forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 5

Memory and low-level use & PCB basics

Geheue en lae-vlak gebruik & PCB basics

Dr Rensu Theart & Dr Lourens Visagie

Lecture Overview

- Types of memory (background)
- Program and data memory
- Global vs. local C variables
- Stack
- Heap
  - Dynamic memory allocation
- PCB Basics
- Integrated circuits

Memory types / Geheue Tipes

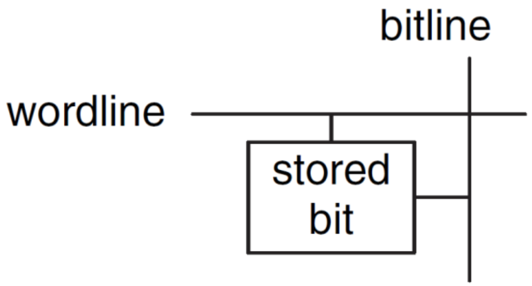
There exists two broad categories of memory.

- RAM = Random Access Memory
  - Data can be read/written (a byte at a time) at almost equal speeds.
  - Usually intended for program data (variables).
  - Volatile, meaning that it loses its data when the power is turned off.

- ROM = Read Only Memory
  - From a program execution point-of-view, can only be read.
  - Can be re-programmed but requires higher supply voltage or special procedure.
  - Usually used for program instructions.
  - Non-volatile, meaning that it retains its data indefinitely, even without a power source.

Bit cells / Bis selle

- Memory is built as an array of **bit cells**, each of which stores 1 bit of data.
- For each combination of address bits, the memory asserts a single *wordline* that activates the bit cell in that row.
  - When the *wordline* is HIGH, the stored bit transfers to or from the *bitline*.
  - Otherwise, the *bitline* is disconnected from the bit cell.
- To **read** a bit cell, the *bitline* is initially left floating (Z). Then the *wordline* is turned ON, allowing the stored value to drive the *bitline* to 0 or 1.
- To **write** a bit cell, the *bitline* Is strongly driven to the desired value. Then the *wordline* is turned ON, connecting the *bitline* to the stored bit. The strongly driven *bitline* overpowers the contents of the bit cell, writing the desired value into the stored bit.



Bit cells organization example

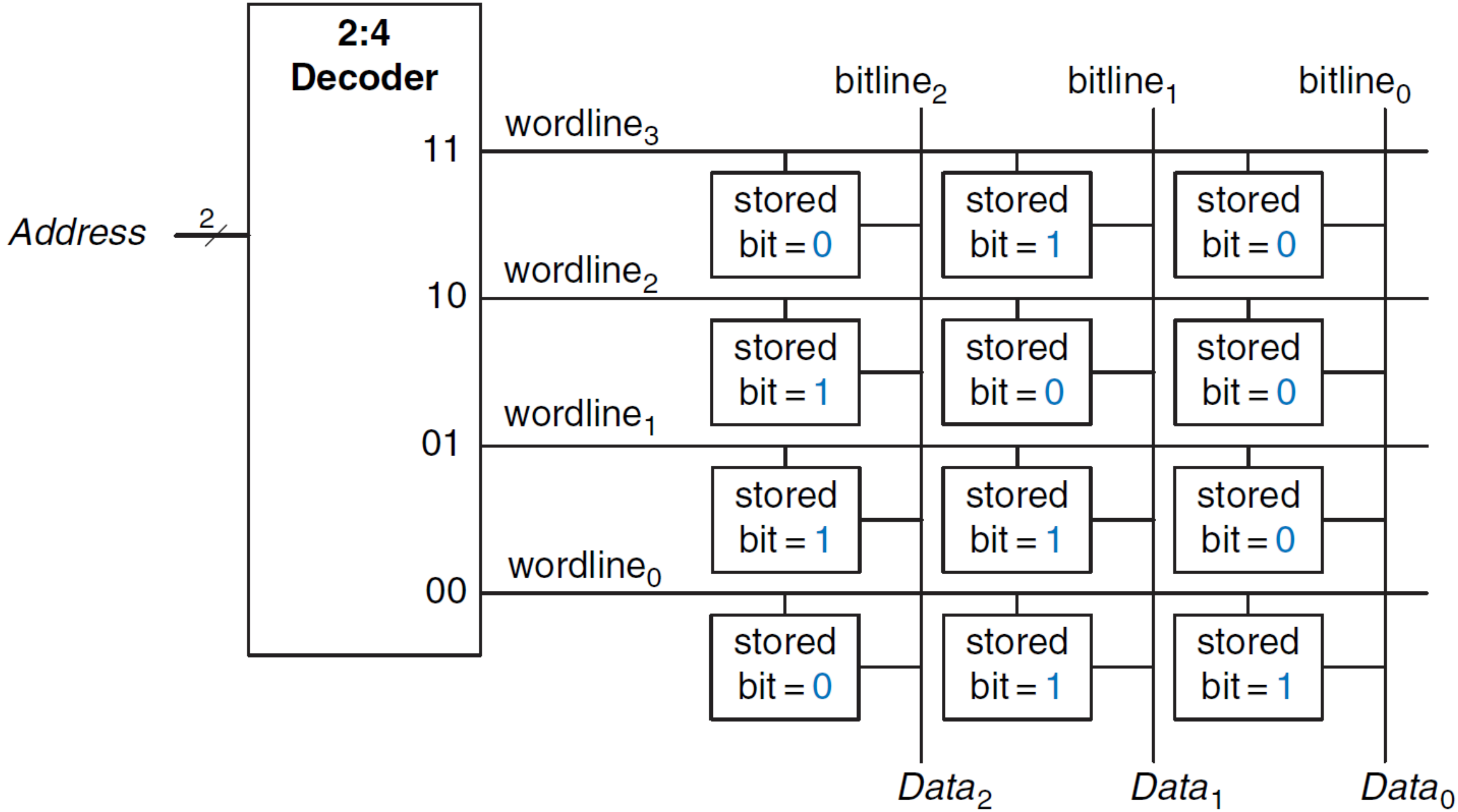
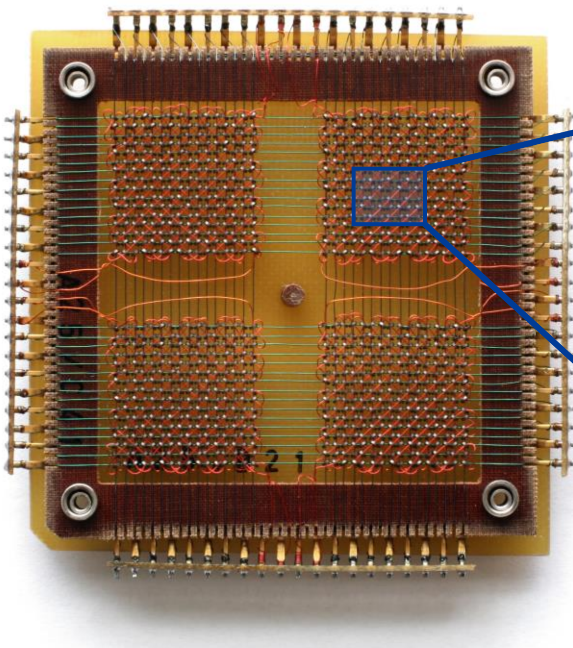
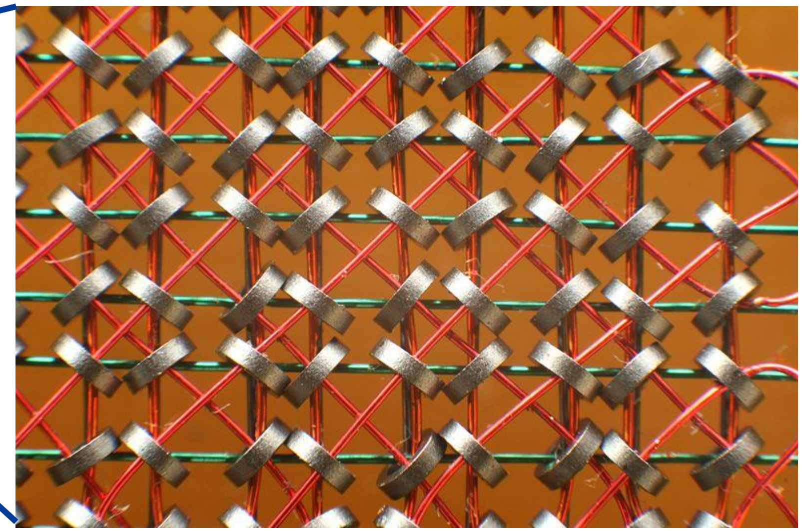


Figure 5.43 **4 × 3 memory array**

Historic memory / Historiese geheue



A 32 x 32 core memory plane storing 1024 bits (or 128 bytes) of data



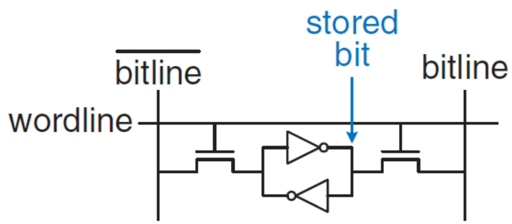
Hand woven magnetic core memory

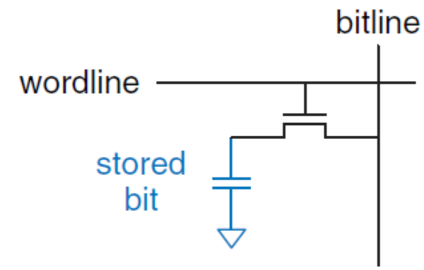
⇒ Cores maintain state even after power switched off making them non-volatile.

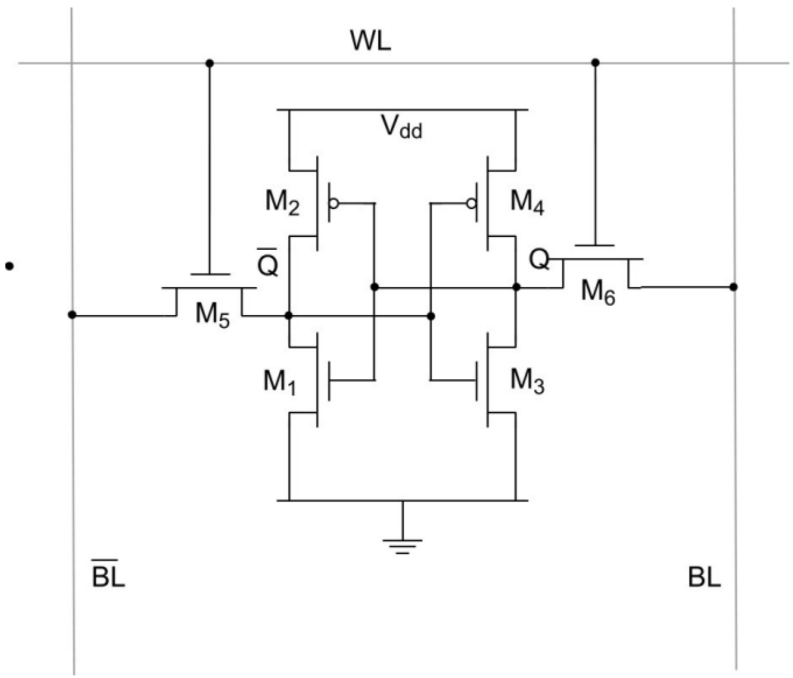
DRAM and SRAM

- Dynamic RAM (**DRAM**)
  - Stores a bit as the presence or absence of charge on a capacitor.
  - Reading destroys the bit value on the capacitor, so the data must be restored (rewritten) after each read.
  - Must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.
  - Can be made with higher density – used in PC for "bulk" system memory
- Static RAM (**SRAM**)
  - Is **static** because the store bits do not need to be refreshed.
  - The data bit is stored on cross coupled inverters (a bistable element) – which is a type of flip-flop.
  - Much faster than ROM.
  - Uses relatively many parts (transistors) per memory cell – lower density than DRAM.
  - SRAM used for cache memory.
- In SRAM and DRAM, memory contents are volatile – they lose their state when power is turned off

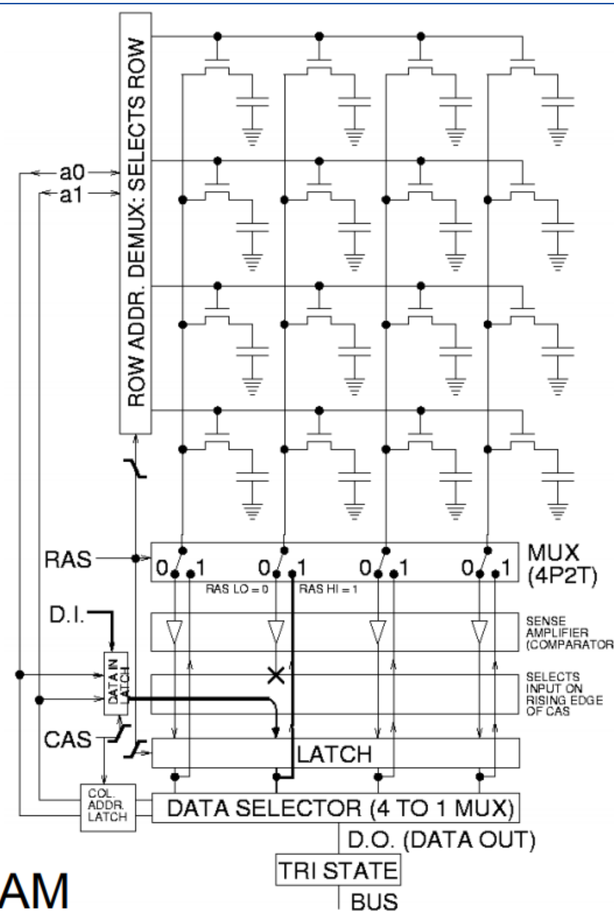
DRAM and SRAM







SRAM



DRAM

Memory comparison / Geheue vergelyking

Table 5.4 Memory comparison

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

ROM and PROM

- **ROM** stores a bit as the presence or absence of a transistor.
  - The contents of the ROM bit cell are specified during manufacturing by the presence or absence of a transistor in each bit cell.

- A **programmable ROM (PROM)** places a transistor in every bit cell but provides a way to connect or disconnect the transistor to ground.
  - Here is an example of a **fuse-programmable ROM** which is only a one-time programmable ROM, since the fuse cannot be repaired.

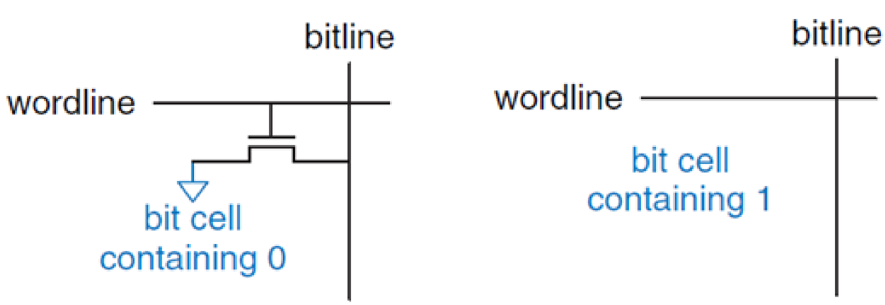


Figure 5.49 ROM bit cells containing 0 and 1

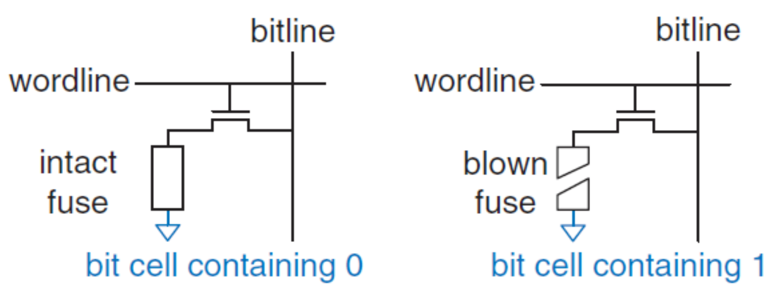


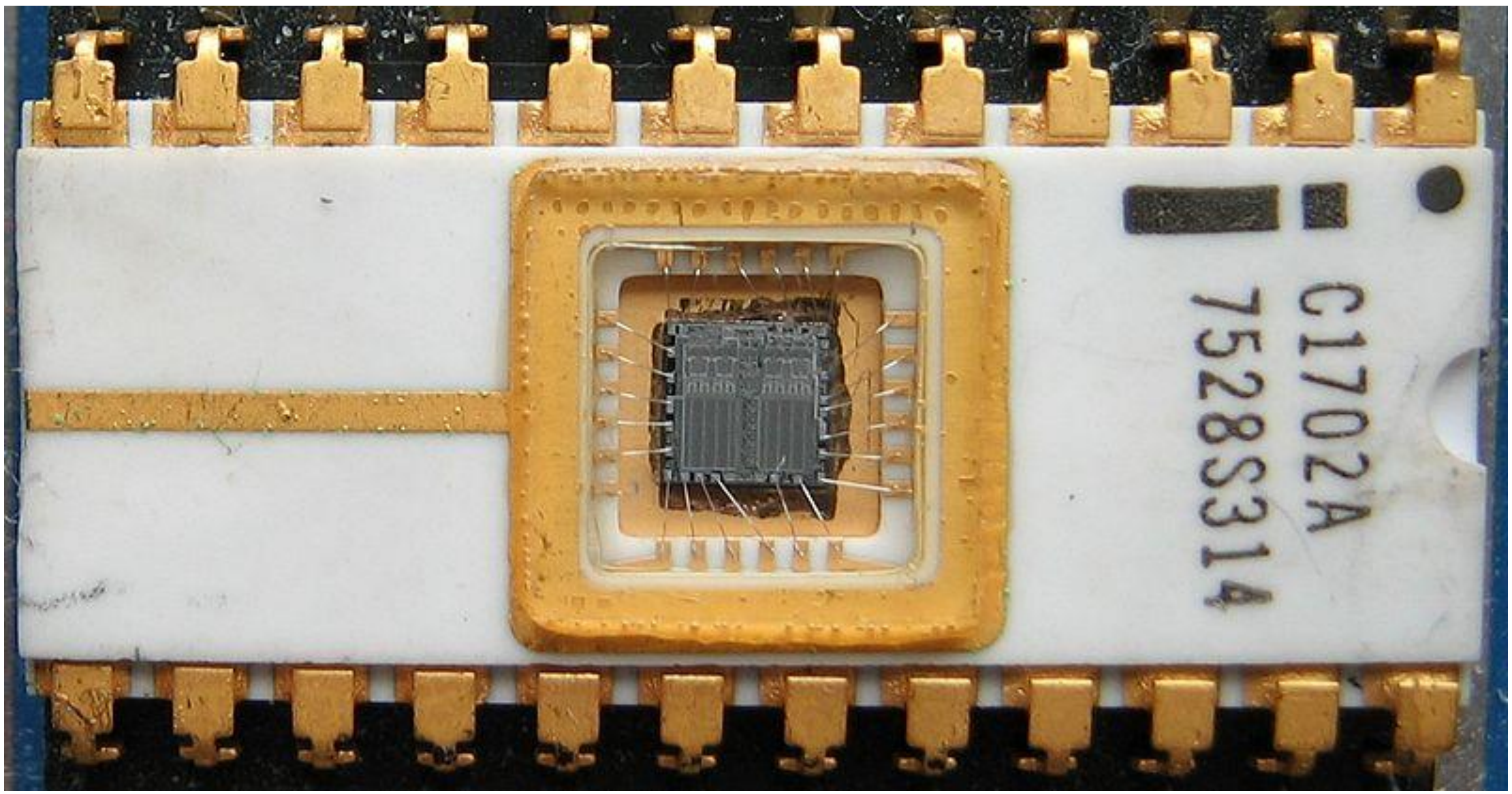
Figure 5.52 Fuse-programmable ROM bit cell

EPROM

- Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND.
- **Erasble PROM (EPROM)**, which replaces the nMOS transistor and fuse with a floating gate transistor, with the gate not physically attached to any wires.
  - If a suitably high voltage (higher than operating voltage) is applied the electrons tunnel through an insulator onto the floating gate, turning the transistor on.
  - It is only erasable by exposing EPROM to intense ultraviolet (UV) light through quartz window shining on silicon

Historic memory / Historiese geheue

- First EPROM chip, an Intel 1702 from 1971.
  - 256 bytes storage
- Erased by exposure to strong ultraviolet light.



EEPROM and Flash

- **Electrically erasable PROM (EEPROM)**, as well as **Flash memory**, contains circuitry on the chip for erasing as well as programming it, without using UV light or removing it from the circuit.
- Higher programming voltage still required but generated inside chip using charge pumps.
- There are some differences between EEPROM and Flash.
- EEPROM
  - Can be erased a byte or word at a time
  - Erase and write takes relatively long
- Flash memory
  - Erased and programmed in blocks (or entire chip at once)
  - High density (SD cards & USB drives).
  - Erase changes all bits in the device to ‘1’
  - Two types: NAND and NOR

Flash memory

	NAND Flash	NOR flash
Density	High	Low
Cost	Low per bit	Higher per bit
Read speed	Medium	High
Write speed	High	Low
Erase speed	High	Low
Interface / access type	Complex I/O interface – (serial transmission of address/control/data signals)	Same as SRAM – Enough address pins to map entire chip – access to any random byte
Application	Frequent write operations - SD cards / USB drives	Infrequent write operations, fast access – Program memory (boot code)

Now…

That's very interesting, but…

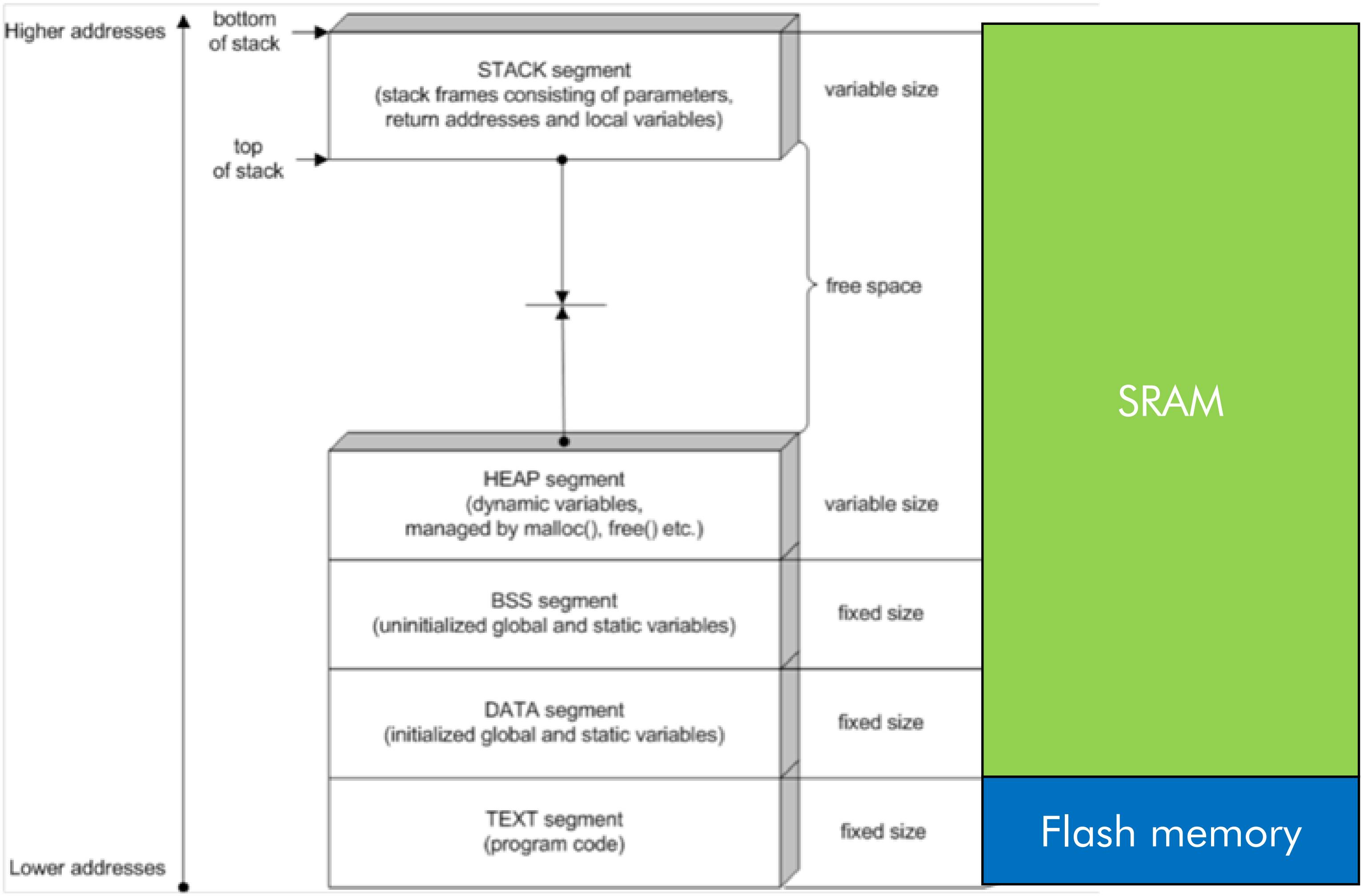
The following is what you need to know as a programmer

Memory / Geheue

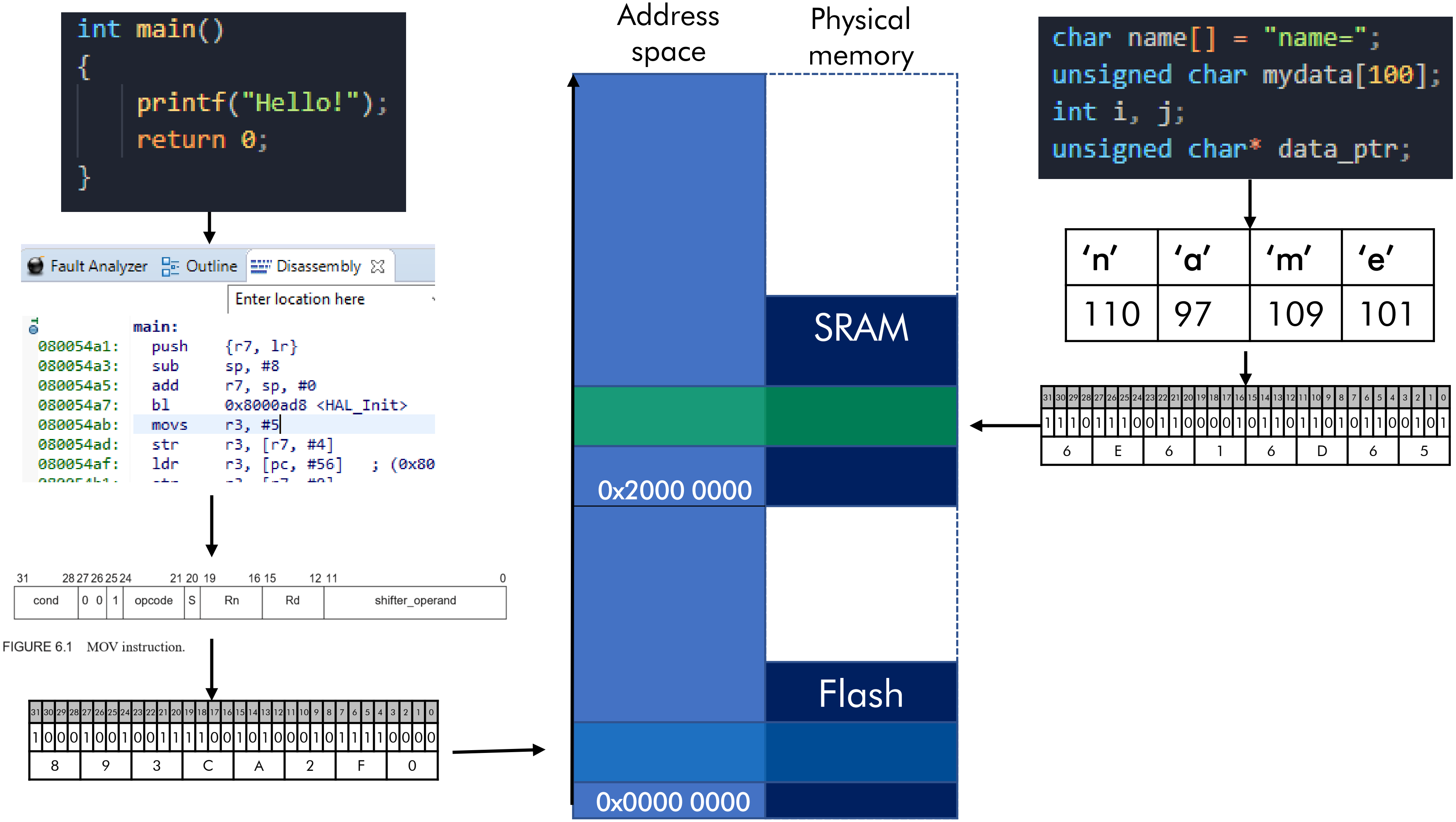
- A memory cell stores a single bit of information – depends on physical implementation:
  - SRAM → output of bi-stable element (flip flop)
  - DRAM → charge of a capacitor
  - Flash memory → Floating-gate transistor
- Single bit of information is interpreted as a logical 0 or 1.
- 8 bits of information grouped together as a byte.
  - Computer memory is organized in bytes, addressed in bytes, size is measured in bytes
- The system bus allows CPU to access memory directly (fetch instructions, load and store data).
- Some memory types cannot be accessed directly by CPU – e.g. hard disks, SD cards, etc.

Memory / Geheue

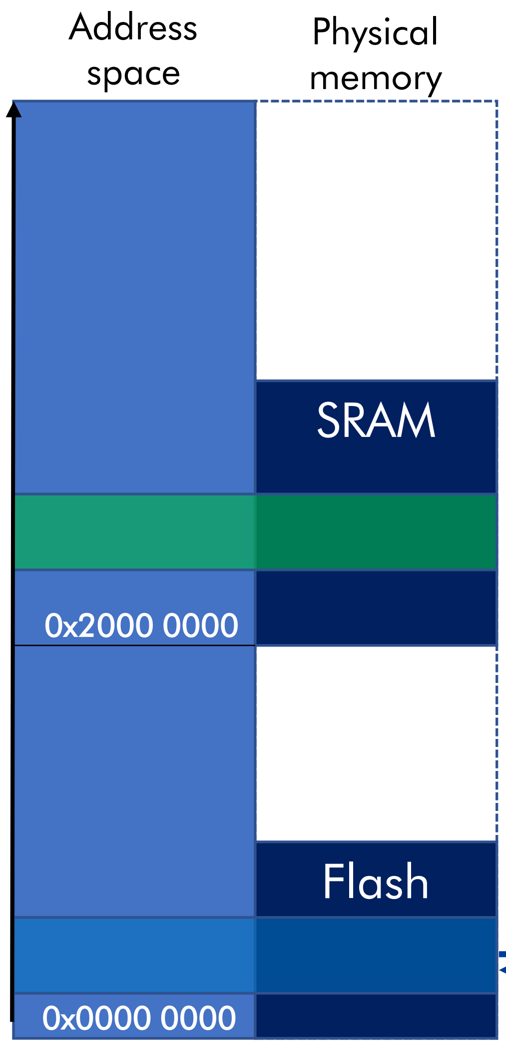
Memory layout of a C program



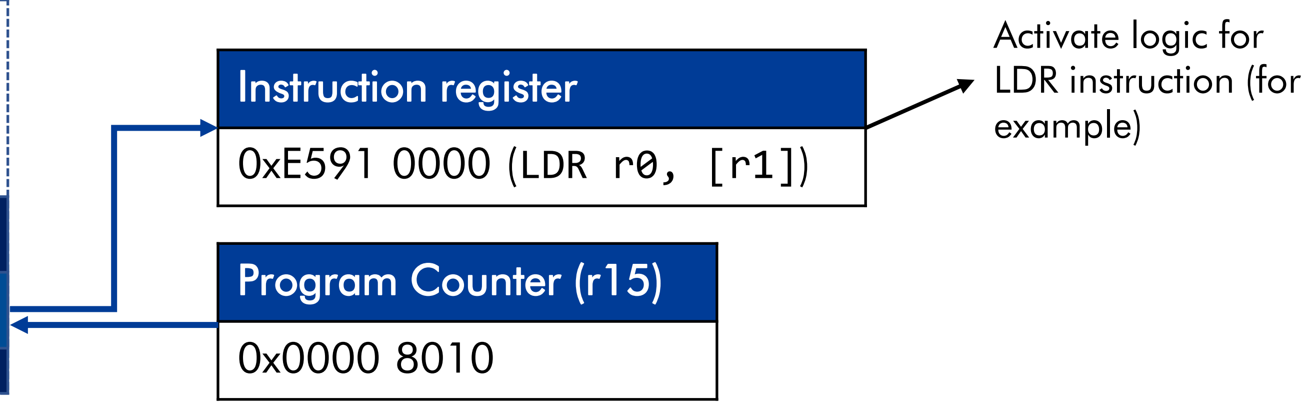
Program vs. data memory / Program en data geheue



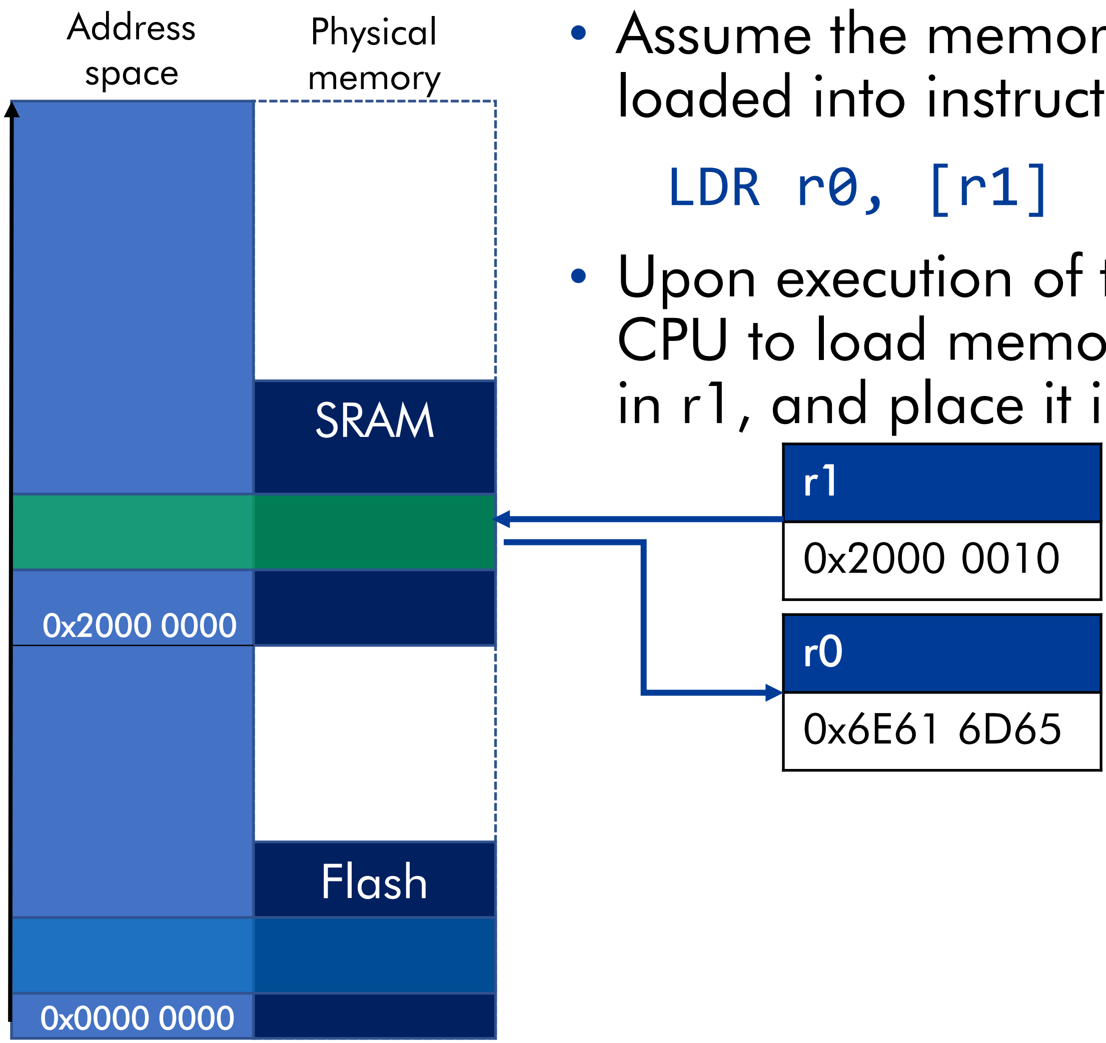
Program vs. data memory / Program en data geheue



- Instruction pointer has address of next instruction
- CPU fetches memory contents at that address (through system bus) and places it in the instruction register
- Instruction is decoded and executed



Program vs. data memory / Program en data geheue



- Assume the memory contents that was loaded into instruction register is

```
LDR r0, [r1]
```

- Upon execution of this instruction: Causes CPU to load memory contents using address in r1, and place it into r0

Program vs. data memory / Program en data geheue

What about…

- Constant values?

```
float energy = mass * 299792458 * 299792458;
```

(where does the 299792458 go in memory?)

- Short-lived variables (i.e. loop variables)?

```
for (int i = 0; i < 100; i++) { array[i] = 0;}
```

(Will ‘`i`’ be stored in memory?)

- Variables with initializers?

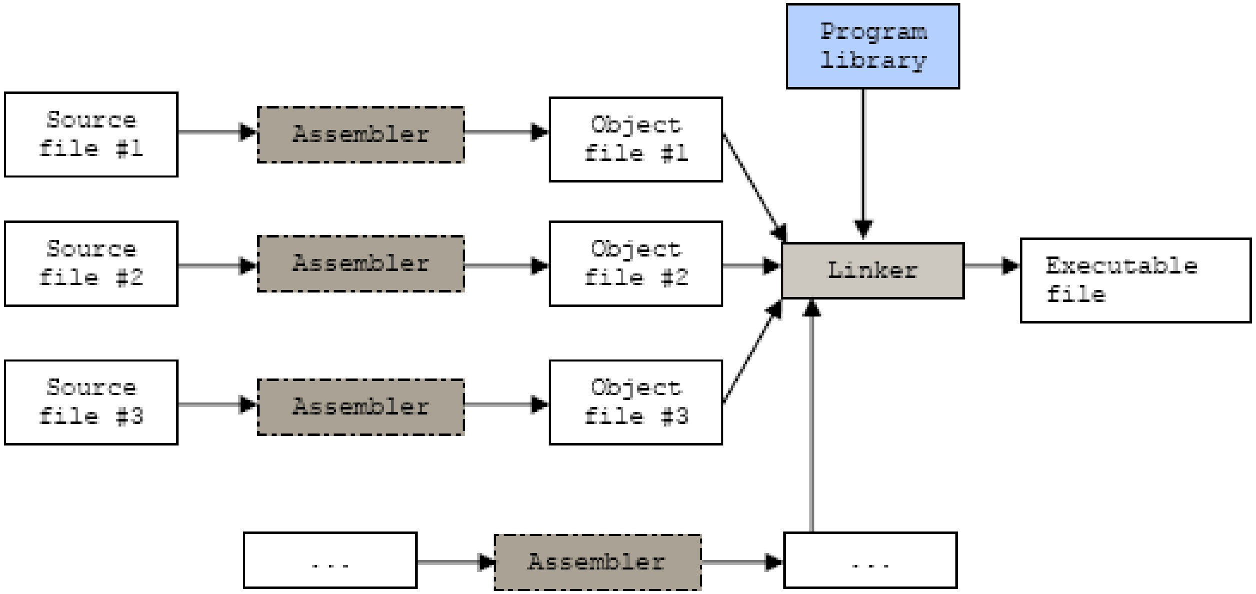
```
int studentnum = 12345678;
studentnum = studentnum + 1;
```

- What else is stored in memory?
  - Vector table… (more about this later)

Program vs. data memory / Program en data geheue

- How does program code end up at address 0 – 0x1FFF FFFF, and data (variables) end up at address 0x2000 0000 and onwards?

⇒ The linker! (takes all the object files and combines it into an executable)



⇒ The debugger/programmer copies the binary file to non-volatile memory (flash memory in our case)

Memory and Linking / Geheue en skakeling

Linker command file: tells the linker where to put everything

```
MEMORY
{
   FLASH (RX) : origin = 0x00000000, length = 0x00040000
   RAM (RWX) : origin = 0x20000000, length = 0x00008000
}

/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    …
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    …
  } >FLASH

  /* Constant data goes into FLASH */
  .rodata :
  {
    …
  } >FLASH

  /* used by the startup to initialize data */
  _sidata = LOADADDR(.data);

  /* Initialized data sections goes into RAM, load LMA copy after code */
  .data :
  {
    …
  } >RAM AT> FLASH

  /* Uninitialized data section */
  . = ALIGN(4);
  .bss :
  {
    …
  } >RAM
```

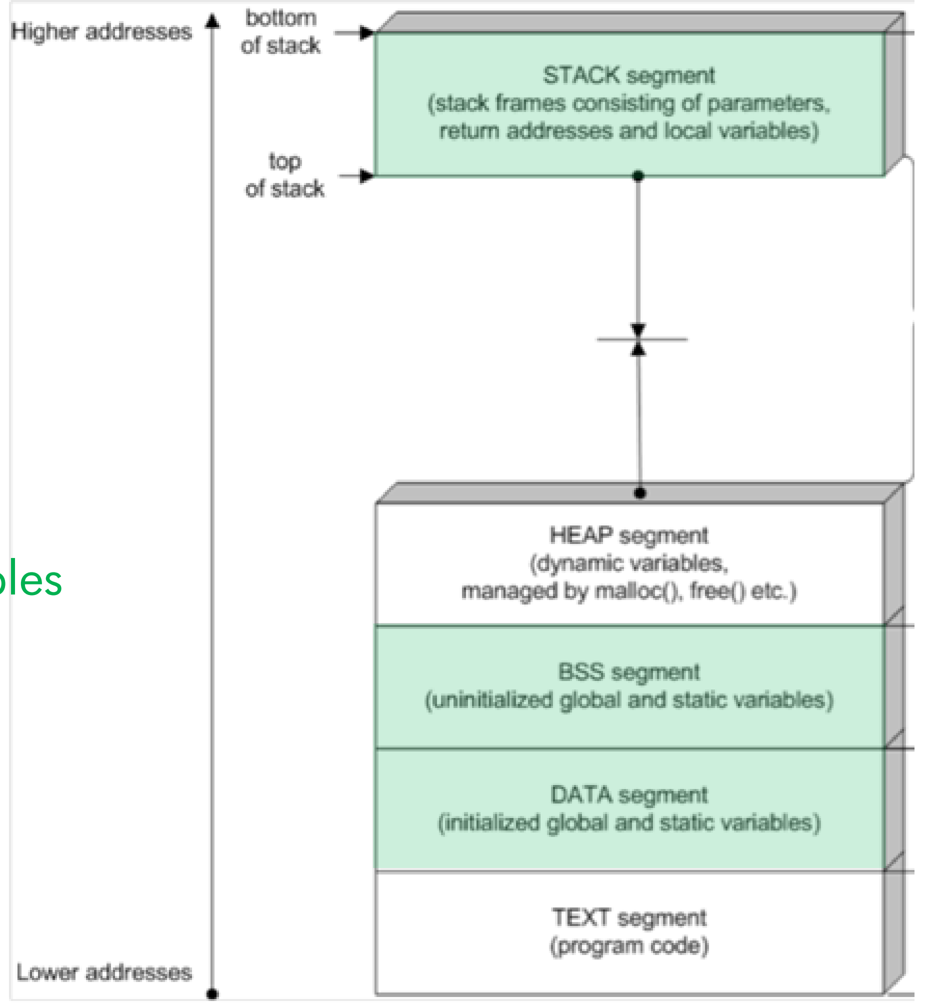
Global vs. Local variables / Globale en lokale veranderlikes

- Global variables
  - Declared outside of a function
  - Visible (can be used) in any function
  - Located in the **DATA** or **BSS** segments
- Local variables
  - Declared inside a function
  - Can only be used inside that function
  - Located on the **STACK**

```c
#include <stdio.h>
int maxVal = 0;
int array[] = {1, 2, 3, 4};
void findMax(void)
{
    int i;
    for (i = 0; i < 4; i++) {…}
}
int main(void)
{
    findMax();
    printf("%d", maxVal);
    return 0;
}
```

Global variables

Local variable



Variables general tips / Veranderlikes algemene wenke

- Initialise variables before using them

```c
int maxVal;
printf("%d", maxVal);
```

```
>>warning: 'maxVal' is used uninitialized in this function [-Wuninitialized]
```

- Good practice is to declare all local variables in the beginning of the function and all global variables towards beginning of source file.
- Do not use the same name for a global and local variable (even though this is allowed).
- Use descriptive variables names.
- Use camelCase for variable and function names:
  - E.g. `newString`, `getNewString()`

Variables general tips / Veranderlikes algemene wenke

- Keep global variables to a minimum → helps to improve modularity

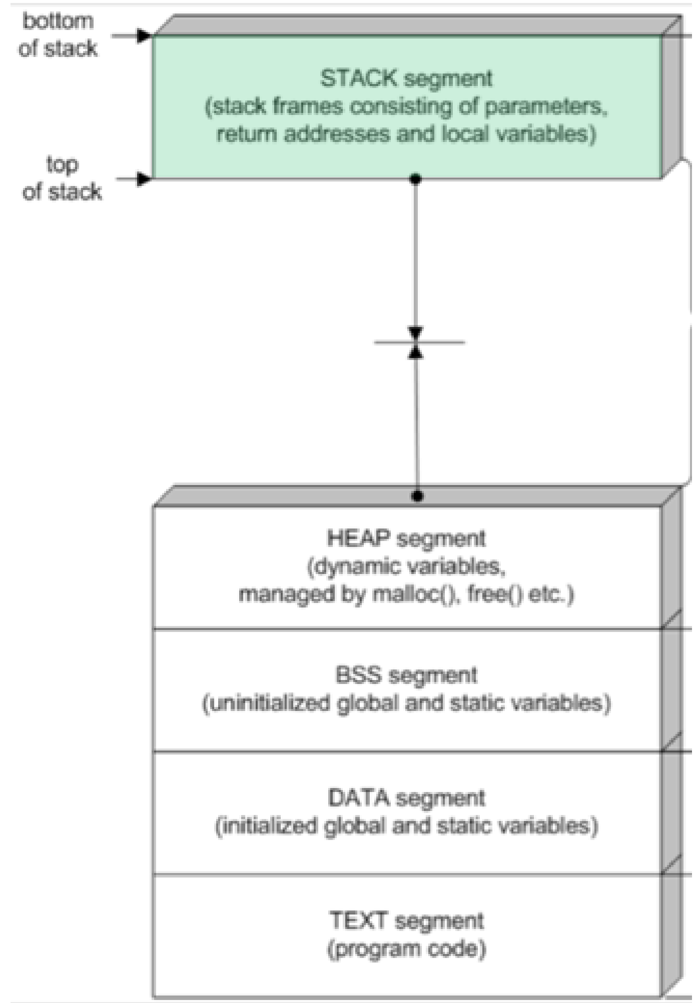
```c
#include <stdio.h>
int findMax(int* array, int len)
{
    int maxVal = 0;
    for (int i = 0; i < len; i++)
    {
        if (array[i] > maxVal)
            maxVal = array[i];
    }
    return maxVal;
}

int main(void)
{
    int values[] = {1, 2, 3, 4};
    printf("%d", findMax(values));
    return 0;
}
```

The `findMax` function can now be re-used with different arguments

Local variables and the stack / Lokale veranderlikes en die stapel

- Local variables are created on the **STACK.**
- They persist only as long as the function is executing.
- The stack grows every time a function starts and shrinks after the function exits. Memory gets re-used.

```c
void do_a()
{
    int testval = 4;
    printf("%d\n", testval);
}
void do_b()
{
    int testval2 = 5;
    printf("%d\n", testval2);
}
int main(void)
{
    do_a();
    do_b();
}
```



Local variables and the stack / Lokale veranderlikes en die stapel

- For recursive functions, the LR and arguments must repeatedly be stored to the stack.

  ⇒ stack keeps on increasing

  ⇒ stack overflow

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

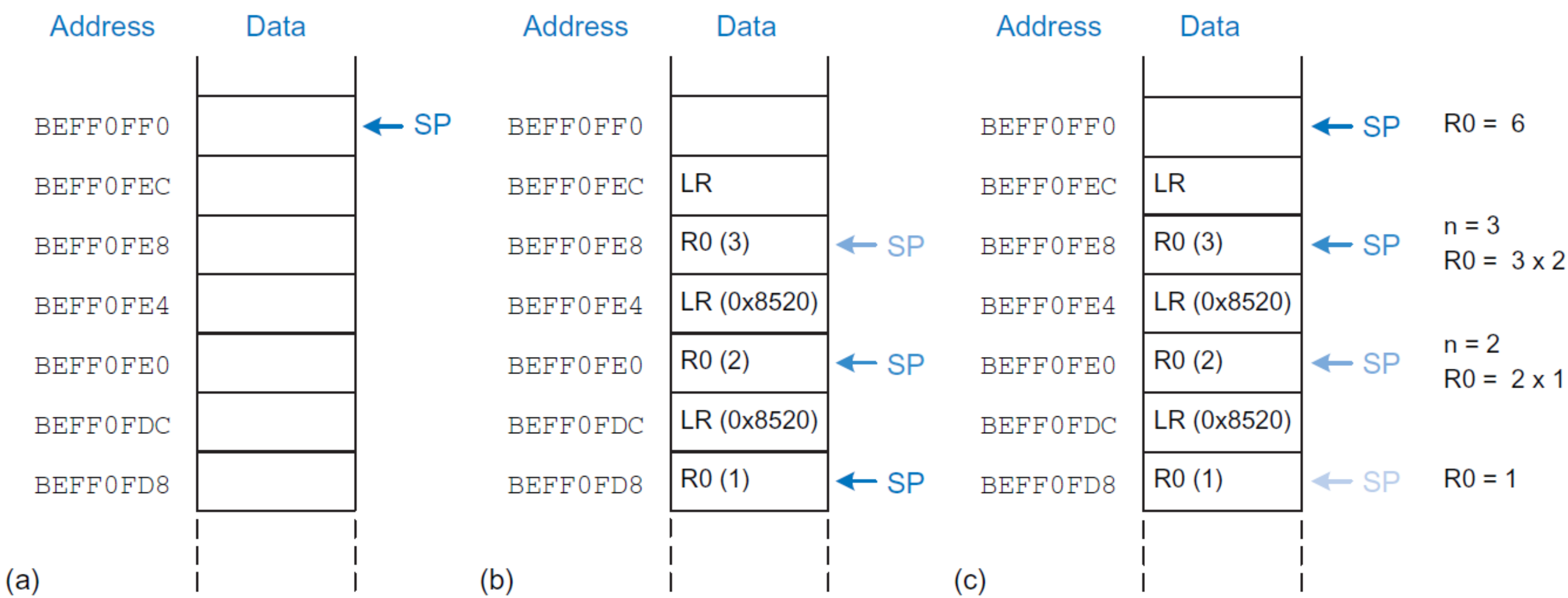
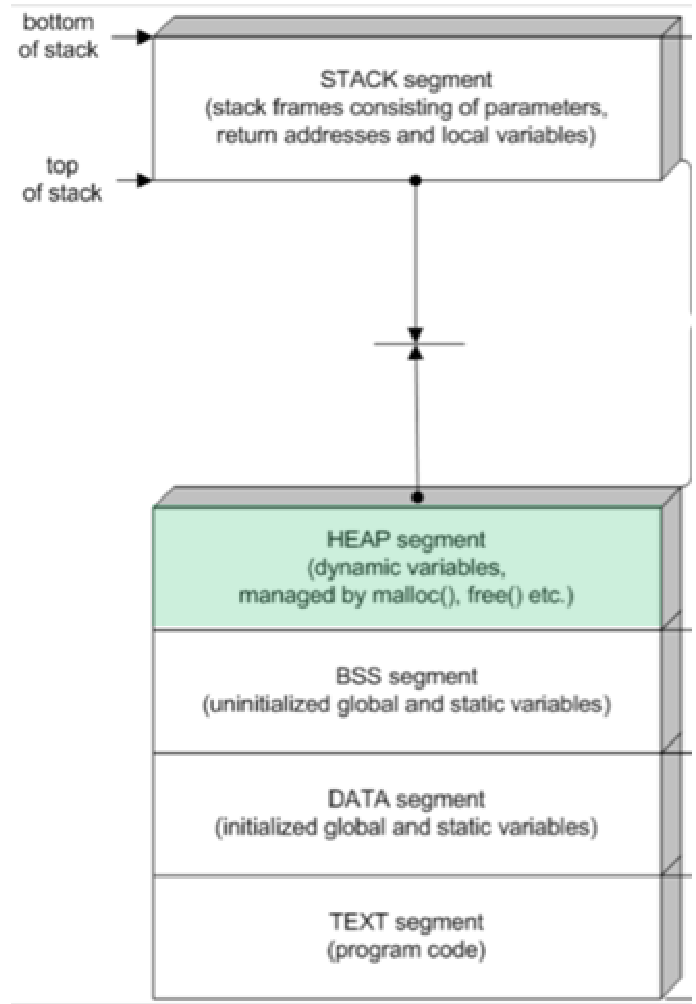


Figure 6.14 Stack: (a) before, (b) during, and (c) after factorial function call with $n = 3$

Dynamic memory allocation/ Dinamiese geheue allokasie

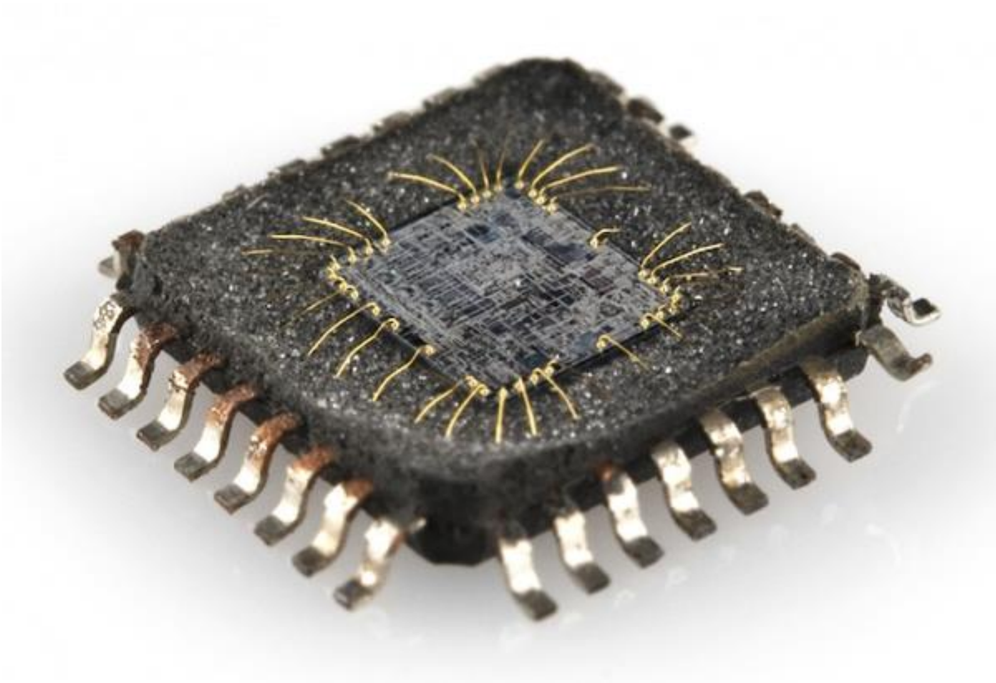
- Use `malloc()` and `free()` functions to dynamically allocate memory
- Must `#include <stdlib.h>`
- Dynamic memory is allocated from the **HEAP**
- Don't do it on an embedded system.
  - Behavior of the critical systems should be deterministic.
  - Also only really useful for dynamically running multiple processes simultaneously that all share the available RAM.

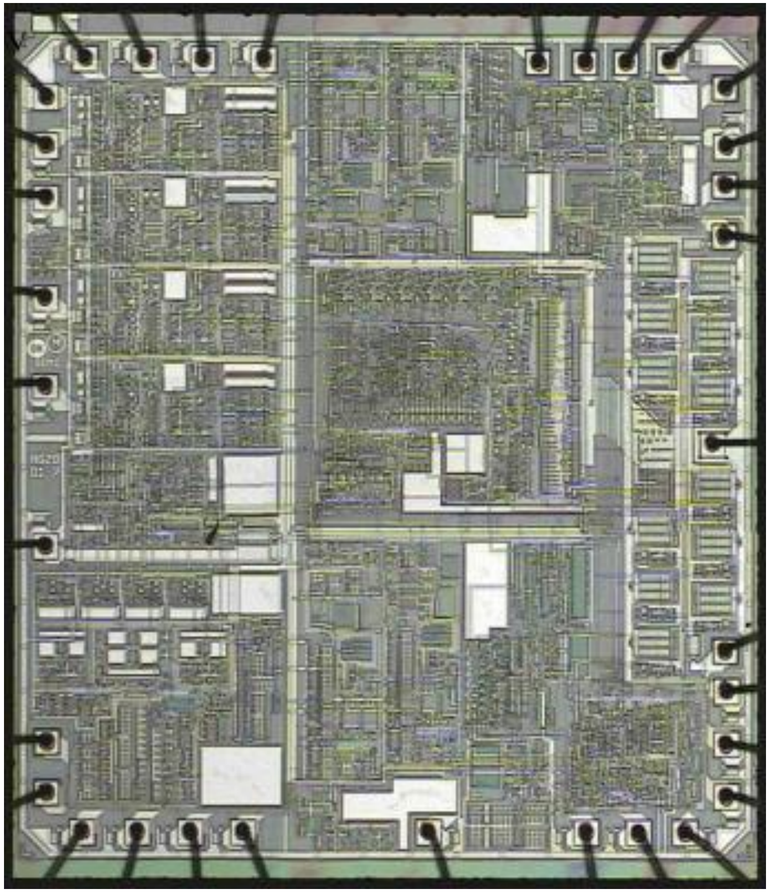


Different digital systems / Verskillende digitale stelsels

- **Microprocessor / CPU** : Single device with registers, ALU (Arithmetic Logic Unit) and logic to decode instructions
- **Microcontroller** : Microprocessor with embedded memory and peripherals, such as GPIO, analog⇄digital conversion, timers
  - (embedded = all in the same device)
- **FPGA (Field Programmable Gate Array)** : array of configurable logic elements (LEs), also called configurable logic blocks (CLBs), connected together with programmable wires.
  - LEs contain small lookup tables with multiplexers and flip-flops
  - Scales easily, so possible to implement very complex logic (even an entire CPU) in FPGA
  - Xilinx and Altera are leading manufacturers
- **ASIC (Application-Specific Integrated Circuit)** : Chips/device that was designed for a particular purpose
  - Examples: graphics accelerators, network interface chips, sensors
  - Hardwired for a specific function
  - Large scale fabrication (high setup cost, but lower unit cost)

Integrated Circuit / Geintegreerde stroombaan

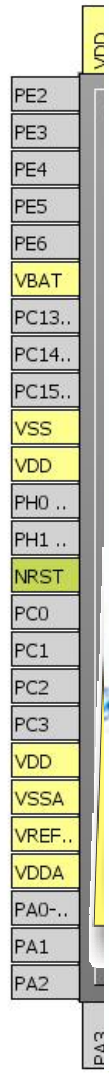


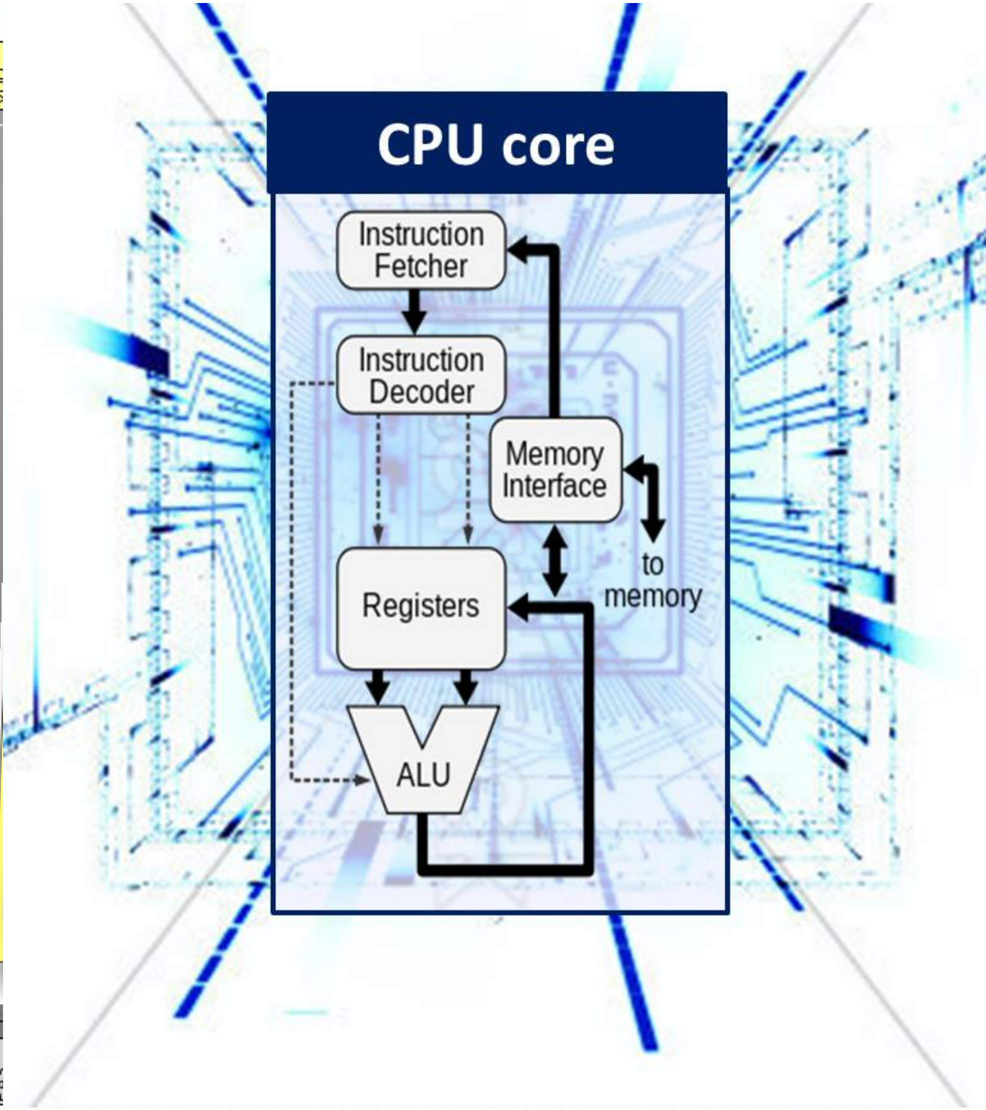


https://learn.sparkfun.com/tutorials/pcb-basics
https://learn.sparkfun.com/tutorials/integrated-circuits

Microcontroller in perspective

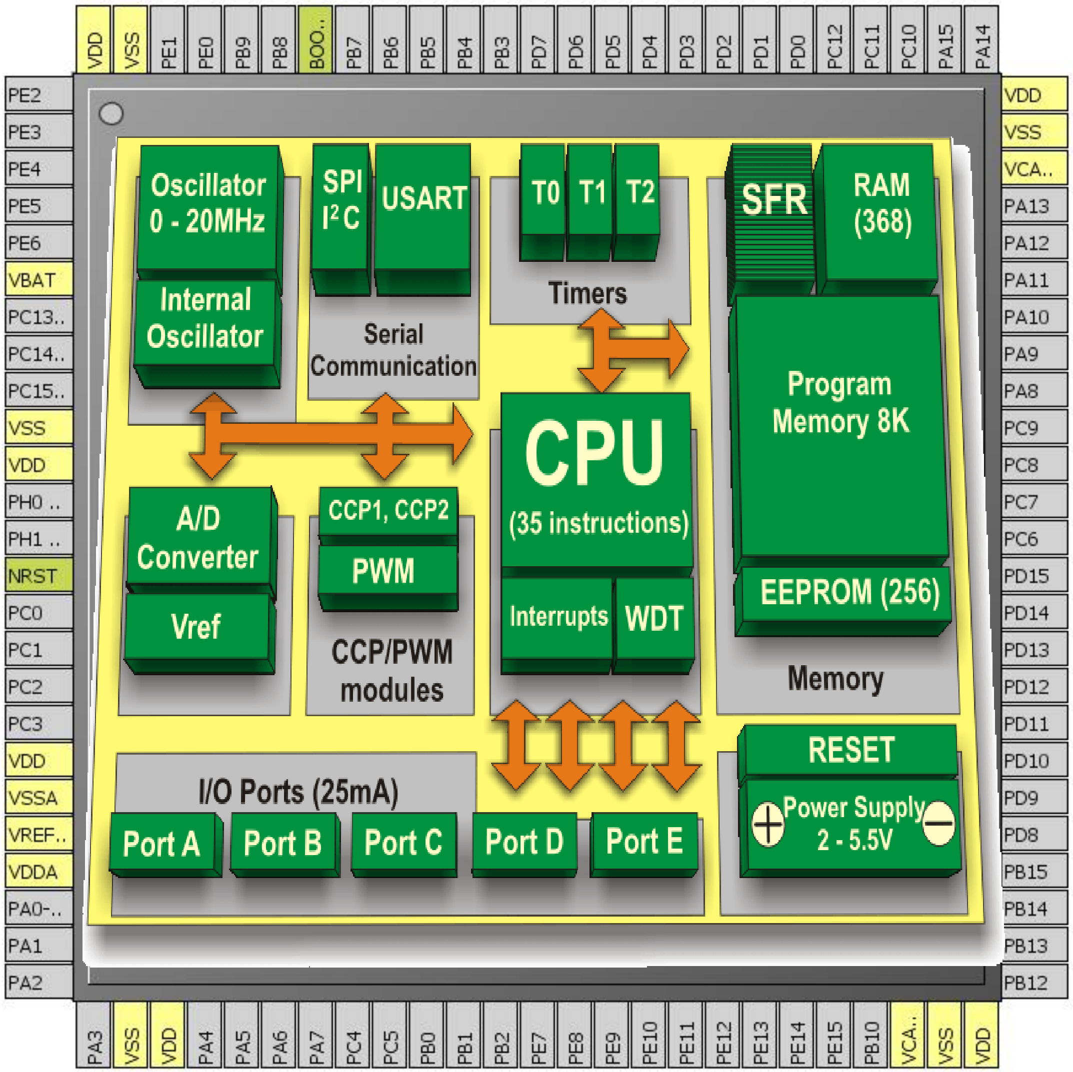
- Inside the **processor core**
  - Registers (r0 to r15, status register etc)
  - ALU (Arithmetic Logic Unit)
  - Logic to fetch and decode instructions
  - Memory interface





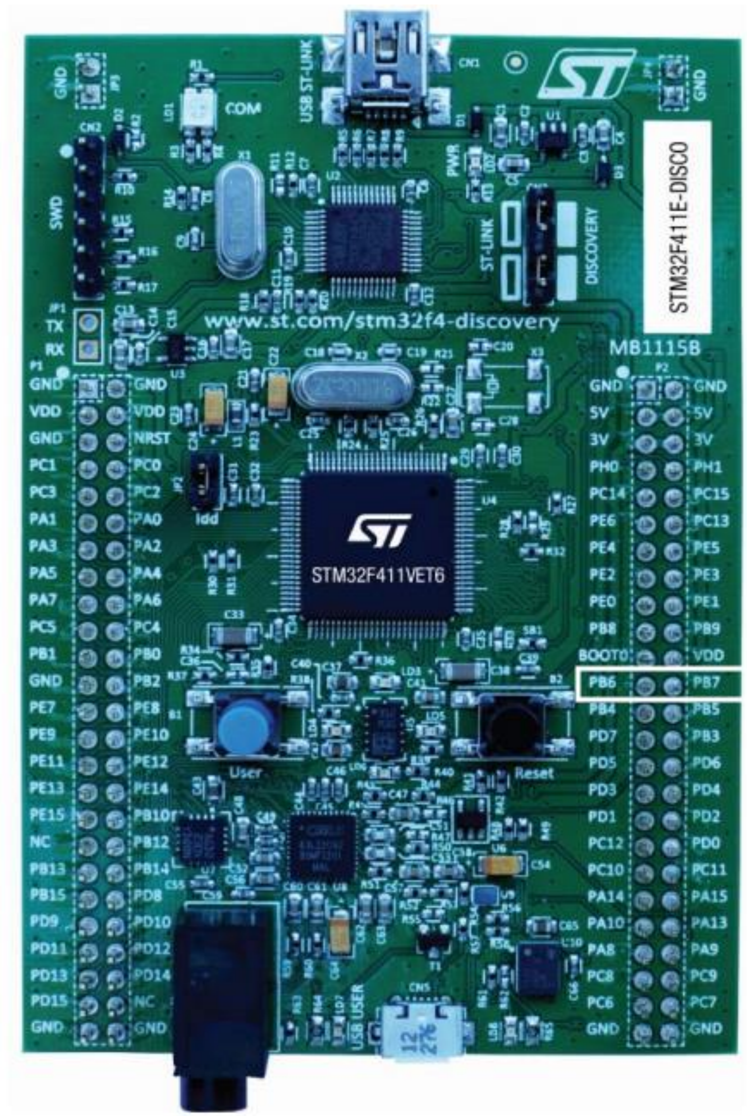
Microcontroller in perspective

- Inside the **microcontroller**
  - The processor core
  - Memory (flash memory for program, SRAM for data)
  - Peripherals (Timers, GPIO ports, Serial ports, A/D)



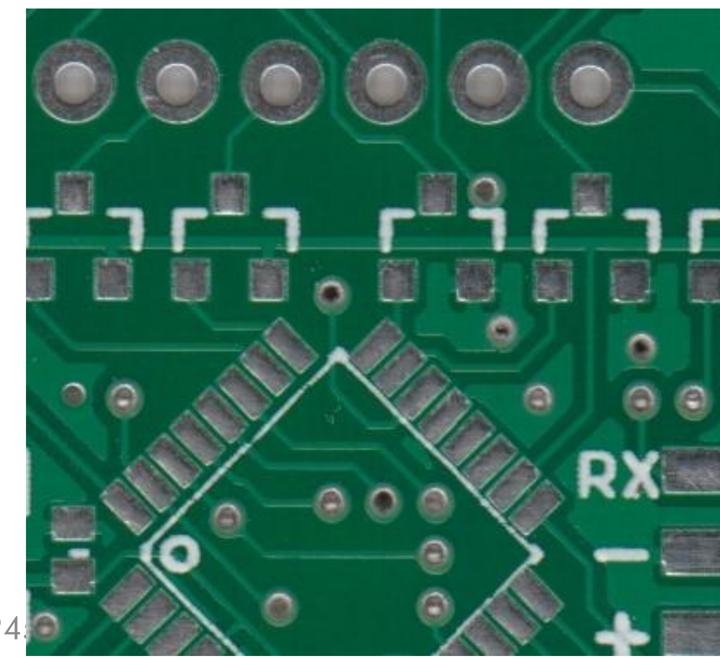
Microcontroller in perspective

- On the **PCB** (printed circuit board)
  - Whatever the design calls for
  - External oscillator
  - External memory
  - Switches
  - Sensors
  - Connectors
  - User input (buttons, touchpad)
  - User display (LCD)
  - External A/D
  - FPGA
  - Other microcontrollers

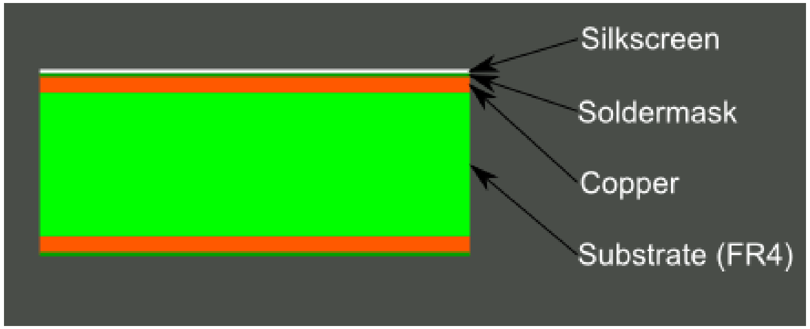


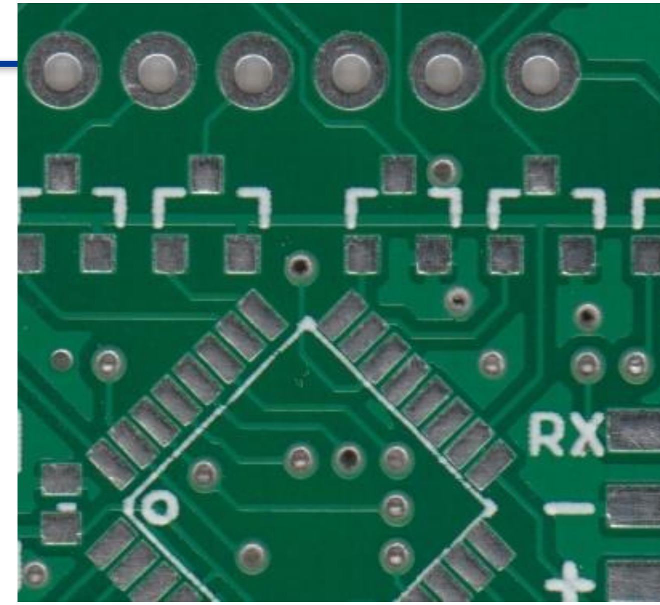
PCB basics

- https://learn.sparkfun.com/tutorials/pcb-basics
- **Printed Circuit Board (PCB)** is a board with conducting tracks and pads
- Components such as microcontrollers, resistors, capacitors are **mounted** onto the PCB by soldering their pins or leads to the **pads** of the PCB
- **Tracks** on the PCB conduct signals and power to physical devices or components on the board



PCB basics



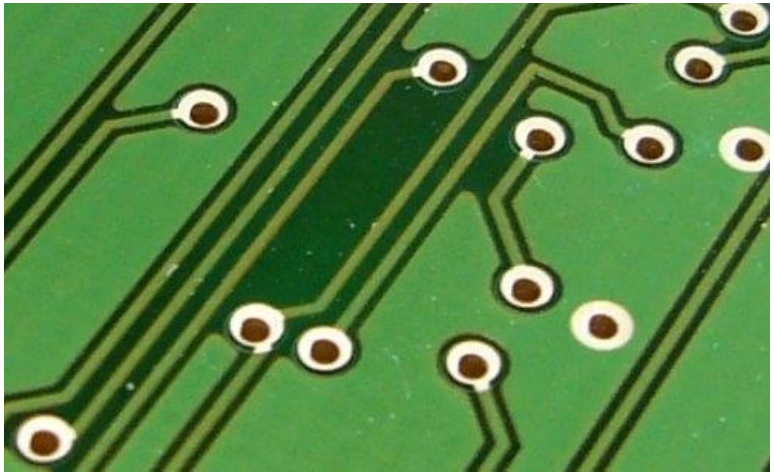


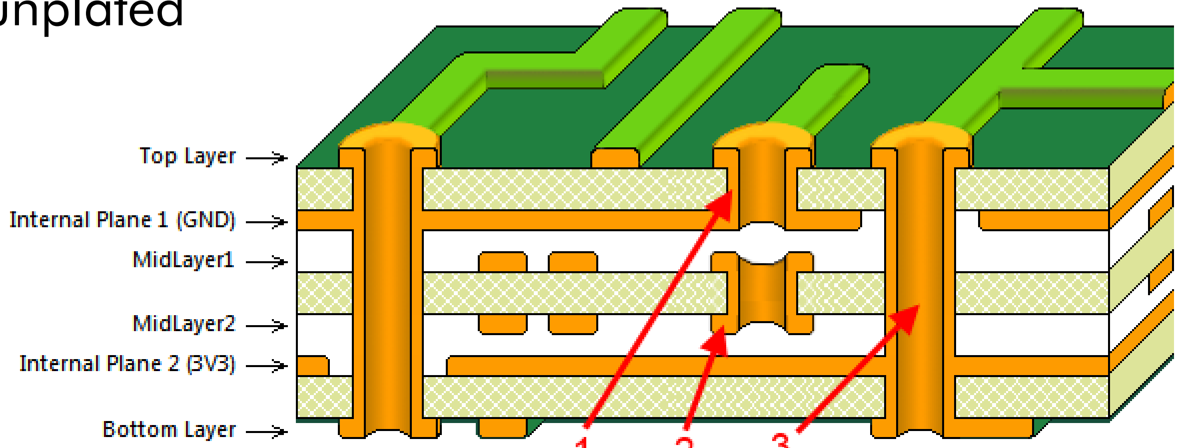
PCB is made in layers:

- **Substrate** is fiberglass (FR4)
- **Copper layer** has all the conducting tracks
- **Soldermask** is a coating that leaves only the pads exposed. Makes soldering easier and prevents accidental contact with other metal parts
- **Silkscreen** is the layer with text, labels and symbols. This layer is there to help with assembly and identification of components

PCB basics

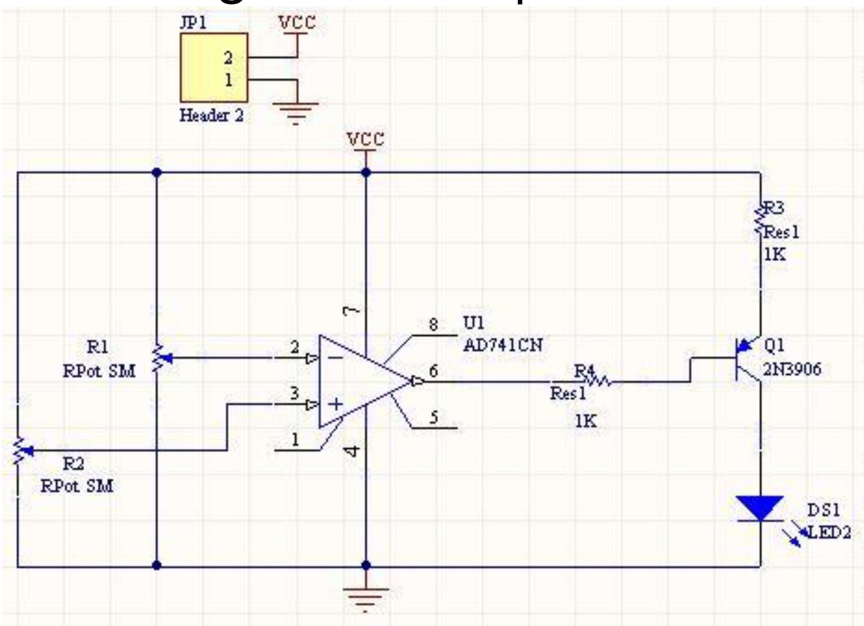
- Number of copper layers vary – complex boards can have up to 16 layers (some even more).
- More layers = more expensive to manufacture.
- Small solder coated holes connect layers vertically – called a **via**
  - Could be plated or unplated

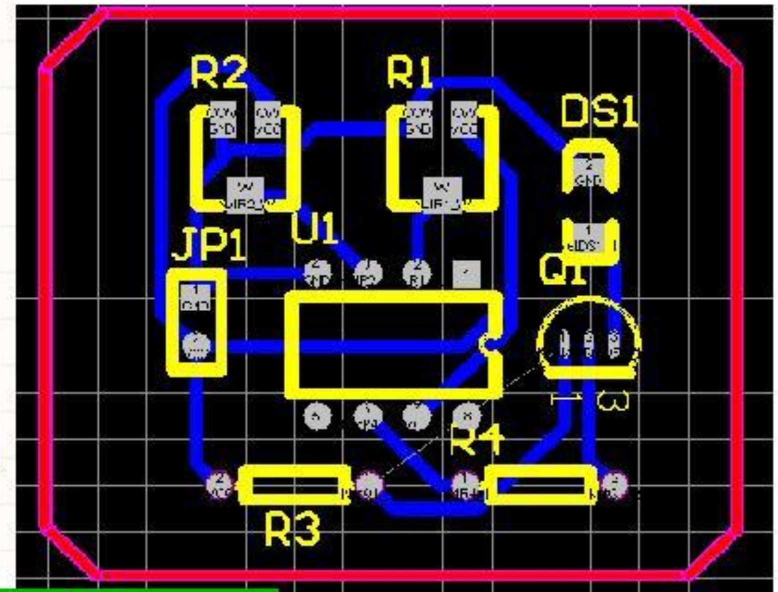


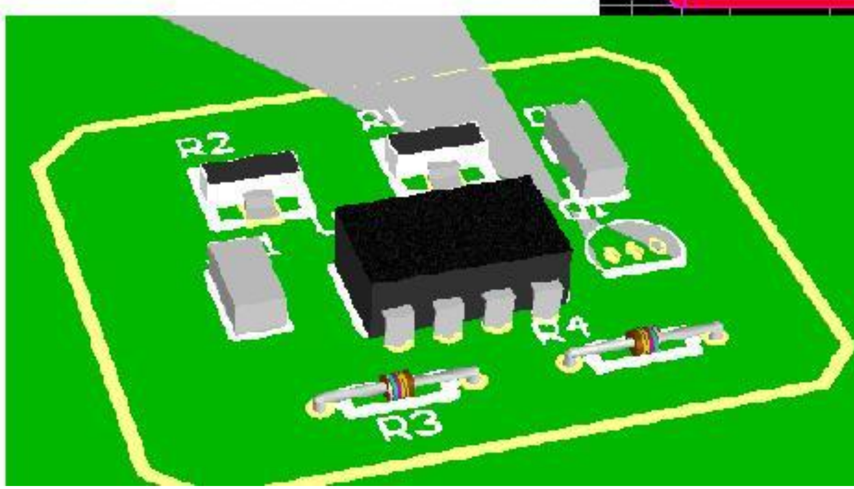


PCB basics

- PCB design has two parts: Schematic design and PCB layout







PCB assembly

- Soldered by hand (few units or a single unit)
- Machine assembled:
  - Solder paste applied to PCB using a stencil (stencil determines where solder should be applied
  - Pick-and-place machine places components on the PCB
  - Reflow oven melts solder and bonds components to the PCB
  - Automated testing
- Assembly often takes place in a cleanroom
- Humidity and temperature controlled
- Electronic components are susceptible to ESD (electrostatic discharge)
- Wear grounding straps

Development board / Ontwikkelingsbord

- Development board for the **STM32F411VET6** microcontroller (STM32F411 Discovery Kit)
- Board and microcontroller manufactured by ST Microelectronics
- In the microcontroller:
  - **ARM Cortex-M4** with floating point unit
  - 512 kb flash memory (program memory)
  - 128 kb SRAM
- Other stuff on the board:
  - 3D MEMS gyroscope, linear accelerometer, magnetic field sensor
  - MEMS microphone, audio digital-to-analog converter (DAC) and amplifier
  - LEDs, pushbuttons