Single Precision Floating Point IEEE- 754

• First write the number in binary.

• Convert it to standard form (eg. 10110.001 = 1.0110001 x $2^4$). The power of 2’s the exponent. The fractional part’s the mantissa.

• Add the exponent to the bias (127 for 32 bit, 1023 for 64 bit) and convert the sum to binary.

• Write down the number as sign bit, exponent, mantissa. Fill out the remaining bits with 0’s.

• Split the number is groups of 4. For each of the 4 bits, write the hexadecimal equivalent.
Single Precision Floating Point IEEE- 754

• 1.125

\[
1.125_{10} = 1.001_2 = 1.001 \times 2^0 \\
\text{Exponent - } 127 + 0 = 127_{10} = 01111111_2 \\
\text{Mantissa} - 001
\]

\[
\begin{array}{cccccccccccccccc}
3 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 3 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

\[
= 0x \ 3F900000
\]
• 6.53125

\[6.53125_{10} = 110.10001_{2} = 1.1010001 \times 2^{2}\]

Exponent - 127 + 2 = 129_{10} = 10000001_{2}
Mantissa – 1010001

\[
\begin{array}{cccccccccccccccccccccccc}
    & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

= 0x 40D10000
Single Precision Floating Point IEEE- 754

-5.546875

\[
5.546875_{10} = 101.100011_2 = 1.01100011 \times 2^2
\]

Exponent - 127 + 2 = 129_{10} = 10000001_2

Mantissa – 01100011

\[
\begin{array}{cccccccc}
3 & 2 & 2 & 1 & 9 & 1 & 5 & 1 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}
\]

\[
\begin{array}{cccccccc}
 & 2 & 3 & 1 & 9 & 1 & 5 & 1 \\
\hline
7 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
3 & 7 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[= 0x \text{COB1800}\]
Range of Single Precision Float (Positive)

- **Largest Number:**
  
  
  \( (1 + 1 - 2^{-23}) \times (2^{254-127}) \)

  \[ = 2^{128} - 2^{104} \]

  \[ = 340282346638528859811704183484516925440 \]

  \[ \approx 3.4028235 \times 10^{38} \]

- **Smallest Number:**

  
  \( (1 + 0.0) \times (2^{1-127}) \)

  \[ = 2^{-126} \]

  \[ \approx 1.175494351 \times 10^{-38} \]
MIPS Instruction Review: ADD

• add d, s1, s2
• Example: add $t0, $t1, $t2
  Destination is $t0, Sources are $t1 and $t2
• Logic:
  – Bitwise addition with carries
  – $0 + 0 = 0$
  – $0 + 1 = 1$
  – $1 + 0 = 1$
  – $1 + 1 = 10$. Sum is 0, carry 1
MIPS Instruction Review: ADDI

• addi d, s, immediate
• Example: addi $t0, $t1, 10
  Destination is $t0, Sources are $t1 and an immediate signed short number (-32768 - +32767)
• Logic: same as ADD
MIPS Instruction Review: SUB

- sub d, s1, s2
- Example: sub $t0, $t1, $t2
  Destination is $t0, Sources are $t1 and $t2
- Logic: Bitwise subtraction with borrows
  - $0 - 0 = 0$
  - $1 - 1 = 0$
  - $1 - 0 = 1$
  - $0 - 1 = 1$ and remove 1 from the next digit
    – Actually, s2’s Two's Compliment is added to s1
MIPS Instruction Review: AND, ANDi

• and d, s1, s2
• Example: and $t0, $t1, $t2
  Destination is $t0, Sources are $t1 and $t2
• Logic:
  – Bitwise
  – 0 & 0 = 0
  – 0 & 1 = 0
  – 1 & 0 = 0
  – 1 & 1 = 1
• andi performs AND operation with an immediate signed short operand. Syntax of ADDi, operation of AND
MIPS Instruction Review: OR, ORi

- `or d, s1, s2`
- Example: `or $t0, $t1, $t2`
  Destination is $t0, Sources are $t1 and $t2
- Logic:
  - Bitwise
  - $0 \mid 0 = 0$
  - $0 \mid 1 = 1$
  - $1 \mid 0 = 1$
  - $1 \mid 1 = 1$
- ori performs OR operation with an immediate operand. Syntax of ADDi, operation of OR
MIPS Instruction Review: XOR, XORi

• xor d, s1, s2
• Example: xor $t0, $t1, $t2
  Destination is $t0, Sources are $t1 and $t2
• Logic:
  — Bitwise
  — 0 ⊕ 0 = 0
  — 0 ⊕ 1 = 1
  — 1 ⊕ 0 = 1
  — 1 ⊕ 1 = 0
• xor performs XOR operation with an immediate operand. Syntax of ADDi, operation of XOR
MIPS Instruction Review: NOR

• nor d, s1, s2
• Example: nor $t0, $t1, $t2
  Destination is $t0, Sources are $t1 and $t2
• Logic:
  – Bitwise
  – $0 \downarrow 0 = 1$
  – $0 \downarrow 1 = 0$
  – $1 \downarrow 0 = 0$
  – $1 \downarrow 1 = 0$
MIPS Instruction Review: LW

- lw d, immediate(pointer)
- Example: lw $t0, 12($t1)
  Destination is $t0, Source Address is $t1 + immediate signed short offset (a multiple of 4)
- Logic:
  - Fetches value at an address in memory and loads it into a register.
MIPS Instruction Review: SW

• sw d, immediate(pointer)
• Example: sw $t0, 12($t1)
  Source is $t0, Destination Address is $t1 + immediate signed short offset (a multiple of 4)
• Logic:
  – Fetches value within a register and stores it in a memory address.
MIPS Instruction Review: SLL

• `sll d, s, immediate`
• `sll $t0, $t1, 2`
  Destination is $t0, Source is $t1, immediate is number of bits to shift (0 to 32)
• Logic:
  • Shifts the bits of number in source register left by the number of bits specified. 0’s are shifted in. Result is stored in destination register.
MIPS Instruction Review: SRL

• srl d, s, immediate
• srl $t0, $t1, 2
  Destination is $t0, Source is $t1, immediate is number of bits to shift (0 to 32)
• Logic:
  • Shifts the bits of number in source register right by the number of bits specified. 0’s are shifted in. Result is stored in destination register.
Convert C code to MIPS

• \$t0 = A[\$t2];
• A[\$t2] = \$t0 & \$t1;
• \$t0 = (A[\$t1] + \$t2) / 2;

• The starting address of array A is in \$s0, and if \$t2 = 4, A[\$t2] represents the 4th element in A. Array elements are numbered from 0.

• Don’t modify the contents of registers unless the C code specifically states to.
\$t0 = A[\$t2];

- sll \$t4, \$t2, 2
- add \$t4, \$t4, \$s0
- lw \$t0, 0(\$t4)

- MIPS is word addressed. Memory is byte addressed. So, we need to multiply MIPS addresses by 4 to get to the memory address. Hence sll.
A[$t2] = $t0 & $t1;

• and $t5, $t0, $t1
• sw $t5, 0($t4)

• We already have A[$t2] in $t4. So, we need not recalculate the address.
$t0 = (A[\$t1] + \$t2) / 2$

• sll $\$t6, $\$t1, 2
• add $\$t6, $\$t6, $\$s0
• lw $\$t7, 0($\$t6)
• add $\$t7, $\$t7, $\$t2
• srl $\$t0, $\$t7, 1

• Right shifting by 1 bit is the same as dividing by 2. Left shifting by 1 bit is the same as multiplying by 2.