NAME : AMMAAR AHMAD
ROLL No. - 1801CS08
CS321 ENDSEM

1. Cache Memory
Needs
Throughout the development of computer to make it fast and cheaper, in last 50 years, we have made enormous progress. In 1970, the ~~ca~~ speed of accessing data from main memory and processor was comparable. But nowadays DRAM are far slower to access, around 10 ns to access data from main memory, compared to processor speed which is less than 1 ns. To bridge the gap between the different speed, we need memory which can keep ~~the~~ up with processor. So cache memory (SRAM) is used. One of the main issues with SRAM is that it is way too expensive and cannot replace the main memory completely. To keep the price bearable to consumers, small size cache memory is used along with DRAM.

Memory System Performance Analysis

Performance of memory is analysed by No. of misses in accessing data from data. Since DRAM is 100 times slower, even 2% decrease in hit rate from 99% to 97% can have huge ~~delay~~ difference in Average memory access time. So to have a better understanding, Miss rate is used instead of hit rate

$$\text{Miss Rate} = \frac{\text{Number of misses}}{\text{Number of total memory access}} \times 100\%$$

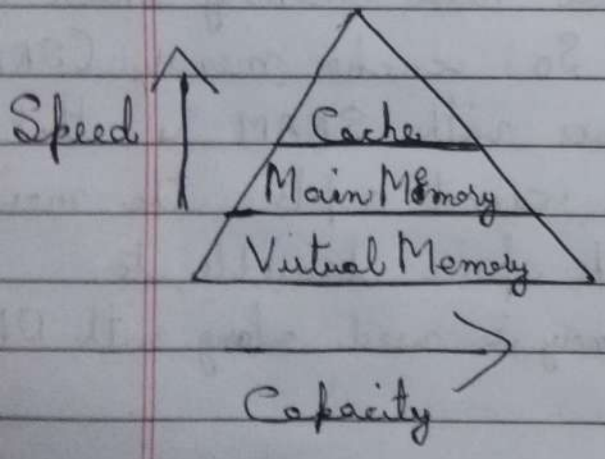
$$\text{Hit Rate} = 1 - \text{Miss rate}$$

Average memory access time (AMAT)

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}\, t_{VM})$$

t_{cache} ⇒ time to access cache memory
t_{MM} ⇒ time to access main memory
t_{VM} ⇒ time to access virtual memory (hard disk)
MR_{cache} ⇒ Miss Rate of Cache memory
MR_{MM} ⇒ Miss Rate of Main memory

Memory Hierarchy



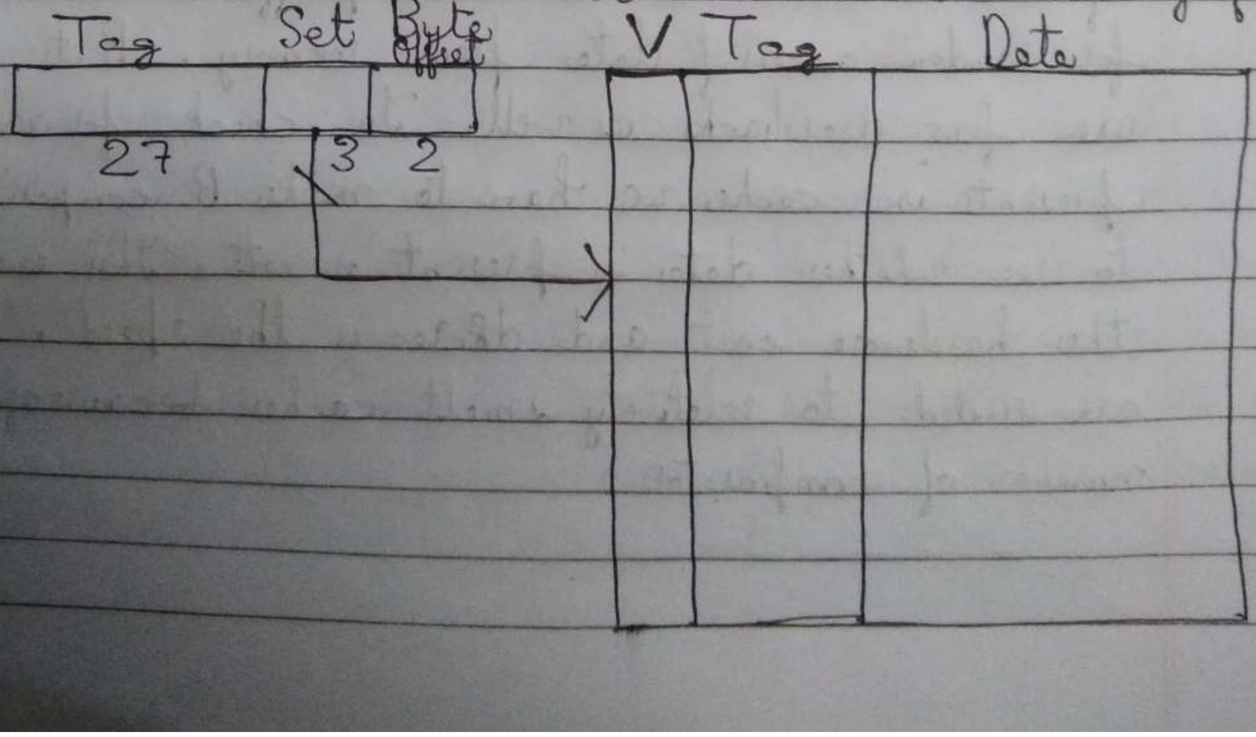
Technology	Cost/GB	Access Time
SRAM	$\sim 10^4$ \$	~ 1 ns
DRAM	~ 100\$	~ 100 ns
Hard Disk	~ 1 \$	$\sim 10^7$ ns

Data in cache
It stores the data need to ~~a~~ be accessed most frequently
To predict the access of data, it uses the knowledge of past pattern of memory access through two major locality :- temporal locality, and spatial locality

Temporal locality :- It states that the memory accessed by processor is likely ~~again~~ to be accessed again, hence it is copied from main memory to cache memory
Spatial locality :- It states that the data in near locality of currently used data is likely to be accessed in future. So the data block is copied to cache memory

Cache Associativity
There are 3 different ways to ~~[illegible]~~ design cache memory
(i) Direct Mapped Cache
(ii) Multi way Set Associative Cache
(iii) Fully Associative Cache

Direct Mapped Cache - ~~A~~ This design has one block for each set. Memory addresses of 32 bit system is mapped accordin to ~~last~~ n+1 to 2nd bits of memory leaving leaving last 2 bit in byte addressable memory
For example ⇒ If Cache has 8 sets then 3rd, 4th and 5th bit of address are used in mapping and first 27 bit are stored in tag of cache memory.
An extra bit V is stored to determined the validity of data



If the Set bits are same for 2 or more frequently used data, then ~~it~~ there is a clash and cache purpose is not well performed. To overcome this issue we have N way set associative cache

N way set associative cache - In this design, each set has N blocks for data, here N is a degree of associativity. Each block still maps to specific set but now it can map to any block in that set. In this way the clash is reduced and data from memory address having same set bit can be ~~used~~ ~~to~~ stored. This reduces the miss rates ~~for~~ than direct mapped cache of same capacity due to fewer conflicts. But, there is a use of extra multiplexer and comparator and this increases the cost of as well as increases the access time.

Fully associative cache - In this design, there is a single set and memory is divided to B blocks where ~~any~~ data from any address can be stored ~~in~~ on any of the blocks. This ~~is~~ is a prefferred cache design for random access of data from memory. But there are few drawbacks as well. To check for any data present in cache we have to make B comparisons to see whether data is present or not. This increases the hardware cost and decreases the speed. They are suited to relatively small caches because of large number of comparators

In the above 3 design, we only took note of temporal locality, to include spatial locality we need block size

Block Size
This uses the spatial locality, a cache uses large block to hold several consecutive words
The advantage of a block size greater than 1 is that when a miss occurs and the word is fetched into a cache, the adjacent words in the block is also fetched. Therefore, subsequent access are more likely to hit because of spatial locality. However a large block size means fewer blocks which may lead to more conflicts increasing the miss rate. ~~More~~ If the adjacent words in the block are not accessed later, the effort of ~~[illegible]~~ fetching them is wasted. Nevertheless most real programs benefit from larger block sizes.

Write Policy

Cache are classified as either write through or write back, In a write through cache, the data written to a cache block is simultaneously written to main memory. In a write back cache, a dirty bit (D) is ~~[illegible]~~ associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory when they are evicted ~~from~~ rom cache. A write through require no dirty bit but requires more main memory and hence not used in modern caches

2. Implementing Jump Instruction and Controlling Hazards

In the implementation of jump instruction we have the instruction J 26 bit address
We can calculate the address of jump instruction by ~~P~~ taking first 4 bits of PC and 26 bits + "00" by the end of decode stage. We have our next address by decode stage. So rather than loading any other instruction we can flush out the next loaded instruction if jump instruction occurs. There need to only flushing of last instruction loaded because, by the decode stages we know that it's jump instruction and PC is set accordingly

changes in the circuit

In the already implemented MIPS Pipelined Processor In the decode stage calculating the possible jump address using first 26 bits of instruction and 4 bit of PC, an ~~extra~~ JumpID is added in econtrol unit. ~~We~~ When JumpID is 1 we jump to the particular address, when JumpID is 0 normal processing continues.
For flushing, jumpID is OR with Branch to flush out the just previous instruction which was in fetch stage. With all instructions taking at most 2 cycles (including stalling and flushing), this gives good CPI of 1<CPI<2

3. Implementing 3 more instructions in Single Cycle MIPS processor
(i) lui
(ii) ori
(iii) andi

To implement the following instruction ALUSrc was increased from 1 to 2 bits and ALUOp was increased from 2 to 3 bits.

Main Truth Table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Br	MW	MtR	ALUOp
R type	000000	1	1	00	0	0	0	010
lw	100011	1	0	01	0	0	1	000
Sw	101011	0	X	01	0	1	X	000
beq	000100	0	X	00	1	0	X	001
addi	001000	1	0	01	0	0	0	000
j	000010	0	X	XX	X	0	X	XXX
lui	001111	1	0	10	0	0	0	000
ori	001101	1	0	01	0	0	0	100
andi	001100	1	0	01	0	0	0	011

ALUOp	Function	ALU Con
000	X	010 (add)
001	X	110 (subtract)
010	100000	010 (add)
010	100010	110 (subtract)
010	100100	000 (and)
010	100101	001 (or)
010	101010	111 (set less than)
~~010~~		
011	X	000 (and)
100	X	001 (or)

An extra 16 bit left shifter was required to shift lower 16 bits to upper 16 bit in lui instruction

4. Amdahl's law set ~~up~~ the upper limit to the speedup that is possible. Given the program which is X percentage parallelizable, it say the speed up curve tends to flatten out as the number of processors used for parallelisation is increased.

$$\text{Speedup} = \frac{1}{1-f+\frac{f}{p}}$$

$f \Rightarrow$ fraction of program parallelizable
$p \Rightarrow$ No. of processors

This Amdahl's law has a great deal of application.
~~Code~~
Code :

```
for (i=0; i<32; i++)              => Not Parallelizable
    a[i] = a[rand()*31];
for (i=0; i<32. i++)              => P1
    c[i] = a[i] ^ b[i];
for (i=0. i<32, i++)              => P2
    d[i] = a[i] - b[i];
for (i=0; i<32, i++)              => P3
    e[i] = a[i] * b[i];
```

If $f = 0.25$ (Parallelizing P3)

$p=2 \Rightarrow \text{Speedup} = \frac{1}{0.875} = \frac{8}{7} = 1.14$

$p=4 \Rightarrow \text{Speedup} = \frac{1}{3/4+1/16} = \frac{16}{13} = 1.23$

$p=8 \Rightarrow \text{Speedup} = \frac{1}{3/4+1/32} = \frac{32}{25} = 1.28$

If $f = 0.5 \Rightarrow$ Parallelizing P2, P3

$P=2 \Rightarrow \text{Speedup} = \frac{4}{3} = 1.33$

$P=4 \Rightarrow \text{Speedup} = \frac{8}{5} = 1.6$

$P=8 \Rightarrow \text{Speedup} = \frac{16}{9} = 1.82$

$P=16 \Rightarrow \text{Speedup} = \frac{32}{17} = 1.88$

If $f = 0.75 \Rightarrow$ Parallelizing P1, P2, P3.

$P=2 \Rightarrow \text{Speedup} = \frac{1}{1/4+3/8} = \frac{8}{5} = 1.6$

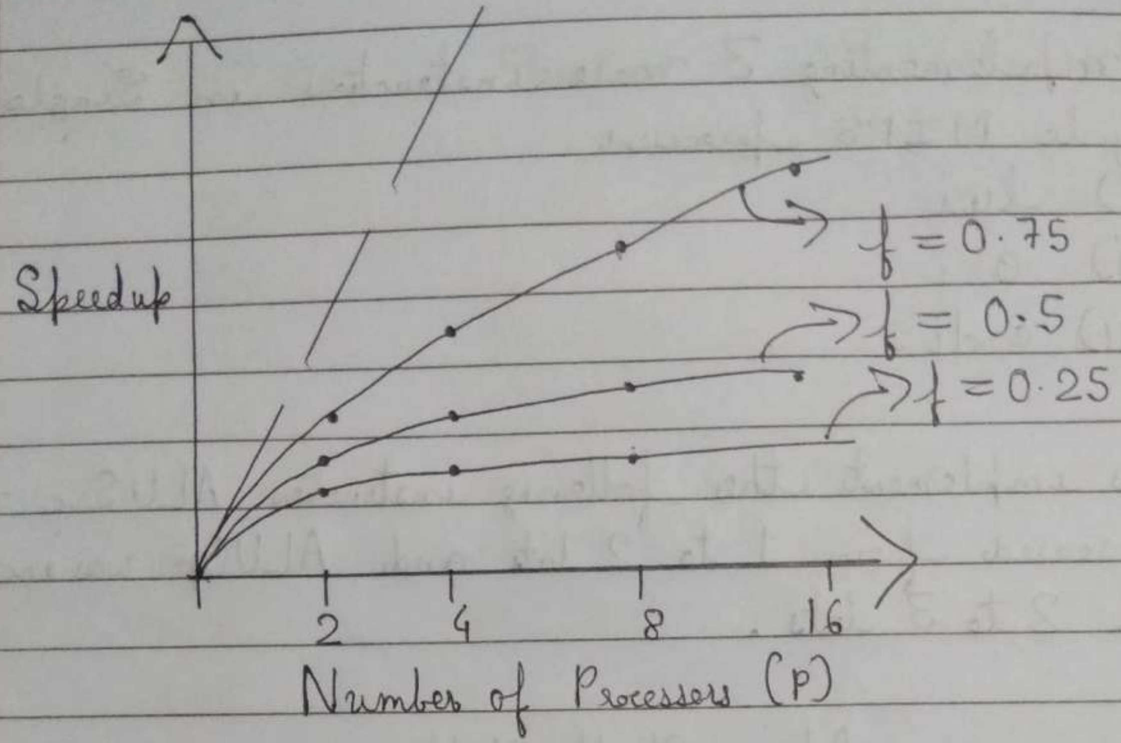
$P=4 \Rightarrow \text{Speedup} = \frac{1}{1/4+3/16} = \frac{16}{7} = 2.28$

$P=8 \Rightarrow \text{Speedup} = \frac{1}{1/4+3/32} = \frac{32}{11} = 2.89$

$P=16 \Rightarrow \text{Speedup} = \frac{1}{1/4+3/64} = \frac{64}{19} = 3.37$

$P=32 \Rightarrow \text{Speedup} = \frac{1}{1/4+3/128} = \frac{128}{35} = 3.66$

As we can see from the values of Speedup from different f we see it is approaching it's asymptotic value



All three curves seems to flatten out as we increase the number of processors. This indicates that hardware can improve performance only upto certain extent. It is the algorithm to solve the problem which affects the performance significantly

Since the entire code of the program is generally not parallelizable. We can only parallelize some parts of program. So the ideal parallelizable efficiency is not achieved, and practically achievable parallelization is always less.
But it is also possible that the already parallelised code sections are not getting full speedup, contrary to our assumption. ~~&~~ There are different reasons for this issue.