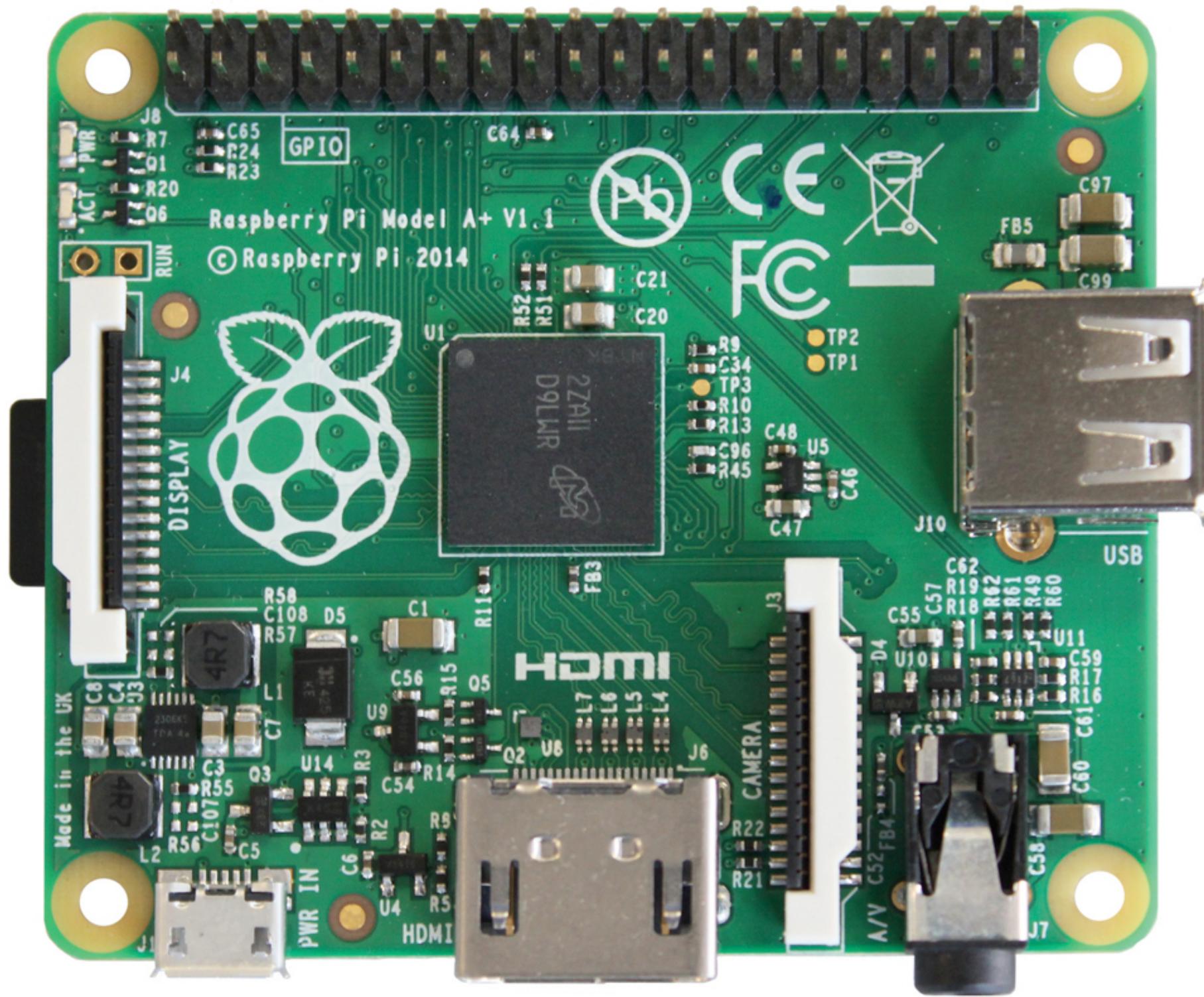


ARM Processor and Memory Architecture

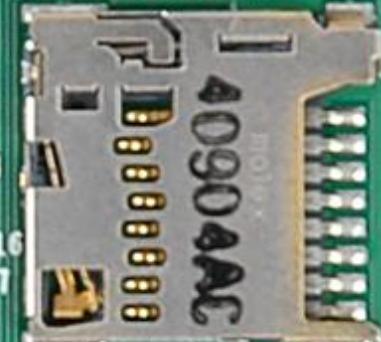
Goal: Turn on an LED



ARUKCE MC1
V-OF3
1439 1-6



MICRO SD CARD



J9

C66

R12	C10	C17	C36	C69	C37	R25
C50	C9	F8	C49	C18	C14	C12
C51	C9	F8	C49	C13	C12	C35
C52	C9	F8	C49	C12	C11	C30
C45	C29					

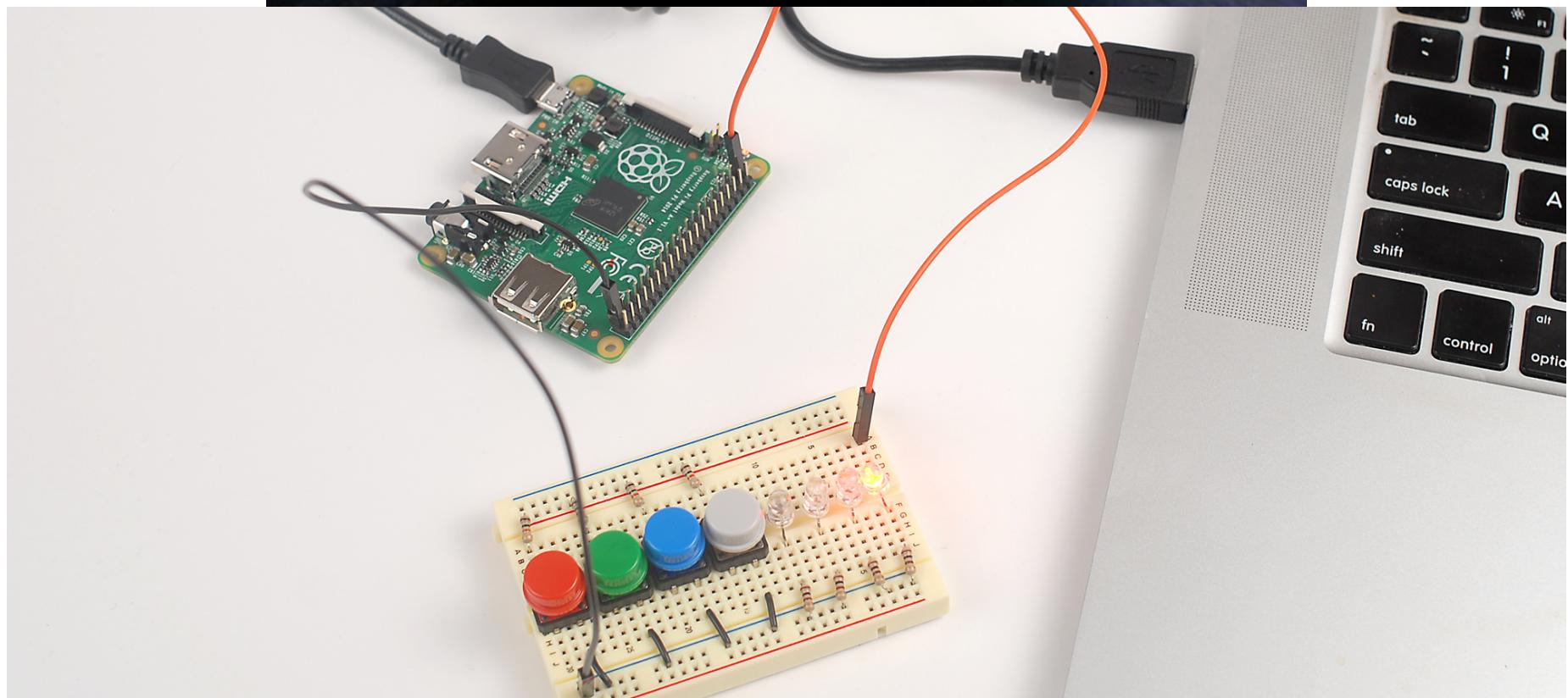
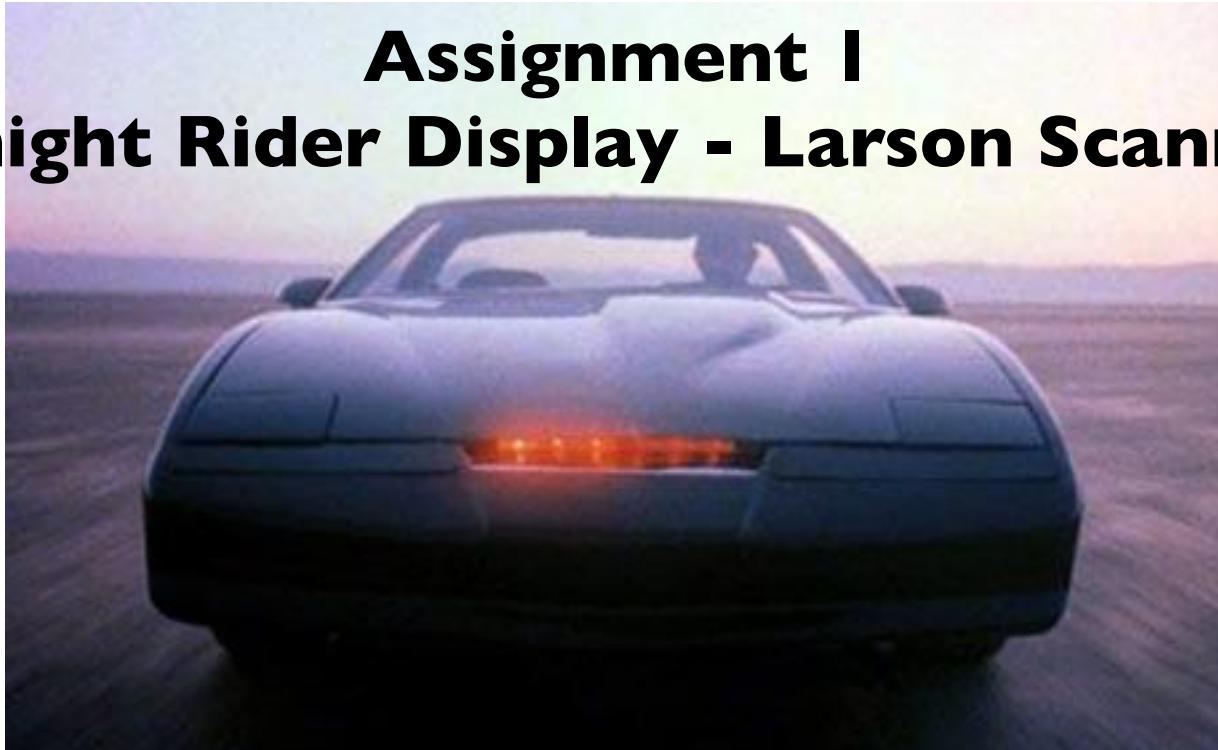
L3
PP21 C23 C28
R31 C26 C42 C3
C19 C41 C16 C43 C67 C31
C44 C10 C15 C33 C22
C25 C27 C38 C24
C10 X1 C11 C38
PP15 PP18
PP14 PP16
PP19 PP17
PP5

J551 N
PP31 J5
PP32 TDI
PP30 TDO
PP38 THS
PP39 TCK
PP37 GND
PP30 PP29 PP34
PP33 PP31

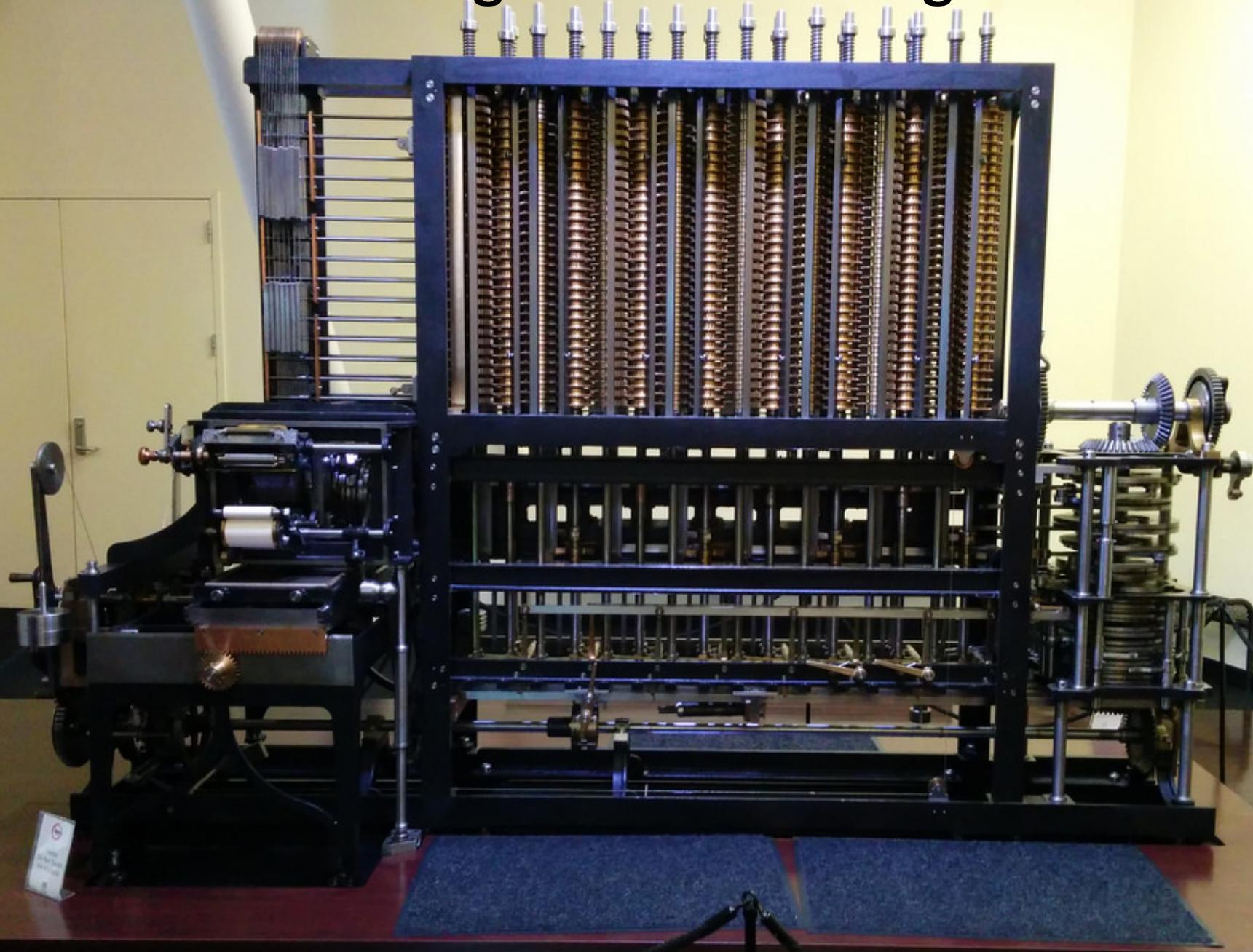
C24
R1m
PP8 PP4
F1
PP7
PP1
PP3
PP2

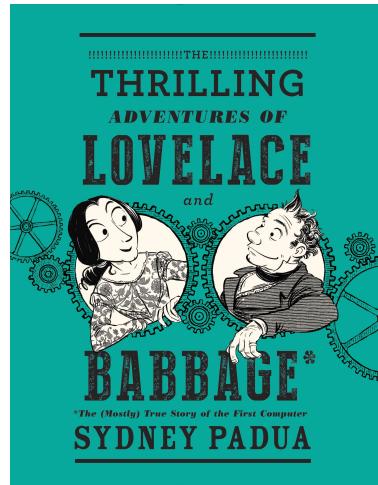
Assignment I

Knight Rider Display - Larson Scanner

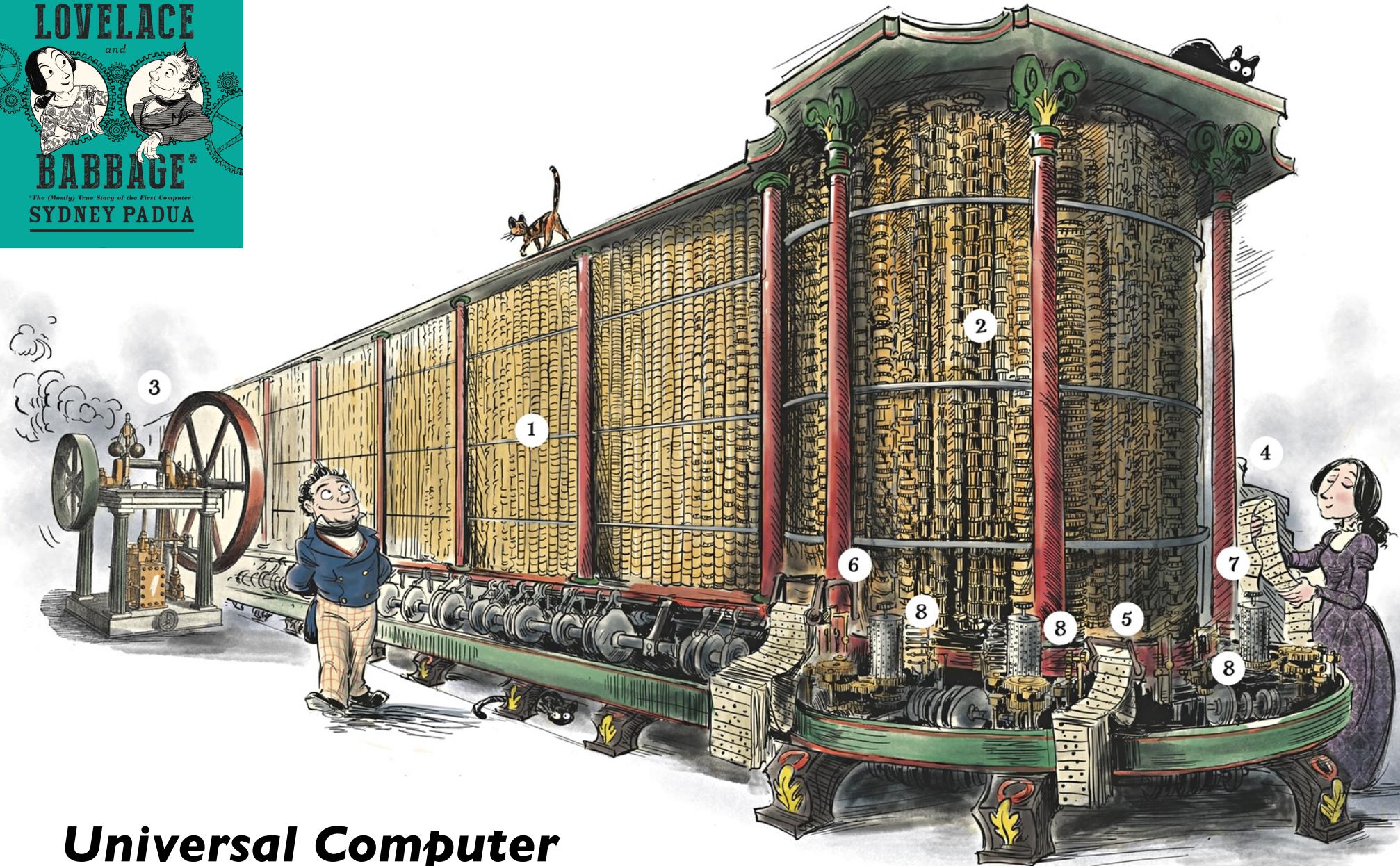


Babbage Difference Engine



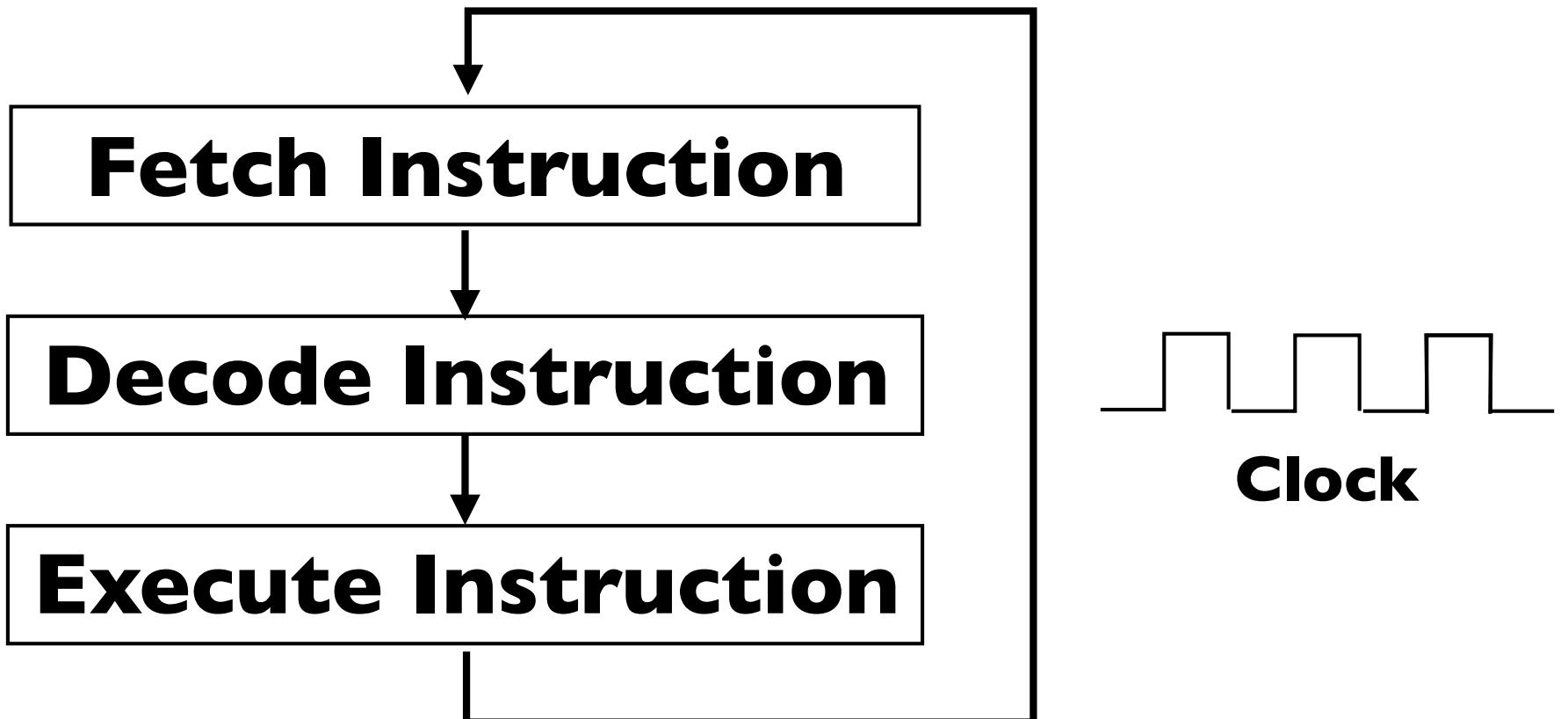


Analytical Engine



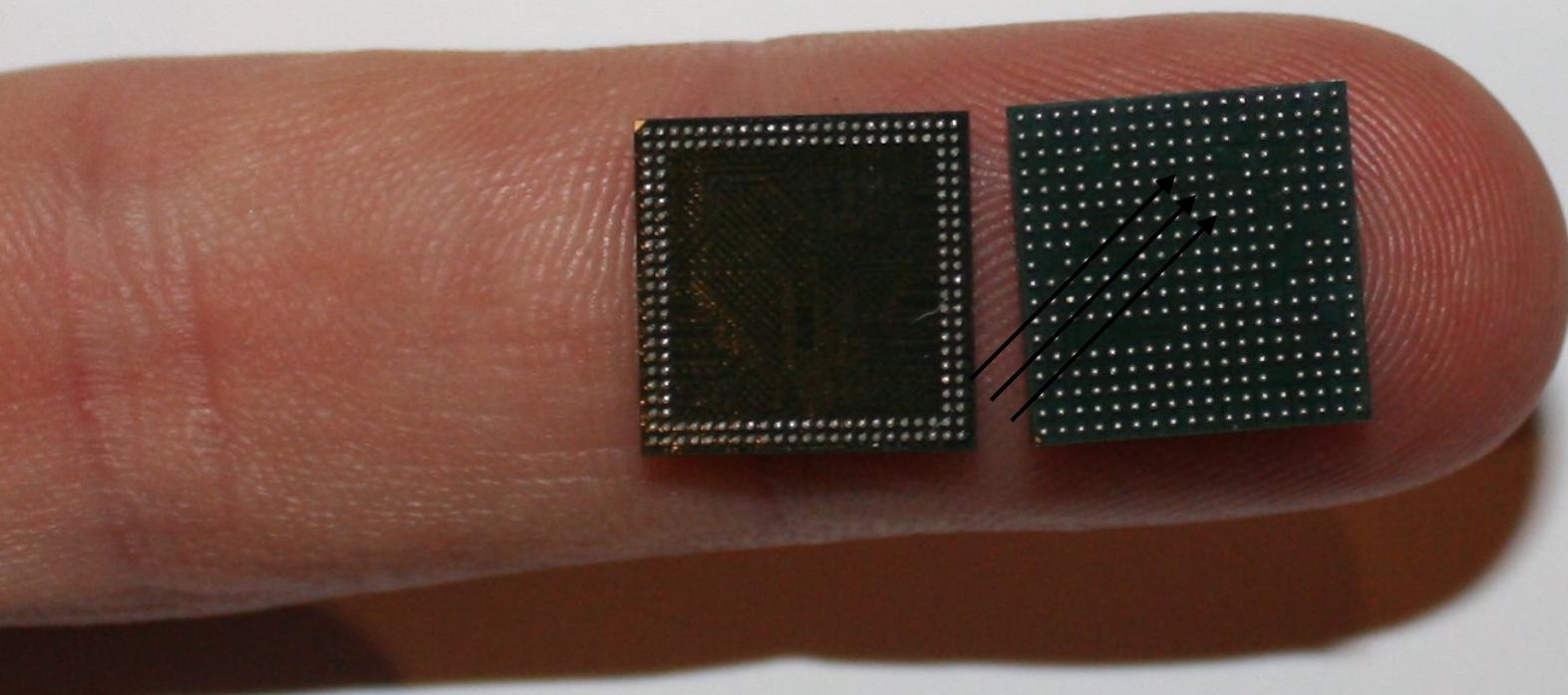
Universal Computer

Running a Program



Package on Package

Broadcom 2865 ARM Processor



Samsung 4Gb (gigabit) SDRAM

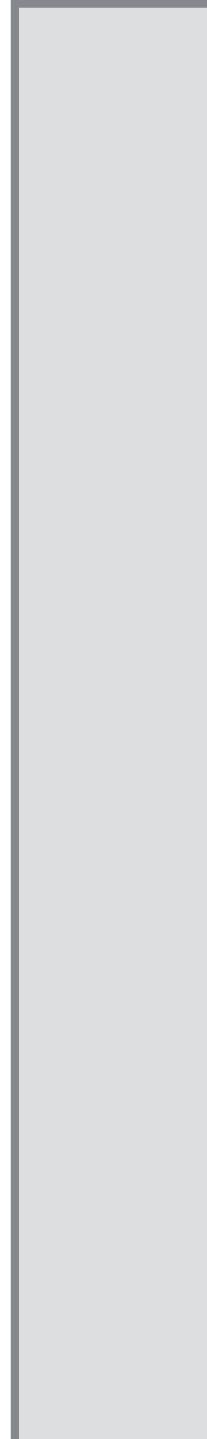
100000000_{16}

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)



Memory Map

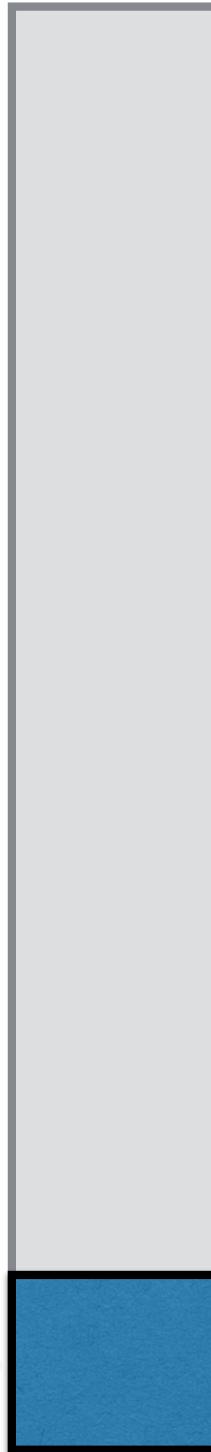
00000000_{16}

100000000_16

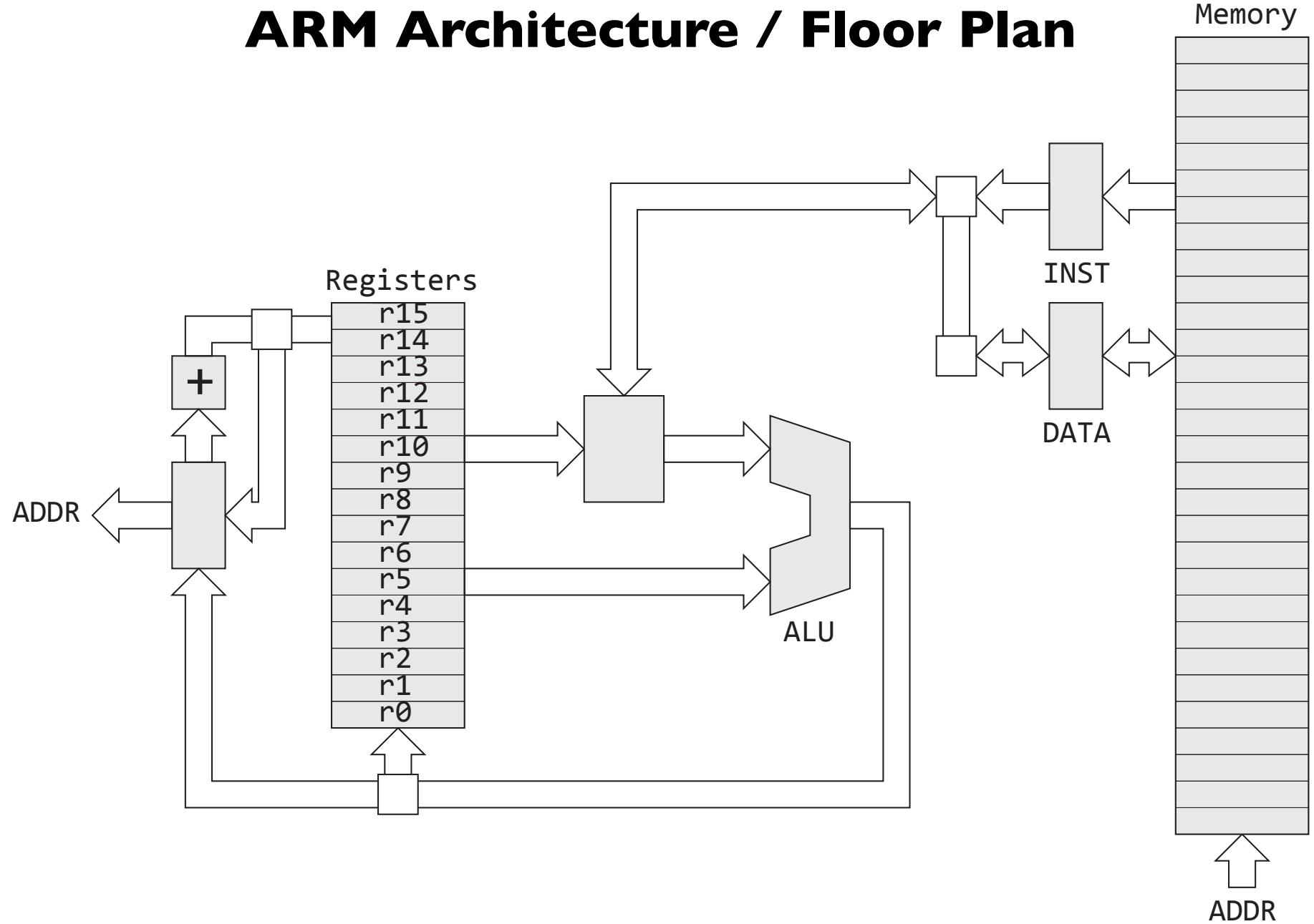
Memory Map

02000000_16

512 MB Actual Memory 

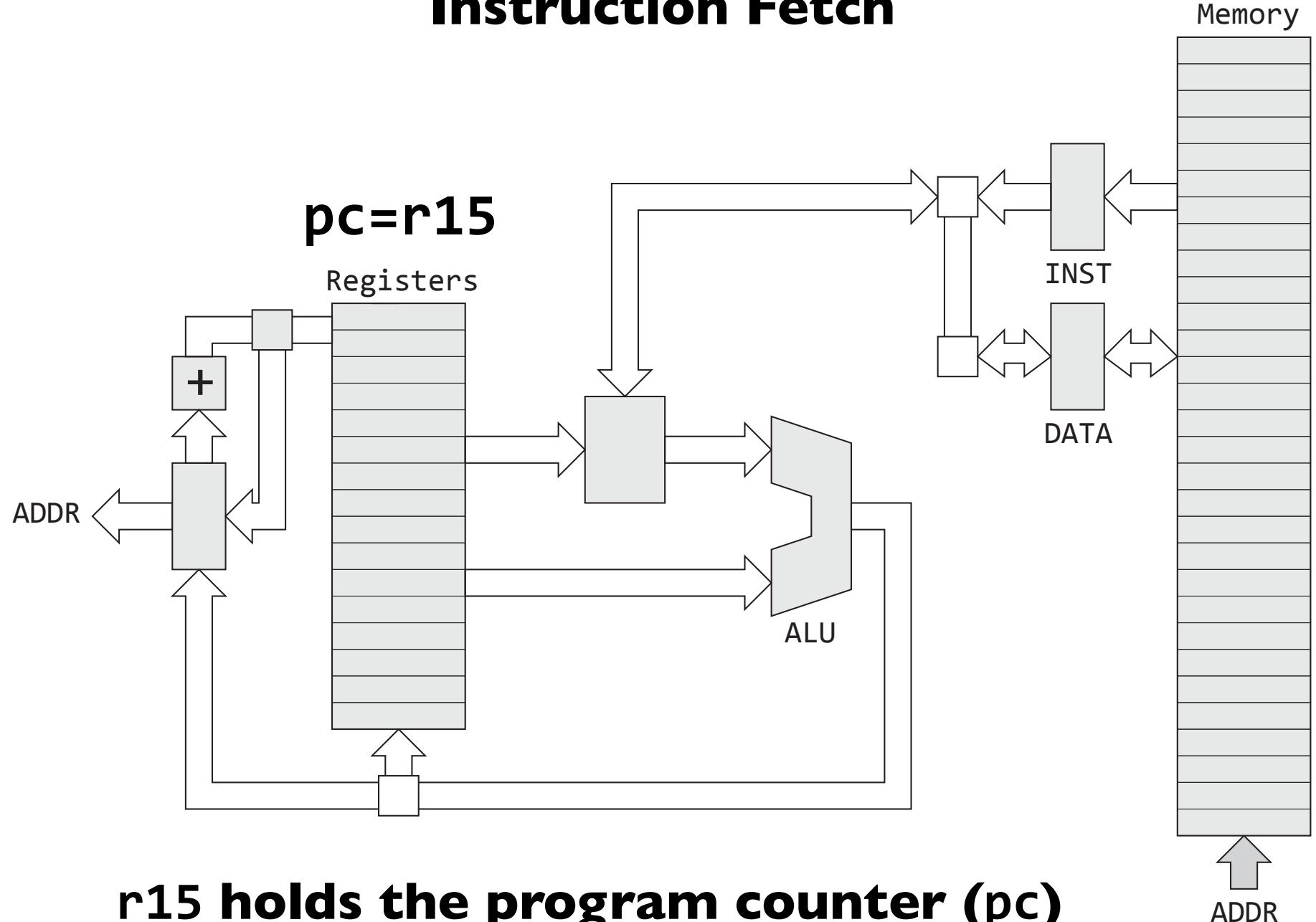


ARM Architecture / Floor Plan



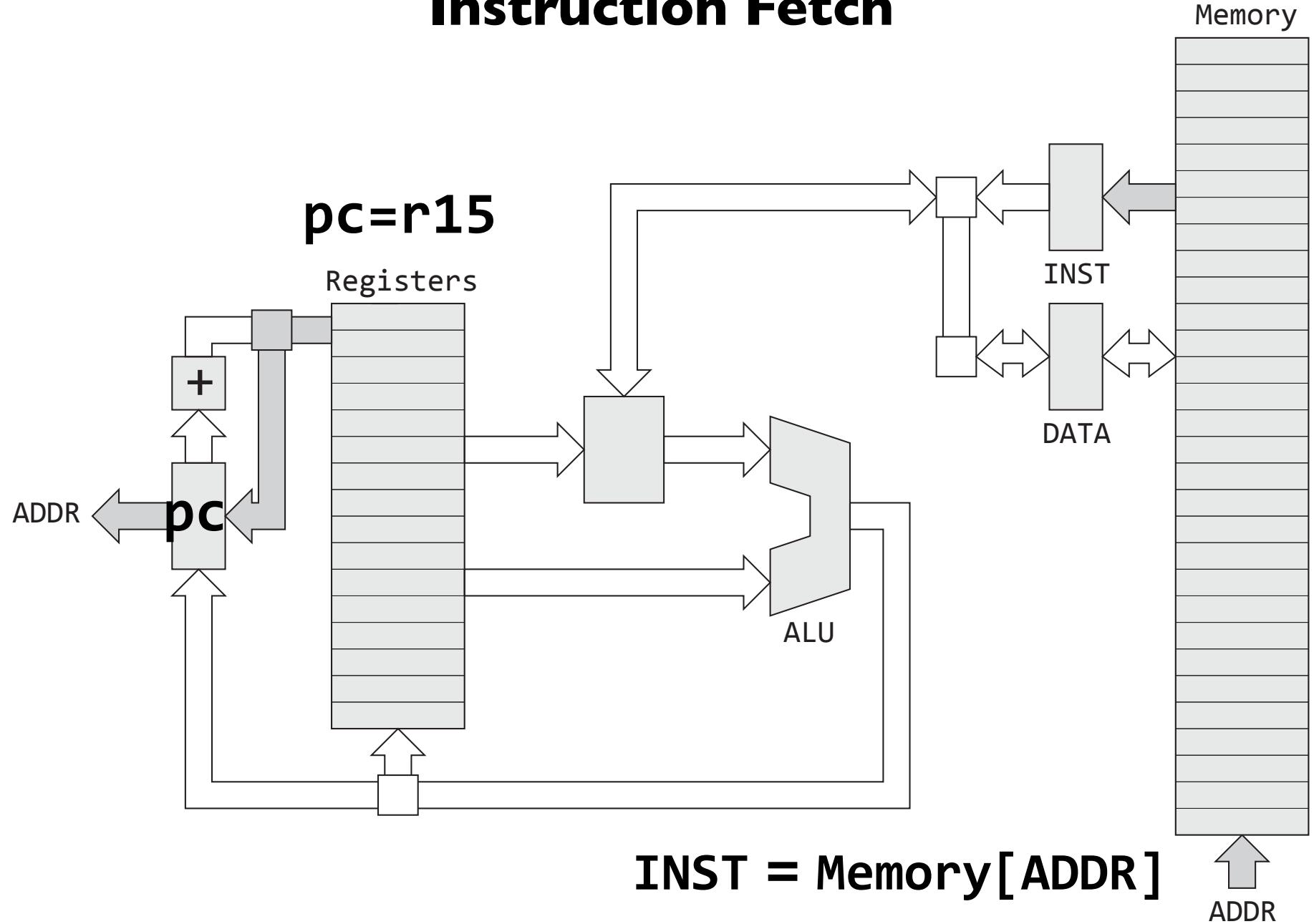
Key Fact: Everything is organized into 32-bit words

Instruction Fetch



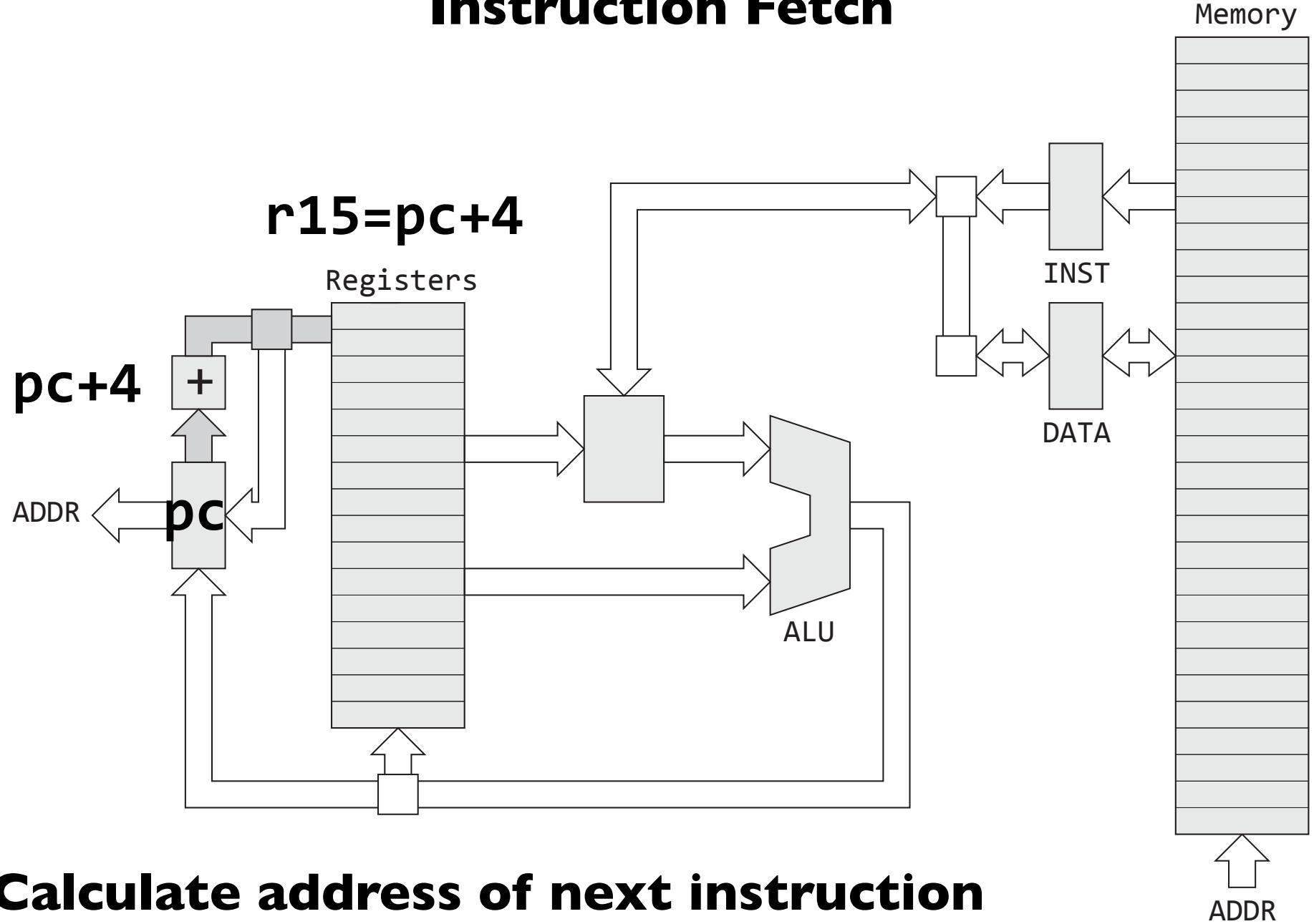
r15 holds the program counter (pc)

Instruction Fetch



Addresses and instructions are 32-bit words

Instruction Fetch



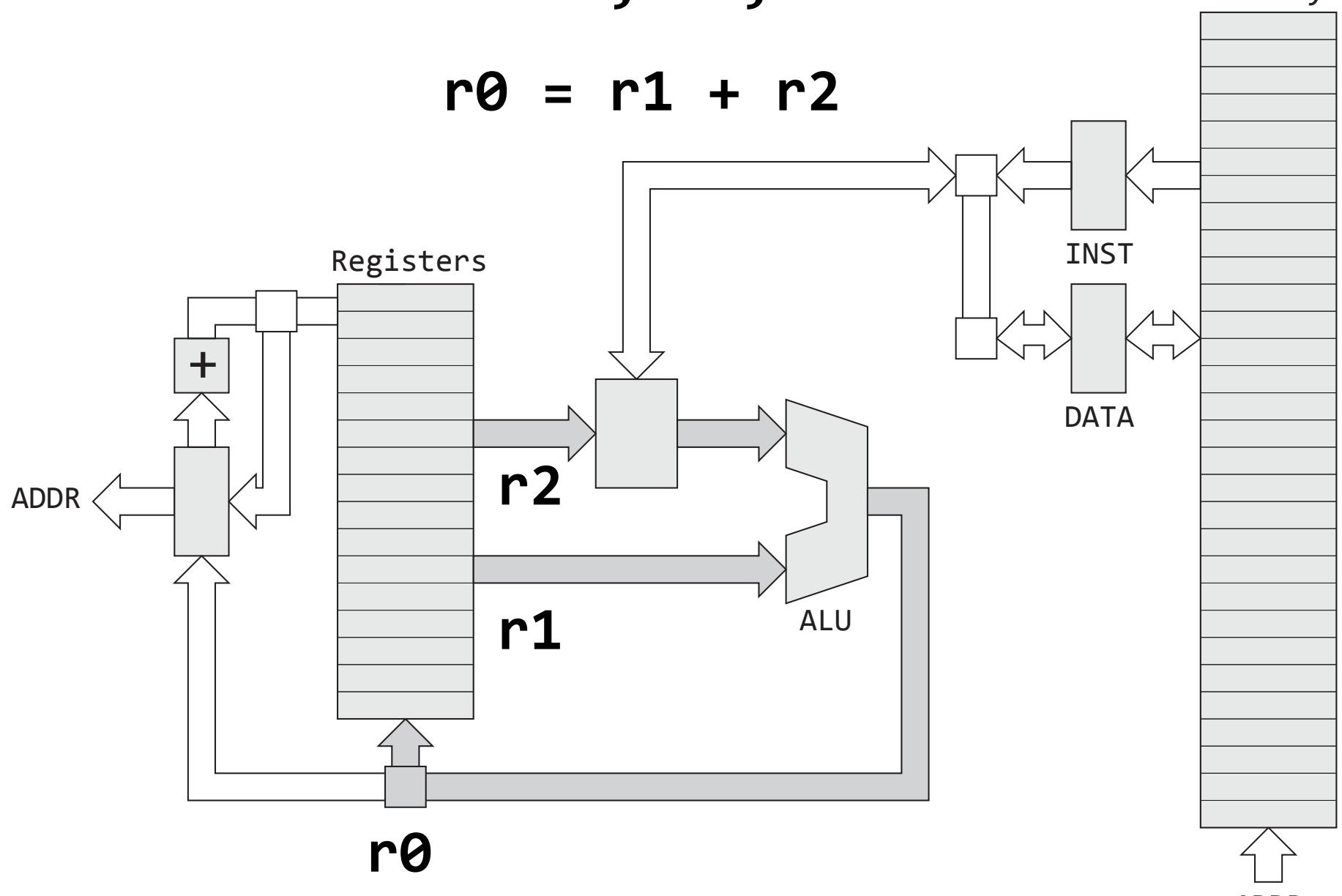
Calculate address of next instruction

Why $pc+4$?

Arithmetic-Logic Unit (ALU)

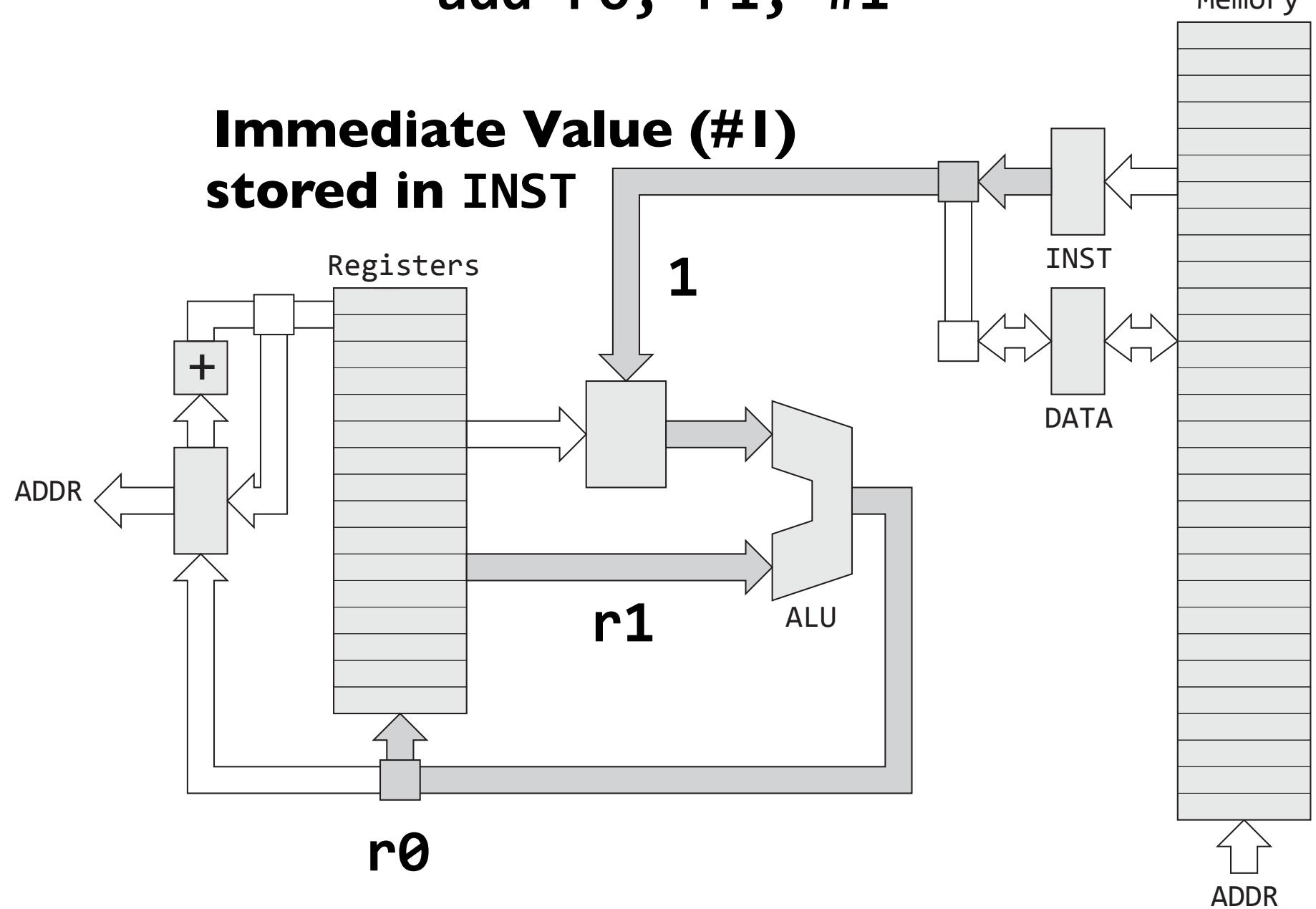
add r0, r1, r2

$$r0 = r1 + r2$$



**ALU only operates on registers
Registers are also 32-bit words**

add r0, r1, #1



Add Instruction

Meaning (defined as math or C code)

$$r_0 = r_1 + r_2$$

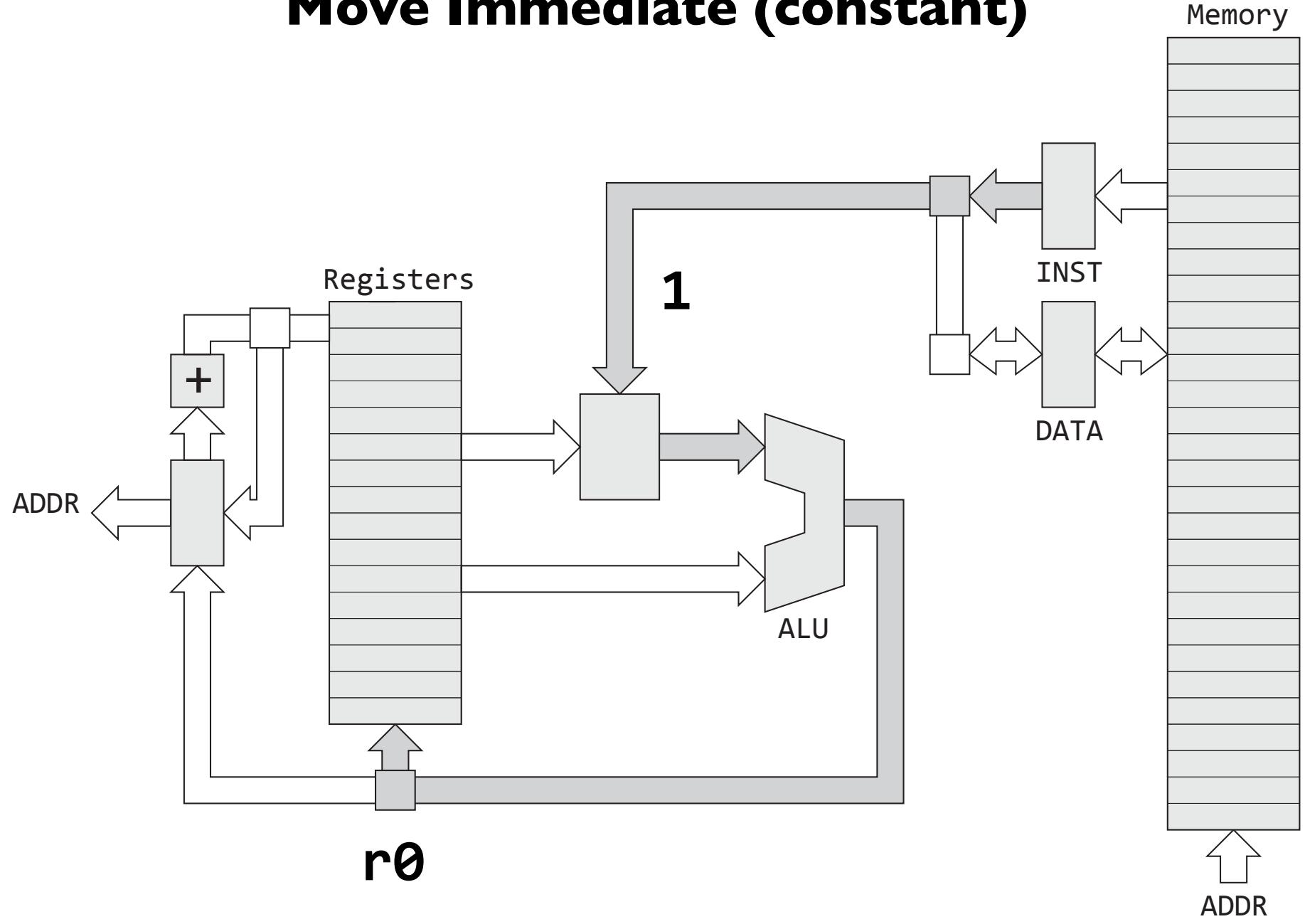
Assembly language (result is leftmost register)

`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

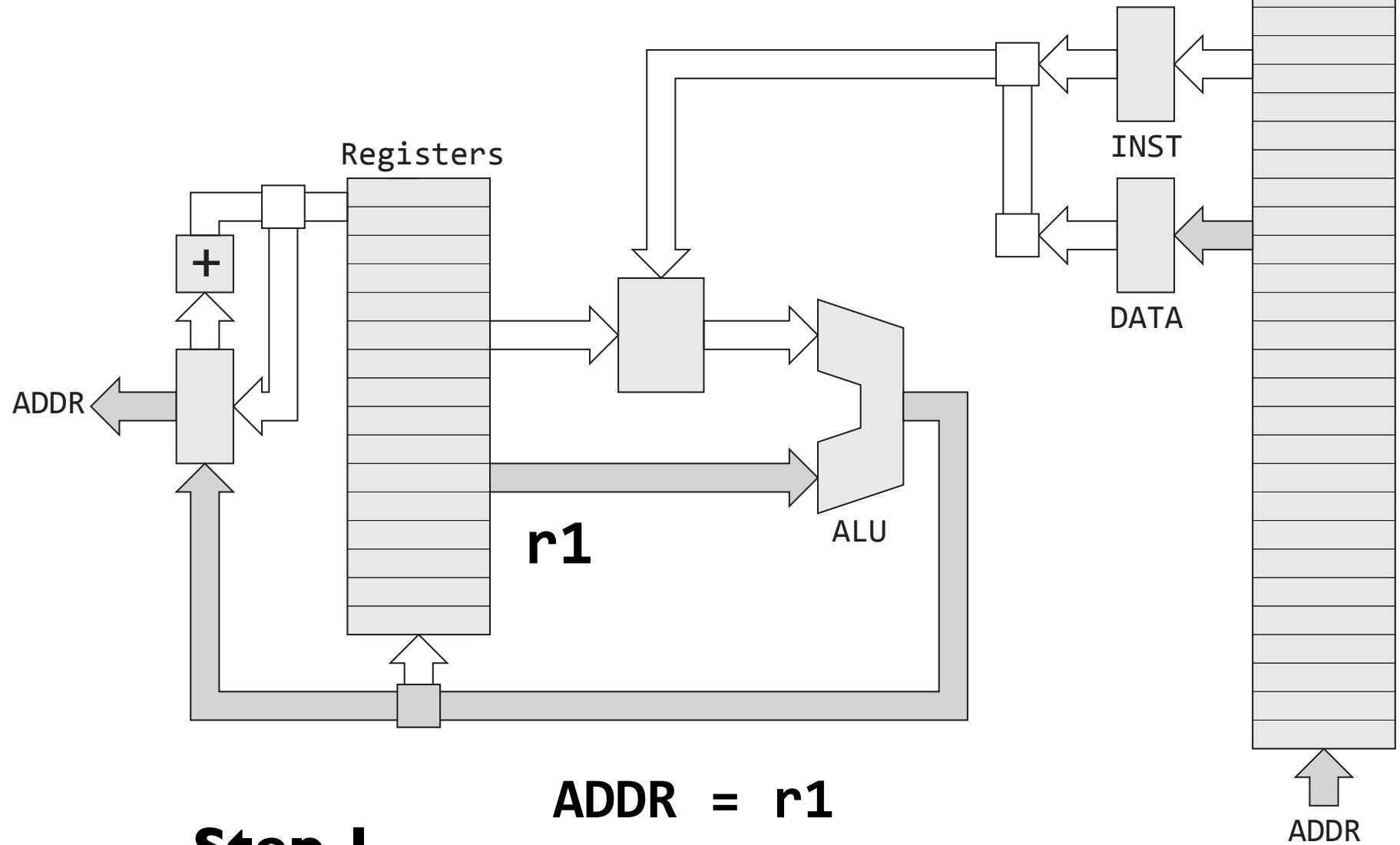
Conceptual Questions

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?

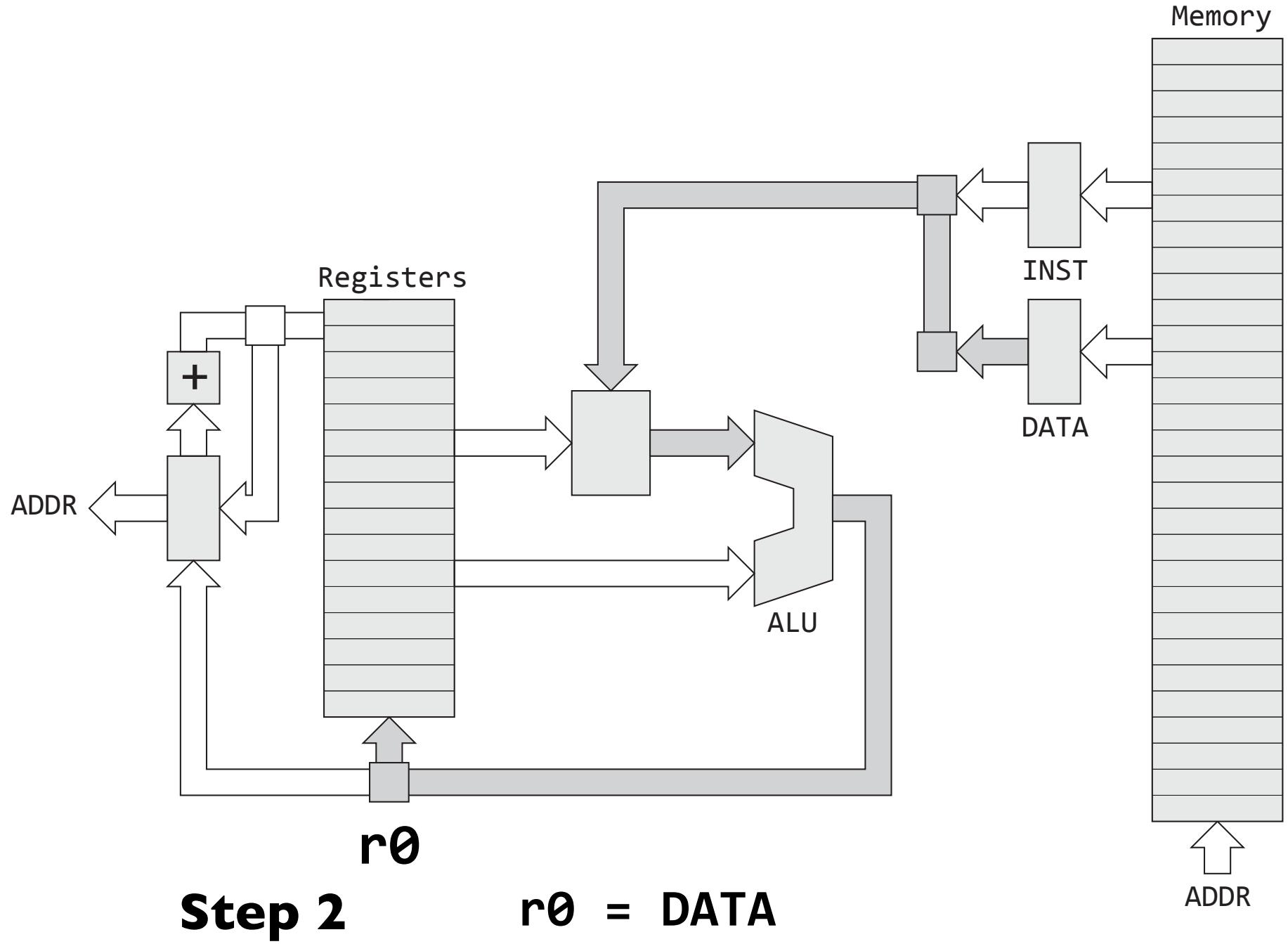
Load and Store Instructions

Load from Memory to Register (LDR)

ldr r0, [r1]

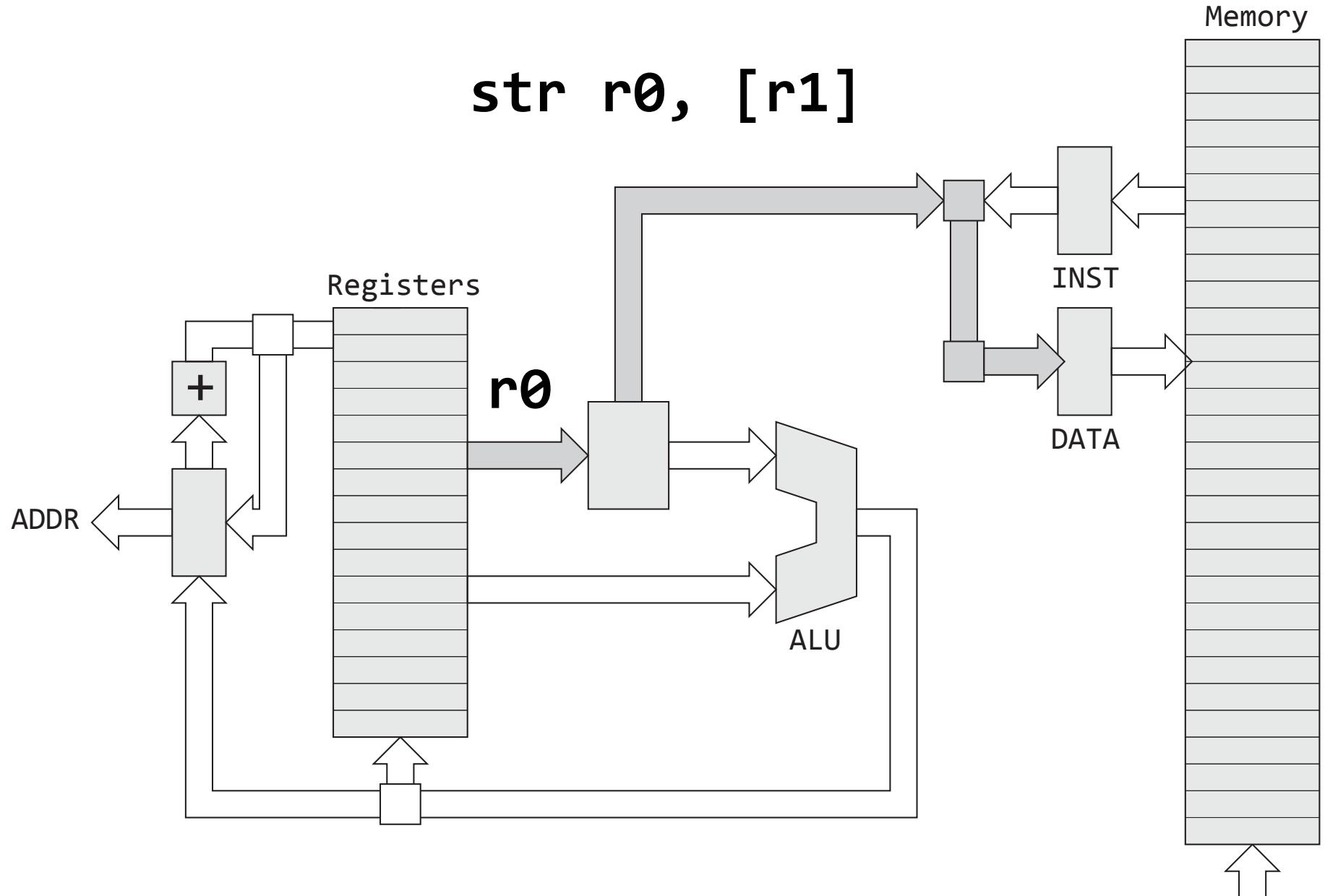


Load from Memory to Register (LDR)



Store Register in Memory (STR)

str r0, [r1]

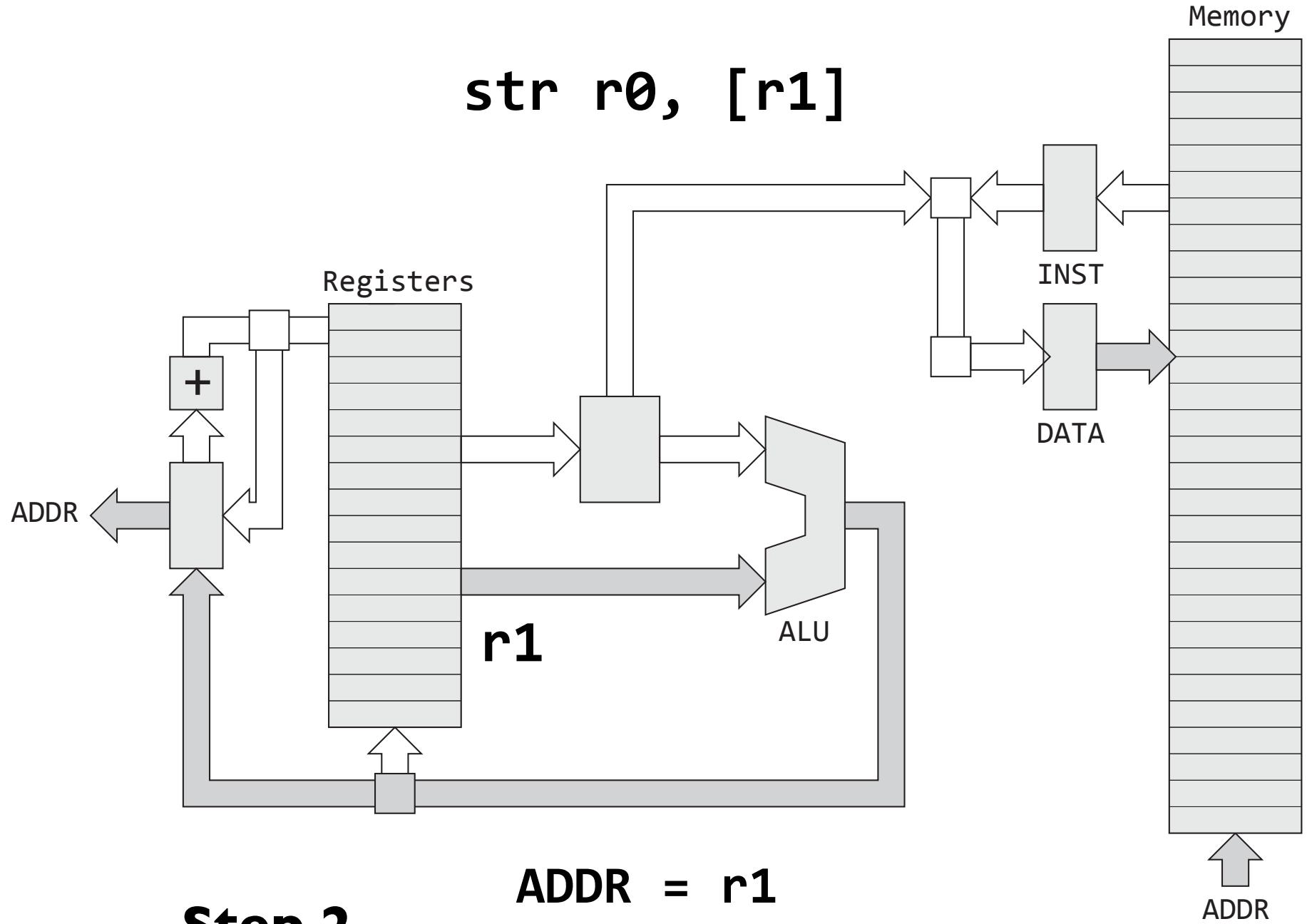


Step I

DATA = r0

ADDR

Store Register in 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

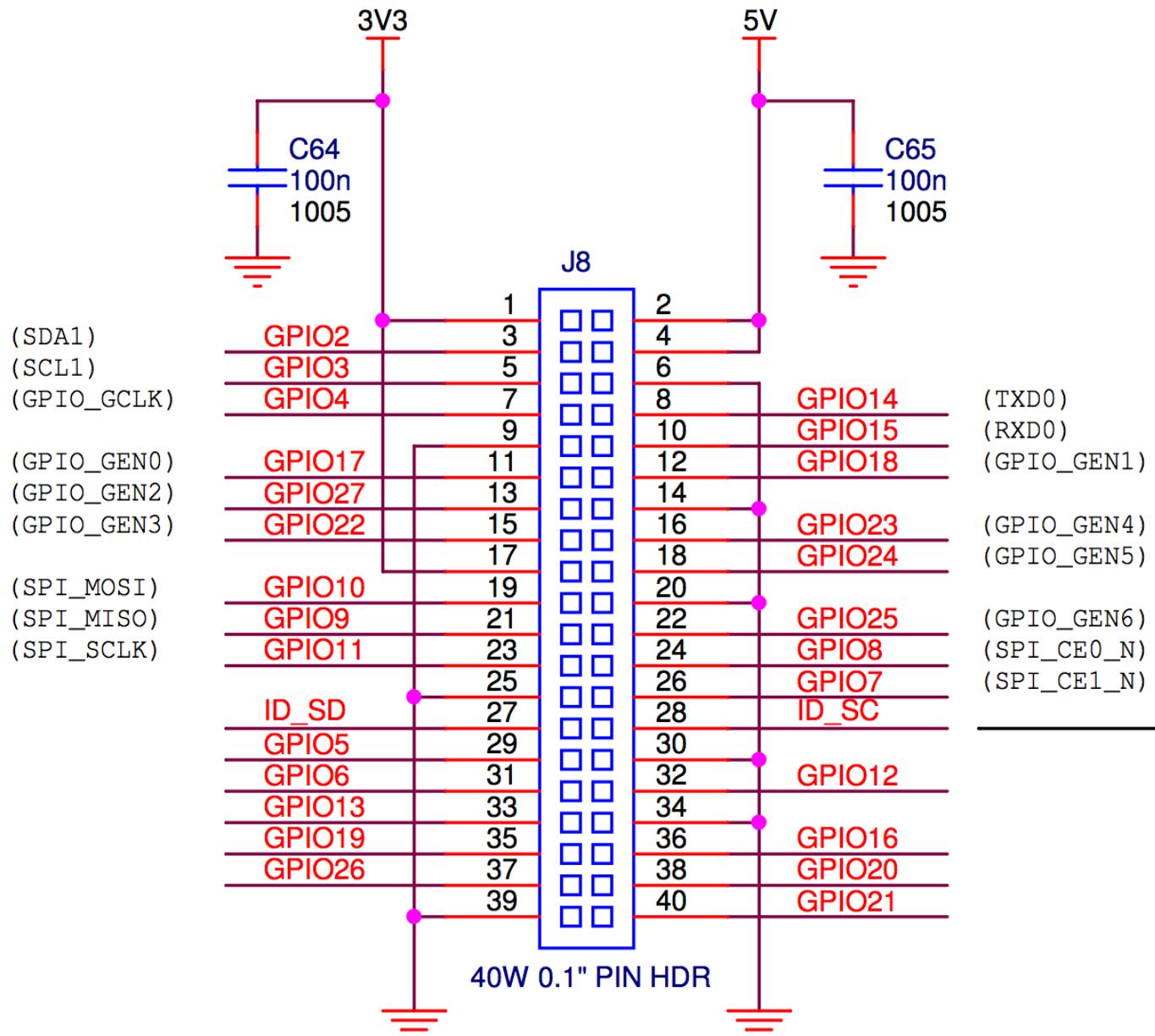
Current Instruction: 2 Total: 6

CSPR Status Bits (NZCV)

0 0 0 0

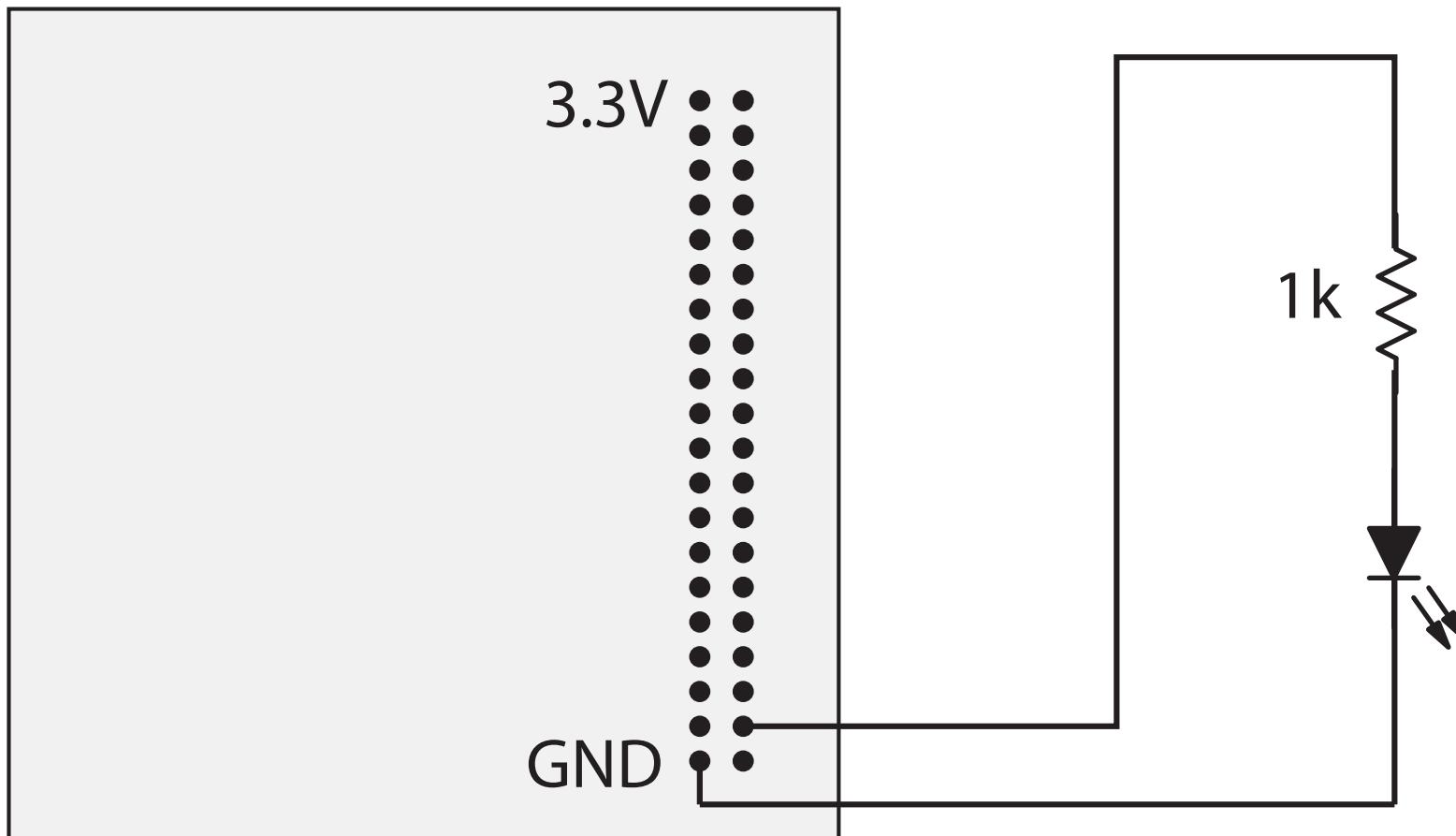
Turning on an LED

General-Purpose Input/Output (GPIO) Pins

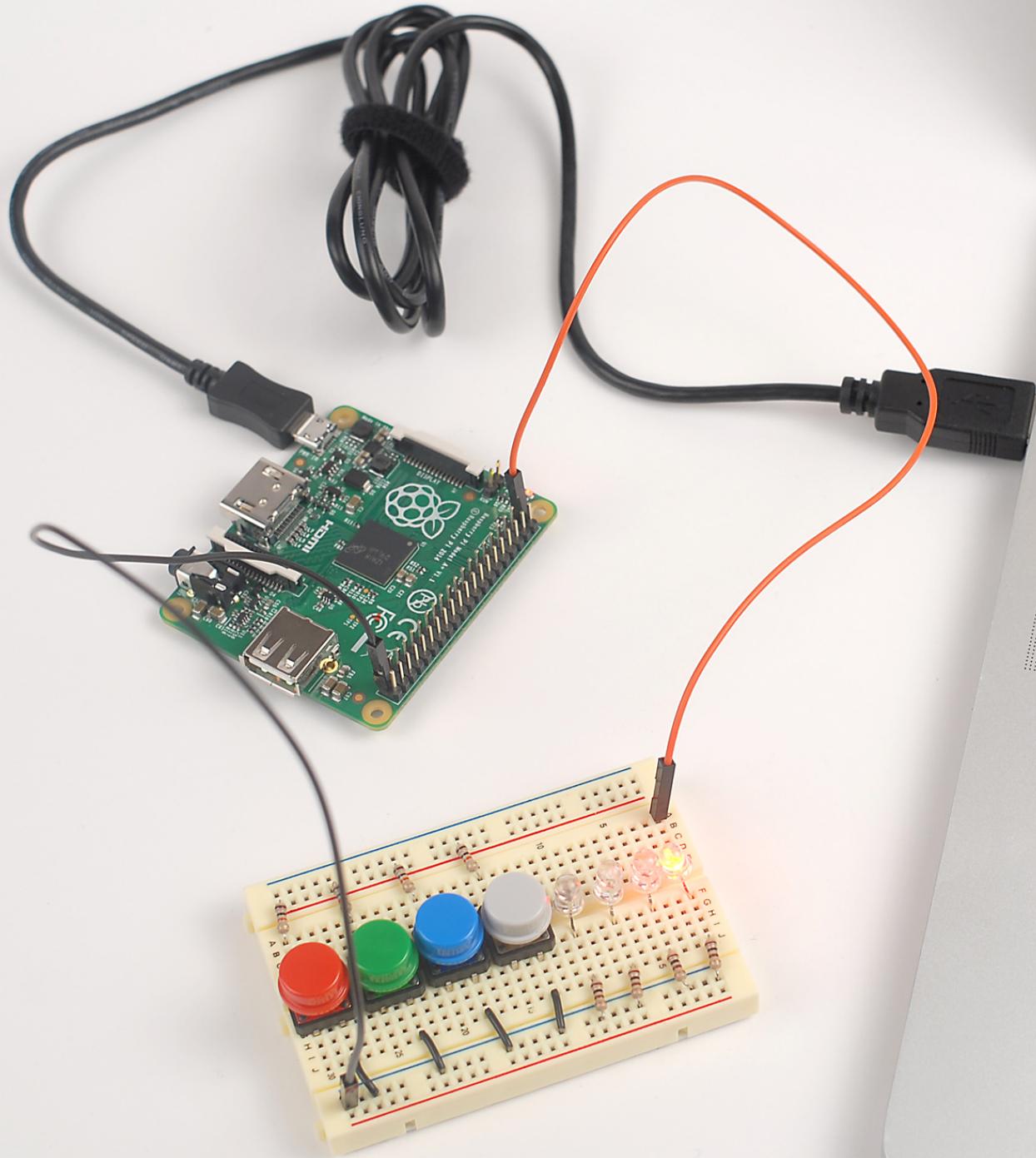


54 GPIO Pins

Connect LED to GPIO 20



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



GPIO Pins are *Peripherals*

**Peripherals are Controlled
by Special Memory Locations**

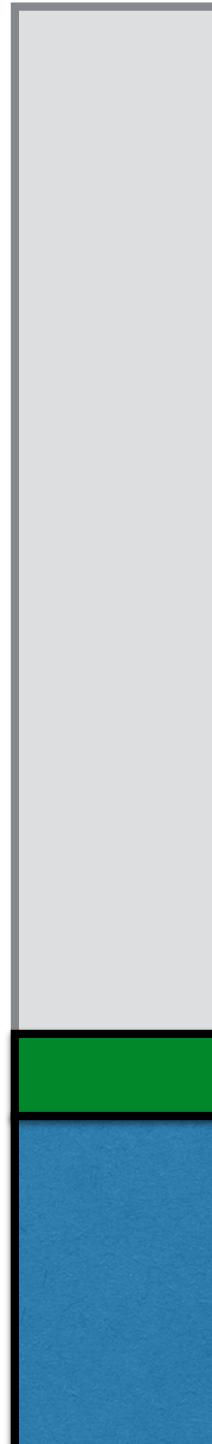
"Peripheral Registers"

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

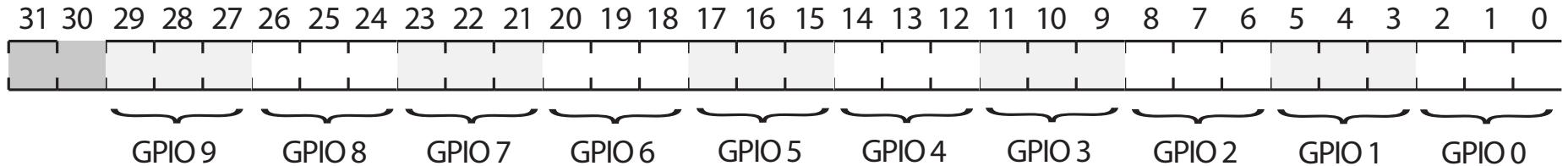
General-Purpose IO Function

GPIO Pins can be configured to be INPUT, OUTPUT, or 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 required to select function

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)

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

```
// Set GPIO20 to be an output  
  
// FSEL2 = 0x20200008  
mov r0, #0x20          // r0 = #0x00000020  
lsl r1, r0, #24        // r1 = #0x20000000  
lsl r2, r0, #16        // r2 = #0x00200000  
orr r1, r1, r2         // r1 = #0x20200000  
orr r0, r1, #0x08      // r0 = #0x20200008  
  
mov r1, #1              // 1 indicates OUTPUT  
str r1, [r0]            // store 1 to 0x20200008
```



Note this also makes GPIO 21-29 into inputs

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

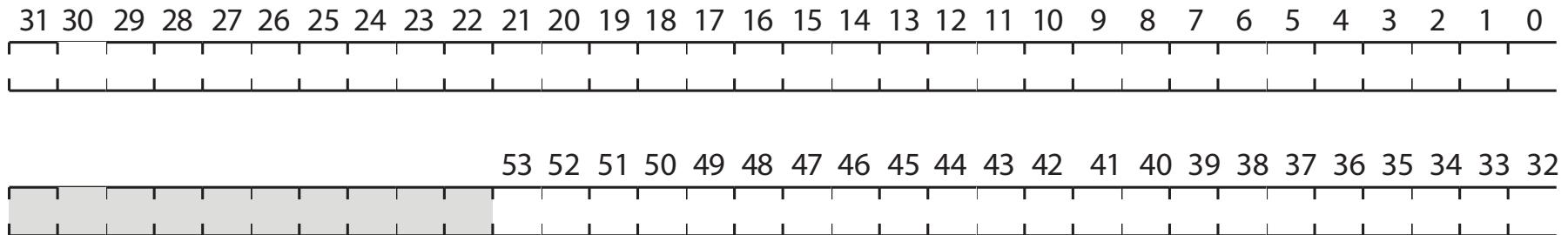
Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin <i>n</i> .	R/W	0

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

```
// Set GPIO20 output High (3.3V)
// Assumes GPIO 20 configured to be output
// FSET0 = 0x2020001c

mov r0, #0x20          // r0 = 0x00000020
lsl r1, r0, #24        // r1 = 0x20000000
lsl r2, r0, #16        // r2 = 0x00200000
orr r1, r1, r2         // r1 = 0x20200000
orr r0, r1, #0x1c      // r0 = 0x2020001c

mov r1, #1              // r1 = 0x00000001
lsl r1, r1, #20        // r1 = 0x00100000
str r1, [r0]            // store r1 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/
```

```
BOOT2020  Macintosh HD
```

```
# Copy to SD card
```

```
% cp on.bin /Volumes/BOOT2020/
```

```
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!!  
#
```



Assignment 0

- Join forum piazza.com/stanford/winter2020/cs107e
- Read and understand our guides on basic topics (electricity, numbers, unix). Practice unix command line
- Checkout the course directory from gitbub
- Checkout your assignment directory and answer the questionnaire, check-in to submit assignment
- Setup your development environment before lab

Key Concepts

- Bits are bits; fundamental bitwise operations
- Memory addresses refer to bytes (8-bits), words are 4 bytes
- Memory stores both instructions and data
- Computers repeatedly fetch, decode, and execute instructions
- Different types of ARM instructions: ALU, Loads and Stores, Branches
- General purpose IO (GPIO), peripheral registers, and MMIO

Further Reading

If you want to learn more about high-level computer organization and instructions, Chapter 2 of Computer Organization and Design: The Hardware/Software Interface (Patterson and Hennessy) is an excellent place to start.

