ARM Instructions

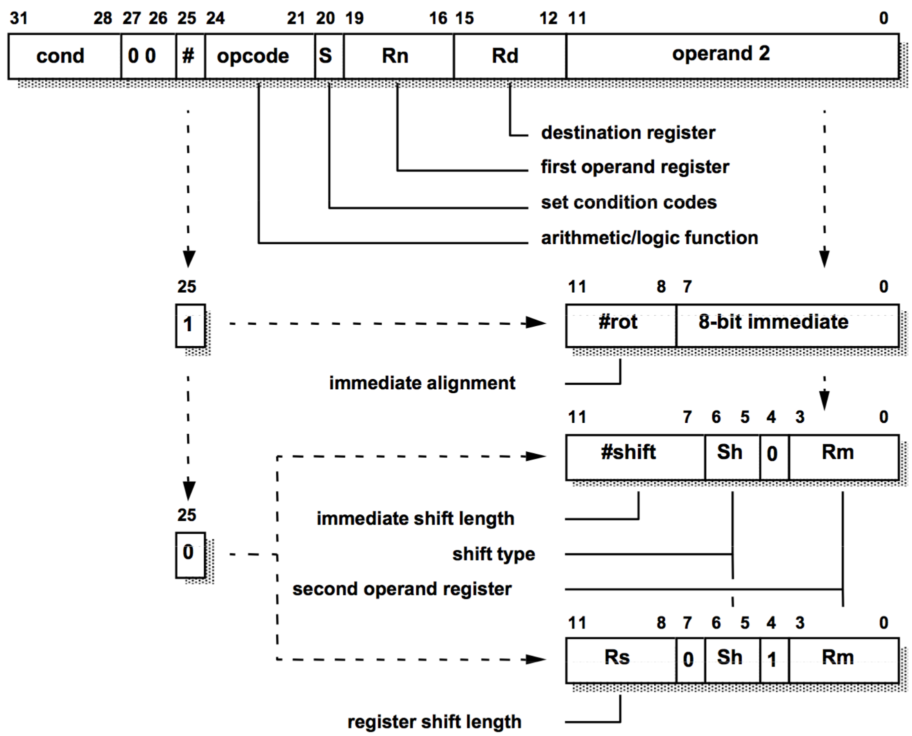
```
// Program to turn on an LED on GPIO 20

ldr r0, =0x20200008
mov r1, #1
str r1, [r0]

ldr r0, =0x2020001C
mov r1, #(1<<20)
str r1, [r0]

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



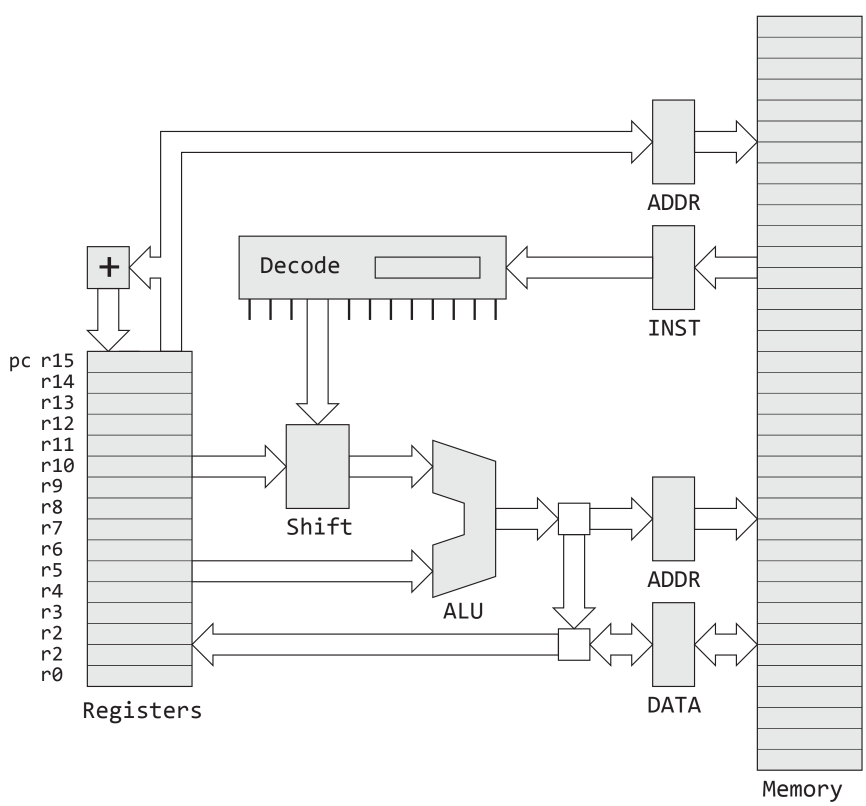
From ARM architecture manual ...

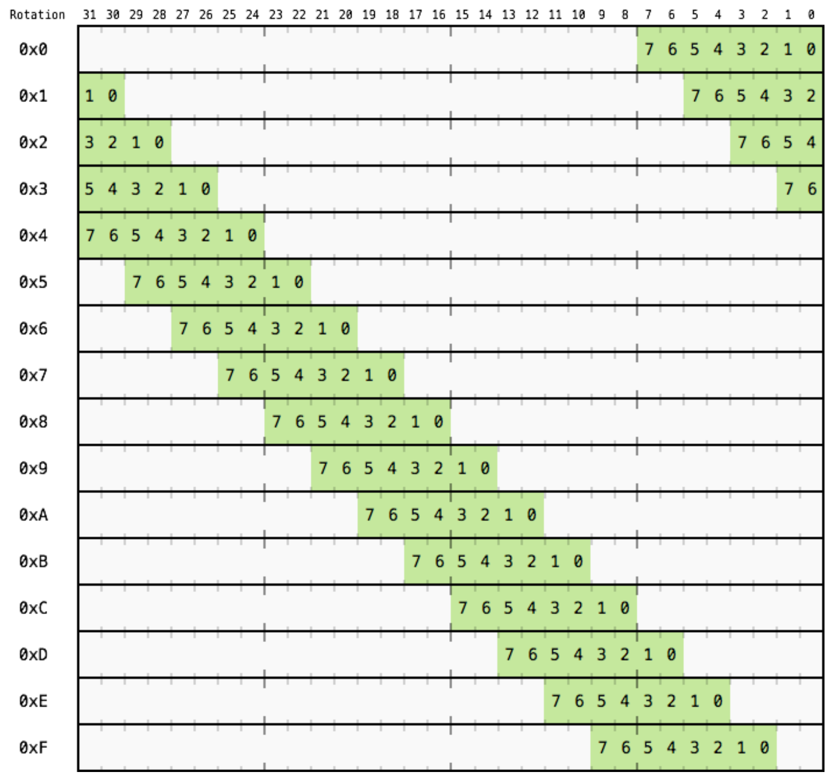
```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu

          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
```






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

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

add r1, r0, #0x10000
          add    r0   r1        #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
1110 00 1 oooo 0 bbbb aaaa iiii uuuu uuuu

add r1, r0, #0x10000
          add    r0   r1           #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*iiii)

# Remember that ra is the result

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

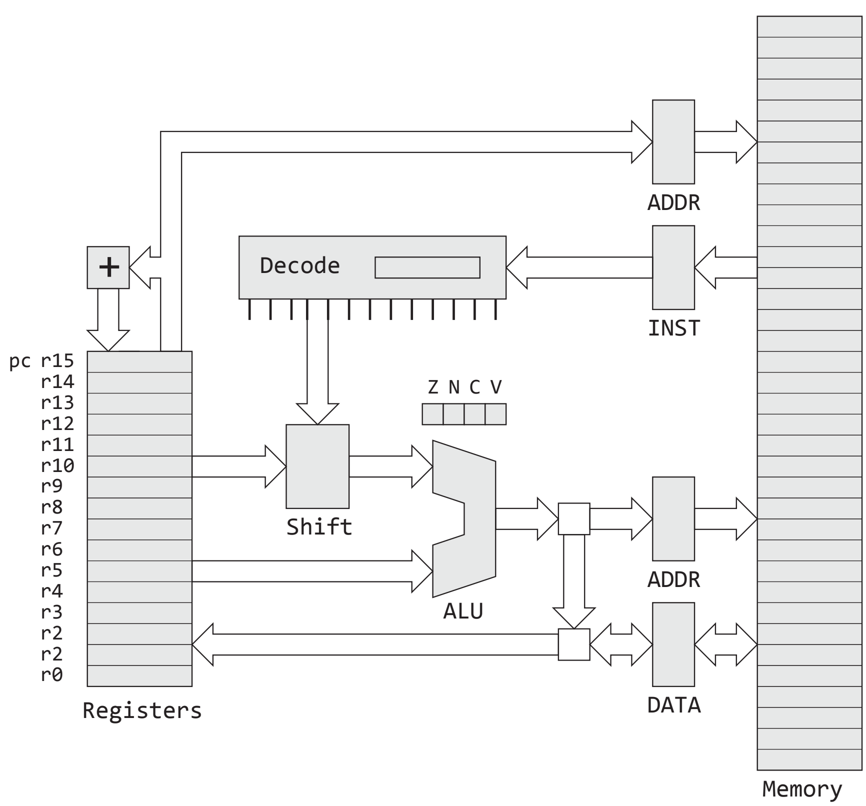
```
# data processing instruction
#  ra = rb op imm
# imm = uuuu uuuu ROR (2*iiii)

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

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

```
// 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 an address
```

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

          op   s rb   ra
1110 00 1 oooo s bbbb aaaa iiii uuuu uuuu

subs r1, r1, #1
          sub  s    1    1    0         1
1110 00 1 0010 1 0001 0001 0000 0000 0001

E2 51 10 01
```



Condition Codes

Z - Result is 0

N - Result is <0

C - Carry generated

V - Arithmetic overflow

```
# data processing instruction
#  ra = rb op #imm
# #imm = uuuu uuuu ROR (2*iiii)
#
# Only execute instruction if condition
# codes is true
#

cond      op     rb   ra
cccc 00 1 oooo s bbbb aaaa iiii 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

```
# branch
cccc      addr
cccc 101L oooo oooo oooo oooo oooo oooo

b = bal = branch always
cccc      addr
1110 101L oooo oooo oooo oooo oooo oooo

bne
cccc      addr
0001 101L oooo oooo oooo oooo oooo oooo
```

Orthogonal Instructions

Any operation

Any condition code

Set or do not set condition code

Register vs. immediate

All registers the same*

Orthogonality is a very nice feature

Blink

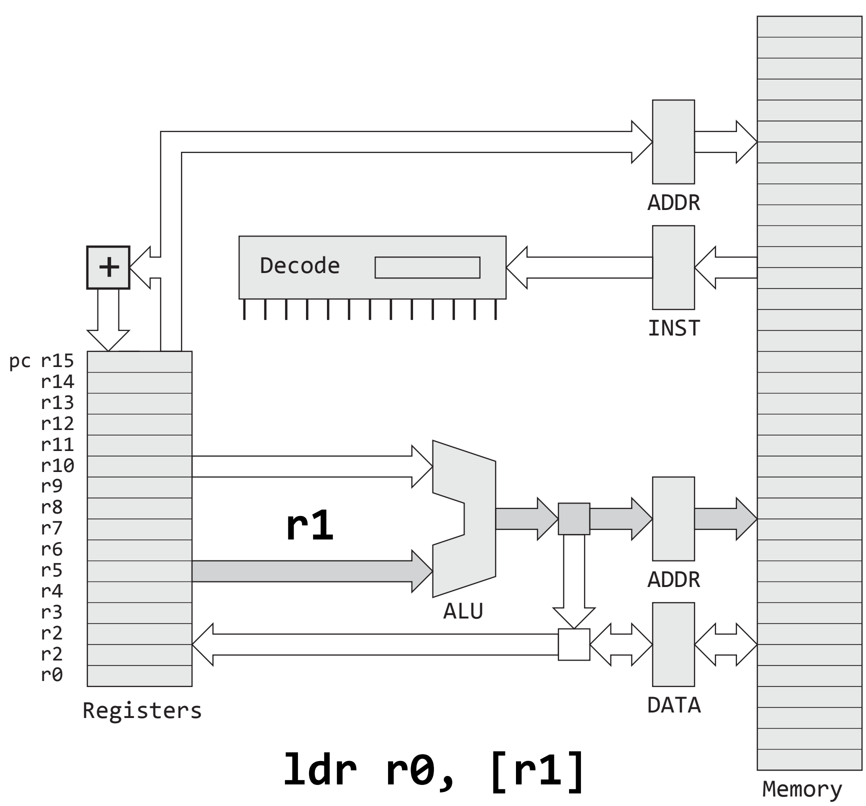
```
// Configure GPIO 20 for OUTPUT

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)
```

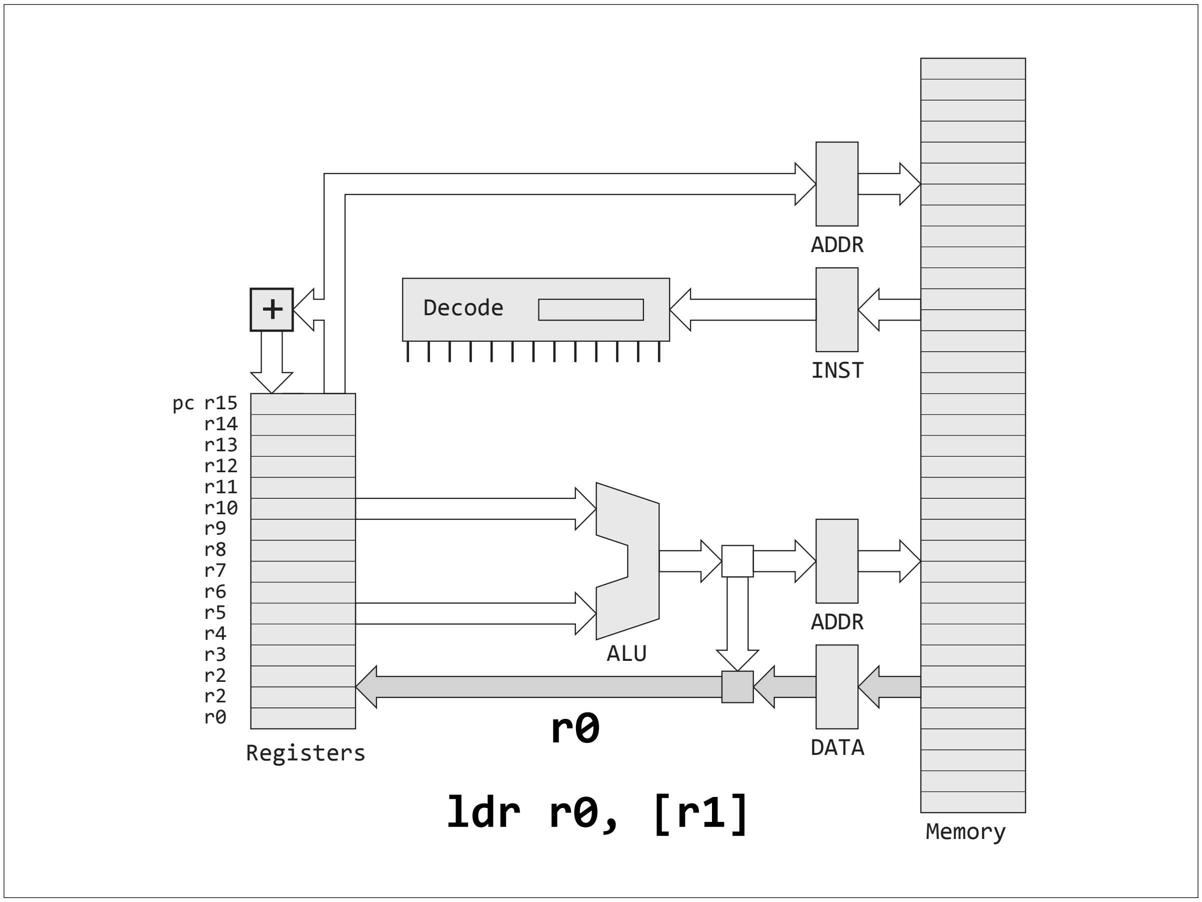
```
…

// r0 points to GPIO SET2 register
ldr r0, =0x2020001C
str r1, [r0]

// delay
mov r2, #0x3F0000
wait1:
    subs r2, #1
    bne wait1
```

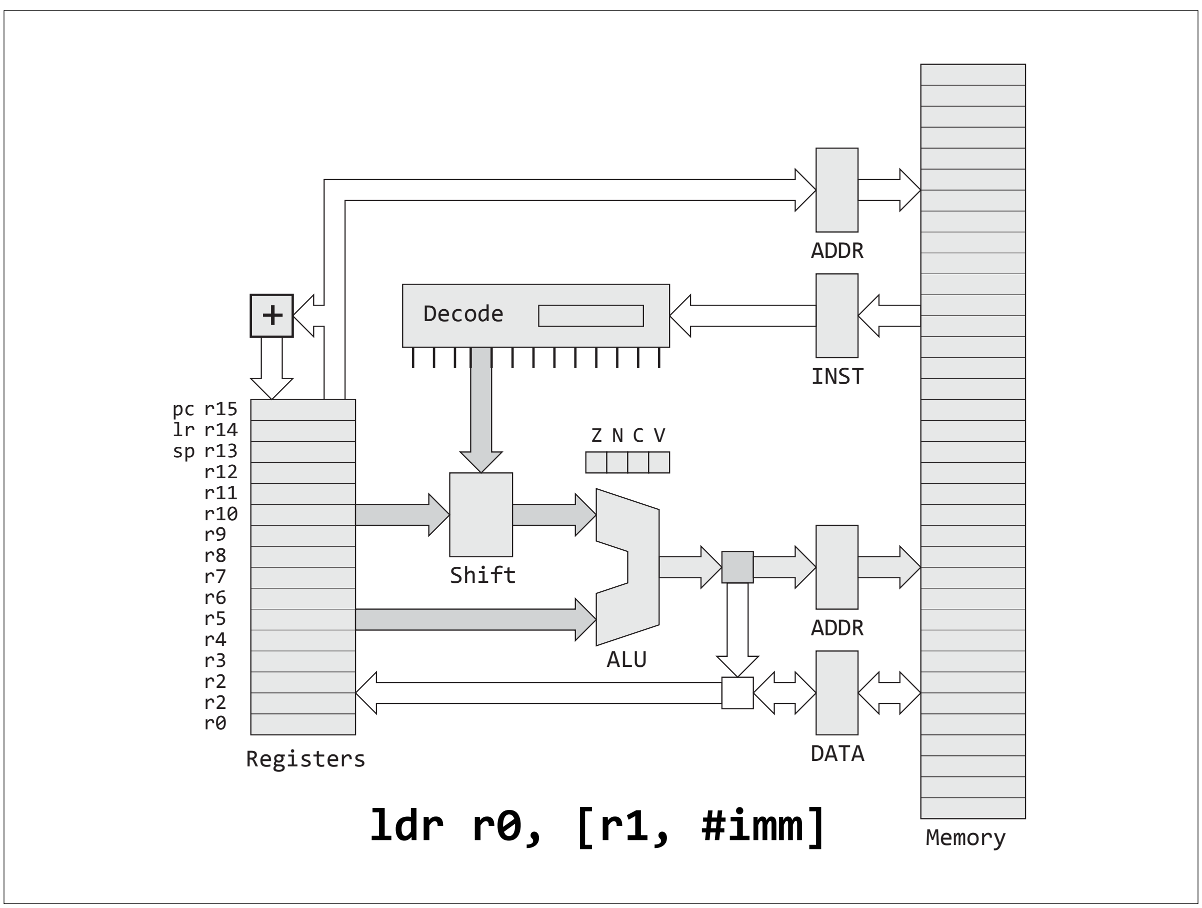
```
…

// r0 points to GPIO CLR2 register
ldr r0, =0x20200028
str r1, [r0]

// delay
mov r2, #0x3F0000
wait2:
   sub r2, #1
   bne wait2
```

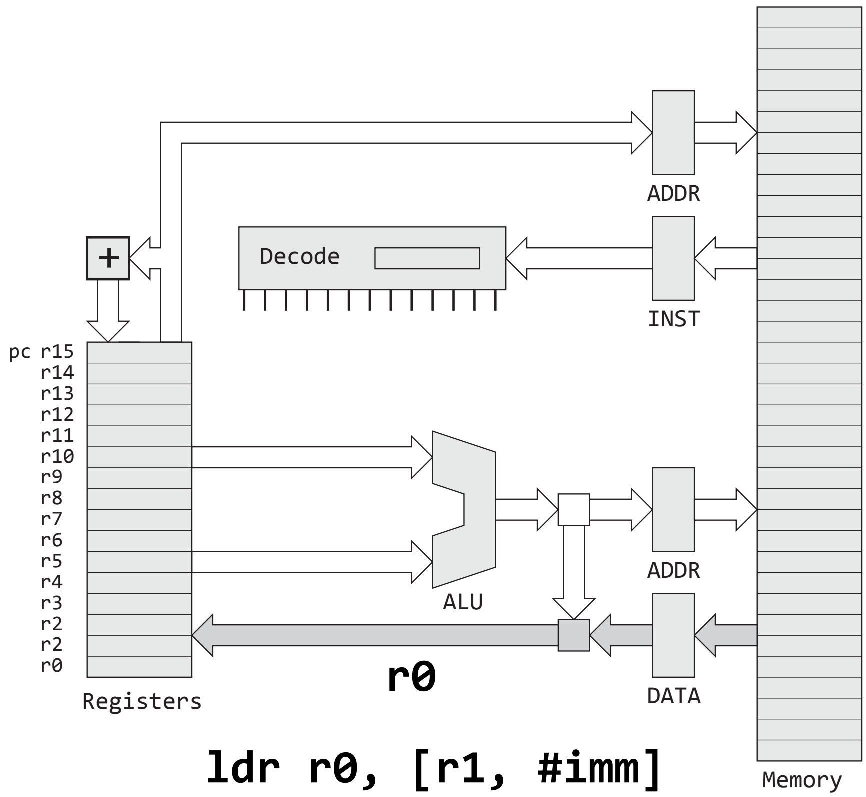
Load/Store Instructions



ldr r0, [r1]


ADDR
Decode
INST
pc r15
r14
r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r2
r0
ALU
ADDR
DATA
Registers
r0
ldr r0, [r1]
Memory


ADDR
Decode
INST
pc r15
lr r14
sp r13
r12
r11
r10
r9
r8
r7
r6
r5
r4
r3
r2
r2
r0
Z N C V
Shift
ALU
ADDR
DATA
Registers
ldr r0, [r1, #imm]
Memory



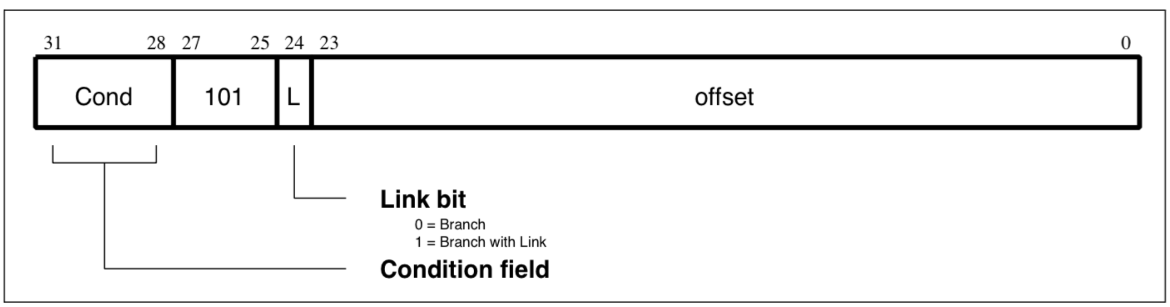
ldr r0, [r1, #imm]

```
// 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, #0x1c]
 4: e3a01001   mov r1, #1
 8: e5801000   str r1, [r0]
 c: e59f000c   ldr r0, [pc, #0x20]
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, #0x1c]
 4: e3a01001   mov r1, #1
 8: e5801000   str r1, [r0]
 c: e59f000c   ldr r0, [pc, #0x20]
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

Fetch	Decode	Execute

Phases are pipelined

Fetch	Decode	Execute		
	Fetch	Decode	Execute	
		Fetch	Decode	Execute

PC is fetching 2 instructions ahead 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