CS 107e
Lecture 2: ARM

Friday, April 6, 2018

Computer Systems from the Ground Up
Spring 2018
Stanford University
Computer Science Department

Lecturer: Chris Gregg

```
// turn on LED connected to GPIO20

// configure GPIO 20 for output
//  FSEL2 = 0x20200008
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x00000008
mov r1, #1    // indicates OUTPUT
str r1, [r0]  // store 1 to address 0x20200008

// set GPI020 to 1
//  SET0 = 0x2020001c
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x0000001c
mov r1, #1
lsl r1, #20   // 1<<20
str r1, [r0]  // store 1<<20 to address 0x2020001c

// loop forever
loop:
b loop
```

Logistics

- Labs start on Tuesday

Today's Topics

- Introduction to your Raspberry Pi
- A bit of computing history: Babbage's Analytical Engine
- "Running a program" on a modern processor
- The ARM memory map
- The ARM Architecture / Floor Plan
  - Instruction Fetch
  - The Arithmetic Logic Unit (ALU)
  - The **`mov`** and **`add`** instructions
  - Load (**`ldr`**) and store (**`str`**) instructions
- Turning on an LED
  - GPIO pins

Today's Overall Goal

Turning on an LED

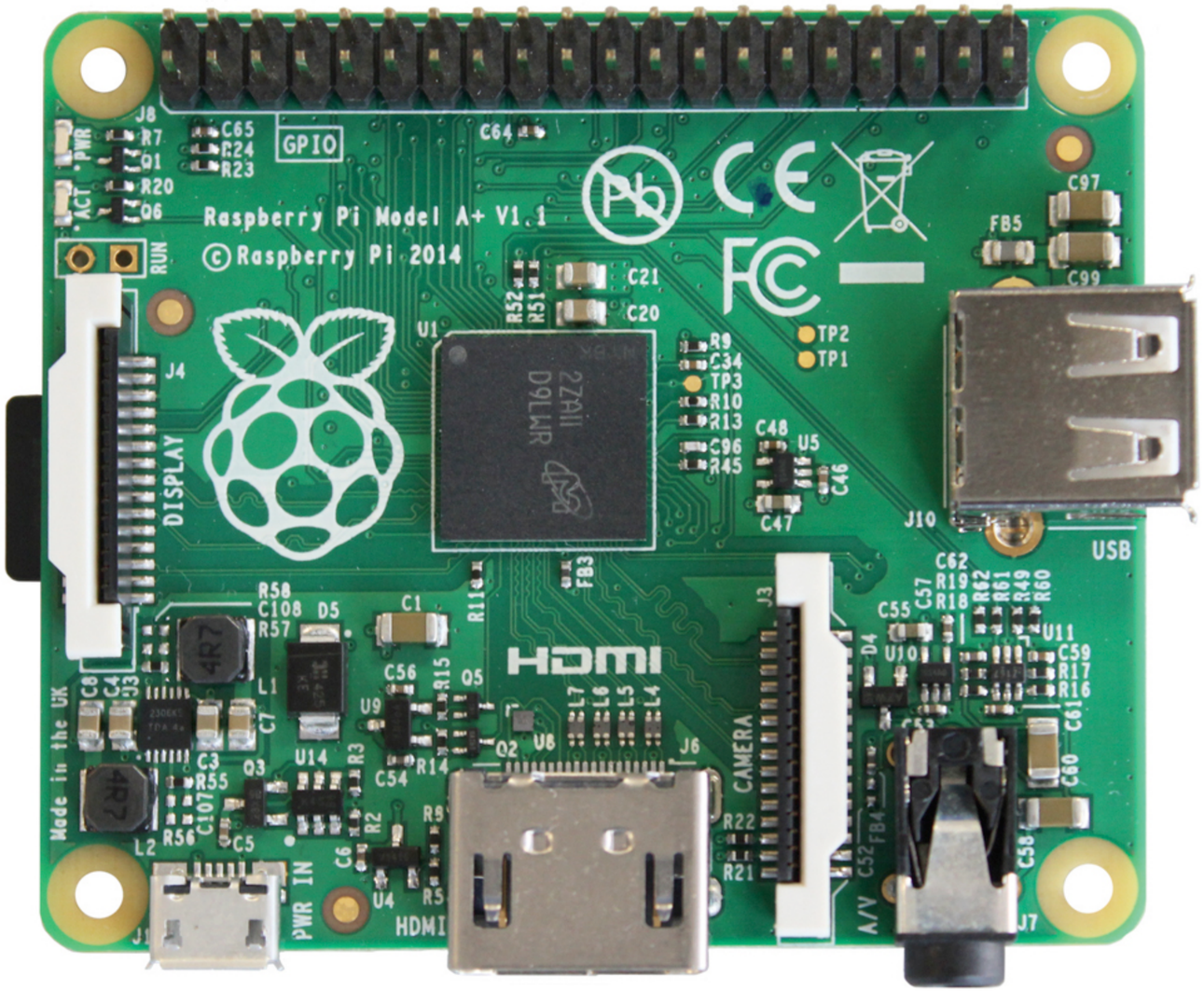
Today's Overall Goal

To turn on an LED programmatically:

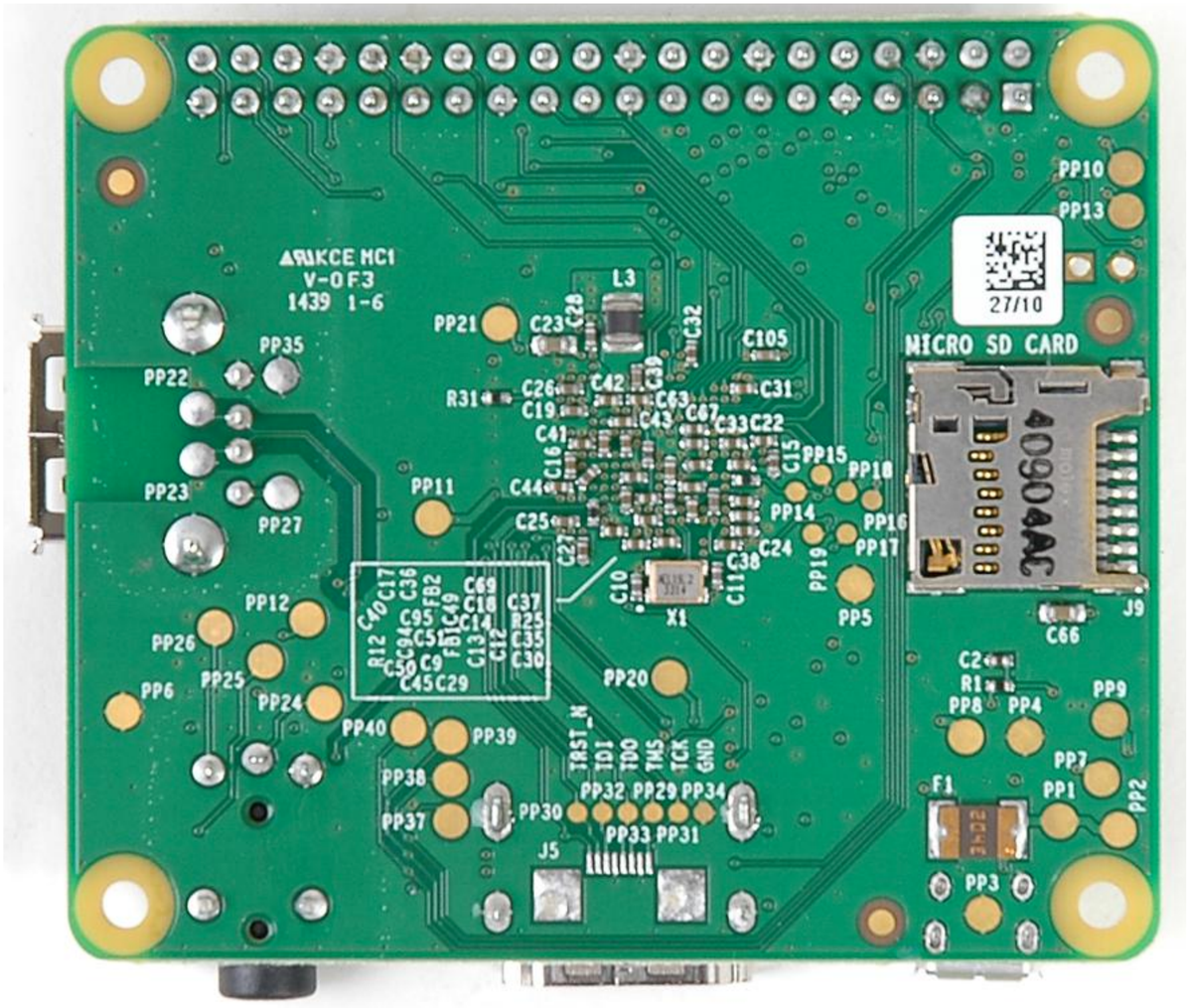
1. The Raspberry Pi (raspi) must be able to run *instructions* that can change the *voltage* on a *pin* from 0 volts to 3.3 volts.
2. We must be able to load those instructions onto the raspi and it must execute those instructions.
3. We must connect an LED between the pin that has its voltage changed and *ground*.

Step 1 above is not trivial! Today we will talk a little bit about the details of how the raspi executes instructions.
Step 2 is a matter of good tools — we have those tools and you'll start using them in lab 1.
Step 3 is actually pretty easy :)

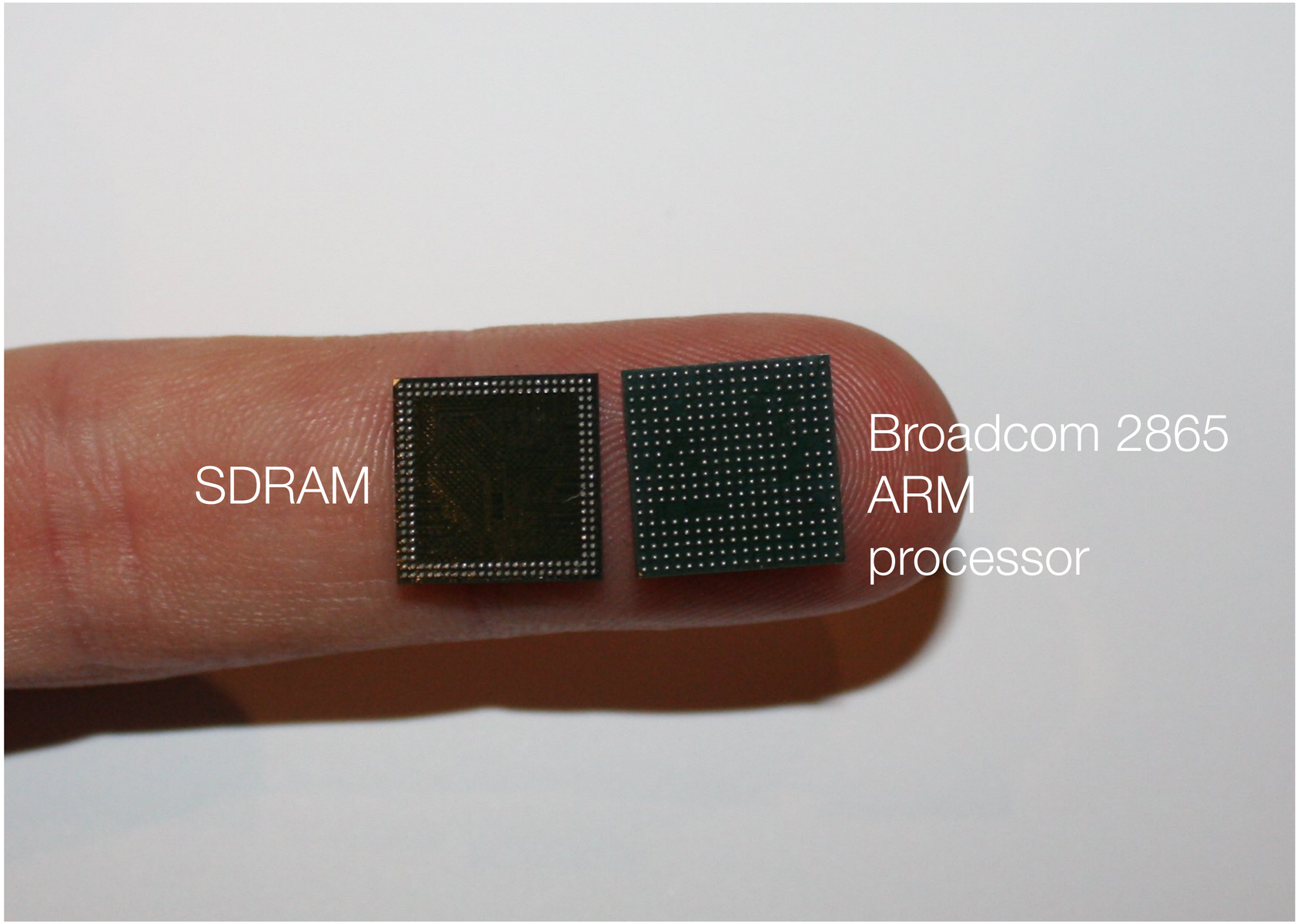
The Raspberry Pi



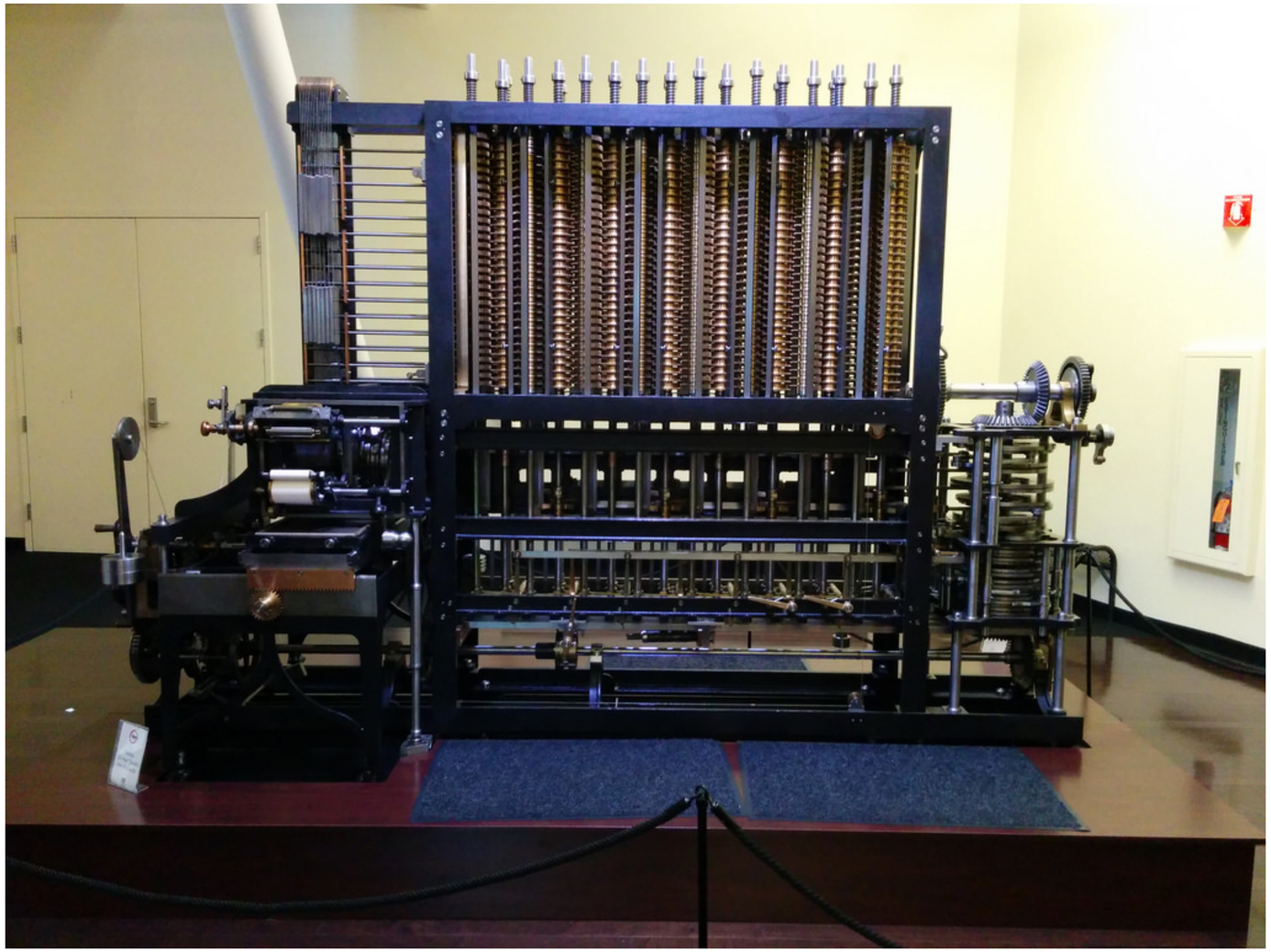
The Raspberry Pi



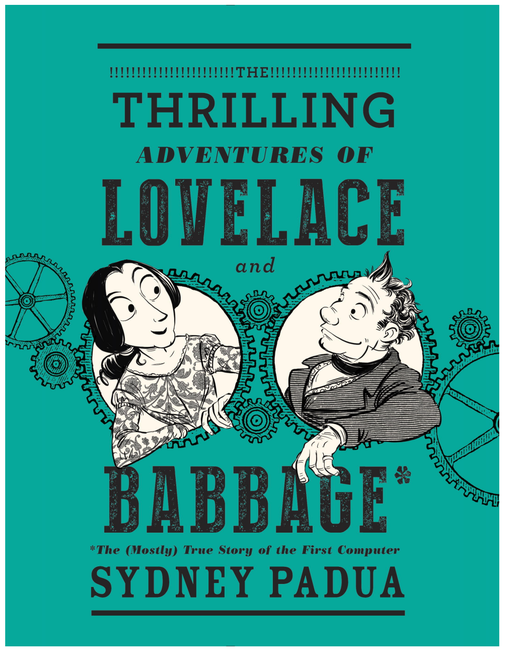
The Raspberry Pi

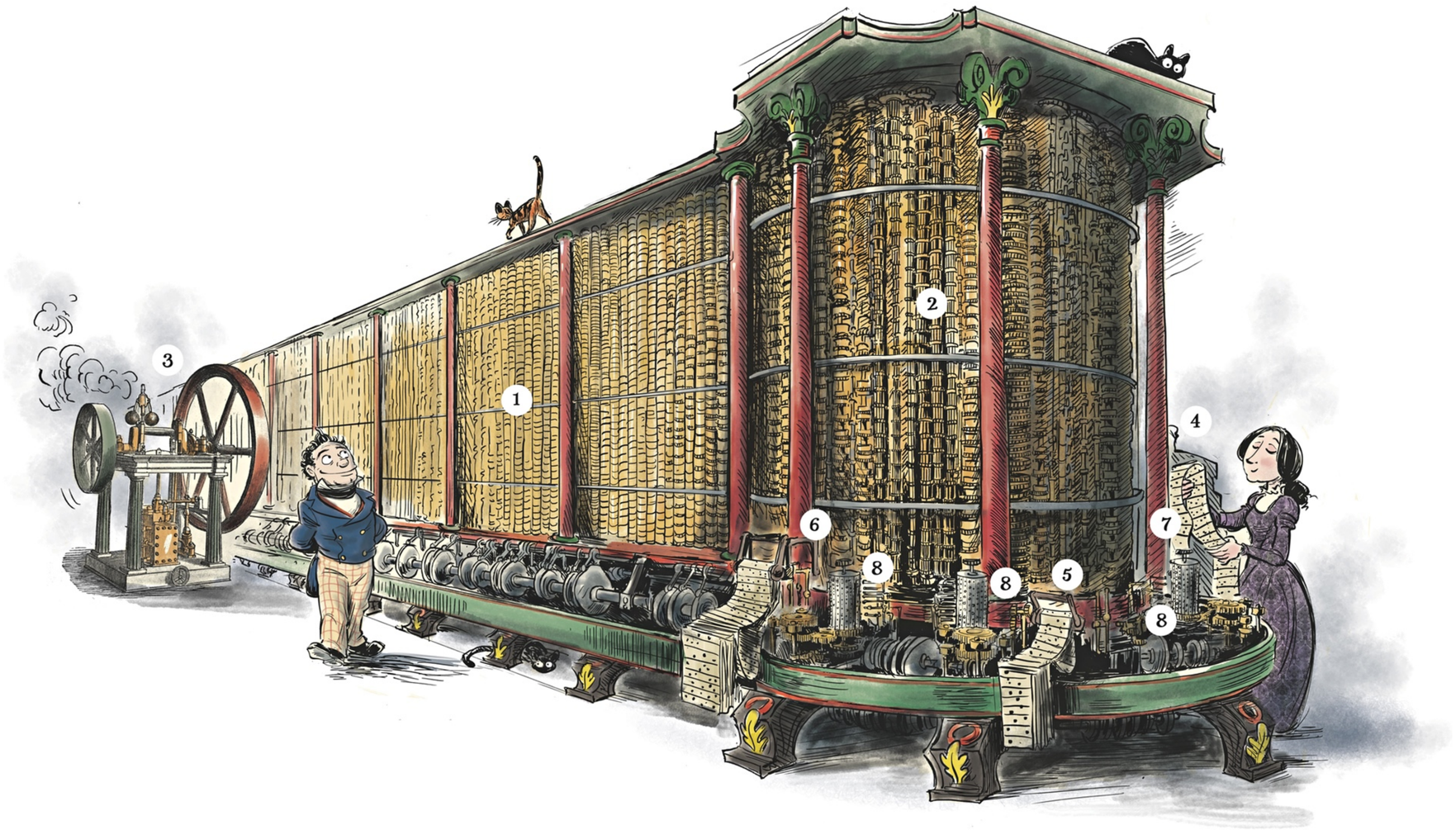


Charles Babbage's Difference Engine



Charles Babbage's Analytical Engine





What are "instructions" on a processor?

The Raspberry Pi has an *instruction set* that describe to a processor what it should do. Instruction sets are different for different processors, and the ARM instruction set happens to be called a "Reduced Instruction Set Computer" or *RISC*. This is compared to your laptop's processor (Intel or AMD x86-64) which has a "Complex Instruction Set Computer" processor.

RISC instructions are simple, and RISC processors are relatively straightforward in their design. Every instruction on a RISC processor takes 1 *clock cycle* (although it is not quite that simple — more on that later), and learning RISC assembly language is not too hard.

Recognize either of these guys?

Recognize either of these guys?

John Hennessy
— created the MIPS processor, a contemporary RISC to the ARM processor.
— Stanford Professor in CS and EE, and President of Stanford from 2000-2016.

David Patterson
— coined the term "RISC"
— Berkeley Professor in EECS.

Two weeks ago, Hennessy and Patterson were named the 2017 Turing Award recipients for their work in developing RISC processors. If it wasn't for these two, your phone would have a completely different processor!

Our First ARM Instruction: `mov`

The first instruction we are going to look at in the ARM instruction set is the **`mov`** instruction. We will take a quick look at a couple of instructions, and then we will investigate how the ARM processor handles instructions.

In textual form ("assembly language") one form of the **`mov`** instruction looks like this:

```
mov r0, #IMM
```

The first *operand*, `r0` is a *register* — registers are 32-bit (4 byte) values that are stored in a *register file* on the processor, and you can think of them as extremely fast memory. They *do not* have addresses, and they are *not* part of the main memory of the processor.

The second operand, "#IMM" stands for "immediate," which is a numeric value prefixed with a "#" character, e.g., "#8" means the number 8.

The instruction above says, "move the value IMM into register `r0`.

Our Second ARM Instruction: `add`

The second instruction we are going to look at in the ARM instruction set is the **`add`** instruction. The add instruction (as you might guess) adds values.

In assembly language the **`add`** instruction looks like this:

```
add r2, r0, r1
```

The instruction above says, "add the values of registers **`r0`** and **`r1`** and put the result into register **`r2`**.

Let's take a look at an emulator to see how the **`mov`** and **`add`** instructions work.

Instructions: simply binary

The program below is what we just looked at in the emulator:

```
mov r0, #8
mov r1, #7
add r2, r0, r1
```

The Raspberry Pi does not work with textual representations of the code — everything is a number, and the instructions above are no exception — by the time they end up on the processor, they have been translated into numbers. Here's how we can do that:

1. Create the assembly program, called "**`simple.s`**" which has the code above.
2. Run the ARM assembler on it, with the command to create the **`simple.o`** *object* file:
   `arm-none-eabi-as simple.s -o simple.o`
3. Run the ARM objcopy utility to convert it to a "binary" which is just numbers:
   `arm-none-eabi-objcopy simple.o -O binary simple.bin`

Instructions: simply binary

The program below is what we just looked at in the emulator:

```
mov r0, #8
mov r1, #7
add r2, r0, r1
```

When we are done converting the assembly code to binary, we can look at it, in various ways. We can *dump* the binary of the object (`.o`) file as follows:

```
$ arm-none-eabi-objdump -d simple.o

simple.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <.text>:
   0: e3a00008    mov   r0, #8
   4: e3a01007    mov   r1, #7
   8: e0802001    add   r2, r0, r1
```

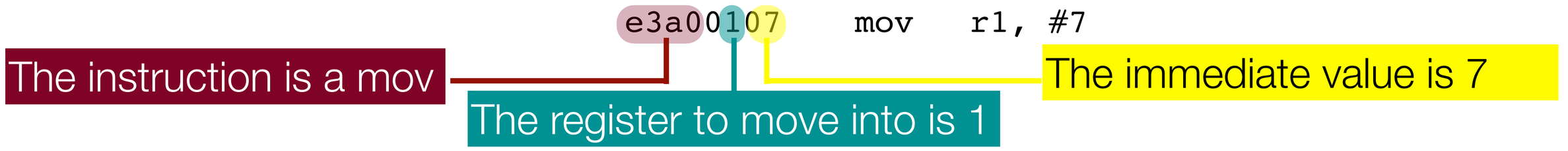
Instructions: simply binary

When we are done converting the assembly code to binary, we can look at it, in various ways. We can *dump* the binary of the object (`.o`) file as follows:

```
$ arm-none-eabi-objdump -d simple.o

simple.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <.text>:
   0: e3a00008    mov   r0, #8
   4: e3a01007    mov   r1, #7
   8: e0802001    add   r2, r0, r1
```

These are the numeric representations of the assembly code! Each instruction is 4 bytes (2 hex digits is one byte).

In fact — once you know what you are looking for, decoding some of the instructions is pretty straightforward (we will have an assignment on this!). For example:



Brief diversion: binary and hexadecimal

Because a byte is made up of 8 bits, we can represent the range of a byte as follows:

`00000000` to `11111111`

This range is 0 to 255 in decimal.

But, neither binary nor decimal is particularly convenient to write out bytes (binary is too long, and decimal isn't numerically friendly for byte representation)

So, we use "hexadecimal," (base 16).

Hexadecimal

Hexadecimal has 16 digits, so we augment our normal 0-9 digits with six more digits: A, B, C, D, E, and F.

The following table shows the hex digits and their binary and decimal values:

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

- In C, we write a hexadecimal with a starting `0x`. So, you will see numbers such as `0xfa1d37b`, which means that it is a hex number.
- You should memorize the binary representations for each hex digit. One trick is to memorize A (`1010`), C (`1100`), and F (`1111`), and the others are easy to figure out.
- Let's practice some hex to binary and binary to hex conversions:

Convert: `0x173A4C` to binary.

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

`0x173A4C` is binary
`0b000101110011101001001100`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b11110010101101101100011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b11110010101101101100011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b11110010101101101110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b11110010101101101110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b11110010101101110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b11110010101101110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b11110010101101110110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b11110010101101110110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b111100101011011011011011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b111100101011011011011011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b11110010101101101110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b11110010101101101110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

Convert: `0b11110010101101101110011` to hexadecimal.

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

(start from the **right**)

`0b11110010101101101110011`
is hexadecimal `3CADB3`

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

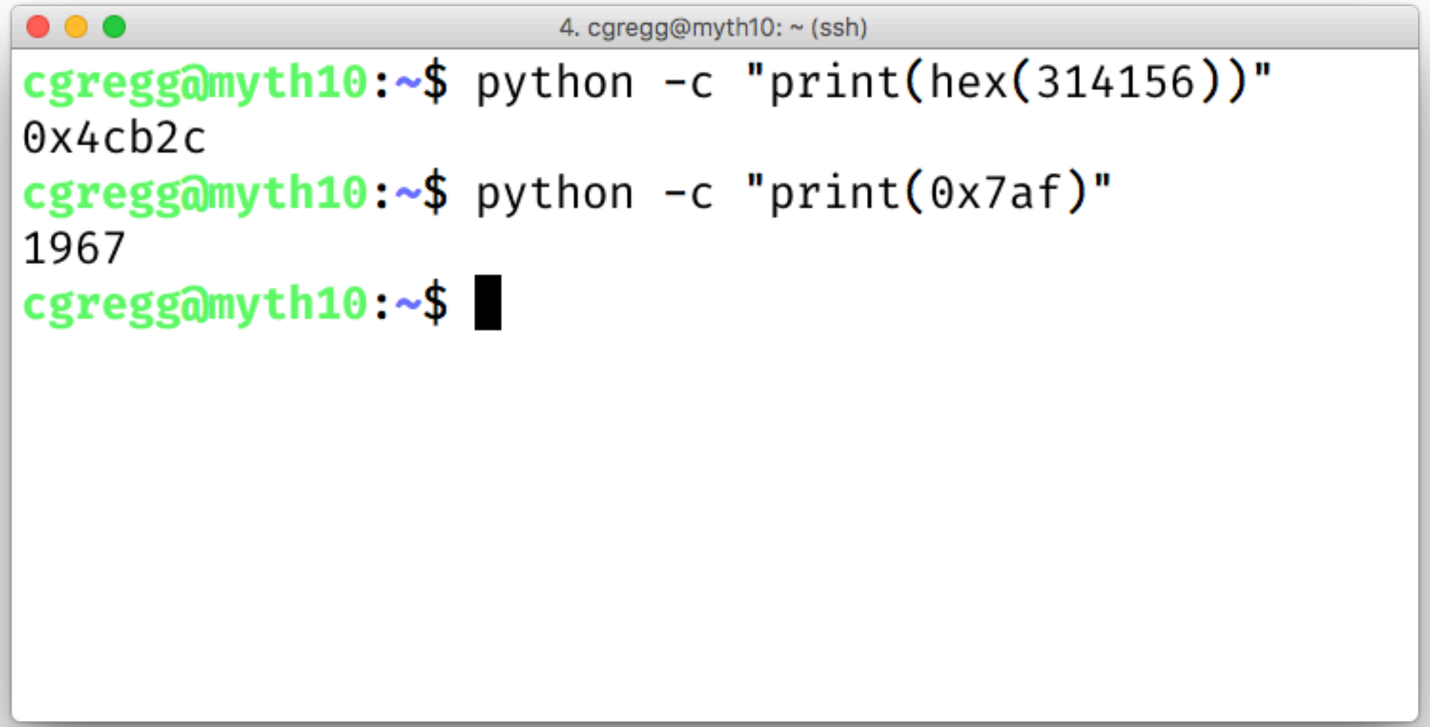
Hexidecimal to Decimal

To convert from hexadecimal to decimal, multiply each of the hexadecimal digits by the appropriate power of 16:

```
0x7AF:


7 * 16^2 + 10 * 16 + 15
= 7 * 256 + 160 + 15
= 1792 + 160 + 15 = 1967
```

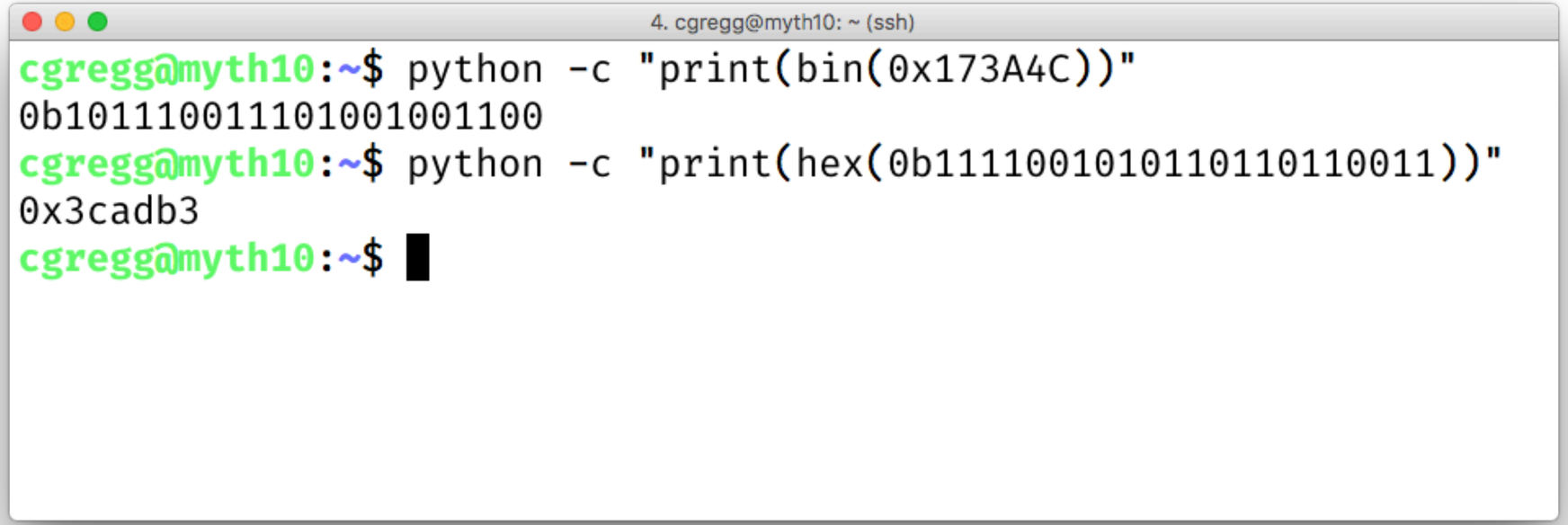
Let the computer do it!

Honestly, hex to decimal and vice versa are easy to let the computer handle. You can either use a search engine (Google does this automatically), or you can use a python one-liner:



Let the computer do it!

You can also use Python to convert to and from binary:



(but you should memorize this as it is easy and you will use it frequently)

Instructions: simply binary

We can also look directly at the binary output with the *hexdump* command:

```
$ hexdump simple.bin
0000000 08 00 a0 e3 07 10 a0 e3 06 f0 a0 e3 01 20 80 e0
```

This doesn't include the assembly instructions, and each byte seems to be reversed!

Instructions: simply binary

We can also look directly at the binary output with the *hexdump* command:

```
$ hexdump simple.bin
0000000 08 00 a0 e3 07 10 a0 e3 06 f0 a0 e3 01 20 80 e0
```

This doesn't include the assembly instructions, and each byte seems to be reversed!

From the objdump:

```
   0: e3a00008    mov   r0, #8
   4: e3a01007    mov   r1, #7
   8: e0802001    add   r2, r0, r1
```

This is reversed from above!

The reversal is because the raspi stores numbers in "little endian" format. In other words, for a four-byte number, the "little end" of the number is the byte that has the least significance to the number. For example, in the decimal number 1234, the 3 is less significant (30) than the 1 (1000). If we break a 4-byte hex number into bytes:

```
e3 a0 00 08
```

The `08` is the least significant byte, and the `e3` is the most significant byte.

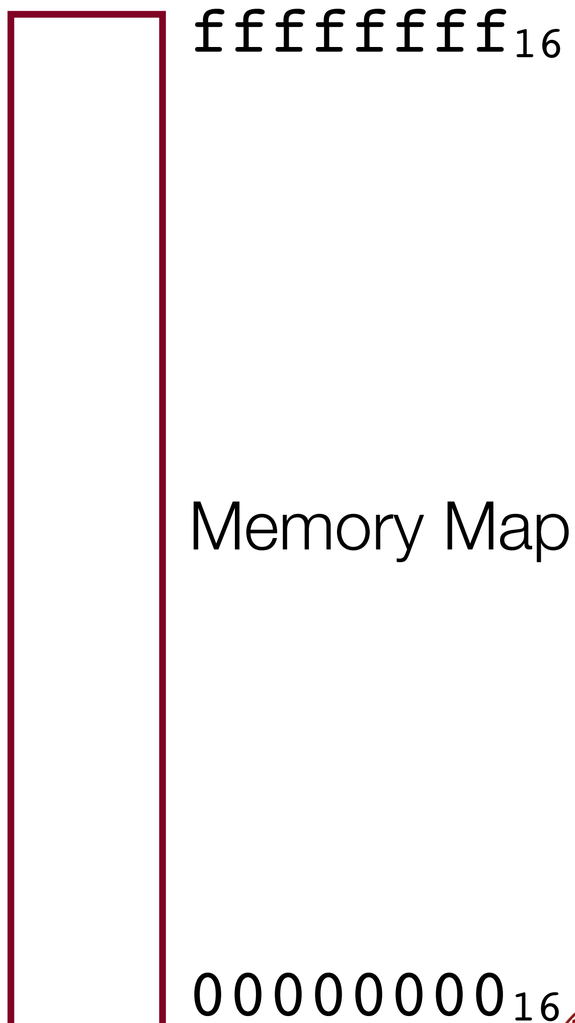
The ARM Memory Map

Before we talk about how instructions are handled, let's look briefly at the memory layout of the raspi.

Memory is *byte-addressable*, and every memory address must fit into a 32-bit value. Therefore, we can address 2^{32} bytes of memory, for a total of 4,294,967,296 bytes, or 4GB of memory.

The diagram to the right shows the memory layout with address `0` on the bottom, and address `0xffffffff` (the maximum) at the top.

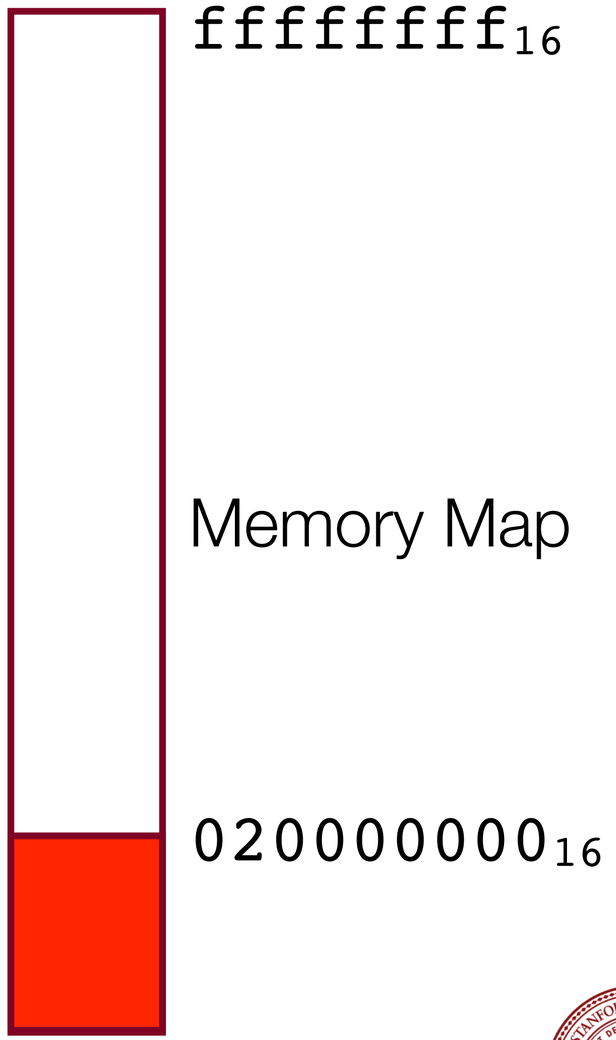
Instructions *and* data are placed into memory, somewhere in that range.



The ARM Memory Map

Your Raspberry Pis only have 512MB (2^{29} bytes, or `0x20000000` bytes) of memory, so the actual memory you have available is from `0` to `0x20000000`, as shown in red in the updated diagram.

We will actually use addresses above `0x20000000`, which refer to hardware addresses (like pins!).

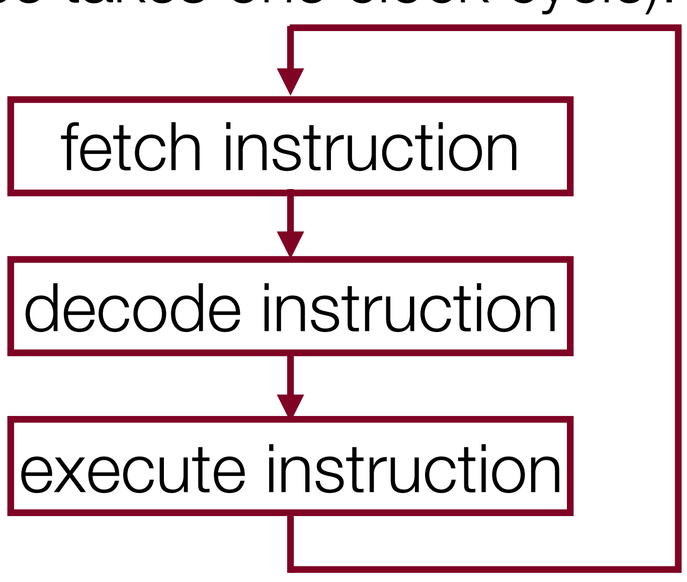


The instruction cycle

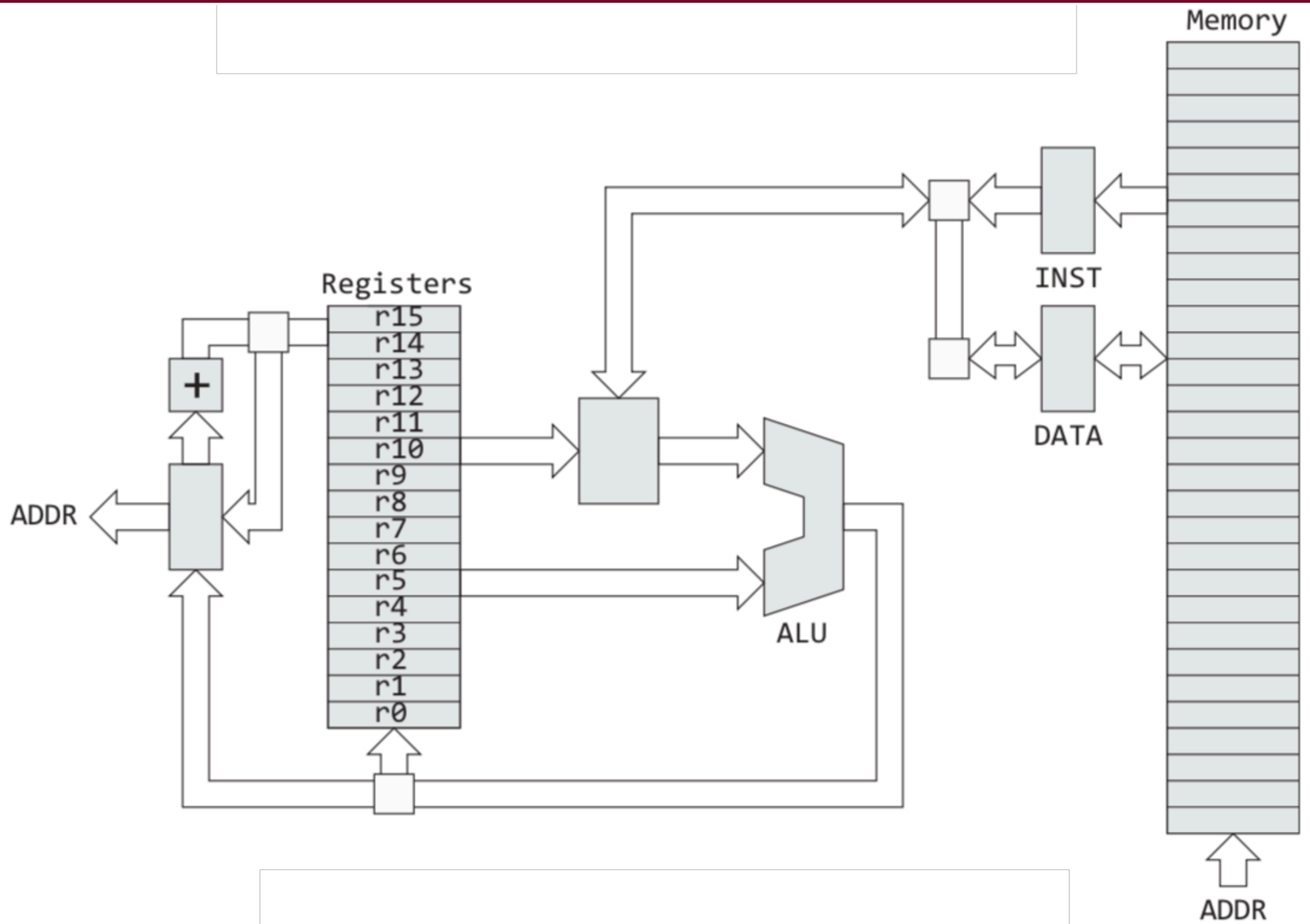
Now that we know a bit about instructions, we can talk about how the ARM processor in the raspi runs a program. A processor runs using a *clock signal* that is a simple on/off signal:

when the signal is high, that is *on* and when the signal is low, that is *off*.

Instructions in a program are handled one after the other (except for *branches*, which we will talk about later). On each clock cycle, the following three things happen (for a single instruction, each of the three takes one clock cycle):

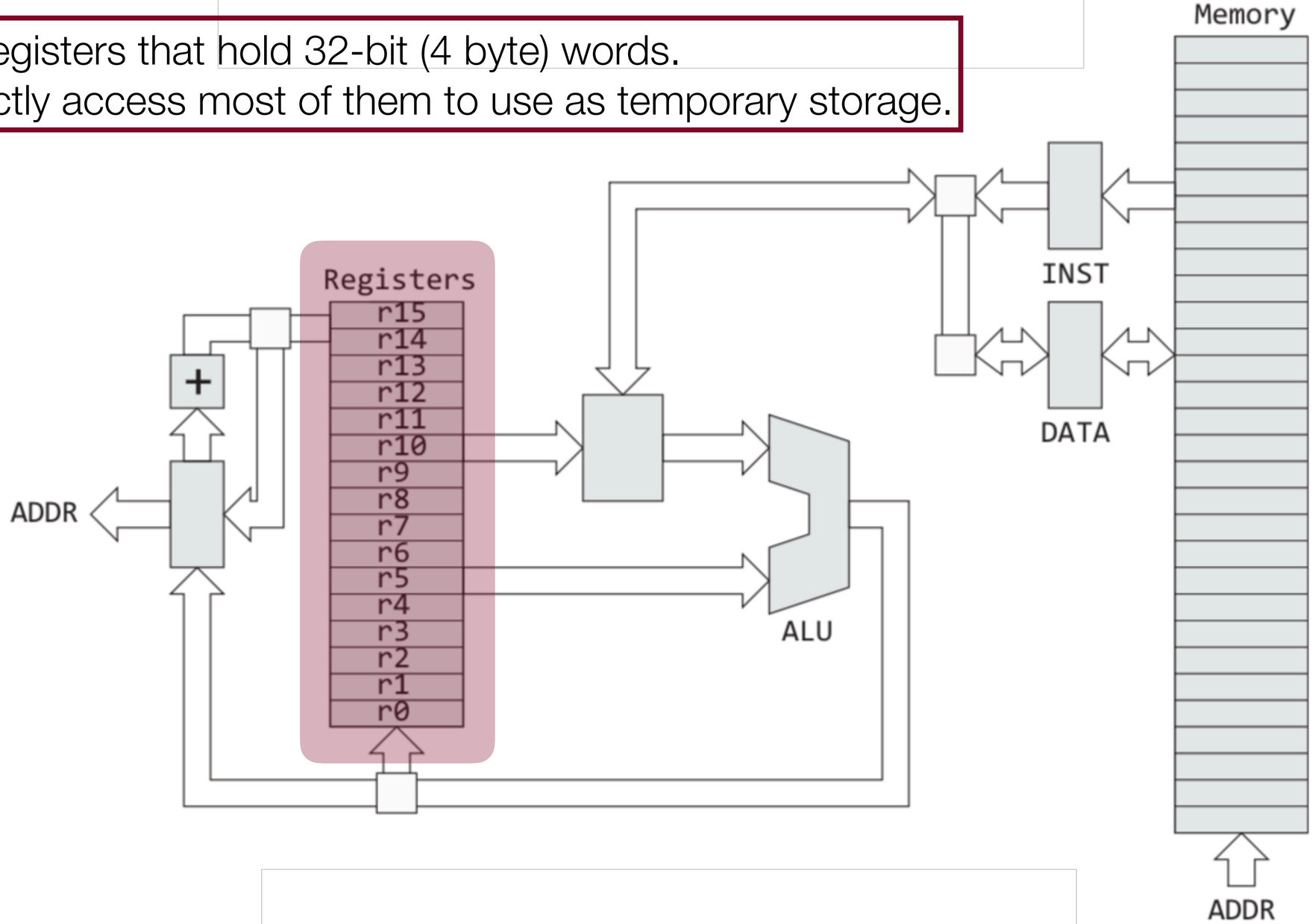


ARM Architecture / Floor Plan



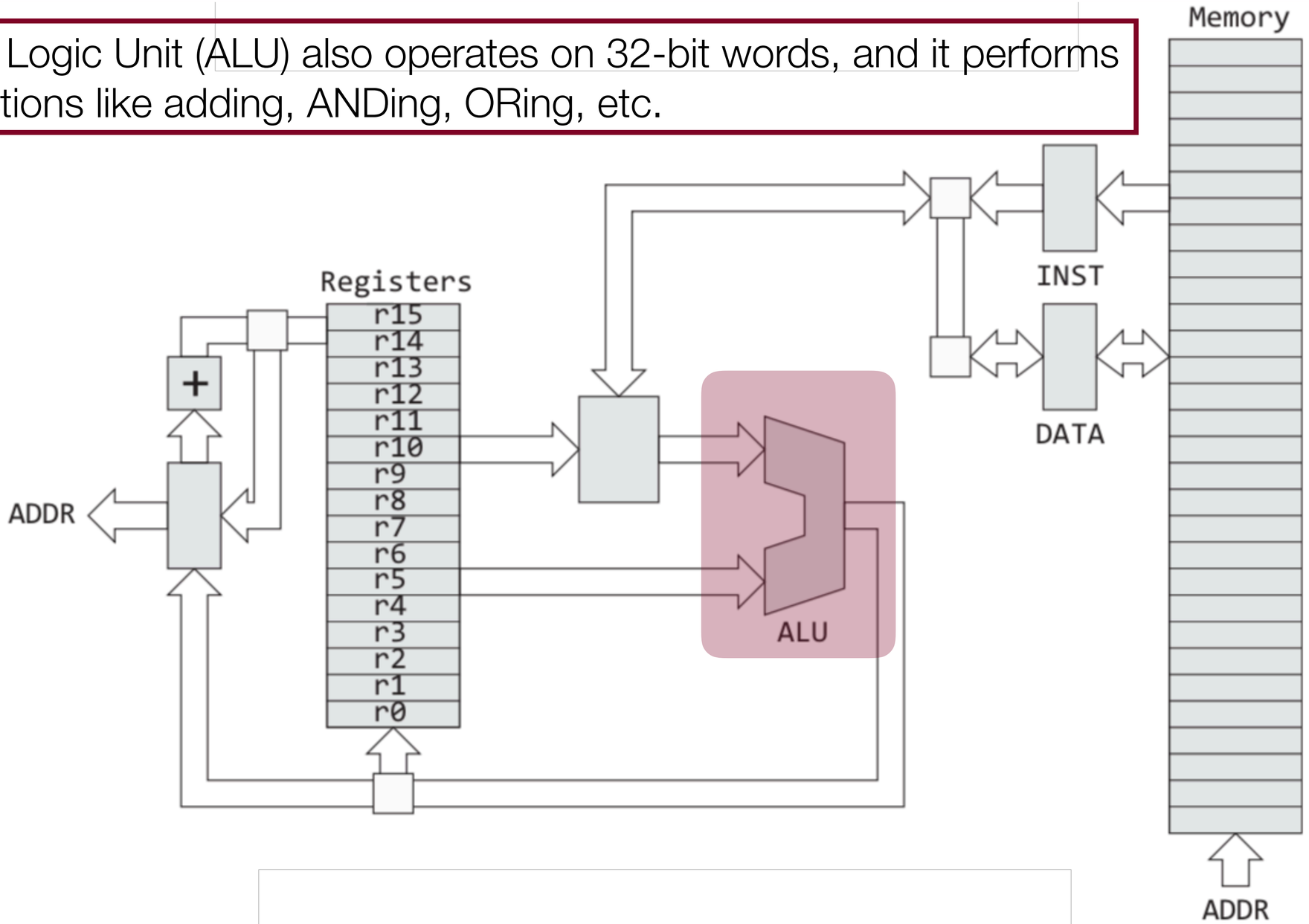
ARM Architecture / Floor Plan

There are 16 registers that hold 32-bit (4 byte) words.
Programs directly access most of them to use as temporary storage.



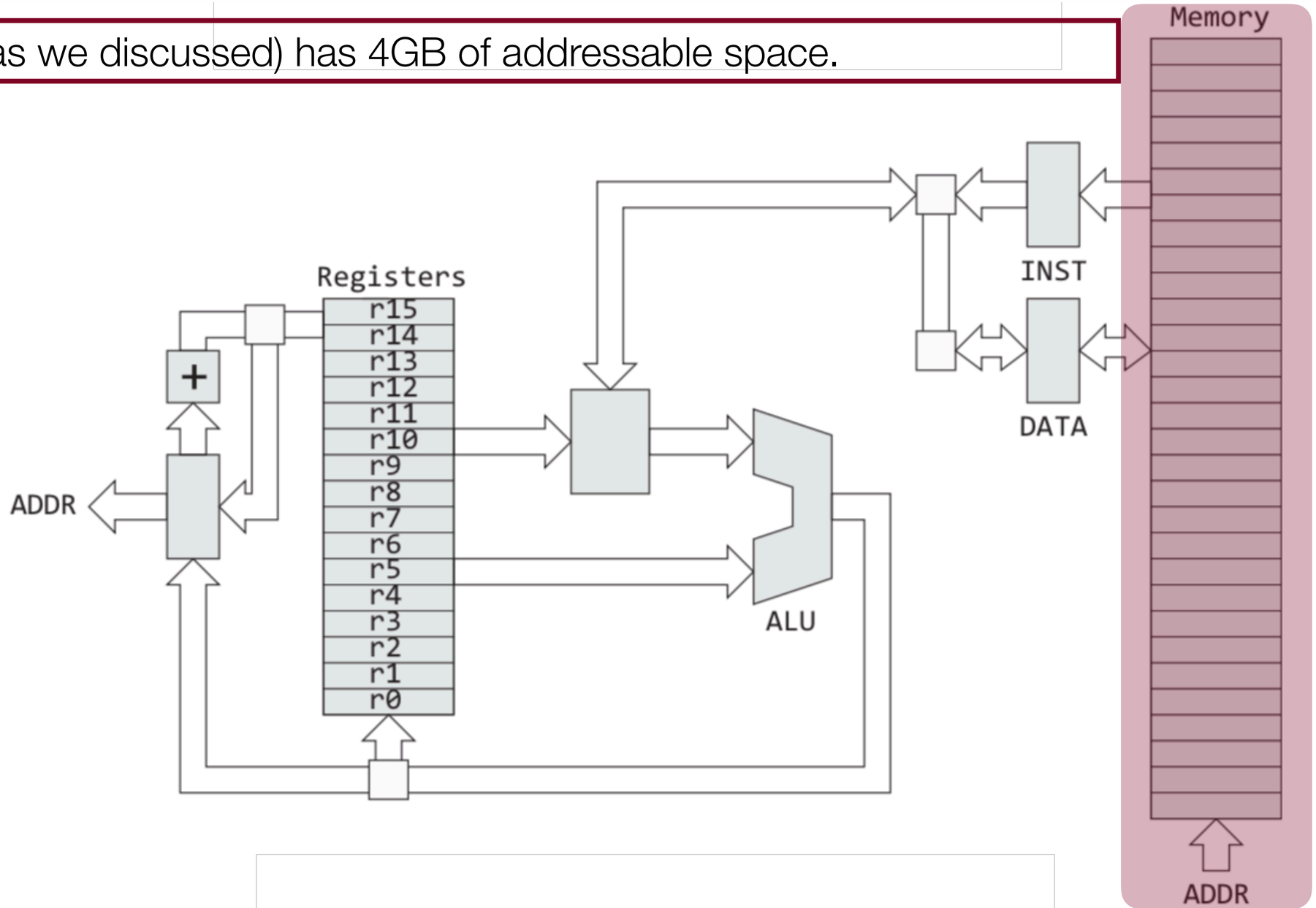
ARM Architecture / Floor Plan

The Arithmetic Logic Unit (ALU) also operates on 32-bit words, and it performs arithmetic functions like adding, ANDing, ORing, etc.



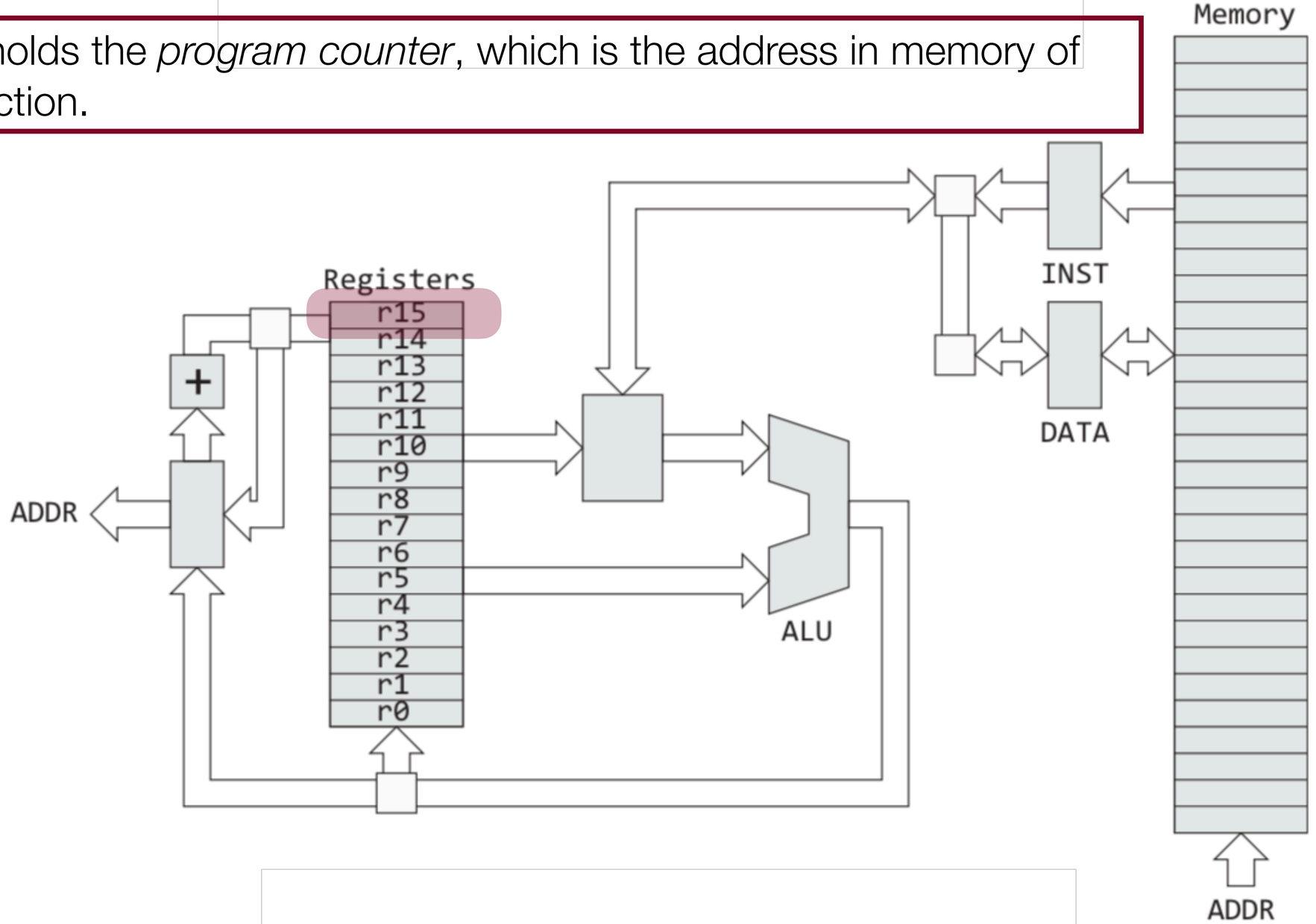
ARM Architecture / Floor Plan

The memory (as we discussed) has 4GB of addressable space.



Instruction Fetch

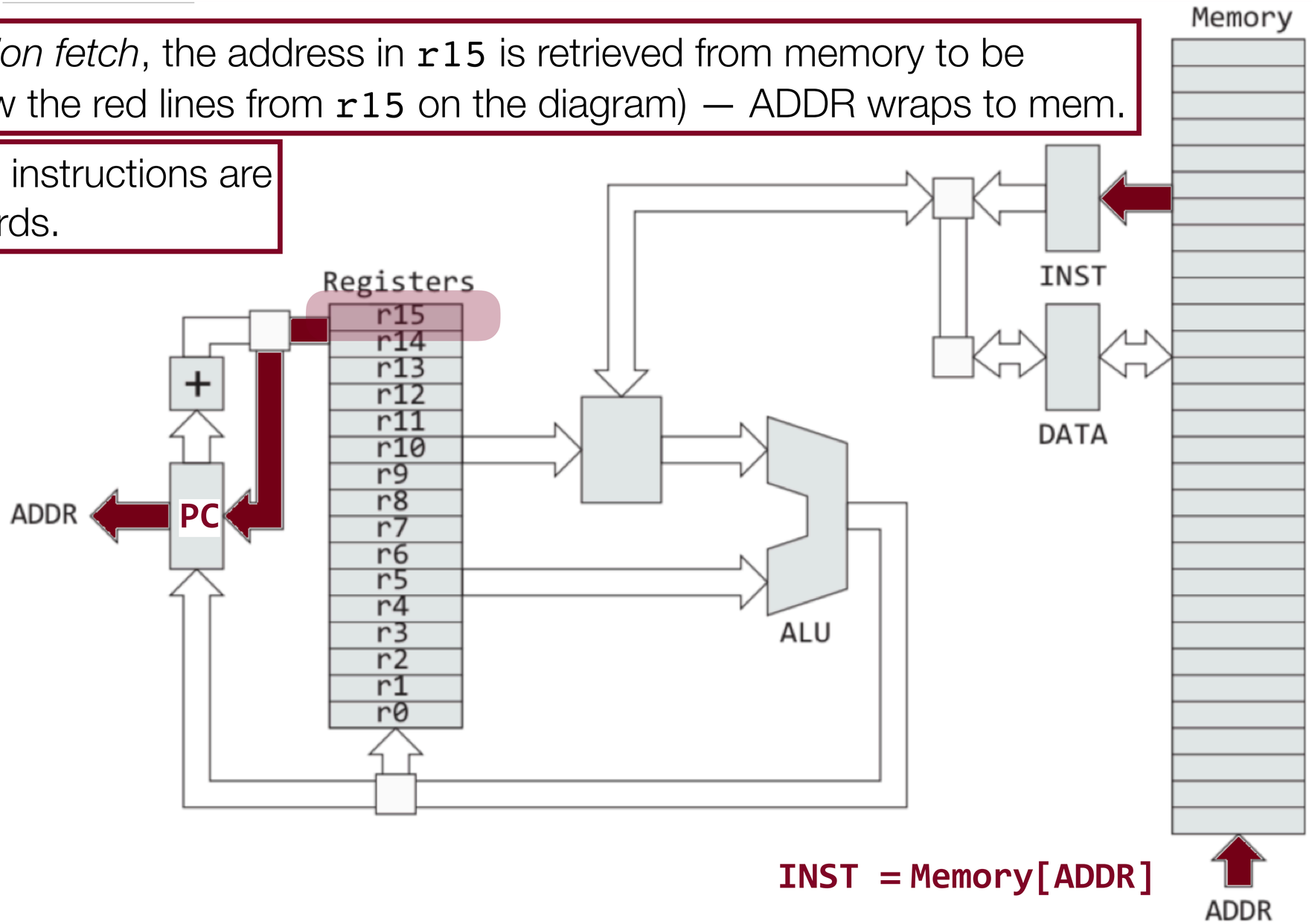
Register `r15` holds the *program counter*, which is the address in memory of the next instruction.



Instruction Fetch

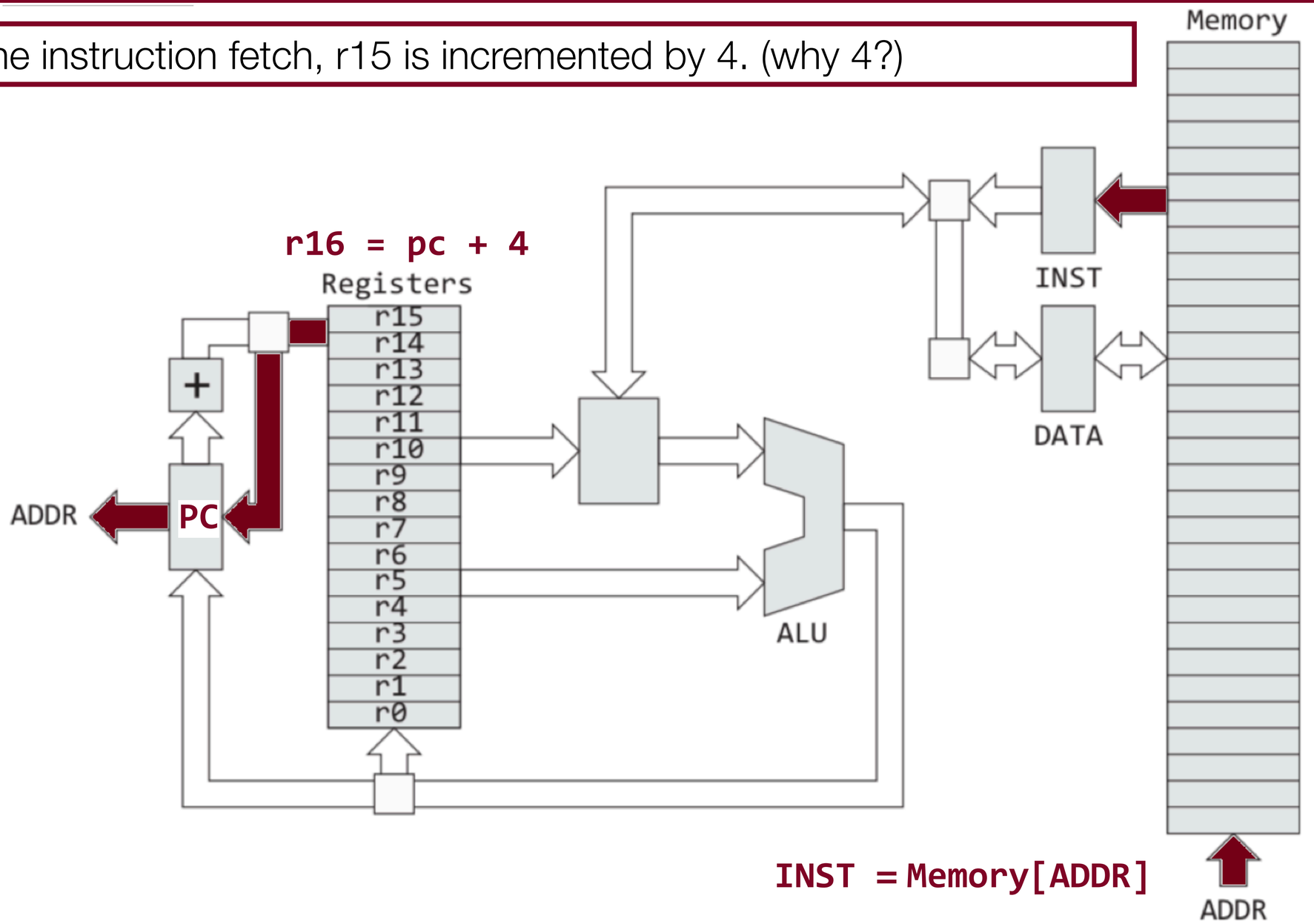
During *instruction fetch*, the address in `r15` is retrieved from memory to be decoded (follow the red lines from `r15` on the diagram) — ADDR wraps to mem.

Addresses and instructions are both 32-bit words.



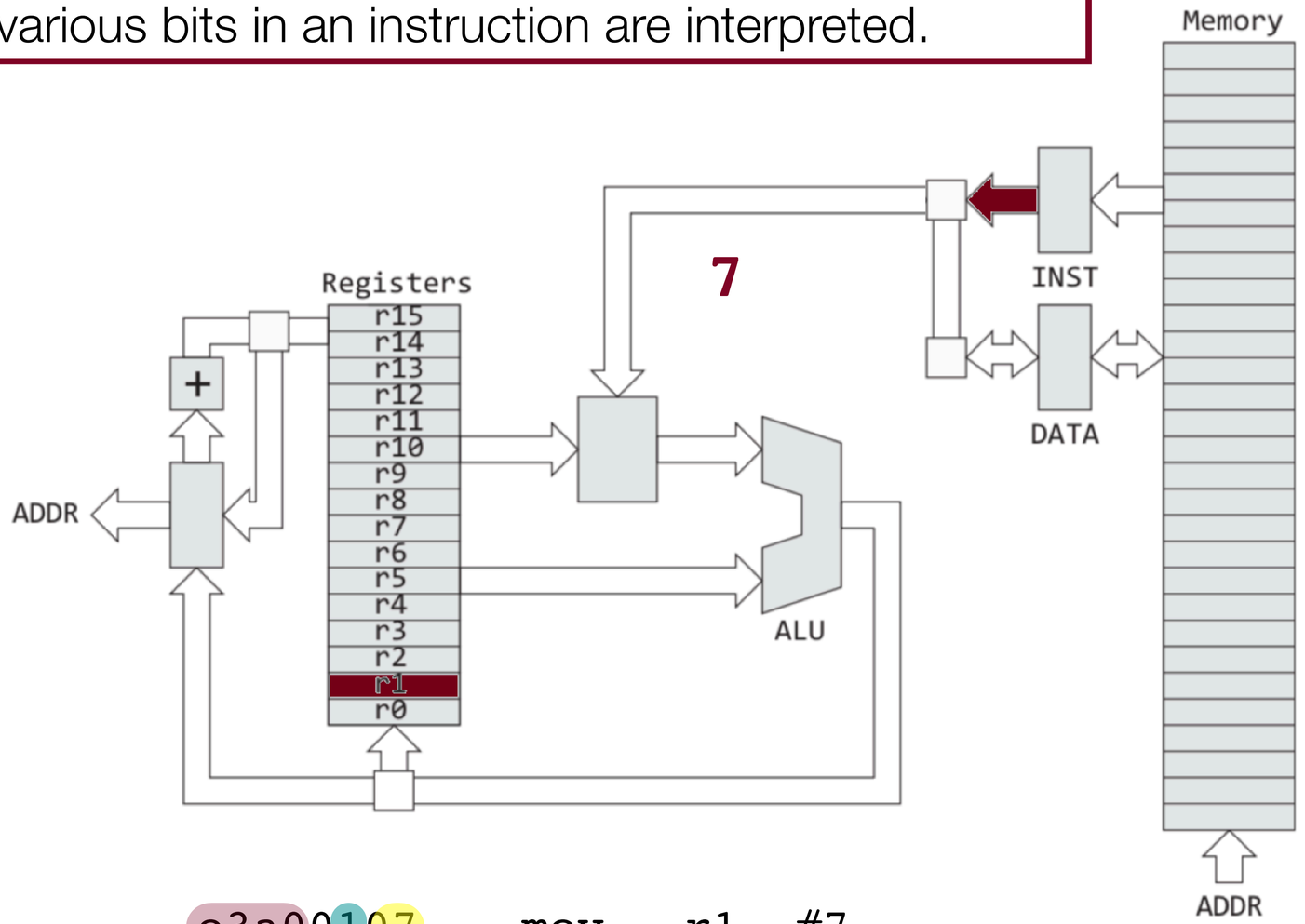
Instruction Fetch

At the end of the instruction fetch, r15 is incremented by 4. (why 4?)



Instruction Decode

During the "decode" step, the various bits in an instruction are interpreted.




```
e3a00107      mov    r1, #7
```

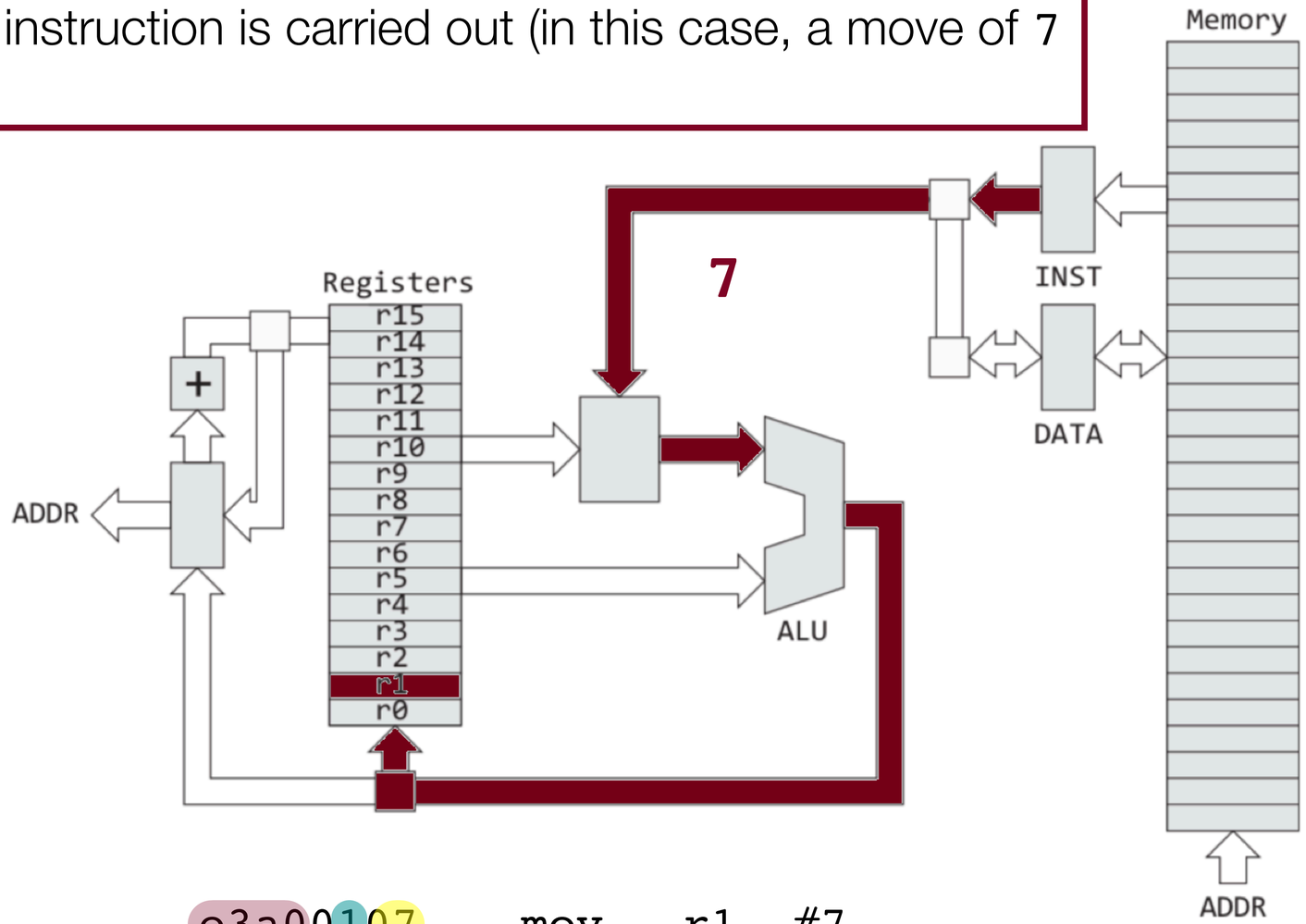
The instruction is a mov

The register to move into is 1

The immediate value is 7

Instruction Execute

During the "execute" step, the instruction is carried out (in this case, a move of `7` into register `r1`.




```
e3a00107      mov    r1, #7
```

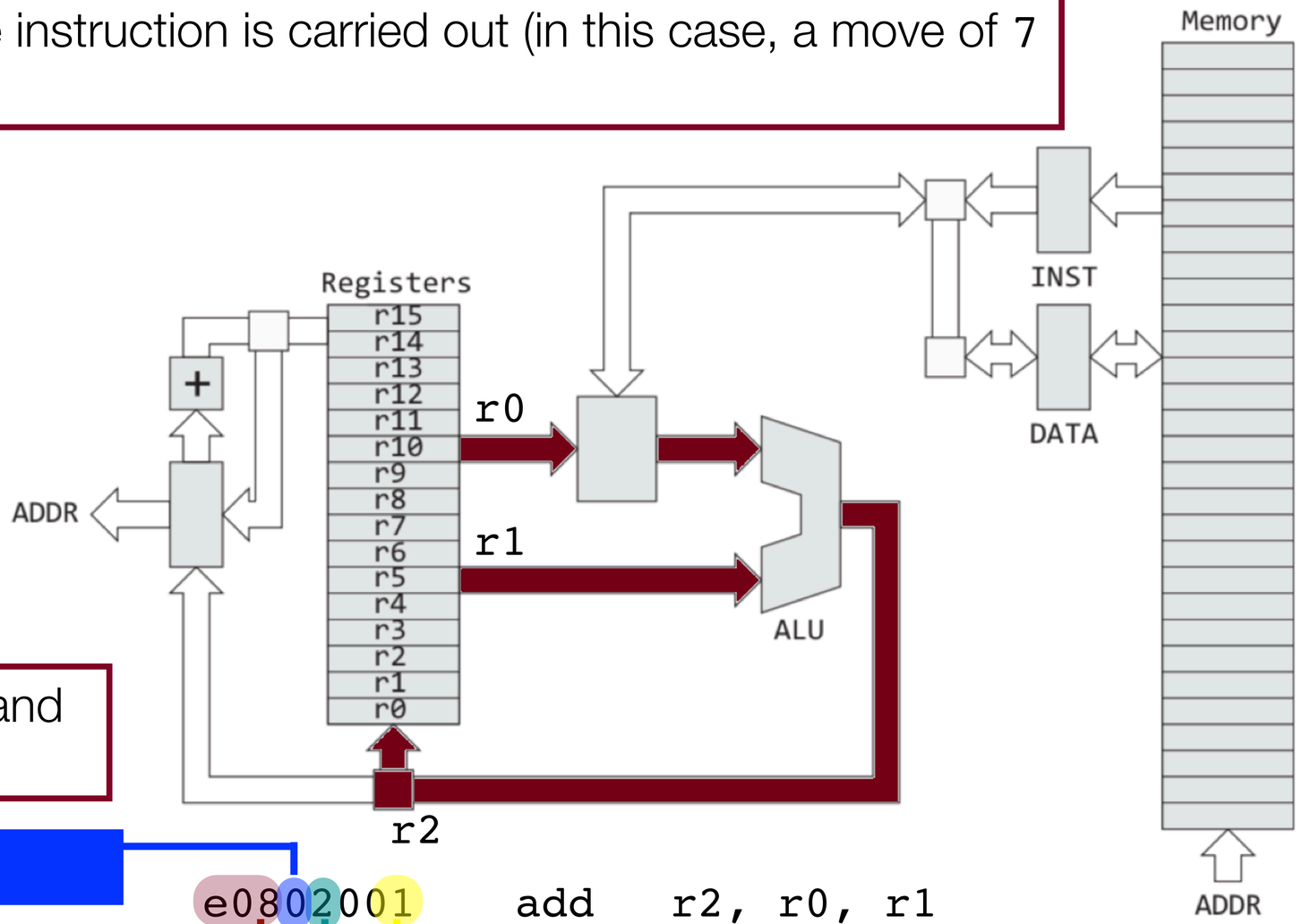
The instruction is a mov

The register to move into is 1

The immediate value is 7

Add Execute Example

During the "execute" step, the instruction is carried out (in this case, a move of `7` into register `r1`.



You do not need to understand this decoding yet!

The first operand is r0

`e0802001` `add r2, r0, r1`

The instruction is an add

The second operand is r1

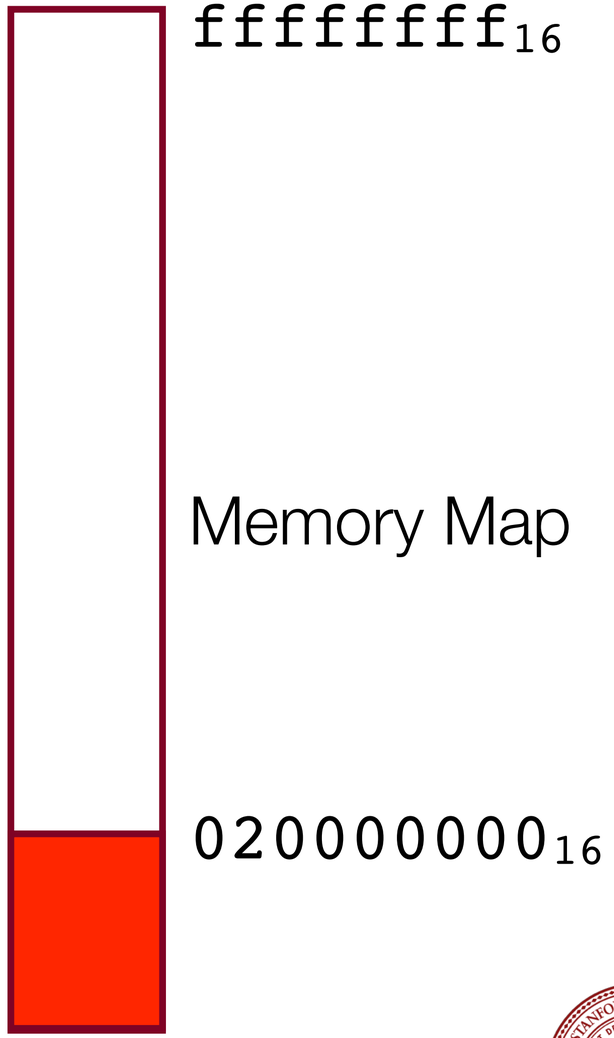
The destination register is r2

Load and Store Instructions

ARM processors only have 16 registers, which each hold 32 bits (4 bytes) each. A program can realistically only use 13 of them to store data, for a total of 13 * 4 = 52 bytes of storage -- not much! This is *not* the same as main memory -- registers are actually on the processor chip, and main memory ("memory" from here on out) is on a separate chip.

As discussed, programs can store data in the (slower) main memory -- your raspi has 512MB of memory available, most of which is used to store instructions and data (the **red** area in the memory map to the right).

All calculations on ARM machines must be done using registers, so if you want to add two numbers that are in memory, you first have to *load* them from memory. We do that using the `ldr` instruction.



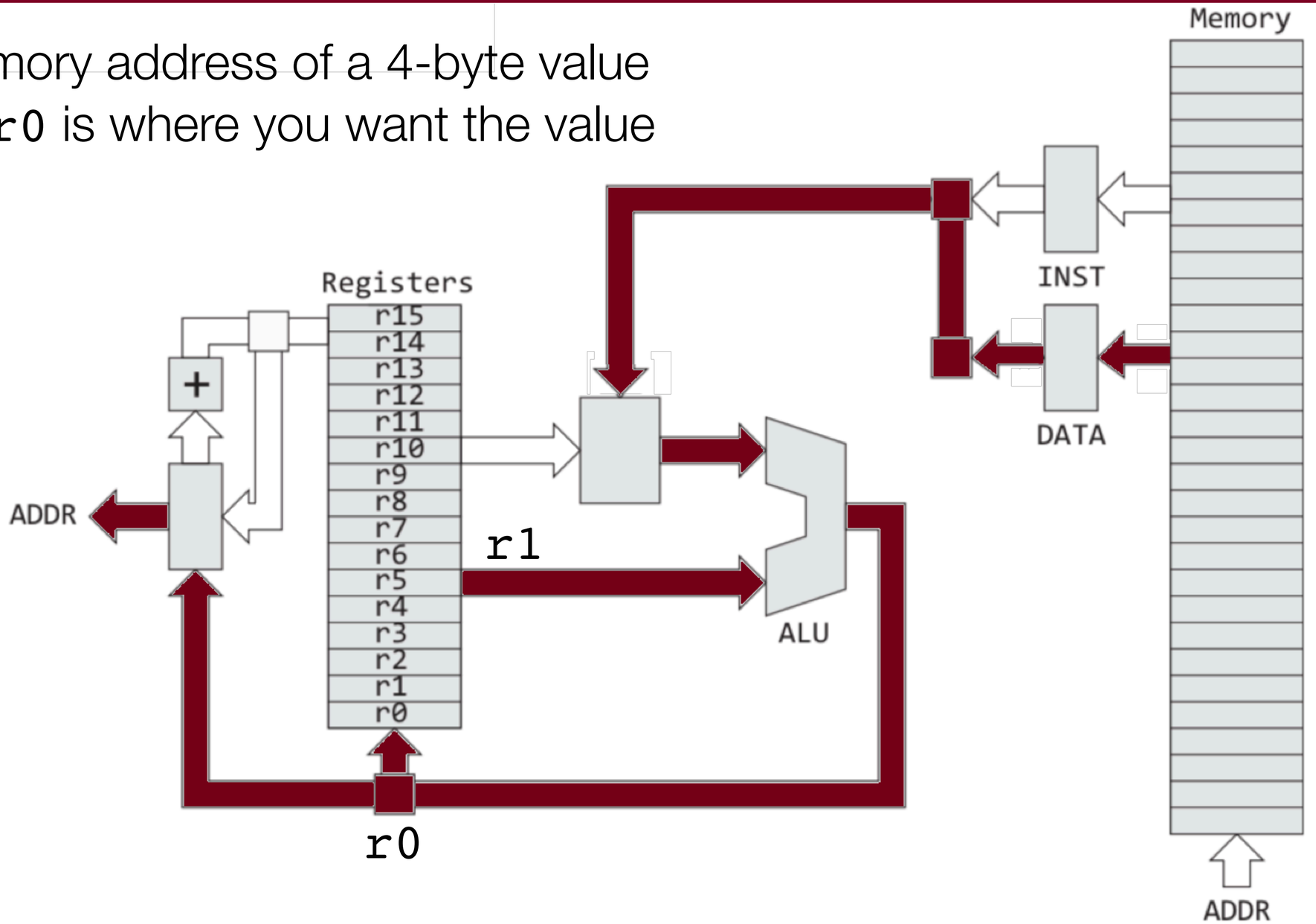
The Load Instruction

Assume `r1` holds the memory address of a 4-byte value you want to retrieve, and `r0` is where you want the value to end up.

To load the value from memory:

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

This means "load from main memory at address `r1` and put the value into `r0`"



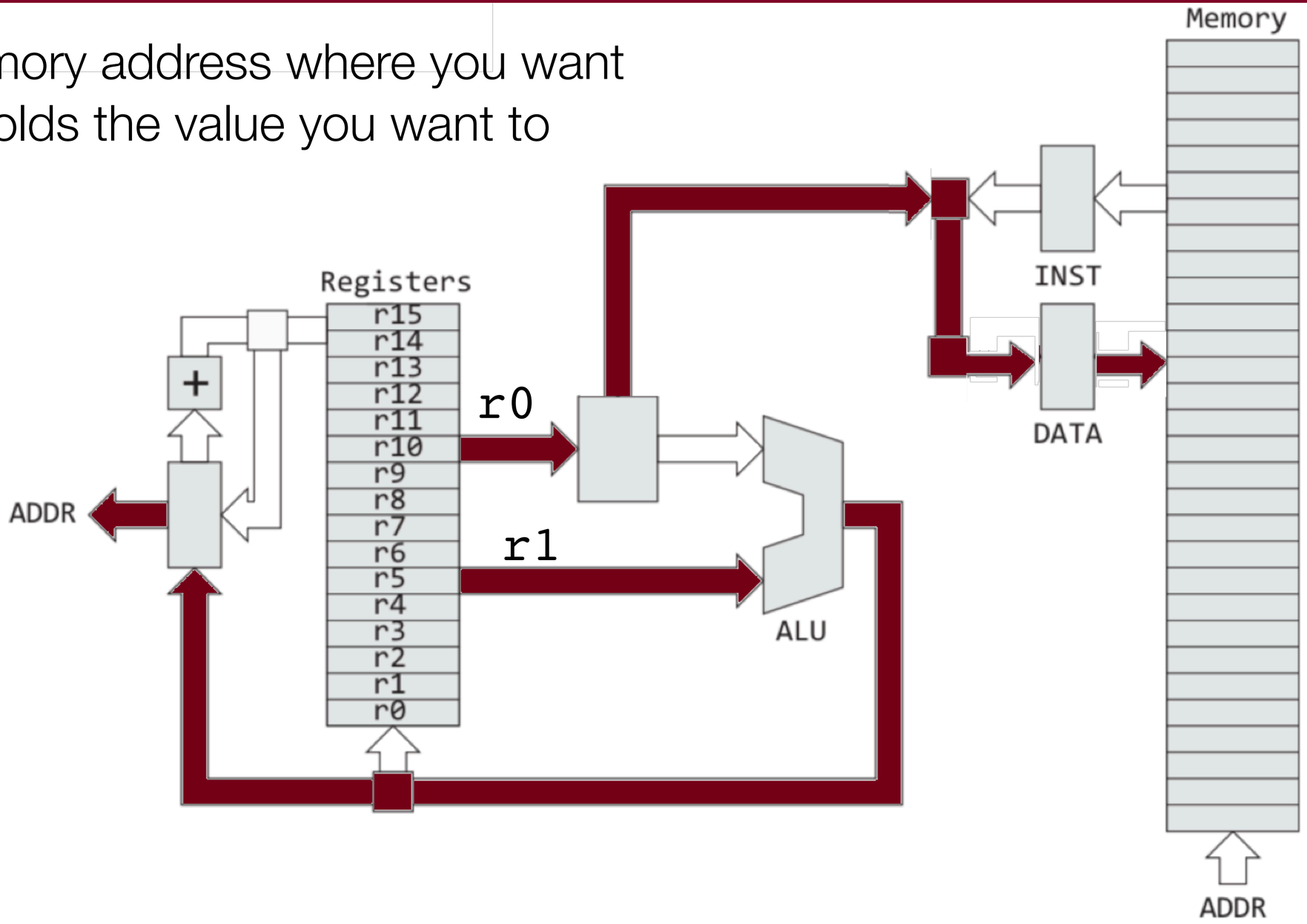
The Store Instruction

Assume `r1` holds the memory address where you want to store a value, and `r0` holds the value you want to store.

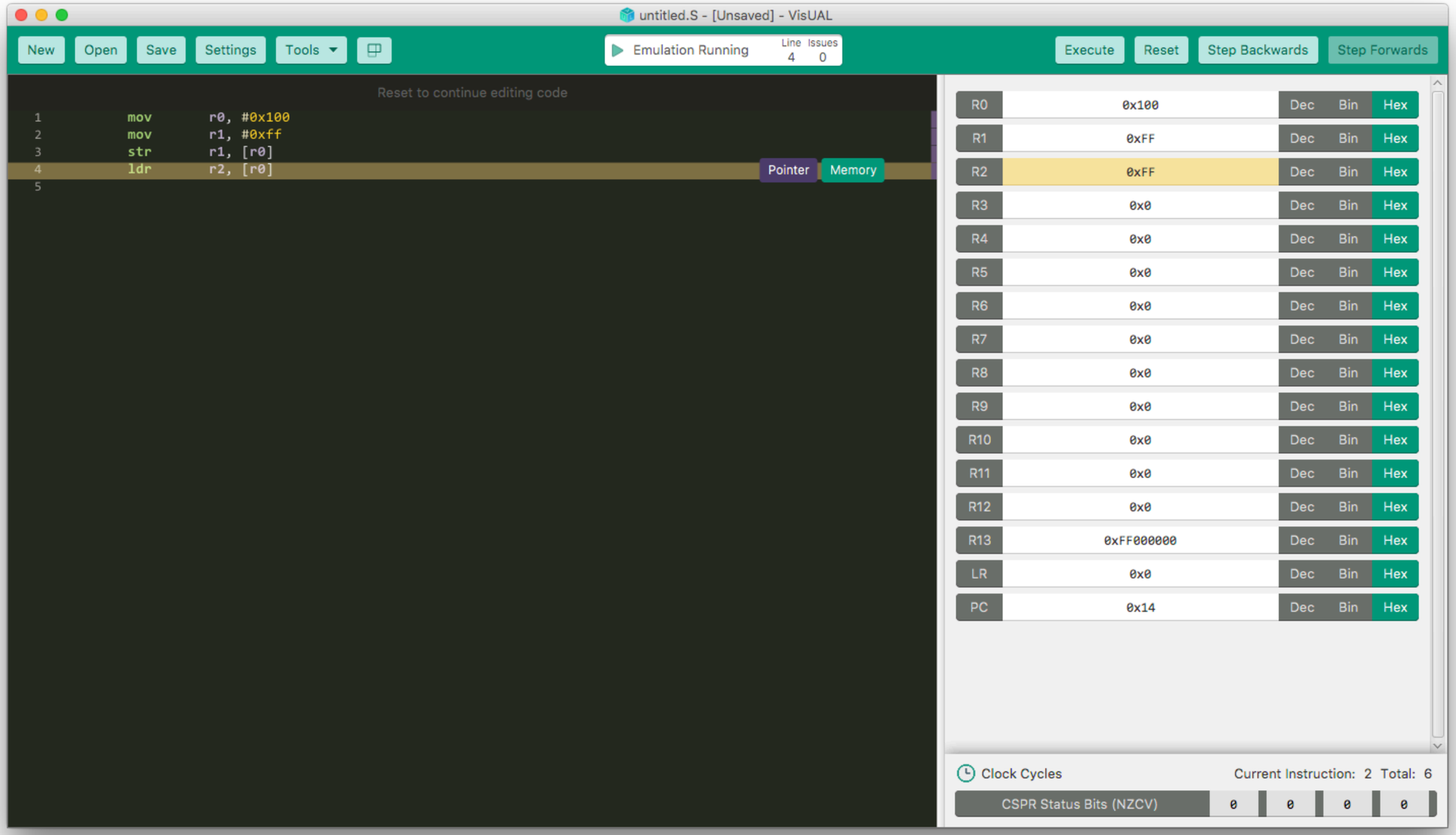
To store the value to memory:

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

This means "store the value in `r0` into memory address `r1`"

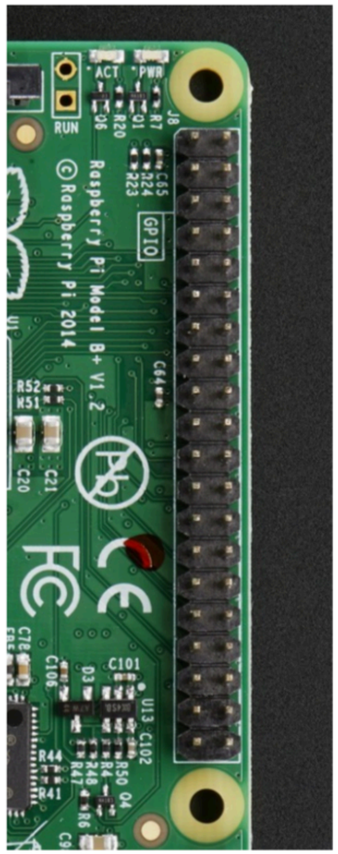


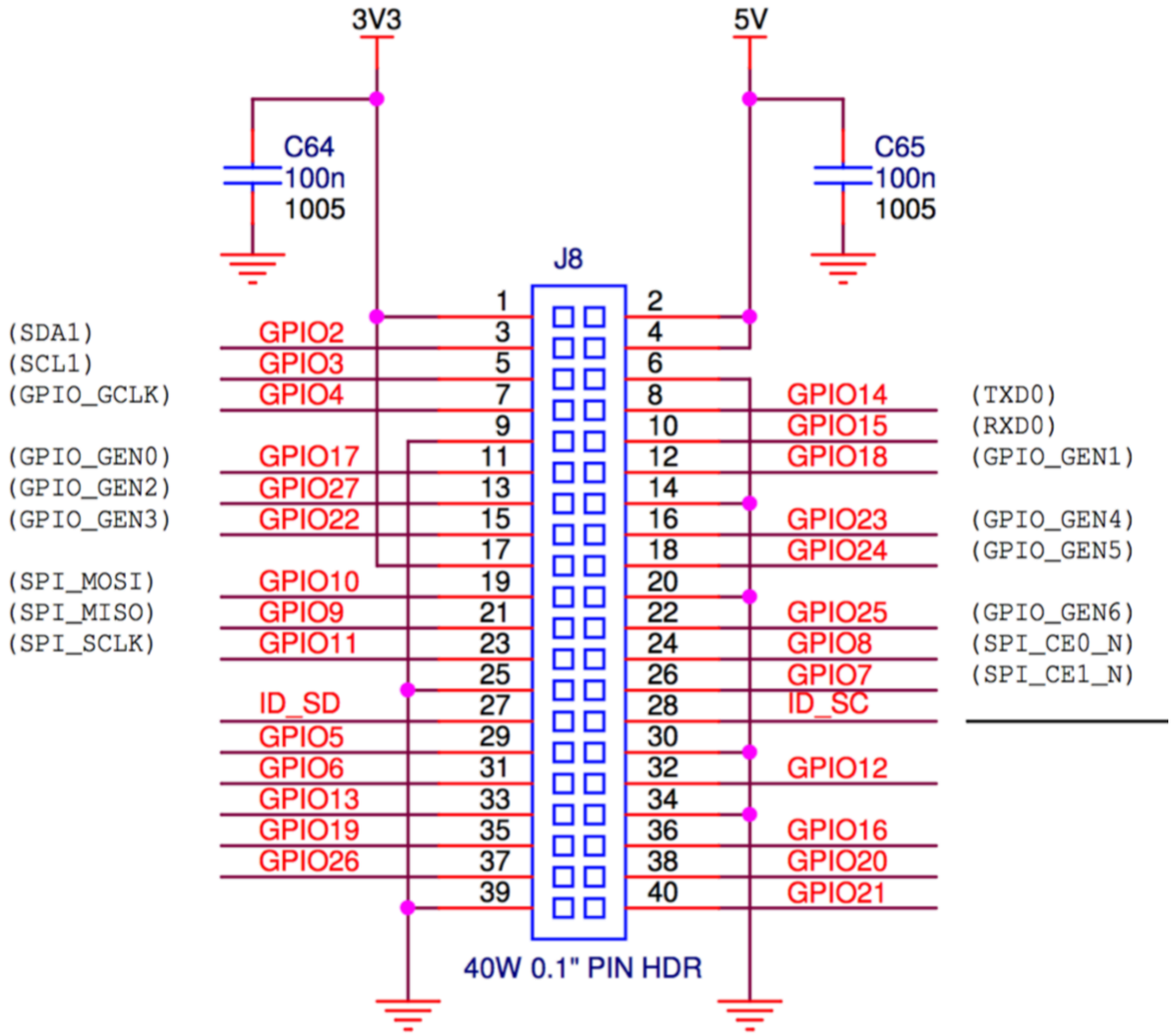
Load and Store Instructions



Turning on an LED

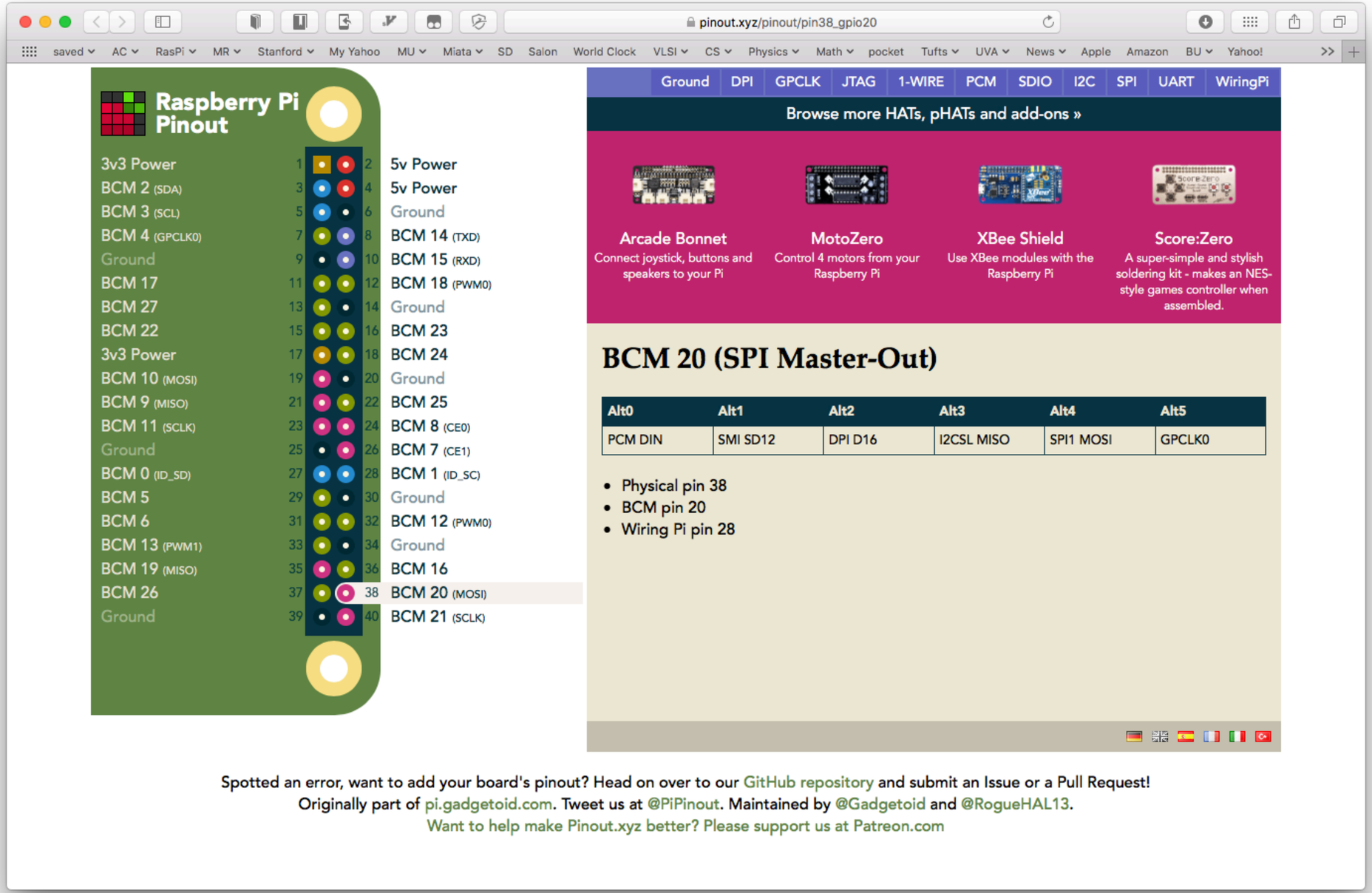
General-Purpose Input/Output (GPIO) Pins



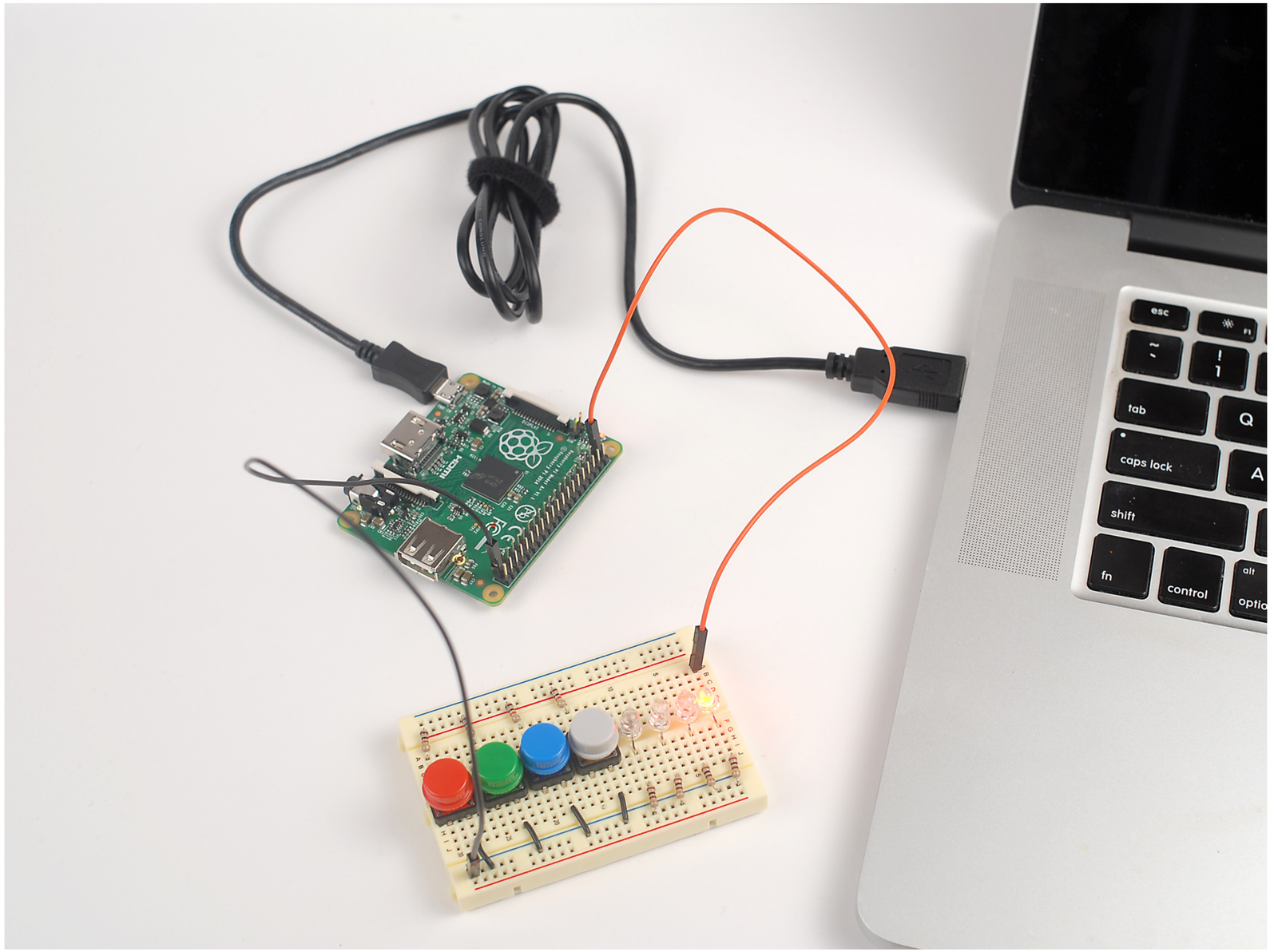


54 GPIO Pins

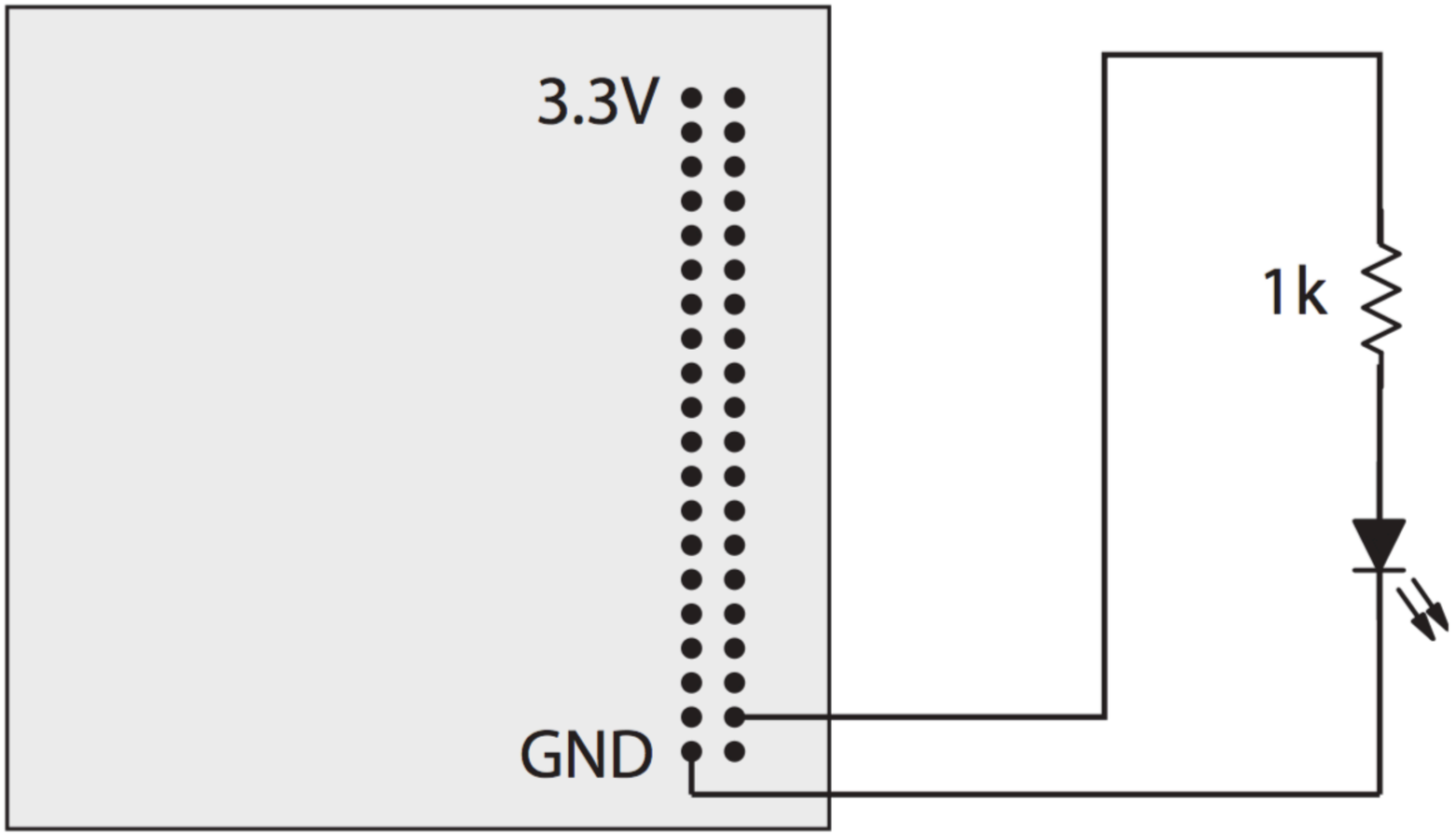
General-Purpose Input/Output (GPIO) Pins



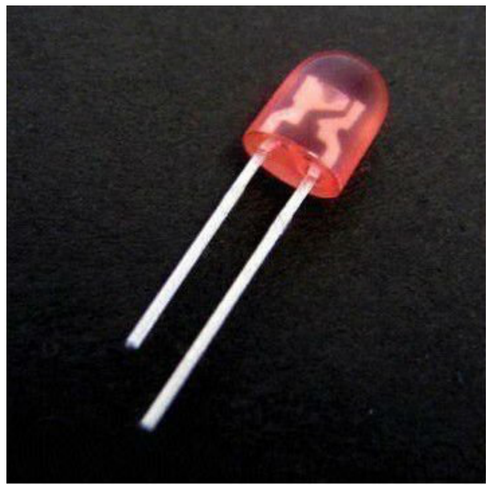
Connecting the raspi!



Connect LED to GPIO 20



Ground
(short leg)



Positive
voltage
(long leg)

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

GPIO Pins

GPIO Pins are called *peripherals*.

Peripherals are Controlled by special registers called *Peripheral Registers*

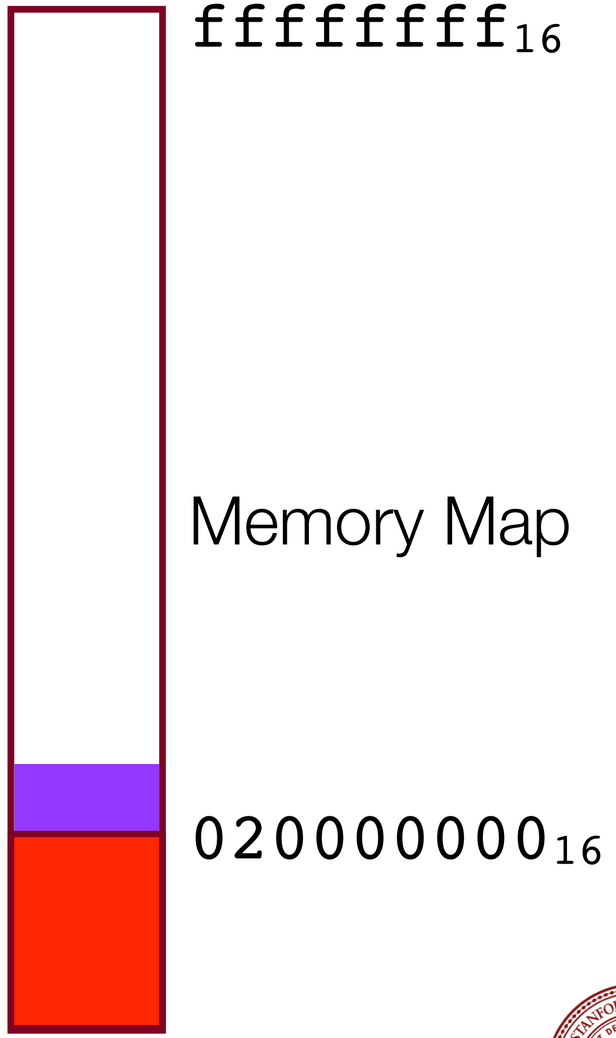
Back to the Memory Map

Peripheral registers are actually *mapped* into the address space.

This is called *Memory-Mapped IO (MMIO)*

MMIO space is *above* physical memory (purple in the diagram)

Ref: BCM2835-ARM-Peripherals.pdf



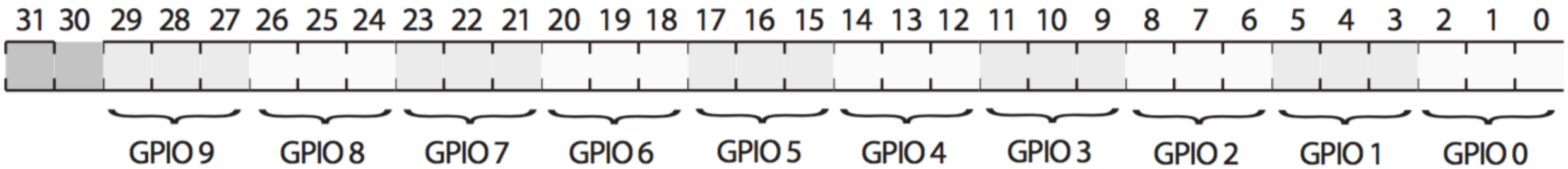
General-Purpose IO Function

GPIO Pins can be configured to be INPUT, OUTPUT, or ALT0-5
(see https://bit.ly/2Eq5FaF for information about ALT0-5)

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 are required to select the function

GPIO Function Select Register



Function is INPUT, OUTPUT, or ALT0-5

8 functions requires 3 bits to specify

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

54 GPIOs require 6 registers

GPIO Function Select Register

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!

Manual says: `0x7E200000`

For our raspi, replace `7E` with `20`: `0x20200000`

Ref: BCM2835-ARM-Peripherals.pdf

To Configure GPIO 20 for OUTPUT

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

Why 0x20200008?

To Configure GPIO 20 for OUTPUT

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 **GPIO Pins 0-9**	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1 **GPIO Pins 10-19**	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2 **GPIO Pins 20-29**	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3 **GPIO Pins 30-39**	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4 **GPIO Pins 40-49**	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5 **GPIO Pins 50-53**	32	R/W
0x 7E20 0018	-	Reserved	-	-

Watch out!

Manual says: `0x7E200000`

For our raspi, replace `7E` with `20`: `0x20200000`

Ref: BCM2835-ARM-Peripherals.pdf

To Configure GPIO 20 for OUTPUT

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 **GPIO Pins 0-9**	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1 **GPIO Pins 10-19**	32	R/W
0x ~~7E~~20 0008 0x20200008	GPFSEL2	GPIO Function Select 2 **GPIO Pins 20-29**	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3 **GPIO Pins 30-39**	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4 **GPIO Pins 40-49**	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5 **GPIO Pins 50-53**	32	R/W
0x 7E20 0018	-	Reserved	-	-

Watch out!

Manual says: `0x7E200000`

For our raspi, replace `7E` with `20`: `0x20200000`

Ref: BCM2835-ARM-Peripherals.pdf

To Configure GPIO 20 for OUTPUT

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

Why all the `orr`-ing?

The `mov` instruction can only take an immediate value of up to 8 bits that can be shifted by a power of two, and a few different forms (see https://stackoverflow.com/a/26762878/561677) (what? we'll cover this next week!)

So: 0x20000000, 0x00200000, and 0x00000008 can all be created by shifting an 8-bit value by a power of two (e.g., `1 << 29 == 0x20000000`)

To Configure GPIO 20 for OUTPUT

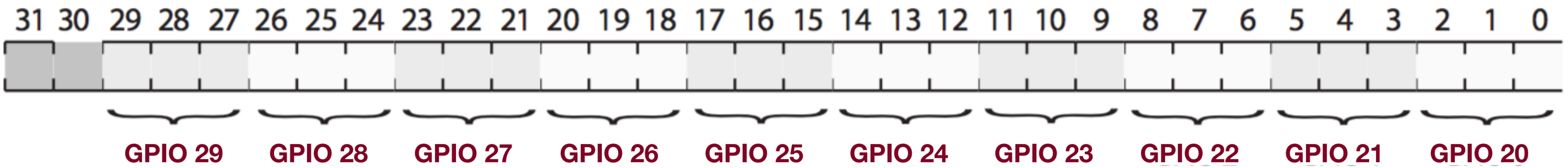
```
// FSEL = 0x20200008
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x00000008
mov r1, #1     // 1 indicates OUTPUT
str r1, [r0]   // store 1 to 0x20200008
```

Why are we moving `1` into `0x20200008`?

To Configure GPIO 20 for OUTPUT

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

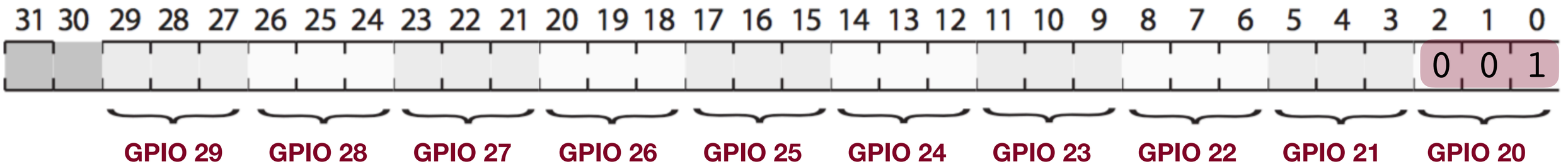
Why are we moving `1` into `0x20200008`?



To Configure GPIO 20 for OUTPUT

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

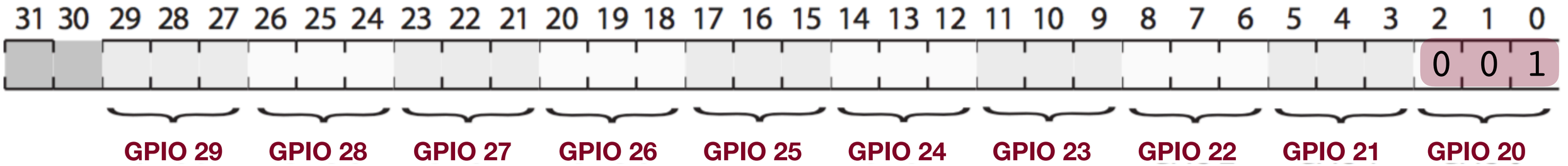
Why are we moving `1` into `0x20200008`?



To Configure GPIO 20 for OUTPUT

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

Why are we moving `1` into `0x20200008`?

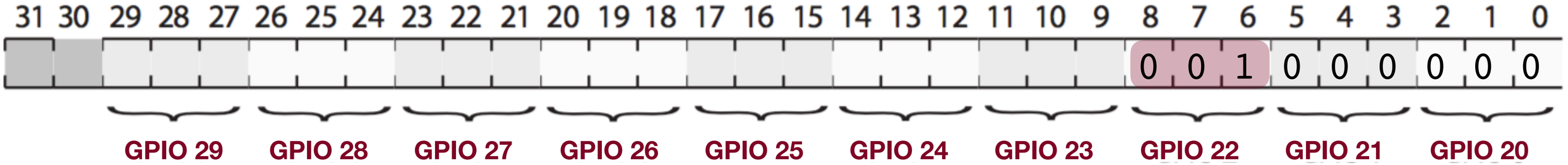


What if we wanted to set GPIO 22 for OUTPUT?

To Configure GPIO 20 for OUTPUT

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

Why are we moving `1` into `0x20200008`?



What if we wanted to set GPIO 22 for OUTPUT?

(Later, we will talk about why `mov` is not the best choice to set a single pin!)

We would have to set bit 6:

```
mov r1, #0x40
```

GPIO Pin Output Set Registers (GPSETn)

[illegible]	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
[illegible]				

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 in 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 *n*	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 *n.*	R/W	0

Table 6-9 – GPIO Output Set Register 1

To Turn on an LED on pin 20

```
// turn on LED connected to GPIO20

// configure GPIO 20 for output
//  FSEL2 = 0x20200008
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x00000008
mov r1, #1    // indicates OUTPUT
str r1, [r0]  // store 1 to address 0x20200008

// set GPI020 to 1
//  SET0 = 0x2020001c
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x0000001c
mov r1, #1
lsl r1, #20   // 1<<20
str r1, [r0]  // store 1<<20 to address 0x2020001c

// loop forever
loop:
b loop
```

To set pin 20, we set the 20th (starting from 0 on the **right**) to `1`.

To Turn on an LED on pin 20

```
// turn on LED connected to GPIO20

// configure GPIO 20 for output
//  FSEL2 = 0x20200008
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x00000008
mov r1, #1    // indicates OUTPUT
str r1, [r0]  // store 1 to address 0x20200008

// set GPI020 to 1
//  SET0 = 0x2020001c
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x0000001c
mov r1, #1
lsl r1, #20   // 1<<20
str r1, [r0]  // store 1<<20 to address 0x2020001c

// loop forever
loop:
b loop
```

To set pin 20, we set the 20th (starting from 0 on the **right**) to `1`.

We shift a `1` left by `20`, and that value will set the correct bit (`lsl` means "left shift logical", which propagates all bits to the left, replacing with 0s)

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

# Copy to SD card
$ cp on.bin /Volumes/CS107E/kernel.img

# Eject and remove SD card

# Insert SD card into SDHC slot on raspi
# Apply power using usb console cable ...
```

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

# Copy to SD card
$ cp on.bin /Volumes/CS107E/kernel.img

# Eject and remove SD card

# Insert SD card into SDHC slot on raspi
# Apply power using usb console cable ...
```