

**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**

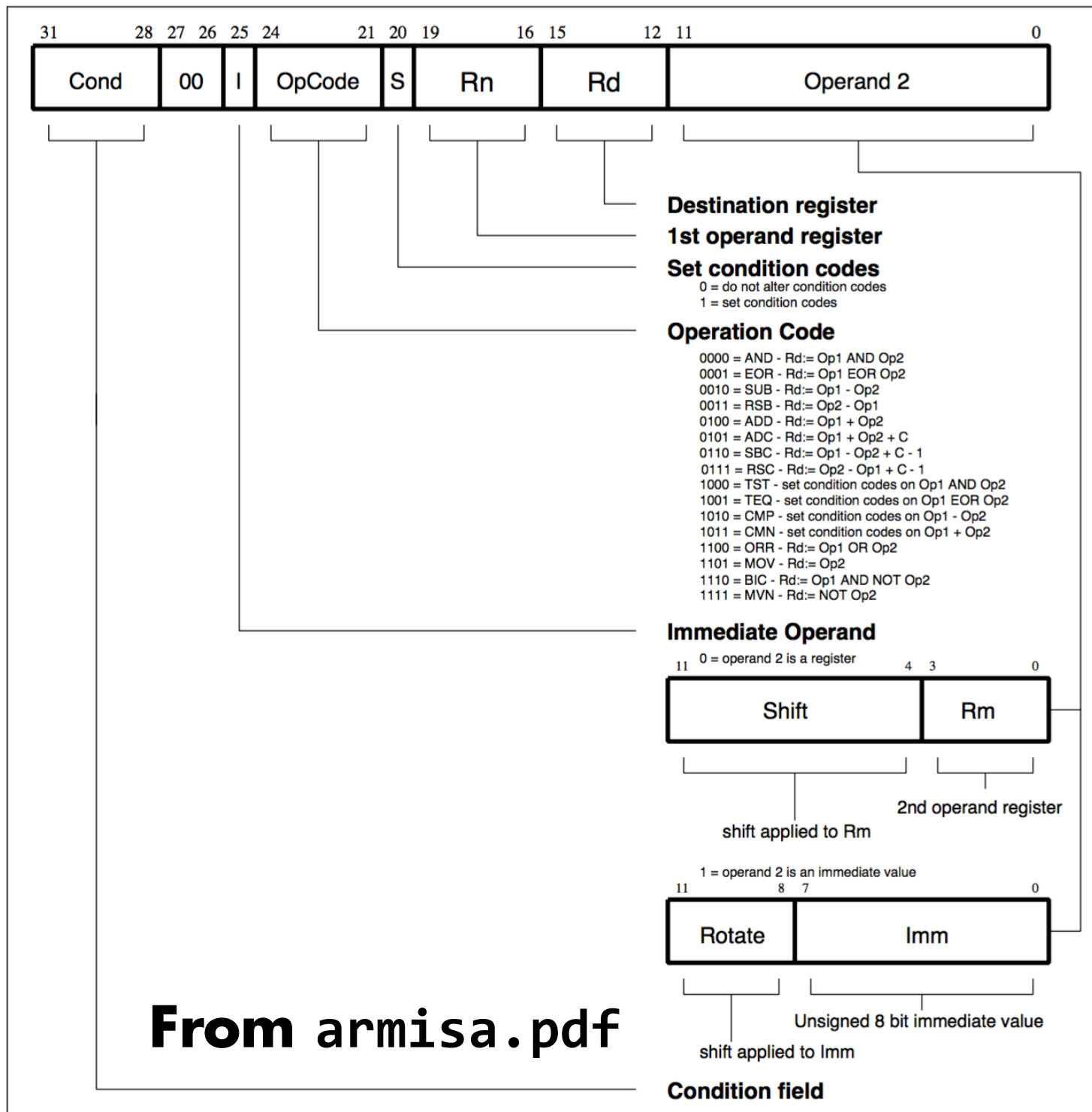
**str r1, [r0]**

**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**



**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

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 #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

little-endian

(lowest byte first)

01

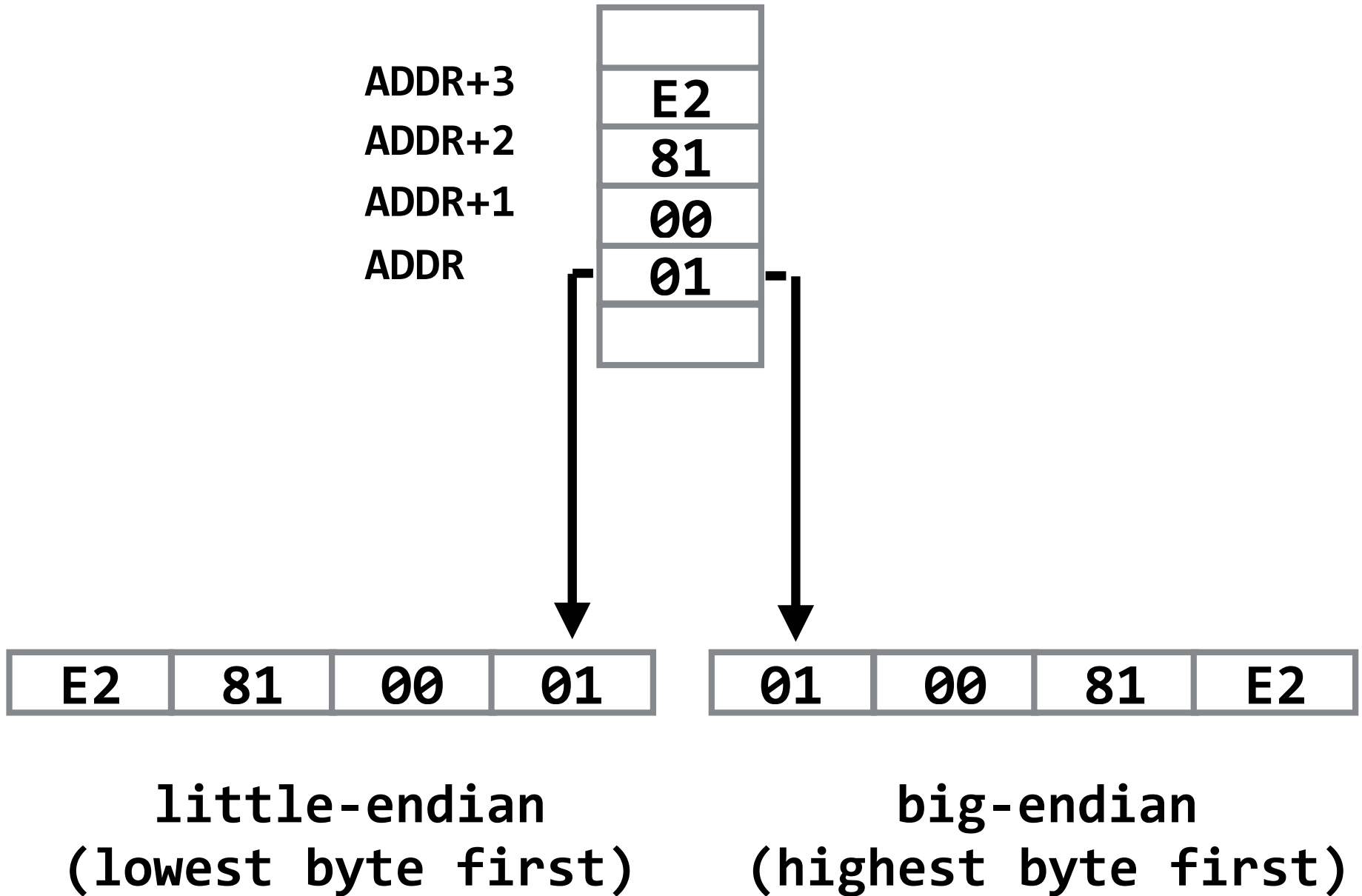
00

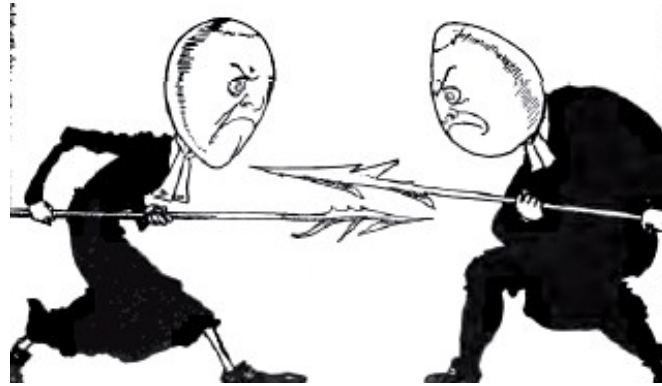
81

E2

big-endian

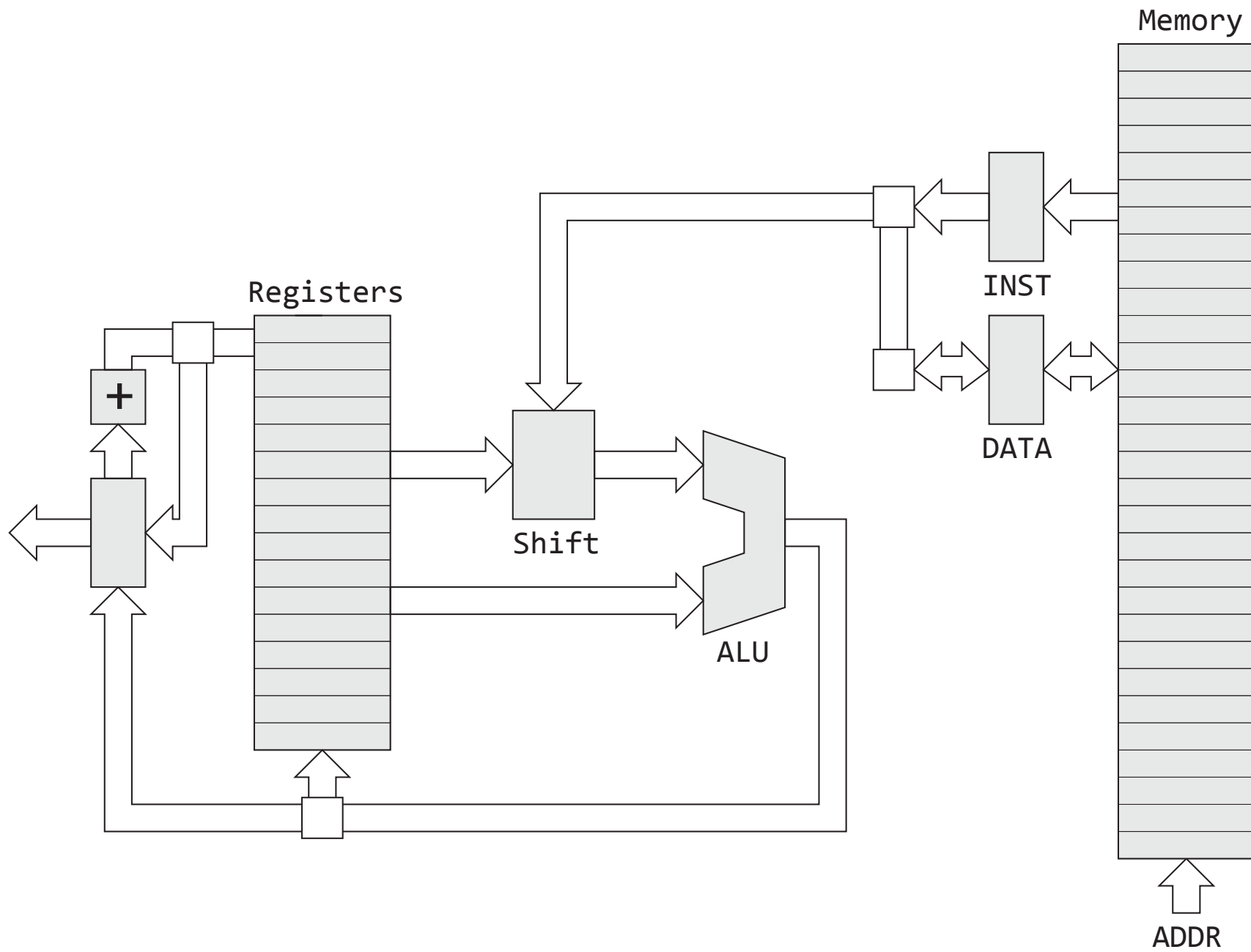
(highest byte first)





**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**



## Rotate Right (ROR)

[illegible]

# 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

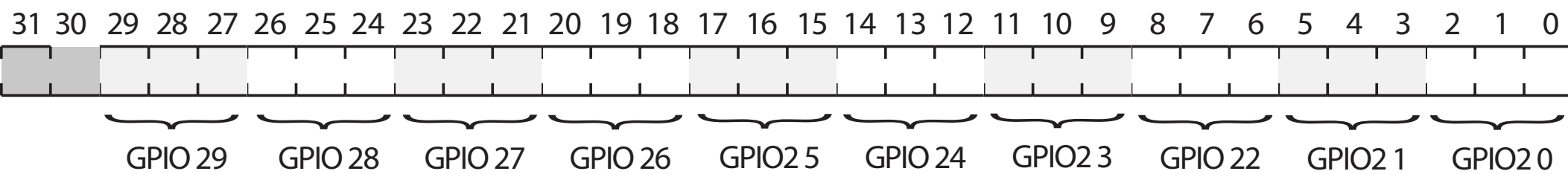
1110	0010	0100	0101	0111	1100	0000	0011
E	2	4	5	7	C	0	3

# **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
```

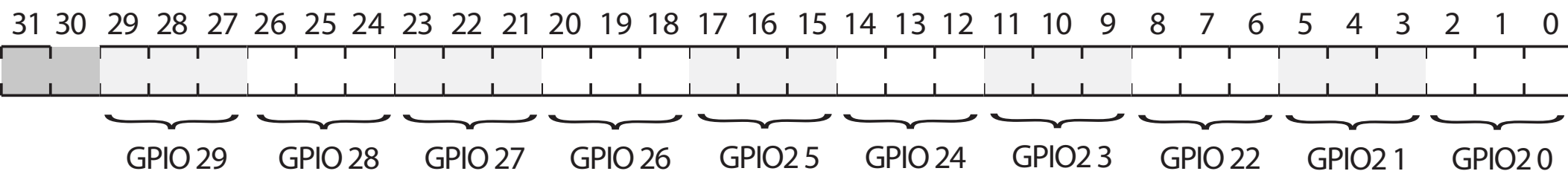
```
// Set GPIO 20 to OUTPUT
```

```
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)
```

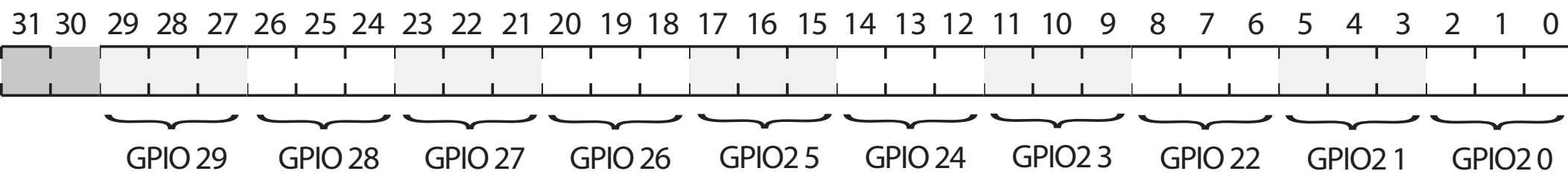
```
orr r0, #0x00000008
```

```
// 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*iii)
#
# 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 0000 0000 0000 0000 0000 0000

b = bal = branch always

cond            addr

**1110** 101L 0000 0000 0000 0000 0000 0000

bne

cond            addr

**0001** 101L 0000 0000 0000 0000 0000 0000

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**
- **Lab assignments sent out before class**
- **Read lab1 guide (now online)**
- **Install tool chain**
- **Bring \$50 lab fee**

## **Assignment 1**

- **Larson scanner**

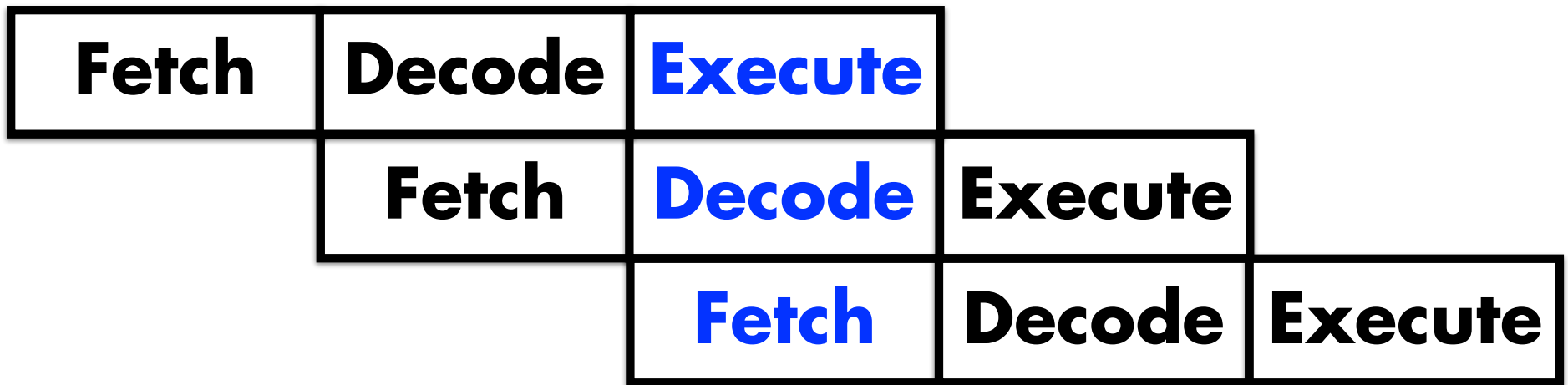
**Extra Slides**

# **Branch Instructions and Pipelining**

# **Processors execute instructions in phases**



## Phases are pipelined



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

**// disassemble on.s**

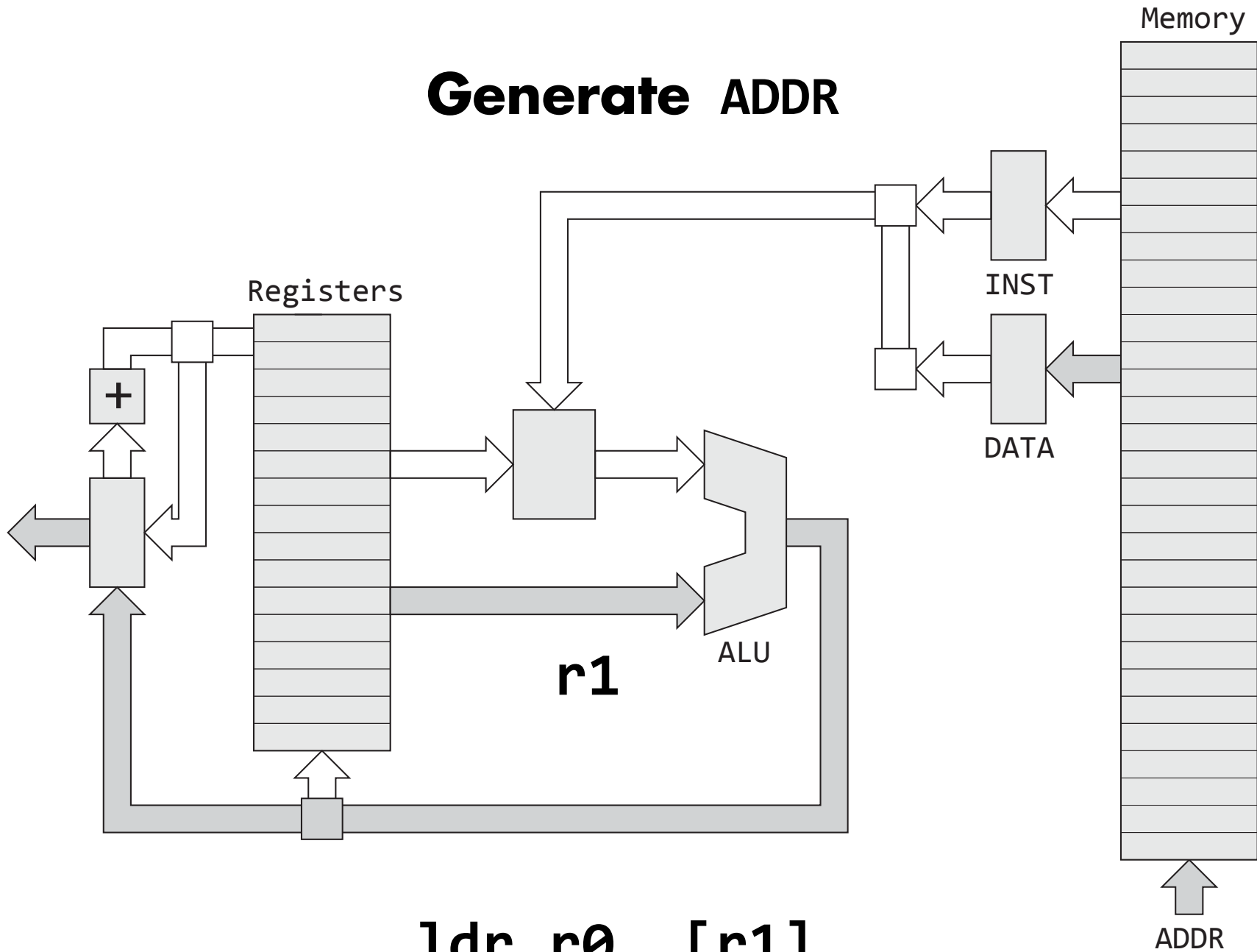
<b>0:</b>	<b>e59f0014</b>	<b>ldr r0, [pc, #0x14]</b>
<b>4:</b>	<b>e3a01001</b>	<b>mov r1, #1</b>
<b>8:</b>	<b>e5801000</b>	<b>str r1, [r0]</b>
<b>c:</b>	<b>e59f000c</b>	<b>ldr r0, [pc, #0x0c]</b>
<b>10:</b>	<b>e3a01601</b>	<b>mov r1, #0x100000</b>
<b>14:</b>	<b>e5801000</b>	<b>str r1, [r0]</b>
<b>18: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

**Indexed Loads**

**PC Relative Addressing**

**$r0 = \text{mem}[r1]$**

**Generate ADDR**

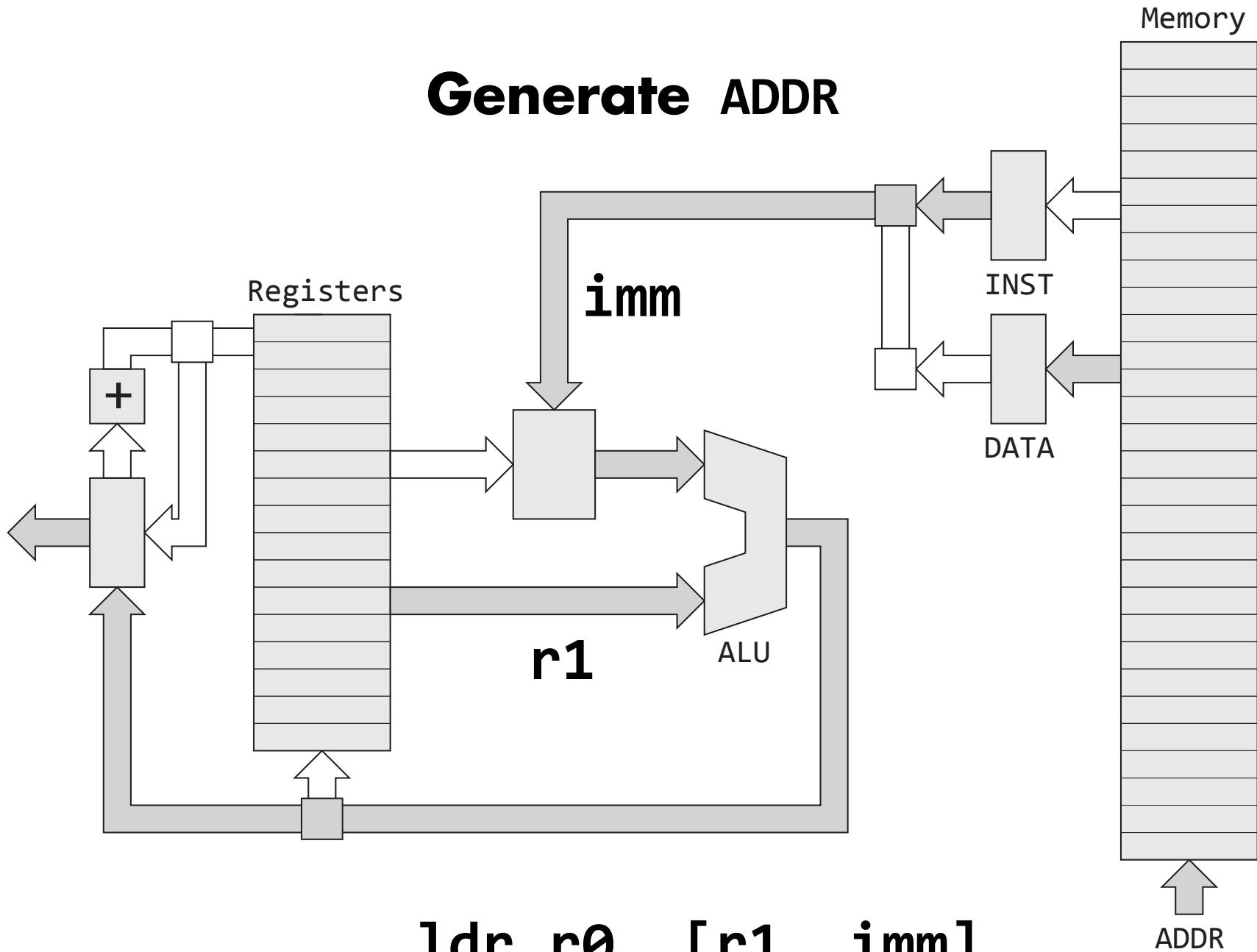


**ldr r0, [r1]**



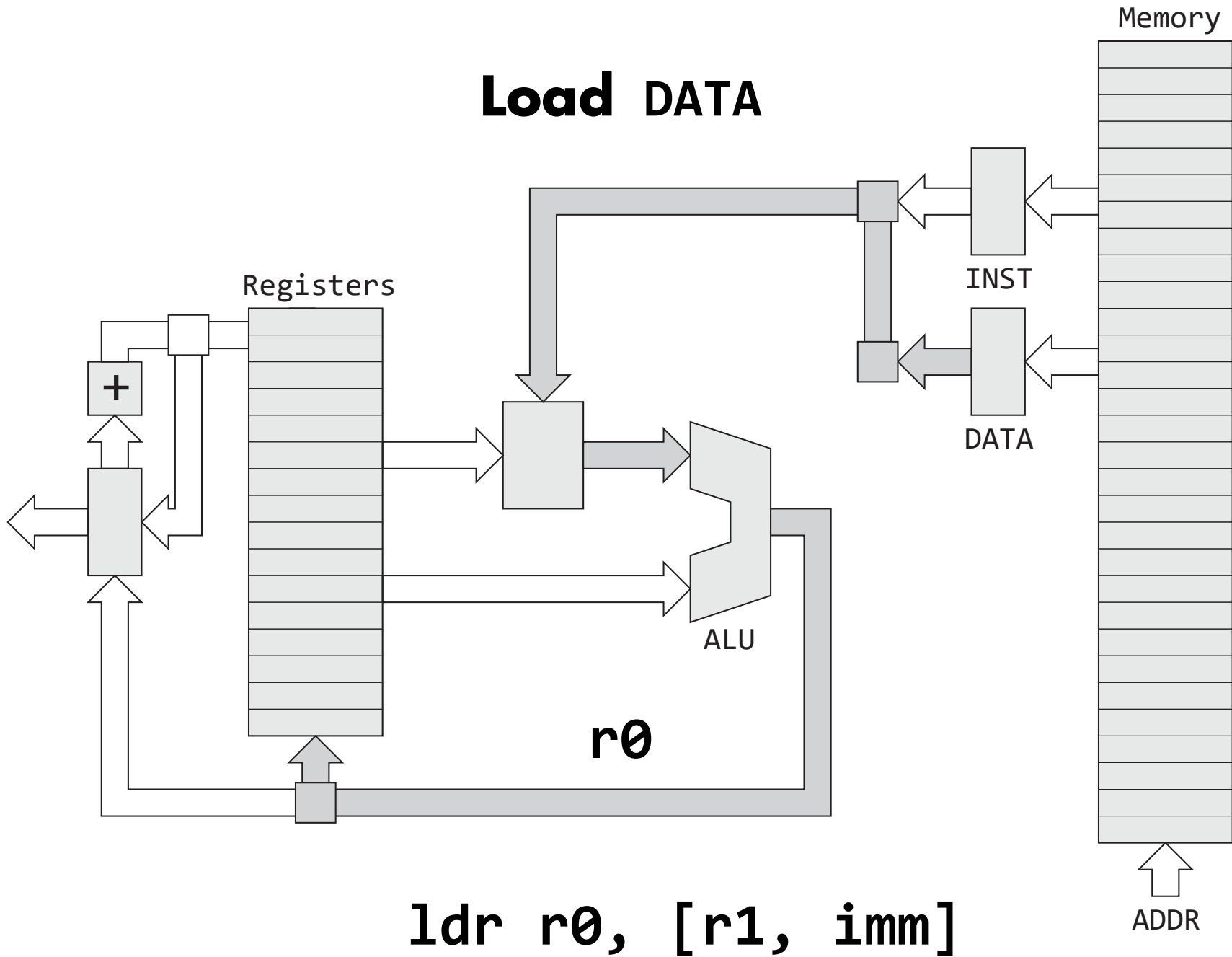
**$r0 = \text{mem}[r1 + \text{imm}]$**

**Generate ADDR**



**$r0 = \text{mem}[r1 + \text{imm}]$**

**Load DATA**



**// disassemble on.s**

**// PC relative addressing**

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

**// disassemble on.s**

**// PC relative addressing**

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