ARM

Assembly Language and Machine Code

Goal: Blink an LED

Concepts

Types of ALU instructions

Bits and bit operations

Condition codes

Branches

```
// Program to turn on an LED

#define FSEL2 0x20200008
ldr r0, =FSEL2
mov r1, #1              ; GPI020 Output
str r1, [r0]

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

loop: b loop
```

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	-	-

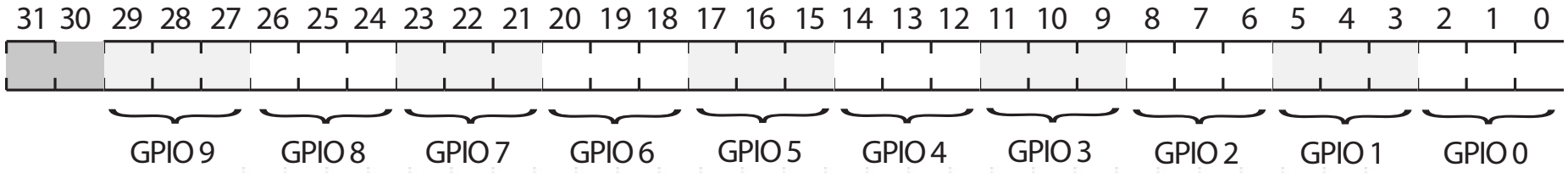
Gotcha

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

Ref: BCM2835-ARM-Peripherals.pdf

GPIO Function Select Register



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

1. 3 bits per GPIO pin
2. Max of 10 pins per 32-bit register
3. GPIO20 is in FSEL2

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 in 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 *n*	R/W	0

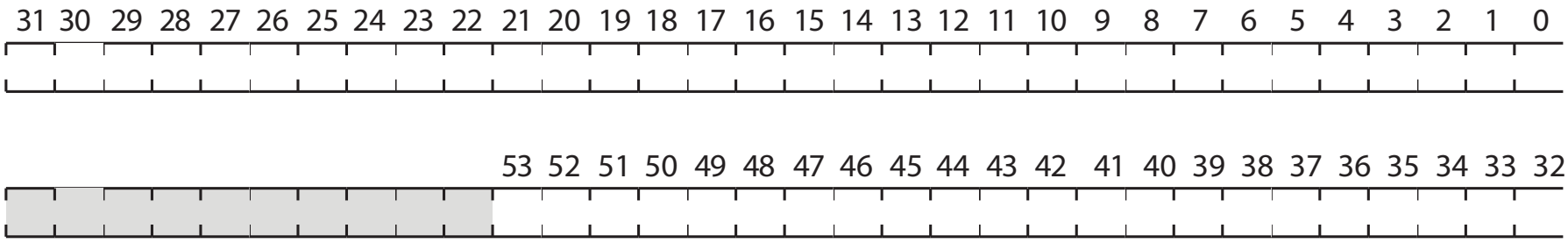
Table 6-8 – GPIO Output Set Register 0

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 *n.*	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
3. GPIO20 is bit 20 of SET0

3 Types of Instructions

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

Data Processing Instructions

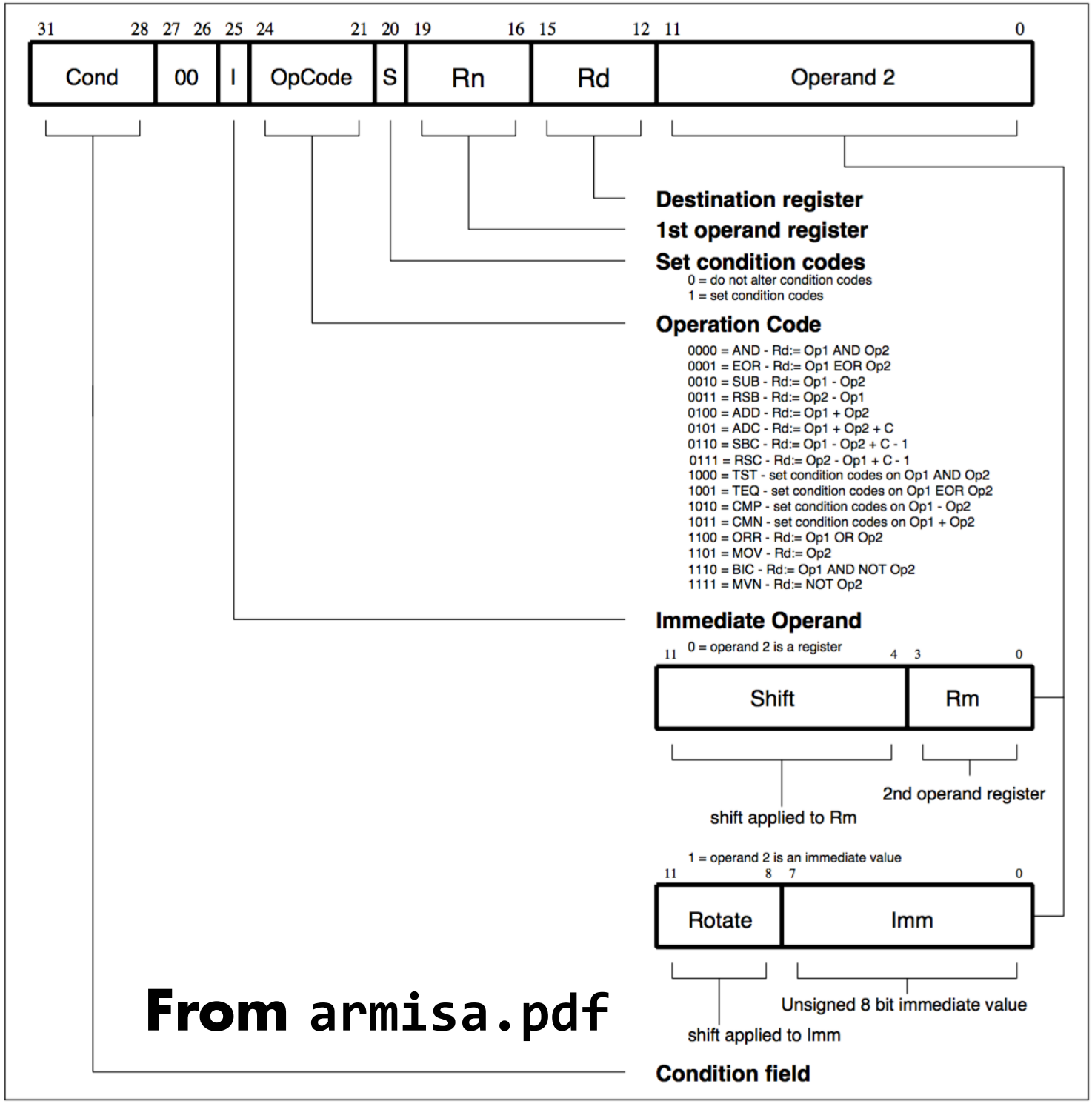
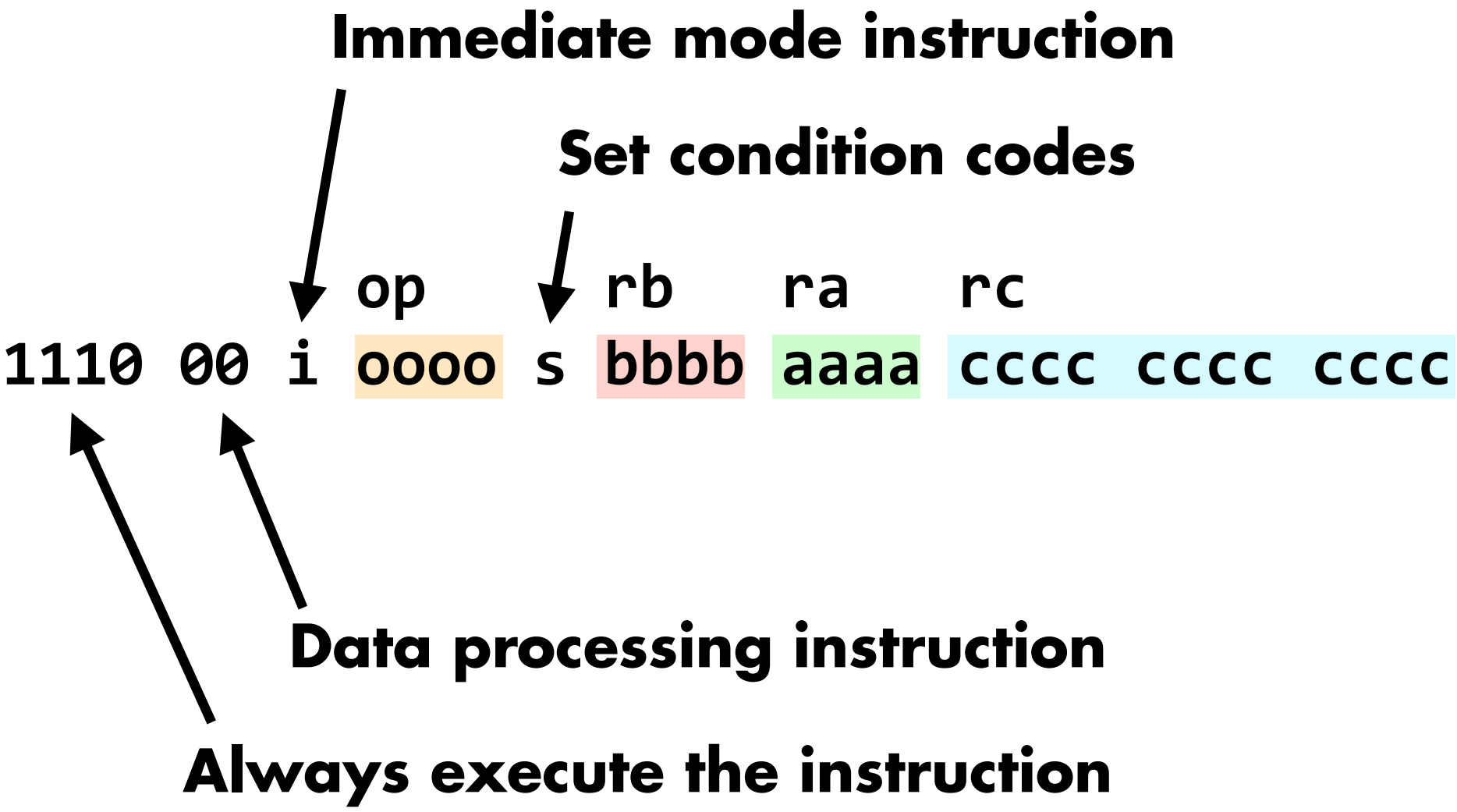


Figure 4-4: Data processing instructions

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

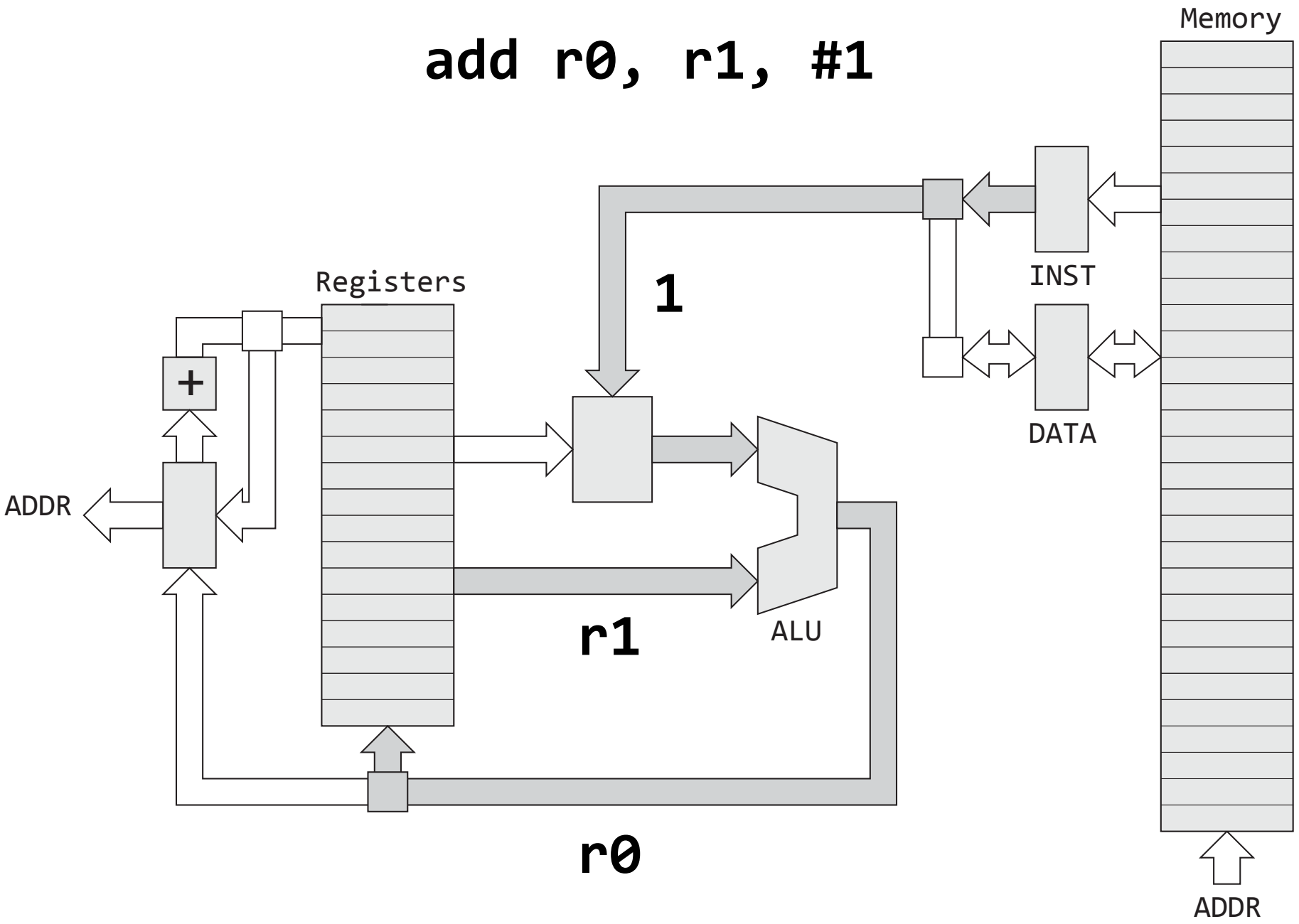


Assembly	Code	Operations
AND	0000	ra=rb&rc
EOR (XOR)	0001	ra=rb^rc
SUB	0010	ra=rb-rc
RSB	0011	ra=rc-rb
ADD	0100	ra=rb+rc
ADC	0101	ra=rb+rc+CARRY
SBC	0110	ra=rb-rc+(1-CARRY)
RSC	0111	ra=rc-rb+(1-CARRY)
TST	1000	rb&rc (ra not set)
TEQ	1001	rb^rc (ra not set)
CMP	1010	rb-rc (ra not set)
CMN	1011	rb+rc (ra 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

           add     rb   ra   rc
1110 00 i 0100 s 0000 0000 cccc cccc cccc
```

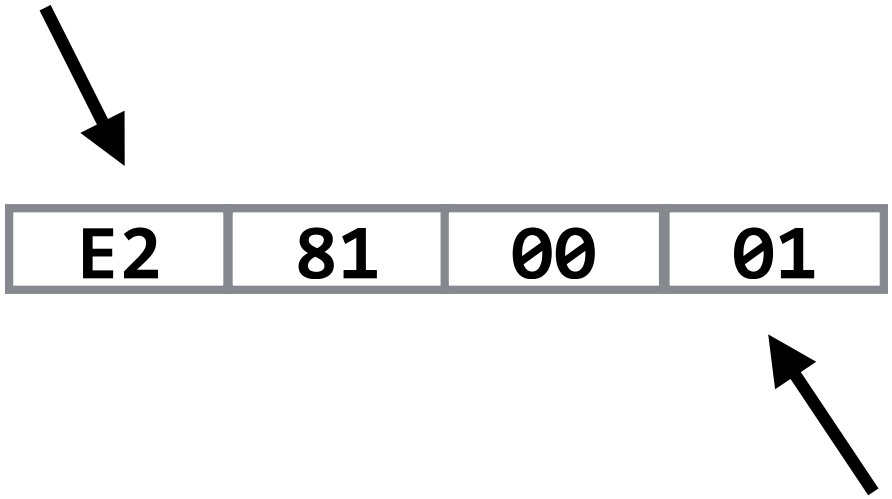
```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu

            op      rb   ra          imm
1110 00 1 0100 0 bbbb aaaa 0000 uuuu uuuu

add r0, r1, #1

# Immediate means the number is part of
# the instruction
# As in immediately available,
# i.e. no need to fetch from memory
```

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

            add     rb    ra          imm
1110 00 1 0100 0 bbbb aaaa 0000 uuuu uuuu

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

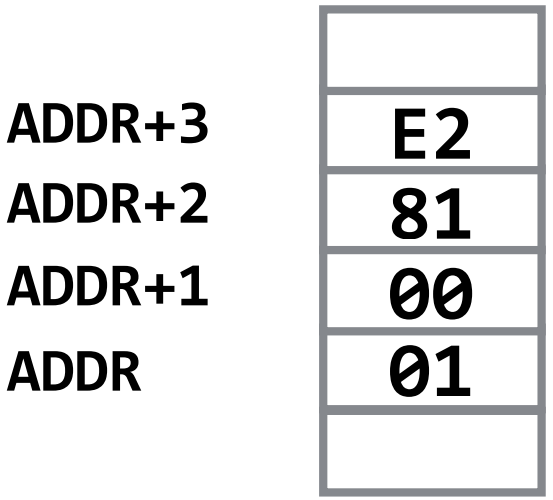

add r0, r1, #1
Memory
INST
DATA
1
Registers
+
ADDR
r1
ALU
r0
ADDR

```
# 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 0001 0000 0000 0000 0001
   E    2    8    1    0    0    0    1
```

most-significant-byte (MSB)

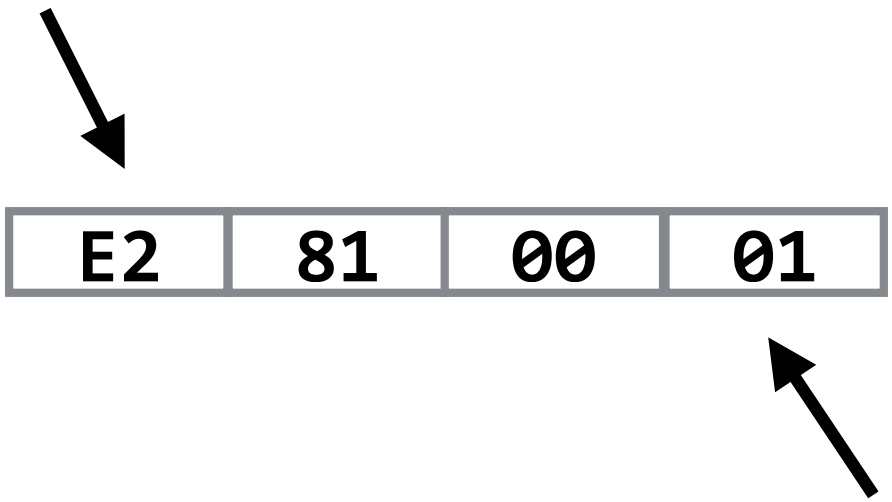


least-significant-byte (LSB)

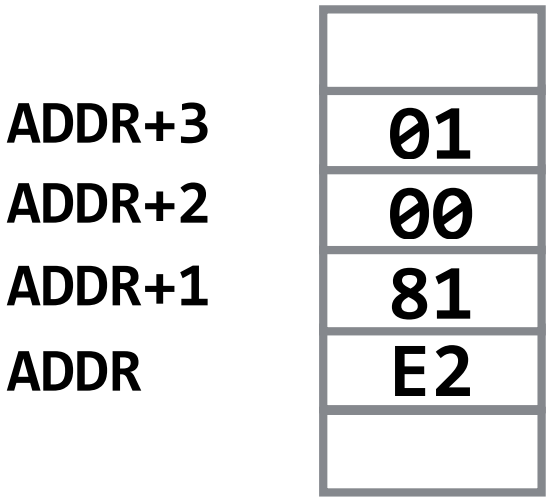


`little-endian`
`(LSB first)`

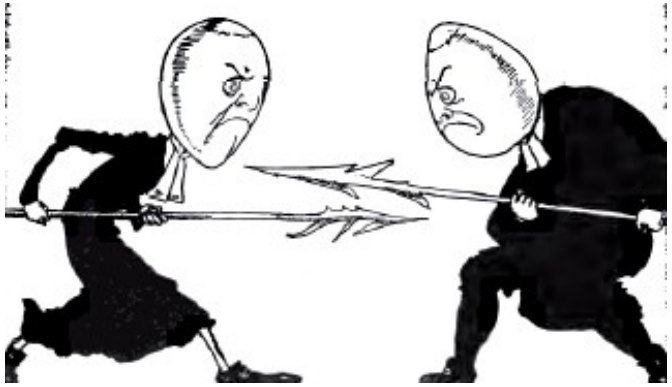
most-significant-byte (MSB)



least-significant-byte (LSB)

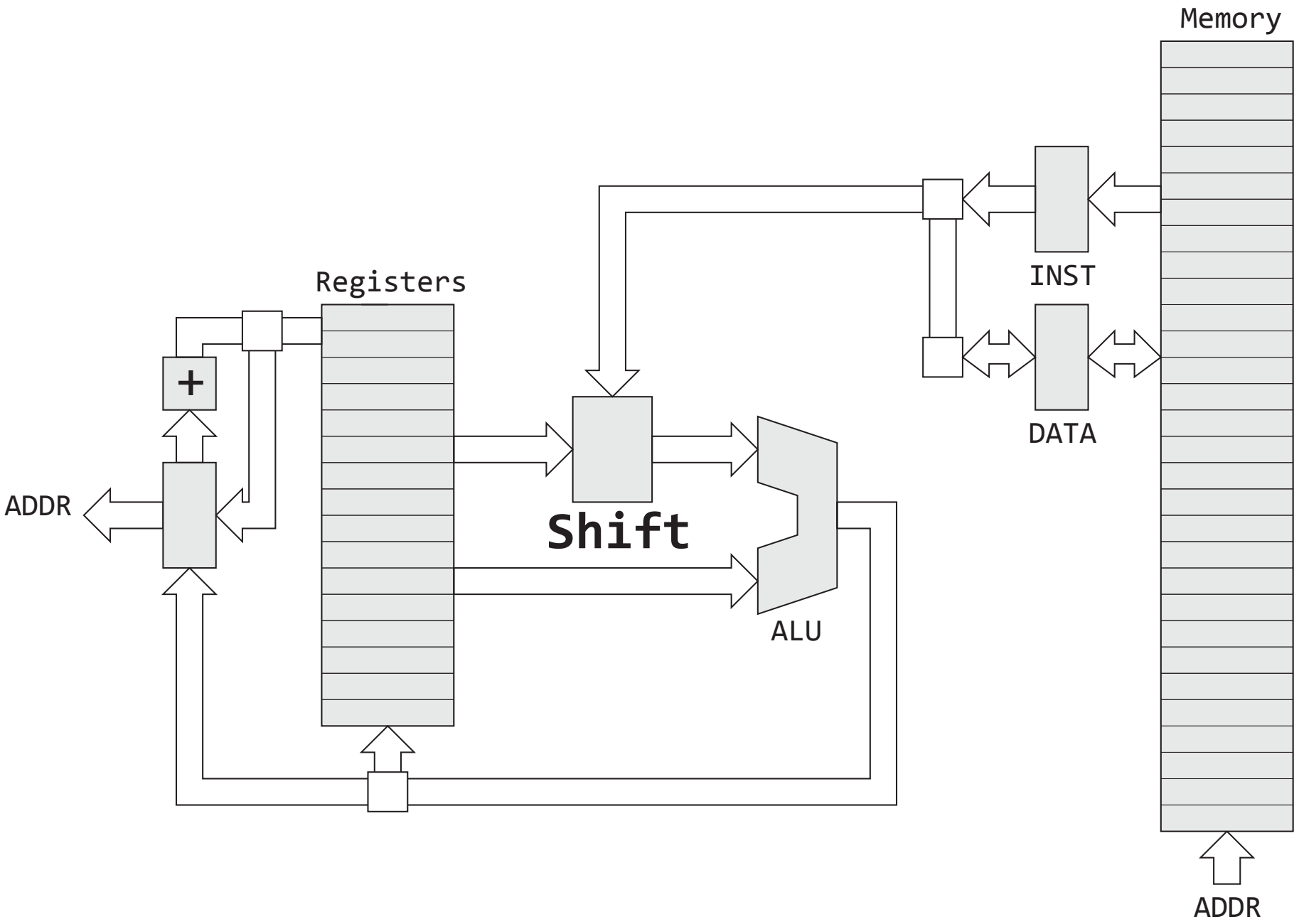


big-endian
(MSB first)

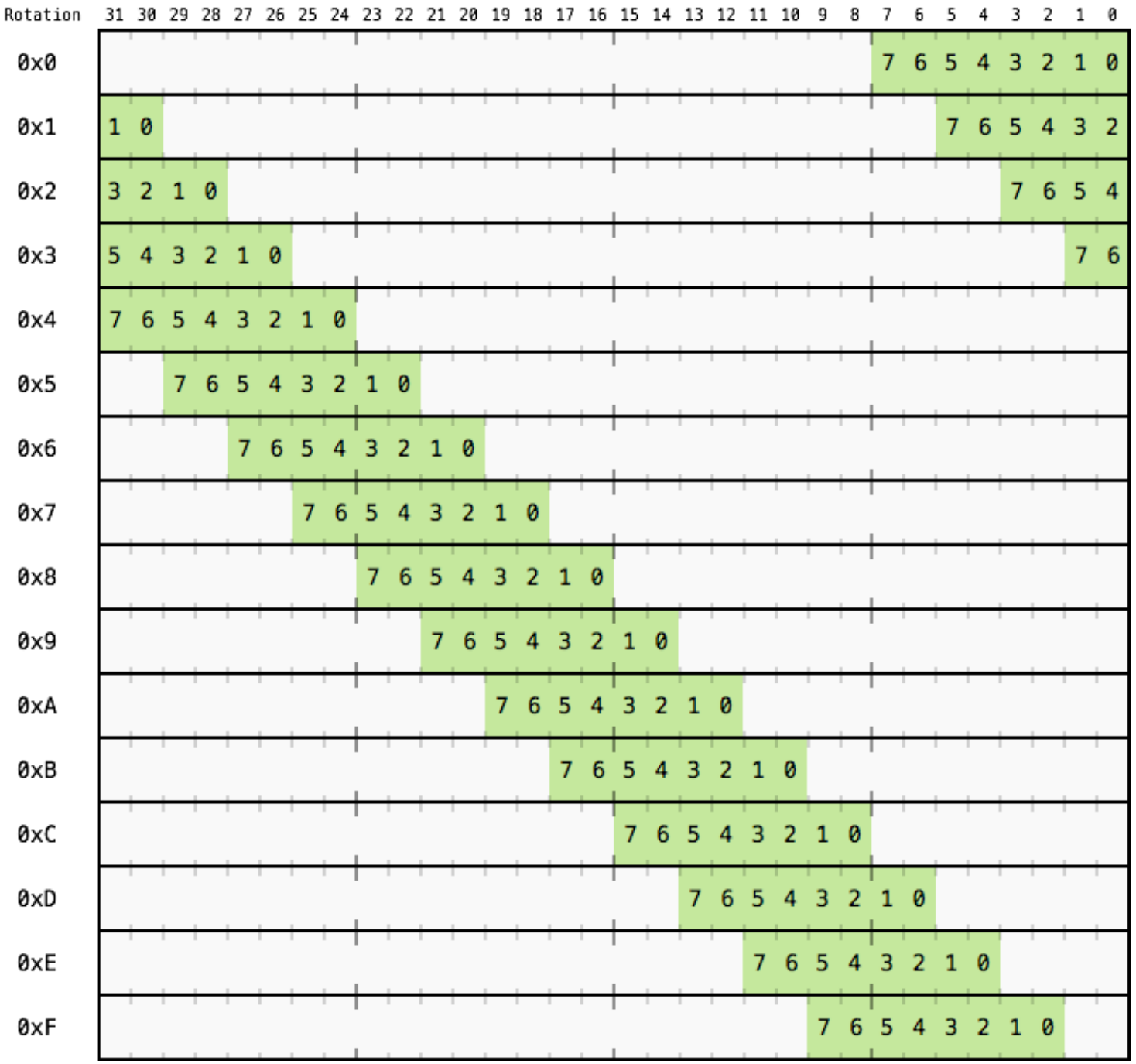


The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's Gulliver s Travels. The inhabitants of Lilliput, who are well known for being rather small, are, in addition, constrained by law to break their eggs only at the little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


Memory
INST
DATA
Registers
+
ADDR
Shift
ALU
ADDR

Rotate Bits Right (ROR)



```
# 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?
```

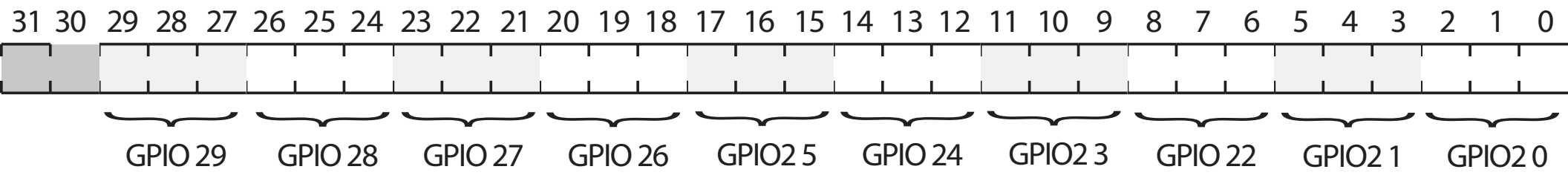
```
# data processing instruction
#  ra = rb op imm
# imm = uuuu uuuu ROR (2*iiii)

            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

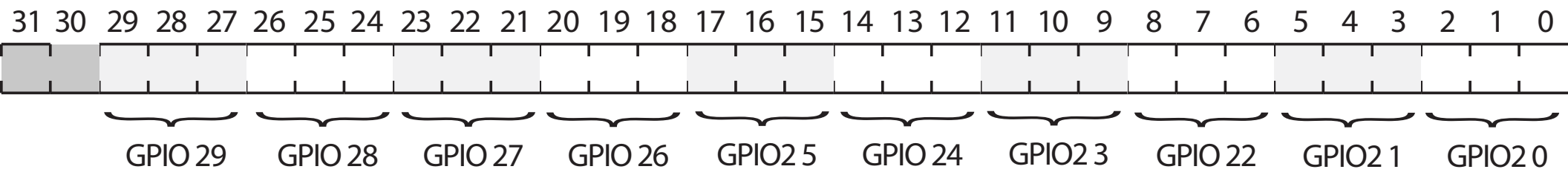
Loading a 32-bit Value into a Register

```
// Replace
#define FSEL2 0x20200008
ldr r0, =FSEL2

// with

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

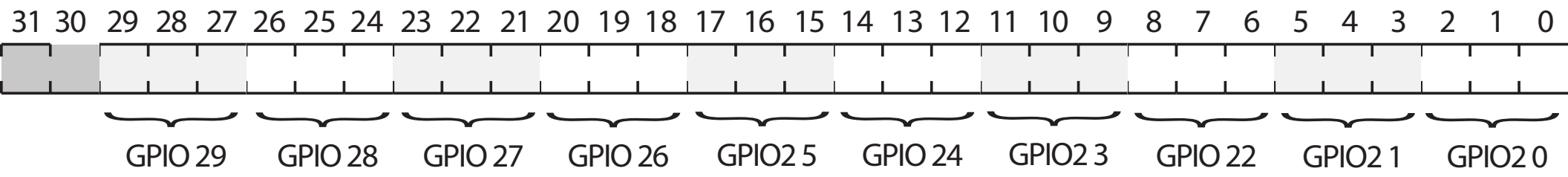
Manipulate FSEL Fields




```
// 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?
```




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

// What value is in FSEL2 now?
// What mode is GPIO 20 set to now?
// What mode is GPIO 22 set to now?
```




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

// Set GPIO 21 to OUTPUT
ldr r1, [r0]
and r1, #~(0x7<<3)
orr r1, #(0x1<<3)
str r1, [r0]

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

Blink

```
// Configure GPIO 20 for OUTPUT

loop:

  // Turn on LED

  // Turn off LED

  b loop
```

```
// Blink an LED
#define FSEL2 0x20200008
#define SET0  0x2020001C
#define CLR0  0x20200028
ldr r0, =FSEL2
mov r1, #1
str r1, [r0]

mov r1, #(1<<20)
loop:
ldr r0, =SET0
str r1, [r0]
ldr r0, =CLR0
str r1, [r0]
b loop
```

```
// Configure GPIO 20 for OUTPUT

loop:

  // Turn on LED

  // delay

  // Turn off LED

  // delay

  b loop
```

Loops and Condition Codes

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

loop:

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

    bne  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]
```

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 rc|imm
#
# i - immediate
# s – set condition code
#

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

```
# 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

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, only 1st assignment

Normally write code in C (next week)

The Fun Begins ...

Labs!!

- **Assemble Raspberry Pi Kit**
- **Lab assignments sent out before class**
- **Read lab1 instructions (now online)**
- **Install tool chain**

Assignment 1

- **Larson scanner**

Definitive References

BCM2865 peripherals document + errata

Raspberry Pi schematic

ARMv6 architecture reference manual

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