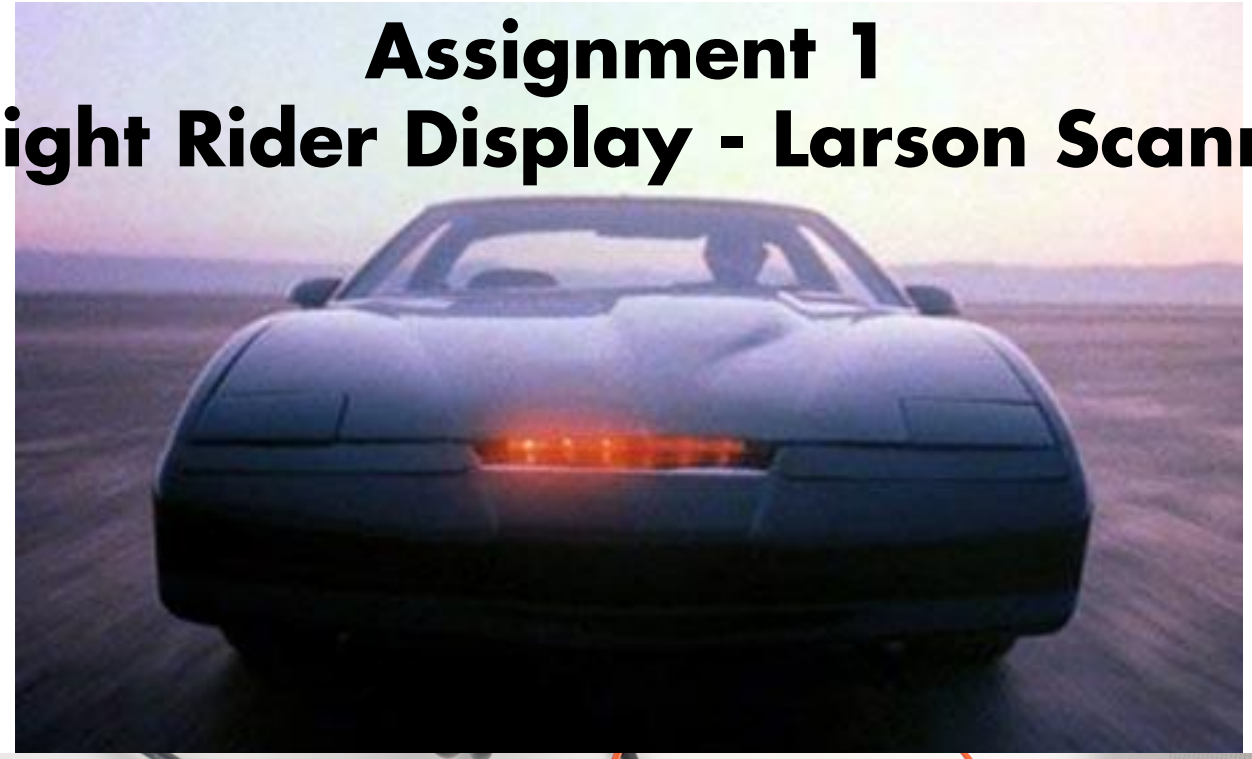
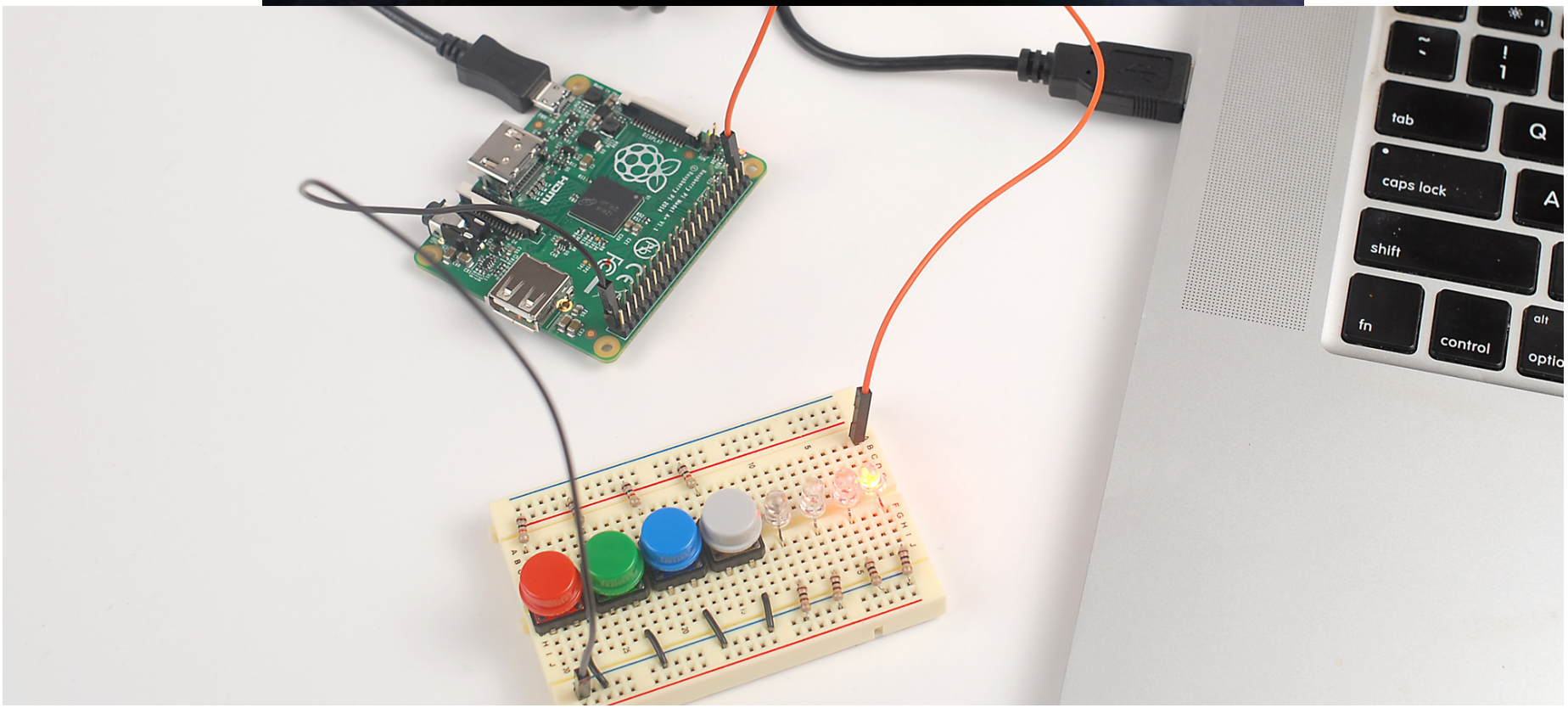
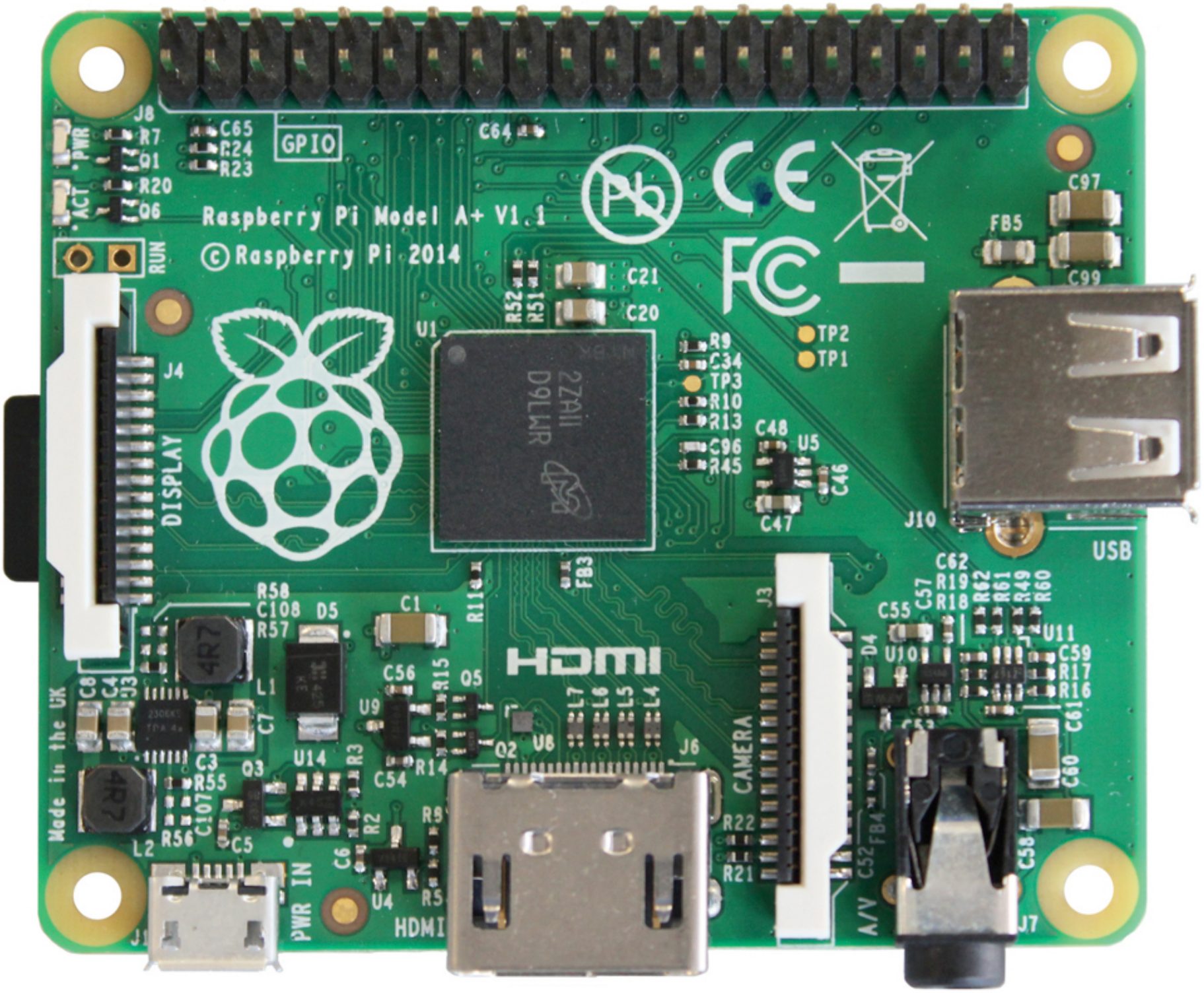
# ARM
# Processor and Memory Architecture

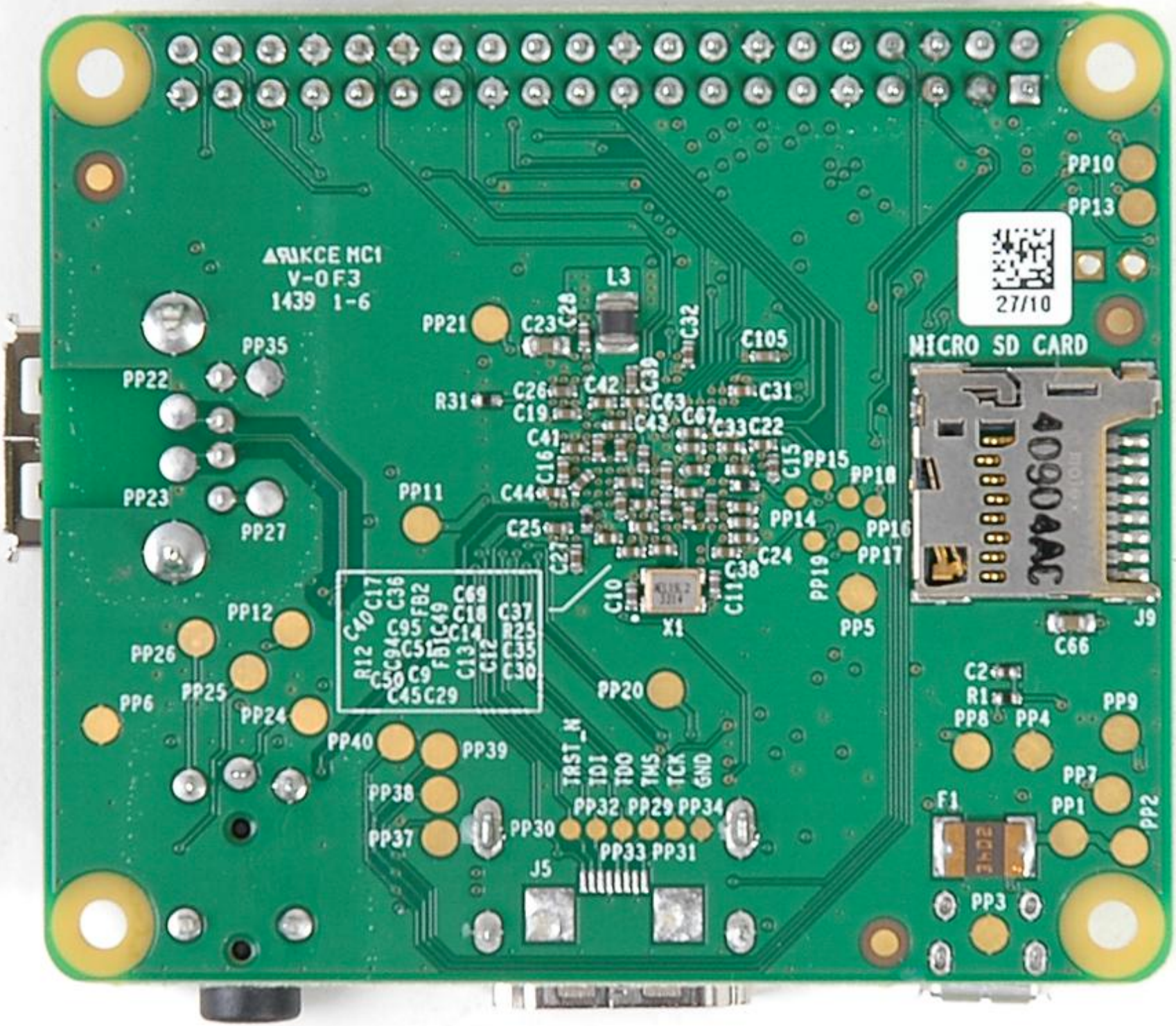
# Goal: Turn on an LED

# Assignment 1
# Knight Rider Display - Larson Scanner

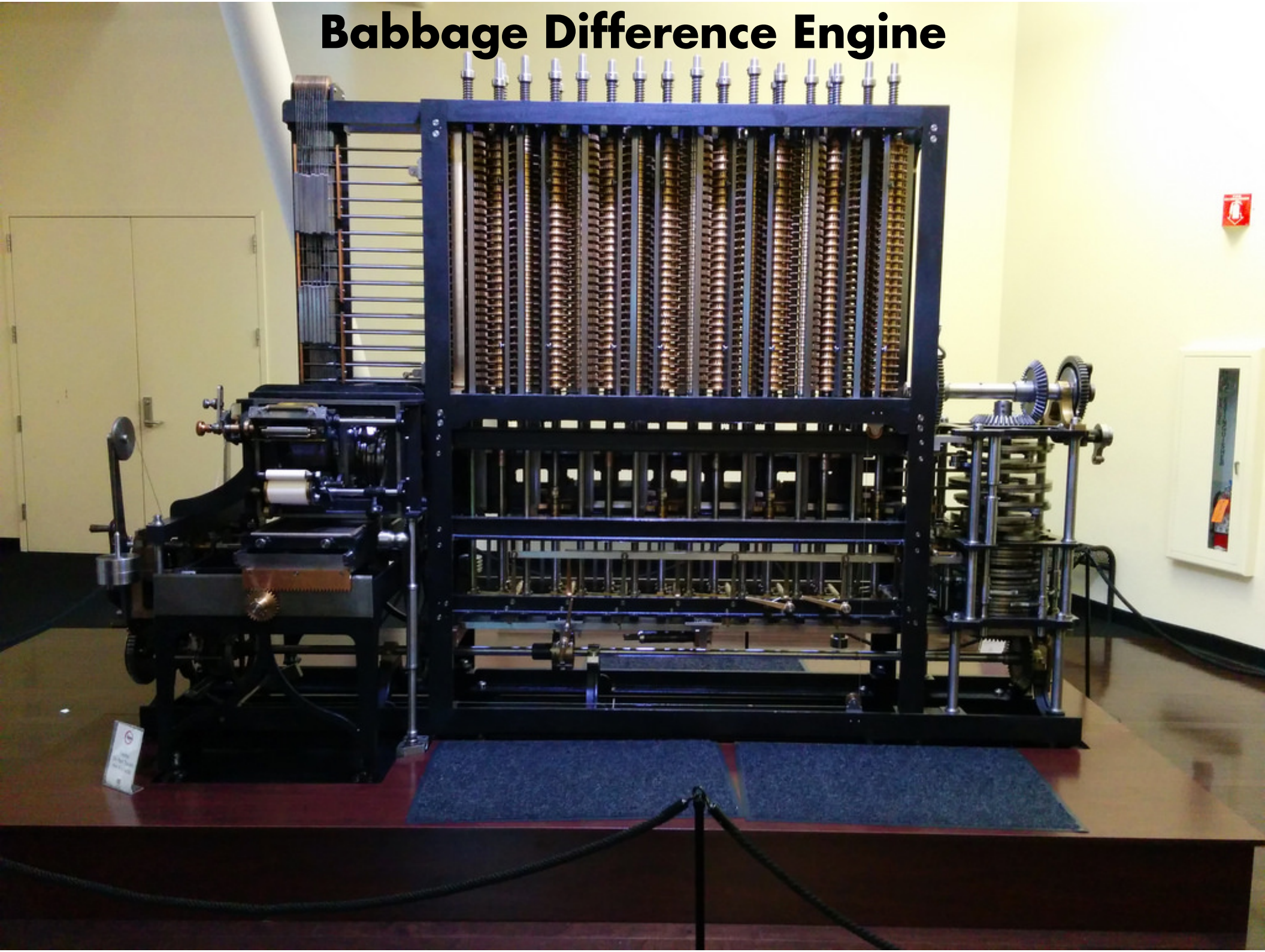


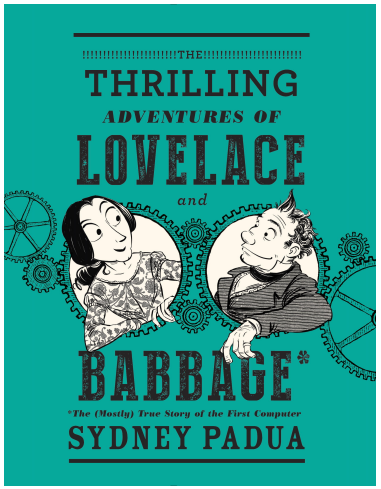



GPIO
Raspberry Pi Model A+ V1.1
© Raspberry Pi 2014
DISPLAY
HDMI
CAMERA
USB
A/V
PWR IN
HDMI
Made in the UK


ARIKCE MC1
V-0 F3
1439 1-6
27/10
MICRO SD CARD
40904AC
L3
X1
J5
J9
F1
PP10
PP13
PP35
PP22
PP23
PP27
PP11
PP21
PP12
PP26
PP25
PP6
PP24
PP40
PP39
PP38
PP37
PP30
PP20
PP32
PP29
PP34
PP33
PP31
PP15
PP18
PP14
PP16
PP17
PP19
PP5
PP8
PP4
PP9
PP7
PP1
PP2
PP3
C66
TRST_N
TDI
TDO
TMS
TCK
GND

# Babbage Difference Engine





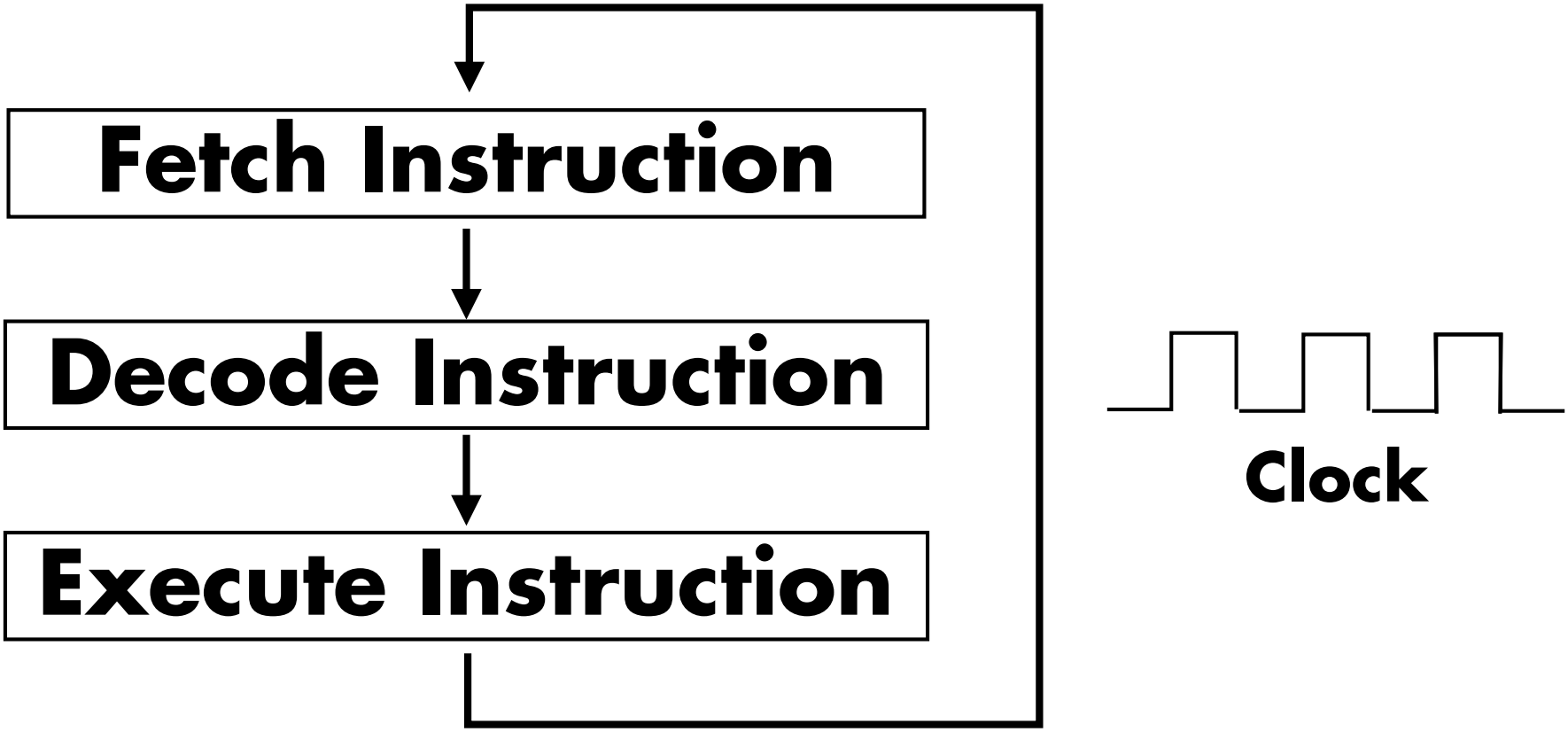

# Analytical Engine



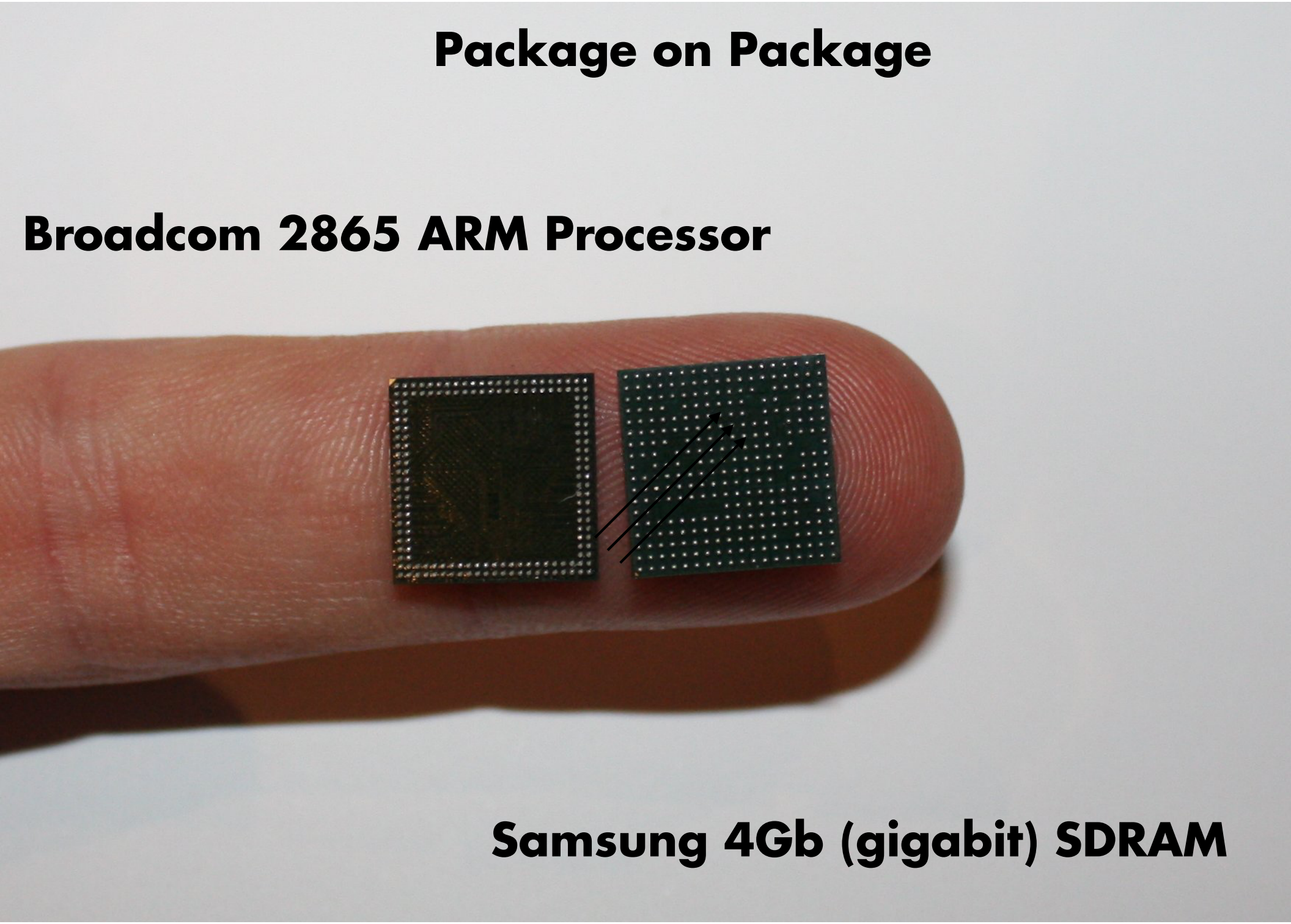

# *Universal Computer*

**http://www.theguardian.com/technology/2015/apr/12/thrilling-adventures-ada-lovelace-charles-babbage-sydney-padua**

# Running a "Program"



# Package on Package

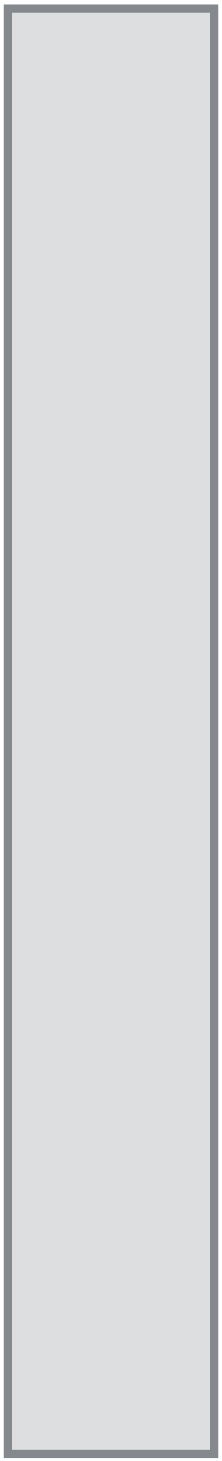


**Memory used to store both instructions and data**

**Storage locations are accessed using 32-bit addresses**

**Maximum addressable memory is 4 GB (gigabyte)**

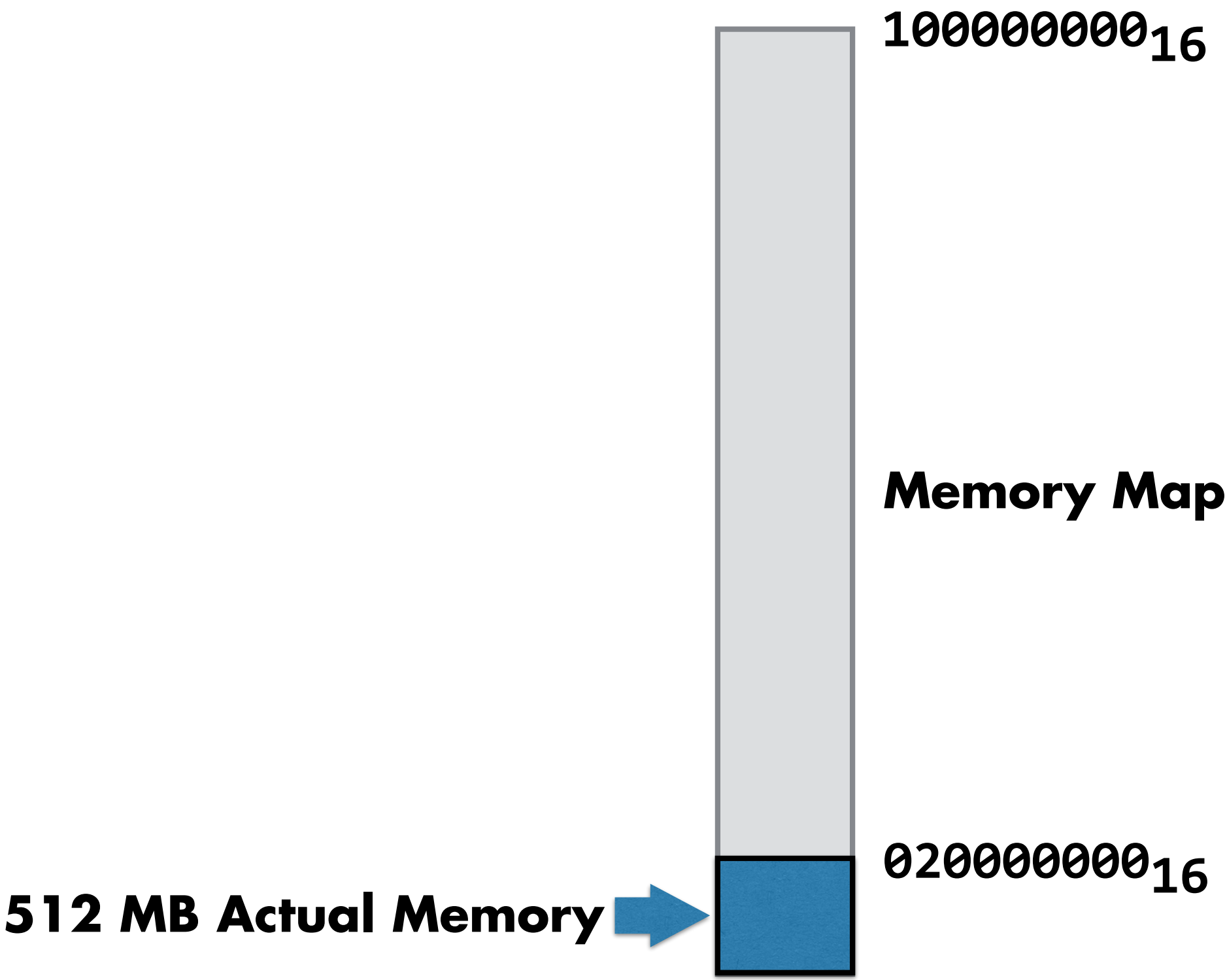
**Address refers to a *byte (8-bits)***



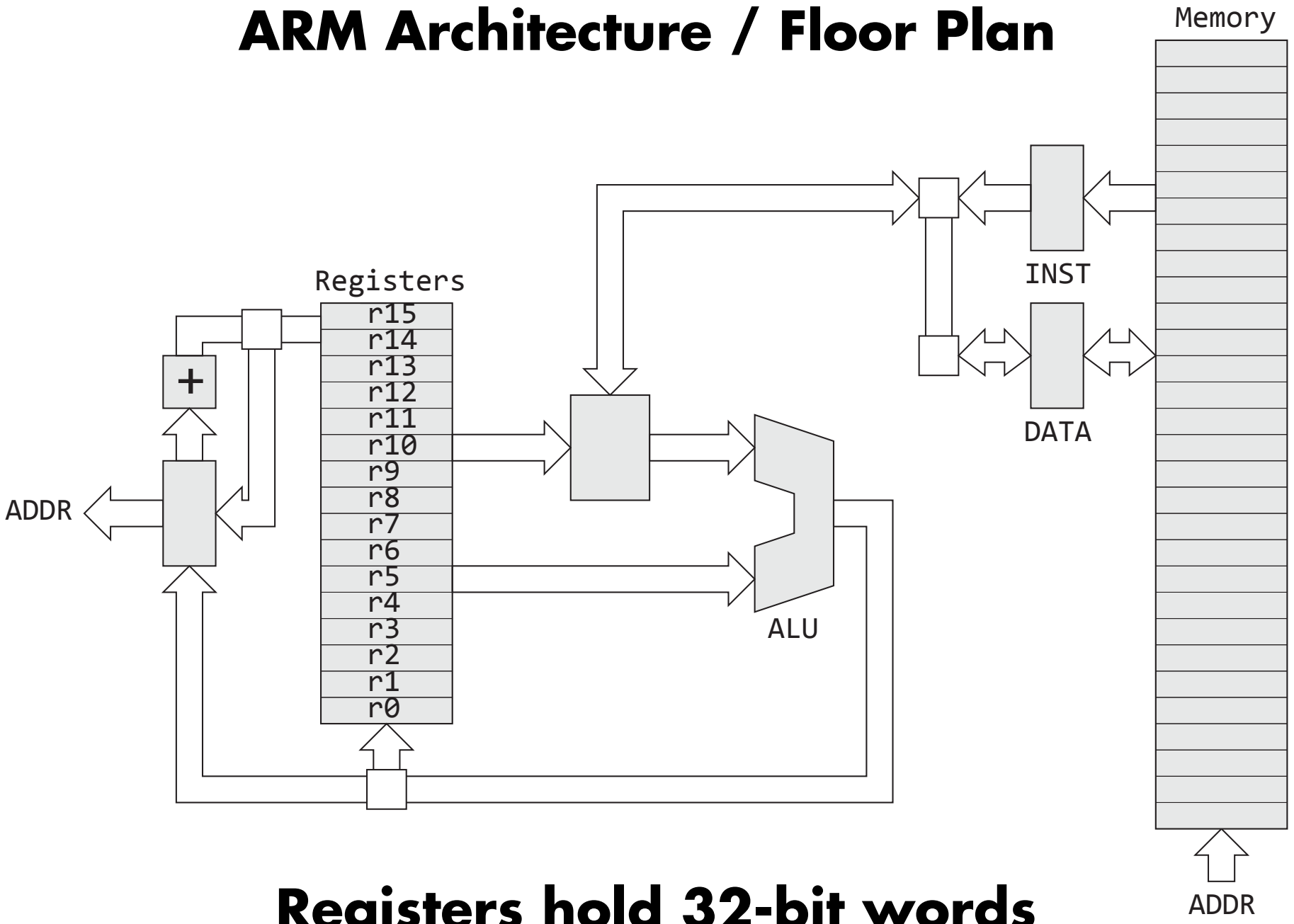
$100000000_{16}$

**Memory Map**

$00000000_{16}$


$100000000_{16}$
Memory Map
$020000000_{16}$
512 MB Actual Memory

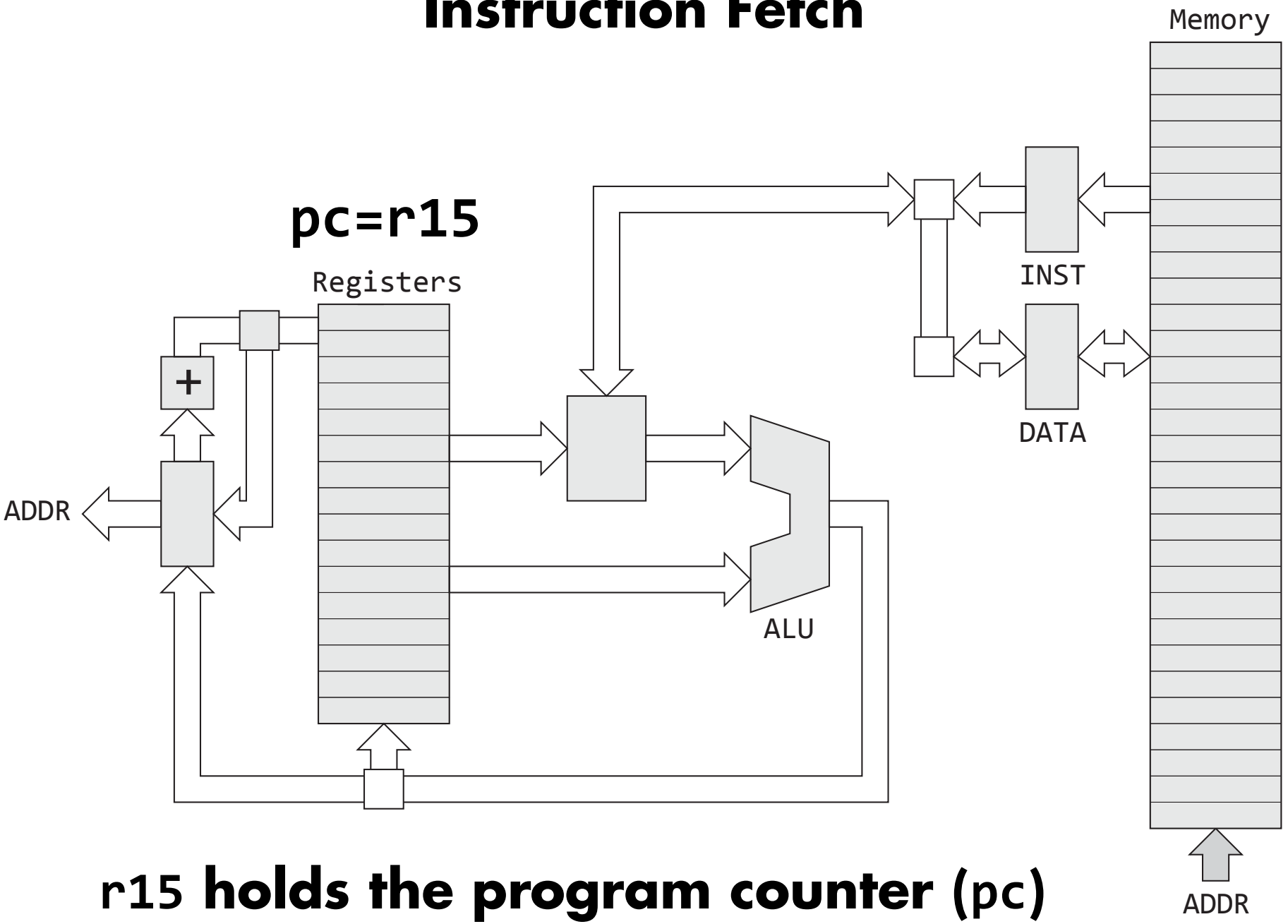
# ARM Architecture / Floor Plan




**Registers hold 32-bit words**

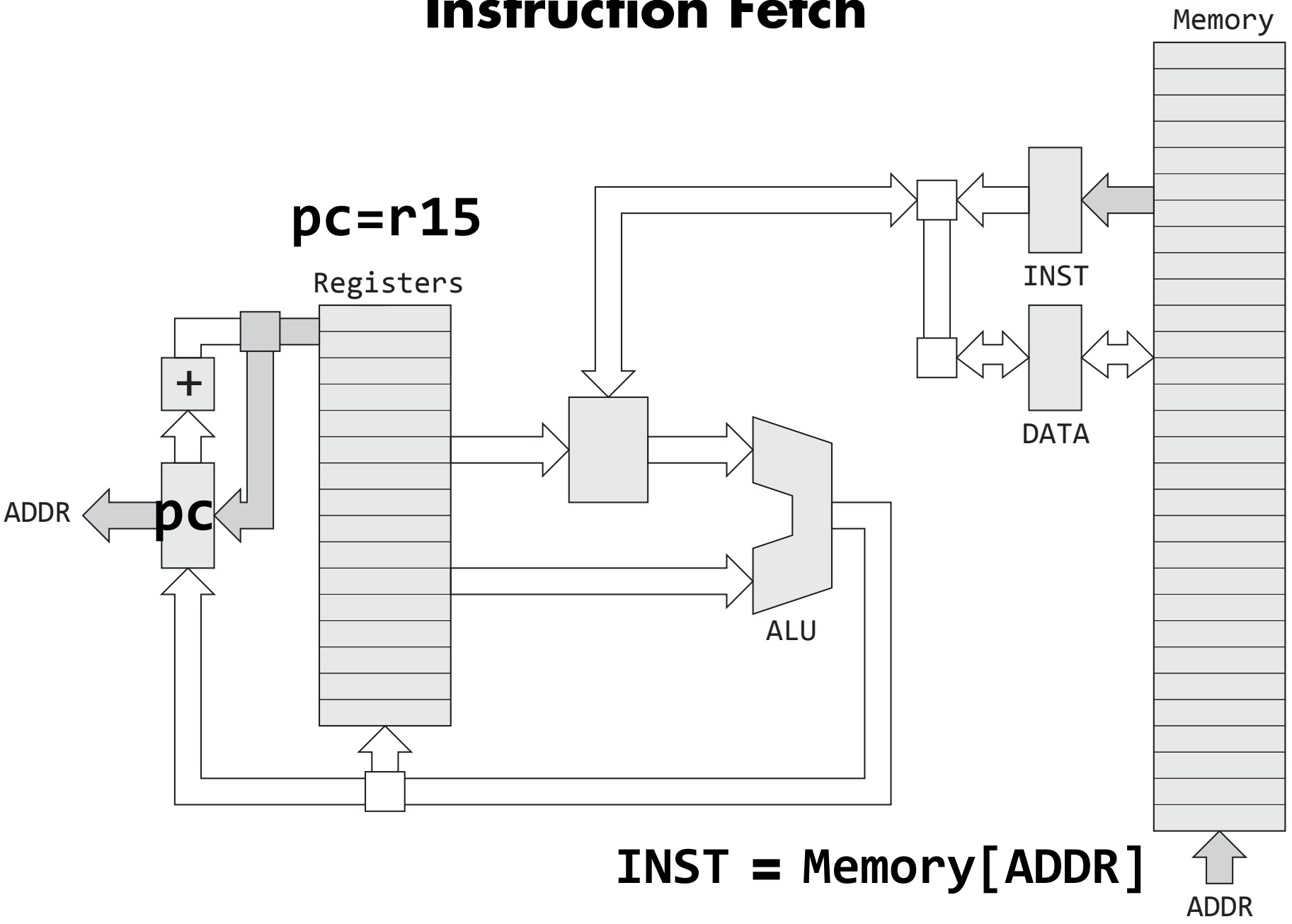
**Arithmetic-Logic Unit (ALU) operates on 32-bit words**

# Instruction Fetch



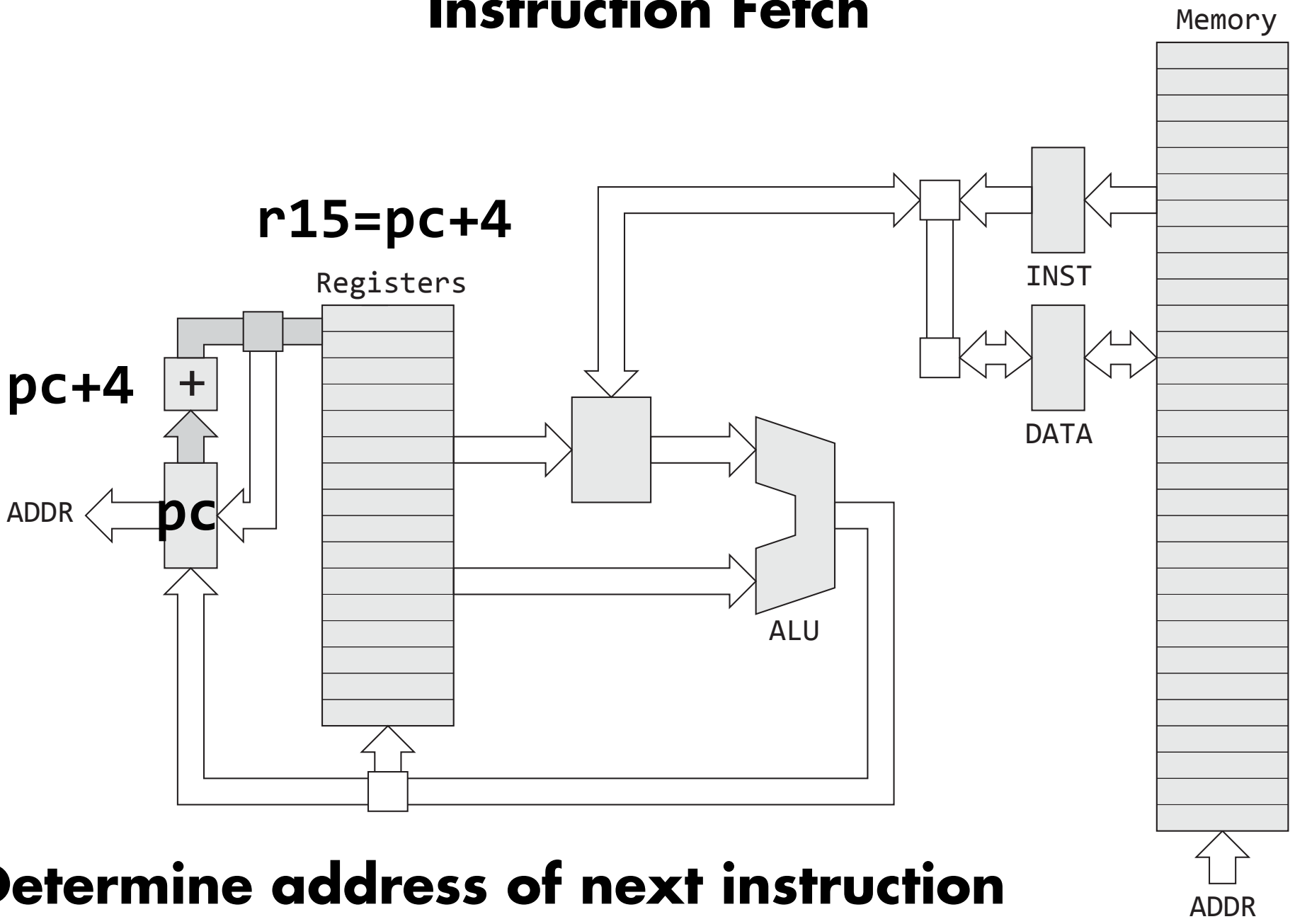

**r15 holds the program counter (pc)**

# Instruction Fetch




**Addresses and instructions are 32-bit words**

# Instruction Fetch

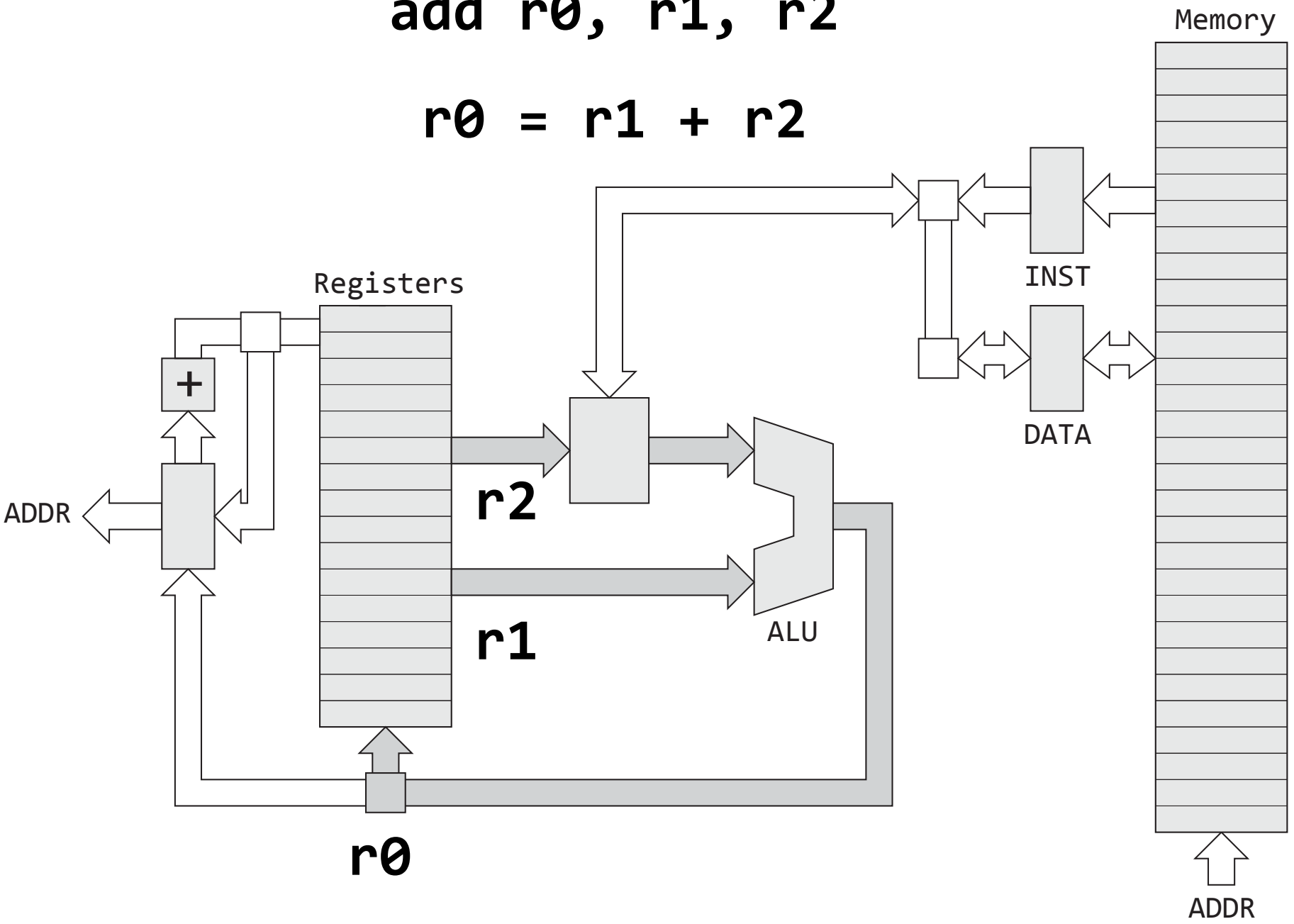



**Determine address of next instruction**

**Why pc+4?**

# Arithmetic-Logic Unit (ALU)

```
add r0, r1, r2

r0 = r1 + r2
```




**ALU operates on 32-bit words**

# Add Instruction

**Meaning (defined as math or C code)**

```
r0 = r1 + r2
```

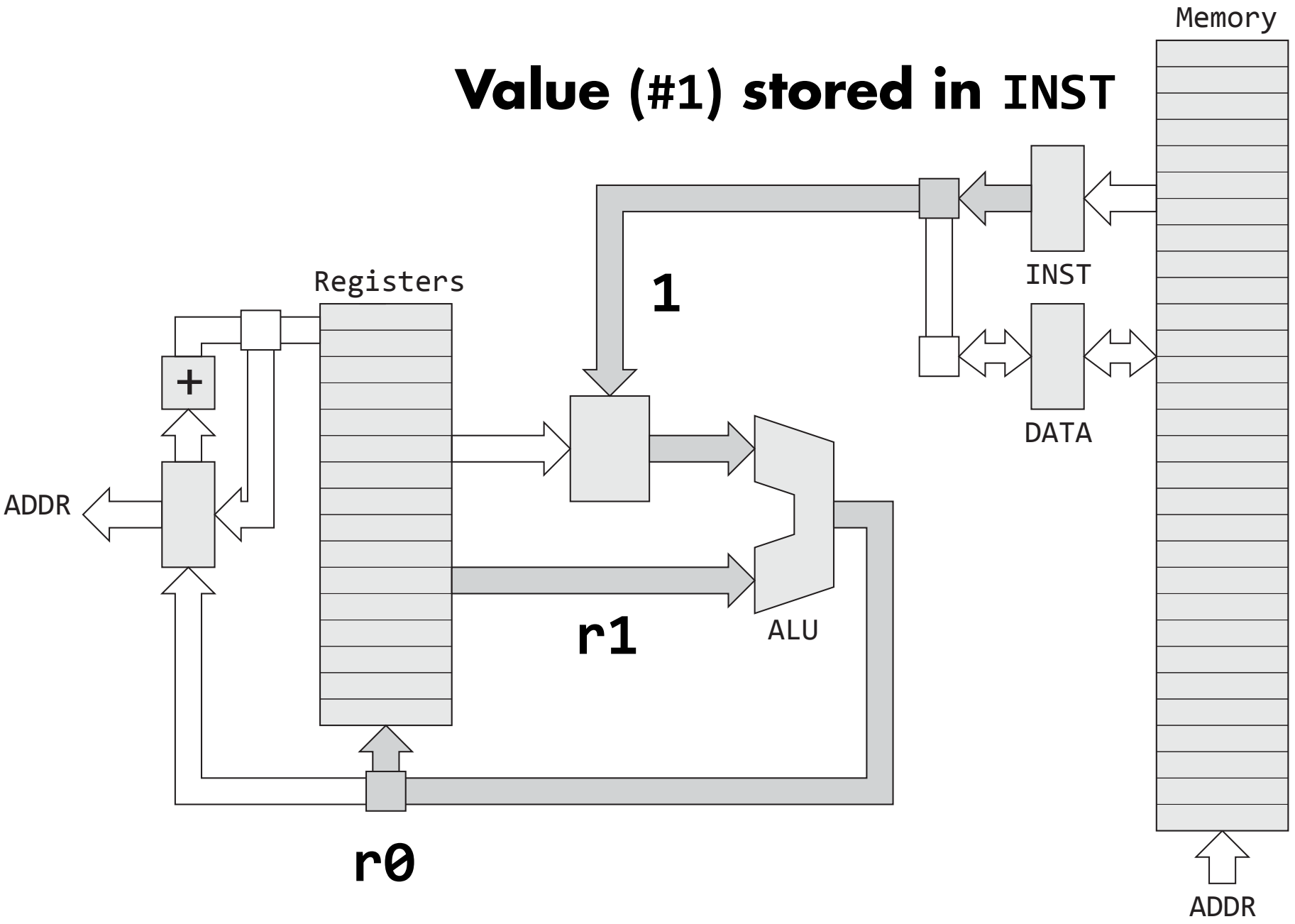
**Assembly language (result is leftmost register)**

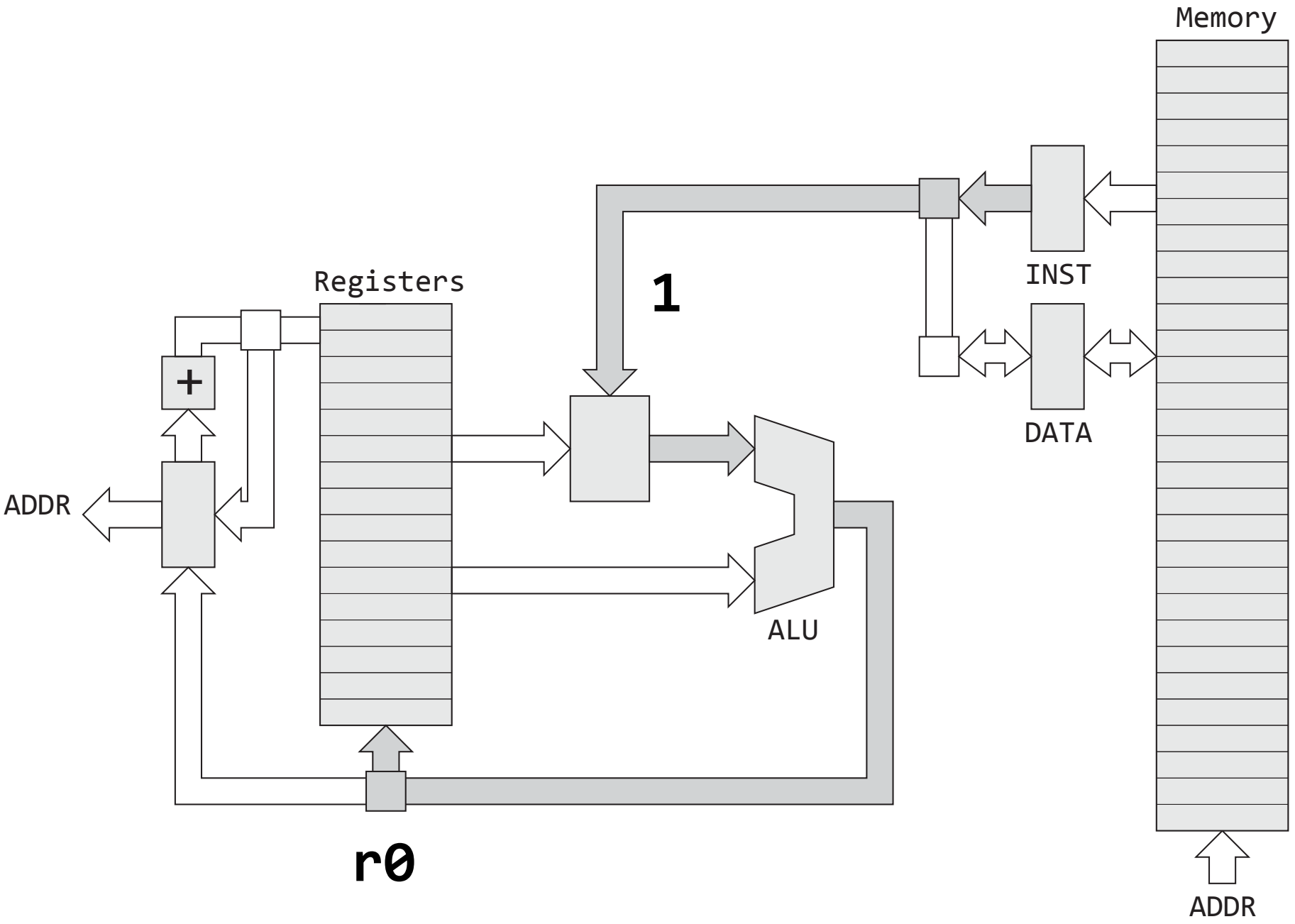
```
add r0, r1, r2
```

**Machine code (more on this later)**

```
E0 81 00 02
```

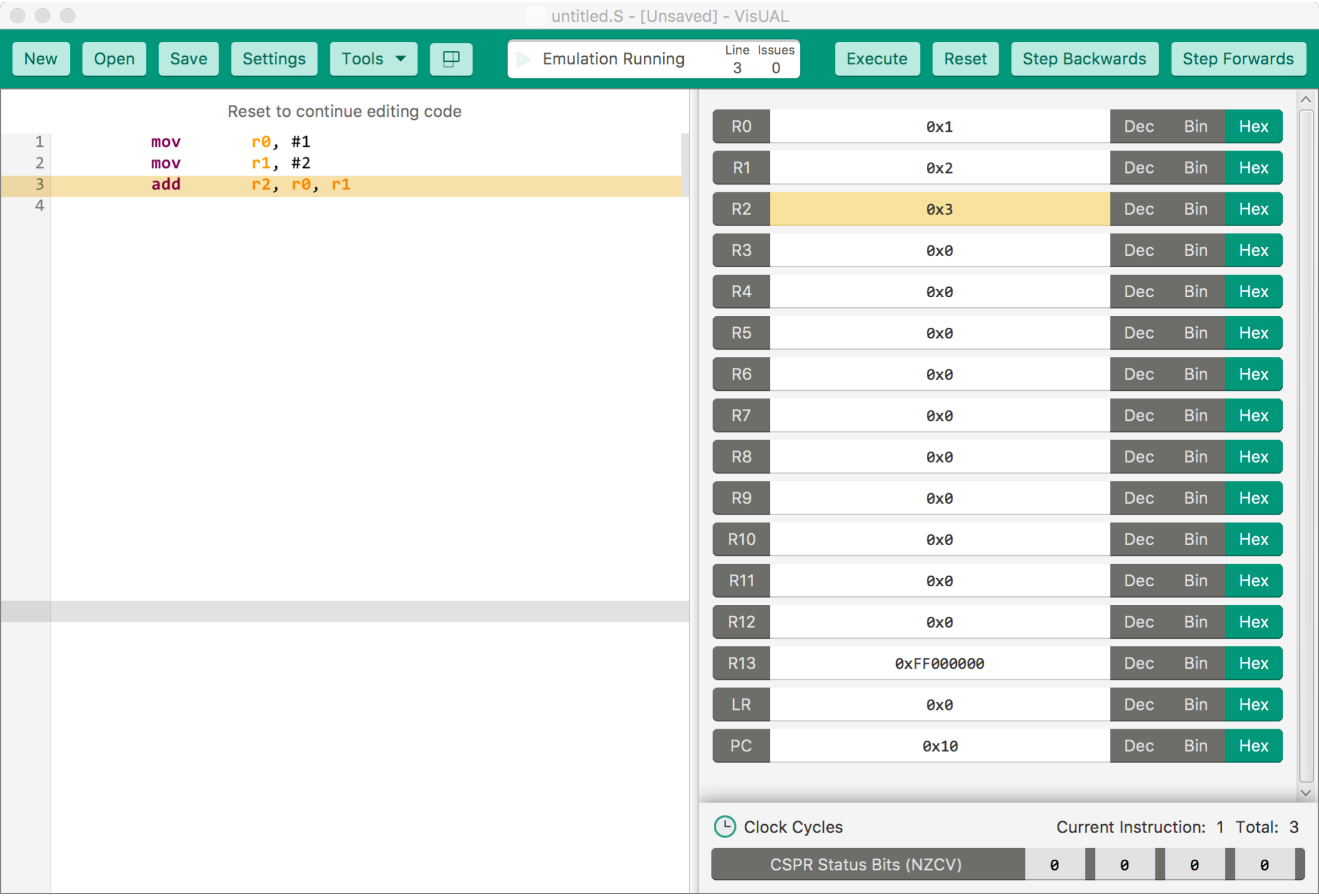
```
# Assemble (.s) into 'object' file (.o)
% arm-none-eabi-as add.s -o add.o

# 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
```


Memory
Value (#1) stored in INST
INST
DATA
1
Registers
+
ADDR
r1
ALU
r0
ADDR
add r0, r1, #1




**mov r0, #1**

# VisUAL


untitled.S - [Unsaved] - VisUAL
New
Open
Save
Settings
Tools
Emulation Running
Line 3
Issues 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
4
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

# Conceptual Questions

1. **What is the difference between a memory location and a register?**

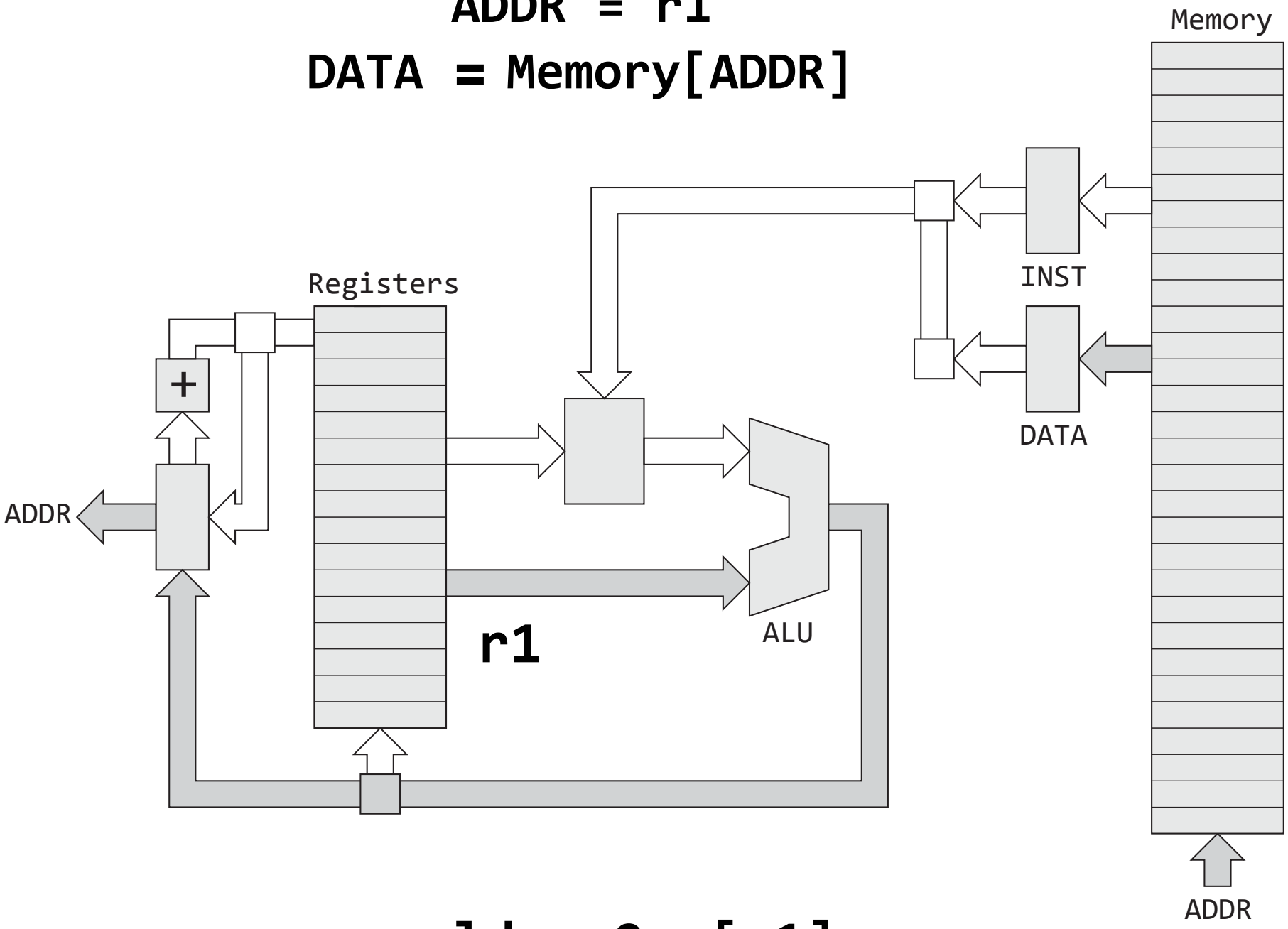
2. **Suppose your program starts at `0x8000`, what assembly language program will jump to and start executing instructions at that location.**

3. **If all instructions are 32-bits, can you move any 32-bit constant value to a register using a `mov` instruction?**

# Load and Store Instructions

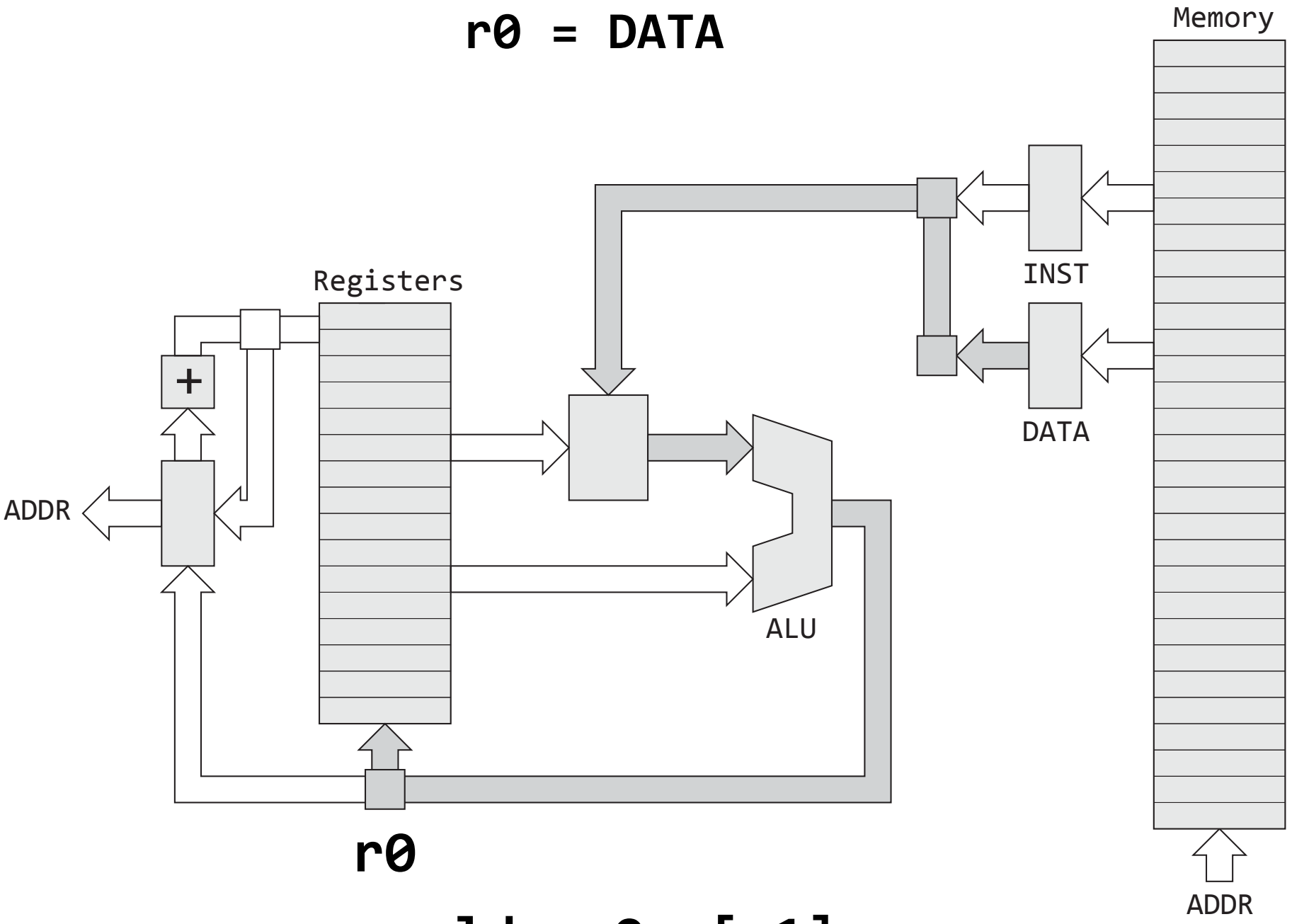
**ADDR = r1**

**DATA = Memory[ADDR]**

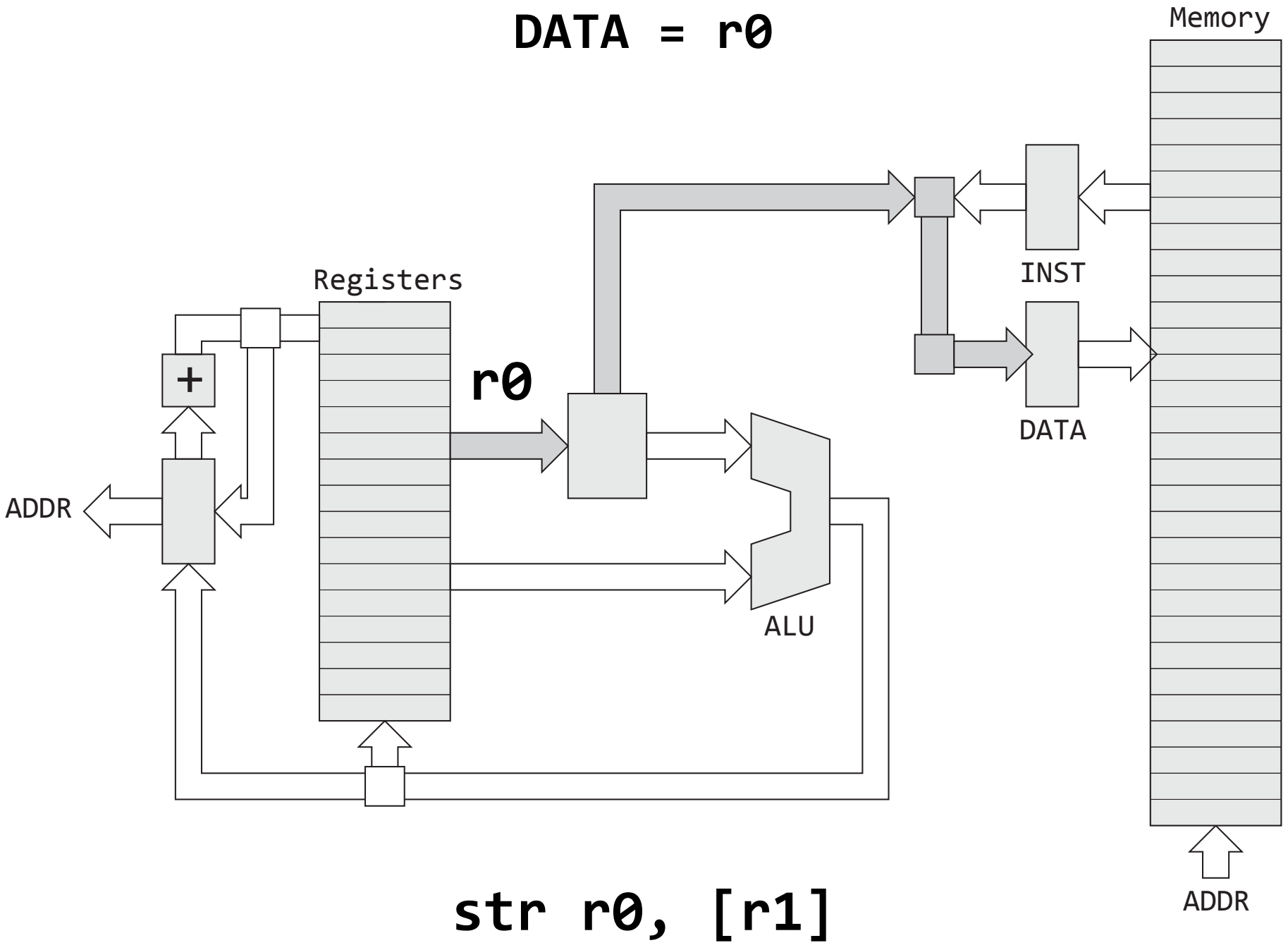



**ldr r0, [r1]**

# r0 = DATA

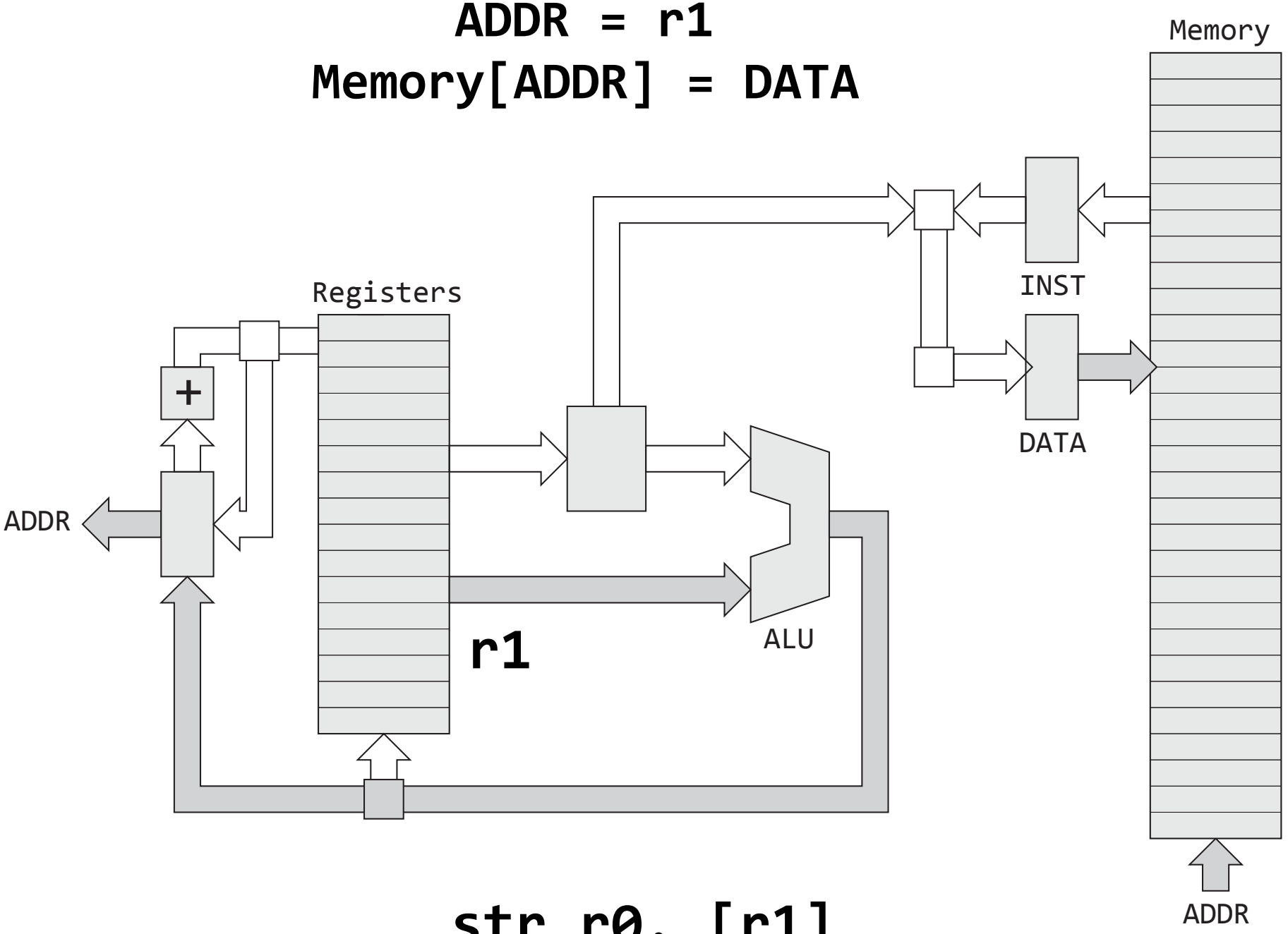



# ldr r0, [r1]

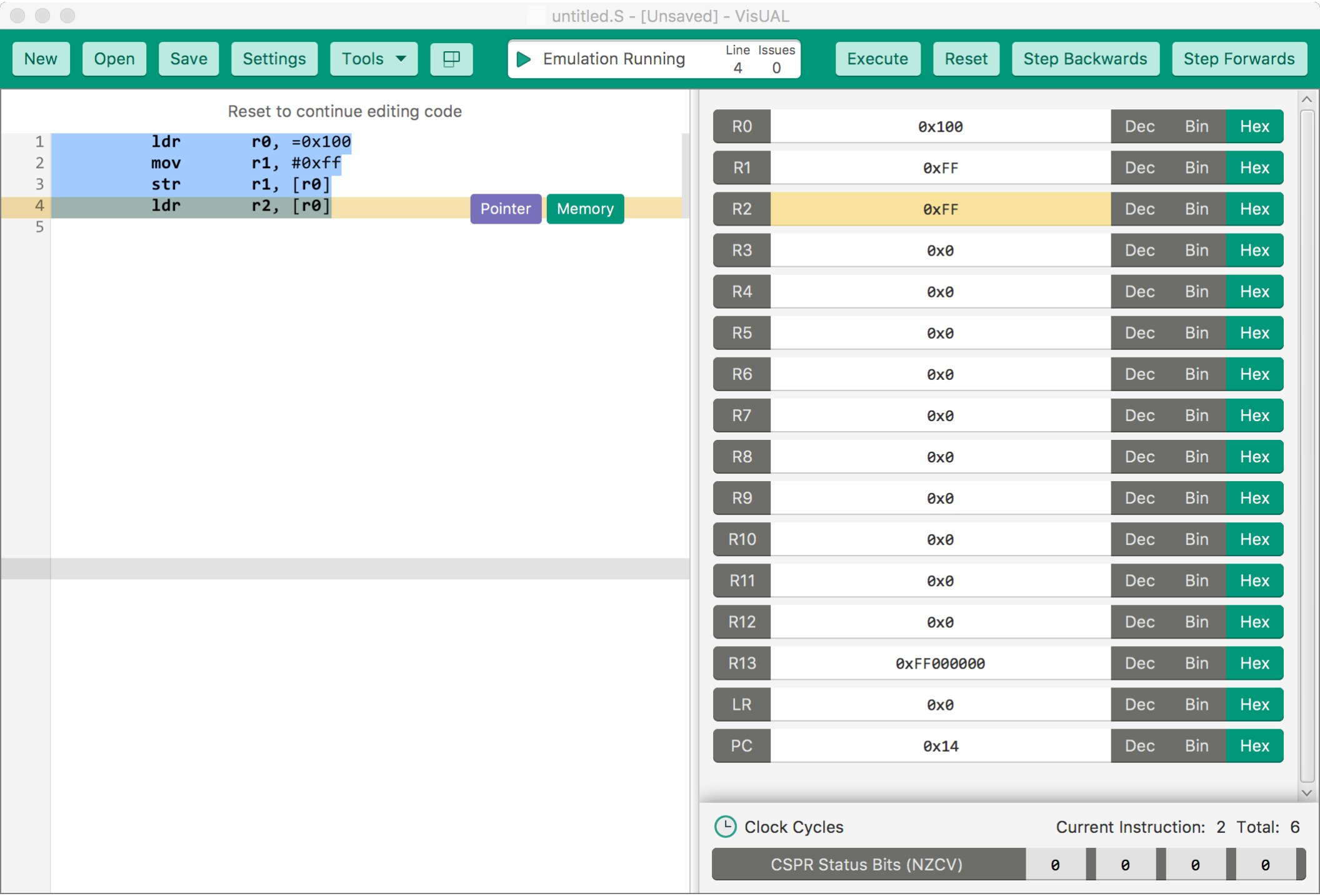

DATA = r0
Memory
INST
DATA
Registers
r0
+
ADDR
ALU
ADDR
str r0, [r1]

**ADDR = r1**

**Memory[ADDR] = DATA**

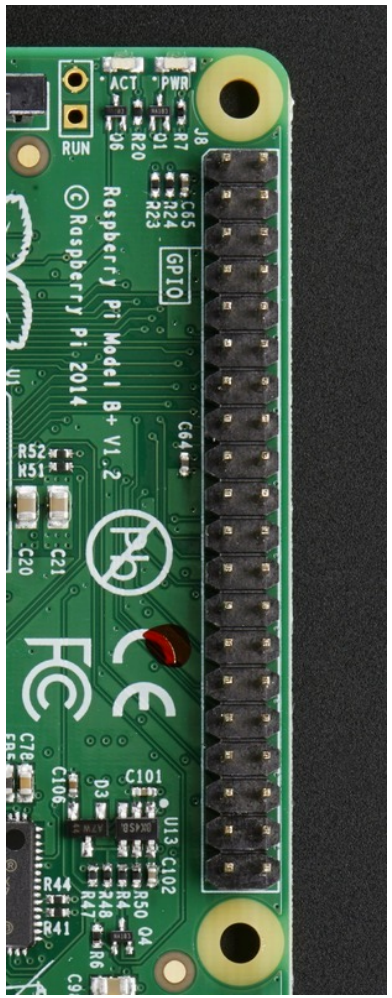


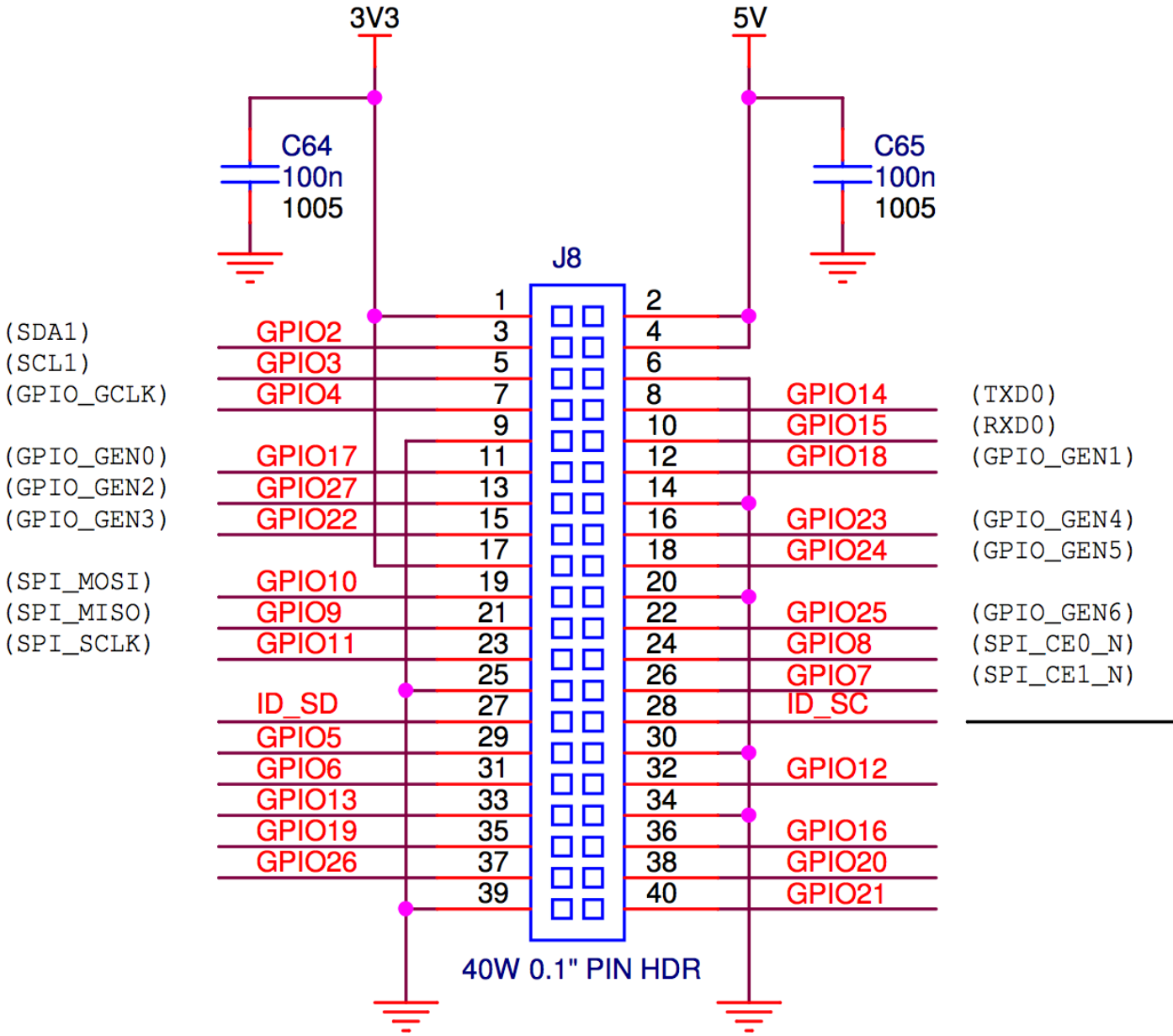

**str r0, [r1]**


untitled.S - [Unsaved] - VisUAL
New
Open
Save
Settings
Tools
Emulation Running
Line 4
Issues 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]
5
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





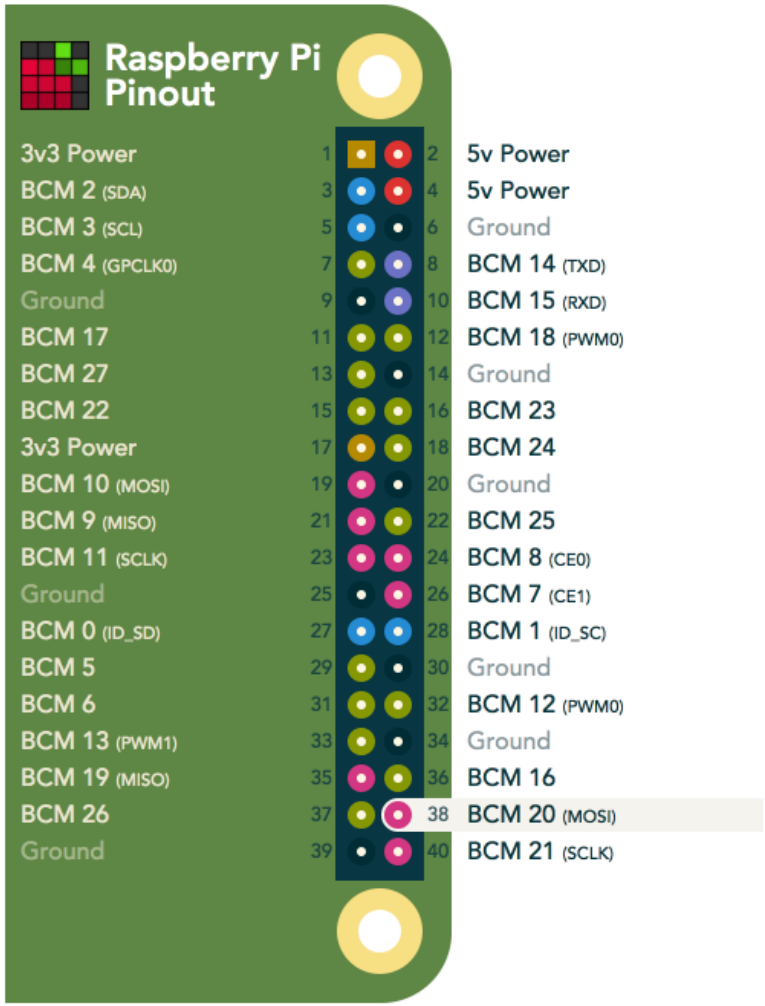

**54 GPIO Pins**

BCM 20 (SPI Master-Out) at Ra ×

Patrick

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

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




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)

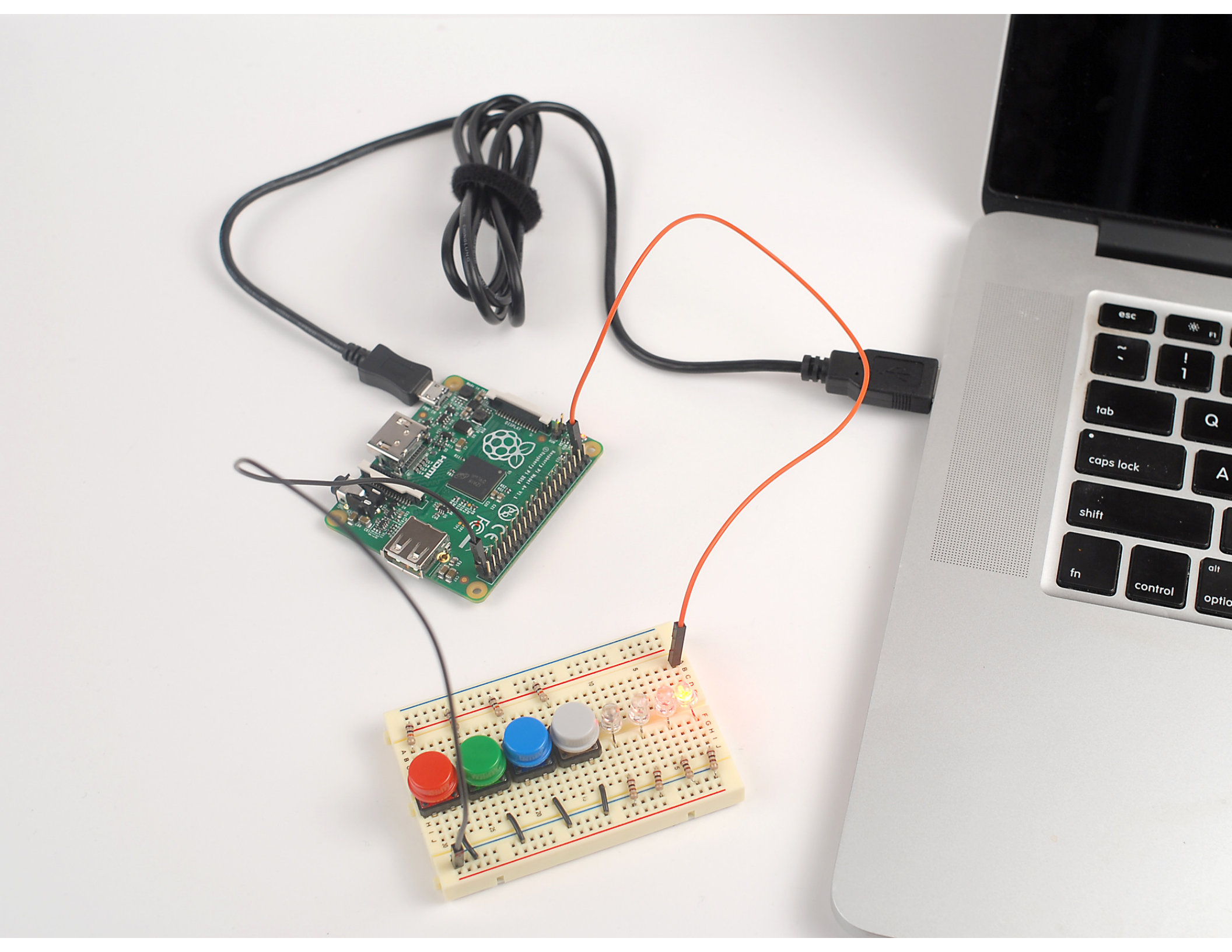
| Alt0 | Alt1 | Alt2 | Alt3 | Alt4 | Alt5 |
|---|---|---|---|---|---|
| PCM DIN | SMI SD12 | DPI D16 | I2CSL MISO | SPI1 MOSI | GPCLK0 |

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

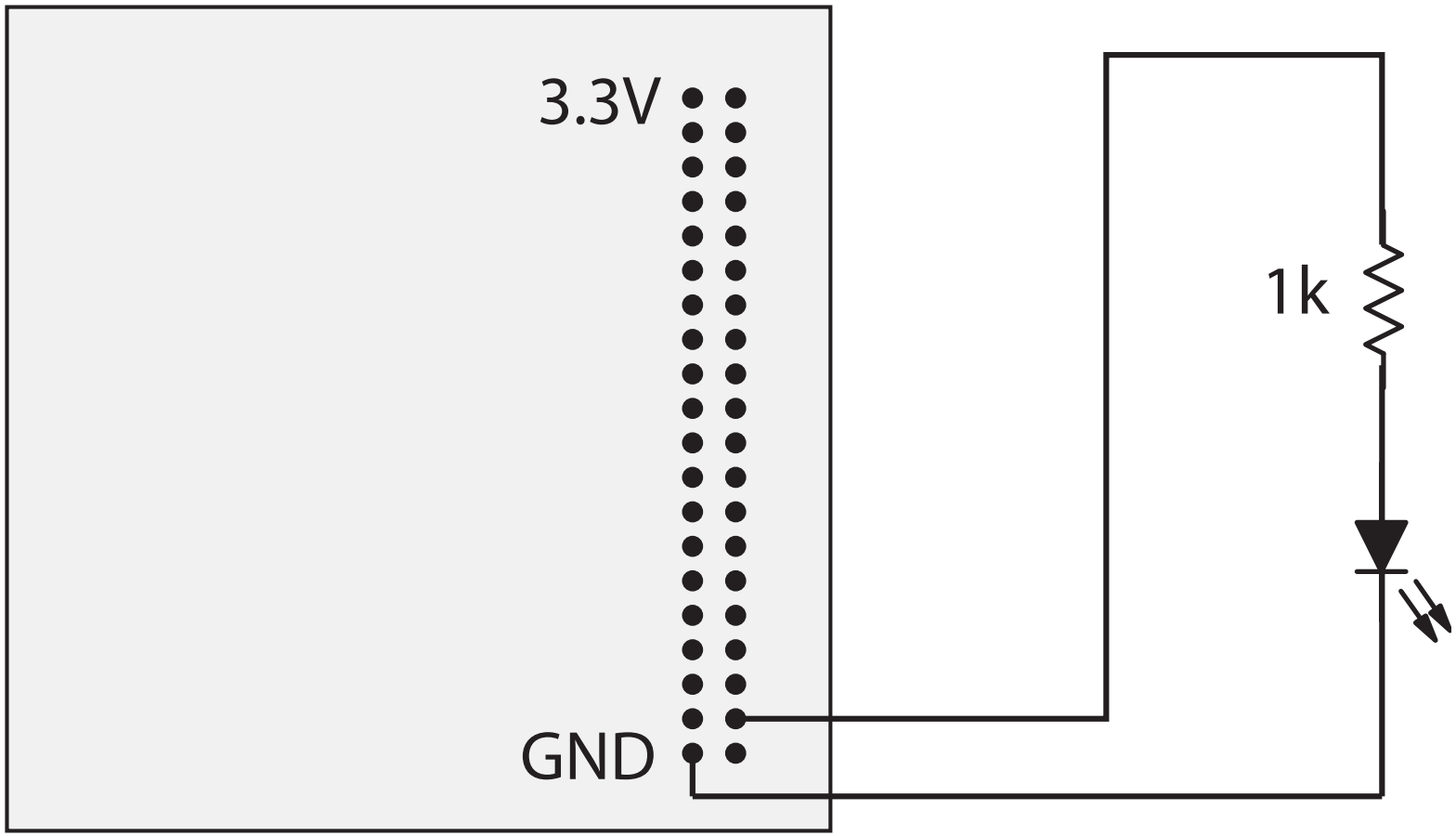
Spotted an error, want to add your board's pinout? Head on over to our GitHub repository and submit an Issue or a Pull Request!

lesson7.html

Show All


esc
tab
caps lock
shift
fn
control

# Connect LED to GPIO 20




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

**GPIO Pins are a *Peripheral***

**Peripherals are Controlled by Special Registers**

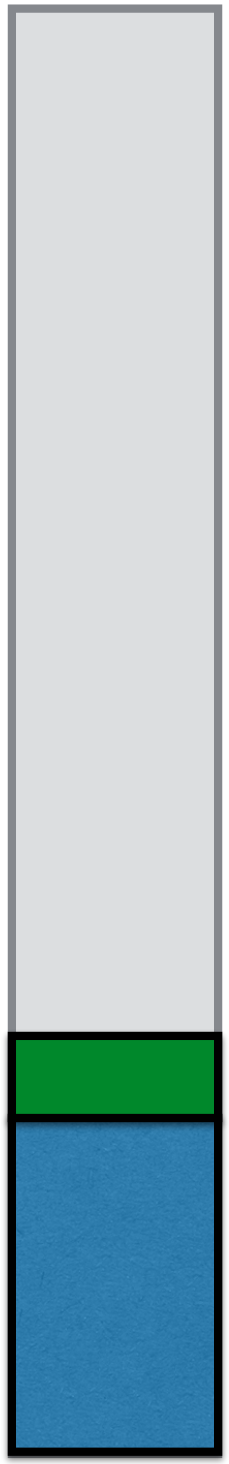
**"Peripheral Registers"**

# Memory Map

**Peripheral registers are mapped into address space**

**Memory-Mapped IO (MMIO)**

**MMIO space is above physical memory**



$100000000_{16}$

**4 GB**

$020000000_{16}$

**512 MB**

**Ref: BCM2835-ARM-Peripherals.pdf**

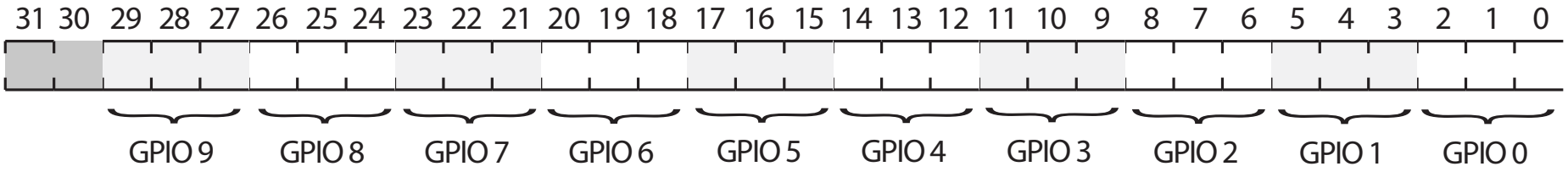
# General-Purpose IO Function

## GPIO Pins can be configured to be INPUT, OUTPUT, or ALT0-5

| 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 ALT0-5**

**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
```

**Ref: BCM2835-ARM-Peripherals.pdf**

```
// 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 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<br>1 = Set GPIO pin *n* | R/W | 0 |

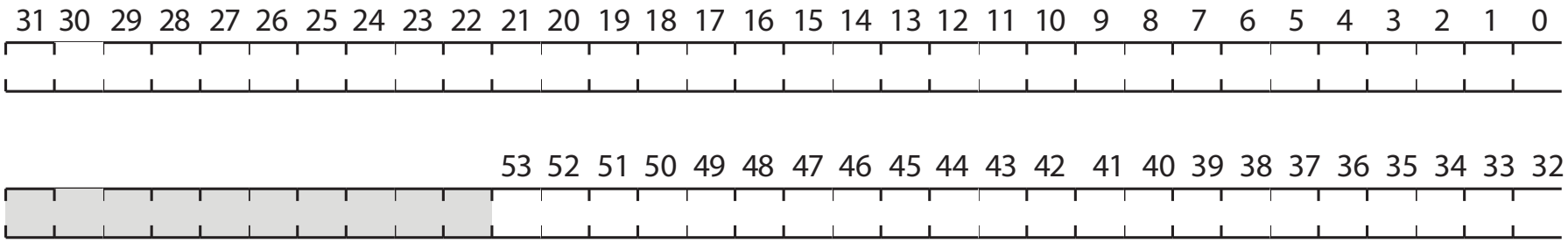
**Table 6-8 – GPIO Output Set Register 0**

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

```
…

//  SET1 = 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
```

```
…

//  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
```

```
# 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!!
#
```

# Concepts

Memory stores both instructions and data

Bits, bytes, and words; bitwise operations

Different types of ARM instructions

- ALU
- Loads and Stores
- Branches

GPIOs, peripheral registers, and MMIO