Software Performance Engineering

Recitation 1.3

Sophia Sun
Tuesday, Sep 16, 2025

Some due dates

- Homework 2 is due yesterday
- Homework 3 will release this Thursday
-
- Project 1 beta: September 23, Tuesday (Next Tueday)
- Project 1 final: October 2, Thursday

Coding warm up & QA time

BIT HACKS

Tips for Bit Manipulation

- Use the appropriate literals
  - A plain 1 is a 32-bit integer. 1 << 63 can give unexpected results.
  - 1ULL means 1 unsigned long long which is uint64_t
    - 00000000 00000000 00000000 00000000
    - 00000000 00000000 00000000 00000001
- Never shift by more than the number of bits in the number
  - ((uint64_t) i) >> 64 is undefined behavior (UB)
- Never shift by a negative amount
  - Also UB

Practice Question

What does the following code do?

```
uint64_t bithack_1(uint64_t x) {
  return (x & (x - 1)) == 0;
}
```

A Returns the index of the lowest 1-bit in `x`.

B Returns a 1 in the position of the least-significant 1 in `x` and a 0 in all other bit positions.

C Returns 1 if `x` is a power of 2, and 0 otherwise.

D Returns 1 if `x` is 0 or a power of 2, and 0 otherwise.

E Returns 1 if `x` is greater than 1, and 0 otherwise.

F None of the above.

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

Why?

1. We know x -1 will find the first set bit on x, when scanning from the least significant bit and invert all bits till the found bit.
   Ex: 1100**1000** – 1 = 1100**0111**
2. Now, let the number be x= abcd100…
3. x – 1 = abcd011….
4. x & x -1 = abcd000…
5. The above number will be 0 if abcd are all 0.
6. Which means x must be of the form 00..010… (only 1 set bit)
7. Which represents all and only powers of 2 or 0

Practice Question

This bit trick resembles the bit trick from lecture for computing $2^{\lceil \lg n \rceil}$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, `0xDEADBEEF >> 12` yields `0x000DEADB`, and `rightrotate(0xDEADBEEF, 12)` produces `0xEEFDEADB`. For each of the assertions in the following the code, determine if the assertion would always succeed, if the assertion would always fail, or if the assertion would sometimes succeed and sometimes fail.

```
void bithack(uint64_t x) {
  uint64_t r = x - 1;
  r |= rightrotate(r, 1);
  r |= rightrotate(r, 2);
  r |= rightrotate(r, 4);
  r |= rightrotate(r, 8);
  r |= rightrotate(r, 16);
  r |= rightrotate(r, 32);
  r++;

  assert(r < 0);  // A.
  assert(r < 1);  // B.
  assert(r < 2);  // C.
  assert(r < 4);  // D.
}
```

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

Practice Question

This bit trick resembles the bit trick from lecture for computing $2^{\lceil \lg n \rceil}$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, `0xDEADBEEF >> 12` yields `0x000DEADB`, and `rightrotate(0xDEADBEEF, 12)` produces `0xEEFDEADB`. For each of the assertions in the following the code, determine if the assertion would always succeed, if the assertion would always fail, or if the assertion would sometimes succeed and sometimes fail.

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

```
void bithack(uint64_t x) {
  uint64_t r = x - 1;
  r |= rightrotate(r, 1);
  r |= rightrotate(r, 2);
  r |= rightrotate(r, 4);
  r |= rightrotate(r, 8);
  r |= rightrotate(r, 16);
  r |= rightrotate(r, 32);
  r++;

  assert(r < 0);  // A.
  assert(r < 1);  // B.
  assert(r < 2);  // C.
  assert(r < 4);  // D.
}
```

Trace: Take x = 2 => r =1
r = 100…01
r = 11100….01
r = 111111…001
.
.
r = 1111….1
r++ will result in 0

We can easily see that if any bit in x-1 is 1, then the final r will be 0.
Otherwise, the final r is 1.
So, it tests if x-1 is 0, that is if x is 1.

Practice Question

This bit trick resembles the bit trick from lecture for computing $2^{\lceil \lg n \rceil}$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, `0xDEADBEEF >> 12` yields `0x000DEADB`, and `rightrotate(0xDEADBEEF, 12)` produces `0xEEFDEADB`. For each of the assertions in the following the code, determine if the assertion would always succeed, if the assertion would always fail, or if the assertion would sometimes succeed and sometimes fail.

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

```
void bithack(uint64_t x) {
  uint64_t r = x - 1;
  r |= rightrotate(r, 1);
  r |= rightrotate(r, 2);
  r |= rightrotate(r, 4);
  r |= rightrotate(r, 8);
  r |= rightrotate(r, 16);
  r |= rightrotate(r, 32);
  r++;

  assert(r < 0);  // A.
  assert(r < 1);  // B.
  assert(r < 2);  // C.
  assert(r < 4);  // D.
}
```

Trace: Take x = 2 => r =1
r = 100…01
r = 11100….01
r = 111111…001
.
.
r = 1111….1

A – Never
B – Sometimes
C – True
D - True

r++ will result in 0
We can easily see that if any bit in x-1 is 1, then r is 0.
Otherwise, r is 1
So, it tests if x-1 is 0, that is if x is 1.

Project 1: continue

Row-column-row algorithm

- An algorithm for rotating block of bits
- Basic idea:
  - rotate row r left by r + 1
  - rotate column c down by c + 1
  - rotate row r left by r

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

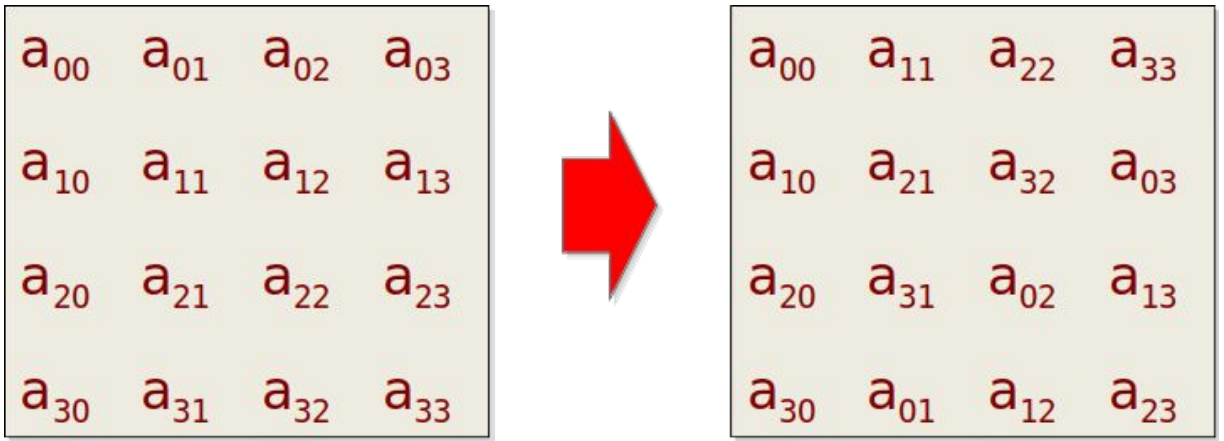
	0	1	2	3
0	M	I	E	A
1	N	J	F	B
2	O	K	G	C
3	P	L	H	D

How to implement row rotation?

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

How to implement column rotation?

Input: $N \times N$ matrix of bits, stored in row-major order.
Goal: Circularly rotate ith column of bits up i rows.



In the example that follows, we have $N = 32$. Each row is stored in a 32-bit word, with column 0 in the most-significant bit.

Naive approach

```
const uint32_t N = 32;
const uint32_t mask = 1 << (N-1);
uint32_t A[N];
for (int i = 0; i < N; i++){
  uint32_t col = 0;
  // gather bits in column i
  for (int j = 0; j < N; j++)
    col = col | (((A[j] << i) & mask) >> j);
  // rotate bits in column i
  col = (col << i) | (col >> (N - i));
  // put column i back
  for (int j = 0; j < N; j++)
    A[j] = (A[j] & ~(mask >> i)) |
        (((col << j) >> i) & (mask >> i));
}
```

Work:
$\Theta(N^2)$.

Divide-and-Conquer approach

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Divide-and-Conquer approach

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

Rotate columns 2 & 3 down by 2, same as rotate up by 2

Divide-and-Conquer approach

	0	1	2	3
0	A	B	K	L
1	E	F	O	P
2	I	J	C	D
3	M	N	G	H

Rotate columns 2 & 3 down by 2, same as rotate up by 2

Divide-and-Conquer approach

	0	1	2	3
0	A	B	K	L
1	E	F	O	P
2	I	J	C	D
3	M	N	G	H

Rotate columns 1 & 3 down by 1, same as rotate up by 3

Divide-and-Conquer approach

	0	1	2	3
0	A	N	K	H
1	E	B	O	L
2	I	F	C	P
3	M	J	G	D

Rotate columns 1 & 3 down by 1, same as rotate up by 3

Divide-and-Conquer approach

	0	1	2	3
0	A	N	K	H
1	E	B	O	L
2	I	F	C	P
3	M	J	G	D

Rotate all columns down by 1, same as rotate up by 3

Divide-and-Conquer approach

	0	1	2	3
0	M	J	G	D
1	A	N	K	H
2	E	B	O	L
3	I	F	C	P

Rotate all columns down by 1, same as rotate up by 3

Divide-and-Conquer approach

	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L
3	M	N	O	P

	0	1	2	3
0	M	J	G	D
1	A	N	K	H
2	E	B	O	L
3	I	F	C	P

Result: rotate all columns c down by c + 1, same as rotate up by N - (c + 1)

Divide-and-conquer approach

```
uint32_t A[32], B[32]; // use B as scratch space

// rotate columns 16...31 down 16 positions
uint32_t stay_mask = 0xFFFF0000; // columns that don't move

for (int j = 0; j < 32; j++)
  B[j] = (A[j] & stay_mask) | (A[(j-16+32) % 32] & ~stay_mask);

// rotate columns 8..15 and 24...31 down 8 positions
stay_mask = 0xFF00FF00;

for (int j = 0; j < 32; j++)
  A[j] = (B[j] & stay_mask) | (B[(j-8+32) % 32] & ~stay_mask);
```

Work: $\Theta(N \lg N)$

Full steps

32-bit matrix example

Coding time!