
MONTANA
STATE UNIVERSITY

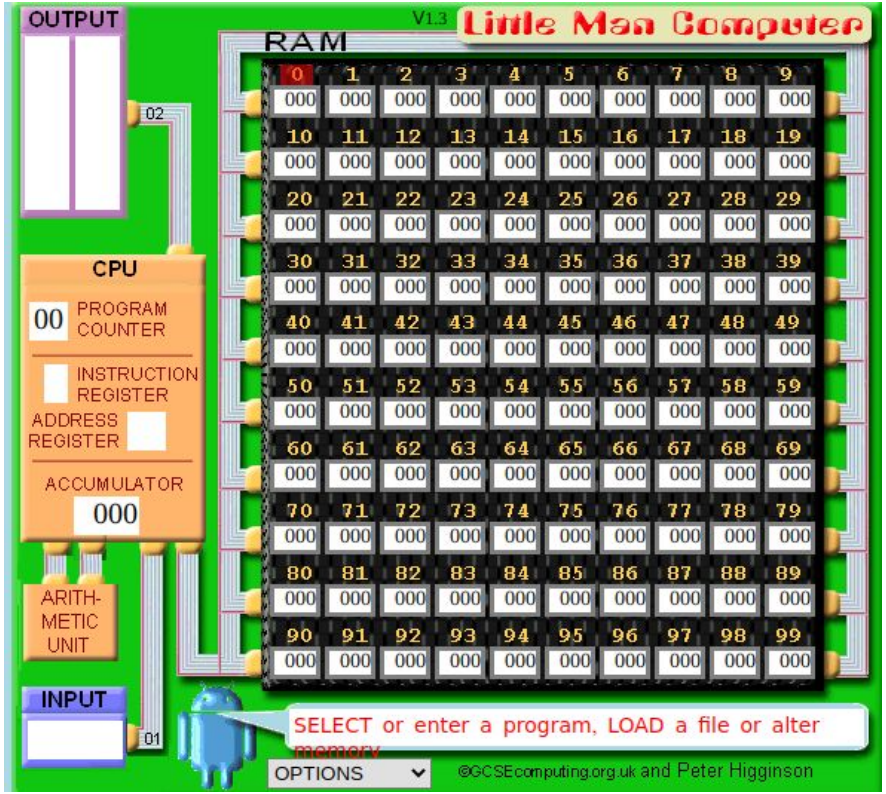
Using LMC Assembly

•••

Doing Things With The Little Man Computer

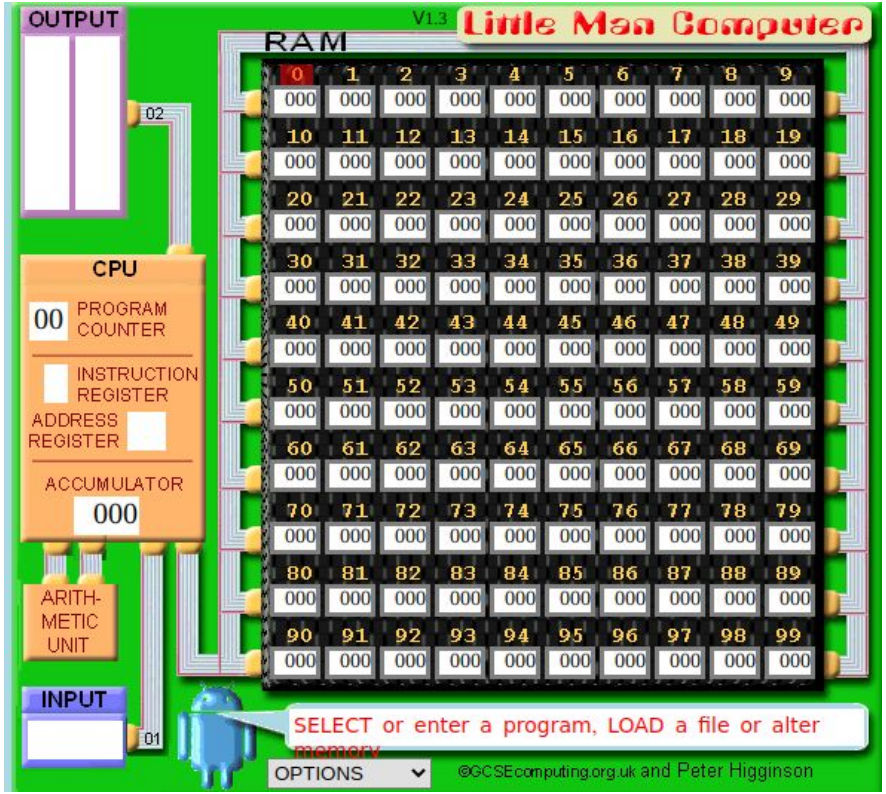
LMC Review

- Recall the LMC architecture
  - Program Counter
  - Instruction & Address registers
  - Accumulator register for work
  - Input/Output areas
  - An ALU
  - 100 memory slots



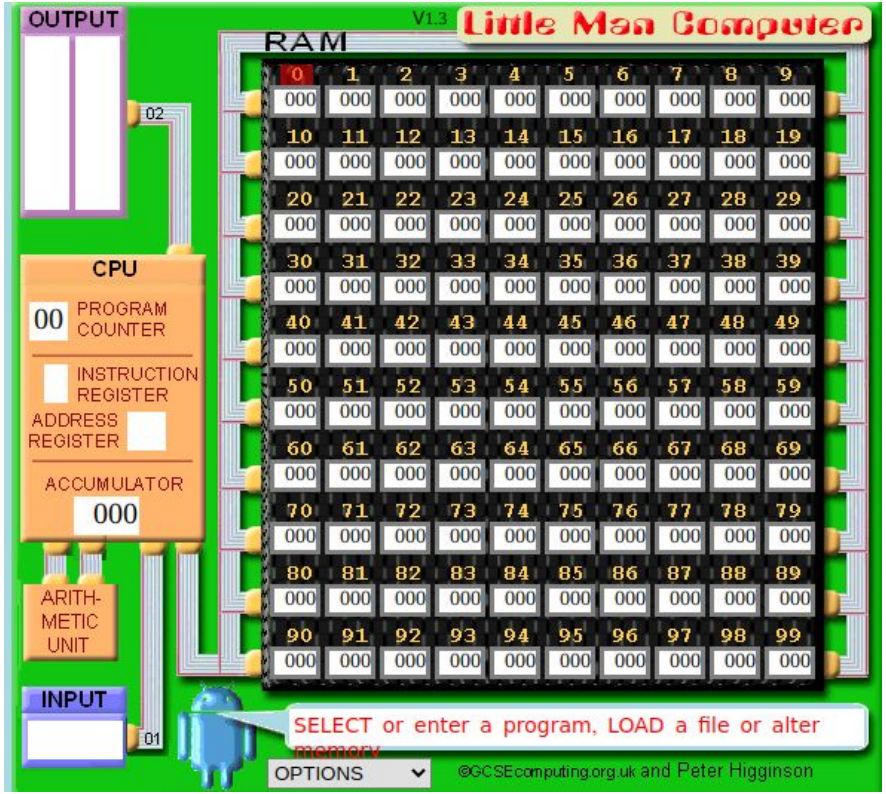
LMC Review

- LMC Execution Cycle
  - Check the Program Counter
  - Fetch the instruction from that address
  - Increment the Program Counter
  - Decode the fetched instruction into the Instruction and Address registers
  - Fetch any data needed
  - Execute the instruction
  - Branch or store the result
  - Repeat!



LMC Review

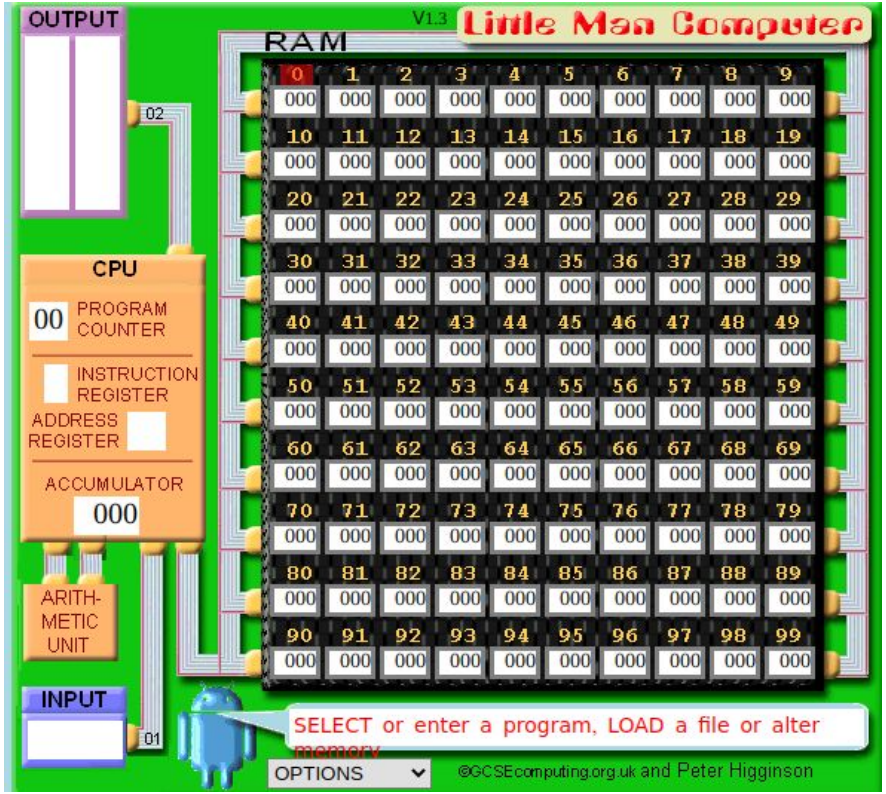
- LMC Instructions
  - ADD addition
  - SUB subtraction
  - STA store to memory
  - LDA load from memory
  - BRA unconditional branch
  - BRZ branch if zero
  - BRP branch if positive
  - INP get user input, put in acc
  - OUT output acc to output area
  - HLT/COB halt
  - DAT data



LMC Review

- We will be using the excellent LMC simulator at

  https://peterhigginson.co.uk/LMC/



Add One

- Our first program is going to be extremely simple
- Ask the user for a number, add one to it, and then output the number

```
        INP
        ADD ONE
        OUT
        HLT
ONE     DAT 1
```

Add One

- INP - get user input and put it in the accumulator
- ADD ONE - add the number stored at the label ONE to the value in the accumulator
- OUT - output the value in the accumulator to the output area
- HLT - stop execution

```
        INP
        ADD ONE
        OUT
        HLT
ONE     DAT 1
```

Add One

- ONE DAT 1 - Store the value 1 at the “current” position and make it available for reference with the label “ONE”
- Not too bad, right?
- This is a simple program but gives you the general flavor of assembly programming

```
        INP
        ADD ONE
        OUT
        HLT
ONE     DAT 1
```

Count Down

- Let’s do something more complex: Print the numbers 10 to 1 in decreasing order

```
        INP
        ADD ONE
        OUT
        HLT
ONE     DAT 1
```

Count Down

- Let’s do something more complex: Print the numbers 10 to 1 in decreasing order

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

Count Down

- LDA - load the number 10 into the accumulator
- BRZ - If the accumulator is 0, branch to EXIT
- OUT - Else, print the accumulator
- SUB - subtract 1 from the accumulator

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

Count Down

- BRA - unconditionally jump back to the start of the loop
- HLT - Halt (exit)
- ONE, TEN - Data slots to hold constants

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

Count Down

- This demonstrates two higher level programming language concepts:
  - Conditional logic (if statements)
  - Looping (for, do, while)
- Compilers take your higher level programming language constructs, like the while loop in C, and create instructions like this

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

MAX

- So far we haven't had to worry too much about data sizing
  - Our computations have fit in the accumulator
- Sometimes not all the working data can fit in the available registers
- What to do?

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

MAX

- Max: “Ask the user for two numbers and print the maximum of the two”
- Now we are in trouble: we need to computer the difference between the two numbers, but we also need to keep the numbers around to print the correct value

```
        LDA TEN
LOOP    BRZ EXIT
        OUT
        SUB ONE
        BRA LOOP
EXIT    HLT
ONE     DAT 1
TEN     DAT 10
```

MAX

- So we have three values that we are interested in
  - The first number entered
  - The second number entered
  - The difference between them
- We are going to have to store numbers somewhere in memory to make this all work

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX

- As a convention, to avoid stepping on the program memory, we will use the TOP of the memory space to store the values
- This is a proto-stack!
  - If we create a calling convention (covered later) we could define a MAX function here

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX

- INP - Get user input
- STA 99 - Store the first number at position 99
- INP - Get user input again
- STA 98 - Store the second number at position 98

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX

- SUB 99 - Subtract the number at position 99 from the accumulator
  - NB: the value in the accumulator is also the value at 98
  - If we had not stored the second value at 98, it would now be lost (unless we did some math)

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX

- Now the magic: The accumulator stores the difference between the two numbers
  - If the first number is greater than the second number, it is positive
  - Else, it is negative

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX

- BRP - branch if the number is positive to the instruction that loads the second number back into memory
- Else LDA - load the *first* number into memory and then branch to PRINT
- LDA - load the *second* number into memory

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX

- OUT - print the current number in the accumulator to output and halt
- Note how the control flow works here, implementing an if/else

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

MAX - Questions

- How would we convert this from a MAX to a MIN computation?
- How could we avoid storing the second value?
- Can we avoid storing the first value?

```
        INP
        STA 99
        INP
        STA 98
        SUB 99
        BRP LOAD_2ND
        LDA 99
        BRA PRINT
LOAD_2ND    LDA 98
PRINT OUT
        HLT
```

Practical LMC

- In this lecture you have seen how to implement some basic algorithms in LMC assembly
- Using branch instructions, we are able to implement rudimentary loops and conditional execution
- We only have one working register, the accumulator, so we have to store values in memory if the working set of data is larger than one value
- Remember: *IT'S JUST CODE*


MONTANA
STATE UNIVERSITY