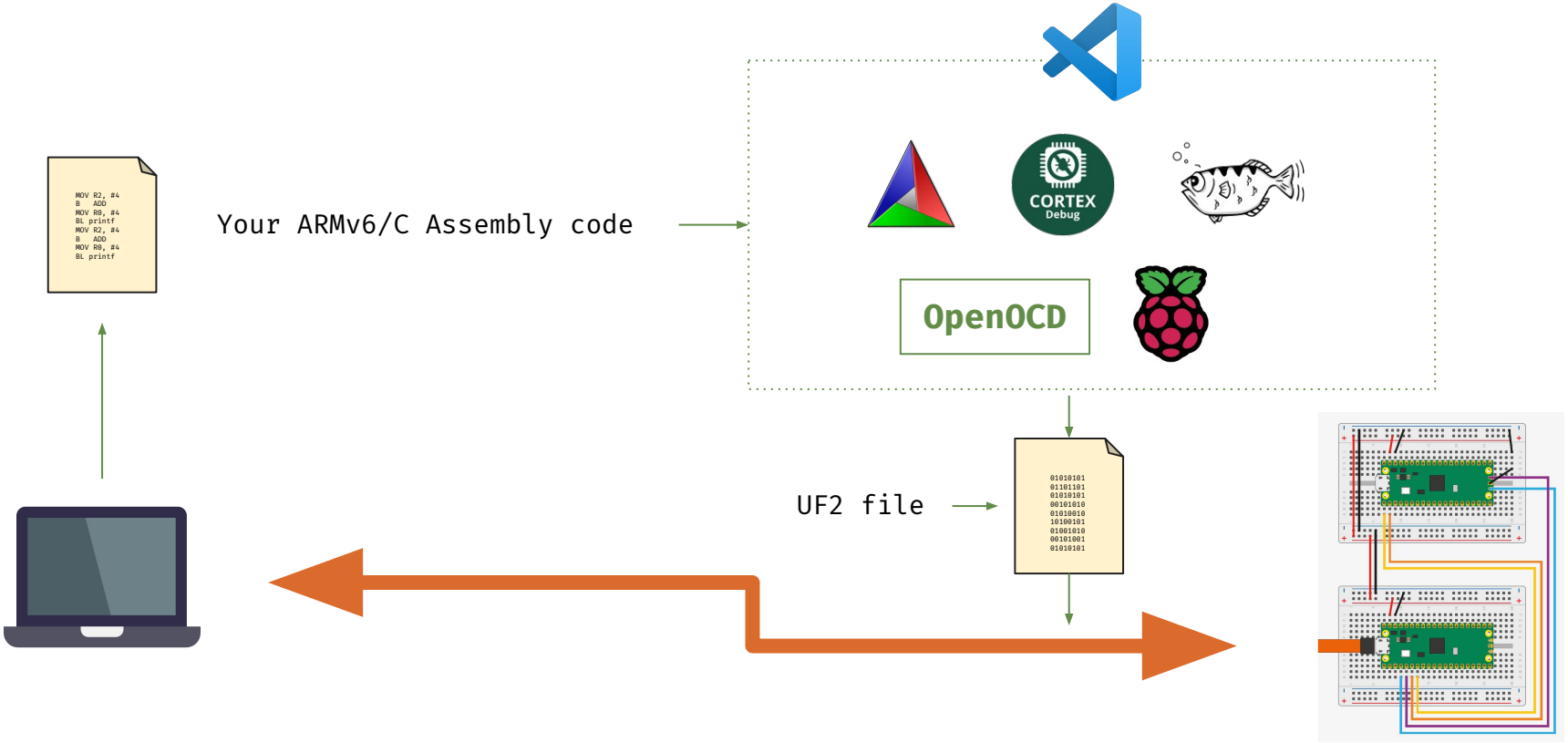
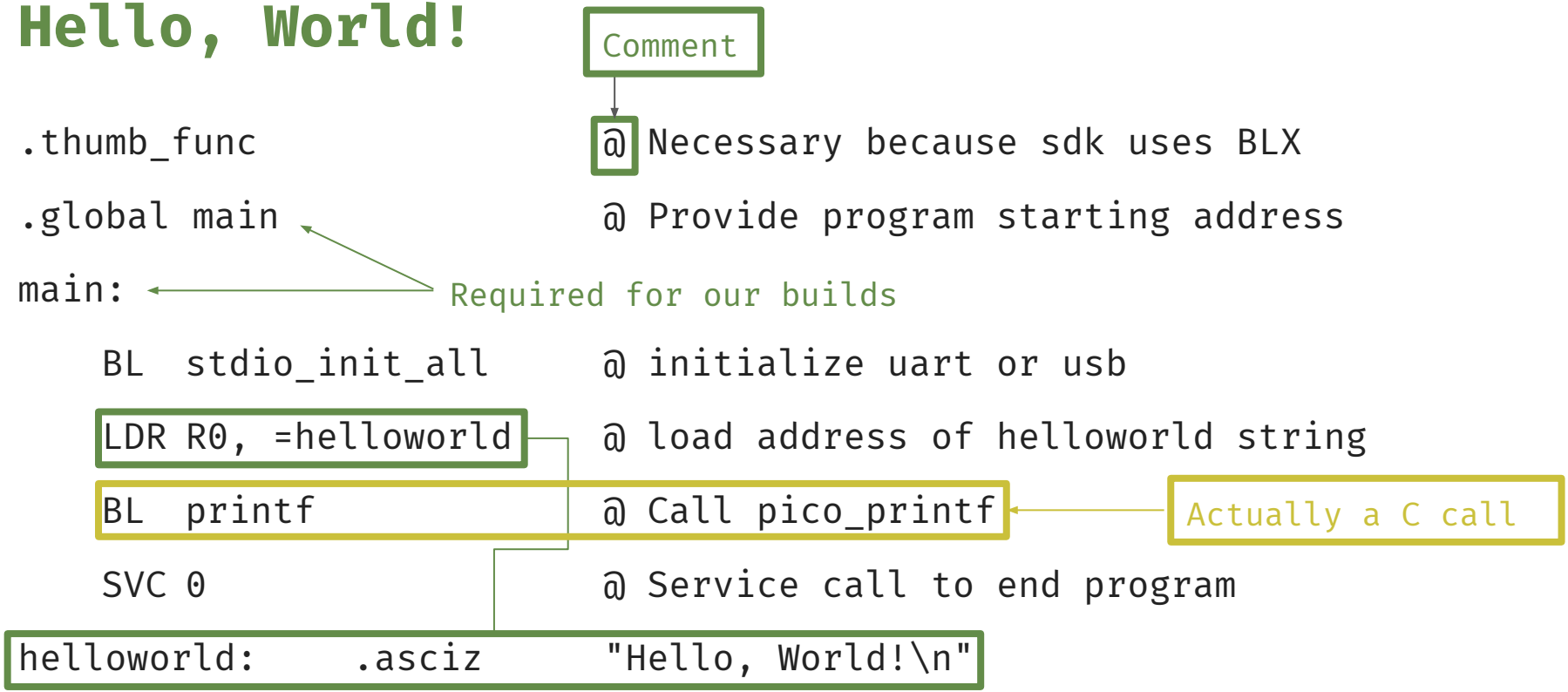

0X BARE
CMPSC 0XC8 ALLEGHENY COLLEGE 0X7E7
C8 METAL

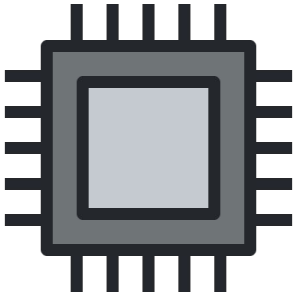


🛠️ ARM Cross-Compiler Toolchain

Hello, World!

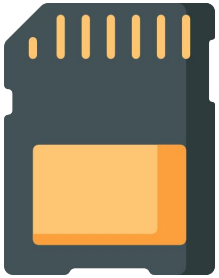

Comment
.thumb_func @ Necessary because sdk uses BLX
.global main @ Provide program starting address
main:
Required for our builds
BL stdio_init_all @ initialize uart or usb
LDR R0, =helloworld @ load address of helloworld string
BL printf @ Call pico_printf
Actually a C call
SVC 0 @ Service call to end program
helloworld: .asciz "Hello, World!\n"

Registers vs. Memory



REGISTERS

- On the processor
  - But *are not* the processor
- Called by r[0-12]
- Operate on data
- Limited to 32 bytes of instruction and/or storage



MEMORY

- Outside of processor
- Called by mnemonics like 0x123f
- Cannot be operated *on*
  - Can only *store*

ARMv6 Assembly opcodes

```
mov     r0, #2          @ moves the value 2 into register 0
mov     r1, r0          @ copies the value 2 from register 0 to register 1
                        @ register 0 is still equal to 2
```

```
add     r0, r1, r2      @ Add r1 and r2 and store in r0
add     r1, r2          @ Add r1 and r2 and store in r1
```

```
mov         r0, #'A'        @ load the value of A into r0
ldr         r1, =outstr     @ load address of outstr into r1
strb        r0, [r1]        @ store the first byte in r0 into the address
                            @ starting at r1
strb        r0, r1          @ Does not work
```

ARMv6 Assembly opcodes

OPCODE R_M , R_N , R_D

ADD R5 , R3 , R2

R5 = R3 + R2

OPCODE R_M , I_{MM}

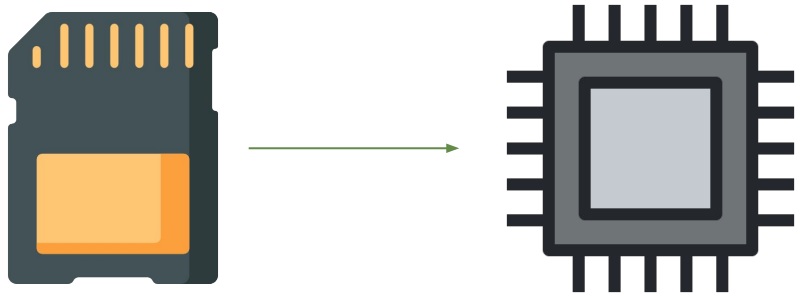
MOV R5 , #200

R5 ← 200_{10}

STRB vs LDR

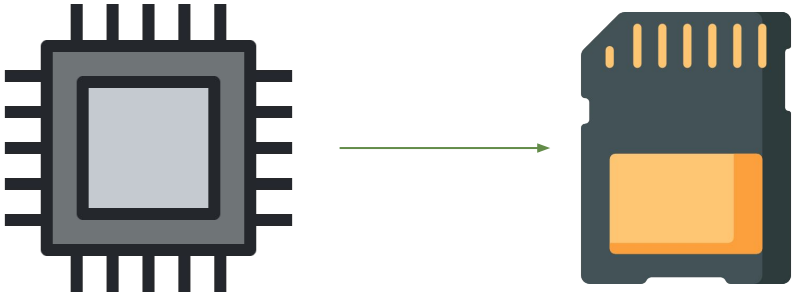
LDR

Load contents of memory into register

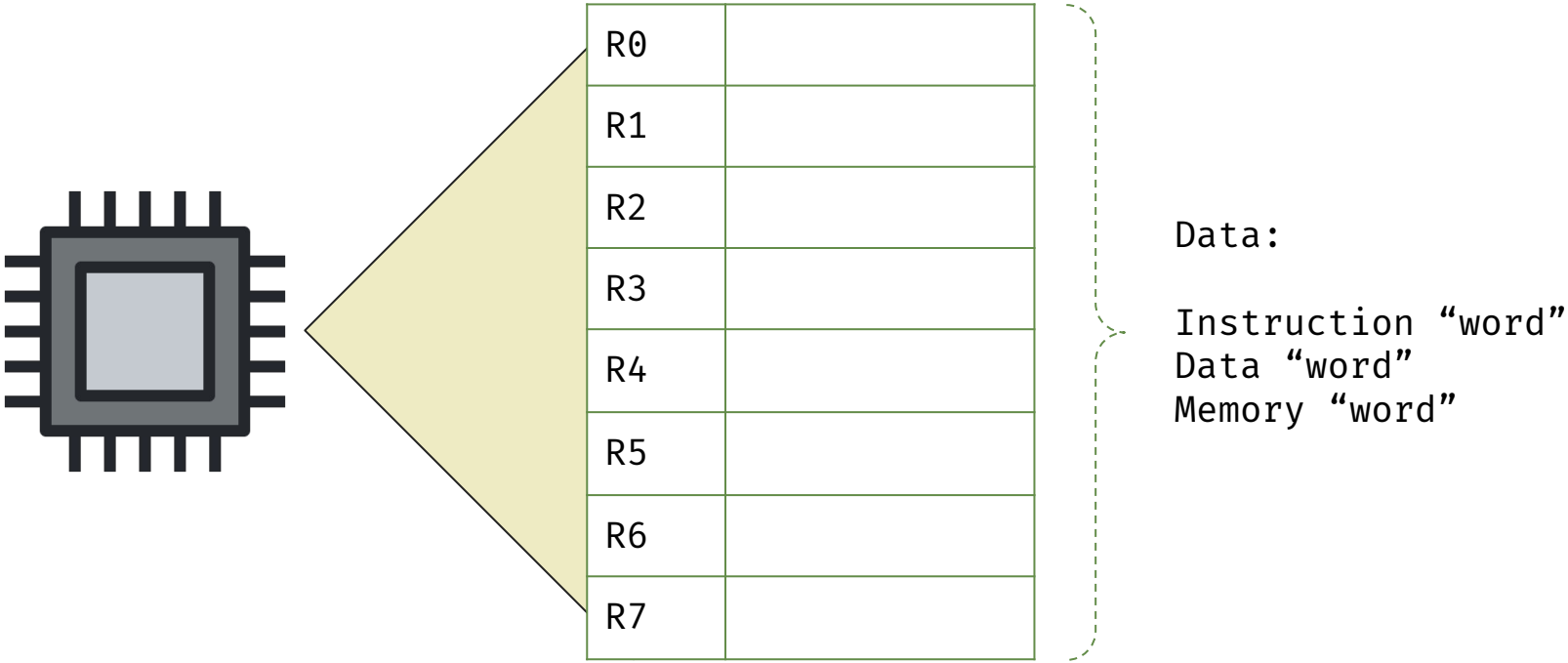


STRB

Store contents of register in memory



Registers as “parking lots”



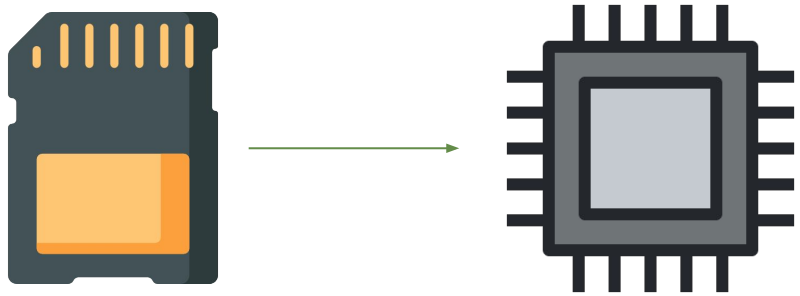
STRB vs./& LDRB

Done by “indirect addressing”:
LDRB R5, [R1]

Represents a memory location

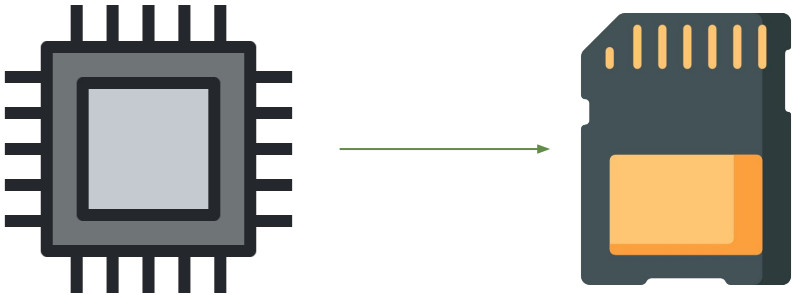
LDRB

Load contents of memory into register (single byte)



STRB

Store contents of register in memory (single byte)



ARMv6 Assembly opcodes

(If there's an ADD…)

OPCODE R_M , R_N , R_D

SUB R5 , R3 , R2

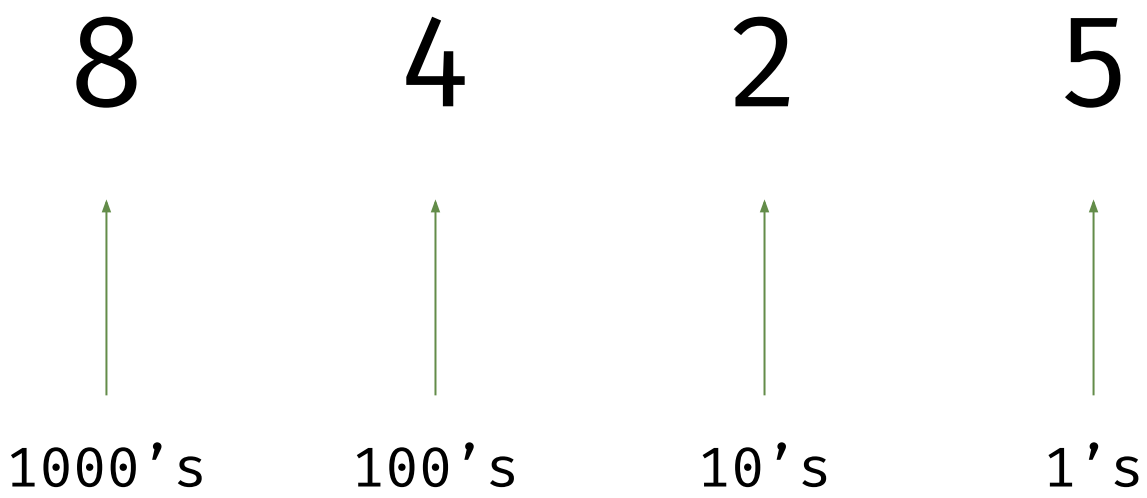
R5 = R3 + R2

OPCODE R_M , I_{MM}

SUB R5 , #200

R5 ← 200_{10}

All your base



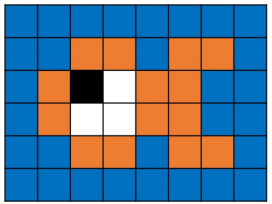
Base 10

All your base

Actual bits (binary)

Binary Interpretation:

00	White	01	Orange
10	Blue	11	Black



(a)

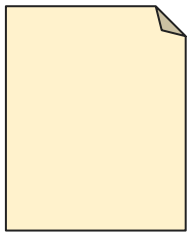
10	10	10	10	10	10	10	10
10	10	01	01	10	01	01	10
10	01	11	00	01	01	10	10
10	01	00	00	01	01	10	10
10	10	01	01	10	01	01	10
10	10	10	10	10	10	10	10

(b)

10101010	10101010
10100101	10010110
10011100	01011010
10010000	01011010
10100101	10010110
10101010	10101010

(c)

0xa24d9100	0xa24d91a1
0xa24d9111	0xa24d91bb
0xa24d913f	0xa24d91c9
0xa24d914b	0xa24d91aa
0xa24d917a	0xa24d91e0
0xa24d9199	0xa24d91ef



gone_fishin.png

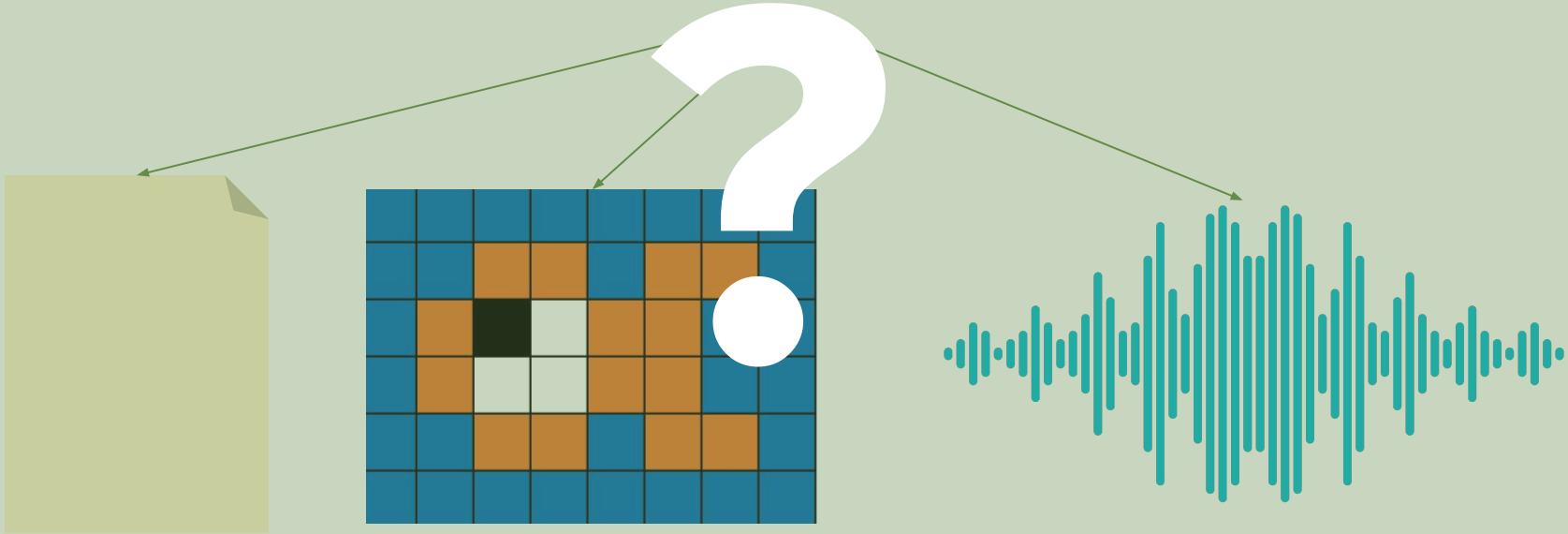
File permissions (octal)

rwx r-- r--
7 5 5

Memory locations (hexadecimal)

Data representation

…0000011001011101111101010110010010100100001001…



A bit? Byte? Nibble?

Short for
“binary digit”

Bit	Single digit	0, 1
Nibble	Two digits	01, 00, 10, 11
Byte	8 bits	01011001

All your base

Base									
Decimal	0	1	2	3	4	8	10	16	31
Binary	0	01	10	11	100	1000	1010	10000	11111
Octal	0	1	2	3	4	10	12	20	37
Hexadecimal	0	1	2	3	4	8	A	10	1F

All your base

Base			
Decimal	64	159	318
Binary			
Octal			
Hexadecimal			

All your base

Base			
Decimal	64	159	318
Binary	1000000		
Octal	100		
Hexadecimal	40		

All your base

Base			
Decimal	64	159	318
Binary	1000000	10011111	
Octal	100	237	
Hexadecimal	40	9F	

All your base

Base			
Decimal	64	159	318
Binary	1000000	10011111	100111110
Octal	100	237	400
Hexadecimal	40	9F	13E

DEC2BIN

200	Even	0

200_{10} → $?_{2}$

DEC2BIN

200	Even	0
200 ÷ 2	Even	0

200_{10} $?_{2}$

DEC2BIN

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0

200_{10} → $?_{2}$

DEC2BIN

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0
50 ÷ 2	Odd	1

200_{10} $?_{2}$

DEC2BIN

200_{10} → $?_{2}$

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0
50 ÷ 2	Odd	1
25 ÷ 2	Even	0

DEC2BIN

200_{10} → $?_{2}$

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0
50 ÷ 2	Odd	1
25 ÷ 2	Even	0
12 ÷ 2	Even	0

DEC2BIN

200_{10} → $?_{2}$

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0
50 ÷ 2	Odd	1
25 ÷ 2	Even	0
12 ÷ 2	Even	0
6 ÷ 2	Odd	1

DEC2BIN

200_{10} → $?_{2}$

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0
50 ÷ 2	Odd	1
25 ÷ 2	Even	0
12 ÷ 2	Even	0
6 ÷ 2	Odd	1
3 ÷ 2	Odd	1

DEC2BIN

200_{10} → $?_{2}$

200	Even	0
200 ÷ 2	Even	0
100 ÷ 2	Even	0
50 ÷ 2	Odd	1
25 ÷ 2	Even	0
12 ÷ 2	Even	0
6 ÷ 2	Odd	1
3 ÷ 2	Odd	1
1 ÷2	No carry	0

DEC2BIN

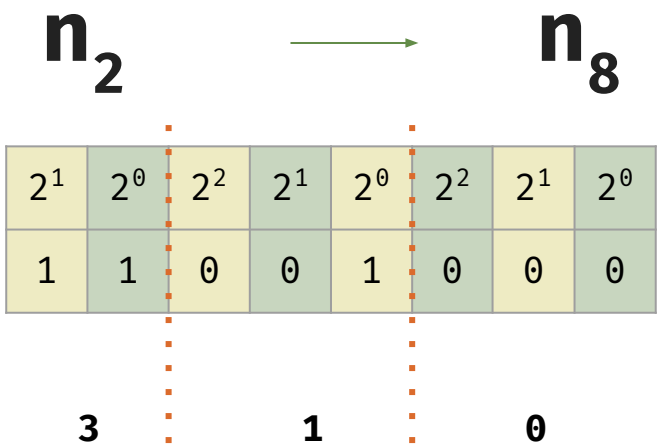
2^0	200	Even	0
2^1	200 ÷ 2	Even	0
2^2	100 ÷ 2	Even	0
2^3	50 ÷ 2	Odd	1
2^4	25 ÷ 2	Even	0
2^5	12 ÷ 2	Even	0
2^6	6 ÷ 2	Odd	1
2^7	3 ÷ 2	Odd	1
2^8	1 ÷2	No carry	0

200_{10} → $?_2$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	0	0

11001000 is an **8 bit** number.

BIN2OCT



DEC2BIN

Try your hand at converting a few numbers into their 8-bit equivalents.

66
123
253
491

DEC2BIN

Try your hand at converting a few numbers into their 8-bit equivalents.

66	01000010
123	
253	
491	

DEC2BIN

Try your hand at converting a few numbers into their 8-bit equivalents.

66	01000010
123	01111011
253	
491	

DEC2BIN

Try your hand at converting a few numbers into their 8-bit equivalents.

66	01000010
123	01111011
253	11111101
491	

DEC2BIN

Try your hand at converting a few numbers into their 8-bit equivalents.

66	01000010
123	01111011
253	11111101
491	NOPE, NOT IN 8 BITS

Using our imagination…

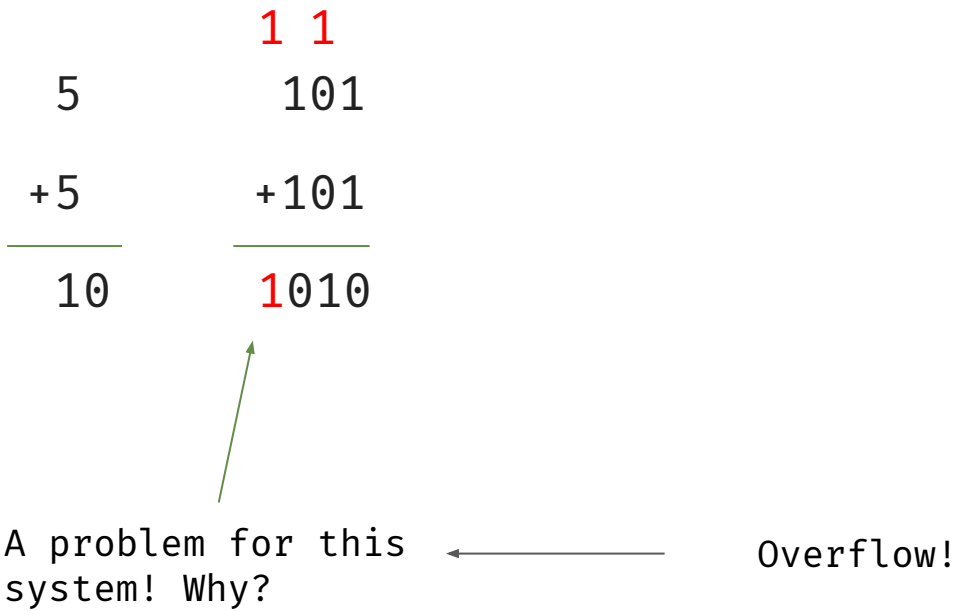
For the following math, we’re going to imagine that:

We are using a system which can store only 3 bit numbers

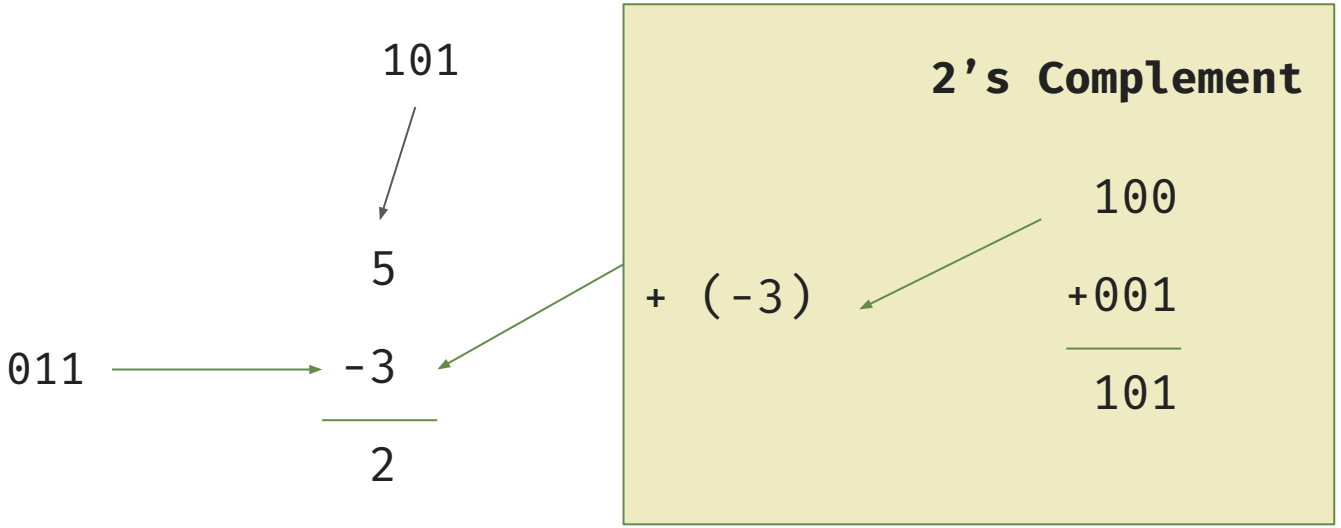
Binary Arithmetic

	11
5	101
+3	+011
8	111

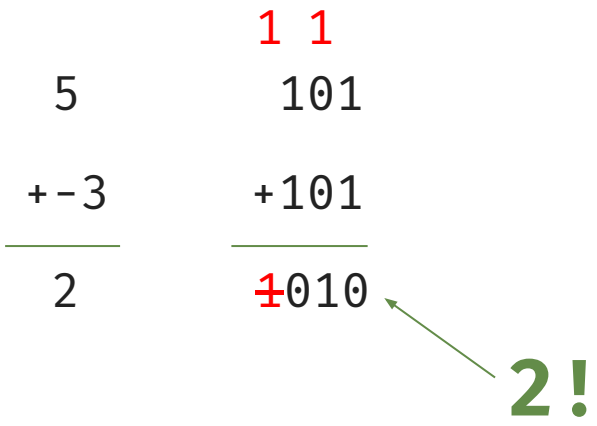
Binary Arithmetic



Binary Arithmetic



Binary Arithmetic



Data size directives (Cortex M0+)

Byte	.byte	8-bit values
Half word	.hword	16-bit values
Word	.word	32-bit values
Double word	.dword	64-bit values

Data size directives

Python	`a = 5`
C	`int a = 5;`
Assembly	`a:  .word  5`

Each of these “reserve” 4 bytes