Bits, Bytes, and Integers (part V)

CSCI 237: Computer Organization
7th Lecture, Friday, Sept. 20

Kelly Shaw

Administrative Details

- Lab #2 checkpoint due Tuesday at 11pm
- Practice problems posted
- Weekly quiz due at 2:30pm today
- Read CSAPP 2.3, 2.1.3-2.1.4, 2.4-2.5
- CS Colloquium on Friday, 2:35pm in Wege
  - Concurrent Communication Contracts
  - Hannah Gommerstadt, Vassar College
  - A concurrent system is a system where multiple processes collaborate on a computation by exchanging messages. A communication contract represents a property of the computation that should remain true throughout the computation. Monitors can be used to check at runtime that a computation adheres to its contract. My work uses session types to monitor concurrent contracts. This talk will introduce session types, and present a variety of contracts that can be monitored.

Last Time: Bits, Bytes, and Integers

- C memory addressing
  - Arrays
  - References
  - Pointers

Today: Memory Layout and C pointers

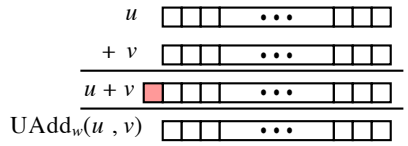
- Integers (Ch 2.2-2.3)
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, multiplication, shifting (Ch 2.3)
- Byte-oriented Memory Organization
  - Words
  - Byte-ordering

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

  s = UAdd$_w(u , v)$ = $u + v$ mod 2^w

```
unsigned char    1110 1001     E9     233
               +  1101 0101   + D5   + 213
               1 1011 1110    1BE    446
                  1011 1110    BE    190
```

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

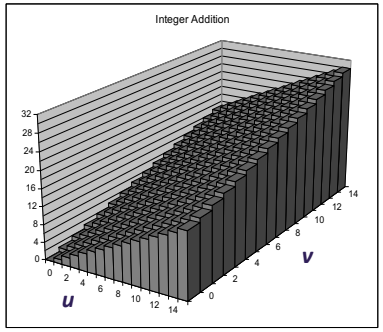
5

5

Visualizing (Mathematical) Integer Addition

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

Add$_4(u , v)$

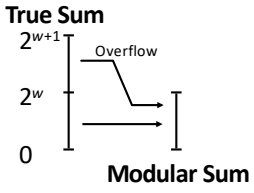


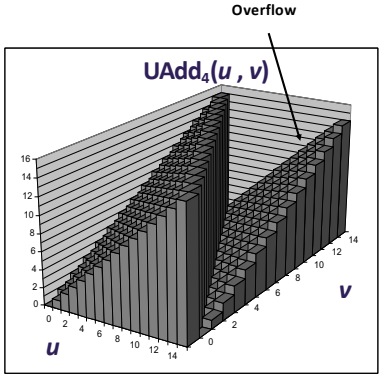
6

6

Visualizing Unsigned Addition

- Wraps Around
  - If true sum ≥ 2^w
  - At most once





7

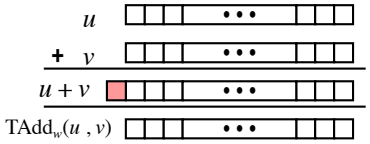
7

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`

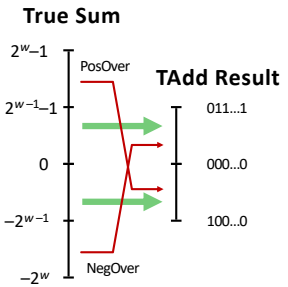
```
          1110 1001     E9     -23
        +  1101 0101   + D5   + -43
    0001 1011 1110    1BE    446
       1011 1110       BE    -66
```

8

8

TAdd Overflow

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



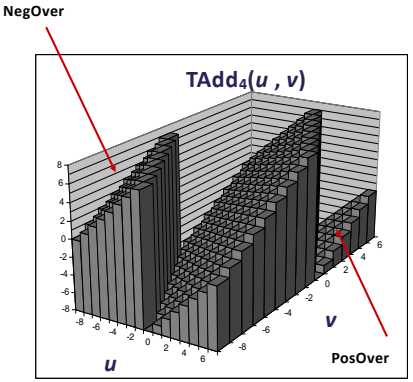
PosOver => truncation subtracts 2^w from sum
NegOver => truncation adds 2^w to sum

9

9

Visualizing 2's Complement Addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - Wraps at most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - Wraps at most once

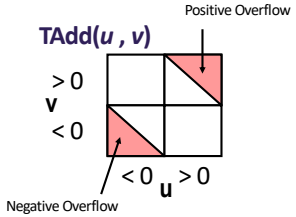


10

10

Another Visualization of TAdd

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



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

11

11

Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*
  - Either signed or unsigned
- But, exact results can be bigger than *w* bits
  - Unsigned: up to 2*w* bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-2 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

12

12

Binary Multiplication

- Same "algorithm" as decimal multiplication

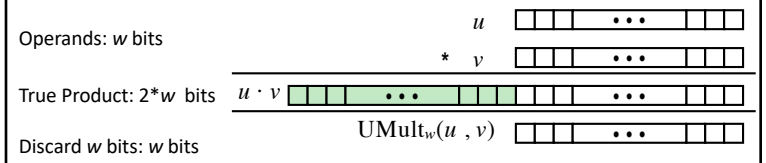
Example: 5*5 = 25

```
    101
*   101
------
    101
   000
+101
------
 11001
```

13

13

Unsigned Multiplication in C



- Standard Multiplication Function
  - Ignores high order *w* bits
- Implements Modular Arithmetic

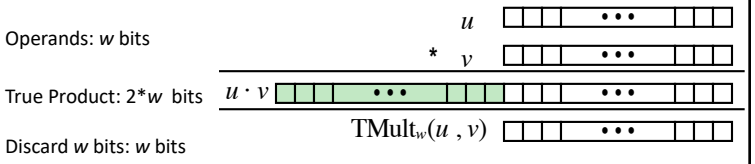
  $\text{UMult}_w(u , v) = u \cdot v \bmod 2^w$

```
              1110 1001      E9      233
*             1101 0101   *  D5   *  213
-----------------------------------------
 1100 0001 1101 1101      C1DD     49629
-----------------------------------------
           1101 1101        DD       221
```

14

14

Signed Multiplication in C



- Standard Multiplication Function
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same (as unsigned multiplication)

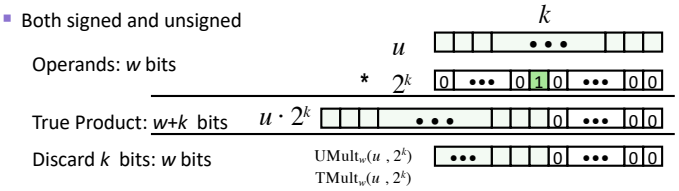
```
              1110 1001      E9      -23
*             1101 0101   *  D5   *  -43
-----------------------------------------
 0000 0011 1101 1101      03DD     989
-----------------------------------------
           1101 1101        DD      -35
```

15

15

Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u * 2^k`
  - Both signed and unsigned



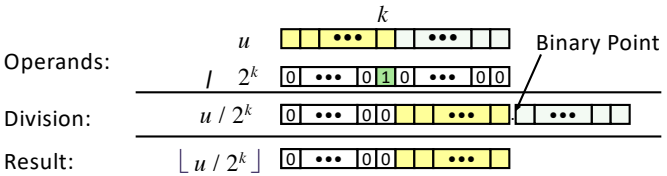
- Most machines shift and add faster than multiply
  - Rewrite expressions to use shifts instead of multiply
  - Compiler generates this code automatically

16

16

Unsigned Power-of-2 Divide with Shift

- Operation
  - `u >> k` gives $\lfloor$ `u / 2^k` $\rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

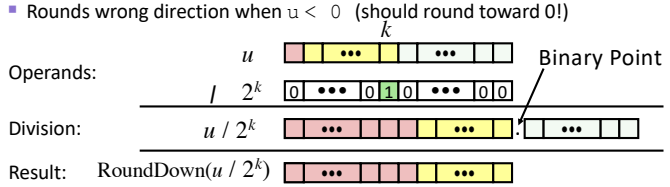
17

17

Signed Power-of-2 Divide with Shift

- Operation
  - `u >> k` gives $\lfloor$ `u / 2^k` $\rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0` (should round toward 0!)

We want::
$\lfloor u / 2^k \rfloor$ if u > 0, but
$\lceil u / 2^k \rceil$ if u < 0



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

18

18

Signed Power-of-2 Divide

- Operation behavior?
  - Want $\lfloor$ `u / 2^k` $\rfloor$ for positive quotients (Round Toward 0)
  - Want $\lceil$ `u / 2^k` $\rceil$ for negative quotients (Round Toward 0)
  - Compute as $\lfloor$ `(u+2^k-1) / 2^k` $\rfloor$
    - In C: `(u + (1<<k)-1) >> k`
    - `2^k-1` Biases dividend toward 0

Add 2^k-1 as a "bias": If dividing by 2^k, add bitvector with rightmost k bits set to 1

19

19

Bias Intuition (k=2, 2^k-1 = 3)

	16	17	18	19
	0100 00	0100 01	0100 10	0100 11
2^k-1 = 3	0000 11			
	0100 00	0100 01	0100 10	0100 11
+	0000 11	0000 11	0000 11	0000 11
	0100 11	0101 00	0101 01	0101 10

Bit k=2 only gets increased if bit positions k < 2 had at least one 1.

20

20

Divide by Power of 2 (k = 2, divide by 4)

	-16/4 = -4	-17/4 = -4.25	-18/4 = -4.5	-19/4 = -4.75
	1100 00	1011 11	1011 10	1011 01
Divide by Shifting 2	111100 ✓	111011 ✗	111011 ✗	111011 ✗
Add bias	1100 00 + 0000 11 = 1100 11	1011 11 + 0000 11 = 1100 10	1011 10 + 0000 11 = 1100 01	1011 01 + 0000 11 = 1100 00
Divide by Shifting 2	111100 ✓	111100 ✓	111100 ✓	111100 ✓
	-4	-4	-4	-4

21

21