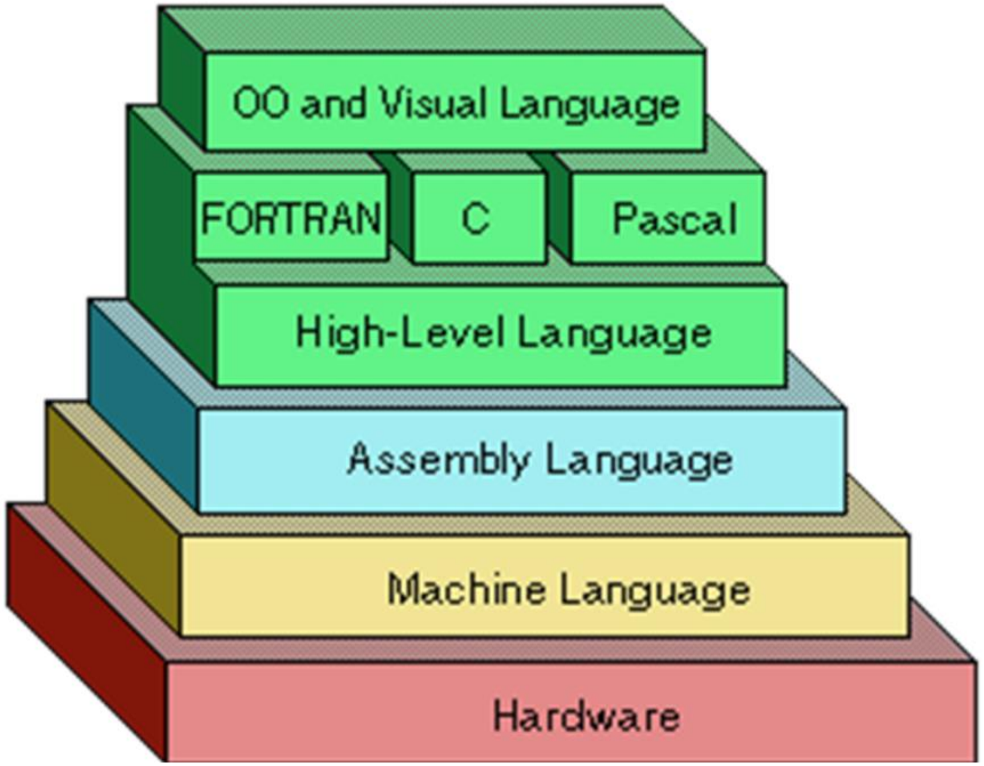
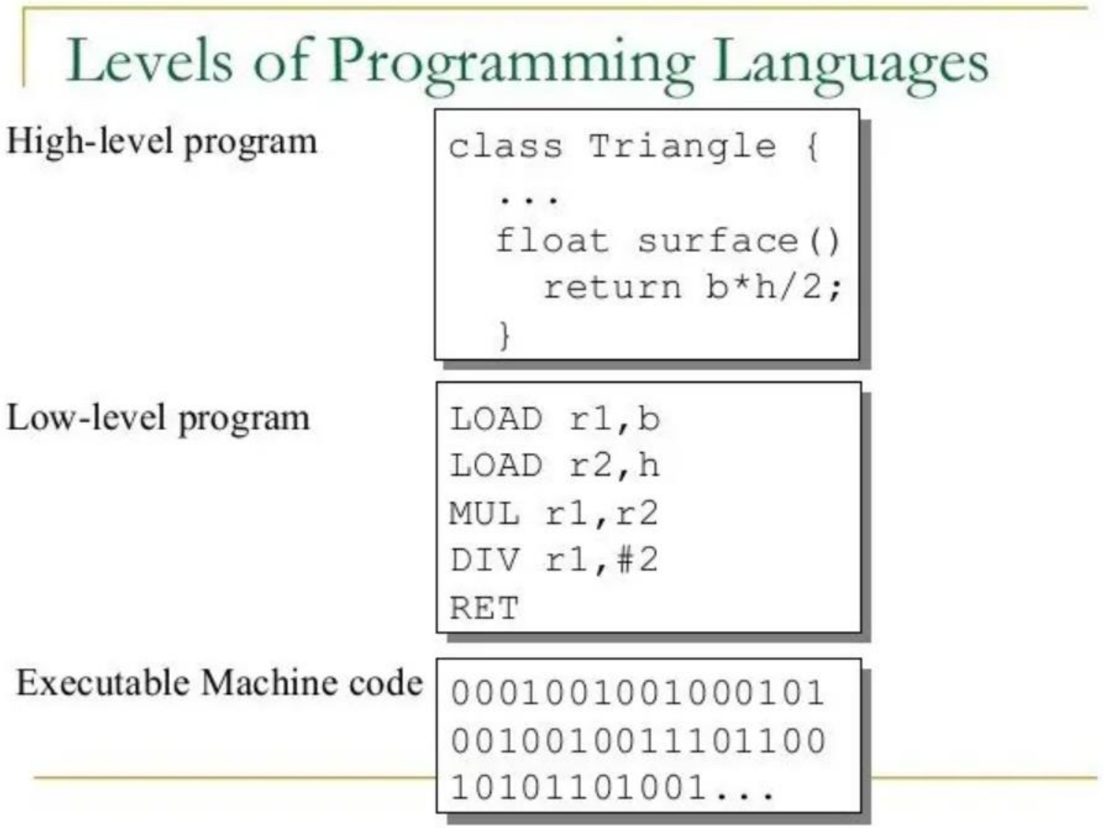
4. Machine-language programming



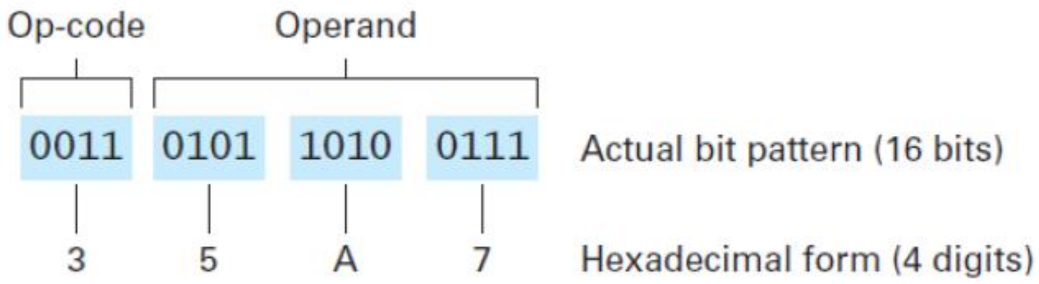


Machine language

- Writing programs in microcode is very time-consuming:
  - Performing even simple tasks (e.g. multiplication) requires multiple microinstructions
- Machine language provides a (little) bit more user-friendly way
  - One step higher than microcode in abstraction level
  - Still quite far removed from higher level languages (C, Python)
  - No need to care about what happens in which clock cycle
- In order to understand machine language programs, microprogrammed computers need an interpreter
  - Interpreter reads the program and translates it “on the fly” to microcode
  - Interpreter program is written on microcode and stored in MPM

Symbolic machine language

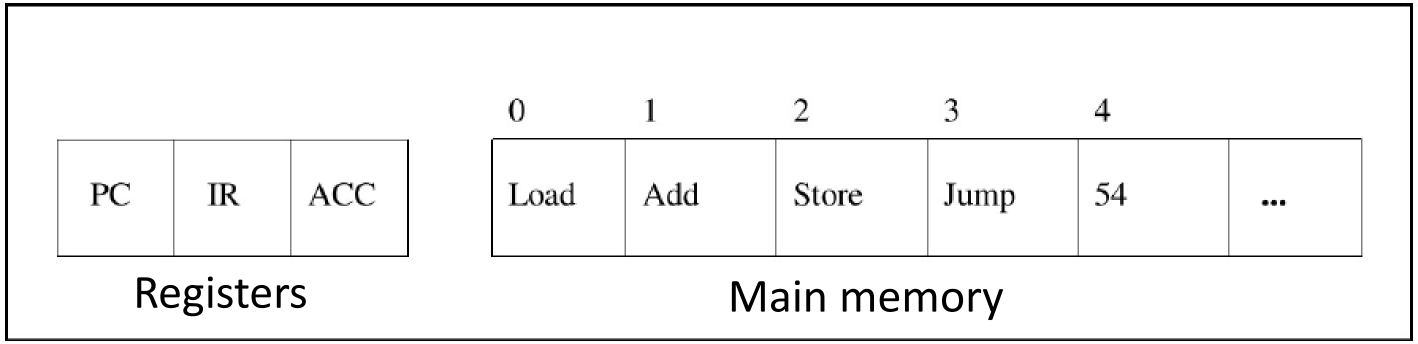
- True machine code is raw binary data, that consists of an operation part and an operand part (as we learned before)
  - The binary code can be shortened to octal or hexadecimal form to improve readability



- The same code can be represented in symbolic machine language:
  - Operation codes are replaced by (more descriptive) operation names
  - Operands are given in decimal form
  - Further improved readability

Machine language example

- As the name implies, machine languages are “tied” to the machine: they are processor-specific (not necessarily in general, but at least in detail level)
- Let’s investigate how machine language works via an example computer that has the following features:
- Special registers
  - PC (program counter; similar to MPC+MPM in microprogrammed computer)
  - IR (instruction register; similar to MIR in microprogrammed computer)
  - ACC (accumulator; includes the data that is currently in process)
- Main memory (RAM)
  - Machine language programs
  - Data to be handled



Machine language example: operations

- Our example computers machine language includes the following commands
- This language uses single operands: one is given, the other one (if needed) is always the accumulator (ACC)
  - Two- or even three-operand languages exist, though

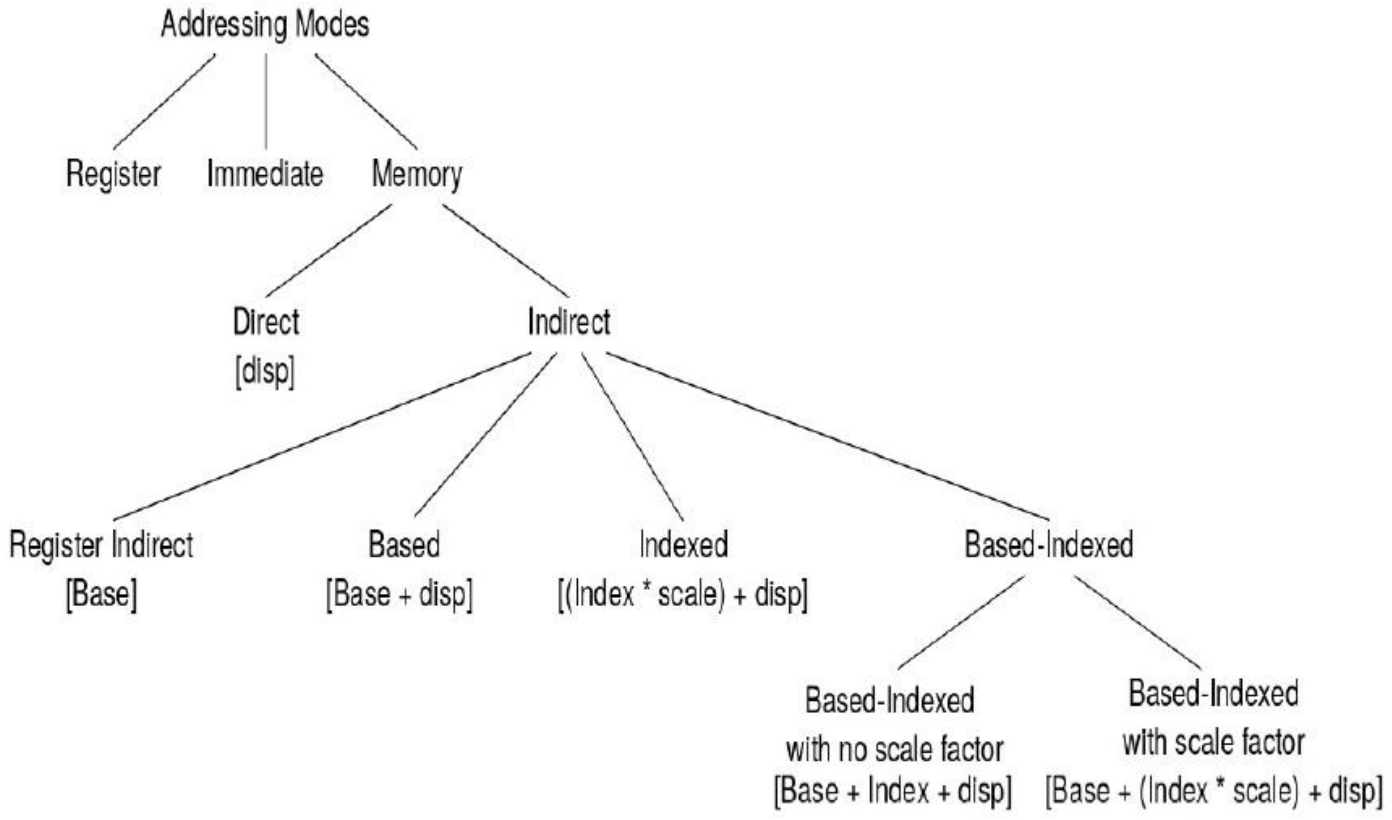
Symbolic command	Action
LOAD M	ACC ← (M)
STORE M	(M) ← ACC
ADD M	ACC ← ACC + (M)
SUBTRACT M	ACC ← ACC - (M)
MULTIPLY M	ACC ← ACC * (M)
DIVIDE M	ACC ← ACC / (M)
JUMP M	Jump to M
JUMPZERO M	Jump to M, if ACC = 0
JUMPNEG M	Jump to M, if ACC < 0
JUMPSUB M	Jump to subprogram that starts from M
RETURN M	Return from subprogram that started from M

Machine language example: features

- Commands are stored in main memory as binary code
- Commands are executed one at a time in given order, unless the order is specifically changed (for example via JUMP commands)
- Word size 16 bits, memory address 12 bits (= 4096 different addresses)
  - 4 bits left for operation code, so 16 different operations possible (in theory)
- In our example language, constant numbers are not available
  - All numbers that we use in calculation must be stored in main memory
- Language can be expanded to cover also “immediate addressing” with an operation called LOADI
  - This way numbers can be given as operands

Addressing modes

- Addressing mode means the way how the operand part of the command defines what the true operand is
- Possibilities:
  - Immediate
  - Direct
  - Indirect
    - Indirect indexed
    - Indirect based-indexed



Addressing modes

- Immediate addressing:
  - Operand specifies the data that will be loaded to ACC
  - LOADI M $\Rightarrow$ ACC $\leftarrow$ M
- Direct addressing:
  - Operand specifies the memory address of the data that will be loaded to ACC
  - LOAD M $\Rightarrow$ ACC $\leftarrow$ (M)
- Indirect addressing:
  - Operand specifies the memory address that contains the memory address of the data that will be loaded to ACC
  - LOADID M $\Rightarrow$ ACC $\leftarrow$ ((M))

Addressing modes

- In some cases, the data that we want to use may change depending on which stage of program execution we’re in
- We can use indexed addressing to take this into account
- Indexed addressing:
  - The sum of operand and index register value gives the memory address of the data that will be loaded to ACC
  - LOADIX M $\Rightarrow$ ACC $\leftarrow$ (IR+M)
- Indirect indexed addressing:
  - The sum of value found in memory address given by the operand and index register value gives the memory address of the data that will be loaded to ACC
  - LOADIDX M $\Rightarrow$ ACC $\leftarrow$ (IR+(M))

Addressing modes: example

- State of memory & IR are following:

Index register value	4

Address	280	281	282	283	284	285	286	…
Content	282	87	13	27	16	66	77	…

- What are the ACC values after following commands?

Addressing mode	Command	ACC value after command
Immediate	LOADI 280	
Direct	LOAD 280	
Indirect	LOADID 280	
Indexed	LOADIX 280	
Indirect indexed	LOADIDX 280	

Addressing modes: example

- State of memory & IR are following:

Index register value	**4**

Address	**280**	**281**	**282**	**283**	**284**	**285**	**286**	**…**
Content	282	87	13	27	16	66	77	…

- What are the ACC values after following commands?

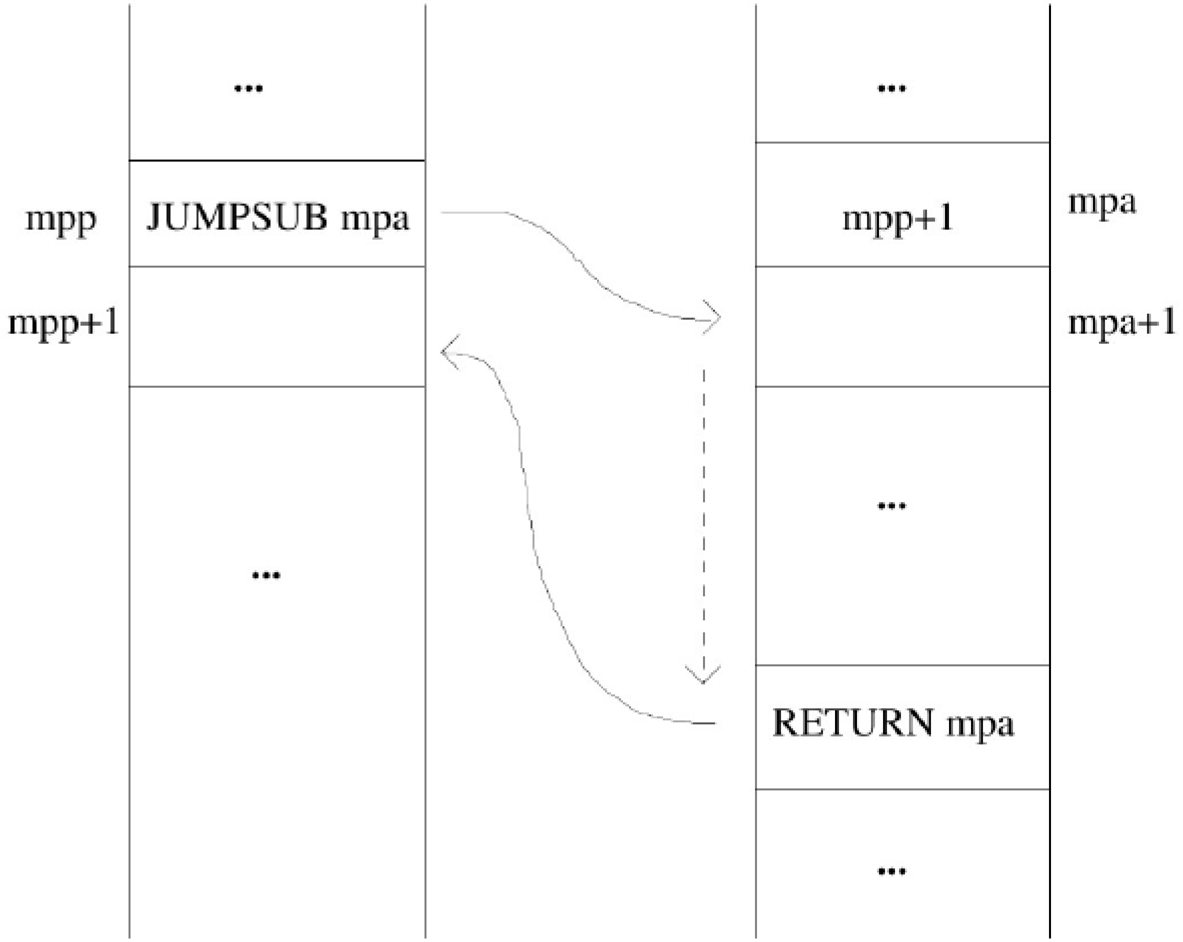
Addressing mode	Command	ACC value after command	
Immediate	LOADI 280	280	
Direct	LOAD 280	282	(280) → 282
Indirect	LOADID 280	13	((280)) → (282) → 13
Indexed	LOADIX 280	16	(280+4) → (284) → 16
Indirect indexed	LOADIDX 280	77	((280)+4) → (282+4) → 77

Execution order of commands

- Execution order of commands can be changed by jumps
- Unconditional jump: JUMP M
  - Jumps to memory address M, no matter what
- Conditional jumps:
  - JUMPZERO M – jumps to memory address M, if ACC = 0
  - JUMPNEG M – jumps to memory address M, if ACC < 0
- Jump to subprogram and back:
  - JUMPSUB M – jumps to address M, where the subprogram begins
  - RETURN M – returns from subprogram that started from memory address M

Jumps to subprograms

- When execution proceeds to mpp, we jump to subprogram that starts from mpa
- Execution continues from mpa+1
- After the subprogram has been executed, final RETURN tells where it started from (mpa)
- From this address we find the information, where we should return in the original program (mpp+1)
- Same subprogram can be called multiple times



Example: Two to the power of n

- Let’s write a program that calculates the value of two to the power of n
- First, a pseudo-code:

```
MODULE exp(n) RETURNS 2^n
  value := 1
  WHILE n > 0 DO
    n := n - 1
    value := value + value
  ENDWHILE
RETURN value
ENDMODULE
```

Why in this order?

Idea: output value starts from 1 and is doubled (= value + value) n times

Example: Two to the power of n

- Then the program in symbolic machine language:

```
MODULE exp(n) RETURNS 2^n
  value := 1
  WHILE n > 0 DO
    n := n - 1
    value := value + value
  ENDWHILE
RETURN value
ENDMODULE
```

Now we see the reason for "strange" order; n was already in the ACC, so changing it now means not having to load it again later!

Memory address	Command	Explanation
371	LOAD 383	Load 1 to ACC
372	STORE 382	Store 1 as initial value of F
373	LOAD 381	Load n to ACC
374	JUMPZERO 384	If n = 0, jump to 384 (aka. Stop)
375	SUBTRACT 383	Subtract 1 from n
376	STORE 381	Save new value of n
377	LOAD 382	Load function value F to ACC
378	ADD 382	ACC = F + F
379	STORE 382	Save new value of F (F = ACC)
380	JUMP 373	Jump to beginning of iteration
381	n	Parameter
382	0	Function value F
383	1	Number constant

Example: $2^x + 2^y$ using subprogram

- If we want to calculate what is $2^x + 2^y$, a natural solution would be to use the previous program twice (1st as n = x and 2nd time as n = y)
- For educational purposes, let's see how this could be implemented using a subprogram!
- First the pseudo-code:

```
MODULE sum(x,y) RETURNS 2^x + 2^y
  RETURN exp(x)+exp(y)
ENDMODULE
```

Example: $2^x + 2^y$ using subprogram

- Main program on the left, subprogram + data on the right

Address	Command	Explanation
287	LOAD 314	Load x to ACC
288	STORE 311	Save x as subprogram input
289	JUMPSUB 299	Execute subprogram (i.e. calc. 2^x)
290	LOAD 312	Load 2^x to ACC
291	STORE 316	Save 2^x as end result value
292	LOAD 315	Load y to ACC
293	STORE 311	Save y as subprogram input
294	JUMPSUB 299	Execute subprogram (i.e. calc. 2^y)
295	LOAD 312	Load 2^y to ACC
296	ADD 316	Add 2^x to ACC value (2^y)
297	STORE 316	Store the end result
298	JUMP 317	Stop program execution

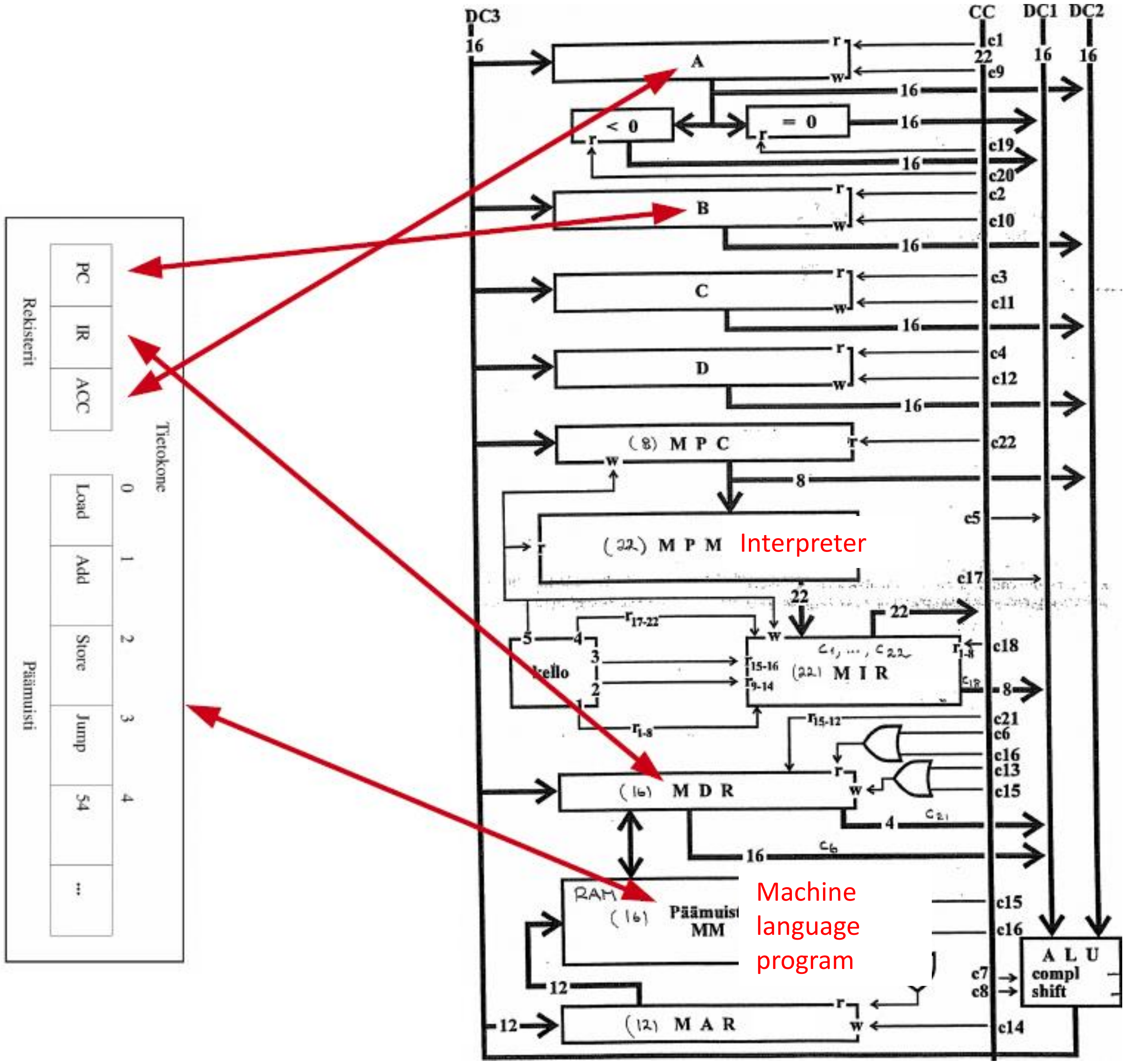
Address	Command	Explanation
299	0 / 290 / 295	Begin / Return1 / Return2
300	LOAD 313	Load 1 to ACC
301	STORE 312	Save 1 as subprogram result w
302	LOAD 311	Load subprogram input to ACC
303	JUMPZERO 310	If n = 0, jump to end of subpr.
304	SUBTRACT 313	Subtract 1 from n
305	STORE 311	Save new n value
306	LOAD 312	Load subprogram result w
307	ADD 312	ACC = w + w
308	STORE 312	w = ACC
309	JUMP 302	Jump back to iteration start
310	RETURN 299	Return to main program
311	0	Subprogram input n
312	0	Subprogram result w
313	1	Constant value
314	x	Main program input x
315	y	Main program input y
316	0	End result

Machine language interpreter for our microprogrammed computer

- Interpreter is a microprogram, which understands and executes machine-language commands
- Operating principle:
  - Fetch the command from main memory
  - Find out the contents of the command
  - Execute the command
- In order to complete this task, the interpreter must remember the address of the next command
- From our example machine language to example microprogrammed computer:
  - ACC = Register A
  - IR = MDR as command register (4 most significant bits for the command)
  - PC = Register B as program counter (we could use C or D too, though)

Machine language interpreter for our microprogrammed computer

- Graph illustrates the relationships between the registers



Summary

- Machine language is processor- or computer-specific programming language
  - Can be presented in either symbolic or numerical (binary, hexadecimal…) form
- Several symbolic machine languages can be implemented for the same processor
- The machine language of “real-world” computers is more complex than of our example computer
  - Larger word size, more operations
- The interpreter program has been stored in read-only memory (MPM) beginning from memory address 0
  - Interpreter program execution is started by setting MPC to zero
- Programs written on high-level languages will be converted to machine language via a compiler (or an interpreter) and the machine language program will be executed using the microcode interpreter

Thank you for listening!