing videos of the Surface Pro 5 and the MacBook Pro 2017. Explain that the students will need to choose which of the laptops they prefer and give reasons why.

Activity: Surface Pro 5 vs MacBook Pro 2017

Students should decide which of the two laptops they prefer and justify why, in their workbooks.

Explain: Marketing Nonsense

The videos the students watched were full of marketing buzzwords that are designed to sell laptops and technology but which are often misleading or don't mean as much as people first think.

"This is the fastest surface pro ever" doesn't mean much if the performance increase is only .1%
Similarly "the best retina display in a MacBook" can also be misleading and discussing how the technology works does little to show users how good a display the laptop has.

As a result, it is often better to look at the technical specifications of any piece of technology and use these to gain a better understanding of what you are actually getting.

Explain: Internal Components

We are going to open up a desktop, show you the most important components and then install them one at a time. By the end we should have a fully functional PC.
We should also understand more about the technology inside computers and be able to make informed decisions when buying tech.

Explain: Motherboard

The motherboard is potentially the most important component with a computer. It is a PCB that houses most of the essential parts of a computer system. It is also where most of the connections between the computer and external devices are contained.

For today, we have pre-installed the motherboard. This is just to avoid damaging it when we brought it here as well as ensuring we won't short any connections later on when we power up the system.

Extension Activity: Seating a CPU

This is an optional task if suitable for the class.
Show the class a CPU chip and then ask for two sensible volunteers to come up to the front and seat them.

Explain: CPU

The CPU is the brains of the computer. On most personal computers, the CPU is housed in a single chip called a microprocessor. The CPU contains the circuitry which processes the instructions when running any computer program. The two most important things to worry about when comparing CPUs is the clock speed and the number of cores.

The clock speed of a CPU is measured in hertz (Hz) and measures how many instructions a CPU can execute per second. It is analogous to the speed of a checkout cashier, the faster the cashier the more items it can scan a second.
A core is the part of a CPU which receives instructions and performs calculations. The more cores a CPU has, the more instructions and calculations it can perform simultaneously. The number of cores is analogous to the number of checkouts open at a super market. More checkouts open means more people can be served simultaneously.

Discuss: Parallelisation

If you are the only person in the supermarket and you get to the checkouts, would you rather have multiple checkouts open or one checkout open with speedy Gonzales as the cashier?

Certain tasks can not be split up efficiently over multiple cores. For example, if you tried to check out your shopping over multiple tills then the time taken for all the items to be scanned may be reduced but you would still have to go to each individual till to pay and pack up into your trolley. In the long run this will take more time than just using a single checkout.
As a result, high clock speeds and multiple cores need to be used effectively in conjunction for very fast computing.

Activity: Installing a Fan

Explain how the CPU fan sits directly on top of the CPU and is attached to the motherboard. Ask for two volunteers to come to the front and install the fans and the power cable for the fan.

Explain: Cooling

The engine of a car produces a lot of heat as it burns fuel. As an engine gets hotter it starts running less efficiently and if it gets too hot it will break. To stop this from happening, cars have massive radiators to cool the engine. The radiators in turn are cooled by air flowing through them.
Many PC components work similarly. As they get hotter they work less efficiently and if they get too hot they can damage the chips inside which can break the component. This is especially relevant for CPUs and GPUs.

So we need to cool our system. This is usually done with airflow and fans but can be done using liquid cooling loops and radiators.

Activity: Installing RAM

Show the students two RAM sticks. Explain how they are shaped with notches that fit into the RAM slots. Ask for two volunteers to come to the front and install the sticks.

Explain: Memory

Your computer needs to be able to store and access data so that programs can be properly run.

Cache memory is like a goldfish brain. Very small but super quick to access. Also if your system loses power or is shut down, the data stored in the cache is lost. This is known as volatile storage.
RAM, random access memory, is like sheets of paper out on your desk. You can store more data on the sheets than you could in a goldfish brain but it takes longer to access. Also if you don't file away your sheets. After using them they will be lost, similar to the goldfish brain.
Mass storage devices are things like solid state drives and hard drives. They are analogous to a file or a folder. They can hold loads of data but take much more time to access the information in them than loose sheets of paper or a goldfish brain. Most importantly they do not lose the data they contained within them after a shutdown or power loss. This is known as non-volatile storage.

Memory capacity is measured in bytes.
Having larger capacity RAM means more space for instructions that can be stored closer to the processor at any one time which reduces the amount of time spent swapping data in and out of RAM.

Activity: Memory

Students should fill out the section in their workbooks on memory which includes volatile/non-volatile memory and memory storage amounts.

Explain: PCI

PCI stands for peripheral component interconnect. These are used inside PCs for connecting peripherals such as dedicated sound cards, graphic cards or wireless internet cards.

Expansion slots allow the lifetime of a system to be extended as new technology becomes available such as newer graphics cards for better visuals and dedicated ethernet cards for greater data transfer rates.

Activity: Installing a Hard Drive

Show the students a hard drive and explain how much data it can store. Ask for two volunteers to come to the front and install it.

Explain: SATA

The hard drive wasn't connected directly to the motherboard, there was no slot for it. Instead it was connected using a SATA cable. This cable is designed to transfer lots of data from hard drives or optical drives to the CPU.

Explain: I/O Device Ports

Input devices such as keyboard and mouse provide a way for the user to input data into the processor and give commands.
Output devices like a monitor present the results of any processing to the user.

Extension Explain: Overclocking

Overclocking increases the operating speed of a given component.
The target of overclocking is increasing the performance of a major chip or subsystem, such as the main processor or graphics controller, but other components, such as system memory (RAM) or system buses (generally on the motherboard), are commonly involved.

Discuss: eMac (2005) vs. iMac Pro

Bring up the two computers and run through the slides explaining the differences in hardware. This is a good demonstration of how technology gets better.

Activity: Compare Technology

Students should choose either the Xbox One X and the Playstation 4 Pro or the HP Envy 13 (2018) and the HP Pavilion 15-cs1006na, google the chosen items and then compare their tech specs. Decide which is better and why.

Explain: Architecture

Just like the architecture of a building, computer architecture is the way that a computer is designed to function in terms of hardware.
The most common architecture is known as Von Neumann architecture.
This architecture is made up of:

- CPU - Control unit, Arithmetic unit and registers
- Memory unit - RAM
- Buses - Data/address/control
- Input device - mouse, keyboard
- Output device - monitor, speakers

Von Neumann: This stores both the instructions and the data within the same memory addresses and uses the same bus for both.
Harvard: This has separate memory addresses for instructions and data meaning it can run a program and access data simultaneously.

Activity: Von Neumann Architecture

Students to write the parts of the Von Neumann architecture next to their corresponding label in the diagram given the slides and in their workbooks.

Explain: Von Neumann vs Harvard

Von Neumann architecture is more flexible than Harvard architecture. We have to decide how much memory we are dedicating to instructions and how much we are dedicating to data in Harvard architecture.
If we set up or system to be able to store lots of data and few instructions, but then we are required to run a program with little data and lots of instructions, we may not be able to.
This does not occur in Von Neumann architecture as the same memory addresses are used for both data and instructions.

Harvard architecture is in theory faster than Von Neumann as it can access data and instructions simultaneously.

Harvard architecture is more costly to develop due to having two buses working simultaneously. This complicates the control unit adding to the cost.

Von Neumann architecture is used in general purpose computers that will be used for many different purposes.
Harvard architecture is used in embedded systems that perform only a few functions like a burglar alarm.

Activity: Flexibility

Students to fill out the section on flexibility of Von Neumann architecture vs Harvard architecture in their workbooks.

Explain: What Is Assembly Language?

Assembly language is a low-level programming language which uses an assembler to convert a program into machine code. Assembly languages usually use short mnemonics as instructions and each one is specific to the computer architecture and operating systems.
Assembly languages are considered to be low-level because they are very close to machine languages. They are only one step removed from a computer's machine language.

Explain: Why Assembly?

A CPU cannot directs read source code. Different CPUs may have different architecture and each different architecture has its own machine language.
This prevents direct source code to machine code translation - we need to use an assembly language to assemble the code which bridges the gap.
For example, a piece of Python code assembled to run on a 64 bit Windows machine will not have the same instruction set as the Python code assembled to run on a 32 bit linux machine.

On the slides there is a simple one line program in Python. Internally it is converted to an Assembly language. If we wrote the same code directly in the assembly language we can see it only takes three instructions yet the two programs provide the same output.

Low-level languages are especially useful when speed of execution is critical or when writing software which interfaces directly with the hardware, e.g. device drivers.
Example: the Voyager space prove launched in 1977 is programmed using an old assembly language. NASA are struggling to find anyone who still has a working knowledge of the language to keep it going.

Activity: What Is an Assembly Language?

Students to write down what Assembly languages are and when they are useful in their workbooks.

Explain: Little Man Computer (LMC)

LMC is a simulator which mimics von Neumann architecture.
Everything in a computers memory is data. Although programs may seem different from data, they are treated in exactly the same way: the computer executes a program, instruction by instruction.
These instructions are the 'data' of the fundamental program cycle:

1. Fetch the next instruction
2. Decode it
3. Execute it

Then the next program cycle starts which will process the next instruction. Even the location of the next instruction is just data.

Explain: The LMC Environment

1. **Accumulator -** This is like the active memory of the simulator. The majority of our instructions will modify the contents of the accumulator.
2. **Program Counter -** This shows the current memory location that the processor is running.
3. **Instruction and Address register -** This shows which type of instruction is being used and which memory address it is being used on.
4. **Memory Addresses -** These are the RAM addresses which are used to store instructions and data.
5. **Input Box -** This is where the user inputs are stored initially before being copied to the accumulator.
6. **Output Box -** This is where a value is copied to from the accumulator to display to the user.

Activity: Fill in the Blanks

Students to fill out the heading names of the environment of LMC in their workbooks.

Explain: Input, Output, Halt

Explain the Input, Output and Halt instructions, their mnemonics and code as well as their functions and what happens after these instructions are completed.

Discuss: Visualising a Program Running

Print off the extra resources for this activity. The instructions will each be on an individual sheet.

The RAM Addresses: You will either need to use 6 boxes labelled 0 to 5 or just lay them out on a table or sellotape to a wall etc. with space for instructions underneath.

Accumulator: Get a student to be the accumulator i.e. the working working memory. This person will be handed numbers on A4 pages and will then hand these to the arithmetic unit if a calculation is required.

Arithmetic Unit: Get a student to be the arithmetic unit. This person is required to perform simple calculations when required, such as adding 1 to the counter.

Program Counter: This person can stand next to the whiteboard and update the counter each time it is given a new value by the bus.

The Bus: Again another student. For this task we simplify and just use a single bus whenever we are moving data, they must have a small handheld whiteboard to write the values on.

Input: The teacher can be the “Input Box” and decide on a value to input.

Output: The Output box can be an area on the whiteboard for writing the final outputted values.

Activity: Visualising a Program Running

First line up the instructions in the following addresses:

00: INP 901
01: OUT 902
02: HLT 000

The Counter and Accumulator should begin with 0 as their number. Once this is set up we can start the activity.

Step 1:
Bus copies value from Counter, Value is 0 so the bus copies the instruction from address 00.
The bus then shows the instruction to the Control Unit (rest of the class). They decide what the instruction is and what needs to be done.
They will realise it's and Input and will need to ask for an input. In this case, the teacher can be the "user" and write a number on a blank A4 page.
The bus will then copy this to the accumulator.
This step is finished so the bus copies the value from the counter, takes it to the arithmetic unit who adds 1 and tells the bus the new value, who returns it to the counter and then the first step is finished.

In Short:
1. Bus copies Value from Counter. Value = 0
2. Bus copies instruction from Address 00
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and asks for Input
5. Bus copies input from Input Box (Teacher)
6. Bus copies value into Accumulator. Accumulator records the value
7. Bus copies value from Counter
8. Bus takes it to Arithmetic Unit. The Arithmetic Unit adds 1
9. Bus returns to Counter who updates with new value. Counter = 1

Activity: Visualising a Program Running

Step 2:

1. Bus copies value from Counter. Value = 1
2. Bus copies instruction from Address 01
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and tells bus to copy value from Accumulator to Output
5. Bus copies Value from Accumulator to Output
6. Bus copies value from Counter
7. Bus takes it to Arithmetic Unit. The Arithmetic Unit adds 1
8. Bus returns to Counter who updates with new value. Counter = 2

Step 3:

1. Bus copies value from Counter. Value = 2
2. Bus copies instruction from Address 02
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and ends the program

Discuss: What Happened?

Ask the students to summarise what the program did, what instructions were used, what did the arithmetic unit do?
Emphasise that we are copying values from the memory addresses to the bus and bus to the accumulator. i.e. at the end of the program all the instructions and data are still in the memory addresses.

Explain: Storing, Loading, DAT

Explain the Store, Load and DAT instructions, their mnemonics and code as well as their functions and what happens after these instructions are completed. Specifically explain the DAT instruction for reserving memory locations.

Explain: Input & Print a Number

The slides show a simple input and print program written in Python. An equivalent program is written in Assembly. We are going to run through this Assembly program by hand on the board to once again demonstrate how a program is executed.

Activity: Running a Program

Convert the Assembly instructions into their corresponding codes and load them into the correct memory addresses. Then start the Counter and Accumulator at 0 and begin executing the instructions one at a time.

Activity: Running a Program

Step 1:

1. Bus copies Value from Counter. Value = 0
2. Bus copies instruction from Address 00
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and asks for Input
5. Bus copies input from Input Box (Teacher)
6. Bus copies value into Accumulator. Accumulator records the value
7. Bus copies value from Counter
8. Bus takes it to Arithmetic Unit. The Arithmetic Unit adds 1
9. Bus returns to Counter who updates with new value. Counter = 1

Step 2:

1. Bus copies value from Counter. Value = 1
2. Bus copies instruction from Address 01
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and tells bus to copy value from Accumulator to Memory Address 05
5. Bus copies Value from Accumulator to Memory Address 05
6. Bus copies value from Counter
7. Bus takes it to Arithmetic Unit. The Arithmetic Unit adds 1
8. Bus returns to Counter who updates with new value. Counter = 2

Step 3:

1. Bus copies value from Counter. Value = 2
2. Bus copies instruction from Address 02
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and tell bus to copy value from Memory Address 05 into Accumulator
5. Bus copies value from Address 05 into Accumulator
6. Bus copies value from Counter
7. Bus takes it to Arithmetic Unit. The Arithmetic Unit adds 1
8. Bus returns to Counter who updates with new value. Counter = 3

Activity: Running a Program

Step 4:

1. Bus copies value from Counter. Value = 3
2. Bus copies instruction from Address 03
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and tells bus to copy blue from Accumulator to Output
5. Bus copies Value from Accumulator into Output
6. Bus copies value from Counter
7. Bus takes it to Arithmetic Unit. The Arithmetic Unit adds 1
8. Bus returns to Counter who updates with new value. Counter = 4

Step 5:

1. Bus copies value from Counter. Value = 4
2. Bus copies instruction from Address 04
3. Bus shows Control Unit (rest of class)
4. Control Unit decodes and ends the program

Discuss: What Happened?

Ask the students to summarise what the program did. Why were variables stored in box 05, what would happen if we had more commands, where would the variables be stored?

Explain: How to Write Assembly Programs

Follow the slides to run through an example of how to think about constructing Assembly programs.

Activity: Storing and Loading 1

Disclaimer: For the following solutions in the session plan, there may be other ways of solving them and your pupils may find other some, especially when getting to the more advanced tasks.

Create a program which takes in and stores two inputs from the user and outputs the first input followed by the second input.

Assembly Language Code

```
         INP                00 INP
         STA First          01 STA 09
         INP                02 INP
         STA Second         03 STA 10
         LDA First          04 LDA 09
         OUT                05 OUT
         LDA Second         06 LDA 10
         OUT                07 OUT
         HLT                08 HLT
First    DAT                09 DAT 00
Second   DAT                10 DAT 00
```

Activity: Storing and Loading 2

Create a program which takes and stores four inputs from the user and always outputs the third input.

Assembly Language Code

```
         INP                00 INP
         STA First          01 STA 11
         INP                02 INP
         STA Second         03 STA 12
         INP                04 INP
         STA Third          05 STA 13
         INP                06 INP
         STA Fourth         07 STA 14
         LDA Third          08 LDA 13
         OUT                09 OUT
         HLT                10 HLT
First    DAT                11 DAT 00
Second   DAT                12 DAT 00
Third    DAT                13 DAT 00
Fourth   DAT                14 DAT 00
```

Activity: Storing and Loading 3

Create a program which takes in three inputs and outputs them in reverse order.

Assembly Language Code

```
        INP                  00 INP
        STA First            01 STA 11
        INP                  02 INP
        STA Second           03 STA 12
        INP                  04 INP
        OUT                  05 OUT
        LDA Second           06 LDA 12
        OUT                  07 OUT
        LDA First            08 LDA 11
        OUT                  09 OUT
        HLT                  10 HLT
First   DAT                  11 DAT 00
Second  DAT                  12 DAT 00
```

Explain: Addition and Subtraction

Explain the Addition and Subtraction instructions, their mnemonics and code as well as their functions and what happens after these instructions are completed.

Activity: Addition and Subtraction 1

Create a program which takes in and stores two inputs from the user and outputs the sum of them.

Assembly Language Code

```
        INP                  00 INP
        STA First            01 STA 06
        INP                  02 INP
        ADD First            03 ADD 06
        OUT                  04 OUT
        HLT                  05 HLT
First   DAT                  06 DAT 00
```

Activity: Addition and Subtraction 2

Create a program which takes in three numbers and stores them and then outputs the sum of the first two numbers with the third subtracted.

Assembly Language Code

```
         INP                 00 INP
         STA First           01 STA 11
         INP                 02 INP
         STA Second          03 STA 12
         INP                 04 INP
         STA Third           05 STA 13
         LDA First           06 LDA 11
         ADD Second          07 ADD 12
         SUB Third           08 SUB 13
         OUT                 09 OUT
         HLT                 10 HLT
First    DAT                 11 DAT 00
Second   DAT                 12 DAT 00
Third    DAT                 13 DAT 00
```

Activity: Addition and Subtraction 1

Create a program which takes in a number, doubles it and then outputs the result.

Assembly Language Code

```
        INP                  00 INP
        STA Number           01 STA 05
        ADD Number           02 ADD 05
        OUT                  03 OUT
        HLT                  04 HLT
Number  DAT                  05 DAT 00
```

Activity: Addition and Subtraction 2

Create a program which takes in a number, multiplies it by eight and then outputs the result.

Assembly Language Code

```
         INP                 00 INP
         STA First           01 STA 09
         ADD First           02 ADD 09
         STA First           03 STA 09
         ADD First           04 ADD 09
         STA First           05 STA 09
         ADD First           06 ADD 09
         OUT                 07 OUT
         HLT                 08 HLT
First    DAT                 09 DAT 00
```

Explain that this could be done in multiple ways. One way is to use the add command seven times i.e. First + First + … + First.
The code above works differently. Initially it stores First, then adds First to itself. Then it "overwrites" the old First value with the new value in the accumulator = First + First. It then adds the new First value to itself and stores the new value in the accumulator in the First variable. It then does one final addition with the newest updated value of First before outputting the final answer. An example with our input being five is shown below.

INP 5	Accumulator = 5	First = 0
STA First	Accumulator = 5	First = 5
ADD First	Accumulator = 10	First = 5
STA First	Accumulator = 10	First = 10
ADD First	Accumulator = 20	First = 10
STA First	Accumulator = 20	First = 20
ADD First	Accumulator = 40	First = 20
OUT		
HLT		

Activity: Addition and Subtraction Challenge

Create a program which takes in a number and multiplies it by forty.

Assembly Language Code

```
       INP
       STA First
       STA Second
       ADD First
       STA First
       ADD First
       STA First
       ADD Second
       STA First
       ADD First
       STA First
       ADD First
       STA First
       ADD First
       OUT
       HLT
First  DAT
Second DAT
```

```
00 INP
01 STA 16
02 STA 17
03 ADD 16
04 STA 16
05 ADD 16
06 STA 16
07 ADD 17
08 STA 16
09 ADD 16
10 STA 16
11 ADD 16
12 STA 16
13 ADD 16
14 OUT
15 HLT
16 DAT 00
17 DAT 00
```

This example highlights why we add and update a variable in multiplication. By updating the variable we have reduced the 39 lines of "ADD First" to only 17 total lines of code which is much more efficient. Again an example is shown below with our input being seven.

INP 7	Accumulator = 7	First = 0	Second = 0
STA First	Accumulator = 7	First = 7	Second = 0
STA Second	Accumulator = 7	First = 7	Second = 7
ADD First	Accumulator = 14	First = 7	Second = 7
STA First	Accumulator = 14	First = 14	Second = 7
ADD First	Accumulator = 28	First = 14	Second = 7
STA First	Accumulator = 28	First = 28	Second = 7
ADD Second	Accumulator = 35	First = 28	Second = 7
STA First	Accumulator = 35	First = 35	Second = 7
ADD First	Accumulator = 70	First = 35	Second = 7
STA First	Accumulator = 70	First = 70	Second = 7
ADD First	Accumulator = 140	First = 70	Second = 7
STA First	Accumulator = 140	First = 140	Second = 7
ADD First	Accumulator = 280	First = 140	Second = 7
OUT			
HLT			

Explain: Branch Always

At this point you should give each student a copy of the cheat sheet so they can see the command tables. Then continue by explaining the Branch Always instruction, its mnemonic and code as well as its function and what happens after this instruction has been completed.

Activity: Looping 1

Create a program which allows the user to input numbers indefinitely and outputs each number.

Assembly Language Code

```
      INP              00 INP
      OUT              01 OUT
      BRA 00           02 BRA 00
```

Activity: Looping 2

Create a program which allows the user to input numbers indefinitely and outputs the running total after each entry.

Assembly Language Code

```
        INP            00 INP
        ADD Total      01 ADD 05
        OUT            02 OUT
        STA Total      03 STA 05
        BRA 00         04 BRA 00
Total   DAT 0          05 DAT 00
```

Explain: Branch If Zero or Positive

Explain the conditional Branching instructions, their mnemonics and code as well as their function and what happens after these instructions are completed.

Explain: Comparing Values in Assembly

In LMC we don't have "if statements" like we have in Python or other languages for comparing. The only way to branch based on a condition is to do a subtraction and then branch based on the result. For example, if we want to output the biggest number of 2 and 5, we would take 2, subtract 5 and then check if the answer is positive (or zero) or negative. Here's an example:

```
          LDA two
          SUB five
          BRP outputTwo
          LDA five
          OUT
          HLT
outputTwo LDA two
          OUT
          HLT
two       DAT 2
five      DAT 5
```

Load 2, subtract 5 and check the result. If it is positive, jump to instruction "outputTwo"

Otherwise, carry on, load 5 and output it before stopping the program.

If it was positive, load 2 and output it before stopping the program.

Activity: Conditional Branching 1

Create a program which allows the user to input two numbers and outputs the smallest number. Hint: if you do a - b and the number is positive, then a is bigger than b.

Assembly Language Code

```
             INP                        00 INP
             STA First                  01 STA 12
             INP                        02 INP
             STA Second                 03 STA 13
             SUB First                  04 SUB 12
             BRP OutPutSecond           05 BRP 09
             LDA First                  06 LDA 12
             OUT                        07 OUT
             HLT                        08 HLT
OutPutSecond LDA Second                 09 LDA 13
             OUT                        10 OUT
             HLT                        11 HLT
First        DAT                        12 DAT 00
Second       DAT                        13 DAT 00
```

Activity: Conditional Branching 2

Create a program which repeatedly allows the user to input two numbers and checks if they are equal. Only output the number if they are equal.

Assembly Language Code

```
         INP              00 INP
         STA First        01 STA 09
         INP              02 INP
         SUB First        03 SUB 09
         BRZ Equal        04 BRZ 06
         BRA 00           05 BRA 00
Equal    LDA First        06 LDA 09
         OUT              07 OUT
         BRA 00           08 BRA 00
First    Dat              09 DAT 00
```

Activity: Conditional Branching 3

Create a program which repeatedly takes in inputs and only outputs them if they are zero.

Assembly Language Code

```
          INP                00 INP
          BRZ Num1=0         01 BRZ 03
          BRA 00             02 BRA 00
Num1=0    OUT                03 OUT
          BRA 00             04 BRA 00
```

Activity: Conditional Branching 4

Create a program which outputs everything except zeroes.

Assembly Language Code

```
          INP                00 INP
          BRZ Num1=0         01 BRZ 03
          OUT                02 OUT
Num1=0    BRA 00             03 BRA 00
```

Activity: Conditional Branching Challenge

Create a program which allows the user to input two numbers and outputs the multiplication of the two numbers.

Assembly Language Code

```
        INP                 00 INP
        STA Total           01 STA 18
        STA First           02 STA 19
        INP                 03 INP
        SUB One             04 SUB 17
        STA Second          05 STA 20
Loop    LDA Total           06 LDA 18
        ADD First           07 ADD 19
        STA Total           08 STA 18
        LDA Second          09 LDA 20
        SUB One             10 SUB 17
        STA Second          11 STA 20
        BRZ Output          12 BRZ 14
        BRA Loop            13 BRA 06
Output  LDA Total           14 LDA 18
        OUT                 15 OUT
        BRA 00              16 BRA 00
One     DAT 1               17 DAT 01
Total   DAT                 18 DAT 00
First   DAT                 19 DAT 00
Second  DAT                 20 DAT 00
```

Explain: Sequences

In order to calculate the equation for a given sequence of numbers we must first look at the difference between them e.g.

Index term: 1 2 3 4 5 . . .

+2 +2 +2 +2

Number: 3 , 5 , 7 , 9 , 11 . . .

So the difference between each number is +2. So the number in front of the nth term in our equation must be 2 i.e. **2n**. The final step is to check if we need to add or subtract from our **2n**.

If we try inserting the index term into our nth term equation **2n** does the answer match up correctly? 2 x 1 = 2
So what should we add to correct this? **+1**
Therefore our equation must be : **2n + 1**
Does it work for all the values?

Discuss: Another Example

Run through the additional example together as a class, getting the students to answer each question to calculate the correct equation.

Activity: Sequences

For the following sequences:

a. Write out the nth term equation.

b. Calculate the 20th term of the sequence.

1. 7, 8, 9, 10, 11 ...
2. 3, 6, 9, 12, 15 ...
3. 12, 17, 22, 27, 32 ...
4. -6, -2, 2, 6, 10 ...
5. 3, -3, -9, -15, -21 ...

6. a. Write out the first 5 terms of the sequence given by 3n - 7.
   b. Calculate the 15th term of the sequence.

Discuss: Solutions

1a) n + 6
1b) 20 + 6 = 26

2a) 3n
2b) 3 x 20 = 60

3a) 5n + 7
3b) (5 x 20) + 7 = 107

4a) 4n - 10
4b) (4 x 20) - 10 = 70

5a) -6n + 9
5b) (-6 x 20) + 9 = -111

6a) -4, -1, 2, 5, 8
6b) (3 x 15) - 7 = 38

Activity: Implementing Sequences in LMC

Students to implement this nth term equation in LMC to produce the first 5 terms in the sequence: 5, 6, 7, 8, 9 …

Students should plan how to write this program in pairs. Encourage them to think about:
What is the nth term equation?
Would you need to use a loop?
What other variables would you need?
Hint: You will need to be adding or subtracting by 1, how could you implement this?

Discuss: Implementing Sequences in LMC

Ask the students how they attempted to create the sequence. Did any of them use a loop? Did they set any variables? If so, what were they?

Explain: The First Value

To get the first result we need to load the first index term = 1 , add 4 to it and then output it.

We always add 4 in our nth term equation so should store 4 as a variable called number2.

We also need to define a variable so we know which index term we are inserting into our equation.

```
            LDA term
            ADD number2
            OUT
StopProgram HLT
term        DAT 1
number2     DAT 4
```

Load 1 and add 4 to it. Then Output the Value 5

Stop the program. (StopProgram is a reference to the Halt which will be useful later)

Define Variables: term = 1
number2 = 4

Explain: Looping for More Values

We need to add one to the term variable before we calculate the next number in our sequence. To do this we define a variable called one which we add to the term variable. We use a loop to repeat the previous calculations and output each new number in the sequence.

```
            LDA term        00
            ADD number2     01
            OUT             02
            LDA term        03
            ADD one         04
            STA term        05
            BRA 00          06
StopProgram HLT             07
term        DAT 1           08
one         DAT 1           09
number2     DAT 4           10
```

Loop back to the first line (00) Always.

Increase the current term by one and store it again.

Explain: Only Outputting the First 5 Values

If we want to stop the loop after 5 values have been output we need to compare our index term variable to a limit. Once our term reaches the same value as the limit, we halt the program.

```
            LDA term          00
            ADD number2       01
            OUT               02
            LDA term          03
            ADD one           04
            STA term          05
            SUB limit         06
            BRZ StopProgram   07
            BRA 00            08
StopProgram HLT               09
term        DAT 1             10
one         DAT 1             11
number2     DAT 4             12
limit       DAT 6             13
```

Check if the term and limit are the same, if so jump to the HLT instruction.

Note: The variable limit is set to 6. Is this correct?
Yes, we increment the loop counter before checking how many times we have looped

Activity: Creating Your Own Sequences

You can now use this code as a starting point for creating your own sequences. What would we change to make the sequence n + 8 for example?

The students should answer the question in their workbooks and try running the code in LMC to see if they are correct.

n - 7
2n + 4
2n - 6
3n + 8
8n - 3

```
            LDA term
            ADD number2
            OUT
            LDA term
            ADD one
            STA term
            SUB limit
            BRZ StopProgram
            BRA 00
StopProgram HLT
term        DAT 1
one         DAT 1
number2     DAT 4
limit       DAT 6
```

Discuss: Solutions

For these tasks, the number before the n is achieved by repeating an "ADD term" instruction the respective amount of times. So for 2n + 4 the only thing we would do to change the code is insert and ADD term after LDA term, this is then doubling our term from 1 to 2, hence giving us 2n.

Code for 2n + 4:

Assembly Language Code

LDA term	00 LDA 11
ADD term	01 ADD 11
ADD number2	02 ADD 13
OUT	03 OUT
LDA term	04 LDA 11
ADD one	05 ADD 12
STA term	06 STA 11
SUB limit	07 SUB 14
BRZ StopProgram	08 BRZ 10
BRA 00	09 BRA 00
StopProgram HLT	10 HLT
term DAT 1	11 DAT 01
one DAT 1	12 DAT 01
number2 DAT 4	13 DAT 04
limit DAT 6	14 DAT 06

Discuss: Solutions

2n - 6:
Number of "ADD term" instructions = 1
number2 = -6

3n + 8:
Number of "ADD term" instructions = 2
number2 = -3

8n - 3:
Number of "ADD term" instructions = 7
number2 = -3

In order to find a given nth term, we only need to change the limit to n + 1. So if we wanted the 20th term in a sequence, we would just change the limit to 21 and write the final value given.

Activity: Advanced LMC 1

Create a program which takes in input and outputs the positive value. If the input is negative, you output the positive so if we input -3 we would output 3.

Assembly Language Code

```
         INP                 00 INP
         BRP Positive        01 BRP 05
         STA First           02 STA 07
         SUB First           03 SUB 07
         SUB First           04 SUB 07
Positive OUT                 05 OUT
         BRA 00              06 BRA 00
First    DAT                 07 DAT 00
```

Activity: Advanced LMC 2

Create a program which takes and input, outputs that value and then counts down and outputs every value until it reaches 0 (or counts up to 0 if the value input is negative.

Assembly Language Code

```
         INP                      00 INP
Output   OUT                      01 OUT
         BRP Positive             02 BRP 05
         ADD One                  03 ADD 10
         BRA CheckIfZero          04 BRA 06
Positive SUB One                  05 SUB 10
CheckIfZero BRZ FinalOutput       06 BRZ 08
         BRA Output               07 BRA 01
FinalOutput OUT                   08 OUT
         BRA 00                   09 BRA 00
One      DAT 1                    10 DAT 01
```

Activity: Advanced LMC 3

Create a program which takes two inputs and checks if they have the same sign (both positive or both negative). If they have the same sign output a zero, otherwise output a 1.

Assembly Language Code

```
          INP
          BRP Positive
          LDA Zero
          STA 98
          BRA nextInput
Positive LDA One
          STA 98
nextInput INP
          BRP Positive2
          LDA Zero
          STA 99
          BRA Subtract
Positive2 LDA One
          STA 99
Subtract SUB 98
          BRZ SameSign
          LDA One
SameSign OUT
          BRA 00
Zero      DAT 0
One       DAT 1
```

```
00 INP
01 BRP 05
02 LDA 19
03 STA 98
04 BRA 07
05 LDA 20
06 STA 98
07 INP
08 BRP 12
09 LDA 19
10 STA 99
11 BRA 14
12 LDA 20
13 STA 99
14 SUB 98
15 BRZ 17
16 LDA 20
17 OUT
18 BRA 00
19 DAT 00
20 DAT 01
```

Activity: Advanced LMC 4

Create a program which takes two inputs and returns the remainder if you divided the first by the second. (Don't worry about negative numbers, but zero by a number and dividing a number by zero should be considered.)

Assembly Language Code

```
         INP
         STA First
         INP
         STA Second
         BRZ DivideByZero
         LDA First
         BRZ ZeroDividedByx
Loop     STA First
         SUB Second
         BRP Loop
         LDA First
         OUT
         HLT
DivideByZero LDA NotGood
         OUT
         HLT
ZeroDividedByx OUT
         HLT
First    DAT
Second   DAT
NotGood  DAT 999
```

```
00 INP
01 STA 18
02 INP
03 STA 19
04 BRZ 13
05 LDA 18
06 BRZ 16
07 STA 18
08 SUB 19
09 BRP 07
10 LDA 18
11 OUT
12 HLT
13 LDA 20
14 OUT
15 HLT
16 OUT
17 HLT
18 DAT 00
19 DAT 00
20 DAT 999
```

Explain: The Fibonacci Sequence

Note: The Fibonacci sequence is made by adding the previous number to the current one, starting with 1:

1
0+1= **1**
1+1= **2**
2+1= **3**
3+2= **5**
5+3= **8**

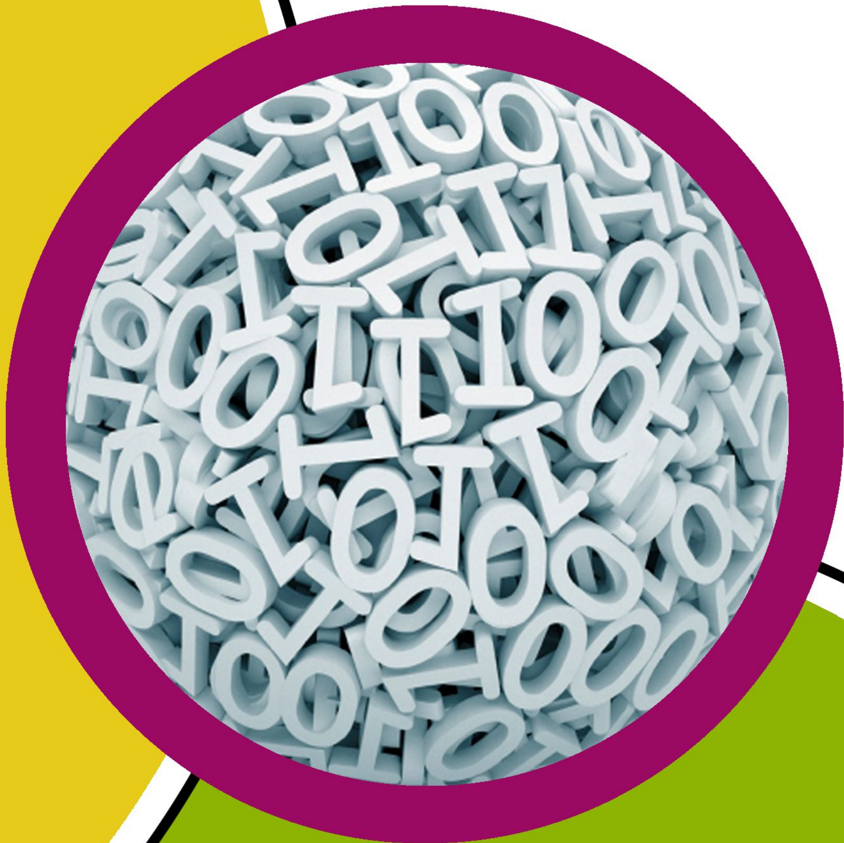
Activity: Very Advanced LMC

Create a program which takes in an input and outputs all of the numbers in the Fibonacci sequence up to that input number. The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21 . . .
You can set one variable to 1 at the beginning. No cheating!

Assembly Language Code

```
        INP
        STA Number
Loop    LDA CurrentNumber
        Sub Number
        BRZ OutputFinal
        BRP SkipOut
        LDA CurrentNumber
        OUT
SkipOut LDA CurrentNumber
        STA Temp
        ADD PreviousNumber
        STA CurrentNumber
        LDA Temp
        STA PreviousNumber
        LDA Number
        SUB PreviousNumber
        BRP Loop
        HLT
OutputFinal LDA
CurrentNumber
        OUT
        HLT
Number  DAT
PreviousNumber DAT
CurrentNumber DAT 1
Temp    DAT 00
One     DAT 1
```

```
00 INP
01 STA 21
02 LDA 23
03 SUB 21
04 BRZ 18
05 BRP 08
06 LDA 23
07 OUT
08 LDA 23
09 STA 24
10 ADD 22
11 STA 23
12 LDA 24
13 STA 22
14 LDA 21
15 SUB 22
16 BRP 02
17 HLT
18 LDA 23
19 OUT
20 HLT
21 DAT 00
22 DAT 00
23 DAT 01
24 DAT 00
25 DAT 01
```



technocamps

@Technocamps

Find us on Facebook