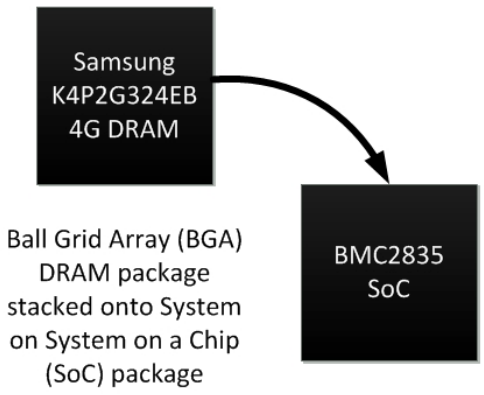
ARM

Architecture and Assembly

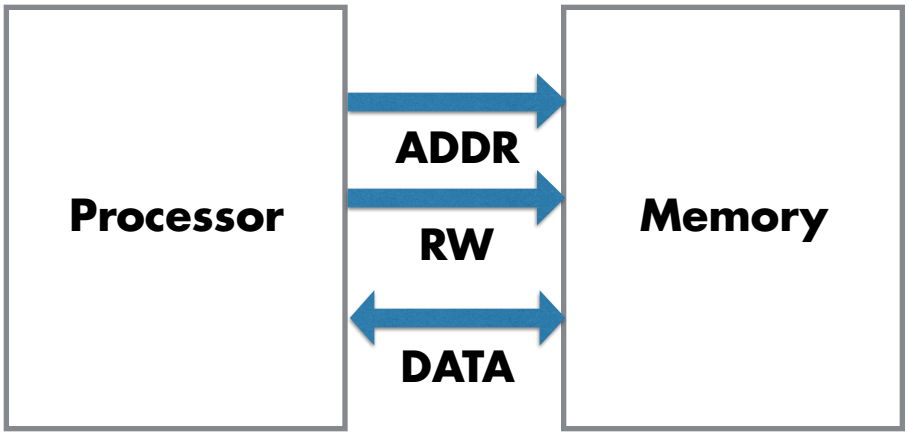
Modest Goal: Turn on an LED

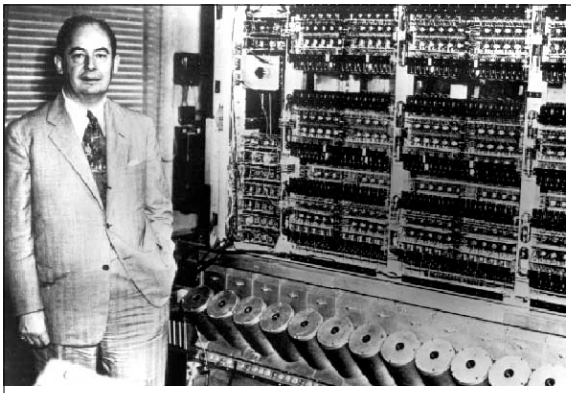
Samsung 2 Gb SDRAM



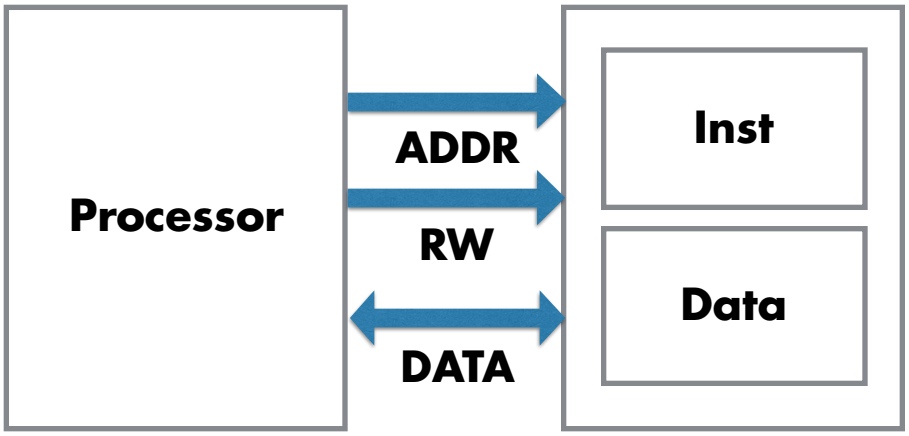
ARMv6 Instruction set architecture (ISA)
ARM1176JZF-S Microarchitecture
Broadcom 2835 ARM Chip

http://www.richardgoyette.com/RandEPIRecce.html





(John) von Neumann Architecture



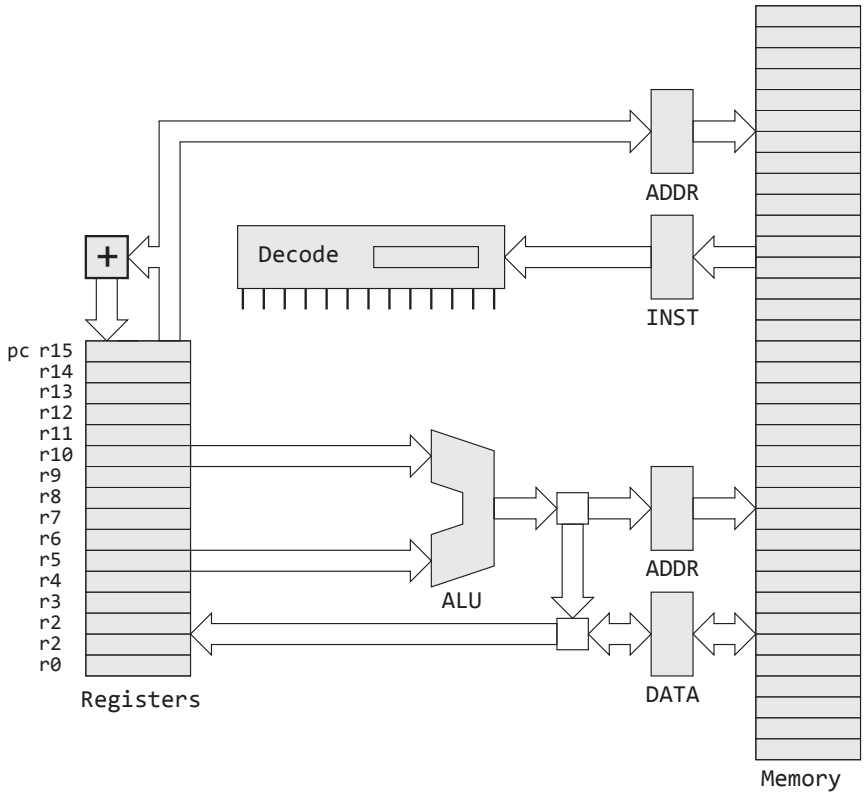
Memory Map

100000000_{16}
4 GB

32-bit addresses

Address refers to a *byte*

010000000_{16}
256 MB



ARM 32-bit Processor

Processor designed around 32-bit "words"

Registers are 32-bits

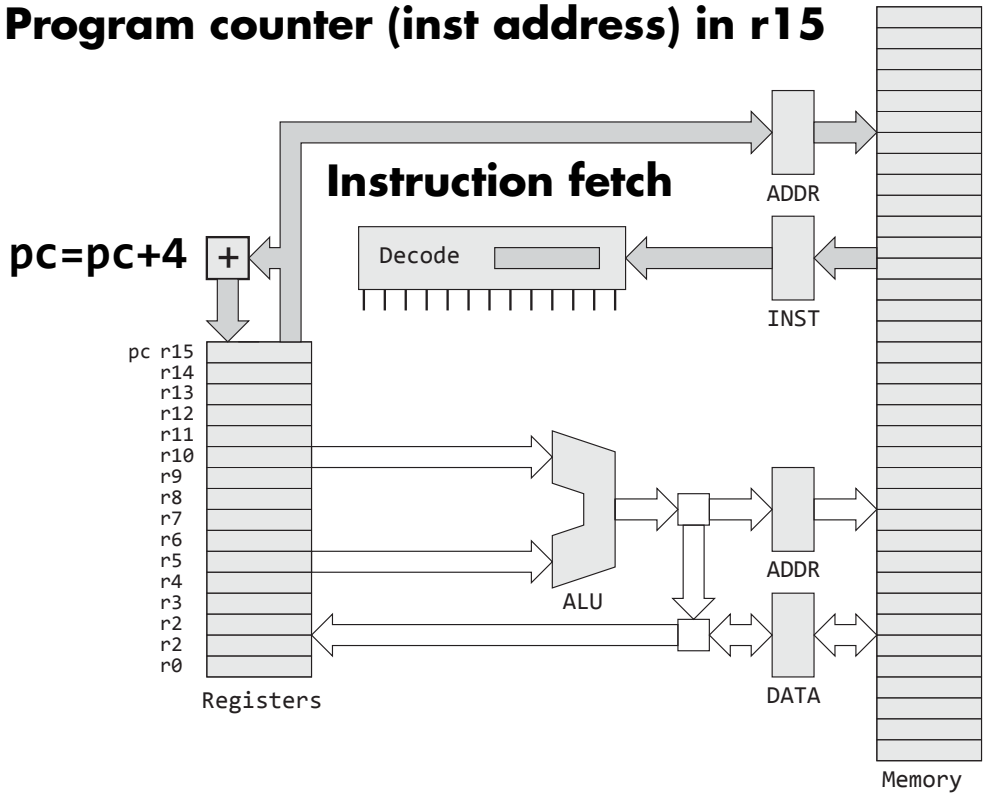
Arithmetic-Logic Unit (ALU) works on 32-bits

Addresses are 32-bits

Instructions are 32-bits

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

Program counter (inst address) in r15



Instructions

Meaning (C)

```
r0 = r1 + 1
```

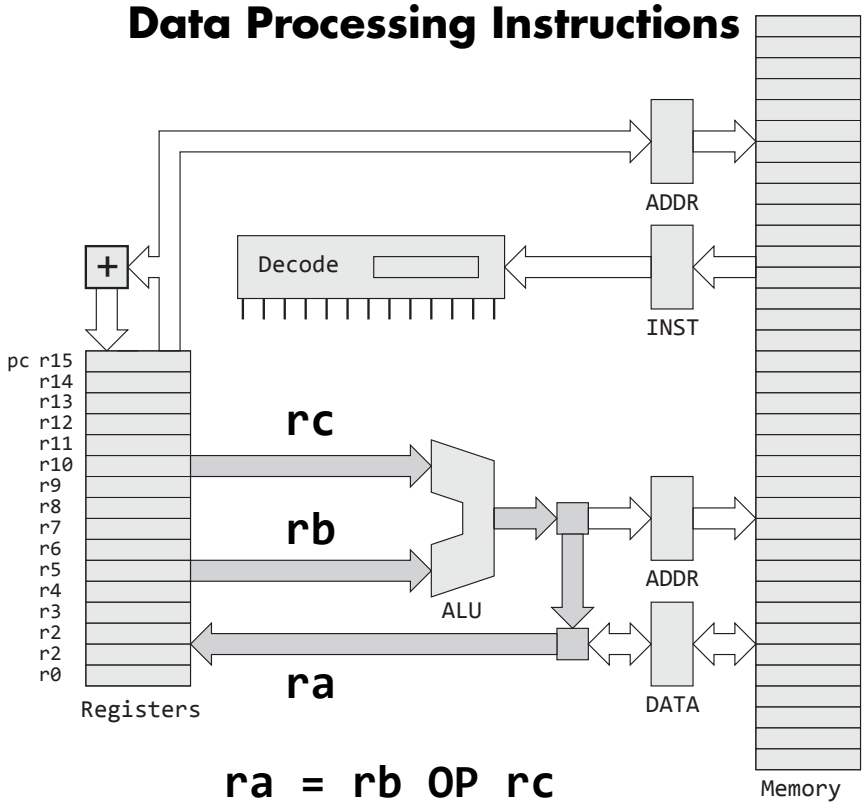
Assembly language

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

Machine language

```
E2 81 00 01
```

Data Processing Instructions




```
ra = rb OP rc
```

Operations (OP)

Arithmetic: ADD, SUB, MUL, (no DIV), ...

Logical/Bit: AND, ORR, EOR, LSL, ...

Comparison: CMP, TST, ...

Move: MOV, ...

More detail in the next lecture ...

```
// Single instruction program

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

Cross-Development

Run tools on your personal computer

Tools prefixed with arm-none-eabi-

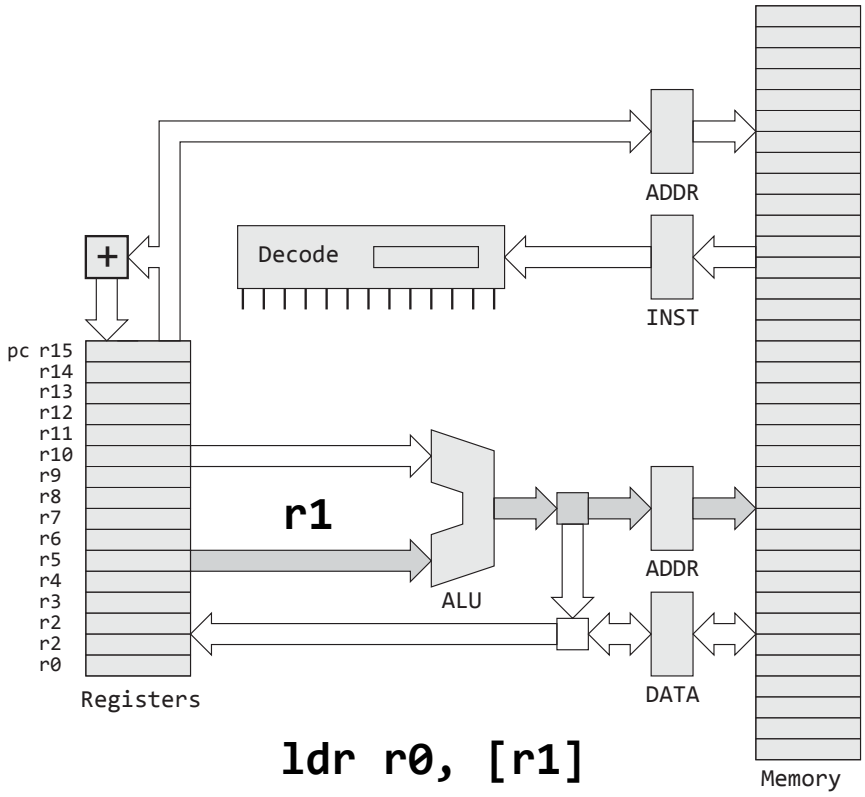
Tools generate binary for raspberry pi

Copy binary to the raspberry pi

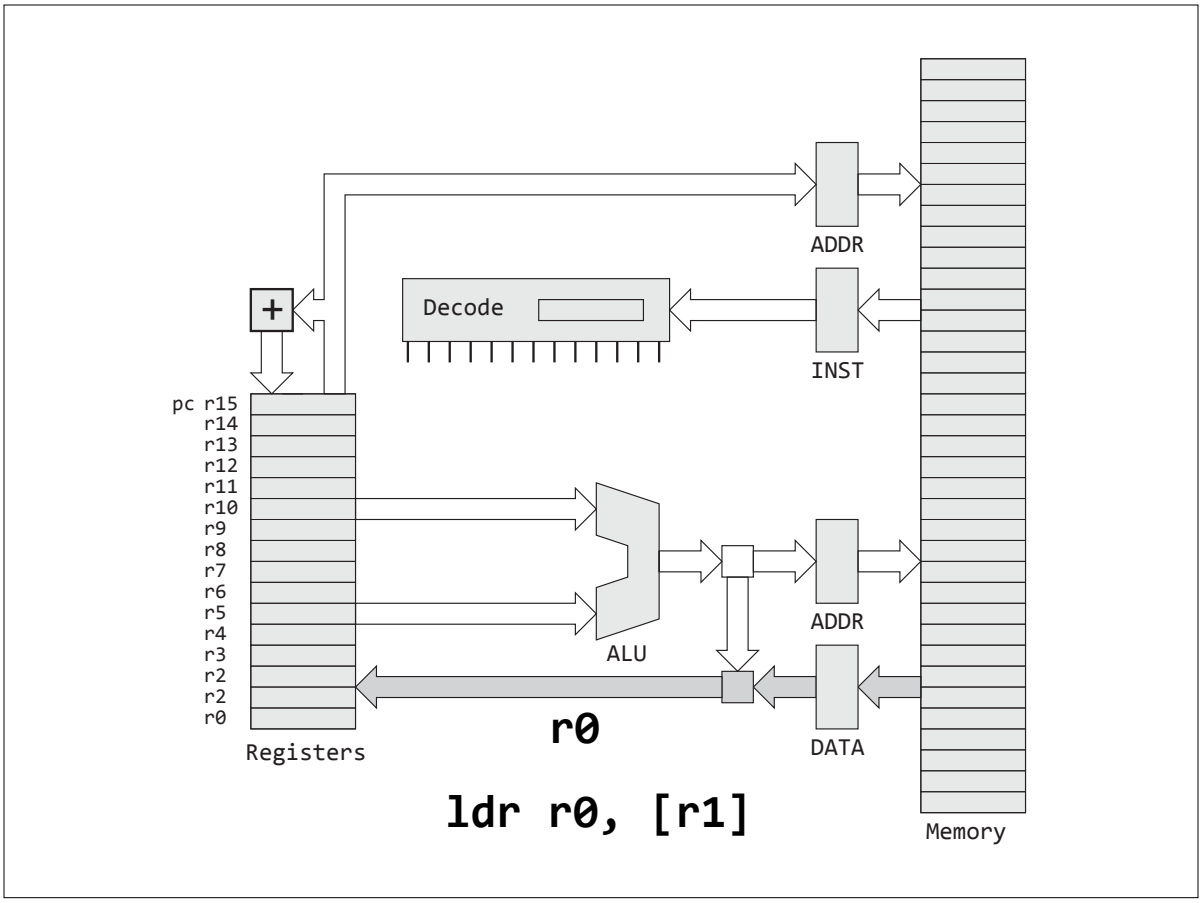
Power-cycle the raspberry pi to run

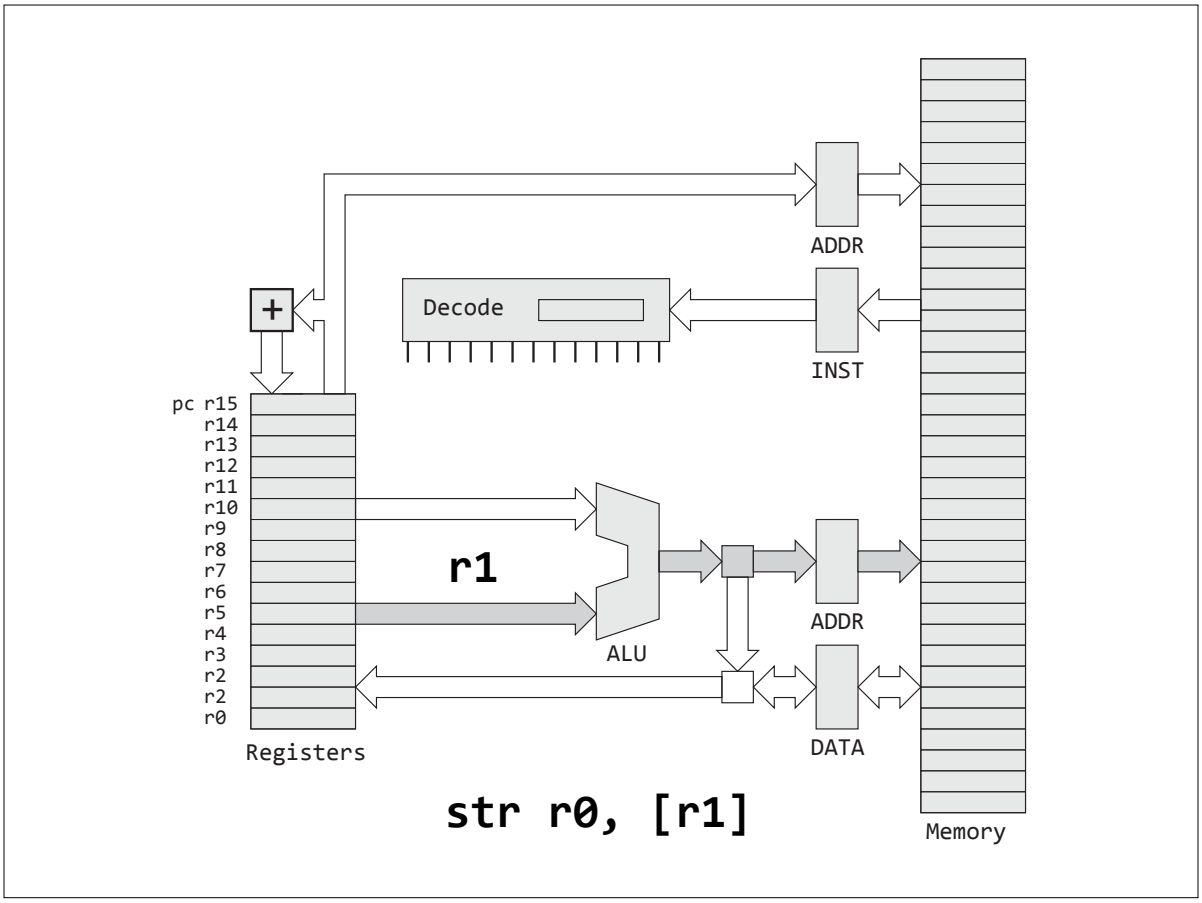
```
# Assemble and generate listing
% arm-none-eabi-as add.s -o add.o -a

# 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
```

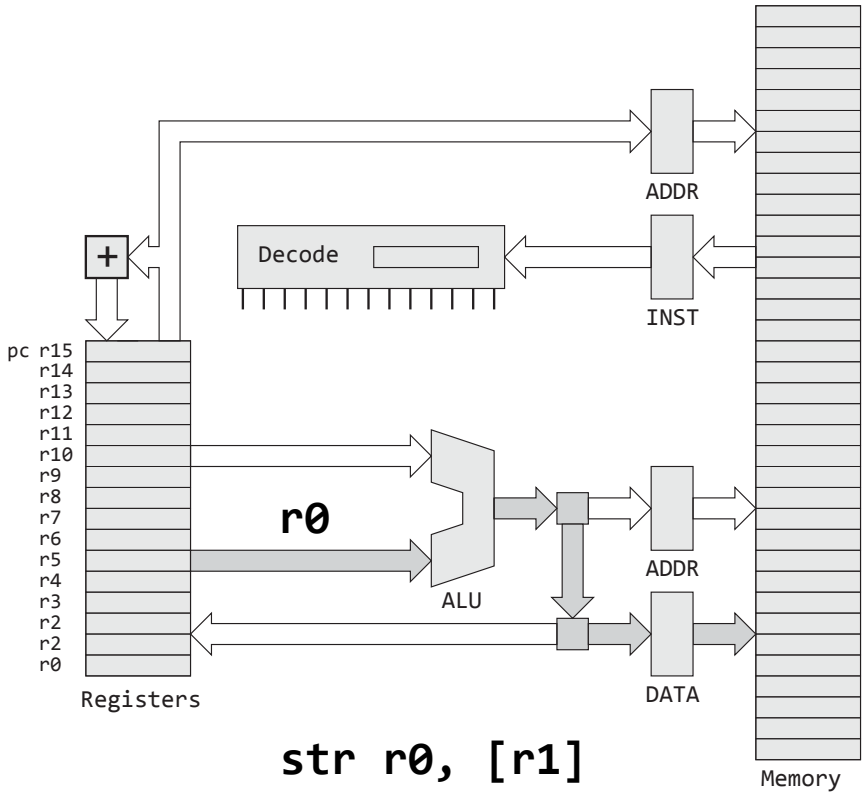
Loads and Stores



ldr r0, [r1]


ADDR
Decode
INST
+
pc r15
r14
r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r2
r0
ALU
ADDR
r0
DATA
Registers
Memory
ldr r0, [r1]


ADDR
Decode
INST
+
pc r15
r14
r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r2
r0
r1
ALU
ADDR
DATA
Registers
Memory
str r0, [r1]



Conceptual Exercises

1. Suppose you have `0x8000` stored in `r0`, how would 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 1 instruction?

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

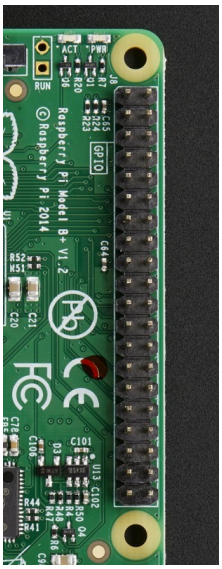
Turning on an LED

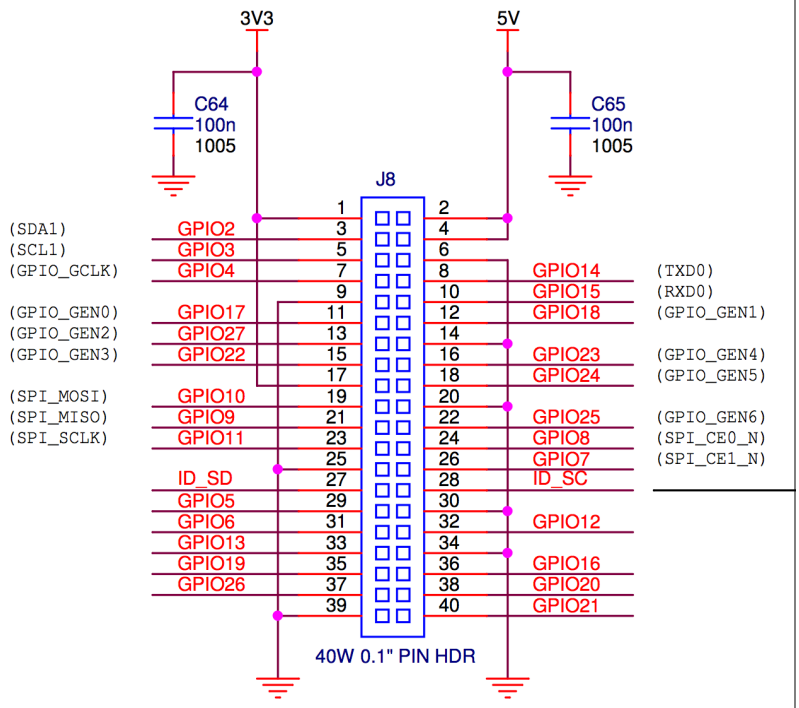
Powering

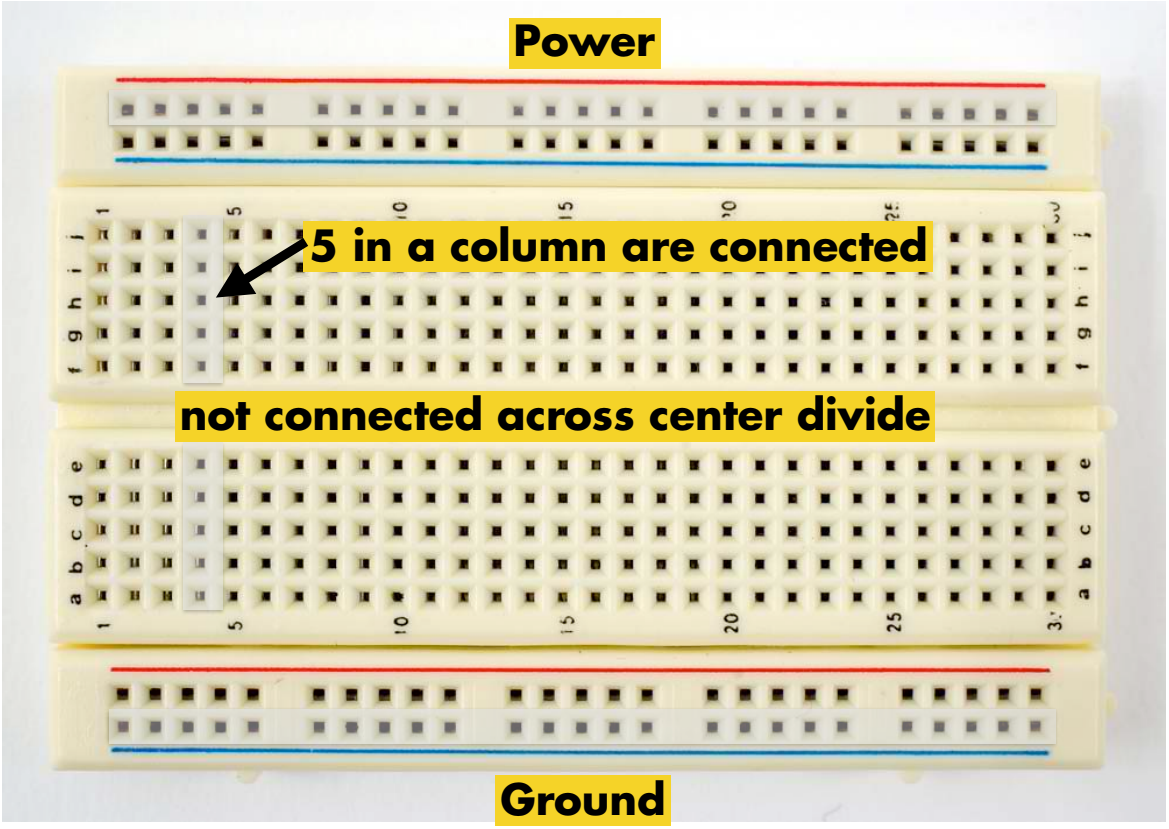
The USB port on my Macbook Pro provides 500 mA @ 5V

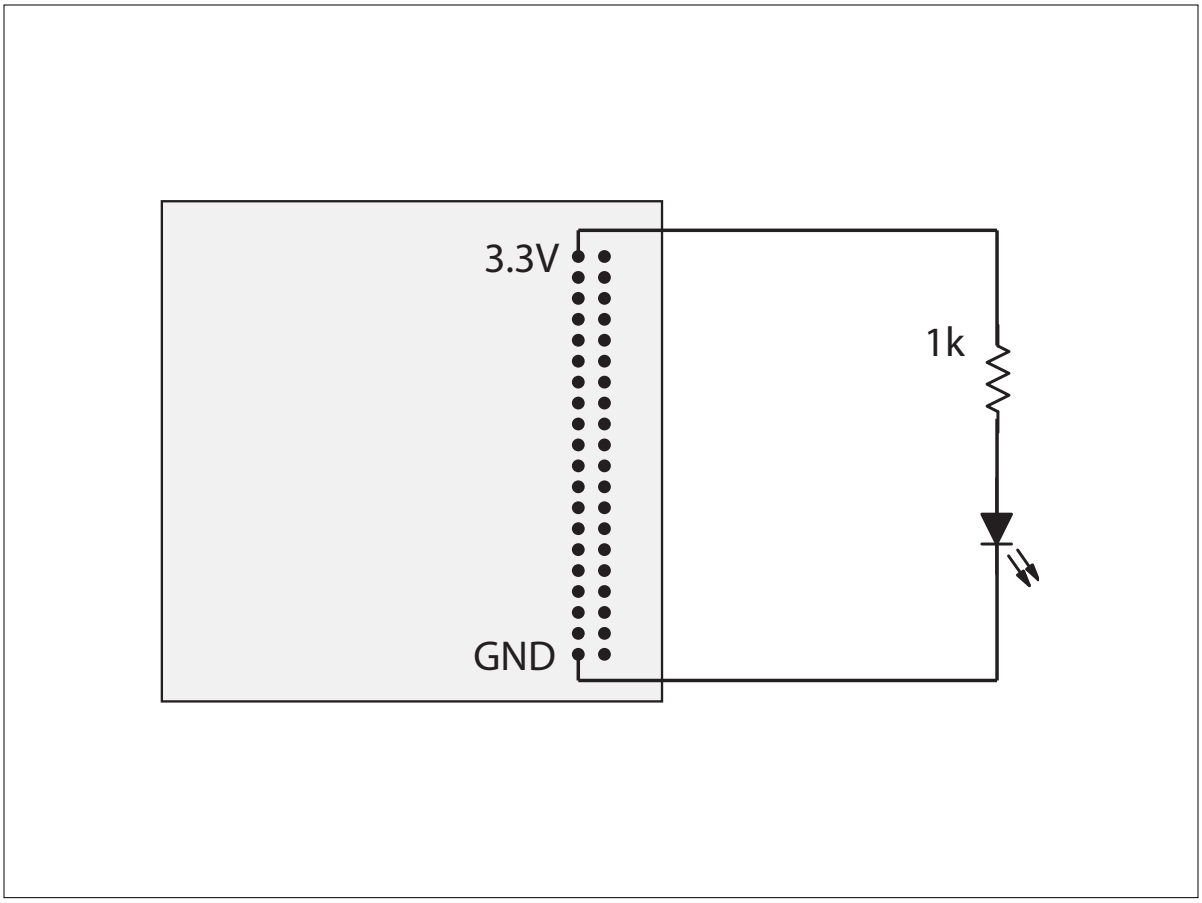
How much power does the Pi need?

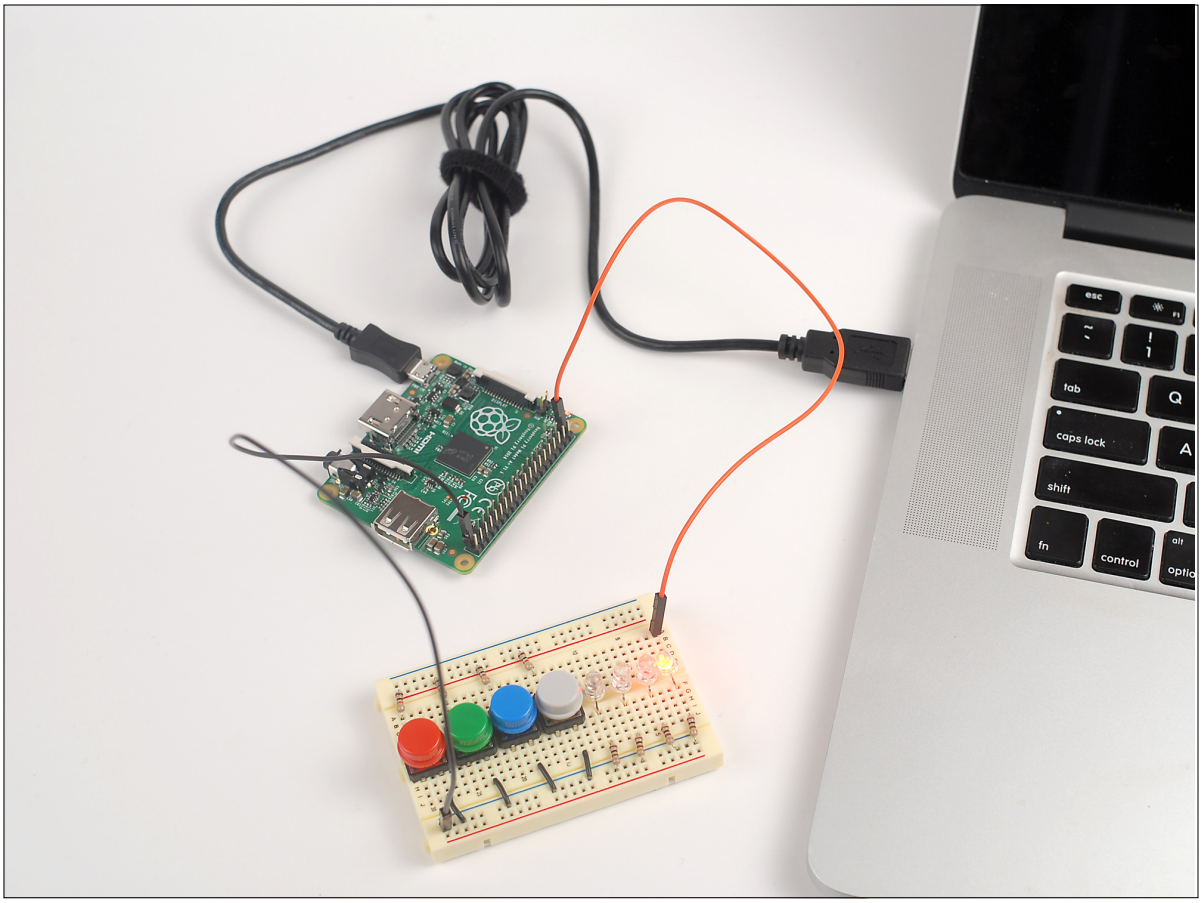
General-Purpose Input/Output (GPIO)





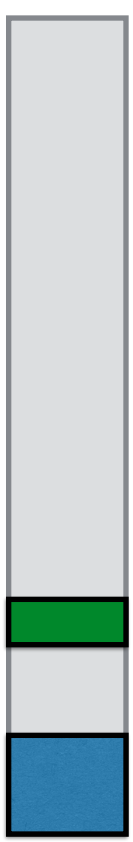



3.3V
1k
GND


esc
tab
caps lock
shift
fn
control

Controlling GPIO Pins

Memory Map



100000000_{16}

4 GB

Peripheral Registers

020000000_{16}

010000000_{16}

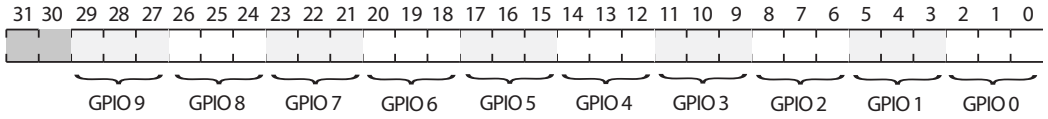
Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
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

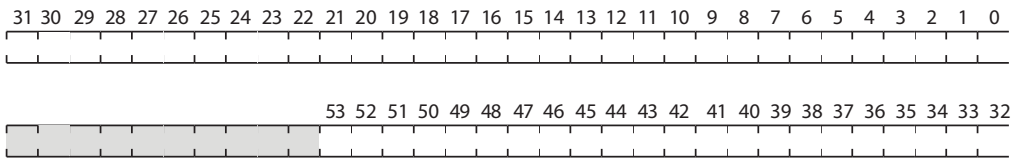
GPIO Function Select Register



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 r1 in FSEL2 register
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 .
```

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`

```
# 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!!
#
```

Details Omitted

Definitive References

ARMv6 architecture reference manual

BCM2865 peripherals document + errata

Raspberry pi schematic

Hints

Start with the simplest program.

Start with baby steps, then check it, and check it again, and check it again, ..., then take another step ...

Start by typing it in by hand; do not learn by cutting and pasting