

CSCU9V4

Systems

Systems lecture 10

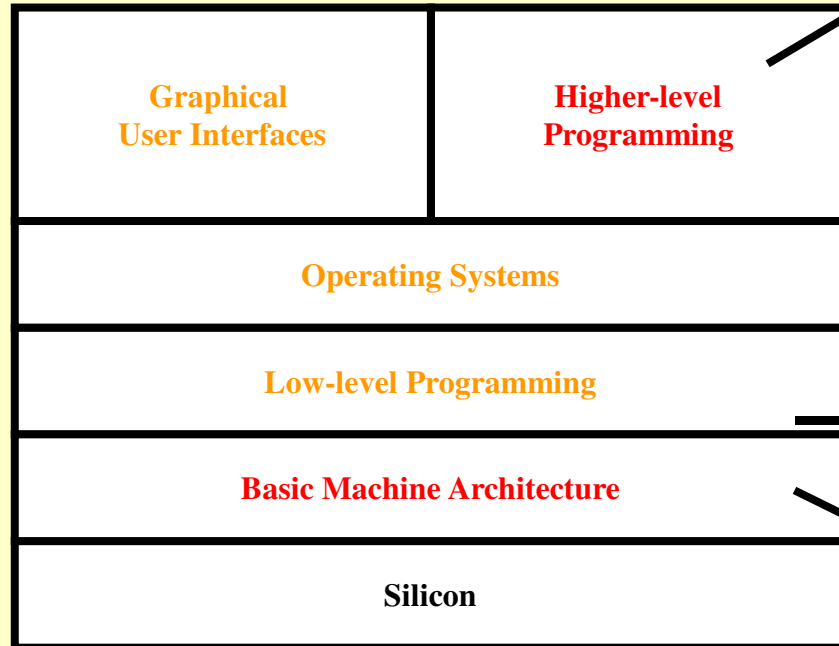
Computer Organisation

A Virtual Machine

Graphical User Interfaces	Higher-level Programming
Operating Systems	
Low-level Programming	
Basic Machine Architecture	
Silicon	

the story so far

• java

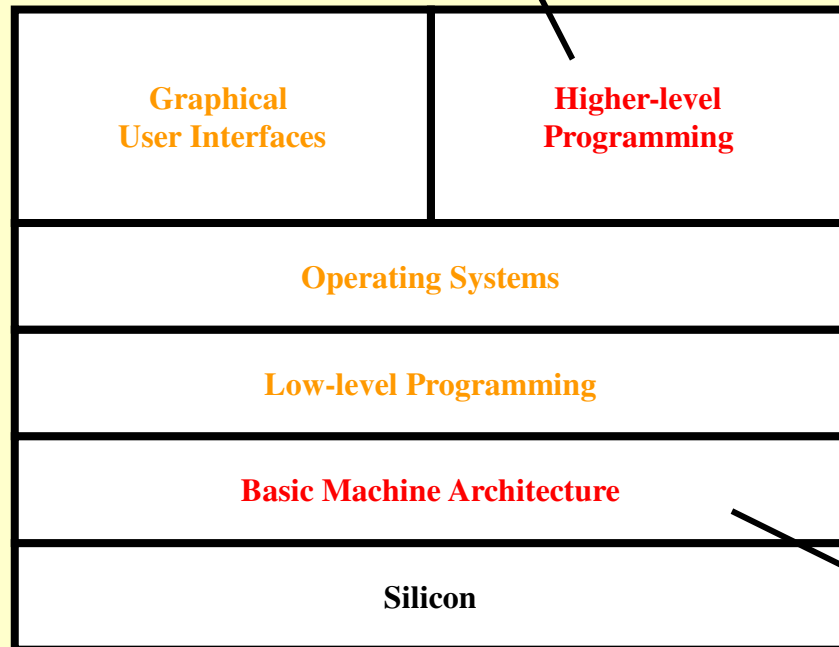


- binary
- hexadecimal
- characters & strings
- integer arithmetic
- 2's complement
- floating point

- CPU
- MM
- buses

convenient
for people

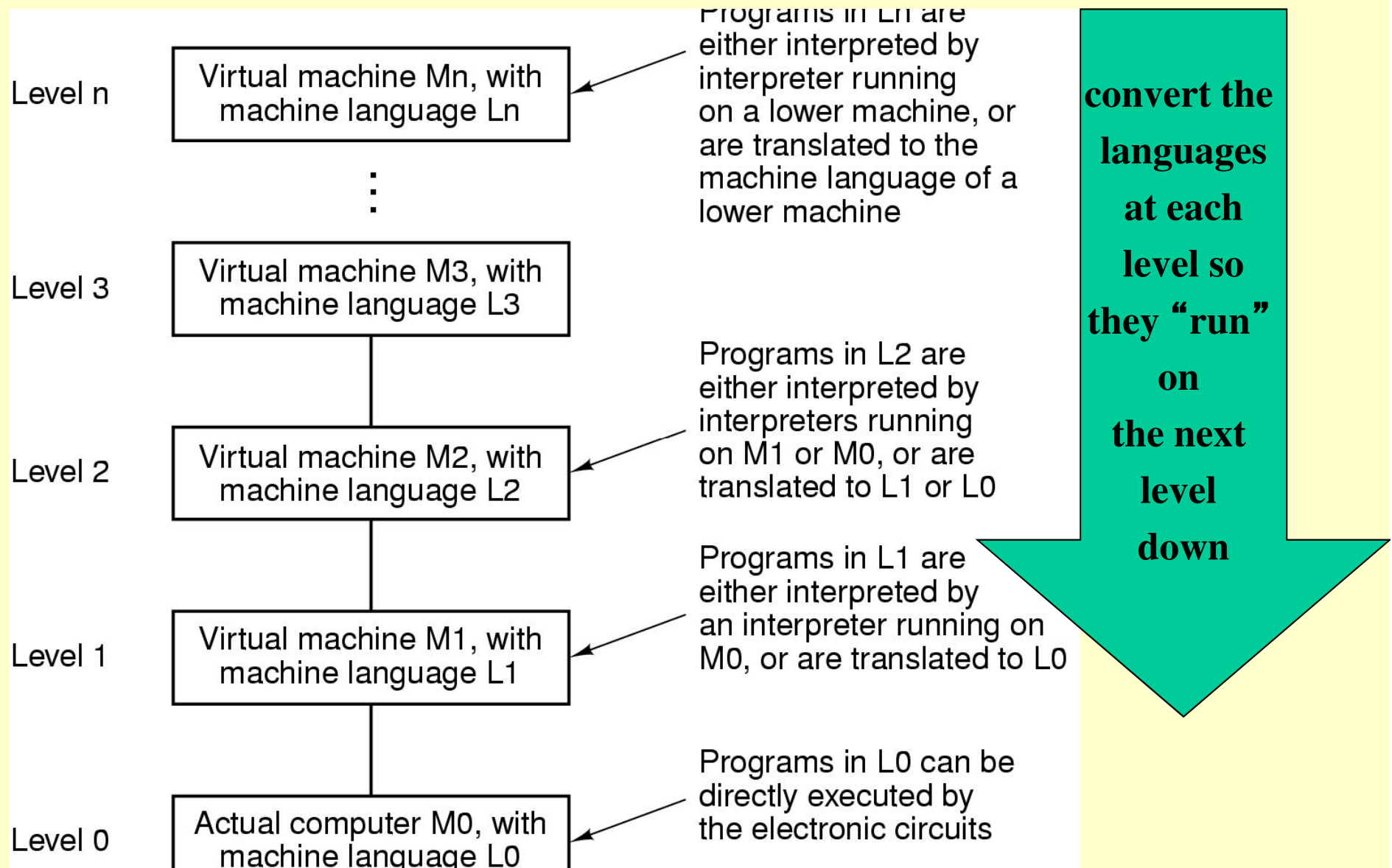
how do we run programs ?



Convert the
languages
at each level so
they “run” on
the next level
down

convenient
for the
technology

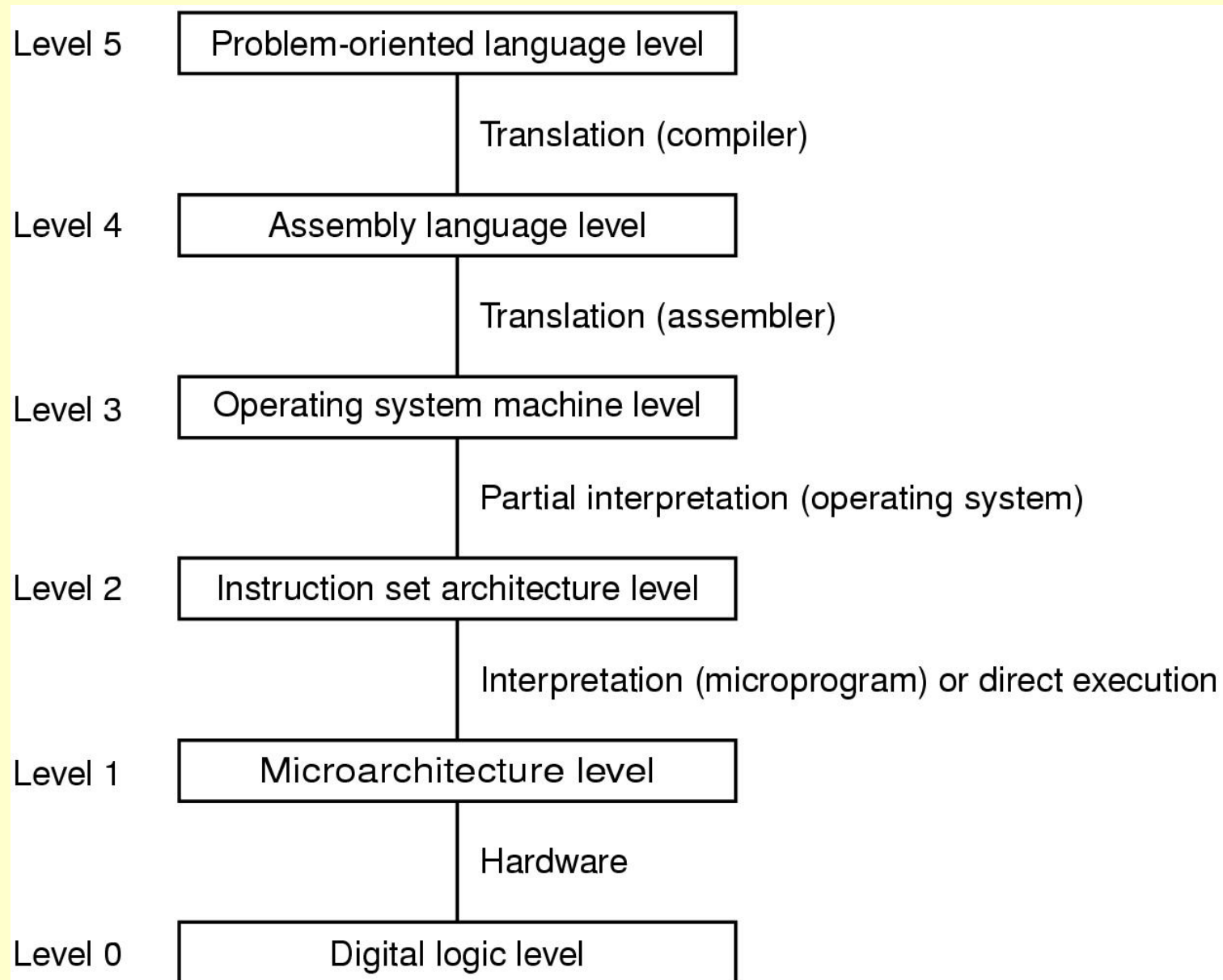
general case: multilevel machine (Tanenbaum)



What happens at each stage?

- If we have a language, say L_n we want to “run” on M_n
- How do we “implement” M_n ? (only m_0 is implemented directly!)
- Two ways:
 - Replace each instruction of program written in L_n with an equivalent set of L_{n-1} instructions
 - M_{n-1} executes the L_{n-1} instructions (**Translation**)
 - Write a program in L_{n-1} which takes programs in L_n as input data
 - program executes the equivalent set of L_{n-1} instructions for every L_n instruction (**Interpretation**)
 - Both are widely used: generally translation is more efficient, but interpretation easier to debug
 - Combinations of both are also used
- Many layers may be used.

An example computer (Tanenbaum)

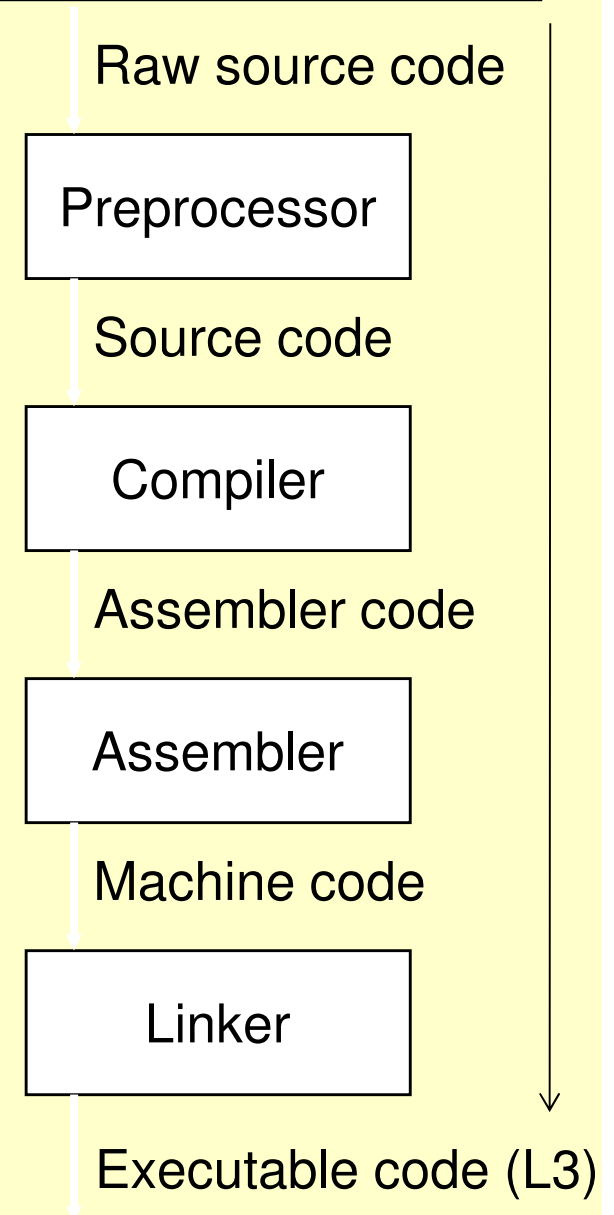


Problem-oriented High-level Languages

- There are many different **types** of language, and many different languages of each.
- Imperative
 - Basic, C, Pascal, Ada, COBOL, FORTRAN, ...
- Object-oriented
 - C++, C#, Java, Objective C, Eiffel,...
- Functional
 - Miranda, Lisp, Haskell,...
- Logic
 - Prolog
- There are ***hundreds*** of different programming languages.
 - There are also many **scripting** languages: usually directly executed by some software package
- The language used is chosen to suit the application domain
 - (it must, of course, be available for the target hardware and O.S.).

Compiler & Linker

- Compiler translates a high level language to level 3 or 4
- Compiler usually come with a group of tools
- Pre-processor
 - Putting together modules of source code
 - Translating macros
- Compiler
 - Lexical analysis, Syntax check, semantic check
 - Generate intermediate code (simplified)
 - Code optimising, assembler code generator
- Assembler
 - Sometimes included in compiler
 - Generates machine code and symbol table
- Linker
 - Changes movable addresses to absolute addresses
 - Combines several parts of a program (library)
 - May originate from various compiler runs



An Example

- High-level language:

- `xlevel = ylevel + 2;`

- Assembly language

- `LOAD R1, 44(R7)`

- `ADD R1, #2`

- `STORE R1, 36(R7)`

- Relocatable machine code

- `0001 01 00 00000000 * (load,normal address)`

- `0011 01 10 00000010 * (add,direct mode)`

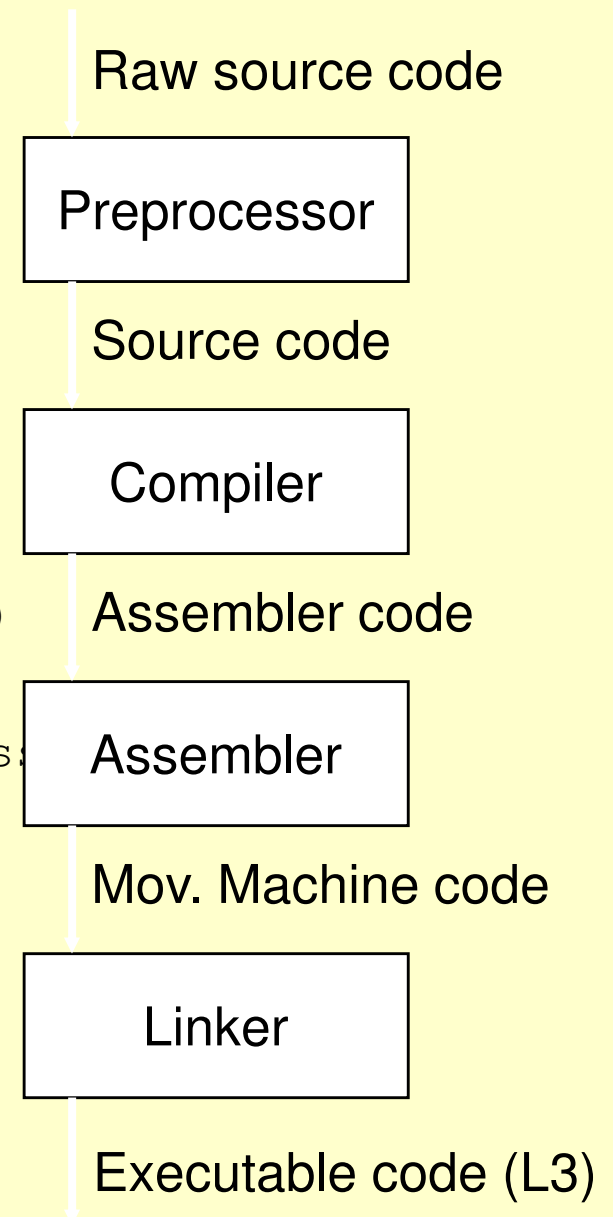
- `0010 01 00 00000100 * (store,normal address)`

- Executable machine code (start at `Locn=00001111`)

- `0001 01 00 00001111 (load, 15)`

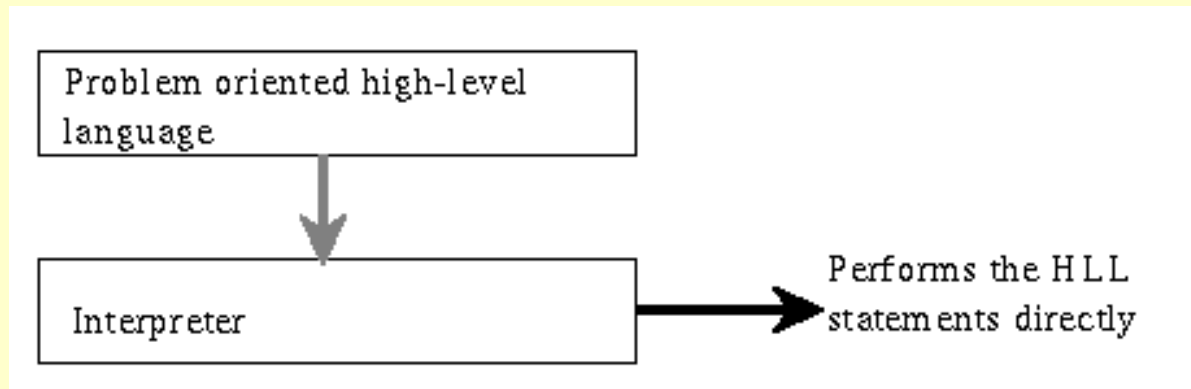
- `0011 01 10 00000010 (add, 2)`

- `0010 01 00 00010011 (store, 19)`



Interpreter

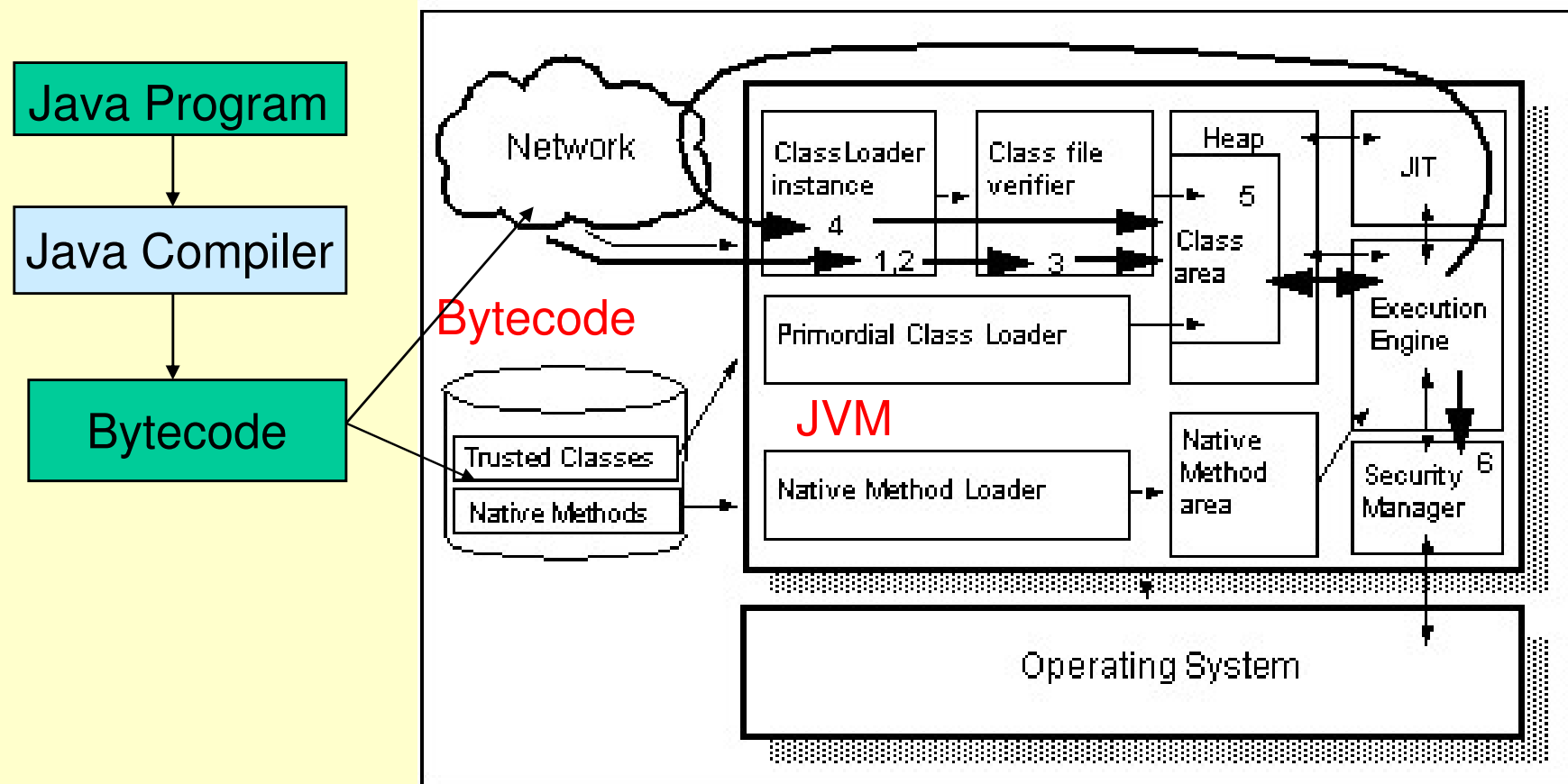
- An *interpreter* is a program which reads in statements in some language, and directly executes them.



- BASIC
- Java virtual machine is an interpreter
 - Does not execute Java directly, but interprets the Java bytecode
 - Portable (i. e. runs on a wide variety of machine types)
- Most scripting languages are of this form (PERL, R, PYTHON, JavaScript, ...)

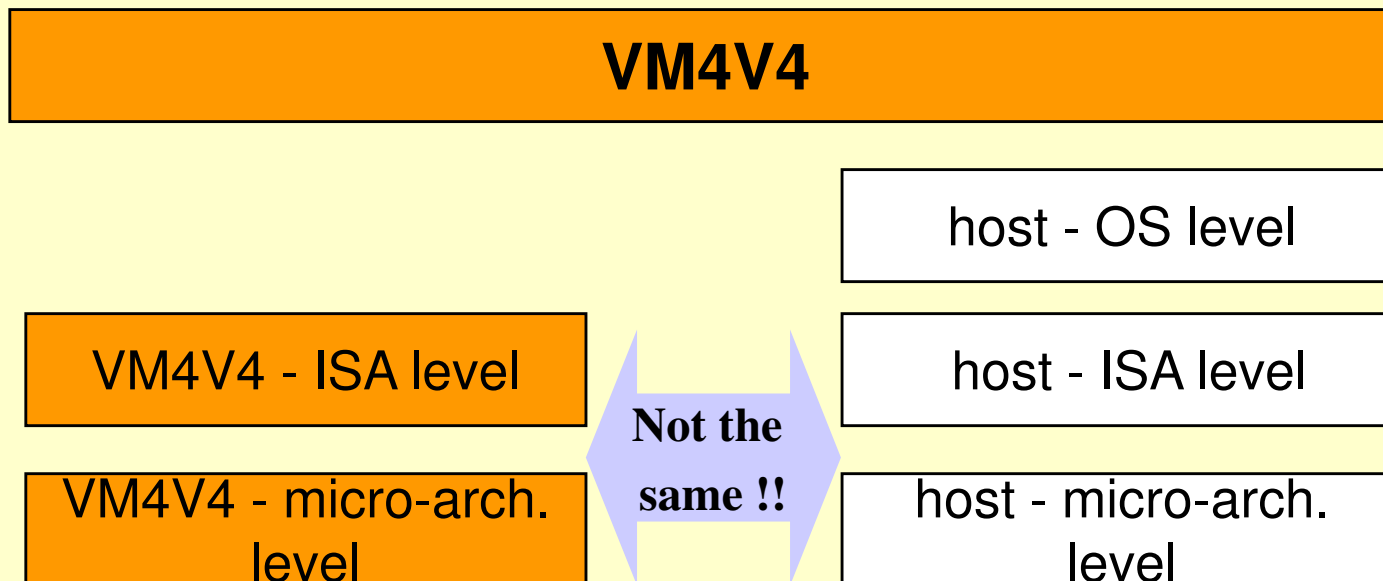
Interpretation and Java

- The Java virtual machine is one well known example of an interpreter.
- It doesn't actually execute Java directly, but it interprets the bytecode produced by the Java compiler.



VM4V4

- based upon the Java virtual machine - but *much* simpler
- VM4V4 is a virtual machine which is physically above level 3
- yet it reflects an imaginary machine with its *instruction set architecture* (level 2) and *micro-architecture level* (level 1).
- however - these imaginary levels are useful as examples.



V4 Virtual machine

The V4 Virtual Machine interface includes a menu bar with 'File' and 'Font'. The top section contains several registers and pointers:

- Local Variables: 04
- Local Variables Pointer: 00
- Stack Pointer: 04
- Program Counter: 62
- Instruction Register: 0000
- Translation: NoOp

An 'Execute Instruction' button is located below the registers.

The middle section displays three memory views, each with a vertical scrollbar:

- Thread Memory:** A table with 7 rows. The first four rows (00-03) are highlighted in green. Row 04 is the current instruction, marked with a play icon. Rows 05-06 are in blue.
- Method Memory:** A table with 7 rows (40-46) in blue.
- Heap Memory:** A table with 7 rows (80-86) in blue.

The bottom section contains file controls:

- Input filename: lssprog.txt
- Output filename: (empty)
- Browse... buttons for both input and output filenames.
- Load Memory button
- Save Memory button

Address	Value	Low Byte	High Byte
00	00000000	00	0
01	00110111	37	55
02	00000000	00	0
03	00000000	00	0
04	00000000	00	0
05	00000001	01	1
06	00000000	00	0

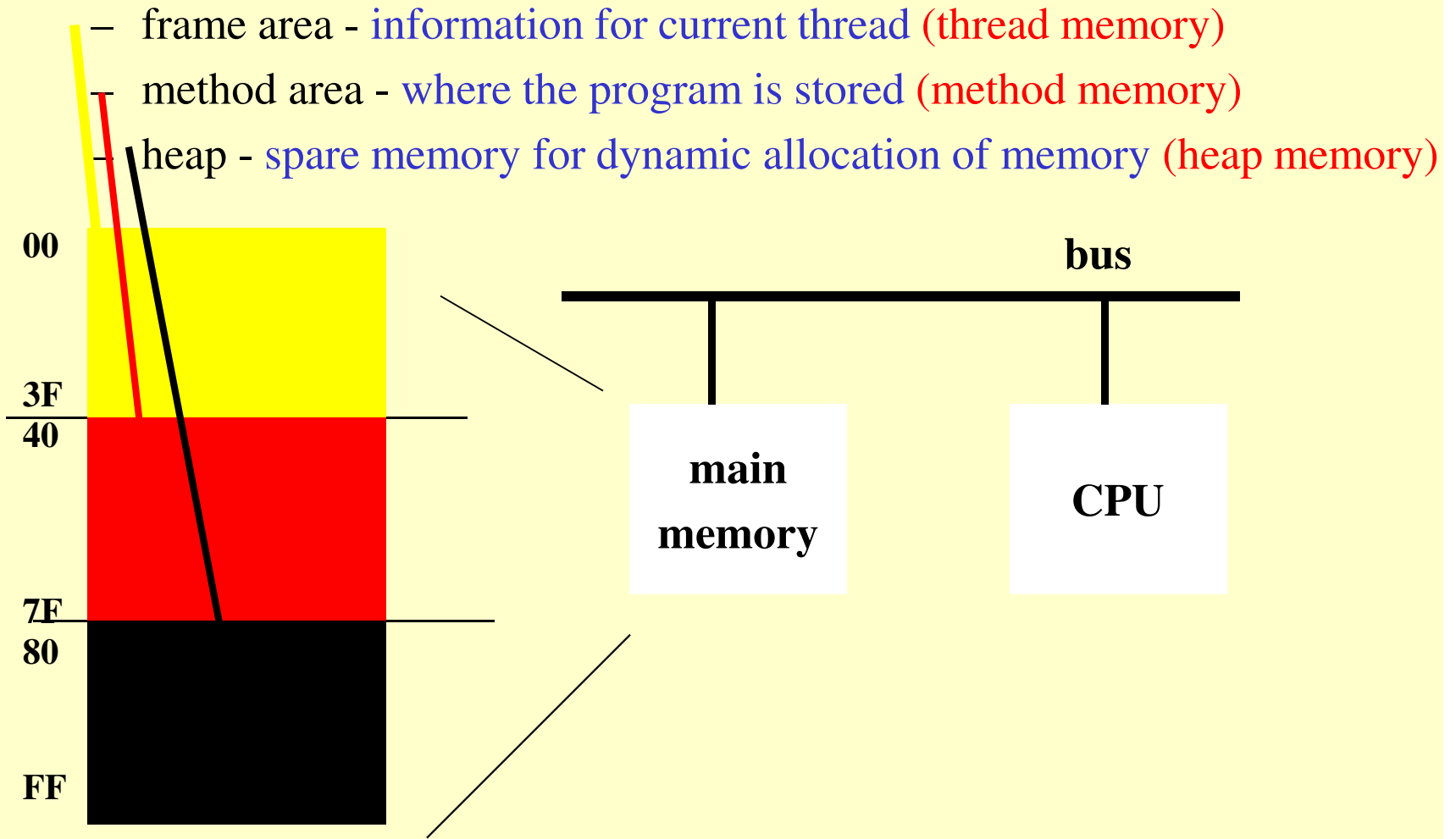
Address	Value	Low Byte	High Byte
40	00010010	12	18
41	00000101	05	5
42	00110110	36	54
43	00000000	00	0
44	00010010	12	18
45	00000000	00	0
46	00110110	36	54

Address	Value	Low Byte	High Byte
80	00000000	00	0
81	00000000	00	0
82	00000000	00	0
83	00000000	00	0
84	00000000	00	0
85	00000000	00	0
86	00000000	00	0

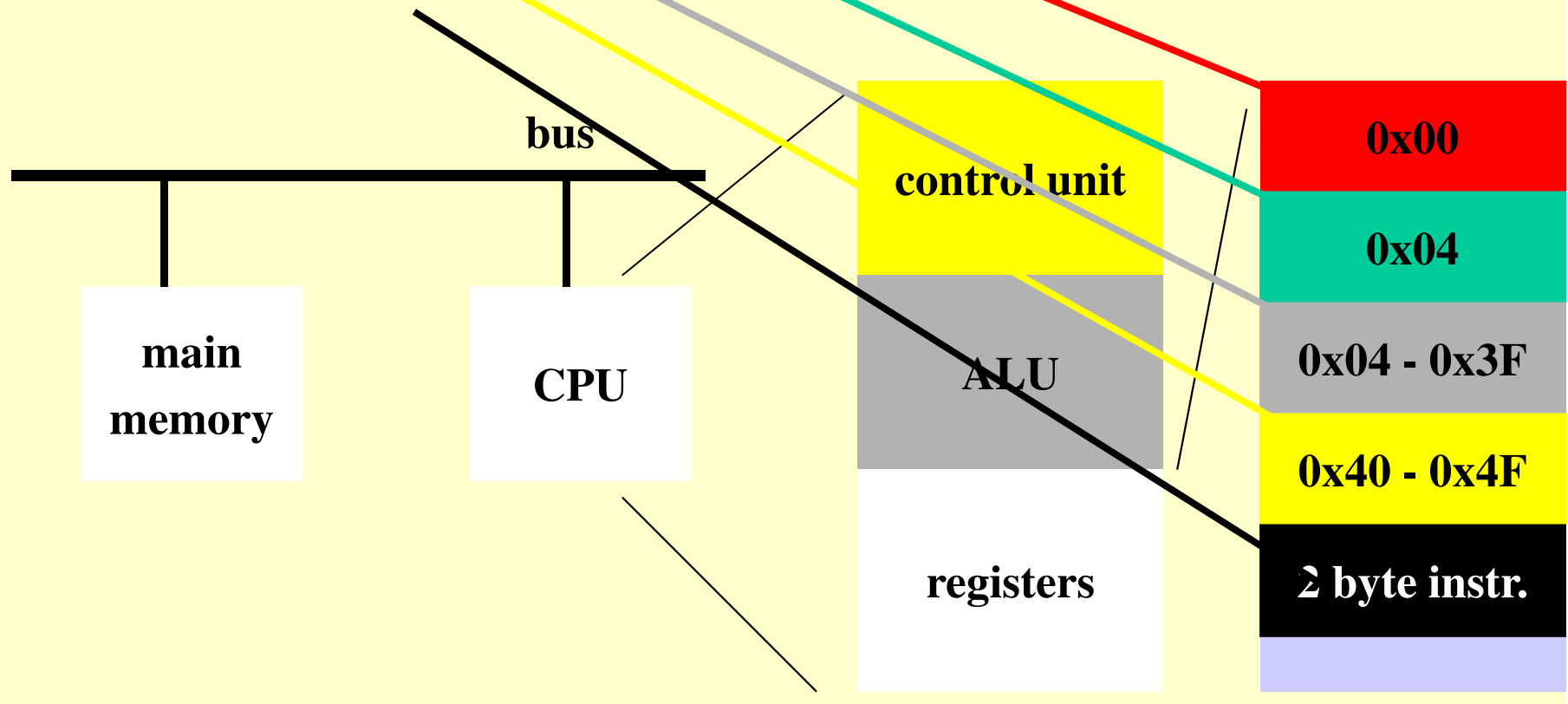
VM4V4 - micro-architectural level

- memory
 - frame area - information for current thread (thread memory)
 - method area - where the program is stored (method memory)
 - heap - spare memory for dynamic allocation of memory (heap memory)
- registers
 - local variables pointer - points to start of current frame, fixed at 0x00
 - number of local variables - how much memory is reserved for local vars.
 - Stack pointer - points to top of stack
 - program counter - points to the *next* instruction
 - instruction register - holds current instruction
 - (a translation of this instruction is provided)

- memory



- registers
 - local variables pointer - points to start of current frame, fixed at 0x00
 - number of local variables - how much memory is reserved for local vars
 - Stack pointer - points to top of stack
 - program counter - points to the *next* instruction
 - instruction register - holds current instruction



VM4V4 - instruction set architecture (isa) level

- Instructions
 - 2 byte instructions - 1 for opcode, 1 for operand
 - integers - 2's comp - 1 byte
 - logical operators - AND, OR
 - arithmetic - add, sub, multiple
 - operations are stack-based
 - branching - conditional & unconditional

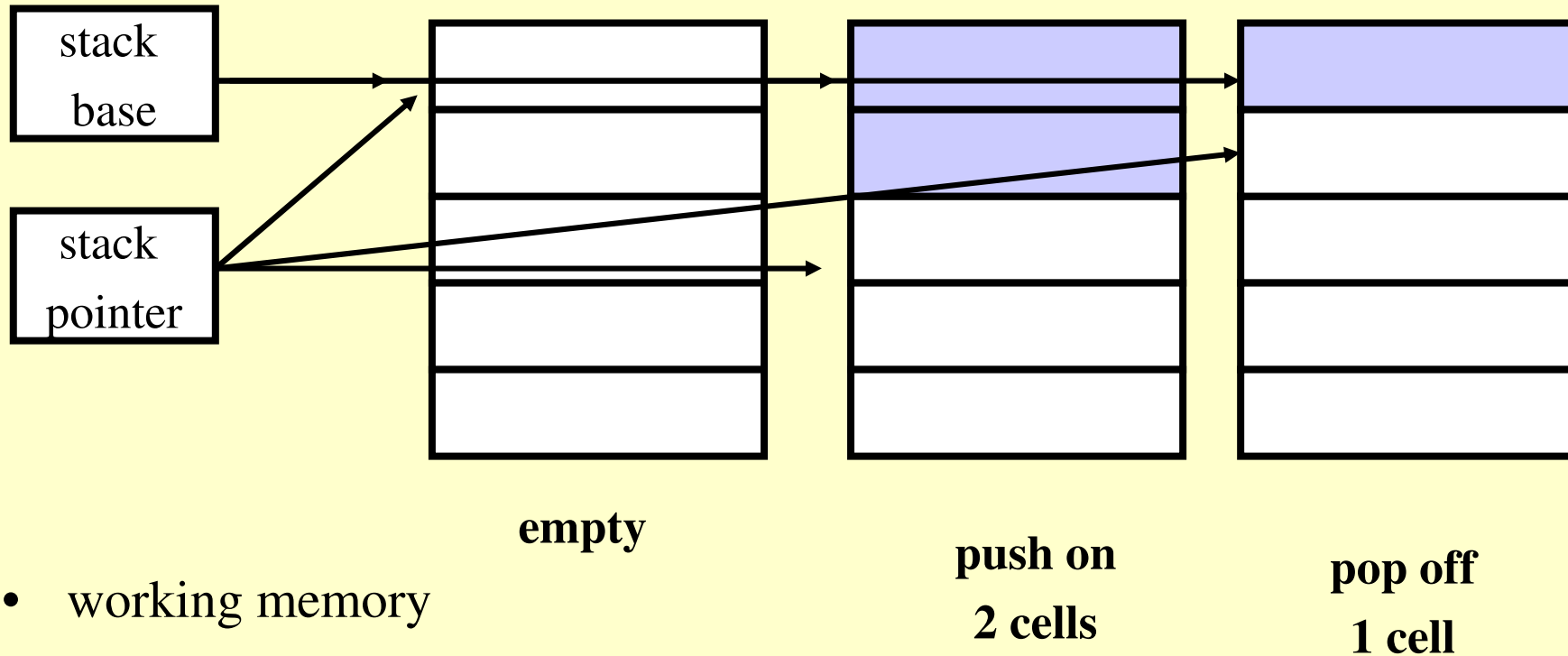
opcode	operand
--------	---------

iadd	
------	--

60xx	
------	--

- pops 2 numbers from stack
- adds them
- pushes answer back onto stack

what is a stack ?



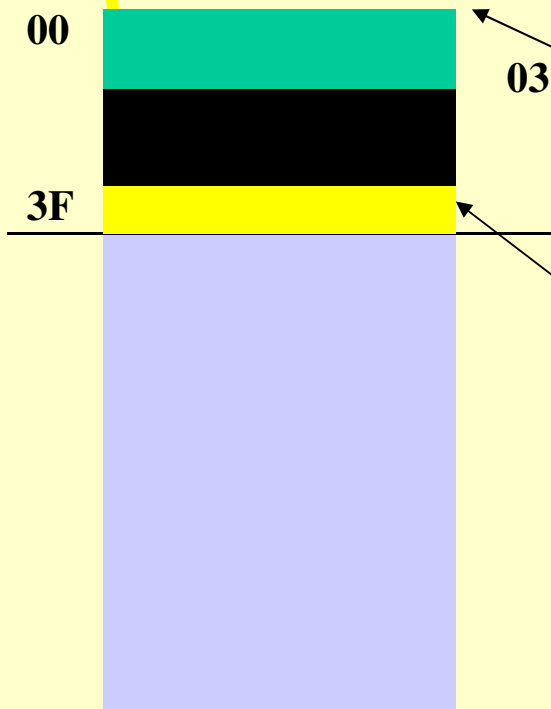
- working memory
- intermediate results
- storage of “local” variables
- dynamic - changing
- SB often fixed

where is the VM4V4 stack ?

VM4V4 - micro-architectural level

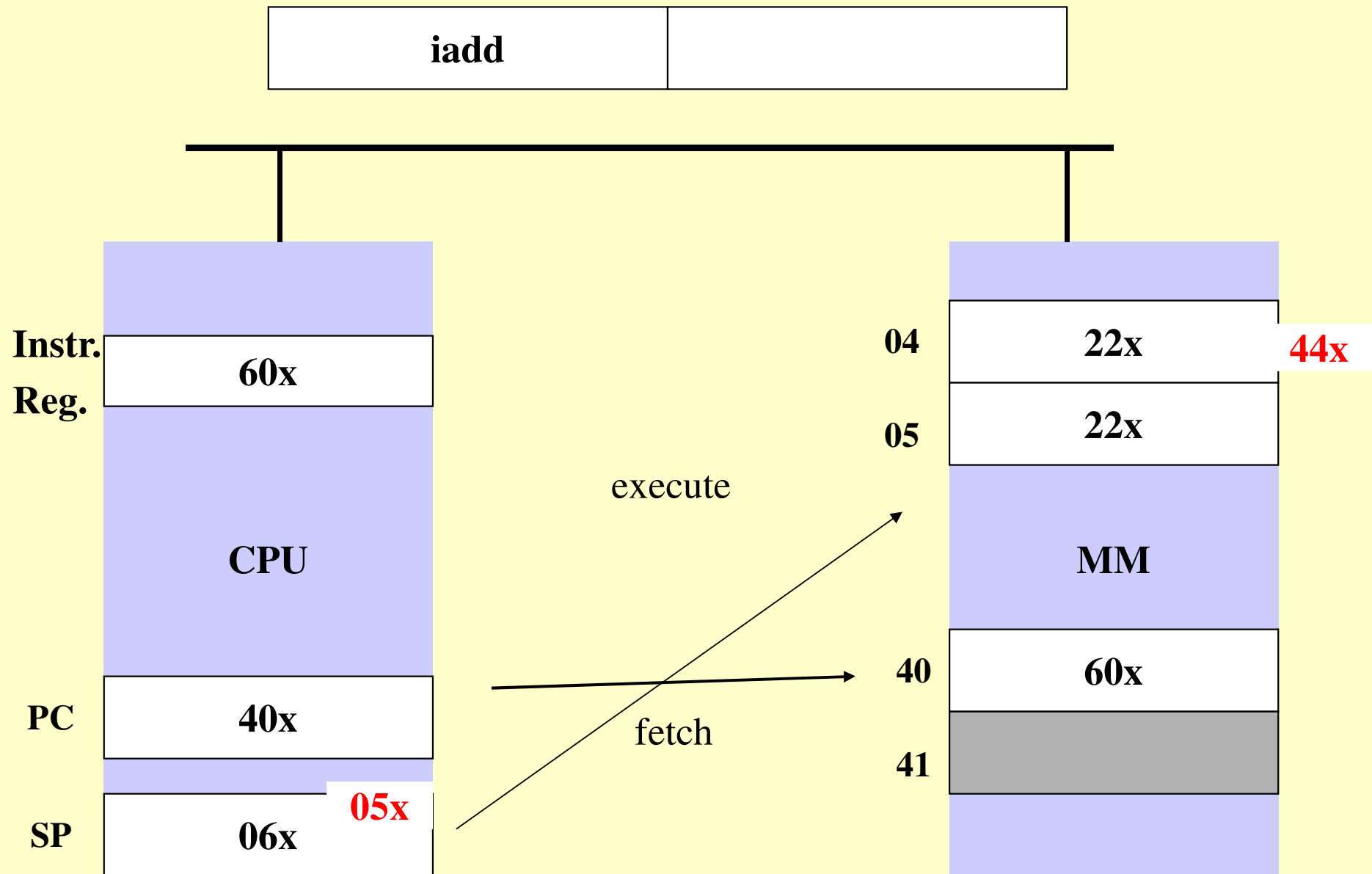
- memory

- frame area - information for current thread
- method area - where the program is stored
- heap - spare memory for dynamic allocation of memory



- registers

- local variables pointer - points to start of current frame, fixed at 0x00
- number of local variables - how much memory is reserved for local vars.
- Stack pointer - points to top of stack
- program counter - points to the *next* instruction
- instruction register - holds current instruction



End of lecture