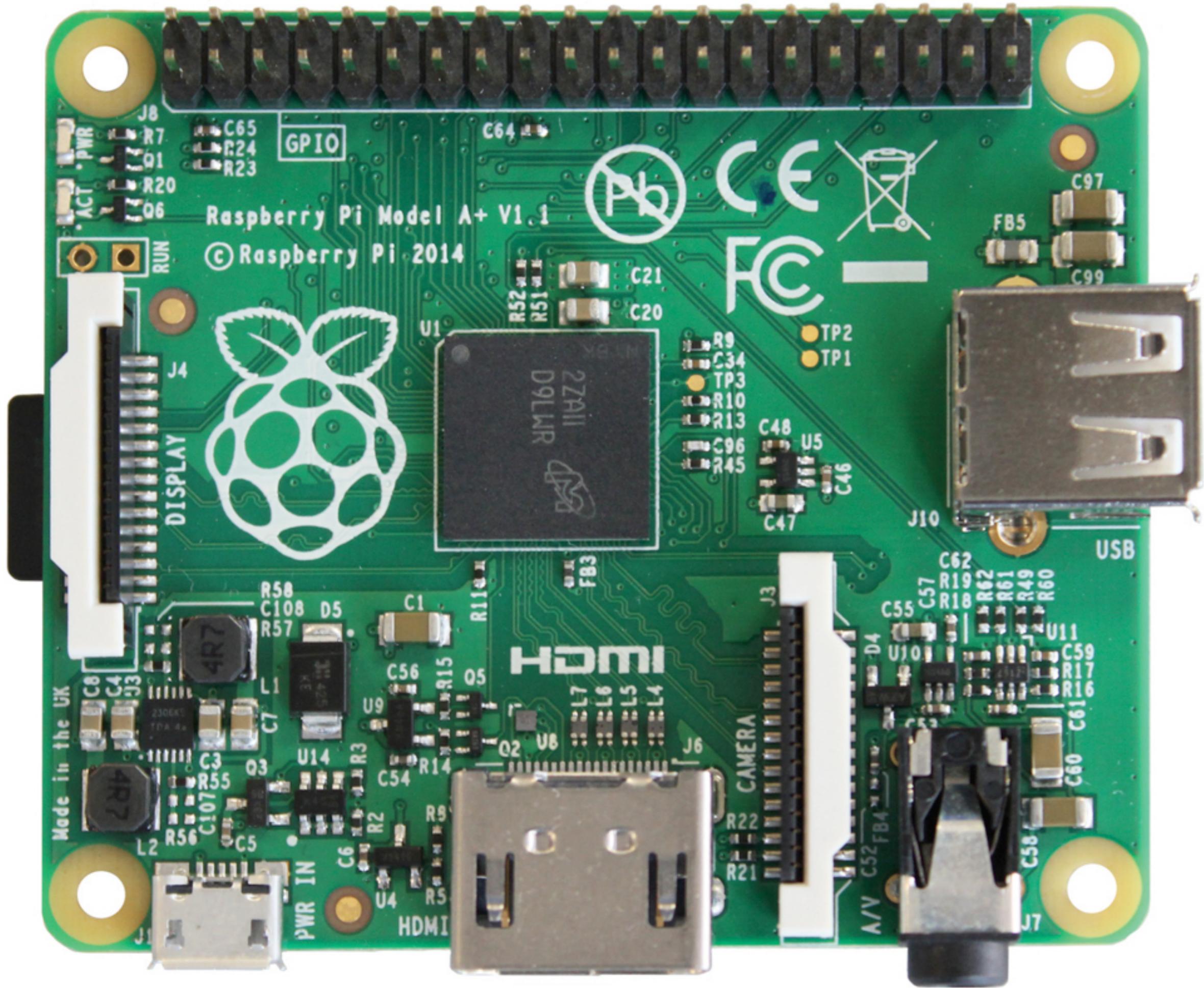
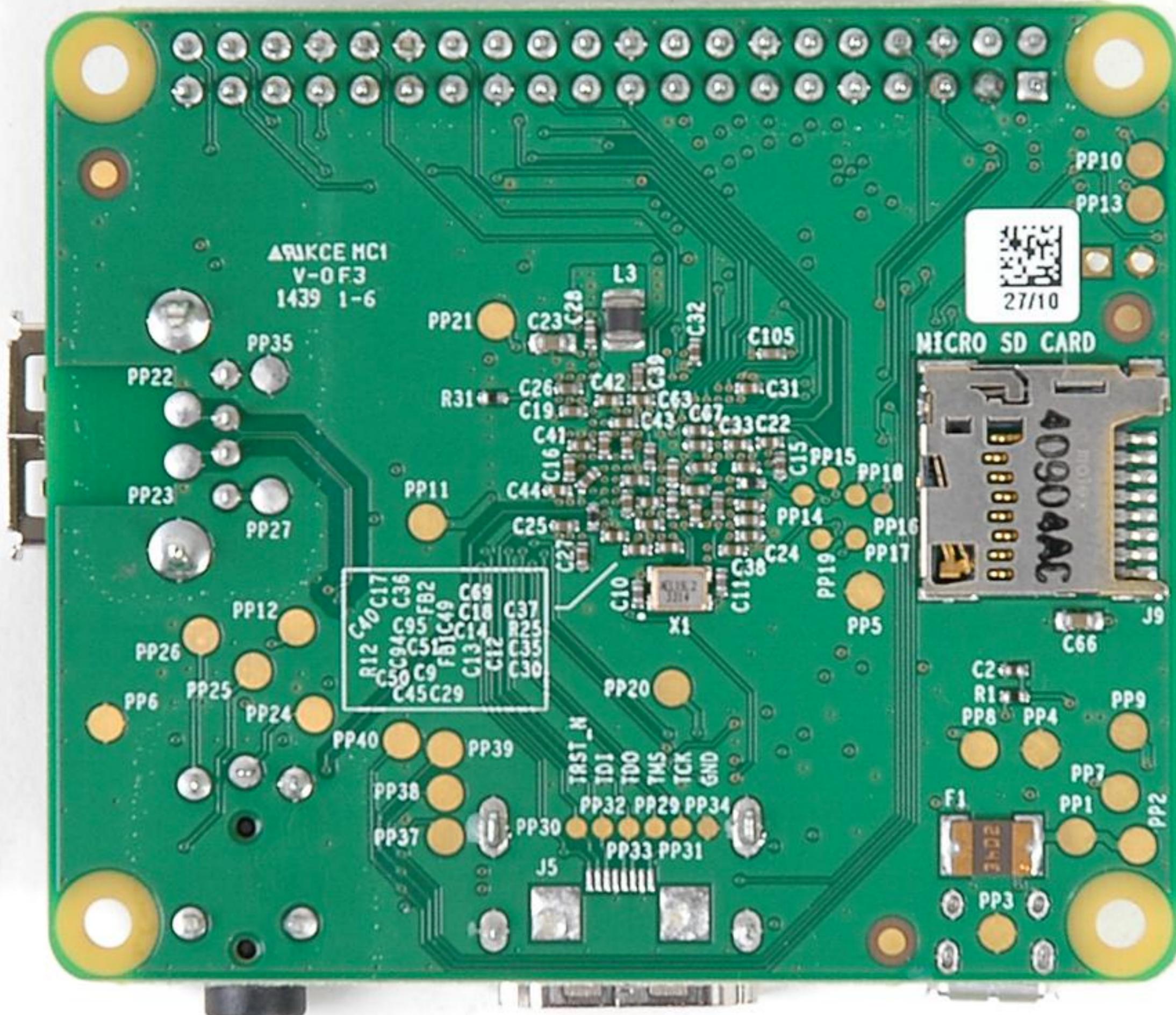


ARM

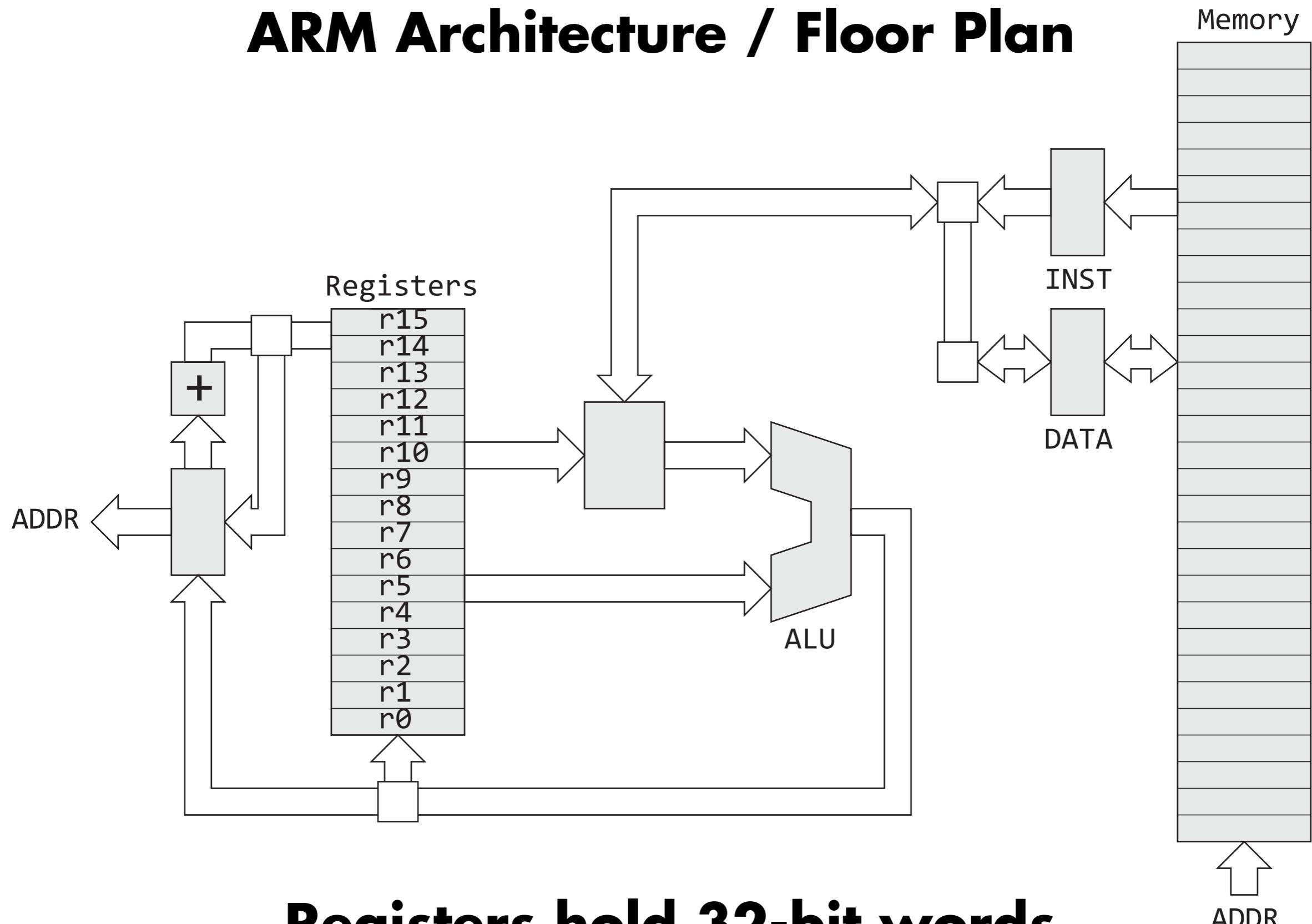
**Assembly Language
and
Machine Code**

**Goal: Blink an LED
(and hack some binary)**





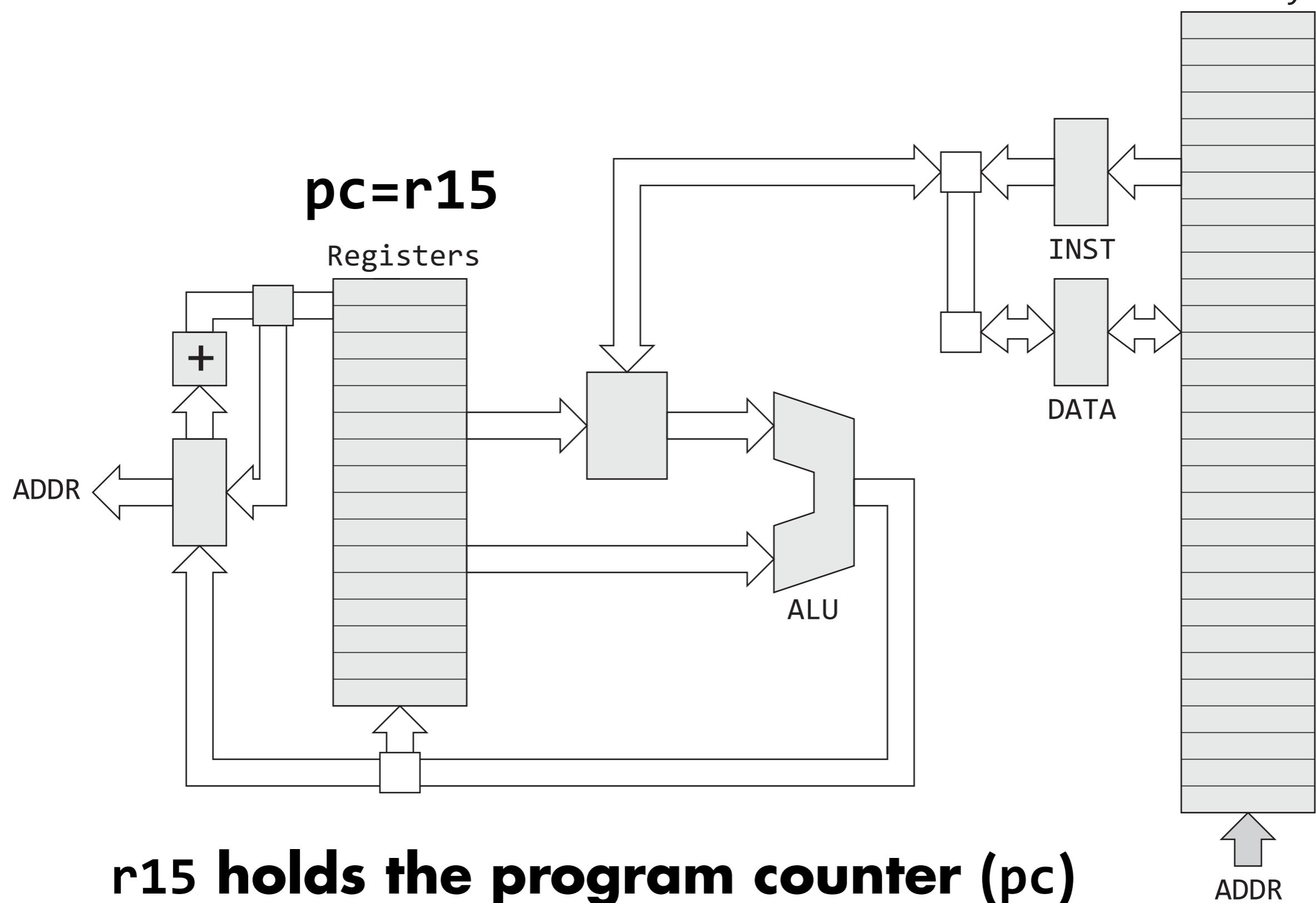
ARM Architecture / Floor Plan



Registers hold 32-bit words

Arithmetic-Logic Unit (ALU) operates on 32-bit words

Instruction Fetch



1000000000_{16}

Memory used to store both instructions and data

Storage locations are accessed using 32-bit addresses

Maximum addressable memory is 4 GB

Address refers to a byte (8-bits)

Memory Map

0000000000_{16}

10000000₁₆

Memory Map

02000000₁₆

512 MB Actual Memory ➔



Add Instruction

Meaning (defined as math or C code)

$$r_0 = r_1 + r_2$$

Assembly language (result is leftmost register)

add r0, r1, r2

Machine code (more on this later)

E0 81 00 02

```
# Assemble (.s) into 'object' file (.o)
% arm-none-eabi-as add.s -o add.o

# Create binary (.bin)
% arm-none-eabi-objcopy add.o -O binary add.bin

# Find size (in bytes)
% ls -l add.bin
-rw-r--r--+ 1 hanrahan  staff  4 add.bin

# Dump binary in hex
% hexdump add.bin
0000000: 02 00 81 e0
```

VisUAL

untitled.S - [Unsaved] - VisUAL

New

Open

Save

Settings

Tools ▾



Emulation Running

Line Issues
3 0

Execute

Reset

Step Backwards

Step Forwards

Reset to continue editing code

```
1 mov r0, #1
2 mov r1, #2
3 add r2, r0, r1
```

R0	0x1	Dec	Bin	Hex
R1	0x2	Dec	Bin	Hex
R2	0x3	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x10	Dec	Bin	Hex

Clock Cycles

Current Instruction: 1 Total: 3

CSPR Status Bits (NZCV)

0 0 0 0

Load and Store Instructions

[New](#) [Open](#) [Save](#) [Settings](#)[Tools ▾](#)[Emulation Running](#)Line Issues
4 0[Execute](#)[Reset](#)[Step Backwards](#)[Step Forwards](#)

Reset to continue editing code

```

1  ldr    r0, =0x100
2  mov    r1, #0xff
3  str    r1, [r0]
4  ldr    r2, [r0]
```

[Pointer](#) [Memory](#)

R0	0x100	Dec	Bin	Hex
R1	0xFF	Dec	Bin	Hex
R2	0xFF	Dec	Bin	Hex
R3	0x0	Dec	Bin	Hex
R4	0x0	Dec	Bin	Hex
R5	0x0	Dec	Bin	Hex
R6	0x0	Dec	Bin	Hex
R7	0x0	Dec	Bin	Hex
R8	0x0	Dec	Bin	Hex
R9	0x0	Dec	Bin	Hex
R10	0x0	Dec	Bin	Hex
R11	0x0	Dec	Bin	Hex
R12	0x0	Dec	Bin	Hex
R13	0xFF000000	Dec	Bin	Hex
LR	0x0	Dec	Bin	Hex
PC	0x14	Dec	Bin	Hex

Clock Cycles

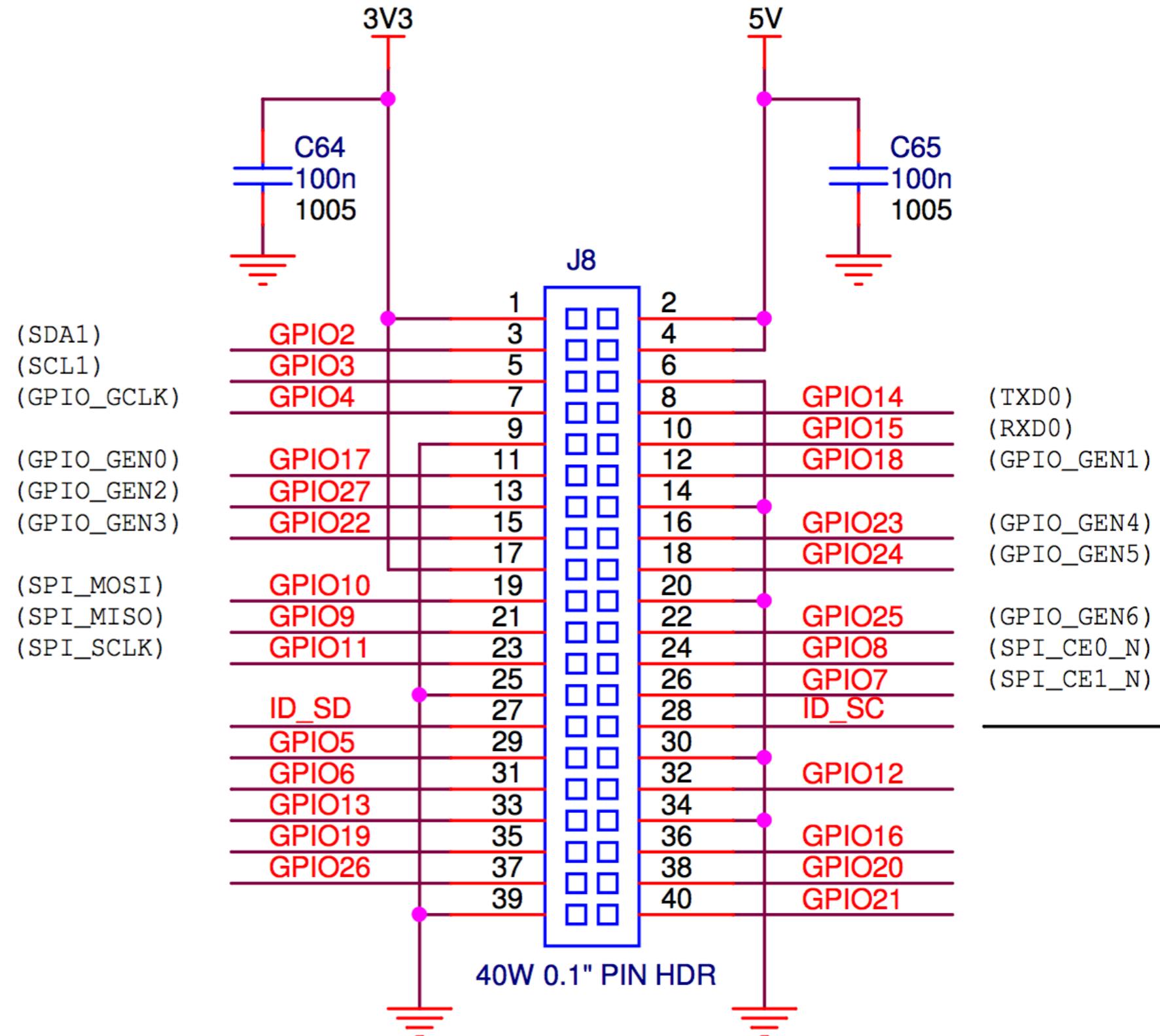
Current Instruction: 2 Total: 6

CSPR Status Bits (NZCV)

0 0 0 0

Turning on an LED

General-Purpose Input/Output (GPIO) Pins



54 GPIO Pins

BCM 20 (SPI Master-Out) at Rpi 4 Model B Rev 2

Secure | https://pinout.xyz/pinout/pin38_gpio20

Bookmarks Getting Started Bookmarks Tableau Feedly Live Ships Map - AIS...

Raspberry Pi Pinout

3v3 Power	1	5v Power	2
BCM 2 (SDA)	3	5v Power	4
BCM 3 (SCL)	5	Ground	6
BCM 4 (GPCLK0)	7	BCM 14 (TXD)	8
Ground	9	BCM 15 (RXD)	10
BCM 17	11	BCM 18 (PWM0)	12
BCM 27	13	Ground	14
BCM 22	15	BCM 23	16
3v3 Power	17	BCM 24	18
BCM 10 (MOSI)	19	Ground	20
BCM 9 (MISO)	21	BCM 25	22
BCM 11 (SCLK)	23	BCM 8 (CE0)	24
Ground	25	BCM 7 (CE1)	26
BCM 0 (ID_SD)	27	BCM 1 (ID_SC)	28
BCM 5	29	Ground	30
BCM 6	31	BCM 12 (PWM0)	32
BCM 13 (PWM1)	33	Ground	34
BCM 19 (MISO)	35	BCM 16	36
BCM 26	37	BCM 20 (MOSI)	38
Ground	39	BCM 21 (SCLK)	40

Ground DPI GPCLK JTAG 1-WIRE PCM SDIO I2C SPI UART WiringPi

Browse more HATs, pHATs and add-ons »

Arcade Bonnet
Connect joystick, buttons and speakers to your Pi

MotoZero
Control 4 motors from your Raspberry Pi

XBee Shield
Use XBee modules with the Raspberry Pi

Score:Zero
A super-simple and stylish soldering kit - makes an NES-style games controller when assembled.

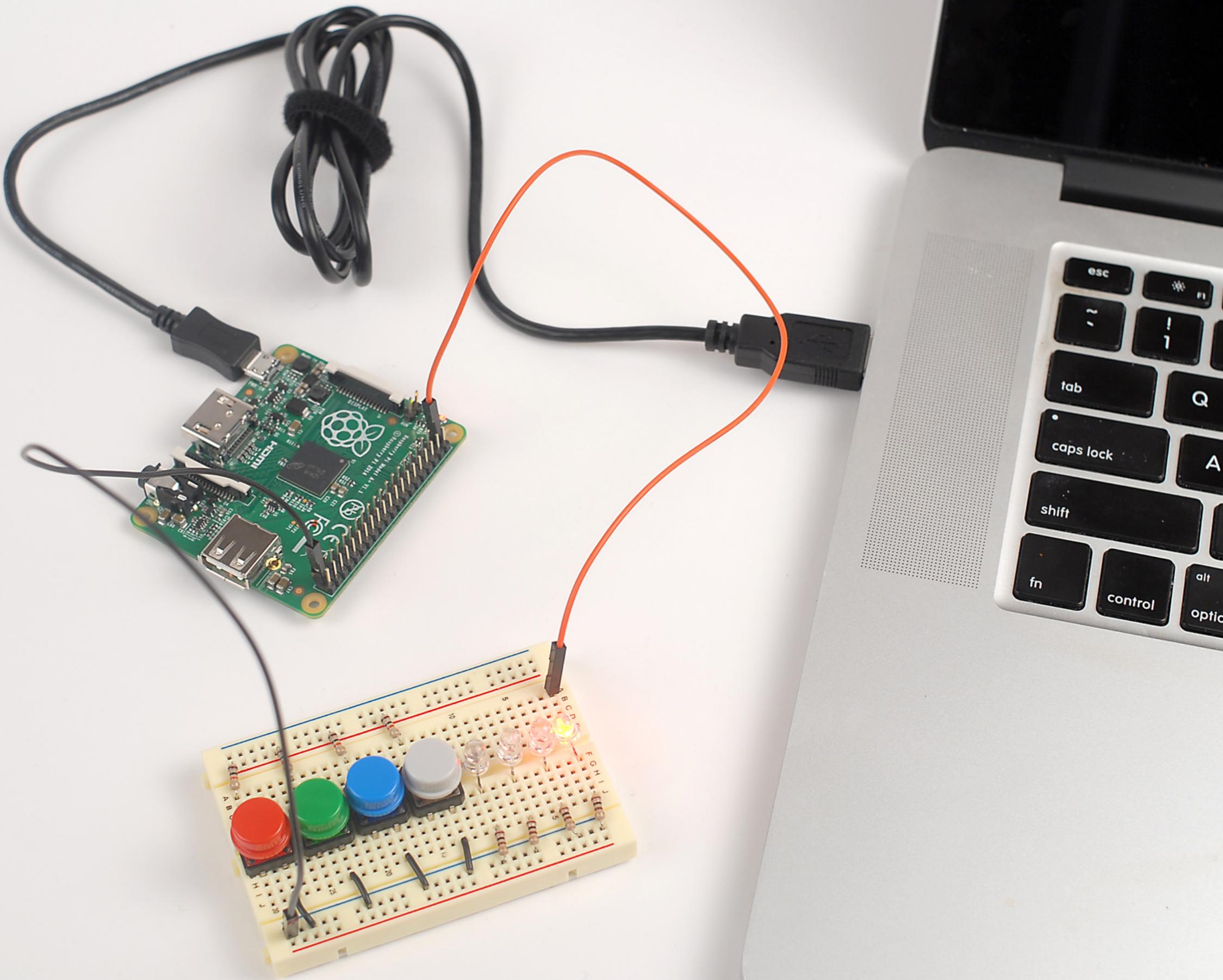
BCM 20 (SPI Master-Out)

Alt0	Alt1	Alt2	Alt3	Alt4	Alt5
PCM DIN	SMI SD12	DPI D16	I2CSL MISO	SPI1 MOSI	GPCLK0

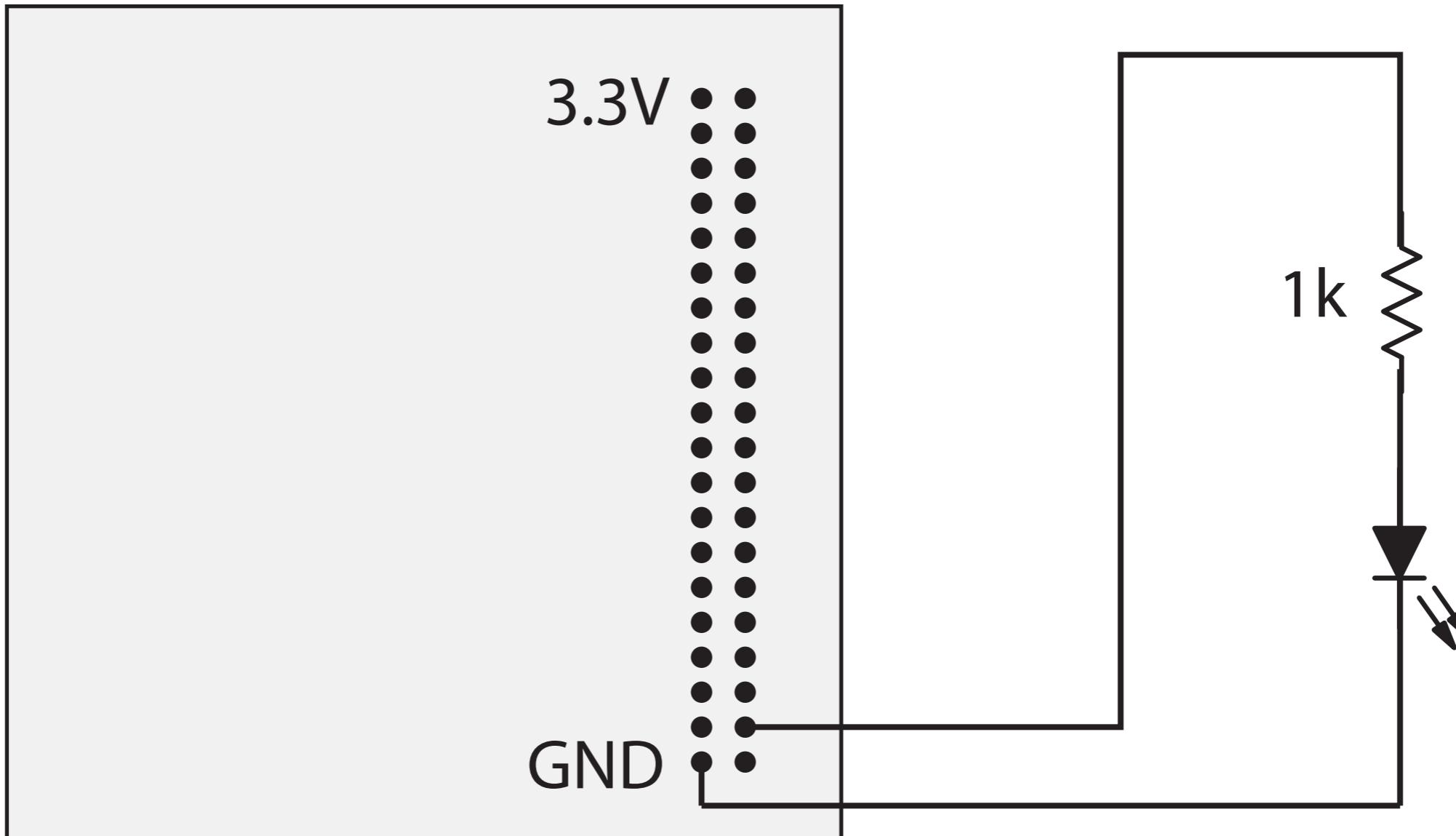
- Physical pin 38
- BCM pin 20
- Wiring Pi pin 28

DE ES FR IT TR

Spotted an error, want to add your board's pinout? Head on over to our [GitHub repository](#) and submit an Issue or a Pull Request!



Connect LED to GPIO 20



**1 -> 3.3V
0 -> 0.0V (GND)**

**GPIO Pins are called
Peripherals**

**Peripherals are Controlled
by Special Registers**

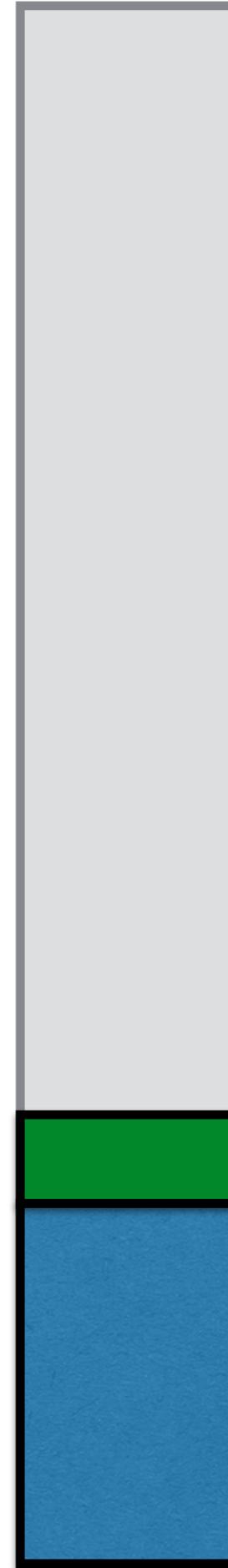
"Peripheral Registers"

Memory Map

**Peripheral registers
are mapped
into address space**

**Memory-Mapped IO
(MMIO)**

**MMIO space is above
physical memory**



10000000_{16}
4 GB

02000000_{16}
512 MB

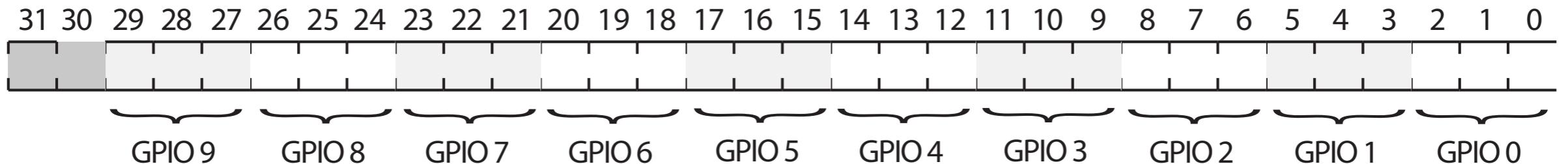
General-Purpose IO Function

GPIO Pins can be configured to be INPUT, OUTPUT, or ALTO-5

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

3 bits required to select function

GPIO Function Select Register



Function is INPUT, OUTPUT, or ALTO-5

8 functions requires 3 bits to specify

10 pins per 32-bit register (2 wasted bits)

54 GPIOs pins requires 6 registers

GPIO Function Select Registers Addresses

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	-	-

Watch out for ...

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

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

// FSEL2 = 0x20200008
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x00000008
mov r1, #1      // 1 indicates OUTPUT
str r1, [r0]    // store 1 to 0x20200008
```

GPIO Pin Output Set Registers (GPSETn)

SYNOPSIS

The output set registers are used to set a GPIO pin. The SET{n} field defines the respective GPIO pin to set, writing a “0” to the field has no effect. If the GPIO pin is being used as an input (by default) then the value in the SET{n} field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations

Bit(s)	Field Name	Description	Type	Reset
31-0	SETn (n=0..31)	0 = No effect 1 = Set GPIO pin <i>n</i>	R/W	0

Table 6-8 – GPIO Output Set Register 0

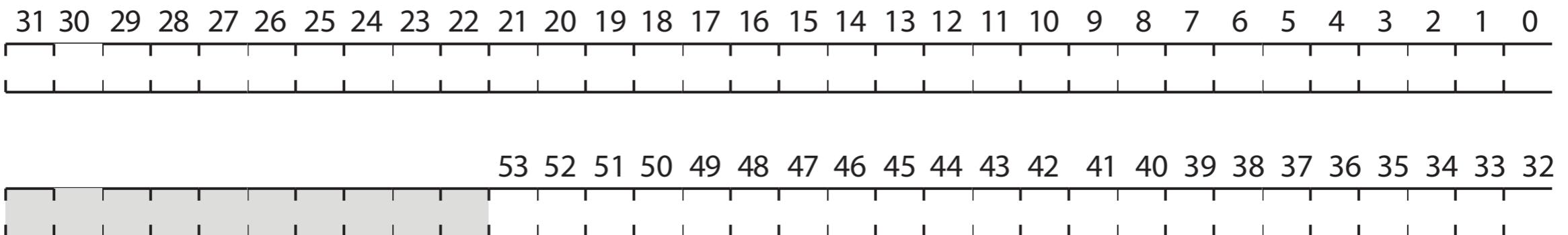
Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin <i>n</i> .	R/W	0

Table 6-9 – GPIO Output Set Register 1

GPIO Function SET Register

20 20 00 1C : GPIO SET0 Register

20 20 00 20 : GPIO SET1 Register



Notes

- 1. 1 bit per GPIO pin**
- 2. 54 pins requires 2 registers**

...

```
// SET0 = 0x2020001c
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x0000001c
mov r1, #1
lsl r1, #20 // bit 20 = 1<<20
str r1, [r0] // store 1<<20 to 0x2020001c
```

```
// loop forever
loop:
b loop
```

What to do on your laptop

Assemble language to machine code

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

Create binary from object file

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

What to do on your laptop

Insert SD card - Volume mounts

% ls /Volumes/

BARE Macintosh HD

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.  
# Power LED (Red) should be on.  
#  
# Raspberry pi boots. ACT LED (Green)  
# flashes, and then is turned off  
#  
# LED connected to GPIO20 turns on!!
```



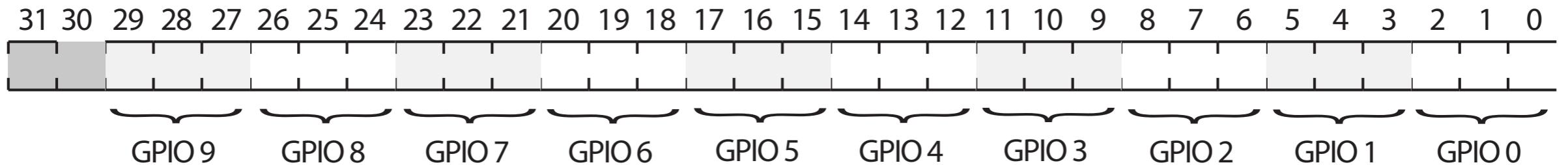
General-Purpose IO Function

GPIO Pins can be configured to be
INPUT, OUTPUT, or ALT0-ALT5

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

3 bits required to select function

GPIO Function Select Register



Function is INPUT, OUTPUT, or ALT0-ALT5

8 functions requires 3 bits to specify

10 pins per 32-bit register (2 wasted bits)

54 GPIOs pins requires 6 registers

GPIO Function Select Registers Addresses

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	-	-

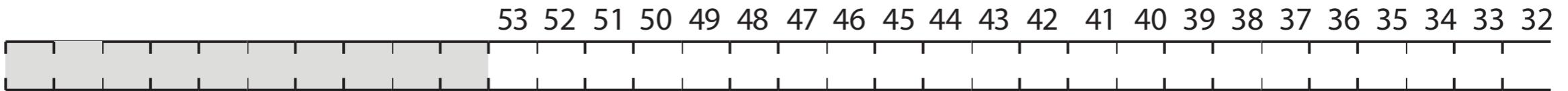
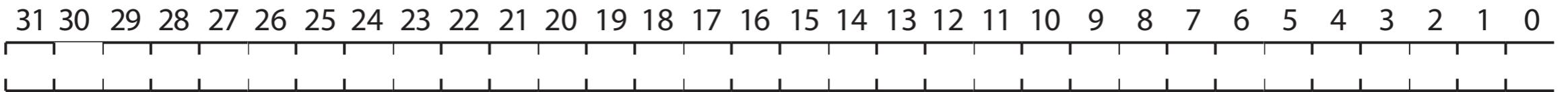
Watch out for ...

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

```
...
// "set" GPIO20 (output 1 = 3.3V)

// SET0 = 0x2020001c
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x0000001c
mov r1, #1
lsl r1, #20 // bit 20 = 1<<20
str r1, [r0] // store 1<<20 to 0x2020001c
```



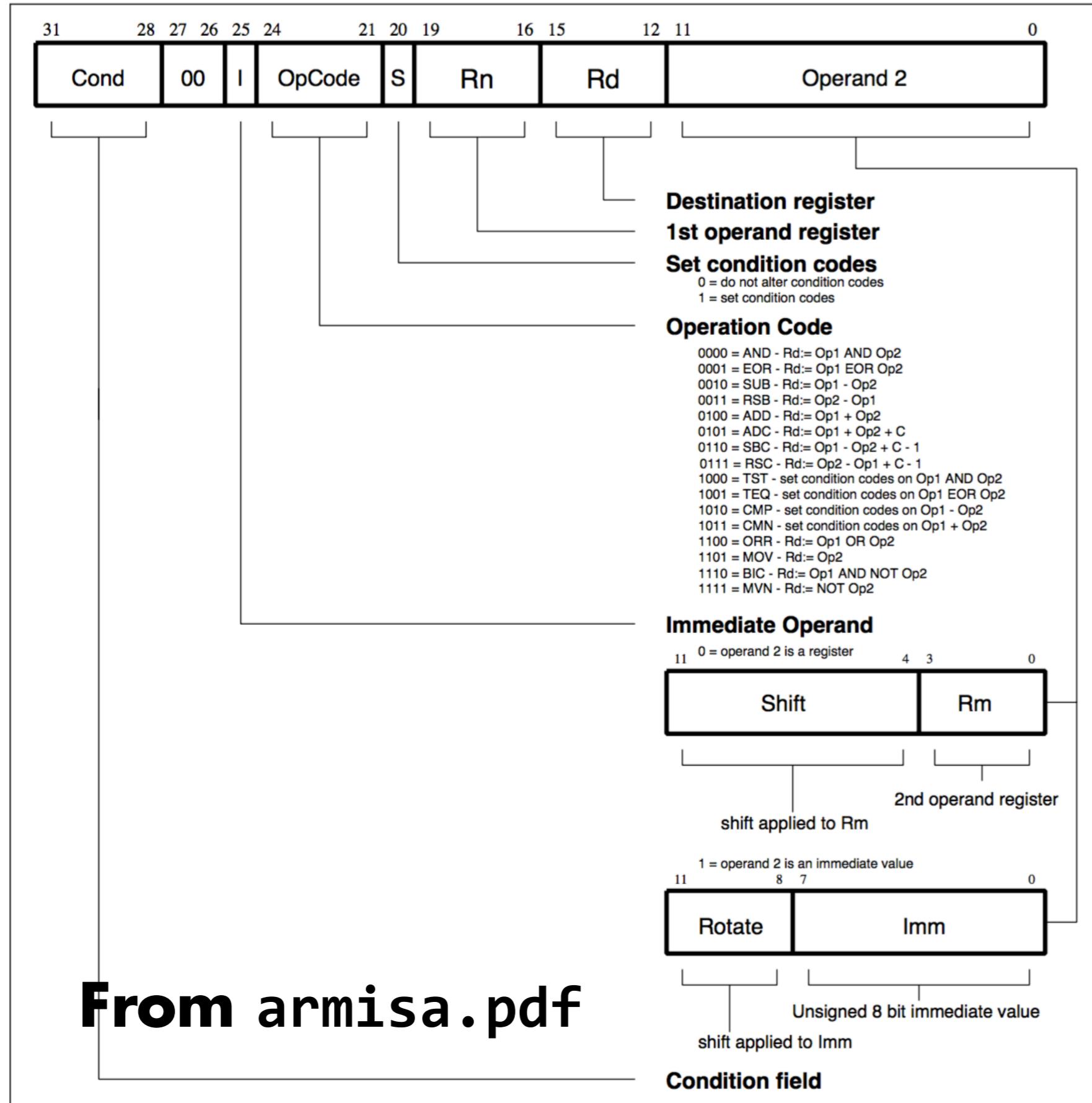
```
ldr r0, SET0
ldr r1, #(1<<20)
ldr r1, #(1<<21)
```

```
// Is the LED connected to GPIO20 on?
// Is the LED connected to GPIO21 on?
```

3 Types of Instructions

- 1. Data processing instructions**
- 2. Loads from and stores to memory**
- 3. Conditional branches to new program locations**

Data Processing Instructions



From armisa.pdf

Figure 4-4: Data processing instructions

```
# data processing instruction  
#  
# ra = rb op rc
```

Immediate mode instruction

Set condition codes

1110 00 i **oooo** s **bbbb** **aaaa** **cccc cccc cccc**

Data processing instruction

Always execute the instruction

Assembly	Code	Operations
AND	0000	$ra=rb\&rc$
EOR (XOR)	0001	$ra=rb^rc$
SUB	0010	$ra=rb - rc$
RSB	0011	$ra=rc - rb$
ADD	0100	$ra=rb+rc$
ADC	0101	$ra=rb+rc+CARRY$
SBC	0110	$ra=rb-rc+(1-CARRY)$
RSC	0111	$ra=rc-rb+(1-CARRY)$
TST	1000	$rb\&rc$ (ra not set)
TEQ	1001	rb^rc (ra not set)
CMP	1010	$rb-rc$ (ra not set)
CMN	1011	$rb+rc$ (ra not set)
ORR (OR)	1100	$ra=rb rc$
MOV	1101	$ra=rc$
BIC	1110	$ra=rb\&\sim rc$
MVN	1111	$ra=\sim rc$

```
# data processing instruction  
#   ra = rb op rc  
#
```

		op	rb	ra	rc		
1110	00	i	oooo	s bbbb	aaaa	cccc	cccc cccc

		add	r1	r0			
1110	00	i	0100	s 0001	0000	cccc	cccc cccc

```
# data processing instruction  
#   ra = rb op #imm  
# #imm = uuuu uuuu
```

	add	r1	r0	imm
1110 00	1	0100	0 0001	0000 0000 uuuu uuuu

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

```
# i=1, s=0  
#  
# As in immediately available,  
# i.e. no need to fetch from memory
```

```
# data processing instruction  
#   ra = rb op #imm  
# #imm = uuuu uuuu
```

	add	r1	r0	imm
1110 00 1	0100	0 0001	0000 0000	uuuu uuuu

add r0, r1, #1

	add	r1	r0	#1
1110 00 1	0100	0 0001	0000 0000	0000 0001

```
# data processing instruction  
#   ra = rb op #imm  
# #imm = uuuu uuuu
```

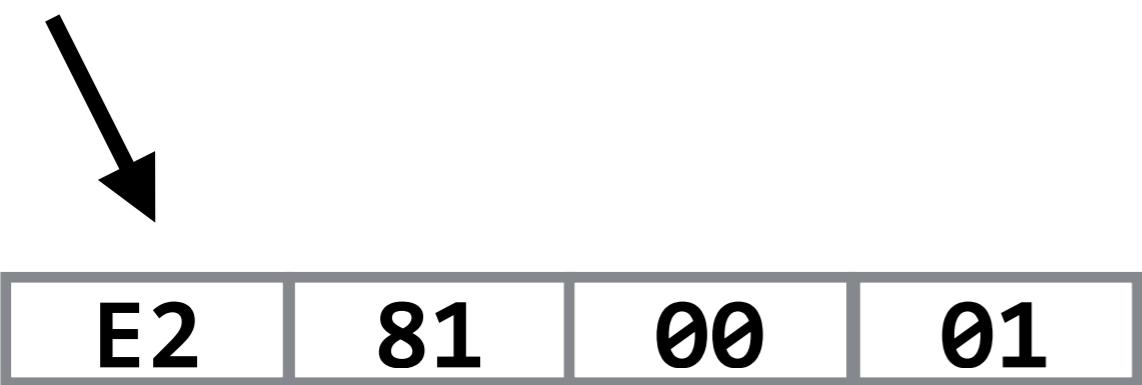
	add	r1	r0	imm
1110 00 1	0100	0 0001	0000 0000	uuuu uuuu

add r0, r1, #1

	add	r1	r0	#1
1110 00 1	0100	0 0001	0000 0000	0000 0001

1110 0010 1000 0001 0000 0000 0000 0000 0001
E 2 8 1 0 0 0 0 1

most-significant-byte (MSB)

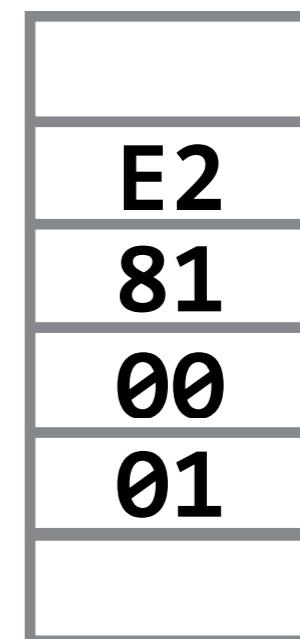


ADDR+3

ADDR+2

ADDR+1

ADDR



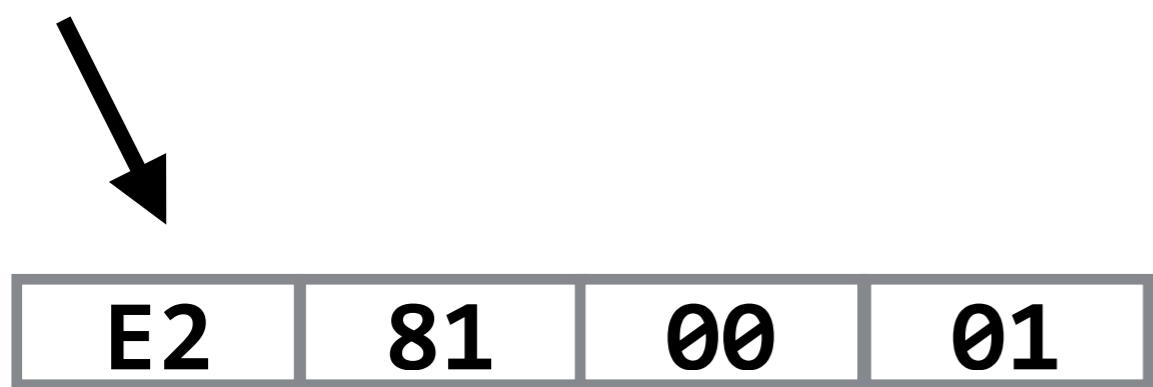
least-significant-byte (LSB)

little-endian
(LSB first)

Intel is little-endian

ARM can do either, Pi configured to be little endian

most-significant-byte (MSB)

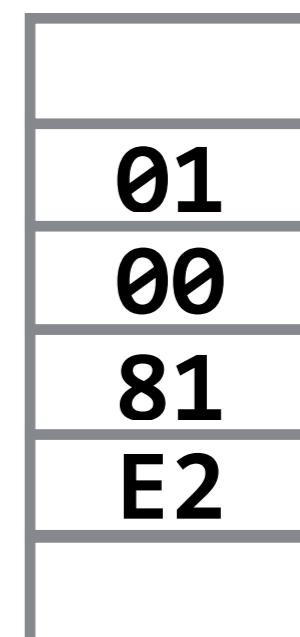


ADDR+3

ADDR+2

ADDR+1

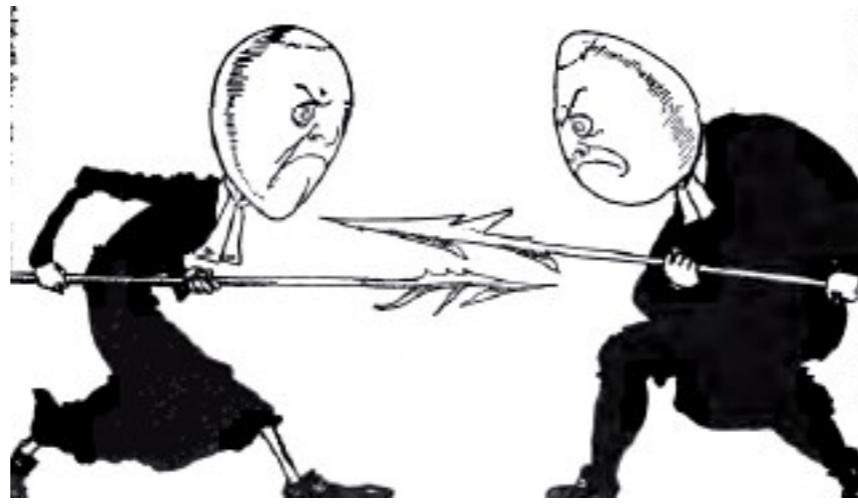
ADDR



least-significant-byte (LSB)

**big-endian
(MSB first)**

Old Macintoshes, Sun SPARC are big endian
Networks use big endian



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

The Fun Begins ...

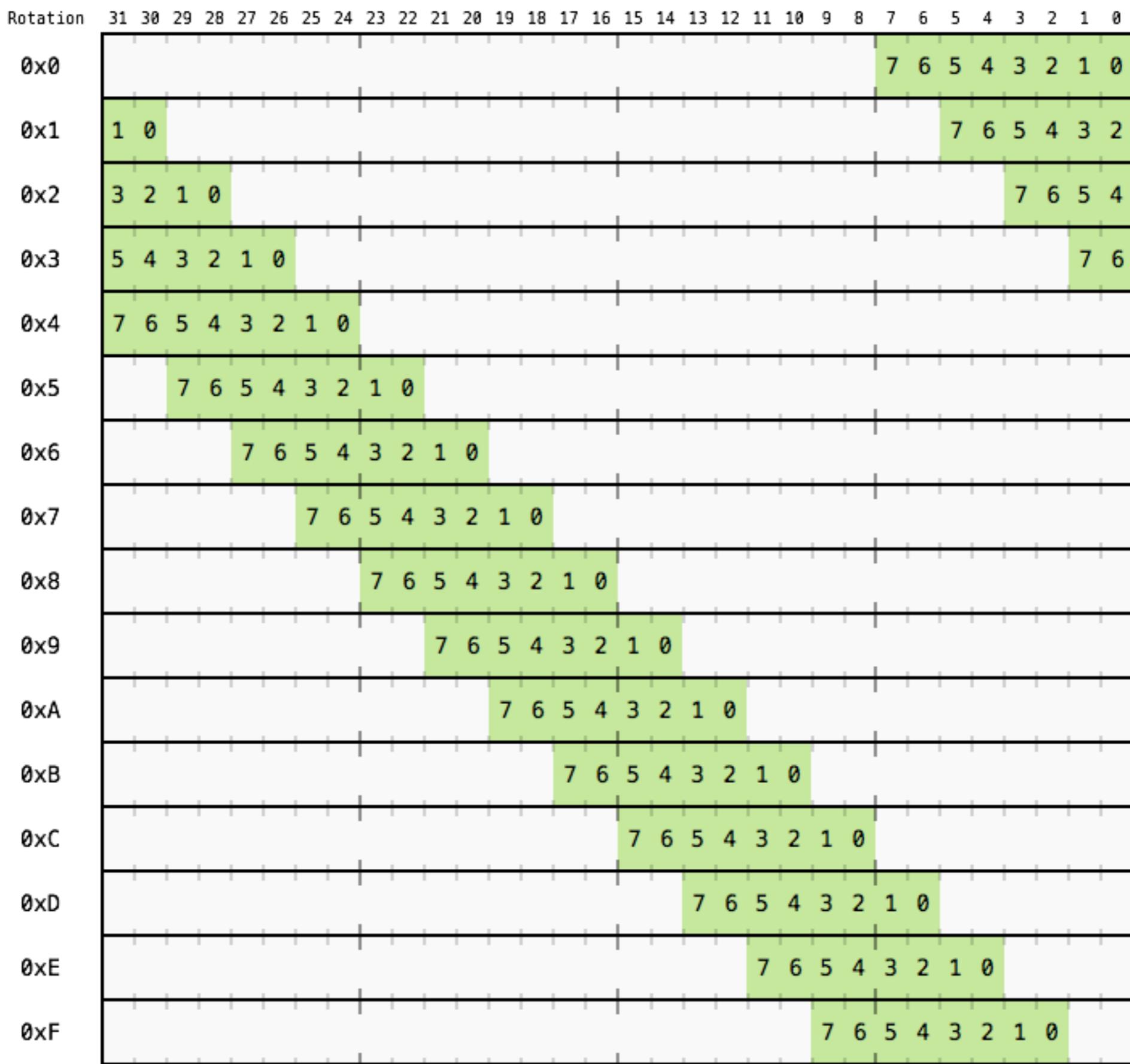
Labs I

- Assemble Raspberry Pi Kit
- Lab assignments sent out tonight
- Read lab I instructions (now online)
- Install tool chain

Assignment I

- Larson scanner

Rotate Bits Right (ROR)



```
# data processing instruction  
#   ra = rb op imm  
# imm = (uuuu uuuu) ROR (2*rrrr)
```

	op	rb	ra	ror	imm
1110 00 1	oooo	0 bbbb	aaaa	rrrr	uuuu uuuu

```

# data processing instruction
#   ra = rb op imm
# imm = (uuuu uuuu) ROR (2*rrrr)

```

	op	rb	ra	ror	imm
1110 00 1	oooo	0 bbbb	aaaa	rrrr	uuuu uuuu

add r0, r1, #0x1000

add	r1	r0	0x01>>>2*8
-----	----	----	------------

1110 00 1	0100	0 0001	0000	1000 0000 0001
-----------	------	--------	------	----------------

$0x01>>>16$

0000 0000 0000 0000 0000 0000 0000 0001

0000 0000 0000 0001 0000 0000 0000 0000

...

```
// SET1 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0
```

```

# data processing instruction
#   ra = rb op imm
# imm = (uuuu uuuu) ROR (2*ieee)

```

	op	rb	ra	ror	imm
1110 00 1	oooo	0 bbbb	aaaa	iiii	uuuu uuuu

add r0, r1, #0x1000

add	r1	r0	0x01>>>2*8
-----	----	----	------------

1110 00 1	0100	0 0001	0000	1000 0000 0001
-----------	------	--------	------	----------------

1110 0010 1000 0001 0000 1000 0000 0001	E 2 8 1 0 8 0 1
---	-----------------

Determine the machine code for

sub r7, r5, #0x300

imm = (uuuu uuuu) ROR (2*ieee)

Remember that ra is the result

	op	rb	ra	ror	imm
1110 00 1	oooo	0 bbbb	aaaa	iiii	uuuu uuuu

// What is the machine code?

Assembly	Code	Operations
hint: SUB	0010	ra=rb-rc

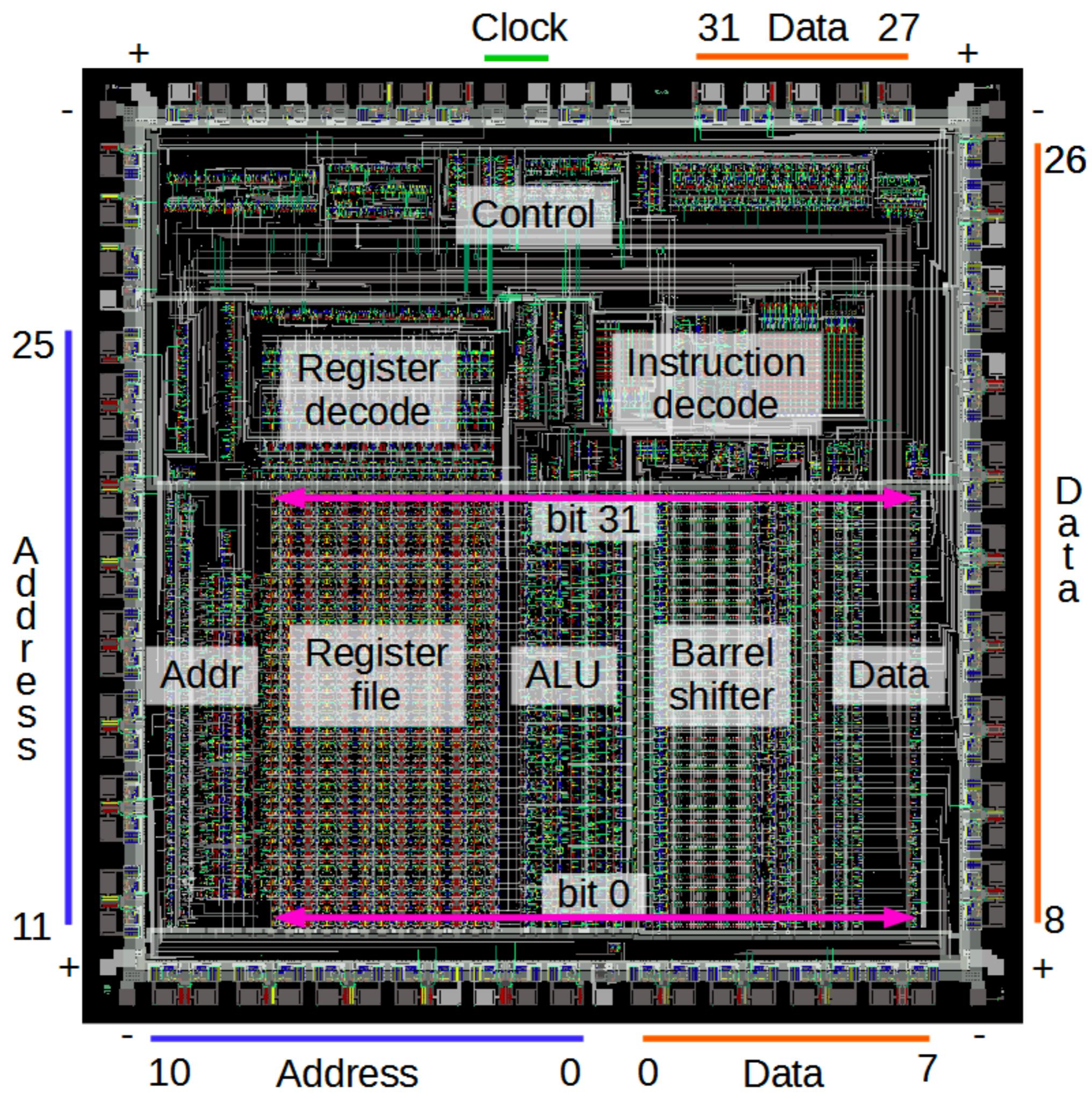
```
# data processing instruction  
#   ra = rb op imm  
# imm = uuuu uuuu ROR (2*ieee)
```

	op	rb	ra	ror	imm
1110 00 1	oooo	0 bbbb	aaaa	iiii	uuuu uuuu

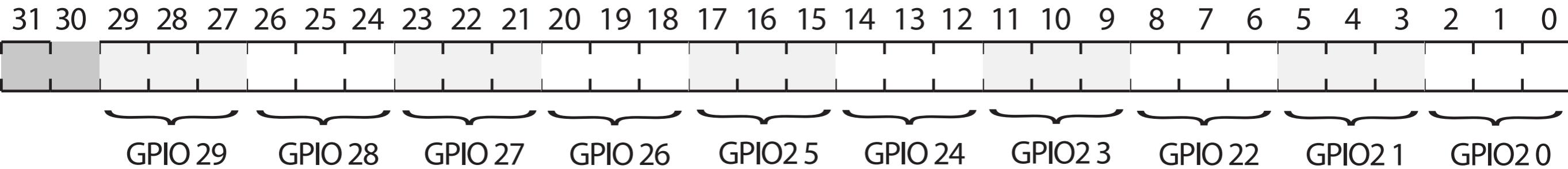
```
sub r7, r5, #0x300
```

	sub	r5	r7	#0x03>>>24
1110 00 1	0010	0 0101	0111	1100 0000 0011

1110 0010 0100 0101 0111 1100 0000 0011
E 2 4 5 7 C 0 3



Manipulating Bit Fields



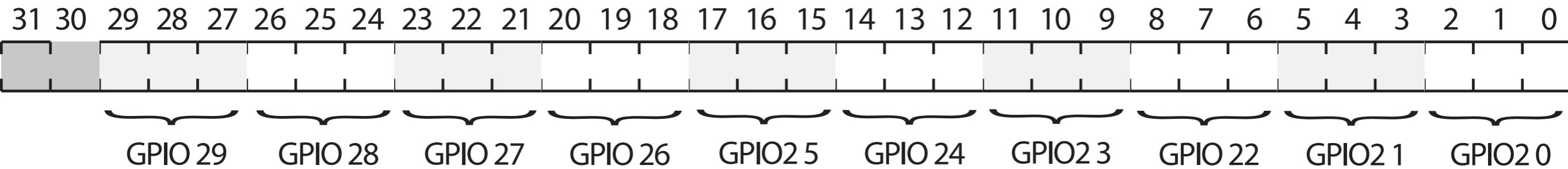
// Set GPIO 20 and 21 both to OUTPUT

```
mov r1, #1  
orr r1, #(1<<3)  
str r1, [r0]
```

// What value is in FSEL2 now?

// What mode is GPIO 20 set to now?

// What mode is GPIO 21 set to now?



// Set GPIO 20 to OUTPUT

mov r1, #1

str r1, [r0]

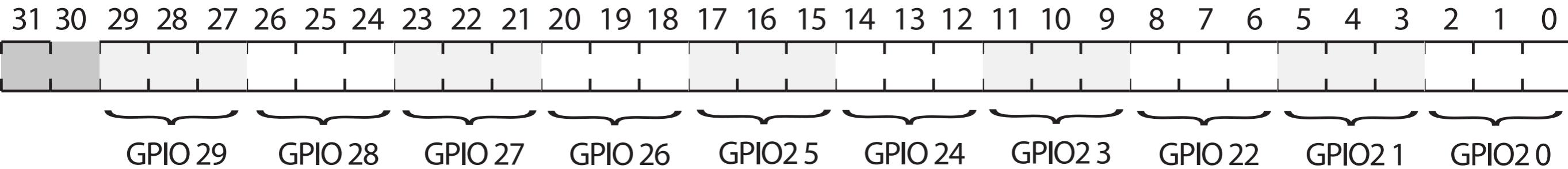
// Set GPIO 21 to OUTPUT

mov r1, #(1<<3)

str r1, [r0]

// What value is in FSEL2 now?

// What mode is GPIO 20 set to now?



// Set GPIO 20 to OUTPUT

```
mov r1, #1  
str r1, [r0]
```

...

// Preserve GPIO20, set GPIO21 to OUTPUT

```
ldr r1, [r0]  
and r1, #~(0x7<<3)  
orr r1, #(0x1<<3)  
str r1, [r0]
```

// What value is in FSEL2 now?

```
// LDR FSEL2, GPIO20 is OUTPUT
ldr r1, [r0]
0000 0000 0000 0000 0000 0000 0000 0000 0001
// 0x7
0000 0000 0000 0000 0000 0000 0000 0000 0111
// 0x7<<3
0000 0000 0000 0000 0000 0000 0000 0011 1000
// ~(0x7<<3)
1111 1111 1111 1111 1111 1111 1111 1100 0111
and r1, #~(0x7<<3)
0000 0000 0000 0000 0000 0000 0000 0000 0001
orr r1, #(0x1<<3)
0000 0000 0000 0000 0000 0000 0000 0000 1001
```

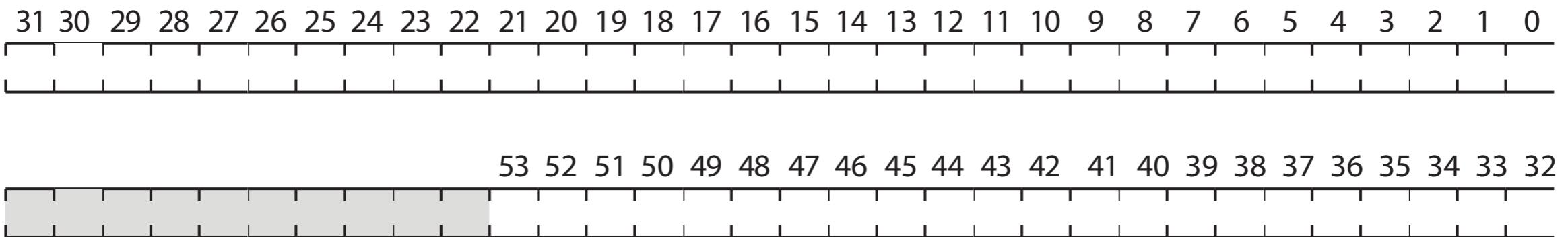
Blink

```
// This is tedious ...
```

```
mov r0, #0x2000000 // #(0x20>>>8)
orr r0, #0x0020000 // #(0x20>>>16)
orr r0, #0x0000008
```

```
// Alternative is to use ldr
ldr r0, FSEL2
```

```
FSEL2: .word 0x20200008
```



```
mov r1, #(1<<20)
```

```
// Turn on LED connected to GPIO20
ldr r0, SET0
str r1, [r0]
```

```
// Turn off LED connected to GPIO20
ldr r0, CLR0
str r1, [r0]
```

```
// Why SET and CLR?
```

```
// Configure GPIO 20 for OUTPUT
```

```
loop:
```

```
// Turn on LED
```

```
// Turn off LED
```

```
b loop
```

Loops and Condition Codes

```
// define constant  
.equ DELAY, 0x3f0000
```

```
mov r2, #DELAY
```

```
loop:
```

```
subs r2, r2, #1 // s set cond code
```

```
bne loop      // branch if r2 != 0
```

Condition Codes

Z - Result is 0

N - Result is <0

C - Carry generated

V - Arithmetic overflow

Carry and overflow will be covered later

```
# data processing instruction  
#   ra = rb op rc|imm  
#  
# i - immediate  
# s - set condition code  
#
```

op	rb	ra						
1110 00	i	oooo	s	bbbb	aaaa	cccc	cccc	cccc

Branch Instructions

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

```
# branch  
cond      addr  
cccc 101L 0000 0000 0000 0000 0000 0000
```

```
b = bal = branch always  
cond      addr  
1110 101L 0000 0000 0000 0000 0000 0000
```

```
bne  
cond      addr  
0001 101L 0000 0000 0000 0000 0000 0000
```

branch

cond

cccc 101L

addr

0000 0000 0000 0000 0000 0000

b = bal =

cond

1110 101L

branch always

addr

0000 0000 0000 0000 0000 0000

bne

cond

0001 101L

addr

0000 0000 0000 0000 0000 0000

The “link” bit — you’ll cover this later,
when you learn about how functions work

Bringing It All Together

```
$ cpp -P blink.s | arm-none-eabi-as -o blink.o  
$ arm-none-eabi-objcopy blink.o -O binary blink.bin  
$ rpi-install blink.bin  
  
$ xxd blink.bin > blink.dump  
// edit blink.dump  
$ xxd -r blink.dump > blink-fast.bin  
  
$ rpi-install blink-fast.bin
```

Orthogonal Instructions

Any operation

Register vs. immediate operands

All registers the same**

Predicated/conditional execution

Set or not set condition code

Orthogonality leads to composability

Summary

Rarely, but sometimes, write assembly

Understanding how processors represent and execute instructions lets you understand how languages work

Reading assembly allows you to *really* know what the processor is doing: we'll see how compilers can trick you!

Finite space leads to tradeoffs and careful design: what if ARM had 32 registers?

Definitive References

BCM2865 peripherals document + errata

Raspberry Pi schematic

ARMv6 architecture reference manual

see Resources on cs107e.github.io