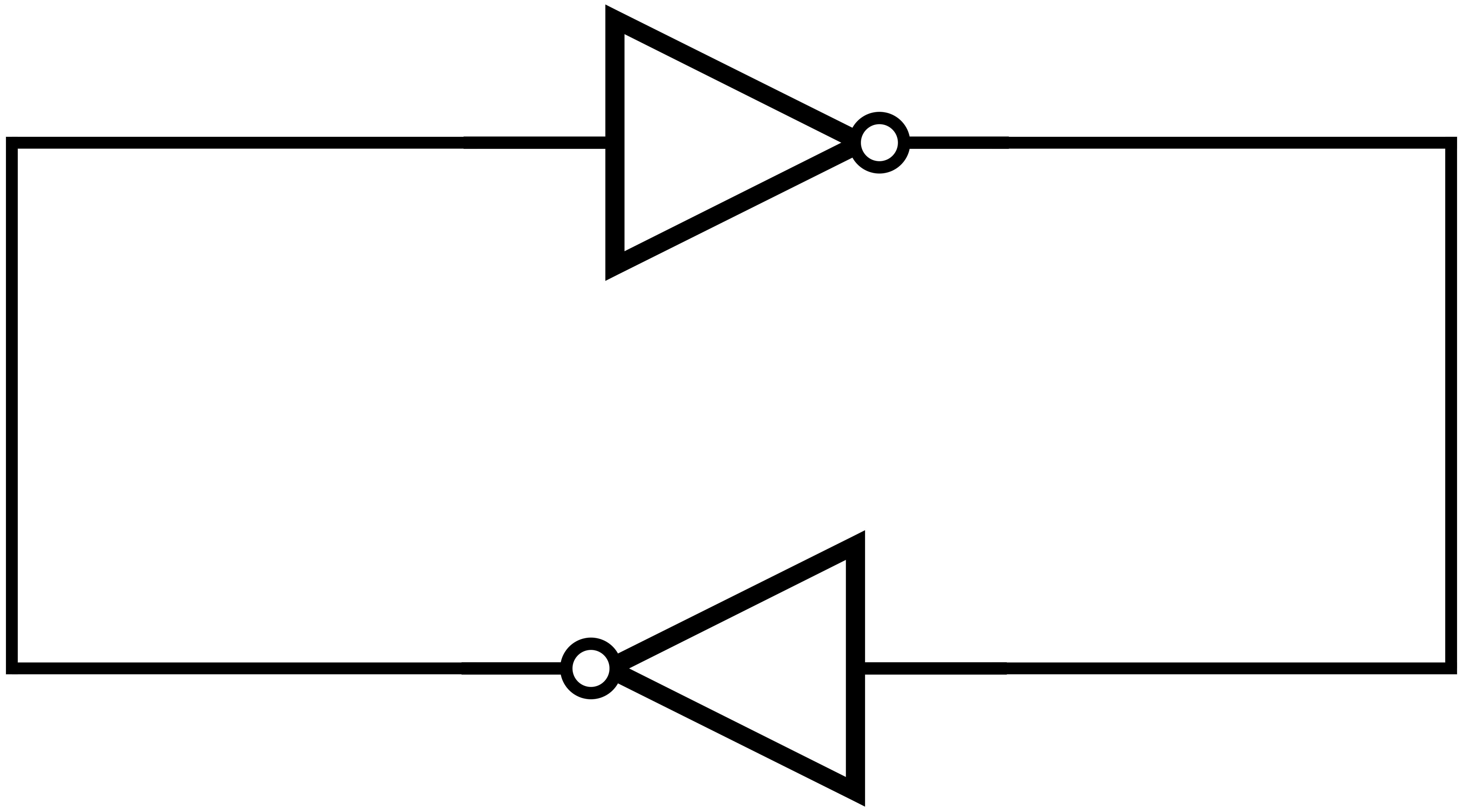
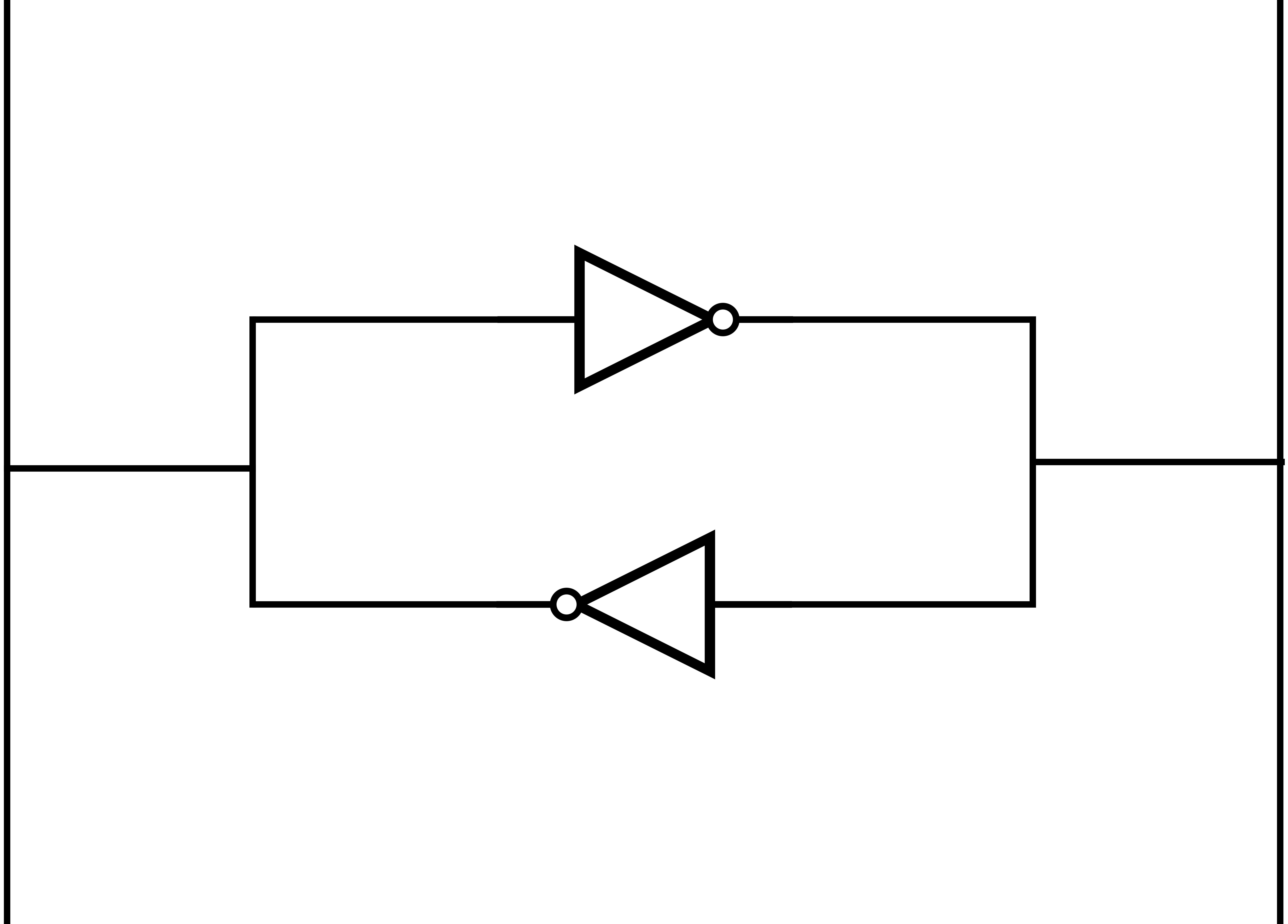
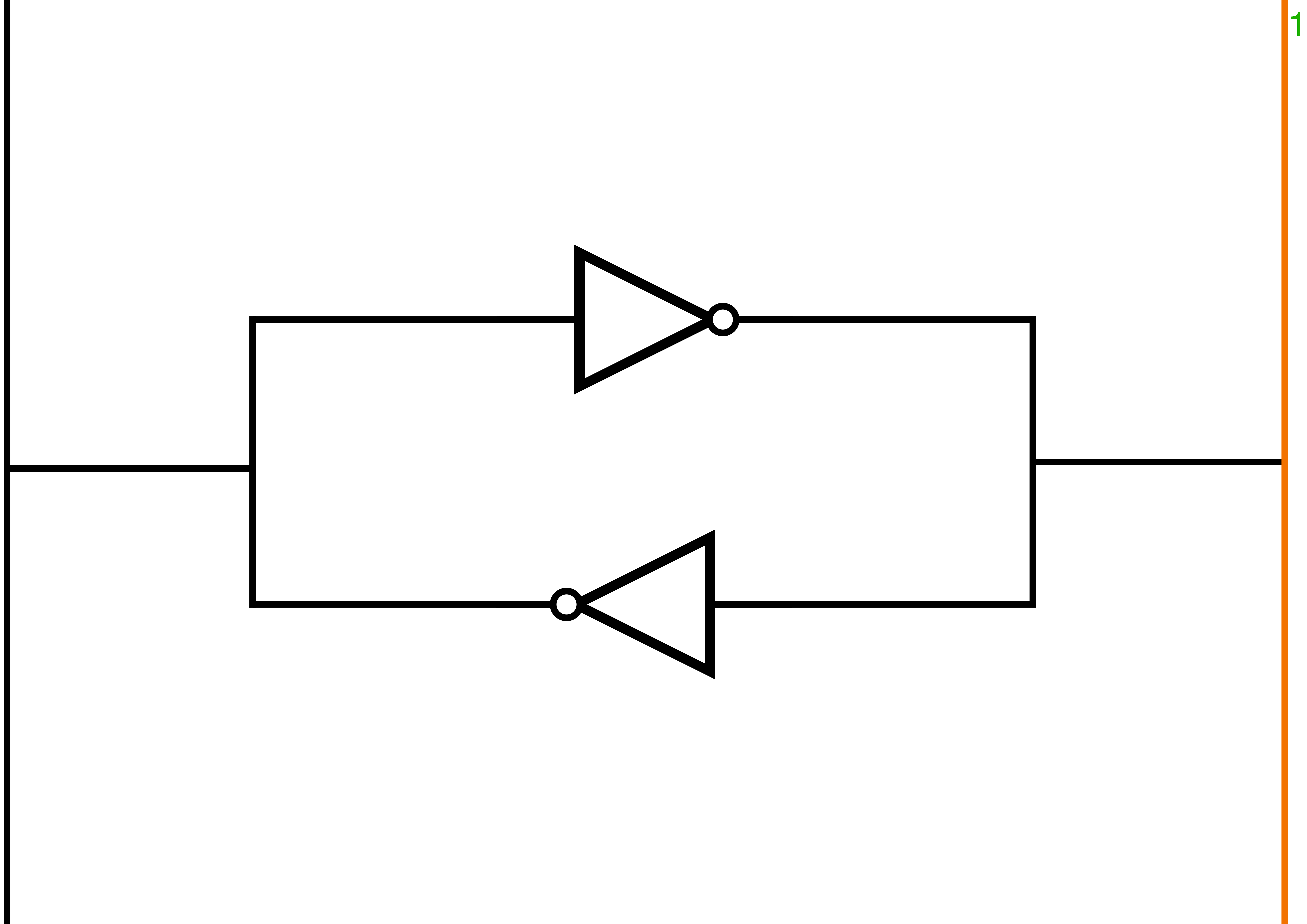
Pointers II

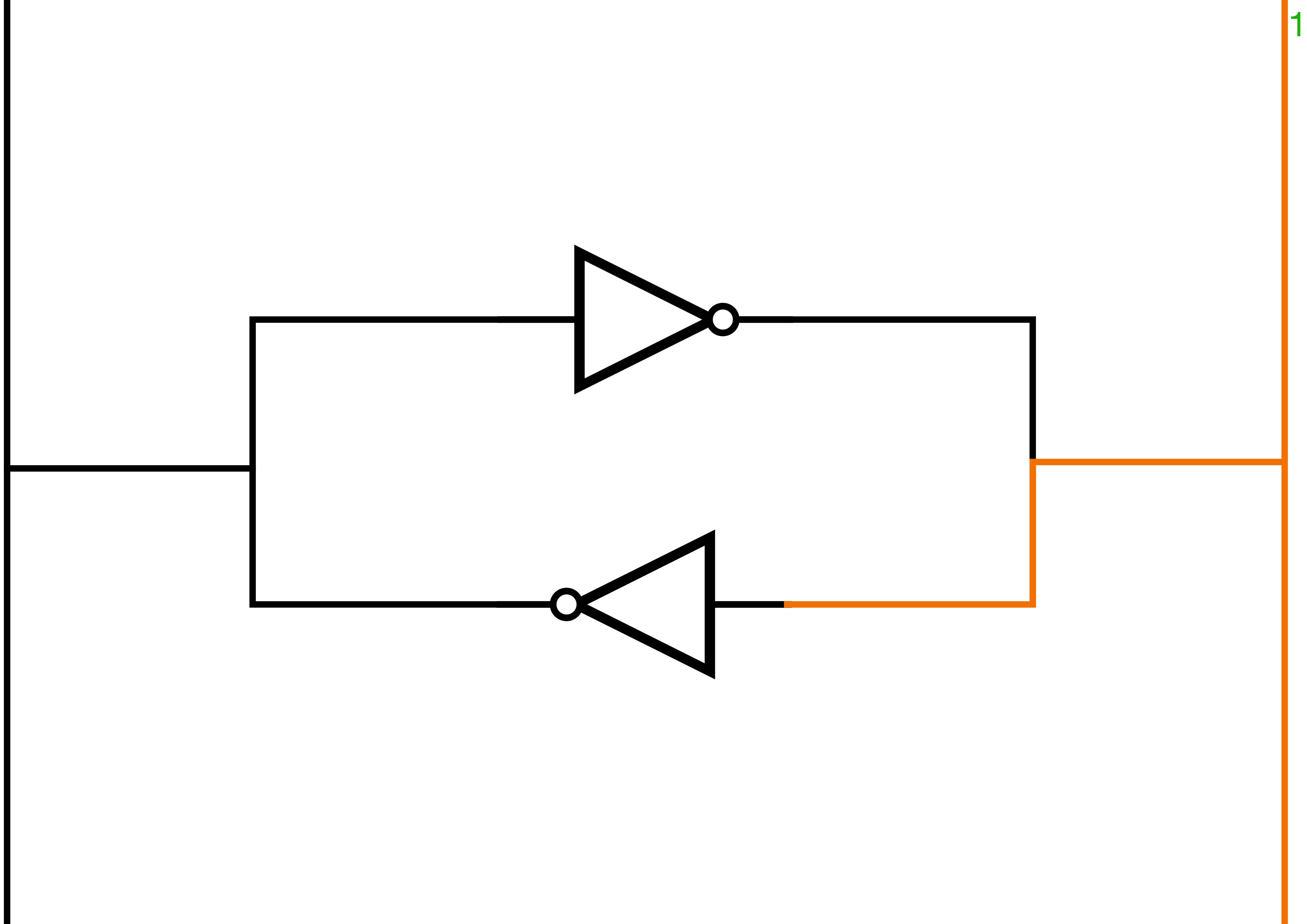
CS143: lecture 8

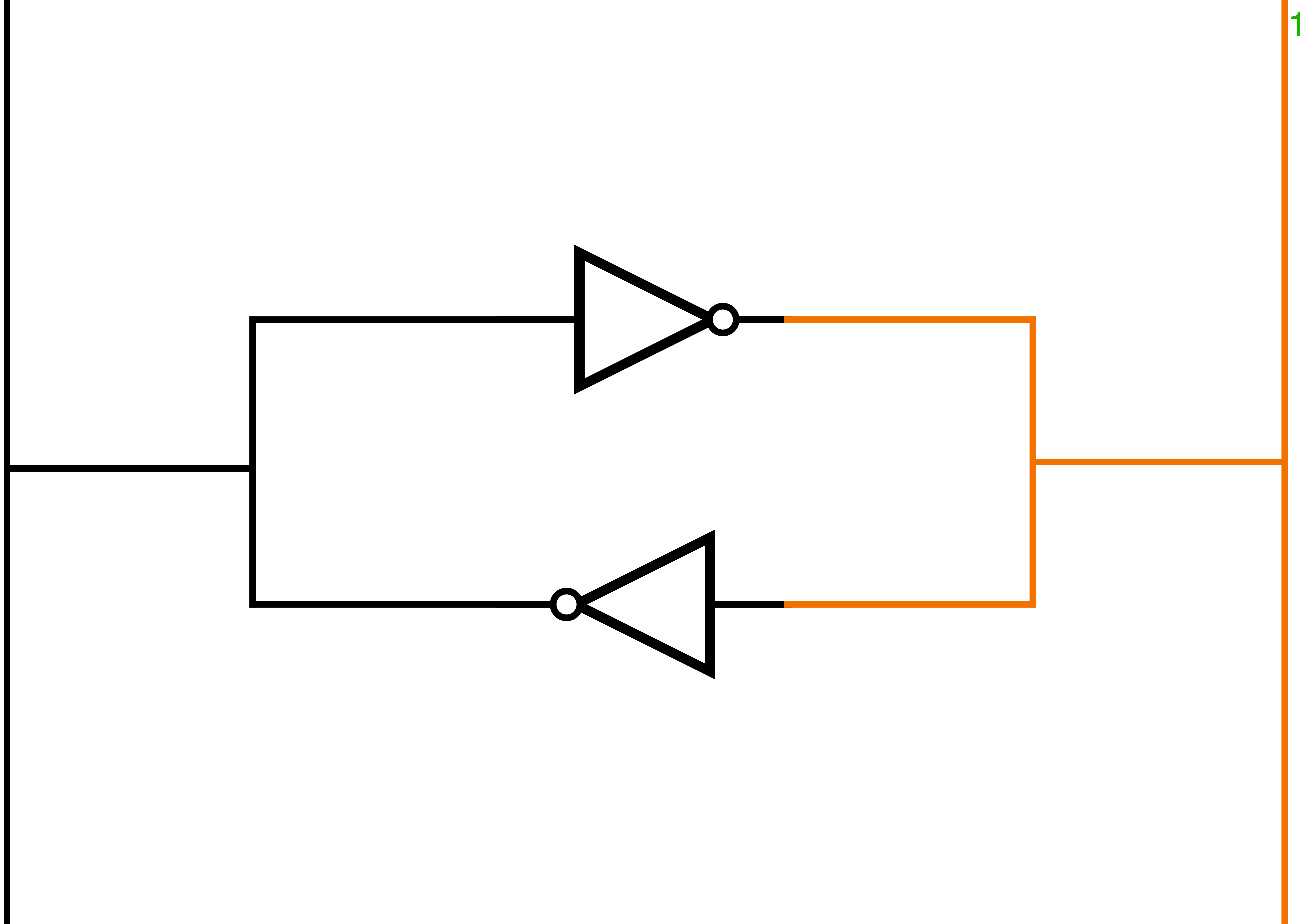
Byron Zhong, June 25

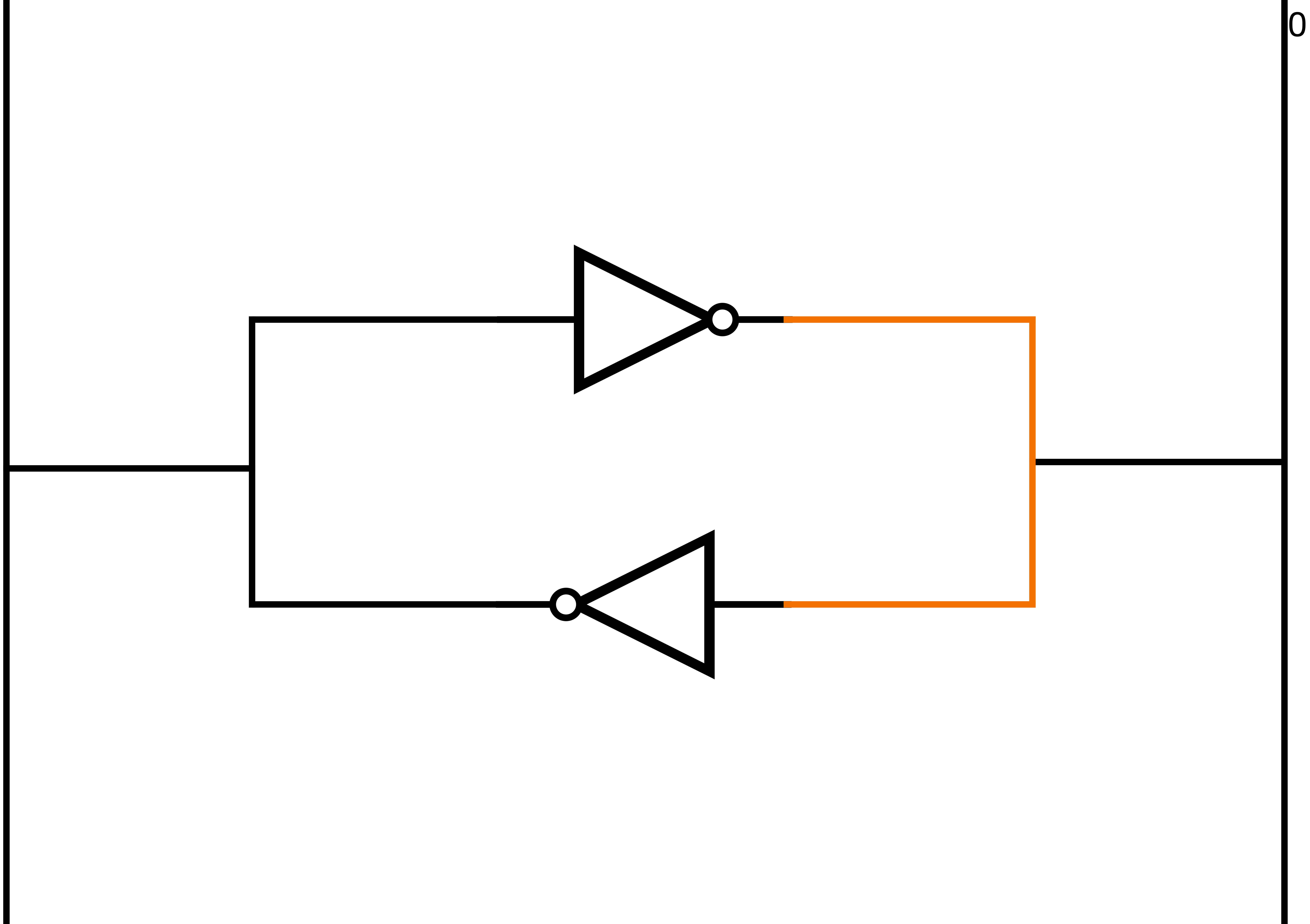


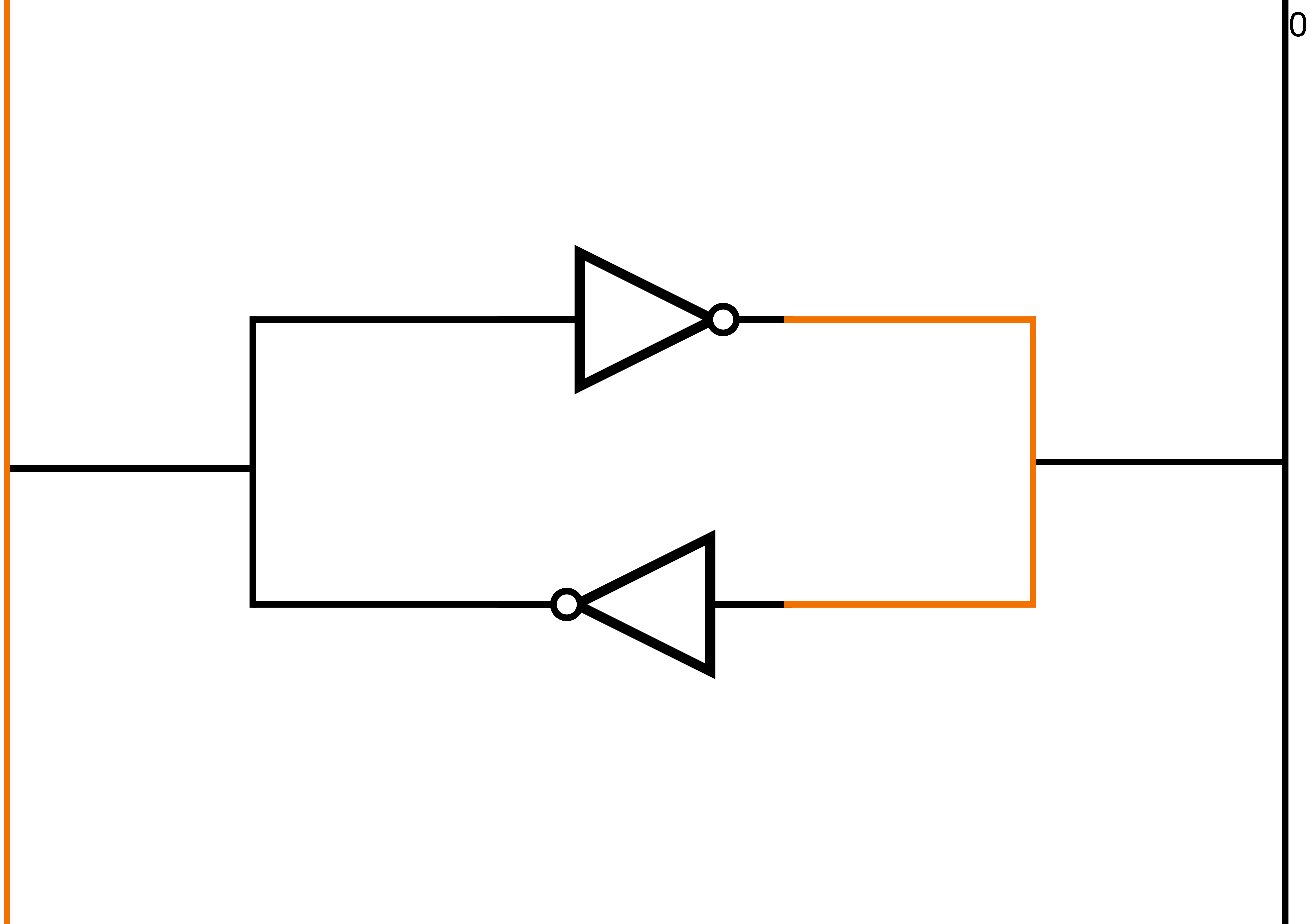


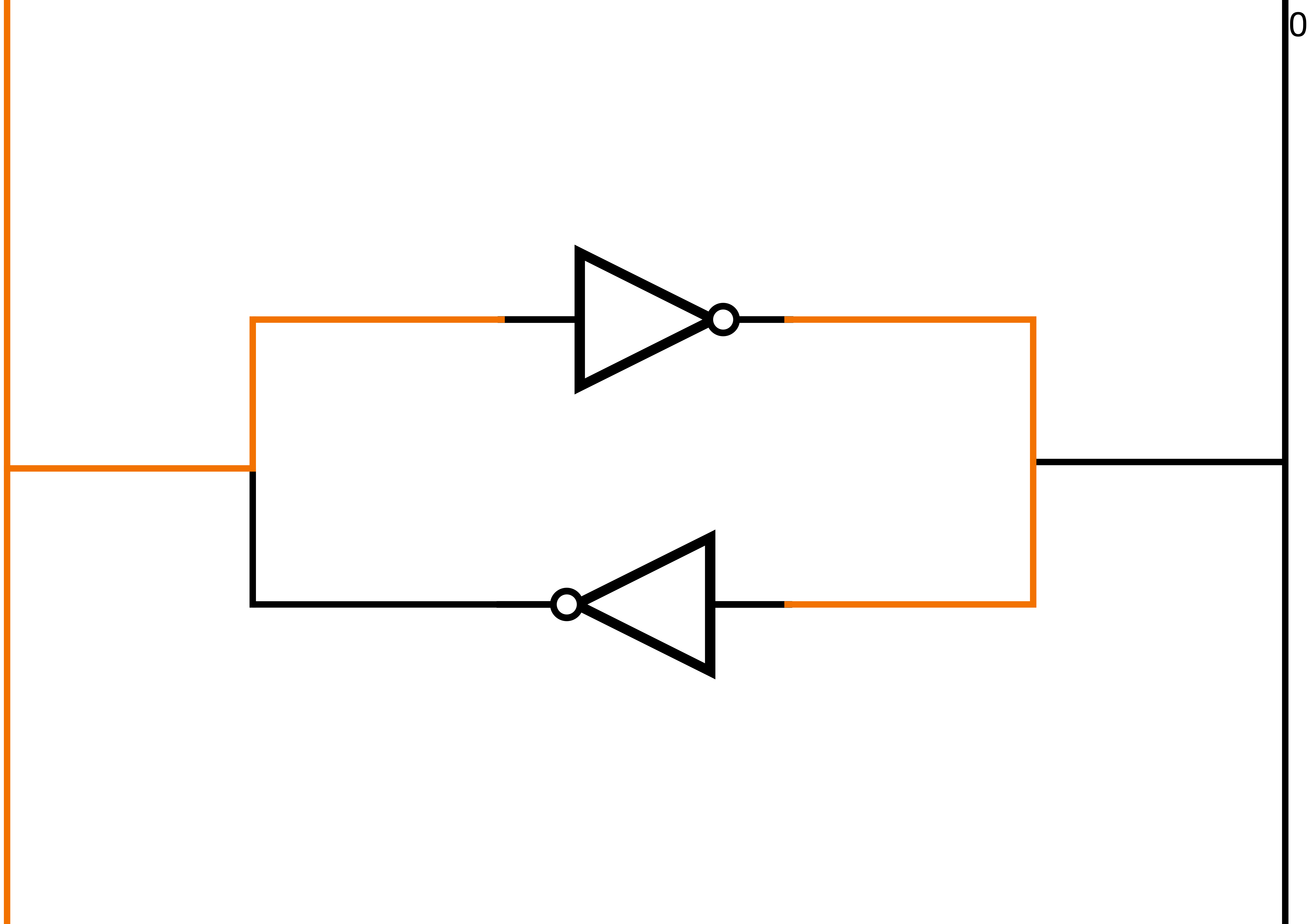

1

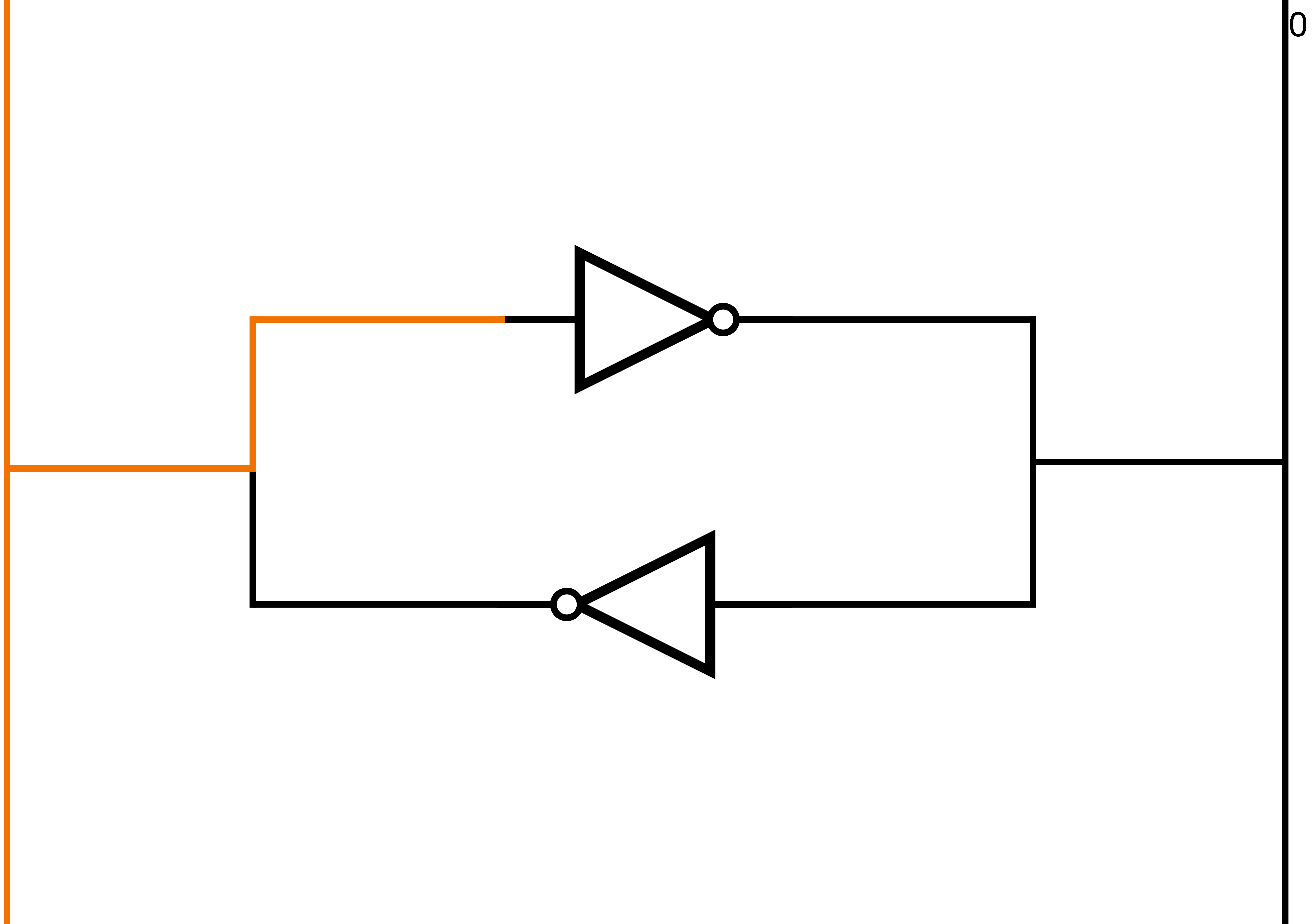

1

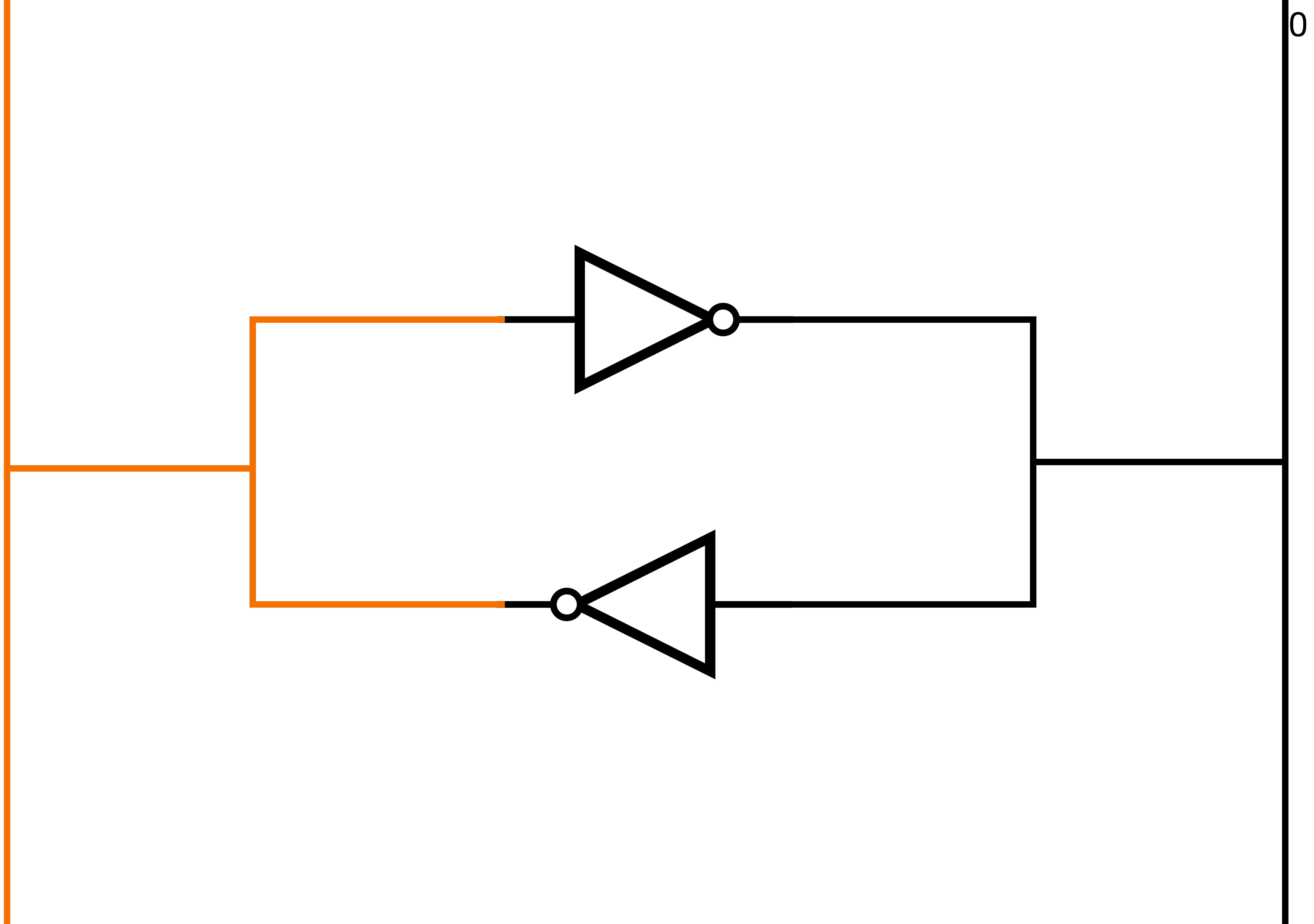

1

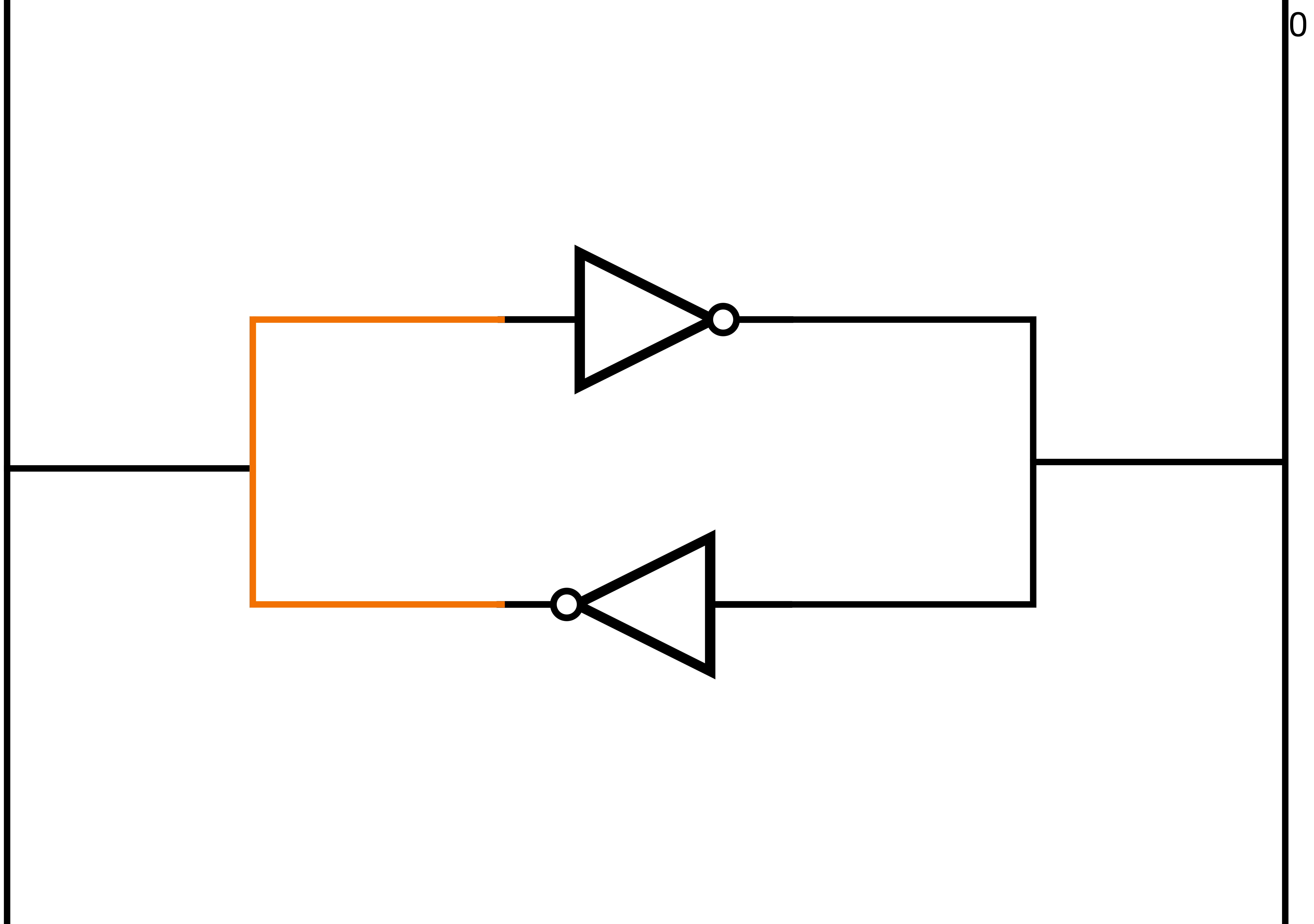

0


0

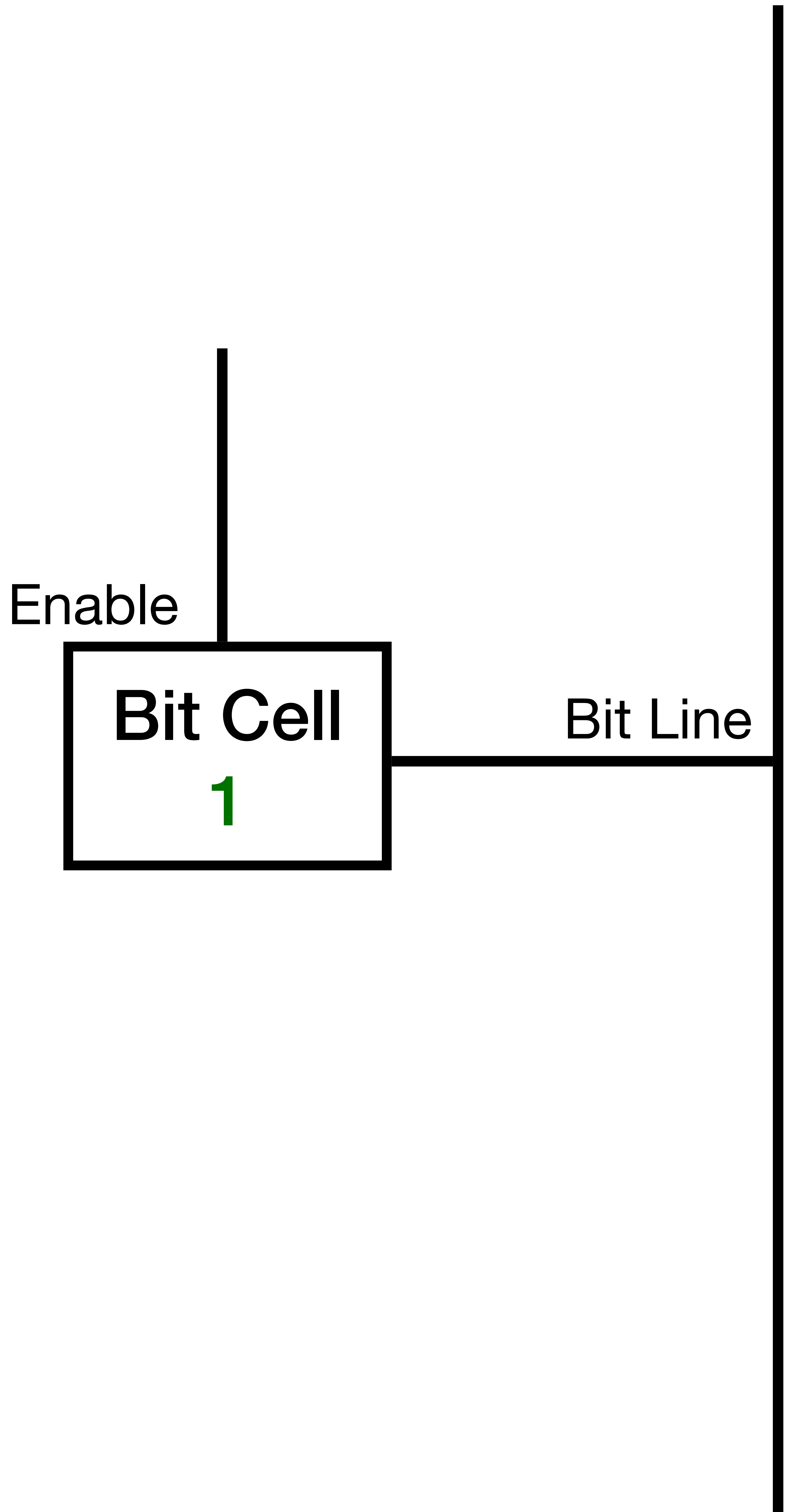

0

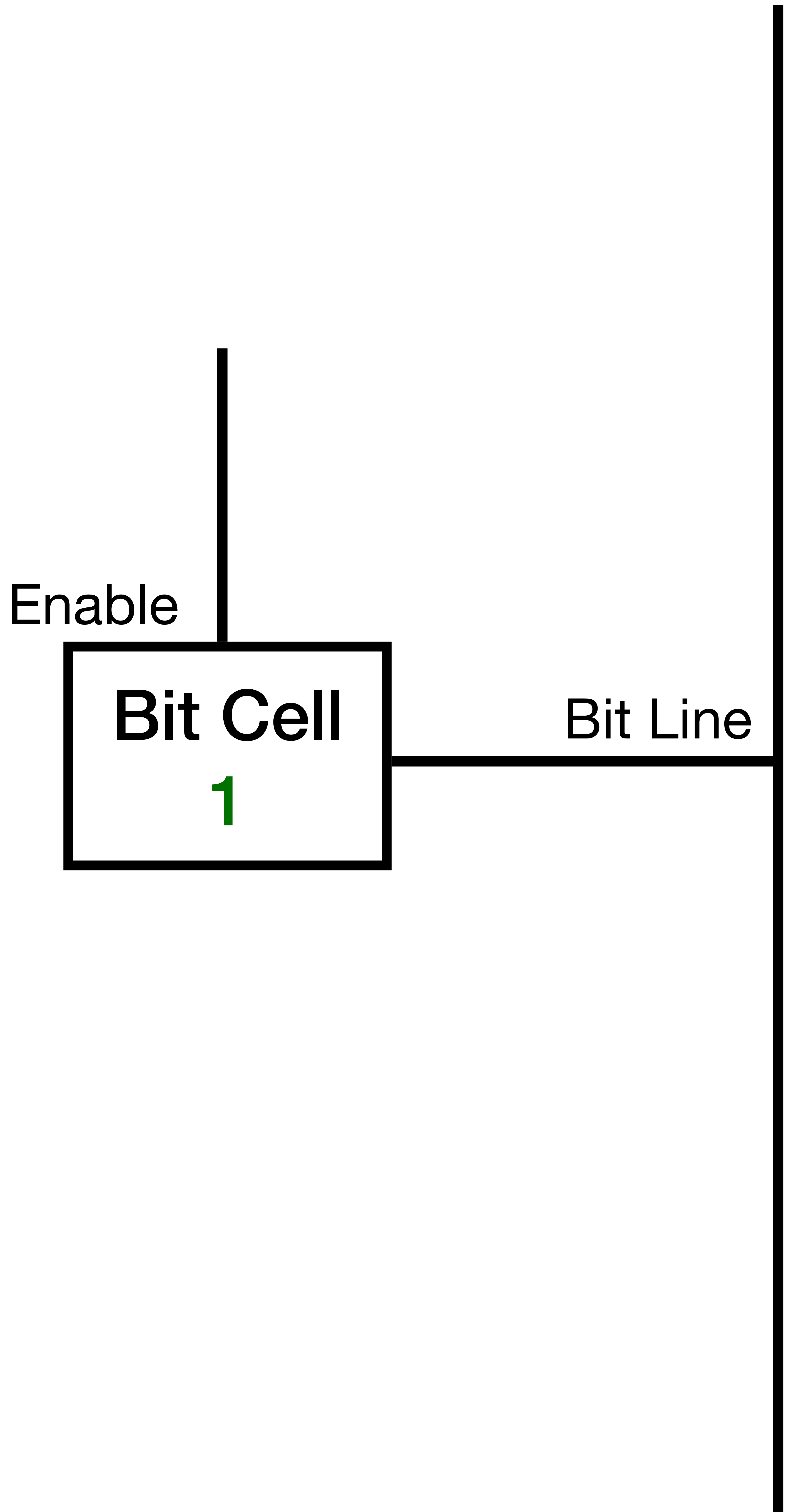

0

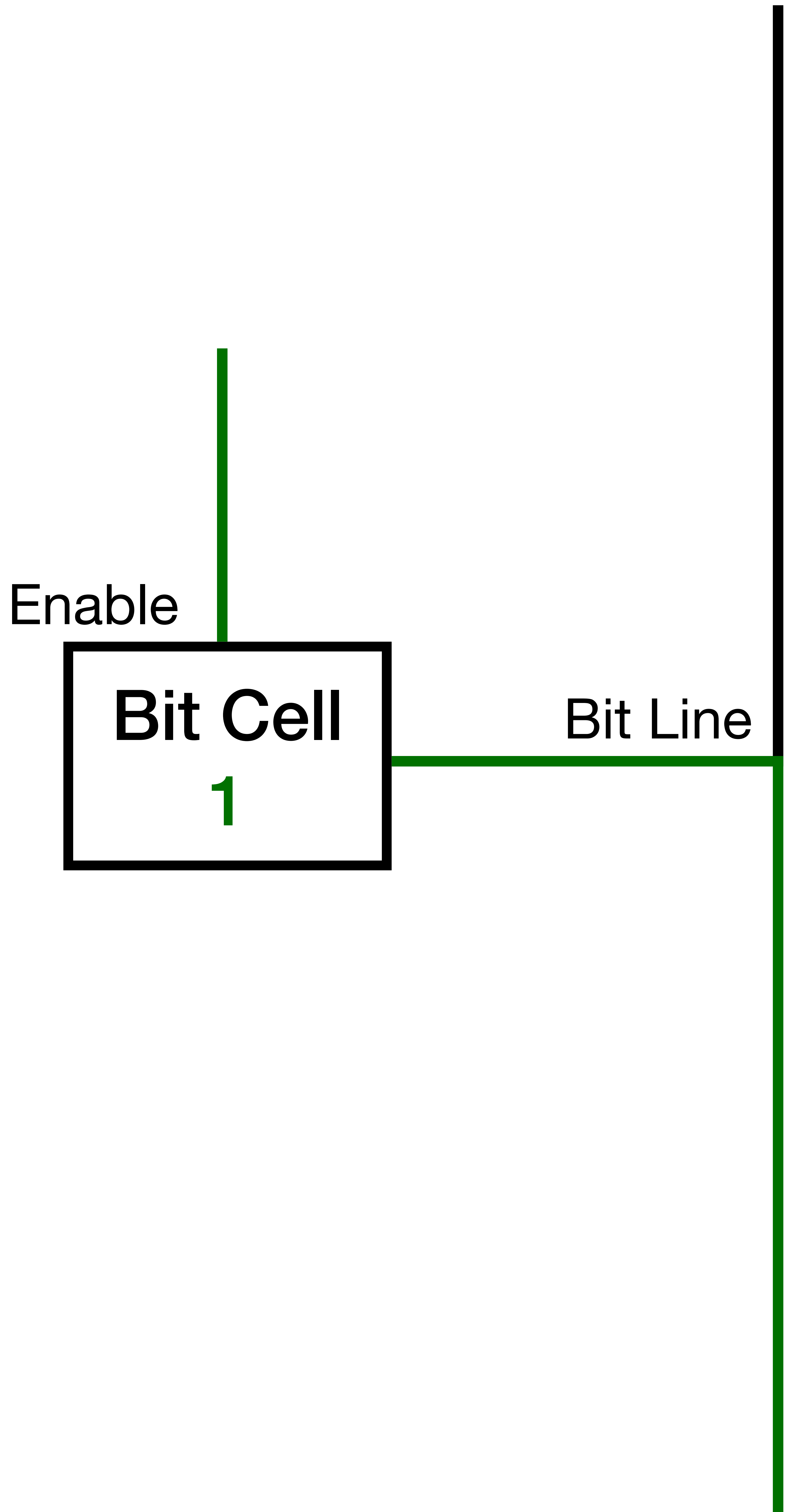

0

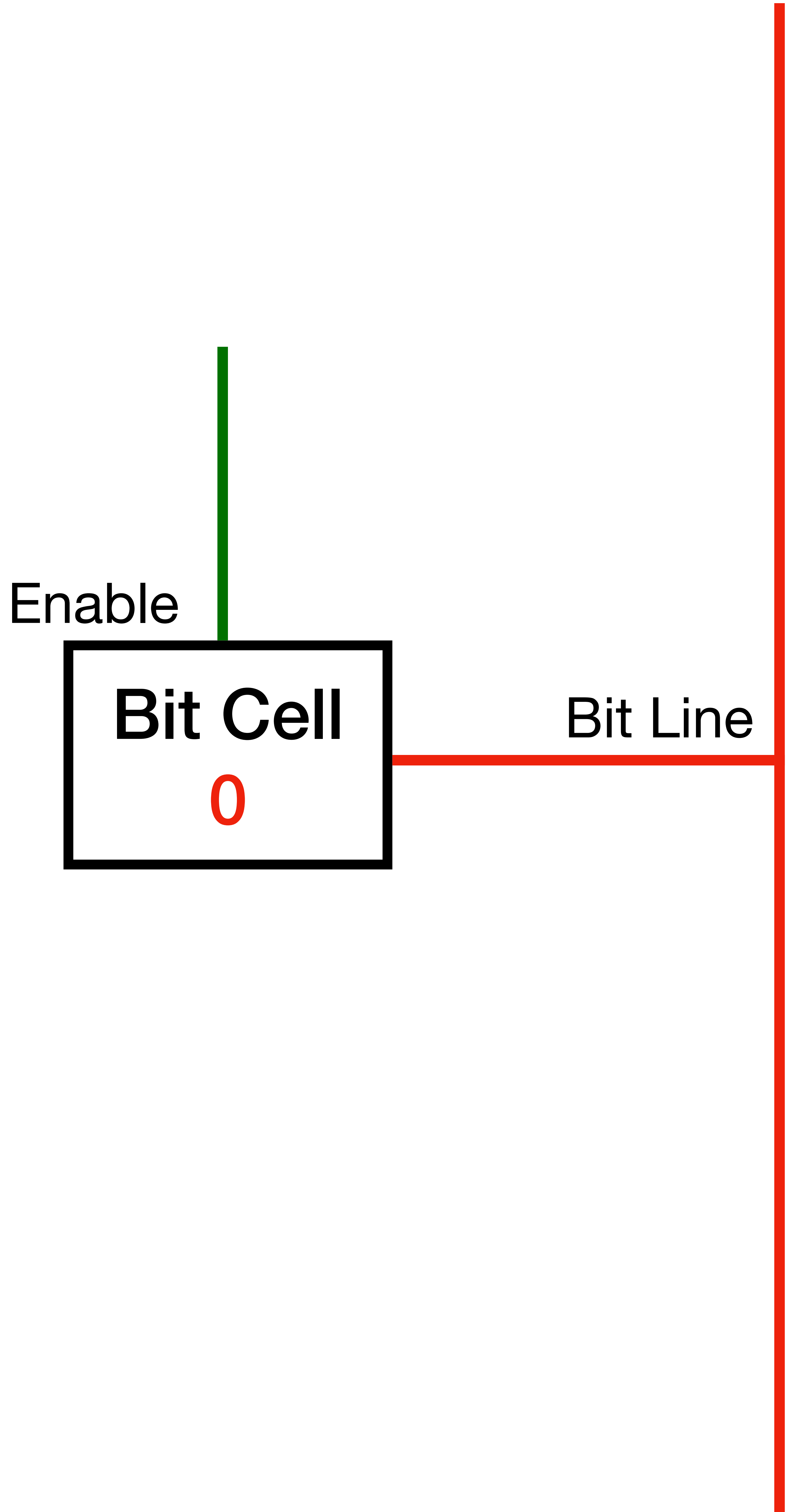

0

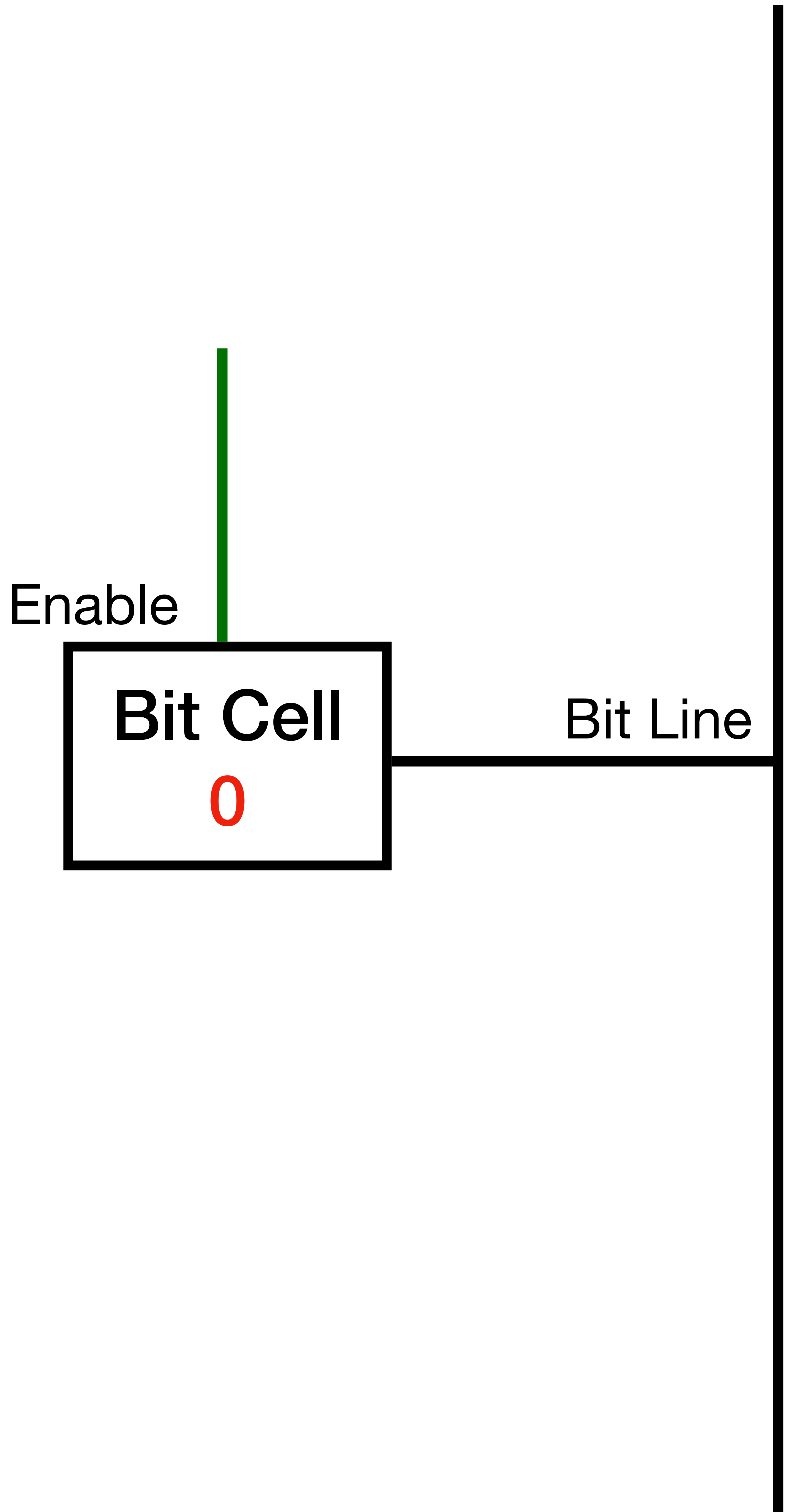

Inverted
Bit Line
Bit Line
Enable
Line

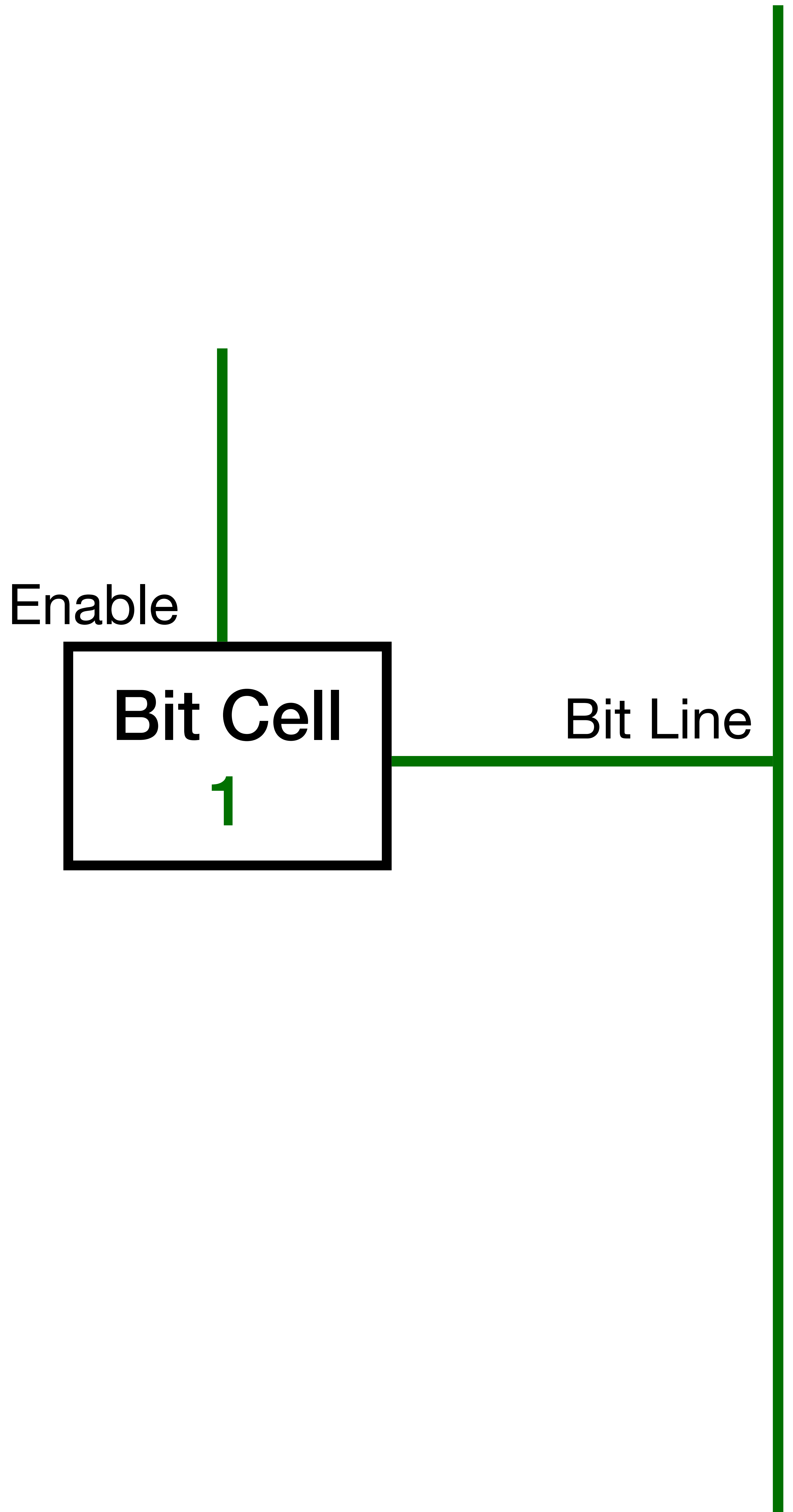

Enable
Bit Cell
1
Bit Line


Enable
Bit Cell
1
Bit Line


Enable
Bit Cell
1
Bit Line


Enable
Bit Cell
0
Bit Line


Enable
Bit Cell
0
Bit Line


Enable
Bit Cell
1
Bit Line


Address
2 bits
Decoder
11
Enable
10
01
00


Bit7
Bit6
Bit5
Bit4
Bit3
Bit2
Bit1
Bit0
11
Enable
10
Address
2 bits
Decoder
01
00


Bit7
Bit6
Bit5
Bit4
Bit3
Bit2
Bit1
Bit0
11
Enable
Bit Cell
10
Address
2 bits
Decoder
01
00


Bit7
Bit6
Bit5
Bit4
Bit3
Bit2
Bit1
Bit0
11
Enable
10
01
00
Address
2 bits
Decoder
Bit Cell

Bit7
Bit6
Bit5
Bit4
Bit3
Bit2
Bit1
Bit0
11
Enable
10
01
00
Address
2 bits
Decoder
Bit Cell

Memory Array

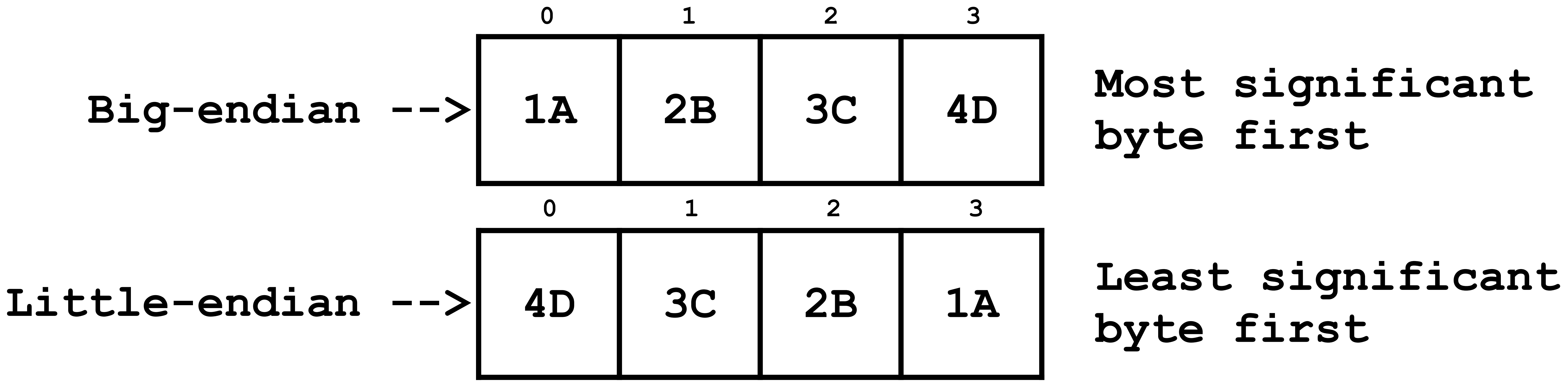
- Each row of data is called a *word.* Most memories use 8-bit word, a *byte*.
- 2^N-word $\times$ M-bit memory array. N is the size of an address. M is the smallest addressable unit.
- An address causes the *enable* lines of all bit cells in a row to turn on, and their contents are read/written simultaneously.
- On modern machines, M is almost always 8.
- What is N, the size of a memory address?
  - 64 on *64-bit machine*, 32 on *32-bit machine.*

Memory Array

- 2^{32} = 4,294,967,296 = ~4.3 G of addressable rows.
- 4.2 gigabytes of addressable memory.
- In order to use beyond 4.2GB, memory addresses need to be bigger.
- 2^{64} = 18,446,744,073,709,551,616 = 18 *exabytes* = ~4.2 million gigabytes

Endian

- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`
- But if an integer has 4 bytes and each byte is addressable, which of the 4 bytes is stored first?



Endian

- Is my machine little-endian or big-endian?
- Let's find out!

Endian

- Our machine is little-endian?????
- We usually write numbers in big-endian: 345 is three hundred and forty-five
- But there are some advantages for little-endian:
    - comparing two numbers of different length (long and int e.g.)
        - `4E3C2B1A`
        - `4E3C2B1A00000000`
    - addition, subtraction circuits work from low to high
    - etc.

Endian

Does it matter?

- Mostly we don't care. Unless you do memory trickery, variables work as you would expect
- However, when we serialize data into byte sequences, you need to pay extra attention:
  - Writing a number to a file
  - Sending a number over a network
- You and the reader must agree on byte order
  - For this purpose, *network byte order* is defined for TCP/IP

`struct`

Making your own types

- Data placed in memory can be: `char, short, int, long, float, double`
- What if you want to store something other than a number?
  - Student?
  - Course?
  - House?
  - ...
- You use numbers to represent them; you *digitize* them.
- In C, we can use *structures* to bundle data together.

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```

Don't forget the ;

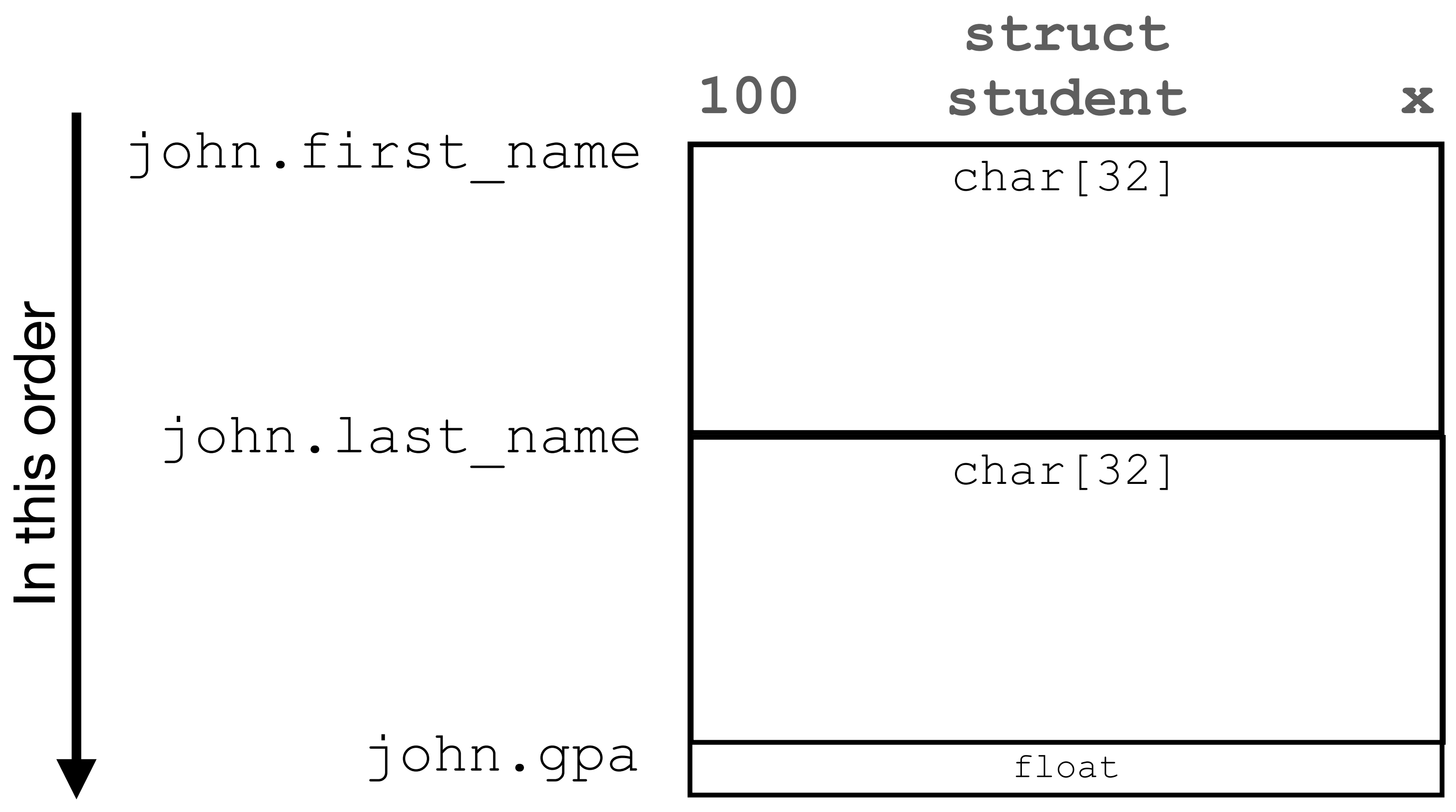
Don't forget the word struct

Assignment (=) doesn't work with arrays

struct

Syntax

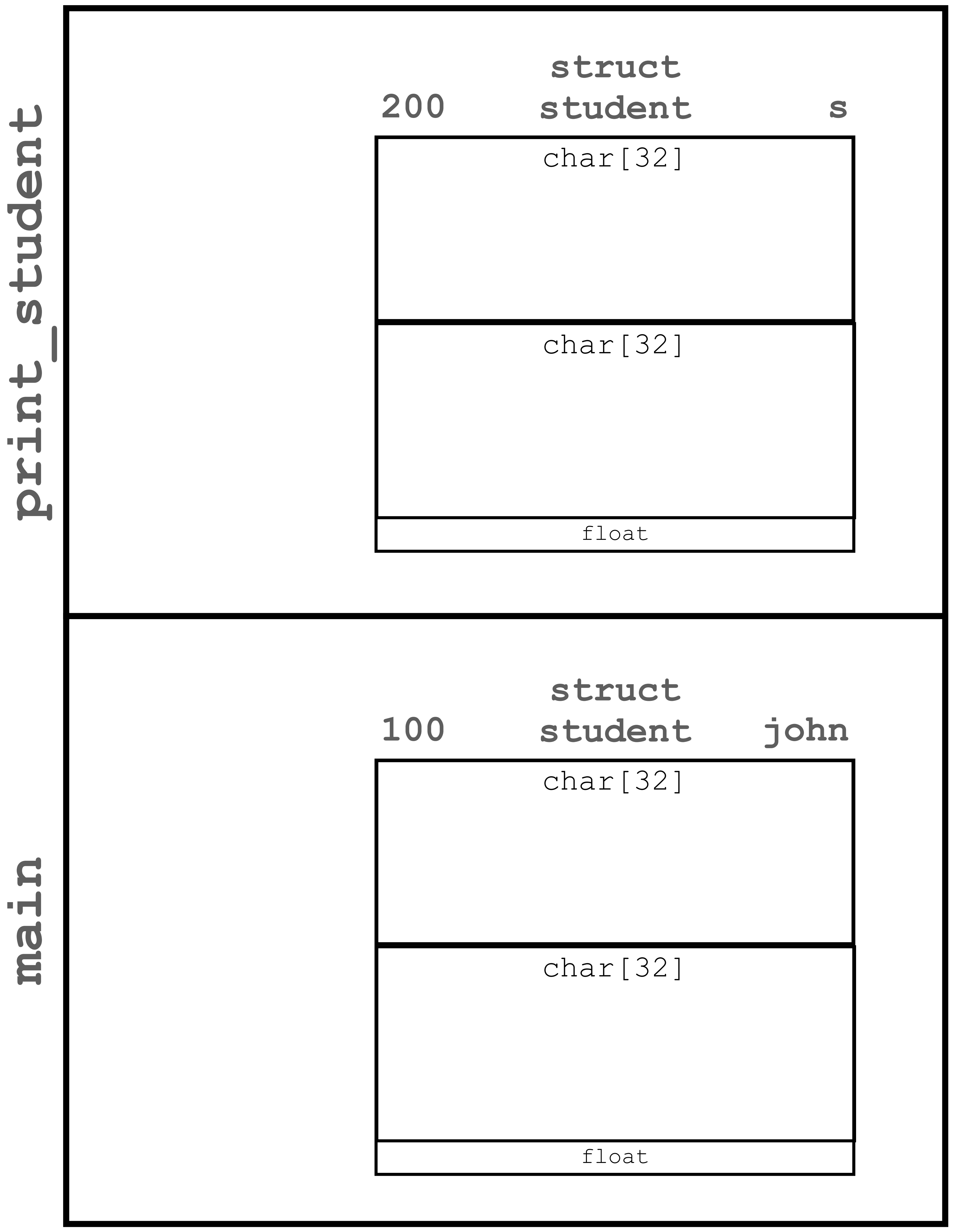
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```



struct

Structures are passed by value

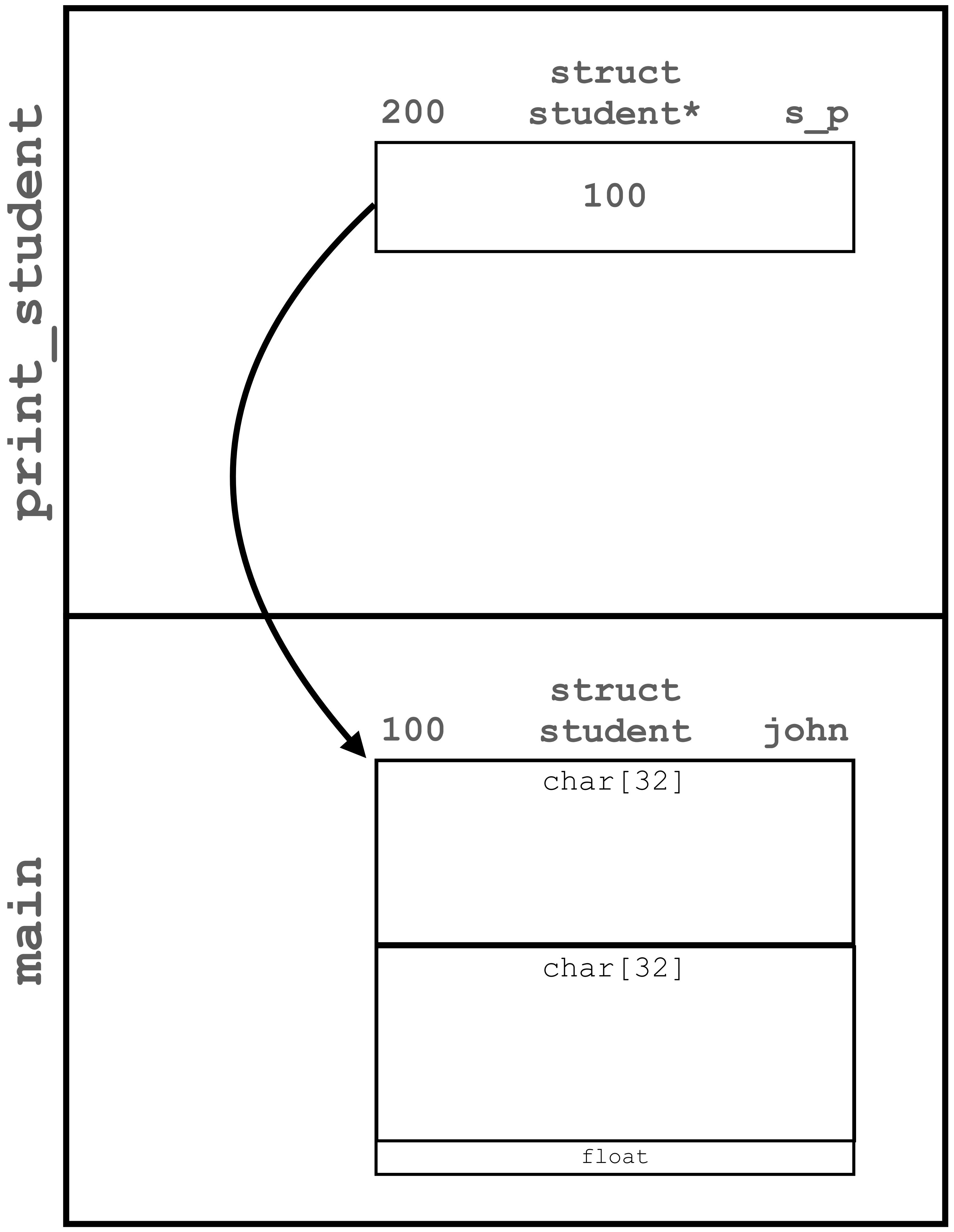
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student s)
{
        printf("%s %s: %.2f\n", s.first_name, s.last_name, s.gpa);
}


int main(void)
{
        struct student john;
        ...
        print_student(john);

        return 0;
}
```



struct

Structures are passed by value

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", (*s_p).first_name,
                                (*s_p).last_name,
                                (*s_p).gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```



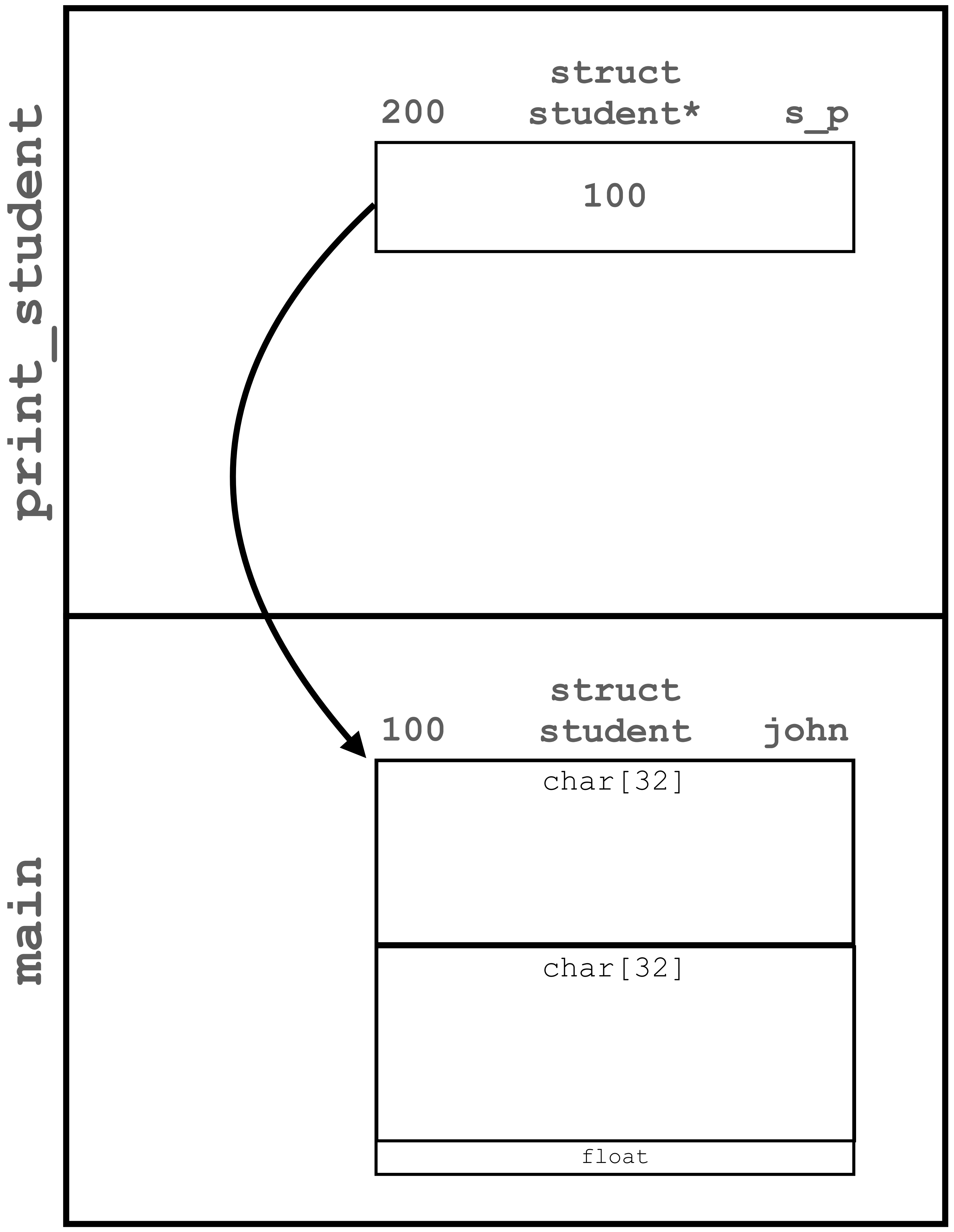
struct

Structures are passed by value

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", s_p->first_name,
                                s_p->last_name,
                                s_p->gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```

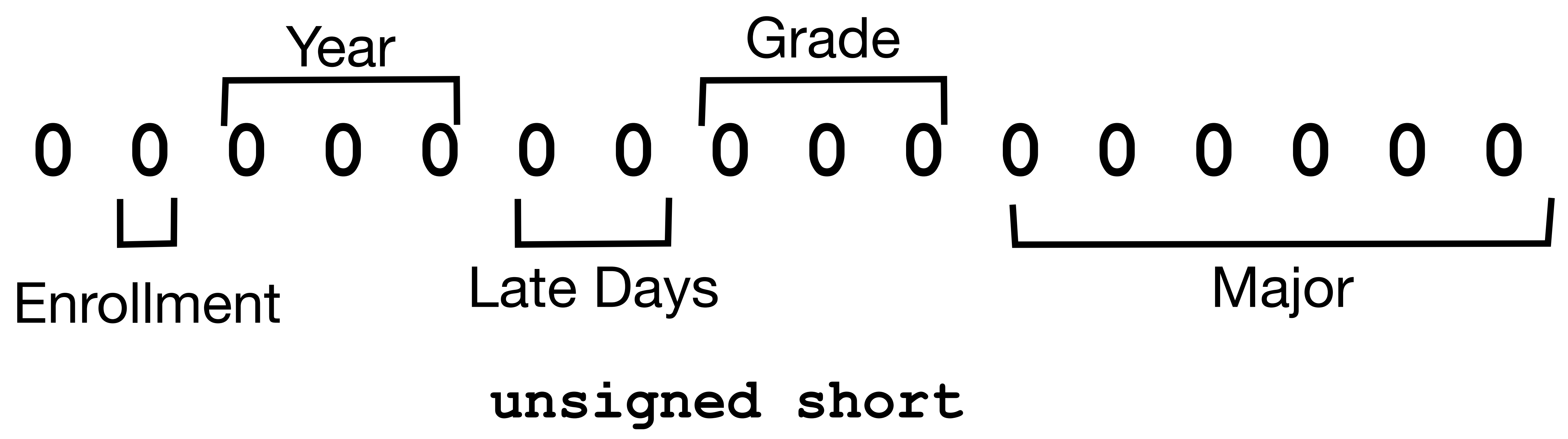
s_p->gpa is a shorthand for (*s_p).gpa

Btw, *s_p.gpa is read as *(s_p.gpa), which is an error



Choices

enum

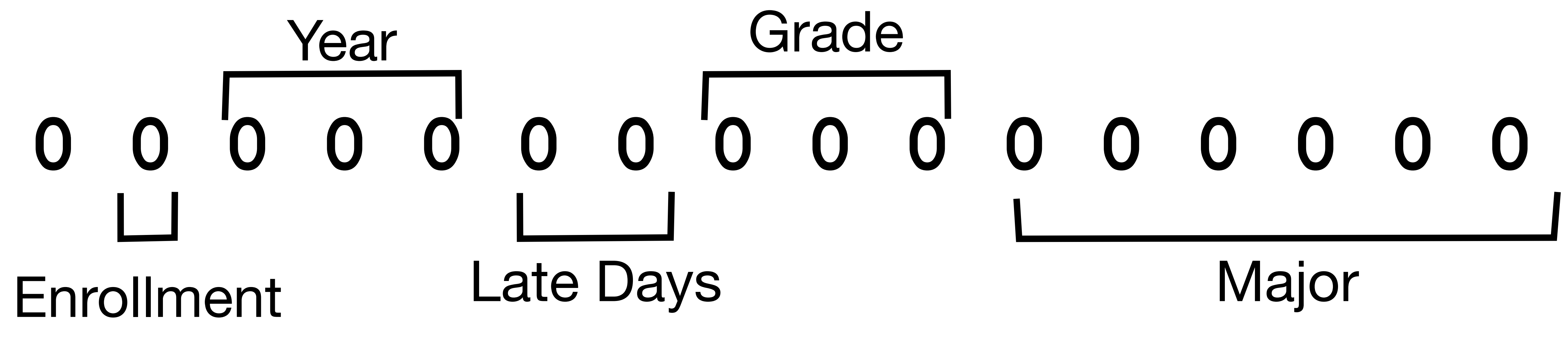



```
enum major {
        ANTHROPOLOGY,
        ARCHITECTURAL_STUDIES,
        ART_HISTORY,
        ASTRONOMY_ASTROPHYSICS,
        BIG_PROBLEMS,
        BIOLOGICAL_CHEMISTRY,
        ...
};
```

- Nothing fancy here: C just assigns an integer sequentially for each choice.
- Can use them as global constants
- `if (major == 2) { ... }`
- `if (major == ART_HISTORY) { ... }`

Choices

`enum`



`unsigned short`

```
enum major {
        ANTHROPOLOGY,
        ARCHITECTURAL_STUDIES,
        ART_HISTORY,
        ASTRONOMY_ASTROPHYSICS,
        BIG_PROBLEMS,
        BIOLOGICAL_CHEMISTRY,
        ...
};
```

```
enum major student_major = STATISTICS;
```

```
enum major student_major = 3;
```

`clang` will *not* complaint about this. But this is bad style.

Choices

switch

```
if (major == ANTHROPOLOGY) {
        ...
} else if (major == ARCHITECTURAL_STUDIES) {
        ...
} else if (major == ART_HISTORY) {
        ...
} else if (major == ASTRONOMY_ASTROPHYSICS) {
        ...
} else if (major == BIG_PROBLEMS) {
        ...
} ...
```

```
switch (major) {
case ANTHROPOLOGY:
        ...;
        break;
case ARCHITECTURAL_STUDIES:
        ...;
        break;
case ART_HISTORY:
        ...;
        break;
case BIOLOGICAL_CHEMISTRY:
        ...;
        break;
case BIG_PROBLEMS:
        ...;
        break;
default:
        break;
}
```

`break` signals the end of a case. Without break, C will execute the next case, *falling through* another case.

The default branch is run when the major matches none of the above cases.

Choices

`switch`

- Why switch?
- Cleaner code
- More efficient than if ... else if ... chain:
    - C stores the branches in a table, and switch will jump to the branch instead of comparing one by one
    - `switch(x)`, `x` has to be an integer.
- `break` is critical! Forgetting the break is really difficult to debug.

Choices

`switch`

- Demo!

File I/O

- Memory is *volatile*. It loses data when power is removed.
- Files are stored on *non-volatile* storage media such as hard drives. It does not need power to preserve data.
- Memory supports *random access*. One can access memory at any address directly.
- Hard drives only support *sequential access*.
- C abstracts file system access via `FILE *`.