INSTRUCTION EXECUTION CYCLE

The instruction cycle (also known as the fetch–decode–execute cycle, or simply the fetch-execute cycle) is the cycle that the central processing unit (CPU) follows from boot-up until the computer has shut down in order to process instructions. Once an instruction has been loaded into the instruction register (IR), and the control unit (CU) has examined and decoded the fetched instruction and determined the required course of action to take, the execution cycle can commence.

This cycle consists of three stages: fetching the instruction, decoding the instruction, and executing the instruction - these three steps are known as the machine cycle. A processor spends all its time in this cycle, endlessly retrieving the next instruction, decoding it, and running it.

INSTRUCTION EXECUTION CYCLE OF GOOGLE TPU

TPU Microarchitecture: TPU uses a 4-stage pipeline for these CISC instructions, where each instruction executes in a separate stage. The plan was to hide the execution of the other instructions by overlapping their execution with the MatrixMultiply instruction. Toward that end, the Read_Weights instruction follows the decoupled access/execute philosophy, in that it can complete after sending its address but before the weight is fetched from Weight Memory. The matrix unit will stall if the input activation or weight data is not ready. We do not have clean pipeline overlap diagrams, because our CISC instructions can occupy a station for thousands of clock cycles, unlike the traditional RISC pipeline with one clock cycle per stage. Interesting cases occur when the activations for one network layer must complete before the matrix multiplications of the next layer can begin; we see a “delay slot,” where the matrix unit waits for explicit synchronization before safely reading from the Unified Buffer.

In each state:

At first, TPU loads the parameters from memory into the matrix of multipliers and adders. Then, the TPU loads data from memory. As each multiplication is executed, the result will be passed to next multipliers while taking summation at the same time.

HOW TPU WORKS?

When Google designed the TPU, we built a domain-specific architecture. That means, instead of designing a general purpose processor, we designed it as a matrix processor specialized for neural network work loads. TPUs can't run word processors, control rocket engines, or execute bank transactions, but they can handle the massive multiplications and additions for neural networks, at blazingly fast speeds while consuming much less power and inside a smaller physical footprint.

The key enabler is a major reduction of the von Neumann bottleneck. Because the primary task for this processor is matrix processing, hardware designer of the TPU knew every calculation step to perform that operation. So they were able to place thousands of multipliers and adders and connect them to each other directly to form a large physical matrix of those operators. This is called systolic array architecture. In case of Cloud TPU v2, there are two systolic arrays of 128 x 128, aggregating 32,768 ALUs for 16 bit floating point values in a single processor.

Let's see how a systolic array executes the neural network calculations. At first, TPU loads the parameters from memory into the matrix of multipliers and adders.

Then, the TPU loads data from memory. As each multiplication is executed, the result will be passed to next multipliers while taking summation at the same time. So the output will be the summation of all multiplication result between data and parameters. During the whole process of massive calculations and data passing, no memory access is required at all.

This is why the TPU can achieve a high computational throughput on neural network calculations with much less power consumption and smaller footprint.

For the IAS computer, there are 21 instructions which fall into 5 categories

They are:

1. Data Transfer: moves data between memory and ALU registers

or between two ALU registers.

e.g.

LOAD MQ : transfer contents of MQ to AC

(00001010)

LOAD MQ, M(X) : transfer contents of mem. loc. X to MQ

(00001001)

STOR M(X) : transfer contents of AC to mem. loc. X

(00100001)

LOAD M(X) : transfer M(X) to AC

(00000001)

LOAD –M(X) : transfer -M(X) to AC

(00000010)

LOAD |M(X)| : transfer absolute value of M(X) to AC

(00000011)

LOAD - |M(X)| : transfer -|M(X)| to AC

(00000100)

2. Unconditional branch: changes the execution sequence.

e.g.

JUMP M(X, 0:19) :Take next inst from left half of M(x)

JUMP M(X, 20:39) :Take next inst from right half of M(x)

3. Conditional branch: branching depending on a condition.

JUMP +M(X, 0:19) :if number in the AC is non-ve, take next inst.
from left half of M(X)

JUMP +M(X, 20:39) :if number in the AC is non-ve, take next inst.
from right half of M(X)

4. Arithmetic: operations performed by ALU.

e.g.

ADD M(X) : add M(X) to AC; put the result in AC

ADD |M(X)| : add |M(X)| to AC; put the result in AC

SUB M(X) : subtract M(X) from AC; put the result in AC

5. Address Modify: change address according to some calculations in ALU.

e.g. STOR M(X, 8:19) :Replace left address at M(X) by 12 right most bits in AC.

STOR M(X, 28:39) :Replace right address at M(X) by 12 right most bits in AC.

For Addition of 2 arrays using IAS Machine Instructions we can use the following method:

Position	Instruction	Comments
0	**1**	**Constant**
1	**999**	**Constant**(Take as "count N")
2	**1000**	**Constant**
3L	**LOAD M(2000)**	**Transfer A(i) to AC**
3R	**ADD M(3000)**	**Compute A(i)+B(i)**
4L	**STOR M(4000)**	**Transfer sum to C(i)**
4R	**LOAD M(1)**	**Load count N**
5L	**SUB M(0)**	**Decrease N by 1**
5R	**JUMP+ M(6,20:39)**	**See N and move to 6R if positive or zero**
6L	**JUMP M(6,0:19)**	**Pause**
6R	**STOR M(1)**	**Update N**
7L	**ADD M(0)**	**Increase AC by 1**
7R	**ADD M(2)**	
8L	**STOR M(3,8:19)**	**Modify address in 3L**
8R	**ADD M(2)**	
9L	**STOR M(3,28:39)**	**Modify address in 3R**
9R	**ADD M(2)**	
10L	**STOR M(4,8:19)**	**Modify address in 4L**
10R	**JUMP M(3,0:19)**	**Move to 3L**

For example, let us take the example of Z=A-(B/C)=(AC-B)/C

For this, the IAS instructions passed will be:

Let,

A=M(400)

B=M(401)

C=M(402)

Z=M(403)

So the code will be:

LOAD	M(400)	Taking the value of A
MUL	M(402)	Multiplying A and C
SUB	M(401)	Subtracting B from AC
DIV	M(402)	Dividing AC-B by C
STO	M(404)	Storing the value of remainder in other memory location
LOAD	MQ	Now the value is in Accumulator
STO	M(403)	Now we store the final value at memory location of Z
HLT		