CS132 T3: Assembler

Topics Covered:

- **68008 Architecture**
- **Processor Operating Cycles**
- **Assembly Language**
- **68008 Instruction Set**
- **Subroutines & Stacks**
- **Addressing Modes**

Alternative to lecture slides: Clements, chapters 5 & 6

Programmer's Model of 68008 CPU

- **Programmer's Model is an abstraction used by assembler level 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)

68008 Programmer's Model

Data Registers

D0 - D7

32 bit registers

store frequently used values/intermediate results

ON CHIP

(strictly, only need one register on chip - advantage of many data registers is that fewer references to external memory are needed)

Registers can be treated as

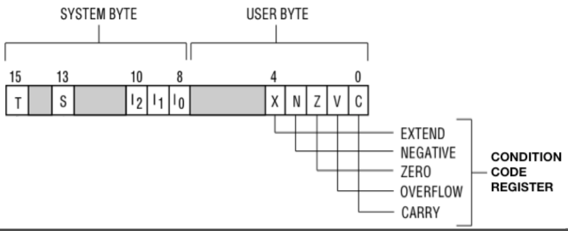
Long -	**32-bits**	
Word -	**16-bits**	**(lowest 16 bits)**
Byte -	**8-bits**	**(lowest 8 bits)**

Status Register

16 bit

Consists of two 8-bit registers

Various status bits that are set or reset upon certain conditions arising in the ALU



Address Registers

A0 - A6

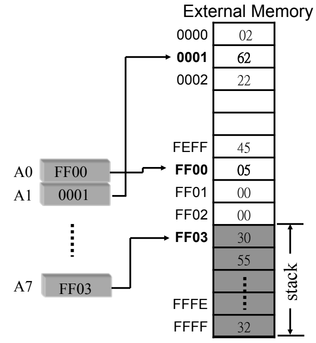
Used as POINTER REGISTERS in the calculation of operand addresses

A7

has additional function: it is used by the processor as a system stack pointer (for holding subroutine return addresses etc)

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

A0-A7, Memory and Stack



Stack Pointer

A7 The stack pointer is used as a pointer into an area of memory called the system stack. It points to the next free location.

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

The stack provides temporary storage of essential processor state (return addresses and registers) during subroutine calls and interrupts.

A0-A6 may also used by programmer as stack pointers for temporary storage of registers eg arithmetic calculations, etc

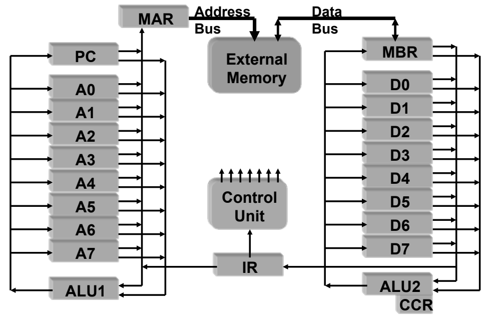
Program Counter

The Program Counter is a 32 bit register.

It keeps track of the address at which the next instruction will be found – ie 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

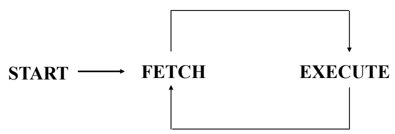


MAR Memory Address Register **MBR** Memory Buffer Register
CCR Condition Code Register **IR** Instruction Register

Processor Operating Cycle

At any time a processor is doing one of two things:

1) **Fetching an Instruction**
2) **Executing the Instruction**



Instruction Fetch Cycle

1. **Contents of PC transferred to MAR address buffers, then PC 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**

Register Transfer Language

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

eg [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 as:

[MS(12345)]

Try not to confuse with assembler instructions.

Reading an Instruction: Fetch

The fetch phase of the cycle is:

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

This part of the cycle is the same for each instruction.

Fetch + Execute

Eg: Add constant byte to data register zero.

1	[MAR] ← [PC]	
	[PC] ← [PC] +1	
2	[MBR] ← [MS([MAR])]	(R/$\overline{W}$ set to Read)
3	[IR] ← [MBR]	
	CU ← [IR(opcode)]	
4	[MAR] ← [PC]	
	[PC] ← [PC] +1	
5	[MBR] ← [MS([MAR])]	
6	ALU ← [MBR] + D0	
7	[D0] ← ALU	

The Lab 68008 Systems

- Small microprocessor system (to be used in term 2):

CPU	68008 (clock speed 8 MHz)
Memory	32 K RAM + 64 K ROM
I/O	Host Computer (RS232) VIA

- The Memory Map:

  1 Mbyte potential memory (or I/O) - most is not present

00000 - 7FFFF	Read Only Memory (64K implemented)
80000 - 87FFF	32K Random Access Memory (Read/Write memory)
C0000 - C0FFF	VIA (Versatile Interface Adaptor)
A0000 - A0FFF	UART (Universal Asynchronous Receiver/Transmitter)
E0000 - EFFFF	Patch Board Enable

Programming

We will program in the high level language C.

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

Assembler programs are assembled to generate loadable binary.

In CS132 T3, we concentrate on the low-level assembler instruction set.

(Will discuss programming in C next term)

Programming Overview

High-Level Language (C) program

↓

Compiler

↓

Low-Level Assembler Language program

↓

Assembler

↓

Machine-Code binary program

↓

µP

Assembly Language

Rather than program in machine code (placing numbers in memory locations) we prefer to program at a (slightly) higher level, at least, in an assembler language.

Assembler language uses EASILY remembered MNEMONICS for each instruction: eg

MOVE D0, D1

Assembler language also allows memory locations and constants to be given symbolic names. Thus a point in a program can be referred to by its name rather than a numeric address.

Machine & Assembler Codes

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

eg

1010 1001	**A9**
0000 0101	**05**

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

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

Which is why we use assembler languages.

Assembler Format

Assembly languages vary but generally have a format similar to:

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

eg

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


Example Assembler Program

```
        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: is a label (symbolic name) 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 this:

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

A) 68008 Instructions

The instruction set is made up of five groups 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 is on one of these data types:

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

(the data type may be omitted if the data type is word, and the dot may also be omitted)

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)]
                   | (this way around because Big-Endian)
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.w   Di, 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

eg 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.

3. Example: logical instructions

E.g. logical and:
AND.B   #$7F, D0

D0	1	1	0	1	1	0	1	0
7F	0	1	1	1	1	1	1	1
D0	0	1	0	1	1	0	1	0

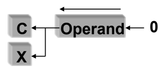
(keep just the lower 7 bits and ignore the MSB)

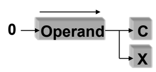
E.g. logical or:
OR.B   D1, D0

D1	1	0	0	0	0	0	1	0
D0	0	0	0	1	0	1	1	0
D0	1	0	0	1	0	1	1	0

3. (cont) Shifts

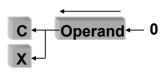
- Logical Shift (L - left, R - right)

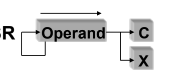
LSL


LSR


(C and X are in Condition Code Register)

- Arithmetic Shift

ASL


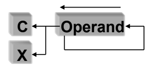
ASR


(ASR preserves sign bit)

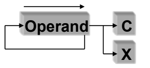
3. (cont) Rotates

- ROtate

ROL

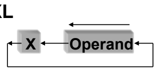


ROR

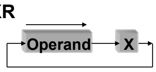


- ROtate through eXtend bit

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

4. (cont) Branch Instructions

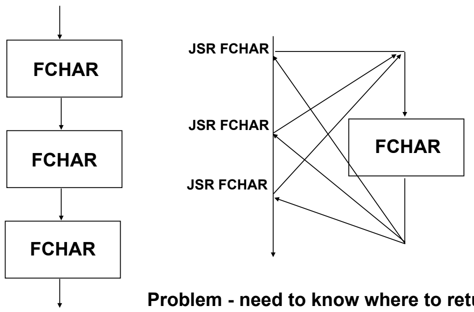
15 branch instructions, including:

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 $(\overline{V})$
BVS branch on overflow set (V)

4. (cont) Subroutines

- Subroutines are useful for frequently used sections of a program
- Write (and debug) a subroutine once, and use that code whenever needed:
  1) Reduces program size
  2) Improves readability
- JSR <label> Jump to SubRoutine
  RTS Return from SubRoutine

Subroutines example

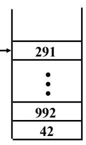


Problem - need to know where to return

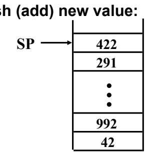
Subroutines and Stacks

The stack is a Last In First Out data structure. A7 points to the Top Of Stack.

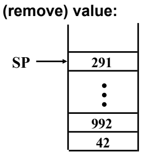
say, initially:
Stack Pointer (A7)



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 the PC

- **So the stack stores where to return from a subroutine**
- **… and a subroutine can call another subroutine (multiple return addresses on the stack in this case)**

B) 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. Some needs to be located within a data structure (list, table, array).

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

Addressing modes allow us to:

- **provide data directly, or**
- **say exactly where it is, or**
- **specify how to go about finding it**

B) (cont) Addressing Modes

Human "addressing modes":

- "Here's £100" **(literal value)**
- "Get the cash from Rootes room 19" **(absolute address)**
- "Go to Rootes room 23 and they'll tell you where to get the cash" **(indirect address)**
- "Go to Rootes 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
3) Absolute
4) Address Register Indirect (five variations)
5) Relative

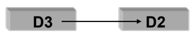
1) Data or Address Register Direct

The address of an operand is specified by either:

a data register, or
an address regiser

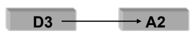
eg

move D3, D2



eg 2

move D3, A2

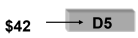


2) Immediate Addressing

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

eg

move.b #$42, D5



which puts the hex value 42 into register D5.
Note the # symbol!

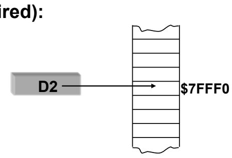
[D5] ← $42

3) Absolute Addressing

The operand specifies the location in memory explicitly (ie no further processing required):

eg

move.l D2, $7FFF0



means:

[MS(7FFF0)] ← [D2]

Absolute addressing does not allow position independent code (ie a program will always use the same address)
Note there is no # symbol!

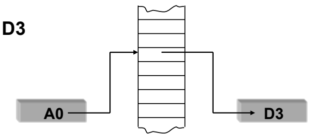
4) Address Register Indirect

This is a family of addressing modes:

- **a) Address Register Indirect**
  - **eg move (A0), D3**
- **b) Address Register Indirect with offset**
  - **eg move 7F(A1), D3**
- **c) Post-Incrementing Address Register Indirect**
  - **eg move.b (A0)+, D3**
- **d) Pre-Decrementing Address Register Indirect**
  - **eg move.b -(A0), D3**
- **e) Indexed Addressing**
  - **eg move.l 1F(A0, A1), D3**

a) Address Register Indirect

eg 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

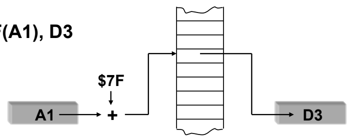
General form:

move (Ai), <ea>

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

b) Address Register Indirect with Offset

eg 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.

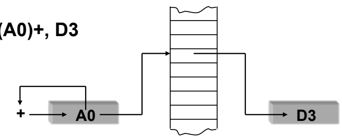
General form:

move d16(Ai), <ea>

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

c) Post-Incrementing Address Register Indirect

eg move.b (A0)+, D3



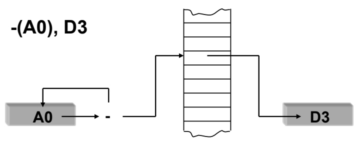
Means take the contents of address register A0, use this number as the address at which the data will be found. Move this data to register D3, and increment A0.

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

The amount by which the address register is incremented depends on the type of data being moved (1 for bytes, 2 for words, 4 for long words)

d) Pre-Decrementing Address Register Indirect

eg move.b -(A0), D3



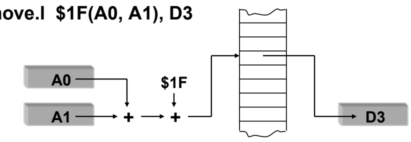
Means decrement the contents of address register A0, and use this number as the address at which the data will be found. Move this data to register D3.

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

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

e) Indexed Addressing

eg 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

Say you have a 2D array, tracking the number of lectures you have. Eg number of lectures on Fri week 2 is located at DIARY+2*7+5.

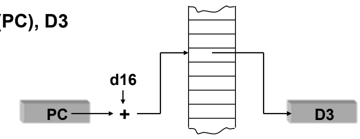
	0 (Sun)	1 (Mon)	2 (Tue)	3 (Wed)	4 (Thu)	5 (Fri)	6 (Sat)
Week 0	*0*	*6*	*7*	*3*	*5*	*8*	*0*
Week 1	*0*	*4*	*5*	*2*	*6*	*6*	*0*
Week 2	*0 (location DIARY+14)*	*6*	*6*	*3*	*5*	*7 (location DIARY+19)*	*0*
Week 3	*0*	*7*	*4*	*1*	*7*	*5*	*0*

```
| 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

(Program Counter Relative Addressing)

eg move d16(PC), D3



Code like this contains no absolute addresses (only addresses relative to the current program counter) so can be placed anywhere in memory. This is "position independent code".

Example code: "accumulator"

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
```


Homework exercises (1&2)

1. The "accumulator" example code (on a previous slide) uses address register indirect mode **(A2)**, and the "more optimal" replacement in the box underneath uses post-incrementing address register indirect mode **(A2)+**. Modify the program to use pre-decrementing address register indirect mode **-(A2)** instead.
2. Explain why each assembly language example here is incorrect.
   (a) MOVE (D1), A0
   (b) ADDA.L A1, D2
   (c) ADDA.W (A1)-, #12
   (d) DC.B $234

Homework exercises (3)

3. What are the contents in D1, A0, memory location 500 and 501 when the end of the program is reached? Also, explain in plain English what this program does.

```
       ORG       300
       MOVEA.L   #N1,  A0
       MOVE.B    #1,   D1
Loop:  MOVE.B    D1,   (A0)+
       LSL.B     #1,   D1
       BCC       Loop
       END
       ORG       500
N1:    DS.B      8
```

When program finishes

[D1] =

[A0] =

[MS(500)] =

[MS(501)] =

Homework exercises (4)

4. Suppose 68008 data registers D0 and D1 are holding one **word** each. Write an assembly program to compare the contents of the two registers, then move the **larger one to D0**, and the **smaller one to D1**. If they are the **same, do not take any action**. (Hint: instruction CMP described in the appendix of Clement's book might be useful. But there is more than one solution and you don't have to use it. You might also need to use some of the conditional branch instructions.)