

In this task, an interpreter for GRAsm is to be implemented. This should execute a program in byte code correctly and recognize any errors that may occur. A virtual state on which the operations are to work is available as an abstraction for the "GRA machine".

The following "virtual" 64-bit registers are available:

Registers	Special functionality
ip	Instruction Pointer
ac	Accumulator Register, Return Register
r0, r1, r2, r3, r4, r5, r6, r7	General Purpose registers

These are defined as:

```
struct grasm_state { uint64_t ip, acc, r0, ..., r7; };
```

arranged in memory.

The following table shows the available instructions and encodings.

Instruktion	Immediate	Encoding	Funktion
stop	--	0x01	Ends the program
nop	--	0x0f	No function
set	uint64_t	0x10 <imm>	ac = imm
set rX	uint64_t	0x18 <0X> <imm>	rX = imm
cpy rX	--	0x11 <0X>	ac = rX
cpy rX rY	--	0x19 <XY>	rX = rY
add	uint64_t	0x20 <imm>	ac += imm
add rX	uint64_t	0x28 <0X> <imm>	rX += imm
add rX	--	0x21 <0X>	ac += rX
add rX rY	--	0x29 <XY>	rX += rY
sub	uint64_t	0x22 <imm>	ac -= imm
sub rX	uint64_t	0x2a <0X> <imm>	rX -= imm
sub rX	--	0x23 <0X>	ac -= rX
sub rX rY	--	0x2b <XY>	rX -= rY
mul	uint64_t	0x24 <imm>	ac *= imm
mul rX	uint64_t	0x2c <0X> <imm>	rX *= imm
mul rX	--	0x25 <0X>	ac *= rX
mul rX rY	--	0x2d <XY>	rX *= rY
xchg rX	--	0x26 <0X>	tmp = ac; ac = rX; rX = tmp
xchg rX rY	--	0x2e <XY>	tmp = rX; rX = rY; rY = tmp
and	uint64_t	0x30 <imm>	ac &= imm
and rX	uint64_t	0x31 <0X> <imm>	rX &= imm
and rX	--	0x32 <0X>	ac &= rX
and rX rY	--	0x33 <XY>	rX &= rY
or	uint64_t	0x34 <imm>	ac = imm
or rX	uint64_t	0x35 <0X> <imm>	rX = imm
or rX	--	0x36 <0X>	ac = rX
or rX rY	--	0x37 <XY>	rX = rY
xor	uint64_t	0x38 <imm>	ac ^= imm
xor rX	uint64_t	0x39 <0X> <imm>	rX ^= imm
xor rX	--	0x3a <0X>	ac ^= rX
xor rX rY	--	0x3b <XY>	rX ^= rY
not	--	0x3c	ac = ~ac
not rX	--	0x3d <0X>	ac = ~rX
not rX rY	--	0x3e <XY>	rX = ~rY
cmp rX	uint64_t	0x40 <0X> <imm>	ac = rX - imm

Instruktion	Immediate	Encoding	Funktion
cmp rX rY	--	0x41 <XY>	ac = rX - rY
tst rX	uint64_t	0x42 <0X> <imm>	ac = rX & imm
tst rX rY	--	0x43 <XY>	ac = rX & rY
shr	uint8_t	0x50 <imm>	ac >>= imm
shr rX	uint8_t	0x51 <0X> <imm>	rX >>= imm
shr rX	--	0x52 <0X>	ac >>= rX
shr rX rY	--	0x53 <XY>	rX >>= rY
shl	uint8_t	0x54 <imm>	ac <<= imm
shl rX	uint8_t	0x55 <0X> <imm>	rX <<= imm
shl rX	--	0x56 <0X>	ac <<= rX
shl rX rY	--	0x57 <XY>	rX <<= rY
ld	uintptr_t	0x60 <imm>	ac = [imm]
ld rX	uintptr_t	0x61 <0X> <imm>	rX = [imm]
ld rX	--	0x62 <0X>	ac = [rX]
ld rX rY	--	0x63 <XY>	rX = [rY]
st	uintptr_t	0x64 <imm>	[imm] = ac
st rX	uintptr_t	0x65 <0X> <imm>	[imm] = rX
st rX	--	0x66 <0X>	[rX] = ac
st rX rY	--	0x67 <XY>	[rX] = rY
go	uintptr_t	0x70 <imm>	ip = <imm>
go rX	--	0x71 <0X>	ip = rX
gr	int16_t	0x72 <imm>	ip = ip + <imm>
jz	uintptr_t	0x73 <imm>	ip = ac == 0 ? <imm> : ip + sizeof(jz)
jz rX	--	0x74 <0X>	ip = ac == 0 ? rX : ip + sizeof(jz)
jrz	int16_t	0x75 <imm>	ip = ac == 0 ? ip + <imm> : ip + sizeof(jrz)
ecall	uintptr_t	0x80 <imm>	ac = <imm>(r0, .., r5)
ecall rX	--	0x81 <0X>	ac = rX(r0, .., r5)

With the instruction *ecall*, (unknown) external functions are called at the given address:

Be sure to follow the calling convention!

The functions accept a maximum of 6 parameters, the result should be stored in *ac*.

Unless otherwise specified, all immediates are encoded in little-endian format.

i.e. *add 0x01234567* -> 20 67 45 23 01 00 00 00.

Especially with the jumps, make sure you set the *ip* correctly!

ip is updated at the end of a (valid) instruction, i.e. after any error handling.

Jumps set the *ip* explicitly.

Relative jumps are calculated relative to the current instruction.

The memory addresses passed for *ld/st* and *ecall* are valid and refer directly to the underlying memory.

The upper 4 bits of <0X> are don't-care for this task, so they should not be checked.

Shifts can shift a maximum of 63 bits, i.e. only 7 bits of the operand are considered.

If errors occur, the following error codes should be returned:

error	Error code	reason
No error	0	--
unknown Opcode	-1	Instruction does not exist or is incomplete
Out-of-Bounds-access	-2	<i>ip</i> >= <i>len</i> , Register not valid

Example program (line breaks, offsets and comments for visualization only):

```

# ac = fac(r0) if r0 else 0 . . .
00: 19 10                # cpy r1 r0
02: 40 01 00 00 00 00 00 00 # cmp r1 0
0c: 75 1f 00             # jrz +31
0f: 40 01 01 00 00 00 00 00 # cmp r1 1
19: 75 12 00             # jrz +18
1c: 2a 01 01 00 00 00 00 00 # sub r1 1
26: 2d 01                # mul r0 r1
28: 72 e7 ff             # gr -25
2b: 11 00                # cpy r0
2d: 01                  # stop

```

TASK:

In x86-64 assembly, implement an interpreter that meets the above requirements; in particular, all instructions must be implemented.

Signature(for the registers passing from C to assembly): `uint64_t grasm_interpreter(struct grasm_state* state, size_t len, uint8_t prog[len]);`