CS 321-End Semester

Name: P. V. Sriram

Roll No.: 1801CS37

Q1) Write a critical analysis report on design strategy of Cache Memory for Computer performance Improvement: including Cache Hierarchy, Cache Associativity and Cache Write Policies: Pros & Cons.

Ans)

Cache Memory

Cache memory is a chip-based computer component that makes retrieving data from the computer's memory more efficient. It acts as a temporary storage area that the computer's processor can retrieve data from easily. This temporary storage area, known as a cache, is more readily available to the processor than the computer's main memory source, typically some form of DRAM.

Cache memory is sometimes called CPU (central processing unit) memory because it is typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU. Therefore, it is more accessible to the processor, and able to increase efficiency, because it's physically close to the processor

Need for a cache

CPU operates at the MIPS (Million instructions per second) level. While the Hard disks (HDDs and SSDs) where the long-term memory is stored, and Main memory (RAM) where the data and instructions required immediately, are stored, operates at a much slower speed.

This performance difference heavily reduces the efficiency of a processor as the data transfer are taking orders of magnitudes more time than the actual instructions themselves.

To avoid such a bottlenecking, architectures now-a-days are equipped with a cache memory. Cache stores the instructions which are most likely to be requested by the CPU contemporarily.

Performance

Cache memories are typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU. Therefore, it is more accessible to the processor, and able to increase efficiency, because it's physically close to the processor.

In order to be close to the processor, cache memory needs to be much smaller than main memory. Consequently, it has less storage space. It is also more expensive than main memory, as it is a more complex chip that yields higher performance.

What it sacrifices in size and price, it makes up for in speed. Cache memory operates between 10 to 100 times faster than RAM, requiring only a few nanoseconds to respond to a CPU requests.

Working

Cache memory takes data or instructions from a certain location of RAM and makes a copy along with information about original addresses. Typically, this copying is done in a word level, creating tables of the addresses and RAM data.

Whenever there is a requirement of data from the processor, first the cache is checked for that particular address. If that memory exists in cache, a “Cache Hit” has occurred and data is transferred. Else a “Cache miss” has occurred and the cache allocates a new entry and copies in data from main memory and then transferred to the processor to fulfil the requirement.

Cache Hierarchy

Cache memory is fast and expensive. Traditionally, it is categorized as "levels" that describe its closeness and accessibility to the CPU. There are three general cache levels:

L1 cache:

- L1 cache or primary cache is the fastest in terms of speed and accessability
- It is usually embedded in the processor chip as CPU cache and hence is closest to the CPU and it runs at same clock speed as the CPU.
- Most expensive of the bunch, hence the smallest

L2 cache:

- L2 cache or secondary cache is the next furthest in terms or distance from the processor and accessibility and hence the speed.
- It can either be on a separate chip (co-processor) and have a high-speed alternative system bus connecting the cache and CPU.
- Less expensive than L1 cache and bigger than L1

L3 cache:

- L3 cache or tertiary cache is specialized memory developed to improve the performance of L1 and L2.
- L1 or L2 can be significantly faster than L3, though L3 is usually double the speed of DRAM. With multicore processors, each core can have dedicated L1 and L2 cache, but they can share an L3 cache.
- If an L3 cache references an instruction, it is usually elevated to a higher level of cache.

Cache Associativity

The performance of cache can be improved using Cache associativity or cache mapping.

There are three different types of mapping

1) Direct Mapping
2) Associative mapping
3) Set-Associative mapping

- In direct mapping, the mapping happens from each block of main memory into only

a single line in the cache. If a line which is pre occupied, has to be loaded with new data for the newer instructions, then the old data is deleted. An 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.

- Fully associative cache mapping is similar to direct mapping in structure but allows a memory block to be mapped to any cache location rather than to a prespecified cache memory location as is the case with direct mapping.

- Set associative cache mapping can be viewed as a compromise between direct mapping and fully associative mapping in which each block is mapped to a subset of cache locations. It is sometimes called N-way set associative mapping, which provides for a location in main memory to be cached to any of "N" locations in the L1 cache.

Data writing policies

Data writing policies deal with the methods of writing into the memory. There are two main types:

Write-through:

- Data is written to both the cache and main memory at the same time.
- Pros: Ensures fast retrieval while making sure the data is in the backing store and is not lost in case the cache is disrupted.
- Cons: Writing data will experience latency as you have to write to two places every 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: Low latency and high throughput for write-intensive applications.
- Cons: There is data availability risk because the cache could fail (and so suffer from data loss) before the data is persisted to the backing store. This result in the data being lost.

Q2) Explain how to extend the MIPS pipelined processor to handle the j instruction (jump

instruction). Give particular attention to how the pipeline is flushed when a jump takes place.

Ans)

Jump instruction

In MIPS there are three instruction formats. R-Type, I-Type and J-Type formats. Here, apart from the J type instructions, other instructions execute in a sequential manner. i.e) In the sequence of the Program Counter. But the control flow instructions such as the jump register, jump and link and the branch instructions, alter the PC according to the programmer needs. What this essentially means is that, for a pipelined architecture, which is specifically tailored to work best for sequential instruction will face a hazard and it has to be dealt with

The Jump instruction happens in three stages where in the jump instruction is first fetched, then it is decoded and finally it shall be executed. In the last stage of the machine cycle, a new address, which is obtained from the instruction (last 26 bits) is put into the Program Counter and instead of simply incrementing the PC by 4. The processor essentially jumps to another place in the program.

Hazard caused

While the Fetch, Decode and Execute of the jump operation were being executed, in the pipelined model, the next two instructions (I1, I2) which are sequentially after the J instruction already start their fetch and decode stages.

While this initially isn't an issue, when the J instruction is executed and we jump to a new PC, we expect the new instruction (I3) to start from there freshly. However, due to the \ design of pipelining there are already two instructions (I1, I2) are already in the queue and are yet to be executed before the actual instruction (I3).

This is not what a programmer expects. As the program is supposed to execute I3 while avoiding I1, I2.

Hazard Control

There are many such hazards which commonly occur in the pipelined architecture. Which is why there is an individual Hazard Unit present in the pipeline architecture.

Methods:

1) Flushing
2) Branch prediction
3) Stalling

Flushing

When the jump instruction is passed into the decode phase, the instruction which is immediately after the jump instruction in the sequence shall be in the fetch stage. We can flush this instruction from the fetch phase by replacing it with a NOP instruction.
MIPS uses sll $0, $0, 0 as the nop instruction (Encoding of all zeros). Essentially, we introduce a bubble into the pipeline which acts like a single cycle delay in performing the jump.

Q3) Design an experiment which conveys one of the concepts in CS321. Clearly state the aim of the experiment, method of solving, and solution. Create a logic-sim set up and simulate the same to convey the concept and include screen shots in the assignment file

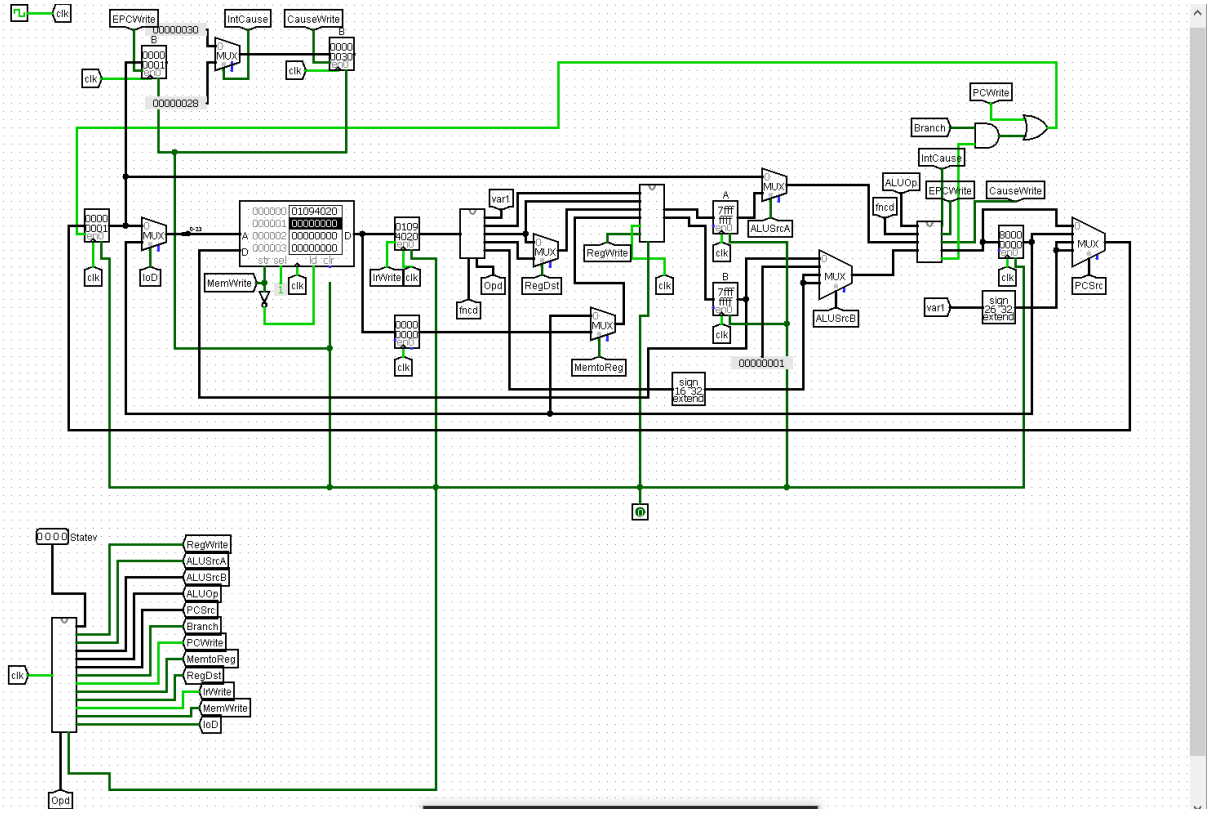
Ans)

Aim:

To implement and demonstrate the exception handling in Multi Cycle MIPS Processor.

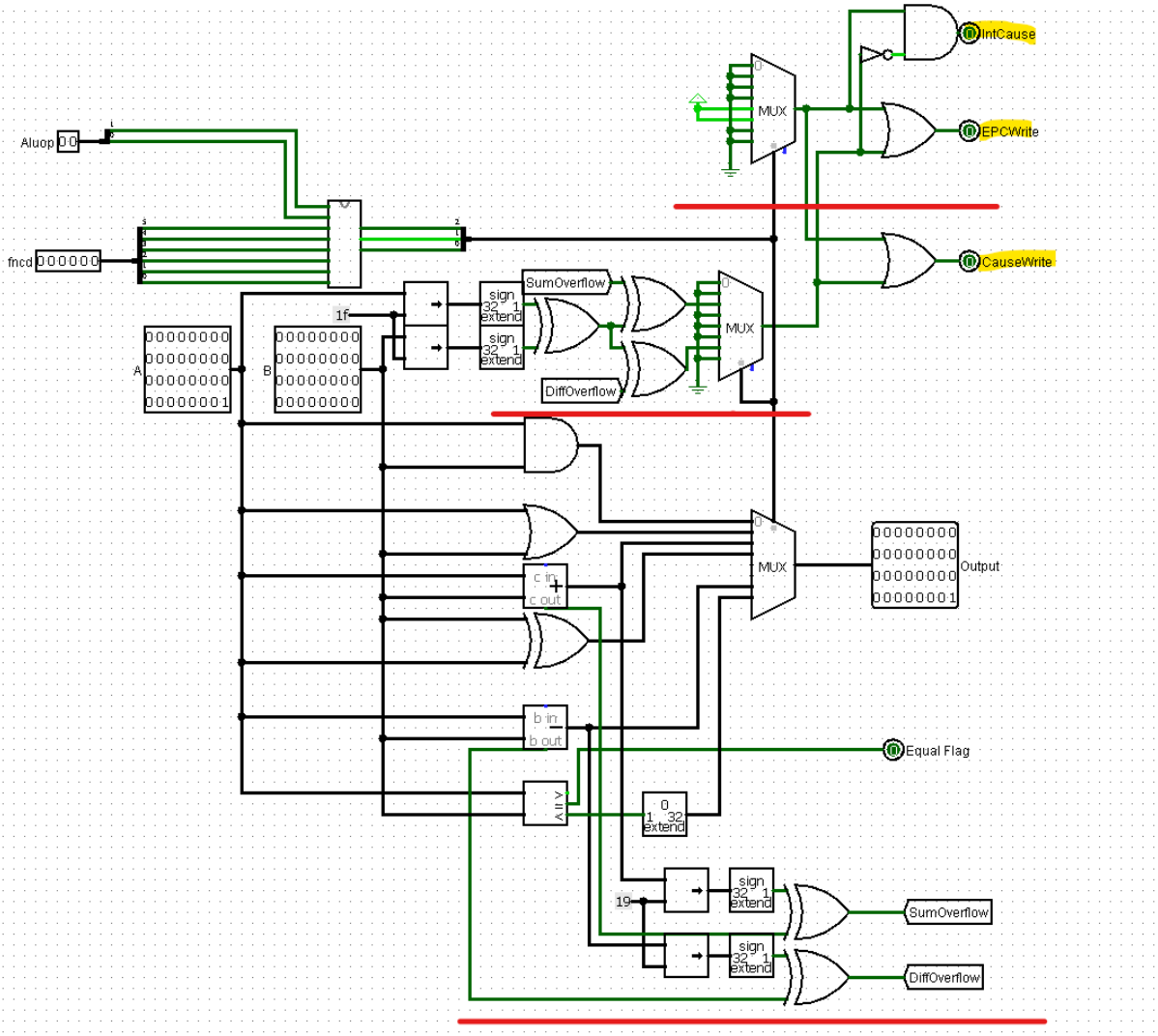
Arithmetic overflow occurs during the execution of an add, addi, or sub instruction. If the result of the computation is too large or too small to hold in the result register, the Overflow output of the ALU will become high during the execute state. This event triggers an exception.

Undefined instruction occurs when an unknown instruction is fetched. This exception is caused by an instruction in the IR that has an unknown opcode or an R-type instruction that has an unknown function code.



Method of Solving:

We insert an extra logic in the Arithmetic logic Unit in order to implement triggers for “Arithmetic Overflow” and “Undefined Instruction”.



For Arithmetic Overflow, we take add additional logic to find out whether if an arithmetic operation has a bit overflow. If there is one, the EPCWrite and CauseWrite are triggered.

In general, an overflow during addition or subtraction is done using the following steps.

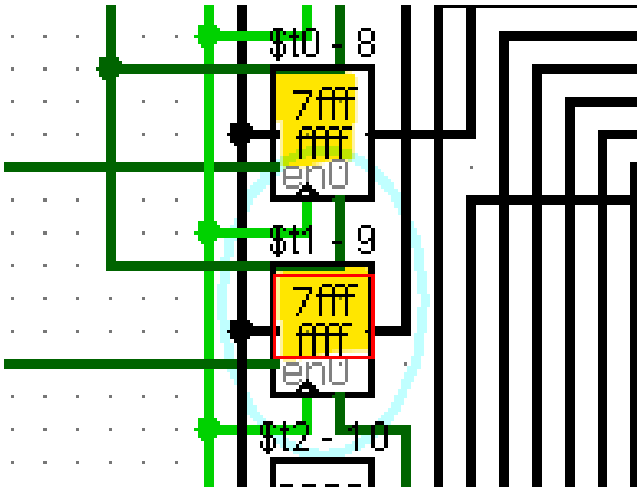
1) XOR of the most significant bits for operands. (x1)
2) XOR of most significant bit of result of the arithmetic operation and carry out (x2).
3) XOR of x1 and x2 will give the status of the overflow.

E.g. Addition of the numbers 0x7fffffff and 0x7fffffff gives an overflow, and we shall verify in the circuit

Code:

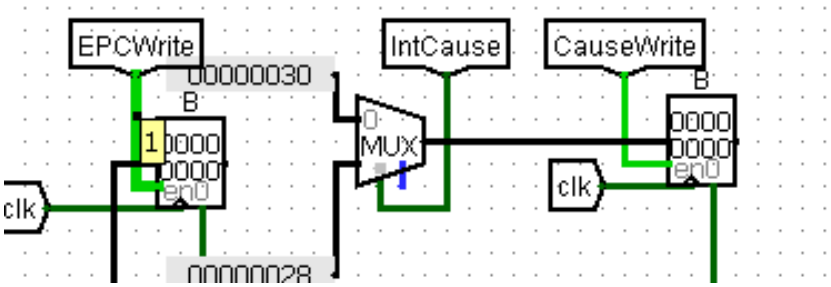
add $t0, $t0, $t1 : 0x01094020

Before



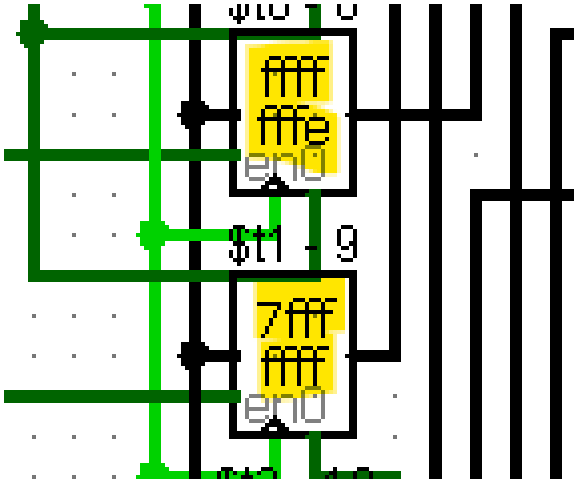
We initially load t0 and t1 with the value 0x7fffffff

While



As we proceed through the cycles, we can see that the EPC write and Cause Write signals were activated. Indicating an overflow. Cause is mapped to 0x30.

After



We can see that the sum (overflow not considered) in stored finally in the $t0 register.

For undefined operations, it is assumed that they shall be mapped to 4, 5 locations of the mux on ALU. Correspondingly, if the opcode activates those states, we render the triggers EPC Write and Cause write active. And we can see the Cause showing 0x28 then.

Q4) Amdahl's law assumes that the problem size is kept constant- by adding more processors the same problem gets solved faster. Assuming a sequential program with appropriate parameters (with X portion that can be parallelized). Do a case study analysis. Include graphical analysis in your study (for example number of processors vs. speed up etc.). Include the program that you are writing for the analysis in the answer sheet.

Ans)

Amdahl's law

Amdahl's Law was proposed by a computer scientist named Gene Amdahl. This law which is also referred to as the Amdahl's argument gives the theoretical speedup in latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved. In other 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.

$$\boldsymbol{Speedup} = \frac{\mathbf{1}}{(\mathbf{1} - \boldsymbol{f}) + \frac{\boldsymbol{f}}{\boldsymbol{p}}}$$

Where f is the parallelizable fraction of a program and P is the number of processors.

We can visualise this law on a single cycle MIPS processor by using a parallelizable MIPS code. The sample code is described as follows.

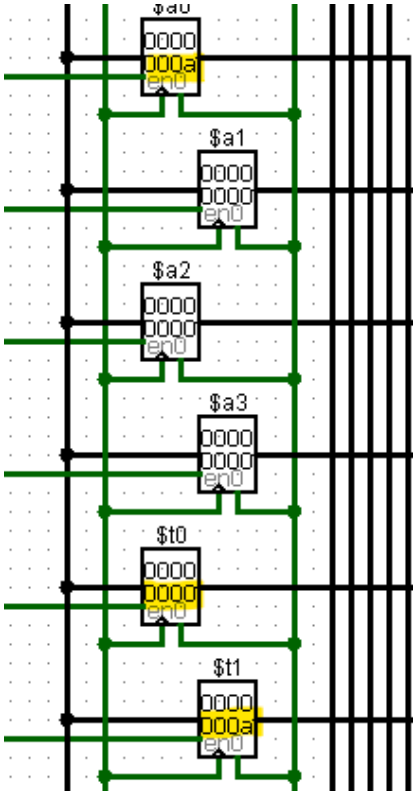
Sample Program

I have written a program which finds the sum of first N natural numbers. The code essentially takes an accumulator and a count and adds the count to the accumulator in a loop while decrementing count at each step.

Code:

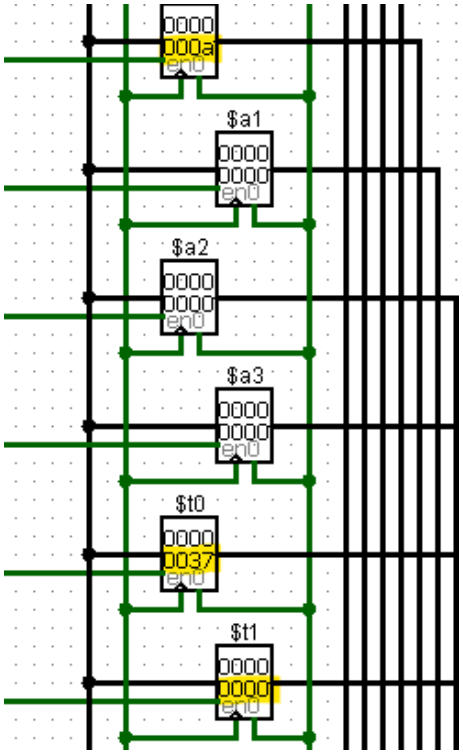
addi $t0, $0, 0	0x20080000
addi $a0, $0, 10	0x2004000a
add $t1, $a0, $0	0x00804820
loop: beq $t1, $0, exit	0x11200003
add $t0, $t0, $t1	0x01094020
addi $t1, $t1,-1	0x2129FFFF
j loop	0x08000003
exit: addi $v0, $t0, 0	0x01001020

Before:



We can see that 10 is loaded in $t1 and $a0 and $t0 is the accumulator

After:



We can see that the answer, 55 is stored in $t0 (0x37 = 55)

Parallelizability of Code

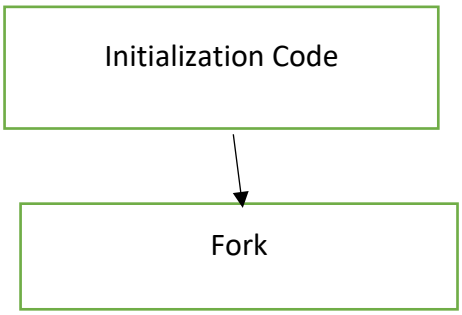
This program essentially consists of three parts:

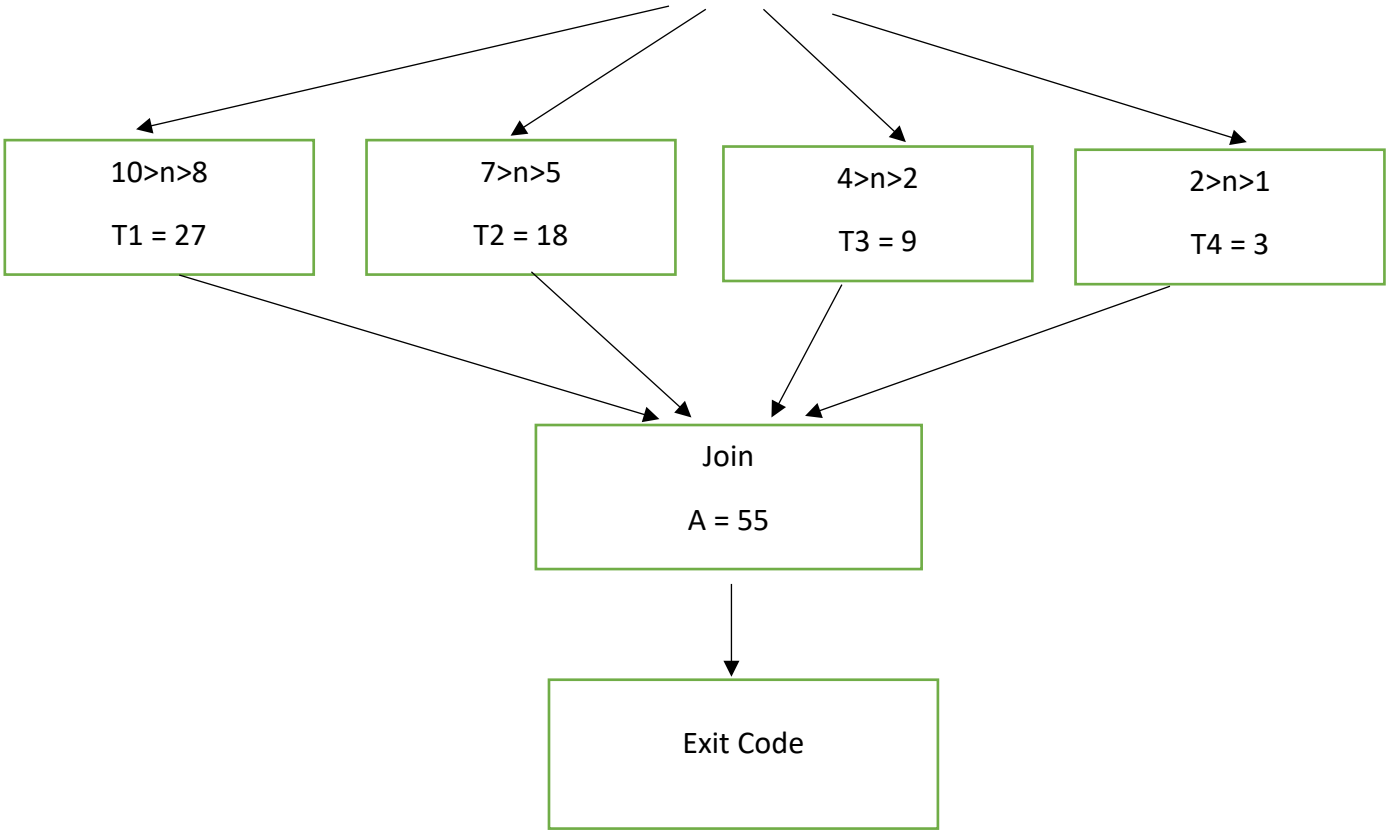
1) The initialization
2) The loop
3) The completion

In the initialization is basically loading the arguments into t0(accumulator), t1(count), a0(input) to their respective values.

In the loop, we add a count variable to the accumulator while decrementing the count for each loop. There are 10 such loops here, and we can see that all the programs here can be represented as independent programs as, if we take n temporary variables for n processors and add the numbers in groups of n, we essentially get the same result.

For e.g.,





We can see that this particular code can be split into parts and we can take advantage of parallel computing.

Calculation

$$\boldsymbol{Speedup} = \frac{\mathbf{1}}{(\mathbf{1} - \boldsymbol{f}) + \frac{\boldsymbol{f}}{\boldsymbol{p}}}$$

Parallelizable part in this case is the loop part. We know that the loop is executed 10 times and there are 4 instructions in the loop. Since we are executing this program in a single cycle MIPS processor, number of instructions is equal to number of cycles.

$$\boldsymbol{f} = \frac{\boldsymbol{Number\ cycles\ from\ independent\ instructions}}{\boldsymbol{Total\ number\ of\ executed\ cycles}}$$

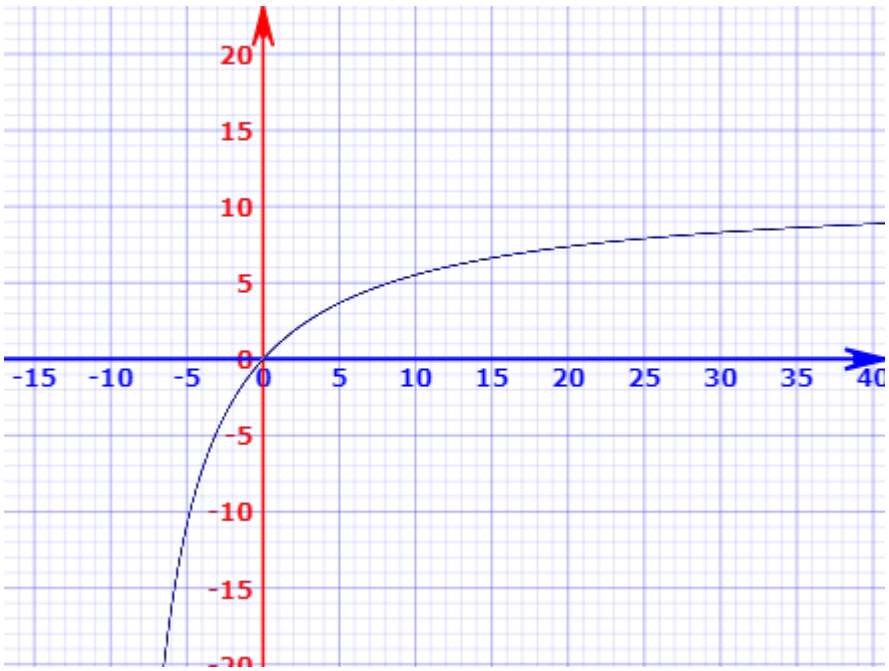
Number of independent cycles = 10 * 4 = 40

Total number of cycles =44

Therefore f = 40 / 44 ~= 0.901

$$\boldsymbol{Speedup = \frac{1}{(0.09) + \frac{0.91}{p}}}$$

Graph (Speedup Vs Number of Processors)



Although there is a performance boost using parallelism, that boost is saturated after a certain point. And for this particular case, using 40 processors would give the best performance, but afterwards, the processors would essentially add no value to the performance