C0 Primitive Operators and Invariants

Anand Subramanian
<asubrama@andrew.cmu.edu>

May 24, 2012

[1]

[1]Some slides borrowed from previous iterations of the class

Introduction

Topics Covered:

- Recap: Primitive operations
- Example 1: Grade-School Multiplication
- Example 2: Exponentiation by Squaring (redux)

Objective:

- Get practice using C0 primitive operators
- Understand how the properties of primitive operators fit in contracts
- Get practicing choosing when and how to use contracts

Recap: Integer operations
$(Z_{2^{32}}, +, *)$ (see lecture 3 notes)

Commutativity of addition	$x + y = y + x$
Associativity of addition	$(x + y) + z = x + (y + z)$
Additive unit	$x + 0 = x$
Additive inverse	$x + (-x) = 0$
Cancellation	$-(-x) = x$
Commutativity of multiplication	$x * y = y * x$
Associativity of multiplication	$(x * y) * z = x * (y * z)$
Multiplicative unit	$x * 1 = x$
Distributivity	$x * (y + z) = x * y + x * z$
Annihilation	$x * 0 = 0$

Left Shift

Drops most significant bit. Appends 0 as least significant bit. How does this affect the sign?

Left Shift

Drops most significant bit. Appends 0 as least significant bit. How does this affect the sign?

Consider $x = 01111010$ (8 bit arithmetic).

k	x << k	decimal equivalent
0	01111010	$+122$
1	11110100	-12 (Think: $+122 * 2 = +244 \equiv_{2^8} -12$)
2	11101000	-24
3	11010000	-48
4	10100000	-96
5	01000000	$+64$ (Think: $-96 * 2 = -192 \equiv_{2^8} +64$)
6	10000000	-128
7	00000000	0

Remember, in 2's complement arithmetic, the most significant bit is interpreted as the sign.

Right Shift

Drops least significant bit. Duplicates most significant bit. How does this affect the sign?

Right Shift

Drops least significant bit. Duplicates most significant bit. How does this affect the sign?

Consider $x = 01111010$.

k	x >> k	decimal equivalent
0	01111010	$+122$
1	00111101	$+61$
2	00011110	$+30$
3	00001111	$+15$
4	00000111	$+7$
5	00000011	$+3$
6	00000001	$+1$
7	00000000	0

Right Shift

Consider $x = 10000110$.

k	$x >> k$	decimal equivalent
0	10000110	-122
1	11000011	-61
2	11100001	-31 (not -30!)
3	11110000	-16 (not -15!)
4	11111000	-8
5	11111100	-4
6	11111110	-2
7	11111111	-1

Multiplication, Division, Modulus and Shift

- $(x/y) * y + x\%y = x$
- The quotient is truncated towards 0
- Any non-zero remainder takes the sign of x

Multiplication, Division, Modulus and Shift

- $(x/y) * y + x\%y = x$
- The quotient is truncated towards 0
- Any non-zero remainder takes the sign of x
- $x >> k$ divides by 2^k but truncates towards $-\infty$
- $x << k$ multiplies by 2^k.
- For $x >> k$ and $x << k$, only the five least significant bits of k are considered. Why?

Example 1: Grade School Multiplication

Multiplying two 4-bit numbers:

$$
\begin{array}{ccccccccc}
 & & & & & x_1 & x_2 & x_3 & x_4 \\
 & & & & & 1 & 0 & 1 & 0 \\
\hline
+ & & & & & 0 & 0 & 0 & 0 \\
+ & & & & x_1 & x_2 & x_3 & x_4 & \\
+ & & & 0 & 0 & 0 & 0 & & \\
+ & & x_1 & x_2 & x_3 & x_4 & & & \\
\hline
\end{array}
$$

We want a loop that does this.

Programming Attempt 1

```
int mult(int x, int y)
//@ensures \result == x*y;
{
    int n = x;
    int k = y;
    int res = 0;
    while (k != 0)
    {
        if ((k & 1) == 1)
            res = res + n;
        k = k >> 1;
        n = n << 1;
    }
    return res;
}
```

Programming Attempt 1: Loop Invariant

```
int mult(int x, int y)
//@ensures \result == x*y;
{
    int n = x;
    int k = y;
    int res = 0;
    while (k != 0)
        //@loop_invariant x*y == k*n + res;
    {
    if ((k & 1) == 1)
                res = res + n;
        k = k >> 1;
        n = n << 1;
    }
    return res;
}
```

Programming Attempt 1: Proving the invariant

Part 1: Before the loop condition is tested for the first time, $k = x$ and $n = y$ and $res = 0$:

$$\begin{aligned} x * y = \quad & k * n + res \\ = \quad & x * y + 0 \\ = \quad & x * y \end{aligned}$$

Programming Attempt 1: Proving the invariant

Part 2: Assume $x * y = k * n + res$ at the start of an iteration. We wish to show that $x * y = k' * n' + res'$ at the end of that iteration (just before the loop condition is tested again). We consider two cases:

Programming Attempt 1: Proving the invariant

Part 2: Assume $x * y = k * n + res$ at the start of an iteration. We wish to show that $x * y = k' * n' + res'$ at the end of that iteration (just before the loop condition is tested again). We consider two cases:

a: k is even, so $k\&1 == 0$.
Then $k' = k{>>}1 = k/2$ and $n' = n * 2$ and $res' = res$.

$$\begin{aligned} k' * n' + res' &= (k/2) * n * 2 + res \\ &= k * n + res \\ &= x * y \end{aligned}$$

Programming Attempt 1: Proving the invariant

Part 2:

b: k is odd, so $k\&1 = 1$.
Then $k' = k{>>}1 = (k-1)/2$ and $n' = n * 2$ and $res' = res + n$.

$$\begin{aligned} k' * n' + res' &= (k-1)/2 * n * 2 + res + n \\ &= (k-1) * n + res + n \\ &= k * n - n + res + n \\ &= k * n + res \\ &= x * y \end{aligned}$$

Programming Attempt 1: Proving the invariant

Part 2:

b: k is odd, so $k\&1 = 1$.
Then $k' = k{>>}1 = (k-1)/2$ and $n' = n*2$ and $res' = res + n$.

$$\begin{aligned} k' * n' + res' &= (k-1)/2 * n * 2 + res + n \\ &= (k-1) * n + res + n \\ &= k * n - n + res + n \\ &= k * n + res \\ &= x * y \end{aligned}$$

So the loop invariant holds at the end of this iteration, which means it also holds for the start of the next iteration since the loop condition does not change the values of any variables.

Programming Attempt 1: Does it terminate?

- What if y is negative?

Programming Attempt 1: Does it terminate?

- What if `y` is negative?
- Remember, >> copies over the sign bit!
- `k` converges to -1 instead of 0 in the loop.

Programming Attempt 1: Does it terminate?

- What if `y` is negative?
- Remember, >> copies over the sign bit!
- `k` converges to -1 instead of 0 in the loop.
- Note how this is not exactly induction. The loop-invariant looks like an induction principle, but it is there to assert a property about variables that have values assigned to them. It does not establish a convergence criterion.
- Perhaps a little more like co-induction? (ask Kristina)

Programming Attempt 1: Fix it with a pre-condition?

```
int mult(int x, int y)
//@requires y >= 0
//@ensures \result == x*y;
{
    int n = x;
    int k = y;
    int res = 0;
    while (k != 0)
        //@loop_invariant x*y == k*n + res;
    {
        if ((k & 1) == 1)
            res = res + n;
        k = k >> 1;
        n = n << 1;
    }
    return res;
}
```

Programming Attempt 2: Eliminate the pre-condition

Programming Attempt 2: Eliminate the pre-condition

```
int mult(int x, int y)
//@ensures \result == x*y;
{
    int n = x;
    int k = y;
    int res = 0;
    while (n != 0)
        //@loop_invariant x*y == k*n + res;
    {
        if ((k & 1) == 1)
            res = res + n;
        k = k >> 1;
        n = n << 1;
    }
    return res;
}
```

Programming Attempt 2: Now it is commutative!

Programming Attempt 2: Now it is commutative!

```
int mult(int x, int y)
//@ensures \result == x*y && \result == mult(y, x);
{
    int n = x;
    int k = y;
    int res = 0;
    while (n != 0)
        //@loop_invariant x*y == k*n + res;
    {
        if ((k & 1) == 1)
            res = res + n;
        k = k >> 1;
        n = n << 1;
    }
    return res;
}
```

Programming Attempt 2: Oops... contracts have effects!

Programming Attempt 2: Oops... contracts have effects!

```
int mult(int x, int y)
//@ensures \result == x*y;
{
        ...
}

int mult_wrapper(int x, int y)
//@ensures \result == mult(y, x)
{
        return mult(x, y);
}
```

Example 2: Exponentiation by Squaring (redux)

```
int exp(int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
    int res = 1;
    int b = x; /* base */ int e = y; /* exponent */
    while (e > 0)
        //@loop_invariant e >= 0;
        //@loop_invariant res * pow(b,e) == pow(x,y);
    {
        if ((e & 1) == 1) /* was e % 2 == 1 */
            res = b * res;
        b = b * b;
        e = e >> 1; /* was e = e / 2 */
    }
    //@assert e == 0;
    return res;
}
```

Why do our contracts still make sense?

- `e >= 0`
- Dividing and >> truncate in the same direction.
- The loop body has the same effect.
- Same proof as in lecture 2.
- Code may even run a bit faster (constant factor).

Slides are incomplete

Conclusion

- Beware how right shift does sign-extension.
- Remember the laws of the operators. Useful to prove contracts.
- Contracts can have effects too.
- Relational operators do not associate and distribute in fixed-width arithmetic.