Chapter 3 Topics

3.1 Machine characteristics and performance
3.2 RISC vs. CISC
3.3 A CISC microprocessor: The Motorola MC68000
3.4 The SPARC: a RISC architecture
Practical Aspects of Machine Cost-Effectiveness

- Cost for useful work is fundamental issue
- Mounting, case, keyboard, etc. are dominating the cost of integrated circuits
- Upward compatibility preserves software investment
  - Binary compatibility
  - Source compatibility
  - Emulation compatibility
- Performance: strong function of application
Performance Measures

- **MIPS**: Millions of Instructions Per Second
  - Same job may take more instructions on one machine than on another

- **MFLOPS**: Million Floating Point OPs Per Second
  - Other instructions counted as overhead for the floating point

- **Whetstones**: Synthetic benchmark
  - A program made-up to test specific performance features

- **Dhrystones**: Synthetic competitor for Whetstone
  - Made up to “correct” Whetstone’s emphasis on floating point

- **SPEC**: Selection of “real” programs
  - Taken from the C/Unix world
Quantitative Performance Measurement

Consider two auto routes, the old one, which allowed an average speed of 34 mph, and the new one, which permitted 46 mph. What is the speedup of the new one over the old one?

Conventionally the speedup is calculated as follows:

\[
\text{Speedup} = \frac{\text{SpeedOnNewRoute}}{\text{SpeedOnOldRoute}} = \frac{S_{\text{new}}}{S_{\text{old}}} = \frac{46}{34} = 1.35
\]

For a speedup of 0.35, or 35%.

Alternately, the % speedup can be calculated directly:

\[
\%\text{Speedup} = \left( \frac{S_{\text{new}} - S_{\text{old}}}{S_{\text{old}}} \right) \times 100 = \left( \frac{46 - 34}{34} \right) \times 100 = \frac{12}{34} \times 100 = 35\%
\]
Quantitative Performance Measurement

Many measurements are in terms of the time, $T$, it takes to accomplish some task. Recall that Time, $T$, is the reciprocal of Speed, $S = 1/T$. If the improvement is measured by recording travel time rather than travel speed the equation changes as follows:

\[
\text{Speedup} = \frac{S_{\text{new}}}{S_{\text{old}}} = \frac{1}{T_{\text{new}}} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{96}{71} = 1.35, \text{ or } 35\%
\]

Once again, the % speedup can be calculated directly:

\[
\%\text{Speedup} = \frac{T_{\text{old}} - T_{\text{new}}}{T_{\text{new}}} \times 100 = \frac{96 - 71}{71} \times 100 = \frac{25}{71} \times 100 = 35\%
\]
A Classic Example

**CLASSIC EXAMPLE: ESTIMATING PERFORMANCE**  A certain computer system takes 125 ms to render a certain graphic image, and this time is reduced to 100 ms when a graphics processor card is added to the system. What is the speedup?

\[
\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{125}{100} = 1.25, \text{ or a 25\% speedup}
\]
Getting Finer-Grained

The execution time can be calculated from the count of how many instructions have executed, $IC$, the average number of clock cycles per instruction, $CPI$, and the clock period, $\tau$.

- This is an important equation that will be used throughout the text.

**Execution time**

$$T = IC \times CPI \times \tau$$

**Example 3.1 Speedup Due to a Clock Frequency Increase**

The master clock in a certain computer system is increased in frequency from 700 MHz to 1.2 GHz. What is the speedup due to this improvement if no other factors such as memory access time interfere with the improvement?

Since, according to the problem definition, neither $IC$ nor $CPI$ changed, and since the clock period, $\tau$, is proportional to the reciprocal of clock frequency,

$$\text{Speedup} = \frac{(IC \times CPI \times \tau)_{\text{old}}}{(IC \times CPI \times \tau)_{\text{new}}} = \frac{1/700}{1/1200} = \frac{1200}{700} = \frac{1.71}{1} = 1.71, \text{ or } 71\% \text{ speedup}$$
CISC Versus RISC 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
Design Characteristics of RISCs

- 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
Other RISC Characteristics

- Prefetching of instructions. (Similar to I8086)
- Pipelining: beginning execution of an instruction before the previous instruction(s) have completed. (Will cover in detail in Chapter 5.)
- Superscalar operation—issuing more than one instruction simultaneously. (Instruction-level parallelism. Also covered in Chapter 5.)
- 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 in this Chapter.
Tbl. 3.1 Developing an Instruction Set Architecture

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

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

- **Instruction interpretation: things done for all instructions**
  - The fetch-execute cycle
  - Exception handling (sometimes deferred)

- **Instruction execution: behavior of individual instructions**
  - Grouping of instructions into classes
  - Actions performed by individual instructions
CISC: The Motorola MC68000

- Introduced in 1979
- One of first 32 bit microprocessors
  - Means that most operations are on 32 bit internal data
  - Some operations may use different number of bits
  - External data paths may not all be 32 bits wide
    - MC68000 had a 24 bit address bus
- Complex Instruction Set Computer - CISC
  - Large instruction set
  - 14 addressing modes
Fig. 3.1 MC68000 Programmer’s Model
Features of the 68000 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
    - Discuss: 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, as well as the Interrupt Mask
RTN Processor State for the MC68000

D[0..7]〈31..0〉: General purpose data registers;
A[0..7]〈31..0〉: Address registers;
A7´〈31..0〉: System stack pointer;
PC〈23..0〉: Program counter in original MC68000
IR〈15..0〉: Instruction register;
Status〈15..0〉: System status byte and user status byte;
SP := A[7]: User stack pointer, also called USP;
SSP := A7´: System stack pointer;
C := Status〈0〉: V := Status〈1〉: Carry and oVerflow flags;
Z := Status〈2〉: N := Status〈3〉: Zero and Negative flags;
X := Status〈4〉: Extend flag;
INT〈2..0〉 := Status〈10..8〉: Interrupt mask in system status byte;
S := Status〈13〉: T := Status〈15〉: Supervisor state and Trace mode flags;
Main Memory in the MC68000

Main memory:
\[ \text{Mb}[0..2^{24}-1][7..0] : \text{Memory as bytes} \]
\[ \text{Mw}[ad][15..0] := \text{Mb}[ad]\#\text{Mb}[ad+1] : \text{Memory as words} \]
\[ \text{Ml}[ad][31..0] := \text{Mw}[ad]\#\text{Mw}[ad+2] : \text{Memory as long words} \]

- The word and longword forms are “big-endian”
  - The lowest numbered byte contains the most significant bit (big end) of the word
- Words and longwords have “hard” alignment constraints not described in the above RTN
  - Word addresses must end in one binary 0
  - Longword addresses must end in two binary zeros
MC68000 Supports Several Operand Types

- Like many CISC machines, the 68000 allows one instruction to operate on several types
  - MOVE.B for bytes, MOVE.W for words, and MOVE.L for longwords; also ADD.B, ADD.W, ADD.L, etc.
  - The default, ADD, for example, is Word operands.
  - Operand length is encoded into the instruction word
- Bits coding operand type vary with instruction
  - For use with RTN descriptions, we assume a function $d := \text{datalen}(\text{IR})$ that returns 1, 2, or 4 for operand length
Fig. 3.2 Some MC68000 Instruction Formats

(a) A 1-word move instruction

(b) A 2-word instruction

(c) A 3-word instruction

(d) Instruction with indexed address
General Form of Addressing Modes in the MC68000

- A general address of an operand or result is specified by a 6-bit field with mode and register numbers.

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Provides access paths to operands

- Not all operands and results can be specified by a general address: some must be in registers.
- Not all modes are legal in all parts of an inst.
- Exception: when specifying the destination of a MOVE instruction the mode and reg fields are reversed.
<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Reg.</th>
<th>Assembler Syntax</th>
<th>Extra Words</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data reg. direct</td>
<td>0</td>
<td>0-7</td>
<td>Dn</td>
<td>0</td>
<td>Dn</td>
</tr>
<tr>
<td>Addr. reg. direct</td>
<td>1</td>
<td>0-7</td>
<td>An</td>
<td>0</td>
<td>An</td>
</tr>
<tr>
<td>Addr. reg. 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+XnLo+disp8]</td>
</tr>
<tr>
<td>Based indexed long</td>
<td>6</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>Rel. indexed short</td>
<td>7</td>
<td>3</td>
<td>disp8(PC,XnLo)</td>
<td>1</td>
<td>M[PC+XnLo+disp8]</td>
</tr>
<tr>
<td>Rel. 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>data</td>
</tr>
</tbody>
</table>
The addressing modes interpret many items:
- The instruction: in the IR register
- The following 16 bit word: described as Mw[PC]
- The D and A registers in the CPU

Many addressing modes calculate an effective memory address.
Some modes designate a register.
Some modes result in a constant operand.
There are restrictions on the use of some modes.
RTN Formatting for Effective Address Calculation

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

- Either an A or a D register can be used as an index
- A 4-bit field in the 2nd instruction word specifies the index register
- Low order 8-bits of 2nd word are used as offset
- Either 16 or 32 bits of index register may be used

Index register can be D or A;
Index specifier for index mode;
Short or long index flag;
Displacement for index mode;
Short or long index value;
Modes That Calculate a Memory Address Using a Register

- md and rg are the 3-bit mode and reg. fields.
- ea stands for effective address

\[
e(a, b) := ( \\
\text{md} = 2 \rightarrow A[a,b] ; \\
\text{md} = 3 \rightarrow (A[a,b]; A[a,b] \leftarrow A[a,b] + d) ; \\
\text{md} = 4 \rightarrow (A[a,b] \leftarrow A[a,b] - d; A[a,b]) ; \\
\text{md} = 5 \rightarrow (A[a,b] + Mw[PC]; PC \leftarrow PC + 2) ; \\
\text{md} = 6 \rightarrow (A[a,b] + \text{index} + \text{dsp8}; PC \leftarrow PC + 2) ;
\]

Mode 2 is register indirect; Mode 3 is autoincrement; Mode 4 is autodecrement; Mode 5 is based or offset addressing; Mode 6 is based indexed addressing;
Mode 7 Uses the reg Field to Expand the Number of Modes

These modes still calculate a memory address

\[ ea \ (md, \ rg) := \]

\[ \ldots \]

\[(md = 7 \land rg = 0) \rightarrow \]
\[(Mw[PC]\{\text{sign extend to 32 bits}\}; \ PC \leftarrow PC + 2): \]

Mode 7, register 0 is short absolute;

\[(md = 7 \land rg = 1) \rightarrow \]
\[(Ml[PC]; \ PC \leftarrow PC + 4): \]

Mode 7, register 1 is long absolute;

\[(md = 7 \land rg = 2) \rightarrow \]
\[(PC + Mw[PC]\{\text{sign extend to 32 bits}\}; \ PC \leftarrow PC + 2): \]

Mode 7, register 2 is program counter relative addressing;

\[(md = 7 \land rg = 3) \rightarrow \]
\[(PC + \text{index} + \text{dsp8}; \ PC \leftarrow PC + 2) \}

Mode 7, register 3 is relative indexed.
Fig. 3.3 Mode 2: Address Register Indirect

- Same picture for autoincrement or decrement
  - Address register incremented after address obtained in autoincrement
  - Address register decremented before address obtained in autodecrement

Ex: MOVE (A6), ...
Fig. 3.4 Mode 6: Based Indexed Addressing

Three things are added to get the address:

- **Disp8**: Index register
- **Reg**: Base address
- **A0**: Memory operand

**Example:**

```
MOVE.W LDISP (A6, D4), ...
```
Modes 7-0 and 7-1: Absolute Addressing

Absolute short addressing

<table>
<thead>
<tr>
<th>...</th>
<th>111</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ex: MOVE.B PRINTERPORT.W, ...

(Sign extend to 32-bits)

Absolute long addressing

<table>
<thead>
<tr>
<th>...</th>
<th>111</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr32Hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addr32Lo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ex: MOVE.W INTVECT.L, ...

Absolute addresses can be 16 or 32 bits
Mode 7-3 Relative Indexed Addressing

Relative indexed addressing

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

- Same as indexed mode but uses PC instead of A register as base
memval(md, rg) :=
  ( (md\langle 2..1 \rangle = 1) \lor (md\langle 2..1 \rangle = 2) \lor (md\langle 2..0 \rangle = 6) \lor
   ((md\langle 2..0 \rangle = 7) \land (rg\langle 2 \rangle =0)) )
:
opnd(md, rg) := ( (d=1) \rightarrow \text{opndb}(md, rg): (d=2) \rightarrow \text{opndw}(md, rg):
   (d=4) \rightarrow \text{opndl}(md, rg) )
opndl(md, rg)\langle 31..0 \rangle := ( \ldots ):
opndw(md, rg)\langle 15..0 \rangle := ( \begin{align*}
  \text{memval}(md, rg) \rightarrow & \text{Mw}[\text{ea}(md, rg)]\langle 15..0 \rangle: \\
  \text{md} = 0 \rightarrow & \text{D}[rg]\langle 15..0 \rangle: \\
  \text{md} = 1 \rightarrow & \text{A}[rg]\langle 15..0 \rangle: \\
  (md = 7 \land rg = 4) \rightarrow & (\text{Mw}[\text{PC}]\langle 15..0 \rangle: \text{PC} \leftarrow \text{PC}+2 )
\end{align*} ):
opndb(md, rg)\langle 7..0 \rangle := ( \begin{align*}
  (md = 7 \land rg = 4) \rightarrow & (\text{Mw}[\text{PC}]\langle 7..0 \rangle: \text{PC} \leftarrow \text{PC}+2 )
\end{align*} ):

A memory address is used with these modes only;
The operand length in the instruction tells which to use.
A long operand can be

A word operand is similar but needs only a 16 bit immediate following the instruction word;
Byte operands

instruction word.
Modes 0 and 1: Register Direct Addressing

- The register itself provides a place to store a result or a place to get an operand.
- There is no memory address with this mode.
Fig. 3.5 Mod 7-4: Immediate Addressing: Operands are stored *in the instruction*.

Instruction word and 1 or 2 following words

<table>
<thead>
<tr>
<th>Byte</th>
<th>Word</th>
<th>Longword</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Byte Diagram" /></td>
<td><img src="image2.png" alt="Word Diagram" /></td>
<td><img src="image3.png" alt="Longword Diagram" /></td>
</tr>
</tbody>
</table>

- Data length is specified by the opcode field, not the Mode/Reg field.
Not Every Addressing Mode Can Be Used for Results

\[
\text{rsltadr}(md, rg) := \text{memval}(md, rg) \land \neg (md=7 \land (rg=2 \lor rg=3)):
\]

- The MC68000 disallows relative addressing (md7 rg 2 or 3) for results
- This is captured in RTN by defining a function that is true (=1) if the memory address specified by the mode is legal for results
- Register immediate is also legal for results, but will be handled separately
Result Modes Must Have a Place to Write Data:
Memory or Register

\[
\text{rsltl}(md, rg) \langle 31..0 \rangle := ( \\
\text{rsltadr}(md, rg) \rightarrow Ml[ea(md, rg)] \langle 31..0 \rangle : \\
md = 0 \rightarrow D[rg] \langle 31..0 \rangle : \\
md = 1 \rightarrow A[rg] \langle 31..0 \rangle 
)
\]

32 bit result;

\[
\text{rsltw}(md, rg) \langle 15..0 \rangle := ( \\
\text{rsltadr}(md, rg) \rightarrow Mw[ea(md, rg)] \langle 15..0 \rangle : \\
md = 0 \rightarrow D[rg] \langle 15..0 \rangle : \\
md = 1 \rightarrow A[rg] \langle 15..0 \rangle 
)
\]

16 bit result;

\[
\text{rsltb}(md, rg) \langle 7..0 \rangle := ( \\
\text{rsltadr}(md, rg) \rightarrow Mb[ea(md, rg)] \langle 7..0 \rangle : \\
md = 0 \rightarrow D[rg] \langle 7..0 \rangle : \\
md = 1 \rightarrow A[rg] \langle 7..0 \rangle 
)
\]

8 bit result.

\[
\text{rslt}(md, rg) := ( \\
(d=1) \rightarrow \text{rsltb}(md, rg) : (d=2) \rightarrow \text{rsltw}(md, rg) : \\
(d=4) \rightarrow \text{rsltl}(md, rg) 
)
\]

The result length in the instruction tells which to use.
Instruction interpretation is simple when exceptions are ignored

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

- Instructions are fetched 16 bits at a time
  - PC is advanced by 2 as each 16-bit word is fetched
  - Addressing mode may advance it a total of 2 or 4 or more words, under command from the control unit.
### Tbl. 3.3  Data Movement Instructions in the MC68000

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Operands</th>
<th>First Instruction Word</th>
<th>X N Z V C</th>
<th>Operation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE.B</td>
<td>EAs, EAd</td>
<td>0001ddddddssssss</td>
<td>- x x 0 0</td>
<td>dst ← src</td>
<td>byte</td>
</tr>
<tr>
<td>MOVE.W</td>
<td>EAs, EAd</td>
<td>0011ddddddssssss</td>
<td>- x x 0 0</td>
<td>dst ← src</td>
<td>word</td>
</tr>
<tr>
<td>MOVE.L</td>
<td>EAs, EAd</td>
<td>0010ddddddssssss</td>
<td>- x x 0 0</td>
<td>dst ← src</td>
<td>long</td>
</tr>
<tr>
<td>MOVEA.W</td>
<td>EAs, An</td>
<td>0011rrr001ssssss</td>
<td>- - - - -</td>
<td>An ← src</td>
<td>word</td>
</tr>
<tr>
<td>MOVEA.L</td>
<td>EAs, An</td>
<td>0010rrr001ssssss</td>
<td>- - - - -</td>
<td>An ← src</td>
<td>long</td>
</tr>
<tr>
<td>LEA.L</td>
<td>EAc, An</td>
<td>0100aaa111ssssss</td>
<td>- - - - -</td>
<td>An ← EA</td>
<td>address</td>
</tr>
<tr>
<td>EXG</td>
<td>Dx, Dy</td>
<td>1100xxx101000yyy</td>
<td>- - - - -</td>
<td>Dx ↔ Dy</td>
<td>long</td>
</tr>
</tbody>
</table>

- The op code location and size depends on the instruction (Compare to SRC).
RTN for a Typical MC68000 Move Instruction

- The instruction format for Move includes mode and register for source and destination addresses

```
op(3..0) := IR(15..12): rg1(2..0) := IR(2..0): md1(2..0) := IR(5..3):
rg2(2..0) := IR(11..9): md2(2..0) := IR(8..6):
```

```
tmp(31..0):
move (:= op(3..2) := 0) → (
    tmp ← opnd(md1, rg1);
    ( Z ← (tmp=0): N ← (tmp<0): V ← 0: C ← 0 ):
rslt(md2, rg2) ← tmp
):
```

- The temporary register tmp is used because every invocation of opnd() causes another fetch
### MC68000 Integer Arithmetic and Logic Instructions

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst. word</th>
<th>X</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
<th>Operation</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>EA,Dn</td>
<td>1101rrrrmmmmmaaaaa</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>dst←dst+src</td>
<td>b, w, l</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>EA,Dn</td>
<td>1001rrrrmmmmmaaaaaa</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>dst←dst-src</td>
<td>b, w, l</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>EA,Dn</td>
<td>1011rrrrxxaaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>dst-src</td>
<td>b, w, l</td>
<td></td>
</tr>
<tr>
<td>CMPI</td>
<td>#dat,EA</td>
<td>00001100wwaaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>dst-imm.data</td>
<td>b, w, l</td>
<td></td>
</tr>
<tr>
<td>MULS</td>
<td>EA, Dn</td>
<td>1100rrr111aaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>Dn←Dn*src</td>
<td>l←w*w</td>
</tr>
<tr>
<td>MULU</td>
<td>EA, Dn</td>
<td>1100rrr011aaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>Dn←Dn*src</td>
<td>l←w*w</td>
</tr>
<tr>
<td>DIVS</td>
<td>EA, Dn</td>
<td>1000rrr111aaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>Dn←Dn/src</td>
<td>l←l/w</td>
</tr>
<tr>
<td>DIVU</td>
<td>EA, Dn</td>
<td>1000rrr011aaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>Dn←Dn/src</td>
<td>l←l/w</td>
</tr>
<tr>
<td>AND</td>
<td>EA, Dn</td>
<td>1100rrrrmmmmmaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>dst←dst∧src</td>
<td>b, w, l</td>
</tr>
<tr>
<td>OR</td>
<td>EA, Dn</td>
<td>1000rrrrmmmmmaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>dst←dst∨src</td>
<td>b, w, l</td>
</tr>
<tr>
<td>EOR</td>
<td>EA, Dn</td>
<td>1011rrrrwwwwaaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>dst←dst⊕src</td>
<td>b, w, l</td>
</tr>
<tr>
<td>CLR</td>
<td>EAs</td>
<td>01000010wwaaaaaa</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>dst←0</td>
<td>b, w, l</td>
</tr>
<tr>
<td>NEG</td>
<td>EAs</td>
<td>01000100wwaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>dst←–dst</td>
<td>b, w, l</td>
</tr>
<tr>
<td>TST</td>
<td>EAs</td>
<td>01001010wwaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>dst−0</td>
<td>b, w, l</td>
</tr>
<tr>
<td>NOT</td>
<td>EAs</td>
<td>01000110wwaaaaaa</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>dst←−dst</td>
<td>b, w, l</td>
</tr>
</tbody>
</table>

aaaaaaa is the 6-bit addressing mode specifier mmmrrr

www: B—100, W—101, L—110

xxx: B—000, W—001, L—010
All 2-operand ALU instructions are either $D \rightarrow EA$ or $EA \rightarrow D$. Which is it?

- Only one operand uses EA
- The other operand is always accessed by Data register direct
- The 3-bit mmm field specifies whether $D$ is the source or destination, and whether it is B, W, or L

<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>

Ex: SUB EA, Dn: $1011 \ rrr \ mmm \ aaaaaa$

Note: There are several exceptions to the rule above. See text and Mfr. Data sheet.
RTN Description of a Typical MC68000 Arithmetic Instruction

- Subtract is a typical arithmetic instruction
- Need a temporary register to hold an address

tmp<31..0>: temporary register for address

\[
\text{sub (:= op=9) } \rightarrow ( \\
\text{ (md2(2)=0) } \rightarrow D[rg2] \leftarrow D[rg2] - \text{opnd(md1, rg1)}: \\
\text{ (md2(2)=1) } \rightarrow (\text{memval(md1, rg1)} \rightarrow (\text{tmp} \leftarrow \text{ea(md1, rg1)}; \\
\quad \text{M[tmp]} \leftarrow \text{M[tmp]} - D[rg2]) \\
\quad \neg \text{memval(md1, rg1)} \rightarrow \text{rslt(md1, rg1)} \leftarrow \text{rslt(md1, rg1)} - D[rg2]) \\
\) \\
\]

- This definition does not handle the condition codes
## MC68000 Arithmetic Shifts and Single Word Rotates

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst. word</th>
<th>XNZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASd</td>
<td>EA</td>
<td>1110000d11aaaaaa</td>
<td>xxxxx</td>
</tr>
<tr>
<td>ASd</td>
<td>#cnt,Dn</td>
<td>1110cccdww000rrr</td>
<td>xxxxx</td>
</tr>
<tr>
<td>ASd</td>
<td>Dm,Dn</td>
<td>1110RRRdww100rrr</td>
<td>xxxxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst. word</th>
<th>XNZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROd</td>
<td>EA</td>
<td>1110011d11aaaaaa</td>
<td>-xx0x</td>
</tr>
<tr>
<td>ROd</td>
<td>#cnt,Dn</td>
<td>1110cccdww111rrr</td>
<td>-xx0x</td>
</tr>
<tr>
<td>ROd</td>
<td>Dm,Dn</td>
<td>1110RRRdww111rrr</td>
<td>-xx0x</td>
</tr>
</tbody>
</table>

- d is L or R for left or right shift, respectively
- EA form has shift count of 1
- ww is word size: 00–Byte, 01–Word, 10–Long Word
MC68000 Logical Shifts and Extended Rotates

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst. word</th>
<th>XNZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSd</td>
<td>EA</td>
<td>1110001d11aaaaaa</td>
<td>xxx0x</td>
</tr>
<tr>
<td>LSd</td>
<td>#cnt,Dn</td>
<td>1110cccdww001rrr</td>
<td>xxx0x</td>
</tr>
<tr>
<td>LSd</td>
<td>Dm,Dn</td>
<td>1110RRRdww101rrr</td>
<td>xxx0x</td>
</tr>
<tr>
<td>ROXd</td>
<td>EA</td>
<td>1110010d11aaaaaa</td>
<td>xxx0x</td>
</tr>
<tr>
<td>ROXd</td>
<td>#cnt,Dn</td>
<td>1110cccdww010rrr</td>
<td>xxx0x</td>
</tr>
<tr>
<td>ROXd</td>
<td>Dm,Dn</td>
<td>1110RRRdww110rrr</td>
<td>xxx0x</td>
</tr>
</tbody>
</table>

- Field ww specifies byte, word, or longword
- N & Z set according to result, C= last bit shifted out
## MC68000 Conditional Branch and Test Instructions

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst. word</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bcc</td>
<td>disp</td>
<td>0110ccccccddddddddd</td>
<td>if (cond) then PC ← PC + disp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDDDDDDDDDDDDDDDDDDDD</td>
<td></td>
</tr>
<tr>
<td>DBcc</td>
<td>Dn,disp</td>
<td>0101cccccl1001rrr</td>
<td>if ¬(cond) then Dn←Dn-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if (Dn≠-1) then PC←PC+disp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>else PC ← PC + 2</td>
</tr>
<tr>
<td>Scc</td>
<td>EA</td>
<td>0101cccccl1aaaaaaaaa</td>
<td>if (cond) then (EA) ← FFH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>else (EA) ← 00H</td>
</tr>
</tbody>
</table>

- disp is ddddddddddd unless ddddddddd = 0, in which case it is contained in the extra word DDDDDDDDDDDDDDDDDDDDDDDDDDD
- DBcc is used for counted loops with an optional end condition.
  - "Decrement and branch until cond."
- Scc sets a byte to the outcome of a test
Conditions That Can Be Evaluated for Branch, Etc.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning Name</th>
<th>Flag expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>true</td>
<td>T 1</td>
</tr>
<tr>
<td>0001</td>
<td>false</td>
<td>F 0</td>
</tr>
<tr>
<td>0100</td>
<td>carry clear</td>
<td>CC C</td>
</tr>
<tr>
<td>0101</td>
<td>carry set CS</td>
<td>C</td>
</tr>
<tr>
<td>0111</td>
<td>equal</td>
<td>EQ Z</td>
</tr>
<tr>
<td>0110</td>
<td>not equal</td>
<td>NE Z</td>
</tr>
<tr>
<td>1011</td>
<td>minus</td>
<td>MI N</td>
</tr>
<tr>
<td>1010</td>
<td>plus</td>
<td>PL N</td>
</tr>
<tr>
<td>0011*</td>
<td>low or same</td>
<td>LS C+Z</td>
</tr>
<tr>
<td>1101</td>
<td>less than</td>
<td>LT N·V+Ñ·V</td>
</tr>
<tr>
<td>1100</td>
<td>greater or equal</td>
<td>GE N·V+Ñ·V</td>
</tr>
<tr>
<td>1110</td>
<td>greater than</td>
<td>GT N·V·Z+Ñ·V·Z</td>
</tr>
<tr>
<td>1111</td>
<td>less or equal</td>
<td>LE N·V+Ñ·V+Z</td>
</tr>
<tr>
<td>0010*</td>
<td>high</td>
<td>HI C·Z</td>
</tr>
<tr>
<td>1000</td>
<td>overflow clear</td>
<td>VC V</td>
</tr>
<tr>
<td>1001</td>
<td>overflow set</td>
<td>VS V</td>
</tr>
</tbody>
</table>

\*Assumes unsigned operands
Conditional Branches First Set Condition Codes, Then Branch

\[
\text{if ( } X = 0 \text{ ) goto LOC}
\]

\[
\begin{align*}
\text{TST} & \quad X \quad ;\text{ands } X \text{ with itself and sets } N \text{ and } Z \\
\text{BEQ} & \quad \text{LOC} \quad ;\text{branch to } \text{LOC if } X=0
\end{align*}
\]

LOC:

- EQ tests the right condition codes for \( =0 \), as above, or \( A=B \) following a compare, CMP A,B
### MC68000 Unconditional Control Transfers

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst.word</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA</td>
<td>disp</td>
<td>01100000ddddddddddd</td>
<td>PC ← PC + disp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDDDDDDDDDDDDDDDDDDDDDDDDD</td>
<td></td>
</tr>
<tr>
<td>BSR</td>
<td>disp</td>
<td>01100001ddddddddddd</td>
<td>-(SP) ← PC; PC ← PC + disp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDDDDDDDDDDDDDDDDDDDDDDDD</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>EA</td>
<td>0100111011aaaaaa</td>
<td>PC ← EA</td>
</tr>
<tr>
<td>JSR</td>
<td>EA</td>
<td>0100111010aaaaaa</td>
<td>-(SP) ← PC; PC ← EA</td>
</tr>
</tbody>
</table>

- Subroutine links push the return address onto the stack pointed to by A7 = SP
### MC68000 Subroutine Return Instructions

<table>
<thead>
<tr>
<th>Op.</th>
<th>Operands</th>
<th>Inst. word</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTR</td>
<td></td>
<td>0100111001110111</td>
<td>(CC \leftarrow (SP)+; \text{PC} \leftarrow (SP)+)</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td>0100111001110101</td>
<td>(\text{PC} \leftarrow (SP)+)</td>
</tr>
<tr>
<td>LINK</td>
<td>An,disp</td>
<td>0100111001011010rrr</td>
<td>(-(SP) \leftarrow \text{An}; \text{An} \leftarrow \text{SP};)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{DDDDDDDDDDDDDDDDDDDDDDDDDD}</td>
<td>(\text{SP} \leftarrow \text{SP} + \text{disp})</td>
</tr>
<tr>
<td>UNLK</td>
<td>An</td>
<td>010011100101111rrr</td>
<td>(\text{SP} \leftarrow \text{An}; \text{An} \leftarrow (SP)+)</td>
</tr>
</tbody>
</table>

- Subroutine linkage uses stack for return address
- LINK and UNLK allocate and de-allocate multiple word stack frames
Figure 3.6 Example Program to Search an Array

- Program searches an array of bytes to find the first carriage return, ASCII code 13

```
CR    EQU    13          ;Define return character.
LEN   EQU    132         ;Define line length.
ORG   $1000          ;Locate LINE at 1000H.
LINE  DS.B   LEN        ;Reserve LEN bytes of storage.
MOVE.B #LEN-1,D0      ;Initialize D0 to count-1.
MOVEA.L #LINE,A0      ;A0 gets start address of array.
LOOP  CMPI.B (A0)+,#CR  ;Make the comparison.
       DBEQ   D0,LOOP      ;Double test: if LINE[131-D0]≠13
       <next instruction> ; then decr. D0; if D0≠-1 branch
                      ; to LOOP, else to next inst.
```
Pseudo Operations in the MC68000 Assembler

- A *Pseudo Operation* is one that is performed by the assembler at assembly time, not by the CPU at run time.

- EQU defines a symbol to be equal to a constant. Substitution is made at assemble time.
  
  ```
  Pi EQU 3.14
  ```

- DS.B (W or .L) defines a block of storage
  
  - Any label is associated with the first word of the block
  
  ```
  Line DS.B 132
  ```
  
  - The program loader (part of the operating system) accomplishes this

  -more-
Pseudo Operations in the MC68000 Assembler (cont’d.)

- #symbol indicates the value of the symbol instead of a location addressed by the symbol
  
  MOVE.L #1000, D0  ;moves 1000 to D0
  MOVE.L 1000, D0   ;moves value at addr. 1000 to D0

  - The assembler detects the difference and assembles the appropriate instruction.

- ORG specifies a memory address as the origin where the following code will be stored
  
  Start ORG $4000 ;next instruction/data will be loaded at address 4000H.

  - The Motorola assembler uses $ in front of a number to indicate hexadecimal

- Character constants are in single quotes: ‘X’
Review of Assembly, Link, Load, and Run Times

- At **assemble time**, assembly language text is converted to (binary) machine language
  - They may be generated by translating instructions, hexadecimal or decimal numbers, characters, etc.
  - Addresses are translated by way of a symbol table
  - Addresses are adjusted to allow for blocks of memory reserved for arrays, etc.
- At **link time**, separately assembled modules are combined & absolute addresses assigned
- At **load time**, the binary words are loaded into memory
- At **run time**, the PC is set to the starting address of the loaded module. (Usually the O.S. makes a jump or procedure call to that address.)
MC68000 Assembly Language Example: Clear a Block

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 the stack pointer, so A7 cannot be used for anything else
Exceptions: Changes to Sequential Instruction Execution

- Exceptions, also called interrupts, cause next instruction fetch from other than PC location
  - Address supplying next instruction called exception vector
- Exceptions can arise from instruction execution, hardware faults, and external conditions
  - Externally generated exceptions usually called interrupts
  - Arithmetic overflow, power failure, I/O operation completion, and out of range memory access are some causes
- A trace bit =1 causes an exception after every instruction
  - Used for debugging purposes
Steps in Handling MC68000 Exceptions

1) Status change
   - Temporary copy of status register is made
   - Supervisor mode bit S is set, trace bit T is reset

2) Exception vector address is obtained
   - Small address made by shifting 8 bit vector number left 2
   - Contents of the longword at this vector address is the address of the next instruction to be executed
   - The exception handler or interrupt service routine starts there

3) Old PC and Status register are pushed onto supervisor stack, addressed by A7’ = SSP

4) PC is loaded from exception vector address
   - Return from handler is done by RTE
     - Like RTR except restores Status reg. instead of CCs
Exception Priorities

- When several exceptions occur at once, which exception vector is used?
- Exceptions have *priorities*, and highest priority exception supplies the vector
- MC68000 allows 7 levels of priority
- Status register contains current priority
- Exceptions with priority ≤ current are ignored
Exceptions and Reset Both Affect Instruction Interpretation

- More processor state needed to describe reset and exception processing

Reset:
- exc_req:
- exc_lev<2..0>: Single bit exception request
- vect<7..0>: Exception Level
- exc := exc_req \land (exc_lev<2..0> > INT<2..0>): There is a request, and the request level is > current mask in status reg.

- exc_lev is the highest priority of any pending exception
Exceptions are Sensed Before Fetching Next Instruction

Instruction_interpretation := ( 
Run ∧ ¬(Reset ∨ exc) → (IR ← Mw[PC] : PC ← PC + 2); Normal execution state
Reset → (INT(2..0) ← 7 : S ← 1 : T ← 0: Machine reset
  SSP ← MI[0] : PC ← MI[4] :
  Reset ← 0 : Run ← 1 );
Run ∧ ¬Reset ∧ exc → (SSP ← SSP - 4; MI[SSP] ← PC; Exception handling
  SSP ← SSP - 2; Mw[SSP] ← Status;
  S ← 1 : T ← 0 : INT(2..0) ← exc_lev(2..0): 
  PC ← MI[vect(7..0)#002] );
instruction_execution )

- Reset starts the computer with a stack pointer from location 0 at the address from location 4
Memory Mapped I/O

- No separate I/O space. Part of cpu memory space is devoted/reserved for I/O instead of RAM or ROM.
- Example: MC68000 has a total 24-bit address space. Suppose the top 32K is reserved for I/O:

```
FFFFFFFFH
  .  .  .
FF8000H      } I/O Space
FF7FFFH
  .  .  .
000000H      } Memory Space
```

Notice that top 32K can be addressed by a *negative* 16-bit value.
Memory Mapped I/O in the MC68000

- Memory mapped I/O allows μprocessor chip to have one bus for both memory and I/O
  - Multiple wires for both address and data
- I/O uses address space that could otherwise contain memory
  - Not popular with machines having limited address bits
- Sizes of I/O & 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 top end of the address space
MC68000 has a 24 bit address bus

Address space runs from 000000H up to FFFFFFFFH.

A 16 bit address constant can be positive - and sign extend to an address running from 000000H up to the maximum positive value, or negative - and sign extend to an address running from FFFFFFFFH down to the last negative 16 bit value.

I/O addresses in latter range can be accessed by a 16 bit constant.
The SPARC (Scalable Processor Architecture) as a RISC Microprocessor Architecture

- The SPARC is a general register, Load/Store architecture
- It has only two addressing modes. Address =
  - (Reg+Reg), or (Reg + 31-bit constant)
- Instructions are all 32 bits in length
- SPARC has 69 basic instructions
- Separate floating point register set
- First implementation had a 4 stage pipeline
- Some important features not inherently RISC
  - Register windows: separate but overlapping register sets available to calling and called routines
  - 32 bit address, big-endian organization of memory
Fig. 3.9 The SPARC Processor State
Fig. 3.10  Register Windows: an Important Concept in SPARC
SPARC Memory

RTN for the SPARC memory:
Mb[0..2^{32}-1]<7..0>:
  Byte memory;
Mh[a]<15..0> := Mb[a]<7..0>#Mb[a+1]<7..0>:
  Halfword memory;
M[a]<31..0> := Mh[a]<15..0>#Mh[a+2]<15..0>:
  Word memory.
Register Windows Format the General Registers

- 32 general integer and address registers are accessible at any one time
  - Global registers G0..G7 are not in any window
  - G0 is always zero: writes to G0 are ignored, reads return 0
  - The other 24 are in a movable window from a total set of 120
- On subroutine call, the starting point changes so that 24-31 before call become 8-15 after
- Regs. 8-15 are used for incoming parameters
- Regs. 24-31 are for outgoing parameters
- Current Window Pointer CWP locates reg. 8
- Overflow of reg. space causes trap
SAVE, RESTORE and the Current Window Pointer

- CWP points to the register currently called G8
- SAVE moves it to point of the old G24
  - This makes the old G24..G31 into the new G8..G15
- If parameters are placed in G24..G31 by the caller, the callee can get them from G8..G15
- When all windows are used, SAVE traps to a routine that saves registers to memory
- Windows wrap around in the available registers
  - Window overflow “spills” the first window & reuses its space
SPARC Operand Addressing

- One mode computes address as sum of 2 registers; G0 gives zero if used
- The other mode adds sign extended 13 bit constant to a register
- These can serve several purposes
  - Indexed: base in one reg., index in another
  - Register indirect: G0+Gn
  - Displacement: Gn+const, n≠0
  - Absolute: G0+const.
- Absolute addressing can only reach the bottom or top 4K bytes of memory
RTN for SPARC Instruction Formats

\[
\begin{align*}
op(1..0) & := IR(31..30): \text{Instruction class, op code for format 1;} \\
disp30(29..0) & := IR(29..0): \text{Word displacement for call, format 1;} \\
a & := IR(29): \text{Annul bit for branches, format 2a;} \\
cond(3..0) & := IR(28..25): \text{Branch condition select, format 2a;} \\
rd(4..0) & := IR(29..25): \text{Destination register for formats 2b & 3;} \\
op2(2..0) & := IR(24..22): \text{Op code for format 2;} \\
disp22(21..0) & := IR(21..0): \text{Constant for branch displacement or sethi;} \\
op3(5..0) & := IR(24..19): \text{Op code for format 3;} \\
rs1(4..0) & := IR(18..14): \text{Source register 1 for format 3;} \\
opf(8..0) & := IR(13..5): \text{Sub-op code for floating point, format 3a;} \\
i & := IR(13): \text{Immediate operand indicator, formats 3b & c;} \\
simm13(12..0) & := IR(12..0): \text{Signed immediate operand for format 3c;} \\
rs2(4..0) & := IR(4..0): \text{Source register 2 for format 3b.}
\end{align*}
\]
Fig. 3.11  SPARC Instruction Formats

<table>
<thead>
<tr>
<th>Format number</th>
<th>SPARC instruction formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31 30 29</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2a. Branches</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2b. sethi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31 30 29 28 25 24 22 21</td>
</tr>
<tr>
<td></td>
<td>0 0 a</td>
</tr>
<tr>
<td></td>
<td>0 0</td>
</tr>
</tbody>
</table>

| 3a. Floating point |                           |
| 3b. Data movement  |                           |
| 3c. ALU            |                           |
| op | rd | op3 | rs1 | opf | rs2 |
| op | rd | op3 | rs1 | 0   | asi | rs2 |
| op | rd | op3 | rs1 | 1   | simm13 |

- Three basic formats with variations
RTN For SPARC Addressing Modes

\[
\text{adr}(31..0) := (i=0 \rightarrow r[rs1] + r[rs2]): \quad \text{Address for load, store,}
\]
\[
i=1 \rightarrow r[rs1] + \text{simm13}(12..0) \{\text{sign ext.}}): \quad \text{and jump;}
\]
\[
\text{calladr}(31..0) := \text{PC}(31..0) + \text{disp30}(29..0) \#002: \quad \text{Call relative address;}
\]
\[
\text{bradr}(31..0) := \text{PC}(31..0) + \text{disp22}(21..0) \#002\{\text{sign ext.}}): \quad \text{Branch address.}
\]
RTN For SPARC Instruction Interpretation

\[
\text{instruction\_interpretation} := (\text{IR } \leftarrow \text{M}[\text{PC}]; \text{instruction\_execution}; \text{update\_PC\_and\_nPC}; \text{instruction\_interpretation})
\]
### Tbl. 3.8 SPARC Data Movement Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Op.</th>
<th>OPCODE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldsb</td>
<td>11</td>
<td>00 1001</td>
<td>Load signed byte</td>
</tr>
<tr>
<td>ldsh</td>
<td>11</td>
<td>00 1010</td>
<td>Load signed halfword</td>
</tr>
<tr>
<td>ldsw</td>
<td>11</td>
<td>00 1000</td>
<td>Load signed word</td>
</tr>
<tr>
<td>ldub</td>
<td>11</td>
<td>00 0001</td>
<td>Load unsigned byte</td>
</tr>
<tr>
<td>lduh</td>
<td>11</td>
<td>00 0010</td>
<td>Load unsigned halfword</td>
</tr>
<tr>
<td>ldd</td>
<td>11</td>
<td>00 0011</td>
<td>Load doubleword</td>
</tr>
<tr>
<td>stb</td>
<td>11</td>
<td>00 0101</td>
<td>Store byte</td>
</tr>
<tr>
<td>sth</td>
<td>11</td>
<td>00 0110</td>
<td>Store halfword</td>
</tr>
<tr>
<td>stw</td>
<td>11</td>
<td>00 0100</td>
<td>Store word</td>
</tr>
<tr>
<td>std</td>
<td>11</td>
<td>00 0111</td>
<td>Store double word</td>
</tr>
<tr>
<td>swap</td>
<td>11</td>
<td>00 1111</td>
<td>Swap register with memory</td>
</tr>
<tr>
<td>ar</td>
<td>10</td>
<td>00 0010</td>
<td>Rdst ← Rsrc1 OR Rsrc2 (or immediate)</td>
</tr>
<tr>
<td>sethi</td>
<td>00</td>
<td>Op2=100</td>
<td>High order 22 bits of Rdst ← disp22</td>
</tr>
</tbody>
</table>
Register and Immediate Moves in the SPARC

- OR is used with a G0 operand to do register to register moves
- To load a register with a 32 bit constant, a 2 instruction sequence is used
  - SETHI #upper22, R17
  - OR R17, #lower10, R17
- Double words are loaded into an even register and the next higher odd one
- Floating point instructions are not covered, but the 32 FP registers can hold single length numbers, or 16 64-bit FP, or 8 128-bit FP numbers
### Tbl. 3.9 Typical SPARC Arithmetic Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>OPCODE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0X 0000</td>
<td>Add or add and set condition codes</td>
</tr>
<tr>
<td>addc</td>
<td>0X 1000</td>
<td>Add with carry: set CCs or not</td>
</tr>
<tr>
<td>sub</td>
<td>0X 0100</td>
<td>Subtract: set CCs or not</td>
</tr>
<tr>
<td>subc</td>
<td>0X 1100</td>
<td>Subtract with borrow: set CCs or not</td>
</tr>
<tr>
<td>mulscc</td>
<td>10 1100</td>
<td>Do one step of multiply</td>
</tr>
</tbody>
</table>

- All are format 3, Op=10
- CCs are set if X=1 and not if X=0
- Both register and immediate forms are available
- Multiply is done by software using MULSCC or using floating point instructions
  - Multiply is hard to do in one clock but multiply step is not
### Tbl. 3.10 SPARC Logical and Shift Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>OPCODE</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0S 0001</td>
<td>AND, set CCs if S=1 or not if S=0</td>
</tr>
<tr>
<td>ANDN</td>
<td>0S 0101</td>
<td>NAND, set CCs or not</td>
</tr>
<tr>
<td>OR</td>
<td>0S 0010</td>
<td>OR, set CCs or not</td>
</tr>
<tr>
<td>ORN</td>
<td>0S 0110</td>
<td>NOR, set CCs or not</td>
</tr>
<tr>
<td>XOR</td>
<td>0S 0011</td>
<td>XNOR(Equiv), set CCs or not</td>
</tr>
<tr>
<td>SLL</td>
<td>10 0101</td>
<td>Shift left logical, count in RSRC2 or imm13</td>
</tr>
<tr>
<td>SRL</td>
<td>10 0110</td>
<td>Shift right logical, count in RSRC2 or imm13</td>
</tr>
<tr>
<td>SRA</td>
<td>10 0111</td>
<td>Shift right arithmetic, count as above</td>
</tr>
</tbody>
</table>

- All instructions use format 3 with op=10
  - Both register and immediate forms are available
- Condition codes set if S=1 & undisturbed if S=0
### Tbl. 3.11 SPARC Branch and Control Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Fmt.</th>
<th>Op</th>
<th>OPCODE or Op2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>2</td>
<td>00</td>
<td>010</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>bcc</td>
<td>2</td>
<td>00</td>
<td>010</td>
<td>Conditional branch</td>
</tr>
<tr>
<td>call</td>
<td>1</td>
<td>01</td>
<td></td>
<td>Call &amp; save PC in R15</td>
</tr>
<tr>
<td>jmp</td>
<td>3</td>
<td>11</td>
<td>1000</td>
<td>Jmp to EA, save PC in Rdst</td>
</tr>
<tr>
<td>save</td>
<td>3</td>
<td>11</td>
<td>1100</td>
<td>New register window, &amp; ADD</td>
</tr>
<tr>
<td>restore</td>
<td>3</td>
<td></td>
<td>11 1101</td>
<td>Restore reg. window, &amp; ADD</td>
</tr>
</tbody>
</table>

Some condition fields:

<table>
<thead>
<tr>
<th>Inst.</th>
<th>COND</th>
<th>Inst.</th>
<th>COND</th>
<th>Inst.</th>
<th>COND</th>
<th>Inst.</th>
<th>COND</th>
<th>Inst.</th>
<th>COND</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>1000</td>
<td>bne</td>
<td>1001</td>
<td>be</td>
<td>0001</td>
<td>ble</td>
<td>0010</td>
<td>bvc</td>
<td>1111</td>
</tr>
<tr>
<td>bcc</td>
<td>1101</td>
<td>bcs</td>
<td>0101</td>
<td>bneg</td>
<td>0110</td>
<td>bvc</td>
<td>1111</td>
<td>bvc</td>
<td>1111</td>
</tr>
<tr>
<td>bvs</td>
<td>0111</td>
<td>bne</td>
<td>1001</td>
<td>be</td>
<td>0001</td>
<td>ble</td>
<td>0010</td>
<td>bvc</td>
<td>1111</td>
</tr>
</tbody>
</table>
Fig. 3.12  Example SPARC Code: add two integers

.begin
.org
progl:  ldw [x], %r1 ! load a word from M[x] into register %r1
       ldw [y], %r2 ! load a word from M[y] into register %r2
       addcc %r1, %r2, %r3 !%r3 ← %r1 + %r2 ; set CCs
       st %r3, [z] ! store sum into M[z]
       jmpl %r15, +8, %r0! return to caller
       nop ! branch delay slot

x:  15 ! reserve storage for x, y, and z
y:  9
z:  0

Note different syntax for SPARC.
Note r15 contains return address—placed there by the OS in this case.
Fig. 3.13 Example of Subroutine Linkage in the SPARC

```
.begin
.org

prog:
  ld  [x], %o0               !Pass parameters in
  ld  [y], %o1               ! first 3 output registers.
  call add3                  !Call subroutine to put result in %o0.
  mov -17, %o2               !Set last parameter in delay slot
  st  %o0, [z]               !Store returned result.

x:  15
y:  9
z:  0
add3:
  save %sp,-(16*4),%sp      !Get new window and adjust stack pointer.
  add %i0, %i1, %l0         !Add parameters that now appear in
  add %l0, %i3, %l0         ! input registers using a local.
  ret                       !Return. Short for jmp %i7+8.
  restore %l0, 0, %o0       !Result moved to caller’s %o0.
.end
```

Pipelining of the SPARC Architecture

- Many aspects of the SPARC design are in support of a pipelined implementation
  - Simple addressing modes, simple instructions, delayed branches, load/store architecture
- Simplest form of pipelining is fetch/execute overlap—fetching next inst. while executing current inst.
- Pipelining breaks inst. processing into steps
  - A step of one instruction overlaps different steps for others
- A new inst. is started (issued) before previously issued instructions are complete
- Instructions guaranteed to complete in order
Fig. 3.14  The SPARC MB86900 Pipeline

- 4 pipeline stages are Fetch, Decode, Execute, and Write
- Results are written to registers in Write stage
Pipeline Hazards

- Will be discussed later, but main issue is:
  - Branch or jump change the PC as late as Exec. or Write, but next inst. has already been fetched
    - One solution is ‘Delayed Branch’
    - One (maybe 2) instruction following branch is always executed, regardless of whether branch is taken
    - SPARC has a delayed branch with one ‘delay slot”, but also allows the delay slot instruction to be annulled (have no effect on the machine state) if the branch is not taken
  - Registers to be written by one instruction may be needed by another already in the pipeline, before the update has happened (Data Hazard)
CISC vs. RISC: Recap

- CISCs supply powerful instructions tailored to commonly used operations, stack operations, subroutine linkage, etc.
- RISCs require more instructions to do the same job
- CISC instructions take varying lengths of time
- RISC instructions can all be executed in the same, few cycle, pipeline
- RISCs should be able to finish (nearly) one instruction per clock cycle
Key Concepts: RISC vs. CISC

- While a RISC machine may possibly have fewer instructions than a CISC, the instructions are always simpler. Multi-step arithmetic operations are confined to special units.
- Like all RISCs, the SPARC is a load/store machine. Arithmetic operates only on values in registers.
- A few, regular, instruction formats and limited addressing modes make instruction decode and operand determination fast.
- Branch delays are quite typical of RISC machines and arise from the way a pipeline processes branch instructions.
- The SPARC does not have a load delay, which some RISCs do, and does have register windows, which many RISCs do not.
Chapter Summary

- Machine price/performance are the driving forces.
  - Performance can be measured in many ways: MIPS, execution time, Whetstone, Dhrystone, SPEC benchmarks.
- CISC machines have fewer instructions that do more.
  - Instruction word length may vary widely
  - Addressing modes encourage memory traffic
  - CISC instructions are hard to map onto modern architectures
- RISC machines usually have
  - One word per instruction
  - Load/store memory access
  - Simple instructions and addressing modes
  - Result in allowing higher clock cycles, prefetching, etc.