# CSE 2421

Integer Multiplication and Division

## Base 10 Multiplication Review (obviously)

a (multiplicand) * b (multiplier)

```
Consider a = 123, b = 123.

              123
            x 123
           ======
              369  (this is 123 * 3)
             246   (This is 123 * 2 shifted left 1 place)
     +      123    (This is 123 * 1 shifted left 2 places)
          ========
            15129
```

# Binary (Base 2) Multiplication

Now consider:

a (multiplicand) * b (multiplier) where a = 1011 (11 decimal), b = 1110 (14 decimal)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 0 | 1 | 1 | (this is 11 in decimal) |
| | | | x | 1 | 1 | 1 | 0 | (this is 14 in decimal) |
| | | | ================== | | | | | |
| | | | | 0 | 0 | 0 | 0 | (this is 1011 x 0) |
| | | | 1 | 0 | 1 | 1 | | (this is 1011 x 1, shifted one position to the left) |
| | | 1 | 0 | 1 | 1 | | | (this is 1011 x 1, shifted two positions to the left) |
| + | 1 | 0 | 1 | 1 | | | | (this is 1011 x 1, shifted three positions to the left) |
| | ========================== | | | | | | | |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | (this is 154 in decimal) |

This is the same process that we use for decimal multiplication!

One could say that binary multiplication is actually easier because each row is either just a “shifted copy” of the multiplicand or it’s 0.

## Hardware for integer multiplication (Simplest version)

•Suppose we want the CPU to multiply two operands:

a (multiplicand) * b (multiplier)

•Typically, in the simplest version of multiplication hardware, 4 additional registers are used:

-Multiplicand register (contains a copy of the multiplicand)

-Multiplier register (contains a copy of the multiplier)

-Shifted multiplicand register

-Result register

## Multiplication hardware cont

•For n bit multiplication, we can denote the bits in the multiplier as $b_0$ (least significant bit) to $b_{n-1}$ (most significant bit)

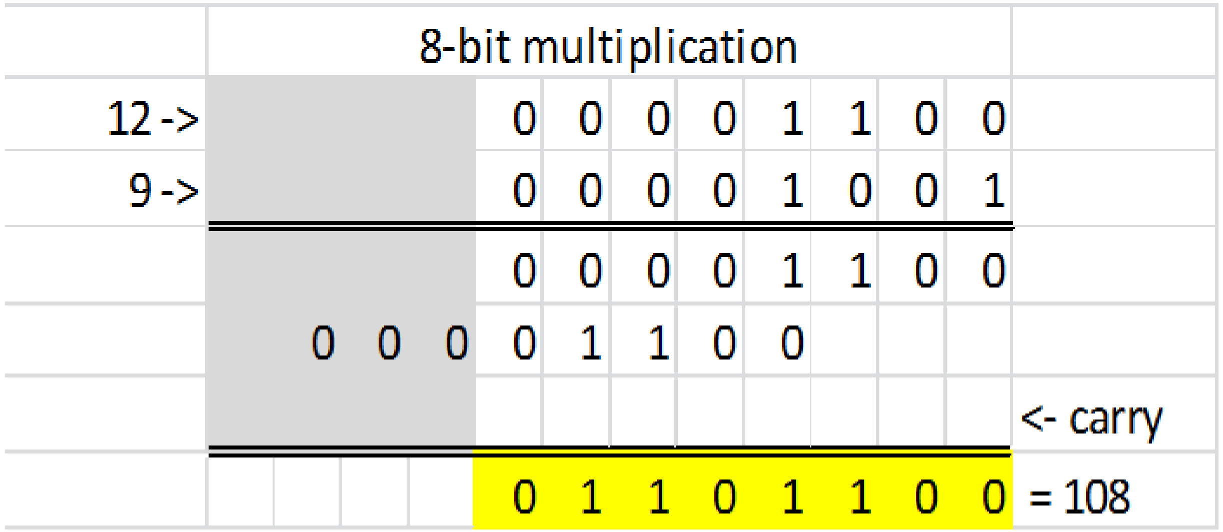
•The hardware initializes the result register to 0

•Then, the hardware does the following (pseudo-code):

```
for (j = 0; j <n; j++) {
    if ($b_j$ == 1) copy multiplicand register contents, left shifted by j bits, to shifted multiplicand register;
    else copy 0 to shifted multiplicand register;
    Add shifted multiplicand register to result register;
}
```

# Integer multiplication

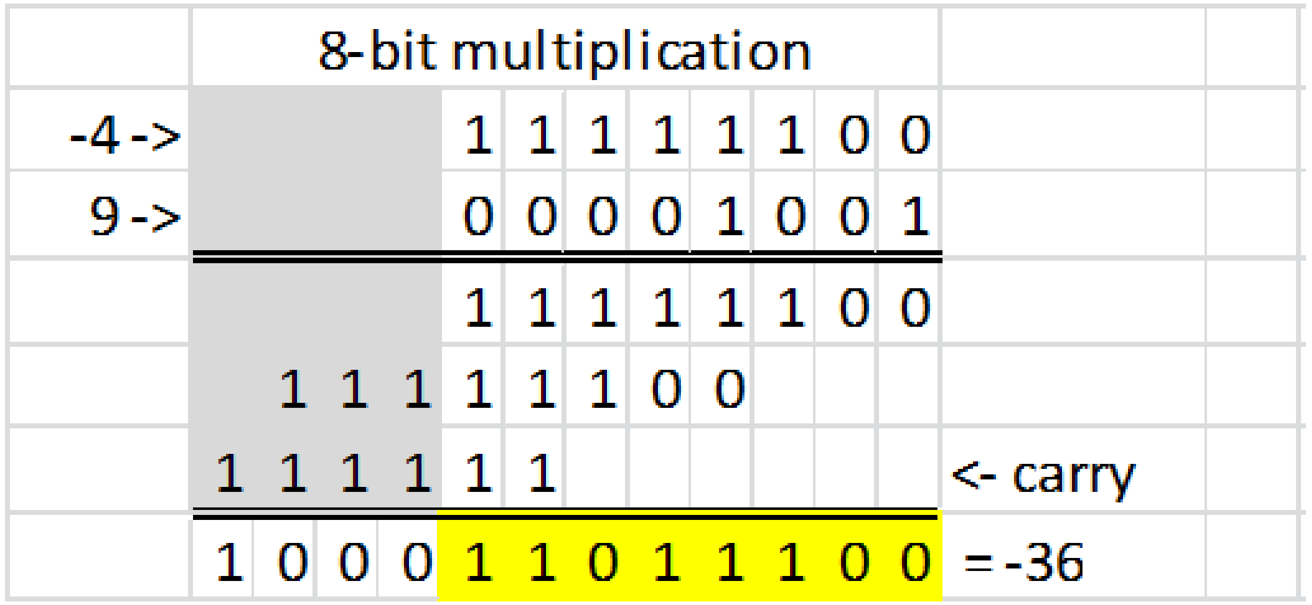
- B2U 8-bit range → 0 to 255



| | 8-bit multiplication | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 -> | | | | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | |
| 9 -> | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | |
| | | | | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | |
| | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | | | | |
| | | | | | | | | | | | | | <- carry |
| | | | | | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | = 108 |

## Integer multiplication

- B2T 8-bit range → -128 to 127



| | 8-bit multiplication | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -4 -> | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | |
| 9 -> | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | |
| | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | |
| | | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | | | | |
| | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | <- carry |
| | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | = -36 |

## Integer multiplication (cont)

- Unsigned i.e. simple binary
  - For x and y, each with the same width (w)
  - x*y yields a w-bit value given by the w bits of the integer product
    - Result interpreted as an unsigned value
    - Overflow occurs if the result will not fit in w bits; to store any possible result for multiplication of two w bit values, 2w bits are needed.

- Signed = similar, but result interpreted as signed value

  Overflow can occur under same conditions as above.

# Integer multiplication (4 bits)

| binary | unsigned | two's comp | result |
|---|---|---|---|
| 0111*0011 | 7*3=21=0001 0101 | same | **0101** |
| | 21 mod 16 = 5 | same | |
| 1001*0100 | 9*4=36=0010 0100 | -7*4=-28=1110 0100 | **0100** |
| fyi | 36 mod 16 = 4 | -28 mod 16 = 4 | |
| 1100*0101 | 12*5=60=0011 1100 | -4*5=-20=1110 1100 | **1100** |
| | 60 mod 16 = 12 | -20 mod 16 = -4 | |
| 1101*1110 | 13*14=182=1011 0110 | **-3*-2=6=0000 0110** | **0110** |
| | 182 mod 16 = 6 | 6 mod 16 = 6 | |
| **1111*0001** | **15*1=15=0000 1111** | **-1*1=-1=1111 1111** | **1111** |
| | 15 mod 16 = 15 | -1 mod 16 = -1 | |

## The world has changed

- The imul instruction on current intel architectures takes 1 clock cycle and has latency of about 3 clock cycles.
- A shift is 1 clock cycle with a latency of 1 clock cycle
- The transistor budget for modern CPUs are *huge* compared to previous generations of chips. It takes a *lot* of adders to make a large, fast multiplier
- The intel 8086 took 100 clock cycles for imul
- Other chips may have high costs to multiply compared to the cost of shifting
- ARM 9 chips need 2-4 clock cycles to multiply

## Multiply by constants (known at compile time)

- First case: Multiplying by a power of 2
  - -Power of 2 represented by k
  - -This is an optimization for the C compiler: left shifting can be used to replace multiplication, which ~~is~~ *used to be* a *much* slower operation on small chips.
    - (8 bit Arduino does 8 bit multiply in 1-2 clocks, 16 bit in 6 clocks, 32 bit in 12 clocks)
  - -So k zeroes added in to the right side of x
    - -Shift left by k: x<<k
  - -Overflow issues the same as x*y
- What is x*4 where x = 5?
- `x = 5 = 00000101`
- `4 = 2k, and k = 2`
- `x<<k = 00010100 = 20`

# Multiply by constants (known at compile time)

- First case: Multiplying by a power of 2
  - -Power of 2 represented by k
  - -This is an optimization for the C compiler: left shifting can be used to replace multiplication, which ~~is~~ *used to be* a *much* slower operation on small chips.
    - (8 bit Arduino does 8 bit multiply in 1-2 clocks, 16 bit in 6 clocks, 32 bit in 12 clocks)
  - -So k zeroes added in to the right side of x
    - -Shift left by k: x<<k
  - -Overflow issues the same as x*y
- What is x*4 where x = 5?
- `x = 5 = 00000101`
- `4 = 2k, and k = 2`
- `x<<k = 00010100 = 20`
- What if x = -5 = 1111 1011
  - -We can shift a maximum of 4 times before we lose the sign bit

## Multiply by constants

- General case
  - -Every binary value is an addition of powers of 2
  - -Shifts, adds, and subtracts generally take 1 clock cycle, **many** fewer clock cycles than multiplication on chips with a constrained transistor count (ARM takes 2-4 cycles)
  - -This is an optimization for the C compiler
  - -For the compiler to do this kind of optimization, one operand must be a constant known at compilation time
  - -Has to be a run of consecutive1's to work (to save significant CPU cycles)
    - -Where n = position of leftmost 1 bit in the run and m=the rightmost (with bit positions designated as w-1 for msb to 0 for lsb)
- Example: suppose for constant k,  k = 7 = 0111  /* n=2, m=0 */
- Using the 1's in the constant (with addition):  $2^2+2^1+2^0 = 7$
  - (x<<n)+(x<<n-1) + … + (x<<m)  /* use the 1's */
- Using the 0 in the constant (with subtraction) : $2^3 – 2^0 = 8 – 1 = 7$
  - (x<<n+1) - (x<<m)  /* use the 0 */

## Multiply by constants

```
x = 5;          /* 0101 */
x*7 = 35        /* 7=0111: n = 2 and m = 0 */
x<<2 + x<<1 + x<<0
  00010100
+ 00001010
+ 00000101
  --------
  00100011 = 35
```

## Multiply by constants

```
x = 5;          /* 0101 */
x*7 = 35        /* 7=0111: n = 2 and m = 0 */
x<<2 + x<<1 + x<<0
  00010100
+ 00001010
+ 00000101
  00100011 = 35
OR
x<<3 - x<<0
  00101000
- 00000101
  00100011 = 35
```

## Multiply by constants (cont)

- What if the bit position n in the constant is the most significant bit?
- Since the formula with subtraction is (x<<n+1) – (x<<m) and shifting n+1 times gives zero, then the formula gives –(x<<m)
- Example: Assuming 4 bit values $2^3$ is the most significant bit (that is, the msb position is 3). If we were to shift something 4 times it effectively 0’s all 4 bits

## Multiply by constants (cont)

•Assuming 4 bit values

```
x = 5; /*0101*/
y=12;  /*1100 so  n=3, m=2 */
z = x*y;
x<<4 - x<<2
  0000       /*8 bits: 01010000 */
- 0100       /*8 bits: 00010100 */
  1100 = -4  /*8 bits: 00111100  - CORRECT!*/
```

**Overflowed**, because information in high-order bits was lost!

We will look more at overflow due to shifting below.

# Multiply by constants (cont)

•What if y is negative?

-Remember original formula (with subtraction): (x<<n+1) – (x<<m) where n=position of leftmost and m=position of rightmost 1 in the constant

-**Negate formula for negatives** (**multiply by –1**), and **negate the constant** (that is, **treat the constant as a positive**):

Formula for negatives: (x<<m) – (x<<n+1)
Instead of subtracting, we can take the 2's complement of x<<n+1, and add:
(x<<m) + (– (x<<n+1))

-Example: y = -6 x= 5 /*Assuming 8-bits*/

x * y

y = 0000 0110 = +6 /*treat constant as positive: n=2 and m=1*/

$2^1 - 2^3 = 2 – 8 = -6$

(x<<1) – (x<<3)

```
  0000 1010 (x=5, shifted 1 left)
– 0010 1000 (x=5, shifted 3 left)
  1110 0010  = -30
```

## Overflow for unsigned multiplication

- In CPUs, the operands and the result of the multiplication are normally stored in the same number of bits
- To eliminate the possibility of overflow for multiplication of w bit operands, we need 2w bits.
- If we only have w bits to store the result, can we describe when overflow will occur due to shifting? [Note that the addition operations may also cause overflow!]
- Using the same kind of notation that we used for the bits in constants above, if we designate the position of the most significant 1 in op1 as $op1_n$, and the position of the most significant 1 in op2 as $op2_n$, when will we get overflow due to shifting?
- Let’s think about the algorithm we described earlier – we need to answer two questions to describe the conditions under which overflow due to shifting will occur for unsigned operands.

## Overflow for unsigned multiplication

•Question 1:
•If the most significant bit in op1 is in position $op1_n$, and if operands and the result have w bit representations, how many bits can op1 be shifted left before overflow due to shifting occurs?

•Question 2:
•What is the relationship between $op2_n$ and the maximum number of bits that op1 is shifted left by the multiplication algorithm?

# Overflow for unsigned multiplication

- Question 1:
- If the most significant bit in op1 is in position $op1_n$, and if operands and the result have w bit representations, how many bits can op1 be shifted left before overflow due to shifting occurs?
    - Answer: $w - 1 - op1_n$
    - Example: $w=4$, $op1_n=2$ (0100), then $4-1-2 = 1$
      we can only shift left once; any more than that causes overflow
- Question 2:
- What is the relationship between $op2_n$ and the maximum number of bits that op1 is shifted left by the multiplication algorithm?
    - Answer: $op2_n$ is the maximum number of bits that op1 is shifted left

## Overflow for unsigned multiplication

•Conclusion: If $op2_n > w - 1 - op1_n$ , overflow due to shifting will occur

So if $op2_n > 1$ in our example, overflow occurs.

•Again, overflow may also occur due to addition, but the result above describes cases where overflow will *definitely* occur due to shifting.

## Example

- $w=8$, $0p_1=4$, $op_2=3$
- $w(8)$ – 1 – $op_1(4)$ = 3

## Example

- $w=8$, $0p_1=4$, $op_2=3$
- $w(8) - 1 - op_1(4) = 3$
- Is $op_2 > w - 1 - op_1$? $3 > 3$?

## Example

- $w=8$, $0p_1=4$, $op_2=3$
- $w(8) - 1 - op_1(4) = 3$
- Is $op_2 > w - 1 - op_1$? $3 > 3$?
- No overflow in this case.

## Dividing by powers of 2

- Even slower than integer multiplication
- Dividing by powers of 2 → right shifting
  - -There are 2 types of right shifting:
    - -Logical – unsigned: fill with 0’s
    - -Arithmetic – two’s complement: fill with copy of msb
- Integer division always truncates
  - -C float-to-integer casts round towards zero.
  - -Division by right shifting rounds *down*.
  - -These rounding errors generally accumulate

## Unsigned Integer Division

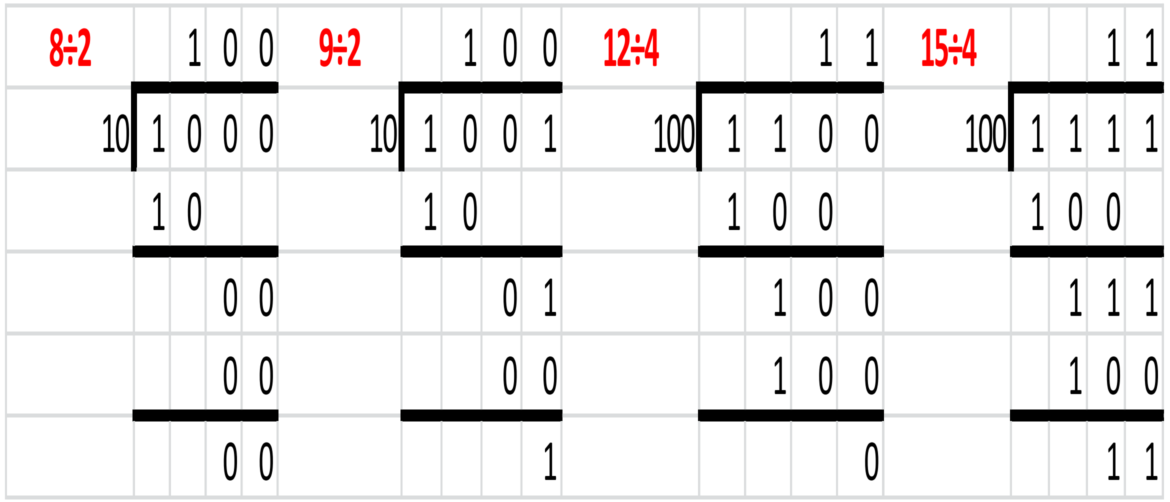
•Dividing by 2k

•Logical right shift by k (x>>k)

•Then rounding down (toward zero)

•Remember that logical shift will shift in zeroes to fill the most significant bits when the original bit pattern is shifted right

# Dividing by powers of 2



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8÷2 | | 1 | 0 | 0 | 9÷2 | | 1 | 0 | 0 | 12÷4 | | | 1 | 1 | 15÷4 | | | 1 | 1 |
| 10 | 1 | 0 | 0 | 0 | 10 | 1 | 0 | 0 | 1 | 100 | 1 | 1 | 0 | 0 | 100 | 1 | 1 | 1 | 1 |
| | 1 | 0 | | | | 1 | 0 | | | | 1 | 0 | 0 | | | 1 | 0 | 0 | |
| | | | 0 | 0 | | | | 0 | 1 | | | 1 | 0 | 0 | | | 1 | 1 | 1 |
| | | | 0 | 0 | | | | 0 | 0 | | | 1 | 0 | 0 | | | 1 | 0 | 0 |
| | | | 0 | 0 | | | | | 1 | | | | | 0 | | | | 1 | 1 |

x = 8 = 1000

y = 2 = 0010 = $2^1$

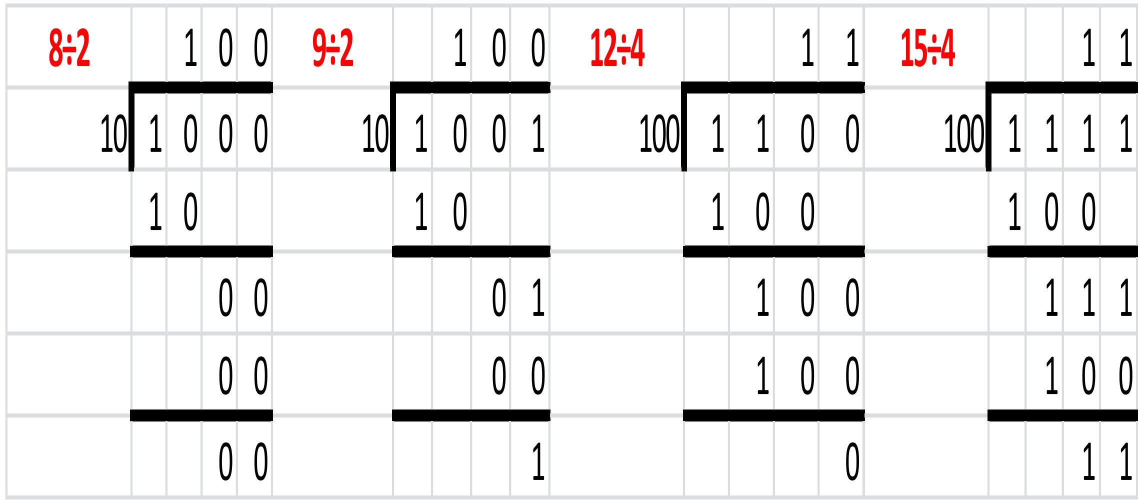
x >> 1 = 100 (x / y)

x = 9 = 1001

y = 2 = 0010 = $2^1$

x >>1 = 100 (x / y)

# Dividing by powers of 2



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8÷2 | | 1 | 0 | 0 | 9÷2 | | 1 | 0 | 0 | 12÷4 | | | 1 | 1 | 15÷4 | | | 1 | 1 |
| 10 | 1 | 0 | 0 | 0 | 10 | 1 | 0 | 0 | 1 | 100 | 1 | 1 | 0 | 0 | 100 | 1 | 1 | 1 | 1 |
| | 1 | 0 | | | | 1 | 0 | | | | 1 | 0 | 0 | | | 1 | 0 | 0 | |
| | | | 0 | 0 | | | | 0 | 1 | | | 1 | 0 | 0 | | | 1 | 1 | 1 |
| | | | 0 | 0 | | | | 0 | 0 | | | 1 | 0 | 0 | | | 1 | 0 | 0 |
| | | | 0 | 0 | | | | | 1 | | | | | 0 | | | | 1 | 1 |

x = 12 = 1100

y = 4 = 0100 = $2^2$

x >> 2  = 11

x = 15 = 1111

y = 4 = 0100 = $2^2$

x >>2 = 11

# Signed Integer Division

- Two’s complement
- Sign extend for arithmetic shift (fill with copy of msb)
- Rounds down (away from zero for negative results)
- -7/2 will yield -4 rather than -3
- `x = -82` `y = 2`$^k$

| k | Binary | | | | | | | | Decimal | $-82/2^k$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | -82 | -82.000000 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | -41 | -41.000000 |
| 2 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | -21 | -20.500000 |
| 3 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | -11 | -10.250000 |
| 4 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | -6 | -5.125000 |
| 5 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | -3 | -2.562500 |

Note rounding effect