Outline

- Recap
- CISC vs. RISC
- Motorola MC68000
Machine Representation

- Computers manipulate bits
  - Bits must represent “things”
    - Instructions, numbers, characters, etc.
    - Must tell machine what the bits mean
- Given N bits
  - $2^N$ different things can be represented
Positional Notation for Numbers

• Base (radix) B number ➔ B symbols per digit
  ▫ Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  ▫ Base 2 (binary) 0, 1

• Number representation
  ▫ $d_{31}d_{30}...d_2d_1d_0$ is 32 digit number
  ▫ Value = $d_{31} \times B^{31} + d_{30} \times B^{30} + ... + d_1 \times B^1 + d_0 \times B^0$
Two’s Complement

- Unbalanced representation
  - Leading zeros for positive
    - $2^{N-1}$ non-negatives
  - Leading ones for negative number
    - $2^{N-1}$ negative number
  - One zero representation
- First bit is sign-bit (must indicate width)
  - Value = $d_{31} \times -2^{31} + d_{30} \times 2^{30} + \ldots + d_{1} \times 2^{1} + d_{0} \times 2^{0}$

Negative value for sign bit
Two’s Complement Notes

• Negation shortcut
  ▪ Invert bits and add 1
    • $\bar{x} + 1 = -x$

• Sign extension
  ▪ Replicate sign bit (msb) of smaller container to fill new bits in larger container

• Overflow
  ▪ Not enough bits to represent a number
  ▪ Indicated by V flag in condition code
Machine Performance

- What is machine performance?
- How can performance be measured?

- Response time
  - How long to complete a task

- Throughput
  - Total work completed per unit time
  - E.g. task/per hour

- Program execution time is best measure of performance
Relative Performance

- \( \text{Performance}_x = \frac{1}{\text{Execution time}_x} \)
- \( \text{Speedup} = n = \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution time}_y}{\text{Execution time}_x} \)
Performance Summary

- CPU Time = #Instructions × CPI × clock cycle time

\[
\text{Execution time } = T = IC \times CPI \times \tau
\]

- \(T := \) CPU time
- \(IC := \) instruction count
  - Number of instructions in a program
- \(CPI := \) clock cycles/instruction
  - Average clock cycles per instruction – depends on instruction mix
- \(\tau := \) duration of clock period
  - Specified in time or in rate (Hz)
RISC vs. CISC Designs

- **CISC: Complex Instruction Set Computer**
  - Many complex instructions and addressing modes
  - Some instructions take many steps to execute
  - Not always easy to find best instruction for a task
- **RISC: Reduced Instruction Set Computer**
  - Few, simple instructions, addressing modes
  - Usually one word per instruction
  - May take several instructions to accomplish what CISC can do in one
  - Complex address calculations may take several instructions
  - Usually has load-store, general register ISA
Memory Bottleneck

- Memory no longer expensive
- Design for speed

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1981 (8086)</th>
<th>2004 (Pentium P4)</th>
<th>Improvement factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>4.7 MHz</td>
<td>4 GHz</td>
<td>~1000</td>
</tr>
<tr>
<td>Clock Period</td>
<td>212 ns</td>
<td>200 ps</td>
<td>~1000</td>
</tr>
<tr>
<td>Memory Cycle Time</td>
<td>100 ns</td>
<td>70 ns</td>
<td>1.4</td>
</tr>
<tr>
<td>Clocks per Memory Cycle</td>
<td>.47</td>
<td>280</td>
<td>~ -500</td>
</tr>
</tbody>
</table>
Dealing with Memory Bottleneck

- Employ one or more levels of cache memory.
  - Prefetch instructions and data into I-cache and D-cache.
    - Out of order execution.
    - Speculative execution.
- One word per instruction (RISC)
- Simple addressing modes (RISC)
- Load-Store architecture (RISC)
- Lots of general purpose registers (RISC)
RISC Design Characteristics

- Simple instructions can be done in few clocks
  - Simplicity may even allow a shorter clock period
- A pipelined design can allow an instruction to complete in every clock period
- Fixed length instructions simplify fetch & decode
- The rules may allow starting next instruction without necessary results of the previous
  - Unconditionally executing the instruction after a branch
  - Starting next instruction before register load is complete
More on RISC

- Prefetch instructions
  - Get instruction/data/location before needed in pipeline
- Pipelining
  - Beginning execution of an instruction before the previous instruction(s) have completed. (Chapter 5.)
- Superscalar operation
  - Issuing more than one instruction simultaneously.
  - Instruction-level parallelism (Chapter 5.)
- Out-of-order execution
- Delayed loads, stores, and branches
  - Operands may not be available when an instruction attempts to access them.
- Register Windows
  - Ability to switch to a different set of CPU registers with a single command. Alleviates procedure call/return overhead. Discussed with SPARC (Chapter 3)
Developing and ISA (Table 3.1)

- **Memories: structure of data storage in the computer**
  - Processor-state registers
  - Main memory organization

- **Formats and interpretation: meaning of register fields**
  - Data types
  - Instruction format
  - Instruction address interpretation

- **Instruction interpretation: things done for all instructions**
  - Fetch-execute cycle
  - Exception handing

- **Instruction execution: behavior of individual instructions**
  - Grouping of instructions into classes
  - Actions performed by individual instructions
The Motorola MC6800

- Introduced in 1979
  - Computers
    - Apple Lisa 2, Apple Macintosh 128, Atari 520STfm and 1040STfm, Commodore Amiga 500 and 1000
    - Still in use today (now Freescale Semiconductor)
- Very early 32-bit microprocessor
  - Most operations on 32-bit internal data
  - Some operations may use different number of bits
  - External datapaths may not all be 32 bits wide
    - 24-bit address bus for MC68000
- Complex instruction set computer
  - Large instruction set
  - 14 addressing modes
New Concepts from MC68000

- **Effective address (EA)**
  - Addressing modes
- **Subroutines**
  - E.g. function calls
- **Starting a program**
  - Assemble, link, load, and run times
- **Exceptions**
  - Interruption of normal sequential instruction execution
- **Memory-mapped I/O**
  - Part of CPU memory reserved for I/O
MC68000 Programmer’s Model

Copyright © 2004 Pearson Prentice Hall, Inc.
Features of Processor State

- Distinction between 32-bit data registers and 32-bit address registers
- 16 bit instruction register
  - Variable length instructions handled 16 bits at a time
- Stack pointer registers
  - User stack pointer is one of the address registers
  - System stack pointer is a separate single register
    - Why a separate system stack?
- Condition code register: System & User bytes
  - Arithmetic status (N, Z, V, C, X) is in user status byte
  - System status has Supervisor & Trace mode flags and the Interrupt Mask
RTN Processor State

- **Registers**
  - \(D[0..7]<31..0>:\)
  - \(A[0..7]<31..0>:\)
  - \(PC<23..0>:\)
  - \(IR<15..0>:\)

- **Stack pointers**
  - \(SP := A[7]:\)
  - \(A7´<31..0>:\)
  - \(SSP := A7´:\)

- **Status<15..0>:**

- **User byte (condition codes)**
  - \(C := \text{Status}<0>:\)
  - \(V := \text{Status}<1>:\)
  - \(Z := \text{Status}<2>:\)
  - \(N := \text{Status}<3>:\)
  - \(X := \text{Status}<4>:\)

- **System byte**
  - \(INT<2..0> := \text{Status}<10..8>:\)
  - \(S := \text{Status}<13>:\)
  - \(T := \text{Status}<15>:\)
Main Memory

• Main memory:
  ▫ \( \text{Mb}[0..2^{24}-1]<7..0> : \) memory as bytes
  ▫ \( \text{Mw}[ad]<15..0> := \text{Mb}[ad]\#\text{Mb}[ad+1] : \) memory as words
  ▫ \( \text{Ml}[ad]<31..0> := \text{Mw}[ad]\#\text{Mw}[ad+2] : \) memory as longwords

• Word and longword forms are big-endian
  ▫ Lowest numbered byte contains most significant bit of word

• Hard word alignment constraints
  ▫ Not described in the RTN
  ▫ Word addresses must in end in on binary 0
  ▫ Longword addresses end in two binary 0
    • What are differences between soft alignment?
Operand Types

• One instruction may operate on several types (CISC design)
  • MOVE.B bytes
  • MOVE.W word
  • MOVE.L longwords
  ▫ Default is word operands
  ▫ Operand length encoded in instruction

• Bits to encode operand type vary with instruction
  ▫ Assumption for RTN description
    • \( d := \text{datalen}(IR) \):
  ▫ Function returns 1, 2, 4 for operand length
Instruction Formats

- Instructions accessed in 16-bit words
- Variable number of words in an instruction

(a) A 1-word move instruction

(b) A 2-word instruction

(c) A 3-word instruction

(d) Instruction with indexed address
Addressing Modes

• General address of operand specified by 6-bit field
  ▫ Access paths to memory and registers
  ▫ See Table 3.2 for details
• 6-bit effective address
  ▫ Mode field provides access paths to operands
• Not all operands/results can be specified by general address - some must be in registers
• Exception
  ▫ Destination of MOVE instruction has mode and reg fields reversed
### Table 3.2: MC68000 Addressing Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Reg</th>
<th>Assembler Syntax</th>
<th>Extra Words</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data register direct</td>
<td>0</td>
<td>0-7</td>
<td>Dn</td>
<td>0</td>
<td>Dn</td>
</tr>
<tr>
<td>Address register direct</td>
<td>1</td>
<td>0-7</td>
<td>An</td>
<td>0</td>
<td>An</td>
</tr>
<tr>
<td>Address register indirect</td>
<td>2</td>
<td>0-7</td>
<td>An)</td>
<td>0</td>
<td>M[An]</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>3</td>
<td>0-7</td>
<td>(An) +</td>
<td>0</td>
<td>M[An]; An ← An + d</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>4</td>
<td>0-7</td>
<td>-(An)</td>
<td>0</td>
<td>An ← An - d; M[An]</td>
</tr>
<tr>
<td>Based</td>
<td>5</td>
<td>0-7</td>
<td>disp16(An)</td>
<td>1</td>
<td>M[An+disp16]</td>
</tr>
<tr>
<td>Based indexed short</td>
<td>6</td>
<td>0-7</td>
<td>disp8(An, XnLo)</td>
<td>1</td>
<td>M[An+Xn Lo+disp8]</td>
</tr>
<tr>
<td>Based indexexec long</td>
<td>7</td>
<td>0-7</td>
<td>disp8(An, Xn)</td>
<td>1</td>
<td>M[An+Xn+disp8]</td>
</tr>
<tr>
<td>Absolute short</td>
<td>7</td>
<td>0</td>
<td>Addr16</td>
<td>1</td>
<td>M[addr16]</td>
</tr>
<tr>
<td>Absolute long</td>
<td>7</td>
<td>1</td>
<td>Addr32</td>
<td>2</td>
<td>M[addr32]</td>
</tr>
<tr>
<td>Relative</td>
<td>7</td>
<td>2</td>
<td>disp16(PC)</td>
<td>1</td>
<td>M[PC+disp16]</td>
</tr>
<tr>
<td>Relative indexed short</td>
<td>7</td>
<td>3</td>
<td>disp8(PC, XnLo)</td>
<td>1</td>
<td>M[PC+Xn Lo+disp8]</td>
</tr>
<tr>
<td>Relative indexed long</td>
<td>7</td>
<td>3</td>
<td>disp8(PC, Xn)</td>
<td>1</td>
<td>M[PC+Xn+disp8]</td>
</tr>
<tr>
<td>Immediate</td>
<td>7</td>
<td>4</td>
<td>#data</td>
<td>1-2</td>
<td>No location, data</td>
</tr>
</tbody>
</table>
RTN Description of Addressing

- Addressing modes interpret many items
  - Instruction in the IR register
  - Following 16-bit word: $M_w[PC]$
  - D and A registers in CPU
- Many addressing modes calculate an effective memory address
- Some modes designate a register
- Some modes result in constant operand
- Restrictions exist for some modes
RTN Formatting for EA Calculation

- XR[0..15]<31..0> :=
  D[0..7]<31..0>#A[0..7]<31..0>:
- xr<3..0> := Mw[PC]<15..12>:
- wl := Mw[PC]<11>:
- dsp8<7..0> := Mw[PC]<7..0>:
- index := (wl=0) → XR[xr]<15..0>:
  (wl=1) → XR[xr]<31..0>:

- Index register can be D or A
- Index number for index mode
- Short/long index flag
- Displacement for index mode
- Short
- Long index value

- 4-bit field specifies index register
- Either 16 or 32 bits of index register may be used
- Low order 8-bits are used as offset

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>d/a</td>
<td>Index reg</td>
<td>w/l</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>disp8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0: 16-bit index
1: 32-bit index

0: index is in data registers
1: index is in address registers
Calculating EA

- \( m_d \) and \( r_g \) are 3-bit mode and reg fields
- \( e_a \) is effective address

Define effective address based on mode and register fields

\[
e_a(m_d, r_g) := (\begin{align*}
\cdot (m_d=2) & \rightarrow A[r_g<2..0>] \\
\cdot (m_d=3) & \rightarrow (A[r_g]; A[r_g] \leftarrow A[r_g] + d): \\
\cdot \ldots
\end{align*})
\]
Addressing Mode Highlights

• Modes 0-6 use a register to calculate a memory address
  ▫ Based modes (5-6) require an extra word (16-bits) to specify address
• Mode 7 does not use a register
  ▫ Functionality is expanded by repurposing \texttt{reg} field
  ▫ All variants require extra words to complete the instruction and specify the memory address
Mode 0 and 1: Register Direct

- Mode 1 = data register
- Mode 2 = address register

- Register itself provides place to store result or location of operand
  - No memory address in this mode
Mode 2: Address Register Indirect

- Modes 3 and 4 are the same
  - Autoincrement (3) – register incremented after obtained
  - Autodecrement (4) – register decremented before address obtained
Mode 6: Based Indexed

- Three items added to get address
- Mode 5 (based) is same only does not contain register index
Mode 7-0 and 7-1: Absolute Addressing

- Mode 7-0 – 16-bit addresses
- Mode 7-1 – 32-bit addresses
Mode 7-3: Relative Indexed

- Same as indexed mode but uses PC rather than an A register as base address

```
Relative indexed addressing

<table>
<thead>
<tr>
<th>d/a</th>
<th>Index reg</th>
<th>w/l</th>
<th>disp8 = ldisp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14 13 12</td>
<td>11</td>
<td>10987</td>
</tr>
</tbody>
</table>

0 = 16 bit index
1 = 32 bit index

0: index is in data reg.
1: index is in address reg.

Ex: MOVE.W LDISP (PC, D4), ...

Main memory

Program counter

Operand

Index (16 or 32)

D0-D7
A0-A7
```
Mode 7-4: Immediate

- Access to constants stored as part of the program
  - Constant stored immediately after the instruction word
  - Data length specified by opcode field, not the \textit{md/reg} field

```
\begin{verbatim}
<table>
<thead>
<tr>
<th>Byte</th>
<th>Word</th>
<th>Longword</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ldots</td>
<td>111 100</td>
<td>\ldots 111 100</td>
</tr>
<tr>
<td>00000000</td>
<td>value8</td>
<td>\ldots</td>
</tr>
<tr>
<td>15  8  7  0</td>
<td></td>
<td>15  0</td>
</tr>
<tr>
<td></td>
<td>\textit{value16}</td>
<td>\ldots</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\textit{value16Hi}</td>
</tr>
<tr>
<td>Example: \texttt{MOVE.B} #12, \ldots</td>
<td>Example: \texttt{MOVE.W} #1234, \ldots</td>
<td>Example: \texttt{MOVE.L} #12345678, \ldots</td>
</tr>
</tbody>
</table>
\end{verbatim}
```

Remember big endianess
Result Operand Addressing

- Not all addressing modes can be used for results
  - \( md = 7 \) and \( rg = 2 \) or 3 not allowed
  - Lead to self-modifying code
  - Register immediate is legal for results
Instruction Interpretation

• Instructions fetched 16-bits at a time
  ▫ PC advanced by 2 as 16-bit word is fetched
  ▫ Addressing mode may cause advance of 2 or 4 more words

• Instruction_interpretation := ( Run → ( (IR<15..0> ← Mw[PC]<15..0>: PC ← PC + 2); instruction_execution ); ):
Data Movement Instructions

• Unlike SRC, instruction fields are not standardized
  ▫ Locations and sizes depend on instruction
  ▫ Allows more instructions to be defined in small word size
• Condition codes can be set during move
  ▫ Negative and zero

• \texttt{tmp<31..0>}:  
• move \texttt{(:= op<3..2> := 0) \rightarrow ( 
  tmp \leftarrow \texttt{opnd(md1, rg1);}  
  ( z \leftarrow (\texttt{tmp=0}): N \leftarrow (\texttt{tmp<0}): V \leftarrow 0: C \leftarrow 0):  
  \texttt{rslt(md2, rg2) \leftarrow tmp)}):
Integer ALU Instructions

- 2-operand instructions must specify destination
  - One operand is EA
  - Second operand is Dn (data register direct)
  - 3-bit mode field specifies destination as EA or Dn and operands as bytes, word, or long

<table>
<thead>
<tr>
<th>Byte</th>
<th>Word</th>
<th>Long</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>Dn</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>110</td>
<td>EA</td>
</tr>
</tbody>
</table>
Example: Subtract

sub (:= op=9) \rightarrow (\neg memval(mdl, rgl)) \Rightarrow

(mdl<2>=0) \rightarrow D[rg2] \leftarrow D[rg2] - opnd(mdl, rgl):

(mdl>2>=1) \rightarrow (memval(mdl, rgl) \rightarrow
tmp \leftarrow ea(mdl, rgl);
M[tmp] \leftarrow M[tmp] - D[rg2]):

rslt(mdl, rgl) \leftarrow rslt(mdl, rgl) - D[rg2])

• SUB EA, Dn 1001 rrr mmm aaaaaa

  opcode  Dn  mode  EA
## Arithmetic Shift and Rotates

- **ww** is word size
- **Condition codes**
  - \(N\) = msb of result
  - \(Z\) = set by result
  - \(C\) = last bit shifted out

### Condition codes
- \(N\) = msb of result
- \(Z\) = set by result
- \(C\) = last bit shifted out

### Table of Operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Opcode Word</th>
<th>XV</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASd</td>
<td>EA</td>
<td>11100000d11aaaaaa</td>
<td>xx</td>
<td>ASL</td>
</tr>
<tr>
<td>ASd</td>
<td>#cnt,Dn</td>
<td>1110ccccdww000rrr</td>
<td>xx</td>
<td>ASR</td>
</tr>
<tr>
<td>ASd</td>
<td>Dm,Dn</td>
<td>1110RRRdww100rrr</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>R0d</td>
<td>EA</td>
<td>1110011d11aaaaaa</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>R0d</td>
<td>#cnt,Dn</td>
<td>1110ccccdww011rrr</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>R0d</td>
<td>Dm,Dn</td>
<td>1110RRRdww111rrr</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>LSd</td>
<td>EA</td>
<td>1110001d11aaaaaa</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>LSd</td>
<td>#cnt,Dn</td>
<td>1110ccccdww001rrr</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>LSd</td>
<td>Dm,Dn</td>
<td>1110RRRdww101rrr</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>ROXd</td>
<td>EA</td>
<td>1110010d11aaaaaa</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>ROXd</td>
<td>#cnt,Dn</td>
<td>1110ccccdww010rrr</td>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>ROXd</td>
<td>Dm,Dn</td>
<td>1110RRRdww110rrr</td>
<td>x0</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2004 Pearson Prentice Hall, Inc.
Program Control Instructions

- Conditional branches
  - Use condition code bits (C, N, V, Z)
  - *E.g.* $BVS = \text{branch if overflow set}$
- $Bcc = \text{branch}$
- $DBcc = \text{decrement and branch}$
- $Scc = \text{sets result byte to outcome of test}$

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Code</th>
<th>Logic</th>
<th>Name</th>
<th>Meaning</th>
<th>Code</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>True</td>
<td>0000</td>
<td>1</td>
<td>F</td>
<td>False</td>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>Carry clear</td>
<td>0100</td>
<td>Ĉ</td>
<td>LS</td>
<td>Low or same</td>
<td>0011</td>
<td>$C + Z$</td>
</tr>
<tr>
<td>CS</td>
<td>Carry set</td>
<td>0101</td>
<td>Ĉ</td>
<td>LT</td>
<td>Less than</td>
<td>1101</td>
<td>$N \cdot \overline{V} + \overline{N} \cdot V$</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>0111</td>
<td>Z</td>
<td>MI</td>
<td>Minus</td>
<td>1011</td>
<td>$N$</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
<td>1100</td>
<td>$\overline{N} \cdot \overline{V} + N \cdot V$</td>
<td>NE</td>
<td>Not equal</td>
<td>0110</td>
<td>Z</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
<td>11100</td>
<td>$N \cdot \overline{V} \cdot Z + N \cdot V \cdot \overline{Z}$</td>
<td>PL</td>
<td>Plus</td>
<td>1010</td>
<td>$\overline{N}$</td>
</tr>
<tr>
<td>HI</td>
<td>High</td>
<td>0100</td>
<td>$C \cdot \overline{Z}$</td>
<td>VC</td>
<td>Overflow clear</td>
<td>1000</td>
<td>$\overline{V}$</td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal</td>
<td>1111</td>
<td>$N \cdot \overline{V} + \overline{N} \cdot V + Z$</td>
<td>VS</td>
<td>Overflow set</td>
<td>1001</td>
<td>$V$</td>
</tr>
</tbody>
</table>
More Program Control

- Unconditional branches
  - Map to C goto statement
    - BRA, JMP
  - Sub-routine varieties store PC on stack
    - BSR, JSR

- Sub-routine return instructions
  - Linage uses stack for return address
  - RTR, RTS
Starting a Program

- **Assembler**
  - Convert assembly language text to (binary) machine language
    - Addresses translated using a symbol table
    - Addresses adjusted to allow room for blocks of reserved memory (e.g. an array definition)

- **Linker**
  - Separately assembled modules combined and absolute addresses assigned

- **Loader**
  - Move binary words into memory

- **Run time**
  - PC set to started address of loaded module.
  - OS usually makes a jump or procedure call to the address
Pseudo Operations

- Operation performed by assembler at assembly time not by CPU at run time
- **EQU** - defines constant symbol
  - PI: EQU 3.14
  - Substitution made at assemble time
- **DS.(B, W, L)** - defines block of storage
  - A label is associated with first word of block
  - Line: DS.B 132
  - Program loader (part of OS) accomplishes this
- **#** indicates value of symbol rather than location addressed by symbol
  - MOVE.L #1000, D0 ; moves 1000 to D0
  - MOVE.L 1000, D0 ; moves value addr. 1000 to D0
  - Assembler detects difference
- **ORG** - defines memory address where following code will be stored
  - Start: ORG $4000 ; next instruction/data at addr. 0x4000
- Character constants in single quotes
  - ‘X’
Example: Clearing Block of Memory

MAIN
...
MOVE.L #ARRAY, A0 ;Base of array
MOVE.W #COUNT, D0 ;Number of words to clear
JSR CLEARW ;Make the call
...
CLEARW BRA LOOPE ;Branch for init. Decr.
LOOPS CLR.W (A0)+ ;Autoincrement by 2
LOOPE DBF D0, LOOPS ;Dec.D0, fall through if -1
RTS ;Finished

- Subroutine expects block base in A0, count in D0
- Linkage uses stack pointer
  - A7 cannot be used for anything else
Exceptions

- Changes sequential instruction execution
  - Next instruction fetch not from PC location
  - Exception vector
    - Address supplying the next instruction
- Arise from instruction execution, hardware faults, external conditions
  - Interrupts – externally generated exceptions
  - ALU overflow, power failure, completion of I/O operation, out of range memory access, etc.
- Trace bit = 1 causes exception after every instruction
  - Used for debugging
Exception Handling Steps

1. Status change
   - Temporary copy of status register made
   - Supervisor mode bit S is set and trace bit T is reset
2. Exception vector address obtained
   - Small address made by shift 8-bit vector number left 2
   - Contents of longword at vector address is new address of next instruction
   - Exception handler or interrupt service routine starts at this address
3. Old PC and Status register are pushed onto supervisor stack, \( A7' = SSP \)
4. PC loaded from exception vector address
5. Return from handler is done by \( \text{RTE} \)
   - Works like \( \text{RTR} \) except Status register is restored rather than CCs
Exception Priority

- Method to determine which exception vector to use when multiple exceptions occur at once
- 7 levels of priority in MC68000
  - Status Register contains current priority
- Exceptions with priority \( \leq \) current priority are ignored

- Exceptions are sensed before fetching next instruction
Memory-Mapped I/O

- Part of CPU memory is devoted/reserved for I/O
  - No separate I/O space
  - Not popular for machines having limited address bits
- Single bus needed for memory and I/O
  - Less packaging pins
- Size of I/O and memory spaces independent
  - Many or few I/O devices may be installed
  - Much or little memory may be installed
- Spaces are separated by putting I/O at the top end of address space

24-bit address space with top 32K reserved for I/O

<table>
<thead>
<tr>
<th>0xFFFF</th>
<th>...</th>
<th>0xFF800</th>
<th>0xFF7FF</th>
<th>...</th>
<th>0x000000</th>
</tr>
</thead>
</table>

I/O Space

Memory Space

Notice top 32K can be addressed by a negative 16-bit value