

ARM Instructions

// Program to turn on an LED on GPIO 20

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

**ldr r0, =0x2020001C ; SET0 register
mov r1, #(1<<20) ; Bit 20 / GPIO20
str r1, [r0]**

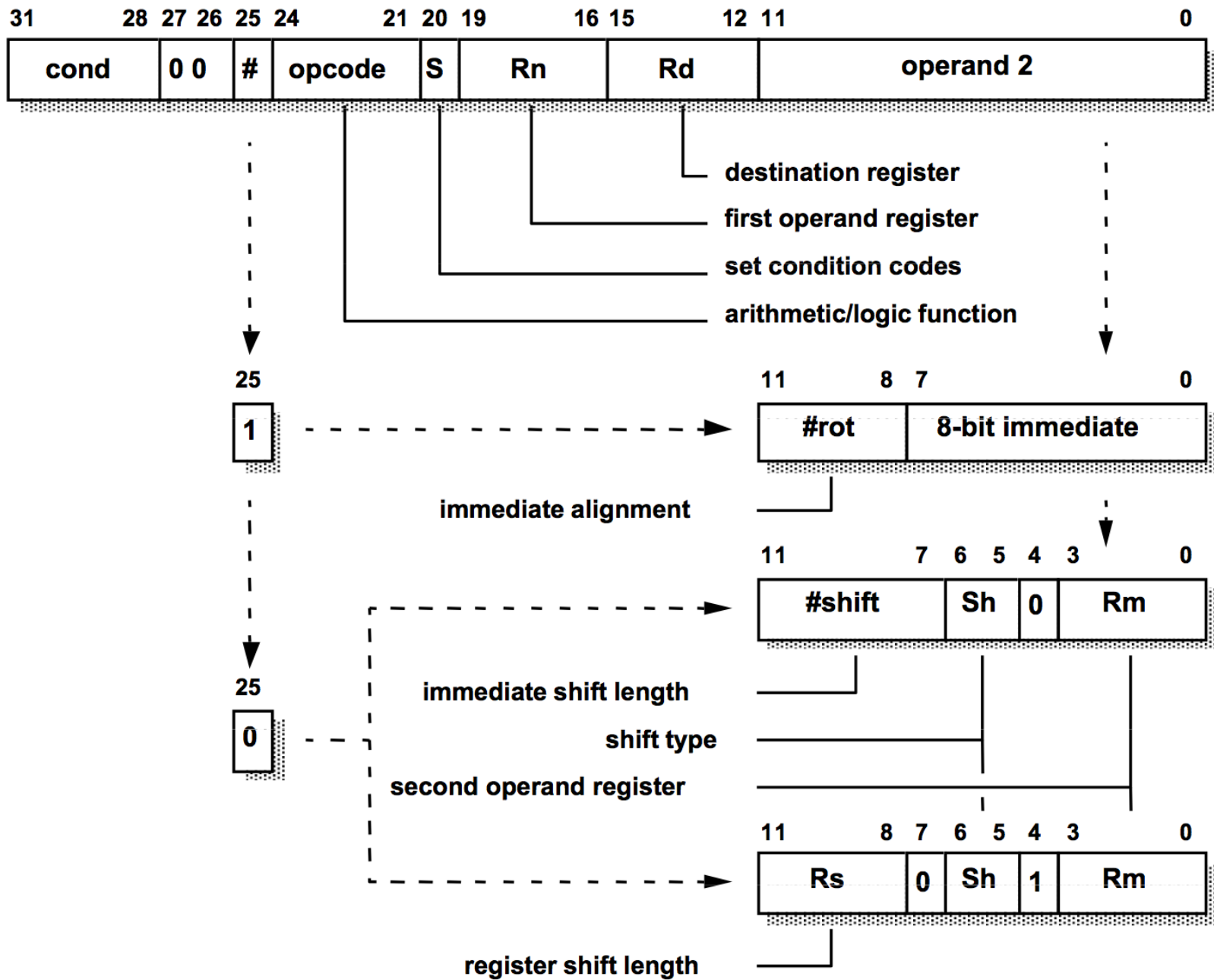
b .

3 Types of Instructions

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

Architecture is quite simple

Data Processing Instructions



From **ARM architecture manual ...**

data processing instruction

ra = rb op #imm

#imm = uuuu uuuu

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

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2(operand1 is ignored)
BIC	1110	operand1 AND NOT operand2(Bit clear)
MVN	1111	NOT operand2(operand1 is ignored)

data processing instruction

ra = rb op #imm

#imm = uuuu uuuu

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

add r1, r0, #1

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

data processing instruction

ra = rb op #imm

#imm = uuuu uuuu

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

add r1, r0, #1

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

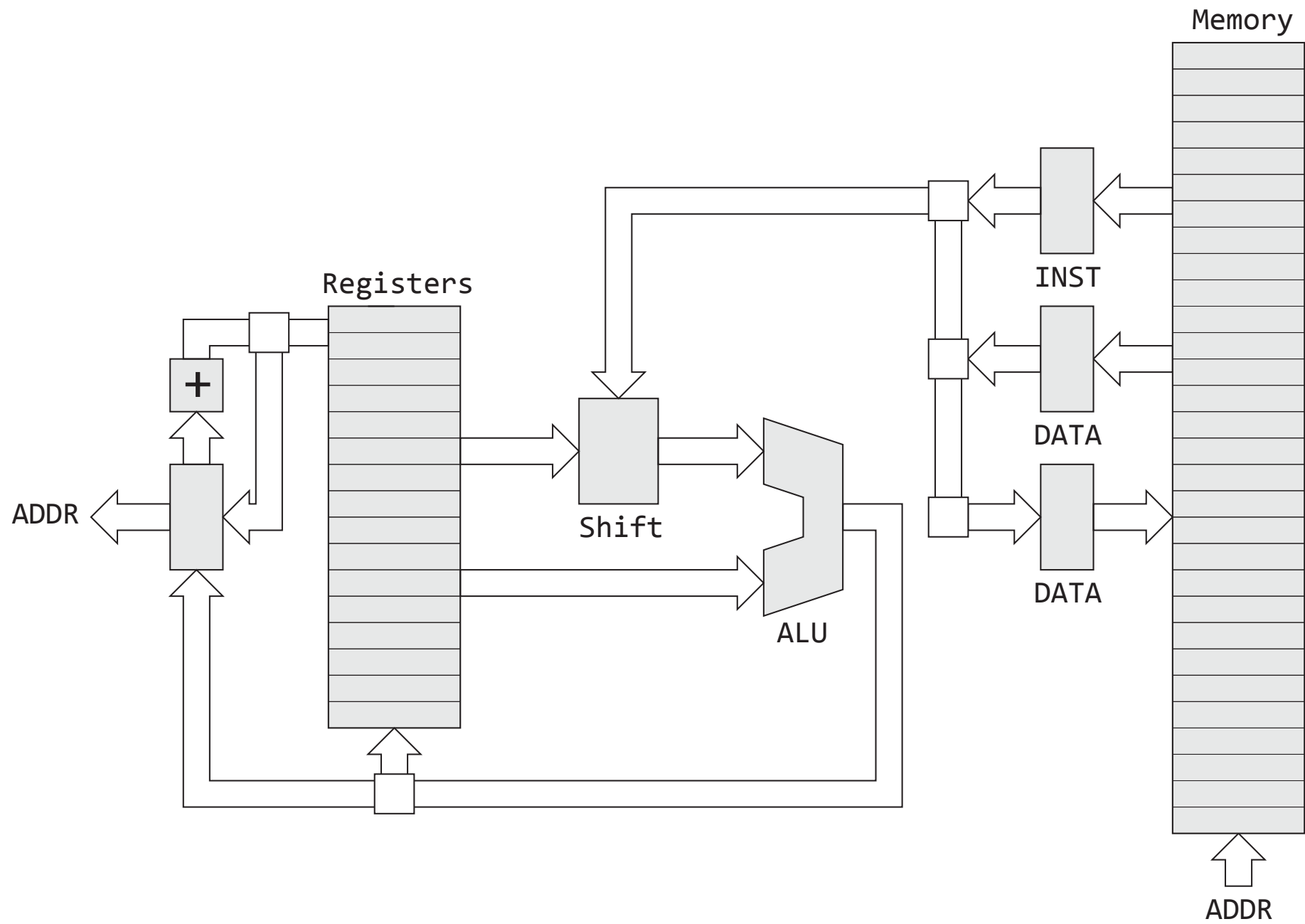
1110	0010	1000	0000	0001	0000	0000	0001			
E	2	8	0	1	0	0	1			

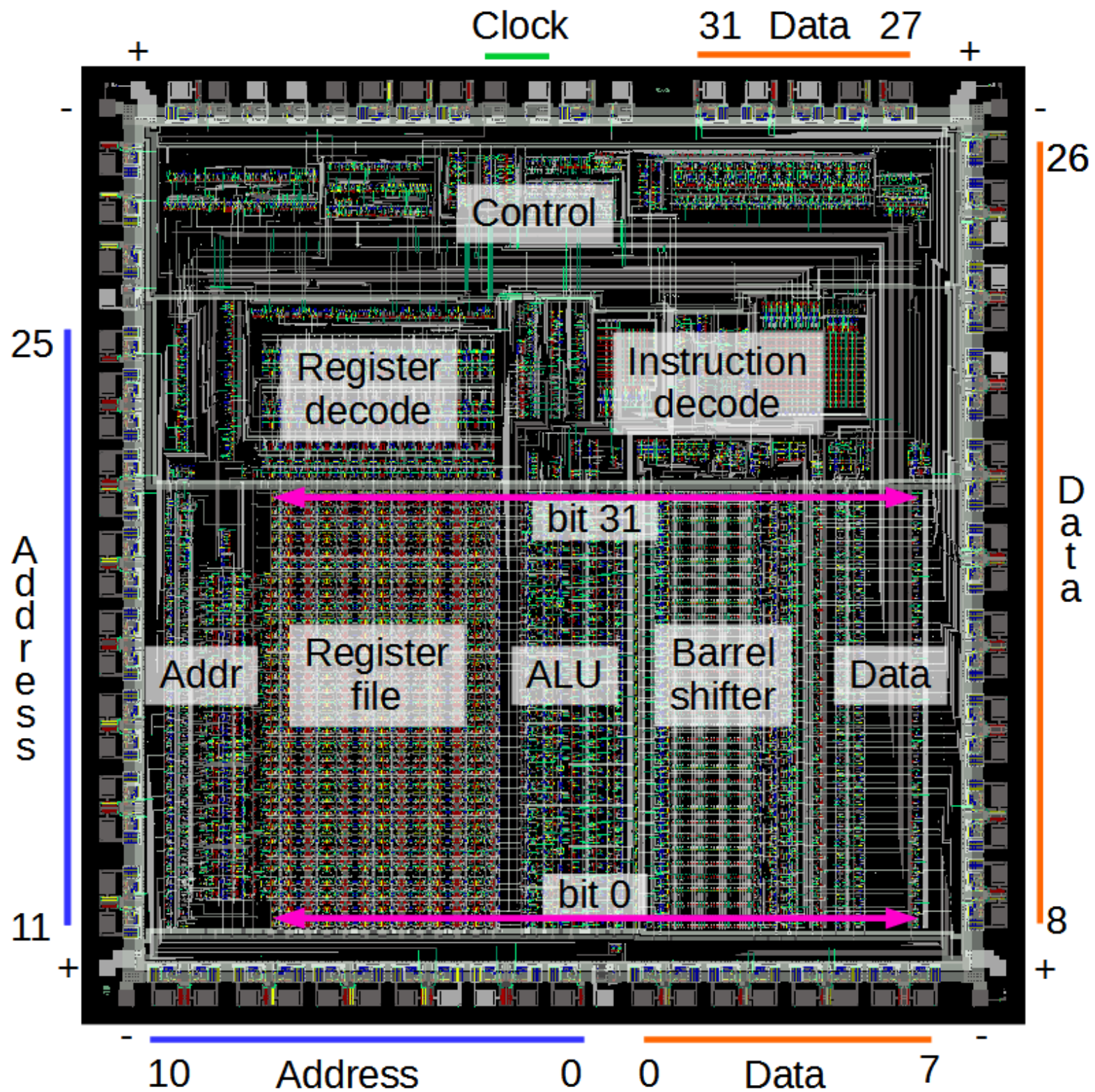
```
const int ADD = 0x04;

// compile data processing instruction
// with immediate

int dataprocinst(
    int op, int ra, int rb, int imm )
{
    int inst = 0xE2000000;
    inst |= (op<<20);
    inst |= (rb<<16)|(ra<<12)|imm;
    return inst;
}

inst = dataprocinst(ADD, 1, 0, 1);
```





Rotate Right

[illegible]

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

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

```
add r1, r0, #0x10000
```

			add		r0		r1	rot		0x10000
1110	00	1	0100	0	0000		0001	1000	0000	0001

```

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

```

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

```
add r1, r0, #0x10000
```

			add		r0		r1		rot		#0x10000
1110	00	1	0100	0	0000	0001	1000	0000	0001		

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

Determine the machine code for

sub r7, r5, #0x300

imm = (uuuu uuuu) ROR (2*iii)

Remember that ra is the result

				op		rb		ra				
1110	00	1	oooo	0	bbbb	aaaa	iii	uuuu	uuuu			


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

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

```
sub r7, r5, #0x300
```

			sub		r5		r7		rot		#0x300
1110	00	1	0010	0	0101		0111		0100		0000 0011

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

// Example: Replace

ldr r0, =0x20200008

// with

mov r0, #0x20000000

orr r0, r0, #0x00200000

orr r0, r0, #0x00000008

Condition Codes

```
// loop
```

```
mov r2, #0x3F0000
```

```
loop: // colon indicates a label
```

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

```
    bne loop
```

```
// A label is just a constant address
```

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

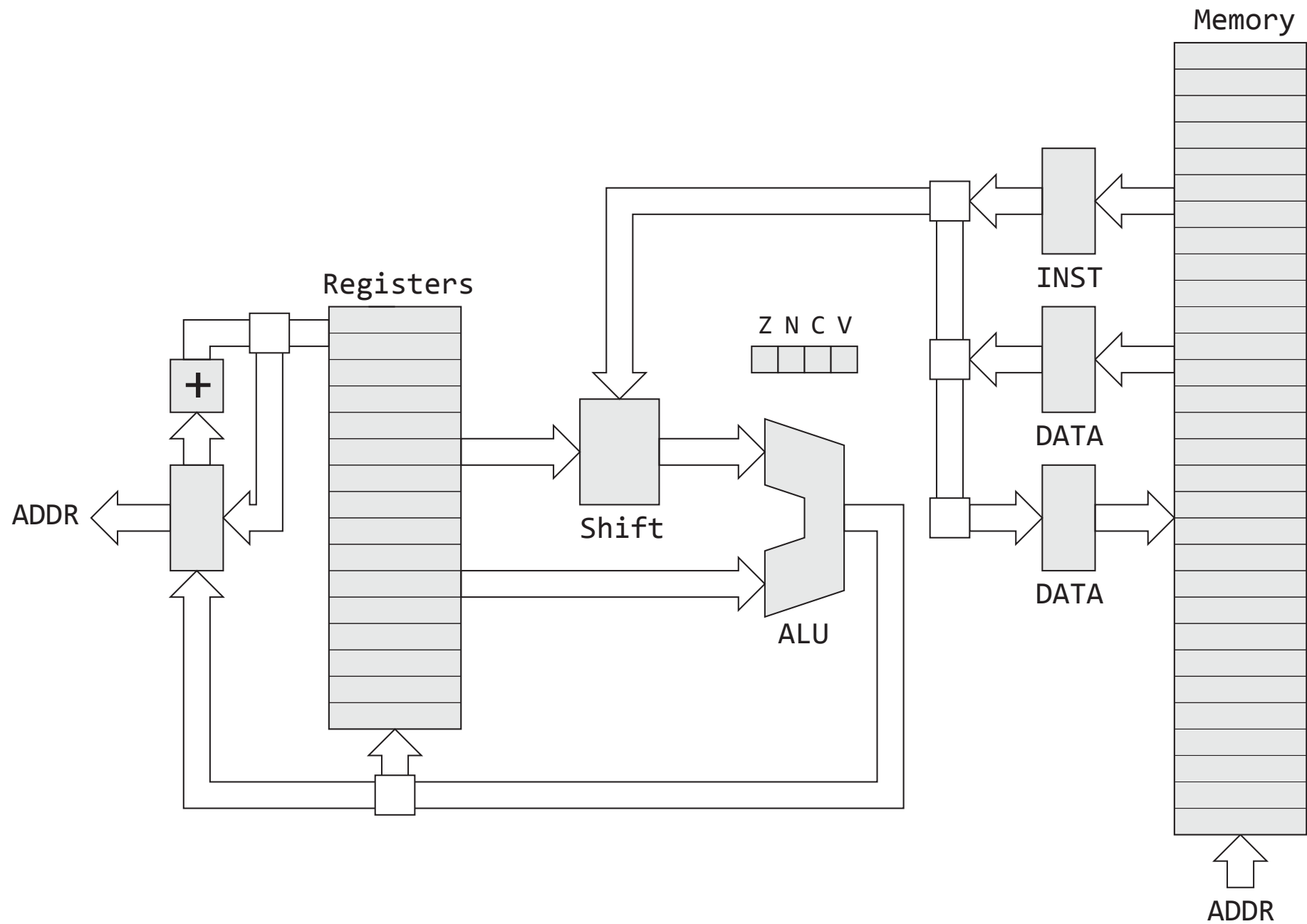
```
# s=1 means set condition code
```

	op	s	rb	ra					
1110 00 1 0000	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	0000	0001		

```
E2 52 20 01
```



Condition Codes

Z - Result is 0

N - Result is <0

C - Carry generated

V - Arithmetic overflow

branch

cccc addr

cccc 101L 0000 0000 0000 0000 0000 0000

b = bal = branch always

cccc addr

1110 101L 0000 0000 0000 0000 0000 0000

bne

cccc addr

0001 101L 0000 0000 0000 0000 0000 0000


```
# data processing instruction
#  ra = rb op #imm
#  #imm = uuuu uuuu ROR (2*iii)
#
# Predicated execution:
#   if condition is true
#
```

```
red      op      rb   ra
cccc 00 1 0000 s bbbb aaaa iiiii uuuu uuuu
```

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

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

ldr r0, =0x20200008

mov r1, #1

str r1, [r0]

// Bit 20 for GPIO 20

mov r1, #(1<<20)

...

// r0 points to GPIO SET2 register

ldr r0, =0x2020001C

str r1, [r0]

// delay

mov r2, #0x3F0000

wait1:

subs r2, #1

bne wait1

...

// r0 points to GPIO CLR2 register

ldr r0, =0x20200028

str r1, [r0]

// delay

mov r2, #0x3F0000

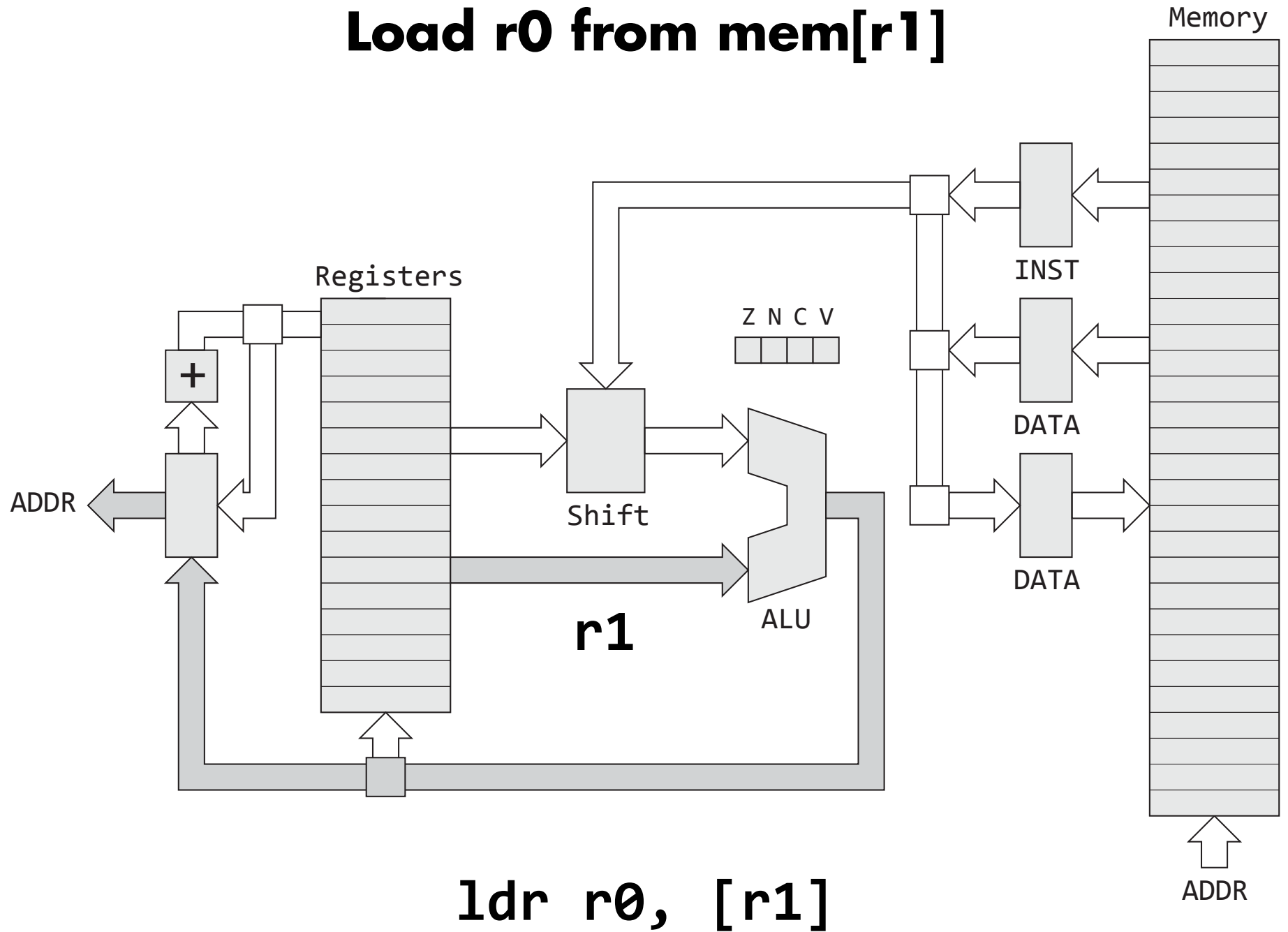
wait2:

sub r2, #1

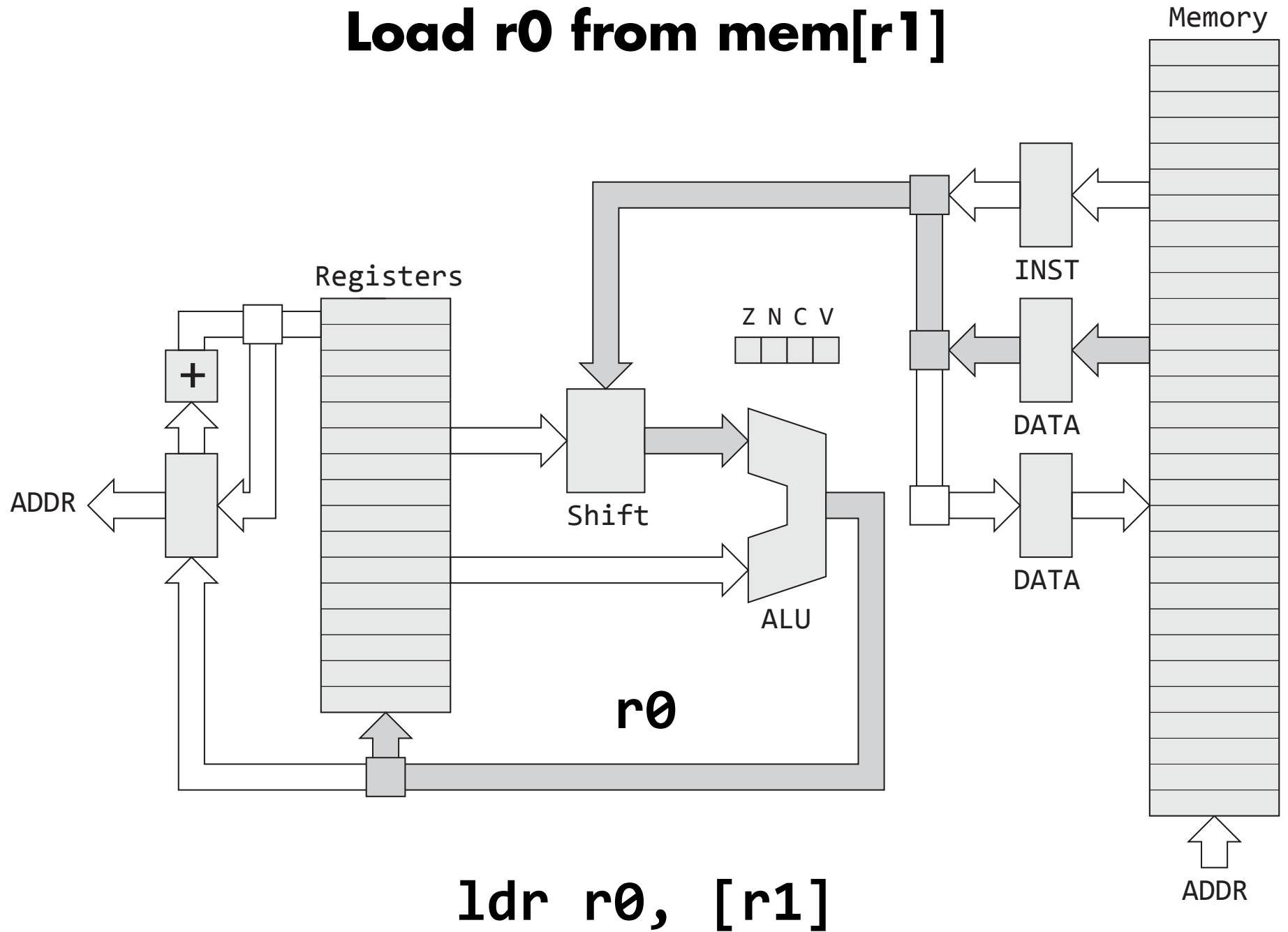
bne wait2

Load/Store Instructions

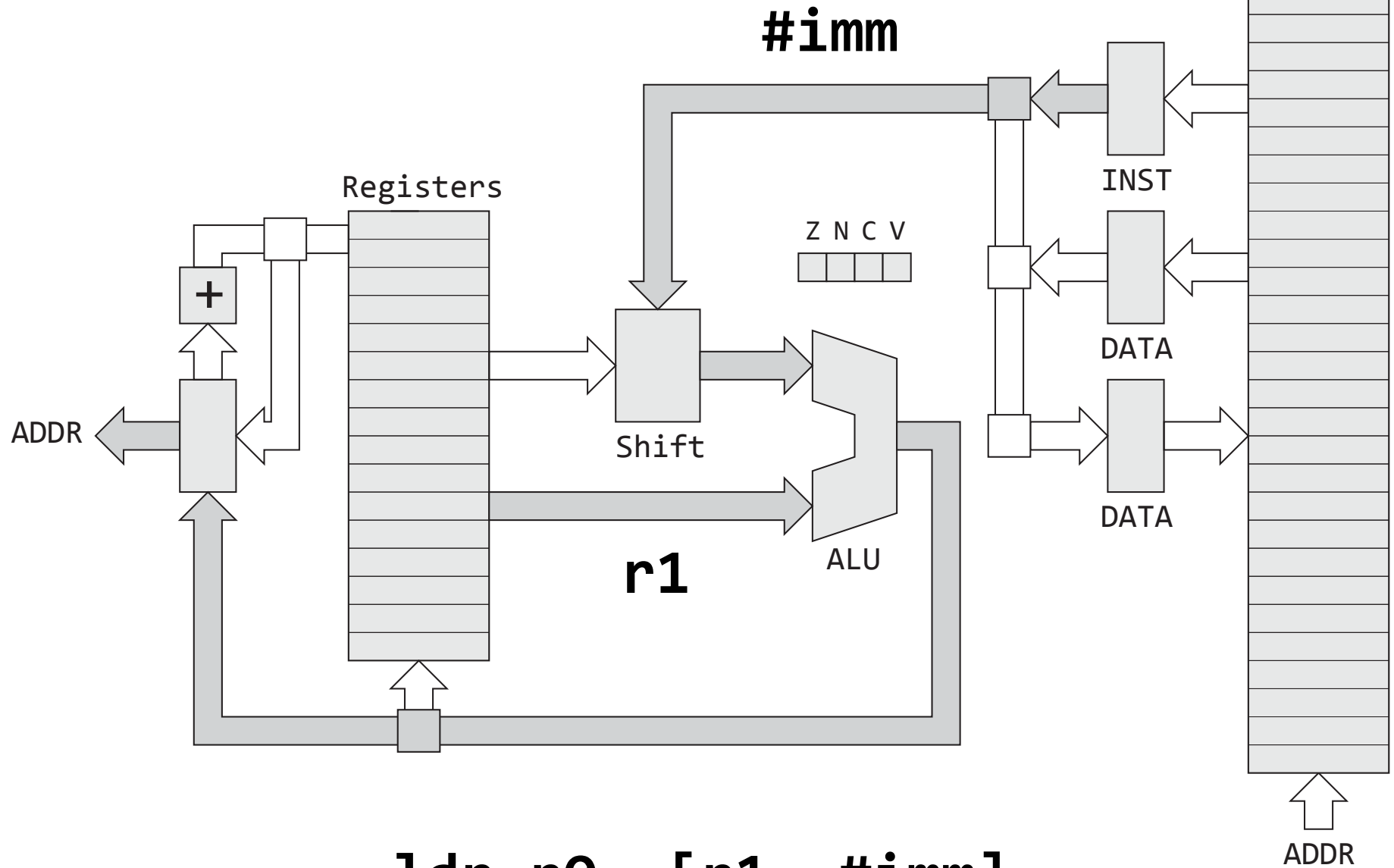
Load r0 from mem[r1]



Load r0 from mem[r1]



Load r0 from mem[r1+imm]



// Program to turn on an LED on GPIO 20

ldr r0, =0x20200008

mov r1, #1

str r1, [r0]

ldr r0, =0x2020001C

mov r1, #(1<<20)

str r1, [r0]

// Program to turn on an LED on GPIO 20

ldr r0, =0x20200000

mov r1, #1

str r1, [r0, #0x08]

mov r1, #(1<<20)

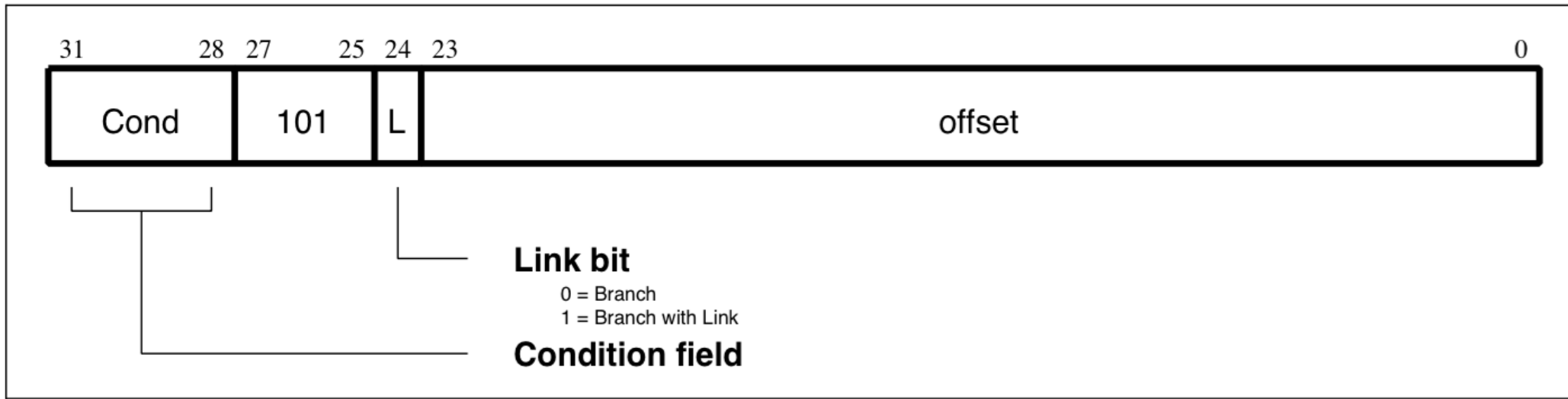
str r1, [r0, #0x1C]

// 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
1c:	20200008	
20:	2020001c	

// 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
1c:	20200008	
20:	2020001c	



b .
e3 ff ff fe

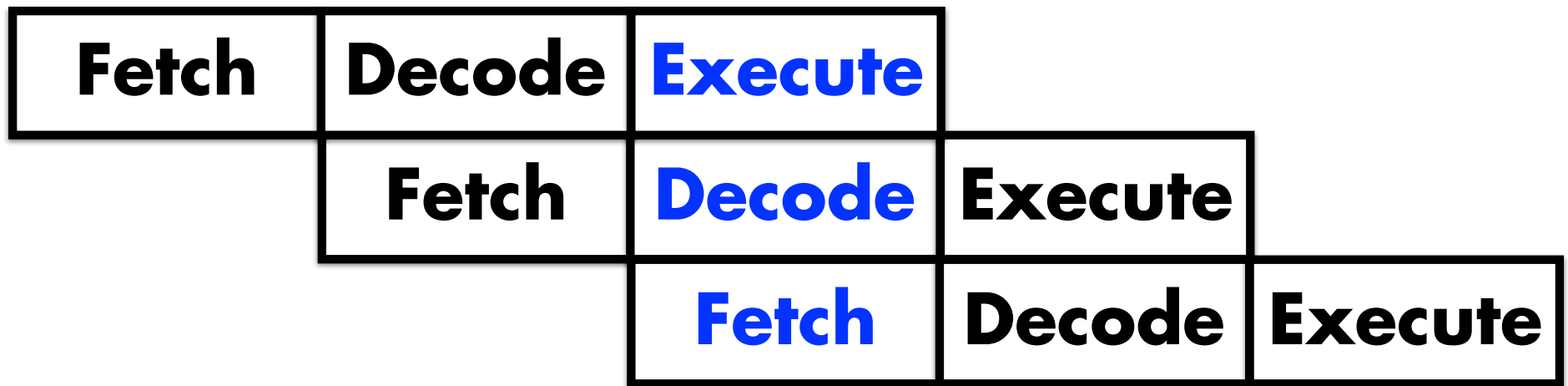
bne .
13 ff ff fe

NB. We will explain the link bit next lecture

Processors execute instructions in phases



Phases are pipelined



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

Assembly Language

Most importantly, you need to understand how processors represent information and execute instructions

Normally write code in C, but sometimes will need to read assembly to figure out what is going on

Instruction set architecture often easier to understand by looking at the bits

Concepts

Bits and bit operations

Types of ALU instructions

Condition codes: setting and branching

Addressing modes in loads & stores

Debugging Hints

Start with the simplest program.

Take baby steps, check that things work, then take another small step ...

If something doesn't work, backup to a known working state. Identify state 0.

Start by typing it in by hand; do not learn by cutting and pasting