

# ARM Cross-Compiler Toolchain

# Hello, World!

```
.thumb_func
```

```
.global main
```

```
main:
```

```
BL  stdio_init_all
```

```
LDR R0, =helloworld
```

```
BL  printf
```

```
SVC 0
```

```
helloworld:  .asciz  "Hello, World!\n"
```

Comment

@

Necessary because sdk uses BLX

@ Provide program starting address

Required for our builds

@ initialize uart or usb

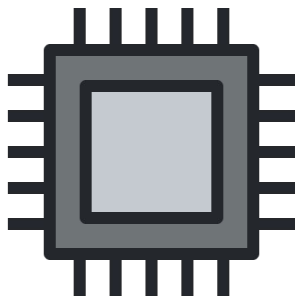
@ load address of helloworld string

@ Call pico\_printf

Actually a C call

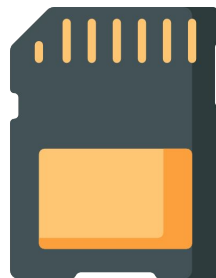
@ Service call to end program

# 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       $R_M$  ,  $I_{MM}$

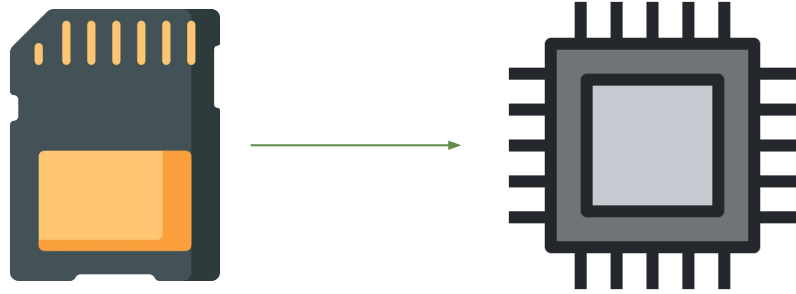
MOV              R5 , #200

$R5 \leftarrow 200_{10}$

# STRB vs LDR

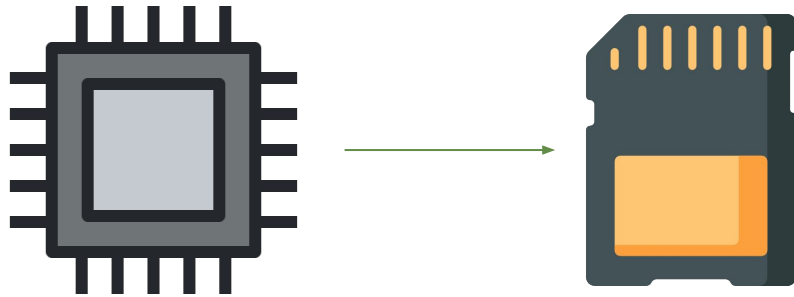
## LDR

Load contents of  
memory into register

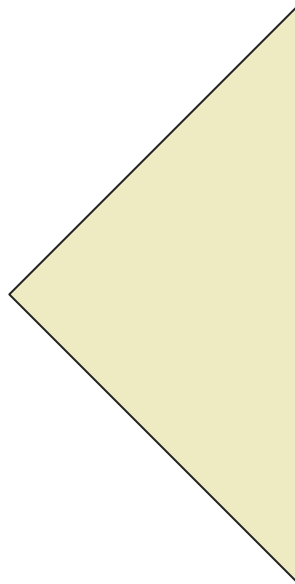
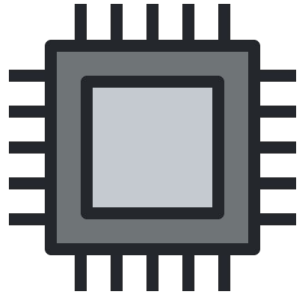


## STRB

Store contents of  
register in memory



# Registers as “parking lots”



R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	

Data:

Instruction “word”  
Data “word”  
Memory “word”



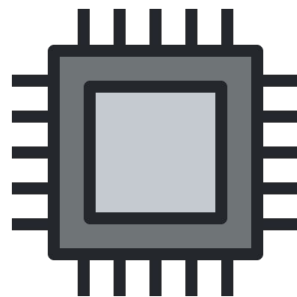
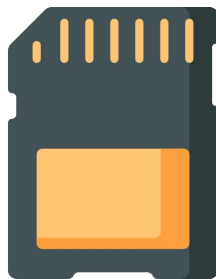
# STRB vs. /& LDRB

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

Represents a  
memory location

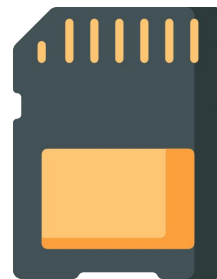
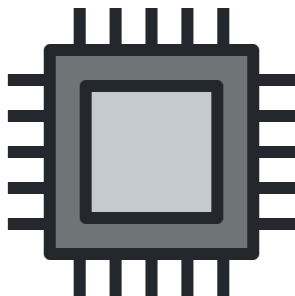
## LDRB

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

$R5 \leftarrow 200_{10}$

All your base

8

4

2

5



1000's

100's

10's

1's

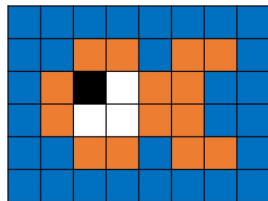
Base 10

# 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

File permissions  
(octal)

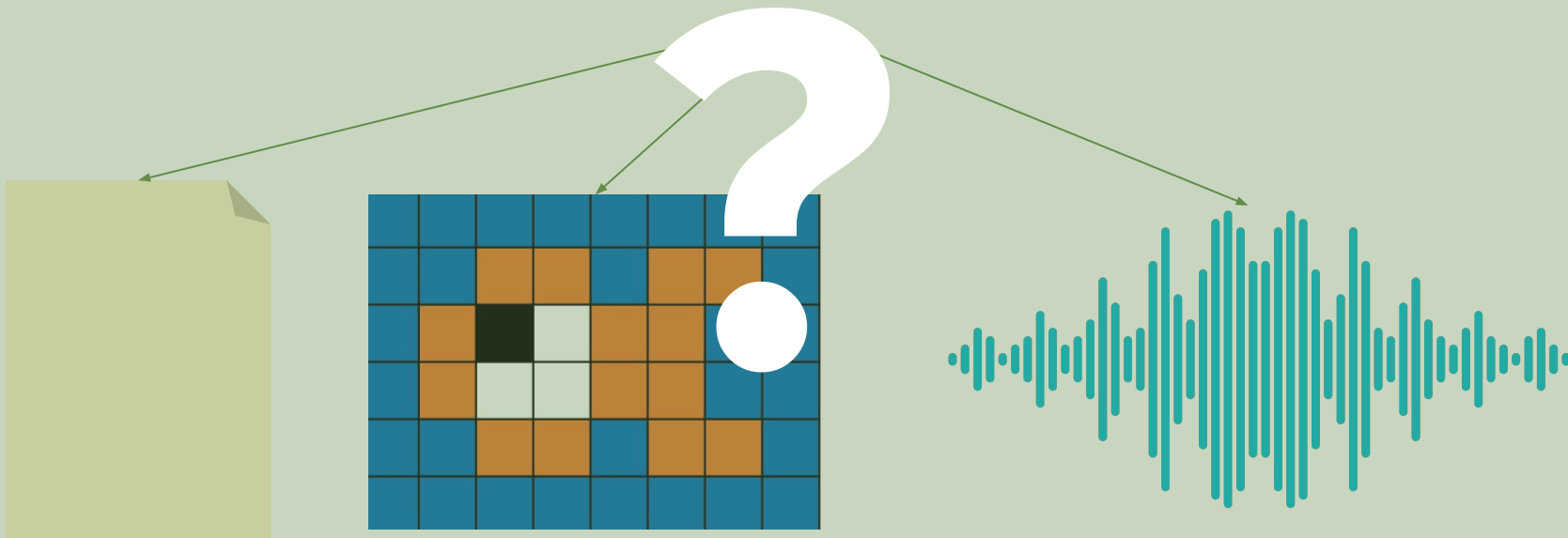
rwX r-- r--  
7 5 5

Memory locations  
(hexadecimal)

gone\_fishin.png

# Data representation

...000001100101110111101010110010010100100001001...



# But, why binary?




**ON (1)**



**OFF (0)**

# 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**<sub>10</sub>  **?**<sub>2</sub>

# DEC2BIN

200	Even	0
$200 \div 2$	Even	0

**200**<sub>10</sub>



**?**<sub>2</sub>

# DEC2BIN

200	Even	0
$200 \div 2$	Even	0
$100 \div 2$	Even	0

**200**<sub>10</sub>



**?**<sub>2</sub>

# DEC2BIN

**200**<sub>10</sub>



**?**<sub>2</sub>

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



# DEC2BIN

**200**<sub>10</sub>



**?**<sub>2</sub>

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

# DEC2BIN

**200**<sub>10</sub>



**?**<sub>2</sub>

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**<sub>10</sub>



**?**<sub>2</sub>

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**<sub>10</sub>



**?**<sub>2</sub>

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**<sub>10</sub>



**?**<sub>2</sub>

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

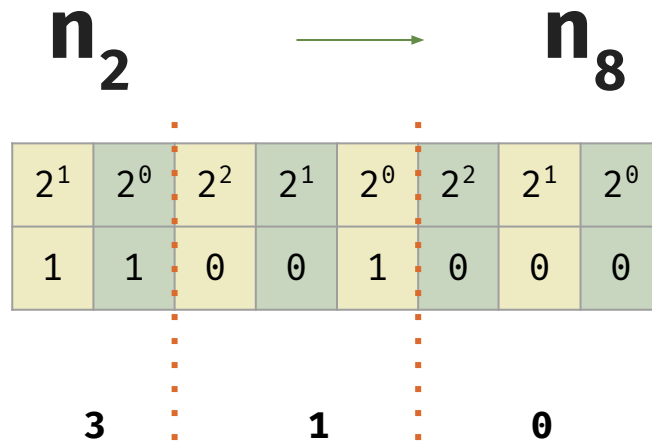
$2^0$	200	Even	0
$2^1$	$200 \div 2$	Even	0
$2^2$	$100 \div 2$	Even	0
$2^3$	$50 \div 2$	Odd	1
$2^4$	$25 \div 2$	Even	0
$2^5$	$12 \div 2$	Even	0
$2^6$	$6 \div 2$	Odd	1
$2^7$	$3 \div 2$	Odd	1
$2^8$	$1 \div 2$	No carry	0

**$200_{10}$**   $\longrightarrow$   **$?_2$**

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

111101011 ← Has to fit in 2 *bytes*

## 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

	111
5	101
+3	+011
<hr/>	<hr/>
8	1000

# Binary Arithmetic

		1 1
5		101
+5		+101
<hr/>		
10		1010

A problem for this  
system! Why?

Overflow!

# Binary Arithmetic

$$\begin{array}{r} 011 \\ + 101 \\ \hline 1000 \\ 2 \end{array}$$

Diagram illustrating binary addition of 011 (3) and 101 (5) to get 1000 (8). The result 1000 is shown as 2 in decimal. Arrows indicate the mapping from binary to decimal values: 011 to 3, 101 to 5, and the result 1000 to 2.

**2's Complement**

$$\begin{array}{r} 100 \\ + 001 \\ \hline 101 \end{array}$$

Diagram illustrating the 2's complement of 100 (4) to get 101 (5). The result 101 is shown as 5 in decimal. An arrow indicates the mapping from the result 101 to 5.



# Binary Arithmetic

	1 1
5	101
+ -3	+101
<hr/>	<hr/>
2	<del>1</del> 010

2!

# Data size directives (Cortex M0+)

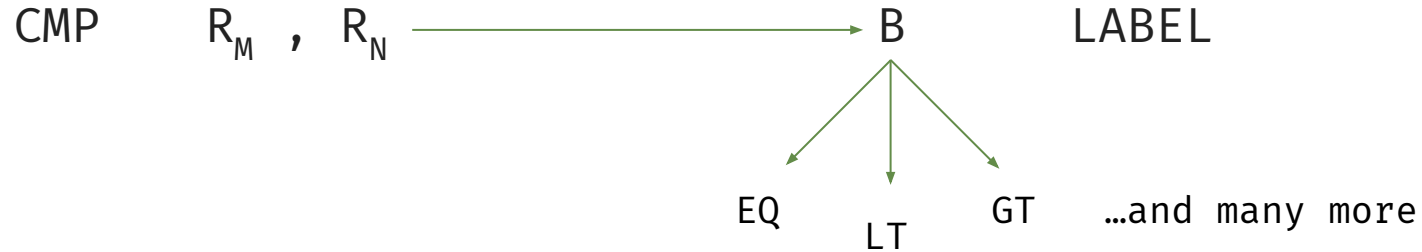
Byte	<code>.byte</code>	8-bit values
Half word	<code>.hword</code>	16-bit values
Word	<code>.word</code>	32-bit values
Double word	<code>.dword</code>	64-bit values

# Data size directives

Python	<code>a = 5</code>
C	<code>int a = 5;</code>
Assembly	<code>a:      .word    5</code>

← Each of these “reserve” 4 bytes

# ARMv6 Jumping Around

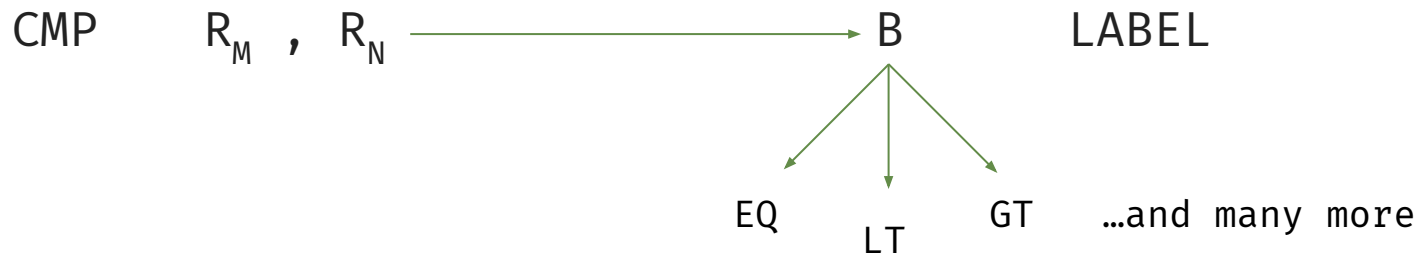


# ARMv6 Taking a break

A new instruction from the instruction set helps us pause programs in execution:

BKPT

# ARMv6 Jumping Around



There's a register  
for that...

31	30	29	28	27	26-0
N	Z	C	V	Q	Reserved

## The xPSR

If these flags are set...things...happen

# ARMv6 *Signed* bytes

Load Register with Signed Byte



LDRSB

Has to be a register with  
an offset



$R_M, [R_N, R_D]$