CS2100 Notes

L3: Number System

$(a_n a_{n-1} ... a_0 . f_1 f_2 ... f_n)_{10} = (a \times 10^n) + (a_{n-1} \times 10^{n-1}) ... + (a_0 \times 10^0) + (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + ... + (f_m \times 10^{-m})$

In programming language C / QTSpim

- Prefix **0** for octal. Eg: 032 represents the octal number $(32)_8$
- Prefix **0x** for hexadecimal. Eg: 0x32 represents the hexadecimal number $(32)_{16}$
- Prefix **0b** for binary

Convert whole number to binary

- use successive division by 2 until the quotient is 0.

$(43)_{10} = (\ 101011\ )_2$

2	43	
2	21 rem 1	← LSB
2	10 rem 1	
2	5 rem 0	
2	2 rem 1	
2	1 rem 0	
	0 rem 1	← MSB

- Decimal whole number to base-R: repeated division by R

Convert decimal fractions to binary

- use successive multiplication by 2, until the fractional product is 0 (or until the desired number of decimal places).

$(0.3125)_{10} = (\ .0101\ )_2$

	Carry	
0.3125×2=0.625	0	←MSB
0.625×2=1.25	1	
0.25×2=0.50	0	
0.5×2=1.00	1	←LSB

- Decimal fractions to base-R: repeated multiplication by R

Binary to Octal/Hexadecimal Conversion

- Binary → Octal: partition in groups of 3
  - (10 111 011 001 . 101 110)2 = (2731.56)8
- Octal → Binary: reverse
- Binary → Hexadecimal

Negative Numbers

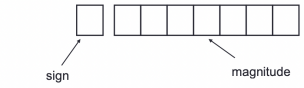
- unsigned: only non-negative values

- signed: both positive and negative
- 3 common representations for signed binary numbers
  1. Sign-and-Magnitude
  2. 1s Complement
  3. 2s Complement

1. Sign-and-Magnitude

- 0 for +
- 1 for -
- **to negate the value: invert sign bit**
- largest value: $01111111 = +127_{10}$
- smallest value: $11111111 = -127_{10}$
- zeros:
  - $00000000 = +0_{10}$
  - $10000000 = -0_{10}$
- range (8-bit): -127_{10} to $+127_{10}$

Eg: a 1-bit sign and 7-bit magnitude format.



- 00110100 ➔ $+110100_2$ = $+52_{10}$
- 10010011 ➔ -10011_2 = -19_{10}

2. 1s Complement

- For n-bit binary number x, $-x = 2^n - x - 1$
- **to negate a value: invert all the bits**
- largest value: $01111111 = +127_{10}$
- smallest value: $10000000 = -127_{10}$
- zeros:
  - $00000000 = +0_{10}$
  - $11111111 = -0_{10}$
- range (8-bit): -127_{10} to $+127_{10}$
- range (n-bit): $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- MSB still represents the sign:
  - 0 for + and 1 for -

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:
  -00001100_2 = $2^8 - 12 - 1$ (calculation done in decimal)
  = 243
  = 11110011_{1s}

(This means that -12_{10} is written as 11110011 in 1s-complement representation.)

1s Complement on Addition/Subtraction

A+B

1. Perform binary addition on the two numbers.
2. If there is a carry out of the MSB, add 1 to the result.
3. Check for overflow. Overflow occurs if result is opposite sign of A and B.

A-B = A+(-B)

1. Take 1s-complement of B.
2. Add the 1s-complement of B to A.

3. 2s Complement

- For n-bit binary number x: $-x = 2^n - x$
- **to negate a value: invert all bits and +1**
- largest value: $01111111 = +127_{10}$
- smallest value: $11111111 = -127_{10}$
- zeros: $00000000 = +0_{10}$
- range (8-bit): -128_{10} to $+127_{10}$
- range (n-bit): $-(2^{n-1})$ to $2^{n-1} - 1$
- MSB still represents the sign:
  - 0 for + and 1 for -
- leftmost digit has a weight of -2^{n-1}
- **NOT TRUE** for other N's complement

- Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 2s-complement is:

  -00001100_2 $= 2^8 - 12$ (calculation done in decimal)
  $= 244$
  $= 11110100_{2s}$

  (This means that -12_{10} is written as 11110100 in 2s-complement representation.)

$$= -2^7 + 2^6 + 2^5 + 2^4 + 2^2$$

Taking the 10's complement of 1 (i.e. finding -1), we have $104^{-1} = 9999 \neq -9 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 9 = -8001$ which is not equal to -1.

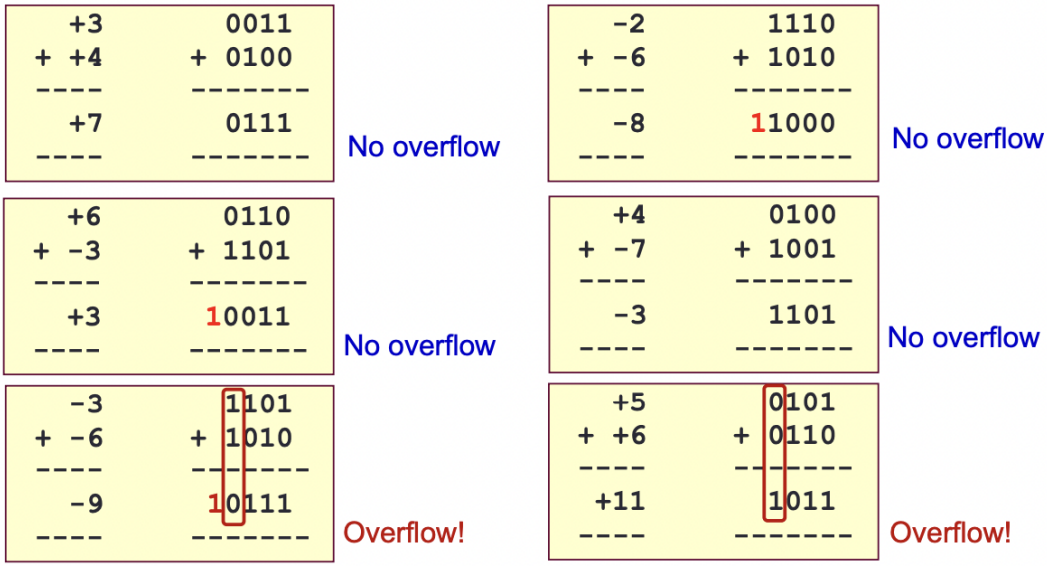
2s Complement on Addition/Subtraction

A+B

1. Perform binary addition
2. Ignore the carry out of the MSB
3. Check for overflow
    - Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.

A-B = A+(-B)

1. Take 2s-complement of B
2. Add the 2s-complement of B to A



4-bit system

Positive values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

PastYearQn.c

```
int i, n = 2147483640;
for (i=1; i<=10; i++) {
    n = n + 1;
}
printf("n = %d\n", n);
```

- What is the output of the above code when run on sunfire?
- Is it 2147483650? ✘

- int type in sunfire takes up 4 bytes (32 bits) and uses 2s complement
- Largest positive integer = $2^{31} - 1 = 2147483647$

1st iteration: n = 2147483641

7th iteration: n = 2147483647

```
01111 ……. 1111111111
                  + 1
10000…….0000000000
```

8th iteration: n = -2147483648

9th iteration: n = -2147483647

10th iteration: n = -2147483646

Excess Representation

- allows the range of values to be distributed evenly between the positive and negative values, by a simple translation (addition/subtraction)
- excess-8 representation to value = binary value in ex-8 - 8
- e.g. For 4-bit numbers, we may use excess-7 or excess-8 (2^4 = 16; 16 / 2 = 8)

Excess-8 Representation	*Value*
0000	-8
0001	-7
0010	-6
0011	-5
0100	-4
0101	-3
0110	-2
0111	-1

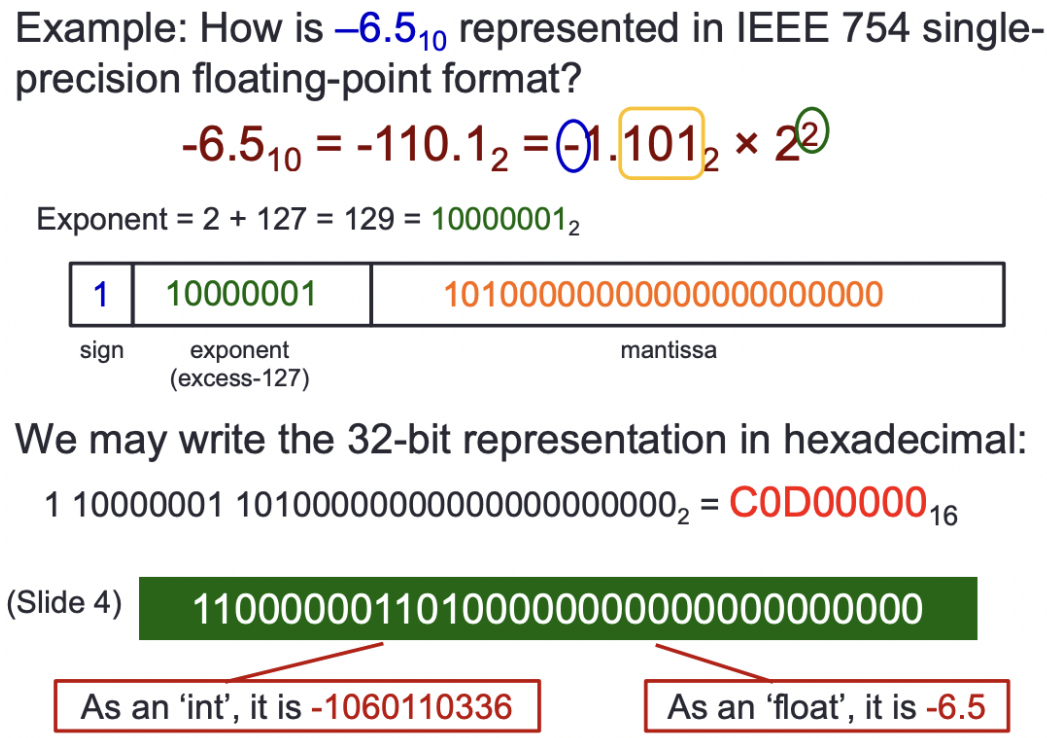
Excess-8 Representation	*Value*
1000	0
1001	1
1010	2
1011	3
1100	4
1101	5
1110	6
1111	7

Fixed-Point Representation

- In fixed-point representation, the number of bits allocated for the whole number part and fractional part are fixed.
- e.g. 8-bit: 6 bits whole number, 2 bits fractional parts

Floating-Point Representation (IEEE 754)

- 3 components: sign, exponent and mantissa (fraction)
- 2 formats:
  1. Single-precision (32 bits): 1-bit sign, **8-bit exponent** with bias 127 (excess-127), **23-bit mantissa**
  2. Double-precision (64 bits): 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), and 52-bit mantissa
- Sign bit: 0 for + and 1 for -
- Mantissa is normalised with an implicit leading bit 1 (this is known as the hidden bit)
  - 111.1_2 → normalised → $1.101_2 \times 2^2$ → only 101 is stored in the mantissa field
  - 0.00101101_2 → normalised → $1.01101_2 \times 2^{-3}$ → only 01101 is stored in the mantissa field
  - if qns states "no hidden bit", it means the leading 1 is also stored (20/21 S2 Midterm Q5)



Disadvantage:

- The leftmost mantissa bit is always 1 because it is normalized, hence it is not possible to represent 0.
- More complex representation

Advantage:

- Much better range and accuracy.

L4: Pointers and Functions

- `%p` is used as the format specifier for addresses
- addresses are printed out in hexadecimal format
- address of a var varies from run to run
- `*ptr` is used to dereference the content of the pointer

Declare a Pointer

```
type *pointer_name
```

- e.g. `int a = 123; int *a_ptr = &a`
- `*a_ptr` is synonymous with `a`

Calling Functions

`<stdio.h>` - `scanf()` and `printf()`

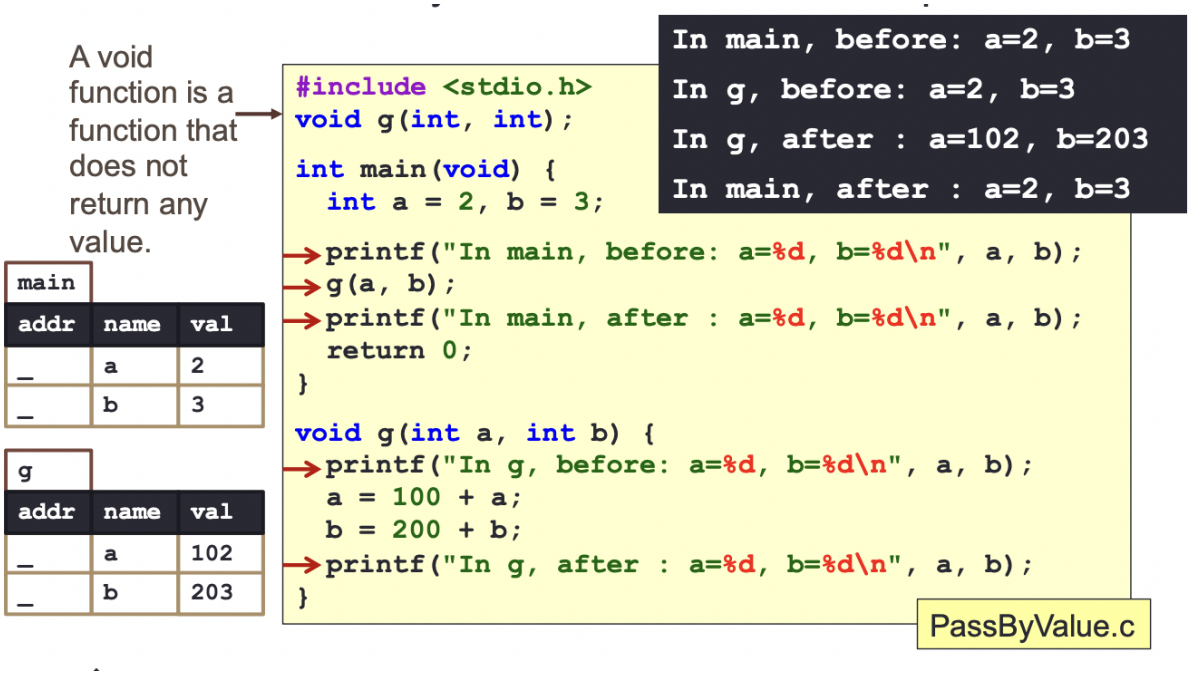
`gcc -lm MathFunctions.c` - link to Math library

`gcc -o lab1 lab1.c` - output file to `lab1` instead of just `a.out`

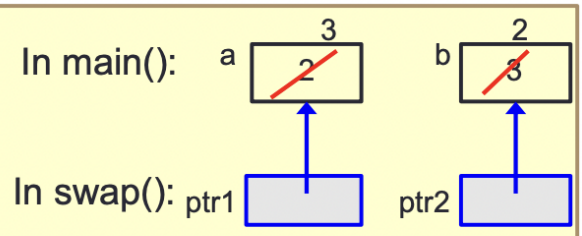
`./a.out` - run program in `a.out`

User-Defined Functions

- put function prototypes at the top of the program, before the main() function, to inform the compiler of the functions that your program may use and their return types and parameter types.
- A function prototype includes only the function's return type, the function's name, and the data types of the parameters (names of parameters are optional).
- Function definitions to follow after the main() function.



```c
#include <stdio.h>
void swap(int *, int *);
int main(void) {
   int a, b;
   printf("Enter two integers: ");
   scanf("%d %d", &var1, &var2);
   swap( &a, &b );
   printf("var1 = %d; var2 = %d\n", var1, var2);
   return 0;
}
void swap(int *ptr1, int *ptr2) {
   int temp;
   temp = *ptr1; *ptr1 = *ptr2; *ptr2 = temp;
}
```



SwapCorrect.c

Array

- Address: `arr` or `&arr[0]` (both gives the address of the first element)
- Array content/element: `arr[0]` or `*arr` (both gives the first element)
- good habit to define array length at the top

ArraySumV1.c

```c
#include <stdio.h>
#define MAX 5

int main(void) {
  int numbers[MAX];
  int i, sum = 0;

  printf("Enter %d integers: ", MAX);
  for (i=0; i<MAX; i++) {
    scanf("%d", &numbers[i]);
  }

  for (i=0; i<MAX; i++) {
    sum += numbers[i];
  }

  printf("Sum = %d\n", sum);
  return 0;
}
```

- Summing all elements in an integer array

ArraySumV2.c

```c
#include <stdio.h>
#define MAX 5

int main(void) {
  int numbers[MAX] = {4,12,-3,7,6};
  int i, sum = 0;

  for (i=0; i<MAX; i++) {
    sum += numbers[i];
  }

  printf("Sum = %d\n", sum);
  return 0;
}
```

```
// a[0] = 54, a[1] = 9, a[2] = 10
int a[3] = {54, 9, 10}

// size of b is 3
int b[] = {1, 2, 3};

// c[0] = 17, c[1] = 3, c[2] = 10, c[3] = 0, c[4] = 0
// when fewer initial values are provided, unfilled elements becomes 0
int c[5] = {17, 3, 10};

/* INCORRECT INITIALIZATIONS */
int e[2] = {1, 2, 3};

int f[5];
f[5] = {8, 23, 12, -3, 6} // compilation err
```

- An array name is a fixed (constant) pointer; it points to the first element of the array, and this cannot be altered.

```
#define N 10

int source[N] = {10, 20, 30, 40, 50};
int dest[N];
dest = source; // ILLEGAL

/********** CORRECT **************/
#define N 10

int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
int i;

for (i = 0; i < N; i++) {
   dest[i] = source[i];
}
// There is another method
// use the <string.h> library function memcpy()
```

Array Function Prototype

```
int sumArr(int [], int);
int sumArr(int arr[], int size); // equivalent
int sumArr(int arr[8], int size); // "8" will be ignored
int sumArr(int *, int); // arr is a pointer

// alternative actual function definition
```

```
int sumArr(int *arr, int size) {...}
*arr = // operation on *arr not arr[index]

// since arr is a pointer -> pass by address & allow modification of elements
```

String

- A string is an array of characters terminated by '\0'

`include <string.h>`

- `strlen(s)`: returns the num of characters in `s`
- `strcmp(s1, s2)`:
  - Compare the ASCII values of the corresponding characters in strings `s1` and `s2`.
  - Return
    - a negative integer if `s1` is lexicographically less than `s2`, or
    - a positive integer if `s1` is lexicographically greater than `s2`, or
    - 0 if `s1` and `s2` are equal.
- `strcmp(s1, s2, n)` : compare first `n` characters of `s1` & `s2`
- `strcpy(s1, s2)`: copy `s2` to `s1`
  - string is an array so cannot just initialise and then `name = "Mattew"`
  - If the string to be copied is too long → only fills in the given size
- `strcpy(s1, s2, n)`: copy the first `n` characters of string pointed to by `s2` to `s1`

```
char str[6];
str[0] = 'e';
str[1] = 'g';
str[2] = 'g';
str[3] = '\0';

char fruit_name[] = "apple";
char fruit_name[] = {'a', 'p', 'p', 'l', 'e'};
```

IO

```
/**** READ STRING FROM STDIN (KEYBOARD) ****/
fgets(str, size, stdin) // reads size - 1 char, or until new line
scanf("%s", str); // reads until white space

/**** PRINT STRING TO STDOUT (MONITOR) ****/
puts(str); // terminates with new line
printf("%s\n", str);

// fgets() also reads in the new line character -> to replace with '\0'
fgets(str, size, stdin);
len = strlen(str);
if (str[len – 1] == '\n')
  str[len – 1] = '\0';
```

Structure Types

- no memory is allocated to a type

```
// to be placed before function prototypes but
// after preprocessor directives (imports and const def)
typedef struct {
  int length, width, height;
} box_t; // creates a new type called box_t

/***** METHOD 1 *****/
box_t box1 = {1, 2, 3};
// initialise inside any function

/***** METHOD 2 *****/
box_t box2;
box2.length = 1;
box2.width = 2;
box3.height = 3;


card_t card2 = { 666666, {30, 6} };
card2.expiryDate.year = 2021;

// pass by value,
// need specify the address if want to modify content
```

Arrow Operator

```
// NOT *player_ptr.name: . has a higher precedence
(*player_ptr).name;
player_ptr->name // equivalent
```

L7-9: MIPS

- 1 byte is 8-bit
- each memory location holds 1 byte
- address 0x10 → 0x14 = 4 byte diff

number of bits taken up by a register = $\lceil log_2(max\ num\ of\ registers) \rceil$

number of bits taken up by an address = $\lceil log_2(num\ of\ addresses) \rceil$

labels for jump and branches are not instructions

- `bgt` = `slt` + `beq` or `bne`

Word alignment

- for `sw` and `lw`: to ensure word-alignment for array (4-byte element), index * 4 to get immediate value

MIPS disallows loading/storing unaligned word using **lw**/**sw**:

- Pseudo-Instructions ***unaligned load word*** (**ulw**) and ***unaligned store word*** (**usw**) are provided for this purpose
- **lh** and **sh**: load halfword and store halfword
- **lwl**, **lwr**, **swl**, **swr**: load word left / right, store word left / right.

Address of A[] ➔ $t0 Result ➔ $t8 i ➔ $t1	Comments

addi $t8, $zero, 0
addi $t1, $zero, 0
addi $t2, $zero, 40
loop: bge $t1, $t2, end
sll $t3, $t1, 2
add $t4, $t0, $t3
lw $t5, 0($t4)
bne $t5, $zero, skip
addi $t8, $t8, 1
skip: addi $t1, $t1, 1
j loop
end:
	
	# end point
	
	# i * 4
	# &A[i]
	# $t3 ← A[i]
	
	# result++
	# i++

Address of A[] ➔ $t0 Result ➔ $t8 &A[i] ➔ $t1	Comments

addi $t8, $zero, 0
addi $t1, $t0, 0
addi $t2, $t0, 160
loop: bge $t1, $t2, end
lw $t3, 0($t1)
bne $t3, $zero, skip
addi $t8, $t8, 1
skip: addi $t1, $t1, 4
j loop
end:
	
	# addr of current item
	# &A[40]
	# comparing address!
	# $t3 ← A[i]
	
	# result++
	# move to next item

Branch

PC = PC + 4 + (immediate * 4) used for branch instructions

- immediate is positive when moving down (forward), negative when moving up (backward)

```
Loop:  beq  $9, $0, End  # rlt addr: 0
       add  $8, $8, $10  # rlt addr: 4
       addi $9, $9, -1   # rlt addr: 8
       j    Loop         # rlt addr: 12
End:                     # rlt addr: 16
```

Field representation in decimal:

opcode	rs	rt	immediate
4	9	0	3

Field representation in binary:

000100	01001	00000	0000000000000011

```
Loop:  beq  $9, $0, End  # rlt addr: 0
       add  $8, $8, $10  # rlt addr: 4
       addi $9, $9, -1   # rlt addr: 8
       beq  $0, $0, Loop # rlt addr: 12
End:                     # rlt addr: 16
```

immediate = -4

Jump

jump instruction → use absolute address (32-bit) and remove first 4 and last 2 (26-bit)

```
Loop:  beq  $9, $0, End  # addr:  8   ← jump target
       add  $8, $8, $10  # addr: 12
       addi $9, $9, -1   # addr: 16
       j    Loop         # addr: 20   ← PC
End:                     # addr: 24
```

PC: 0000.....10100

Address to jump to: 0000.....01000 (rep 8)

immediate = **2** (26 bits)

Check your understanding by constructing the new PC value

opcode	target address
000010	00000000000000000000000010

Operation	Opcode in MIPS	Immediate Version (if applicable)
Addition	`add $s0, $s1, $s2`	`addi $s0, $s1,` $C16_{2s}$ $C16_{2s}$ is [-2^{15} to $2^{15}-1$]
Subtraction	`sub $s0, $s1, $s2`	
Shift left logical	`sll $s0, $s1, C5` `C5` is [0 to 2^5-1]	
Shift right logical	`srl $s0, $s1, C5`	
AND bitwise	`and $s0, $s1, $s2`	`andi $s0, $s1, C16` `C16` is a 16-bit pattern
OR bitwise	`or $s0, $s1, $s2`	`ori $s0, $s1, C16`
NOR bitwise	`nor $s0, $s1, $s2`	
XOR bitwise	`xor $s0, $s1, $s2`	`xori $s0, $s1, C16`

Category	Instruction	Example	Meaning	Comments
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Word from register to memory
Data transfer	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = `$s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	`$s1` = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional branch	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

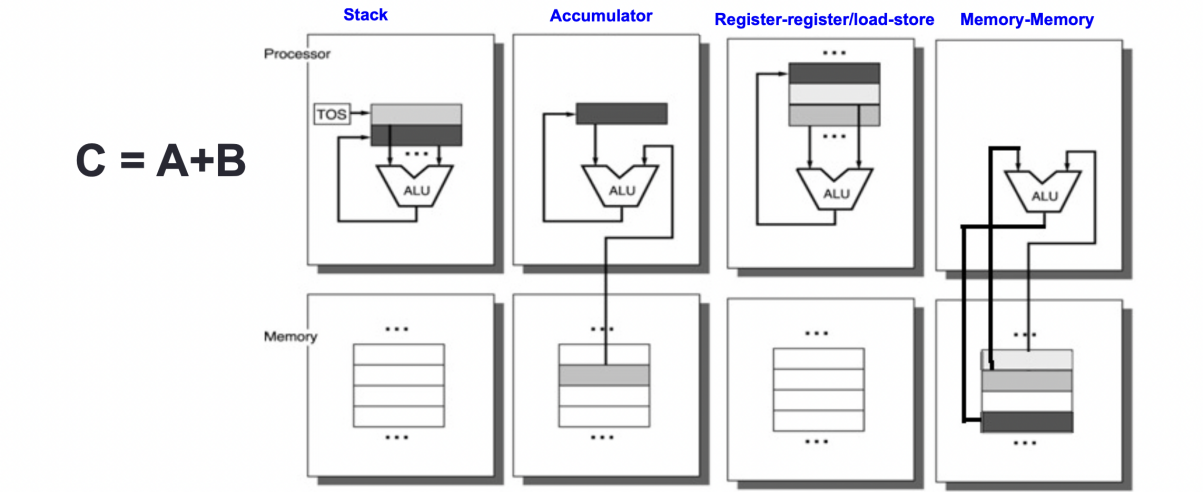
L10: ISA

Typical type and size:

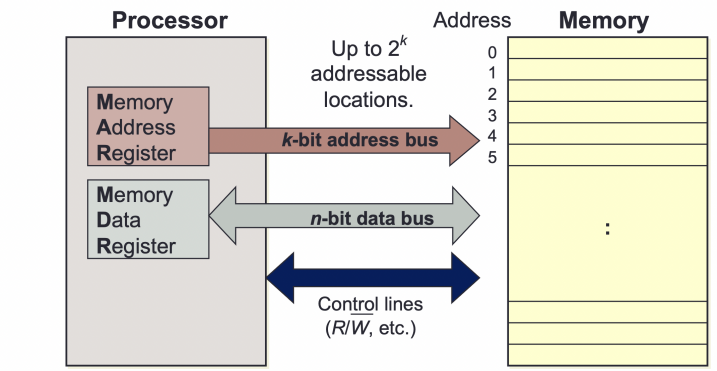
- Character (8 bits),

- half-word (eg: 16 bits),
- word (eg: 32 bits),
- single-precision floating point (eg: 1 word),
- double-precision floating point (eg: 2 words).

Stack	Accumulator	Register (load-store)	Memory-Memory
`Push A`	`Load A`	`Load R1,A`	`Add C, A, B`
`Push B`	`Add B`	`Load R2,B`	
`Add`	`Store C`	`Add R3,R1,R2`	
`Pop C`		`Store R3,C`	



- Given *k*-bit address, the address space is of size 2^k
- Each memory transfer consists of one word of ***n*** bits



- 32-bit processor
  - one word is 32-bit
  - 4 BYTES
- 24-bit processor
  - one word is 24-bit
  - 3 BYTES

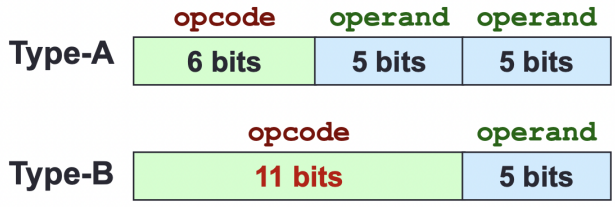
Endianness

- the relative ordering of the bytes in a multiple-byte word stored in memory

Big-endian:	Little-endian:
Most significant byte stored in lowest address. Example: IBM 360/370, Motorola 68000, MIPS (Silicon Graphics), SPARC.	Least significant byte stored in lowest address. Example: Intel 80x86, DEC VAX, DEC Alpha.
Example: **0xDE AD BE EF** Stored as: 0 DE 1 AD 2 BE 3 EF	Example: **0xDE AD BE EF** Stored as: 0 EF 1 BE 2 AD 3 DE

Maximum and Minimum no. of instructions

- What is the maximum number of instructions?



Answer:

$1 + (2^6 - 1) \times 2^5$

$= 1 + 63 \times 32$

$= \mathbf{2017}$

- **Reasoning:**
  1. For every 6-bit prefix (front-part) given to Type-B, we get 2^5 unique patterns, e.g. `[111111]XXXXX`
  2. So, we should minimize Type-A instruction and give as many 6-bit prefixes as possible to Type-B

     ➔ 1 Type-A instruction, $2^6 - 1$ prefixes for Type-B

- Design an expanding opcode for the following to be encoded in a 36-bit instruction format. An address takes up 15 bits and a register number 3 bits.
  - 7 instructions with two addresses and one register number.
  - 500 instructions with one address and one register number.
  - 50 instructions with no address or register.

One possible answer:

3 bits	15 bits	15 bits	3 bits
000 → 110 opcode	address	address	register
111	000000 + 9 bits opcode	address	register
111	000001 : 110010 + 9 0s opcode	unused	unused

▼ A certain machine has 12-bit instructions and 4-bit addresses. Some instructions have one address and others have two. Both types of instructions exist in the machine. What is the maximum number of instructions with **one** address (not total)?

$$(2^4 - 1)(2^4) = 2^8 - 2^4 = 240$$

if asked for total max ⇒ 240 + 1 = 241

▼ A certain machine has 12-bit instructions and 4-bit addresses. Some instructions have one address and others have two. Both types of instructions exist in the machine. What is the minimum total number of instructions, assuming the encoding space is completely utilised (that is, no more instructions can be accommodated)?

$$2^4 - 1 + 2^4 = 31$$

An ISA has 16-bit instructions and 5-bit addresses. There are two classes of instructions:
class *A* instructions have one address, while class *B* instructions have two addresses. Both
classes exist and the encoding space for opcode is completely utilised.

▼ (a) What is the minimum total number of instructions?

Class *A*: `ppppppppppp xxxxx`
Class *B*: `qqqqqq xxxxx yyyyy`
`ppppppppppp` and `qqqqqq` are opcodes, `xxxxx` and `yyyyy` are addresses.

- class *B* $(2^6 - 1)$ opcodes, leaving one 6-bit opcode as the prefix for class *A* 11-bit opcodes.
- class *A* has (2^5) opcodes.
- $Total = (2^6 - 1) + (2^5) = 95$

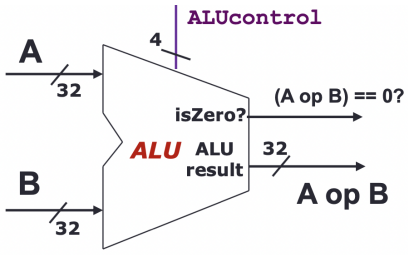
▼ (b) What is the maximum total number of instructions?

- give class B only 1 opcode, leaving $(2^6 - 1)$ prefixes for class A opcodes.
- class A has $(2^6 - 1)(2^5) = 2016$ opcodes.
- $Total = 2016 + 1 = 2017$

L11: Datapath

Arithmetic Logic Unit (ALU)

- **ALU (Arithmetic Logic Unit)**
  - Combinational logic to implement arithmetic and logical operations
- **Inputs:**
  - Two 32-bit numbers
- **Control:**
  - 4-bit to decide the particular operation
- **Outputs:**
  - Result of arithmetic/logical operation
  - A 1-bit signal to indicate whether result is zero



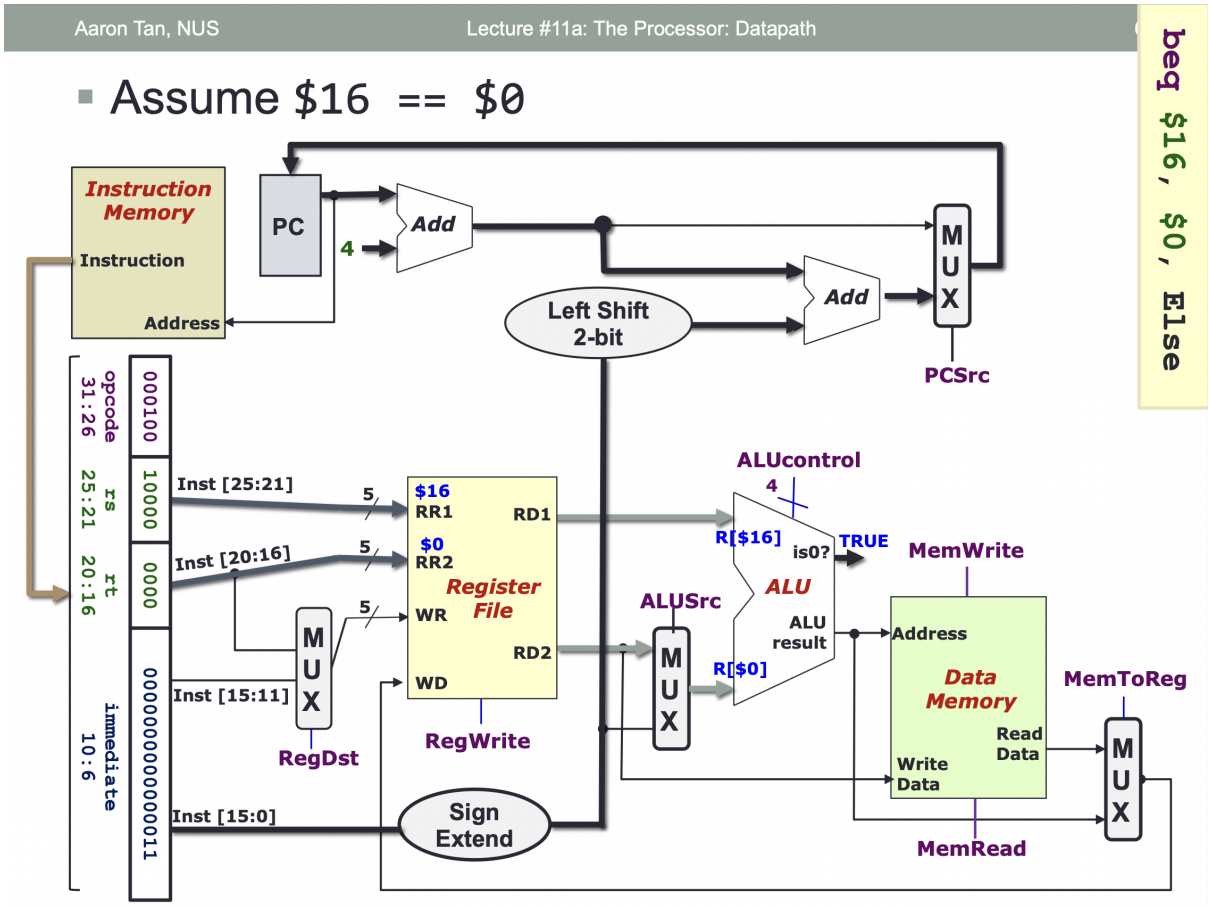
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Memory

- WR is from `Mem(...)` if `MemToReg` is 1 (i.e. use `ReadData`)

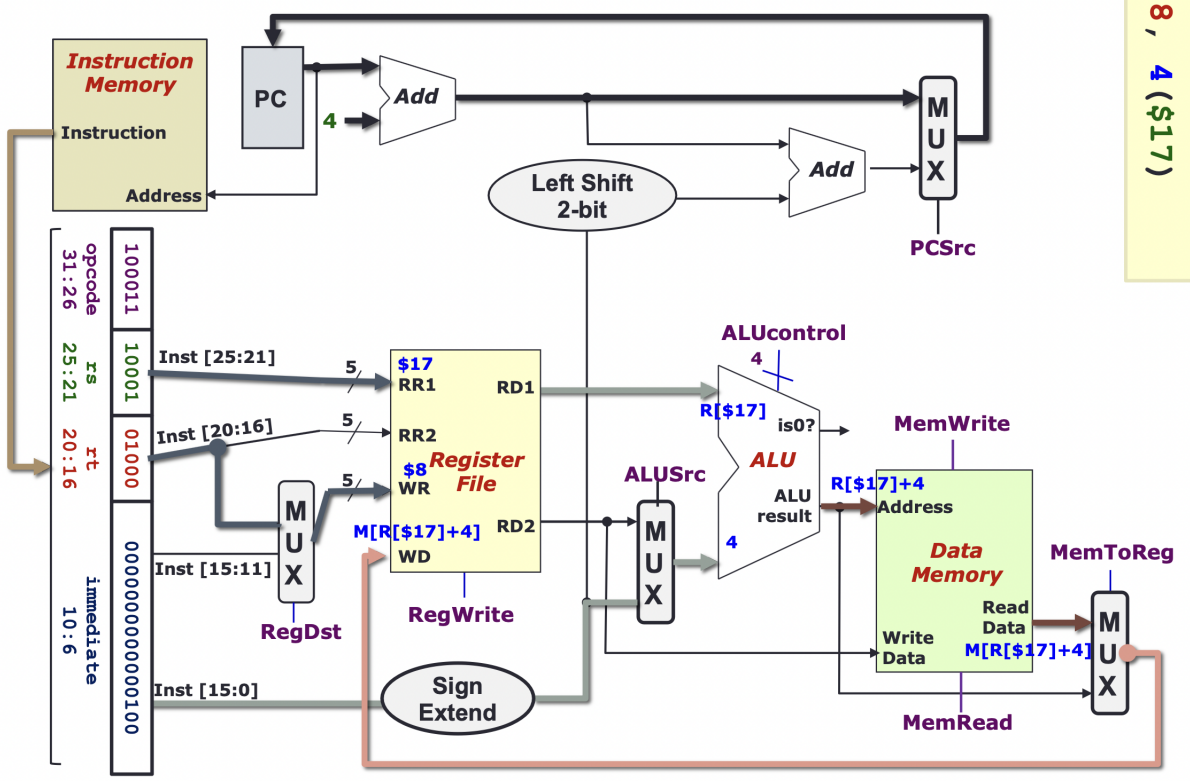
Instruction Memory Access Stage

- Only the `load` and `store` instructions need to perform operation in this stage:
  - Use memory address calculated by ALU Stage
  - Read from or write to data memory
- All other instructions remain idle
  - Result from ALU Stage will pass through to be used in Register Write stage (see section 5.5) if applicable
- Input from previous stage (**ALU**):
  - Computation result to be used as memory address (if applicable)
- **Output to the next stage (Register Write):**
  - Result to be stored (if applicable)



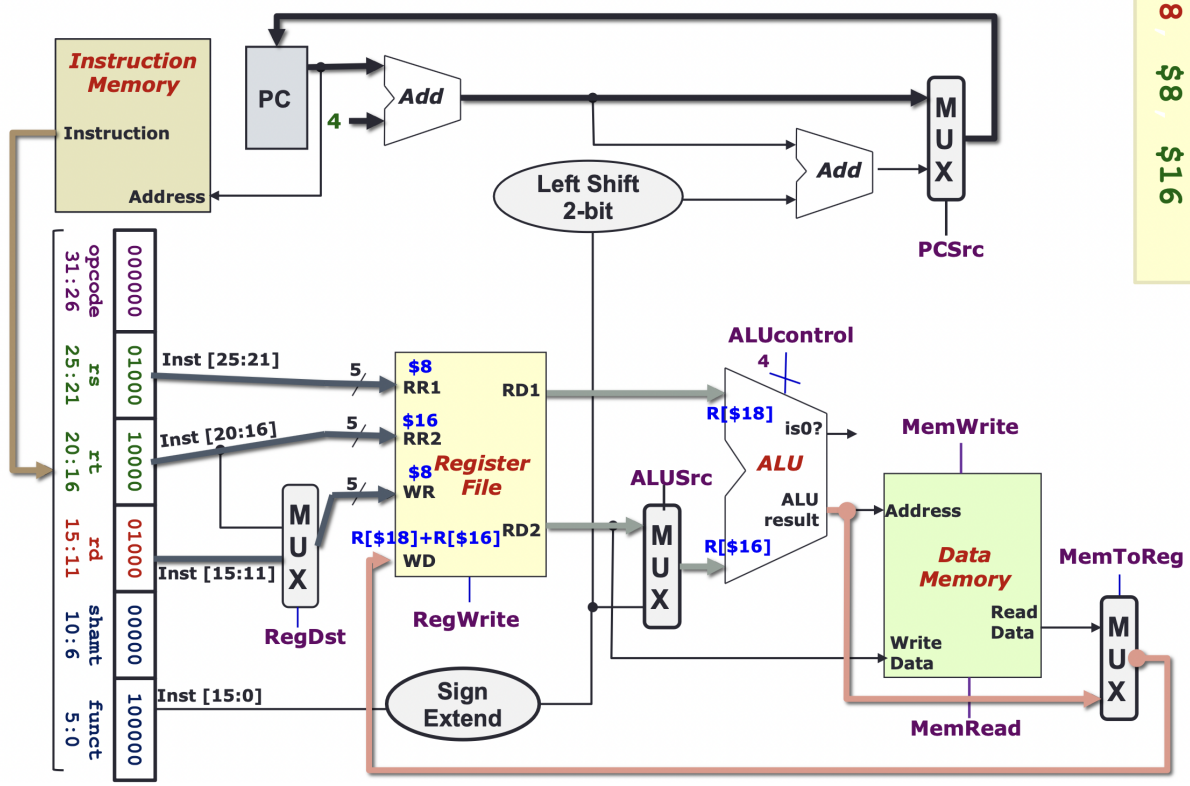
lw $8, 4($17)

- Assume $16 != $0



add $8, $8, $16

- Assume $16 != $0



L12: Control

Control Signals

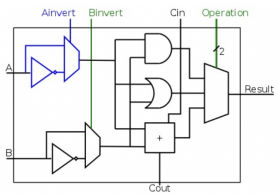
Control signal	False (0)	True (1)	Purpose	Execution Stage
RegDst (top 0, bottom 1)	Write Register (WR) = inst[20:16]	WR = inst[15:11]	Select the destination register number	Decode / Operand fetch
RegWrite	no register write	new value to be written to register	enable writing of register	Decode / Operand fetch
ALUSrc	operand 2 = register read data 2 (RD2)	operand 2 = SignExt(Inst[15:0])	select the 2nd operand for ALU	ALU
ALUControl			select the operation to be performed	ALU
MemRead	not performing memory read access	read memory using address	enable reading/writing of data memory	Memory
MemWrite	not performing memory write operation	memory[Address] ← RD2		
MemToReg (top 1, bottom 0)	WD = ALU result	register write data (WD) = memory read data	select the result to be written back to register file	RegWrite
PCSrc	next PC = PC + 4	next PC = SignExt(Inst[15:0]) << 2 + (PC+4)	select the next PC value	Memory / RegWrite

Design ALU Control

Simplified 1-bit MIPS ALU

4 control bits are needed:

- **Ainvert**:
  - 1 to invert input A
- **Binvert:**
  - 1 to invert input B
- **Operation** (2-bit)
  - To select one of the 3 results

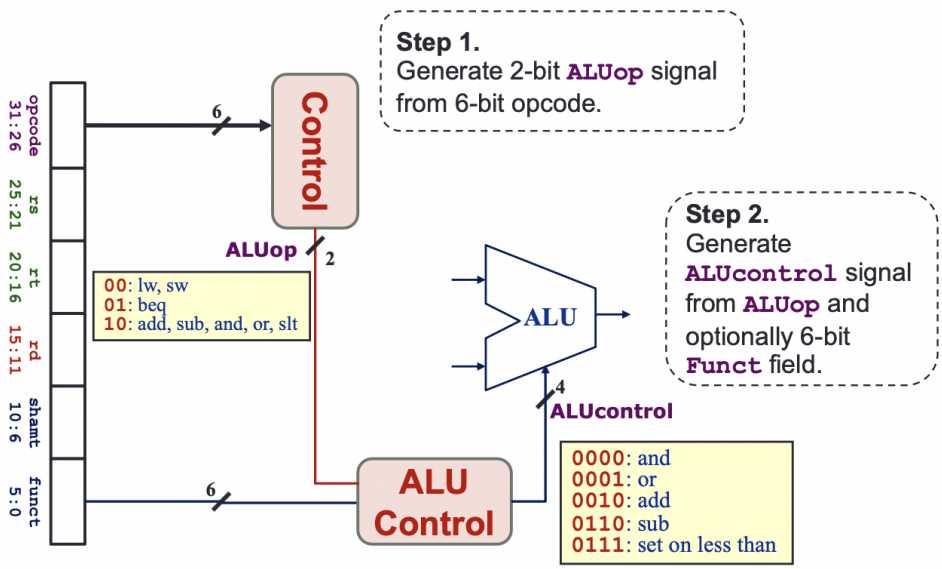


ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

Intermediate Signal: `ALUop`

Basic idea:

1. Use Opcode to generate a 2-bit `ALUop` signal to represent classification of the instructions
2. Use `ALUop` signal and Function Code field (for R-type instructions) to generate the 4-bit `ALUcontrol` signal



Instruction type	ALUop
lw / sw	00
beq	01
R-type	10

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Generation of 2-bit **ALUop** signal will be discussed later

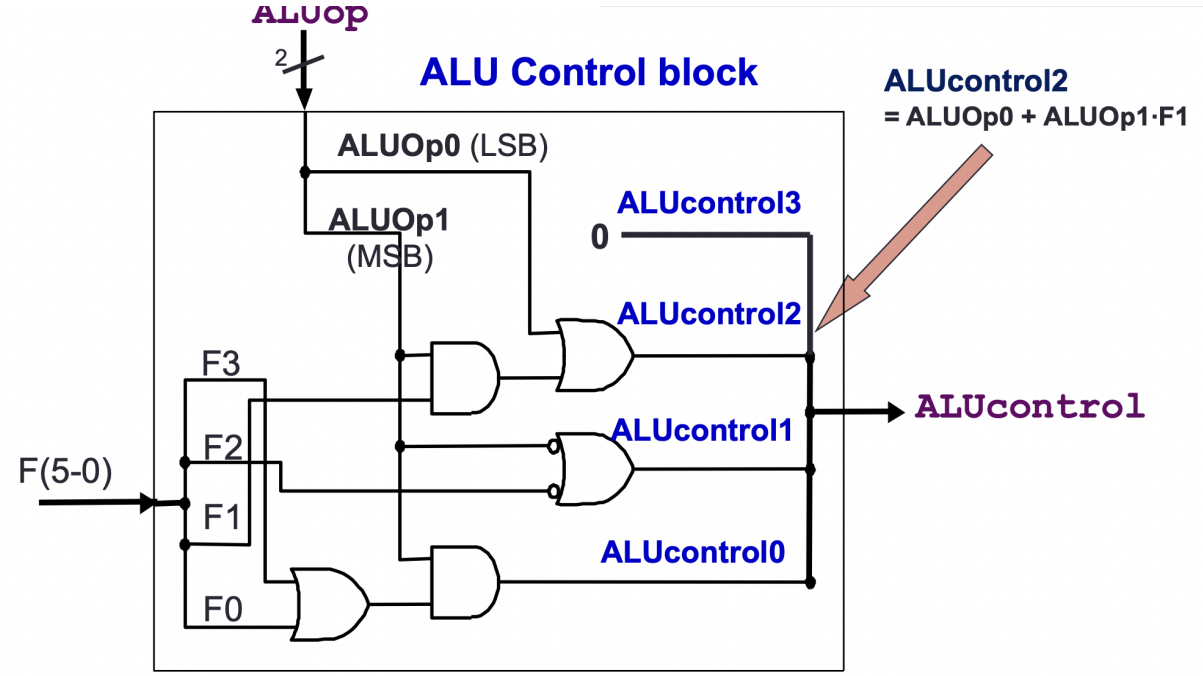
- Input: 6-bit **Funct** field and 2-bit **ALUop**
- Output: 4-bit **ALUcontrol**
- Find the simplified expressions

ALUcontrol3 = 0

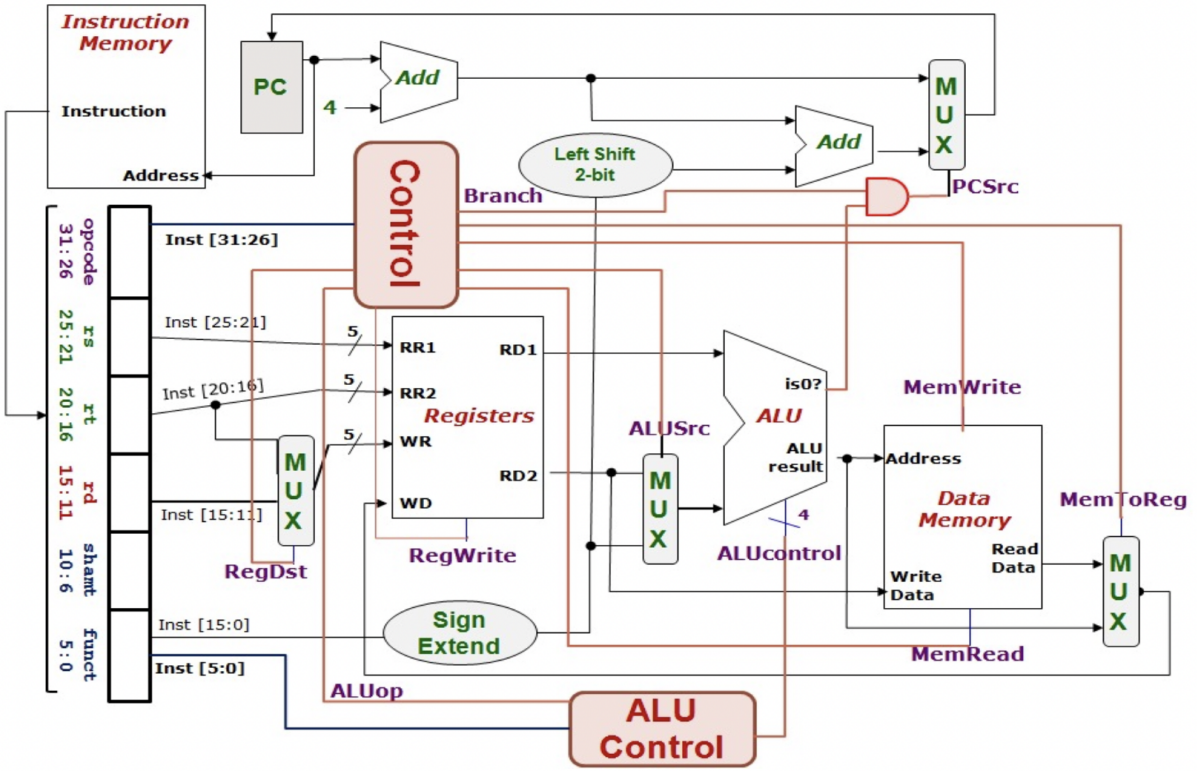
ALUcontrol2 = ?

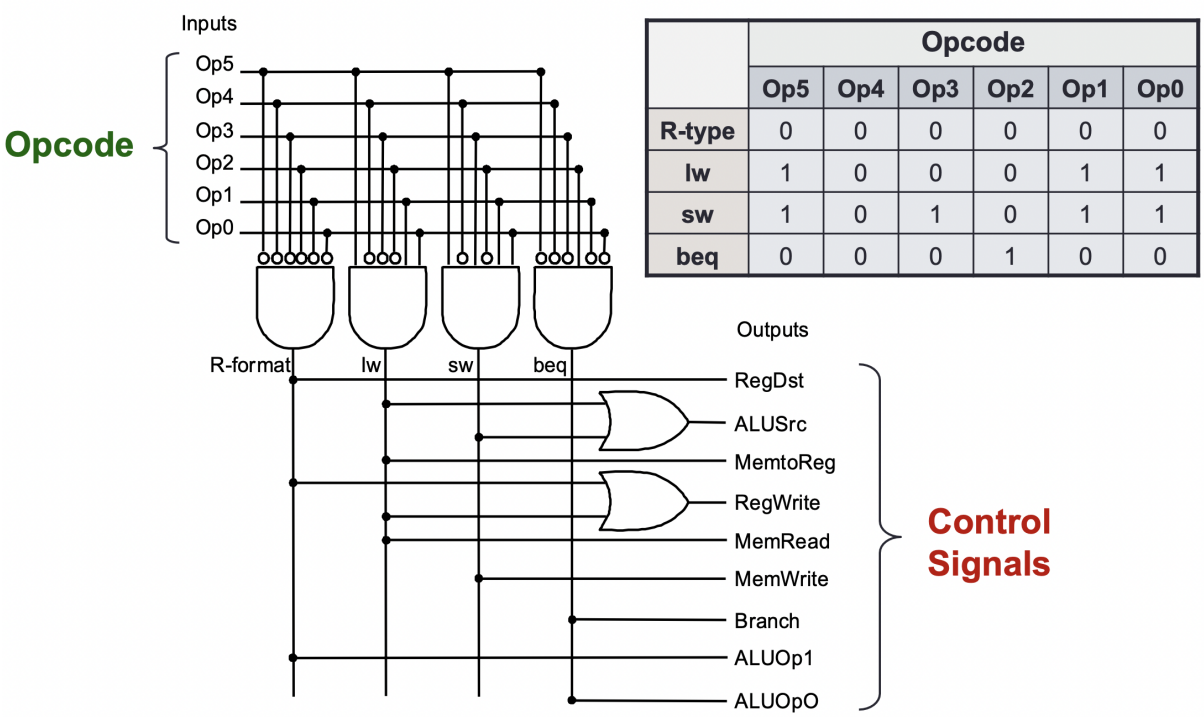
ALUop0 + ALUop1· F1

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0 X~~	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0 X~~	~~1 X~~	~~0 X~~	0	0	0	0	0 0 1 0
sub	1	~~0 X~~	~~1 X~~	~~0 X~~	0	0	1	0	0 1 1 0
and	1	~~0 X~~	~~1 X~~	~~0 X~~	0	1	0	0	0 0 0 0
or	1	~~0 X~~	~~1 X~~	~~0 X~~	0	1	0	1	0 0 0 1
slt	1	~~0 X~~	~~1 X~~	~~0 X~~	1	0	1	0	0 1 1 1



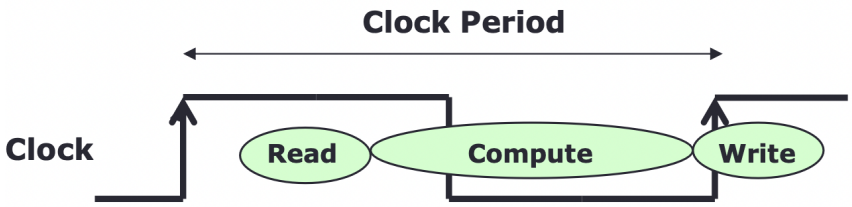
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1





	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

Clock period



- Calculate cycle time assuming negligible delays: memory (2ns), ALU/adders (2ns), register file access (1ns)

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

- All instructions take as much time as the slowest one (i.e., load)

  ➔ Long cycle time for each instruction

Solution 1: Multicycle Implementation

Break up the instructions into execution steps:

1. Instruction fetch
2. Instruction decode and register read
3. ALU operation
4. Memory read/write
5. Register write

- In a multi-cycle implementation, every instruction takes the same amount of time as the number of stages it needs to use.
  - e.g. total 5 stages, each stage 1ns, but uses only 3 stages → instruction takes 3ns
- Each execution step takes one clock cycle
  - **Cycle time is much shorter, i.e., clock frequency is much higher**
- Instructions take variable number of clock cycles to complete execution
- A multicycle implementation is better than single-cycle since not all instructions need to pass through every stage of the datapath.

- In a multicycle implementation where every stage takes the same amount of time to complete, the instruction that passes through all the stages will have the same execution time as in a single-cycle implementation.
- In a multicycle implementation where every stage may take a different amount of time to complete, the instruction that passes through all the stages will NOT have the same execution time as in a single cycle implementation.

Instruction	Inst-Mem	Reg-Read	ALU	Data-Mem	Reg-Write	Total
eg	1	2	3	4	5	15

In single-cycle (assuming that this is the longest instruction), it would take 15 units of time.

In multicycle, where we have a cycle for each stage, each cycle needs to be at least 5 units long. Therefore multicycle would take 5 * 5 = 25 units of time.

Solution 2: Pipelining

- Break up the instructions into execution steps
  - one per clock cycle
- Allow different instructions to be in different execution steps simultaneously

L13: Boolean Algebra

Boolean Algebra

Boolean values:

- True (T or 1)
- False (F or 0)

Connectives:

- Conjunction (AND)
  - $A \cdot B$ or $A \wedge B$
  - can just use full stop in exam
  - AB means it is a 2-bit value
- Disjunction (OR)
  - $A + B$ or $A \vee B$

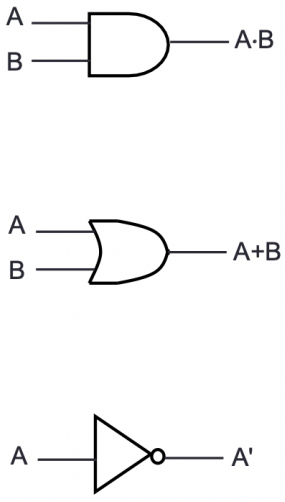
Truth tables

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	A'
0	1
1	0

Logic gates



- Negation (NOT)
  - A' or $\bar{A}$ or $\neg A$

Proof using Truth Table

- Prove: **x · (y + z) = (x · y) + (x · z)**
  - Construct truth table for LHS and RHS

x	y	z	y + z	x · (y + z)	x · y	x · z	(x · y) + (x · z)
0	0	0	0	**0**	0	0	**0**
0	0	1	1	**0**	0	0	**0**
0	1	0	1	**0**	0	0	**0**
0	1	1	1	**0**	0	0	**0**
1	0	0	0	**0**	0	0	**0**
1	0	1	1	**1**	0	1	**1**
1	1	0	1	**1**	1	0	**1**
1	1	1	1	**1**	1	1	**1**

  - Check that column for LHS = column for RHS

Precedence of Operators

- NOT (') > AND (.) > OR (+)
- use parentheses to overwrite precedence

Laws of Boolean Algebra

Identity laws

- $A + 0 = 0 + A = A$
- $A \cdot 1 = 1 \cdot A = A$

Inverse / Complement laws

- $A + A' = A' + A = 1$
- $A \cdot A' = A' \cdot A = 0$

Commutative laws

- $A + B = B + A$
- $A \cdot B = B \cdot A$

Associative laws

- $A + (B + C) = (A + B) + C = A + B + C$
- $A \cdot (B \cdot C) = (A \cdot B) \cdot C = A \cdot B \cdot C$

Distributive laws

- $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
- $A + (B \cdot C) = (A + B) \cdot (A + C)$

Duality

- If the AND/OR operators and identity elements 0/1 in a boolean equation are interchanged, it remains valid.
- dual equation of $a + (b \cdot c) = (a + b) \cdot (a + c)$ is $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- e.g. If $(x + y + z)' = x' \cdot y' \cdot z'$ is valid, then its dual $(x \cdot y \cdot z)' = x' + y' + z'$ is also valid.
- e.g. If $x + 1 = 1$ is valid, then its dual $x \cdot 0 = 0$ is also valid.

Theorems

Idempotency

- $X + X = X$
- $X \cdot X = X$

One element / Zero element

- $X + 1 = 1 + X = 1$
- $X \cdot 0 = 0 \cdot X = 0$

Involution

- $(X')' = X$

Absorption 1

- $X + X \cdot Y = X$
- $X \cdot (X + Y) = X$

Absorption 2

- $X + X' \cdot Y = X + Y$
- $X \cdot (X' + Y) = X \cdot Y$

De Morgans' (can be generalised to more than 2 var)

- $(X + Y)' = X' \cdot Y'$
- $(X \cdot Y)' = X' + Y'$

Consensus

- $X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$
- $(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y)\dot{(}X' + Z)$

Useful equations

$X \bigoplus Y = X' \cdot Y + X \cdot Y'$

$X \bigodot Y = (X \bigoplus Y)' = X' \cdot Y' + X \cdot Y$

Proving a Theorem

Prove absorption theorem X + X.Y = X

- X + X.Y

  = X.1 + X.Y (by identity law)

  = X.(1+Y) (by distributivity)

  = X.1 (by one element law)

  = X (by identity law)

- By the principle of duality, we may also cite (without proof) that X.(X+Y) = X

Standard Forms

- Every Boolean expression can be expressed in SOP or POS form.

Literals

- A Boolean variable on its own or in its complemented form
  - e.g. x
  - x'
  - y
  - y'

	Expression	*SOP?*	*POS?*
(1)	X'·Y + X·Y' + X·Y·Z	✓	×
(2)	(X+Y')·(X'+Y)·(X'+Z')	×	✓
(3)	X' + Y + Z	✓	✓
(4)	X·(W' + Y·Z)	×	×
(5)	X·Y·Z'	✓	✓
(6)	W·X'·Y + V·(X·Z + W')	×	×

Product term (AND)

- A single literal or a logical product (AND) of several literals
  - e.g. x
  - x.y.z'
  - A.B
  - d.g'.v.w

Sum term (OR)

- A single literal or a logical sum (OR) of several literals
  - e.g. x
  - x+y+z'
  - A'+B
  - A+B
  - c+d+h'+j

Sum-of-Products (SOP) expression

- A product term or a logical sum (OR) of several product terms
  - e.g. x
  - x+y.z'
  - x.y'+x'.y.z
  - A.B+A'.B
  - A+B'.C+A.C'+C.D

Product-of-Sums (POS) expression

- A sum term or a logical product (AND) of several sum terms
  - e.g. x
  - x.(y+z')
  - (x+y').(x'+y+z)
  - (A+B).(A'+B')
  - (A+B+C).D'.(B'+D+E')

Minterm

- A minterm of n variables is a product term that contains n literals from all the variables.
- e.g. On 2 variables x and y, the minterms are: x'·y', x'·y, x·y' and x·y

	Boolean expression	Minterm notation
(1)	A'·B'·C·D	m3
(2)	**A·B'·C·D'**	m10
(3)	**A·B'·C·D**	m11
(4)	A·B·C·D'	**m14**
(5)	A·B'·C'·D	**m9**

look at non-neg from left to right (binary sum)

Maxterm

- A maxterm of n variables is a sum term that contains n literals from all the variables.
- e.g. On 2 variables x and y, the maxterms are: x'+y', x'+y, x+y' and x+y

	Boolean expression	Maxterm notation
(1)	A+B+C'+D'	M3
(2)	**A'+B'+C+D'**	M13
(3)	**A+B+C+D**	M0
(4)	A+B+C'+D	**M2**
(5)	A'+B+C+D'	**M9**

look at neg from left to right

x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	x'·y'	m0	x+y	M0
0	1	x'·y	m1	x+y'	M1
1	0	x·y'	m2	x'+y	M2
1	1	x·y	m3	x'+y'	M3

- In general, with n variables we have up to 2^n minterms and 2^n maxterms.
- Each minterm is the complement of its corresponding maxterm.
- Likwise, each maxterm is the complement of its corresponding minterm.
  - e.g. m2 = x·y'
    m2' = (x·y')' = x' + (y')' = x' + y = M2

Canonical Forms

Canonical/normal form: a unique form of representation.

- Sum-of-minterms = Canonical sum-of-products
- Product-of-maxterms = Canonical product-of-sums

Sum-of-Minterms

Obtain sum-of-minterms expression by gathering the minterms of the function (where output is 1).

F1 = x·y·z' = m6

$$F2 = x' \cdot y' \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z + x \cdot y \cdot z' + x \cdot y \cdot z$$
$$= m1 + m4 + m5 + m6 + m7$$
$$= \sum m(1,4,5,6,7) \; or \; \sum m(1,4\text{–}7)$$

x	y	z	F1	F2	F3
0	0	0	0	0	0
0	0	1	0	(1)	1
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	0	(1)	1
1	0	1	0	(1)	1
1	1	0	(1)	(1)	0
1	1	1	0	(1)	0

$$F3 = x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z$$
$$= m1 + m3 + m4 + m5$$
$$= \sum m(1,3,4,5) \; or \; \sum m(1,3\text{–}5)$$

Product-of-Maxterms

Obtain product-of-maxterms expression by gathering the maxterms of the function (where output is 0).

$$F2 = (x + y + z) \cdot (x + y' + z) \cdot (x + y' + z')$$
$$= M0 \cdot M2 \cdot M3$$
$$= \prod M(0,2,3)$$

x	y	z	F1	F2	F3
0	0	0	0	(0)	0
0	0	1	0	1	1
0	1	0	0	(0)	0
0	1	1	0	(0)	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	0	1	0

$$F3 = (x + y + z) \cdot (x + y' + z) \cdot (x' + y' + z) \cdot (x' + y' + z')$$
$$= M0 \cdot M2 \cdot M6 \cdot M7$$

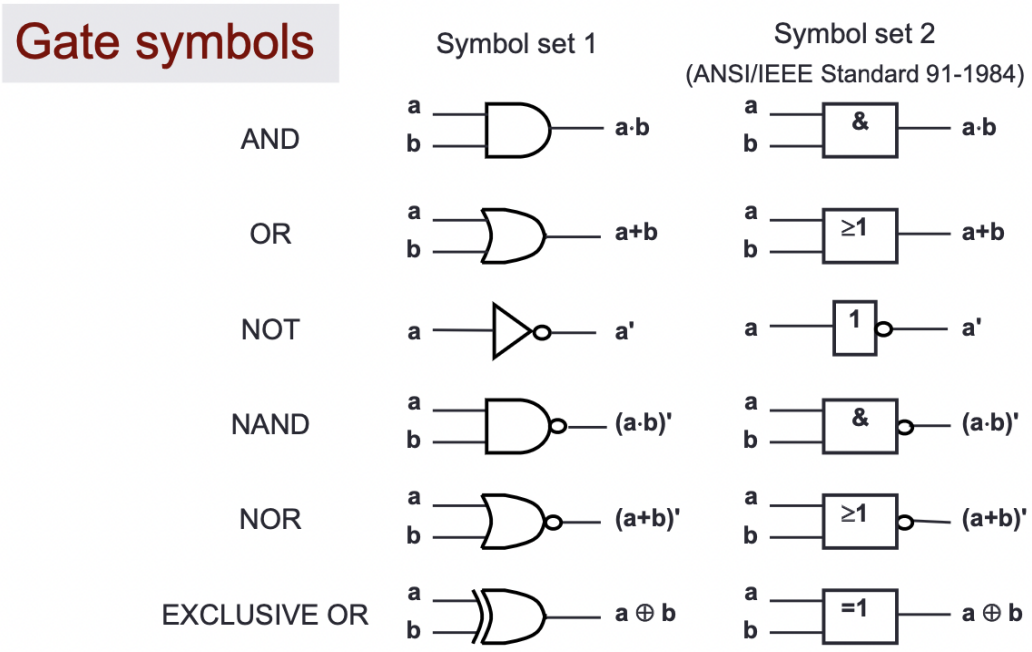
$= \prod M(0, 2, 6, 7)$

$F2 = \sum m(1, 4, 5, 6, 7) = \prod M(0, 2, 3)$

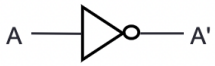
- F2' = m0 + m2 + m3
- F2 = (m0 + m2 + m3)'

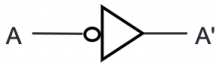
  = m0' · m2' · m3' (by DeMorgan's)
  = M0 · M2 · M3 (as m*x*' = M*x*)

L14: Logic Circuits



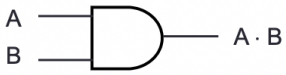
Inverter (NOT gate)





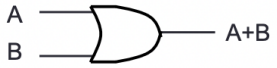
A	A'
0	1
1	0

AND gate



A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR gate



A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

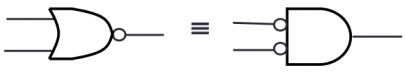
NAND gate

A	B	(A · B)'
0	0	1
0	1	1
1	0	1
1	1	0

NAND Negative-OR

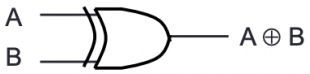
NOR gate

A	B	(A + B)'
0	0	1
0	1	0
1	0	0
1	1	0



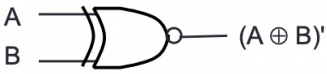
NOR Negative-AND

XOR gate



A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

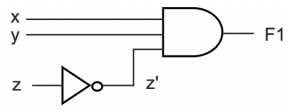
XNOR gate



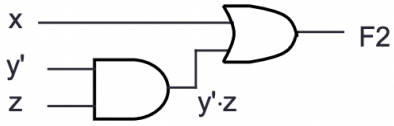
XNOR can be represented by ⊙
(Example: A ⊙ B)

A	B	(A ⊕ B)'
0	0	1
0	1	0
1	0	0
1	1	1

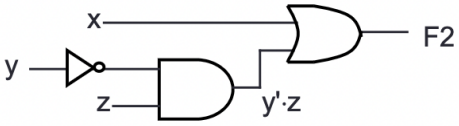
- Fan-in: the number of inputs of a gate.
- Gates may have fan-in more than 2.
- Given a Boolean expression, we may implement it as a logic circuit.
    - e.g. F1 = x.y.z'



Example: F2 = x + y'·z

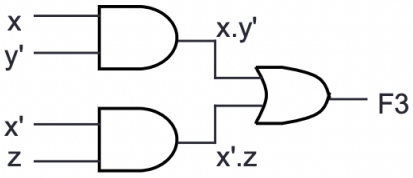


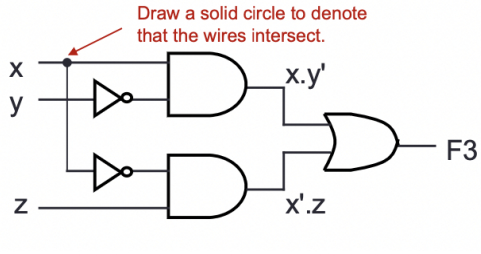
If complemented literals are available



If complemented literals are not available

Example: F3 = x·y' + x'·z

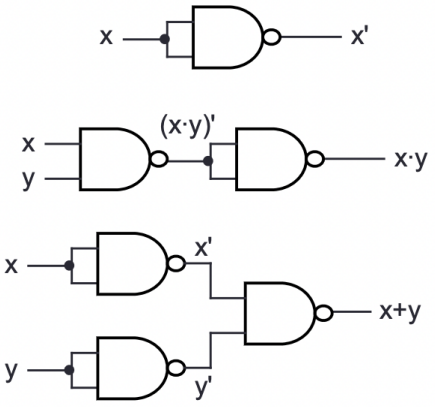




Universal Gates

- the set {AND, OR, NOT} is a complete set of logic.
- Universal gates are gates that can implement the complete set of logic by itself
- e.g. NAND and NOR

- Proof: Implement NOT/AND/OR using only NAND gates.

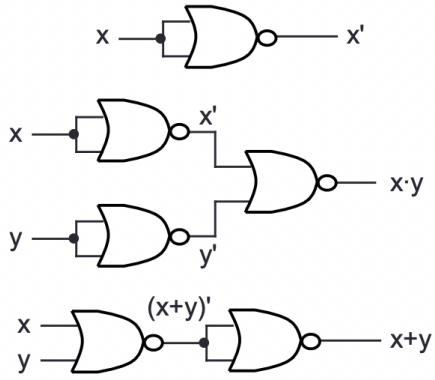


(x·x)' = x' (idempotency)

((x·y)'·(x·y)')' = ((x·y)')' (idempotency)
= x·y (involution)

((x·x)'·(y·y)')' = (x'·y')' (idempotency)
= (x')'+(y')' (DeMorgan)
= x+y (involution)

- Proof: Implement NOT/AND/OR using only NOR gates.



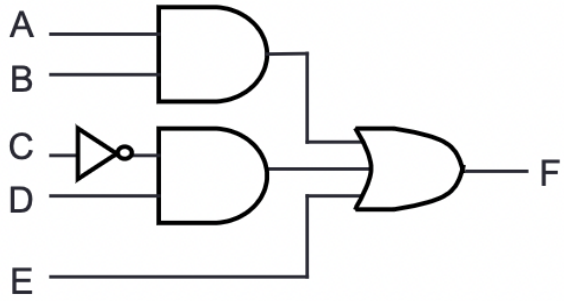
(x+x)' = x' (idempotency)

((x+x)'+(y+y)')' = (x'+y')' (idempotency)
= (x')'·(y')' (DeMorgan)
= x·y (involution)

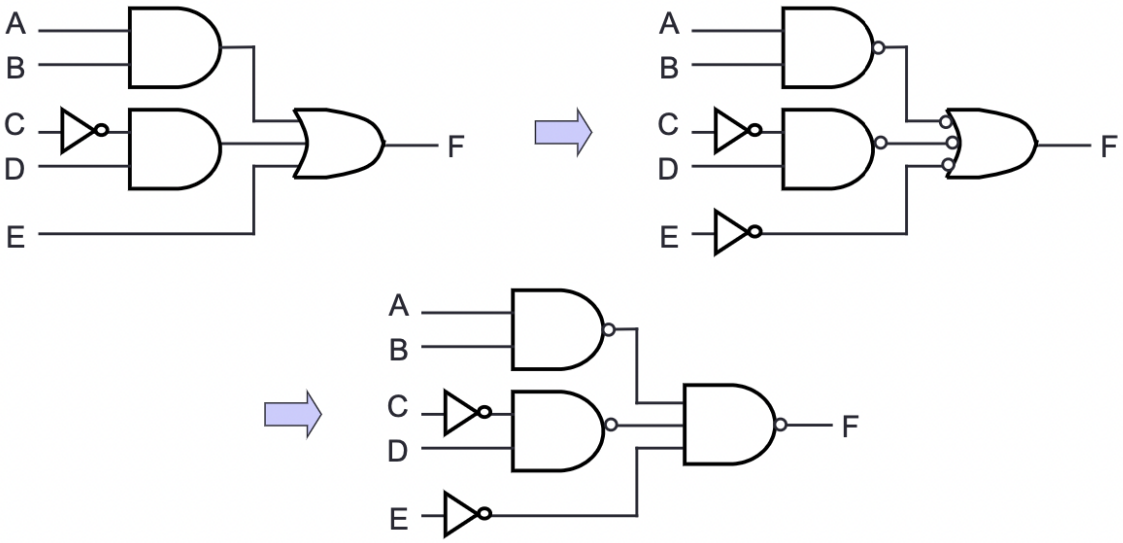
((x+y)'+(x+y)')' = ((x+y)')' (idempotency)
= x+y (involution)

SOP and NAND Circuits

- An SOP expression can be easily implemented using
  - 2-level AND-OR circuit
  - 2-level NAND circuit
- e.g. F = A.B + C'.D + E



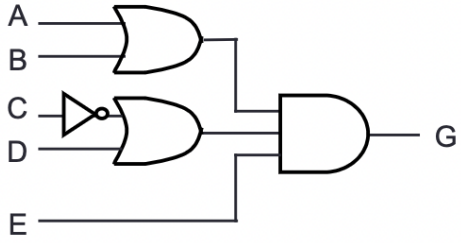
Using 2-level AND-OR circuit



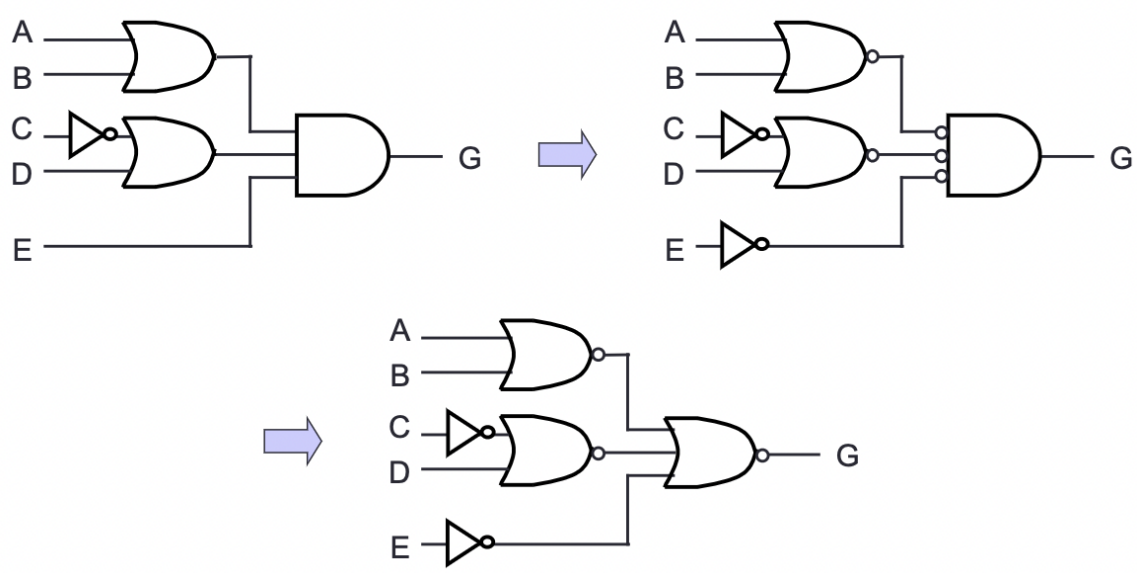
Using 2-level AND-OR circuit

POS and NOR Circuits

- A POS expression can be easily implemented using
  - 2-level OR-AND circuit
  - 2-level NOR circuit
- G = (A+B) . (C'+D) . E



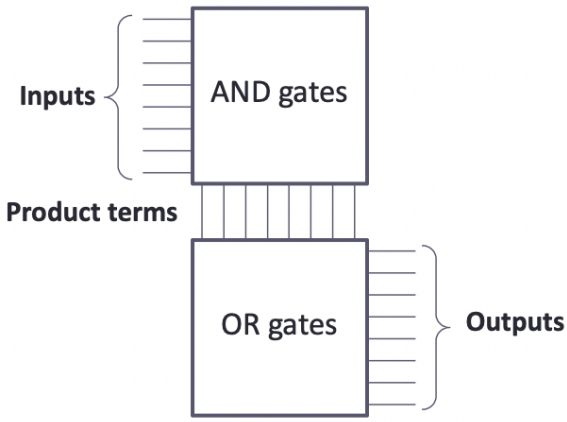
Using 2-level OR-AND circuit



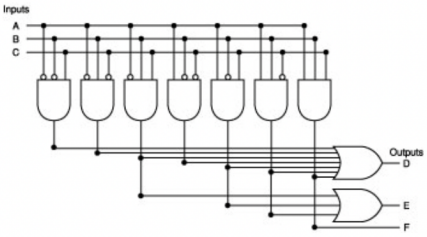
Using 2-level NOR circuit

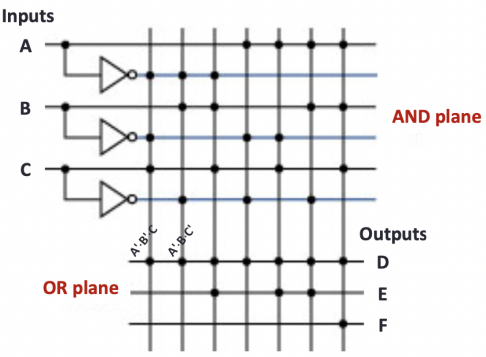
Programming Logic Array (PLA)

- A programmable integrated circuit – implements sum- of-products circuits (allow multiple outputs).
- **2 stages**
  - AND gates = product terms
  - OR gates = outputs



Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1





simplified rep

Read Only Memory (ROM)

- Similar to PLA
  - Set of inputs (called addresses)
  - Set of outputs
  - Programmable mapping between inputs and outputs
- Fully decoded: able to implement any mapping.
- In contrast, PLAs may not be able to implement a given mapping due to not having enough minterms.

L15: Simplification

Algebraic Simplification

F(a,b,c,d) = a·b·c + a·b·d + a'·b·c' + c·d + b·d'

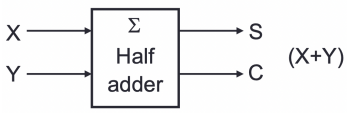
a·b·c + **a·b·d** + a'·b·c' + c·d + **b·d'**	
= a·b·c + **a·b + a'·b·c'** + c·d + b·d'	(absorption 2)
= **a·b·c + a·b** + b·c' + c·d + b·d'	(absorption 2)
= a·b + **b·c'** + c·d + **b·d'**	(absorption 1)
= a·b + c·d + b·(**c'+d'**)	(distributivity)
= a·b + **c·d + b·(c·d)'**	(DeMorgan's)
= **a·b** + c·d + **b**	(absorption 2)
= b + c·d	(absorption 1)

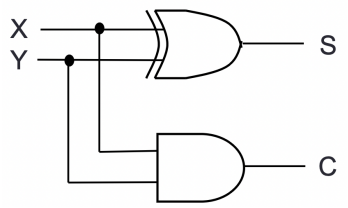
a·b·d + b·d'
= b· (a·d + d')
= b· (a + d')
= a·b + b·d'

Number of literals reduced from 13 to 3.

Half-Adder

- Half adder is a circuit that adds 2 single bits (X, Y) to produce a result of 2 bits (C, S).
- **In canonical form (sum-of-minterms):**
  - C = X·Y
  - $S = X' \cdot Y + X \cdot Y' = X' \cdot Y + X \cdot Y' = X \bigoplus Y$





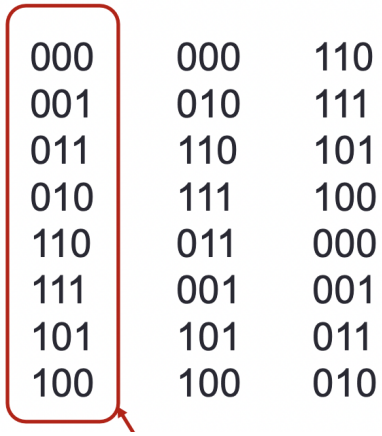
Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

C: carry; S: sum

Gray Code

- Unweighted (not an arithmetic code)

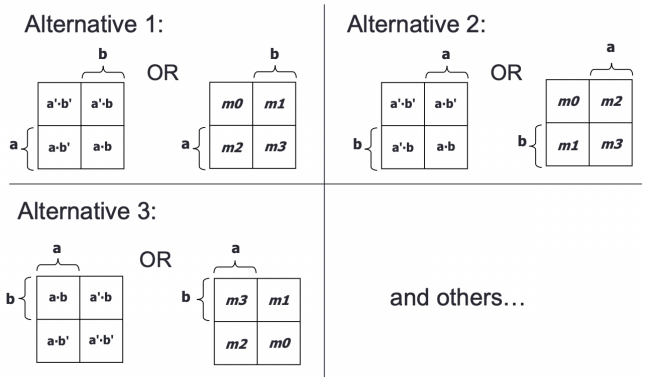
- Only a single bit change from one code value to the next.
- Not restricted to decimal digits: *n* bits → 2^n values.
- There are many gray codes

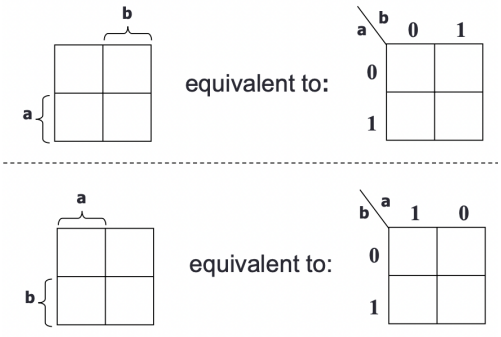


circled one is the standard gray code; for permutations of the bits must be used, cannot have duplicate/unused permutation

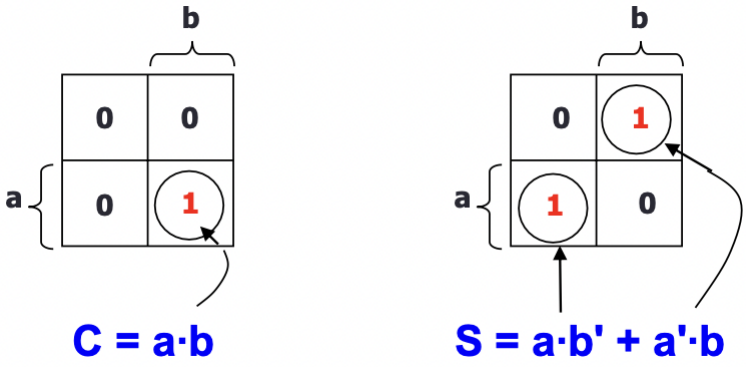
K-Maps Simplification (SOP)

- disadv: Limited to 5 or 6 variables
- In general, each cell in an n-variable K-map has n adjacent neighbours.
- A '**1**' in the square the corresponds to a minterm of the function
- A '0' otherwise
- e.g. Let the 2 variables be a and b.



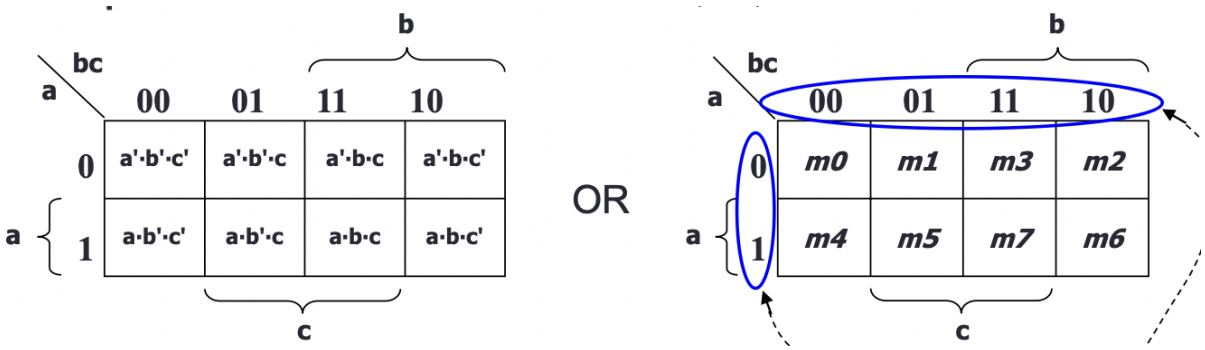


2-Var K-Map



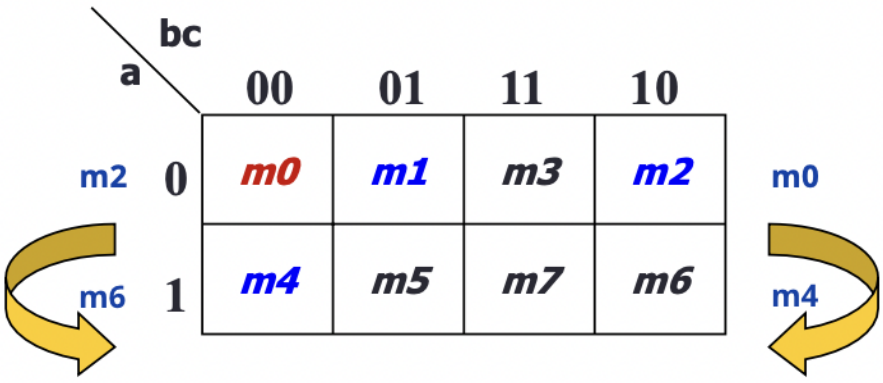
3-Var K-Map

- minterms of adjacent cells differ by only *ONE literal*.
- Other arrangements which satisfy this criterion may also be used.



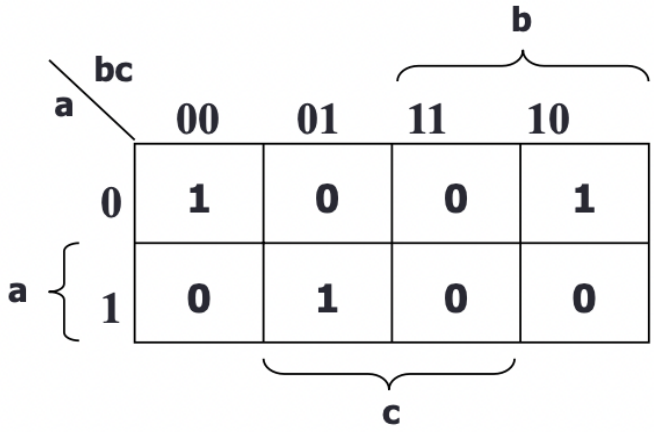
Note Gray code sequence: the value differs from its previous by 1 bit.

- **There is wrap-around in the K-map:**
  - a'·b'·c' (*m0*) is adjacent to a'·b·c' (*m2)*
  - a·b'·c' (*m4*) is adjacent to a·b·c' (*m6*)



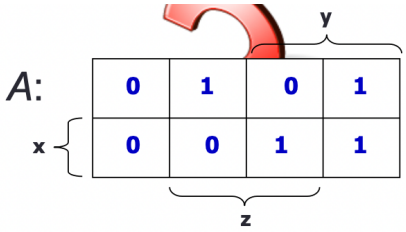
Each cell in a 3-variable K-map has 3 adjacent neighbours. e.g. m0 has 3 adjacent neighbours: m1, m2 and m4.

The K-map of a 3-variable function *F* is shown below. What is the sum-of-minterms expression of *F*?

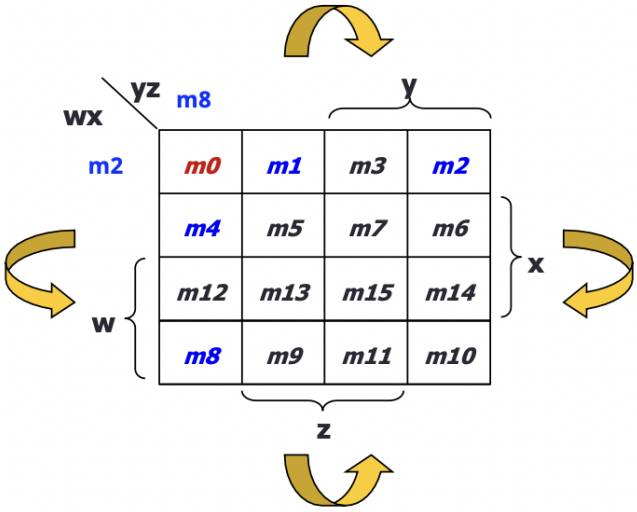


$\sum m(0,2,5) = a' \cdot b' \cdot c' + a' \cdot b \cdot c' + a \cdot b' \cdot c$

Draw the K-map for this function A: A(x, y, z) = x·y + y·z' + x'·y'·z



4-Var K-Map

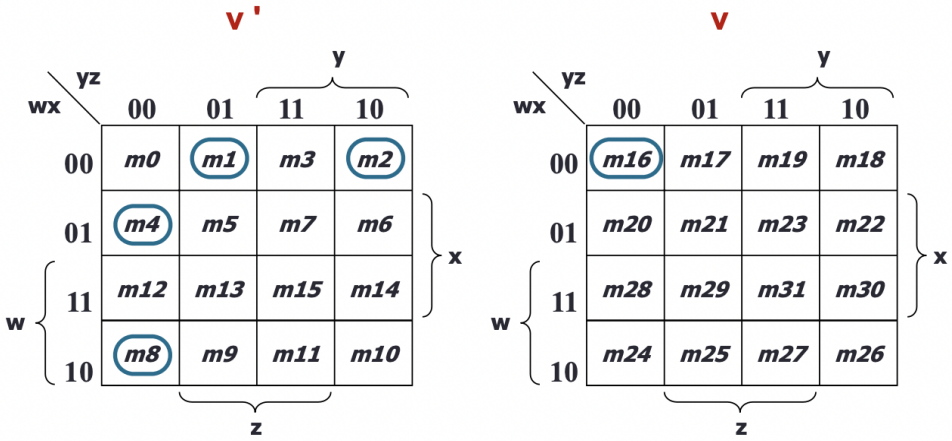


The cell corresponding to minterm m0 has neighbours m1, m2, m4 and m8.

5-Var K-Map

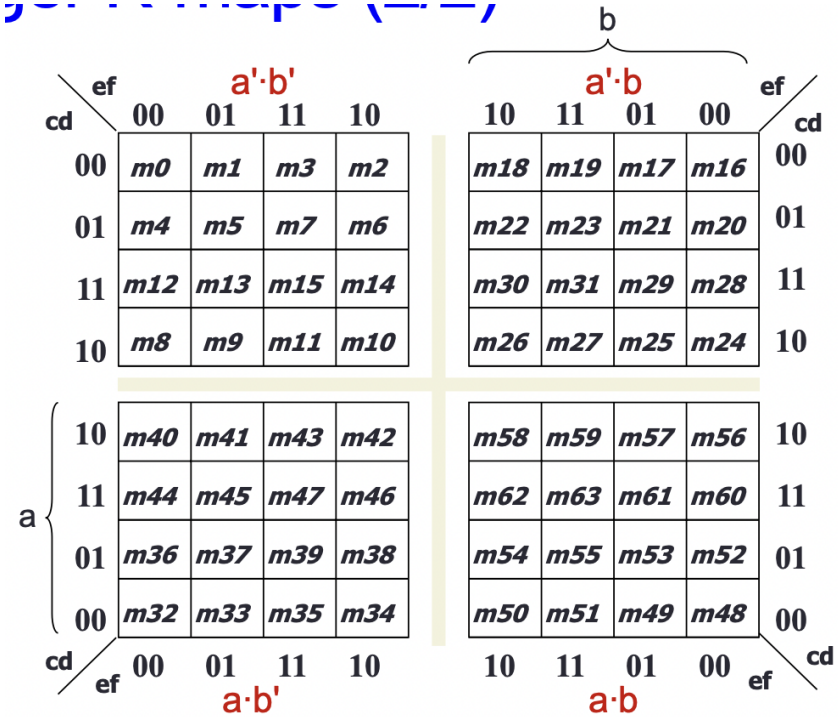
- Organised as two 4-variable K-maps. One for v' and the other for v.

- Can visualise this as one 4-variable K-map being on TOP of the other 4-variable K-map.



6-Var K-Map

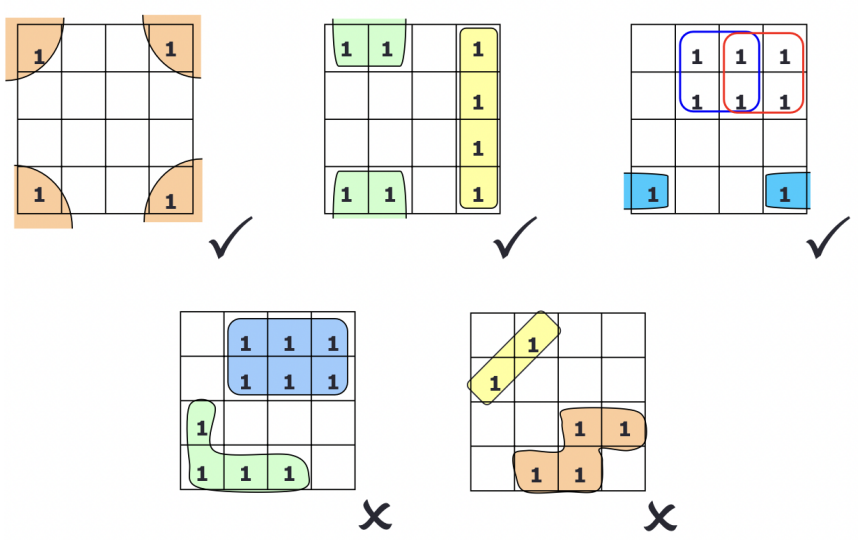
- organised as four 4- variable K-maps, mirrored along two axes.



Use K-Maps to simplify expressions

- In a K-map, each cell containing a ‘1’ corresponds to a minterm of a given function F where the output is 1.
- Each valid grouping of **adjacent cells** containing ‘1’ then corresponds to a simpler product term of F.

- A group must have size in powers of two: 1, 2, 4, 8, ...

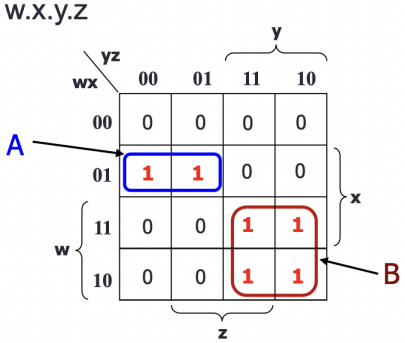


1. Group as many cells as possible → reduce as many terms as possible
2. Select as few groups as possible to cover all the cells (minterms) of the function
    - in writing out final ans, write in order of significance as shown
    - e.g. F(w,x,y,z), final ans = w'.x.y', not y'.x.w'

$$F(w,x,y,z) = w'\cdot x\cdot y'\cdot z' + w'\cdot x\cdot y'\cdot z + w\cdot x'\cdot y\cdot z' + w\cdot x'\cdot y\cdot z + w\cdot x\cdot y\cdot z' + w\cdot x\cdot y\cdot z$$
$$= \Sigma\, m(4, 5, 10, 11, 14, 15)$$

A = w'.x.y'.z' + w'.x.y'.z = w'.x.y'.(z' + z) = **w'.x.y'**

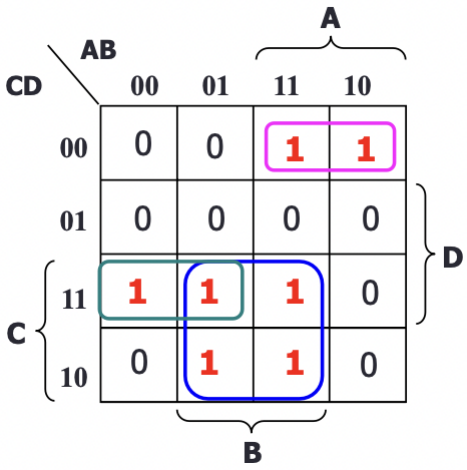
B = w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z
= w.x'.y.(z' + z) + w.x.y.(z' + z)
= w.x'.y + w.x.y
= w.(x'+x).y
= **w.y**



F(w,x,y,z) = A + B = w'·x·y' + w·y

Convert to Minterms

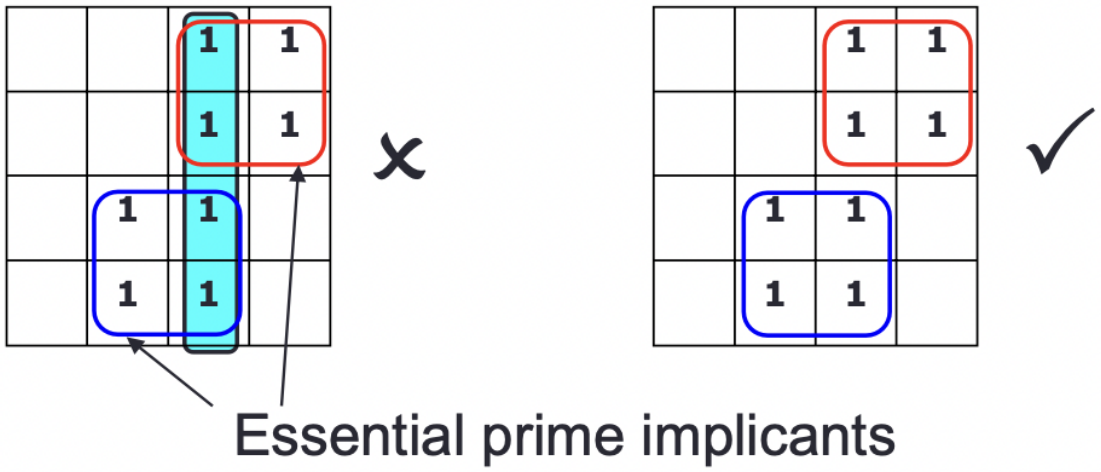
$$F(A,B,C,D) = A.(C+D)'.(B'+D') + C.(B+C'+A'.D)$$
$$= A.(C'.D').(B'+D') + B.C + C.C' + A'.C.D$$
$$= A.B'.C'.D' + A.C'.D' + B.C + A'.C.D$$



Prime Implicants (PIs) & Essential Prime Implicants (EPIs)

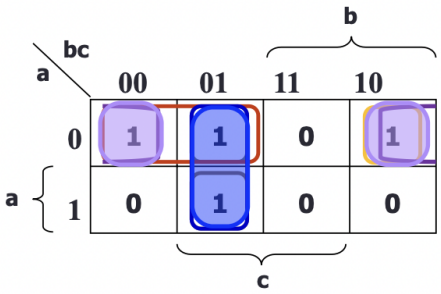
- **Implicant:** a product term that could be used to cover minterms of the function.
- **Prime implicant (PI):** a product term obtained by combining the maximum possible number of minterms from adjacent squares in the map. (That is, it is the biggest grouping possible.)

- **Essential prime implicant (EPI):** a prime implicant that includes at least one minterm that is not covered by any other prime implicant.
- No redundant groups:

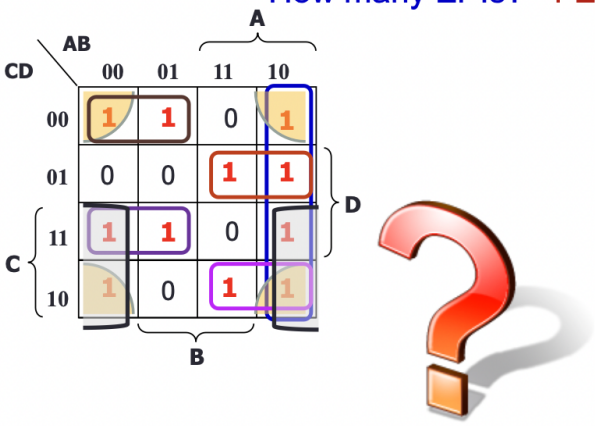


5-3. Identify the prime implicants and essential prime implicants of the two K-maps below.

How many PIs? 7 PIs
How many EPIs? 4 EPIs

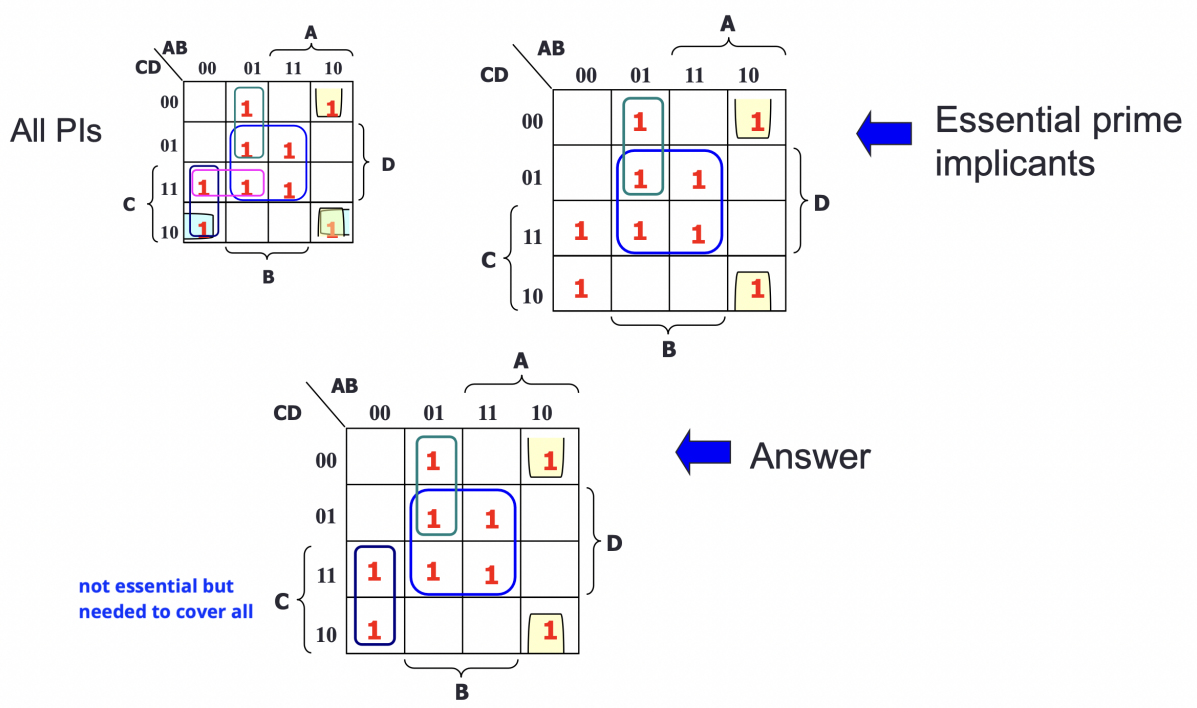


How many PIs? 3 PIs
How many EPIs? 2 EPIs



Algorithm:

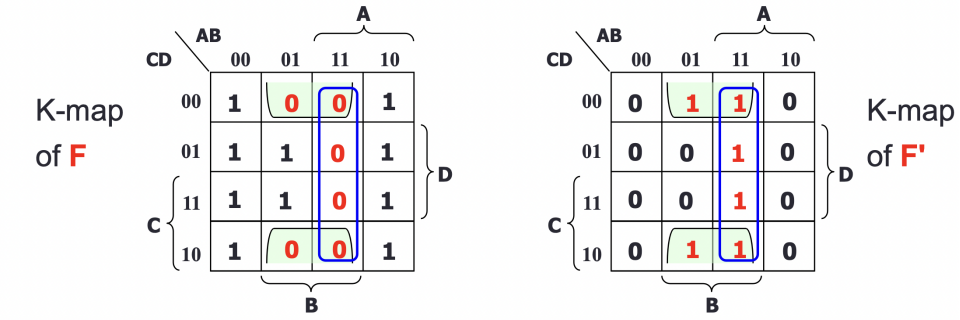
1. Circle all prime implicants on the K-map.
2. Identify and select all essential prime implicants for the cover.
3. Select a minimum subset of the remaining PIs to complete to cover those minterms not covered by the EPIs.



F(A,B,C,D) = B·D + A'·B·C' + A·B'·D' + A'·B'·C

Use K-Maps to find POS

- Simplified POS expression can be obtained by grouping the maxterms (i.e. 0s) of the given function.
- e.g. Given $F = \sum m(0,1,2,3,5,7,8,9,10,11)$



This gives the SOP of F' to be

F' = B·D' + A·B

To get POS of F, we have

F = (B·D' + A·B)'
= (B·D')' · (A·B)' (DeMorgan)
= **(B'+D) · (A'+B')** (DeMorgan)

Don't Care Conditions (X or d)

- outputs are not specified or are invalid: can be either ‘1’ or ‘0’.
- X could be chosen to be either ‘1’ or ‘0’, depending on which choice results in a simpler expression.
- e.g. A circuit takes in a 3-bit value ABC and outputs 2-bit value FG which is the sum of the input bits. It is also known that inputs 000 and 111 never occur.
  - $F(A,B,C) = \sum m(3,5,6) + \sum d(0,7)$
  - $G(A,B,C) = \sum m(1,2,4) + \sum d(0,7)$

Assuming all inputs are valid.

A	B	C	F	G
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

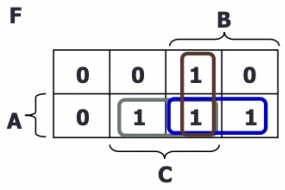
Assuming inputs 000 and 111 are invalid.

A	B	C	F	G
0	0	0	X	X
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	X	X

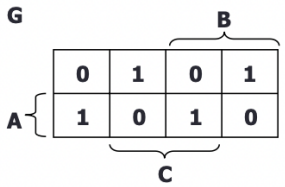
- Without don’t-cares:

  F(A,B,C) = Σm(3, 5, 6, 7)

  G(A,B,C) = Σm(1, 2, 4, 7)



F = A·C + A·B + B·C

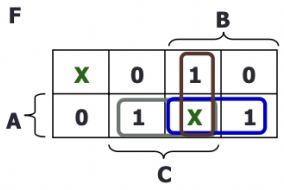


G = A·B'·C' + A'·B'·C + A·B·C + A'·B·C'

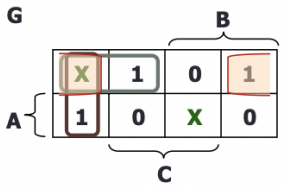
- With don’t-cares:

  F(A,B,C) = Σm(3, 5, 6) + Σd(0, 7)

  G (A,B,C)= Σm(1, 2, 4) + Σd(0, 7)

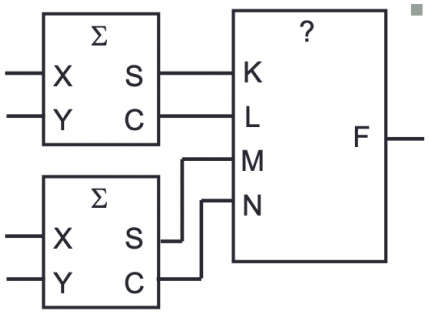


F = A·C + A·B + B·C



G = B'·C' + A'·B' + A'·C'

- Suppose you are given the truth table for a function F(K,L,M,N) as follows:
- You are also told that the inputs K, L, M, N are taken from the outputs of two half adders as shown:

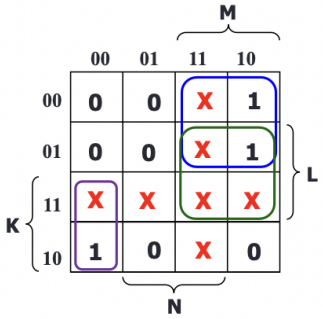


- Then you may revise the truth table:

K	L	M	N	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	X
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

K & L, M & N cannot both be 1 (half adder property)

- K-map of *F*:

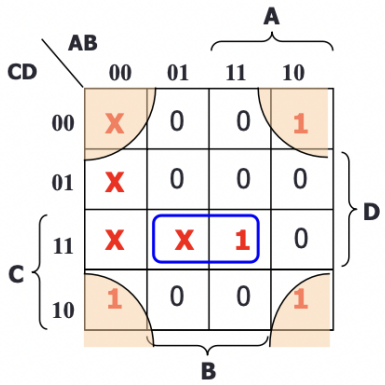


- PIs: K'·M, L·M, and K·M'·N'
  (Note: K·L and M·N not considered PIs as they consist of only X's.)
- EPIs: K'·M and K·M'·N'
- Simplified SOP:

  F = K'·M + K·M'·N'

K	L	M	N	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	X
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

$$F(A,B,C,D) = \Sigma m(2,8,10,15) + \Sigma d(0,1,3,7)$$



Do we need to have an additional term A'·B' to cover the 2 remaining X's?

No, because all the 1's (minterms) have been covered.

Answer: **F(A,B,C,D) = B'·D' + B·C·D**

L17: Combinatorial Circuits

Each output depends entirely on the immediate (present) inputs.

Design methods:

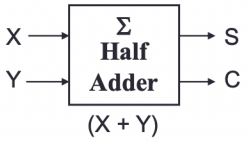
- Gate-level design (with logic circuits)
- Block-level design (with functional blocks)

Types of Integrated Circuit (IC) chips: SSI (small-scale integration), MSI (medium), LSI (large), VLSI (very large), ULSI (ultra large)

Gate-Level (SSI) Design

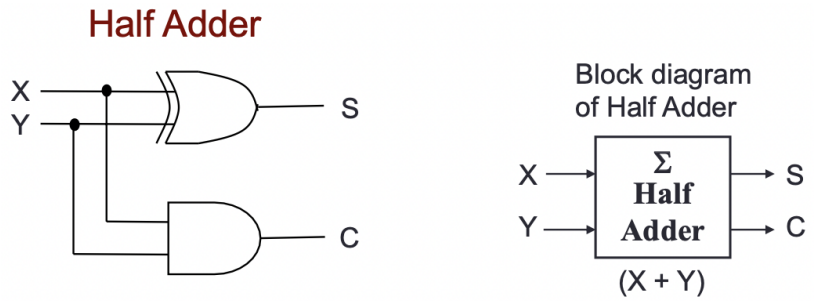
Half Adder

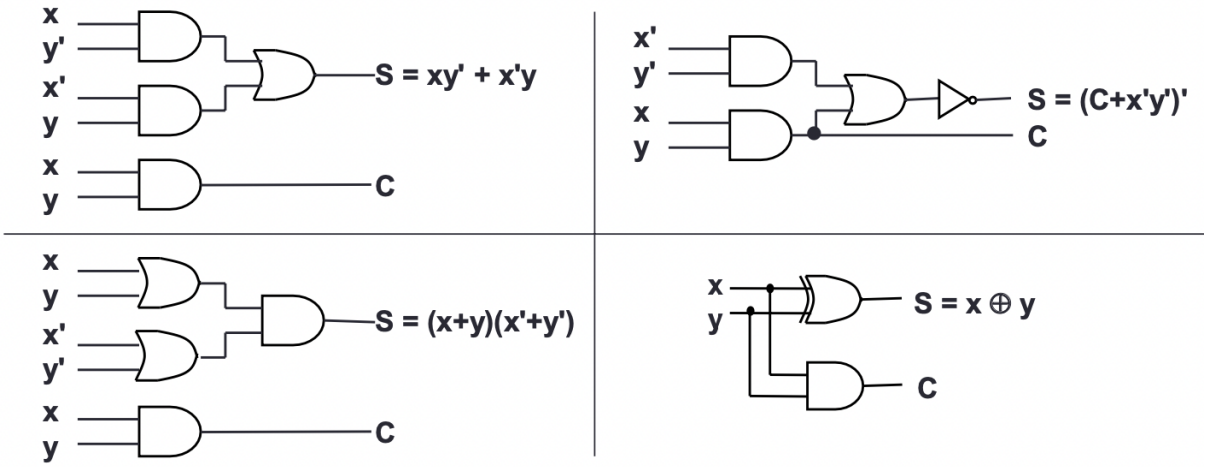
1. State problem
   - e.g. build a half adder
2. Determine and label the inputs and outputs of circuit
   - 2 inputs and 2 outputs labelled
3. Draw the truth table



4. Obtain simplified Boolean functions
   - C = X . Y
   - $S = X' \cdot Y + X \cdot Y' = X \oplus Y$
5. Draw the logic diagram

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0





Full adder

- add two binary numbers

Truth table:

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Note:
Z - carry in (to the current position)
C - carry out (to the next position)

C

X \ YZ	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Using K-map, simplified SOP form:

$C = X \cdot Y + X \cdot Z + Y \cdot Z$

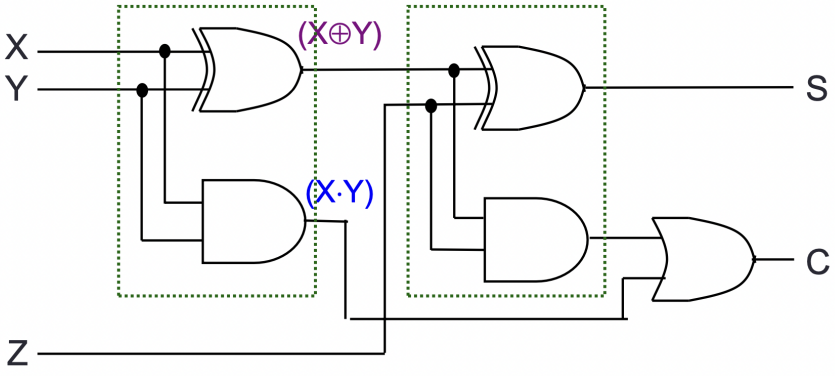
$S = X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z$

S

X \ YZ	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$C = X \cdot Y + (X \oplus Y) \cdot Z$

$S = X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$ (XOR is associative)



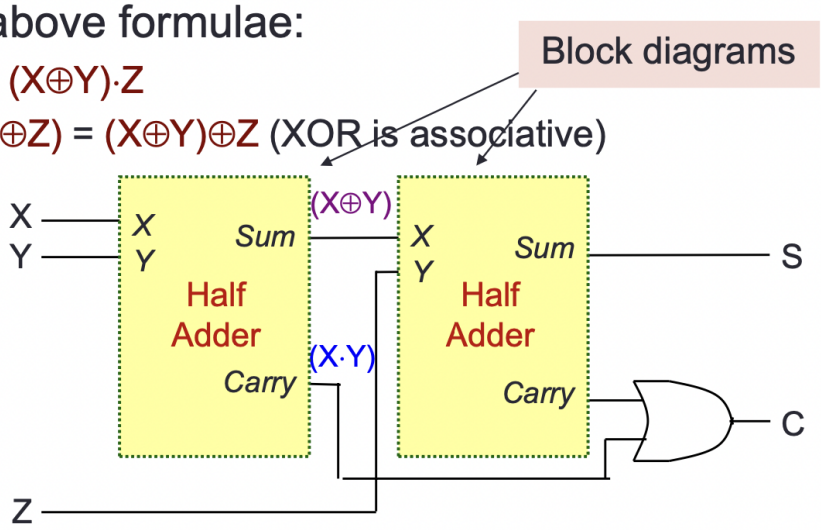
Full Adder made from two Half-Adders (+ an OR gate).

simplified C & S using boolean algebra

Circuit for above formulae:

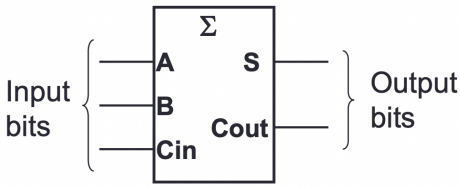
C = X·Y + (X⊕Y)·Z

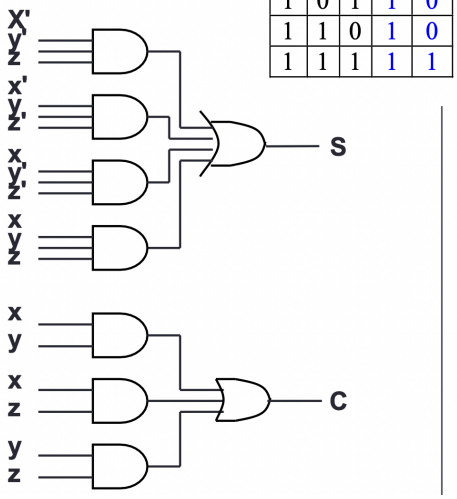
S = X⊕(Y⊕Z) = (X⊕Y)⊕Z (XOR is associative)

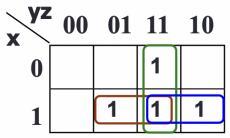


Full adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

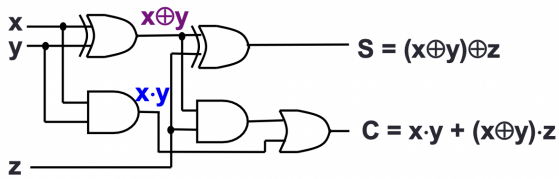






C = xy + xz + yz

S = x'y'z + x'yz' + xy'z' + xyz

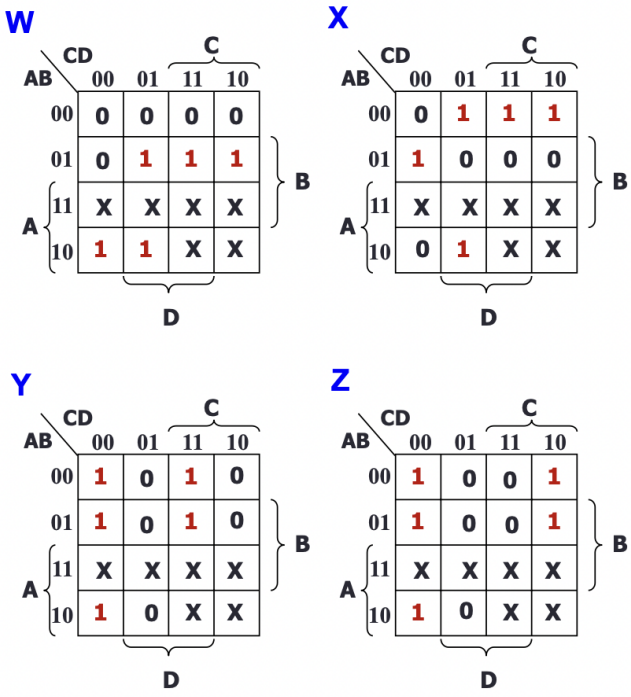


BCD to Excess-3 Code Converter

Truth table:

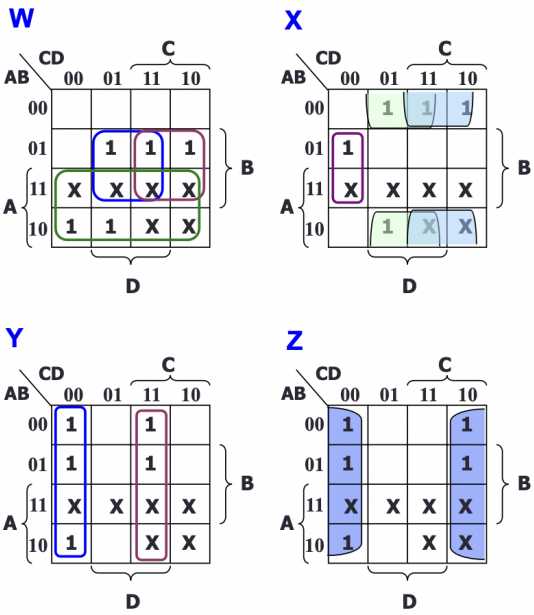
	BCD				Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X
12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

- K-maps:



BCD is meant to only represent 0 - 9, hence m10 to m15 are invalid;

Excess-3 code = BCD Code + 0011



$W = A + B \cdot C + B \cdot D$

$X = B' \cdot C + B' \cdot D + B \cdot C' \cdot D'$

$Y = C.D + C'.D'$

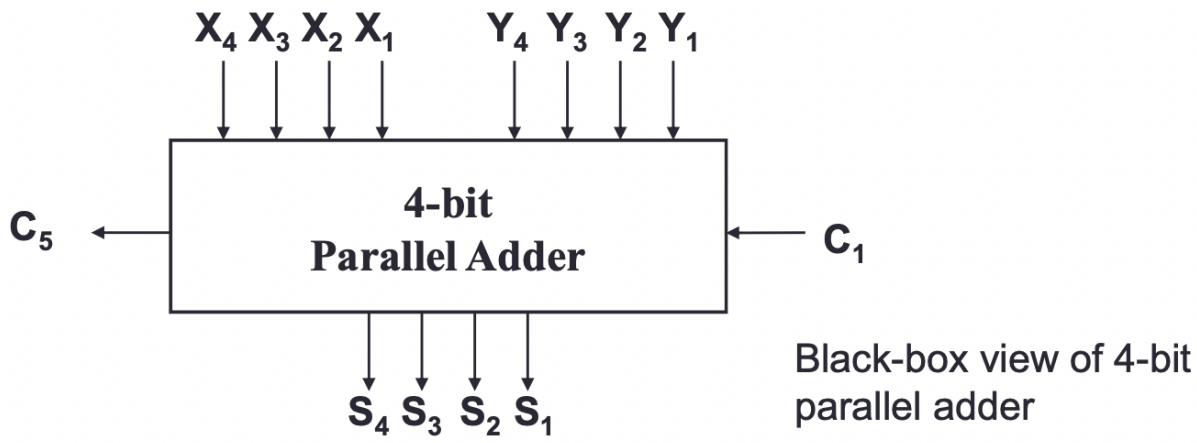
$Z = D'$

Block-Level Design

- relies on algorithms or formulae of the circuit

4-bit Parallel Adder

- A circuit to add two 4-bit numbers together and a carry-in, to produce a 5-bit result.
- Parallel Adder: inputs are presented simultaneously (in parallel).
- aka Ripple-Carry Adder.



Black-box view of 4-bit parallel adder

- 5-bit result is sufficient because the largest result is 1111 + 1111 + 1 = 11111

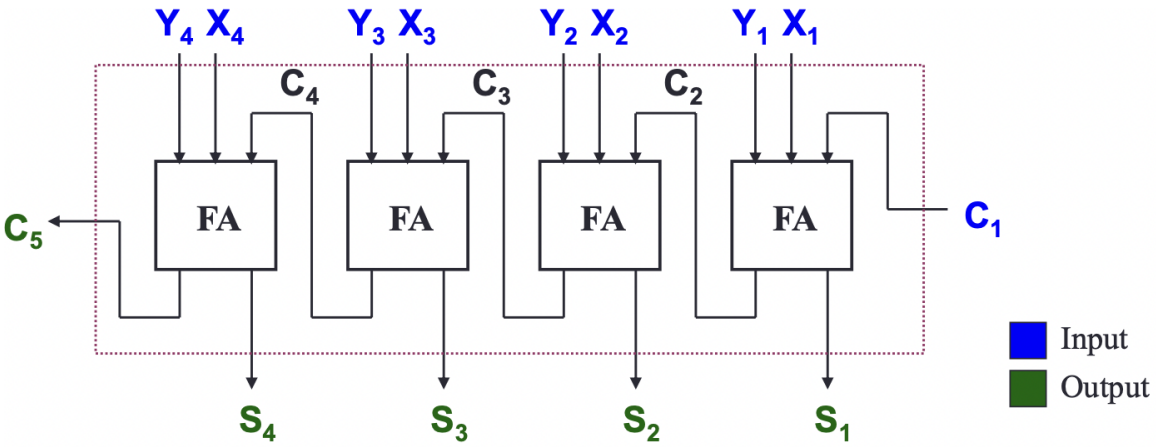
Addition formula for each pair of bits (with carry in),

$$C_{i+1}S_i = X_i + Y_i + C_i$$

has the same function as a full adder:

$$C_{i+1} = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot C_i$$
$$S_i = X_i \oplus Y_i \oplus C_i$$

C =	1 1 0 0
X =	1 0 1 0
Y =	1 1 1 1
X + Y =	1 1 0 0 1

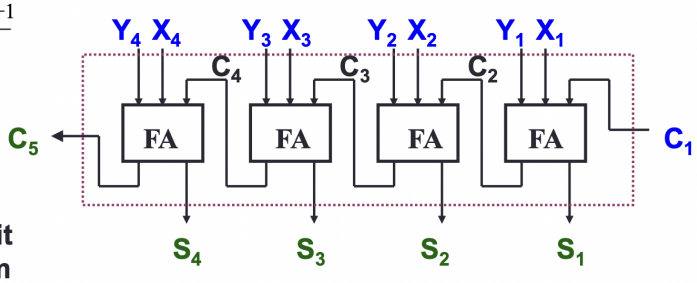


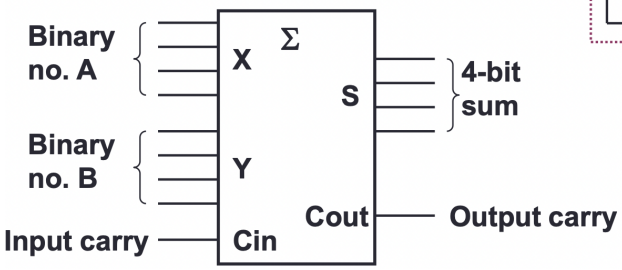
FA = Full Adder

Subscript i	4	3	2	1	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

2 ways:

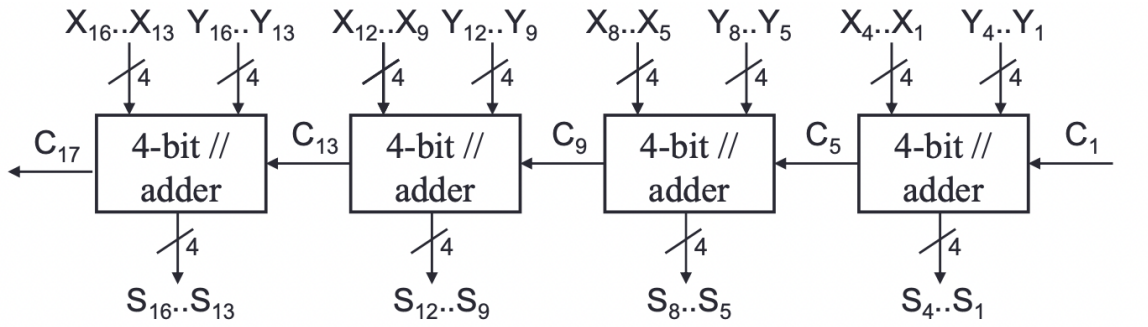
- Serial (one FA)
- Parallel (*n* FAs for *n* bits)



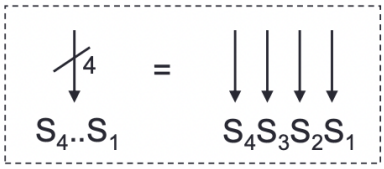


16-bit Parallel Adder

- constructed from four 4-bit parallel adders

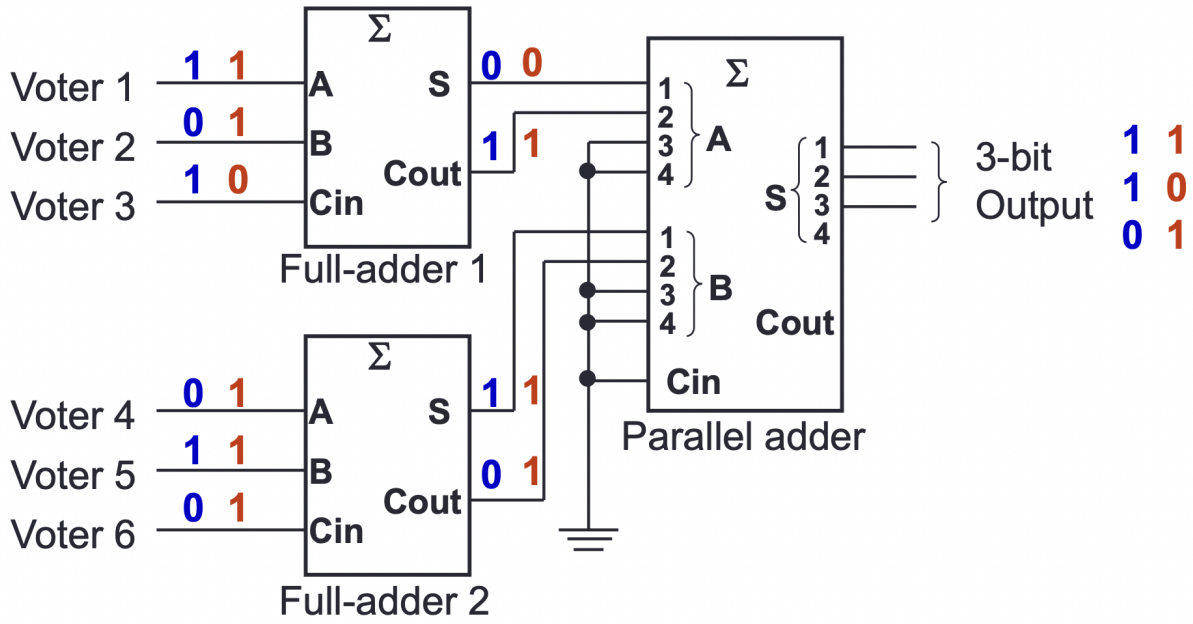


A 16-bit parallel adder



Example: 6-Person Voting System

- Each FA can sum up to 3 vote (X = 1; Y = 1; C = 1)

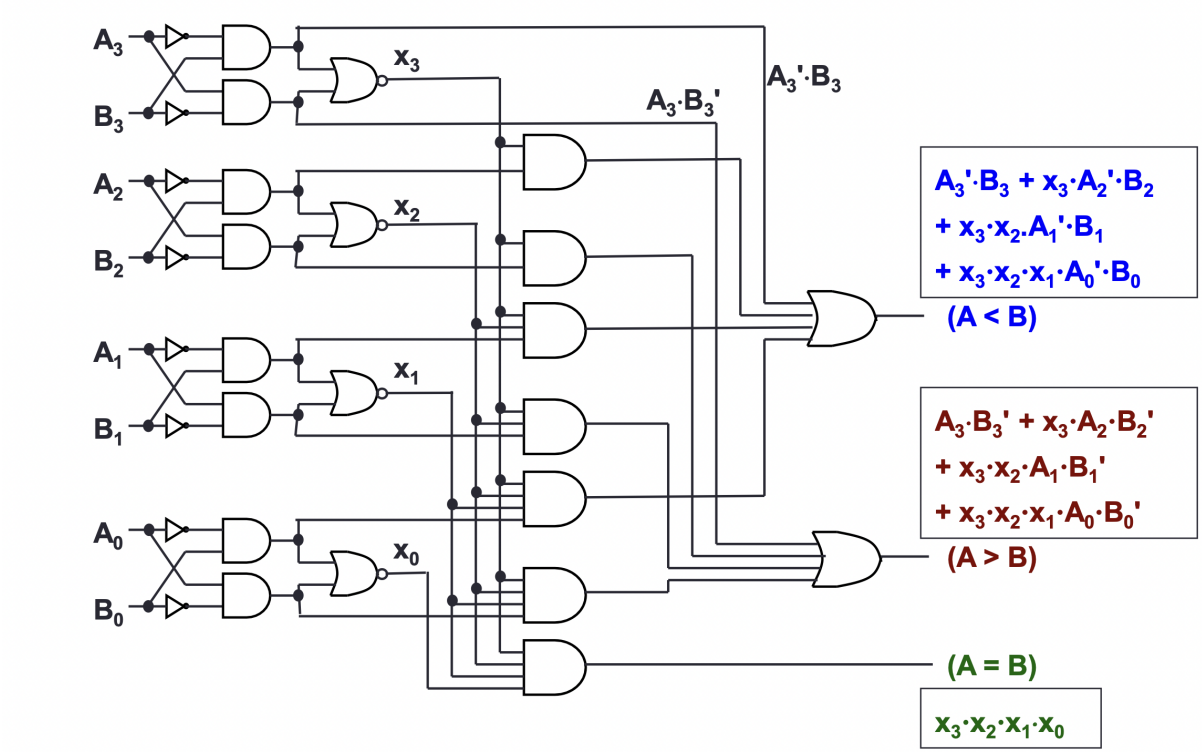


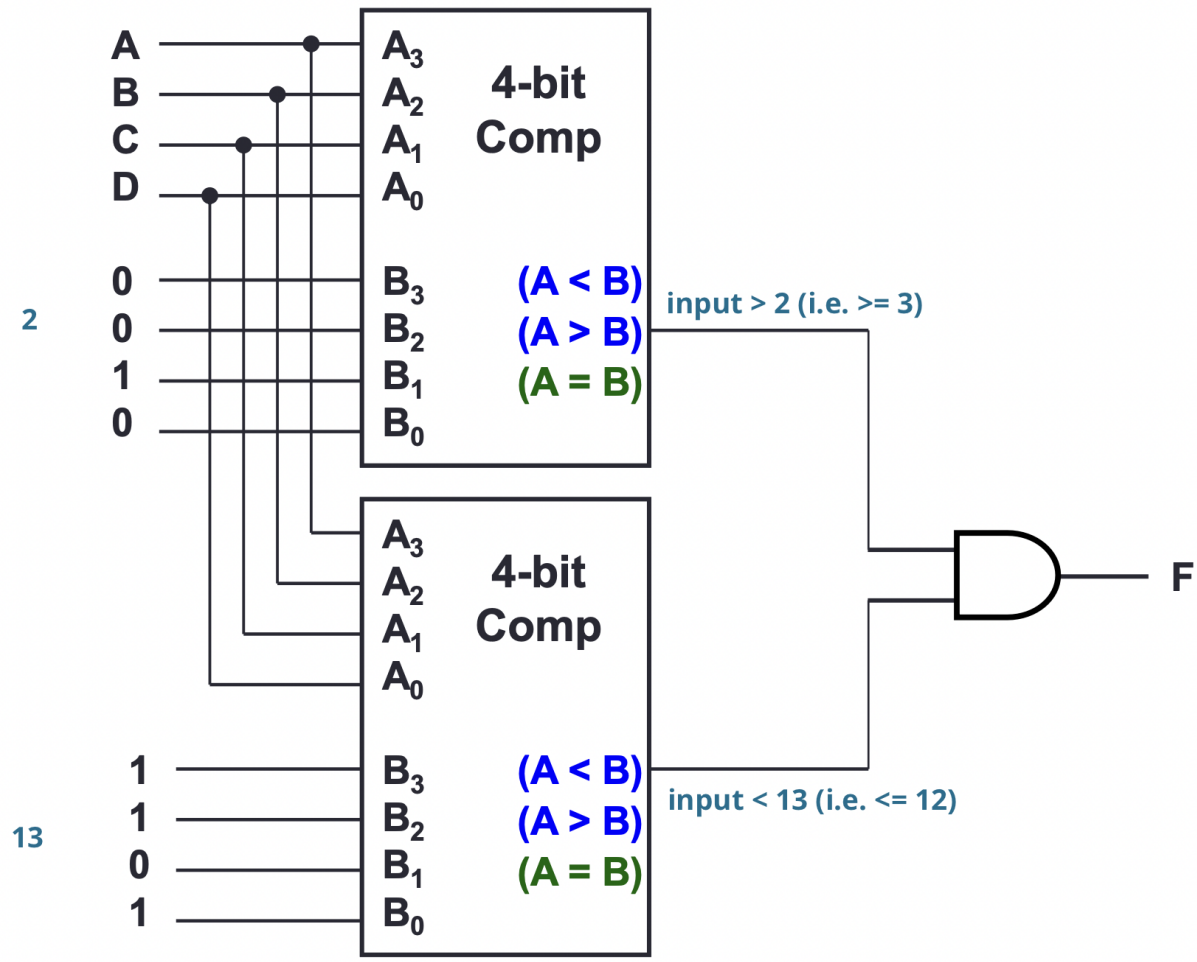
Magnitude Comparator

- compares 2 unsigned values A and B, to check if A>B, A=B, or A<B.
- Compare two 4-bit unsigned (non-negative) integers $A(a_3a_2a_1a_0)$ and $B(b_3b_2b_1b_0)$
  - if $(a_3 > b_3)$, then A > B

- if $(a_3 < b_3)$, then A < B
- if $(a_3 = b_3)$, then if $(a_2 > b_2)$...

Let $A = A_3A_2A_1A_0$, $B = B_3B_2B_1B_0$; $x_i = A_i \cdot B_i + A_i' \cdot B_i'$





returns 1 if 3 ≤ ABCD ≤ 12

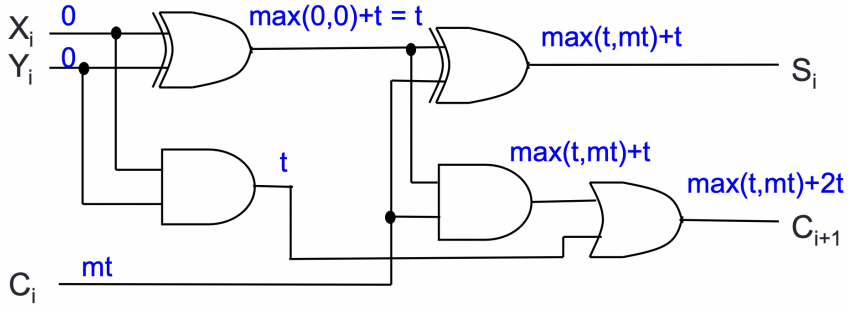
Circuit Delays

Given a logic gate with delay t. If inputs are stable at times $t_1,\ t_2, ..., t_n$, then the earliest time in which the output will be stable is $max(t_1, t_2, ..., t_n) + t$

Analyse the delay for the repeated block.

where X_i, Y_i are stable at 0t, while C_i is assumed to be stable at mt.

Performing the delay calculation:



- When i=1,m=0;S1 =2t and C2 =3t
- When i=2,m=3;S2 =4t and C3 =5t
- When i=3,m=5;S3 =6t and C4 =7t
- When i=4,m=7;S4 =8t and C5 =9t

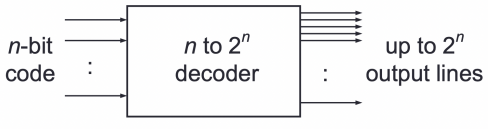
Delay for n-bit ripple-carry parallel adder

- Propagation delay of ripple-carry parallel adders is proportional to the number of bits it handles.
- $S_n = ((n-1) \times 2 + 2)t$
- $C_n = ((n-1) \times 2 + 3)t$ → max delay

L18: MSI Components

Decoder (n → 2^n)

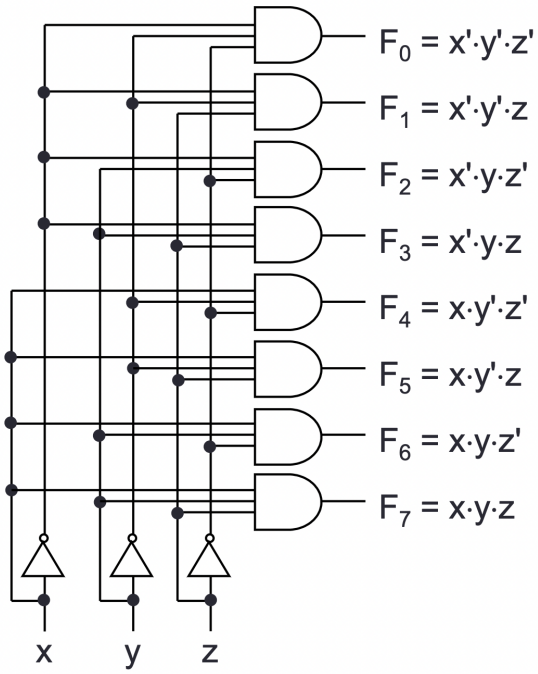
- Given a code, identify entity
- Convert binary information from n input lines to (a maximum of) 2^n output lines.



- Known as n-to-m-line decoder, or simply n:m or $n \times m$ decoder ($m \leq n$).
- May be used to generate 2^n minterms of n input variables
- output has only one 1 and other digits are all 0

- Design a 3×8 decoder.

x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**



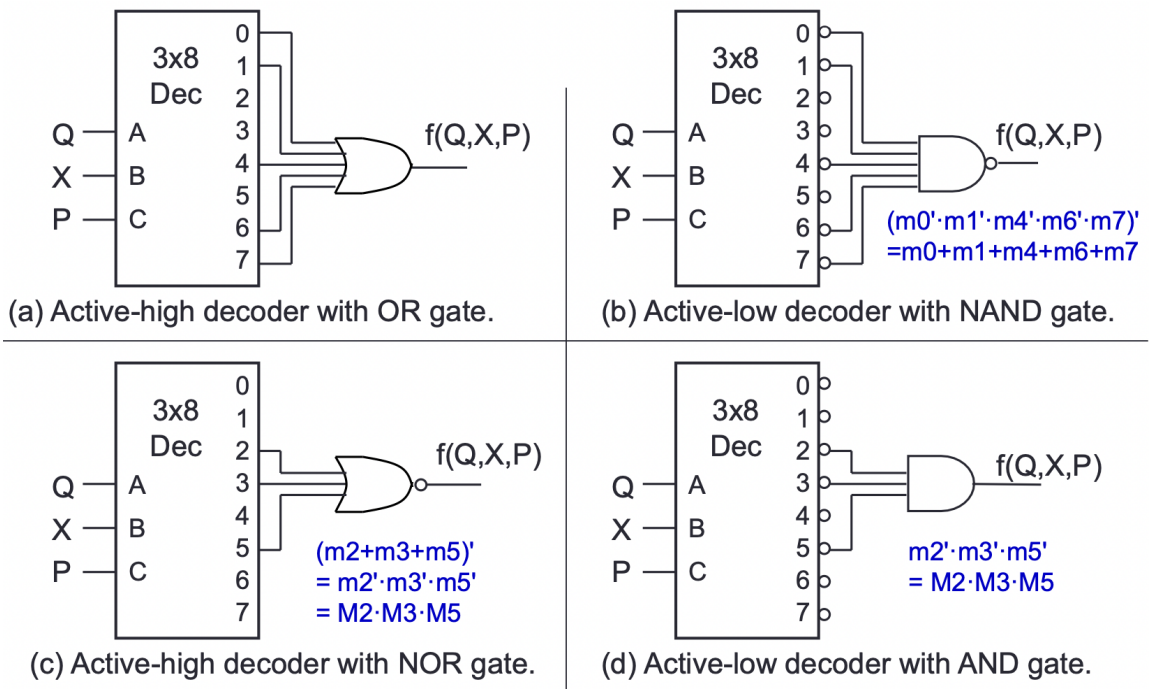
each output is a minterm

- Any combinational circuit with n inputs and m outputs can be implemented with an $n : 2^n$ decoder with m OR gates.

Functions

- A Boolean function, in sum-of-minterms form
  - a decoder to generate the minterms, and
  - an OR gate to form the sum.
- e.g. $f(Q, X, P) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$
  - Using a decoder with active-high outputs with an OR gate: $f(Q, X, P) = m_0 + m_1 + m_4 + m_6 + m_7$

- Using a decoder with active-low outputs with a NAND gate:
  $f(Q, X, P) = (m_0' \cdot m_1' \cdot m_4' \cdot m_6' \cdot m_7')'$
- Using a decoder with active-high outputs with a NOR gate:
  $f(Q, X, P) = (m_2 + m_3 + m_5)' = (M_2 \cdot M_3 \cdot M_5)$
- Using a decoder with active-low outputs with an AND gate:
  $f(Q, X, P) = m_2' \cdot m_3' \cdot m_5'$



(a) Active-high decoder with OR gate.

(b) Active-low decoder with NAND gate.

(c) Active-high decoder with NOR gate.

(d) Active-low decoder with AND gate.

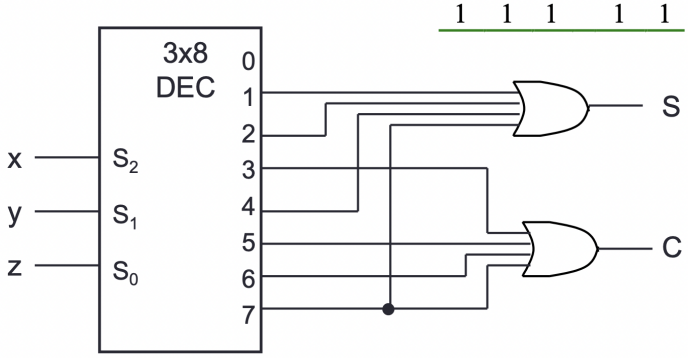
Decoder for Full Adder

- Example: Full adder

  S(x, y, z) = Σ m(1,2,4,7)

  C(x, y, z) = Σ m(3,5,6,7)

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Decoders with Enable

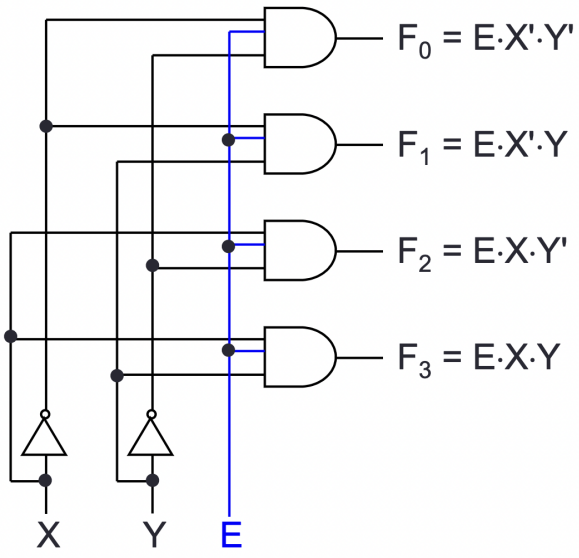
One-Enable

- device is only activated when the enable, E = 1.

Truth table:

E	X	Y	F_0	F_1	F_2	F_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	d	d	0	0	0	0

Circuit of a 2×4 decoder with enable:



Zero-Enable (most MIPS)

- denoted by E' or $\bar{E}$.
- The decoder is enabled when the signal is zero (low).

E	X	Y	F_0	F_1	F_2	F_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	d	d	0	0	0	0

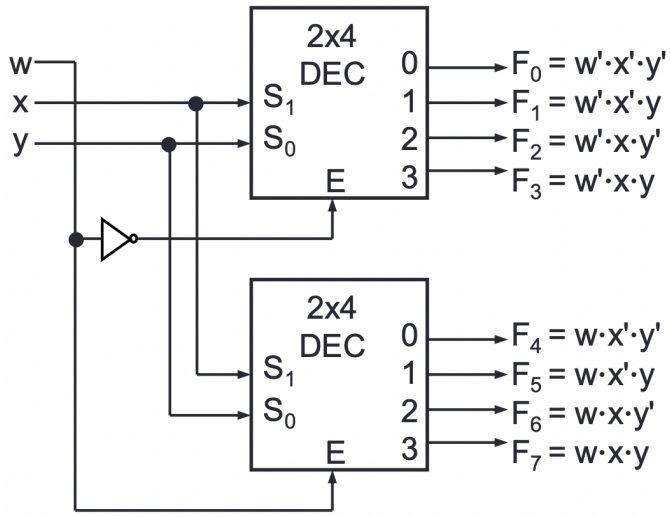
Decoder with 1-enable

E'	X	Y	F_0	F_1	F_2	F_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	d	d	0	0	0	0

Decoder with 0-enable

- Normal outputs = active high outputs (F0 = 1000)
- Negated outputs = active low outputs (F0 = 0111)

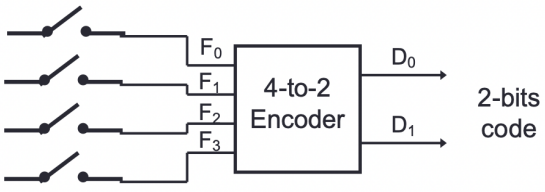
Constructing Larger Decoders from smaller ones



A 3x8 decoder can be built from two 2x4 decoders (with one- enable) and an inverter.

Encoders (2^n → n)

- Given a set of input lines, of which exactly one is high and the rest are low, the encoder provides a code that corresponds to that high input line.
- Contains 2^n (or fewer) input lines and n output lines.



- Implemented with OR gates.

F_0	F_1	F_2	F_3	D_1	D_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1
0	0	0	0	X	X
0	0	1	1	X	X
0	1	0	1	X	X
0	1	1	0	X	X
0	1	1	1	X	X
1	0	0	1	X	X
1	0	1	0	X	X
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X

D0 = F1 + F3; D1 = F2 + F3

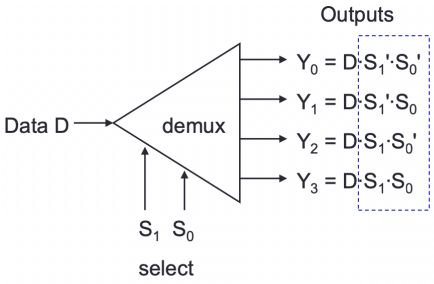
Priority Encoder

- If ≥ 2 inputs or equal to 1, the input with the highest priority takes precedence.
- If all inputs are 0 → invalid.

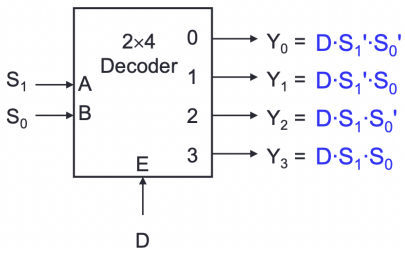
Inputs				Outputs		
D_0	D_1	D_2	D_3	*f*	*g*	*V*
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Demultiplexers (1 → 2^n)

- Given an input line and a set of selection lines, a demultiplexer directs data from the input to one selected output line.
- identical to a decoder with enable

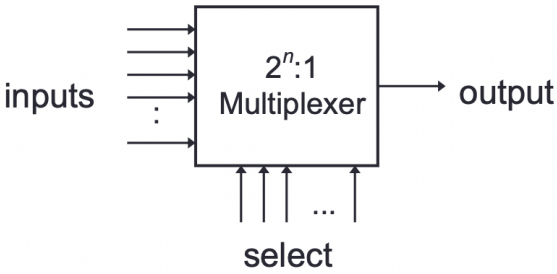


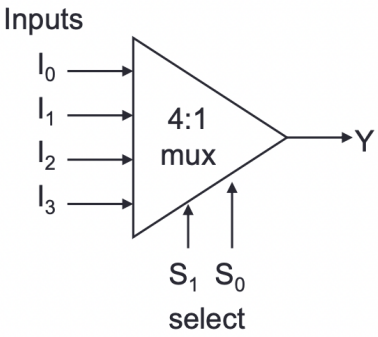
S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D



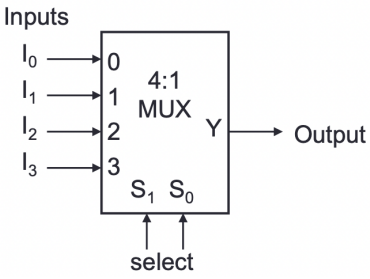
Multiplexer (2^n → 1)

- aka data selector
- selects one of 2^n inputs to a single output line, using n selection lines.





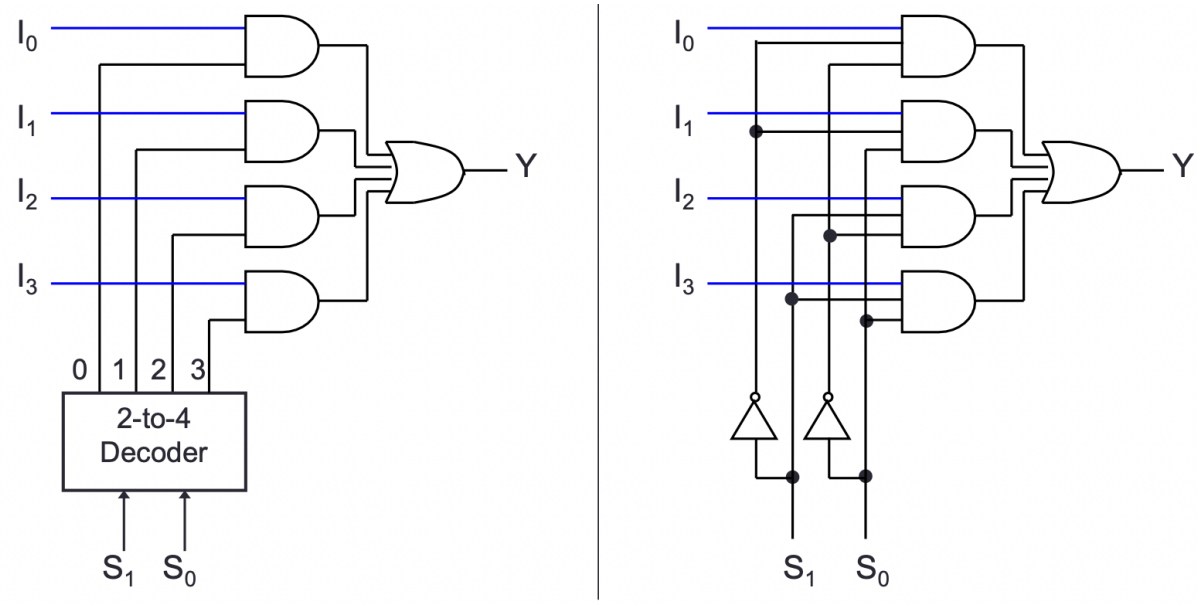
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

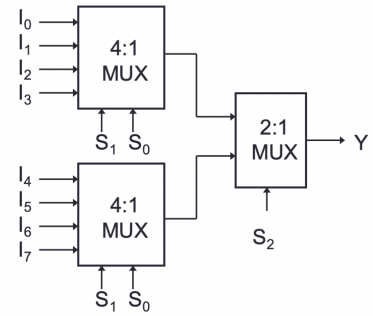
- $Y = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0) = I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3$

- A $2^n - to - 1 - line$ or $2^n : 1$ MUX, is made from an $n : 2^n$ decoder by adding to it 2^n input lines, one to each AND gate.

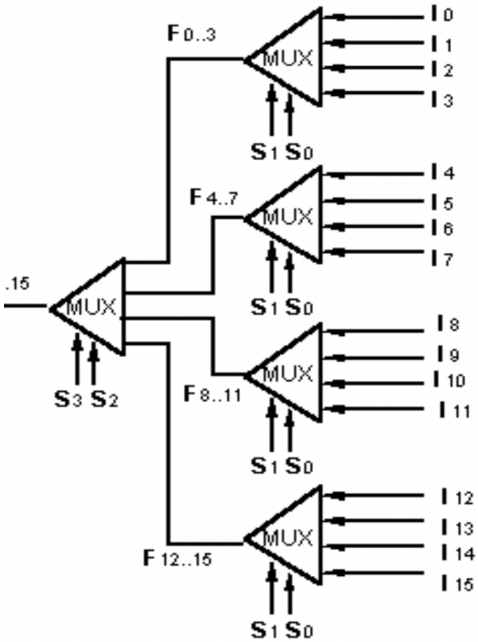


4:1 multiplexer circuit

Constructing Larger MUX from smaller ones



S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7



Converting Larger MUX to a smaller one

1. Express Boolean function in sum-of-minterms form.

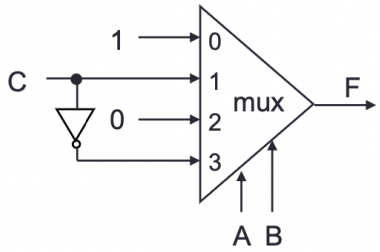
e.g. $F(A, B, C) = \sum m(0, 1, 3, 6)$

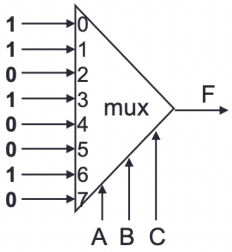
2. Reserve one variable (e.g. MSB) for input lines of multiplexer, and use the rest for selection lines

   Example: C is for input lines; A and B for selection lines.

3. Draw truth table.

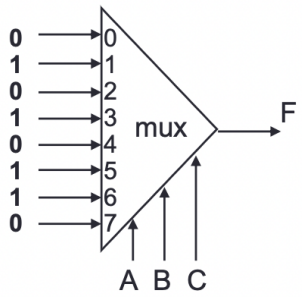
A	B	C	F	MUX input
0	0	0	1	1
0	0	1	1	
0	1	0	0	C
0	1	1	1	
1	0	0	0	0
1	0	1	0	
1	1	0	1	C'
1	1	1	0	





Functions

1. Express in sum-of-minterms form.
   - $F(A, B, C) = A' \cdot B' \cdot C + A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot C' = \sum m(1, 3, 5, 6)$
2. Connect *n* variables to the *n* selection lines.
3. Put a ‘1’ on a data line if it is a minterm of the function, or ‘0’ otherwise.



This method works because:

Output = $I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3 + I_4 \cdot m_4 + I_5 \cdot m_5 + I_6 \cdot m_6 + I_7 \cdot m_7$

Supplying '1' to I_1, I_3, I_5, I_6 , and '0' to the rest:

Output = $m_1 + m_3 + m_5 + m_6$

From slide 34 (4:1 mux)

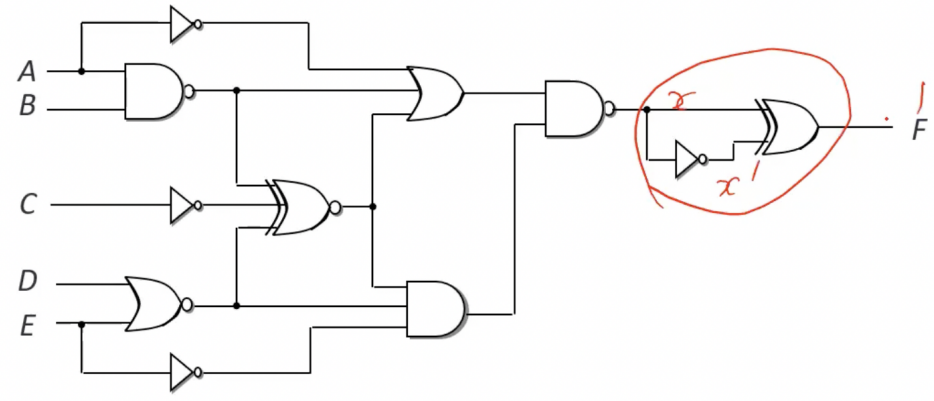
Expressing

$I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$

in minterms notation, it is equal to

$I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3$

Extra Qns



Given a Boolean function with 20 variables.

What is $m12345 \cdot m35790 \cdot m54321$?

given any two minterms which are diff → pdt = 0