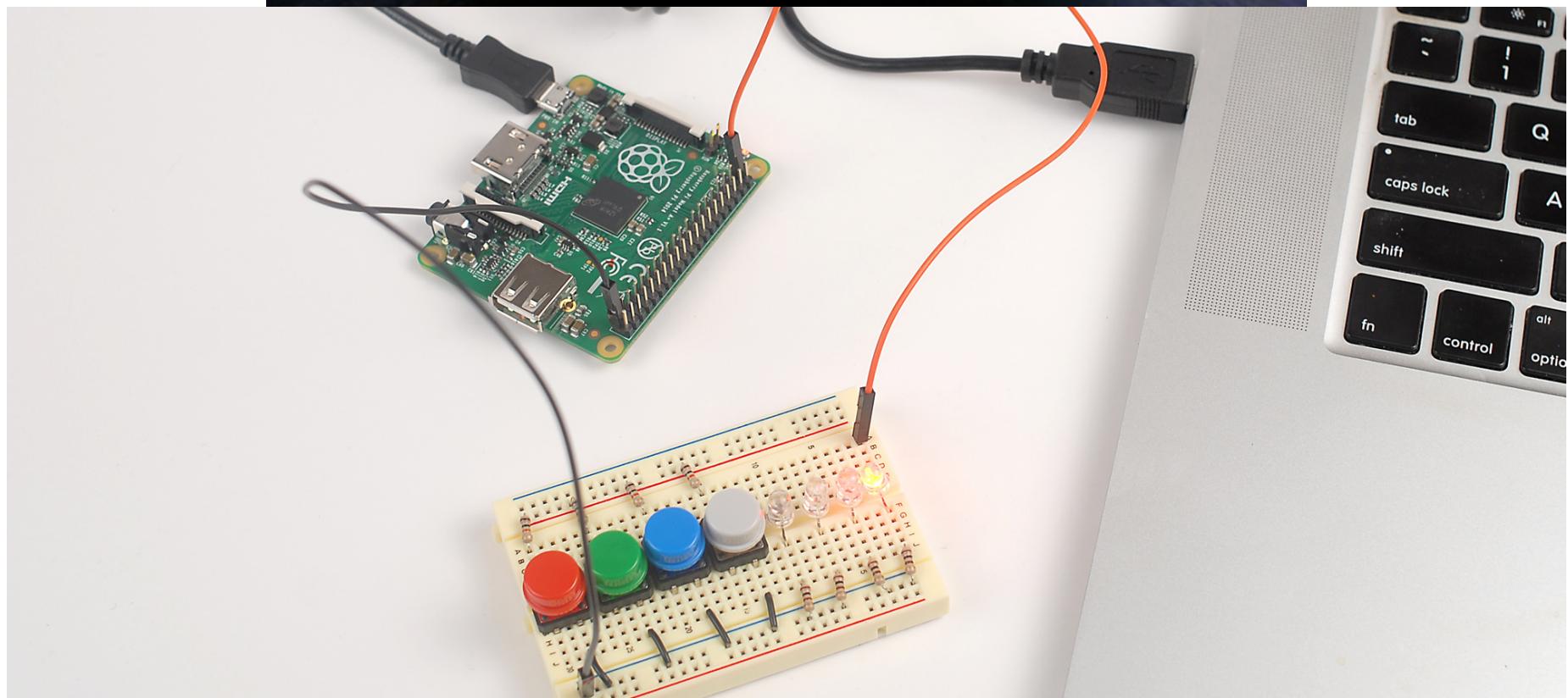
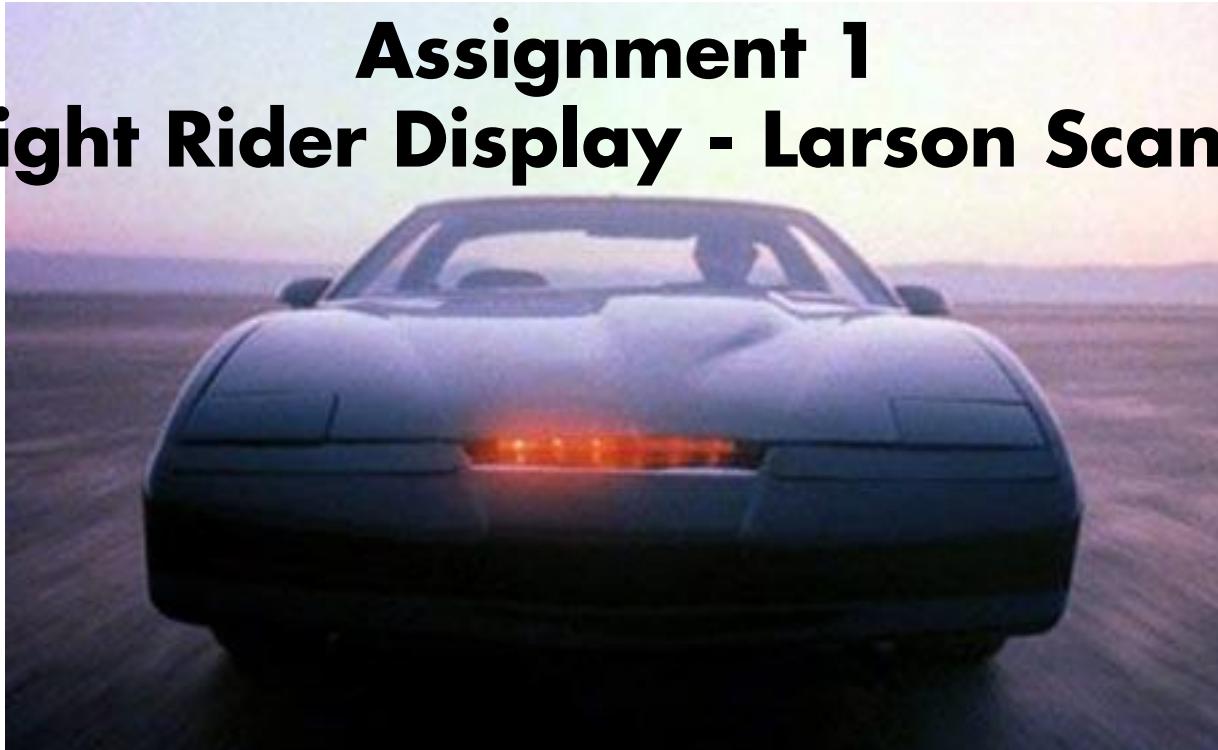


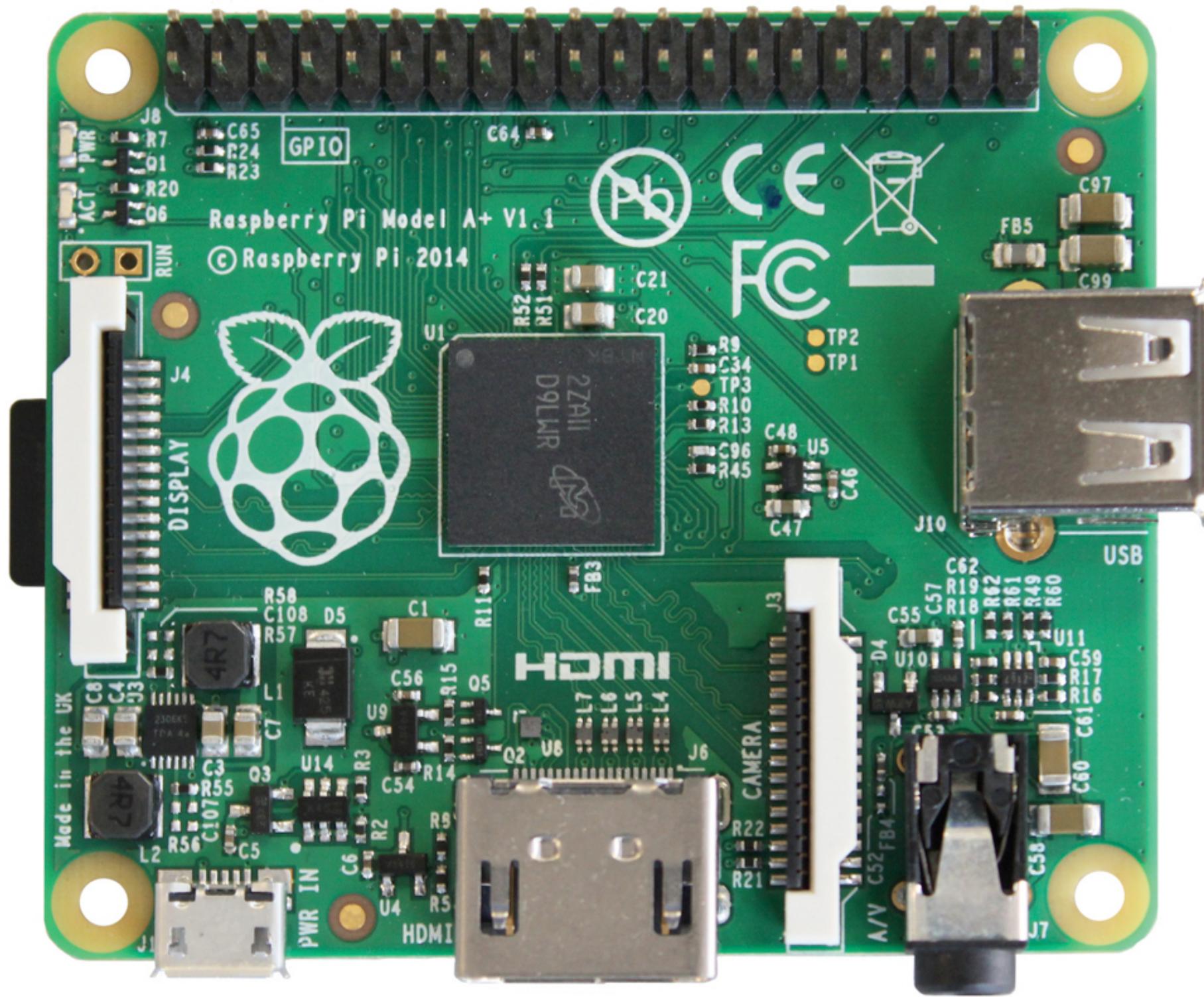
ARM Processor and Memory Architecture

Goal: Turn on an LED

Assignment 1

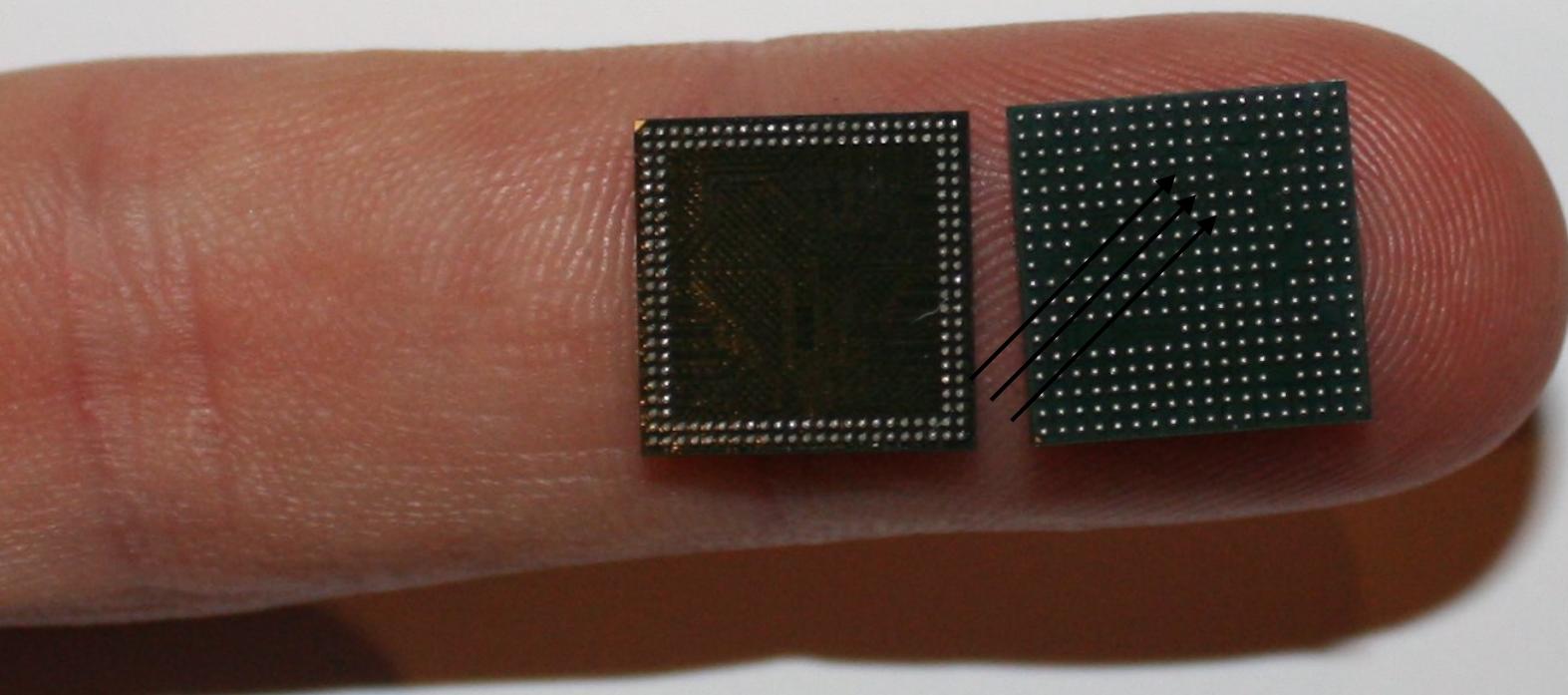
Knight Rider Display - Larson Scanner





Package on Package

Broadcom 2865 ARM Processor



Samsung 4Gb (gigabit) SDRAM

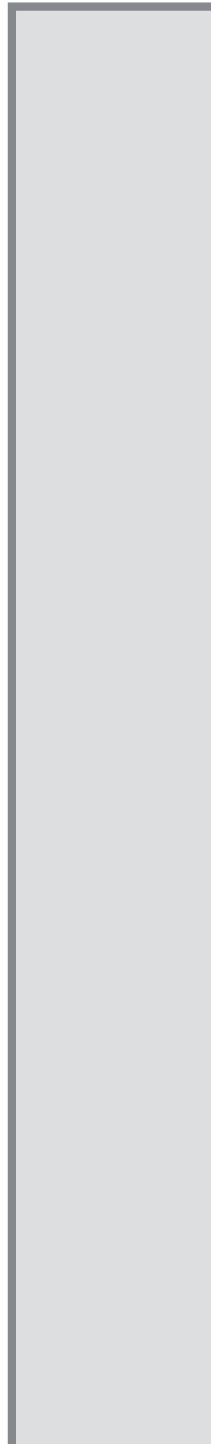
Memory is a large array

Storage locations are accessed using 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 (gigabytes)

Memory used to store both instructions and data



10000000₁₆

Memory Map

00000000₁₆

$$2^{10} = 1024 = 1 \text{ KB}$$

$$2^{20} = 1 \text{ MB}$$

$$2^{30} = 1 \text{ GB}$$

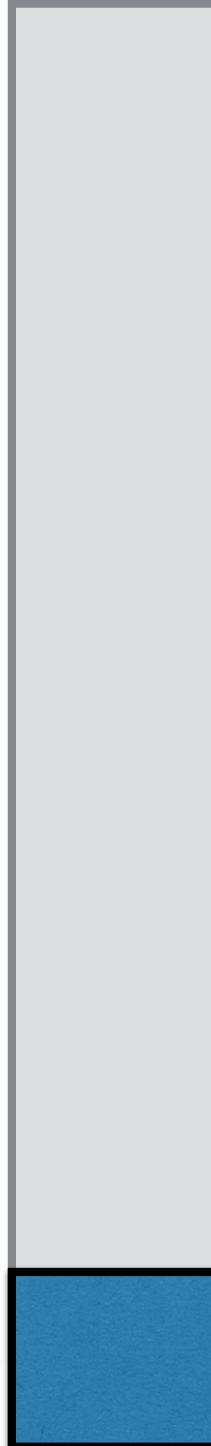
$$2^{32} = 4 \text{ GB}$$

100000000_16

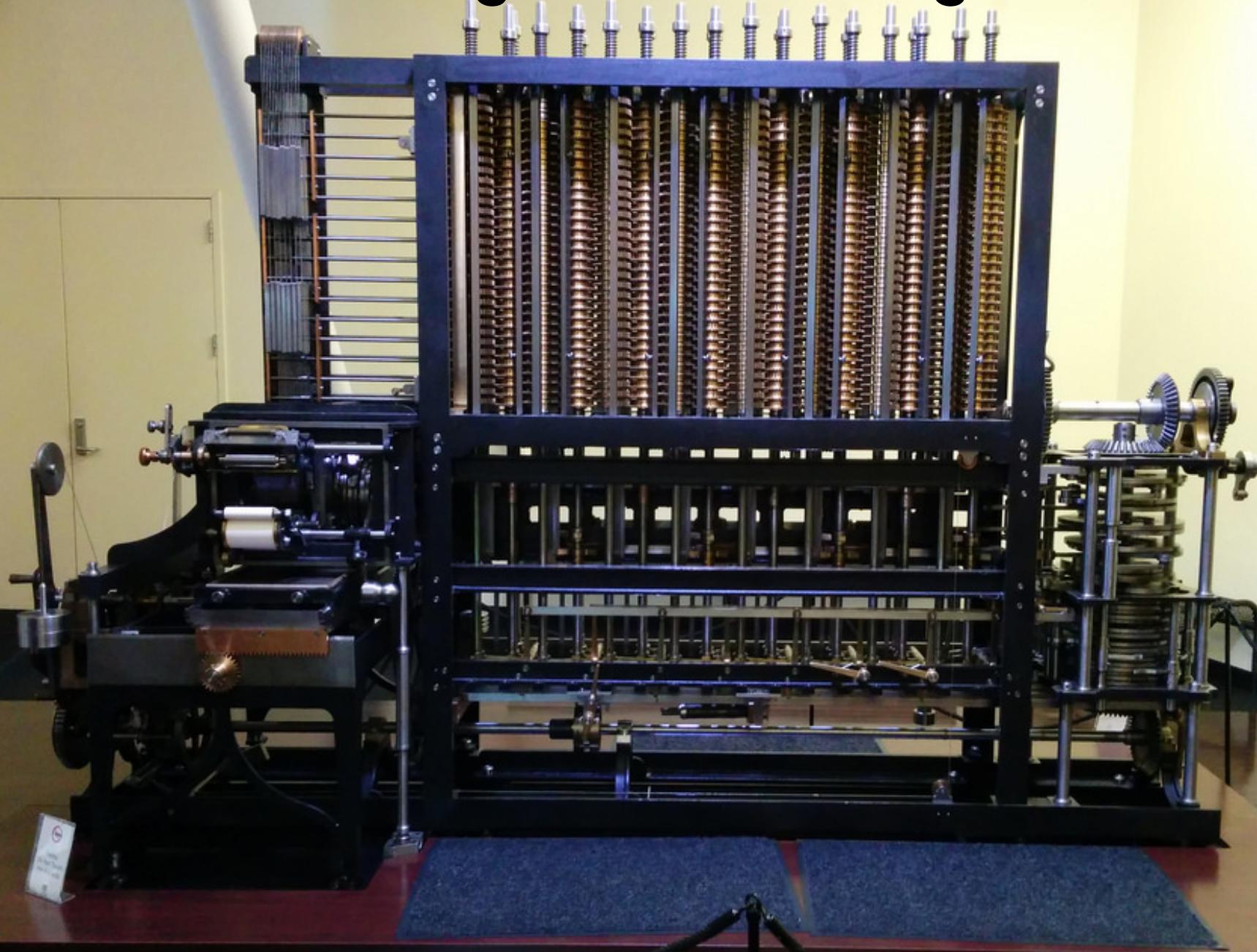
Memory Map

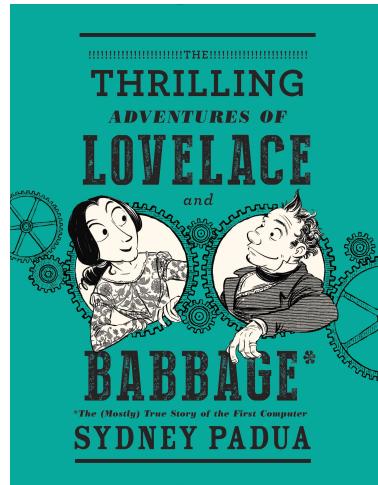
02000000_16

512 MB Actual Memory 

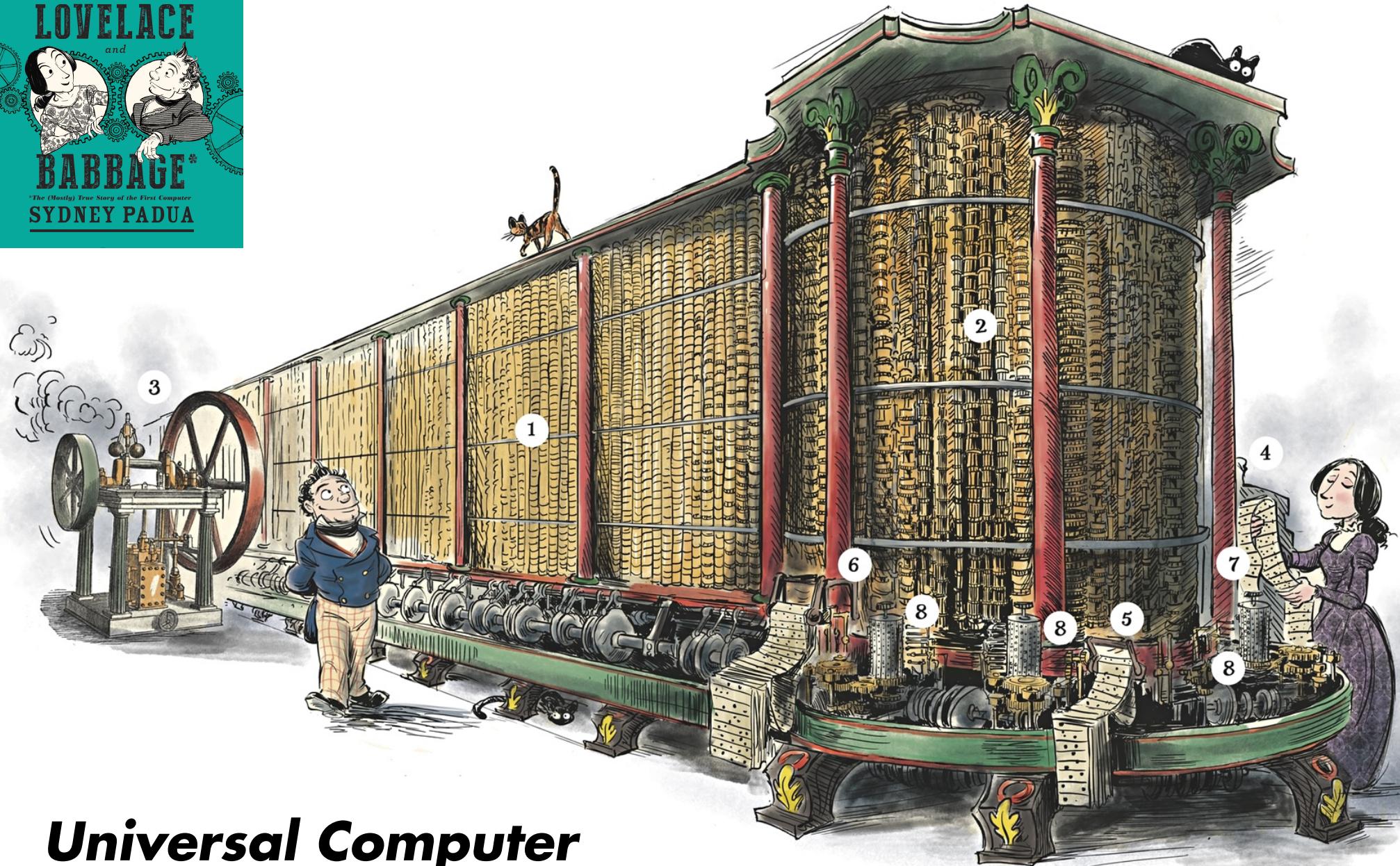


Babbage Difference Engine



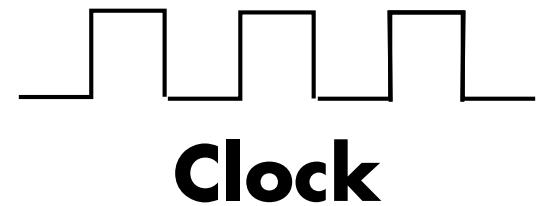
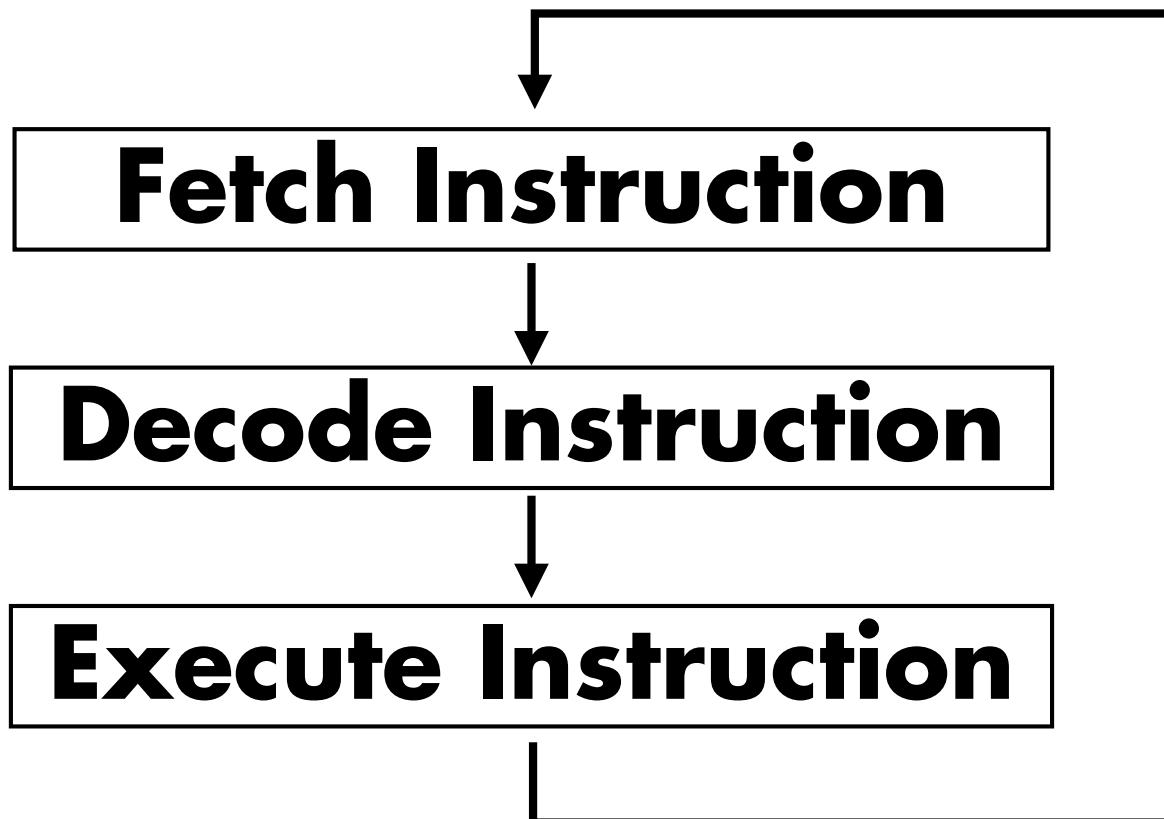


Analytical Engine

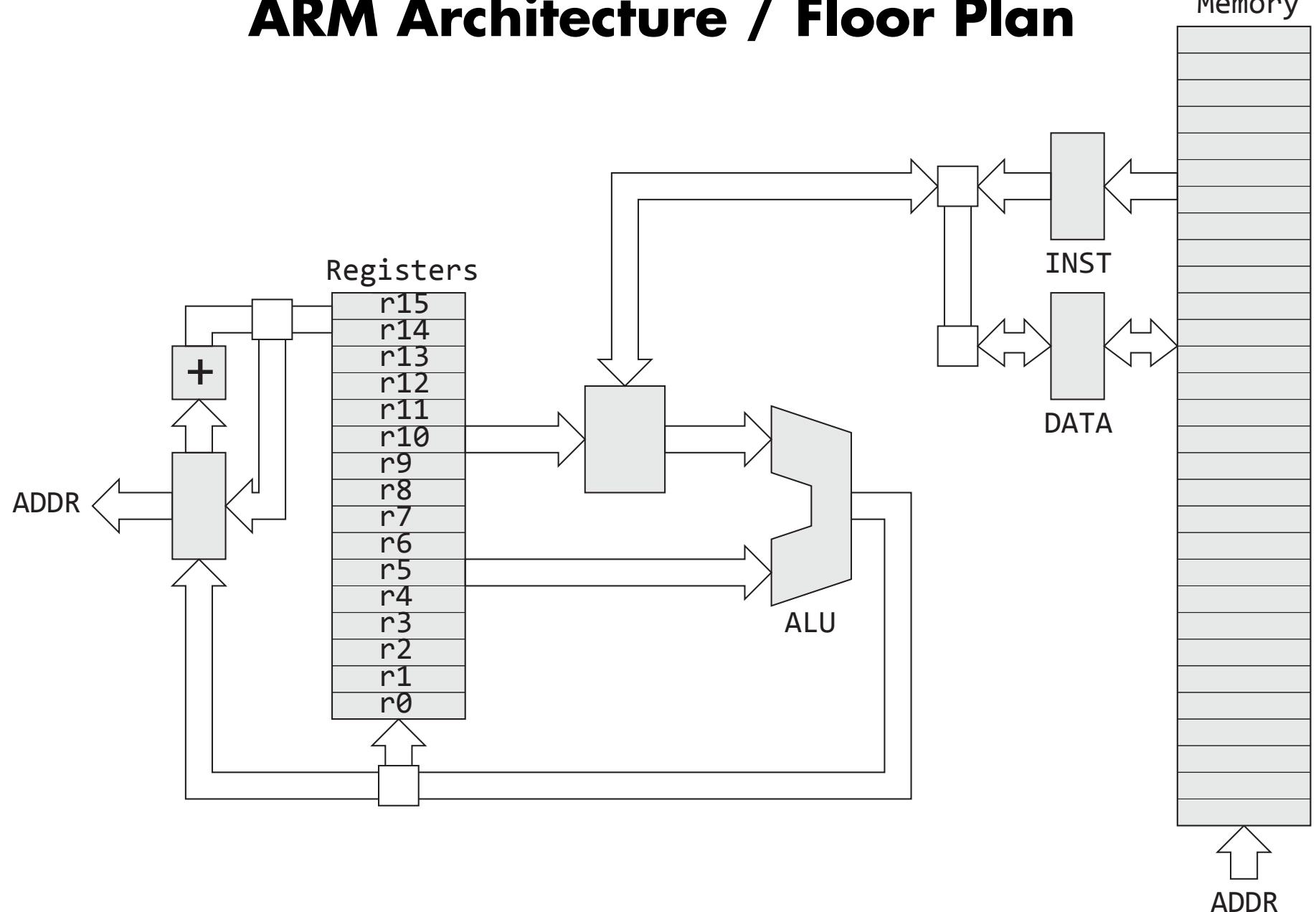


Universal Computer

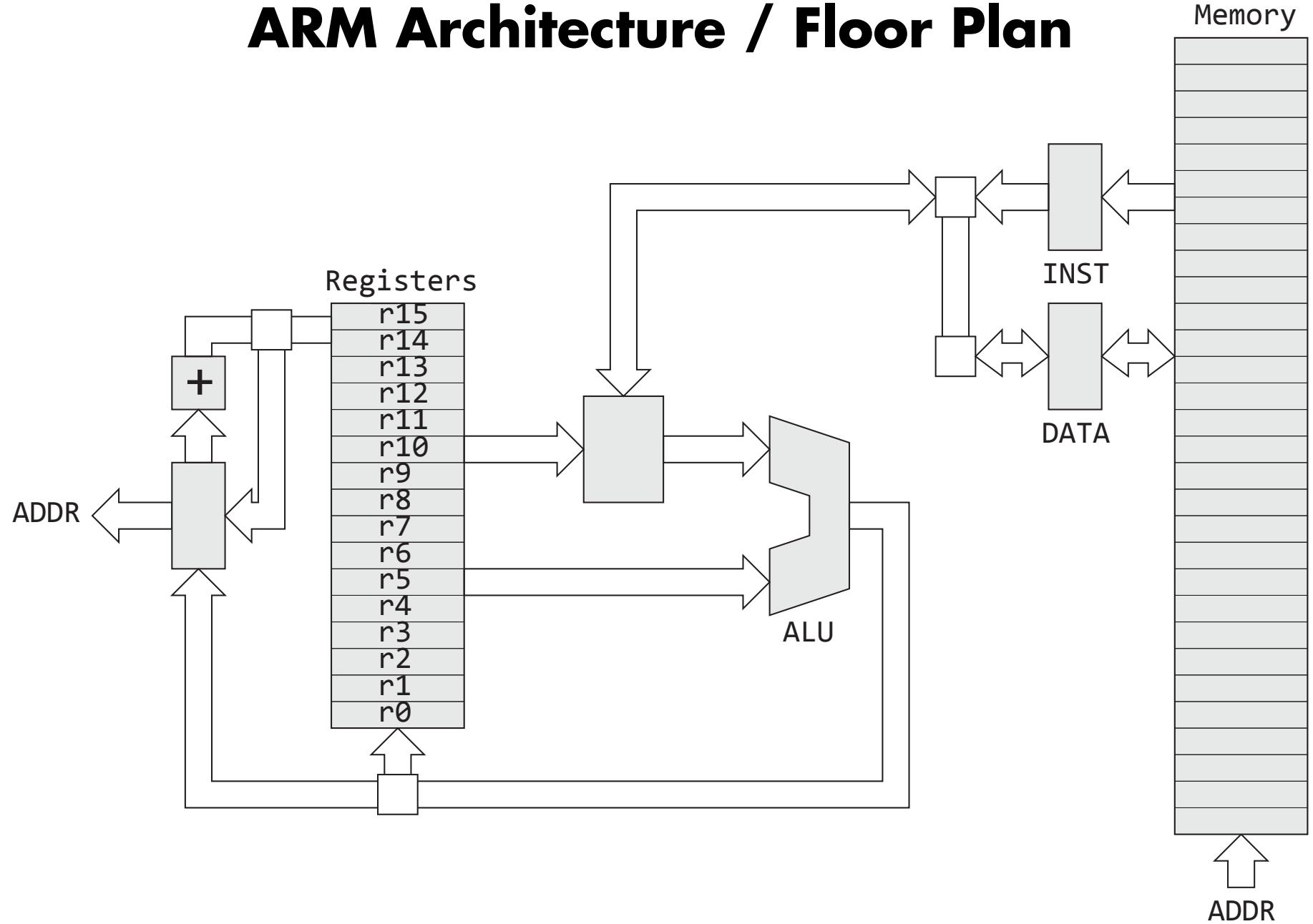
Running a Program



ARM Architecture / Floor Plan



ARM Architecture / Floor Plan

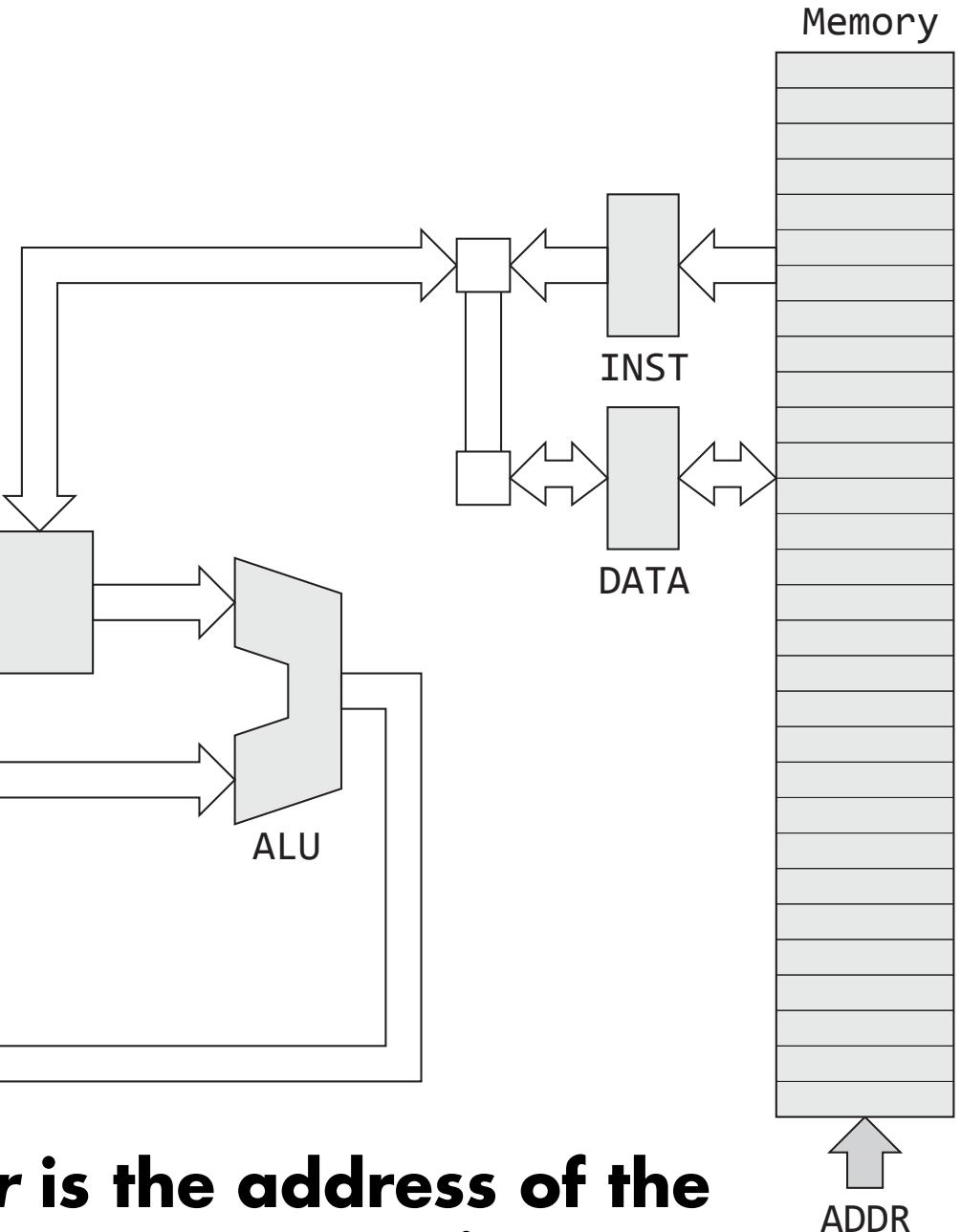
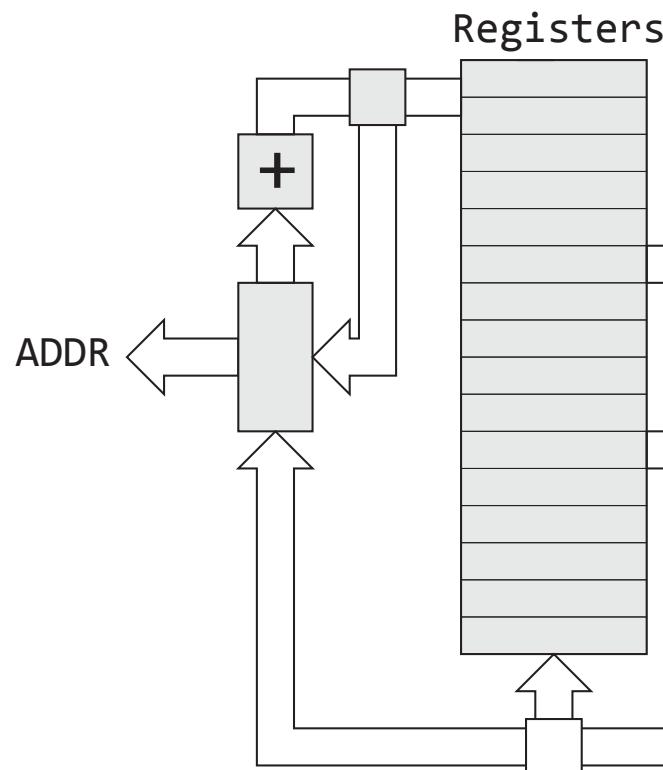


Key Architectural Fact: Each box is a 32-bit word

ARM Architecture / Floor Plan

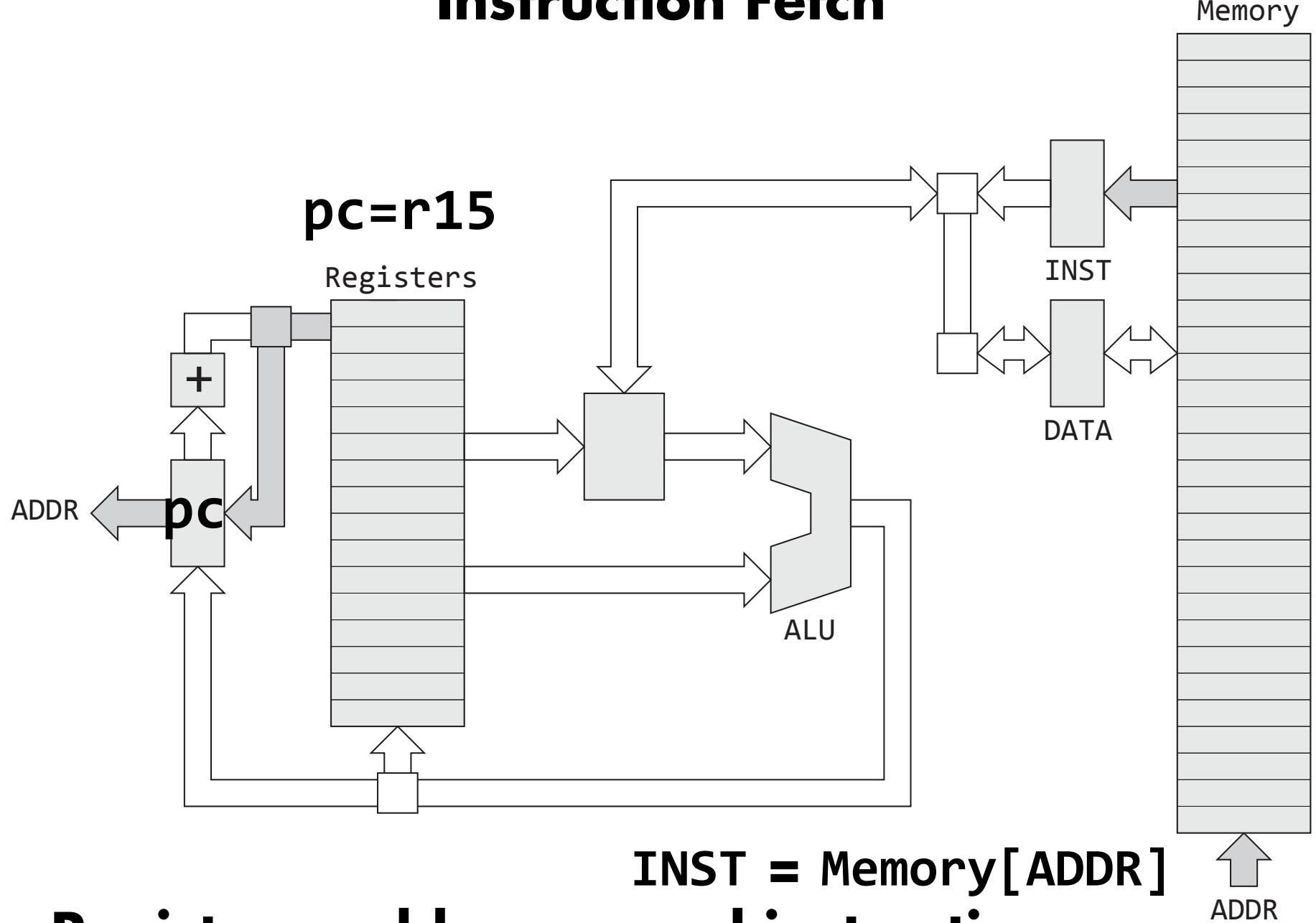
r15 holds the program counter (pc)

$$pc=r15$$



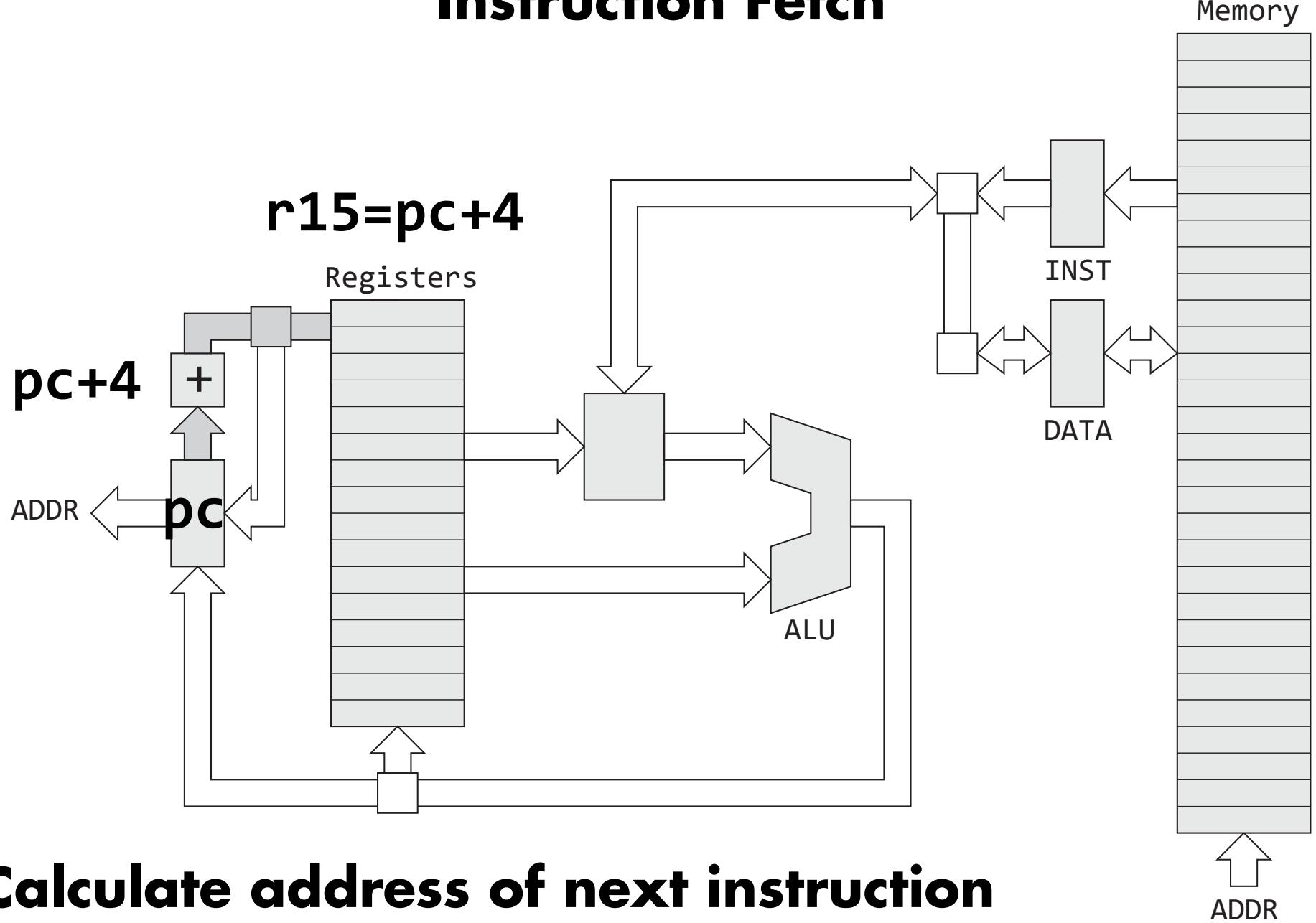
The **program counter** is the address of the next instruction to execute (not quite)

Instruction Fetch



**Registers, addresses and instructions
are 32-bit words**

Instruction Fetch



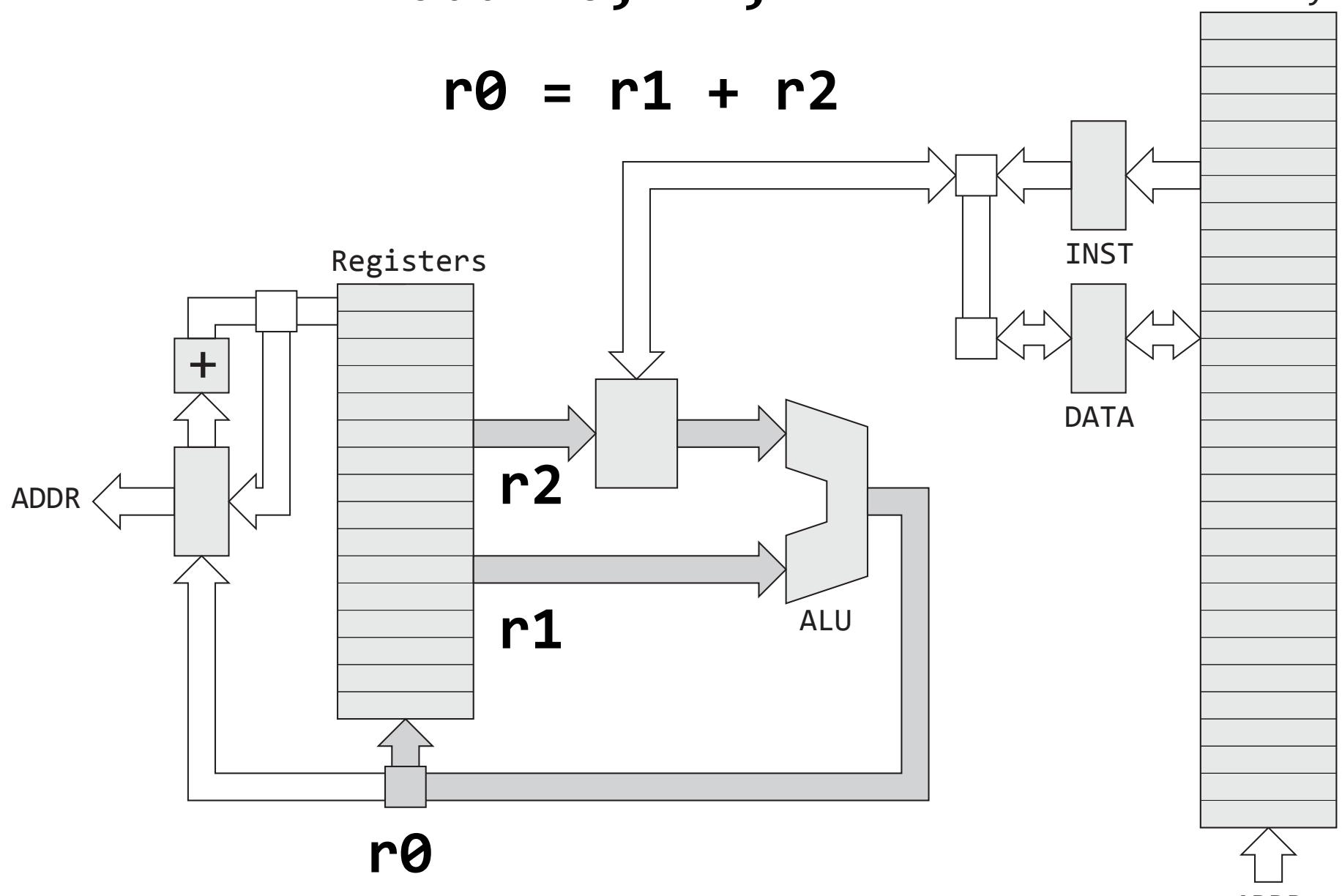
Calculate address of next instruction

Why $pc + 4$?

Arithmetic-Logic Unit (ALU)

add r0, r1, r2

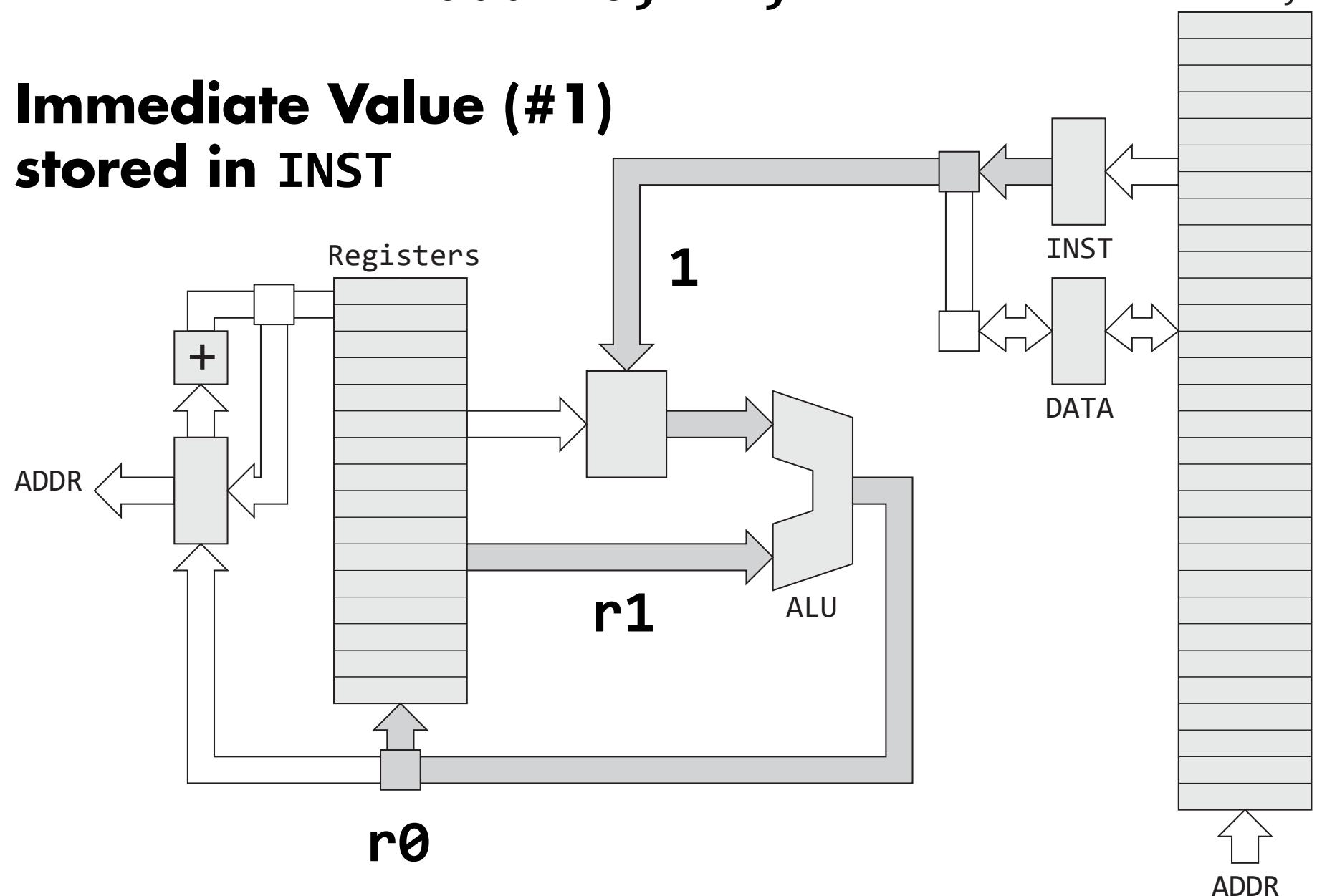
$$r0 = r1 + r2$$



ALU only operates on data in registers

add r0, r1, #1

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



Add Instruction

Meaning (defined as math or C code)

$r0 = r1 + r2$

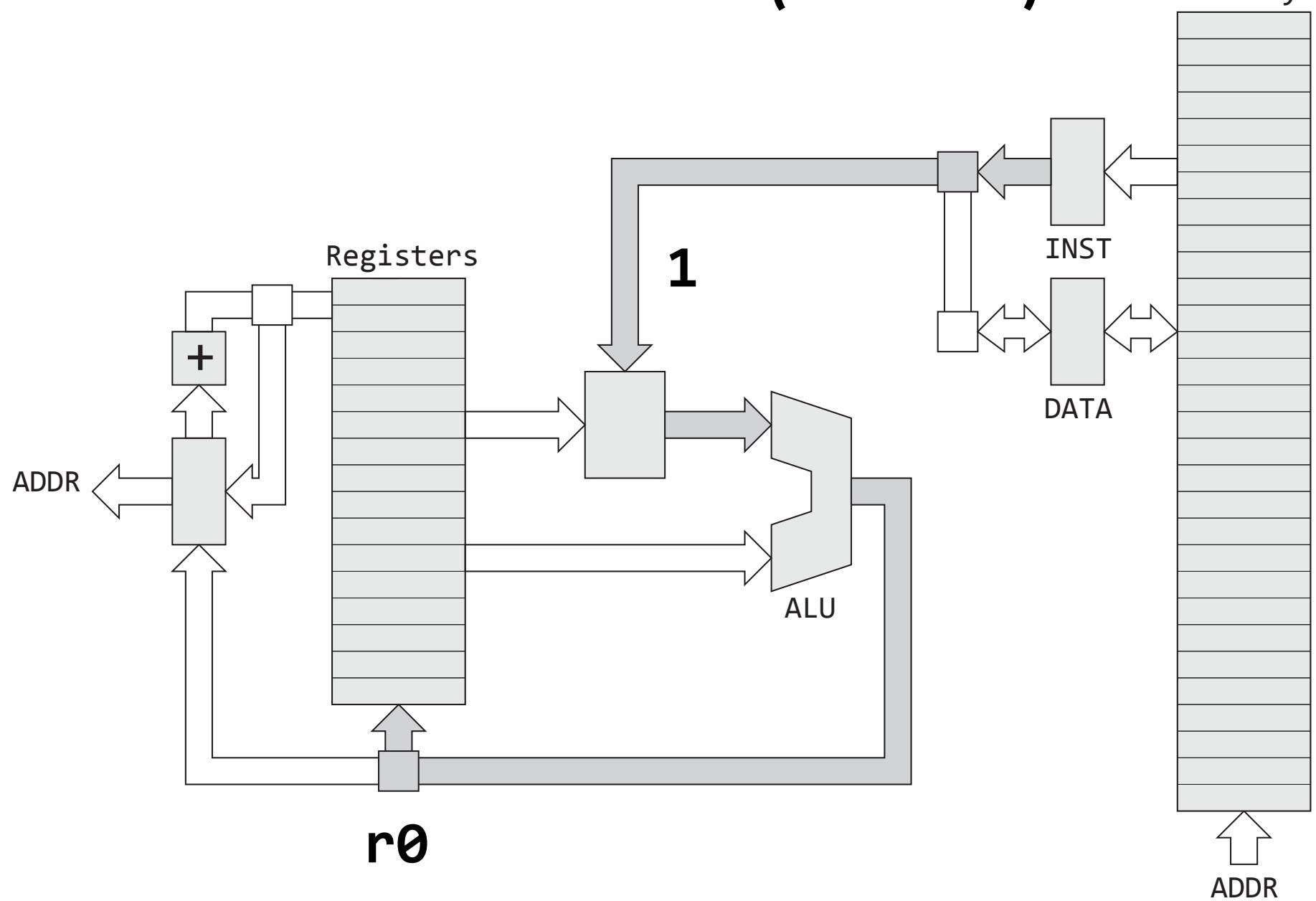
Assembly language (result is leftmost register)

`add r0, r1, r2`

Machine code (more on this later)

`E0 81 00 02`

Move Immediate (constant)



`mov r0, #1`

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

(L) Clock Cycles Current Instruction: 1 Total: 3

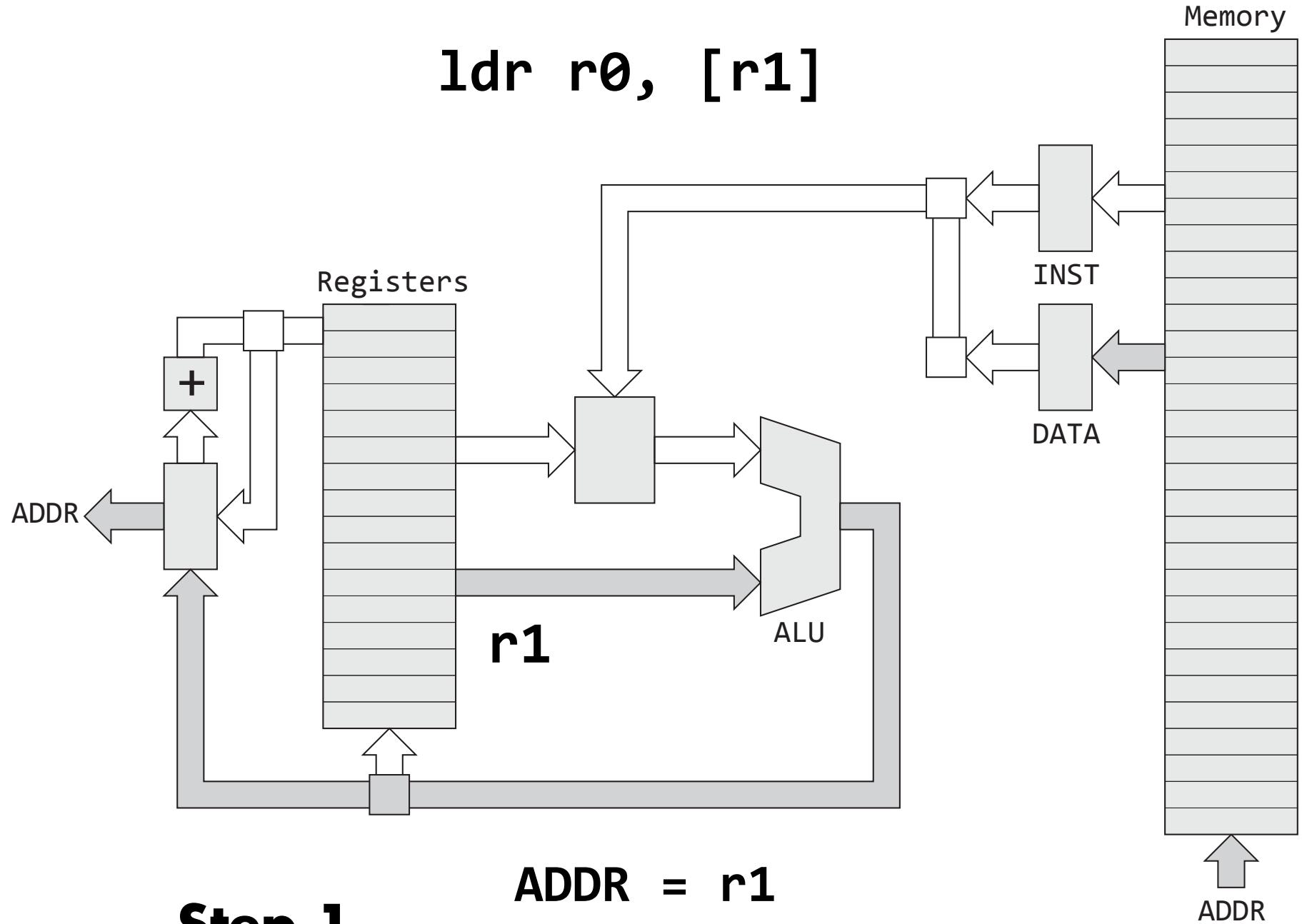
CSPR Status Bits (NZCV) 0 0 0 0

Conceptual Questions

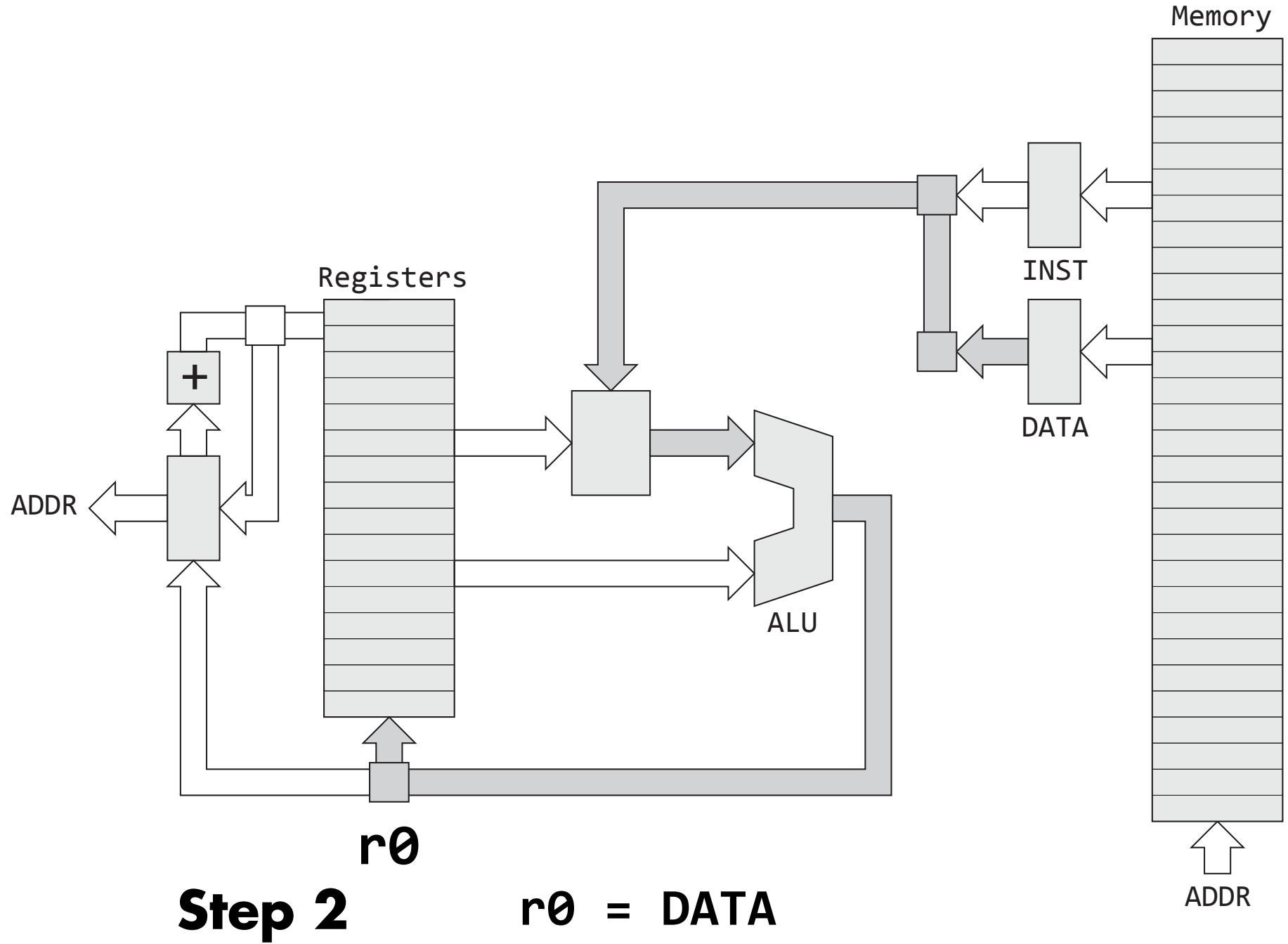
- 1. Suppose your program starts at 0x8000, what assembly language instruction could you execute to jump to and start executing instructions at that location.**
- 2. If all instructions are 32-bits, can you move any 32-bit constant value into a register using a single mov instruction?**
- 3. What is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

Load and Store Instructions

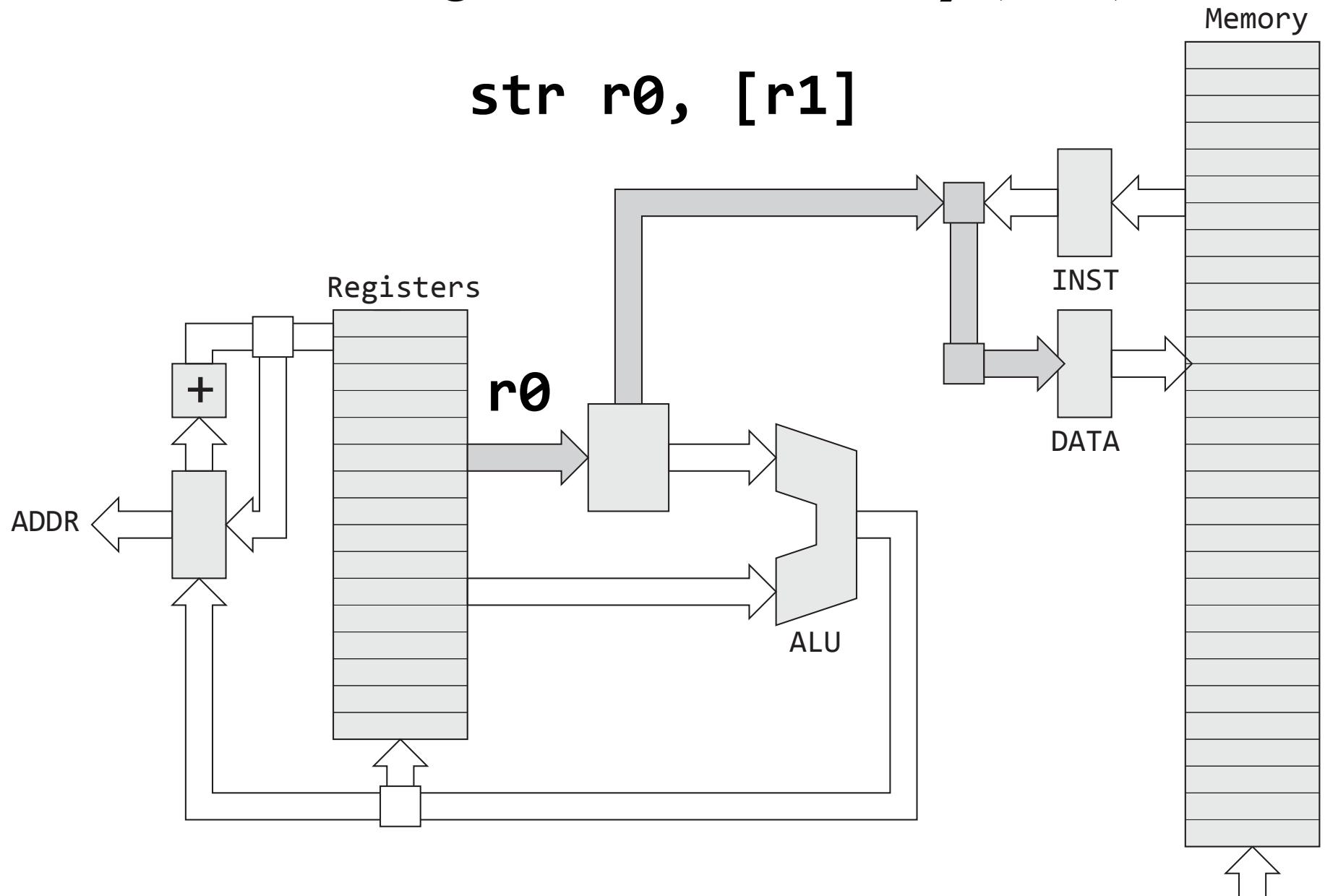
Load Register from Memory (LDR)



Load Register from Memory (LDR)



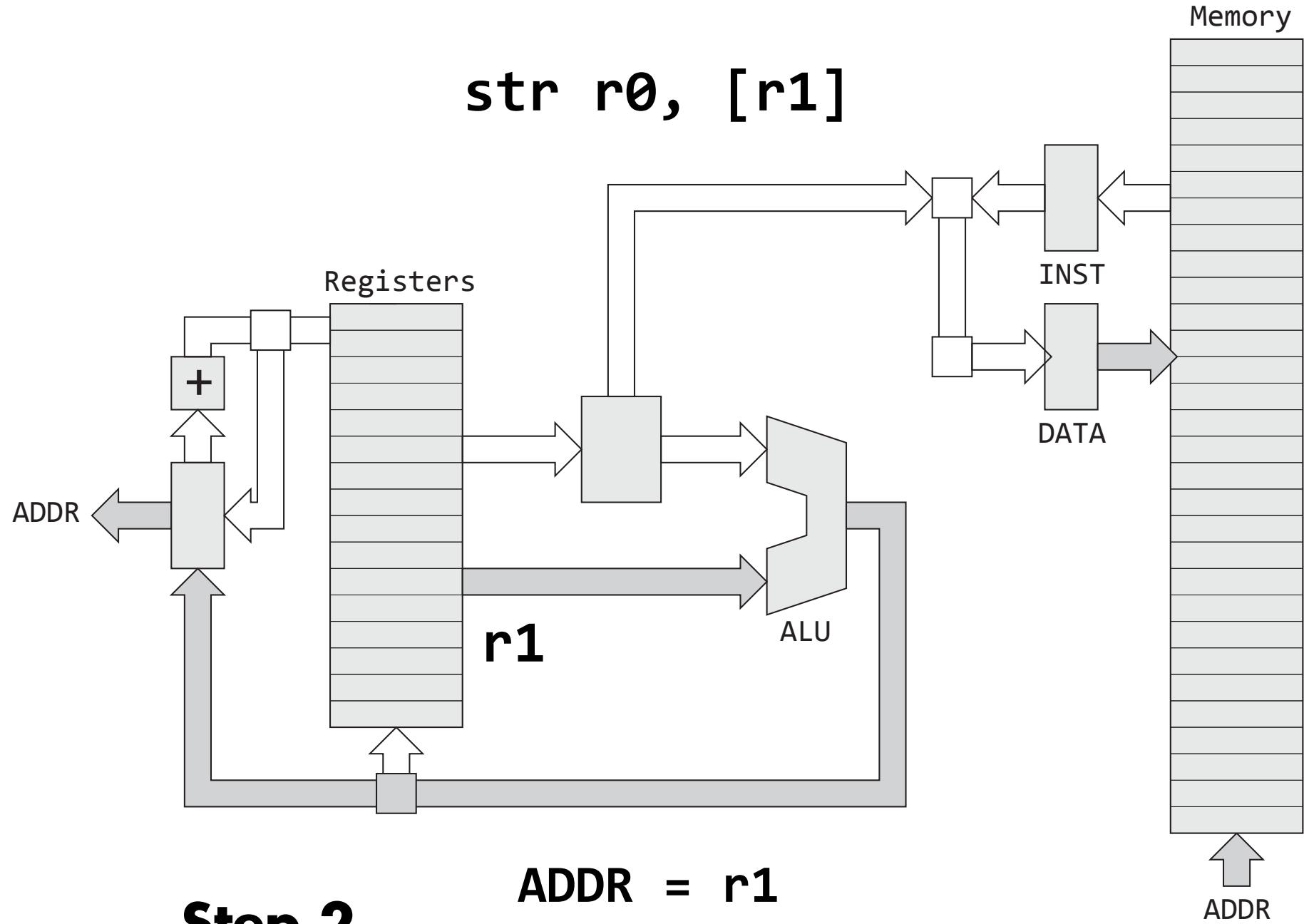
Store Register into Memory (STR)



Step 1

DATA = r0

Store Register into Memory (STR)



Step 2

$$\text{ADDR} = \text{r1}$$
$$\text{Memory}[\text{ADDR}] = \text{DATA}$$

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

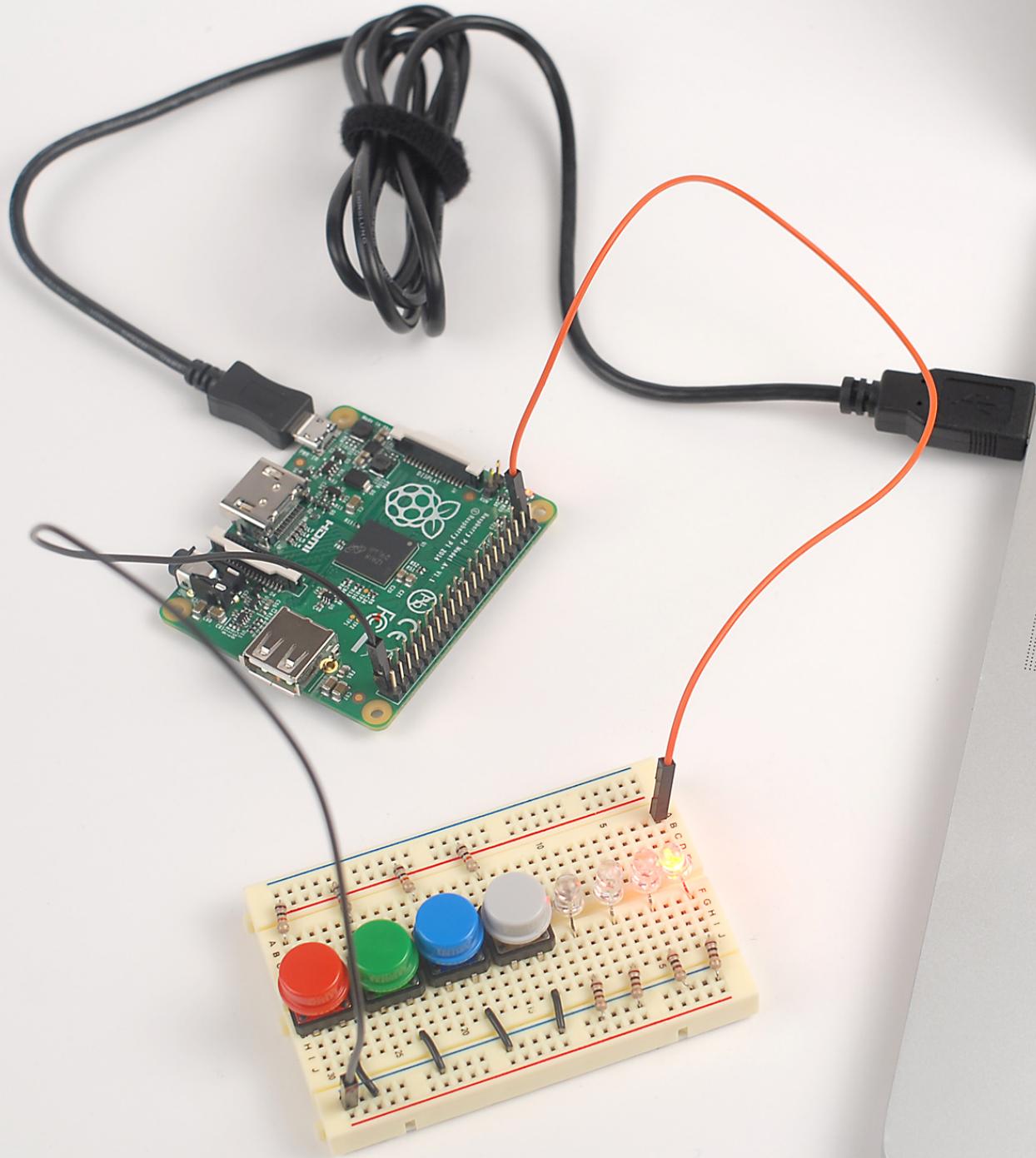
Clock Cycles

Current Instruction: 2 Total: 6

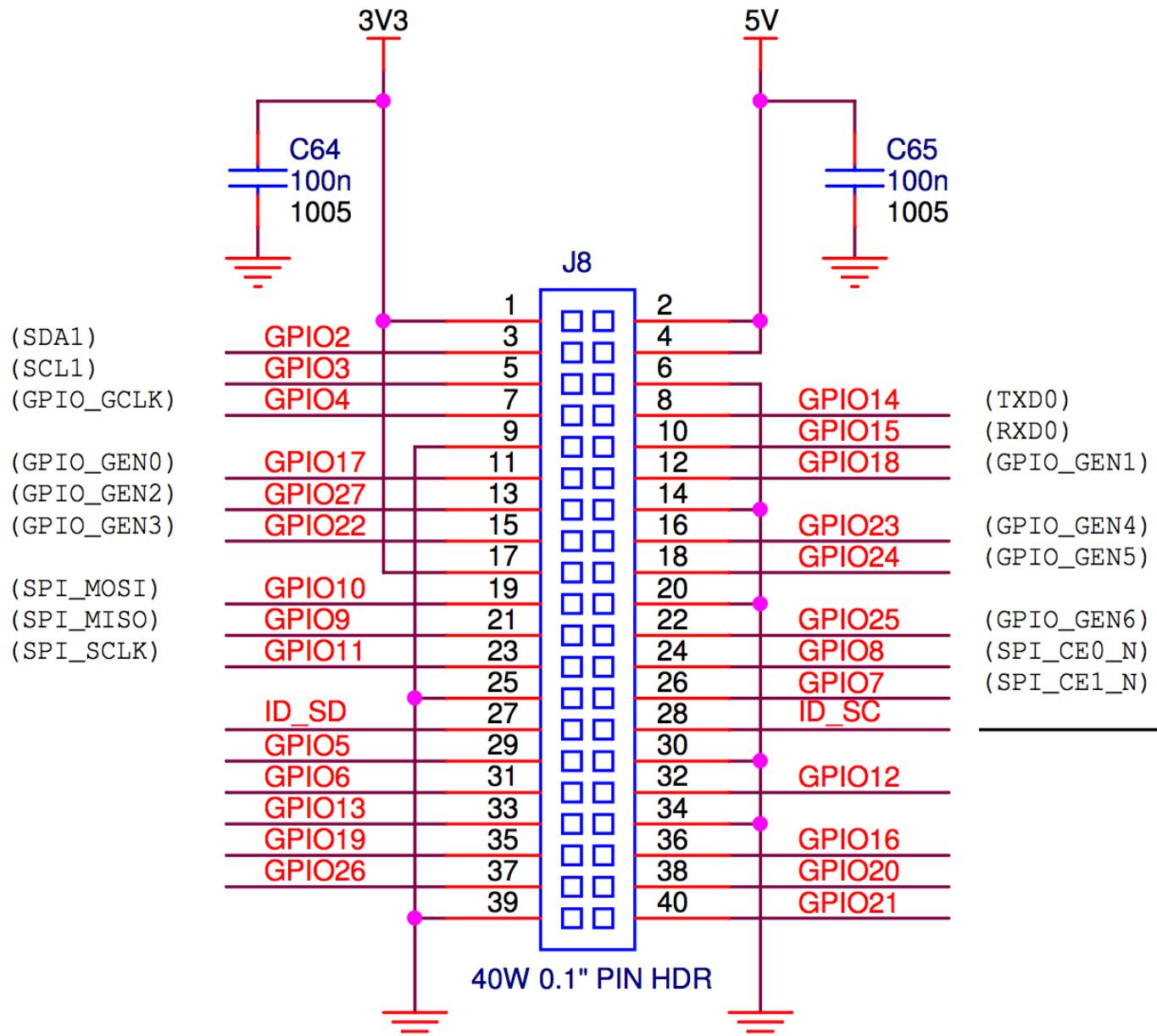
CSPR Status Bits (NZCV)

0 0 0 0

Turning on an LED

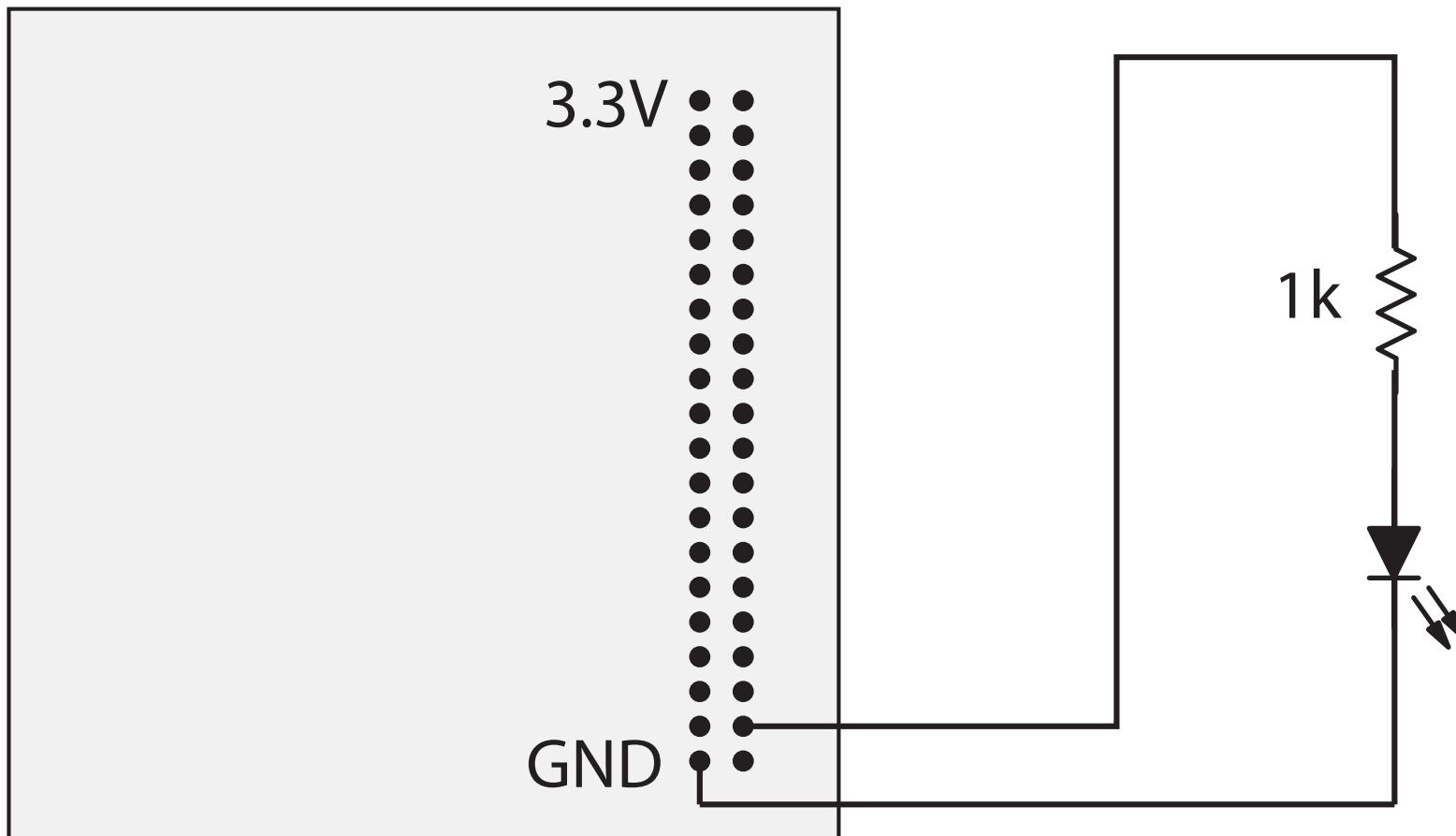


General-Purpose Input/Output (GPIO) Pins



54 GPIO Pins

Connect LED to GPIO 20



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

GPIO Pins are *Peripherals*

**Peripherals are Controlled
by Special Memory Locations**

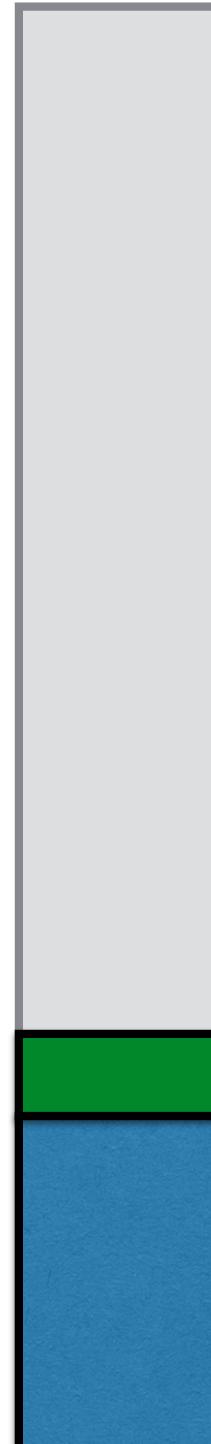
"Peripheral Registers"

Memory Map

Peripheral registers
are *mapped*
into address space

Memory-Mapped IO
(MMIO)

MMIO space is above
physical memory

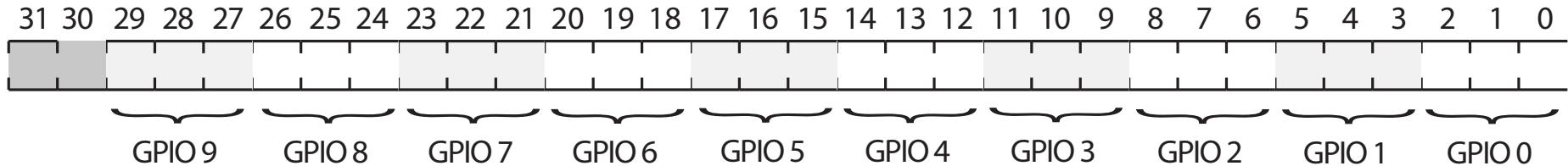


100000000_16
4 GB

020000000_16

512 MB

GPIO Function Select 0 Register

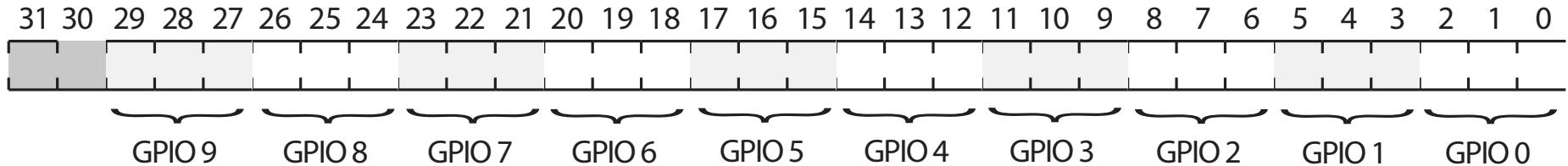


Bit pattern	Pin Function
000	The pin is an input
001	The pin is an output
100	The pin does alternate function 0
101	The pin does alternate function 1
110	The pin does alternate function 2
111	The pin does alternate function 3
011	The pin does alternate function 4
010	The pin does alternate function 5

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

3 bits are used to select function

GPIO Function Select 10Register



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

8 functions requires 3 bits to specify

10 pins times 3 bits = 30 bits

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 ...

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

```
// Set GPIO20 to be an output
// GPIO20 FSEL is in the 2nd register
// FSEL2 = 0x20200008

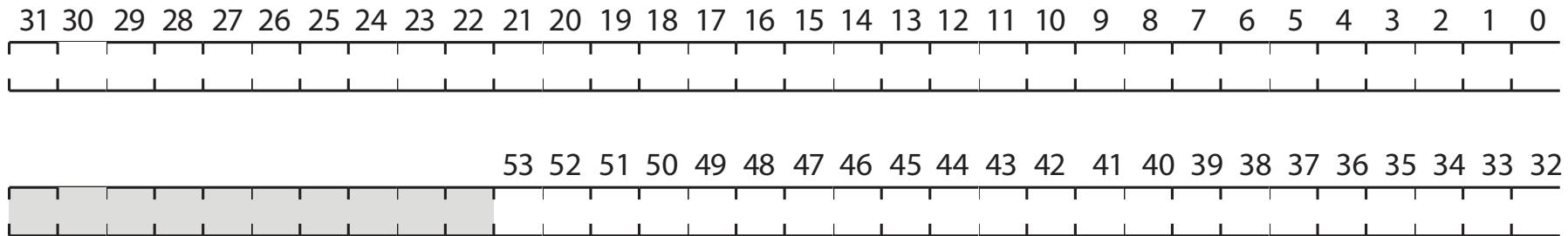
mov r0, #0x20          // #0x00000020
lsl r1, r0, #24        // #0x20000000
lsl r2, r0, #16        // #0x00200000
orr r1, r1, r2         // #0x20200000
orr r1, r1, #0x08      // #0x20200008

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 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

```
// 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
```

```
# Convert assembly language  
# to machine code  
% arm-none-eabi-as on.s -o on.o
```

```
# Extract machine code into a "binary"  
% arm-none-eabi-objcopy on.o -O binary  
on.bin
```

```
# What to do on your laptop
```

```
# Insert SD card - Volume mounts
```

```
% ls /Volumes/
```

```
BOOT2022  Macintosh HD
```

```
# Copy to SD card
```

```
% cp on.bin /Volumes/BOOT22/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

Memory stores bits, all information stored as bits

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