15-740

Computer Arithmetic
A Programmer's View
Oct. 6, 1998

Topics

- **Integer Arithmetic**
  - Unsigned
  - Two's Complement
- **Floating Point**
  - IEEE Floating Point Standard
  - Alpha floating point

Notation

W: Number of Bits in “Word”

C Data Type	Sun, etc.	Alpha
`long int`	32	64
`int`	32	32
`short`	16	16
`char`	8	8

Integers

- **Lower case**
- **E.g.,** x**,** y**,** z

Bit Vectors

- **Upper Case**
- **E.g.,** X**,** Y**,** Z
- **Write individual bits as integers with value 0 or 1**
- **E.g.,** $X = x_{w-1}, x_{w-2}, \ldots x_0$
  - Most significant bit on left

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \; 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \; 2^{w-1} + \sum_{i=0}^{w-2} x_i \; 2^i$$

Sign Bit

```
short int x =  15740;
short int y = -15740;
```

- **C `short` 2 bytes long**

	Decimal	Hex	Binary
x	15740	3D 7C	00111101 01111100
y	-15740	C2 84	11000010 10000100

Sign Bit

- **For 2's complement, most significant bit indicates sign**
  - 0 for nonnegative
  - 1 for negative

Numeric Ranges

Unsigned Values

- *UMin* = 0

  000…0
- *UMax* **=** $2^w - 1$

  111…1

Two's Complement Values

- *TMin* = -2^{w-1}

  100…0
- *TMax* **=** $2^{w-1} - 1$

  011…1

Other Values

- Minus 1

  111…1

Values for *W* = 16

	Decimal	Hex	Binary
`UMax`	65535	`FF FF`	`11111111 11111111`
`TMax`	32767	`7F FF`	`01111111 11111111`
`TMin`	-32768	`80 00`	`10000000 00000000`
`-1`	-1	`FF FF`	`11111111 11111111`
`0`	0	`00 00`	`00000000 00000000`

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	**255**	**65,535**	**4,294,967,295**	**18,446,744,073,709,551,615**
TMax	**127**	**32,767**	**2,147,483,647**	**9,223,372,036,854,775,807**
TMin	**-128**	**-32,768**	**-2,147,483,648**	**-9,223,372,036,854,775,808**

Observations

- |*TMin* | = *TMax* + 1
  - **Asymmetric range**
- *UMax* = 2 * *TMax* + 1

C Programming

- **`#include <limits.h>`**
  - K&R Appendix B11
- **Declares constants, e.g.,**
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- **Values platform-specific**

Unsigned & Signed Numeric Values

X	**B2U(*X*)**	**B2T(*X*)**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

class07.ppt

Example Values

- **$W = 4$**

Equivalence

- **Same encodings for nonnegative values**

Uniqueness

- **Every bit pattern represents unique integer value**
- **Each representable integer has unique bit encoding**

Can Invert Mappings

- **U2B(x) = B2U^{-1}(x)**
  - Bit pattern for unsigned integer
- **T2B(x) = B2T^{-1}(x)**
  - Bit pattern for two’s comp integer

Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

```
short int          x =  15740;
unsigned short int ux = (unsigned short) x;
short int          y  = -15740;
unsigned short int uy = (unsigned short) y;
```

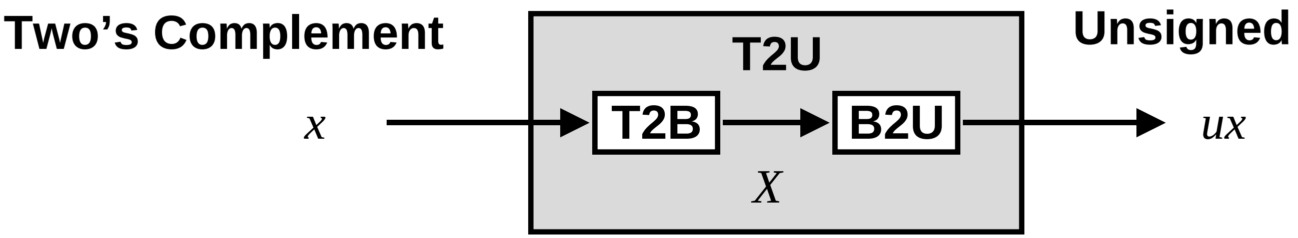
Resulting Value

- **No change in bit representation**
- **Nonnegative values unchanged**

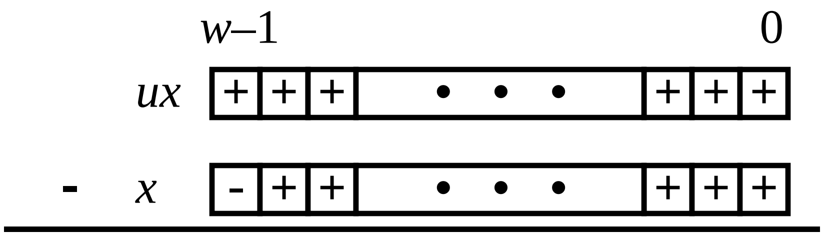
  ux = 15740
- **Negative values change into (large) positive values**

  uy = 49796

Relation Between 2’s Comp. & Unsigned



Maintain Same Bit Pattern



$$+2^{w-1} - -2^{w-1} = 2*2^{w-1} = 2^{w}$$

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^{w} & x < 0 \end{cases}$$

Signed vs. Unsigned in C

Constants

- **By default are considered to be signed integers**
- **Unsigned if have "U" as suffix**

```
0U, 4294967259U
```

Casting

- **Explicit casting between signed & unsigned same as U2T and T2U**

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

- **Implicit casting also occurs via assignments and procedure calls**

```
tx = ux;
uy = ty;
```

Casting Surprises

Expression Evaluation

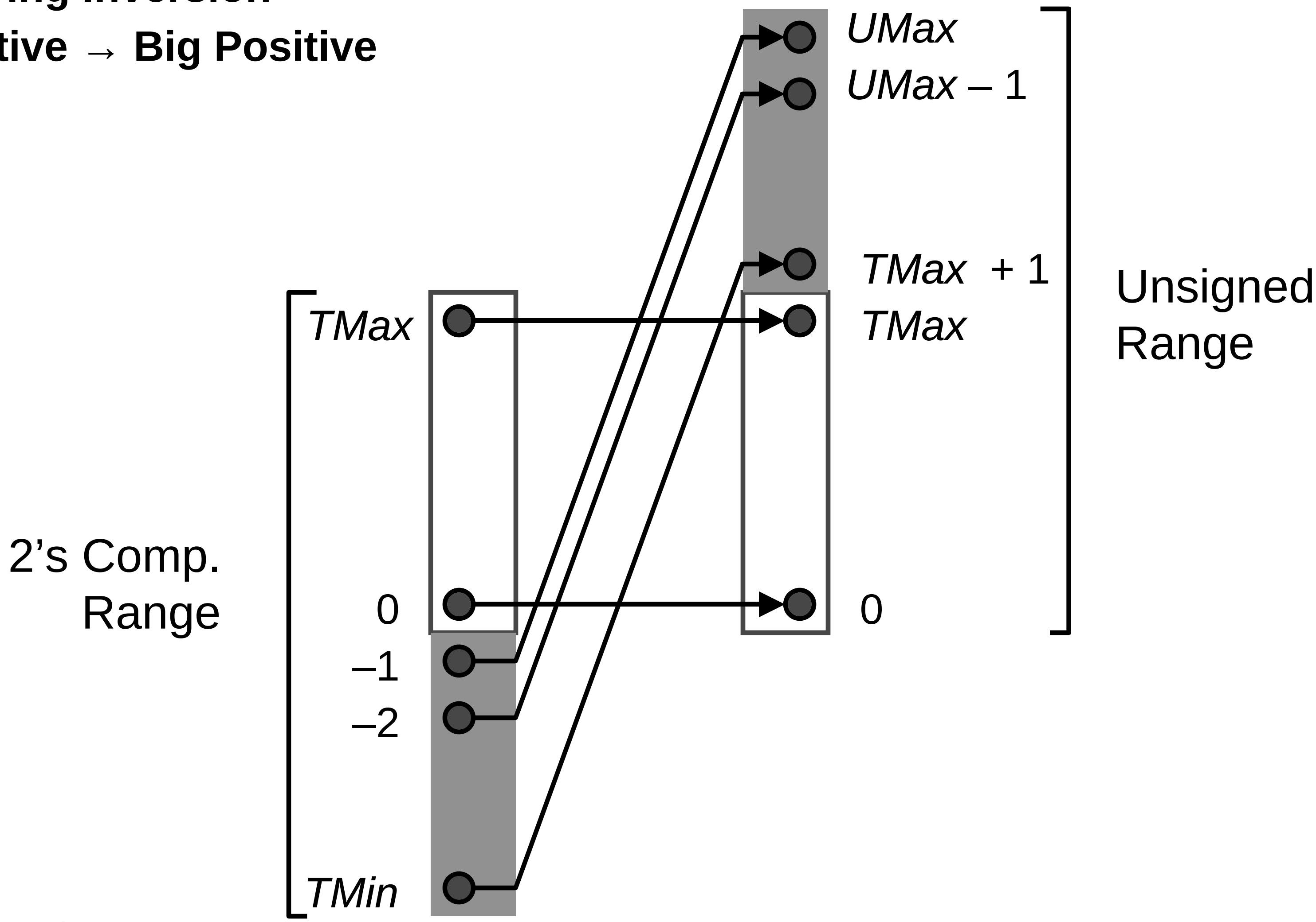
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=
- Examples for *W* = 32

Constant$_1$	Constant$_2$	Relation	Evaluation
`0`	`0U`	==	unsigned
`-1`	`0`	<	signed
`-1`	`0U`	>	unsigned
`2147483647`	`-2147483648`	>	signed
`2147483647U`	`-2147483648`	<	unsigned
`-1`	`-2`	>	signed
`(unsigned) -1`	`-2`	>	unsigned
`2147483647`	`2147483648U`	<	unsigned
`2147483647`	`(int) 2147483648U`	>	signed

Explanation of Casting Surprises

2's Comp. → Unsigned

- **Ordering Inversion**
- **Negative → Big Positive**



`class07.ppt`

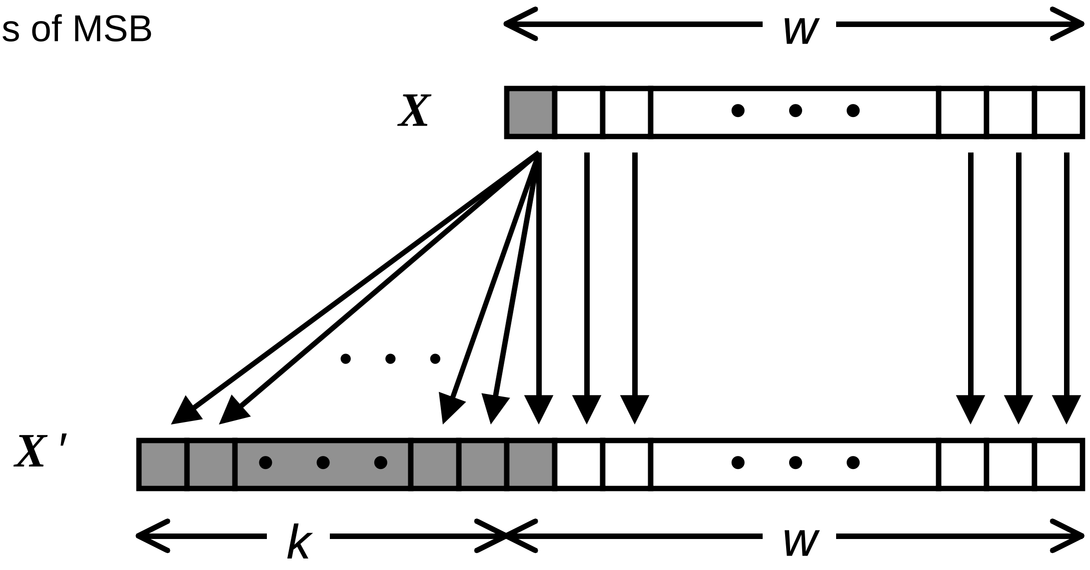
Sign Extension

Task:

- **Given *w*-bit signed integer *x***
- **Convert it to *w*+*k*-bit integer with same value**

Rule:

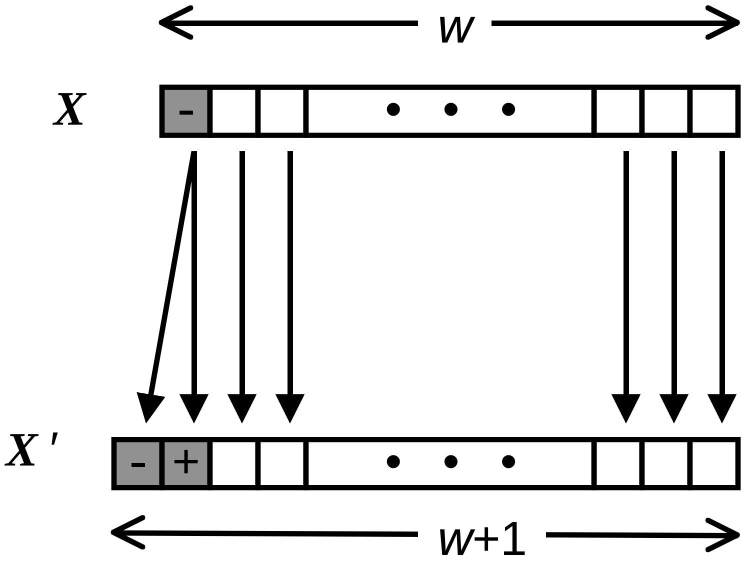
- **Make *k* copies of sign bit:**
- $X' = \underbrace{x_{w-1},\ldots,x_{w-1}}_{k\text{ copies of MSB}}, x_{w-1}, x_{w-2},\ldots, x_0$



Justification For Sign Extension

Prove Correctness by Induction on *k*

- **Induction Step: extending by single bit maintains value**



- **Key observation:** $-2^{w-1} = -2^{w} + 2^{w-1}$
- **Look at weight of upper bits:**

$X \quad -2^{w-1} x_{w-1}$

$X' \quad -2^{w} x_{w-1} + 2^{w-1} x_{w-1} = -2^{w-1} x_{w-1}$

Integer Operation C Puzzles

- **Assume machine with 32 bit word size, two’s complement integers**
- **For each of the following C expressions, either:**
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```
int x = foo();

int y = bar();

unsigned ux = x;

unsigned uy = y;
```

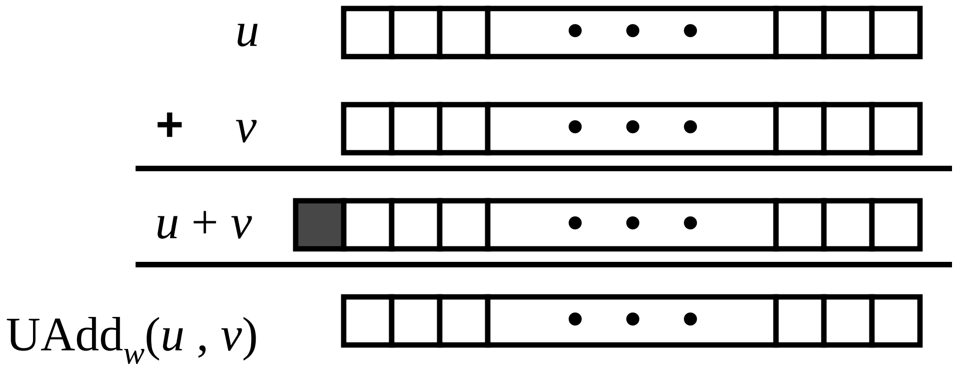
- `x < 0` `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` `(x<<30) < 0`
- `ux > -1`
- `x > y` `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` `x + y > 0`
- `x >= 0` `-x <= 0`
- `x <= 0` `-x >= 0`

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Standard Addition Function

- **Ignores carry output**

Implements Modular Arithmetic

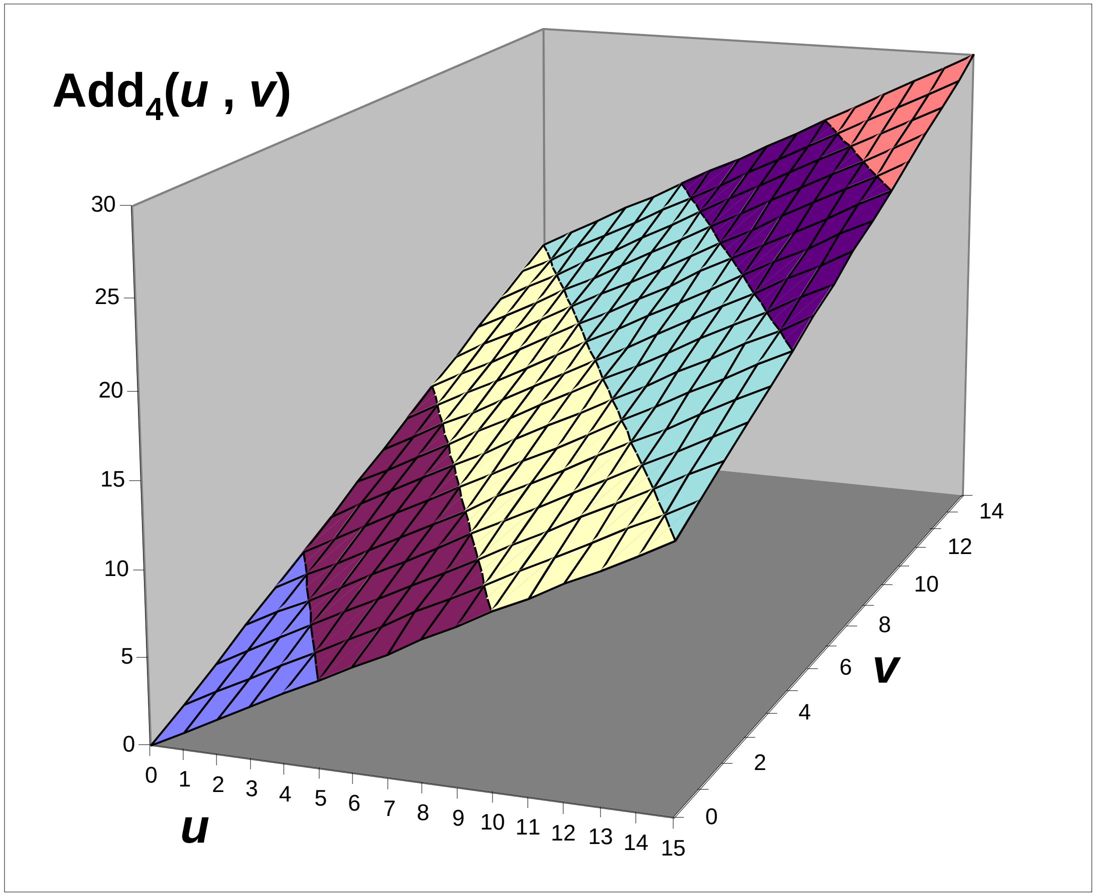
$s = \mathrm{UAdd}_w(u, v) = u + v \mod 2^w$

$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Visualizing Integer Addition

Integer Addition

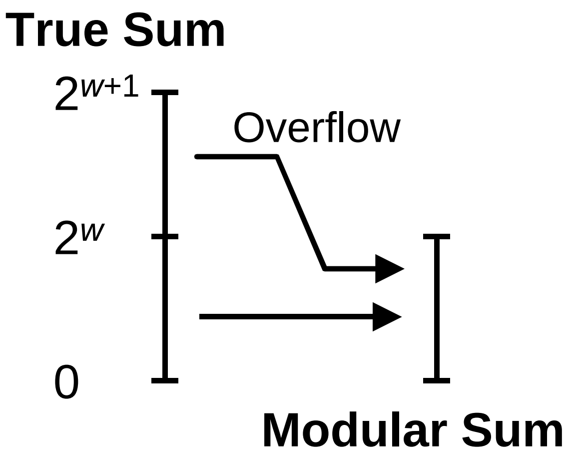
- 4-bit integers *u* and *v*
- Compute true sum Add$_4$(*u* , *v*)
- Values increase linearly with *u* and *v*
- Forms planar surface

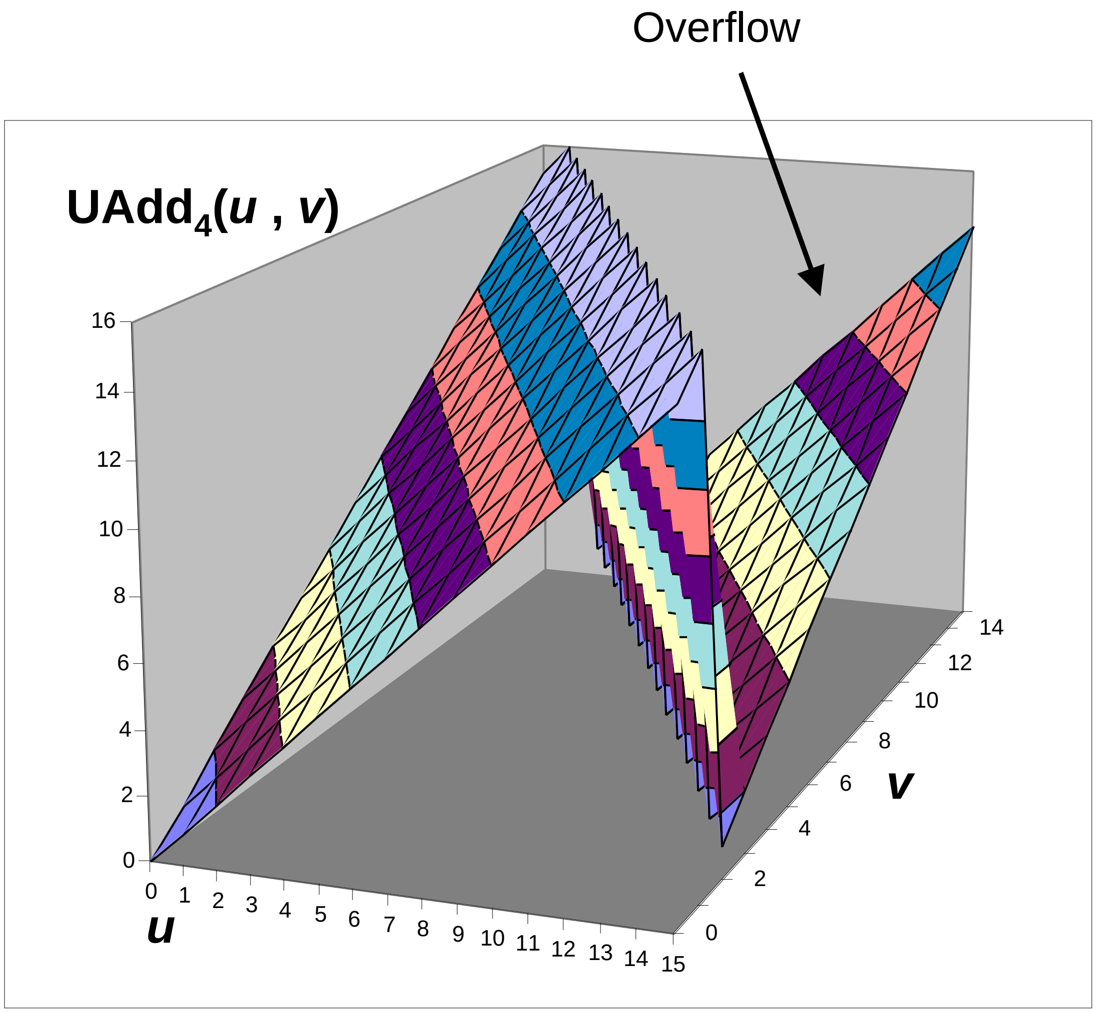


Visualizing Unsigned Addition

Wraps Around

- **If true sum ≥ 2^w**
- **At most once**





Mathematical Properties

Modular Addition Forms an *Abelian Group*

- **Closed under addition**

  $0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$

- **Commutative**

  $\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$

- **Associative**

  $\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$

- **0 is additive identity**

  $\text{UAdd}_w(u, 0) = u$

- **Every element has additive inverse**

  – Let $\text{UComp}_w(u) = 2^w - u$

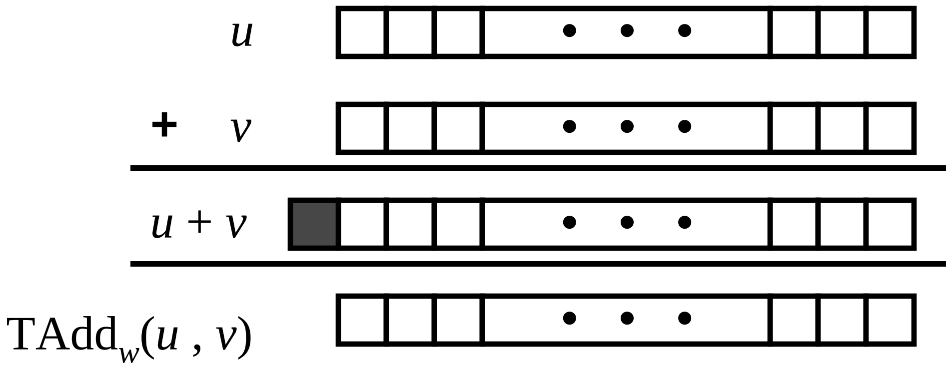
  $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

Two’s Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



TAdd and UAdd have Identical Bit-Level Behavior

- **Signed vs. unsigned addition in C:**

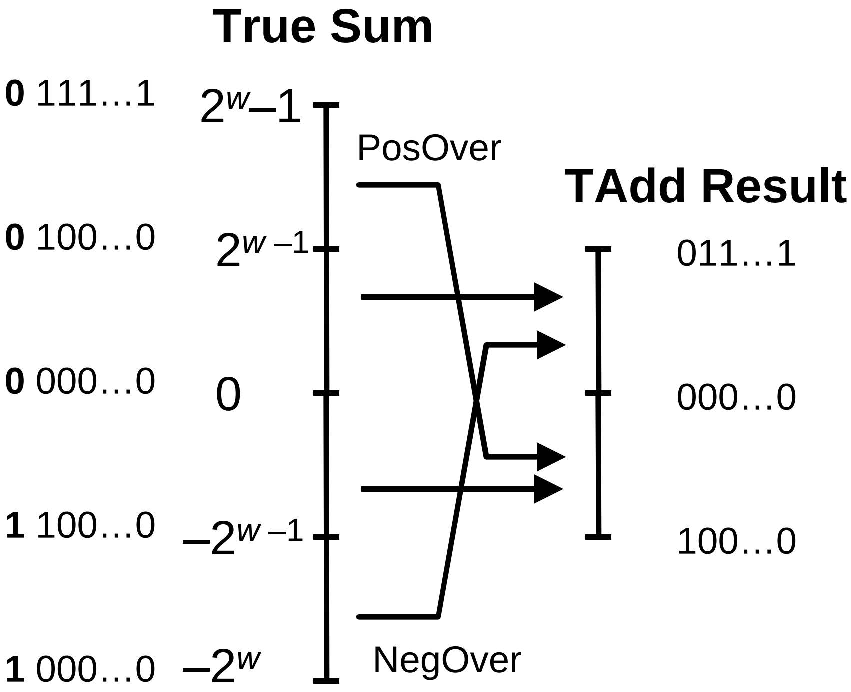
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
 t = u + v
```

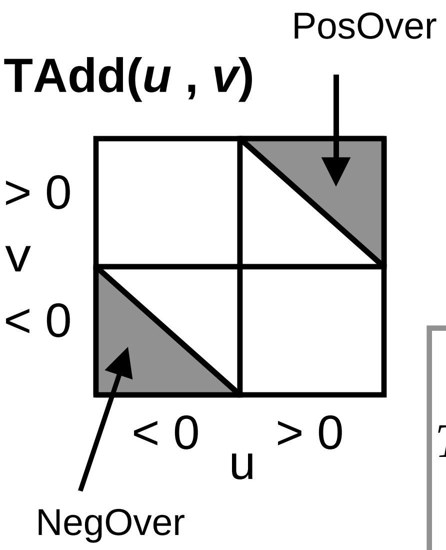
- **Will give `s == t`**

Characterizing TAdd

Functionality

- **True sum requires *w*+1 bits**
- **Drop off MSB**
- **Treat remaining bits as 2's comp. integer**





$$TAdd_w(u,v) = \begin{cases} u+v+2^{w-1} & u+v < TMin_w \quad \text{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^{w-1} & TMax_w < u+v \quad \text{(PosOver)} \end{cases}$$

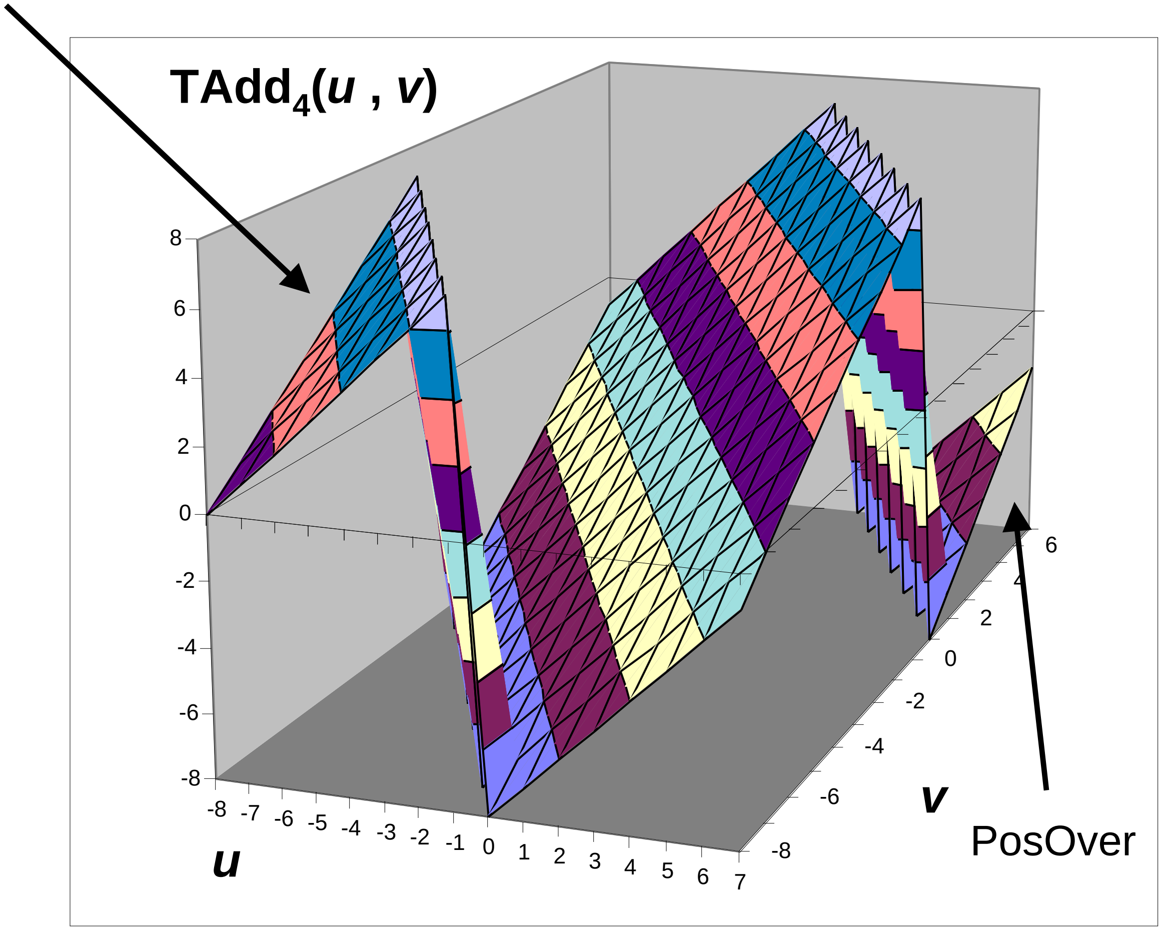
Visualizing 2’s Comp. Addition

Values

- **4-bit two’s comp.**
- **Range from -8 to +7**

Wraps Around

- **If sum ≥ 2^{w-1}**
  - Becomes negative
  - At most once
- **If sum < -2^{w-1}**
  - Becomes positive
  - At most once



NegOver

PosOver

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- $\text{TAdd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$
  - Since both have identical bit patterns

Two's Complement Under TAdd Forms a Group

- **Closed, Commutative, Associative, 0 is additive identity**
- **Every element has additive inverse**
  - Let $\text{TComp}_w(u) = \text{U2T}(\text{UComp}_w(\text{T2U}(u))$

  $\text{TAdd}_w(u, \text{TComp}_w(u)) = 0$

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Two's Complement Negation

Mostly like Integer Negation

- TComp(*u*) = –*u*

***TMin* is Special Case**

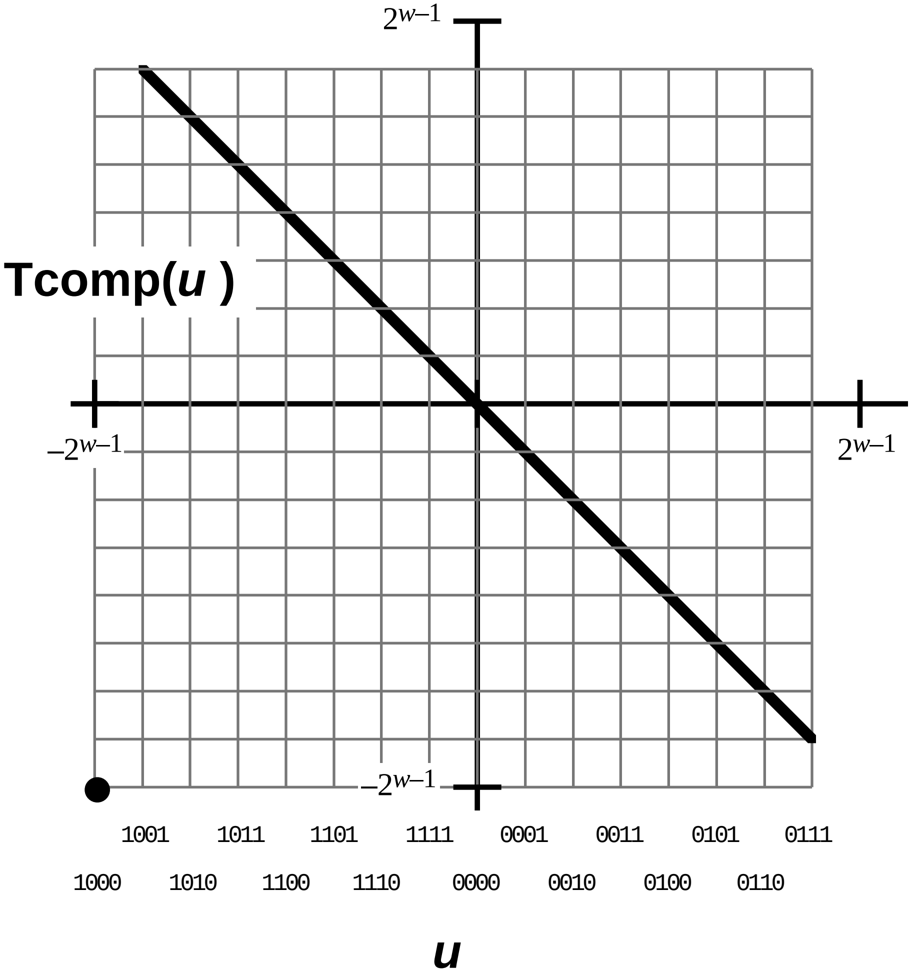
- TComp(*TMin*) = *TMin*

Negation in C is Actually TComp

```
mx = -x
```

- *mx* = TComp(*x*)
- **Computes additive inverse for** TAdd

```
x + -x == 0
```



Negating with Complement & Increment

In C

`~x + 1 == -x`

Complement

- Observation: `~x + x == 1111…11`$_2$ `== -1`

x	1	0	0	1	1	1	0	1
+ ~x	0	1	1	0	0	0	1	0
-1	1	1	1	1	1	1	1	1

Increment

- $\sim x + \cancel{x} + (\cancel{-x} + 1) \quad == \quad \cancel{-1} + (-x + \cancel{1})$
- `~x + 1` == `-x`

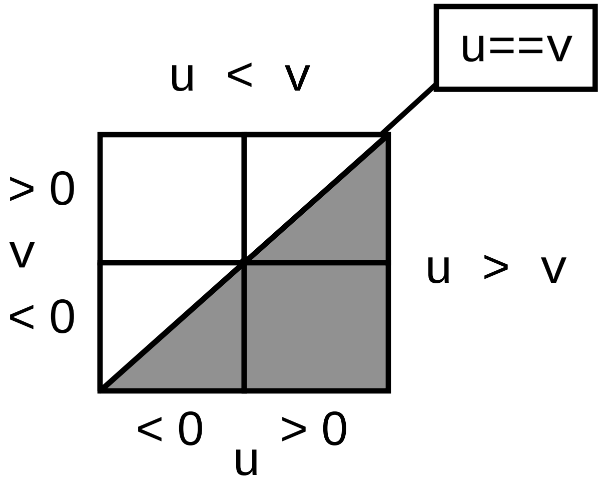
Warning: Be cautious treating `int`'s as integers

- **OK here: We are using group properties of** TAdd **and** TComp

Comparing Two’s Complement Numbers

Task

- **Given** signed numbers *u*, *v*
- Determine whether or not *u* > *v*
  - Return 1 for numbers in shaded region below



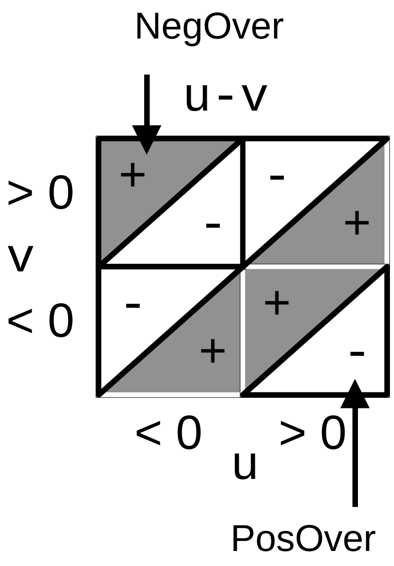
Bad Approach

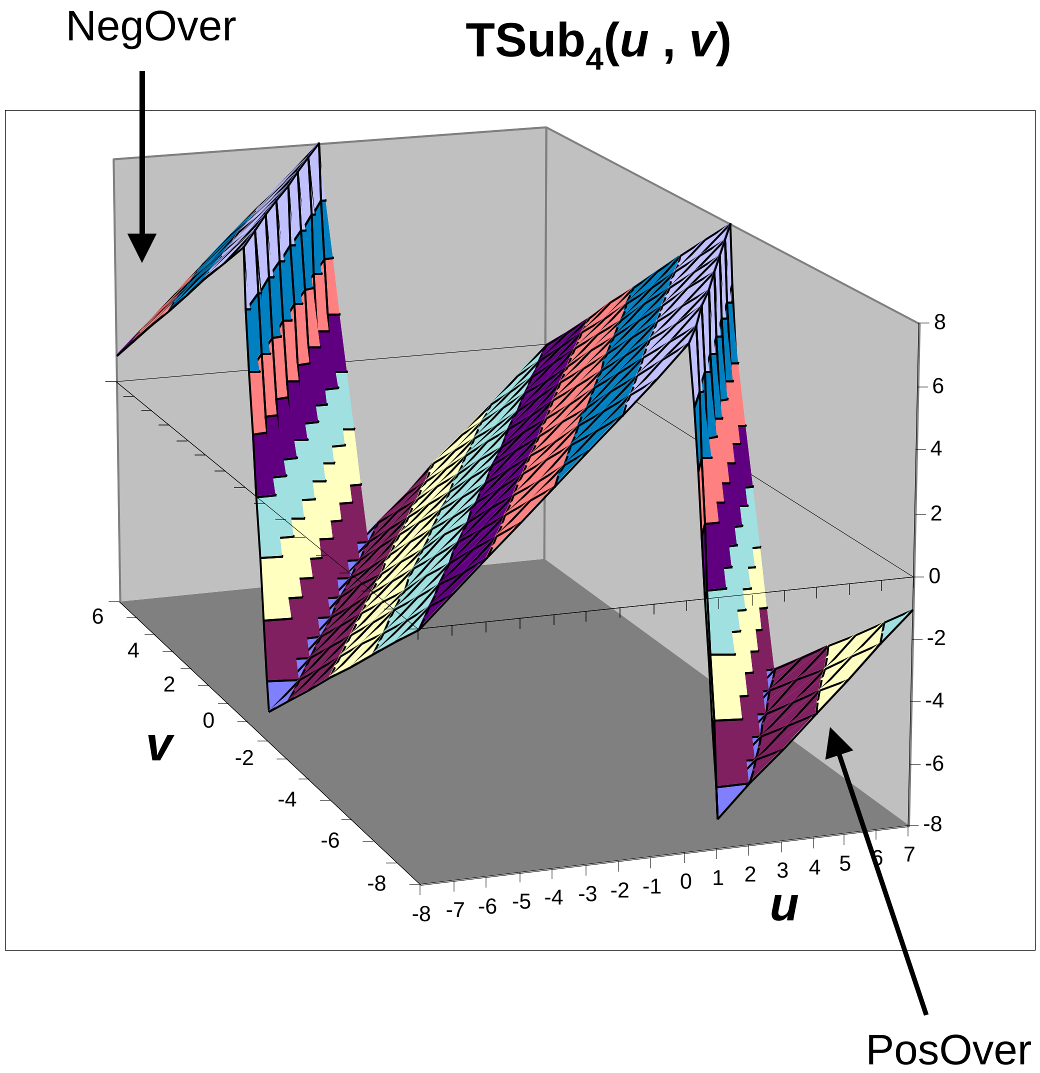
- Test `(u–v) > 0`
  - TSub(*u*,*v*) = TAdd(*u*, TComp(*v*))
- Problem: Thrown off by either Negative or Positive Overflow

Comparing with TSub

Will Get Wrong Results

- **NegOver:** `u < 0`, `v > 0`
  - but `u-v > 0`
- **PosOver:** `u > 0`, `v < 0`
  - but `u-v < 0`





Multiplication

Computing Exact Product of w-bit numbers x, y

- **Either signed or unsigned**

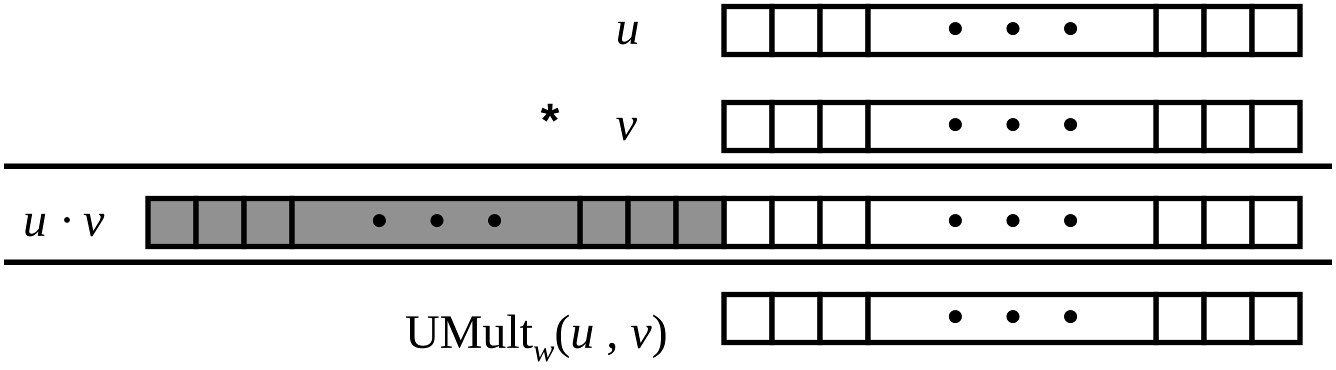
Ranges

- **Unsigned:** $0 \leq x * y \leq (2^{w} - 1)^{2} = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- **Two's complement min:** $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits
- **Two's complement max:** $x * y \leq (-2^{w-1})^{2} = 2^{2w-2}$
  - Up to $2w$ bits, but only for $TMin_{w}^{2}$

Maintaining Exact Results

- **Would need to keep expanding word size with each product computed**
- **Done in software by "arbitrary precision" arithmetic packages**
- **Also implemented in Lisp, ML, and other "advanced" languages**

Unsigned Multiplication in C



Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits

Standard Multiplication Function

- **Ignores high order *w* bits**

Implements Modular Arithmetic

$\mathrm{UMult}_w(u , v) = u \cdot v \bmod 2^w$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

```
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

- **Truncates product to *w*-bit number** up **=** $\text{UMult}_w(ux, uy)$
- **Simply modular arithmetic**

$up = ux \quad uy \mod 2^w$

Two’s Complement Multiplication

```
int x, y;
int p = x * y;
```

- **Compute exact product of two *w*-bit numbers *x*, *y***
- **Truncate result to*w*-bit number** $p = \text{TMult}_w(x, y)$

Relation

- **Signed multiplication gives same bit-level result as unsigned**
- `up == (unsigned) p`

Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms Commutative Ring

- **Addition is commutative group**
- **Closed under multiplication**

  $0 \leq \mathrm{UMult}_w(u, v) \leq 2^w - 1$
- **Multiplication Commutative**

  $\mathrm{UMult}_w(u, v) = \mathrm{UMult}_w(v, u)$
- **Multiplication is Associative**

  $\mathrm{UMult}_w(t, \mathrm{UMult}_w(u, v)) = \mathrm{UMult}_w(\mathrm{UMult}_w(t, u), v)$
- **1 is multiplicative identity**

  $\mathrm{UMult}_w(u, 1) = u$
- **Multiplication distributes over addtion**

  $\mathrm{UMult}_w(t, \mathrm{UAdd}_w(u, v)) = \mathrm{UAdd}_w(\mathrm{UMult}_w(t, u), \mathrm{UMult}_w(t, v))$

Properties of Two’s Comp. Arithmetic

Isomorphic Algebras

- **Unsigned multiplication and addition**
  - Truncating to *w* bits
- **Two’s complement multiplication and addition**
  - Truncating to *w* bits

Both Form Rings

- **Isomorphic to ring of integers mod** 2^w

Comparison to Integer Arithmetic

- **Both are rings**
- **Integers obey ordering properties, e.g.,**

$u > 0 \quad \Rightarrow \quad u + v > v$

$u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$

- **These properties are not obeyed by two’s complement arithmetic**

TMax **`+ 1        ==`** *TMin*

`15213 * 30426 ==  -10030`

Integer C Puzzle Answers

- **Assume machine with 32 bit word size, two's complement integers**
- *TMin* **makes a good counterexample in many cases**

`x < 0`	`((x*2) < 0)`	**False:**	*TMin*
`ux >= 0`		**True:**	0 = *UMin*
`x & 7 == 7`	`(x<<30) < 0`	**True:**	$x_1 = 1$
`ux > -1`		**False:**	0
`x > y`	`-x < -y`	**False:**	`-1`, *TMin*
`x * x >= 0`		**False:**	`30426`
`x > 0 && y > 0`	`x + y > 0`	**False:**	*TMax*, *TMax*
`x >= 0`	`-x <= 0`	**True:**	–*TMax* < 0
`x <= 0`	`-x >= 0`	**False:**	*TMin*

Floating Point Puzzles

- **For each of the following C expressions, either:**
  - – Argue that is true for all argument values
  - – Explain why not true

```
int x = …;

float f = …;

double d = …;
```

Assume neither d nor f is NAN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0          ((d*2) < 0.0)`
- `d > f            -f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

IEEE Floating Point

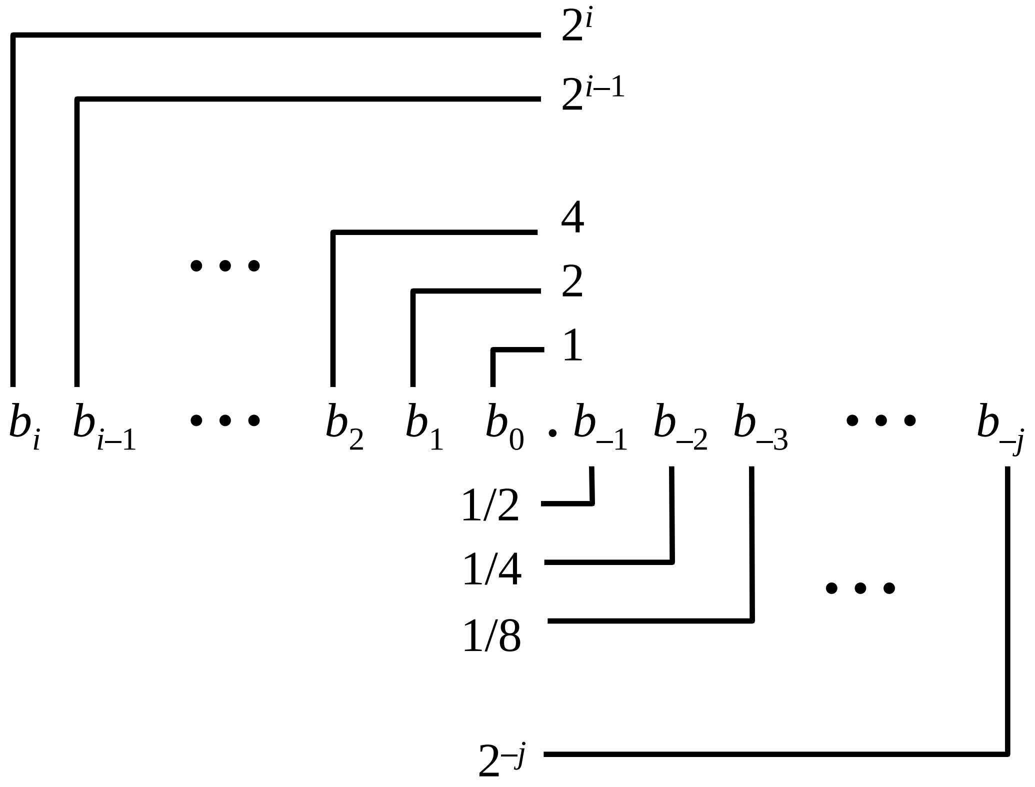
IEEE Standard 754

- **Estabilished in 1985 as uniform standard for floating point arithmetic**
  - Before that, many idiosyncratic formats
- **Supported by all major CPUs**

Driven by Numerical Concerns

- **Nice standards for rounding, overflow, underflow**
- **Hard to make go fast**
  - Numercial analysts predominated over hardware types in defining standard

Fractional Binary Numbers



Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^{i} b_k \quad 2^k$$

Fractional Binary Number Examples

Value	Representation
5-3/4	$\mathtt{101.11}_2$
2-7/8	$\mathtt{10.111}_2$
63/64	$\mathtt{0.111111}_2$

Observation

- **Divide by 2 by shifting right**
- **Numbers of form** $\mathtt{0.111111...}_2$ **just below 1.0**
  - Use notation 1.0 – ε

Limitation

- **Can only exactly represent numbers of the form** $x/2^k$
- **Other numbers have repeating bit representations**

Value	Representation
1/3	$\mathtt{0.0101010101[01]...}_2$
1/5	$\mathtt{0.001100110011[0011]...}_2$
1/10	$\mathtt{0.0001100110011[0011]...}_2$

Floating Point Representation

Numerical Form

- $-1^{s}\ m\ \ 2^{E}$
  - Sign bit s determines whether number is negative or positive
  - Mantissa m normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

Encoding

s	exp	significand

- **MSB is sign bit**
- **Exp field encodes *E***
- **Significand field encodes *m***

Sizes

- **Single precision: 8 exp bits, 23 significand bits**
  - 32 bits total
- **Double precision: 11 exp bits, 52 significand bits**
  - 64 bits total

“Normalized” Numeric Values

Condition

- **`exp` ≠ `000…0` and `exp` ≠ `111…1`**

Exponent coded as *biased* value

E* = *Exp* – *Bias

  - *Exp* : unsigned value denoted by **`exp`**
  - *Bias* : Bias value
    - » Single precision: 127
    - » Double precision: 1023

Mantissa coded with implied leading 1

$m = 1.\mathtt{xxx…x}_2$

  - **`xxx…x`**: bits of significand
  - Minimum when **`000…0`** (m = 1.0)
  - Maximum when **`111…1`** (m = 2.0 – ε)
  - Get extra leading bit for “free”

Normalized Encoding Example

Value

`Float F = 15740.0;`

- $15740_{10} = 11110101111100_2 = 1.1101101101101_2 \text{ X } 2^{13}$

Significand

m **=** $1.\underline{1101101101101}_2$

sig = $\mathbf{\underline{1101101101101}0000000000_2}$

Exponent

E **=** **13**

Bias **=** **127**

Exp **=** **140 = $\mathbf{10001100_2}$**

Floating Point Representation of 15740.0:

Hex: **4 6 7 5 f 0 0 0**
Binary: **0100 0110 0111 0101 1111 0000 0000 0000**

140: **100 0110 0**

15740: ***1*111 0101 1111 00**

Denormalized Values

Condition

- **`exp` = `000…0`**

Value

- **Exponent value *E* = –*Bias* + 1**
- **Mantissa value *m* =** $0.xxx\ldots x_2$
  - **`xxx…x`:** bits of significand

Cases

- **`exp` = `000…0`, `significand` = `000…0`**
  - Represents value 0
  - Note that have distinct values +0 and –0
- **`exp` = `000…0`, `significand` ≠ `000…0`**
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”

Interesting Numbers

Description	**Exp**	**Significand**	**Numeric Value**
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}}$ X $2^{-\{126,1022\}}$
• Single ≈ 1.4 X 10^{-45}			
• Double ≈ 4.9 X 10^{-324}			
Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon)$ X $2^{-\{126,1022\}}$
• Single ≈ 1.18 X 10^{-38}			
• Double ≈ 2.2 X 10^{-308}			
Smallest Pos. Normalized	00…01	00…00	1.0 X $2^{-\{126,1022\}}$
• Just larger than largest denormalized			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon)$ X $2^{\{127,1023\}}$
• Single ≈ 3.4 X 10^{38}			
• Double ≈ 1.8 X 10^{308}			

Memory Referencing Bug Example

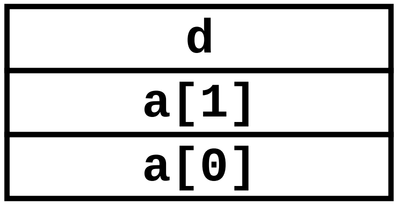
Demonstration of corruption by out-of-bounds array reference

```
main ()
{
  long int a[2];
  double d = 3.14;
  a[2] = 1073741824; /* Out of bounds reference */
  printf("d = %.15g\n", d);
  exit(0);
}
```

	Alpha	MIPS	Sun
-g	5.30498947741318e-315	3.1399998664856	3.14
-O	3.14	3.14	3.14

Referencing Bug on Alpha

Alpha Stack Frame (-g)



```
long int a[2];
double d = 3.14;
a[2] = 1073741824;
```

Optimized Code

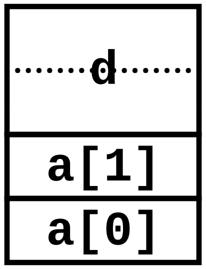
- Double d stored in register
- Unaffected by errant write

Alpha -g

- `1073741824` = `0x40000000` = 2^{30}
- **Overwrites all 8 bytes with value `0x0000000040000000`**
- **Denormalized value** 2^{30} X (smallest denorm 2^{-1074}) **=** 2^{-1044}
- $\approx 5.305 \times 10^{-315}$

Referencing Bug on MIPS

MIPS Stack Frame (-g)



```
  long int a[2];
  double d = 3.14;
  a[2] = 1073741824;
```

MIPS -g

- **Overwrites lower 4 bytes with value `0x40000000`**
- **Original value `3.14` represented as `0x40091eb851eb851f`**
- **Modified value represented as `0x40091eb840000000`**
- *Exp* = 1024 *E* = 1024–1023 = 1
- **Mantissa difference:** $\mathtt{.0000011eb851f}_{16}$
- **Integer value:** $11eb851f_{16} = 300{,}647{,}711_{10}$
- **Difference =** $2^{1} \times 2^{-52} \times 300{,}647{,}711 \approx 1.34 \times 10^{-7}$
- **Compare to** 3.140000000 – 3.139999866 = 0.000000134

Special Values

Condition

- **`exp`** **= `111…1`**

Cases

- **`exp`** **= `111…1`, `significand`** **= `000…0`**
  - Represents value ∞(infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞,   1.0/−0.0 = −∞
- **`exp`** **= `111…1`, `significand`** **≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞
  - No fixed meaning assigned to significand bits

Special Properties of Encoding

FP Zero Same as Integer Zero

- **All bits = 0**

Can (Almost) Use Unsigned Integer Comparison

- **Must first compare sign bits**
- **NaNs problematic**
  - Will be greater than any other values
  - What should comparison yield?
- **Otherwise OK**
  - Denorm vs. normalized
  - Normalized vs. infinity

Floating Point Operations

Conceptual View

- **First compute exact result**
- **Make it fit into desired precision**
  - Possibly overflow if exponent too large
  - Possibly round to fit into significand

Rounding Modes (illustrate with $ rounding)

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
• **Zero**	$1.00	$2.00	$1.00	$2.00	–$1.00
• $-\infty$	$1.00	$2.00	$1.00	$2.00	–$2.00
• $+\infty$	$1.00	$2.00	$2.00	$3.00	–$1.00
• **Nearest Even (default)**	$1.00	$2.00	$2.00	$2.00	–$2.00

A Closer Look at Round-To-Even

Default Rounding Mode

- **Hard to get any other kind without dropping into assembly**
- **All others are statistically biased**
  - Sum of set of positive numbers will consistently be over- or under-estimated

Applying to Other Decimal Places

- **When exactly halfway between two possible values**
  - Round so that least signficant digit is even
- **E.g., round to nearest hundredth**

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

Binary Fractional Numbers

- **"Even" when least significant bit is `0`**
- **Half way when bits to right of rounding position = $100\ldots_2$**

Examples

- **Round to nearest 1/4 (2 bits right of binary point)**

Value	Binary	Rounded	Action	Rounded Value
2-3/32	10.00011_2	10.00_2	(<1/2—down)	2
2-3/16	10.00110_2	10.01_2	(>1/2—up)	2-1/4
2-7/8	10.11100_2	11.00_2	(1/2—up)	3
2-5/8	10.10100_2	10.10_2	(1/2—down)	2-1/2

FP Multiplication

Operands

$(-1)^{s1}\ m1\ \ 2^{E1}$

$(-1)^{s2}\ m2\ \ 2^{E2}$

Exact Result

$(-1)^{s}\ m\ \ 2^{E}$

- **Sign** s**:** $s1$ ^ $s2$
- **Mantissa** m**:** $m1 * m2$
- **Exponent** E**:** $E1 + E2$

Fixing

- **Overflow if** E **out of range**
- **Round** m **to fit significand precision**

Implementation

- **Biggest chore is multiplying mantissas**

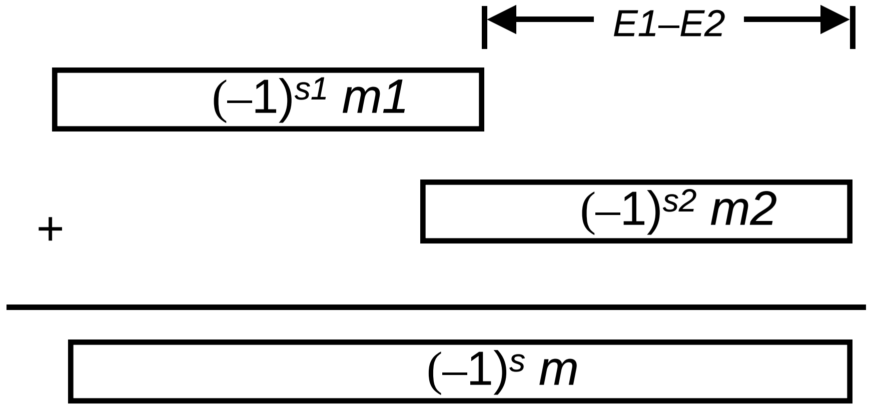
FP Addition

Operands

$(-1)^{s1}\ m1\ \ 2^{E1}$

$(-1)^{s2}\ m2\ \ 2^{E2}$

- **Assume** $E1$ **>** $E2$



Exact Result

$(-1)^{s}\ m\ \ 2^{E}$

- **Sign** s**, mantissa** m**:**
  - Result of signed align & add
- **Exponent** E**:** $E1 - E2$

Fixing

- **Shift** m **right, increment** E **if** $m \geq 2$
- **Shift** m **left** k **positions, decrement** E **by** k **if** $m < 1$
- **Overflow if** E **out of range**
- **Round** m **to fit significand precision**

Mathematical Properties of FP Add

Compare to those of Abelian Group

- **Closed under addition?** **YES**
  - But may generate infinity or NaN
- **Commutative?** **YES**
- **Associative?** **NO**
  - Overflow and inexactness of rounding
- **0 is additive identity?** **YES**
- **Every element has additive inverse** **ALMOST**
  - Except for infinities & NaNs

Montonicity

- $a \le b \quad a+c \le b+c$**?** **ALMOST**
  - Except for infinities & NaNs

Algebraic Properties of FP Mult

Compare to Commutative Ring

- **Closed under multiplication? YES**
  - But may generate infinity or NaN
- **Multiplication Commutative? YES**
- **Multiplication is Associative? NO**
  - Possibility of overflow, inexactness of rounding
- **1 is multiplicative identity? YES**
- **Multiplication distributes over addtion? NO**
  - Possibility of overflow, inexactness of rounding

Montonicity

- $a \le b$ & $c \ge 0$ $a * c \le b * c$**?** **ALMOST**
  - Except for infinities & NaNs

Floating Point in C

C Supports Two Levels

`float`	**single precision**
`double`	**double precision**

Conversions

- **Casting between `int`, `float`, and `double` changes numeric values**
- **`Double` or `float` to `int`**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - Generally saturates to TMin or TMax
- **`int` to `double`**
  - Exact conversion, as long as int has ≤ 54 bit word size
- **`int` to `float`**
  - Will round according to rounding mode

Answers to Floating Point Puzzles

```
int x = …;

float f = …;

double d = …;
```

Assume neither
d nor `f` is NAN

`x == (int)(float) x`		**No: 24 bit mantissa**
`x == (int)(double) x`		**Yes: 53 bit mantissa**
`f == (float)(double) f`		**Yes: increases precision**
`d == (float) d`		**No: looses precision**
`f == -(-f);`		**Yes: Just change sign bit**
`2/3 == 2/3.0`		**No: 2/3 == 1**
`d < 0.0`	`((d*2) < 0.0)`	**Yes!**
`d > f`	`-f < -d`	**Yes!**
`d * d >= 0.0`		**Yes!**
`(d+f)-d == f`		**No: Not associative**

Alpha Floating Point

Implemented as Separate Unit

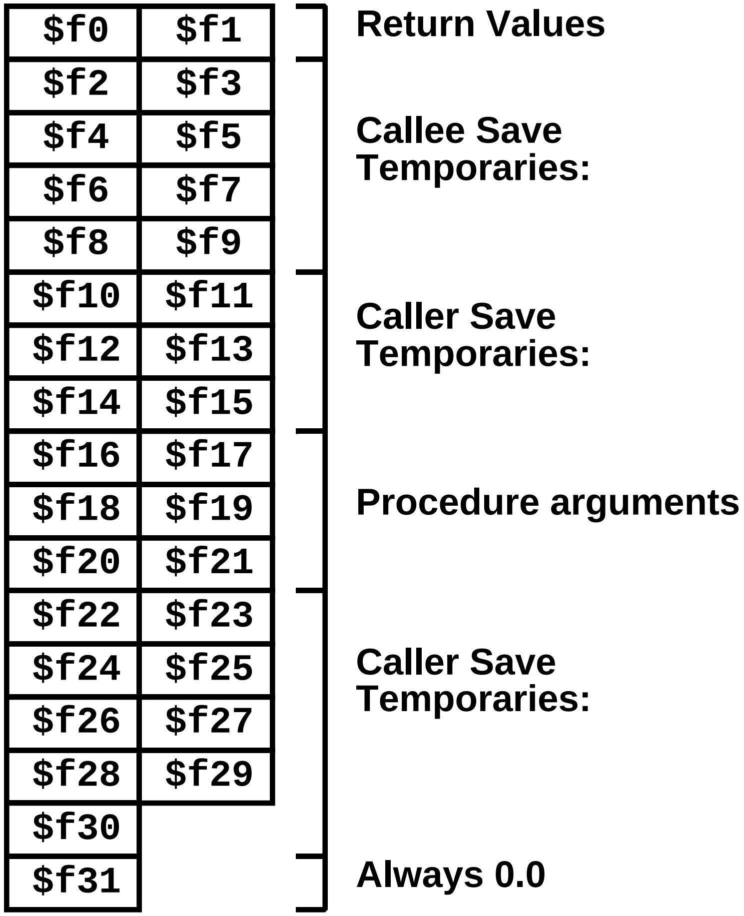
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats

- S_Floating (C `float`): 32 bits
- T_Floating (C `double`): 64 bits

Floating Point Data Registers

- 32 registers, each 8 bytes
- Labeled `$f0` to `$f31`
- `$f31` is always `0.0`



`$f0`	`$f1`	Return Values
`$f2`	`$f3`	Callee Save Temporaries:
`$f4`	`$f5`	
`$f6`	`$f7`	
`$f8`	`$f9`	
`$f10`	`$f11`	Caller Save Temporaries:
`$f12`	`$f13`	
`$f14`	`$f15`	
`$f16`	`$f17`	Procedure arguments
`$f18`	`$f19`	
`$f20`	`$f21`	
`$f22`	`$f23`	Caller Save Temporaries:
`$f24`	`$f25`	
`$f26`	`$f27`	
`$f28`	`$f29`	
`$f30`		
`$f31`		Always 0.0

Floating Point Code Example

Compute Inner Product of Two Vectors

- **Single precision arithmetic**

```
float inner_prodF
  (float x[], float y[],
   int n)
{
  int i;
  float result = 0.0;
  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
     cpys $f31,$f31,$f0 # result = 0.0
     bis $31,$31,$3     # i = 0
     cmplt $31,$18,$1   # 0 < n?
     beq $1,$102        # if not, skip loop
     .align 5
$104:
     s4addq $3,0,$1     # $1 = 4 * i
     addq $1,$16,$2     # $2 = &x[i]
     addq $1,$17,$1     # $1 = &y[i]
     lds $f1,0($2)      # $f1 = x[i]
     lds $f10,0($1)     # $f10 = y[i]
     muls $f1,$f10,$f1  # $f1 = x[i] * y[i]
     adds $f0,$f1,$f0   # result += $f1
     addl $3,1,$3       # i++
     cmplt $3,$18,$1    # i < n?
     bne $1,$104        # if so, loop
$102:
     ret $31,($26),1    # return
```

Numeric Format Conversion

Between Floating Point and Integer Formats

- **Special conversion instructions `cvttq`, `cvtqt`, `cvtts`, `cvtst`, …**
- **Convert source operand in one format to destination in other**
- **Both source & destination must be FP register**
  - Transfer to and from GP registers via memory store/load

C Code

```
float double2float(double d)
{
  return (float) d;
}
```

Conversion Code

```
cvtts $f16,$f0
```

[Convert T_Floating to S_Floating]

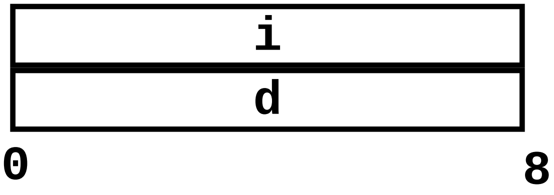
```
double long2double(long i)
{
  return (double) i;
}
```

```
stq $16,0($30)
ldt $f1,0($30)
cvtqt $f1,$f0
```

[Pass through stack and convert]

Getting FP Bit Pattern

```
double bit2double(long i)
{
  union {
    long i;
    double d;
  } arg;
  arg.i = i;
  return arg.d;
}
```



```
  stq $16,0($30)
  ldt $f0,0($30)
```

```
double long2double(long i)
{
  return (double) i;
}
```

```
  stq $16,0($30)
  ldt $f1,0($30)
  cvtqt $f1,$f0
```

- **Union provides direct access to bit representation of double**
- **`bit2double` generates double with given bit pattern**
  - NOT the same as **`(double) i`**
  - Bypasses rounding step

Alpha 21164 Arithmetic Performance

Integer

Operation	*Latency*	*Issue Rate*	*Comment*
• Add	1	2 / cycle	Two integer pipes
• LW Multiply	8	1 / 8 cycles	Unpipelined
• QW Multiply	16	1 / 16 cycles	Unpipelined
• Divide	∞	0 / cycle	Not implemented

Floating Point

Operation	*Latency*	*Issue Rate*	*Comment*
• Add	4	1 / cycle	Fully pipelined
• Multiply	4	1 / cycle	Fully pipelined
• SP Divide	10	1 / 10 cycle	Unpipelined
• DP Divide	23	1 / 23 cycle	Unpipelined