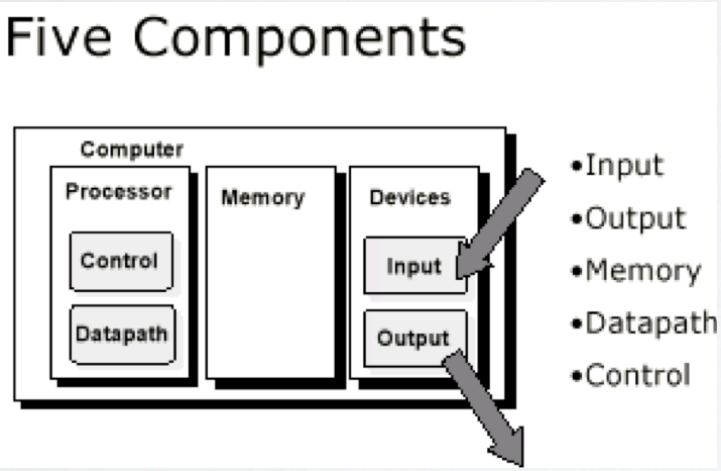
CAS CS 210 : COMPUTER SYSTEMS

Professor: Appavoo

CS:APP2e : Chapter 1

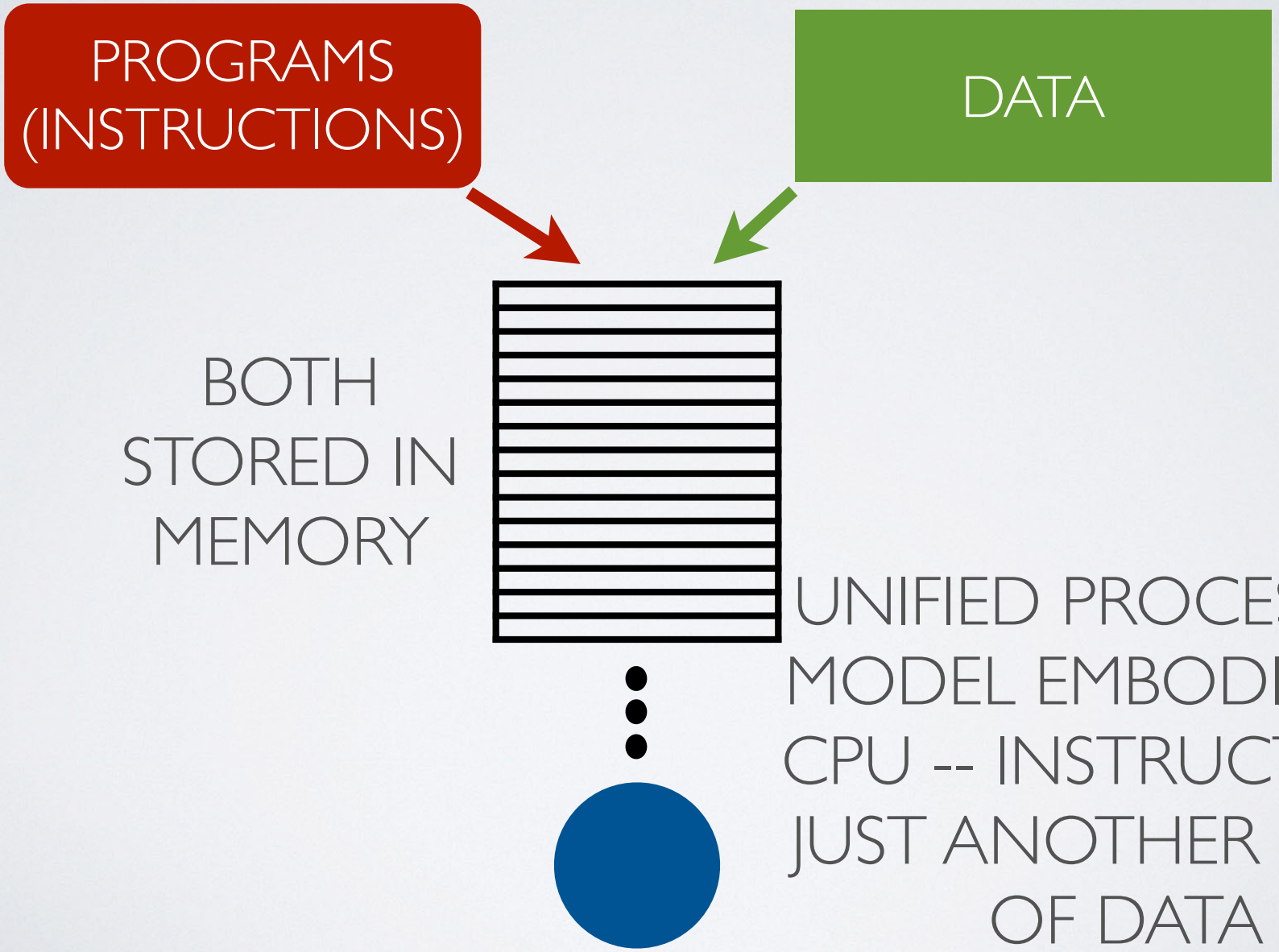
SYSTEMS PROGRAMMING

THE VON NEUMANN MODEL A DIFFERENT VIEW OF PROGRAMMING



- Control and coordinate the resources of a computer
- Place and move data in and between the components
- Common Model means similar approach to systems software

STORED PROGRAM CONCEPT



MEMORY IS FUNDAMENTAL

```
#include <stdio.h>
int i=100;
char sa[4] = "foo";

int main(void)
{
  printf("size of i=%d location of i=%p value of i=%d\n",
       sizeof(i), &i, i);
  printf("size of sa=%d location of sa=%p value of sa=%s\n",
       sizeof(sa), &sa, sa);
  printf("size of main=%d location of main=%p\n",
       sizeof(main), &main);
  return 0;
}
```

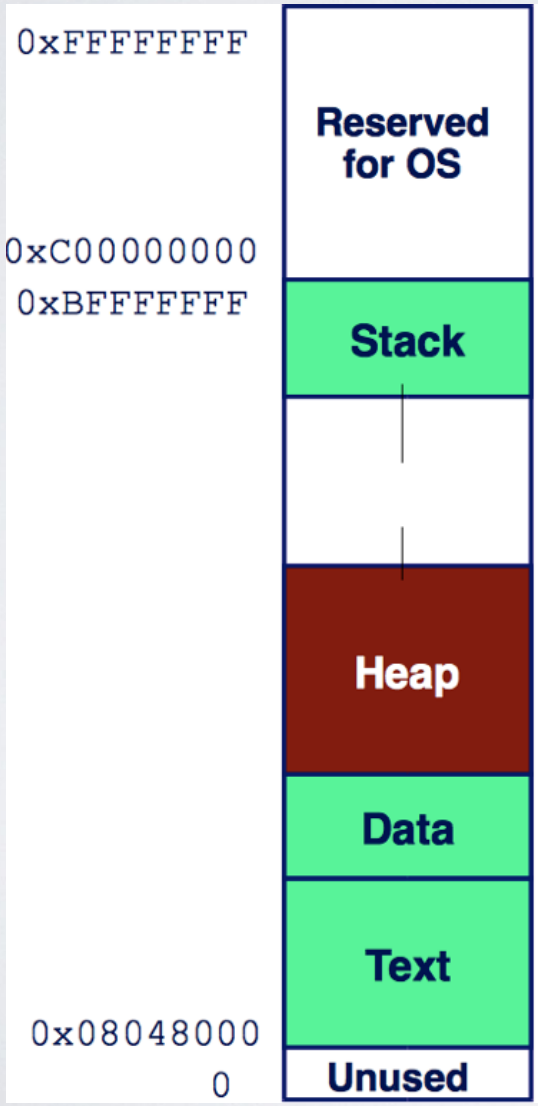
MEMORY — ARRAY OF BYTES

address	value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Software has location and size — your code has geometry in memory

Programming as painting — creating structures in memory

COMPILER AND OS DIVIDE MEMORY AND MAP YOUR PROGRAM



Linux Memory Layout

Stack

- Runtime user stack

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new` at run time

Data

- Statically allocated data
- Data declared in code

Text

- Executable machine instructions
- Read-only

SYMBOL TABLE — THE MAP

```
#include <stdio.h>

int i=100;
char sa[4] = "foo";

int main(void)
{
  printf("size of i=%d location of i=%p value of i=%d\n",
       sizeof(i), &i, i);
  printf("size of sa=%d location of sa=%p value of sa=%s\n",
        sizeof(sa), &sa, sa);
  printf("size of main=%d location of main=%p\n",
       sizeof(main), &main);
  return 0;
}
```

```
nm -S -g -l memory
```

SIZES

```
char c=100;
short s=100;
int i=100;
long long l=100;
int main(void)
{
  printf("%d %d %d %d\n", sizeof(c), sizeof(s), sizeof(i),sizeof(l));
}
```

LOCATIONS (ADDRESSES)

```
char c=100;          char * cptr = &c;
short s=100;         short * sptr = &s;
int i=100;           int * iptr= &i;
long long l=100;     long long *lptr = &l;
int main(void)
{
  printf("%p %p %p %p", cptr, sptr, iptr, lptr);
}
```

DRAW!!!!! PLEASE! PLEASE! PLEASE!

'C' : DATA SIZES

Sizes (in bytes) of C numeric data types

C Declaration	32-bit	64-bit
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char *	4	8
float	4	4
double	8	8
void *	4	8

Pointers combine address and type to provide an **exact interpretation** for the values of bytes at a particular address.

T *p;

p is a pointer to an object of type T

eg. :

int *iptr;

char *cptr;

ANOTHER EXAMPLE

```
csa2:~/210inClass$ gcc -m32 -g mem.c -o mem2
csa2:~/210inClass$ ./mem2
sizeof(int)=4
gv=100 lv=101
&gv=0x8049708 &lv=0xfff7a12c
&main=0x80483c4 main=0x80483c4
csa2:~/210inClass$
```

```
 1 #include <stdio.h>
 2
 3 int gv;
 4
 5 int main(void)
 6 {
 7   int lv;
 8
 9   gv = 100;      // write memory
10   lv = gv + 1;   // read and write memory
11
12   printf("sizeof(int)=%d\n", sizeof(int));
13   printf("gv=%d lv=%d\n", gv, lv);
14   printf("&gv=%p &lv=%p\n", &gv, &lv);
15   printf("&main=%p main=%p\n", &main, main);
16
17   return 1;
18 }
```

assignment '=' is really copying/moving values between the parts of the computer as needed

```
3 int gv;
```

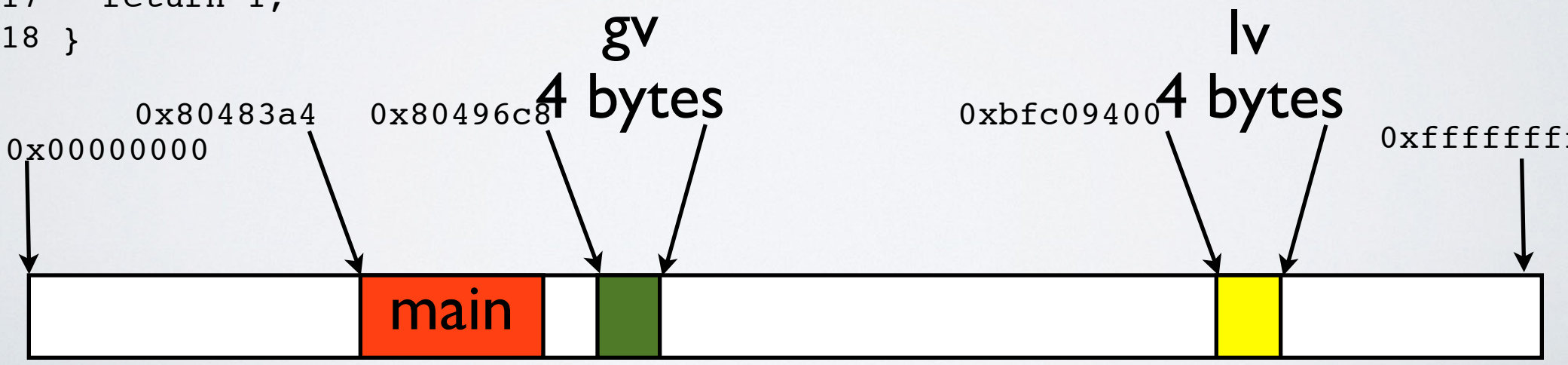

```
7   int lv;
```


```
9   gv = 100;
```


```
10   lv = gv + 1;
```

```
 1 #include <stdio.h>
 2
 3 int gv;
 4
 5 int main(void)
 6 {
 7   int lv;
 8
 9   gv = 100;      // write memory
10   lv = gv + 1;   // read and write memory
11
12   printf("sizeof(int)=%d\n", sizeof(int));
13   printf("gv=%d lv=%d\n", gv, lv);
14   printf("&gv=%p &lv=%p\n", &gv, &lv);
15   printf("&main=%p main=%p\n", &main, main);
16
17   return 1;
18 }
```

```
csa2:~/inClass$ gcc -g -m32 mem2.c -o mem2
csa2:~/inClass$ ./mem2
sizeof(int)=4
gv=100 lv=101
&gv=0x8049708 &lv=0xfff7a12c
&main=0x80483c4 main=0x80483c4
csa2:~/inClass$
```



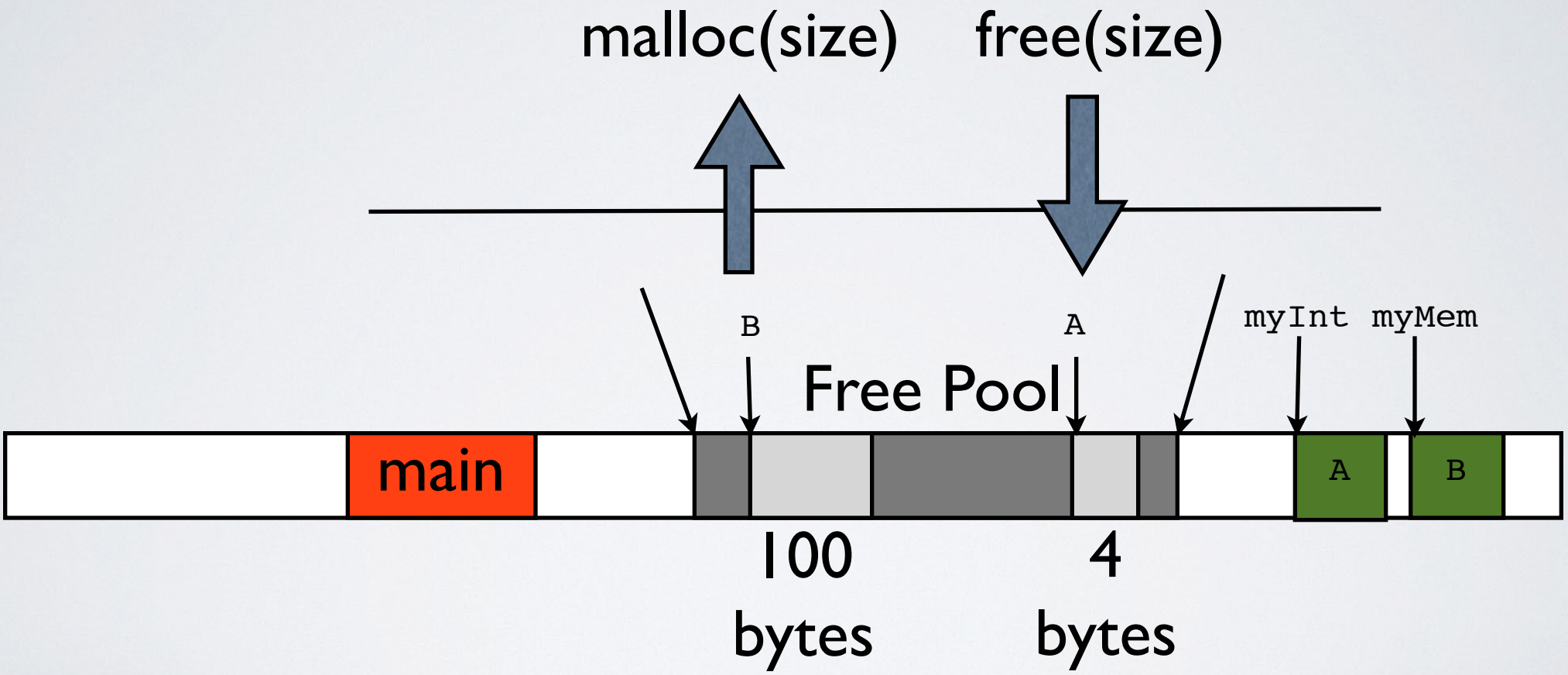
```c
#include <stdio.h>
#include <stdlib.h>

struct MyStruct {
  int v1;
  int v2;
};

int main(void)
{
  struct MyStruct s1;
  struct MyStruct *sptr;

  s1.v1 = 100;
  s1.v2 = s1.v1 + 1;

  sptr = malloc(sizeof(struct MyStruct));

  sptr->v1 = 200;
  sptr->v2 = sptr->v1 + 1;

  printf("sizeof(struct MyStruct)=%d\n", sizeof(struct MyStruct));
  printf("&s1=%p s1.v1=%d s1.v2=%d\n",
      &s1, s1.v1, s1.v2);
  printf("&sptr=%p sptr=%p sptr->v1=%d sptr->v2=%d\n",
      &sptr, sptr, sptr->v1, sptr->v2);
  return 1;
 }
```

```
csa2:~/inClass$ gcc -m32 struct.c -o struct
csa2:~/inClass$ ./struct
sizeof(struct MyStruct)=8
&s1=0xffb977c8 s1.v1=100 s1.v2=101
&sptr=0xffb977c4 sptr=0x8bf7008 sptr->v1=200
sptr->v2=201
csa2:~/inClass$
```

```
1 int *myInt = malloc(sizeof(int));
2 void *myMem = malloc(100);
```

myInt A

myMem B



```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 struct ListNode {
 5   int val;
 6   struct ListNode *next;
 7 };
 8
 9 struct ListNode *head = NULL;
10
11 int main(void)
12 {
13   struct ListNode *tmp;
14
15   tmp =  malloc(sizeof(struct ListNode));
16   tmp->val = 1;
17   tmp->next = NULL;
18   head = tmp;
19
20   tmp = malloc(sizeof(struct ListNode));
21   tmp->val = 3;
22   tmp->next = NULL;
23
24   tmp->next = head;
25   head = tmp;
26   return 1;
27 }
```

what's this doing?

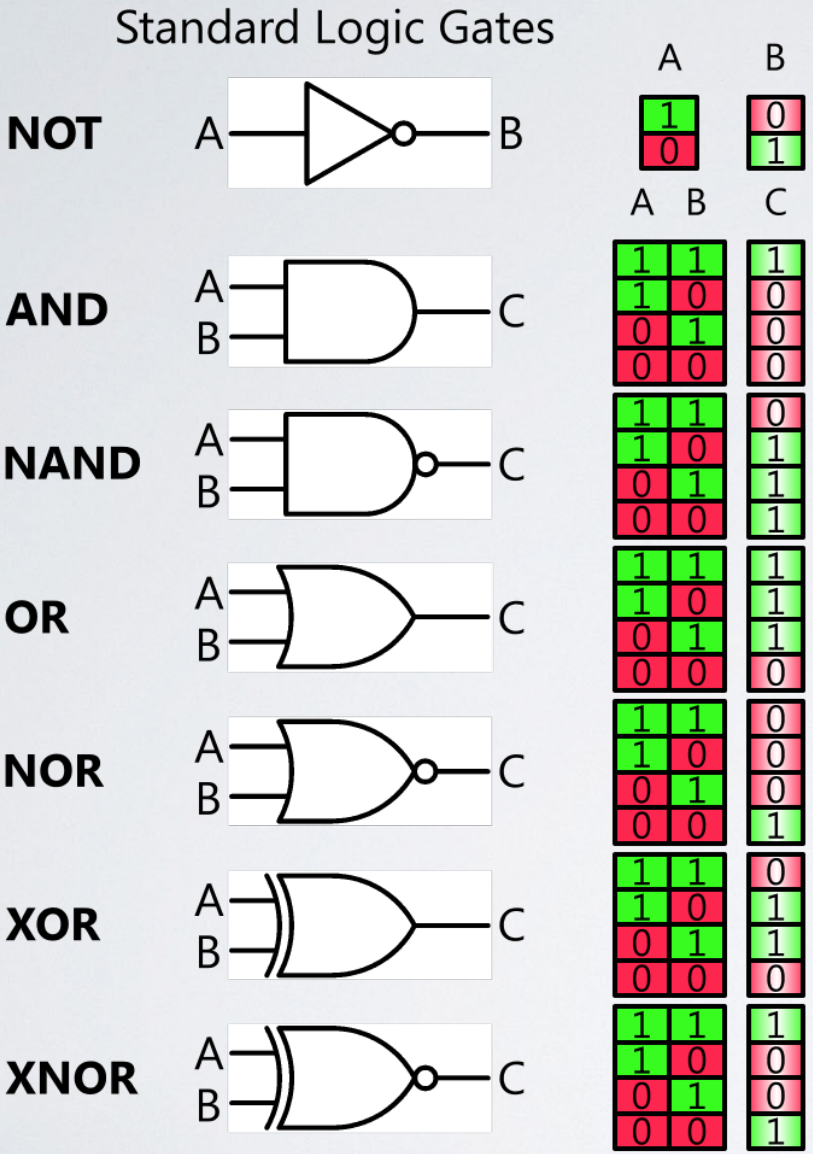
Lets use GDB

LOGIC GATES

BOOLEAN (DIGITAL) LOGIC

- Two voltage signals: high or low
- Think of signals as
  - (logically) true or false
  - 1 or 0
  - asserted or not
- 0 and 1 are complements of one another

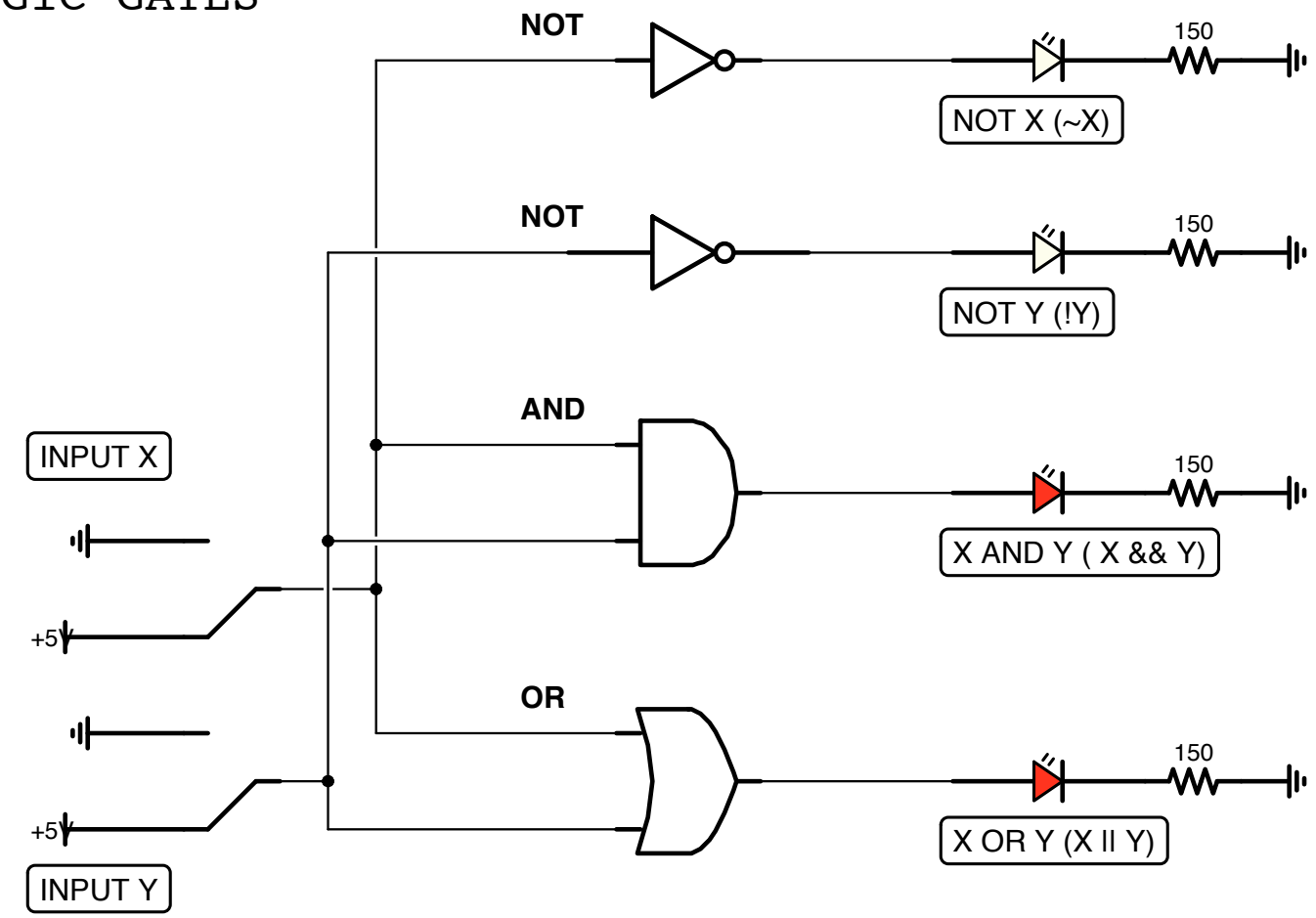
GATES

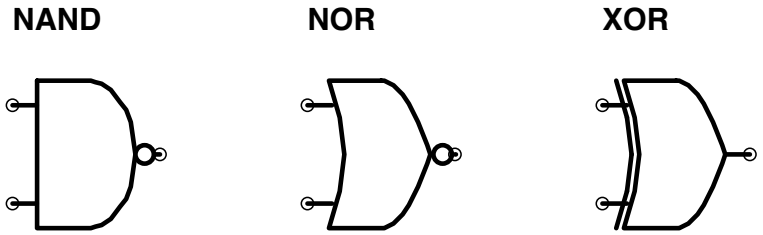


- EACH GATE IMPLEMENTS A TRUTH TABLE
- EACH LINE REPRESENTS EITHER AND INPUT COLUMN OR AN OUTPUT COLUMN
- STATEMENTS/FUNCTIONS CAN BE REPRESENTED AS CIRCUITS OF INTERCONNECTED GATES

- BASIC GATE DEMO

BASIC
LOGIC GATES





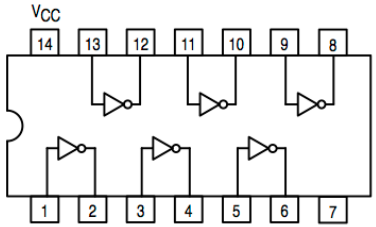
INTEGRATED CIRCUITS

http://www.skot9000.com/ttl/

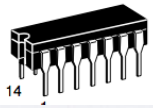
MOTOROLA

HEX INVERTER

SN54/74LS04



HEX INVERTER
LOW POWER SCHOTTKY

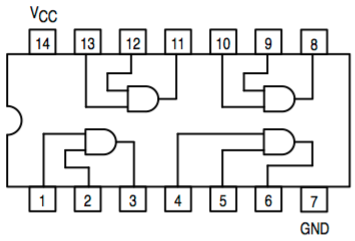


J SUFFIX
CERAMIC
CASE 632-08

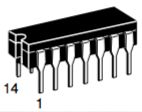
MOTOROLA

QUAD 2-INPUT AND GATE

SN54/74LS08



QUAD 2-INPUT AND GATE
LOW POWER SCHOTTKY

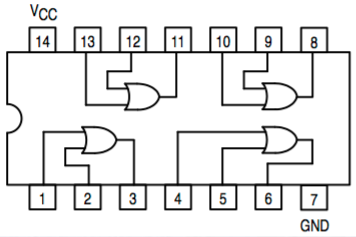


J SUFFIX
CERAMIC
CASE 632-08

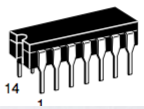
MOTOROLA

QUAD 2-INPUT OR GATE

SN54/74LS32



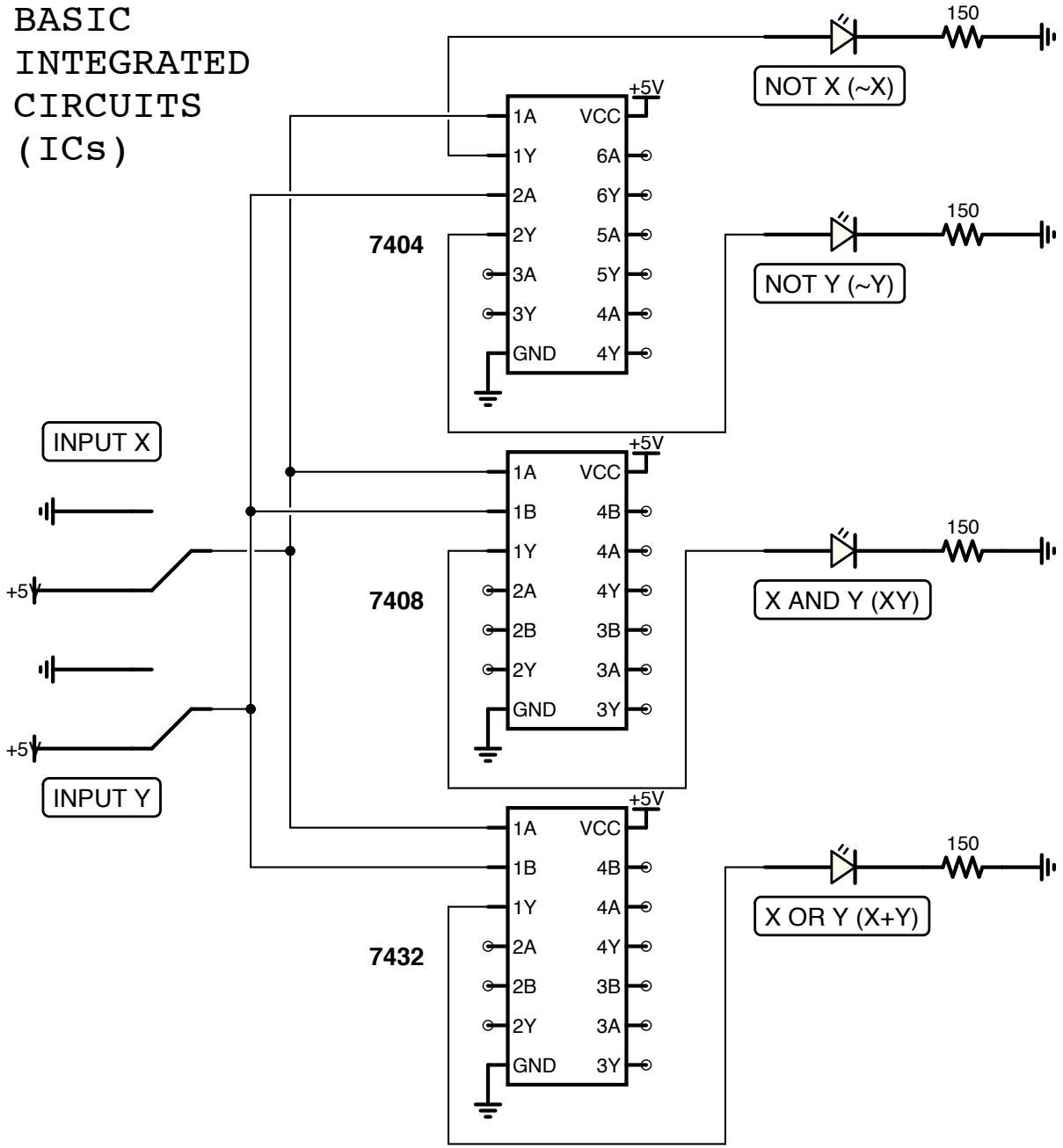
QUAD 2-INPUT OR GATE
LOW POWER SCHOTTKY



J SUFFIX
CERAMIC
CASE 632-08

- BASIC IC DEMO

BASIC INTEGRATED CIRCUITS (ICs)



The Same Basic Logic Gate Network from a few a couple of slides back implemented with actual ICs

2 KINDS OF NETWORKS

- Logical functions without memory are called **combinational**
  - Output depends only on current input
- Logical functions with memory are called **sequential**
  - Output depends on both input and state (value stored in memory)

THE CORE OF HOW COMPUTERS ARE BUILT AND WHAT WE PROGRAM

LET'S BUILD

COMBINATIONAL

- An input line may be split to provide input to more than one gate
- input and output lines only through gates
- No doubling back (acyclic graph) output can from gate g cannot be used as an input for g, or any gate directly or indirectly leading to g.
- out off is an instantaneous function of inputs present — no time dependencies or prior inputs

TRUTH TABLES & LOGIC

- A combinational logic function can be completely specified by defining values of its outputs for each possible set of input values
- With **n** inputs, the truth table has $\mathbf{2^n}$ entries (or input combinations)

TRUTH TABLES & LOGIC

```
(a && b) || (!a && !b)
```

a	b	!a	!b	a && b	!a && !b	(a&&b) || (!a&&!b)
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

BOOLEAN EXPRESSION AS LOGIC GATE NETWORK

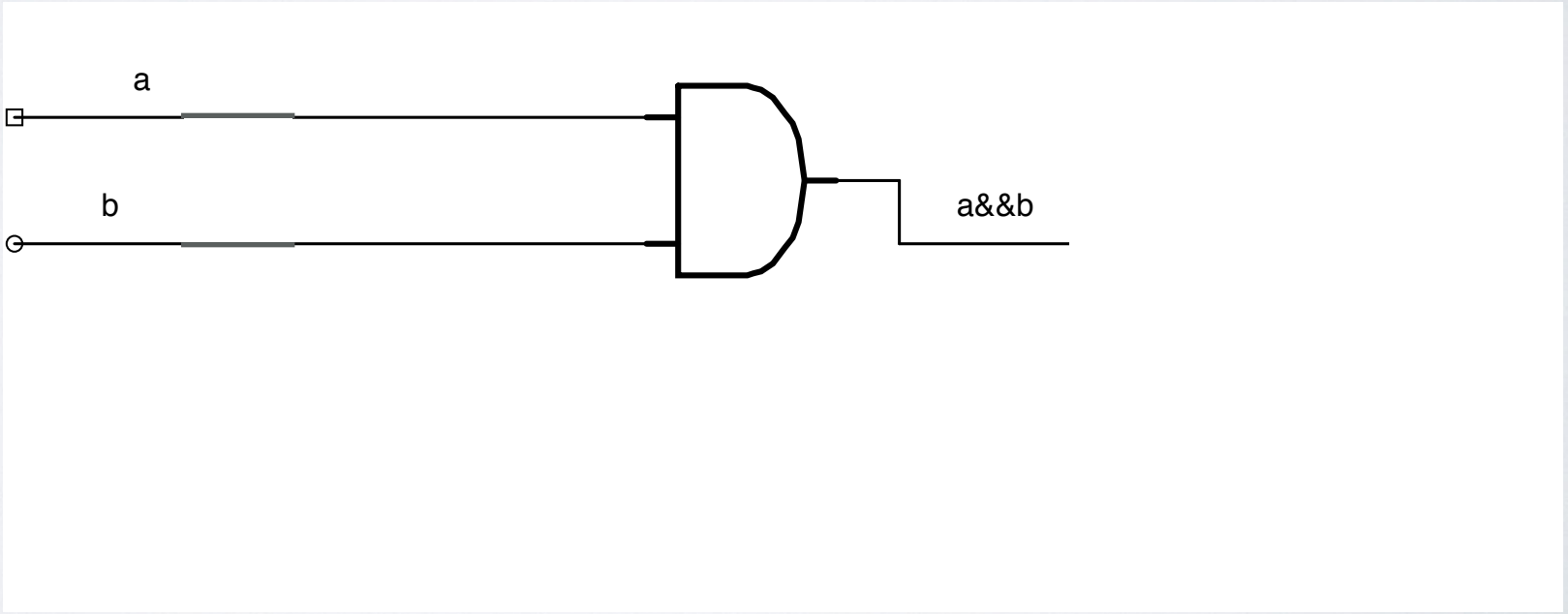
Operator	Description
&&	AND
||	OR
!	NOT

```
(a && b) || (!a && !b)
```

BOOLEAN EXPRESSION AS LOGIC GATE NETWORK

Operator	Description
&&	AND
||	OR
!	NOT

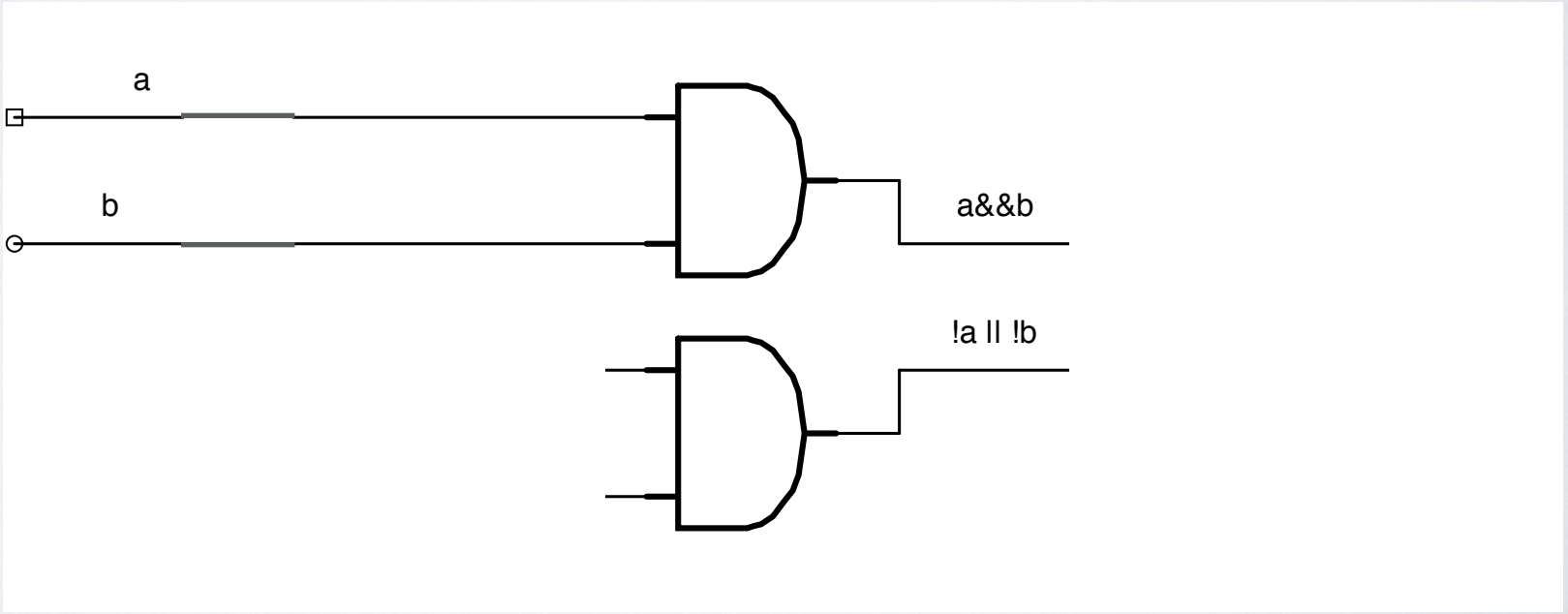
```
(a && b) || (!a && !b)
```



BOOLEAN EXPRESSION AS LOGIC GATE NETWORK

Operator	Description
&&	AND
||	OR
!	NOT

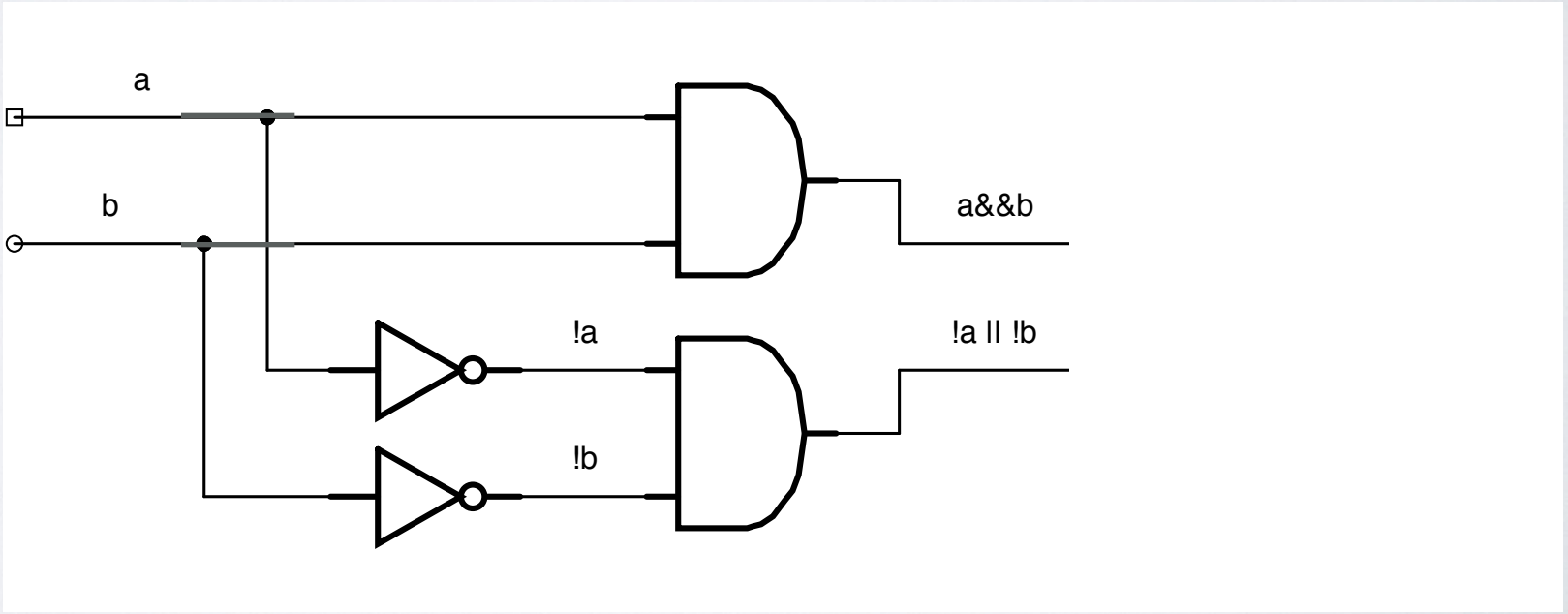
```
(a && b) || (!a && !b)
```



BOOLEAN EXPRESSION AS LOGIC GATE NETWORK

Operator	Description
&&	AND
||	OR
!	NOT

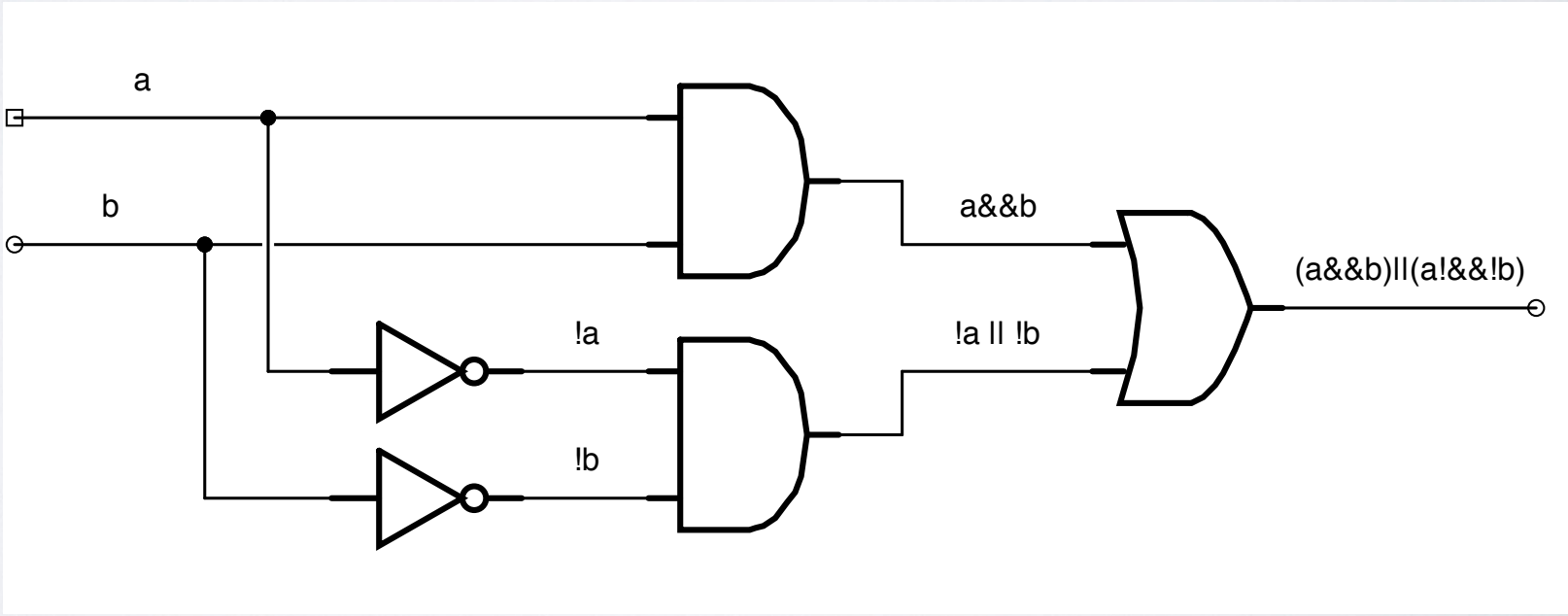
```
(a && b) || (!a && !b)
```



BOOLEAN EXPRESSION AS LOGIC GATE NETWORK

Operator	Description
&&	AND
||	OR
!	NOT

```
(a && b) || (!a && !b)
```



ADDITION

```
    1   1   0
+   0   1   1
-------------
```

ADDITION

```
    1   1   0
+   0   1   1
-----------
            1
```

ADDITION

```
      0
    1 1 0
+   0 1 1
-----------
      0 1
```

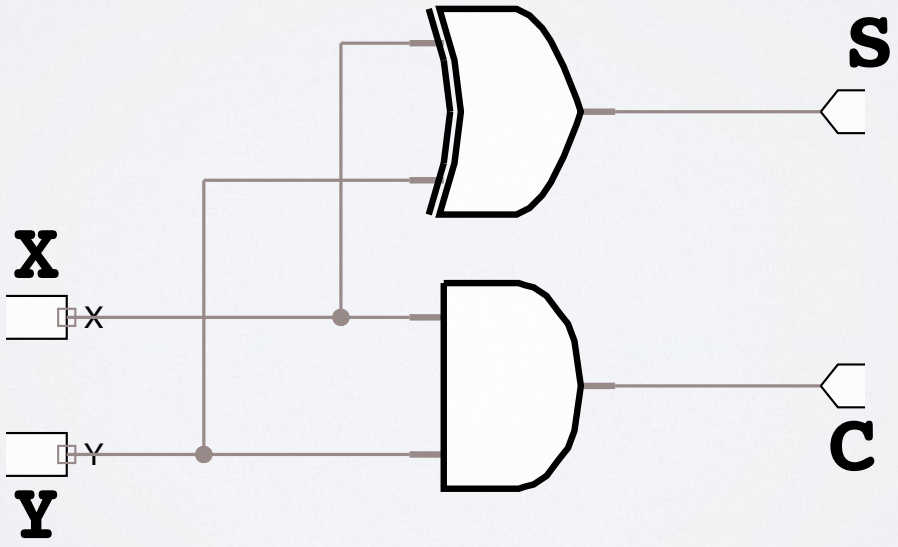
ADDITION

```
     1   0
     1   1   0
+    0   1   1
  -----------
(1)  0   0   1
```

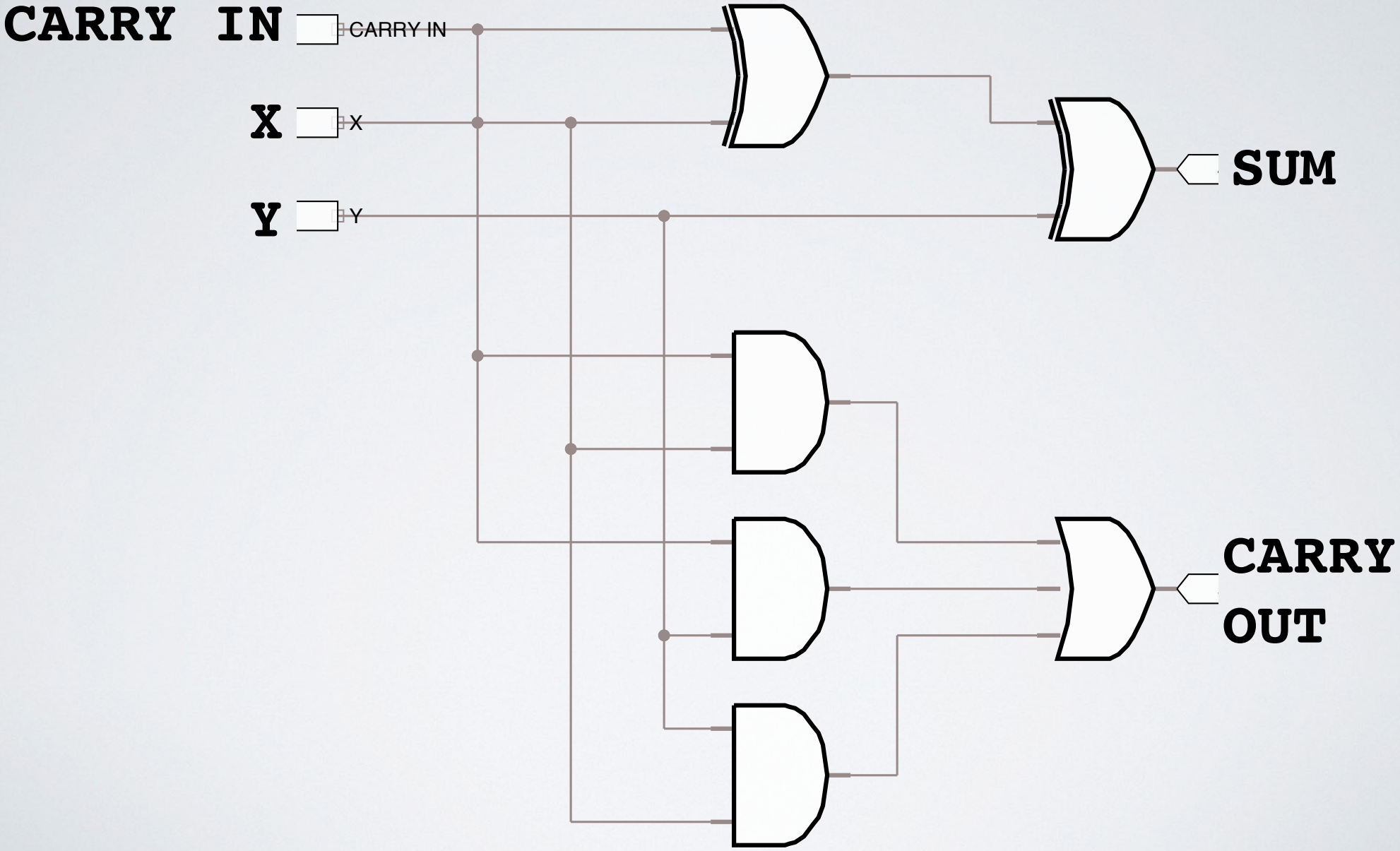
ADDITION — HALF ADDER

```
      1   0
      1   1   0
  +   0   1   1
  -----------
(1)   0   0   1
```

x	y	c	s
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

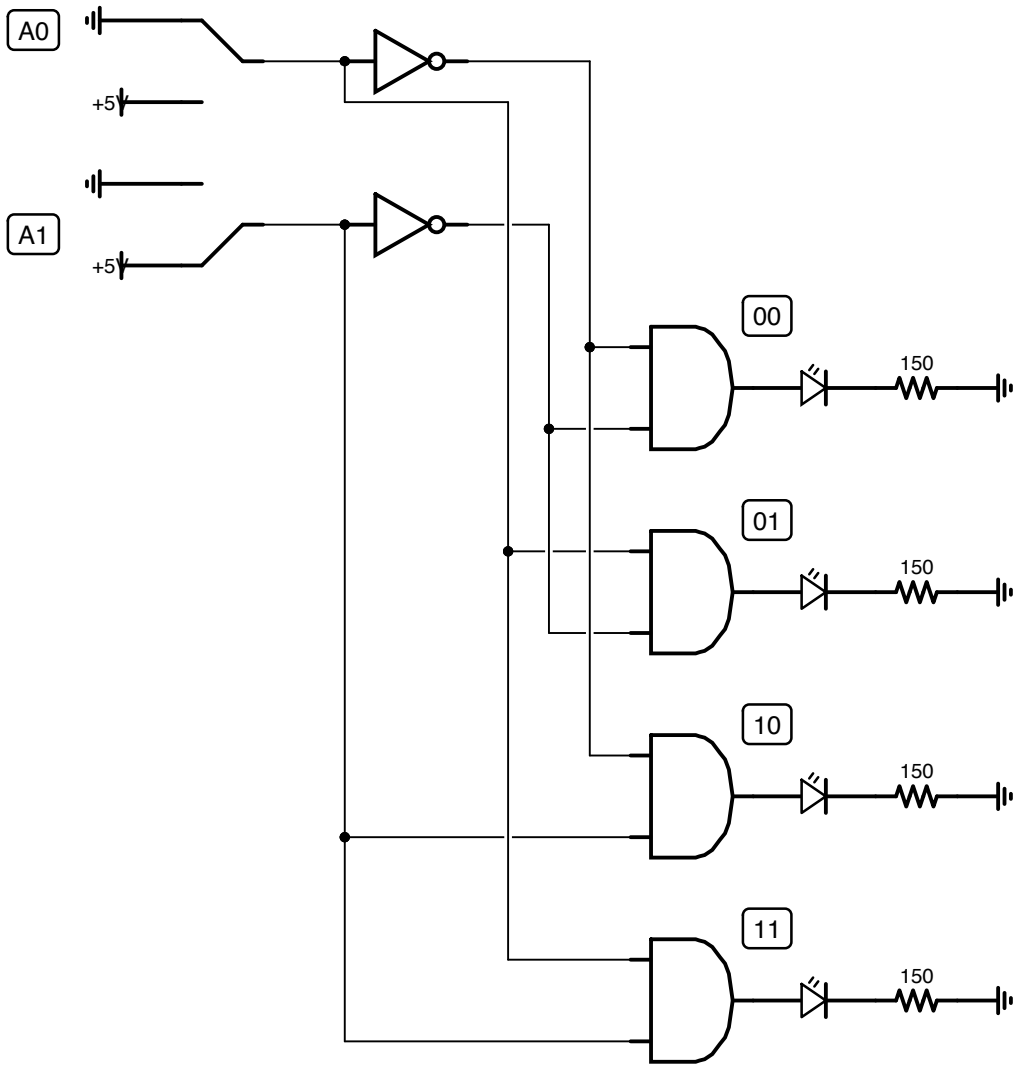


ADDITION — FULL ADDER



BASIC DECODER — DEMO

BASIC DECODER



Decoder Truth Table (2 inputs 4 output)

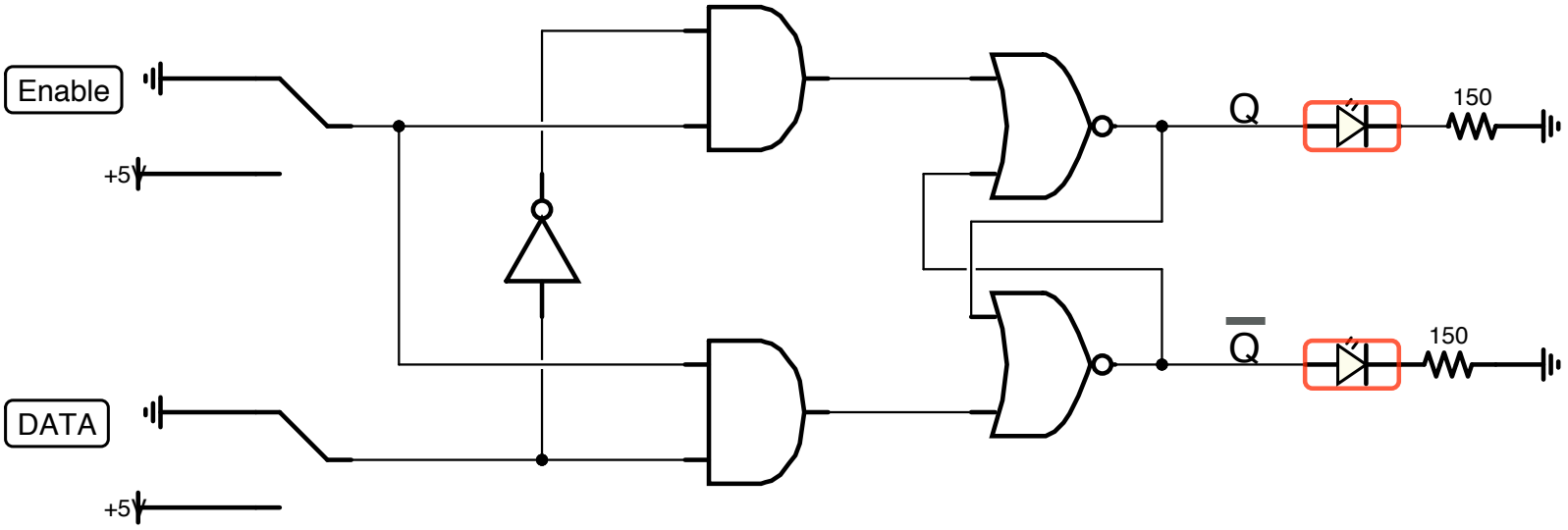
A1	A0	00	01	10	11
0	0	1	0	0	0
0	0	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Notice each unique value of A0 and A1 only activates/asserts one and only one output — We can use this to create an “Address Bus” where values of the inputs select a particular memory cell we want to access.

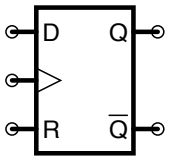
SEQUENTIAL

BASIC DATA LATCH
SINGLE BIT MEMORY CELL
D-FLIP FLOP — DEMO

BASIC DATA LATCH (D-FlipFlop) – Memory

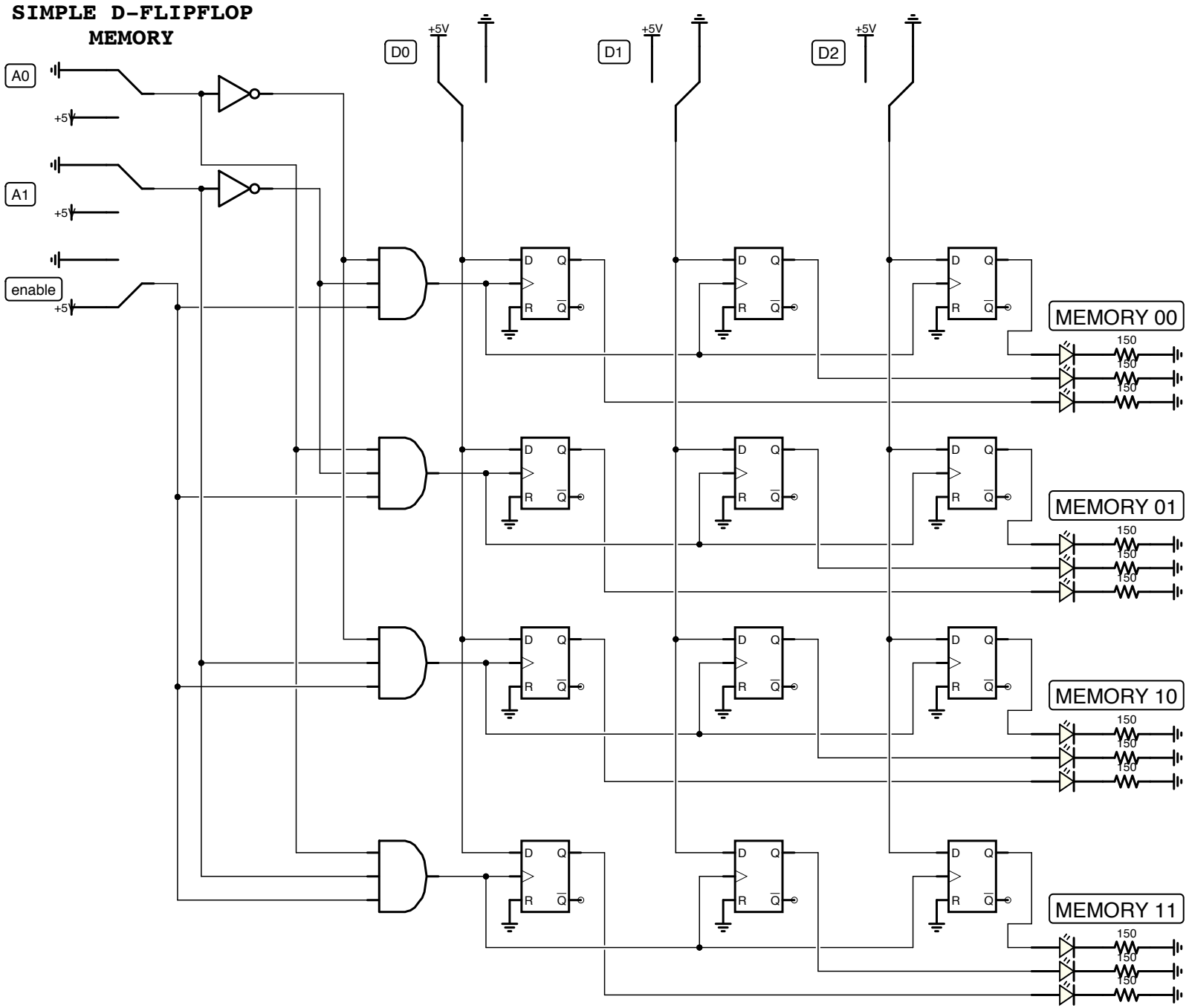


Symbol for a
D Flip-Flop
(A little more fancy
that the one above since
it has a Reset-R line)
> is the enable



The doubling-back of outputs to inputs give Flips-Flops their unique nature in which their outputs depend on prior input. This D-Flip-Flop will act like a single bit memory. When the enable is turned on it will latch onto the value on the Data line. It's Q output will then continue to reflect this value until the enable line is turned on again. We can build a multi-bit memory using D-Flip-Flops and a Decoder.

PUTTING IT TOGETHER SIMPLE 3 BIT 4 LOCATION MEMORY — DEMO


SIMPLE D-FLIPFLOP
MEMORY
A0
A1
enable
+5V
D0
D1
D2
D
Q
R
MEMORY 00
MEMORY 01
MEMORY 10
MEMORY 11
150

NEXT CLASS

CHAPTER TWO CSAPP: DATA REPRESENTATION