Instruction Set Architecture

Instructor: Preetam Ghosh
Preetam.ghosh@usm.edu
Language

**HLL** : High Level Language Program written by Programming language like C, C++, Java.

\[
\begin{align*}
\text{Sentence} & \quad a = b + c; \\
& \quad d = a - e;
\end{align*}
\]

**Assembly Language**: The Pneumonic translation of Binary code into English Language. One to one correspondence between Binary Language and Assembly language. MIPS is the Assembly language used in the book.

\[
\begin{align*}
\text{Instruction} & \quad add \ a, b, c \\
& \quad sub \ d, a, e
\end{align*}
\]

**Binary Code**: Language expressed by Binary numbers understood by the computer Hardware.
Instruction Set
Computer Operation

C Language Sentence

\[
\begin{align*}
& a = b + c \\
& d = a - e
\end{align*}
\]

Assembly Instruction (MIPS)

\[
\begin{align*}
& add & a,b,c \\
& sub & d,a,e
\end{align*}
\]

1. Where is a, b, c, d and e
2. Who is doing add and sub
1. Where is a, b, c, d and e
2. Who is doing add and sub

Address
CSC 626/726 Preetam Ghosh
Hardware Software Interface

**Memory**

<table>
<thead>
<tr>
<th>Base Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a[1]</td>
</tr>
<tr>
<td>2</td>
<td>a[2]</td>
</tr>
<tr>
<td>3</td>
<td>a[3]</td>
</tr>
<tr>
<td>4</td>
<td>a[4]</td>
</tr>
<tr>
<td>5</td>
<td>Data b</td>
</tr>
</tbody>
</table>

**Computer Chip**

1. Where is Data
   - In Memory (a[1-4], b) Starting address at **Base Register** for an array
   - In Registers ($s2, $s3,…,$t0) Explicit address
2. Who Transfers Data between Memory and Registers
3. Who is the Controller of ALU

CSC 626/726 Preetam Ghosh
HLL to Hardware

C Language statement


A: Array

h: variable

\[
\begin{array}{c|c}
\end{array}
\]

\[
\begin{array}{c|c}
\$s4 & \text{variable h} \\
\$s3 & \text{Base Register: address of } A[0] \\
$t0 & \\
\end{array}
\]

MIPS Instruction

\[
\begin{align*}
lw & \quad \$s2,32(\$s3) \\
add & \quad \$t0,\$s2,\$s4 \\
sw & \quad \$t0,48(\$s3) \\
\end{align*}
\]

4 Bytes per data (32 bits)

CSC 626/726 Preetam Ghosh
Machine Representation

\[ lw \quad \$t_0 , 32(\$s_3) \]

\[ add \quad \$t_0 , \$s_2 , \$t_0 \]

\[ sw \quad \$t_0 , 48(\$s_3) \]

Machine is in Binary, but this expressions are not in binary. We need Binary Translation. **Machine Language**

<table>
<thead>
<tr>
<th>Op Code (6)</th>
<th>rs (5)</th>
<th>rt (5)</th>
<th>rd (5)</th>
<th>shamt(5)</th>
<th>function(6)</th>
</tr>
</thead>
</table>

**Op Code**: Operation of the instruction

**rs**: The first source operand register

**rt**: The second source operand register

**rd**: Destination operand register. Gets results of the operation

**shamt**: Shift amount for shift instruction

**function**: Selects the functions within a opcode field.

CSC 626/726 Preetam Ghosh
It is difficult to remember this large binary sequence. Hexadecimal conversion is used. It is a number system to the base 16. So 4 bits represent one symbol and there are 16 symbols.

\[ 0_{\text{hex}} = 0000_2 \]
\[ 1_{\text{hex}} = 0001_2 \]
\[ 9_{\text{hex}} = 1001_2 \]
\[ a_{\text{hex}} = 1010_2 \]
\[ f_{\text{hex}} = 1111_2 \]

CSC 626/726 Preetam Ghosh
Instruction Set Architecture

• A computer architect must decide on the set of instructions that are executable in hardware on their designed machine.

• These instructions must:
  – satisfy the design goals of the target machine in terms of cost and performance, and
  – support all the language constructs specified in a high-level or assembly-level language to be run on the target machine.
Instruction Set Architecture

• A good interface:
  – Lasts through many implementations (portability, compatibility)
  – Is used in many different ways (generality)

  [Diagram showing an interface connected to multiple uses and implementations (imp 1, imp 2, imp 3), indicating convenience and efficiency over time.]

  – Provides convenient functionality to higher levels
  – Permits an efficient implementation at lower levels

CSC 626/726 Preetam Ghosh
Architecture Thrust

Some Important Questions:

1. What is the *nature of the programs*
   1. Simplicity
   2. Structured
   3. Integer or floating points

2. High Level [*Language Mapping* (CISC)]
   1. Simplify compilation by easy mapping to instruction
   2. How to reduce code size

3. *Memory* Size
   1. Moore’s Law on memory growth, memory addressing
   2. Memory limitation of embedded system

*Semantic Gap*: Gap between High Level Language and Computer Architecture
Instruction set Architecture

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>_operand-1</th>
<th>_operand-2</th>
<th>_operand-3</th>
</tr>
</thead>
</table>

? : Where are these operands
What types of operands
How to get access to these operands

1. **Stack**
   - All operations on a Stack. Operand Implicit

2. **Accumulator**
   - One operand and Result on Accumulator

3. **Register-Memory**
   - Operand on Register and on Memory

4. **Load-Store**
   - All Operand on explicit GPR

CSC 626/726 Preetam Ghosh
Evolution of Instruction Sets

- Single Accumulator (EDSAC 1950)
- Accumulator + Index Registers (Manchester Mark I, IBM 700 series 1953)
- Separation of Programming Model from Implementation
  - High-level Language Based (B5000 1963)
  - Concept of a Family (IBM 360 1964) Register-Memory
  - General Purpose Register Machines
  - Complex Instruction Sets (CISC) (VAX, Intel 1977-80)

CSC 626/726 Preetam Ghosh
History of STACK Processor

1965: Burroughs B5000 stack processor for programming language ALGOL. TOS, NTOS are only HW Register

1968: Burroughs B6500

1964-70: IBM & DEC’s Arguments
Stack machine performance depends on register speed, memory speed. There are too many copy operations. Intel 80x86 uses stack for floating point operation


CSC 626/726 Preetam Ghosh
Evolution of Instruction Sets

- Unfortunately, there are *no standards to follow in designing an instruction set*.

- The trend up to early 80’s was based on the *CISC (Complex Instruction Set Computers) instruction set design philosophy*:
  - include many instructions in the set,
  - have complex instructions that carry out the job of several simpler instructions (e.g., loop instruction),
  - have many instruction formats and addressing modes,
  - have many different kinds of registers,
  - ...

CSC 626/726 Preetam Ghosh
Evolution of Instruction Sets

• A CISC architecture causes:
  – Size of the machine language programs?
    • simpler (smaller) with less number of instructions to execute.
  – Complexity of the machine architecture?
    • more complex (due to complex instructions), requiring more time for execution of each instruction.
  – Compiler optimizations?
    • more complex (due to many choices in deciding which instructions and/or addressing modes to use).
Evolution of Instruction Sets

• Since the early 80’s the trend for new ISA design has been based on the *RISC (Reduced Instruction Set Computers)* instruction set design philosophy:

  – *Simplicity favors regularity*
    - all instructions are of the same size
    - all instructions of the same type follow the same format
Evolution of Instruction Sets

- RISC ISA design philosophies:
  - Smaller is faster
    - small number of instructions
    - relatively small number of register types
  - Good design demands compromise
    - only a few instruction formats to handle special needs
  - Make the common case fast
    - most often executed instructions or heavily used features need to be optimized
  - Pipelining, single-cycle execution, compiler technology, etc.
### Advantages & Disadvantages of different Instruction Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td>Simple encoding, Operand and result Location fixed</td>
<td>Operand must be in correct order in the Stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operand has to be loaded on stack.</td>
</tr>
<tr>
<td><strong>Accumulator</strong></td>
<td>Simple instruction as only one Operand to be specified</td>
<td>Operation must be in correct order.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Have the correct operand on accumulator.</td>
</tr>
<tr>
<td><strong>Register-Memory</strong></td>
<td>Least number of instruction</td>
<td>Complex instruction set. Decoding is complex.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Potential of variable length instruction.</td>
</tr>
<tr>
<td><strong>Load-Store</strong></td>
<td>Operand on Register, operand can be used without additional instruction.</td>
<td>Larger Instruction and encoding complexity.</td>
</tr>
</tbody>
</table>
Re-look at Instruction set design-RISC

1980s : Ditzel & Patterson : **RISC (Reduced Instruction Set)** Architecture

**Research Outputs**

2. 1980 Berkeley : RISC-I and RISC –II: Patterson’s team, MOS based 32 bit registers. Targeted towards Smalltalk and LISP.
3. 1981 Stanford : MIPS Computer; Hennessy published explanation of RISC advantages over VAX
4. 1986 : HP converted its Minicomputer to RISC (HP Precision Architecture)
5. 1987 : SUN SPRAC based on RISC-II
6. 1990 : IBM RISC RS 6000

CSC 626/726 Preetam Ghosh
Classifying ISAs

- 4 classes:
  - For example: \( C \leftarrow A + B \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C,R1</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop</td>
<td>C</td>
<td></td>
<td>Store C,R3</td>
</tr>
</tbody>
</table>

A typical CISC uses a mix

Also called Register-Register Architecture; e.g.: DLX
Classifying ISAs

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register</td>
<td>Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute (see Ch 3).</td>
<td>Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.</td>
</tr>
<tr>
<td>(0,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-memory</td>
<td>Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.</td>
<td>Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.</td>
</tr>
<tr>
<td>(1,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Most compact. Doesn’t waste registers for temporaries.</td>
<td>Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.</td>
</tr>
<tr>
<td>(3,3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \((m, n)\) means \(m\) memory operands and \(n\) total operands
Interpreting Address

1. **Little Endian**
   (PDP-11, Intel 80x86)

2. **Big Endian**
   (IBM360/370, Motorola)

**Alignment**

Byte Address: \( A \)
Object size: \( s \)
Byte address oriented memory will align if \( Mod[A,s] = 0 \)

**Addressing Mode**

Effective Address
PC – Relative addressing: Mainly used for control transfer
Alignment

**Byte Addressing**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

Memory

Register

**Misalignment complexity**

2. Higher HW access

Memory

Register

1. Lower HW access

Alignment Network

Merging Function

CSC 626/726 Preetam Ghosh
Memory Addressing

• *Addressing Modes*
  – Register
  – Immediate
  – Displacement
  – Register Indirect
  – Indexed
  – Direct or Absolute
  – Memory Indirect
  – Autoincrement
  – Auto decrement
  – *Scaled*
Addressing Modes

- **Register**
  - Value is in a register
    - Add R4, R3
    - $\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$

- **Immediate**
  - Constant value is in the instruction
    - Add R4, #3
    - $\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$

- **Displacement**
  - Relative addressing for access to local variables
    - Add R4, 100(R1)
    - $\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100+\text{Regs}[R1]]$
Addressing Modes

• **Indirect or Register deferred**
  - Address of the operand is in a register
    • Add R4, (R1)
    • Regs[R4] ← Regs[R4] + Mem[Regs[R1]]

• **Indexed**
  - Base + index addressing; useful in array addressing
    • Add R3, (R1+R2)

• **Direct or Absolute**
  - Static addressing for access to local variables
    • Add R1, (1001)
    • Regs[R1] ← Regs[R1] + Mem[1001]
Addressing Modes

• **Memory indirect**
  - The address of the address of the operand is in a register
    • Add R1, @R3
    • Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]

• **Auto-increment or Auto-decrement**
  - Useful for stepping through arrays or accessing stack elements
    • Add R1, (R2)+
      - Regs[R1] ← Regs[R1] + Mem[Regs[R2]]
      - Regs[R2] ← Regs[R2] + d
    • Add R1, -(R2)
      - Regs[R2] ← Regs[R2] - d
      - Regs[R1] ← Regs[R1] + Mem[Regs[R2]]
Addressing Modes

1. Register
   Add R4, R3

2. Immediate
   Add R4, #3

3. Displacement
   Add R4, 100(R1)

4. Register Indirect
   Add R4, (R1)

5. Indexed
   Add R4, (R1+R2)

6. Absolute
   Add R4, (1000)

7. Memory Indirect
   Add R4, @ (R3)

8. Auto Increment
   Add R4, (R2+)
   Reg[R2] ← Reg[R2] + d

9. Auto decrement
   Add R4, -(R2)
   Reg[R2] ← Reg[R2] − d

10. Scaled
    Add R4, 100(R2)[R3]
DSP

**DSP: Digital Signal Processing**

Signal processing requires very high capacity to *handle real time data*.

- *Iterative numeric algorithms*
- Use dot products that require *multiply and accumulate*
- Stringent real time requirements
- Streaming data from A/D converter as infinite stream, results to emit in real time.
- *High data bandwidth*
- *predictable memory access patterns*
- *Predictable program flow, a set of nested loops*
- *Sensitive to small numeric error*

Traditionally these functions were implemented on *ASIC*

*GaAs, InP* Technology gives the speed and power efficiency in ASIC. But these are not at present suitable for Chip design. *Defect rate: 100+/cm²*
DSP replaces ASIC design

- Significant cost of custom ASIC design
  - Custom mask costs millions of dollars
  - Lapse time high
  - No design flexibility

- CMOS Silicon Technology advanced to support
  - High gate count per cell (100K/mm²)
  - Power level going down from 3 volts to almost 1 volt [Power dissipation drops in square law]

- Processor Technology for signal processing started making sense
  - Custom processor architecture
  - Memory on Chip
  - New addressing and complex instruction structure
  - Multi processor architecture

- 1980: NEC mPD7710, ATT DSP1
- 1982: Texas Instrument TMS32010
- 1997: Texas Instrument TMS320C62xx [VLIW, parallelism, RISC like instruction]
# DSP Specialty

1. Need Higher level of accuracy in fixed point arithmetic

<table>
<thead>
<tr>
<th>Generation</th>
<th>Year</th>
<th>Example DSP</th>
<th>Data Width</th>
<th>Accumulator Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1982</td>
<td>TI TMS32010</td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>2</td>
<td>1987</td>
<td>Motorola DSP56001</td>
<td>24 bits</td>
<td>56 bits</td>
</tr>
<tr>
<td>3</td>
<td>1995</td>
<td>Motorola DSP56301</td>
<td>24 bits</td>
<td>56 bits</td>
</tr>
<tr>
<td>4</td>
<td>1997/8</td>
<td>TI TMS320C6201</td>
<td>16 bits</td>
<td>40 bits</td>
</tr>
</tbody>
</table>

CSC 626/726 Preetam Ghosh
Multimedia Media Processor

Class of Embedded Processor dedicated for Multimedia processing
Cost Sensitive
Operates on limited set of data types
  8 bits per pixel (VGA) to high color [16 bits per pixel: R=5, G=6, B=5] to true color [32 bits formats for RGB & A]
Deal with infinite and continuous streams of data
Considerable parallelism on application [MPEG, 3D Graphics, Adobe photographs, audio conferencing]
Very Long Instruction Word (VLIW)
SIMD (Single Instruction Multiple Data streams) Vector Processing
Partitioned ALU for Vector Processing [i860 8, 8 bits, 4 16 bits, 2 32 bits operands]
MMX and SSE for Integer and Floating Point SIMD
New Addressing Modes for DSP & Media Processors

1. **Circular Buffer**

   ![Circular Buffer Diagram]

   ```
   If current = Bottom
   Then
   current = top
   Else
   current = current + 1
   ```

2. **Bit Reverse addressing**

<table>
<thead>
<tr>
<th>Radix 2 FFT data items</th>
<th>0(000)</th>
<th>1(001)</th>
<th>2(010)</th>
<th>3(011)</th>
<th>4(100)</th>
<th>5(101)</th>
<th>6(110)</th>
<th>7(111)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(000)</td>
<td>0(000)</td>
<td>4(100)</td>
<td>2(010)</td>
<td>6(110)</td>
<td>1(001)</td>
<td>5(101)</td>
<td>3(011)</td>
<td>7(111)</td>
</tr>
</tbody>
</table>
   ```

CSC 626/726 Preetam Ghosh
Addressing Modes Decisions

• Implications
  – Need to include displacement and immediate addressing modes in an architecture.
  – A 32-bit instruction (with a 16-bit displacement or immediate field) is sufficient to handle the majority of displacement and immediate values.
    • Will pay a penalty to handle larger values for displacement and immediate, but only use them occasionally.
  • Expect better performance compared to an architecture with a 64-bit (or irregular) instruction length.

Why?
Addressing Modes Decisions

- Displacement values are widely distributed
  - But a 12 bit displacement captures 75% of the full 32-bit displacements and 16 bits captures about 99%
  - Thus, a 16-bit displacement is sufficient.
  - Data from running SPECint92 and SPECfp92 on a MIPS machine.
Addressing Modes Decisions

- Frequency of instruction use affects addressing modes
  - Study of 3 SPEC89 programs on a VAX:

  Need to implement Displ, Imm, and Reg Def
Addressing Modes Decisions

- **Immediate addressing**
  - Very useful in arithmetic, compare and register assignment operations.
  - Data from DLX architecture programs.

![Bar chart showing percentage of operations that use immediates]

- Loads: 10% 45%
- Compares: 77%
- ALU operations: 58% 78%
- All instructions: 10% 35%
Addressing Modes Decisions

- Distribution of immediate values
  - 80% of immediate values fit within 16 bits.
# Usage of address Modes

<table>
<thead>
<tr>
<th>Address Mode</th>
<th>TeX</th>
<th>Spice</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Indirect</td>
<td>1%</td>
<td>6%</td>
<td>1%</td>
</tr>
<tr>
<td>Scaled</td>
<td>0%</td>
<td>16%</td>
<td>6%</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>24%</td>
<td>3%</td>
<td>11%</td>
</tr>
<tr>
<td>Immediate</td>
<td>43%</td>
<td>17%</td>
<td>3%</td>
</tr>
<tr>
<td>Displacement</td>
<td>32%</td>
<td>55%</td>
<td>40%</td>
</tr>
</tbody>
</table>

*Note: Data for VAX machine with SPEC89 programs*

CSC 626/726 Preetam Ghosh
### Addressing mode of DSP

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>% occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>30.02</td>
</tr>
<tr>
<td>Displacement</td>
<td>10.82</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>17.42</td>
</tr>
<tr>
<td>Direct</td>
<td>11.99</td>
</tr>
<tr>
<td>Auto increment(preincrement)</td>
<td>0</td>
</tr>
<tr>
<td>Auto increment(post increment)</td>
<td>18.84</td>
</tr>
<tr>
<td>Auto increment(preincrement on immediate field)</td>
<td>0.77</td>
</tr>
<tr>
<td><strong>Auto increment(Circular Buffer)</strong></td>
<td><strong>0.08</strong></td>
</tr>
<tr>
<td>Auto increment(post increment by immediate field)</td>
<td>0</td>
</tr>
<tr>
<td>Auto increment(increment by R0 content)</td>
<td>1.54</td>
</tr>
<tr>
<td>Auto increment (increment by R0 on circular buffer)</td>
<td>2.15</td>
</tr>
<tr>
<td><strong>Auto increment (by R0 with bit reversing)</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td>Auto decrement (post decrement)</td>
<td>6.08</td>
</tr>
<tr>
<td><strong>Auto decrement (post decrement on circular buffer)</strong></td>
<td><strong>0.04</strong></td>
</tr>
<tr>
<td>Auto decrement post decrement by R0 contents)</td>
<td>0.16</td>
</tr>
<tr>
<td><strong>Auto decrement (post with R0 on circular buffer)</strong></td>
<td><strong>0.08</strong></td>
</tr>
<tr>
<td><strong>Auto decrement (post with R0 on bit reversing)</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

Note: Data for 54 DSP routines of C library programs in TI TMS320C54xDSP
Operands

- Integer
- Two’s Complement
- Single Precision floating point, Double Precision floating point, IEEE 754 floating point
- Character: (8 bits)
- Binary
- Binary coded decimal or packed decimal
- 3D type data
  - Vertex (x,y,z,w) four components
  - triangle (3-vertices)
  - pixels 32 bits (RGB and A)
- DSP
  - fixed point

CSC 626/726 Preetam Ghosh
Numbering Systems

1. **Signed Magnitude**: High order bit is sign bit and (n-1) bits are magnitude bits.
   
   $-3 : 1011$

2. **Two’s complement**: A number and its negative adds to $2^n$.
   
   $-3 : 1101$.

   Two’s complement of $a_{n-1}a_{n-2}....a_1a_0$

   \[ a_{n-1}a_{n-2}....a_1a_0 = -a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + ... + a_1 \cdot 2^1 + a_0 \]

3. **One’s Complement**: The negative of the number is obtained by complementing each bit.
   
   $-3 : 1100$. 

CSC 626/726 Preetam Ghosh
Floating Point Number

IEEE 754-1985 Floating Point Standard
A computer word is divided into two parts

*Exponent (e)*
*Fraction field (f)*

\[
\text{Significant} = 1 + \text{fraction} \\
\text{Number} = \text{Significant}. 2^{\text{exponent}}
\]

In IEEE standard of Single precision, Exponent is 8 bit field. The range of Exponent \(E_{\text{max}} = 127\) and \(E_{\text{min}} = -126\). Exponent bias = 127
No of bits in fraction field = 23
Sign bit= 1.
Number = \((1 + \text{fraction}).2^{\text{exponent} - 127}\)
Packed decimal Number

Packed decimal or Binary coded decimal
4-bits are used to code 0 … 9 and two decimal digits are packed into each byte.
Numeric character string is called unpacked decimal
Operations packing and unpacking are provided to convert string to decimal number.
One reason to provide the decimal number is to get the result exactly matching the decimal numbers. This is required for Financial transactions.
Fixed Point

DSP use Fixed point arithmetic

Integer has binary point right to the least significant bit, fixed point has binary point just right to the sign bit.

Fixed point data thus are between $-1$ and $+1$.

Example

0100 0000 0000 0000
0000 1000 0000 0000
0100 1000 0000 1000

2’s Complement:

\[2^{14} \]
\[2^{11} \]
\[(2^{14} + 2^{11} + 2^3)\]

Fixed point:

\[2^{-1} \]
\[2^{-4} \]
\[(2^{-1} + 2^{-4} + 2^{-12})\]
Fixed Point

**DSP use Fixed point arithmetic**

Fixed point is a low cost floating point that shares an Exponent between multiple Fixed point variables. It is often called blocked floating point.

\[
\text{<Fraction>}.2^{\text{exponent}}
\]

Variable \( A_i \)
Variable \( A_j \)
Variable \( A_k \)

**Fixed Point Variable**

Variable \( B \)

Exponent variable shared by many fixed variable.
Programmer manually aligns the exponent and fraction. DSP provides extra long registers to avoid overflow in these operations
Instruction Types

1. ALU  Integer Arithmetic & logical operations
2. Data Transfer  Load & Store, move
3. Control  Branch, Jump, Procedure Call, return, traps
4. System  OS and VM management instructions
5. Floating Point  Add, multiply, divide, compare
6. Decimal  Add, multiply, Decimal to character conversion
7. String  move, compare, search
Instruction Set Operations

- Typical machine instructions needed include:
  - Data transfer (reg-reg, reg-memory, memory-reg, ...)
  - Arithmetic (integer/floating point: add, subtract, multiply, ...)
  - Logic and non-numeric (Boolean, bit manipulation, string operations, etc.)
  - Program control (branches, jumps, PC manipulation, proc calls, ...)
  - I/O operations
  - System operations (OS calls, memory mngmnt, ...)

CSC 626/726 Preetam Ghosh
Instruction Set Operations

- Top 10 instructions for Intel 80x86, using SPECint92:

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 instruction</th>
<th>Integer average (% total executed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>96%</td>
</tr>
</tbody>
</table>
# Integer vs. Floating Point Immediate Operation

**Addressing Mode: Immediate**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>22%</td>
<td>23%</td>
</tr>
<tr>
<td>ALU</td>
<td>19%</td>
<td>25%</td>
</tr>
<tr>
<td>All Operations</td>
<td>16%</td>
<td>21%</td>
</tr>
</tbody>
</table>

*Note: Data from Alpha Processor with SPEC2000, with full optimization*

CSC 626/726 Preetam Ghosh
## Data Access Distribution

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Ward</td>
<td>59%</td>
<td>70%</td>
</tr>
<tr>
<td>Word</td>
<td>26%</td>
<td>29%</td>
</tr>
<tr>
<td>Half Word</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Byte</td>
<td>10%</td>
<td>1%</td>
</tr>
</tbody>
</table>

*Note: SPEC2000 programs on VAX*
## Operations in 80x86

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Conditional Branch</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Compare</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>And</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Sub</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>Call</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Rest</td>
<td>4</td>
</tr>
</tbody>
</table>
# Operations in DSP

<table>
<thead>
<tr>
<th>Instruction</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Mem</td>
<td>32.2 + 2.5</td>
</tr>
<tr>
<td>Load Mem</td>
<td>9.4 + 2.0</td>
</tr>
<tr>
<td>Add Mem</td>
<td>6.8 + 1.3</td>
</tr>
<tr>
<td>Call</td>
<td>5.0</td>
</tr>
<tr>
<td>Push Mem</td>
<td>5.0</td>
</tr>
<tr>
<td>Subtract Mem</td>
<td>4.9 + 0.9</td>
</tr>
<tr>
<td>Multiple Accumulate Mem</td>
<td>4.6</td>
</tr>
<tr>
<td>Move Mem – mem</td>
<td>4.0</td>
</tr>
<tr>
<td>Change Status</td>
<td>3.7</td>
</tr>
<tr>
<td>Pop Mem</td>
<td>2.8</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>2.6</td>
</tr>
<tr>
<td>Return</td>
<td>2.5</td>
</tr>
<tr>
<td>Branch</td>
<td>2.0</td>
</tr>
<tr>
<td>Repeat</td>
<td>2.0</td>
</tr>
<tr>
<td>Multiply</td>
<td>1.8</td>
</tr>
<tr>
<td>NOP</td>
<td>1.5</td>
</tr>
</tbody>
</table>

*Note: TI’s TMS320C540X. Larger number of stores due to (1) writing 40 bit accumulator contents to 16 bit words and also, transfer between registers. The Index registers of TI DSP have memory address.*
# SIMD in Multi Media Processors

<table>
<thead>
<tr>
<th>Instruction Cat.</th>
<th>Alpha</th>
<th>HP</th>
<th>Intel</th>
<th>PowerPC SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Subtract</td>
<td>X</td>
<td>4H</td>
<td>8B,4H,2W</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td>Saturate(+/-)</td>
<td>X</td>
<td>4H</td>
<td>8B,4H</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td>Multiply</td>
<td>X</td>
<td>X</td>
<td>4H</td>
<td>16B,8H</td>
</tr>
<tr>
<td>Compare 8B</td>
<td>X</td>
<td>8B,4H,2W</td>
<td>16B,8H,4W</td>
<td>4H,2W</td>
</tr>
<tr>
<td></td>
<td>(&gt;=)</td>
<td>(=,&gt;)</td>
<td>(&gt;=,=,&lt;,&lt;=)</td>
<td>(=,not=,&gt;,&lt;,&lt;=)</td>
</tr>
<tr>
<td>Shift Reg left</td>
<td>X</td>
<td>4H</td>
<td>4H,2W</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td>Shift right(Ari.)</td>
<td>X</td>
<td>4H</td>
<td>X</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td>Shift &amp; add (Sat)</td>
<td>X</td>
<td>4H</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Multiply &amp; add</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>And/or/Xor</td>
<td>8B,4H</td>
<td>8B,4H</td>
<td>8B,4H,2W</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td></td>
<td>2W</td>
<td>2W</td>
<td></td>
<td>8B,4H,2W</td>
</tr>
<tr>
<td>Absolute Diff</td>
<td>8B</td>
<td>X</td>
<td>X</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td>Max/Min</td>
<td>8B,4W</td>
<td>X</td>
<td>X</td>
<td>16B,8H,4W</td>
</tr>
<tr>
<td></td>
<td>4H:4B</td>
<td>8B</td>
<td>2W:2H</td>
<td>8H:8B</td>
</tr>
<tr>
<td>Unpack</td>
<td>2B:2W</td>
<td>X</td>
<td>2B:2W</td>
<td>4B:4W</td>
</tr>
<tr>
<td></td>
<td>4B:4H</td>
<td>4B:4H</td>
<td>4B:4H</td>
<td>4B:4H</td>
</tr>
<tr>
<td>Permute/shuffle</td>
<td>X</td>
<td>4H</td>
<td>X</td>
<td>16B,8H,4W</td>
</tr>
</tbody>
</table>

CSC 626/726 Preetam Ghosh
SIMD in Multi Media Processors (Cond)

Alpha: Alpha MAX  
HP: HP PA-RISC MAX2  
Intel: Pentium MMX  
Power PC: AltiVec  
SPARC VIS  
B: Bytes (8 bits)  
H: half Word (16 bits)  
W: Word (32 bits)  
ALU: 64 or 128 bits  

*Saturation Arithmetic*: Ignore carry with overflow. Work on the closest number to the overflowed number. This is required for real time data

2*2W means two operand each with two words
Control Flow Instructions

Standard Techniques
1. Jump
2. Branch
3. Procedure Call
4. Procedure returns

PC-relative
Simple, only displacement bits are needed
Position Independence
Fewer bits are required, as jumps normally close to PC

Register indirect jumps
When target is not known at compile time
Case or Switch statements
Virtual functions or Methods
Function Pointers
Dynamically shared Library

CSC 626/726 Preetam Ghosh
# Branch Condition Evaluation

**Condition Code (CC)**  
**Condition Register (CR)**  
**Compare and Branch (CB)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Test</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>80X86, SPARC</td>
<td>Test special bits set by ALU, possibly by program</td>
<td>Some time condition is set free</td>
<td>CC is extra state</td>
</tr>
<tr>
<td></td>
<td>PowerPC</td>
<td></td>
<td></td>
<td>Constrains the inst ordering</td>
</tr>
<tr>
<td>CR</td>
<td>Alpha, MIPS</td>
<td>Test arbitrary registers with result of a comparison</td>
<td>Simple</td>
<td>Uses up registers</td>
</tr>
<tr>
<td>CB</td>
<td>VAX, PA-RISC</td>
<td>Compare is part of branch. Often compare is limited subset.</td>
<td>One instruction rather than two</td>
<td>Too much work for pipeline</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For a branch</td>
<td>architecture</td>
</tr>
</tbody>
</table>
## Conditional Branches in Application

<table>
<thead>
<tr>
<th>Condition</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Equal</td>
<td>2%</td>
<td>5%</td>
</tr>
<tr>
<td>Equal</td>
<td>18%</td>
<td>16%</td>
</tr>
<tr>
<td>Greater than or Equal</td>
<td>11%</td>
<td>0%</td>
</tr>
<tr>
<td>Greater Than</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>33%</td>
<td>44%</td>
</tr>
<tr>
<td>Less Than</td>
<td>35%</td>
<td>34%</td>
</tr>
</tbody>
</table>
Register Saving during Procedure Call

1. **Caller Saving**: Calling procedure must save registers that it must have after completion of the procedure call.

2. **Callee Saving**: The called procedure will save the registers and restore them back when the control is returned.

---

**ABI**: Application Binary Interface
Instruction Encoding

1. **Opcode**

2. **Instruction format**

*variable* : All addressing mode is included in the field (Intel 80X86)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Add.-1 Specifier</th>
<th>Add.-1 field</th>
<th>Add.-n Specifier</th>
<th>Add.-n field</th>
</tr>
</thead>
</table>

*fixed*: Addressing mode is specified in Opcode (MIPS, SPARC)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address field.-1</th>
<th>Address field.-2</th>
<th>Address field.-3</th>
</tr>
</thead>
</table>

*Hybrid*: Example IBM 360/70

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address specifier</th>
<th>Address specifier</th>
<th>Address field.-1</th>
<th>Address field.-2</th>
</tr>
</thead>
</table>

CSC 626/726 Preetam Ghosh
MIPS64 Processor Architecture Definition

1. RISC Processor
2. 32 64 bit General Purpose Integer Register [R0...... R31]
3. 32 Floating Point Registers [F0...... F31] holds 32 Single precision or 32 double precision values
4. Value of R0 is always 0
5. Data types 8-bit bytes, 16 bit half words, 32 bits words, 64 bit double words for integer, 32 bit single precision and 64 bit double precision floating point
6. Operations on 64 bit Integers
7. Addressing modes: Immediate and displacement with 16 bit fields
8. Byte addressed memory with 64 bit address field
9. Mode bit to select either Big Endian or Little Endian
10. Memory Access to GPR can be byte, half word, word and double word.
11. All memory access is aligned
12. All instructions are 32 bits with 6 bit opcode
13. Instruction type: I-type, R-Type and J-Type
RISC Processors

Desk top or Server Processors
Digital Alpha
HP PA-RISC
IBM & Motorola Power PC
Silicon Graphics MIPS
Sun Microsystems SPARC

Embedded System
Advanced RISC Machine ARM
Advanced RISC Machine Thumb
Hitachi SuperH
Mitsubishi M32R
Silicon Graphics MIPS16
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Alpha</th>
<th>MIPS I</th>
<th>PA-RISC 1.1</th>
<th>PowerPC</th>
<th>SPARC v.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction size (bits)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Address space (size, model)</td>
<td>64 bits, flat</td>
<td>32 bits, flat</td>
<td>48 bits, segmented</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Unaligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Protection</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
<td>Page</td>
</tr>
<tr>
<td>Minimum page size</td>
<td>8 KB</td>
<td>4 KB</td>
<td>4 KB</td>
<td>4 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>I/O</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
<tr>
<td>Integer registers (number, model, size)</td>
<td>31 GPR, 64 bits</td>
<td>31 GPR, 32 bits</td>
<td>31 GPR, 32 bits</td>
<td>32 GPR, 32 bits</td>
<td>31 GPR, 32 bits</td>
</tr>
<tr>
<td>Separate floating-point registers</td>
<td>31 × 32 or 31 × 64 bits</td>
<td>16 × 32 or 16 × 64 bits</td>
<td>56 × 32 or 28 × 64 bits</td>
<td>32 × 32 or 32 × 64 bits</td>
<td>32 × 32 or 32 × 64 bits</td>
</tr>
<tr>
<td>Floating-point format</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
<td>IEEE 754 single, double</td>
</tr>
</tbody>
</table>

**Figure C.1** Summary of the first version of five recent architectures for desktops and servers. Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with [Figure C.34](#). Later versions of these architectures all support a flat, 64-bit address space.
Comparison

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>Thumb</th>
<th>SuperH</th>
<th>M32R</th>
<th>MIPS16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction size (bits)</td>
<td>32</td>
<td>16</td>
<td>16</td>
<td>16/32</td>
<td>16/32</td>
</tr>
<tr>
<td>Address space (size, model)</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
<td>32 bits, flat</td>
<td>32/64 bits, flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Integer registers</td>
<td>15 GPR x 32 bits</td>
<td>8 GPR + SP, LR x 32 bits</td>
<td>16 GPR x 32 bits</td>
<td>16 GPR x 32 bits</td>
<td>8 GPR + SP, RA x 32/64 bits</td>
</tr>
<tr>
<td>I/O</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>

**Figure C.2** Summary of five recent architectures for embedded applications. Except for number of data addressing modes and some instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure C.34.