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

For this, the IAS instructions passed will be:

Let,

A=M(300)

B=M(301)

C=M(302)

Z=M(303)

So the code will be:

```
LOAD    M(300)        Taking the value of A
SUB     M(301)        Subtracting B from A
DIV     M(302)        Dividing A-B by C
STOR    M(304)        Storing the value of remainder in other memory location
LOAD    MQ            Now the value is in Accumulator
STOR    M(303)        Now we store the final value at memory location of Z
```

<u>Samvit Swaminathan (19BCE0629)</u>

- The issue which is not in the scope of a programmer is **<u>Advancement in industry and Test Environment Duplication.</u>**

- I can confirm that the difference in the number of PC processors and Embedded processors by looking at their presence in my house. It has:
  1. **21** Embedded Processor
  2. **5** PC processors

<u>Bottlenecks</u>

1) A "Bottleneck" is a point of congestion in a production system that occurs when workloads arrive too quickly for the production process to handle. The inefficiencies brought about by the bottleneck often create delays and higher production costs.
2) When the process we have chosen is wrong and takes long time to process. For example, if we are compiling a code with infinite loop, then the code will not give desired output and will run continuously until we forcefully stop it.
3) Installation of compiler might not be done properly. Or if your code contains a syntax error that forces the code to go in an undesired manner.
4) If the prototype is OS-specific, then it will show error. Like we cannot install Android application in IOS device and vice versa or a Windows software in Mac.
5) The software might not be made for that processor.
6) The processor demands might not be met, like the properties might not be compatible.
7) There can be a hardware error, like error in mouse or keyboard which can cause error in code or the screen might not

be clearly visible due to lack of resolution etc.
8) This performance problem can be reduced by introducing a cache memory (special type of fast memory) in between the CPU and the main memory. This is because the speed of the cache memory is almost same as that of the CPU.

Check Yourself

Approximate Relative Access Time
Semiconductor memory has much faster access times than other types of data storage; a byte of data can be written to or can be read from semiconductor memory within a few nanoseconds, while access time for rotating storage such as hard disks is in the range of 5-9 milliseconds.

Volatility
Disk Memory and Flash Storage are non-volatile.
DRAM is volatile.

Cost
DRAM is more inexpensive. Flash memory is more expensive than hard drives as it has higher cost per bit.

MIPS:

MIPS is the abbreviation for "Million Instructions Per Second”. It is a method of measuring the raw speed of a computer's processor.
Million instructions per second (MIPS) is an older, obsolete measure of a computer's speed and power, MIPS measures roughly the number of machine instructions that a computer can execute in one second. Since the MIPS measurement doesn't consider other factors such as the computer's input and output speed or processor properties, it isn't a fair way to measure the performance of a computer. For example, a computer rated at 80 MIPS may be able to computer certain functions faster than another computer rated at 100 MIPS.
MIPS is a RISC-type, Load/Store instruction set. The early implementations like MIPS 1 and MIPS 2 were 32 bits while MIPS 3, 4 and 5 are 64 bits.

MFLOPS:

MFLOP is the abbreviation of Mega Floating-point operations per second. MFLOPs are a common measure of computer speed used to perform floating-point calculations. Another common measure of computer speed and power is MIPS which indicates integer performance. A floating-point operation is an addition, subtraction, multiplication, or division operation applied to a number in a single or double precision floating-point representation. Such data items are heavily used in scientific calculations and are specified in programming languages using key words like float, real, double, or double precision.

NUMERICALS ON CPU EXECUTION TIME

$$\text{CPU time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

CPU time = CPU clock cycles x clock cycle time

Q1. CPU clock rate is 10 Mhz. Program takes 2 million cycles to execute. What is the CPU time?

=> 2000000 * (1 / 10000000) = 0.2 seconds

Q2. CPU clock rate is 50 Mhz. Program takes 450 million cycles to execute. What is the CPU time?

=> 450000000 * (1 / 50000000) = 09 seconds

Q3. Let assume that a benchmark has 100 instructions: 40 instructions are loads/stores (each take 2 cycles)

70 instructions are added (each takes 3 cycle)
10 instructions are square root (each takes 50 cycles)
What is the Cycles per Instruction (CPI) for this benchmark?

=> CPI = ((0.4 * 2) + (0.7 * 3) + (0.1 * 50)) = 7.9

INTERNAL ARCHITECTURE OF GOOGLE TENSOR PROCESSING UNIT

Tensor Processing Units (TPUs) are Google's custom-developed application-specific integrated circuits (ASICs) used to accelerate machine learning workloads. These TPUs are designed from the ground up with the benefit of Google's deep experience and leadership in machine learning.

In a Google data center, TPU devices are available in the following configurations for both TPU v2 and TPU v3:

- **TPU v2:**
  - 8 Gb of HBM for each TPU core
  - One MXU for each TPU core
  - Up to 512 total TPU cores and 4 Tb of total memory in a TPU Pod
- **TPU v3:**
  - 16 Gb of HBM for each TPU core
  - Two MXUs for each TPU core
  - Up to 2048 total TPU cores and 32 Tb of total memory in a TPU Pod

Single-device TPUs

A single-device TPU configuration in a Google data center is one TPU device with no dedicated high-speed network connections to other TPU devices. Your TPU node connects only to this single device.

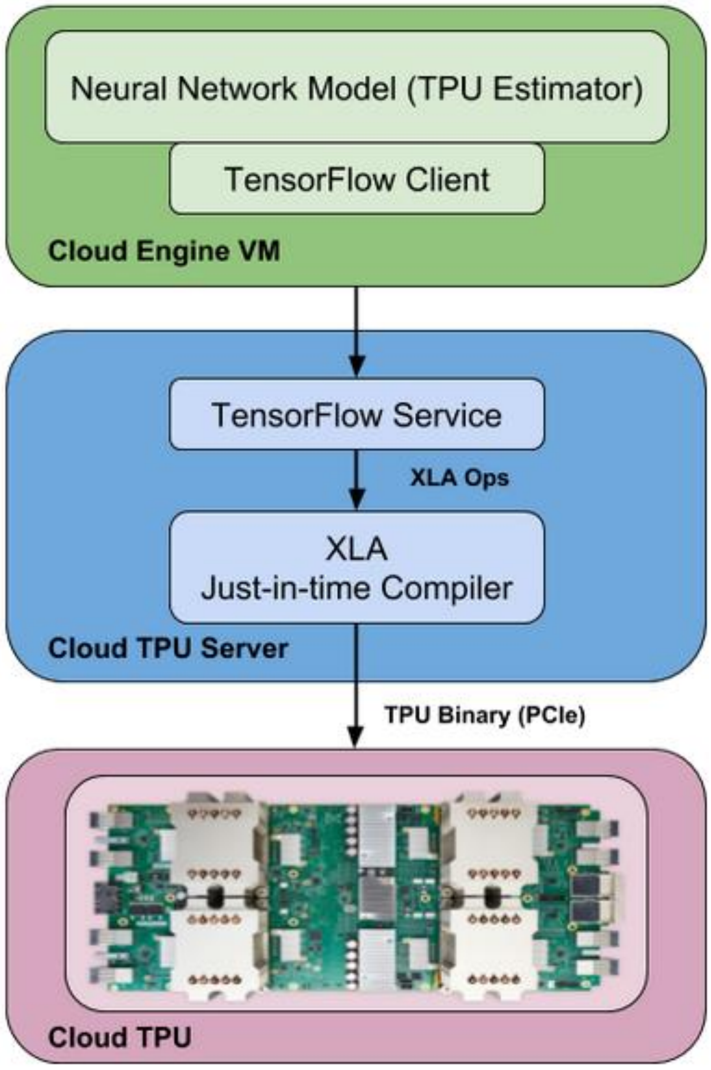
Single device TPUs, which are individual TPU devices that are not connected to each other over a dedicated high-speed network. You cannot combine multiple single device TPU types to collaborate on a single workload. For single-device TPUs the chips are interconnected on the device so the communication between the TPU chips is not very difficult and there is no requirement of a host (example: CPU).

TPU Pods

A TPU pod configuration in a Google data center has multiple TPU devices connected to each other over a dedicated high-speed network connection. The hosts in your TPU node distribute your machine learning workloads across all the TPU devices.

A TPU pod configuration in a Google data center has multiple TPU devices connected to each other over a dedicated high-speed network connection. In a TPU Pod, the TPU chips are interconnected on the device so that communication between chips does not require host CPU or host networking resources. Additionally, each of the TPU devices in a TPU Pod are connected to each other over dedicated high-speed networks that also do not require host CPU or host networking resources.

Software architecture
(Google Colab's Free TPU)



TPU Estimator

TPU Estimators are a set of high-level APIs that build upon estimators which simplify building models for Cloud TPU and which extract maximum TPU performance. When writing a neural network model that uses Cloud TPU, you should use the TPU Estimator APIs.

TensorFlow Client

A TensorFlow server runs on a Cloud TPU server. TPU Estimators translate your programs into TensorFlow operations, which are then converted into a computational graph by a TensorFlow client. This graph is then sent to the server by the TensorFlow client.

TensorFlow Server

TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments. TensorFlow Serving makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs.

A TensorFlow server runs on a Cloud TPU server. The server then receives a computational graph from the TensorFlow client after which the below listed actions take place:

- Takes inputs from Cloud Storage.
- It differentiates the portions of the graph into 2 parts, the ones that should run on Cloud TPU and the ones that must run using the CPU.
- Generate XLA operations corresponding to the sub-graph that is to run on Cloud TPU.
- Uses XLA compiler.

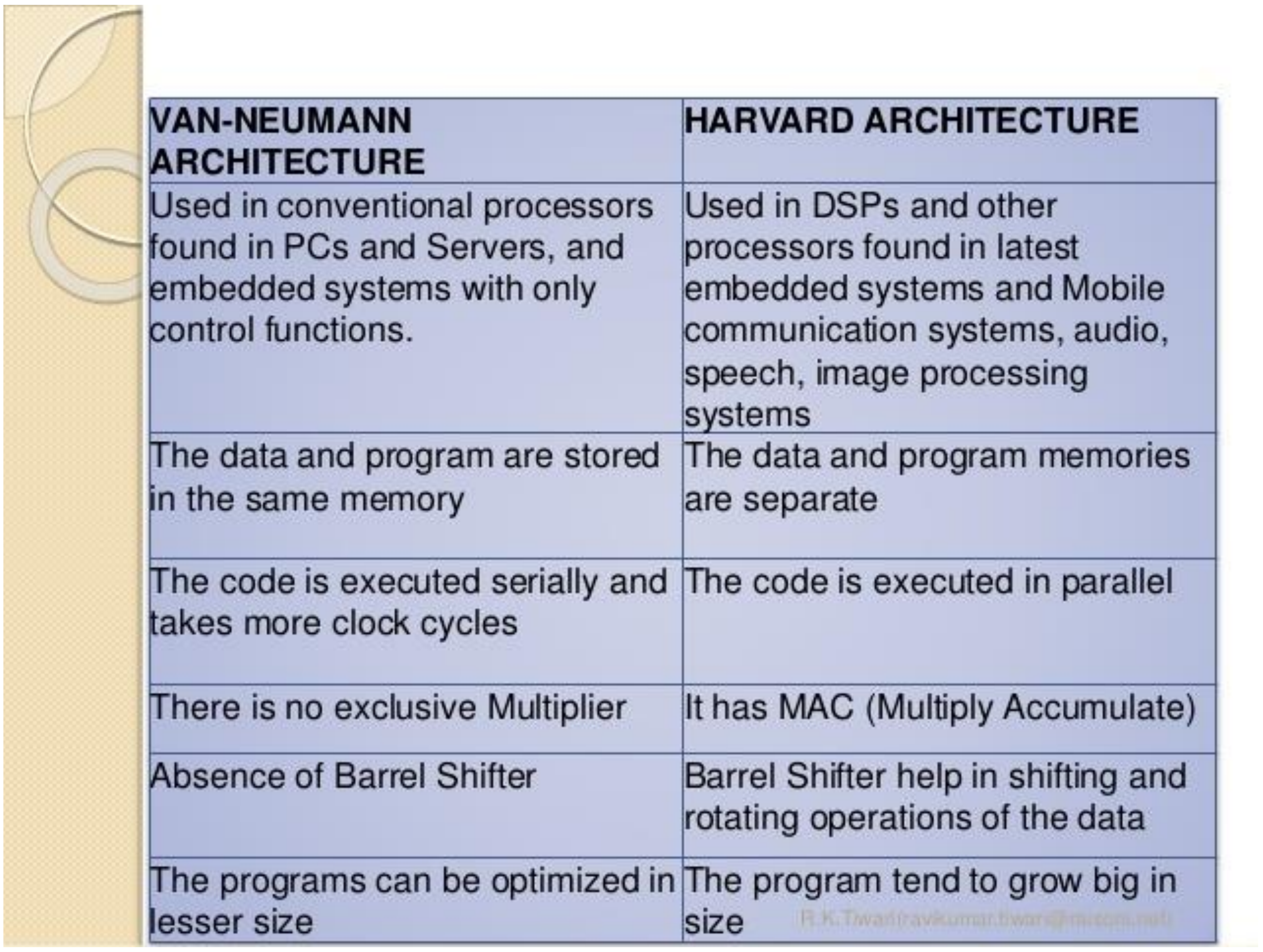
XLA Compiler

XLA is the secret compiler sauce that helps TensorFlow optimize compositions of primitive ops automatically. TensorFlow, augmented with XLA, retains flexibility without sacrificing runtime performance, by analyzing the graph at runtime, fusing ops together and producing efficient machine code for the fused subgraphs.

XLA is a just-in-time compiler that takes High Level Optimizer (HLO) operations as input that are produced by the TensorFlow server. XLA generates binary code which then run on Cloud TPU, that includes orchestration of data from on-chip memory to hardware execution units and inter-chip communication. The generated binary is loaded onto Cloud TPU using PCIe connectivity between the Cloud TPU server and the Cloud TPU and is then launched for execution.

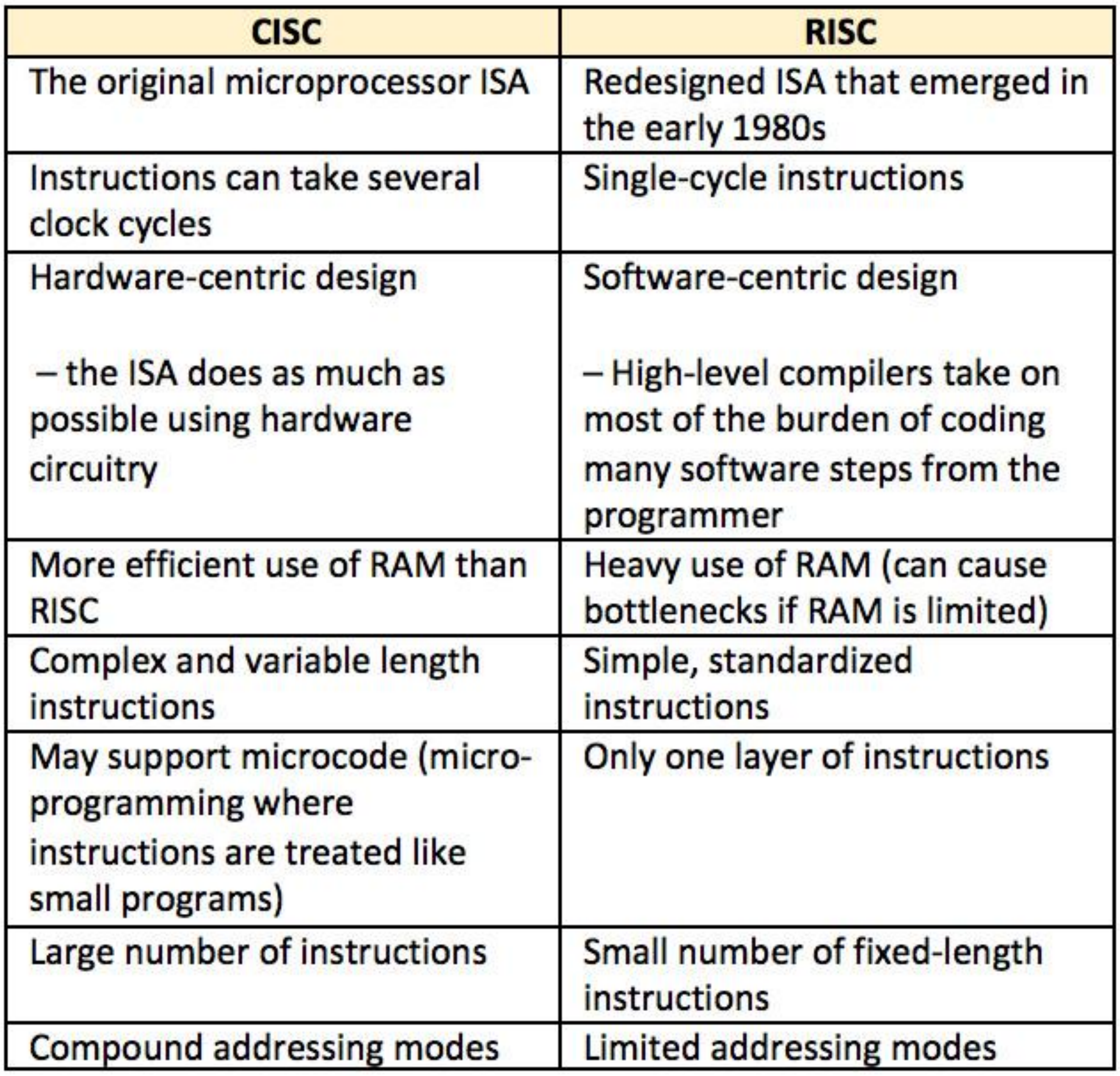
Von Neumann vs Harvard Architecture

Any instruction set can be used with any type of architecture. There is no specific instruction set for an architecture.



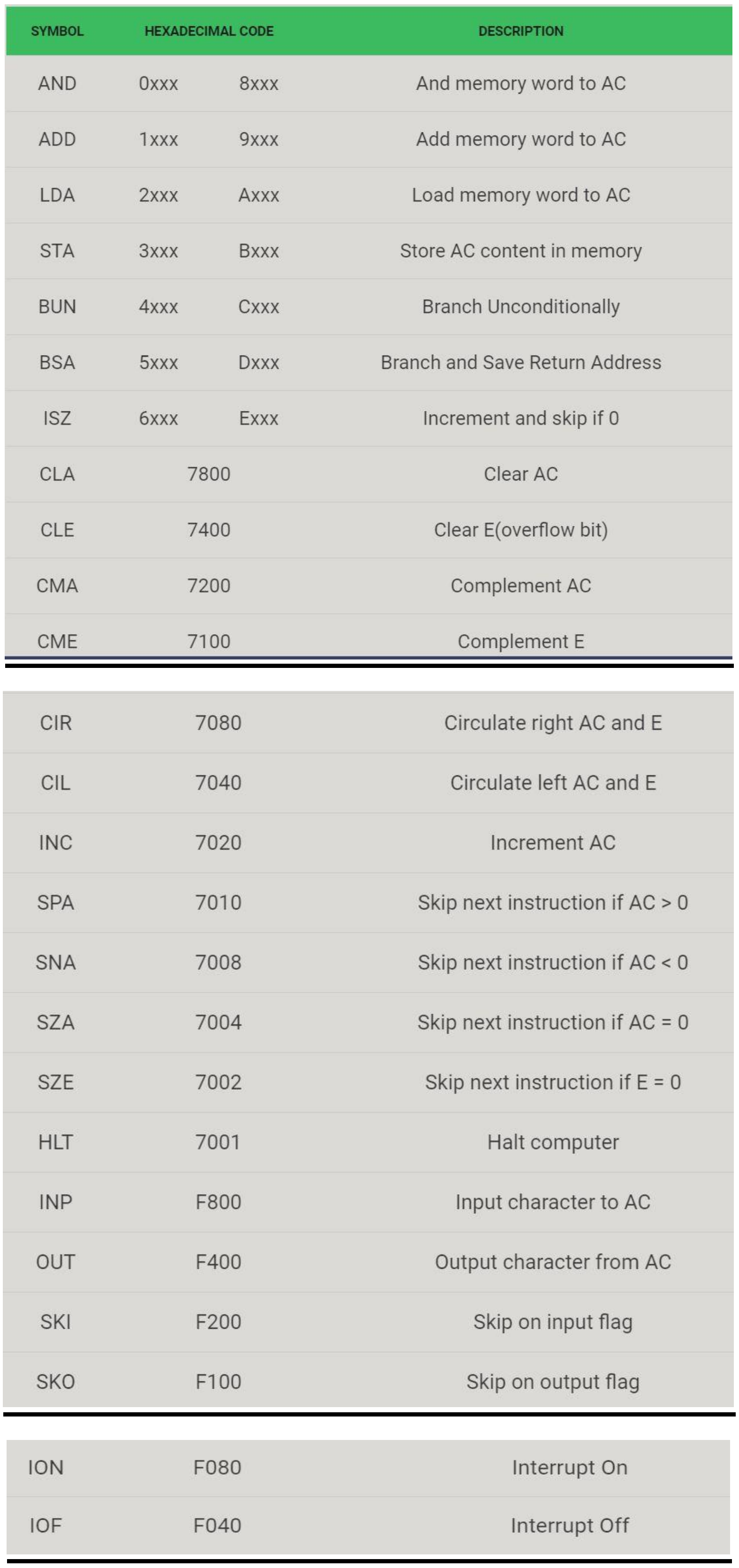
VAN-NEUMANN ARCHITECTURE	HARVARD ARCHITECTURE
Used in conventional processors found in PCs and Servers, and embedded systems with only control functions.	Used in DSPs and other processors found in latest embedded systems and Mobile communication systems, audio, speech, image processing systems
The data and program are stored in the same memory	The data and program memories are separate
The code is executed serially and takes more clock cycles	The code is executed in parallel
There is no exclusive Multiplier	It has MAC (Multiply Accumulate)
Absence of Barrel Shifter	Barrel Shifter help in shifting and rotating operations of the data
The programs can be optimized in lesser size	The program tend to grow big in size

CISC vs RISC (Instruction Sets)



CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

Samvit Swaminathan (19BCE0629)

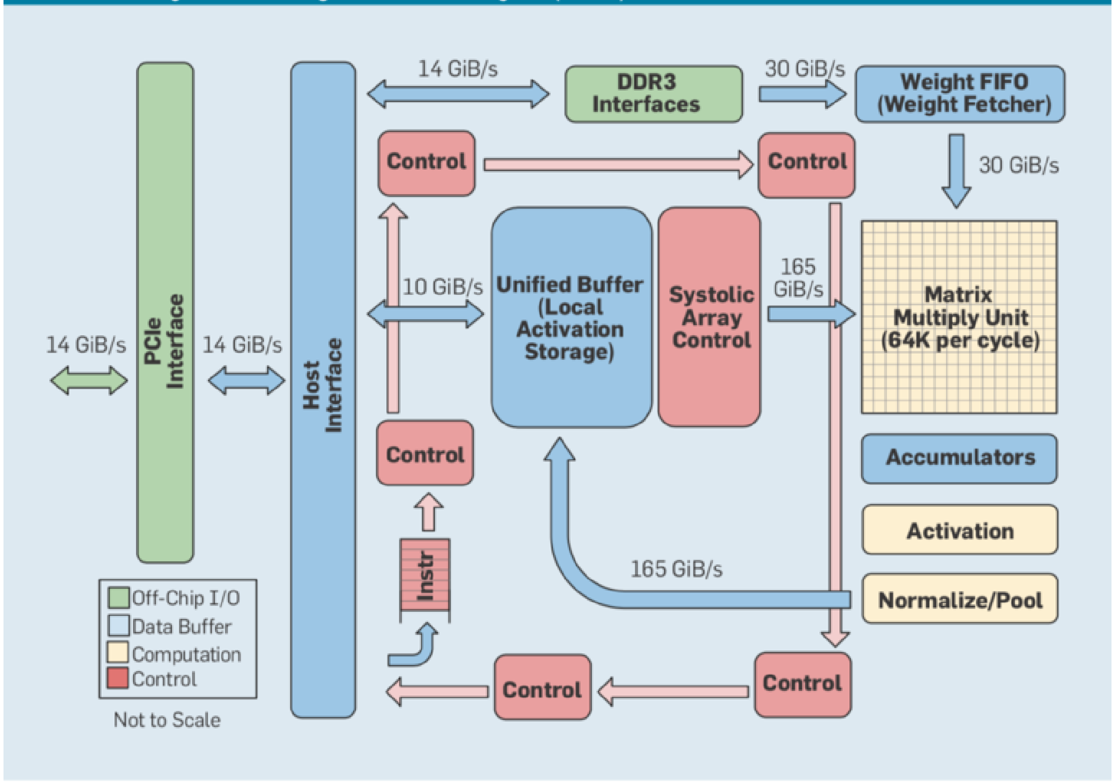


SYMBOL	HEXADECIMAL CODE		DESCRIPTION
AND	0xxx	8xxx	And memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store AC content in memory
BUN	4xxx	Cxxx	Branch Unconditionally
BSA	5xxx	Dxxx	Branch and Save Return Address
ISZ	6xxx	Exxx	Increment and skip if 0
CLA	7800		Clear AC
CLE	7400		Clear E(overflow bit)
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC > 0
SNA	7008		Skip next instruction if AC < 0
SZA	7004		Skip next instruction if AC = 0
SZE	7002		Skip next instruction if E = 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt On
IOF	F040		Interrupt Off

Instructions Set Architecture (ISA) of Google TPU

Google TPU follows the Complex Instruction Set Computers method **(CISC)** as the base of the TPU instruction set. A CISC design focuses on implementing high-level instructions that run more complex tasks such as calculating using the operations of multiply-and-add many times with each instruction. Let us look at the block diagram of the TPU.

ure 8. Functional organization of Google Tensor Processing Unit (TPU v1).



The TPU includes the following computational resources (task specific chips):

- Matrix Multiplier Unit (MXU): 65, 536 8-bit multiply-and-add units for matrix operations.
- Unified Buffer (UB): 24MB of SRAM that work as registers.
- Activation Unit (AU): Hardwired activation functions.

To control how the MXU, UB and AU proceed with operations, we defined a dozen high-level instructions specifically designed for neural network inference like, “Read_Host_Memory”, “Read_Weights”, “Write_Host_Memory” etc. This instruction set focuses on the major mathematical operations required for neural network inference.

Advantages of TPU:

The following are some notable advantages of TPUs:

1. Accelerates the performance of linear algebra computation, which is used heavily in machine learning applications.
2. Minimizes the time-to-accuracy when you train large, complex neural network models.
3. Models that previously took weeks to train on other hardware platforms can converge in hours on TPUs.

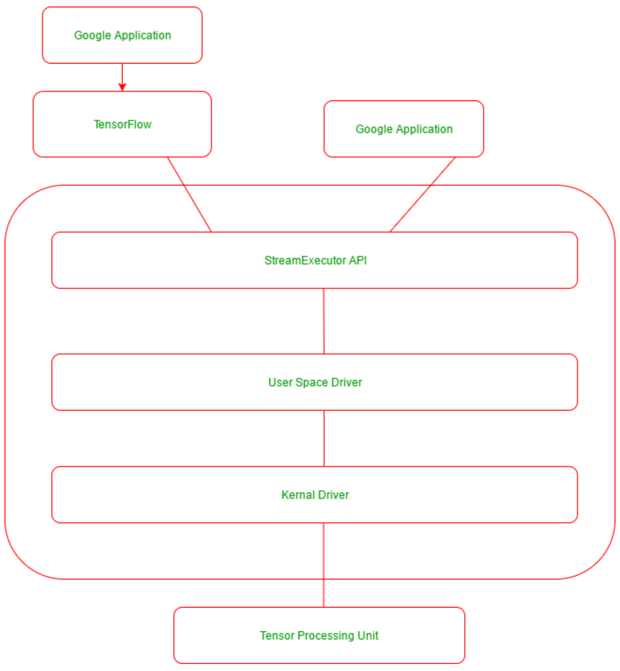
When to use a TPU:

The following are the cases where TPUs are best suited in machine learning:

- Models dominated by matrix computations.
- Models with no custom TensorFlow operations inside the main training loop.
- Models that train for weeks or months
- Larger and exceptionally large models with very large effective batch sizes.

In short, the TPU design contains the crux of neural network calculation and can be programmed for a wide variety of neural network models. To program it, a compiler and a software stack was created that translates API information from TensorFlow graphs into TPU instructions.

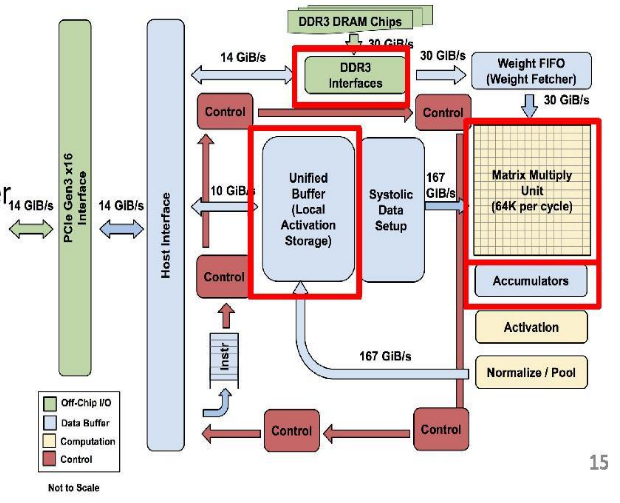
The block given below shows the above-mentioned process:



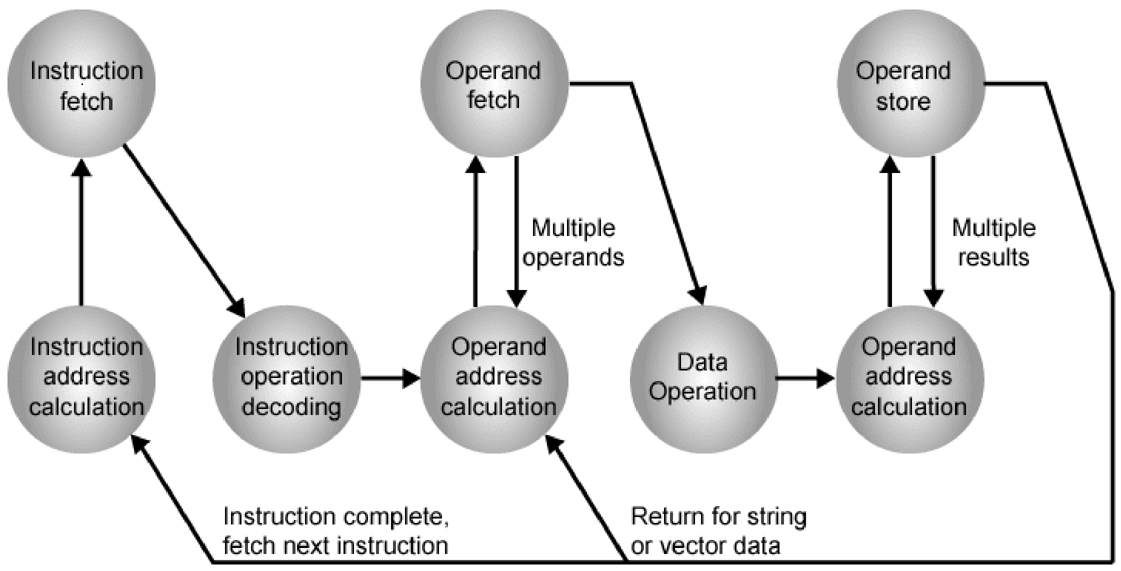
Instruction Cycle State Diagram

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
    - 65,536 * 2 * 700M
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory
- 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- Two 2133MHz DDR3 DRAM channels
- 8 GiB of off-chip weight DRAM memory

TPU: High-level Chip Architecture



- **TPU Schematic Architecture**



Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

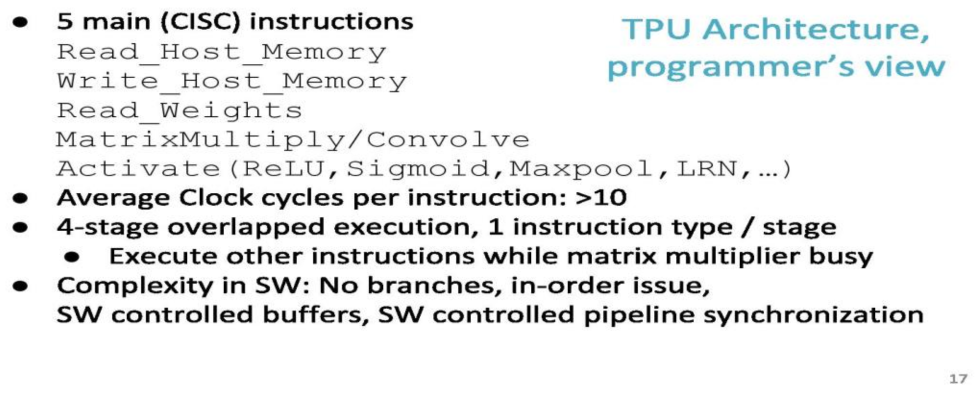
Figure 10.1 Instruction Cycle State Diagram

The TPU includes the following computational resources (task specific chips):

- Matrix Multiplier Unit (MXU): 65,536 8-bit multiply-and-add units for matrix operations.
- Unified Buffer (UB): 24MB of SRAM that work as registers.
- Activation Unit (AU): Hardwired activation functions.

Some high Level Instructions

To control how the MXU, UB and AU proceed with operations, we defined a dozen high-level instructions specifically designed for neural network inference. Few of these operations are highlighted in the image below.



TPU Architecture, programmer's view

- **5 main (CISC) instructions**
  ```
  Read_Host_Memory
  Write_Host_Memory
  Read_Weights
  MatrixMultiply/Convolve
  Activate(ReLU,Sigmoid,Maxpool,LRN,...)
  ```
- **Average Clock cycles per instruction: >10**
- **4-stage overlapped execution, 1 instruction type / stage**
  - **Execute other instructions while matrix multiplier busy**
- **Complexity in SW: No branches, in-order issue,**
  SW controlled buffers, SW controlled pipeline synchronization

17