# ARM

# Assembly Language and Machine Code

# Concepts

Types of ALU instructions

Bits and bit operations

Condition codes

Branches

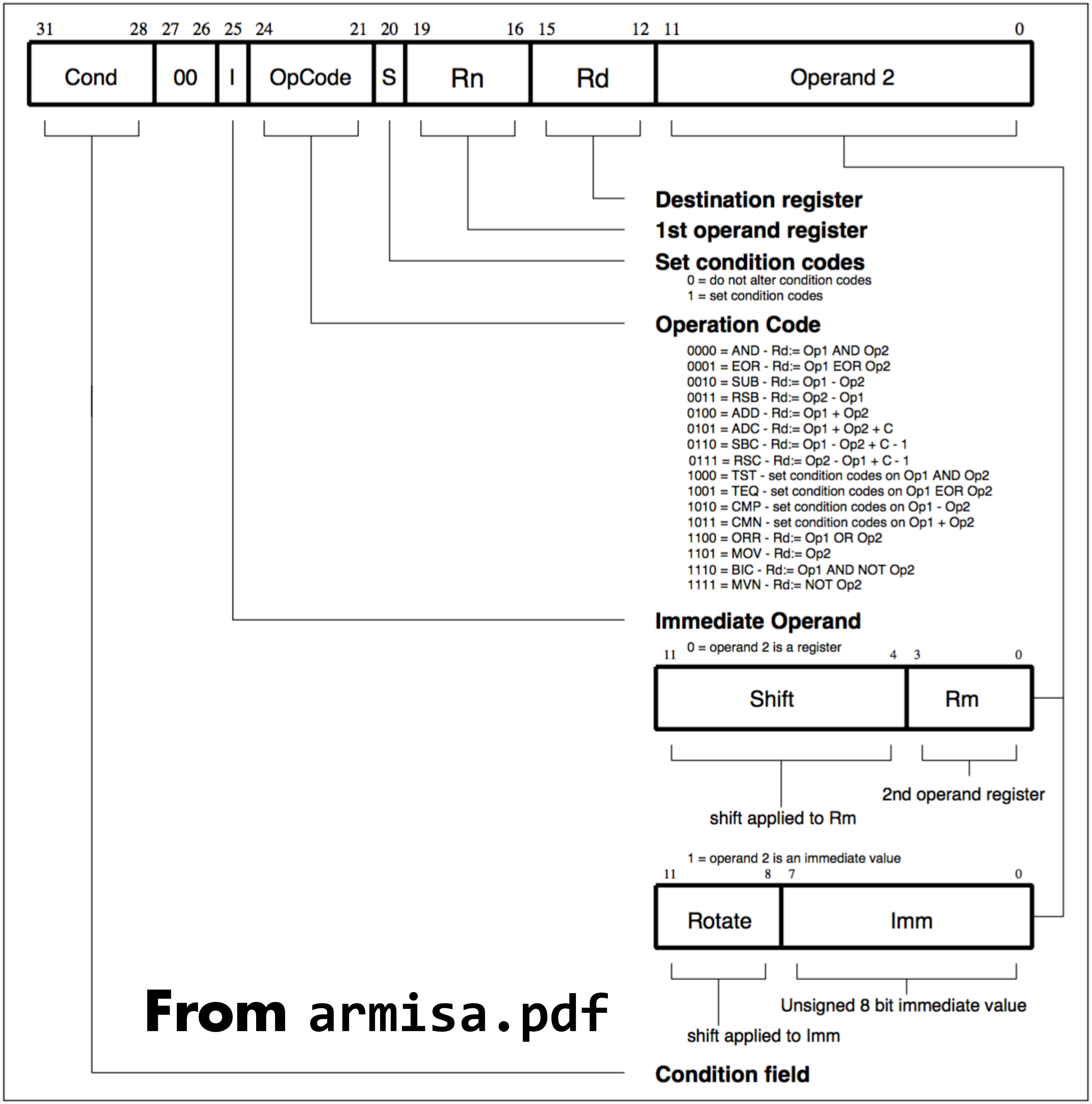
Addressing modes in loads & stores

```
// Program to turn on an LED

#define FSEL2 0x20200008
ldr r0, =FSEL2
mov r1, #1              ; GPIO20 Output
str r1, [r0]

#define SET0  0x2020001C
ldr r0, =SET0
mov r1, #(1<<20)        ; Bit 20
str r1, [r0]

loop: b loop
```

# 3 Types of Instructions

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

# Data Processing Instructions




**From** `armisa.pdf`

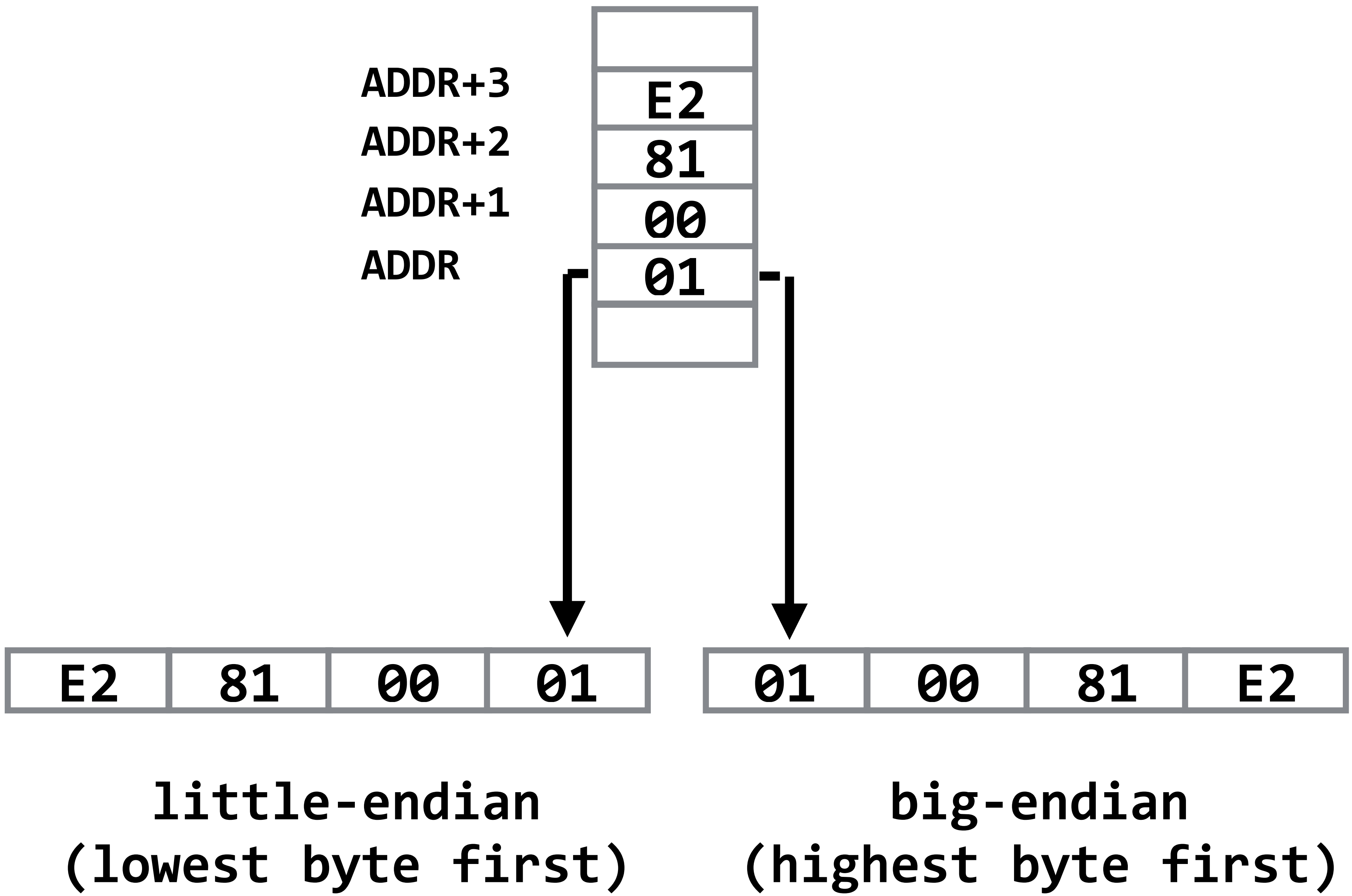
***Figure 4-4: Data processing instructions***

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

               op      rb   ra         imm
1110 00 1 oooo 0 bbbb aaaa 0000 uuuu uuuu

add r0, r1, #1
```

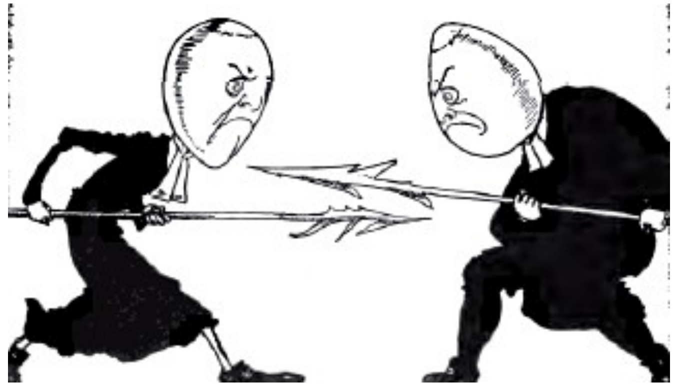
<table>
<tr><th>Assembly</th><th>Code</th><th>Operations</th></tr>
<tr><td>AND</td><td>0000</td><td>ra=rb&rc</td></tr>
<tr><td>EOR (XOR)</td><td>0001</td><td>ra=rb^rc</td></tr>
<tr><td>SUB</td><td>0010</td><td>ra=rb-rc</td></tr>
<tr><td>RSB</td><td>0011</td><td>ra=rc-rb</td></tr>
<tr><td>ADD</td><td>0100</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 #imm
# #imm = uuuu uuuu

               op       rb    ra            imm
1110 00 1 oooo 0 bbbb aaaa 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

          op      rb   ra          imm
1110 00 1 oooo 0 bbbb aaaa 0000 uuuu uuuu

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

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

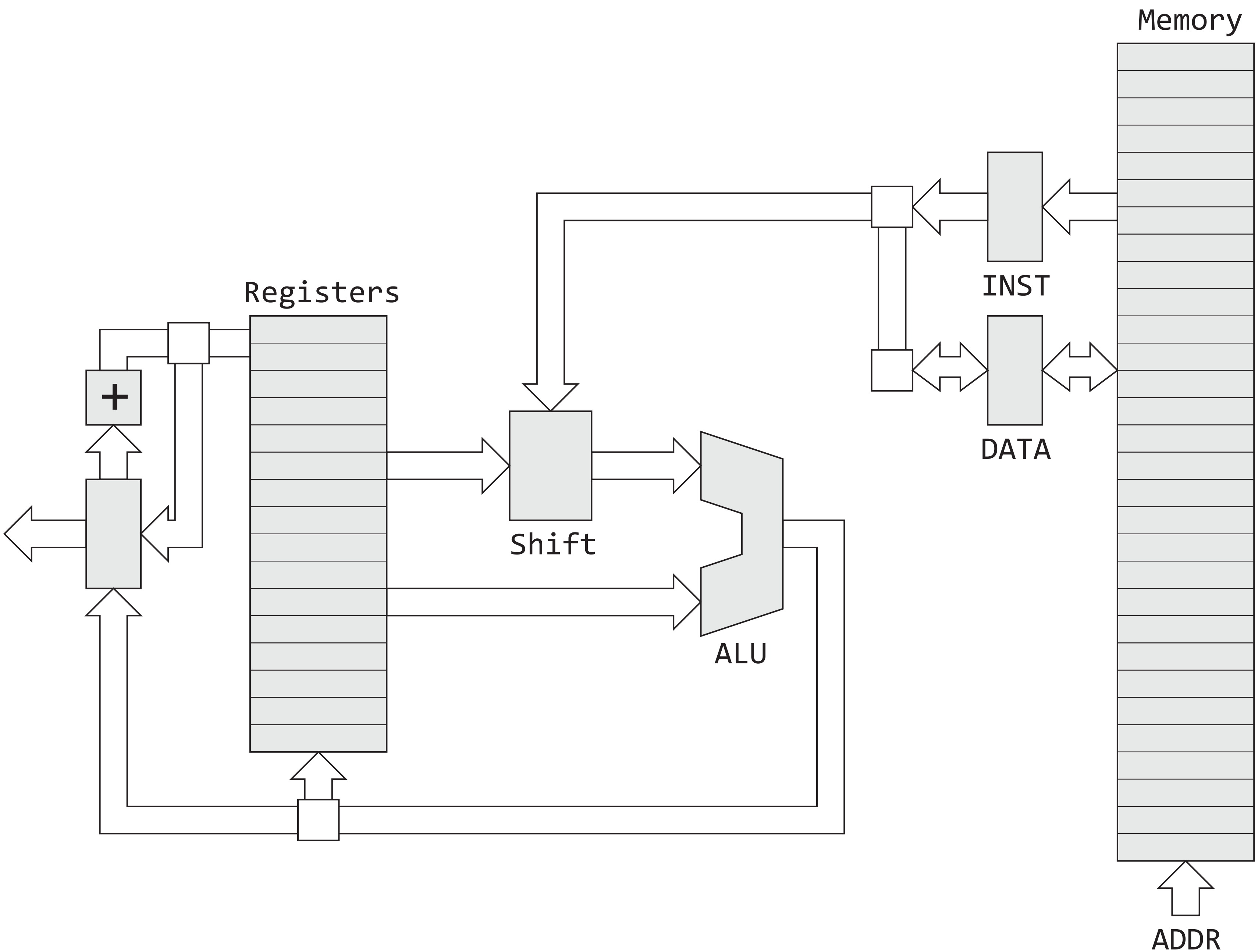


ADDR+3
ADDR+2
ADDR+1
ADDR
E2
81
00
01
E2
81
00
01
01
00
81
E2
little-endian
(lowest byte first)
big-endian
(highest byte first)



**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**


Memory
INST
DATA
Registers
+
Shift
ALU
ADDR

# Rotate Right (ROR)

| 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*iiii)

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

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

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

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

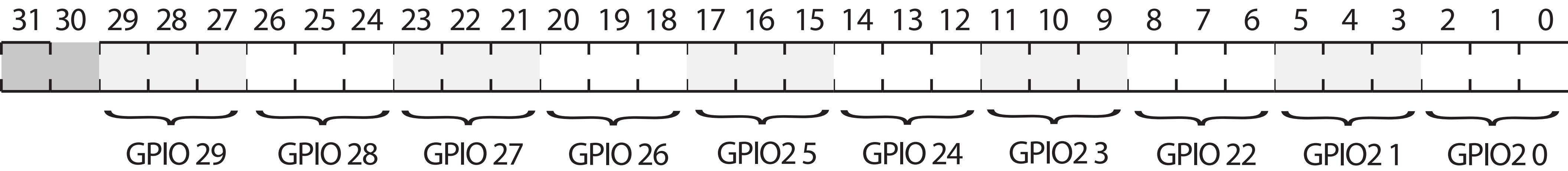
```
# data processing instruction
#  ra = rb op imm
# imm = (uuuu uuuu) ROR (2*iiii)

          op     rb   ra   ror  imm
1110 00 1 oooo 0 bbbb aaaa iiii 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*iiii)

# Remember that ra is the result

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

// What is the machine code?
```

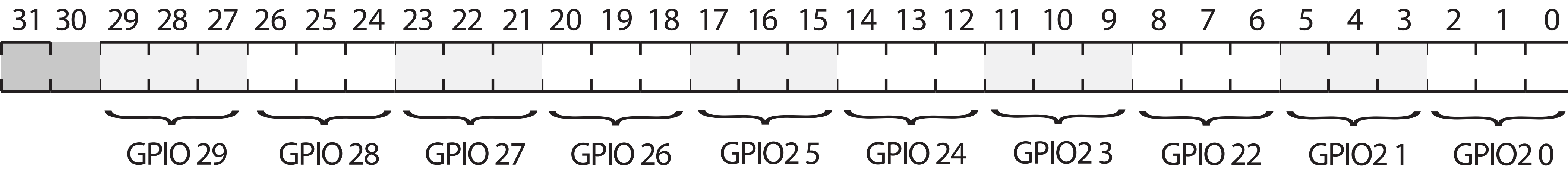
```
# data processing instruction
#  ra = rb op imm
# imm = uuuu uuuu ROR (2*iiii)

            op      rb   ra   ror
1110 00 1 oooo 0 bbbb aaaa iiii 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
```

# Bit Manipulations

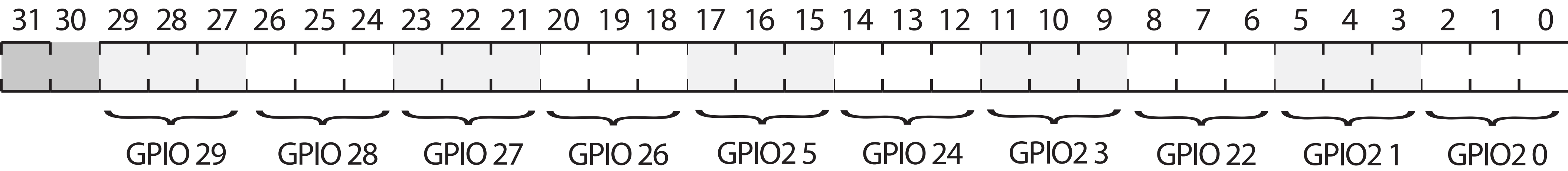
```
// May replace
#define FSEL2 0x20200008
ldr r0, =FSEL2

// with

mov r0, #0x20000000 // #(0x20>>>8)
orr r0, #0x00200000 // #(0x20>>>16)
orr r0, #0x00000008
```




```
// FSEL2 into r0
mov r0, #0x20000000 // #(0x20>>>8)
orr r0, #0x00200000 // #(0x20>>>16)
orr r0, #0x00000008
// Set GPIO 20 to OUTPUT
mov r1, #1
str r1, [r0]

// Also set GPIO 21 to OUTPUT

// How?
```




```
// FSEL2 into r0
mov r0, #0x20000000 // #(0x20>>>8)
orr r0, #0x00200000 // #(0x20>>>16)
orr r0, #0x00000008

// Set GPIO 20 and 21 both to OUTPUT
mov r1, #1
orr r1, #(1<<3)
str r1, [r0]
```




```
// FSEL2 into r0
mov r0, #0x20000000 // #(0x20>>>8)
orr r0, #0x00200000 // #(0x20>>>16)
orr r0, #0x00000008
// Set GPIO 20 to OUTPUT
mov r1, #1
…
// Set GPIO 21 to OUTPUT
ldr r1, [r0]
bic r1, #(0x7<<3)
orr r1, #(0x1<<3)
str r1, [r0]
```

# Condition Codes

```
// loop
#define DELAY 0x3F0000
mov r2, #DELAY

loop:

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

    bne  loop
```

# Condition Codes

`Z` - **Result is 0**

`N` - **Result is <0**

`C` - **Carry generated**

`V` - **Arithmetic overflow**

***Carry and overflow will be covered later***

```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu ROR (2*iiii)
#
# s – set condition code
#

     op        rb   ra
1110 00 1 oooo s bbbb aaaa iiii uuuu uuuu
```

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

# s=1 means set condition code
          op   s rb   ra
1110 00 1 oooo s bbbb aaaa iiii uuuu uuuu

subs r2, r2, #1
          sub  s r2   r2      0         1
1110 00 1 0010 1 0010 0010 0000 0000 0001

E2 52 20 01
```

# Branch Instructions

```
# 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
```

| 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 |

# Blink

```
// Configure GPIO 20 for OUTPUT

loop:

  // Turn on LED

  // delay

  // Turn off LED

  // delay

  b loop
```

```
// Program to turn on an LED
// Setup GPIO 20
#define FSEL2 0x20200008
ldr r0, =FSEL2
mov r1, #1
str r1, [r0]

// Bit 20 for GPIO 20
mov r1, #(1<<20)
```

```
…

// r0 points to GPIO SET0 register
#define SET0 0x2020001C
ldr r0, =SET0
str r1, [r0]

// delay
#define DELAY 0x3F0000
mov r2, #DELAY
wait1:
    subs r2, #1
    bne wait1
```

```
…

// r0 points to GPIO CLR0 register
#define CLR0 0x20200028
ldr r0, =CLR0
str r1, [r0]

// delay
mov r2, #DELAY
wait2:
   subs r2, #1
   bne wait2
```

```
// GPIO registers don't act like memory

// r0 points to GPIO SET0 register
ldr r0, =SET0
str r1, [r0]

// r0 points to GPIO CLR0 register
ldr r0, =CLR0
str r1, [r0]
```

# 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

Reading assembly allows you to know what the processor is doing

Rarely write assembly,

Normally write code in C (next week)

# The Fun Begins ...

## Labs!!

- Assembly Raspberry Pi Kit
- Read lab1 guide
- Install tool chain
- Bring $50 lab fee

## Assignment 1

- Larson scanner

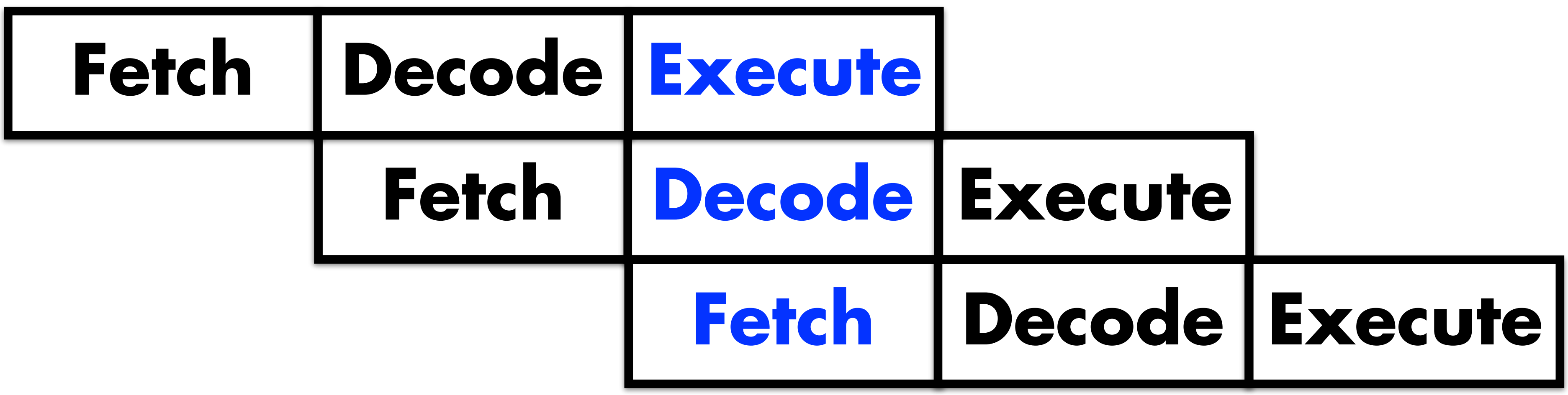
# Extra Slides

# Pipelining

# Processors execute instructions in phases

| | | |
|---|---|---|
| Fetch | Decode | Execute |

# Phases are pipelined




**PC value is 2 instructions ahead (PC+8) of the executing instruction (PC+8)**

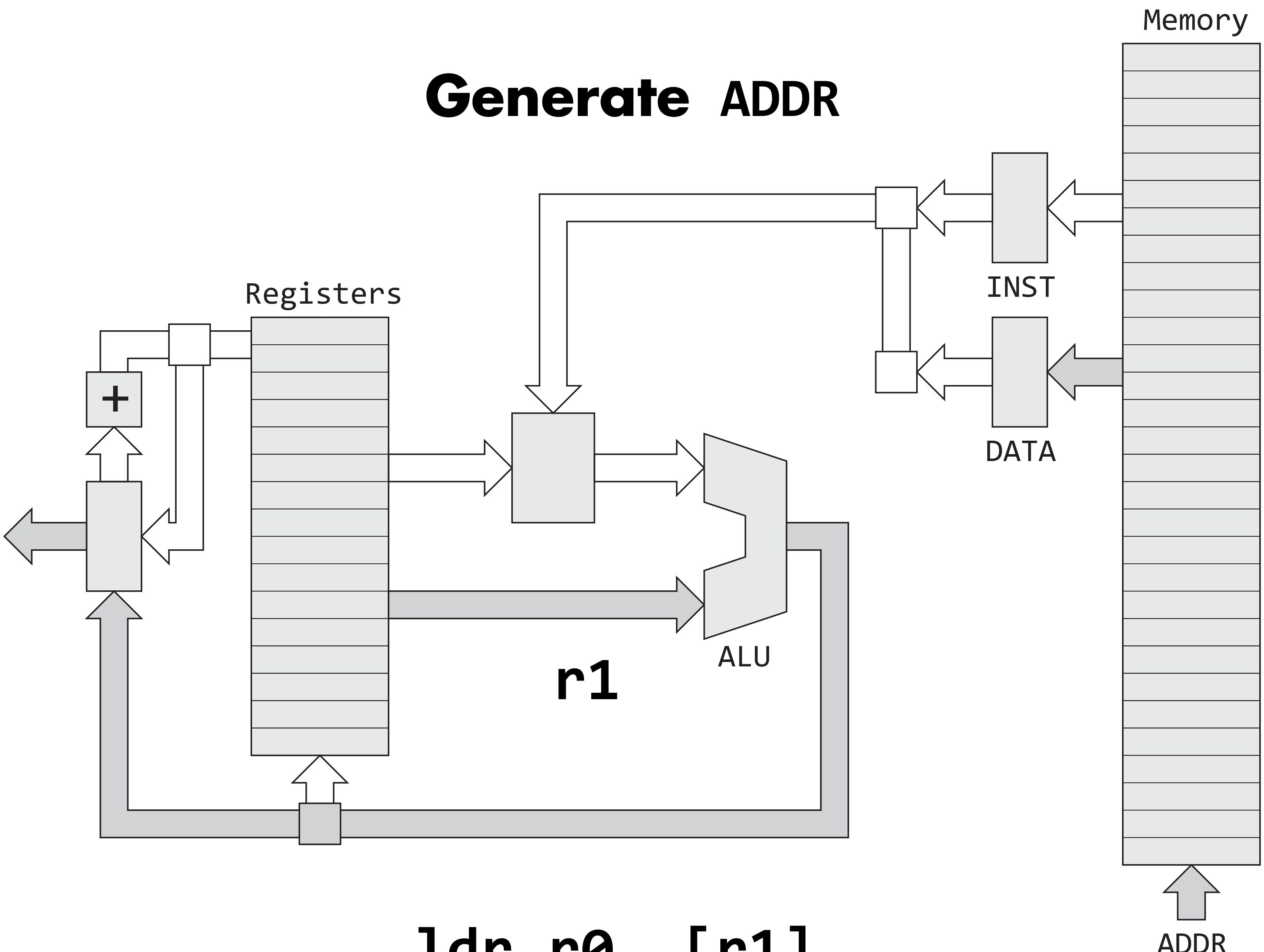
```
// disassemble on.s

 0: e59f0014   ldr r0, [pc, #0x14]
 4: e3a01001   mov r1, #1
 8: e5801000   str r1, [r0]
 c: e59f000c   ldr r0, [pc, #0x0c]
10: e3a01601   mov r1, #0x100000
14: e5801000   str r1, [r0]
18: eafffffe   b 18 // [pc, -2*4]
1c: 20200008
20: 2020001c
```

# Indexed Loads

# PC Relative Addressing

r0 = mem[r1]

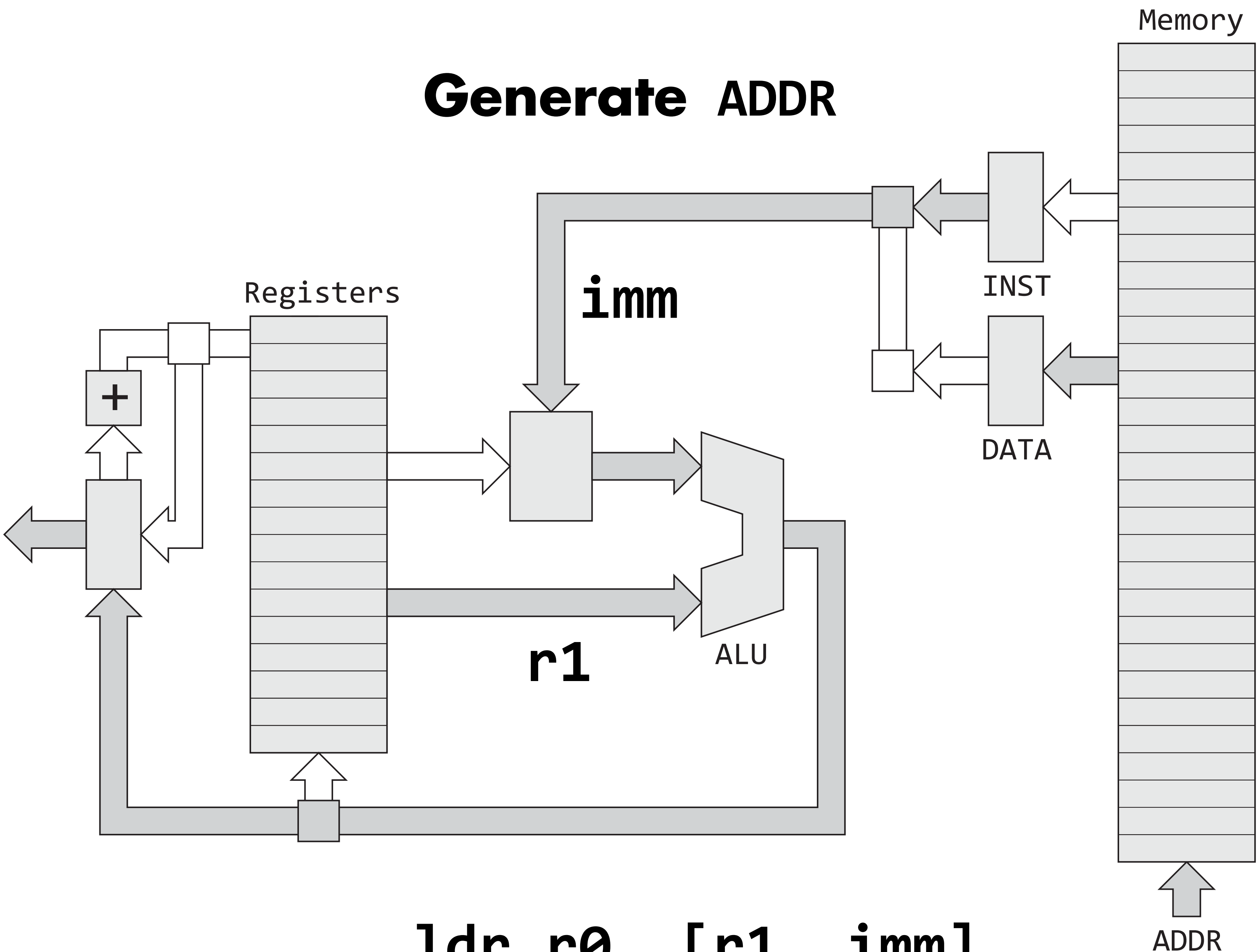
# Generate ADDR




ldr r0, [r1]

$r0 = mem[r1+imm]$

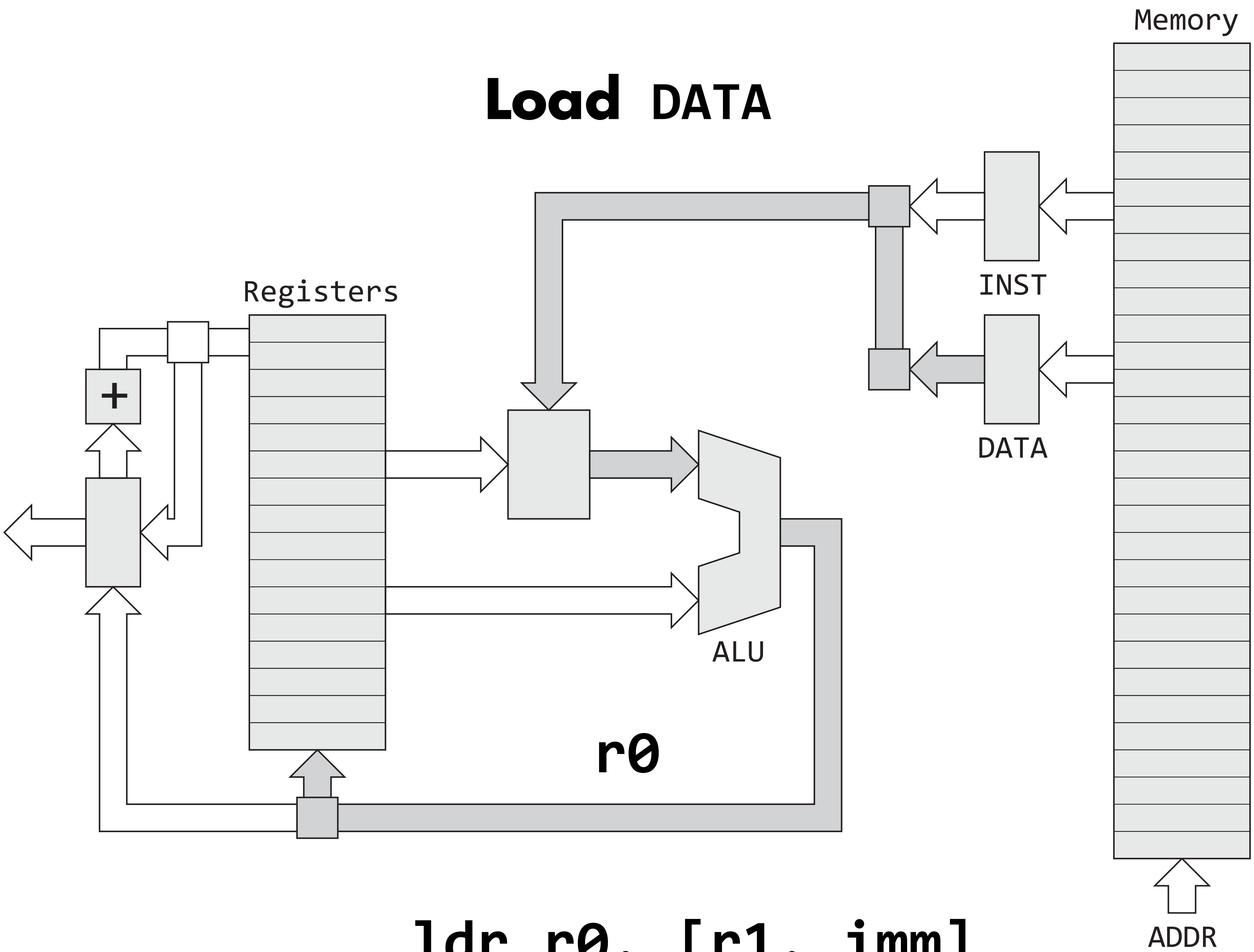
## Generate ADDR




**ldr r0, [r1, imm]**

# r0 = mem[r1+imm]

## Load DATA




**ldr r0, [r1, imm]**

```
// disassemble on.s

// PC relative addressing

 0: e59f0014   ldr r0, [pc, #0x14]
 4: e3a01001   mov r1, #1
 8: e5801000   str r1, [r0]
 c: e59f000c   ldr r0, [pc, #0x0c]
10: e3a01601   mov r1, #0x100000
14: e5801000   str r1, [r0]
18: eafffffe   b 18 // [pc, -2*4]
1c: 20200008
20: 2020001c
```

```
// disassemble on.s

// PC relative addressing

 0: e59f0014   ldr r0, [pc, #0x14]
 4: e3a01001   mov r1, #1
 8: e5801000   str r1, [r0]
 c: e59f000c   ldr r0, [pc, #0x0c]
10: e3a01601   mov r1, #0x100000
14: e5801000   str r1, [r0]
18: eafffffe   b 18 // [pc, -2*4]
1c: 20200008
20: 2020001c
```