Floating Point Addition - Special Cases

- overflow: when the result has an exponent that is too large in the positive direction, meaning that the magnitude of the entire number (whether the number is positive or negative) is too far away from zero to be represented.
  - represent as infinity, also for division by zero

- invalid result – NaN
  - special cases, like 0/0 or ∞*0 or sqrt(-1)

- both can propagate during computation
  - no exception (like integer division by zero), no silent error (like integer overflow)
Special Cases

- Underflow: when the result has an exponent that is too large in the negative direction, meaning that magnitude of the entire number (whether the number is positive or negative) lies between zero and the smallest fraction we can actually represent.
Subnormal Representation

- actual range of exponent: -2...4
- smallest positive number
  - $1.0000_{\text{two}} \times 2^{-2} = 0.25$
- smallest gap between adjacent numbers
  - $0.0001_{\text{two}} \times 2^{-2} = 0.015625$
- can be represented as
  - $1.0001_{\text{two}} \times 2^{-3} = 0.0001_{\text{two}} \times 2^{-2}$
Floating Point

- consequences:
  - e.g., distributive law not guaranteed!
  - in general: pay attention to order of calculations!
Higher Precision

- mostly used to increase size of fraction
- length of fields in IEEE 754:

<table>
<thead>
<tr>
<th>format</th>
<th>exponent (# of bits)</th>
<th>fraction (# of bits)</th>
<th>bias (decimal amt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>8</td>
<td>23</td>
<td>127</td>
</tr>
<tr>
<td>double</td>
<td>11</td>
<td>52</td>
<td>1023</td>
</tr>
<tr>
<td>quadruple</td>
<td>15</td>
<td>112</td>
<td>16383</td>
</tr>
</tbody>
</table>
CS 230 – Introduction to Computers and Computer Systems

Module 2 – Assembly Language
Overview

- assembly language: MIPS
  - arithmetic operations
  - data movement
  - conditional execution
  - subroutines
Machine Code

- binary code – comprised of 0s and 1s
- “direct” execution by processor
- program (bits) grouped into instructions
  - fixed vs. variable length
  - operation code (opcode) + operands
  - instructions control processor
    - opcode designates operation
    - operands designate data
Assembly Language

- human-readable “programming language”
  - very simple compared to, e.g., Scheme, Python
- almost direct mapping to machine code, except:
  - labels
  - data declaration
  - pseudo instructions
- compilation by assembler program
  - into machine code
Instruction Set

- repertoire of instructions
- different processors have different sets – x86, ARMv7, MIPS
  - many commonalities
- depending on use cases
  - scalar processor – one data item at a time
    - typical workstation/laptop/etc. CPU
  - vector processor – array of data (vector)
    - e.g., graphics processor (GPU)
  - hybrids...
Turing Completeness

- well-defined theoretical concept
- fundamental capabilities of instruction set
- minimum requirements
  - while or if/goto & change memory locations
- typical programming languages / instruction sets are Turing complete
- ... vs. data description languages, e.g. XML
MIPS Architecture

- MIPS: _Microprocessor without _Interlocked _Pipeline _Stages
  - interlocking?
  - pipeline?
  - details later (Module 3)

- multiple revisions, systems, and compilers

=> not just a single standard
MIPS Architecture

- MIPS: Microprocessor without Interlocked Pipeline Stages
  - interlocking?
  - pipeline?
  - details later (Module 3)

- multiple revisions, systems, and compilers

=> not just a single standard
MIPS Assembly Language

- each instruction takes 32 bit = 4 bytes = *word*
- arithmetic instructions operate on registers
  - separate memory load and store
- 32 registers available, 32 bit wide
- instructions have up to 3 operands:
  - 1\(^{st}\) operand is destination
  - 2\(^{nd}\) and 3\(^{rd}\) are sources
  - same register can be source and destination
MIPS Arithmetic

• All MIPS arithmetic instructions have 3 operands
• Operand order is fixed (e.g., destination first)

Example:

C code: \( A = B + C \)
MIPS code: `add $s0, $s1, $s2`
MIPS Arithmetic

C code: \[ A = B + C + D + E; \]

MIPS code
\[
\text{(arithmetic):}\quad \text{add } \$s0, \$s1, \$s2 \\
\text{sub } \$s0, \$s0, \$s4}
\]

Trade-off: Fast individual instructions with a small fixed length instruction set but high-level code translates to denser machine code.
Model of a Computer

![Diagram of a computer model with input device, central processing unit, memory unit, and output device.]
MIPS Registers

• *Operands must be in registers* – only 32 registers provided (which require 5 bits to select one register). Reason for small number of registers: *Electronic signals have to travel further on a physically larger chip increasing clock cycle time.*

• For now, we will use $s0, s1, \ldots$ for registers that correspond to variables in higher level languages (C, Java) and $t0, t1, \ldots$ for temporary registers needed to compile the program into MIPS instructions.
MIPS Arithmetic

C Code: \( f = (g + h) - (i + j); \)

- The variables \( f, g, h, i, \) and \( j \) are assigned to the registers \( s0, s1, s2, s3, \) and \( s4, \) respectively. What is the compiled MIPS code?

add $t0,$s1,$s2  # register $t0 contains \( g + h \)
add $t1,$s3,$s4  # register $t1 contains \( i + j \)
sub $s0,$t0,$t1  # \( f \) gets \( $t0 - $t1 \), which is \( (g + h) - (i + j) \)
Memory Organization

- Registers get the data by loading from memory and set the data by storing to memory.
- Viewed as a large single-dimension array with access by address.
- A memory address is an index into the memory array.
- *Byte addressing*: the index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte.

```
0 | 8 bits of data
1 | 8 bits of data
2 | 8 bits of data
3 | 8 bits of data
4 | 8 bits of data
5 | 8 bits of data
6 | 8 bits of data
...```

MIPS Memory Organization

Bytes are load/store units, but most data items use larger *words*

For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>Address (in bytes)</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Registers correspondingly hold 32 bits of data

$2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$

$2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$

i.e., words are *aligned*

*Big-Endian*
Load/Store Instructions


- A is an array and the variable h associates with the register \( s2 \). The starting address, or base address, of the array is in \( s3 \). What is the MIPS code?

MIPS code

(load):
\[
\text{lw } \$t0, 32(\$s3)
\]

(arithmetic):
\[
\text{add } \$t0, \$s2, \$t0
\]

(store):
\[
\text{sw } \$t0, 32(\$s3)
\]

Load word has destination first, store has destination last.

- Remember MIPS arithmetic operands are registers, not memory locations. Therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory.