Pointers II

CS143: lecture 8

Konstantinos Ameranis, July 2

Characters and numbers

Command line arguments

Review

Command line arguments

Review

```
int main(int argc, char *argv[])
```

Command line arguments

Review

```
int main(int argc, char *argv[])
```

- `argc` — Argument count

Command line arguments

Review

```
int main(int argc, char *argv[])
```

- `argc` — Argument count
- `argv` — Argument vector

Command line arguments

Review

```
int main(int argc, char *argv[])
```

- `argc` — Argument count
- `argv` — Argument vector
- `argv[0]` is the name of the program

Command line arguments

Review

```
int main(int argc, char *argv[])
```

- `argc` — Argument count
- `argv` — Argument vector
- `argv[0]` is the name of the program
- All arguments are strings

Command line arguments

Review

```
int main(int argc, char *argv[])
```

- `argc` — Argument count
- `argv` — Argument vector
- `argv[0]` is the name of the program
- All arguments are strings

```
./rgb2hex 128 64 32
0x804020
```

Command line arguments

```
./rgb2hex 128 64 32
0x804020
```

Command line arguments

```
./rgb2hex 128 64 32
0x804020
```

- How to turn string command line arguments to numbers

Command line arguments

```
./rgb2hex 128 64 32
0x804020
```

- How to turn string command line arguments to numbers
- Three ways

Command line arguments

```
./rgb2hex 128 64 32
0x804020
```

- How to turn string command line arguments to numbers
- Three ways
    1. Pure C, working with ASCII

Command line arguments

```
./rgb2hex 128 64 32
0x804020
```

- How to turn string command line arguments to numbers
- Three ways
    1. Pure C, working with ASCII
    2. Using `stdlib.h`

Command line arguments

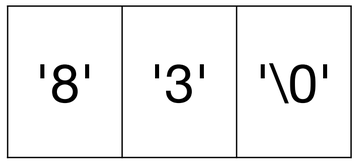
```
./rgb2hex 128 64 32
0x804020
```

- How to turn string command line arguments to numbers
- Three ways
    1. Pure C, working with ASCII
    2. Using `stdlib.h`
    3. Using `stdio.h`

Strings to numbers

Pure C

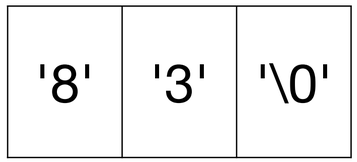
*char int

- "83" → 83
- "83" → 

Strings to numbers

Pure C

*char int

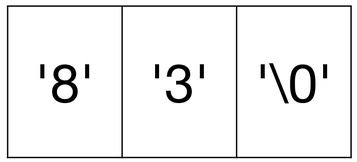
- "83" → 83
- "83" → 


```
int digit = '8' - '0';
```

Strings to numbers

Pure C

*char int

- "83" → 83
- "83" → 

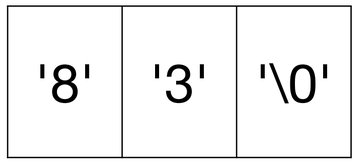

```
int digit = '8' - '0';
```

- By subtracting `'0'` from the character, you can recover the digit

Strings to numbers

Pure C

*char int

- "83" → 83
- "83" → 


```
int digit = '8' - '0';
```

- By subtracting `'0'` from the character, you can recover the digit
- 83 = 8 * 10 + 3

Strings to numbers

Pure C

```
// Alphanumeric to integer
int atoi(const char *s) {
    int number = 0;
    for (int i = 0; s[i] != '\0'; i++) {
        int digit = s[i] - '0';
        if ((digit < 0) || (digit > 9)) {
            // This is not a valid integer string
            // ??? Error handling?
        }
        number = number * 10 + digit;
    }
    return number;
}
```

Strings to numbers

Pure C

Strings to numbers

Pure C

```
if ((digit < 0) || (digit > 9)) {
    // This is not a valid integer string
    // ??? Error handling?
}
```

Strings to numbers

Pure C

```
if ((digit < 0) || (digit > 9)) {
    // This is not a valid integer string
    // ??? Error handling?
}
```

- What should you do if there is an error?

Strings to numbers

Pure C

```
if ((digit < 0) || (digit > 9)) {
    // This is not a valid integer string
    // ??? Error handling?
}
```

- What should you do if there is an error?

1. `exit(EXIT_FAILURE);` → Bad idea if writing a library

Strings to numbers

Pure C

```
if ((digit < 0) || (digit > 9)) {
    // This is not a valid integer string
    // ??? Error handling?
}
```

- What should you do if there is an error?

1. `exit(EXIT_FAILURE);` → Bad idea if writing a library
2. `int atoi(const char *s, int &errno);` → Pass a reference to return an error code (if applicable) → Too cumbersome and ugly

Strings to numbers

Pure C

```
if ((digit < 0) || (digit > 9)) {
    // This is not a valid integer string
    // ??? Error handling?
}
```

- What should you do if there is an error?

1. `exit(EXIT_FAILURE);` → Bad idea if writing a library
2. `int atoi(const char *s, int &errno);` → Pass a reference to return an error code (if applicable) → Too cumbersome and ugly
3. Global `errno` that is set when appropriate

Strings to numbers

Pure C

Strings to numbers

Pure C

- Global `errno` that is set when appropriate

Strings to numbers

Pure C

- Global `errno` that is set when appropriate
- `errno` is an `enum`

Strings to numbers

Pure C

- Global `errno` that is set when appropriate
- `errno` is an `enum`
- Different values signify different errors

Strings to numbers

Pure C

- Global `errno` that is set when appropriate
- `errno` is an `enum`
- Different values signify different errors

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

int main(void){
    for (int i = 0; i < 134; i++) {
        printf("%3d: %s\n", i, strerror(i));
    }
    return EXIT_SUCCESS;
}
```

Strings to numbers

`stdlib.h`

Strings to numbers

`stdlib.h`

- Many different ways to write an integer

Strings to numbers

`stdlib.h`

- Many different ways to write an integer
  - `354`

Strings to numbers

`stdlib.h`

- Many different ways to write an integer
  - `354`
  - `+354`

Strings to numbers

`stdlib.h`

- Many different ways to write an integer
  - `354`
  - `+354`
  - `0x161`

Strings to numbers

`stdlib.h`

- Many different ways to write an integer
  - `354`
  - `+354`
  - `0x161`
  - `0o542`

Strings to numbers

`stdlib.h`

- Many different ways to write an integer
  - `354`
  - `+354`
  - `0x161`
  - `0o542`
  - `0b101100010`

Strings to numbers

`stdlib.h`

Strings to numbers

`stdlib.h`

- `stdlib` has dedicated functions for this

Strings to numbers

`stdlib.h`

- `stdlib` has dedicated functions for this
  - `int atoi(const char *s);` → Alphanumeric to integer

Strings to numbers

`stdlib.h`

- `stdlib` has dedicated functions for this
  - `int atoi(const char *s);` → Alphanumeric to integer
  - `long atol(const char *s);` → Alphanumeric to long

Strings to numbers

`stdlib.h`

- `stdlib` has dedicated functions for this
  - `int atoi(const char *s);` → Alphanumeric to integer
  - `long atol(const char *s);` → Alphanumeric to long
  - `long long atoll(const char *s);` → Alphanumeric to long long

Strings to numbers

`stdlib.h`

- `stdlib` has dedicated functions for this
  - `int atoi(const char *s);` → Alphanumeric to integer
  - `long atol(const char *s);` → Alphanumeric to long
  - `long long atoll(const char *s);` → Alphanumeric to long long
  - `double atof(const char *s);` → Alphanumeric to float

Strings to numbers

`stdio.h`

Strings to numbers

`stdio.h`

- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read

Strings to numbers

`stdio.h`

- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read
- Can scan from

Strings to numbers

`stdio.h`

- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read
- Can scan from
  - Standard input → `scanf`

Strings to numbers

`stdio.h`

- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read
- Can scan from
  - Standard input → `scanf`
  - File pointer → `fscanf`

Strings to numbers

`stdio.h`

- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read
- Can scan from
  - Standard input → `scanf`
  - File pointer → `fscanf`
  - String → `sscanf`

Strings to numbers

`stdio.h`

- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read
- Can scan from
  - Standard input → `scanf`
  - File pointer → `fscanf`
  - String → `sscanf`
- Return value is how many variables were read

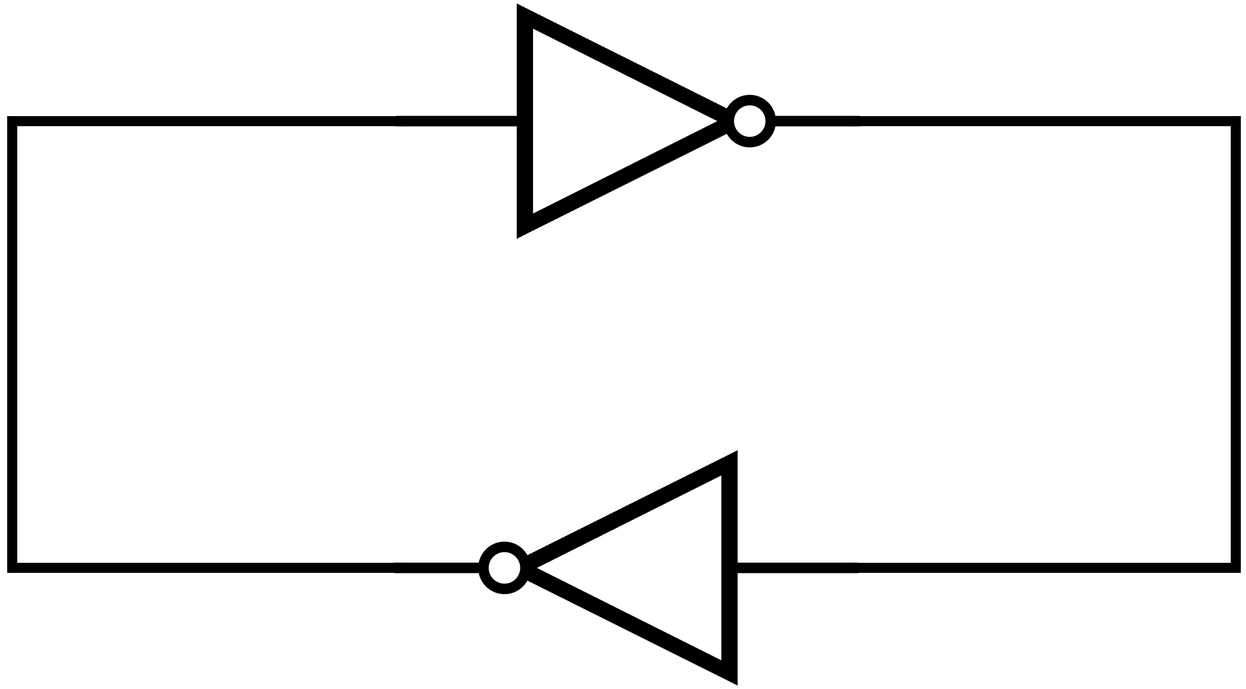
Strings to numbers

`stdio.h`

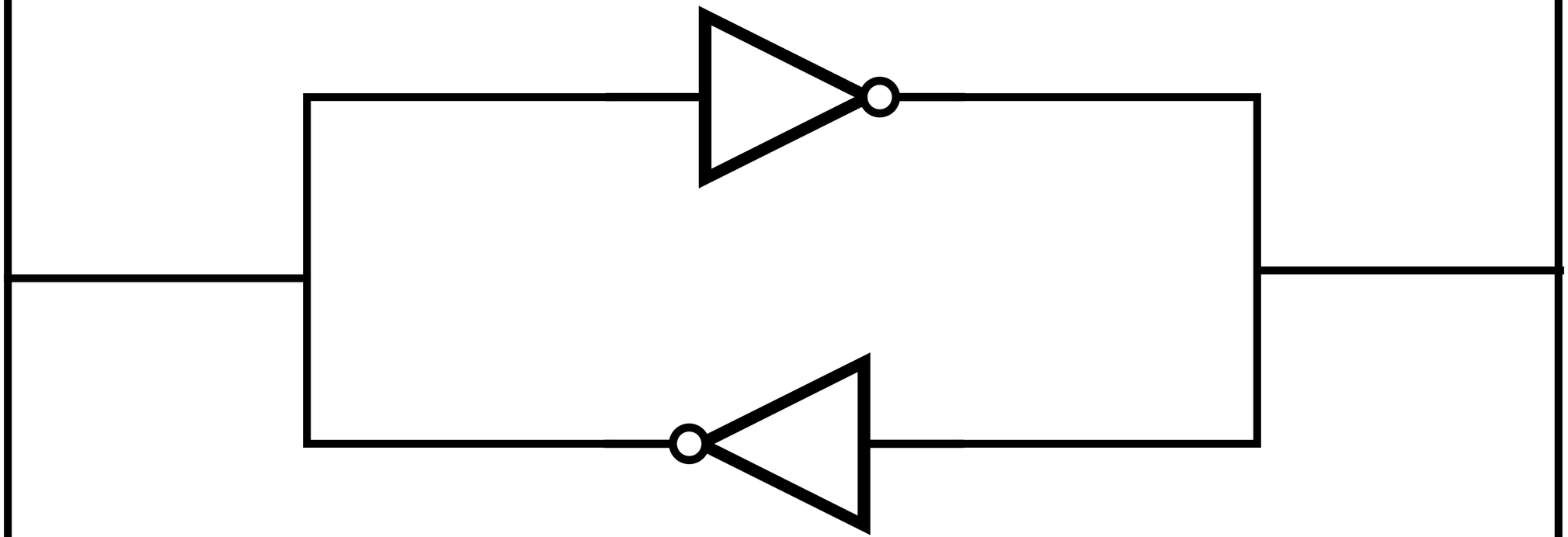
- `scanf` has similar syntax to printf, but passing by reference to **write** instead of read
- Can scan from
  - Standard input → `scanf`
  - File pointer → `fscanf`
  - String → `sscanf`
- Return value is how many variables were read
- `sscanf("354", "%d %d %d", &a, &b, &c) == 1`

Accessing Memory

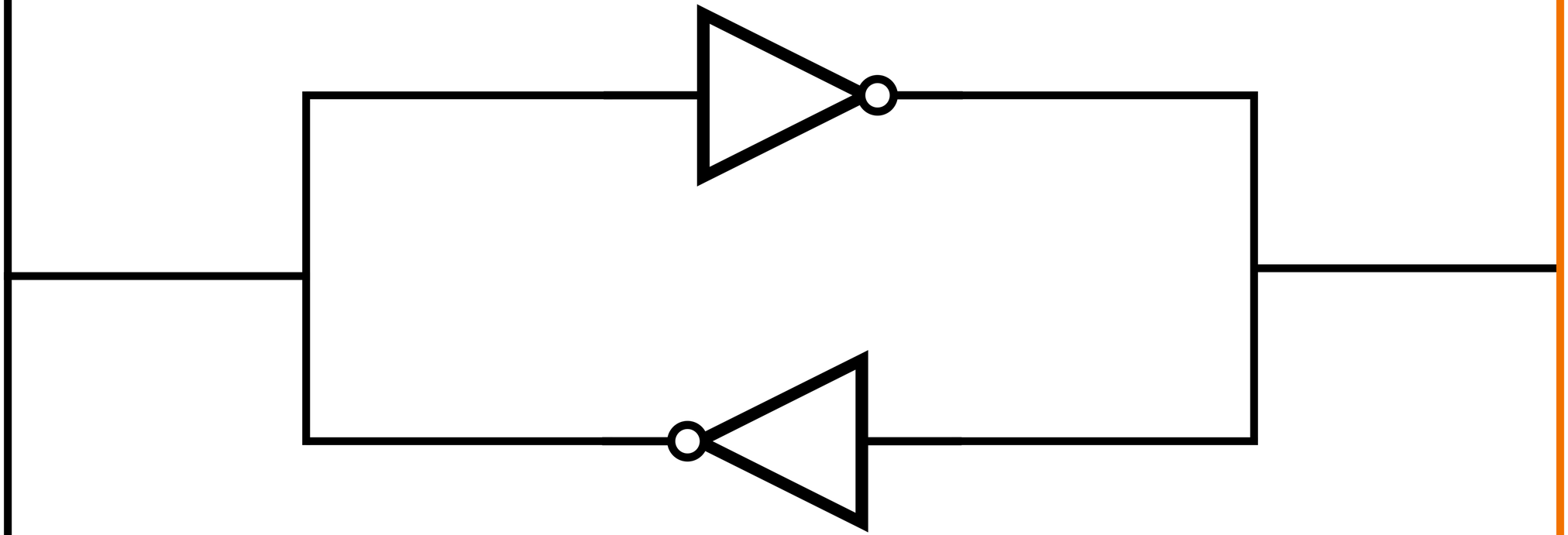
Flip-flop



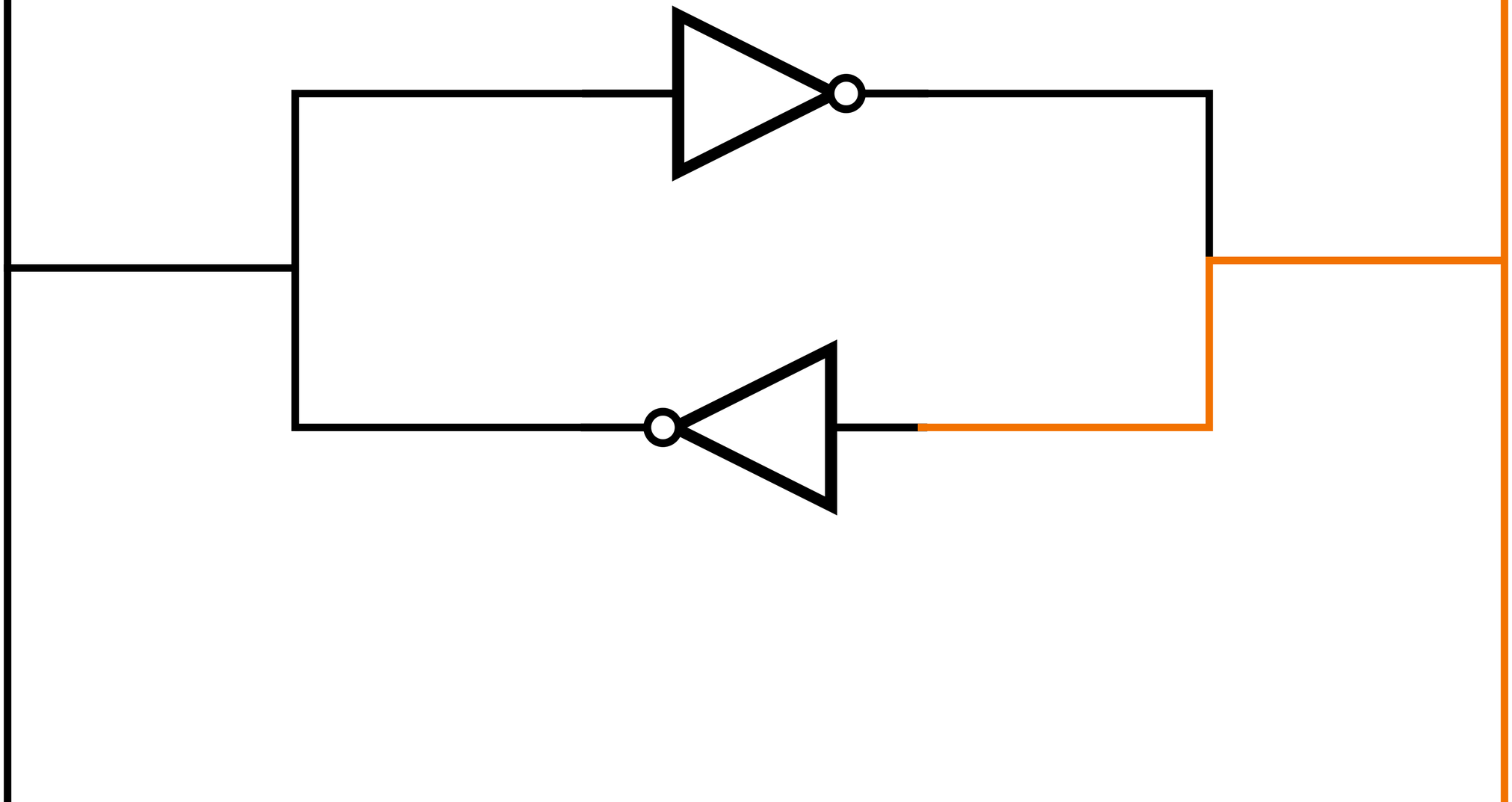
Flip-flop



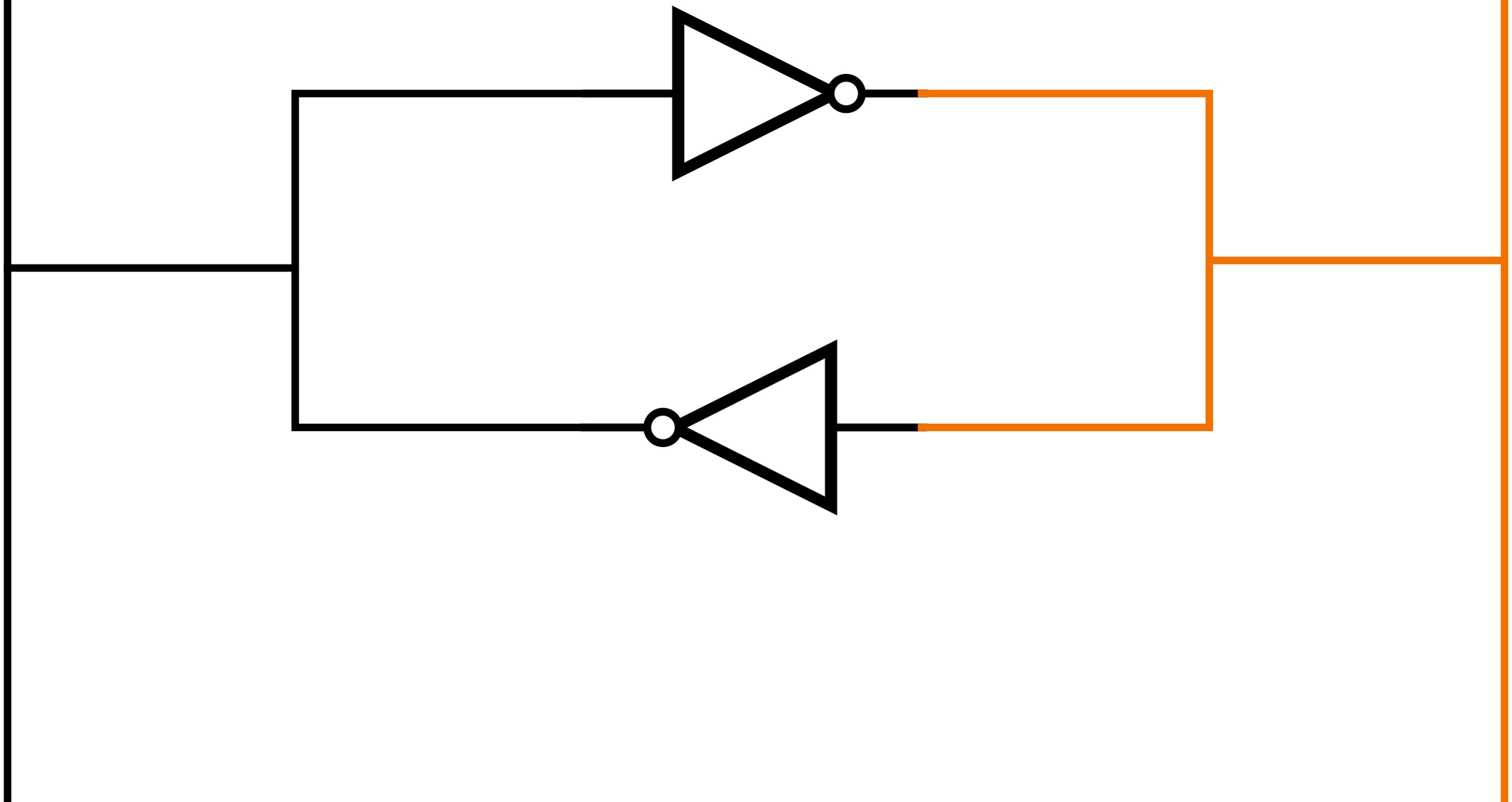
Flip-flop



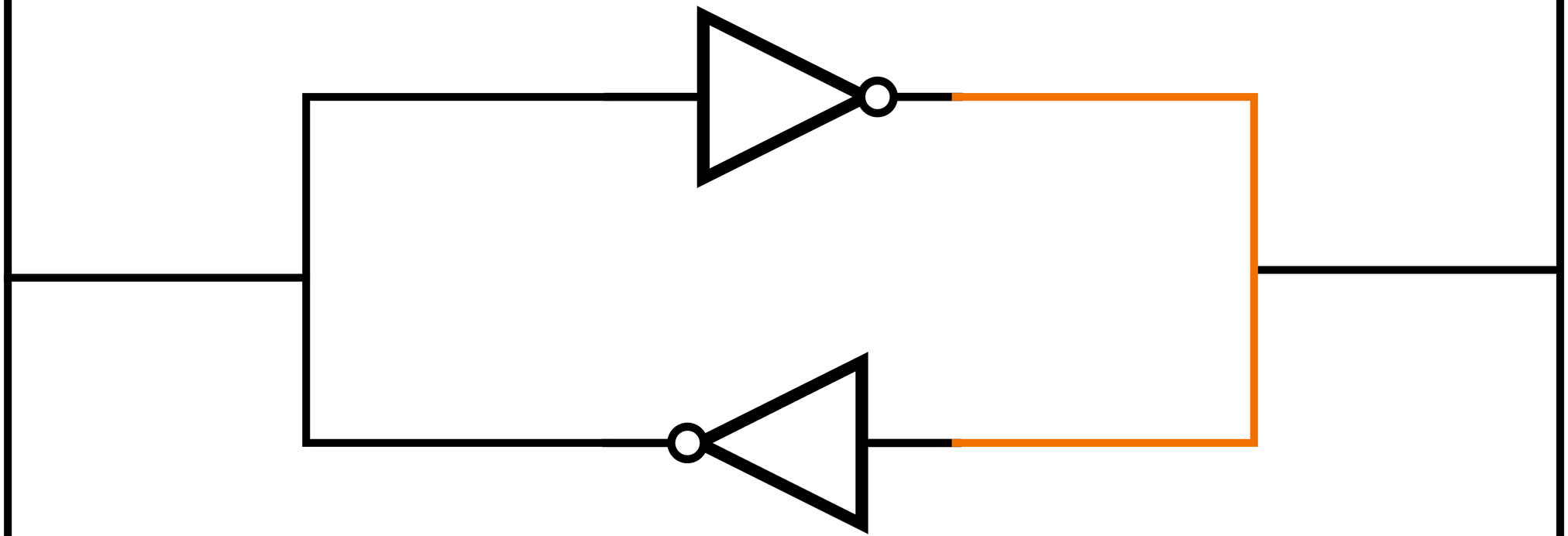
Flip-flop



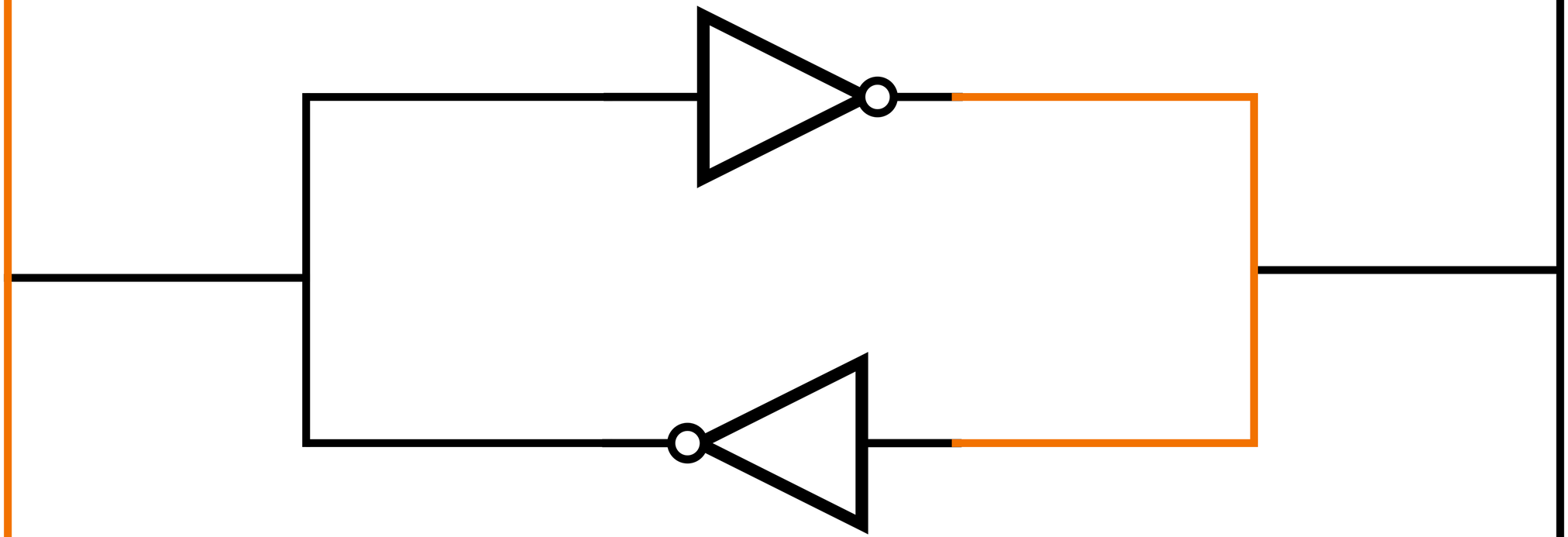
Flip-flop



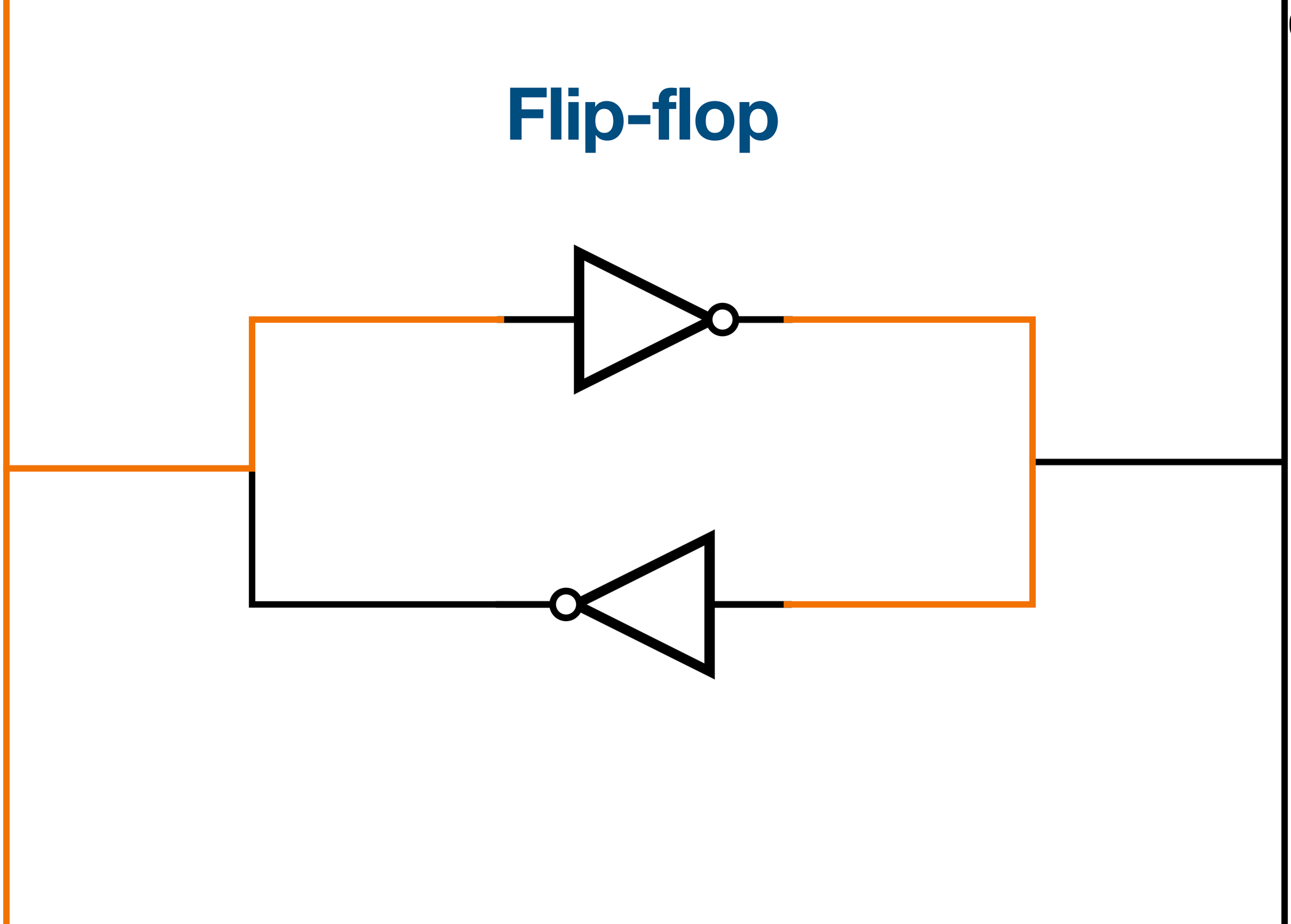
Flip-flop



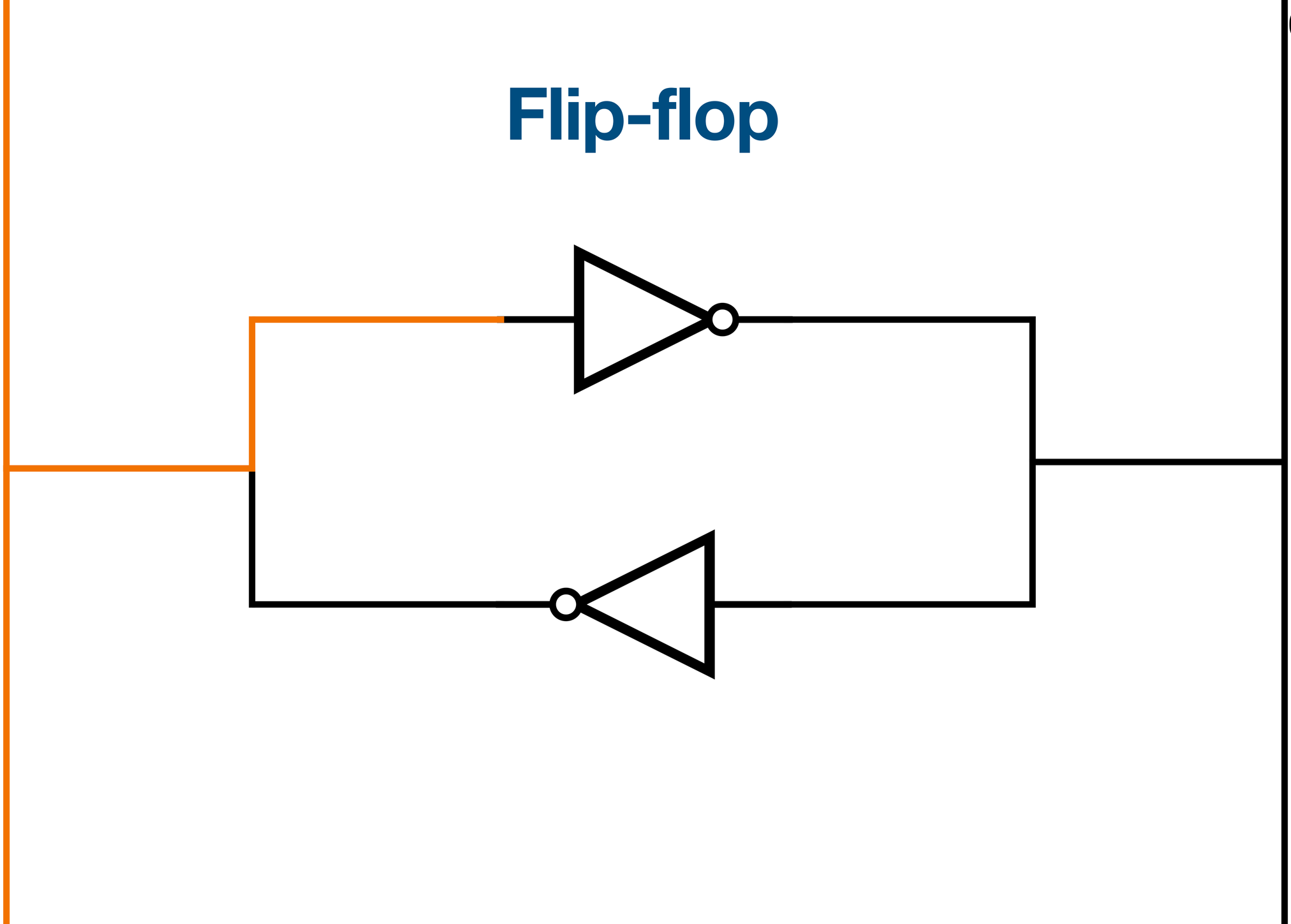
Flip-flop



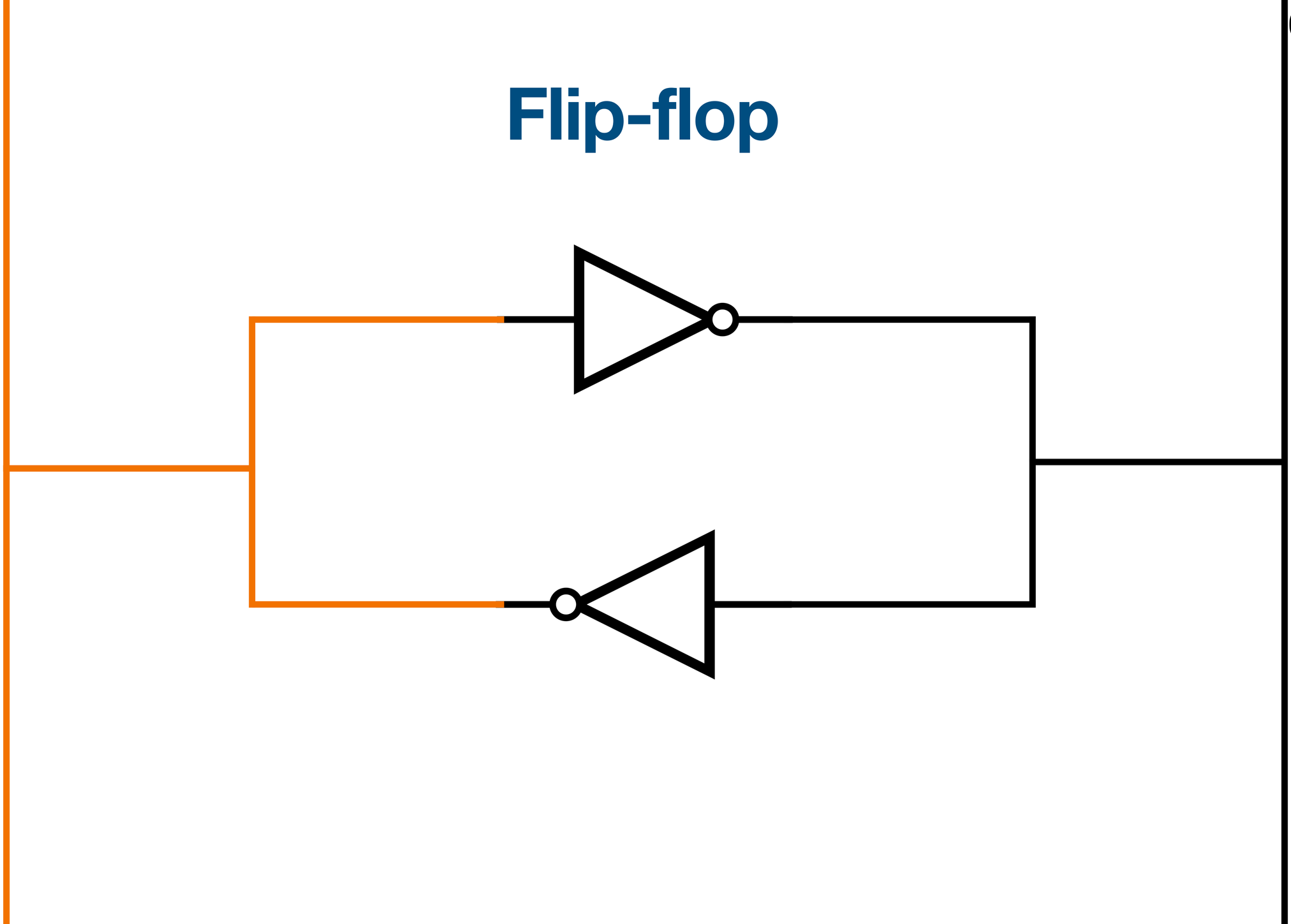
Flip-flop



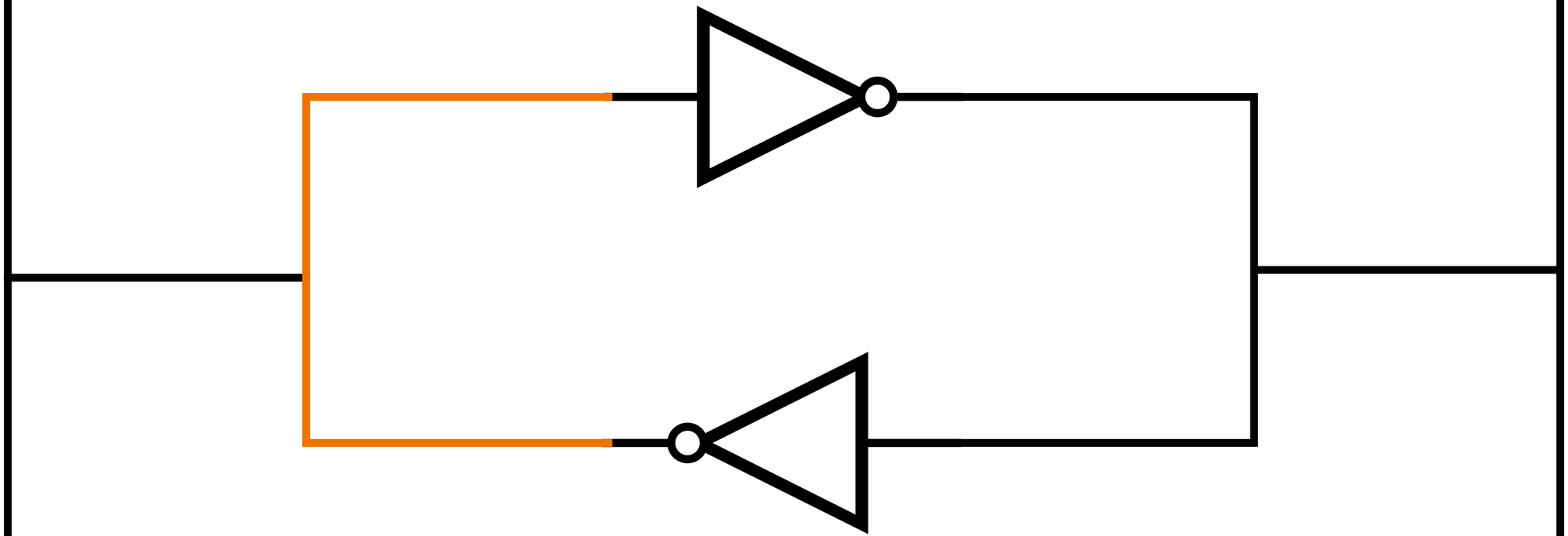
Flip-flop



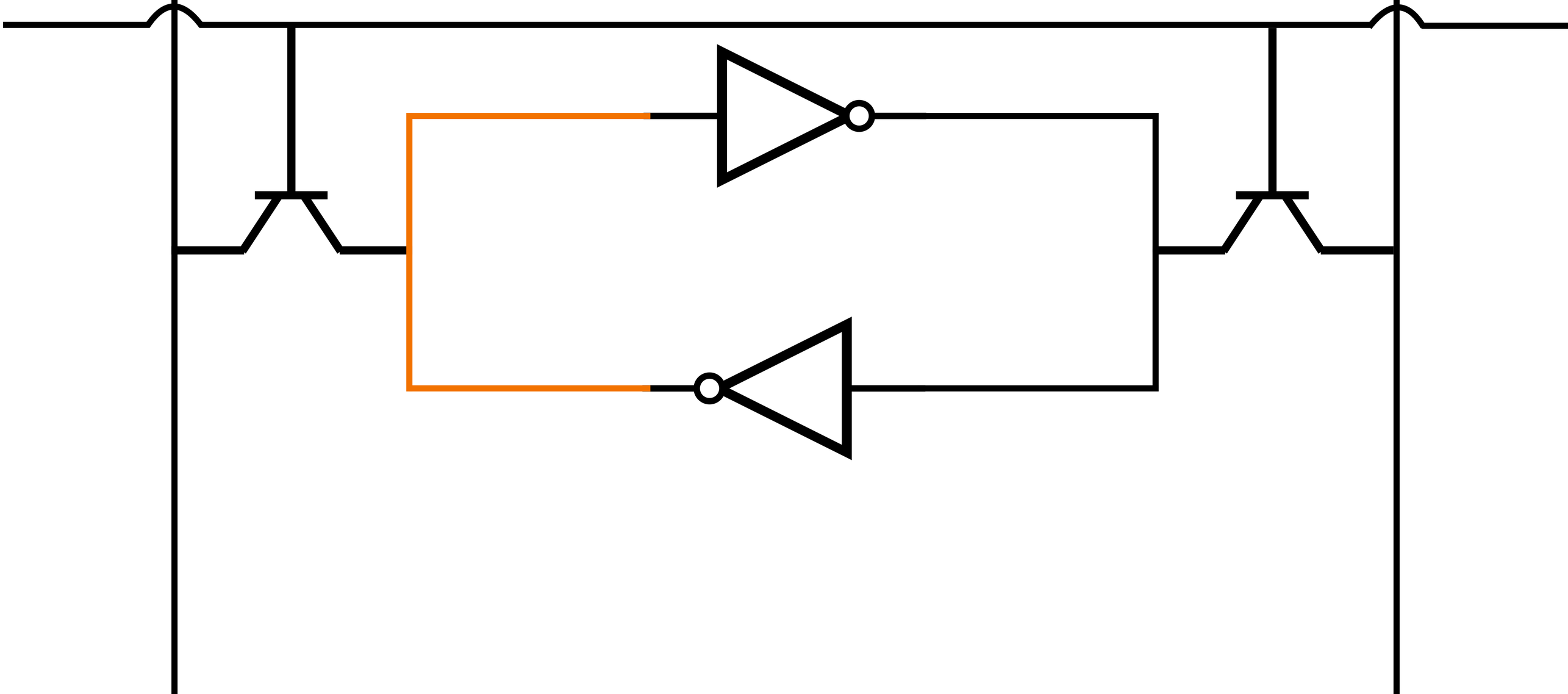
Flip-flop



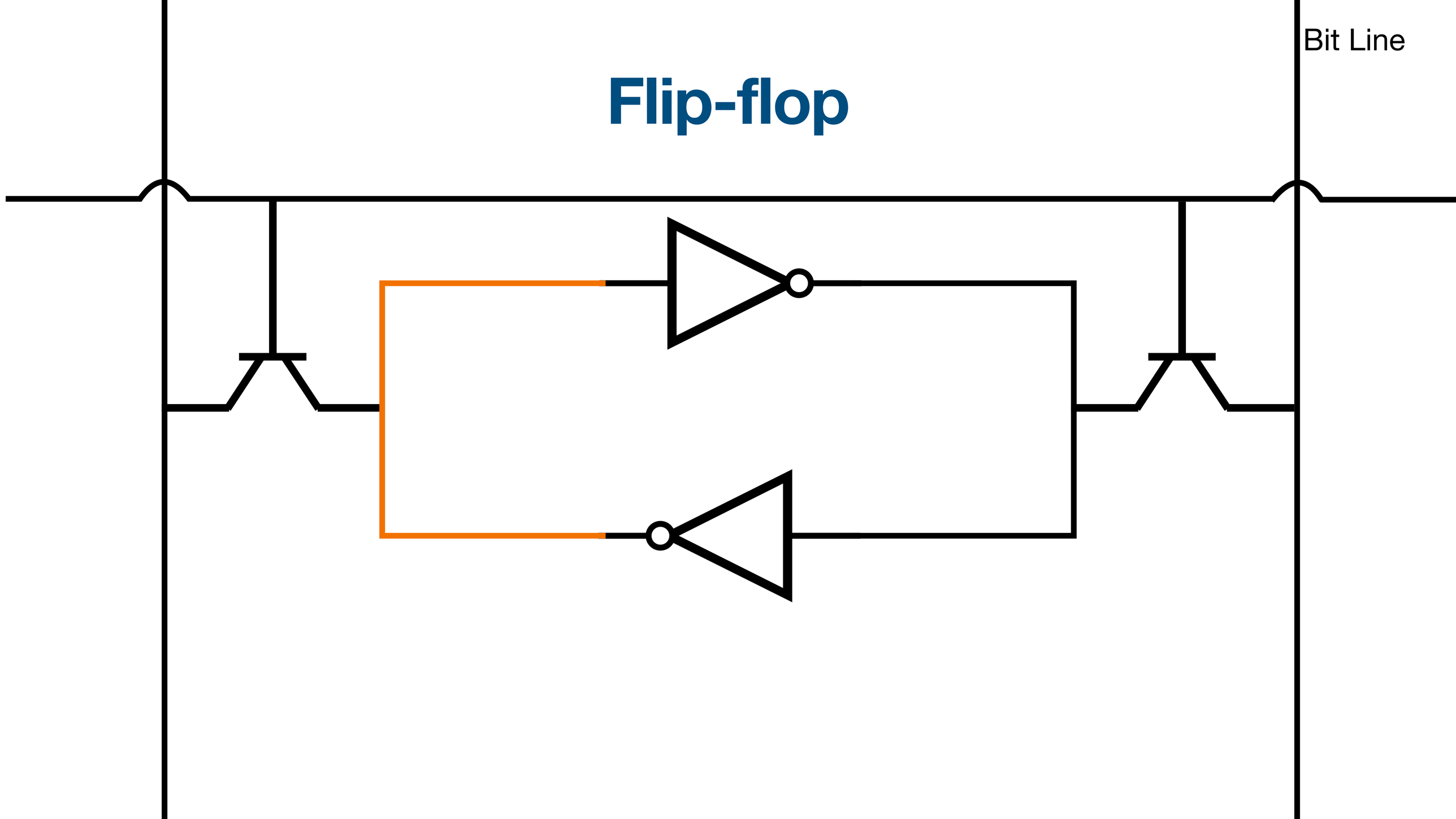
Flip-flop



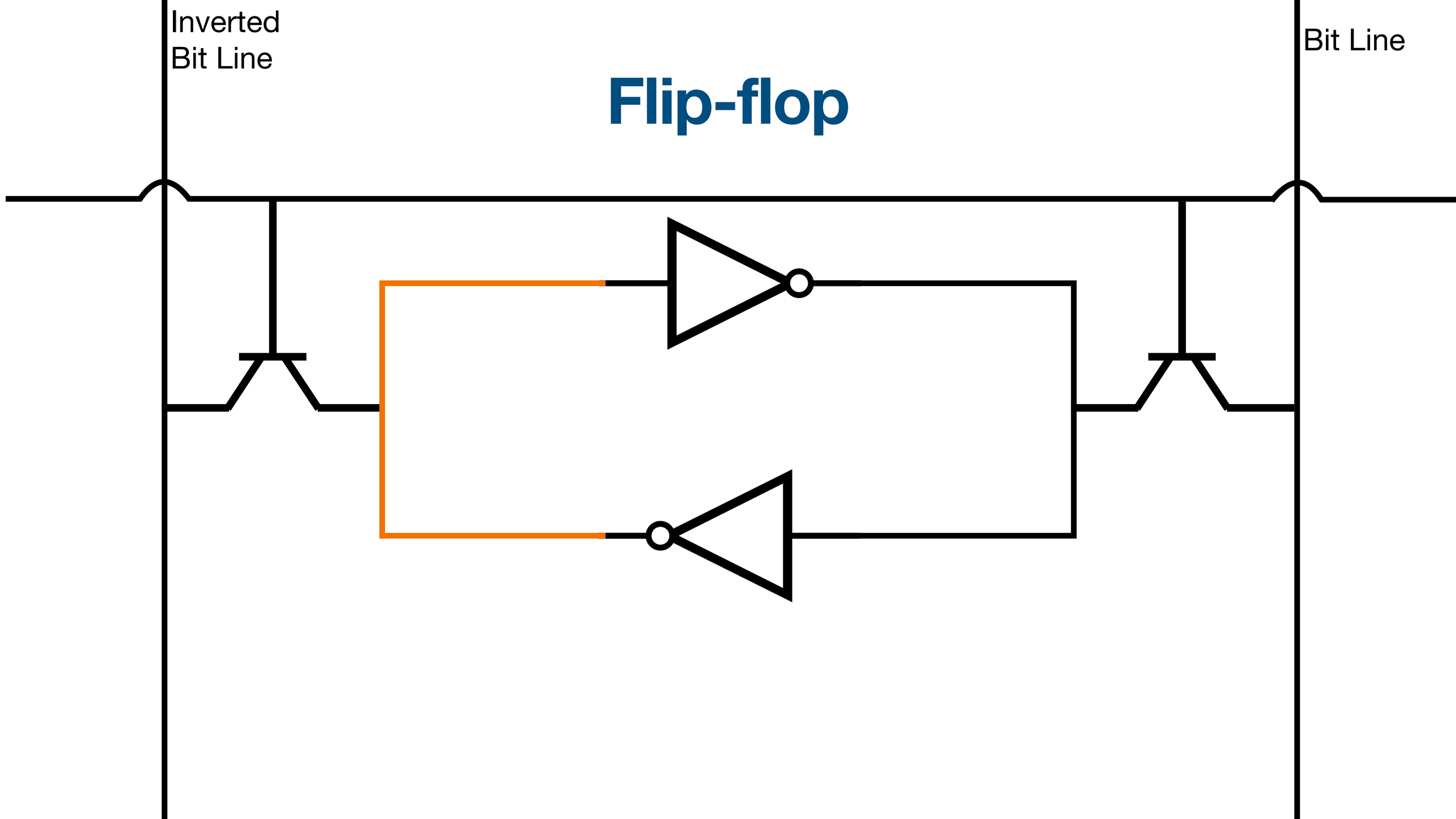
Flip-flop



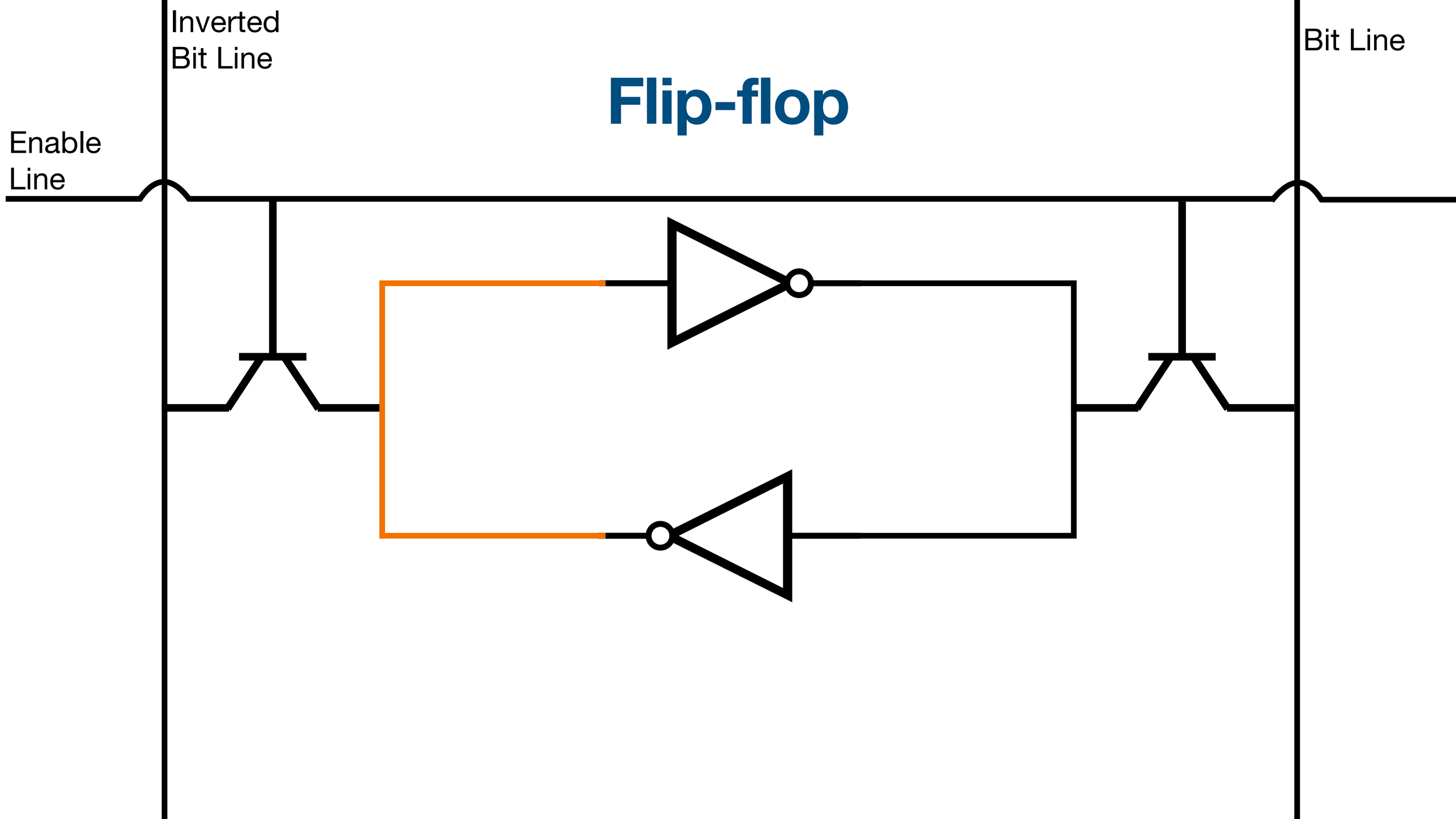
Flip-flop



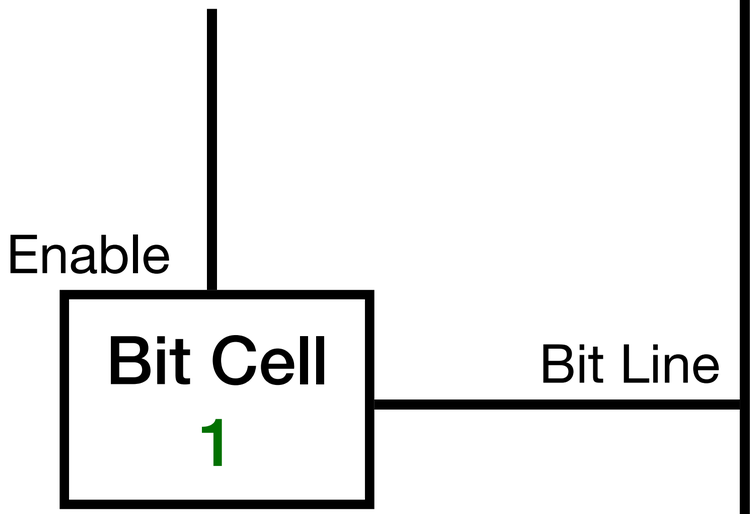
Flip-flop



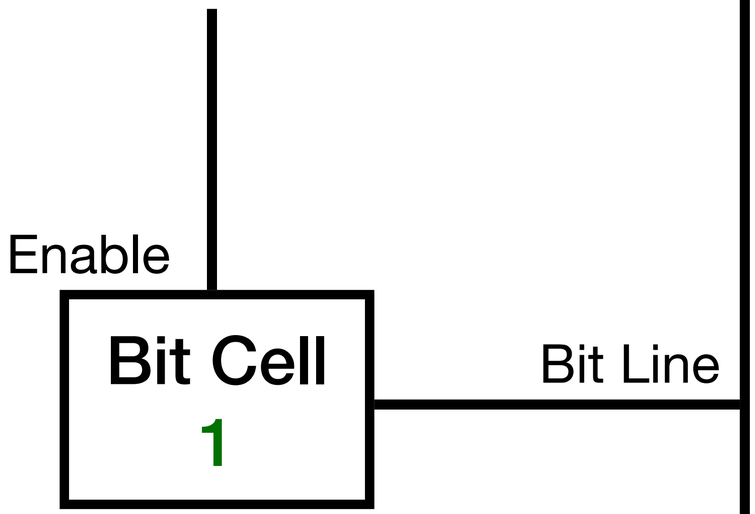
Flip-flop



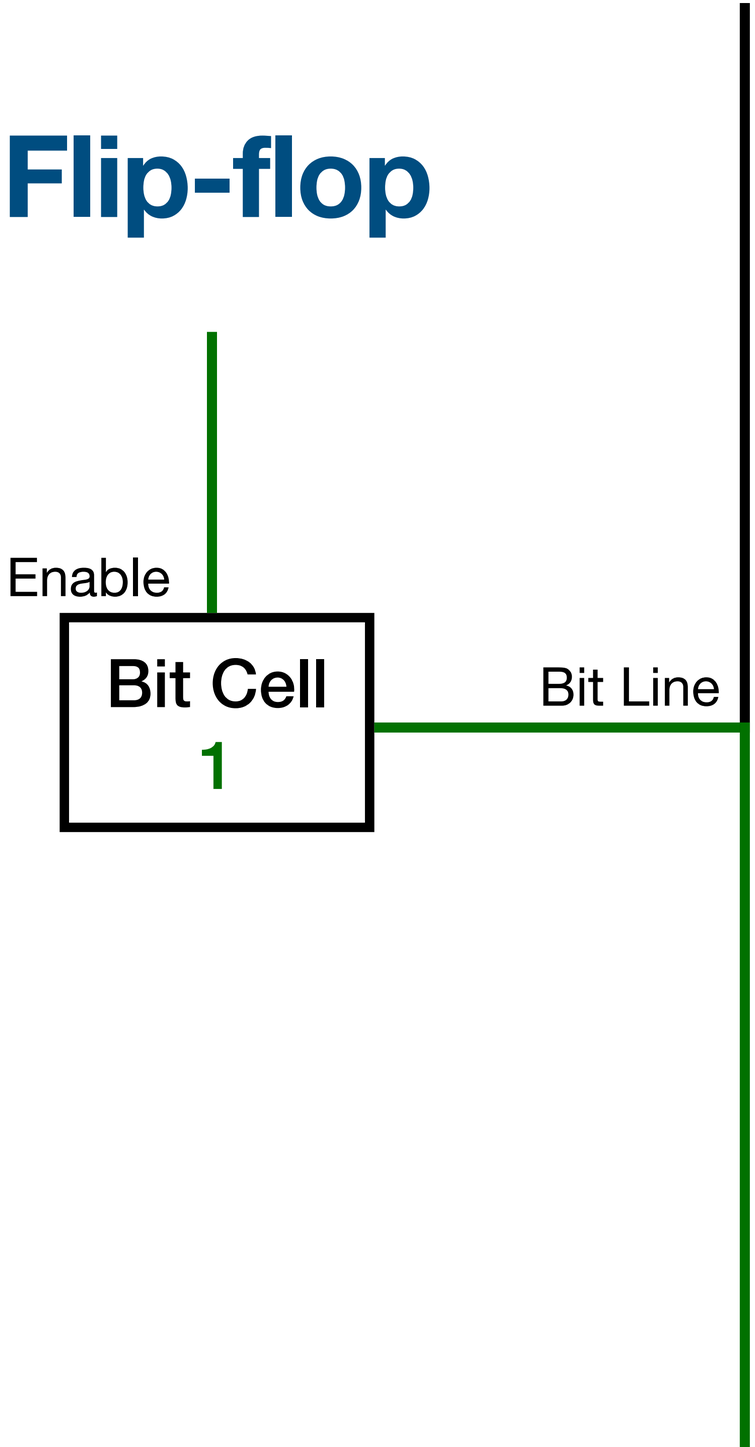
Flip-flop



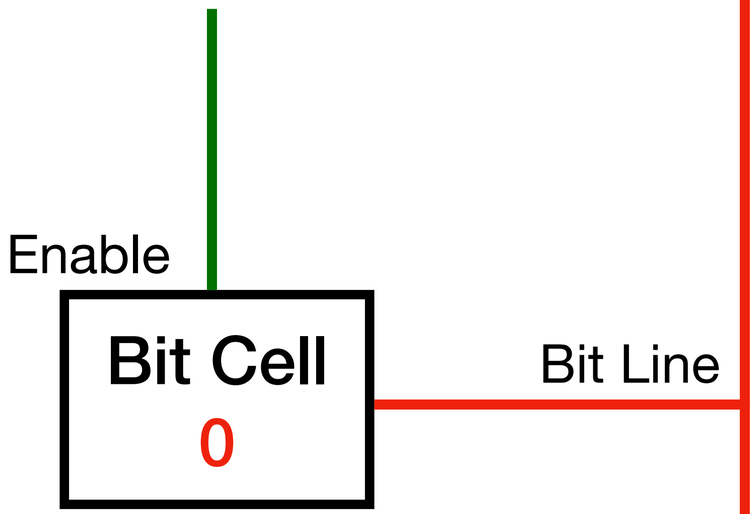
Flip-flop



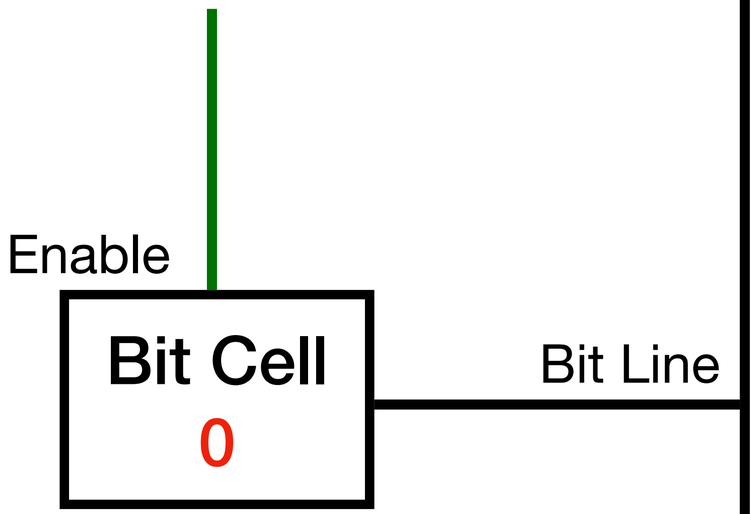
Flip-flop



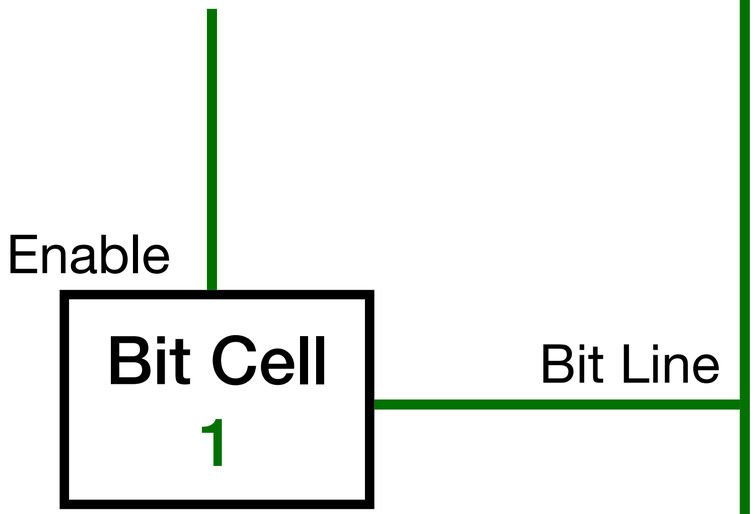
Flip-flop



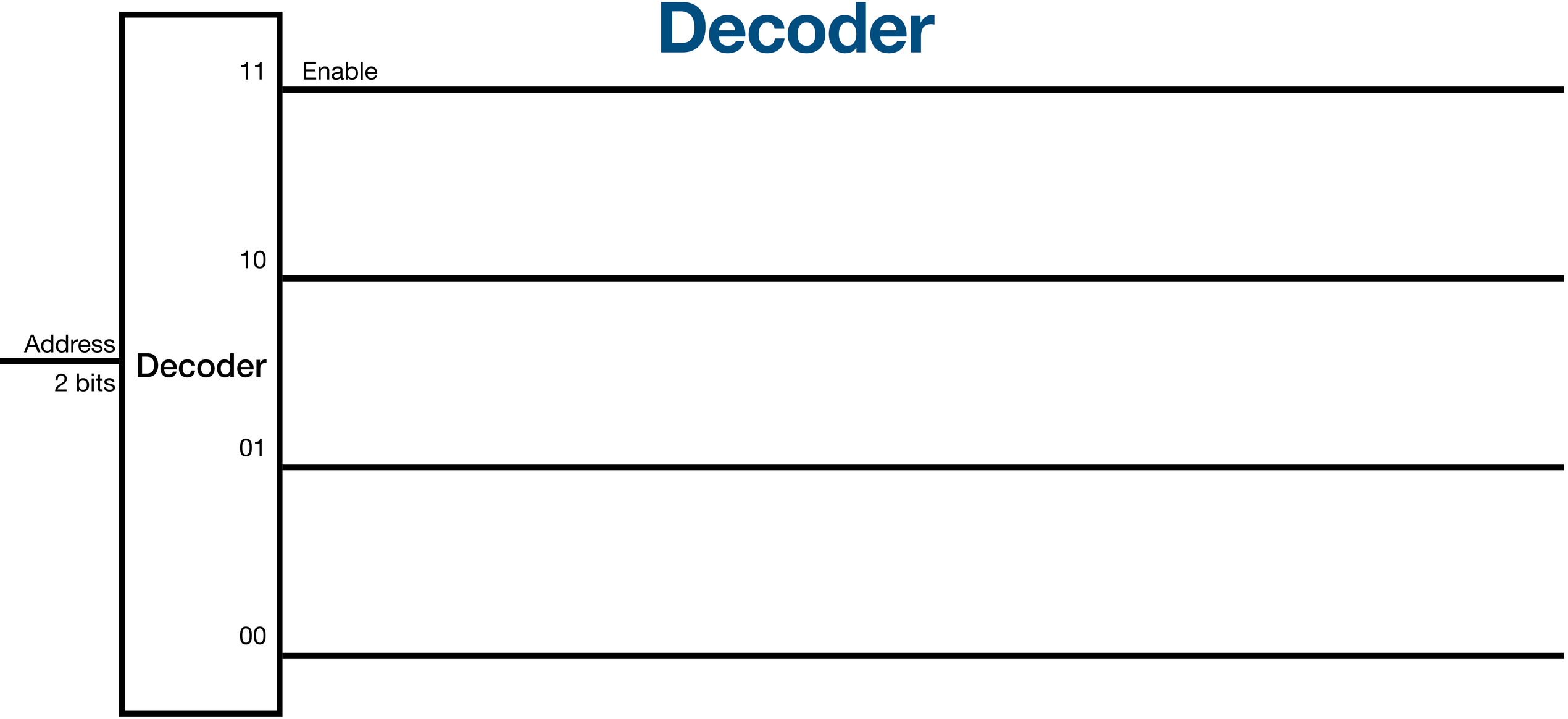
Flip-flop

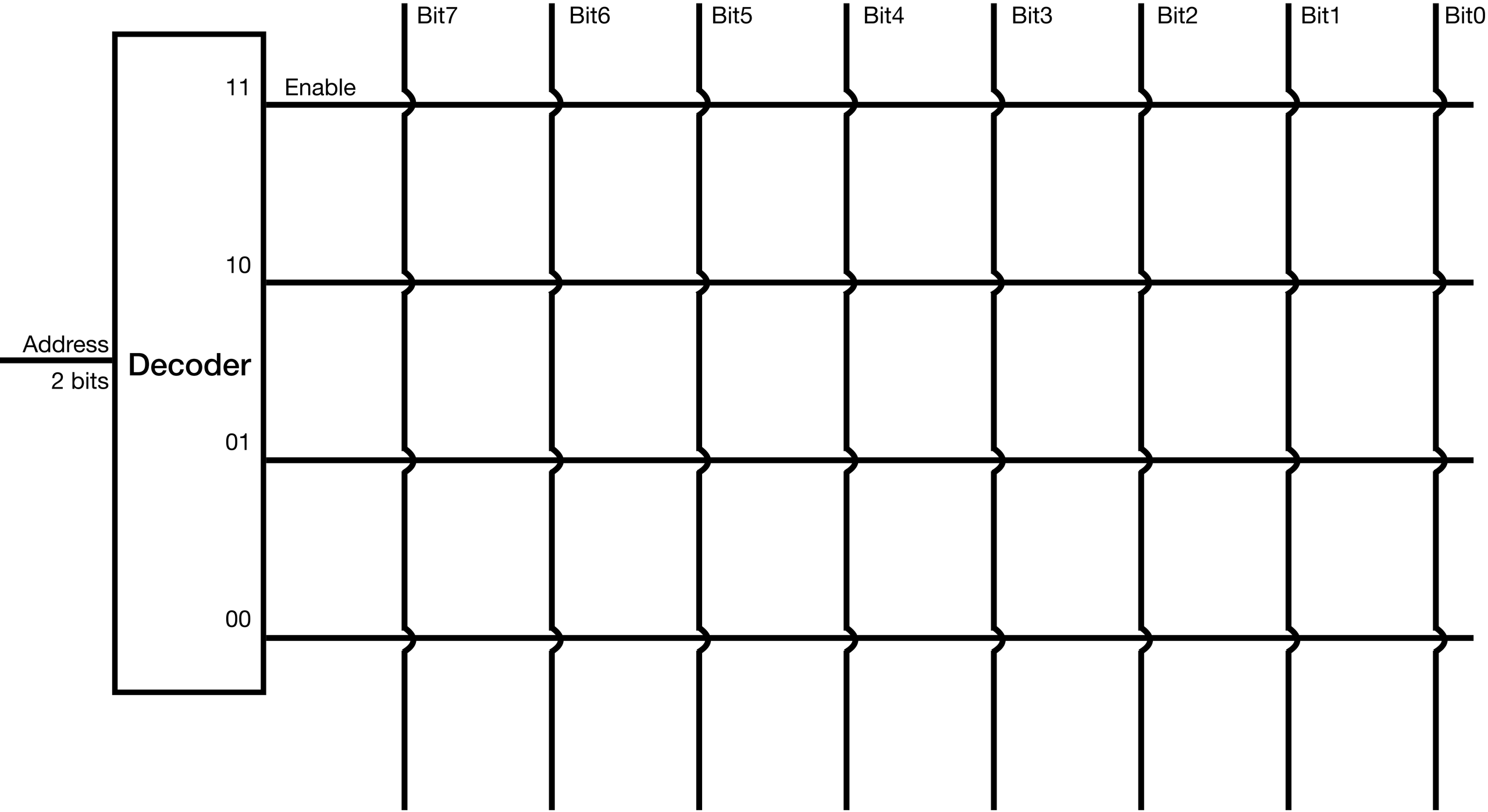


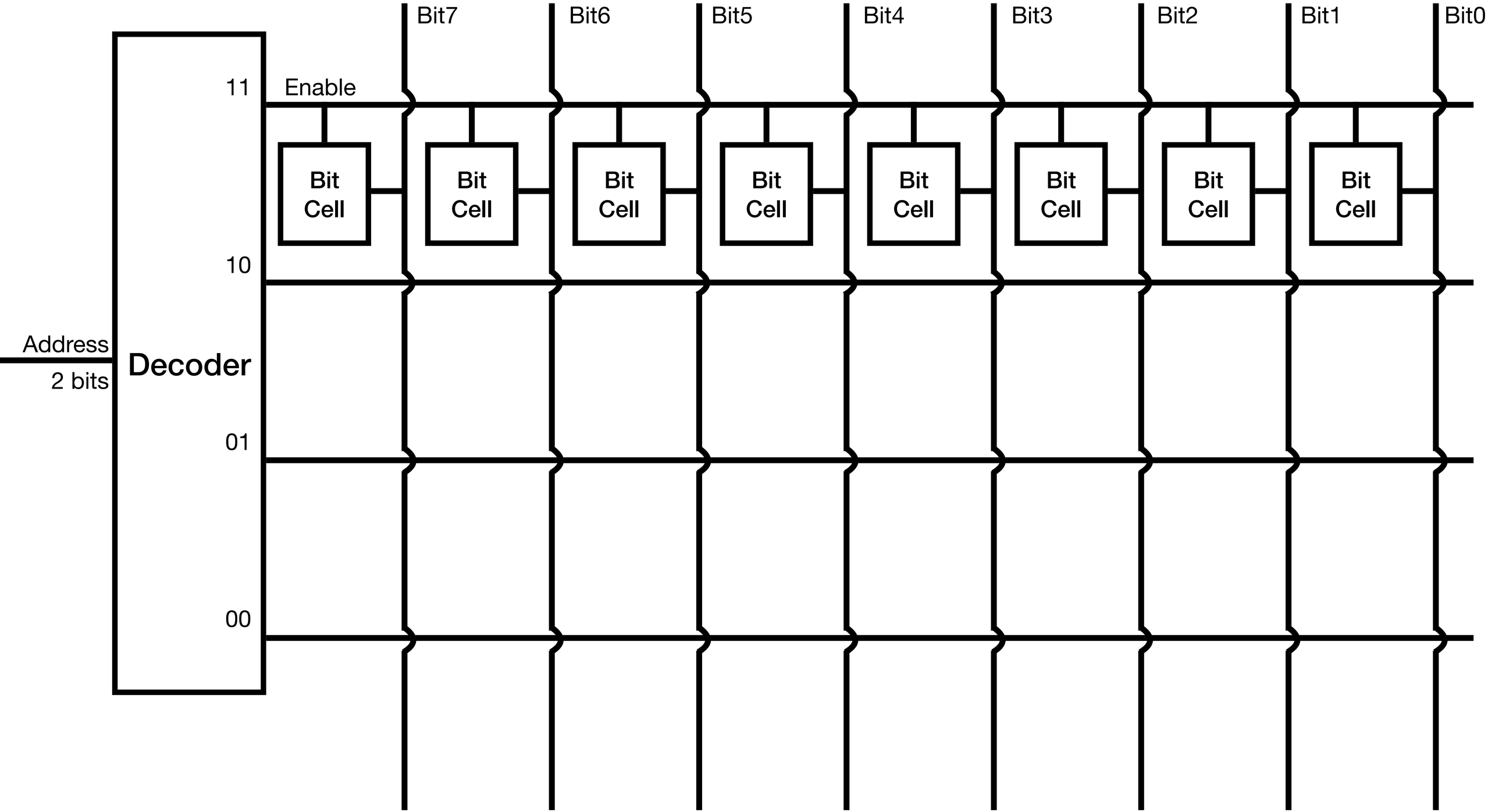
Flip-flop

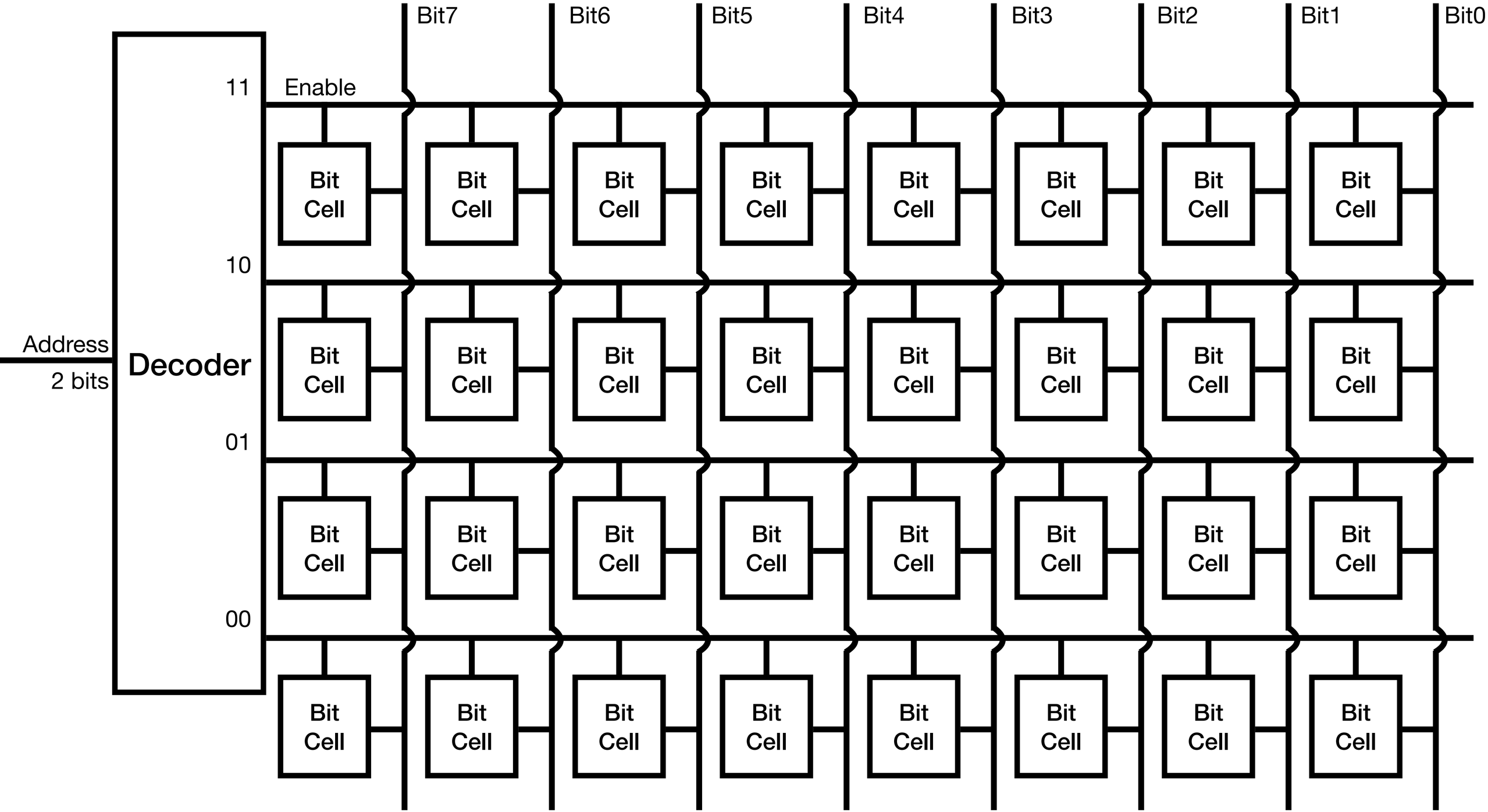


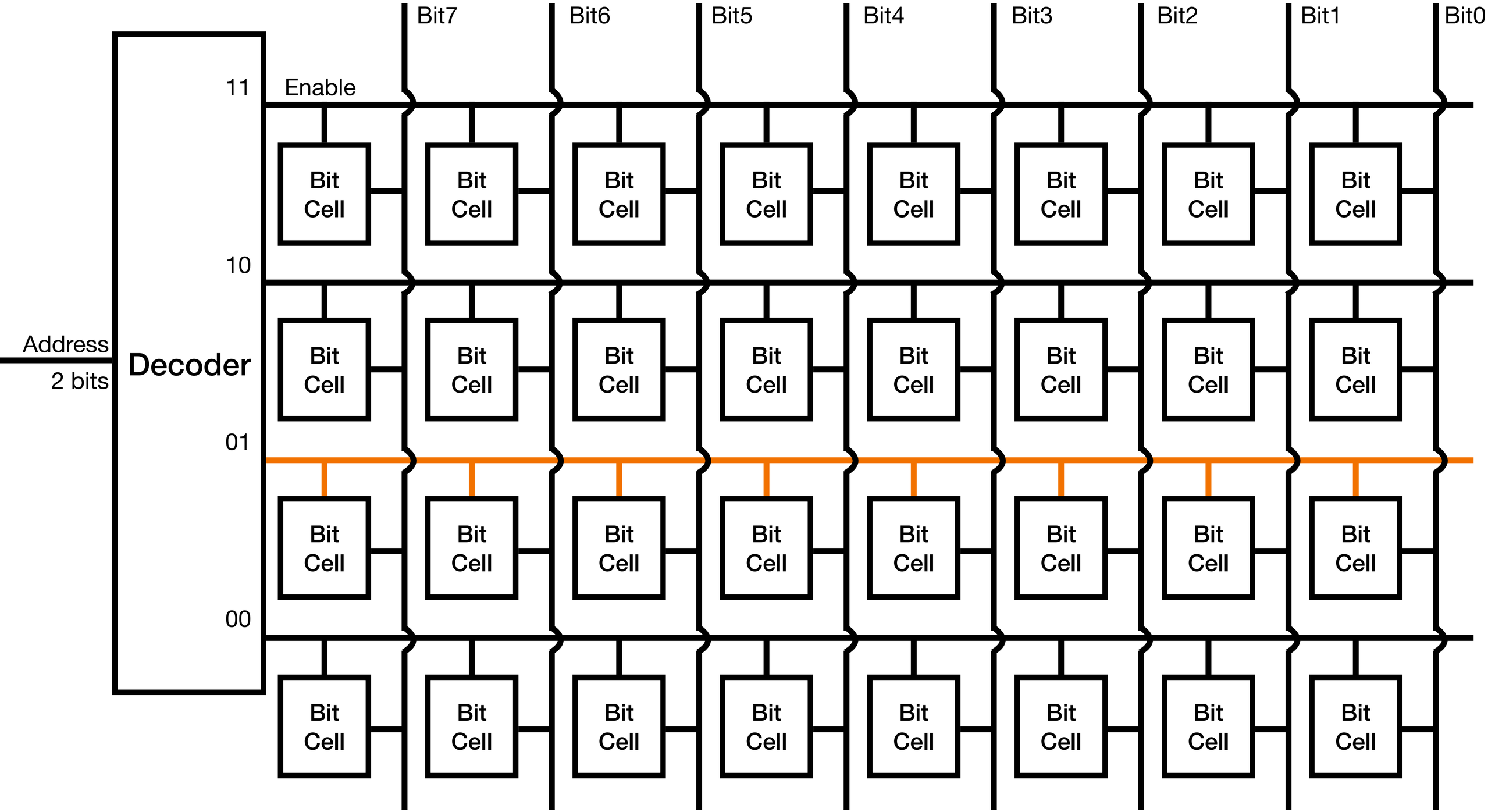
Decoder




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
10
01
00
Address
2 bits
Decoder
Bit Cell
Bit Cell
Bit Cell
Bit Cell
Bit Cell
Bit Cell
Bit Cell
Bit Cell


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

More pointers

Memory Array

Memory Array

- Each row of data is called a *word.* Most memories use 8-bit word, a *byte*.

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.

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.

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.

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?

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

Memory Array

- 2^{32} = 4,294,967,296 = ~4.3 G of addressable rows.

Memory Array

- 2^{32} = 4,294,967,296 = ~4.3 G of addressable rows.
- 4.2 gigabytes of addressable memory.

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.

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

Endian

- We think of an integer as one atomic value:

Endian

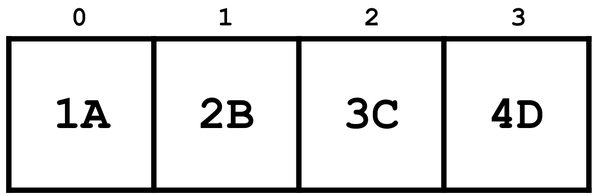
- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`

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?

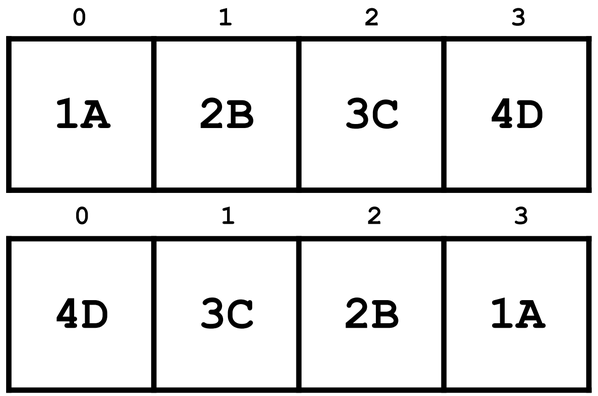
Endian

- 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?



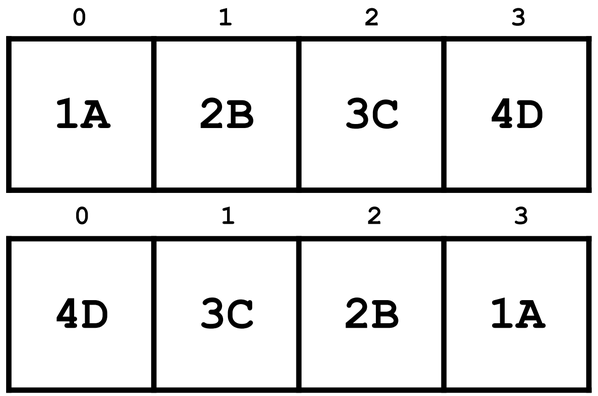
Endian

- 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

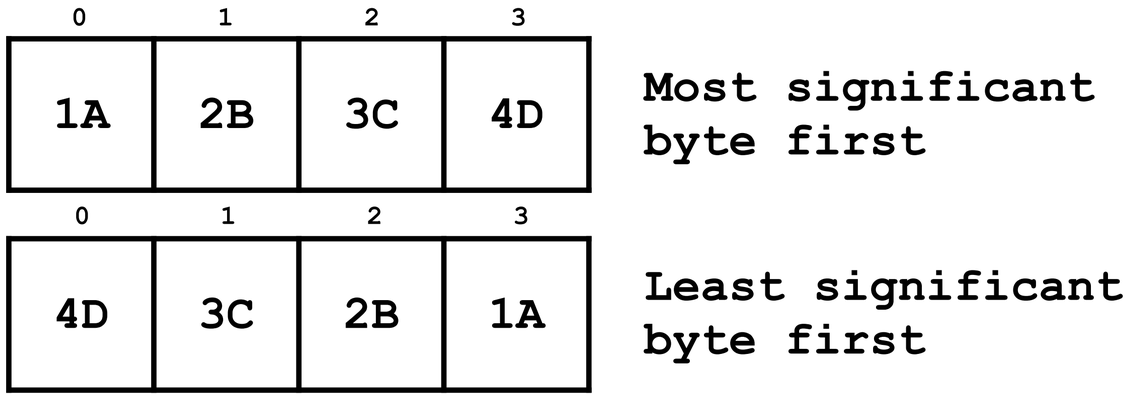
- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`
- But if an integer has 4 bytes and each byte is addressable, which of the 4 bytes is stored first?



Most significant byte first

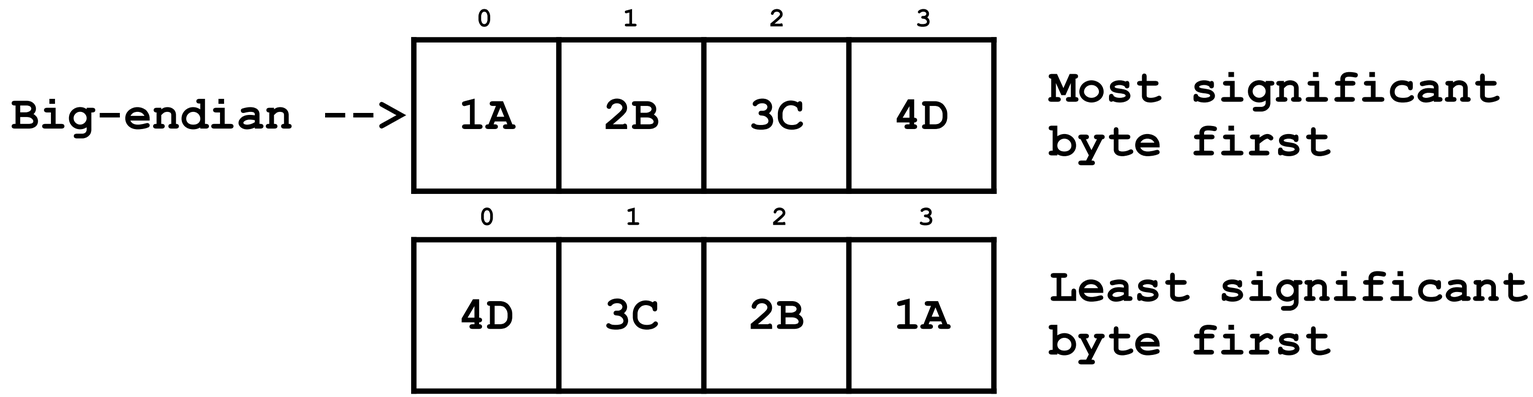
Endian

- 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?



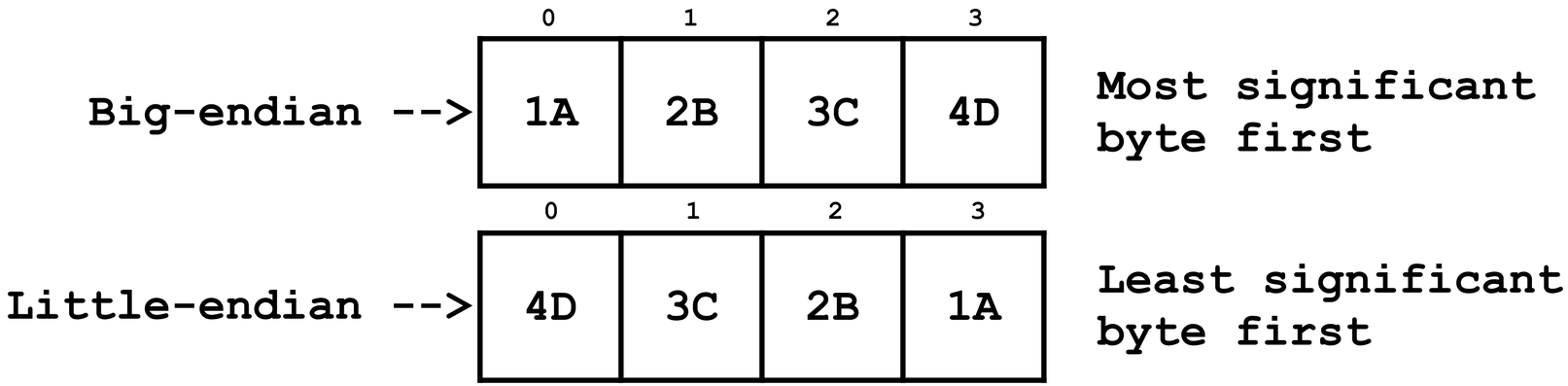
Endian

- 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

- 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

Endian

- Is my machine little-endian or big-endian?

Endian

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

Endian

Endian

- Our machine is little-endian?????

Endian

- Our machine is little-endian?????
- We usually write numbers in big-endian: 345 is three hundred and forty-five

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:

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.)

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`

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`

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

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?

Endian

Does it matter?

- Mostly we don't care. Unless you do memory trickery, variables work as you would expect

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:

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

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

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

Endian

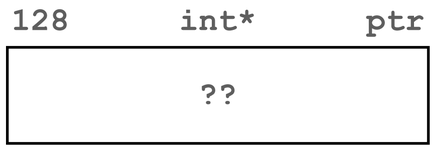
Does it matter?

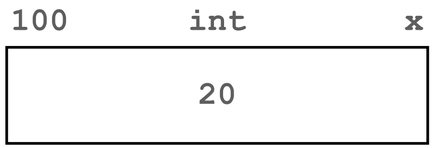
- Mostly we don't care. Unless you do memory trickery, variables work as you would expect
- However, when we serialize data into byte sequences, you need to pay extra attention:
  - Writing a number to a file
  - Sending a number over a network
- You and the reader must agree on byte order
  - For this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

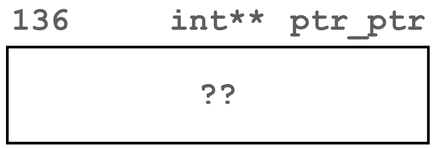
Pointers

Review

```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;
```



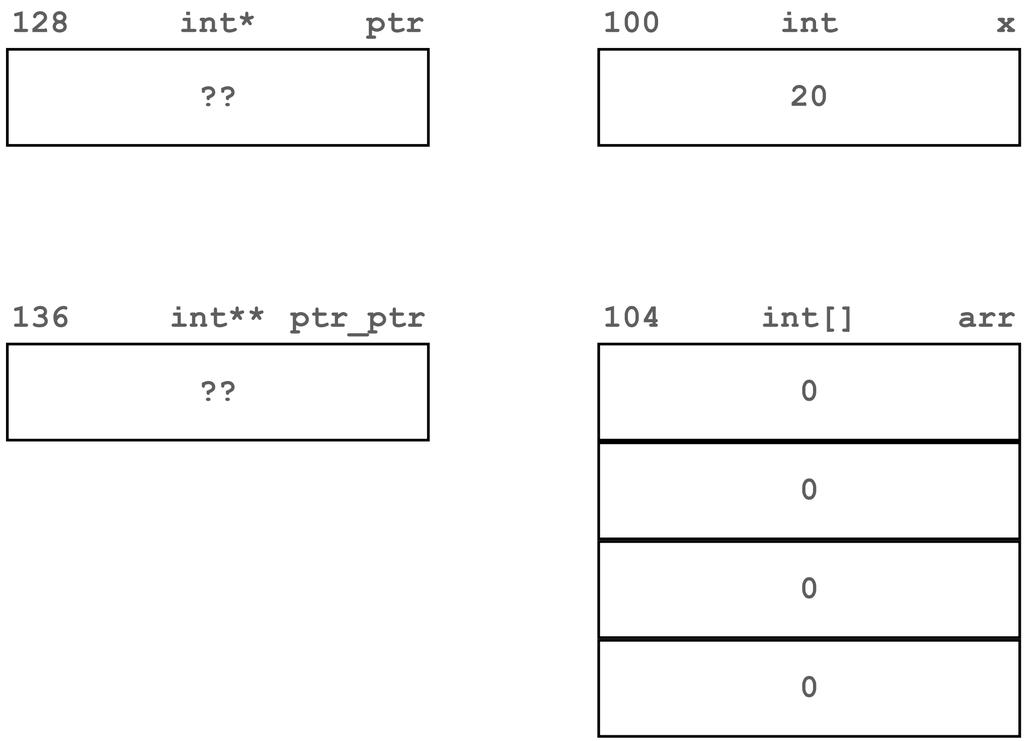




Pointers

Review

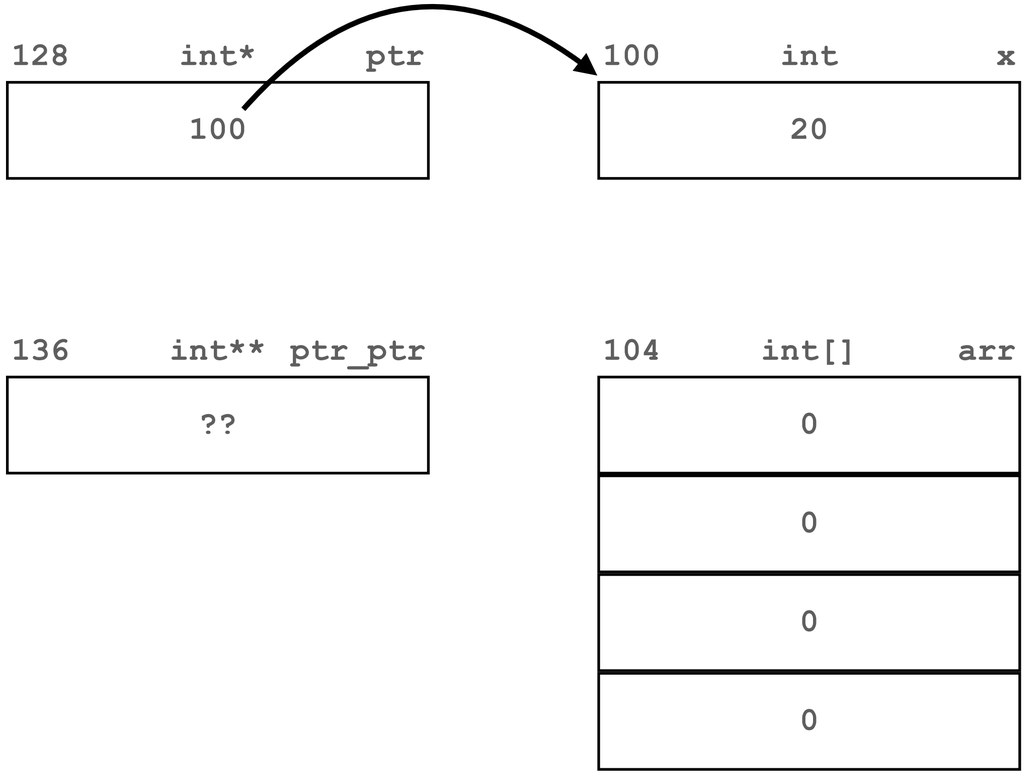
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
```



Pointers

Review

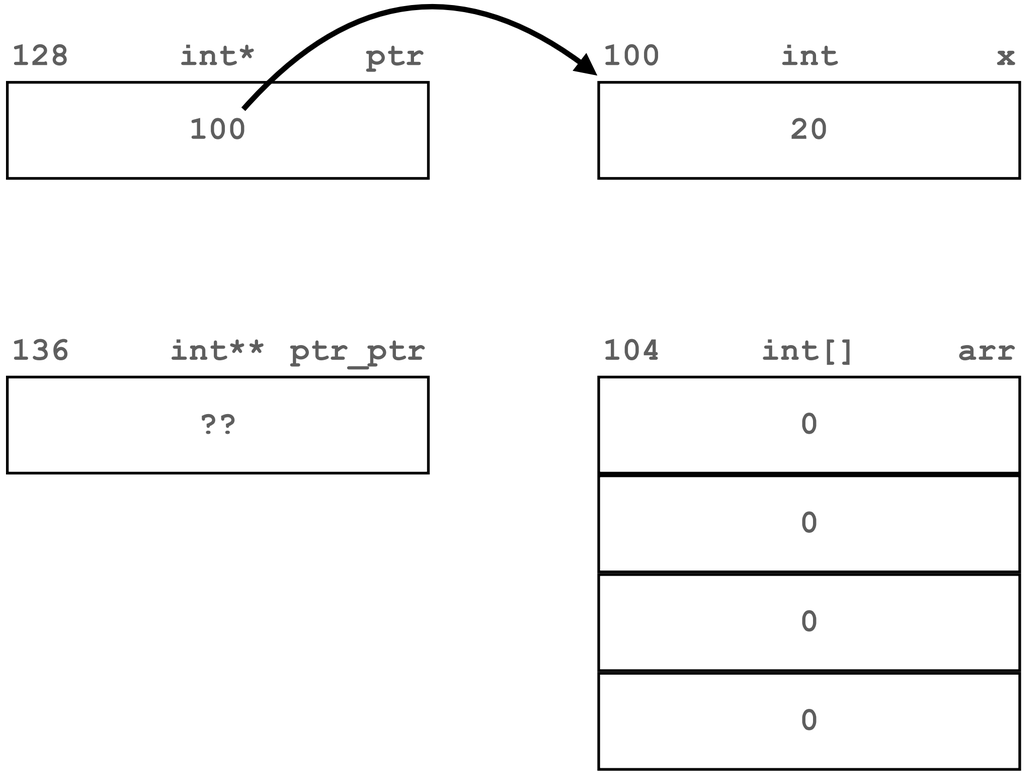
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
```



Pointers

Review

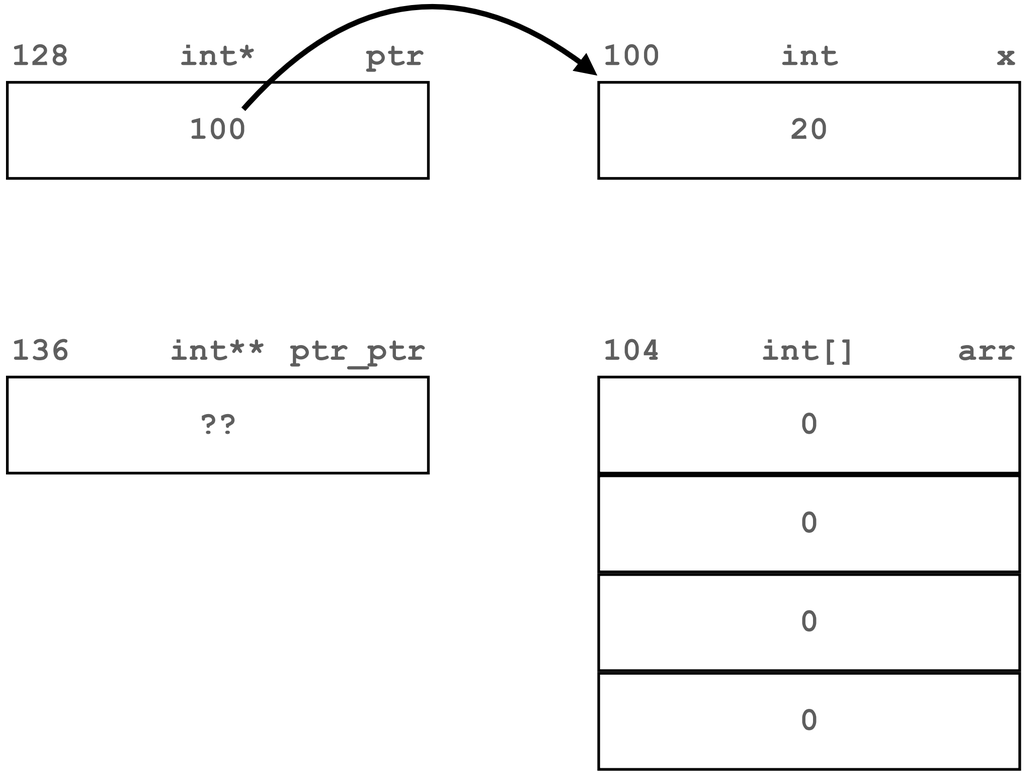
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);
```



Pointers

Review

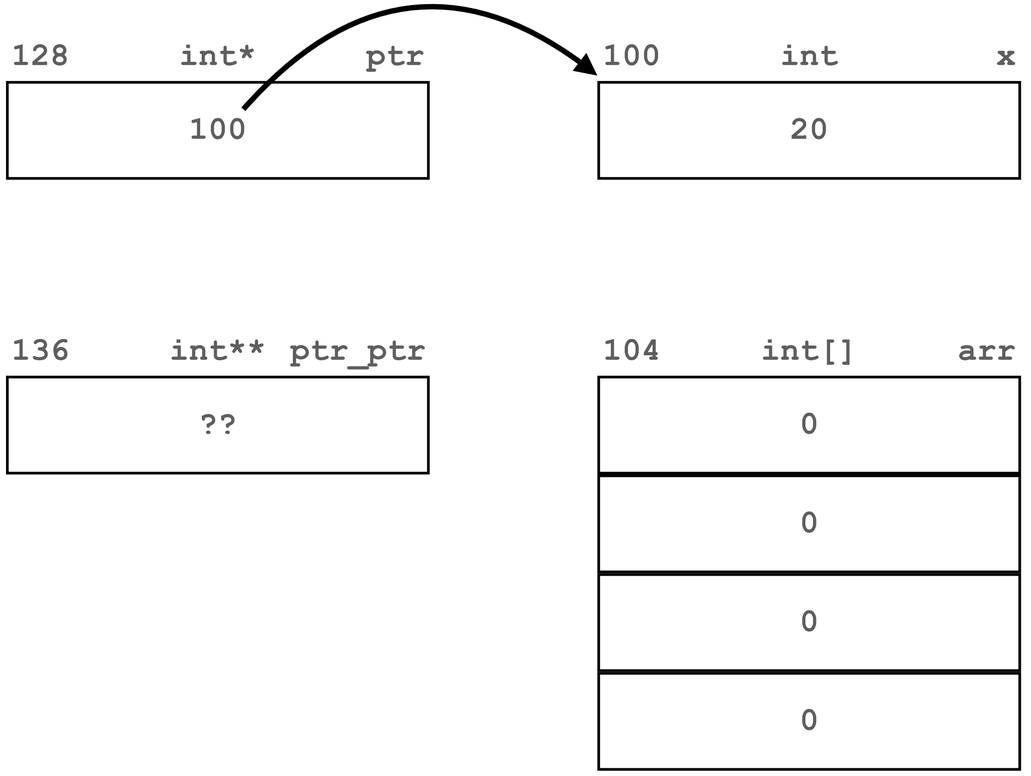
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);       <-- 20
```



Pointers

Review

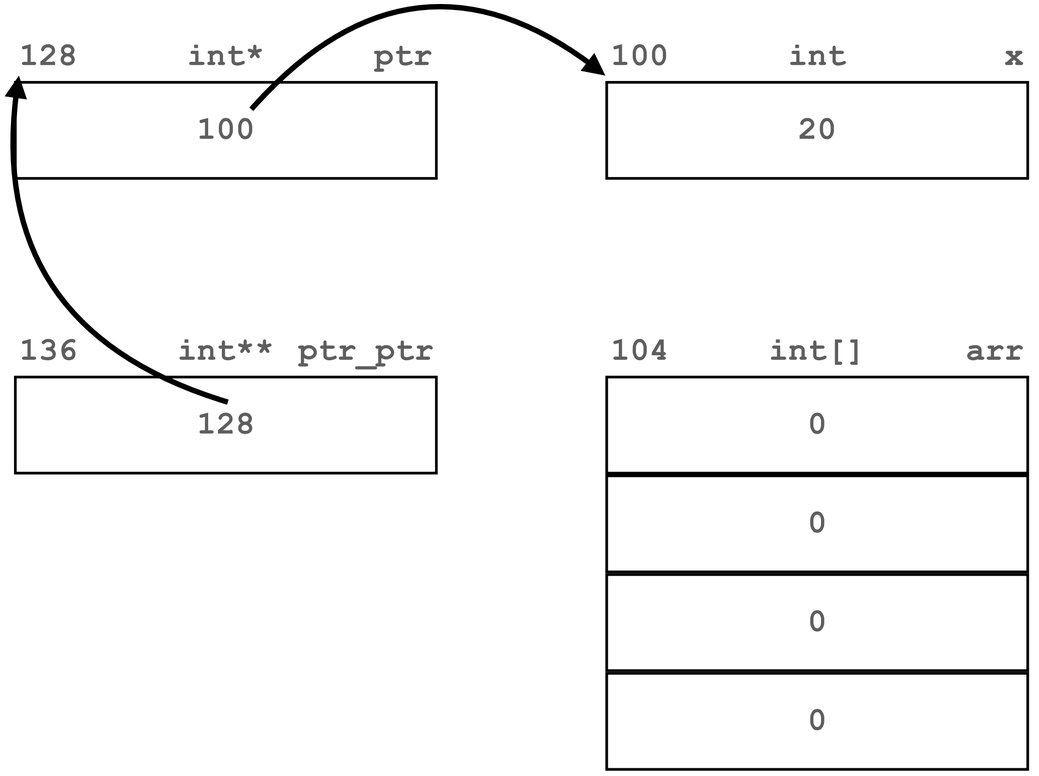
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
```



Pointers

Review

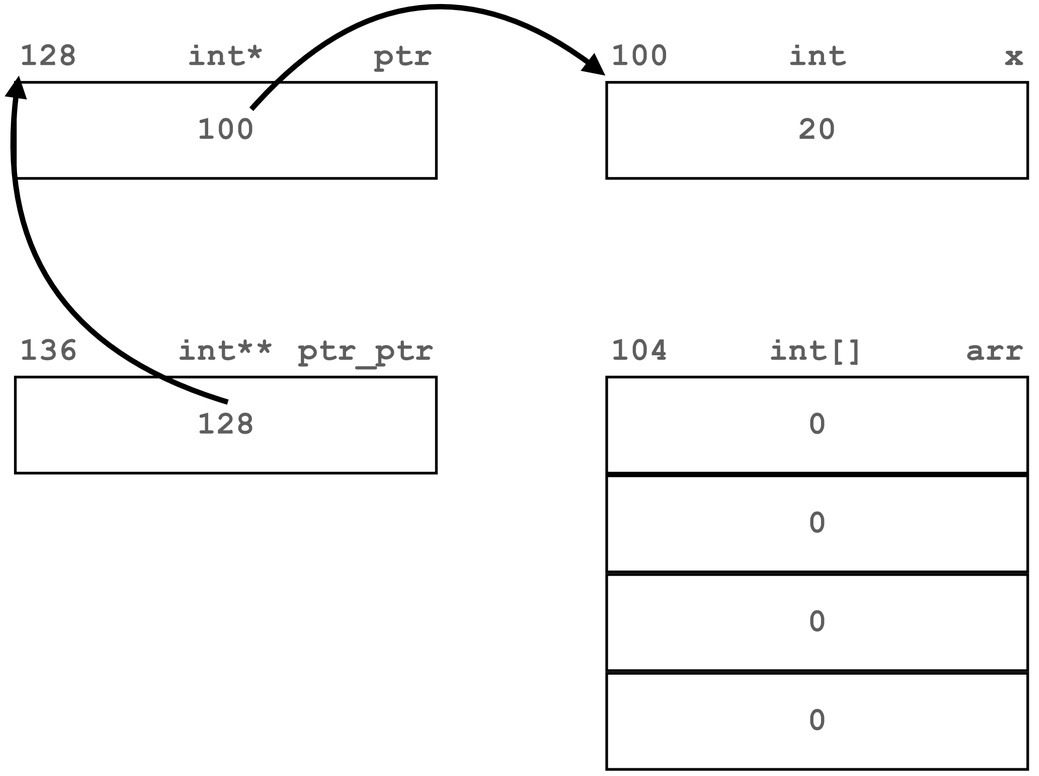
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
```



Pointers

Review

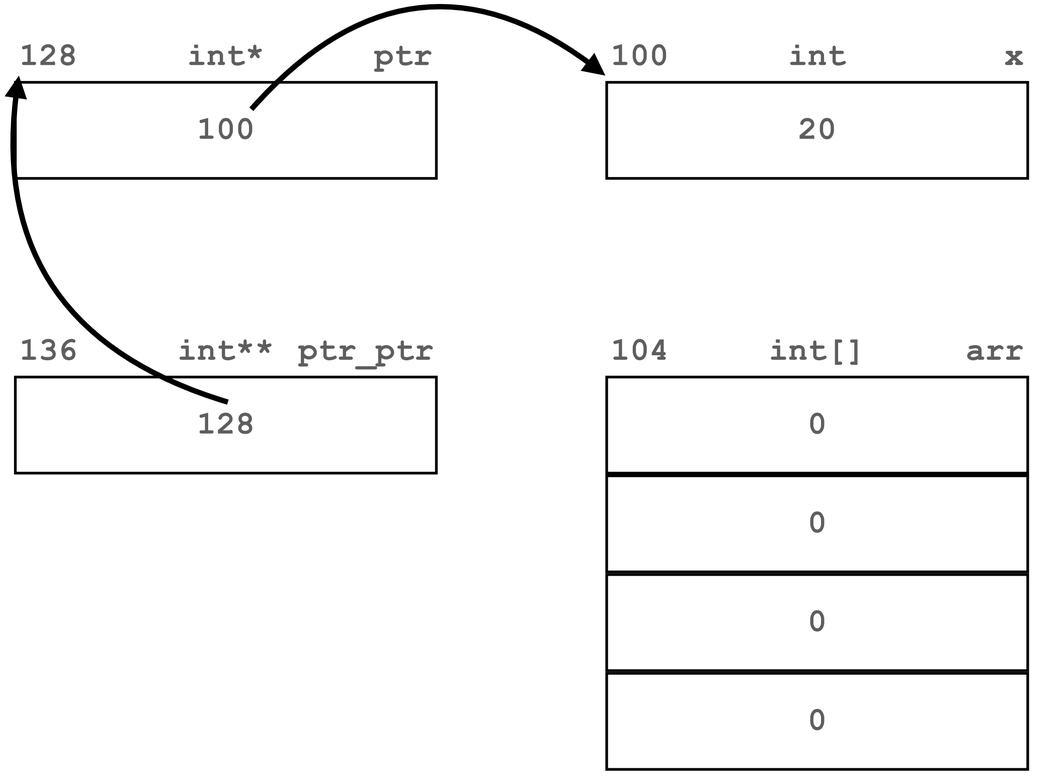
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
```



Pointers

Review

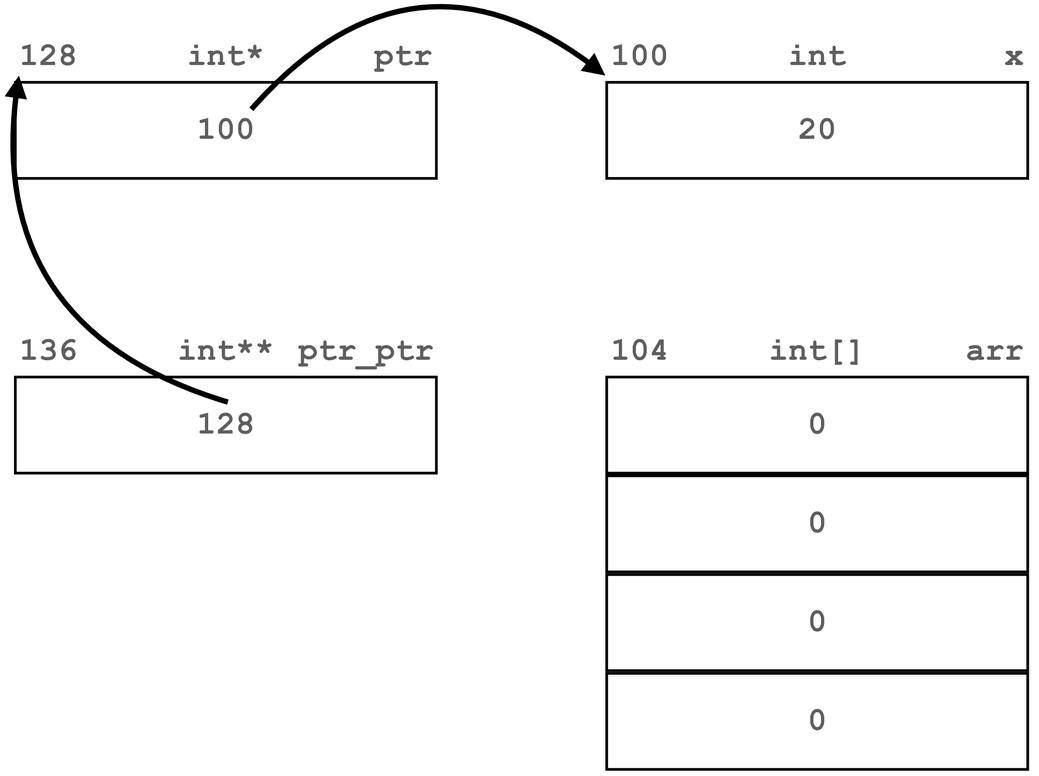
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);  <-- 100
```



Pointers

Review

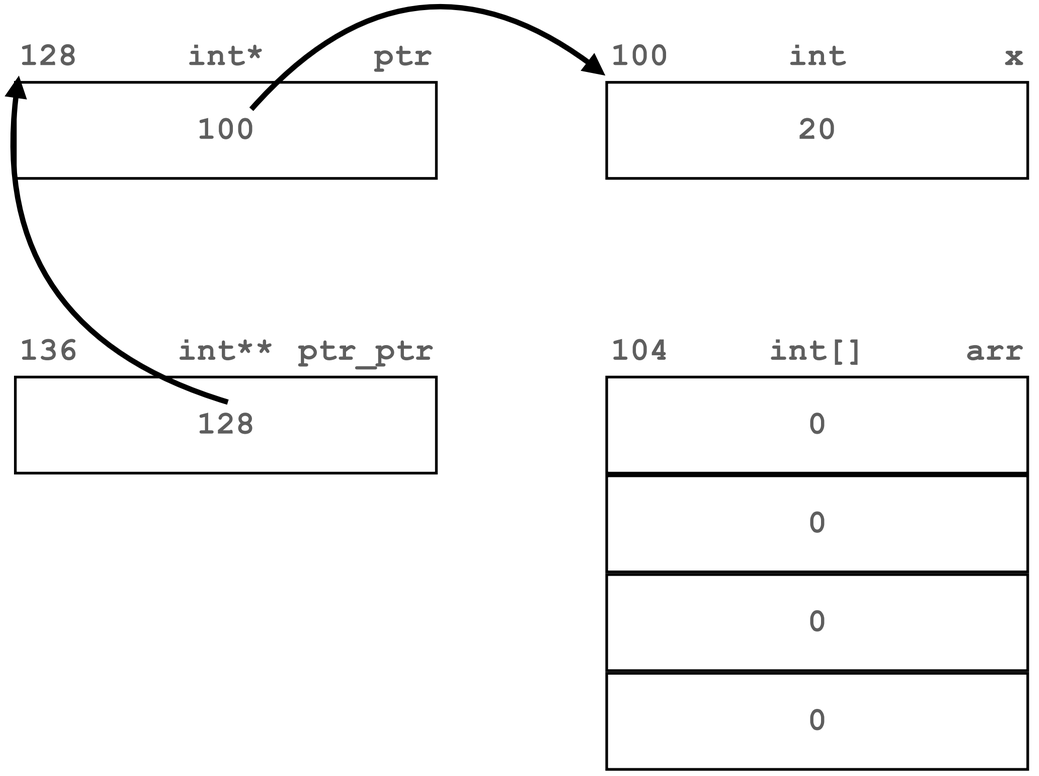
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);
```



Pointers

Review

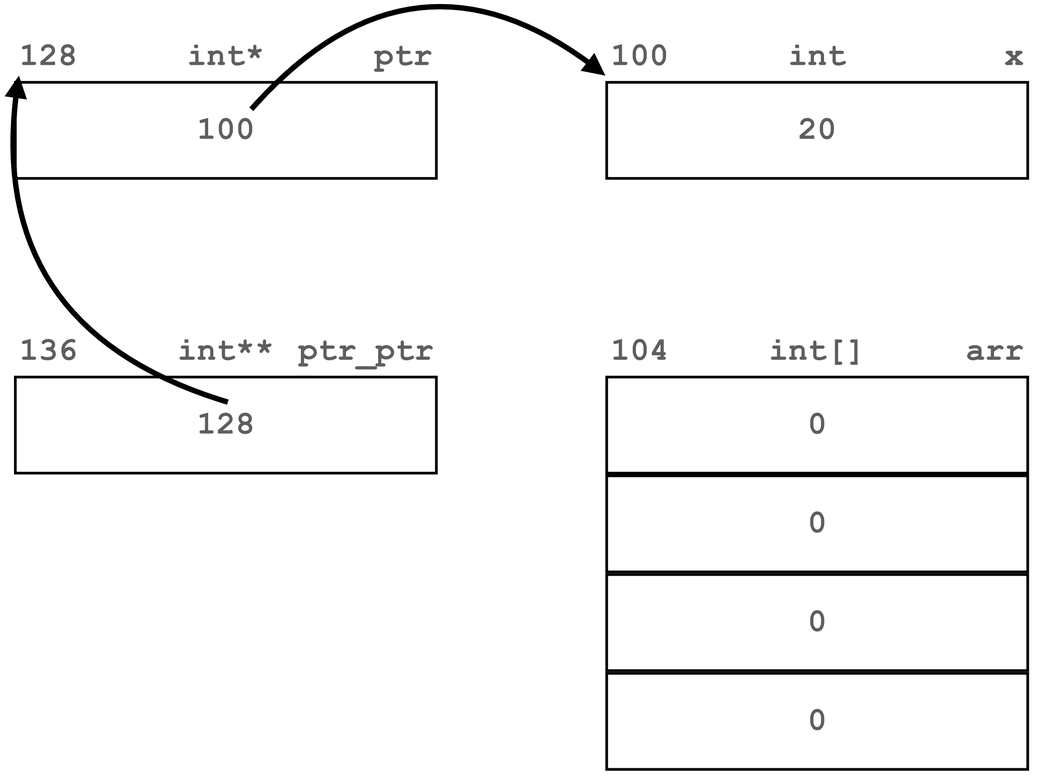
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr); <-- 20
```



Pointers

Review

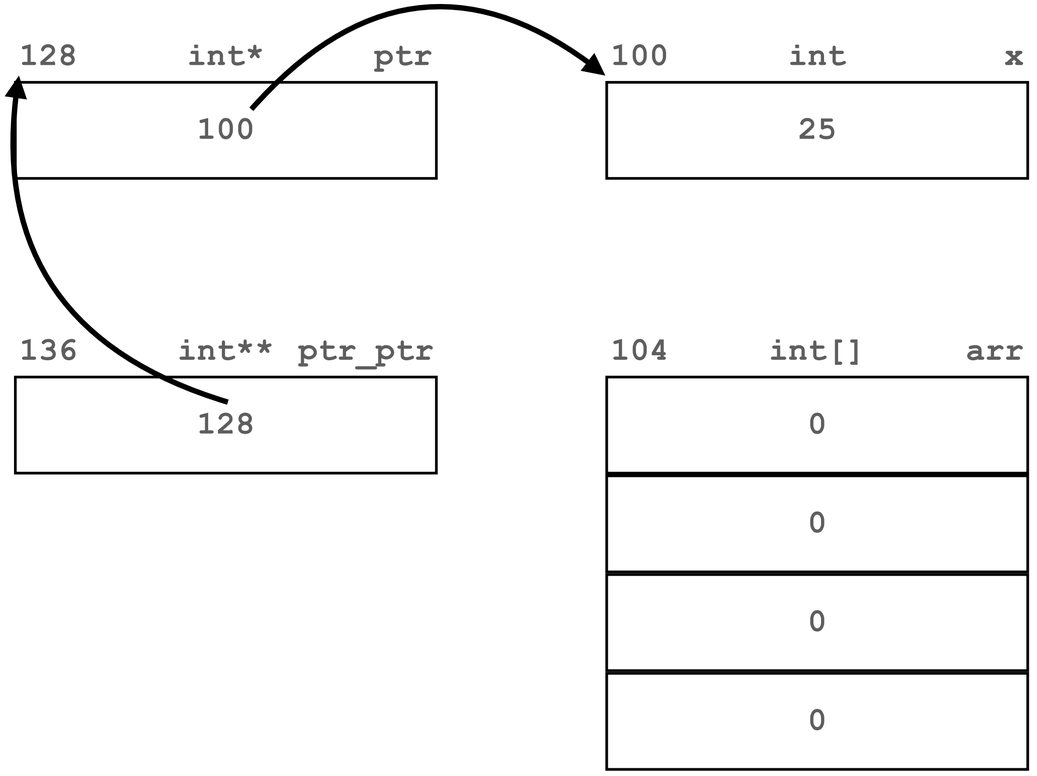
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
```



Pointers

Review

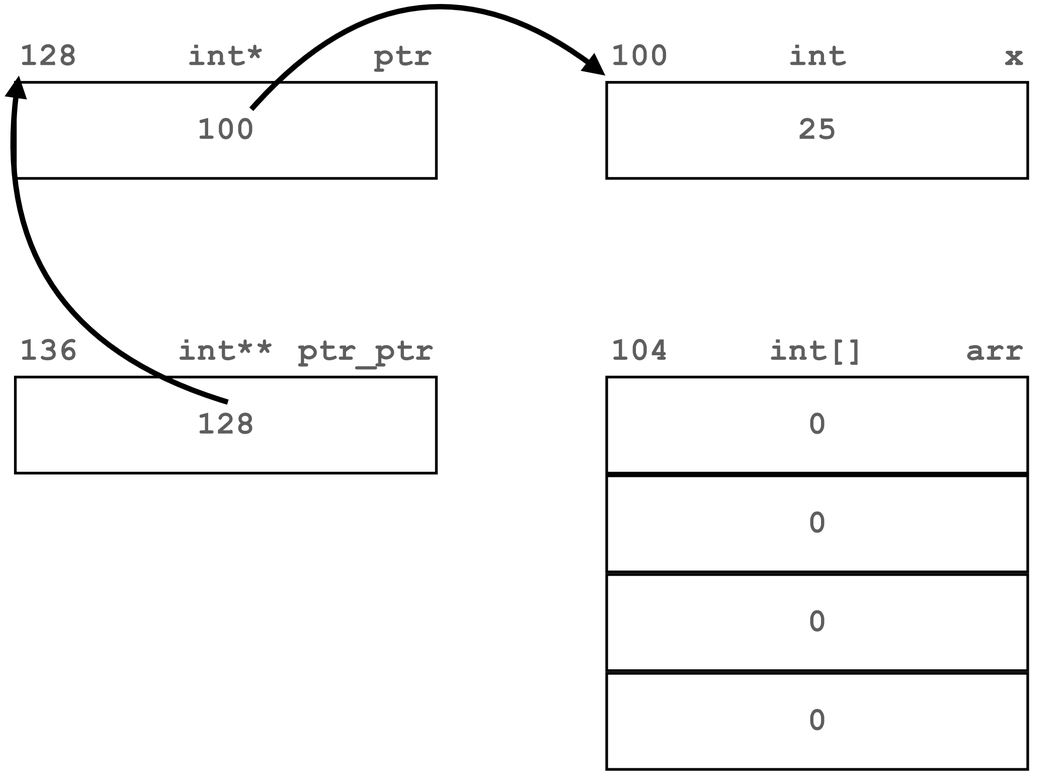
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
```



Pointers

Review

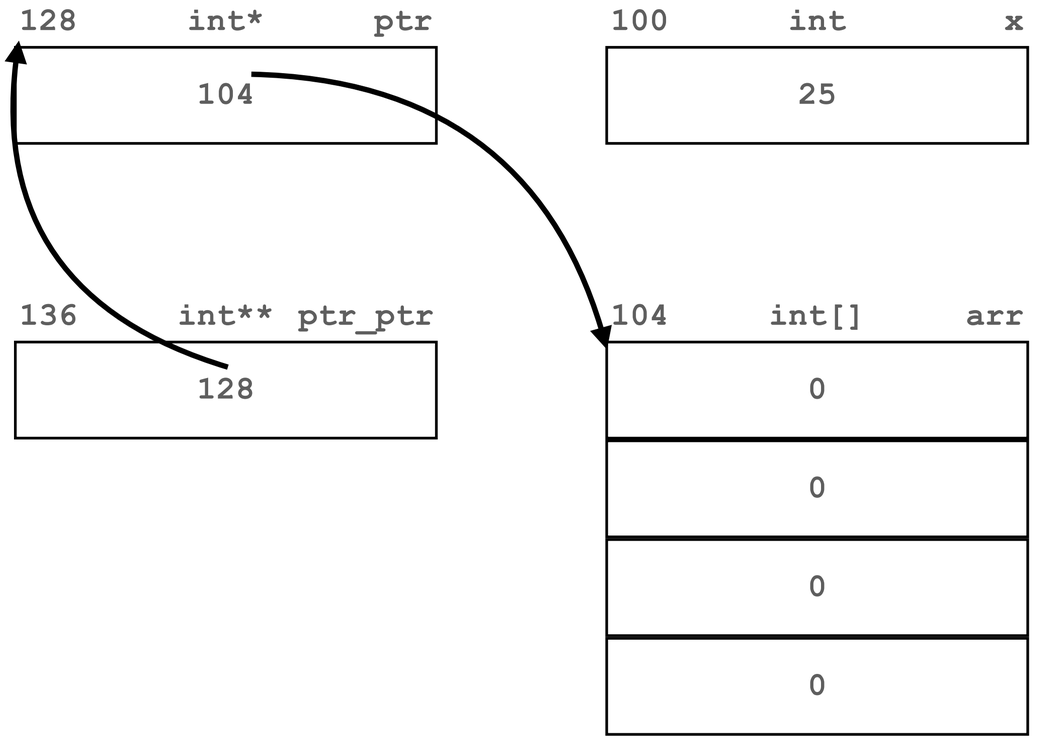
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
```



Pointers

Review

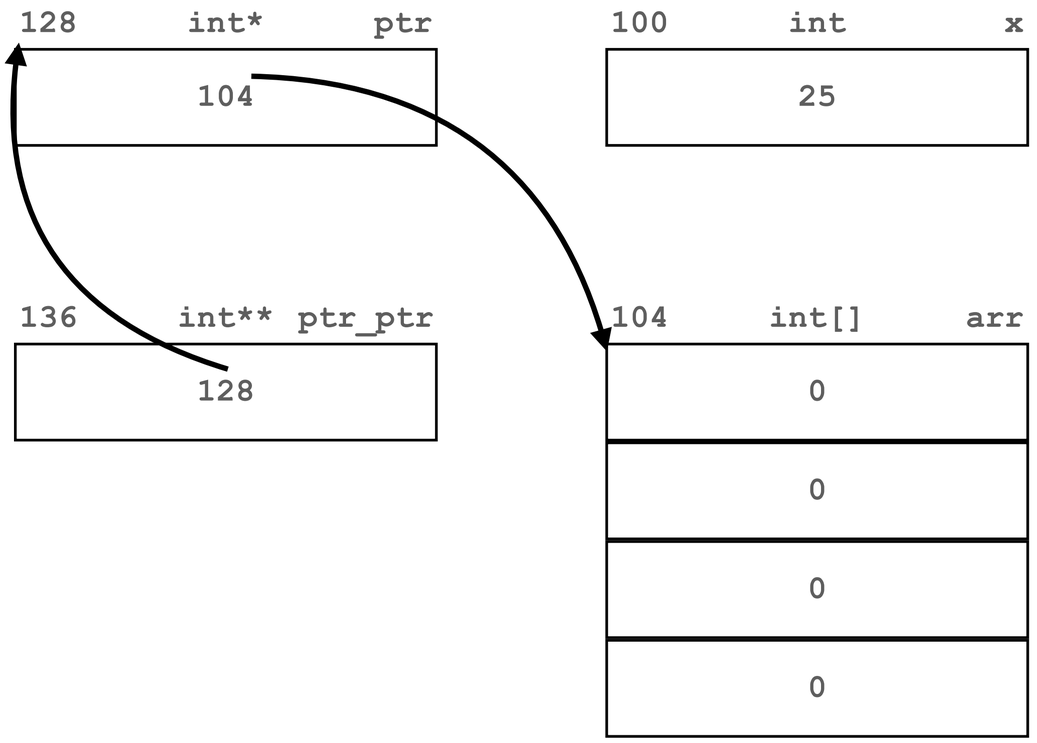
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
```



Pointers

Review

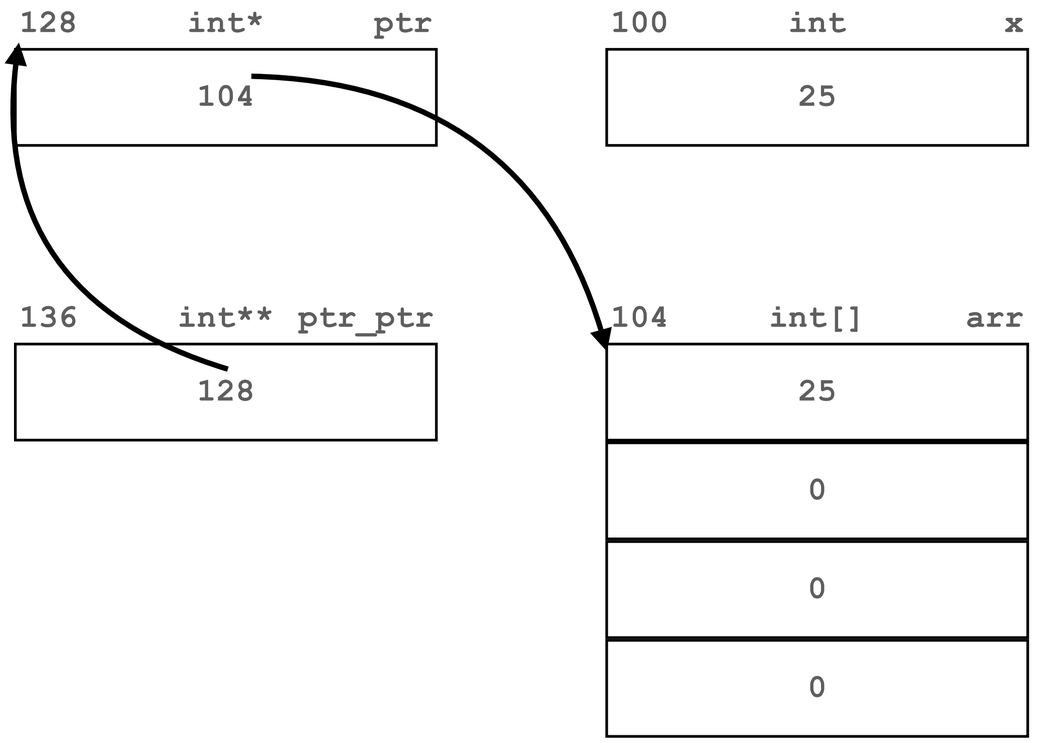
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;
```



Pointers

Review

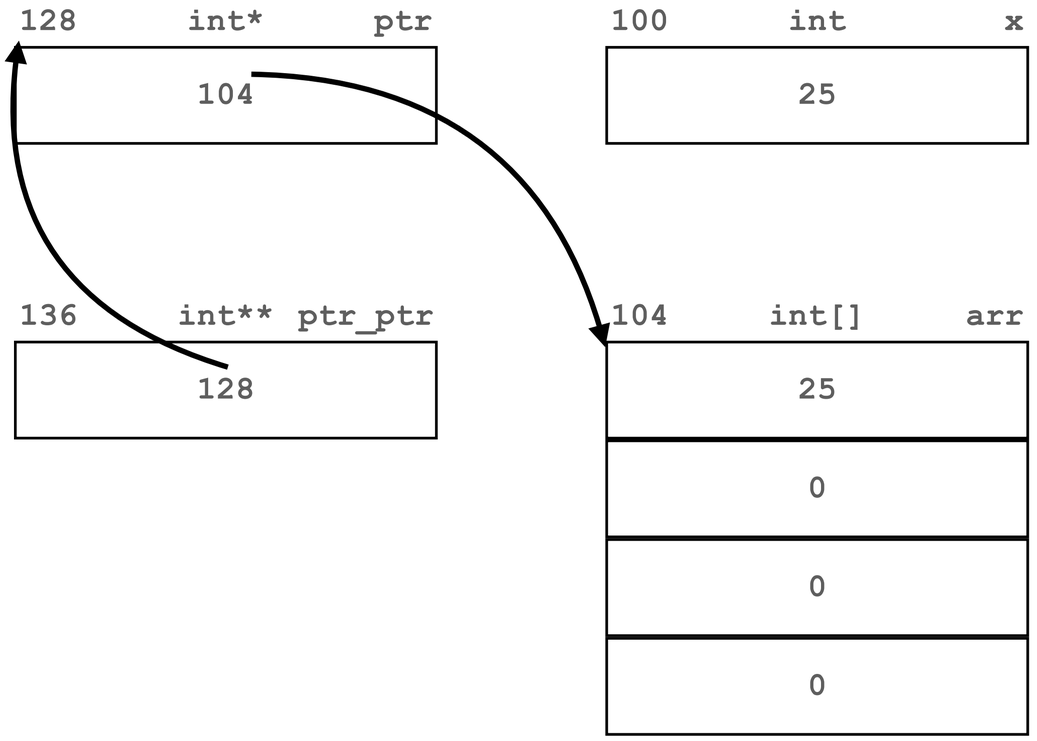
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;
```



Pointers

Review

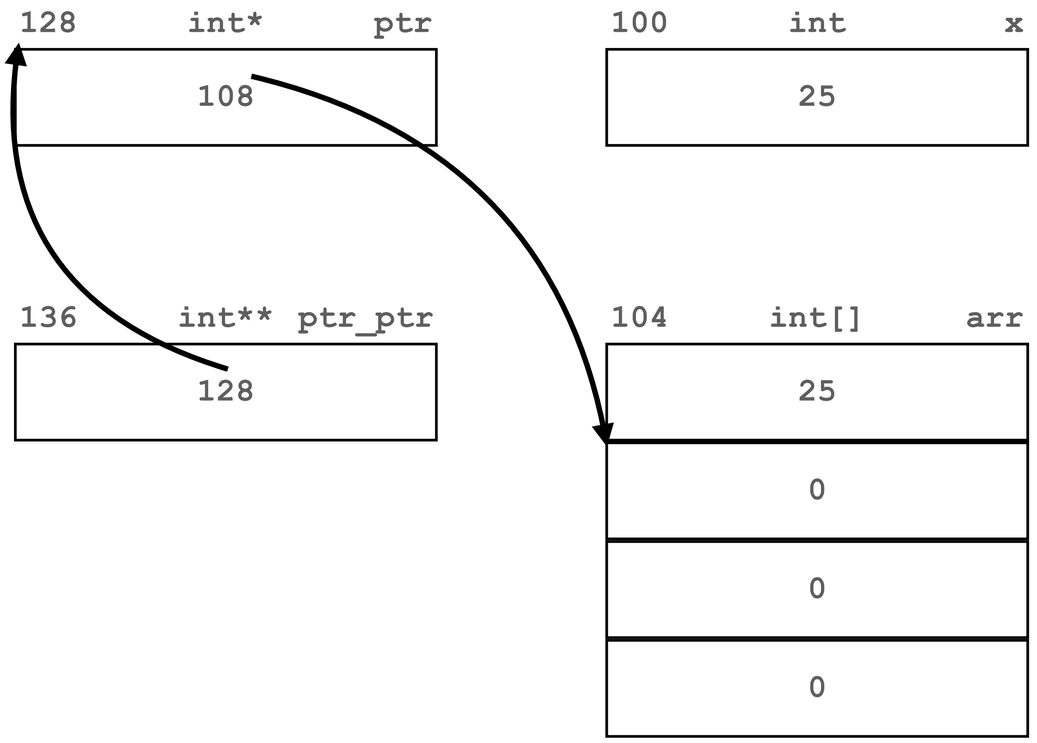
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
```



Pointers

Review

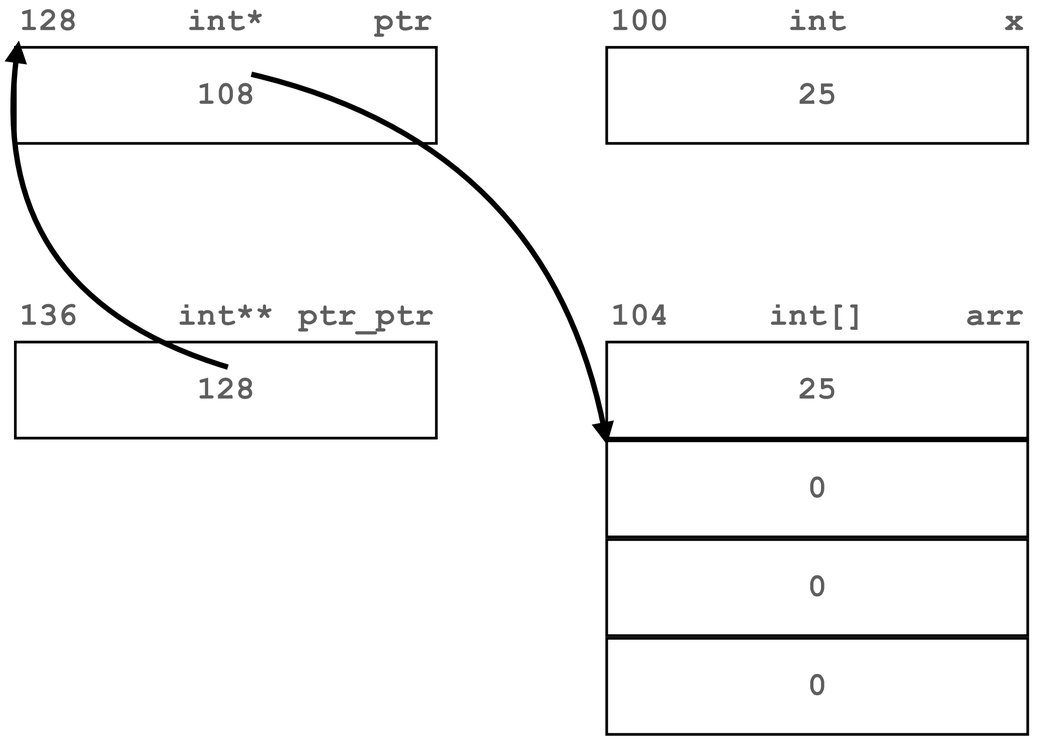
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
```



Pointers

Review

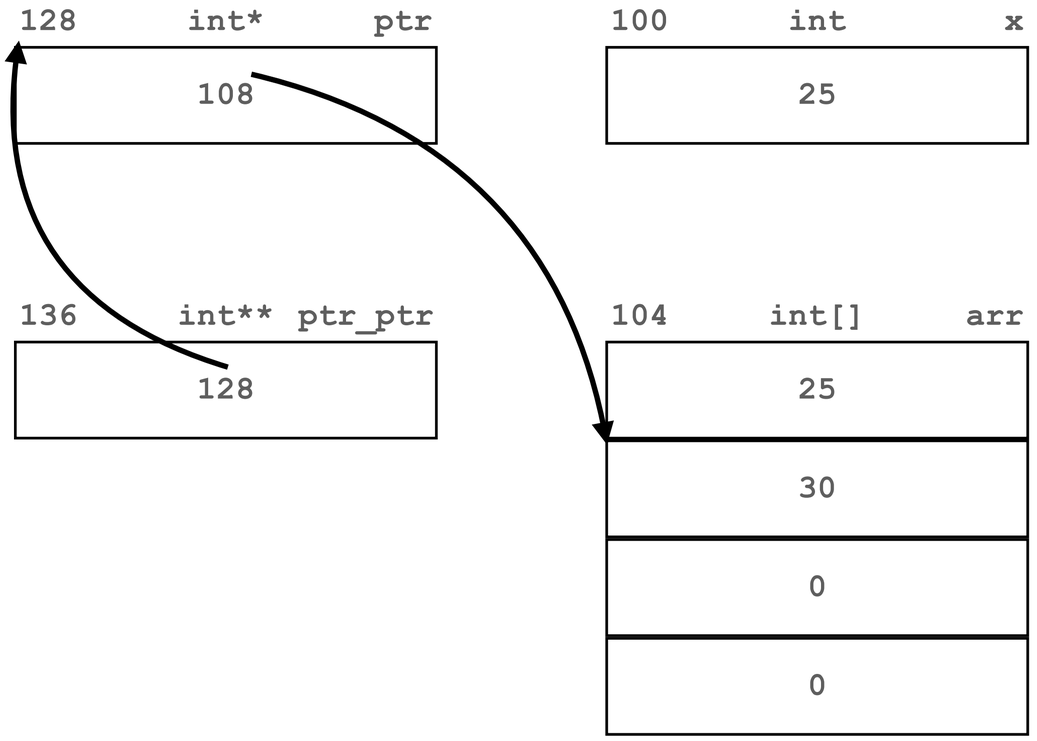
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
```



Pointers

Review

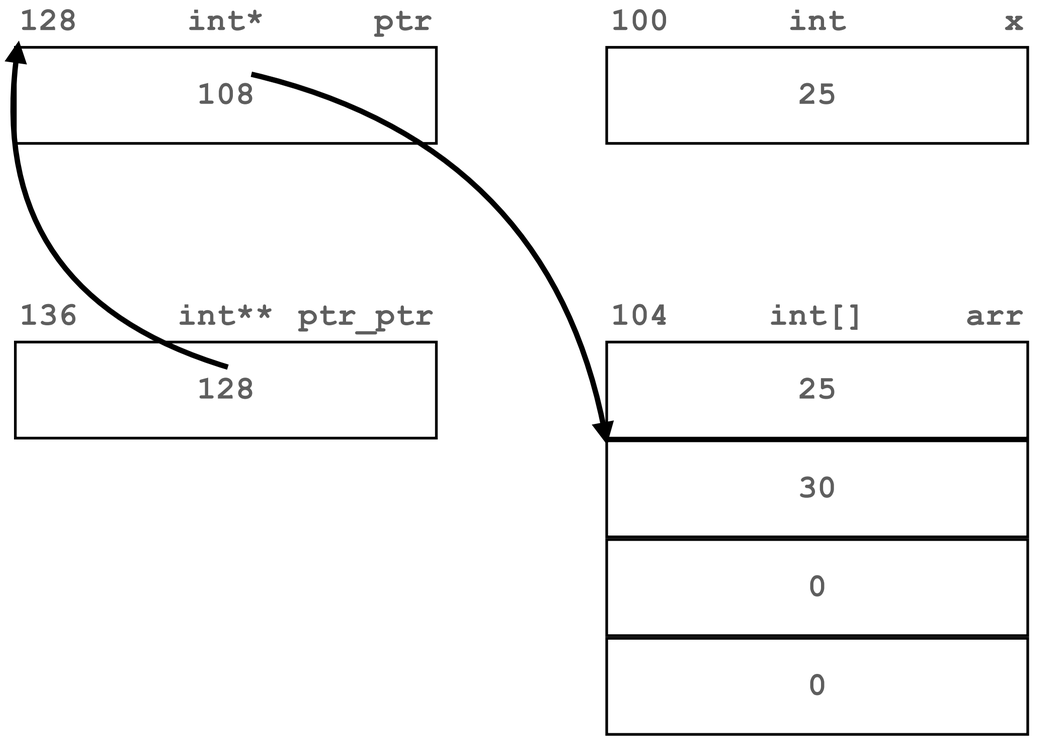
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
```



Pointers

Review

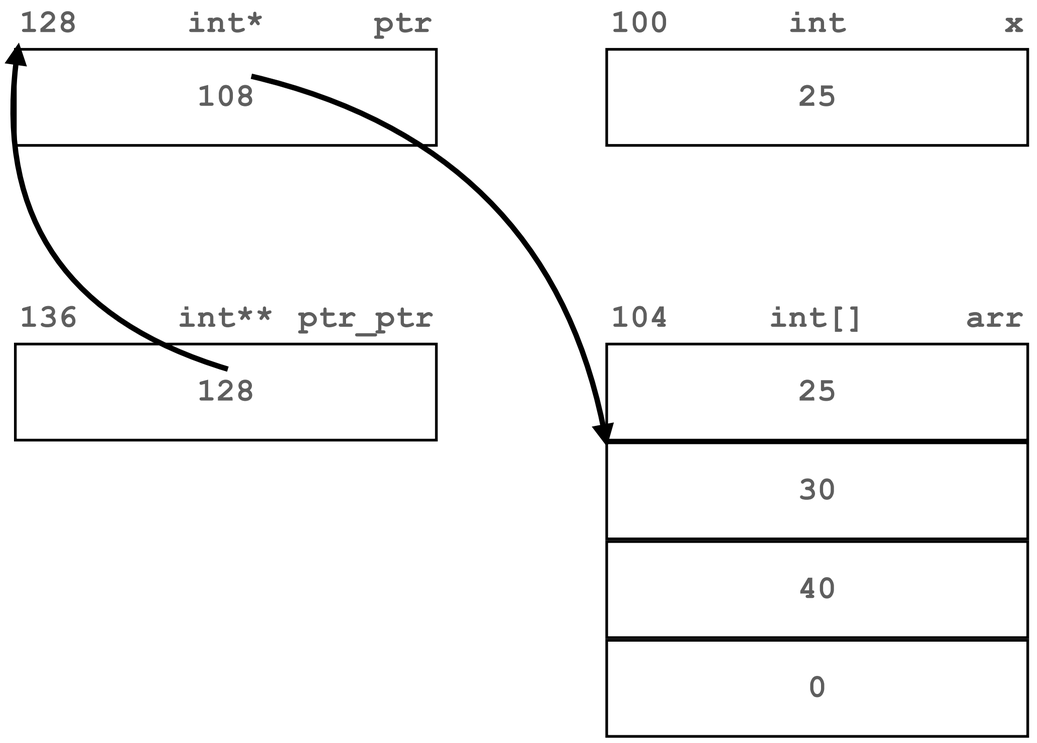
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
ptr[1] = 40;
```



Pointers

Review

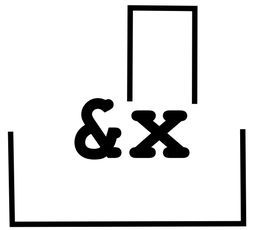
```
int x = 20;
int arr[4] = { 0 };
int *ptr;
int **ptr_ptr;

ptr = &x;
printf("%d\n", *ptr);

ptr_ptr = &ptr;
printf("%p\n", *ptr_ptr);
printf("%d\n", **ptr_ptr);

*ptr = 25;
ptr = &arr[0];
*ptr = 25;

*ptr_ptr = &arr[1];
*ptr = 30;
ptr[1] = 40;
```



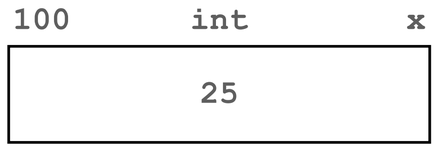
Pointers

Review

```
type : int
value: 25
```



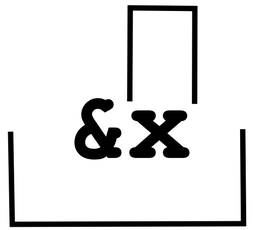

```
type : int *
value: 100
```



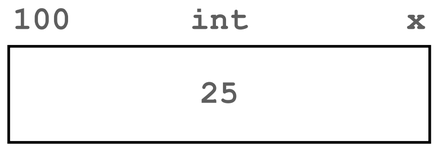
Pointers

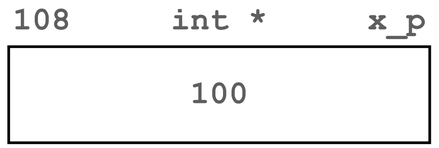
Review

```
type : int
value: 25
```




```
type : int *
value: 100
```

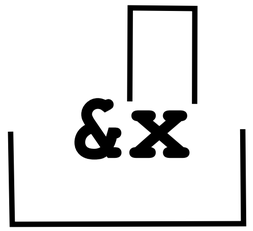




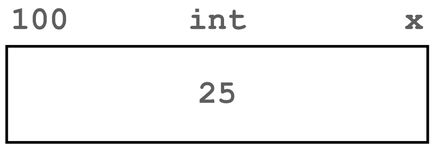
Pointers

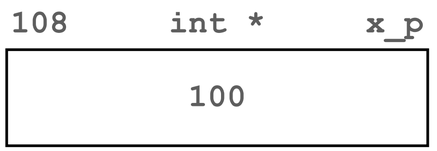
Review

```
type : int
value: 25
```

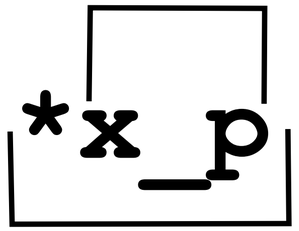



```
type : int *
value: 100
```






```
type : int *
value: 100
```

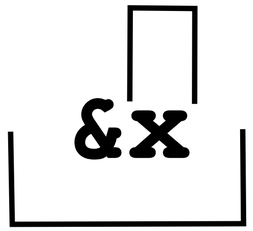



```
type : int
value: 25
```

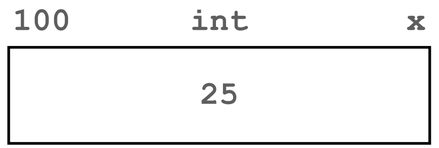
Pointers

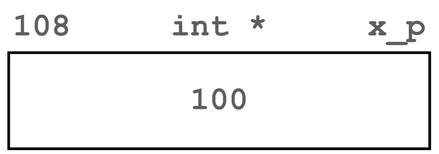
Review

```
type : int
value: 25
```

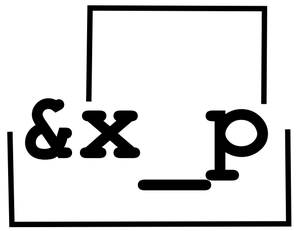



```
type : int *
value: 100
```



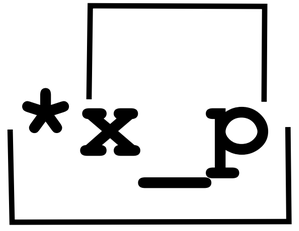



```
type : int *
value: 100
```




```
type : int **
value: 108
```

```
type : int *
value: 100
```

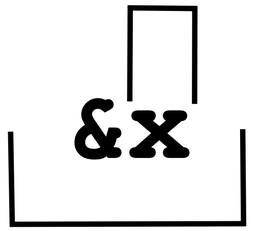



```
type : int
value: 25
```

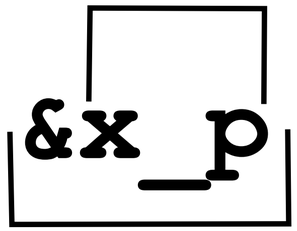
Pointers

Review

```
type : int
value: 25
```

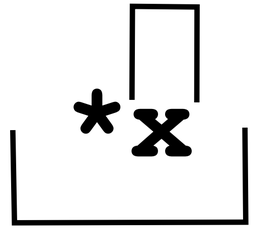



```
type : int *
value: 100
```

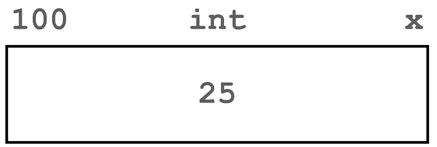


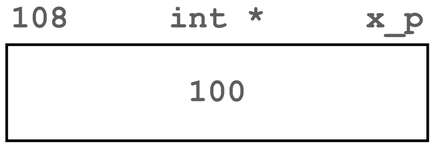

```
type : int **
value: 108
```

```
type : int
value: 25
```

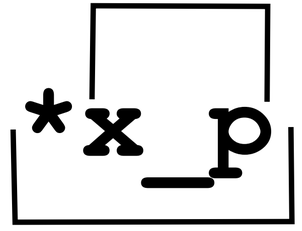



```
error
```






```
type : int *
value: 100
```

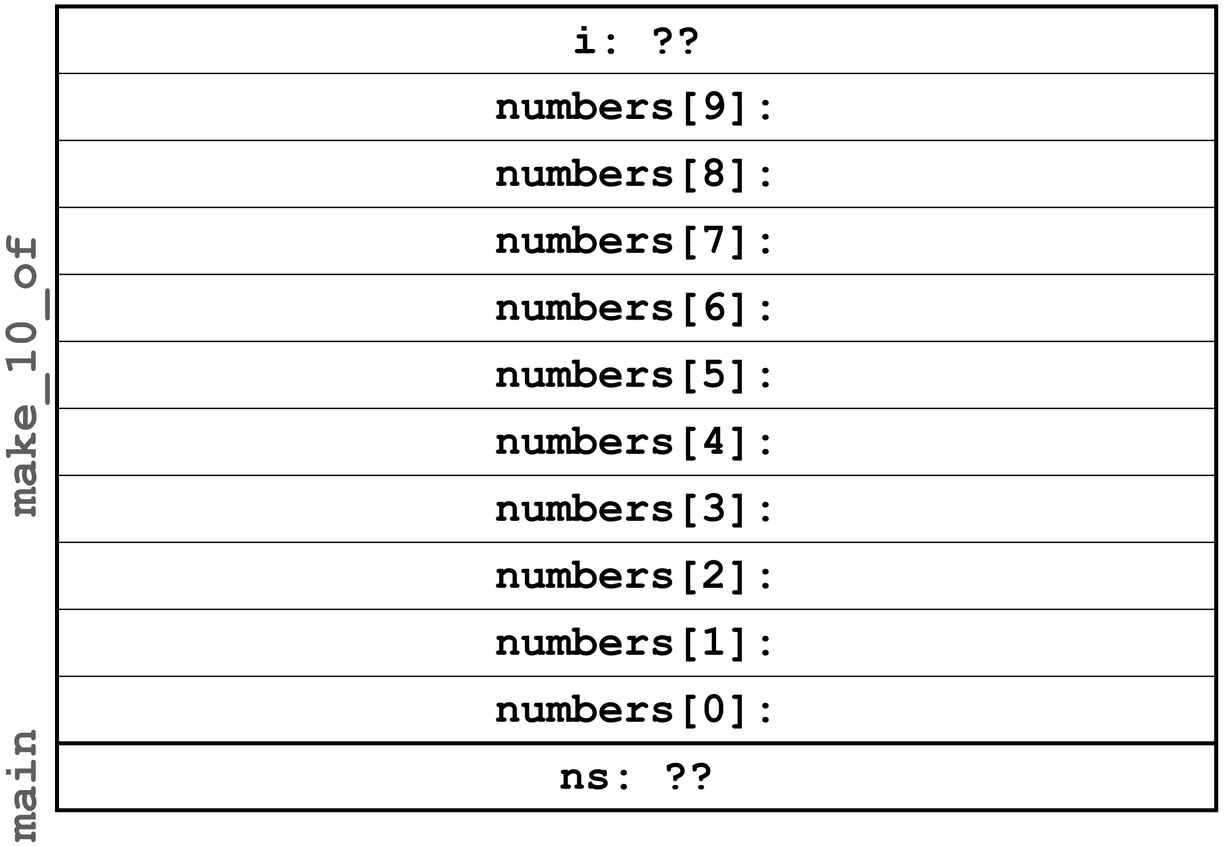



```
type : int
value: 25
```

Pointers

How about returning an array?

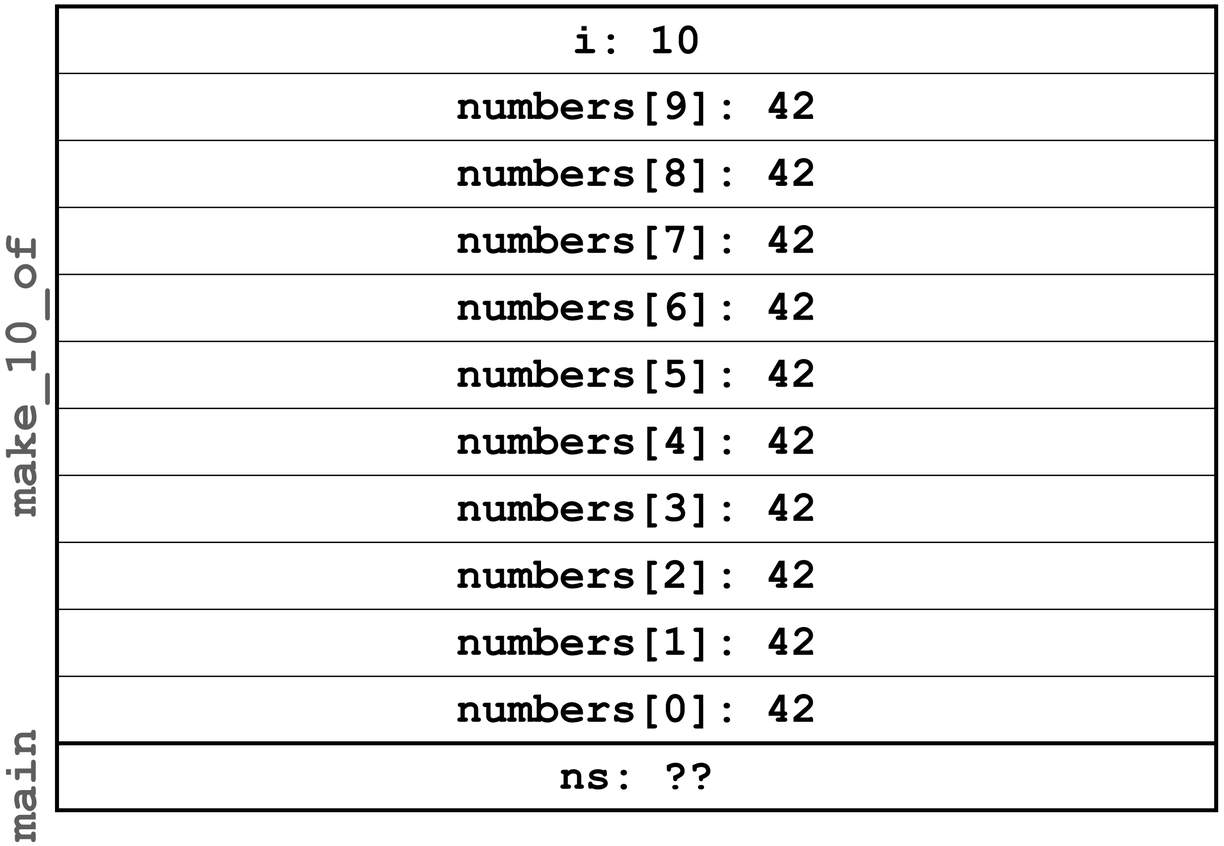
```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```



Pointers

How about returning an array?

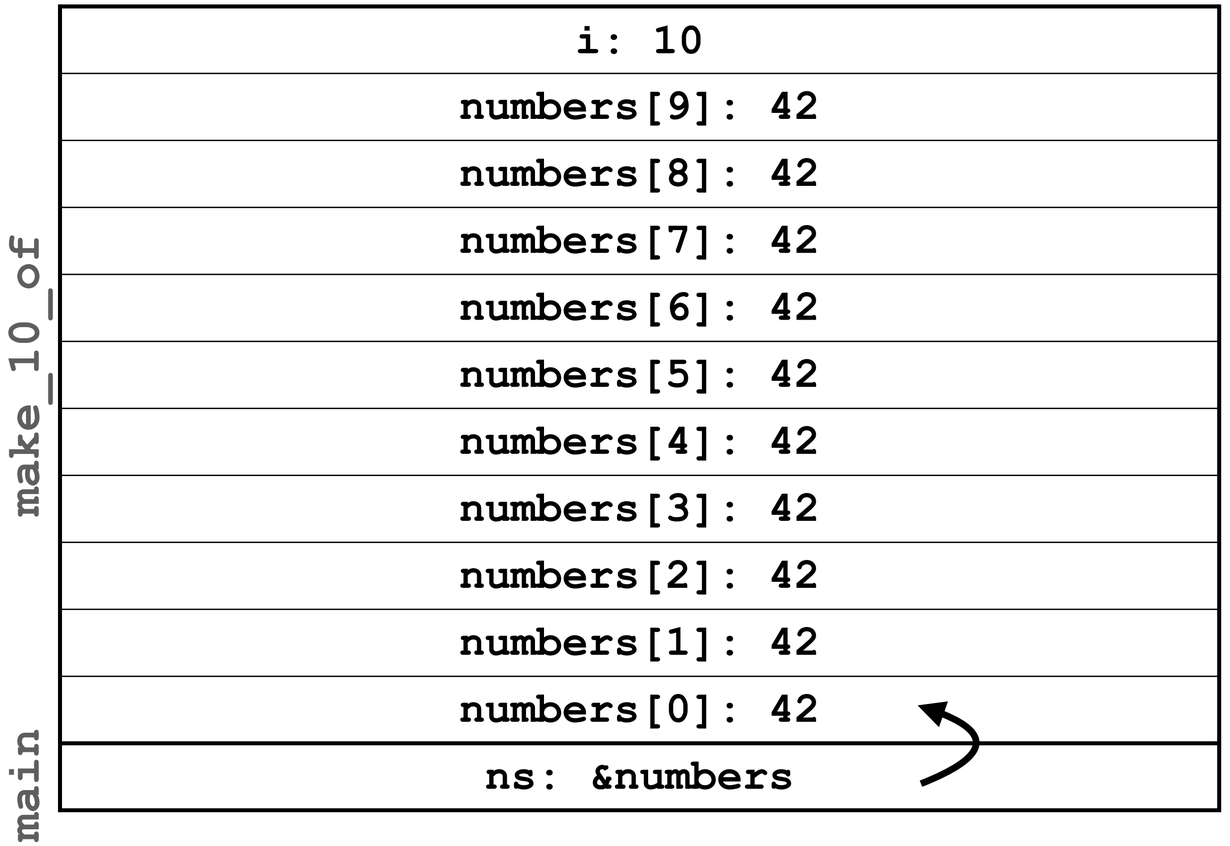
```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```



Pointers

How about returning an array?

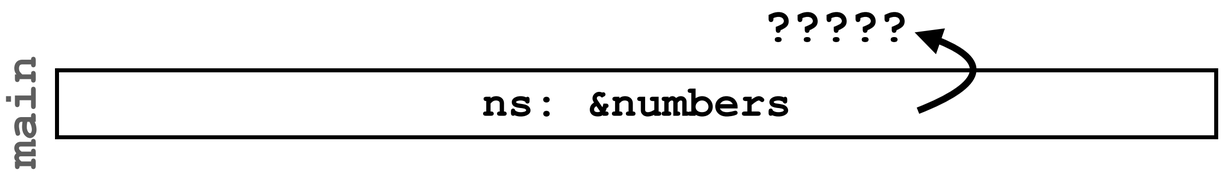
```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```



Pointers

How about returning an array?

```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```

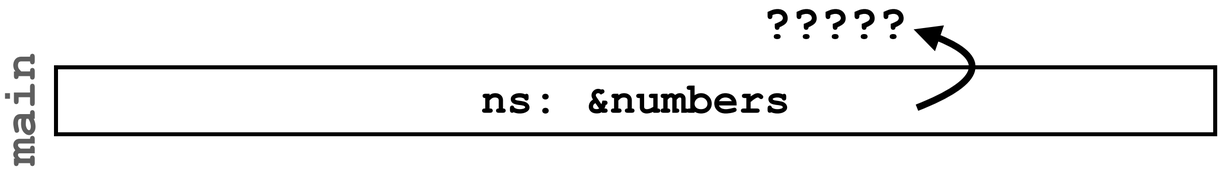


Pointers

How about returning an array?

```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```

- `numbers` is recycled after returning

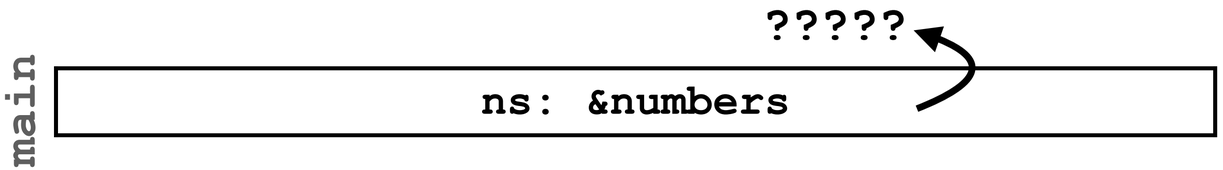


Pointers

How about returning an array?

```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```

- `numbers` is recycled after returning
- `ns` becomes a pointer to invalid/ non-existent/dead data.



Pointers

How about returning an array?

```
int *make_10_of(int n)
{
        int numbers[10];

        for (int i = 0; i < 10; i++) {
                numbers[i] = n;
        }

        return numbers;
}


int main()
{
        int *ns = make_10_of(42);
        ...
        return 0;
}
```

- `numbers` is recycled after returning
- `ns` becomes a pointer to invalid/ non-existent/dead data.
- We call `ns` a dangling pointer