Teaching London Computing

A Level Computer Science

Topic 5: Computer Architecture and Assembly

Queen Mary University of London

King's College London

Computing At School
EDUCATE · ENGAGE · ENCOURAGE

Department for Education

SUPPORTED BY
MAYOR OF LONDON

Network of Excellence Computer Science Teaching
UNIVERSITY PARTNER

William Marsh
School of Electronic Engineering and Computer Science
Queen Mary University of London

Aims

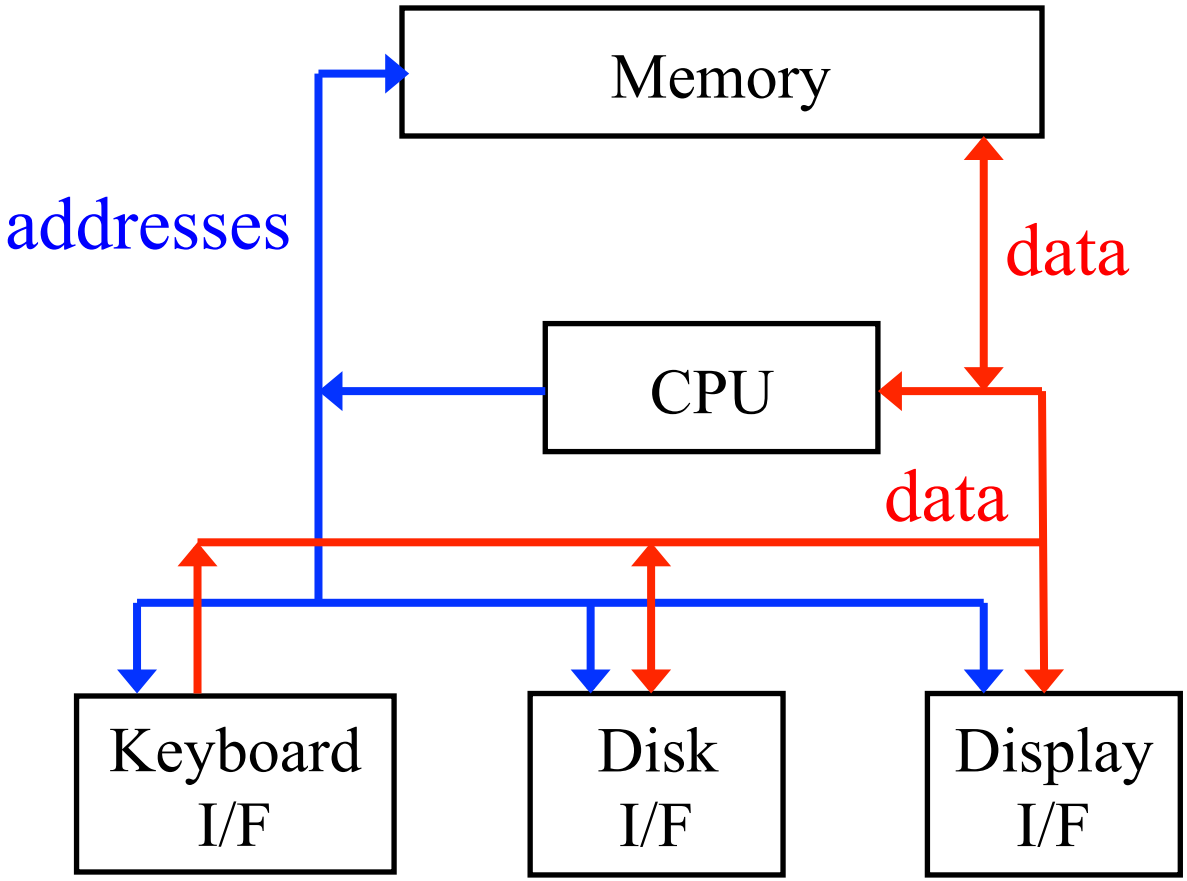
- Understand CPU structure in more detail
  - Fetch-execute cycle
  - Faster CPUs
- Write simple assembly code
  - Use Little Man Computer
- Understand principles of compiling
- Compare compilers and interpreters

CPU Structure

What's in a CPU

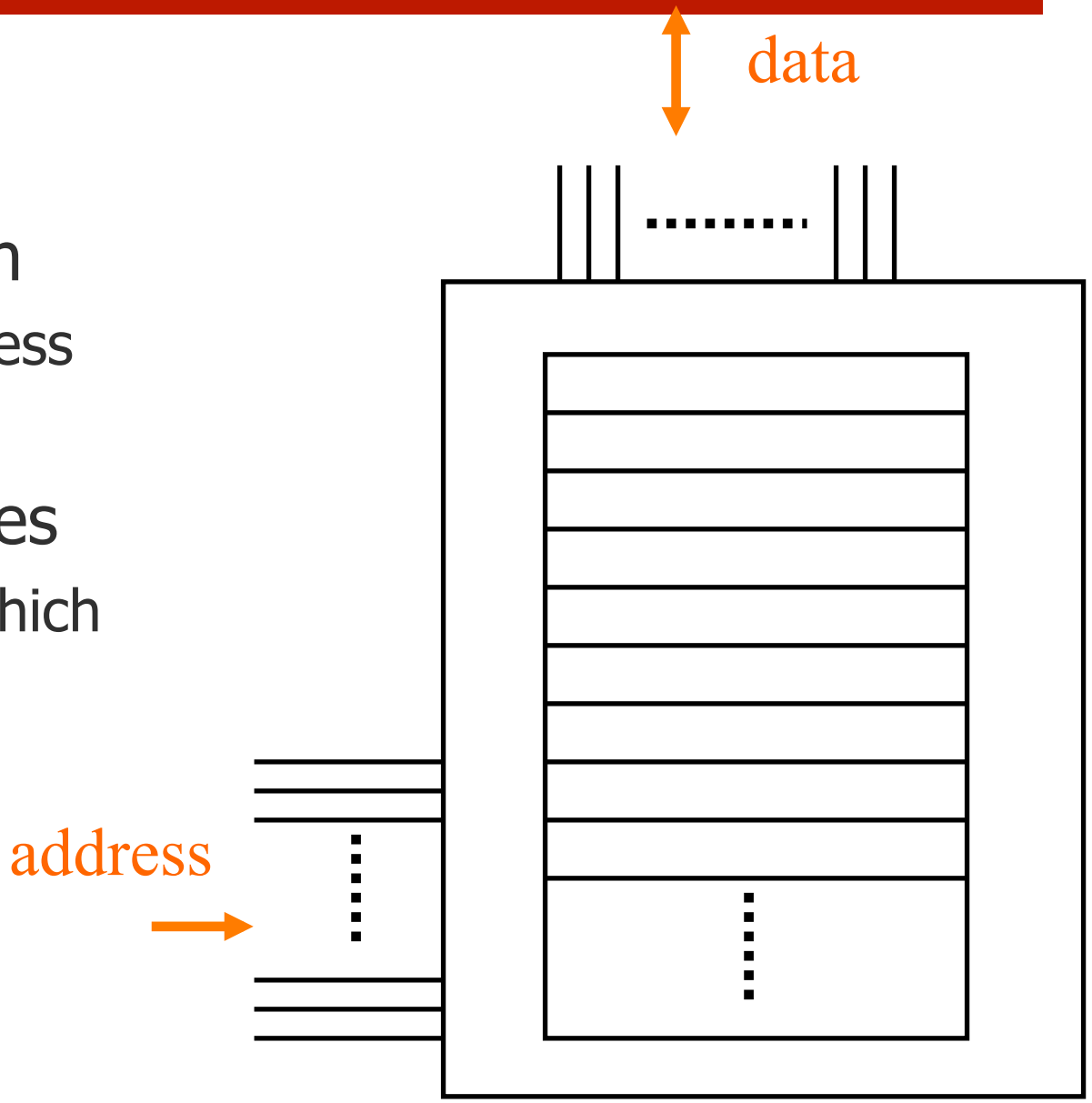
Simple Computer

- Processor
  - CPU
- Memory
  - Data
  - Program instructions
- I/O
  - Keyboard
  - Display
  - Disk



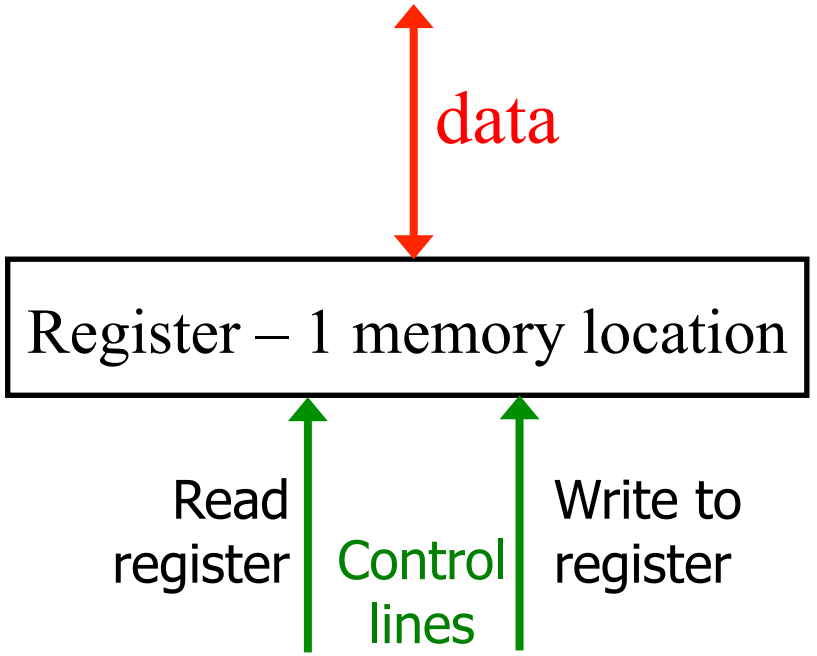
Memory

- Each location
  - has an address
  - hold a value
- Two interfaces
  - address – which location?
  - data – what value?

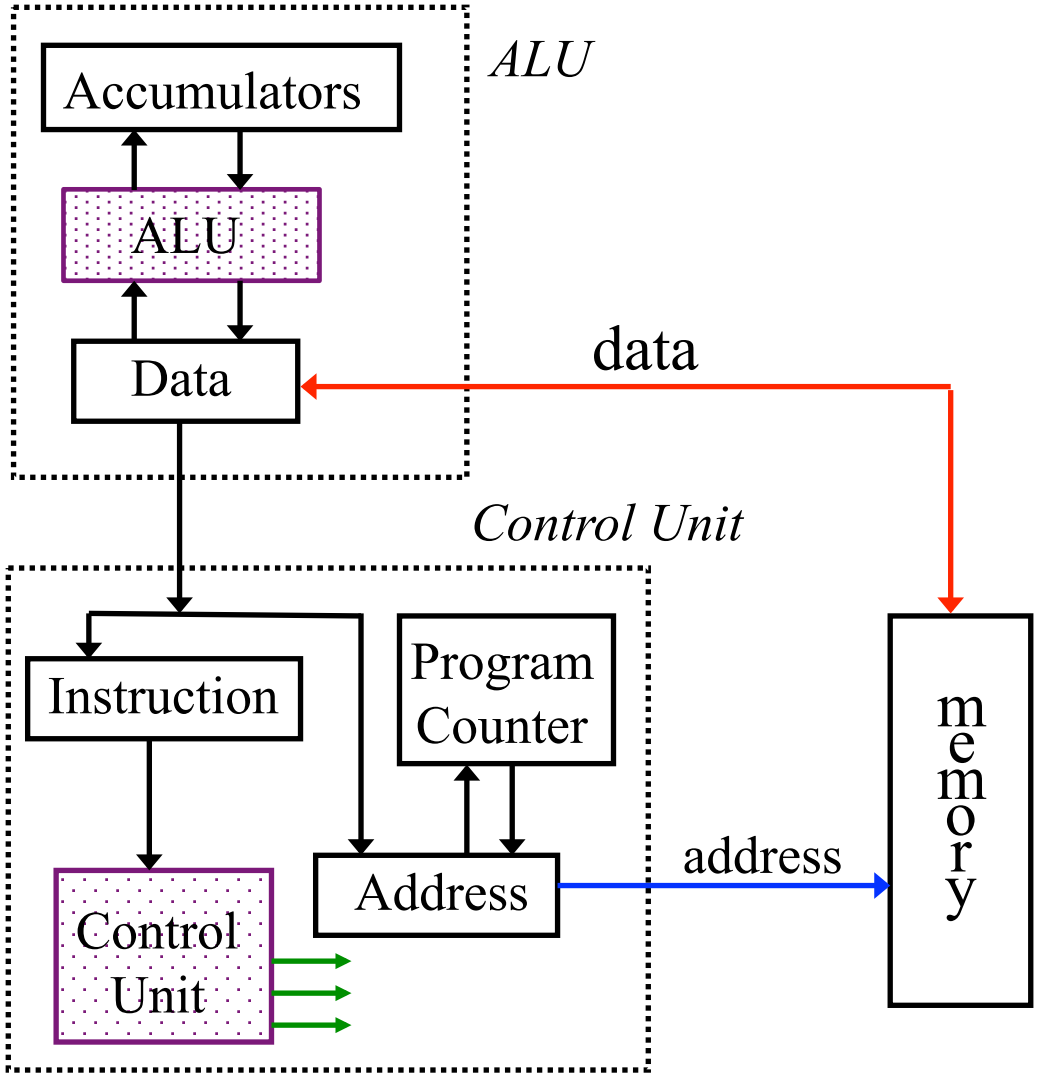


Registers (or Accumulators)

- A storage area inside the CPU
- VERY FAST
- Used for arguments and results to one calculation step



CPU Structure



- Accumulators
  - Data for calculation
- Data
  - Word to/from memory
- PC
  - Address of next instruction
- Instruction
- Address
  - For memory access

Instructions

- Instruction
  - What to do: Opcode
  - Where: memory address
- Instructions for arithmetic
  - Add, Multiply, Subtract
- Memory instructions
  - LOAD value from memory
  - STORE value in memory

Add Instruction

- One address and accumulator (ACC)
  - Value at address added to accumulator

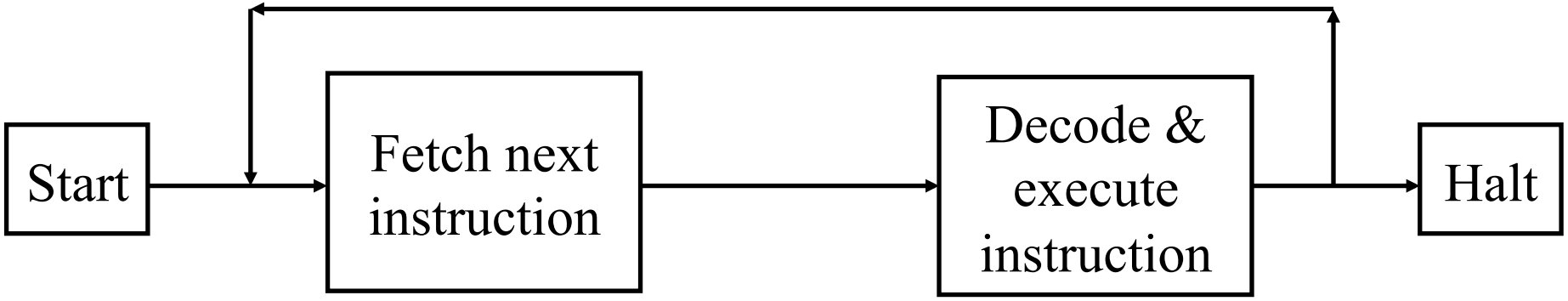
- ACC = ACC + Memory[Address]

Fetch-Execute Cycle

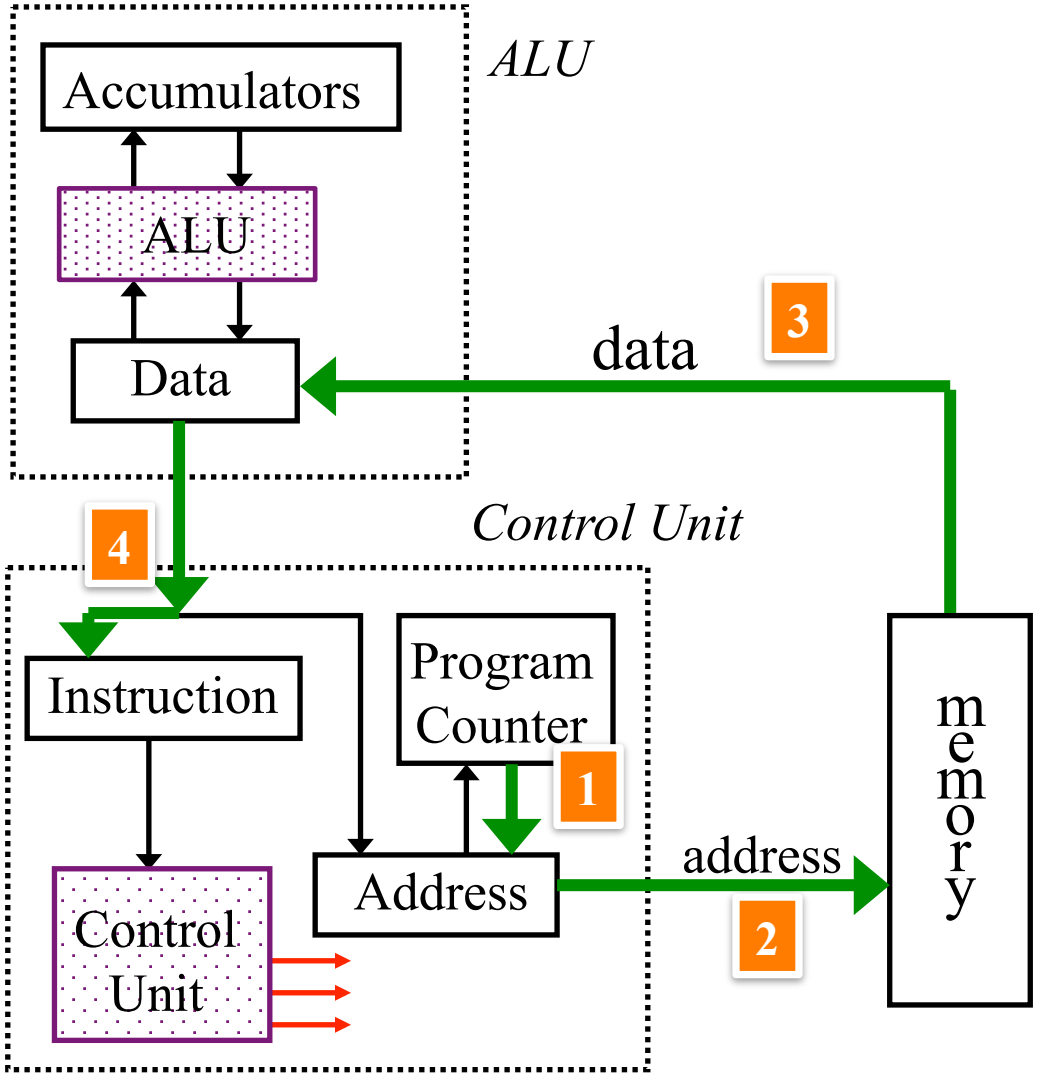
How the Computer Processes Instructions

Fetch-Execute

- Each instruction cycle consists on two subcycles
- Fetch cycle
  - Load the next instruction (Opcode + address)
  - Use Program Counter
- Execute cycle
  - Control unit interprets the opcode
  - ... an operation to be executed on the data by the ALU

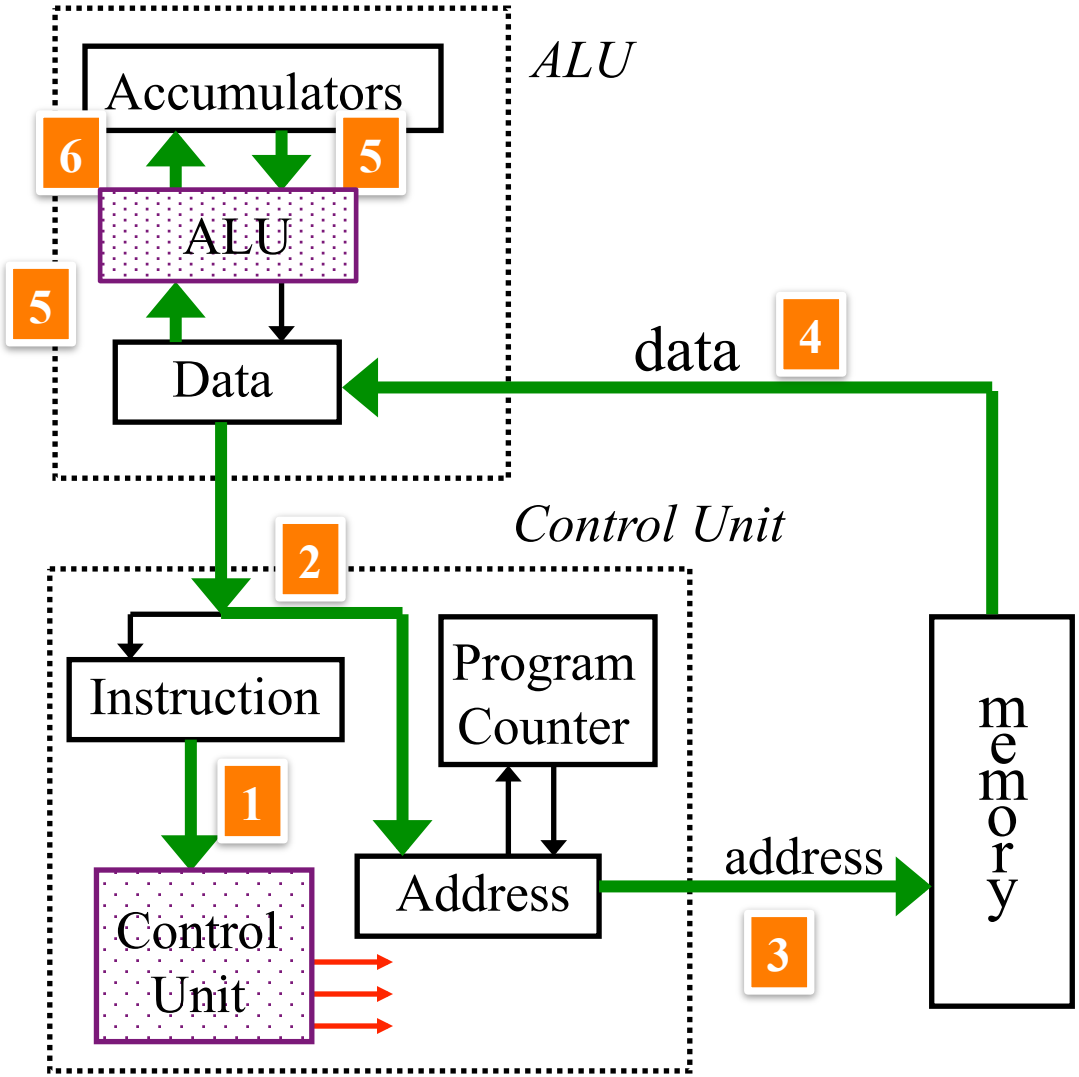


Fetch Instruction



1. Program counter to address register
2. Read memory at address
3. Memory data to 'Data'
4. 'Data' to instruction register
5. Advance program counter

Execute Instruction



1. Decode instruction
2. Address from instruction to 'address register'
3. Access memory
4. Data from memory to 'data register'
5. Add (e.g.) data and accumulator value
6. Update accumulator

Summary of CPU Architecture

- Memory contains data and program
  - Program counter: address of next instruction
  - Instructions represented in binary
  - Each instruction has an ‘opcode’
- Instructions contain addresses
  - Addresses used to access data
- Computer does ‘fetch-execute’
  - ‘Execute’ depends on opcode
- Computer can be built from < 10,000 electronic switches (transistors)

Discussion

- Could we act out the way a computer works?

Making a Faster Computer

Clock

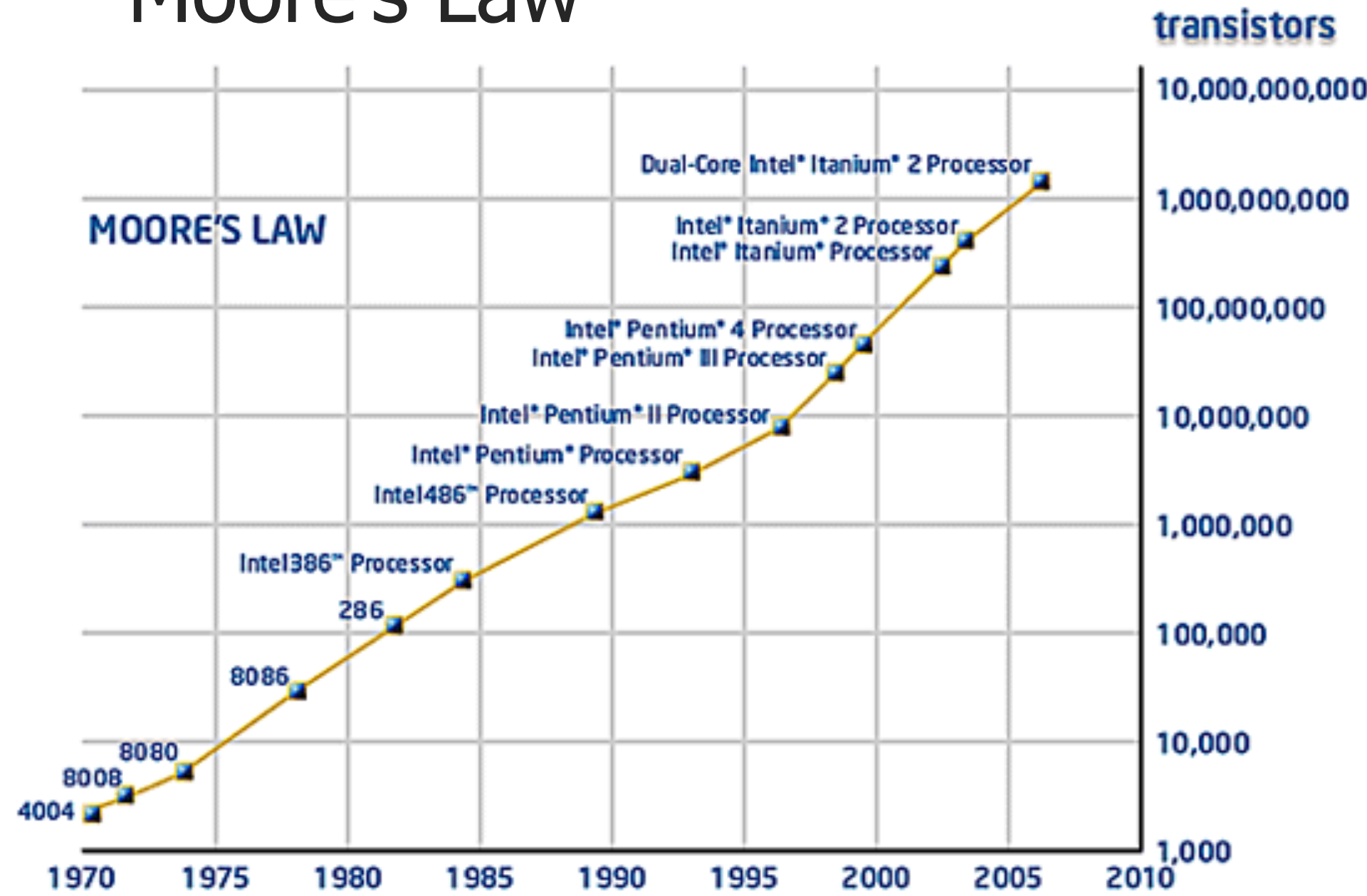
- Steps in sequence are controlled by a clock
- Clock frequency
  - 8086 – 10 MHz
  - Pentium 4 – 3 GHz
  - **However, clock speeds no longer increasing much**

- Processing power depends on
  - Clock speed
  - Clock cycles to execute an instruction

1GHz Clock

- 10^9 cycles per second
  - 1 cycle every 1 ns
  - Light travels 1 m in approx 3 ns
  - Billion instructions per second (up to)
- Each instruction takes longer
  - … many instruction in process at once
- Memory access is slower
  - 10 of ns

Moore's Law


transistors
MOORE'S LAW
Dual-Core Intel® Itanium® 2 Processor
Intel® Itanium® 2 Processor
Intel® Itanium® Processor
Intel® Pentium® 4 Processor
Intel® Pentium® III Processor
Intel® Pentium® II Processor
Intel® Pentium® Processor
Intel486™ Processor
Intel386™ Processor
286
8086
8080
8008
4004
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
1970
1975
1980
1985
1990
1995
2000
2005
2010

Faster Computer

1. Faster clocks
   - smaller IC lines (90 → 45 → 32 → 22nm)
   - shorter time to switch
2. More Registers; Bigger registers
   - 32 → 64 bits
3. Fetch-execute pipeline
   - overlap the fetch and execute
4. Cache memory
5. Multiple cores

Intel 86 Family

- 8086 – 1978
  - 16 bits
- 80386 – 1986
  - 32 bits
- Pentium 4 – 2000
  - 32 bits → 64 bits
- Intel core i3, i5, i7
- COMPATIBLE compiled program

- More instruction
- Longer words
- More on chip
  - floating point
  - cache memory
- Instructions in parallel
  - pipeline
  - superscalar
  - multi-core

Little Man Computer

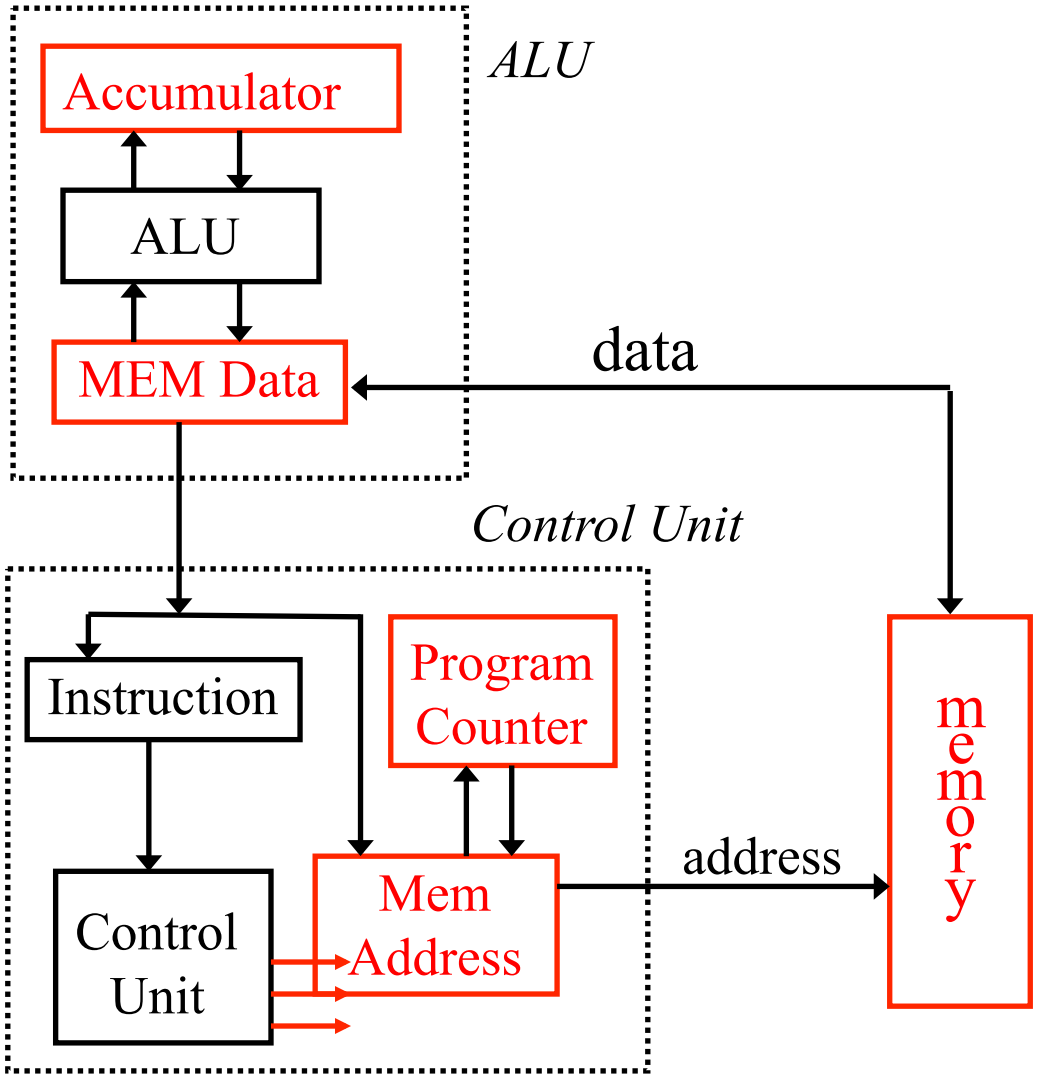
A Simulator for a Simple CPU

Advantages of LMC

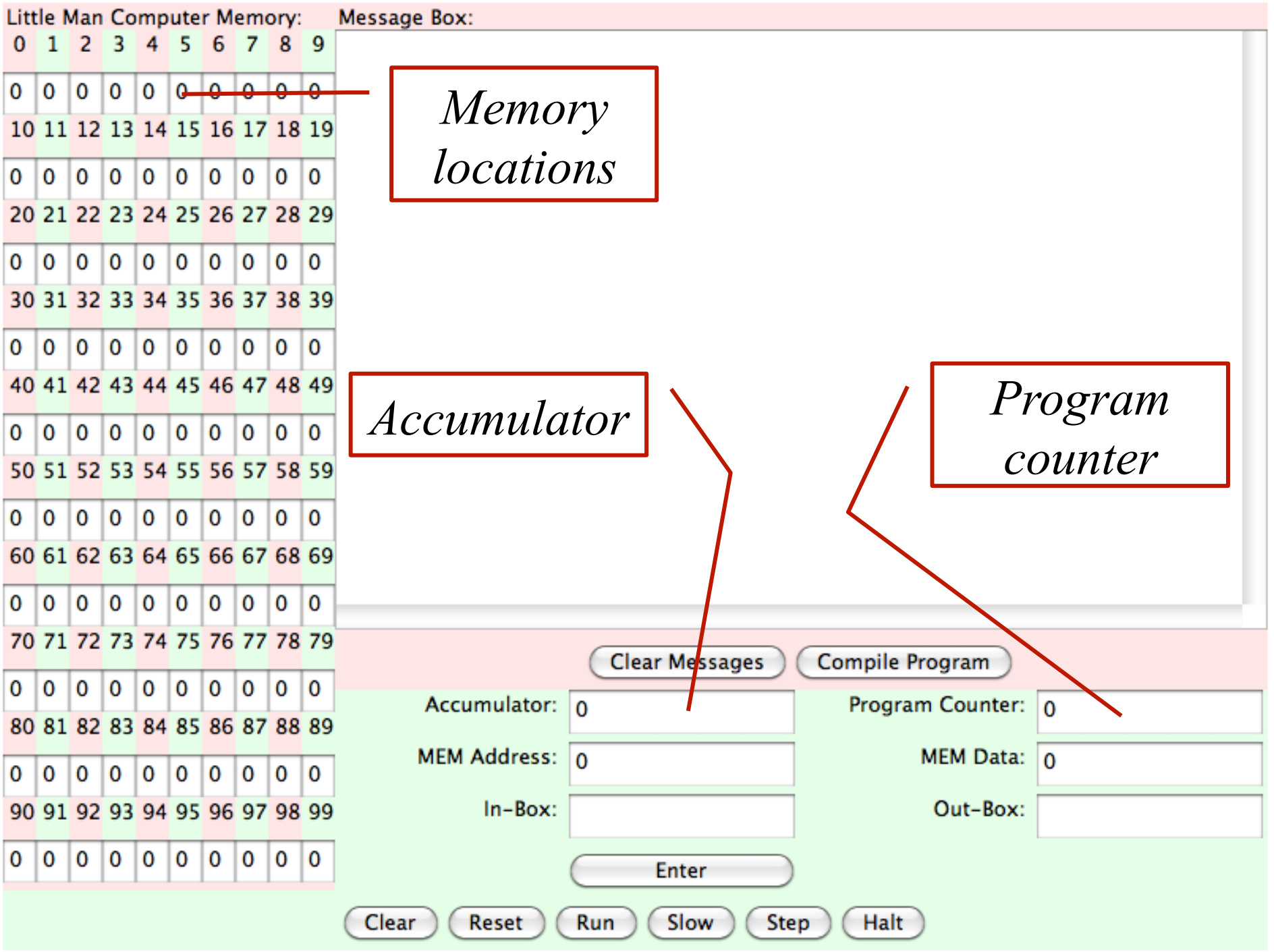
- Very simple
- Number of implementations
  - Java applet
  - Excel
  - Flash, .net
  - MacOS

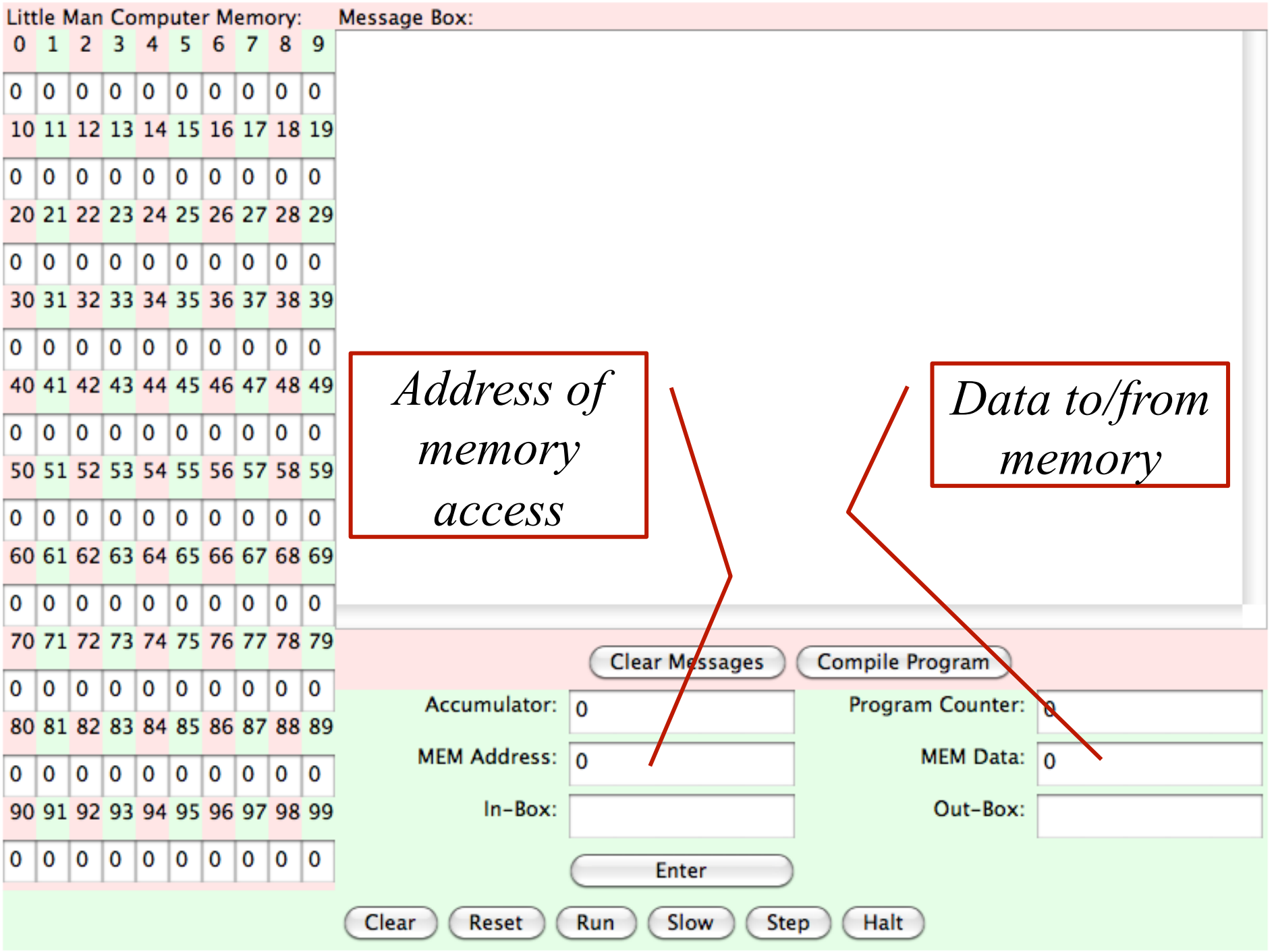
- Some incompatibilities
- Feature: uses decimal not binary

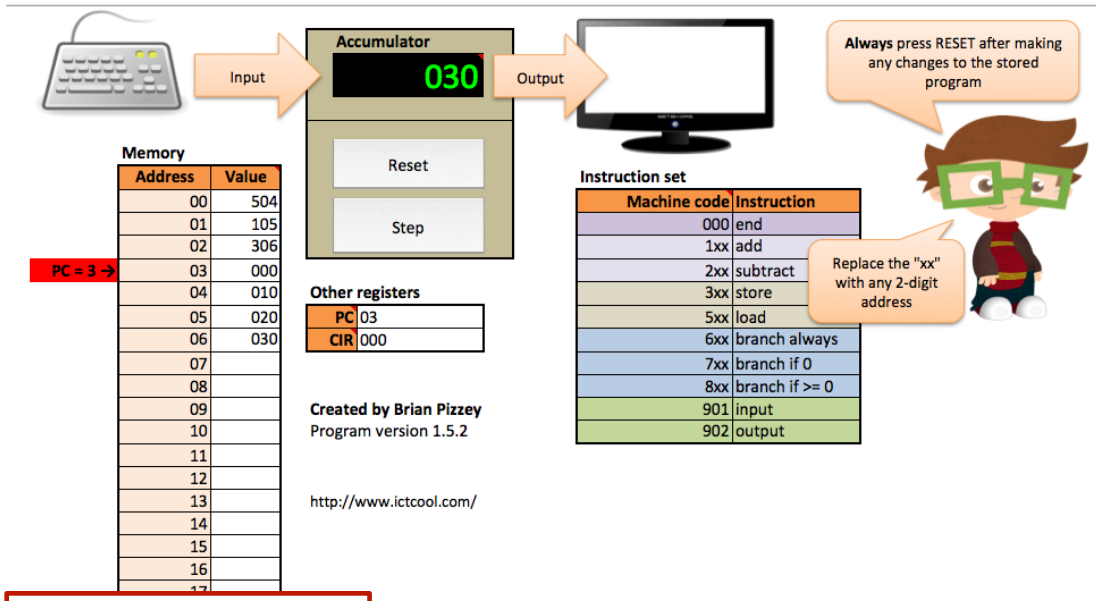
LMC CPU Structure



- Visible registers shown in red


Little Man Computer Memory:
Message Box:
0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 0 0 0 0 0
10 11 12 13 14 15 16 17 18 19
0 0 0 0 0 0 0 0 0 0
20 21 22 23 24 25 26 27 28 29
0 0 0 0 0 0 0 0 0 0
30 31 32 33 34 35 36 37 38 39
0 0 0 0 0 0 0 0 0 0
40 41 42 43 44 45 46 47 48 49
0 0 0 0 0 0 0 0 0 0
50 51 52 53 54 55 56 57 58 59
0 0 0 0 0 0 0 0 0 0
60 61 62 63 64 65 66 67 68 69
0 0 0 0 0 0 0 0 0 0
70 71 72 73 74 75 76 77 78 79
0 0 0 0 0 0 0 0 0 0
80 81 82 83 84 85 86 87 88 89
0 0 0 0 0 0 0 0 0 0
90 91 92 93 94 95 96 97 98 99
0 0 0 0 0 0 0 0 0 0
Memory locations
Accumulator
Program counter
Clear Messages
Compile Program
Accumulator: 0
Program Counter: 0
MEM Address: 0
MEM Data: 0
In-Box:
Out-Box:
Enter
Clear
Reset
Run
Slow
Step
Halt

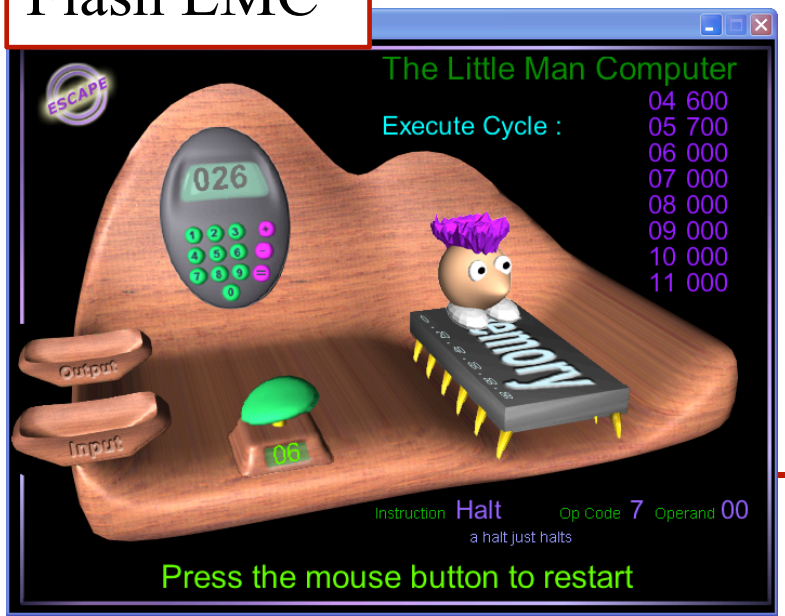

Little Man Computer Memory:
Message Box:
0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 0 0 0 0 0
10 11 12 13 14 15 16 17 18 19
0 0 0 0 0 0 0 0 0 0
20 21 22 23 24 25 26 27 28 29
0 0 0 0 0 0 0 0 0 0
30 31 32 33 34 35 36 37 38 39
0 0 0 0 0 0 0 0 0 0
40 41 42 43 44 45 46 47 48 49
0 0 0 0 0 0 0 0 0 0
50 51 52 53 54 55 56 57 58 59
0 0 0 0 0 0 0 0 0 0
60 61 62 63 64 65 66 67 68 69
0 0 0 0 0 0 0 0 0 0
70 71 72 73 74 75 76 77 78 79
0 0 0 0 0 0 0 0 0 0
80 81 82 83 84 85 86 87 88 89
0 0 0 0 0 0 0 0 0 0
90 91 92 93 94 95 96 97 98 99
0 0 0 0 0 0 0 0 0 0
Address of memory access
Data to/from memory
Clear Messages
Compile Program
Accumulator:
0
Program Counter:
0
MEM Address:
0
MEM Data:
0
In-Box:
Out-Box:
Enter
Clear
Reset
Run
Slow
Step
Halt

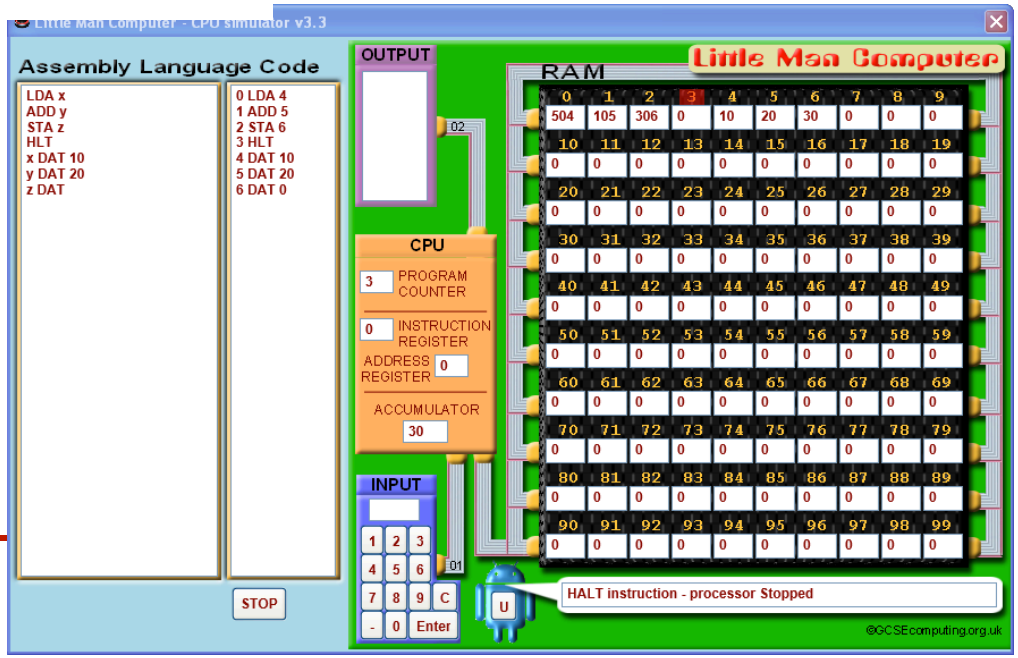


Excel LMC

Windows .net LMC

Flash LMC





Feature Comparison

- Step and Animation
  - Most allow single stepping – see fetch-execute
  - Animation – e.g. flash version
- Can program using mnemonics
  - LDA, STA, ADD …
  - Supported in e.g. applet
  - Not supported in Excel, Flash versions
- Programming versus demonstration

LMC Instructions

Instructions

OpCode	Address

- Opcode: 1 decimal digit
- Address: two decimal digits – xx

Code	Mnemonic	Description
000	HLT	Halt
1xx	ADD	Add: acc + memory → acc
2xx	SUB	Subtract: acc – memory → acc
3xx	STA	Store: acc → memory
5xx	LDA	Load: memory → acc
6xx	BR	Branch always
7xx	BRZ	Branch is acc zero
8xx	BRP	Branch if acc > 0
901	IN	Input
902	OUT	Output

Add and Subtract Instruction

ADD	Address

SUB	Address

- One address and accumulator (ACC)
  - Value at address combined with accumulator value
  - Accumulator changed

- Add: ACC ← ACC + Memory[Address]
- Subtract: ACC ← ACC – Memory[Address]

Load and Store Instruction

LDA	Address

STA	Address

- Move data between memory and accumulator (ACC)

- Load: ACC ← Memory[Address]
- Store: Memory[Address] ← ACC

Branch Instructions

- Changes program counter
- May depend on accumulator (ACC) value

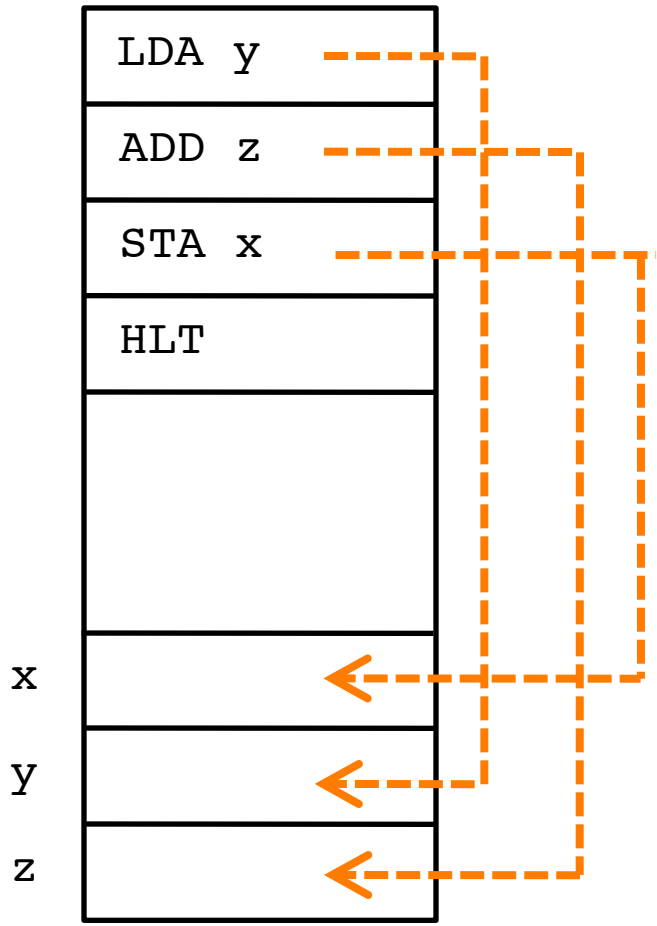
- BR: PC ← Address
- BRZ: if ACC == 0 then PC ← Address
- BRP: if ACC > 0 then PC ← Address

LMC Example

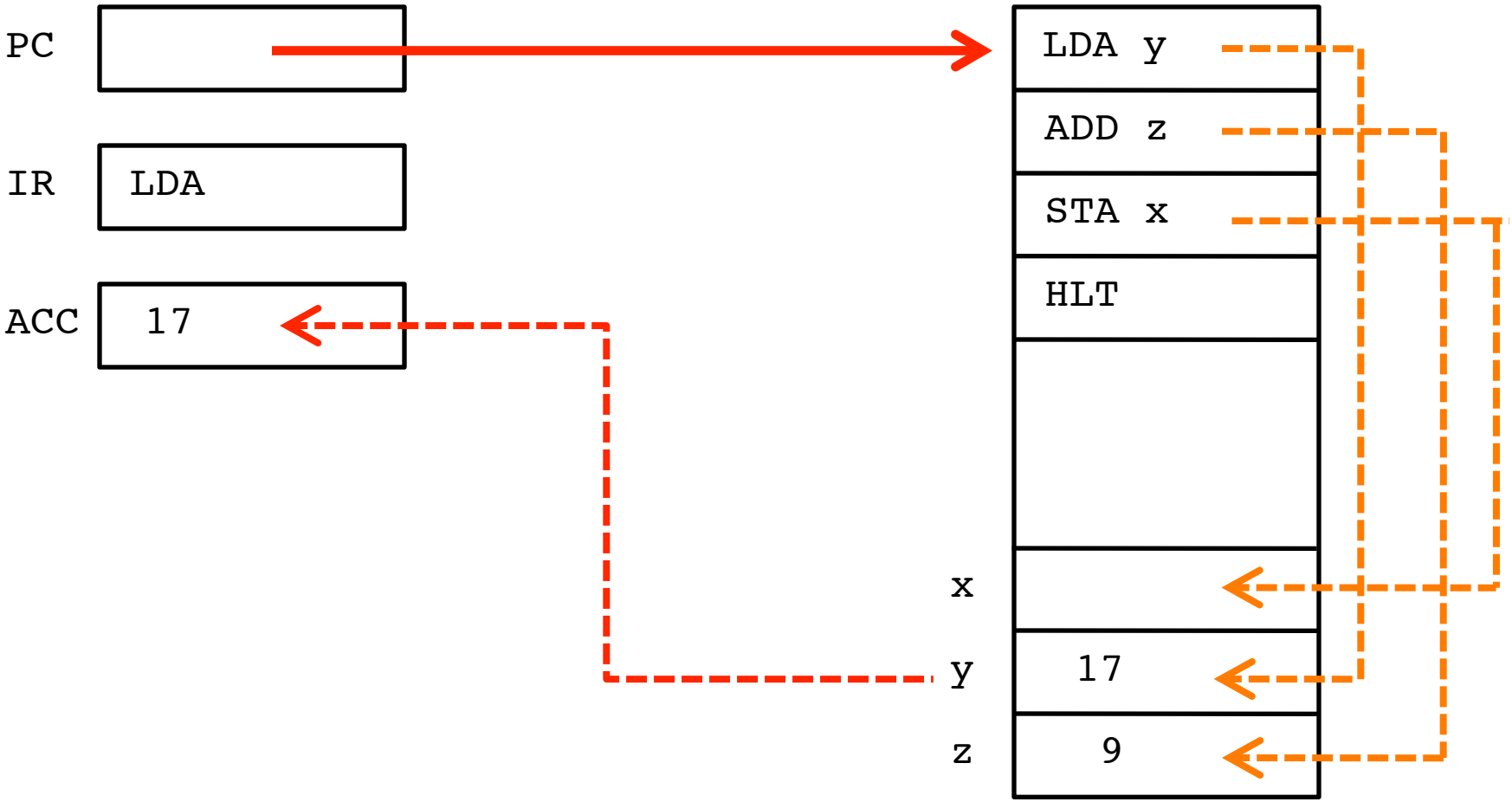
Compiling a Simple Program

- Compiler translates high level program to low level

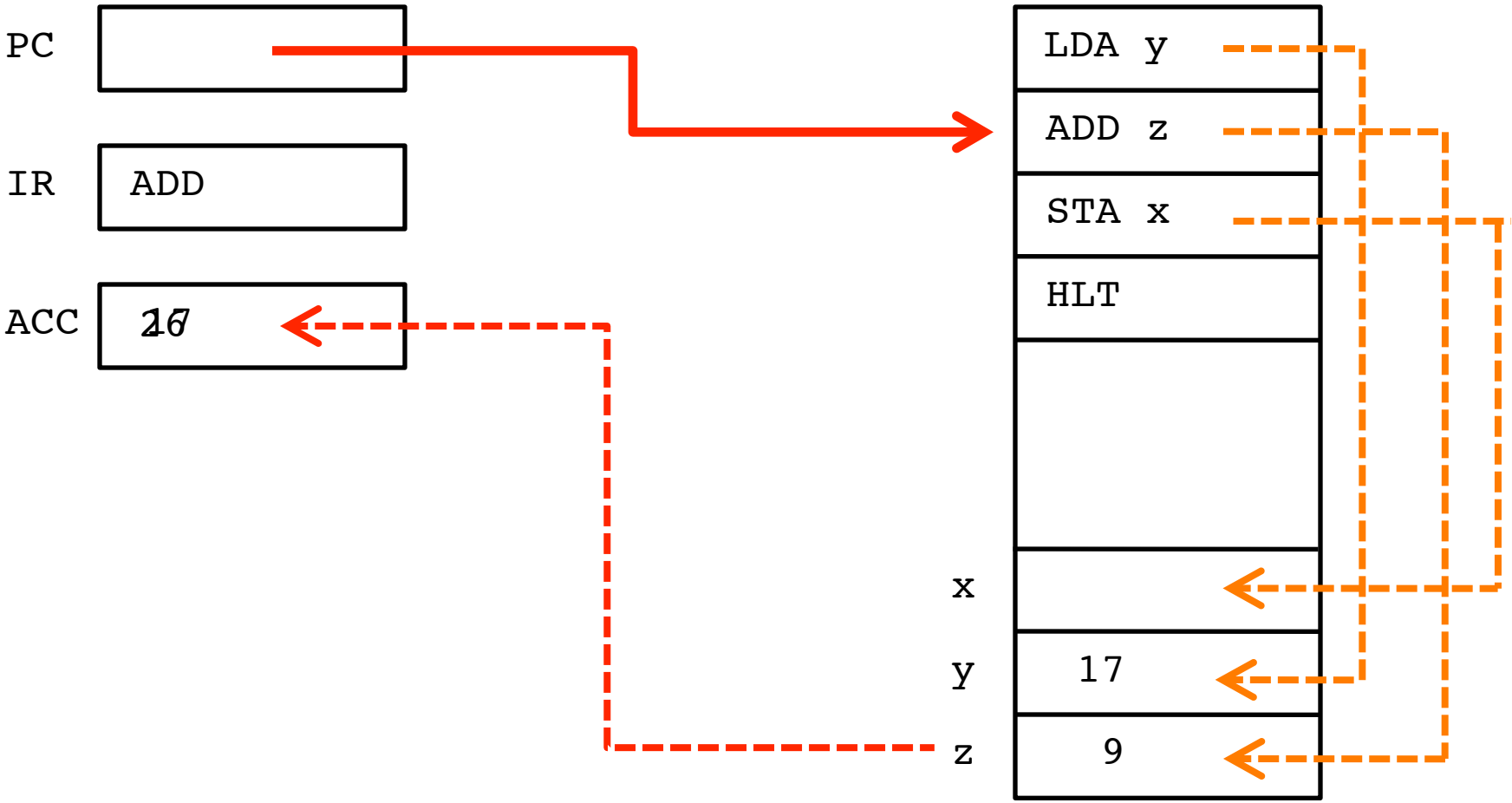
- E.g. x = y + z



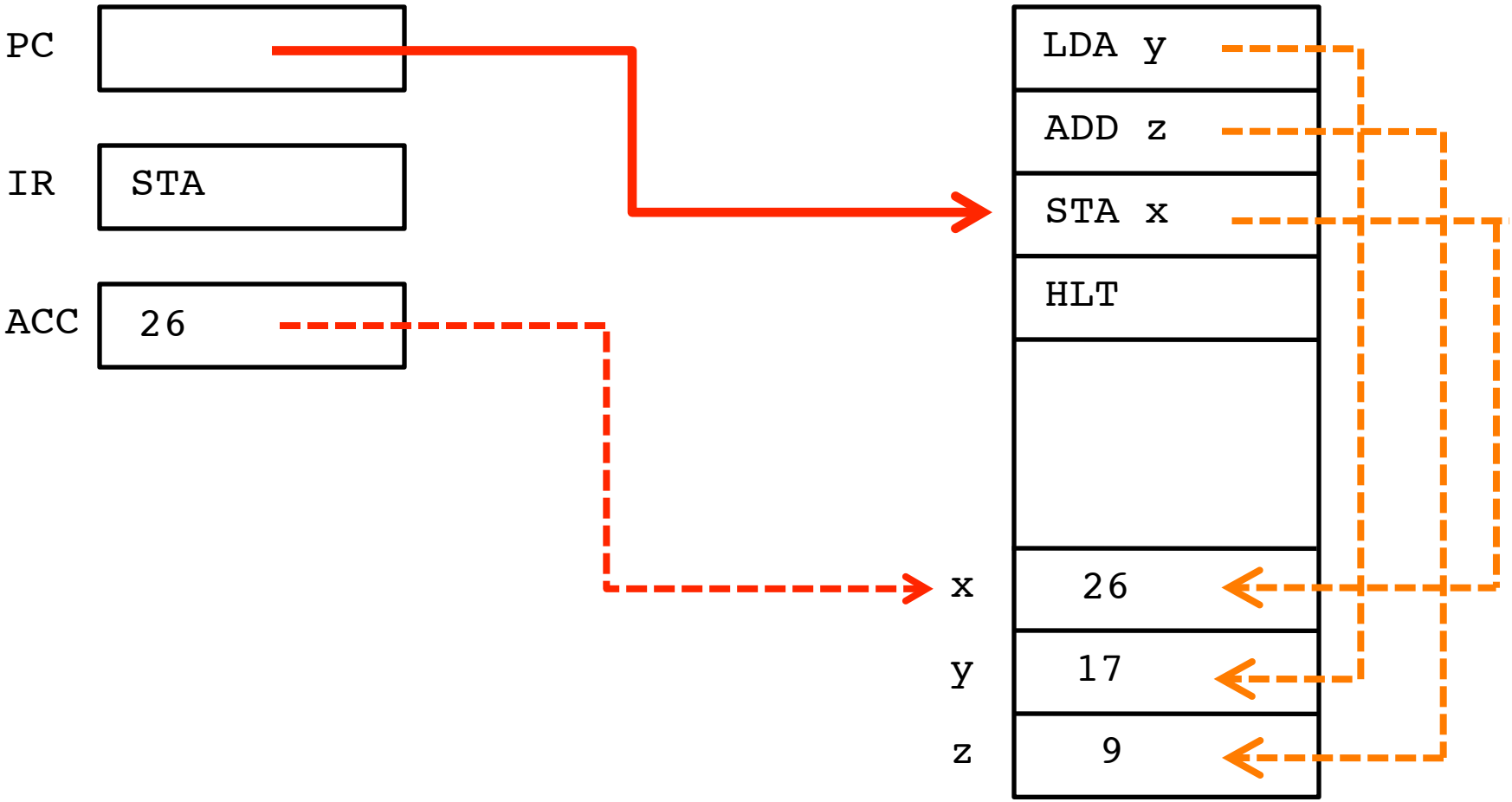
Running the Simple Program



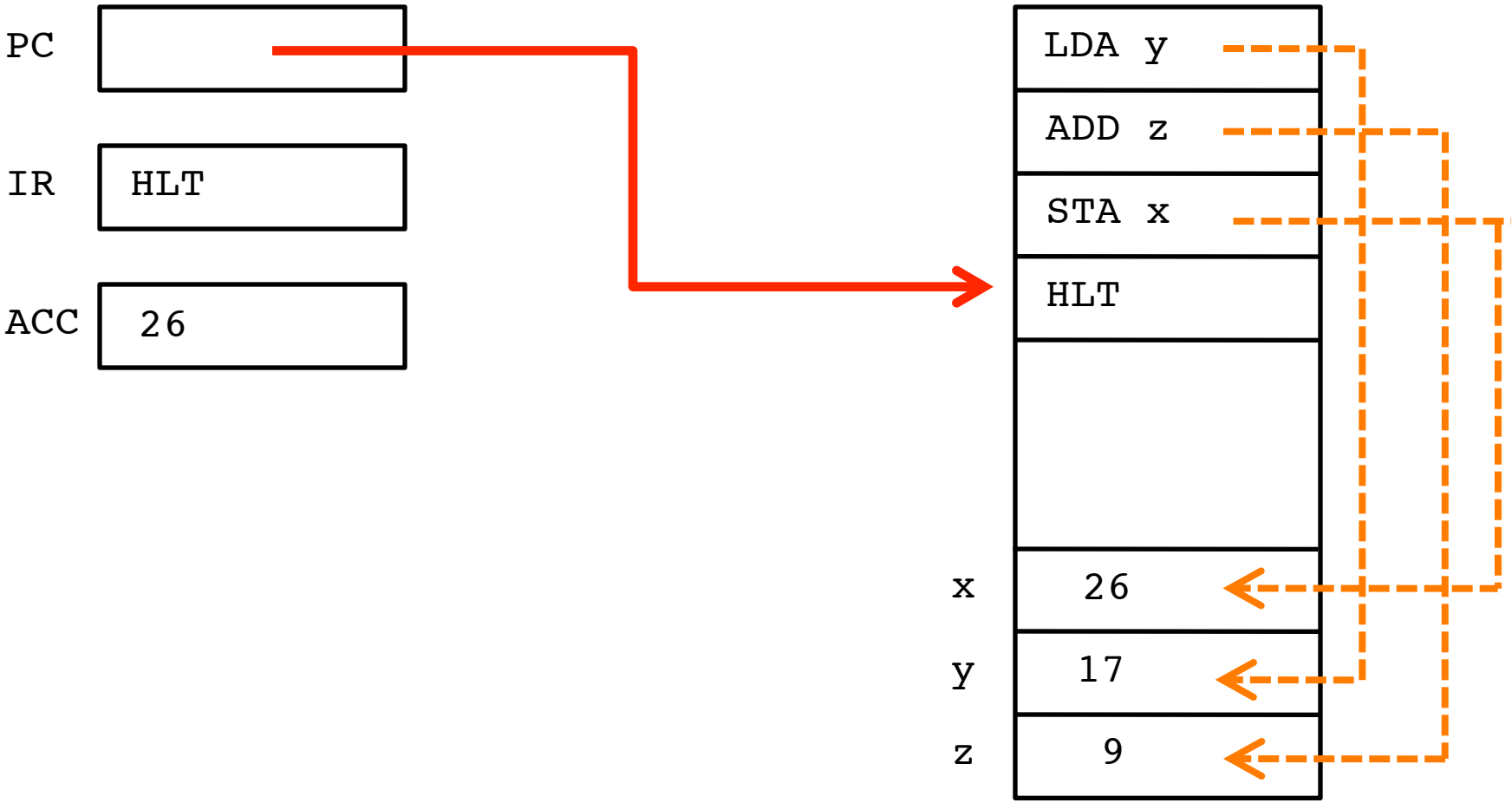
Running the Simple Program



Running the Simple Program



Running the Simple Program



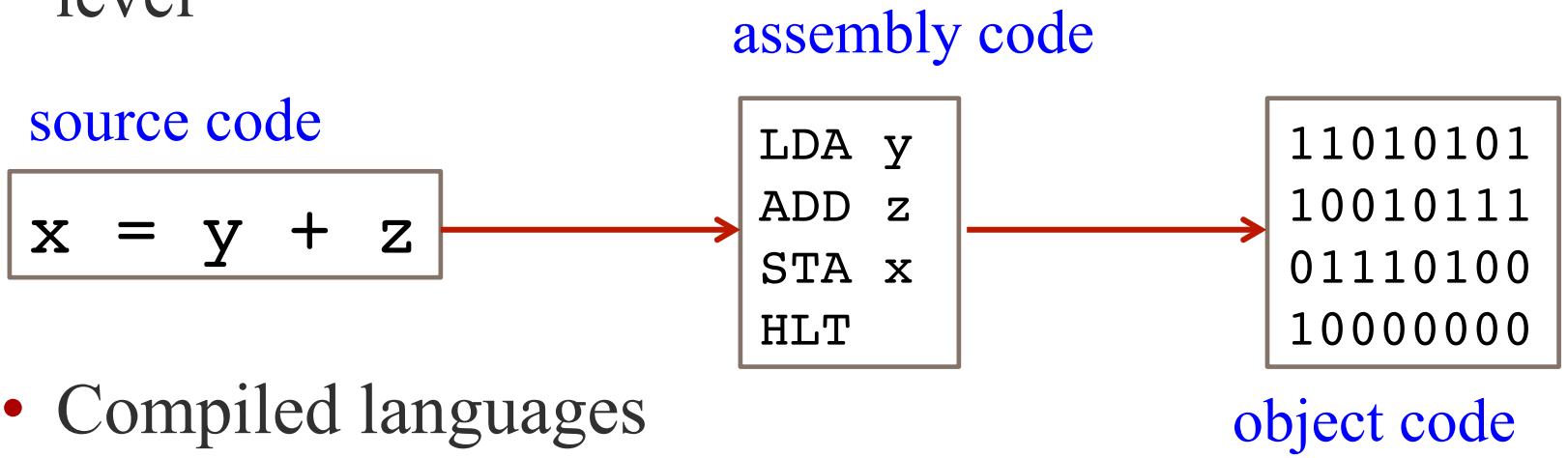
Practice Exercises

- Try the first three exercises on the practical sheet

Compilers and Interpreters

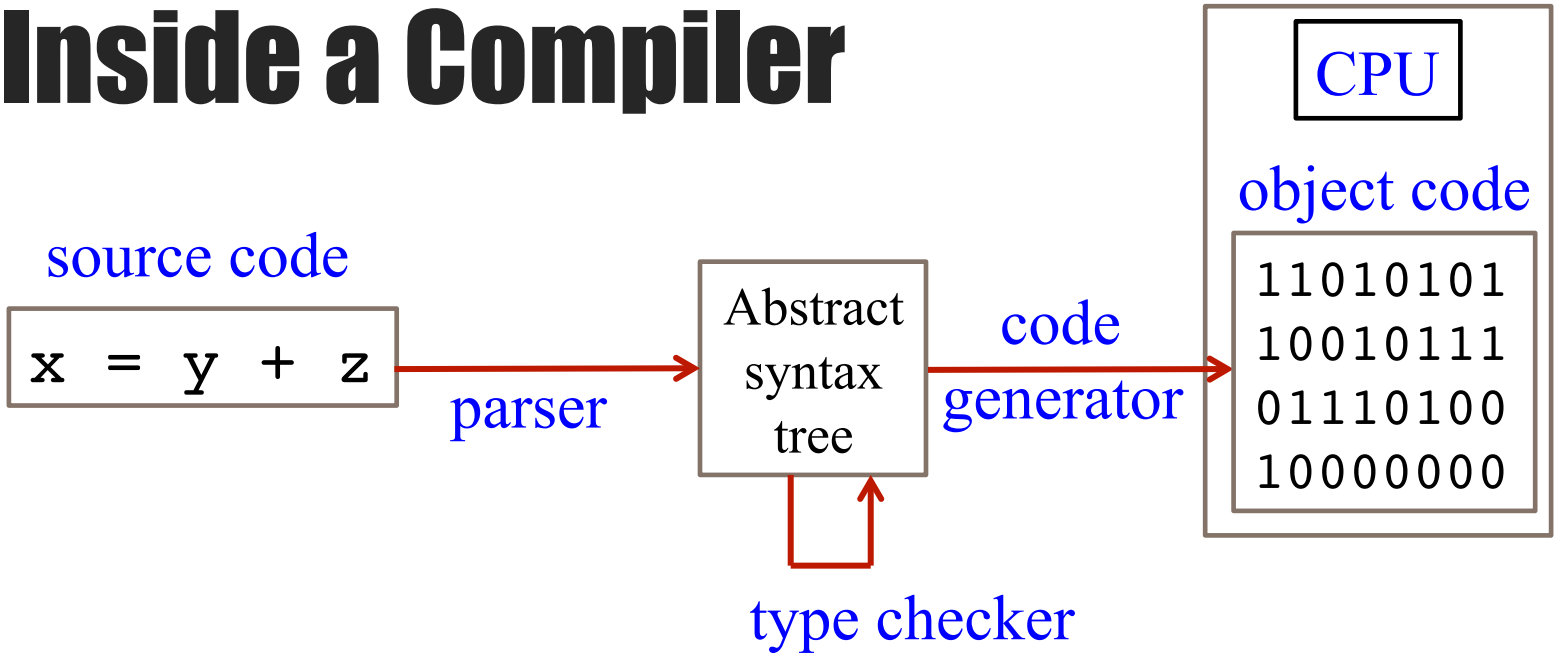
Compiler

- Compiler **translates** high level program to low level



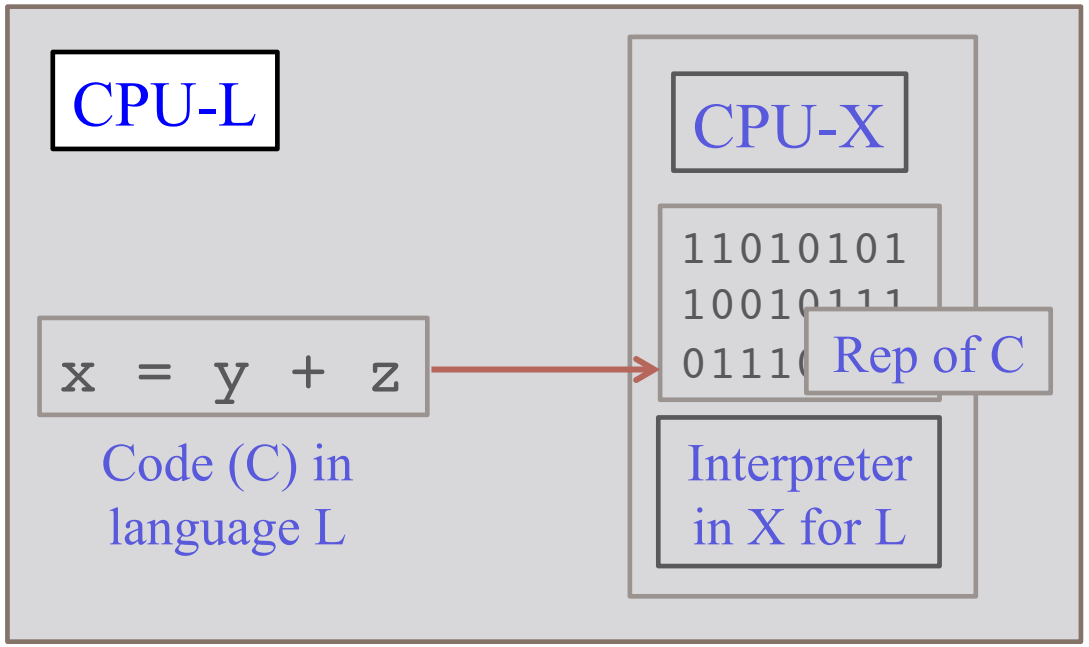
- Compiled languages
  - Statically typed
  - Close to machine
  - Examples: C, C++, (Java)
  - Compiler for each CPU

Inside a Compiler



- Parser: checks that the source code matches the grammar
- Type checker: checks types of variables and expressions
- Code generator: generates (optimised) code

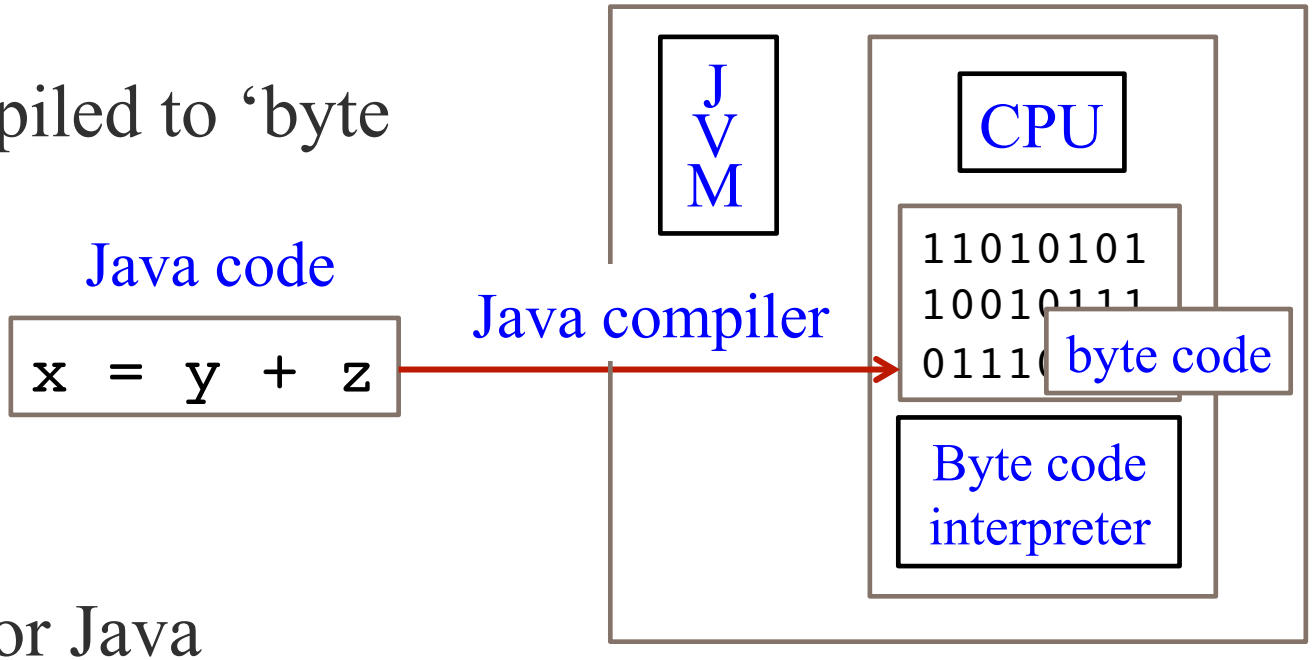
Principle of an Interpreter



- One machine can emulate another
  - 'micro-coded' CPU
  - Intel CPUs and RISC inside

Java and .NET Language

- Java is compiled to 'byte code'



- Byte code for Java 'virtual machine' (JVM)
  - One compiler
  - Libraries
  - Other languages
- The JVM is emulated on real computers
  - 'JIT' compiler

How Does an Interpreter Work?

- Example: LMC emulator

```
def readMem(memory):
    global mdr
    mdr = memory[mar]
```

```
acc = 0
mdr = 0
mar = 0
pc = 0
memory = [504,105,306, 0,
          11, 17,...]
```

State of the LMC: registers and memory

```
def execute(memory, opcode, arg):
   global acc, mar, mdr, pc
   if opcode == ADD:
      mar = arg
      readMem(memory)
      acc = acc + mdr
   elif opcode == SUB:
      mar = arg
      readMem(memory)
      acc = acc – mdr
   ...
```

```
def fetch(memory):
   global pc, mar
   mar = pc
   pc = pc + 1
   readMem(memory)
```

Update state following rules

Summary

- A CPU executes Fetch-Execute
  - Fetch: next instructions from memory
  - Execute: data to/from memory and accumulator
  - Opcode determines execute
- One machine can emulator another
- Program execution
  - Compiler: translates
  - Interpreter: parses then interprets
  - Blend: Java or .NET and byte code