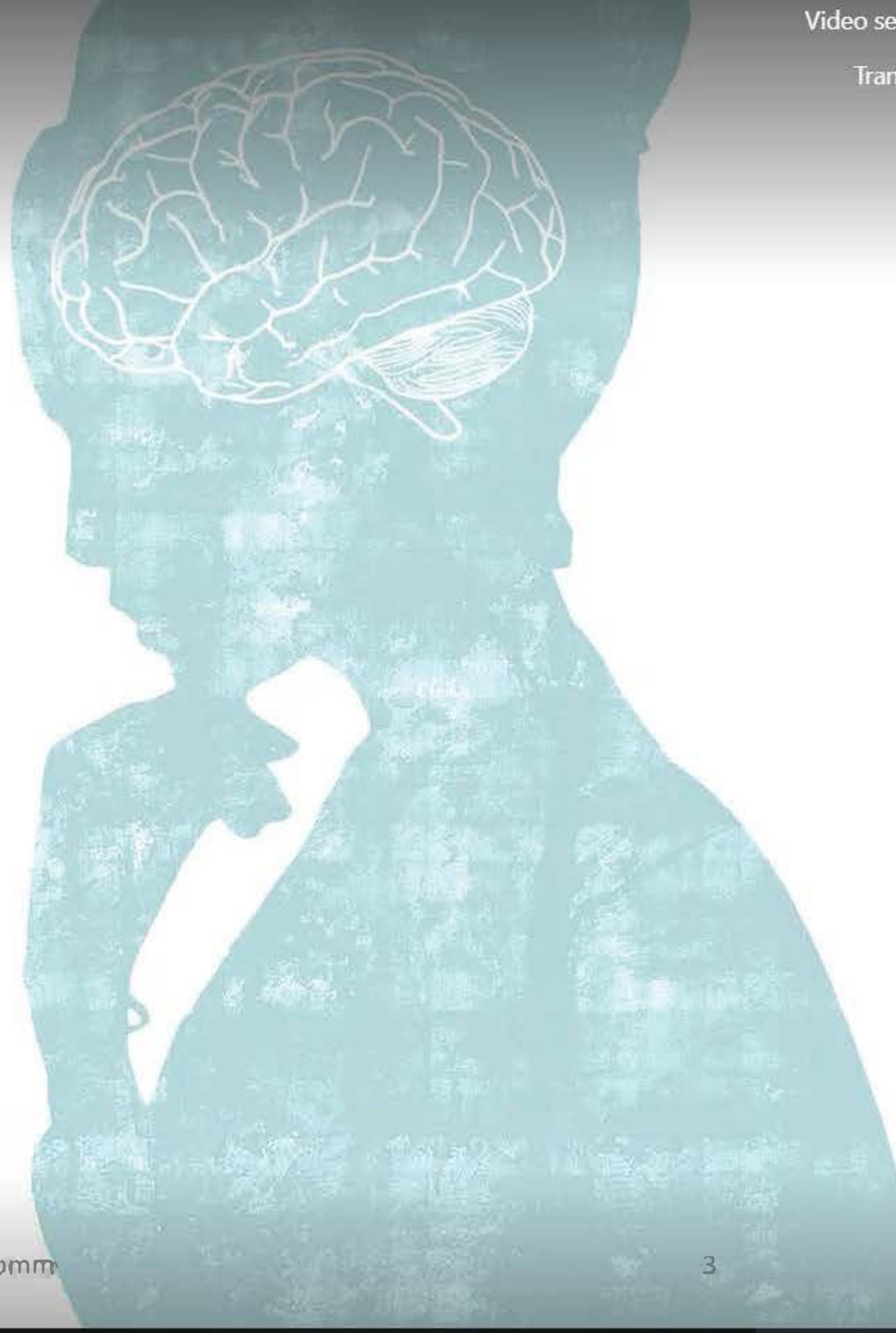
Outline

- Introduction
- SoC lifecycle
- CPU
- DSP
- HW Accelerators
- Benchmarking & KPIs
- Secure coding



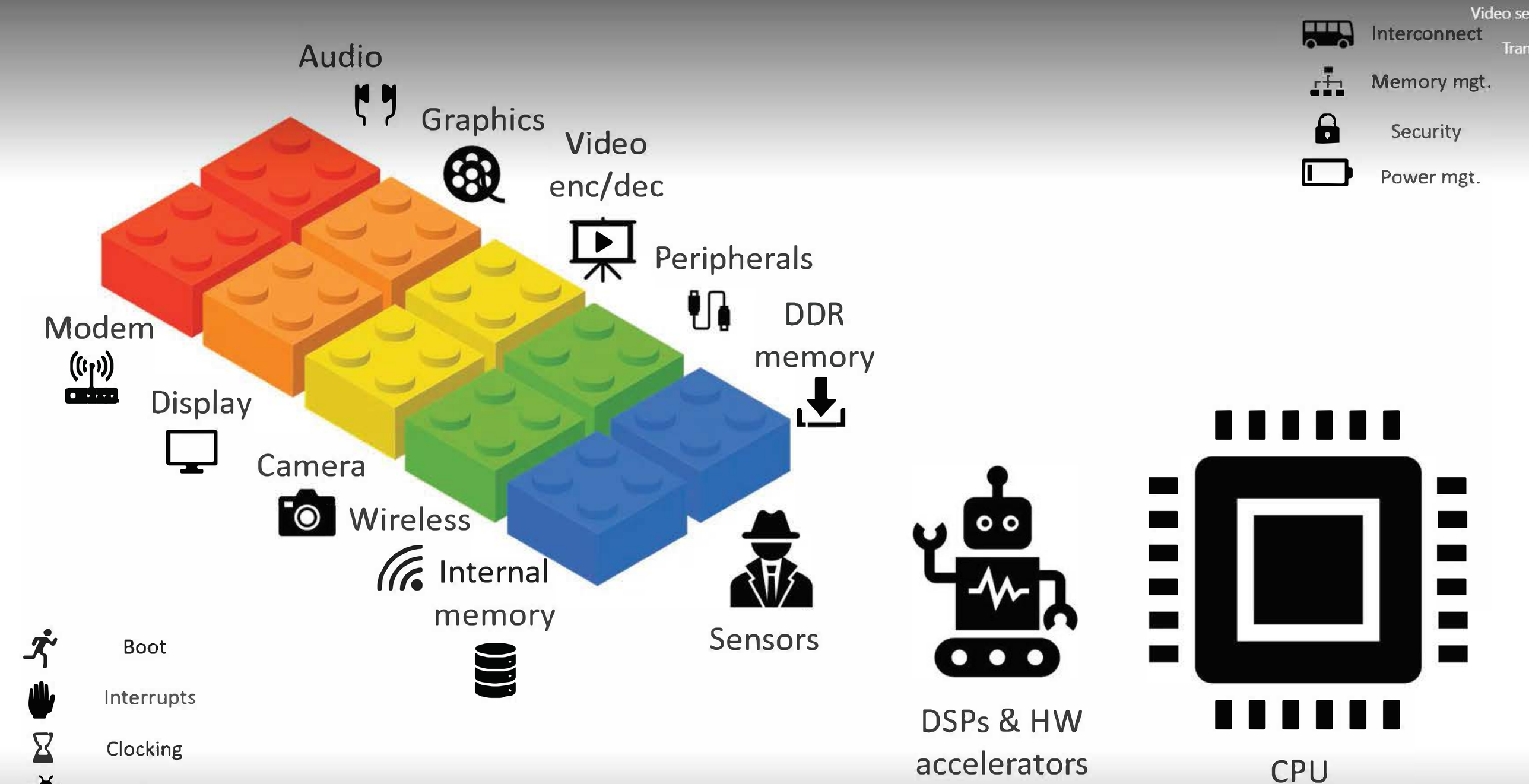
What is Architecture?

- Set of rules and methods that describe the functionality, organization, and implementation of computer systems
- Structure of various internal components of that system and how they interact with one another
- Involves instruction set architecture design, microarchitecture design, logic design, and implementation
  - ISA: defines the machine code that is processed, word size, memory address modes, processor registers, and data type
  - Microarchitecture: how a processor will implement the ISA
  - Systems design: includes all of the other hardware components within a computing system, such as DMA, virtualization, and multiprocessing

What is a System?

- A computer system includes the hardware, operating system and peripheral equipment needed and used for full operation
- Includes the individual processors, memory, caches, physical data streaming technologies (buses, NoCs) and specialized hardware

Let’s build a phone!


Audio
Graphics
Video enc/dec
Peripherals
DDR memory
Modem
Display
Camera
Wireless
Internal memory
Sensors
Interconnect
Memory mgt.
Security
Power mgt.
Boot
Interrupts
Clocking
Debug
DSPs & HW accelerators
CPU

Goals of Systems Arch

- Predictability of the platform
  - Design based on SoC arch platform
  - Meet the design intent
- Scalable (across tiers)
  - Driven by performance KPIs, overlapping features, Perf-power-area (PPA) trade-off
- IP / Design reuse
  - HW/SW aspects
  - Minimize time taken for design implementation, validation, productization
- Application specific

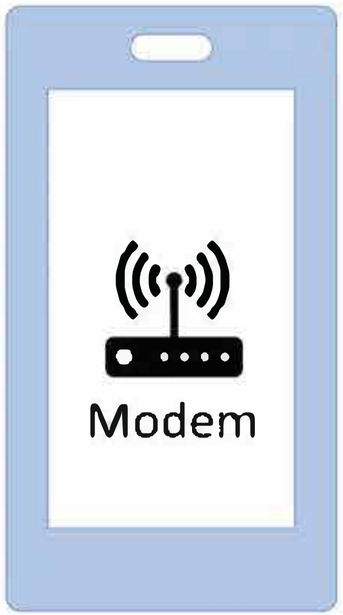
Architecture vs Sytems

SoC Architecture: Not product-specific	SoC HW systems: Product specific
Well-defined design, dependencies, requirements from various internal components	Fixed system specifications driven by trade-offs between requirements and PPA
Independent of product specific details	Independent of implementation details
Examples	
HW IO coherency, QoS, Power management	Size of the cache, product specific extensions to the base platform, power domains

Product Scaling

Compute

bigger form factor, high performance, low power



Mobile

AR / VR / XR

Low power, low latency, high performance, AI

Gaming

High performance GPU, Low power, AI enabled

Automotive

Safety requirements, image processing, autonomous driving

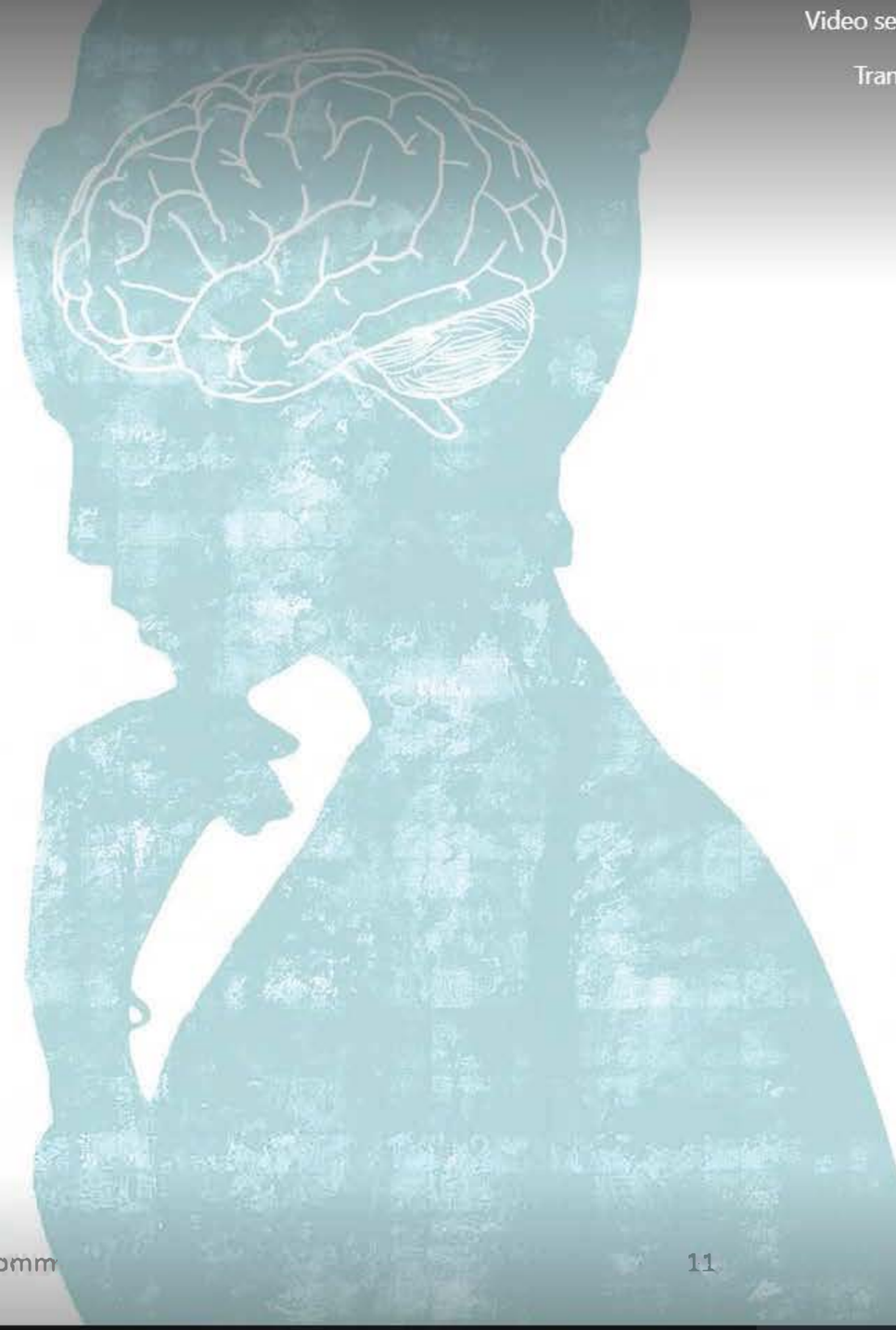
Server

High performance, low power, scaled up compute

Internet of Things

Low power, cost sensitive, specific form factors & requirements

SoC lifecycle

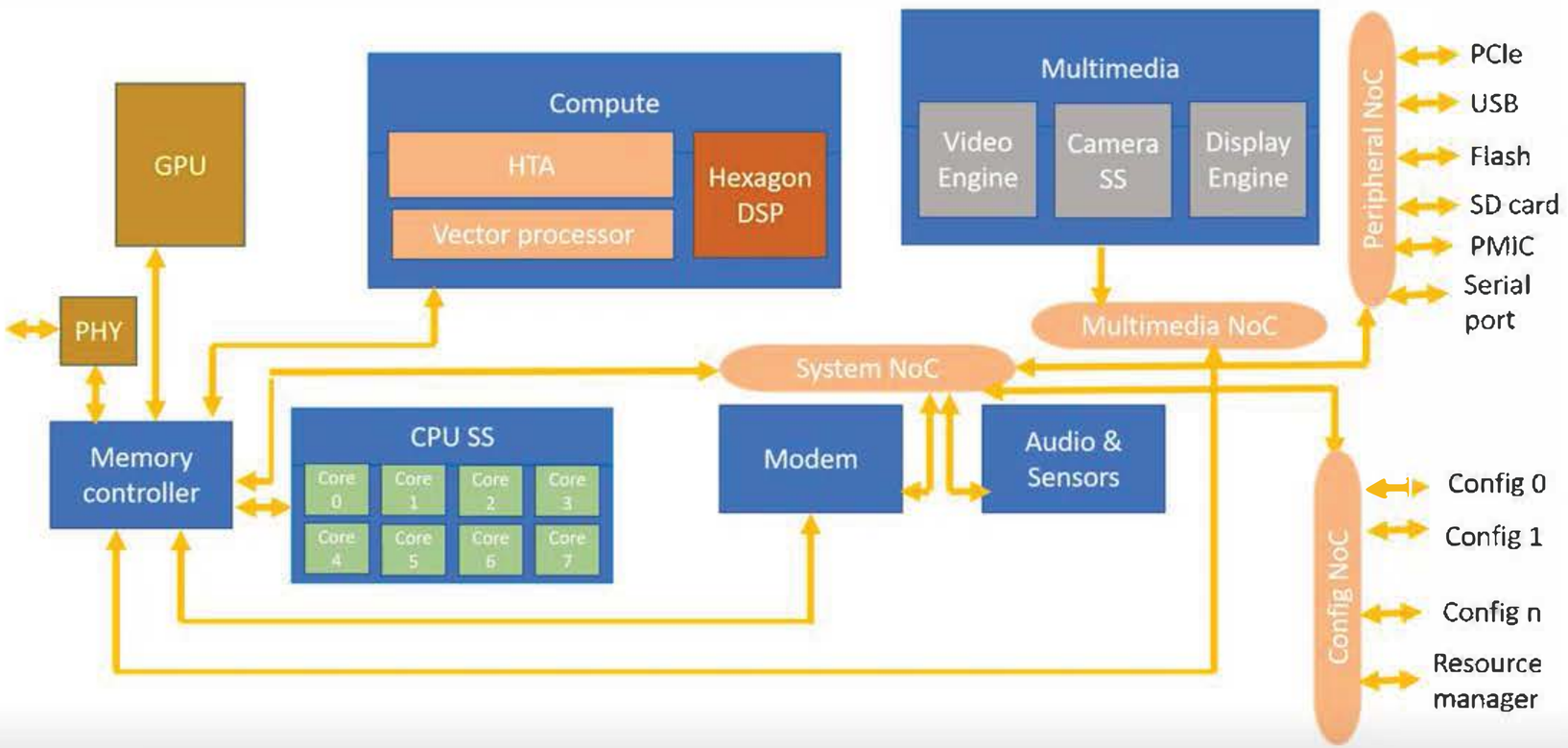


What is an SoC?

- A system on chip
- Integration circuit comprising most components of a system: CPU, Multimedia cores, Digital Signal Processors, Peripherals, Memory controller, GPU, HW accelerator

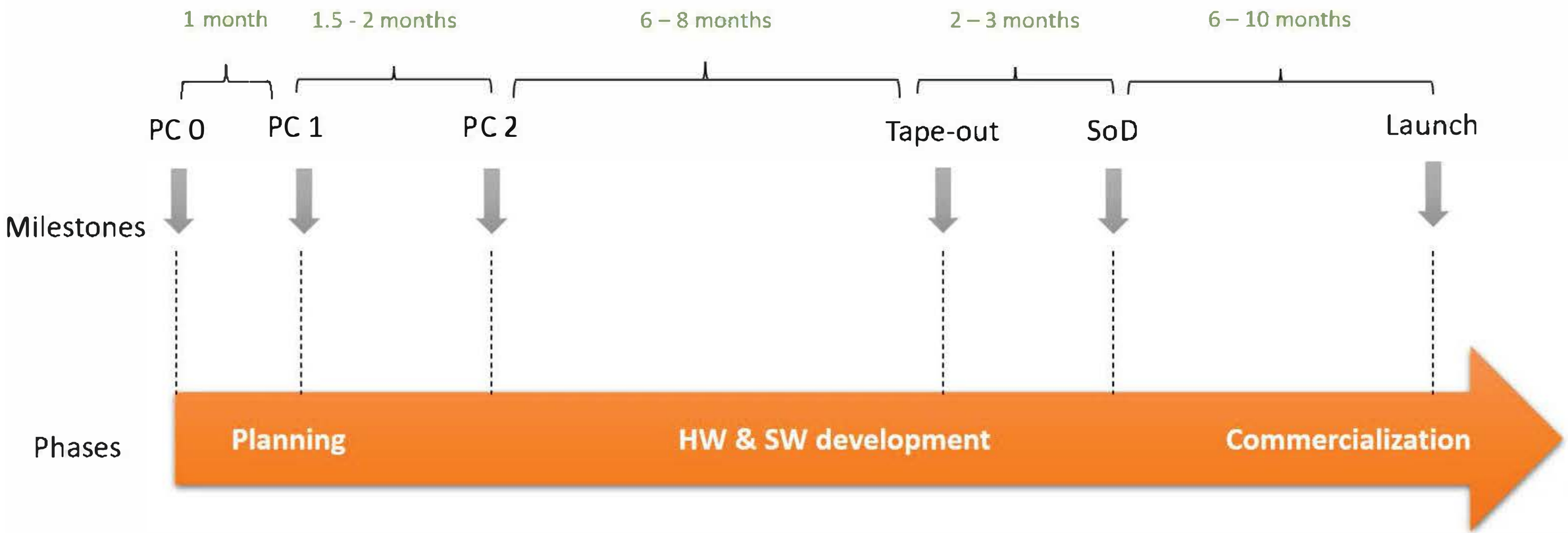
A **network-on-chip** is a network-based communications subsystem on an integrated circuit, between modules in a system on a chip

NoC technology applies computer networking to on-chip communication and better than conventional bus arch



Chipset lifecycle

PC – Product council phase
SoD – Silicon on Dock

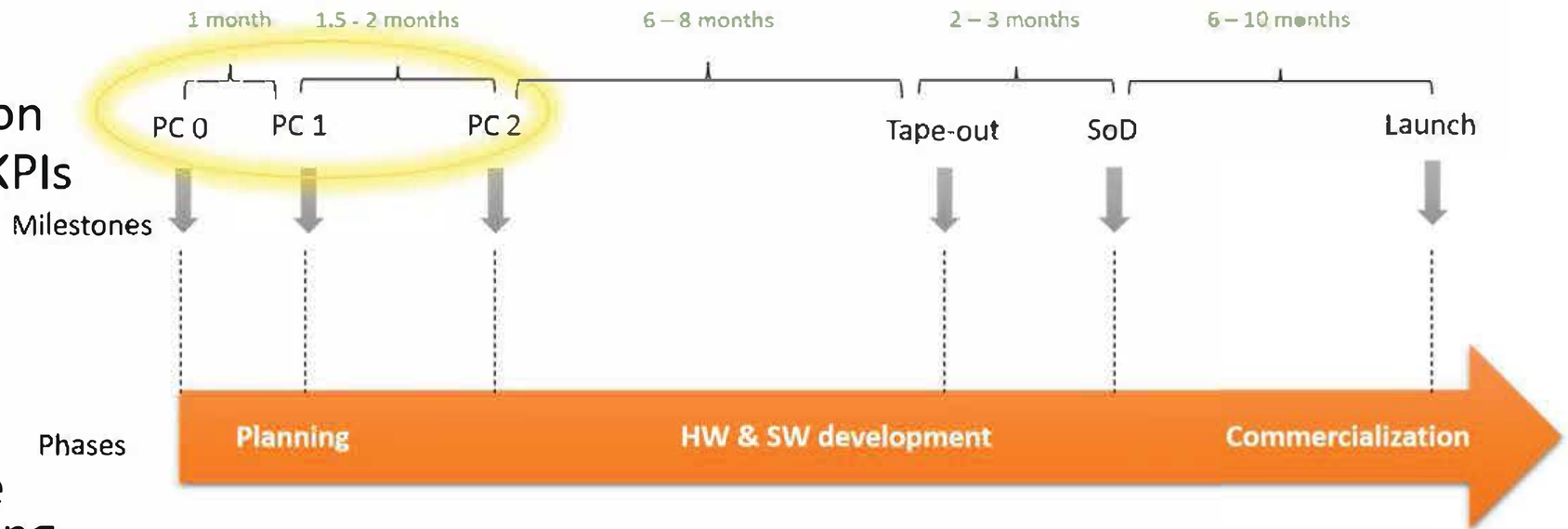


The total duration from planning to launch takes 18-24 months

Chipset lifecycle: Planning

PC – Product council phase
SoD – Silicon on Dock

- PC 0
  - Product concept articulation
  - Establish key use cases & KPIs
  - Business justification
  - Engineering commitment
- PC 1
  - Concept commit
  - Outline the features of the product, without committing
- PC 2
  - Finalize the features of the product
  - Assign resources

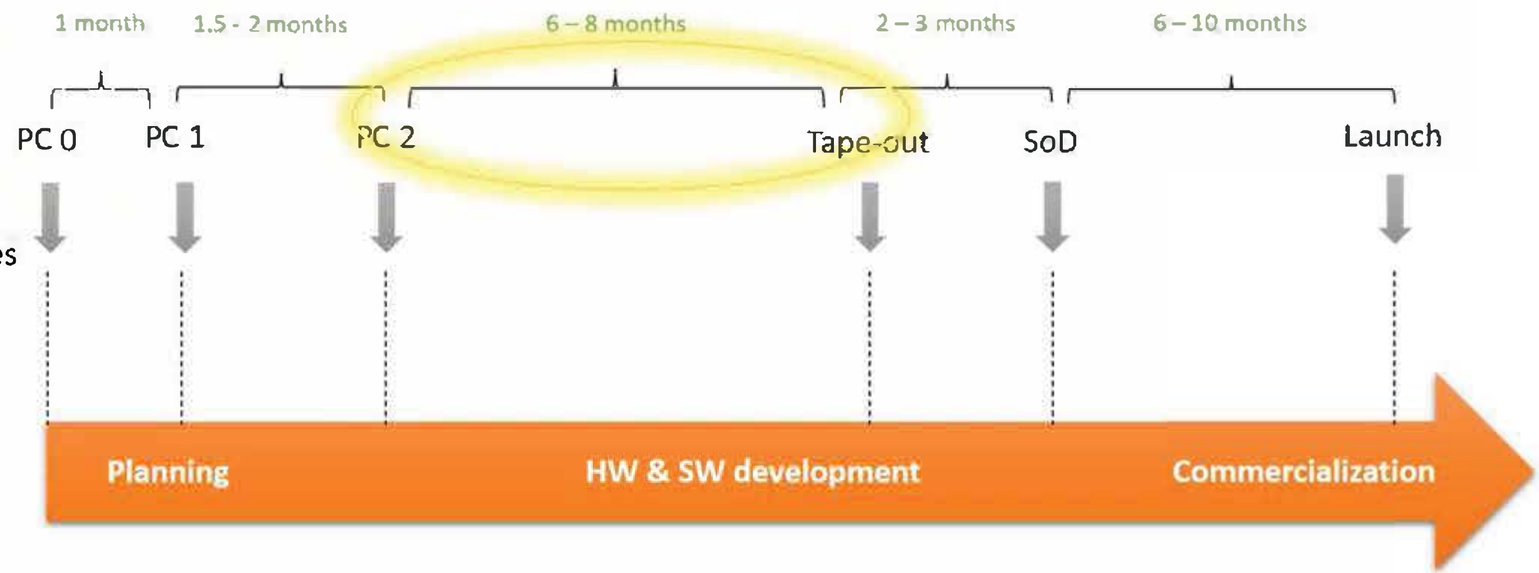


- Business case approval
- Hardware Architecture
- Software Architecture feasibility
- Key differentiating feature feasibility
- Identify risks / challenges
- High level impact assessment

Chipset lifecycle: Pre-silicon

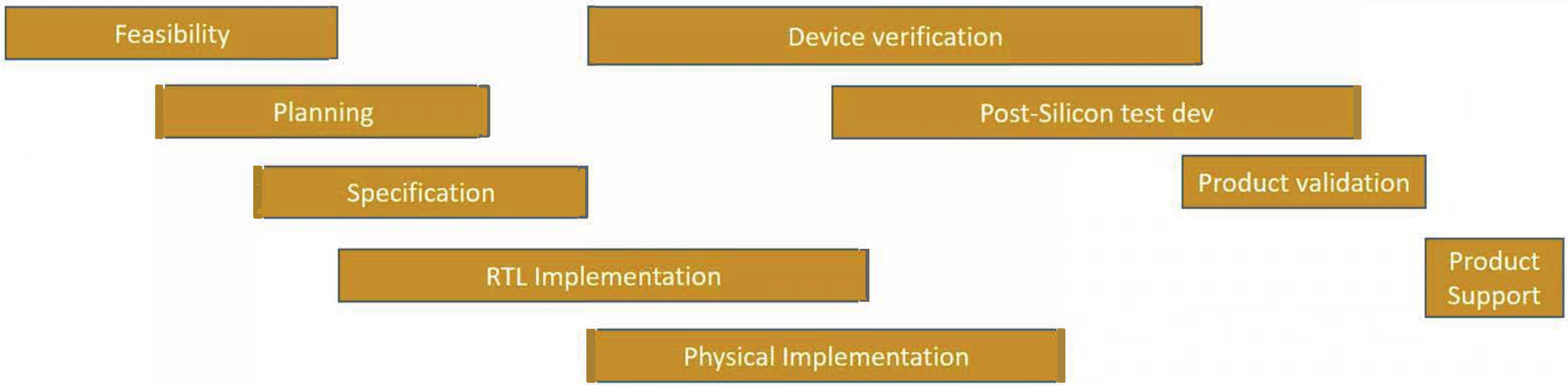
PC – Product council phase
SoD – Silicon on Dock

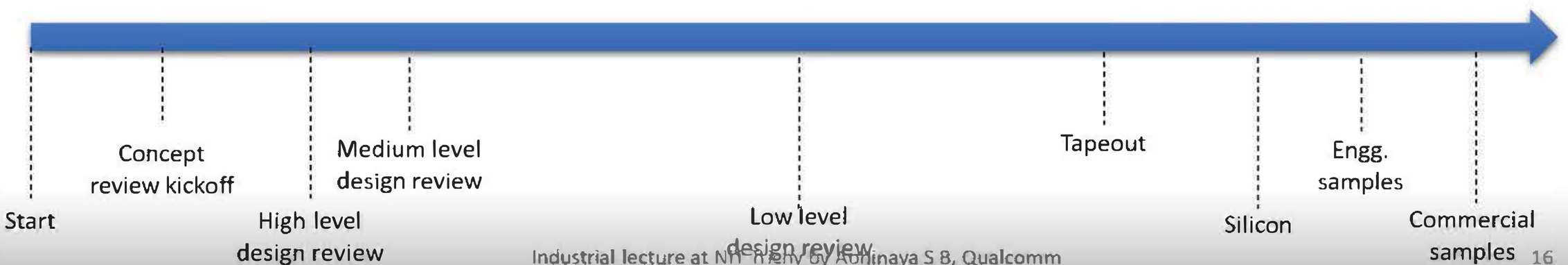
- Implementation of requirements via HW
- SW pre-silicon planning & execution
  - Brainstorm SW changes
  - Simulation / emulation schedule for testing platform
  - Define targets for power & perf KPIs
- Customer engagement for lead launch



- Hardware Implementation
- Hardware architecture
- Systems engineering
- VLSI development process

VLSI Development Process

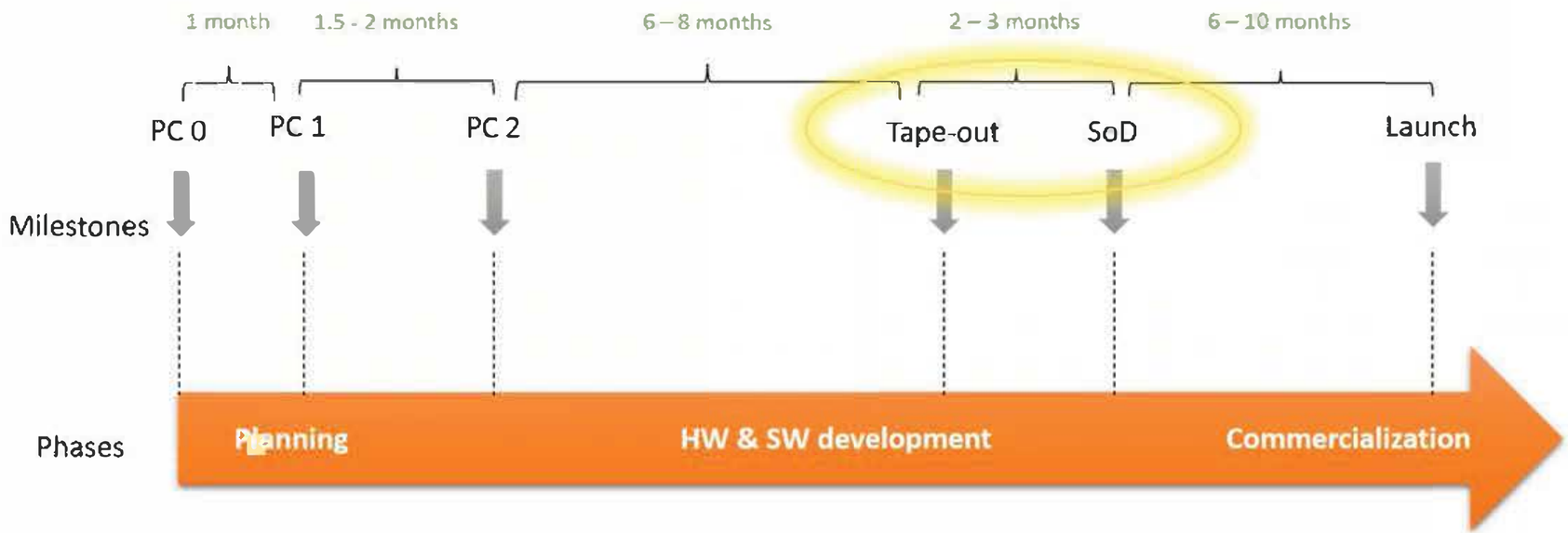




Chipset lifecycle: Silicon on Dock

PC – Product council phase
SoD – Silicon on Dock

- Software and associated systems prepare for chipset samples
- SoD:
  - Validate that all HW blocks are working as expected
  - Bring up all features of SoC



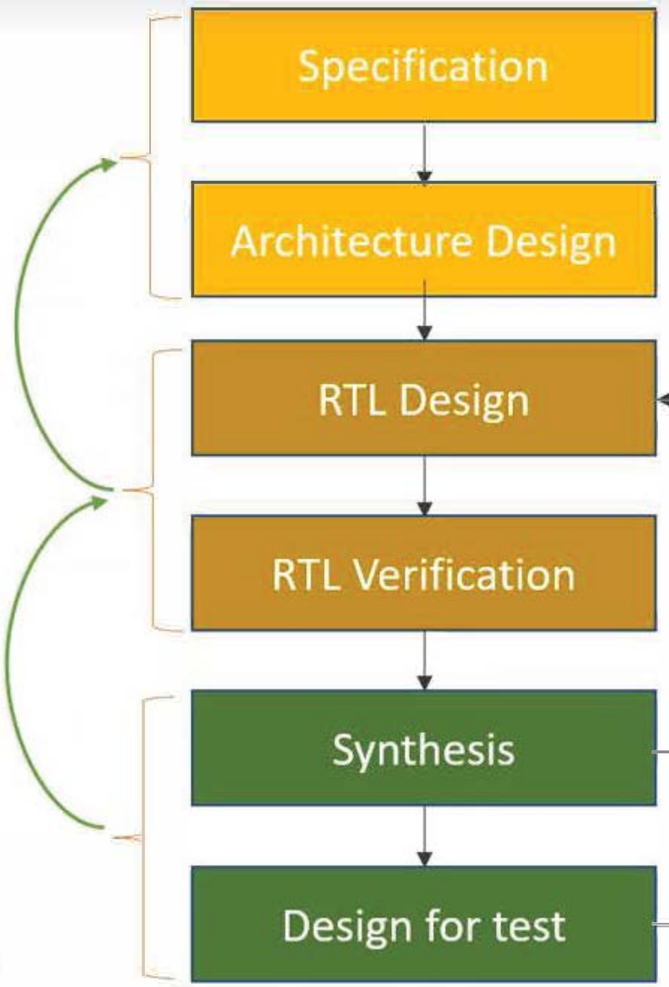
- Software & Hardware implementation (R&D)
- HW & SW architecture
- Digital, Analog, Physical design etc.
- Digital verification
- DSP engineering

ASIC flow

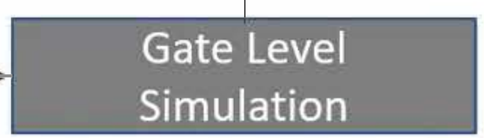
Pre-silicon verification
RTL code functionality should match system level spec

Formal Equivalence verification
Netlist should match with RTL code

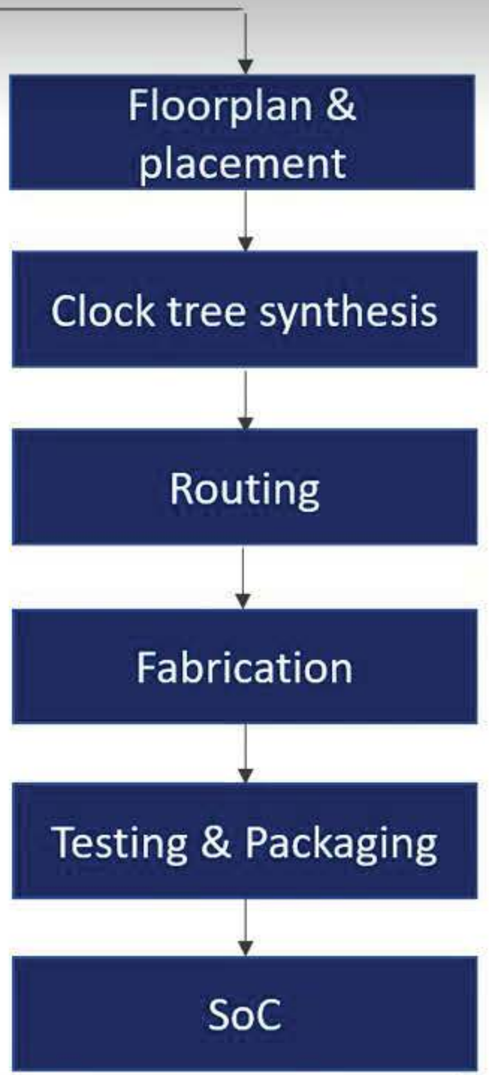
Layout vs Schematic verification
Layout should match netlist



Is the design meeting
Performance requirements?
Power envelope targeted?
Area targeted?
Schedule timelines?



Design for Test
Die should match with layout (during fabrication)



Post-silicon validation
To check for bugs in manufactured silicon

Chipset lifecycle: Silicon on Dock

PC – Product council phase
SoD – Silicon on Dock

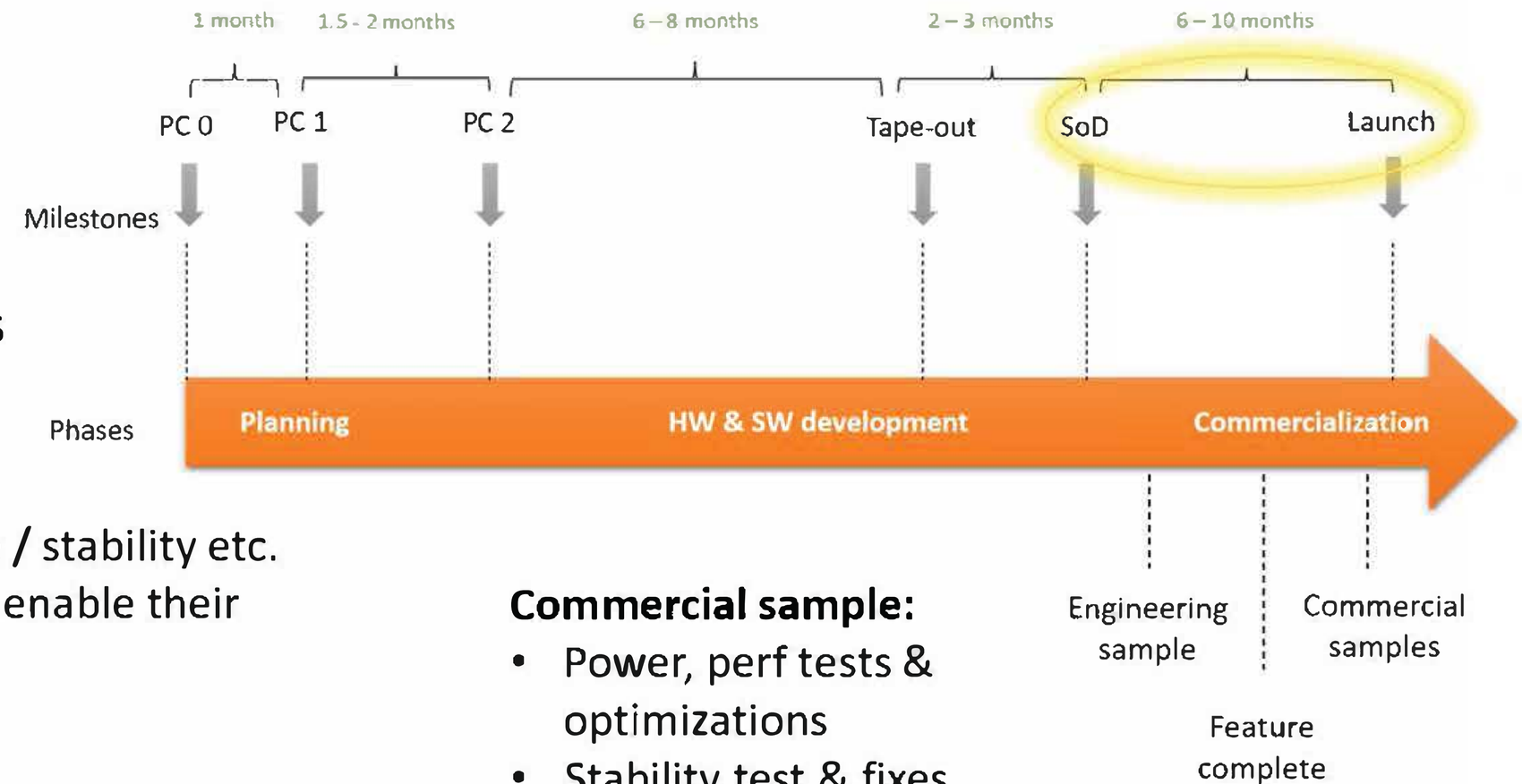
- Software and associated systems prepare for chipset samples
- SoD:
  - Validate that all HW blocks are working as expected
  - Bring up all features of SoC

- Software & Hardware implementation (R&D)
- HW & SW architecture
- Digital, Analog, Physical design etc.
- Digital verification
- DSP engineering

Chipset lifecycle: Post-Silicon

PC – Product council phase
SoD – Silicon on Dock

- Software development & test
- Hardware verification & modification
- Customer product launches



Engineering sample:

- Key features are up
- No guarantees on performance / stability etc.
- Early sampling to customers to enable their development process

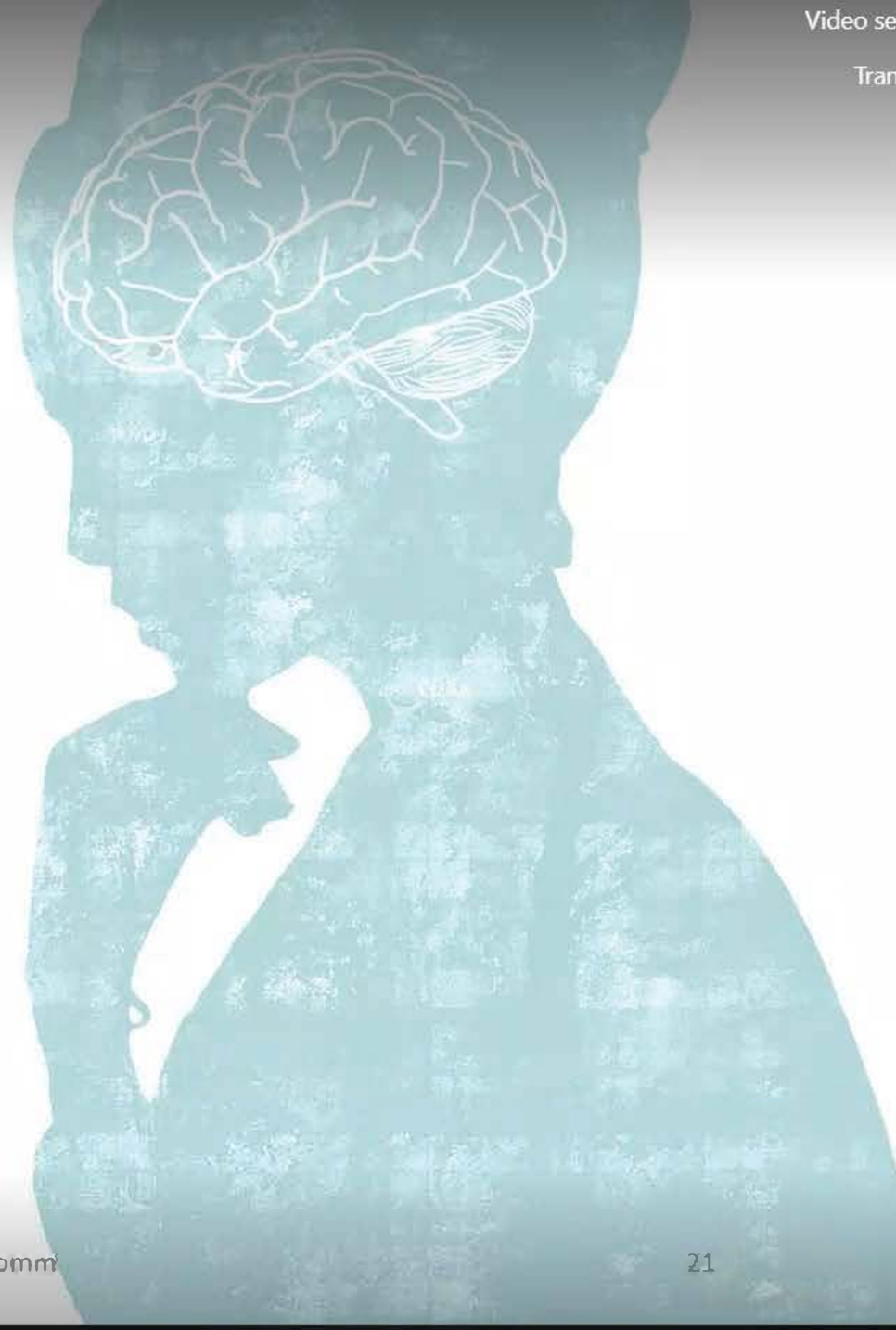
Feature complete:

- All features are up & verified
- Reasonable guarantee of stability & performance

Commercial sample:

- Power, perf tests & optimizations
- Stability test & fixes

CPU

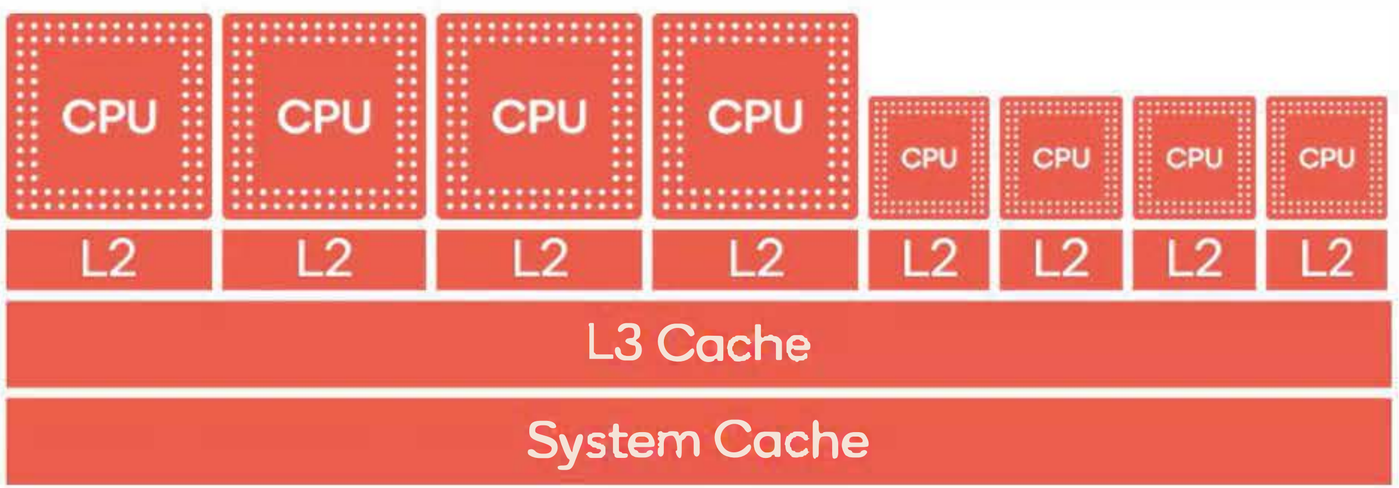


arm

- ARM is a family of reduced instruction set computer (RISC) instruction set architectures for computer processors
- Arm Ltd. develops the architectures and licenses them to other companies, who use it in their system on a chip (SoC) design
- It also designs cores that implement these instruction set architectures and licenses these designs to other companies
- Due to their low costs, minimal power consumption, and lower heat generation than their competitors, ARM processors are desirable for light, portable, battery-powered devices, including smartphones, laptops and tablet computers, and other embedded systems
- ARM is the most widely used family of instruction set architectures (ISA)

Qualcomm® Kryo™ CPU

- Have ARM-based big.LITTLE architecture
- big.LITTLE architecture is composed of *big* cores which are optimized for performance and *little* cores optimized for power efficiency



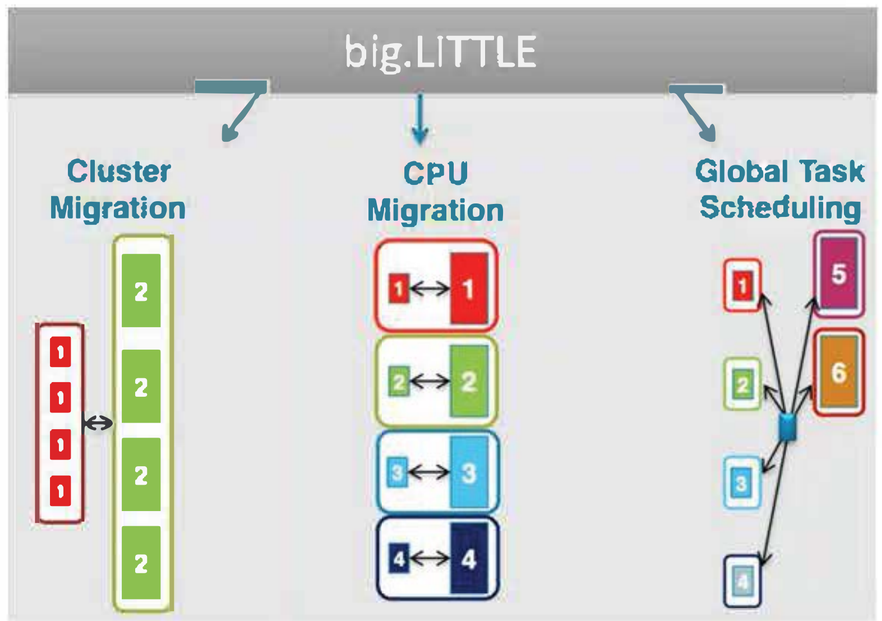
Qualcomm Snapdragon 865 has Kryo 585 CPU, based on Cortex-A77

Qualcomm® Kryo™ CPU: Power

Optimization level

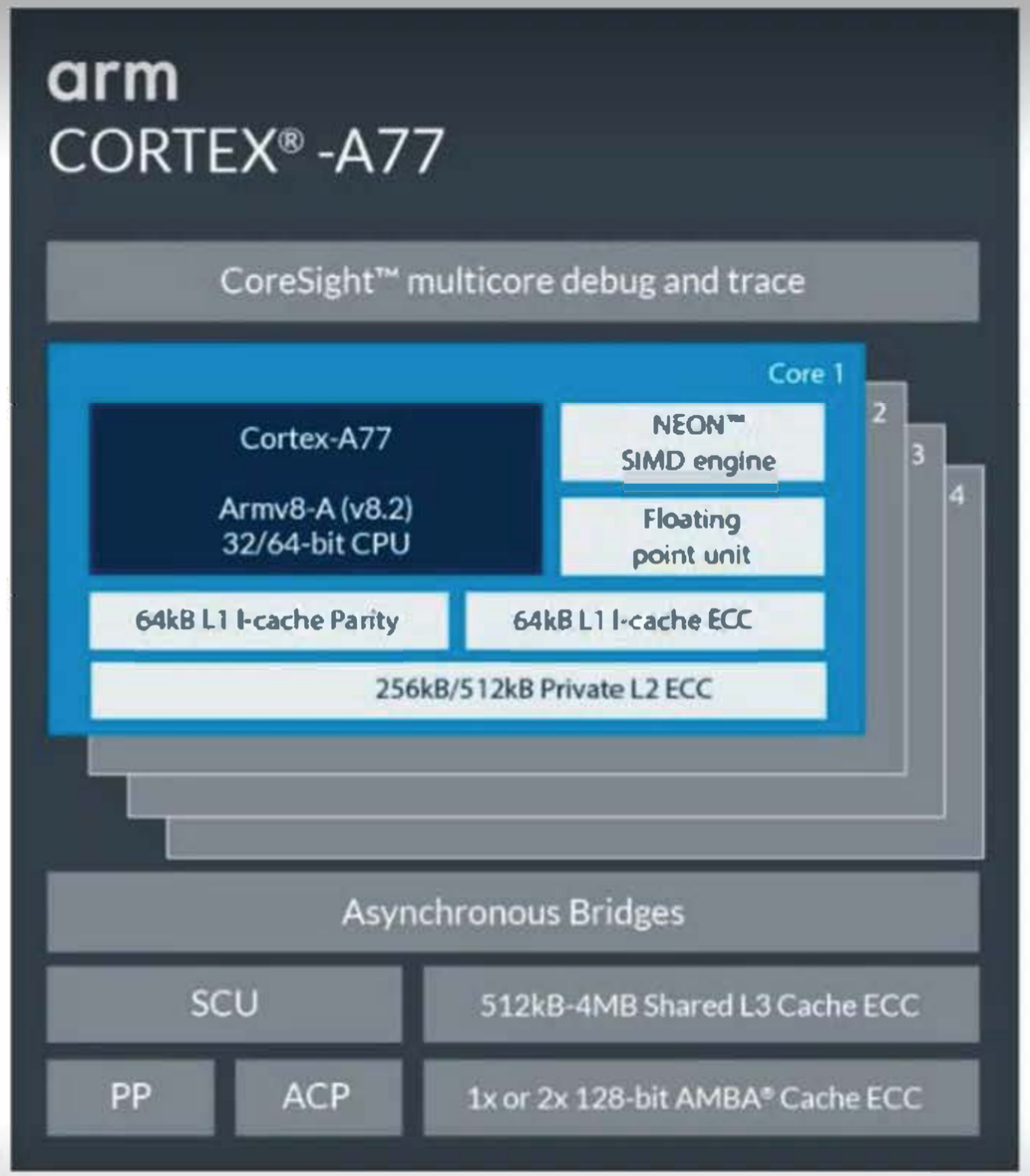
Task scheduling methods:

1. **Core clustering:** Cores of the same size treated as one cluster; most appropriate cluster chosen based on system demands
2. **In-kernel switching / CPU migration:** A big and little core are paired into a virtual core in which only one of the 2 physical cores in the virtual core is used (based on demand)
3. **Global task scheduling:** All physical cores are available all the time. Tasks allocated on a per-core basis depending on demands. Unused cores can be powered off.



NEON Intrinsics

- Advanced SIMD architecture for ARM processors
- The NEON instruction set makes a bank of specialized registers available for SIMD processing
- Includes typical SIMD operations for moving data between NEON and general purpose registers; for data processing and type conversion
- Hand-coded assembly, intrinsic functions, or automatic vectorization by the compiler can lead to tremendous performance gains for multimedia applications



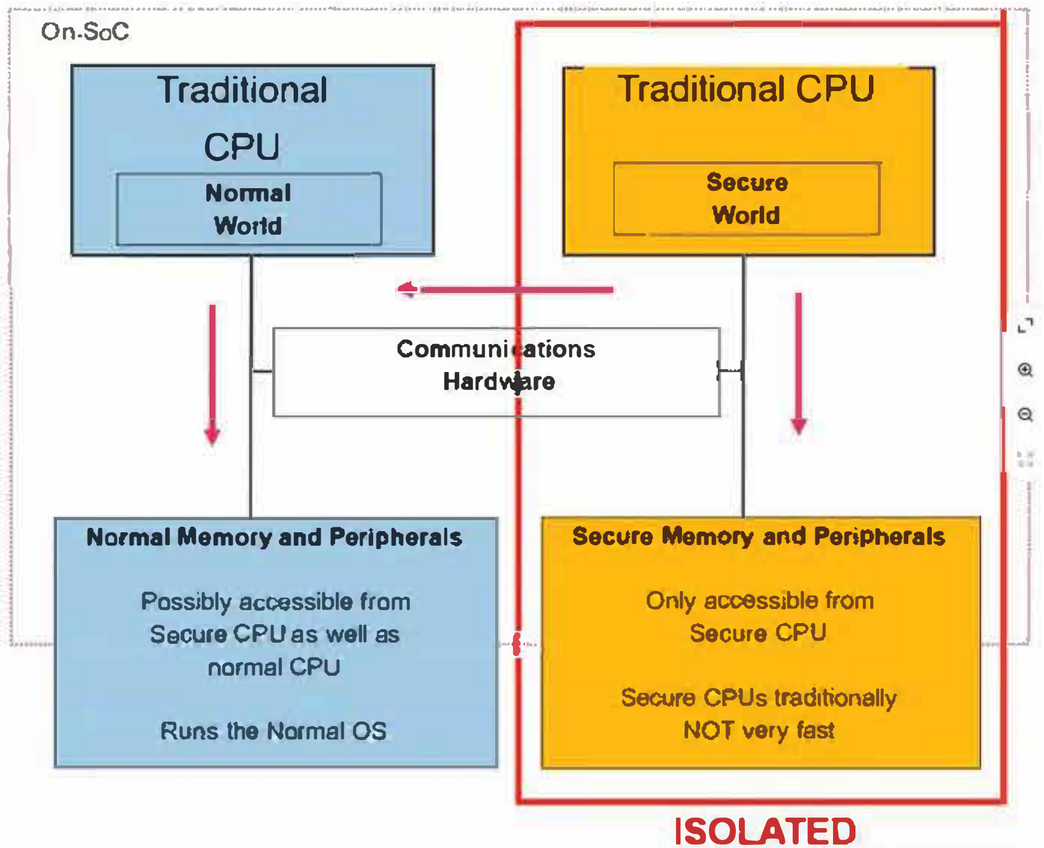
https://developer.qualcomm.com/sites/default/files/docs/adreno-gpu/developer-guide/cpu/cpu.html

ARM TrustZone

Efficient security solution with hardware-enforced isolation built into the CPU

Traditional Secure Architectures:

- 2 separate cores
  - Each core has access to its own memory "world" – either normal or secure
- Communication
  - Happens outside the CPU, but on the chip
- Two separate memory & peripheral systems
- Relies on being able to trust the security of the OS

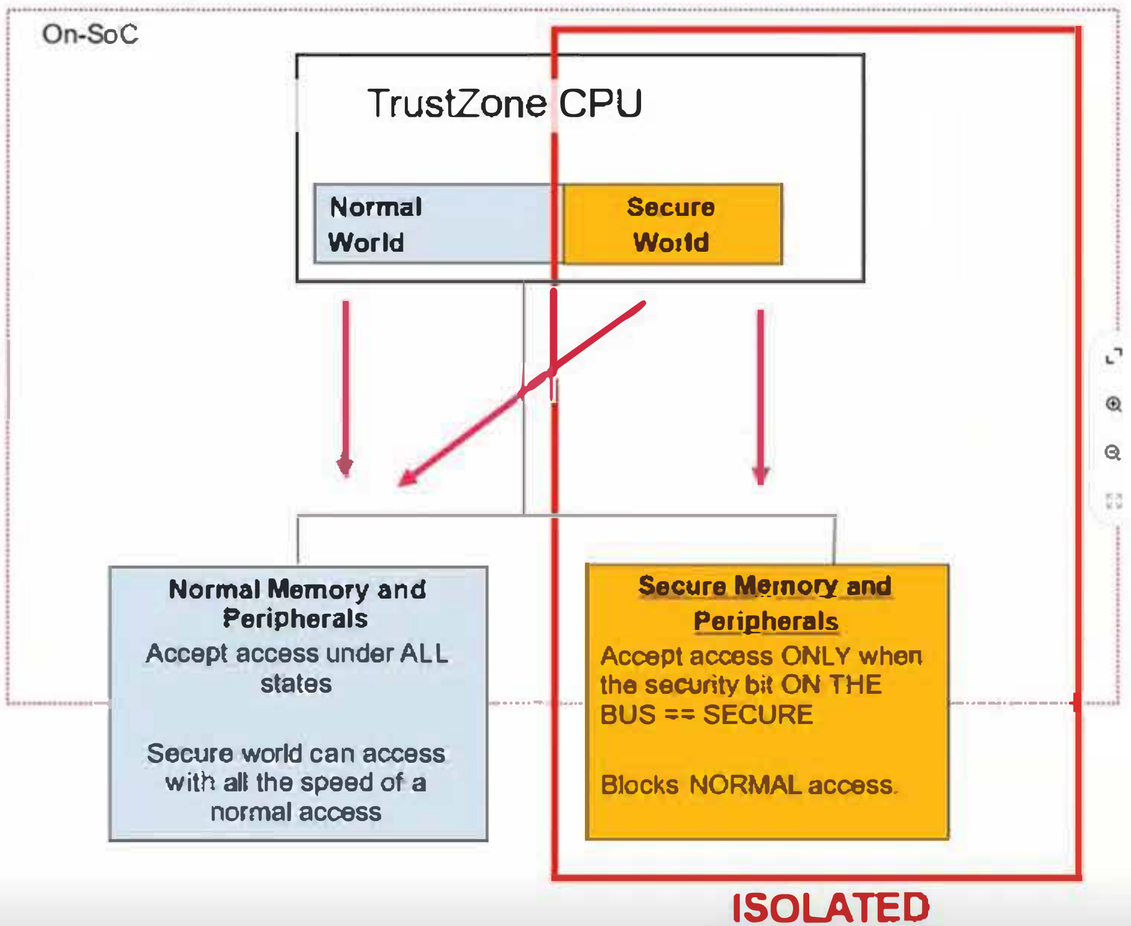


ARM TrustZone

Efficient security solution with hardware-enforced isolation built into the CPU

Traditional Secure Architectures:

- The core has 2 operating worlds - normal or secure
  - The MMU holds two separate states to accommodate this
  - The CPU separates secure and normal data
- Core state is reflected on the bus
  - Memory system can block normal access to the secure region



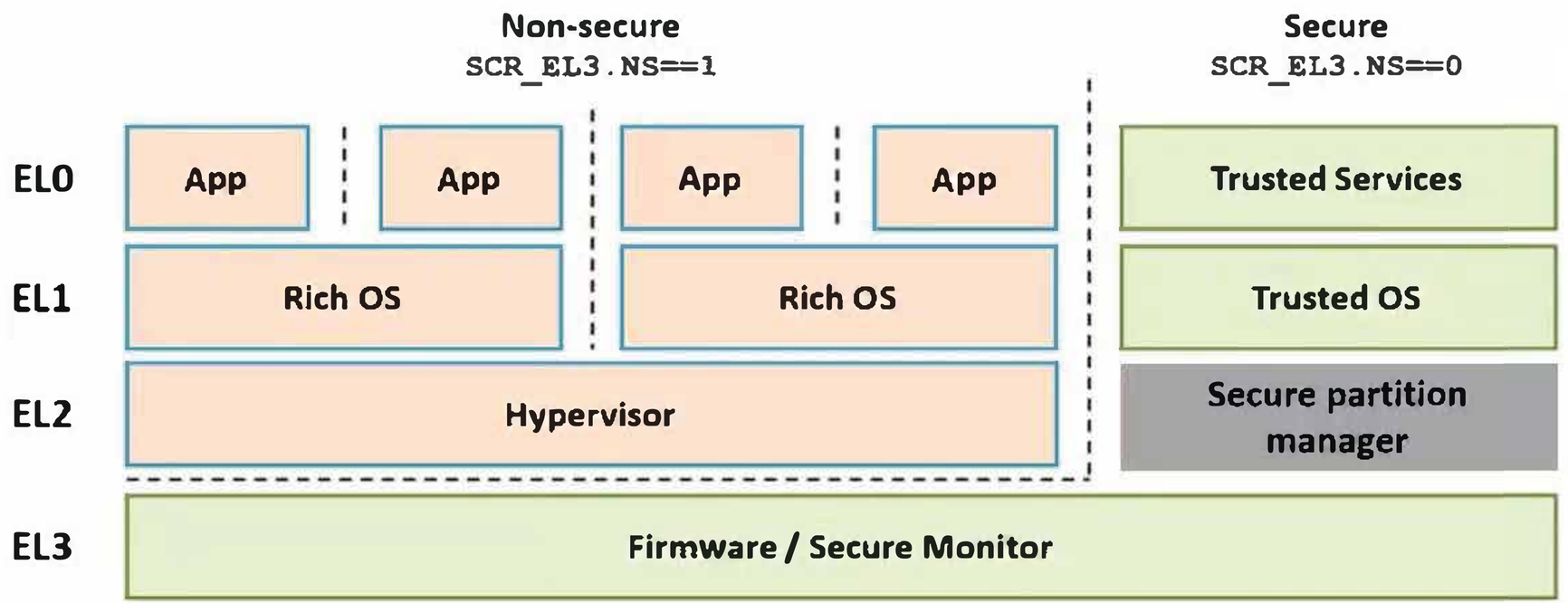
Exception levels

- AArch64 has 4 exception levels, and 2 security states
- AArch64 or ARM64 is the 64-bit extension of the ARM architecture

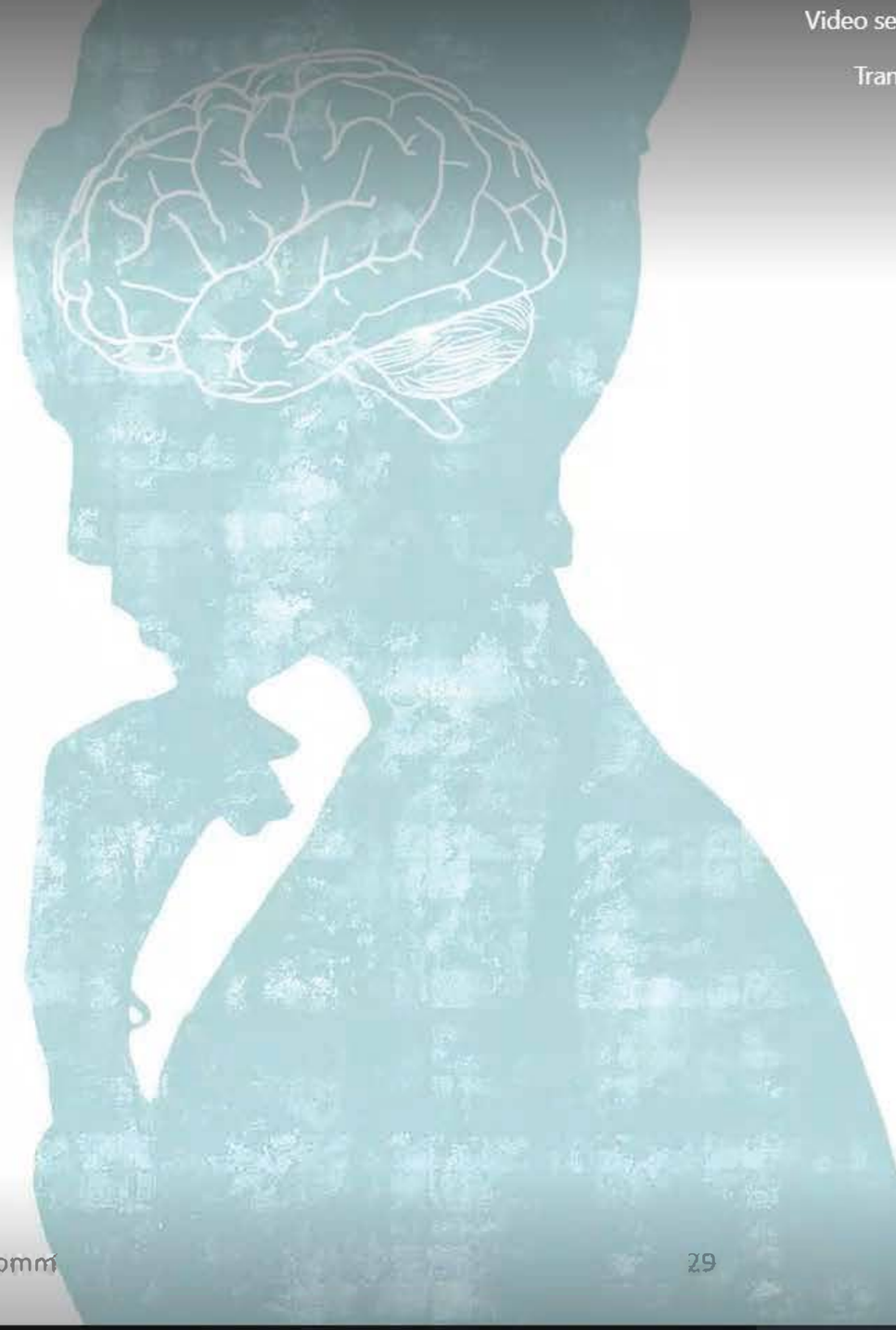
A **hypervisor**, also known as a virtual machine monitor or VMM, is software that creates and runs **virtual machines** (VMs).

A hypervisor allows one host computer to support multiple guest VMs by virtually sharing its resources, such as memory and processing.

Firmware is a specific class of computer software that provides the low-level control for a device's specific hardware



DSP



Hexagon DSP Processor

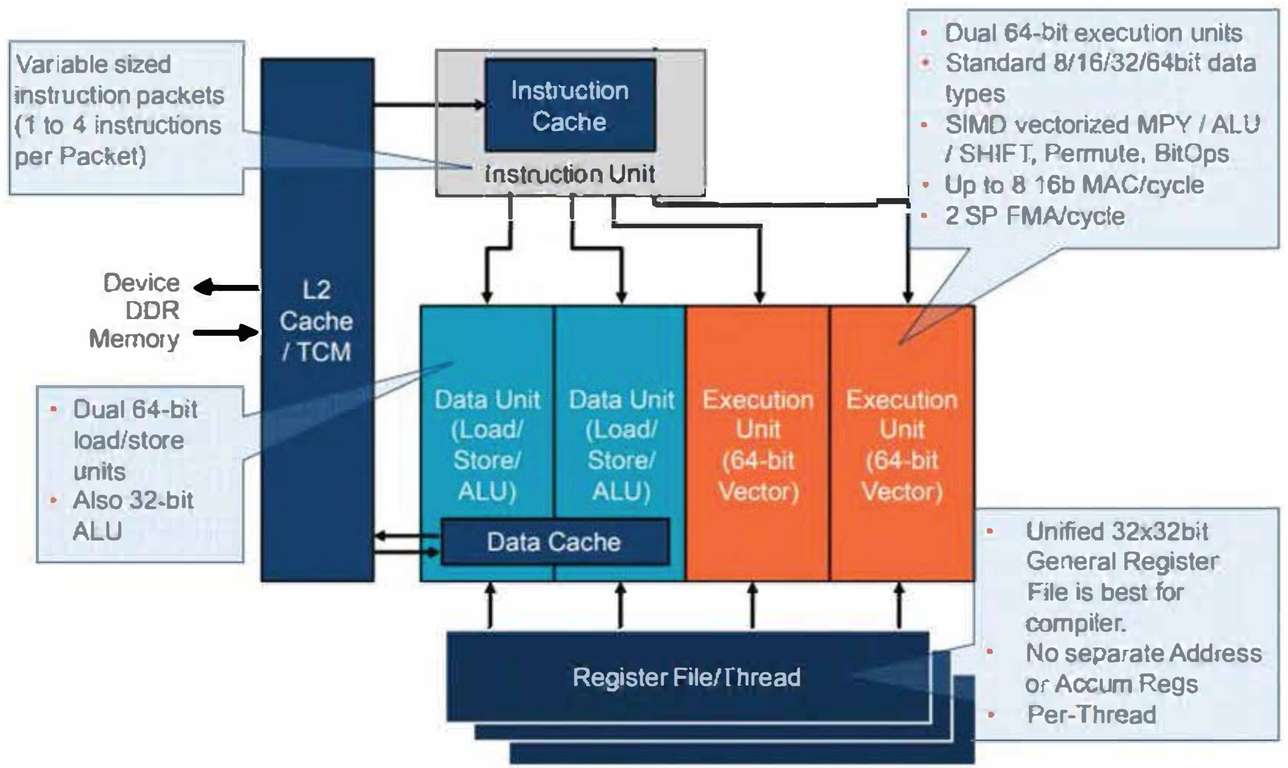
- Supports processing needs of multimedia and modem functions
- It is an advanced, variable instruction length, Very Long Instruction Word (VLIW) processor architecture with hardware multi-threading
- Compute DSP, Audio DSP, Modem DSP, Sensor DSP
- Hexagon cores are optimized for both high performance and energy efficiency (more important metric for embedded systems)
- Hardware multi-threaded to enable superior concurrency
- HW threads can be considered as separate processor cores with shared memory, and are programmed using conventional software threading

QuRT: An RTOS

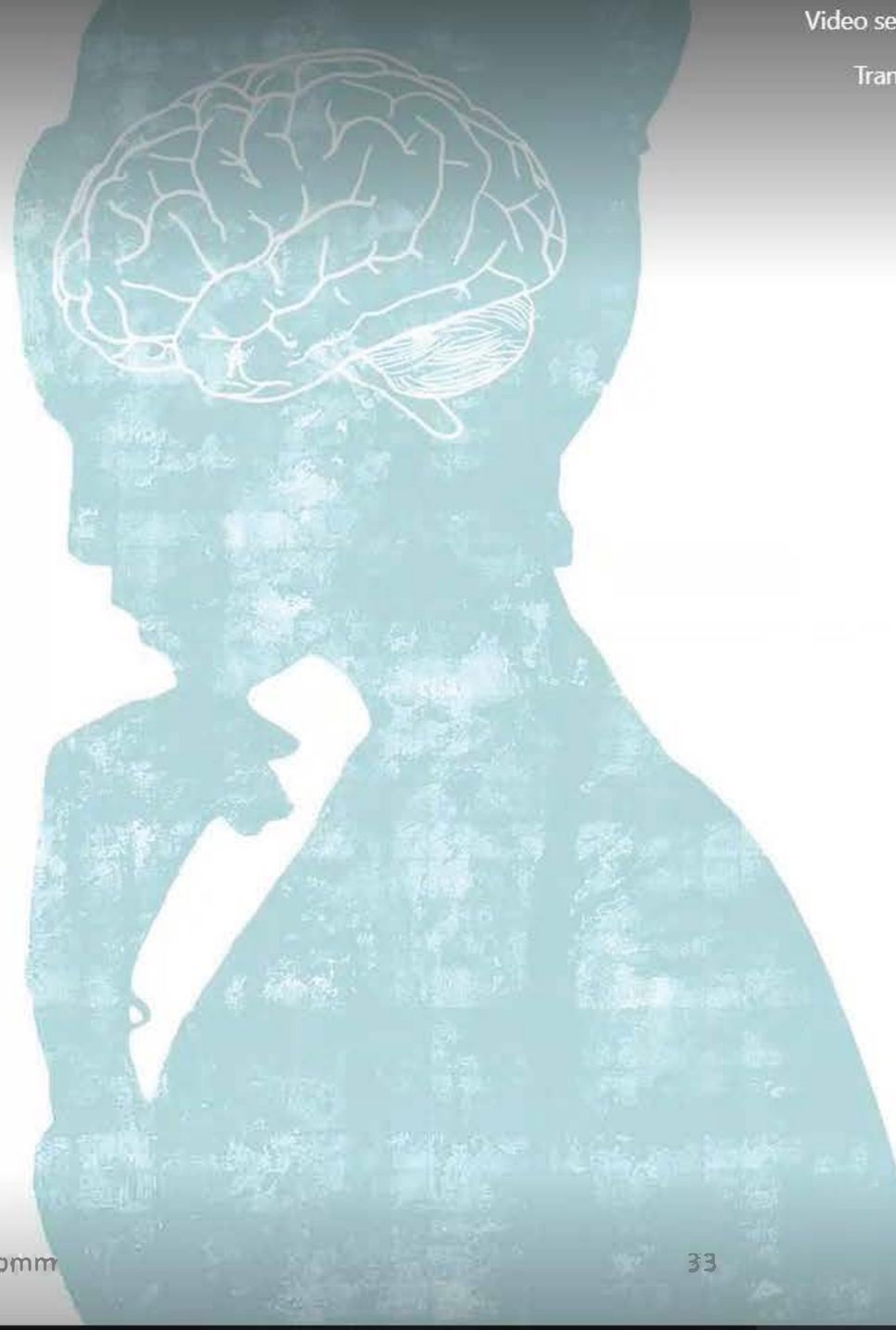
- GPOS's (General purpose OS) tend to work with HW in which each processor core runs a single thread of execution at a time
- RTOS (Real-time operating system) is designed to provide a predictable execution pattern, where processing needs to conform to the constraints of a time-bound system (processing to be completed at a certain frequency or the system as a whole will fail)
- An RTOS is typically light weight and small in size compared to a GPOS, and generally provides only the functionality required to run certain types of applications on specific hardware

VLIW: Very Long Instruction Word

- A computer processing architecture in which a language compiler or pre-processor breaks program instruction down into basic operations that can be performed by the processor in parallel
- Leverages instruction level parallelism

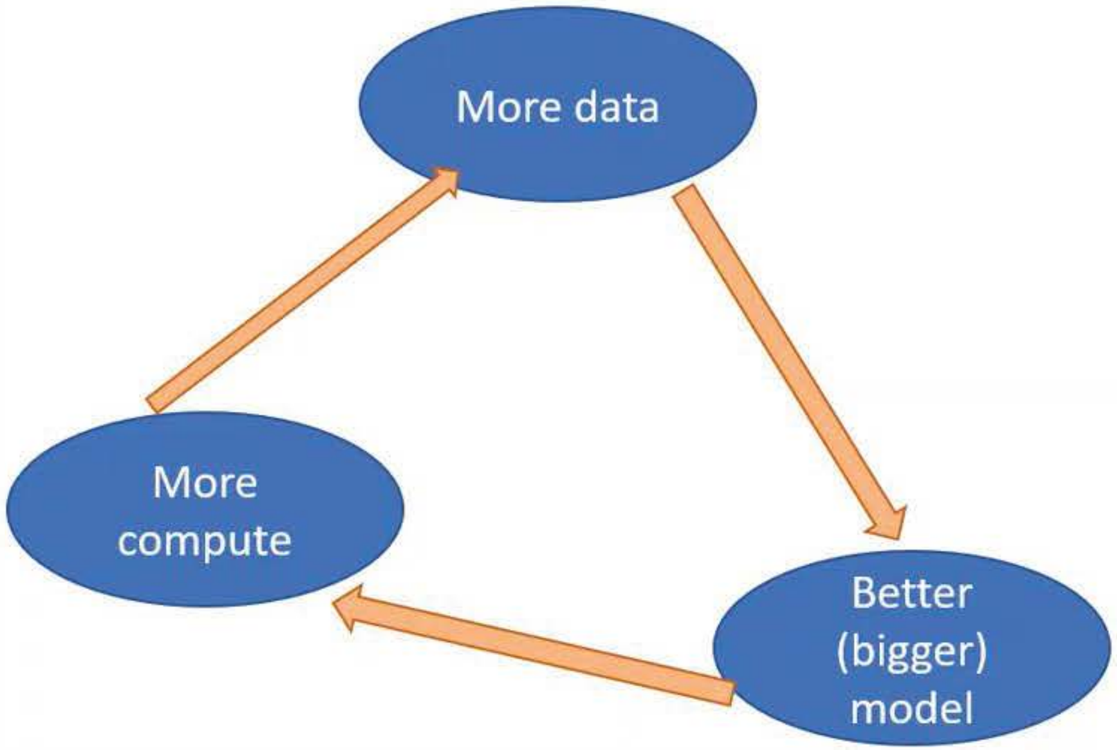


HW Accelerators

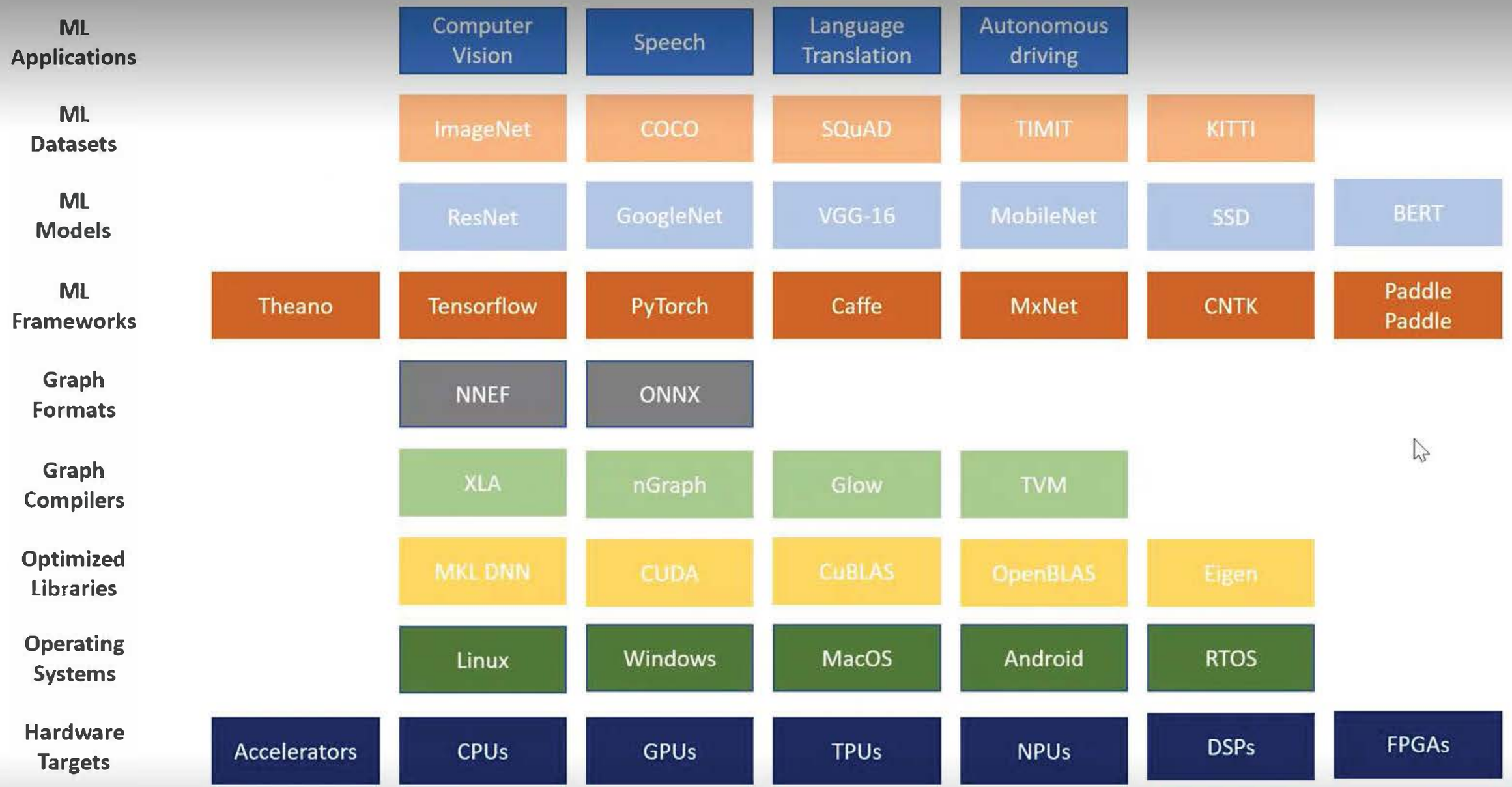


HW Accelerators

- Growth of AI – fuelled by computing power
- Improvement in DL algos hand-in-hand with improvement in HW accelerators
- Low power, real time inference
- Knowledge of computing system essential for algo efficiency



https://youtu.be/Qv0PDxQY6jY


ML Applications
Computer Vision
Speech
Language Translation
Autonomous driving
ML Datasets
ImageNet
COCO
SQuAD
TIMIT
KITTI
ML Models
ResNet
GoogleNet
VGG-16
MobileNet
SSD
BERT
ML Frameworks
Theano
Tensorflow
PyTorch
Caffe
MxNet
CNTK
Paddle Paddle
Graph Formats
NNEF
ONNX
Graph Compilers
XLA
nGraph
Glow
TVM
Optimized Libraries
MKL DNN
CUDA
CuBLAS
OpenBLAS
Eigen
Operating Systems
Linux
Windows
MacOS
Android
RTOS
Hardware Targets
Accelerators
CPUs
GPUs
TPUs
NPUs
DSPs
FPGAs

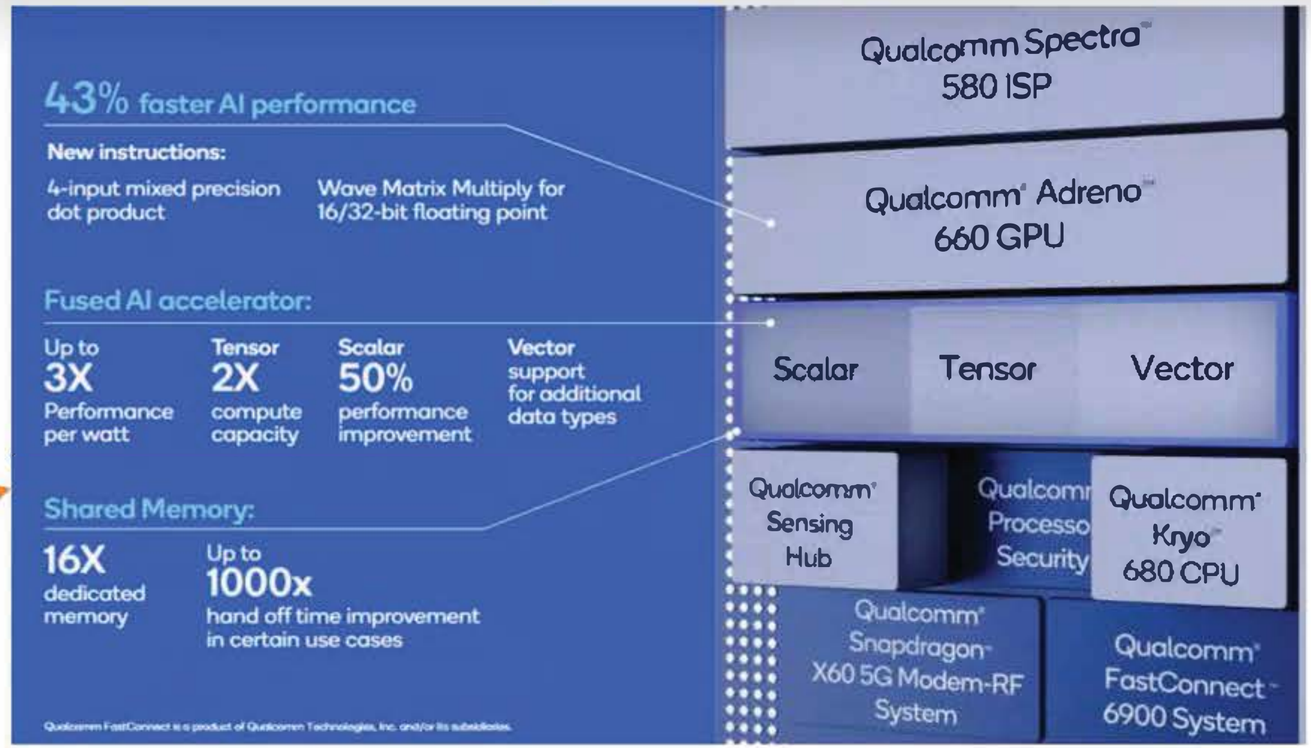
Evaluating AI algos: Considerations

- Accuracy & Inference latency
- IoT, Autonomous driving, Mobile systems, drone, robotics
  - Energy efficiency
  - Reliability, security, privacy
  - Model size, number of computations (memory footprint, power)
  - Weight, cost

- Need domain specific accelerators!

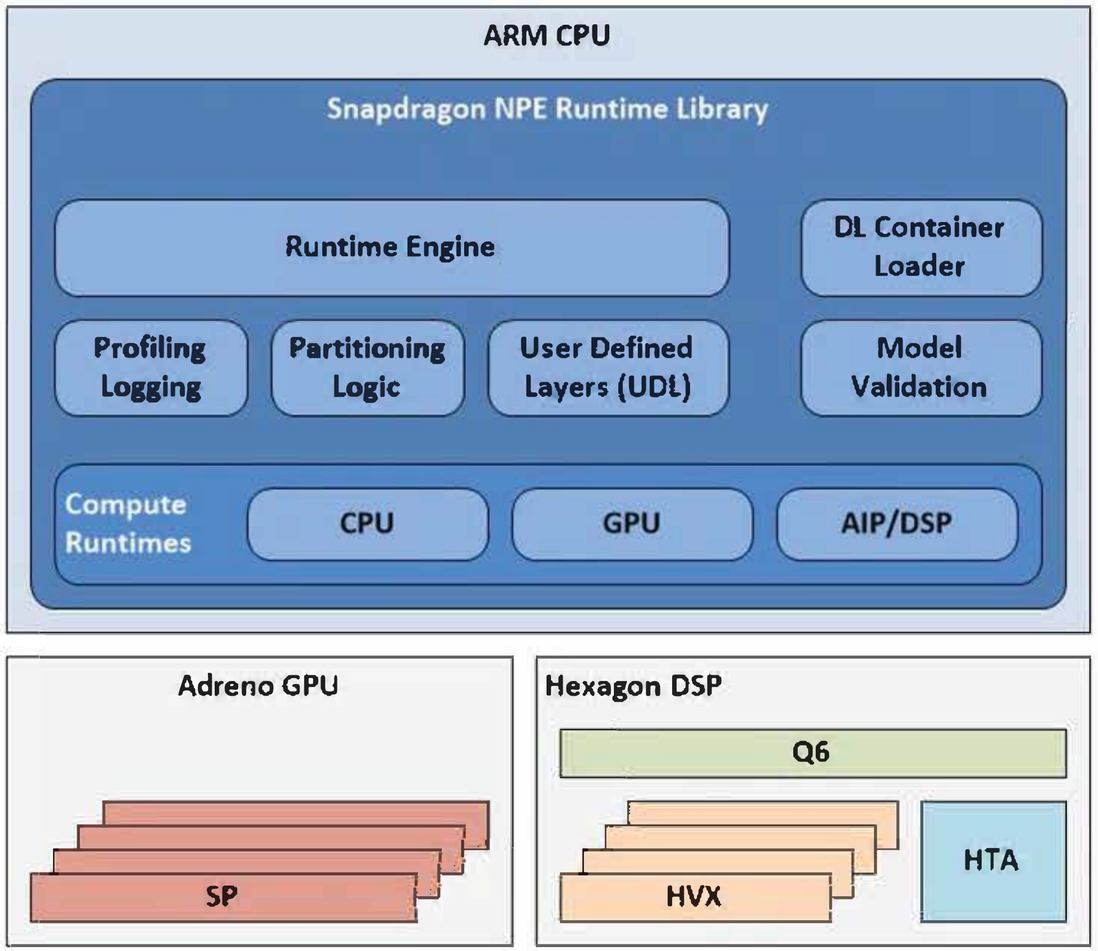
Accelerators in today's market

Source	HW accelerators for AI processing
Amazon	AWS Inferentia
Alibaba	Ali-NPU
Baidu	Kunlun
Bitmain	Sophon
Cambricon	MLU
Google	TPU
Graphcore	IPU
Intel	NNP, Myriad, EyeO
Nvidia	NVDLA
Huawei	Ascend
Apple	Neural Engine
Samsung	NPU
Qualcomm	AI Engine



- Scalar + Vector + Tensor processing units
- Hexagon processor with fused AI accelerator arch
- 16x larger memory (1000x better data access)
- 26 TOPS

Snapdragon NPE Runtime



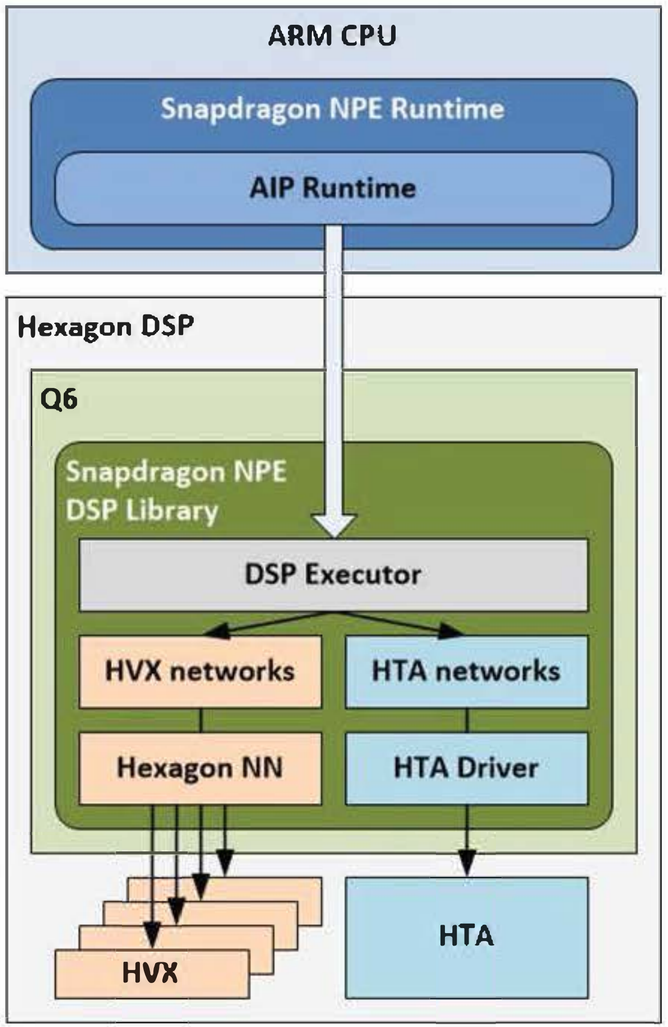
DL Container Loader : Loads a DLC (Deep Learning Container .dlc file) created by one of the snpe-*framework*-to-dlc conversion tools

Model Validation : Validates that the loaded DLC is supported by the required runtime

Runtime Engine : Executes a loaded model on requested runtime(s)

CPU: supports 32-bit FP or 8-bit quantized execution
GPU: supports hybrid or full 16-bit FP modes
DSP: runs the model on Hexagon DSP using Q6 and Hexagon NN, executing on HVX; supports 8-bit quantized execution
AIP: Runtime Runs the model on Hexagon DSP using Q6, Hexagon NN, and HTA; supports 8-bit quantized execution.

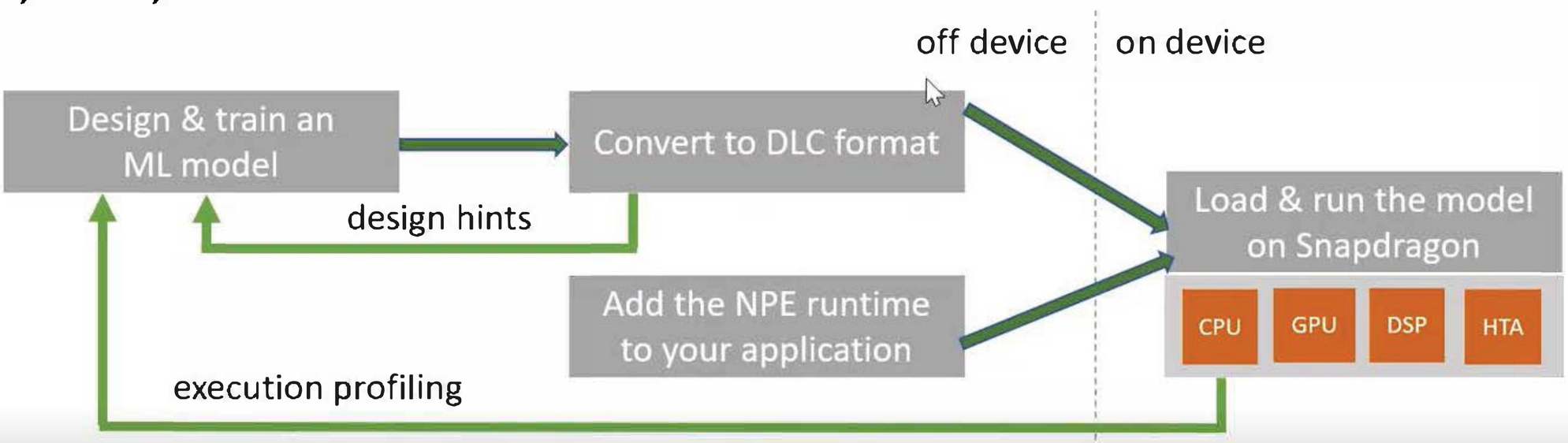
Snapdragon AIP Runtime



- The AIP (AI Processor) Runtime is a software abstraction of Q6, HVX and HTA into a single entity (AIP) for the execution of a model across all three
- A user, who loads a model into Snapdragon NPE and selects the AIP runtime as a target, will have parts of the model running on HTA, and parts on HVX, orchestrated by the Q6
- In order to execute parts of the model on HTA, the model needs to be analyzed offline, and binaries for the relevant parts need to be embedded into the DLC

AI SDK

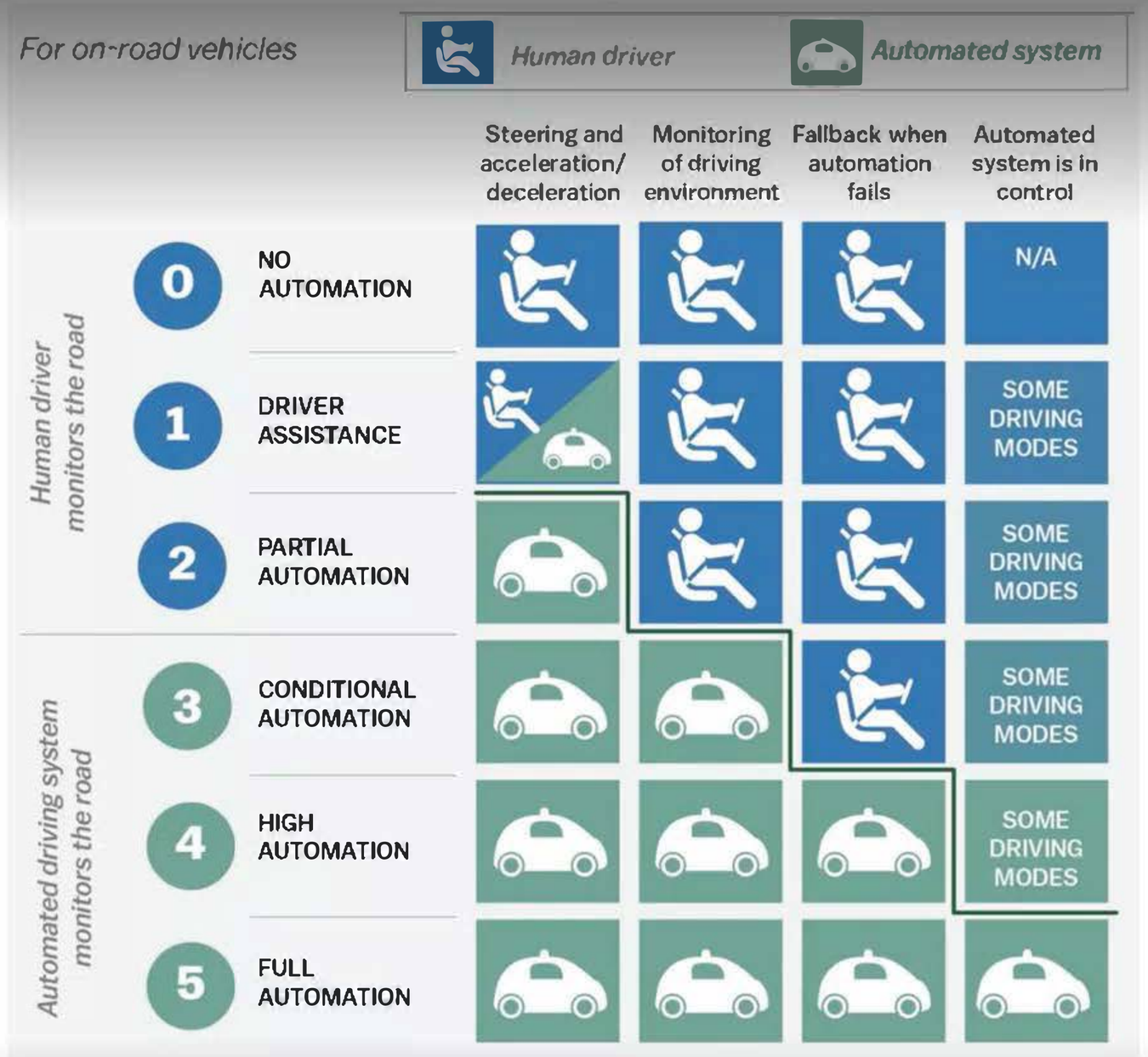
- For optimized ML inference on HW platform
- Qualcomm Neural Processing SDK is designed to help developers run one or more neural network models trained in Caffe/Caffe2, ONNX, or TensorFlow on Snapdragon mobile platforms, whether that is the CPU, GPU, DSP or the AI accelerator



Innovations & Optimizations at various levels

- Model uses lesser parameters
- Model converted to a format most efficient for the intended target
- Target has efficient data access strategies
- Smart power optimizations depending on what process in running
- Parallel processing
- Memory optimizations
- Computation strategies with maximum data reuse

Autonomous Driving



LEVELS OF DRIVING AUTOMATION

0	1	2	3	4	5
NO AUTOMATION	DRIVER ASSISTANCE	PARTIAL AUTOMATION	CONDITIONAL AUTOMATION	HIGH AUTOMATION	FULL AUTOMATION
Manual control. The human performs all driving tasks (steering, acceleration, braking, etc.).	The vehicle features a single automated system (e.g. it monitors speed through cruise control).	ADAS. The vehicle can perform steering and acceleration. The human still monitors all tasks and can take control at any time.	Environmental detection capabilities. The vehicle can perform most driving tasks, but human override is still required.	The vehicle performs all driving tasks under specific circumstances. Geofencing is required. Human override is still an option.	The vehicle performs all driving tasks under all conditions. Zero human attention or interaction is required.

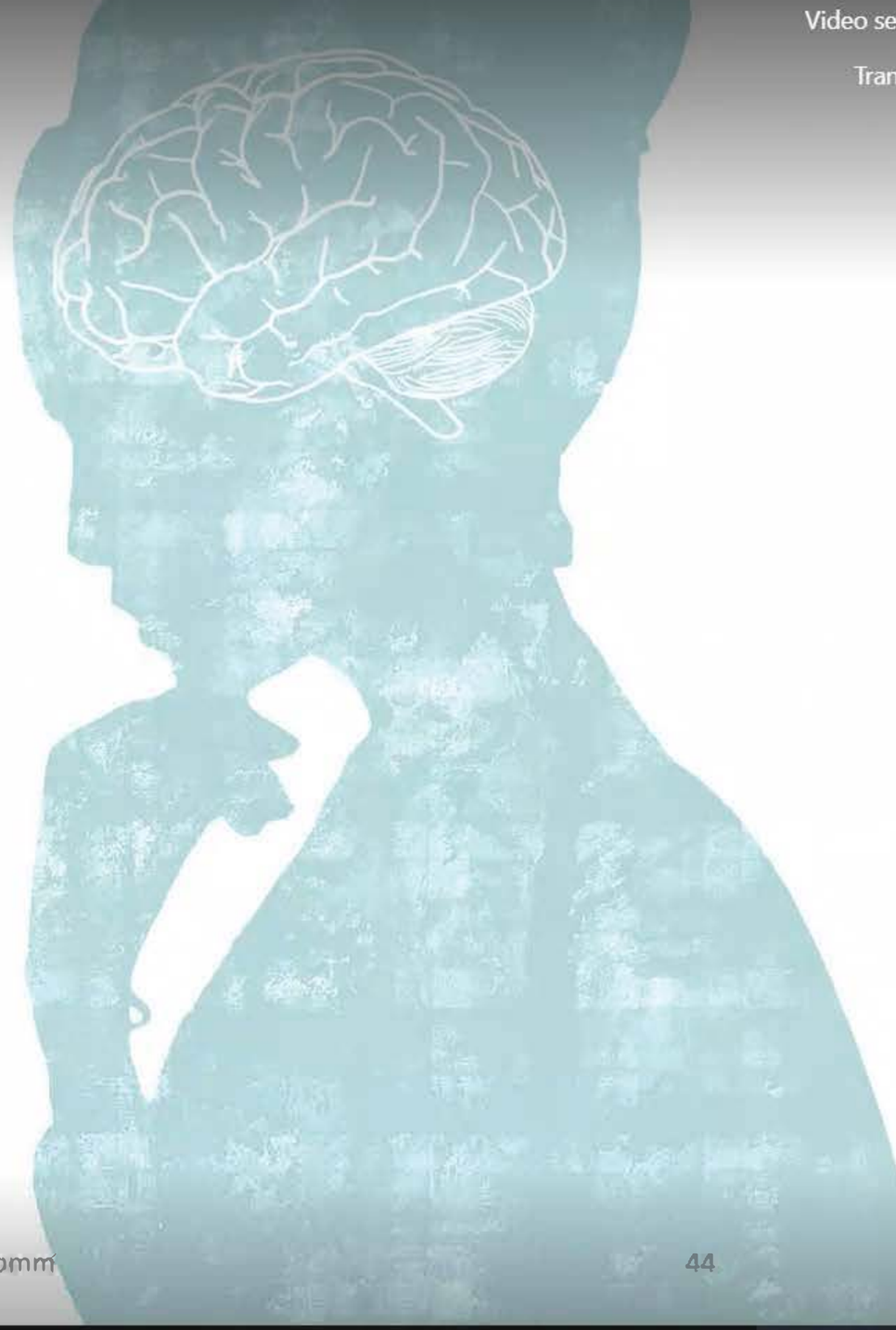
THE HUMAN MONITORS THE DRIVING ENVIRONMENT

THE AUTOMATED SYSTEM MONITORS THE DRIVING ENVIRONMENT

https://www.synopsys.com/automotive/autonomous-driving-levels.html

ADAS - Advanced Driver Assistance Systems

Benchmarking & KPIs



Benchmarking & KPIs

Key Performance Indicators:

- Industry standard benchmarks
  - Antutu
  - Geekbench
  - Inception
  - DXO scores
- Application specific metrics
  - Camera resolution
  - Image quality
  - Frame rate
  - Call quality
  - Refresh rate

Key metrics & knobs:

- Memory bandwidth
- Access latency
- MIPS / TOPS
- Use case concurrency
- Priority, QoS
- Peak / average BW
- Data compression
- Bottleneck resolution
- Scheduled housekeeping

Microbenchmarks
Memory latency, memory BW, cache BW, Cache maintenance operations, TLB, Barriers, Crypto

Realworld Usecases
Launch latency, cache sensitivity, SW optimizations

Synthetic Benchmarks
LMBench, Dhrystone, Coremark, Linpack, Stream

Mobile Benchmarks
Antutu, Geekbench, SPEC, Octane, Kraken

CPU Performance

Synthetic benchmarks

CoreMark, Dhrystone, and Whetstone are processor core related benchmarks.

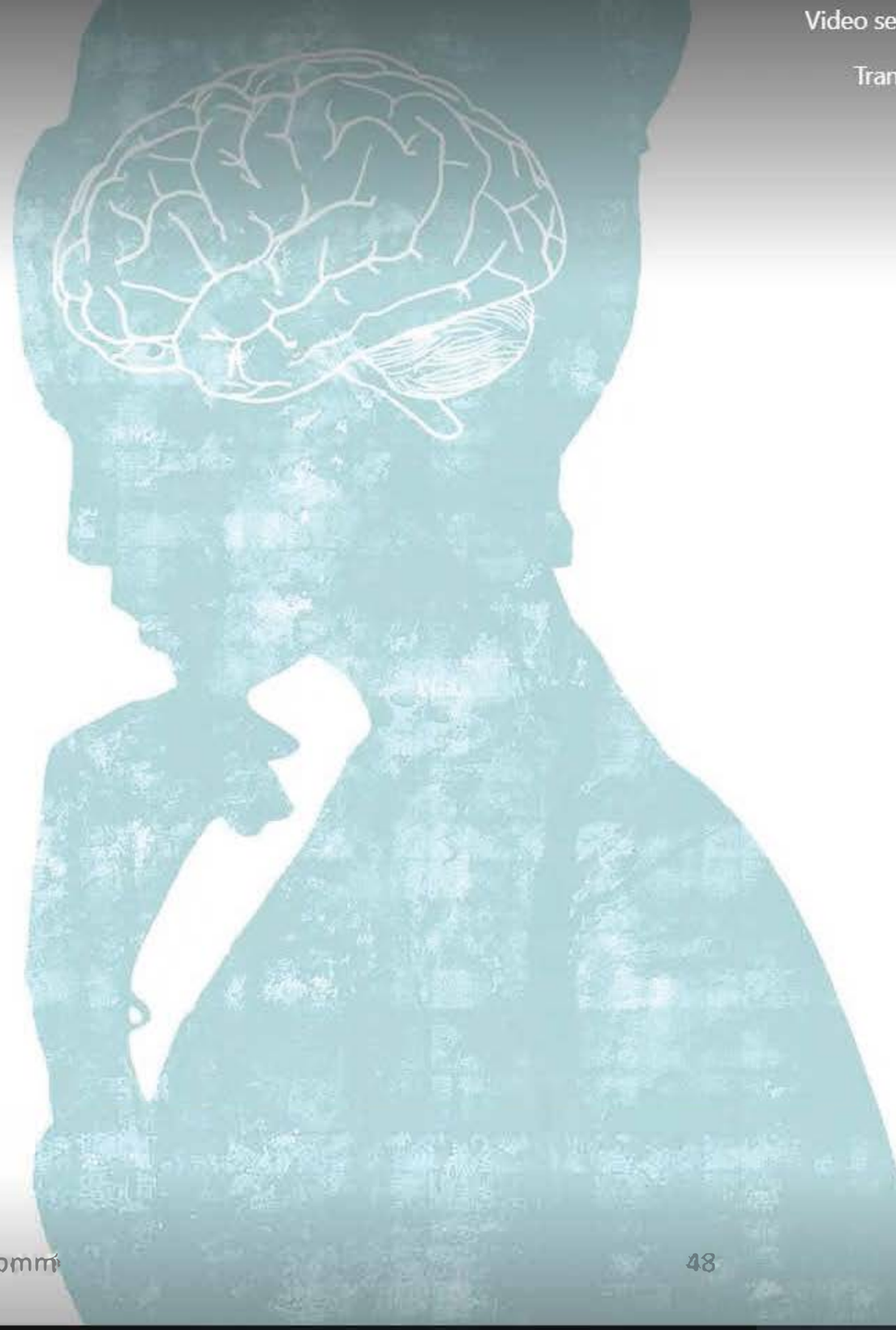
Benchmark	Emphasis	Hardware elements utilized
CoreMark	Processor core, cache memory R/W	CPU Pipeline, L1 cache
Dhrystone	Integer and branch operations	CPU Pipeline, Integer ALU, L1 cache
Whetstone	Floating point operations	CPU Pipeline, NEON/FPU, L1 cache

STREAM and LMbench are memory intensive benchmarks.

Benchmark	Emphasis
STREAM	Mostly sequential data stream of memory writes and reads
LMBench	Random memory writes and reads

https://rocketboards.org/foswiki/pub/Documentation/LinuxBenchmarking/How_to_Run_CV_SoC_benchmarks_051316.pdf

Secure Coding



Secure Coding: Buffer Overflows

- Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another
- A buffer overflow (or overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer
- The program attempting to write the data to the buffer overwrites adjacent memory locations
- Caused due to improper input checks or insufficient memory allocation
- Buffer overflows on executable code can cause unpredictable behaviour or crashes

Buffer Overflows: Unsafe functions

```
char *gets(char *str)
```

Reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.

```
char *strcat(char *destination, const char *source)
char* strcpy(char* destination, const char* source)
```

They do not check the length of the input being copied to the buffer!

`strcpy` does not guarantee NULL termination!!

Buffer Overflows: Safe functions

```
char *fgets(char *str, int n, FILE *stream)
```

Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

```
char* strncat(char* dest, const char* src, size_t count)
```

Appends the first count characters of source to destination, and ends with a NULL char.

```
char* strncpy(char* dest, const char* src, size_t count)
```

Copies the first count characters from source to destination, or until it reaches end of source string (NULL char).

Cumbersome to keep track of available space in buffer!

`strncpy` still does not guarantee NULL termination!!

Do not inform the caller of source string truncation!!!

Buffer Overflows: Safer functions

```
size_t strlcat(char *dest, const char *src, size_t size)
```

Appends the NULL-terminated source string to the end of destination string. It will append at most (size - strlen(dst) – 1) bytes, NULL-terminating the result.

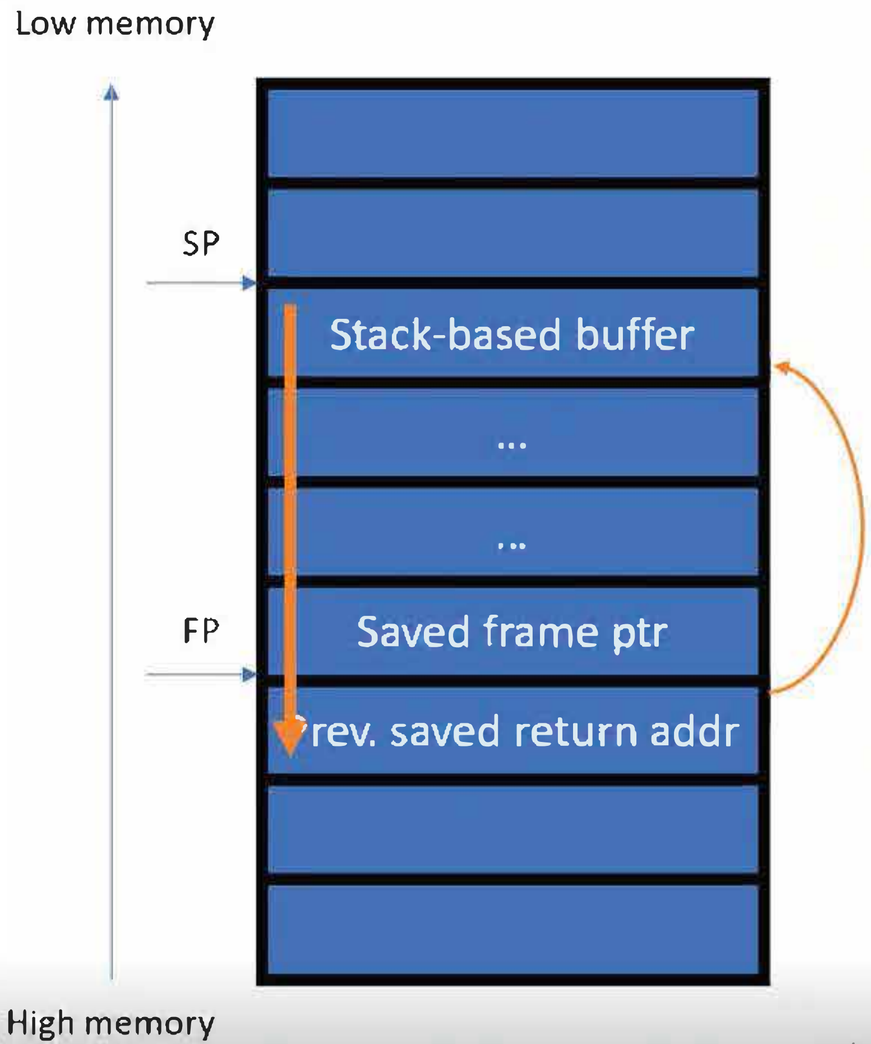
```
size_t strlcpy(char *dest, const char *src, size_t size)
```

Copies up to size - 1 characters from the NULL-terminated source string to destination string, NULL-terminating the result.

strl* truncate the source if necessary, result is always NULL-terminated.

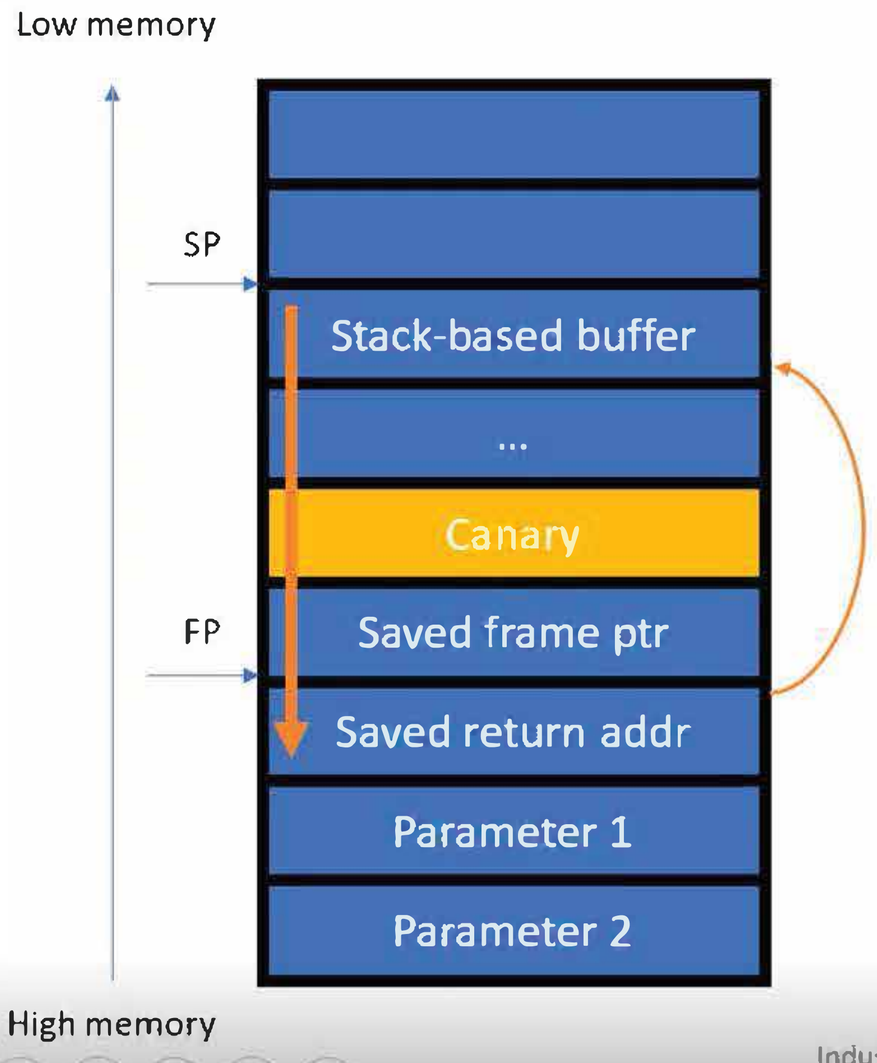
strn* takes length of souce to be copied, while strl* takes the maximum size of the result that can be created. Truncation identified from return value.

Stack-based Buffer Overflows



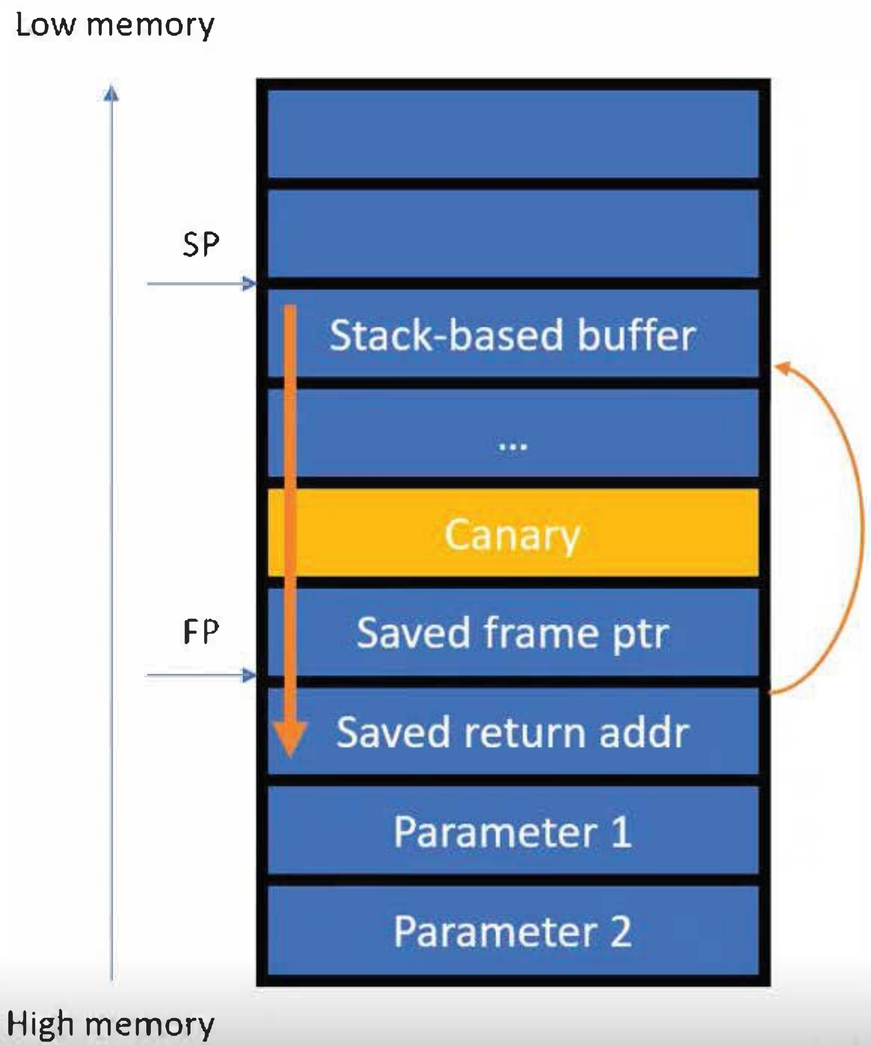
- Parameters are passed to functions using registers
- Most recent saved return address is written to link register (not the stack)
- The previously saved return address is written to stack (from link register)
- A malicious attacker can overflow the saved return address to execute malicious code (after 2 stack frames are deallocated)

Stack Canaries



- Miners would bring canaries to the mines. Dying canaries were an indicator of noxious gases in the mine.
- Canary is placed before the return address on the stack and initialized with a known value.
- Before jumping to return address, verify if canary value is unchanged, else abort!

Issues with Stack Canaries



- An attacker can ensure that the canary value is overwritten with original value
  - Can fork a child process (which inherits canary) and try brute force to find it
  - Calling `execve` in the child process solves the issue
- Information leak may disclose the value of the canary
- Local variable function ptrs live on stack, can be overwritten without affecting canary

Mitigation strategies

Data Execution Prevention (W^E)

- Memory pages should be marked as readable and writable (data) or readable and execute (code)
- Only one of writable or executable permissions should be assigned to memory locations
- Pointing to malicious code from data stack causes memory access violation, and segmentation fault

Return to libc!!

- Instead of pointing to malicious code, the attacker can modify the return address to point to a standard C library with arguments to do the attacker's bidding (change stack permissions, copy attacker's code to unprotected region)
- **Address Space Layout Randomization:** Randomize the address space; location of text, stack, heap, DLL, dynamic linker

Avoiding integer under/overflow

- Occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits – either higher than the maximum or lower than the minimum representable value.
- `UINT8 a,b,c;`
- Check each arithmic operation with its complementary operation
- ```
  if (a > UINT8_MAX - b)
    goto error;
  ```
- ```
  if (a > UINT8_MAX / b)
    goto error;
  ```

True or False: There is no need to check for integer under / overflow while using the increment or decrement operators

Type conversion rules

C language allows both implicit and explicit type conversion of a variable via assignment, comparisons, formal parameter binding on function calls and direct type casting

direct type conversion

assignment type conversion

comparison type conversion

function type conversion

```
signed int i = (int) 4.5;
signed int i = 1;
unsigned int j = 2;
float f = 3;
if (i < j){…}

void function(int arg1, float arg2);
function(5.3, 2);
```

explicit

implicit

Values are cast from one type to another via truncation, zero-extension and sign-extension.

Fun Facts! No #1

A char is 1 byte, but how many bits are in a byte?

8? Not necessarily!

No. of bits in a byte is not defined by C standard. On some systems it can be 16 bits, not 8.

Fun Facts! No #2

Is a short 2 bytes?

According to C standard, the size of a short must be at least 2 bytes, must be equal to or smaller than the size of an int. If you want a data type exactly 16 bits wide, use int16_t or uint16_t.

Fun Facts! No #3

What is the width of an int?

According to C standard, the size of an int should be equal to or greater than the size of a short, and equal to or smaller than the size of a long.

Fun Facts! No #4

How large is a size_t in C?

A size_t must be large enough to represent all addresses on a system. Therefore it must be the same size as a pointer. The size_t type is unsigned.

Fun Facts! No #5

What happens if a small negative number is passed to malloc?

Malloc takes in an argument of type size_t.

size_t is of unsigned type.

So a small negative number will be interpreted as a large positive number. Allocator may attempt to allocate an extremely large chunk of memory, which may fail.

Type conversion vulnerability

```
signed int i = -1;
unsigned int j = 1;
if (i < j) {printf("test!");}
```

- Due to implicit type conversion during the comparison, i will be treated as an unsigned integer, which is a large positive number. So, i would be greater than j.
- Type conversion vulnerability leads to unintuitive behaviour.

Perilous Potholes called Pointers

- C language does not use smart pointers, there is no tracking of pointer aliasing or assurance that pointers are valid
- This may lead to the following issues:
  - NULL pointer dereference
  - Stale pointers
    - Double-free: freeing the same pointer twice or more
    - Use-after-free: Using a pointer after freeing it
  - Errors with pointer arithmetic

Perilous Potholes called Pointers

NULL pointer dereference

If NULL pointer deference is done of a pointer representing a function pointer to a kernel module, a malicious user space process can map the zero page to an address of another function. When this function is called by kernel module, it could raise the privilege of the malicious user process

Most modern OSes reserve the page starting at address 0 and mark it as guard page. R/W/X of this page causes page fault and crashes the program!

Perilous Potholes called Pointers

Double-free vulnerability

```
char *A, *B;
A = malloc(size_of_chunk);
free(A);
B = malloc(size_of_chunk);
free(A);
```

They will be assigned the same region in memory, but we end up freeing the newly allocated chunk given to B. Usually this is distributed over many lines of code, functions and files. Very careful code-review needed.

Perilous Potholes called Pointers

Use-after-free vulnerability

```
char *ptr = malloc(100*sizeof(char));
for (int i = 0; i < 100; i++)
   ptr[i] = 'A';
free(ptr);
printf("%s",ptr);
```

If the attacker inserts values in memory before the print, the print can display wrong values. If the pointer here had been a function pointer, the attacker could execute malicious code by leveraging this.

Very careful code-review needed for mitigation.

External input validation

- Create whitelist of inputs that are allowed - small, well-defined list
- Canonicalization – convert all inputs of the same functional effect to one format before applying input validation rules (converting to lower case etc.)
- External input: comes across a security boundary
  - Through keyboard, mic or other input device
  - Through wireless device, Bluetooth, Wifi, network
  - From one processor to another (apps to modem, HLOS or TrustZone)
  - Data read from persistent storage such as file system

Thank You