

**ARM**

**Assembly Language  
and  
Machine Code**

**Goal: Blink an LED**

# 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 (Starting next lecture)**

# 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 (Starting next lecture)**

# **Review:**

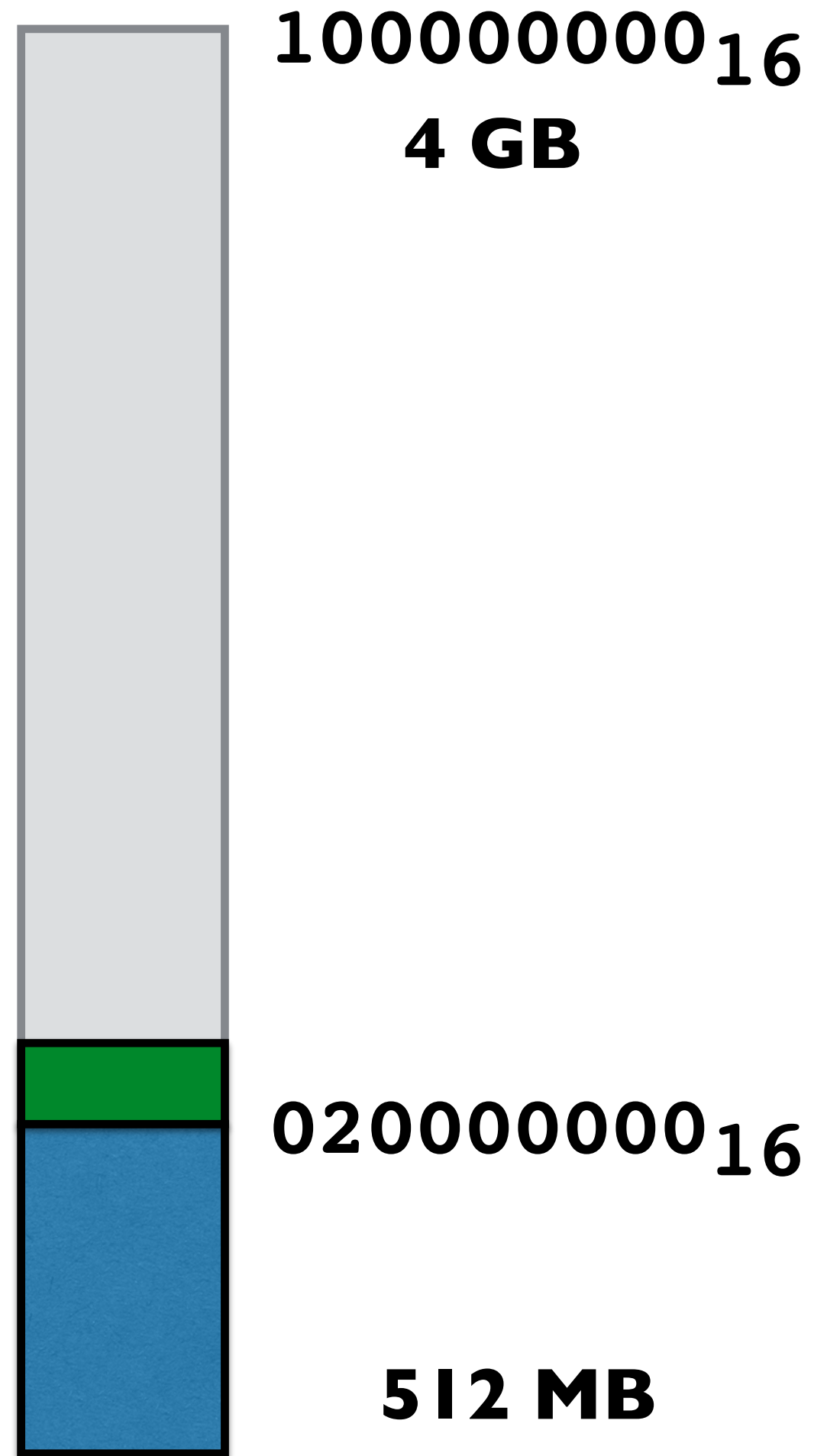
## **Turning on an LED**

# Memory Map

**Peripheral registers  
are mapped  
into address space**

**Memory-Mapped IO  
(MMIO)**

**MMIO space is above  
physical memory**



# GPIO Function Select Registers Addresses

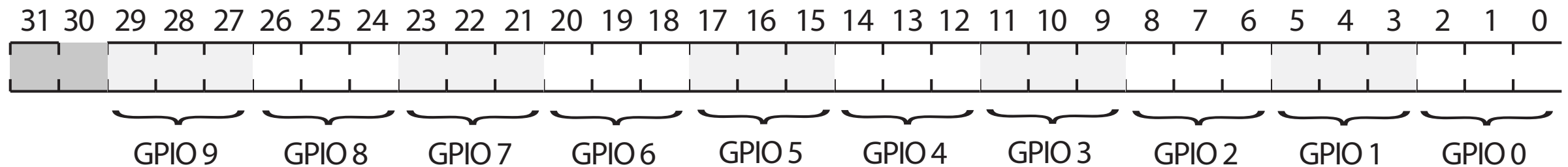
Address	Field Name	Description	Size	Read/Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
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**

# GPIO Function Select Register



**Function is INPUT, OUTPUT, or ALT0-ALT5**

**3 bits per GPIO pin**

**10 pins per 32-bit register (2 wasted bits)**

**54 GPIOs pins requires 6 registers**

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

<b>Bit pattern</b>	<b>Pin Function</b>
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

**Specifying 1 of 8 functions requires 3 bits**



```

// configure GPIO 20 for output
//
// FSEL0 = 0x20200000 (GPIO0-GPIO9)
// FSEL1 = 0x20200004 (GPIO10-GPIO19)
// FSEL2 = 0x20200008 (GPIO20-GPIO29)
// ...
mov r0, #0x20          // r0 = 0x00000020
lsl r1, r0, #24        // r1 = 0x20000000
lsl r2, r0, #16        // r2 = 0x00200000
orr r0, r1, r2         // r0 = 0x20200000
orr r0, r0, #0x08      // r0 = 0x20200008
mov r1, #1             // r1 = 1 is OUTPUT
str r1, [r0]           // store 1 to 0x20200008
                        // - GPIO20 now output
                        // - GPIO21-29 now input

```

## 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 an 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**

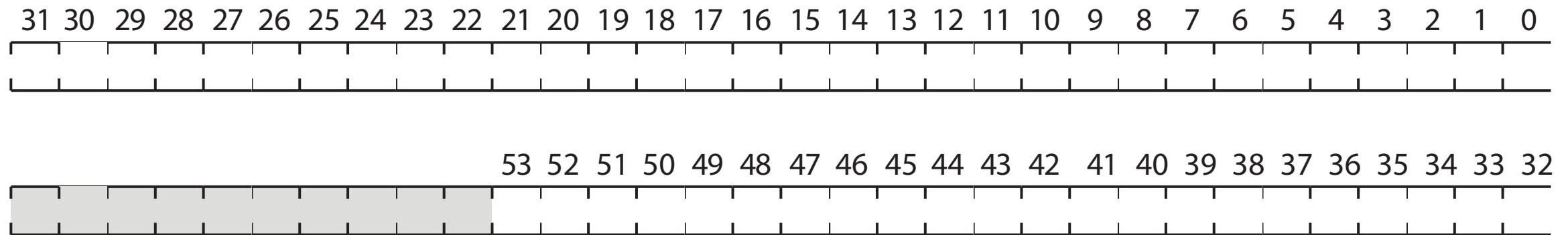
Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin <i>n</i> .	R/W	0

**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**

```

/// SET0 = 0x2020001c
mov r0, #0x20          // r0 = 0x00000020
lsl r1, r0, #24         // r1 = 0x20000000
lsl r2, r0, #16         // r2 = 0x00200000
orr r0, r1, r2          // r0 = 0x20200000
orr r0, r0, #0x1c       // r0 = 0x2020001c
mov r1, #1              // r1 = 0x00000001
lsl r1, #20              // r1 = 0x00010000
str r1, [r0]            // store 1<<20 to
0x2020001c

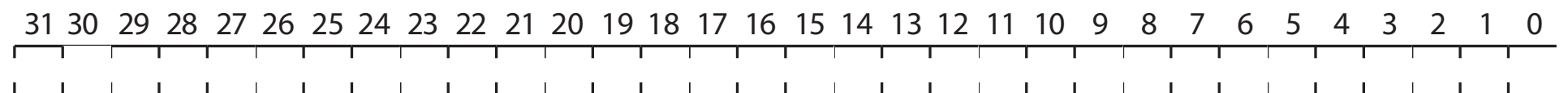
```

```

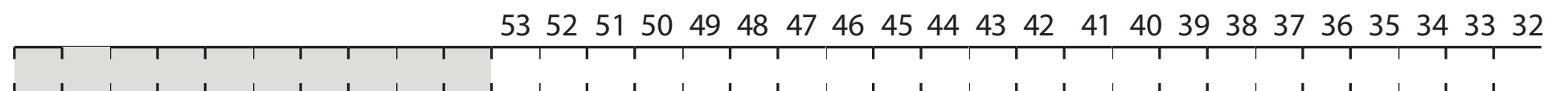
// loop forever
anna:
b anna

```

**SET0**



**SETI**

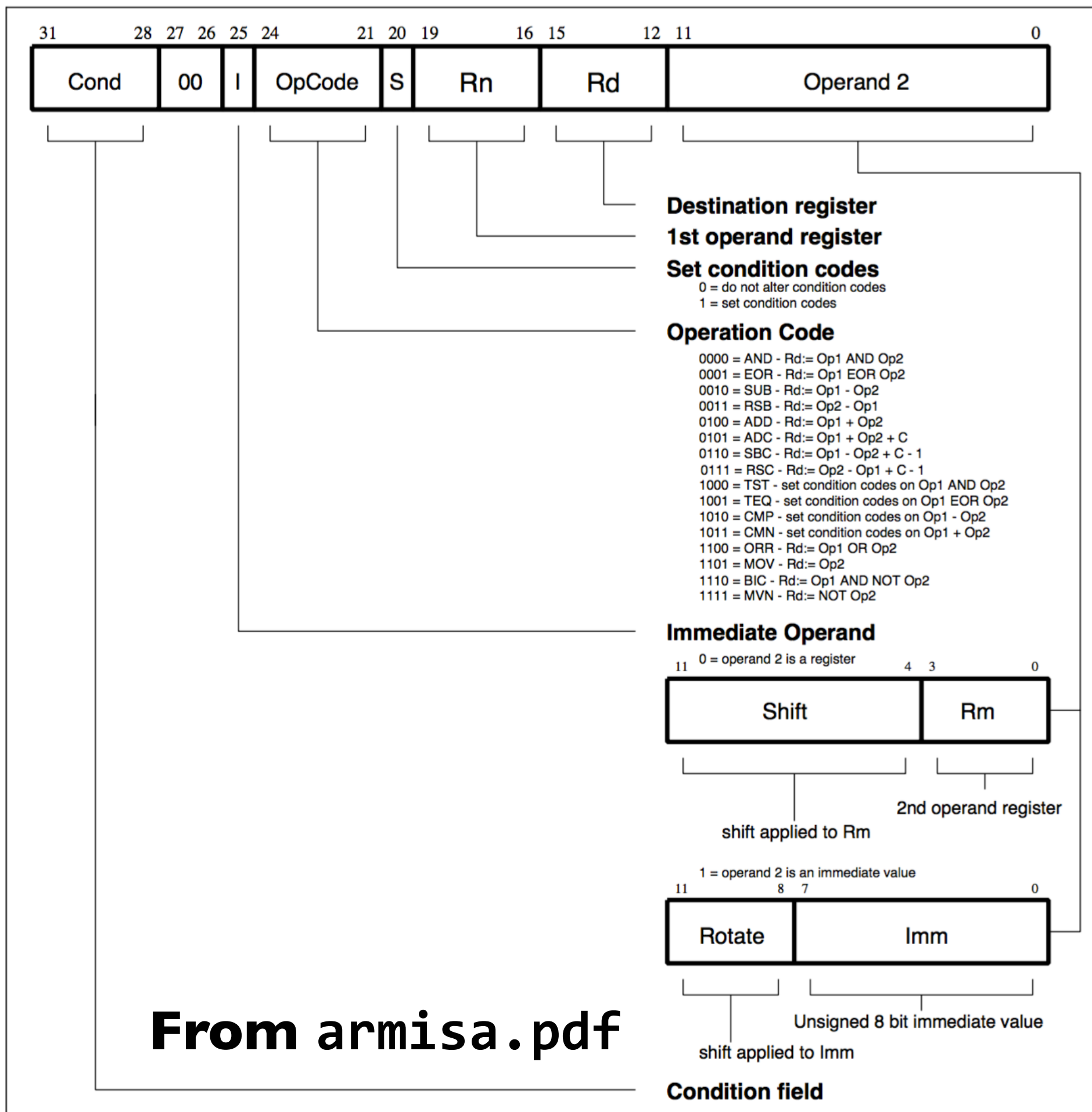


**How are instructions  
formed in a 32-bit value?**

# **There are three Types of Instructions**

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

# **Data Processing Instructions and Machine Code**



From armisa.pdf

Figure 4-4: Data processing instructions



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

**Immediate mode instruction**

**Set condition codes**

1110 00 i op s rb ra rc  
oooo 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	$ra = rb + rc + CARRY$
SBC	0110	$ra = rb - rc + (1 - CARRY)$
RSC	0111	$ra = rc - rb + (1 - CARRY)$
TST	1000	$rb \& rc$ (ra not set)
TEQ	1001	$rb \wedge rc$ (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

# add r0, r1, r2

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

# i=0, s=0

			add		r1	r0	r2		
1110	00	0	0100	0	0001	0000	0000	0000	0010

# data processing instruction

# ra = rb op rc

# add r0, r1, r2

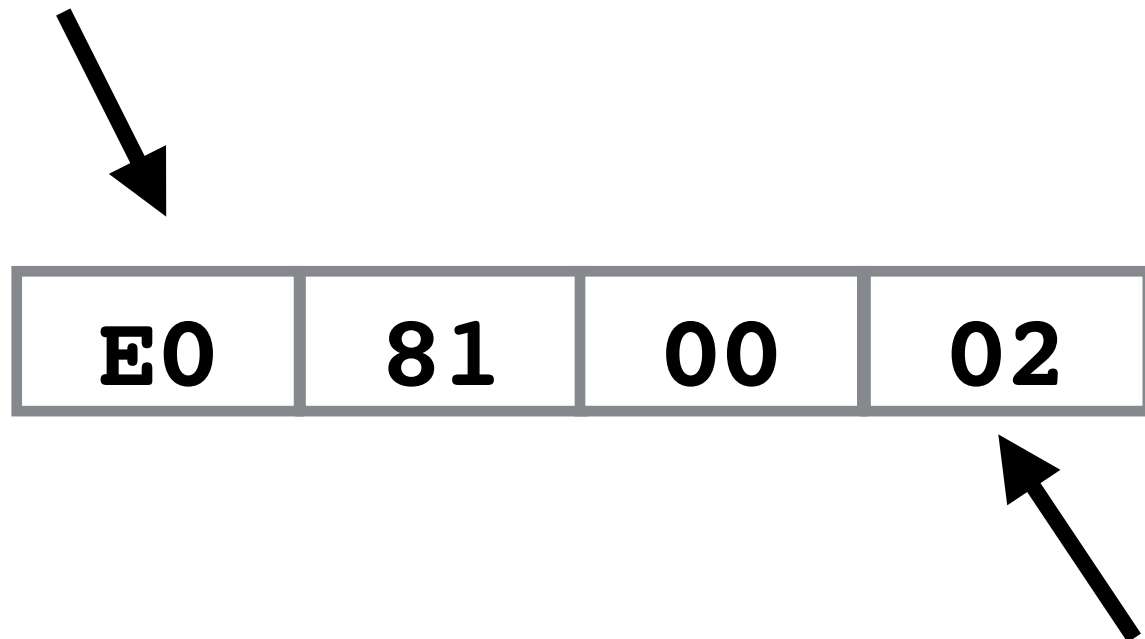
			op		rb	ra	rc		
1110	00	i	oooo	s	bbbb	aaaa	cccc	cccc	cccc

# i=0, s=0

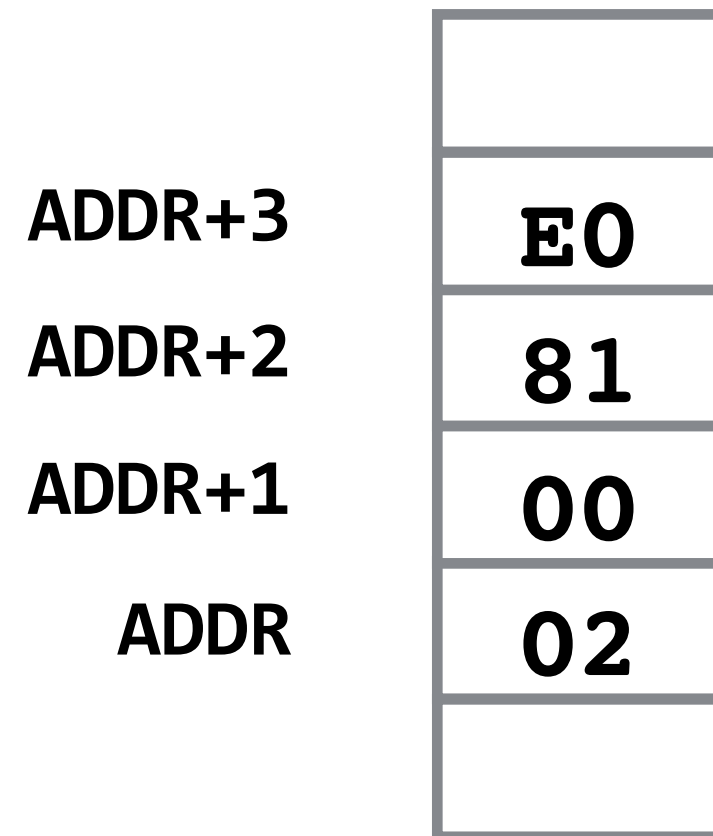
			add		r1	r0	r2		
1110	00	0	0100	0	0001	0000	0000	0000	0010

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

**most-significant-byte (MSB)**



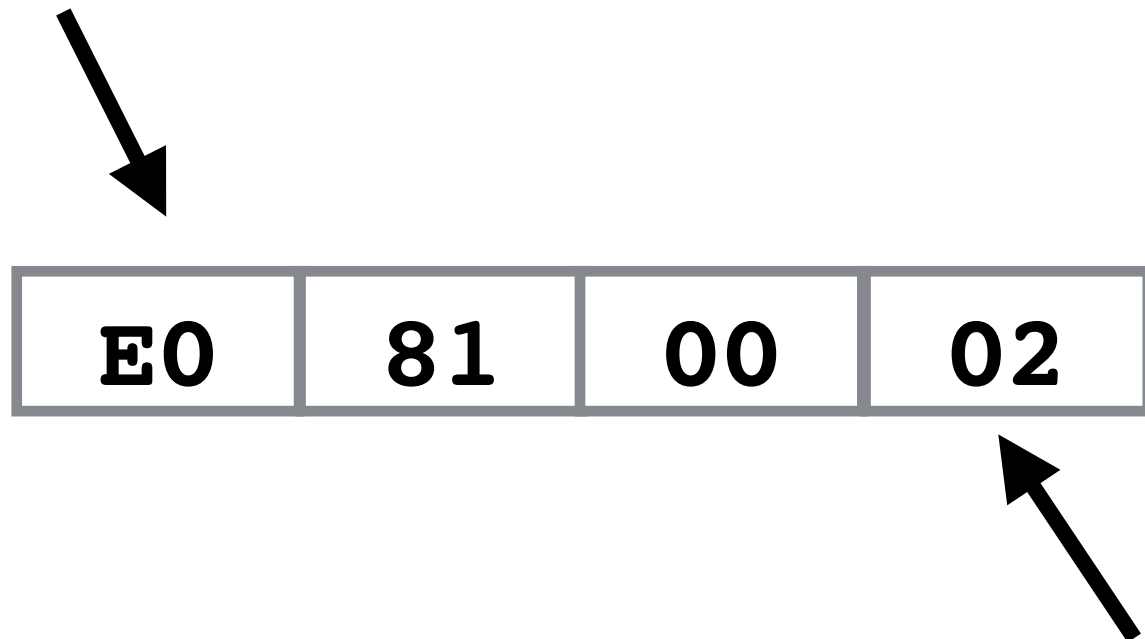
**least-significant-byte (LSB)**



**little-endian  
(LSB first)**

**ARM uses little-endian**

**most-significant-byte (MSB)**



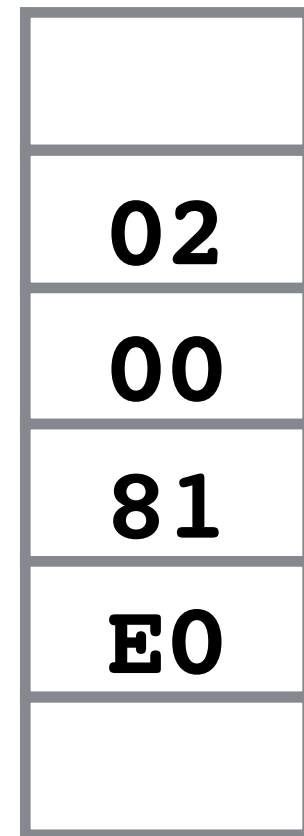
**least-significant-byte (LSB)**

ADDR+3

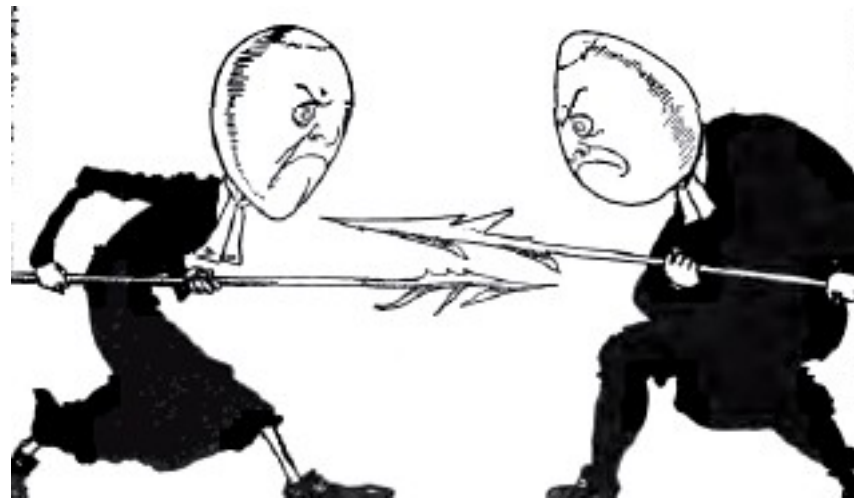
ADDR+2

ADDR+1

ADDR



**big-endian**  
(MSB first)

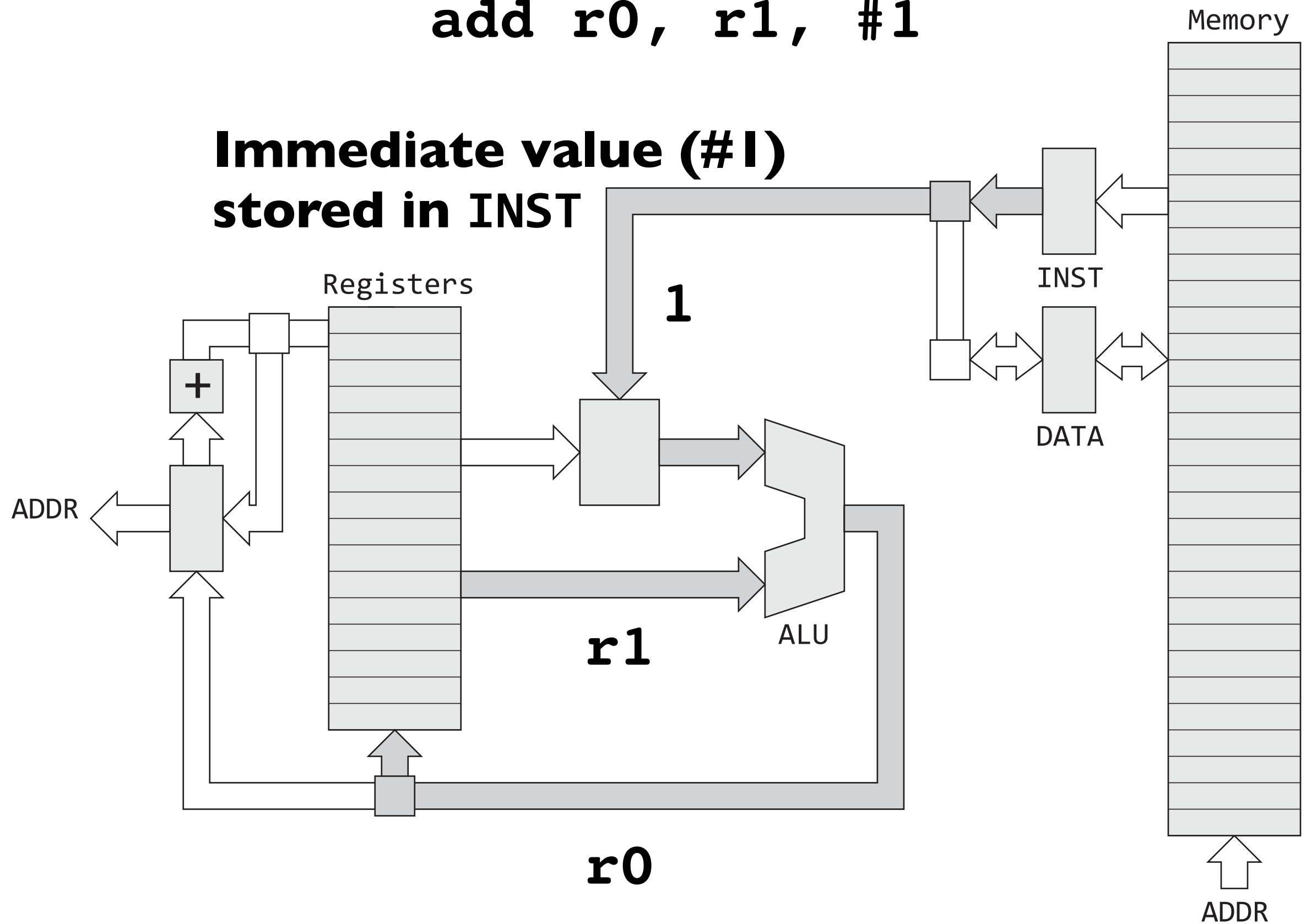


**The 'little-endian' and 'big-endian' terminology which is used to denote the two approaches [to addressing memory] is derived from Swift's *Gulliver's Travels*. The inhabitants of Lilliput, who are well known for being rather small, are, in addition, constrained by law to break their eggs only at the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

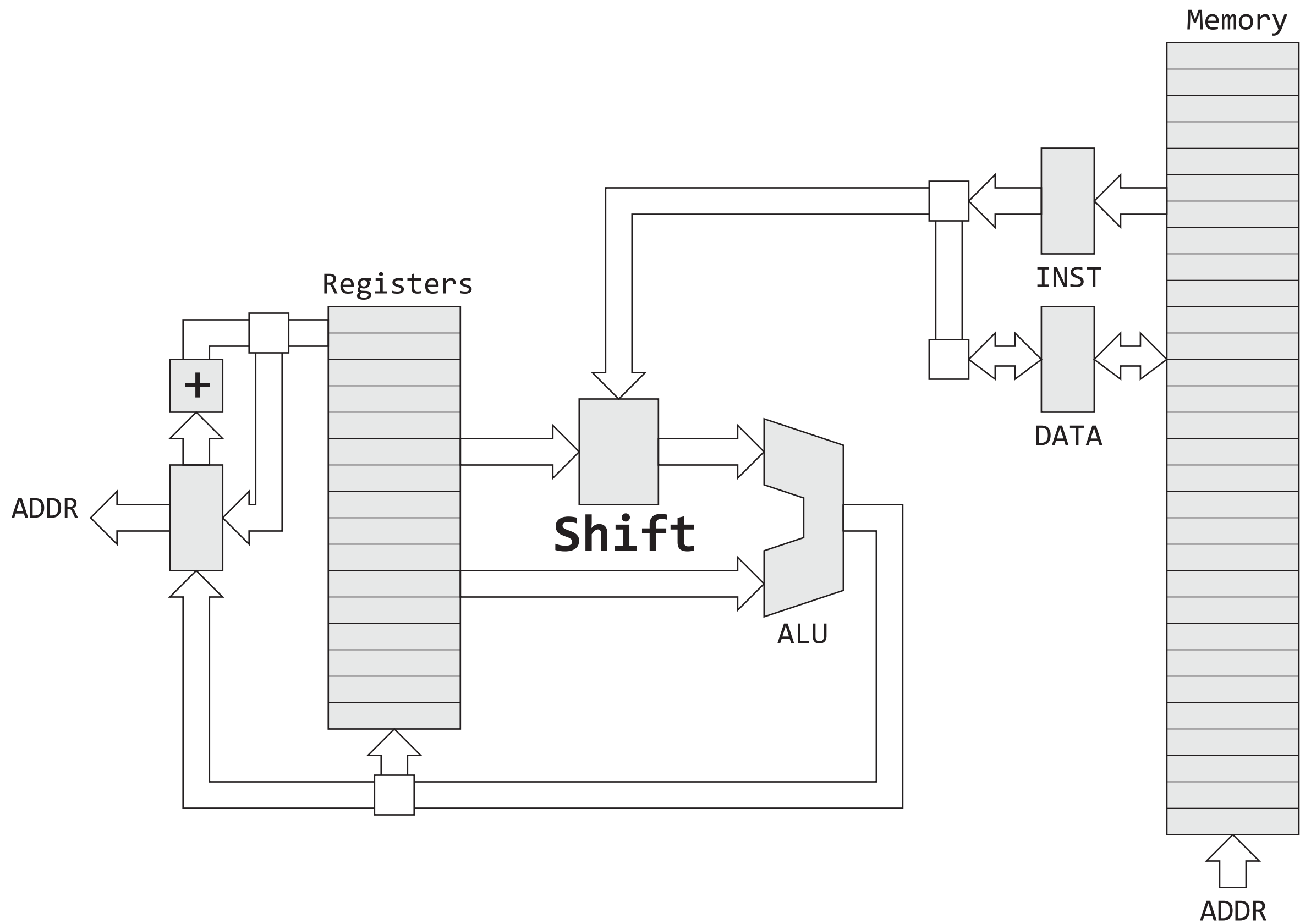
# #imm = uuuu uuuu

			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	uuu
1110	00	1	oooo	0	bbbb	aaaa	rrrr	uuuu uuuu

ROR means *Rotate Right* (imm>>>rotate)

```
# 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
```

```
0000 0000 0000 0000 0000 0000 0000 0001
```

```
0x01>>>16
```

```
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	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

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

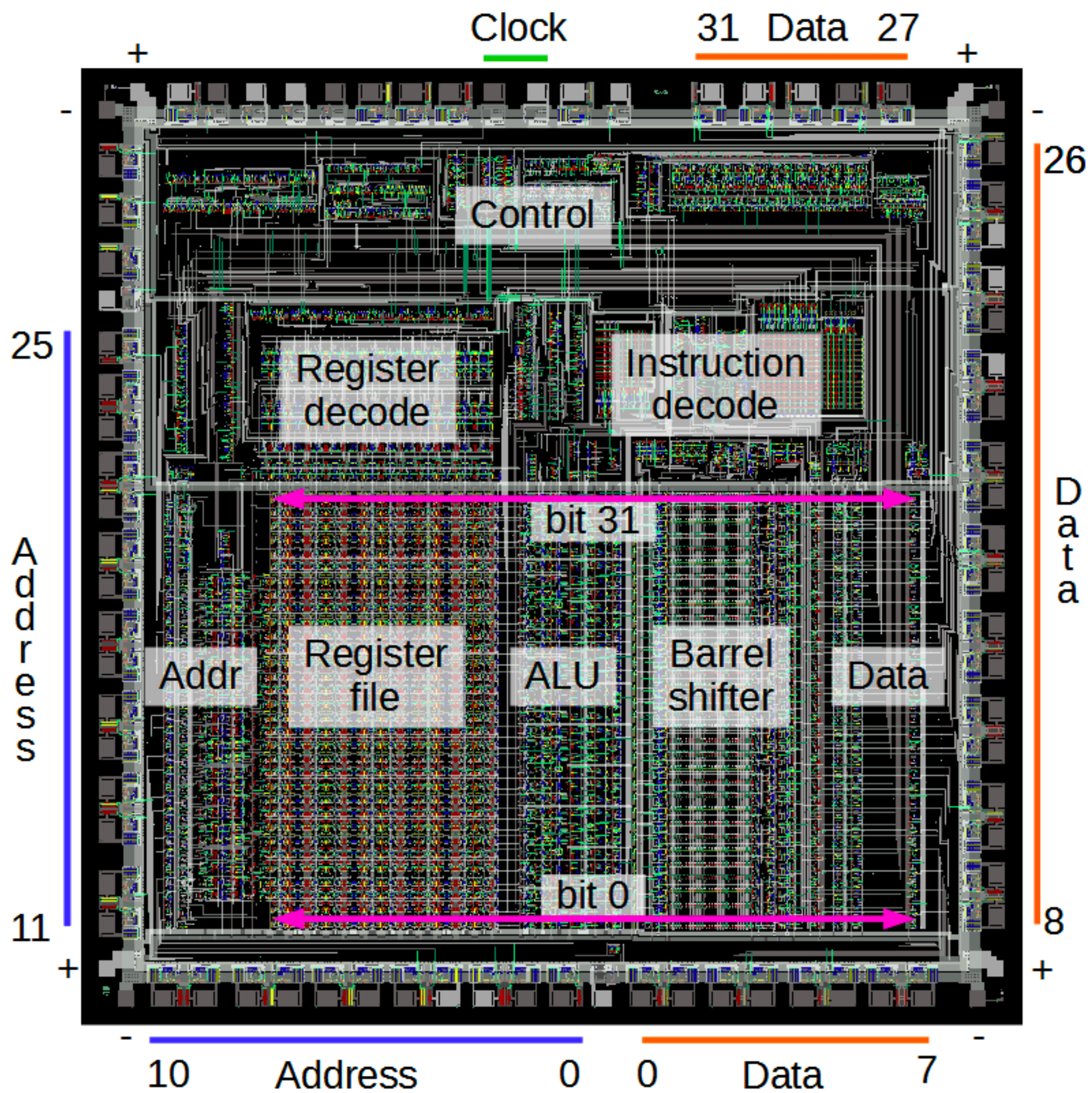
```
# data processing instruction
# ra = rb op imm
# imm = uuuu uuuu ROR (2*rrrr)
```

		op		rb		ra		ror	
1110	00	i	oooo	s	bbbb	aaaa	rrrr	uuuu	uuuu

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

		sub		r5		r7		#0x03>>>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



```
/// SET0 = 0x2020001c  
mov r0, #0x20 // r0 = 0x00000020  
lsl r1, r0, #24 // r1 = 0x20000000  
lsl r2, r0, #16 // r2 = 0x00200000  
orr r0, r1, r2 // r0 = 0x20200000  
orr r0, r0, #0x1c // r0 = 0x2020001c
```

```
// SET0 = 0x2020001c  
mov r0, #0x20000000 // 0x20>>>8  
orr r0, #0x00200000 // 0x20>>>16  
orr r0, #0x0000001c // 0x1c>>>0
```

```
/// SET0 = 0x2020001c  
mov r0, #0x20 // r0 = 0x00000020  
lsl r1, r0, #24 // r1 = 0x20000000  
lsl r2, r0, #16 // r2 = 0x00200000  
orr r0, r1, r2 // r0 = 0x20200000  
orr r0, r0, #0x1c // r0 = 0x2020001c
```

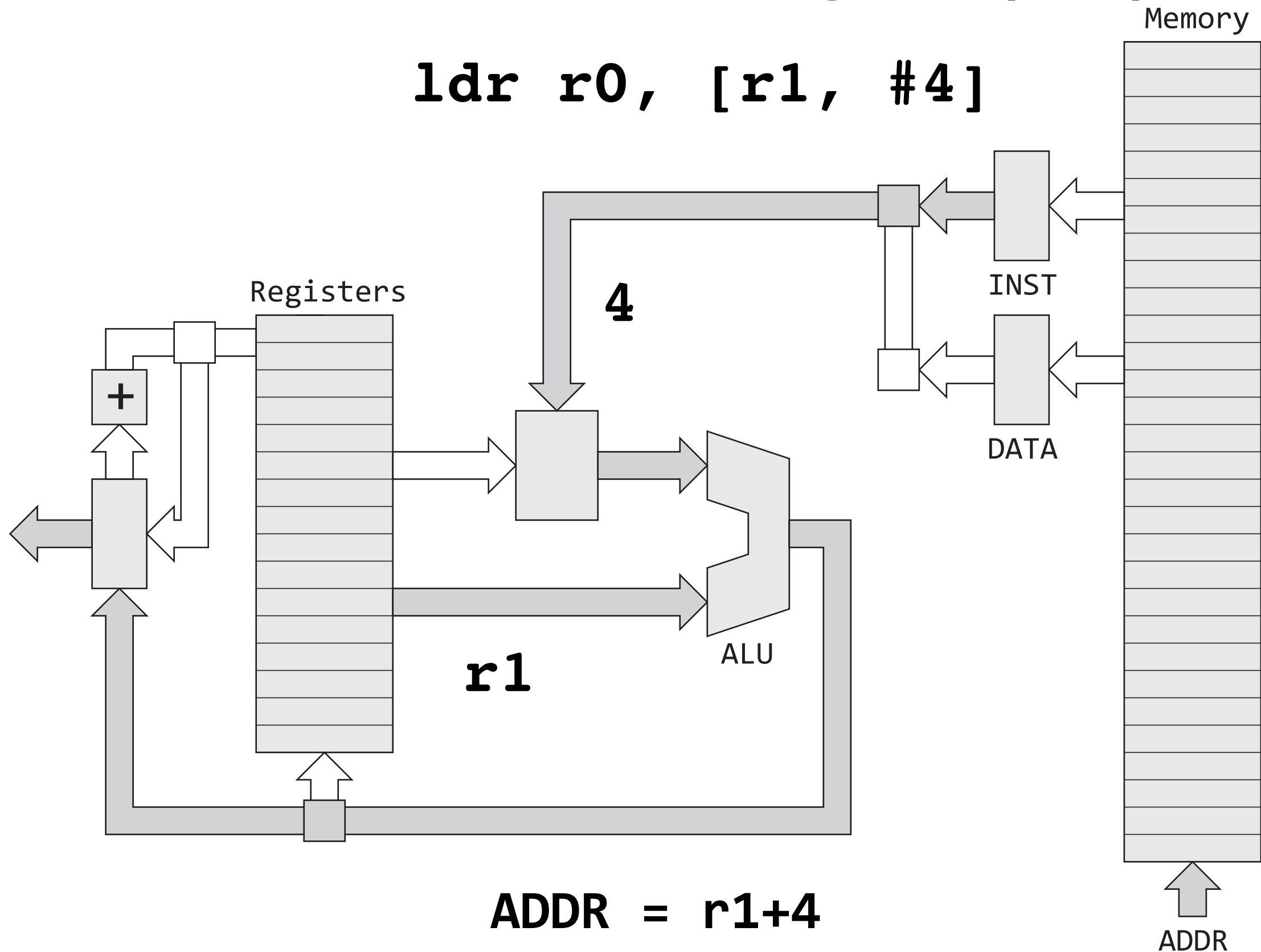
```
// SET0 = 0x2020001c  
mov r0, #0x20000000 // 0x20>>>8  
orr r0, #0x00200000 // 0x20>>>16  
orr r0, #0x0000001c // 0x1c>>>0
```

Using the barrel shifter lets us make the  
code 40% shorter (and 40% faster)



# Load from Memory to Register (LDR)

**ldr r0, [r1, #4]**



**ADDR = r1+4**

**DATA = Memory[ADDR]**

```
// configure GPIO 20 for output
```

```
ldr r0, =0x20200008
```

```
mov r1, #1
```

```
str r1, [r0]
```

```
// set bit 20
```

```
ldr r0, =0x2020001C
```

```
mov r1, #0x00100000
```

```
str r1, [r0]
```

```
anna: b anna
```

```
// configure GPIO 20 for output  
ldr r0, [pc + 20]  
mov r1, #1  
str r1, [r0]
```

```
// set bit 20  
ldr r0, [pc + 12]  
mov r1, #0x00100000  
str r1, [r0]
```

```
anna: b anna
```

```
.word 0x20200008  
.word 0x2020001C
```



```
// 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]
```

```
anna: b anna
```

Some students think that "loop" is the necessary name, but it's just a label!

```
FSEL2: .word 0x20200008
```

```
SET0: .word 0x2020001C
```

We can label our words, which gives us nicely named constants

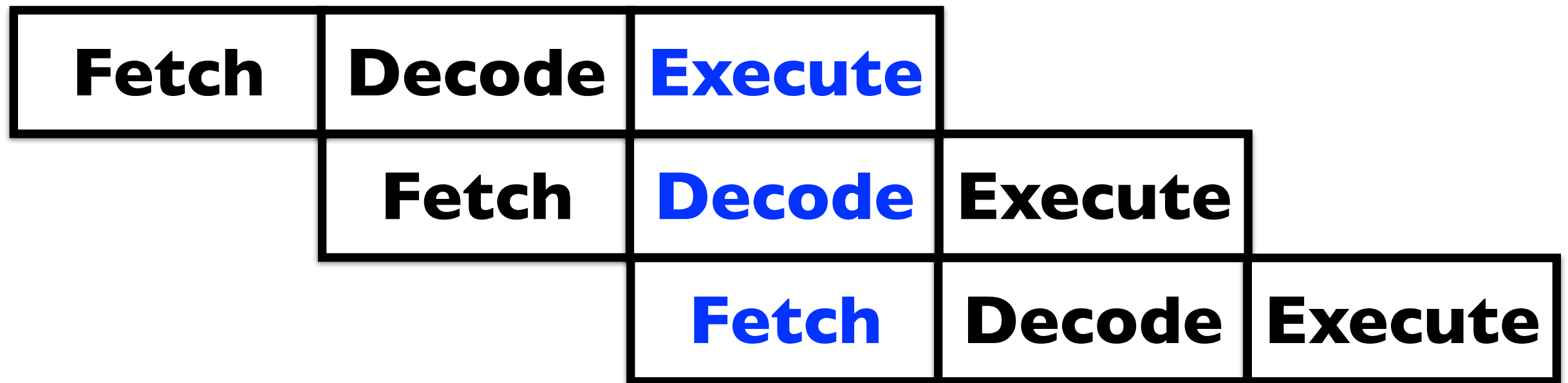
# **3 steps to run an instruction**

<b>Fetch</b>	<b>Decode</b>	<b>Execute</b>
--------------	---------------	----------------

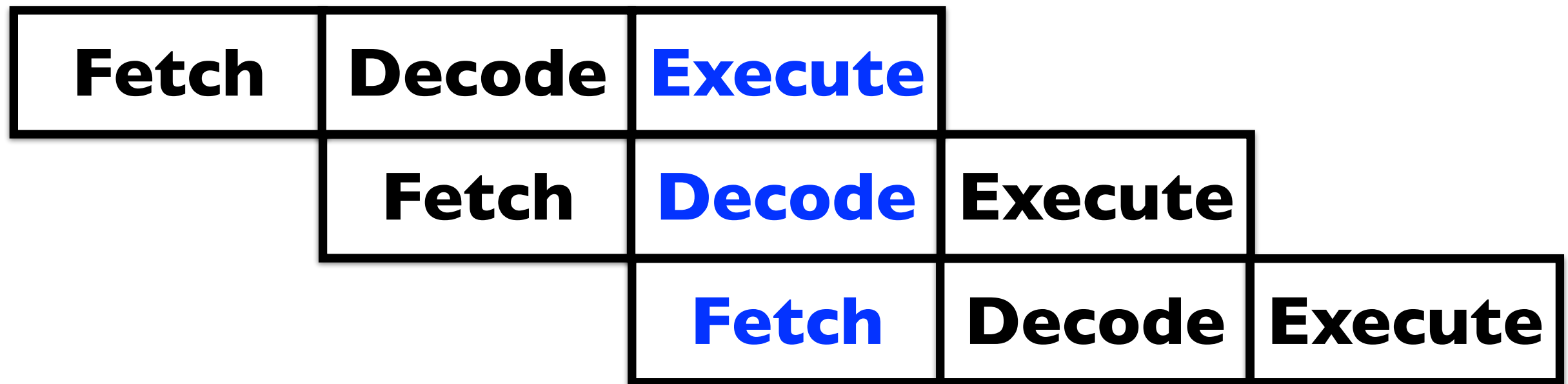
**3 instructions takes 9 steps**



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

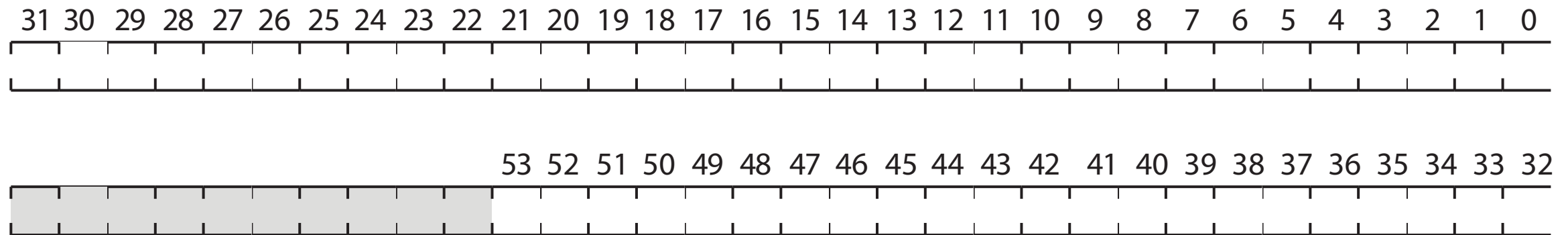


**To speed things up,  
steps are overlapped ("pipelined")**



**PC value in the executing instruction is equal to  
the pc value of the instruction being fetched -  
which is 2 instructions ahead (PC+8)**

**Blink**



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

```
// Turn on LED connected to GPIO20
```

```
ldr r0, SET0
```

```
str r1, [r0]
```

```
// Turn off LED connected to GPIO20
```

```
ldr r0, CLR0
```

```
str r1, [r0]
```

```
...
```

```
SET0: .word 0x2020001C
```

```
CLR0: .word 0x20200028
```

 We can label our words, which gives  
us nicely named constants

```
// Configure GPIO 20 for OUTPUT
```

```
anna:
```

```
// Turn on LED
```

```
// Turn off LED
```

```
b anna
```



# **Loops and Condition Codes**

```
// define constant  
.equ DELAY, 0x3f0000
```

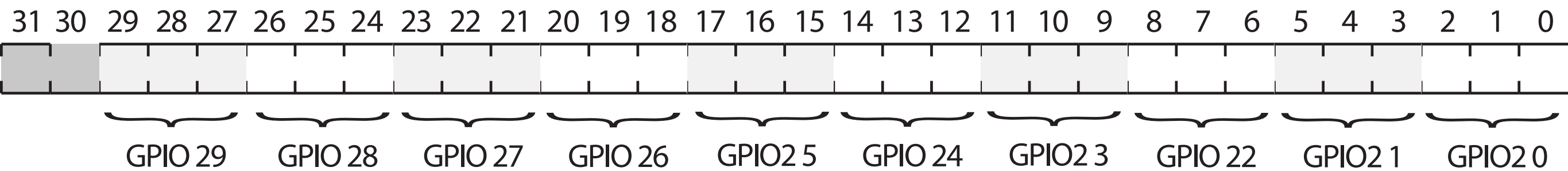
```
mov r2, #DELAY
```

```
delay:
```

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

```
    bne delay      // branch if r2 != 0
```

# **Manipulating Bit Fields**



**// Set GPIO 20 to OUTPUT**

**mov r1, #1**

**str r1, [r0]**

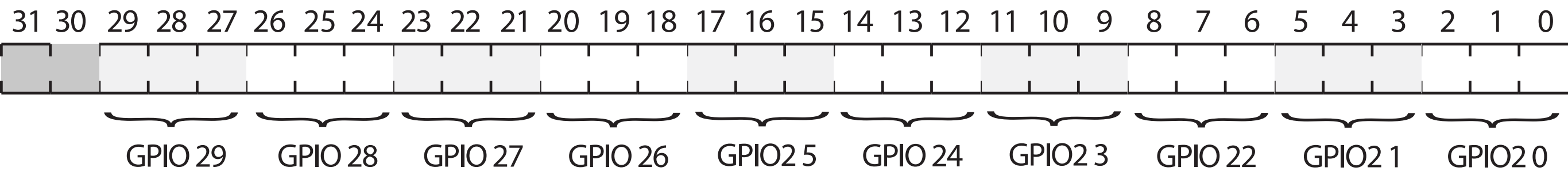
**// Set GPIO 21 to OUTPUT**

**mov r1, #(1<<3)**

**str r1, [r0]**

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

**// What mode is GPIO 20 set to now?**



**// 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 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**

# 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 - we'll see how compilers can trick you!**

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

**Normally write code in C (starting next lecture)**



# **The Fun Begins ...**

## **Lab I**

- **Read lab I instructions (now online)**
- **Assemble Raspberry Pi Kit**
- **Bring USB-C to USB-A adapter (if you need it)**

## **Assignment I**

- **Larson scanner**
- **YEAH office hours Thu 4:30-5:30pm in B02**

# **Definitive References**

**BCM2835 peripherals document + errata**

**Raspberry Pi schematic**

**ARM I I / ARMv6 reference manual**

**see Resources on [cs107e.github.io](https://cs107e.github.io)**

# **Extra Material on Branches**

# **Branch Instructions**

# Condition Codes

**Z - Result is 0**

**N - Result is  $<0$**

**C - Carry generated**

**V - Arithmetic overflow**

***Carry and overflow will be covered later***

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 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