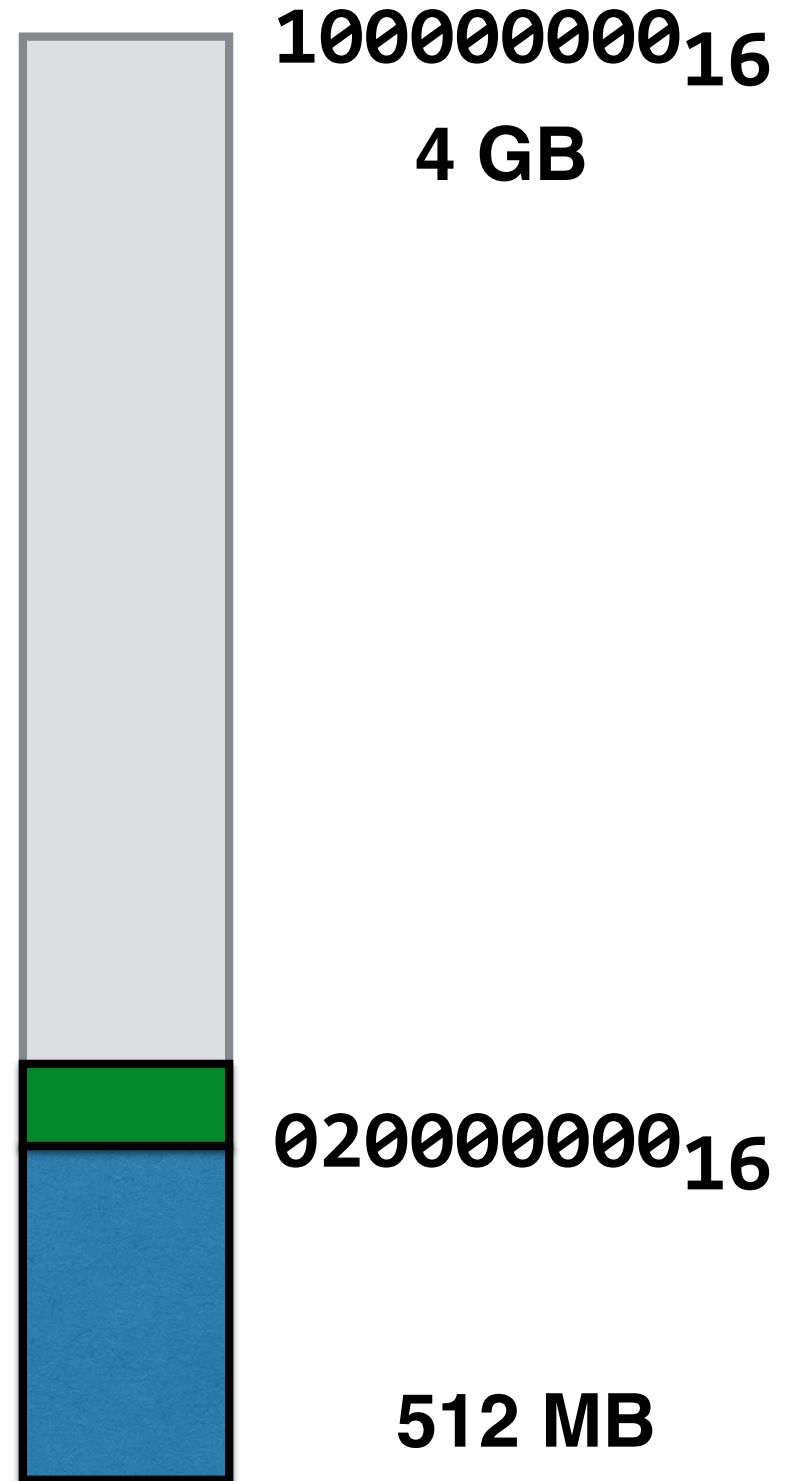# ARM

Assembly Language
and
Machine Code

Goal: Blink an LED
(and hack some binary)

Memory Map

Peripheral registers are mapped into address space

Memory-Mapped IO (MMIO)

MMIO space is above physical memory

Ref: <u>BCM2835-ARM-Peripherals.pdf</u>

100000000_{16}

4 GB

020000000_{16}

512 MB

```
// Configure GPIO20 as an OUTPUT

// FSEL2 = 0x20200008
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x00000008
mov r1, #1      // 1 indicates OUTPUT
str r1, [r0]  // store 1 to 0x20200008
```
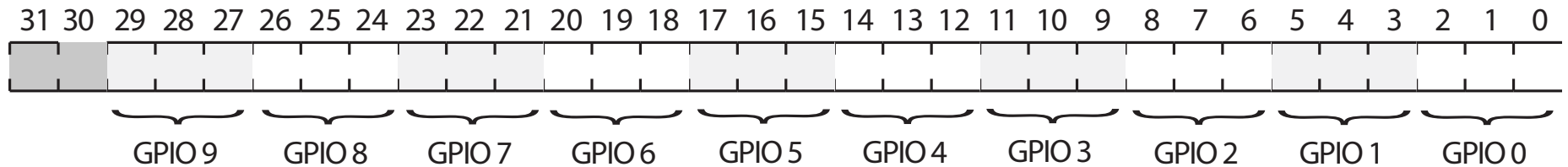
General-Purpose IO Function

GPIO Pins can be configured to be INPUT, OUTPUT, or ALT0-ALT5

Bit pattern	Pin Function
000	The pin in 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

| 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 |

GPIO 9 | GPIO 8 | GPIO 7 | GPIO 6 | GPIO 5 | GPIO 4 | GPIO 3 | GPIO 2 | GPIO 1 | GPIO 0

Function is INPUT, OUTPUT, or ALT0-ALT5

8 functions requires 3 bits to specify

10 pins per 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

Ref: BCM2835-ARM-Peripherals.pdf
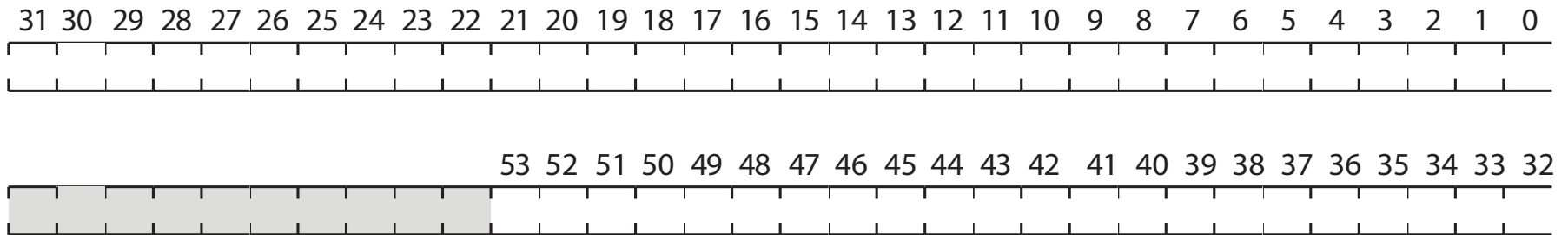
```
…
// "set" GPIO20 (output 1 = 3.3V)

//  SET0 = 0x2020001c
mov r0, #0x20000000
orr r0, #0x00200000
orr r0, #0x0000001c
mov r1, #1
lsl r1, #20  // bit 20 = 1<<20
str r1, [r0] // store 1<<20 to 0x2020001c
```

GPIO Function SET Register

```
20 20 00 1C : GPIO SET0 Register
20 20 00 20 : GPIO SET1 Register
```

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

										53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

Notes

1. 1 bit per GPIO pin
2. 54 pins requires 2 registers

GPIO Pin Output Set Registers (GPSETn)
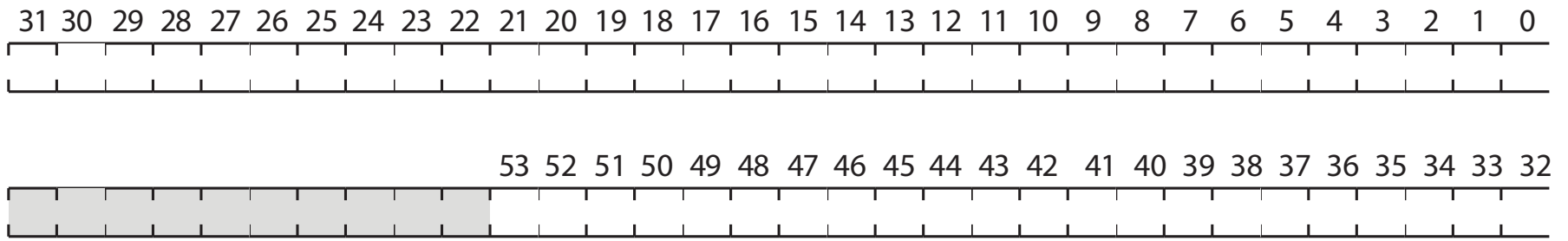
SYNOPSIS   The output set registers are used to set a GPIO pin. The SET{n} field defines the
respective GPIO pin to set, writing a "0" to the field has no effect. If the GPIO pin is
being used as in input (by default) then the value in the SET{n} field is ignored.
However, if the pin is subsequently 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

```
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
```

```
53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
```

```
ldr r0, SET0
ldr r1, #(1<<20)
ldr r1, #(1<<21)

// Is the LED connected to GPIO20 on?
// Is the LED connected to GPIO21 on?
```

3 Types of Instructions

1. Data processing instructions

2. Loads from and stores to memory

3. Conditional branches to new program locations
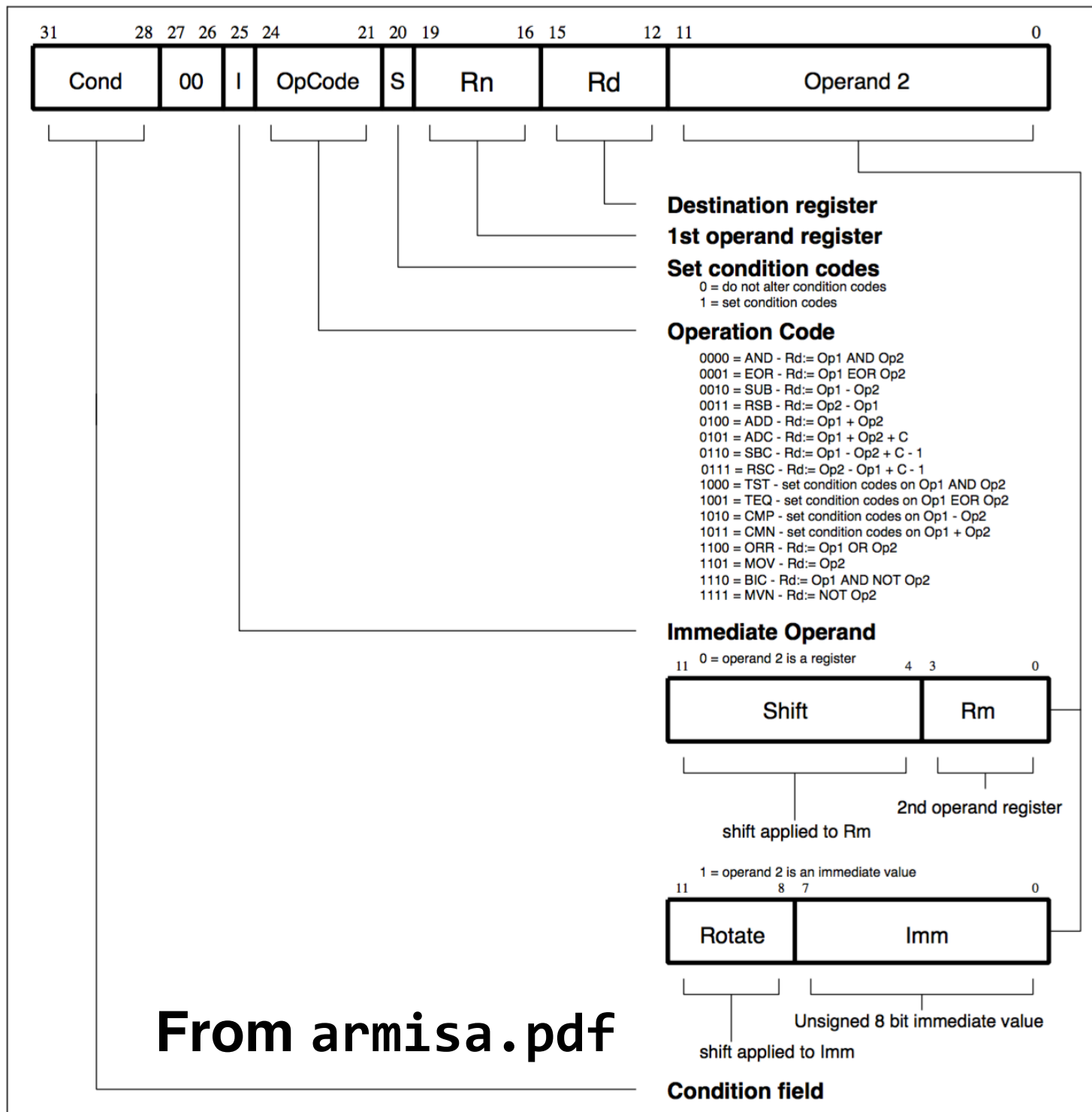
Data Processing Instructions

From `armisa.pdf`

Figure 4-4: Data processing instructions

The figure shows the bit field layout for data processing instructions:

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 ... 0
Cond	00	I	OpCode	S	Rn	Rd	Operand 2

Destination register

1st operand register

Set condition codes
0 = do not alter condition codes
1 = set condition codes

Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

Immediate Operand

11   0 = operand 2 is a register   4 3   0

Shift	Rm

shift applied to Rm

2nd operand register

1 = operand 2 is an immediate value
11   8 7   0

Rotate	Imm

shift applied to Imm

Unsigned 8 bit immediate value

Condition field

```
# 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

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       r1      r0
1110 00 i 0100 s 0001 0000 cccc cccc cccc
```

add r0, r1, #1
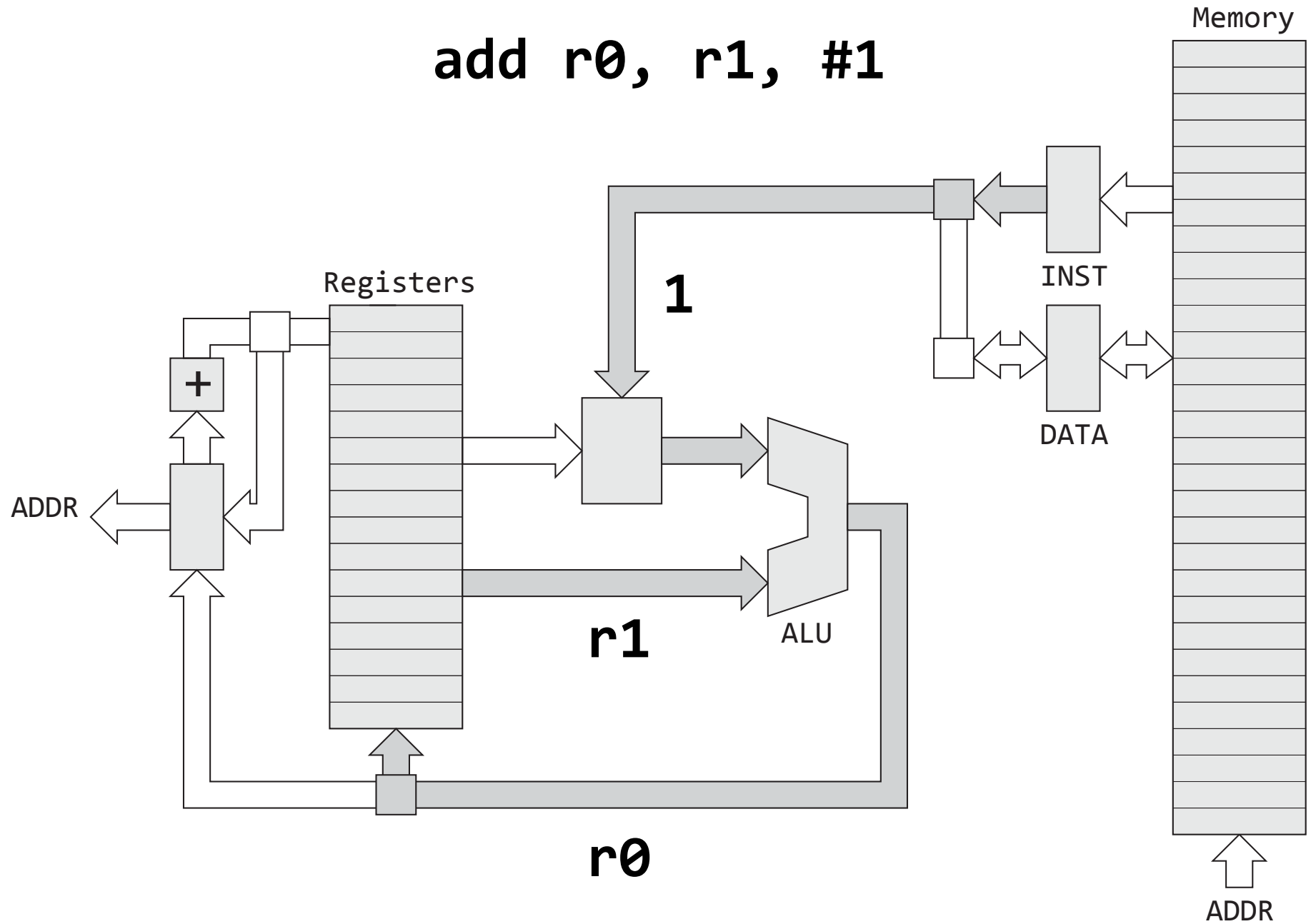
```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu


          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
```
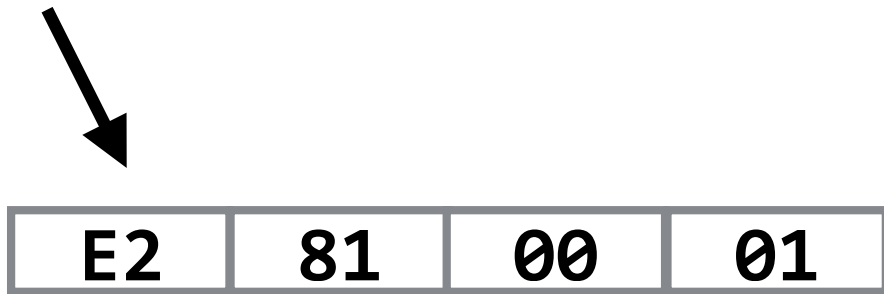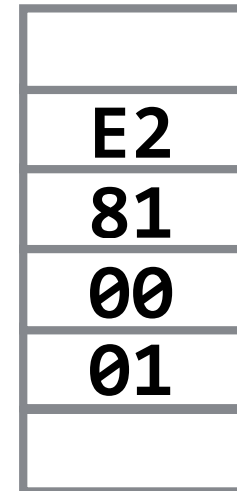
most-significant-byte (MSB)

E2	81	00	01

least-significant-byte (LSB)

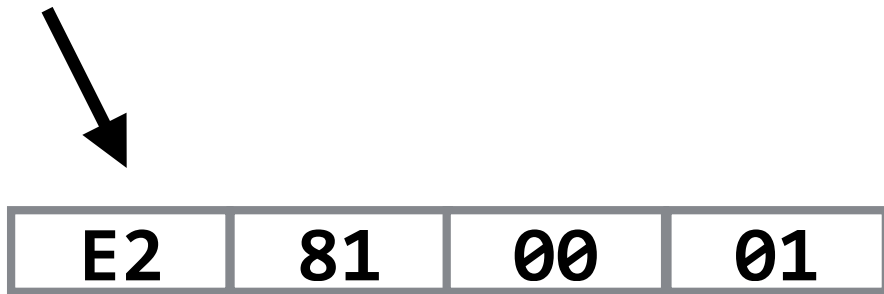|	
ADDR+3	E2
ADDR+2	81
ADDR+1	00
ADDR	01

**little-endian
(LSB first)**

Intel is little-endian
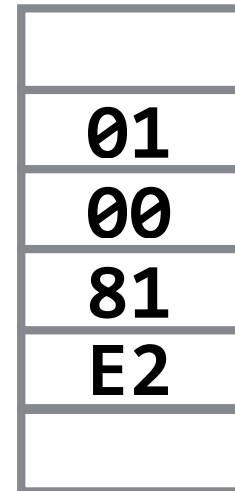ARM can do either, Pi configured to be little endian

most-significant-byte (MSB)

E2	81	00	01

least-significant-byte (LSB)

ADDR+3	01
ADDR+2	00
ADDR+1	81
ADDR	E2

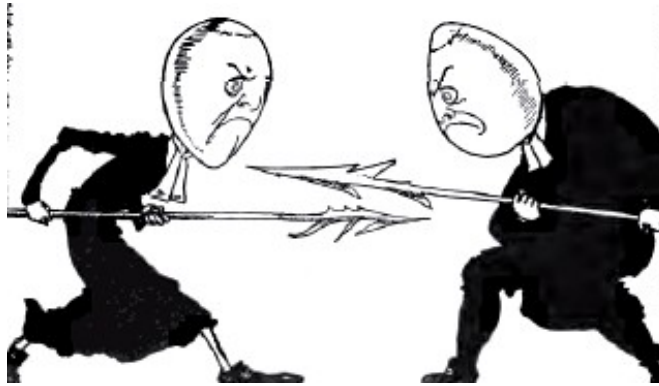**big-endian (MSB first)**

Old Macintoshes, Sun SPARC are big endian
Networks use big endian

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

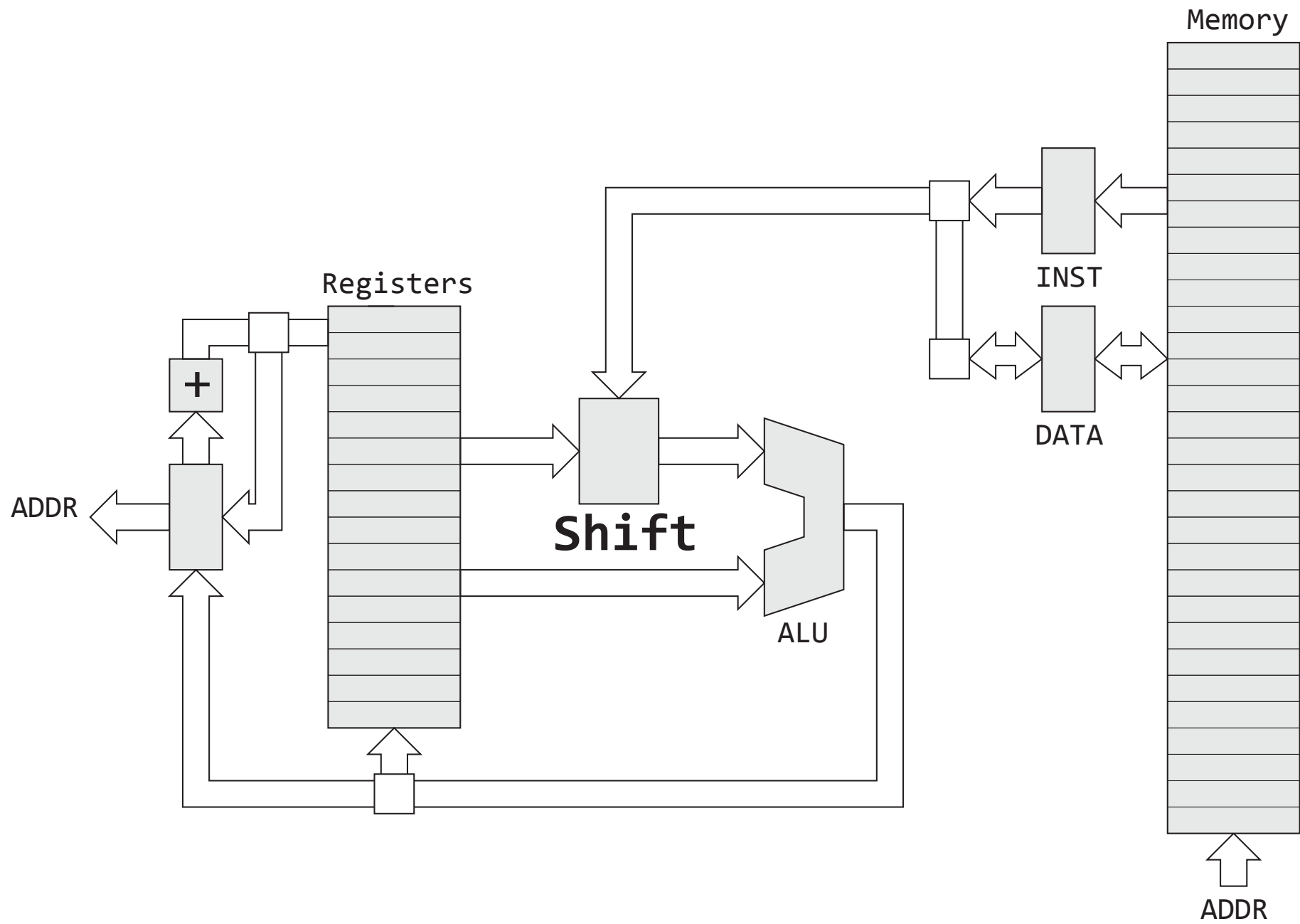# The Fun Begins ...

Labs1

- Assemble Raspberry Pi Kit

- Lab assignments sent out tonight

- Read lab1 instructions (now online)

- Install tool chain

Assignment 1

- Larson scanner

Memory

INST

DATA

Registers

Shift

ALU

ADDR

+

ADDR

Rotate Bits 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*rrrr)


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

```
# 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


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

...

```
//  SET1 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0
```
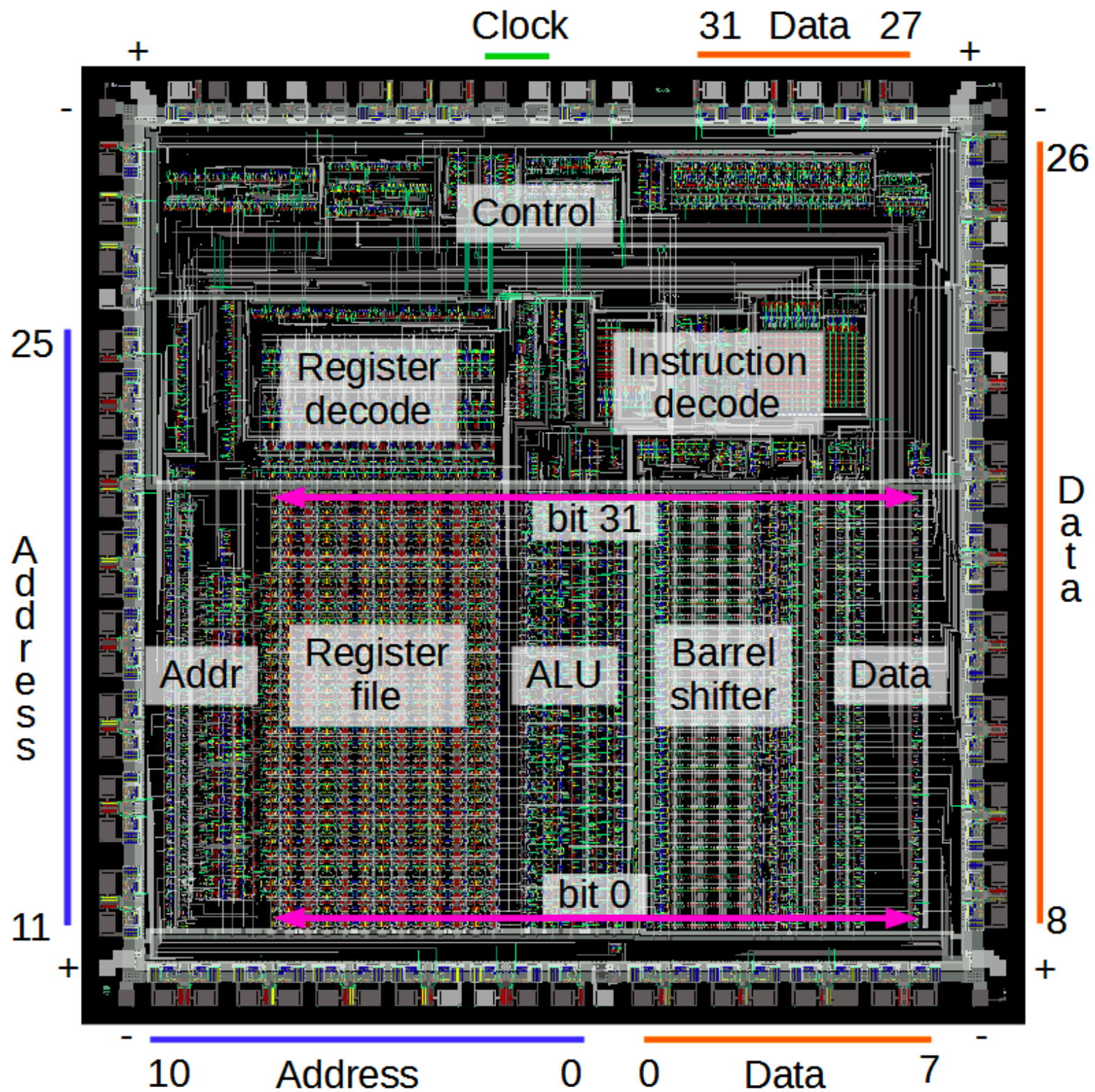
```
# data processing instruction
#  ra = rb op imm
# imm = (uuuu uuuu) ROR (2*iiii)


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


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?

hint:

Assembly	Code	Operations
SUB	0010	ra=rb-rc

Answer:

op (oooo) = 0010
rb (bbbb) = 0101 (r5)
ra (aaaa) = 0111 (r7)
iiii = 1100
uuuu uuuu = 0000 0011

1110 0010 0100 0101 0111 1100 0000 0011 = 0xE2457C03

```
# 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


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
```
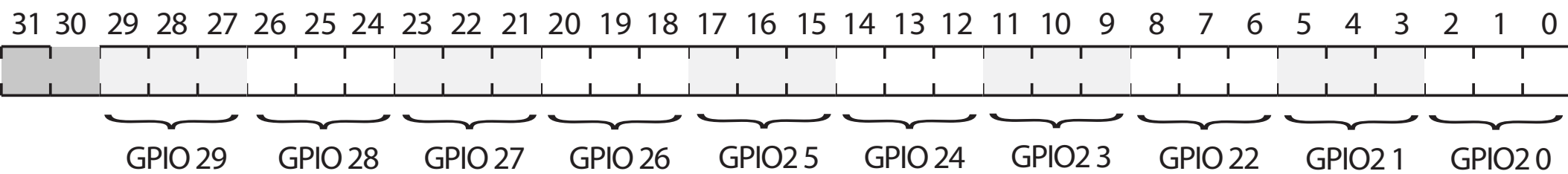
Manipulating Bit Fields

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

GPIO 29 | GPIO 28 | GPIO 27 | GPIO 26 | GPIO 25 | GPIO 24 | GPIO 23 | GPIO 22 | GPIO 21 | GPIO 20

```
// 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 21 set to now?
```

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

GPIO 29 · GPIO 28 · GPIO 27 · GPIO 26 · GPIO25 · GPIO 24 · GPIO23 · GPIO 22 · GPIO21 · GPIO20
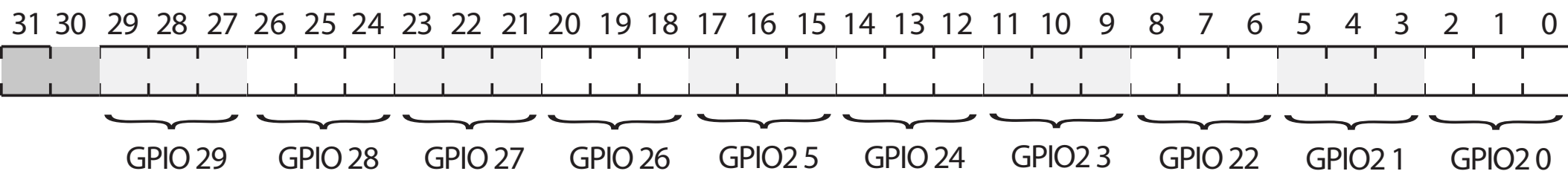
```
// Set GPIO 20 to OUTPUT
mov r1, #1
str r1, [r0]

// 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?
```

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

GPIO 29   GPIO 28   GPIO 27   GPIO 26   GPIO2 5   GPIO 24   GPIO2 3   GPIO 22   GPIO2 1   GPIO2 0
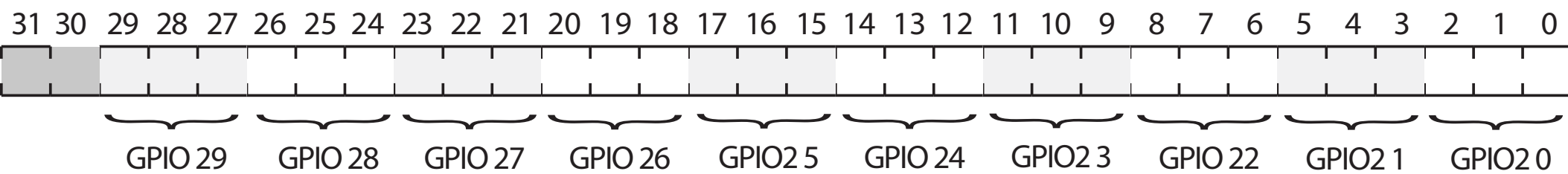
```
// Set GPIO 20 to OUTPUT
mov r1, #1
str r1, [r0]
…


// 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?
```

```
// LDR FSEL2, GPIO20 is OUTPUT
ldr r1, [r0]
0000 0000 0000 0000 0000 0000 0000 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 0000 0000 0000 0000 0000 0000 1001
```
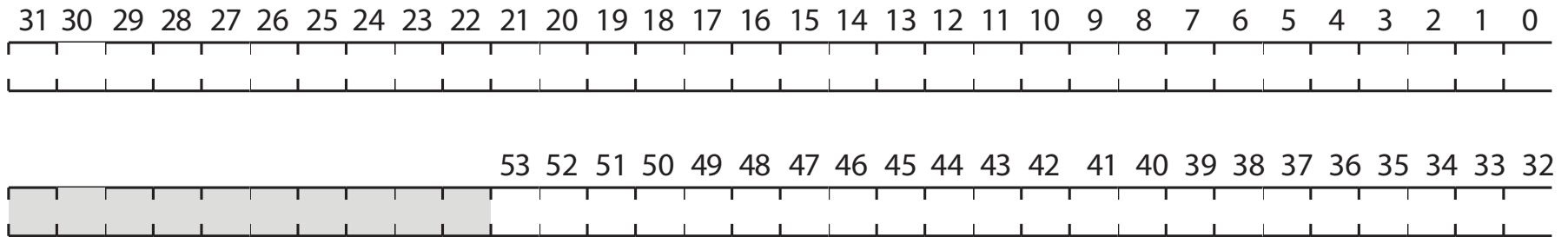
Blink

```
// This is tedious …

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

// Alternative is to use ldr
ldr r0, FSEL2

FSEL2: .word 0x20200008
```

```
 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
┌──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘

                          53 52  51 50  49 48  47 46  45 44  43 42  41 40  39 38  37 36  35 34  33 32
┌──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘
```

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

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

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

// Why SET and CLR?
```

```
// 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
```

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
```

Branch Instructions

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

```
# 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
```

The "link" bit — you'll cover this later, when you learn about how functions work

Bringing It All Together

```
$ cpp -P blink.s | arm-none-eabi-as -o blink.o
$ arm-none-eabi-objcopy blink.o -O binary blink.bin
$ rpi-install blink.bin

$ xxd blink.bin > blink.dump
// edit blink.dump
$ xxd -r blink.dump > blink-fast.bin

$ rpi-install blink-fast.bin
```

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

Rarely, but sometimes, write assembly

Understanding how processors represent and execute instructions lets you understand how languages work

Reading assembly allows you to *really* know what the processor is doing: we'll see how compilers can trick you!

Finite space leads to tradeoffs and careful design: what if ARM had 32 registers?

Definitive References

BCM2865 peripherals document + errata

Raspberry Pi schematic

ARMv6 architecture reference manual

see Resources on cs107e.github.io