Machine Programming

Philipp Reinecke

ReineckeP@cardiff.ac.uk

Language Categories

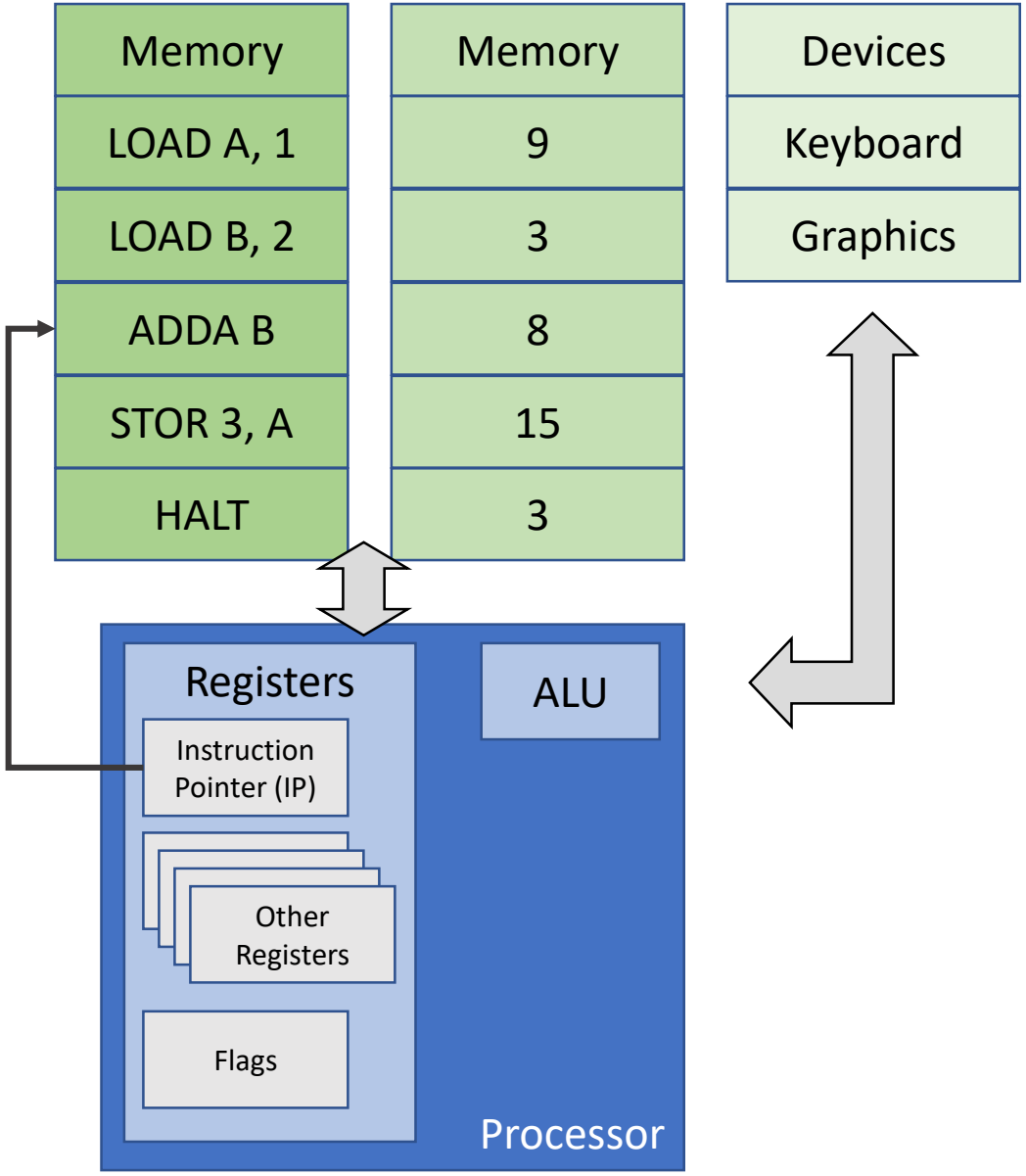
High-Level Languages
platform independent
(e.g. C++, Java)

Assembly Languages
platform-specific
Mnemonics

Machine Languages
platform-specific
Binary Values

A Simple Processor

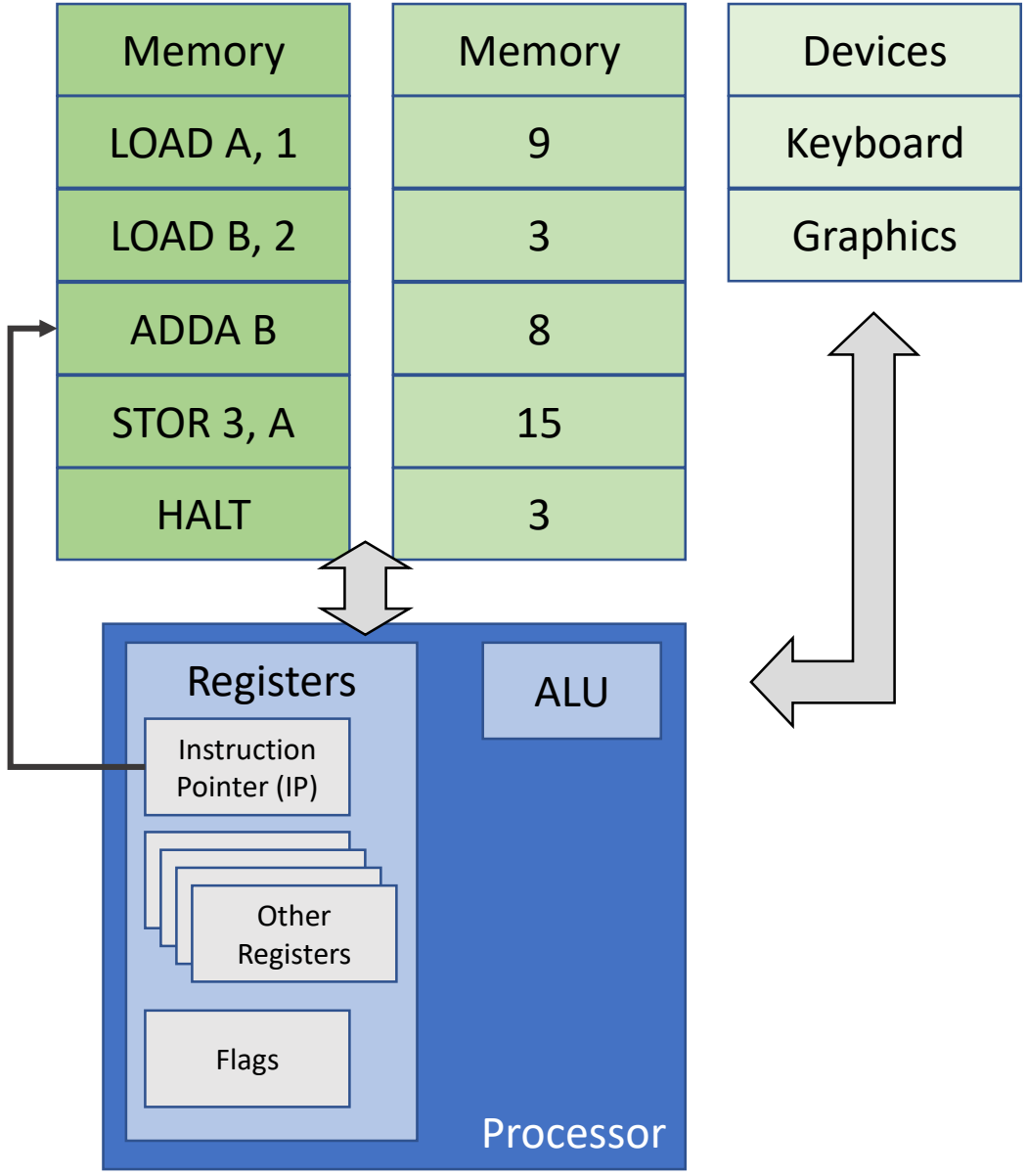
- Memory
  - Contains instructions
  - Contains data
  - May be same space or different spaces (von Neumann/Harvard architectures)
- Devices
  - Input/output, e.g. graphics
  - May generate interrupts (e.g. keyboard)
- Arithmetic-Logic Unit (ALU)
  - Used for computations
- Registers
  - Instruction Pointer (IP): Points to memory location to get next instruction from
  - Other registers:
    - General use
    - Special use, e.g. memory addressing
  - Flags:
    - Boolean status values
    - E.g. overflow, zero



Instructions

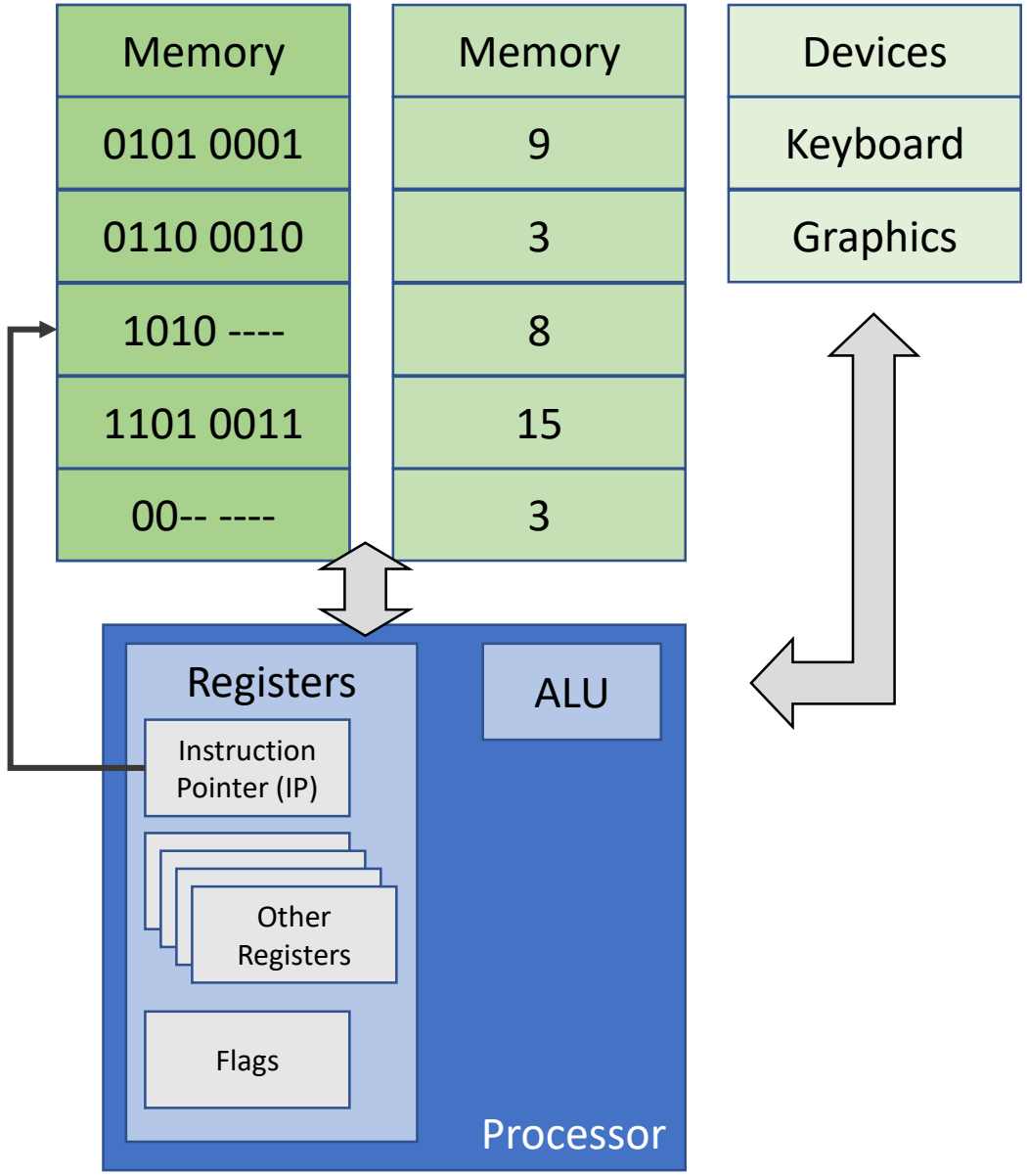
- Typically one instruction per line
- Not case-sensitive
- Each line specifies one operation and its operands (if any)
- Usually between 0 and 4 operands
- Typical format:

```
<Mnemonic> <Operand1>, <Operand2>, …
```



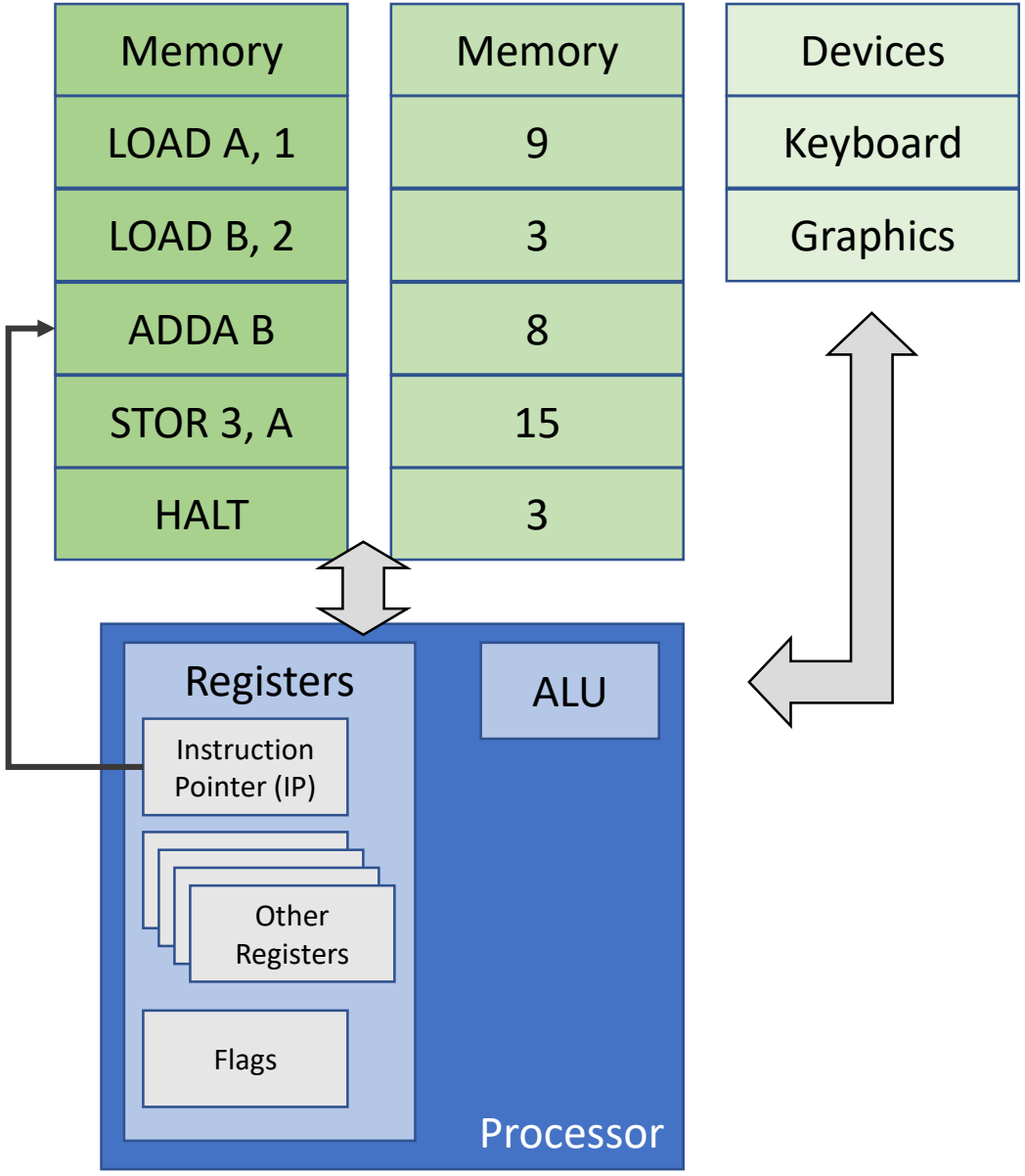
Instruction Code Formats

- In machine code, instructions are represented as binary value
- Different groups of bits represent different parts
  - Opcode: Operation to be performed
  - Operands
- Trade-off: Number of bits for instruction code
  - More operands -> Shorter programs, more memory per instruction
  - Less operands -> Longer programs, less memory per instruction
  - Example:
    - `ADD A, B, C` (C := A+B) needs space for specifying all three operands
    - `ADD A, B` (A := A+B) needs space for only two operands, but needs additional instruction to store result in C



Instruction Types

- Data Transfer
  - From/to memory to/from processor
  - From/to external devices to/from processor
- Data Operation
  - Arithmetic instructions
  - Logic instructions
  - Shift instructions
- Program Control
  - Jumps: Conditional/unconditional
  - Subroutines: Call and Return
  - Interrupts: Hardware/software
  - Exceptions/traps
  - No operation
  - Halt
- Not all processors have all instruction types
- Instructions may
  - May affect flags
  - May evaluate flags



Data Transfer Instructions

- Load data from memory into processor
- Store data into memory
- Move (copy) data within processor
- Typical mnemonics:
  - `LOAD/STOR, MOV`
- Input data from device to processor
- Output data from processor to device
- Typical mnemonics:
  - `IN, OUT`
- Device input/output can also be done via memory mapping

Data Operation

- Arithmetic operations
  - Integer addition, subtraction, multiplication, division
  - Floating-point instruction set for real values
  - Can produce overflows and exceptions for invalid operations
  - Examples: `ADD, SUB, MUL, DIV, FADD, FSUB, FMUL, FDIV`
- Logic instructions
  - Operate on bit values – true/false
  - Examples: `AND, OR, XOR, NOT`
- Shift instructions
  - Move data around within register
  - Can be used to multiply/divide by powers of 2
  - Examples: `SHL, SHR`

Program Control

- Usual flow: IP increased after each instruction
- Jumps
  - Set IP to point to new value
  - Unconditional jump: `JMP <value>`
  - Conditional jump:
    - Evaluate flag, only jump if true (1)
    - Example: Jump if zero: `JZ <value>`
  - New IP often given as label: `@LABEL`
- Subroutines
  - Needed:
    - Ability to return from subroutine
    - Local variables?
  - Store old next IP in special memory location (stack)
  - Jump to address
  - (Execute subroutine)
  - Load old next IP into IP
  - Example: `CALL <value>, RET`
  - Stack access: `PUSH <value>, POP <value>`

Program Control: Subroutines

- Required:
  - Ability to return from subroutine
  - Ability to pass in data
  - Ability to return data
  - Preservation of register values
  - Local variables?
- Simple Method:
  - Store register values on stack
  - Store input values in registers
  - Store old next IP on stack
  - Jump to address
  - Retrieve input data
  - Execute subroutine
  - Store output data in register(s)
  - Retrieve old next IP (jump back)
  - Restore register values
- Example: `CALL <value>, RET`
- Stack access: `PUSH <value>, POP <value>`
- More complex methods use stack frames for parameter passing and local variables

Program Control

- Interrupts
  - Can be used to signal events
  - Jump to subroutine (address stored in interrupt descriptor table (IDT))
  - Can also be called, e.g.: `INT 80h`
- No operation
  - Do nothing
  - Used for waiting and alignment
  - Example: `NOP`
- Halt
  - Stop execution
  - Example: `HALT`

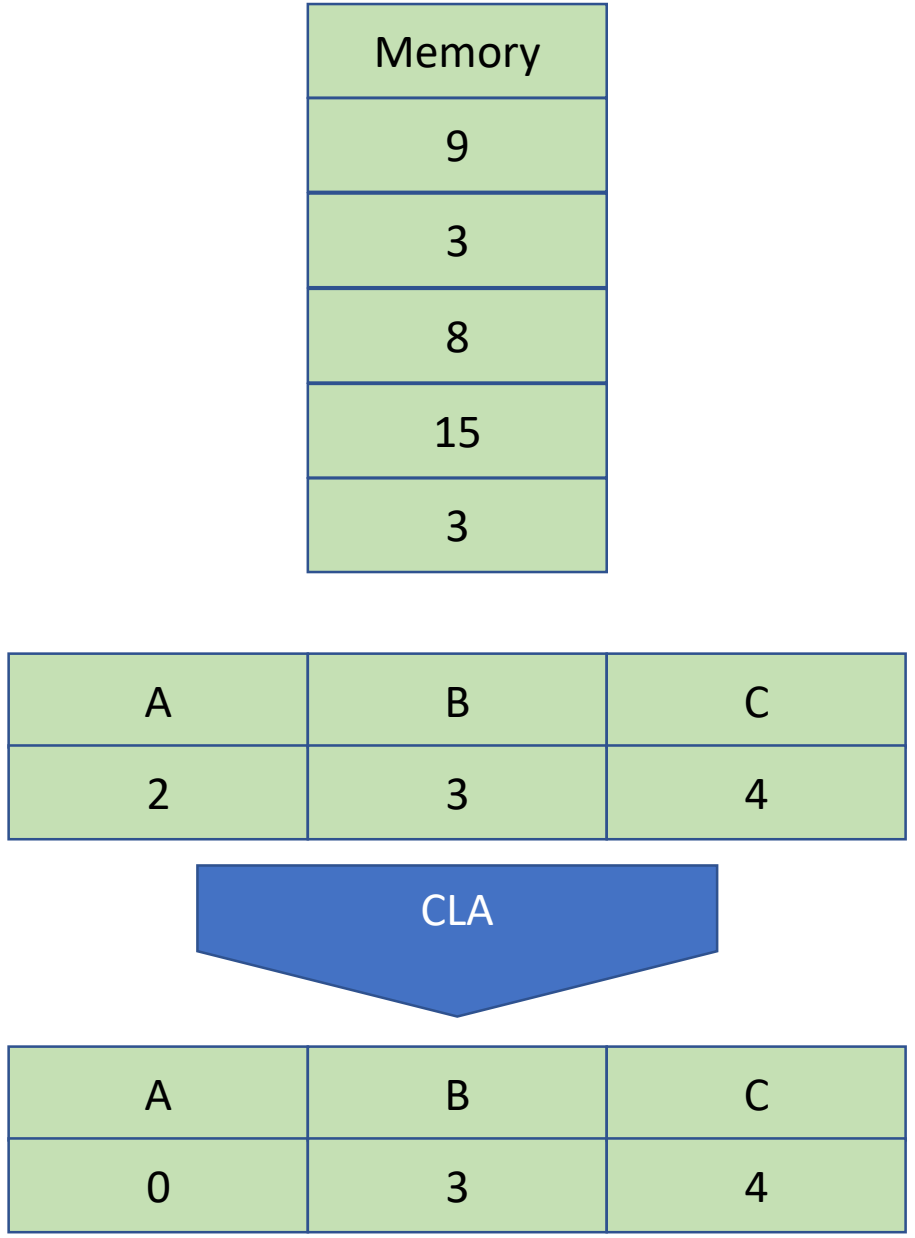
Addressing Modes

- Data needs to be moved around
- Addressing modes describe
  - Where to get data from
  - Where to put data
- Common addressing modes:
  - Implicit
  - Immediate
  - Direct
  - Indirect
  - Register direct/register indirect
  - Relative
  - Index/Base Address
- Supported addressing modes vary

Implicit Mode

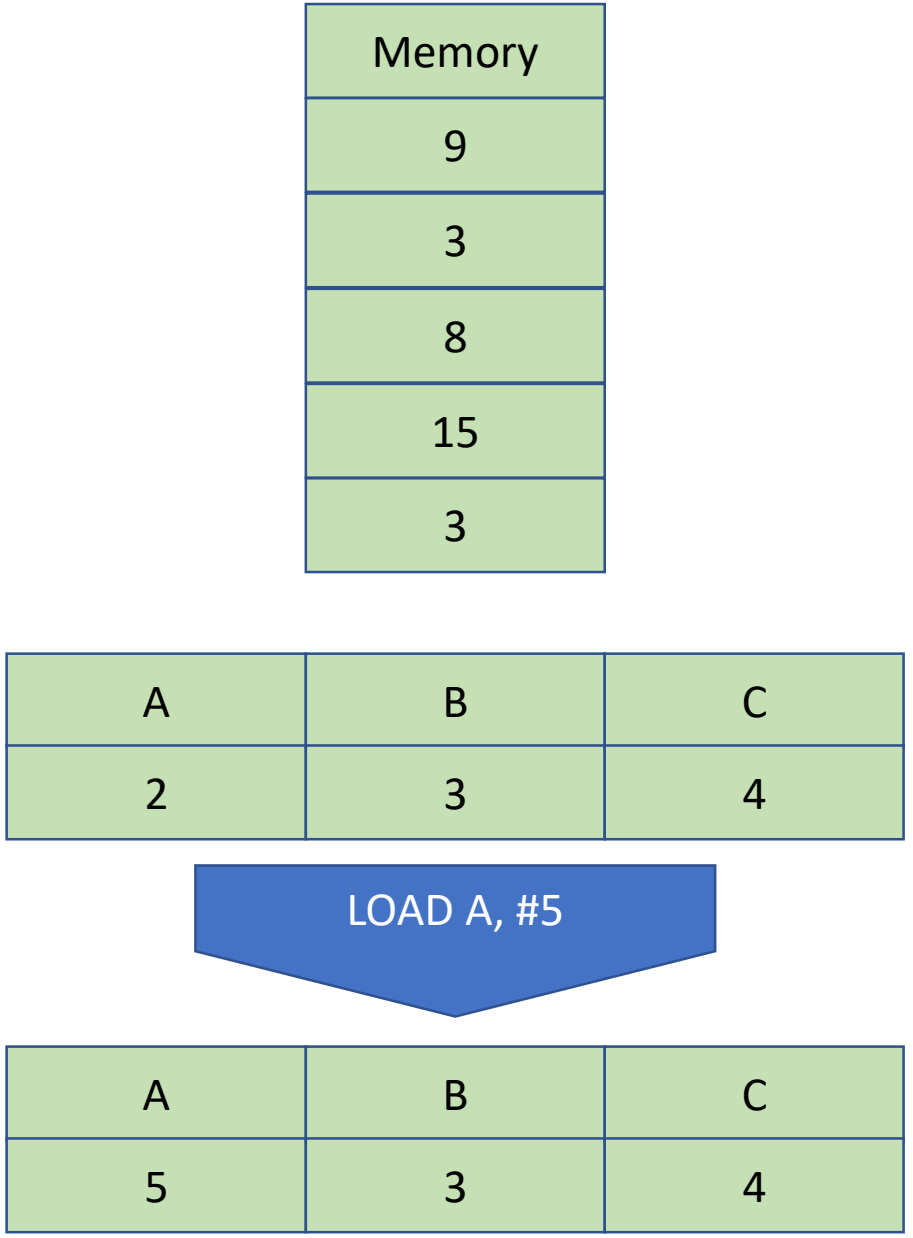
- No operand given, instruction always applies to same operand
- Examples:

`CLA`	Clear register `A`
`ADDA <v>`	Add `<v>` to register `A`



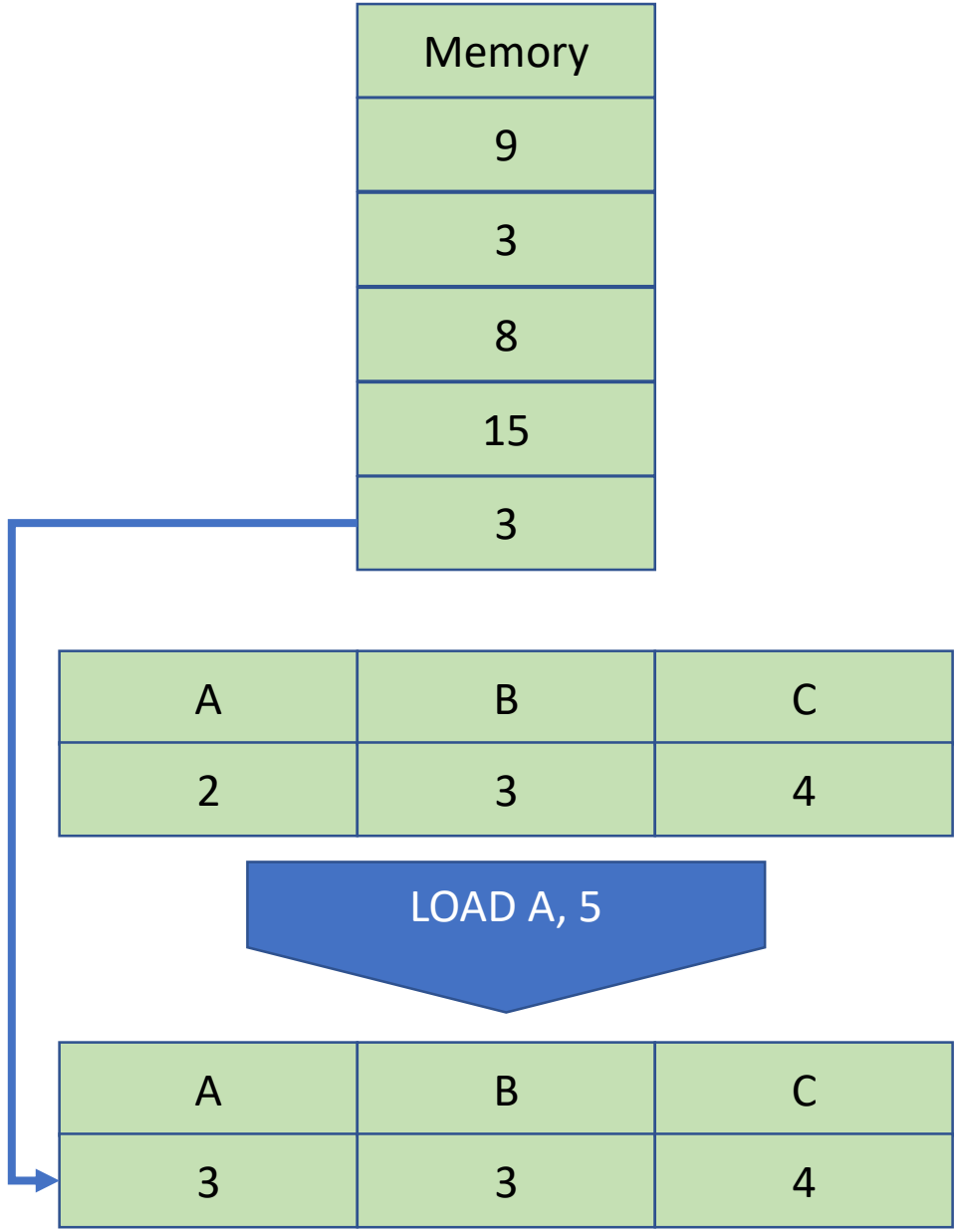
Immediate Mode

- Use the given value
- `LOAD A, #5` -> Use value 5



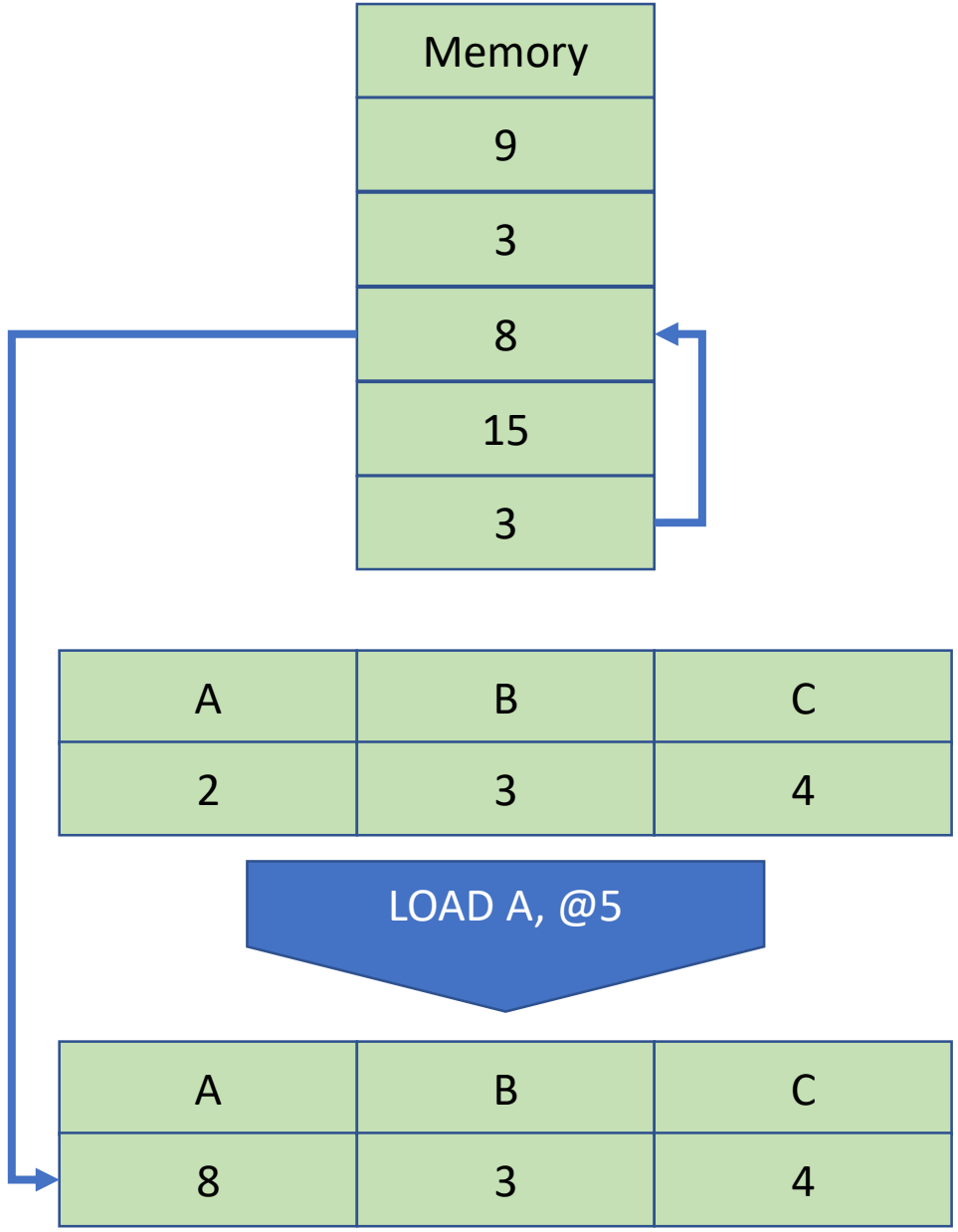
Direct Mode

- Use value at given address
- `LOAD A, 5` -> Use value at address 5



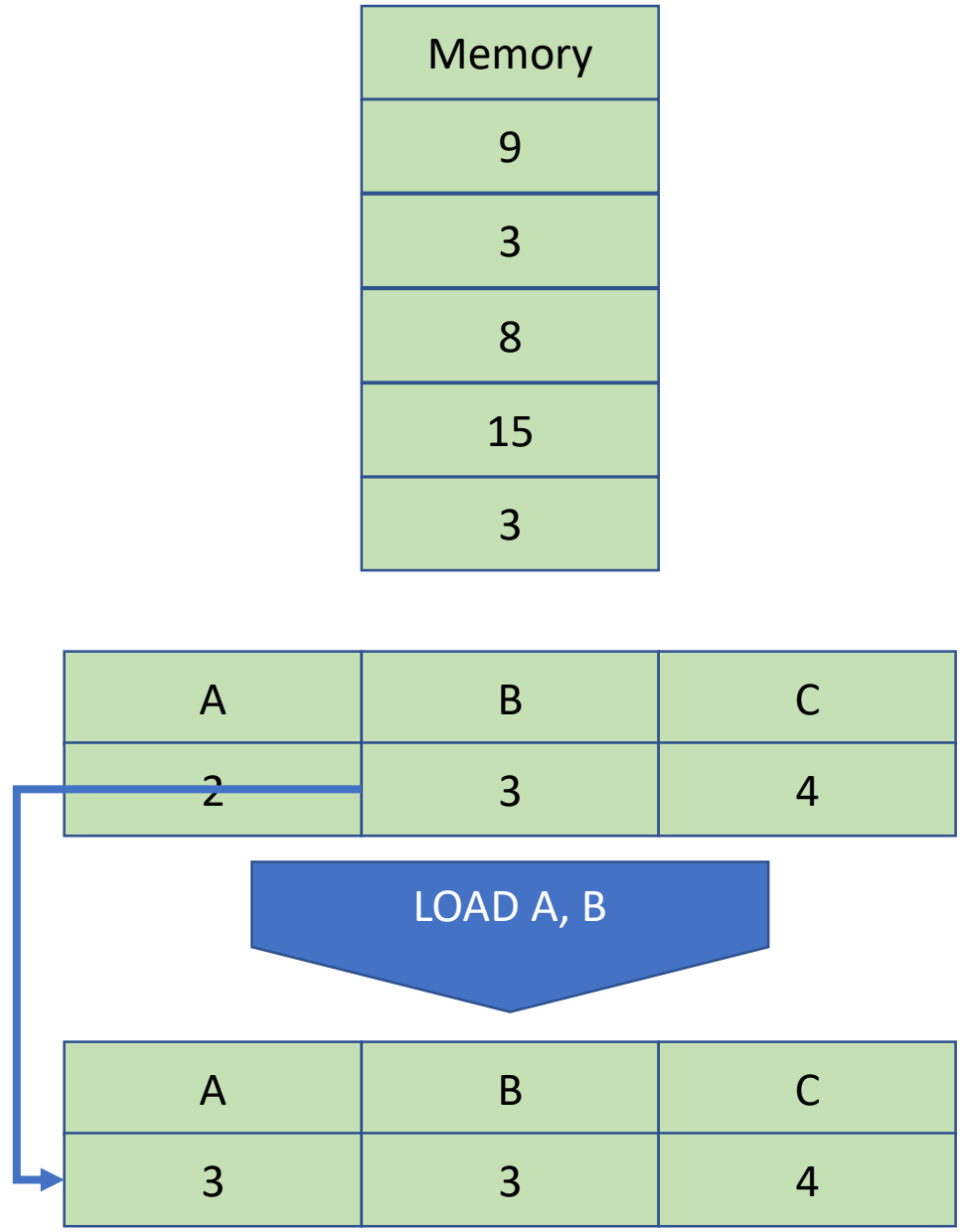
Indirect Mode

- Use value at given address as address
- `LOAD A, @5` -> Go to address 5, use value there as address for the final memory location



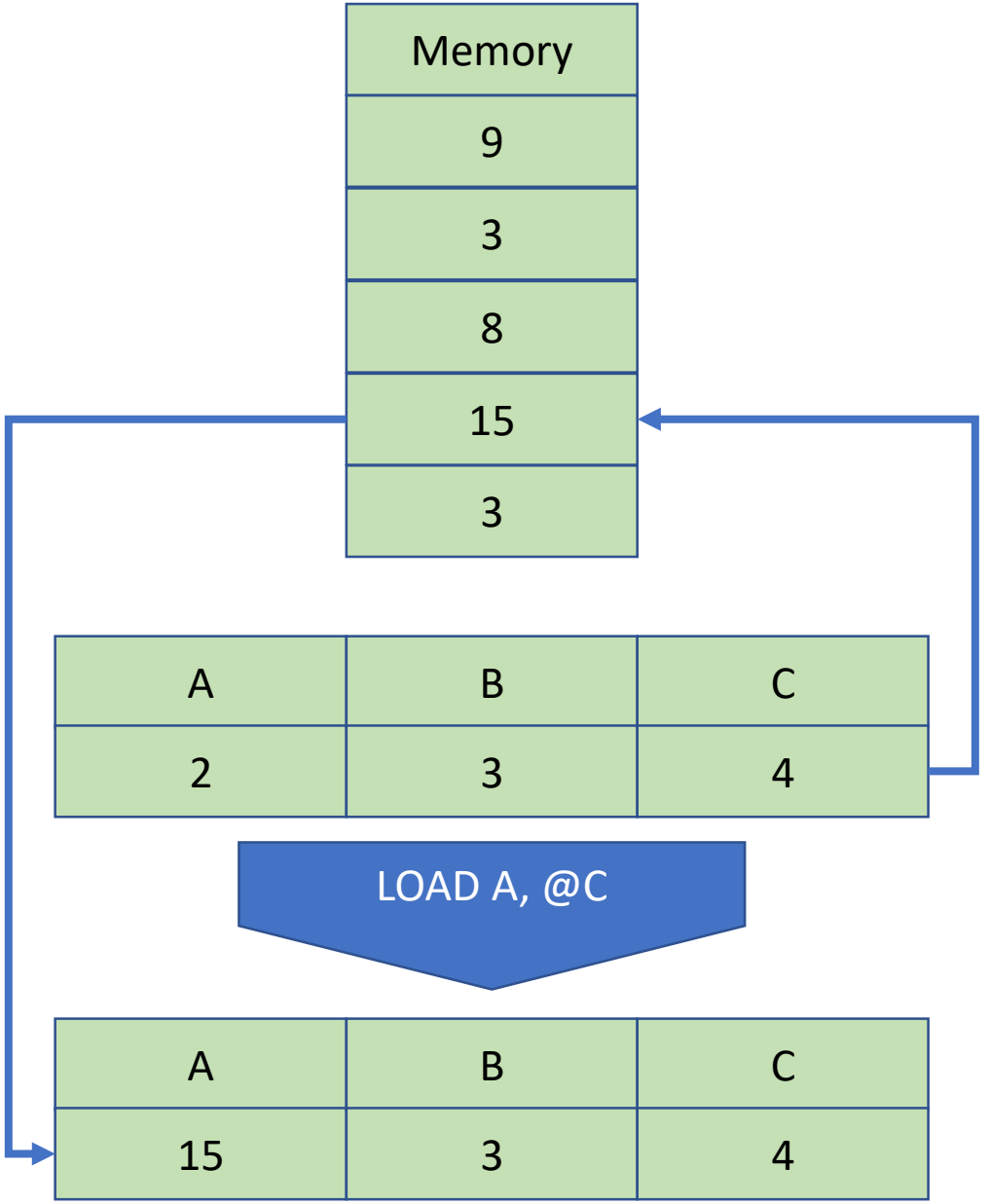
Register Direct Mode

- Use value from given register
- `LOAD A, B` -> Use value in register `B`



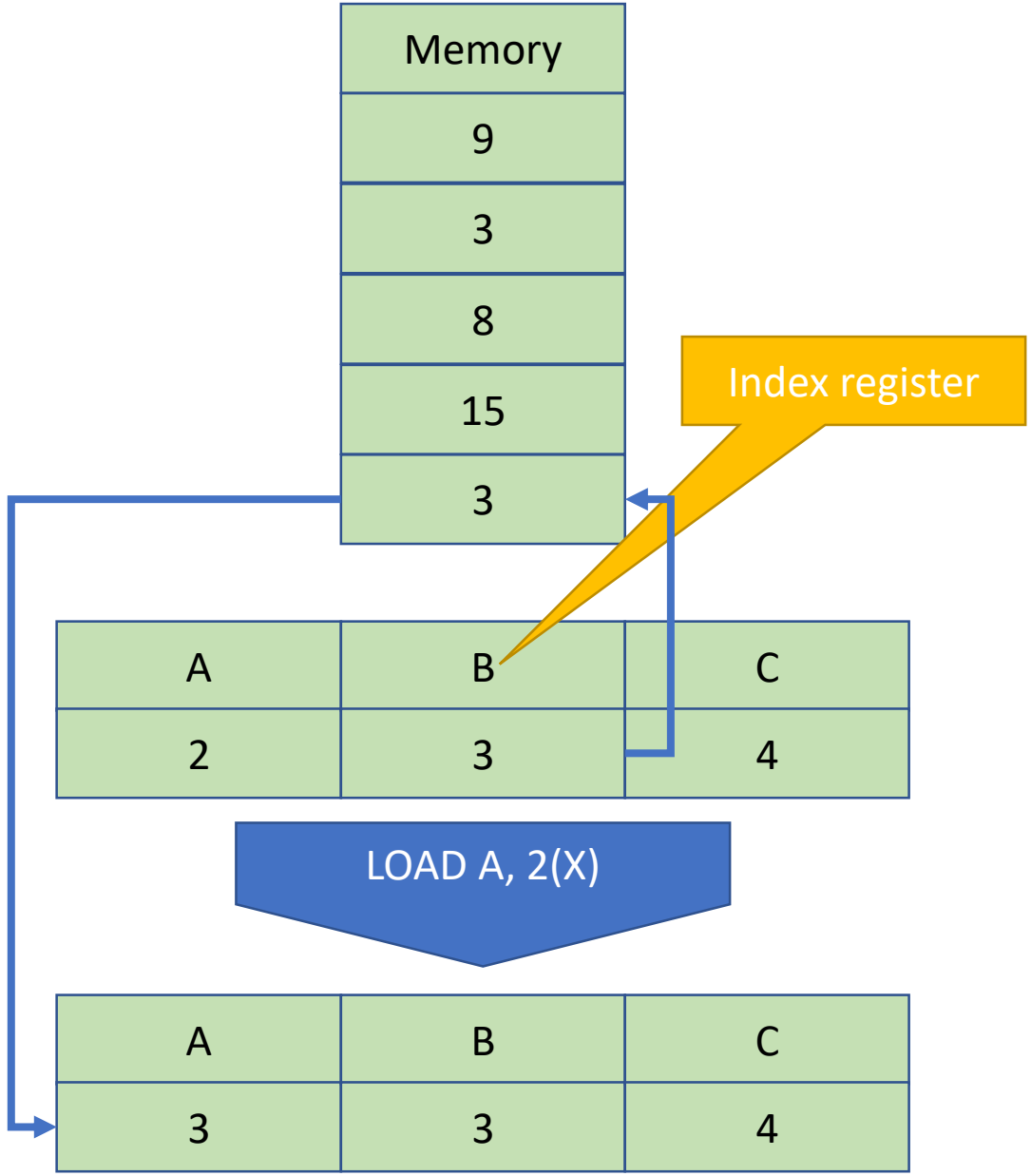
Register Indirect Mode

- Use value from given register as address
- `LOAD A, @C` -> Get address from register `C`



Relative Modes

- Address from register contents + given offset
- Index mode: Operand is base address
  - `LOAD A, 2(X)`
- Base address: Operand is offset
- Relative mode: Operand is offset, added to IP



Implementing Common Structures

- If/then/else
- While loops
- For loops
- Subroutines

Example Processor

- 5 Registers: `A, B, C, D, S`
- Stack (address is in `S`)
- Memory
- Flags:
  - Zero flag – set if last instruction resulted in zero
  - Equal flag – set by `CMP` operation
- Instruction Set
  - `INC/DEC <register>`
  - `ADD <register1>, <register2>`
  - `XOR <register1>, register2>`
  - `LOAD <register>, <memory>/STOR <memory>, <register>`
  - `CMP <register1>, <register2>`
  - `JMP`
  - `JNZ`
  - `JE`
  - `CALL @LABEL`
  - `RET`
  - `PUSH/POP`
- Addressing modes: Immediate, Direct, Indirect, Register Direct, Register Indirect

If/then/else

```
IF A == 3 THEN
        B := 4
ELSE
        B := 5
ENDIF
```

```
LOAD B, #3
CMP A, B
JE @EQUAL
LOAD B, #5
JMP @ENDIF
@EQUAL:
        LOAD B, #4
@ENDIF
```

While Loop

```
WAIT_FLAG = 0
WHILE !WAIT_FLAG DO
       A := A+B
DONE
```

```
LOAD D, #1
XOR C, C
STOR 1, C
@WHILE:
       LOAD C, 1
       CMP C, D
       JE @ENDWHILE
       ADD A, B
       JMP @WHILE
@ENDWHILE:
```

For Loop

```
FOR (i=0; i<10; i++) DO
        A := A + i
DONE
```

```
LOAD C, #0
LOAD D, #10
@FOR:
        CMP C, D
        JE @ENDFOR
        ADD A, C
        INC C
        JMP @FOR
@ENDFOR
```

Subroutines

```
FUNCTION DO_SOMETHING()
        …
ENDFUN

…

DO_SOMETHING()
```

```
@DO_SOMETHING:
        …
        RET


…
CALL @DO_SOMETHING
…
```

Subroutines with parameters

```
FUNCTION DO_SOMETHING(INT X, INT Y)
        …
ENDFUN

…

DO_SOMETHING()
```

```
@DO_SOMETHING:
        ADD A, B
        RET


PUSH B
PUSH A
LOAD B, #3
LOAD A, 4
CALL @DO_SOMETHING
POP A
POP B
…
```

Subroutines with return values

```
FUNCTION DO_SOMETHING()
        …
        X := RESULT
        RETURN X
ENDFUN

…

DO_SOMETHING()
```

```
@DO_SOMETHING:
        …
        LOAD A, C
        RET



…
PUSH A
PUSH B
PUSH C
CALL @DO_SOMETHING
LOAD D, A
POP C
POP B
POP A
…
```

Advanced Topics

- Pipelines
- Out-of-order execution
- Speculative execution -> Spectre, Meltdown
- Self-modifying code

Self-modifying Code

- In von-Neumann architecture, code is just data
- Programs may modify themselves (or other programs)
- Sometimes used by viruses
- Example:

```
@WHILE:
     STOR @IP, #66
     JMP @WHILE
@DONE:
```

Binary code for
JMP @DONE

Conclusion

- Assembly languages are highly platform-dependent
- Many different instruction sets exist, e.g. x86, MIPS, 8051
- Very close to hardware
  - Limited atomic capabilities
  - Can be tedious
- Appropriate for
  - Highly optimised code
  - Hard real-time
  - Embedded systems (limited resources)
  - Malicious software