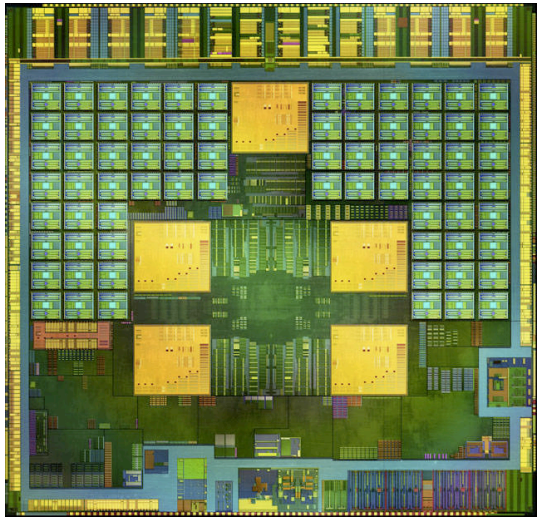
Introduction aux Systèmes d'Exploitation

Part II: Assembly Language (Units 5-8)

Unit 5: From source code to execution

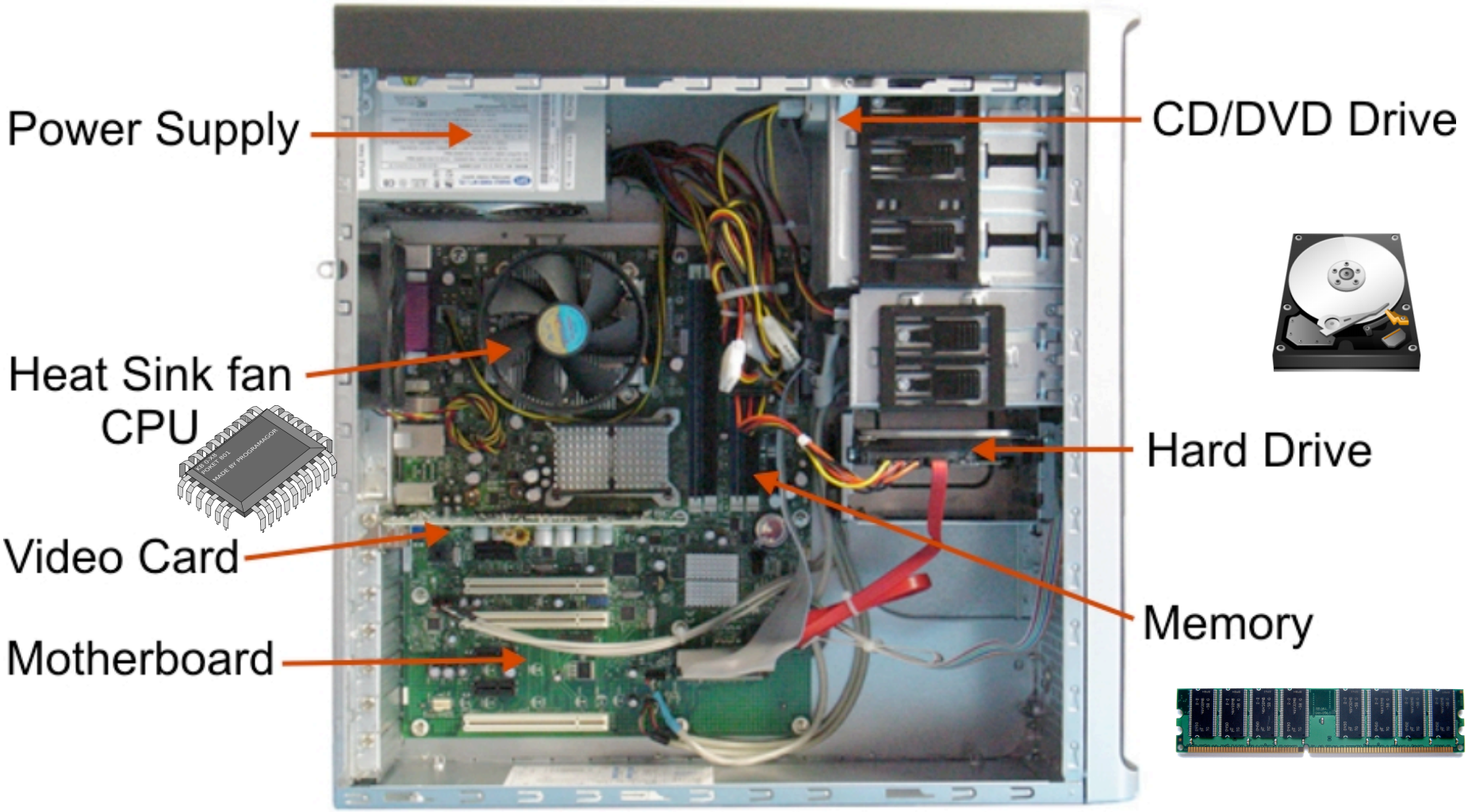
François Taïani



Unit's Outline

- What's in a computer?
- From a Java program to execution
  - ➔ Bytecode
  - ➔ The JVM
  - ➔ Machine code
  - ➔ Assembly
- What's in a processor?

What's in a Computer



SATA Connector (x4)

BIOS Flash Chip in PLCC Socket

Southbridge (with heatsink)

Floppy Drive Connector

IDE Connector (x2)

CMOS Backup Battery

24-pin ATX Power Connector

Integrated graphics processor (with heatsink)

Super IO Chip

PCI Slot (×3)

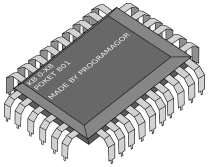
DIMM Memory Slots (×4)

CPU Fan & Heatsink Mount

Integrated audio codec chip

Integrated Gigabit Ethernet chip

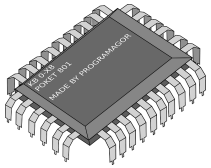
CPU Socket (Socket 939)

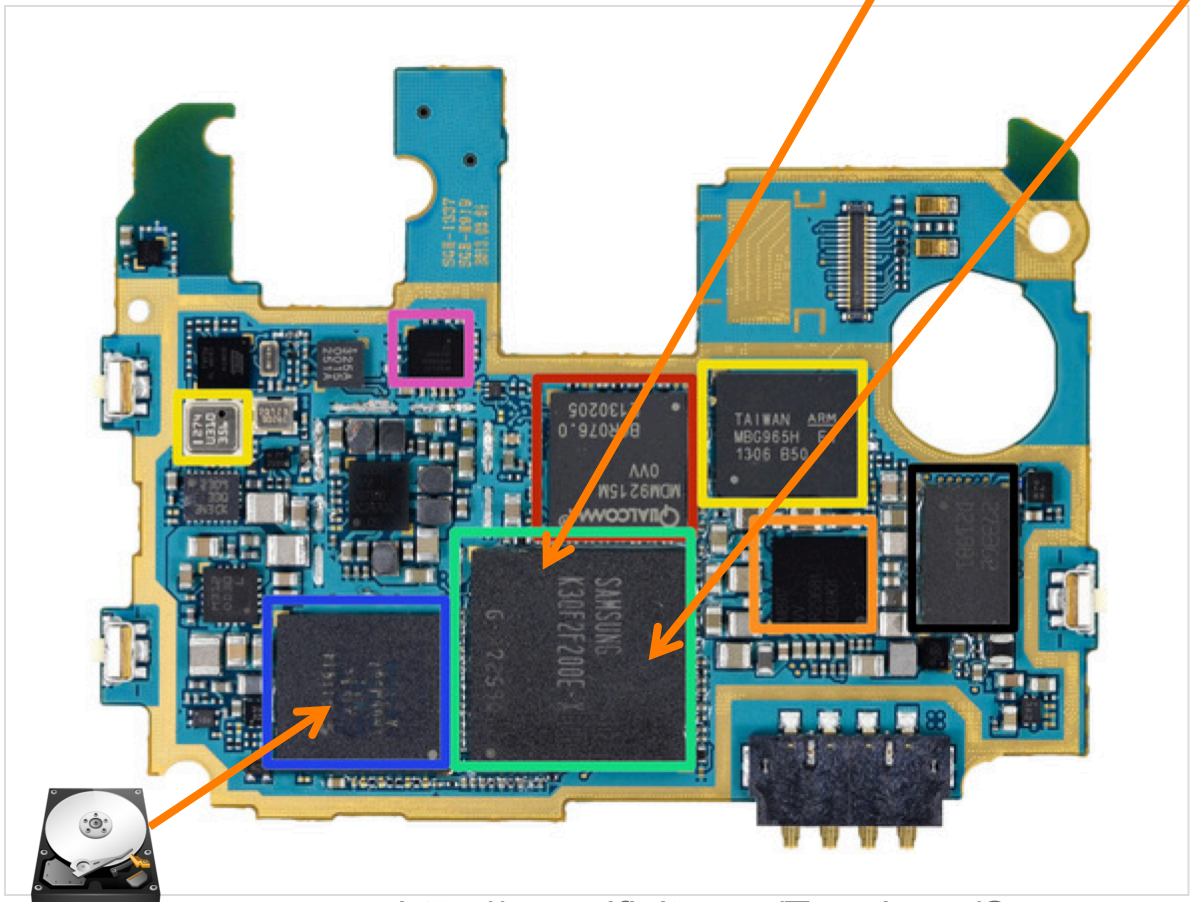


PCI Express Slot

Connectors For Integrated Peripherals

PS/2 Keyboard and Mouse, Serial Port, Parallel Port, VGA, Firewire/IEEE 1394a, USB (×4), Ethernet, Audio (×6)





Step 13

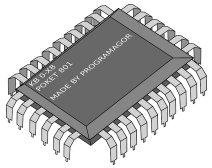
Edit 3

- Qualcomm MDM9215M 4G GSM/UMTS/LTE modem
- Qualcomm PM8917 power management
- ARM Holdings MBG965H (right)
- Samsung K3QF2F200E 2 GB LPDDR3 RAM (we suspect the Snapdragon 600 APQ8064T 1.9 GHz Quad-Core CPU lurks below)
- Toshiba THGBM5G7A4JBA4W 16 GB eMMC (eMMC integrates a NAND flash memory and a controller chip in a single package)
- Qualcomm WCD9310 audio codec
- Broadcom BCM4335 Single-Chip 5G Wi-Fi MAC/Baseband/Radio (thanks to a tip from a user, and our friends at chipworks, we believe there's something underneath this chip)
- Bosch Sensortec BMP180 barometric pressure sensor (left) - Samsung outputs the height in meters as second parameter.

source: http://www.ifixit.com/Teardown/Samsung+Galaxy+S4+Teardown/13947

Key Elements

- A Central Processing Unit (CPU)



- Some volatile memory

  - ➔ xxRAM : random access memory (e.g. SDRAM)
- Some permanent storage

  - ➔ Hard disk drive (hdd)
  - ➔ Solid State Disk (SDD, not a disk!)
- Specialized processors (GPU)
- Additional I/O devices
  - ➔ Screen, keyboard, GSM, GPS, accelerator
- All connected through different types of bus

From Program to Execution

- How does code interact with the machine?

```
public class HelloWorld {

  static public void main(String[] args) {
    System.out.println("Hello world");
  } // EndMain

} // EndClass HelloWorld
```

?

Hello world

The programme

```
public class HelloWorld {

   static public void main(String[] args) {
     System.out.println("Hello world");
   } // EndMain

} // EndClass HelloWorld
```

- Stored as a file on permanent storage
  - ➔ file = a sequence of bytes (byte: sequence of 8 bits)
  - ➔ permanent = storage retains data without power
- Characters encoded as sequence of bits
  - ➔ Traditional encoding : ASCII (7 bits), iso-latin-1 (8 bits)
  - ➔ More recent : UTF-8 (variable length)

```
localhost ftaiani [JAVA_EXAMPLE] $ xxd -b HelloWorld.java
0000000: 01110000 01110101 01100010 01101100 01101001 01100011  public
0000006: 00100000 01100011 01101100 01100001 01110011 01110011   class
000000c: 00100000 01001000 01100101 01101100 01101100 01101111   Hello
0000012: 01010111 01101111 01110010 01101100 01100100 00100000  World
0000018: 01111011 00001010 00001010 00100000 00100000 01110011  {..  s
000001e: 01110100 01100001 01110100 01101001 01100011 00100000  tatic
0000024: 01110000 01110101 01100010 01101100 01101001 01100011  public
000002a: 00100000 01110110 01101111 01101001 01100100 00100000   void
0000030: 01101101 01100001 01101001 01101110 00101000 01010011  main(S
0000036: 01110100 01110010 01101001 01101110 01100111 01011011  tring[
000003c: 01011101 00100000 01100001 01110010 01100111 01110011  ] args
0000042: 00101001 00100000 01111011 00001010 00100000 00100000  ) {.
0000048: 00100000 00100000 01010011 01111001 01110011 01110100    Syst
000004e: 01100101 01101101 00101110 01101111 01110101 01110100  em.out
0000054: 00101110 01110000 01110010 01101001 01101110 01110100  .print
000005a: 01101100 01101110 00101000 00100010 01001000 01100101  ln("He
0000060: 01101100 01101100 01101111 00100000 01110111 01101111  llo wo
0000066: 01110010 01101100 01100100 00100010 00101001 00111011  rld");
000006c: 00001010 00100000 00100000 01111101 00100000 00101111  .  } /
0000072: 00101111 00100000 01000101 01101110 01100100 01001101  / EndM
0000078: 01100001 01101001 01101110 00001010 00001010 01111101  ain..}
000007e: 00100000 00101111 00101111 00100000 01000101 01101110   // En
0000084: 01100100 01000011 01101100 01100001 01110011 01110011  dClass
000008a: 00100000 01001000 01100101 01101100 01101100 01101111   Hello
0000090: 01010111 01101111 01110010 01101100 01100100 00001010  World.
```

Binary Encoding of Characters

- 01110000
  - ➔ the number 112 in base 2 ($112 = 2^7+2^6+2^5$)
  - ➔ the ASCII code for the character ‘p’ (lowercase P)
- ASCII contains some non-printable characters
  - ➔ shown as dots ‘.’ on printout
  - ➔ ‘00001010’ = 10 in base 2 = Line Feed (LF, or ‘\n’)
- Printing numbers in binary format uses space
  - ➔ Compacted format : hexadecimal = base 16
  - ➔ Digit : 0, .., 9, A, B, C, D, F
  - ➔ $0111\ 0000_{bits} = 70_{hex} = 7\times16 = 112$ → ‘p’

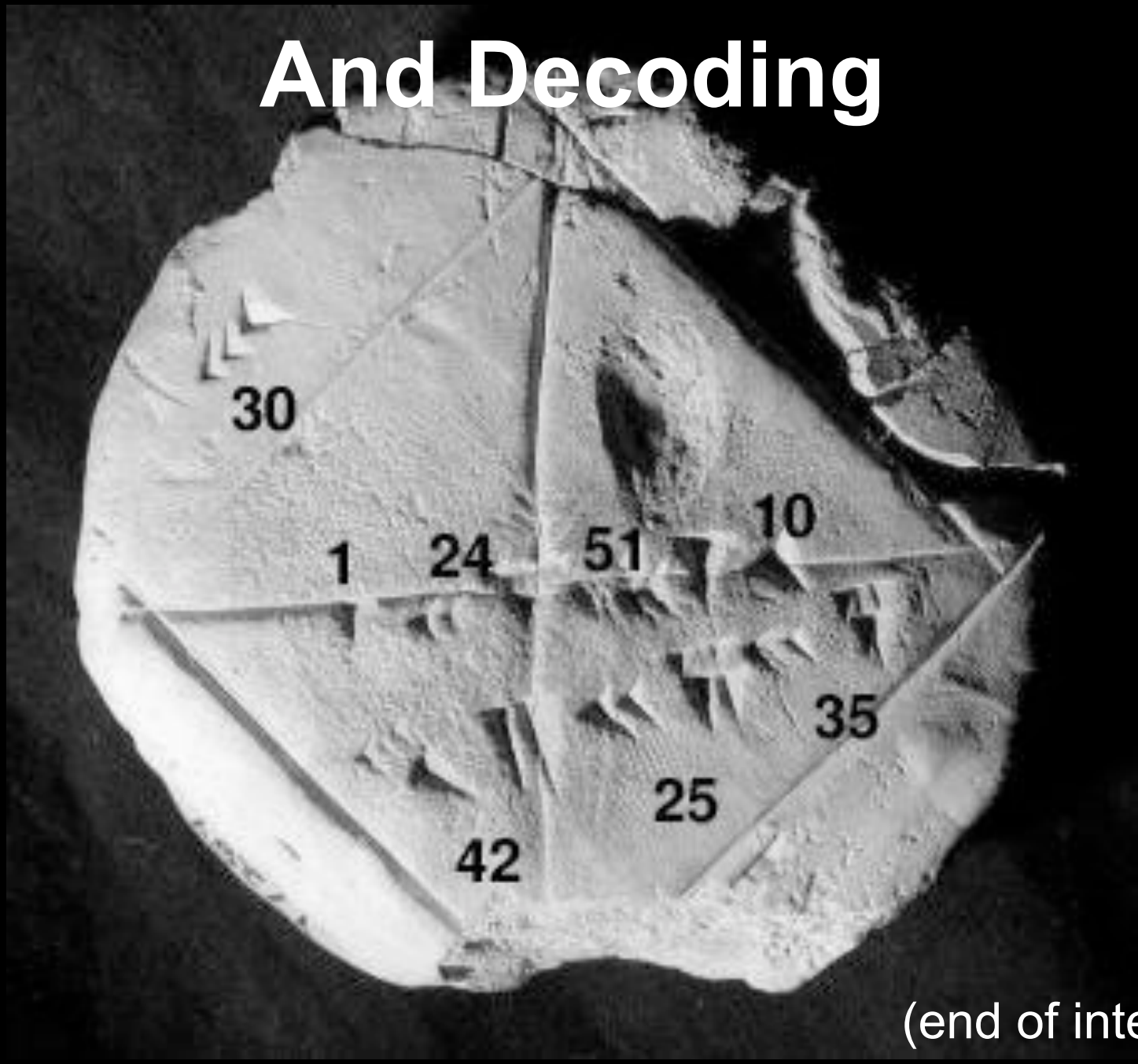
```
localhost ftaiani [JAVA_EXAMPLE] $ xxd -c6 HelloWorld.java
0000000: 7075 626c 6963  public
0000006: 2063 6c61 7373   class
000000c: 2048 656c 6c6f   Hello
0000012: 576f 726c 6420  World
0000018: 7b0a 0a20 2073  {..  s
000001e: 7461 7469 6320  tatic
0000024: 7075 626c 6963  public
000002a: 2076 6f69 6420   void
0000030: 6d61 696e 2853  main(S
0000036: 7472 696e 675b  tring[
000003c: 5d20 6172 6773  ] args
0000042: 2920 7b0a 2020  ) {.
0000048: 2020 5379 7374    Syst
000004e: 656d 2e6f 7574  em.out
0000054: 2e70 7269 6e74  .print
000005a: 6c6e 2822 4865  ln("He
0000060: 6c6c 6f20 776f  llo wo
0000066: 726c 6422 293b  rld");
000006c: 0a20 207d 202f  .  } /
0000072: 2f20 456e 644d  / EndM
0000078: 6169 6e0a 0a7d  ain..}
000007e: 202f 2f20 456e   // En
0000084: 6443 6c61 7373  dClass
000008a: 2048 656c 6c6f   Hello
0000090: 576f 726c 640a  World.
```

Why Bother about Binary/Hex?

- Because bits do not always encode text
  - ➔ So called « binary files »
  - ➔ Covers many types : executable, images, video, compressed data, proprietary formats, network packets ...
- Printing the files as text yields gibberish
- To understand these files
  - ➔ You need to look at the binary/hexadecimal values
  - ➔ And know how to interpret them
  - ➔ Thankfully many tools available

Interlude: On Encoding

And Decoding



Babylonian clay tablet YBC 7289 (~1800-1600 BC)

(end of interlude)

From Program to Execution

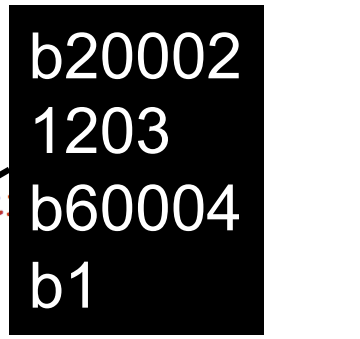
- Source code → no meaning for computer
- Transformation step : compilation
  - ➔ javac HelloWorld.java
- Produces a new file : HelloWorld.class
  - ➔ Not a text file!

```
localhost ftaiani [JAVA_EXAMPLE] $ xxd -c 20 HelloWorld.class
0000000: cafe babe 0000 0033 001d 0a00 0600 0f09 0010 0011  .......3..........
0000014: 0800 120a 0013 0014 0700 1507 0016 0100 063c 696e  .................<in
0000028: 6974 3e01 0003 2829 5601 0004 436f 6465 0100 0f4c  it>...()V...Code...L
000003c: 696e 654e 756d 6265 7254 6162 6c65 0100 046d 6169  ineNumberTable...mai
0000050: 6e01 0016 285b 4c6a 6176 612f 6c61 6e67 2f53 7472  n...([Ljava/lang/Str
0000064: 696e 673b 2956 0100 0a53 6f75 7263 6546 696c 6501  ing;)V...SourceFile.
0000078: 000f 4865 6c6c 6f57 6f72 6c64 2e6a 6176 610c 0007  ..HelloWorld.java...
000008c: 0008 0700 170c 0018 0019 0100 0b48 656c 6c6f 2077  .............Hello w
00000a0: 6f72 6c64 0700 1a0c 001b 001c 0100 0a48 656c 6c6f  orld...........Hello
00000b4: 576f 726c 6401 0010 6a61 7661 2f6c 616e 672f 4f62  World...java/lang/Ob
00000c8: 6a65 6374 0100 106a 6176 612f 6c61 6e67 2f53 7973  ject...java/lang/Sys
00000dc: 7465 6d01 0003 6f75 7401 0015 4c6a 6176 612f 696f  tem...out...Ljava/io
00000f0: 2f50 7269 6e74 5374 7265 616d 3b01 0013 6a61 7661  /PrintStream;...java
0000104: 2f69 6f2f 5072 696e 7453 7472 6561 6d01 0007 7072  /io/PrintStream...pr
0000118: 696e 746c 6e01 0015 284c 6a61 7661 2f6c 616e 672f  intln...(Ljava/lang/
000012c: 5374 7269 6e67 3b29 5600 2100 0500 0600 0000 0000  String;)V.!.........
0000140: 0200 0100 0700 0800 0100 0900 0000 1d00 0100 0100  ....................
0000154: 0000 052a b700 01b1 0000 0001 000a 0000 0006 0001  ...*................
0000168: 0000 0001 0009 000b 000c 0001 0009 0000 0025 0002  .................%..
000017c: 0001 0000 0009 b200 0212 03b6 0004 b100 0000 0100  ....................
0000190: 0a00 0000 0a00 0200 0000 0400 0800 0500 0100 0d00  ....................
00001a4: 0000 0200 0e                                      .....
```

A .class File

- Contains some (important) meta data
  - ➔ The class' name
  - ➔ The names of its interfaces, methods, ..., values of const
  - ➔ The code of its method (Bytecode)
- Disassembling bytecode: javap -verbose -c HelloWorld

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=1, args_size=1
       // Field java/lang/System.out:Ljava/io/PrintSt
       0: getstatic     #2
       // String Hello world
       3: ldc           #3
       // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       5: invokevirtual #4
       8: return
```



Assembly Code

- Disassembled bytecode
  - ➔ A particular example of **assembly code**
- A much more readable representation of binary code
  - ➔ Instead of « b60004 », « invokevirtual #4 »
  - ➔ « invokevirtual » is a **mnemonics**
- Special programs to manipulate assembly code
  - ➔ disassembler
  - ➔ assembler

Executing Bytecode

- A computer cannot interpret bytecode « alone »
- It needs a interpreter: a Java Virtual Machine
  - ➔ Loads the bytecode from permanent storage
  - ➔ Works with the rest of the computer to interpret it
- But where does the JVM come from?
  - ➔ It's a program like any other
  - ➔ The implementation of Oracle : coded in C & C++

Executing the JVM

- Like Java, C → no meaning, needs to be compiled
  - ➔ E.g. entry point of the openjdk JVM

```
#else /* JAVAW */
int
main(int argc, char ** argv)
{
    int margc;
    char** margv;
    const jboolean const_javaw = JNI_FALSE;

    margc = argc;
    margv = argv;
#endif /* JAVAW */

    return JLI_Launch(margc, margv,
                   sizeof(const_jargs) / sizeof(char *), const_jargs,
                   sizeof(const_appclasspath) / sizeof(char *), const_appclasspath,
                   FULL_VERSION,
                   DOT_VERSION,
                   (const_progname != NULL) ? const_progname : *margv,
                   (const_launcher != NULL) ? const_launcher : *margv,
                   (const_jargs != NULL) ? JNI_TRUE : JNI_FALSE,
                   const_cpwildcard, const_javaw, const_ergo_class);
}
```

Compiling the JVM

- You can do it at home!
  - ➔ Open source implementation: openjdk
- Compilation process a bit complex
  - ➔ Multiple files, multiple platforms, optimization
  - ➔ Automated thanks to building tools (here 'make')
- But at the heart of it : a C/C++ compiler
  - ➔ On Linux: gcc, on MacOS: gcc + possibly clang (?)
- Result: the java executable (+javac, javap, etc.)

```
debianfrancoist ftaiani [ftaiani] $ xxd -c 24 /usr/lib/jvm/java-6-openjdk/jre/bin/java | head -25
0000000: 7f45 4c46 0201 0100 0000 0000 0000 0000 0200 3e00 0100 0000  .ELF..............>.....
0000018: e01b 4000 0000 0000 4000 0000 0000 0000 f086 0000 0000 0000  ..@.....@.............
0000030: 0000 0000 4000 3800 0800 4000 1e00 1d00 0600 0000 0500 0000  ....@.8...@.............
0000048: 4000 0000 0000 0000 4000 4000 0000 0000 4000 4000 0000 0000  @.......@.@.....@.@.....
0000060: c001 0000 0000 0000 c001 0000 0000 0000 0800 0000 0000 0000  ........................
0000078: 0300 0000 0400 0000 0002 0000 0000 0000 0002 4000 0000 0000  ..................@.....
0000090: 0002 4000 0000 0000 1c00 0000 0000 0000 1c00 0000 0000 0000  ..@.....................
00000a8: 0100 0000 0000 0000 0100 0000 0500 0000 0000 0000 0000 0000  ........................
00000c0: 0000 4000 0000 0000 0000 4000 0000 0000 2c81 0000 0000 0000  ..@.......@.....,.......
00000d8: 2c81 0000 0000 0000 0000 2000 0000 0000 0100 0000 0600 0000  ,.........  .............
00000f0: 3081 0000 0000 0000 3081 6000 0000 0000 3081 6000 0000 0000  0.......0.`.....0.`.....
0000108: b004 0000 0000 0000 4805 0000 0000 0000 0000 2000 0000 0000  ........H.........  .....
0000120: 0200 0000 0600 0000 7881 0000 0000 0000 7881 6000 0000 0000  ........x.......x.`.....
0000138: 7881 6000 0000 0000 1002 0000 0000 0000 1002 0000 0000 0000  x.`.....................
0000150: 0800 0000 0000 0000 0400 0000 0400 0000 1c02 0000 0000 0000  ........................
0000168: 1c02 4000 0000 0000 1c02 4000 0000 0000 4400 0000 0000 0000  ..@.......@.....D.......
0000180: 4400 0000 0000 0000 0400 0000 0000 0000 50e5 7464 0400 0000  D...............P.td....
0000198: dc7a 0000 0000 0000 dc7a 4000 0000 0000 dc7a 4000 0000 0000  .z.......z@......z@.....
00001b0: 2c01 0000 0000 0000 2c01 0000 0000 0000 0400 0000 0000 0000  ,.......,...............
00001c8: 51e5 7464 0600 0000 0000 0000 0000 0000 0000 0000 0000 0000  Q.td....................
00001e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000  ........................
00001f8: 0800 0000 0000 0000 2f6c 6962 3634 2f6c 642d 6c69 6e75 782d  ......../lib64/ld-linux-
0000210: 7838 362d 3634 2e73 6f2e 3200 0400 0000 1000 0000 0100 0000  x86-64.so.2.............
0000228: 474e 5500 0000 0000 0200 0000 0600 0000 1200 0000 0400 0000  GNU.....................
0000240: 1400 0000 0300 0000 474e 5500 7e26 2814 6154 e9c0 3a35 a12d  ........GNU.~&(.aT..:5.-
```

The Java Executable

- An « Object File » : like a .class file
  - ➔ Contains metadata (libraries, external symbols, ...)
  - ➔ Values of constants
  - ➔ And code
- Contrarily to class file
  - ➔ Code directly executable: **machine code**

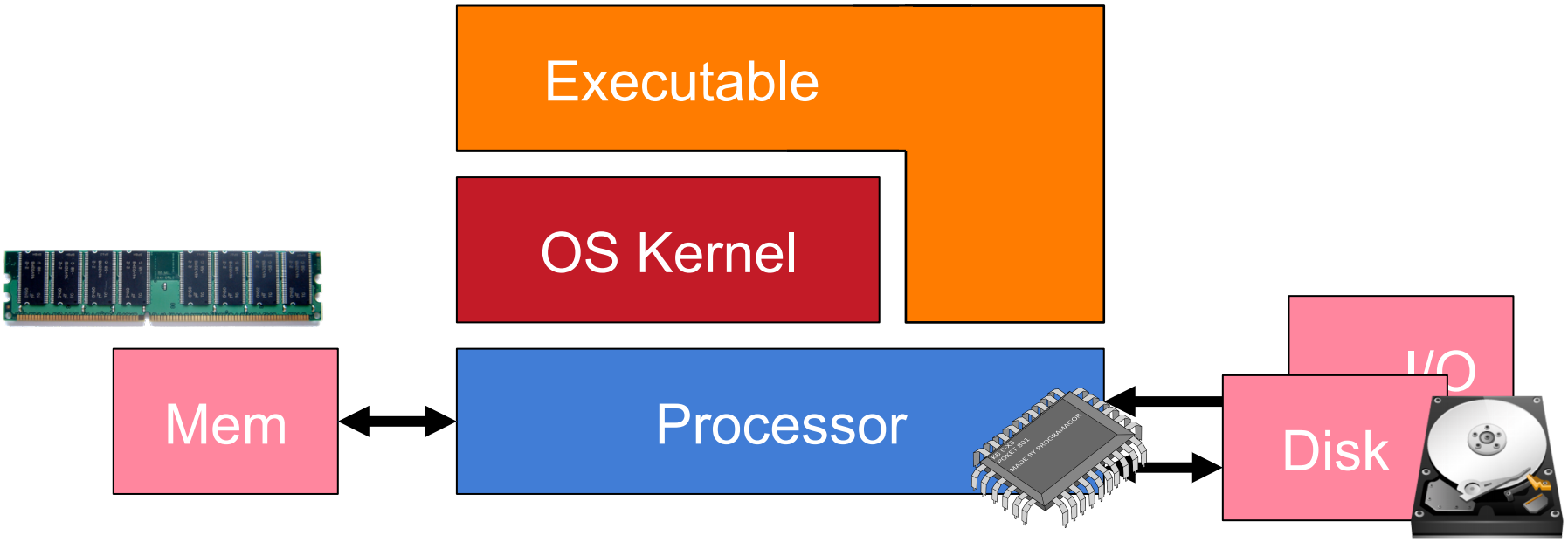
```
0000000000004017b0 <.init>:
            48 83 ec 08
            e8 53 04 00 00
            e8 e2 04 00 00
            e8 9d 51 00 00
            48 83 c4 08
            c3
```

Machine Code

- ‘48 83 ec 08’
  - ➔ Machine code for an intel processor x86 64 bits
  - ➔ Meaning: « subtract 8 from register rsp » (stack pointer)
  - ➔ Corresponding assembly code: sub rsp,0x8
- Note: each processor type has its own machine code
  - ➔ Own set of instructions
    (aka "ISA" or "Instruction Set Architecture")
  - ➔ Own mnemonics
  - ➔ Own assembly language
  - ➔ All of the above would be different on a ARM processor

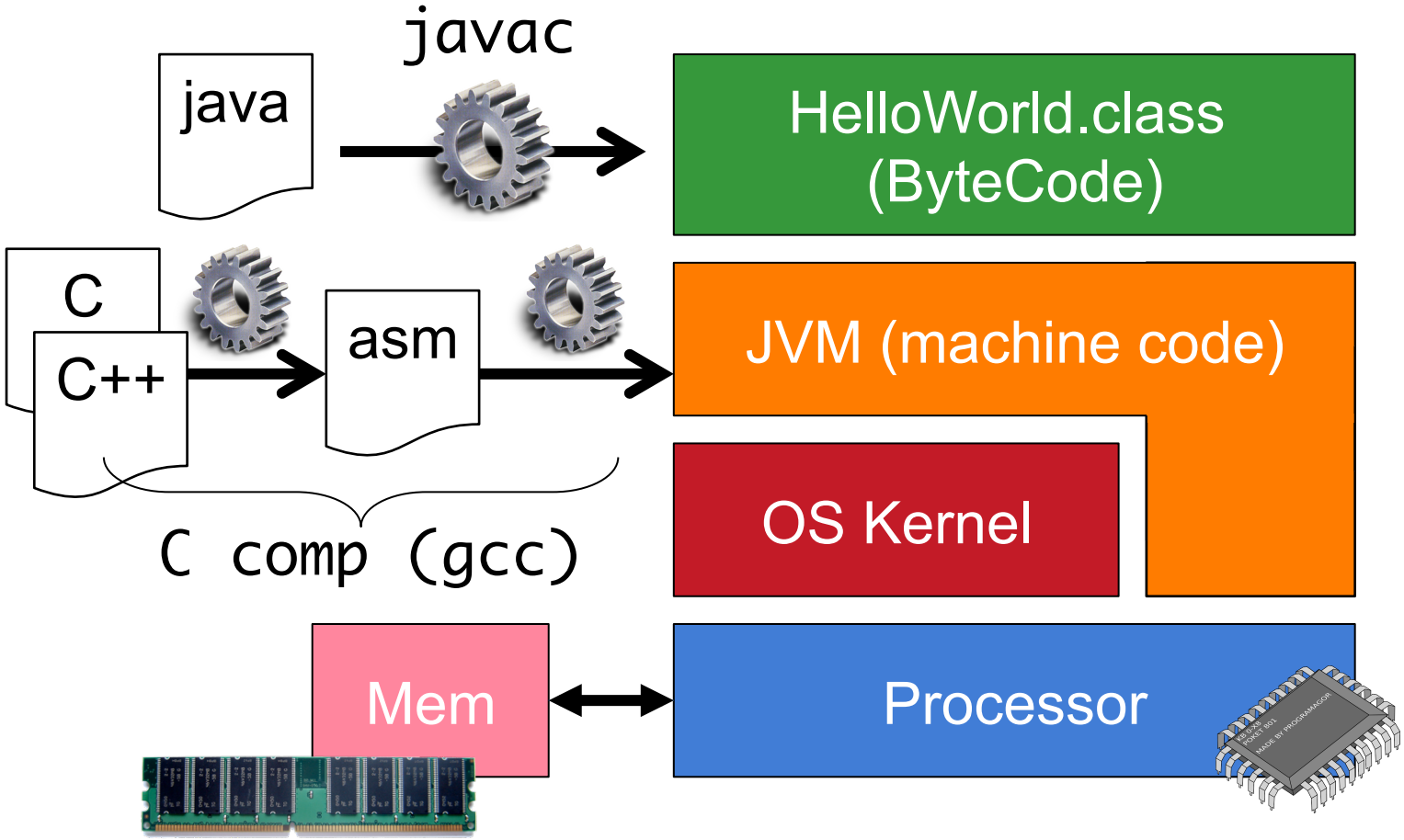
Machine Code

- Machine code "directly" executable: a bit of a short-cut
  - ➔ Some machine code can execute on bare-metal (embedded systems, boot loaders, ...)
  - ➔ Most commonly needs an OS

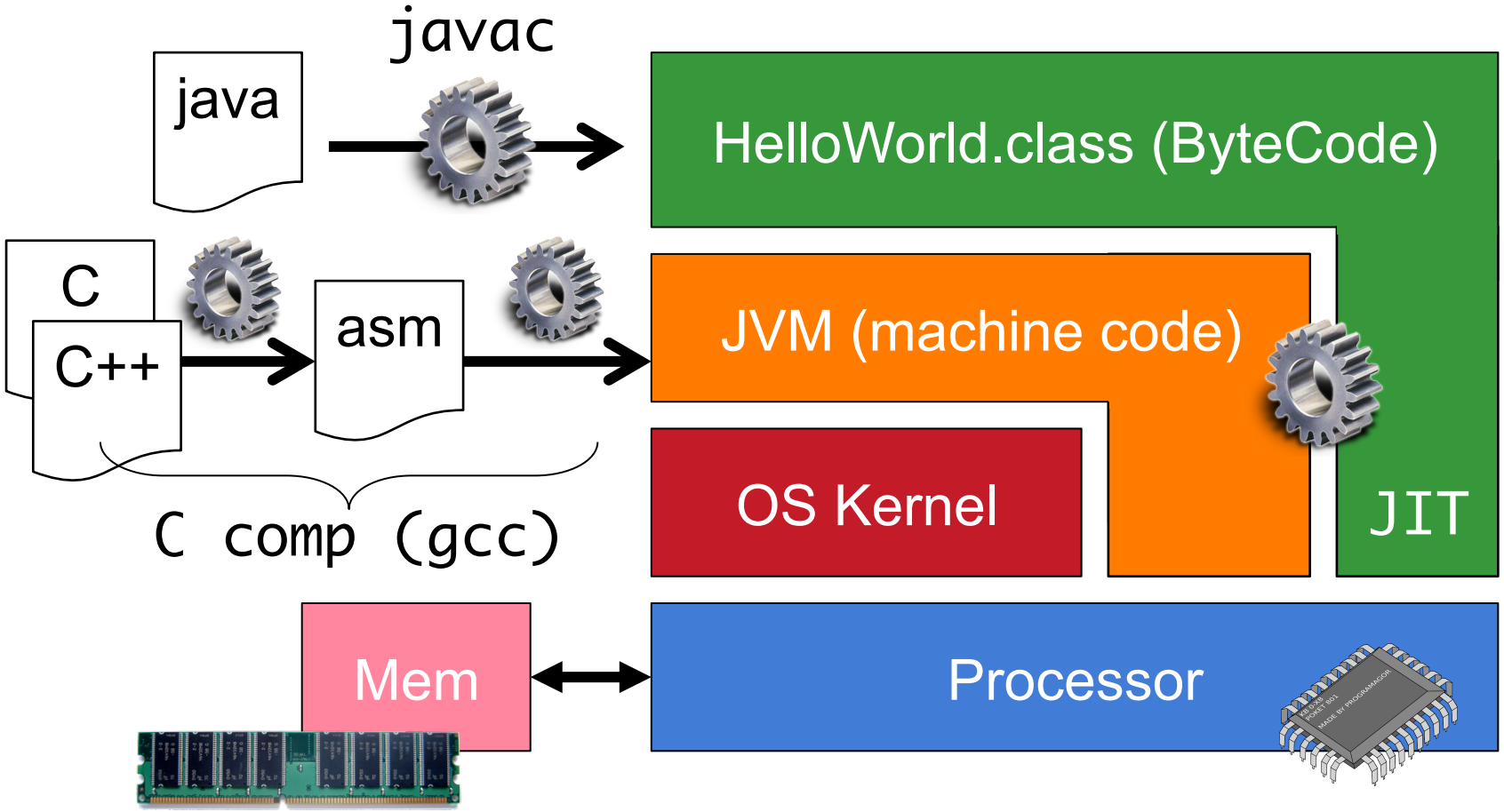


  - ➔ Most of the time : on processor, but OS for special ops

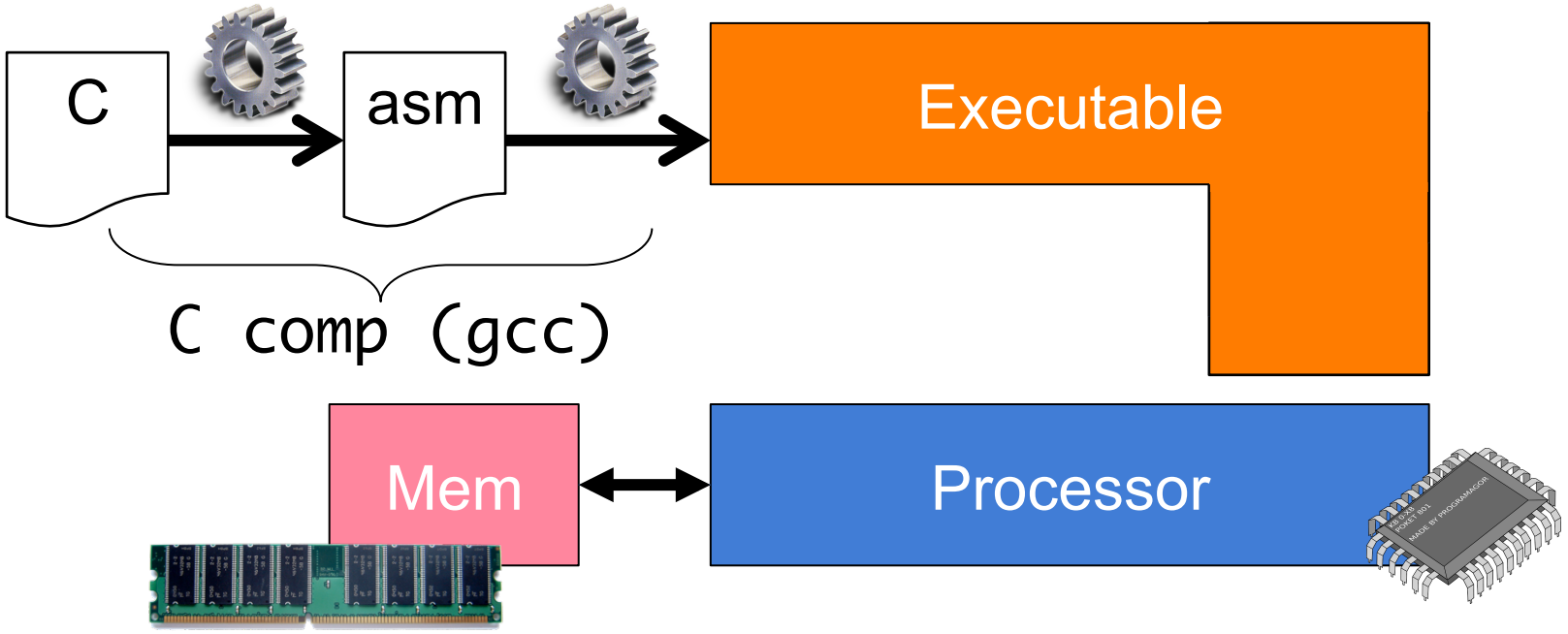
Summing up from Java to ISA



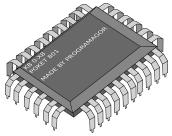
Summing up from Java to ISA



This Module's Focus



Key Characters We'll Revisit

- The **CPU** 
  - ➔ Where code and data meet for a program's execution
- The **Assembly Language**

```
sub rsp,0x8
```

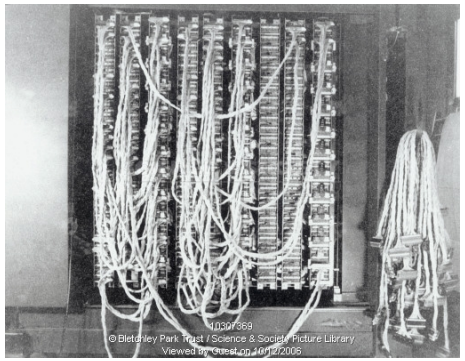
  - ➔ A human readable representation of machine code
- The **C Language**

```
int main(..)
```

  - ➔ The most popular system-level programming language
  - ➔ JVM, Linux kernel, and many libraries written in C (& C++)
- **Memory** 
  - ➔ The stack, the heap, and how they are used
- **Libraries**
- (The **Shell** `$ ls -a *`)

The CPU

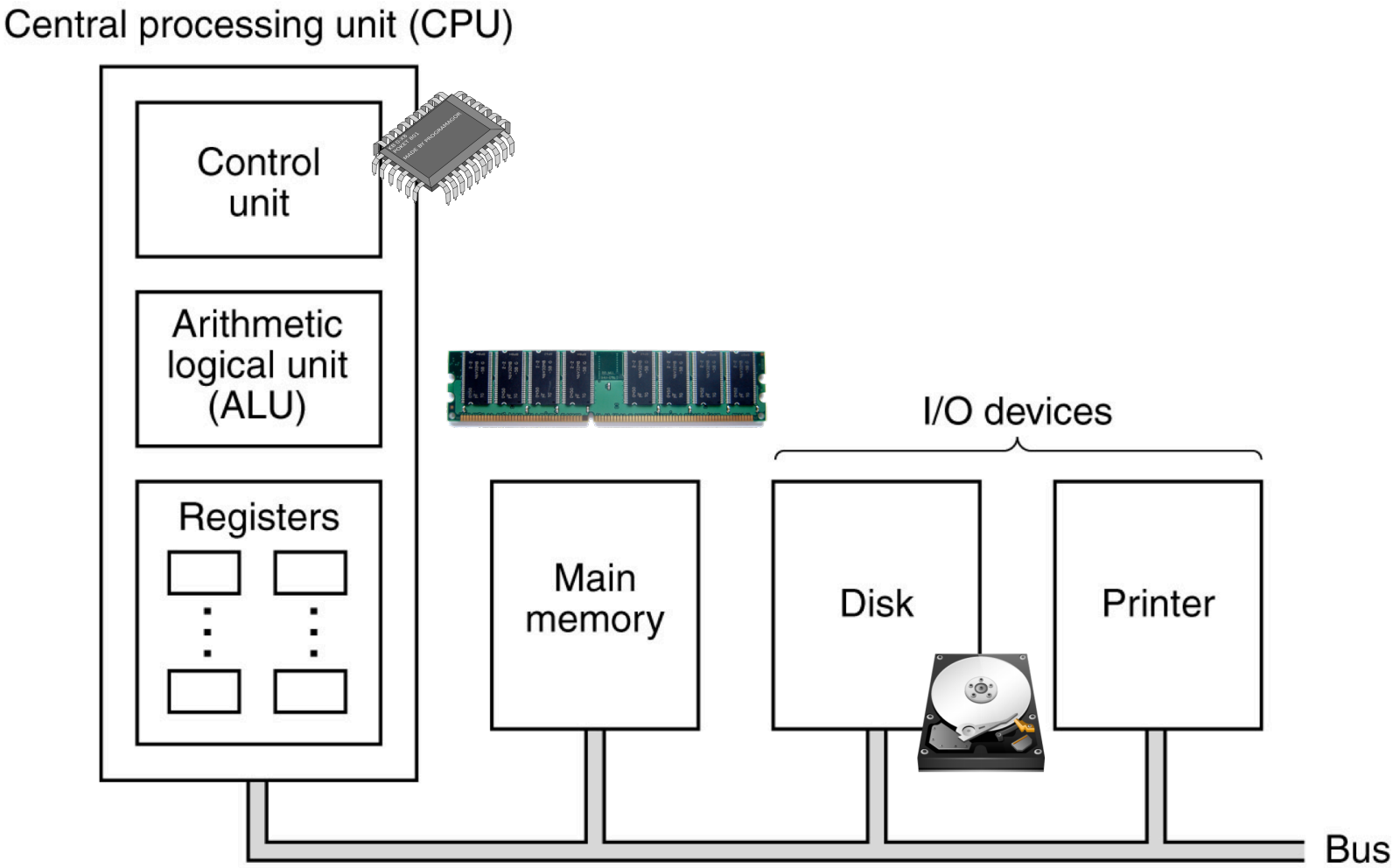
- Role
  - ➔ **Executes** machine code on data
- Machine **code & data**
  - ➔ both stored in main (volatile) memory
  - ➔ Was not always so
  - ➔ How do they get there? → see part on loaders
- What's in a CPU
  - ➔ **Control unit**: run the execution loop
  - ➔ Arithmetic and Logic Unit (**ALU**): executes instructions
  - ➔ **Registers**: limited number of very fast storage units



On Registers

- Small number
  - ➔ 16 on a x86-64 processor (same on a ARMv8)
- These registers have a fixed size in bits
  - ➔ Current computers : 64 bit
  - ➔ Early 8086 (first member of x86 family) : 16 bit
  - ➔ Embedded systems (and 80's microcomputers): 8 bit
- Size in bits
  - ➔ Linked to maximum directly addressable memory
  - ➔ E.g. 2^{16} = 64 kBytes
  - ➔ (but often extended,
    e.g. in 8086 extended to 2^{20} = 1 MBytes with segments)

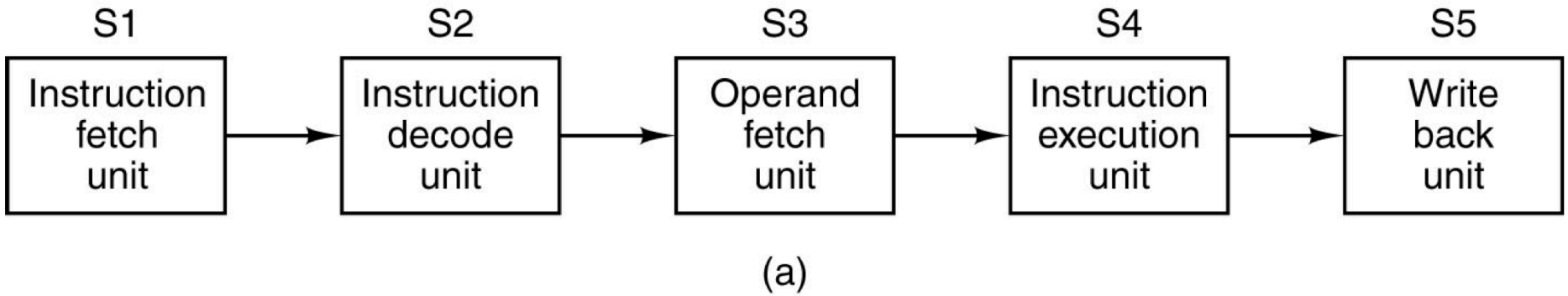
Typical (mono-core) CPU



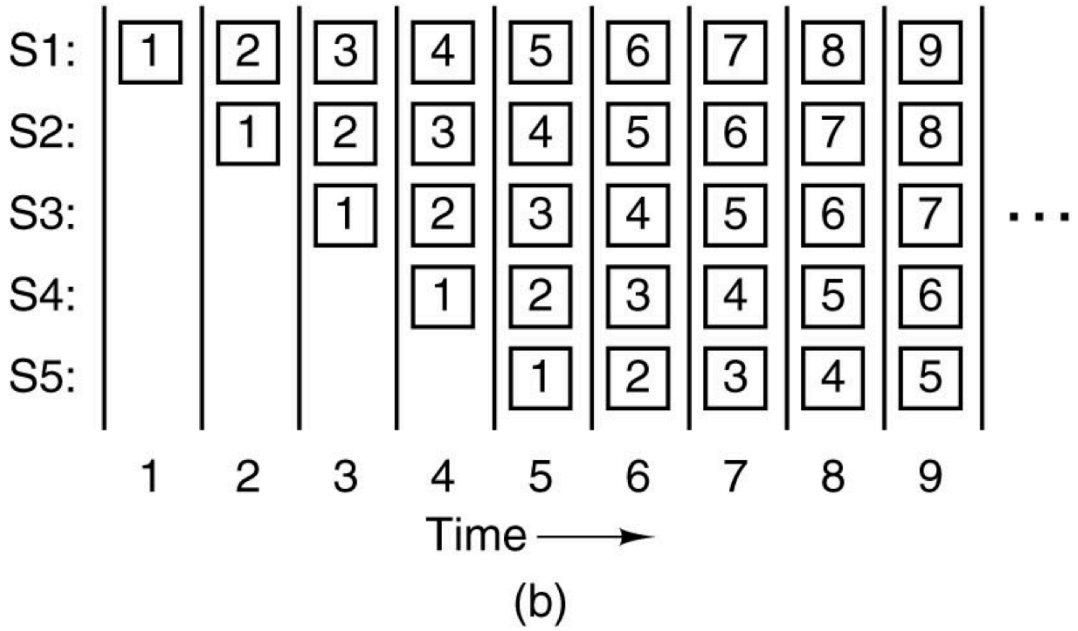
CPU execution cycle

- Two key registers
  - ➔ **Instruction counter**: *address* of next instruction
  - ➔ **Instruction register**: *value* of current instruction
- Infinite loop
  1. **Fetch instruction** from memory
  2. **Decode** instruction
  3. **Fetch operand** from memory (if needed)
  4. **Execute** instruction on operand (ALU)
  5. **Write back** results to memory (if needed)
  6. Loop back from start

Optimization: Pipelining



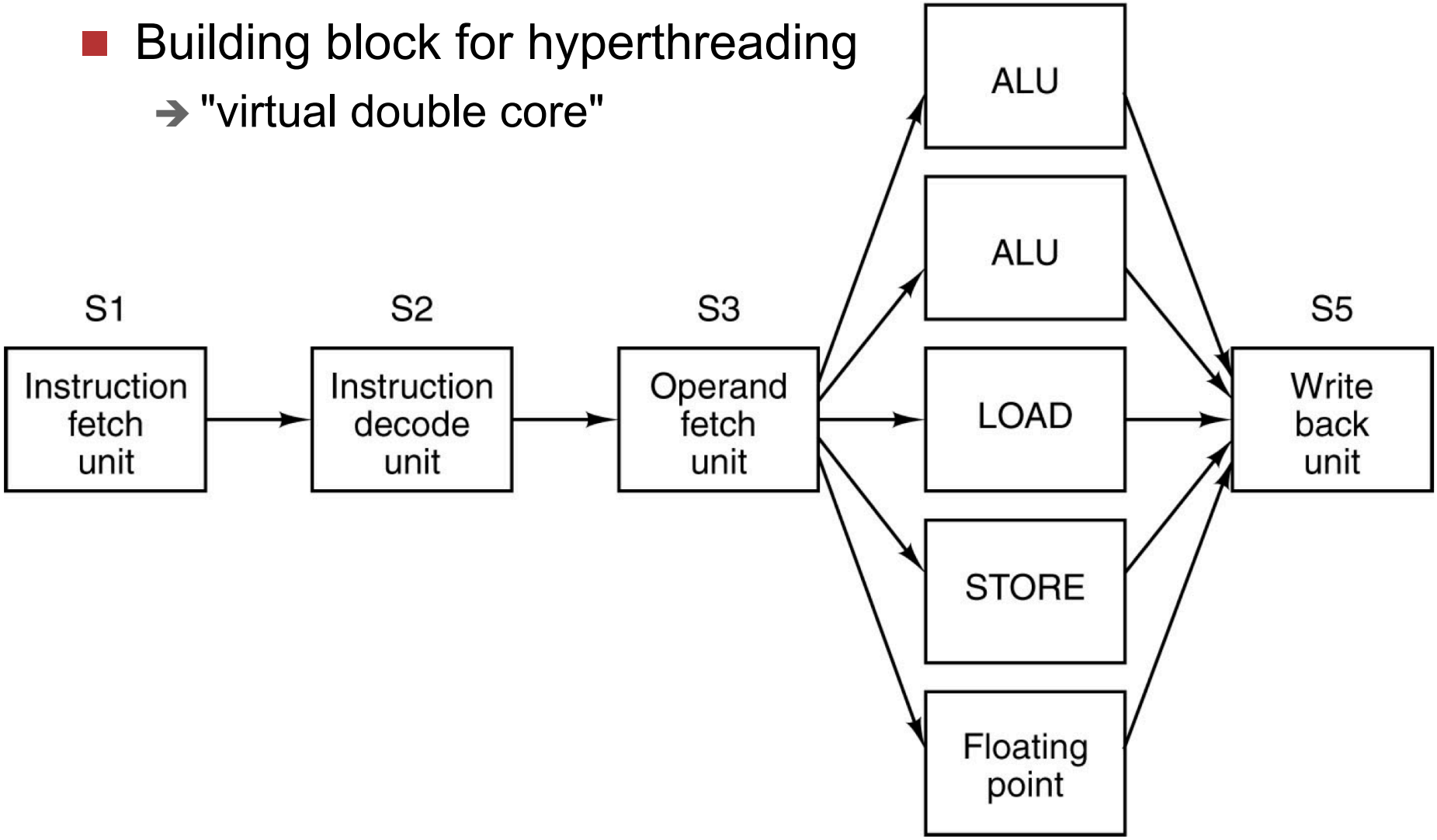
(a)



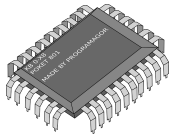
(b)

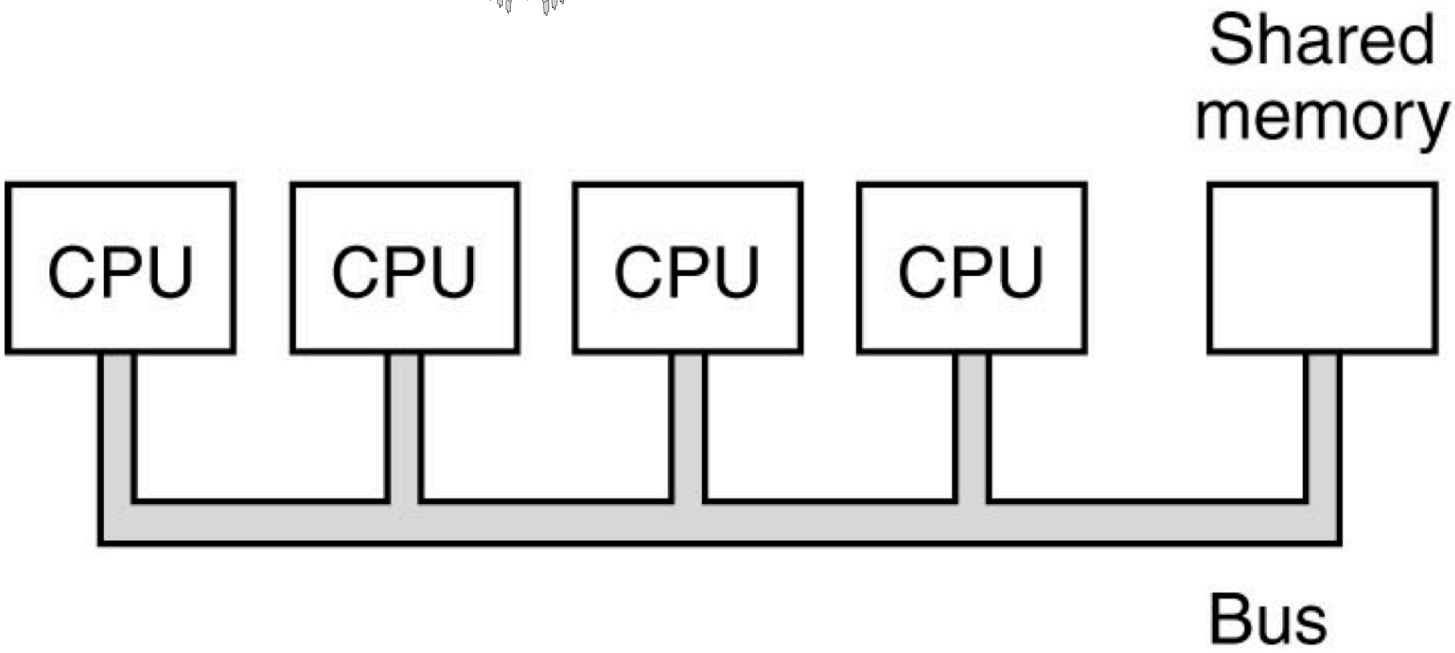
More: Superscalar Architecture

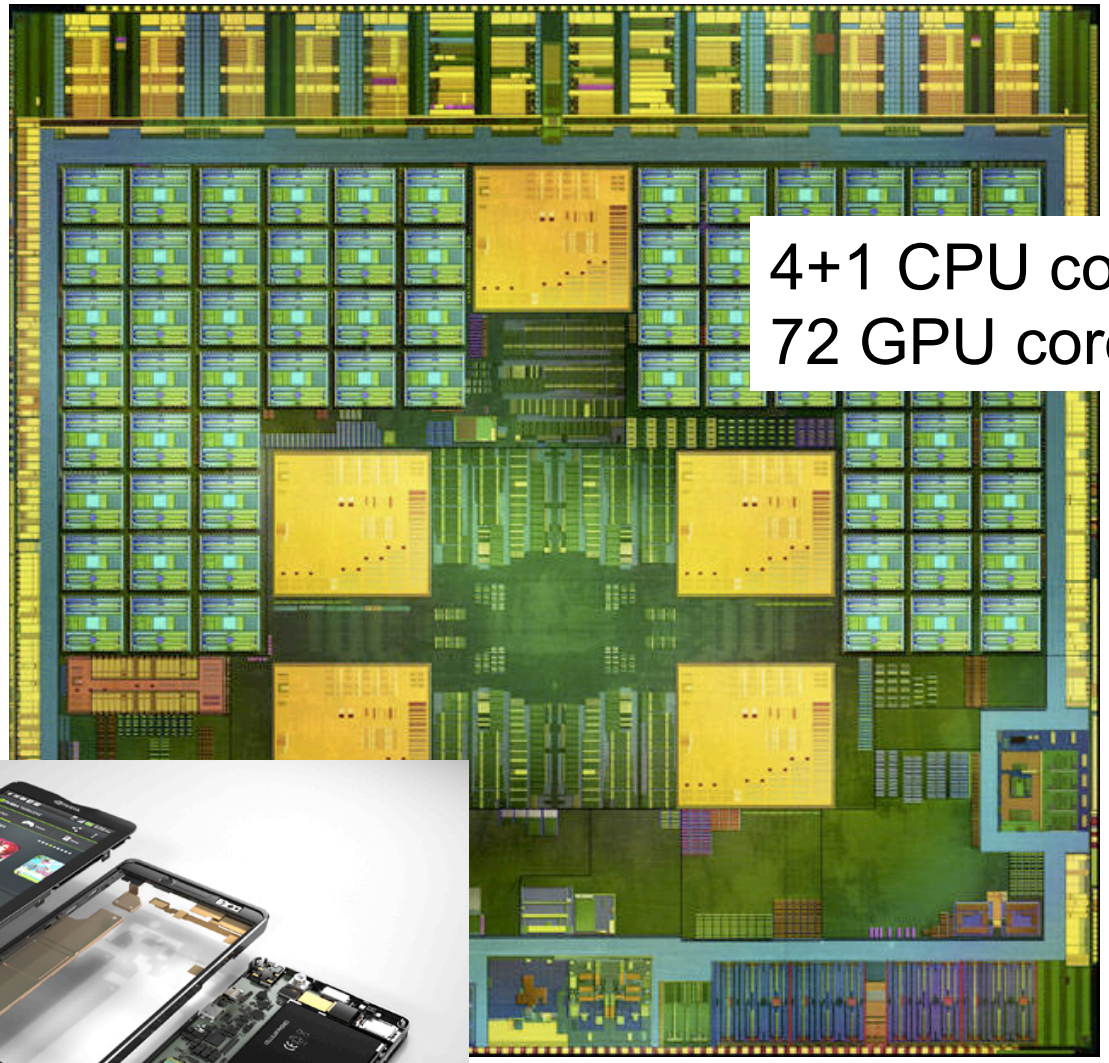
- Building block for hyperthreading
  - ➔ "virtual double core"



Even more: multi-core







4+1 CPU cores
72 GPU cores

Tegra 4 processor
from Nvidia

http://www.theregister.co.uk/Print/2013/01/11/ten_stars_of_ces_2013/

Learning Outcome

- You should be able to explain the role and differences between source code, bytecode, assembly code, and machine code.
- You should be able to explain how a Java/C/C++ program executes, in particular the steps involved from source code to actual execution.
- You should be able to explain what makes up a CPU, and the different optimizations that are in used today in modern architectures.