# Motorola 68000 Notes

## Architecture

- **Bits:** 16/32
- **Endianness:** Big
- **Registers:**
  - 8x 32-bit data registers `D0` - `D7`
  - 7x 32-bit address registers `A0` - `A6` *
  - stack pointer (SP) *A7 is the stack pointer
  - status register (SR; the low byte of which is the CCR, or "user byte")
- **Data width:** 16 bits
- **Address width:** 24 bits

## Opcodes

### SR Conditions

| Condition | Mnemonic | Cond |
|---|---|---|
| True | T | 0 0 0 0 |
| False | F | 0 0 0 1 |
| Higher | HI | 0 0 1 0 |
| Lower or same | LS | 0 0 1 1 |
| Carry clear | CC | 0 1 0 0 |
| Carry Set | CS | 0 1 0 1 |
| Not Equal | NE | 0 1 1 0 |
| Equal | EQ | 0 1 1 1 |
| Overflow Clear | VC | 1 0 0 0 |
| Overflow Set | VS | 1 0 0 1 |
| Plus | PL | 1 0 1 0 |
| Minus | MI | 1 0 1 1 |
| Greater or Equal | GE | 1 1 0 0 |
| Less Than | LT | 1 1 0 1 |
| Greater Than | GT | 1 1 1 0 |
| Less or Equal | LE | 1 1 1 1 |

## Memory <-> Register Direction

| Direction | D | D |
|---|---|---|
| Register to memory | 0 | 1 |
| Memory to register | 1 | 0 |

## Data register <-> Effective Address Direction

| Direction | D |
|---|---|
| Dn ♦ <ea> → Dn | 0 |
| <ea> ♦ Dn → <ea> | 1 |

## Operation Sizes

| Operation Size | Suffix | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|
| Byte | `.b` | 0 0 | N/A | 0 1 |
| Word | `.w` | 0 1 | 0 | 1 1 |

| Operation Size | Suffix | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|
| Long | `.l` | 1 0 | 1 | 1 0 |

## Addressing Modes

| Addressing Mode | Format | M | Xn |
|---|---|---|---|
| Data register | $\mathrm{D}n$ | 0 0 0 | reg |
| Address register | $\mathrm{A}n$ | 0 0 1 | reg |
| Address | $(\mathrm{A}n)$ | 0 1 0 | reg |
| Address with postincrement | $(\mathrm{A}n)+$ | 0 1 1 | reg |
| Address with predecrement | $-(\mathrm{A}n)$ | 1 0 0 | reg |
| Address with displacement | $(d_{16},\ \mathrm{A}n)$ | 1 0 1 | reg |
| Address with index* | $(d_8,\ \mathrm{A}n,\ Xn)$ | 1 1 0 | reg |
| Program Counter with displacement | $(d_{16},\ \mathrm{PC})$ | 1 1 1 | 0 1 0 |
| Program counter with index* | $(d_8,\ \mathrm{PC},\ Xn)$ | 1 1 1 | 0 1 1 |
| Absolute short | `(xxx).W` | 1 1 1 | 0 0 0 |
| Absolute long | `(xxx).L` | 1 1 1 | 0 0 1 |
| Immediate | `#imm` | 1 1 1 | 1 0 0 |

*Brief Extension Word

| | | | | |
|---|---|---|---|---|
| `M` | `[Xn Xn Xn]` | `S` | `0 0 0` | `[d d d]` |

## Example: `move #43690,D1`

```
move #43690,D1
```

### Move opcode format (bits)

The `move` instruction is comprised of parts of the Operation Sizes and Addressing Modes tables. `S` in the below diagram refers to $S_3$ from the operation sizes table in particular:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | S | | Xn | | | M | | | M | | | Xn | | |

...followed by the data (16 bits).

### Values

Referencing the above format and putting all of the values together, we come up with the following opcode structure for `move #43690,D1`:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | S | | Xn | | | M | | | M | | | Xn | | |
| 0 | 0 | WORD | | n=1 | | | Dn | | | imm | | | imm | | |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

...followed by the data, which in this case is `43690 = 0xaaaa = 0b1010101010101010`, leaving us with a final opcode of:

```
 0b   0011 0010 0011 1100 1010 1010 1010 1010
=0x   32 3c aa aa
```

## Parsing Read Addresses from Opcodes

Let's say you have an instruction like `add D1,(A3)`. How do you know what data you're operating on?

The opcode for `add D1,(A3)` is constructed as such:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | Dn | | | D | S | | M | | | Xn | | |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

...where $S = S_1$ from the operation sizes table above.

We're interested in the M bits to get the address mode. We can get these by applying a bitmask to the opcode via `AND 0x38`, as `0x38 = b0000000000111000`, which will select for exactly the M bits:

```
add D1,(A0) = 93 53 = 0b 1101 0011 0101 0011

  1101 0011 0101 0011
```

```
& 0000 0000 0011 1000
---------------------
  0000 0000 0001 0000
```

To coerce this to the values in the address modes table's M column, we can perform `0b10000 >> 3` to right shift the value by 3. `0b10000 >> 3 = 0b10`, so now we can look up the Mode against the address mode table and we find that `0b10` = address mode (not to be confused with *address register* mode), so we know we're operating on the value at some memory address held in some address register.

To find the address register that holds the memory address of the data we're operating on, we can apply another bitmask to the opcode via `AND 0x7=0b111`, which will select only the Xn bits:

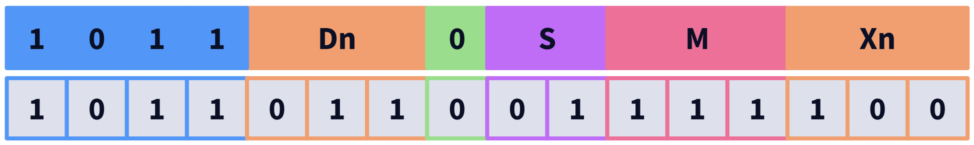
```
  1101 0011 0101 0011
& 0000 0000 0000 0111
---------------------
  0000 0000 0000 0011
```

`0b11` = 3, so now we know we're in address mode and looking at register A3, specifically the value at the address held in register A3.

## Parsing Values from Opcodes

Same as above, except let's look at `cmp #20,D3`.

`cmp #20,D3` opcode is constructed:



First, we parse the size. You know the drill. We want the bits in the positions where the 1's are here: `0b0000000011000000`, so we construct a bitmask that will grab those bits, which happens to be that exact value (this is a tautology).

```
  1011 0110 0111 1100
& 0000 0000 1100 0000
---------------------
  0000 0000 0100 0000
```

Then we right shift the value by 6 to get the size to look up against the operation size table: `0b1000000 >> 6 = 0b1`.

We're also gonna wanna parse the *effective address* of the opcode, which is the bottom 6 bits. We construct the mask, `0x3F = 0b111111`, and `AND` it with the opcode:

```
  1011 0110 0111 1100
& 0000 0000 0011 1111
---------------------
  0000 0000 0011 1100
```

Then we apply the same algorithm for reading an address to find the data.

## Instruction Timings

### Move Instructions

#### Byte/Word

| **Operand** | **Dn** | **An** | **(An)** | **(An)+** | **-(An)** | **d(An)** | **d(An,ix)** | **xxx.W** | **xxx.L** |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| **Dn** | 4 | 4 | 8 | 8 | 8 | 12 | 14 | 12 | 16 |
| **An** | 4 | 4 | 8 | 8 | 8 | 12 | 14 | 12 | 16 |
| **(An)** | 8 | 8 | 12 | 12 | 12 | 16 | 18 | 16 | 20 |
| **(An)+** | 8 | 8 | 12 | 12 | 12 | 16 | 18 | 16 | 20 |
| **-(An)** | 10 | 10 | 14 | 14 | 14 | 18 | 20 | 18 | 22 |
| **d(An)** | 12 | 12 | 16 | 16 | 16 | 20 | 22 | 20 | 24 |
| **d(An,ix)** | 14 | 14 | 18 | 18 | 18 | 22 | 24 | 22 | 26 |
| **xxx.W** | 12 | 12 | 16 | 16 | 16 | 20 | 22 | 20 | 24 |
| **xxx.L** | 16 | 16 | 20 | 20 | 20 | 24 | 26 | 24 | 28 |
| **d(PC)** | 12 | 12 | 16 | 16 | 16 | 20 | 22 | 20 | 24 |
| **d(PC,ix)** | 14 | 14 | 18 | 18 | 18 | 22 | 24 | 22 | 26 |
| **#xxx** | 8 | 8 | 12 | 12 | 12 | 16 | 18 | 16 | 20 |

#### Long

| Operand | Dn | An | (An) | (An)+ | -(An) | d(An) | d(An,ix) | xxx.W | xxx.L |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| **Dn** | 4 | 4 | 12 | 12 | 12 | 16 | 18 | 16 | 20 |
| **An** | 4 | 4 | 12 | 12 | 12 | 16 | 18 | 16 | 20 |
| **(An)** | 12 | 12 | 20 | 20 | 20 | 24 | 26 | 24 | 28 |
| **(An)+** | 12 | 12 | 20 | 20 | 20 | 24 | 26 | 24 | 28 |
| **-(An)** | 14 | 14 | 22 | 22 | 22 | 26 | 28 | 26 | 30 |
| **d(An)** | 16 | 16 | 24 | 24 | 24 | 28 | 30 | 28 | 32 |
| **d(An,ix)** | 18 | 18 | 26 | 26 | 26 | 30 | 32 | 30 | 34 |
| **xxx.W** | 16 | 16 | 24 | 24 | 24 | 28 | 30 | 28 | 32 |
| **xxx.L** | 20 | 20 | 28 | 28 | 28 | 32 | 34 | 32 | 36 |
| **d(PC)** | 16 | 16 | 24 | 24 | 24 | 28 | 30 | 28 | 32 |
| **d(PC,ix)** | 18 | 18 | 26 | 26 | 26 | 30 | 32 | 30 | 34 |
| **#xxx** | 12 | 12 | 20 | 20 | 20 | 24 | 26 | 24 | 28 |