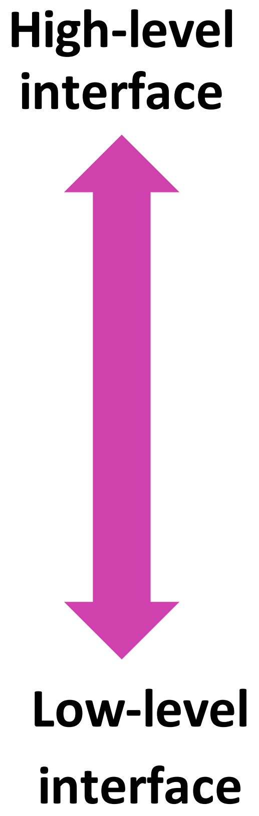
# Computer architecture

Arm and Arm architecture primer

# Instruction Set Architecture



- Execute high-level languages directly.
- Execute complex instructions (CISC).
- Tailor instruction set for pipelined and high-performance implementations. Expose the instruction pipeline to the compiler so it can optimize code and help simplify the hardware (RISC).
  **We will explore this approach.**
- Provide additional explicit information about the dependencies between instructions. E.g., VLIW or
- Specify individual data transfers, e.g., Transport Triggered Architectures (TTA)

## Instruction Set Architecture

- The best instruction set is the one that yields the “best” implementation.
- Changing the instruction set is difficult and happens infrequently .
- The factors that influence instruction set design do change over time, e.g., applications, programming languages, compiler technology, transistor budgets, and the underlying fabrication technology.
- We need to take care not to include “features” that will be regretted later.

## The RISC Approach

- The RISC approach aims to ensure that we **make the common-case fast** by carefully selecting the most useful instructions and addressing modes, etc.
- Instructions are designed to make good use of the register file.
- A RISC ISA is designed to ensure a simple high-performance implementation is possible.

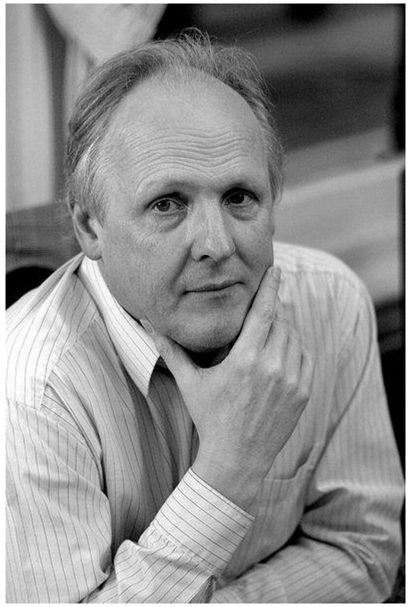
## Instruction Set Architecture

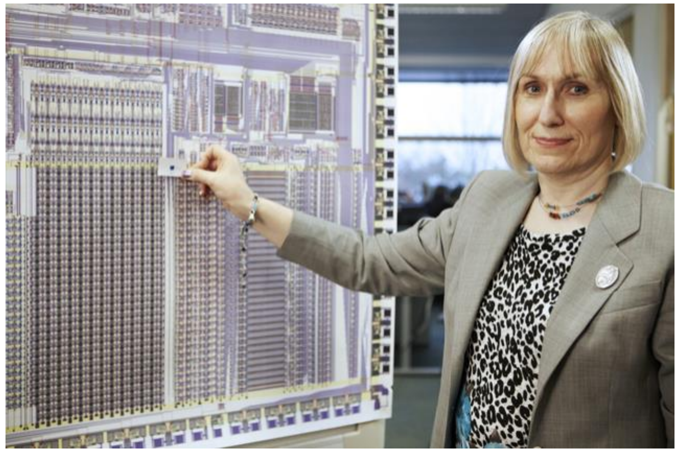
Common features of RISC instruction sets:

- Fixed length instruction encodings (or a small number of easily decoded formats)
- Each instruction follows similar steps when being executed.
- Access to data memory is restricted to special load/store instructions
  (a so-called **load/store architecture**).

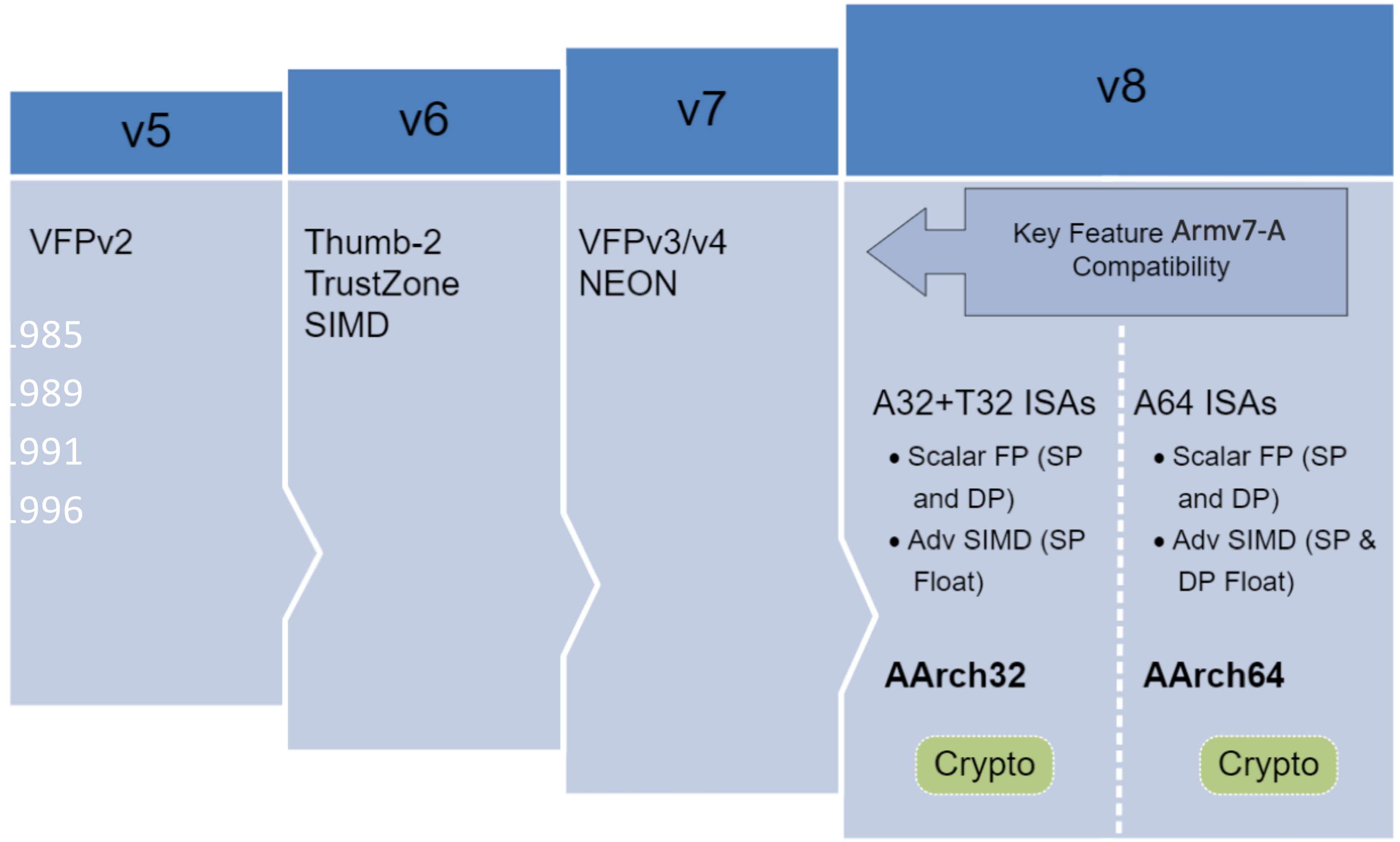
# Arm1: The First Arm Processor (1985)

- Arm: **A**dvanced **R**ISC **M**achine (Arm)
- The first Arm processor was designed by Sophie Wilson and Prof. Steve Furber. It was inspired by early research papers from Berkeley and Stanford on RISC.
- Arm1
  - 25,000 transistors
  - 3-stage pipeline
  - 8 MHz clock
  - No on-chip cache


Prof. Steve Furber


Sophie Wilson

# Case Study: The Armv8 Architecture



Announced 2011

ALGEBRA

## AArch64 – How Does It Differ from Older Arm ISAs?

- Conditional execution mostly dropped
- No free shifts in arithmetic instructions
- Program counter not a part of integer register set
- No load/store multiple instructions
- Adopts a more regular instruction encoding

## A64 Instructions

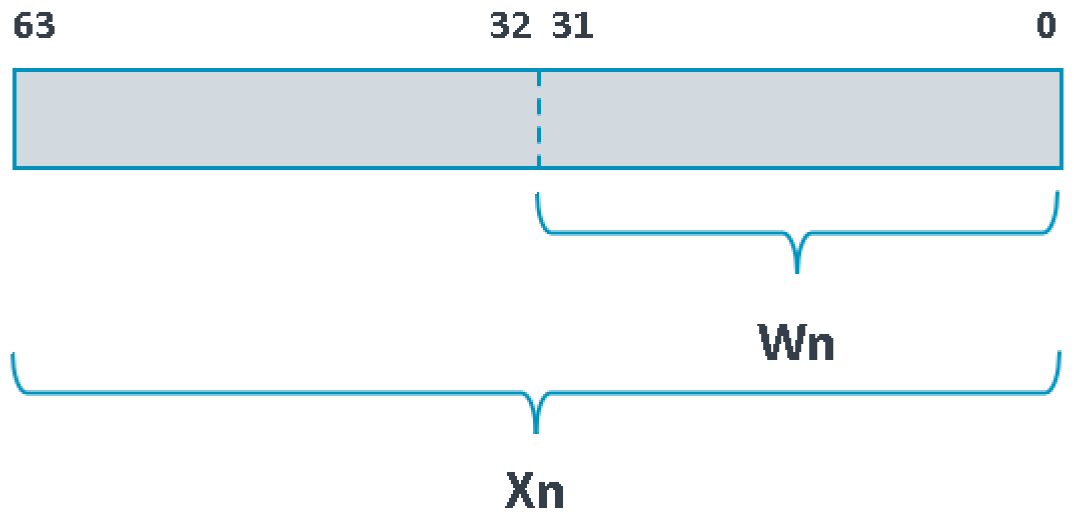
- 64-bit pointers and registers
- Fixed-length 32-bit instructions
- Load/store architecture
- Simple addressing modes
- 32 x 64-bit general-purpose registers (including the R31 the zero/stack register)
- The PC cannot be specified as the destination of a data processing instruction or load instruction.

# AArch64 - Registers

In the AArch64 Execution state, each register (X0-X30) is 64-bits wide. The increased width (vs. 32-bit) helps to reduce register pressure in most applications.

Each 64-bit general-purpose register (X0 - X30) also has a 32-bit form (W0 - W30).

Zero register – X31



## AArch64 – Load/Store Instructions

**LDR** – load data from an address into a register.

**STR** – store data from a register to an address.

```
LDR X0, <addr>   ; load from <addr> into X0
STR X0, <addr>   ; store contents of X0 to <addr>
```

In these cases, X0 is a 64-bit register, so 64-bits will be loaded or stored from/to memory.

## AArch64 – Addressing Modes

**Base register only**: Address to load/store from is a 64-bit base register.

```
    LDR X0, [X1]             ; load from address held in X1
    STR X0, [X1]             ; store to address held in X1
```

**Base plus offset**: We can add an immediate or register offset (**register indexed**).

```
    LDR X0, [X1, #8]         ; load from address [X1 + 8 bytes]
    LDR X0, [X1, #-8]        ; load from address [X1 – 8 bytes]
    LDR X0, [X1, X2]         ; load from address [X1 + X2]
    LDR X0, [X1, X2, LSL #3] ; left-shift X2 three places
                                   before adding to X1
```

•

## AArch64 – Addressing Modes

**Pre-indexed**: source register changed before load

```
    LDR W0, [X1, #4]!    ; equivalent to:
                           ADD X1, X1, #4
                           LDR W0, [X1]
```

**Post-indexed:** source register changed after load

```
    LDR W0, [X1], #4     ; equivalent to:
                           LDR W0, [X1]
                           ADD X1, X1, #4
```

# AArch64 – Data Processing

- Values in registers can be processed using many different instructions:
  - Arithmetic, logic, data moves, bit field manipulations, shifts, conditional comparisons, etc.
- These instructions always operate between registers, or between a register and an immediate.

Example loop:

```
        MOV X0, #<loop count>
Loop:
        LDR W1, [X2]
        ADD W1, W1, W3
        STR W1, [X2], #4
        SUB X0, X0, #1
        CBNZ X0, loop
```

## AArch64 - Branching

`B <offset>`

PC relative branch (+/- 128MB)

`BL <offset>`

Similar to B, but also stores return address in LR (link register), likely a function call

`BR Xm`

Absolute branch to address stored in **Xm**

`BRL Xm`

Similar to BR, but also stores return address in LR

# AArch64 - Branching

`RET Xm` or simply `RET`

- Similar to BR, likely a function return
- Uses LR if register is omitted

**Subroutine calls:**

The Link Register (LR) stores the return address when a subroutine call is made. This is then used at the end of our subroutine to return back to the instruction following our subroutine call.

•

## AArch64 – Conditional Execution

The A64 instruction set does not include the concept of widespread predicated or conditional execution (as earlier Arm ISAs did).

The NZCV register holds copies of the N, Z, C, and V condition flags.

A small set of conditional data processing instructions are provided that use the condition flags as an additional input. Only the conditional branch is conditionally executed.

- Conditional branch
- Add/subtract with carry
- Conditional select with increment, negate, or invert
- Conditional compare (set the condition flags)

## AArch64 – Conditional Branches

**B.cond**

Branch to label if condition code evaluates to true, e.g.,

```
CMP X0, #5
B.EQ label
```

`CBZ/CBNZ` – branch to label if operand register is zero (CBZ) or not equal to zero (CBNZ)

`TBZ/TBNZ` – branch to label if specific bit in operand register is set (TBZ) or clear (TBNZ)

```
TBZ W0, #20, label ; branch if
(W0[20]==#0b0)
```

# AArch64- Conditional Operations

`CSEL` – select between two registers based on a condition

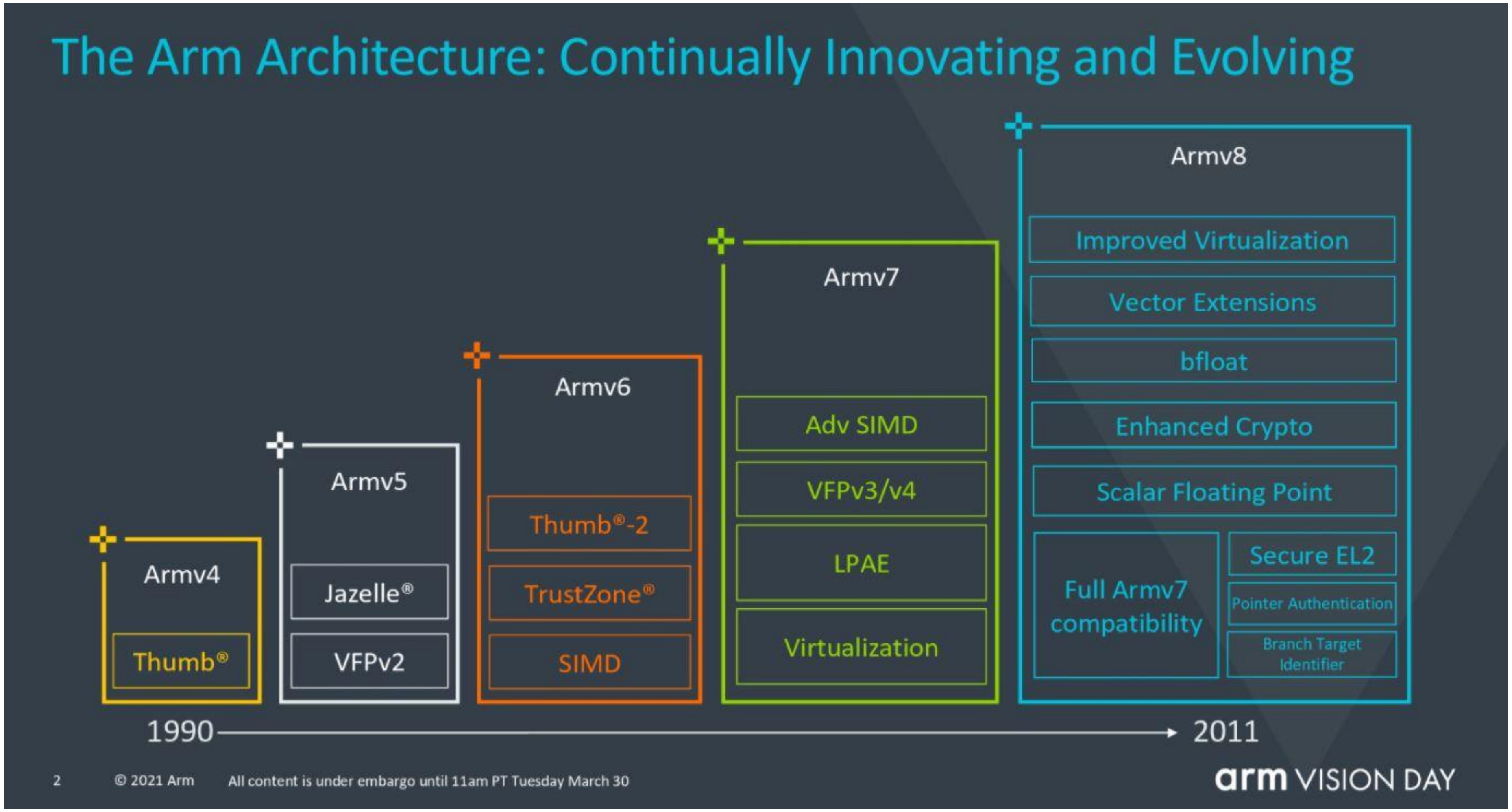
```
CSEL X7, X2, X0, EQ ; if (cond==true) X7=X2,
else X7=X0
```

There are also variants of this that cause the second source register to be incremented, inverted, or negated.

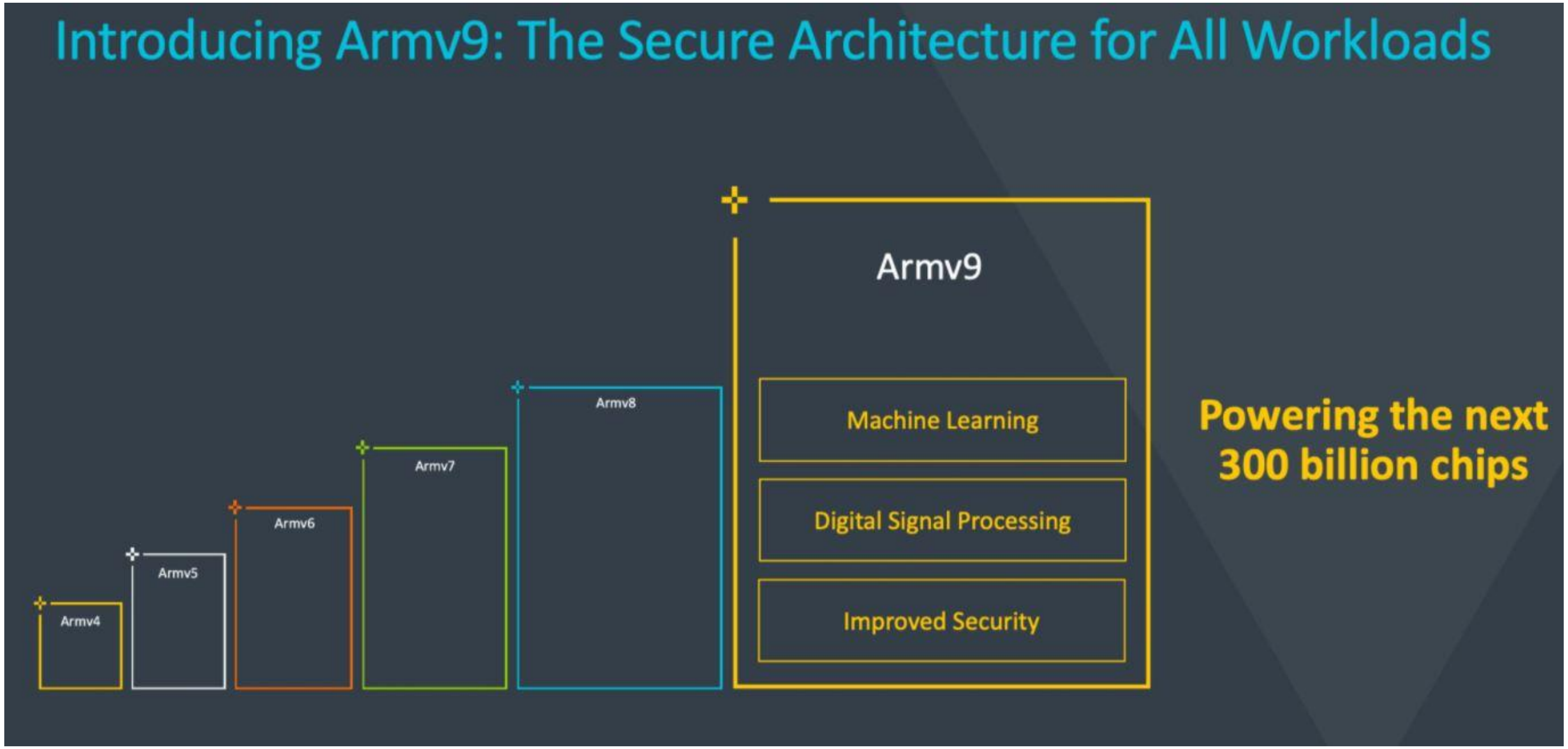
## Armv9 architecture

- Launched in 2021
- Focus on :
  - security (Arm CCA, Confidential Compute Architecture)
  - AI, SVE2 (Scalable Vector Extension)
  - Total compute design
- In a way, you could say that Armv8 was a bit desktop-computer oriented (ARM designs for desktop computers and servers prove that point)
- In a way, you could say that Armv9 is more leaning towards supercomputing, architecture-wise – for SC, for HPC
- Reason is very simple – design demands and needs, as we discussed numerous times before
- We're going to further demonstrate this as we move towards parallelism and memory as when we cover those topics it's going to become even more obvious why

# Part of Arm history, visualized, part I



## Part of Arm history, visualized, part II


Introducing Armv9: The Secure Architecture for All Workloads
Armv4
Armv5
Armv6
Armv7
Armv8
Armv9
Machine Learning
Digital Signal Processing
Improved Security
Powering the next 300 billion chips

## Thank you for your attention!