ARM

Assembly Language and Machine Code

Goal: Blink an LED

Some Context

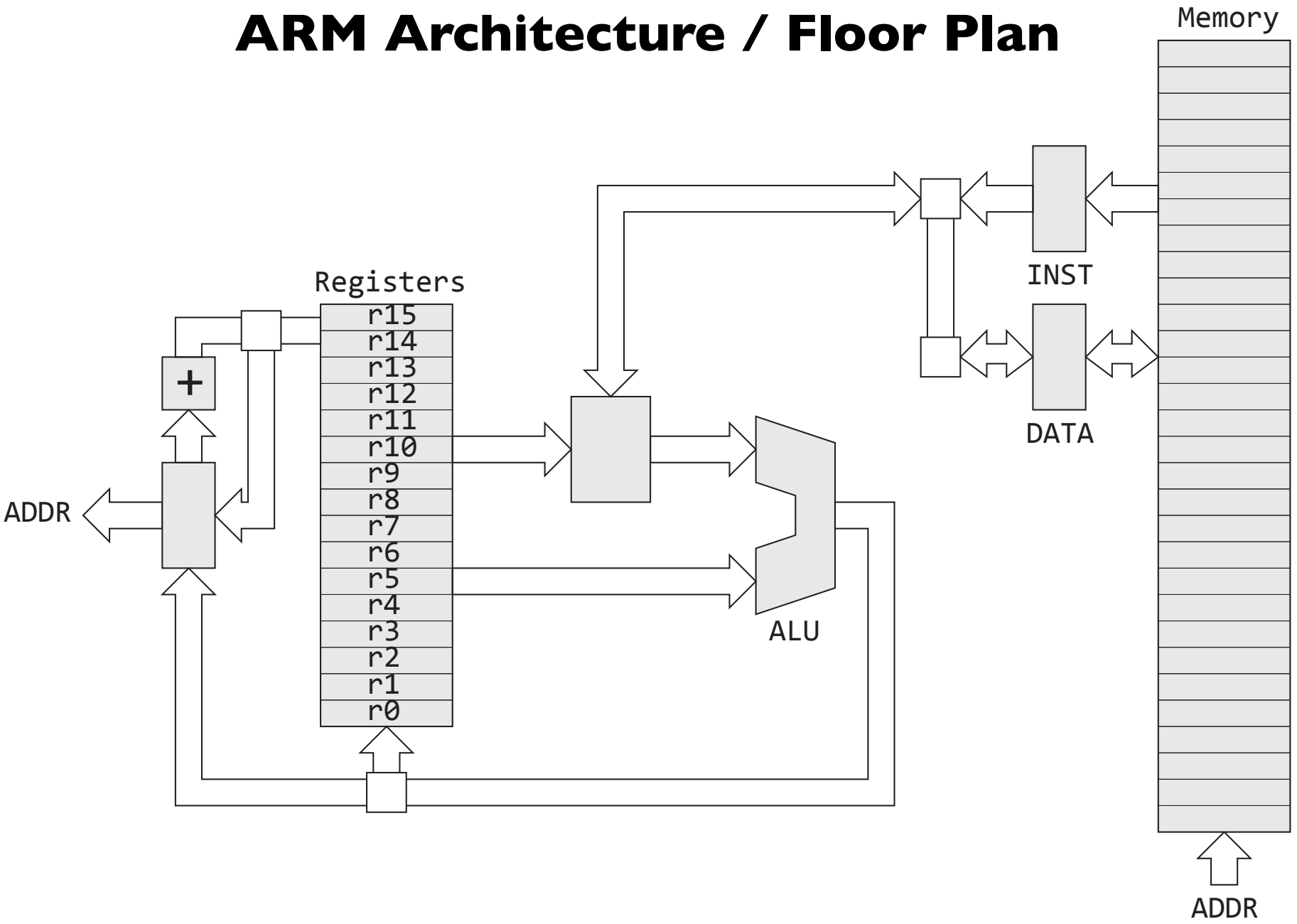
You need to understand how processors represent and execute instructions

Instruction set architecture often easier to understand by looking at the bits. Encoding instructions in 32-bits requires trade-offs, careful design

Only write assembly when it is needed. Reading assembly more important than writing assembly
Allows you to see what the compiler and processor are actually doing

Normally write code in C (Starting next lecture)

ARM Architecture / Floor Plan



Key Fact: Everything is organized into 32-bit words

Key Concepts

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

Memory stores both instructions and data

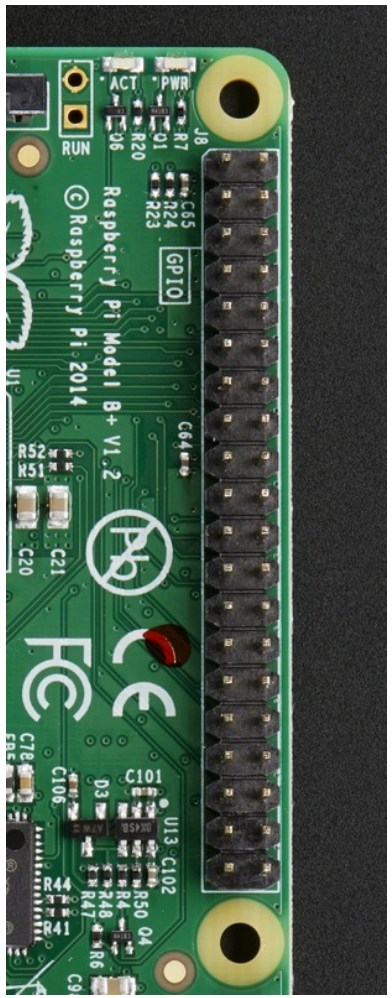
Computers repeatedly fetch, decode, and execute instructions

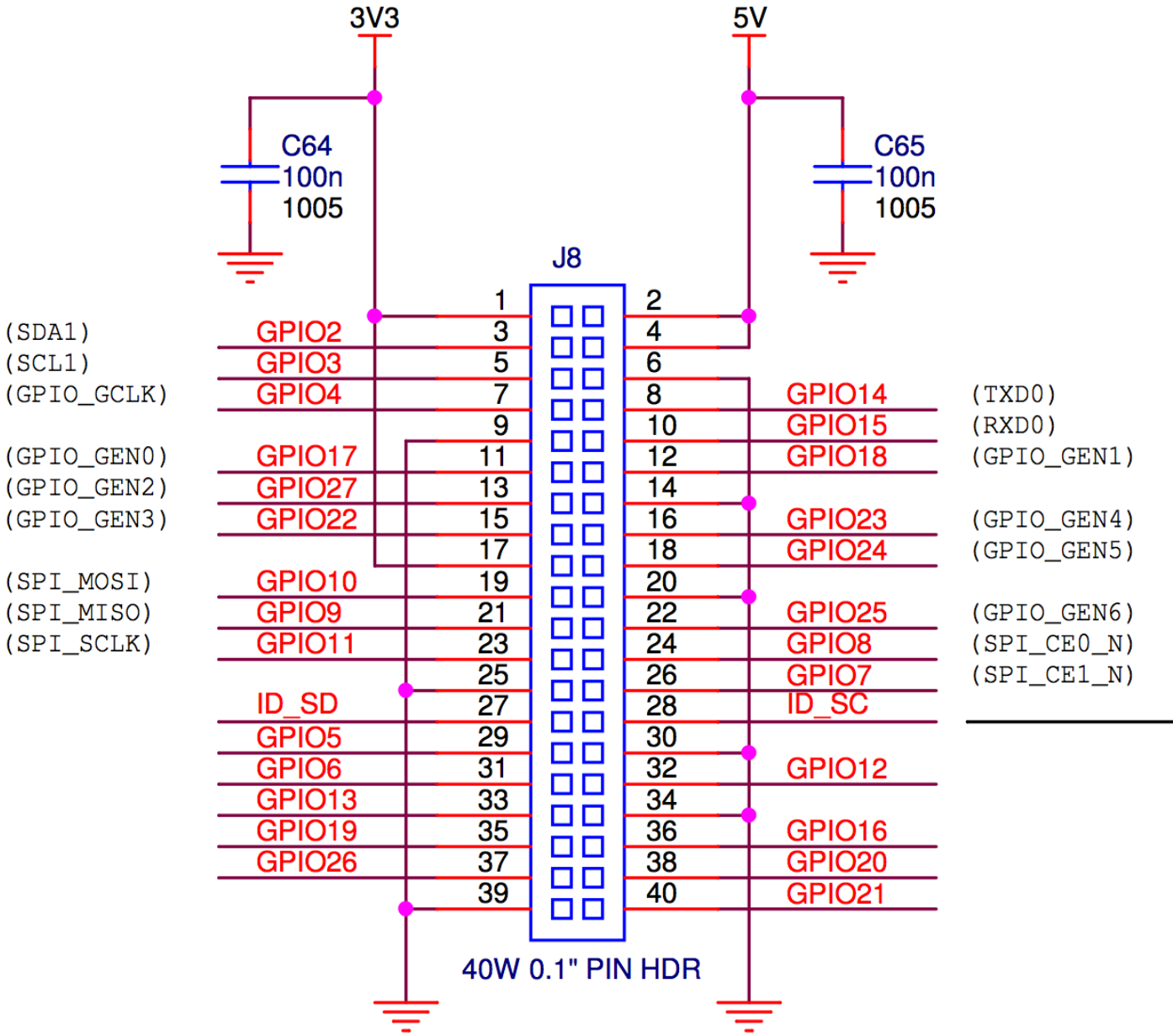
Types of ARM instructions: ALU, Loads and Stores, Branches

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

Bits are bits; fundamental bitwise operations

General-Purpose Input/Output (GPIO) Pins





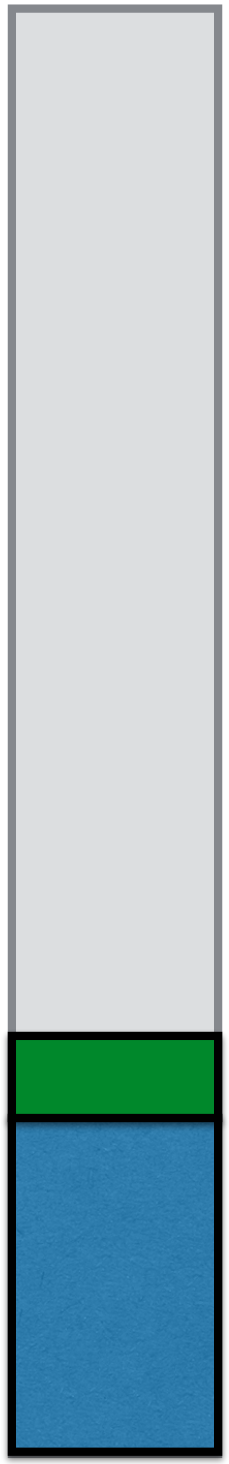
54 GPIO Pins

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

Using GPIO Pins as Output

GPIO function select registers (GPFSELn, 0x20200000)

- Configure pin function (input, output, special)
- 3b/pin => 6 registers (each register controls 10 pins)

GPIO function select registers (GPSETn, 2020001c)

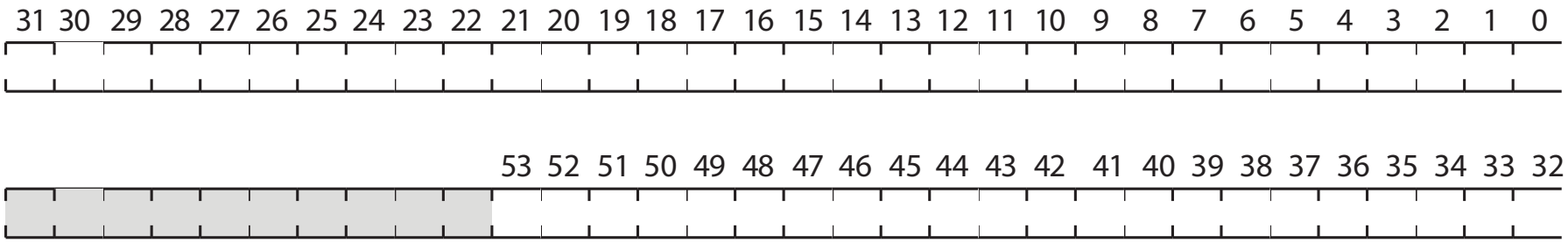
- Set the value of a pin to 1 (high voltage)
- 1b/pin => 2 registers

GPIO function select registers (0x2020002b)

- Reset the value of a pin to 0 (low voltage)
- 1b/pin => 2 registers

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

Turning on an LED

```
// 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

Back to the ARM Instruction Set Architecture

3 Types of Instructions

1. Data processing instructions

2. Loads from and stores to memory

3. (Conditional) branches to new program locations (non sequential)

Data Processing Instructions and Machine Code

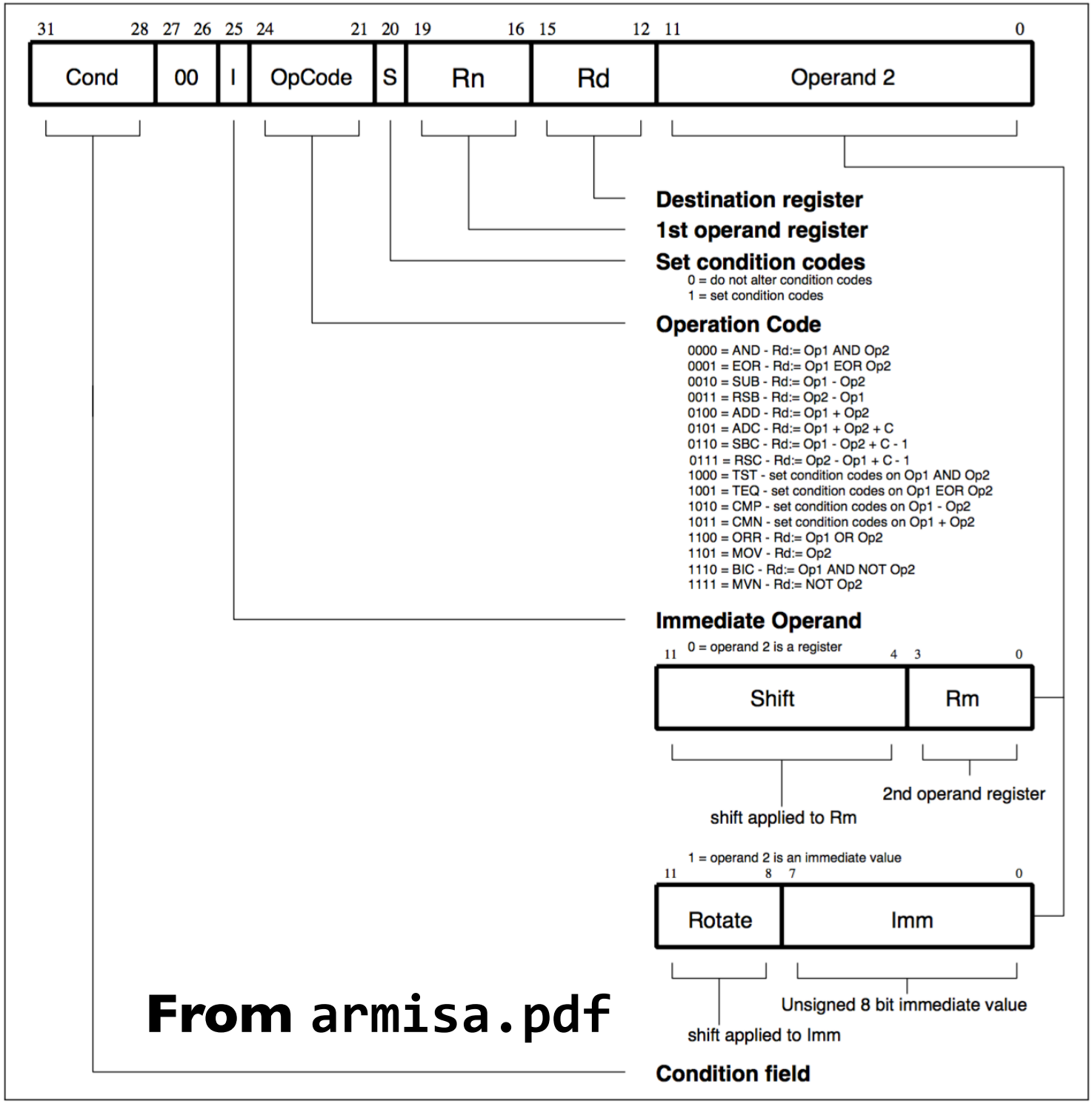
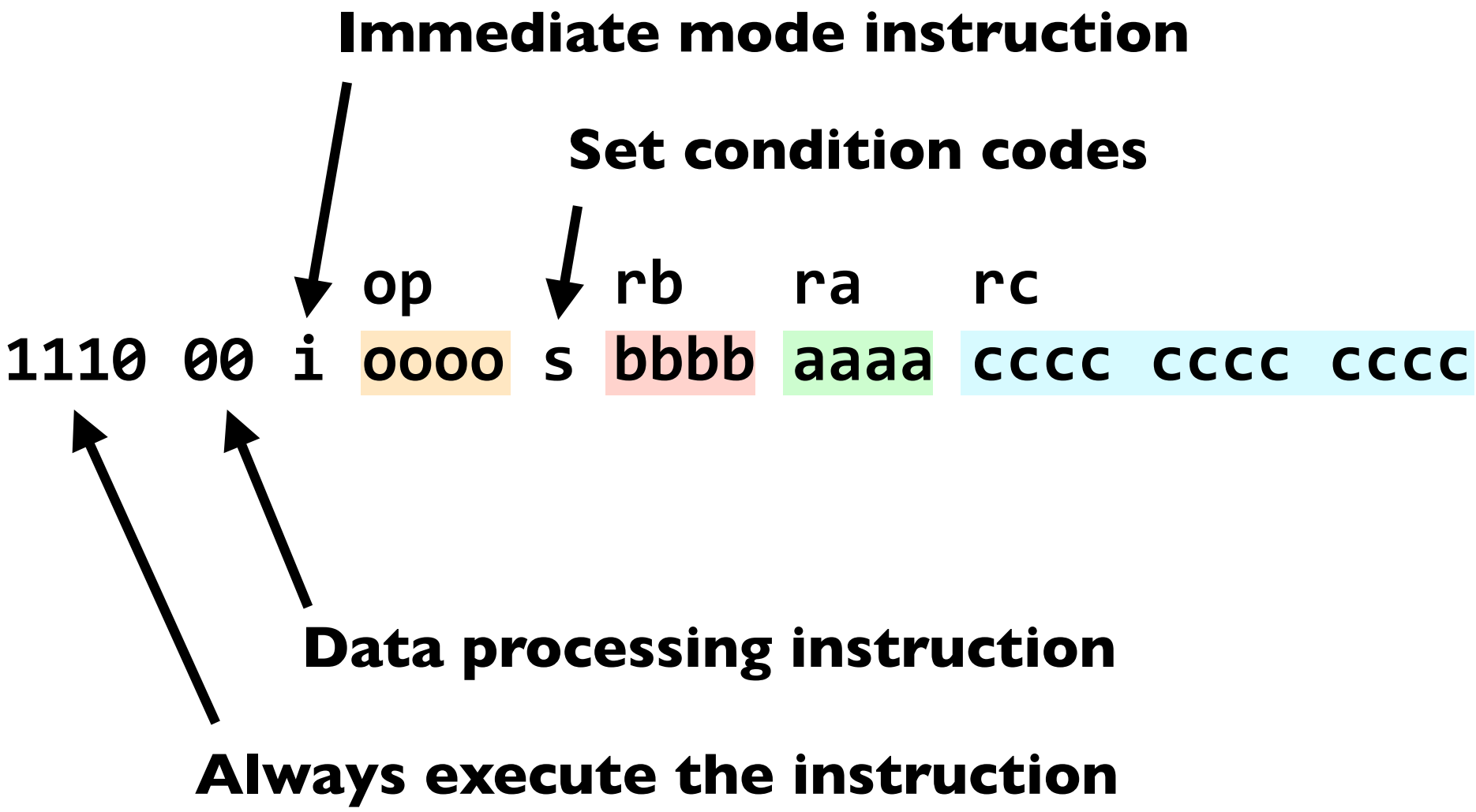


Figure 4-4: Data processing instructions

```
# data processing instruction
#
# ra = rb op rc
```

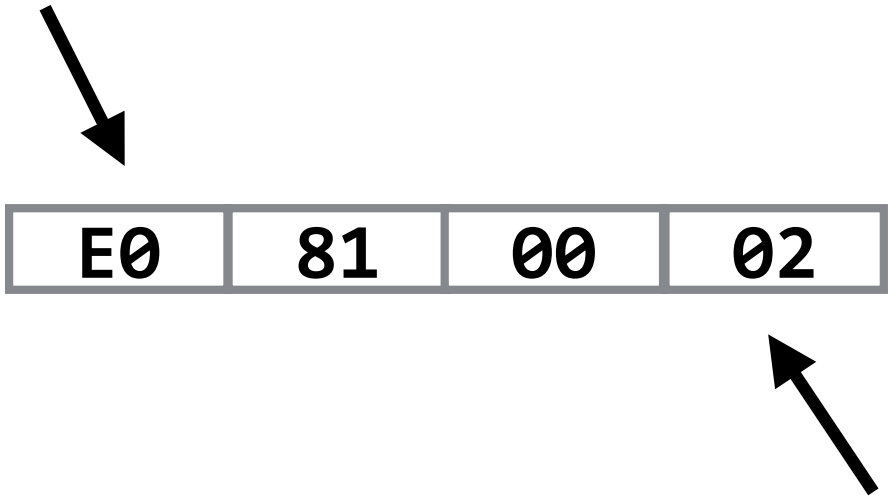


Assembly	Code	Operations
AND	0000	ra=rb&rc
EOR (XOR)	0001	ra=rb^rc
SUB	0010	ra=rb-rc
RSB	0011	ra=rc-rb
ADD	0100	ra=rb+rc
ADC	0101	ra=rb+rc+CARRY
SBC	0110	ra=rb-rc+(1-CARRY)
RSC	0111	ra=rc-rb+(1-CARRY)
TST	1000	rb&rc (ra not set)
TEQ	1001	rb^rc (ra not set)
CMP	1010	rb-rc (ra not set)
CMN	1011	rb+rc (ra not set)
ORR (OR)	1100	ra=rb|rc
MOV	1101	ra=rc
BIC	1110	ra=rb&~rc
MVN	1111	ra=~rc

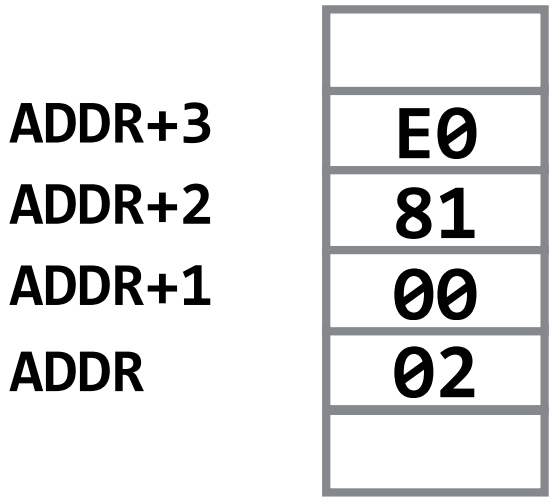
```
# data processing instruction
#  ra = rb op rc
#

           op      rb   ra   rc
1110 00 i oooo s bbbb aaaa cccc cccc cccc

# i=0, s=0
           add     r1   r0   r2
1110 00 0 0100 0 0001 0000 0000 0000 0010
```

```
# data processing instruction
#  ra = rb op rc
#

           op      rb   ra   rc
1110 00 i oooo s bbbb aaaa cccc cccc cccc

# i=0, s=0
           add     r1   r0   r2
1110 00 0 0100 0 0001 0000 0000 0000 0010

1110 0000 1000 0001 0000 0000 0000 0010
   E    0    8    1    0    0    0    2
```

most-significant-byte (MSB)



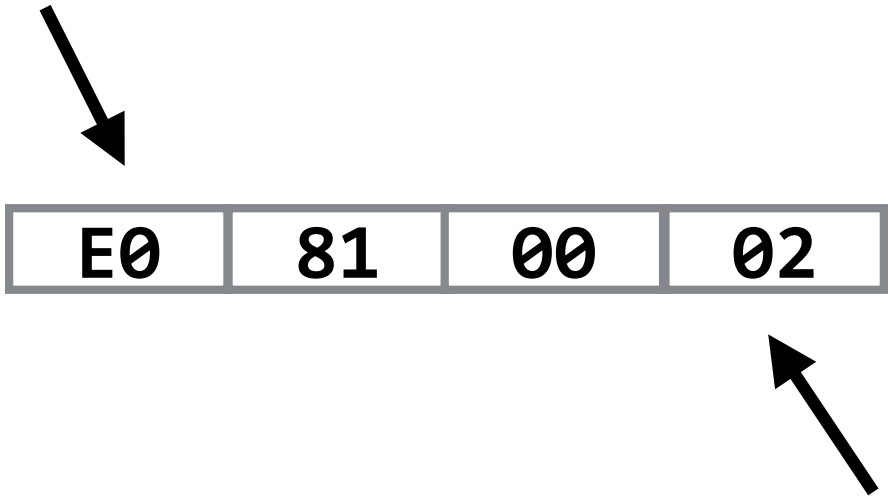
least-significant-byte (LSB)



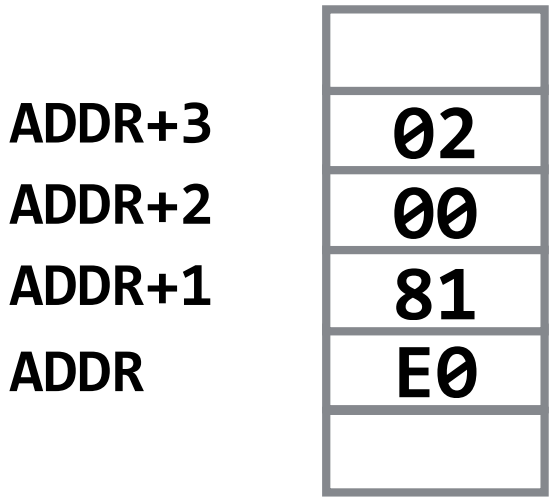
little-endian
(LSB first)

ARM uses little-endian

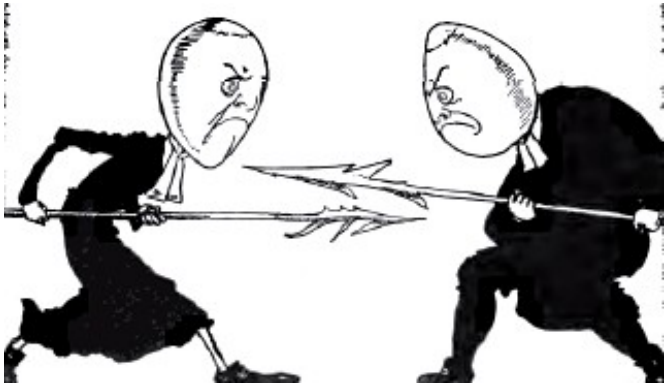
most-significant-byte (MSB)



least-significant-byte (LSB)



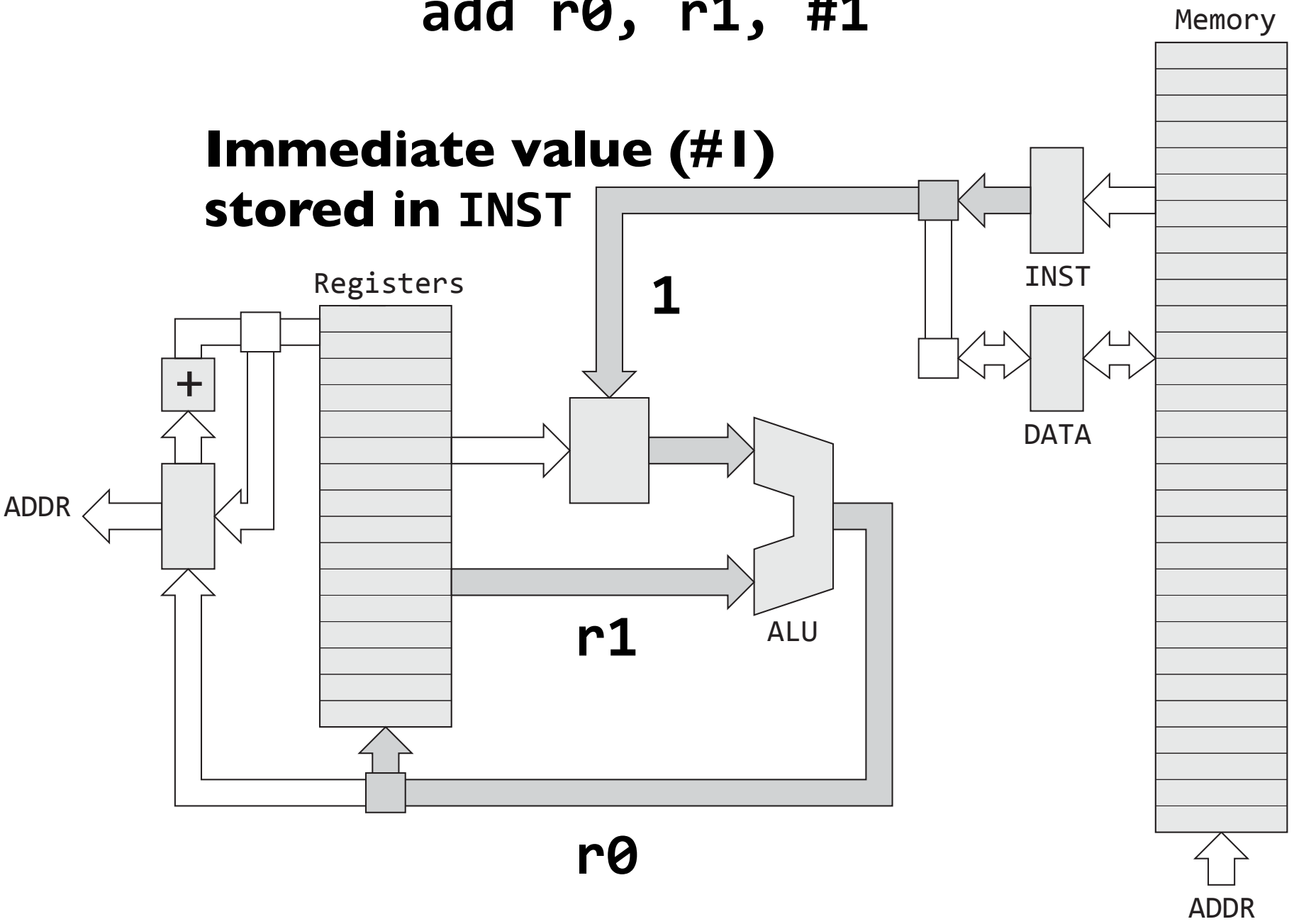
big-endian
(MSB first)



The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's Gulliver s Travels. The inhabitants of Lilliput, who are well known for being rather small, are, in addition, constrained by law to break their eggs only at the little end. When this law is imposed, those of their fellow citizens who prefer to break their eggs at the big end take exception to the new rule and civil war breaks out. The big-endians eventually take refuge on a nearby island, which is the kingdom of Blefuscu. The civil war results in many casualties.

Read: Holy Wars and a Plea For Peace, D. Cohen

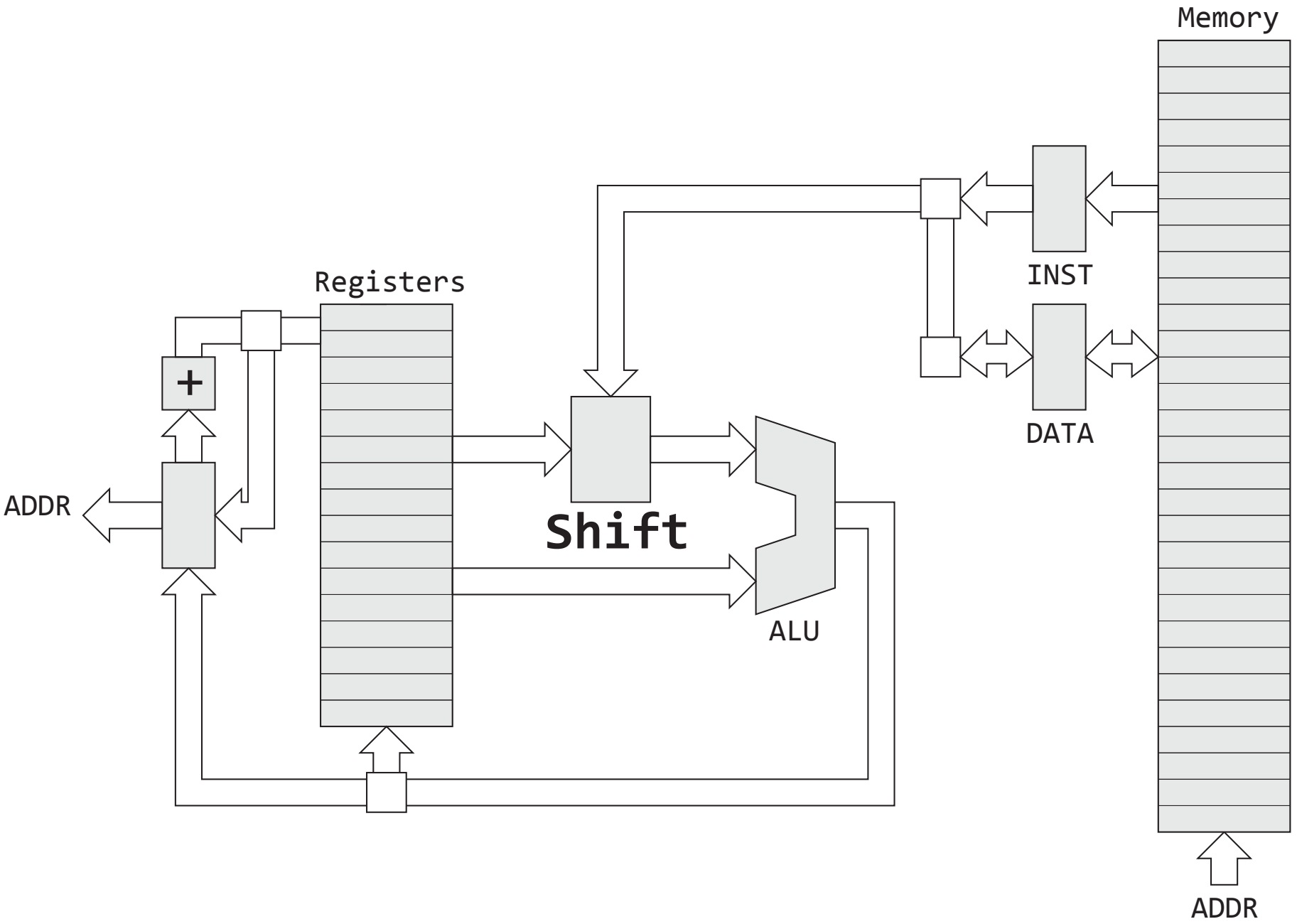
add r0, r1, #1


Memory
Immediate value (#1)
stored in INST
Registers
1
INST
DATA
+
ADDR
r1
ALU
r0
ADDR

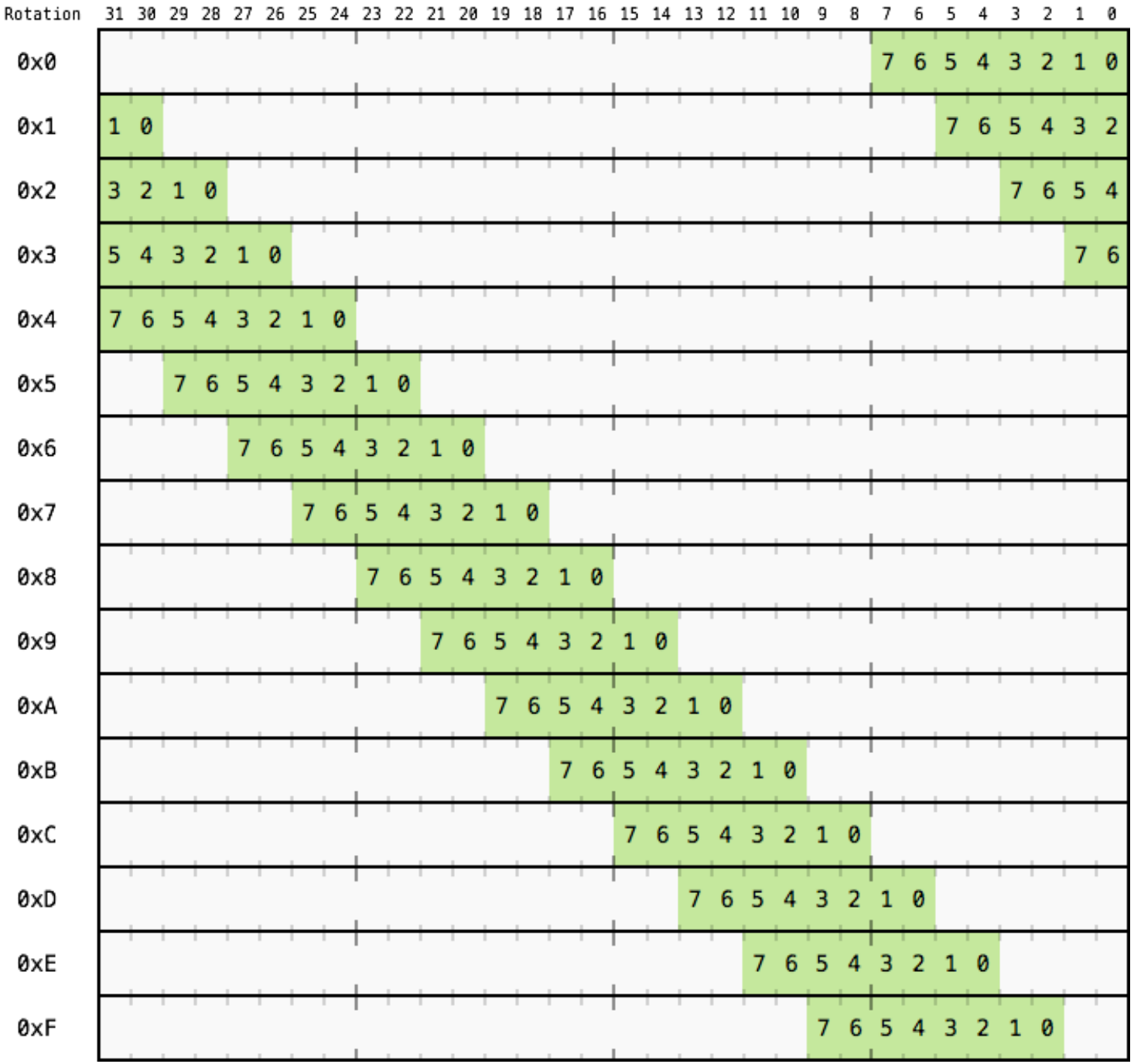
```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu

            add     r1    r0          imm
1110 00 1 0100 0 0001 0000 0000 uuuu uuuu

add r0, r1, #1

# i=1, s=0
#
# As in immediately available,
# i.e. no need to fetch from memory
```

```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu

            add     r1    r0          imm
1110 00 1 0100 0 0001 0000 0000 uuuu uuuu

add r0, r1, #1
            add     r1    r0               #1
1110 00 1 0100 0 0001 0000 0000 0000 0001
```

```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu

            add     r1    r0          imm
1110 00 1 0100 0 0001 0000 0000 uuuu uuuu

add r0, r1, #1
            add     r1    r0                #1
1110 00 1 0100 0 0001 0000 0000 0000 0001

1110 0010 1000 0001 0000 0000 0000 0001
   E    2    8    1    0    0    0    1
```


Memory
INST
DATA
Registers
+
ADDR
Shift
ALU
ADDR

Rotate Right (ROR) - Rotation amount = 2x



```
# data processing instruction
#  ra = rb op imm
# imm = (uuuu uuuu) ROR (2*rrrr)

            op      rb   ra   ror  imm
1110 00 1 oooo 0 bbbb aaaa rrrr uuuu uuuu

ROR means Rotate Right (imm>>>rotate)
```

```
# data processing instruction
#  ra = rb op imm
# imm = (uuuu uuuu) ROR (2*rrrr)

            op      rb   ra   ror  uuu
1110 00 1 oooo 0 bbbb aaaa rrrr uuuu uuuu

add r0, r1, #0x10000
            add     r1   r0   0x01>>>2*8
1110 00 1 0100 0 0001 0000 1000 0000 0001

0x01>>>16
0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0001 0000 0000 0000 0000
```

```
# data processing instruction
#  ra = rb op imm
# imm = (uuuu uuuu) ROR (2*rrrr)

           op      rb   ra   ror  imm
1110 00 1 oooo 0 bbbb aaaa rrrr uuuu uuuu

add r0, r1, #0x10000
           add     r1   r0   0x01>>>2*8
1110 00 1 0100 0 0001 0000 1000 0000 0001

1110 0010 1000 0001 0000 1000 0000 0001
   E    2    8    1    0    8    0    1
```

```
# Determine the machine code for

sub r7, r5, #0x300

# imm = (uuuu uuuu) ROR (2*rrrr)

# Remember that ra is the result

                op    rb    ra    ror   imm
1110 00 i oooo s bbbb aaaa rrrr uuuu uuuu

// What is the machine code?
```

hint:

Assembly	Code	Operations
SUB	0010	ra=rb-rc

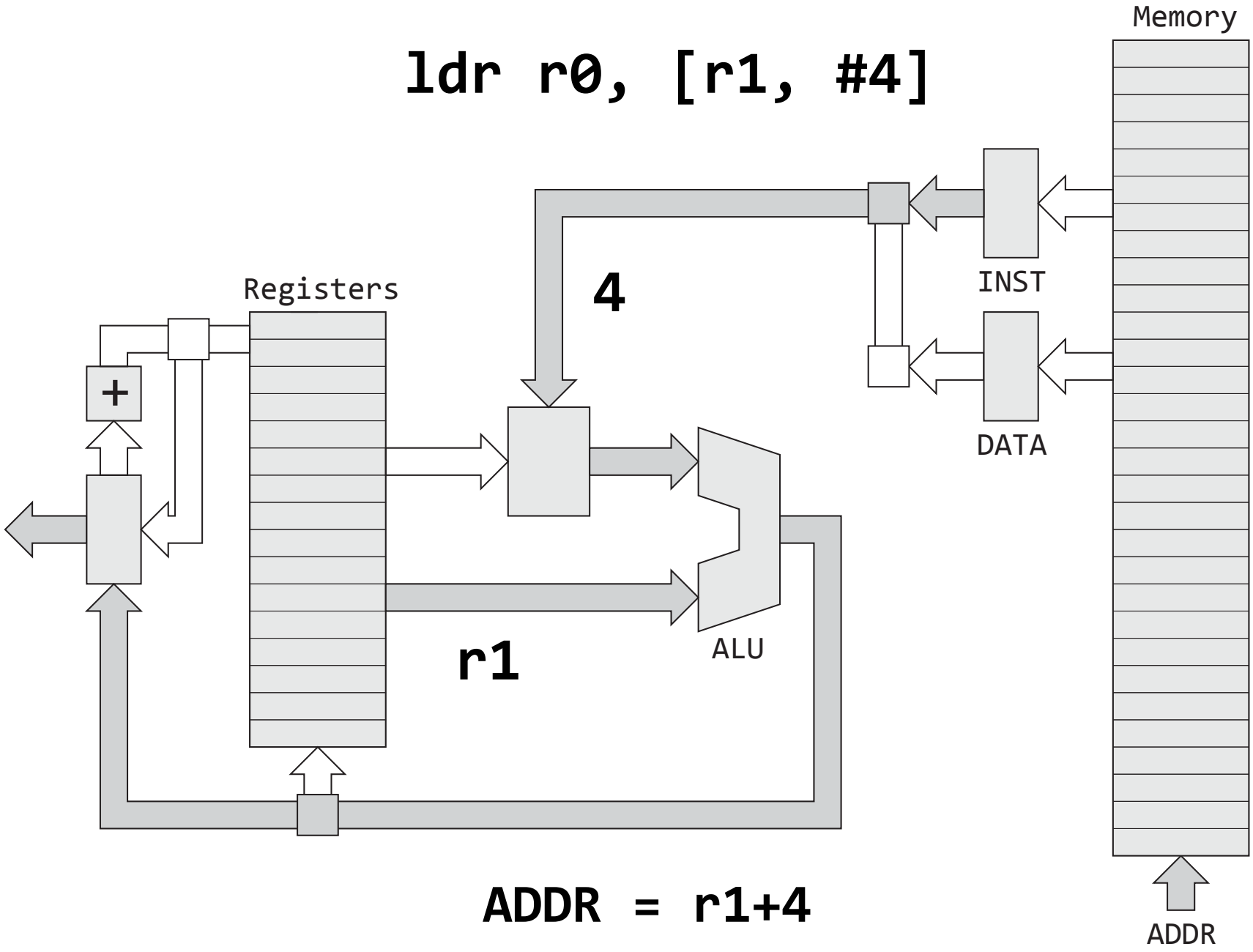
```
# data processing instruction
#  ra = rb op imm
# imm = uuuu uuuu ROR (2*rrrr)

           op      rb   ra   ror
1110 00 i oooo s bbbb aaaa rrrr uuuu uuuu

sub r7, r5, #0x300
           sub     r5   r7   #0x03>>>24
1110 00 1 0010 0 0101 0111 1100 0000 0011

1110 0010 0100 0101 0111 1100 0000 0011
   E    2    4    5    7    C    0    3
```

```
///  SET0 = 0x2020001c
mov r0, #0x20       // r0 = 0x00000020
lsl r1, r0, #24     // r1 = 0x20000000
lsl r2, r0, #16     // r2 = 0x00200000
orr r0, r1, r2      // r0 = 0x20200000
orr r0, r0, #0x1c   // r0 = 0x2020001c

//  SET0 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0
```

```
///  SET0 = 0x2020001c
mov r0, #0x20        // r0 = 0x00000020
lsl r1, r0, #24      // r1 = 0x20000000
lsl r2, r0, #16      // r2 = 0x00200000
orr r0, r1, r2       // r0 = 0x20200000
orr r0, r0, #0x1c    // r0 = 0x2020001c

//  SET0 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0
```

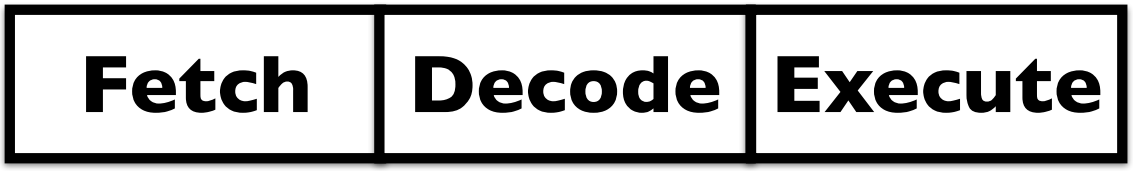
Using the barrel shifter lets us make the code 40% shorter (and 40% faster)

Load from Memory to Register (LDR)

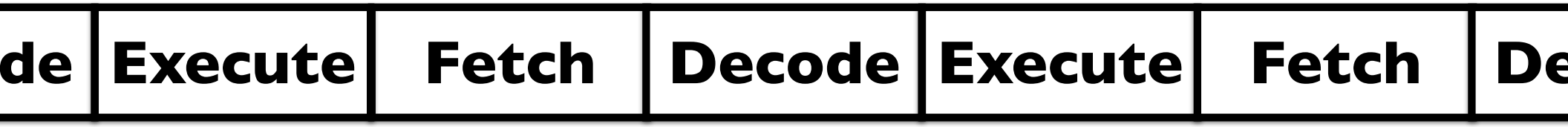


```
// configure GPIO 20 for output
ldr r0, [pc + 20]
mov r1, #1
str r1, [r0]

// set bit 20
ldr r0, [pc  + 12]
mov r1, #0x00100000
str r1, [r0]

loop: b loop

.word 0x20200008
.word 0x2020001C
```

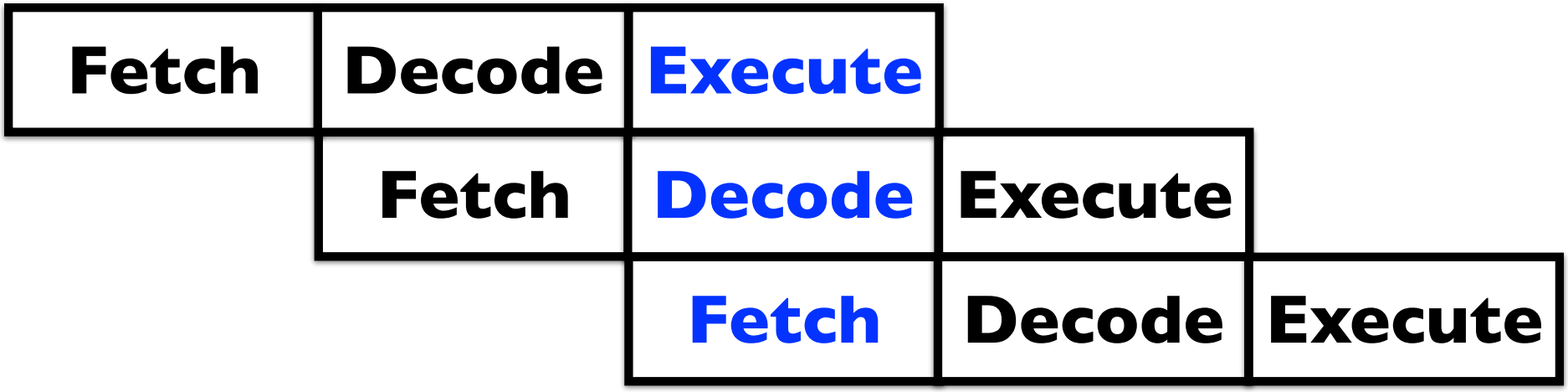
3 steps to run an instruction



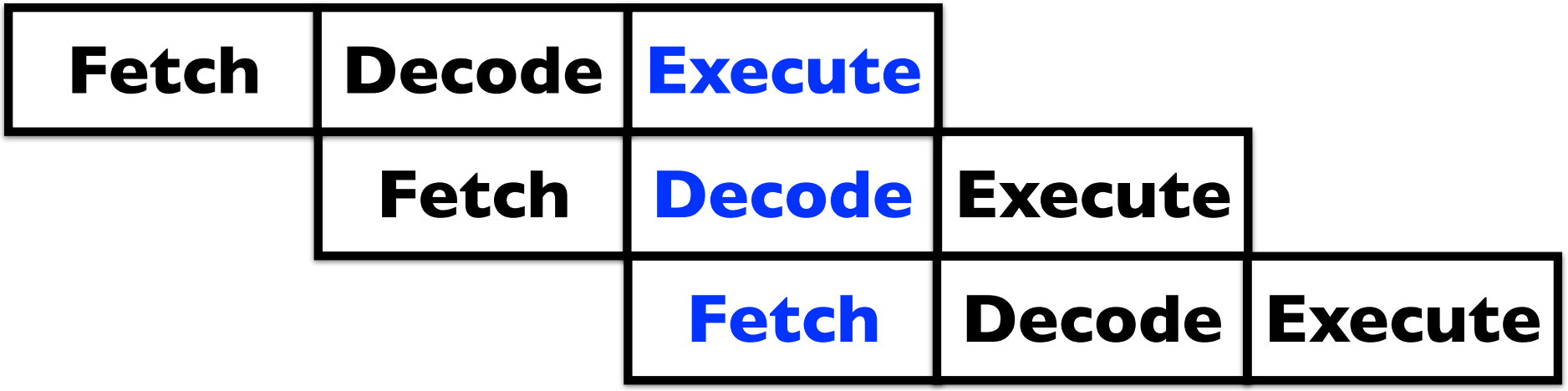
3 instructions takes 9 steps



To speed things up, steps are overlapped ("pipelined")



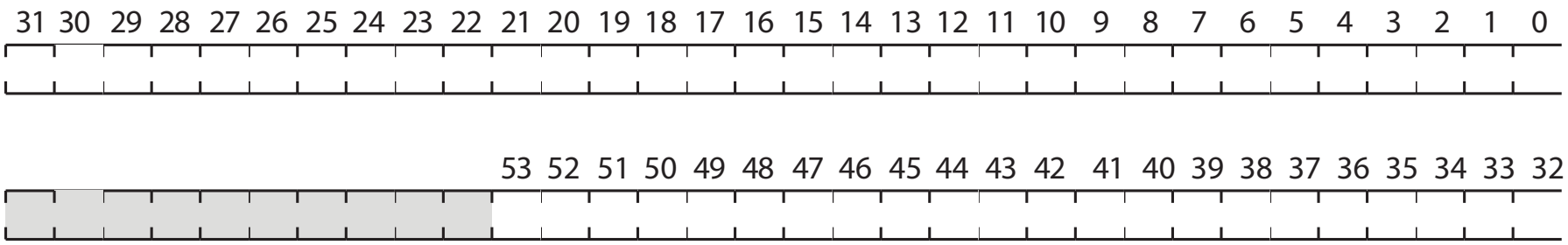
To speed things up, steps are overlapped ("pipelined")



PC value in the executing instruction is equal to the pc value of the instruction being fetched - which is 2 instructions ahead (PC+8)

```
// configure GPIO 20 for output
ldr r0, =0x20200008
mov r1, #1
str r1, [r0]

// set bit 20
ldr r0, =0x2020001C
mov r1, #0x00100000
str r1, [r0]

loop: b loop
```

Blink




```
mov r1, #(1<<20)

// Turn on LED connected to GPIO20
ldr r0, SET0
str r1, [r0]

// Turn off LED connected to GPIO20
ldr r0, CLR0
str r1, [r0]
```

```
// Configure GPIO 20 for OUTPUT

loop:

  // Turn on LED

  // Turn off LED

  b loop
```

Loops, Branches, and Condition Codes

```
// define constant
.equ DELAY, 0x3f0000


mov r2, #DELAY

loop:

    subs r2, r2, #1 // s set cond code

    bne  loop       // branch if r2 != 0
```

Condition Codes

Z - Result is 0

N - Result is <0

C - Carry generated

V - Arithmetic overflow

Carry and overflow will be covered later

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

```
# branch
cond       addr
cccc 101L oooo oooo oooo oooo oooo oooo

b = bal = branch always
cond       addr
1110 101L oooo oooo oooo oooo oooo oooo

bne
cond       addr
0001 101L oooo oooo oooo oooo oooo oooo
```

Orthogonal Instructions

Any operation

Register vs. immediate operands

All registers the same**

Predicated/conditional execution

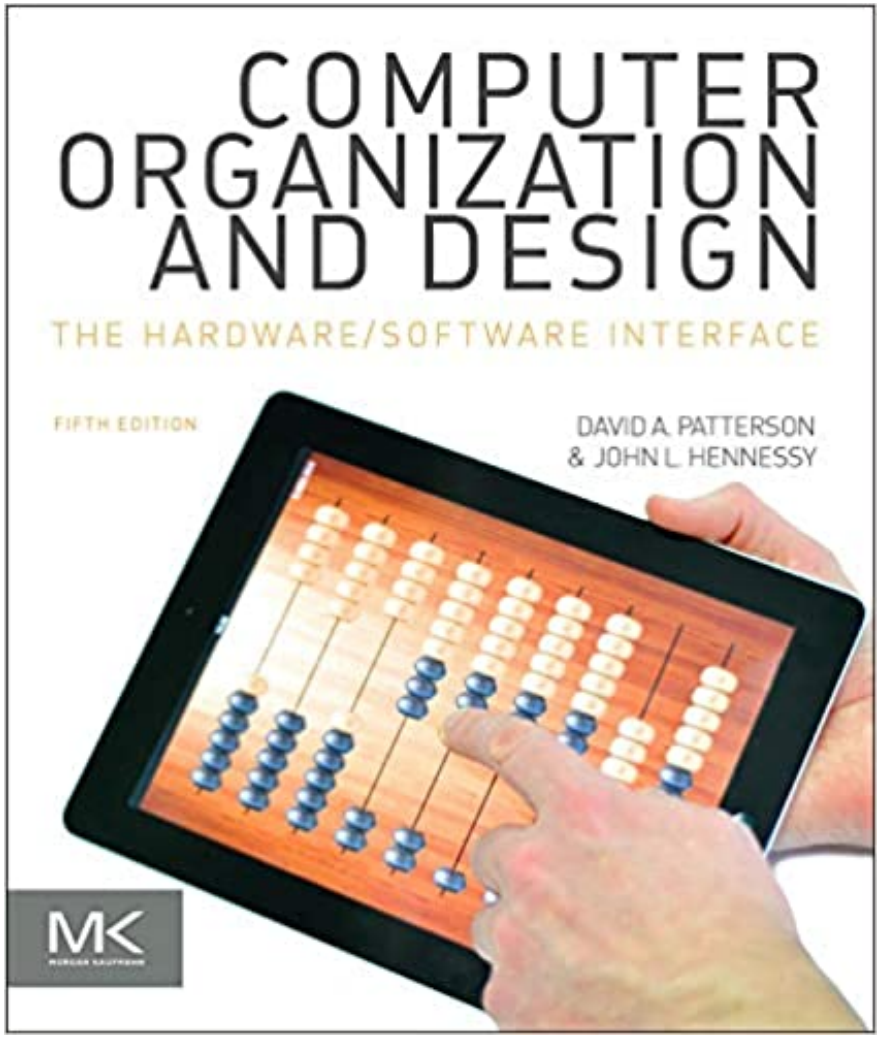
Set or not set condition code

Orthogonality leads to composability

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.

Or take EE180 in Spring!



The Fun Begins ...

Lab 1

- **Install tool chain before lab**
- **Read lab 1 instructions (now online)**
- **Assemble Raspberry Pi Kit**
- **Bring USB-C to USB-A adapter (if you need it)**

Assignment 1

- **Larson scanner**
- **Attend YEAH office hours on Wed**

Definitive References

BCM2835 peripherals document + errata

Raspberry Pi schematic

ARM11/ARMv6 reference manual

see `Resources` **on cs107e.github.io**