

Data Processing Instructions and Machine Code

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	-	-

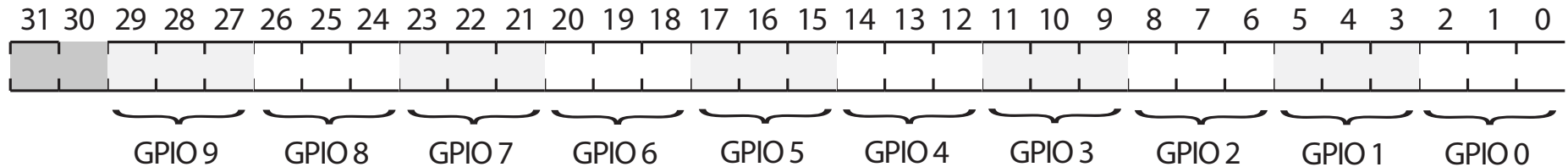
Watch out ...

Manual says: 0x7E200000

Replace 7E with 20: 0x20200000

Ref: BCM2835-ARM-Peripherals.pdf

GPIO Function Select



Bit pattern	Pin Function
000	The pin is 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

**GPIO Pins can be configured to be
INPUT, OUTPUT, or ALT0-5**

3 bits are used to select function

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 <i>n</i>	R/W	0

Table 6-8 – GPIO Output Set Register 0

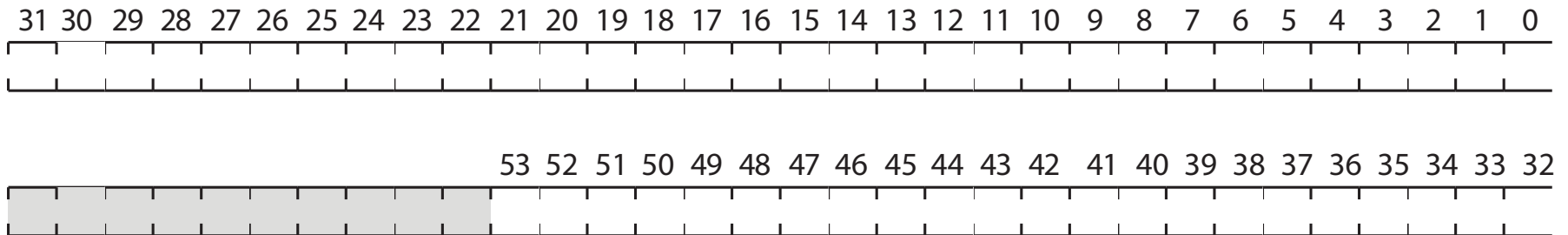
Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin <i>n</i> .	R/W	0

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**

Data Processing Instructions and Machine Code

3 Types of Instructions

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

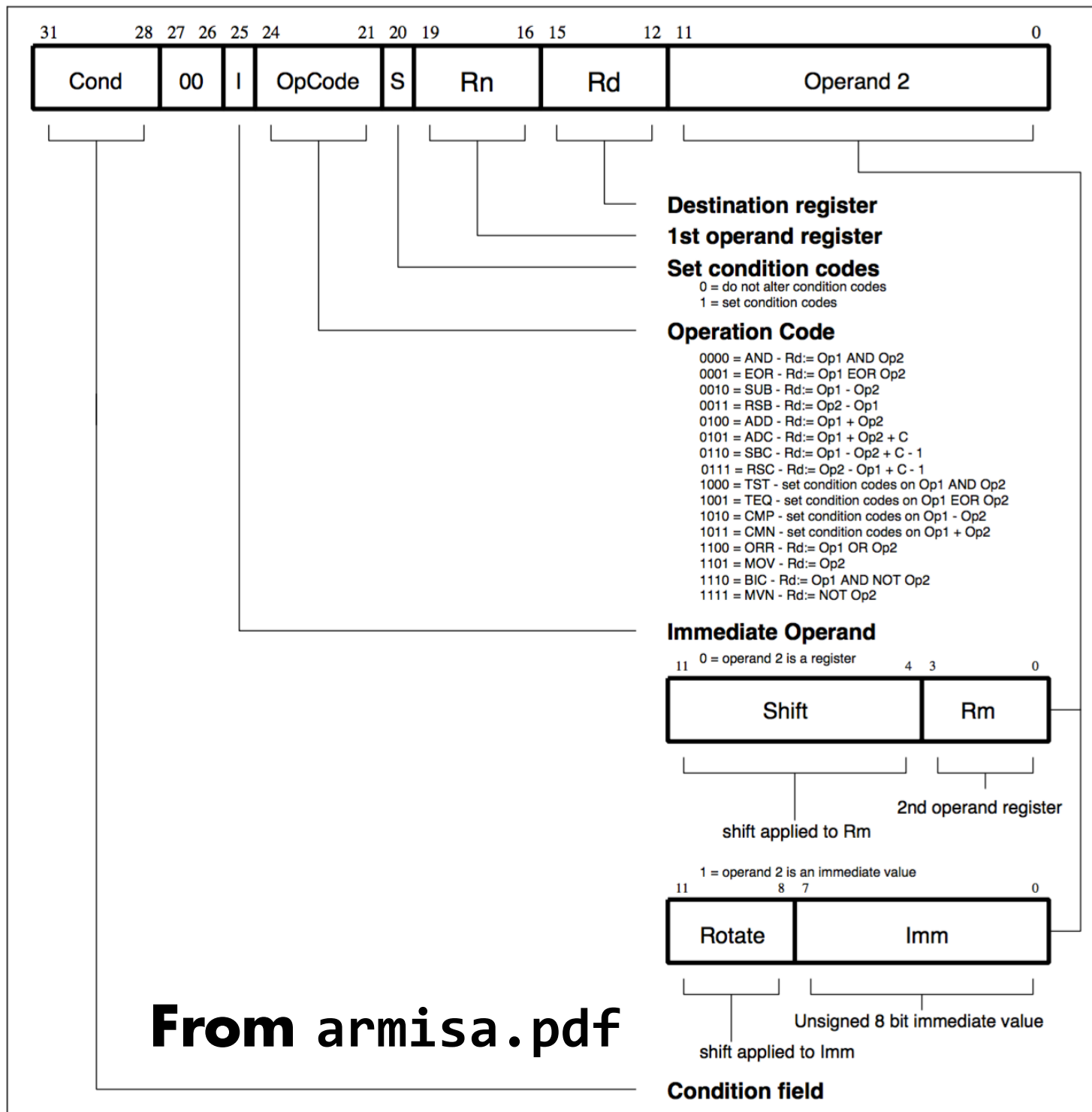


Figure 4-4: Data processing instructions

data processing instruction

#

ra = rb op rc

Immediate mode instruction

Set condition codes

			op		rb	ra	rc
1110	00	i	0000	s	bbbb	aaaa	cccc cccc cccc

Data processing instruction

Always execute the instruction

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

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

			op		rb	ra	rc		
1110	00	i	0000	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

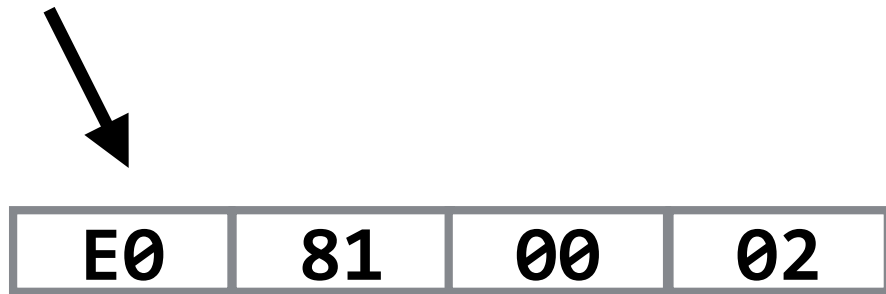
# Create binary (.bin)
% arm-none-eabi-objcopy add.o -O binary add.bin

# Find size (in bytes)
% ls -l add.bin
-rw-r--r--+ 1 hanrahan  staff  4 add.bin

# Dump binary in hex
% hexdump add.bin
00000000: 02 00 81 e0
```

32-bit word consists of 4 consecutive bytes

most-significant-byte (MSB)



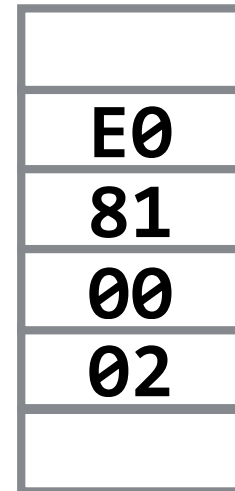
least-significant-byte (LSB)

ADDR+3

ADDR+2

ADDR+1

ADDR

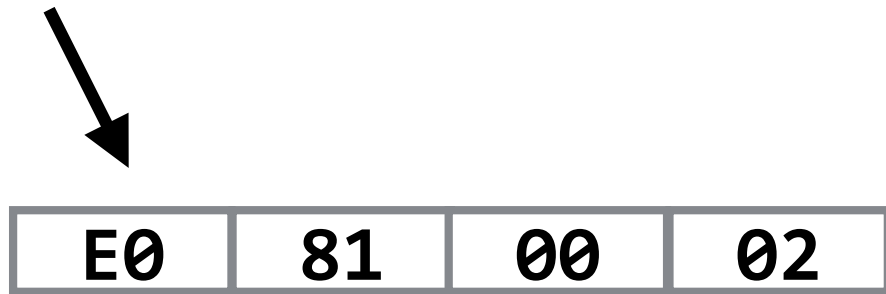


**little-endian
(LSB first)**

ARM uses little-endian

32-bit word consists of 4 consecutive bytes

most-significant-byte (MSB)



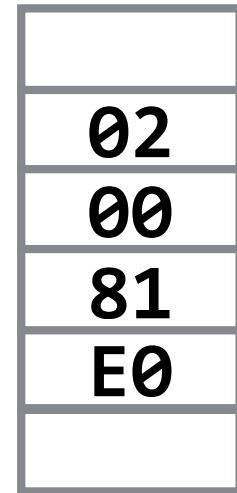
least-significant-byte (LSB)

ADDR+3

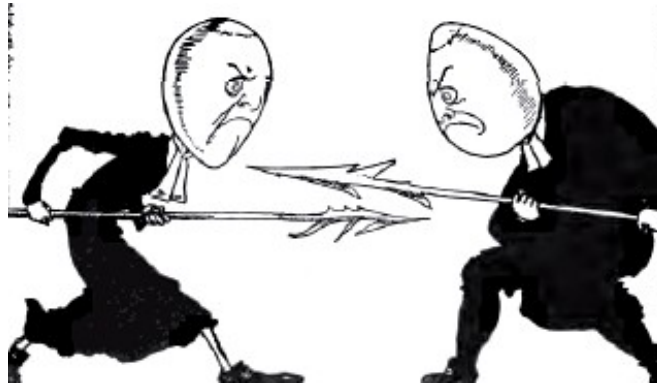
ADDR+2

ADDR+1

ADDR



**big-endian
(MSB first)**

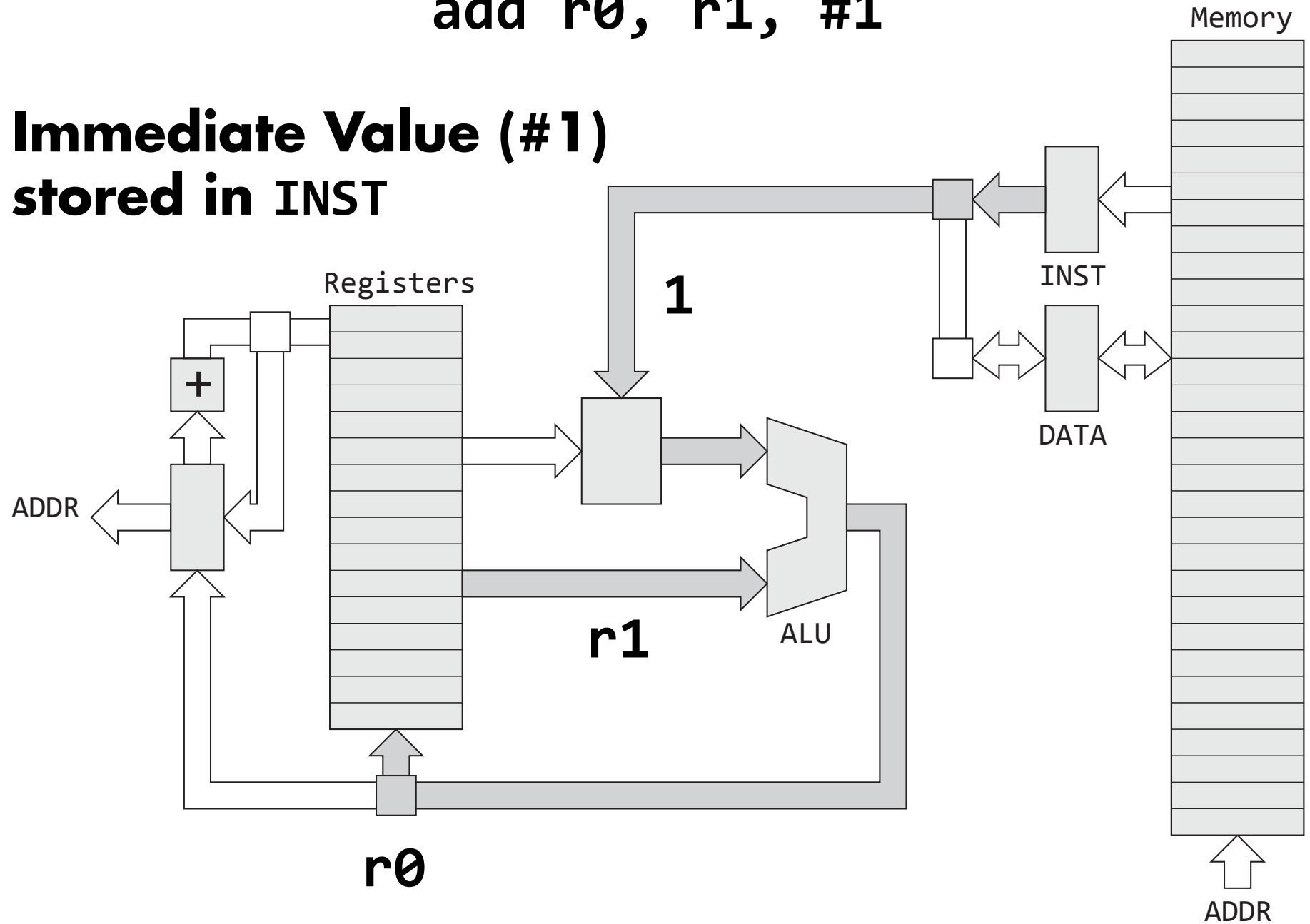


The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's Gulliver s Travels. The inhabitants of Lilliput, who are well known for being rather small, are, in addition, constrained by law to break their eggs only at the little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

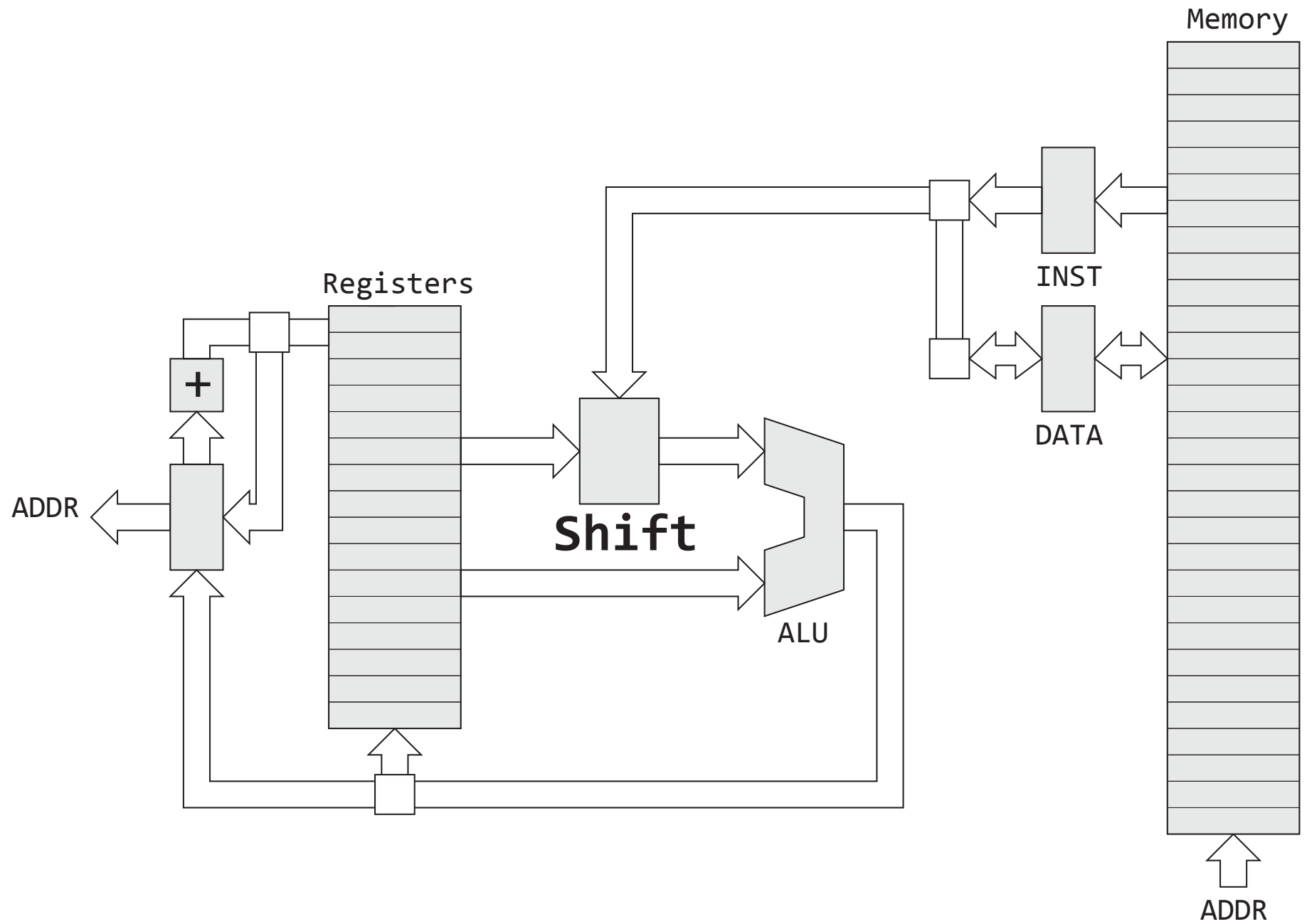
#imm = uuuu uuuu

			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



Rotate Right (ROR) - Rotation amount = 2x

[illegible]

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

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

ROR means *Rotate Right* ($\text{imm} \gg \text{rotate}$)



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

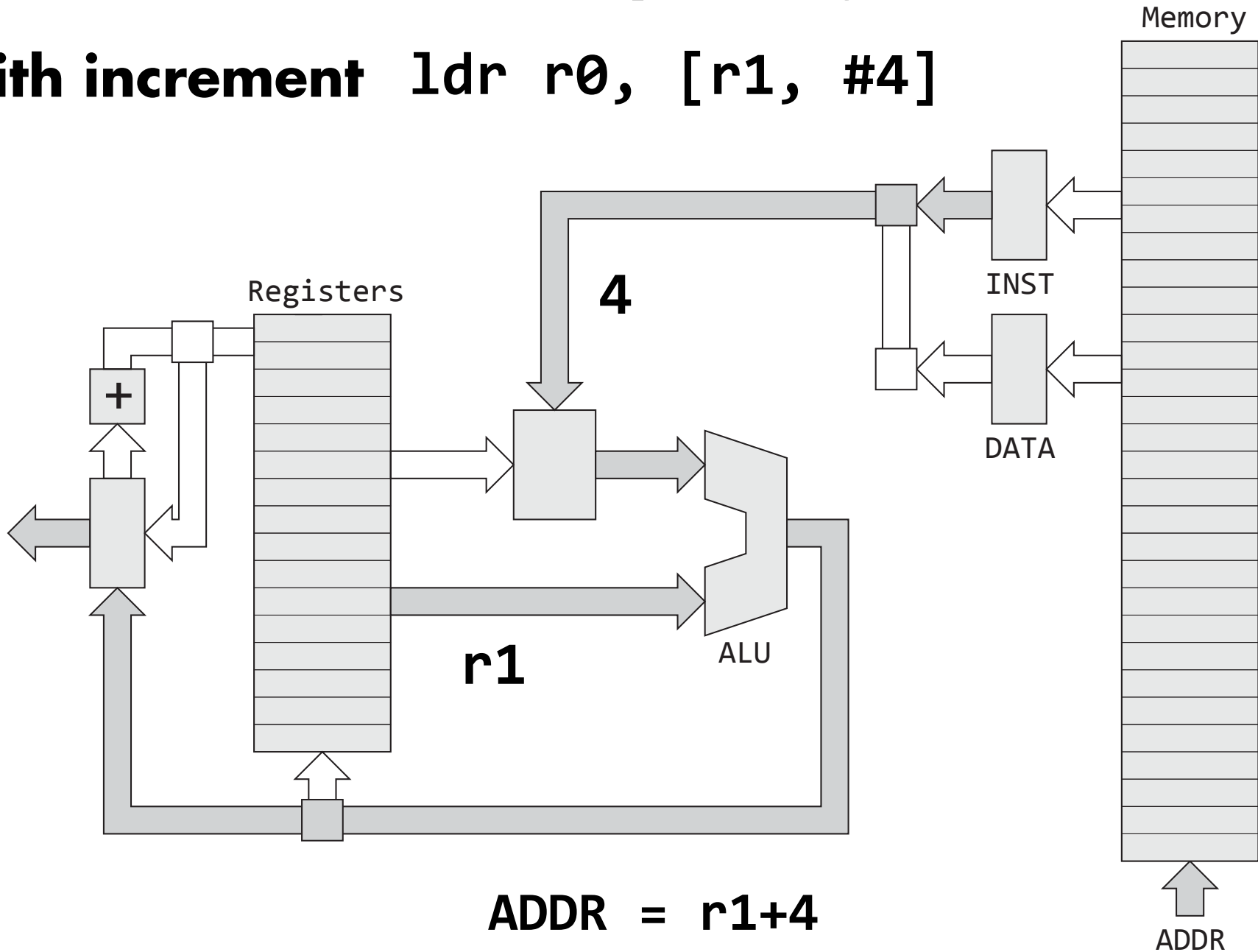
1110	0010	0100	0101	0111	1100	0000	0011
E	2	4	5	7	C	0	3

...

```
// 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]`



```
// 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
```

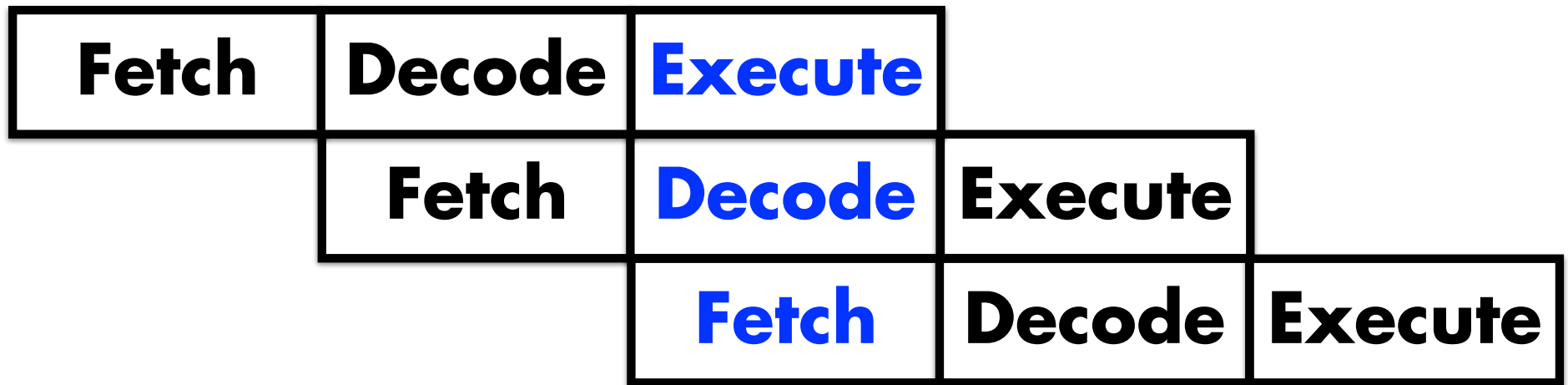
3 steps to run an instruction



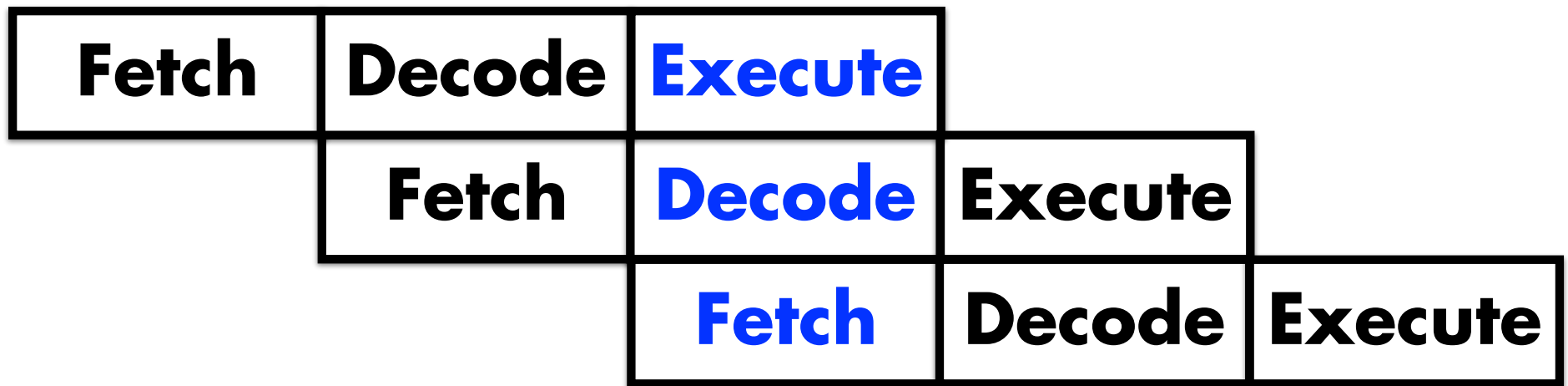
3 instructions takes 9 steps



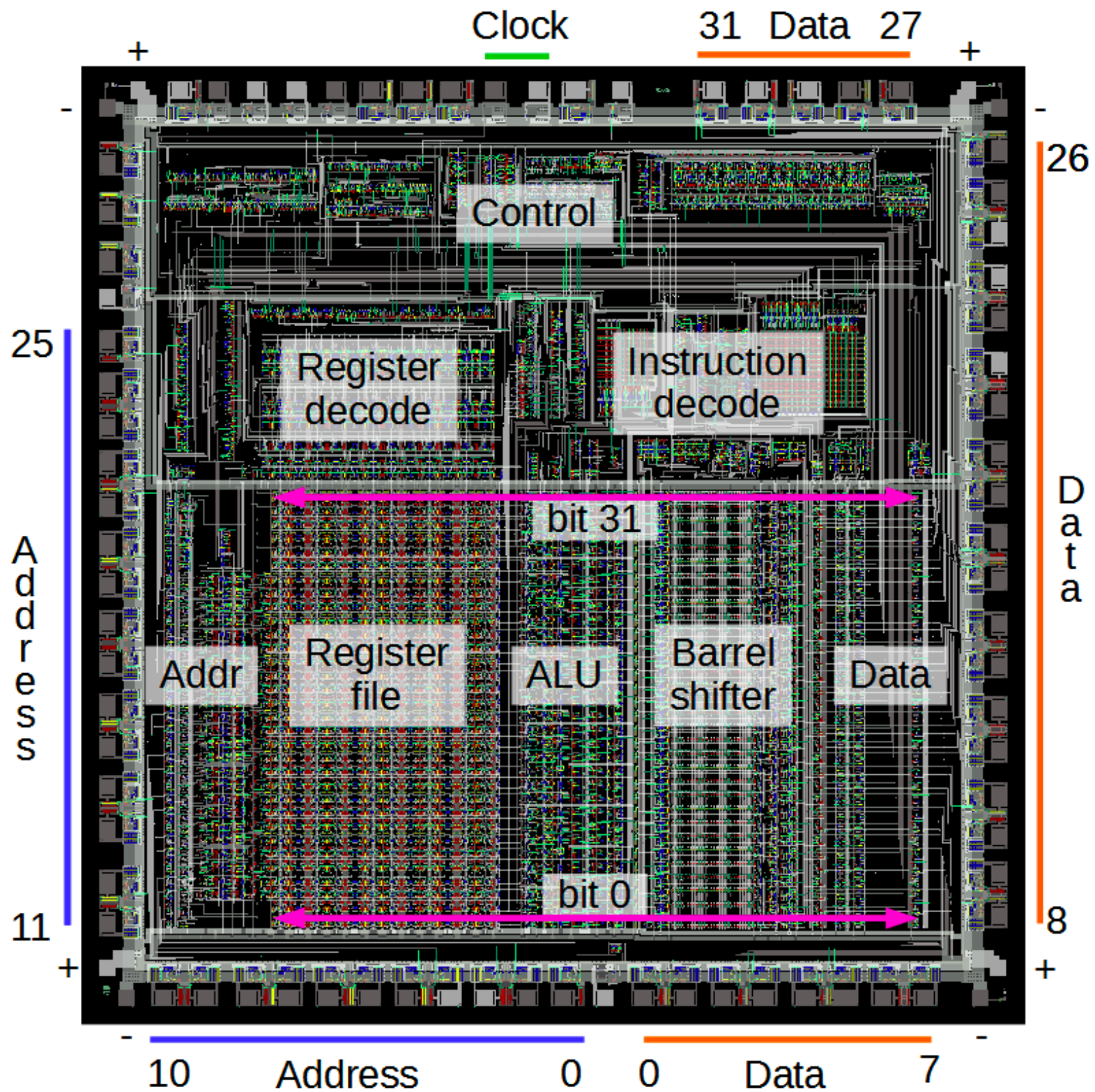
**To speed things up,
steps are overlapped ("pipelined")**



**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)**





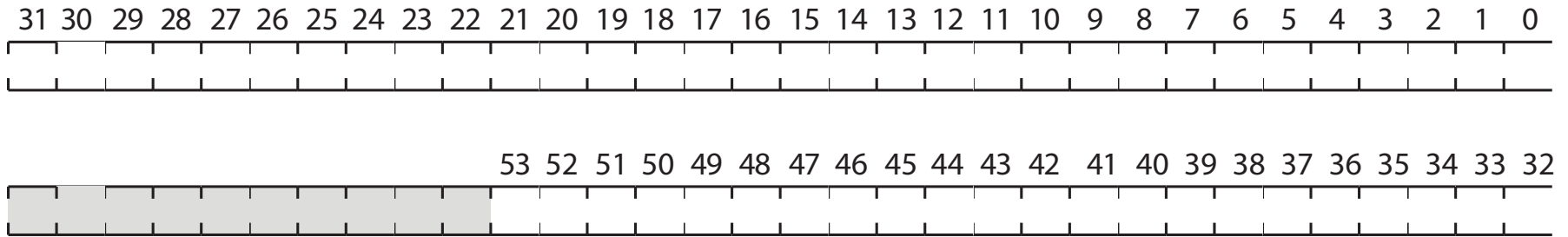
Prof. Stephen B.
Furber



Ms. Sophie M.
Wilson

**Principal Designers
of the
ARM Architecture and Processors**

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 11am-12noon**

Definitive References

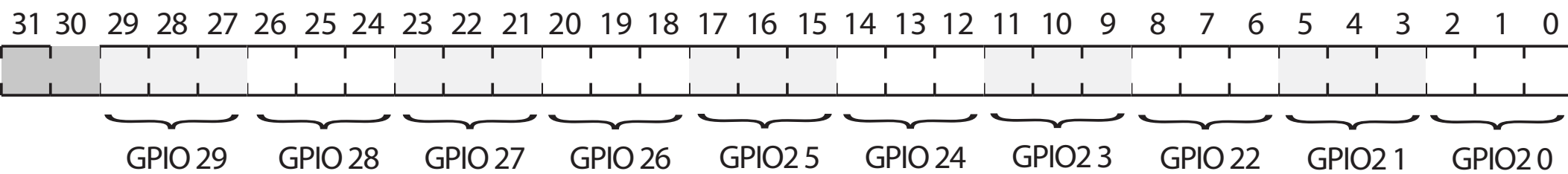
BCM2865 peripherals document + errata

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)

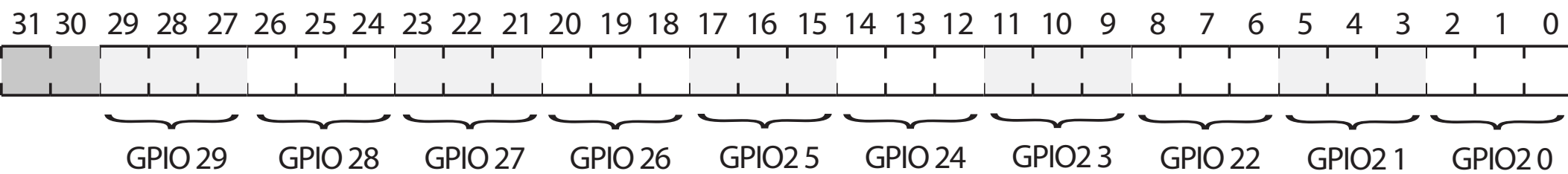
1111 1111 1111 1111 1111 1111 1100 0111

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?