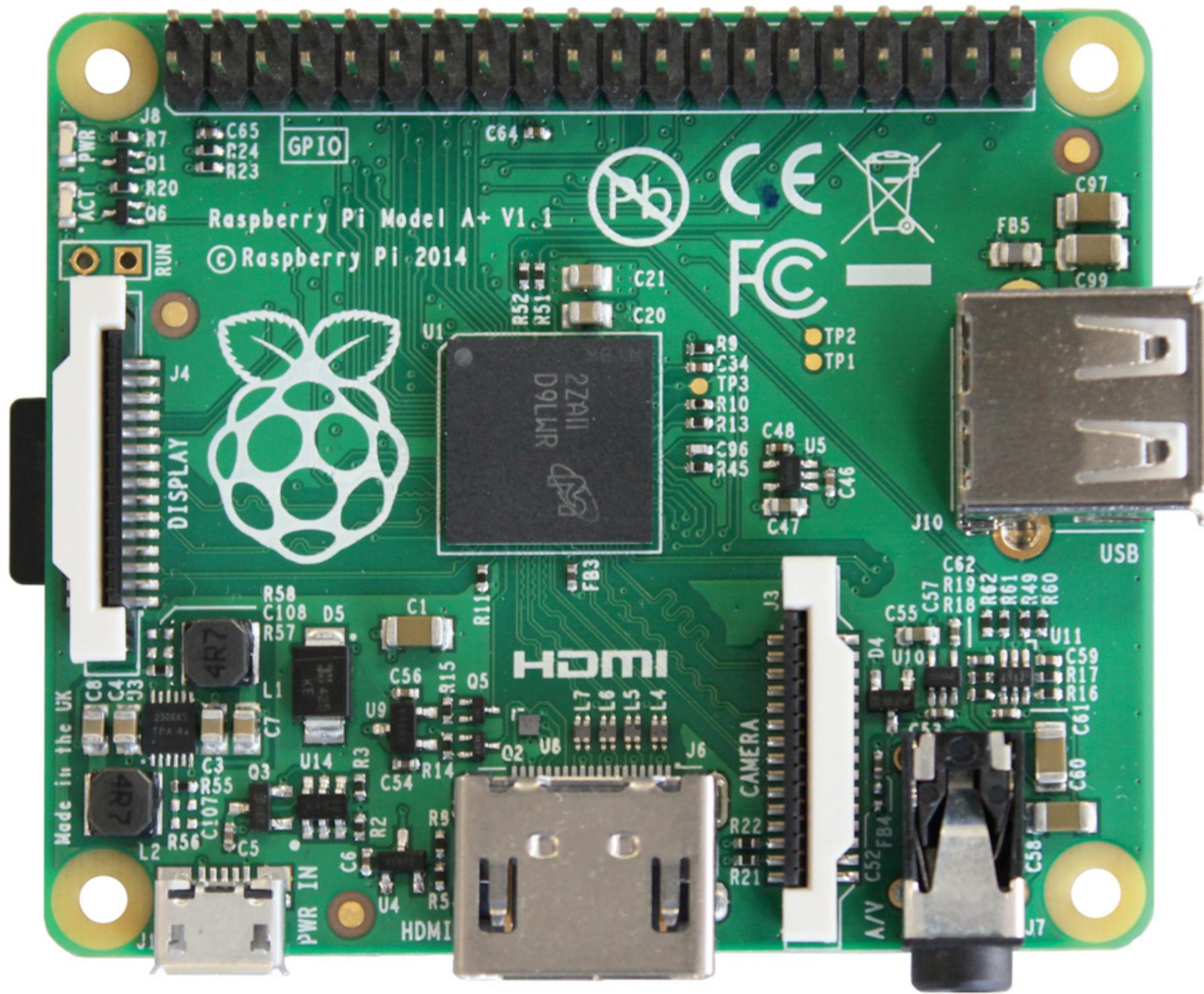
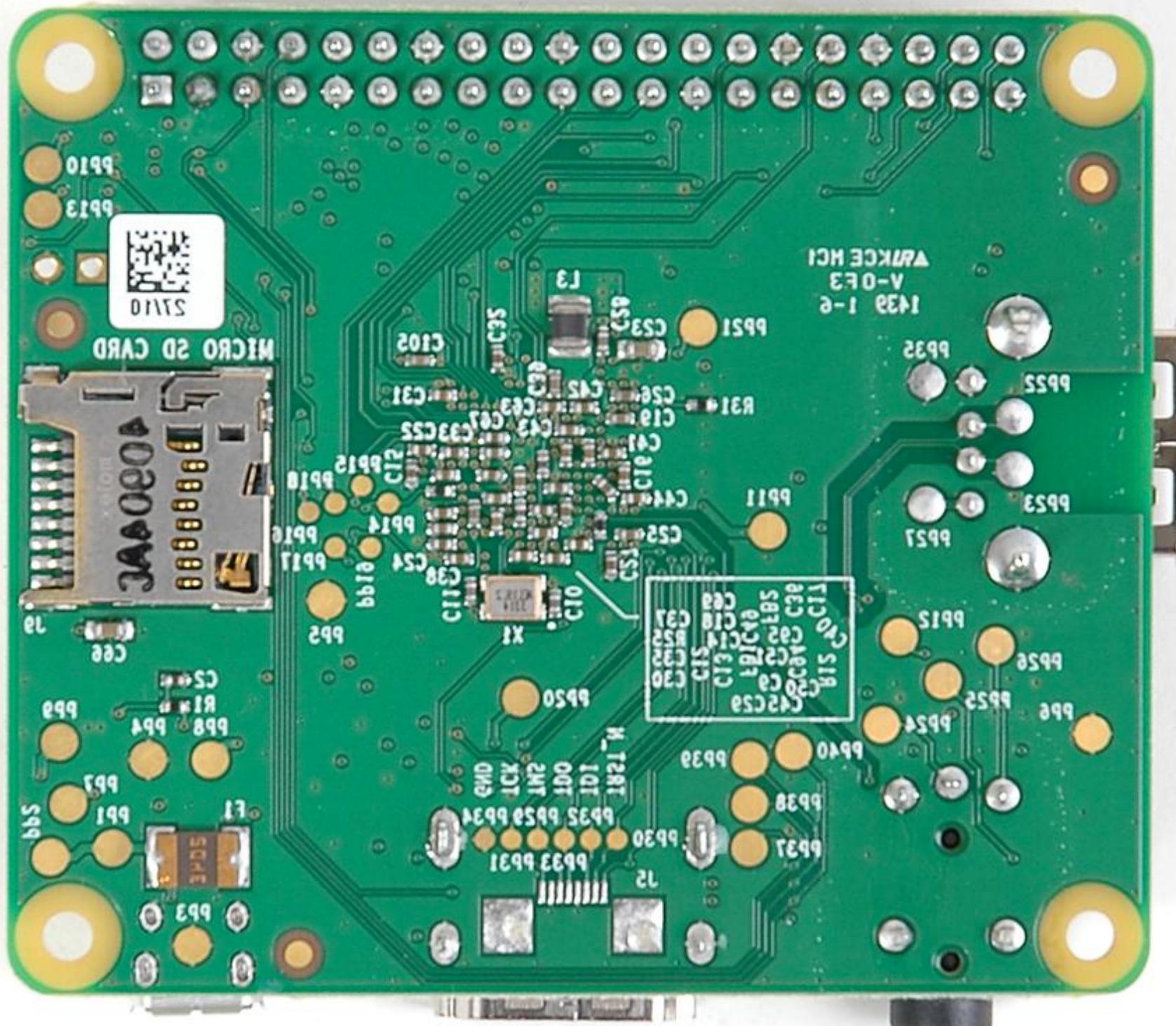


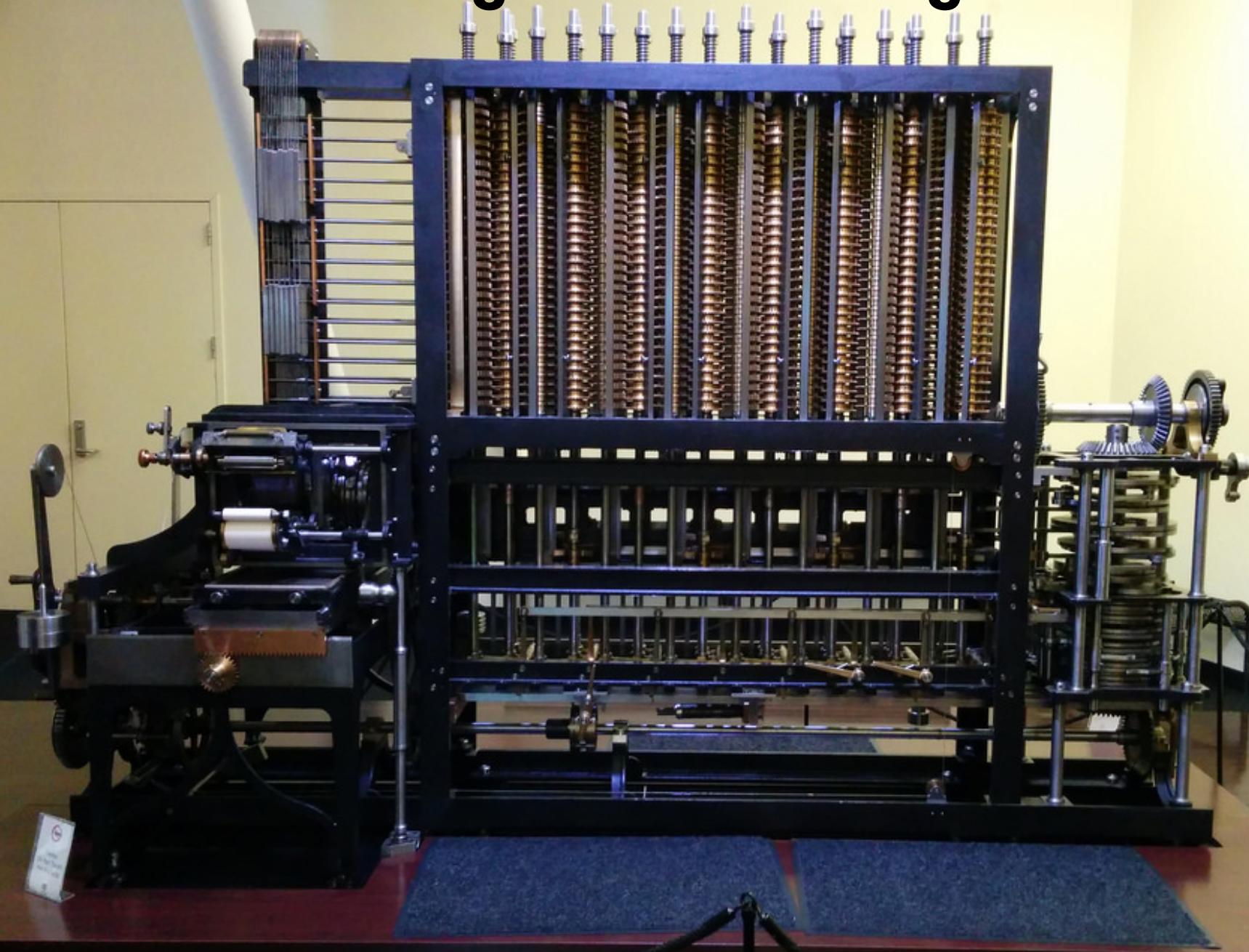
ARM Processor and Memory Architecture

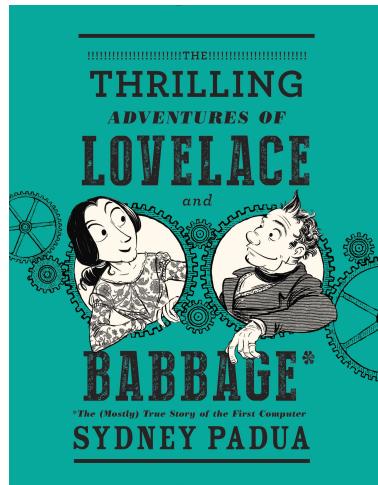
Goal: Turn on an LED



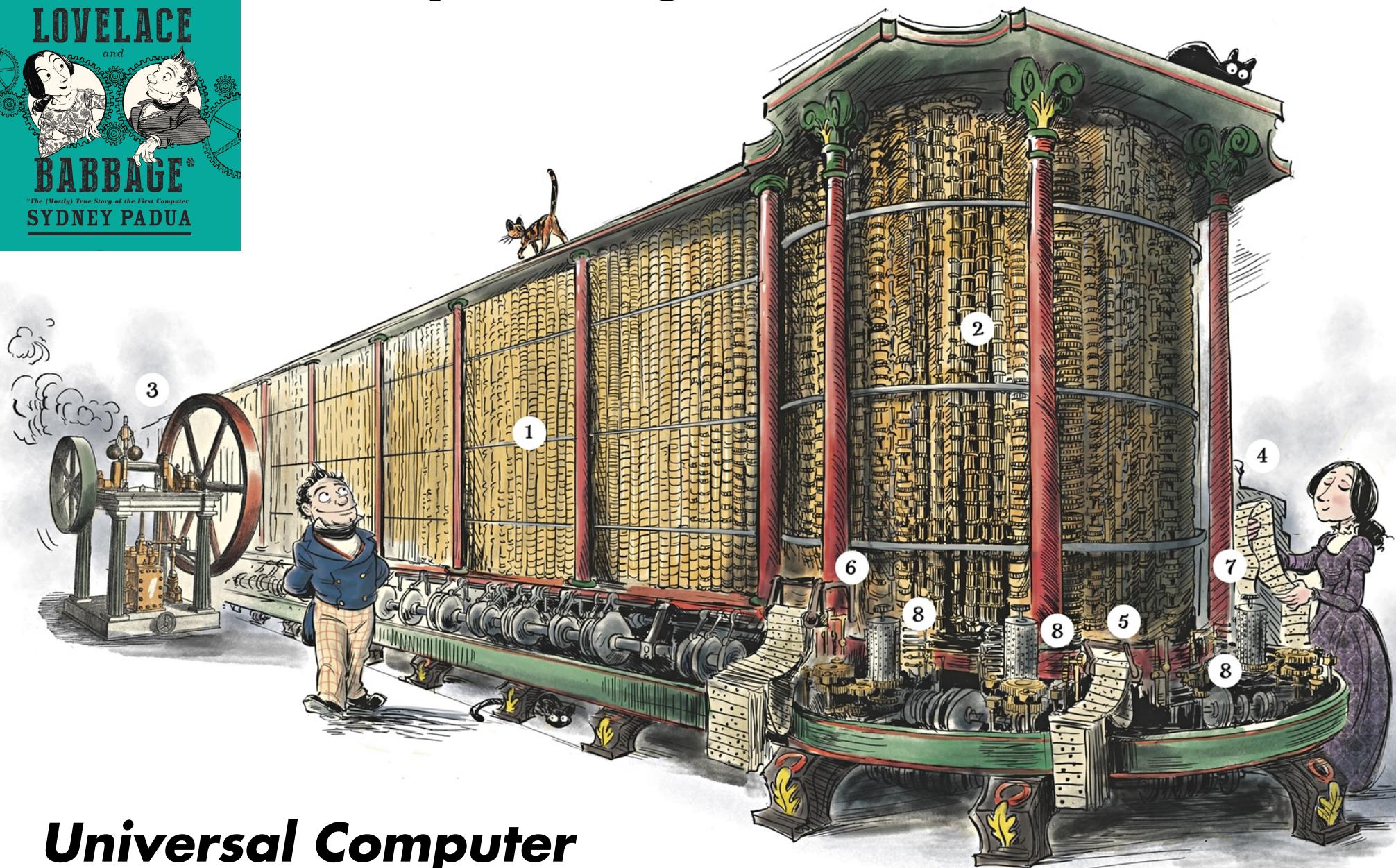


Babbage Difference Engine



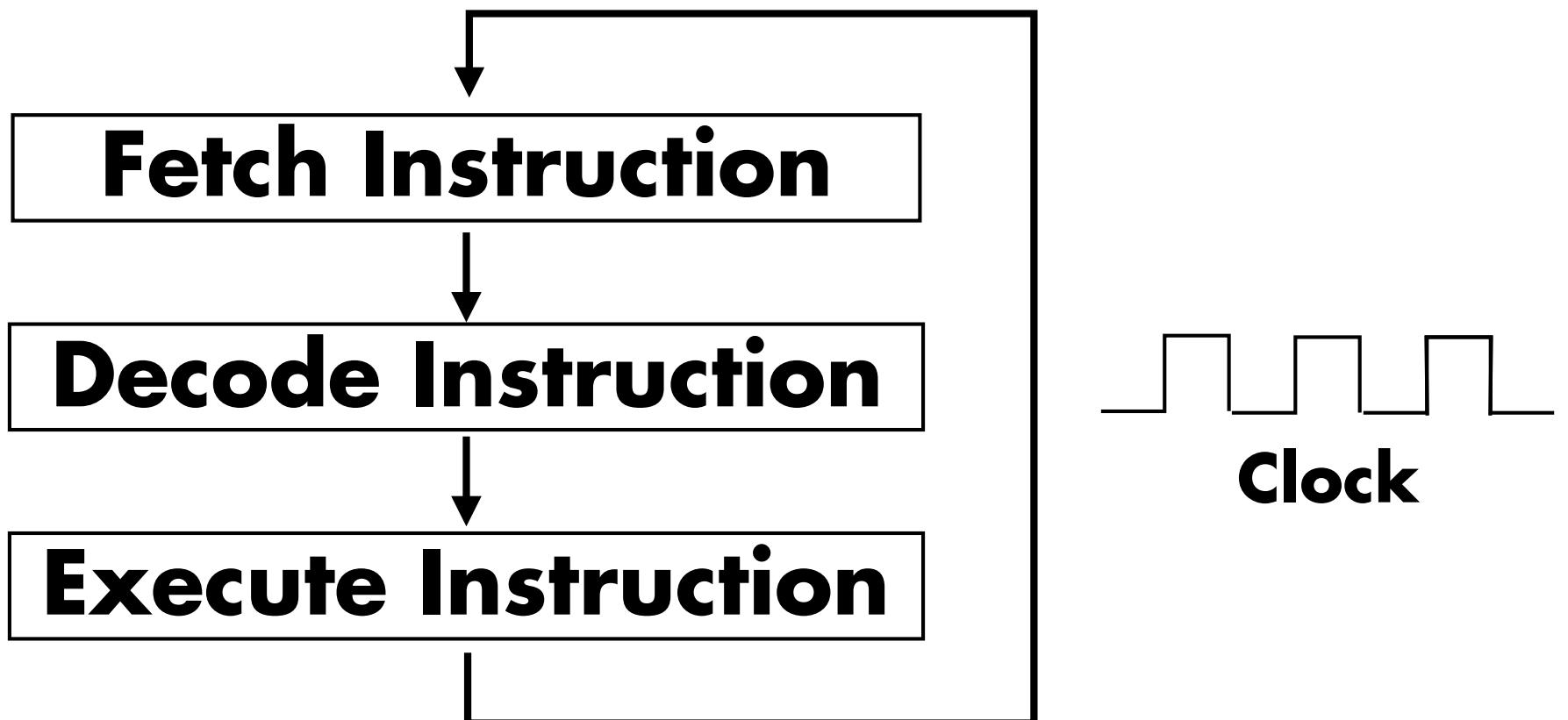


Analytical Engine



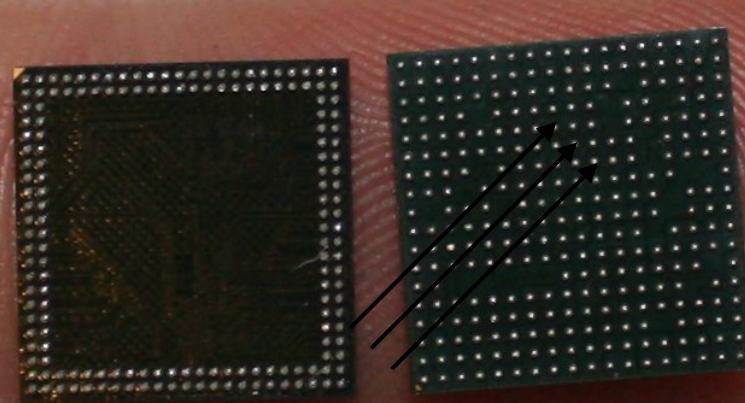
Universal Computer

Running a "Program"



Package on Package

Broadcom 2865 ARM Processor



Samsung 4Gb SDRAM

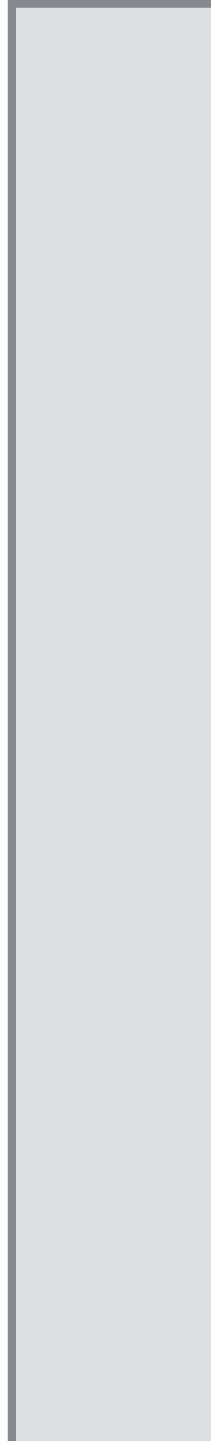
Memory used to store information

Stores both instructions and data

Storage locations are accessed using 32-bit addresses

Address refers to a byte (8-bits)

Maximum addressable memory 4 GB



10000000₁₆

Memory Map

00000000₁₆

Memory used to store information

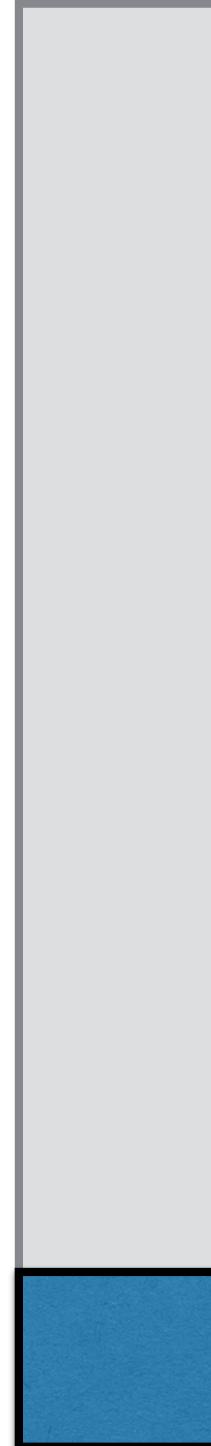
Stores both instructions and data

Storage locations are accessed using 32-bit addresses

Address refers to a byte (8-bits)

Maximum addressable memory 4 GB

Actual memory is 512 MB



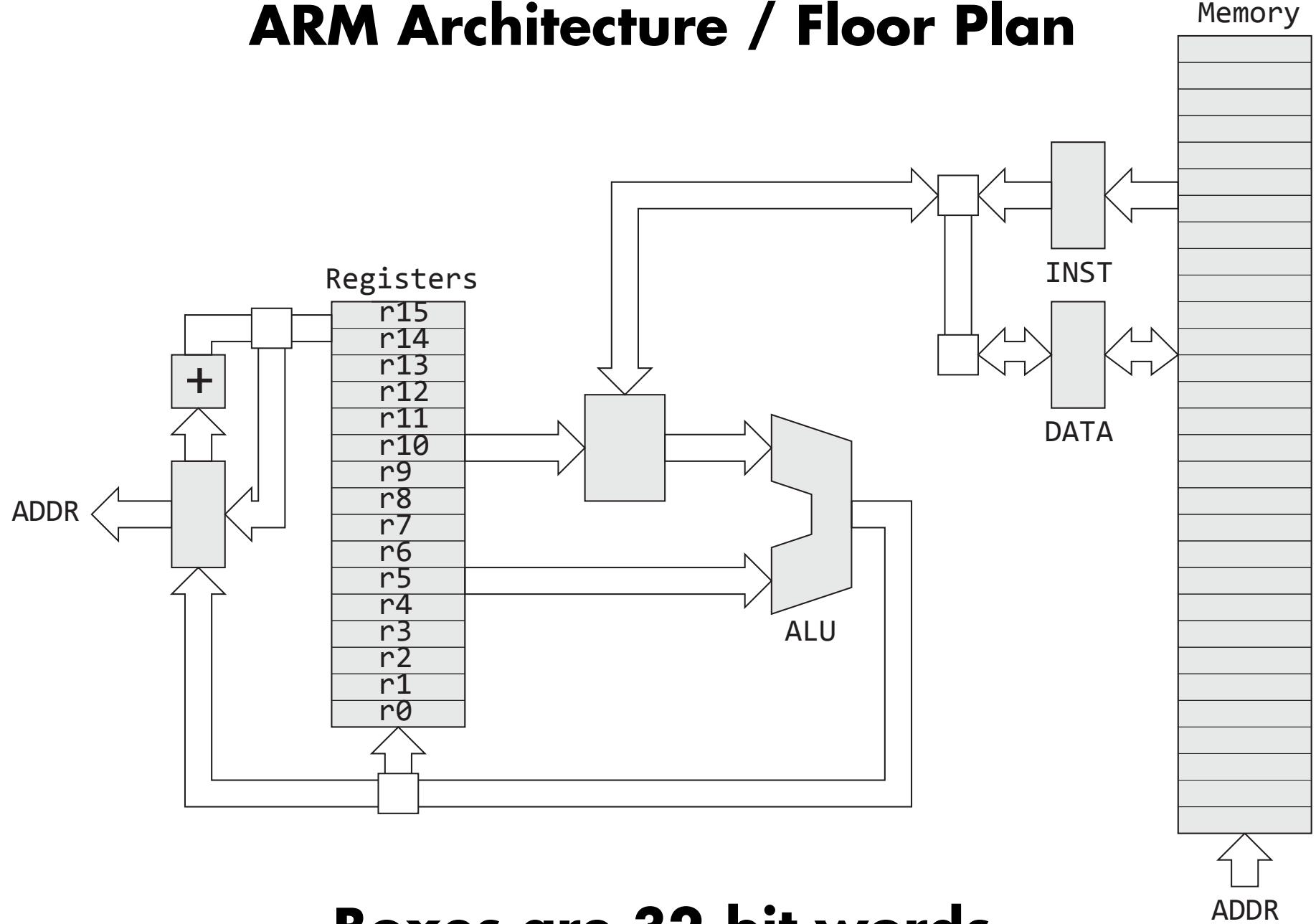
100000000_{16}

Memory Map

020000000_{16}

512 MB

ARM Architecture / Floor Plan

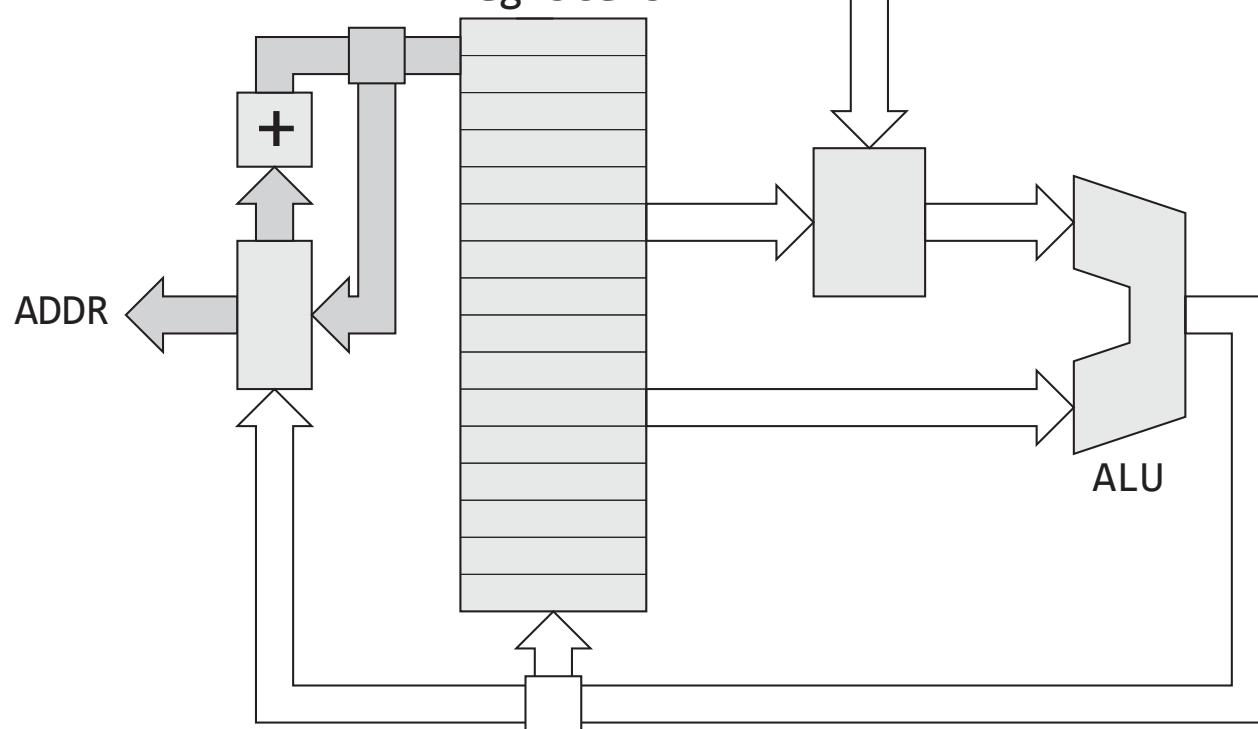


Boxes are 32-bit words

pc : program counter

$pc = pc + 4$

$pc = r15$



**Fetch INST from mem[pc]
Decode**

Memory

ADDR

ALU Instructions

Add Instruction

Meaning (defined as math or code)

$$r_0 = r_1 + r_2$$

Assembly language

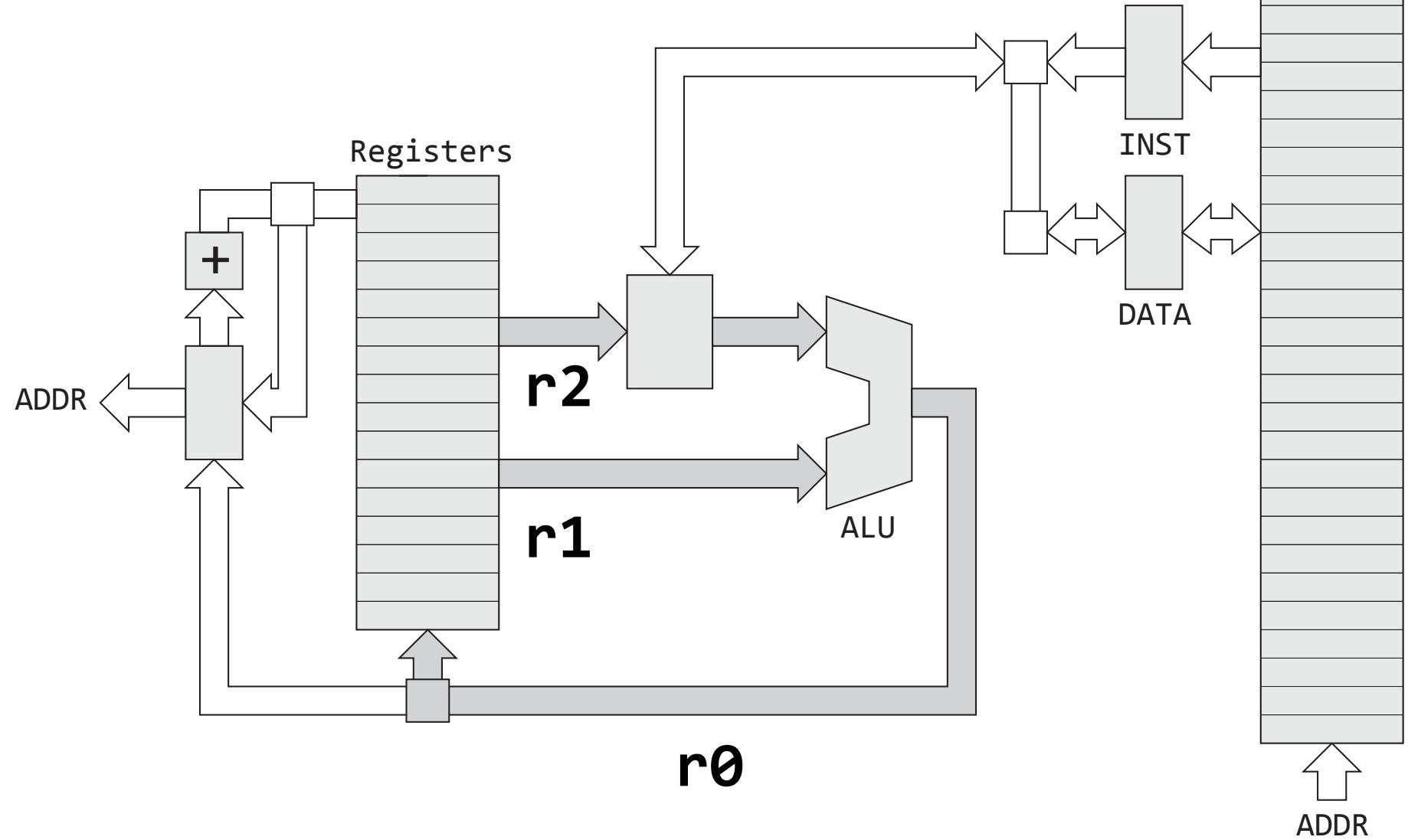
add r0, r1, r2

Machine code

E0 81 00 02

add r0, r1, r2

$$r0 = r1 + r2$$



Execute INST

```
# 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
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

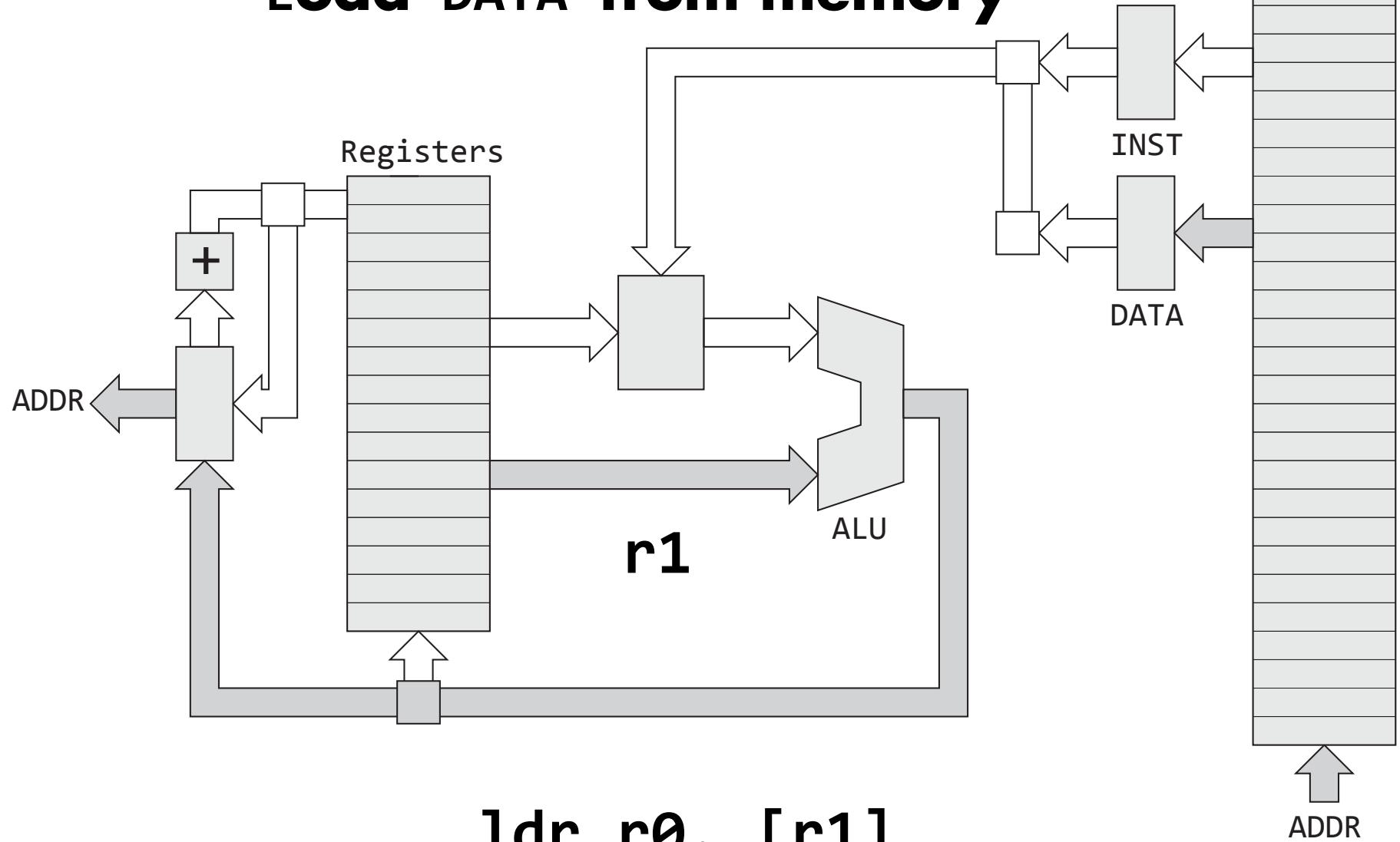
(L) Clock Cycles Current Instruction: 1 Total: 3

CSPR Status Bits (NZCV) 0 0 0 0

Load and Store Instructions

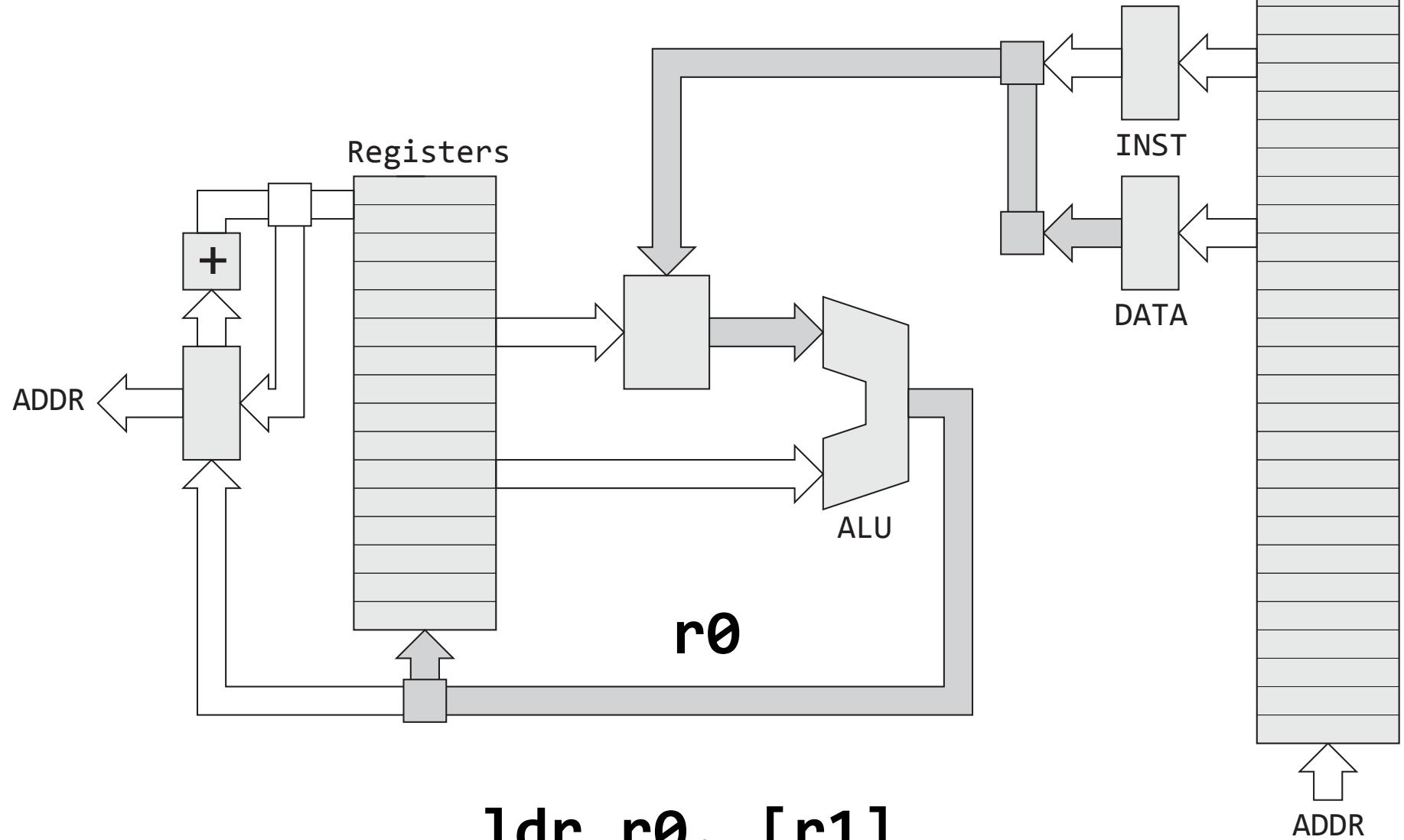
$$r0 = \text{mem}[r1]$$

Generate ADDR
Load DATA from memory



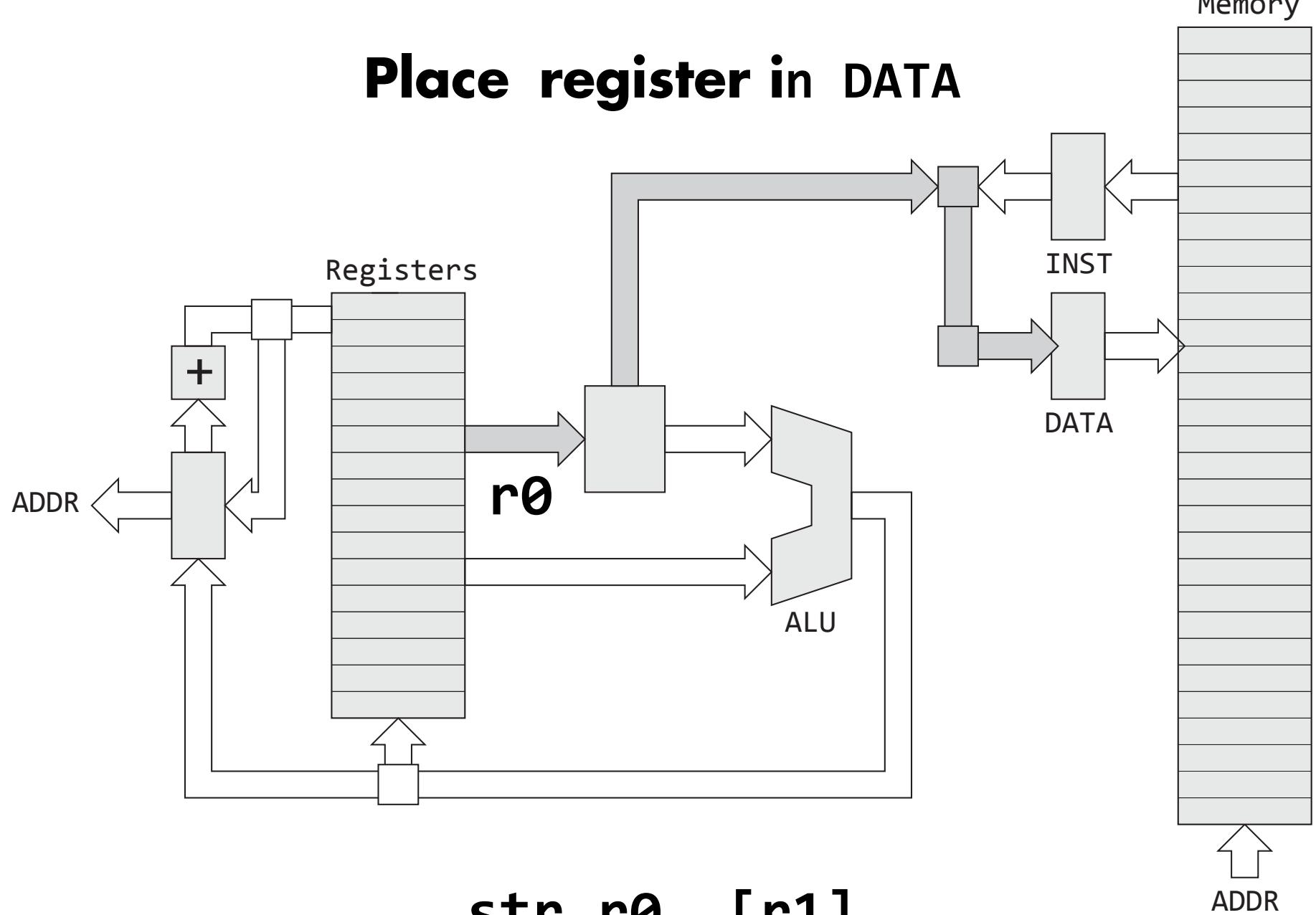
$$r0 = \text{mem}[r1]$$

Place DATA in register



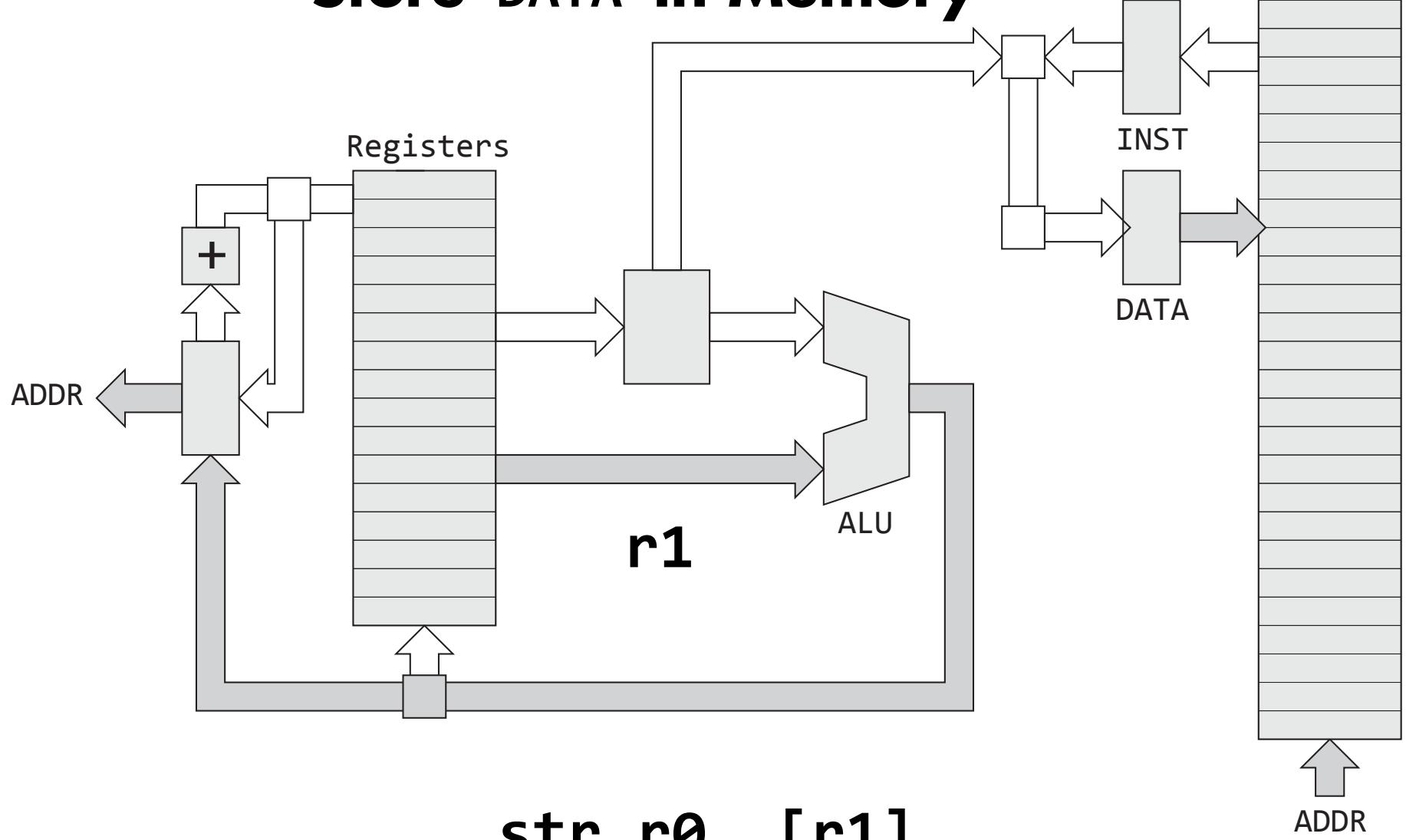
mem[r1] = r0

Place register in DATA



$\text{mem}[r1] = r0$

**Generate ADDR
Store DATA in Memory**



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

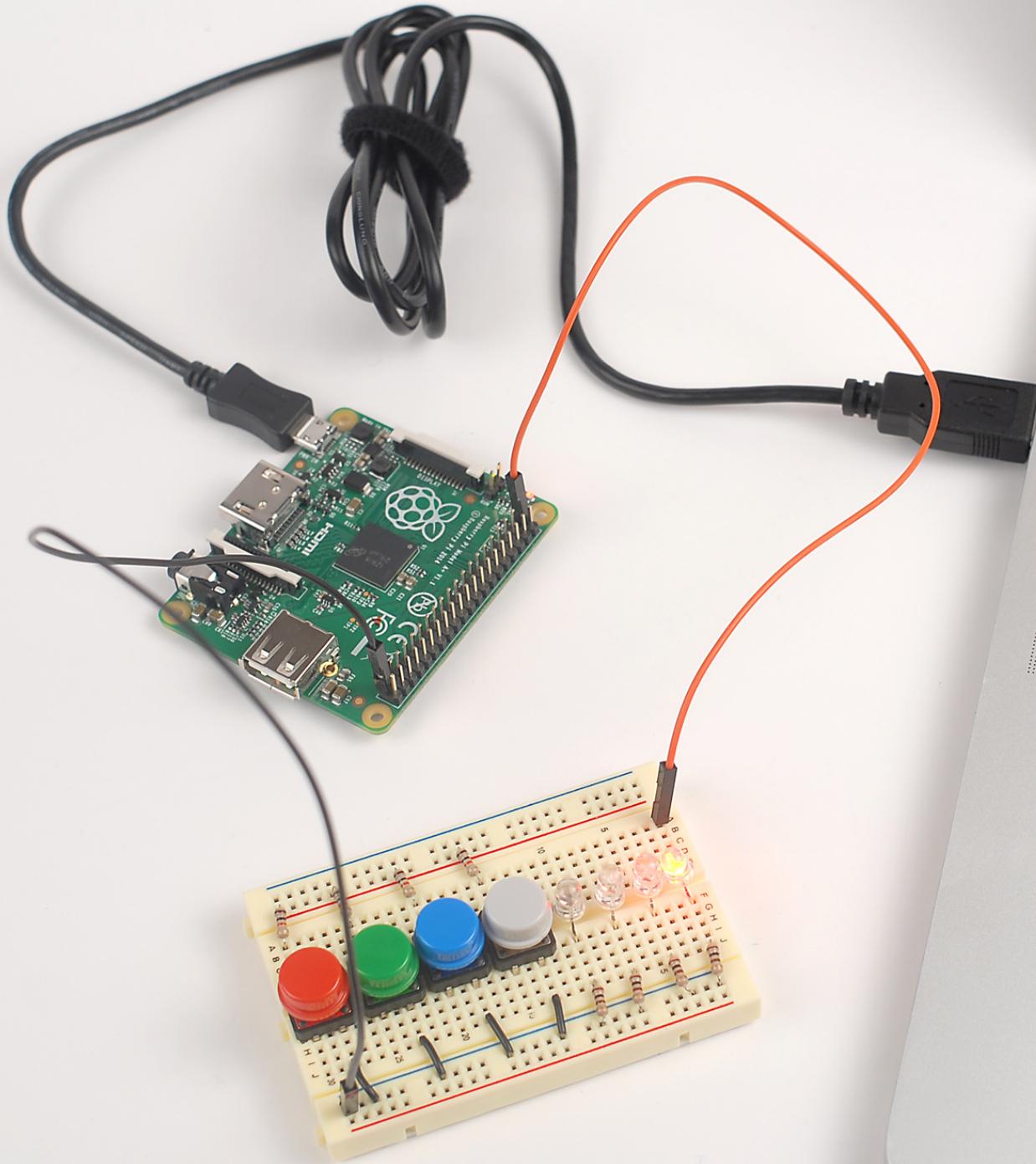
CSPR Status Bits (NZCV)

0 0 0 0

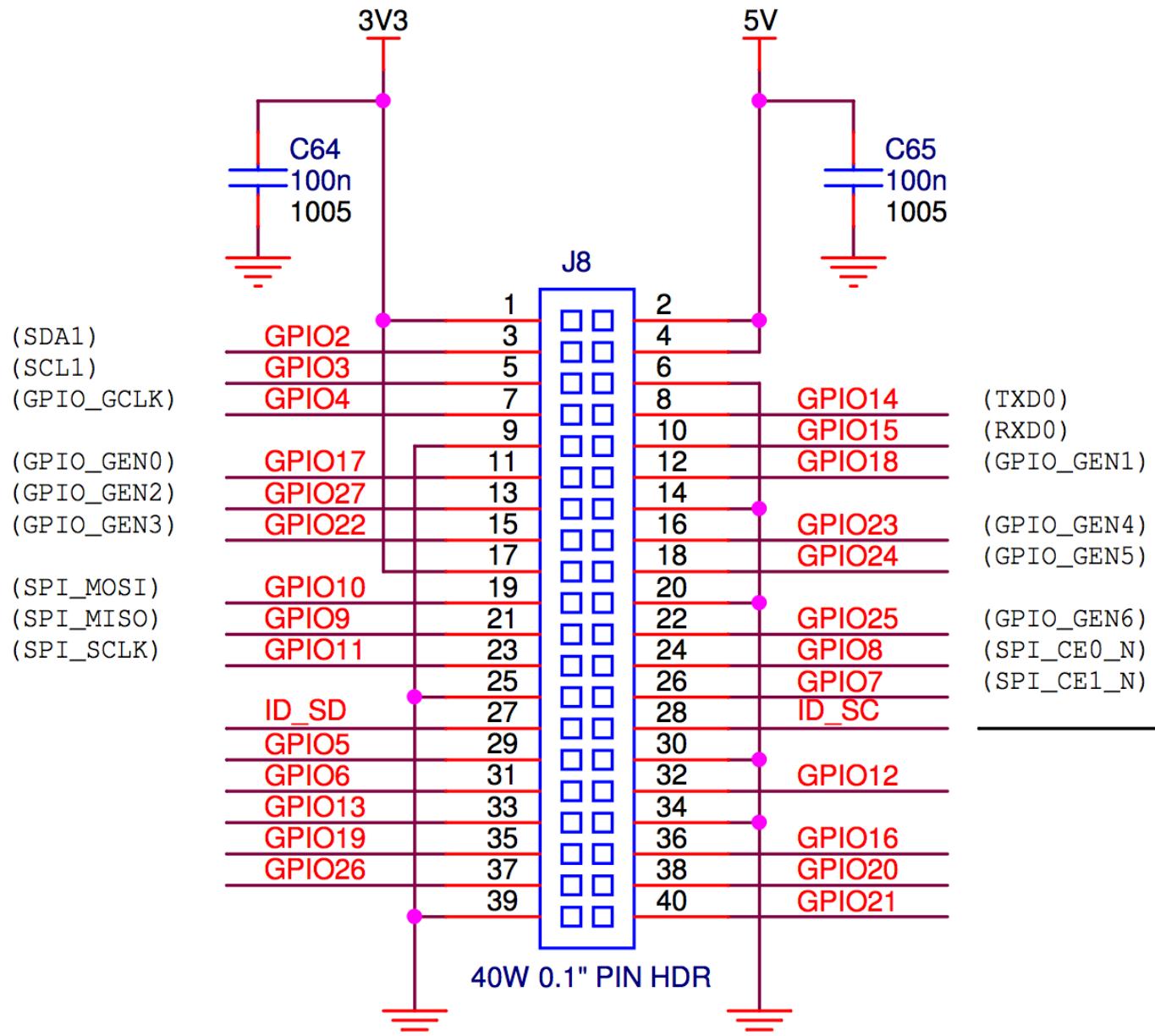
Conceptual Questions

- 1. Suppose your program starts at 0x8000, how could you jump and start executing instructions at that location?**
- 2. All instructions are 32-bits. Can you mov any 32-bit immediate constant to a register using the mov instruction?**
- 3. What instruction do you think takes longer to execute, ldr or add?**

Turning on an LED

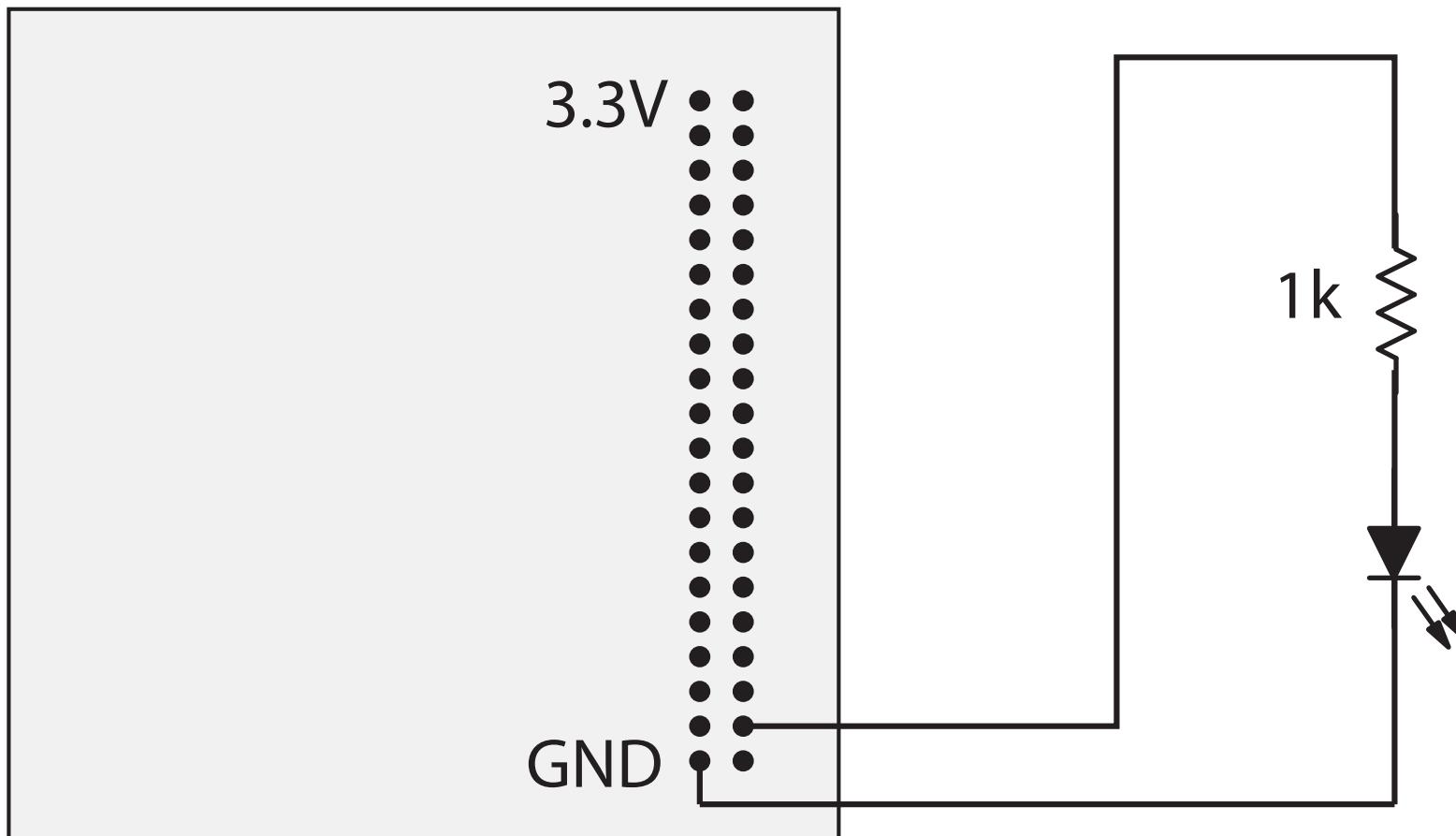


General-Purpose Input/Output (GPIO) Pins



54 GPIO Pins

Connect LED to GPIO 20

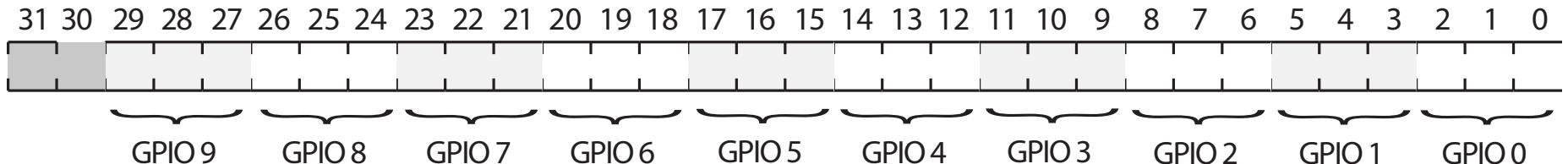


Logic 1 -> 3.3V
Logic 0 -> 0.0V (GND)

How to Control GPIO Pins?

Peripheral Registers!

GPIO Function Select Register



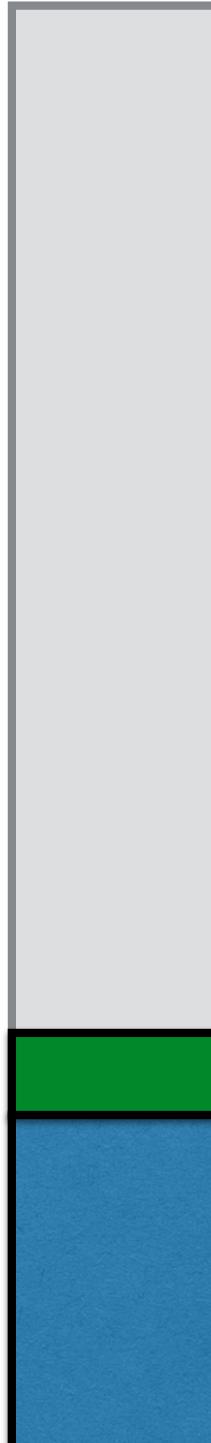
3 bits per GPIO pin

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

Max of 10 pins per 32-bit register

Memory Map

100000000_{16}
4 GB



Memory-Mapped IO (MMIO)

Peripheral Registers

02000000_{16}

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	-	-

Gotcha

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

```
// Turn on an LED via GPIO 20  
  
// FSEL2=0x20200008 controls pins 20-29  
  
// load r0 with GPIO FSEL2 address  
ldr r0, =0x20200008  
  
// GPIO 20 function select is bits 0-2  
// The value of 1 indicates OUTPUT  
// load r1 with 1  
mov r1, #1  
  
// store value in r1 to address in r0  
str r1, [r0]
```

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

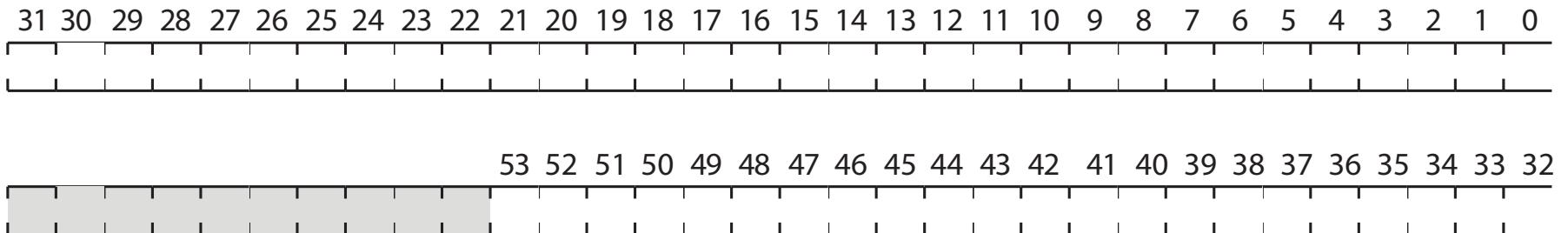
Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin <i>n</i> .	R/W	0

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**

...

```
// load r0 with GPIO SET0 register addr  
ldr r0, =0x2020001C
```

```
// set bit 20 in r1  
mov r1, #0x100000 // 0x100000 = 1 << 20
```

```
// store bit in GPIO SET0 register  
str r1, [r0]
```

...

```
// load r0 with GPIO SET0 register addr  
ldr r0, =0x2020001C
```

```
// set bit 20 in r1  
mov r1, #0x100000 // 0x100000 = 1 << 20
```

```
// store bit in GPIO SET0 register  
str r1, [r0]
```

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

```
# 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!!  
#
```



Next Lecture

More Instructions in Detail