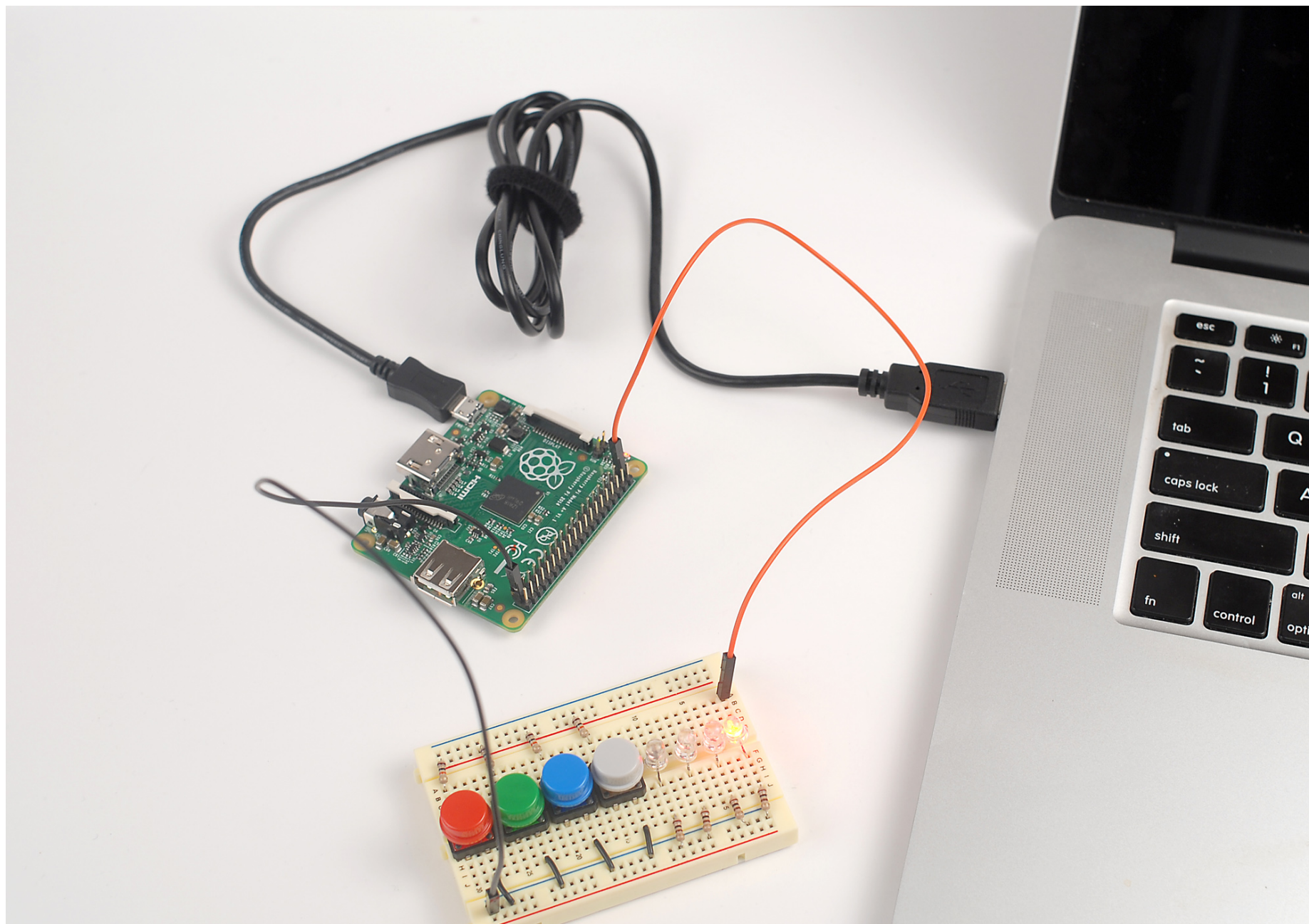


ARM

**Processor and Memory
Architecture**

Goal: Turn on an LED

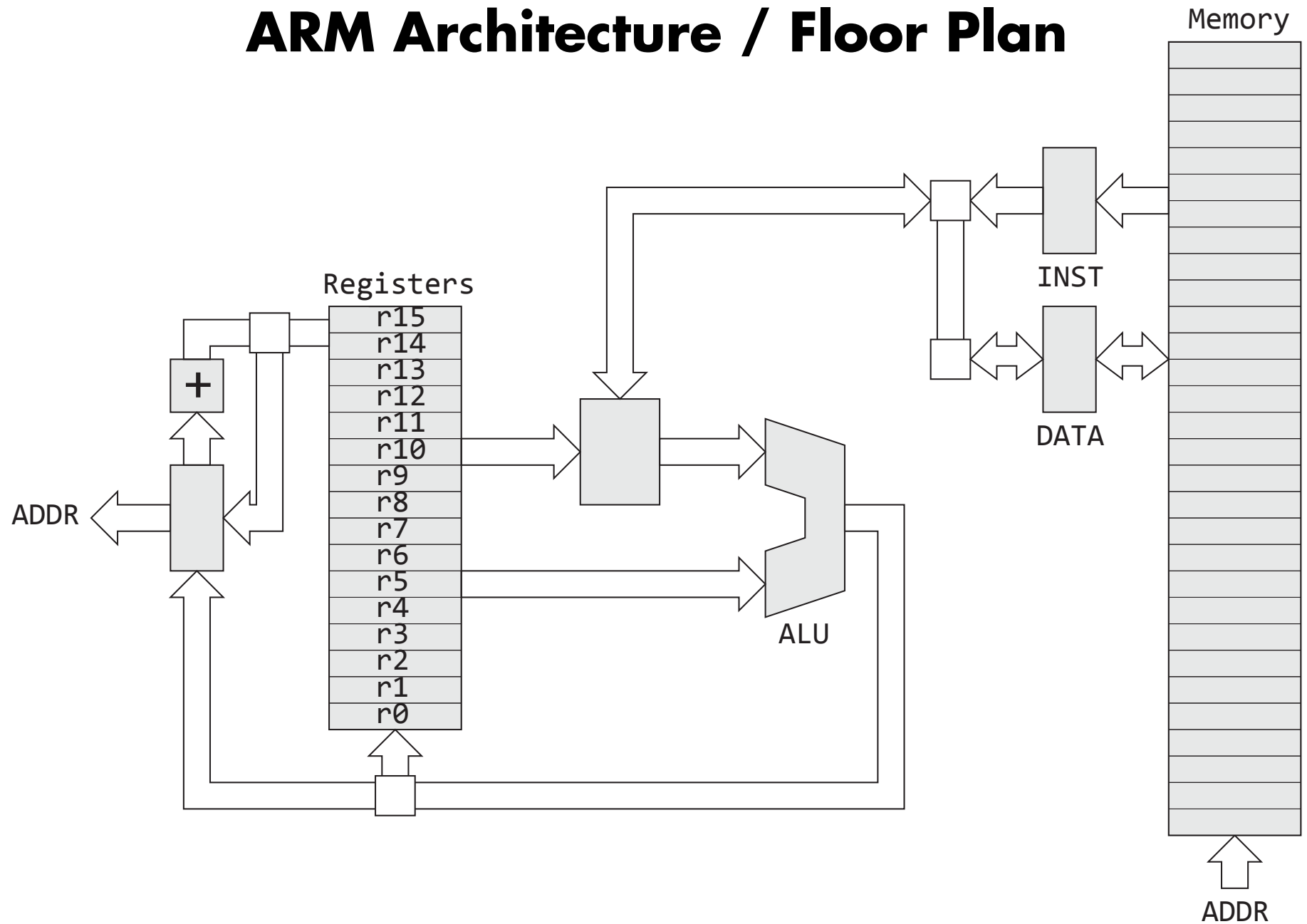


Assignment 1

Knight Rider Display - Larson Scanner



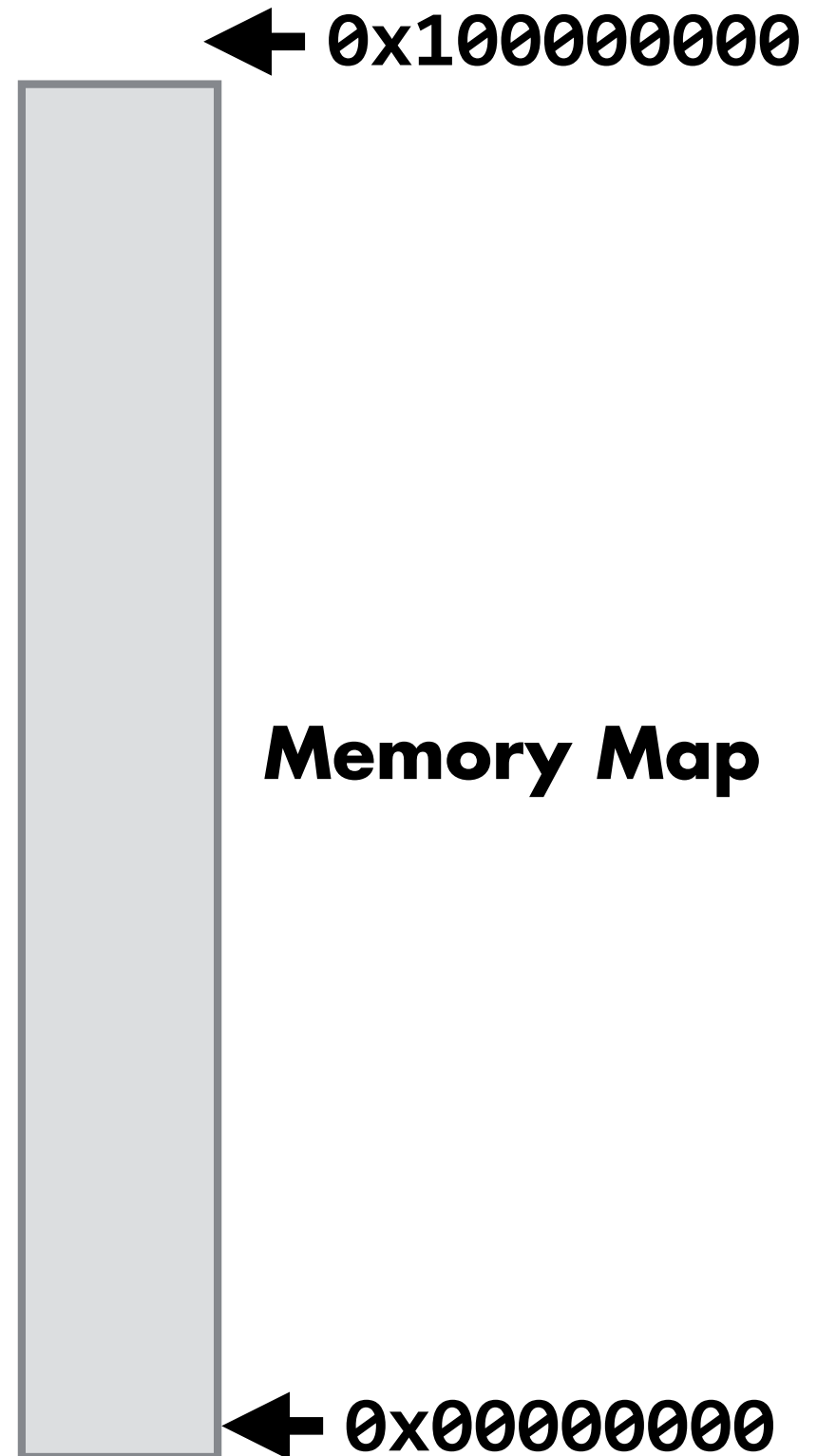
ARM Architecture / Floor Plan



Memory is a large array

Storage locations are accessed using a 32-bit index, called the *address*

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





Fred Brooks
1932-2022

Brooks was asked "What do you consider your greatest technological achievement?"

Brooks responded, "The most important single decision I ever made was to change the IBM 360 series from a 6-bit byte to an 8-bit byte, thereby enabling the use of lowercase letters. That change propagated everywhere."

<https://www.wired.com/2010/07/ff-fred-brooks/>

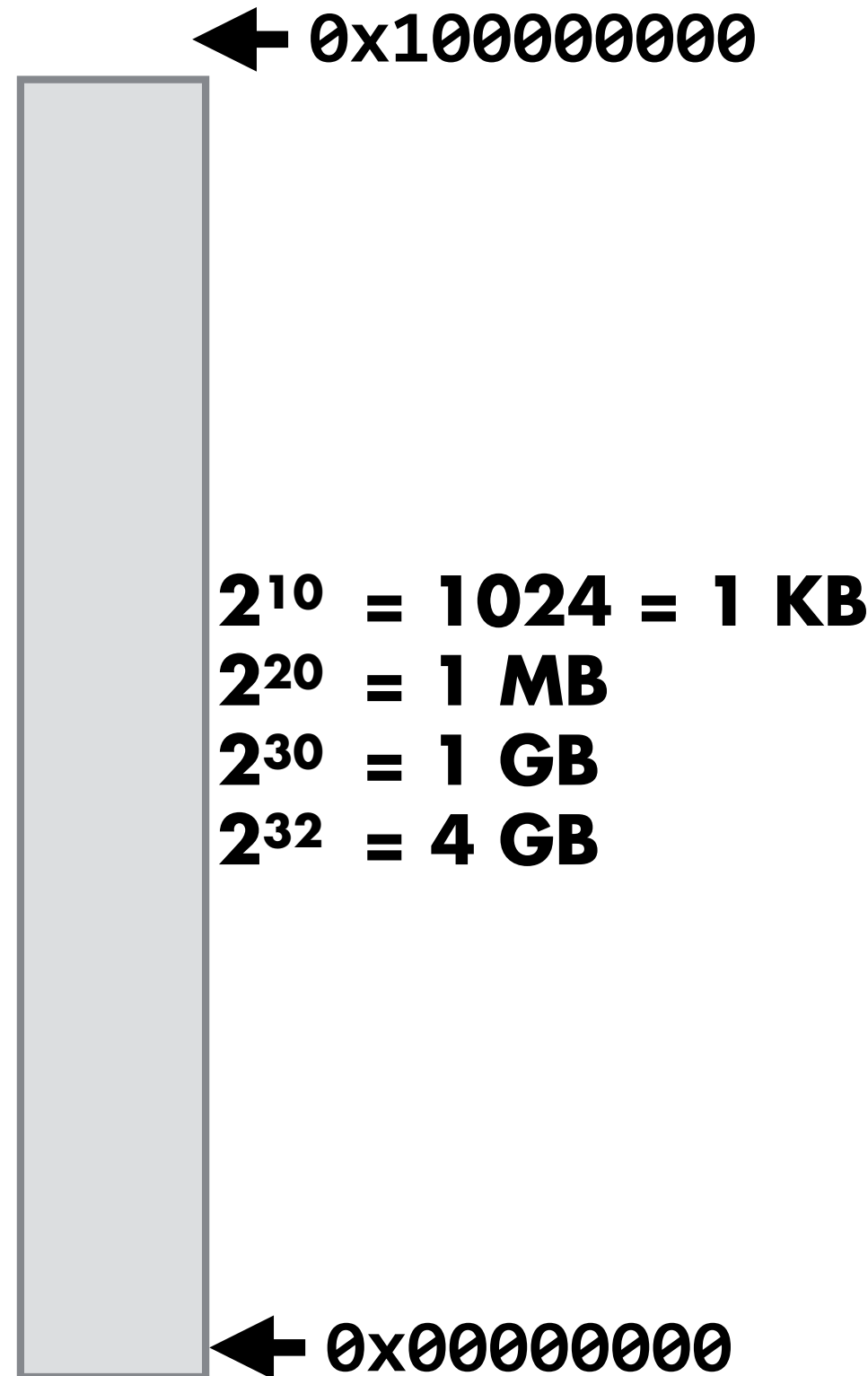
Memory is a large array

Storage locations are accessed using a 32-bit index, called the *address*

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

4 consecutive bytes form a *word* (32-bits)

Maximum addressable memory is 4 GB (gigabyte)



Memory is a large array

Storage locations are accessed using a 32-bit index, called the *address*

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

4 consecutive bytes form a *word* (32-bits)

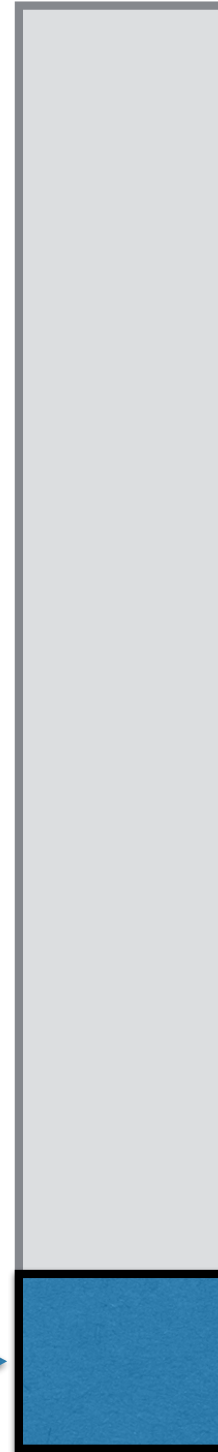
Maximum addressable memory is 4 GB (gigabyte)

512 MB Actual Memory →

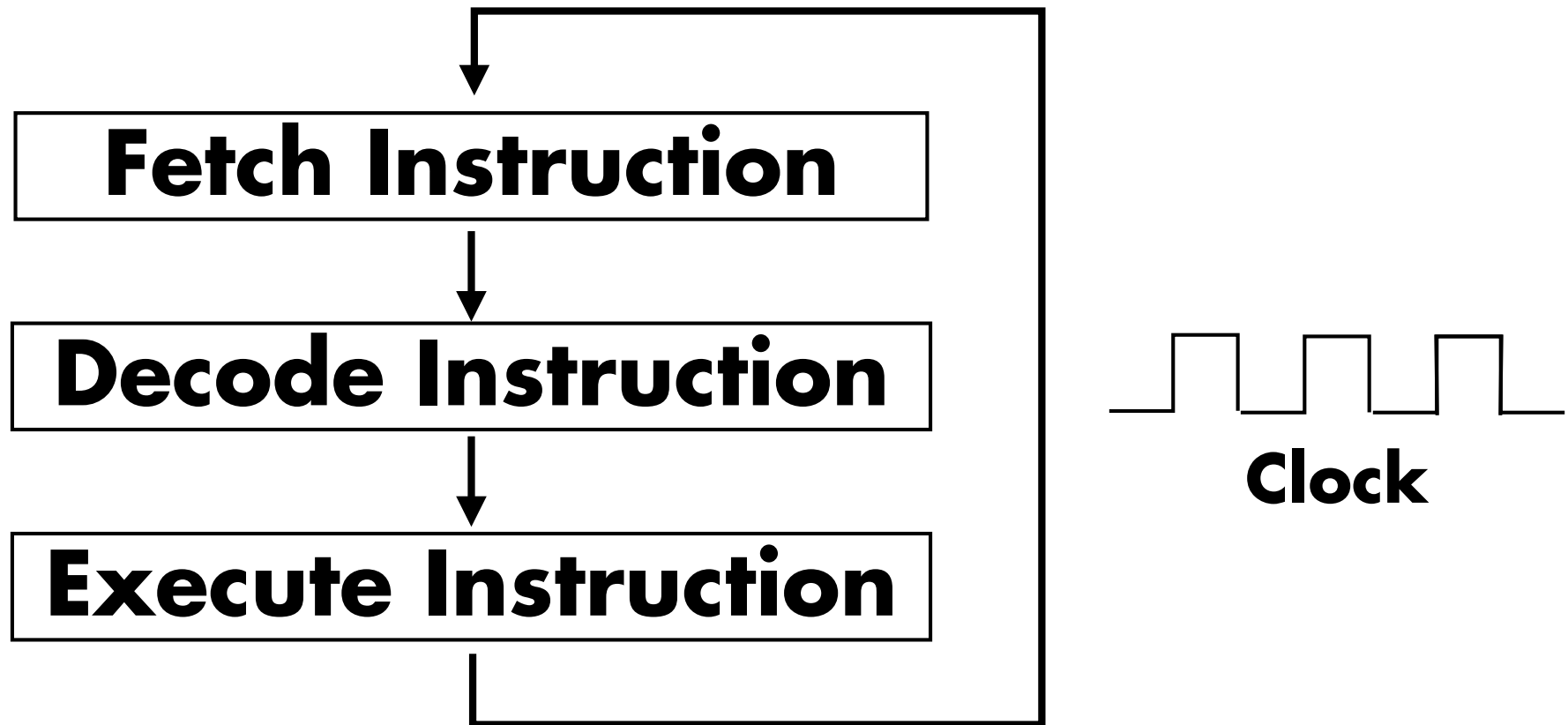
← 0x100000000

Memory Map

← 0x020000000

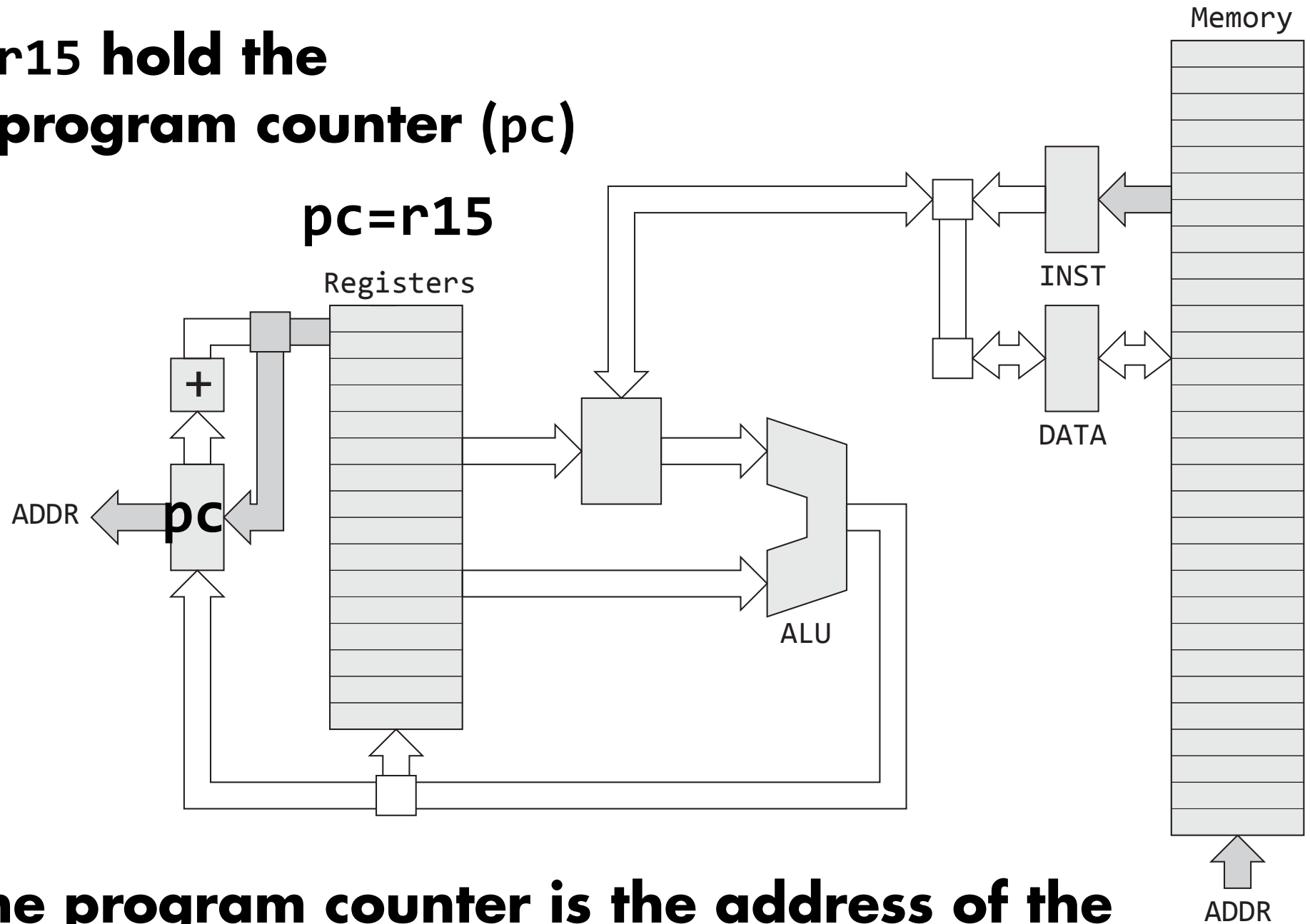


Running a Program



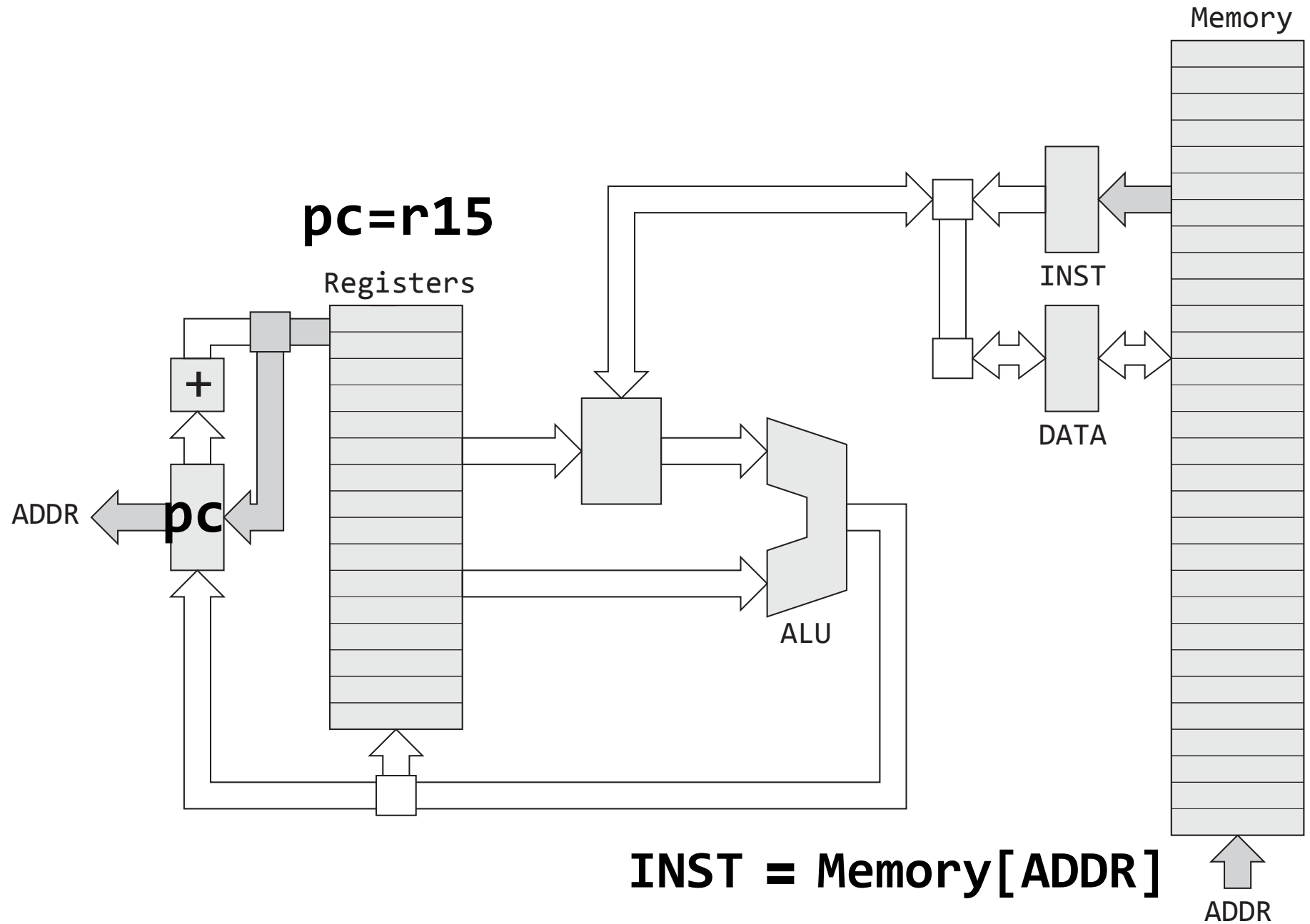
ARM Architecture / Floor Plan

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



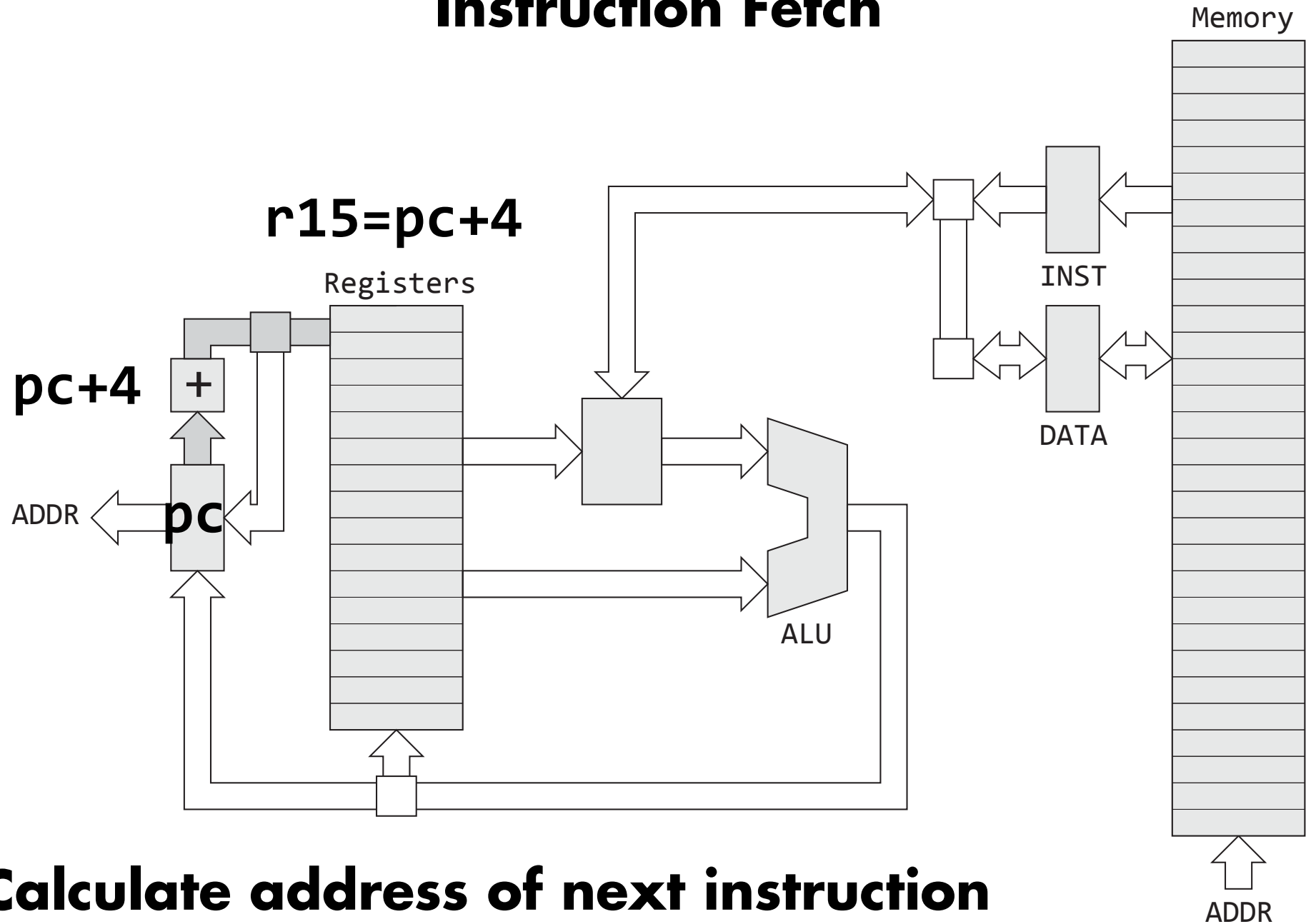
The program counter is the address of the next instruction to execute (not quite).

ARM Architecture / Floor Plan



Registers, addresses, and instructions are 32-bit words

Instruction Fetch



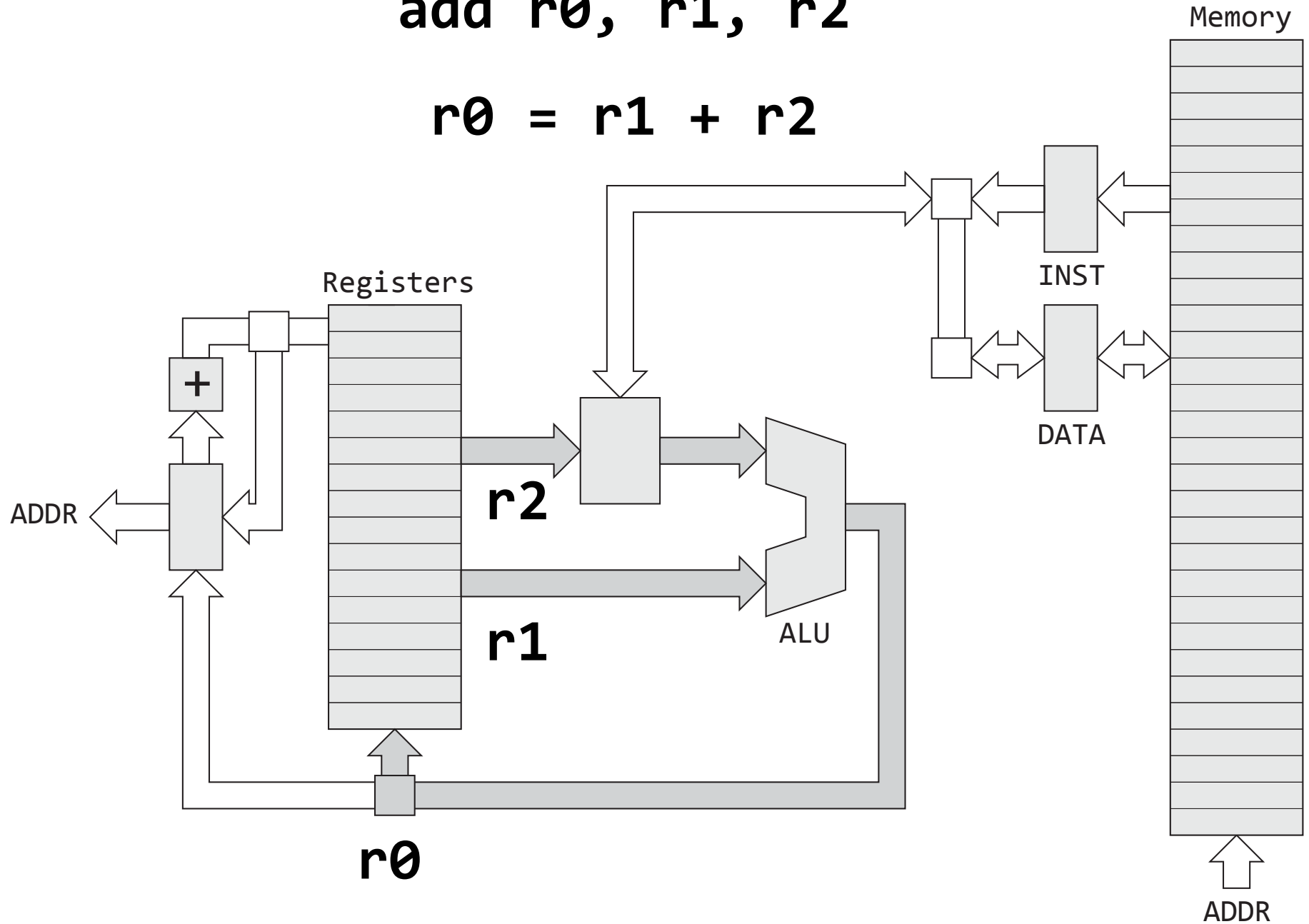
Calculate address of next instruction

Why pc+4?

Arithmetic-Logic Unit (ALU)

add r0, r1, r2

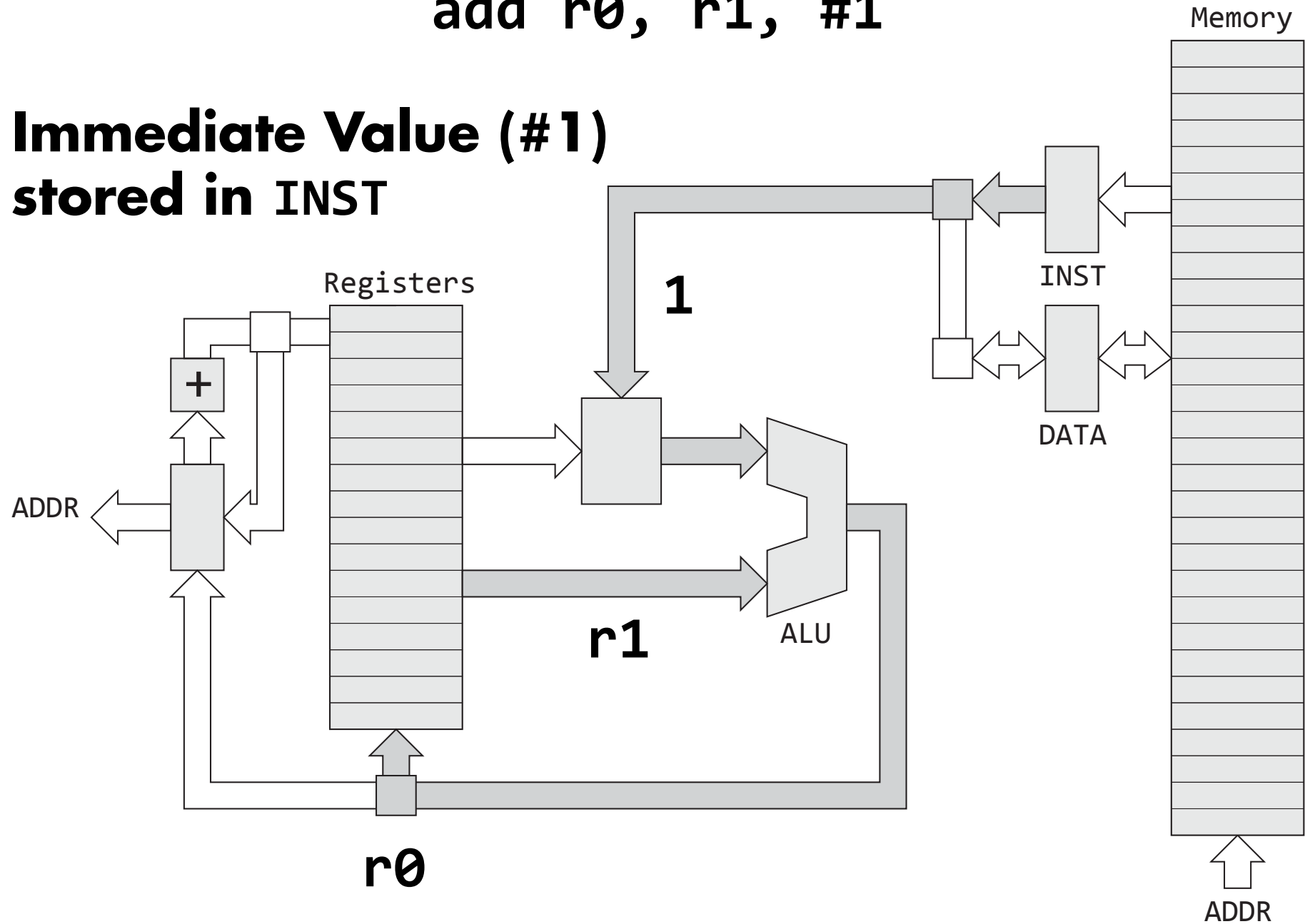
$r0 = r1 + r2$



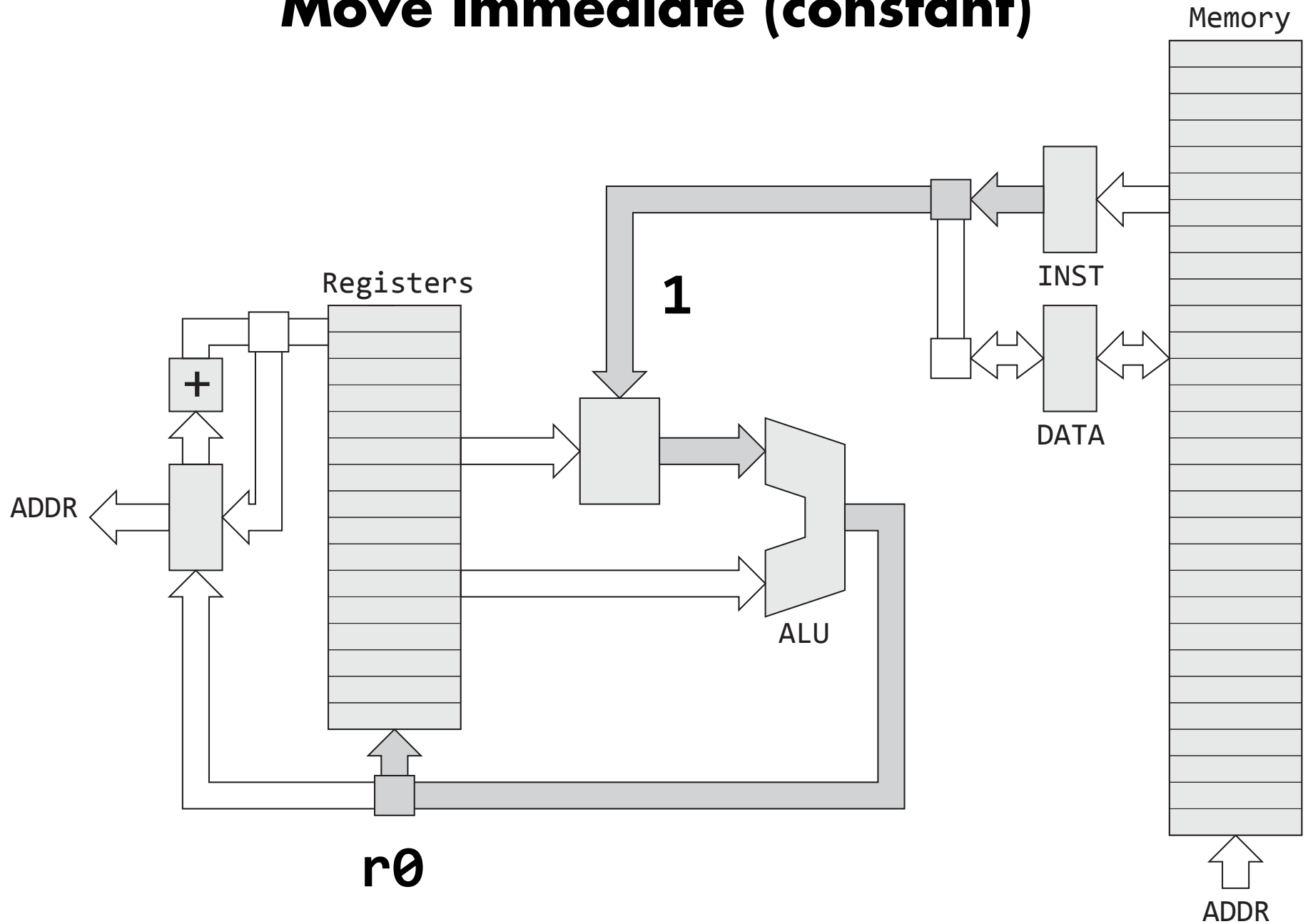
ALU only operates on data in registers

add r0, r1, #1

**Immediate Value (#1)
stored in INST**



Move Immediate (constant)



`mov r0, #1`

VisUAL

untitled.S - [Unsaved] - VisUAL

NewOpenSaveSettingsTools

Emulation Running

Line 3Issues 0

ExecuteResetStep BackwardsStep Forwards

Reset to continue editing code

1234

movr0, #1

movr1, #2

addr2, 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: 1Total: 3

CSPR Status Bits (NZCV)

0

0

0

0

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)

`02 00 81 e0`

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

# Extract instructions into a 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

# Display binary contents as bytes in hex
% xxd -g 1 add.bin
00000000: 02 00 81 e0
```

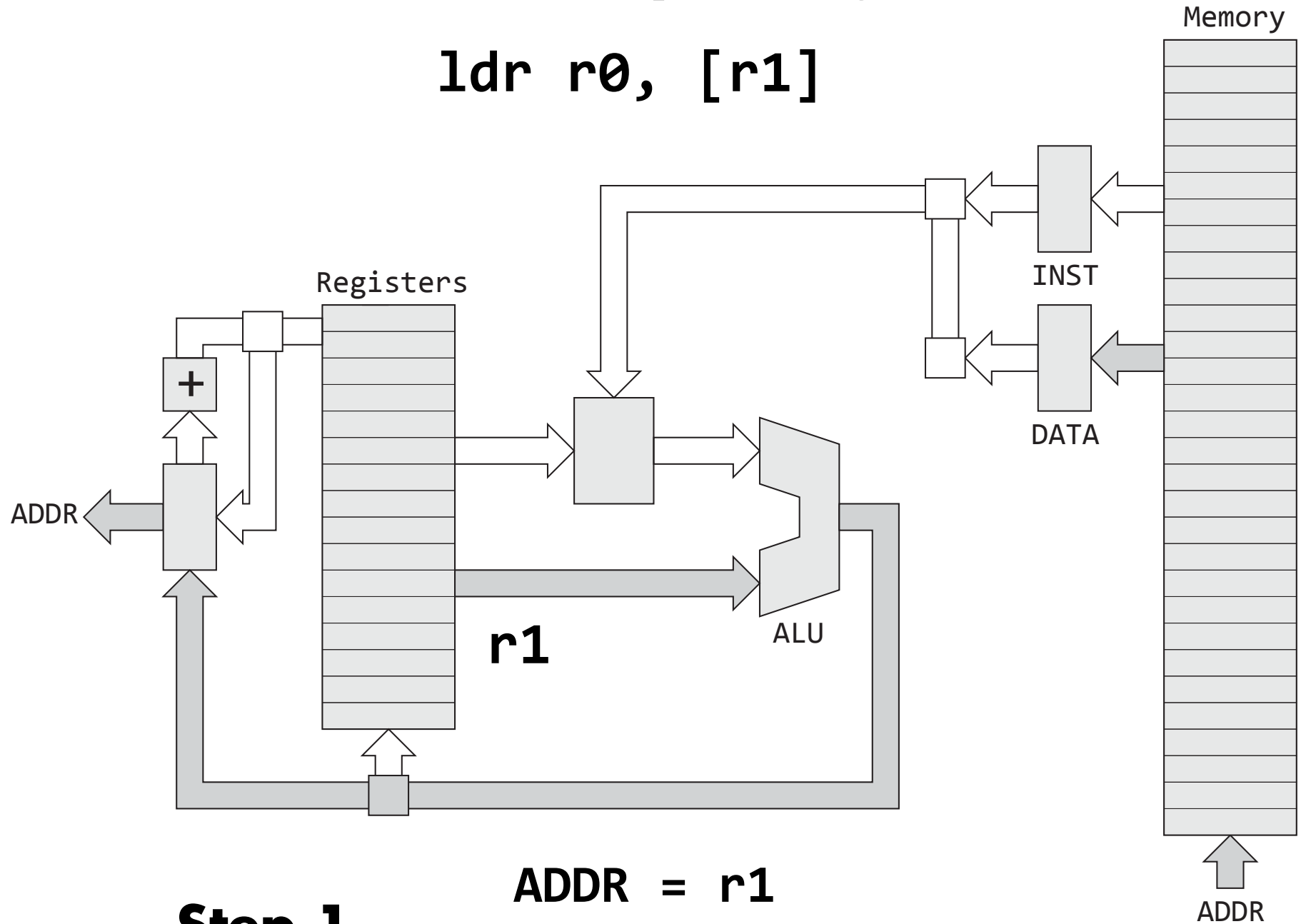
Conceptual Questions

- 1. Suppose your program starts at 0x8000, what assembly language instruction could you execute to jump to and start executing instructions at that location.**
- 2. If all instructions are 32-bits, can you move any 32-bit constant value into a register using a single `mov` instruction?**
- 3. What is the difference between a memory location and a register?**

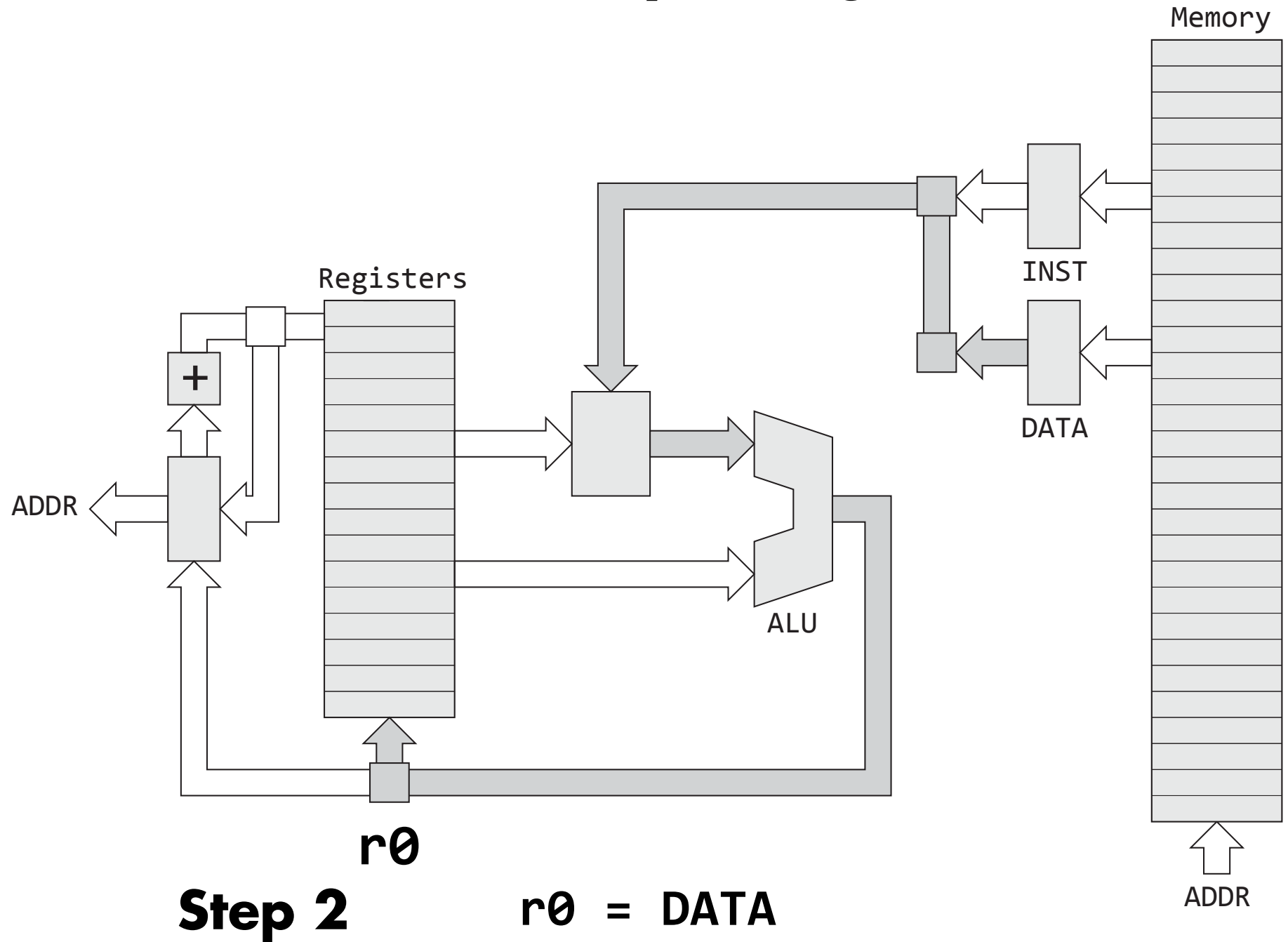
Load and Store Instructions

Load from Memory to Register (LDR)

ldr r0, [r1]

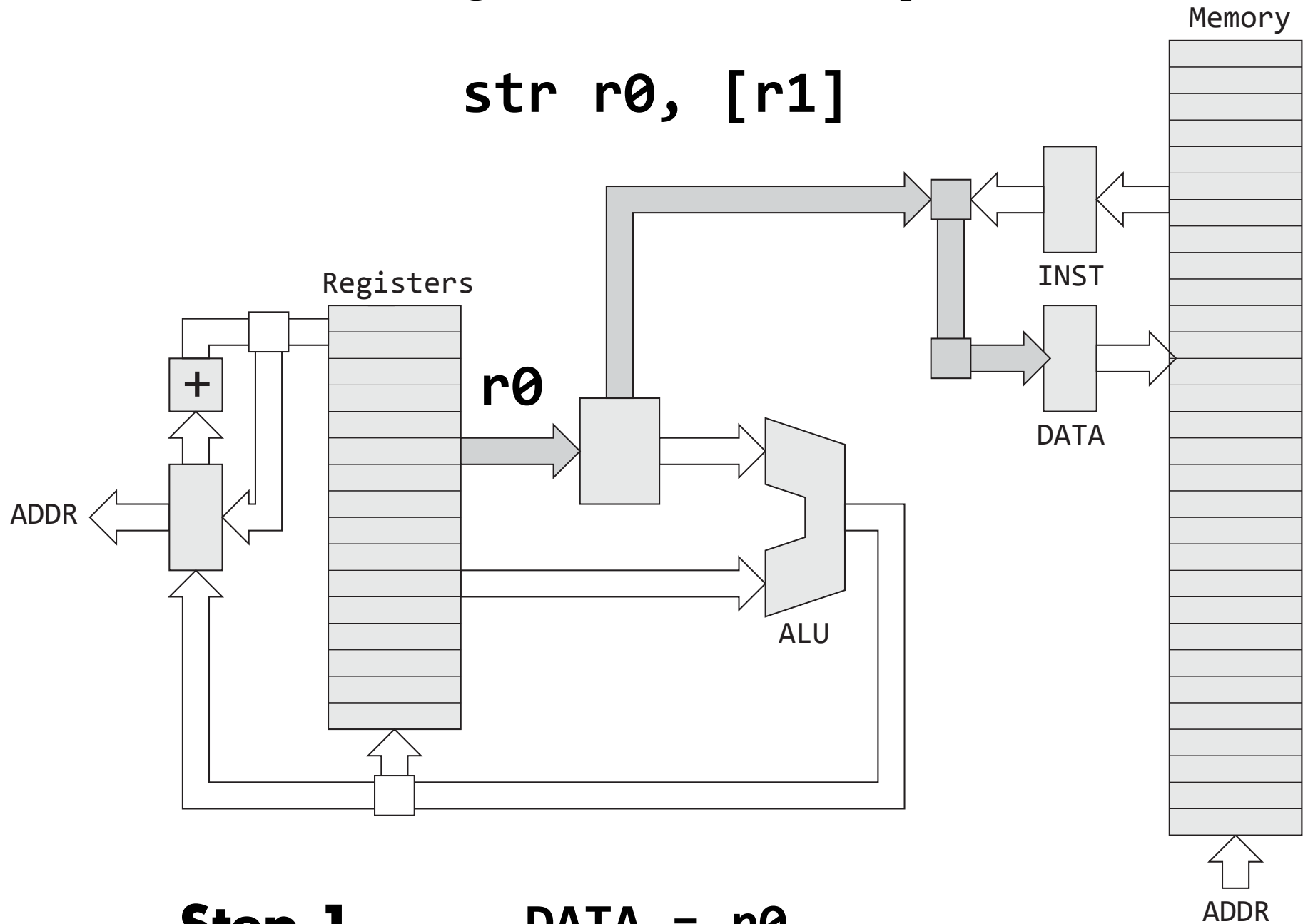


Load from Memory to Register (LDR)



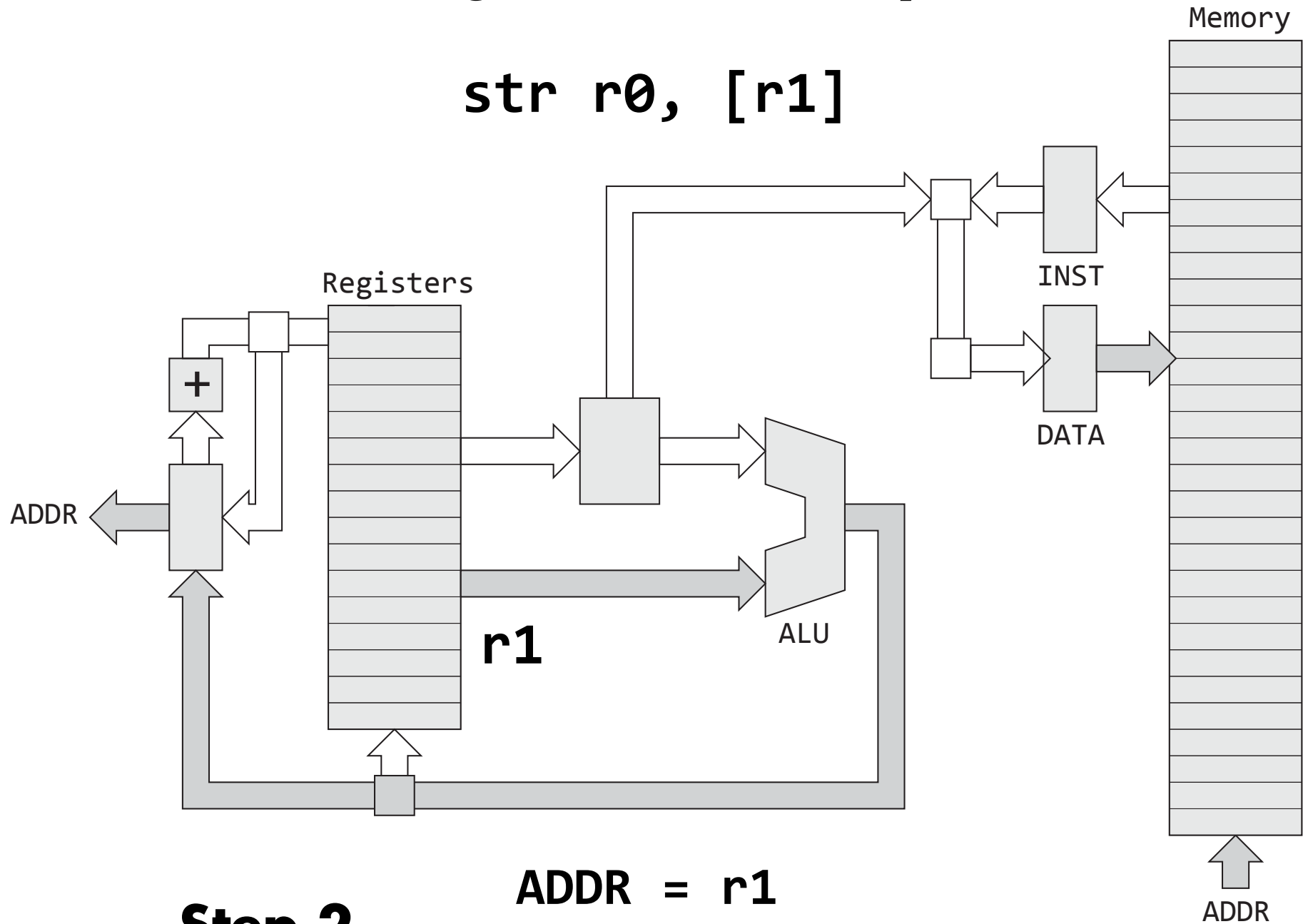
Store Register in Memory (STR)

`str r0, [r1]`



Store Register in Memory (STR)

str r0, [r1]



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)

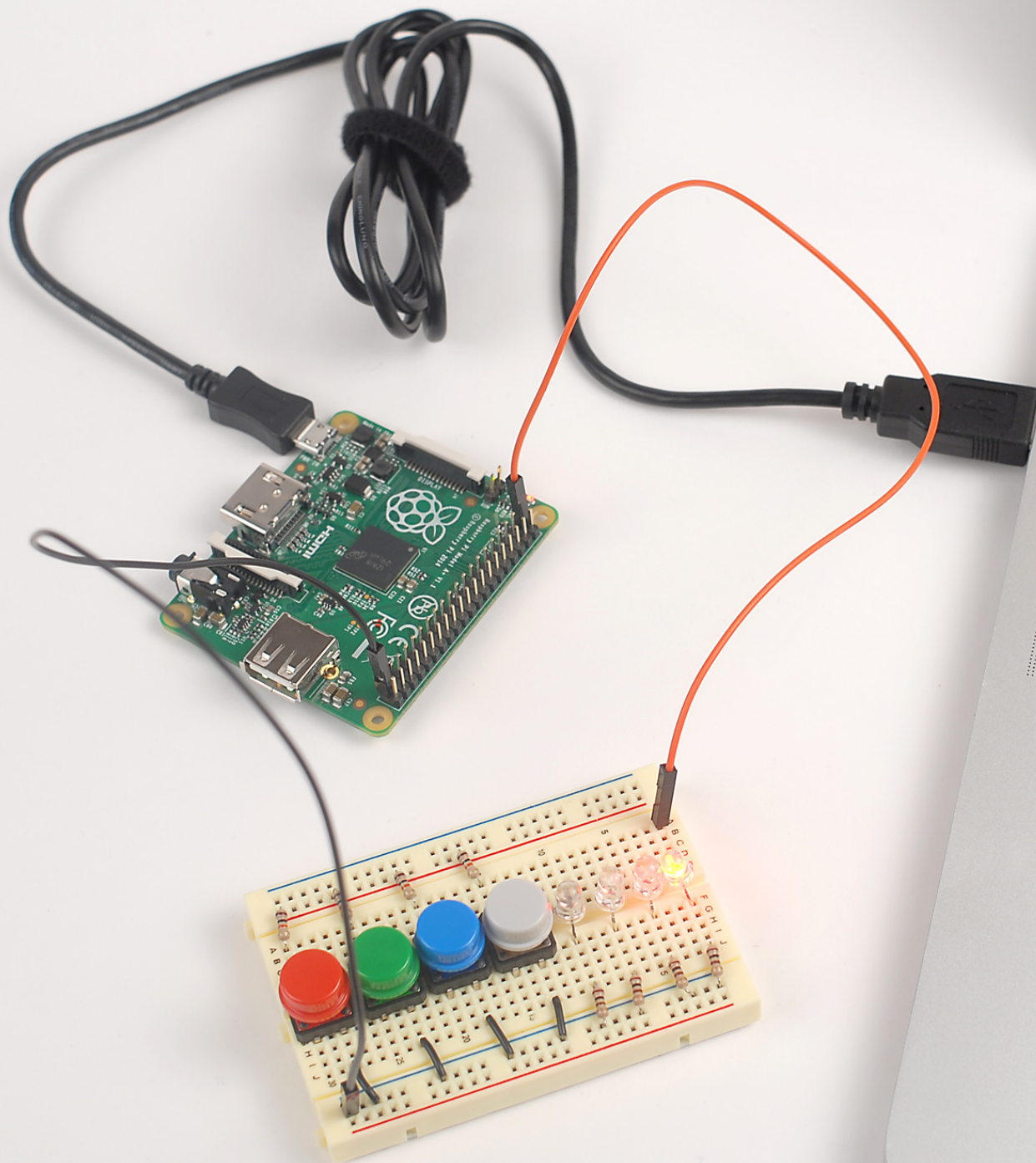
0

0

0

0

Turning on an LED



**Computers have Peripherals
that Interface to the World**

GPIO Pins are Peripherals

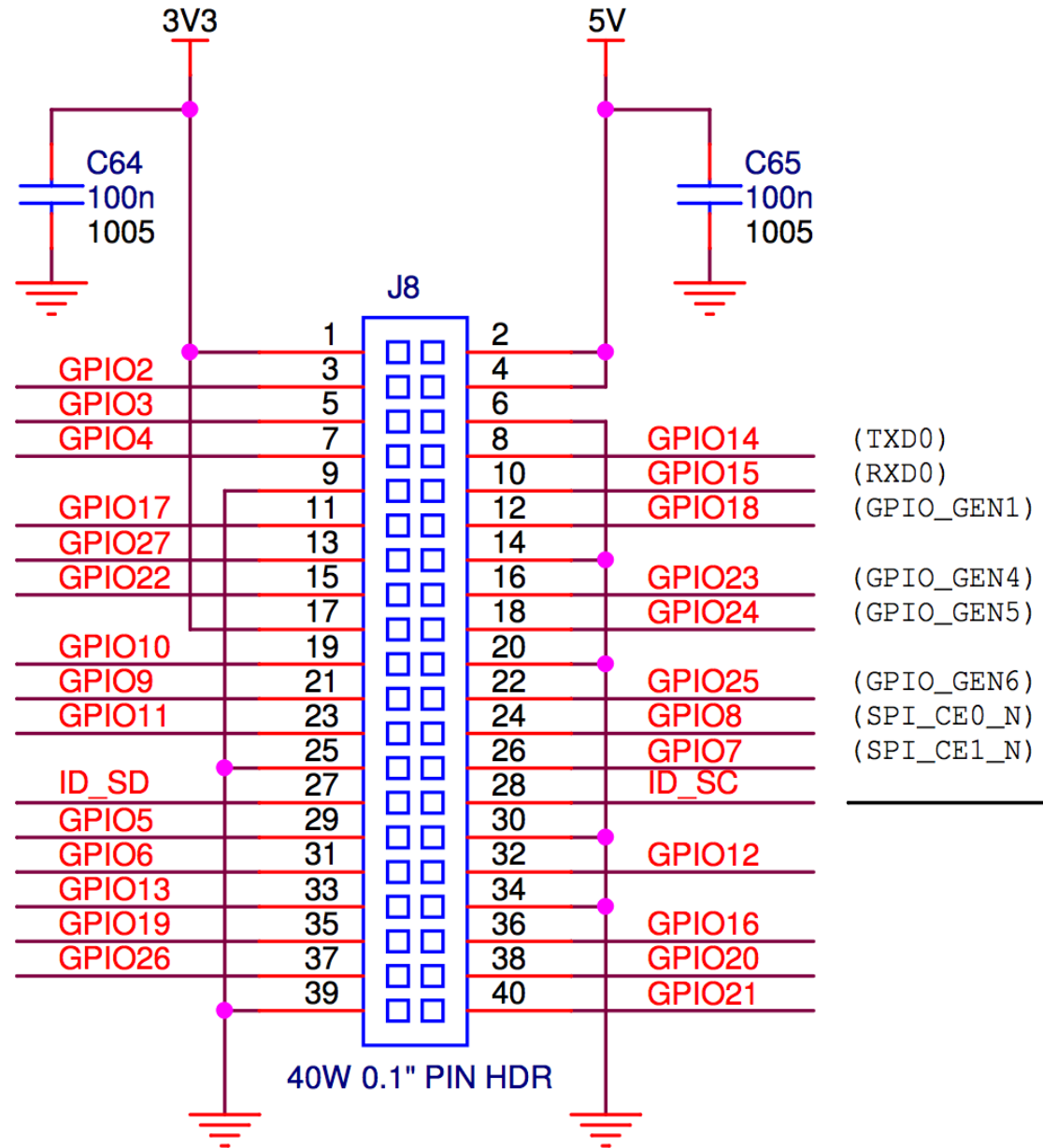
General-Purpose Input/Output (GPIO) Pins



(SDA1)
(SCL1)
(GPIO_GCLK)

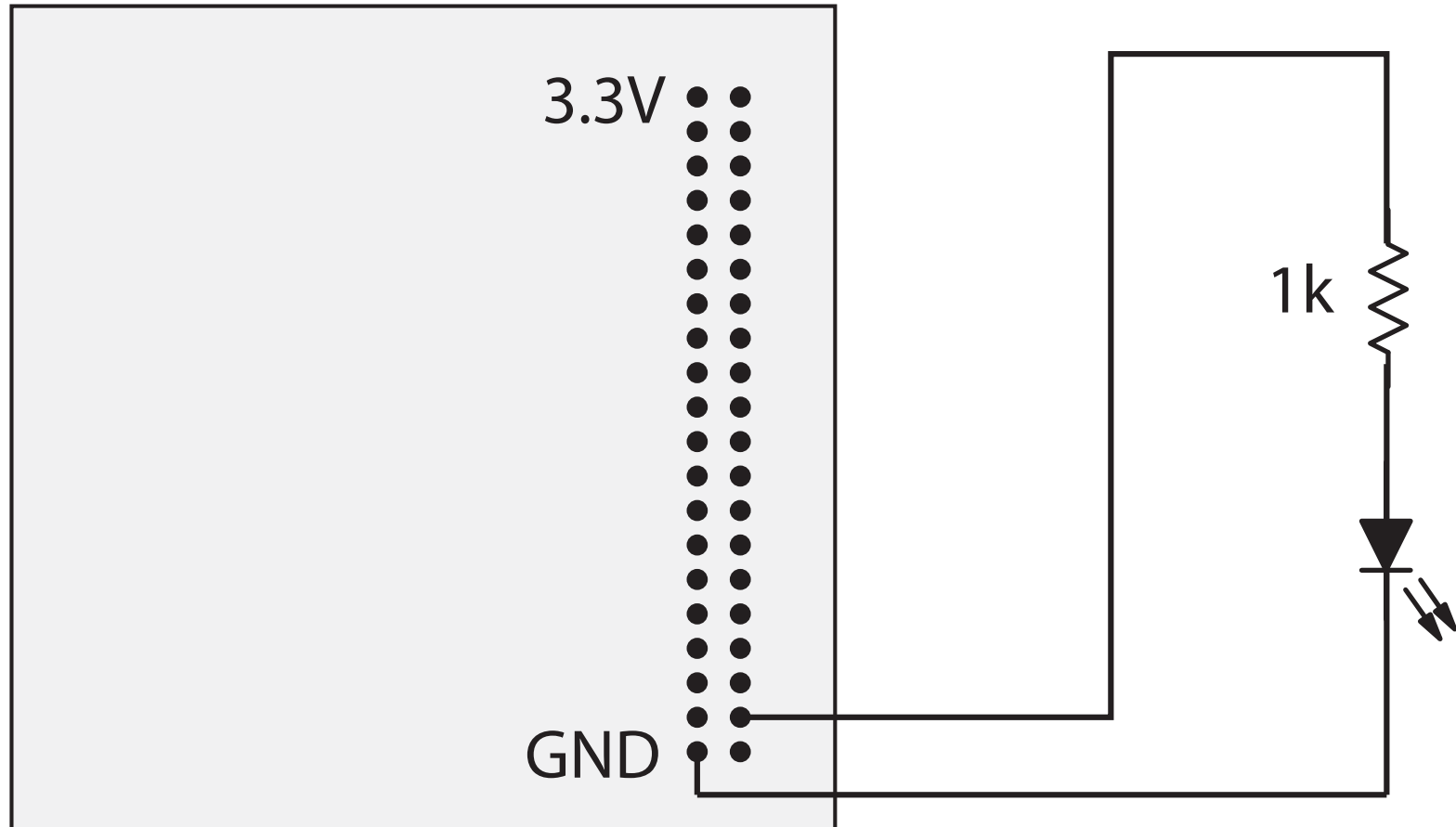
(GPIO_GEN0)
(GPIO_GEN2)
(GPIO_GEN3)

(SPI_MOSI)
(SPI_MISO)
(SPI_SCLK)



54 GPIO Pins

Connect LED to GPIO 20



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

**Peripherals are Controlled
by
Special Memory Locations**

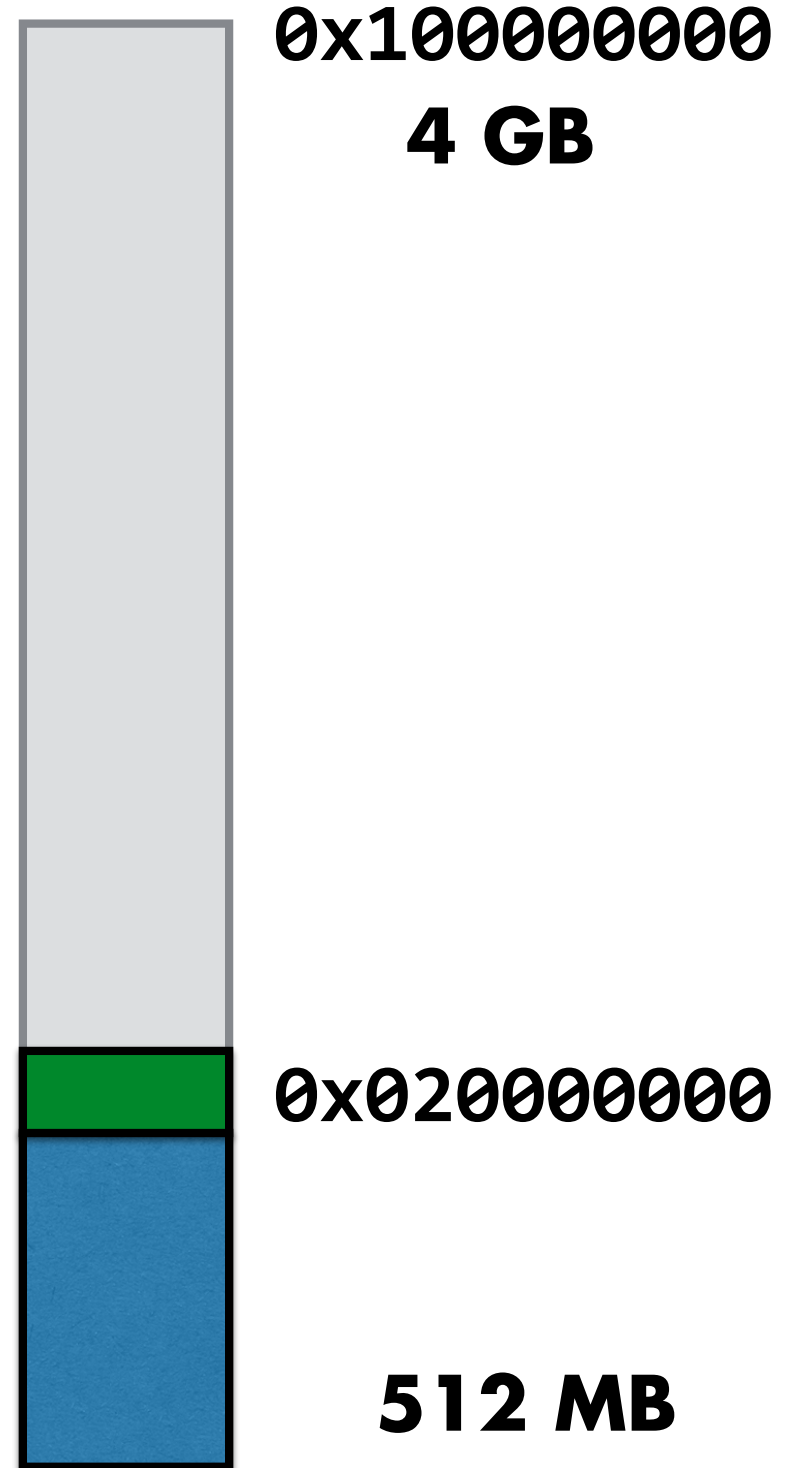
"Peripheral Registers"

Memory Map

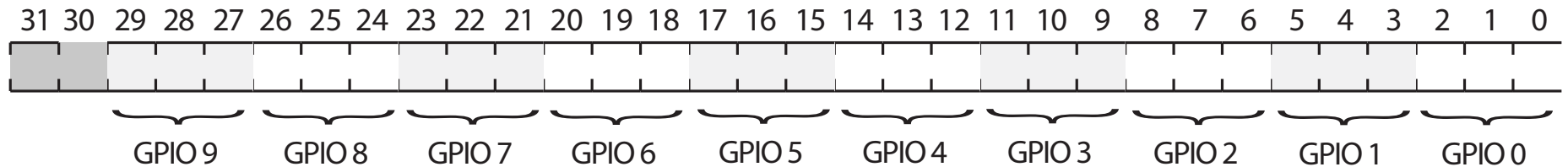
Peripheral registers
are *mapped*
into address space

Memory-Mapped IO
(MMIO)

MMIO space is above
physical memory



GPIO Function Select Register

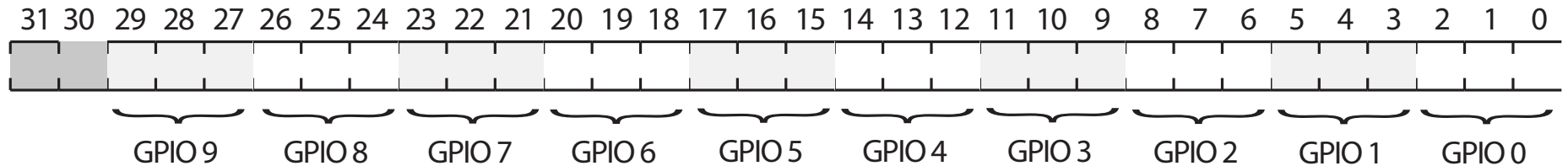


"Function" is INPUT, OUTPUT (or ALT0-5)

8 functions requires 3 bits to specify

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

GPIO Function Select Register



"Function" is INPUT, OUTPUT (or ALT0-5)

8 functions requires 3 bits to specify

10 pins times 3 bits = 30 bits

32-bit register (2 wasted bits)

Pi has 54 GPIOs - 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 ...

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

Ref: BCM2835-ARM-Peripherals.pdf

// Set GPIO20 to be an output

// FSEL2 = 0x20200008

mov r0, #0x20 // #0x00000020

lsl r1, r0, #24 // #0x20000000

lsl r2, r0, #16 // #0x00200000

orr r0, r1, r2 // #0x20200000

orr r0, r0, #0x08 // #0x20200008

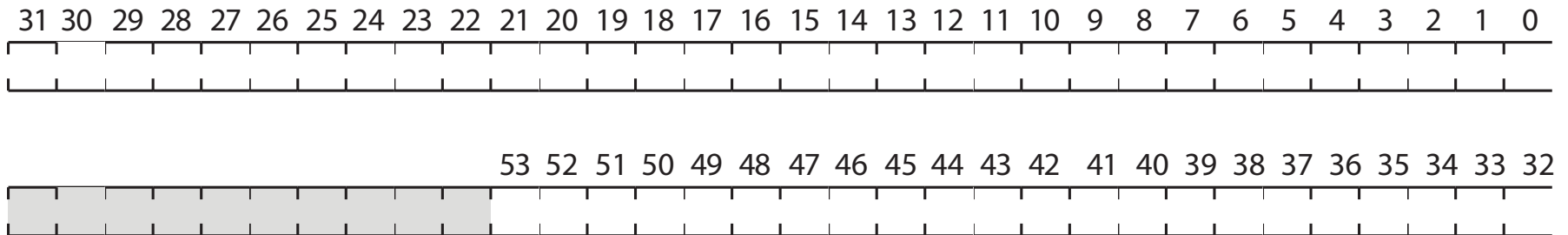
mov r1, #1 // 1 indicates OUTPUT

str r1, [r0] // store 1 to 0x20200008

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**

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 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

```
// Set GPIO20 output High (3.3V)
```

```
// FSET0 = 0x2020001c
```

```
mov r0, #0x20
```

```
lsl r1, r0, #24
```

```
lsl r2, r0, #16
```

```
orr r1, r1, r2
```

```
orr r1, r1, #0x08
```

```
mov r1, #1           // 0x00000001
```

```
lsl r1, r1, #20      // 0x00100000
```

```
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/

ON Macintosh HD

Copy to SD card

% cp on.bin /Volumes/ON/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!!  
#
```



Key Concepts

Bits are bits; bitwise operations

Memory addresses refer to bytes (8-bits), words are 4 bytes

Memory stores both instructions and data

Computer:s repeatedly fetch, decode, and execute instructions

Different types of ARM instructions

- **ALU**

- **Loads and Stores**

- **Branches**

General purpose IO (GPIO), peripheral registers, and MMIO