Bit operators: Be careful!

|  |  |  |
| --- | --- | --- |
| **bang - Compute !x without using !**  int bang(int x) {      int sign = x >> 31;  //if x is positive or zero, this will be 0.  if this is negative, this will be -1.      int msign = (~x +1)>>31;  //if x is positive, this will be -1.  If x is negative or zero, this will be 0.      return (sign|msign) + 1 ;      ///alternatively, we can do ~(sign|msign) & 1<<31;  } | **Bitcount**  int bitCount(int x) {      int count;      int shift1 = (85<<24) | (85<<16)|(85<<8)| 85;//0x55555555      int shift2 = (51<<24) | (51<<16)|(51<<8)| 51;      int shift3 = (15<<24) | (15<<16)|(15<<8)| 15;      int shift4 = (255<<16)|255;      int shift5 = (255<<8)|255;      count = (x & shift1) + ((x >> 1) & shift1);      count = (count & shift2) + ((count >> 2) & shift2);      count = (count & shift3) + ((count >> 4) & shift3);      count = ((count + (count >> 8)) & shift4);       count = ((count + (count >> 16)) & shift5);    return count; | c = (v & 0x55555555) + ((v >> 1) & 0x55555555);  c = (c & 0x33333333) + ((c >> 2) & 0x33333333);  c = (c & 0x0F0F0F0F) + ((c >> 4) & 0x0F0F0F0F);  c = (c & 0x00FF00FF) + ((c >> 8) & 0x00FF00FF);  c = (c & 0x0000FFFF) + ((c >> 16)& 0x0000FFFF);  So if I have number 395 in binary 0000000110001011 (0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 1)  After the first step I have: 0000000101000110 (0+0 0+0 0+0 0+1 1+0 0+0 1+0 1+1) = 00 00 00 01 01 00 01 10  In the second step I have:    0000000100010011 ( 00+00   00+01   01+00   01+10 ) = 0000 0001 0001 0011  In the fourth step I have:    0000000100000100 (   0000+0001   0001+0011   ) = 00000001 00000100  In the last step I have: |
| **bitOr** - x|y using only ~ and &    int bitOr(int x, int y) {      int bit = ~(~x & ~y);      return bit; |
| \* bitRepeat - repeat x's low-order n bits until word is full.   \*   Can assume that 1 <= n <= 32.   \*   Examples: bitRepeat(1, 1) = -1   \*             bitRepeat(7, 4) = 0x77777777   \*             bitRepeat(0x13f, 8) = 0x3f3f3f3f   \*             bitRepeat(0xfffe02, 9) = 0x10080402   \*             bitRepeat(-559038737, 31) = -559038737   \*             bitRepeat(-559038737, 32) = -559038737 | 0000000000000101 (   00000001+00000100   )  which is equal to 5, thecorrect result |
| /\* DONE   \* **fitsBits** - return 1 if x can be represented as an   \*  n-bit, two's complement integer.    int fitsBits(int x, int n) {      int signbit = x >> 31;      int ifnegative = ~x & signbit; //should result in ~x if negative      int ifpositive = x & ~signbit; //should result in x is positive        return !((ifpositive + ifnegative) >>(n+~0));       }  int logicalShift(int x, int n) {        int check = x>> n;      int mask = (((x>>31)<<31)>>n)<<1;      return check^mask;  } | int bitRepeat(int x, int n) {      int possible = !(n/32); //returns 0 if n is 32 or greater //this is 2, right here      int ncheck = n\*possible;                                    //3, total      int one = -1;////then 4      int mask = x & ~((one << (ncheck-1))<<1); //the one on the end is to account for the minus 1      int storage = (mask<<ncheck)| mask;        n\*=2; //1      possible = !(n/32); //returns 0 if n is 32 or greater//3      ncheck = n\*possible; //4      storage = (storage<<ncheck)|(storage);        n\*=2; //1      possible = !(n/32); //returns 0 if n is 32 or greater//3      ncheck = n\*possible; //4      storage = (storage<<ncheck)|(storage); //8      // storage<<ncheck | storage   -->  00000010 + 00001000 -- > 00001010      //                storage<<ncheck | storage      --> 00001010 + 10100000 ---> 10101010      n\*=2;      possible = !(n/32); //returns 0 if n is 32 or greater      ncheck = n\*possible;      storage = (storage<<ncheck)|(storage);        n\*=2;      possible = !(n/32); //returns 0 if n is 32 or greater      ncheck = n\*possible;      storage = (storage<<ncheck)|(storage);       return storage; | int **getByte**(int x, int n) {    return (x>>(n<<3)) & 255;  }  /\* DONE   \* isLessOrEqual - if x <= y  then return 1, else return  int isLessOrEqual(int x, int y) {        int isYouDifferent = !(x>>31)^!(y>>31); //if different signs, return true;      int isXDifferent = isYouDifferent & !!(x>>31); //if x is negative and y is positive, return true;      int differenceSign = !isYouDifferent & !((y+ ~x +1)>>31); //0 if is bigger than y, 1 if they're the same.        return isXDifferent | differenceSign;    }  **isPositive -** return 1 if x > 0, return 0 otherwise    int isPositive(int x) {      int isZero = !x; //if this equals 0, then return one.  if it equals literally anything else, return 0;      int check = !(x>>31); //if this is positive, than this is 0.    return check + ~isZero + 1;  } |

How to add using only bitwise operators:

int Add(int x, int y)

{

    // Iterate till there is no carry

    while (y != 0)

    {

        // carry now contains common set bits of x and y

        int carry = x & y;

        // Sum of bits of x and y where at least one of the bits is not set

        x = x ^ y;

        // Carry is shifted by one so that adding it to x gives the required sum

        y = carry << 1;

    }

    return x;

}

1(10 minutes).Why do CPU caches interact so poorly with GNU/Linux exceptionhandlers? Give a brief example.

Exceptions are...exceptional(rare),so data and instructions from exception handlers are likely not cached.In fact,since exceptions are run in kernel mode, the data accessed will be things from the kernel’s stack/heap/data and will likely require code that could not ever be accessed in the userspace (except in the case where the same exceptionhas previously occurred).

2.The book says that the ‘cltq’ instruction is just a shorthand for ‘movslq %eax, %rax’.And yet if you put the following assembly language program:

|  |  |
| --- | --- |
| .globlfuna funa:  pushq %rbx movq %rsi,%rbx call g cltq addq %rbx,%rax popq %rbx ret  .globlfunb funb:  pushq %rbx movq %rsi,%rbx call g  movslq%eax,%rax addq %rbx,%rax popq %rbx ret  into the file fun.s and run these shell commands: gcc − O2 − c fun.s  objdump − d  fun.o  the resulting output will contain something like: | 0: 53 push %rbx 1: 48 89 f3 mov %rsi,%rbx 4: e8 00 00 00 00 callq 9<funb+0x9> 9: 48 98 cltq  b: 48 01 d8 add %rbx,%rax  e: 5b pop %rbx  f: c3 retq  0000000000000010<funb>:  10: 53 push %rbx  11:48 89 f3 mov %rsi,%rbx  14:e8 00 00 00 00 callq 19<funa+0x9>  19:48 63 c0 movslq %eax,%rax  1c: 48 01 d8 add %rbx,%rax  1f:5b pop %rbx 20: c3 retq    push %rbx  so the two instructions do differ. |

Explain the seeming disagreemen tbetween the book and theobjdump output.

ctlq is not a true shorthand for movslq %eax,%rax,since thatimplies that ctlq instructions will always translate to movslq %eax,%rax. As seen in the code, both instructions are unique, and ctlq is simply a shorter instruction with the same effect.

Source code that corresponds to above:

longf (arg1,longarg2,...){ returng(...)+arg2; }

...where arg1 can be any integer style datatype (ie.not a double),and f() and g()can take any number of arguments. Due to the fact that after calling g(),the value in %eax is sign extended into %rax,it is inferred that g()returns an int.As a result,the function body may also be some variation of the following form:

int ret =g(...); long lret =(long)ret; returnlret+arg2;

Due to C’stype casting rules however(the operands in an expression are converted to the type of the highest ranking type in the expression), having “return g(...)+arg2” will implicitly convert the return type of g(...)into a long.

2c(10minutes).What symbol tableand relocation entries should appear in fun.o? Briefly explain.

There are three globally available names that are known to this snippet of code:

g(),funa(),funb()

As a result the symbol table will contain entries for g, funa, and funb. However, in this code snippet, the only references to these globally available symbols are when funa and funb call function g. As a result, there will be two relocation entries, one for where funa calls g and one for where funb calls g.

3 (15minutes). Where are PTEs most commonly used by the Nehallem microarchitecture implementation fo the Core 7? Where else can PTEs appear

movq (%rsp), %rsp

looks very much like code that will walk up through stack frames. This particular instruction grabs the quadword pointed to by the current stack pointer, and loads it into the stack pointer, overwriting it.

Each step of those five, adds neighbouring bits together in groups of 1, then 2, then 4 etc. The method is based in divide and conquer.

In the first step we add together bits 0 and 1 and put the result in the two bit segment 0-1, add bits 2 and 3 and put the result in the two-bit segment 2-3 etc...

In the second step we add the two-bits 0-1 and 2-3 together and put the result in four-bit 0-3, add together two-bits 4-5 and 6-7 and put the result in four-bit 4-7 etc...

|  |  |  |
| --- | --- | --- |
| unsigned f2u(float x){  union u{  float f;  unsigned u;  };  union u u1;  u1.f = x;  return u1.u;} | unsigned u2f(unsigned y){  union u{  float f;  unsigned u;  };  union u u1;  u1.u = y;  return u1.f;} | 1b: If X is a NaN, then the expression will evaluate to 0. Hence, No.  1c.  Y1 = +0  Y2 = -0 |

1d. u2f: A (2 points); other answer (0 point)

f2u: C,F (2 points); C (1 point); F (1 point); other answer (0 point)

1e. B: invalid, you’re trashing your return address by moving stuff there

D: float D(unsigned x){return x;}

E: unsigned E(float t) {return t;}

2b. Kaby Lake has 2 FPUs and that is the bottleneck (see dataflow diagram). Do fmul and fadd of two consecutive iterations in parallel.

2c. In direct-mapped, collision = crash

2-way : if if both arrays map to the same location, the 2 load boxes in the dataflow diagram will not collide

3

Multiplexing: Good for activities with delays. For example, applications which need to wait for

user inputs.

Process Parallelism: Good for loosely coupled systems. Different parts run independently.

Thread Parallelism: Good for systems where different parts share information like in web servers. (For process and thread you were expected to give any logical example with explanation of why process or thread suits it better)

SIMD: Single instruction multiple data. Works best for operations on arrays, matrices where same instruction has to be performed on all data. ML is an application.

Instruction Level Parallelism: Can be used in most cases for optimisation where there are less data dependencies and not too many branches. (This is useful whenever the other four are not.)

4.

Pros for exclusive: More capacity, hence more data can be cached; Less redundancy of data; Holds one copy for each data value

Cons for exclusive: More management to keep track of uniqueness; Different cache lines are difficult to support; Have to search L1 and L2 to get hold of specific data

Pros for inclusive: Simpler in design; Different cache line sizes are supported (like 16 for L1 and 64 for L2, resulting in faster access); Faster data access

Cons for inclusive: More time to cache; Cache may get filled quickly with data being replicated: need to maintain correct state in both caches

5

Q.5 a. The callee saved register RBX is being stomped on. %rbx value being altered.

Q.5 b. For Stack Alignment. Need memory for PUTS function. Partial credit if either of

the 2 points missing.

Q.5 c. Call pthread\_join with the appropriate arguments. Possible solution to resolve the

bug would be having an array of thread IDs, initialized by the loop that calls pthread\_create, and then used by a later loop that calls pthread\_join. Partial marks for stating that the output string is being printed in random (maybe even interleaved) order (race condition). Partial credit also for stating that threads are not being cancelled or no exit.