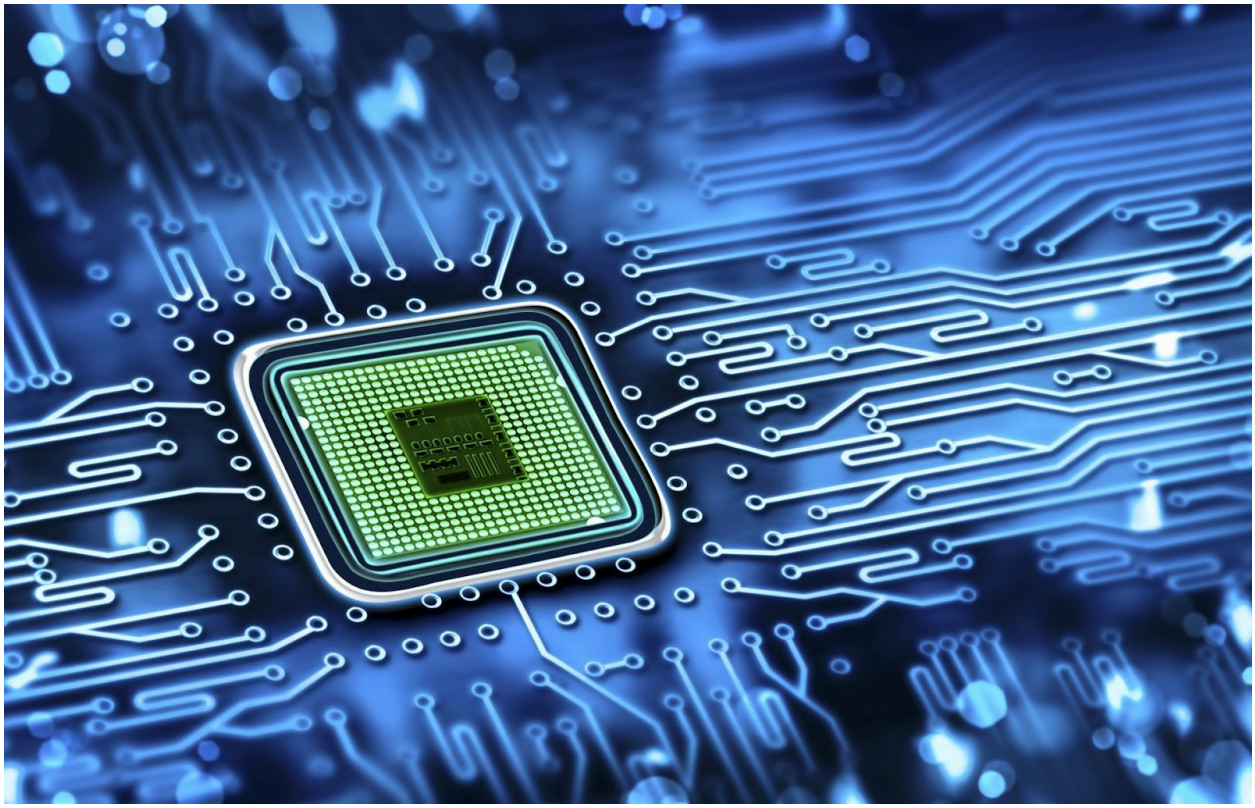
EE1193

Introduction to Hardware Description Language

Pipeline Processor



Introduction

In this project we have designed a Pipeline processor consisting of three stages, i.e, Decoding, executing and writeback stages. In computing, a **processor** or **processing unit** is an electronic circuit which performs operations on some external data source, usually memory or some other data stream. The term is frequently used to refer to the central processor (central processing unit) in a system, but typical computer systems (especially SoCs) combine a number of specialised "processors".

Working

A three-stage pipelined processor has decode, execute and write back stages. The first stage decodes the function that will be executed in the execute stage. The output of the execute stage is written to a particular memory area in sequence in write backstage. The processor accepts 32-bit numbers as inputs.

Stage-I : Decoder

In the processor that we implemented, the decode stage looks at bits [31:24] and generates the signals to control muxes in the execute stage. The remaining bits are passed as in registers to execute stage along with the control signals. We have checked the last bit and sent it as a control signal. We get either a ‘0’ or ‘1’ as output.

Stage-II : Execution

In this stage, we have implemented two functions i.e, addition and Logical And. We have implemented this stage as follows, bits [15:8] and [7:0] of the inputs are treated as two bytes for these functions, which are selected by a control signal from the decode stage. The output and bit [23:16] of the input are passed in register to the next stage.

Stage-III : Write back

In this stage, we are storing the output generated by the execution stage into the location, in a register pointed by the bits [23:16] of input.

Pipelining

In this implementation of the processor, we are using registers to store the values generated by the stages to increase the throughput of each stage. We have used 5 registers to achieve this maximum throughput. The uses and the explanation for the efficiency for using pipelining are given below,

Uses & Efficiency

In this Project of building a processor, we have utilized the power of pipelining. Pipelining is the use of a pipeline. Without a pipeline, a computer processor gets the first instruction from memory, performs the operation it calls for, and then goes to get the next instruction from memory, and so forth. While fetching (getting) the instruction, the arithmetic part of the processor is idle. It must wait until it gets the next instruction. With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can be performed. The staging of instruction fetching is continuous. The result is an increase in the number of instructions that can be performed during a given time period.

How it's different & Efficient?

For example, if each stage has a delay of 'n' seconds.The time we have to wait for giving the next input is '3n' seconds but by using pipeline we can give input after n seconds only because the given input is being stored in a register bank and it transfers the data after each clock cycle. Therefore the throughput is increased. Hence the efficiency of power consumption increases.

Calculations

Let's take input and go through all the steps we have covered. Let the input be "11000011 11000011 00100011 00100011", in the first stage, we look at bit X[31], for getting the selector of the MUX which we are going to use in the next stage. Here X[31] is '1', hence the Selector pin for the execution stage is '1'. In the execution stage, we use the mux for implementing the specified operation. In this case, as the output is '1' we add the bits [15:8] and [7:0] of the inputs else the Logical And function will be performed for each of the given inputs. Here, in this case, 00100011 + 00100011 = 01000110 will be stored in a register according to the algorithm provided by the write back stage. In the Write backstage, we are using the [23:16] bits of input as the index in the array for storing the output value of execution stage. In this case " 11000011", i.e, in decimal representation it is 195. We are using 195th slot of the storage memory reg to store the value "01000110", i,e, storage[195]=01000110.

Advancements:

There is a chance of collisions for storing the input i.e if the [23:16] bits of the two different inputs are same the hash to the same slot. So, in order not to overwrite the previous inputs we should search for a probe which is not already filled in the hash table. For this, we can use the efficient method of double hashing which is explained as follows,

(hash1(key) + i * hash2(key)) % TABLE_SIZE

Here hash1() and hash2() are hash functions and TABLE_SIZE is the size of the hash table. hash1(key) = key % TABLE_SIZE, **hash2(key) = PRIME – (key % PRIME)** where PRIME is a prime smaller than the TABLE_SIZE.

Submitted by:

Akash Tadwai - ES18BTECH11019

Vamshi T - ES18BTECH11021