CS321 – Computer Architecture

Name: M. Maheeth Reddy	**Roll No.**: 1801CS31	**Date**: 27 November 2020

Autumn 2020 – End Semester Examination

Ans 1:

Cache Memory

Cache Memory is a high-speed memory used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. It is a smaller yet faster memory, located close to a processor core. It is implemented using SRAM. It stores copies of the data from frequently used main memory locations. Most CPUs have a hierarchy of multiple cache levels (L1, L2 and L3). The L1 cache is further divided into instruction-specific and data-specific caches.

Use of Cache Memory

Generally, SSDs, HDDs are used for long-term storage of data and RAM is used to store data and program instructions that the central processing unit requires in the near future. The CPU operates at a very high speed when compared to rate of data retrieval from these devices. So, this causes a performance bottleneck between the CPU and the RAM, and the overall performance of the computer system reduces.

To prevent this from happening, computer systems are commonly equipped with **cache memory**. This cache memory **stores data or instructions** that the CPU is **likely to use in the immediate future**. The cache memory is located very close to the CPU, either on the motherboard in the immediate vicinity of the CPU or on the CPU chip itself and connected by a dedicated data bus. So, instructions and data can be read from it and written to it much more quickly than is the case with normal RAM. So, a small amount of cache memory can result in a significant increase in the computer's performance.

How Does Cache Memory Work?

Cache memory works by taking data or instructions at certain memory addresses in RAM and copying them into the cache memory, along with a record of the original address of those instructions or data.

This results in a table containing a small number of RAM memory addresses, and copies of the instructions or data that those RAM memory address contain.

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

If the processor finds that the memory location is in the cache, a cache hit has occurred and data is read from cache

If the processor does not find the memory location in the cache, a cache miss has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

Cache Hierarchy

Modern computer systems have more than one piece of cache memory, and these caches vary in size and proximity to the processor cores, and therefore also in speed. There are three cache levels: L1 cache, L2 cache and L3 cache.

Level 1

L1 cache is cache memory that is built into the CPU itself. It runs at the same clock speed as the CPU. It is the most expensive type of cache memory so its size is extremely limited. But because it is very fast it is the first place that a processor will look for data or instructions that may have been buffered there from RAM.

In fact, in most modern CPUs, the L1 cache is divided into two parts: a data section (L1d) and an instruction section (L1i). These hold data and instructions, respectively.

A modern CPU may have a cache size on the order of 32 KB of L1i and L1d per core.

Level 2

L2 cache may also be located in the CPU chip, although not as close to the core as L1 cache. Or more rarely, it may be located on a separate chip close to the CPU. L2 caches are less expensive and larger than L1 caches, so L2 cache sizes tend to be larger, and may be of the order of 256 KB per core.

Level 3

Level 3 cache tends to be much larger than either L1 or L2 cache, but it also different in another important way. Whereas L1 and L2 caches are private to each core of a processor, L3 tends to be a shared cache that is common to all the cores. This allows it to play an important role in data sharing and inter-core communication. L3 cache may be of the order of 2 MB per core.

Cache Associativity / Cache Mapping

Cache memory, is extremely fast – meaning that it can be read from very quickly.

But there is a potential bottleneck: before data can be read from cache memory, it has to be found. The processor knows the RAM memory address of the data or instruction that it wants to read. It has to search the memory cache to see if there is a reference to that RAM memory address in the memory cache, along with the associated data or instruction.

There are a number of ways that data or instructions from RAM can be mapped into memory cache, and these have direct implications for the speed at which they can be found. But there is a trade-off: minimizing the search time also minimizes the likelihood of a cache hit, while maximizing the chances of a cache hit maximizes the likely search time.

The following cache mapping methods are commonly used:

Direct Mapping

In Direct mapping, each memory block is assigned to a specific line in the cache. When a new block needs to be loaded in a line already used by a memory block, the old block is trashed and new one is loaded.

The address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

Cache line number = (Main memory block number) % (Number of lines in the cache)

The drawback with direct mapped cache is that it severely limits what data or instructions can be stored in the memory cache, so cache hits are rare.

Associative Mapping

It is also known as fully associated mapping. In this mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word ID bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form since the chances of a cache hit are much higher.

Set-Associative Mapping

Set associative cache mapping combines the best of direct and associative cache mapping techniques. It addresses the problem of possible thrashing in the direct mapping method.

This is done by allowing a block to map to a group of lines which are termed as set. Then a block in memory can map to any one of the lines of a specific set. Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address.
In this case, the cache consists of a number of sets, each of which consists of a number of lines.

Number of lines in the cache number of sets	=	Number of sets	*	Number of lines in each set
Cache Set Number	=	Main Memory Block Number	%	Number of sets

Comparison of Different types of Cache Misses in the above Mappings

	Direct mapped	Set associative	Fully associative
Cache size	Large	Medium	Small
Compulsory Miss	Same	Same	Same
Conflict Miss	High	Medium	Zero
Capacity Miss	Low	Medium	High
Coherence Miss	Same	Same	Same

Data Writing Policies

Data can be written to memory using a variety of techniques, but the two main ones involving cache memory are:

- **Write-through.** Data is written to both the cache and main memory at the same time.

- **Write-back.** Data is only written to the cache initially. Data may then be written to main memory, but this does not need to happen and does not inhibit the interaction from taking place.

Pros and Cons of Write Policies:

- Write-through read misses do not need to be written back to evicted line contents.
- Write-back involves no repeated writes.
- Write-through is always combined with write buffers so that it does not wait for lower level memory.

Ans 2:

Handling of J instruction in MIPS Pipelined Processor

J instruction: is the unconditional jump instruction in the MIPS instruction set. It is a J-type instruction and takes a 26-bit operand as input. It performs a branching to another instruction present at the address specified by the operand.

The execution of R-type and I-type instructions happens serially or in a sequential manner. This is not the case for the instructions like **J**, which alter the control flow of the program. These instructions alter the Program Counter (PC) according to the destination address specified. Other instructions with similar behaviour are Branch Instructions (BEQ, BNE, BGTZ etc.), Jump and Link.

Execution of instructions in MIPS Pipelined Architecture:

In MIPS, instructions are executed in multiple cycles. Their execution generally comprises of the following phases:

- Fetch
- Decode
- Execute
- Memory
- Writeback

The MIPS Pipelined Architecture is specifically designed for parallel execution of sequential instructions. When an instruction, say A0 undergoes its Execute phase, two other instructions are also getting executed parallelly. One of the instructions, say A1 would be in its Decode Phase while the other one, say A2 would just begin its Fetch Phase. So, when all phases of A0's execution are done, A1 and A2 are the next ones in line waiting to be executed, and another instruction A3 would begin its Fetch phase.

In case of J instruction, only Fetch, Decode and Execute phases occur. When J instruction is executed, the PC value is changed and control goes to a different location in the program. But there is a problem. When J instruction is executed, the next two instructions A1 and A2 are still in line for execution. But, since the control has jumped, we would expect another instruction, say A15, to line up for execution and A1 and A2 must be cleared from the queue. This situation is called a Hazard and must be dealt with to avoid erroneous program execution.

In the MIPS Pipelined Processor, Hazards can be controlled using the following three methods:

1. Data Forwarding
2. Stalling
3. **Flushing**

Flushing is used to deal with hazards caused due to Jump instruction.

When the J instruction is in the Decode Phase, the immediate next instruction in memory, A1 will be in the Fetch Phase. This instruction can be FLUSHED from the Fetch Phase by substituting an NOP instruction into its place.

NOP instruction is implemented using **sll $0, $0, 0** instruction. After this, we could introduce a BUBBLE into the pipeline which will function like a Single Cycle Delay in performing the JUMP.

This is how the MIPS pipelined processor is extended to handle the J instruction.

Ans 3:

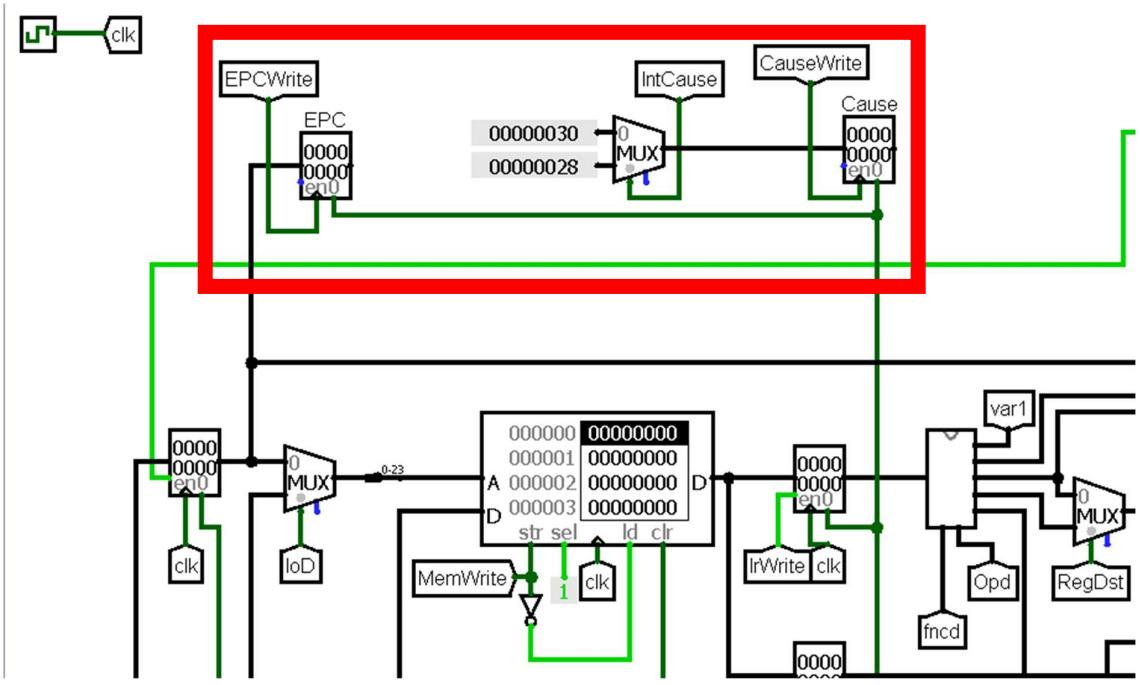
Exception Handling in MIPS Multi-Cycle Architecture

Aim: Implement and demonstrate Exception Handling in MIPS Multi-Cycle Architecture

Method of Solving:

A separate block consisting of two registers for Exception handling is implemented. We inculcate extra logic in the ALU of the MIPS processor, in order to generate the signals, IntCause, CauseWrite, EPCWrite which are essential for the aim of the experiment.

Solution:



I have used the MIPS Multi-Cycle Processor provided in Lab 9 of CS322 Course. In this experiment I have extended its working for handling of exceptions.

In this experiment, I have handled the **Arithmetic Overflow** exception.

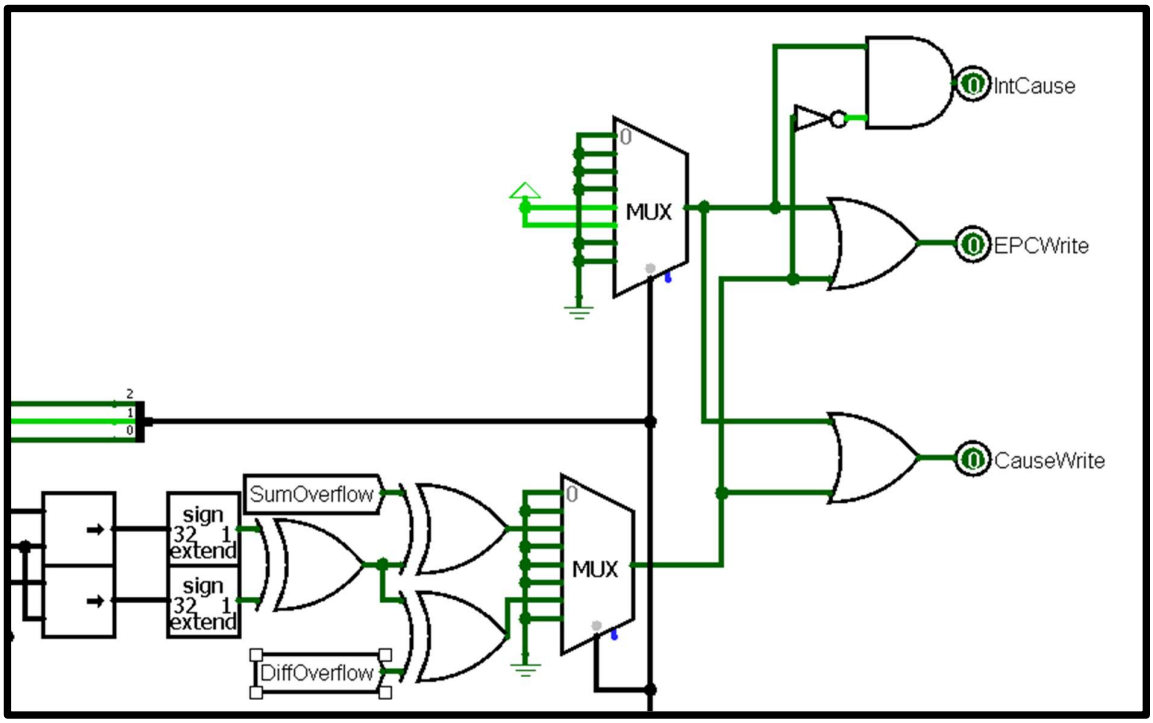
Arithmetic Overflow:

This exception occurs whenever the result of an instruction incorrectly produces the opposite sign. In other words, it occurs when:

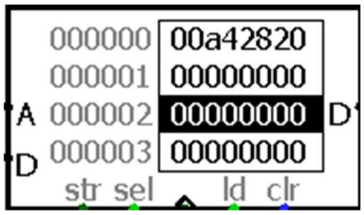
- Adding two huge positive numbers gives a negative result
- Adding two huge negative numbers gives a positive result

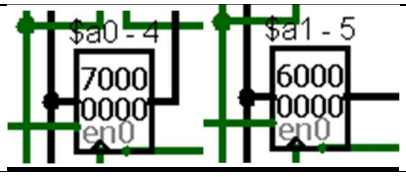
This is called overflow and is detected using the following procedure:

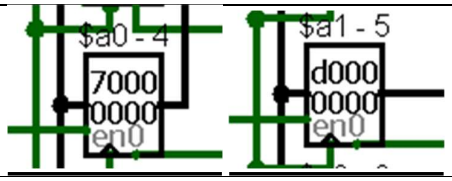
- Calculate XOR of MSB's of both operands
- Calculate XOR of MSB of Result and Carry Out Bit
- Calculate XOR of the results obtained
    - If result is 1, then there is overflow
    - Otherwise, no overflow



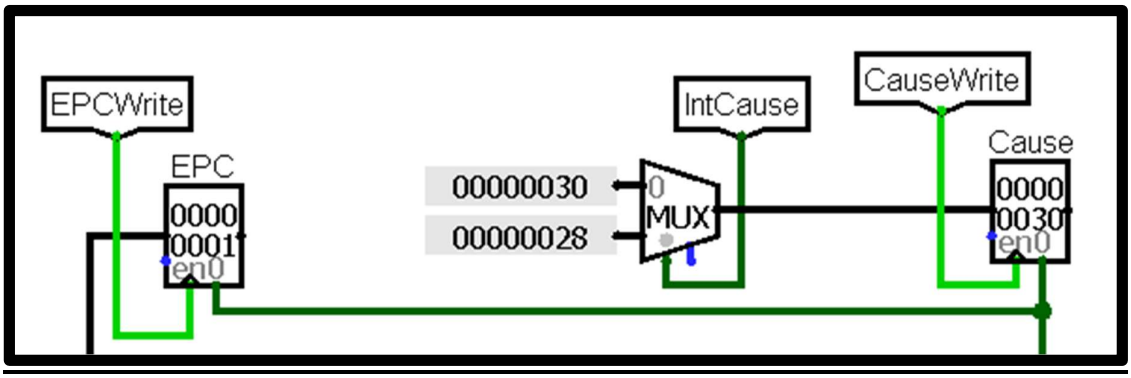
To demonstrate the Exception, I used the following instruction: add a1 a1 a0



Register Values Before Execution	

Register Values After Execution	

Notice the Exception PC and Cause register got loaded on 3rd cycle of execution



Ans 4:

Amdahl's Law

This law was proposed in 1967 by Gene Amdahl, a computer architect from IBM and Amdahl Corporation. It is also known as ***Amdahl's Argument***.

It is a formula which gives the theoretical speedup in terms of latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved.

In simpler words, it is a formula used to find the maximum improvement possible by just improving a particular part of a system. It is often used in *parallel computing* to predict the theoretical speedup when using multiple processors.

Speedup is defined as the ratio of performance for the entire task using the enhancement and performance for the entire task without using the enhancement or speedup can be defined as the ratio of execution time for the entire task without using the enhancement and execution time for the entire task using the enhancement.

Speedup is given by the formula

$$Speedup = \frac{1}{1 - f + \frac{f}{P}}$$

where, **f** is the Parallelizable fraction of the program

P is the number of Processors

Sample Program

I have written an MIPS program to calculate the square of a number. The square of the number in register a0 is calculated through repeated addition. Initially, the register t0 is initialized to

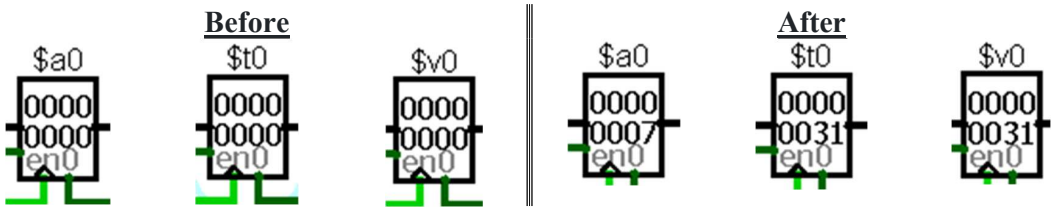
zero. Another register t1 counts the remaining number of loops to be executed. a0 is repeatedly added to t0 in a loop until t1 becomes zero.

In this program a0 is 7, so finally value in t1 will be 49 which is returned to a calling function through v0. I have executed the program in the Single Cycle Architecture provided in Lab8 of CS322 course.

The program is shown below:

Label	Instruction	Machine Code
square:	addi $t0, $0,0	0x20080000
	addi $a0, $0,7	0x20040007
	add $t1, $a0, $0	0x00804820
loop:	beq $t1, $0, exit	0x11200003
	add $t0, $t0, $a0	0x01044020
	addi $t1, $t1, -1	0x2129FFFF
	j loop	0x08000003
exit:	add $v0, $t0, $0	0x01001020

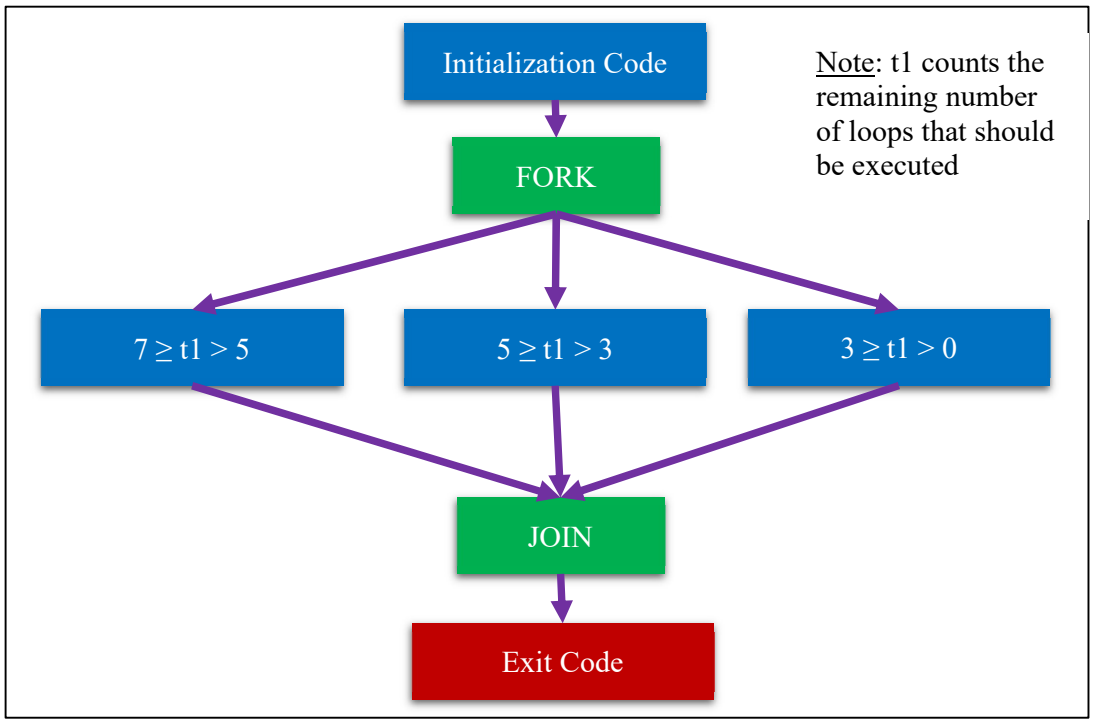
Screenshot of Program Execution



How is Parallelization achieved in this Program

Consider there are 3 processors P0, P1, P2. The loop in this program that calculates the square does the following calculation repeatedly, adding t0 with a0 (add $t0, $t0, $a0) 7 times. These calculations are clearly independent of each other. Hence, we can parallelize this loop.

So, we could perform 2 iterations of the loop in P0, 2 more in P1, and remaining 3 in P2. So, in this way the program is parallelizable.



Calculation

The loop part is parallelizable portion of this code. There are 4 instructions in the loop and each of them are executed 7 times.

$$Speedup = \frac{1}{1 - f + \frac{f}{P}}$$

$$\text{Here, } f = \frac{Number\ of\ cycles\ for\ independent\ instructions}{Total\ number\ of\ executed\ cycles}$$

Number of cycles for independent instructions = 7 * 4 = 28

Total number of cycles = 32

Hence f = $\frac{28}{32}$ = 0.875

So, to plot the graph of **number of processors vs. speedup** shown below, the formula to be used is

$$Speedup = \frac{1}{0.125 + \frac{0.875}{P}}$$

Graph

On Y-axis: Speedup

On X-axis: P

Observation:

It is observed that as the number of processors increases, the increase in speedup value decreases.

Inference:

In the above program, I distributed the execution of 7 cycles among 3 processors. Each of them executed 2,2,3 loops respectively. If there were 7 processors, then we could execute one loop in each processor and we could execute the program in one cycle itself. This is the maximum speed we can achieve. If we increase the number of processors to 8 and above, there would be no change in the speed of execution. This is what we are observing in the graph too.