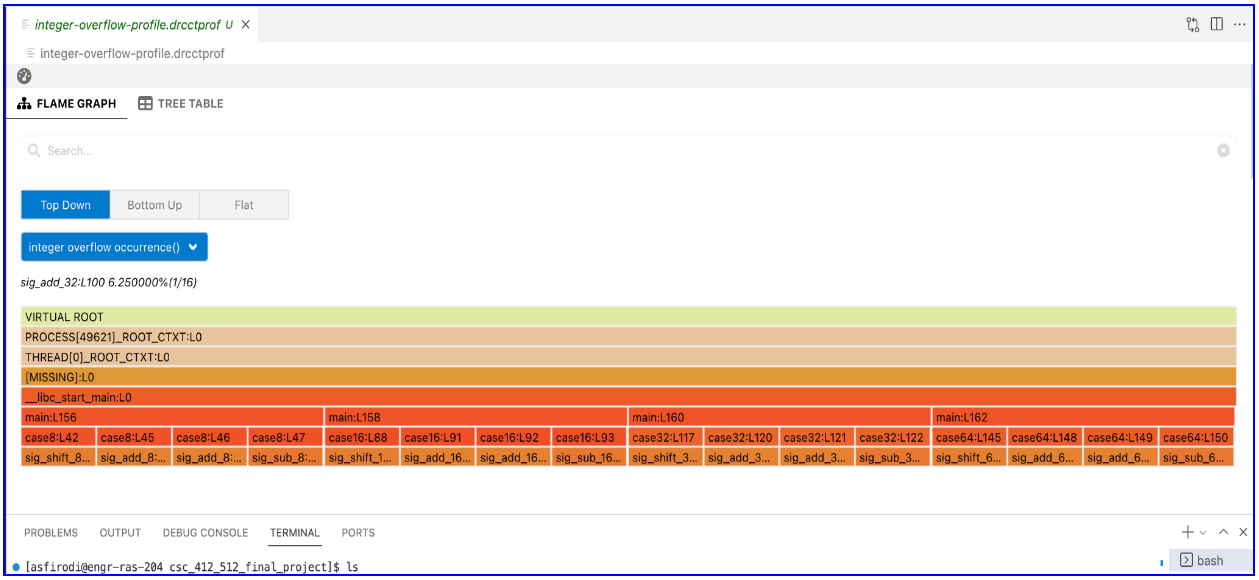
Project 3 - Extra Credits

Abhishek Firodiya (asfirodi) [591]

Q. 1) Visualization



The screenshot shows all 16 test cases - 4 for each case - where the integer overflow has happened.

Q. 2 and 3)

Intel x86 FLAGS register[1]						
Bit #	**Mask**	**Abbreviation**	**Description**	**Category**	**=1**	**=0**
FLAGS						
0	0x0001	CF	Carry flag	Status	CY(Carry)	NC(No Carry)
1	0x0002		Reserved, always 1 in **EFLAGS** [2][3]			
2	0x0004	PF	Parity flag	Status	PE(Parity Even)	PO(Parity Odd)
3	0x0008		Reserved[3]			
4	0x0010	AF	Adjust flag	Status	AC(Auxiliary Carry)	NA(No Auxiliary Carry)
5	0x0020		Reserved[3]			
6	0x0040	ZF	Zero flag	Status	ZR(Zero)	NZ(Not Zero)
7	0x0080	SF	Sign flag	Status	NG(Negative)	PL(Positive)
8	0x0100	TF	Trap flag (single step)	Control		
9	0x0200	IF	Interrupt enable flag	Control	EI(Enable Interrupt)	DI(Disable Interrupt)
10	0x0400	DF	Direction flag	Control	DN(Down)	UP(Up)
11	0x0800	OF	Overflow flag	Status	OV(Overflow)	NV(Not Overflow)
12-13	0x3000	IOPL	I/O privilege level (286+ only), always 1[clarification needed] on 8086 and 186	System		
14	0x4000	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System		
15	0x8000		Reserved, always 1 on 8086 and 186, always 0 on later models			
EFLAGS						
16	0x0001 0000	RF	Resume flag (386+ only)	System		
17	0x0002 0000	VM	Virtual 8086 mode flag (386+ only)	System		
18	0x0004 0000	AC	Alignment check (486SX+ only)	System		
19	0x0008 0000	VIF	Virtual interrupt flag (Pentium+)	System		
20	0x0010 0000	VIP	Virtual interrupt pending (Pentium+)	System		
21	0x0020 0000	ID	Able to use CPUID instruction (Pentium+)	System		
22-31	0xFFC0 0000		Reserved	System		
RFLAGS						
32-63	0xFFFF FFFF… …0000 0000		Reserved			

Fig : Flags (https://en.wikipedia.org/wiki/FLAGS_register#FLAGS)

We are interested in the Overflow flag(11), the Carry Flag(0) and the Sign Flag(7)

For example, adding 127 with 127 using 8-bit registers gives 254, which using 8-bit arithmetic is 1111 1110 binary (two's complement of -2).
This shows that two positive numbers are added to get a negative number. So, we can say that overflow happens when we get a negative sum of positive operands or vice versa.

So, we can see that the overflow flag is set when the most significant bit i.e. the sign bit is changed by adding two numbers with the same sign or subtracting two numbers with opposite signs.
[Overflow cannot occur when the sign of either two addition operands are different or two subtraction operands are the same.
For example, $127 + -128 = -1$, $-128 - -128 = 0$. Here, there is no overflow.]

"When binary values are interpreted as unsigned numbers, the overflow flag is ignored. One of the advantages of two's complement arithmetic is that the addition and subtraction operations do not need to distinguish between signed and unsigned operands. For this reason, most computer instruction sets do not distinguish between signed and unsigned operands, generating both (signed) overflow and (unsigned) carry flags on every operation, and leaving it to following instructions to pay attention to whichever one is of interest.
Internally, the overflow flag is usually generated by an XOR of the internal carry into (0) and the sign bit (7)" (Reference: https://en.wikipedia.org/wiki/Overflow_flag)

So, as for the algorithm, instead of using relational operators, we can just check if 11th MSB is 1 or not (for addition and subtraction) and if XOR of 0th and 7th MSB is 1 or not.

Results for test cases from [test1.c]() —

case 8:

```
shl 1 7 -> 0000000000000000000000000000000000000000000000000000001010000010
add 64 64 -> 0000000000000000000000000000000000000000000000000000101010000010
add -64 -65 -> 0000000000000000000000000000000000000000000000000000101000000011
sub -64 65 -> 0000000000000000000000000000000000000000000000000000101000010010
```

case 16:

```
shl 1 15 -> 0000000000000000000000000000000000000000000000000000001010000110
add 16384 16384 -> 0000000000000000000000000000000000000000000000000000101010000110
add -16384 -16385 -> 0000000000000000000000000000000000000000000000000000101000000111
sub -16384 16385 -> 0000000000000000000000000000000000000000000000000000101000010110
```

case 32:

```
shl 1 31 -> 0000000000000000000000000000000000000000000000000000001010000110
add 1073741824 1073741824 ->
0000000000000000000000000000000000000000000000000000101010000110
add -1073741825 -1073741824 ->
0000000000000000000000000000000000000000000000000000101000000111
sub -1073741824 1073741825 ->
0000000000000000000000000000000000000000000000000000101000010110
```

case 64:

```
shl 1 63 -> 0000000000000000000000000000000000000000000000000000001010000110
add 4611686018427387904 4611686018427387904 ->
0000000000000000000000000000000000000000000000000000101010000110
add -4611686018427387905 -4611686018427387904 ->
0000000000000000000000000000000000000000000000000000101000000111
sub -4611686018427387904 4611686018427387905 ->
0000000000000000000000000000000000000000000000000000101000010110
```

Some cases with no Overflow —

```
add 64 1 -> 0000000000000000000000000000000000000000000000000000001000000110
add 0 1 -> 000000000000000000000000000000000000000000000000000000100000010
sub 6299720 6299720-> 0000000000000000000000000000000000000000000000000000001001000110
shl -1 7 -> 0000000000000000000000000000000000000000000000000000001010000011
```

OF is not 1, neither CF^SF is 1.

The code is used to check whether there is overflow or not–

```
static inline bool
IntegerOverflow_new(Instruction *instr, uint64_t flagsValue)
{
    std::bitset<64> bitFlagsValue(flagsValue);
    if (instr->getOperatorType() == OperatorType::kOPadd
        || instr->getOperatorType() == OperatorType::kOPsub) {
        if(bitFlagsValue[11])
            return true;
        return false;
    }
    else if (instr->getOperatorType() == OperatorType::kOPshl) {
        if(bitFlagsValue[11] || (bitFlagsValue[0]^bitFlagsValue[7]))
            return true;
        return false;
    }
    return false;
}
```

And call this function in OnAfterInsExec() as–

```
if (IntegerOverflow_new(instr, flagsValue)) {
    ctxtContainer->addCtxt(contxt);
}
```

Comparison of algorithm used in the project (from slide 12) vs this new approach based on test1.c test cases–

```
 ** overflow happened **
OF: 0 CF: 0 SF: 1 XOR: 1

Time taken by og function: 6 microseconds

Time taken by new function: 1 microseconds
ip(4006a1):shl 1 7 -> -128 0000000000000000000000000000000000000000000000000000001010000010
```

```
 ** overflow happened **
OF: 1 CF: 0 SF: 1 XOR: 1

Time taken by og function: 7 microseconds

Time taken by new function: 1 microseconds
ip(400647):add 64 64 -> -128 0000000000000000000000000000000000000000000000000000101010000010
```

```
 ** overflow happened **
OF: 1 CF: 0 SF: 0 XOR: 0

Time taken by og function: 6 microseconds

Time taken by new function: 1 microseconds
ip(400a21):sub -4611686018427387904 4611686018427387905 -> 9223372036854775807 00000000000000000000000000000000000000000000000000
00101000010110
```

As we can see, the time taken by the new approach is less than the original approach.

Testing on new test cases (test2.c based on test1.c) —

int8_t x = 1<<7;
int8_t y = -1<<7;
int8_t z = sig_shift_8(-1, 8); //
int8_t w = sig_shift_8(1, 8); //
int8_t m = 127;
int8_t n = -128;
int8_t t1 = sig_add_8(m, n);
int8_t t2 = sig_sub_8(m, n); //
int8_t t3 = m + n;
int8_t t4 = m - n;
	int16_t x = 1<<15;
	int16_t y = -1<<15;
	int16_t z = sig_shift_16(-1, 16); //
	int16_t w = sig_shift_16(1, 16); //
	int16_t m = 32767;
	int16_t n = -32768;
	int16_t t1 = sig_add_16(m, n);
	int16_t t2 = sig_sub_16(m, n); //
	int16_t t3 = m + n;
	int16_t t4 = m - n;

int32_t x = 1<<31;
int32_t y = -1<<31;
int32_t z = sig_shift_32(-1, 32); //
int32_t w = sig_shift_32(1, 32); //
int32_t m = 2147483647;
int32_t n = -2147483648;
int32_t t1 = sig_add_32(m, n);
int32_t t2 = sig_sub_32(m, n); //
int32_t t3 = m + n;
int32_t t4 = m - n;
	int64_t x = 1L<<63; //
	int64_t y = -1L<<63; //
	int64_t z = sig_shift_64(1, 64);
	int64_t w = sig_shift_64(-1, 64);
	int64_t m = 9223372036854775807;
	int64_t n = -9223372036854775807;
	int64_t t1 = sig_add_64(m, n);
	int64_t t2 = sig_sub_64(m, n); //
	int64_t t3 = m + n;
	int64_t t4 = m - n;

`int8_t x = 1<<8` is `0` & `warning: overflow in implicit constant conversion [-Woverflow]`
also
`int8_t x = -1<<8` is `0` & `warning: overflow in implicit constant conversion [-Woverflow]`

Example comparison of 2 approaches used (using algorithm from slide 12 vs new approach) on new test cases test2.c–

```
 ** overflow happened **
OF: 0 CF: 1 SF: 0 XOR: 1

Time taken by og function: 26 microseconds

Time taken by new function: 1 microseconds
ip(4006a1):shl -1 8 -> 0 0000000000000000000000000000000000000000000000000000001001000111
```