**Summary Notes (Of important points) (18.19 Sem 2 Syllabus)**

**Data Representation and Number Systems**

1. **Decimal to Binary Conversion**

* **For whole numbers:** Repeated division-by-2 method
  + Keep dividing until cannot quotient is 0, copy the remainders **bottom to up**
* **For fractions:** Repeated multiplication-by-2 method
  + Keep multiplying until fractional part become 0, copy the first digit to the left of the “dot” **top to bottom**

1. **Binary to Octal & Binary to Hex (reverse)**

* Binary to Octal: Group bits in group of 3
  + Octal to Binary: do the reverse
* Binary to Hex: Group bits in group of 4
  + Hex to Binary: do the reverse

1. **ASCII Table**

* In ASCII table, it starts with capital letter first then lowercase letter.
* Each capital letter and it’s corresponding lowercase letter differs by a **decimal value of 32**
* This means that by adding 32 or doing e.g. (‘A’ – ‘a’ = 32) and add the value to a capital letter you’ll get the corresponding small letter. Doing the other way round, which is minus a lower case letter with 32, you’ll get it’s corresponding capital letter.

1. **Data Representation System**

* **Sign and Magnitude**
  + The front most bit as sign, the rest as magnitude
  + To negate a number, simply invert the sign bit
  + Sign: 0 +ve, 1 -ve **(same for 1s complement and 2s complement)**
* **1s Complement**
  + -x = 2n– x – 1, n is the number of bits of the system
  + To obtain the negated value of a number, simply **invert all the bits**
    - Or use the formula above by first converting the decimal value of a number x using the above formula and convert the result to binary (without considering the sign bit thing, just convert as normal)
  + Note that in this system, there is negative 0, which consists of nx 1 bits
* **2s Complement**
  + -x = 2n – x
  + To obtain the negated value of a number, simply **invert all the bits and add 1**
    - Or use the formula above by first converting ….. (same as above)
  + Note that in this system, there is **no negative 0**
* For fractions,
  + 1s complement: invert all the bits as usual
  + 2s complement: invert all the bits and add 1 to the **rightmost digit** (rightmost also means after the “.”

1. **Addition**

* **1s**
  + Perform binary addition as usual
  + If there is a carry out of the MSB, **add 1** to the result
  + Check for overflow (overflow only happens in addition and happens when the MSB of the result is different with the MSB of the operands)
* **2s**
  + Perform binary addition as usual
  + **Ignore** the carry out MSB
  + Check for overflow (overflow only happens in addition and happens when the MSB of the result is different with the MSB of the operands)

1. **Subtraction**

* **1s & 2s**
  + Take the negated 1s / 2s complement of the second operand respectively and perform binary addition with the same procedures as mentioned above

1. **Excess Representation**

* **Decimal to excess N :** change X to X + N, and convert the result to binary.
* **Excess N to decimal :** convert the binary to decimal and subtract the result by N.

1. **Fixed-Point Representation**

* Number of bits allocated for the whole number and fractional part are fixed.

1. **IEEE 754 Single-Precision Floating Point Rep**

* 1 bit sign, 8-bits exponent **(excess-127)**, 23-bit mantissa (fractional part)
* There is an implicit leading digit 1
* Decimal to IEEE 745 Single Precision ….
  + Convert the number to binary, normalize it, and fill in the parts accordingly (sign, exponent, mantissa)
  + Normalize means to convert the format of the number such that there is only 1 digit before the “.”, e.g. -1.101 x 22
  + **Important things to note when filling up the structure table**
    - Convert the exponent value by **adding 127** to it (since the exponent is is represented in excess-127)
    - The fractional part is stored at the front most digits of the mantissa

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**C Programming (Part of it)**

1. Pointer is a variable which **stores the address** of another variable.
2. **Syntax**

* **To declare a pointer (Syntax)**
  + Type \*pointer\_name;
* **To assign address of a variable to it**
  + Pointer = &variable\_name /\* & accesses the address of the variable \*/
* **To access the content of the variable where the pointer is pointing to**
  + Use **\*** to access the content
  + E.g. int a = \*pointer\_name // Store the int data that the pointer is pointing to into a
* **To print address**
  + Use %p format specifier
* **To increment the address of the pointer**
  + Use ++ or + 1
  + It will increment the address **according to the** **type of the variable** that the pointer is pointing to
  + E.g.
  + int \*p = &a; p++ or p += 1 /\* Increment the address by 4\*/
  + char \*u = &b; u++ or u+=1 /\* Increment the address by 1\*/

1. You cannot store an address to a **non-pointer** variable.
2. Must assign an address to a pointer **before** you can use it!
3. When dealing with questions related to pointer, draw out the **box-pointer diagram** clearly! (Cuz they can be tricky)

**Functions**

1. If you define your own function and you choose to written the function definition after the main function, you then need to write the **function prototype** of the function before the main function.

* A function prototype consists of the **return type, function name,** and the **data types of the parameters** (name of the parameters is optional)

1. Parameters are **passed-by-values**.

**Arrays**

1. In C, array name is a **fixed pointer** to the first element in the array.

* Hence, you cannot reassign another array y to the array name of array x
* E.g. x = y /\*illegal if both are array\*/
* However, you can do so in java

1. Array parameters in function (different syntax)

* int sumArray(int \*) /\* Function prototype\*/
* int sumArray(int \*arr) /\* Function definition\*/
* int sumArray(int []) /\* Function prototype\*/
* int sumArray(int arr[]) /\* Function definition\*/

1. You can actually do sth like this

* int a = 5;
* int b[] = {1, 2, a} /\* This will copy the value of a and put inside b[2] \*/

1. **Syntax**

* To declare **an array of pointers**
  + Type \*a[] /\*This means that a is an array of pointers pointing towards variable of type Type\*/
* To declare **pointer to an array**
  + Type \*a
  + E.g.
  + Int a[] = {1, 2, 3}
  + int \*b = a
    - The above is valid because array name is a fixed pointer towards the first element in the array
  + Or Type (\*a)[size] /\*this points to the **entire array** \*/
  + Note this is different with Type \*a
  + [**https://www.geeksforgeeks.org/pointer-array-array-pointer/**](https://www.geeksforgeeks.org/pointer-array-array-pointer/)

**Strings**

1. In C, string is an **array of characters terminating with a null character ‘\0’** (the 0 is a zero)
2. **Syntax**

* To initalise a string:
  + char[] a = “hello” /\*\0 is automatically added at the end\*/
* char[] a = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’} /\* Need to explicitly add ‘\0’ if declared in this way. \*/

1. **To scan a string**

* fgets(str, size, stdin)
* scanf(“%s”, str)

1. **To print a string**

* puts(str) /\* terminates with a newline character \*/
* printf(“%s\n”, str)
* Both of them print until they hit a null character or the end of the char array
* They only work on valid string, meaning that the string must terminates with ‘\0’

1. **Comparing scanf and fgets**

* Both scanf() and fgets() reads in newline character
* scanf() reads until it hits a whitespace
* fgets() reads until size n – 1 or when it hits a newline character
  + the specified size when passed to fgets()

**Structures**

1. Structure is a data structure which allows us to store **heterogenous data**

* It’s like object in java, but it cannot contain functions

1. A structure can contain another structure
2. **Syntax**

* **Declaration** (It’s like declaring your own data type)
* typedef struct {

<attributes…>

} structure\_name**;** /\* Do no forget the semicolon \*/

* Syntax for initialization is similar to that of array
* e.g. result = {“John”, 80, ‘A’}
* **To access attributes**
* Use “.”, e.g. result.student = “Nick”;
* **To access attributes of structures through structure pointer**
* (\*struct\_name).attribute;
* Or simply struct\_name->attribute;
* Cannot write\*struct\_name.attribute cuz ‘.’ has a higher precedence than \*, hence it is like accessing the attribute of a pointer, but pointer doesn’t have attribute!

1. Unlike array, we can do **assignment with structures**

* e.g. result2 = result1 /\***Copy** the entire result1 structure’s data into that of result2\*/

1. Thus, when **passing a structure to a function,** we are essentially **copying the entire structure argument into the structure parameter**
2. We can have an array of structures.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**MIPS**

1. Words are aligned in memory if they begin at an address which is **a multiple of the number of bytes in a word.**
2. In beq or bne, the **immediate** specifies the **number of instructions** to skip over

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Boolean Algebra**

**A screenshot of a cell phone

Description automatically generated**

**A close up of text on a white background

Description automatically generated**

**Important**

1. A.B’ + A’.B = AB
2. A.B + A’.B’ = (AB)’ or AB

The **product** of two distinct **minterms** is always 0.

* If I’m not mistaken, this is because there is always at least a pair of literals (same letter) that is the complement of each other, hence when you perform AND on the two minterms, it will always result in 0 because one of the literal in the pair will the definitely be 0, which cause the entire AND to become 0.
  + E.g. m0.m1 = (x’.y’).(x’.y), in this case the pair is y’ and y

The **sum** of two distinct **maxterms**  is always 1.

* Same reasoning as above. In this case, one of the literal in the pair will definitely be 1, hence making the entire OR to become 1.
  + E.g. M0+M1 = (x+y)+(x+y’), in this case the pair is y and y’

**Simplification**

1. **Gray Code (Reflected Binary Code)**

* Only 1 bit change from one code to the next one

1. **K-map**

* Each cell in a n-variables k-map has **n neighbours**

**A drawing of a person

Description automatically generated A close up of a logo

Description automatically generated A screenshot of a cell phone

Description automatically generated**

2-variables 3-variables 4 variables

1. A grouping in K-map must have a size of **power-of-two**: 1, 2, 4, 8…

* In general, grouping of 2n cells eliminates n variables

1. **PIs and EPIs**

* **Implicant:** a product term that could be used to cover the sum of minterms of a function
* **Prime Implicant(PI):** a product term obtained by combining the **maximum number of minterms** from adjacent squares in the map.
* **Essential Prime Implicant(EPI)**: a prime implicant that contains **at least one minterm that is not covered** by any other prime implicant

1. **Algorithm to find Simplified SOP expression using Kmap**

* Circle all PIs on Kmap
* Identify and select all the EPIs for the cover
* Select a **minimum subset** of the remaining PIs to complete the cover, that is to cover those minterms not covered by the EPIs

1. Note that Kmap guarantees the **simplest** SOP, if everything is done correctly.
2. **To find simplified POS from kmap**

* Do the opposite: Group 0s together, and if the group is inside a variable, use ‘ instead to mark the variable. Also, use “+” to combine different variables and use “.” to connect different sum terms.
* Or to be safer, draw the complement version of the kmap and get the simplified SOP of the complement version. Then negate the simplified SOP of the complement kmapwe got to get the simplified POS of the **non-complement kmap**

1. Use the “Don’t-Cares” to help to form a bigger group.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Logic Circuit**

1. **Universal gates**

* {AND, OR, NOT}
* {NAND}
* {NOR}

1. A **SOP** expression can be implemented easily using:

* 2-level AND-OR circuit (AND first, then OR)
* 2-level NAND circuit
  + To do this, first draw out the 2-level AND-OR circuit first, then put a bubble at the output of all the AND gates, and then put bubbles at the inputs of all the OR gate to convert them into a NAND gates

1. A **POS** expression can be implemented easily using:

* 2-level OR-AND circuit (OR first, then AND)
* 2-level NOR circuit
  + To do this, first draw out the 2-level OR-AND circuit, then put a bubble at the output of all the OR gates, and then put bubbles at the inputs of all the AND gates to convert them into NOR gates

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Combinational Circuit**

1. **Gate-Level Design**

* Draw truth table, draw kmap, find simplified SOP, implement using logic **gates**

1. **Block-level Design**

* There ain’t a fixed way to solve the problem, but what you can usually do is to observe patterns in the truth table, and try to use the given **blocks** to exploit the pattern and implement the circuit

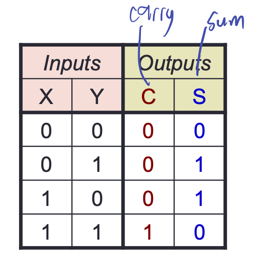
1. Delay in Sum and Carry of a **n-bit parallel adder**

* S = 2nt
* C = (2n + 1)t // Maximum delayof the parallel adder

**Half Adder**

* Can only add 2 bits together
* S = X Y
* C = X.Y

**A picture containing athletic game

Description automatically generated**

**MSI Components**

|  |  |
| --- | --- |
| **MSI Component** | **Remark** |
| **Decoder** | * Given a code, **call/select** the entity * Can be used to **generate** **2n minterms** given n inputs * The order of the outputs corresponds to the order of the **minterms**   + E.g. F0 = minterm 0, F1 = minterm 1, F2 = minterm 2, … * Hence, selecting an output is like making the corresponding minterm to be 1 * The enable signal (if any) is combined with the output minterms using AND   + One-enable: enabled using 1   + Zero-enable: enabled using 0 * **Active High** = normal output * **Active Low** = negated output * **Important** * A **Boolean function** in sum-of-minterm form can be **implemented** by using   + a decoder (to generate the minterms)   + a OR gate (to sum the minterms) * **Any** combinational circuit with **n** inputs and **m** outputs can be **implemented** using a **n:2n** decoder and **m** OR gates * Larger decoder can be made using smaller decoders   A close up of a logo  Description automatically generated |
| Encoder | * Reverse of decoder: see who raise his hand, and generate the person’s code. * Given 2n inputs, generate an n-bit output * Implemented using OR gates. * For a **normal encoder**, at any one time there is **always only** 1 input which is 1   + The inputs cannot be all zeros   + Cannot handle the case where there is more than 1 input which is 1 * **Priority encoder**: Handle the case where there is more than 1 input which is 1   + See which 1 has the highest priority, and generate the code accordingly   + When all inputs are 0, the input is considered invalid |
| **Multiplexer** | * **Select** one of the input and pass it out * The circuit is made from a n:2n decoder and AND gates which combine each output of the decoder with the corresponding data * Larger multiplexers can be made using smaller multiplexers * **Important** * A boolean function with n variables can be implemented with a 2n:1 multiplexer   + Express the function in sum-of-minterms form (i.e. )   + Connect the variables to the selection line   + Put 1 on the data line if the minterm is 1, and 0 otherwise * It can also be implemented using a 2n-1:1 multiplexer   + Express the function in sum-of-minterms form (i.e. )   + **Reserve one variable** **for the input lines** of the multiplexer, and connect the rest to the selection line   + **Draw the truth table** of the function, **group the inputs** by the selection line values, then determine the multiplexer input accordingly by comparing the value of C to F for each group   A picture containing text  Description automatically generated |
| Demultiplexer | * Directs the data to the **selected** output line. * The circuit is **identical** to that of decoder   A close up of text on a white background  Description automatically generated |

**Sequential Circuit**

**Latches**

1. Pulse-triggered

|  |  |
| --- | --- |
| **Type** | **Remark** |
| S-R Latch | A picture containing screenshot  Description automatically generated |
| D latch | The value of the output follows the value of D |

**Flip-Flop**

1. **Edge-triggered** (Changes state at rising / falling edge)

|  |  |
| --- | --- |
| **Type** | **Remark** |
| S-R flip flop | Q(t + 1) = S + R’.Q |
| D flip-flop | Q(t + 1) = D  A drawing of a person  Description automatically generated |
| J-K flip-flop | Q(t + 1) = J.Q’ + K’.Q |
| T flip-flop | Q(t) = T.Q’ + T’.Q  A screenshot of a cell phone  Description automatically generated |

1. **Asynchronous Inputs**

* The output can change regardless of the clock
* In an active-high asynchronous flip-flop:
  + **PRE/SD** is high, Q immediately become high (SET)
  + **CLR/RD** is high, Q immediately become low (RESET)
* Flip-flop only functions normally if both PRE and CLR is low

1. M flip-flops -> 2m states
2. **Analysis of Sequential Circuit**

* Derive the **state equation** and **output function** from the given circuit
  + Or instead of using state equation, you can also find the **flip-flop input functions** and use it to determine the state transition
* Draw **a compact state table** and fill it up accordingly
  + Should have these columns: present state, next state (based on inputs), and flip-flop inputs
* Draw a **state diagram**
  + Convention: x/y --> input/output

1. **Excitation Table (Used in design of circuit)**

A close up of text on a white background

Description automatically generated

1. **Designing of sequential Circuit**

* Given a state diagram, first determine:
  + How many flip-flops are needed? (from the total number of states)
  + What is the type of flip-flop to be used? (D flip-flop usually results in the easiest state table)
  + How many inputs are there?
* Make a state table with the necessary columns and fill it up using the state diagram
* From the state table, draw kmaps for each **flip-flop input function.**
* Determine the **simplified SOP expression** of the flip-flops function from the kmaps
* Draw the circuit

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pipelining**

**Introduction**

1. **Instructions Execution Time**

* Single-cycle: **I x Tsingle cycle**
  + Tsingle cycle =
* Multiple-cycle: **I x Average CPI x CTmulti**
  + CTmulti = max(Tstage)
  + CPI = num of cycles per instruction
* Pipeline: Numcycles x CTpipeline = **(I + N – 1) x (max(Tstage) + Td)**
  + N = number of stages in the pipeline (5 for MIPS)
  + Td = pipeline overhead

1. Speeduppipeline=

**Pipeline Hazards**

1. **3 Types of Hazards**

* Structural Hazards
* Data Hazards (Instruction Dependency)
* Control Hazards (Instruction Dependency)

1. Draw **table** when solving the number of cycles



1. **Solution to Data Hazards**

* **Data forwarding**
  + But it doesn’t work all the time, esp when load instructions (need to stall)



1. **Solutions to Control Hazards**

* **Early Branching**
  + If no early branching (i.e. **decide** in MEM), then need to **delay** **3 cycles**
  + If early branching (i.e. **decide** in ID), **delay** **1 cycle**
  + However, if the register involved in the comparison is produced by preceding instructions, then the **branch instruction** (not the next instruction) need to be delayed.
* **Branch Prediction**
  + Simple prediction: assume instruction to be **not taken**
  + If not taken: no pipeline stall
  + If taken: flush the instruction and its successor
* **Delayed Branch**
  + Move non-control dependent instruction (if any) into the **delayed slot.**
  + Hence, the delayed slot is utilized and technically there is no delay.

**Tips in determining how many cycles are needed to execute a piece of code**

* If there is **no data forwarding,**
  + Then the **ID** stage of an instruction, which has data dependency (register) on other instruction x, can only happen at the **WB** stage of the instruction X. Cuz the writing during WB happens during the first half of the cycle whereas the reading during ID only happen during the second half of the cycle.
  + Be careful of lw as well cuz it can incur delay
* If there is **no early branching**,
  + Then the **IF** stage of the instruction that immediately follows the branching instruction can only happen during the **WB** stage of the branching instruction because the branching decision is only computed during the MEM stage and can only be executed during the WB stage.
  + i.e. there will be **3 clock cycles delay**
* If there **is data forwarding**
  + Then be careful of lw cuz it usually will incur delay
  + Sw also
* If there **is early branching,** 
  + Then the **IF** stage of the instruction that immediately follows the branching instruction will happen at the **EXE** stage of the branching instruction because the branching decision is only computed at the ID stage and can only be executed at the next stage
  + i.e. there is still **1 clock cycle delay**.

1. Note that the data from the **memory** is only available after the MEM stage, i.e. at the start of the WB stage
2. To calculate the number of instructions executed in a code involving loops:

* **A:** First identify how many cycles are **not** included in the loop **before** the loop is executed (if any).
* **B:** Then identify how many cycles are involved in one loop
  + Be careful with this to **ensure there is no overlapping** between cycles of different loops (e.g. in tutorial 10q2d, the last W is not included as it is covered by the first F of the next loop / next instruction)
* **C:** Then identify how many cycles are **not** included in the loop **after** the loop is executed (if any).
* Then, the total number of cycles would be A + (B \* N) + C, where N is the number of iterations.
* Referring to the answer slides provided for tutorial 10 q2d, there are a total of 2 cycles that are **not** included in the loop **before** the loop (the IF and IF of the first two instructions)
* Then in each loop there are 18 cycles, note that the last W is not included as it is covered by the first F of the next loop / next instruction
* Then after the loop there are a total of 9 cycles not included in the loop (bne & beq)
* Hence the total number of cycles are = 2 + (4\*18) + 9 = 83

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Cache**

1. **Cache**

* is like a table, on which you can put books (memory block). It helps to reduce the frequency of direct memory access, which is very slow.
* is based on the **principle of locality**
* Uses **static RAM** (faster access compared to dynamic RAM)

1. **2 types of locality**

* Temporal Locality
  + If a data is accessed, then most likely it will be accessed soon again
* Spatial Locality
  + If a data is accessed, then its nearby data is most likely to be accessed in the near future

1. **Average Memory Access Time**

* (Hit rate x Hit Time) + [(1 – Hit rate) \* Miss Penalty]

1. 1 KB = 210 byes = 1024 bytes

* 1 MB = 220 bytes, 1 GB = 230 byes

1. **3 types of Misses**

* Compulsory Miss / Cold Miss
* Conflict Miss
* Capacity Miss
  + Occur because the block was discarded from the cache as the cache was full

1. **3 types of caching techniques**

* Direct map
  + Each memory block match to exactly one slot in the cache and hence many collisions can occur, esp when accessing memory block with the same index but different tag frequently
  + **Cache Block Index Mapping Function** = Block number % number of blocks in cache
  + **Mapping** 
    - Given block size as 2M, number of blocks in cache as 2N
    - Offset = M bits
    - Block Index = N bits
    - Tag = 32 – N – M bits
* N-way Set associative
  + Each memory block match to exactly one **set** in the cache. The memory block can go to any slots inside the set.
  + A set has N block slots and a block that mapped to the set can go into any of the slots.
  + Alleviate the portion of miss rate that is due to conflict miss
    - A direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2
  + **Cache Set Index Mapping Function** = Block number % number of **sets** in cache
  + **Mapping**
    - Given block size as 2M, number of **sets** in cache is 2N
    - Offset = M bits
    - **Set** index = N bits
    - Tag = 32 – N – M bits
* Fully associative
  + You just put the memory block into the cache without any form of mapping (i.e. you can put it anywhere you want). There is no index involved, only the tag (the block number).
  + Need to search through the entire cache using the tag(block number) to find the block
  + **Tag**
    - Given block size as 2M
    - Offset = M bits
    - Tag = 32 – M bits

1. **2 Writing Policies**

* **Write-Through** Cache
  + Write to both the cache & memory
  + Optimization: Write into cache & write buffer (and continue with the program). Memory controller will write into memory from write buffer.
  + “Through” --> Cache & memory
* **Write-Back** Cache
  + Write into the cache only.
  + Only write into the memory if the block is evicted (refer details)
  + Optimization: Use a **dirty bit** to indicate whether the block in the cache was updated.
    - Dirty bit: 0 if was not updated; 1 if was updated
    - Only write into the memory if the block is evicted **and** the dirty bit is 1

1. **2** **Write Miss Policies**

* **Write Allocate**
  + Load the entire block into the cache
  + Write only the word that is changed
  + Writing back to the memory depends on writing policies
* **Write Around** 
  + Do not load the entire block into the cache
  + Only write directly into the memory

1. **Block Replacement Policies**

* Least **Recently** Used (LRU)
* First-in-First-out (FIFO)
* Random Replacement (RR)
* Least Frequently Used (LFU)