

# CSCI 6461 Computer Systems Architecture

Machine Language

# Assembly in Convenient

2

- Assembly language is convenient for humans to read.
- Digital circuits understand only 1's and 0's.
- A program written in assembly is translated to machine language.
- ARM32 uses 32-bit instructions.
- ARM defines three main instruction formats:
  - Data-processing instructions
  - Memory instructions
  - Branch instructions

# How to encode instructions?

3

- 32-bit data, 32-bit instructions
- For design simplicity, would prefer a single instruction format but...
  - Different Instructions have different needs
- Multiple instruction formats allow flexibility
  - ADD, SUB: use 3 register operands
  - LDR, STR: use 2 register operands and a constant
- Number of instruction formats kept small
  - Smaller is faster

# Interpreting Machine Code

4

- All three formats start with a 4-bit condition field and a 2-bit op.
- The best place to begin is to look at the op.
- If it is 00, then the instruction is a data-processing instruction.
- If it is 01, then the instruction is a memory instruction. If it is 10, then it is a branch instruction.
- Based on that, the rest of the fields can be interpreted

# Data processing Instructions

## Mathematical and Logical Operations

# Data Processing Instructions

6

- Data-processing instructions have
  - a destination register (Rd)
  - a 1st source register (Rn)
  - a 2nd source that is either an immediate or a register, possibly shifted (Src2)
- 6 fields: cond, op, funct, Rn, Rd, Src2

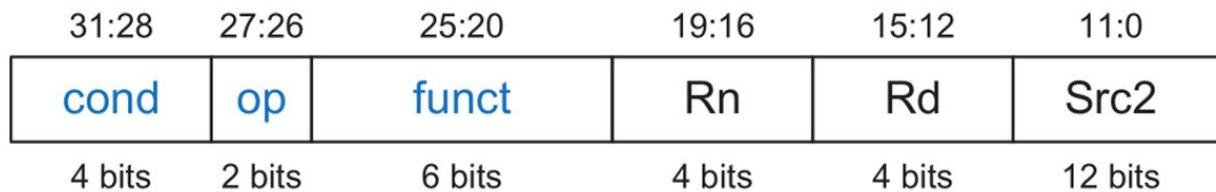
31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

op – the opcode (operation code)

op is 00 for data-processing instructions

# Detail

7



cond	Mnemonic
0000	EQ
0001	NE
0010	CS/HS
0011	CC/LO
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL (or none)

cond – conditional execution based on flags

eg, cond = 1110 for **unconditional** instructions

funct – function code

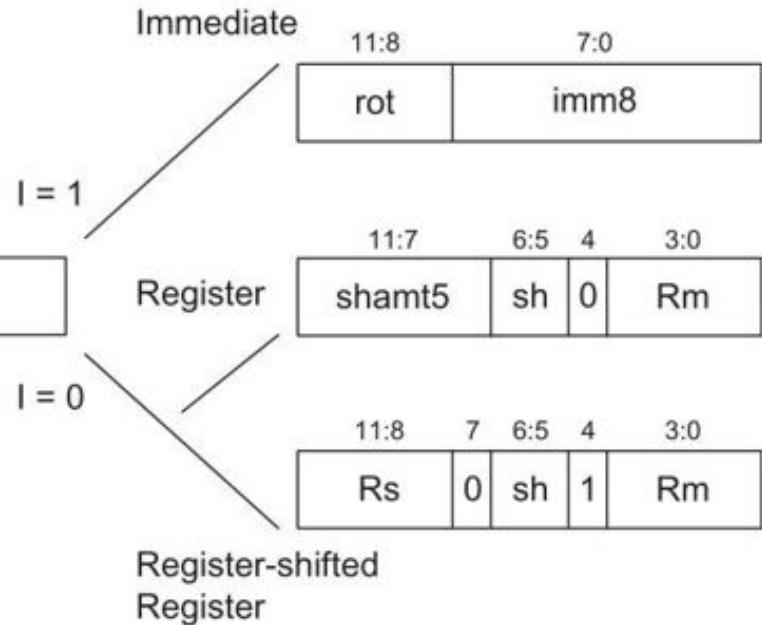
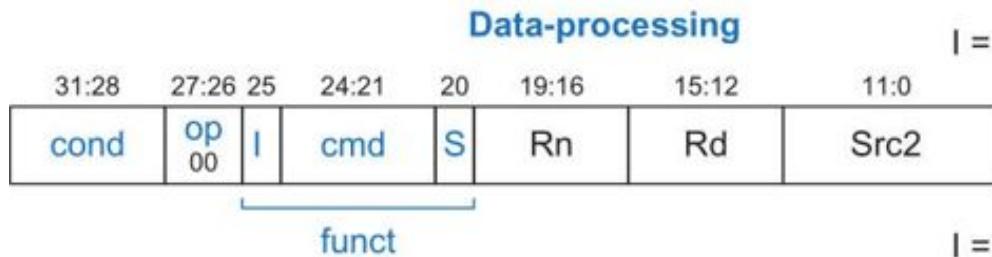
Rn – the 1st source register Src2 – the 2nd source

Rd – the destination register

# funct has 3 parts

8

funct has 3 subfields: I, cmd, S



# Some of the Operations

9

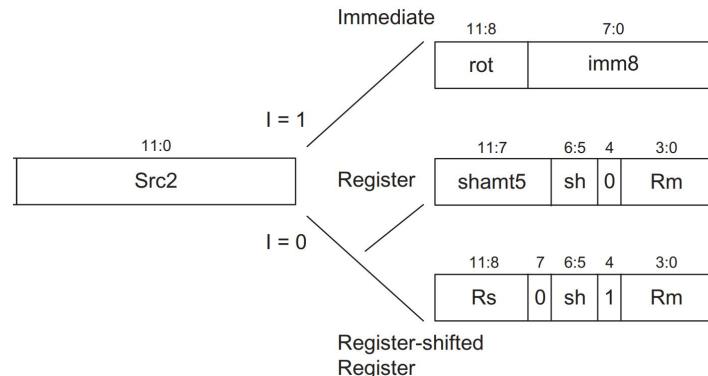
Note that multiply is missing. The implementation of mul requires a different approach discussed briefly later

cmd	Name	Description
0000	AND Rd, Rn, Src2	Bitwise AND
0001	EOR Rd, Rn, Src2	Bitwise XOR
0010	SUB Rd, Rn, Src2	Subtract
0011	RSB Rd, Rn, Src2	Reverse Subtract
0100	ADD Rd, Rn, Src2	Add
0101	ADC Rd, Rn, Src2	Add with Carry
0110	SBC Rd, Rn, Src2	Subtract with Carry
0111	RSC Rd, Rn, Src2	Reverse Sub w/ Carry
1000 ( $S = 1$ )	TST Rd, Rn, Src2	Test
1001 ( $S = 1$ )	TEQ Rd, Rn, Src2	Test Equivalence
1010 ( $S = 1$ )	CMP Rn, Src2	Compare
1011 ( $S = 1$ )	CMN Rn, Src2	Compare Negative
1100	ORR Rd, Rn, Src2	Bitwise OR

# Note

10

- S = 1 set the condition flags cmd
- Src2 can be
  - an immediate
  - a register Rm optionally shifted by a constant **shamt5**
  - a register Rm shifted by another register Rs



# Example: ADD R5, R6, R7

11

- Consider the instruction ADD R5, R6, R7
- **cond = 1110** (14) for unconditional execution
- **op = 00** (0) for data-processing instructions
- **cmd = 0100** (4) for ADD
- Src2 is a register so **I=0**
- **Rd = 0101**, **Rn = 0110**, **Rm = 0111**
- **shamt = 0**, **sh = 0**

cond	op	I	cmd	S	Rn	Rd	shamt5	sh	Rm	
1110	00	0	0100	0	0110	0101	00000	00	0	0111

# Is this correct?

12

00000000	e0865007	<u>_start:</u>	4 add r5, r6, r7
----------	----------	----------------	------------------

Yes, **ADD R5, R6, R7** really is 0xE0865007 according to CPUlator

# Another Example: ROR R3, R5, #21

13

- Rn and Rm are the first and second source operands, respectively
- cond = 1110 (14) for unconditional execution
- op = 00 (0) for data-processing instructions
- cmd = 1101 (13) for all shifts (**LSL**, **LSR**, **ASR**, and **ROR**)
- Src2 is an immediate-shifted register so I=0
- Rd = 101, Rn = 0, Rm = 0101 (Rn is not used and should be 0)
- shamf5 = 21, sh = 11 (**11=ROR**)

1110	00	0	1101	0	0000	0011	10101	11	0	0101
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

# Is this correct?

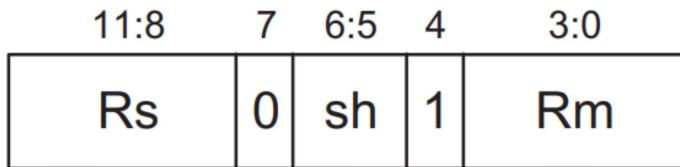
14

00000000	ela03ae5	<b>_start:</b>
		4 ror r3, r5, #21

Yes, CPUlator confirms ROR R3, R5, #21 is compiled to 0xE1A03ae5

# Register shifted Register

15



For example, ASR R5, R1, R12

Src2 is configured:

Rs = 1100, sh=10, Rm=0001(Rn is not used and should be 0)

1110	00	0	1101	0	0000	0101	1100	0	10	1	0001
cond	op	I	cmd	S	Rn	Rd	Rs	sh		Rm	

# Src2 bit 4

16

The ARM7 ISA **seems** to use the following convention for bit 4 in src2:

- Shift operations using immediates: 0
- Shift operations using registers: 1
- Other data operations using registers: 0, eg
  - ADD R5, R6, R7
  - SUB R8, R9, R10

Rd	shamt5	sh	Rm
0101	00000	00	0111
1000	00000	00	1010

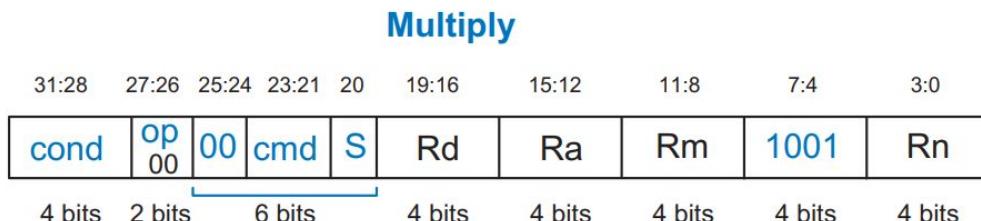
Bit fields labeled above the table:

- 15:12
- 11:7
- 6:5
- 4 (highlighted with a red dashed box)
- 3:0

# MUL operations

17

Multiply instructions use the encoding shown here. The 3-bit cmd field specifies the type of multiply



cmd	Name	Description
000	MUL Rd, Rn, Rm	Multiply
001	MLA Rd, Rn, Rm, Ra	Multiply Accumulate
100	UMULL Rd, Rn, Rm, Ra	Unsigned Multiply Long
101	UMLAL Rd, Rn, Rm, Ra	Unsigned Multiply Accumulate Long
110	SMULL Rd, Rn, Rm, Ra	Signed Multiply Long
111	SMLAL Rd, Rn, Rm, Ra	Signed Multiply Accumulate Long

# MUL example (cmd = 000)

18

mul r0, r1, r2

1110 00 000000 0000 0000 0010 **1001** 0001

Rd=0000

Ra=0000

Rm=0010

Rn=0001

# MLA example (cmd = 001)

19

mla r0, r1, r2, r3

1110 00 000010 0000 0011 0010 **1001** 0001

Rd=0000

Ra=0011

Rm=0010

Rn=0001

# MUL does not support immediates

20

From the above definition of the MUL instruction, note these types of instructions are not possible:

```
mul r0, r0, #8000
```

```
mul r0, r6, #256
```

Q: how can we get around this limitation?

# Representation of immediates

21

Data-processing instructions have an unusual immediate representation involving an 8-bit unsigned immediate, imm8, and a 4-bit rotation, rot. imm8 is rotated right by  $2 \times \text{rot}$  to create a 32-bit constant.

rot	32-bit Constant	;	11:8	7:0
0000	0000 0000 0000 0000 0000 0000 1111 1111			
0001	1100 0000 0000 0000 0000 0000 0011 1111			
0010	1111 0000 0000 0000 0000 0000 0000 1111			
...	...			
1111	0000 0000 0000 0000 0000 0011 1111 1100			

# Representation of immediates

22

This can lead to anomalies in representing immediates in data-processing instructions

```
add r0, r0, #257 // this is illegal
```

```
add r0, r0, #8000 // this is fine
```

- #257 fails because we cannot represent it in 8 bits and we cannot apply any ROR operation to get to it
- So why is #8000 perfectly fine?

# #8000 Representation

23

- #8000 cannot represent it in 8 bits but can be in 32 bits
- We start with 01111101 (125)
- Then ROR by 1101
  - Which is really  $(13 \times 2 = 26)$
  - What do we get?

00000000000000000000000000000001111101 (125)

00000000000000000000000000000001111101000000 (8000)

# Question: Rotate vs Shift

24

Why does ARM use **rotate** operations and not shift for representing immediates?

# mov example

25

Finally, move is encoded as a 1101 data processing instruction, for example:

```
mov r3, #3
```

can be readily parsed as:

```
1110 00 1 1101 0 0000 0011 000000000011
```

# Memory Instructions

LDRx / STRx

# Memory Instructions

27

Memory instructions have three operands:

- a register that is the destination on an LDR and a source on an STR
- a base register
- an offset that is either an immediate or an optionally shifted register

# Memory Instructions

28

- Memory Instructions have the same six overall fields:
  - cond, op, funct, Rn, Rd, Src2 Rn – the base register
  - Src2 – the offset
  - Rd – the destination register in a load or the source register in a store
- op is 01 for memory instructions

# Two motivating examples

29

```
ldr r0, =a  
ldr r1, [ r0 ]  
  
.data  
  
a: .word 10
```

note the differences

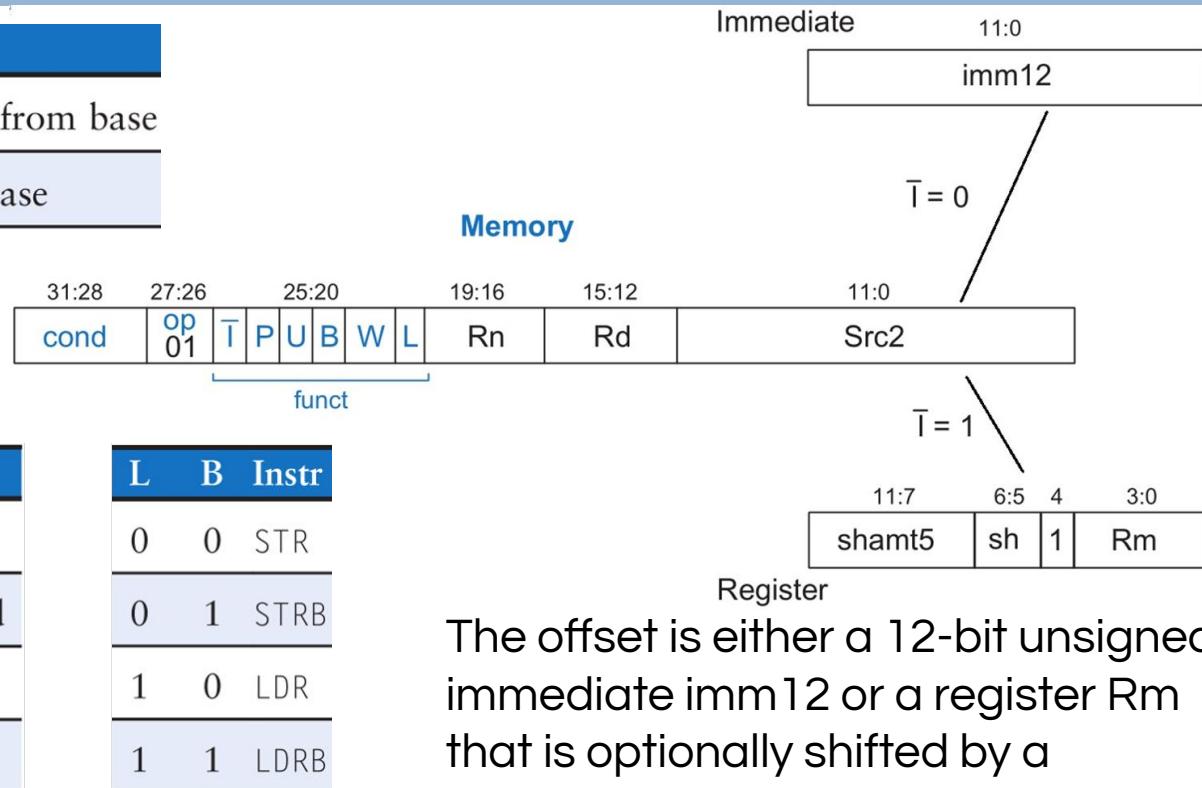
```
ldr r0, =a  
ldr r1, [ r0 ]  
mov r2, #6  
add r3, r2, r1
```

```
.data  
  
a: .word 10
```

# Memory Instructions

30

Bit	$\bar{I}$	U
0	Immediate offset	Subtract offset from base
1	Register offset	Add offset to base



# funct control bits

31

- funct is composed of six control bits: I bar, P, U, B, W, and L.
- The I-bar (immediate) and U (add) bits determine whether the offset is an immediate or register
- and whether it should be added or subtracted.
- Lets translate the following assembly language statement into machine language.

STR R11, [R5], #-26

# STR R11, [R5], #-26

32

- STR is a memory instruction, so it has an op of 01
- According to the above table L = 0 and B = 0 for STR.
- The instruction uses post-indexing, so P = 0 and W = 0.
- The immediate offset is subtracted from the base, so I bar = 0 and U = 0

## Machine Code

31:28	27:26	25:20	19:16	15:12	11:0	
1110	01	000000	0101	1011	0000 0001	1010

E      4      0      5      B      0      1      A

# Quick check

33

00000000	e405b01a	<b>_start:</b>	4 str r11, [r5], #-26
----------	----------	----------------	-----------------------

As you can see, cpulator agrees with our calculations, and the machine code is 0xE405B01A

# Solution to motivating examples

34

```
ldr r0, [ pc, #-0 ]
```

```
1110 01 0 1 0 0 0 1 1111 0000 000000000000  
E51F0000
```

```
ldr r0, [ pc, #8 ]
```

```
1110 01 0 1 1 0 0 1 1111 0000 000000001000  
E59F0008
```

```
ldr r0, =a  
ldr r1, [ r0 ]  
.data  
a: .word 10
```

```
ldr r0, =a  
ldr r1, [ r0 ]  
mov r2, #6  
add r3, r2, r1  
.data  
a: .word 10
```

note the differences

# PC Relative Addressing

## Literal Pools

# pc Relative Addressing

36

- The offset is computed as:
  - $\text{offset} = (\text{address\_of\_constant}) - (\text{PC} + 8)$
- Position-independent code (PIC)
  - code works regardless of where it's loaded (shared libraries, kernels), because the offset is relative to the current PC
- 12-bit signed immediate  $\rightarrow \pm 4\text{KB}$  range
- No extra register needed
- Fast, compact, atomic and placed into the **literal pool**

# Literal Pools

37

- ARM32 **literal pools** store
  - constants too large for immediate instruction encoding.
  - 32-bit integers, floats, addresses, and string references.
- Placed in read-only `.text` section, loaded via PC-relative `ldr`.
- Created automatically or via `.ltorg` (pseudo-instruction).
- Cannot be modified at runtime

# Literal Pool - Example

38

1	.global _start		00000000	e59f0014	ldr r0, [pc, #20] ; 0x1c
2	_start:		00000004	e5802004	str r2, [r0, #4]
3	ldr r0, =array		00000008	e59f7010	ldr r7, =0x00FFFFAA
4	str r2, [r0, #4]		0000000c	e0865007	ldr r7, [pc, #16] ; 0x20
5	ldr r7, =0x00FFFFAA		00000010	e0865007	add r5, r6, r7
6	add r5, r6, r7		00000014	e51f1000	add r5, r6, r7
7	add r5, r6, r7		00000018	e0865007	ldr r1, =array
8	ldr r1, =array				ldr r1, [pc, #-0] ; 0x1c
9					labelx:
10	labelx:		0000001c	00000028	labelx:
11	add r5, r6, r7		00000020	00ffffaa	add r5, r6, r7
12			00000024	00000000	3 ldr r0, =array
13	.ltorg		00000028	00000001	andeq r0, r0, r8, LSR #32
14			0000002c	00000002	5 ldr r7, =0x00FFFFAA
15	.data		00000030	00000003	rsceqs pc, pc, r10, LSR #31
16	array: .word 1,2,3,4,5,6		00000034	00000004	andeq r0, r0, r0
			00000038	00000005	andeq r0, r0, r1
			0000003c	00000006	andeq r0, r0, r2
					andeq r0, r0, r3
					andeq r0, r0, r4
					andeq r0, r0, r5
					andeq r0, r0, r6

# Literal Pool Updates?

39

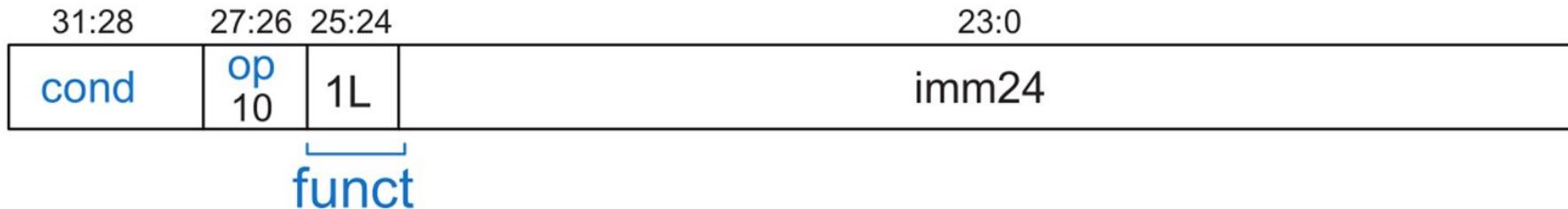
- It's not possible to update the literal pool values (the pages are mapped as `rx` only)
- So an update (eg, `str r2, [r0, #4]`) will cause the following sequence of events:
  - The CPU updates one cache line (typically 64 bytes) in the L1 data cache
  - The cache line is now marked dirty
  - Some time later, the cache line is written to RAM (once the line is evicted or sync'd)

# Branch Instructions

# Branch Instructions

41

Branch instructions use a single 24-bit signed immediate operand, imm24



- op is 10 for branch instructions. The funct field is 2 bits.
- The upper bit of funct is always 1 for branches. The lower bit, L: 1 for BL and 0 for B
- The remaining 24-bit two's complement imm24 field is used to specify an instruction address relative to PC+8

# Branch Example

42

Consider the following code

```
blt there
add r0, r1, r2
sub r0, r0, r9
add sp, sp, #8
svc #2
there:
    sub r0, r0, #1
    add r3, r3, #0x5
```

Calculate the immediate field and show the machine code for the branch instruction in the following assembly program.

# BTA - Branch Target Address

43

The processor calculates the BTA from the instruction by sign-extending the 24-bit immediate, shifting it left by 2 (to convert words to bytes), and adding it to PC + 8.

			Machine Code							
31:28	27:26	25:24	23:0							
1011	10	10	0000	0000	0000	0000	0000	0011		
cond	op funct						imm24			

# Validate with cpulator

44

00000000	ba000003	<b>_start:</b>	blt	0x14	(0x14: THERE)
----------	----------	----------------	-----	------	---------------

The results from cpulator are in agreement with our binary / hexadecimal code 0xBA000003

# Another Example

45

test:

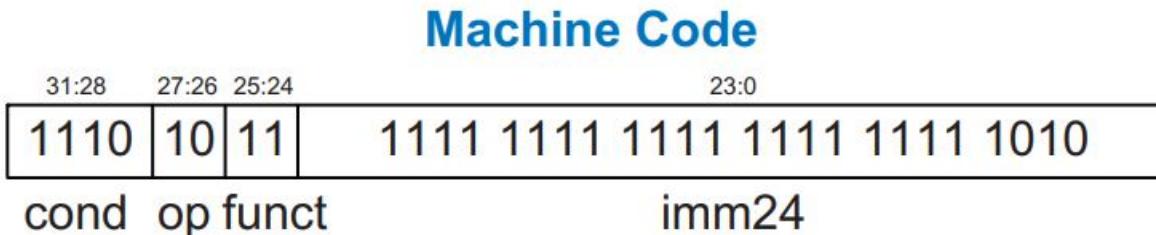
```
ldr r5, [r0, r3]
strb r5, [r1, r3]
add r3, r3, #1
ldr r6, [r1], #4
bl test
ldr r3, [r1], #4
sub r4, r3, #9
```

Calculate the immediate field and show the machine code for the branch instruction in the following assembly program.

# Moving the PC back

46

We need an immediate which is -6 (the BTA will be computed from that)



Where -6 is 1111 1111 1111 1010

# Summary

47

We have looked at the generation of binary instructions for:

- data
- branch and
- memory
- instructions.

From the binary we can easily generate the hexadecimal code and from there validate with our assembler

# Reverse Engineering

48

Of course, we can proceed in the other direction too

For example, we can parse:

1110 01 000000 0101 1011 000000011010

to str r11, [r5], #-26

with a little bit of practice

