

CC2511 Week 9

Part 1: Types of memory

Part 2: Performing calculations on the microcontroller

What is memory?

- Memory is like one big array of data.
- Each byte has an address.

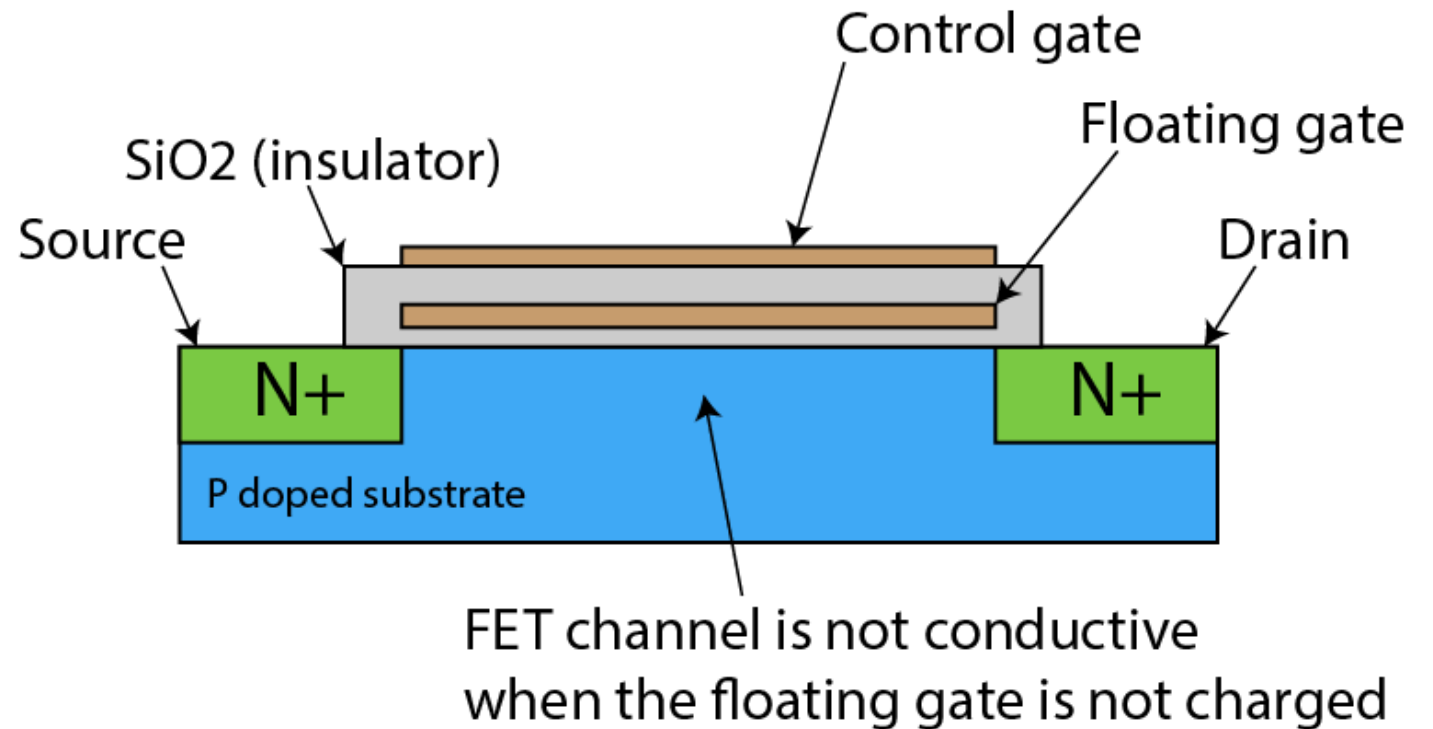
Address	Data
0x0000_0000	
0x0000_0001	
0x0000_0002	
...	
...	
...	
0xFFFF_FFFE	
0xFFFF_FFFF	

Types of memory

- **Volatile** memory requires continuous power to be applied. If the power is removed the stored data is lost.
 - Dynamic Random Access Memory (DRAM)
 - Static Random Access Memory (SRAM)
- **Non-volatile** memory continues to store data even when unpowered.
 - Flash memory
 - Computer hard drives

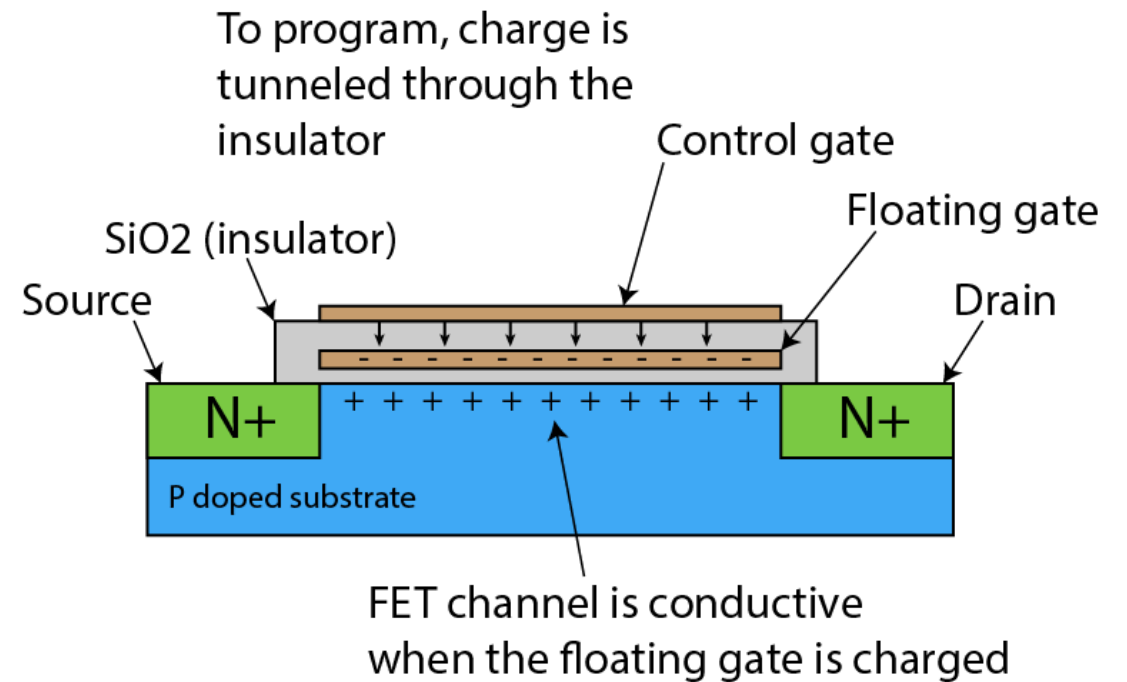
Flash memory

- Flash memory: non-volatile storage used to hold the program code.
- Each bit is physically stored in a transistor-like structure that has a floating, isolated gate.



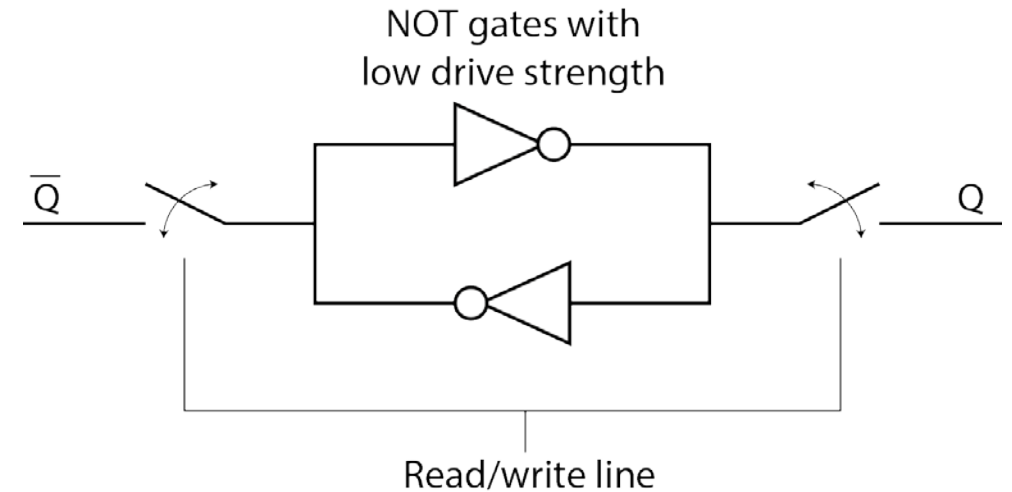
Flash memory

- Programming degrades the dielectric layer. Eventually it fails.
- The charge will remain on the floating gate for a very long time (on the scale of years).
- The presence of the charge alters the threshold voltage of the transistor, enabling the stored bit to be read.



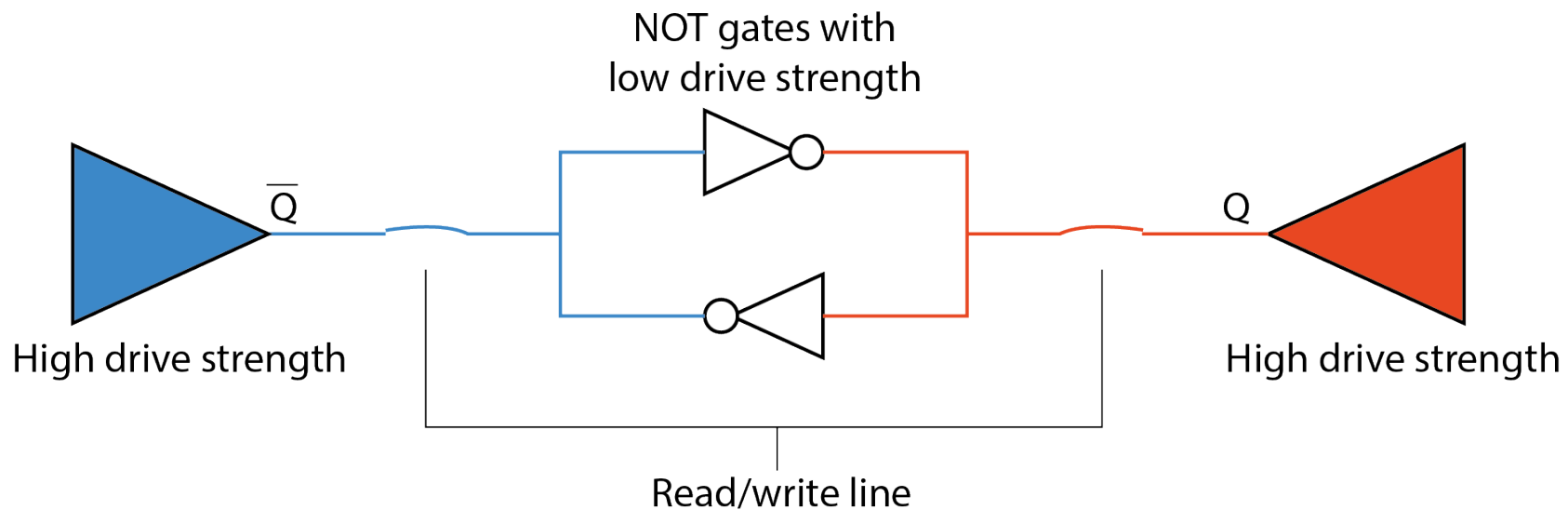
Static Random Access Memory (SRAM)

- Very fast volatile memory that is used to store variables during program execution.
- Stable as long as power is applied.
- Very fast to read and write.
- The back-to-back NOT gates have two stable states: $Q=0$ and $Q=1$.



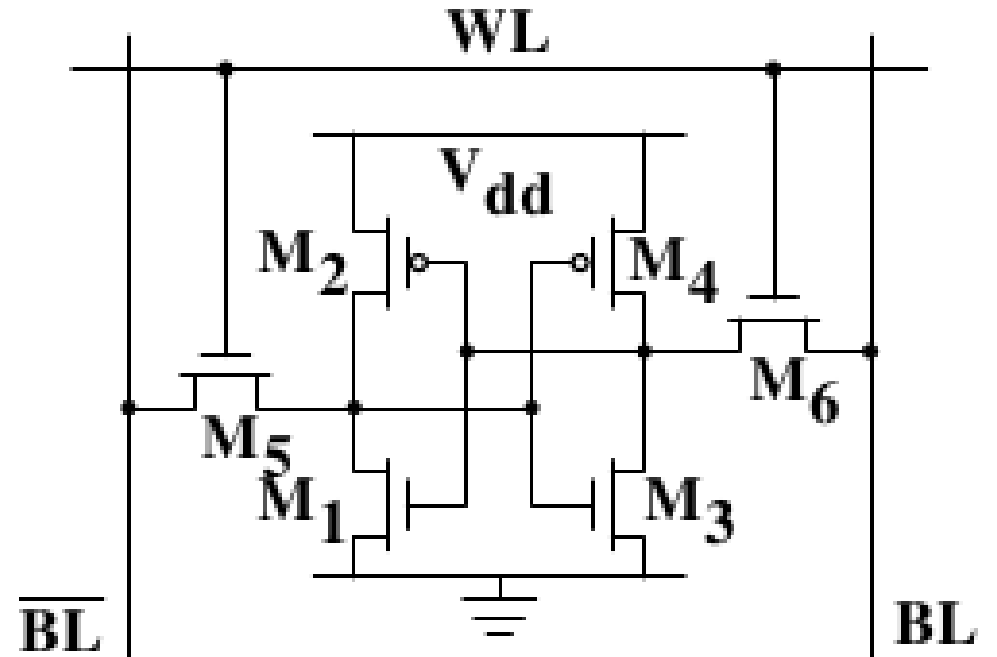
SRAM operation

- To write a new value, strongly drive Q and \overline{Q} to the new values.
- This will overpower the NOT gates and “flip” the cell to the other state.



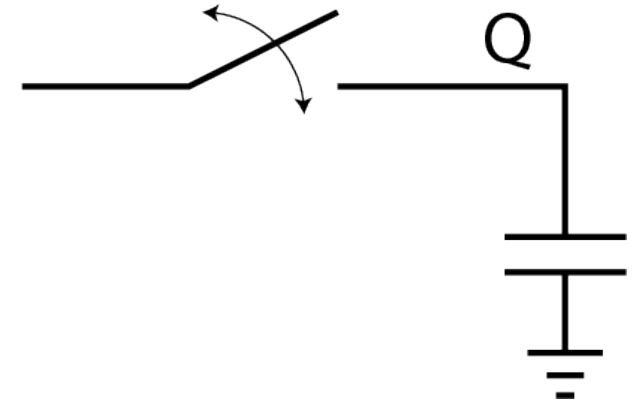
Static Random Access Memory (SRAM)

- Real implementation has 6 transistors (therefore consumes a lot of space).
- Requires constant power.
- Cell state is available immediately when “WL” is raised (therefore extremely fast).



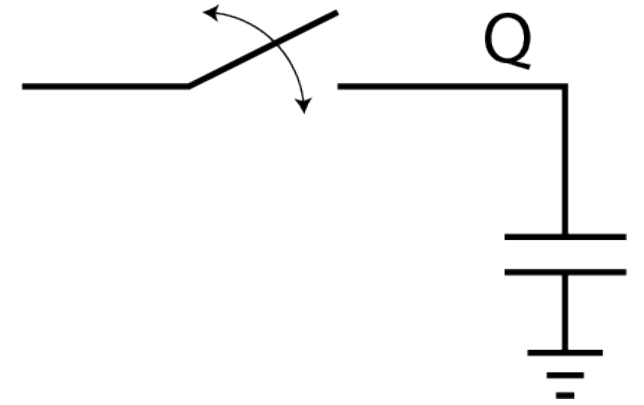
Dynamic Random Access Memory

- Uses a single capacitor per bit.
- Cheaper and smaller than SRAM but slower to read/write
- Time taken to read/write is the time to charge/discharge the capacitor.



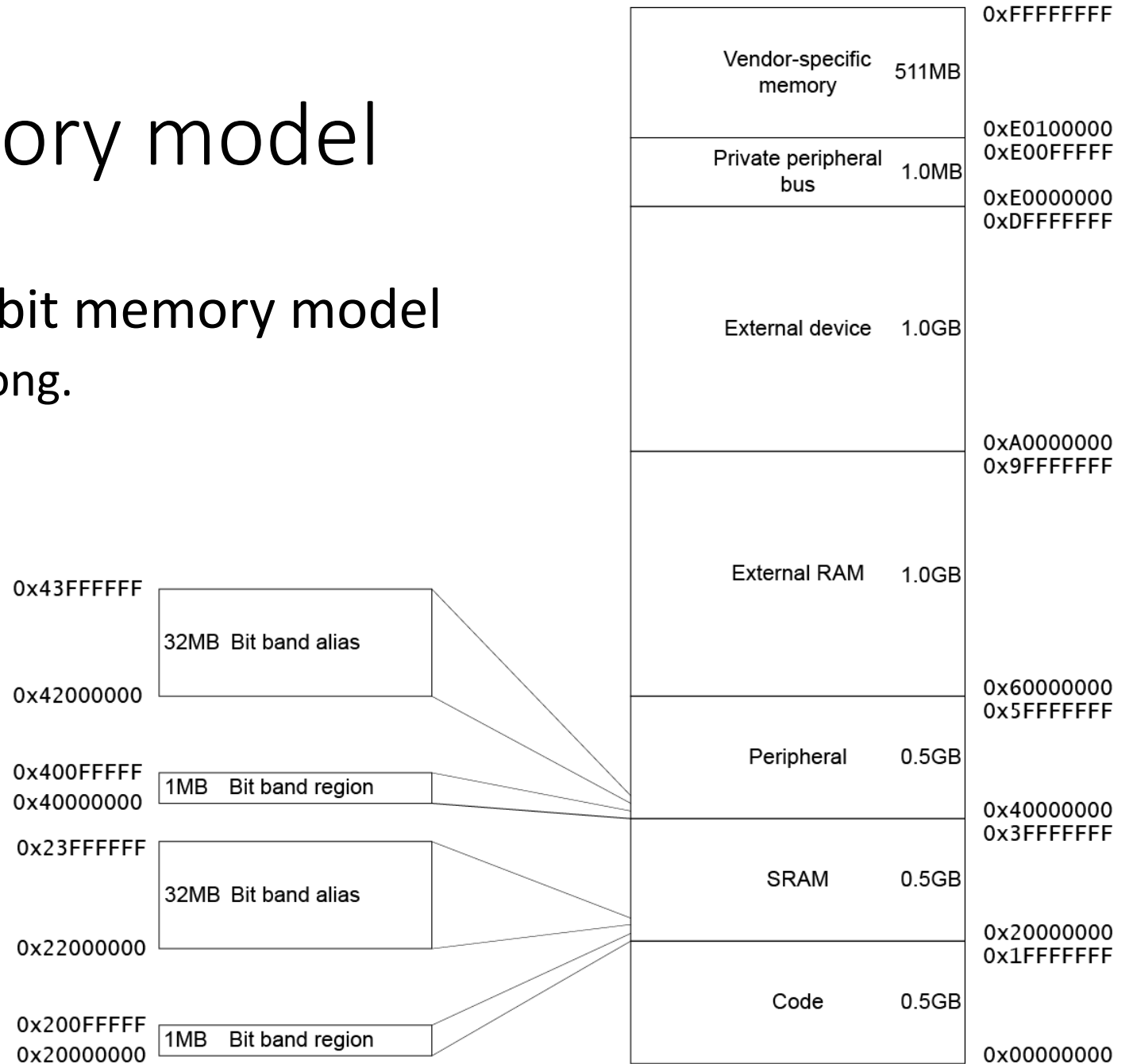
DRAM

- Needs to be frequently “refreshed” by reading the value and then writing it back.
 - Otherwise the capacitor will discharge over time.
- Not found in our microprocessor system, but this makes up the bulk of standard computer memory. (Anything with more than a hundred MB is probably DRAM).



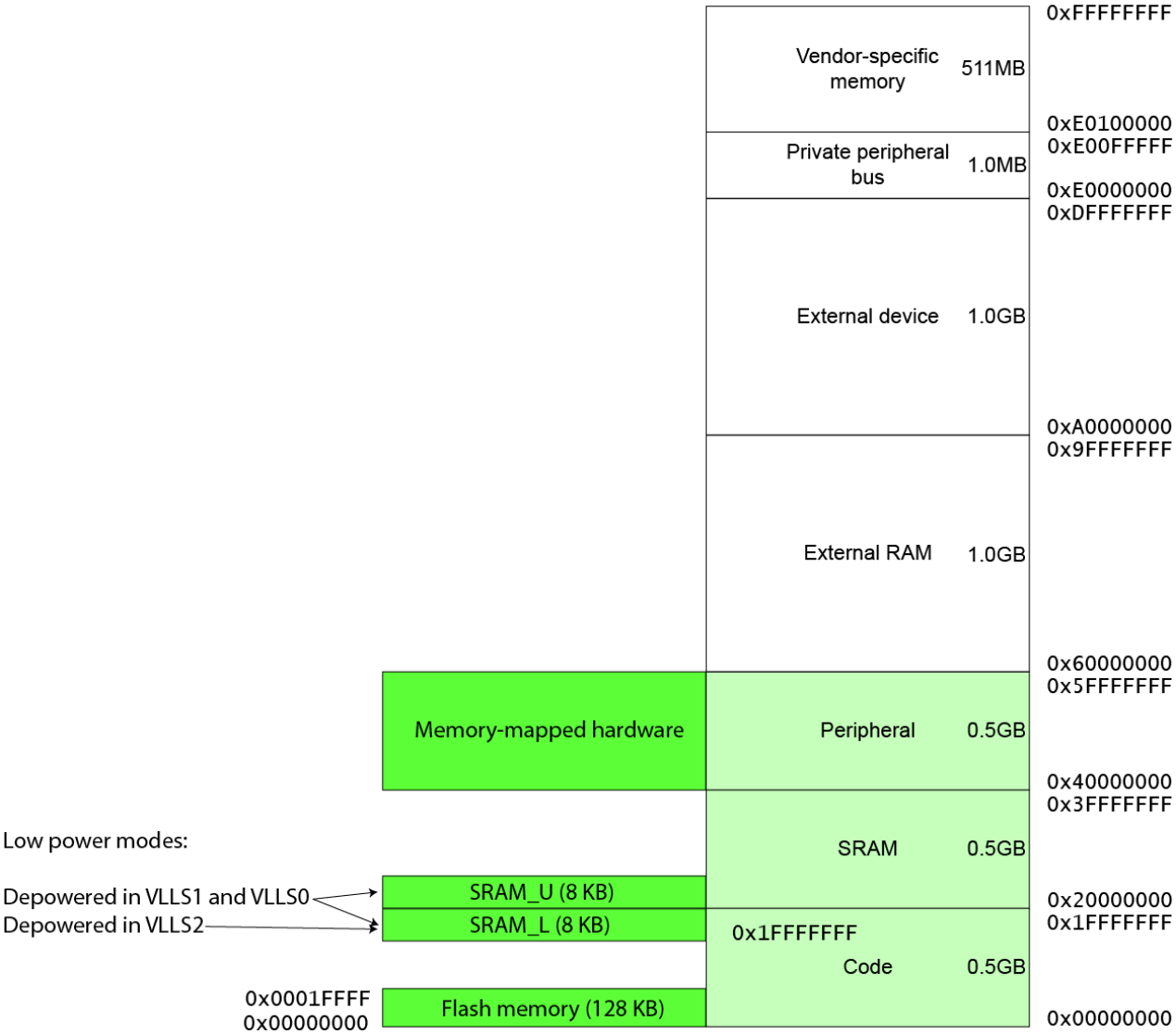
The generic memory model

- ARM Cortex M4 has a 32 bit memory model
 - All addresses are 32 bits long.
- Any given processor implements only part of this memory model.



MK20DX128

Generic ARM
Cortex M4



Memory allocation

- Consider an example:

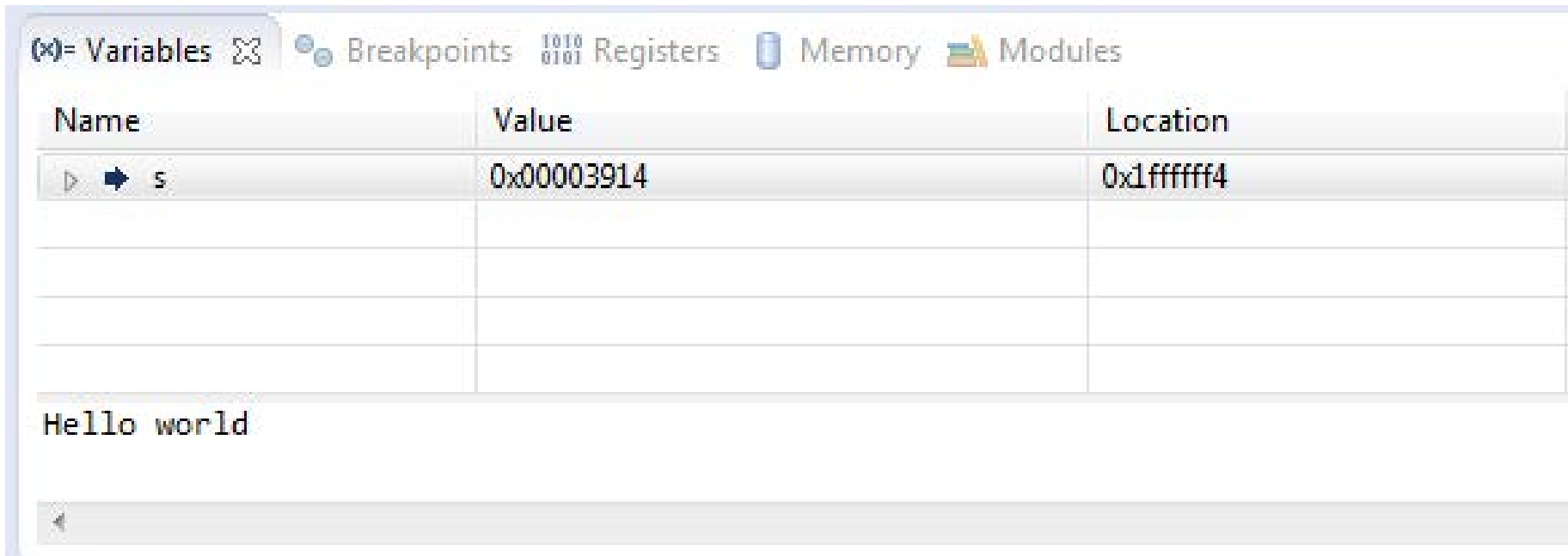
```
char *s = "Hello world";
```

- How does the compiler arrange this in memory?

Memory allocation

```
char *s = "Hello world";
```

- How does the compiler arrange this in memory?



The screenshot shows a debugger interface with a tab labeled "Variables". Below the tab is a table with three columns: "Name", "Value", and "Location". The first row shows a variable named "s" with a value of "0x00003914" and a location of "0x1ffffff4". Below the table, the text "Hello world" is displayed, representing the string stored in memory.

Name	Value	Location
▶ ➡ s	0x00003914	0x1ffffff4

Hello world

Where is the pointer?

Where does the pointer point?

0x= Variables Breakpoints Registers Memory Modules		
Name	Value	Location
▶ s	0x00003914	0x1fffff4
Hello world		

The value of s is 0x0000_3914 (flash).
The location of s is 0x1FFF_FFF4 (RAM).

0x2000_1FFF	SRAM_U (8 KB)
0x2000_0000	
0x1FFF_FFFF	SRAM_L (8 KB)
0x1FFF_E000	
0x0001_FFFF	Flash memory (128 KB)
0x0000_0000	

Constants are stored in flash memory

- Constants are stored in flash memory alongside the program code.
- The precise organisation is determined by the compiler.

Variables are stored in RAM

- Variables are stored in random access memory (RAM).
 - Colloquially programmers will say “memory” when they mean RAM.
- There are three ways that RAM storage is allocated:
 1. Statically (fixed for the life of the program)
 2. On the stack (local to a particular function)
 3. On the heap (managed by the operating system)

Static allocation

- Global variables are statically allocated.
- Place variable declarations **outside** any function, at the “top level” of the source code:

```
uint16 values[10]; // statically allocated  
int main(void)  
{  
    // ...  
}
```

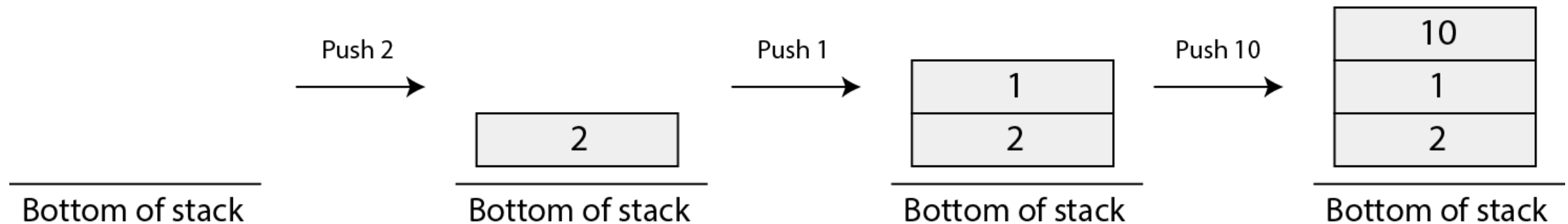
Static allocation

- Variables allocated outside a function last for the lifetime of the program. (They are always available.)

```
uint16 values[10]; // always available
int main(void)
{
    // ...
}
```

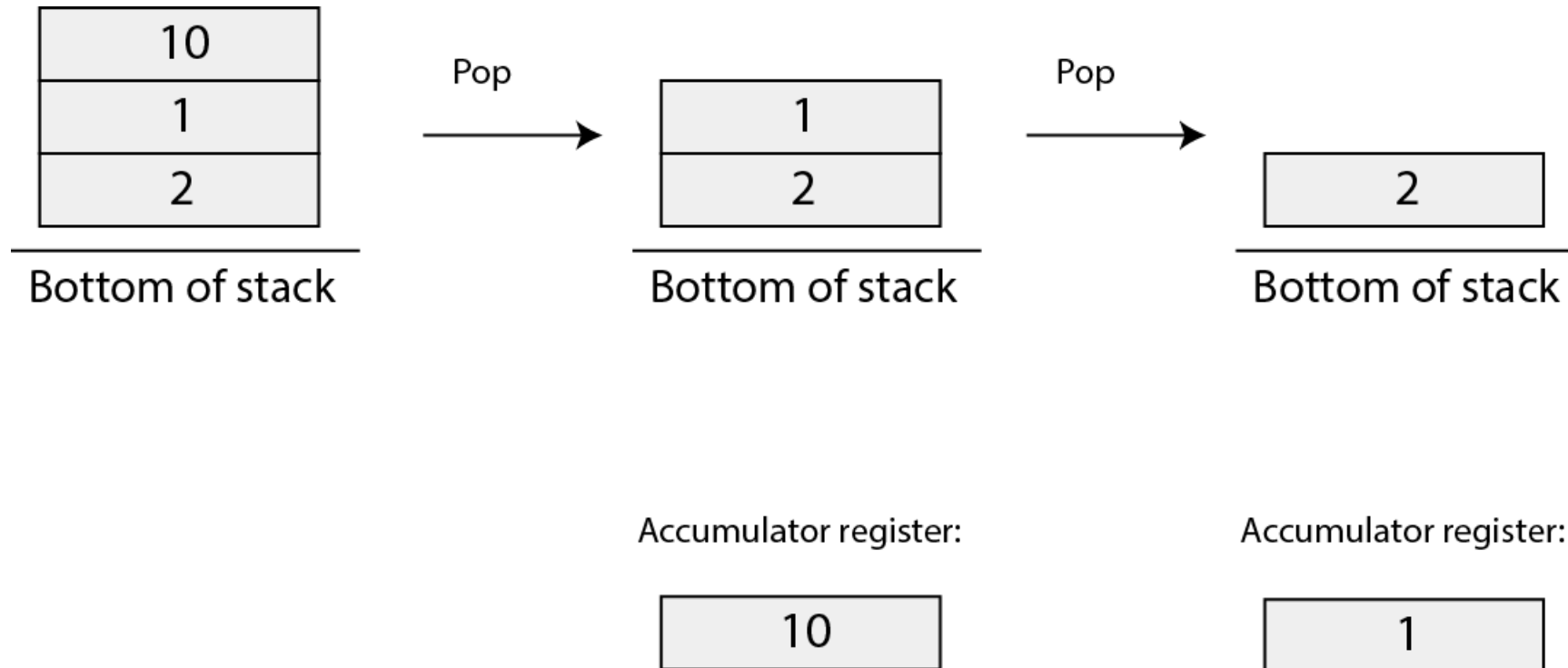
Stacks and the push instruction

- A **stack** is a data structure implemented by almost every CPU.
- It provides a hierarchy where new values are added “on top” of old values.
- Values are added onto the **top of the stack** using a **push** instruction.



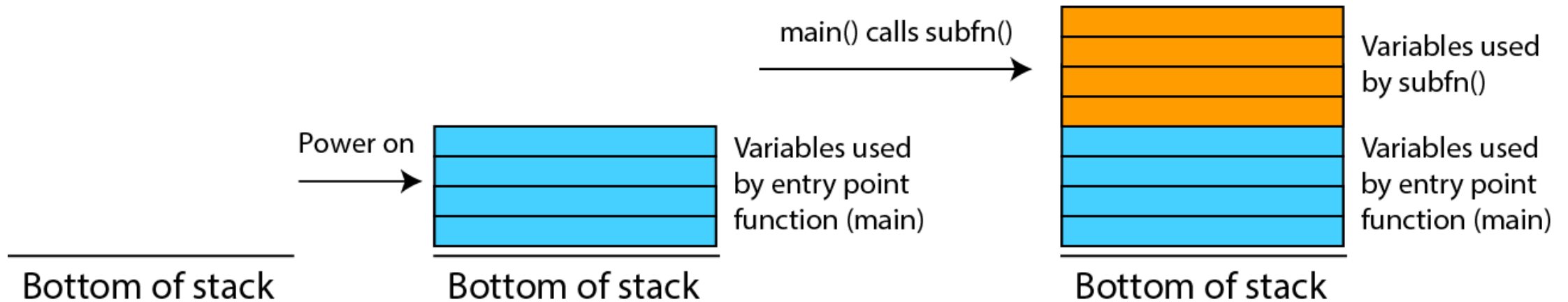
Stack: pop instruction

- Values are removed from the top of the stack using a pop instruction.



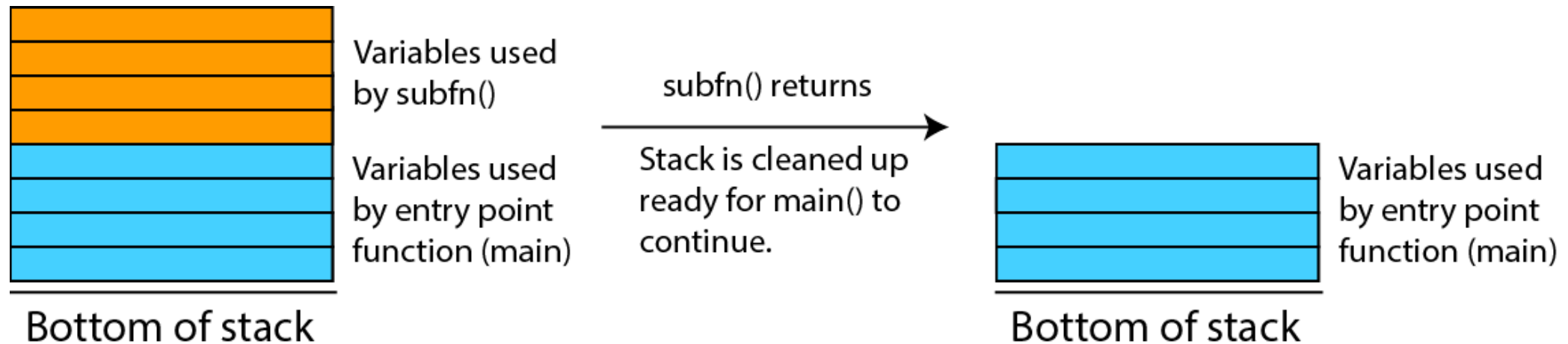
Stack layout for function calls

- Function calls add their own variables to the top of the stack



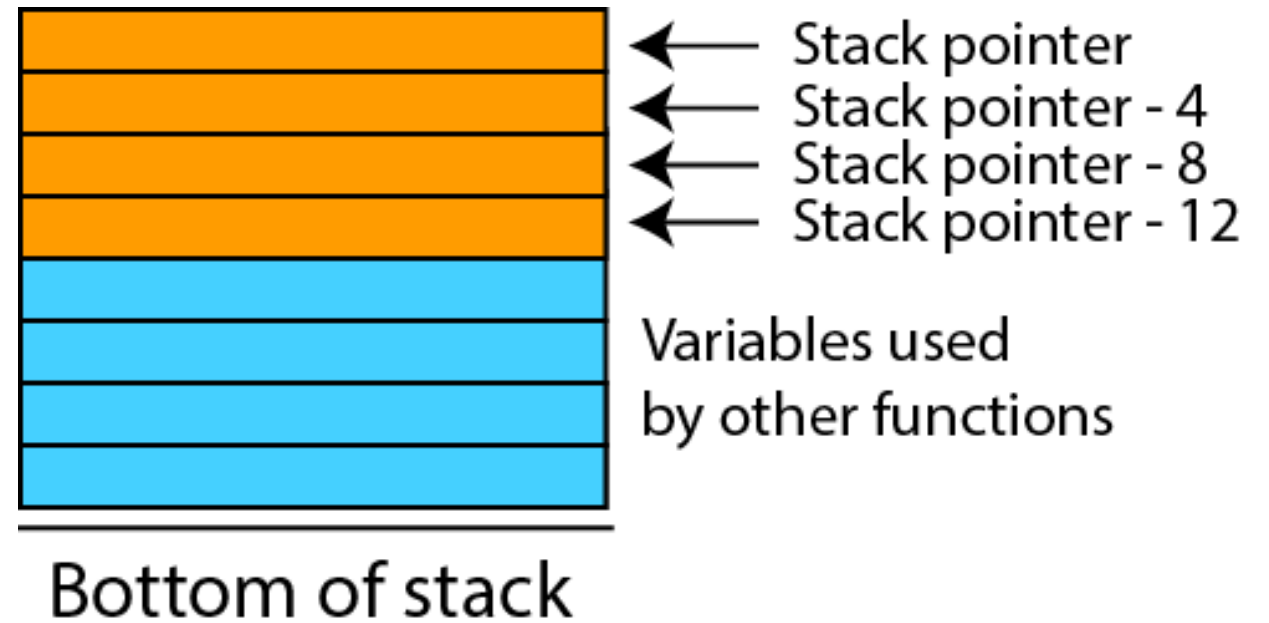
Returning from functions

- When a function returns, it cleans up the stack and removes the items that it placed there.



Random access to stack items

- The stack pointer is maintained by the CPU and adjusted for every PUSH and POP instruction.
- Programs can access items on the stack using **addresses relative to the stack pointer**.



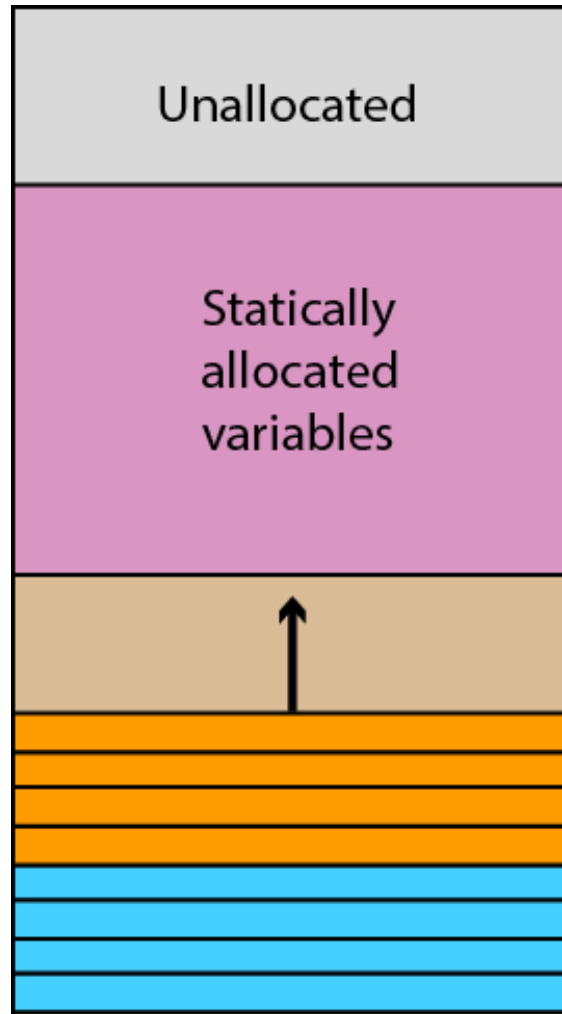
Allocating variables on the stack

- Variables declared **inside functions** are allocated on the stack.

```
int main(void)
{
    uint16 values[10]; // allocated on the stack
    // ...
}
```

Memory layout

- The figure shows how memory might be laid out in a simple system.
- Question: what happens if too many items are placed on the stack?



Example only!

The actual layout in memory is dependent upon the processor, compiler, and any operating system.

For example, often multiple stacks are present for different tasks.

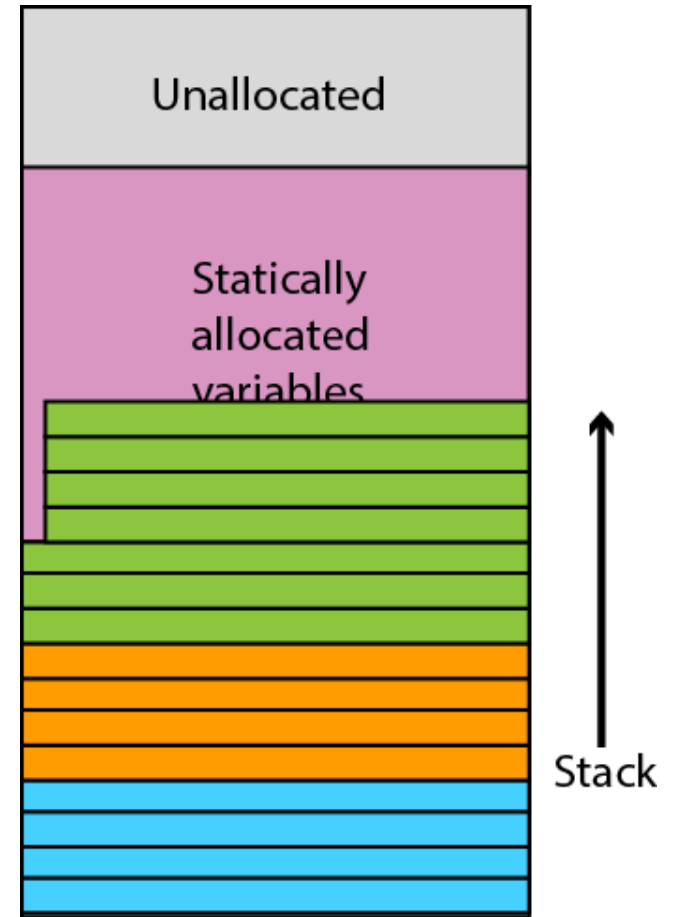
Stack

Size of the stack

- The stack has a limited size.
- Using too much stack space is called a **stack overflow**.

A stack overflow occurs if:

- Your function calls are nested too deeply, or
- Your functions allocate too much memory on the stack.



Allocating big arrays

- Consider:

```
int main(void)
{
    int many_values[10000]; // Wrong!
    // ...
}
```

- This is **broken code** because it's likely that the stack will overflow.

Allocating big arrays

- The stack should only be used for small amounts of memory (small arrays or small numbers of variables).
- The total amount of memory available on the stack is unpredictable
 - It depends on what other functions happened to call this one.
 - Deeply nested function calls might cause stack overflows.
- **Use static allocation for big arrays.**

Dynamic allocation

- Dynamic memory allocation allows program to request memory at run time.
- C functions `malloc` (memory allocate) and `free` are used for dynamic allocation.
- **Generally avoided in embedded systems.**

Part 2: Calculations

- Floating point
- Fixed point

First, a reminder

```
int a = 1;
```

```
int b = 3;
```

```
float c = a/b;
```

- After the code is run, c contains zero!

Integer division

```
int a = 1;
```

```
int b = 3;
```

```
float c = a/b;
```

- After the code is run, c contains zero!
- An integer division is done which always rounds down.

Integer division

Fix by typecasting at least one operand to float or double.

```
int a = 1;
```

```
int b = 3;
```

```
float c = (float)a/b;
```

Floating point operations in embedded systems

- Many microprocessors do not include floating point instructions.
 - The K20DX128 processor lacks floating point instructions.
 - The K22FN512 process has floating point instructions (but only for single precision!)
- If you use floats or doubles on a machine that lacks the instructions, the compiler will insert function calls to code that **emulates floating point in software**.
- This is slow!

Doing math with integers

- Generally, a microcontroller with a floating point unit costs more than one without.
- You might need to do some math but not enough to justify a higher-end CPU.
- What can we do with just integers?

“Fixed point” math

- Integer math is not as limiting as it might first appear.
- We can use integers to implement decimal operations.
- The trick is to introduce an **effective scaling parameter** where the number x is interpreted as x/R .
- Example: if the scaling factor $R = 256$ then 128 is interpreted as $128/256 = 0.5$.
- This is called **fixed point** (as opposed to floating point).

Fixed point math

- All numbers are of the form a/R
- We perform calculations on the numerators a and b which are integers
- The scaling parameter R is implied
- Example:
 - Calculate $2*a$ (using integer math) but interpret it as $2*a/R$

Addition/subtraction in fixed point

- Addition and subtraction automatically work in fixed point if the scaling parameter is the same:

$$\frac{a}{R} + \frac{b}{R} = \frac{a + b}{R}$$

$$\frac{a}{R} - \frac{b}{R} = \frac{a - b}{R}$$

(because addition and subtraction are linear operators)

Multiplication in fixed point

- We calculate $a \times b$ using integer multiplication.
- The answer we want is $\frac{a}{R} \times \frac{b}{R} = \frac{ab}{R^2}$.
- The calculation that we actually do is $a \times b$.
- Our calculation $a \times b$ will be interpreted as ab/R . We are out by a factor of R .
- **Every time we do a multiplication we need to divide by R in order to keep our system consistent.**

Efficiency: choose R as a power of 2

- In binary, multiplication and division by powers of 2 can be performed by shifting bits left or right.
 - In decimal numbers, we multiply/divide by powers of 10 by adding or removing zeroes. This is the same principle.
- Example:

```
int a = 15;
```

```
a = a << 1; // shift left by one place
```

```
// a is now 30
```

Multiplication in fixed point

- Remember, must divide by R after every multiplication.
- Example: Choose $R = 256 = 2^8$

```
int a, b;
```

```
// ... input values for a and b ...
```

```
int c = (a * b) >> 8;
```

```
// Shift right by 8 is equivalent to dividing by  $2^8$ 
```

```
// c now contains  $a*b$  in our fixed point system
```

Multiplication overflow in fixed point

- Consider a fixed point system with 8 bit integers and $R=256$.
- Calculate $0.9*0.9$ in this system.

```
uint8_t a = (uint8_t)(0.9*256); // = 230
```

```
uint8_t b = (a * a) >> 8;
```

- $230*230=52900$ cannot fit in an 8 bit integer.
- After the left shift, $52900/256 = 206$ does fit in an 8 bit integer.
- **Need to use temporary variables during the calculation that are larger than the final size.**

Multiplication overflow

```
uint8_t b = (a * a) >> 8;
```

- On a 32 bit machine (like our microprocessors), the arithmetic instructions operate on 32 bit values.
- The C compiler is permitted to use 32 bit temporary values during this calculation. However, this is implementation-dependent.
 - If the CPU has an 8 bit multiply instruction then the compiler will probably use that instead.

Explicit typecasting

- Can explicitly cast to a larger size integer to force the compiler to use a wider multiplication instruction:

```
uint8_t b = ((uint32_t)a * (uint32_t)a) >> 8;
```

Example: fixed point trigonometry

- Consider a fixed point system that represents the range $[0, 1)$.
- **How would you implement $\sin x$ in this system?**
- Example application: you're reading accelerometer data and need trigonometric functions to work with the resultant force vectors. A rough approximation is fine.

Example: fixed point trigonometry

- Example: write a C function that implements $\sin x$ (in radians) by its Taylor series expansion

$$\sin x \approx x - \frac{x^3}{6}$$

- This is a reasonable approximation for small x .
- Use 16 bit fixed point numbers with a scaling factor $R = 2^{16} = 65536$.

Solution

```
uint16_t sin(uint16_t x)
{
    // Compute  $\sin(x) = x - x^3/6$ 
    uint32_t y = (uint32_t)x; // use 32 bit multiply
    return y - (((y * y) >> 16) * y) >> 16) / 6;
}
```


Accuracy of the solution

x	Exact	Fixed point approximation
0.000000	0.000000	0.000000
0.100000	0.099833	0.099840
0.200000	0.198669	0.198672
0.300000	0.295520	0.295506
0.400000	0.389418	0.389349
0.500000	0.479426	0.479179
0.600000	0.564642	0.564004
0.700000	0.644218	0.642847
0.800000	0.717356	0.714672
0.900000	0.783327	0.778515

—

Summary

- Non-volatile memory such as flash maintains its data when power is removed.
- Volatile memory such as SRAM and DRAM requires the constant application of power, but is typically much faster than flash storage.
- Variables used inside a function are allocated on the stack.
- Global variables are statically allocated.
- Floating point numbers can sometimes be replaced with “fixed point” numbers where an implied $/R$ is used to represent fractions.