

01. C Syntax

github/joyntls

- uninitialised variables will contain random values

```
// Converts distance in miles to kilometres.

#include <stdio.h> /* printf, scanf definitions */
#define KMS_PER_MILE 1.609 /* conversion constant */

int main(void) {
    float miles,           // input - distance in miles
          kms;             // output - distance in kilometres

    /* Get the distance in miles */
    printf("Enter distance in miles: ");
    scanf("%f", &miles);

    // Convert the distance to kilometres
    kms = KMS_PER_MILE * miles;

    // Display the distance in kilometres
    printf("That equals %.2f km.\n", kms);

    return 0;
}

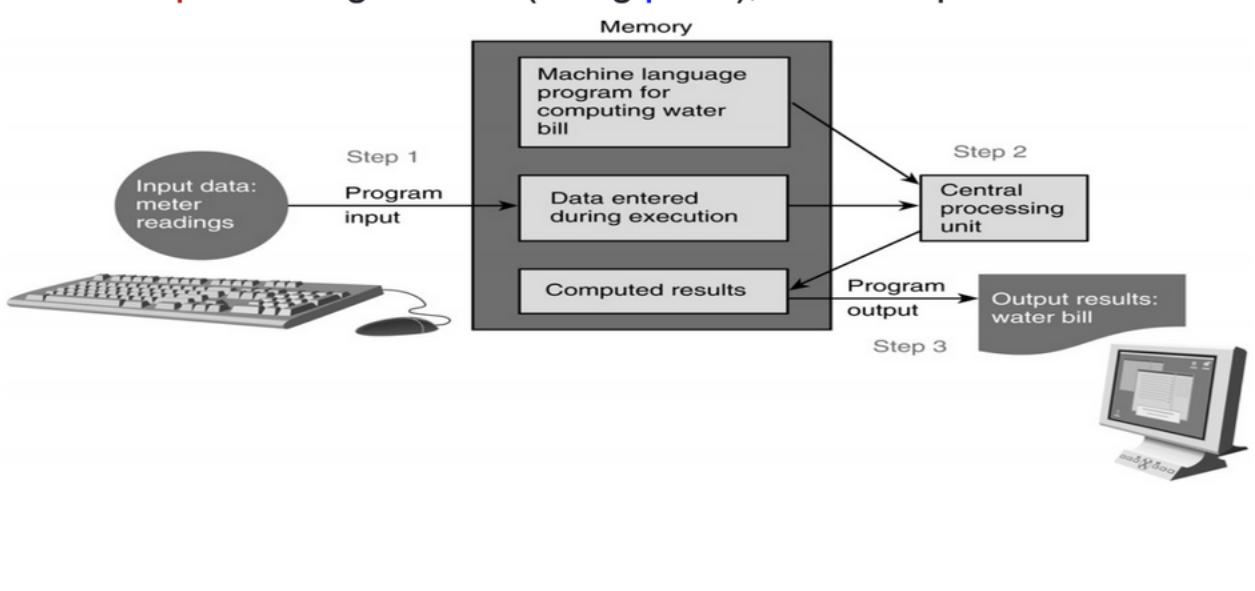
In C, semi-colon (;) terminates a statement.
Curly bracket { } indicates a block.
In Python: block is by indentation
```

Annotations:

- preprocessor directives: `#include <stdio.h>`, `#define KMS_PER_MILE 1.609`
- standard header file: `<stdio.h>`
- constant: `KMS_PER_MILE`
- reserved words: `int`, `main`, `void`, `float`, `scanf`, `printf`
- variables: `miles`, `kms`
- functions: `main`, `printf`, `scanf`
- special symbols: `;`, `{}`, `*`, `/* */`
- hash-defined statement: `#define`
- declaration: `int main(void)`
- statement: `return 0;`
- comment: `/* Get the distance in miles */`
- comment: `/* Converts distance in miles to kilometres. */`
- comment: `/* printf, scanf definitions */`
- comment: `/* conversion constant */`
- comment: `/* Enter distance in miles: */`
- comment: `/* That equals 9.2f km.\n */`
- comment: `/* ANSI C */`
- note: `NOT a variable`
- note: `just a substitution!!`
- note: `Only /* ... */ is ANSI C`
- note: `In C, semi-colon (;) terminates a statement.`
- note: `Curly bracket { } indicates a block.`
- note: `In Python: block is by indentation`

Programme Structure

- A basic C program has 4 main parts:
 - Preprocessor directives: eg: `#include <stdio.h>`, `#include <math.h>`, `#define PI 3.142`
 - Input: through `stdin` (using `scanf`), or file input
 - Compute: through arithmetic operations and assignment statements
 - Output: through `stdout` (using `printf`), or file output



Assignment Statement

- returns the value of the variable assigned

```
a = 12;           // will return 12
z = (a = 12);    // hence this is possible
z = a = 12;      // same thing
```

Operations

- binary operators: `+` `-` `*` `/` `%`
 - left associative → evaluates left to right
 - e.g. $45 / 15 / 2 \Rightarrow 3 / 2 \Rightarrow 1$
 - `%` is remainder (not modulo) $\Rightarrow -10 \% 4 = -2$
- unary operators: `+` `-`
 - right associative → it 'belongs' to the item on its right
 - $-6/3$ is the same as $(-6)/3$ NOT $-(6/3)$

Associativity & Precedence

Operator Type	Operator	Associativity
Primary expression operators	<code>() expr++ expr--</code>	Left to right
Unary operators	<code>* & + - ++expr --expr (typecast)</code>	Right to left
Binary operators	<code>* / %</code> <code>+ -</code>	Left to right
Assignment operators	<code>= += -= *= /= %=</code>	Right to left

switch/case

- fall-through behaviour → if you remove all the `break`s, when the value matches a case, every case afterwards will be run

```
switch (<variable or expression>) {
```

```
  case value1:
```

```
  /* ... */
```

```
  break;
```

```
  case value2:
```

```
  /* ... */
```

```
  break;
```

```
  default:
```

```
  /* executes if <var/exp> not equals value1 or value2 */
```

```
}
```

loops

```
while ( condition )
{
    // loop body
}
```

```
do
{
    // loop body
} while ( condition );
```

```
for ( initialization; condition; update )
{
    // loop body
}
```

Initialization: initialize the **loop variable**

Condition: repeat loop while the condition on **loop variable** is **true**

Update: change value of **loop variable**

02. C Syntax (types & pointers)

 [github/joyntls](https://github.com/joyntls)

Typing

- **strongly-typed** → every variable has to be declared with a data type
- **weakly-typed** → type depends on how the variable is used

Truth Values

- FALSE values: `false` or `0` or `null`
- TRUE values: everything else
 - `true` will be printed as `1`

Data Types

- `int` : 4 bytes, -2^{31} to $2^{31} - 1$
- `float` / `double` : 4 bytes / 8 bytes
- `char` : 1 byte

Mixed-Type Arithmetic

```
int m = 10/4;           // m = 2
int n = 10/4.0;         // n = 2
float p = 10/4;         // p = 2.0, same as float p = (float)(10/4
float q = 10/4.0;       // q = 2.5
```

Type Casting

```
/* syntax: (type) expression */
int ii = 5; float ff = 15.34
float a = (float) ii / 2;    // a = 2.5
float b = (float) (ii / 2); // b = 2.0, floor division
int c = (int) ff / ii;      // c = 3
```

syntax

```
#include <stdio.h>
#define MAX 10

// function prototype:
int readArray(int [], int);

int main(void) {
    int array[MAX];
    readArray(array, MAX);
    return 0;
}

// function definition:
int readArray(int arr[], int n) {
    printf("Enter up to %d integers.\n", n);
    int i;
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
}
```

Placeholder	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float or double	printf
%f	float	scanf
%lf	double	scanf
%e	float or double	printf (for scientific notation)

- `%5d` - integer in a width of 5, right justified
- `%8.3f` - real number in a width of 8, with 3 dp, right justified

 for `scanf`: use format specifier without indicating width, dp, etc

- `%p` specifier: prints address (for pointers)

pointers

- **pointer** → a variable that contains the address of another variable
 - symbol table keeps track of the variable's address

```
int main(void) {
    int a = 3, *b; // b is a pointer to an int
    b = &a;          // b points to the address of a
    *b = 5;          // set a through b, a=5
}
```

- `&` - address operator
 - `&x` → address of the memory cell where the value of `x` is stored
 - gets the address of a variable
- `*` - declares a pointer
 - `type *pointer_name` (e.g. `int *x`)
- `*` - dereferencing (access variable through pointer)
 - `*x = 32`
 - following through the pointer to get the value
- format specifier for pointers: `%p`
- incrementing a pointer: brackets needed - `(*p1)++;`
 - without brackets: increments pointer to next address (depending on size of the data type) aka `+= sizeof(*p1)`

```
double a, *b;
b = &a; // legal
double c, d;
*d = &c; // legal
double e, f;
f = &e; // ILLEGAL!
```

03. Number Systems

Data Representation

- 1 byte = 8 bits
- word = multiple of a byte (e.g. 1 byte, 2 bytes, 4 bytes)
 - 64-bit machine \Rightarrow 1 word is 8 bytes
- N bits can represent up to 2^N values
- to represent M values: $\lceil \log_2 M \rceil$ bits required

Weighted Number systems

- weighted number system \rightarrow has a **base (radix)**
 - base/radix R has weights in powers of R

Prefixes in C

- prefix `0` for octal (e.g. `032` = $(32)_8$)
- prefix `0x` for hexadecimal (e.g. `0x32` = $(32)_{16}$)
- prefix `0b` for binary

Conversion

- decimal to binary
 - whole numbers: repeated **division** by 2, LSB \rightarrow MSB
 - fractions: repeated **multiplication** by 2, MSB \rightarrow LSB
- decimal to base-R:
 - for whole numbers: repeated **division** by R
 - for fractions: repeated **multiplication** by R
- binary \rightarrow octal: partition in groups of 3
- octal \rightarrow binary: convert each digit into 3-bit binary
- binary \rightarrow hexadecimal: partition in groups of 4
- hexadecimal \rightarrow binary: convert each digit into 4-bit binary

ASCII

- American Standard Code for Information Interchange
- 7 bits plus 1 parity bit (for error checking) $\Rightarrow 2^7 = 128$
- in C: `char` datatype is 1 byte = 8 bit integer
 - corresponds to ASCII - can typecast int/char
 - e.g. convert uppercase char to lowercase: `c = c + 'a' - 'A'`

Negative Numbers

- **unsigned** numbers: only non-negative values
- **signed** numbers: include all values (positive and negative)

 for negating non-whole numbers: same as whole numbers (ignore the decimal point, then put it back)

Overflow

- positive + positive = negative, OR
- negative + negative = positive
- 1s addition:
 - if there is a carry out, add 1 to the result (wrap around)

$\begin{array}{r} -2 \\ + -5 \\ \hline -7 \end{array}$	$\begin{array}{r} 1101 \\ + 1010 \\ \hline 10111 \\ + 1 \\ \hline 1000 \end{array}$	
$\begin{array}{r} -3 \\ + -7 \\ \hline -10 \end{array}$	$\begin{array}{r} 1100 \\ + 1000 \\ \hline 10100 \\ + 1 \\ \hline 0101 \end{array}$	

- 2s addition:

- ignore the carry out

 **r's complement of N** = $r^n - N$ for n integer digits

(r-1)'s complement = $(r^n - N) - 1$

Sign-and-Magnitude

- MSB represents the sign (0 is positive)
- **range (8-bit)**: -127_{10} to $+127_{10}$
 - 2 zeroes: `0000000` ($+0_{10}$) and `10000000` (-0_{10})
- **negating a number**: reverse the first bit
- **issues**
 1. there are two zeroes (which may be useful for limits!)
 2. not good for performing arithmetic due to the zero in front

1s Complement

- negated value of x , $-x = 2^n - x - 1$
- **negating a number**: invert the bits
- **range (8-bit)**: -127_{10} to $+127_{10}$
 - 2 zeroes: `0000000` ($+0_{10}$) and `1111111` (-0_{10})
 - **range (n-bits)**: -2^{n-1} to 2^{n-1}

2s Complement

- = 1s complement + 1
- negated value of x , $-x = 2^n - x$
- **negating a number**
 - invert the bits, then add 1
- **range (8-bit)**: -128_{10} to $+127_{10}$
 - zero: `0000000` = $+0_{10}$
 - **range (n-bits)**: -2^{n-1} to $2^{n-1} - 1$

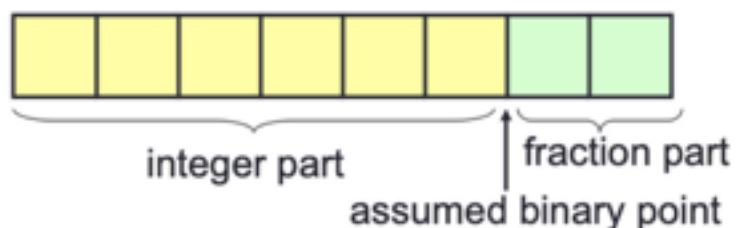
Excess Representation

- `00...00` = -2^n
- `10...00` = 0
- to express n in Excess- M representation: $n + M$

04. Number Representations

Fixed-point representation

- reserve a certain number of bits for the whole number part and the fraction part



- If 2s complement is used, we can represent values like:

$$011010.11_{2s} = 26.75_{10}$$

$$111110.11_{2s} = -000001.01_2 = -1.25_{10}$$

- issues: limited range

Floating-point representation

- IEEE 754 floating-point representation

- exponent is excess-127



single-precision: 1-bit sign / 8-bit exponent / 23-bit mantissa

- 3 components:
 - sign
 - exponent
 - mantissa** (fraction)
- mantissa is **normalised** with an implicit leading bit 1
 - to maximise the numbers to be stored
 - normalise it to $1.xxxx \cdot 2^n \Rightarrow$ the rightmost bit is always 1 \Rightarrow no need to store it
- comparison to fixed-point representation
 - advantage:** much better range and accuracy
 - disadvantage:** more complex representation

05. Arrays, Strings & Structs

 [github/joyntls](https://github.com/joyntls)

Arrays

- a **homogenous** collection of data → data is all of the same type
- declaration: `arr = elementType[size]`
 - `arr` refers to `&arr[0]`
- an array name is a fixed (constant) pointer
 - points to the first element in the array
 - cannot be reassigned - `arr1 = arr2` is illegal!

```
// an array can ONLY be initialised at the time of declaration
int evens[5] = {2, 4, 6, 8, 10};
// if you initialise values, no need to declare length of arr
int odds[] = {1, 3, 5};
// uninitialized values will be zero value
int some[5] = {1, 2, 3}; // some = [1, 2, 3, 0, 0]

int numbers[3];
printf("Enter 3 integers:");
for (i = 0; i < 3; i++) {
    scanf("%d", &numbers[i]);
}
```

in function prototypes

```
// parameter names are optional
int sumArray(int [], int); // valid
int sumArray(int arr[], int size); // valid
int sumArray(int *, int); // pointer is valid too

// size can be specified but will be ignored
int sumArray(int arr[8], int size);

// function definition
int sumArray(int *arr, int size) { ... }
int sumArray(int arr[8], int size) { ... } // size ignored
```

Strings

- array of characters terminated with a null character `\0`
 - ASCII value of 0
- string functions: `#include <string.h>`

```
char my_str[] = "hello";
char my_str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

I/O

- in
 - `fgets(str, size, stdin)` - reads (size - 1) chars, or until newline
 - `scanf("%s", str)` - reads until whitespace
- out
 - `puts(str)` - terminates with newline
 - `printf("%s\n", str)` - prints until it encounters `\0` in `str`

string functions

- `strlen(s)` → returns number of characters in s up to `\0`
- `strcmp(s1, s2)` → compares the ASCII values of corresponding characters
 - returns `s1 - s2`
 - negative number / positive number / 0 if they are equal
- `strncmp(s1, s2, n)` → strcmp for first n characters of s1 and s2
- `strcpy(s1, s2)` → copy s2 into s1
 - cannot directly assign `s1 = "Hello"`
 - can copy: `strcpy(s1, "Hello")`
- `strncpy(s1, s2, n)` → copy first n characters of s2 into s1

Structs

- allow grouping of heterogenous data
- passed by value into functions
 - unless: passing array of structs to a function
 - array members of structs are deeply copied
- can be reassigned

```
// declare BEFORE function prototypes
typedef struct {
    int length, width;
    float height;
} box_t;

typedef struct {
    int id;
    box_t smaller_box;
} nested_box_t;

// initialising struct variables
box_t mybox = {2, 3, 5.1};
nested_box_t big_box = {0, {4, 3, 6.7}};

// accessing members
box.length = 1;
big_box.smaller_box.width = 2;
```

create new types called `box_t` and `nested_box_t`

 **no** memory is allocated to a type

arrow operator: `->`

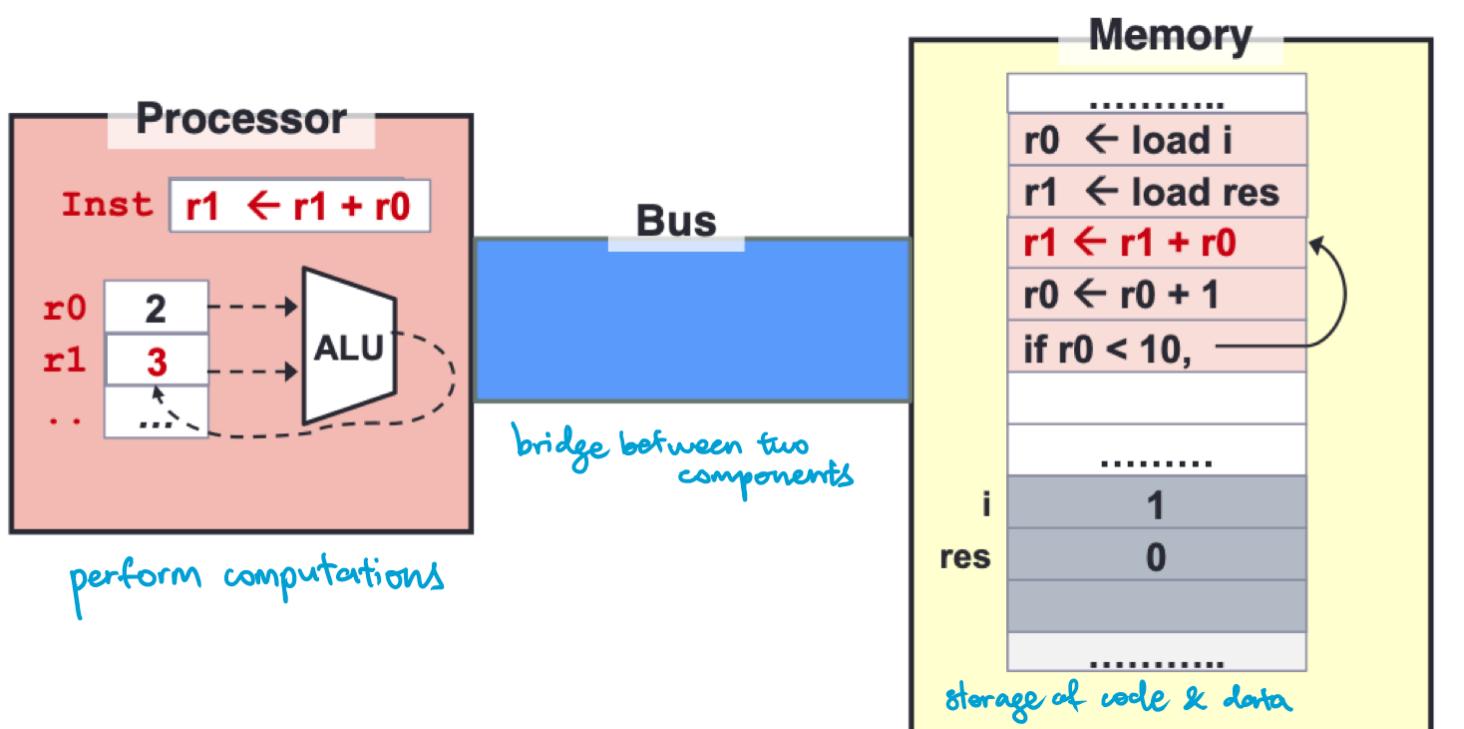
- `(*player_ptr).name` is equivalent to `player_ptr->name`

 `*player_ptr.name` means `*(player_ptr.name)` (dot has higher precedence)

06. MIPS + ISA

Instruction Set Architecture

- **ISA** → abstraction of the interface between the hardware and low-level software
 - software: to be translated into the instruction set
 - hardware: implements the instruction set
- **stored-memory concept** → both **instructions** and **data** are stored in memory
- load-store model → limit memory operations and relies on registers for storage during execution
- C –compiler → Assembly –assembler → Machine code



- major types of assembly instruction:
 - **memory**: move values between memory and registers
 - **calculation**: arithmetic and other operations
 - **control flow**: change the sequential execution (sequence in which instructions are executed)

Registers

Name	Register number	Usage
\$zero	0	Constant value 0
\$v0-\$v1	2-3	Values for results and expression evaluation <small>pass function values out</small>
\$a0-\$a3	4-7	<small>to pass params to functions</small> Arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Program variables
\$t8-\$t9	24-25	More temporaries
\$gp	28	Global pointer <small>try not to touch</small>
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address

\$at (register 1) is reserved for the assembler.

\$k0-\$k1 (registers 26-27) are reserved for the operation system.

07. MIPS Assembly Language

instructions

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

- `add $s0, $s1, $zero` synonymous with `move $s0, $s1`
- to get a "NOT" operation: `nor $t0, $t0, $zero`
- `lui` → load upper immediate (sets the upper 16 bits of the register)

 registers have no type
⇒ contains either data values or memory address

loading large constants

1. use `lui` to set the upper 16 bits (e.g. `lui $t0, 0xAAAA`)
 - lower bits filled with zeroes
2. use `ori` to set the lower-order bits (e.g. `ori $t0, $t0, 0xF0F0`)
 - lower bits will be set

memory instructions

- `lw target, disp(src)` → load contents of Mem[src+disp] to target
- `sw src, disp(target)` → store contents of src to Mem[targ+disp]
- `lb / sb` → Load/Store byte (doesn't need word-align)

control flow

- `bne` → branch if Not Equal (`bne $t0, $t1, label`)
- `beq` → branch if Equal (`beq $t0, $t1, label`)
- `j` → jump unconditionally (`beq $t0, $t1, label`)
- `slt` → set to 1 on less than, else 0 (`slt dest, src1, src2`)

instruction formats

R-format (Register format: `op $r1, $r2, $r3`)

- Instructions which use 2 source registers and 1 destination register
- e.g. `add, sub, and, or, nor, slt, etc`
- Special cases: `srl, sll, etc.`

I-format (Immediate format: `op $r1, $r2, Immd`)

- Instructions which use 1 source register, 1 immediate value and 1 destination register
- e.g. `addi, andi, ori, slti, lw, sw, beq, bne, etc.`

J-format (Jump format: `op Immd`)

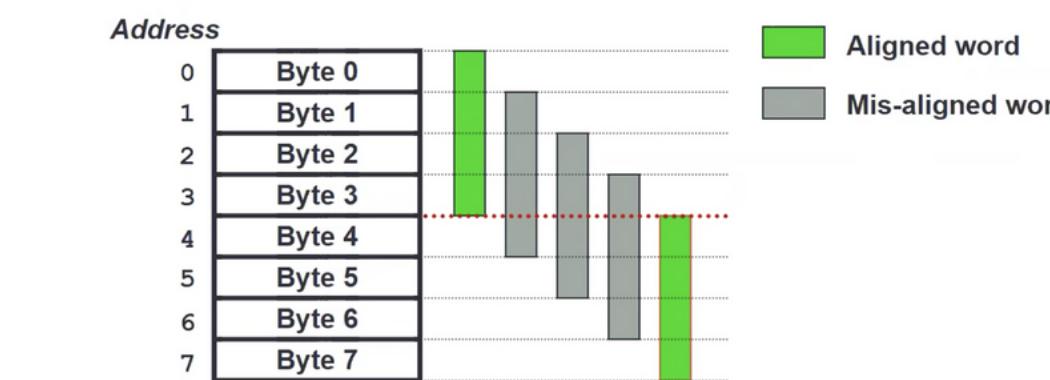
- `j` instruction uses only one immediate value

memory organisation

- each location has an **address** → an index into the array
 - for a k -bit address, the address space is of size 2^k
 - largest address possible: $2^k - 1$
 - bc we start from 0

- **byte addressing** → one byte (8 bits) in every location/address
 - more than one byte → **word addressing**
- load-store architectures can only load data at **word boundaries** (divisible by n bytes)

Example: If a word consists of 4 bytes, then:



MIPS

- load-store register architecture
 - 32 registers, each 32-bit (4 bytes) long
 - each word contains 4 bytes
 - memory addresses are 32-bit long
- 2^{30} memory words ($2^{32}/4$)
 - accessed only by data transfer instructions (aka **memory instructions**)
- MIPS uses **byte addresses** ⇒ consecutive words (word boundaries) differ by 4
 - e.g. `Mem[0], Mem[4], ...`

08. MIPS Instruction Encoding

 [github/joyntls](https://github.com/joyntls)

Instruction Formats

 [MIPS_Reference_Data_page1.pdf](#) 89.7KB

R-Format



each field is a 5/6-bit unsigned integer
opcode always = 0, shamt set to 0 for all non-shift instructions
rs set to 0 for sll/srl

I-Format



immediate is a signed integer 2s complement (up to 2^{16} values)

J-Format



- MIPS will take the 4 MSBs from PC+4 (next instruction after the jump instruction)
- omit 2 LSB since instruction addresses are word-aligned
- maximum jump range $\Rightarrow 2^{26+2+4} = 2^{32}$

PC-Relative Addressing

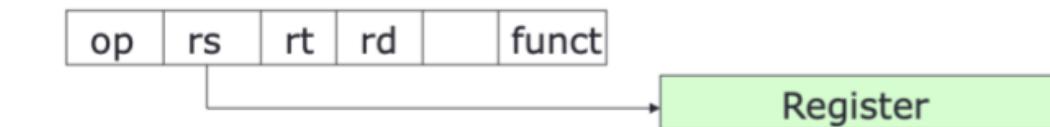
- **Program Counter (PC)** \rightarrow special register that keeps address of the instruction being executed in the processor
- target address = PC + 16-bit **immediate** field
 - can branch $\pm 2^{15}$ words = $\pm 2^{17}$ bytes from the PC
 - interpret **immediate** as the number of words since instructions are word-aligned \Rightarrow larger range!
- next branch calculation:
 - if branch is not taken: PC+4
 - if branch is taken: (PC+4) + (immediate x 4)

Summary

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3	Three operands; data in registers
Data transfer	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
	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
Conditional	branch on equal	beq \$s1, \$s2, 25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
branch	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
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call	

Addressing Modes

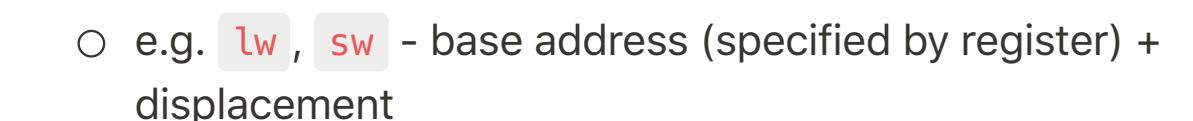
- **register addressing** \rightarrow operands are registers



- **immediate addressing** \rightarrow operand is a constant encoded in the instruction itself (e.g. `ori`, `addi`, `andi`, `slti`)



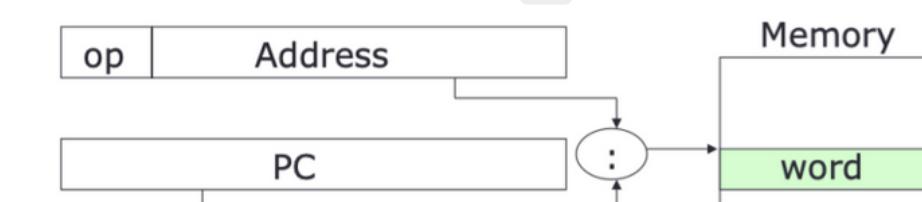
- **base addressing (aka displacement addressing)** \rightarrow operand is at the memory location whose address is the sum of a register and a constant in the instruction



- **PC-relative addressing** \rightarrow address is the sum of PC and constant in the instruction (e.g. `beq`, `bne`)
 - branch address is relative to PC+4



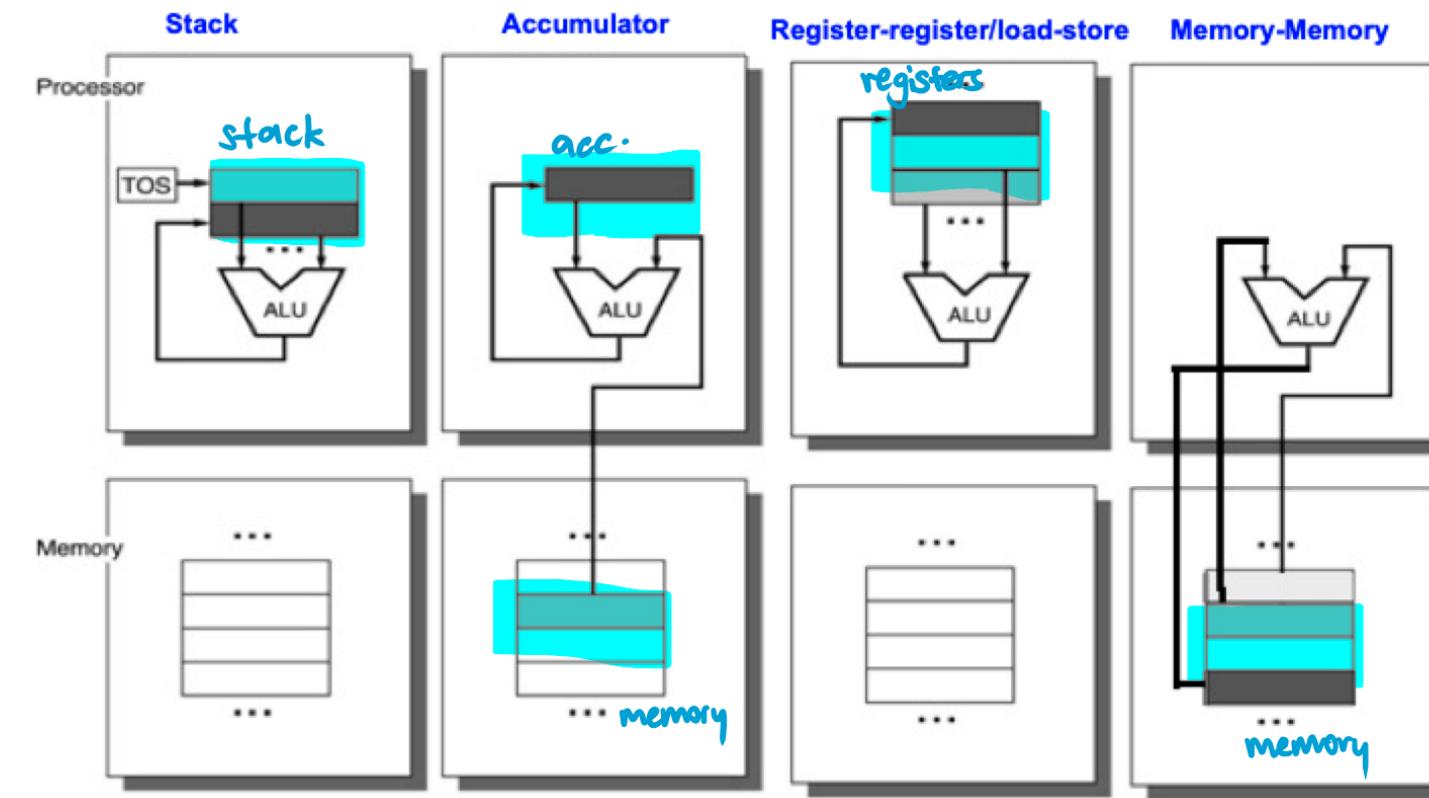
- **pseudo-direct addressing** \rightarrow 26-bit of instruction concatenated with the 4 MSBs of PC (e.g. `j`)



09. Instruction Set Architecture

1. Data Storage

Storage Architectures



ISA	Instructions	Explanation
Stack	push @src pop @dest add	Load value in @src onto top of stack. Transfer value at top of stack to @dest. Remove top two values in stack, add them, and load the sum onto top of stack.
Accumulator	load @src add @src store @dest	Load value in @src into accumulator. Add value in @src and value in accumulator, and put sum back into accumulator. Store the value in accumulator into @dest.
Memory-Memory	add @dest, @src1, @src2	Add values in @src1 and @src2, and put the sum into @dest.
Register-Register	load \$reg, @src add \$dest, \$src1, \$src2 store \$reg, @dest	Load value in @src into \$reg. Add values in \$src1 and \$src2, and put sum into \$dest. Store value in \$reg into @dest.

2. Memory Addressing Modes

- **endianess** → the relative ordering of bytes in a multiple-byte word stored in memory
 - **big endian** → MSB stored in lowest address
 - **small endian** → LSB stored in lowest address
- **addressing mode** → ways to specify an operand in an assembly
- 3 addressing modes in MIPS
 1. **register** → operand is in a register (e.g. `add $t1, $t2, $t3`)
 2. **immediate** → operand is specified directly in the instruction (e.g. `addi $t1, $t2, 98`)
 3. **displacement** → operand is in memory with address calculated as base + offset (e.g. `lw $t1, 20($t2)`)
 - a form of immediate mode instruction

3. Operations in the Instruction Set

- every instruction set should have a set of standard operations

in general, covered in RISC + CISC	
Data Movement	load (from memory) store (to memory) memory-to-memory move $\$t1 \rightarrow \$t0$ register-to-register move <code>add \$t0, \$t1, \$zero</code> input (from I/O device) output (to I/O device) <code>push, pop</code> (to/from stack) } in MIPS
Arithmetic	integer (binary + decimal) or FP <code>add, subtract, multiply, divide</code>
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control flow	Jump (unconditional), Branch (conditional)
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, move, compare
Graphics	pixel and vertex operations, compression/decompression

4. Instruction Formats

instruction length

- fixed-length instructions
 - ✓ easy fetch and decode
 - ✓ simplified pipelining and parallelism
 - ✓ instruction bits are scarce
- variable-length instructions
 - ✗ require multiple steps to fetch and decode instructions
 - ✓ more flexible

instruction fields

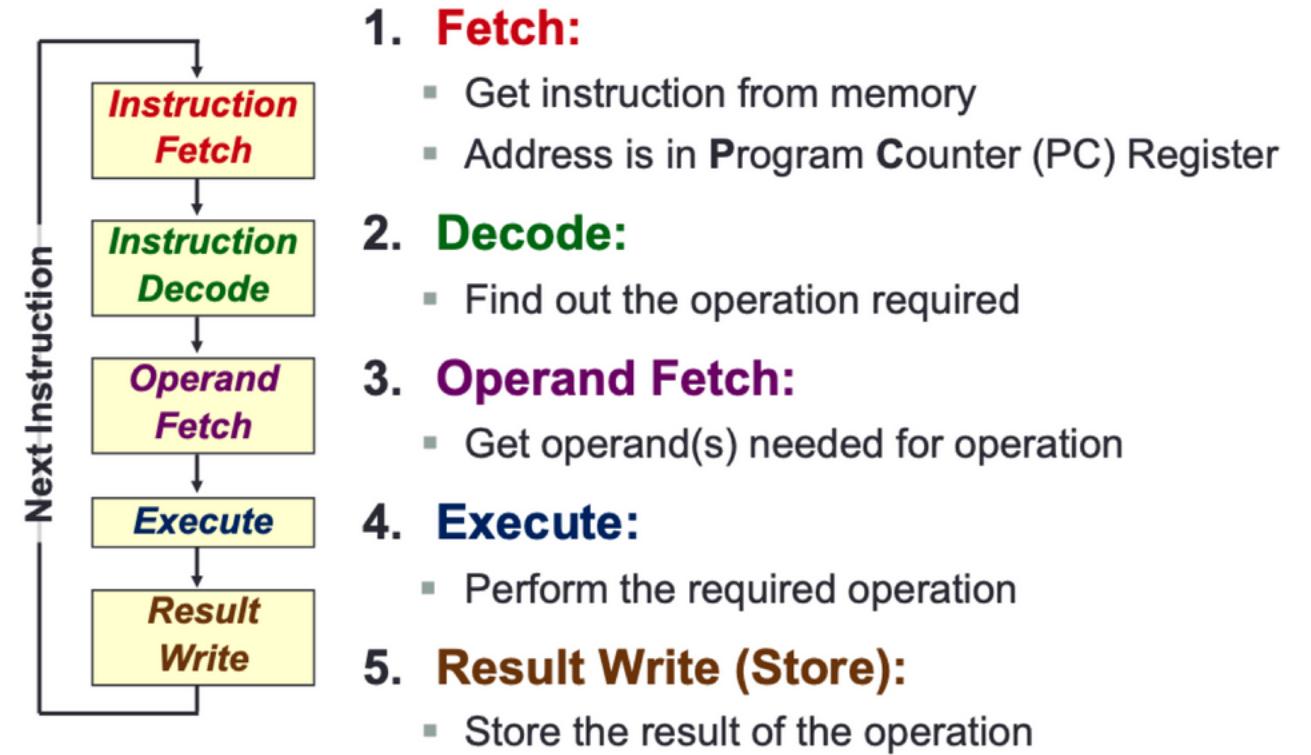
- type and size of operands (i.e. how to divide up the instructions)
- instruction costs of:
 - **opcode** → unique code to specify the desired operation
 - designates the type and size of operands
 - **operands** → zero or more additional information needed for the instruction
- 32-bit architecture should support
 - 8-, 16-, 32-bit integer operations
 - 32- and 64-bit floating point operations

5. Instruction Encoding

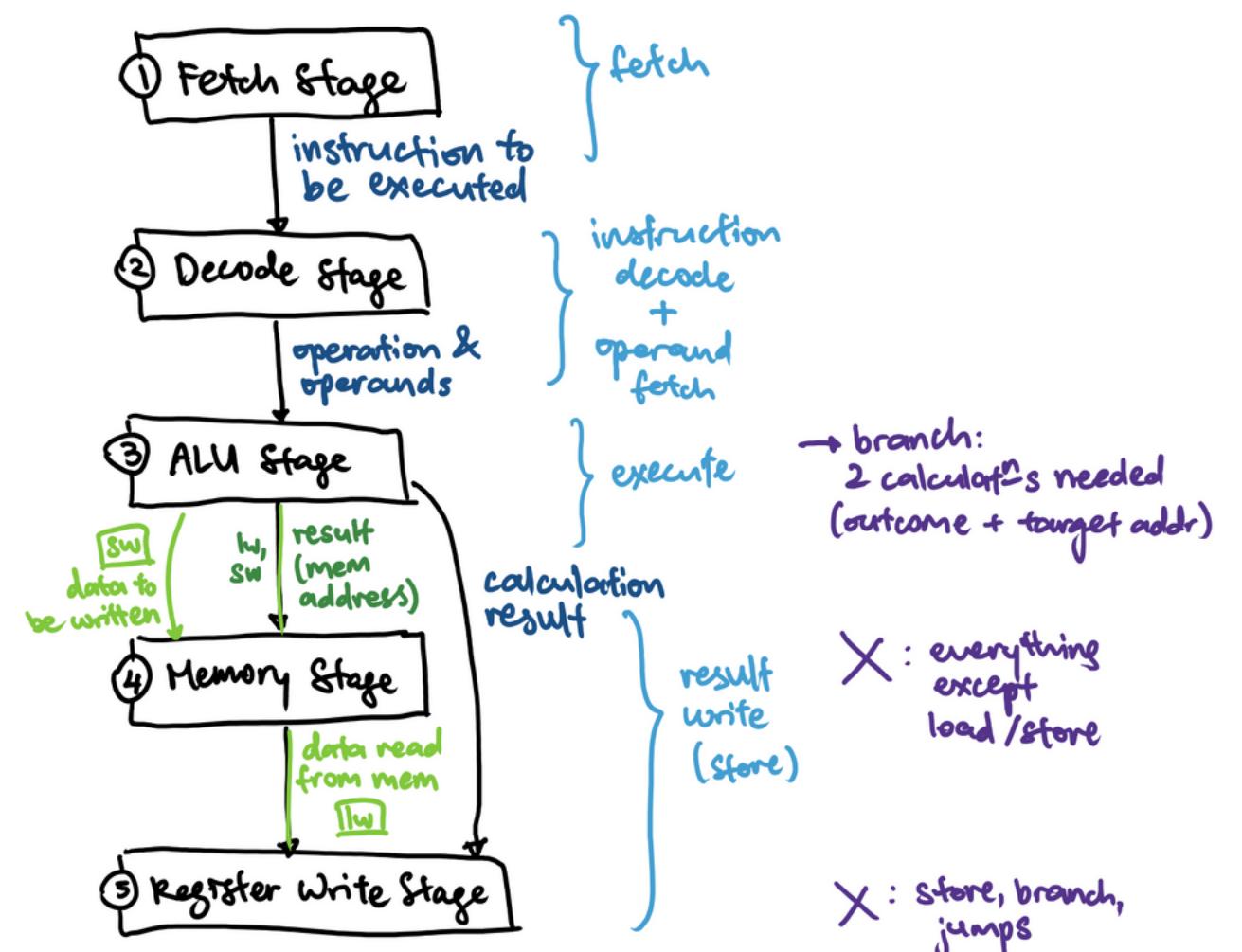
- choice of variable/fixed/hybrid encoding
- **expanding opcode scheme** → opcode variable lengths for different instructions ⇒ maximise instruction bits
 - use unused bits to define opcode ⇒ larger instruction set

10. Datapath

Instruction Execution Cycle



in MIPS



overview

- programmer writes program in high-level language (e.g. C)
- compiler translates to assembly language (MIPS)
- assembler translates to machine code (binaries)
- processor executes machine code (binaries)

two main components of a Processor

- datapath → collection of components that process data
 - takes in data from operands, processes it, writes the data back
 - performs the arithmetic, logical, memory operations
- control → tells datapath, memory and I/O devices what to do according to program instructions
 - generates control signals

1. Fetch Stage

- use PC to fetch instruction from memory
- increment PC by 4 to get the next instruction (using an Adder)
- output (to Decode): instruction to be executed

2. Decode Stage

- gathers data from the instruction fields
 - read opcode and determine the instruction type and field lengths
 - read data from all necessary registers
- output (to ALU): operation and the necessary operands

3. ALU (execution) Stage

- output (to memory stage): calculation result

4. Memory Stage

- only `load` and `store` instructions needed to perform operations in this stage
 - uses memory address calculated by ALU stage (input)
- all other instructions are idle in this stage
 - result from ALU stage will pass through this stage to be used in Register Write stage
- inputs:
 - computation result to be used as memory address
 - register value to be written to memory (only `sw`)
- outputs (to Register Write stage): result to be stored (only `lw`)

5. Register Write Stage

- write the result of some computation into a register
 - do nothing: stores / branches / jumps
- input:
 - destination register number
 - computation result (from either memory or ALU)

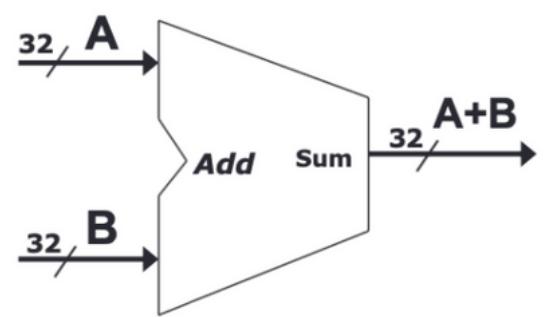
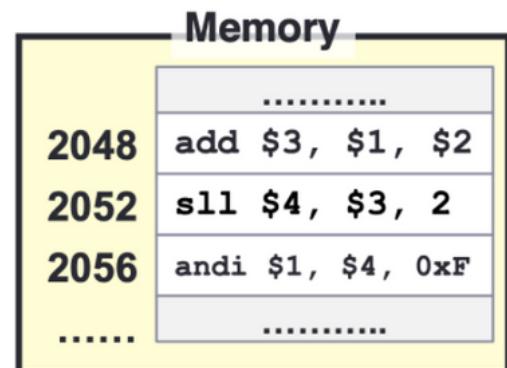
11. Elements of the Datapath

Instruction Memory [1]

- stores instructions (sequential circuit)
- supplies instructions given an address
 - input: instruction address M
 - outputs: contents of address M (binary pattern of instructions)

Adder [1]

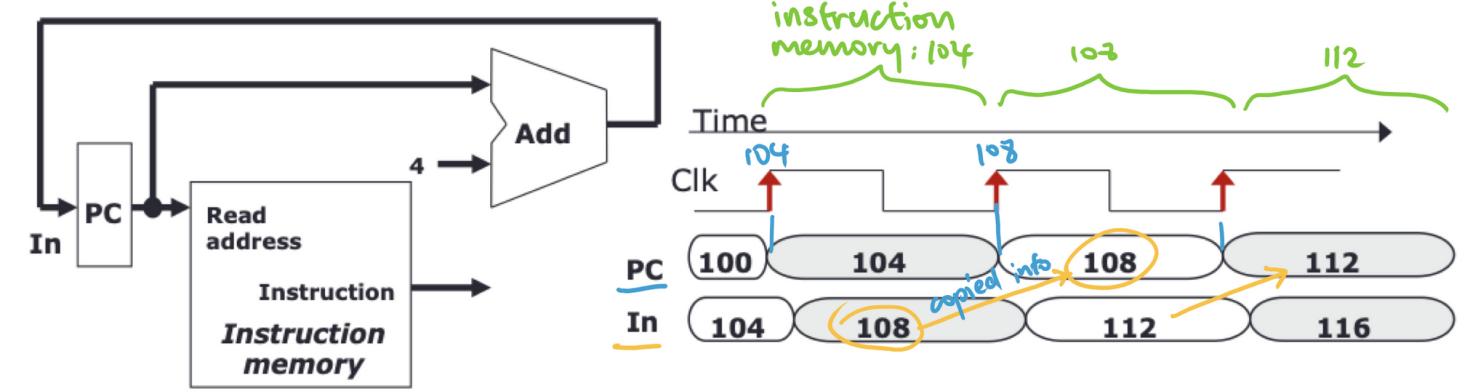
- combinational logic to implement addition of 2 numbers



Memory | Adder

Clock [1]

- a square wave used by the processor
 - times operations inside the processor (e.g. reading & updating PC)
- allows us to read and update the PC at the same time
 - PC is read during the first half of the clock period
 - PC is updated only at the rising edge

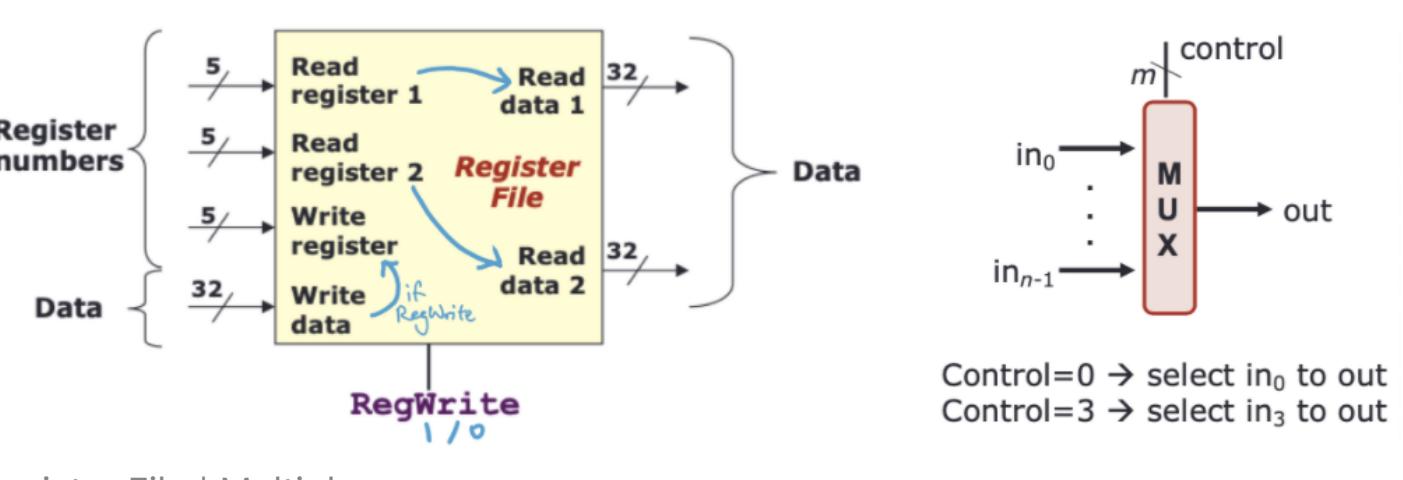


Register File [2]

- collection of 32 registers (each 32 bits wide)
 - can be read by specifying register number
- **read** at most 2 registers per instruction
- **write** at most one register per instruction
- **RegWrite** → control signal to indicate writing of register

Multiplexer

- selects one input from multiple input lines
 - inputs: n lines of same width
 - outputs: select i^{th} input line if control = i
 - control: m bits where $n = 2^m$



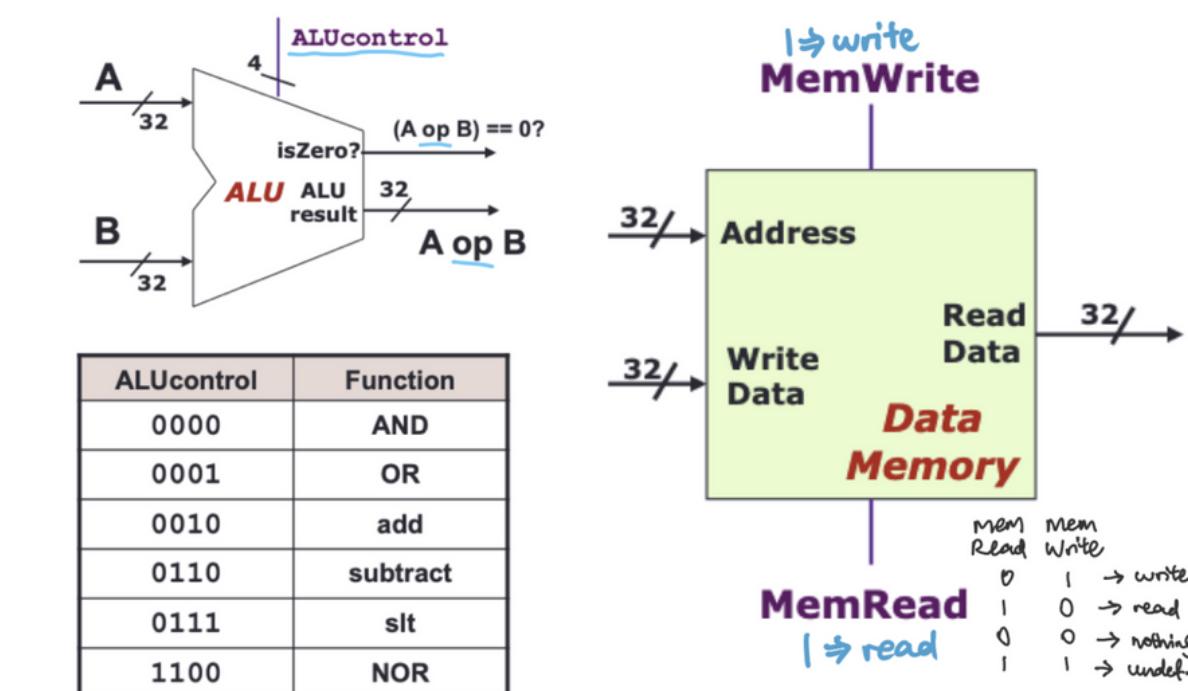
Register File | Multiplexer

ALU [3]

- combinational logic to implement arithmetic and logical operations
- inputs: two 32-bit numbers
- outputs:
 - result of arithmetic/logical operation
 - 1-bit signal to indicate whether result is zero
- control (**ALUcontrol**): 4-bit to decide the operation
 - set using opcode + funct field
- 2 calculations needed for branch instructions (branch outcome + branch target address)

Data Memory [4]

- storage element for the data of a program
- inputs: memory address
 - & data to be written (for store instructions)
- outputs: data read from memory (for load instructions)
- control: **MemRead** and **MemWrite** controls
 - only one can be asserted at any point in time



ALU | Data Memory

12. Control

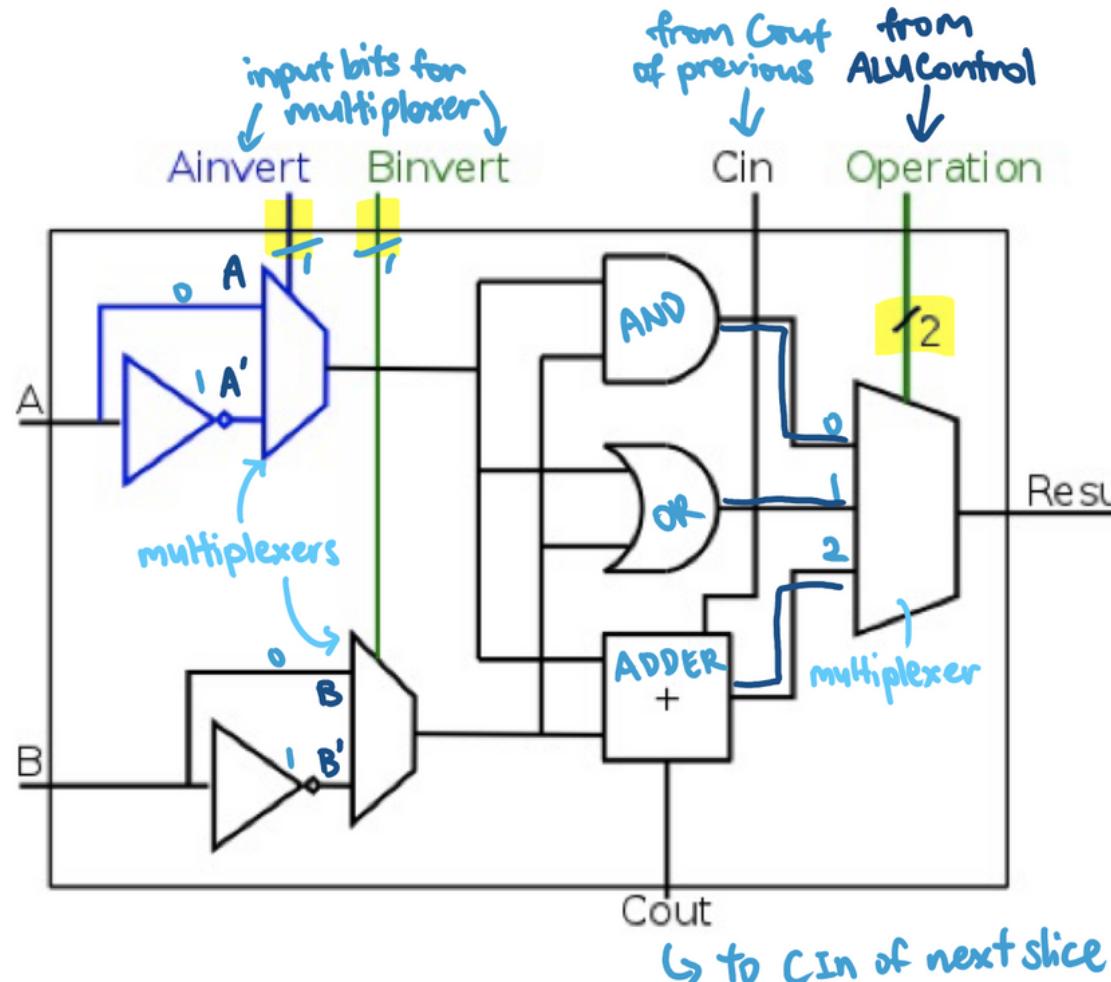
Control Signals

(can all be generated using opcode directly)

- **RegDst** @ Decode/Operand Fetch
 - 0/1 → write register = `Inst[20:16]` / `Inst[15:11]`
- **RegWrite** @ Decode/Operand Fetch
 - 0/1 → No register write / WD written to WR
- **ALUSrc** @ ALU (determines first input)
 - 0 → `Operand2 = Register Read Data 2`
 - 1 → `Operand2 = SignExt(Inst[15:0])` (sign ext immediate)
- **MemRead** @ Memory
 - 0/1 → no read / reads memory using Address (returned in RD)
- **MemWrite** @ Memory
 - 0/1 → no write / writes Register RD 2 into mem[Address]
- **MemToReg** @ RegWrite
 - 0/1 → register write data = ALU result / memory read data
- **PCSrc** @ Memory/RegWrite
 - 0/1 → next PC = $PC + 4$ / $PC = \text{SignExt}(Inst[15:0]) \ll 2 + (PC + 4)$
 - PCSrc = set to 1 if Branch AND is0 are both 1
 - aka (isBranchInstruction AND branchIsTaken)

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2 nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

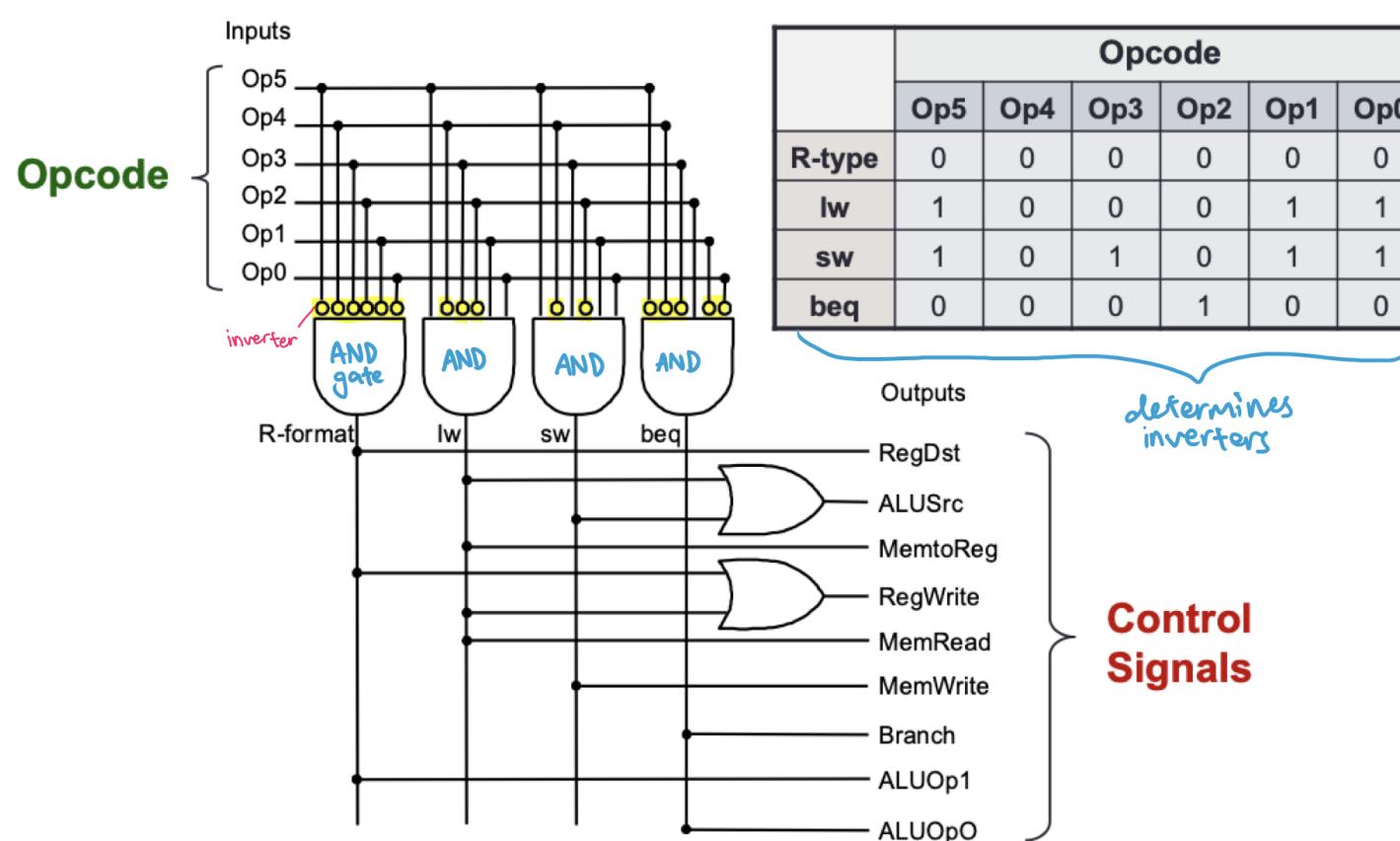
ALU



ALU operation controlled by 2-bit ALUControl

Controller Design

- determines Control Signals from opcode



Multilevel Decoding

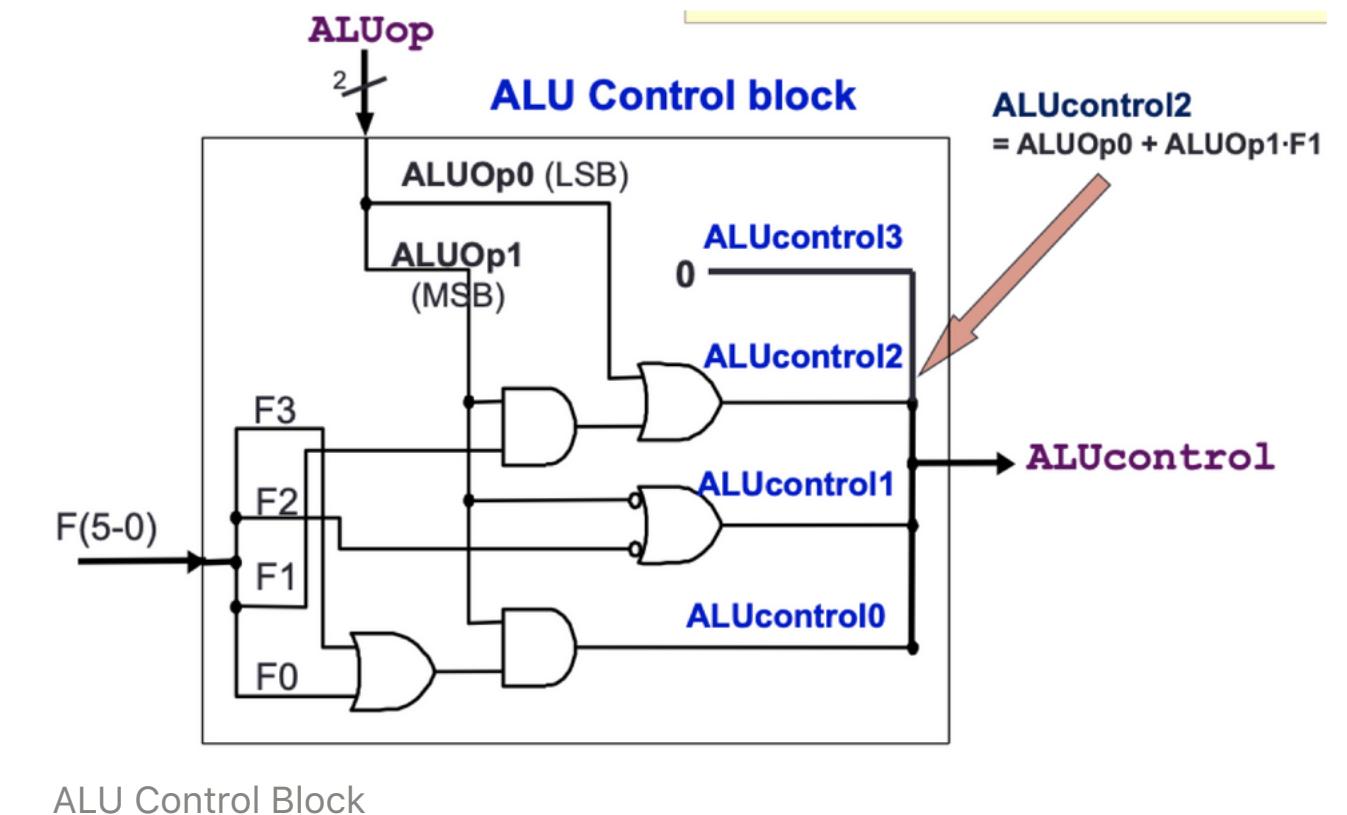
- to determine ALUControl signal
 - depends on 12 variables (6-bit opcode + 6-bit funct)
- reduce the number of cases, then generate the full output
 - reduce the size of the main controller - simplify design process
- how it works
 1. use opcode to generate 2-bit **ALUop** signal
 2. use **ALUop** signal and funct (for R-type) to generate 4-bit **ALUcontrol** signal

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	X	add	0010
sw	00	store word	X	add	0010
beq	01	branch equal	X	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



ALU Control Block

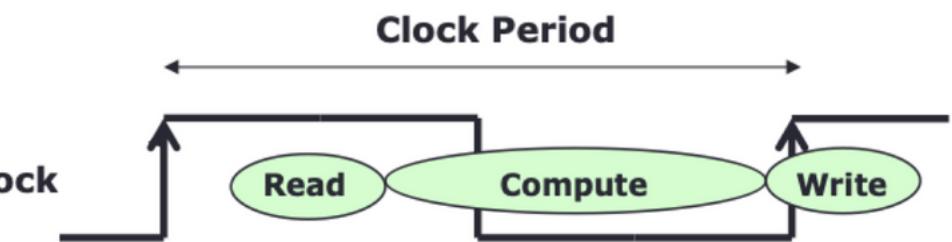
13. Instruction Execution

Instruction Execution

- coordinating the stages together: fetch, decode, memory, write etc
- 2 ways:
 - single cycle implementation
 - multicycle implementation

Single Cycle Implementation

- how it works
 1. read contents of one or more storage elements
 2. perform computation through some combinational logic
 3. write results to one or more storage elements (register/memory)
- all performed within a clock period
 - avoids reading a storage element when it's being written
- time taken depends on slowest instruction
- **disadvantage**
 - clock cycle must be long enough to accommodate the slowest instruction ⇒ all instructions will take the same time as the slowest instruction



Multicycle Implementation

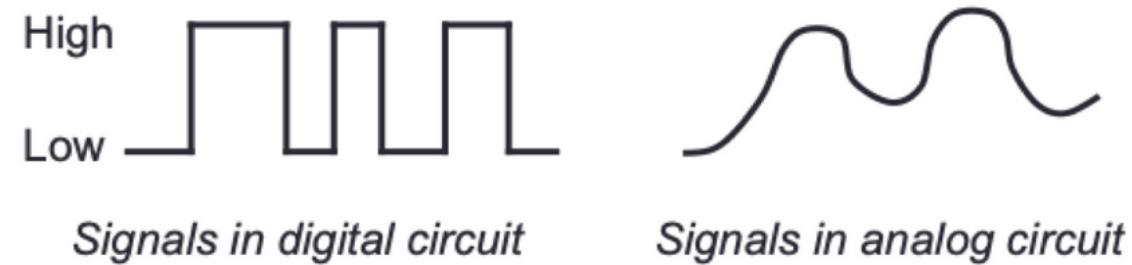
- how it works: break up the instruction into execution steps
 1. instruction fetch
 2. instruction decode and register read
 3. ALU operation
 4. memory read/write
 5. register write
- each execution step takes one clock cycle
- time taken depends on number of steps
 - cycle time is determined by the slowest step
- **disadvantage**
 - may not necessarily be faster - depends on mix of instructions

14. Boolean Algebra

 [github/joyntls](https://github.com/joyntls)

Digital Circuits

- combinational \rightarrow no memory, output depends solely on input
- sequential \rightarrow with memory, output depends on both input and current state =



Boolean Algebra

- connectives in order of precedence:
 - negation $A' \Rightarrow$ equivalent to \neg
 - conjunction $A \cdot B \Rightarrow$ equivalent to \wedge
 - disjunction $A + B \Rightarrow$ equivalent to \vee

Duality

- duality** \rightarrow if the AND/OR operators and identity elements 0/1 are interchanged in a boolean equation, it remains valid
- e.g. the 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 $x + 1 = 1$ is valid, then its dual $x \cdot 0 = 0$ is also valid.

Functions

- boolean functions: e.g. $F1(x, y, z) = x \cdot y \cdot z'$
 - to prove that $F1 = F2$: use truth table
- complement function** \rightarrow given boolean function F , the complement of F , denoted F' , is obtained by interchanging 1 with 0 in the function's output values.

Boolean Algebra Laws

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 *	
$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive laws	
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$

Idempotency	
$X + X = X$	$X \cdot X = X$
One element / Zero element	
$X + 1 = 1$	$X \cdot 0 = 0$
Involution	
$(X')' = X$	
Absorption 1	
$X + X \cdot Y = X$	$X \cdot (X + Y) = X$
Absorption 2	
$X + X' \cdot Y = X + Y$	$X \cdot (X' + Y) = X \cdot Y$
DeMorgans' (can be generalised to more than 2 variables)	
$(X + Y)' = X' \cdot Y'$	$(X \cdot Y)' = X' + Y'$
Consensus	
$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$

left/right equations are duals of each other

Standard Forms

- literals** \rightarrow a boolean variable on its own or its complemented form
- product term** \rightarrow a single literal or a logical product (AND) of several literals
- sum term** \rightarrow a single literal or a logical sum (OR) of several literals
- sum-of-products (SOP) expression** \rightarrow a product term or a logical sum of several product terms
- product-of-sums (POS) expression** \rightarrow a sum term or logical product of several sum terms

Minterms and Maxterms

- minterm** (of n variables) \rightarrow a product term that contains n literals from all the variables; denoted $m0$ to $m[2^n - 1]$
- maxterm** (of n variables) \rightarrow a sum term that contains n literals from all the variables; denoted $M0$ to $M[2^n - 1]$

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

$m0 =$ complement of $M0$, and vice versa

Canonical Forms

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

$$F1 = x \cdot y \cdot z' = m6$$

$$F2 = m1 + m4 + m5 + m6 + m7 = \Sigma m(1,4,5,6,7) \text{ or } \Sigma m(1,4-7)$$

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

canonical/normal form \rightarrow unique form of representation

- sum-of-minterms**: canonical SOP
- product-of-maxterms**: canonical POS

15. Logic Circuits & Simplification

Logic Gates

	Symbol set 1	Symbol set 2 (ANSI/IEEE Standard 91-1984)
AND $A \cdot B$		
OR $A + B$		
NOT A'		
NAND $(A \cdot B)'$		
NOR $(A + B)'$		
EXCLUSIVE OR $(X \oplus Y)$ $A \oplus B$		
XNOR $(A \oplus B)'$		

Universal Gates

- universal gate → can implement a complete set of logic
- {AND, OR, NOT} are a complete set of logic
 - AND/OR/NOT are sufficient for building any boolean function

SOP and POS

- an SOP expression can be easily implemented using
 - 2-level AND-OR circuit
 - 2-level NAND circuit
- a POS expression can be easily implemented using
 - 2-level OR-AND circuit
 - 2-level NOR circuit

Algebraic Simplification

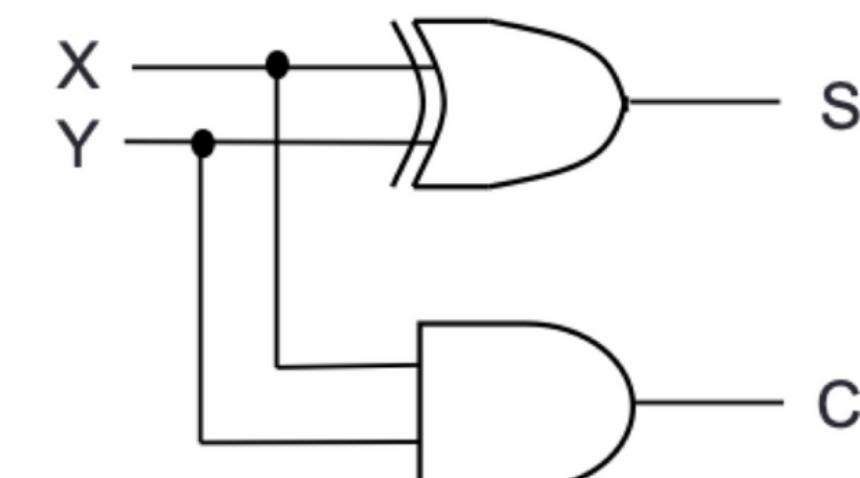
- aims to minimise
 - number of literals (prioritised over number of terms)
 - number of terms

Half Adder

- half adder → a circuit that adds 2 single bits (X, Y) to produce a result of 2 bits (C, S).
 - $C = X \cdot Y$; $S = X + Y$

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

outputs



implementation of a half adder

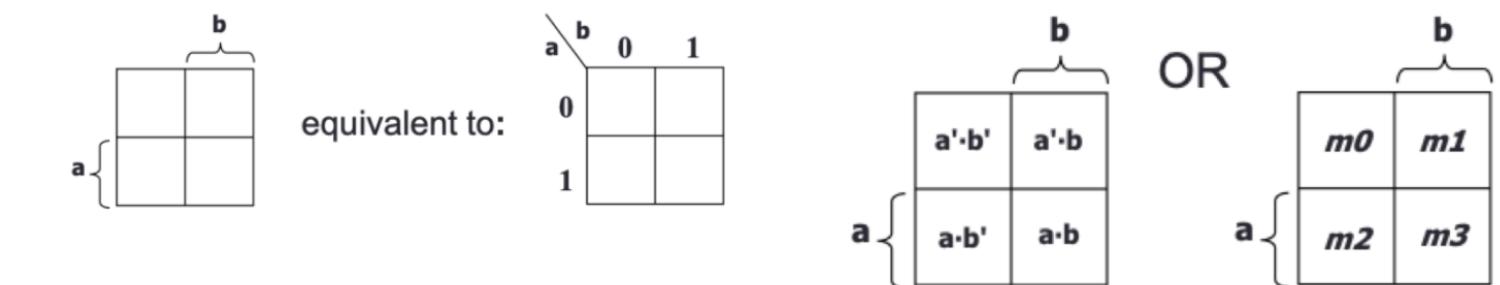
Gray Code

- only a single bit change from one code value to the next

Decimal	Binary	Gray Code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

- not restricted to decimal digits → n bits can have up to 2^n values
- aka reflected binary code
- not unique - there are multiple possible Gray code sequences

K-Maps



- each valid grouping of adjacent cells containing '1' corresponds to a simpler product term
 - group must have size in powers of 2
 - grouping 2^n adjacent cells eliminates n variables
- tips
 - larger group = fewer literals in the resulting product term
 - fewer groups = fewer product terms in simplified SOP expression

(E)PIs

- implicant → a product term that could be used to cover minterms of the function
- prime implicant → a product term obtained by combining the maximum possible number of minterms from adjacent squares in the map
- essential prime implicant → a prime implicant that includes at least one minterm that is not covered by any other prime implicant

K-maps to find POS

- shortcut: grouping the maxterms (0s) of the given function
- long way:
 - convert K-map of F to K-map of F' (by flipping 0/1s)
 - get SOP of F' ⇒ POS=(SOP)'

Don't-Care Conditions

- denoted d , e.g.: $F(A, B, C) = \sum m(3, 5, 6) + \sum d(0, 7)$

16. Combinational Circuits & MSI Components

[github/joyntls](https://github.com/joyntls)

Circuit Delays

- given a logic gate with delay t , if the inputs are stable at times t_1, t_2, \dots, t_n , then the earliest time in which the output will be stable is $\max(t_1, t_2, \dots, t_n) + t$
- delay of a combinational circuit: repeat \wedge for all gates



- common circuit delays

- n -bit parallel adder will have delay
 - $S_n = ((n - 1) * 2 + 2)t$
 - $C_{n+1}((n - 1) * 2 + 3)t$
 - max delay = $((n - 1) * 2 + 3)t$

Enable

- one-enable** \rightarrow device is only activated when $E = 1$
- zero-enable** \rightarrow device is only activated when $E = 0$

denoted E' or \bar{E}

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

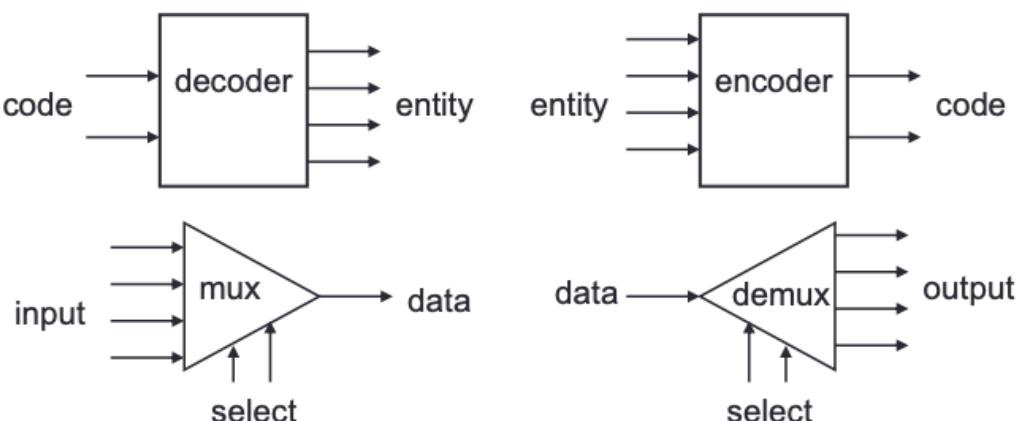
E'	X	Y	F_0	F_1	F_2	F_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	d	d	0	0	0	0

Decoder with 0-enable

Zero-Enable/Negated Outputs

- active-high outputs** \rightarrow normal outputs (selected line is 1)
- active-low outputs** \rightarrow negated outputs (selected line is 0)

MSI Circuits

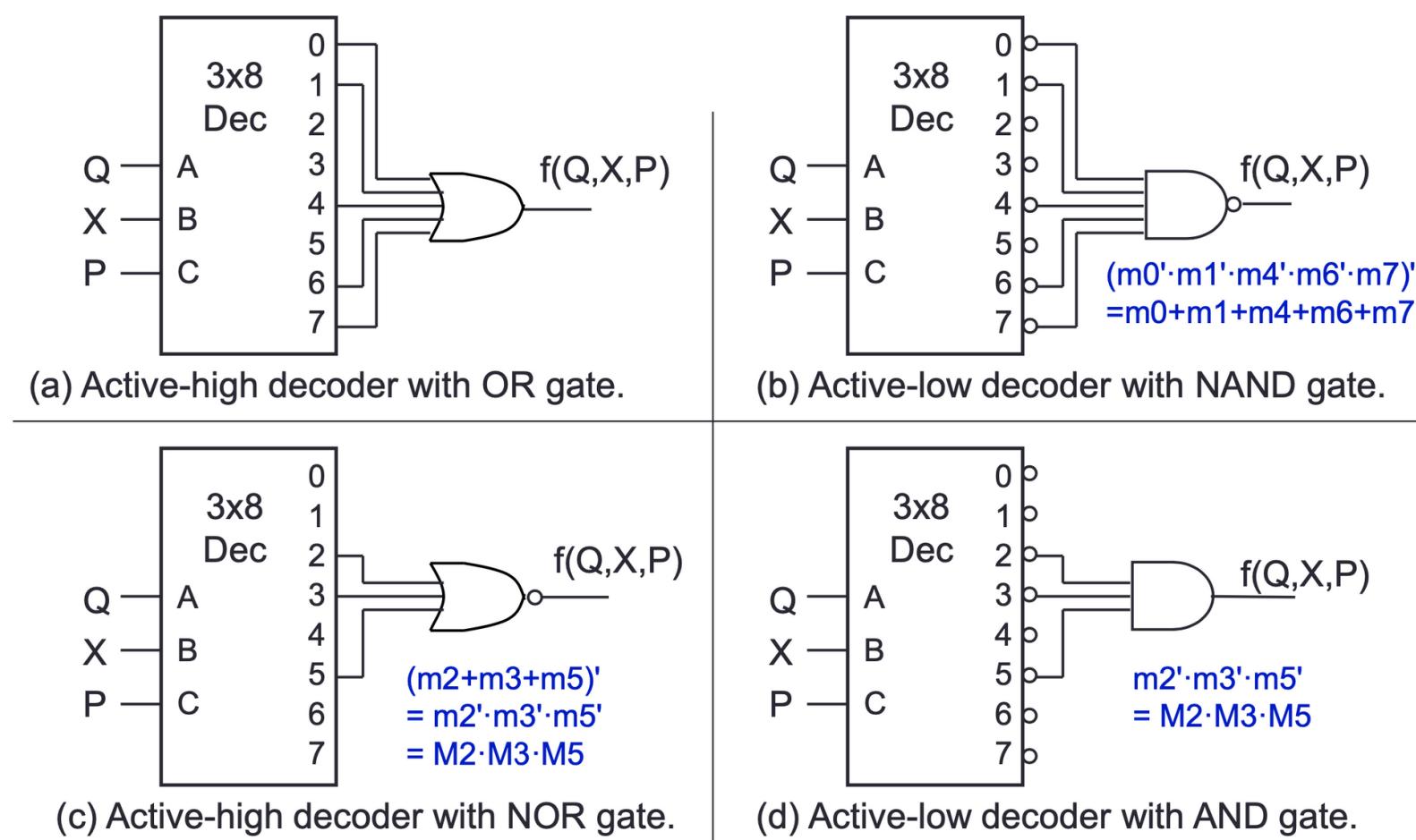


Implementing Functions with Decoders

- any combinational circuit with n inputs and m outputs can be implemented with an $n : 2^n$ decoder with m OR gates
 - input: a boolean function, in sum-of-minterms form
 - output: a decoder to generate the minterms, and an OR gate to form the sum

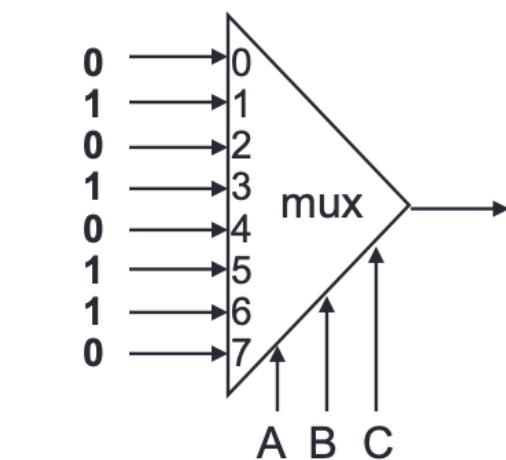
(2/2)

$$f(Q, X, P) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$$



Implementing Functions with Multiplexers

- a 2^n -to-1 multiplexer can implement a Boolean function of n input variables
 - input lines correspond to minterms

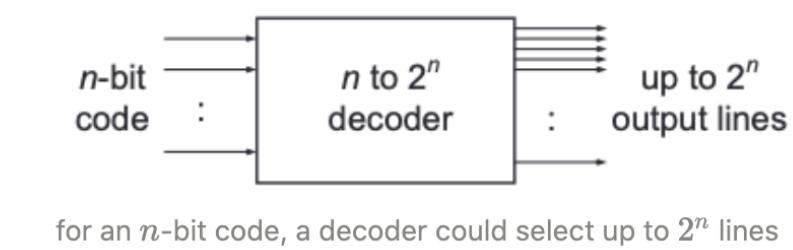
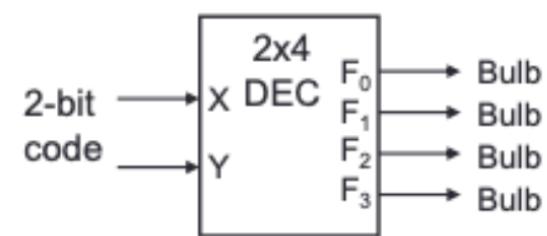


$$F(A, B, C) = \sum m(1, 3, 5, 6)$$

17. MSI Components

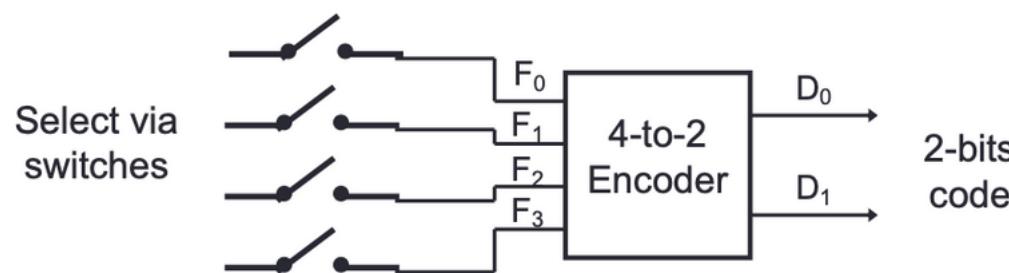
Decoders

- convert binary information from n input lines, to up to 2^n input lines
 - selects only one output line
- aka n -to- m -line decoder, $n : m$ decoder, $n \times m$ decoder where $m \leq 2^n$



Encoders

- encoder** → given a set of input lines, of which exactly one is high and the rest are low, provide a code that corresponds to the high input line
 - opposite of decoder
- $\leq 2^n$ input lines and n output lines
- implemented with OR gates

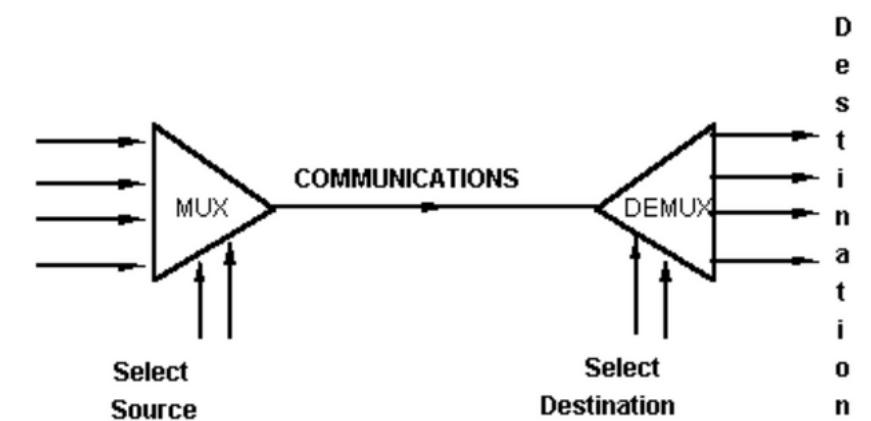


Priority Encoders

- if multiple inputs are equal to 1, the highest priority takes precedence
- all inputs 0: invalid input
- Example of a **4-to-2 priority encoder**:

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

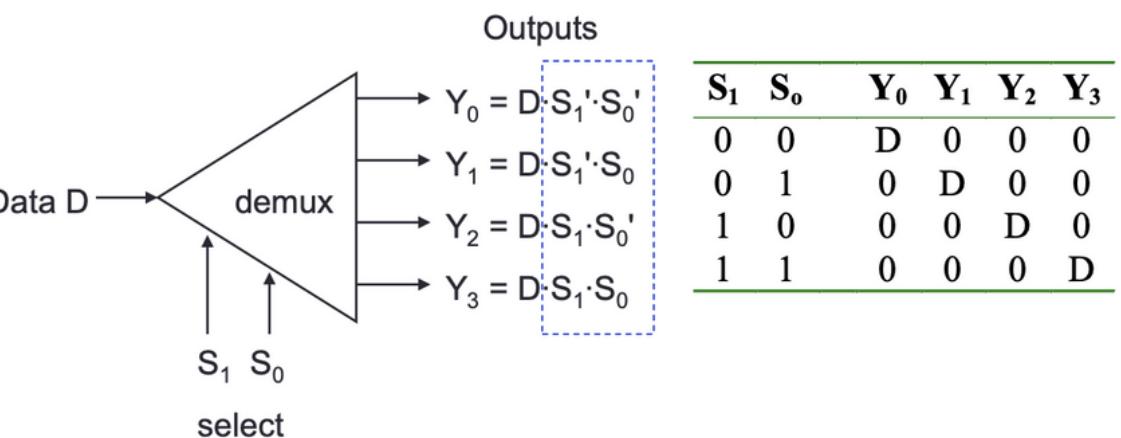
Multiplexers & Demultiplexers



- helps share a single communication line among a number of devices
- only one source and one destination can use the communication line

Demultiplexer

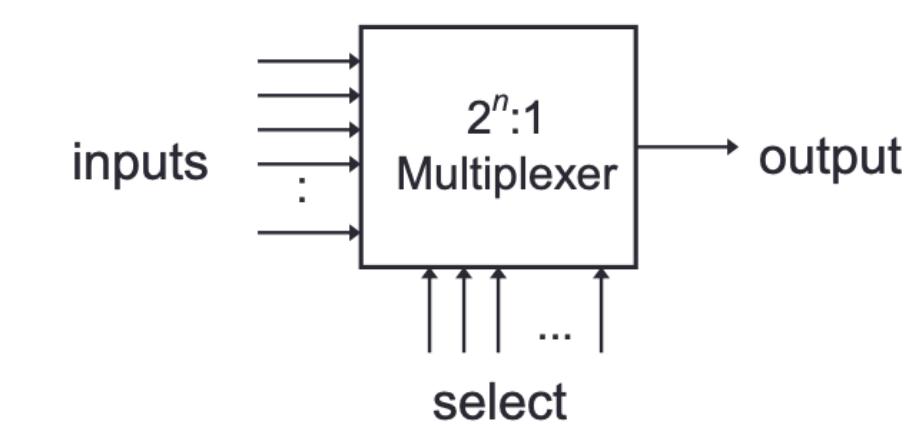
- directs data from the input line to one selected output line
 - input: an input line and set of selection lines
 - "output": directs data to one selected line



 identical to a decoder with enable!

Multiplexer

- steers one of 2^n input lines to a single output line, using n selection lines
 - input: multiple input lines and multiple selection lines
 - output: one output line
- $\blacksquare = \text{sum of the (product of data lines and selection lines)}$



18. Sequential Logic

- sequential circuit → output depends on both present inputs and state
- 2 types of sequential circuits
 - synchronous** → outputs change only at a specific time
 - asynchronous** → outputs change at any time
- classes of sequential circuits
 - bistable** → 2 stable states (e.g. latches / flip-flops)
 - monostable** → 1 stable state; **astable** → no stable state

Latches

- pulse-triggered (vs flipflops: edge-triggered)

Flip-Flops

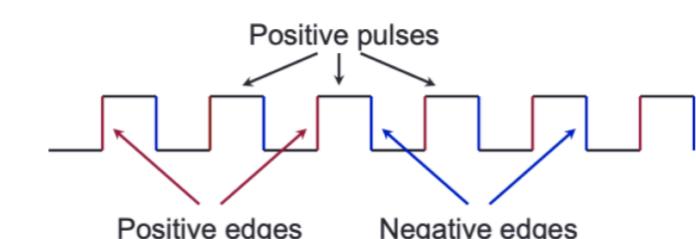
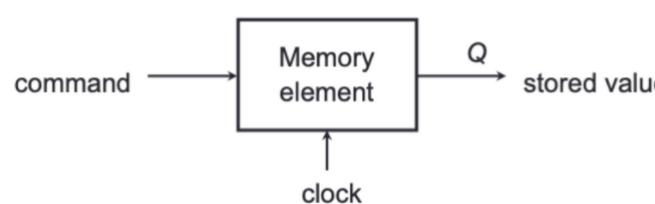
- synchronous bistable devices
 - data on inputs is transferred to the flipflop's output only on the triggered edge of the clock pulse
- output changes state at a specific point on the clock
 - change state at either rising or falling edge of the clock signal

Memory Elements

- memory element** → device which can remember value indefinitely, or change value on command from its inputs

Memory with Clock (types of triggering/activation)

- pulse-triggered: ON = 1, OFF = 0
- edge-triggered
 - positive edge-triggered: ON = from 0 to 1, OFF = other time
 - negative edge-triggered: ON = from 1 to 0, OFF = other time



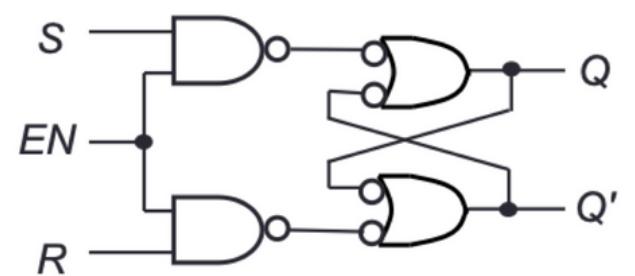
Latches

S-R Latch

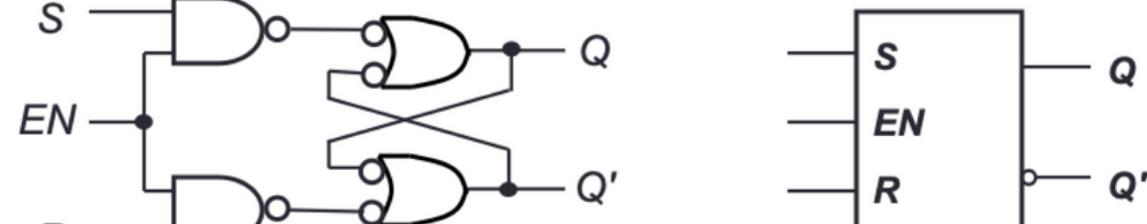
- 2 complementary outputs: Q and Q'
 - $Q = HIGH$ → latch is in **SET** state
 - $Q = LOW$ → latch is in **RESET** state

Gated S-R Latch

- Gated S-R latch** = S-R Latch + enable input (EN) + 2 NAND gates
- outputs change only when EN is high



S	R	Q	Q'	
0	0	NC	NC	No change. Latch remained in present state.
1	0	1	0	Latch SET.
0	1	0	1	Latch RESET.
1	1	0	0	Invalid condition.

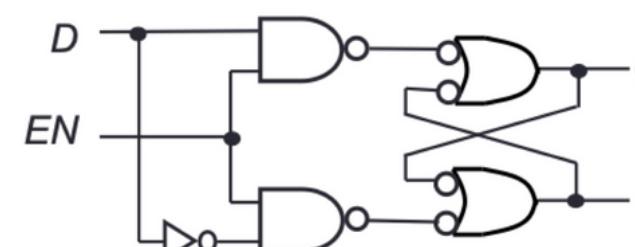


S	R	Q(t+1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	Indeterminate	

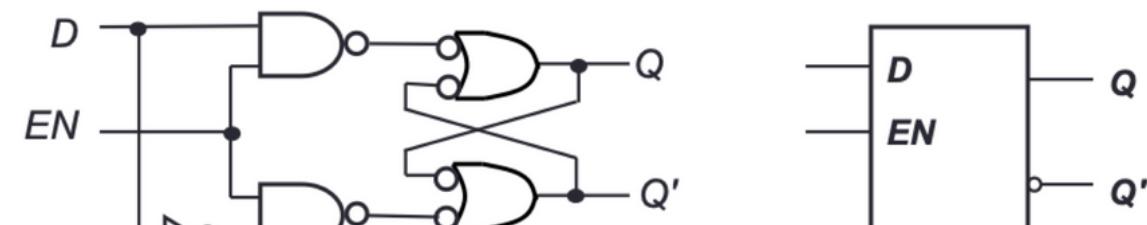
for active-high input S-R latch

Gated D Latch

- Gated D latch** → make input R equal to S'
- eliminates undesirable condition of invalid state in the S-R latch

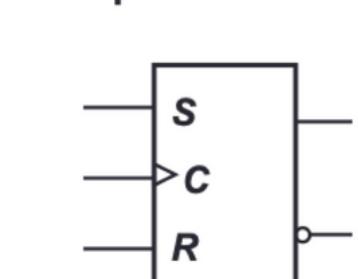


EN	D	Q(t+1)	
1	0	0	Reset
1	1	1	Set
0	X	$Q(t)$	No change



Flip-Flops

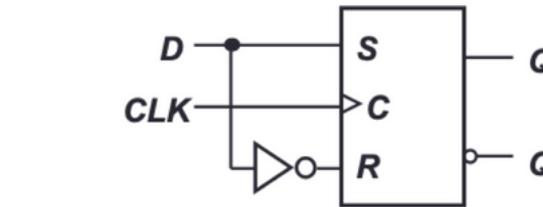
S-R flip-flop



S	R	CLK	Q(t+1)	Comments
0	0	X	$Q(t)$	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

X = irrelevant ("don't care")
↑ = clock transition LOW to HIGH

D flip-flop

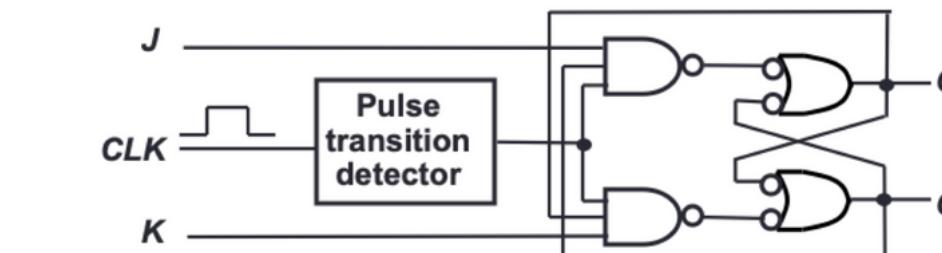


D	CLK	Q(t+1)	Comments
1	↑	1	Set
0	↑	0	Reset

A positive edge-triggered D flip-flop formed with an S-R flip-flop.

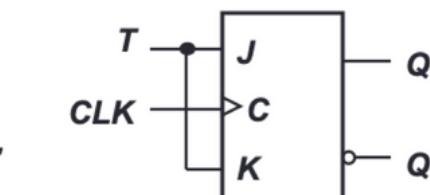
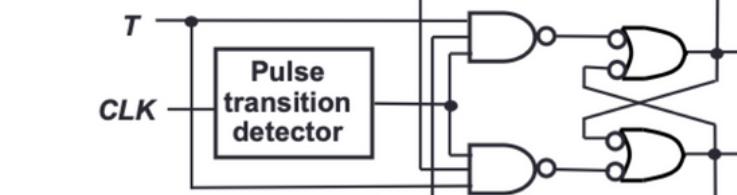
J-K flip-flop

- no invalid state!
- includes a **toggle state** → Q changes on each active clock edge
 - $Q(t+1) = J \cdot Q' + K' \cdot Q$



T flip-flop

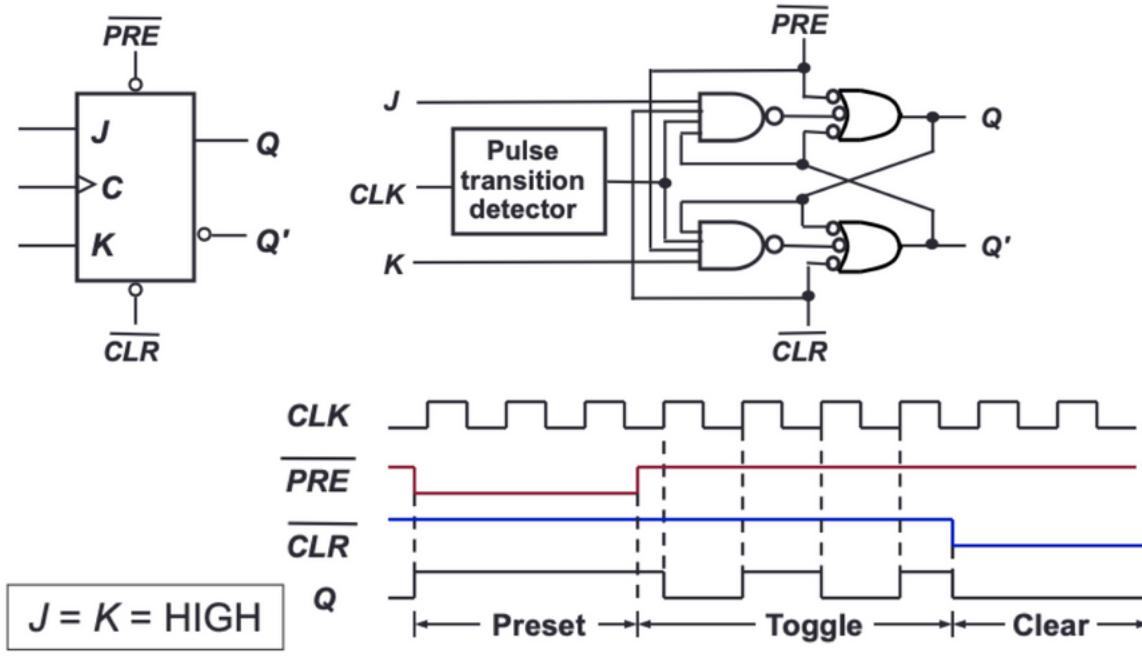
- single input version of J-K flipflop, formed by typing both inputs together
 - $Q(t+1) = T \oplus Q$



19. Sequential Logic (2)

Asynchronous Inputs

- **asynchronous** → inputs affect the state of the flip-flop independent of the clock
- e.g. J-K flip-flop with active-low PRESET and CLEAR asynchronous inputs



- active-high:
 - PRE = 1 → Q is immediately set to HIGH
 - CLR = 1 → Q is immediately cleared to LOW
 - normal operation mode: both PRE and CLR are LOW

Synchronous Sequential Circuits

Flip-Flop Characteristic Tables

J	K	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q(t)'$	Toggle

S	R	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

D	$Q(t+1)$
0	0
1	1

T	$Q(t+1)$
0	$Q(t)$
1	$Q(t)'$

Excitation Tables

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

JK Flip-flop

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

D Flip-flop

T Flip-flop

20. Sequential Circuits Design

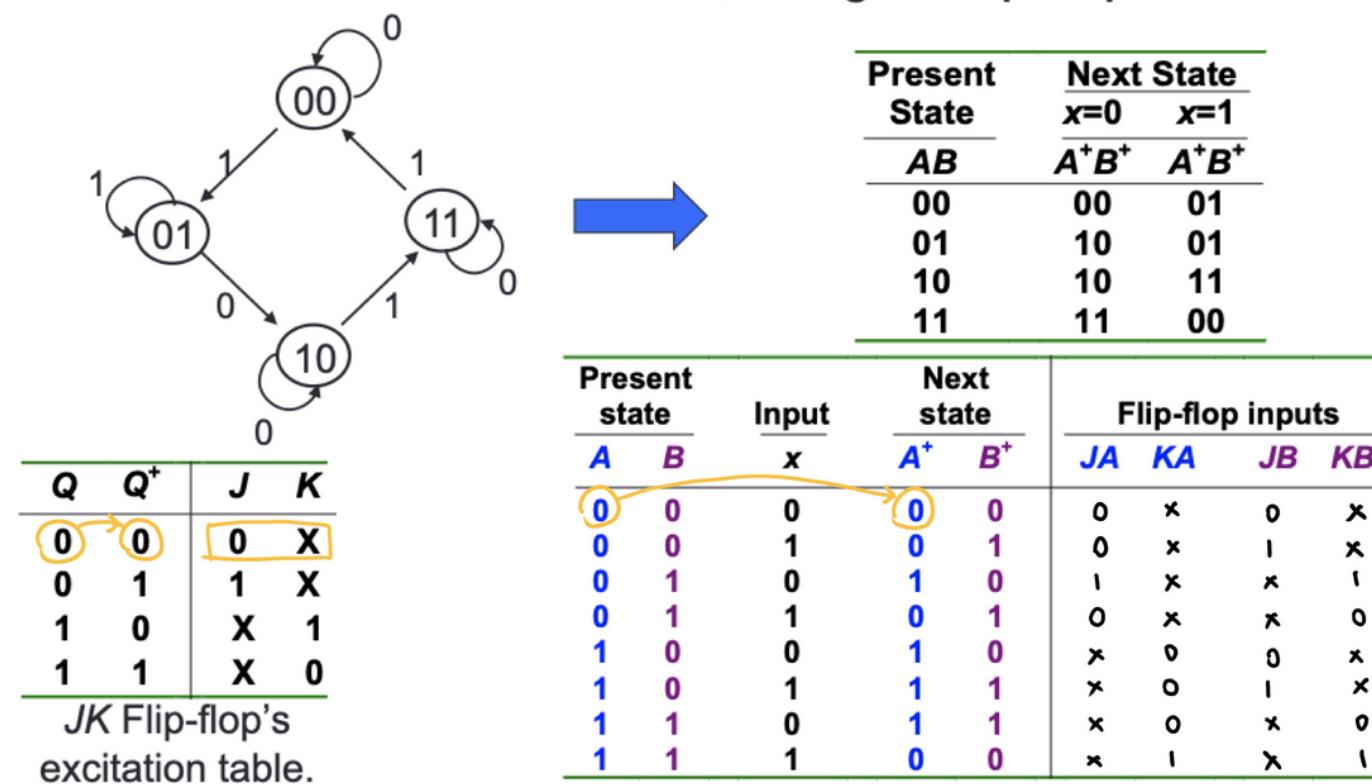
Sequential Circuits Design

(from state equations/table/diagram to logic circuit)

1. convert state diagram \rightarrow state table
2. fill in flip-flop inputs based on excitation table
3. use K-maps to get simplified logic expressions for flip-flop inputs (e.g. JA , KA , JB , KB)
4. draw circuit implementing

 unused states \rightarrow use don't-care for input

- Circuit state/excitation table, using JK flip-flops.

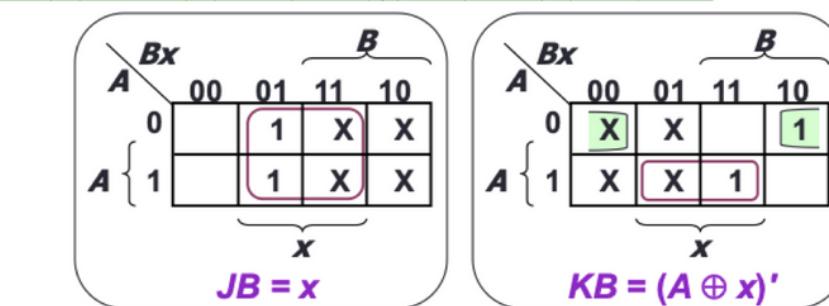


 self-correcting \rightarrow any unused state can transition to a used state after a finite number of cycles

(cont.)

- From state table, get flip-flop input functions.

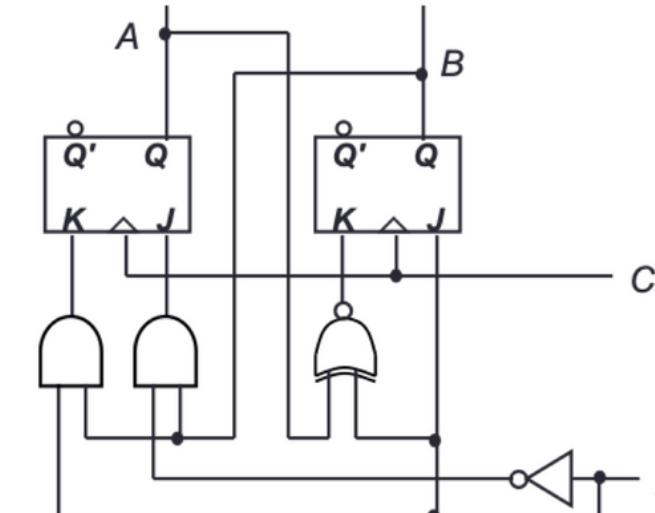
Present state	Input	Next state		Flip-flop inputs					
		A	B	A^+	B^+	JA	KA	JB	KB
00	0	0	0	0	0	0	x	0	x
01	1	0	1	0	1	0	x	1	x
10	0	1	0	1	0	1	x	0	1
11	1	1	0	0	1	0	x	0	0



- Flip-flop input functions:

$$\begin{aligned} JA &= B \cdot x' \\ KA &= B \cdot x \end{aligned}$$

- Logic diagram:



Unused States

- use don't-care for input/next state/flip-flop inputs/output

Present state	Input	Next state		Flip-flop inputs				Output				
		A	B	A^+	B^+	C	SA	RA	SB	RB	SC	RC
00	0	0	0	0	1	0	0	x	0	x	0	0
01	1	0	1	0	1	0	0	x	1	0	0	0
10	0	1	0	1	0	1	0	x	0	1	0	0
11	1	1	0	0	1	1	1	0	0	1	0	0

Given these Derive these Are there other unused states?

Unused state 000: ,110,111

0	0	0	0	x	x	x	x	x	x	x	x
0	0	0	1	x	x	x	x	x	x	x	x

Sequential Circuits Analysis

(from logic circuit to state equations/table/diagram)

1. obtain flip-flop input functions from the circuit
2. fill in the state table (using characteristic table)
3. draw state diagram

$$\begin{aligned} JA &= B \\ KA &= B \cdot x' \\ JB &= x' \\ KB &= A' \cdot x + A \cdot x' = A \oplus x \end{aligned}$$

Fill the state table using the above functions, knowing the characteristics of the flip-flops used.

J	K	$Q(t+1)$	Comments		Present state	Input	Next state		Flip-flop inputs			
			A	B			A^+	B^+	JA	KA	JB	KB
0	0	Q(t)	No change		00	0	0	1	0	0	1	0
0	1	0	Reset		01	1	0	0	0	0	0	1
1	0	1	Set		10	0	1	1	1	1	0	0
1	1	Q(t)'	Toggle		11	1	1	0	0	1	0	0

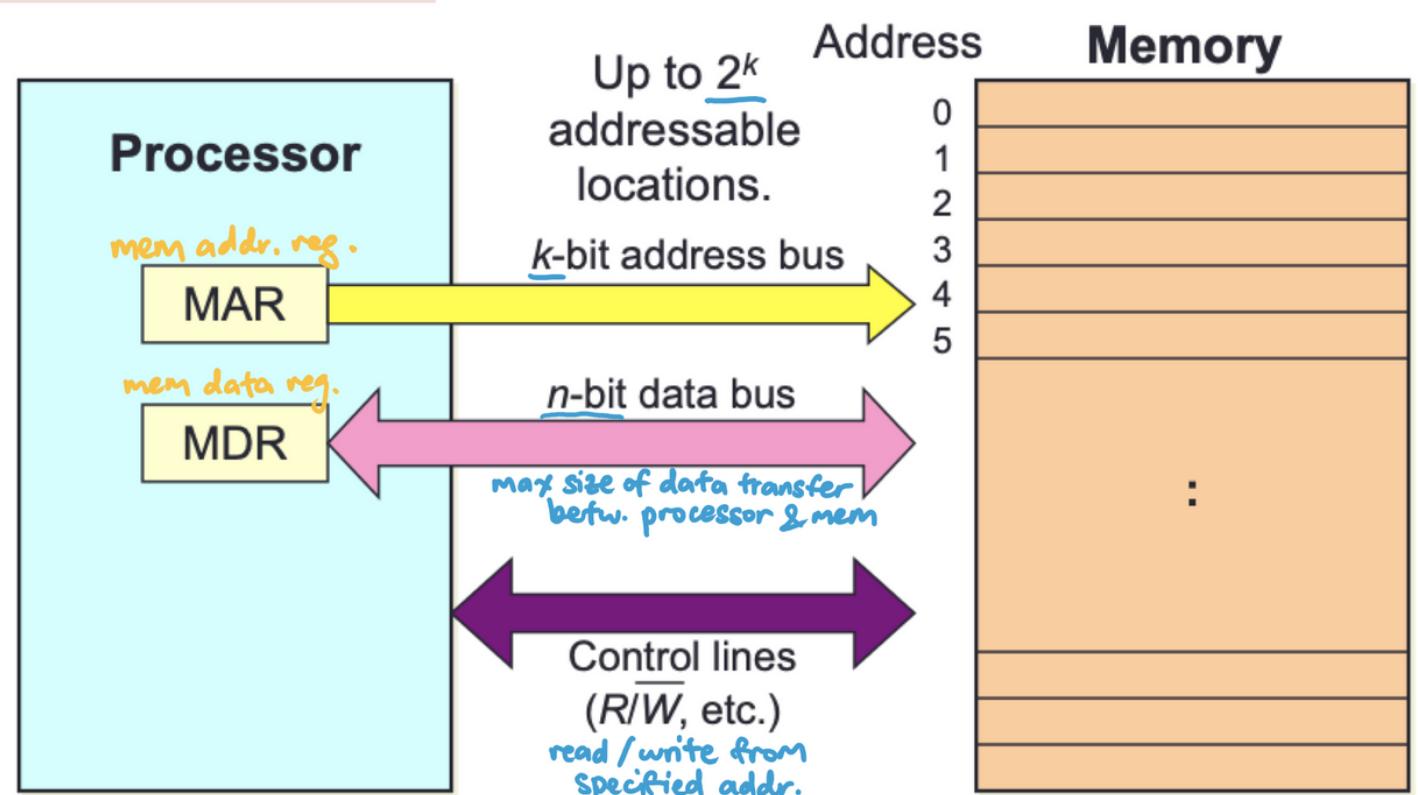
21. Memory

Memory

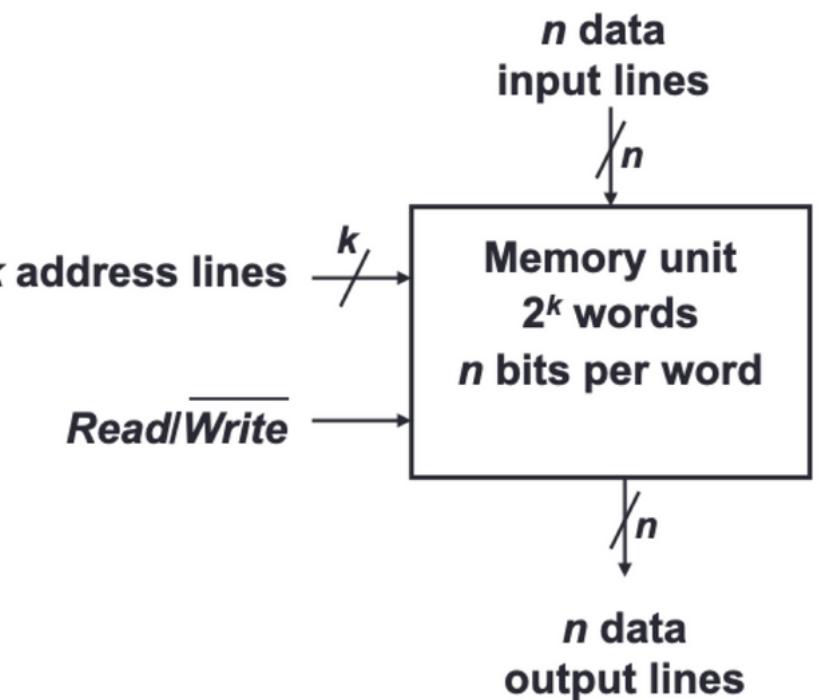
- stores programs and data
- 1 byte = 8 bits; 1 word = multiple of bytes, usually size of register
- memory unit stores binary information in words (groups of bits)
- data consists of n lines for n -bit words
 - **data input lines** → provides information to be written into memory
 - **data output lines** → carries information to be read from memory
- **address** consists of k lines
 - specifies which word (of the 2^k words available) to be selected for reading/writing
- **control lines** R/W → specifies direction of transfer of data

Data Transfer

Data transfer

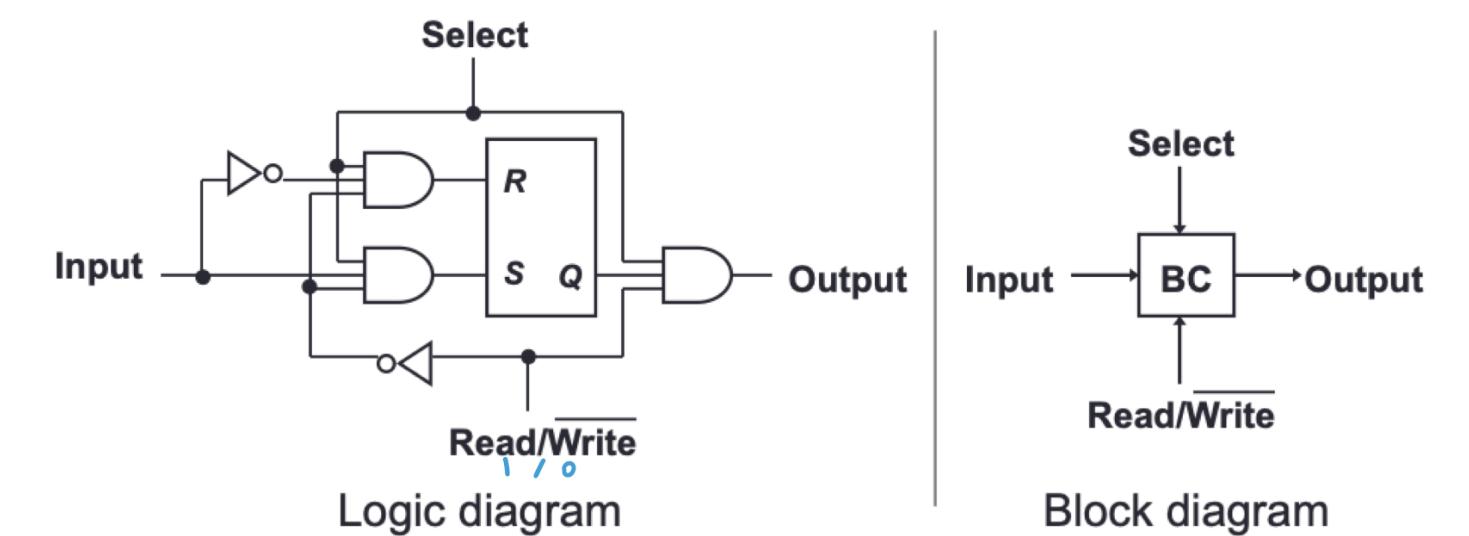


Memory Unit



Memory Cell

- 2 types of RAM
 - static RAMs → use flip-flops as the memory cells
 - dynamic RAMs → use capacitor charges to represent data



Read/Write Operations

- write
 - transfers address of the desired word to the address lines
 - transfers the word (data bits) to be stored in memory to the data input lines
 - activates the Write control (set R/W to 0)
- read
 - transfers address of the desired word to the address lines
 - activates the Read control (set R/W to 1)

Memory Enable	Read/Write	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

Memory Arrays

- array of RAM chips → memory chips combined to form larger memory

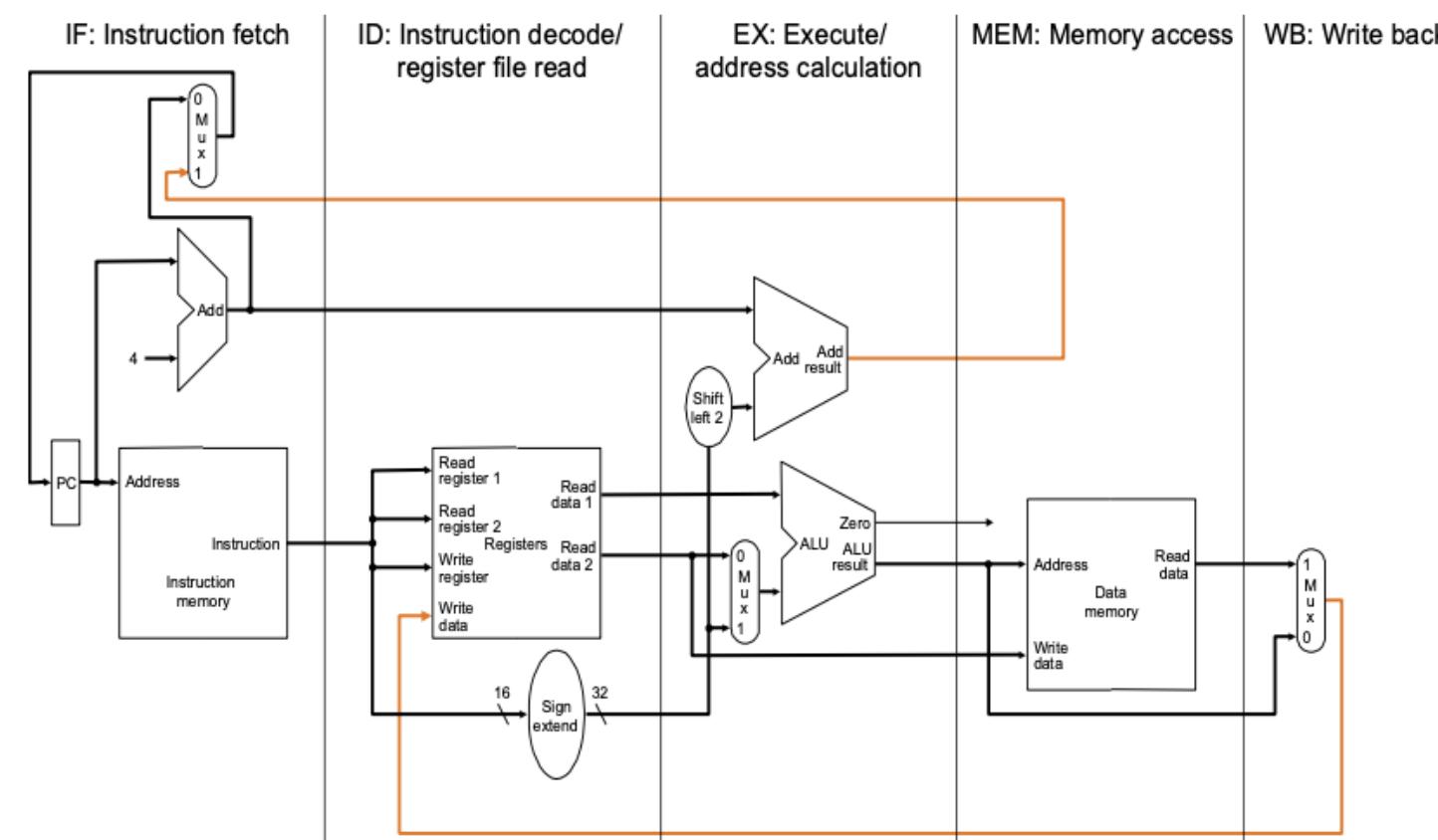
22. MIPS Pipelining

- improves the throughput of the entire workload
 - pipelined implementation
 - one cycle per pipeline stage
 - data required for each stage needs to be stored separately

MIPS Pipeline

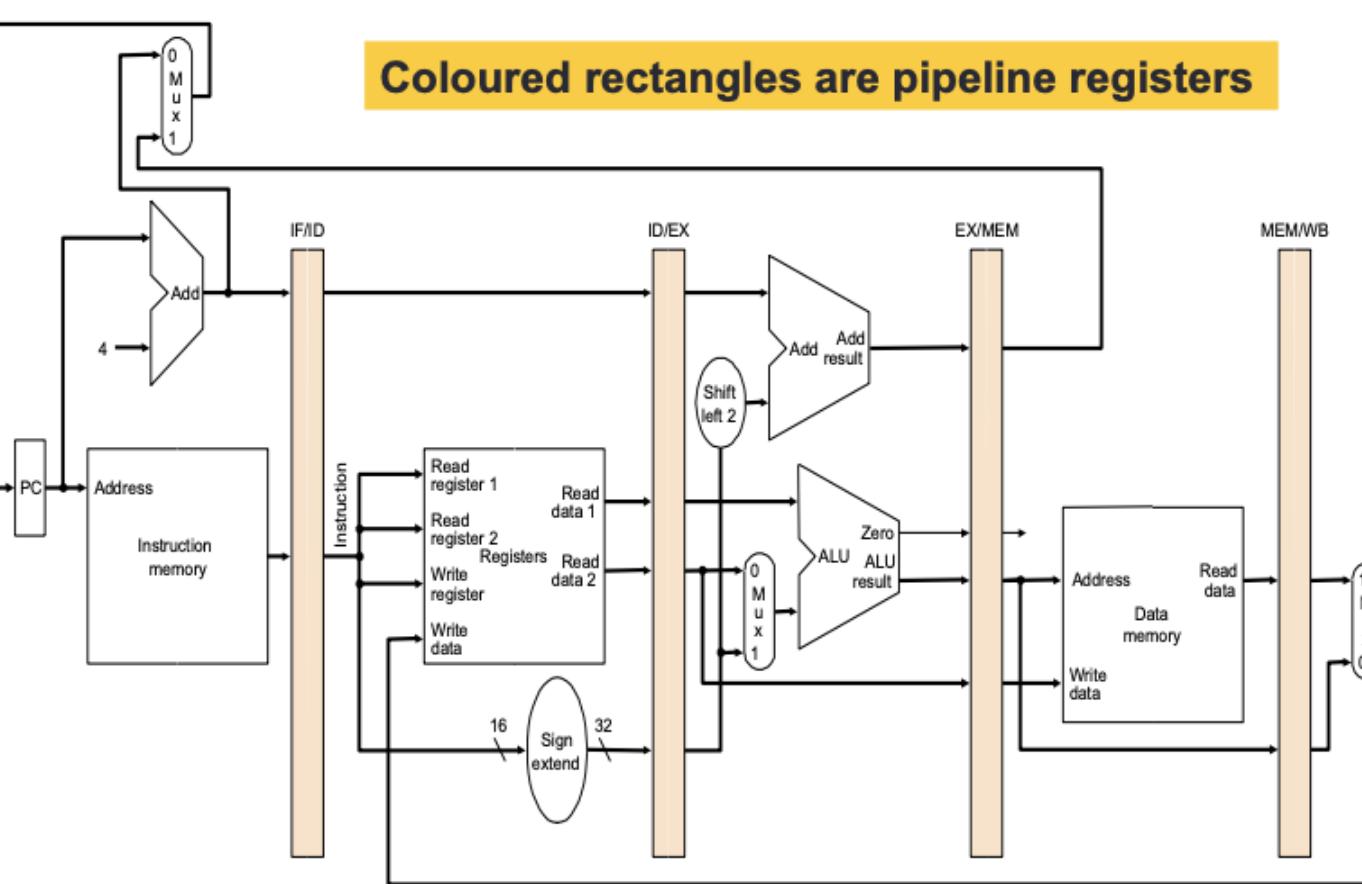
- MIPS pipeline stages:
 1. **IF** - instruction fetch
 2. **ID** - instruction decode and register read
 3. **EX** - execute an operation or calculate an address
 4. **MEM** - access an operand in data mem
 5. **WB** - write result back to a register

- each stage is 1 clock cycle
 - exceptions: updating PC and writing back to register file



Pipeline Registers

- not necessarily 32-bits even though MIPS registers are 32 bits
 - a collection of registers
 - pipeline registers :
 - **IF/ID** : register between IF and ID
 - **ID/EX** : register between ID and EX
 - **EX/MEM** : register between EX and MEM
 - **MEM/WB** : register between MEM and WB
 - no register needed for the end of the **WB** stage



Pipeline Control

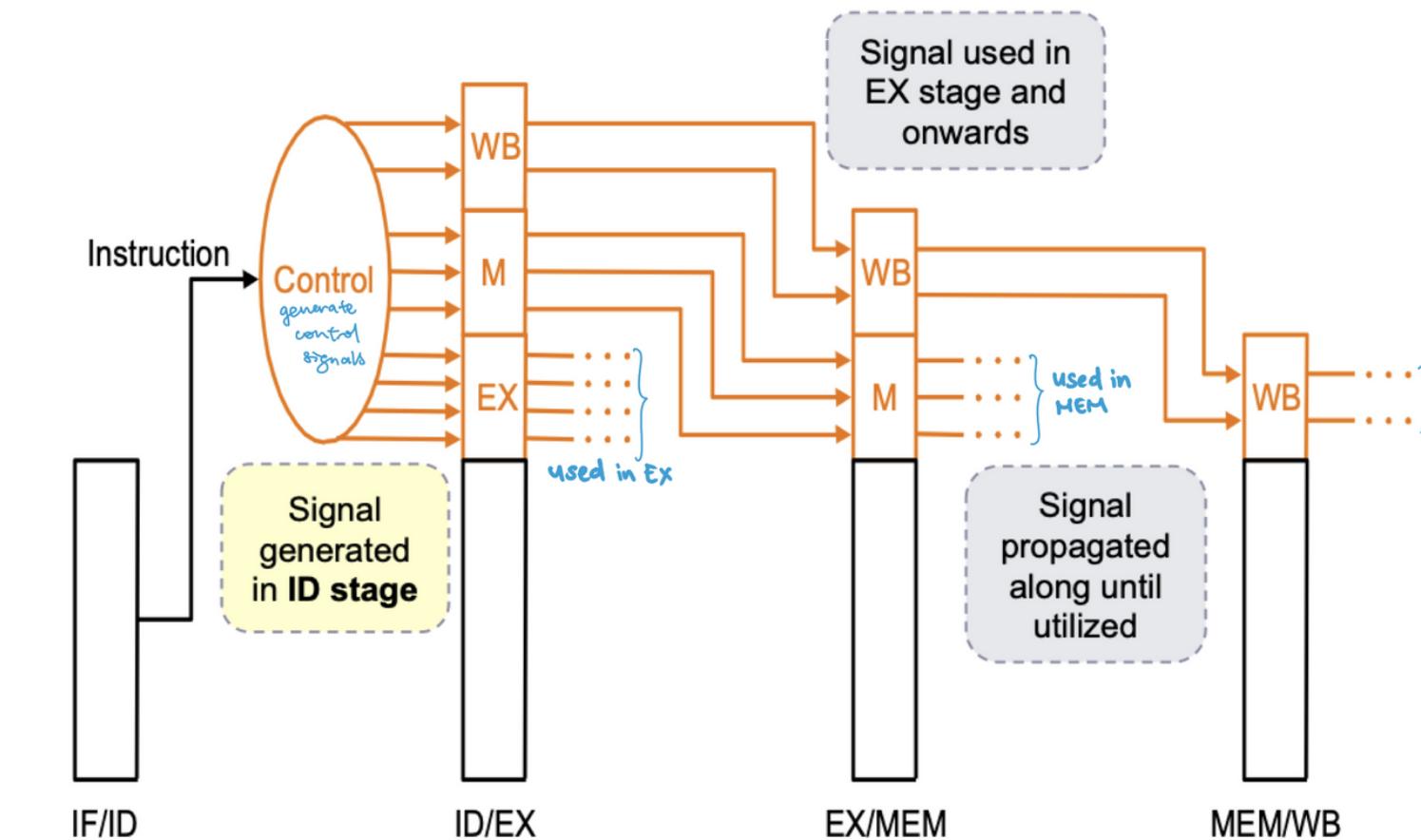
- same control signals as single-cycle datapath
 - each control signal belongs to a particular pipeline stage
 - avoid mixing up controls from different pipelines

Grouping

- group control signals according to pipeline stage

	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

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUOp		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0



23. Pipeline Datapath

IF stage

- at the end of a cycle, receives and stores into IF/ID pipeline register:
 - receives PC + 4
 - receives instruction read from instructionMemory[PC]
-

ID stage

IF/ID register supplies:

- 2 register numbers to be read
- 16-bit offset (to be signExt to 32-bit)

ID/EX register receives:

- data values read from reg file
 - 32-bit immd value
 - PC + 4
-

EX stage

EX/EX register supplies:

- data values read from reg file
- 32-bit immd value
- PC + 4
- write register number

EX/MEM register receives:

- (PC + 4) + (immd * 4)
 - ALU result
 - `isZero` signal
 - RD2 from register file
 - write register number
-

MEM stage

EX/MEM register supplies:

- (PC + 4) + (immd * 4)
- ALU result
- `isZero` signal
- RD2 from register file
- write register number

MEM/WB register receives:

- ALU result
 - mem read data
 - write register number
-

WB stage

MEM/WB register supplies

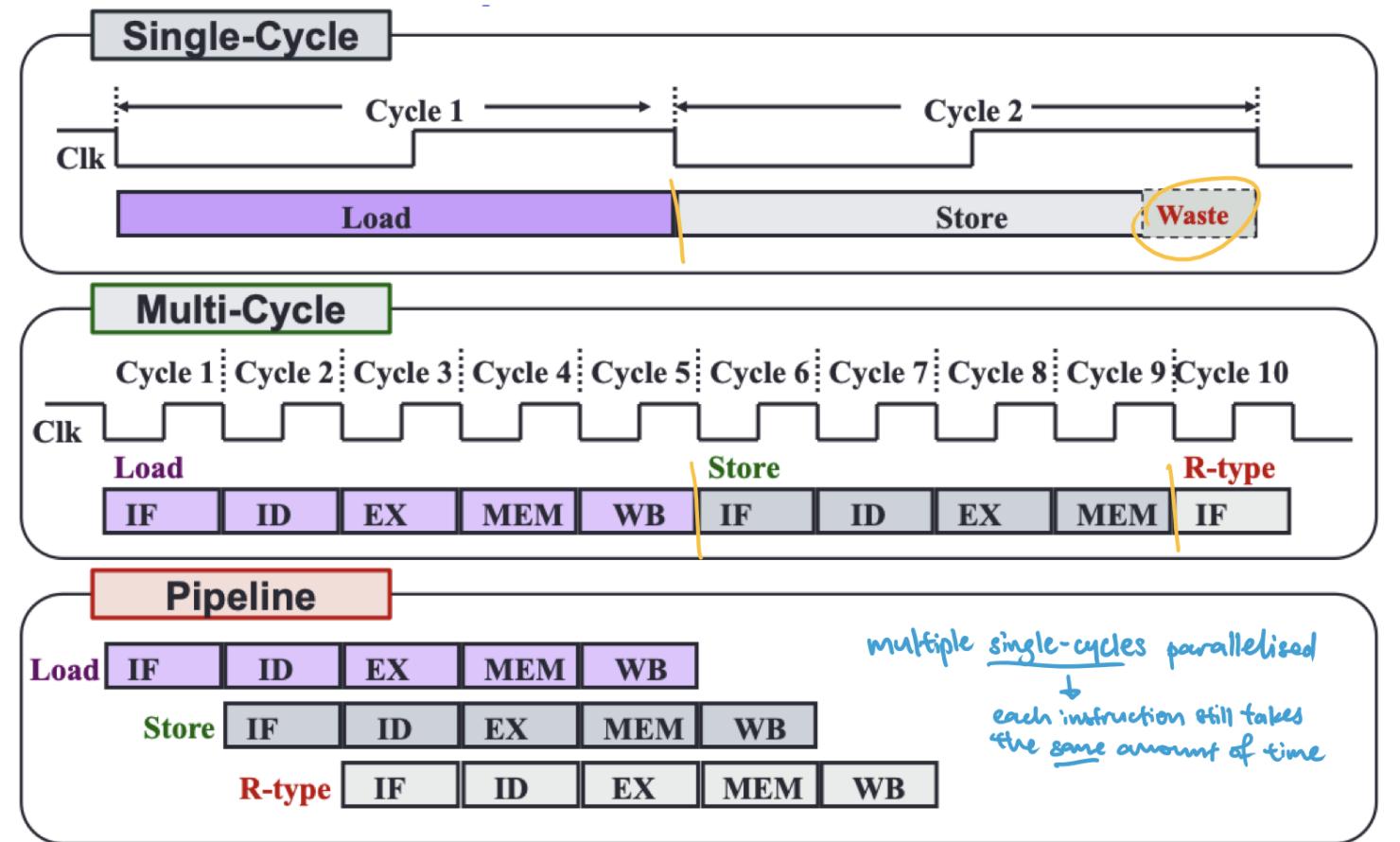
- ALU result
- mem read data
- write register number

end of cycle:

result is written back to the reg file (if applicable) using WR number

24. Pipeline Implementations

 [github/joyntls](https://github.com/joyntls)



Performance Comparison

let

- CT = cycle time,
- T_k = time for operation in stage k ,
- N = number of stages

single-cycle

- cycle time, $CT_{seq} = \max(\sum_{K=1}^N T_k)$
- execution time for I instructions = $I \times CT_{seq}$

multi-cycle

- cycle time, $CT_{multi} = \max(T_k)$
- execution time for I instructions = $I \times \text{Average CPI} \times CT_{multi}$
 - average CPI used since each instruction takes diff number of cycles

pipelining

- cycle time, $CT_{pipeline} = \max(T_k) + T_d$
 - where T_d = overhead for pipelining (e.g. pipeline register)
- cycles needed for I instructions = $(I + N - 1)$
 - $N - 1$ cycles to fill up pipeline
- execution time for I instructions = $(I + N - 1) \times CT_{pipeline}$

Ideal Speedup

- assumptions:

- every stage takes the same amt of time $\Rightarrow \sum_{k=1}^N T_k = N \times T_1$
- no pipeline overhead $\Rightarrow T_d = 0$
- $I \gg N$

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{Time}_{\text{seq}}}{\text{Time}_{\text{pipeline}}} = N$$

25. Pipelining Hazards

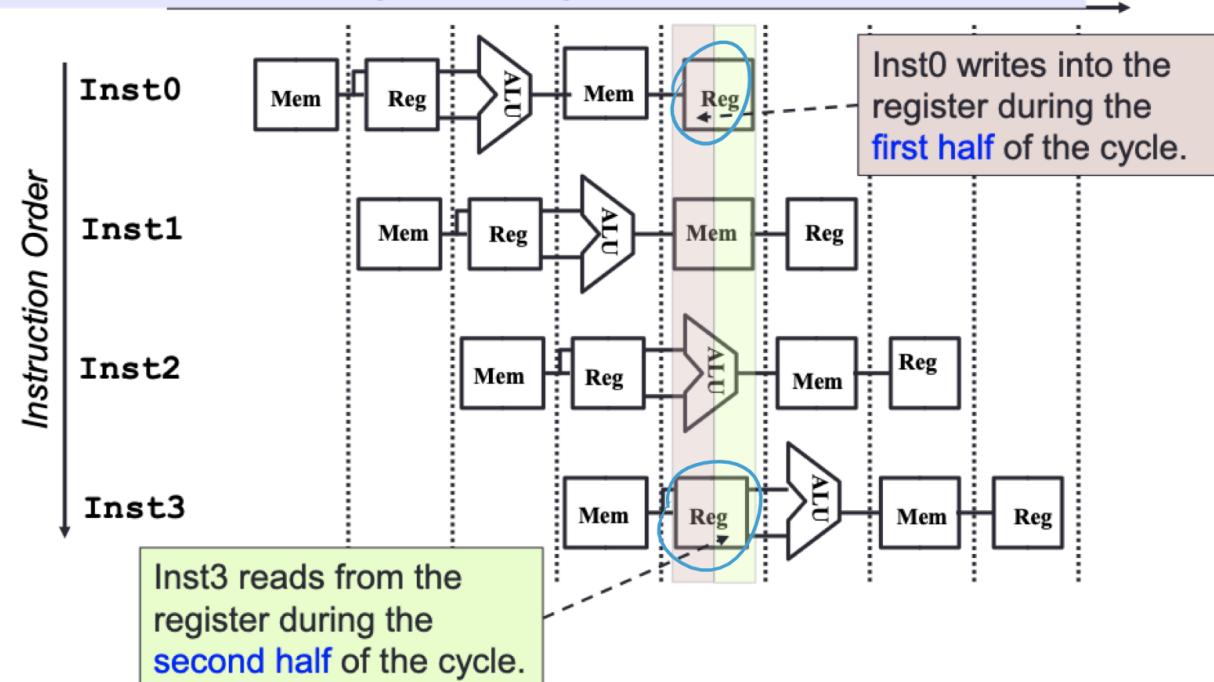
github/joyntls

Structural Hazards

- solves read/write conflict in MEM module
- split cycle into half
 - first half: write into register
 - second half: read from register

Recall that registers are very fast memory.

Solution: **Split cycle into half**; first half for writing into a register; second half for reading from a register.

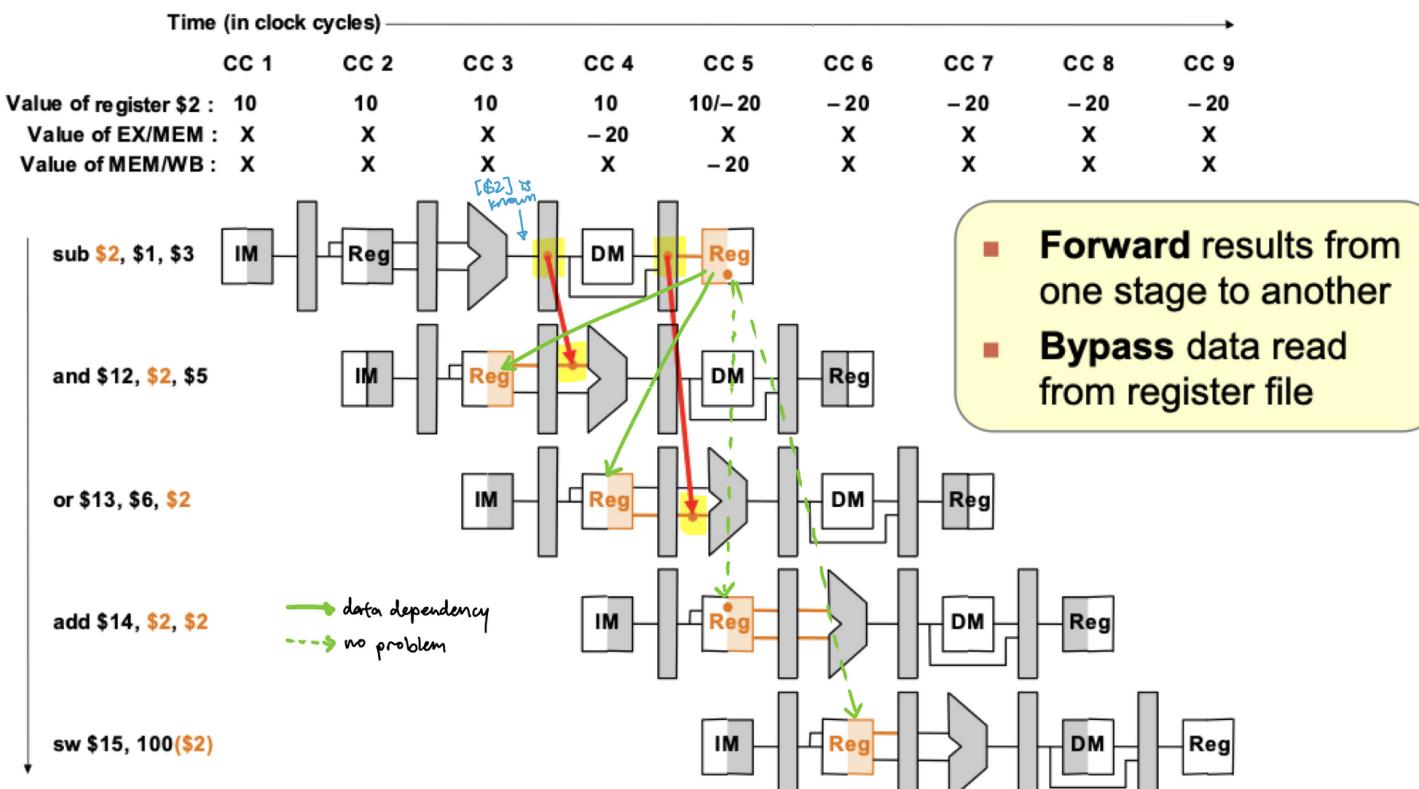


Data Hazards

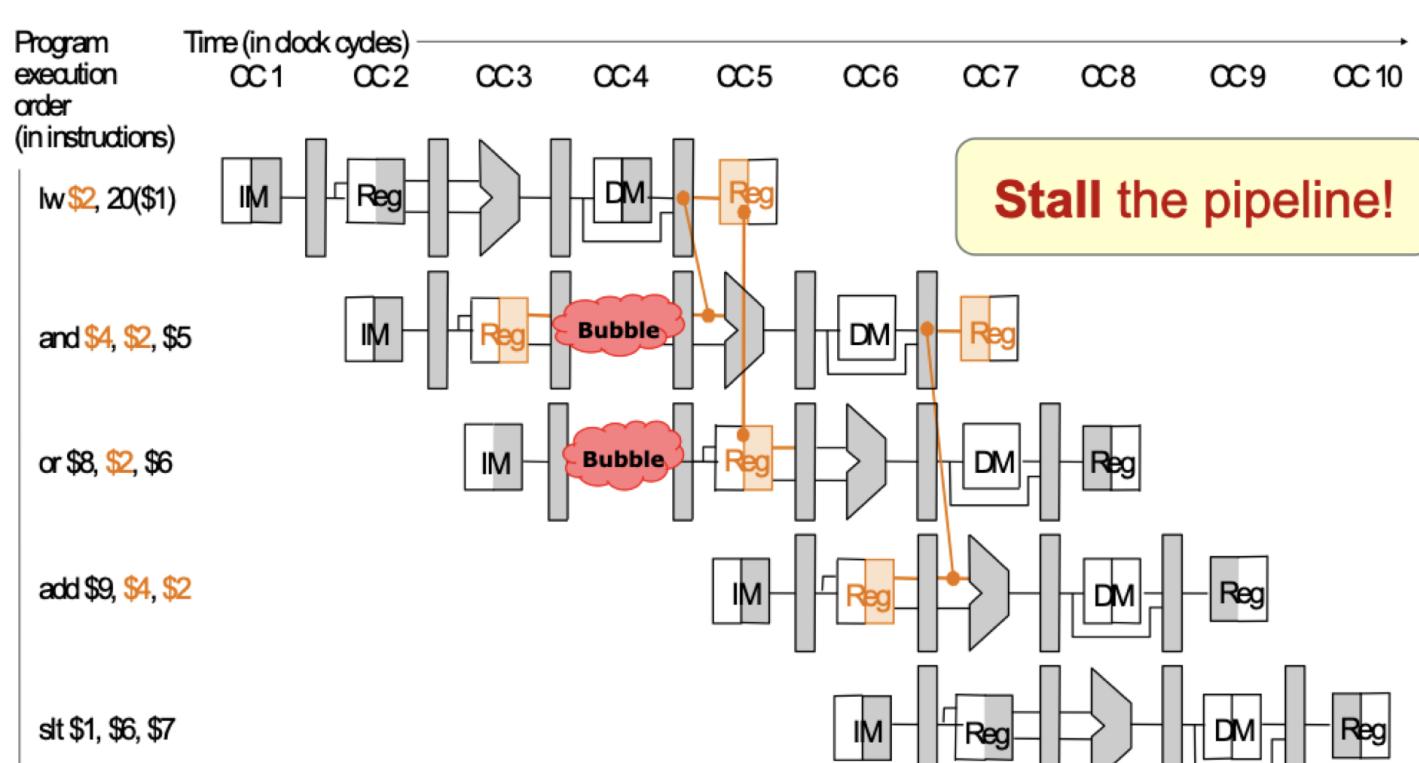
- instruction dependency: read-after-write

Forwarding

- forward data right after EX stage is completed
 - usually no delay incurred



- for **lw**, data is only ready after MEM stage
 - may incur extra 1 cycle delay!



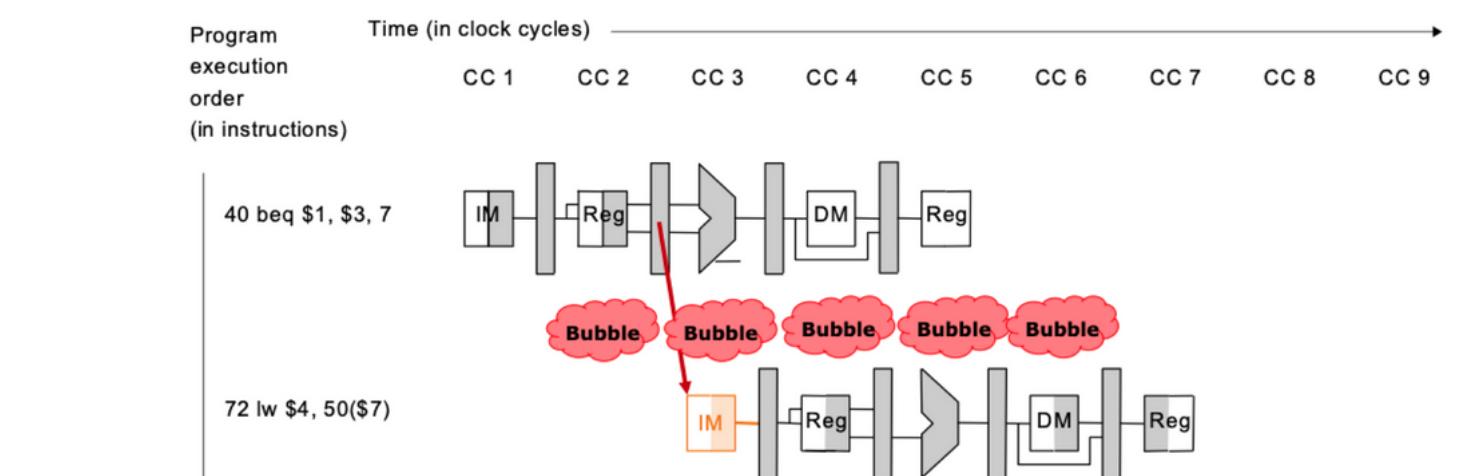
Control Hazards

- branch decision in MEM stage
- stall the pipeline → 3 cycles delay (next IM starts after current MEM)

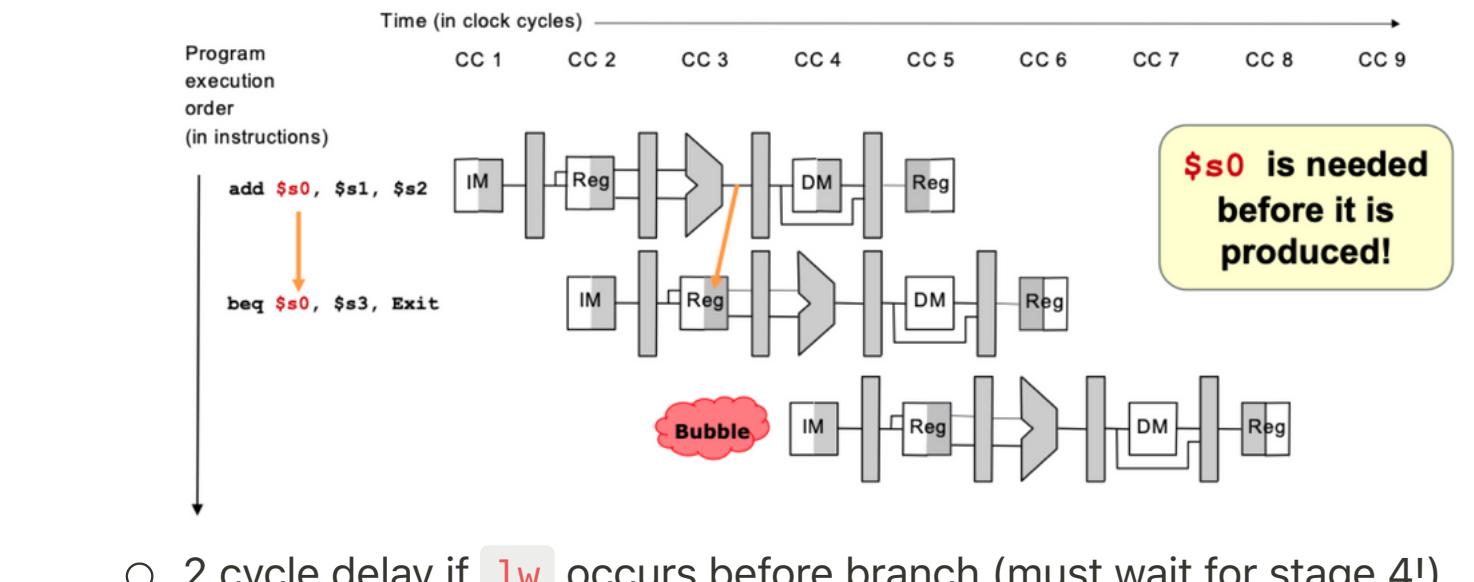
Early Branch Resolution

- branch decision in ID instead of MEM

- 1 cycle delay



- 2 cycle delay if there is data dependency as well



Branch Prediction (assume branch is not taken)

- if branch is taken: stall 1 or 3 cycles depending on early branching

Delayed Branch

- if the branch outcome takes X number of cycles to be known, move non-control dependent instructions into the X slots following a branch
 - aka **branch delay slot** (the 'no-op' slots)
 - instructions that will be executed regardless of the branch outcome

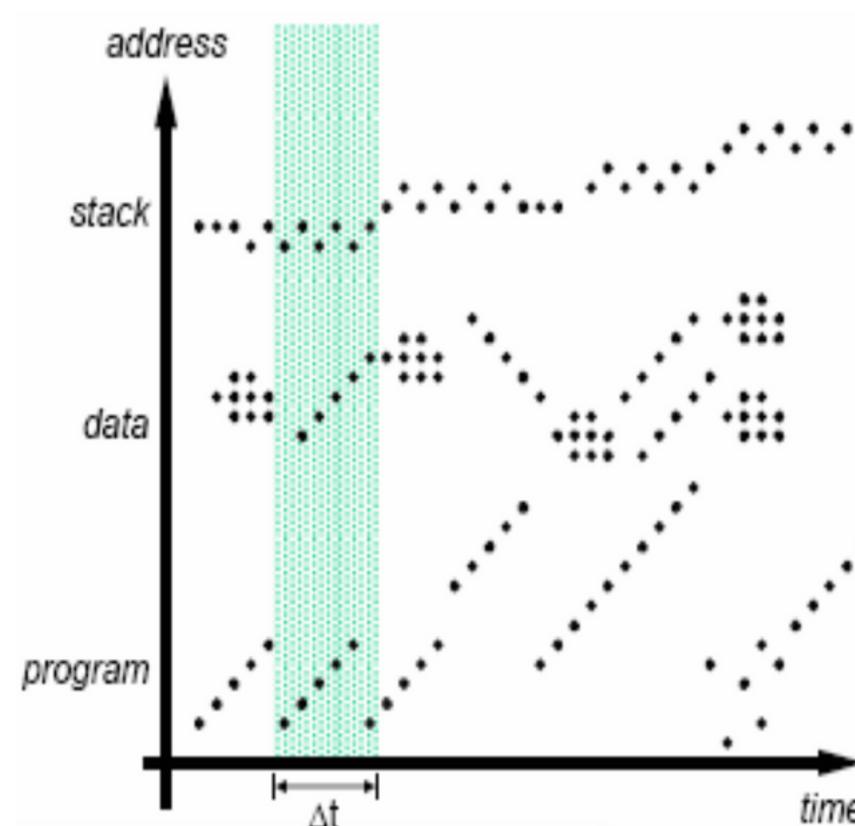
26. Cache

Locality

- principle of locality → program accesses only a small portion of the memory address space within a small time interval
 - temporal locality → if an item is referenced, it will tend to be referenced again (e.g. loops)
 - spatial locality → if an item is referenced, nearby items will tend to be referenced soon (e.g. array)

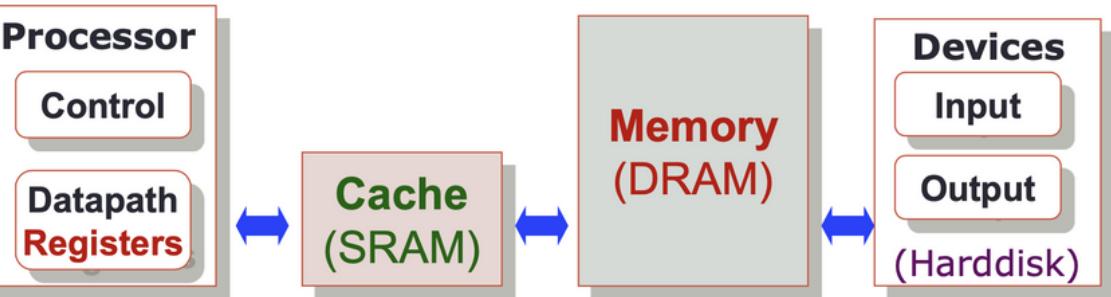
Working Set

- set of locations accessed during Δt
- aim: capture the working set and keep it in the memory closest to the CPU

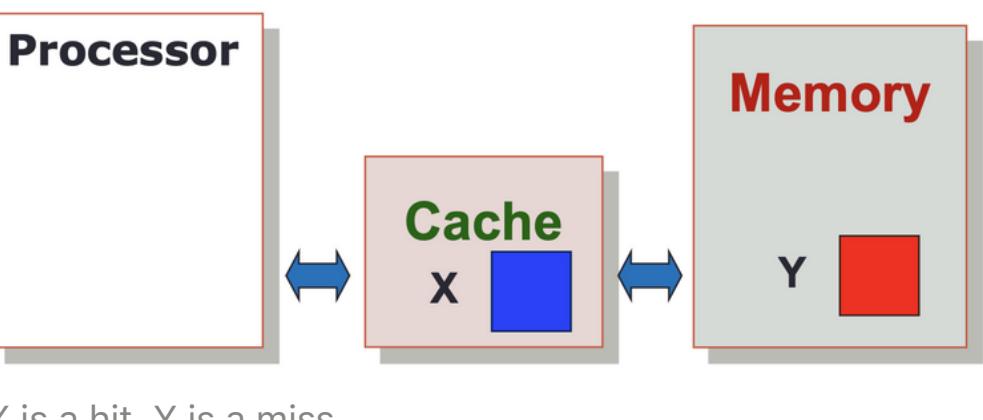


Cache

- cache → small but fast SRAM near CPU



Memory Access Time



X is a hit, Y is a miss

- hit → data is in cache
 - hit rate → fraction of memory accesses that hit
 - hit time → time to access cache
- miss → data is not in the cache
 - miss rate = 1 - hitrate
 - miss penalty = time to replace cache block + hit time
 - hit time < miss penalty
- average access time

$$\text{average access time} = (\text{hit rate} \times \text{hit time}) + (\text{miss rate} \times \text{miss penalty})$$

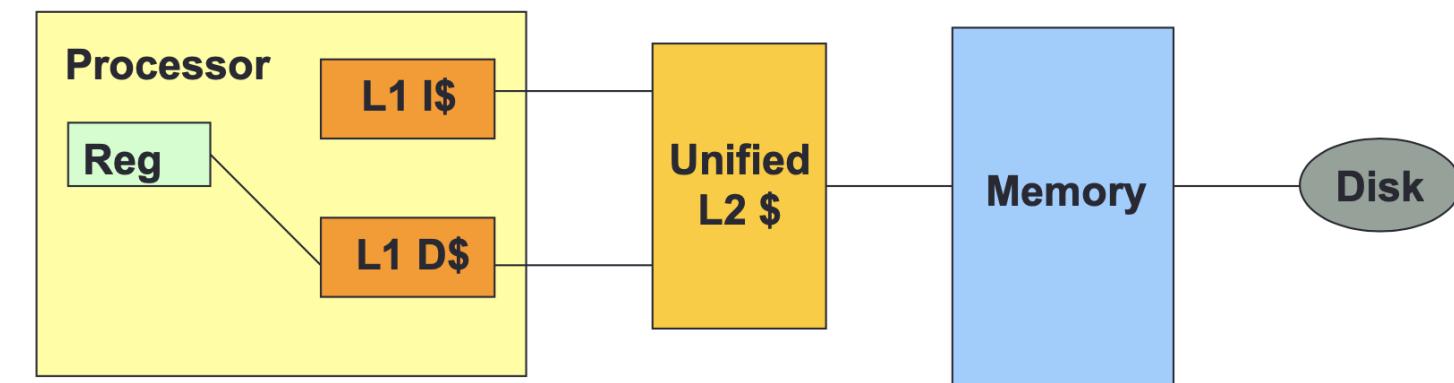
Block Size Trade-off

larger block size:

- ✓ takes advantage of spatial locality
 - more likely to use the surrounding data
- ✗ larger miss penalty - takes longer time to fill up the block
- ✗ fewer cache blocks - miss rate increases (larger block size = fewer cache blocks)

Multilevel Cache

- separate data and instruction cache

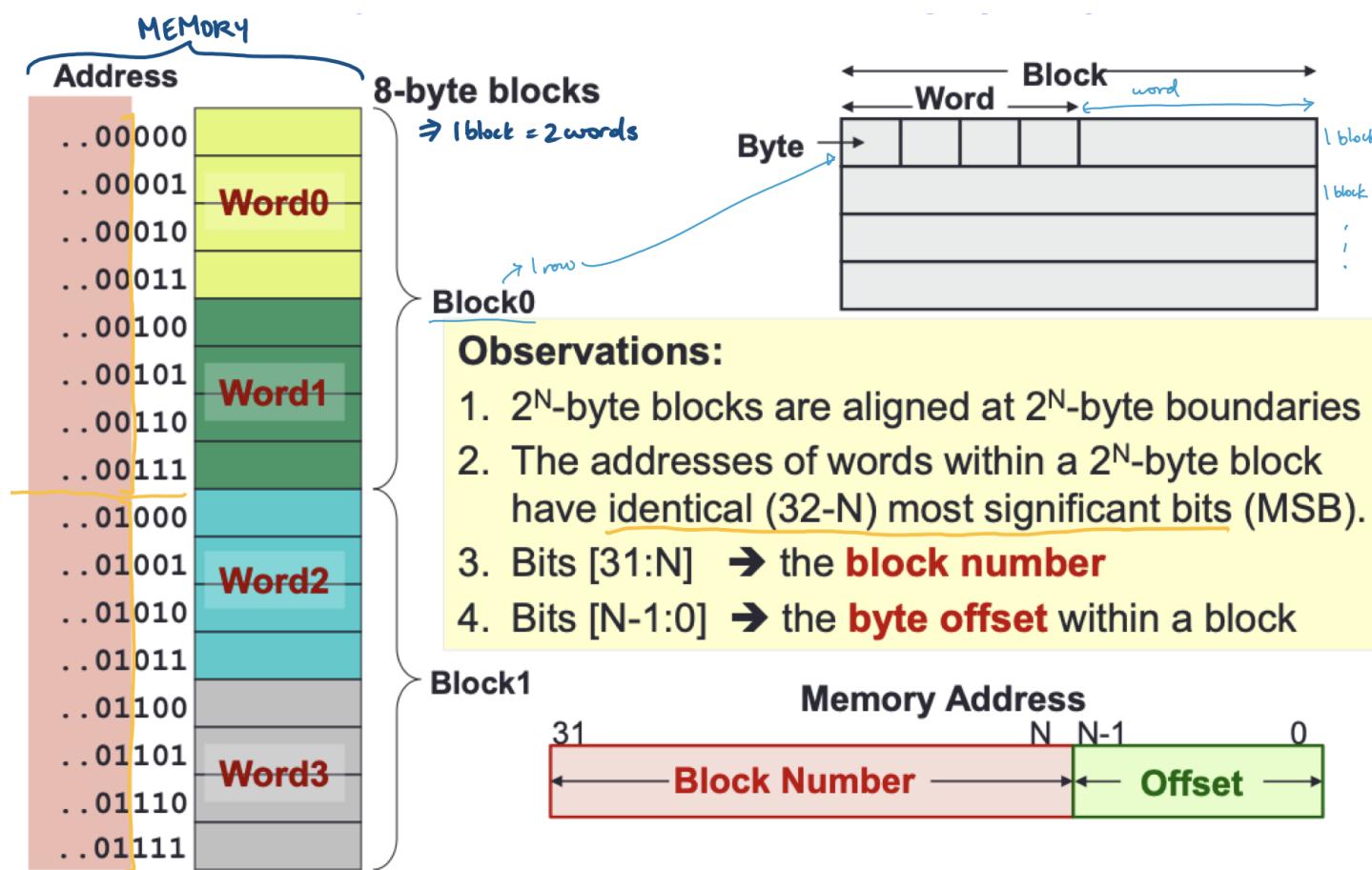


💡 let h be the hit rate required to sustain an average access time t , where the SRAM cache has access time s ns and the DRAM cache has access time d ns.

$$t = (h * s) + (1 - h) \times (d + s)$$

27. Direct Mapped Cache

- cache block/cache line → unit of transfer between memory and cache
- cache contains:
 - the data block (line)
 - tag of the memory block
 - valid bit indicating whether the cache line contains valid data

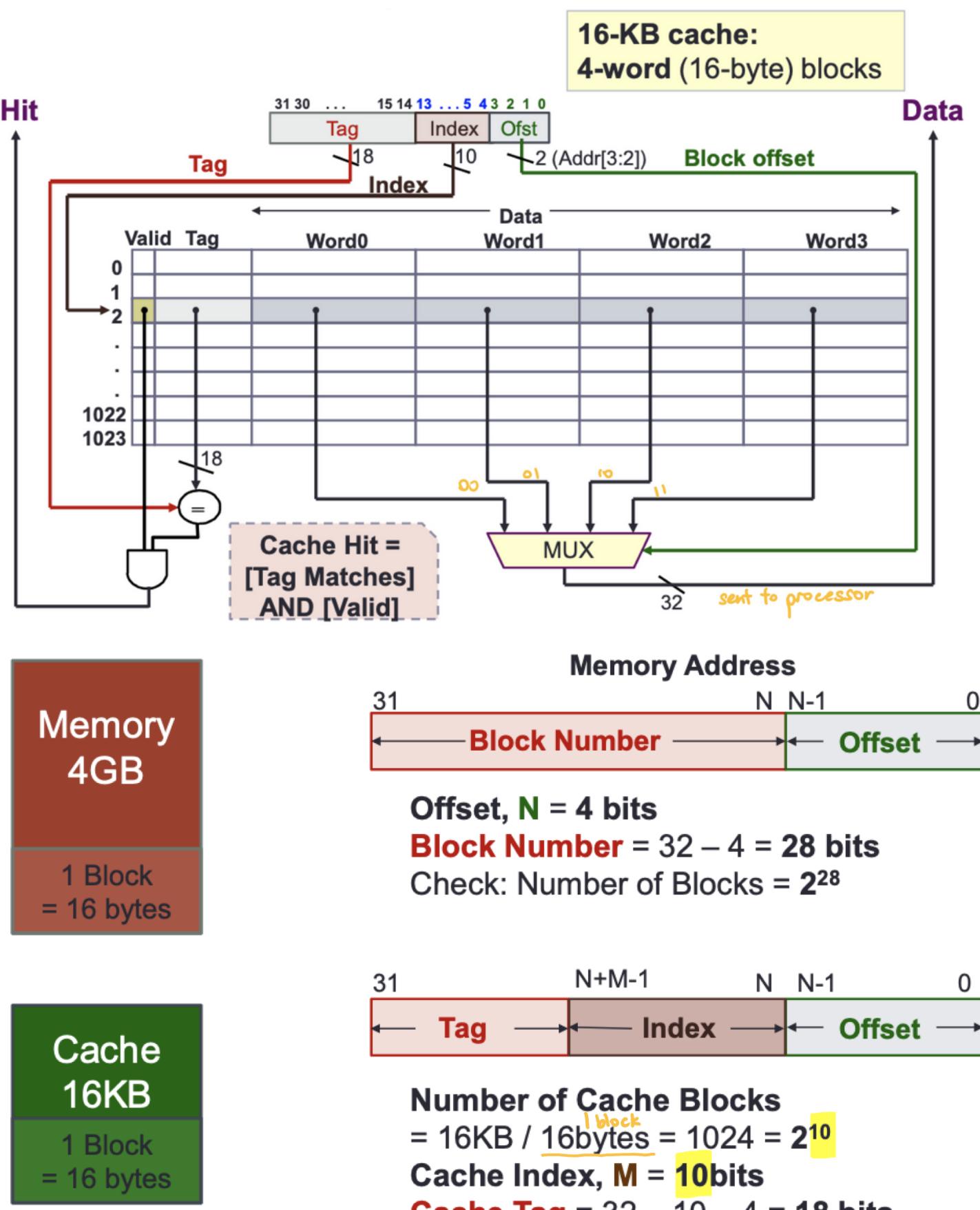


- map function:
 - cache index = (BlockNumber) mod (NumberOfCacheBlocks)



- memory address
 - offset → which byte within the block
 - index → which line in the cache
 - tag → which block is mapped to that line

cache circuitry



Types of Cache Misses

- compulsory miss → on first access to a block - the block must be brought into the cache
 - aka cold start miss / first reference miss
- conflict miss → when several blocks are mapped to the same block/set
 - i.e. if the same address is accessed
 - aka collision miss / interference miss
- capacity miss → when blocks are discarded from cache as cache cannot contain all blocks needed

Writing Data

- Solution 1: Write-through cache** (write to both cache & mem)
 - put a write buffer between cache and main memory
- Solution 2: Write-back cache** (only write to mem on block replace)
 - add an additional dirty bit to each cache block

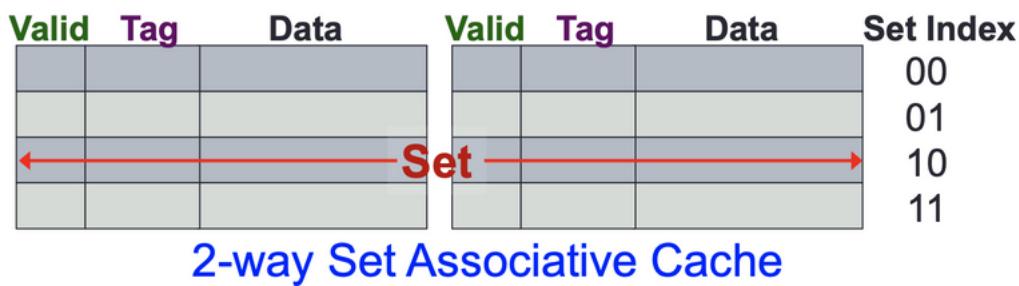
Handling Cache Misses

- on read miss:
 - load data into cache, then load from cache to register
- on write miss: (data to write to is not in the cache)
 - option 1: write allocate**
 - load complete block into cache
 - change only the required word in cache
 - write policy handles writing to main mem
 - option 2: write around**
 - bypass the cache, write to main memory
 - do not load block to cache

28. Set Associative Cache

Set Associative Cache

- solves conflict miss
- **N-way Set Associative Cache** → a memory block can be placed in a fixed number, N of locations in the cache (where $N > 1$)
 - aka each set contains N cache blocks



Set Index	Block 0				Block 1			
	Valid	Tag	word offset=000	word offset=100	Valid	Tag	word offset=000	W1
0	1	0	M[0]	M[4]	0	1	2	M[32] M[36]
1	1	0	M[8]	M[12]	0			

- cache set index = (BlockNumber) mod (numberOfCacheSets)



Cache Block size = 2^N bytes

Number of cache sets = 2^M

Offset = N bits

Set Index = M bits

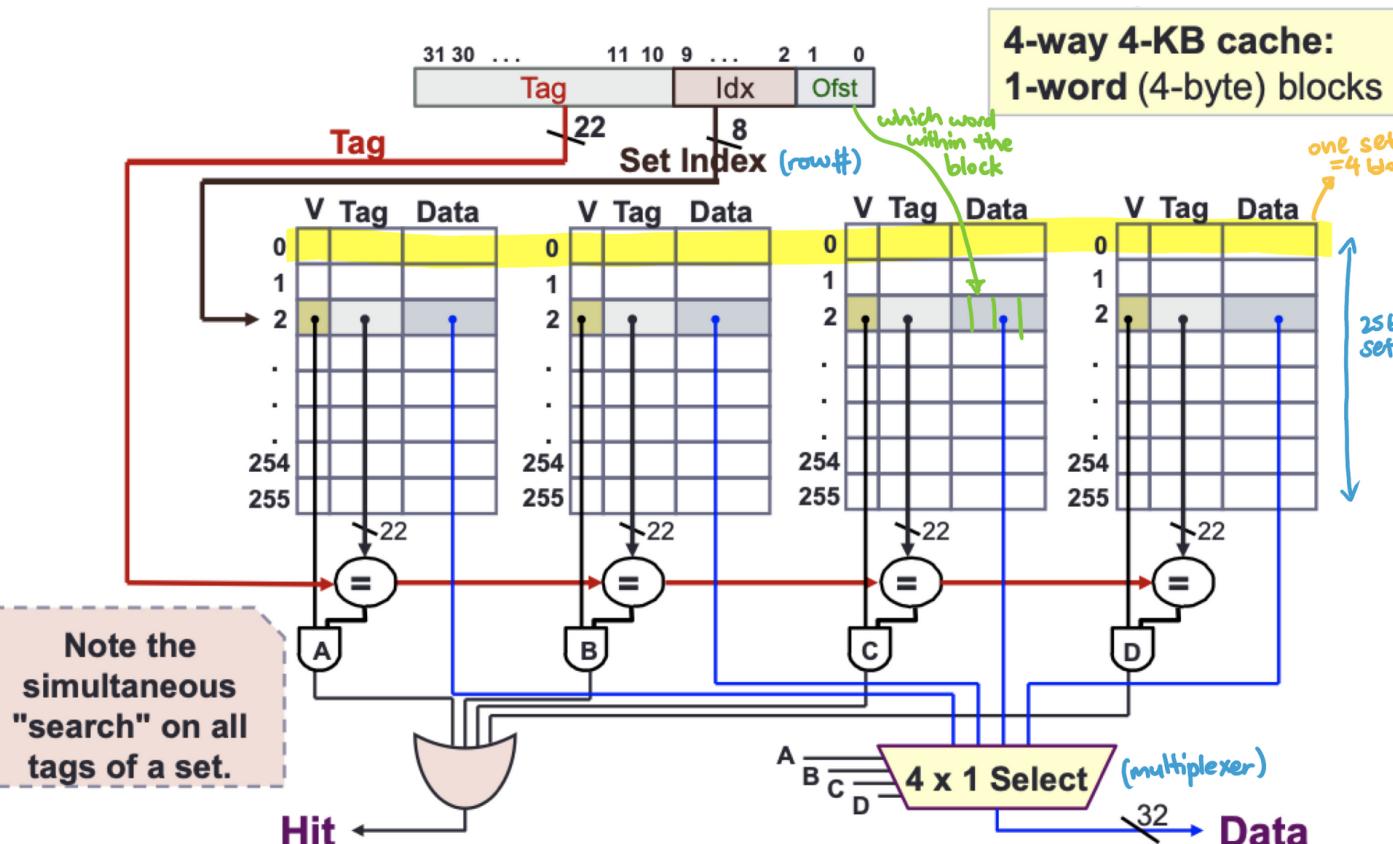
Tag = $32 - (N + M)$ bits

Observation:
It is essentially
unchanged from the
direct-mapping formula

- performance: direct-mapped cache of size N has the same miss rate as a 2-way set associative cache of size $N/2$

circuitry

- how it works
1. use address Set Index to identify the set
 2. for each block in the set, check if valid bit = 1 and check if tag matches tag in address



Block Replacement Policy

- for SA/FA cache - can choose where to place a memory block
- **Least Recently Used (LRU)** → temporal locality
 - replace the block that has not been accessed for the longest time
 - disadvantages: hard to keep track if there are many choices
- other policies:
 - FIFO (first in first out)
 - RR (random replacement)
 - LFU (least frequently used)

29. Fully Associative Cache

github/joyntls

Fully Associative Cache

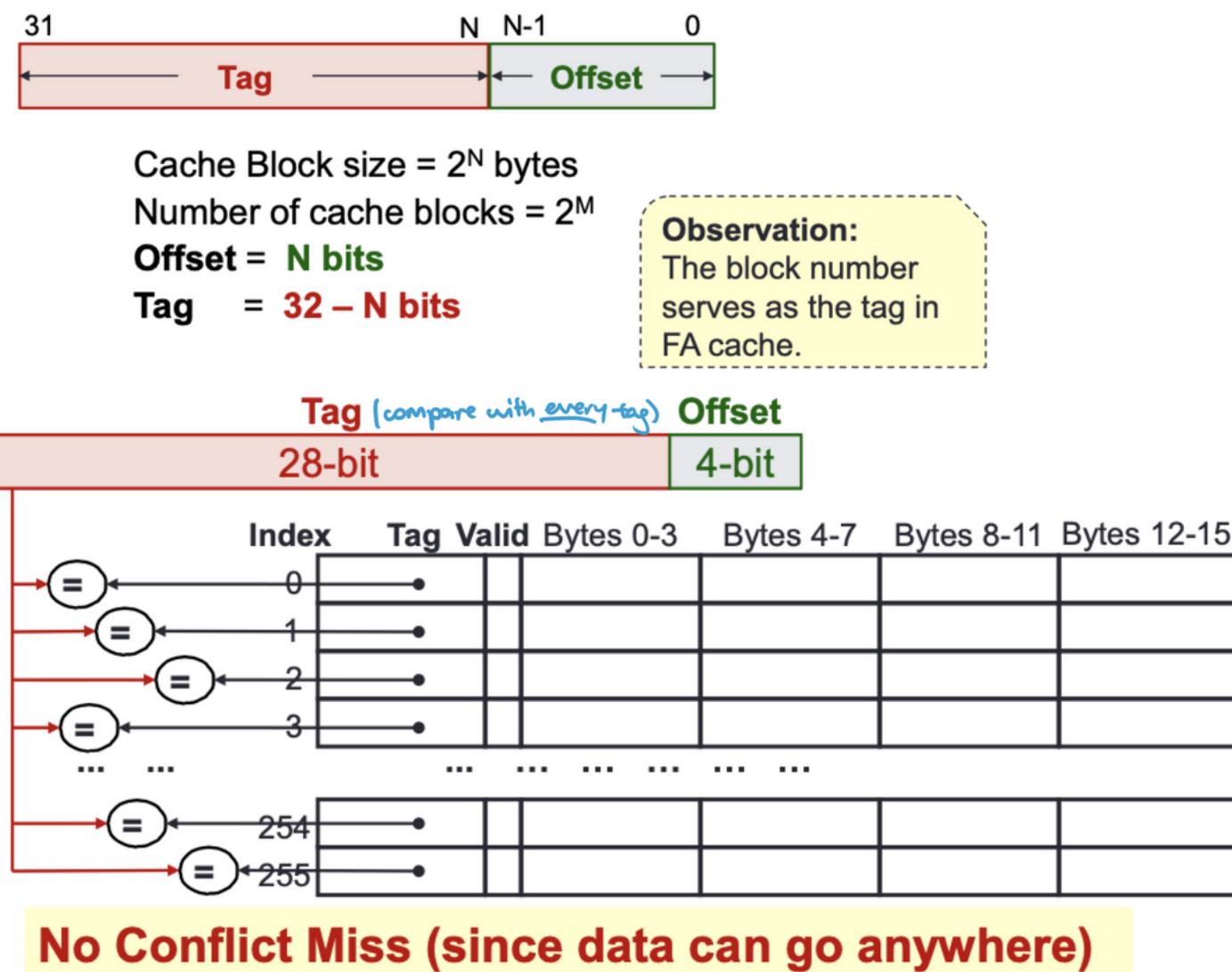
- **Fully Associative Cache** → a memory block can be placed in any location in the cache
- no mapping function (tag=block number)
 - ✓ memory block can be placed in any location
 - ✗ need to search all cache blocks for memory access
 - hence usually for small cache
- no conflict miss!

performance

- cold/compulsory miss: remains the same regardless of cache size/associativity
 - conflict miss = 0 for FA cache
 - decreases when associativity increases, given the same cache size
 - capacity miss:
 - remains the same regardless of associativity
 - decreases as cache size increases
- total miss = cold miss + conflict miss + capacity miss

 for FA, since conflict miss = 0,
capacity miss (FA) = total miss (FA) - cold miss (FA)

Circuitry



Cache Organisation Summary

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data
0						
1						

Eight-way set associative (fully associative)

Tag	Data										

Block Placement: Where can a block be placed in cache?

Direct Mapped:

- Only one block defined by index

N-way Set-Associative:

- Any one of the N blocks within the set defined by index

Fully Associative:

- Any cache block

Block Identification: How is a block found if it is in the cache?

Direct Mapped:

- Tag match with only one block

N-way Set Associative:

- Tag match for all the blocks within the set

Fully Associative:

- Tag match for all the blocks within the cache

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:

- No Choice

N-way Set-Associative:

- Based on replacement policy

Fully Associative:

- Based on replacement policy

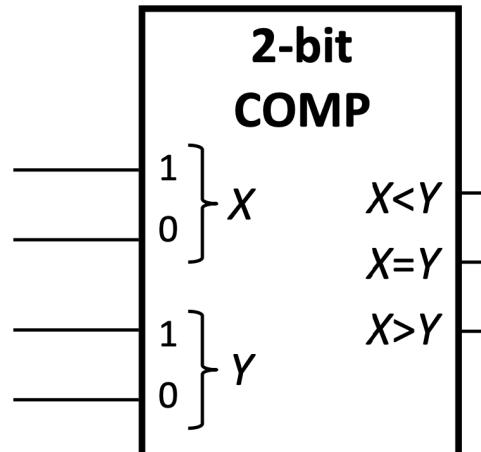
Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

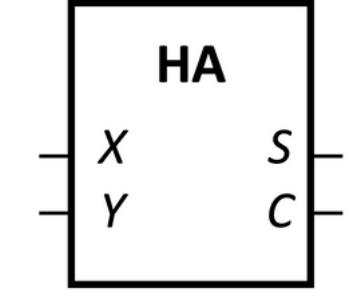
Write Miss Policy: Write allocate vs write no allocate

Block Diagrams

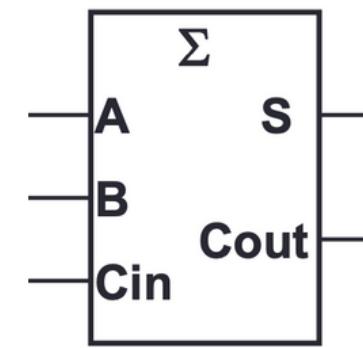
comparators



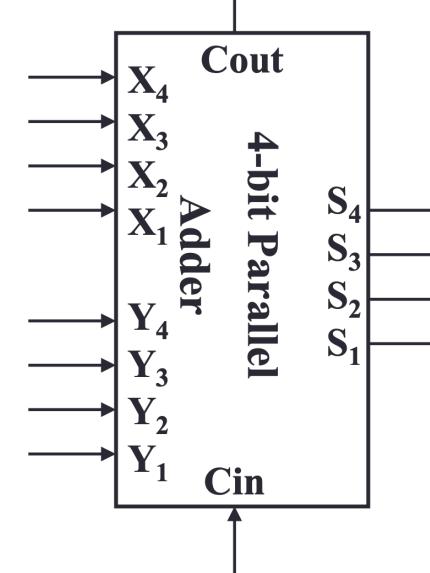
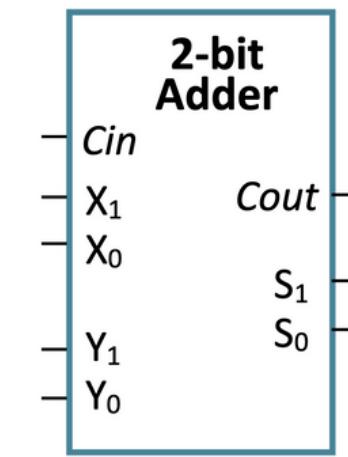
half adder



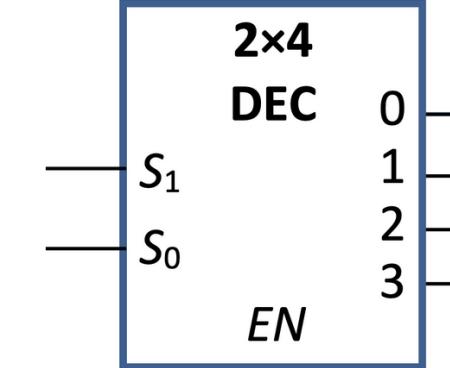
full adder



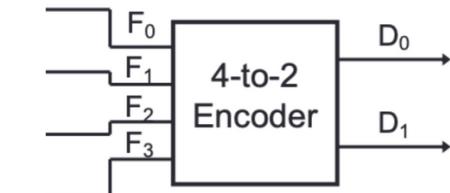
other adders



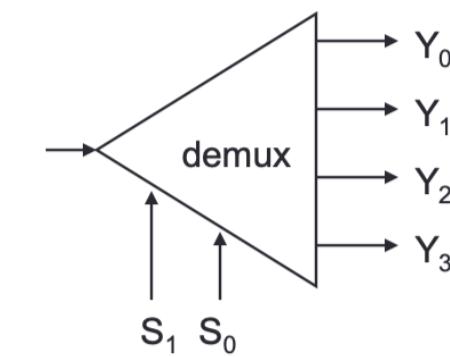
decoder



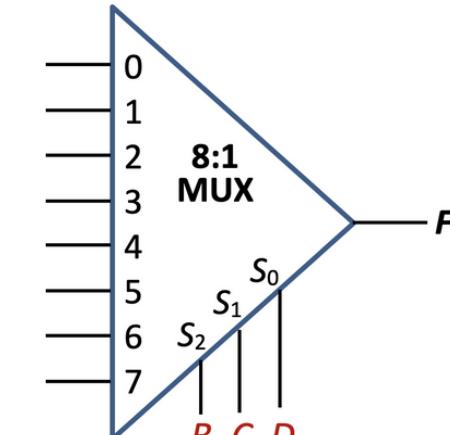
encoder



demultiplexer



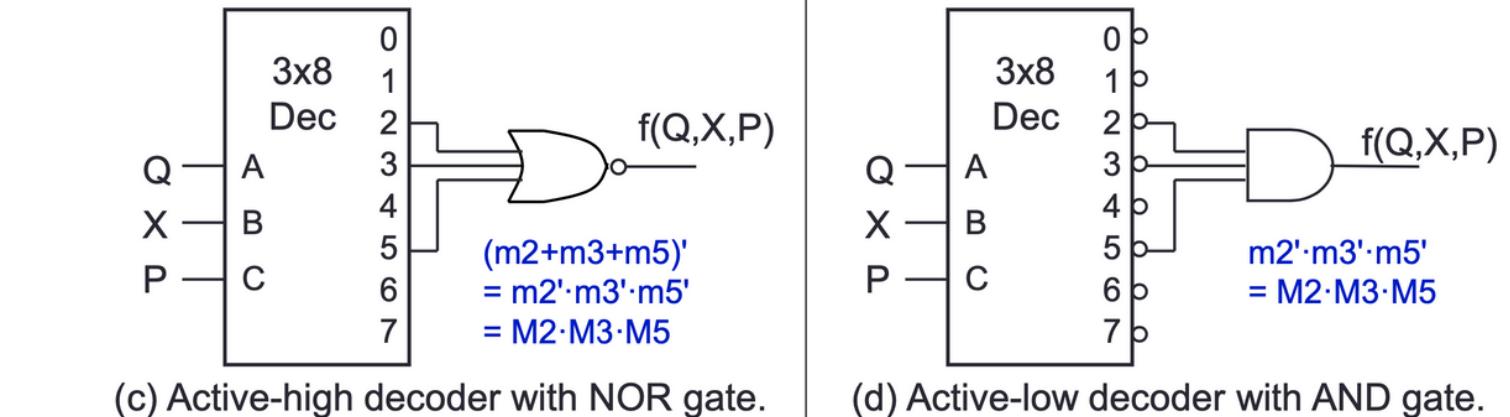
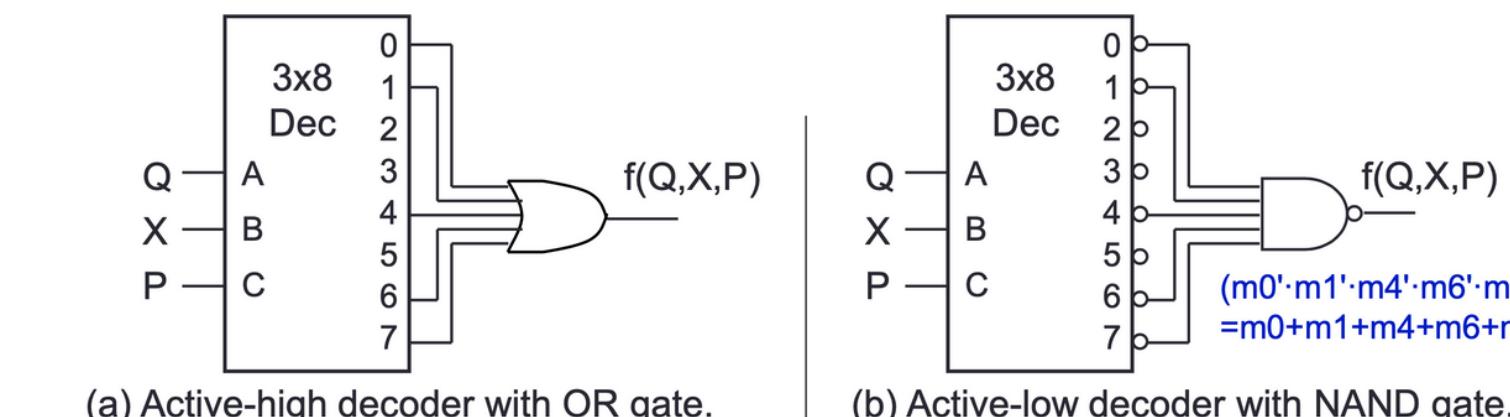
multiplexer



decoder functions

(2/2)

$$f(Q, X, P) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$$



priority encoder

- Example of a 4-to-2 priority encoder:

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

random examples

- Example: How is -6.5_{10} represented in IEEE 754 single-precision floating-point format?

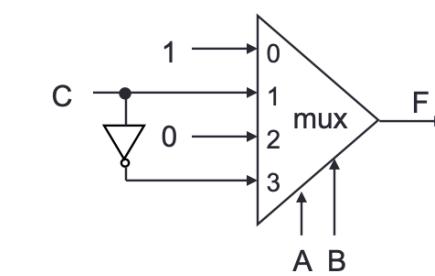
$$-6.5_{10} = -110.1_2 = 0.101_2 \times 2^2$$

$$\text{Exponent} = 2 + 127 = 129 = 10000001_2$$



- Draw the truth table for function, by grouping inputs by selection line values, then determine multiplexer inputs by comparing input line (C) and function (F) for corresponding selection line values.

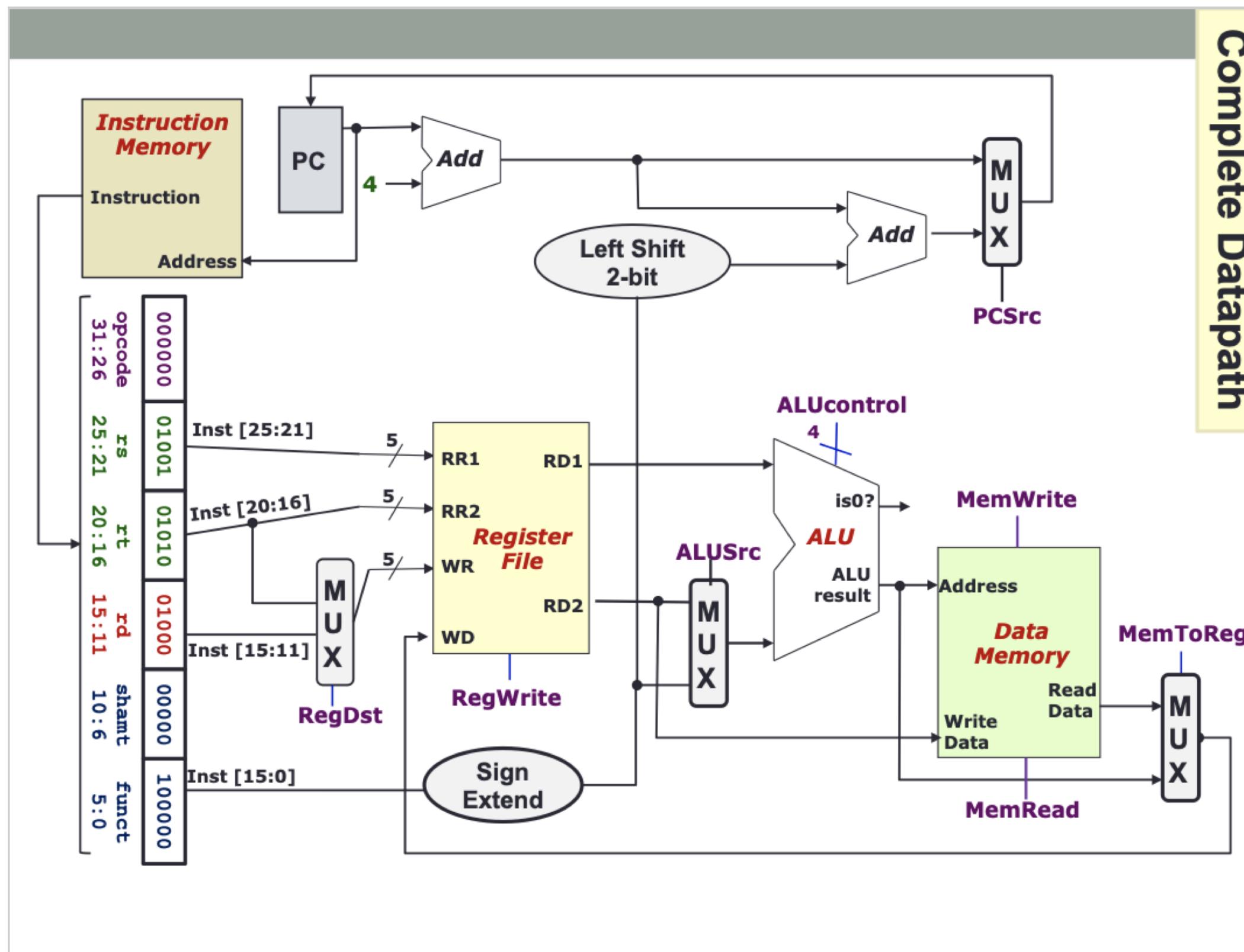
A	B	C	F	MUX input
0	0	0	1	1
0	0	1	1	C
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	C
1	1	0	1	1
1	1	1	0	C'



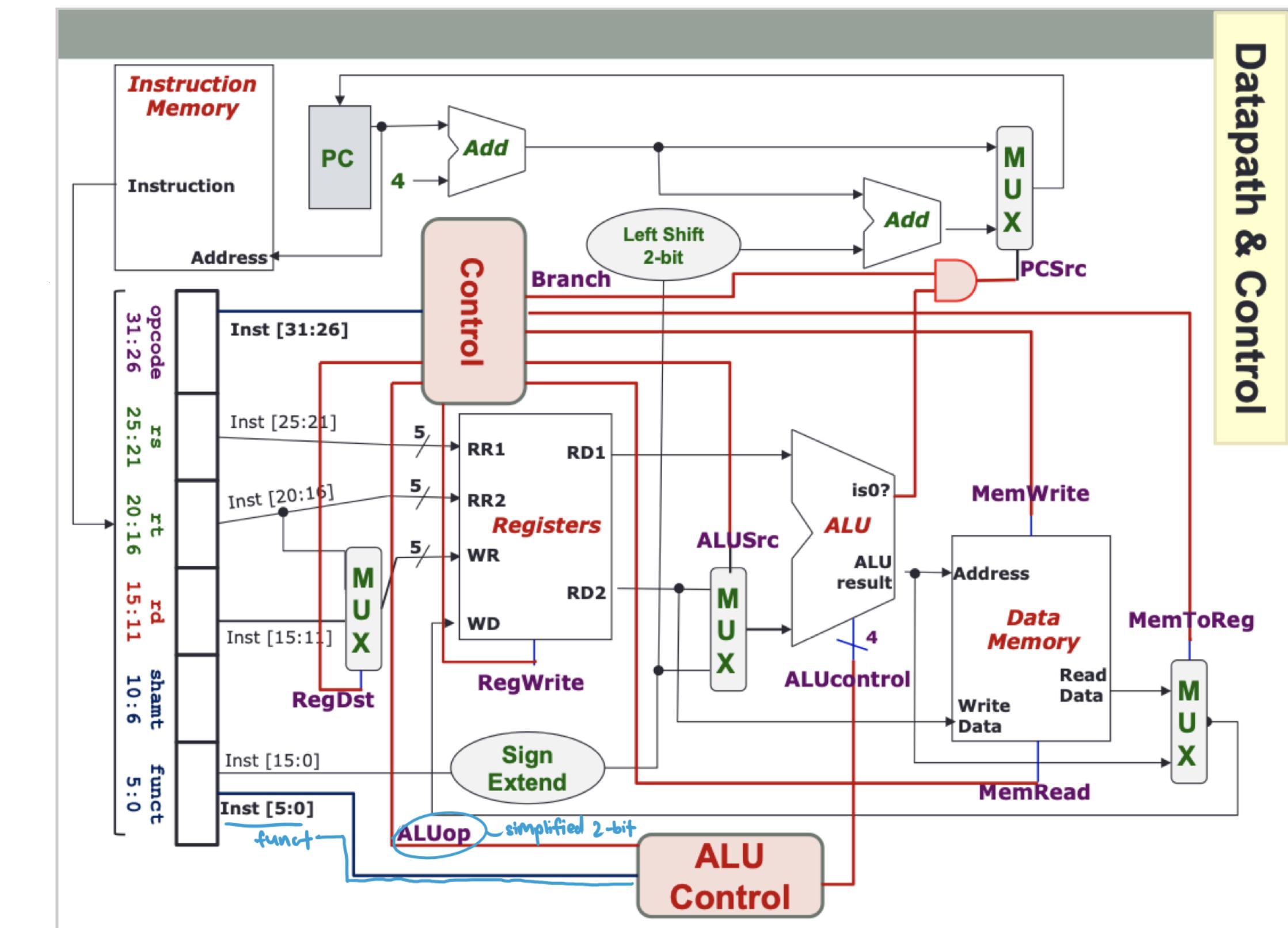
Complete Datapath Diagram

github/joyntls

▼ datapath



▼ datapath & control



	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

Control Flow Determination

github/jovyntls

Control Signals

(can all be generated using opcode directly)

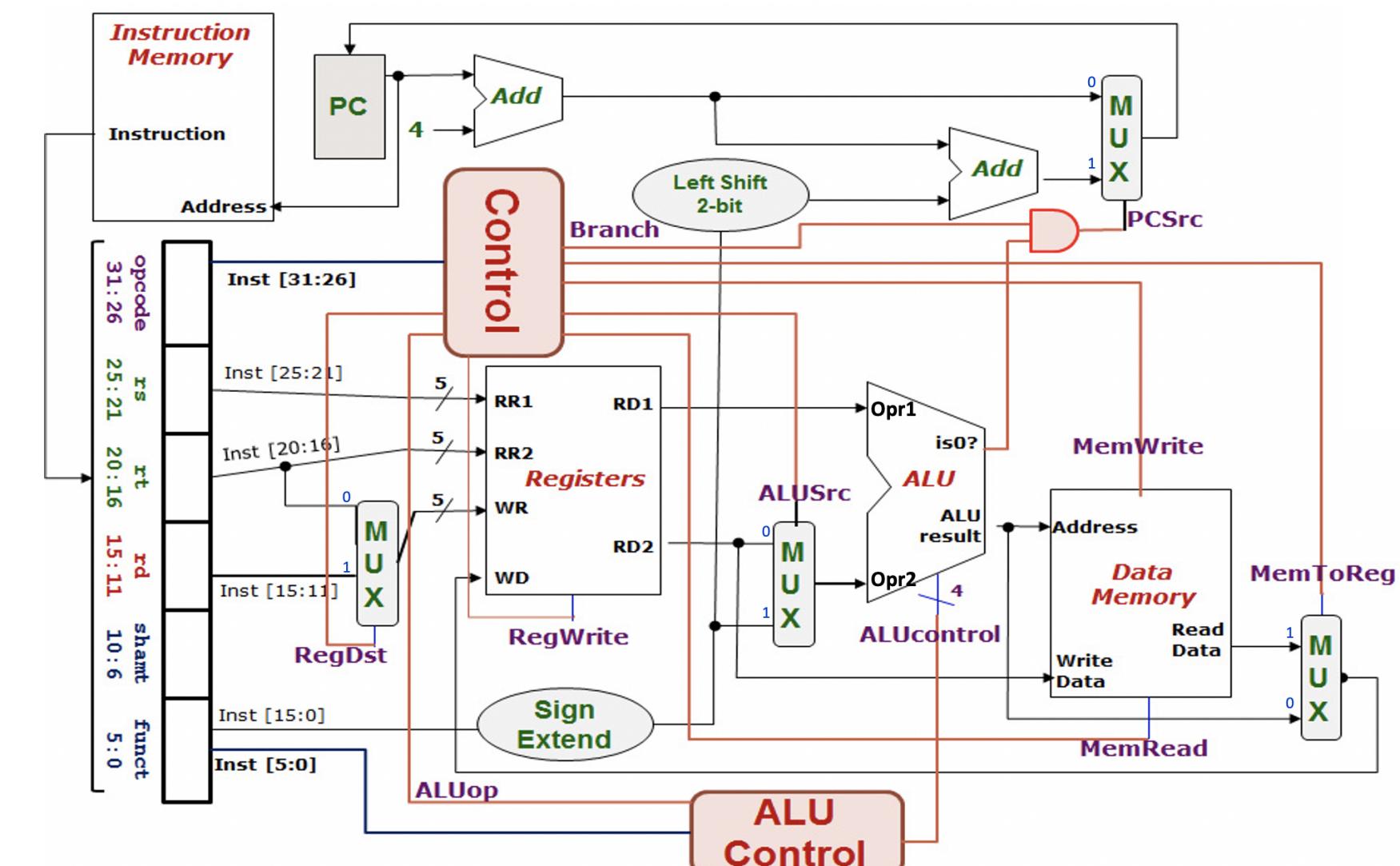
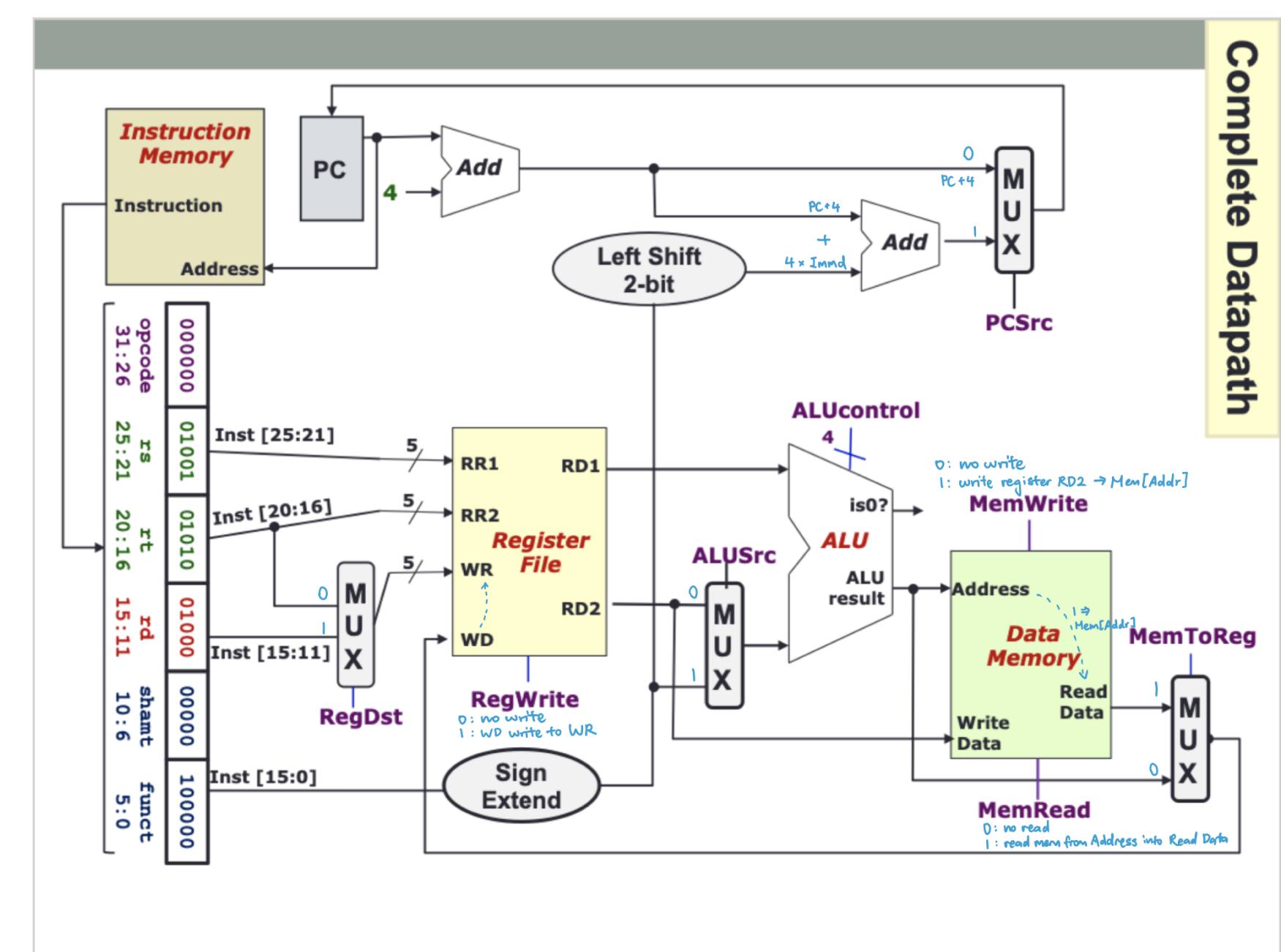
- **RegDst**
 - 0 → write register = `Inst[20:16]`
 - 1 → write register = `Inst[15:11]`
 - **RegWrite**
 - 0 → No register write
 - 1 → New value will be written (WD written to WR)
 - **ALUSrc** - determines first input of ALU
 - 0 → `Operand2 = Register Read Data 2`
 - 1 → `Operand2 = SignExt(Inst[15:0])` (sign ext immediate)
 - **MemRead**
 - 0 → not performing memory read access
 - 1 → read memory using *Address* (returned in *Read Data*)
 - **MemWrite**
 - 0 → not performing memory write operation
 - 1 → write Register Read Data 2 into memory[*Address*]
 - **MemToReg** - chooses what to be written back into register
 - 0 → register write data = ALU result
 - 1 → register write data = memory read data
 - **PCSrc**
 - 0 → next PC = PC + 4
 - 1 → next PC = `SignExt(Inst[15:0]) << 2 + (PC + 4)`
 - what it does
 - PCSrc = set to 1 if *Branch* AND *is0* are both 1
 - aka (*isBranchInstruction* AND *branchIsTaker*)

	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	Writing to address (NOT reg register)	1	X	0	0	1	0	0
beq	X	0	X	0	0	0	1	0	1

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	X	add	0010
sw	00	store word	X	add	0010
beq	01	branch equal	X	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



K-Maps

 [github/joyvntls](https://github.com/joyvntls)

3-Variable

		bc	
		00	01
a	0	m_0	m_1
	1	m_4	m_5
		11	10
m_3	m_2	m_7	m_6

4-Variable

		yz	
		wx	x
w	0	m_0	m_1
	1	m_4	m_5
		11	10
m_3	m_2	m_7	m_6
		01	11
m_{12}	m_{13}	m_{15}	m_{14}
		10	00
m_8	m_9	m_{11}	m_{10}
		z	
m_{12}	m_{13}	m_{15}	m_{14}
		y	
m_8	m_9	m_{11}	m_{10}

5-Variable

		V=0			
		WX	YZ	00	01
W	0	m_0	m_1	m_3	m_2
	1	m_4	m_5	m_7	m_6
		11	10	m_9	m_{10}
m_{12}	m_{13}	m_{15}	m_{14}	m_{24}	m_{25}
		10	00	m_{27}	m_{26}

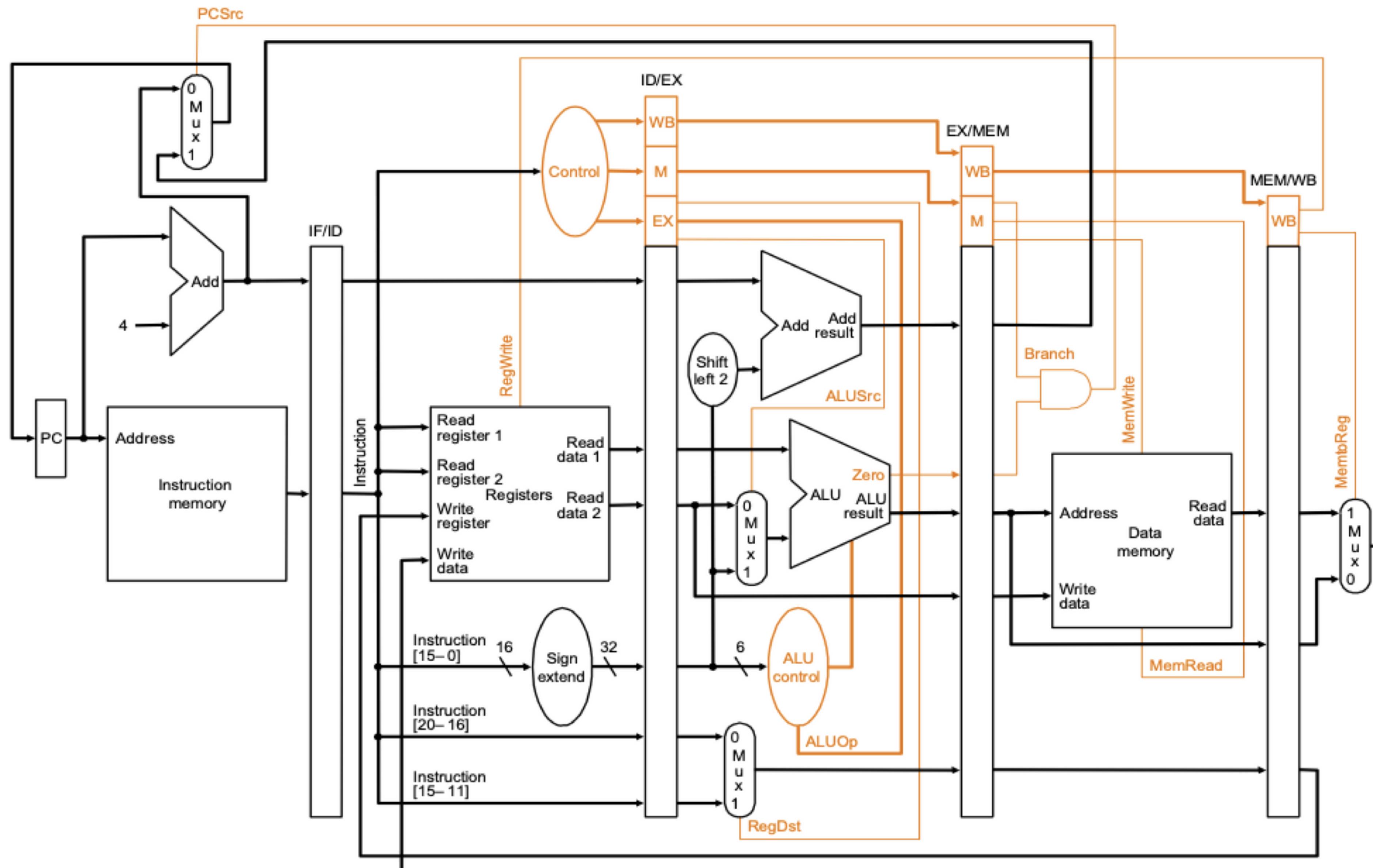
6-Variable

		V=1			
		WX	YZ	00	01
W	0	m_{16}	m_{17}	m_{19}	m_{18}
	1	m_{20}	m_{21}	m_{23}	m_{22}
		11	10	m_{28}	m_{29}
m_{12}	m_{13}	m_{15}	m_{14}	m_{31}	m_{30}
		10	00	m_{24}	m_{25}

		a'b'			
		cd	ef	00	01
a	0	m_0	m_1	m_3	m_2
	1	m_4	m_5	m_7	m_6
		11	10	m_{12}	m_{13}
m_{12}	m_{13}	m_{15}	m_{14}	m_{15}	m_{14}
		10	00	m_8	m_9
		a'b			
		cd	ef	10	11
		m_{40}	m_{41}	m_{43}	m_{42}
		m_{44}	m_{45}	m_{47}	m_{46}
		m_{36}	m_{37}	m_{39}	m_{38}
		m_{32}	m_{33}	m_{35}	m_{34}
		a'b'			
		cd	ef	10	11
		m_{58}	m_{59}	m_{57}	m_{56}
		m_{62}	m_{63}	m_{61}	m_{60}
		m_{54}	m_{55}	m_{53}	m_{52}
		m_{50}	m_{51}	m_{49}	m_{48}

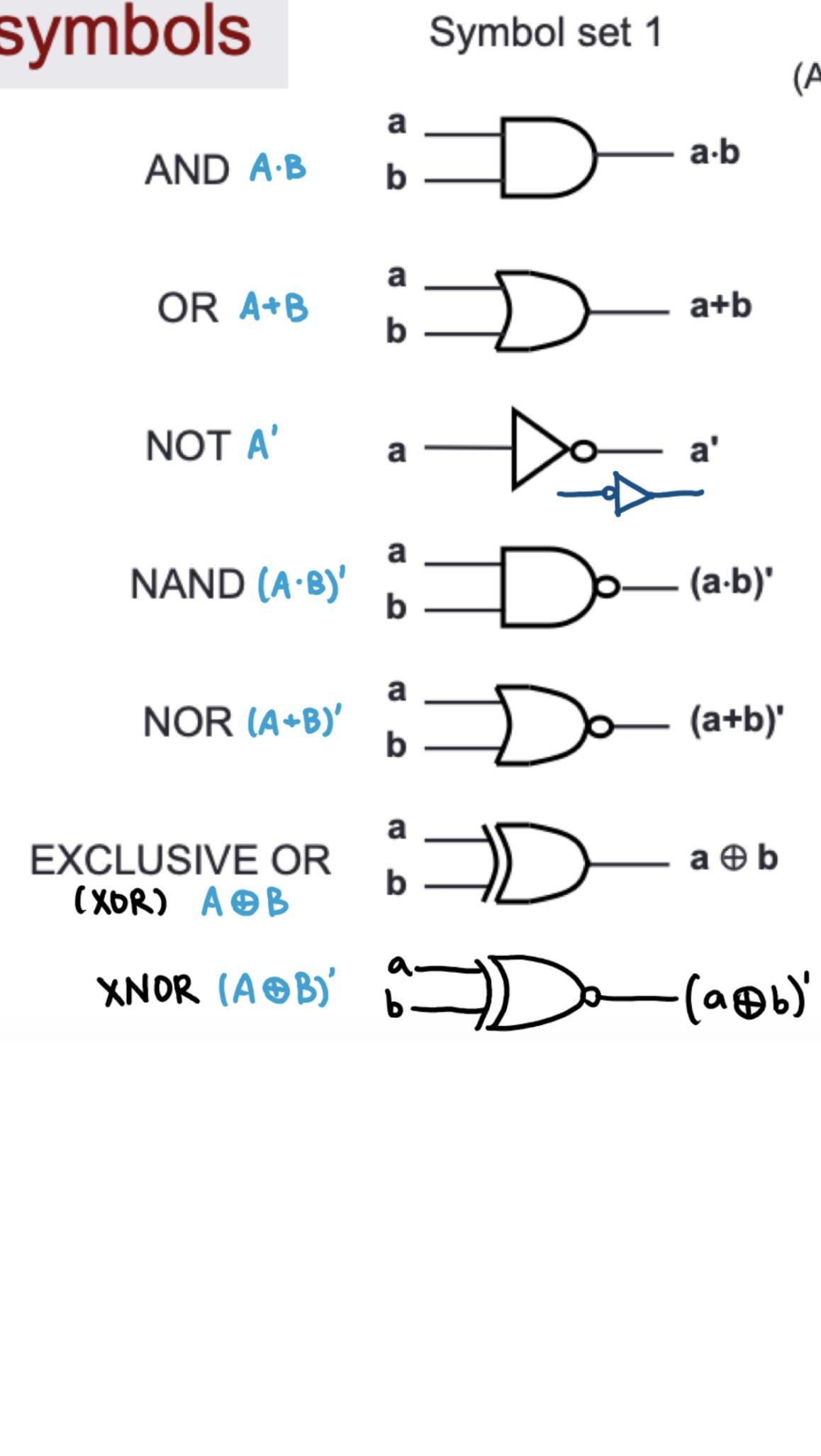
		AB			
		CD	00	01	11
A	0	$A' B' C' D'$	$A' B' C' D$	$A' B' C D$	$A' B' C D'$
	1	$A' B C' D'$	$A' B C' D$	$A' B C D$	$A' B C D'$
		11	$A B C' D'$	$A B C' D$	$A B C D$
		$A B' C' D'$	$A B' C' D$	$A B' C D$	$A B' C D'$
		10	$A B' C' D'$	$A B' C' D$	$A B' C D$

MIPS Pipeline with Control

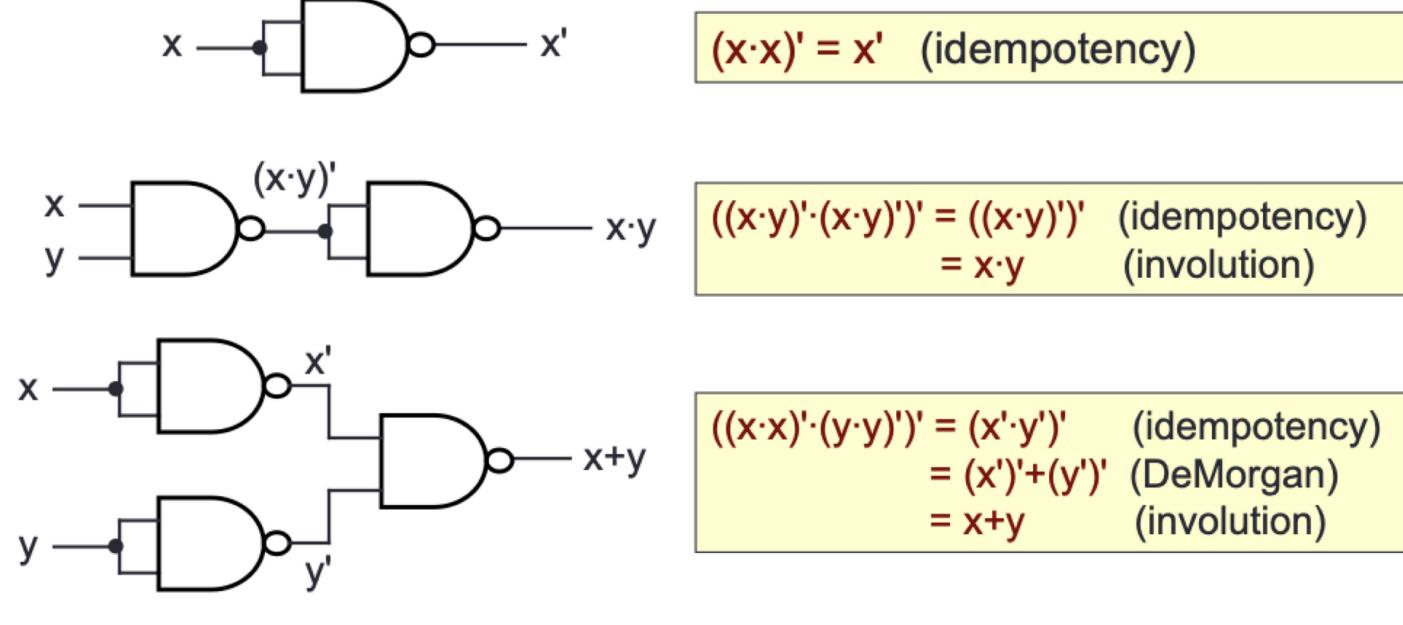


Universal Gates

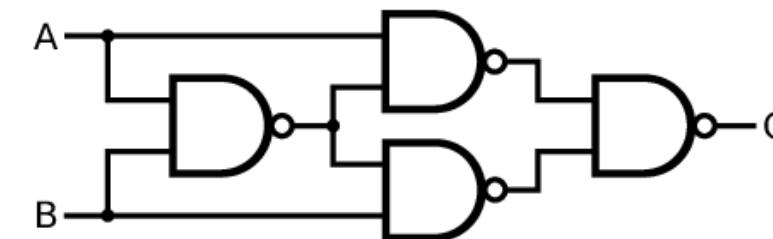
Gate symbols



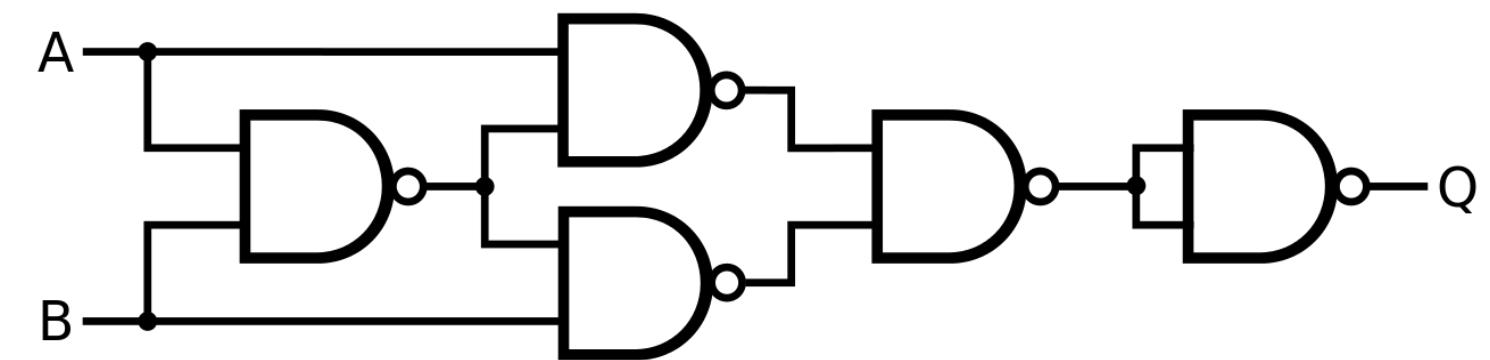
NAND



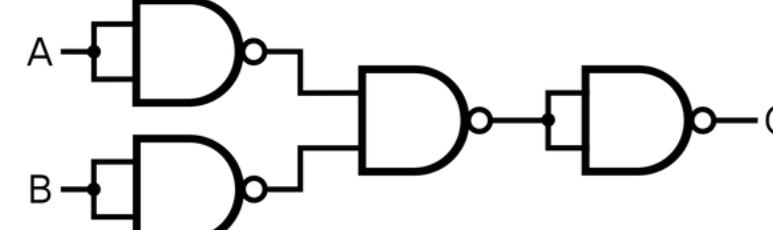
XOR



XNOR

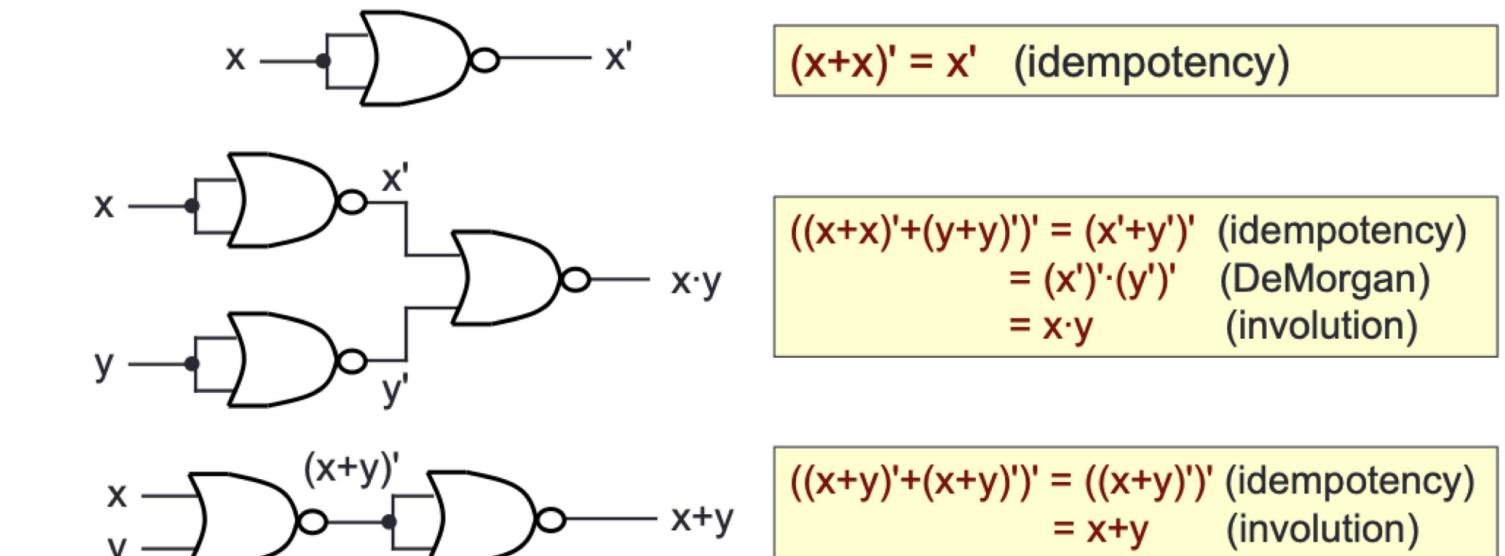


NOR

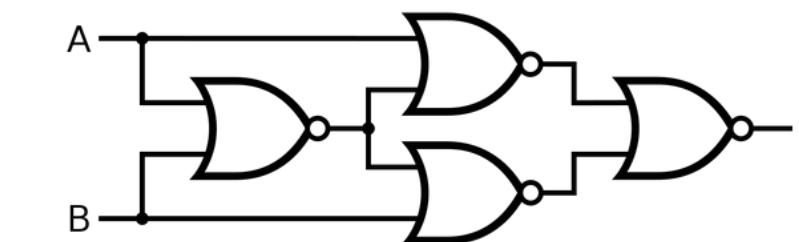


NOR

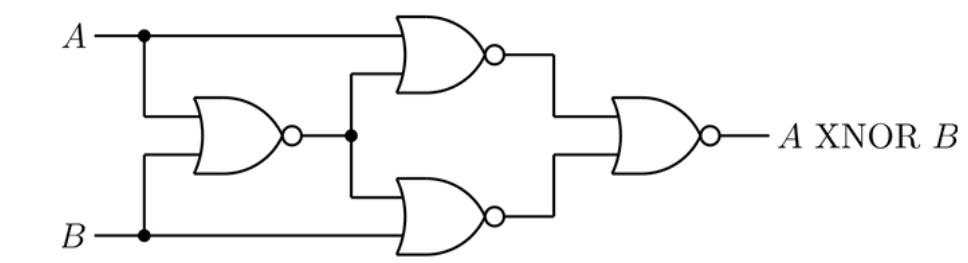
- Proof: Implement NOT/AND/OR using only NOR gates.



XOR



XNOR



NAND

