ARM

Assembly Language and Machine Code

Goal: Blink an LED

3 Types of Instructions

1. Data processing instructions
2. Loads from and stores to memory
3. Conditional branches to new program locations

Data Processing Instructions and Machine Code

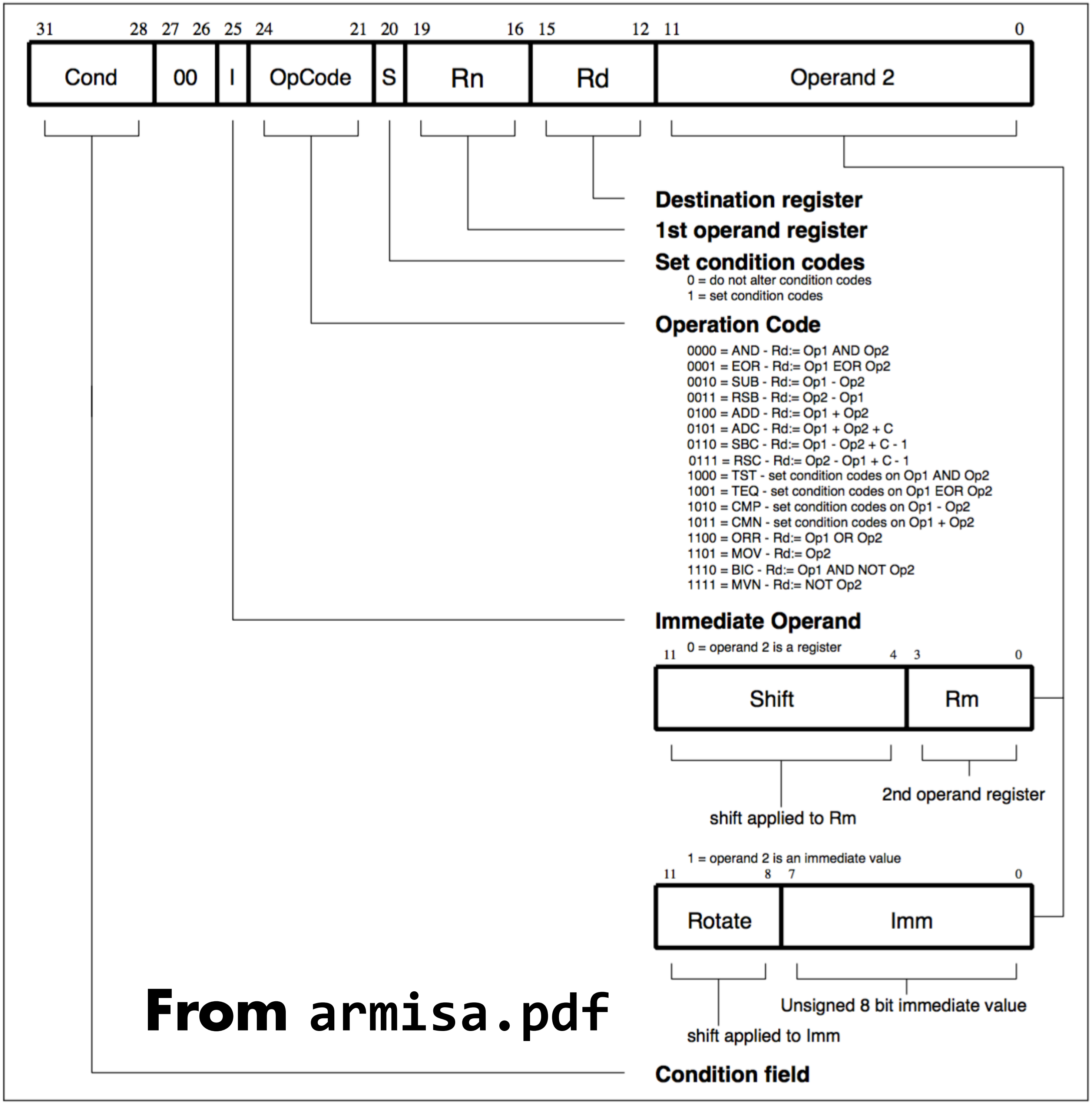


Figure 4-4: Data processing instructions

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

Immediate mode instruction

Set condition codes

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

Data processing instruction

Always execute the instruction

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

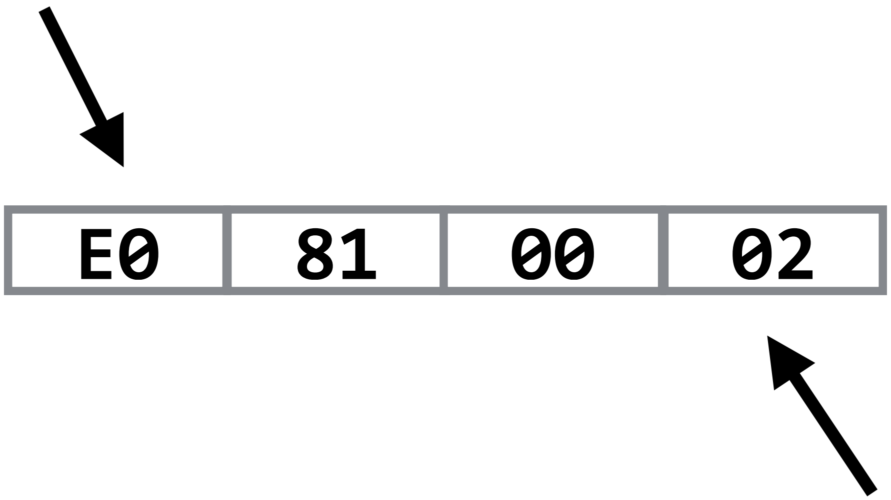
```
# data processing instruction
#  ra = rb op rc
#

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


            add      r1   r0   r2
1110 00 0 0100 0 0001 0000 0000 0000 0010

# i=0, s=0
```

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

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


           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
```

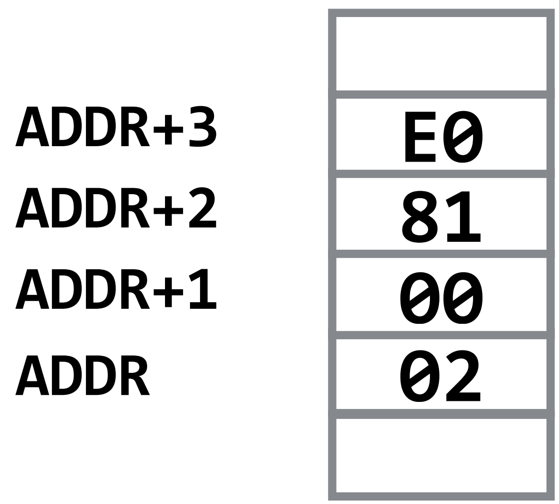
```
# Assemble (.s) into 'object' file (.o)
% arm-none-eabi-as add.s -o add.o

# 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
```

32-bit word consists of 4 consecutive bytes

most-significant-byte (MSB)



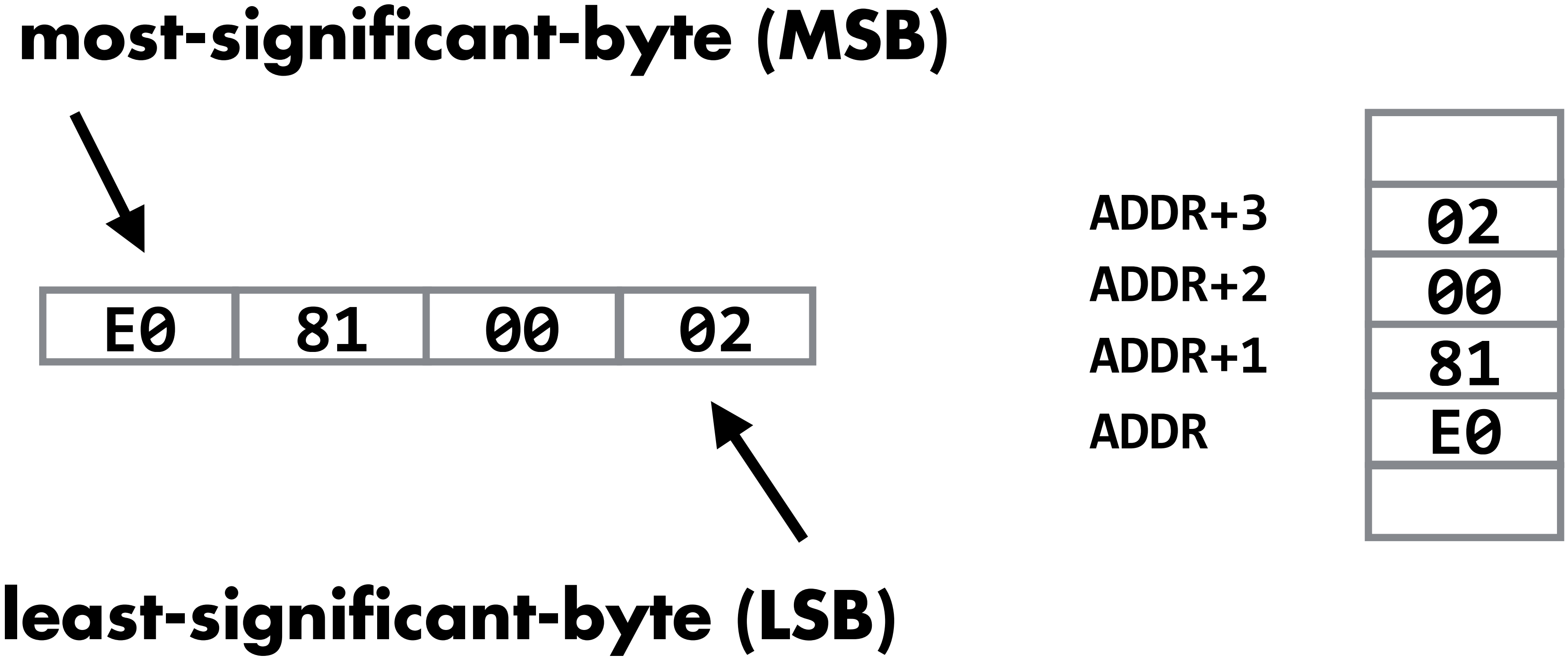
least-significant-byte (LSB)



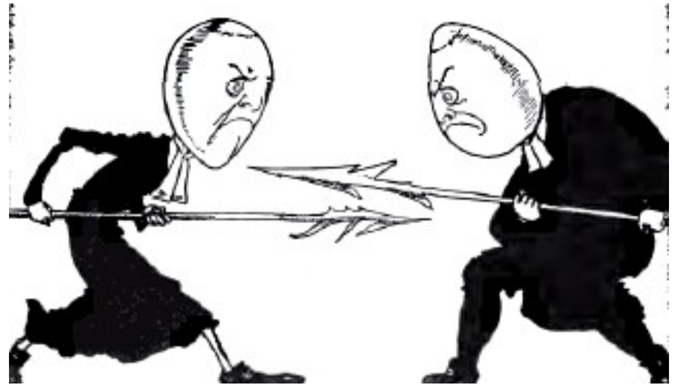
little-endian (LSB first)

ARM uses little-endian

32-bit word consists of 4 consecutive bytes



big-endian
(MSB first)

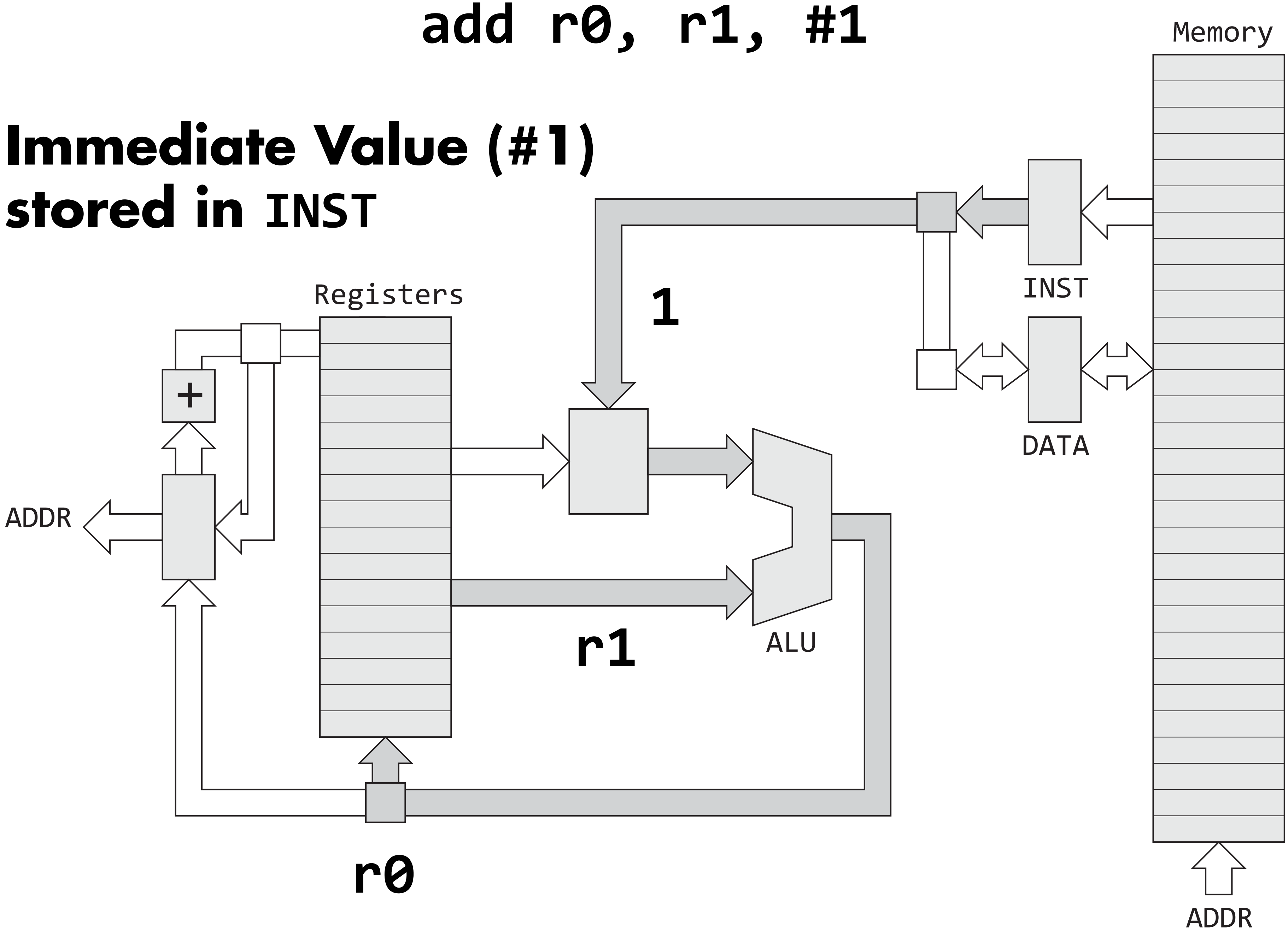


The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's Gulliver s Travels. The inhabitants of Lilliput, who are well known for being rather small, are, in addition, constrained by law to break their eggs only at the little 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

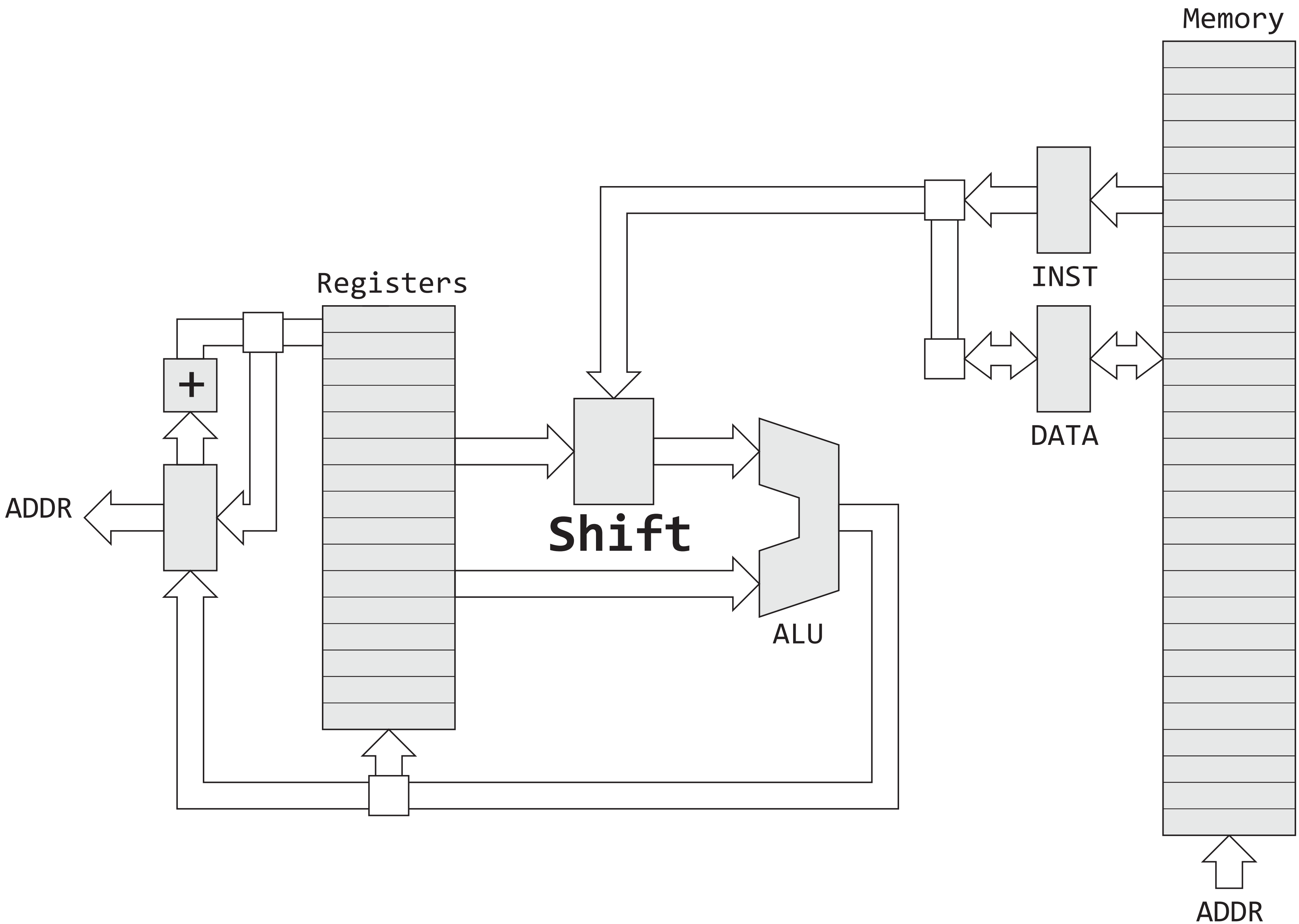
Immediate Value (#1) stored in INST



```
# 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
Shift
ALU
ADDR
+
ADDR

Rotate Right (ROR) - Rotation amount = 2x

Rotation	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0																									7	6	5	4	3	2	1	0
0x1	1	0																									7	6	5	4	3	2
0x2	3	2	1	0																									7	6	5	4
0x3	5	4	3	2	1	0																									7	6
0x4	7	6	5	4	3	2	1	0																								
0x5			7	6	5	4	3	2	1	0																						
0x6					7	6	5	4	3	2	1	0																				
0x7							7	6	5	4	3	2	1	0																		
0x8									7	6	5	4	3	2	1	0																
0x9											7	6	5	4	3	2	1	0														
0xA													7	6	5	4	3	2	1	0												
0xB															7	6	5	4	3	2	1	0										
0xC																	7	6	5	4	3	2	1	0								
0xD																			7	6	5	4	3	2	1	0						
0xE																					7	6	5	4	3	2	1	0				
0xF																							7	6	5	4	3	2	1	0		

```
# 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)

Made up notation!

Note only 4-bits available to specify the rotation

```
# 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
```

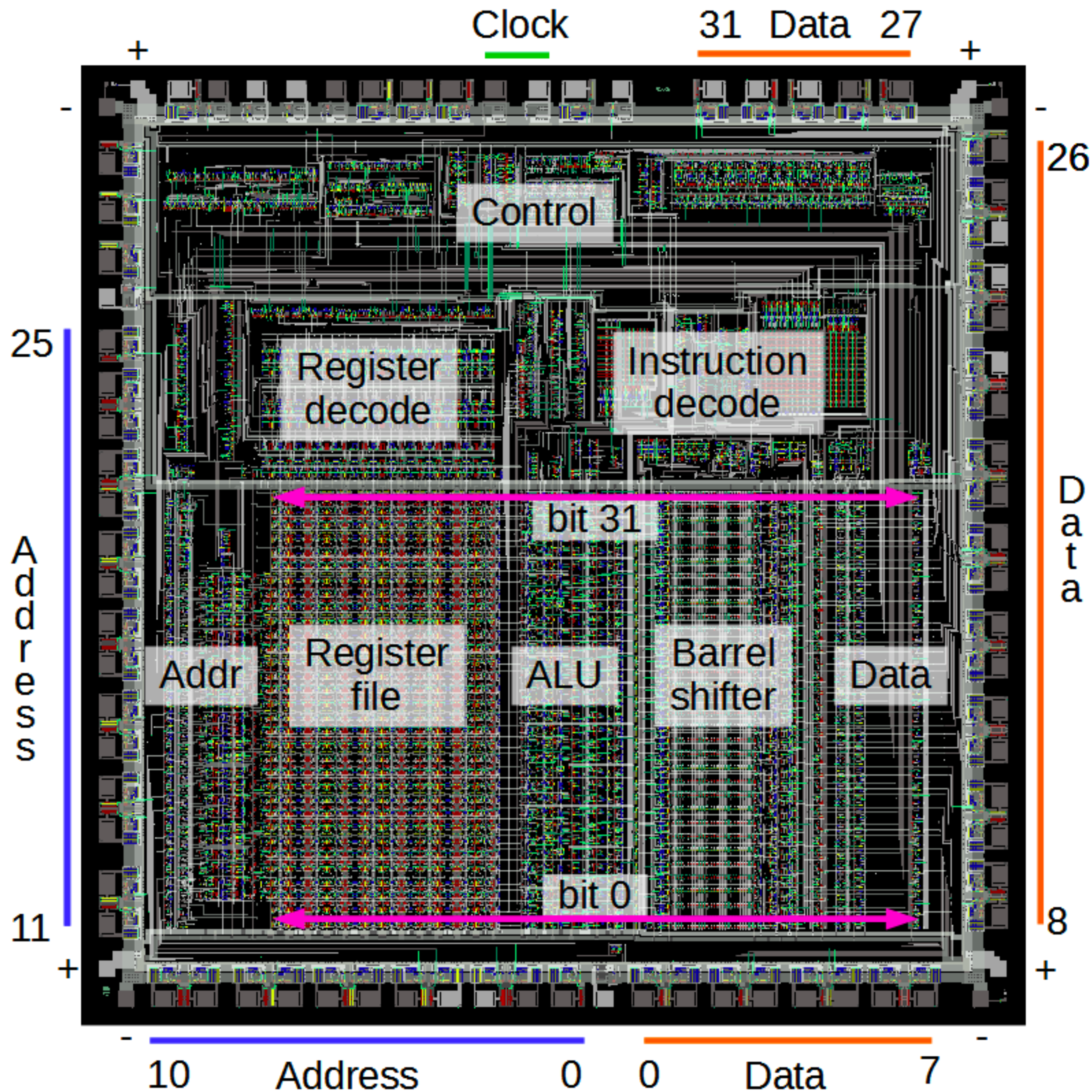
```
# data processing instruction
#  ra = rb op imm
# imm = (uuuu uuuu) ROR (2*rrrr)

                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>>>(2*12)
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
```

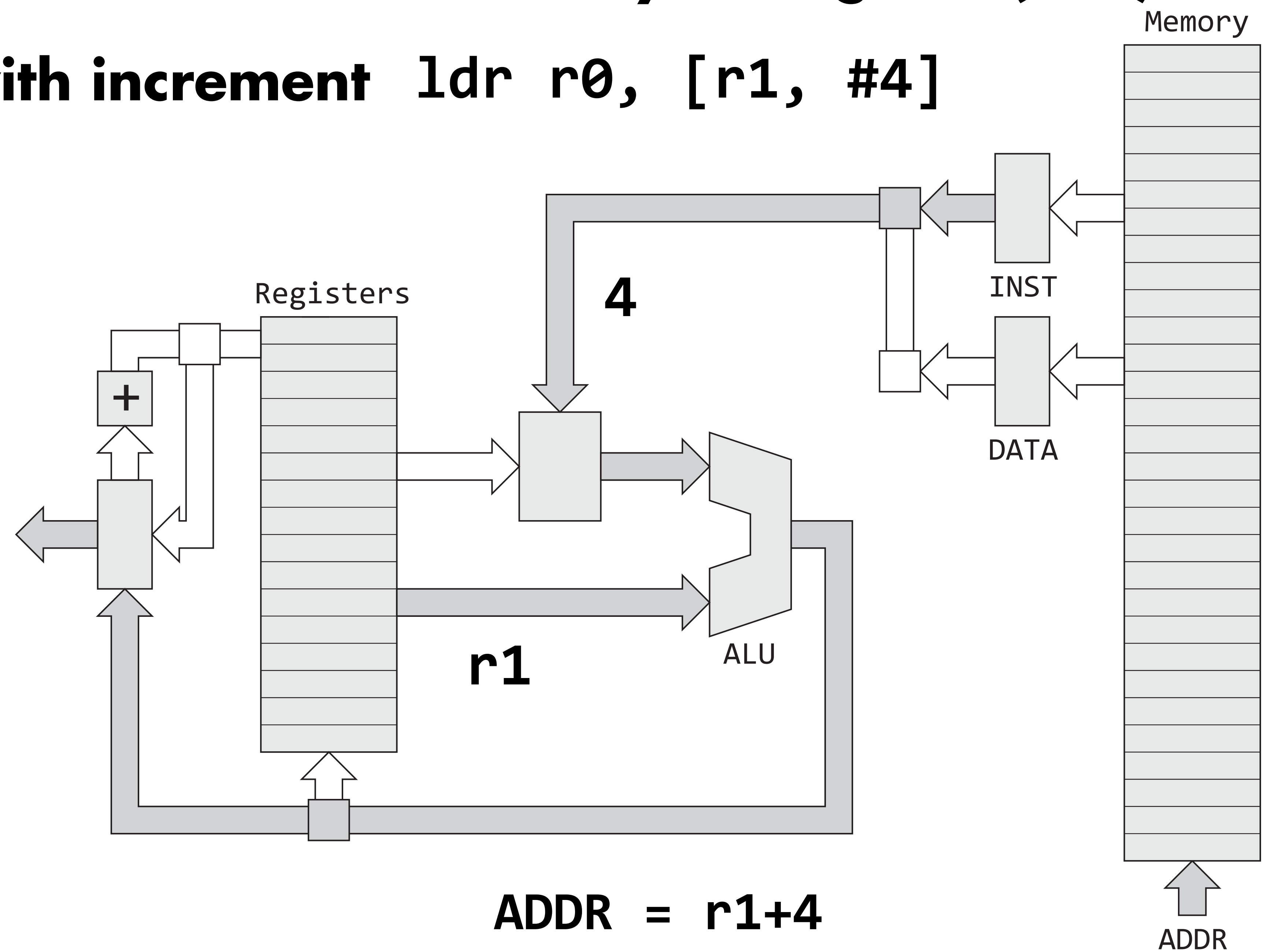

Clock
31 Data 27
Control
Register decode
Instruction decode
bit 31
Addr
Register file
ALU
Barrel shifter
Data
bit 0
25
Address
11
26
Data
8
10 Address 0
0 Data 7
+
-

…

```
//  SET1 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0
```

Load from Memory to Register (LDR)

with increment `ldr r0, [r1, #4]`



ADDR = r1+4

DATA = Memory[ADDR]

```
// configure GPIO 20 for output
ldr r0, FSEL2
mov r1, #1
str r1, [r0]

// set bit 20

ldr r0, SET0
mov r1, #0x00100000
str r1, [r0]

loop: b loop

FSEL0: .word 0x20200000
FSEL1: .word 0x20200004
FSEL2: .word 0x20200008
SET0:  .word 0x2020001C
SET1:  .word 0x20200020
CLR0:  .word 0x20200028
CLR1:  .word 0x2020002C
```

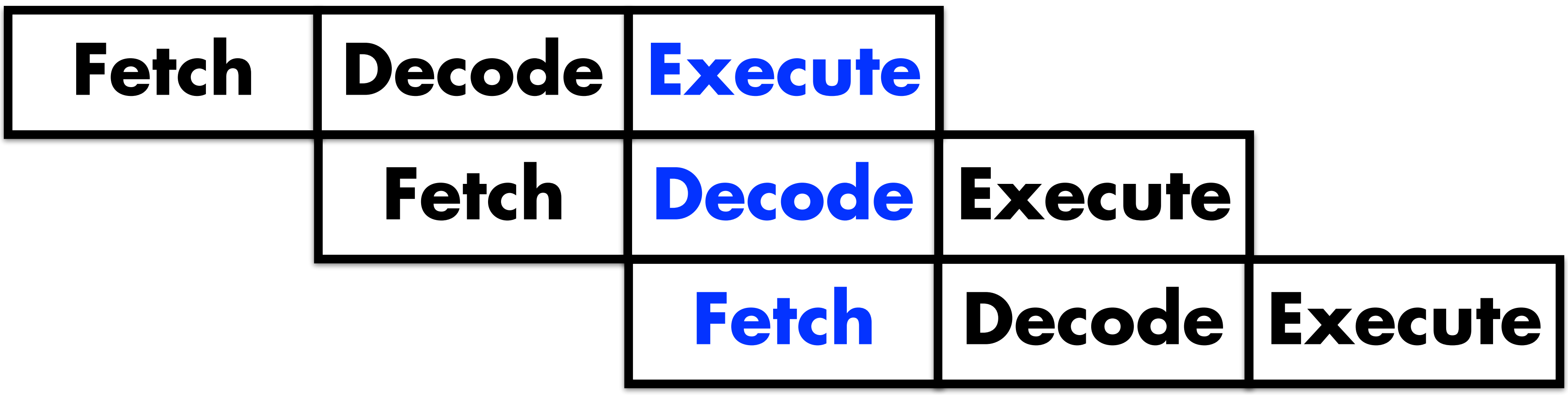
3 steps to run an instruction

Fetch	Decode	Execute

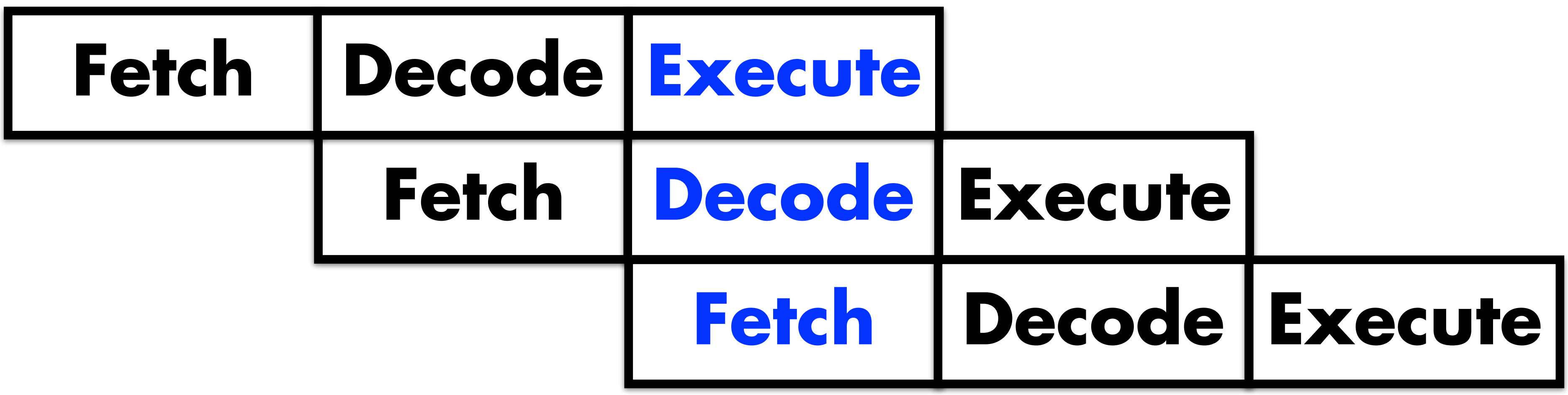
3 instructions takes 9 steps

de	Execute	Fetch	Decode	Execute	Fetch	De

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

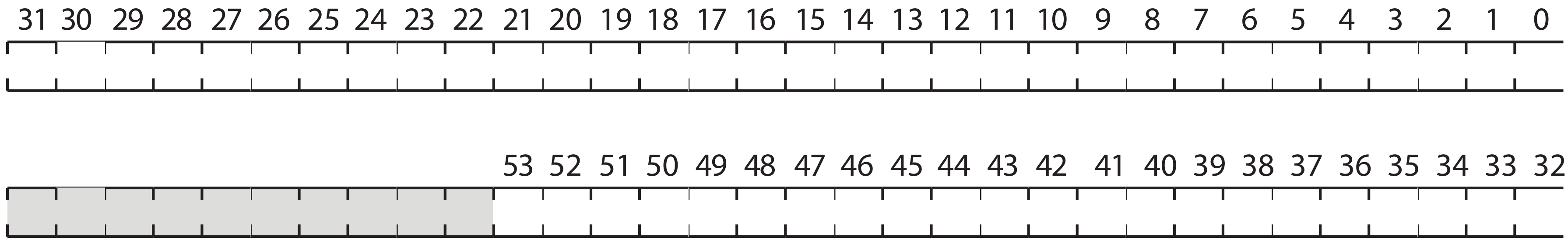


To speed things up, steps are overlapped ("pipelined")



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

Blink




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

// Turn on LED connected to GPIO20
// Writing a 1 to a bit in SET makes the output 1
// Writing a 0 to a bit in SET has no effect
ldr r0, SET0
str r1, [r0]

// Turn off LED connected to GPIO20
// Writing a 1 to a bit in CLR makes the output 0
// Writing a 0 to a bit in CLR has no effect
ldr r0, CLR0
str r1, [r0]
```

```
// Configure GPIO 20 for OUTPUT

loop:

  // Turn on LED

  // Turn off LED

  b loop
```

Loops 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
```

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

Summary

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 (Julie, starting Fri)

The Fun Begins ...

Lab1

- **Assemble Raspberry Pi Kit**
- **Introduction to breadboarding**
- **SDHC card and the boot loader**
- **blink and button**
- **Read lab1 instructions (now online)**

Assignment 1

- **Larson scanner**
- **YEAH office hours Thu 3-4pm in B21**

Definitive References

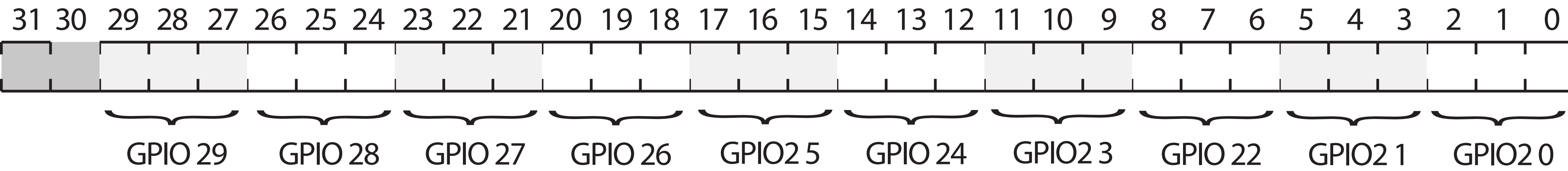
BCM2865 peripherals document + errata

Raspberry Pi schematic

ARMv6 architecture reference manual

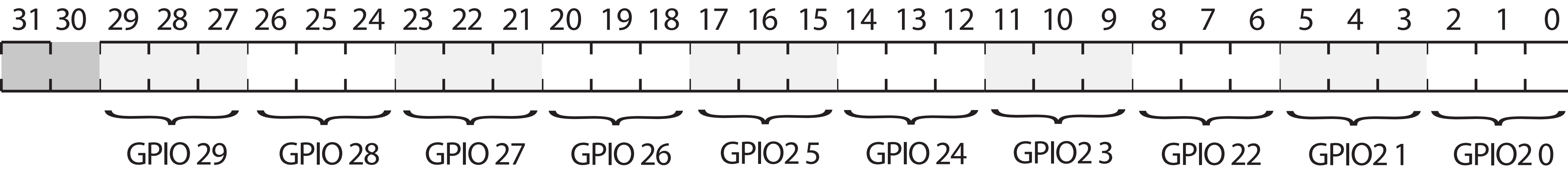
see `Resources` **on** **cs107e.github.io**

Manipulating Bit Fields




```
// Set GPIO 20 to OUTPUT
mov r1, #1
str r1, [r0]

// Set GPIO 21 to OUTPUT
mov r1, #(1<<3)
str r1, [r0]

// What value is in FSEL2 now?
// What mode is GPIO 20 set to now?
```

```
// LDR FSEL2, GPIO20 is OUTPUT
ldr r1, [r0]
0000 0010 0000 0000 0000 0000 0010 0001
// 0x7
0000 0000 0000 0000 0000 0000 0000 0111
// 0x7<<3
0000 0000 0000 0000 0000 0000 0011 1000
// ~(0x7<<3)
1111 1111 1111 1111 1111 1111 1100 0111
and r1, #~(0x7<<3)
0000 0000 0000 0000 0000 0000 0000 0001
orr r1, #(0x1<<3)
0000 0010 0000 0000 0000 0000 0000 1001
```




```
// Set GPIO 20 to OUTPUT
mov r1, #1
str r1, [r0]
…

// Preserve GPIO20, set GPIO21 to OUTPUT
ldr r1, [r0]
and r1, #~(0x7<<3)
orr r1, #(0x1<<3)
str r1, [r0]

// What value is in FSEL2 now?
```