Simplifying a Sum-of-products Logic Expression using algebraic rules

Ex) $\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$ original SOP expr.

$\bar{A}BC + ABC + A\bar{B}C + AB\bar{C}$ algebraic reordering

$BC + A\bar{B}C + AB\bar{C}$ apply complementary rule $A+\bar{A}=1$

$B(C+\bar{C}A) + A\bar{B}C$ Distributive

$B(C+A) + A\bar{B}C$ Identities

$BC + AB + A\bar{B}C$ Demorgan

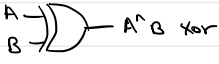
$BC + A(B+\bar{B}C)$ Refactoring

$BC + A(B+C)$ Identities

$BC + AB + AC$ Distributive

(final simplified Expression equivalent To original expression)

08/09



Proof:

w	y	$\bar{w}$	$\bar{w}y$	$w+\bar{w}y$	$w+y$
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

All possible input combinations

Same result for all possible w,y values

DeMorgan

$\neg(P \wedge Q) \iff \neg P \vee \neg Q$ formal statement

"Negation of Conjunction is equal to disjunction of negations."

{ $\wedge$ AND Conjunction
{ $\vee$ OR Disjunction

Format proof using Truth tables

P	Q	$P \wedge Q$	$\neg(P \wedge Q)$	$\neg P$	$\neg Q$	$\neg P \vee \neg Q$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Same

can prove any two-variable proposition using truth tables.

P	Q	$P \vee Q$	$\neg(P \vee Q)$	$\neg P$	$\neg Q$	$\neg P \wedge \neg Q$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Same

Gray Binary Code:

Formal Definition: A gray code assigns to each of a contiguous set of integers or to each member of a circular list: A list of symbols s.t. any two adjacent words differ by just one symbol.

* For binary, our symbols are 0 and 1.

Gray Binary code is a binary numbering system where successive values differ by only 1 bit.

ex)

0	0000	
1	0001	
2	0010	changed 2 bits
3	0011	
4	0100	changed 3 bits
5	0101	
6	0110	
7	0111	
8	1000	changed 4 bits

*Gray binary code is a reflected binary code.

The N bit gray code can be recursively generated from the list for N-1 bit gray code.

1 bit Gray Binary code

0
1

2 bit Gray Binary code

00
01
11
10

*Differ only in 1 bit

3 bit Gray Binary Code

000
001
011
010
110
111
101
100

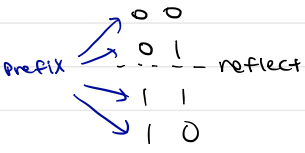
*Differ only in 1 bit

*differently ordered than regular bits

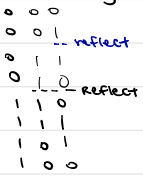
Steps to generate N-Bit Gray Binary code from N-1 bit gray binary code

1. List the N-1 bit gray binary code in order
2. List the entries in the N-1 bit gray binary code in reverse order (Reflection step)
3. Prefix the first half of your new list with 0, prefix the second half with one. (Prefix step)

ex) 2 bit Gray Binary Code



3 bit Gray Binary Code



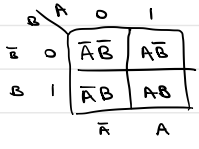
- N bit gray code is permutation from 0 to N-1 code.
  $0 \ldots 2^N - 1$
- Each number only appears once.
- N-1 bit list is embedded in the first half of N bit gray binary code sequence.
- Each entry list differs by only one bit (last → first, is only one bit different) ✡ Circular list.

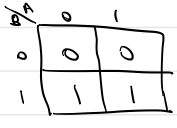
✡ ✡ ✡

Karnaugh Maps (K-map):

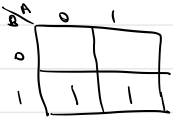
- Using visual representation of logic circuits to find the minimal number of logic circuits.

Two-variable Karnaugh map





ex)



Short hand
Just fill in the boxes where output=1

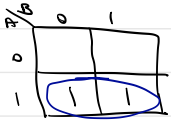
From Truth table

$\bar{A}B + AB = 1$

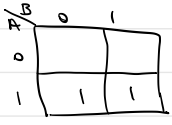
Sum of products logic expression derived directly from truth table.

(SOP logic expression)

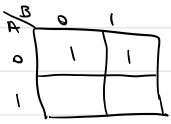
ex)



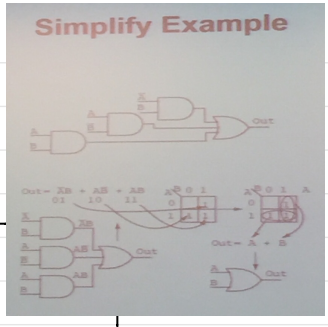
$A\bar{B} + AB$ simplifies to A.



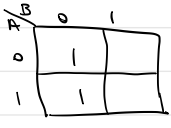
A



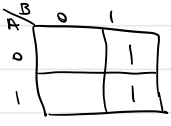
$\bar{A}$



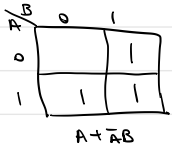
ex)



$\bar{B}$



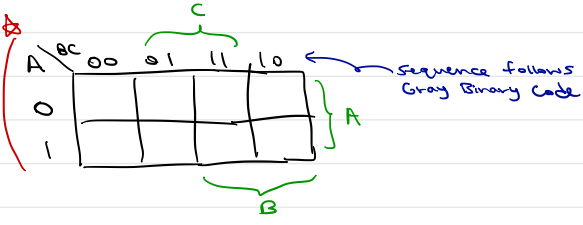
B

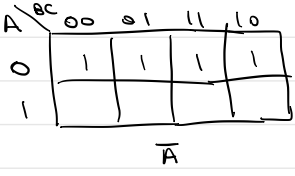


$A + \bar{A}B$

$= A + B$ ← simpler better

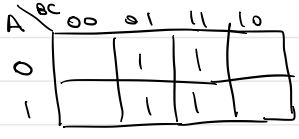
Three variable karnaugh map:





$\bar{A}$

ex)

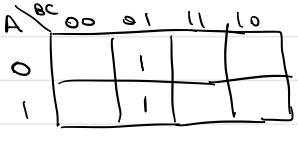


C

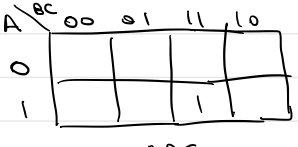
$\bar{C}$

*Wraps around like a cylinder.

$\bar{A}C$



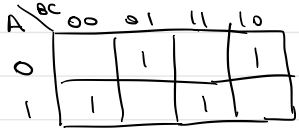
$\bar{B}C$



ABC

*The more numbers there are, simpler the expression.

More ex)

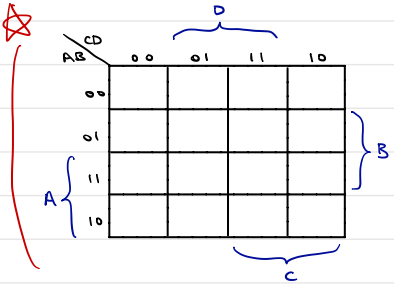


$\bar{A}\bar{B}C + \bar{A}B\bar{C} +$

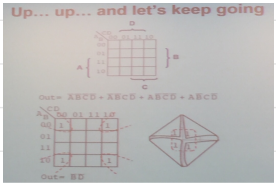
$A\bar{B}\bar{C} + ABC$

* Exception: Cannot simplify

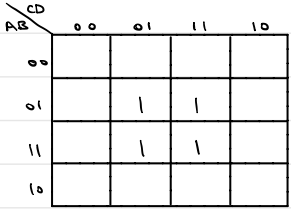
4 variable karnaugh map:



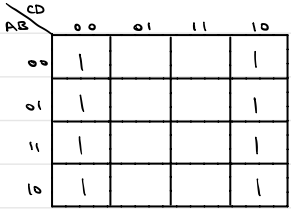
* wraps around top-bottom and left-right to form a torus.



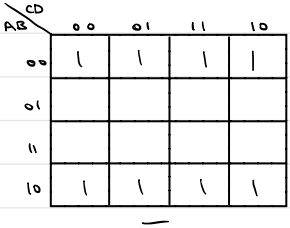
ex)



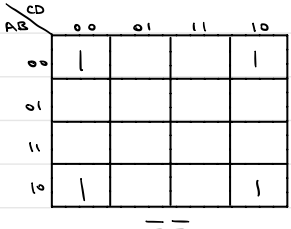
BD



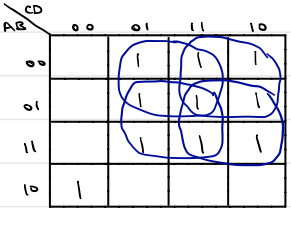
$\bar{D}$



$\bar{B}$



$\bar{B}\bar{D}$



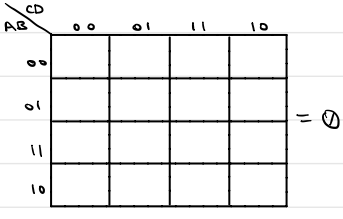
$\bar{A}D + \bar{A}C + BD + BC$

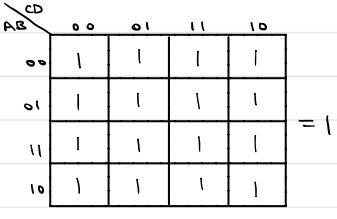
$+ A\bar{B}\bar{C}\bar{D}$

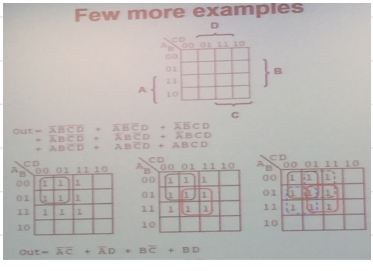
* Gray Binary Code allows the grouping to work!

Steps:

1. Design initial circuit with a truth table.
2. Determine how many inputs, which outputs are 1.
3. Draw a karnaugh map for N inputs.
4. Fill in karnaugh map cells corresponding to outputs == 1
5. Find groups of rows, columns. The larger the group, the simpler the expression.
6. Build simplified expression from karnaugh map groups.
7. Build simplified circuit.







Circuit Adder:

Adds two binary numbers

(insufficient) ex) half adder

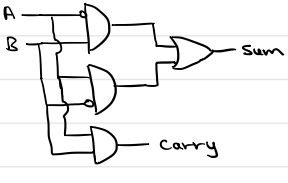
- Half adder - adds two binary bits; outputs are 1 bit sum and 1 bit carry

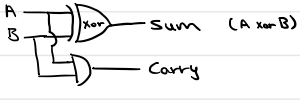
ex) Inputs A B

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sum = $\bar{A}B + A\bar{B}$

Carry = AB





(A xor B)

*Two half adder ≠ full adder

Full adder:

Full adder adds two binary digits (bits) and an input carry. Outputs are Sum and Carry (out).

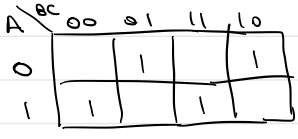
A	B	Carry In	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$Sum = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$

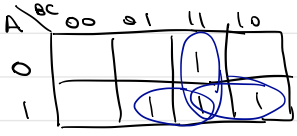
*C for Carry in.

$Carry\ out = ABC + \bar{A}BC + A\bar{B}C + AB\bar{C}$

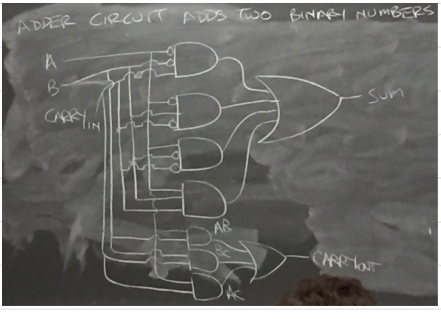
*SOP for Sum, Carry out.

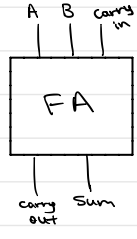


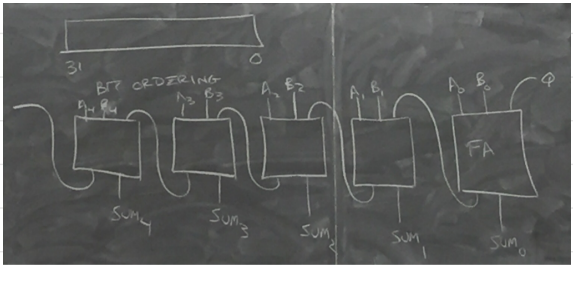
Sum did not simplify



Carry out simplifies to $AB + BC + AC$ from Karnaugh map



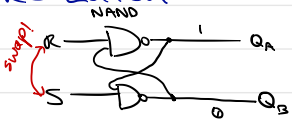




Combinational Circuit: Output depends only on current inputs

Sequential Circuit: output depends on current input signals and a sequence of past input signals.

RS Latch:



NAND = Not AND

*Memory for 1 bit

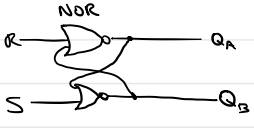
Set

$R=0, S=1 \rightarrow$ Set $Q_A=0, Q_B=1$

$R=1, S=0 \rightarrow$ Set $Q_A=1, Q_B=0$

$R=1, S=1 \rightarrow$ Q_A is held to 0 or 1 *Put in memory

$R=0, S=0 \rightarrow$ Undefined; do not do this!



Negated or gates

*memory for 1 bit

Set

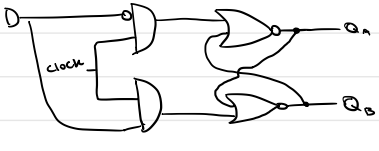
$R=0, S=1 \rightarrow$ Set $Q_A=1, Q_B=0$

$R=1, S=0 \rightarrow$ Set $Q_A=0, Q_B=1$

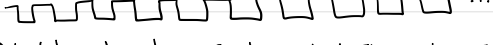
$R=1, S=1 \rightarrow$ Undefined; do not do this!

$R=0, S=0 \rightarrow$ Q_A is held to 0 or 1 *Put in memory

D Latch: Control circuits with clock



- Input D is either 0 or 1
- Q_A, Q_B is either 0,1 or 1,0 never 0,0
- Clock signal: a regular electric signal looks like a square wave



- Output of D Latch only changes when clock signal changes from 0 to 1.

Hardware Memory:

Static Random Access Memory (SRAM)

- Bistable – two stable states
- cell – hardware term for 1 bit
- As long as power on, SRAM will retain its value.
- SRAM is used for cache memory.

SRAM Advantages:

1. SRAM faster than DRAM
2. Structurely more complex
3. More expensive per unit storage than DRAM
4. Does not need periodic refresh.

Dynamic Random Access Memory (DRAM):

- Storing each bit in its own capacitor
- Capacitor ⟷ charge / discharge
- Capacitor gradually lose charge over time → refresh
- Refresh periodically to not lose data
  - Refresh interval (typical): 64 millisec
  - Capacitor Refresh Logic
- DRAM Memory controller handles transfers of w bits to/from memory chip.
- DRAM memory arranged as two-Dimensional array.

DRAM Advantages:

- Structural simplicity
- More bits per unit area

DRAM Organization:

DRAM Notation d x w
- w: word size in bits
- d: # of words on chip

* Cell = 1 bit
* Supercell = word

DRAM memory access takes two signals

- Row access Strobe — select a row
- Column access Strobe — select a column in the row

* Memory Controller translates (linear) main memory address into supercell row, column
* Multiple DRAM chips are stored on memory modules

DRAM example:

- 64 megabyte memory module using eight 64 megabit DRAM chips, each DRAM chips, each DRAM chip 8 meg x 8. Each chip is numbered 0 to 7. Word (supercell) size is 8 bits.
- Two-Dimensional array on each DRAM chip is 4096 x 2048 (Not quite square)

DRAM Main Memory Access:

- DRAM memory Controller translates main memory address into Row I, Column J
- I and J are send to DRAM memory module which addresses the same cell on every trip.
- Circuitry memory module aggregates or distributes bytes into 64 bit transfers to/from memory module intersecting. Adjacent bytes of main memory are actually stored in separate DRAM chips.

Simm - Single Inline Memory Module:

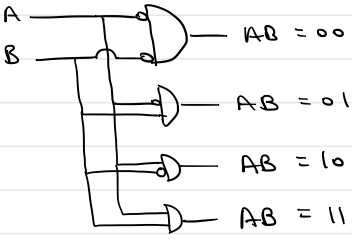
- 72 pins, 32 bit memory transfers
- Originally for 32 bit machines.
- Paired up for 64 bit machines to get 64 bit transfers.

DIMM – Dual Inline Memory Module:

- 168 pins, 64-bit memory transfers
- Replaced paired Simms in 64 bit machines

Access to SRAM memory – Hardware Decoder

- N inputs, 2^N outputs N is 32 for 32 bit machines
- Exactly one output for every possible input pattern



2 bit decoder circuit

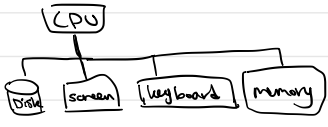
Hardware Interrupts:

Def: An electronic signal sent from hardware devices to the Processor (CPU) to indicate an event that needs attention.

- Happen asynchronously (middle of execution)
- Implemented using signals from external device
- 100 different hardware interrupt signals
- Inform OS with IO request

ex) Press key on keyboard → interrupt

- As processor Fetch–Decode–Execute to see any interrupt occurs
- Interrupt stops FDE cycle
- Invokes interrupt handler → FDE cycle
- Interrupt can happen anytime



Analogy: Prof. Stop when someone raises hand → answer question → continue lecturing

* Process has to finish on all processors.

Polling: CPU go through every single peripherals

- Not too efficient
- Timer interrupts (Not imply program errors)
- OS > kernel

Software interrupts:

- Originate either by an exceptional condition within the processor or by special machine instructions that cause an interrupt when executed

TRAP or Exception

Interrupts vs Signals vs Exceptions

- Signal handler — handle various signals during execution
- CPU respond to hardware interrupt (efficient for CPU to manage multiple peripherals.)
- Can be handled and then resume execution
- Exception
  - return value
  - print out error message
  - send diff. error message for other program to catch
  - stop the program

↳ cannot resume execution

Variadic C functions

A variadic function can take different number of arguments

ex) int printf(const char * FMT, ...)

#include <STDARG.H>

C macros for manipulation of variadic argument lists.

```
VA_List AP;
VA_Start(AP, FMT);
```

↑ instance of va_list

↖ last named arg

```
VA_End(AP)
```

↖ Does necessary clean up

```
VA_ARG(AP, Typename)
```

← a typename

↖ returns a value of type specified in typename arg and advances AP to next unnamed arg

```
#include <STDARG.H>
#include <STDIO.H>
double average (int count, ...) {
  VA_List AP;
  int j;
  double sum = 0.0;
  VA_Start(AP, count);
  for (j=0; j < count; j++) {
    sum += VA_Arg(AP, int);
  }
  VA_END(AP);
  return sum/count;
}
```

elipse

argument pointer

← initialize AP

↖ last named arg

It is also possible to manipulate the entire set of variadic arguments all at once.

```
void error(const char *fmt, ...){
    va_list args;
    va_start(args, fmt);
    printf("Error");
    vprintf(fmt, args);
    print("\n");
    va_end(args);
}
```

vprintf()
vfprintf()
vsprintf()

Three ways to know the size of the variadic argument list.

1. Pass the number of variadic args (see prev. ex.)
2. Pass some special sentinal value
3. Use a pattern in one arg that matches a variadic arg to something in the pattern (e.g. scanf() / printf() format specifiers)

*can infer type of variadic arg from format specifier.