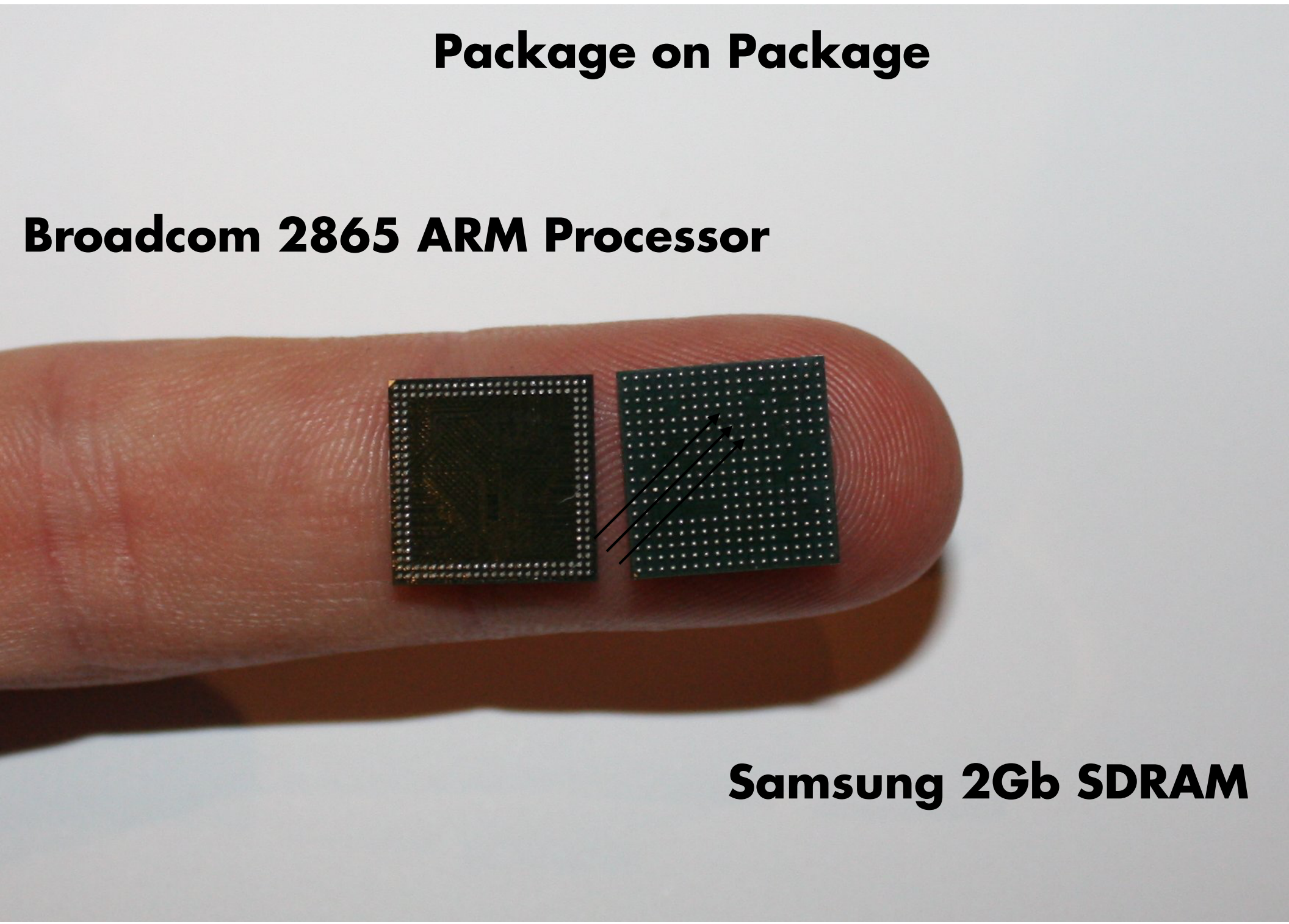
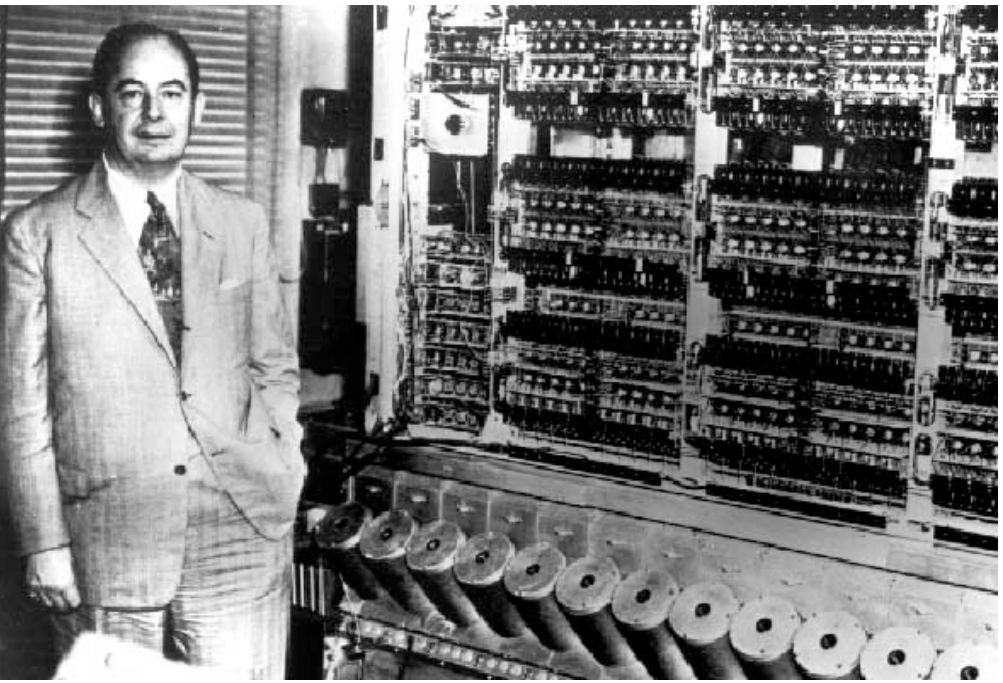
ARM

Architecture and Assembly

Modest Goal: Turn on an LED

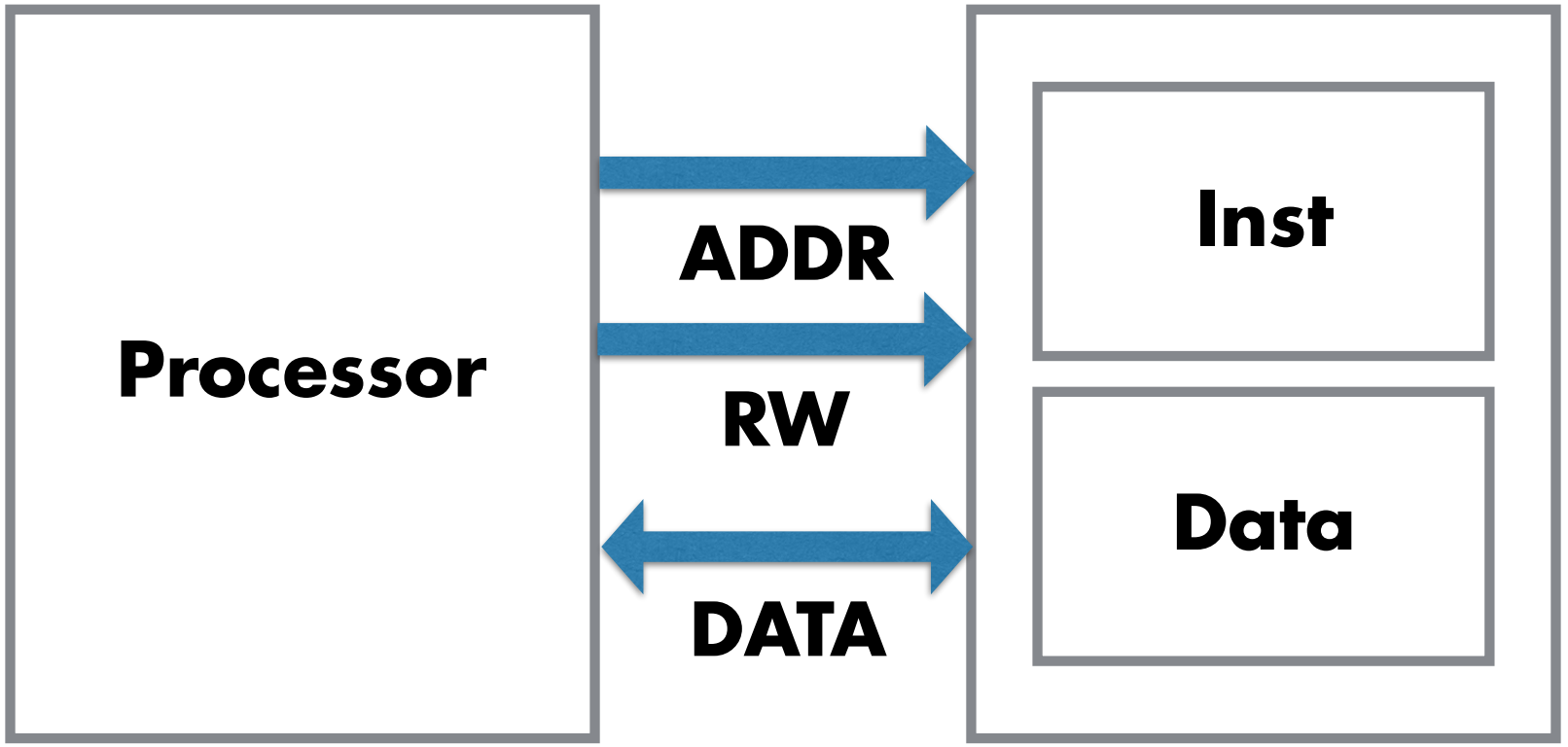
Package on Package





(John) von Neumann Architecture

Instructions and data stored in the same memory



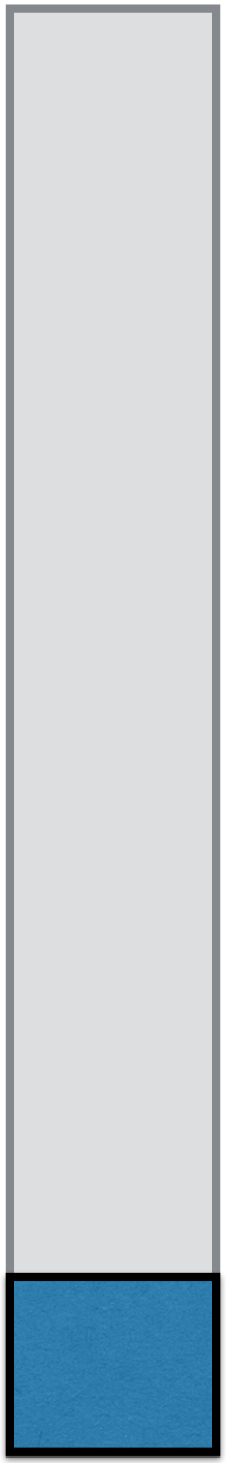
Memory used to store information

Stores both instructions and data

Storage locations are named using 32-bit addresses

Address refers to a *byte (8-bits)*

Maximum memory 4 GB
Actual memory is 256 MB



100000000_{16}

Memory Map

010000000_{16}

256 MB

ARM 32-bit Architecture

Processor designed around 32-bit "words"

Registers are 32-bits

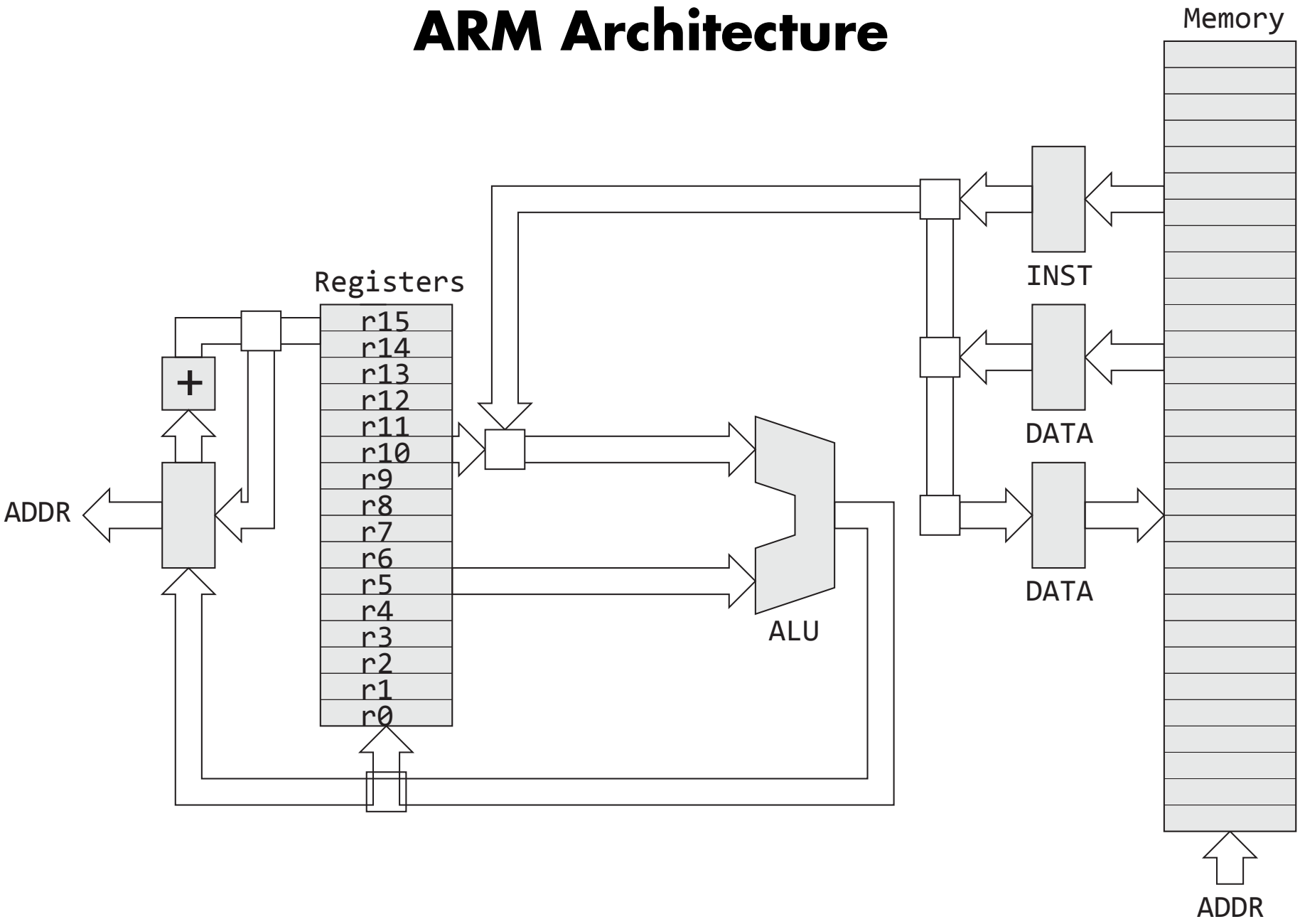
Arithmetic-Logic Unit (ALU) works on 32-bits

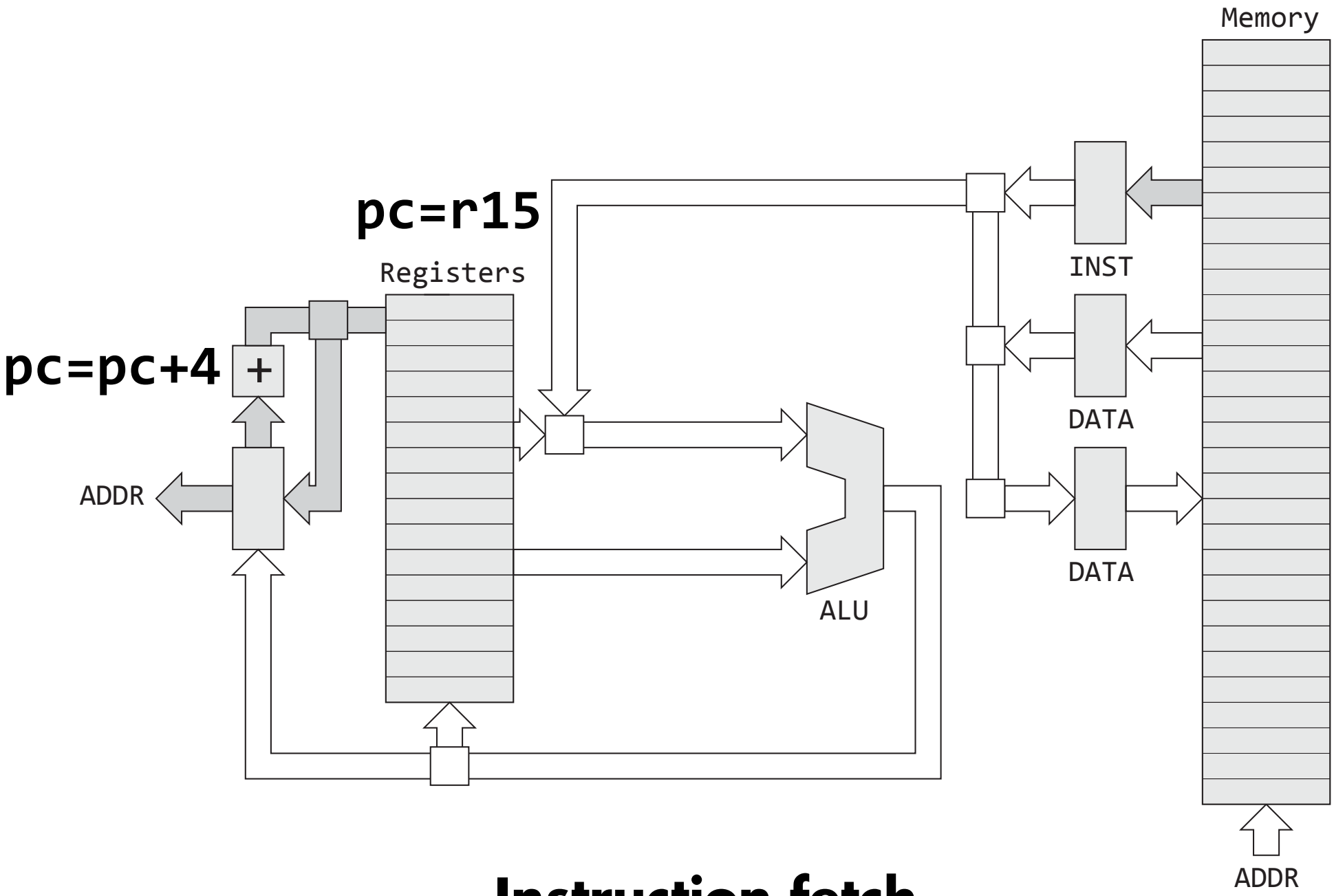
Addresses are 32-bits

Instructions are fixed size - 32-bits

The fact that everything is 32-bits simplifies things quite a bit!

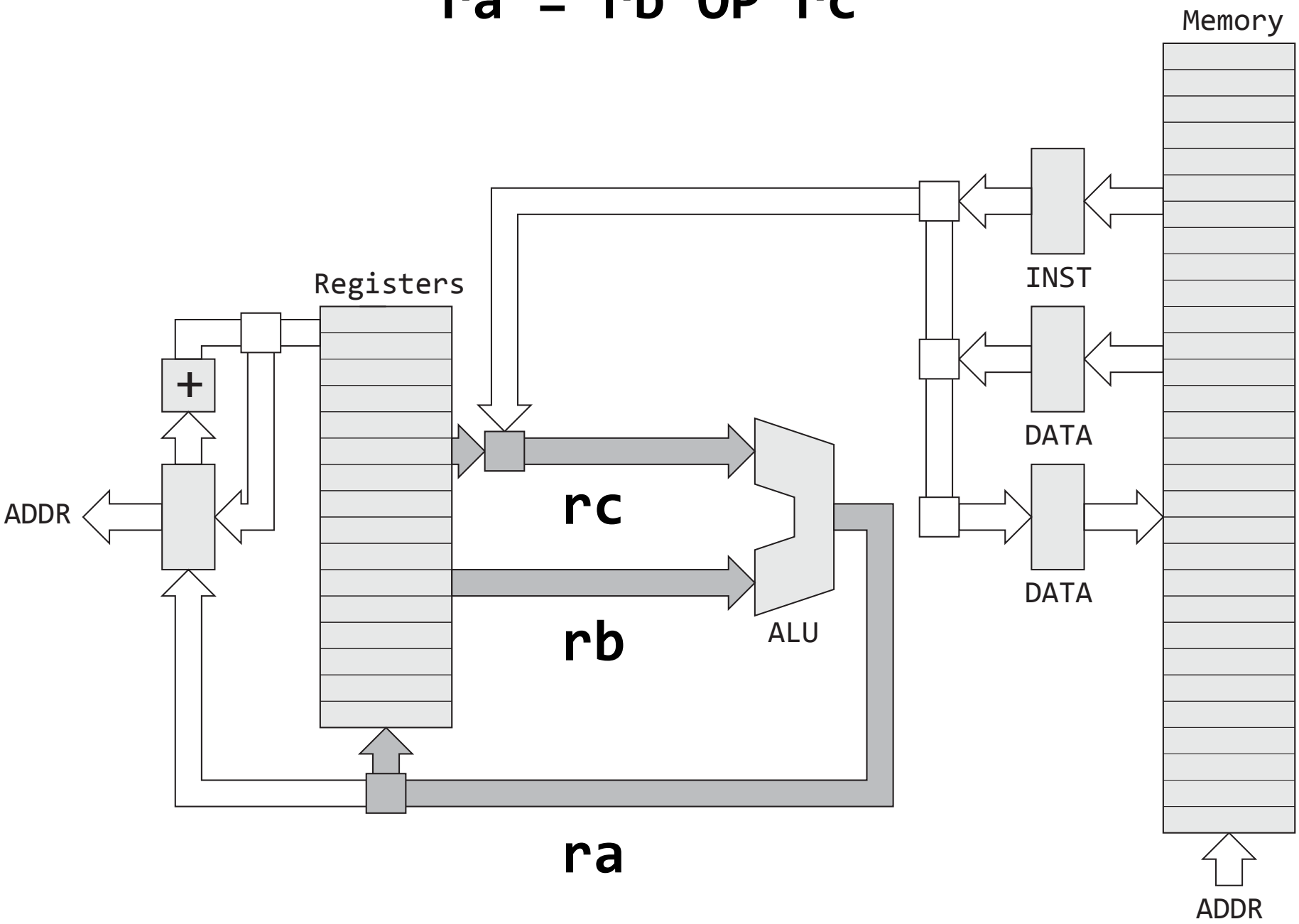
ARM Architecture





Instruction fetch

ra = rb OP rc



Instructions

Meaning (C)

```
r0 = r1 + 1
```

Assembly language

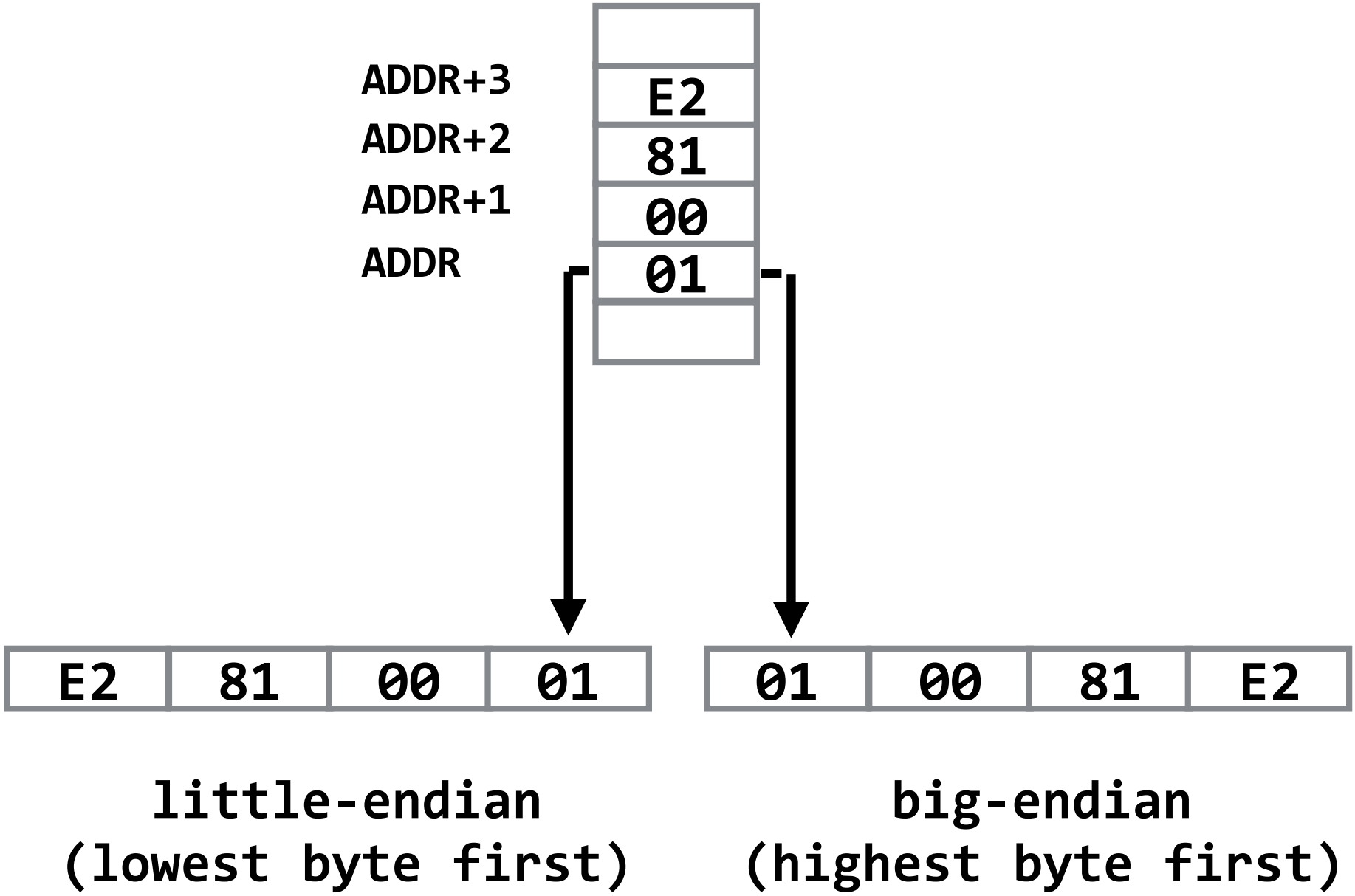
```
add r0, r1, #1
```

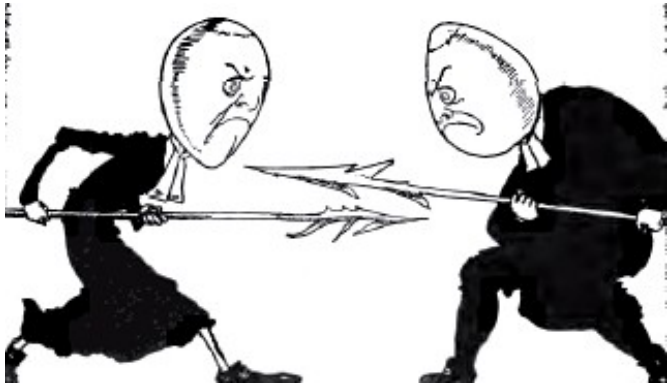
Machine language

```
E2 81 00 01
```

```
// Single instruction program

add r0, r1, #1 // #n is an immediate
```

```
# Assemble and generate listing
% arm-none-eabi-as add.s -o add.o -a

# Create binary
% arm-none-eabi-objdump add.o -O binary add.bin

# Size
% ls -l add.bin
-rw-r--r--+ 1 hanrahan  staff  4 add.bin

# Dump binary
% xxd -g 1 add.bin
0000000: 01 00 81 e2 // little-endian
```


ADDR+3
ADDR+2
ADDR+1
ADDR
E2
81
00
01
E2
81
00
01
01
00
81
E2
little-endian
(lowest byte first)
big-endian
(highest byte first)



The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's Gulliver s Travels. The inhabitants of Lilliput, who are well known for being rather small are, in addition, constrained by law to break their eggs only at the little end. When this law is imposed, those of their fellow citizens who prefer to break their eggs at the big end take exception to the new rule and civil war breaks out. The big-endians eventually take refuge on a nearby island, which is the kingdom of Blefuscu. The civil war results in many casualties.

Read: Holy Wars and a Plea For Peace, D. Cohen

Cross-Development

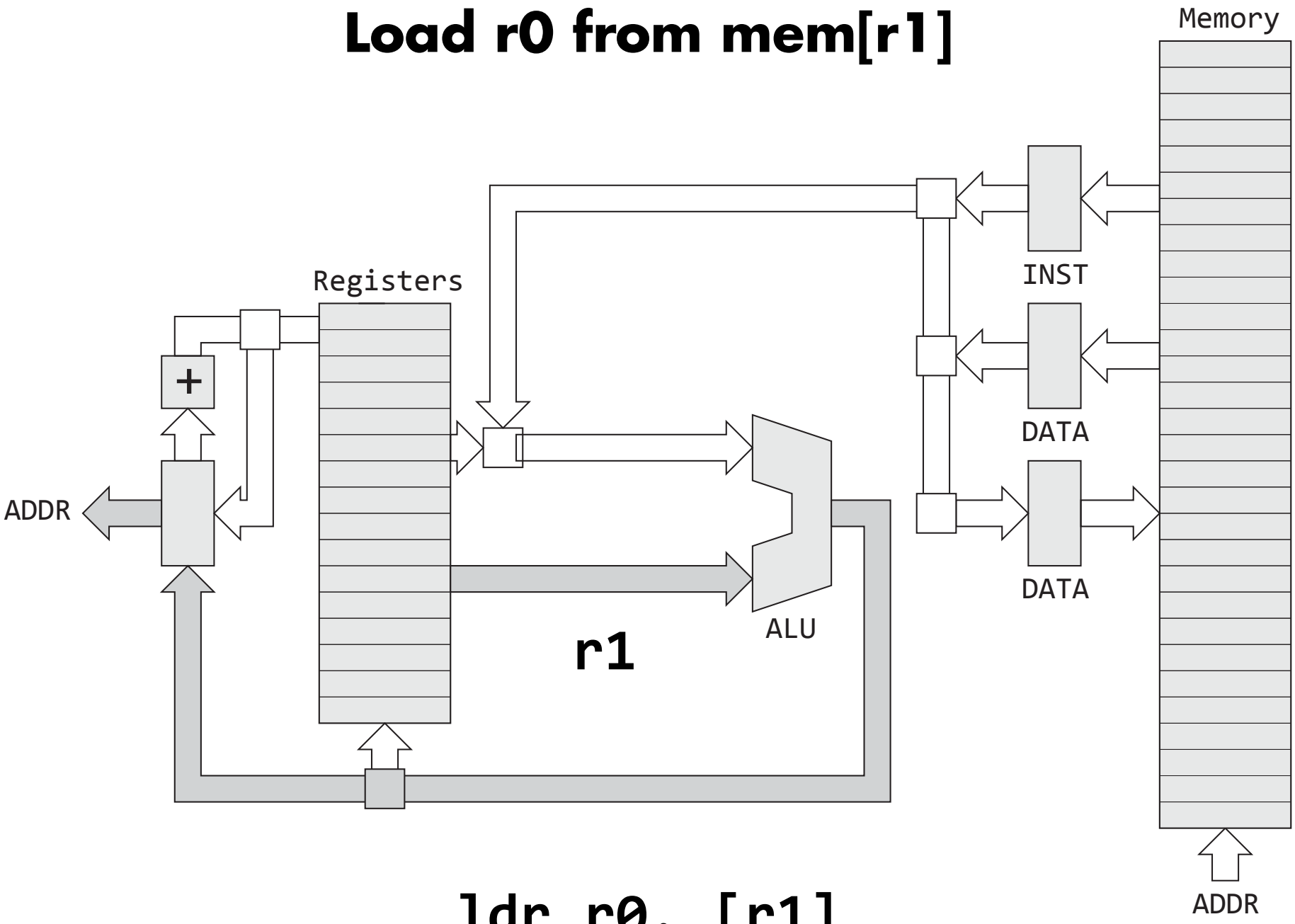
Run tools on your personal computer

Tools prefixed with arm-none-eabi-

Tools generate binary for raspberry pi

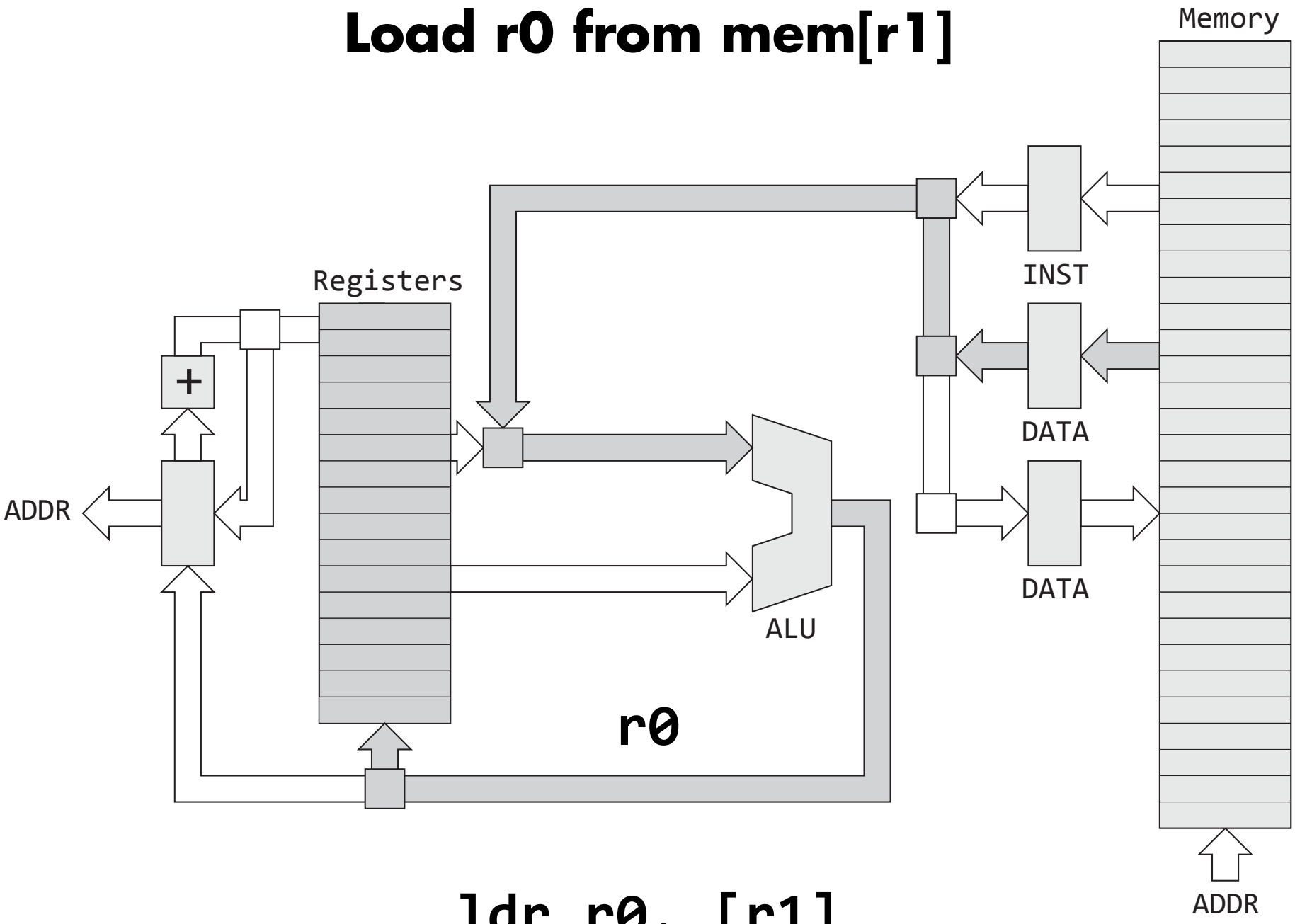
Loads and Stores

Load r0 from mem[r1]



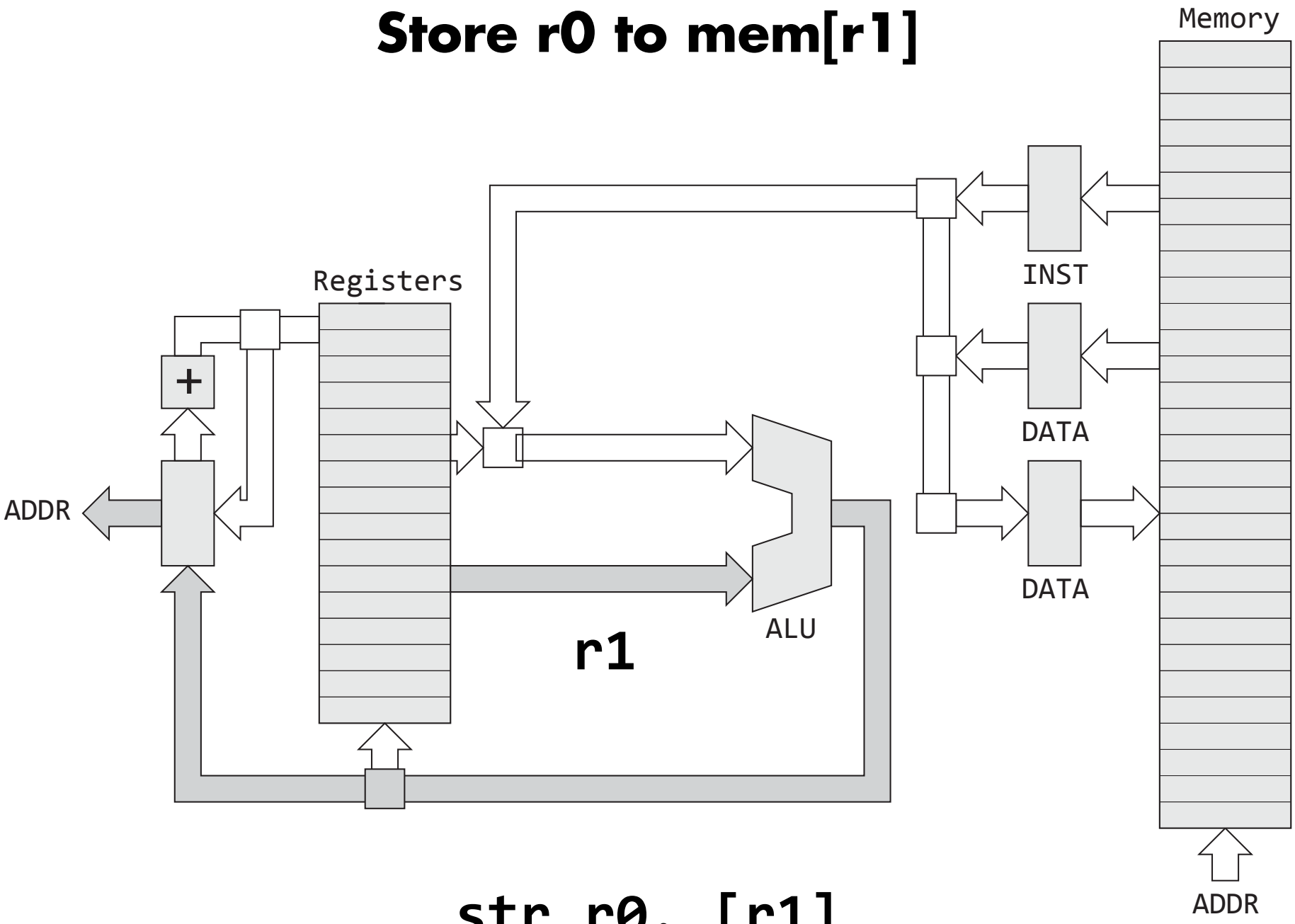

```
ldr r0, [r1]
```

Load r0 from mem[r1]



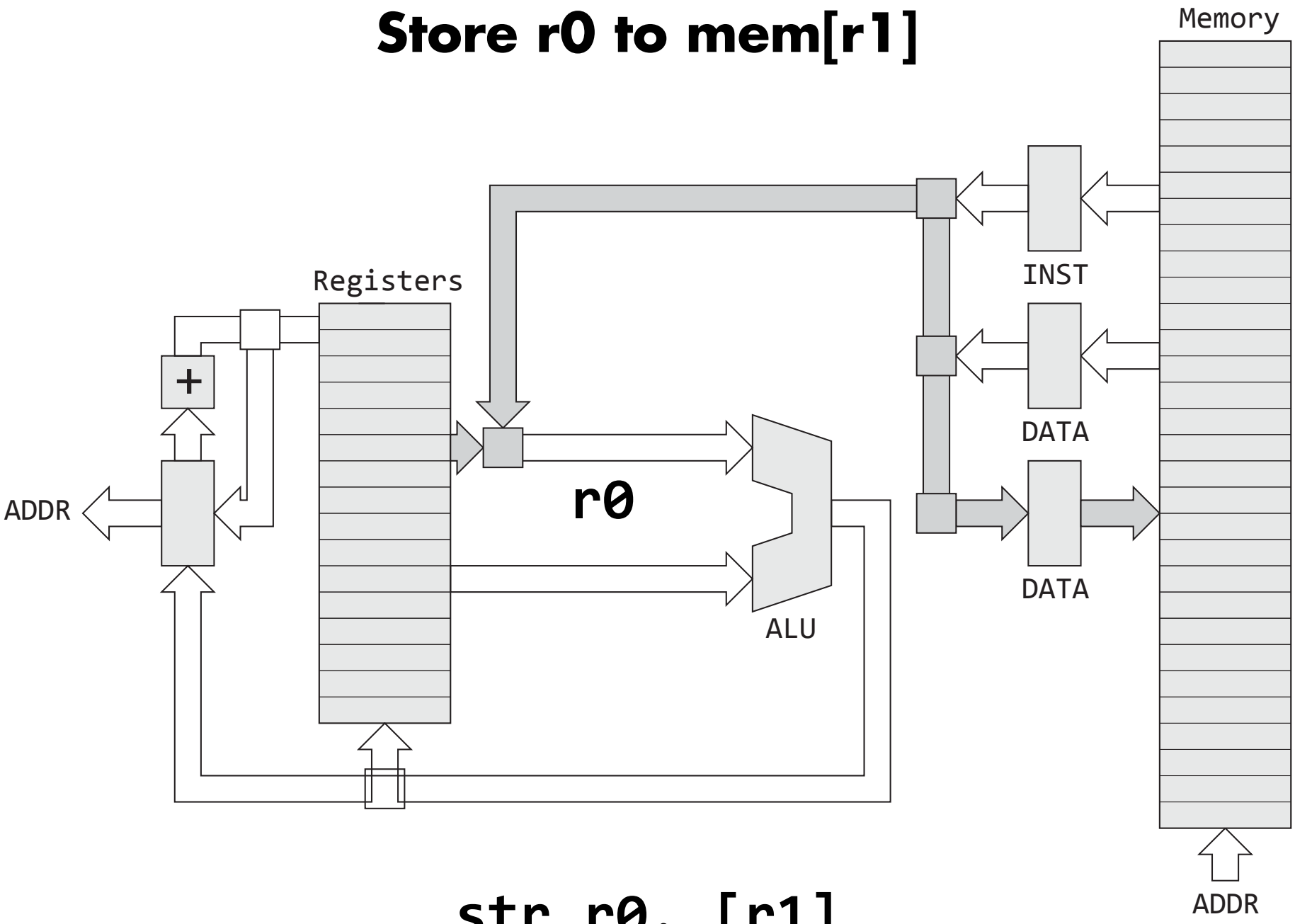

```
ldr r0, [r1]
```

Store r0 to mem[r1]




```
str r0, [r1]
```

Store r0 to mem[r1]




```
str r0, [r1]
```

Conceptual Exercises

1. Suppose you have `0x8000` stored in `r0`, how could you jump and start executing instructions at that location?

2. All instructions are 32-bits. Can you add any 32-bit immediate constant to a register using the `add` instruction?

3. What instruction do you think takes longer to execute, `str` or `add`?

Turning on an LED

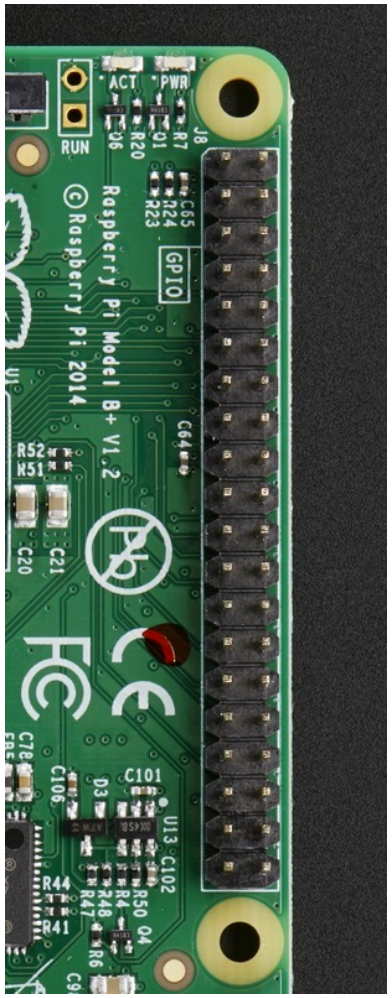
Powering

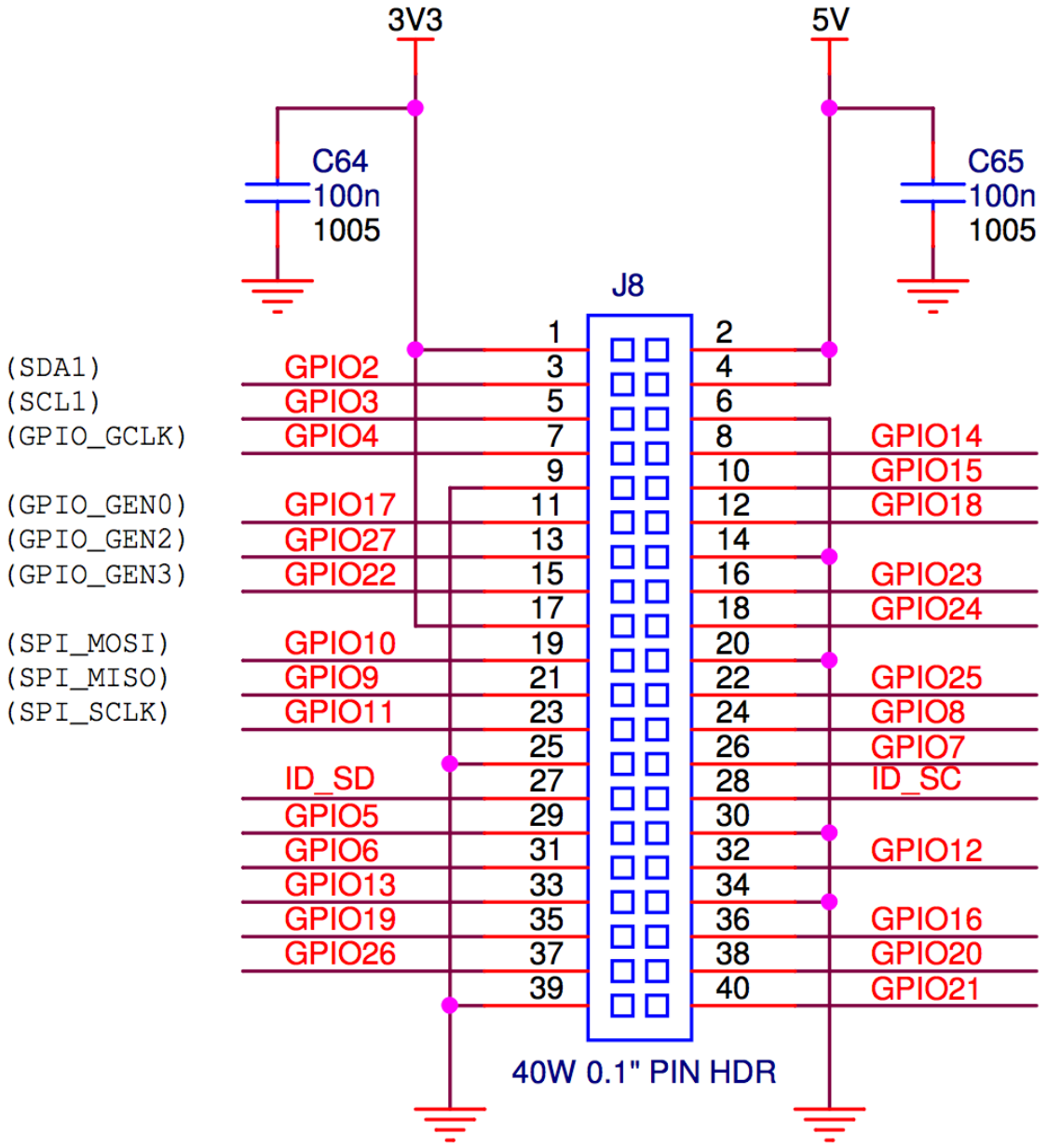
The USB port on my Macbook Pro provides 500 mA @ 5V

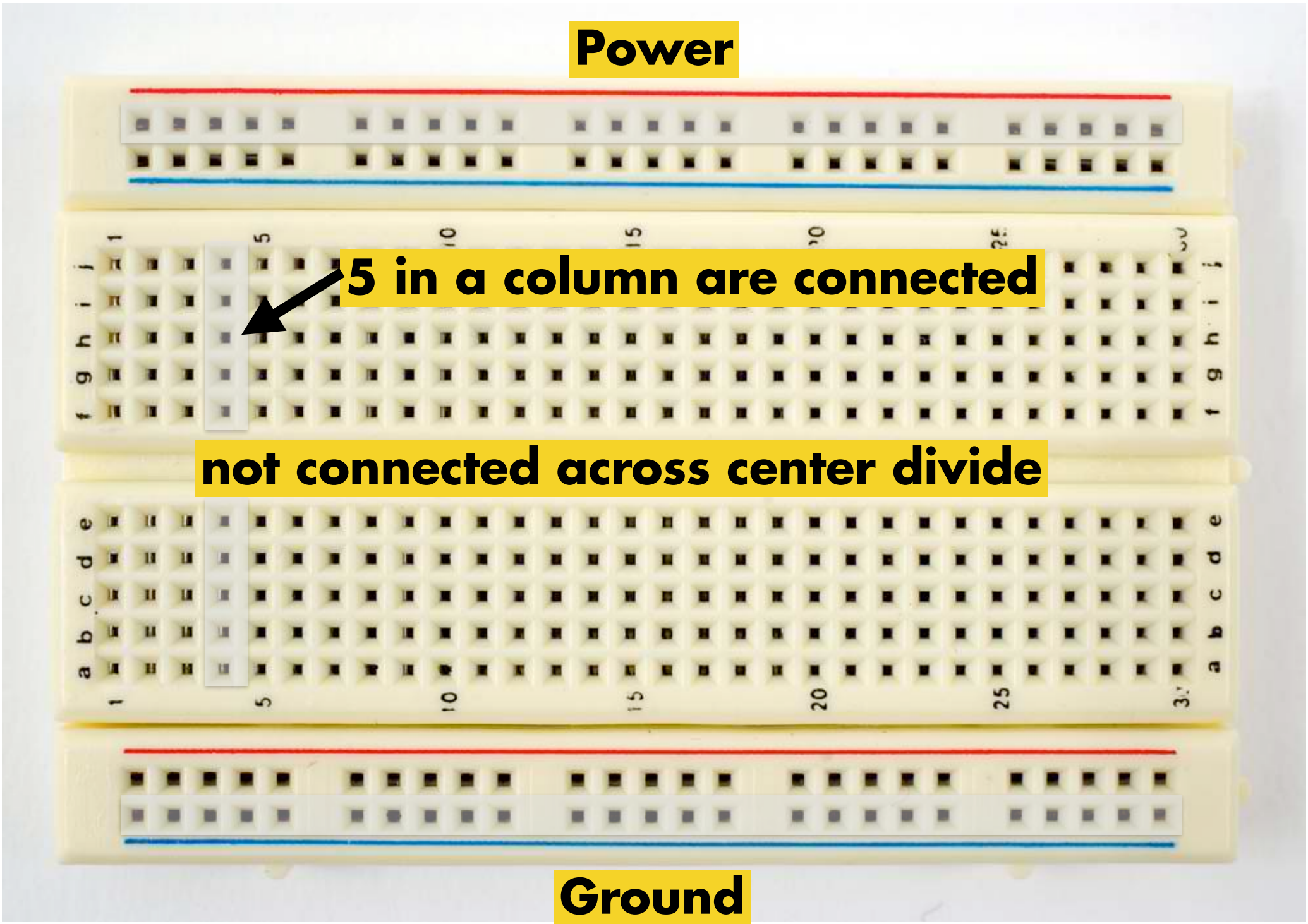
Power can be provided through the GPIO header

How much power does the Pi need?

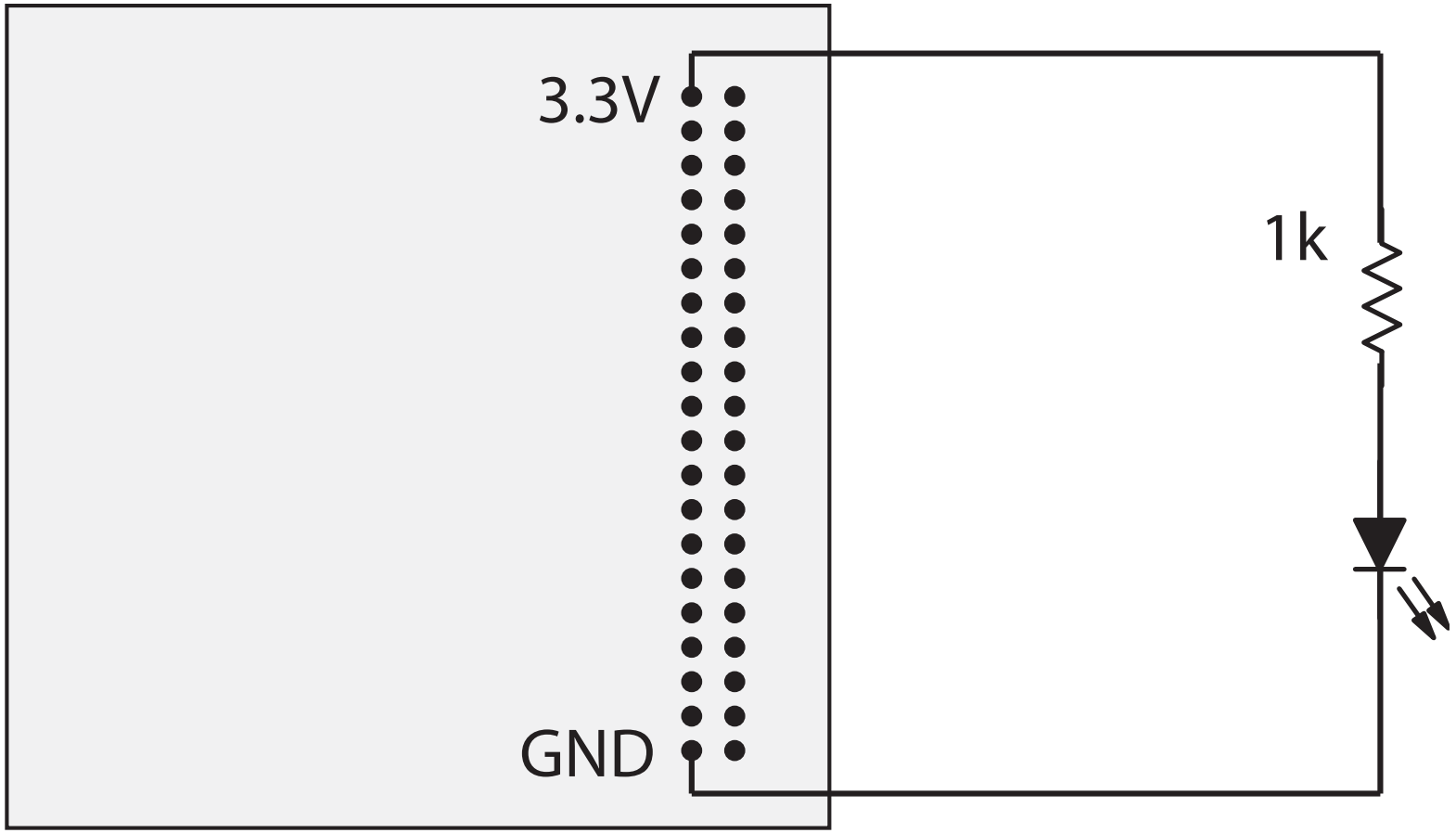
General-Purpose Input/Output (GPIO)



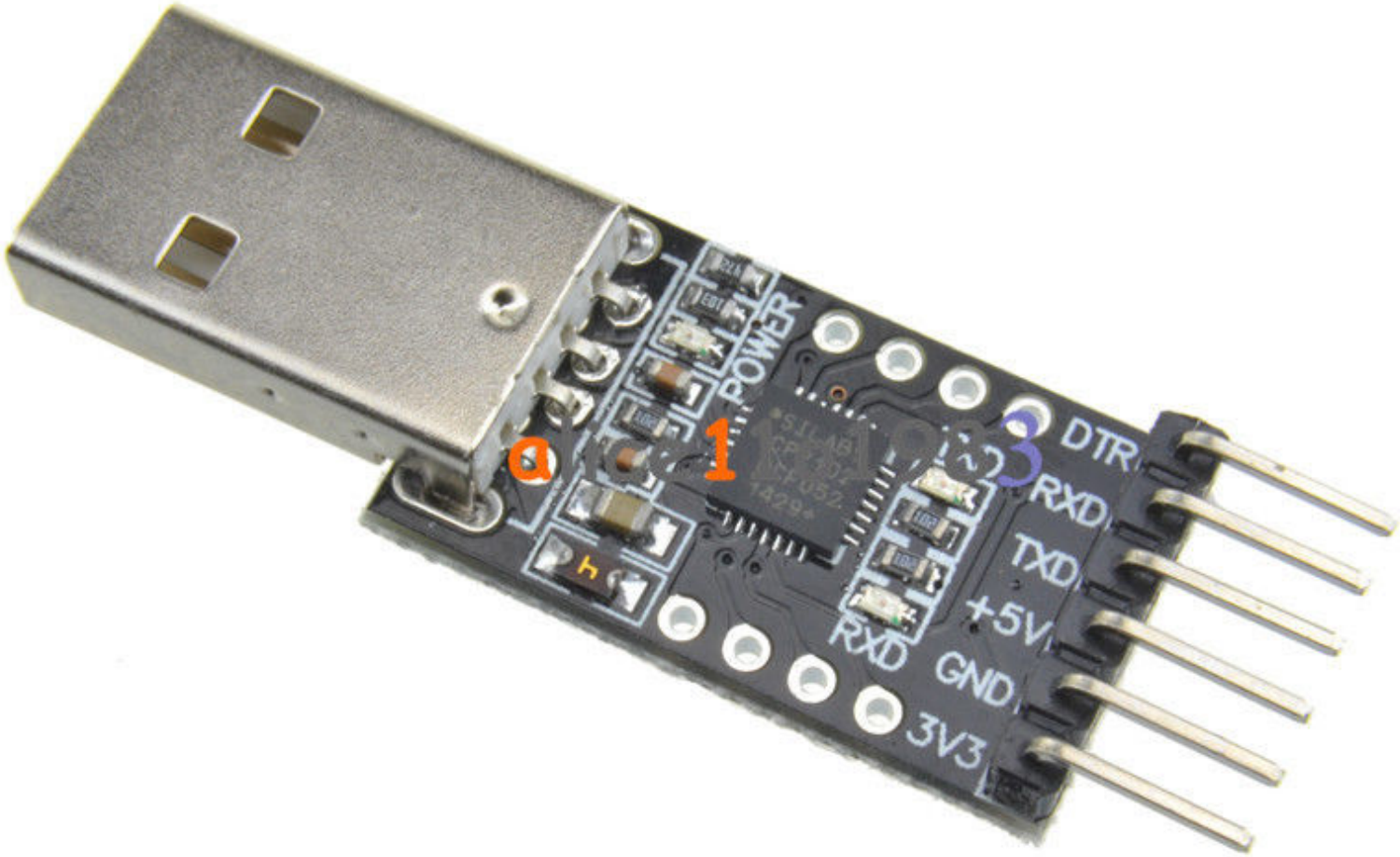


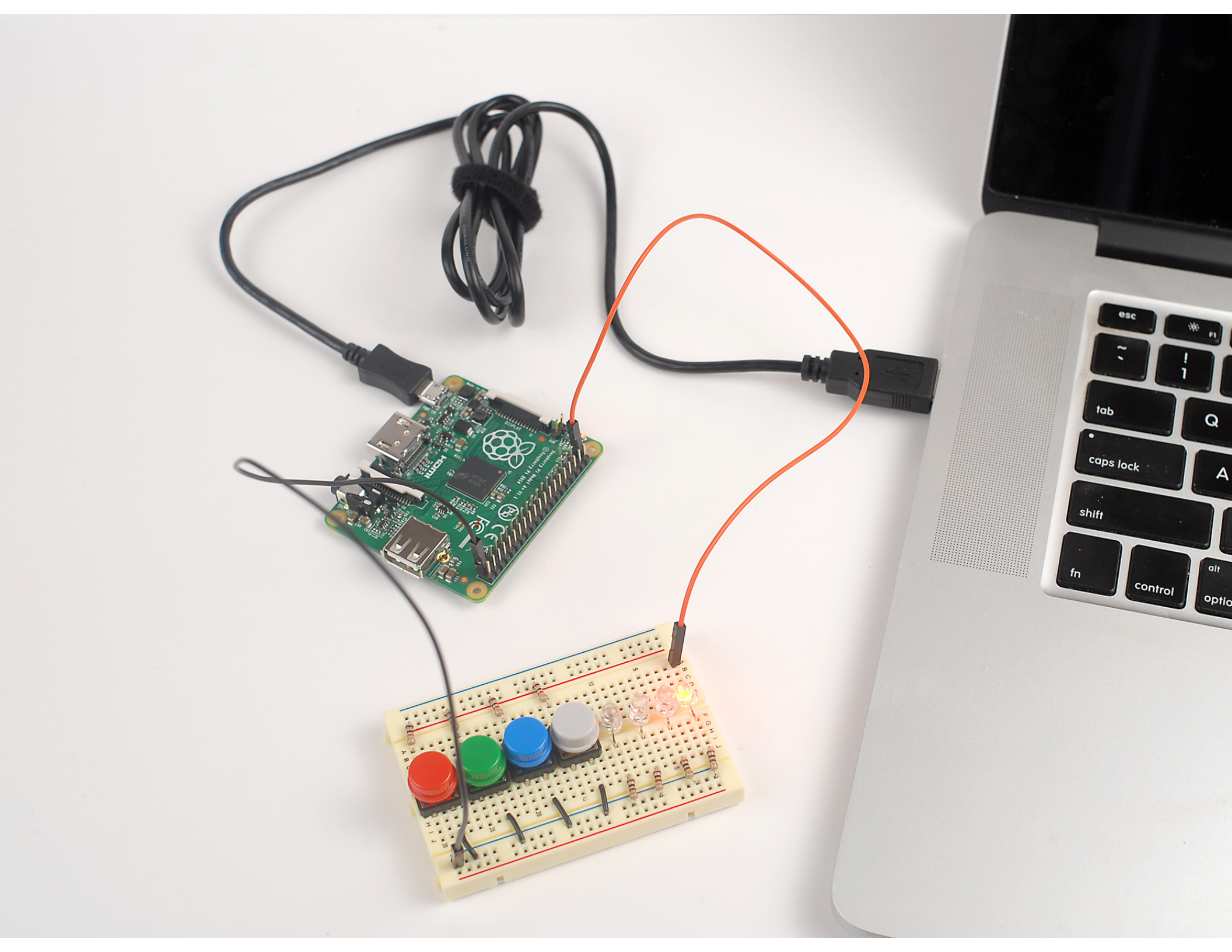

Power
5 in a column are connected
not connected across center divide
Ground

Logic 1 -> 3.3V (Vcc)
Logic 0 -> 0.0V (GND)



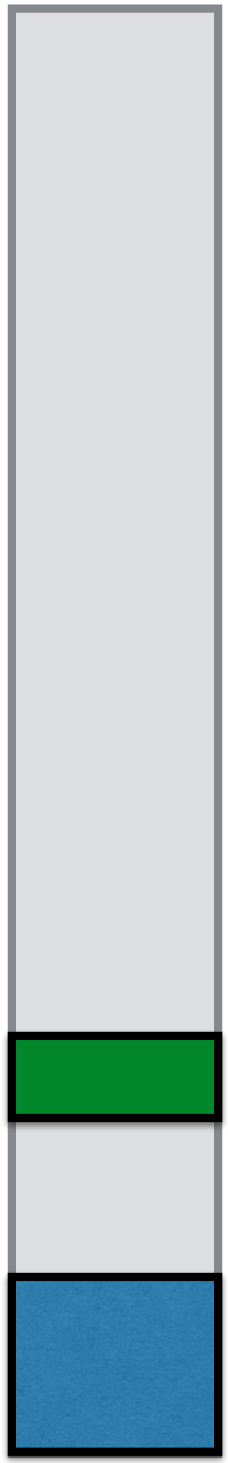
From the datasheet: A maximum of 16mA per pin with the total current from all pins not exceeding 51mA.


POWER
DTR
RXD
TXD
+5V
GND
3V3
RXD
a
1
3
h


esc
tab
caps lock
shift
fn
control

Controlling GPIO Pins

Memory Map



100000000_{16}

4 GB

Peripheral Registers

020000000_{16}

010000000_{16}

Ref: BCM2835-ARM-Peripherals.pdf

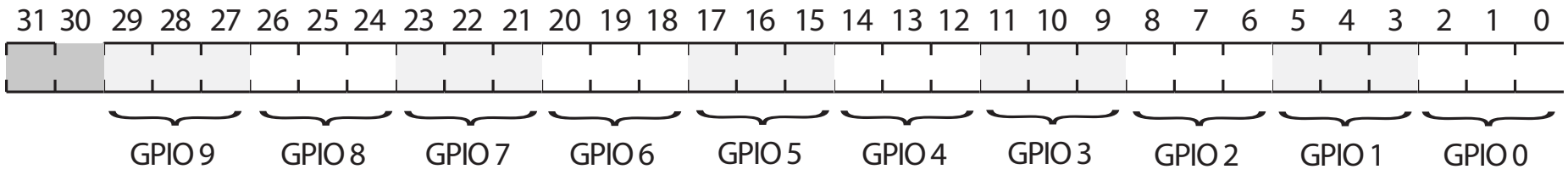
Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-

Notes

1. 0x 7E00 0000 -> 0x 2000 0000
2. 54 GPIO pins
3. 3-bits per GPIO pin
4. => 6 GPIO function select registers

Ref: BCM2835-ARM-Peripherals.pdf

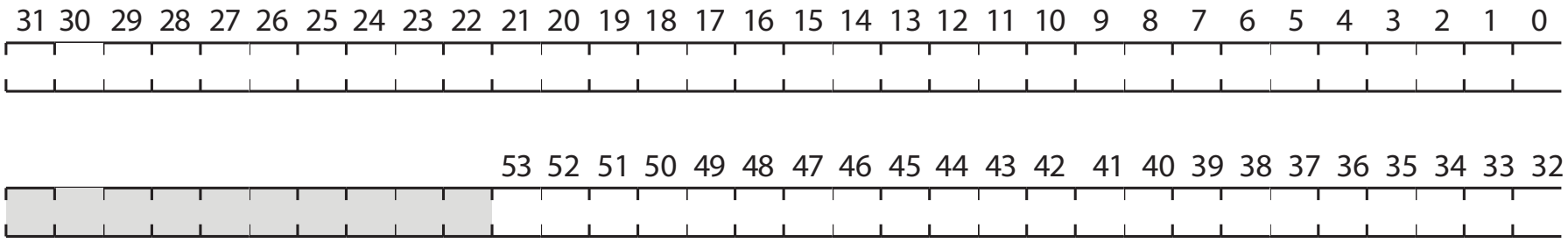
GPIO Function Select Register



Bit Pattern	Pin Function
000	The pin is an input
001	The pin is an output
010	The pin does alternate function 0
011	The pin does alternate function 1
100	The pin does alternate function 2
101	The pin does alternate function 3
110	The pin does alternate function 4
111	The pin does alternate function 5

```
// Turn on an LED via GPIO 20

// FSEL2 controls pins 20-29

// load r0 with GPIO FSEL2 address
ldr r0, =0x20200008

// GPIO 20 function select is bits 0-2
// load r1 with 1 (OUTPUT)
mov r1, #1

// store the value in r1
// to the address in r0
str r1, [r0]
```

20 20 00 1C : GPIO SET0 Register
20 20 00 20 : GPIO SET1 Register



```
…

// load r0 with GPIO SET0 register addr
ldr r0, =0x2020001C

// set bit 20 in r1
mov r1, #0x100000 // 0x100000 = 1 << 20

// store bit in GPIO SET0 register
str r1, [r0]
```

```
…

// load r0 with GPIO SET0 register addr
ldr r0, =0x2020001C

// set bit 20 in r1
mov r1, #(1<<20)

// store bit in GPIO SET0 register
str r1, [r0]
```

```
…

// load r0 with GPIO SET0 register addr
ldr r0, =0x2020001c

// set bit 20 in r1
mov r1, #(1<<20)

// store bit in GPIO SET0 register
str r1, [r0]

// loop forever using a branch
b .
```

```
# What to do on your laptop

# Assemble
% arm-none-eabi-as on.s -o on.o

# Create binary
% arm-none-eabi-objcopy on.o -O binary
on.bin

# Copy to SD card
% cp on.bin /Volumes/BARE/kernel.img
```

```
# What to do on your laptop

# Insert SD card - Volume mounts
% ls /Volumes/
BARE   Macintosh HD MobileBackups

# Copy to SD card
% cp on.bin /Volumes/BARE/kernel.img

# Eject and remove SD card
```

```
#
# Insert SD card into SDHC slot on pi
#
# Apply power using usb console cable
#  PWR LED - 3.3V present
#  ACT LED - SD card access
#
# Raspberry pi boots
#
# LED connected to GPIO20 turns on!!
#
```

Booting on Power On

1. Run hardware boot sequence
    1. Initializes the processor and memory
    2. Reads files from SDHC card
2. GPU runs `bootcode.bin`
3. GPU runs `start.elf`
4. GPU loads `kernel.img` at `0x8000`
5. ARM processor jumps to `0x8000`

Memory-Mapped IO (MMIO)

Peripherals which control input and output devices are controlled by special peripheral registers.

These registers are mapped into the address space of the processor.

These registers may not behave as memory.

- **Memory stores values. If you write a value, then read it, you get the value you wrote.**
- **Writing a 1 into a SET registers sets the output; writing a 0 into a SET register does not effect the output value. Reading the SET register always returns 0**

Details Omitted

Definitive References

ARMv6 architecture reference manual

BCM2865 peripherals document + errata

Raspberry pi schematic