CPE/EE 421 Microcomputers
Instructor: Dr Aleksandar Milenkovic
Lecture Notes S02

*Material used is in part developed by Dr. D. Raskovic and Dr. E. Jovanov

Instruction Sets
- Software costs growing faster than hardware costs (1970s)
  - Machine language vs. HLL
  - Support for high-level languages
  - Gap between high level languages and computer hardware - semantic gap
- CISC - Complex Instruction Set Architecture
  - Variety of instructions and addressing modes
  - DEC VAX
- HLLCA - High Level Language Computer Architecture

RISC Architectures
- Resolve problems using simpler architecture
  - "The case for the reduced instruction set computers" Patterson & Ditzel [1986]
- Stanford MIPS (Hennessy, 1981)
- Commercial processors: MIPS R2000 (1986), IBM RS6000, SPARC, PowerPC, etc.
- Good design methodology
- Efficient pipelining and compiler-assisted scheduling of pipeline
- Make the Common Case Fast
  - Favor the frequent case

Conventional Computer Architecture

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>( \log_2 N )</td>
</tr>
</tbody>
</table>

Von Neumann Architecture

PE (Processing Element)
Control Unit
ALU
Read/Write Memory
I/O (peripherals)
address
control
data
RISC Methodology

Program execution time:

\[ T = \sum_{i} n_i \cdot CPI_i \cdot T_{cycle} \]

For all instructions in the instruction set

- \( n_i \): instruction count
- \( CPI_i \): cycles per instruction
- \( T_{cycle} \): processor cycle time [s]

80x86 Instruction Mix for SPECint92 Programs [PatHen96]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond. branch</td>
<td>70%</td>
</tr>
<tr>
<td>load</td>
<td>42%</td>
</tr>
<tr>
<td>store</td>
<td>7%</td>
</tr>
<tr>
<td>mov reg-reg</td>
<td>6%</td>
</tr>
<tr>
<td>and</td>
<td>4%</td>
</tr>
<tr>
<td>sub</td>
<td>4%</td>
</tr>
<tr>
<td>xor, not, etc.</td>
<td>1%</td>
</tr>
<tr>
<td>uncond. branch</td>
<td>1%</td>
</tr>
<tr>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>return, jmp indirect</td>
<td>1%</td>
</tr>
<tr>
<td>shift</td>
<td>1%</td>
</tr>
</tbody>
</table>

Instruction Execution

<table>
<thead>
<tr>
<th>Operation</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>ST</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A\times B = C</td>
<td>Mem</td>
<td>PC</td>
<td>1000</td>
<td>1000</td>
<td>…</td>
</tr>
</tbody>
</table>

When \( f = 5\text{MHz} \) (70ns) \n\[ T_{cycle} = 200\text{ns} \]
\[ T_{mem} = 200\text{ns} \]
\[ \text{Total} = 300\text{ns} \]

When \( f = 1000\text{MHz} \) (90ns) \n\[ T_{cycle} = 1\text{ns} \]
\[ T_{mem} = 50\text{ns} \]
\[ \text{Total} = 302\text{ns} \]
System Memory

- W Bytes
- System memory address
- Word address
- Byte address

Example: 2 banks, 32 KW each, 4-byte words

Technological factors

- Instruction execution phases
  - IF - instruction fetch
  - ID - instruction decode
  - OF - operand fetch
  - EX - execute
  - ST - result store

- Technology development
  - 70's $T_{cycle} = 200$ ns, $T_{memory} = 200$ ns
  - 90's $T_{cycle} = 1$ ns, $T_{memory} = 100$ ns

- Changes
  - Pipelining
  - Memory hierarchy

Motorola 68000

- CISC processor
- Sixteen 32-bit registers
  - Eight general purpose data registers
  - Eight general purpose address registers
- User/supervisor space
- 64-pin package
- Clock 8MHz, 12.5 MHz

Programming Model of the 68000*

- *Registers, Addressing Modes, Instruction Set
- NOTE: The 68000 architecture forms a subset of the 68020's architecture (i.e., 68020 is backward compatible)
- NOTE:
  - D[31] – subscripted 2 digits mean bit location
  - D[0] – unsubscripted one digit means register name
**Memory Organization**

- **Long word address** - Address of the high order 16 bits of the longword
- **Big-Endian** - The most significant unit is stored at the lowest address

---

**Special Purpose Registers**

- **Status Register**
  - **PC** - Program Counter: 32 bits, contains the address of the next instruction to be executed
  - **ADD.B D0, D1**
  - **78**
  - **DF**
  - **C, V and X Bits of Status Register**

---

**The Bits of the MC68000 Status Register**

---

**C, V and X Bits of Status Register**

---

Dejan Raskovic
Memory organization - Motorola MCM6706A

- Log₂ 32K = 15 address lines
- 8 data lines
- 3 control lines
- Power supply

Assembly Language Programming

- Machine code/Assembly language
- A form of the native language of a computer
- Development environment
- Assembly program structure
- Assembly directives

Assembly Language Program: Example

<table>
<thead>
<tr>
<th>Label</th>
<th>Field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACK-SP</td>
<td>EQU $08</td>
<td>ASCII code for backspace</td>
</tr>
<tr>
<td>DELETE</td>
<td>EQU $01</td>
<td>ASCII code for delete</td>
</tr>
<tr>
<td>LINE-BUF</td>
<td>DS.B 64</td>
<td>Reserve 64 bytes for line buffer</td>
</tr>
<tr>
<td>ORG $004000</td>
<td></td>
<td>Program origin</td>
</tr>
<tr>
<td>LEA LINE-BUF,A2</td>
<td></td>
<td>Points to line buffer</td>
</tr>
<tr>
<td>NEXT</td>
<td>BSR GET_DATA</td>
<td>Call subroutine to get input</td>
</tr>
<tr>
<td>CMP.B #BACK-SP, D0</td>
<td>Test for backspace</td>
<td></td>
</tr>
<tr>
<td>CMP.B #DELETE, D0</td>
<td>Test for delete</td>
<td></td>
</tr>
<tr>
<td>CMP.B #CAR-RET, D0</td>
<td>Test for carriage return</td>
<td></td>
</tr>
<tr>
<td>CMP.B #EXIT, D0</td>
<td>Test for carriage return then exit</td>
<td></td>
</tr>
<tr>
<td>MOVE.B DD(D0)+</td>
<td>Else store input in memory</td>
<td></td>
</tr>
</tbody>
</table>

Remainder of program

END $001000

$ represents HEX
% represents BIN
# indicates a literal or immediate value
(i.e., not an address)

Assembly Language Program

- 3 fields associated with each line:
  - LABELS
    - Start in the first column of a line
    - Refers to the address of the line it labels
    - Must be 8 or less characters
    - Start with a non-number
  - INSTRUCTION
    - Mnemonic (op code) and 0 or more parameters (operands)
    - Parameters separated by commas
  - COMMENTS
    - Can appear after instruction
    - Can also be used in label field

Assembly Language Program (cont’d)

- Macroassembler
  - A MACRO: a unit of inline code that is given a name by the programmer
  - Example:
    - Instruction to push data on the stack:
      ```
      MOVE.W D0, -(A7)
      ```
    - Define the macro:
      ```
      PUSH D0
      ```
      to replace it
    - Can define a macro for more than one instruction

Assembler Directives

- EQU – The equate directive
- DC – The define a constant directive
- DS – The define a storage directive
- ORG – The origin directive
- END – The end directive

The DC Directive

<table>
<thead>
<tr>
<th>Address</th>
<th>DC.B 10.05</th>
<th>DC.L 001234</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>0A</td>
<td>01</td>
</tr>
<tr>
<td>00102</td>
<td>0A</td>
<td>02</td>
</tr>
<tr>
<td>00104</td>
<td>01</td>
<td>03</td>
</tr>
<tr>
<td>00106</td>
<td>01</td>
<td>04</td>
</tr>
<tr>
<td>00108</td>
<td>02</td>
<td>05</td>
</tr>
<tr>
<td>0010A</td>
<td>0A</td>
<td>06</td>
</tr>
<tr>
<td>0010C</td>
<td>0A</td>
<td>07</td>
</tr>
<tr>
<td>0010E</td>
<td>01</td>
<td>08</td>
</tr>
<tr>
<td>00110</td>
<td>01</td>
<td>09</td>
</tr>
<tr>
<td>00112</td>
<td>02</td>
<td>0A</td>
</tr>
<tr>
<td>00114</td>
<td>04</td>
<td>0B</td>
</tr>
<tr>
<td>00116</td>
<td>00</td>
<td>0C</td>
</tr>
<tr>
<td>00118</td>
<td>02</td>
<td>0D</td>
</tr>
<tr>
<td>0011A</td>
<td>00</td>
<td>0E</td>
</tr>
</tbody>
</table>
## The DC Directive (cont’d)

**Assembler listing**

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00001000</td>
<td>0A42</td>
<td>ORG $001000</td>
</tr>
<tr>
<td>2</td>
<td>00001000</td>
<td>000A</td>
<td>FIRST: DC.B 10,66</td>
</tr>
<tr>
<td>3</td>
<td>00010002</td>
<td>000A</td>
<td>DC.L $0A1234</td>
</tr>
<tr>
<td>4</td>
<td>38203830</td>
<td>3820</td>
<td>DATE: DC.B ‘April 8 1985’</td>
</tr>
<tr>
<td>5</td>
<td>00001012</td>
<td>0000</td>
<td>DC.L 1,2</td>
</tr>
</tbody>
</table>

- DC – define a constant
  - Normally preceded by a label to enable referring
  - Prefixes:
    - Decimal
    - $ - Hexadecimal
    - % - Binary

## DS – The Define Storage Directive

- Reserves the specified amount of storage
- Label DS.<size> <operand>
  - .B, .W, or .L – Number of elements
  - | Label | Size | Description |
    |------|------|-------------|
    | DS.B  | 4    | Reserve 4 bytes of memory |
    | DS.B  | 8    | Reserve 128 bytes of memory |
    | DS.L  | 16   | Reserve 16 longwords (64 bytes) |
    | DS.W  | 1    | Reserve 1 word (2 bytes) |
    | TABLE | 256  | Reserve 256 words |

- Unlike DC does not initialize the values
- Useful to reserve areas of memory that will be used during run time
- Label is set to equal the first address of storage

---

## ORG – The Origin Assembler Directive

- Defines the value of the location counter
- ORG <operand> Absolute value of the origin

<table>
<thead>
<tr>
<th>Label</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG</td>
<td>$001000</td>
<td>Origin for data</td>
</tr>
<tr>
<td>TABLE</td>
<td>DS.W 256</td>
<td>Save 256 words for “TABLE”</td>
</tr>
<tr>
<td>POINTER1</td>
<td>DS.L 1</td>
<td>Save one longword for “POINTER1”</td>
</tr>
<tr>
<td>POINTER2</td>
<td>DS.L 1</td>
<td>Save one longword for “POINTER2”</td>
</tr>
<tr>
<td>VECTOR_1</td>
<td>DS.L 1</td>
<td>Save one longword for “VECTOR_1”</td>
</tr>
<tr>
<td>INIT</td>
<td>DC.W 0,FFFF</td>
<td>Store two constants ($0000, $FFFF)</td>
</tr>
<tr>
<td>SETUP1</td>
<td>EQU 0</td>
<td>Equate “SETUP1” to the value 0</td>
</tr>
<tr>
<td>SETUP2</td>
<td>EQU 055</td>
<td>Equate “SETUP2” to the value 55</td>
</tr>
<tr>
<td>ACIAC</td>
<td>EQU 50000</td>
<td>Equate “ACIAC” to the value 50000</td>
</tr>
<tr>
<td>RDRF</td>
<td>EQU 0</td>
<td>RDRF = Receiver Data Register Full</td>
</tr>
<tr>
<td>PIA</td>
<td>EQU ACIAC+4</td>
<td>Equate “PIA” to the value $0004</td>
</tr>
</tbody>
</table>

## Assembler Directives: Example

<table>
<thead>
<tr>
<th>Label</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG</td>
<td>$001000</td>
<td>Origin for program</td>
</tr>
<tr>
<td>ENTRY</td>
<td>LEA ACIAC.A0</td>
<td>point to the ACIA</td>
</tr>
<tr>
<td>MOVE.B #SETUP2,(A0)</td>
<td>Write initialization constant into ACIA</td>
<td></td>
</tr>
<tr>
<td>GET_DATA</td>
<td>BTST.B #RDRF(A0)</td>
<td>Any data received?</td>
</tr>
<tr>
<td>BNE GET_DATA</td>
<td>Repeat until data ready</td>
<td></td>
</tr>
<tr>
<td>MOVE.B (IA),(A0)</td>
<td>Read data from ACIA</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>$001000</td>
<td></td>
</tr>
</tbody>
</table>
Addressing Modes

Addressing modes are concerned with how the CPU accesses the operands used by its instructions.

Assembler Directives: Example (cont'd)

```
1 00000000 00000000 00000000 00000000
2 00000000 00000000 00000000 00000000
3 00000000 00000000 00000000 00000000
4 00000000 00000000 00000000 00000000
5 00000000 00000000 00000000 00000000
6 00000000 00000000 00000000 00000000
7 00000000 00000000 00000000 00000000
8 00000000 00000000 00000000 00000000
9 00000000 00000000 00000000 00000000
10 00000000 00000000 00000000 00000000
11 00000000 00000000 00000000 00000000
12 00000000 00000000 00000000 00000000
13 00000000 00000000 00000000 00000000
14 00000000 00000000 00000000 00000000
15 00000000 00000000 00000000 00000000
16 00000000 00000000 00000000 00000000
17 00000000 00000000 00000000 00000000
18 00000000 00000000 00000000 00000000
19 00000000 00000000 00000000 00000000
```

Register Transfer Language (RTL)

- Unambiguous notation to describe information manipulation
- Registers are denoted by their names (e.g., D1-D7, A0-A7)
- Square brackets mean “the contents of”
- Base number noted by a prefix (% - binary, $ - hex)
- Backward arrow indicates a transfer of information (←)

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Location (i.e., address) in the main store</td>
</tr>
<tr>
<td>A i</td>
<td>Address register (i = 0 to 7)</td>
</tr>
<tr>
<td>D i</td>
<td>Data register (i = 0 to 7)</td>
</tr>
<tr>
<td>X i</td>
<td>General register (i = 0 to 7)</td>
</tr>
<tr>
<td>[M]</td>
<td>The contents of memory location M</td>
</tr>
<tr>
<td>[X]</td>
<td>The contents of register X</td>
</tr>
<tr>
<td>[D i (0:7)]</td>
<td>Bits 0 to 7 inclusive of register D_i</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Enclose a parameter required by an expression</td>
</tr>
<tr>
<td>[M(=a)]</td>
<td>The contents of a memory location specified by a</td>
</tr>
<tr>
<td>d8</td>
<td>An 8-bit signed offset (-128 to 127)</td>
</tr>
<tr>
<td>d16</td>
<td>A 16-bit signed offset (-32K to 32K - 1)</td>
</tr>
<tr>
<td>d32</td>
<td>A 32-bit signed offset (-2G to 2G - 1)</td>
</tr>
</tbody>
</table>

ADD <source>,<destination>
MOVE <source>,<destination>
**Register Direct Addressing**

*Register direct addressing* is the simplest addressing mode in which the source or destination of an operand is a data register or an address register. The contents of the specified source register provide the source operand. Similarly, if a register is a destination operand, it is loaded with the value specified by the instruction. The following examples all use register direct addressing for source and destination operands.

MOVE.B D0,D3
\[ D3[0:7] \leftarrow D0[0:7] \]

SUB.L A0,D3
Subtract the source operand in register A0 from register D3

CMP.W D2,D0
Compare the source operand in register D2 with register D0

ADD D3,D4
Add the source operand in register D3 to register D4

**MOVE.B D0,D1**

The destination operand is data register D1

The effect of this instruction is **TO COPY** the contents of data register D0 in to data register D1
Register Direct Addressing

- Register direct addressing uses short instructions because it takes only three bits to specify one of eight data registers.
- Register direct addressing is fast because the external memory does not have to be accessed.
- Programmers use register direct addressing to hold variables that are frequently accessed (i.e., scratchpad storage).

Immediate Addressing

- In immediate addressing the actual operand forms part of the instruction. An immediate operand is also called a literal operand. Immediate addressing can be used only to specify a source operand.
- Immediate addressing is indicated by a # symbol in front of the source operand.
- For example, MOVE.B #24,D0 uses the immediate source operand 24.

Immediate Addressing

The instruction MOVE.B #4,D0 uses a literal source operand and a register direct destination operand.

The literal source operand, 4, is part of the instruction.
The destination operand is a data register.

MOVE.B #4,D0

Immediate Addressing Example

Typical application is in setting up control loops:

\[ \text{for}(i=0; i<128; i++) \]
\[ A(i) = \text{xFF}; \]

68000 assembly language implementation:

- `MOVE.L #0x1000,A0` Load A0 with the address of the array
- `MOVE.B #128, D0` D0 is the element counter
- `SLQ #1, D0` Increment element counter
- `BNE LOOP` Repeat until all the elements are set

Direct (or Absolute) Addressing

In direct or absolute addressing, the instruction provides the address of the operand in memory.

Direct addressing requires two memory accesses. The first is to access the instruction and the second is to access the actual operand.

For example, CLR.B 1234 clears the contents of memory location 1234.
Direct (or Absolute) Addressing

This instruction has a direct source operand

The source operand is in memory

The destination operand uses data register direct addressing

The effect of MOVE.B 20,D0 is to read the contents of memory location 20 and copy them to D0

Problem: [D0] ← [M(1001)] + [D0]  
A = Y + A

Instruction:

Effective Address: 0000 1001
Summary of Fundamental Addressing Modes

- Consider the high-level language example: \( Z = Y + 4 \)
- The following fragment of code implements this construct:

```
ORG $400   Start of code
MOVE.B Y,D0
ADD #4,D0
MOVE.B D0,Z
ORG $600   Start of data area
Y DC.B 27   Store the constant 27 in memory
Z DS.B 1    Reserve a byte for Z
```

The Assembled Program

```
1 00000400 ORG $400
2 00000400 10390000600 MOV.B Y,D0
3 00000406 06000018 ADD.B #24,D0
4 0000040A 13C000000601 MOV.B D0,Z
5 00000410 4E722700 STOP #$2700
6
7 00000600 ORG $600
8 00000600 1B Y: DC.B 27
9 00000601 00000001 Z: DS.B 1
10 00000400 END $400
```

Memory Map of the Program

- \( Y \) is a variable accessed via the direct address 000600
- \( Z \) is a variable accessed via the direct address 000601
- This is a literal operand stored as part of the instruction
Moving data from a 32-bit register to memory using the MOVEP instruction

NOTE: The instruction takes 24 clock cycles to execute.

Bytes from the register are stored in every other memory byte.

Summary

- Register direct addressing is used for variables that can be held in registers
- Literal (immediate) addressing is used for constants that do not change
- Direct (absolute) addressing is used for variables that reside in memory
- The only difference between register direct addressing and direct addressing is that the former uses registers to store operands and the latter uses memory