CSO-Recitation 13

CSCI-UA 0201-007

R13: Assessment 11

Assessment 11

Q1 Boolean laws

Which of the following Boolean laws hold? Below, A, B, C could refer to either a Boolean variable or a Boolean expression

A. R1: A+0=A
B. R2: A+0=0
C. R3: A+1=1
D. R4: A+1=A
E. R5: A·(B+C)=A·B + A·C
F. R6: A + $\bar{A}$=1
G. R7: A · $\bar{A}$ = 0

Basic law:

- A ·0 =0, A ·1=A
- A+0=A, A+1=1

Distribution law

Inverse law

Q2 Simplify boolean expression

- Simplify boolean expression (A+B) $\cdot(\bar{A} + \bar{B})$.
- You may write `*` for $\cdot$, and write `barA` for $\bar{A}$ (or `barB` or $\bar{B}$)
- (A+B)*(barA+barB)
- =(A+B)*barA + (A+B)*barB   Distribution law
- =barA*A + barA*B + barB*A + barB*B   Distribution law
- =0+barA*B+barB*A+0   Inverse law
- =barA*B+barB*A   Basic law

Q3 Simplify boolean expression

- When simplifying the Boolean expression in Q2, which of the Boolean laws in shown Q1 are needed?

A. R1
B. R2
C. R3
D. R4
E. R5
F. R6
G. R7

Q4 Boolean circuit

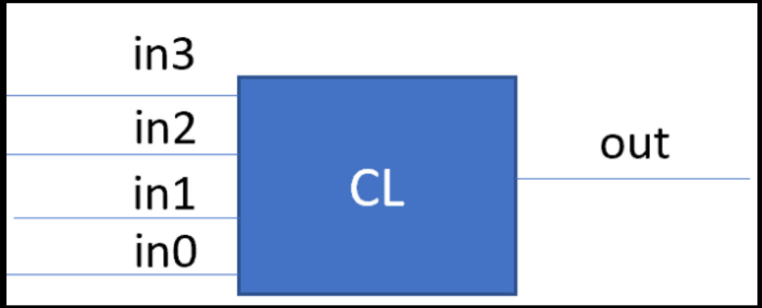
- If you are to use a single logic gate to implement the simplified expression in Q2. Which gate should you use?

A. AND
B. OR
C. NOR
D. NAND
E. XOR
F. None of the above

NAND(A,B) = bar(A*B) = barA+barB

XOR(A,B) = A*barB + B*barA

Q5 Combinatorial circuit



- In this question, you are asked to implement a combinatorial circuit that takes a 4-bit input and outputs a single bit indicating whether the unsigned 4-bit integer represented by $b3b2b1b0$ is a prime number or not.
- **Q5.1** Truth table
- How many total rows does the truth table corresponding to the 4-bit prime number detector circuit have?
- 16

#row of truth table:

- have 4 input signals, each represents 1 bit
- how many bit patterns?
- 2^4 = 16

Q5.2 Truth Table

- How many of the rows in the truth table of Q5.1 corresponds to the output bit value *o*=1?
- 6

    - Prime: 2, 3, 5, 7, 11, 13

Q5.3 Product of terms

- The prime number detector circuit can be built as a sum of products where each product term corresponds to a row in Q5.2. Please write the product term that corresponds to the input b3b2b1b0 = (1011)2
- B3 Barb2 b1 b0

- (1011)2 = 11, is a prime
    - output = 1
- b3, b1, b0 = 1, remain the same
- b2 = 0, => Barb2

Q5.4 ROM

- If you are to using a ROM to implement the prime number detector circuit. What is the minimal size of the ROM required?
- 4 variables
- h = 2^4 = 16
  - i.e. #input bit patterns
- w = 1 (one bit to indicate whether this is a prime number or not)

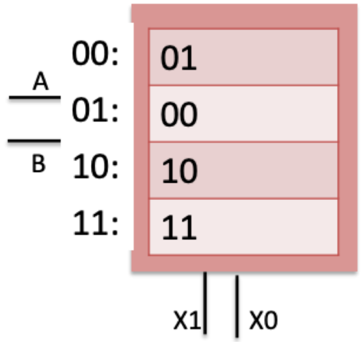
ROM (read-only memory)

- A n x m ROM can store the truth table for m functions defined on $\log_2 n$ variables.

$$X_1 = A$$

$$X_0 = \overline{A} \bullet \overline{B} + A \bullet B$$

A	B	X1	X0
0	0	0	1
0	1	0	0
1	0	1	0
1	1	1	1



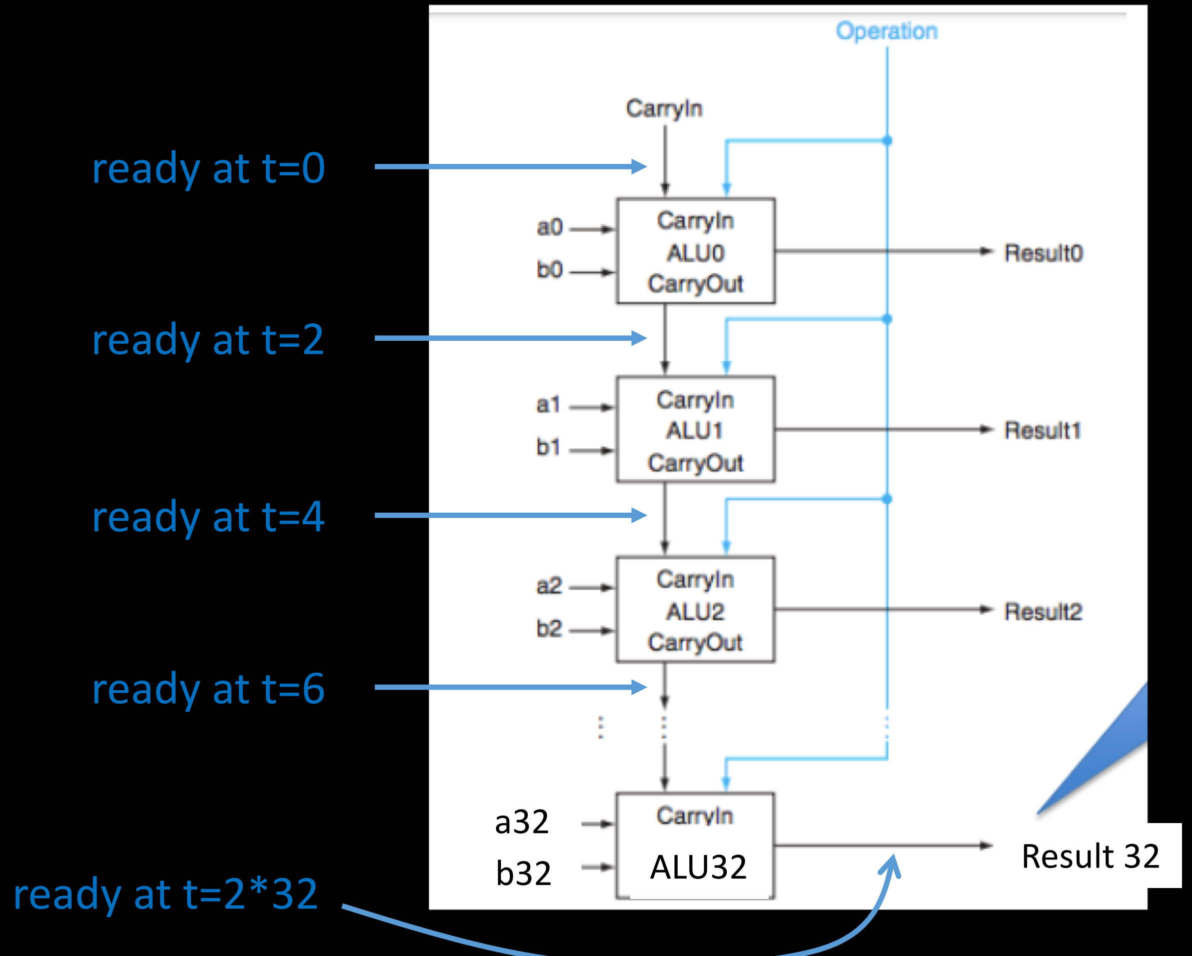
Q5.4 ROM

- Following Q5.4, what is the value of the ROM entry at index or address $(1010)_2$?
- 0

- $(1010)_2$ = 10, not a prime
  - output = 0

Q6 Ripple carry

- If a 1-bit adder's gate delay is 2, then what is the gate delay of a 32-bit ripple carry?
- 64



Lab 4 Optimization

Simple implicit list implementation

- First-fit algorithm
- Simply optimized realloc function (3 cases):
  - Shrink: directly decrease the size
  - Expand:
    - Next chunk is a free chunk, and the size is sufficient: utilize the next chunk
    - Otherwise, free the current chunk and allocate a new one

Simple implicit list implementation

Even though the utilization is high, performance is low

Results for mm malloc:

trace	valid	util	min	ops	secs	Kops	PerfInde
0	yes	99%	2012279	5694	0.013695	416	62
1	yes	99%	1679165	5848	0.011058	529	60
2	yes	99%	3165325	6648	0.022223	299	60
3	yes	99%	3421135	5380	0.015841	340	62
4	yes	50%	8190	14400	0.000233	61803	69
5	yes	90%	14532295	4800	0.008793	546	65
6	yes	88%	14432586	4800	0.009881	486	59
7	yes	54%	1152000	12000	0.158506	76	32
8	yes	47%	576000	24000	0.352995	68	28
9	yes	35%	615040	14401	0.000467	30837	61
10	yes	76%	28119	14401	0.000321	44863	85

Performan … + 40.0 * (your throughput)/(libc's throughput)

11 out of … rage performance index 58.5 (out of 100.0)

Utilization is low.

Simple implicit list implementation

Results for mm malloc:

trace	valid	util	min	ops	secs	Kops	PerfIndex
0	yes	99%	2012279	5694	0.013695	416	62
1	yes	99%	1679165	5848	0.011058	529	60
2	yes	99%	3165325	6648	0.022223	299	60
3	yes	99%	3421135	5380	0.015841	340	62
4	yes	50%	8190	14400	0.000233	61803	69
5	yes	90%	14532295	4800	0.008793	546	65
6	yes	88%	14432586	4800	0.009881	486	59
7	yes	54%	1152000	12000	0.158506	76	32
8	yes	47%	576000	24000	0.352995	68	28
9	yes	35%	615040	14401	0.000467	30837	61
10	yes	76%	28119	14401	0.000321	44863	85

Performance index = 60.0 * util + 40.0 * (your throughput)/(libc's throughput)
11 out of 11 traces passed, average performance index 58.5 (out of 100.0)

Simple implicit list implementation

```
Results for mm malloc:
```

trace	valid	util	min	ops	secs	Kops	PerfIndex
0	yes	99%	2012279	5694	0.013695	416	62
1	yes	99%	1679165	5848	0.011058	529	60
2	yes	99%	3165325	6648	0.022223	299	60
3	yes	99%	3421135	5380	0.015841	340	62
4	yes	50%	[illegible]				
5	yes	90%	[illegible]				
6	yes	88%	[illegible]				
7	yes	54%	[illegible]				
8	yes	47%	576000	[illegible]	[illegible]	68	28
9	yes	35%	615040	14401	0.000467	30837	61
10	yes	76%	28119	14401	0.000321	44863	85

First try to optimize these two traces
(realloc trace)

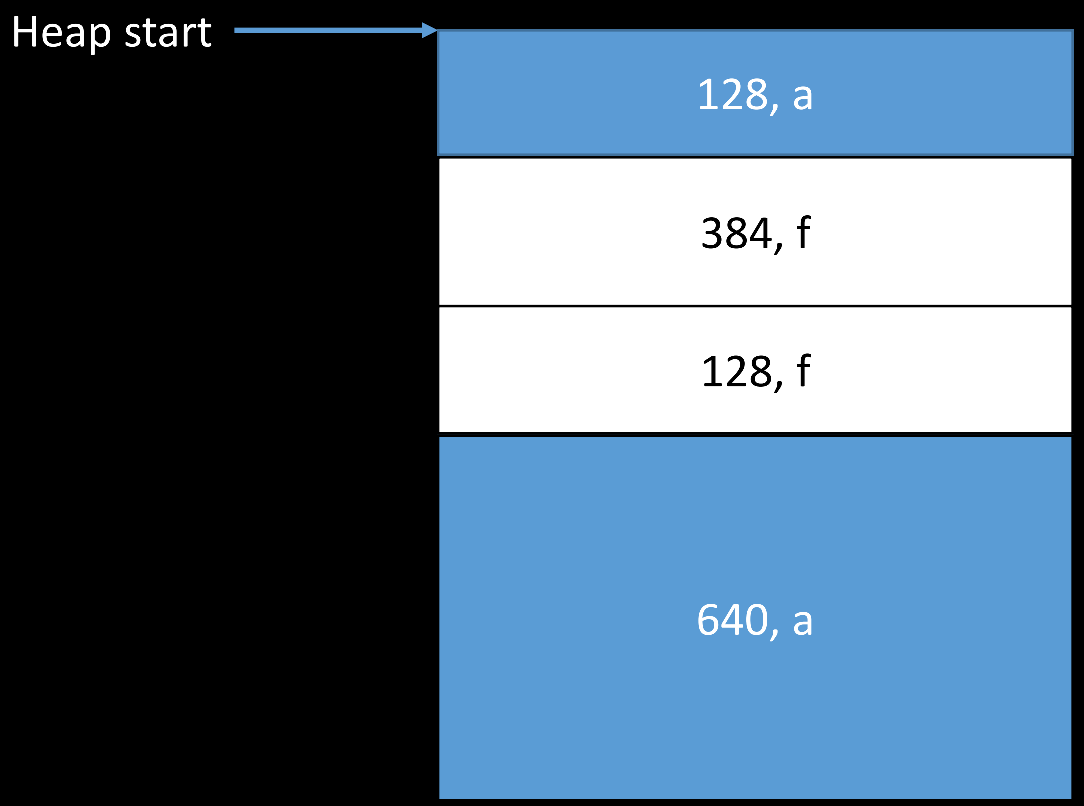
```
Performance index = 60.0 * util + 40.0 * (your throughput)/(libc's throughput)
11 out of 11 traces passed, average performance index 58.5 (out of 100.0)
```

realloc trace

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```

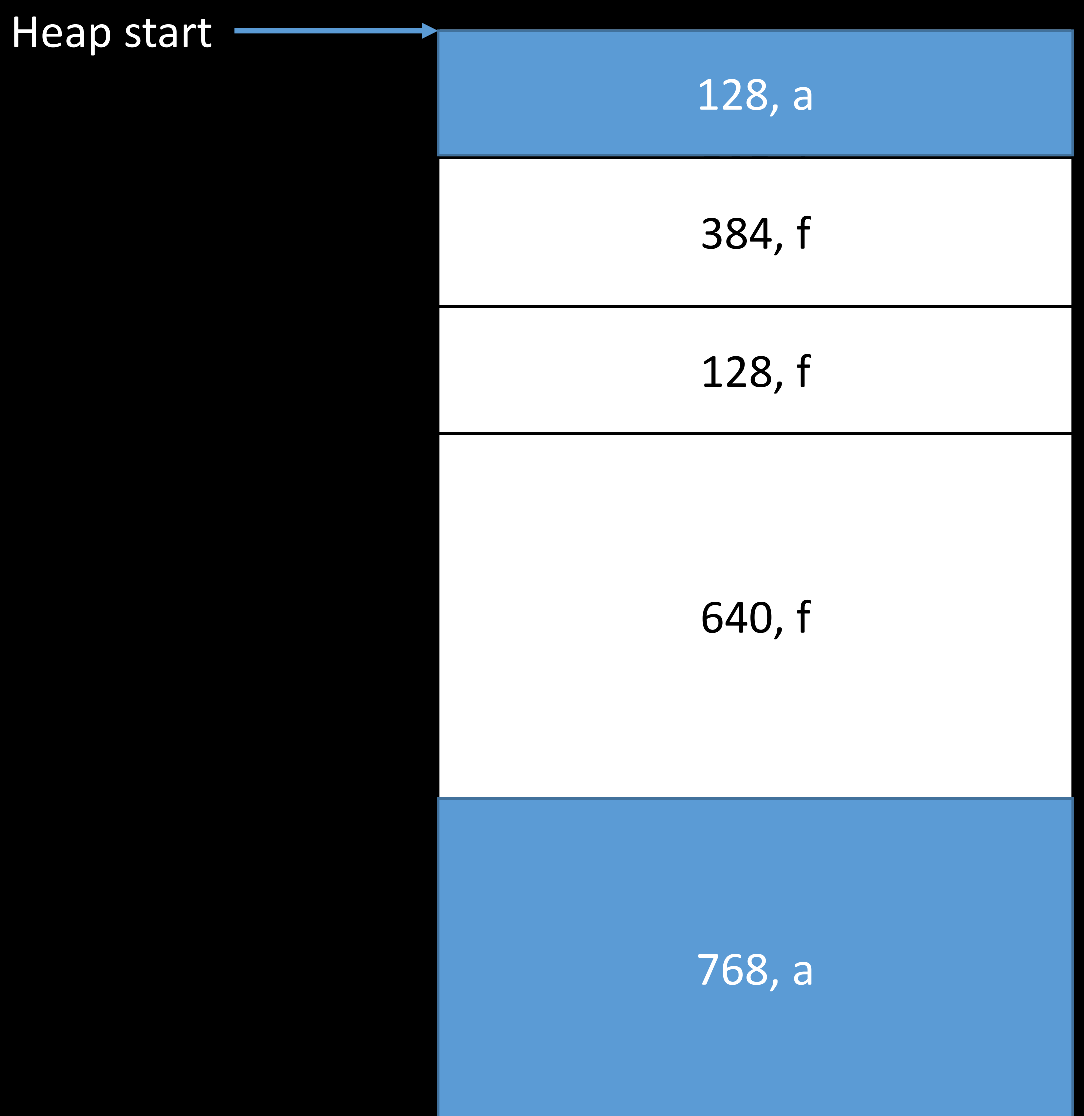
realloc trace

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```



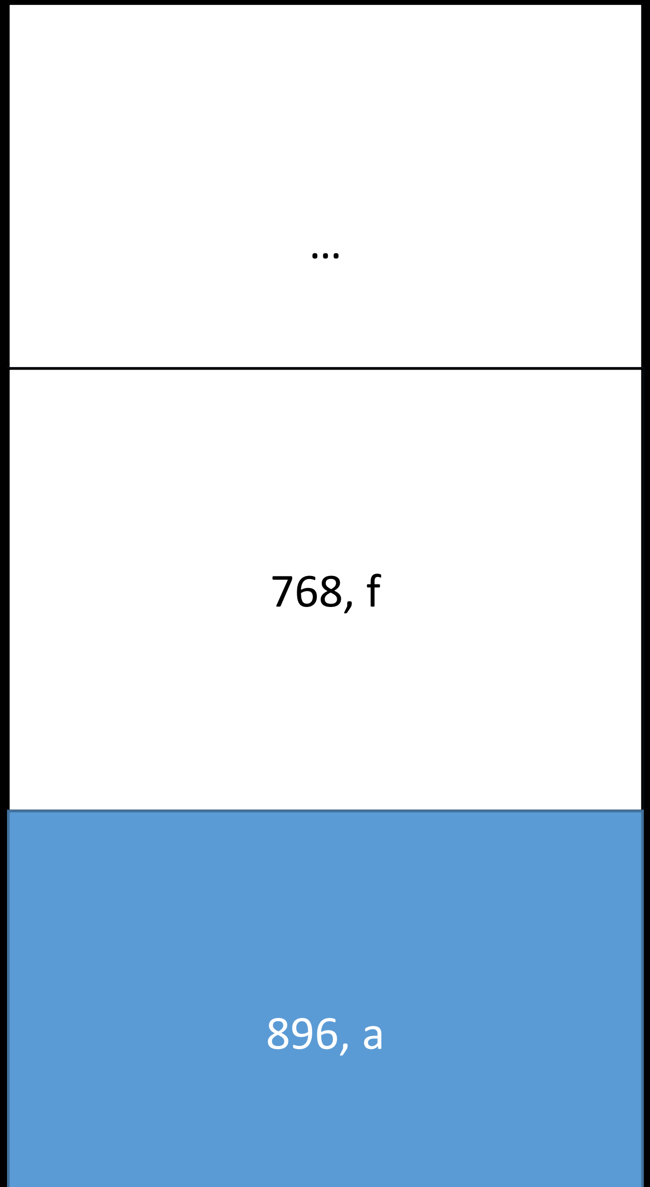
realloc trace

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```



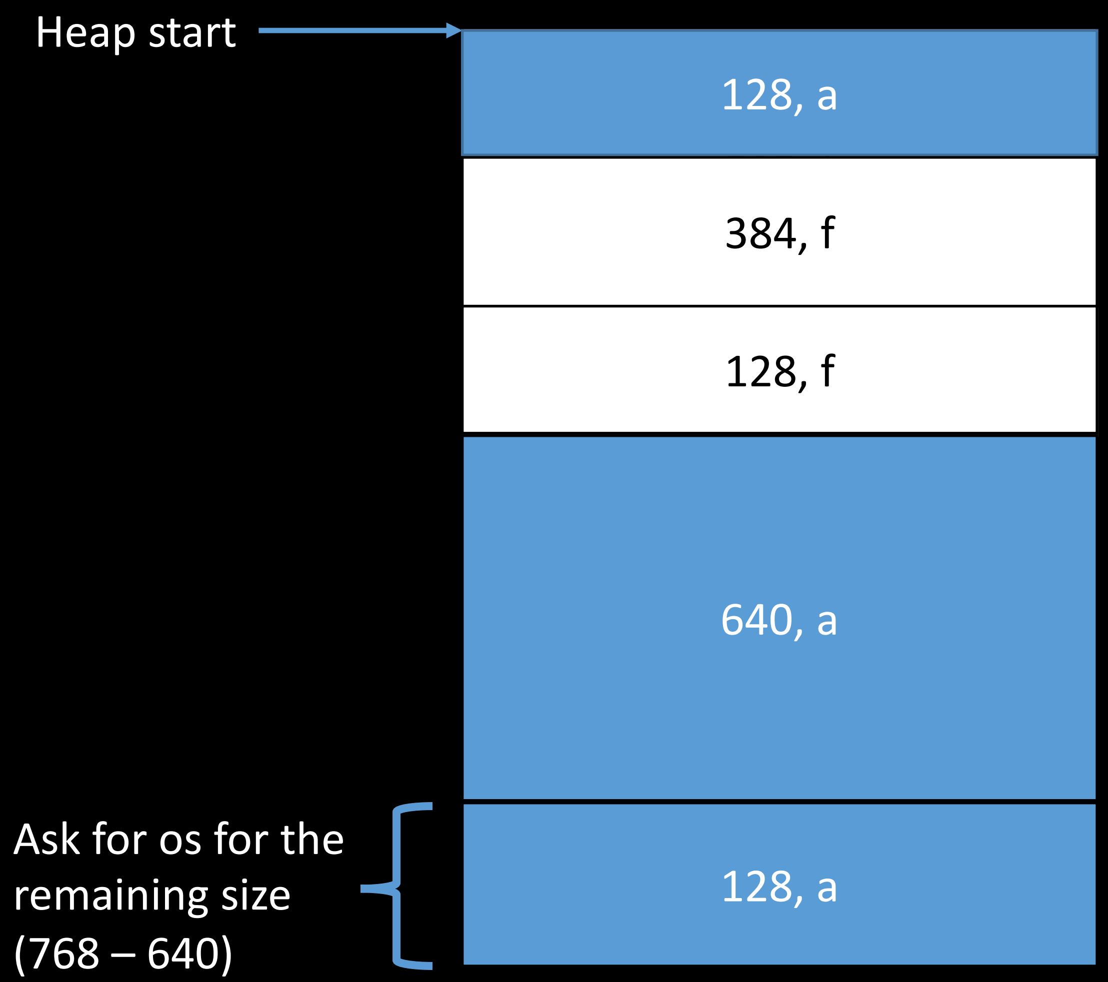
realloc trace

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```



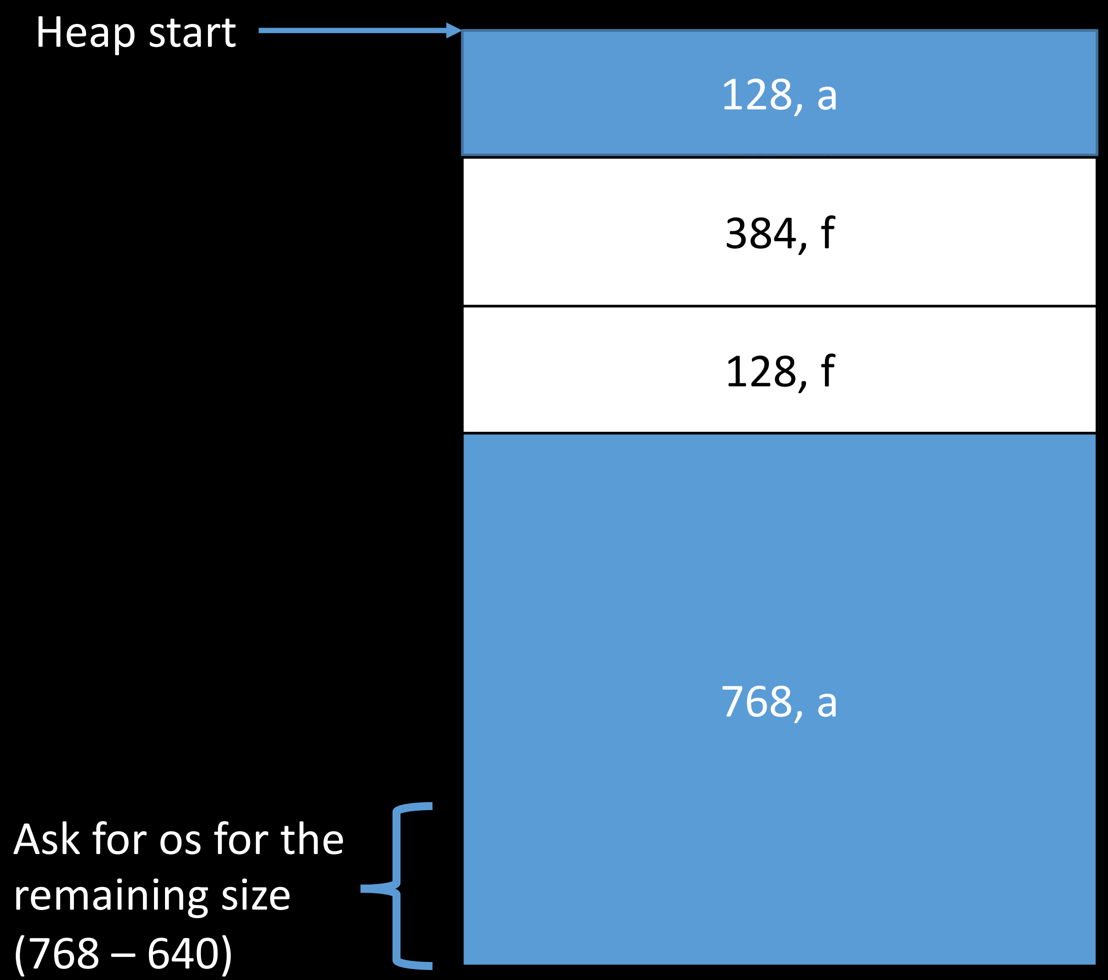
realloc trace - optimization

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```

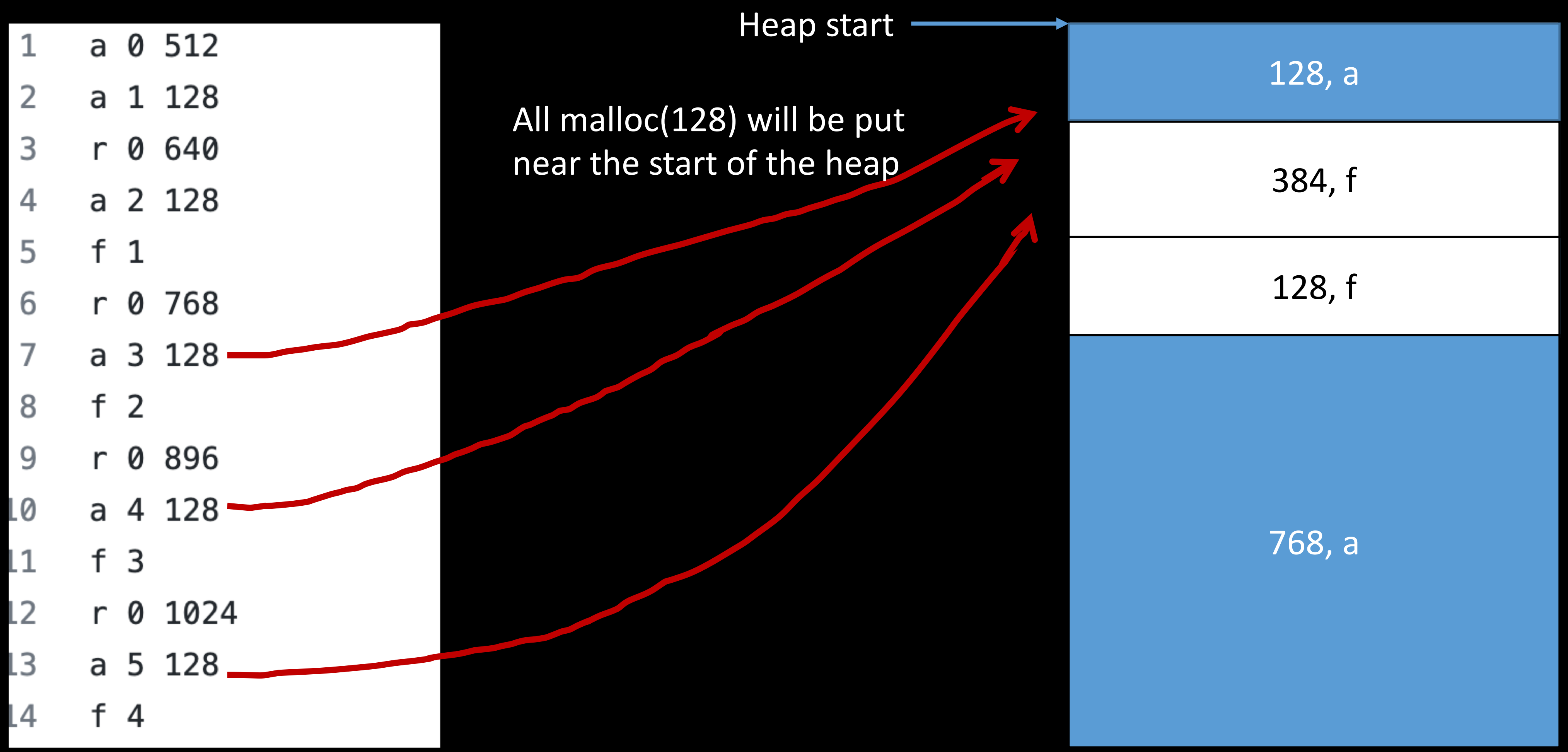


realloc trace - optimization

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```

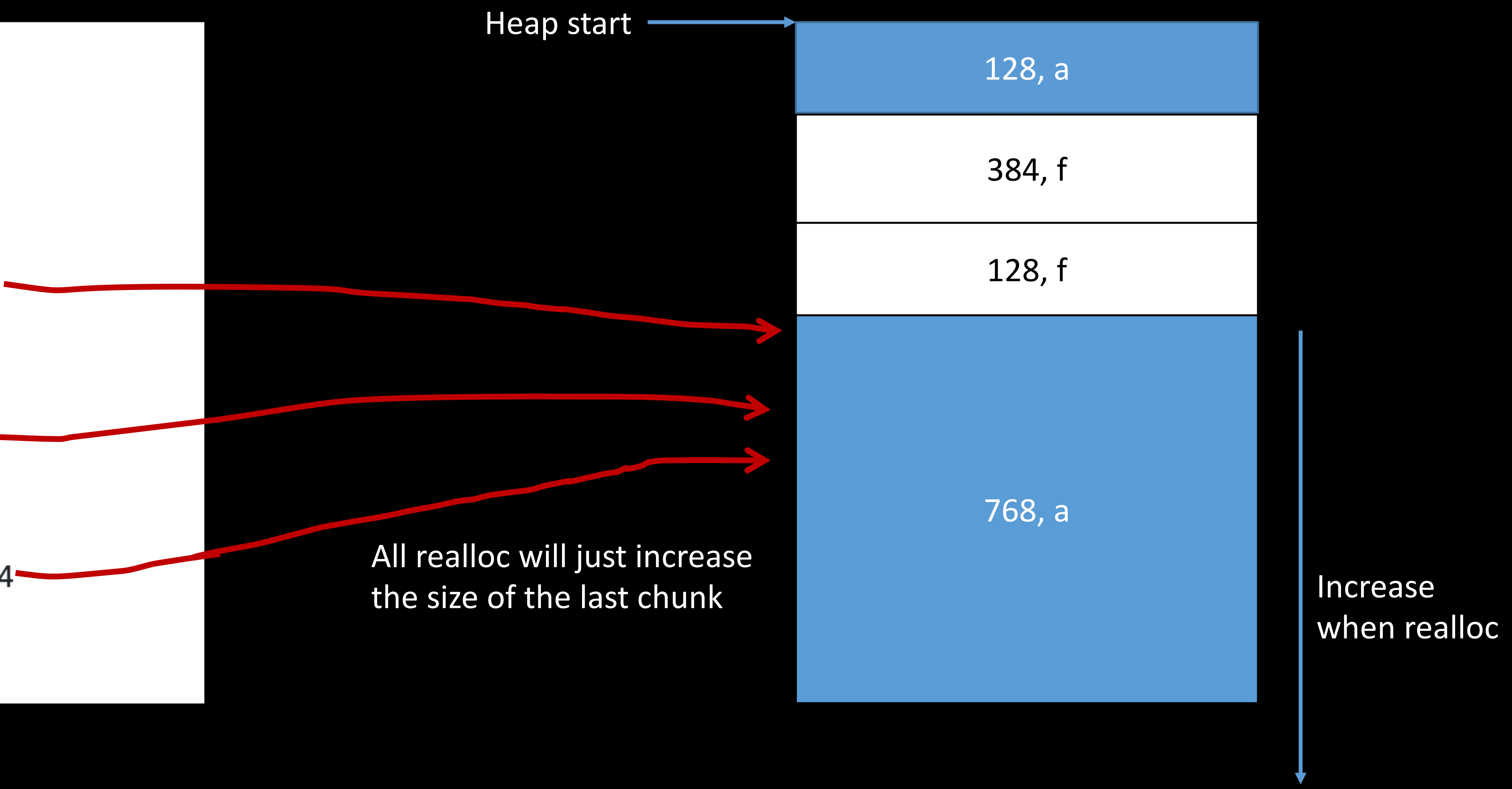


realloc trace - optimization



realloc trace - optimization

```
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
a 3 128
f 2
r 0 896
a 4 128
f 3
r 0 1024
a 5 128
f 4
```



realloc trace - optimization

Before:

Results for mm malloc:

trace	valid	util	min	ops	secs	Kops	PerfIndex
0	yes	99%	2012279	5694	0.013695	416	62
1	yes	99%	1679165	5848	0.011058	529	60
2	yes	99%	3165325	6648	0.022223	299	60
3	yes	99%	3421135	5380	0.015841	340	62
4	yes	50%	8190	14400	0.000233	61803	69
5	yes	90%	14532295	4800	0.008793	546	65
6	yes	88%	14432586	4800	0.009881	486	59
7	yes	54%	1152000	12000	0.158506	76	32
8	yes	47%	576000	24000	0.352995	68	28
9	yes	35%	615040	14401	0.000467	30837	61
10	yes	76%	28119	14401	0.000321	44863	85

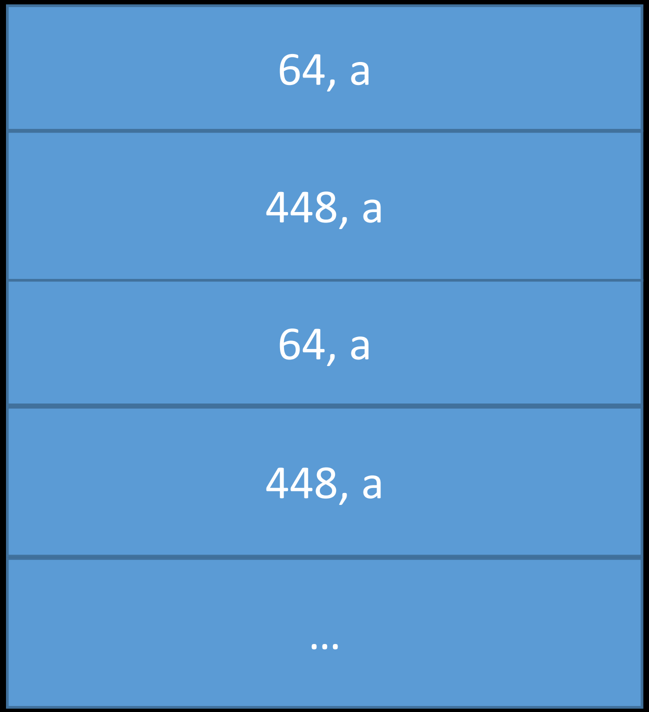
Performance index = 60.0 * util + 40.0 * (your throughput)/(libc's throughput)
11 out of 11 traces passed, average performance index 58.5 (out of 100.0)

After:

9	yes	100%	615040	14401	0.000380	37897	99
10	yes	87%	28119	14401	0.000268	53735	92

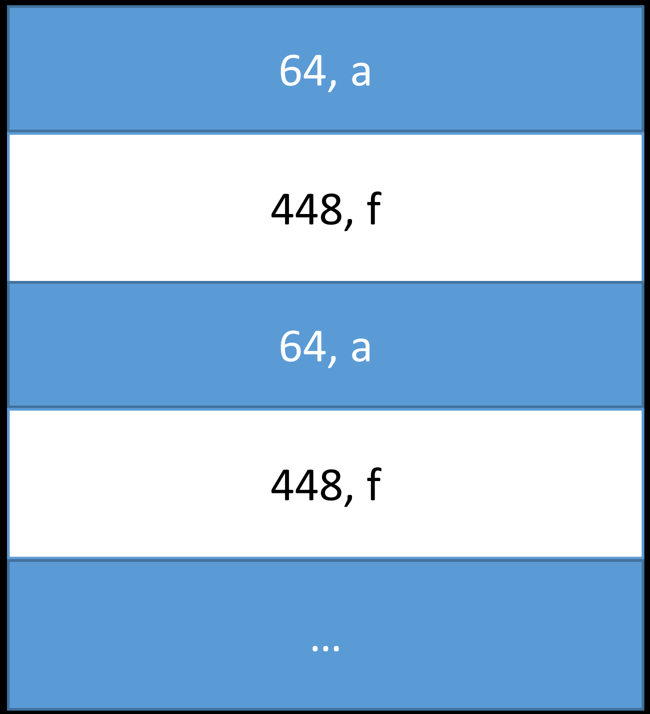
binary trac

a 0 64
a 1 448
a 2 64
a 3 448
a 4 64
a 5 448
…
f 1
f 3
f 5
…
a 4000 512
a 4001 512
a 4002 512
a 4003 512
…



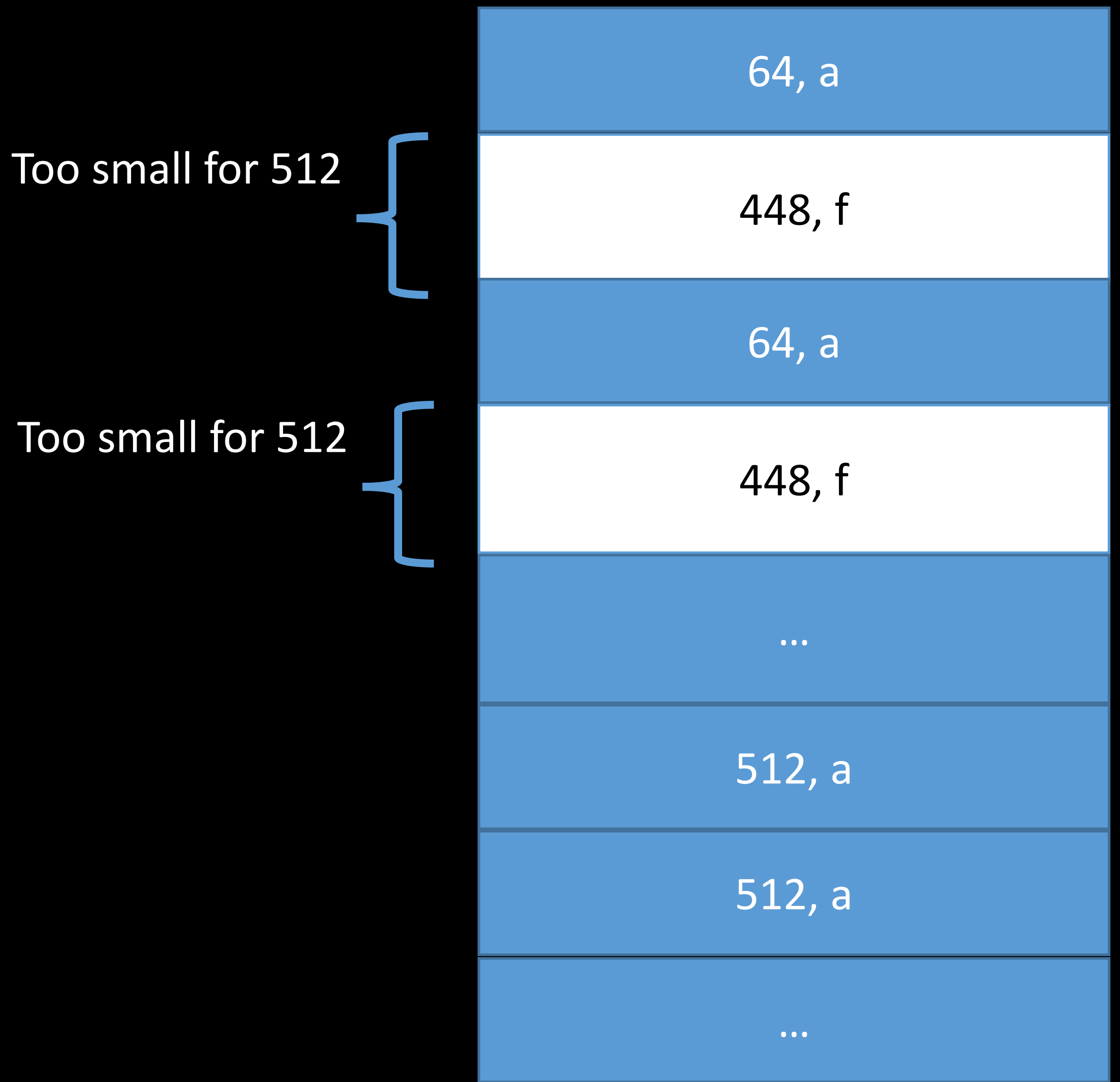
binary trace

a 0 64
a 1 448
a 2 64
a 3 448
a 4 64
a 5 448
...
f 1
f 3
f 5
...
a 4000 512
a 4001 512
a 4002 512
a 4003 512
...



binary trace

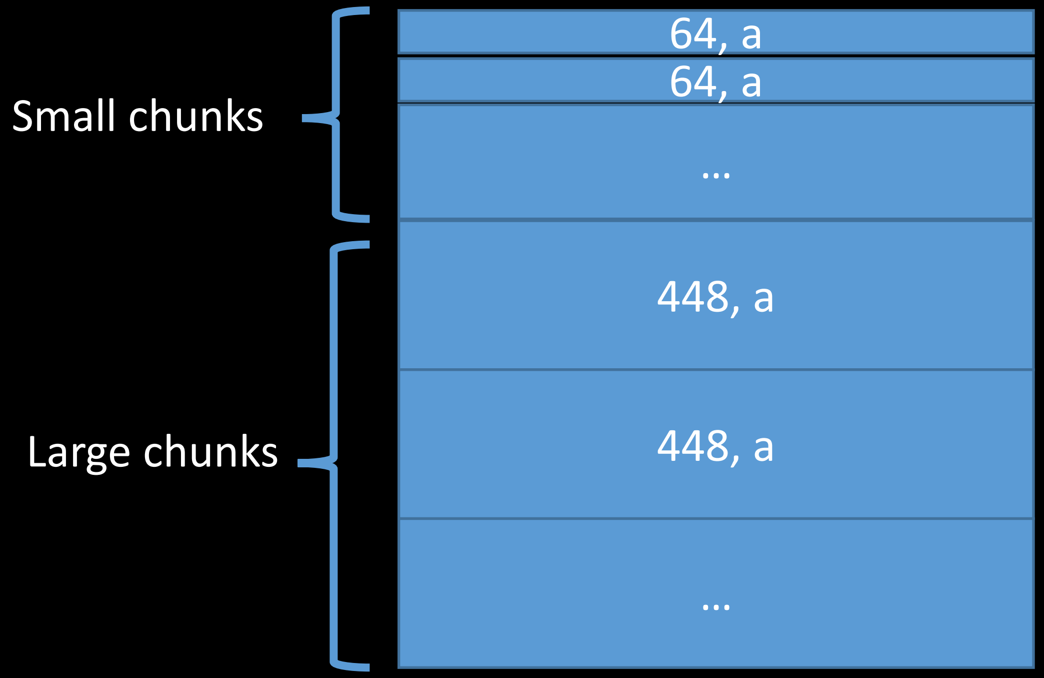
a 0 64
a 1 448
a 2 64
a 3 448
a 4 64
a 5 448
…
f 1
f 3
f 5
…
a 4000 512
a 4001 512
a 4002 512
a 4003 512
…



binary trace - optimization

a 0 64
a 1 448
a 2 64
a 3 448
a 4 64
a 5 448
…
f 1
f 3
f 5
…
a 4000 512
a 4001 512
a 4002 512
a 4003 512
…

- We can separate the memory allocation for small and large chunks
  - Try to put the small chunks together
- *JRockit JVM* follows this principle (https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html)



binary trace - optimization

Before:

Results for mm malloc:

trace	valid	util	min	ops	secs	Kops	PerfIndex
0	yes	99%	2012279	5694	0.016873	337	61
1	yes	99%	1679165	5848	0.015378	380	61
2	yes	99%	3165325	6648	0.027702	240	61
3	yes	99%	3421135	5380	0.018212	295	62
4	yes	50%	8190	14400	0.000344	41893	63
5	yes	90%	14532295	4800	0.011592	414	61
6	yes	88%	14432586	4800	0.011259	426	60
7	yes	51%	1152000	12000	0.185830	65	30
8	yes	47%	576000	24000	0.400840	60	28
9	yes	98%	615040	14401	0.000351	40971	98
10	yes	76%	28119	14401	0.000345	41708	70

Performance index = 60.0 * util + 40.0 * (your throughput)/(libc's throughput)

11 out of 11 traces passed, average performance index 59.5 (out of 100.0)

After:

7	yes	82%	1152000	12000	0.119570	100	49
8	yes	79%	576000	24000	0.300637	80	47

Other optimization

- To optimize utilization:
  - Add footer to fully utilize the free chunks (coalesce adjacent free chunks is possible)
- To optimize the performance:
  - Use segregated and explicit list to find a suitable chunk faster
  - Use next fit instead of first fit

Results for mm malloc:

trace	valid	util	ops	secs	Kops
0	yes	99%	5694	0.000394	14470
1	yes	99%	5848	0.000340	17195
2	yes	99%	6648	0.000378	17611
3	yes	99%	5380	0.000735	7318
4	yes	99%	14400	0.000449	32071
5	yes	95%	4800	0.000693	6926
6	yes	95%	4800	0.000654	7339
7	yes	95%	12000	0.000599	20043
8	yes	88%	24000	0.003397	7064
9	yes	99%	14401	0.000240	59979
10	yes	97%	14401	0.000222	64928
Total		97%	112372	0.008100	13872

Perf index = 58 (util) + 40 (thru) = 98/100

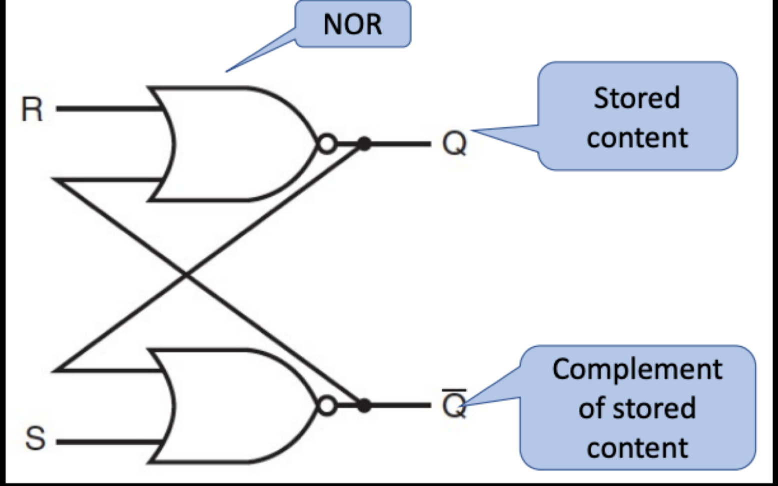
Sequential logic

Building Blocks

Sequential Logic

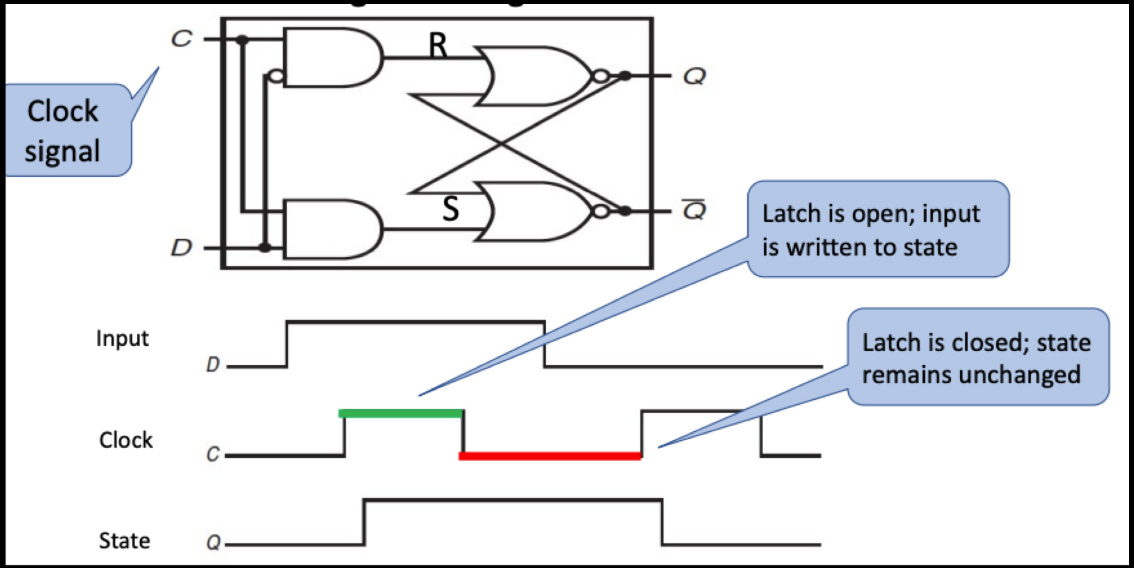
- There is memory
  - Outputs depend on prior state as well as the current inputs
  - State can be stored and used later
- We rely on clock signals
  - Clock signals tell us when things should happen
  - We should only write to state when the clock is set a certain way

SR Latch



- Constructed from two NOR gates
  - You can either Set the latch (make it remember 1), or Reset it (make it remember 0)
- Two inputs: S and R
- Two outputs: Q and NOT Q
  - Q is what it remembers, NOT Q is the opposite
- Both S and R cannot be 1 at the same time, or sadness occurs

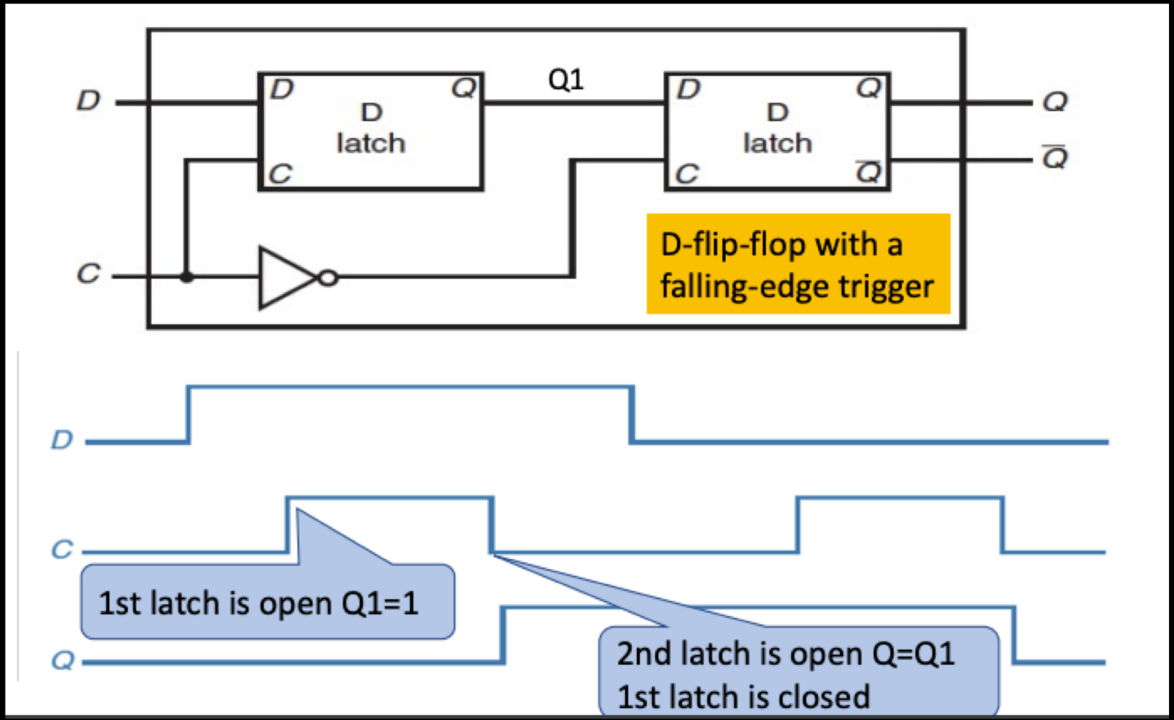
D Latch



- Constructed from some additional logic and an SR Latch
- Two inputs: C and D
- You can have the latch remember D as long as C is true
- Two outputs: Q and NOT Q
  - Q is what the latch remembers, NOT Q is the inverse
- Ensures that S and R inputs to the SR Latch aren't both true

D Flip Flop

- Constructed from some additional logic and two D latches
- Same inputs and outputs as D latches
- But, the output is only stored on a chosen clock edge



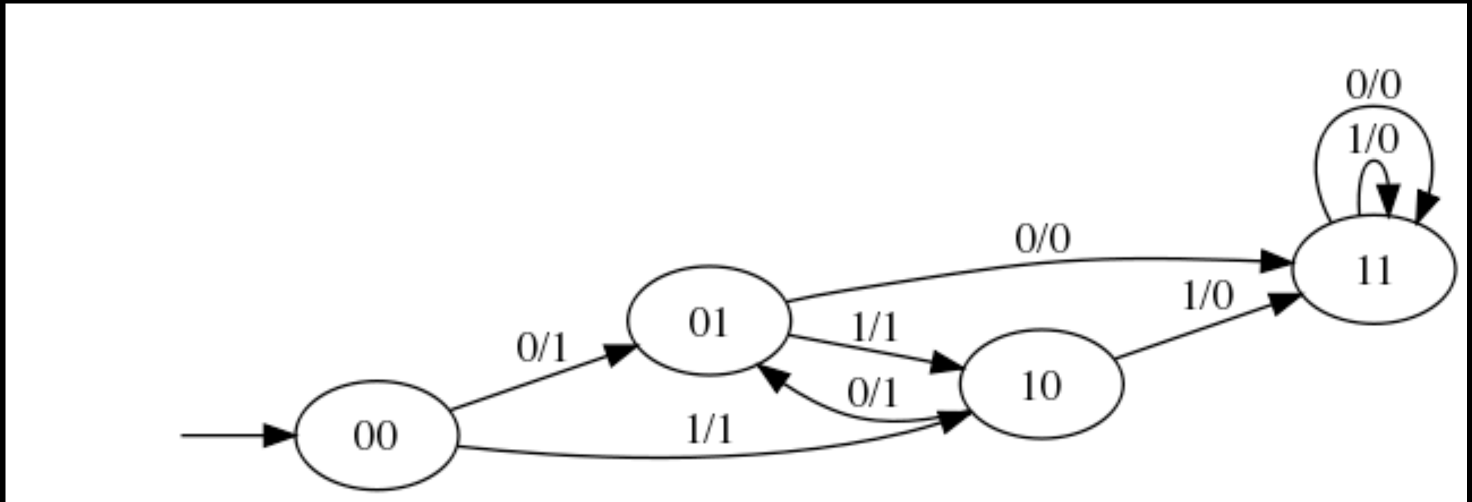
Finite State Machines

Finite State Machines

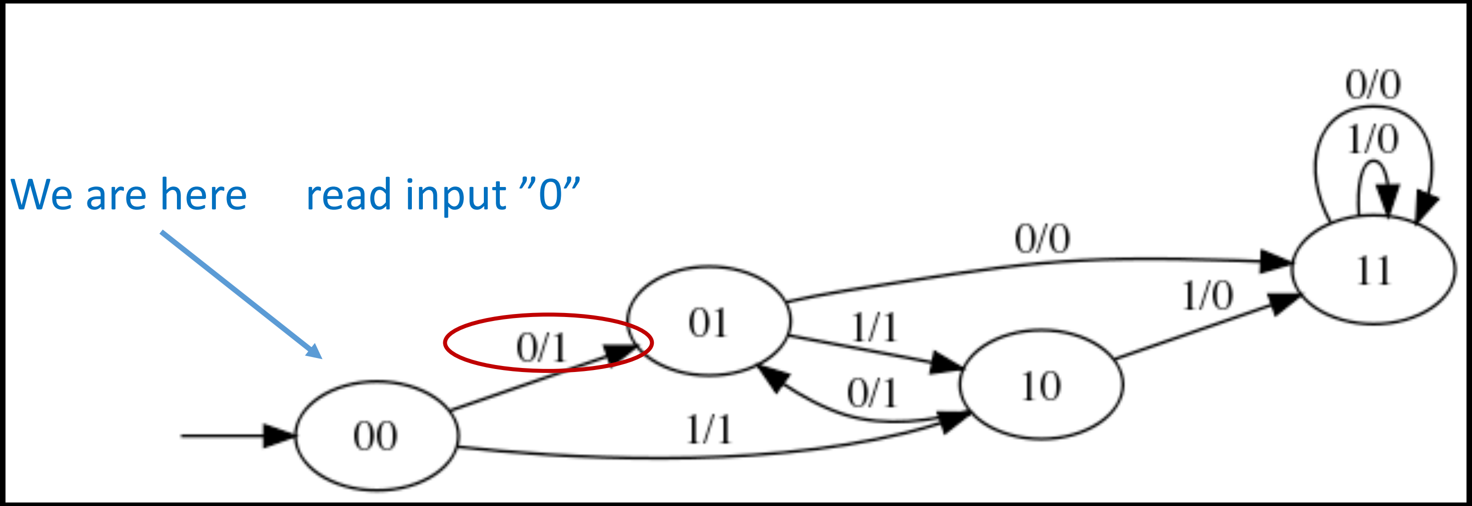
- There are a number of states, inputs, and outputs
- To the beat of the clock, we read in inputs and go to new states, and set the outputs
- Both the output and the next state are defined by the current state and the inputs
- Can be expressed as a flowchart or a truth table

An FSM example

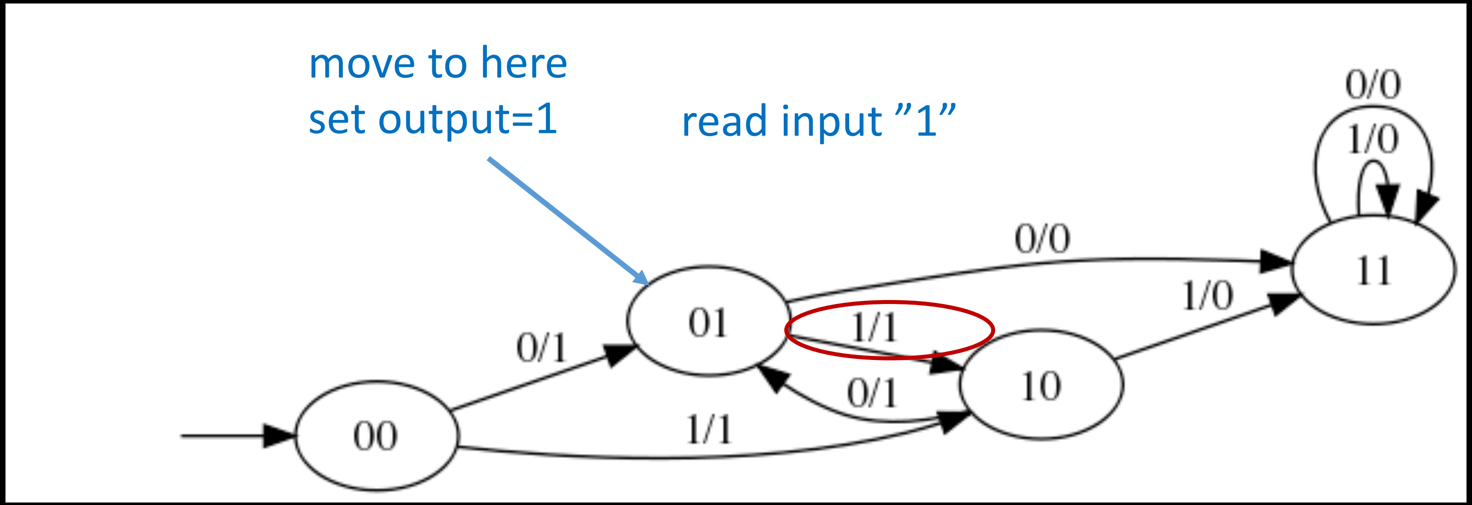
- There are 4 states
- Nodes represent states
  - Initial state is 00
- “x/y” on the arrow edge is “transition condition”
  - when input=x, follow this edge to transit into the pointed state
  - set output=y in the meantime



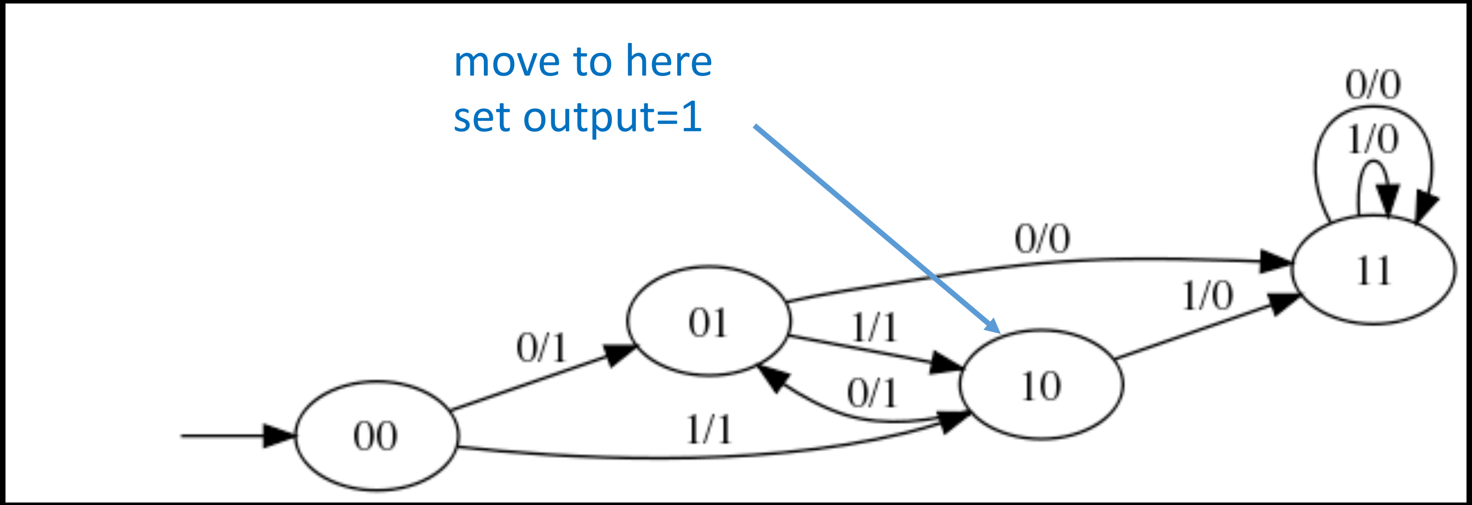
An FSM example



An FSM example

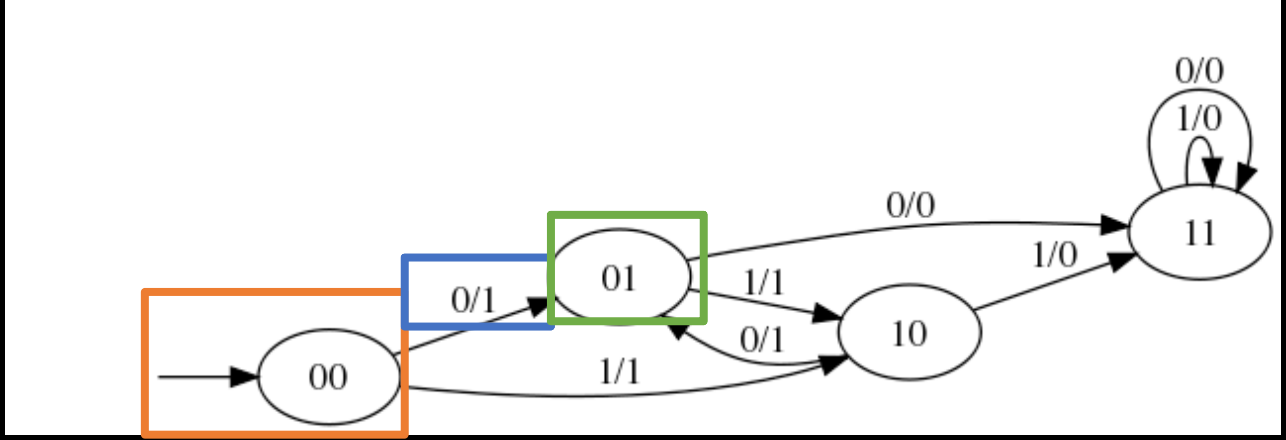


An FSM example



An FSM example

- The corresponding truth table



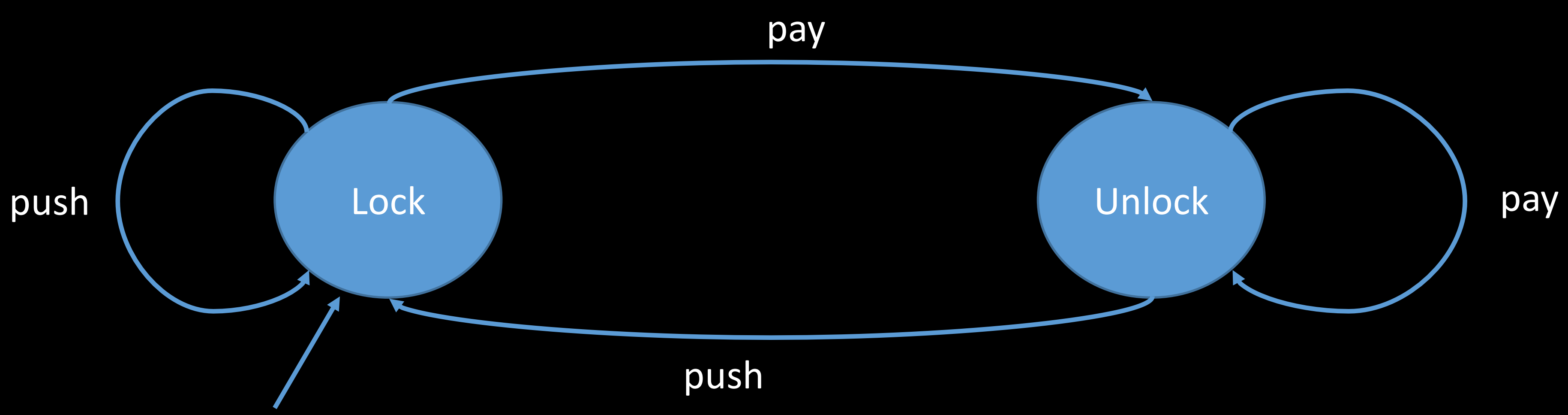
2. Observe that the following is a truth table for the FSM:

st1	st0	input	|	next st1	next st0	output
0	0	0	|	0	1	1
0	0	1	|	1	0	1
0	1	0	|	1	1	0
0	1	1	|	1	0	1
1	0	0	|	0	1	1
1	0	1	|	1	1	0
1	1	0	|	1	1	0
1	1	1	|	1	1	0

Another FSM Example

- The NYC Subway Turnstile
- There is a lock controlled by the FSM
- If the user didn't pay yet then the lock is active and the user can't push through
- If the user pays, the lock unlocks until they push through
- Draw an FSM for this
- Write out a truth table
- Create the circuit

Another FSM Example



current state	input	next state
(lock / unlock)	(pay / push)	