# *Registers*

| | |
|---|---|
| **r0** | **01000001 (65)** |
| **r1** | **00001010 (10)** |
| **r2** | |
| **r3** | |

**10 10 00 01**

Which operation?
***opcode***

Where are operands?
***source registers***

Where to store result?
***destination register***

# *Instruction Set*

1. If the opcode is `00`, **add** the contents of the source registers, store the result in the destination register.
2. If the opcode is `01`, **subtract** the contents of the second source from the first, store the result in the destination register.
3. If the opcode is `10`, **multiply** the contents of the source registers, store the result in the destination register.

# What is in **r2** after this runs?

A problem!

Common approach: multiply to twice the bits, store the lower half of them.

Here would produce: `00000010` **10001010**

# *Registers*

| | |
|---|---|
| **r0** | **01000001 (65)** |
| **r1** | **00001010 (10)** |
| **r2** | |
| **r3** | |

**10 10 00 01**

# *Instruction Set*

1. If the opcode is `00`, **add** the contents of the source registers, store the result in the destination register.
2. If the opcode is `01`, **subtract** the contents of the second source from the first, store the result in the destination register.
3. If the opcode is `10`, **multiply** the contents of the source registers, store the result in the destination register.

# What is in **r2** after this runs?

Another approach: specify *two* registers for the answer.

Tough to do in our mini instruction set!

| *Machine Instruction* | | *Assembly Instruction* |
|---|---|---|
| `00 Rd Rn Rm` | `put Rn + Rm in Rd` | `add rd, rn, rm` |
| `01 Rd Rn Rm` | `put Rn - Rm in Rd` | `sub rd, rn, rm` |
| `10 Rd Rn Rm` | `put Rn * Rm in Rd` | `mul rd, rn, rm` |
| `11 Rd Imm4` | `put Imm4 in Rd` | `mov rd, #15` |
| `10 01 10 11` | `put r2 * r3 in r1` | `mul r1, r2, r3` |

What assembly instruction on the right corresponds to the machine instruction?

`01 11 10 01`

A: `mul r3, r2, r1`
B: `sub r3, r2, r1`
C: `add r1, r2, r3`
D: `add r3, r2, r1`

**program.s**

```
mov r1, #15
mov r2, #8
add r3, r1, r2
```

**$ as program.s**

Assembler →

**program.bin**

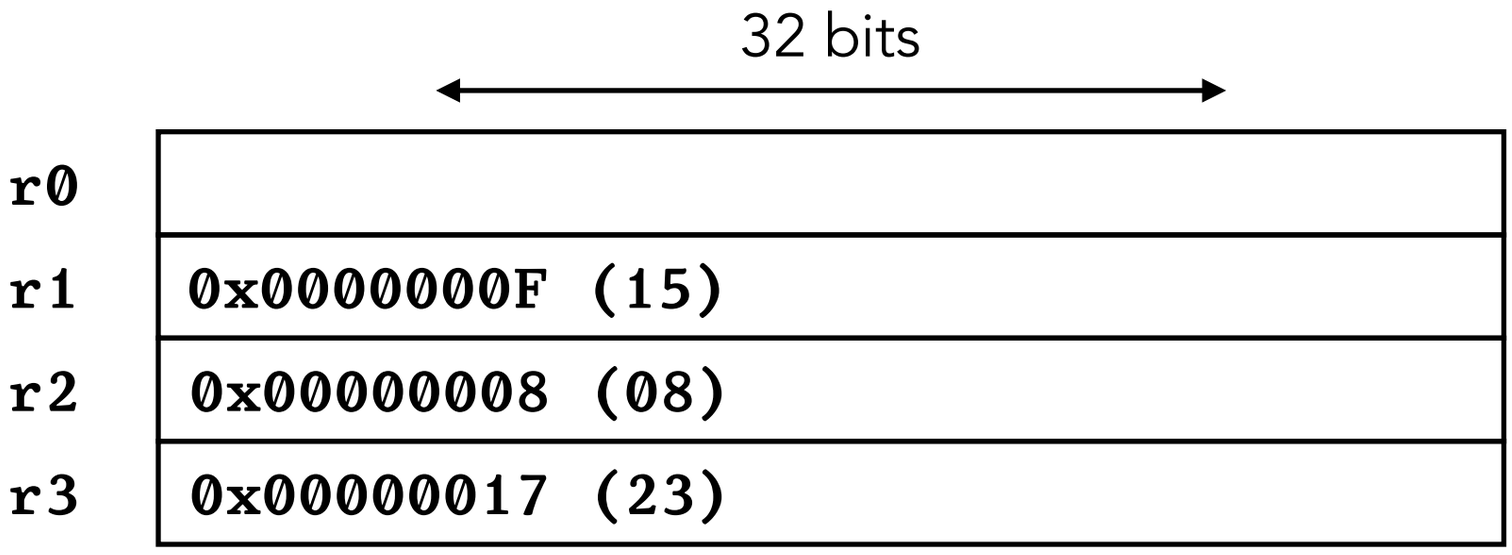
```
11 01 1111
11 10 1000
00 11 01 10
```

From now on, we will (mostly) speak in assembly instructions.

We'll call out interesting things about the binary encoding as necessary.

`program.s` is valid ARM Assembly!

# *Registers*




# *Instruction Set*

```
mov r1, #15
mov r2, #8
add r3, r1, r2
```

What is in **`r3`** after this runs?

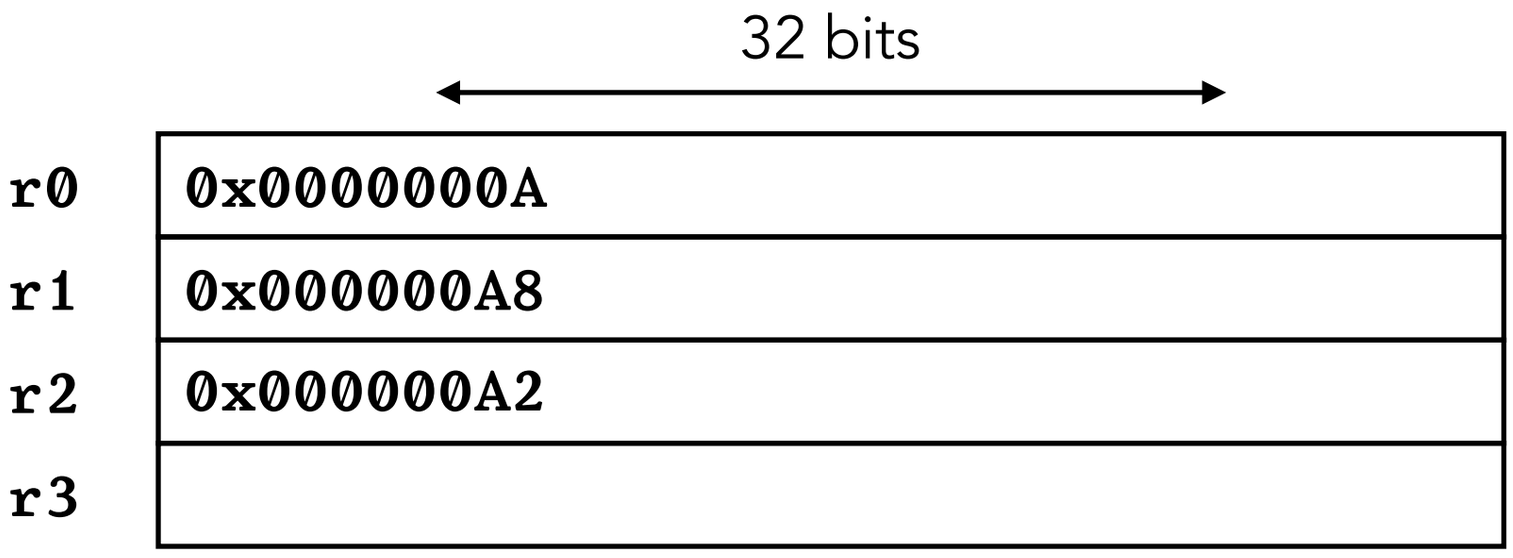
A: `0x00000017 (23)`

B: `0x0000000F (15)`

C: `0x00000008 (08)`

D: `0x0000FFF7 (-7)`

# *Registers*



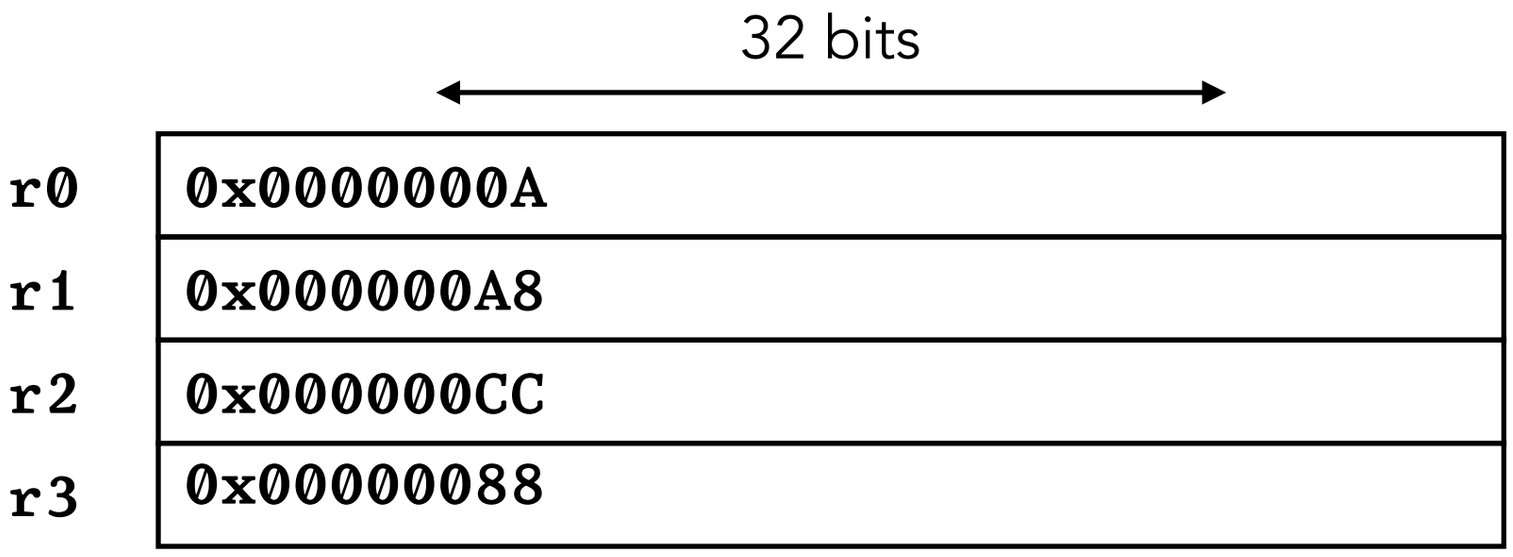

# *Instruction Set*

```
eor r2, r0, r1
```

```
00001010 (0A)
eor 10101000 (A8)
--------------
10100010 (A2)
```

"Exclusive or" results in 1
if positions are *different*.
Operates bitwise.

# *Registers*




```
orr r3, r0, r1
and r3, r2, r3
```

# What is in **r3** after this runs?

A: 0x00000088

B: 0x000000CC

C: 0x000000AA

D: 0x000000BA

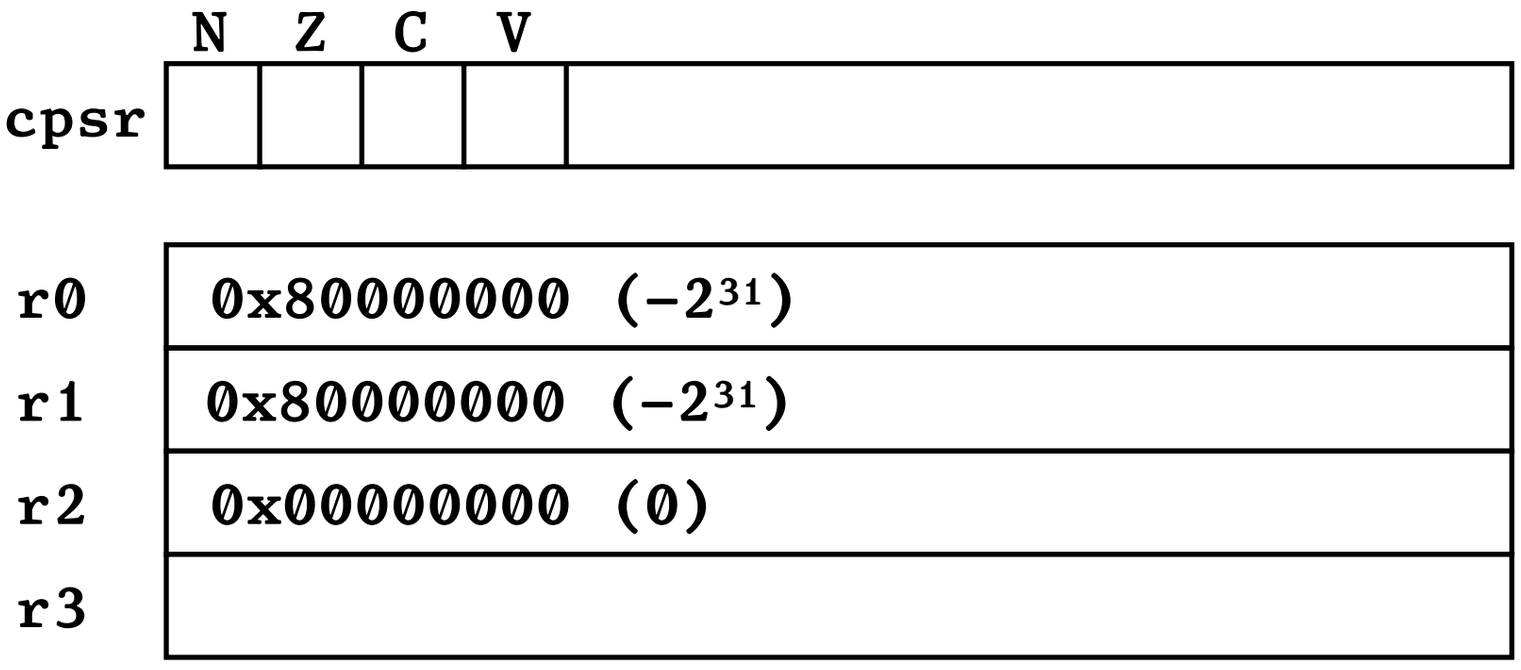
# *Instruction Set*


1. **eor**: Resulting bit is 1 if bits are different.
2. **and**: Resulting bit is 1 if bits are 1
3. **orr**: Resulting bit is 1 if either bit is 1

# Bitwise Operators

- **Exclusive or** (`eor`): Result is 1 if bits different. Also called XOR, written `e ^ e`
- **Or** (`orr`): Result is 1 if either is 1. Also written `e | e`
- **And** (`and`): Result is 1 if both are 1. Also written `e & e`
- …

# Registers




```
add r2, r1, r0
```

What is in **r2** after this runs?

A: 0x00000000 (0)
B: 0x00000000 (−2^32)
C: 0xFFFFFFFF (−1)
D: 0xFFFFFFFF (−2^32)
E: Something else

# Instruction Set


1. **adds**: Perform an addition on the two sources, store it in the destination register, and set the **Current Process Status Register**:
   - N set to 1 if the result was *Negative*
   - Z set to 1 if the result was exactly *Zero*
   - C set to 1 if the result would unsigned *"carry"*
   - V set to 1 if the result would signed *"oVerflow"*

# Registers

This was accidentally 0 in class! Sorry!

| | N | Z | C | V | |
|---|---|---|---|---|---|
| cpsr | 0 | 1 | 1 | 1 | |

| | |
|---|---|
| r0 | 0x80000000 ($-2^{31}$) |
| r1 | 0x80000000 ($-2^{31}$) |
| r2 | 0x00000000 (0) |
| r3 | |

```
adds r2, r1, r0
```

Note the **s**!

# What will the **cpsr** be after?

A: N=0, Z=1, C=1, V=0
B: N=0, Z=1, C=0, V=1
C: N=0, Z=1, C=1, V=1
D: N=1, Z=0, C=1, V=0
E: Something else

# Instruction Set

1. **adds**: Perform an addition on the two sources, store it in the destination register, and set the **Current Process Status Register**:
   - N set to 1 if the result was *Negative*
   - Z set to 1 if the result was exactly *Zero*
   - C set to 1 if the result would unsigned *"carry"*
   - V set to 1 if the result would signed *"oVerflow"*

# Registers

| | N | Z | C | V | |
|---|---|---|---|---|---|
| cpsr | 1 | 0 | 0 | 1 | |

| | |
|---|---|
| r0 | 0x40000000 ($2^{30}$) |
| r1 | 0x40000000 ($2^{30}$) |
| r2 | 0x80000000 ($-2^{31}$) |
| r3 | |

```
adds r2, r1, r0
```

Note the **s**!

# What will the **cpsr** be after?

A: N=0, Z=0, C=0, V=1
B: N=0, Z=1, C=0, V=1
C: N=1, Z=0, C=1, V=1
D: N=1, Z=0, C=0, V=1
E: Something else

# Instruction Set

1. **adds**: Perform an addition on the two sources, store it in the destination register, and set the **Current Process Status Register**:
   - N set to 1 if the result was *Negative*
   - Z set to 1 if the result was exactly *Zero*
   - C set to 1 if the result would unsigned *"carry"*
   - V set to 1 if the result would signed *"oVerflow"*

```
adds r2, r1, r0
```

Note the **s**!

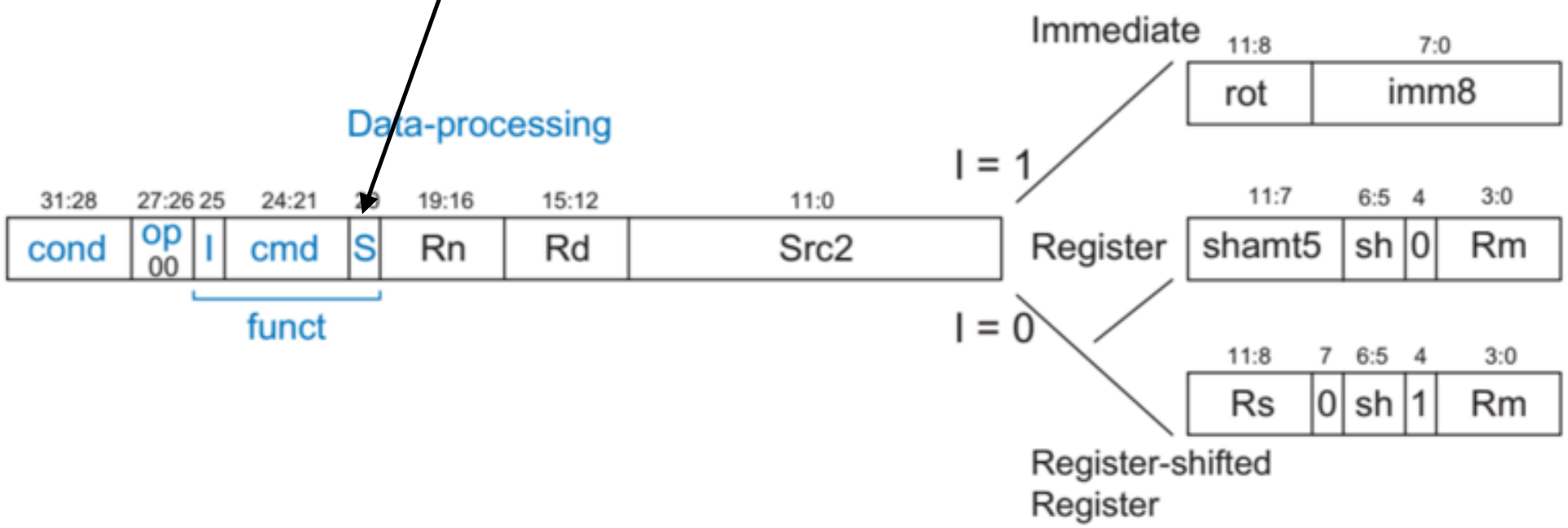



**Figure B.1 Data-processing instruction encodings**

Harris et al, *Digital Design and Computer Architecture, Second Edition*. 2016. Appendix B

# Registers

| | N | Z | C | V | |
|---|---|---|---|---|---|
| cpsr | 1 | 0 | 0 | 1 | |

| | |
|---|---|
| r0 | 0x40000000 ($2^{30}$) |
| r1 | 0x40000000 ($2^{30}$) |
| r2 | 0x80000000 ($-2^{31}$) |
| r3 | |

# Instruction Set

1. `movvs`: Move a value to a register **if the V status bit is 1**. If it is 0, do nothing.

```
adds r2, r1, r0     @ if it overflowed,
movvs r2, #0        @ set value to 0
```

**Table 6.3** Condition mnemonics

| cond | Mnemonic | Name | CondEx |
|---|---|---|---|
| 0000 | EQ | Equal | $Z$ |
| 0001 | NE | Not equal | $\overline{Z}$ |
| 0010 | CS/HS | Carry set / unsigned higher or same | $C$ |
| 0011 | CC/LO | Carry clear / unsigned lower | $\overline{C}$ |
| 0100 | MI | Minus / negative | $N$ |
| 0101 | PL | Plus / positive or zero | $\overline{N}$ |
| 0110 | VS | Overflow / overflow set | $V$ |
| 0111 | VC | No overflow / overflow clear | $\overline{V}$ |
| 1000 | HI | Unsigned higher | $\overline{Z}C$ |
| 1001 | LS | Unsigned lower or same | $Z$ OR $\overline{C}$ |
| 1010 | GE | Signed greater than or equal | $\overline{N \oplus V}$ |
| 1011 | LT | Signed less than | $N \oplus V$ |
| 1100 | GT | Signed greater than | $\overline{Z}(\overline{N \oplus V})$ |
| 1101 | LE | Signed less than or equal | $Z$ OR $(N \oplus V)$ |
| 1110 | AL (or none) | Always / unconditional | Ignored |

The suffix on the instruction is used to choose the "cond" part

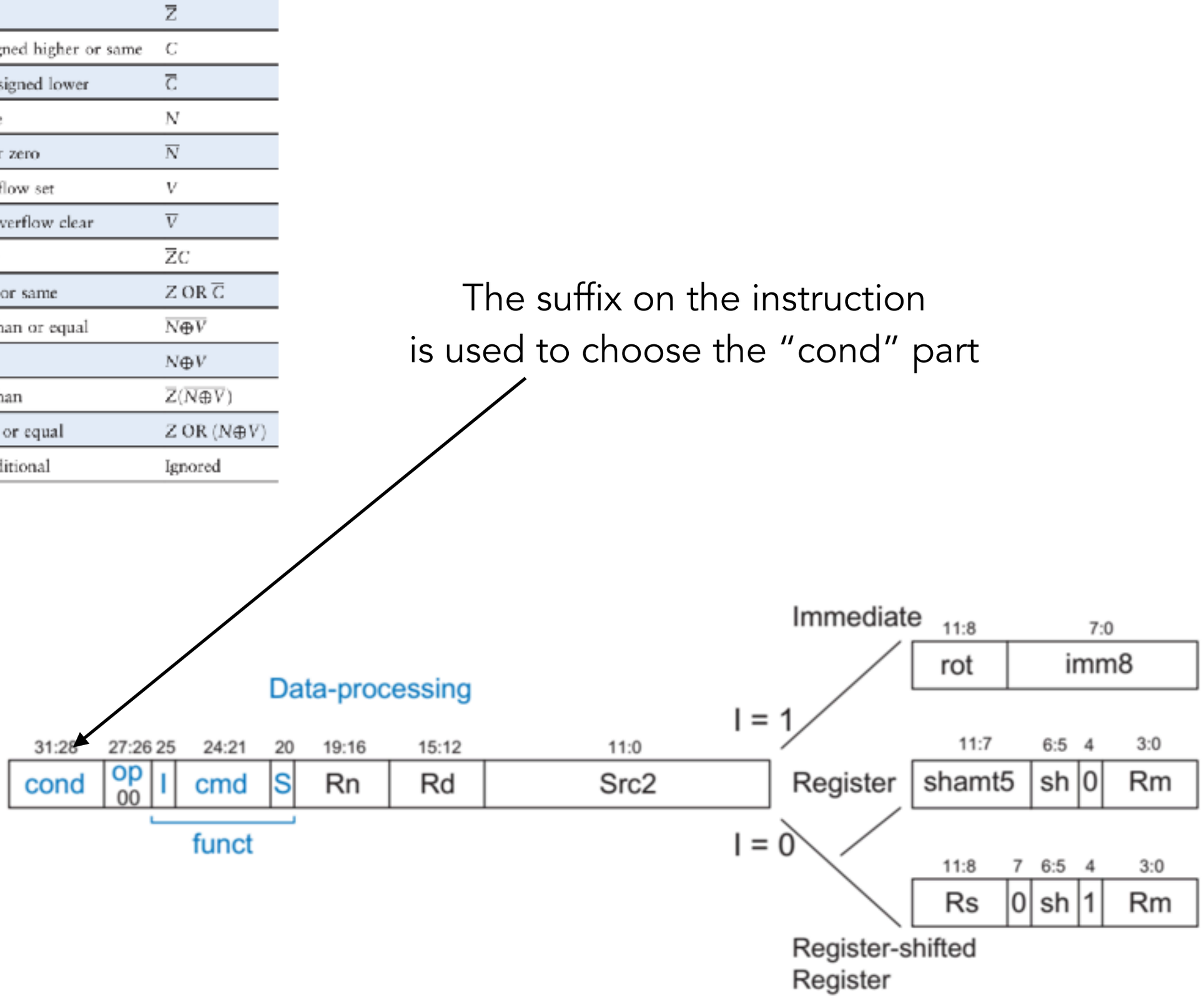



**Figure B.1** Data-processing instruction encodings

# Registers

| | N | Z | C | V | |
|---|---|---|---|---|---|
| cpsr | 1 | 0 | 0 | 1 | |

| | |
|---|---|
| r0 | 0x00000000 (0) |
| r1 | 0x7FFFFFFF ($2^{31} - 1$) |
| r2 | |
| r3 | |

# Instruction Set

1. `movvs`: Move a value to a register **if the V status bit is 1**. If it is 0, do nothing.

```
adds r1, r1, #1     @ if it overflowed,
addvs r0, #1        @ set value to 0, and
movvs r1, #0        @ increment r0
```

# What is in **`r0`**, **`r1`** after this runs?

A: `r0 = 0, r1 = 1`

B: `r0 = 1, r1 = 1`

C: `r0 = 0, r1 = 0x80000000`

D: `r0 = 1, r1 = 0x80000000`

E: Something else