

Re: Implementing an 8 bit fixed point register

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2008-07/msg00123.html>

- *From:* Grant Edwards <grante@xxxxxxxx>
 - *Date:* Tue, 01 Jul 2008 17:44:19 -0500
-

On 2008-07-01, Terry Reedy <tjreedy@xxxxxxxx> wrote:

A bytearray subclass could enforce that all 'bits' (stored as bytes) are 0 or 1, have a customized representation to your taste, and add methods like `.flipall()`.

It seems like implementing ALU operations on such arrays would be a lot more work than implementing bit-indexing on a type derived on a more "integer" like base. I'm pretty fuzzy on how one sub-classes basic things like integers, so maybe I'm all wet, and adding `__getitem__` and `__setitem__` to an integer type isn't even possible.

If one only wants bit operations, then the array approach is easy. If one only wants int arithmetic and all-bits logic, then int approach is easy. OP did not completely specify needs.

He said he's writing a microprocessor simulator, so he's going to want integer operations, all-bits logical operations, and individual bit access (by number and by name) and bit-slice access.

The problem with the int approach is that ints are immutable.

That dawned on me after I started googling around a little.

Therefore, if one wants to subclass int to hide the bit masking for the bit operations, one must override *every* operation one might use, including all arithmetic and all-bits logic, even when the int operation gives the correct answer other than the class of the result.

Re: Implementing an 8 bit fixed point register

Since we're doing fixed-width operations, Python's int operations don't give the correct answer other than the class of the result.

```
class bits(int):
...
def __add__(self,other):
return bit(self+other)
...
```

If one does not,

```
a,b = bits(1),bits(2)
c = a+b #c is now an int, not a bits
```

So there is a tendency to not subclass and instead either leave the extra functionality unmasked in repeated code or to put it in functions instead.

```
setters = (1,2,4,8,16,32,64, ..., 2147483648)# slightly pseudocode
unsetters = (~1,~2,~4,...~2147483648) # ditto
def bitset(n, bit): return n | setters[bit]
def bitunset(n,bit): return n & unsetters[bit]
```

On a machine with a barrel shifter, it's probably faster to do this:

```
def bitset(n,bit): return n | (1<<bit)
def bitclr(n,bit): return n & ~(1<<bit)
```

But, your approach could be easily modified to support slices:

```
import operator
masks = [(1<<n) for n in range(32)]
def mask(bits):
if type(bits) is slice:
return reduce(operator.__or__,masks[bits])
return masks[bits]

def bitset(n,bits):
return n | mask(bits)
```

Is there a literal syntax for a slice? This doesn't seem to work:

```
bitset(n,0:4)
```

Re: Implementing an 8 bit fixed point register

thus not getting the nice `reg[n]` functionality, nor an easy display of individual bits without callings **another** function.

One the other hand, with mutable arrays, setting bits is a mutation and so no override of `__setitem__` is required unless one wants to be fancy and enforce setting to 0 or 1.

More importantly, I presume that slices are supported so when you need values of bit-fields, you can do this:

```
op = instruction[6:8]
src = instruction[3:6]
dest = instruction[0:3]
```

The half-closed interval notation for slices is probably going to drive the programmer up the wall because all of the documentation that's being followed uses closed intervals.

It is a trade-off.

And Python programmers are awfully spoiled. :)

—

Grant Edwards grante Yow! I think my career
at is ruined!
visi.com

.