# Extension Words

More Information Required!

# So Far …

we can translate:

- data register direct
- address register direct
- register indirect
- register indirect with post-increment
- register indirect with pre-decrement
- quick immediate
- implied register

Now we consider the remaining 7 addressing modes, which require extension words.

# What are extension words?

Words that immediately follow the opcode word.

Why are extension words necessary?

The remaining addressing modes require additional data.

Ex. `move.w  x,d3    translates to   3639`

```
    0011 0110 0011 1001
                mm mrrr
```

Which is absolute long mode – but where is the actual address?

# Where are Extension Words Located?

Extension words always follow the instruction word. Why?

In the fetch-execute cycle, after the fetch phase (i.e. fetching the opcode), the PC is pointing to the word immediately after the opcode, i.e. either the 1st extension word or the next instruction.

Why is extension data always *word* or *longword* sized?

The PC must remain word-aligned, remember opcodes are always word sized.

# Number of Bytes in Extension Words

Number of bytes used:

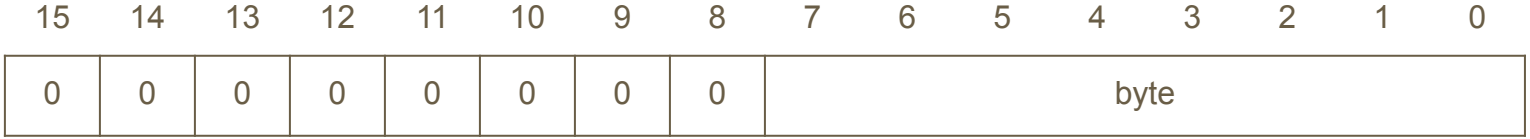
- 0 no extension data
- 2 1 word
- 4 2 words or 1 long
- 6 1 word and 1 long (either order)
- 8 2 longs

See ref. card for the general formats same page as Status Register.

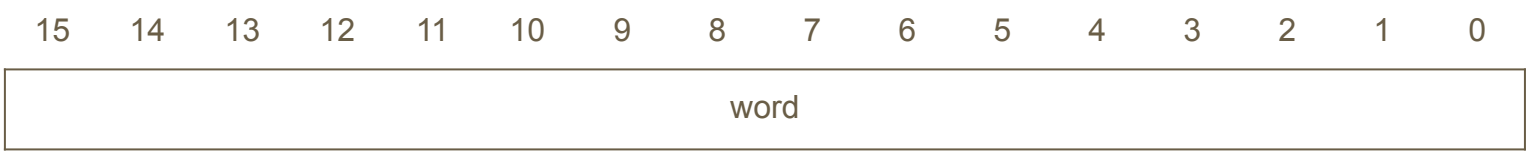
# Immediate Addressing Mode

Simple!  The immediate data is placed in 1-2 extension words.

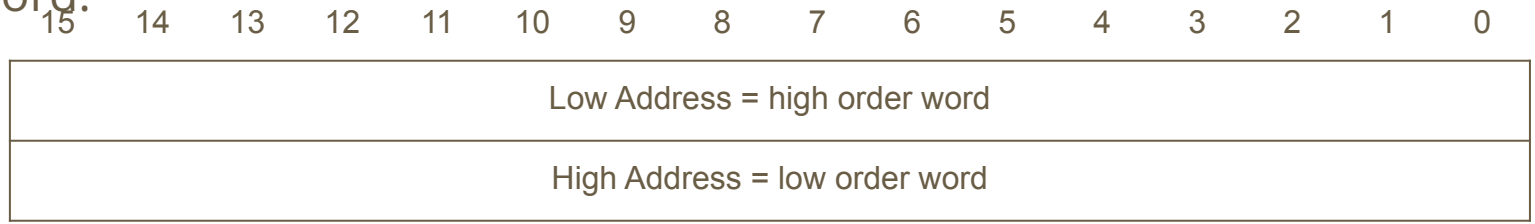
Byte:




Word:




Longword:



# Extension Word Exercises

Exercises: assemble:

1. cmp.b #'A',d0
2. cmp.w #$BEEF,d1
3. addq.l #8,sp

# Absolute Addressing Long/Short

Also simple.

| | | |
|---|---|---|
| absolute long | = | long address stored in 2 words<br>(qualify label with ".l", the default) |
| absolute short | = | word address – sign-extended to long<br>(qualify label with ".w") |

# Register Indirect with Offset

Also simple: displacement is signed 16-bit value, so it is one extension word .

Exercise: assemble:

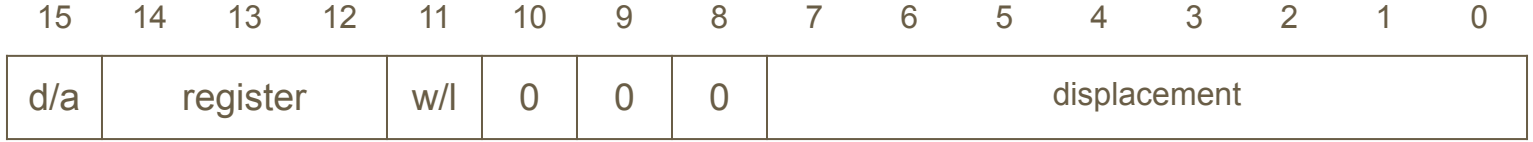
1. move.w -8(a6),a2
2. move.b #42,4(a6)

Note: **source** extension words are stored first.

# Indexed Register Indirect with Offset

Register field in the opcode specifies the base address register to use.

Must also specify the index register, as well as the signed 8-bit displacement. Both of these are encoded into the following 16 bit extension word.




Extension word format (and see ref. card):

d/a: 0 = data register
1 = address register

w/l: 0 = word
1 = long

# Branches

Recall: jmp & jsr load the PC with a new address.

They are translated in the obvious way.

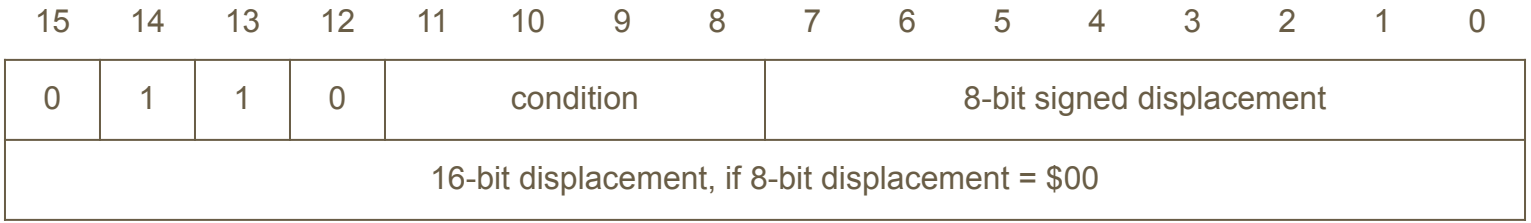
Exercise: Assume my_label is address $1000. assemble:

1. jmp my_label
2. jsr 0(a1,d2)

# Branches

Recall: the other branch instructions (BRA, Bcc, DBcc, BSR) add a signed displacement to the PC.

The Bcc instruction (opcode + extension words) has the form:




See reference card under **Condition Tests**

**Note:** The signed displacements are in 2's complement binary

# Branches

Note:

- there are 16 conditional branches, including BT and BF
- BRA is really just BT
- since BF is useless, that opcode is used for BSR

Branches use a word displacement by default

i.e. if it is not b*cc*.s then it is a word displacement

# Short Branches

By default branches use a word (16-bit) displacement, but a byte (8-bit) displacement can be used instead

- fast
    - does not require an extension word
- limited to +126 forwards or -128 backwards
- good for “tight loops”
- specified with size of “.s”

**Note:** When assembling if the branch does not specify **.S** it is not a short branch

# Branches - Displacement

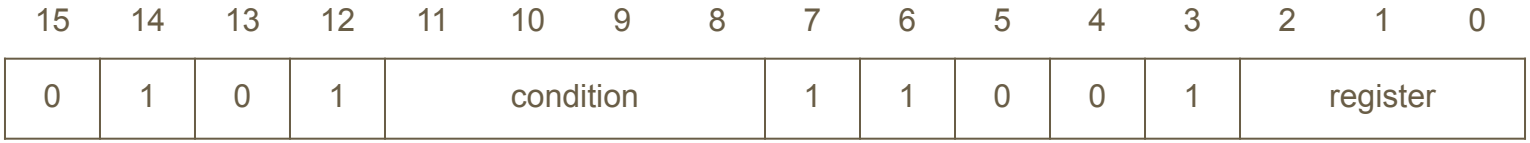
Regardless of size the displacement is always:

```
(destination address) - (instruction address + 2)
```

Question: why “+ 2”?

Answer: because PC is incremented by 2 at the end of the fetch phase.

# Decrement Branches (DB*cc*)




Note:

- DBRA and DBF are the same
- DBT exists, but does nothing!
- DBSR does not exist!
- no short version

  i.e. there is one extension word

# Branching Exercises

Assume that the first three instructions are at location $800 and my_label is $1000. Assume the fourth instruction is at location $1200

Assemble:

1. `bra     my_label`
2. `bsr     my_label`
3. `bmi.s   my_label`
4. `dbeq    d7,my_label`

# Move Memory (movem) Instruction

Recall: the **movem** instruction takes a list of registers

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | s/d | 0 | 0 | 1 | w/l | Mode | | | Register | | |
| d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 |

s/d: 0 - effective address is the destination
1 - effective address is the source

w/l: 0 = word
1 = long

**Note:** For each rN in the extension word it has a value of 1 if the register is included in the instruction; 0 otherwise.

# PC Relative Modes (Advanced)

The 68000 provides two PC relative addressing modes:

1. `d16(PC)` `PC relative with offset`
2. `d8(PC,Xn)` `Indexed PC relative with offset`

At run-time, the current address in PC is used as the base address for the offset.

Aside from using PC instead of a general purpose address register, these 2 modes behave identically to the address register versions.

# PC Relative Modes (Advanced)

**Idea:** a program can be written to **not use absolute addresses**! This allows position independent code to be written, i.e. it will be relocatable.

These modes can be used for accessing operands. They can also be used for doing PC-relative jmp and jsr instructions.

These modes are not supported for bra, dbra, b*cc* and bsr. Why?

They are already in relative mode – displacement is the same regardless of where in memory

# PC Relative Modes (Advanced)

**Alternative:** the loader can update all absolute addresses at load-time to achieve relocation

These modes are not commonly used on the ST

# PC Relative Modes (Advanced) - Example

```
; Once this code/data is loaded into RAM, it can't be relocated
; because the x address is fixed:
    add.w   x,d0
    nop
x:  dc.w0
```

```
; Once this code/data is loaded into RAM, it can be moved around
; and still work! (e.g. OS boot loaders usually relocate themselves)
label:  add.w   x(pc),d0    ; the disp. will actually be
        nop                     ; x-(label+2)
```

```
x:      dc.w0
```

# Reference Card Information

- Up till now we have ignored the two numbers in the columns on the instruction portion of the reference card.
- The value in the # column is the number of bytes used in the instruction
- The ~ column is the number of clock cycles required to execute this instruction
- A memory fetch takes 4 clock cycles on the 68000