Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 4 - Assembler

Session Outline

Microprocessor Fundamentals

Register Transfer Language

Assembly Language

Subroutines and Stacks

Addressing Modes

Microprocessor Fundamentals

Central Processing Unit (CPU)

Controls and performs instructions

Modern CPUs commonly housed on a silicon chip known as a microprocessor

Microprocessor CPU implementations dominate alternatives

CPUs are incredibly complex in their implementation but are usually viewed as having two key components

Arithmetic Logic Unit (ALU) - Performs mathematical and logical operations

Control Unit (CU) - Decodes program instructions and handles logistics for the execution of decoded instructions

Fundamentals of CPU Operation

CPU continuously performs instruction cycle

- Computational instructions are retrieved from memory, decoded to form recognisable operations and executed to impact the current state of a CPU
- Commonly known as the fetch-execute cycle or the fetch-decode-execute cycle

Instruction cycle takes place over several CPU clock cycles

- Recall the significance of clocks in sequential logic circuits

Fetch-decode-execute cycle relies on interaction of several CPU components, including ALU and CU

Fetch-Decode-Execute Components

Arithmetic Logic Unit (ALU) - Performs mathematical and logical operations

Control Unit (CU) - Decodes program instructions and handles logistics for the execution of decoded instructions

Program Counter (PC) - Tracks the memory address of the next instruction to be executed

Instruction Register (IR) - Contains most recent instruction fetched

Memory Address Register (MAR) - Contains address of the region of memory to be read or written, i.e., location of data to be accessed

Memory Data Register (MDR) - Contains data fetched from memory or data ready to be written to memory

Fetch-Decode-Execute Cycle

The specifics of an instruction cycle will vary with microprocessor may vary

Fetch

- Instruction retrieved from memory location held by PC
- Retrieved instruction stored in IR
- PC incremented to point to next instruction in memory

Decode

- Retrieved instruction / operation code / opcode decoded
- Read effective address to establish opcode type

Execute

- CU signals functional CPU components
- May result in changes to data registers, PC, ALU, I/O, etc...

I’m Confused... Where’s the Assembler?

Understanding something about microprocessor operation and their instruction cycles provides an ideal context for us to appreciate programming in an assembly language

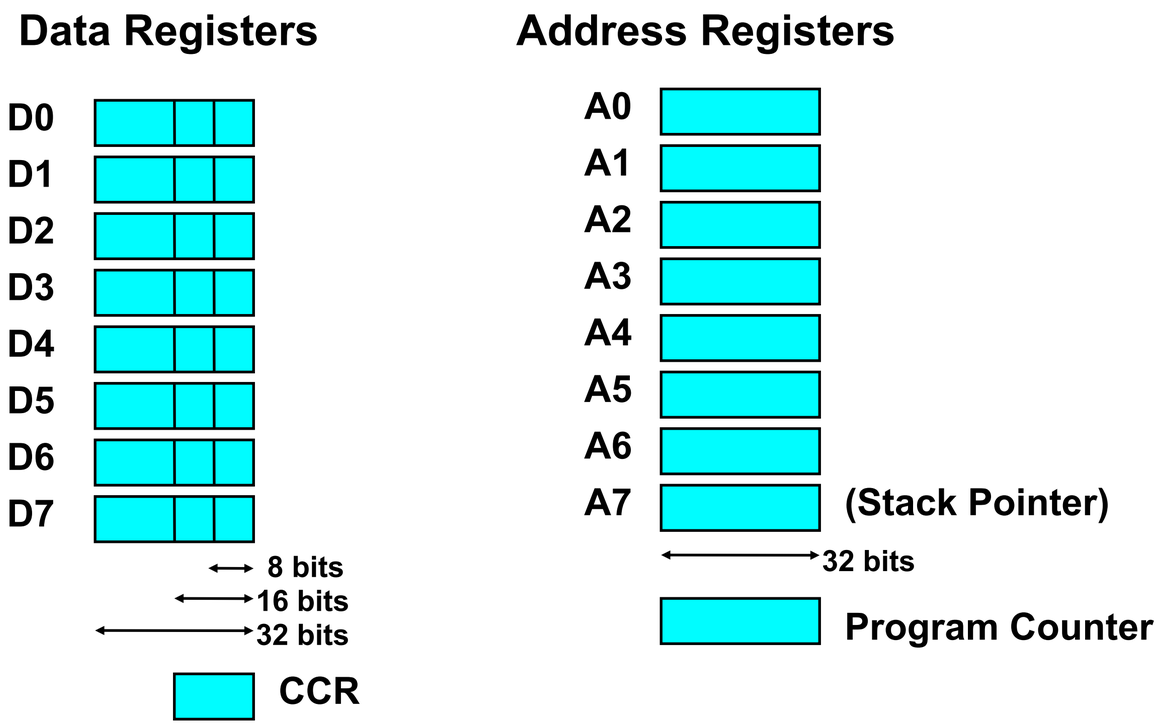
We will revisit many of these issues in subsequent topics

Of course, we can’t just start writing assembler without some idea of what we can and can’t do

How can we think about programming in assembly?

A Worked Example - The 68008 Architecture

To appreciate the general, we will consider a specific example with common characteristics



Programmer’s Model of 68008 CPU

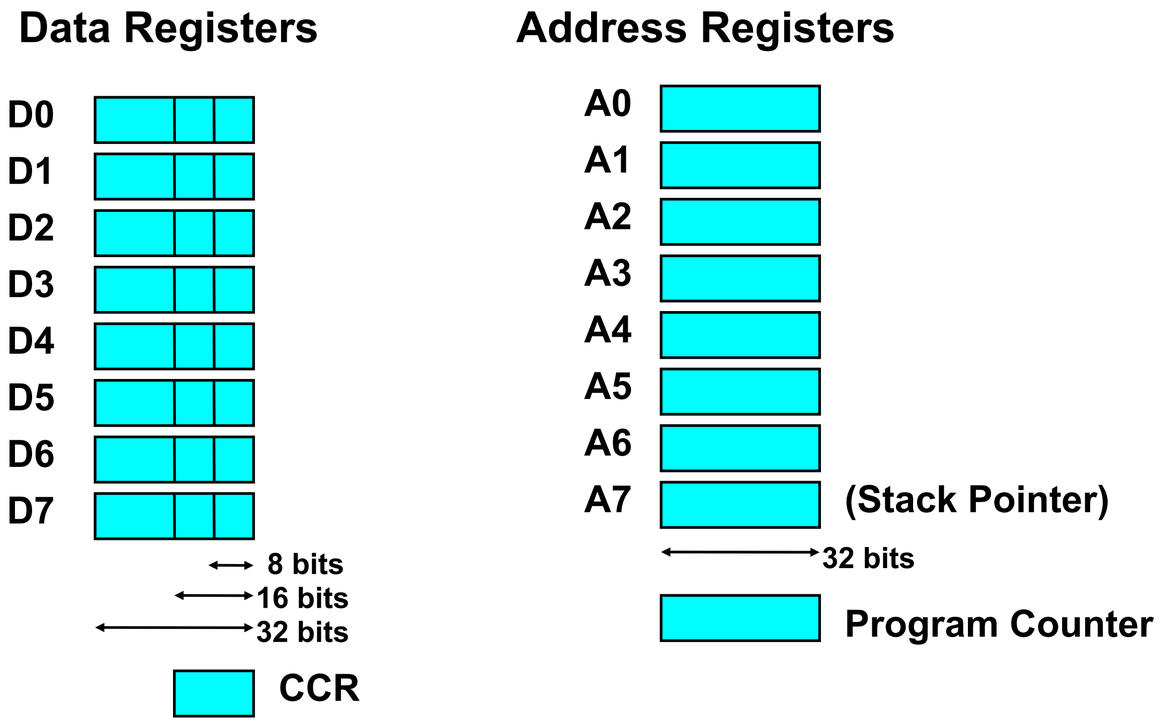
The programmer's model is a commonly used abstraction, invariably used by assembler programmers, of the internal architecture of a processor

68008 processor has identical instruction set to the 68000 processor, but has smaller external buses

- Internal registers are 32-bits wide
- Internal data buses are 16-bits wide
- 68008 has an 8-bit external data bus (16-bit on 68000)
- 68008 has 20-bit external address bus (24-bit for 68000)

A 68008 Programmer's Model

Understanding available registers is the cornerstone of a useful programmer's model



Data Registers

D0 - D7, 32 bit registers, store frequently used values / intermediate results

ON CHIP

Strictly speaking we would only need one register on chip - the advantage of many data registers is that fewer references to external memory are required

Registers can be treated as long, word or byte

Long - 32-bits

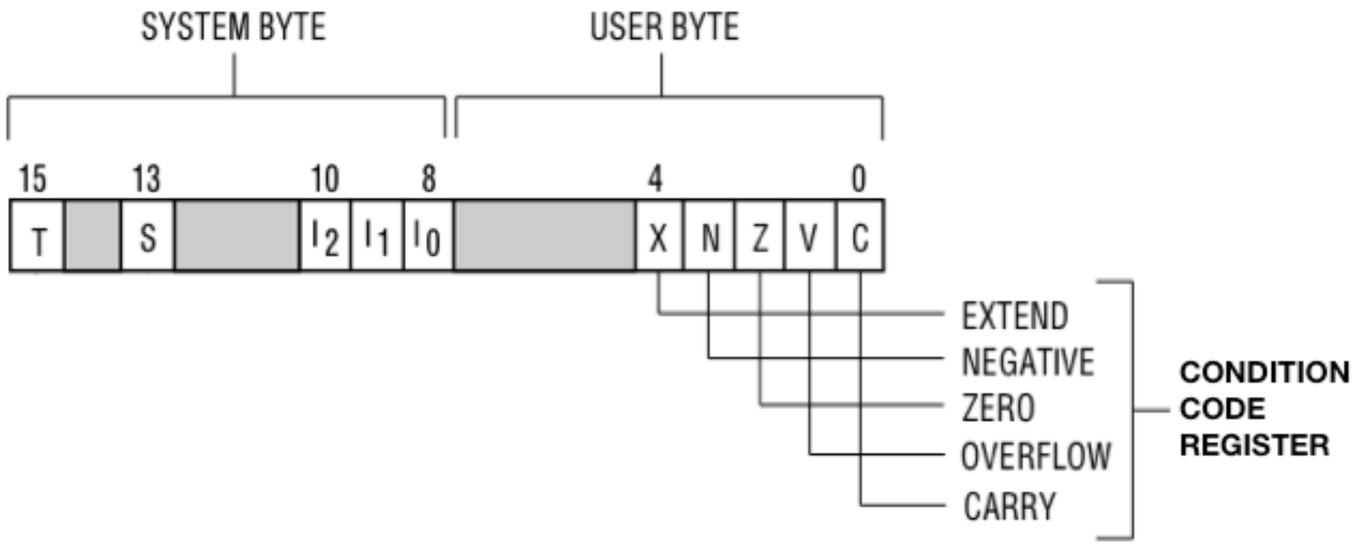
Word - 16-bits (lowest 16 bits)

Byte - 8-bits (lowest 8 bits)

Status Register

16 bit - Consists of two 8-bit registers

Various status bits that are set or reset upon certain conditions arising in the arithmetic and logic unit (ALU)



Address Registers

A0 - A6

Used as POINTER REGISTERS in the calculation of operand addresses

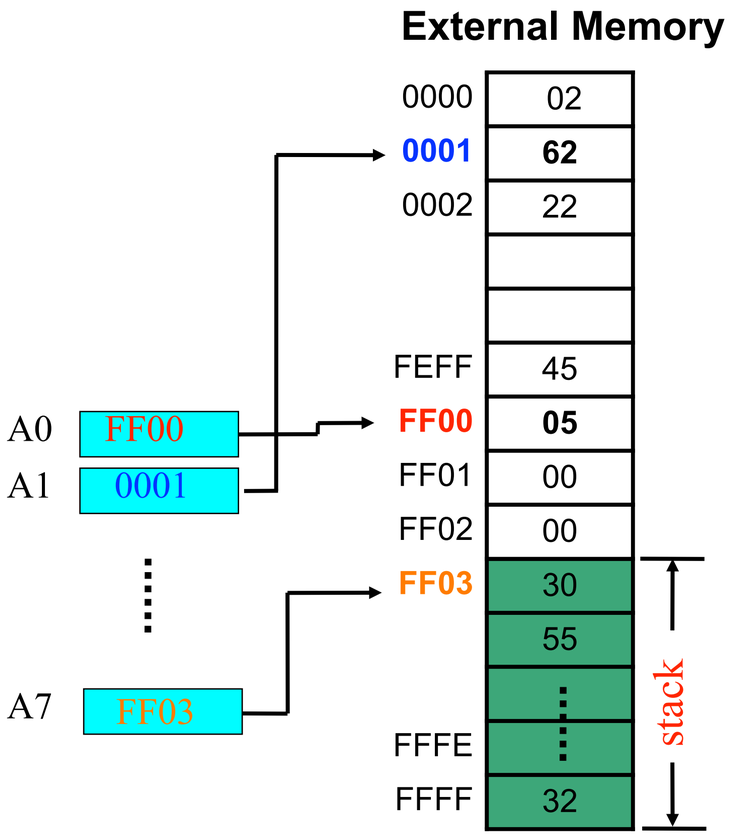
A7 - Additionally used by the processor as a system stack pointer to hold subroutine return addresses etc...

Operations on addresses do not alter status register Condition Code Register (CCR)

ALU has capacity to incur changes in status

A0-A7, Memory and Stack

The diagram below illustrates the relationship between address registers, external memory, i.e., not data registers, and the stack



Stack Pointer

A7 - The stack pointer is used as a pointer into an area of memory called the system stack.

Points to the next free location

The stack is a Last In First Out (LIFO) structure

The stack provides temporary storage of essential processor state, e.g., return addresses and registers, during subroutine calls and interrupts

You will meet this use of the stack again in many different contexts

A0-A6 may also used by programmer as stack pointers for temporary storage of registers, e.g., arithmetic calculations

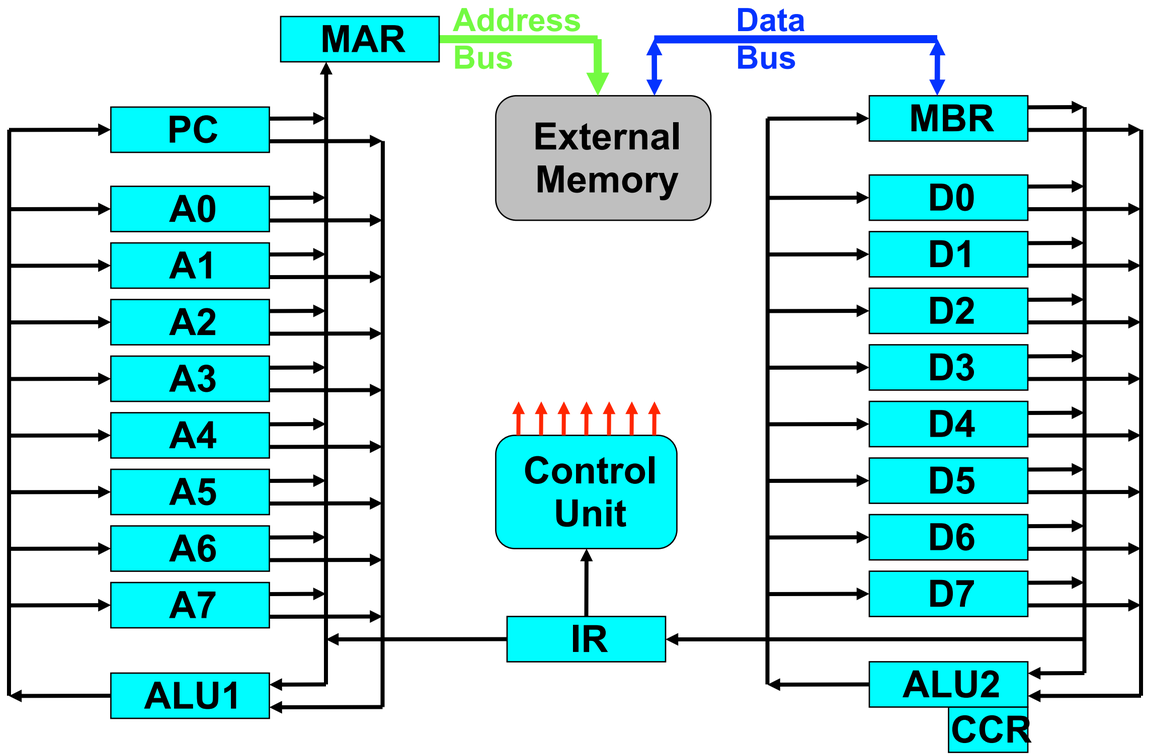
Program Counter (PC)

A 32-bit register that keeps track of the address at which the next instruction will be found

In simple terms, it points to the next instruction in memory

When the current instruction has been read, the PC is incremented to point to the next instruction

68008 Internal Architecture



MAR - Memory Address Register
CCR - Condition Code Register

MBR - Memory Buffer Register
IR - Instruction Register

Register Transfer Language

Register Transfer Language

Used to describe the operations of a microprocessor as it is executing instructions

> For example, [MAR] ← [PC] means transfer contents of Program Counter to the Memory Address Register.

Computer's memory is called Main Store (MS)

> The contents of memory location 12345 is written [MS(12345)]

Try not to confuse register transfer language with assembler instructions

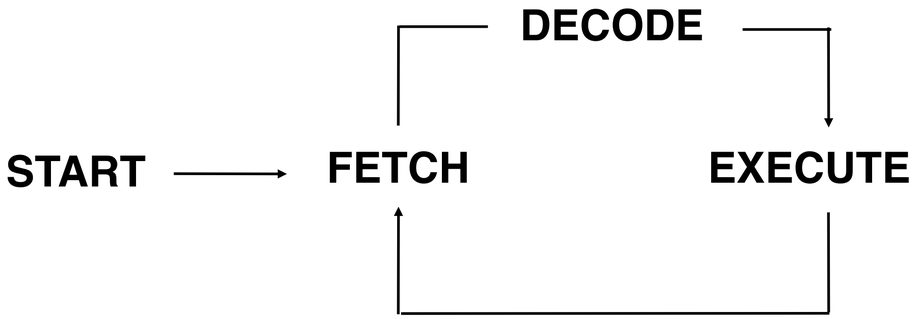
Instruction Cycle

We know quite a bit about the instruction cycle and the components involved in the process

To highlight the power of RTL we can use it represent stages of the instruction cycle

The RTL representation we will see makes no attempt to account for the pipelining of instructions

Pipelining is a common and simple method of speeding up the fetch-execute cycle - you might like to research pipelining in you own time



RTL - Instruction Fetching in Words

1. Contents of Program Counter transferred to MAR address buffers and the Program Counter is incremented
2. MBR loaded from external memory (R/$\overline{W}$ line set to Read)
3. Opcode transferred to Instruction Register from MBR
4. Instruction is decoded

RTL - Instruction Fetching

We can represent the fetch stage of the fetch-execute cycle in RTL

1. [MAR] ← [PC]
2. [PC] ← [PC] +1
3. [MBR] ← [MS([MAR])] ($\overline{R}$/W set to Read)
4. [IR] ← [MBR]
5. CU ← [IR(opcode)]

This part of the cycle is the same for each instruction

RTL - Fetch and Execute

We can represent complex actions, such as adding a constant byte to D0

1. [MAR] ← [PC]
2. [PC] ← [PC] +1
3. [MBR] ← [MS([MAR])] ($\overline{R}/W$ set to Read)
4. [IR] ← [MBR]
5. CU ← [IR(opcode)]

…

RTL - Fetch and Execute

…

6. [MAR] ← [PC]

7. [PC] ← [PC] +1

8. [MBR] ← [MS([MAR])]

9. ALU ← [MBR] + D0

10. [D0] ← ALU

Assembly Language

C Programming

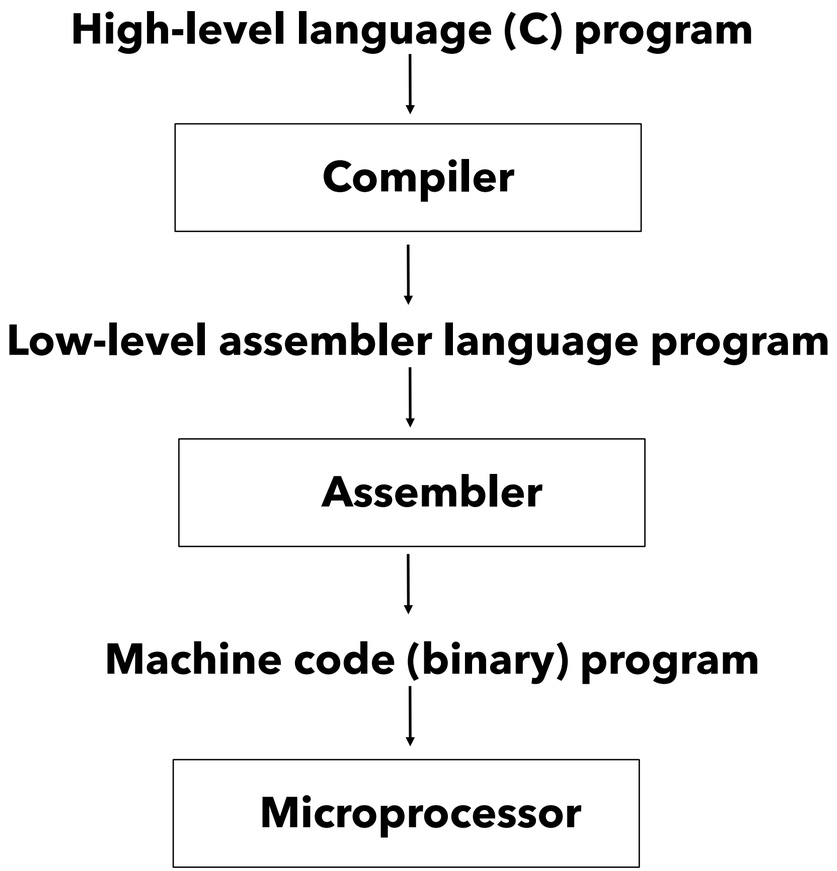
You can work at the hardware-software interface by using the C programming language

C is compiled to assembler - the low level instruction set of the microprocessor

Assembler programs are assembled in order to generate loadable binary

We concentrate on understanding the instruction set

Compiler, Assembler and Microprocessor



Assembly Language

Rather than programming in machine code, i.e., placing numbers in memory locations, we prefer to program at a slightly higher level, i.e., in an assembler language

Assembler language uses easily remembered mnemonics for each instruction, e.g., MOVE D0, D1

Assembler language also allows memory locations and constants to be given symbolic names

A point in a program can be referred to by its name rather than a numeric address

Machine and Assembler Codes

Microprocessors "understand" programs of 0's and 1's

For example:

1010 1001 A9

0000 0101 05

Hex is an aid, but what does the following small program do?

D8 A2 FF 9A 18 A9 05 69 07 8D 11 00

This is why we use assembly languages

Assembler Format

Assembly languages vary but typically have a similar format:

<LABEL>: <OPCODE> <OPERAND(S)> | COMMENT

For example:

```
START:  move.b   #5, D0    | load D0
                           | with 5
```

Example Assembler Program

Understanding assembler programs is about recognising each small operation and how it fits as part of a larger sequence of operations

```
        ORG         $4B0        | this program starts at hex 4B0
        move.b #5, D0           | load D0 with 5
        add.b       #$A, D0     | add 10 to D0
        move.b D0, ANS          | store result in ANS
                                |
ANS:    DS.B        1           | leave 1 byte of memory empty
                                | and give it the name ANS
```

Numbers:

 # : indicates a constant. A number without # prefix is an address

Default number base is DECIMAL

$: means in HEX

% : means in BINARY

Assembler Directives:

ANS: A label, i.e., a symbolic name, that must be terminated by a colon

DS (Define Storage): instructs the assembler to reserve some memory

ORG (Origin): tells the assembler where in memory to start putting the instructions or data

68008 Instruction Set

There are two aspects to the 68008 instruction set:

Instructions - The commands that tell the processor what operations to perform

Addressing Modes - The ways in which the processor can access data or memory locations, i.e., the ways in which addresses may be calculated by the CPU

68008 Instructions

The 68008 instruction set is made up of five categories of instructions:

1. Data Movement
2. Arithmetic
3. Logical
4. Branch
5. System Control

We will look briefly at the first four groups

Form of 68008 Assembler Instructions

Assembler instructions are written in the form:

operation.datatype source, destination

The operation can be on one of three data types:

byte	**.b**	**(8 bits)**
word	**.w**	**(2 bytes)**
long word	**.l**	**(4 bytes)**

The data type and dot may be omitted if the data type is word

1. Data Movement Instructions

```
move.b     D0, D1        | [D1(0:7)] ← [D0(0:7)]
moveb  D0, D1            | the same
move.w     D0, D1        | [D1(0:15)] ← [D0(0:15)]
move    D0, D1           | the same
move.l  $F20, D3         | [D3(24:31)] ← [MS($F20)]
                         | [D3(16:23)] ← [MS($F21)]
                         | [D3( 8:15)] ← [MS($F22)]
                         | [D3( 0:7)] ← [MS($F23)]
                         | (Big-Endian mean this way around)
exg.b    D4, D5          | exchange
swap     D2              | swap lower and upper words
lea    $F20, A3          | load effective address [A3] ←[$F20]
```

2. Arithmetic Instructions

The 68008 does not have hardware floating point support – instructions operate on integers

```
add.l    Di, Dj          | [Dj] ← [Di] + [Dj]
addx.w   Di, Dj          | also add in x bit from CCR
sub.b    Di, Dj          | [Dj] ← [Dj] - [Di]
subx.b Di, Dj            | also subtract x bit from CCR
mulu.w   Di, Dj          | unsigned multiplication:
                         | [Dj(0:31)] ← [Di(0:15)] * [Dj(0:15)]
muls.wDi, Dj             | signed multiplication
divu.b  Di, Dj
divs.l   Di, Dj
```

3. Logical Instructions

These instructions perform bit-wise operations on data and include AND, OR, EOR, NOT

For example, if register D3 contains 1010 0101, then...

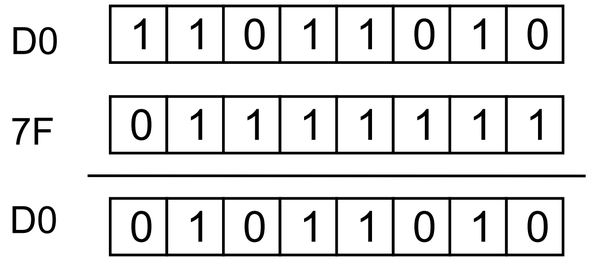
AND.B #%11110000, D3

... will produce the result 1010 0000 in the least significant byte of register D3

Examples of Logical Instructions - Logic

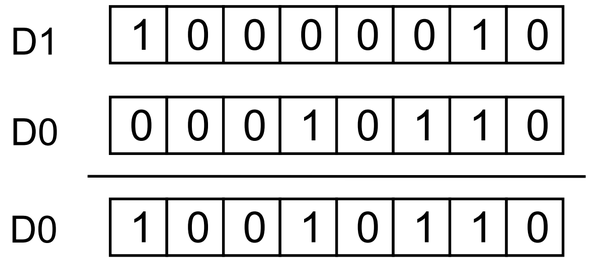
Once you understand a few of these operations the operation of the rest becomes much clearer

Logical AND:
AND.B #$7F, D0



(keep just the lower 7 bits and ignore the MSB)

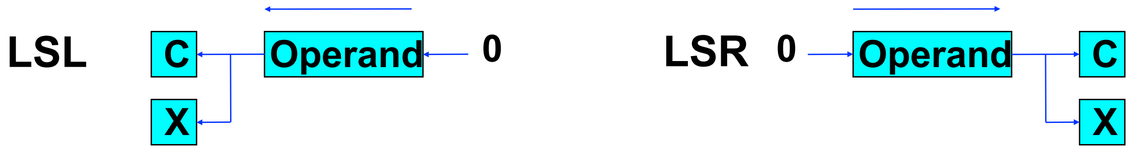
Logical OR:
OR.B D1, D0



Examples of Logical Instructions - Shifts

Shift operations are often fundamental to many computational tasks- remember we achieved multiplication by 2 using a left-shift operation

- **Logical Shift (L - left, R - right)**



(C and X are in Condition Code Register)

- **Arithmetic Shift**

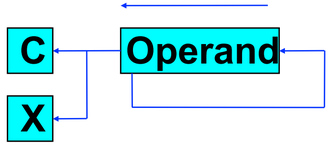
(ASR preserves sign bit)

Examples of Logical Instructions - Rotates

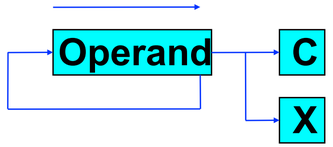
Note the difference between rotating (RO) and rotating through extend bit (ROX) - what does ROX achieve?

- **ROtate**

ROL

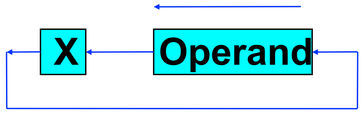


ROR

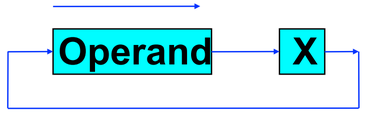


- **ROtate through eXtend bit**

ROXL



ROXR



4. Branch Instructions

Branch instructions cause the processor to branch (jump / GOTO) the labelled address

The instruction can test the state of the CCR bits and branch if a certain condition is met

CCR flags are set by the previous instruction

Form: Bcc <label> (where cc is a condition code)

If a branch is taken, [PC] ← label

Examples of Branch Instructions

15 branch instructions, including:

BRA branch unconditionally
BCC branch on carry clear $(\overline{C})$
BCS branch on carry set (C)
BEQ branch on equal (Z)
BGE branch on greater than or equal $(N.V + \overline{N}.\overline{V})$
…
BPL branch on plus (ie positive) $(\overline{N})$
BVC branch on overflow clear
BVS branch on overflow set (V) $(\overline{V})$

Subroutines and Stacks

Subroutines

Subroutines are useful for frequently used sections of a program

Write (and debug) a subroutine once, and use that program code whenever it is needed

Reduces program size

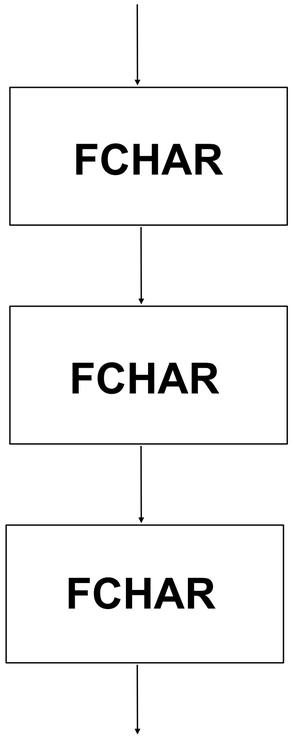
Improves readability

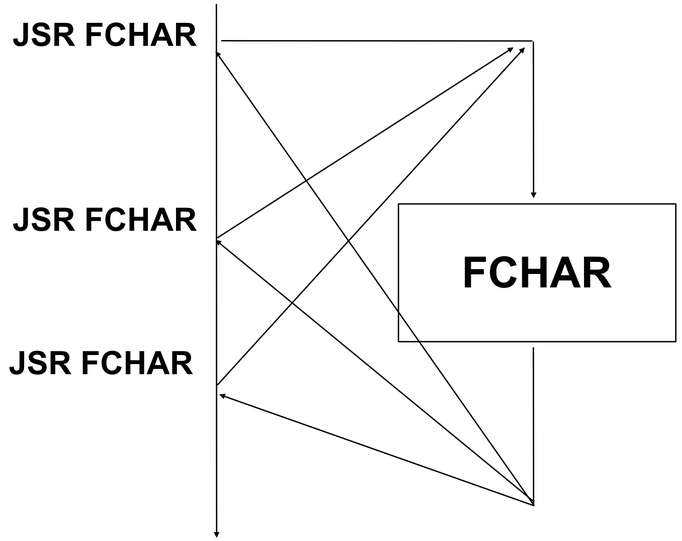
JSR <label> Jump to SubRoutine

RTS Return from SubRoutine

Subroutine Example

Can you see how subroutines work here - what assumption do you make when you return from a subroutine?





Problem - we need to know where to return

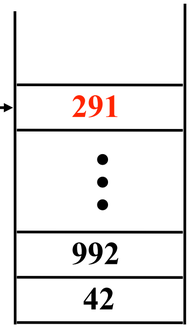
Subroutines and Stacks

A stack can be used to capture the last in first out (LIFO) aspect of the assumption we were making

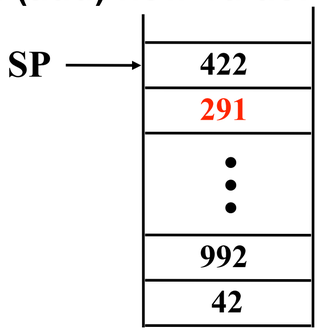
The stack is a last in first out structure. As before, A7 points to the Top Of Stack.

Say, initially:

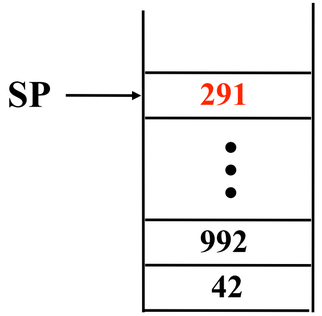
Stack Pointer (A7)



Then push (add) new value:



Then pop (remove) value:



Use of Stack for Subroutine Calls

Subroutine Call (JSR)

Saves (pushes) the contents of the PC on the stack

Puts start address of subroutine in PC

Return from Subroutine (RTS)

Restores (pops) the return address from the stack and puts it in PC

So the stack stores where to return from a subroutine

A subroutine can call another subroutine - we have multiple return addresses on the stack in this case

Addressing Modes

Addressing Modes

How we tell the computer where to find data it needs

Need to organise application data

- Some data never changes, some is variable and some needs to be located within a data structure, e.g., list, table or array

In 68008 systems data can be located in a data register, within the instruction itself or in external memory

Addressing modes have expressive power

- Provide data directly
- Specify exactly where data is
- Specify how to go about finding data

Human “Addressing Modes”

“Here’s £100” (literal value)

“Get the cash from Room 119” (absolute address)

“Go to Room 26 and they’ll tell you where to get the cash” (indirect address)

“Go to Room 42 and get the cash from the fifth room to the right” (relative address)

68008 Addressing Modes

1. Data or Address Register Direct
2. Immediate Addressing
3. Absolute Addressing
4. Address Register Indirect - five variations
5. Relative Addressing

1. Data or Address Register Direct

Perhaps the simplest addressing mode to comprehend if you understand the 68008 architecture

The address of an operand is specified by either a data register or an address register

Example 1:

```
move    D3, D2
```

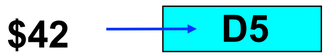
Example 2:

```
move    D3, A2
```

2. Immediate Addressing

The operand forms part of the instruction and remains constant throughout the execution of a program

move.b #$42, D5



The above puts the hex value 42 into register D5

Note the # symbol!

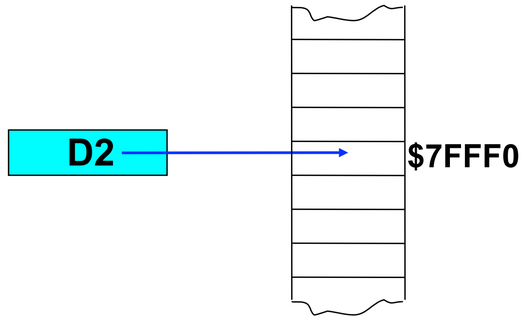
[D5] ← $42

3. Absolute Addressing

The operand specifies the location in memory explicitly, meaning that no further processing required

move.l D2, $7FFF0

[MS(7FFF0)] ← [D2]



Absolute addressing does not allow position independent code because a program will consistently use the same memory address

Note there is no # symbol

4. Address Register Indirect

Address Register Indirect

move (A0), D3

Address Register Indirect with offset

move 7F(A1), D3

Post-Incrementing Address Register Indirect

move.b (A0)+, D3

4. Address Register Indirect

Pre-Decrementing Address Register Indirect

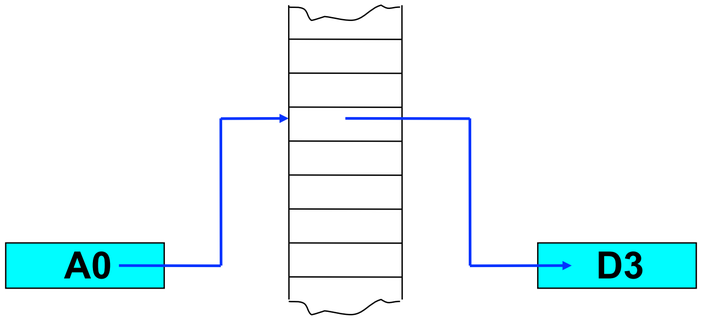
move.b -(A0), D3

Indexed Addressing

move.l 1F(A0, A1), D3

Address Register Indirect

move (A0), D3



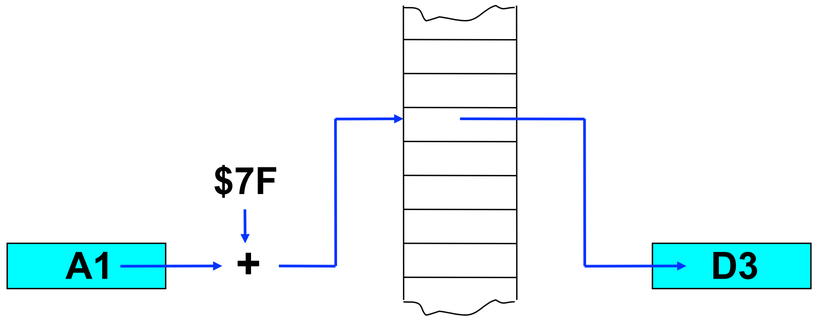
Means take the contents of address register A0 and use this number as the address at which the data will be found. Move this data to register D3

General form: **move (Ai), <ea>**

Remember <ea> is an effective address and can be Dj, (Aj), etc...

Address Register Indirect with Offset

move 7F(A1), D3



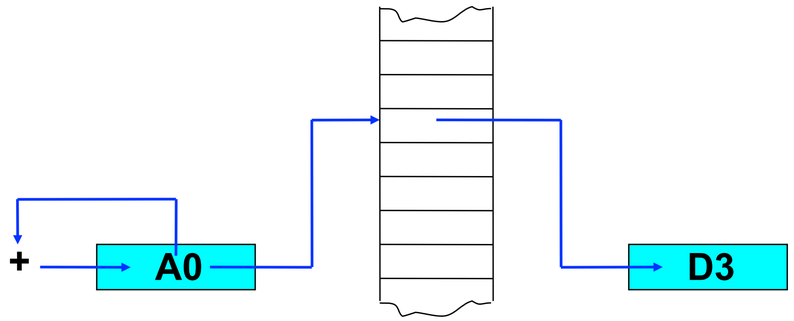
Means take the contents of address register A1, add to this number a (16-bit two's complement) constant and use this result as the address at which the data will be found. Move this data to register D3

General form: **move d16(Ai), <ea>**

This is where d16 is a 16-bit two's complement number

Post-Incrementing Address Register Indirect

move.b (A0)+, D3



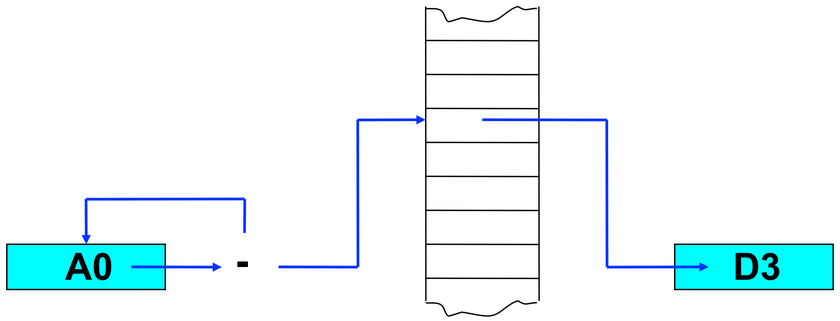
Means take the contents of address register A0, use this number as the address at which the data will be found. Move this data to register D3 and increment A0

General form: **move (Ai)+, <ea>**

Amount by which address register is incremented depends on the type of data being moved, i.e., 1 for bytes, 2 for words, 4 for long words

Pre-Decrementing Address Register Indirect

move.b -(A0), D3



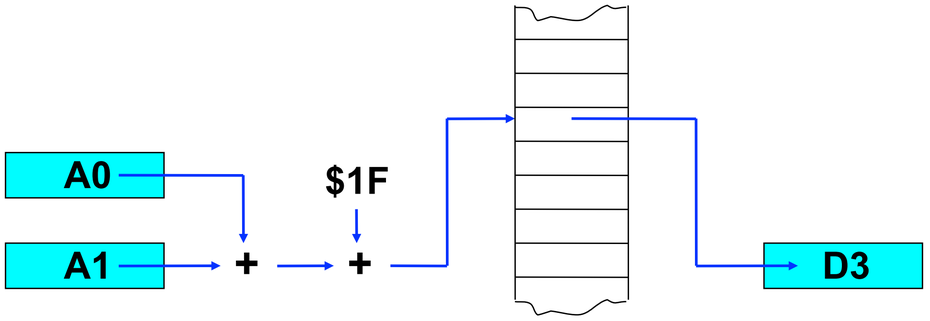
Means decrement the contents of address register A0, and use this number as the address at which the data will be found. Move this data to register D3.

General form: **move -(Ai), <ea>**

Amount by which the address register is decremented depends on the type of data being moved

Indexed Addressing

move.l $1F(A0, A1), D3



Means add the contents of address register A0 to A1, add to this the (8-bit two's complement) hex constant 1F and use this number as the address at which the data will be found. Move this data to register D3.

General form: **move d8(Ai, Xj), <ea>**

Xj can be an address or data register, **d8** is an 8-bit two's complement constant

Indexed Addressing Example

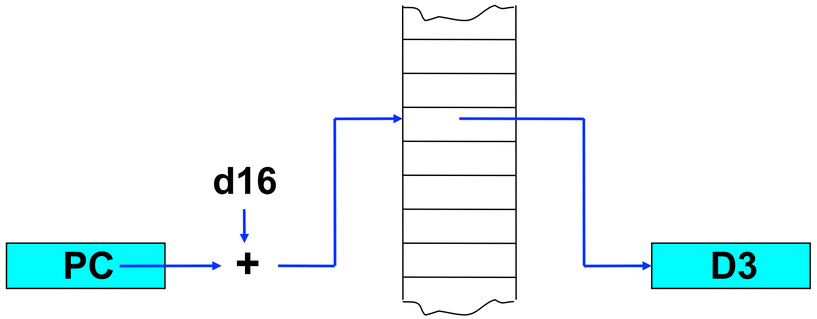
Imagine you're using a 2D-array to track the number of lectures you have, e.g., the number of lectures on Friday Week 2 is located at DIARY+2*7+5

	0 (Sun)	1 (Mon)	2 (Tue)	3 (Wed)	4 (Thu)	5 (Fri)	6 (Sat)
Week 0	*0*	*6*	*7*	*3*	*5*	*8*	*0*
Week 1	*0*	*4*	*5*	*2*	*6*	*6*	*0*
Week 2	*0 (location DIARY+14)*	*6*	*6*	*3*	*5*	*7 (location DIARY+19)*	*0*
Week 3	*0*	*7*	*4*	*1*	*7*	*5*	*0*

```
| pre:  week number in D0
| post: D1 contains number of lectures on Tuesday in week D0
lecsOnTue:   lea      DIARY, A0       | A0 points to start of DIARY
             mulu     #7, D0          | D0 is now a no. of days
             move.b   #2(A0,D0),D1    | D1 is now a no. of lectures
             rts
DIARY:       DS.B     28              | reserve 28 bytes (4 weeks)
                                      | (data initialised elsewhere)
```

5. Relative Addressing

move d16(PC), D3



Code like this contains no absolute addresses, i.e., it uses only addresses relative to the current program counter, and therefore can be placed anywhere in memory

This addressing mode can be used to write “position independent code”

Be careful - small addressing errors can have big consequences

The Accumulator Example

The example below shows how the use of an alternative addressing mode can reduce the work we have to do in developing elegant assembler programs

Original (pseudo code)

```
int N
int W[100]
int i, sum

sum = 0
for i = 0 to N-1
       sum = sum + W[i]
```

Assembly language version

```
        move.l   N, D1          | load D1 with number of items
        movea.l  #W, A2         | load A2 with addr. of 1st item
        clr.l    D0             | D0 used to accumulate sum
loop:   add.w    (A2), D0       | add next number from array
        adda.l   #2, A2         | increment A2: point to next word
        subq.l   #1, D1         | decrement counter
        bgt      loop           | if D1 > 0 then branch to loop
        move.l   D0, sum        | store result in sum
N:      DS.L     1
W:      DS.W     100            | (data initialised elsewhere)
sum:    DS.L     1
```

or, the two boxed lines could be the more optimal…

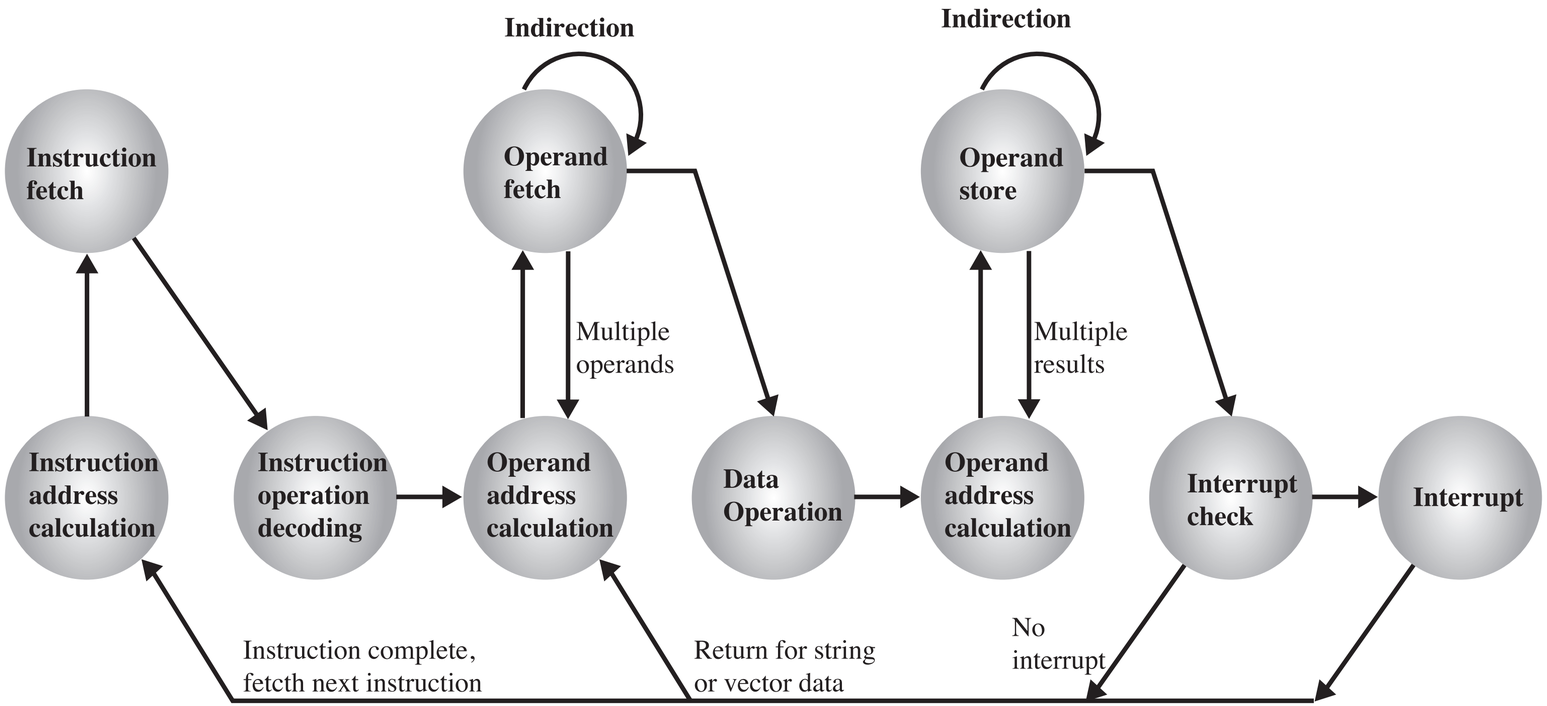
```
loop:   add.w    (A2)+, D0      | also increments A2
```

Fetch-Decode-Execute Revisited

Having studied assembler and addressing modes, we can consider the fetch-decode -execute cycle in it's full form

Elements of this might not fit together perfectly until we've explored memory system and input-output mechanisms

We'll be back!!



Reproduced from: W.Stallings, "Computer Organization and Architecture", 9th Edition, Pearson, 2013