CS2100

Course Summary

The objective of this course is to familiarise students with the fundamentals of computing devices. Through this course students will understand the basics of data representation, and how the various parts of a computer work, separately and with each other. This allows students to understand the issues in computing devices, and how these issues affect the implementation of solutions. Topics covered include data representation systems, combinational and sequential circuit design techniques, assembly language, processor execution cycles, pipelining, memory hierarchy and input/output systems.

Topics

C

- Overview of C
- Number Systems
- Functions
- Pointers
- Arrays
- Strings
- Structures

MIPS

- Introduction To MIPS
- Memory Instructions
- Decision Making & Loops
- Instruction Formats
- Memory Instructions
- Decision Making & Loops
- Instruction Formats
- Datapath
  - Datapath (Summary)
- Control
- MIPS Instruction Set

Logic

- Boolean Algebra
- Logic Circuit
- Simplification
  - Quine-McCluskey (optional)
- MSI Components
- Combinational Circuits
- Sequential Logic

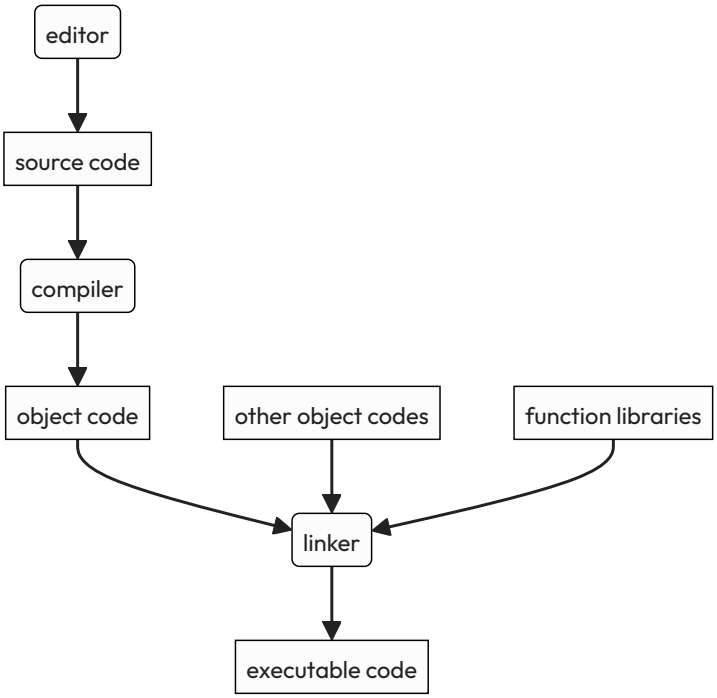
Memory

- Memory
- Pipelining
- Cache

Overview of C

> **C Programming language**
>
> Imperative procedural language.
> 80

Process

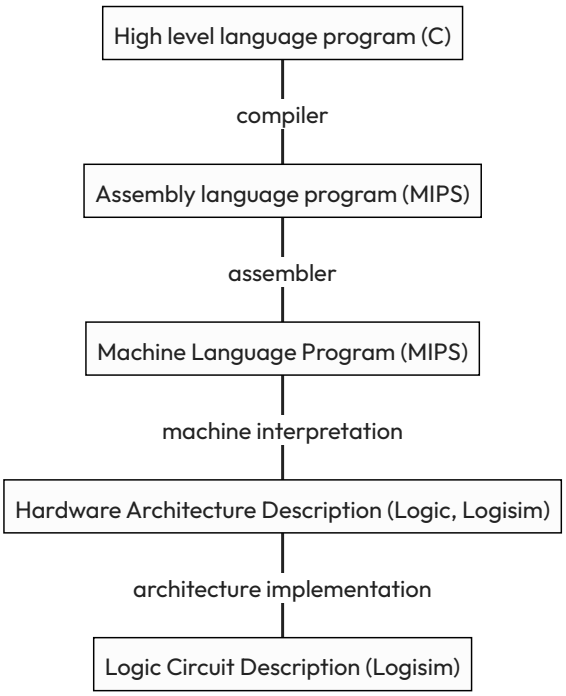


An example of a compiler is `gcc` .

To display all details: `gcc -v HelloWorld.c`

- Preprocessing
- Compilation
- Assembler
- Linker

Abstraction



Computation

Function

A function body has two parts:
Declaration statement: What type of memory cells needed
Executable statement: Describe processing on memory cells

Declaration statement

```
int count
```

User defined identifier - name of variable of function.

- Must not begin with a digit, but can contain letters, digits and underscores
- Case sensitive
- Guideline: start with lowercase letter

Reserved words `int, void, double, return` etc
Standard identifiers `printf, scanf` etc

Executable statement

`printf, scanf`
Computational and assignment statements

`lvalue = assigned value`
Stores a value or a computational result in a variable. `lvalue` must be assignable.
Assignment can be cascaded, with associativity from right to left: `b = (c = 3 + 6)`

Side-effect An assignment statement also returns the value of the right-hand side.

Arithmetic operations

Precedence rule

1. Primary expression operators (Left to right)
    - `() expr++ expr --`
2. Unary operators (Right to left)
    - `* & + - ++expr --expr (typecast)`
3. Binary operators
    - `* / %`
    - `+ -`
4. Ternary operator
    - `? :`
5. Assignment operators
    - `= += -= *= /= %=`

Execution:

- Left to right
- Parenthesis rule
- Precedence rule
- Associative rule
- Truncation (if result can be stored)
    - `int n; n = 9 * 0.5` results in `4` being stored in `n`

Typecasting

Uses a cast operator to change the type of an expression
`int aa = 10; float pp = (float) aa / 4;`

Modulo vs Remainder

Python `%` is a modulo, while C's `%` is a remainder. This results in a different value for negative values.

In Python:

```
a = 10 % 4  # a = 2
b = -10 % 4 # b = 2
```

In C:

```
a = 10 % 4  // a = 2
b = -10 % 4 // b = -2
```

Selection Structures

There are two selection structures:

```
if () {

}
elif () {

}
else {

}
```

```
switch () {
        case x:
                ...
                break;
        case y:
```

```
                ...
                break;
        default:
                ...
                break;
}
```

Relational Operators `< <= > >= == !=`
Truth Values `true false` - `false` is represented as `0`, and any other value can represent `true`
Logical Operators `&& || !`

> **Short-circuit evaluation**
>
> For AND statements `expr1 && expr2`
>
> > **Short-circuit evaluation**
> >
> > if `expr1` is false, skip evaluating `expr2` and return false immediately
>
> For OR statements `expr1 || expr2`
>
> > **Short-circuit evaluation**
> >
> > if `expr1` is true, skip evaluating `expr2` and return true immediately

Repetition

```
while ()
{

}
```

```
do {

} while ()
```

```
for (initalization; condition; update) {

}
```

`break` only breaks out of the inner-most loop containing the statement in a nested loop.
Similarly, `continue` skips to the next iteration of the inner-most loop containing the statement in a nested loop

Number Systems

Data Representation

Data is internally represented as a sequence of bits - 0 or 1.

- Byte: 8 bits
- Word: Multiple of bytes (differs per architecture)

Calculation of values

n bits can represent up to 2^n values.
To represent m values, $\lceil log_2 m \rceil$

Number Systems

Weighted-positional number system with a **base/radix** n where each position has a weight of power of n.

- Decimal (base 10)
  - Weights in power of 10
- Binary (base 2)
- Octal (base 8)
- Hexadecimal (base 16)
  - Extra digits: A, B, C, D, E, F to refer to 10, 11, 12, 13, 14, 15 respectively.

Notations

- Prefix 0 for octal
- Prefix 0x for hexadecimal in C and QTSpim
- Prefix 8'b for binary in Verilog
- Prefix 8'h for hexadecimal in Verilog
- Prefix 8'd for decimal in Verilog

ASCII Code

ASCII and Unicode are used to represent characters.

Note that in C, integers and characters are somewhat interchangable:

```
printf("%c", 65) // A
printf("%d", A) // 65
```

Format specifiers

When using printf, using a format specifier `%[flags][width][.precision][length]specifier` allows for a format to be specified for a variable.

For example, `printf("%s", str)` specifies `str` to be printed as a string. Using multiple format specifiers could also work: `printf("%d %x %o", 100, 100, 100)` prints the following parameters in the specified formats of decimal, hexadecimal and

octal, in that order.

Negative Numbers

Unsigned numbers do not allow for non-negative values, but signed numbers do.

The 3 common representations of signed binary numbers are as following -
Sign-and Magnitude

sign bit	magnitude bit						

The example given above has a 1-bit sign, and a 7-bit magnitude format.

With such an implementation, the range for a $n-$bit sign-and magnitude representation is $-(2^{n-1}-1)$ to $2^{n-1}-1$.

For operations: to negate a number, just invert the sign bit.

1s complement

Given a number x expressable as an $n-$bit binary number, the negated value is obtained using $-x = 2^n - x - 1$.

With such an implementation, the range for a $n-$bit 1s complement representation is $-(2^{n-1}-1)$ to $2^{n-1}-1$. The most significant bit represents the sign similarly - 0 for positive, 1 for negative.

For operations: to negate a number, invert all the bits.

> **Tip**
>
> Use the formula to get the negated value and convert it into binary for fast calculation.
>
> **In 8-bit 1s complement**
>
> $-100 = 2^8 - 100 - 1 = 155 = 10011011$

2s complement

Given a number x expressable as an $n-$bit binary number, the negated value is obtained using $-x = 2^n - x$.

With such an implementation, the range for a $n-$bit 1s complement representation is $-(2^{n-1})$ to $2^{n-1}-1$. The most significant bit represents the sign similarly - 0 for positive, 1 for negative.

For operations: to negate a number, invert all the bits, and add 1.

> **Tip**
>
> Use the formula to get the negated value and convert it into binary for fast calculation.
>
> **In 8-bit 2s complement**
>
> $-100 = 2^8 - 100 = 156 = 10011100$

Radix complement

The 1s and 2s complement can be abstracted into any base/radix.

Diminished radix complement

Given the number of digits n, and a radix b, the $(b-1)s$ complement is:

$$-x = b^n - x - 1$$

1s complement, for base 2 ($b = 2, b - 1 = 1$)

Radix complement

Given the number of digits n, and a radix b, the $(b)s$ complement is:

$$-x = b^n - x$$

2s complement, for base 2 ($b = 2$)

Excess representation

Allows range of values to be distributed evenly between positive and negative values by a simple translation.

Example

For a 3-bit excess representation, the first entry 000 can be represented as the lowest number -4 and the last entry 111 represented as the highest number 3, which is a translation of -4. This specific representation is called **Excess-4 representation on 3-bit numbers**

Algorithms

Overflow

Since signed numbers are of a fixed range, if the result of a computation goes beyond this range, an overflow occurs. This can be detected easily by ensuring that

- +ve + +ve MUST be +ve
- -ve + -ve MUST be -ve

2s complement

Addition of integers $A + B$

1. Perform binary addition on two numbers
2. Ignore carry out
3. Check for overflow - overflow occurs if
    - carry in and carry out of MSB are different OR
    - result is opposite sign of A and B

Subtraction of integers $A - B$

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

1s complement

Addition of integers $A + B$

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

Subtraction of integers $A - B$

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

Real Numbers

Fixed-point representation allocates a specific **fixed** number of bits for the whole number and fractional parts representatively. For example:

integer						fraction	

Floating-point representation allow us to represent very large or very small numbers based on range.

IEEE 754 Floating-Point
Contains 3 components:

- sign
- exponent
- mantissa (fraction)

The base/radix is assumed to be 2.
There are two formats:

- Single precision (32-bit)
    - 1-bit sign
    - 8-bit exponent (excess-127)
    - 23-bit mantissa
- Double precision (64-bit)
    - 1-bit sign
    - 11-bit exponent (excess-1023)
    - 52-bit mantissa

To represent a number in single-precision floating point format,

1. Convert it to binary
2. Reduce it to standard form ($x.y \times 2^z$)
3. Retrieve
    1. The sign (based on the negative or positive sign)
    2. The exponent (in excess-127 representation - $(z + 127)$)
    3. The mantissa (values after the binary point)

-6.5_{10} in IEEE 754 Floating-Point Rep

First, converting this to binary, we get -110.1_2.

Then, reducing it to standard form, we get $-1.101_2 \times 2^2$.

Next, the sign retrieved is 1, since the sign is negative. The exponent is 2, thus the excess-127 representation is 129, which is represented as 1000001. Lastly, we retrieve the mantissa by padding zeros to the back of 101, which results in the value 10100000000000000000000.

Putting it all together, we get the hexadecimal representation $C0D00000_{16}$

Conversions

Decimal to Binary (fractional)

Steps:

1. Convert decimal integral to binary
    1. Divide decimal number by 2 and store remainders in array
    2. Divide quotient by 2
    3. Repeat step 2 until quotient = 0
    4. Reverse all remainders
2. Convert decimal fractional to binary
    1. Multiply fractional decimal number by 2
    2. Integral part of resultant decimal number will be first digit of fraction binary number
    3. Repeat
3. Combine parts (concatenate together)

Converting decimal to binary

$n = 4.47, k = 3$

Step 1: Conversion of 4 to binary (Integral)

1. $4/2$: Remainder = 0 : Quotient = 2
2. $2/2$: Remainder = 0 : Quotient = 1
3. $1/2$: Remainder = 1 : Quotient = 0

So equivalent binary of integral part of decimal is 100.

Step 2: Conversion of $.47$ to binary (Fractional)

1. $0.47 \times 2 = 0.94$, Integral part: 0
2. $0.94 \times 2 = 1.88$, Integral part: 1
3. $0.88 \times 2 = 1.76$, Integral part: 1

So equivalent binary of fractional part of decimal is $.011$

Step 3: Combine the result of step 1 and 2.

Final answer can be written as:
$100 + .011 = 100.011$

Binary to Decimal (fractional)

Steps:

1. Convert binary integral to decimal
    1. Multiply each digit separately from left side of radix point by powers of two (from 0)
    2. Add everything together
2. Convert binary fractional to decimal
    1. Divide every digit separately from right side by powers of two (from 1)
    2. Add everything together
3. Add number together

Binary fractional back to decimal

n = 110.101

Step 1: Conversion of 110 to decimal **(binary integral)**
=> $1102 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$
=> $1102 = 4 + 2 + 0$
=> $1102 = 6$
So equivalent decimal of binary integral is 6.

Step 2: Conversion of .101 to decimal **(binary fractional)**
=> $0.1012 = (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$
=> $0.1012 = 0.5 + 0.0.125$
=> $0.1012 = 0.625$
So equivalent decimal of binary fractional is 0.625

Step 3: Add result of step 1 and 2.
=> $6 + 0.625 = 6.625$

Functions

Libraries

Libraries offer functions to be used:

- `<stdio.h>` required for `scanf, printf`
- Math library `<math.h>` (compile with `lm` option - `gcc -lm ....`)

User Defined Functions

Function prototype

A function prototype includes only the function return type, function name, and data types of parameters (optionally with names)

Example

`double area(double)`
The function return type: `double`
The function name: `area`
The parameter of the function: `double`

Good practice

Put function prototypes at top of program before `main` function, to inform compiler of functions that program may use and their return types and parameter types.

Example

```
#include ...
double area(double);

int main(void) {
        ...
        area(..);
}

double area(double x) {
        return x * x;
}
```

Without the prototypes, the compiler will generate error/warning messages.

If the function prototype is removed, the compiler assumes the default (implicit) return type of `int`. This can result in a conflict with the function header of the function which causes an error.

Scope

Scope rule

Local parameters and variables are only accessible in the function they are declared.

Example

```
int f(void) {
        int a;
}
int g(void) {
        return a;
}
```

This example does not work as the variable `a` is local to `f` and cannot be used in `g`, unless it is initialised separately.

Why do local parameters only exist during the execution of the function?

When a function is called,

- an activation record is created in the call stack
- memory is allocated for the local parameters & variable

  However, when it is done, the activation record is removed, and memory allocated for local parameters and variables is released.

  Automatic and static variables

  Automatic variables are variables which exist in memory only during the execution of the function they are local to. Static variables exist in the memory even after function is executed.

Pass-by-value

Pass-by-value means that the arguments from a caller are passed into the formal parameters for a function by its value - meaning any modification to the argument within the function does not affect the value stored in the argument variable.

Function with Pointer Parameters

With pass-by-value, using regular parameters, a function is unable to return more than one value or modify values of variables defined outside it.

Thus, to achieve this requirement, a function with pointer parameter is needed.

Swapping values

An example of this is a swapping function.

```
int main(void) {
        int a;
        int b;

        swap(&a, &b)
}

void swap(int *ptr1, int *ptr2) {
        int temp = *ptr1; // gets the value in ptr1 into temp
        *ptr1 = *ptr2; // changes the value in ptr1 to the value in ptr2
```

```
        *ptr2 = temp; // changes the value in pr2 to temp (value in ptr1)
}
```

Pointers

Pointer

A pointer refers to the memory address of a variable.

The pointer of a variable is referred to using the address operator `&`.

String formatting

%p is used to specify the addresses, which are printed out in hexadecimal format.

Example

`printf("a = %d", a)` prints the value of `a` in decimal.
`printf(&a = %p", a)` prints the address of `a`.

To store an address of another variable, use a pointer variable. The variable is then considered to be pointing to the variable with the memory address.

Example

```
int a;
int *a_ptr = a;
```

The **indirection operator** `*` allows access of the value of a pointer variable. In the example above, then `a` is synonymous with `*a_ptr`

Incrementing Pointer

Incrementing a pointer results in an incrementation of the address by the size of the data type that the pointer is referring to. For example, if a pointer for an `int` variable (which takes up 4 bytes) is incremented, the address it is pointed to is incremented by 4 bytes.

Example

```
int a;
int *b = &a; // if the memory address here is ffbff0a4

*b++; // it increments by 4 to ffbff0a8
```

Common sizes for data types

`int` : 4 bytes
`float` : 8 bytes
`char` : 1 byte
`double` : 8 bytes

Segmentation Fault

Do not use the indirection operator to access the value of a non-initialised pointer variable, as this will cause a segmentation fault. An example of this:

Example

```
int *n;
*n = 123;
```

Use Cases

- Pass addresses of two or more variables so function can pass back to caller new values for variables
- Pass address of first element of array to a function so function can access all elements in array.

Arrays

Arrays

Homogenous collection of data.

Declaration includes element type, array name, and size (maximum number of elements). Array elements occupy contiguous memory locations.

An array can be initialised at declaration with

- size declared as the same size of the array assigned to it `a[3] = {0, 1, 2}`
- size not declared `a[] = {0, 1, 2}`
- size declared as a bigger size `a[4] = {0, 1, 2}`

Incorrect initialisations/assignments

- when there are excess elements `a[1] = {1, 2}`
- when the array is declared, then initialised in a different statement. `a[1]; a = {0};`
  - this also means that an array cannot be reassigned to another array `a[2]; b[2] = a;`

Pointers

The array name refers to the address of the first element in the array.

Example

Given the array `int a[3];`, `a` is a memory address. `a` is equivalent to `&a[0]`.

Cloning an array

1. Write a loop assigning the element of each array to the other array consecutively
2. `memcpy` from `<string.h>` (not covered in module)

Array as a function parameter

For prototypes, it is fine to just specify the `[ ]` after the datatype to signify that it is an array.

Example

```
int function(int []); // (name can be added, not compulsory)
```

In the function header, the number within the square brackets is ignored, so the array size should be provided through another parameter `(int arr[], int size)`

Alternatively, as an array is a pointer, the alternative syntax is as such:

- prototype: `int function (int *)`
- header: `int function (int *array)`

Modification

As an array is passed in through pointers, whether intentional, the function **can** modify the content of the array passed into it.

Strings

String

A string is an array of characters, that is terminated by a null character `\0` (ASCII value of zero).
7
To convert an array of characters into a string, add a null character `'\0'` at the end of the array. String functions then can be used to manipulate these strings (`<string.h>`).

Declaration, Assignment, Initialisation

```
char str[];
```

When assigning characters to the array, the array must have the null character `\0` .

Initialising can be done in two ways:

- through initialising a character array `char str[] = {'o', 'k', '\0')`
- through assigning a string (no null character needed) `char str[] = "ok"`

Input/Output

Input:

- `fgets (str, size, stdin)` reads $size - 1$ char, or until newline
- `scanf ("%s", str)` reads until whitespace

Output:

- `puts(str)` terminates with newline
- `printf("%s\n", str)`
- `fgets` reads in newline character, so replacement of `\n` for `\0` may be necessary

String Functions

- `strlen(s)` Returns number of characters in s
- `strcmp(s1, s2)` Compares ASCII values of the corresponding characters in `s1, s2`
- `strncmp(s1, s2, n)` Compare first n characters of `s1, s2`
- `strcpy(s1, s2)` Copies string pointed to by `s2` into array pointed to by `s1` .
- `strncpy(s1, s2, n)` Copies first n characters pointed to by `s2` to `s1` .

Structures

Structures

Allows grouping of heterogeneous members (of different types).

Example

```
typedef struct {
        int x, y, z;
} box_t
```

A `typedef` is a definition of a type, not a declaration of a variable

- No memory is allocated to a type

Initialisation is similar to array initialisation - `.. = {}` with nested brackets if a structure contains a nested structure.

Member access using the `.` operator `box.x` (and for nested structures `box.colour.red`)

Assignments unlike arrays, the entire structure can be reassigned to another structure `box1 = box2`

Passing Structures into Function

The entire structure is copied into the parameter - members of the actual parameter is copied into the corresponding members of the formal parameter.

Thus, since a separate copy of it is made in the called function, the original structure variable is not modified by the function. Similar to normal variables, the pointer will have to be passed in instead.

Array of structures

Array name of an array of structures is still a pointer, thus, the function is still able to modify the array elements in this situation.

Arrow Operator

To access the structure variable from the pointer of the structure, the syntax is as follows

- `(*player_ptr).name)`
- `player_ptr->name`

Precedence of dot operator

Since the dot operator precedes `*`, `*player_ptr.name` results in `*(player_ptr.name)` instead of the intended `*(player_ptr).name`. Thus, `*player_ptr.name` should not be used.

Introduction To MIPS

Instruction Set Architecture

An abstraction on the interface between the hardware and the low-level software.

Machine code has instructions in binary, while assembly is still human readable (to a certain extent) as compared. Machine code is written in hexadecimal.

Walkthrough

The two major components in **Processor** and **Memory**. The processor performs computations while the memory stores the code and data. The code and data resides in the memory and transfers into the processor during execution.

To prevent frequent access of memory, **registers** are used to provide temporary storage for values in the processor.

Register

Provides temporary storage of values in the processor.
The arithmetic operations then can work directly on registers, which speed up the processes.

The load-store model

Load: Moving data from memory into a register
Store: Moving data from a register into memory

Memory instruction is then needed to move the data into registers and memory.

Instructions

- memory
- calculation
- control flow
  - repetition (loops)
  - selection (conditional)

General Purpose Registers

Registers have no **data type** unlike program variables - machine/assembly instruction assumes the data stored in the register is of the correct type.

Name	Register Number	Usage
`$zero`	0	Constant value 0
`$at`	*1*	***Reserved** for assembler*
`$v0-$v1`	2 - 3	Values for results and expression evaluation
`$a0-$a3`	4 - 7	Arguments
`$t0-$t7`	8 - 15	Temporaries
`$s0-$s7`	16 - 23	Program variables

Name	Register Number	Usage
`$t8-$t9`	24 - 25	More temporaries
`$k0-$k1`	*26 - 27*	***Reserved** for operating system*
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

Assembly Language

Each line of assembly code contains at most one instruction.

`#` Anything tagged with the hex-sign to end-of-line will be ignored by the assembler (comments)

General Instruction Syntax

```
OPERATION DESTINATION, SOURCE, SOURCE
```

Naturally, most of the MIPS arithmetic/logic operations have **three** operands: 2 sources and 1 destination. MIPS arithmetic operations are mainly register-to-register.

Addition

Example

In C, `a = b + c` is converted into the MIPS assembly code `add $s0, $s1, $s2`.

Variable mapping is assumed (the values of `a`, `b`, `c` are loaded into registers `$s0, $s1, $s2` respectively.)

Subtraction

Example

In C, `a = b - c` is converted into the MIPS assembly code `sub $s0, $s1, $s2`

Complex Expressions

Example

In C, `a = b + c - d` is converted into the MIPS assembly code:

```
add $t0, $s1, $s2
sub $s0, $t0, $s3
```

Remark

A single MIPS instruction can handle at most **two** source operands. Thus, a complex statement needs to be broken into multiple MIPS instructions.

Use of temporary registers

To store intermediate results, use the temporary registers $t0 to $t7 for intermediate results.

Immediate Operands

Immediate values are numerical constants frequently used in operations.

`addi`

The `addi` instruction is used to add numerical constants. The syntax is then `addi OPERAND, OPERAND, IMMEDIATE_VALUE`. The constant can range from -2^{15} to $2^{15}-1$, using the 2s complement number system on a 16-bit number.

`subi` does not exist

Use `addi` with a negative constant instead.

Register Zero `$zero`

The assignment of `f = g` is equivalent to `add $s0, $s1, $zero` (after assumed variable mapping). There is a equivalent pseudo-instruction `move $s0, $s1`.

Logical Operations

Arithmetic operations view the content of a register as a single quantity. Logical operations view the content of a register as 32 raw bits instead - allowing for operation on individual bits/bytes within a word.

Logical Operation	C Operator	Java Operator	MIPS Instruction
Shift Left	`<<`	`<<`	`sll`
Shift Right	`>>`	`>>`, `>>>`	`srl`
Bitwise `AND`	`&`	`&`	`and, andi`
Bitwise `OR`	|	|	`or`, `ori`
Bitwise `NOR`	`~`	`~`	`nor`
Bitwise `XOR`	`^`	`^`	`xor`

** Truth tables for bitwise operations*

`a`	`b`	`a AND b`
0	0	0
0	1	0
1	0	0
1	1	1

`a`	`b`	`a OR b`
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Shifting

```
sll TEMPORARY_REGISTER, VARIABLE_REGISTER, CONSTANT
srl TEMPORARY_REGISTER, VARIABLE_REGISTER, CONSTANT
```

The operation shifts the value in the `variable_register`, `$s0` for example, by the values in the constant, and fills the emptied position with zeroes, into the `temporary_register`, `$t0` for example.

Bitwise operations

`AND` operation:

Can be used for masking operations (setting irrelevant parts of the register to 0)

- to ignore bits in certain positions, place 0 in the positions to be ignored.
- to only focus on certain positions, place 1 in the interested positions.

`OR` operation:

Forces certain bits to 1s.

`NOR` operation:

Can be used to implement the `NOT` operation to toggle bits.
`nor $t0, $t0, $zero`

> **Why is a `NOT` instruction not provided in the instruction set?**
>
> Keep instruction set small.

`XOR` operation:

This can also be used to implement the `NOT`.
`xor $t0, $t0, $t2` where `$t2` contains all 1s.

Large Constant

> **How to load 32-bit constant into register?**
>
> 1. Use `lui` (load upper immediate) to set upper 16-bit:
>   `lui $t0, 0xAAAA`

2. Use `ori` (or immediate) to set lower-order bits
   ```
   ori $t0, $t0, 0xF0F0
   ```

Memory Instructions

The memory can be viewed as a large single-dimension array of memory locations with an address.

The **address space** given a $k-$bit address is of size 2^k.

Address	Content
0x0000000	8 bits
0x0000001	8 bits

Every location/address contains one byte (8 bits).

A memory address can be used to access:

- a single byte (byte addressable)
- a single word (word addressable)

Word

Usually 2^n bytes.

The common unit of transfer between processor and memory, and usually coincides with register size, integer size, and instruction size in most architecture.

Word alignment refers to whether a word begins at a byte address that is a multiple of the number of bytes in a word.

Word alignment

Given that a word consists of $2^2 = 4$ bytes, a word at the memory address 0 is word-aligned, but words at $1, 2, 3$ are not word-aligned.

How do we check for word alignment?

Use an `AND` operation (to perform masking)

For this example, for a $4-$byte word, the `AND` operation used can be `(address & 0x3) == 0`.
Since `0x3` is equivalent to `100b`, if the last two bits have any bits 1 after the `AND` operation, it can be said that the memory address is not a multiple of 4.

Memory Instructions

MIPS is a load-store-register architecture

- which 32 registers, each 32-bit (4-byte) long
- each word contains 32 bits (4 bytes)
- memory addresses are 32-bit long

There are 32 registers (as seen in General Purpose Registers), as well as 2^{30} memory words. As MIPS uses byte addresses, consecutive words differ by 4.

MIPS

In MIPS, data **must be in registers to perform arithmetic**. Thus, data in the memory words must be transferred (using the relevant data transfer instructions) into the registers, before the arithmetic can be performed.

As MIPS is a load-store-register architecture, the main instructions are `load` and `store`.

Load Word

Used to transfer data from the memory into the registers in the processor.

Syntax: `lw REGISTER_2, OFFSET(REGISTER_2)`

Example

Given that `$s0` contains `0x8000`:
`lw $t0, 4($s0)`

- the memory address is retrieved `$s0 + 4 = 0x8000 + 4 = 0x8004`
- the memory word `Mem[8004]` is loaded into `$t0`.

Load Word

Used to transfer data from registers into the memory.
Syntax: `sw REGISTER_2, OFFSET(REGISTER_2)`

Example

Given that `$s0` contains `0x8000`:
`sw $t0, 4($s0)`

- the memory address is retrieved `$s0 + 4 = 0x8000 + 4 = 0x8004`
- the content of `$t0` is stored into the word at `Mem[8004]`

There are other variants of `lw, sw` for example - bytes (`lb, sb`). In this scenario, offset no longer needs to be a multiple of 4.

Unaligned word

MIPS disallows loading/storing unaligned word using `lw, sw`. However, pseudo-instructions `ulw` and `usw` are provided for this purpose.

Other memory instructions: `lh, sh` for halfword, `lwl, lwr, swl, swr` for loading word left/right, etc...

Decision Making & Loops

Decision Making

Decision making in high-level languages are generally symbolised by `if` and `goto` statements. While `goto` is discouraged in high-level languages, it is necessary in assembly.

Decision-making instructions are responsible for the **control flow** of the program, and changes the next instruction to be executed.

Decision making statements in MIPS are split into two types:

- Conditional (branch)
- Unconditional (jump)

Label

A label is an anchor in the assembly code to indicate point of interest (branch target).

Conditional

`beq` (branch if equal)

Syntax: `beq $r1, $r2, L1`
Goes to the statement labeled `L1` if the value in the register `$r1` equals the value in `$r2`.

C equivalent code

```
if (a == b) goto L1
```

`bne` (branch if not equal)

Syntax: `bne $r1, $r2, L1`
Goes to the statement labeled `L1` if the value in the register `$r1` does notequals the value in `$r2`.

C equivalent code

```
if (a != b) goto L1
```

Translating `if` statements can have multiple translations.

Example

Given the following C statement:

```
if (i == j) {
        f = g + h;
}
```

with the variable mappings: `$f -> $s0, $g -> $s1, $h -> $s2, $i -> $s3, j -> s4`

Using `beq`

```
      beq $s3, s4, L1
      j Exit
L1:   add $s0, $s1, $s2
Exit:
```

Using `bne`

```
      beq $s3, s4, Exit
      add $s0, $s1, $s2
Exit:
```

In this scenario, the `bne` translation is more efficient. This uses a common technique of inverting the condition for shorter code.

Example

Given the following C statement:

```
if (i == j) {
        f = g + h;
}
else {
        f = g - h;
}
```

with the variable mappings: `$f -> $s0, $g -> $s1, $h -> $s2, $i -> $s3, j -> s4`

A possible solution

```
      bne $s3, s4, Else
      add $s0, $s1, $s2
      j Exit
L1:   sub $s0, $s1, $s2
Exit:
```

Loops

Loops are used as a way to iterate through code.

Key concept

Any form of loop can be written in assembly with the help of conditional branches and jumps.

`while` -loop

```
while (j == k)
        i = i + 1;
```

can be rewritten with `goto` statements:

```
Loop: if (j ! = k)
                goto Exit;
          i = i+1;
          goto Loop;
Exit:
```

which can then be rewritten into the MIPS code with variable mapping `i -> $s3, j -> $s4, k -> $s5`

```
Loop: bne $s4, $s5, Exit
          addi $s3, $s3, 1
          j Loop
Exit:
```

Inequalities

There are no in-built `blt/bmt` operations to `branch-if-less-than`. Thus, use `slt` or `slti` (`set-less-than`) to get a value to `bne/beq` on.

Building a `blt` operation:

`blt $s1, $s2, L` can be formatted into:

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

This is an example of a pseudo-instruction (the assmbler translates the `blt` instruction to an equivalent set of instruction in MIPS).

Arrays and Loops

Note that in certain scenarios, use of pointers can produce more efficient code.

Why can pointers produce more efficient code?

When not using pointers, the `sll` instruction is required to get the address increment from the offset.

Example

Since the word is 4bytes, the operation $i \times 4$ needs to be done to obtain the memory address increment for the offset.
When pointers are used, all that has to be done is an `addi $t1, $t1, 4` where `$t1` holds the current address for the element.

Instruction Formats

Fixed-length

All MIPS instruction has a fixed length of 32 bits.

Formats

To reduce complexity of processor design, instruction encodings should be as regular as possible. This means that there should be minimal number of formats (as few variations of the combination of operands)

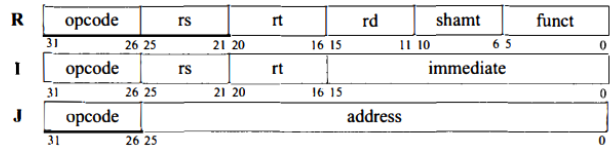
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

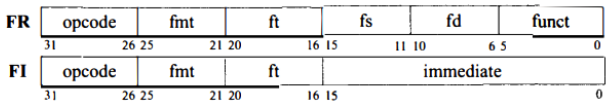
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVEDACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

The classification can be seen as follows:

Classification Name	Format	Example
R-format (register)	2 source registers, 1 destination register	`op $r1, $r2, $r3`
I-format (immediate)	1 destination register, 1 source register, 1 immediate value	`op $r1, $r2, Immd`
J-format (jump)	1 immediate value	`op Immd`

R-format

The fields can be defined (32-bit) as follows:

`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
specifies instruction when combined with funct set to 0	specify register for first operand	specify register for second operand	specify register to receive result	amount instruction will shift by (set to 0 for non-shift instructions)	specifies instruction when combined with opcode

Note that each field is an independent 5-bit/6-bit *unsigned integer*:

- 5 bits unsigned: 0 - 31
- 6 bits unsigned: 0 - 63

I-format

Immediate values

Immediates may be much larger than the 5-bit `shamt` field allows. Thus, a new instruction format has to be defined, but it can be partially consistent with R-format.

The compromise is as follows: if the instruction has an immediate value, it can only use up to 2 registers.

`opcode`	`rs`	`rt`	`immediate`
6 bits	5 bits	5 bits	16 bits
specifies instruction	specify source register operand	specify register for receiving result	`rd`, `shamt`, `funct` are merged to form a 16-bit field used for a constant value

32-bit constants

Since the maximum amount of data that can be represented by `immediate` is 16-bit wide, we cannot put 32-bit constants in the instructions. Hence, the reason why we can only use part of it, and have to handle the upper and lower bits separately.

Instruction Addresses

Instructions are stored in memory, and thus need addresses. As instructions are 32-bit long, they are also word-aligned.

Program Counter

A special register, keeping address of instructions being executed in the processor

Using the I-format, note that since the memory address is 32 bits, the `immediate` field is not enough to specify entire target address.

This might cause an issue when we are using branching.This is as the `bne` and `beq` uses immediate values as their operand.

Branches and its usage

Branches are used generally for decision-making (conditionals) or iterations.

Since loops are generally small, the change in PC will not be high.
Unconditional jumps are also done without using the branch instructions, but instead the jump instructions, which uses a different format.

Thus, to be able to store the addresses in the `immediate` field, the target address is stored as the relative target address to the PC, generated as the PC + 16-bit immediate field. The immediate field is a signed 2's complement integer.

This can then branch to $\pm 2^{15}$ bytes from the PC, enough to cover most loops.
In addition, since instructions are word-aligned, the `immediate` can be interpreted as a number of words, allowing for branching to 2^{15} words instead of bytes, which is equivalent to 2^{17} bytes from the PC.

The two scenarios are then as follows:

- if branch is not taken, $PC = PC + 4$ (go to next instruction)
- if branch is taken, $PC = (PC + 4) + (immediate \times 4)$
  - hardware design specifies to add `immediate` to `PC + 4` instead of `PC`

J-format

The format is as follows:

`opcode`	`target address`
6 bits	26 bits
Kept identical to R and I-format for consistency	Combined to make room for larger target addresses.

How do we specify the memory address, as the `target address` field is 26-bit, while the actual memory addresses are 32-bit?

Note that jumps will only jump to instruction addresses, which are word-aligned. This means that the last 2-bits will always be 00, meaning the address will always end with 00 for instruction addresses. Thus, we can now specify 28-bits of 32-bit address.

To get the next 4 bits, MIPS takes the 4 most significant bits from `PC + 4`, the next instruction after the jump instruction. Thus, the memory address is taken:

`PC + 4`	`target address` **from instruction**	**default 2 bit**
4 bits	26 bits	2 bits
Most significant bits of `PC`	Specified in instruction	Always `00`

Use of MSB of PC

With the use of the first 4-bits from MSB, note that jumping can only be done within the block.

Example

There cannot be a jump from `0x0....` to `0x1....` as the MSB dictates that the target address will have the first 4 bits be `0000` (from `0`).

Thus, there is a 256mb boundary:

$$2^8 = 256 \text{ mb}$$

Addressing Modes

Register addressing

Operands are registers

`add, sub, and, or, xor, nor, slt, sll, srl` instructions

Immediate addressing

Operand is a constant within the instruction itself

`addi, andi, ori, xori, slti` instructions

Base addressing

Operand is at memory location whose address is a sum of aregister and a constant in the instruction

`lw, sw` instructions

PC-relative addressing

Address is sum of PC and constant in the instruction

`beq, bne` instructions

Pseudo-direct addressing

26-bit of instruction concatenated with upper 4-bits of PC (+ LSB 2-bits set to `00b`)

`j` instruction

Note that

- branches use **PC-relative** addressing
- jumps use **pseudo-direct** addressing

Memory Instructions

The memory can be viewed as a large single-dimension array of memory locations with an address.

The **address space** given a $k-$bit address is of size 2^k.

Address	Content
0x0000000	8 bits
0x0000001	8 bits

Every location/address contains one byte (8 bits).

A memory address can be used to access:

- a single byte (byte addressable)
- a single word (word addressable)

Word

Usually 2^n bytes.

The common unit of transfer between processor and memory, and usually coincides with register size, integer size, and instruction size in most architecture.

Word alignment refers to whether a word begins at a byte address that is a multiple of the number of bytes in a word.

Word alignment

Given that a word consists of $2^2 = 4$ bytes, a word at the memory address 0 is word-aligned, but words at $1, 2, 3$ are not word-aligned.

How do we check for word alignment?

Use an `AND` operation (to perform masking)

For this example, for a $4-$byte word, the `AND` operation used can be `(address & 0x3) == 0`.
Since `0x3` is equivalent to `100b`, if the last two bits have any bits 1 after the `AND` operation, it can be said that the memory address is not a multiple of 4.

Memory Instructions

MIPS is a load-store-register architecture

- which 32 registers, each 32-bit (4-byte) long
- each word contains 32 bits (4 bytes)
- memory addresses are 32-bit long

There are 32 registers (as seen in General Purpose Registers), as well as 2^{30} memory words. As MIPS uses byte addresses, consecutive words differ by 4.

MIPS

In MIPS, data **must be in registers to perform arithmetic**. Thus, data in the memory words must be transferred (using the relevant data transfer instructions) into the registers, before the arithmetic can be performed.

As MIPS is a load-store-register architecture, the main instructions are `load` and `store`.

Load Word

Used to transfer data from the memory into the registers in the processor.

Syntax: `lw REGISTER_2, OFFSET(REGISTER_2)`

Example

Given that `$s0` contains `0x8000`:
`lw $t0, 4($s0)`

- the memory address is retrieved `$s0 + 4 = 0x8000 + 4 = 0x8004`
- the memory word `Mem[8004]` is loaded into `$t0`.

Load Word

Used to transfer data from registers into the memory.
Syntax: `sw REGISTER_2, OFFSET(REGISTER_2)`

Example

Given that `$s0` contains `0x8000`:
`sw $t0, 4($s0)`

- the memory address is retrieved `$s0 + 4 = 0x8000 + 4 = 0x8004`
- the content of `$t0` is stored into the word at `Mem[8004]`

There are other variants of `lw, sw` for example - bytes (`lb, sb`). In this scenario, offset no longer needs to be a multiple of 4.

Unaligned word

MIPS disallows loading/storing unaligned word using `lw, sw`. However, pseudo-instructions `ulw` and `usw` are provided for this purpose.

Other memory instructions: `lh, sh` for halfword, `lwl, lwr, swl, swr` for loading word left/right, etc...

Decision Making & Loops

Decision Making

Decision making in high-level languages are generally symbolised by `if` and `goto` statements. While `goto` is discouraged in high-level languages, it is necessary in assembly.

Decision-making instructions are responsible for the **control flow** of the program, and changes the next instruction to be executed.

Decision making statements in MIPS are split into two types:

- Conditional (branch)
- Unconditional (jump)

Label

A label is an anchor in the assembly code to indicate point of interest (branch target).

Conditional

`beq` (branch if equal)

Syntax: `beq $r1, $r2, L1`
Goes to the statement labeled `L1` if the value in the register `$r1` equals the value in `$r2`.

C equivalent code

```
if (a == b) goto L1
```

`bne` (branch if not equal)

Syntax: `bne $r1, $r2, L1`
Goes to the statement labeled `L1` if the value in the register `$r1` does notequals the value in `$r2`.

C equivalent code

```
if (a != b) goto L1
```

Translating `if` statements can have multiple translations.

Example

Given the following C statement:

```
if (i == j) {
        f = g + h;
}
```

with the variable mappings: `$f -> $s0, $g -> $s1, $h -> $s2, $i -> $s3, j -> s4`

Using `beq`

```
     beq $s3, s4, L1
     j Exit
L1:  add $s0, $s1, $s2
Exit:
```

Using `bne`

```
     beq $s3, s4, Exit
     add $s0, $s1, $s2
Exit:
```

In this scenario, the `bne` translation is more efficient. This uses a common technique of inverting the condition for shorter code.

Example

Given the following C statement:

```
if (i == j) {
        f = g + h;
}
else {
        f = g - h;
}
```

with the variable mappings: `$f -> $s0, $g -> $s1, $h -> $s2, $i -> $s3, j -> s4`

A possible solution

```
     bne $s3, s4, Else
     add $s0, $s1, $s2
     j Exit
L1:  sub $s0, $s1, $s2
Exit:
```

Loops

Loops are used as a way to iterate through code.

Key concept

Any form of loop can be written in assembly with the help of conditional branches and jumps.

`while` -loop

```
while (j == k)
        i = i + 1;
```

can be rewritten with `goto` statements:

```
Loop: if (j ! = k)
                goto Exit;
          i = i+1;
          goto Loop;
Exit:
```

which can then be rewritten into the MIPS code with variable mapping `i -> $s3, j -> $s4, k -> $s5`

```
Loop: bne $s4, $s5, Exit
          addi $s3, $s3, 1
          j Loop
Exit:
```

Inequalities

There are no in-built `blt/bmt` operations to `branch-if-less-than`. Thus, use `slt` or `slti` (`set-less-than`) to get a value to `bne/beq` on.

Building a `blt` operation:

`blt $s1, $s2, L` can be formatted into:

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

This is an example of a pseudo-instruction (the assmbler translates the `blt` instruction to an equivalent set of instruction in MIPS).

Arrays and Loops

Note that in certain scenarios, use of pointers can produce more efficient code.

Why can pointers produce more efficient code?

When not using pointers, the `sll` instruction is required to get the address increment from the offset.

Example

Since the word is 4bytes, the operation $i \times 4$ needs to be done to obtain the memory address increment for the offset.
When pointers are used, all that has to be done is an `addi $t1, $t1, 4` where `$t1` holds the current address for the element.

Instruction Formats

Fixed-length

All MIPS instruction has a fixed length of 32 bits.

Formats

To reduce complexity of processor design, instruction encodings should be as regular as possible. This means that there should be minimal number of formats (as few variations of the combination of operands)

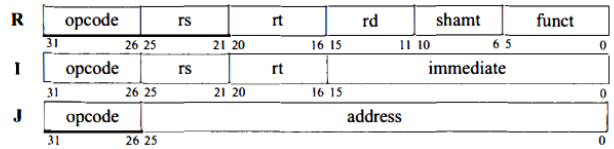
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

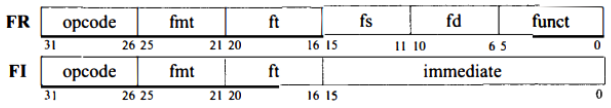
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVEDACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

The classification can be seen as follows:

Classification Name	Format	Example
R-format (register)	2 source registers, 1 destination register	`op $r1, $r2, $r3`
I-format (immediate)	1 destination register, 1 source register, 1 immediate value	`op $r1, $r2, Immd`
J-format (jump)	1 immediate value	`op Immd`

R-format

The fields can be defined (32-bit) as follows:

`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
specifies instruction when combined with funct set to 0	specify register for first operand	specify register for second operand	specify register to receive result	amount instruction will shift by (set to 0 for non-shift instructions)	specifies instruction when combined with opcode

Note that each field is an independent 5-bit/6-bit *unsigned integer*:

- 5 bits unsigned: 0 - 31
- 6 bits unsigned: 0 - 63

I-format

Immediate values

Immediates may be much larger than the 5-bit `shamt` field allows. Thus, a new instruction format has to be defined, but it can be partially consistent with R-format.

The compromise is as follows: if the instruction has an immediate value, it can only use up to 2 registers.

`opcode`	`rs`	`rt`	`immediate`
6 bits	5 bits	5 bits	16 bits
specifies instruction	specify source register operand	specify register for receiving result	`rd`, `shamt`, `funct` are merged to form a 16-bit field used for a constant value

32-bit constants

Since the maximum amount of data that can be represented by `immediate` is 16-bit wide, we cannot put 32-bit constants in the instructions. Hence, the reason why we can only use part of it, and have to handle the upper and lower bits separately.

Instruction Addresses

Instructions are stored in memory, and thus need addresses. As instructions are 32-bit long, they are also word-aligned.

Program Counter

A special register, keeping address of instructions being executed in the processor

Using the I-format, note that since the memory address is 32 bits, the `immediate` field is not enough to specify entire target address.

This might cause an issue when we are using branching.This is as the `bne` and `beq` uses immediate values as their operand.

Branches and its usage

Branches are used generally for decision-making (conditionals) or iterations.

Since loops are generally small, the change in PC will not be high.
Unconditional jumps are also done without using the branch instructions, but instead the jump instructions, which uses a different format.

Thus, to be able to store the addresses in the `immediate` field, the target address is stored as the relative target address to the PC, generated as the PC + 16-bit immediate field. The immediate field is a signed 2's complement integer.

This can then branch to $\pm 2^{15}$ bytes from the PC, enough to cover most loops.
In addition, since instructions are word-aligned, the `immediate` can be interpreted as a number of words, allowing for branching to 2^{15} words instead of bytes, which is equivalent to 2^{17} bytes from the PC.

The two scenarios are then as follows:

- if branch is not taken, $PC = PC + 4$ (go to next instruction)
- if branch is taken, $PC = (PC + 4) + (immediate \times 4)$
  - hardware design specifies to add `immediate` to `PC + 4` instead of `PC`

J-format

The format is as follows:

`opcode`	`target address`
6 bits	26 bits
Kept identical to R and I-format for consistency	Combined to make room for larger target addresses.

How do we specify the memory address, as the `target address` field is 26-bit, while the actual memory addresses are 32-bit?

Note that jumps will only jump to instruction addresses, which are word-aligned. This means that the last 2-bits will always be 00, meaning the address will always end with 00 for instruction addresses. Thus, we can now specify 28-bits of 32-bit address.

To get the next 4 bits, MIPS takes the 4 most significant bits from `PC + 4`, the next instruction after the jump instruction. Thus, the memory address is taken:

`PC + 4`	`target address` **from instruction**	**default 2 bit**
4 bits	26 bits	2 bits
Most significant bits of `PC`	Specified in instruction	Always `00`

Use of MSB of PC

With the use of the first 4-bits from MSB, note that jumping can only be done within the block.

Example

There cannot be a jump from `0x0....` to `0x1....` as the MSB dictates that the target address will have the first 4 bits be `0000` (from `0`).

Thus, there is a 256mb boundary:

$$2^8 = 256 \text{ mb}$$

Addressing Modes

Register addressing

Operands are registers

`add, sub, and, or, xor, nor, slt, sll, srl` instructions

Immediate addressing

Operand is a constant within the instruction itself

`addi, andi, ori, xori, slti` instructions

Base addressing

Operand is at memory location whose address is a sum of aregister and a constant in the instruction

`lw, sw` instructions

PC-relative addressing

Address is sum of PC and constant in the instruction

`beq, bne` instructions

Pseudo-direct addressing

26-bit of instruction concatenated with upper 4-bits of PC (+ LSB 2-bits set to `00b`)

`j` instruction

Note that

- branches use **PC-relative** addressing
- jumps use **pseudo-direct** addressing

Datapath

There are two major components for a processor.

Datapath

A collection of components that process data, and performs arithmetic, logical and memory operations.

Control

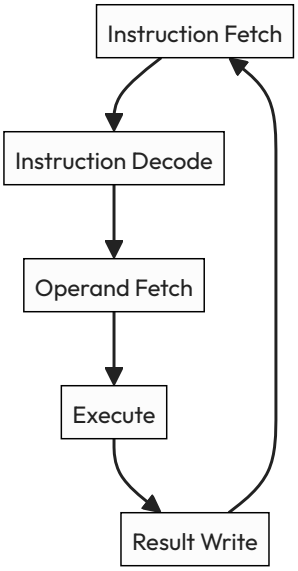
Tells the datapath, memory and I/O devices what to do according to program instructions.

MIPS Processor

Simplest possible implementation:

- Arithmetic and logical operations
  - `add`, `sub`
  - `and`, `or`
  - `addi`
  - `slt`
- Data transfer instructions
  - `lw`, `sw`
- Branches
  - `beq`, `bne`

Instruction Execution Cycle



1. Instruction Fetch
   This gets the instruction from memory. The address of this memory instruction is in the PC register.
2. Instruction Decode
   This decodes the encoded instruction from the memory instruction

3. Operand Fetch
   This retrieves the operand needed for operation.
4. Execute
   This performs the required operation.
5. Result Write (Store)
   Store result of the performed operation.

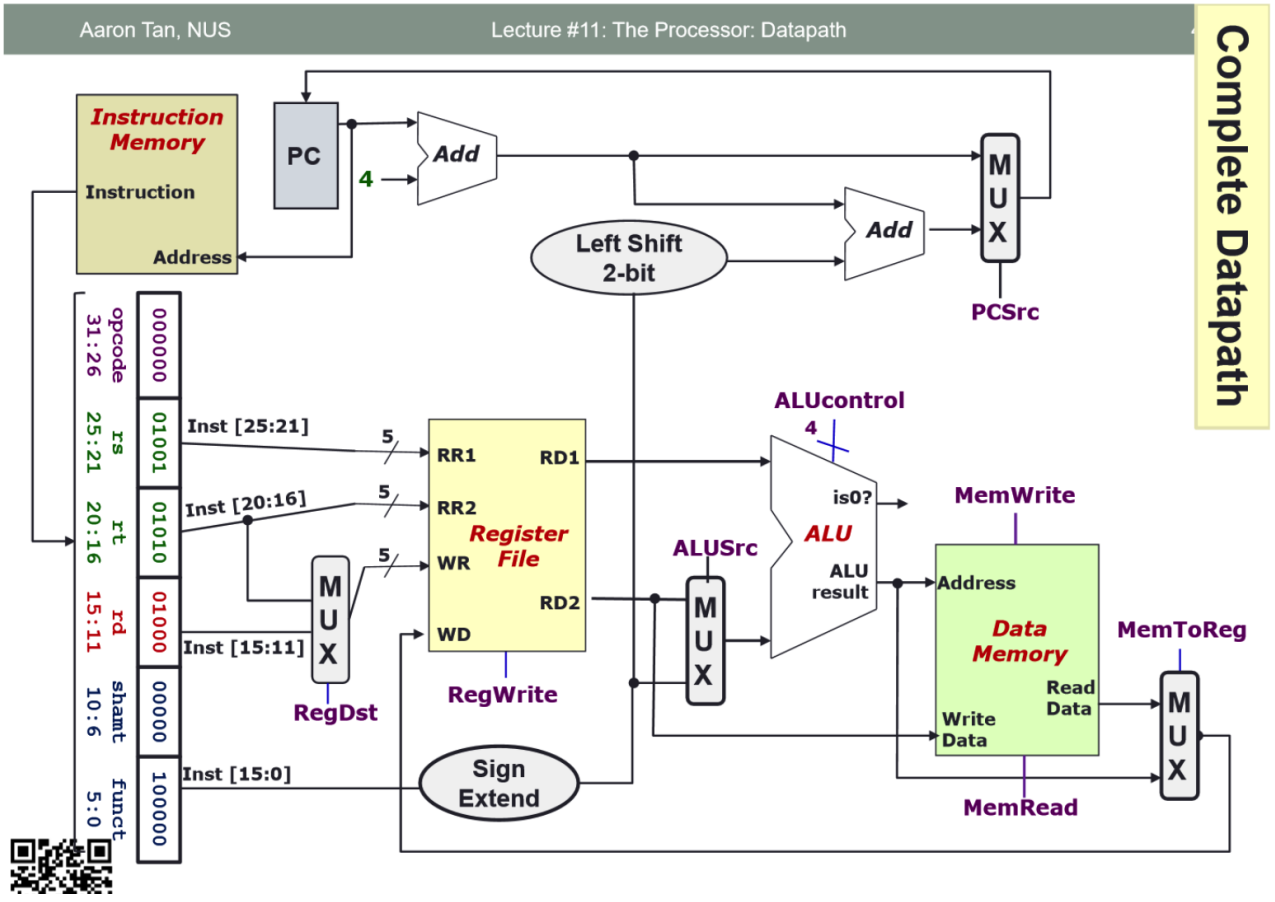
MIPS Instruction Execution

For MIPS, some design changes can be seen. Notably:

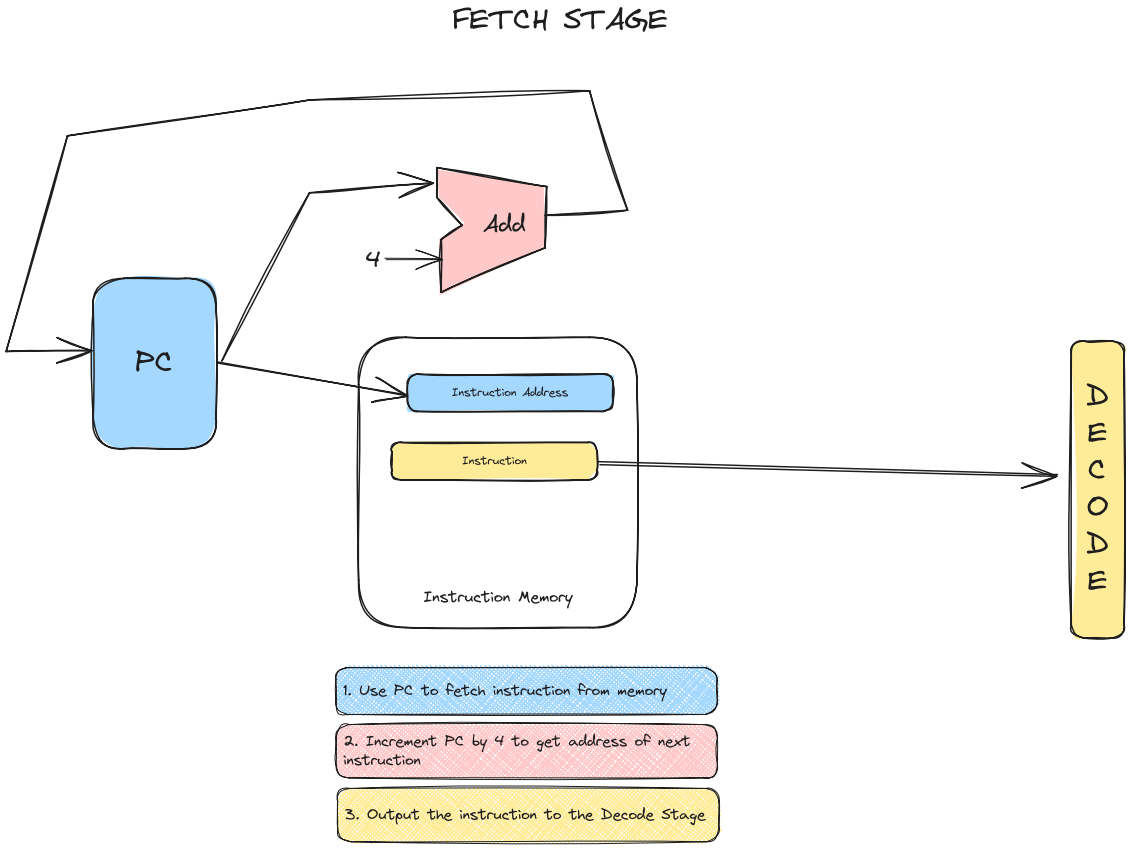
1. Merging decode and operand fetch
2. Splitting execute into ALU and memory access.

The execution flow is then as follows:

1. Fetch
2. Decode + Operand Fetch
3. ALU
4. Memory access
5. Result Write



Fetch Stage



Instruction memory

The instruction memory acts as a storage element for the instructions. The instruction memory is a sequential circuit that stores information through an internal state. The clock signal is assumed and not shown.

Effectively, this **supplies instruction, given the address**.

Adder

Combinational logic to implement the addition of two numbers.

Takes in two 32-bit numbers A, B and outputs the sum $A + B$.

Clocking

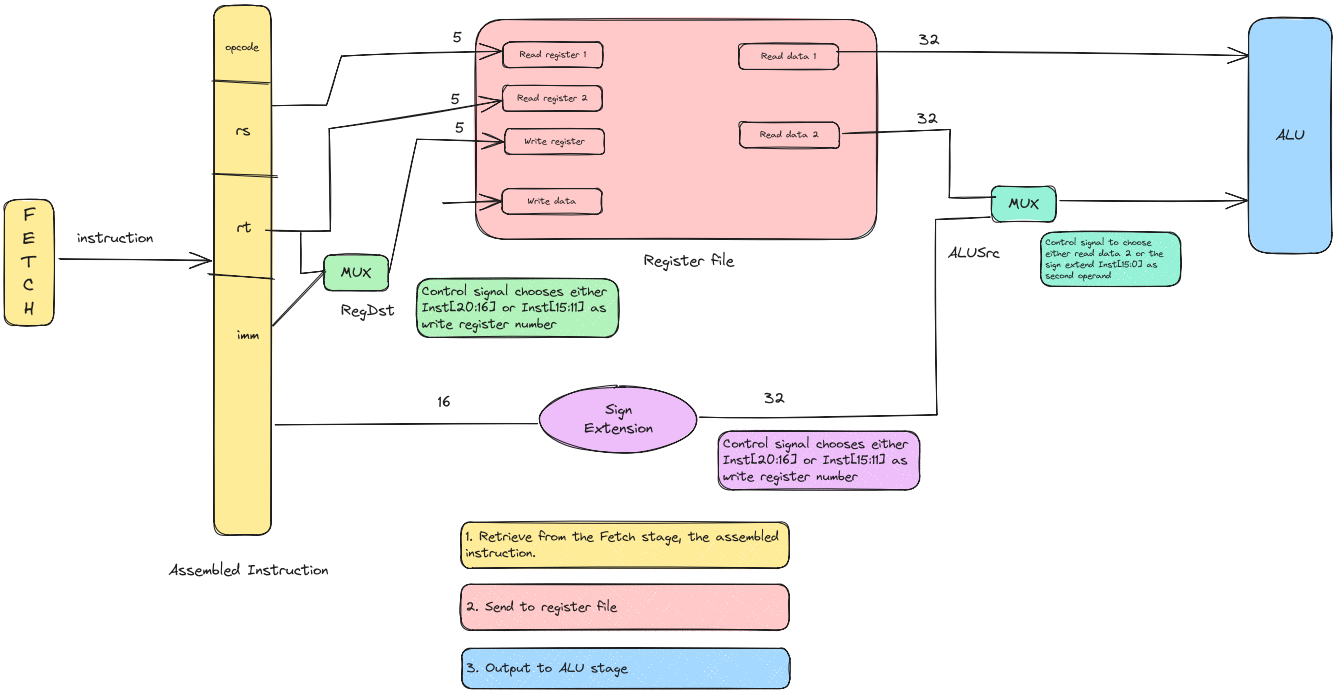
Allows the reading and updating of PC at the same time.

The PC is read during the first half of the clock period, and then is updated with PC + 4 at the next rising clock edge.

```
function FETCH() {
        inst = IM(PC); // read instruction at address from PC
        PC = Add(PC, 4); // update PC to PC+4
        return inst; // instruction is passed into decode stage
}
```

Decode Stage

DECODE STAGE



Register file

A collection of 32 registers (32-bit wide) which can be read and written to by specifying register number.

Per instruction, it can

- read at most two registers
- write at most one register

`RegWrite`

This is used as a control signal to indicate whether there is a writing of register.
`1` = Write, `0` = No Write

Multiplexer

A multiplexer is a device that selects between several analog or digital input signals and forwards the selected input to a single output line.

In this context, it selects one input from multiple input lines, has a control with m bits where n^{2m}, and selects the i^{th} input line if control = i.

`RegDst`

This control signal chooses whether to use the destination register is `Inst[20:16]` or `Inst[15:11]`.

This allows the use of `Inst[15:11]` as the write register (for R-format) or `Inst[20:16]` as the write register (for 2 registers).

`ALUSrc`

Control signal chooses whether to read data 2 (from the register) or read the sign extended `Inst[15:0]` as the second operand.

This allows the use of reading the data from the second register (for R-format) or `Inst[15:0]` when there is an immediate field.

`0`: `R-format`
`1`: `I-format` except for branch

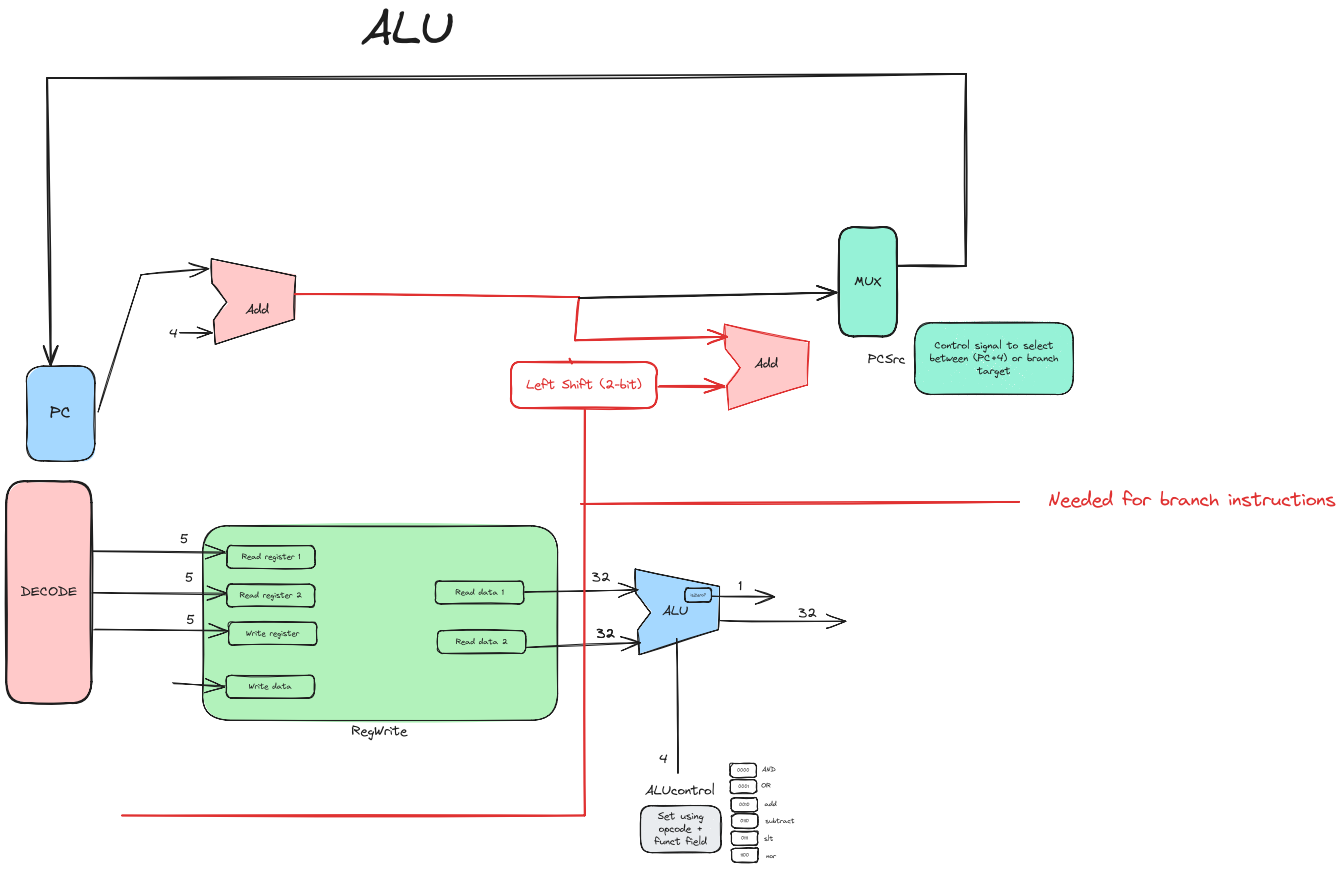
```
function DECODE(inst) {
        // 1. Read register
        RR1 = inst[21:25]; // $rs
        RR2 = inst[16:20]; // $rs
        RD1, RD2 = RegRead(RR1, RR2); // read both registers
        // 2. Store Write register for later use
        WR = Mux(inst[16:20], // $rt
                inst[11:15], $rd
                RegDst) // control signal
        // 3. Choose output
        IMM = SignExtend(inst[0:15]);
        return [RD1, Mux(RD2, IMM, ALUSrc)];
        // returns the 1st register, as well as either the 2nd register, or the immediate value
}
```

ALU Stage

ALU

Arithmetic-Logic Unit (also known as execution)

Performs most of the "real work". Takes input from the Decode stage (Operation and Operands) and outputs the calculation result to memory.



The ALU handles the combinational logic to implement arithmetic and logical operations. It takes in two 32-bit numbers and uses a 4-bit control signal. It outputs a result as well as a 1-bit signal to indicate whether result is zero.

Handling branch instruction

Requires two calculations:

1. Branch outcome
2. Branch target address

The branch outcome can be found using the ALU - the `isZero?` signal is enough to handle the equal/not equal check.

The multiplexer `PCSrc` effectively chooses between the `PC+4` or the branch target based on the `isZero` result, as well as the `Branch` control signal.

`PCSrc`

`0`: `PC + 4` (next instruction (non-branch))
`1`: `PC + 4 + immediate x 4` (next instruction (branch))

```
function ALU(A, B, ALUcontrol) {
        case 0000: return [A & B, A & B == 0]; // AND
        case 0001: return [A | B, A | B == 0]; // OR
        case 0010: return [A + B, A + B == 0]; // ADD
        case 0110: return [A - B, A - B == 0]; // SUB
        case 0111: return [A < B, A < B == 0]; // SLT
        case 1100: return [~(A | B), ~(A | B) == 0]; // NOR
        // returns the value of the operation, as well as ifZero
}
```

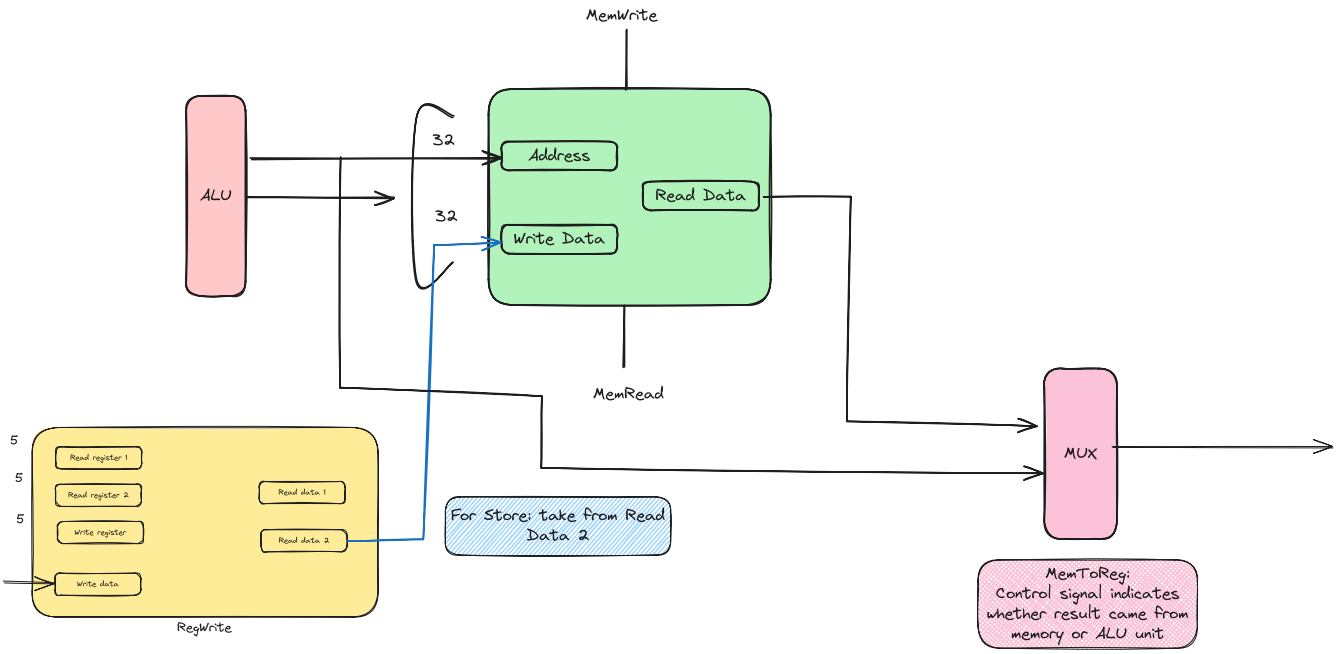
Memory

Memory access stage

For this stage, only the `load` and `store` instructions are needed to perform the operations.

Other operations are going to be passed through for the Register Write stage.

It takes an input from the ALU and outputs to the next stage (Register Write).



There are three cases of instructions:

Load
For this, the address of the value to be loaded is passed through into the ALU result.

Store

For this, `read data 2` will be directed into the `Write Data`, and the address of the value is passed through into the ALU result.

Non-memory instruction

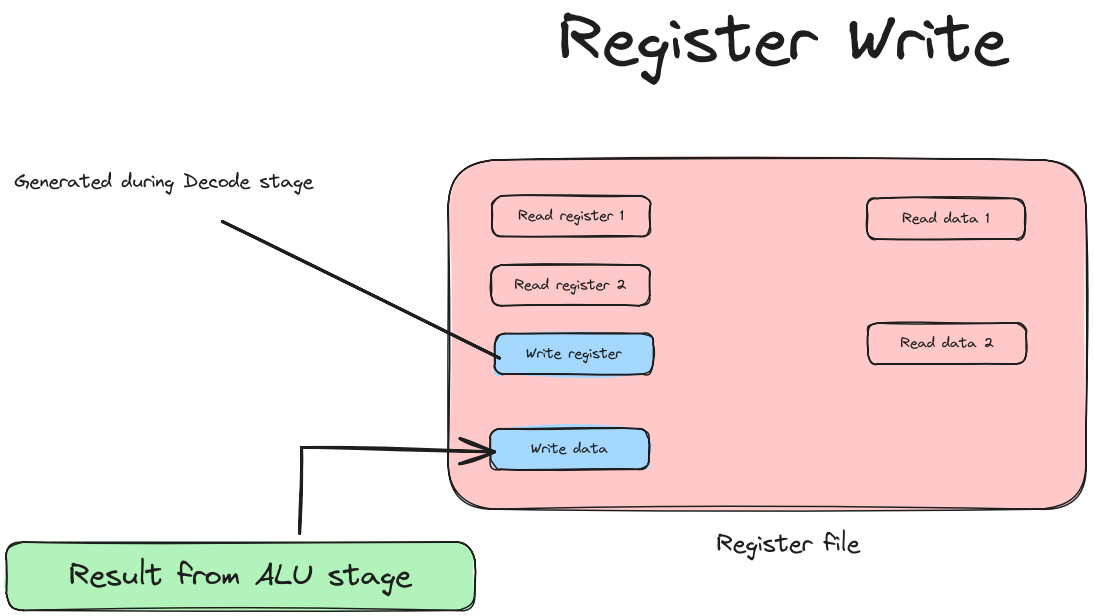
The ALU result and the value from `Read Data` goes through a multiplexer, where `MemToReg` will indicate result came from the ALU unit.

```
function MEM(alu_res, data) {
        mem_res = DataMem(alu_res, data, MemWrite, MemRead);
        return Mux(alu_res, mem_res, MemToReg);
        // based on MemToReg, gets the value of the Read 2 register
        // or the ALU rsult
}
```

Register Write

Register Write

Most instructions write the result of some computation into a register - this stage is responsible for that. Takes in input from the memory (where it is the computation result, either from memory or ALU)



MemToReg

`0` : no write
`1` : write

```
function WRITEBACK(data) {
        RegWrite(data, WR, RegWrite)
}
```

Datapath (Summary)

```
function FETCH();
function DECODE(inst);
function ALU(A, B, ALUcontrol);
function MEM(alu_res, data);
function WRITEBACK(data);

function DATAPATH() {
        inst, RR1, RR2, RD1, RD2, WR, IMM, A, B, alu_res, mem_res, data;
        inst = FETCH();
        A,B = decode(inst);
        alu_res = ALU(A, B, ALUcontrol);
        data = MEM(alu_res, RD2, MemWrite, MemRead);
        WRITEBACK(data, WR, RegWrite);
}

function FETCH() {
        inst = IM(PC); // read instruction at address from PC
        PC = Add(PC, 4); // update PC
        return inst;
}

function DECODE(inst) {
        RR1 = inst[21:25]; // $rs
        RR2 = inst[16:20]; // $rd/rt
        RD1, RD2 = RegRead(RR1, RR2); // read from registers

        WR = Mux(
                inst[16:20] // $rt
                inst[11:15] // $rd
                RegDst); // control signal: 0 for $rt, 1 for $rd

        IMM = SignExtend(inst[0:15]);
        return [RD1,
                Mux(RD2,
                IMM,
                ALUSrc)]; // control signal: 0 for $rt, 1 for IMM
}

function ALU(A, B, ALUcontrol) {
        case 0000: return [A & B, A & B  == 0]; // AND
        case 0001: return [A | B, A | B  == 0]; // OR
        case 0010: return [A + B, A + B  == 0]; // ADD
        case 0110: return [A - B, A - B  == 0]; // SUB
        case 0111: return [A < B, A < B  == 0]; // SLT
        case 1100: return [~ (A | B), ~ (A | B)  == 0]; // NOR
}

function MEM(alu_res, data) {
        mem_res = DataMem(
                alu_res,
                data,
                MemWrite, // 1 if write, 0 if no write
                MemRead // 1 if read, 0 if not
        );
}

function WRITEBACK(data) {
        RegWrite(data,
                WR,
                RegWrite); // 1 if write, 0 if no write
}
```

Control Signals

Signal	Stage	Purpose	Value
`RegDst`	Decode	Select destination register number	`0`: `$rt` `1`: `$rd`
`RegWrite`	Decode/Writeback	Enable write of register	`0`: no write `1`: write
`ALUSrc`	ALU	Select 2nd operand for ALU	`0`: `$rt` `1`: `IMM`

`ALUcontrol`	ALU	Select operation to be performed	`0000`: `AND`
			`0001`: `OR`
			`0010`: `ADD`
			`0110`: `SUB`
			`0111`: `SLT`
			`1100`: `NOR`

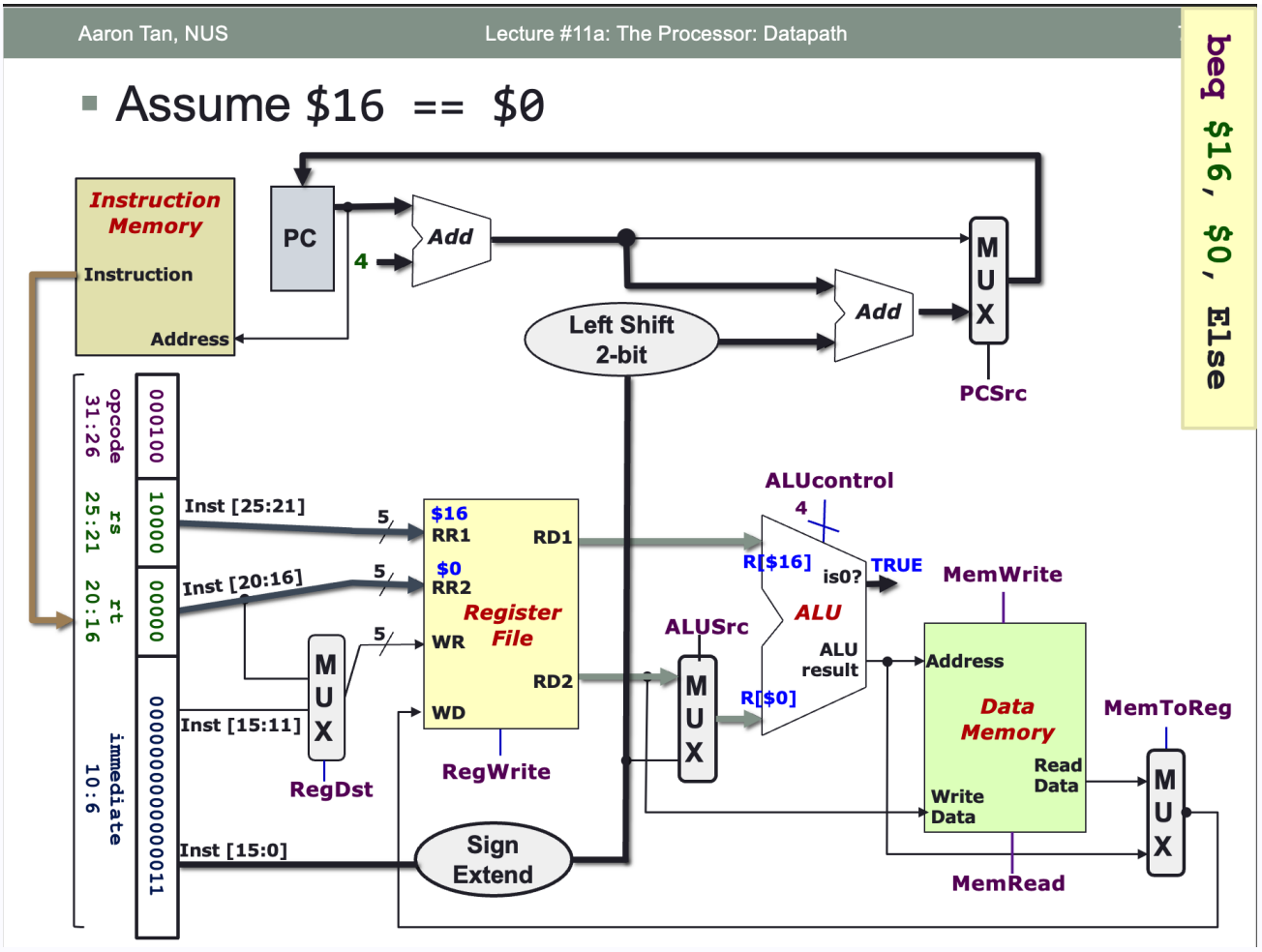
`MemRead`	Memory	Enable reading of data memory	`0`: no read `1`: write
`RegWrite`	Decode/Writeback	Enable write of register	`0`: no write `1`: write
`MemToReg`	Writeback	Select result to be written back	`0`: ALU result `1`: Memory data
`PCSrc`	Memory/Writeback	Select next `$PC` value	`0`: `$PC + 4` `1`: `$(PC + 4) + IMM`

Stages

Stage	Action	Input	Output
`FETCH`	Gets the instruction from the instruction address. Adds 4 to the PC.		Instruction
`DECODE`	Reads the opcode to determine instruction type and field lengths. Reads data from all necessary registers.	Instruction	Operand 1 and 2 for ALU
`ALU`	Computes results	Operands 1 and 2	Result, as well as `isZero`
`MEMORY`	Reads calculated address from ALU stage and data to be stored if any Reads/writes from memory	Calculated address and result from `ALU`	Data read from memory or result from `ALU`
`WRITEBACK`	Read result from memory stage to `WriteData` and `WR` Writes result	Result from `MEMORY`	

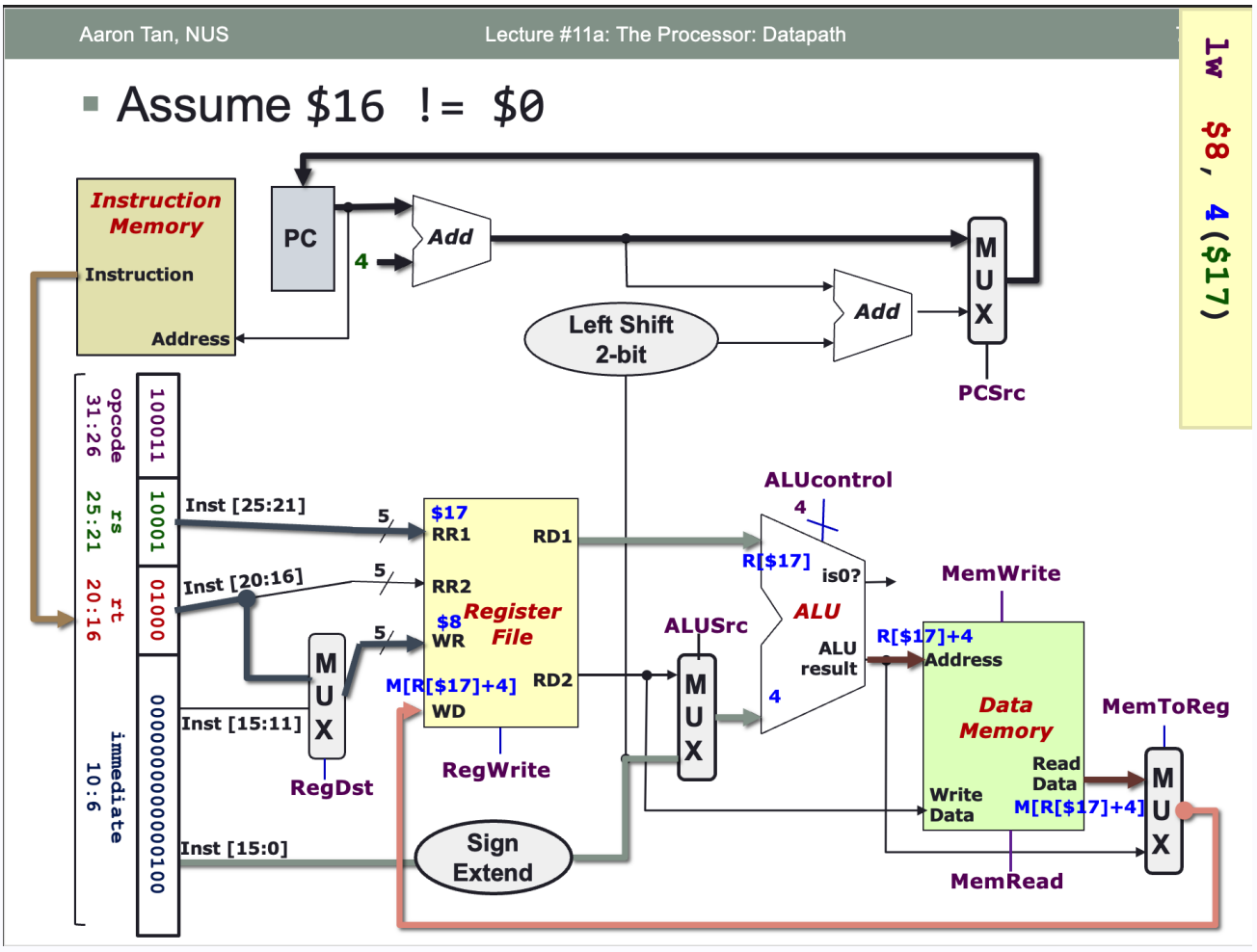
Examples

Branch instruction `beq`

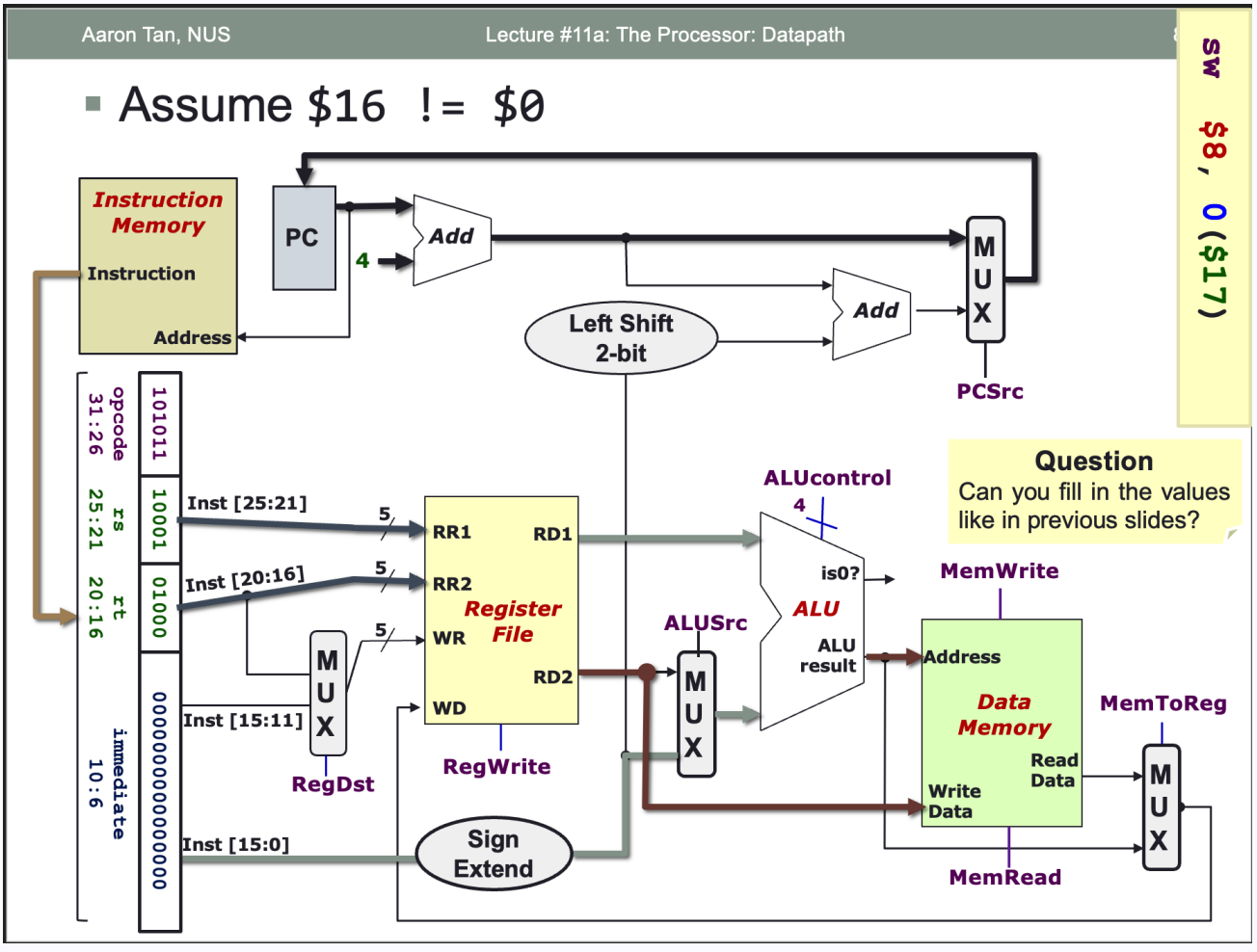

Aaron Tan, NUS
Lecture #11a: The Processor: Datapath
beq $16, $0, Else
Assume $16 == $0
Instruction Memory
Instruction
Address
PC
4
Add
Add
Left Shift 2-bit
M U X
PCSrc
opcode 31:26
rs 25:21
rt 20:16
immediate 15:0
000100
10000
00000
0000000000000011
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
5
5
5
$16
RR1
$0
RR2
WR
WD
RD1
RD2
Register File
RegDst
RegWrite
Sign Extend
ALUcontrol
4
R[$16]
is0?
TRUE
ALU
ALU result
R[$0]
ALUSrc
MemWrite
Address
Data Memory
Write Data
Read Data
MemRead
MemToReg

f

I-Format `lw`

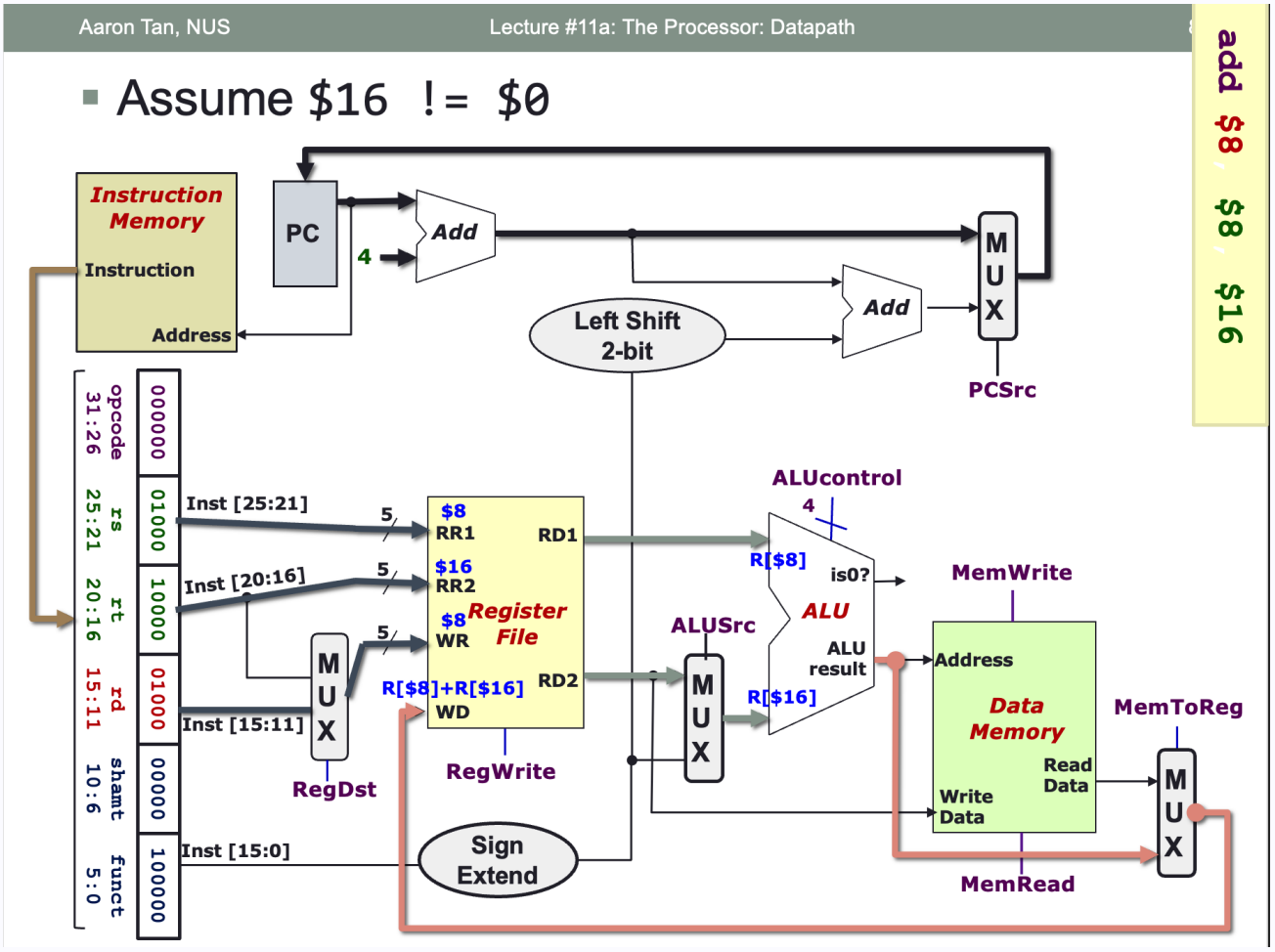


I-Format sw



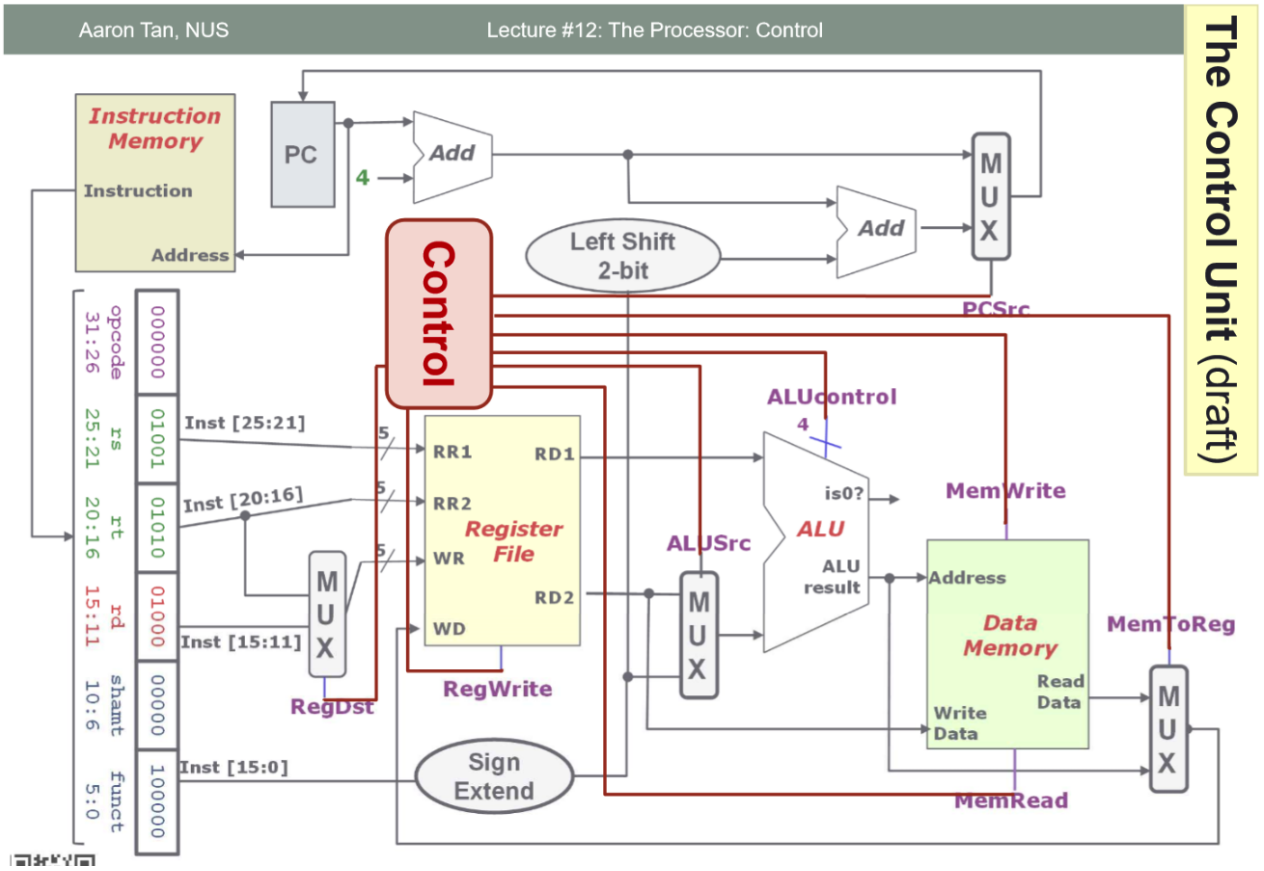
made by zaidan sani.

R-format


Aaron Tan, NUS
Lecture #11a: The Processor: Datapath
add $8, $8, $16
Assume $16 != $0
Instruction Memory
Instruction
Address
PC
4
Add
Add
Left Shift 2-bit
M U X
PCSrc
ALUcontrol
4
opcode 31:26
rs 25:21
rt 20:16
rd 15:11
shamt 10:6
funct 5:0
000000
01000
10000
01000
00000
100000
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
5
$8 RR1
$16 RR2
$8 WR
Register File
RD1
RD2
R[$8]+R[$16] WD
RegWrite
RegDst
ALUSrc
R[$8]
R[$16]
is0?
ALU
ALU result
MemWrite
Address
Data Memory
Read Data
Write Data
MemRead
MemToReg
Sign Extend

Control

Control signals are generated based on the instruction to be executed. Thus, a combinational circuit to generate signals based on opcode and function codes.



Control Signals

Signal	Stage	Purpose	Value
`RegDst`	Decode	Select destination register number	`0` : `$rt` (`Inst[20:16]`) `1` : `$rd` (`Inst[15:11]`)
`RegWrite`	Decode/Writeback	Enable write of register	`0` : no write `1` : write
`ALUSrc`	ALU	Select 2nd operand for ALU	`0` : `Operand2 = Register RD 2` `1` : `Operand2 = SignExt(Inst[15:0)`

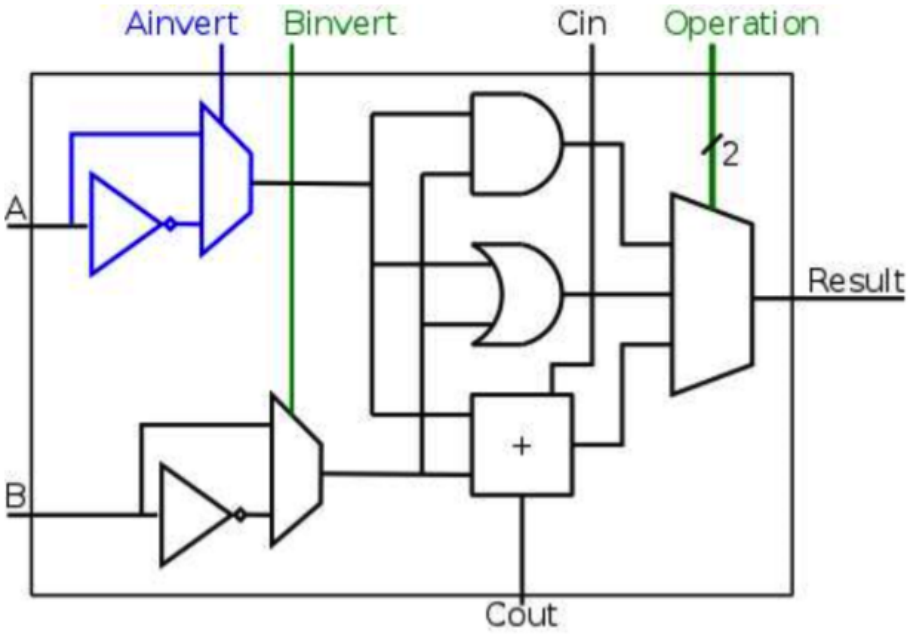
`ALUcontrol`	ALU	Select operation to be performed	`0000` : `AND`
			`0001` : `OR`
			`0010` : `ADD`
			`0110` : `SUB`
			`0111` : `SLT`
			`1100` : `NOR`

`MemRead`	Memory	Enable reading of data memory	`0` : no read `1` : write
`MemWrite`	Decode/Writeback	Enable write of register	`0` : no write `1` : write
`MemToReg`	Writeback	Select result to be written back	`0` : ALU result `1` : Memory data
`PCSrc`	Memory/Writeback	Select next `$PC` value	`0` : `$PC + 4` `1` : `$(PC + 4) + IMM`

made by zaidan sani.

Control	Signal	R-format	lw	sw	beq
0	RegDst	1	0	0	0
1	ALUSrc	0	1	1	0
2	MemToReg	0	1	0	0
3	RegWrite	1	1	0	0
4	MemRead	0	1	0	0
5	MemWrite	0	0	1	0
6	Branch	0	0	0	1
7	ALUop1	1	0	0	0
8	ALUop0	0	0	0	1

Control Bits



Ainvert
1
A
0

Binvert
1
B
0

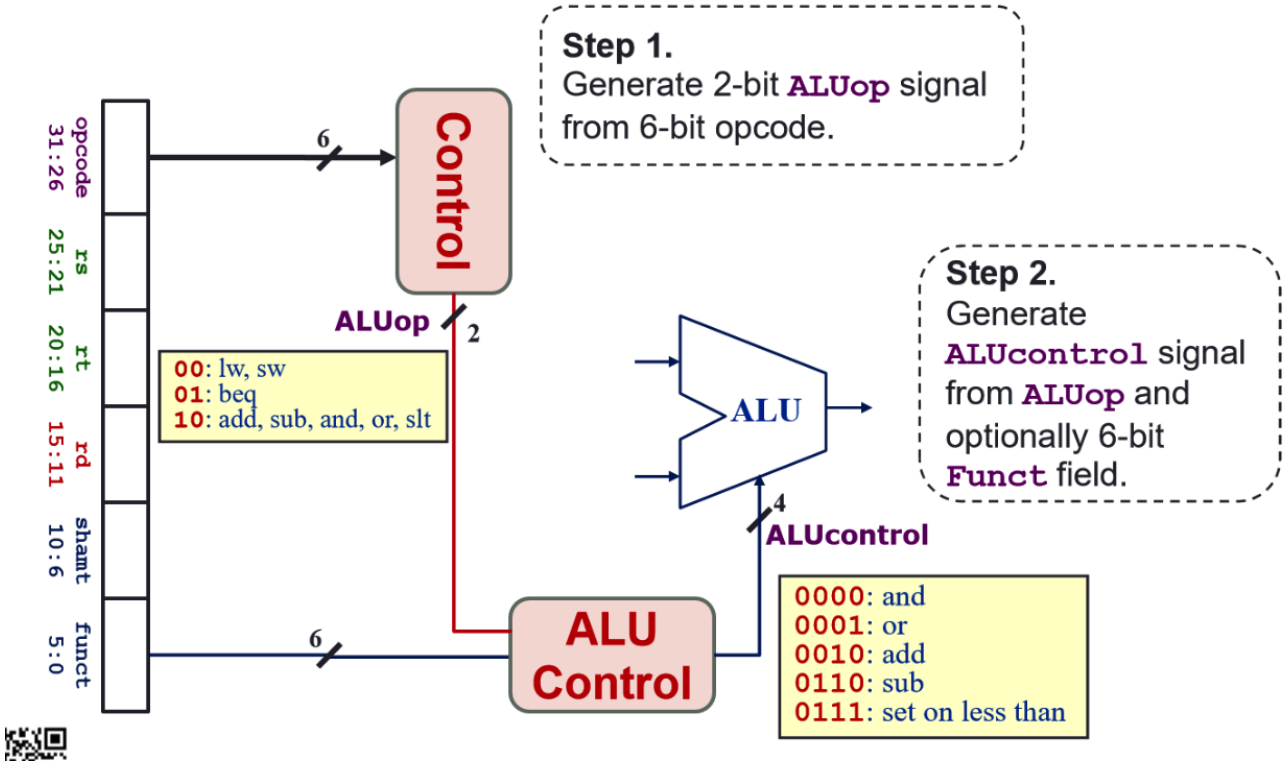
Operation

Ainvert	Binvert	Operation	Function
0	0	00	AND
0	0	01	OR
0	0	10	ADD
0	1	10	SUB

made by zaidan sani.

Ainvert	Binvert	Operation	Function
0	1	11	SLT
1	1	00	NOR

Two-Level Implementation



Opcode	ALUop	Instruction Operation	funct	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	sub	0110
R-type	10	add	100000	add	0010
R-type	10	sub	100010	sub	0110
R-type	10	and	100100	and	0000
R-type	10	or	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

5. Design of ALU Control Unit (1/2)

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

ALUcontrol3 = 0

ALUcontrol2 = ?

ALUop0 + ALUop1· F1

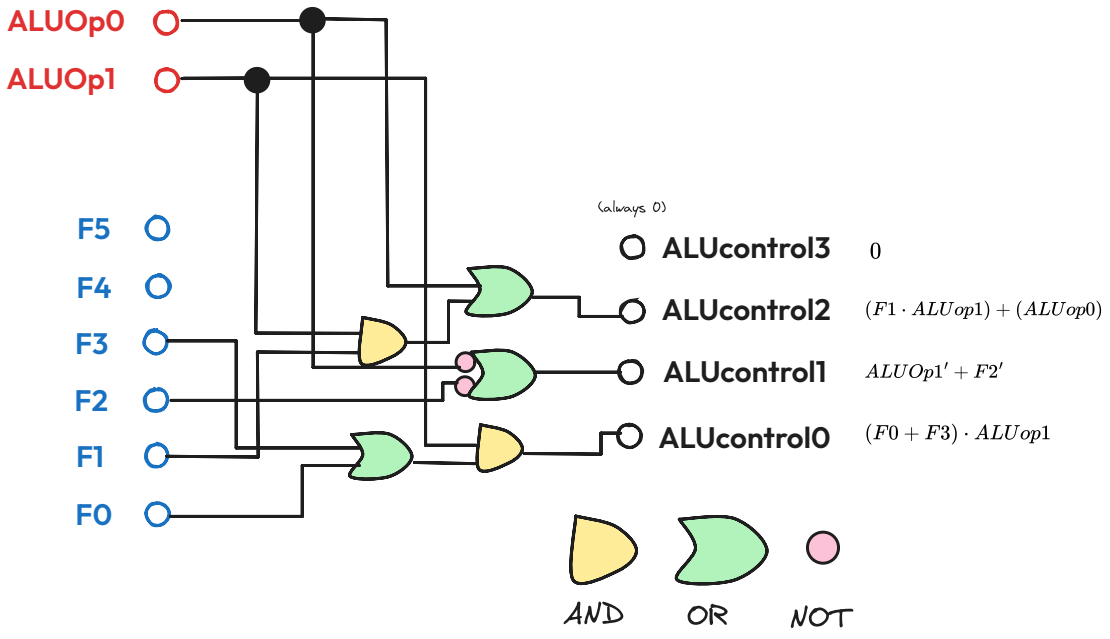
	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0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

https://app.sli.do/event/5P8F43kQW2Vg2Y8TTRVzHu

X

The `X` bits mean that the bits are not used to determine the ALU control signal.

```
ALUcontrol3 = 0
ALUcontrol2 = (F1 AND ALUop1) OR (ALUop0)
ALUcontrol1 = !ALUop1 OR !F2
ALUcontrol0 = (F0 OR F3) AND ALUop1
```



Deriving `ALUcontrol`

```
ALUcontrol3 = 0
```

For all the operations, `ALUcontrol3` = 0;

```
ALUcontrol2 = (ALUop1 AND F1) OR ALUop0)
```

`ALUcontrol2` is `1` for `sub`, `slt`, `beq`.

For `sub` and `slt`, the differentiating factor is `ALUop1 = 1` and `F1 = 1`
For `beq`, it is when `ALUop0 = 1`.

```
ALUcontrol1 = !ALUop1 AND !F2
```

`ALUcontrol1` is `1` for `add`, `sub`, `slt`, and all non `Rformat`.

When it is `R format`, so `ALUop1 = 1`. The differentiating factor is that `F2 = 0` for these instructions and `1` otherwise.

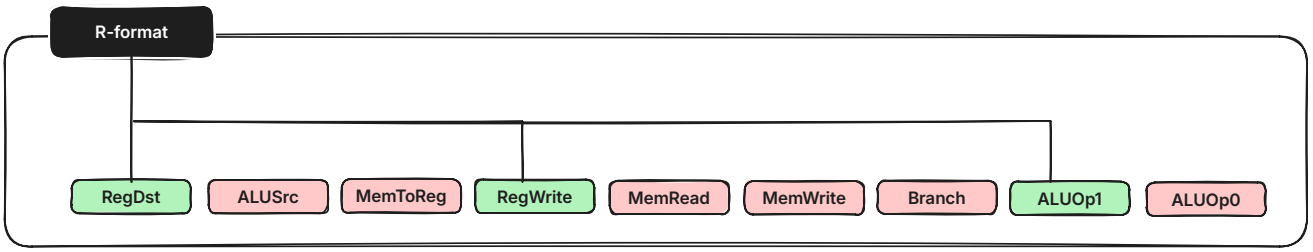
```
ALUcontrol0 = ALUop1 AND (F3 OR F0)
```

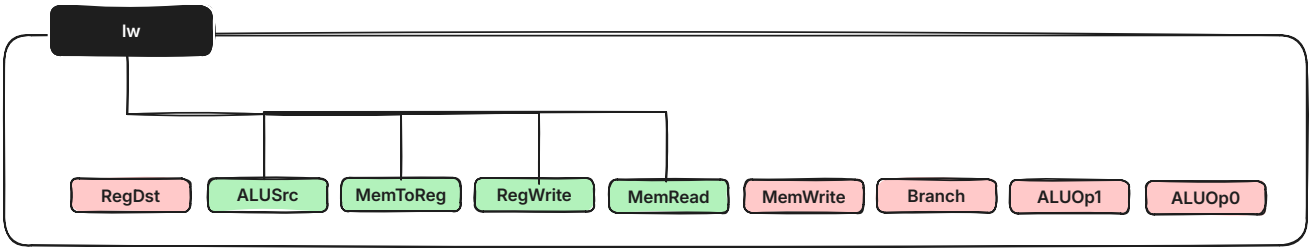
`ALUcontrol1` is `1` for `or`, `slt`.

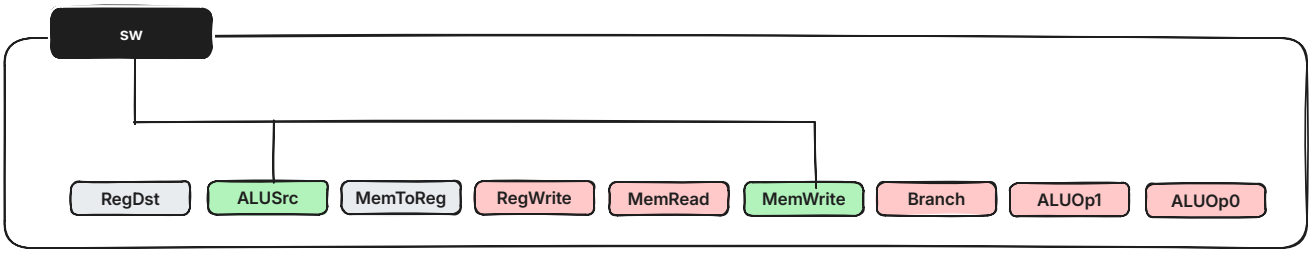
All the instructions are `R format`, so `ALUop1 = 1`.
`slt` and `or` do not share any common bits.

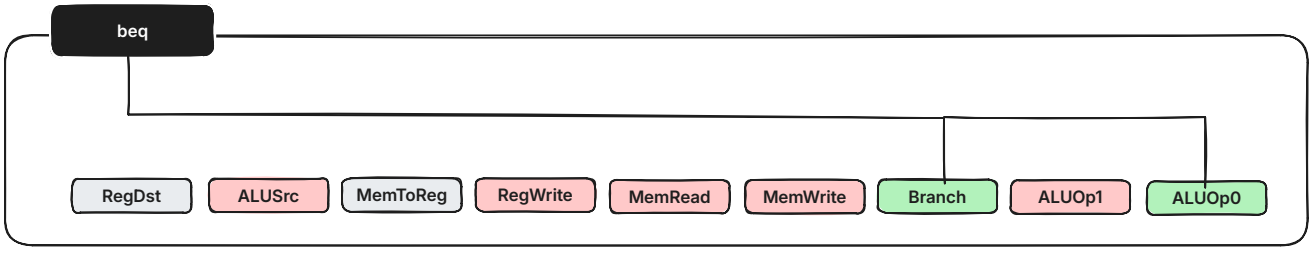
Thus, for `slt`, it is the only instruction with `F3 = 1`, so that can indicate the `slt`.
For `or`, it is the only instruction with `F0 = 1`, so that can indicate `or`.

Combinational Circuit Implementation

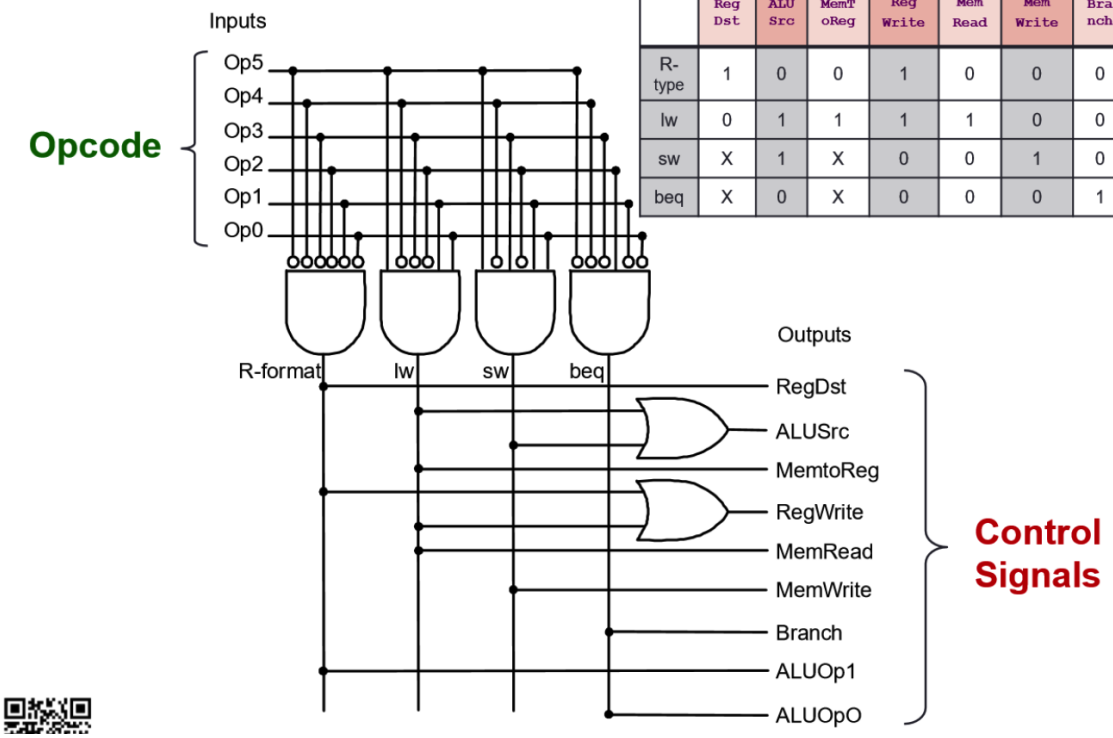


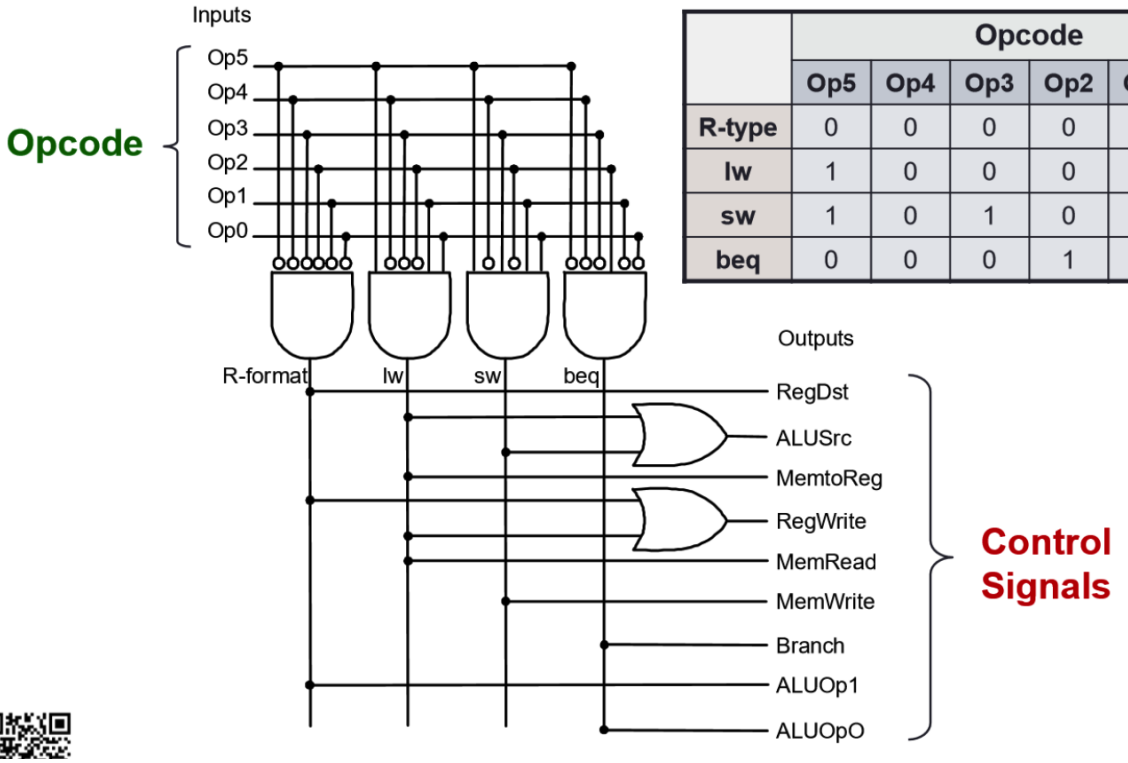






	Reg Dst	ALU Src	MemT oReg	Reg Write	Mem Read	Mem Write	Bra nch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1





	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

> The circuit connections indicate that the instructions set the particular control signal.

The `0` above the `OR` gates at the input refer to a collapsed `NOT` gate.

> R-format
>
> `000000` has collapsed `NOT` gates everywhere.

Instruction Execution

Instruction execution refers to the process of:

1. Reading contents of register/memory
2. Perform computation through computational logic
3. Write results to one or more storage elements

All of this should be performed within a clock period.

> A storage element should not be read as it is being written.

> Speed of single cycle implementation
>
> Using a single cycle implementation, all instructions will take as much time as the slowest one, resulting in a long cycle time for each implementation. This is as the instructions cannot overlap in execution.

Multicycle implementation

Break instructions into execution steps -

1. fetch
2. decode + read

3. ALU
4. memory read/write
5. register/write

By allocating this execution steps one clock cycle instead, one cycle is shorter, and the frequency is much higher.

> **Variable number of clock cycles are needed to complete different instructions.**

Pipelining

Breaks instructions into execution steps one per clock cycle.
Allows different instructions to be in different execution steps simultaneously.

MIPS Instruction Set

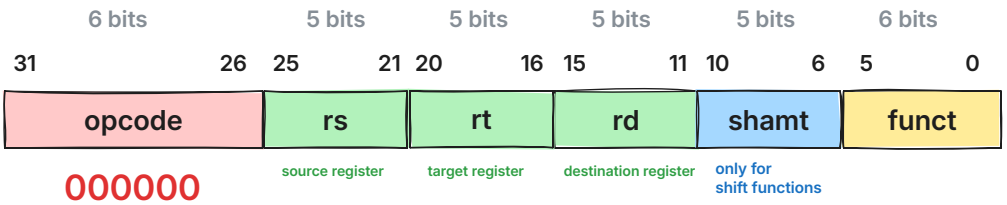
Registers

0. `$zero`
1. `$at`
2. `$v0`
3. `$v1`
4. `$a0`
5. `$a1`
6. `$a2`
7. `$a3`
8. `$t0`
9. `$t1`
10. `$t2
11. `$t3`
12. `$t4`
13. `$t5`
14. `$t6`
15. `$t7`
16. `$s0`
17. `$s1`
18. `$s2`
19. `$s3`
20. `$s4`
21. `$s5`
22. `$s6`
23. `$s7`
24. `$t8`
25. `$t9`
26. `$k0
27. `$k1`
28. `$gp`
29. `$sp`
30. `$fp
31. `$ra`

R-format

Opcode = `0x0` (000000_2)

R-format

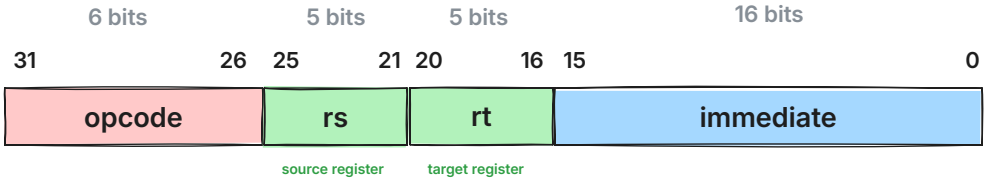


if first 2 hex digits of instruction is < 0x04,
instruction is of R-format

Mnemonic	Operation	Funct	(binary)
add	R[rd] = R[rs] + R[rt]	20	10 0000
addu	R[rd] = R[rs] + R[rt]	21	10 0001
and	R[rd] = R[rs] & R[rt]	24	10 0100
jr	PC = R[rs]	08	00 1000
nor	R[rd] = ~(R[rs] OR R[rt])	27	10 0111
or	R[rd] = R[rs] OR R[rt]	25	10 0101
slt	R[rd] = (R[rs] < R[rt]) ? 1 : 0	2a	10 1010
sltu	R[rd] = (R[rs] < R[rt]) ? 1 : 0	2b	10 1011
sll	R[rd] = R[rt] << shamt	0	00 0000
slt	R[rd] = R[rt] >> shamt	2	00 0010
sub	R[rd] = R[rs] – R[rt]	22	10 0010
subu	R[rd] = R[rs] – R[rt]	23	10 0011

I-format

I-format



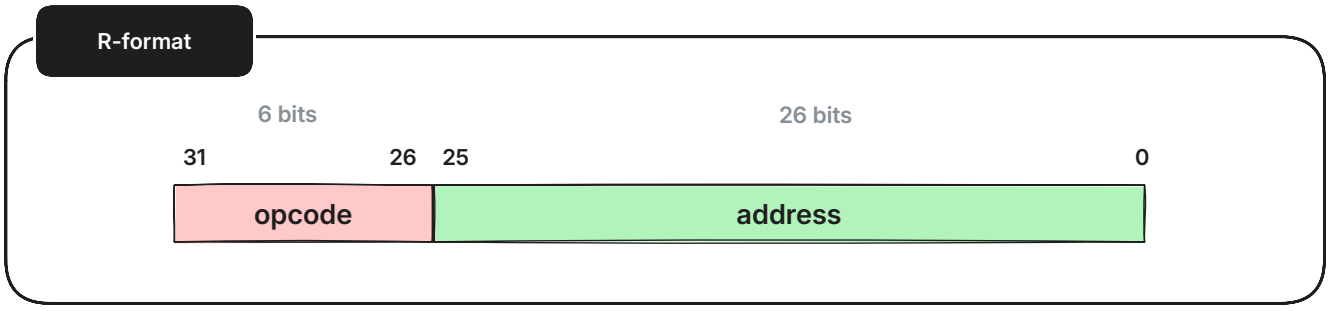
if first 2 hex digits of instruction is > 0x04,
instruction is of I-format

Mnemonic	Operation	Opcode	(binary)
addi	R[rt] = R[rs] + SignExtImm	0x08	00 1000
addiu	R[rt] = R[rs] + SignExtImm	0x09	00 1001
andi	R[rt] = R[rs] & ZeroExtImm	0x0c	00 1100
beq	if (R[rs] == R[rt]) PC = PC + 4 + BranchAddr	0x04	00 0100
bne	if (R[rs] != R[rt]) PC = PC + 4 + BranchAddr	0x05	00 0101
lui	R[rt] = {imm, 16'b0}	0x0f	00 1111

made by zaidan sani.

Mnemonic	Operation	Opcode	(binary)
lw	R[rt] = M[R[rs] + SignExtImm]	0x23	10 0011
ori	R[rt] = R[rs] or ZeroExtImm	0x0d	00 1101
slti	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	0x0a	00 1010
sltiu	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	0x0b	00 1011
sw	M[R[rs] + SignExtImm] = R[rt]	0x2b	10 1011
sb	M[R[rs] + SignExtImm](7:0) = R[rt](7:0)	0x28	10 1000

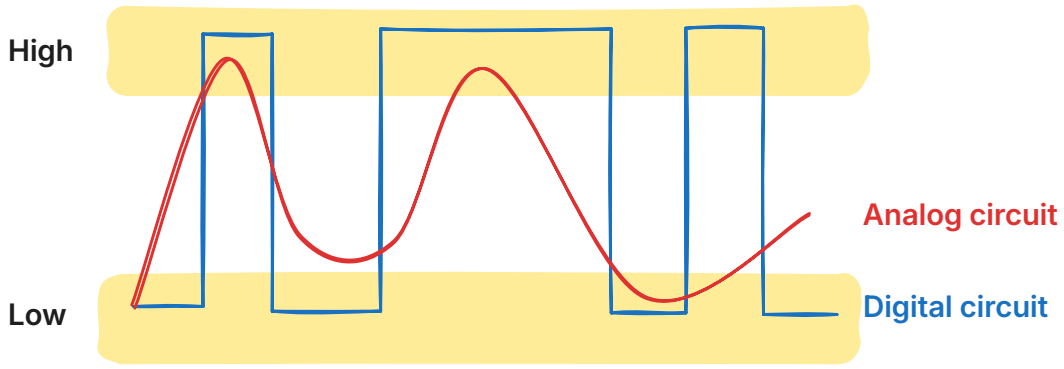
J-format



Mnemonic	Operation	Opcode	(binary)
j	PC = JumpAddr	0x02	00 0010

Boolean Algebra

Digital Circuits



Advantages of digital circuits

- More reliable
- Specified accuracy
- Abstraction can be applied using simple mathematical model
- Ease design, analysis and simplification of digital circuit

Combinational

No memory, output depends solely on the input

- Gates
- Decoders, multiplexers
- Adders, multipliers

Sequential

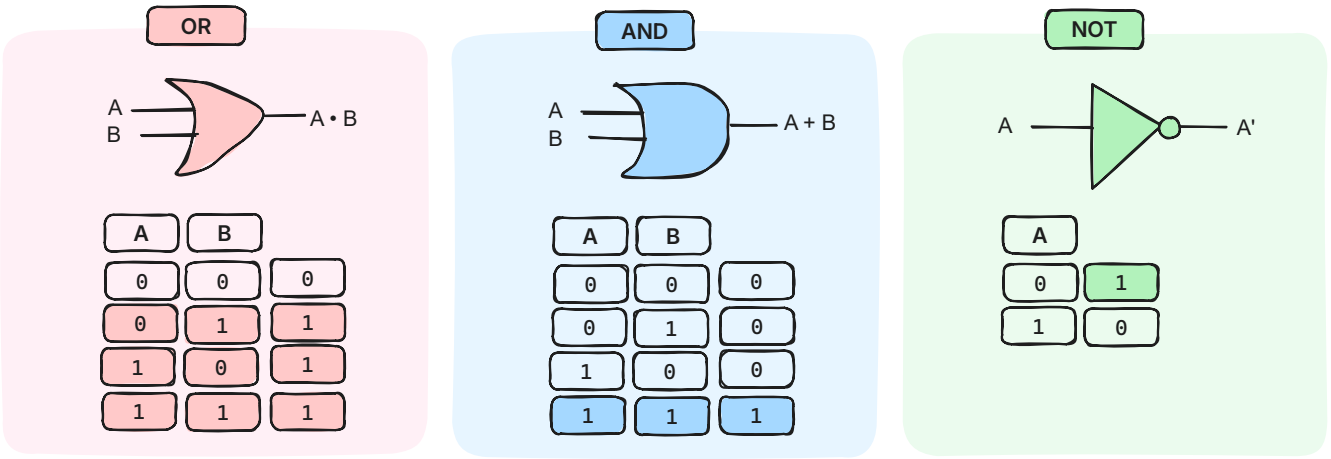
With memory, output depends on both input and current state

- Counters, registers, memories

Boolean Values

`true` : 1
`false` : 0

Connectives



Conjunction AND : $A \cdot B, A \wedge B$
Disjunction OR : $A + B, A \vee B$
Negation NOT : $A', \bar{A}, \neg A$

Precedence of Operators

1. Parenthesis
2. NOT
3. AND
4. OR

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

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

Distributive laws

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

Duality

made by zaidan sani.

If the `AND/OR` operations and identity elements `0/1` in a Boolean equation are interchanged, the equation remains valid.

- $(x + y + z)' = x' \cdot y' \cdot z'$ is valid $\implies (x \cdot y \cdot z)' = x' + y' + z'$ is valid

Idempotency

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

One element/zero element

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

Involution

$$(X')' = X$$

Absorption

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

Absorption

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

DeMorgan's

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

Consensus

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

Complement

Obtained by interchanging 1 with 0 in the function's output values

Standard Forms

1. Sum-Of-Products (SOP)
2. Product-of-Sums (POS)

Literal

Boolean variable on its own

- x, x'

Product term

Single literal or logical product `AND` of several literals

$x, x \cdot y$

Sum-of-products (SOP) expression

A product term or a logical sum `OR` of several product terms

$x, (x \cdot y) + (x' \cdot y')$

Sum term

Single literal or a logical sum `OR` of several literals

$x, x + y$

Product-of-sums (POS) expression

A sum term or a logical product `AND` of several sum terms

$x, (x + y) \cdot (x' + y')$

Minterms and Maxterms

Minterm

A product term that contains n literals from all the variables.

for x, y, minterms: $x' \cdot y', x' \cdot y, x \cdot y', x \cdot y$

Maxterm

A sum term that contains n literals from all the variables.

for x, y, minterms: $x' + y', x' + y, x + y', x + y$

For n variables, there is up to 2^n maxterms/minterms.

x	y	Minterms	Maxterms	Notation
0	0	$x' \cdot y'$	$x + y$	m/M0
0	1	$x' \cdot y$	$x + y'$	m/M1
1	0	$x \cdot y'$	$x' + y$	m/M2
1	1	$x \cdot y$	$x' + y'$	m/M3

Each minterm is the complement of the corresponding maxterm and vice versa.

Canonical forms

- Sum-of-minterms = Canonical sum-of-products

- Product-of-maxterms = Canonical product-of-sums

Sum-of-minterms

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

given a function with the following truth table:

x	y	F	
0	0	1	m0
0	1	0	m1
1	1	1	m2
1	0	0	m3

the sum-of-minterms

$$
\begin{aligned}
F &= (x' \cdot y') + (x \cdot y) \\
&= m0 + m2 \\
&= \sum m(0, 2)
\end{aligned}
$$

Product-of-maxterms

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

given a function with the following truth table:

x	y	F	
0	0	1	M0
0	1	0	M1
1	1	1	M2
1	0	0	M3

the product-of-maxterms

$$
\begin{aligned}
F &= (x + y') \cdot (x' + y) \\
&= M1 \cdot M3 \\
&= \prod M(1, 3)
\end{aligned}
$$

Conversion of Standard Forms

The conversion is just the opposite of the current result.

For example, given the example:

given a function with the following truth table:

x	y	F	
0	0	1	m0
0	1	0	m1
1	1	1	m2
1	0	0	m3

the sum-of-minterms

$$
\begin{aligned}
F &= (x' \cdot y') + (x \cdot y) \\
&= m0 + m2 \\
&= \sum m(0, 2)
\end{aligned}
$$

We can then also, derive F':

$$
\begin{aligned}
F' &= m1 + m3 \\
F &= (m1 + m3)' \\
&= m1' \cdot m3' && \text{De Morgan's Law} \\
&= M1 \cdot M3 && mx' = Mx
\end{aligned}
$$

Similarly from the product-of-maxterms:

given a function with the following truth table:

x	y	F	
0	0	1	M0
0	1	0	M1
1	1	1	M2
1	0	0	M3

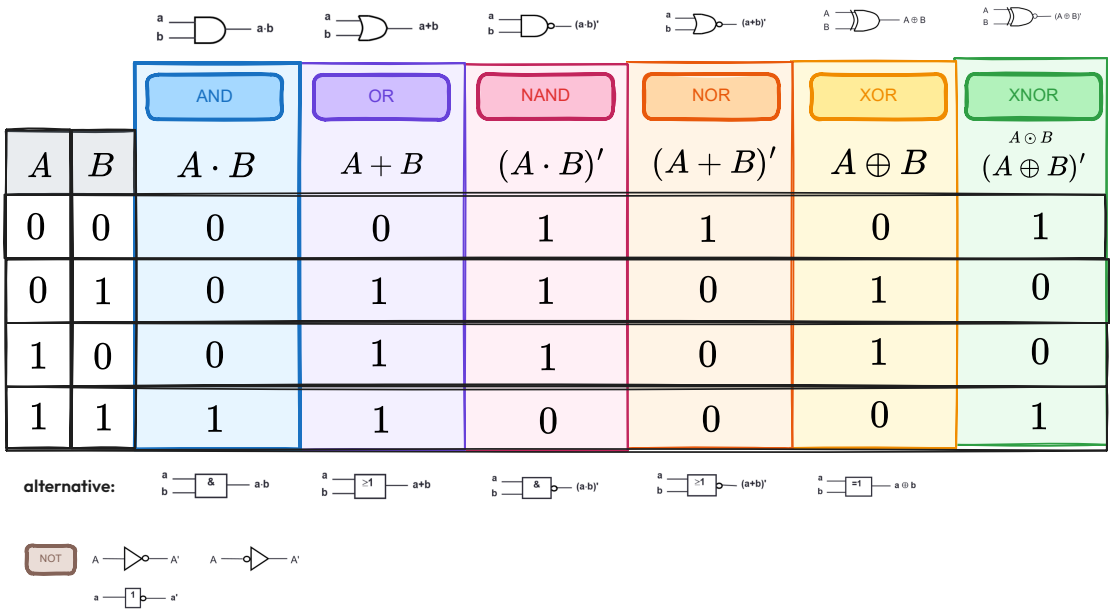
the product-of-maxterms

$$
\begin{aligned}
F &= (x + y') \cdot (x' + y) \\
&= M1 \cdot M3 \\
&= \prod M(1, 3)
\end{aligned}
$$

$$
\begin{aligned}
F' &= M0 \cdot M2 \\
F &= (M0 \cdot M2)' \\
&= M0' + M2' && \text{De Morgan's Law} \\
&= m0 + m2 && mx' = Mx
\end{aligned}
$$

Logic Circuit

Logic Gates



		AND	OR	NAND	NOR	XOR	XNOR
A	B	$A \cdot B$	$A + B$	$(A \cdot B)'$	$(A + B)'$	$A \oplus B$	$A \odot B$ $(A \oplus B)'$
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

Logic Circuits

> `fan-in`
>
> Number of inputs of a gate

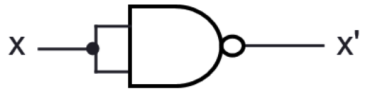
Universal Gates

> `AND` , `OR` , `NOT` **gates are sufficient for building any boolean function.**
>
> This is called a complete set of logic.

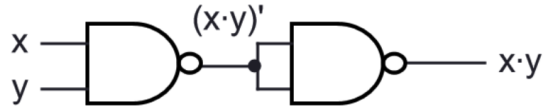
Other gates may still be used due to

- usefulness (`XOR` : parity bit generation)
- economical
- self-sufficient (`NAND/NOR` gates)

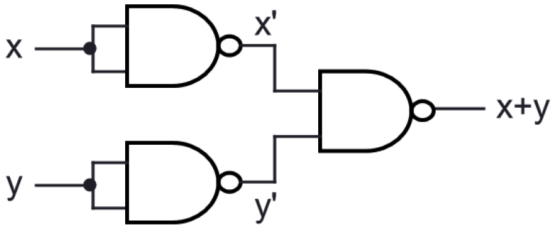
`NAND` as a universal gate



$(x \cdot x)' = x'$ (idempotency)

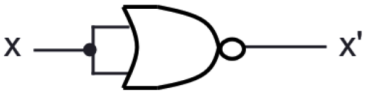


$((x \cdot y)' \cdot (x \cdot y)')' = ((x \cdot y)')'$ (idempotency)
$= x \cdot y$ (involution)

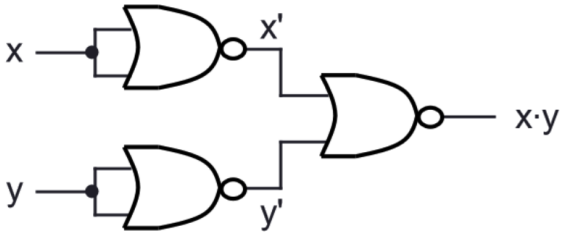


$((x \cdot x)' \cdot (y \cdot y)')' = (x' \cdot y')'$ (idempotency)
$= (x')' + (y')'$ (DeMorgan)
$= x + y$ (involution)

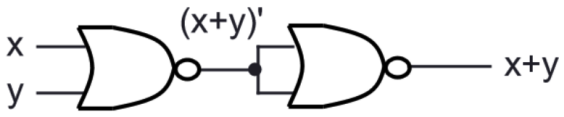
NOR as a universal gate



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



$((x + x)' + (y + y)')' = (x' + y')'$ (idempotency)
$= (x')' \cdot (y')'$ (DeMorgan)
$= x \cdot y$ (involution)



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

Implementing SOP and POS

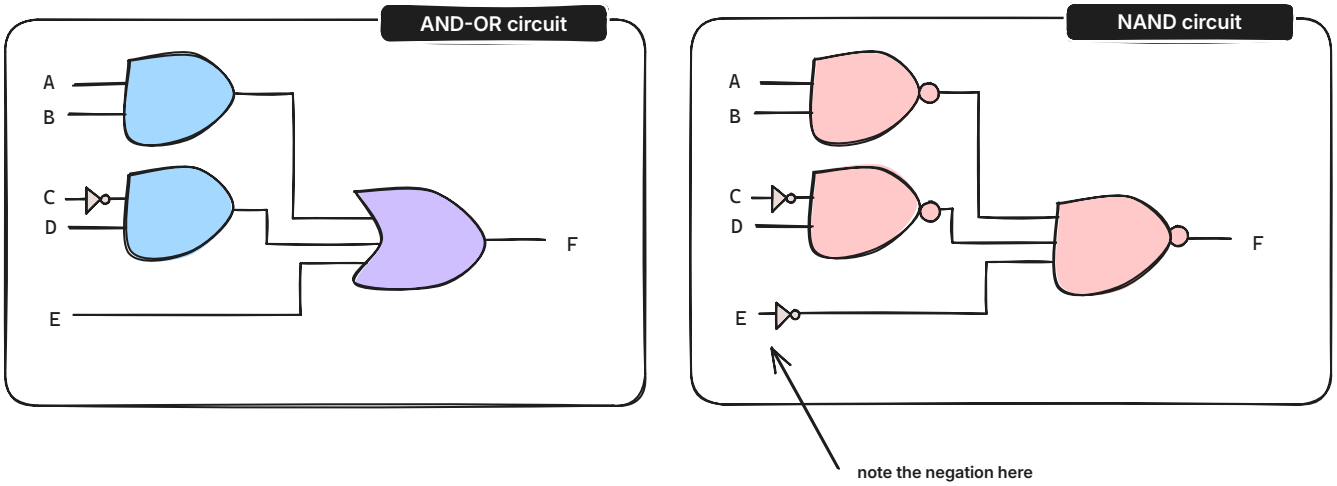
SOP expressions

Sum-of-product expression implementation

Can be implemented using:

1. 2 level AND-OR circuit (first level: AND, second level: OR)
2. 2 level NAND circuit

$$F = A \cdot B + C' \cdot D + E$$



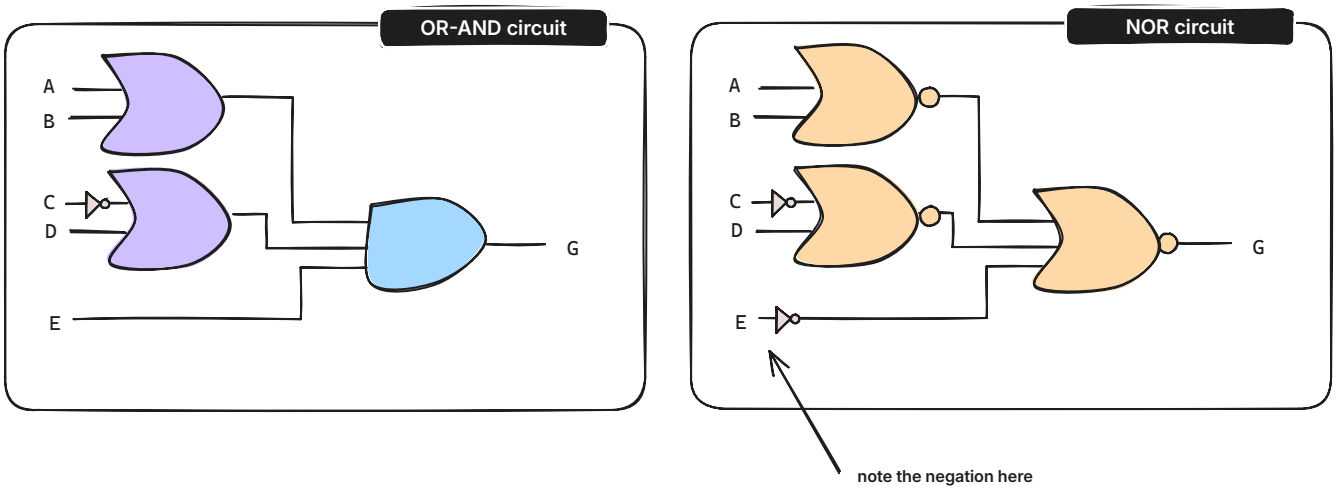
POS expressions

Product-of-sums expression implementation

Can be implemented using:

1. 2 level OR-AND circuit (first level: OR, second level: AND)
2. 2 level NOR circuit

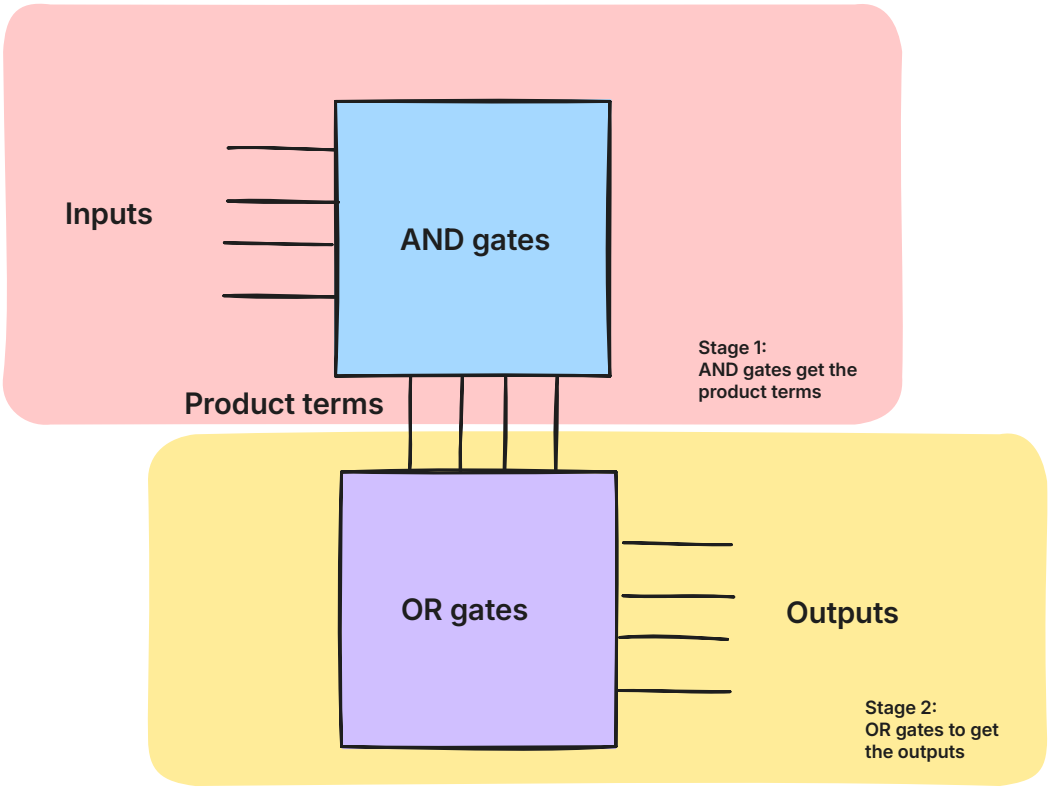
$$G = (A + B) \cdot (C' + D) \cdot E$$



Programming Logic Array

Programming logic array

Programmable integrated circuit implements sum-of-products circuits (to allow multiple outputs)


Inputs
AND gates
Stage 1:
AND gates get the
product terms
Product terms
OR gates
Outputs
Stage 2:
OR gates to get
the outputs

Simplification

Purpose of simplification

Simpler expression leads to circuit with lesser logic gates, resulting in cheaper, low-power-requirement, and sometimes faster.

Techniques

Algebraic Simplification

Aims to minimise number of literals and number of terms.

- Example 1: Simplify $(x+y)\cdot(x+y')\cdot(x'+z)$

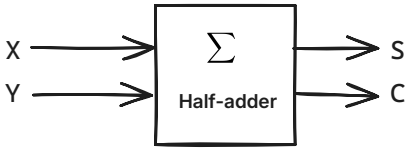
$$
\begin{aligned}
&(x+y)\cdot(x+y')\cdot(x'+z) \\
&= (x\cdot x + x\cdot y' + x\cdot y + y\cdot y')\cdot(x'+z) && \text{(distributivity)} \\
&= (x + x\cdot y' + x\cdot y + y\cdot y')\cdot(x'+z) && \text{(idempotency)} \\
&= (x + x\cdot(y'+y) + y\cdot y')\cdot(x'+z) && \text{(distributivity)} \\
&= (x + x\cdot(1) + 0)\cdot(x'+z) && \text{(complement)} \\
&= (x + x)\cdot(x'+z) && \text{(identity)} \\
&= x\cdot(x'+z) && \text{(idempotency)} \\
&= x\cdot x' + x\cdot z && \text{(distributivity)} \\
&= 0 + x\cdot z && \text{(complement)} \\
&= x\cdot z && \text{(identity)}
\end{aligned}
$$

Number of literals reduced from 6 to 2.

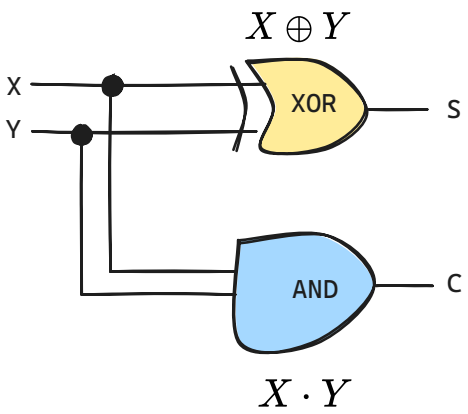
Half-Adder

Half adder

A circuit that adds 2 single bits (X, Y) to produce a result of 2 bits (C, S).



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



made by zaidan sani.

$$
\begin{aligned}
C &= X \cdot Y \\
S &= X' \cdot Y + X \cdot Y' \\
&= X \oplus Y
\end{aligned}
$$

Gray Code

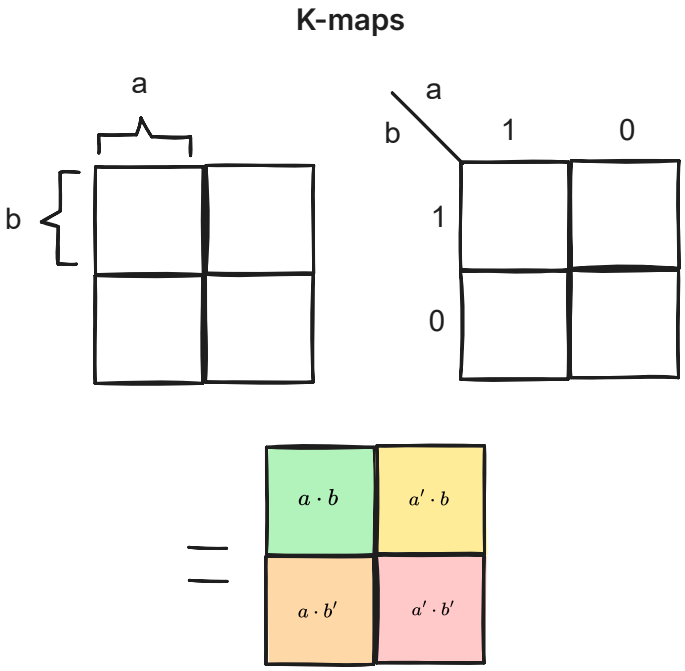
- Unweighted (not an arithmetic code)
- Only a single bit change

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

K-Maps

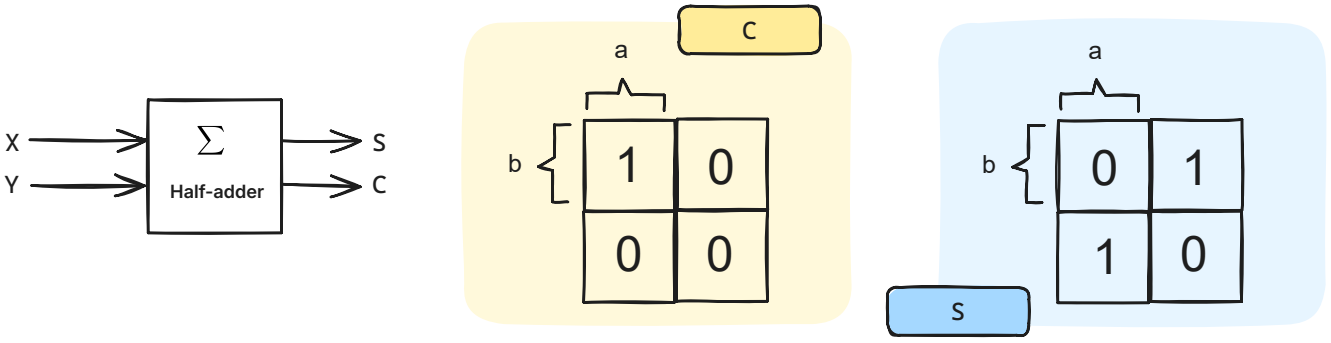
> Karnaugh-map
>
> Abstract form of Venn diagram, organised as a matrix of squares, where each square represents a minterm, and two adjacent squares represent minterms that differ by exactly one literal.

Objective: Fewest possible product terms and literals.
Advantage: Easy to use
Disadvantage: Limited to 5 or 6 variables



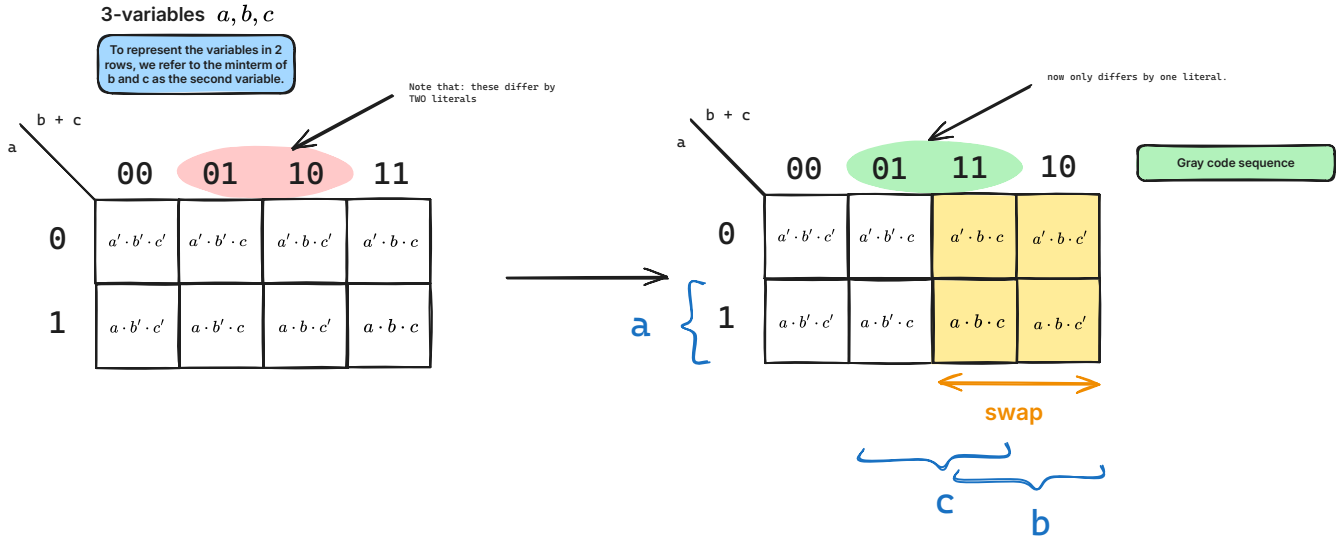
> Filling the K-map
>
> Put a `1` to correspond to a minterm in the function, and otherwise.

$$S = a \cdot b' + a' \cdot b$$

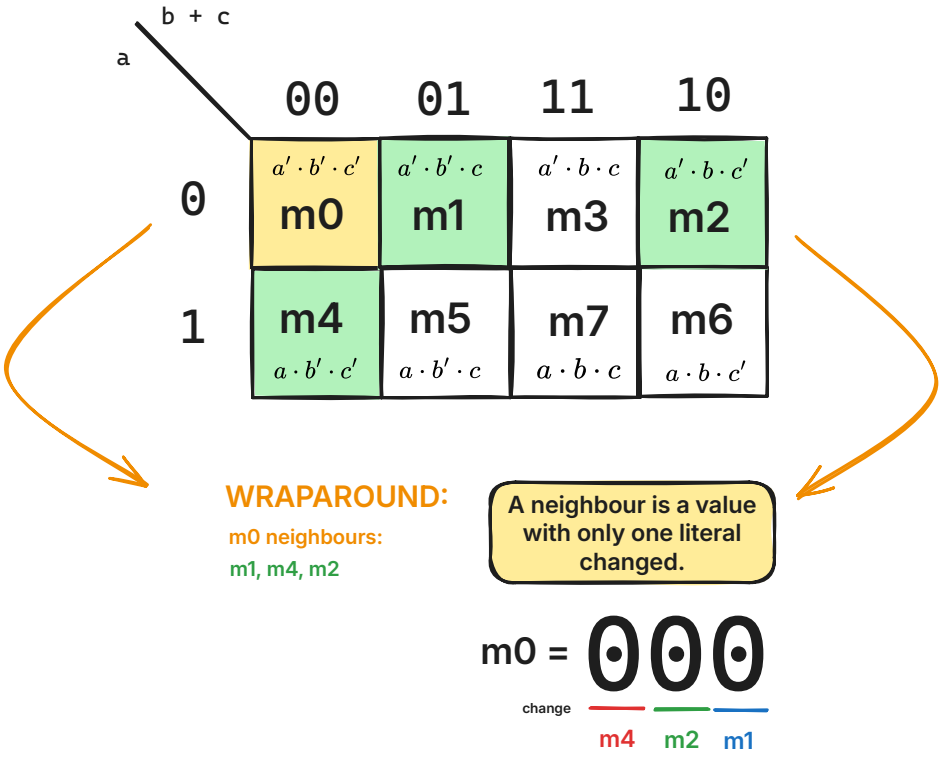


$$S = a \cdot b' + a' \cdot b$$

Variable K-maps



Each cell in a $n-$ variable K-map has n adjacent neighbours. The K-map allows for this with a wrap-around.



made by zaidan sani.

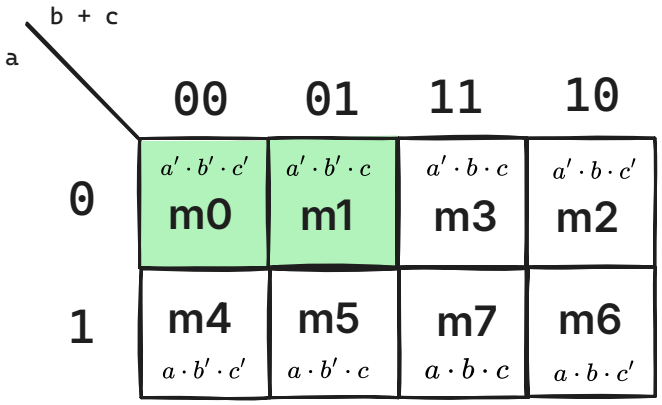
As the number of variables increase, it is harder to draw out. For $5-$variable K-maps, where there will be an expected $2^5 = 32$ amount of squares, it can be organised as two $4-$ variable K-maps, where, one K-map is on top of the other.

Using K-maps

$$A + A' = 1$$

The unifying theorem (complement law) allows us to use

- each valid grouping of adjacent cells to correspond to a simpler product term of F



Assume that the green squares are the minterms with output `1` for function F.

We currently have:

$$\begin{aligned} F &= m0 + m1 \\ &= a' \cdot b' \cdot c' + a' \cdot b' \cdot c \end{aligned}$$

Since $m0, m1$ are adjacent, we can group them together, meaning we eliminate the variable c from the product term.

$$\begin{aligned} F &= (a' \cdot b')(c' + c) \qquad \text{Complement law} \\ &= a' \cdot b' \end{aligned}$$

To use effectively:

1. Group as many cells as possible
   - The larger the group, the fewer the number of literals in product term.
2. Select as few groups as possible (the lesser the groups, the fewer the number of product terms)

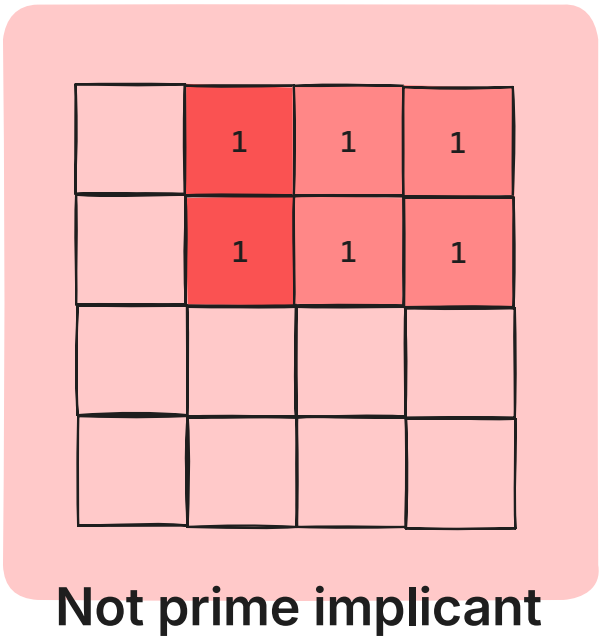
Implicants

Implicants

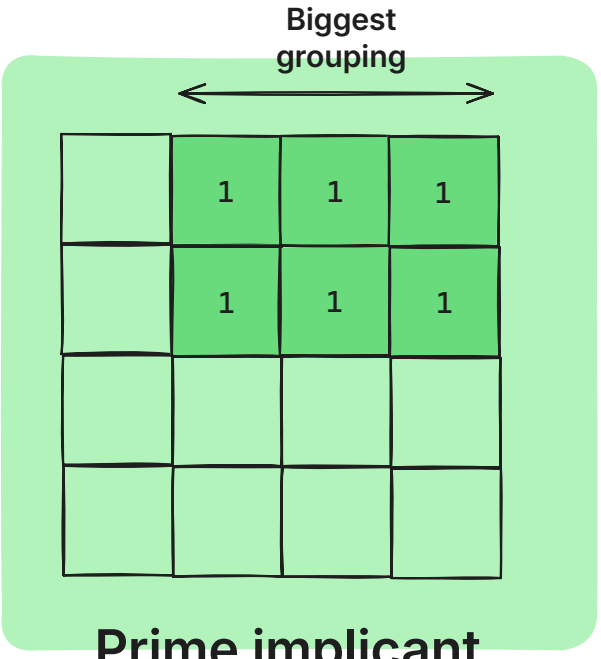
A product term that could be used to cover minterms of a function

Prime implicants

A product term obtained by combining the maximum possible number of minterms from adjacent squares in the map.



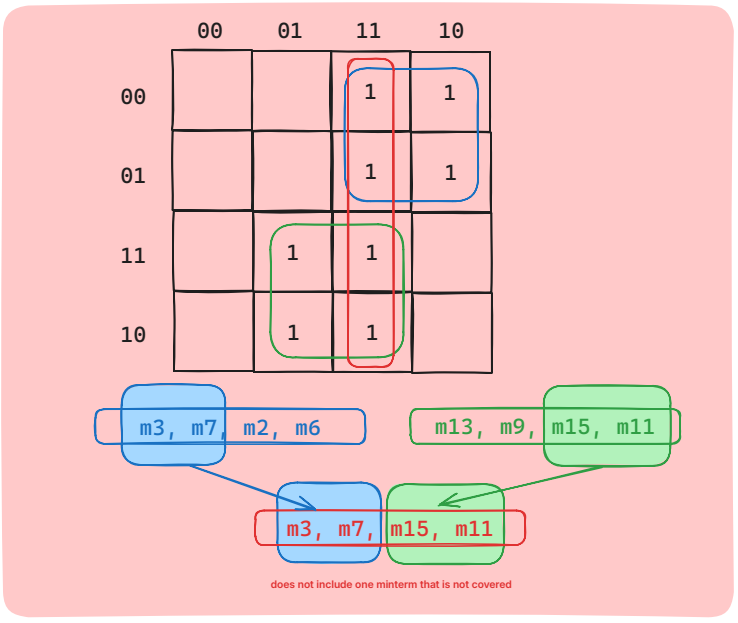
Not prime implicant

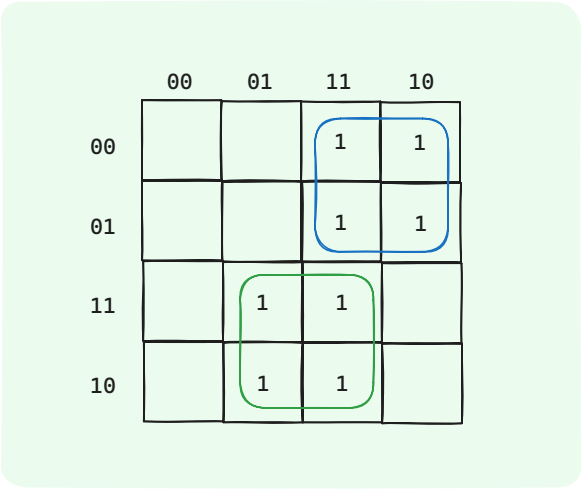


Prime implicant

Essential prime implicants

A prime implicant that includes at least one minterm that is not covered by any other prime implicant

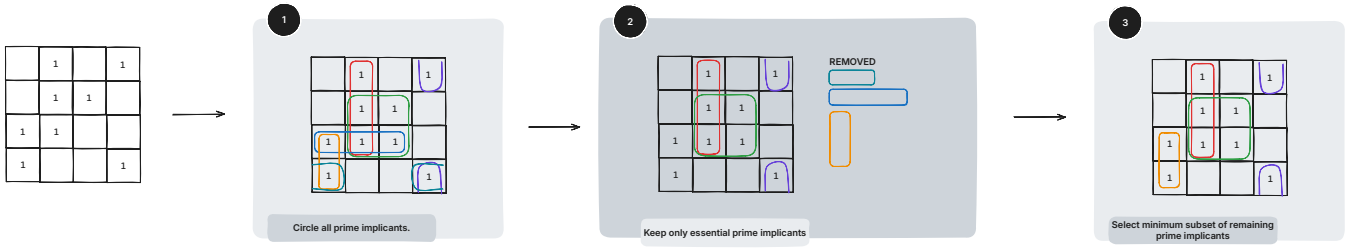




only prime implicants

Algorithm

1. Circle all prime-implicants
2. Identify and select all essential prime-implicants
3. Select a minimum subset



made by zaidan sani.

Simplified POS expression

The Simplified POS expression can be found using two methods:

- the K-map of the maxterms, and grouping 0s together
- complement of the SOP expression

Don't Care conditions

Don't care conditions

In some cases, outputs are not specified or are invalid. These outputs can then be either `1` or `0`. These conditions are denoted `X` or `d`.

A	B	C	D
0	0	0	0
0	1	1	1
1	1	1	0
1	1	1	1

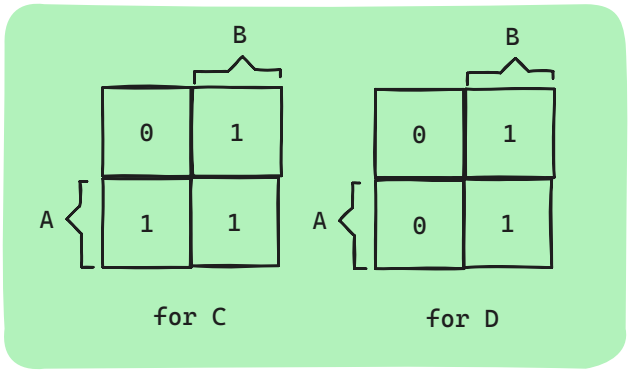
example possible truth table (if all inputs valid)

A	B	C	D
0	0	X	X
0	1	1	1
1	1	1	0
1	1	X	X

example possible truth table if inputs 00 & 11 invalid

This changes the eventual SOP/POS expressions.

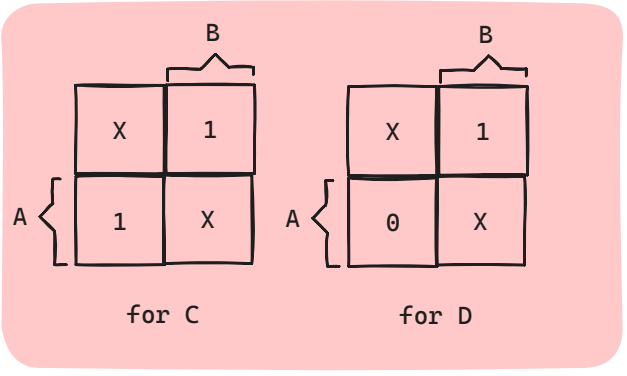
Assuming all inputs are valid:



$C = m(1,2) = A + A \cdot B$

$D = \sum m(1,3) = B$

Assuming all inputs are invalid:



$C = \sum m(1,2) + \sum d(0,4) = 1$

$D = m(1) + \sum d(0,4) = A' \cdot B + B$

Quine-McCluskey

> This topic is optional.

Phase 1

Step 1: List out all minterms in groups with same number of 1s in their binary codes
Step 2: Combine codes that differ by 1 bit into bigger group and write combined code in the next column
Step 3: Repeat step 2

Phase 2 Identify EPIs

Step 1: Draw PI chart and spot EPIs - EPIs are columns containing a single tick
Step 2: Draw reduced PI chart if minterms are not covered.

- Find out minterms covered by EPIs
- Remove EPIs and minterms covered from the chart
- Find minimum number of remaining PIs to cover remaining minterms

MSI Components

Integrated Circuit

Also known as IC, chip, or microchip

Set of electronic circuits on one small flat piece of semiconductor material

Scale of integration

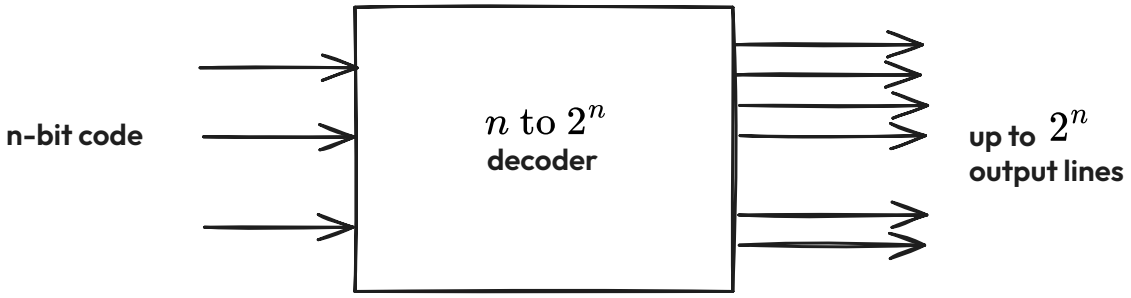
Number of components fitted into standard size IC.

Decoder

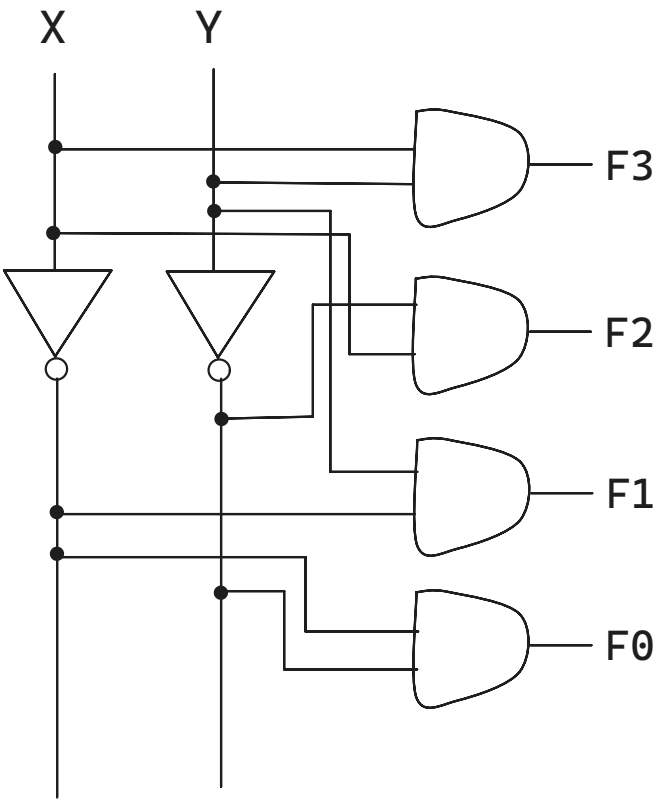
Decoder

Input: a code (from n input lines)
Output: 2^n output lines.

`n-to-m`, `n:m`, `n x m` $(m \leq 2^n)$



X	Y	F0	F1	F2	F3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



2x4 decoder

Each output of a $n \times m$ decoder is a minterm of a $n-$ variable function.

Implementing functions

A Boolean function in sum-of-minterms form can utilise a decoder to generate the minterms, and an OR gate to form the sum.

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

> Good when circuit has many outputs, and each function is expressed with a few minterms.

Enable

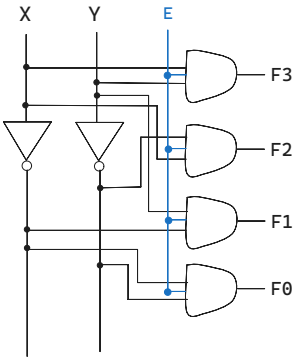
Decoders often come with an enable control signal so that device is only activated when $E = 1$.

E	X	Y	F0	F1	F2	F3
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	0	0	0	0	0	0

HIGH-ENABLED

1-enabled

2x4 decoder

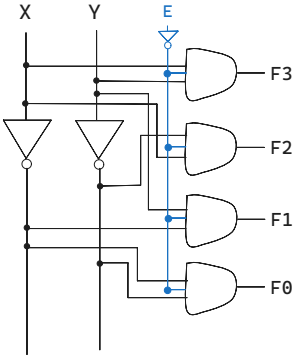


E	X	Y	F0	F1	F2	F3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	0	0	0	0	0	0

LOW-ENABLED

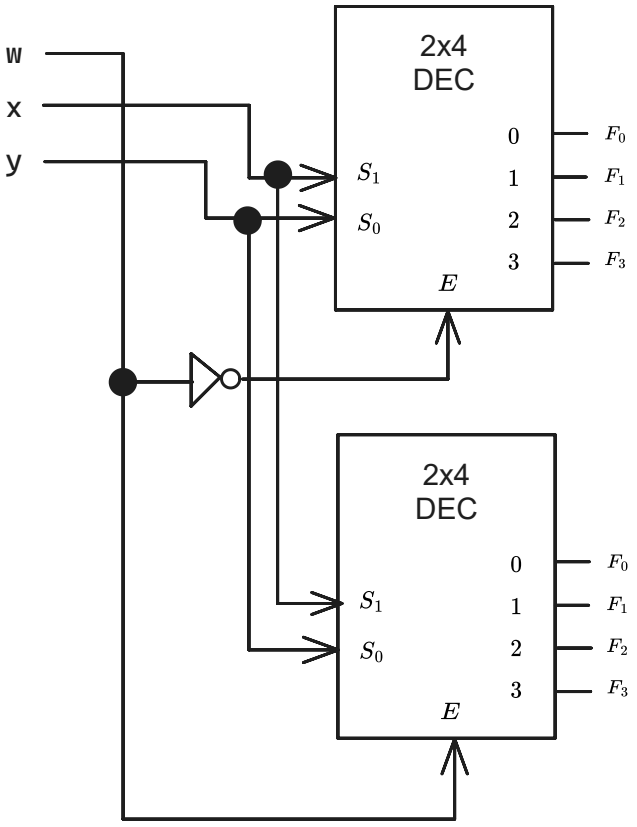
0-enabled

2x4 decoder



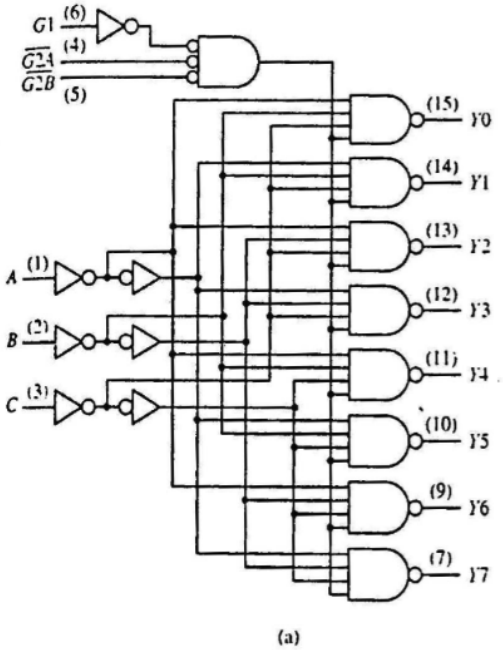
Constructing Larger Decoders

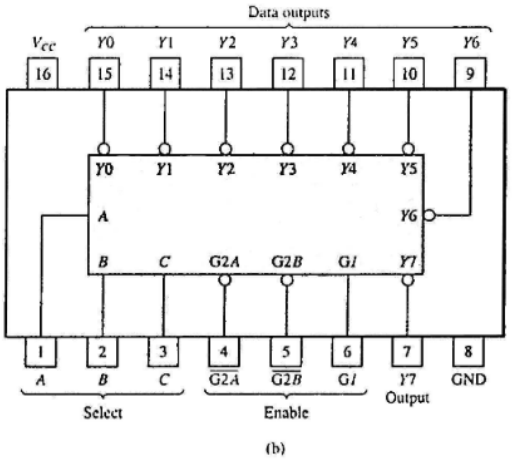
Larger decoders can be constructed from smaller ones.



MSI Decoder

- 74138 (3-to-8 decoder)





74138 decoder module.
(a) Logic circuit.
(b) Package pin configuration.

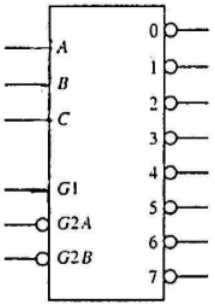
INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	$\overline{G2}$*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

*$\overline{G2} = \overline{G2A} + \overline{G2B}$
H = high level, L = low level, X = irrelevant
(c)

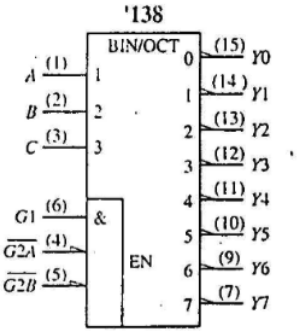
74138 decoder module.
(c) Function table.

74138 decoder module.
(d) Generic symbol.
(e) IEEE standard logic symbol.

Source:The Data Book Volume 2, Texas Instruments Inc.,1985



(d)



(e)

Encoding

Encoder

Input: 2^n input lines
Output: n bits of code

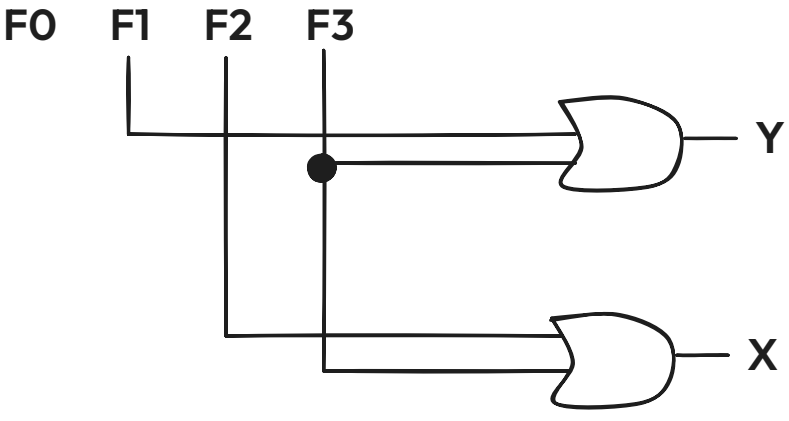
X	Y	F0	F1	F2	F3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(the rest are don't cares)

With this, we can obtain:

$$X = F2 + F3$$
$$Y = F1 + F3$$

4x2 encoder



At any one time, only one input line of an encoder has a value of `1` (high), the rest are `0` (low).

Priority Encoders

A priority encoder is one with priority

- if two or more inputs, inputs with highest priority takes precedence:

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

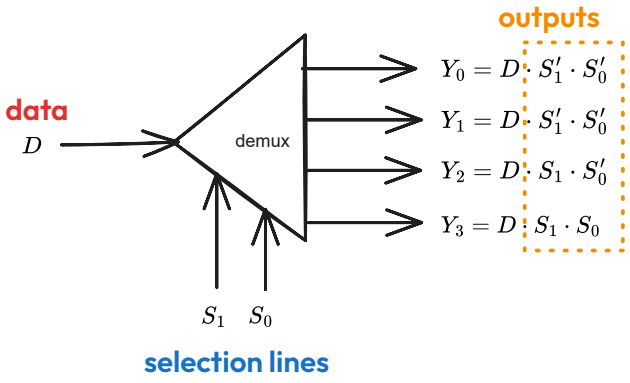
Helps share a single communication line among a number of devices. Only one source and one destination can use the communication line.

Demultiplexer

Demultiplexer

Directs data from input to one selected output line.

Input: Input line, set of selection lines
Output: Data from input to one selected output line



Demultiplexer circuit is identical to a decoder with enable.

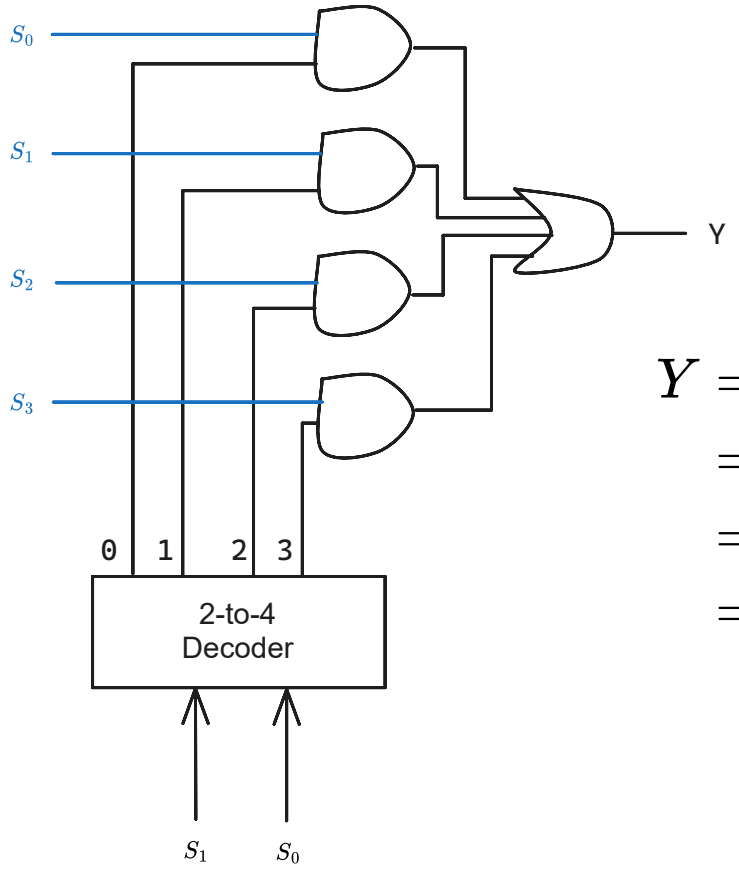
Multiplexer

Multiplexer

(Data selector)

Input: A number of input lines, a number of selection lines
Output: Data to one output line - sum of (product of data lines and selection lines)

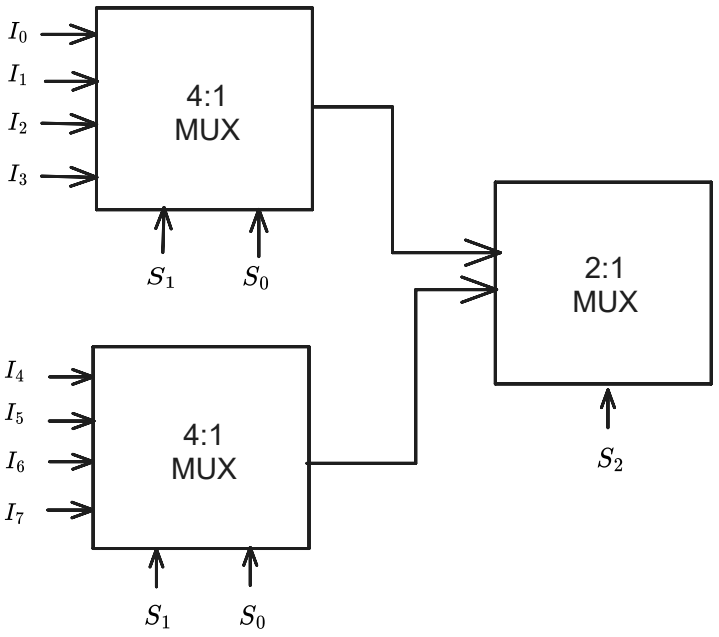
A 2^n-to-1-line multiplexer - $2^n : 1$ MUX is made from an $n : 2^n$ decoder by adding to it 2^n input lines, one to each `AND` gate:



$$
\begin{aligned}
Y &= I_0 \cdot (S_1' \cdot S_0') \\
&= I_1 \cdot (S_1' \cdot S_0) \\
&= I_2 \cdot (S_1 \cdot S_0') \\
&= I_3 \cdot (S_1 \cdot S_0)
\end{aligned}
$$

Larger Multiplexers

Larger multiplexers can be constructed from smaller ones.



Implementing Functions

A 2^n-to-1 multiplexer can implement a Boolean function of n input variables,

1. Express function in sum-of-minterms form
2. Connect n variables to the n selection lines
3. Put a `1` on a dataline if it is a minterm of the function, and `0` otherwise.

Using Smaller Multiplexers

1. Express Boolean function in sum-of-minterms-form
2. Reserve one variable for input lines of MUX, and use the rest for selection lines
3. Draw truth table for function, by grouping inputs by selection line values, then determine multiplexer inputs by comparing input line and function for corresponding selection line values.

> if C is the chosen line, note the F output for C = `0,1` for each input line.

Combinational Circuits

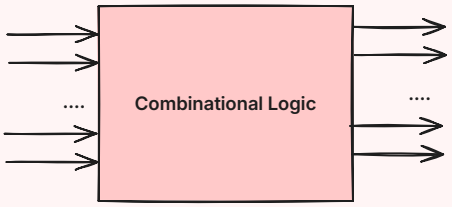
Combinational circuit

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

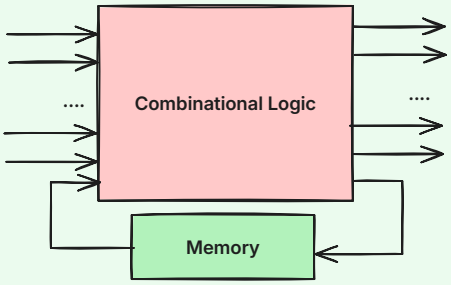
Sequential circuit

Each output depends on both present inputs and state

Combinational Circuit

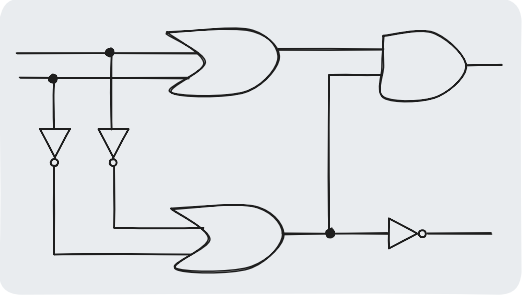


Sequential Circuit

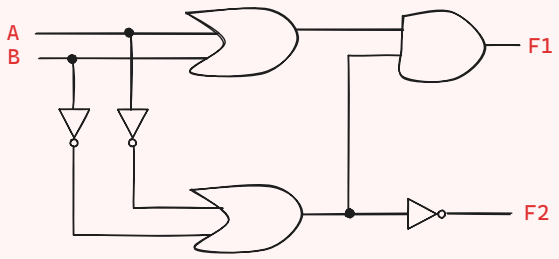


Analysis Procedure

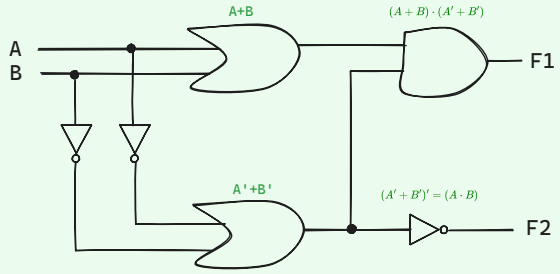
1. Label inputs and outputs
2. Obtain the functions of intermediate points and the outputs
3. Draw the truth table
4. Deduce the functionality of circuit



Step 1: Label inputs and outputs



Step 2: Functions of intermediate points



Step 3: Truth Table

A	B	$A+B$	$A'+B'$	$F1$	$F2$
0	0	0	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	1	0	0	1

Step 4: Deduce functionality

Half-adder.

Design Methods

Combinational circuit design methods:

- Gate level design method (with logic gates)
- Block level design method (with functional blocks)

Design methods use logic gates and function blocks available as Integrated Circuit (IC) chips. Types of IC chips based on packing density: SSI, MSI, LSI, VLSI, ULSI.

Main objectives:

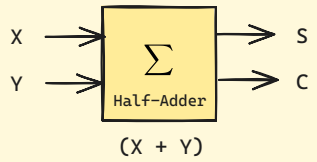
- Reduce cost (number of gates for small circuits, number of IC packages for complex circuits)
- Increase speed
- Design simplicity (block reuse)

Gate-level design

1. State problem
2. Determine label inputs and outputs of circuit
3. Draw truth table
4. Obtain simplified Boolean functions
5. Draw logic diagram

Step 1 Problem:
Build a half-adder

Step 2 Determining and labeling input/outputs



(X + Y)

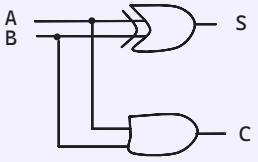
Step 3 Truth table

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

Step 4 Obtain simplified Boolean functions

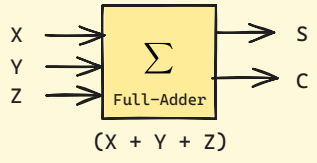
$$C = X \cdot Y$$
$$S = X \oplus Y$$

Step 5 Logic diagram



Step 1 Problem:
Build a full-adder

Step 2 Determining and labeling input/outputs



(X + Y + Z)

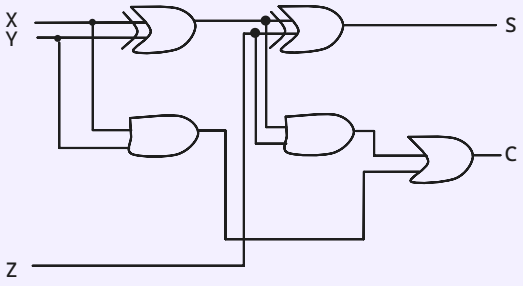
Step 3 Truth table

X	Y	Z	S	C
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

Step 4 Obtain simplified Boolean functions

$$\begin{aligned} C &= X \cdot Y + X \cdot Z + Y \cdot Z \\ &= X \cdot Y + (X \oplus Y) \cdot Z \\ S &= X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z \\ &= X \oplus Y \oplus Z \end{aligned}$$

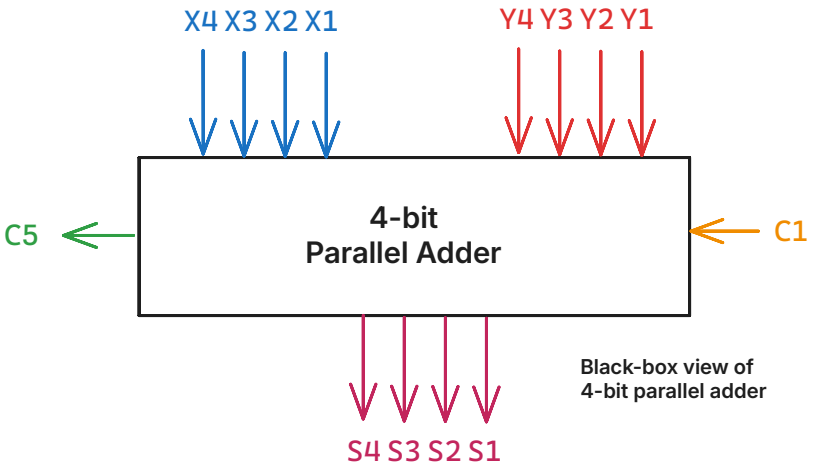
Step 5 Logic diagram



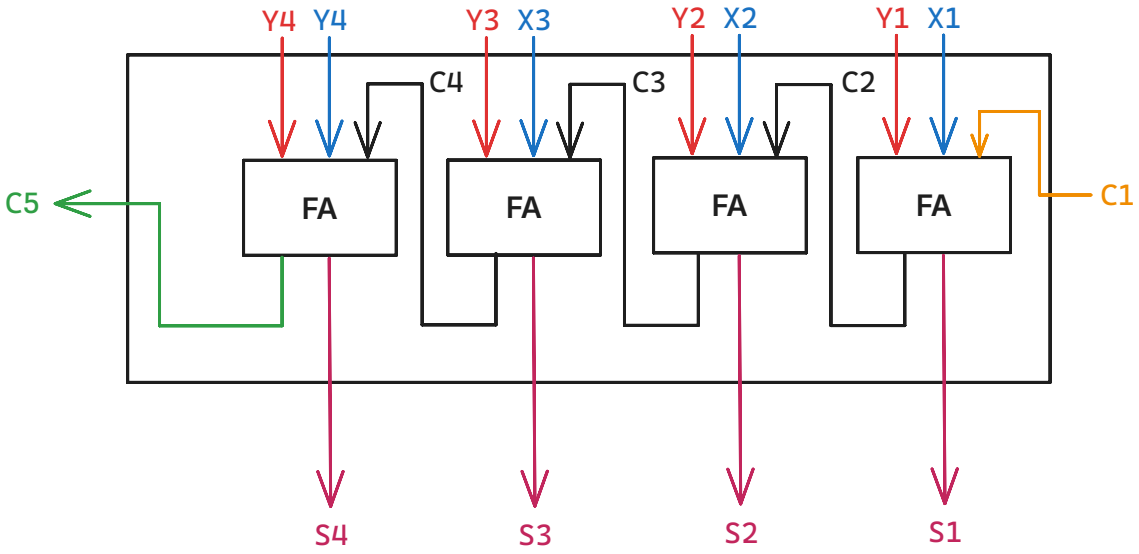
Block-Level Design

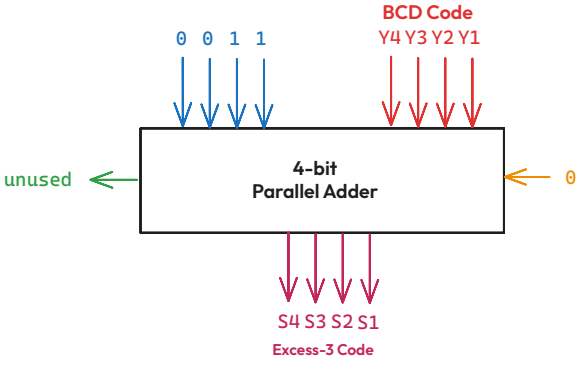
More complex circuits are built using block-level methods. We can use **4-bit parallel adders** as building blocks, we can create the following:

1. BCD-to-Excess-3 Code Converter
2. 16-bit Parallel Adder

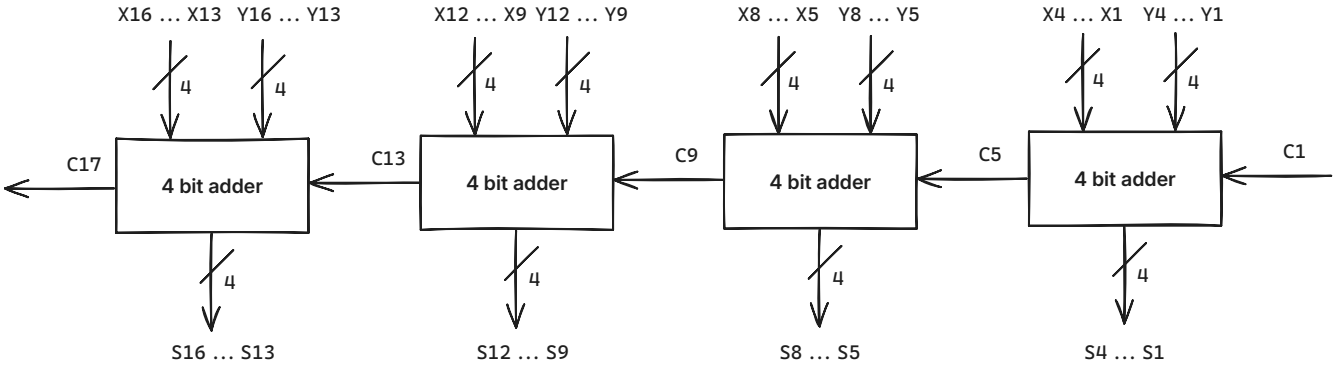


Black-box view of
4-bit parallel adder





16-bit Parallel Adder



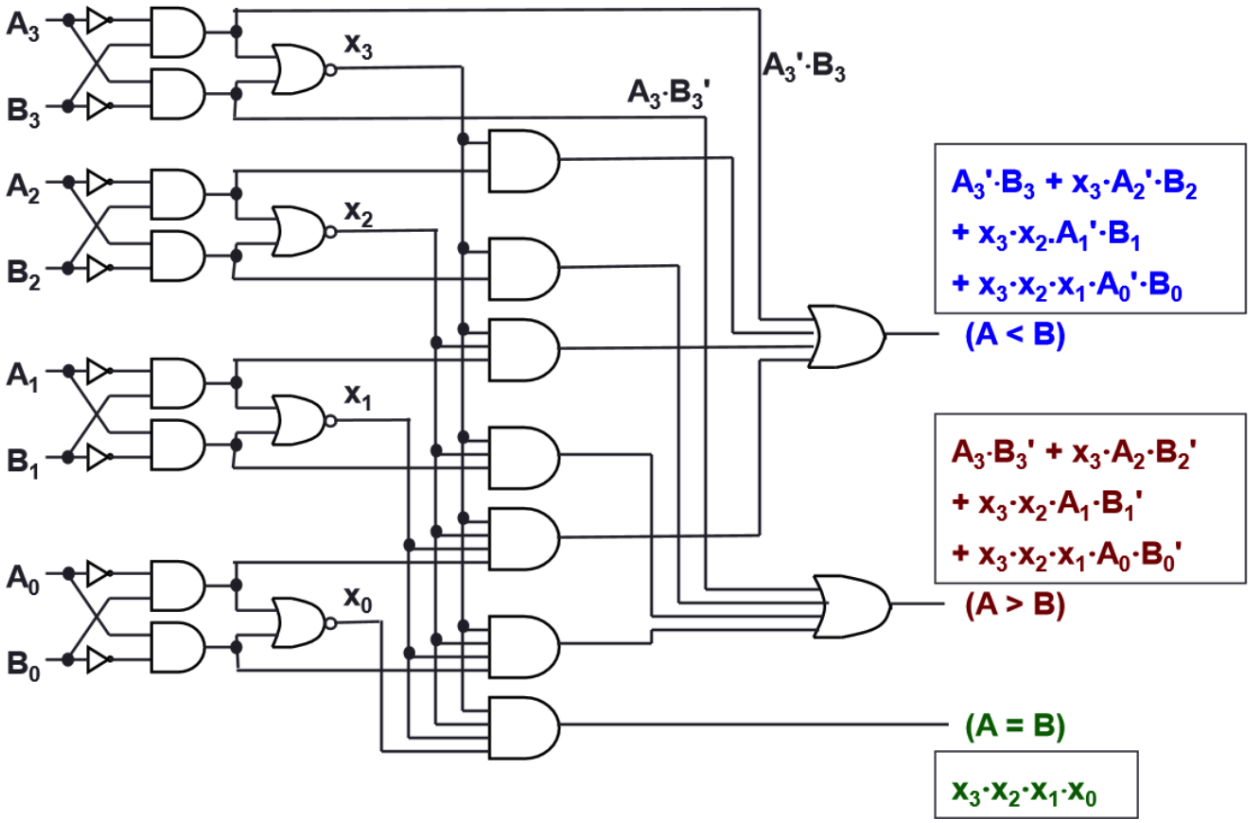
made by zaidan sani.

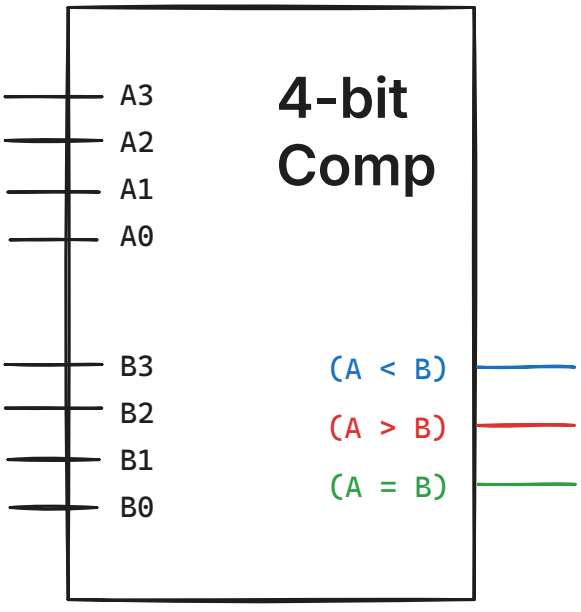
Magnitude Comparator

Magnitude Comparator

Compares 2 unsigned values A, B to check $A > B, A = B, A < B$

$$\text{if } A_n > B_n : A > B$$
$$\text{if } A_n < B_n : A < B$$
$$\text{if } A_n = B_n : \text{check } A_{n-1}, B_{n-1}$$





Circuit Delays

Circuit delay

Given a logic gate with delay t, if inputs are stable at times $t_1, \ldots, t_n$, then the earliest time in which the output will be stable is:

$$max(t_1, \ldots, t_n) + t$$

Sequential Logic

There are 2 types of sequential circuits:

- Synchronous: outputs change at specific time
- Asynchronous: Changes at any time

Multivibrator

Class of sequential circuits

- Bistable (2 stable states)
- Monostable or one-shot (1 stable state)
- Astable (no stable state)

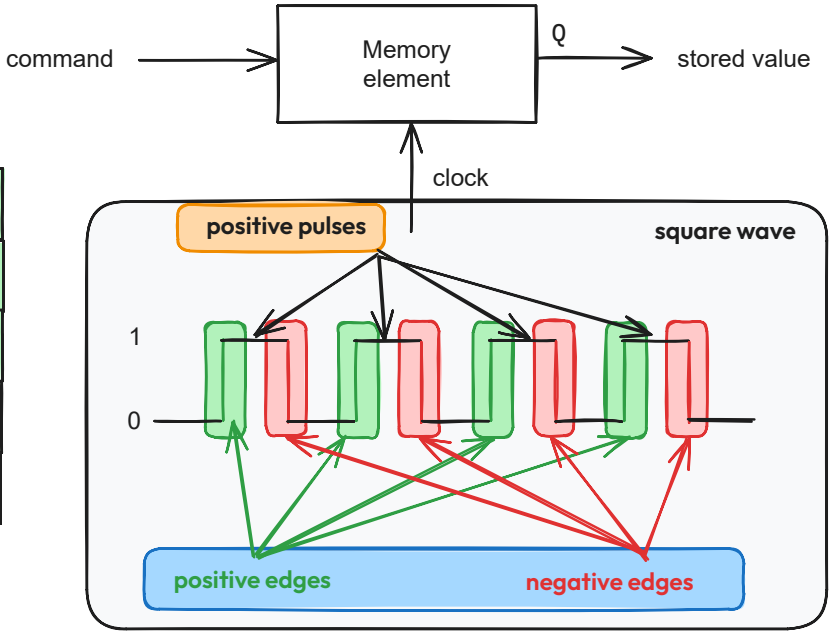
Bistable logic devices

Latches, and flip flops.

Memory Elements

Memory element

A device which can remember value indefinitely, or change value on command from its inputs.



Command at time (t)	current state Q(t)	next state Q(t+1)
set	X	1
reset	X	0
memorise	1	1
	0	0

Edge-triggered

Flip-flops

Positive edge-triggered (ON on positive edges, OFF otherwise)

Negative edge-triggered (ON on negative edges, OFF otherwise)

Pulse-triggered

Latches

ON = 1, OFF = 0

Latch

S–R Latch

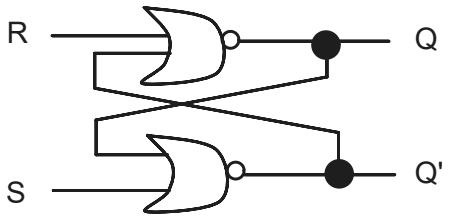
S–R Latch

Two inputs: `S` and `R`
Two complementary outputs: `Q` and `Q'`

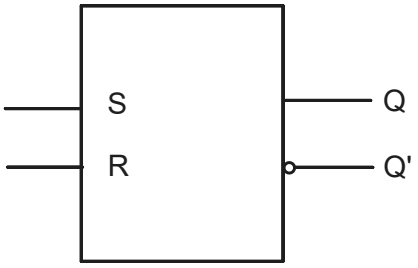
State

`Q` = HIGH: `SET`
`Q'` = LOW: `RESET`

S	R	Q	Q'	
1	0	1	0	LOW
0	0	1	0	initial
0	1	0	1	HIGH
0	0	0	1	HIGH
1	1	0	0	invalid



S	R	Q	Q'	
0	0	NC	NC	No change.
1	0	1	0	Latch SET
0	1	0	1	Latch RESET
1	1	0	0	Invalid



Active-Low `S–R` Latch

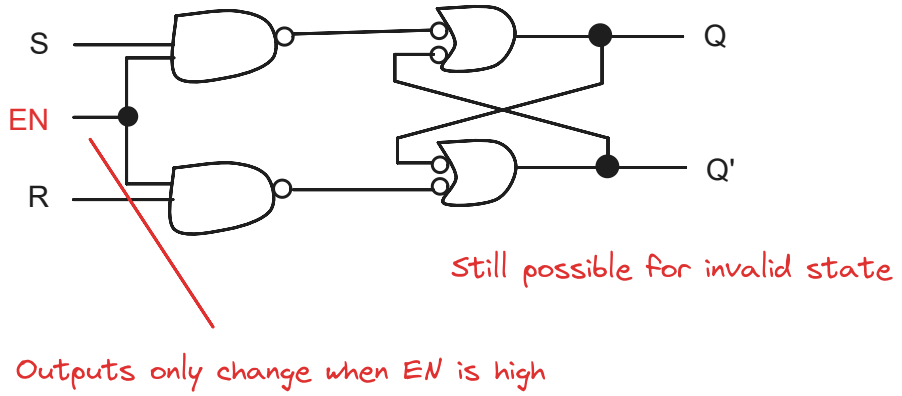
NAND gates are used instead for this. For these latches, given the input `SR`:

- `00` becomes an invalid command (no change in Active-High)
- `11` becomes a nochange command (invalid in Active-High)
- `01` becomes a set command (reset in Active-High)
- `10` becomes a reset command (set in Active-High)

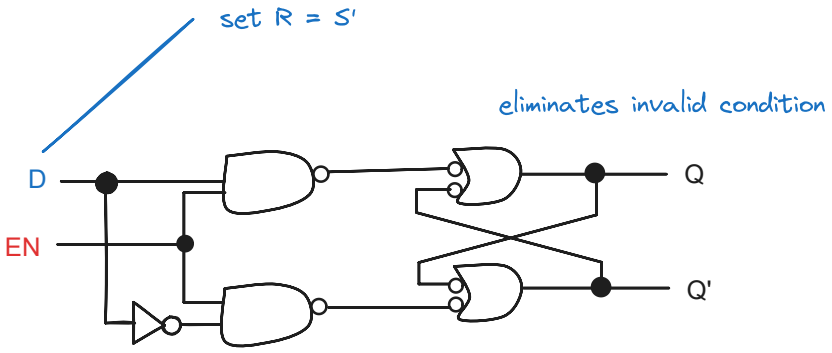
Gated S–R Latch

Gated `S–R` Latch

`S–R` latch + enable input (EN) and 2 NAND gates



Gated D Latch



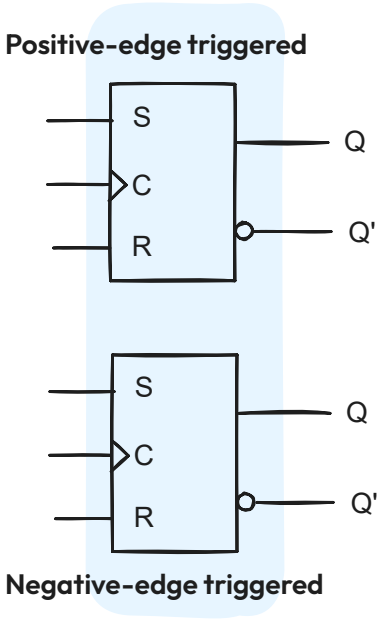
EN	D	Q(t+1)	
1	0	0	Reset
1	1	1	Set
0	X	Q(t)	No Change

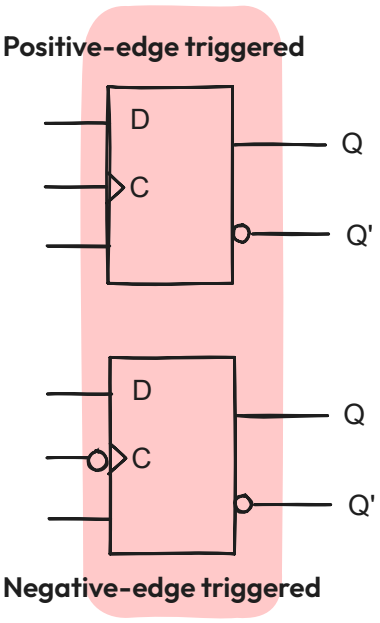
D

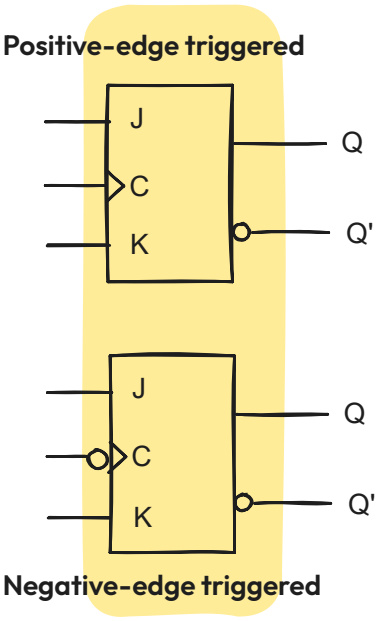
Flip-Flops

Synchronous bistable devices

Output changes state as a specified point on a triggering input called the clock:







S-R flip-flop

has invalid conditions

D flip-flop

convert S-R flip-flop into a D flip-flop; add an inverter

Use Case

Parallel data transfer

J-K flip flop

Q and Q' are fed back to pulse-steering NAND gates

Toggle state when JK = 11

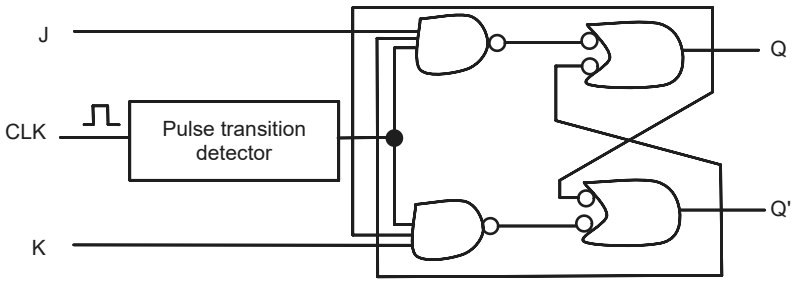
S–R flip flop

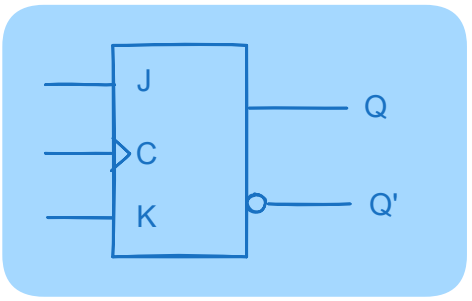
S	R	CLK	$Q(t+1)$	
0	0	X	$Q(t)$	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid

D flip flop

D	CLK	$Q(t+1)$	
1	↑	1	Set
0	↑	0	Reset

J–K flip flop

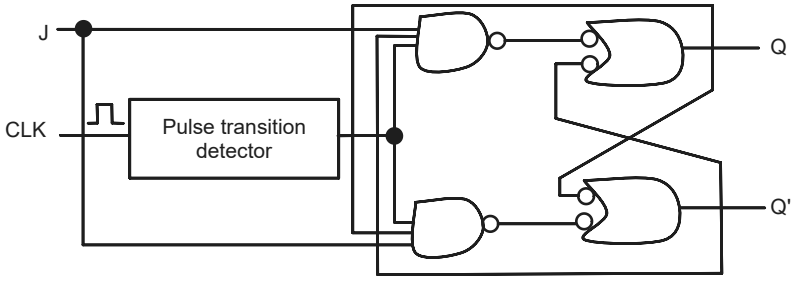


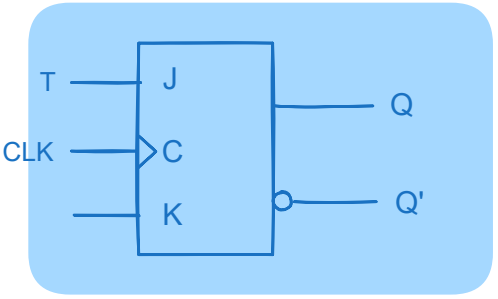


J	K	CLK	$Q(t+1)$	
0	0	↑	$Q(t)$	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	$Q(t)'$	Toggle

$$Q(t+1) = J \cdot Q' + K' \cdot Q$$

T flip flop





T	CLK	$Q(t+1)$	
0	↑	$Q(t)$	No change
1	↑	$Q(t)'$	Toggle

$$Q(t+1) = T \cdot Q' + T' \cdot Q$$

Asynchronous Inputs

The `S-R`, `D`, `J-K` inputs are synchronous inputs as data on these inputs are transferred to the flip-flop's output only on the triggered edge of the clock pulse.

Asynchronous

Affect the state of the flip flop independent of the clock

`PRE = HIGH` : Q is immediately set to HIGH

`CLR = HIGH` : Q is immediately cleared to LOW

Flip-flop in normal operation mode when both PRE and CLR are LOW.

made by zaidan sani.

Synchronous Sequential Circuits

> **Note**
>
> Flip flops make up the memory while the gates form one or more combinational sub-circuits.

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

D	CLK	$Q(t+1)$	
1	↑	1	Set
0	↑	0	Reset

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

T	CLK	$Q(t+1)$	
0	↑	$Q(t)$	No change
1	↑	$Q(t)'$	Toggle

Analysis of Sequential Circuits

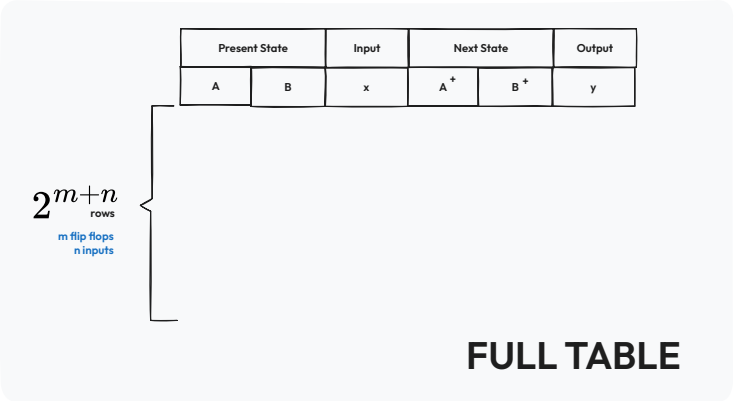
> **Analysing behaviour**
>
> 1. Derive state table
> 2. and hence its state diagram

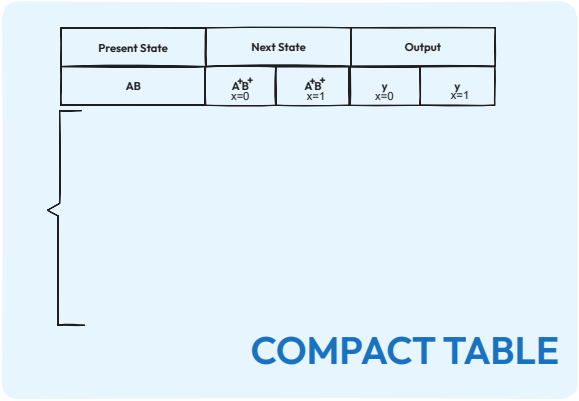
Requires state equations to be derived for the flip-flop inputs as well as output functions for the circuit outputs other than the flip-flops other than flip flops.

$A(t)$ or (A) and $A(t+1)$ or A^+ can represent the present state and next state of a flip-flop represented by A.

> **State table**
>
> Similar to a truth table.
> Inputs and present state on left side.
> Outputs and next state on the right side.

m flip-flops and n inputs $\rightarrow 2^{m+n}$ rows

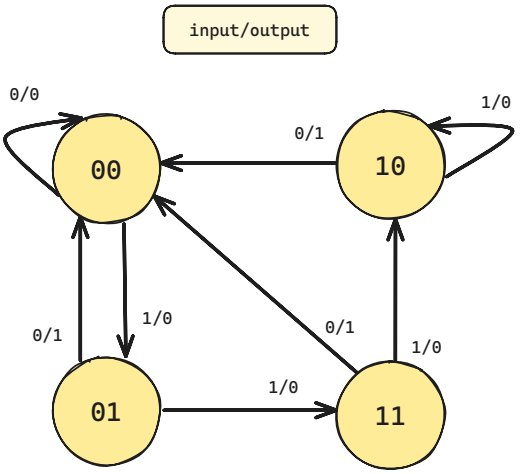




State diagram

- Each state is denoted by a circle
- Each arrow denotes a transition of the sequential circuit

Present State	Next State		Output	
AB	A^+B^+ x=0	A^+B^+ x=1	y x=0	y x=1
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0



Circuit output functions

The outputs of a sequential circuit are functions of the present states of the flip-flops and the inputs. The algebraic description of these are the circuit output functions.

Flip-flop input functions/equations

Algebraic descriptions of the part of the circuit that generates inputs to the flip-flops.

This determines the next state generation - using the functions and the characteristic tables, the next states of flip-flops can be obtained.

Flip-flop Excitation Tables

Analysis

Starting from a circuit diagram, derive state table or state diagram

Characteristic tables

Design

Starting from a set of specifications (in the form of state equations, state table, state diagram), derive logic circuit.

Excitation tables

Excitation tables

Input: Required transition from present state to next state
Output: Flip-flop inputs

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

JK Flip-flop

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

SR Flip-flop

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

D Flip-flop

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

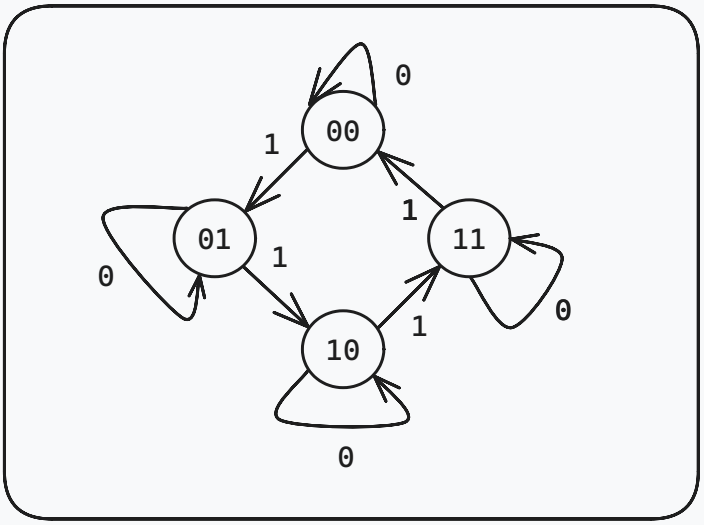
T Flip-flop

Design procedure:

- Circuit specifications
  - Description of circuit behaviour - state diagram/state table
- Derive state table
- Perform state reduction if necessary
- Perform state assignment
- Determine number of flip-flops and label them
- Choose type of flip-flop to be used
- Derive circuit excitation and output tables from the state table
- Derive circuit output functions and flip-flop input functions
- Draw the logic diagram

Worked example:

Given state diagram:



How many flip-flops are needed?

Since two bits are needed to represent each state, two flip-flops are needed.

The flip-flops can be allocated as `A` and `B`.

How many input variables are there?

The input variables are only `0` and `1`, meaning there is one input variable.

The input variable can be allocated `x`.

Present State AB	A^+B^+ **at** $x = 0$	A^+B^+ **at** $x = 1$
00	00	01
01	10	01
10	10	11
11	11	00

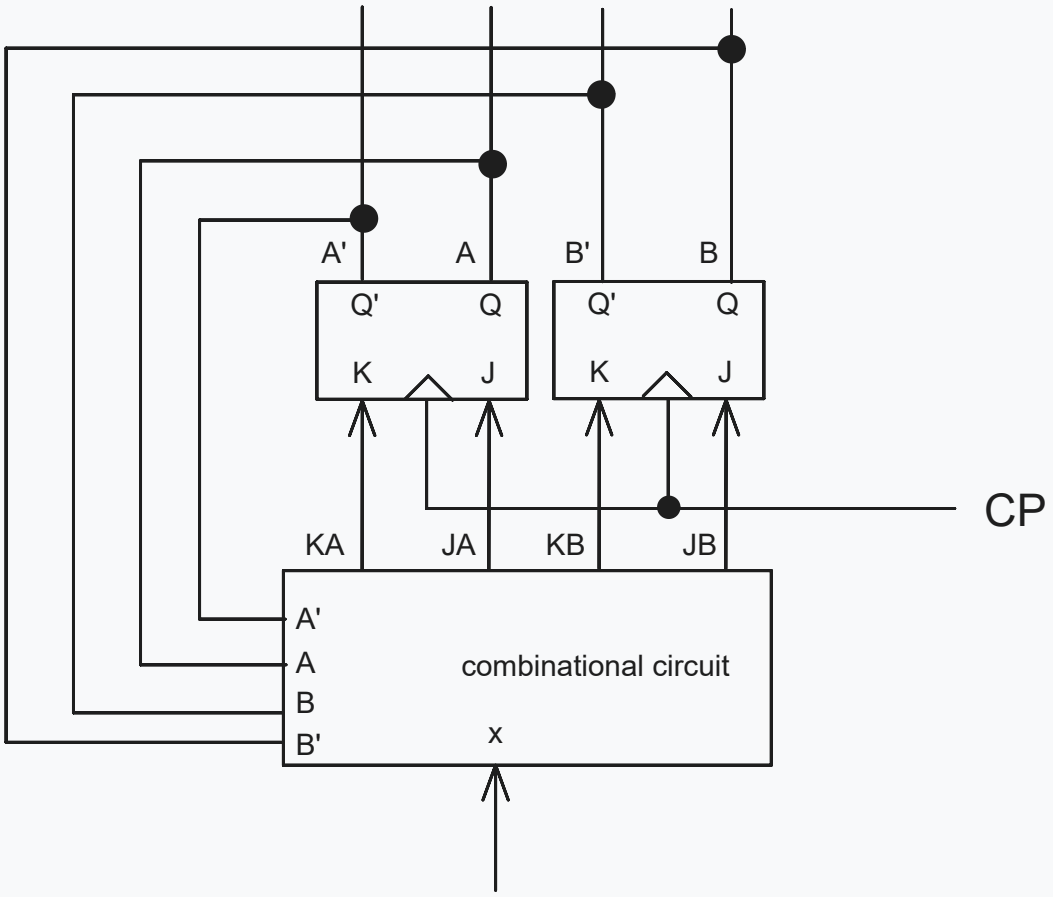
Using *JK* Flip-flop excitation table

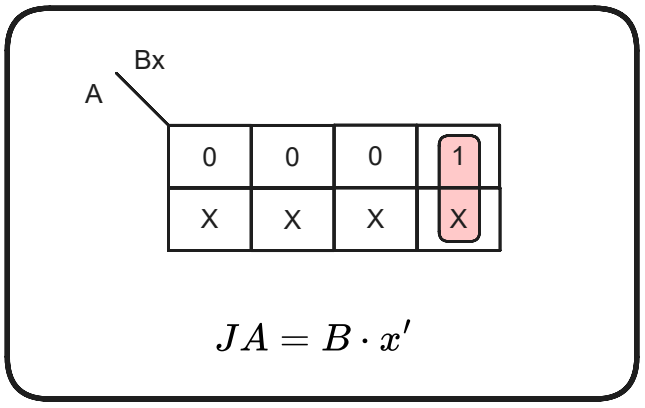
Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

We can get the state table:

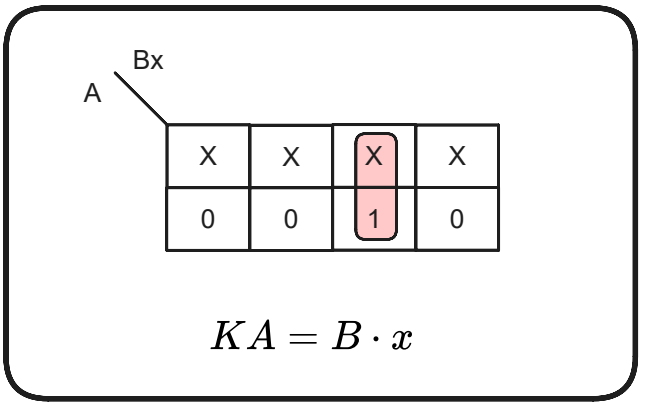
AB	x	A^+B^+	JA	KA	JB	KB
00	0	00	0	X	0	X
00	1	01	0	X	1	X
01	0	10	1	X	X	1

AB	x	A^+B^+	JA	KA	JB	KB
01	1	01	0	X	X	0
10	0	10	X	0	0	X
10	1	11	X	0	1	X
11	0	11	X	0	X	0
11	1	00	X	1	X	1

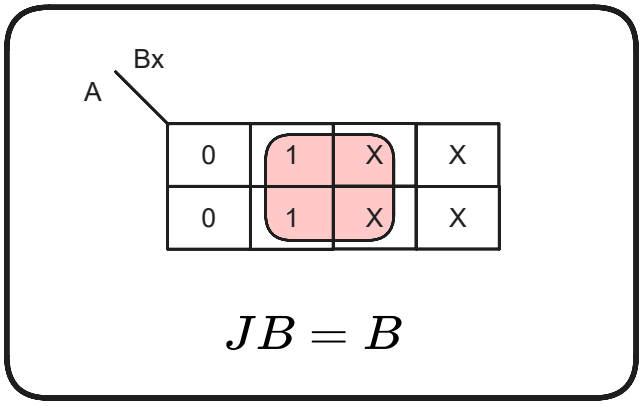




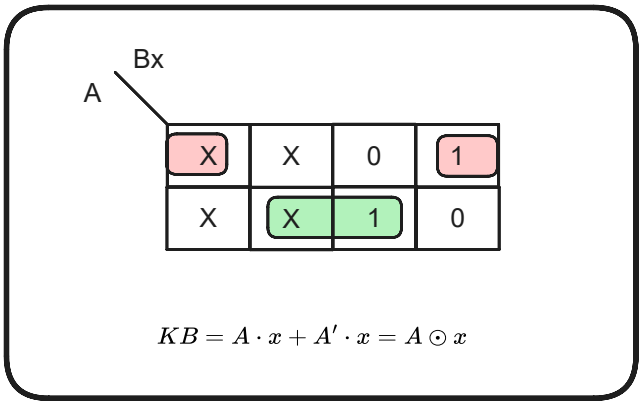
$$JA = B \cdot x'$$



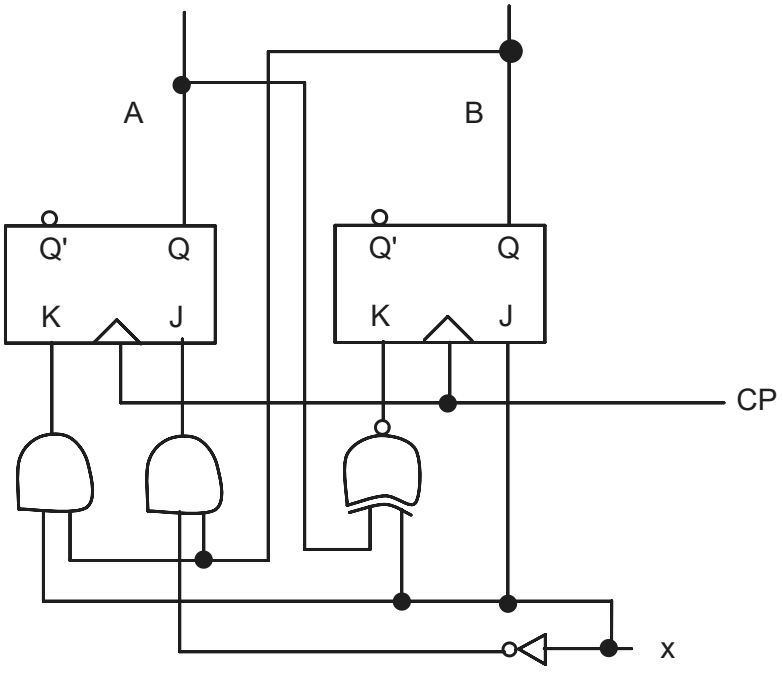
$$KA = B \cdot x$$



$$JB = B$$



$$KB = A \cdot x + A' \cdot x = A \odot x$$



Sink State

Sink state

A state that never moved out of itself to other states

Memory

Byte

8 bits

Word

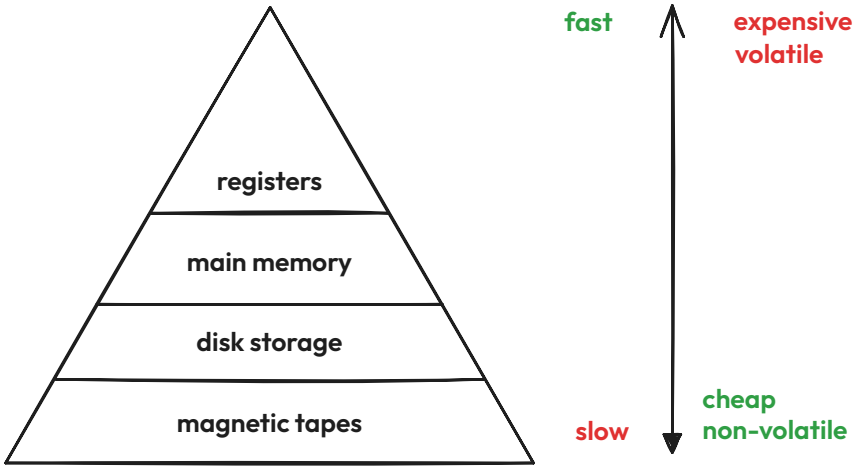
A unit of transfer between main memory and registers, usually size of register.

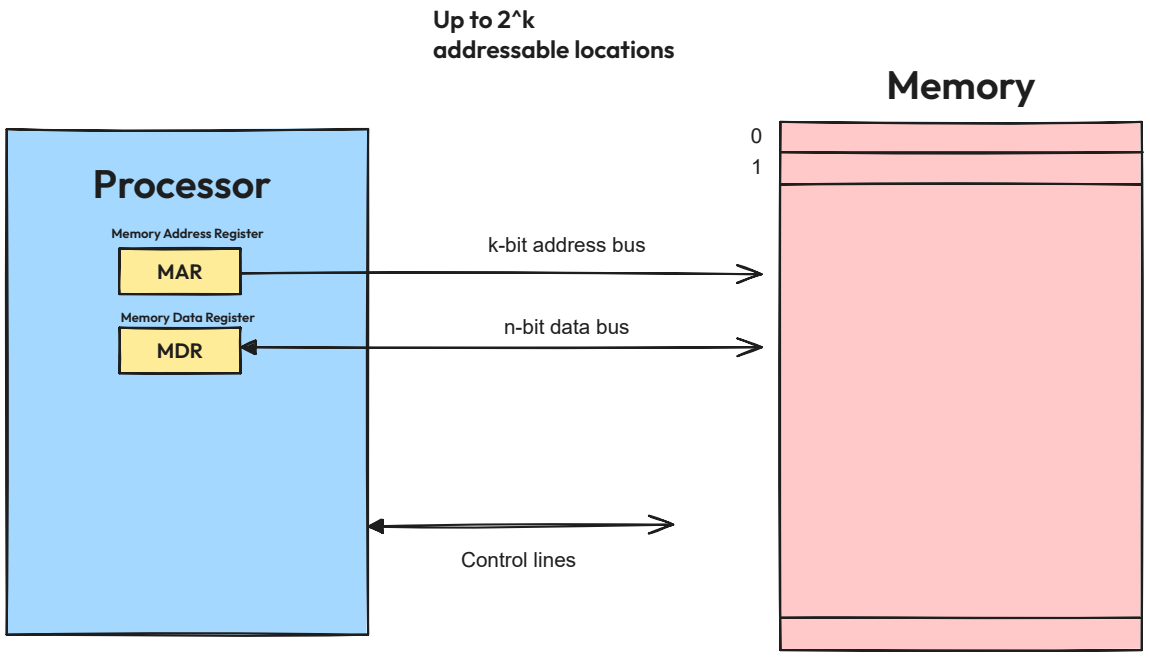
Common sizes

1 KB = 2^{10} bytes
1 MB = 2^{20} bytes
1 GB = 2^{30} bytes
1 TB = 2^{40} bytes

Desirable properties:

- fast access
- large capacity
- economical cost
- non-volatile





Memory unit stores binary information in groups of bits called words.

Data

Consists of n lines, for $n-$bit words.

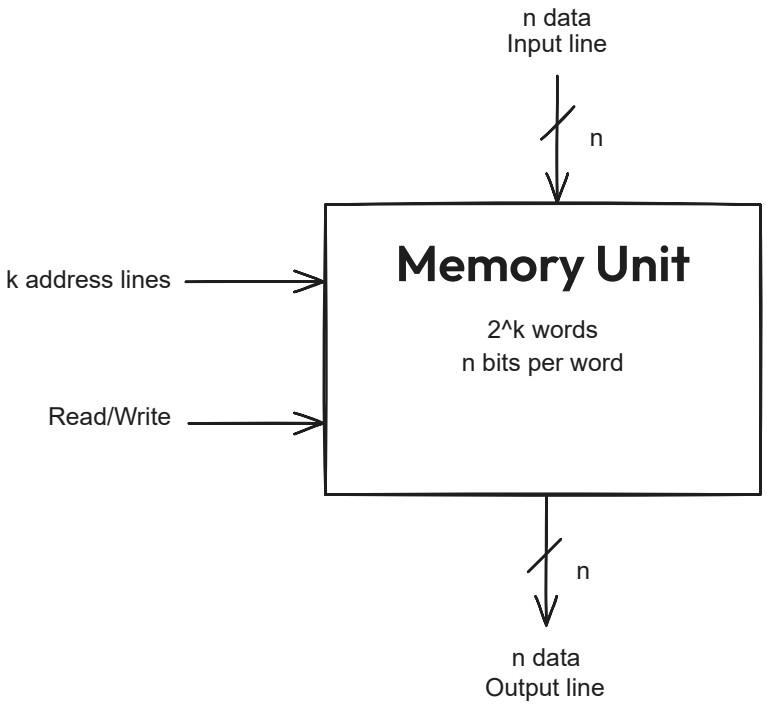
Data input lines provide information to be stored (written), and data output lines carry information out (read).

Address

Consists of k lines, specifying which word out of the 2^k available to be selected for reading or writing.

Control line

Specify direction of transfer of data (Read/Write)



Read/Write Operations

`Write`

Transfers address of desired word to address lines
Transfers data bits to be stored in memory to the data input line
Activates `Write` control line (set R/W to 0)

`Read`

Transfers address of desired word to address lines
Activates `Read` control line (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

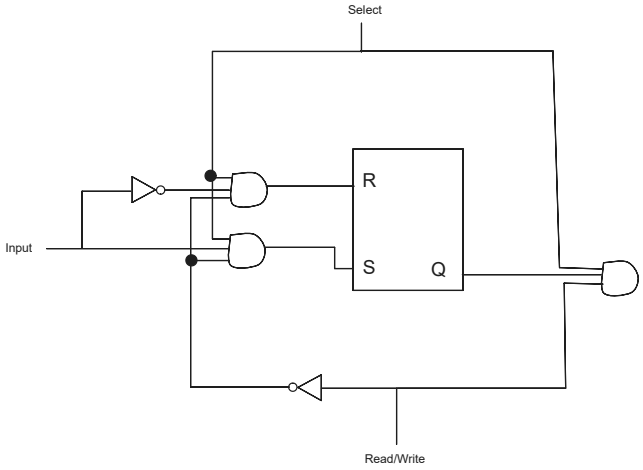
Memory Cell

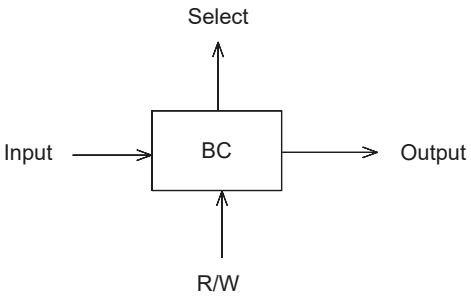
RAM

- Static RAM
  - flip-flops
- Dynamic RAM
  - capacitor charges to represent data

Difference

Dynamic RAM is simpler in circuitry but need to be constantly refreshed.





Pipelining

> **Motivation**
>
> Pipelining allows for a higher speedup over pure sequential processing.

Pipelining doesn't help latency for a single task, but helps the **throughput** of the entire workload - by having multiple tasks operate simultaneously using different resources.

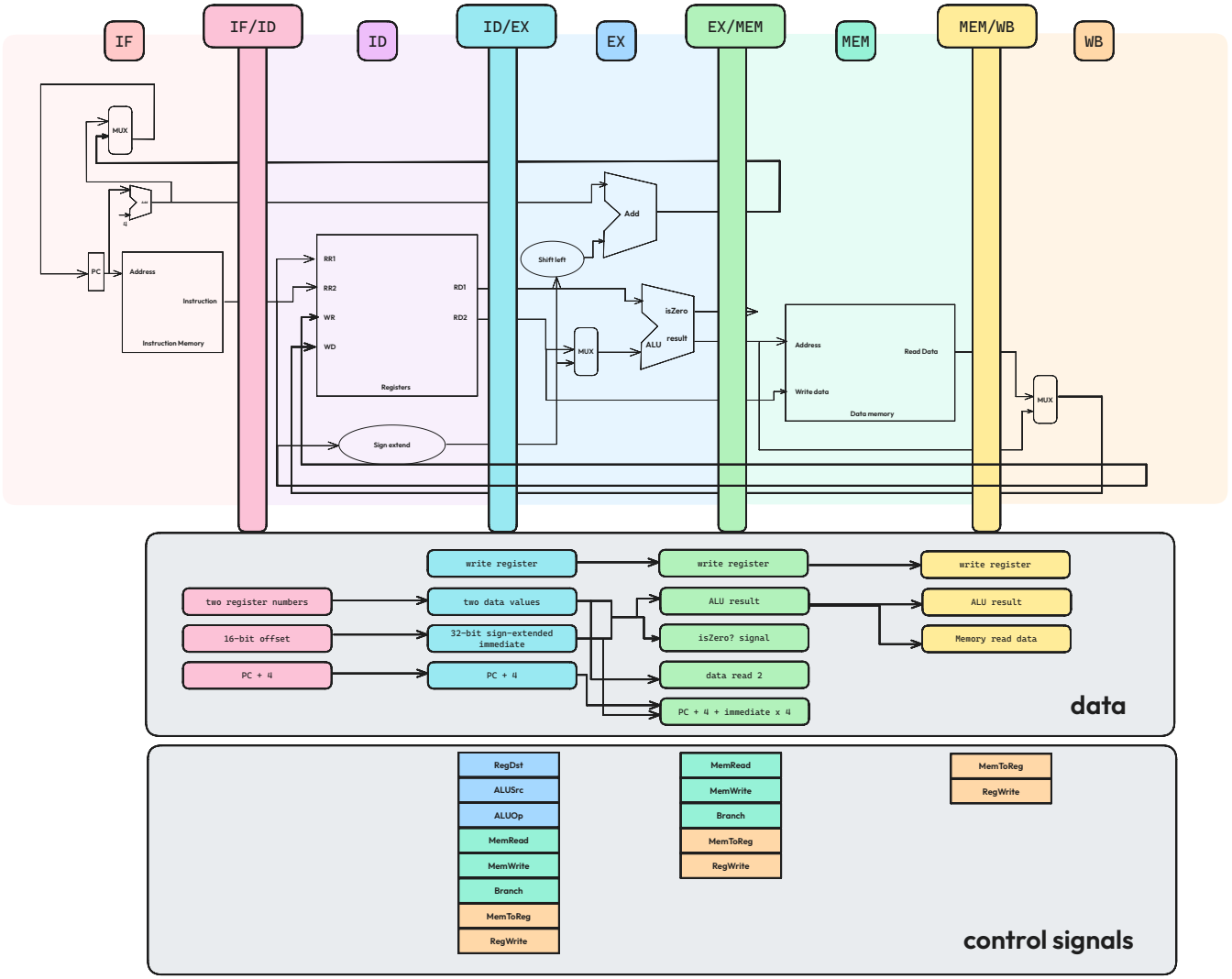
> **Possible delays:**
>
> - Pipeline rate, limited by slowest pipeline stage
> - Stall for dependencies

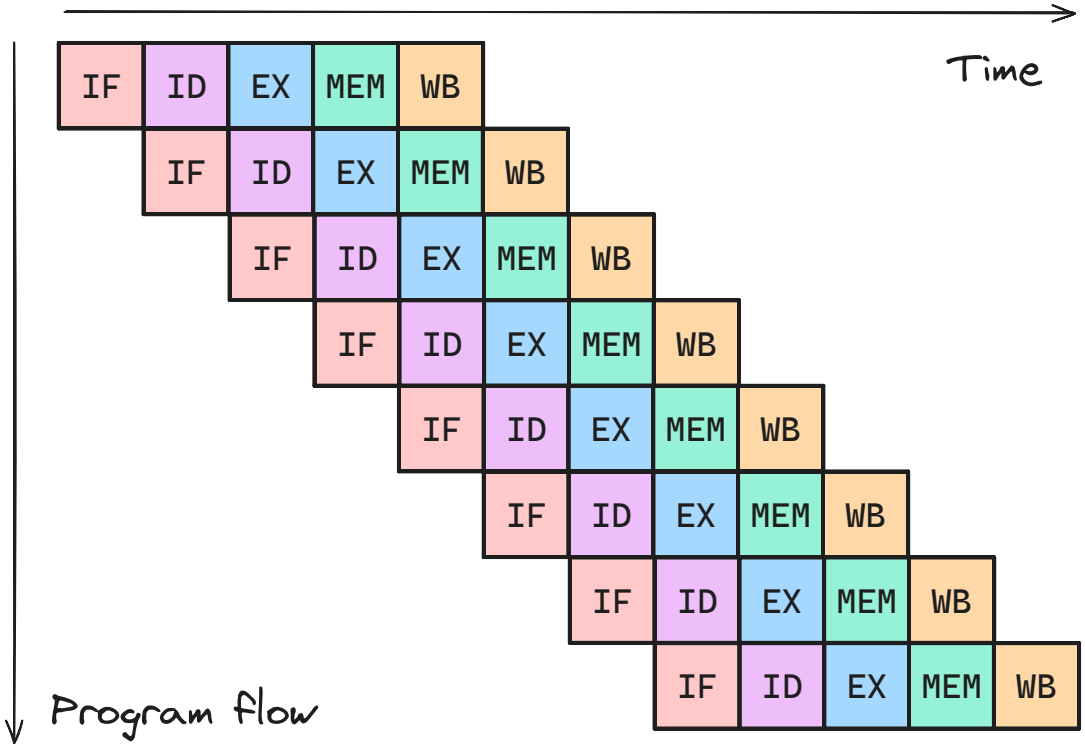
MIPS Pipeline Stages

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 memory
5. `WB` Write back the result into a register

(related to the Datapath stages)

Each execution stage will take 1 clock cycle.





Implementation

- One cycle per pipeline stage

made by zaidan sani.

- Data required for each stage needs to be stored separately.

There are two types of data that needs to be stored:

- data used by subsequent instructions
  - stored in programmer-visible elements
    - `PC`
    - registers
    - memory
- data used by same instruction in later pipeline stages
  - stored in pipeline registers
    - `IF/ID`
    - `ID/EX`
    - `EX/MEM`
    - `MEM/WB`

Stages

In the `ID` stage:

- `IF/ID` supplies:
  - register numbers for reading two registers
  - 16-bit offset to be sign-extended
  - `PC + 4`
- `ID/EX` receives:
  - data values read from register file
  - 32-bit immediate value
  - `PC + 4`
  - write register

Here, the signals are generated.

In the `EX` stage:

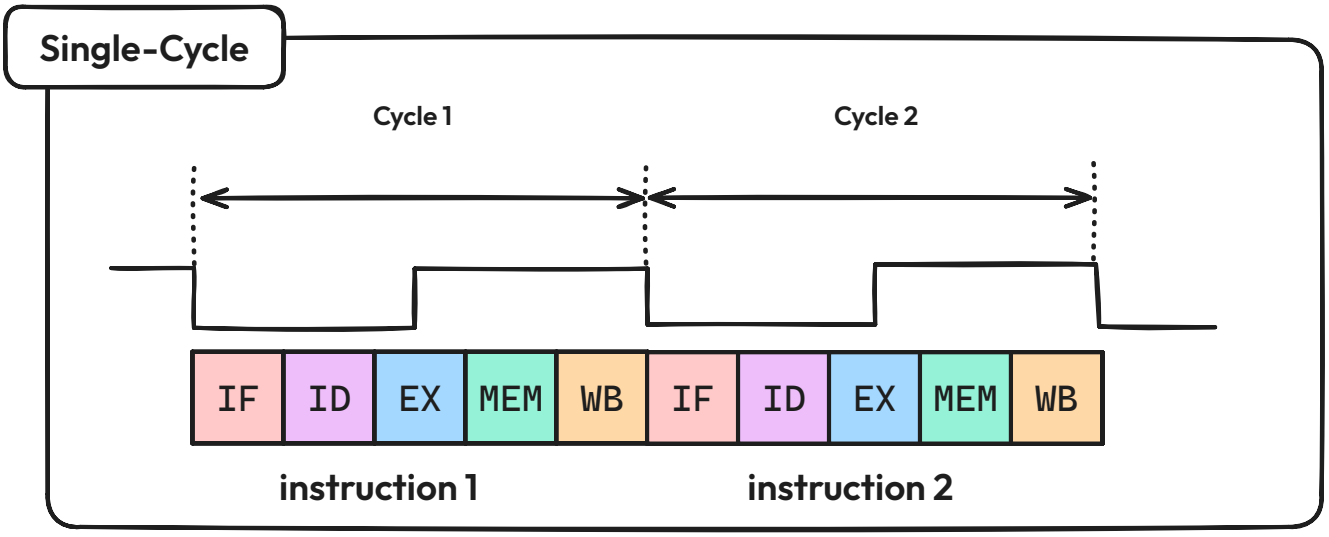
- `ID/EX` supplies:
  - data values read from register file
  - 32-bit immediate value
  - `PC + 4`
  - write register
  - signals
    - `RegDst`
    - `ALUSrc`
    - `ALUop`
    - `MemRead`
    - `MemWrite`
    - `Branch`
    - `MemToReg`
    - `RegWrite`
- `EX/MEM` receives:
  - `PC+4 + (IMM X 4)`
  - ALU result
  - `isZero?`
  - data read 2
  - write register
  - signals
    - `MemRead`
    - `MemWrite`

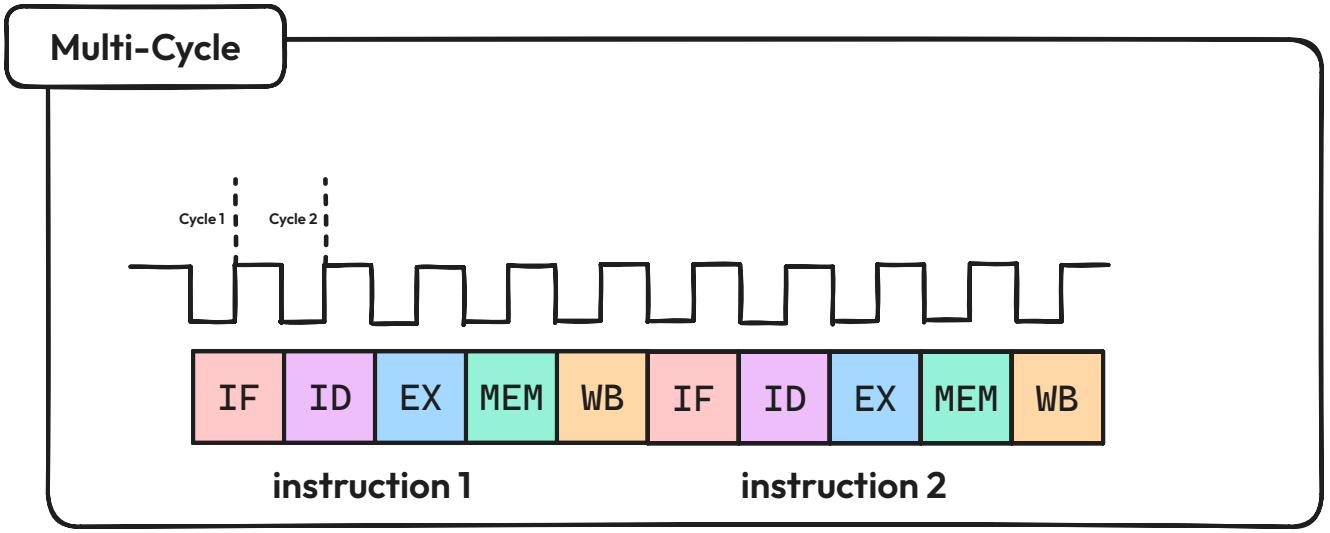
        - `Branch`
        - `MemToReg`
        - `RegWrite`

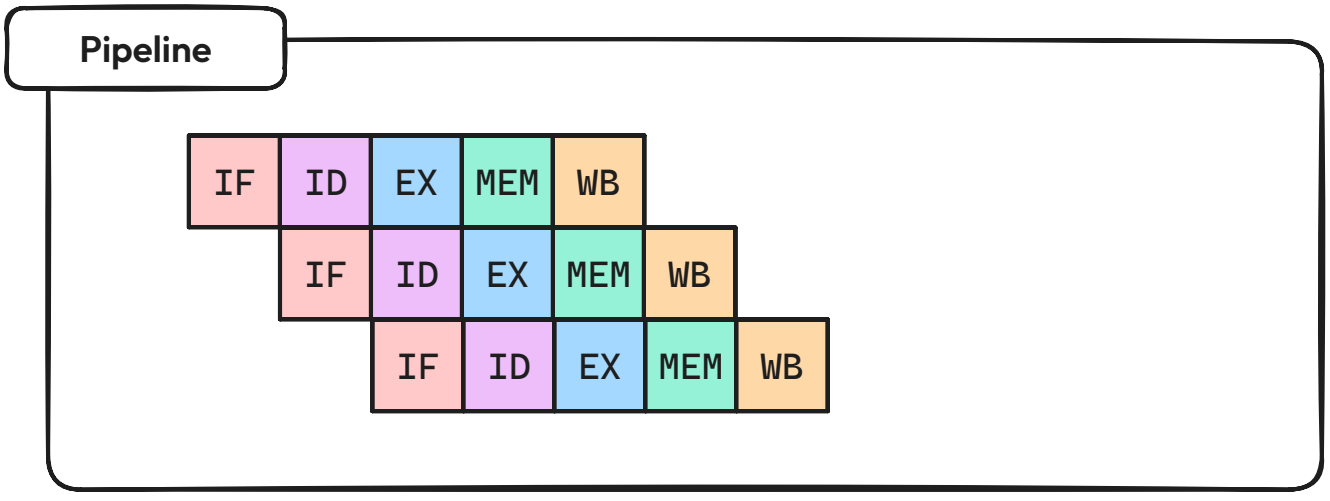
In the `WB` stage:

- `EX/MEM` supplies:
    - `PC+4 + (IMM X 4)`
    - ALU result
    - `isZero?`
    - data read 2
    - write register
    - signals
        - `MemRead`
        - `MemWrite`
        - `Branch`
        - `MemToReg`
        - `RegWrite`
- `MEM/WB` receives:
    - ALU result
    - memory read data
    - write register
    - signals
        - `MemToReg`
        - `RegWrite`

Performance comparison







Single-cycle processor

Cycle time:

$$CT_{seq} = max(\sum_{k=1}^{N} T_k)$$

where T_k = time for operation in stage k, N = number of stages.

Execution time:

$$Time_{seq} = I \times CT_{seq}$$

> Since all the stages are in one single cycle, the time for operation is the maximum time of an instruction as a bottleneck.

made by zaidan sani.

Multicycle processor

Cycle time:

$$CT_{multi} = max(T_k)$$

where T_k is the time for operation in stage k.

> ✎ **The cycle time is based on the stage with the most time taken for it.**

Execution time:

$$Time_{multi} = I \times AverageCPI \times CT_{multi}$$

> ✎ **Average CPI needed because each instruction takes different number of cycles.**

Pipelining processor

Cycle time:

$$CT_{pipeline} = max(T_k) + T_d$$

where T_k is the time for operation in stage k, and T_d refers to the overhead for pipelining.

> ✎ **Since there is a pipeline register, there is overhead here.**

Execution time:

$$Time_{pipeline} = (I + N - 1) \times CT_{pipeline}$$

where $(I + N - 1)$ refers to the cycles needed for I instructions.

Ideal speedup

> ✎ **Assumptions for ideal case**
>
> - every stage takes same amount of time
> - no pipeline overhead
> - $I > N$ (number of instructions significantly larger than number of stages)

$$\begin{aligned} Speedup_{pipeline} &= \frac{Time_{seq}}{Time_{pipeline}} \\ &= I \times \frac{\sum_{k=1}^{N} T_k}{(I + N - 1) \times (max(T_k) + T_d)} \\ &= \frac{I \times N \times T_1}{(I + N - 1) \times T_1} \\ &\approx \frac{I \times N \times T_1}{I \times T_1} \\ &= N \end{aligned}$$

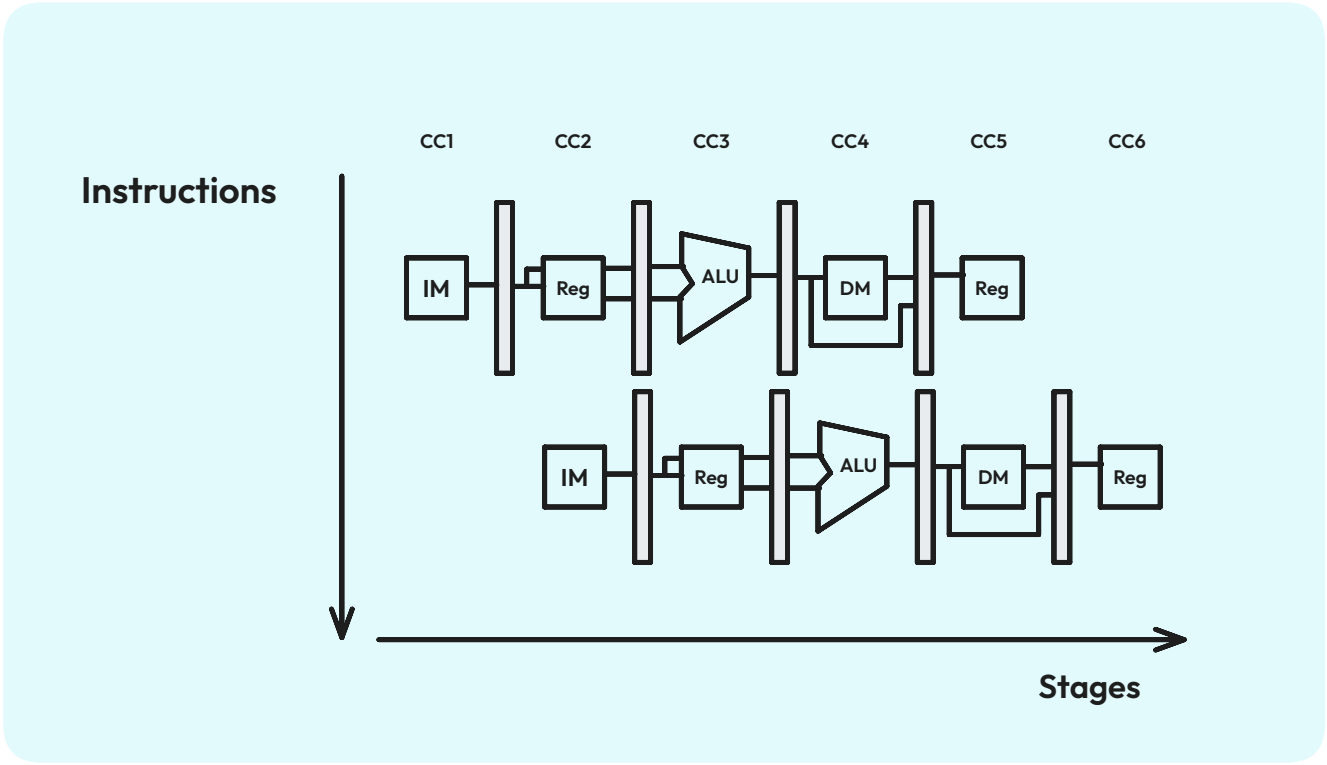
> ✓ **Pipeline processor can gain N times speedup where N is the number of pipeline stages.**

Hazards

> ⚠ **Pipeline hazards**
>
> Hazards that prevent next instruction from immediately following previous instruction

- Structural hazards

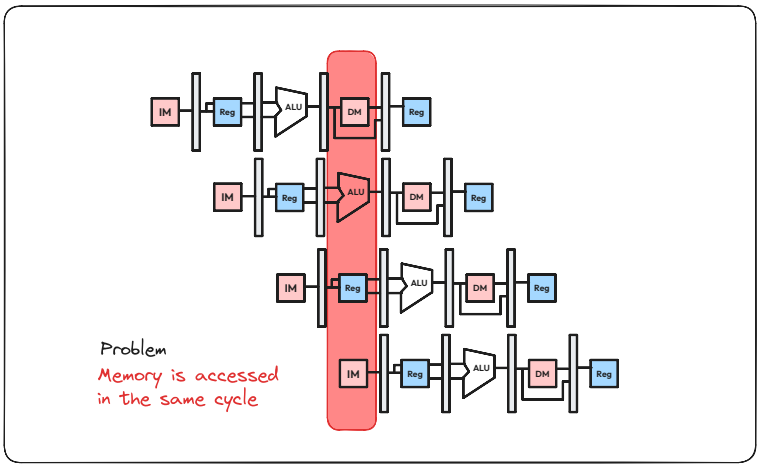
- Simultaneous use of a hardware resource
- Data hazards
  - Data dependencies between instructions
- Control hazards
  - Change in program flow



Structural Hazards

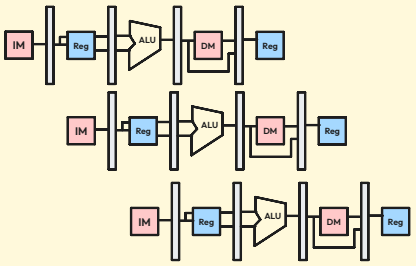
If there is only one single memory module

There might be a conflict if two instructions access memory in the same cycle.

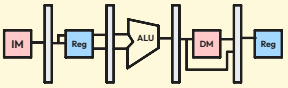


SOLUTIONS

Stall pipeline

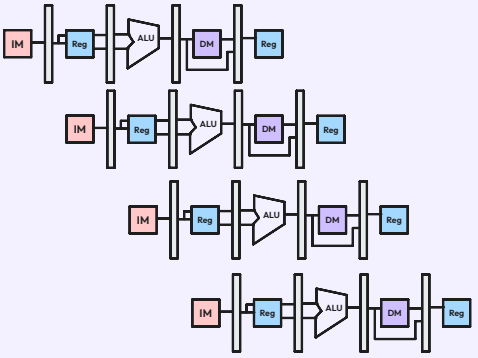


STALL



Delay 4th instruction
by 1 cycle

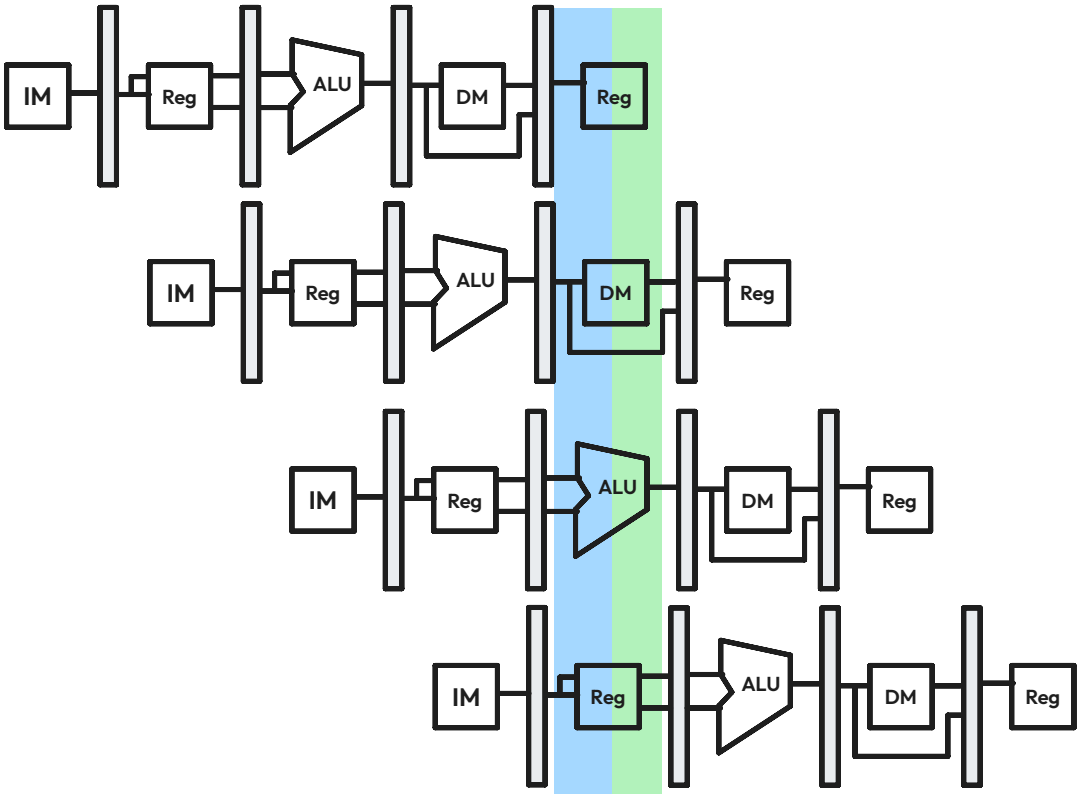
Separate memory



Separate the memory into
Data Memory and Instruction Memory

Conflict for registers?

Registers are very fast memory, allowing for the cycle to be split into half, one for writing and one for reading.



Instruction Dependencies

Instruction Dependencies

Instructions can have relationships that prevent pipeline execution.

Data dependency

When different instructions access the same register

Control dependency

An instruction `j` is control dependent on `i` if `i` controls whether or not `j` executes

- `i` is generally a branch instruction

`RAW` : Read-After-Write

`RAW`

Occurs when a later instruction reads from destination reigster written by an earlier instruction
(True data dependency)

```
i1: add $1, $2, $3
i2: sub $4, $1, $5
```

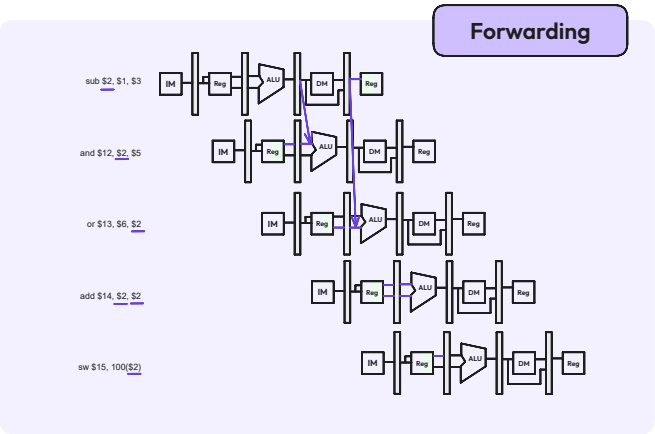
Stale result

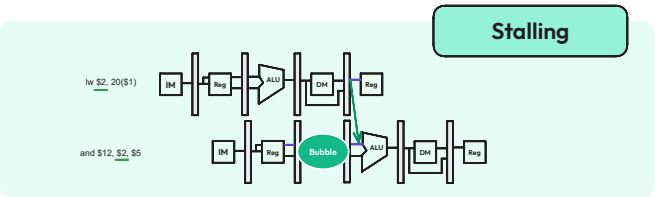
An old result.

If `i2` reads register `$1` before `i1` writes back, `i2` gets a stale result.

There are two possible solutions:

- Forwarding
  - Forward the result to any later instructions before reflecting it in register file and replace the data read from register file.
- Stalling
  - Useful when forwarding does not work (data is not even loaded yet).





`WAR` : Write-After-Read

Does not cause pipeline hazards.

Only affects processor only when instructions are executed out of program order.

`WAW` : Write-After-Write

Does not cause pipeline hazards

Only affects processor only when instructions are executed out of program order.

Control Dependency

Why can't we just stall the pipeline?

Introduces large clock cycle delay - branching being common, a 3-cycle stall penalty is too heavy.

To minimise the control hazard penalty:

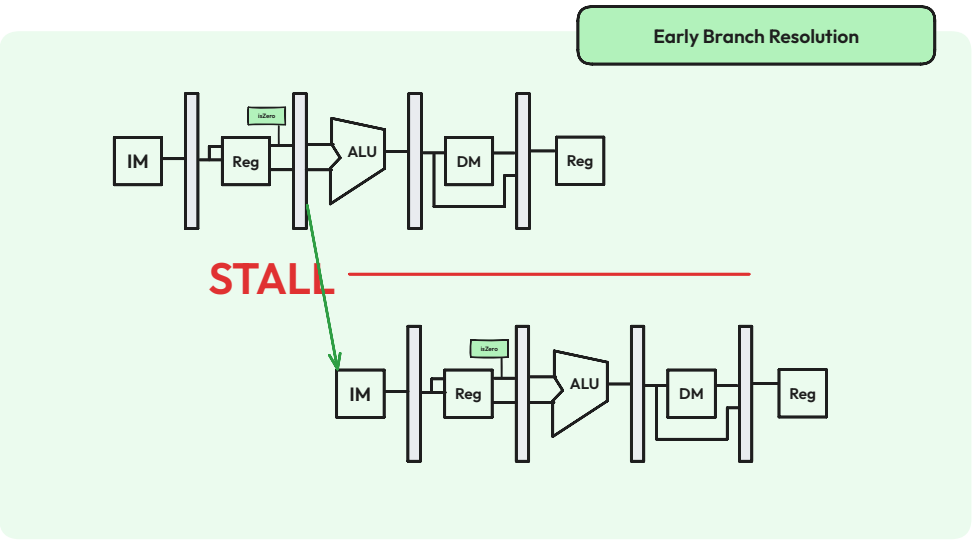
- Early branch resolution
  - Move branch decision calculation to earlier pipeline stage
- Branch prediction
  - Guess outcome before produced

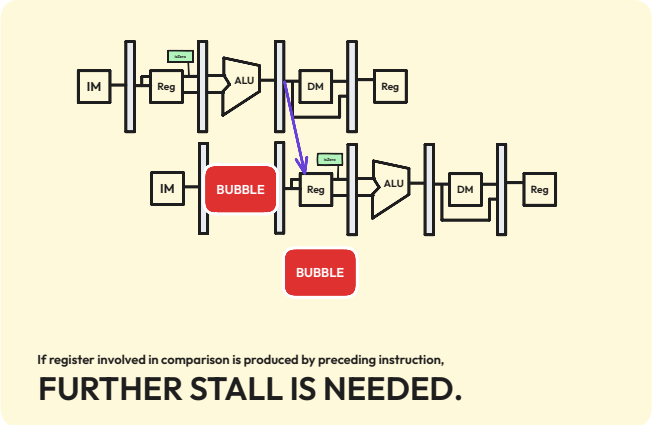
- Delayed branching
  - Do something useful while waiting for the outcome

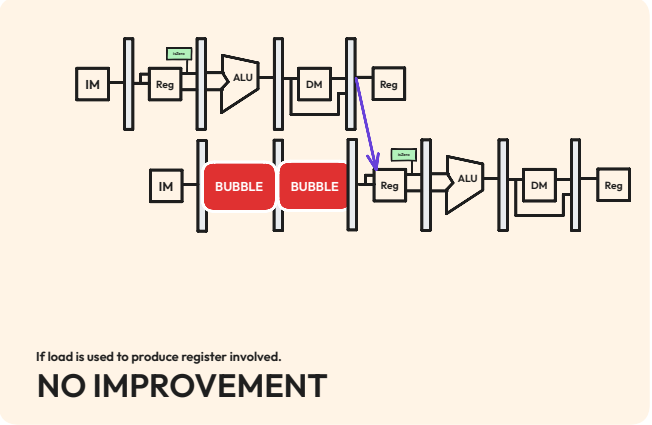
Early branch resolution

Make the decision in `ID` stage instead of `MEM`.

Move register comparison to ID stage.







Problems

If instruction producing value into comparison register is before, extra clock cycles are needed
- 1 more if `load` instruction
- results in no performance improvement.

Branch Prediction

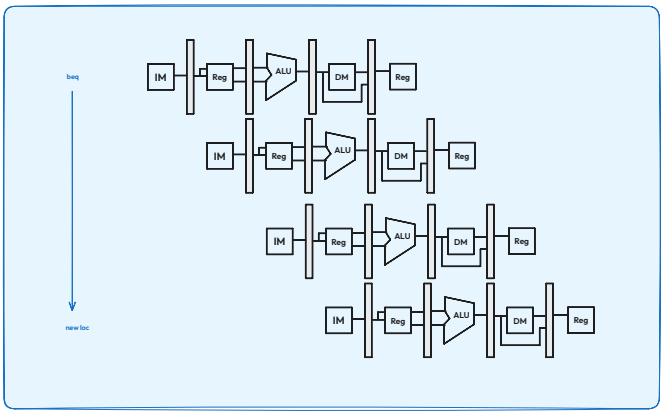
There are other branch prediction schemes. Only one is covered here.

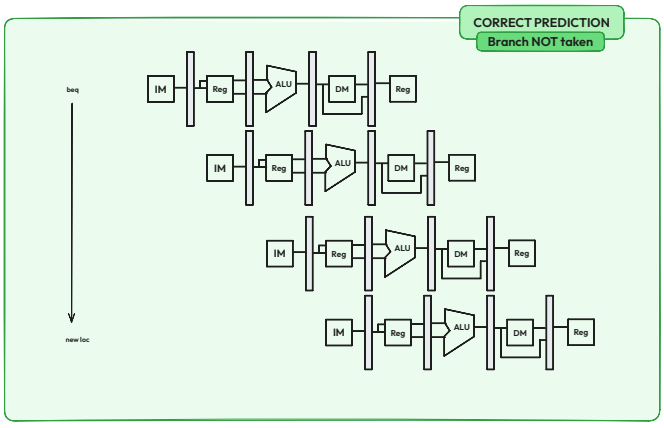
Simple prediction: All braNches are assumed to not be taken.

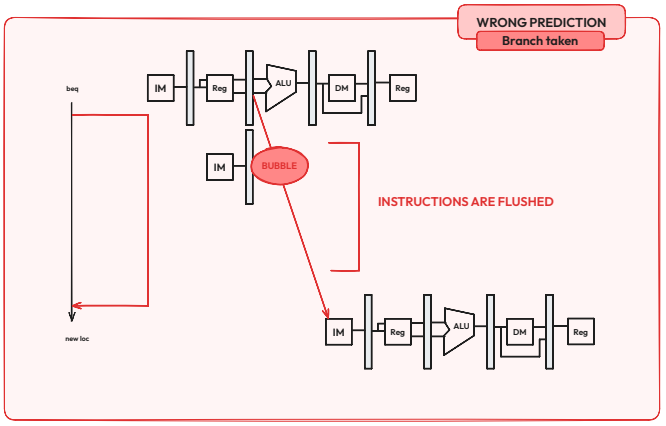
After outcome of branch:

- Not taken
  - No pipeline stall
- Taken

- Wrong instructions in pipeline
- Successor instructions are flushed







Delayed Branch

Motivation

Branch outcome takes X cycles to be known. Thus, these cycles can be used to execute non-control dependent instructions into the X slot following a branch. These instructions are executed regardless of the branch outcome.

```
or $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11

Exit:
```

Since the `or` instruction does not affect the remaining instructions, and gets executed regardless, it can be moved:

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
xor $10, $1, $11

Exit:
```

Possible scenarios

✓ Best case scenario

Instruction preceding branch which can be moved into delayed slot

✕ Worst case scenario

Instruction cannot be found.

- Add `nop` instruction

Multiple Issue Processors

This is optional reading.

Multiple issue processors

Multiple instructions in every pipeline stage

Static multiple issue

EPIC or VLIW

- Compiler specifies the set of instructions that execute together in given clock cycle.
- Simple hardware, complex compiler.

Dynamic multiple issue

Superscalar processor - dominant design of modern processes

- Hardware decides which instructions to execute together.
- Complex hardware, simpler compiler

2-wide superscalar pipeline

Fetch and dispatch two instructions at a time, allowing for up to 2 instructions to complete per cycle.

Cache

Summary

	Direct-Mapped	Set Associative	Fully Associative
Block Placement	Only one block, defined by index	Any one of N blocks defined by index	Any cache block
Block Identification	Tag match with only one block	Tag match for all blocks within set	Tag match for all blocks within cache
Block Replacement	Choose block based on index (no choice)	Choose block based on replacement policy	Choose block based on replacement policy

Context

If operands are in memory, it has to be loaded into the registers of the processor, and then operated, and stored back into memory.

Memory Technology

DDR SDRAM

Double Data Rate Synchronous Dynamic Random Access Memory

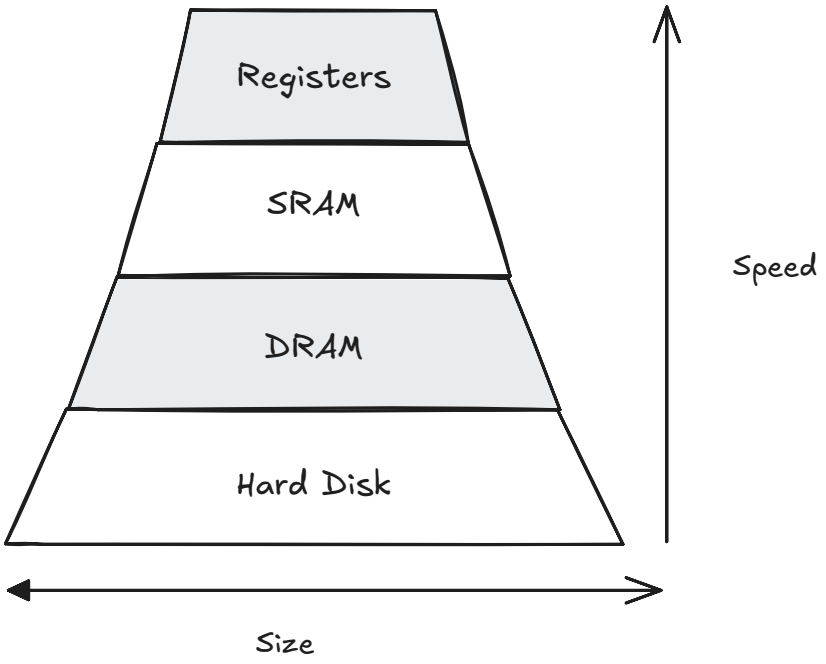
Motivation

Requirement: Big and fast memory

Key concept

Hierarchy of memory technologies should be used:

- Small but fast near CPU
- Large but slow farther away from CPU (for cost)



Cache

Keep frequently and recently used data in smaller but faster memory

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 soon

Spatial locality

If an item is referenced, nearby items will tend to be referenced soon.

Working Set

Working set

Set of locations accessed during Δt

Aim

Capture working set and keep it in memory closest to CPU

Memory Access

To make slow main memory appear faster:

- Cache
- Hardware management

To make small main memory appear bigger:

- Virtual memory
- OS managed

Terminology

Hit

Data is in cache

Hit rate

Fraction of memory that is in cache

Hit time

Time to access cache

Miss

Data is not in cache

Miss rate

Fraction of memory not in cache $(1 - Hitrate)$

Miss penalty

Time to replace cache block + hit time

By definition $>$ hit time

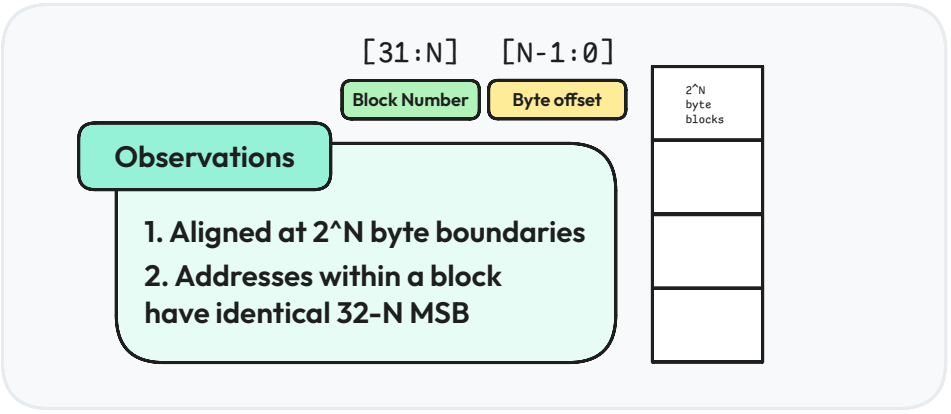
Average access time:

$$\text{Hit rate} \times \text{Hit time} + (1 - \text{Hit rate}) \times \text{Miss penalty}$$

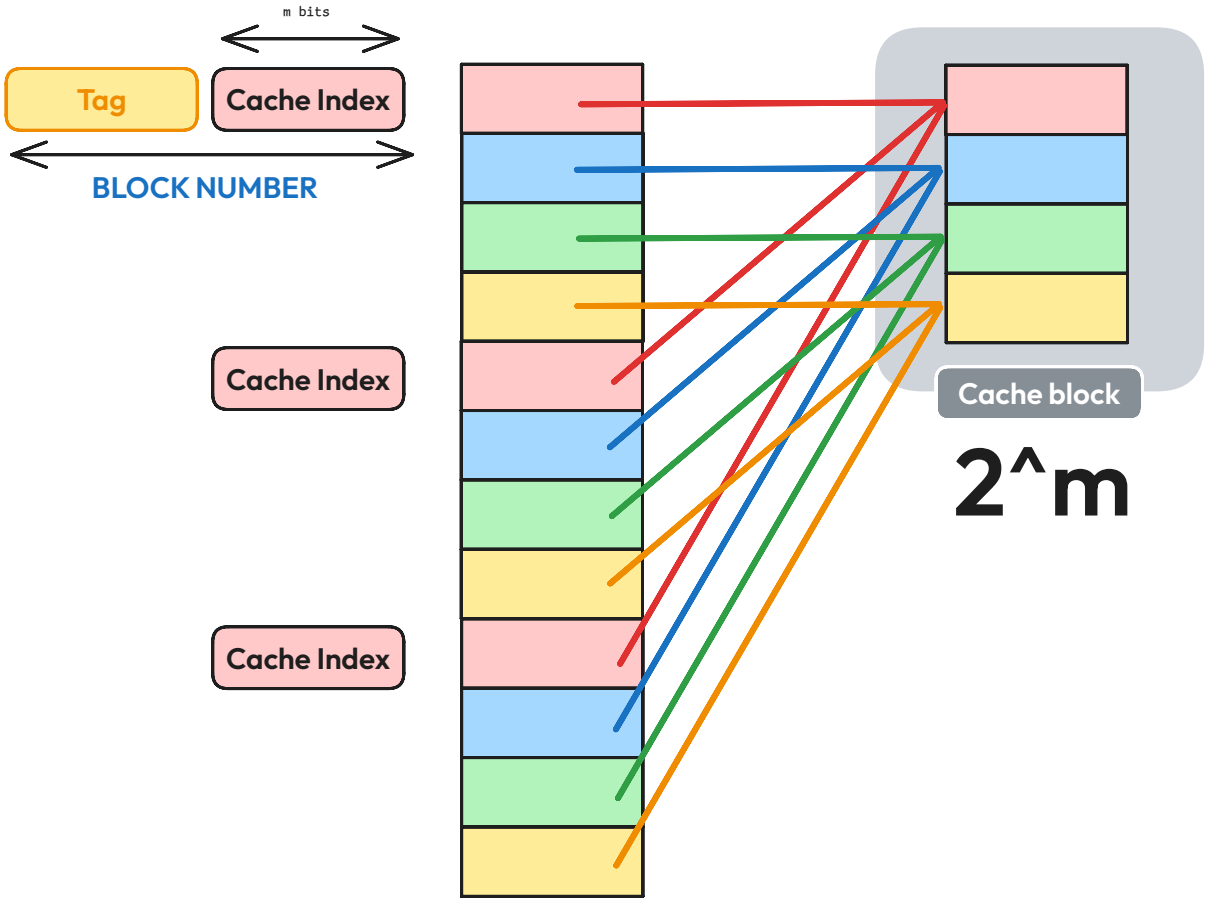
Memory-Cache Mapping

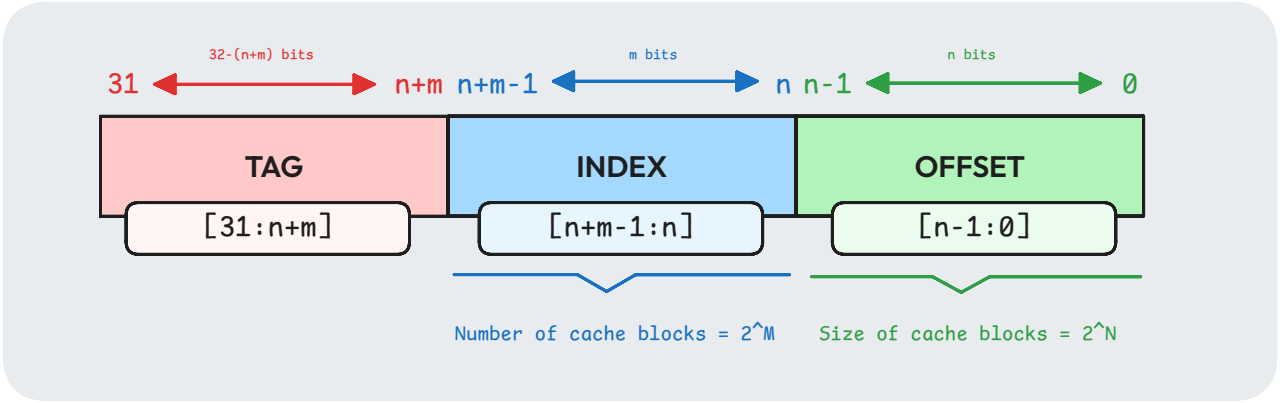
Cache block/line

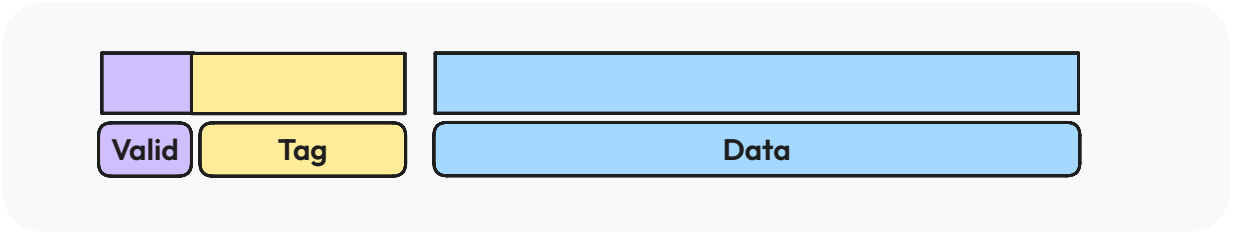
Unit of transfer between memory and cache

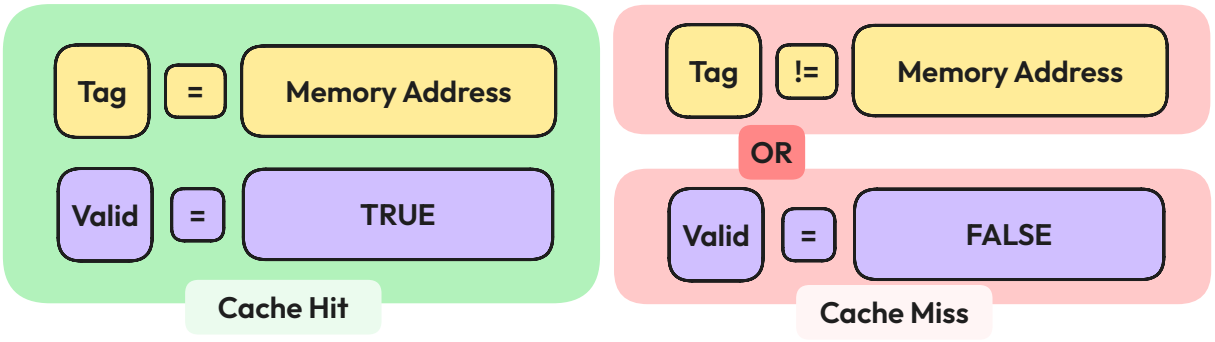


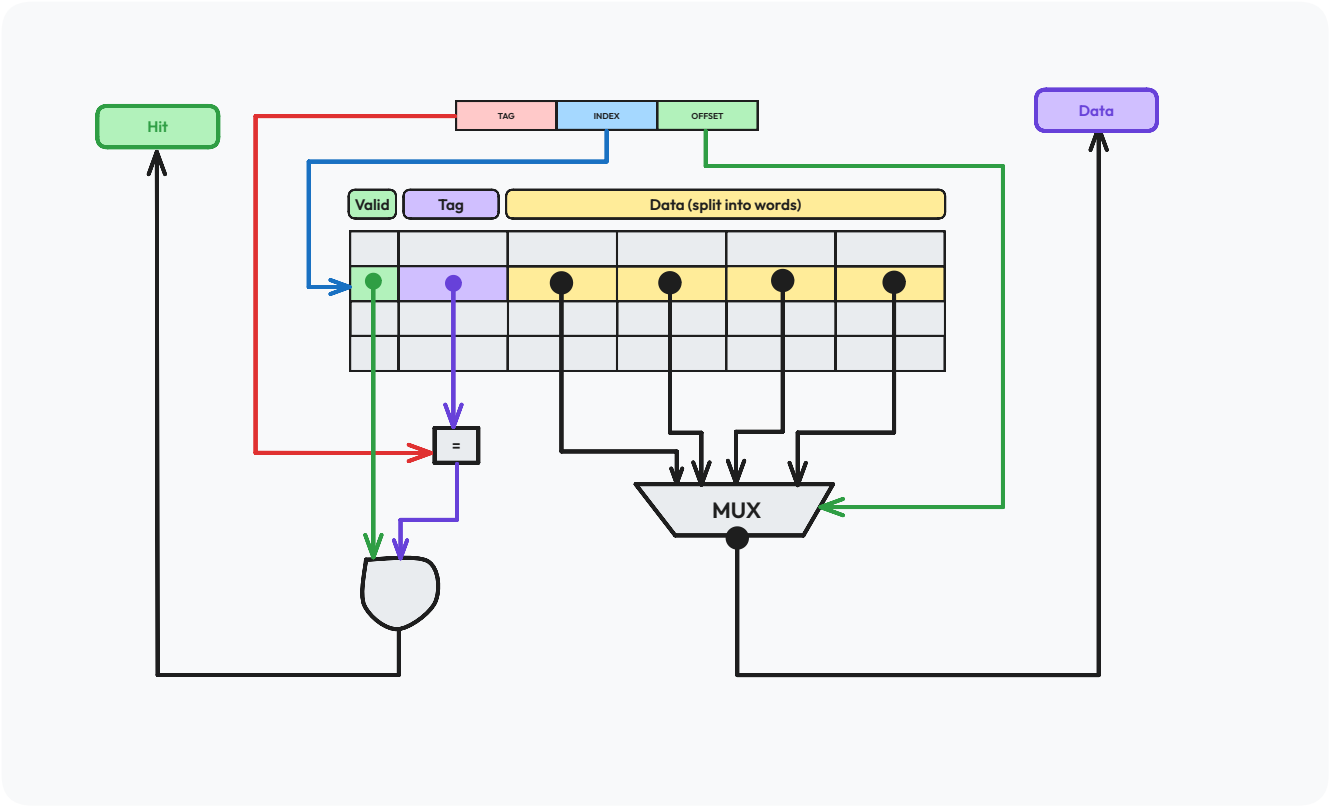
Direct Mapped Cache



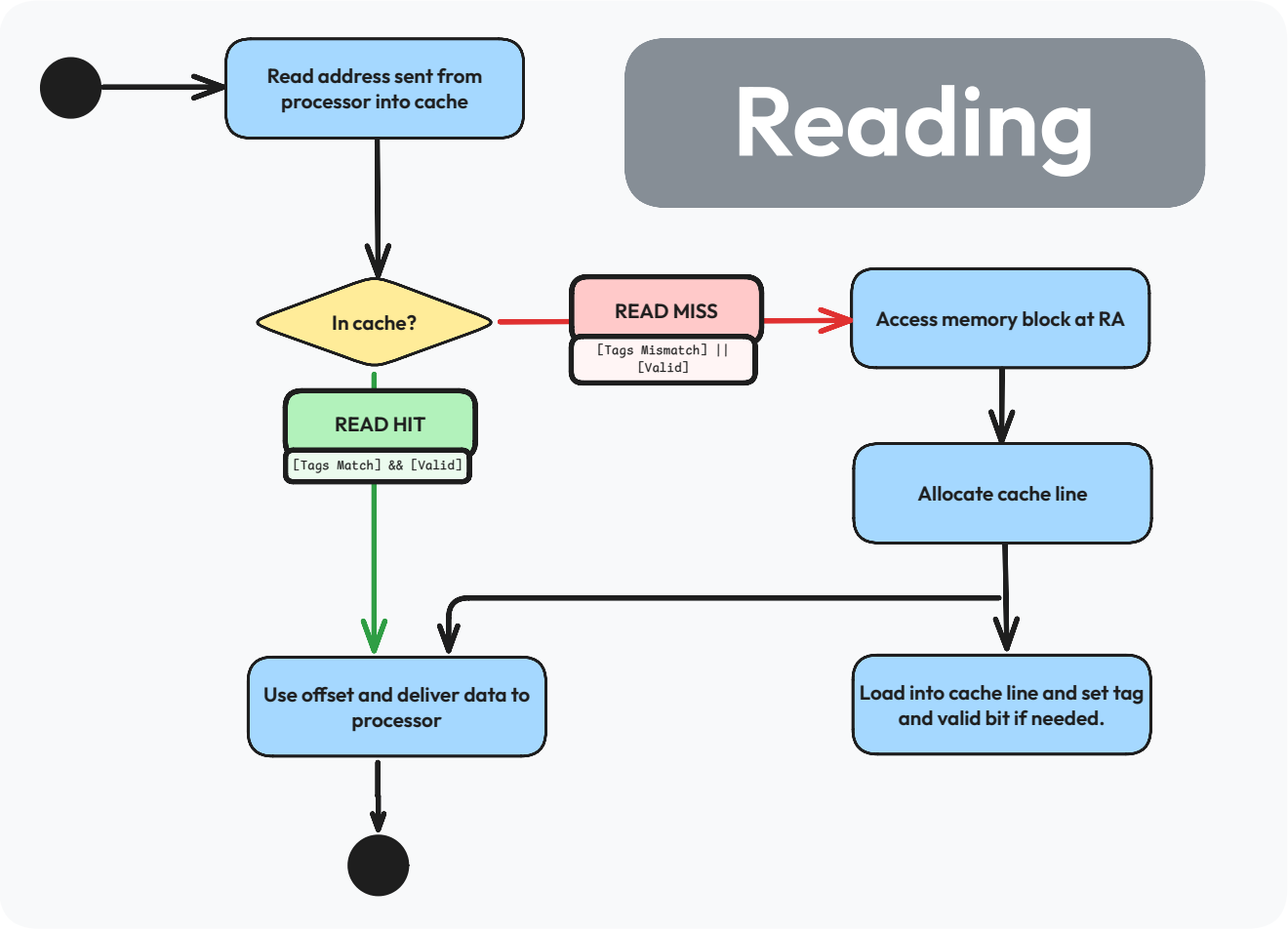








Reading Data



Cache Misses	
Compulsory misses (cold start/first reference)	Occurs at first access to a block, as block must be brought into cache.
Conflict misses (collision/interference)	Occurs in direct mapped/set associative, when several blocks are mapped to the same block/set
Capacity misses	Occurs when blocks are discarded from cache as cache cannot contain all blocks

Writing Policy

Motivation

When writing data, the modified data is only in cache, but not in memory.

Solutions:

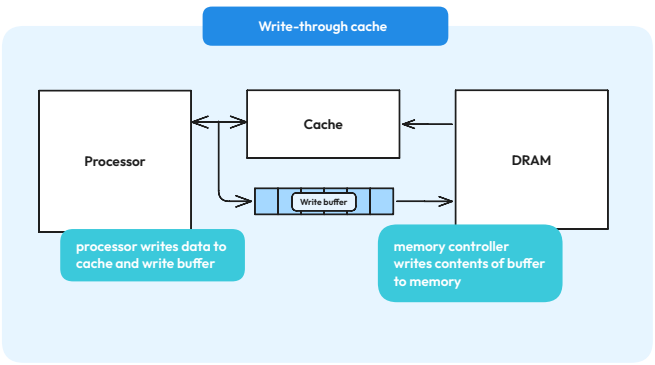
1. Write-through cache
2. Write-back cache

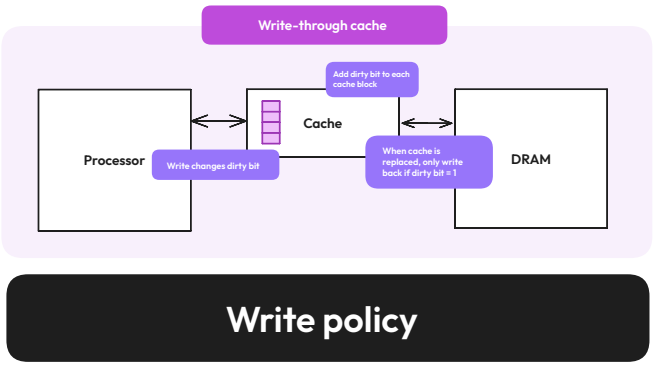
Write-through cache

Write data both to cache and to main memory

Write-back cache

Only writes to cache during write operation, and only writes to main memory when cache block is evicted.





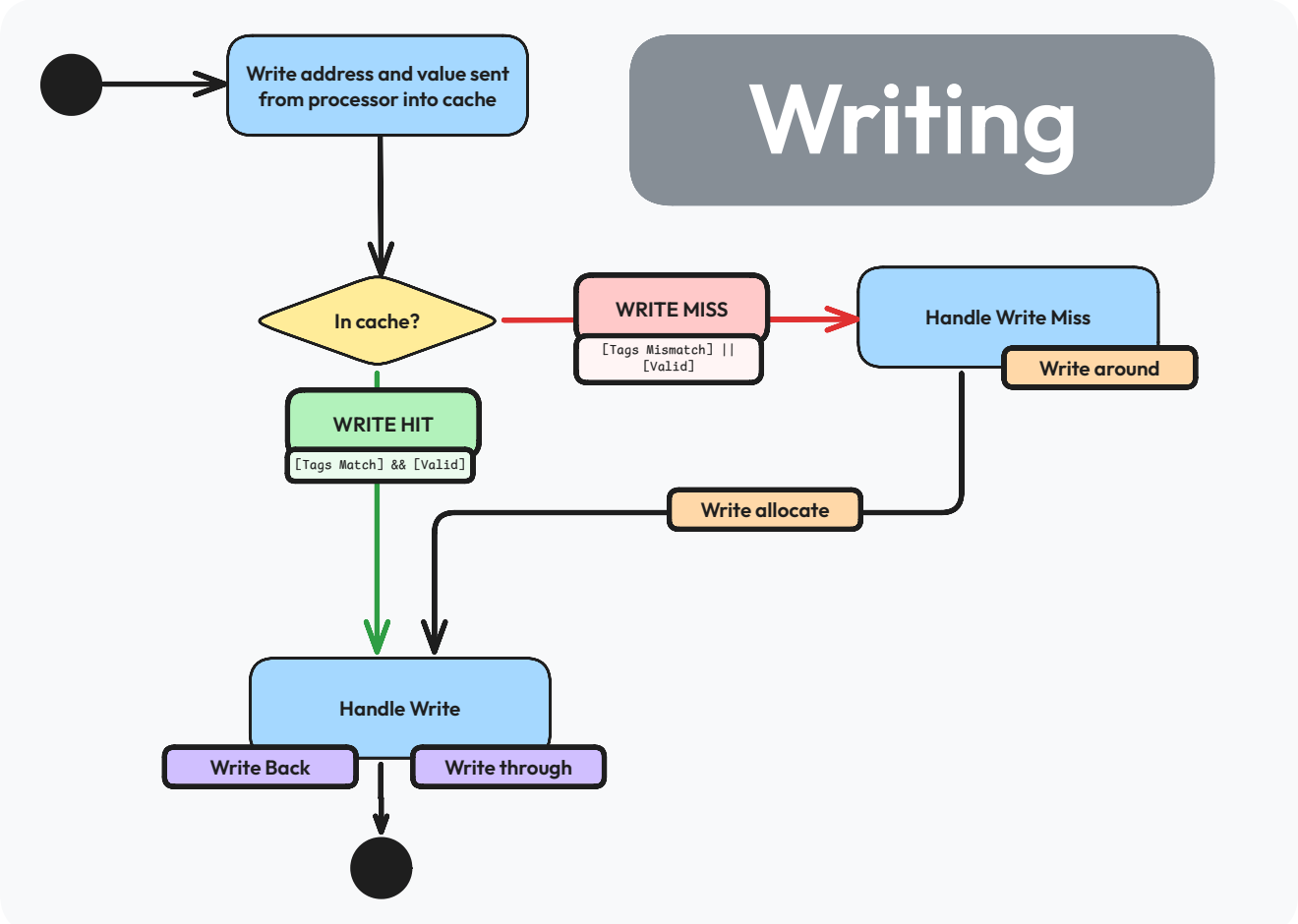
Handling Cache Misses

Write allocate

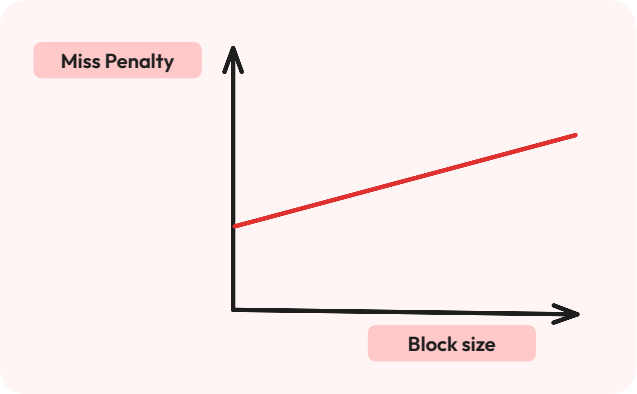
- load complete block into cache
- change only required word in cache
- write to main memory (based on write policy)

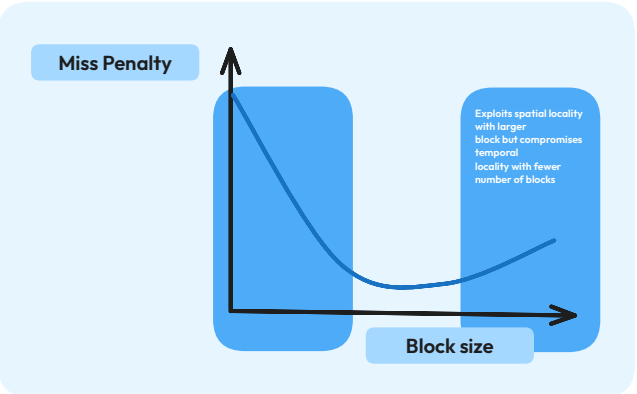
Write around

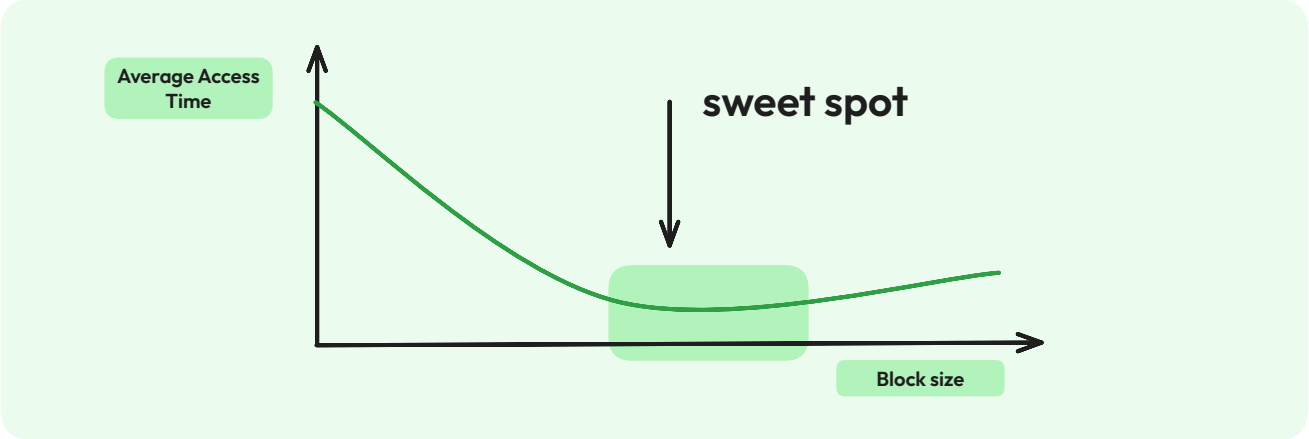
- write directly to main memory



Block Size Trade-offs







Set-Associative Cache

Motivation

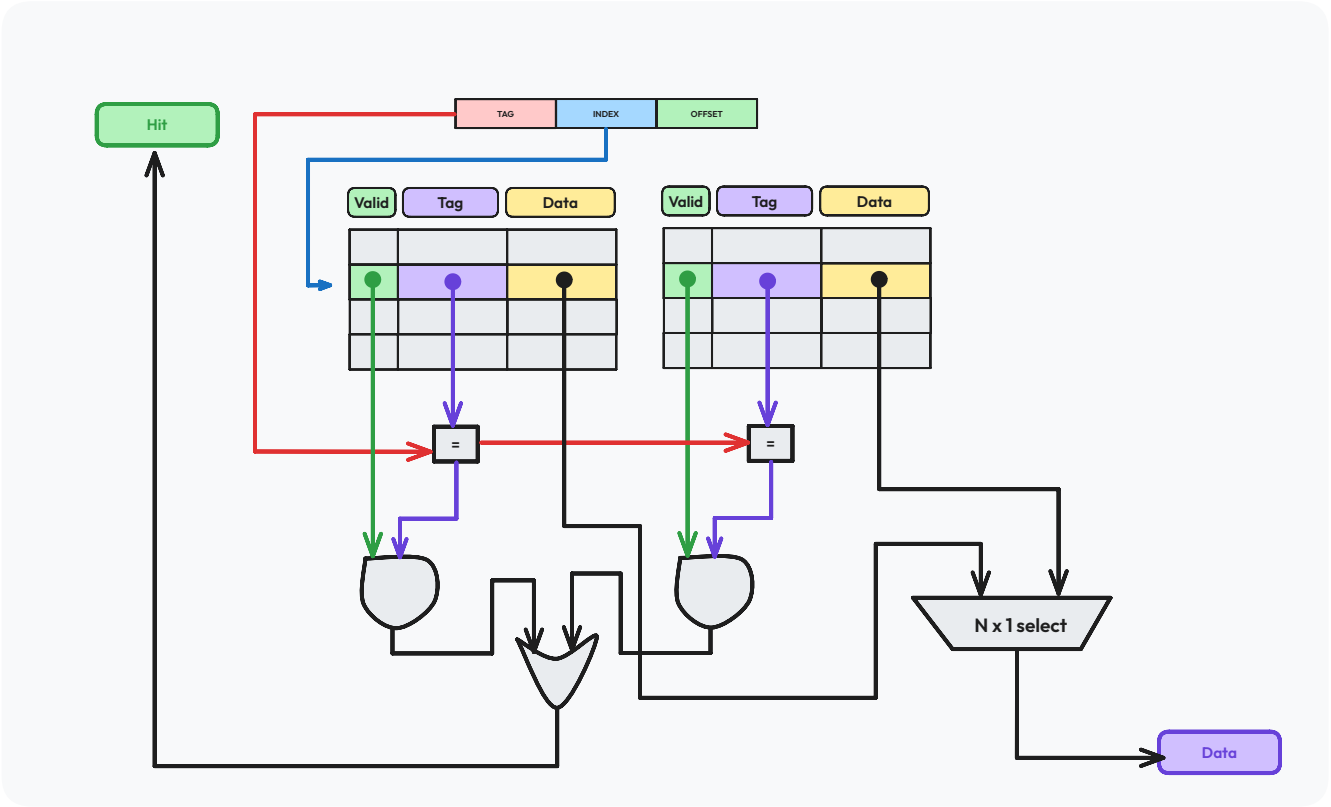
Address conflict misses.

$N-$way set associative cache

A memory block can be placed in a fixed number N of locations in the cache, where $N > 1$.

Effective idea:

- instead of mapping to a cache block, it maps to a unique set of cache blocks where it can be placed in any of the N cache blocks in that set.
- requires searching of both to look for memory block



✓ Advantage of associativity

A direct-mapped cache of size N has the same miss rate of a 2-way set associative cache of size $N/2$.

Fully-Associative Cache

Fully-associative cache

Can be placed in any location in cache

Block number serves as tag in FA cache.

✓ No conflict misses

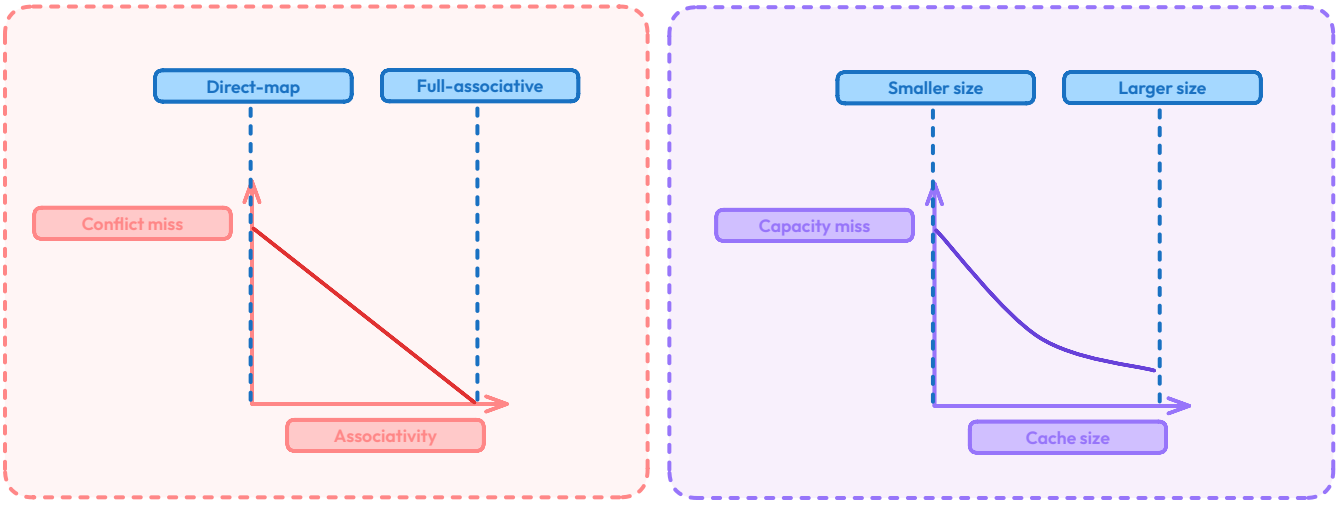
No conflict misses happens since data can go anywhere

✕ Capacity misses

Cache cannot contain all blocks needed

Cache Performance

1. Cold/compulsory miss remains the same irrespective of cache size/associativity
2. Conflict miss goes down with increasing associativity on the same cache size
3. Conflict miss is 0 for FA caches
4. For same cache size, capacity miss remains the same irrespective of associativity
5. Capacity miss decreases with increasing cache size



Cache Misses	
Cold/compulsory	same irrespective of cache size/associativity
Conflict miss	for same cache size, decreases with increasing associativity
Capacity miss	decreases with increasing cache size

Block Replacement Policy

Motivation

In set associative and fully associative caches, a memory block can choose where to be placed, potentially replacing another cache block if full.

Least recently used

When replacing a block, choose one which has not been accessed for the longest time. (Temporal locality)

⚠ **Hard to track if there are many choices**

- FIFO
- RR
- LFU