COA Practical List

|  |  |
| --- | --- |
| Expt No | Expt Name |
|  | Implementation of Booth’s Multiplier |
|  | Implementation of Bitwise Division algorithm |
|  | Adder |
|  | ALU – 4 bits |
|  | Controller Design – Booth |
|  | Fully Associative Cache |
|  | Direct Mapped Cache |
|  | 2/ 4Way set Associative Cache |
|  | Virtual Memory simulation |
|  | LRU implementation |
|  | Study on Dynamic Branch Prediction |

**Experiment No 1**

**Title:** A program for binary multiplication.(Booth’s Algorithm)

**Objective**: To study of n bit parallel binary Addition.

**Pre-requisites:**1. Binary Arithmetic

**References:**

1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, “Computer Organization”,

Fifth Edition, Tata McGraw-Hill.

1. John P. Hayes, “Computer Architecture and Organization”, Third Edition.
2. William Stallings, “Computer Organization and Architecture: Designing for Performance”, Eighth Edition, Pearson.

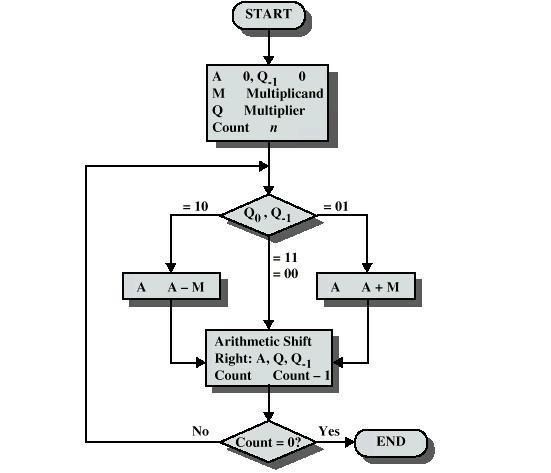
**Theory :**

With unsigned multiplication there is no need to take the sign of the number into consideration. However in signed multiplication the same process cannot be applied because the signed number is in a 2’s compliment form which would yield an incorrect result if multiplied in a similar fashion to unsigned multiplication. That’s where Booth’s algorithm comes in. Booth’s algorithm preserves the sign of the result.

**Algorithm:**

1. Multiplier and multiplicand are placed in the Q and M registers reply
2. One bit register(Q-1) placed logically to the right of least significant bit of the register Q0.
3. The result of multiplication will appear in A and Q registers.
4. A and Q-1 are initialized to zero.
5. If the combination of two bits is same(0-0 or 1-1), then all the bits of A , Q and Q-1 registers are shifted to right by 1 bit.
6. If two bits differ , then the multiplicand is added to or subtracted from register A depending upon whether two bits are 0-1 or 1-0.
7. Following the addition or subtraction, the right shift occurs. 
8. In either case, the right shift is such that the leftmost bit of A, namely A n-1, not only is shifted into A n-2, but also remains in A n-1. This is called **Arithmetic Shift,** because it preserves the sign bit.

**Flowchart of Booth’s Algorithm:**



**Post Lab Questions:**

1. Write advantages of this algorithm
2. Explain hardware description of this algorithm

**Experiment No 2**

**Title:** Write a program to implement Restoring algorithm for Division.

**Objective:** 1. Understanding behaviour of division algorithm for unsigned numbers

1. Implementing restoring division algorithms.

**References:** 1. Computer Organization – Carl Hamacher, Zvonko Vranesic, Swaft Zaky.

2. Computer Organization and Architecture – William Stallings

**Theory:** Division operation implements as follows: it position the divisor appropriately with respect to the dividend and performs a subtraction If the reminder is Zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. On the other hand, if the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

**Restoring Division**

An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotients in register Q and the remainder is in register A. the required subtraction are facilitated by using 2’s complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

**Non-restoring Division**

he restoring -division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful is the result is negative. Consider the sequence of operation that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A-M. If A is negative, we restore it by performing A+M, and then we shift it left and subtract M. This is equivalent to performing 2A+M. the q0 bit is appropriately set 0 or 1 after the correct operation has been performed.

**Algorithm:**

**Restoring Division**

Do the following n times:

1. Shift A and Q left one binary position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q0 to 0 add M back to A (that is restore A); otherwise, set q0 to 1.

**Non-restoring Division**

**Step 1:** do the following n times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q0 to 1; otherwise, set q0 to 0.

**Step 2:** if the sign of A is 1, add M to A

**Post Lab Assignment:**

1. Explain how to perform division on signed operands.

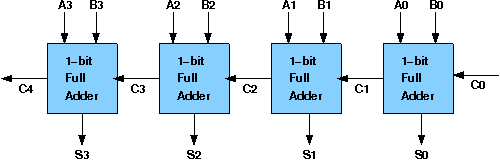
**Experiment No 3**

**Title: Ripple Carry Adders :**

**Objective: To design ripple Carry Adder using Full adders using Simulator**

Arithmetic operations like addition, subtraction, multiplication, division are basic operations to be implemented in digital computers using basic gates likr AND, OR, NOR, NAND etc. Among all the arithmetic operations if we can implement addition then it is easy to perform multiplication (by repeated addition), subtraction (by negating one operand) or division (repeated subtraction).

Half Adders can be used to add two one bit binary numbers. It is also possible to create a logical circuit using multiple full adders to add N-bit binary numbers.Each full adder inputs a **Cin**, which is the **Cout** of the previous adder. This kind of adder is a **Ripple Carry Adder**, since each carry bit "ripples" to the next full adder. The first (and only the first) full adder may be replaced by a half adder.The block diagram of 4-bit Ripple Carry Adder is shown here below -



The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic.In a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is 31 \* 2(for carry propagation) + 3(for sum) = 65 gate delays.

## Design Issues :

The corresponding boolean expressions are given here to construct a ripple carry adder. In the half adder circuit the sum and carry bits are defined as

**sum = A ⊕ B**

**carry = AB**

In the full adder circuit the the Sum and Carry output is defined by inputs A, B and Carryin as

**Sum=ABC + ABC + ABC + ABC**

**Carry=ABC + ABC + ABC + ABC**

Having these we could design the circuit.But,we first check to see if there are any logically equivalent statements that would lead to a more structured equivalent circuit.

With a little algebraic manipulation,one can see that

**Sum= ABC + ABC + ABC + ABC**

**= (AB + AB) C + (AB + AB) C**

**= (A ⊕ B) C + (A ⊕ B) C**

**=A ⊕ B ⊕ C**

**Carry= ABC + ABC + ABC + ABC**

**= AB + (AB + AB) C**

**= AB + (A ⊕ B) C**

**Procedure to perform the experiment:Design of Ripple Carry Adders**

1. Start the simulator as directed.This simulator supports 5-valued logic.
2. To design the circuit we need 3 full adder, 1 half adder, 8 Bit switch(to give input), 3 Digital display(2 for seeing input and 1 for seeing output sum), 1 Bit display(to see the carry output), wires.
3. The pin configuration of a component is shown whenever the mouse is hovered on any canned component of the palette or press the 'show pin config' button. Pin numbering starts from 1 and from the bottom left corner(indicating with the circle) and increases anticlockwise.
4. For half adder input is in pin-5,8 output sum is in pin-4 and carry is pin-1, For full adder input is in pin-5,6,8 output sum is in pin-4 and carry is pin-1
5. Click on the half adder component(in the Adder drawer in the pallet) and then click on the position of the editor window where you want to add the component(no drag and drop, simple click will serve the purpose), likewise add 3 full adders(from the Adder drawer in the pallet), 8 Bit switches, 3 digital display and 1 bit Displays(from Display and Input drawer of the pallet,if it is not seen scroll down in the drawer)
6. To connect any two components select the Connection menu of Palette, and then click on the Source terminal and click on the target terminal. According to the circuit diagram connect all the components, connect 4 bit switches to the 4 terminals of a digital display and another set of 4 bit switches to the 4 terminals of another digital display. connect the pin-1 of the full adder which will give the final carry output. connect the sum(pin-4) of all the adders to the terminals of the third digital display(according to the circuit diagram shown in screenshot). After the connection is over click the selection tool in the pallete.
7. To see the circuit working, click on the Selection tool in the pallet then give input by double clicking on the bit switch, (let it be 0011(3) and 0111(7)) you will see the output on the output(10) digital display as sum and 0 as carry in bit display.

**Components :**

The components needed to create 4 bit ripple carry adder is listed here -

* 4 full-adders
* wires to connect
* LED display to obtain the output

**Post Lab Questions:**

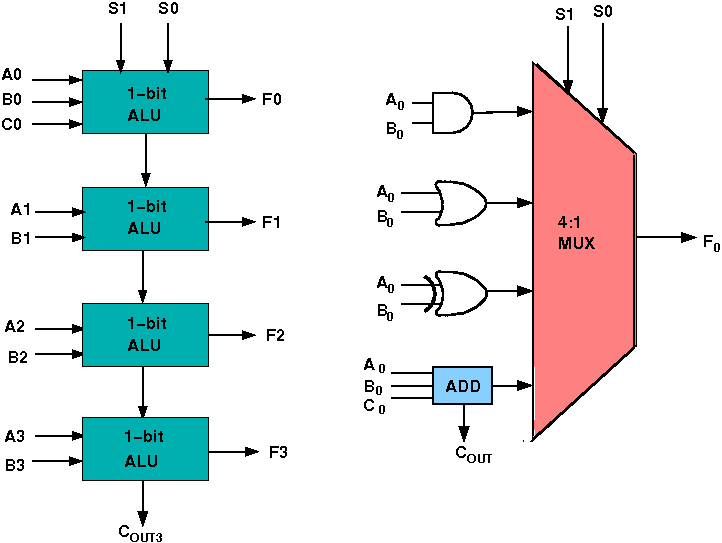
Explain Carry Look Ahead Adder .Give its advantage.

**Experiment No 4**

**Title: 1 Bit and 4 bit ALU :**

**Objective:To implement 1 bit ALU and 4 bit ALU using 1 bit ALU**

ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction,division, multiplication and logical oparations like and, or, xor, nand, nor etc. A simple block diagram of a 4 bit ALU for operations and,or,xor and Add is shown here :



The 4-bit ALU block is combined using 4 1-bit ALU block

**Design Issues :**

The circuit functionality of a 1 bit ALU is shown here, depending upon the control signal S1 and S0 the circuit operates as follows:

for Control signal **S1 = 0** , S0 = 0, the output is **A And B**,

for Control signal **S1 = 0** , S0 = 1, the output is **A Or B**,

for Control signal **S1 = 1** , S0 = 0, the output is **A Xor B**,

for Control signal **S1 = 1** , S0 = 1, the output is **A Add B**.

**The truth table for 16-bit ALU with capabilities similar to 74181 is shown here:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Required functionality of ALU (inputs and outputs are active high) | | | | | |
| **Mode Select** | | | | **Fn for active HIGH operands** | |
| **Inputs** | | | | **Logic** | **Arithmetic (note 2)** |
| S3 | S2 | S1 | S0 | (M = H) | (M = L) **(Cn=L)** |
| L | L | L | L | A' | A |
| L | L | L | H | A'+B' | A+B |
| L | L | H | L | A'B | A+B' |
| L | L | H | H | Logic 0 | minus 1 |
| L | H | L | L | (AB)' | A plus AB' |
| L | H | L | H | B' | (A + B) plus AB' |
| L | H | H | L | A ⊕ B | A minus B minus 1 |
| L | H | H | H | AB' | AB minus 1 |
| H | L | L | L | A'+B | A plus AB |
| H | L | L | H | (A ⊕ B)' | A plus B |
| H | L | H | L | B | (A + B') plus AB |
| H | L | H | H | AB | AB minus 1 |
| H | H | L | L | Logic 1 | A plus A (Note 1) |
| H | H | L | H | A+B' | (A + B) plus A |
| H | H | H | L | A+B | (A + B') plus A |
| H | H | H | H | A | A minus 1 |

The L denotes the logic low and H denotes logic high

**Experiment No 5**

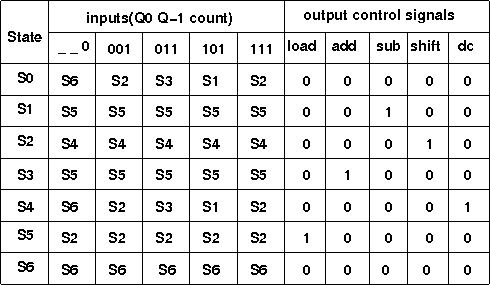
**Title: Design of Control unit:**

**Objective:To Design and Implement controller for Booth’s Multiplier to generate the required control signals using Finite State Machine.**

**Theory:**

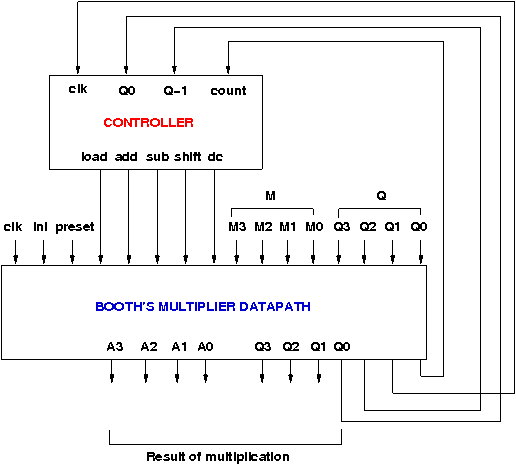
1. Start the simulator as directed.This simulator supports 5-valued logic.
2. To perform the experiment on the given modules, we need the datapath specified for booth's multiplication, a controller with a specified state chart, a clock input, bit switch (to give input, which will toggle its value with a double click), bit displays (for seeing output), wires.
3. Instantiating the controller: A control unit can be seen as a finite state machine, so its behavior can be represented in a state table. The controller of the simulator accepts the Moore type state chart and must contain an end state. State names will automatically be generated in the form of S<sub<n< sub="">. In the left pane of the simulator, click on the ASM chart button in the controller subsection. Give the required informations in the appeared form as follows:
   * Number of states: 7
   * Number of inputs: 3
   * Number of outputs: 5

The controller will generate 5 output control signals. After entering these informations, the second form will appear where you can set the names of the inputs and outputs. Here inputs are the Q0, Q-1, count. Outputs control signals are load, add, sub, shift, dc. The order of given input/outputs are maintained while creating terminals of the controller. for example, the first output signal will appear in the left most output terminal (lower terminals), second output will appear in the second left most bit and so on. In case of input terminals, the left most bit is for clock input, so the first input appears in the second terminal, and then the order is maintained. Then the third form will appear where you actually specify the state chart i.e. state, outputs of that state and transition conditions . The fields of the chart will be generated dynamically according to previously given information on states, inputs and outputs.After entering the following state chart, click on the controller component in the palette of the simulator then click on the position of the design editor where you want to put the component (no drag and drop, simple click will serve the purpose).



</sub<n<>

1. Instantiate the Booth's multiplier datapath from the sequential ckt drawer in the palette (by clicking as mentioned previously).
2. The pin configuration of the component is shown whenever the mouse is hovered on any canned component of the palette or pressing the *show pin configuration* button on the toolbar will show it constantly in the left pane. Pin numbering starts from 1 and from the bottom left corner(indicating with the circle) and increases anticlockwise.
3. Pin configuration of the datapath module:
   * M : Multiplicand (4 bit), Q : Multiplier (4 bit)
   * Initialization : Inl:1, preset:1, set M, Q, start clock
   * Starting multiplication: Inl:0, preset:0, start clock
   * Result: FQ0 to FA3, at end state (here it is S6). These are the content of A(4bit) and Q(4bit) register, total 8 bit (FQ0 is LSB, FA3 is MSB)
   * I/P:
     + Clk:32, Inl:31, preset:30
     + Control pins: load:29, add:28, sub:27, shift:26, dc:25
     + Multiplicand: M3 : 24, M2 : 23, M1 : 22, M0 : 21
     + Multiplier: Q3 : 20, Q2 : 19, Q1 : 18, Q0 : 17
   * O/P: FQ-1 is output of Q-1 bit register, similarly FQ0 to FQ3 are for Q register and FA0 to FA3 are for A register.
     + Datapath to controller input: Count, clkToController, FQ-1, FQ0
     + Count : 16, clkToController : 15, FQ-1 : 14, FQ0 : 13
     + FQ1 : 12, FQ2 : 11, FQ3 : 10
     + FA0 : 9, FA1 : 8, FA2 : 7, FA3 : 6
4. To connect any two components select the Connection menu of Palette, and then click on the Source terminal and click on the target terminal. According to the following diagram connect all the components. Connect the controller outputs to the specified control input terminals of the datapath, specified datapath outputs to the inputs of the controller, the clock input, Bit switches with the inputs and Bit displays component with the outputs (from Display and Input drawer of the pallet,if it is not seen scroll down in the drawer). After the connection is over click the selection tool in the pallete.



1. At first initialize the multiplier by giving the specified inputs specified earlier, this will load the multiplier and multiplicand, then start the multiplication operation by giving the specified inputs specified earlier. At the end state (S6), the multiplication result will be seen through ports FQ0 to FA3 (FQ0 is LSB, FA3 is MSB). The current state of the controller is shown in the left pane as it transits from one state to another. The controller can be reset by clicking the *reset controller* button in the top toolbar, to start with a new input.

**Experiment No 6**

**Title: Fully Associative Mapping**

**Objective: To create a fully associative cache**

**Components :**  To build a associative Cache with 4 bit memory address and 2 bit data address without any replacement policy, we need :

1. Decoder with enable and decoder without enable
2. Multiplexer with enable and multiplexer without enable
3. Single bit memory elements
4. XOR gates, NOR gates, AND gates
5. Bit switches to give inputs
6. Display units to check the outputs.
7. Wires to connect.

**Theory:**

Cache memory is a small (in size) and very fast (zero wait state) memory which sits between the CPU and main memory. The notion of cache memory actually rely on the correlation properties observed in sequences of address references generated by CPU while executing a programme(principle of locality).When a memory request is generated, the request is first presented to the cache memory, and if the cache cannot respond, the request is then presented to main memory.

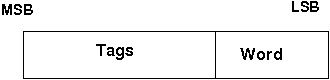
* Hit: a cache access finds data resident in the cache memory
* Miss: a cache access does not find data resident, so it forces to access the main memory.

Cache treats main memory as a set of blocks.As the cache size is much smaller than main memory so the number of cache lines are very less than the number of main memory blocks. So a procedure is needed for mapping main memory blocks into cache lines.cache mapping scheme affects cost and performance. There are three methods in block placement-

* Direct Mapped Cache
* Fully Associative Mapped Cache
* Set Associative Mapped Cache

Fully Associative Cache

Any main memory block can mapped into any cache line. main memory address is divided into two groups which are tags and word bits. Words are low-order bits and identifies the location of a word within a block and tags are high-order bits which identifies the block.



So for eg: In a pentium system,main memory is 4GB in size and RAM is 8KB in size.With each block of 32 Bytes in size.The RAM consists of 256 blocks and the main memory consists of 227 blocks i.e 27 bits are needed for tag directory .Since,any block of the MM can be mapped in any block of RAM.

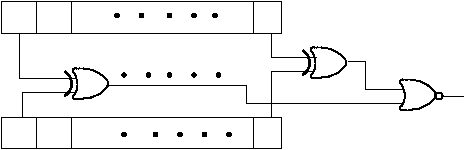
#### 

#### 

#### Design issues:

No replacement policy has been implemented in the experiment.

The comparator Circuit through which tag is compared with specified bits of address:



**Post Lab Questions:**

1. show the memory address representation for the main memory and Ram in bits and Solve for tag size,search time for the following configuration using Fully Associative set mapping: a) MM=4GB , RAM=8KB ,Block size= 32B b) MM=64B , RAM=8B ,Block size= 4B

**Experiment No 7**

**Title: Direct Mapping**

**Objective: To create a directly mapped cache**

**Components :**  To build a associative Cache with 4 bit memory address and 2 bit data address without any replacement policy, we need :

1. Decoder with enable and decoder without enable
2. Multiplexer with enable and multiplexer without enable
3. Single bit memory elements
4. XOR gates, NOR gates, AND gates
5. Bit switches to give inputs
6. Display units to check the outputs.
7. Wires to connect.

**Theory:**

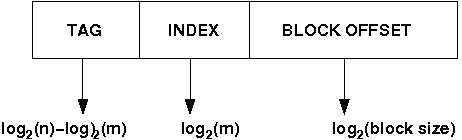
**Design of Direct Mapped cache :**

A given memory block of any set can be mapped into one and only block.

**Block identification:** let the main memory contains n blocks(which require log2(n)) and cache contains m blocks, so n/m different blocks of memory can be mapped (at different times) to a cache block. Each cache block has a tag saying which block of memory is currently present in it, each cache block also contain a valid bit to ensure whether a memory block is in the cache block currently.

* **Number of bits in the tag: log2(n/m)**
* **Number of sets in the Cache: m**
* **Number of bits to identify the correct set: log2(m)**

The memory address is divided into 3 parts- tag(most MSB), index, block offset(most LSB) in order to do the cache mapping.

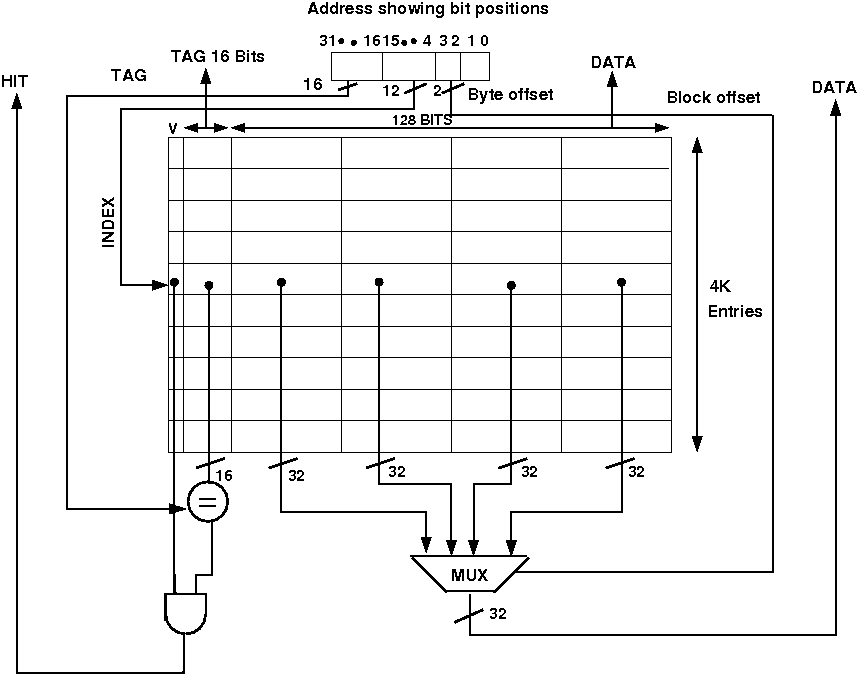


**Select set using index, block from set using tag.**

**Select location from block using block offset.**

**tag + index = block address**

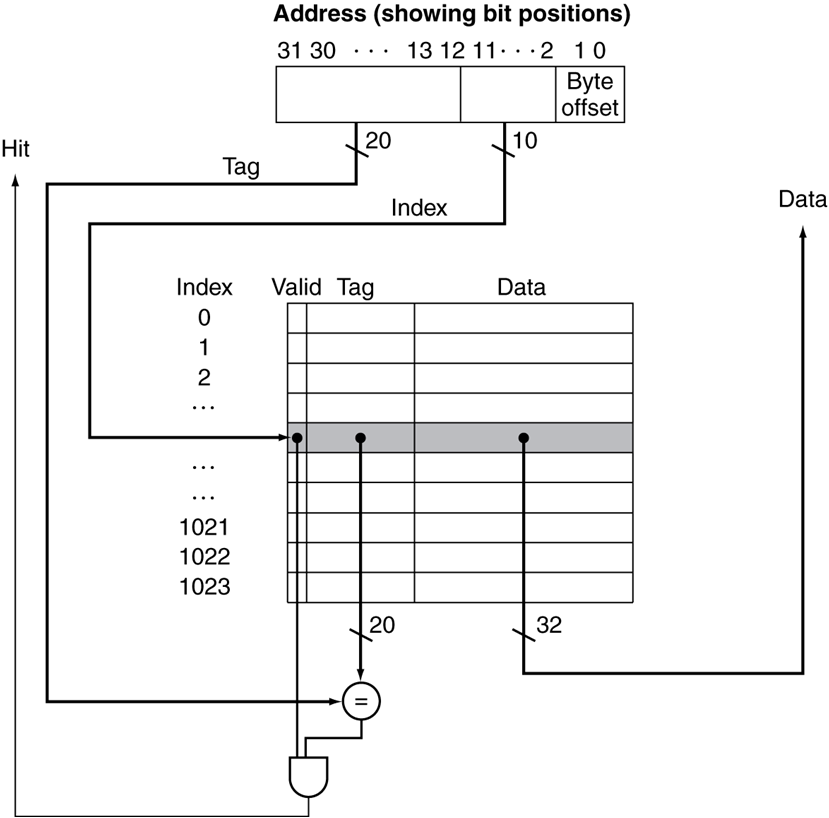
Diagram of a direct mapped cache (here main memory address is of 32 bits and it gives a data chunk of 32 bits at a time):



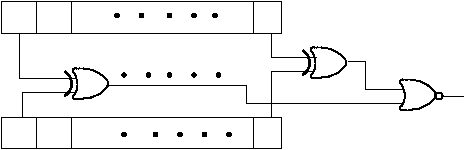
If a miss occur CPU bring the block from the main memory to the cache, if there is no free block in the corresponding set it replaces a block and put the new one. CPU uses different replacement policies to decide which block is to replace. The disadvantage of the direct mapped cache is that it is easy to build, but suffer the most from thrashing due to the 'conflict misses' giving more miss penalty.

**Design issues:**

Bellow is a simple cache which holds 1024 words or 4KB, memory address is 32 bits. The tag from the cache is compared against the most significant bits of the address to determine whether the entry in the cache corresponds to the requested address as the cache has 210 or 1024 words and a block size of one word, 10 bits are used to index the cache, leaving 32-10-2=20 bits to be compared against the tag. If the tag and the most significant 20 bits of the address are equal and the valid bit is on then the request hits in the cache otherwise miss occurs. No replacement policy has been implemented in the circuit.



The comparator Circuit through which tag is compared with specified bits of address:



**Post Lab Questions:**

* show the memory address representation for the main memory and Ram in bits and Solve for tag size,search time for the following configuration using direct associative mapping: a) MM=4GB , RAM=8KB ,Block size= 32B b) MM=64B , RAM=8B ,Block size= 4B

**Experiment No 8**

**Title: Set Associative Mapping**

**Objective: To create a 2 way set associative mapped cache**

**Theory:**

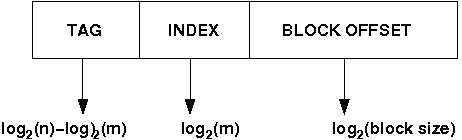
**Design of Direct Mapped cache :**

A given memory block of any set can be mapped into one and only block.

**Block identification:** let the main memory contains n blocks(which require log2(n)) and cache contains m blocks, so n/m different blocks of memory can be mapped (at different times) to a cache block. Each cache block has a tag saying which block of memory is currently present in it, each cache block also contain a valid bit to ensure whether a memory block is in the cache block currently.

* **Number of bits in the tag: log2(n/m)**
* **Number of sets in the Cache: m**
* **Number of bits to identify the correct set: log2(m)**

The memory address is divided into 3 parts- tag(most MSB), index, block offset(most LSB) in order to do the cache mapping.

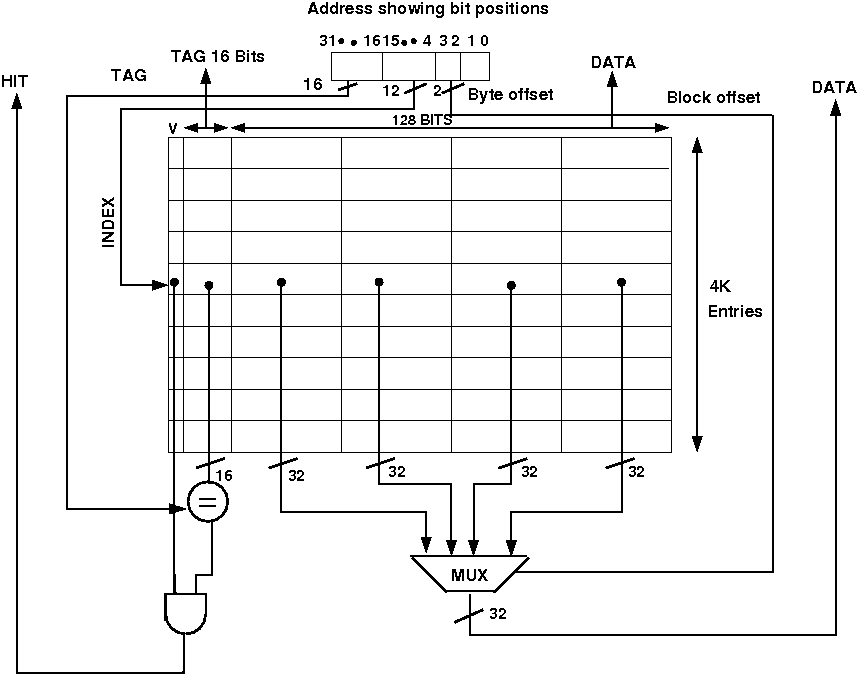


**Select set using index, block from set using tag.**

**Select location from block using block offset.**

**tag + index = block address**

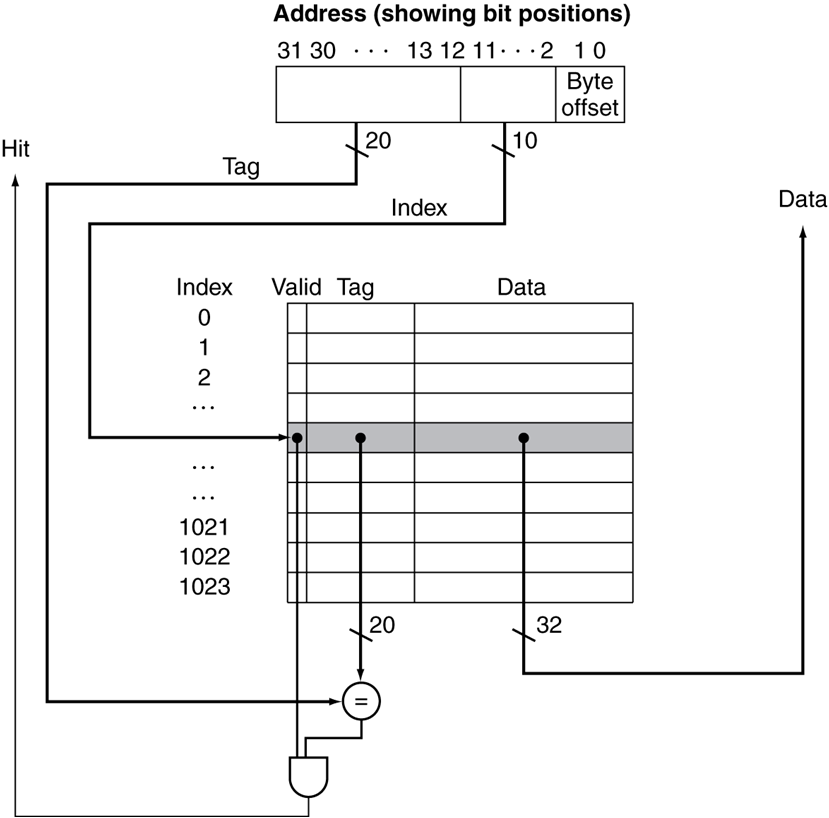
Diagram of a direct mapped cache (here main memory address is of 32 bits and it gives a data chunk of 32 bits at a time):



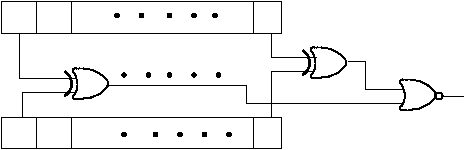
If a miss occur CPU bring the block from the main memory to the cache, if there is no free block in the corresponding set it replaces a block and put the new one. CPU uses different replacement policies to decide which block is to replace. The disadvantage of the direct mapped cache is that it is easy to build, but suffer the most from thrashing due to the 'conflict misses' giving more miss penalty.

**Design issues:**

Bellow is a simple cache which holds 1024 words or 4KB, memory address is 32 bits. The tag from the cache is compared against the most significant bits of the address to determine whether the entry in the cache corresponds to the requested address as the cache has 210 or 1024 words and a block size of one word, 10 bits are used to index the cache, leaving 32-10-2=20 bits to be compared against the tag. If the tag and the most significant 20 bits of the address are equal and the valid bit is on then the request hits in the cache otherwise miss occurs. No replacement policy has been implemented in the circuit.



The comparator Circuit through which tag is compared with specified bits of address:



**Post Lab Questions:**

show the memory address representation for the main memory and Ram in bits and Solve for tag size,search time for the following configuration using set associative mapping: a) MM=4GB , RAM=8KB ,Block size= 32B b) MM=64B , RAM=8B ,Block size= 4B

**Experiment No 9**

**Title: Virtual Memory Management**

**Objective:To study the concept of Virtual Memory Management**

**Theory:**

Virtual memory is a memory management capability of an operating system (OS) that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory(RAM) to disk storage.This can be achieved by using :

1. Paging
2. Segmentation

Virtual Memory Management using Paging:

1. The **Virtual memory** space is divided into equal size blocks called “**Pages**”
2. The **Physical memory** space is also divided into equal size blocks called “**Page Frames**”
3. The pages from the virtual memory are loaded into the page frames of the main memory
4. for eg: MM size=4 GB,Page size=4KB, no. of pages=4GB/4KB=22\*230/22\* 210 =232/212 =220 =1M.
5. When a page is needed the processor checks whether the page exists in the main memory
6. If found ,it is called a “**HIT**” and the operation is performed in the main memory
7. If the desired the page is not found in the main memory it is called a **“MISS”** or a **“Page Fault”**
8. On a page fault the desired page is loaded from the main memory from the virtual memory
9. If no empty page frame could be found in the main memory then page replacement algorithms are used
10. An old page is replaced from the main memory with the new desired page from the virtual memory
11. Popular Algorithms are:LRU(Least Recently Used),Optimal,LFU(Least Frequently Used),FIFO(First In First Out),are used to determine which page of the physical memory has to replaced
12. If there are too many page faults the system is said to be **thrashing**,this must be avoided
13. while replacing ,the dirty bit is used to be determined if a page is to be modified in the

physical memory

1. if dirty bit=”1”,then the page has been modified and must be copied in the main memory
2. if dirty bit=”0”,then the page must be copied as it is.
3. since any page of virtual memory can be loaded into any page frame of the physical memory a page table is required to give the mapping from virtual memory page no. to physical memory page frame no.
4. **virtual memory** address can be divided into two parts **|page no.|location in page|**
5. **physical memory** address can be divided into two parts **|page frame no.|location in page|**
6. But this process is very slow ,for accessing any location we have to access the page frame from page table ,then access the page in physical memory address
7. page table is also in the memory this means an additional stage to access the page frame from page table
8. to speed up the process the “**Translation Look-aside buffer** ”is used (called TLB)
9. the TLB is a on chip cache present inside the processor
10. it stores 32 most recently used page table entries
11. This makes the subsequent access to these pages much faster as there is no need to access the page table
12. Due to principle of “**Locality of reference**” most systems get hit ratio greater than 98%,on TLB,thus making the operations faster
13. An application may require several pages ,All are not loaded into physical memory at once .Instead, pages are loaded as and when the pages are required by the processor.This is called **Demand Paging**

|  |  |  |
| --- | --- | --- |
| **sr no.** | **Paging** | **Segmentation** |
| **1.** | **Paging is physical division of memory** | **Segmentation is logical Division of memory** |
| **2.** | **Pages are of fixed size** | **Segments are of variable sizes** |
| **3.** | **Fragmentation is high as pages are of fixed size** | **Fragmentation is low as segments are of variable size** |
| **4.** | **completely hidden from programmer** | **controlled by programmer** |
| **5.** | **Does not offer much control over protection mechanism** | **As segments are controlled by programmer ,we can assign different privilege levels to segments .Hence,provide better control over mechanism** |
| **6.** | **Does not differentiate between code and data** | **Code and Data Segments are treated differently** |
| **7.** | **pages are loaded based on the algorithms such as FIFO,LRU, LFU , etc** | **segments are loaded based on the algorithms such as First Fit,Best Fit,Worst Fit, etc** |
| **8.** | **Does not allow sharing of data or code between processes/tasks** | **segments can either be global or local ,global segments can be shared among all tasks** |

**Post Lab Questions:**

1. **consider a fully associative cache with 8 blocks . the memory block requests are in order-1, 4, 3, 2, 25, 36, 12, 23, 34, 45, 56, 54, 2, 3, 4, 16, 19, 25, 19, 7, 6**
2. **if MFU replacement policy is used which cache block will have memory block 6?**
3. **calculate the hit ratio and miss ratio.**

**Experiment No 10**

**Title:** A program for LRU page replacement algorithm.

**Objective**: To study page replacement algorithm.

**References:**

1. John P. Hayes, “Computer Architecture and Organization”, Third Edition. 2. William Stallings, “Computer Organization and Architecture: Designing for Performance”, Eighth Edition, Pearson.

**Theory:**

When CPU wants some page which is not in main memory then page fault occurs, the OS has to select a page to remove from main memory and make a place to get a new required page. The replacement algorithm decides which page to be replaced from main memory to a particular region to make the place available for the new page from virtual memory.

**LRU page replacement Policy**

This policy replace the page in the set which has experienced the fewest references i.e. not been referenced for the longest time. By the principal of locality, this should be the page least likely to be referenced in the near future. The algorithm is likely to maintain a linked list of all pages in a memory, with most recently used page at the front and the least recently used page at the rear.

**Conclusion:**

**Postlab Questions:**

1. Find page fault for LRU for page address stream 6 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 5,

no. of page frames=4.

1. Compare LRU and FIFO.

**Experiment No 11**

**Title: To study a Branch Prediction Controller**

**Objective: Branch prediction is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded.**

**References :** Computer Architecture and Organization –Zaky /Hamacher

Computer Architecture and Organization -Stallings

**Theory:**

In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed; the result is likely to be the same. Hence, the algorithm may be described by the two-state machine in Figure *a*.

The two states are:

LT: Branch is likely to be taken

LNT: Branch is likely not to be taken

This simple scheme, which requires one bit of history information for each branch instruction, works well inside program loops. Once a loop is entered, the branch instruction that controls looping will always yield the same result until the last pass through the loop is reached. In the last pass, the branch prediction will turn out to be incorrect, and the branch history state machine will be changed to the opposite state. Unfortunately, this means that the next time this same loop is entered, and assuming that there will be more than one pass through the loop, the machine will lead to the wrong prediction.

Better performance can be achieved by keeping more information about execution history. An algorithm that uses 4 states, thus requiring two bits of history information for each branch instruction, is shown in Figure *b*.

The four states are:

ST: Strongly likely to be taken

LT: Likely to be taken

LNT: Likely not to be taken

SNT: Strongly likely not to be taken

Again assume that the state of the algorithm is initially set to LNT. After the branch instruction has been executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT. As program execution progresses and the same instruction is encountered again, the state of the branch prediction algorithm continues to change as shown. When a branch instruction is encountered, the instruction fetch unit predicts that the branch will be taken if the state is either LT or ST, and it begins to fetch instructions at the branch target address. Otherwise, it continues to fetch instructions in sequential address order. It is instructive to examine the behavior of the branch prediction algorithm in some detail. When in state SNT, the instruction fetch unit predicts that the branch will not be taken. If the branch is actually taken, that is if the prediction is incorrect, the state changes to LNT. This means that the next time the same branch instruction is encountered; the instruction fetch unit will still predict that the branch will not be taken. Only if the prediction is incorrect twice in a row will the state change to ST. After that, the branch will be predicted as taken. Let us reconsider what happens when executing a program loop. Assume that the branch instruction is at the end of the loop and that the processor sets the initial state of the algorithm to LNT. During the first pass, the prediction will be wrong (not taken), and hence the state will be changed to ST. In all subsequent passes the prediction will be correct, except for the last pass. At that time, the state will change to LT. When the loop is entered a second time, the prediction will be correct (branch taken). We now add one final modification to correct the mispredicted branch at the time the loop is first entered. The cause of the misprediction in this case is the initial state of the branch prediction algorithm. In the absence of additional information about the nature of the branch instruction, we assumed that the processor sets the initial state to LNT. The information needed to set the initial state correctly can be provided by any of the static prediction schemes discussed earlier. Either by comparing addresses or by checking a prediction bit in the instruction, the processor sets the initial state of the algorithm to LNT or LT. In the case of a branch at the end of a loop, the compiler would indicate that the branch should be predicted as taken, causing the initial state to be set to LT. With this modification, branch prediction will be correct all the time, except for the final pass through the loop. Misprediction in this latter case is unavoidable.

The state information used in dynamic branch prediction algorithms may be kept by the processor in a variety of ways. It may be recorded in a look-up table, which is accessed using the low-order part of the branch instruction address.

