ANGSTROMERS
Navigating Silicon Waves

RISC-V: Research & Innovation

Day 1

Session 1

Dr. Cyril Prasanna Raj P.
Director – Angstromers Engg. Services
Professor – Dept. of ECE
CIT, Bangalore

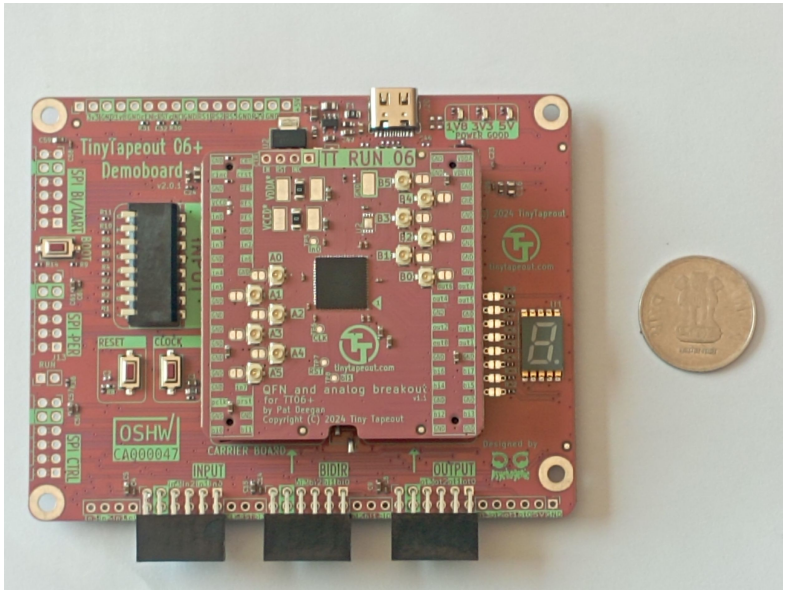
cyrilyahoo@gmail.com
cyril@cambridge.edu.in

Answer these three questions

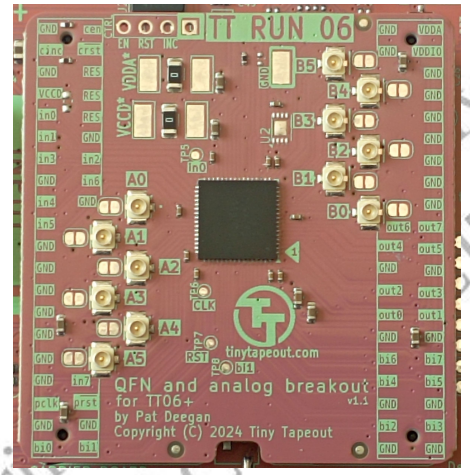
1. Integrated circuit is black (gray) in color ?
   A. Yes or No
2. Integrated circuit has pins on two sides (or four sides) ?
   A. Yes or No
3. The component in the picture is called integrated circuit ?
   A. Yes or No

Integrated circuit

CITs First IC Developed in 4 months

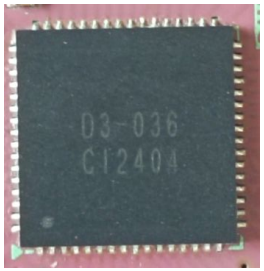


Display Controller Test Platform

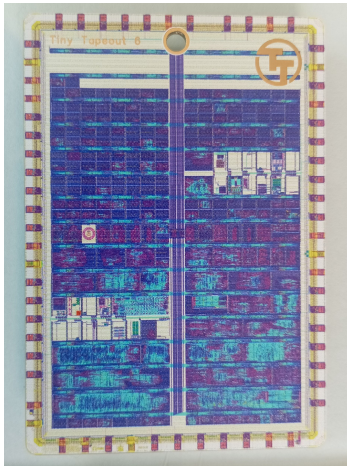


Daughter card with Display Controller Mounted

Out First Chip D3-036/ CI2404



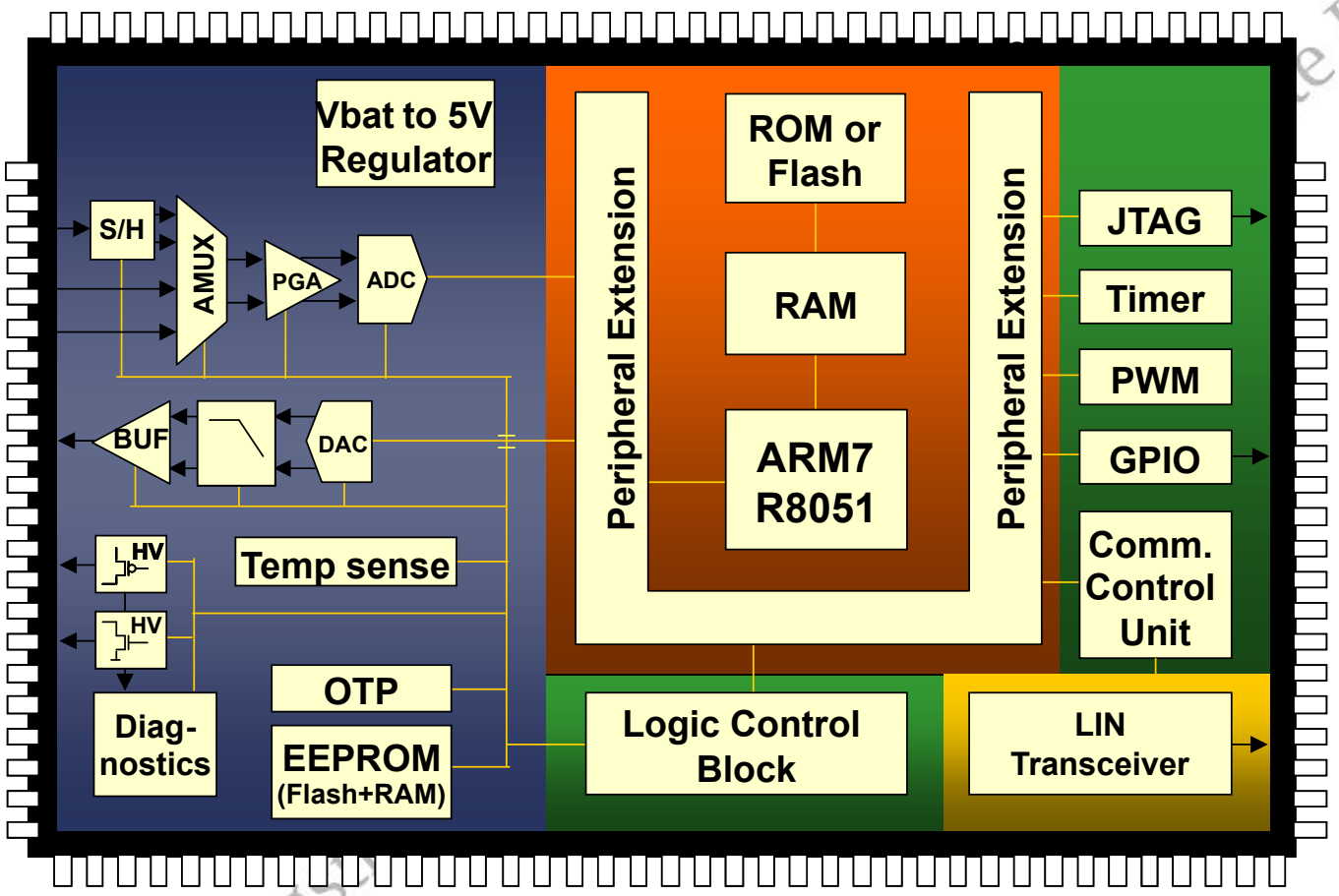
Display Controller IC



GDSII of IC

CCCIR team has designed and tested the display controller IC using 130nm CMOS Skywater Library under Tiny Chip Program
We have used 350 standard cells and area of 100 x 160 um with 50 MHz clock speed.

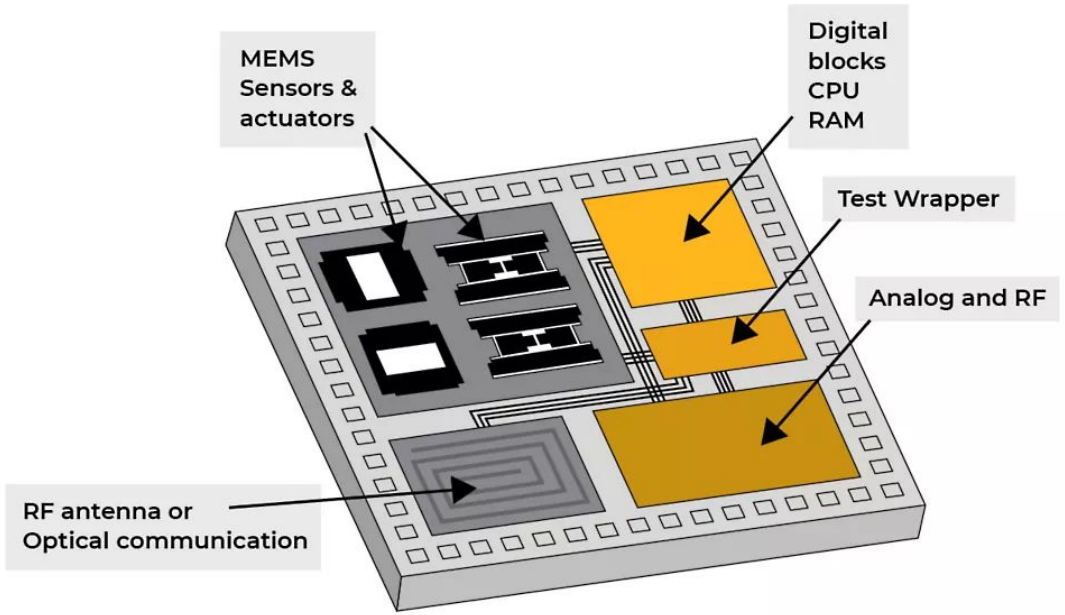
Integrated Circuit – Internal Architecture

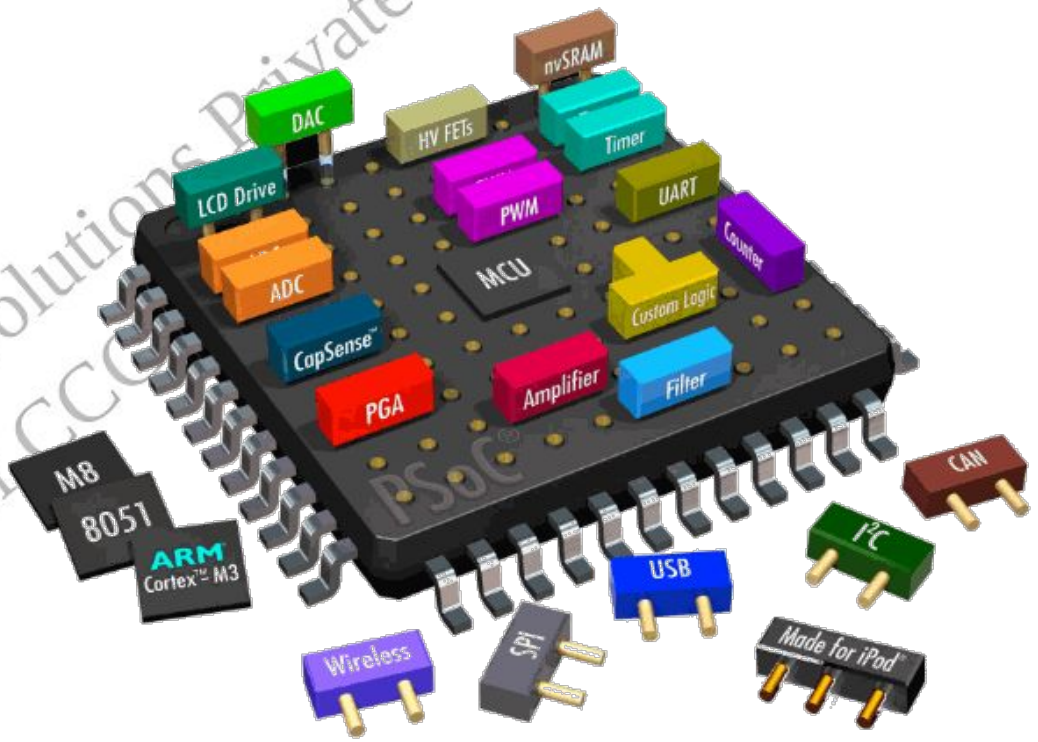


Primary building blocks of SOC are:

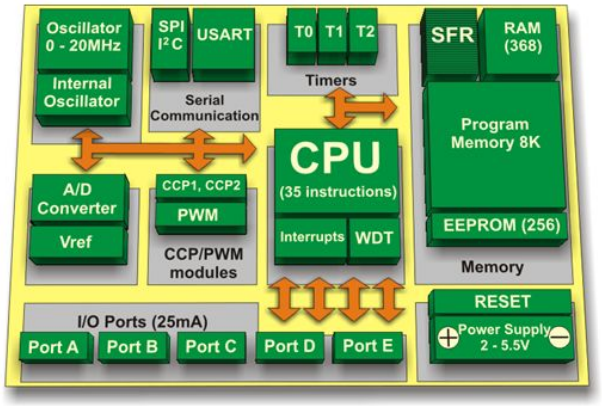
1. Digital
2. Analog
3. Mixed signal

System on Chip

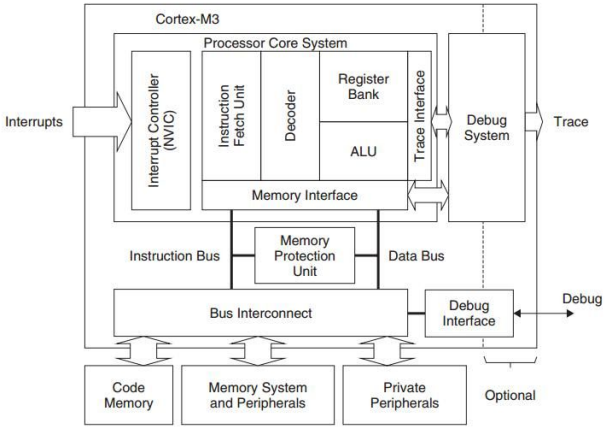




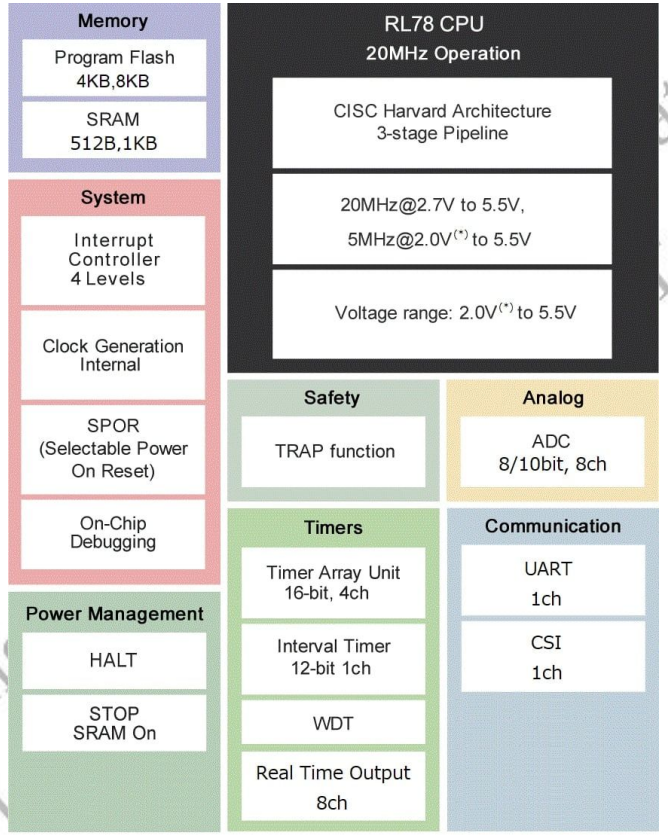
Microcontroller Internal Architecture



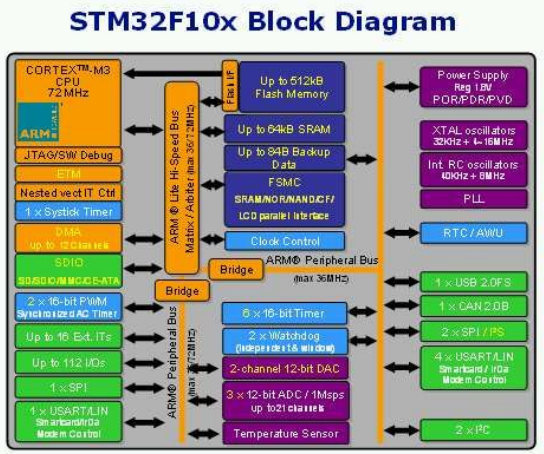
PIC

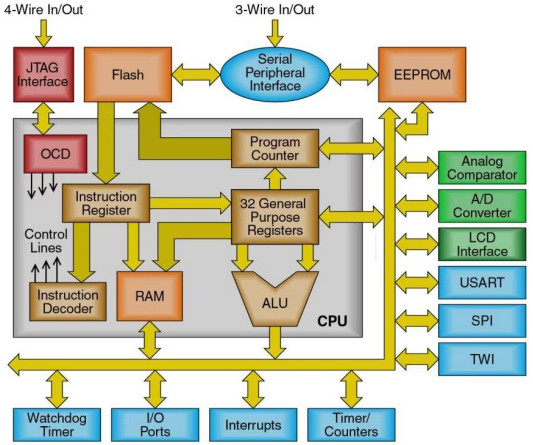


ARM



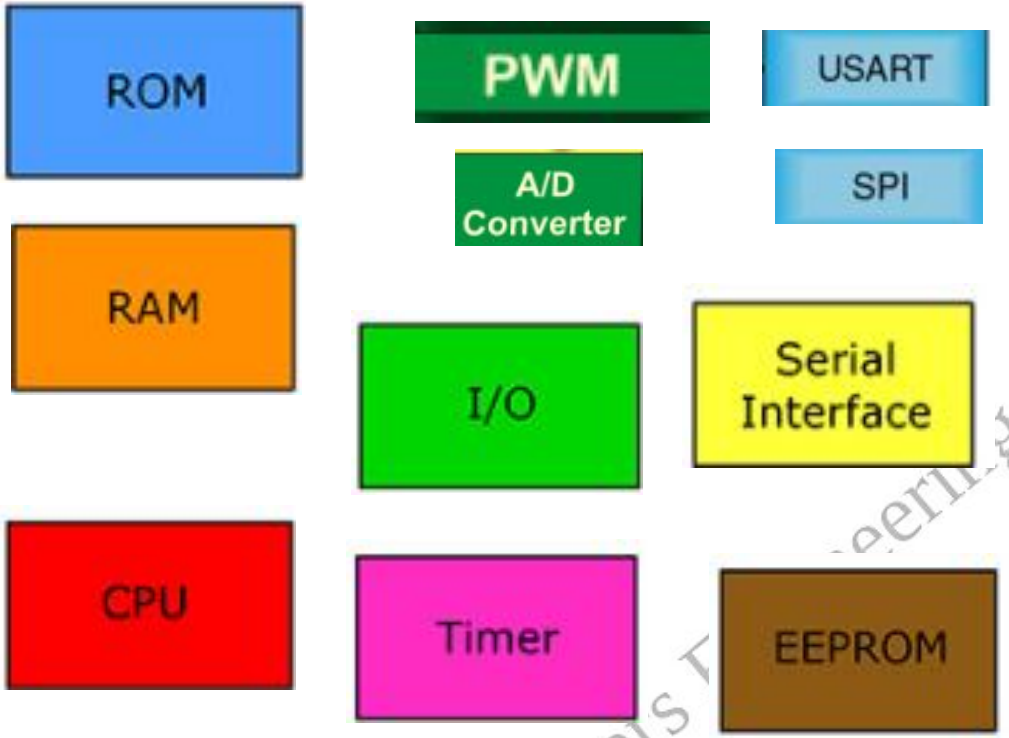
RL78/G1M - 8-bit General-purpose Microcontroller

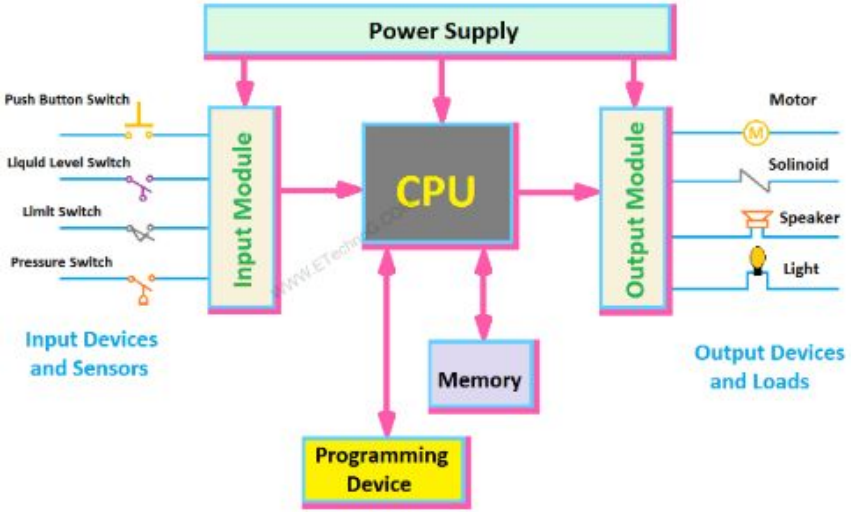




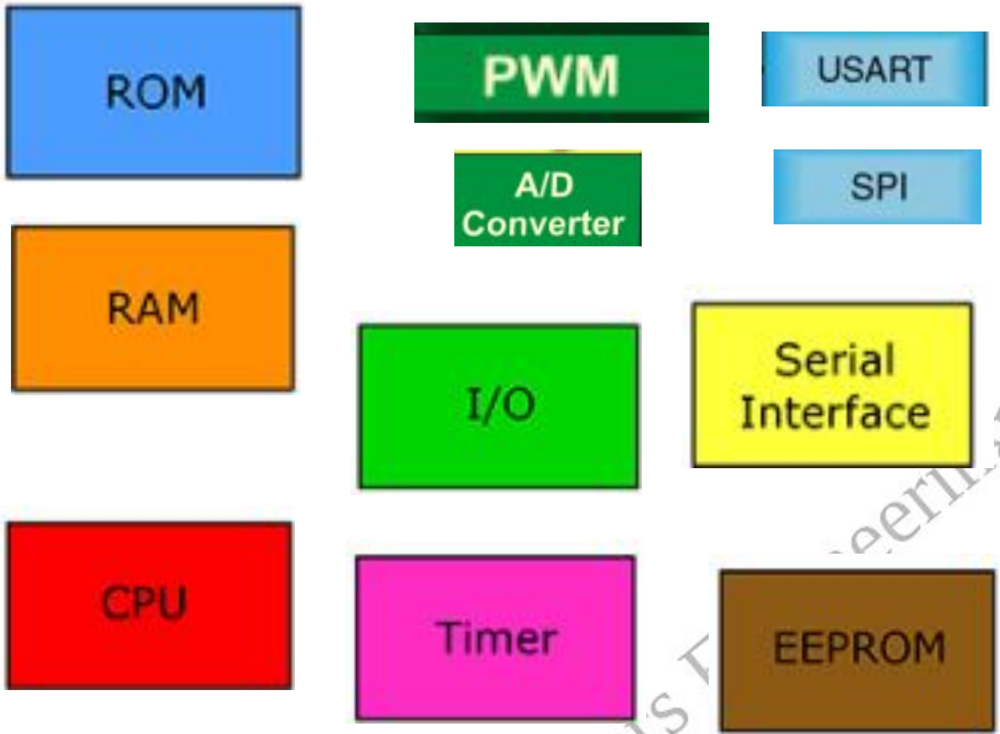
AVR

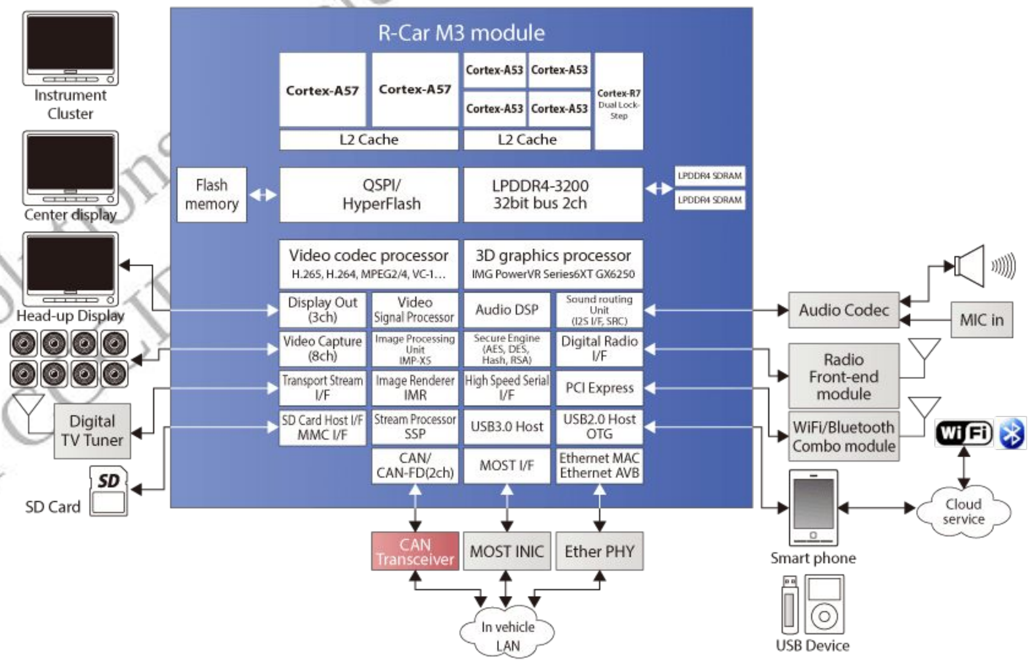
Exercise 1 – Develop a block diagram for microcontroller for PLC





Exercise 2 – Develop a block diagram for microcontroller for automotive





Limitations of existing controllers and why RISCV

- Why RISCV?
  - Open-Source & Royalty-Free
  - Customization & Specialization
  - Geopolitical & Supply Chain Independence
  - Efficiency & Performance for AI & Edge Computing

Processor Licensing (Upfront Fee)

- **Cost Range:Cortex-M (MCUs & IoT) → $1M – $10M**
- **Cortex-A (Mobile, Embedded, AI, SoCs) → $10M – $50M**
- **Neoverse (Data Center, HPC, AI) → $50M+**

Royalty Fees (Per Chip Sold)

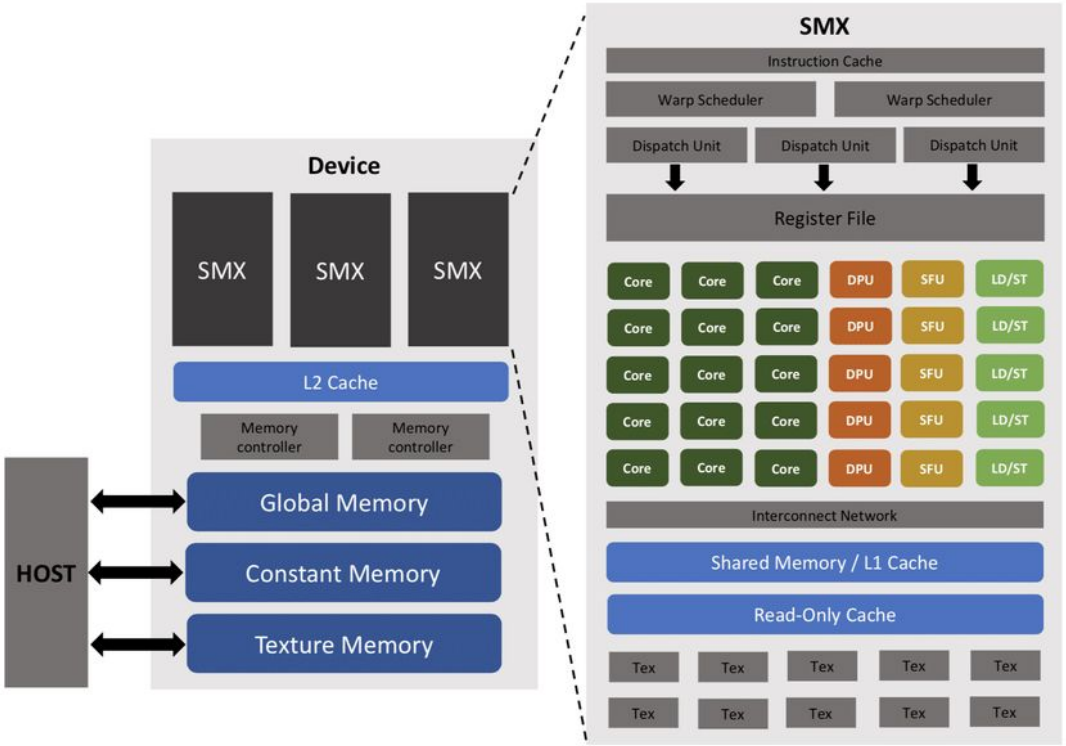
- Usually **1% – 2%** of the chip price
- If a company sells **100M Cortex-M chips at $2 each**, and ARM takes **1.5% royalty**, that's **$3M in royalties**.

Architecture License (Custom Cores)

- **Apple, Qualcomm, NVIDIA, Amazon, Samsung license the full ARM instruction set** to build custom cores (like Apple M-series, Qualcomm Kryo).
- **Costs $100M+** with ongoing royalties

Feature	ARM	x86	RISC-V
License Fee	$1M – $100M+	❌ Not Available	✅ Free
Royalties	1% – 2% per chip	❌ Not Applicable	✅ None
Customization	❌ Limited (unless paying $100M+)	❌ Only Intel & AMD	✅ Fully customizable
Who Controls It?	ARM (SoftBank)	Intel & AMD	Open-source (anyone can use)
Cost for Startups	💰 Expensive	💰 Very expensive	🆓 Free

GPU Internal Architecture



Grace-Hopper Superchip (GH200)

NVIDIA Core Architectures (Generations)

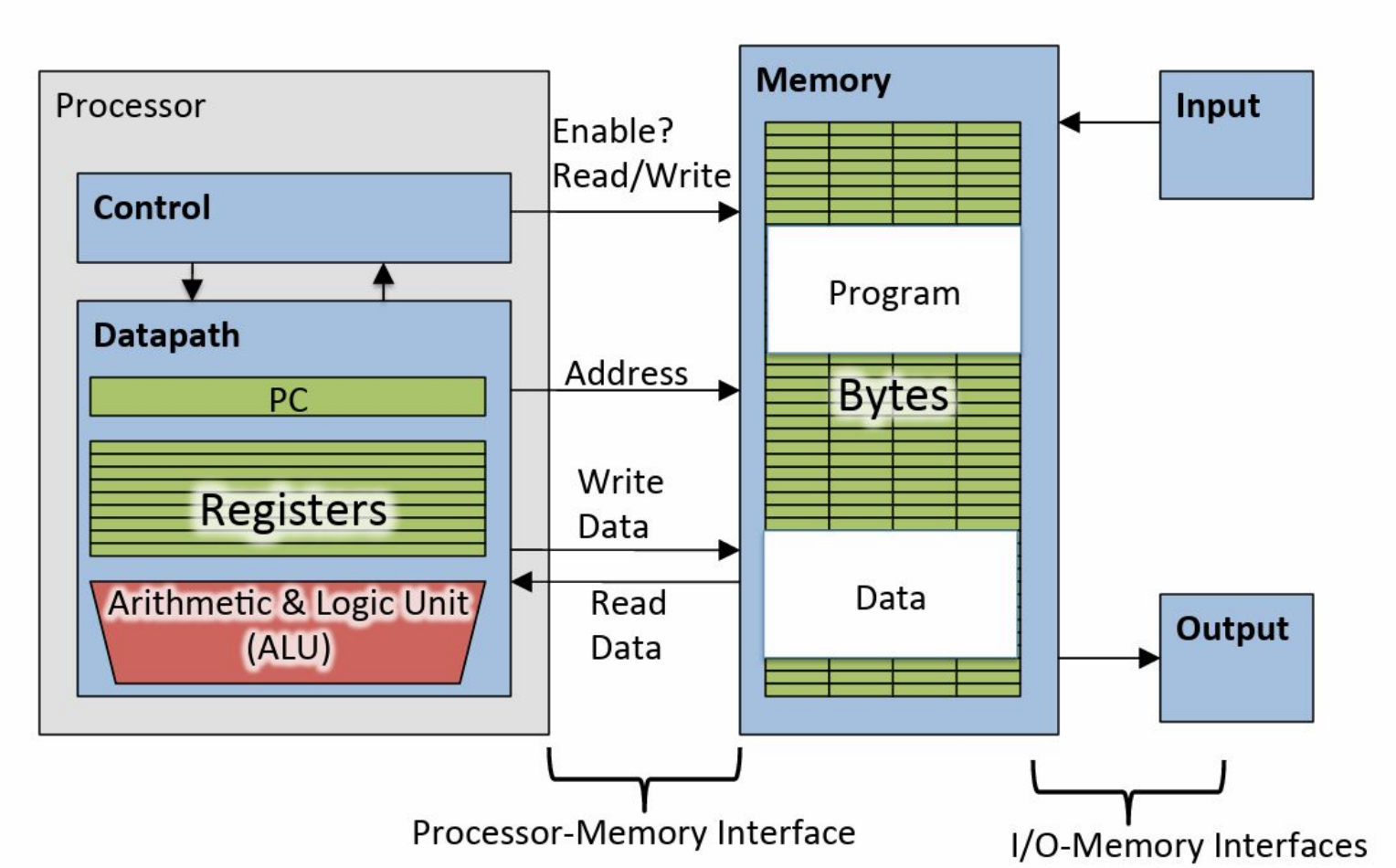
- Tesla (2006-2008) – First CUDA-capable architecture.
- Ampere (2020-2022) – Improved Tensor and RT Core performance.
- Ada Lovelace (2022-present) – Latest gaming GPUs (RTX 40 series), enhanced AI and ray tracing.
- Hopper (2022-present, for AI/Datacenter) – Optimized for AI and data centers, used in H100 GPUs.

CUDA & Tensor cores are the main compute units in NVIDIA GPUs

NVIDIA Jetson series (Jetson Xavier, Jetson Orin) integrates ARM-based CPUs (Cortex-A series) with CUDA-capable GPUs for AI applications in robotics, edge computing, and autonomous machines.

- Combines an ARM-based NVIDIA Grace CPU with an NVIDIA Hopper GPU
  - Grace CPU is based on **72 ARM Neoverse V2 cores**
  - Hopper GPU provides **Tensor Cores for AI acceleration**

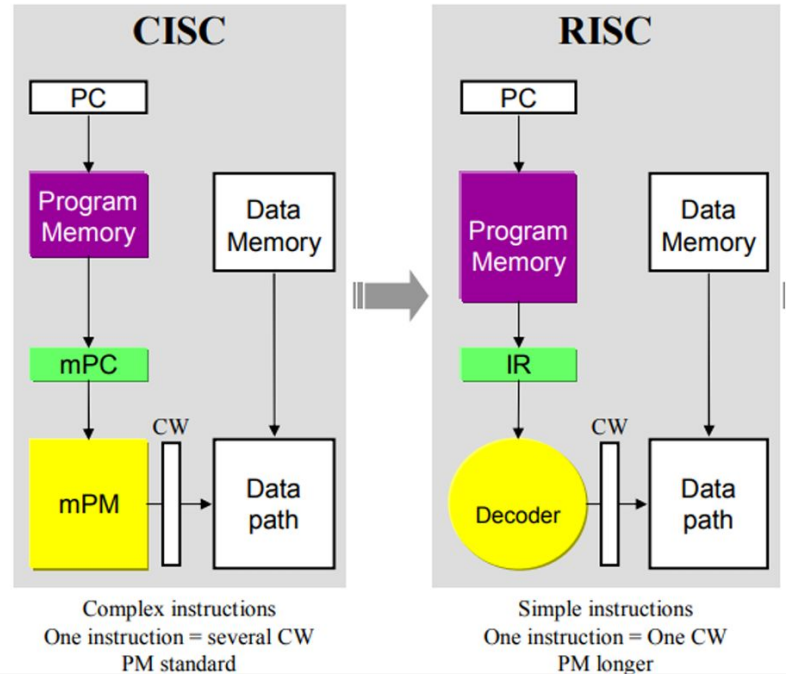
Components of a Computer



CPU Types

- Complex Instruction Set Computer (CISC) processors and Reduced Instruction Set Computer (RISC) processors

CISC	*RISC*
More than 300 instructions in instruction set	Less than 100 instructions in instruction set
Non-uniform sizes and formats of instructions	Fixed sizes and formats of instructions
Large number of transistors are used for instructions storage	Large number of transistors are used for more registers
Long cycle time per instruction	One cycle time per instruction
More addressing modes	Less addressing modes
Difficult to pipeline	Easy to pipeline
Emphasis on hardware	Emphasis on software



RISC-V Introduction

- Unlike proprietary processor architectures [(Intel, AMD) (ARM stands for Advanced RISC Machine, and it's a proprietary instruction set architecture (ISA) for central processing units (CPUs))]
  - RISC-V is an open-source instruction set architecture (ISA)
    - For development of custom processors targeting a variety of end applications
    - Originally developed at the University of California, Berkeley
    - 10 billion chips containing RISC-V cores had shipped by the end of 2022
- How Does RISC-V Work?
  - RISC-V is defined by member companies of RISC-V International, the global nonprofit organization behind the ISA.
  - Royalty-free RISC-V ISA features a small core set of instructions upon which all the design's software runs.
  - It allows designers to tailor the architecture for a variety of different end markets
  - They can optimize the power, performance, and area (PPA) for those applications.

Advantages of RISC-V

1. **Global community**: RISC-V is supported by a global community that collaborates on technical standards, driving innovation and fostering a diverse ecosystem. The global community has contributed publicly available IP to the creation of a shared set of extensions that may be used with the base ISA. None of the extensions are proprietary, they are all open and publicly available.

2. **Open collaboration**: Open collaboration enables companies and developers to work together, sharing ideas and contributing to the evolution of the ISA and related extensions.

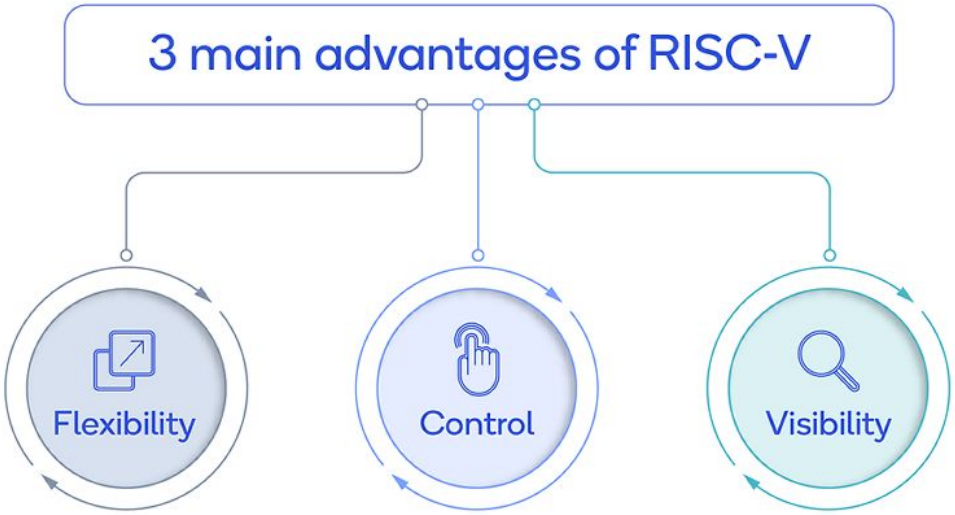
3. **RISC-V is a standard, not open source.** The RISC-V ISA and extensions go through a robust governance process to be ratified and frozen. The modular approach of a frozen standard ISA and extensions guard against fragmentation while enabling innovation.

4. **Enhanced cybersecurity**: Public review and scrutiny of the standard improves cybersecurity robustness and resilience.

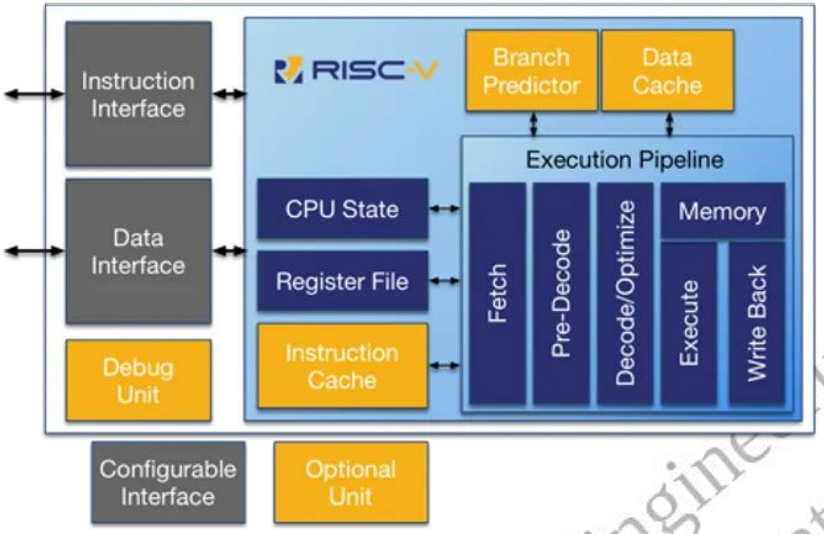
5. **Economic and technological value**: The open nature of RISC-V promotes competition and encourages investment in new technologies. RISC-V has innovation potential for anyone by enabling freedom of design, an aspect that is out of reach on any other architecture due to proprietary licensing models that restrict the ability to innovate.

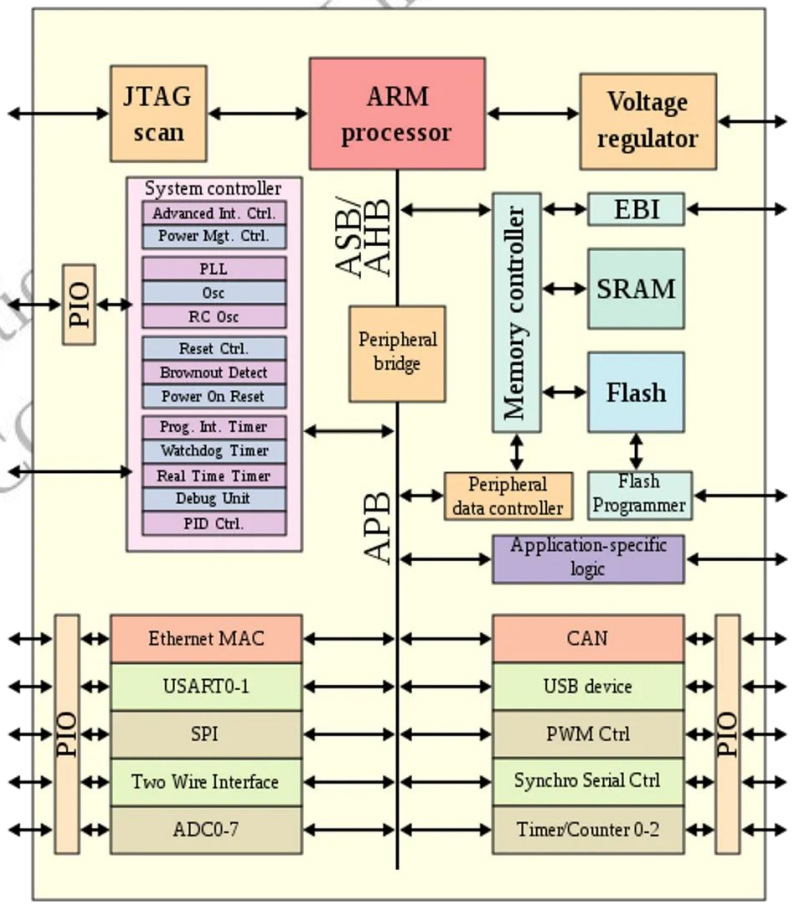
Advantages of RISC-V

- These advantages are quickly being recognized across industries, and businesses have been taking advantage of RISC-V cores for all kinds of applications including:
  - Artificial intelligence (AI) image sensors,
  - Security management,
  - AI computing,
  - Machine control systems for 5G networks, and
  - More sophisticated storage, graphics and machine learning applications.



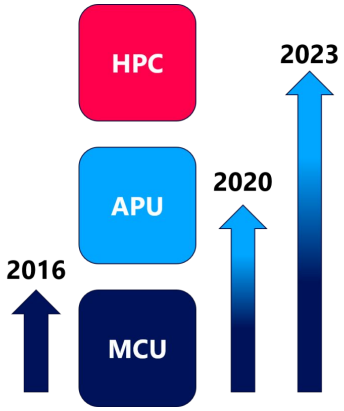
Comparing RISCV and ARM



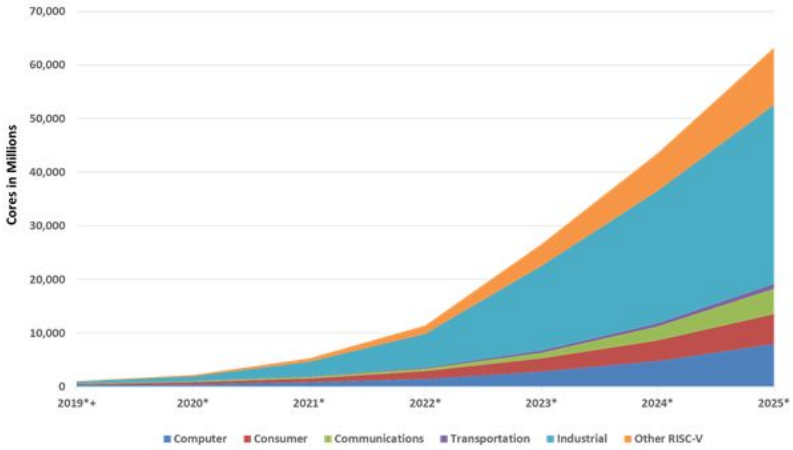


Is RISC-V The Future?

- Trend 1: RISC-V is shifting up in performance
  - more and more suppliers will create complex RISC-V cores for high-performance computing in the future
- Trend 2: Barriers between processor types are breaking down
  - RISC-V ISA, having a minimalist base integer instruction set and providing for custom extensions, it is an ideal starting point for creating special accelerators.



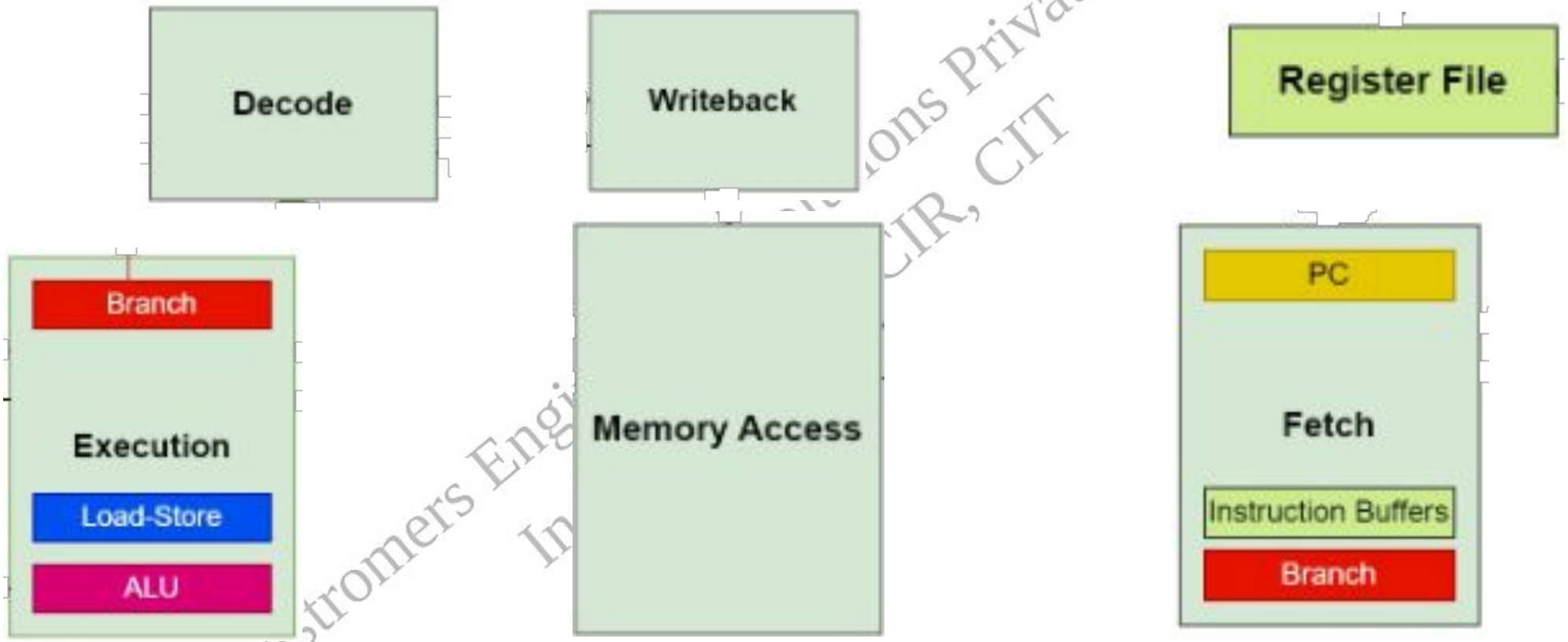
- Trend 3: Customers want to avoid a monopoly supplier
  - microprocessors have been dominated by the Intel/AMD X86 duopoly - 1980
  - Arm became the de-facto standard in the mobile phone processor market monopoly- 1990
  - “Arm fatigue” and disquiet with the monopolist position and vendor lock-in in key markets
  - acquisition by SoftBank, that neutrality was seriously eroded, and the possible Nvidia merger may mean that Arm, having singled one of their former customers out for strategic partnership, won’t eventually be considered a safe choice at all.



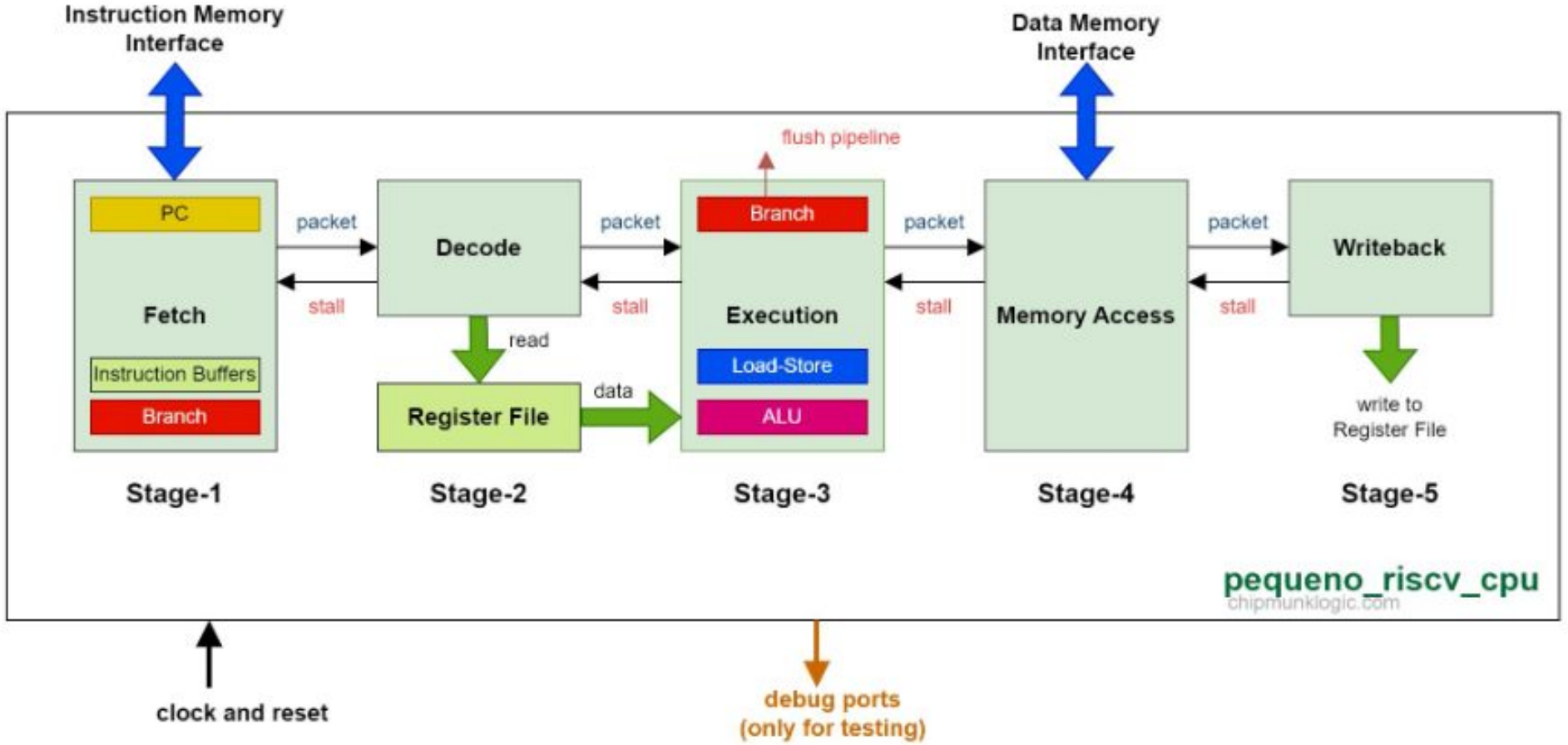
Source: Semico Research Corp.

Exercise 3

- Organize these blocks to form RISCV architecture and draw the directional map for data flow



Exercise 1 – Solution



RISCV Overview – Five stages of RISCV

Fetch

- Fetch 32-bit instruction from memory
- Increment PC = PC + 4

Decode

- Gather data from the instruction
- Read opcode; determine instruction type, field lengths
- Read in data from register file

Execute

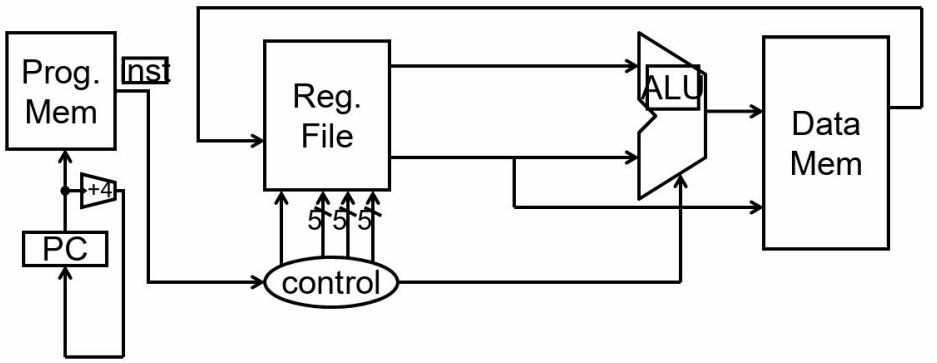
- Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)

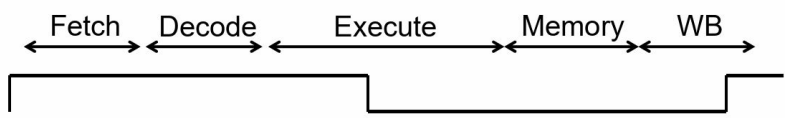
Memory

- Used by load and store instructions only
- Other instructions will skip this stage

Write Back

- Write to register file
- Update PC





A single cycle processor – this diagram is not 100% spatial

Types of Code

- Five types of code

Instruction	Example		Fetch	Decode	Execute	Memory	Write Back
Load	`R1 <- [1]`	address of 1 is loaded to R1	✓	✓	✓	✓	✓
Store	`[A] <- R1`	R1 is stored in address A	✓	✓	✓	✓	X
Branch	`BNE R1, R2, Loop`	R1 ≠ R2, not equal go to Loop	✓	✓	✓	X	X
Jump	`JMP Loop`	directly to the place stated in the code	✓	✓	X	X	X
R type*	`R1 <- R2 + R3`	R2 and R3 are added and stored in R1	✓	✓	✓	X	✓

* Uses pipeline

```
R1 <- [1]
R2 <- [2]
R3 <- [3]
```

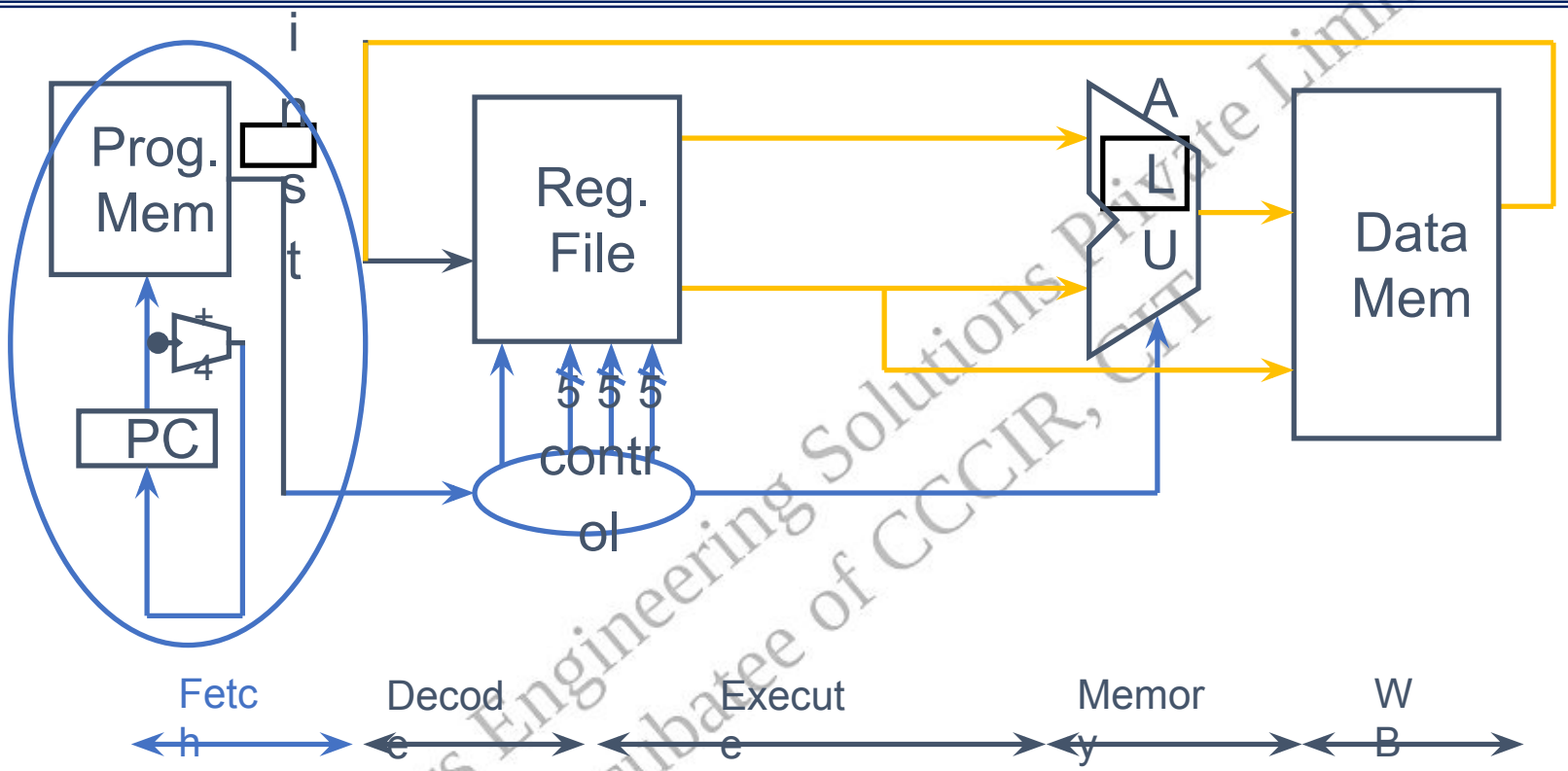
Code	Instruction line	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
R1 <- [1]	1	Fetch	Decode	Execute	Memory	Write		
R2 <- [2]	2		Fetch	Decode	Execute	Memory	Write	
R3 <- [3]	3			Fetch	Decode	Execute	Memory	Write

Five Stages of RISC-V Datapath

Basic CPU execution loop

1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

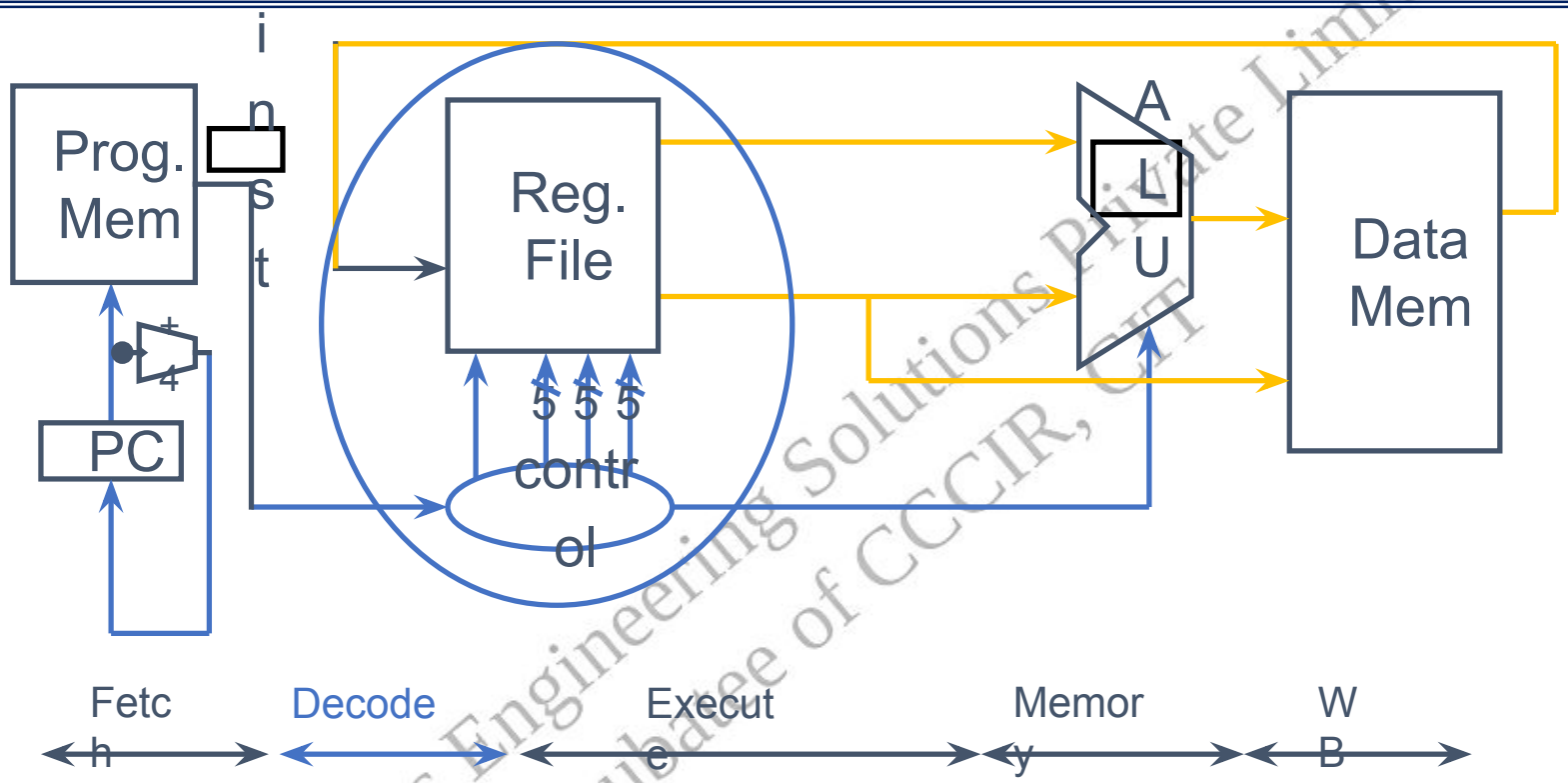
Stage 1: Instruction Fetch



Fetch 32-bit instruction from memory

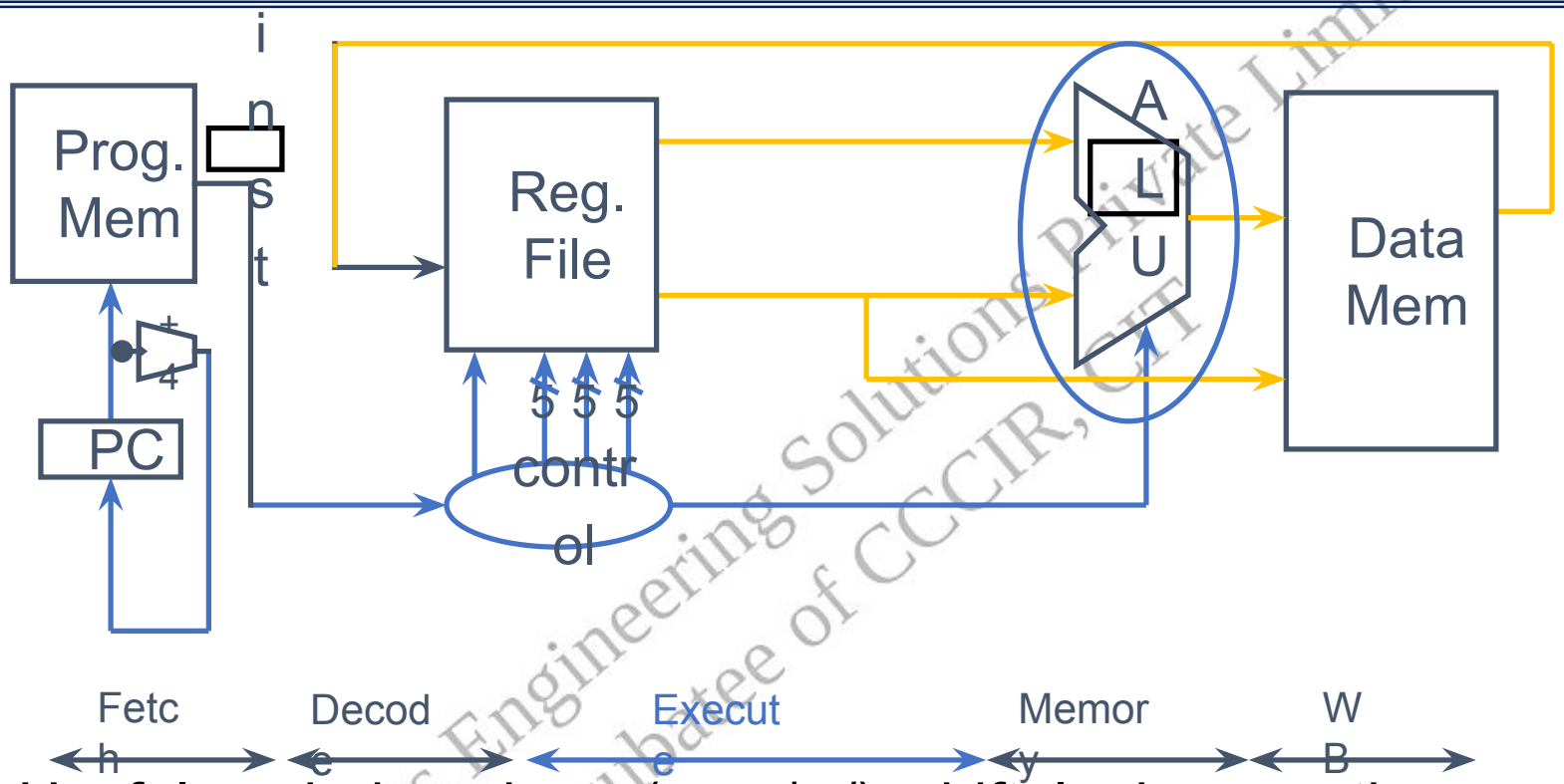
Increment PC = PC + 4

Stage 2: Instruction Decode



Gather data from the instruction
Read opcode; determine instruction type, field lengths
Read in data from register file
(0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

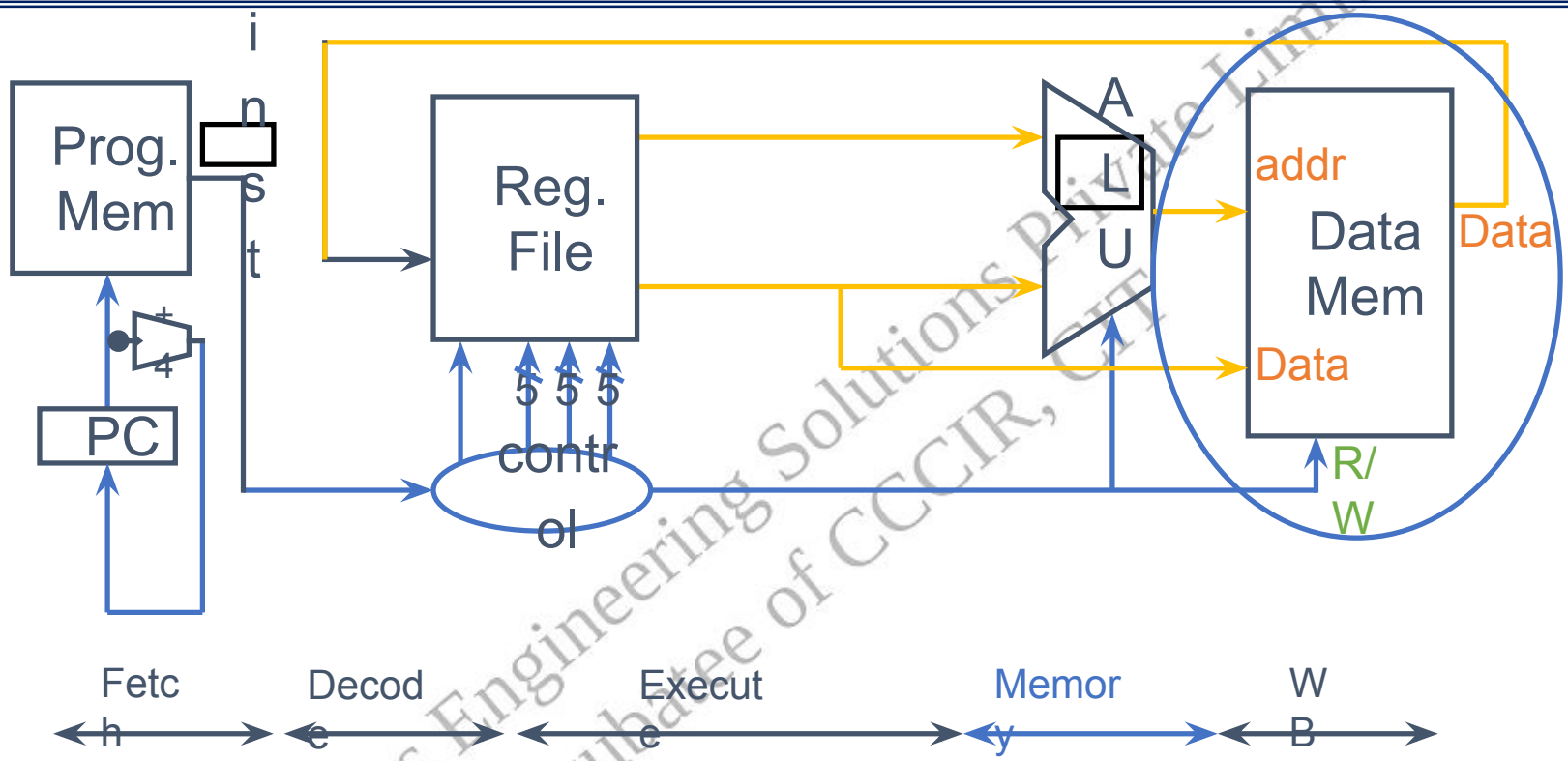
Stage 3: Execution (ALU)



Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)

Load/Store? lw x2, x3, 32 □ Compute address

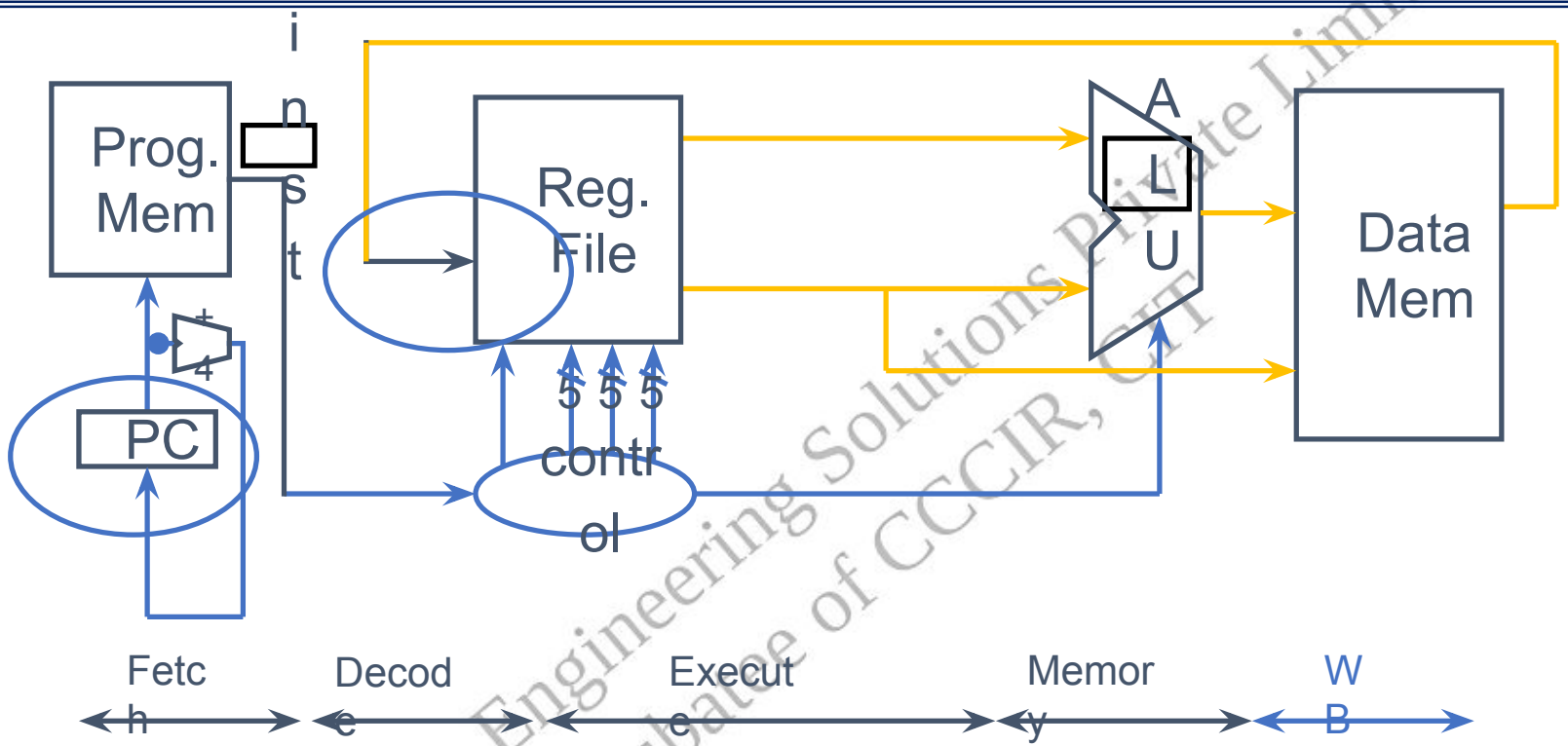
Stage 4: Memory Access



Used by load and store instructions only

Other instructions will skip this stage

Stage 5: Writeback



Write to register file

- For arithmetic ops, logic, shift, etc, load. What about stores?

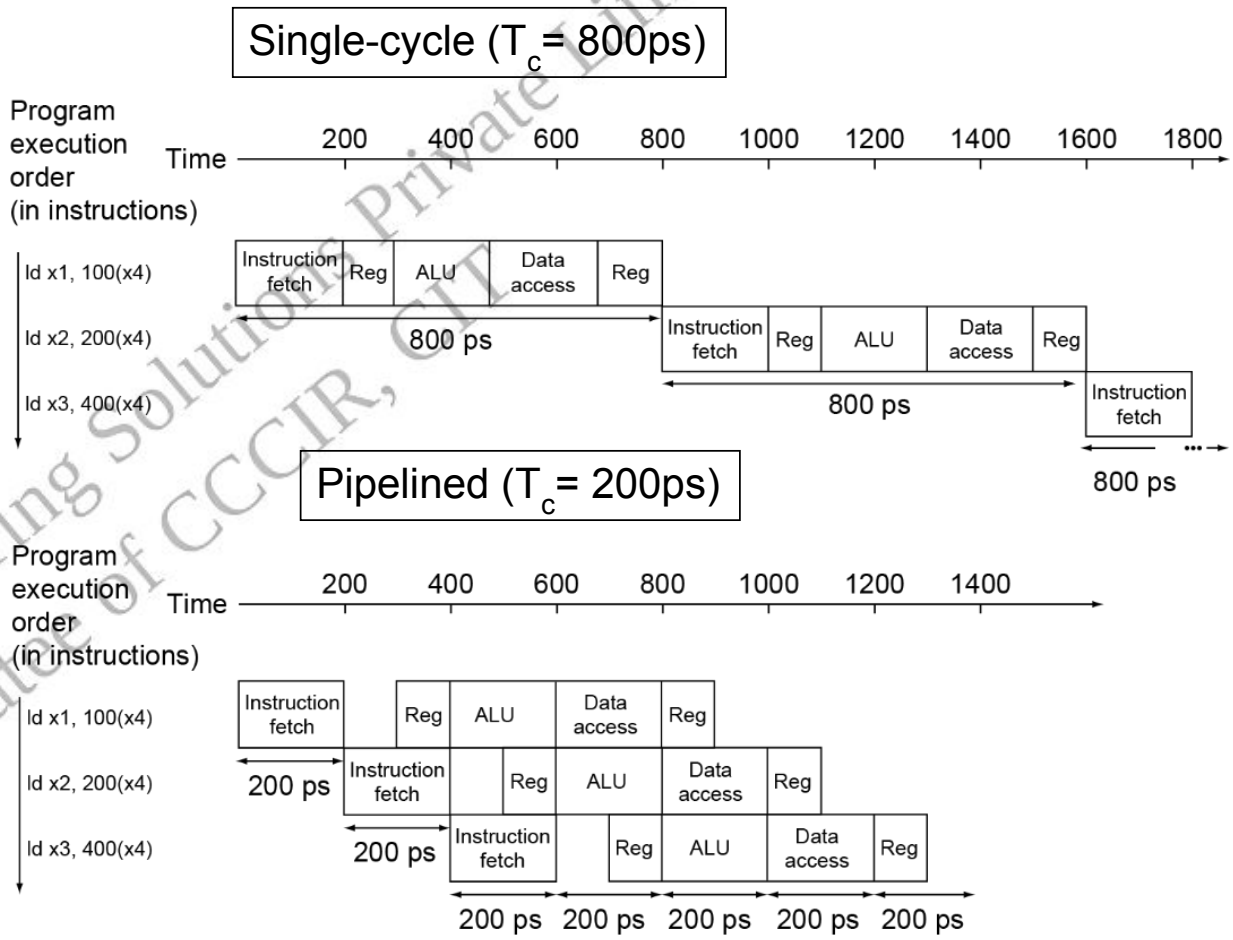
Update PC

- For branches, jumps

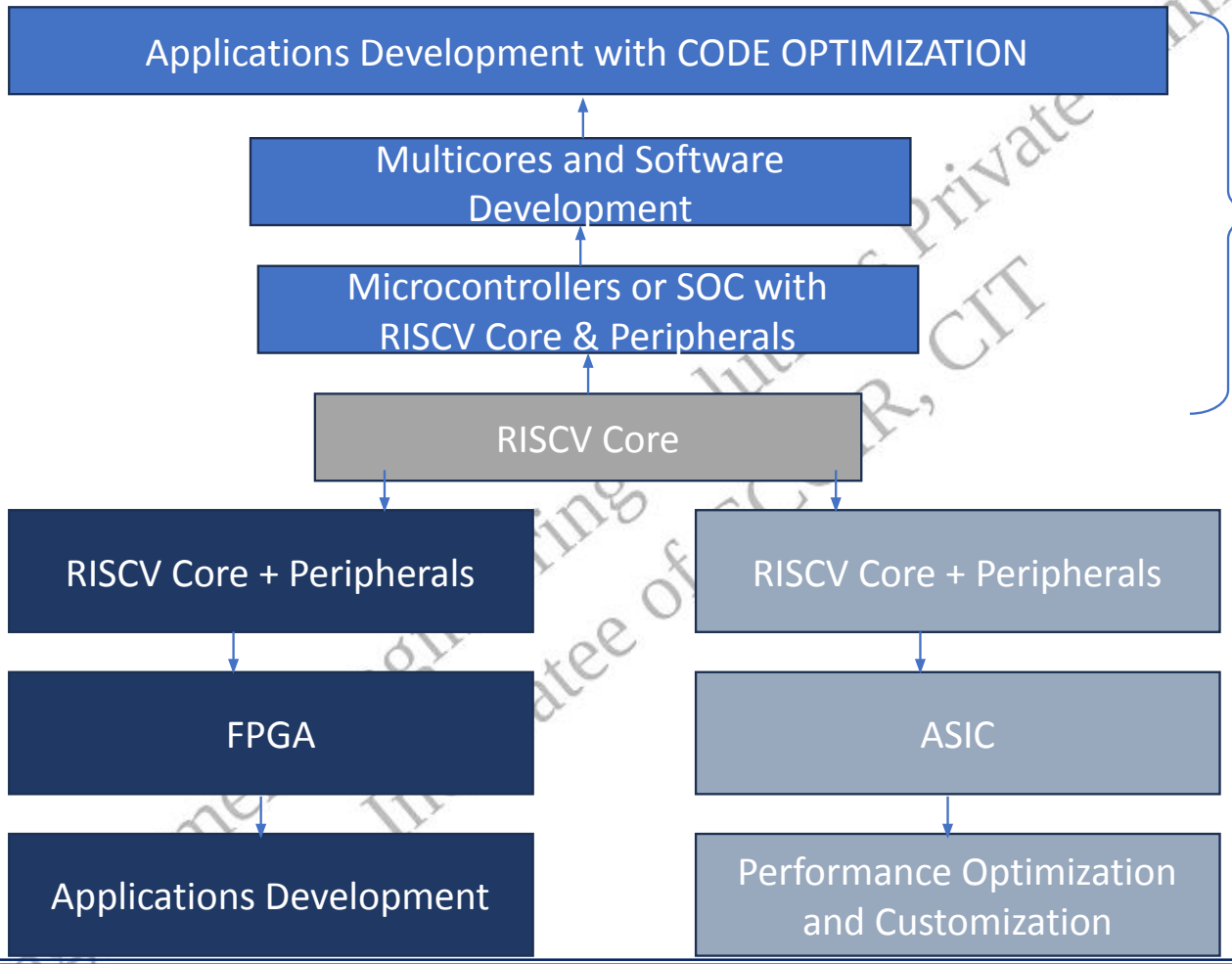
RISC-V Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register
- Pipeline Performance
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



RISCV ECOSYSTEM



Software Approach With RISCV Simulators

Firmware development with RISC V (Soft Core)

Hardware development with RISC V Hard Core

CDAC RISCV

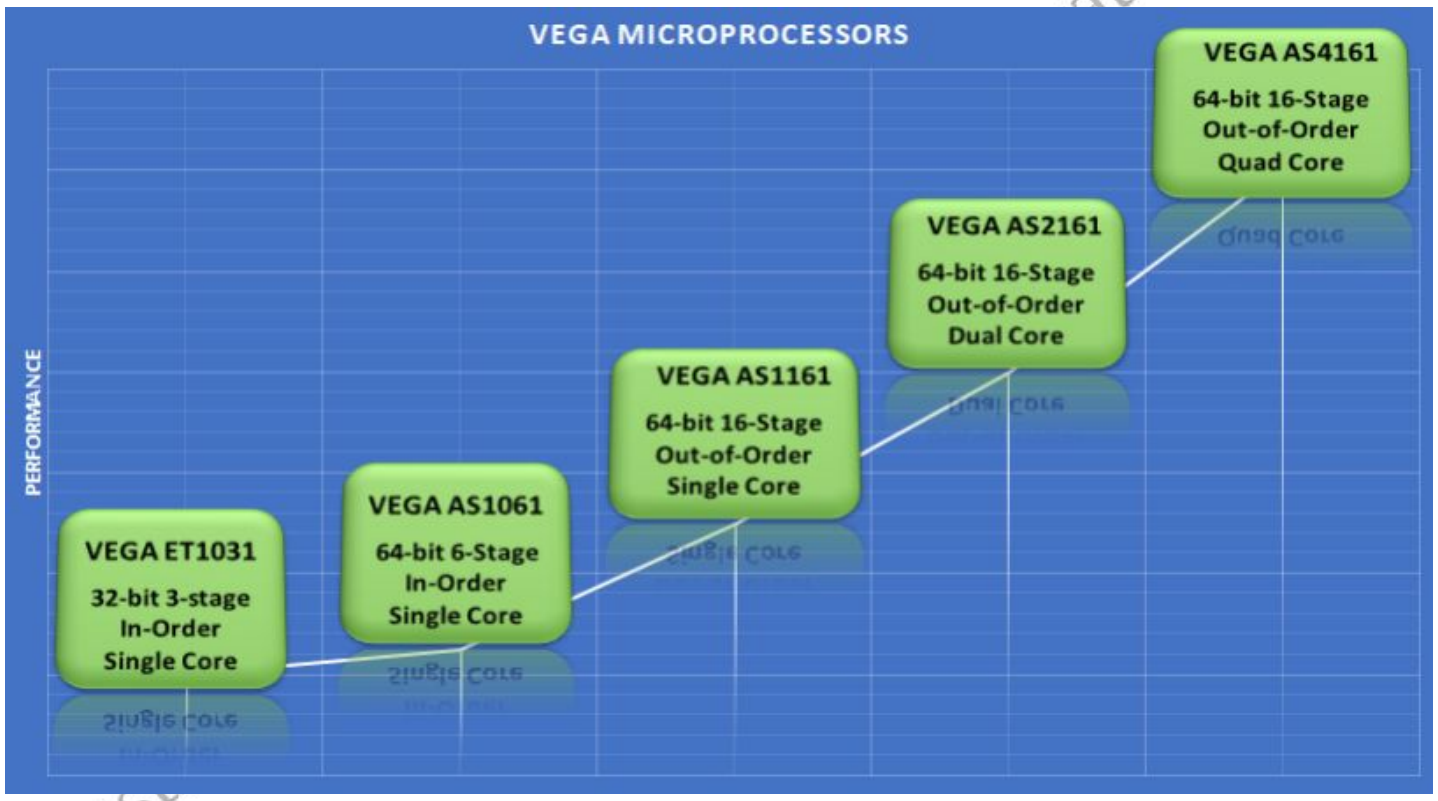

ANGSTROMERS


CCCIR
CAMBRIAN
CONSULTANCY CENTER AND INDUSTRIAL RESEARCH


CAMBRIDGE
INSTITUTE OF TECHNOLOGY
An Autonomous Institution


Chips to Startup
Programme

CDAC-VEGA MICROPROCESSORS



Key Features	VEGA ET1031	VEGA AS1061	VEGA AS1161	VEGA AS2161	VEGA AS4161
RISC-V ISA	32-bit RV32IM	64- bit RV64IMAFDC	64-bit RV64IMAFD	64-bit RV64IMAFD	64-bit RV64IMAFD
No of cores	1	1	1	2	4
Pipeline	In-order	In-order	Out-of-Order	Out-of-Order	Out-of-Order
Pipeline stages	3-Stage	6-Stage	13-16 Stage	13-16 Stage	13-16 Stage
Superscalar	No	No	Yes	Yes	Yes
Processor modes	Machine	Machine/ Supervisor/User	Machine/ Supervisor/User	Machine/ Supervisor/User	Machine/ Supervisor/User
MMU	Optional	Yes	Yes	Yes	Yes
Debug	Optional	Yes	Yes	Yes	Yes
Branch Predictor	No	Yes	Yes	Yes	Yes
L1 ICache	TIM	8KB	32KB	32KB	32KB
L1 DCache	TIM	8KB	32KB	32KB	32KB
L2 Caches	No	No	No	512KB	1024KB
Bus Interface	AHB/AXI4	AHB/AXI4	AHB/AXI4/ACE	AHB/AXI4/ACE	AHB/AXI4/ACE
IEEE 754-2008 compliant FPU	No	Single and Double precision	Single and Double precision	Single and Double precision	Single and Double precision
Availability	Now	Now	Now	Now	Now

The complete ecosystem for embedded design with the VEGA processors, comprises of Board Support Packages, SDK with integrated tool chain, IDE plug-ins and Debugger for the development, testing and debugging. Linux and FreeRTOS & ZephyrOS Operating Systems have been ported and are available as part of the ecosystem.

VEGA Series

VEGA AS4161 features a quad core out-of-order processing engine with a 16 stage pipeline for high performance compute requirements.

VEGA ET1031

VEGA ET1031 is a 32-bit single core 3-stage in-order RISC-V processor.
Read more >>

VEGA AT1051

VEGA AT1051 is a 32-bit CPU IP core based on RISC-V Instruction Set Architecture.
Read more >>

VEGA AS1061

VEGA AS1061 is a 64-bit single core 6-stage in-order RISC-V processor.
Read more >>

VEGA AS1161

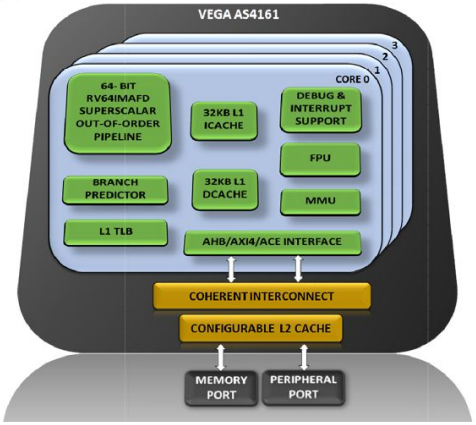
VEGA AS1161 is a 64-bit single core 16-stage pipeline out-of-order RISC-V processor
Read more >>

VEGA AS2161

VEGA AS2161 64-bit dual core 16-stage pipeline out-of-order RISC-V processor.
Read more >>

VEGA AS4161

VEGA AS4161 is a 64-bit quad core 16-stage pipeline out-of-order RISC-V processor.
Read more >>

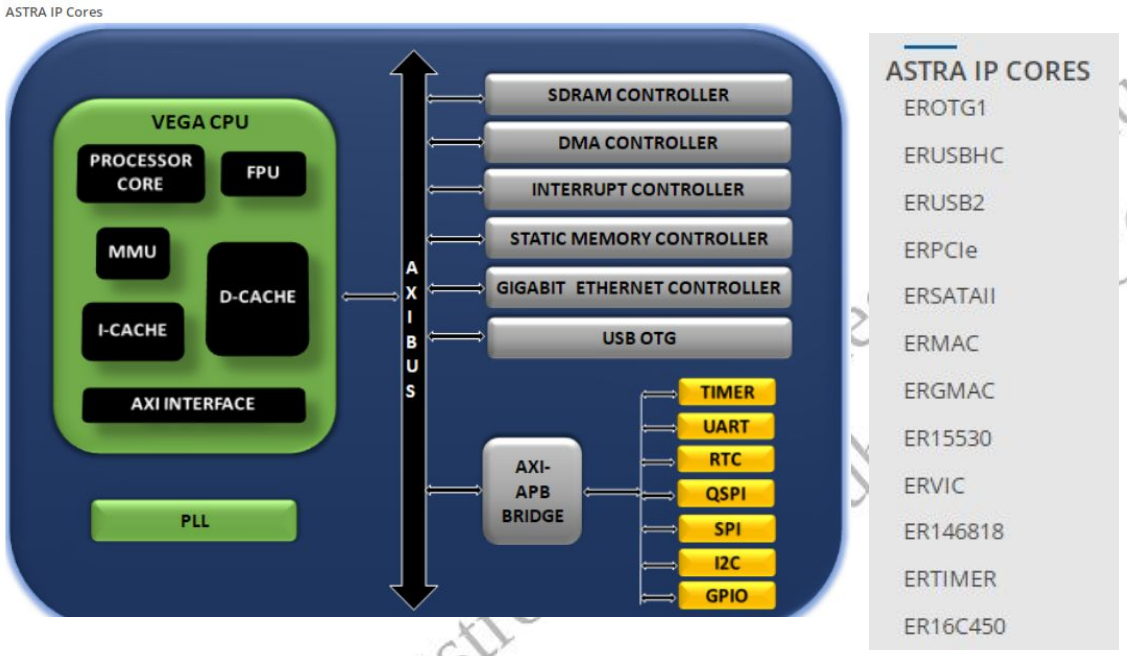


VEGA AS4161 : 64-bit High performance Quad core Microprocessor

- **Key features**
- RISC-V 64G (RV64IMAFD) Instruction Set Architecture
- 13-16 stage out-of-order pipeline implementation
- Advanced branch predictor: BTB, BHT, RAS
- Harvard architecture, separate Instruction and Data memories
- User-, Supervisor- and Machine-mode privilege levels
- Fully-featured memory subsystem with Linux support
  - Memory Management Unit
  - Page-based virtual memory
  - Configurable L1 cache
  - Configurable L2 cache
- High performance Multi core Interconnect
- High-performance IEEE 754-2008 compliant floating-point unit
- AXI4- / ACE, compliant external interface
- Platform level Interrupt Controller
  - up to 127 IRQs
  - Low interrupt latency
- Vectored interrupt support

IP CORES FROM CDAC

ASTRA SYSTEM & PERIPHERAL IPs



ASTRA IP CORES

- EROTG1
- ERUSBHC
- ERUSB2
- ERPCIe
- ERSATAII
- ERMAC
- ERGMAC
- ER15530
- ERVIC
- ER146818
- ERTIMER
- ER16C450

1.Processor IPs

- VEGA ET1031 (32-bit single core 3-stage in-order RISC-V processor)
- VEGA AS1061 (64-bit single core 6-stage in-order RISC-V processor)
- VEGA AS1161 (64-bit single core 16-stage pipeline out-of-order RISC-V processor)
- VEGA AS2161 (64-bit dual core 16-stage pipeline out-of-order RISC-V processor)
- VEGA AS4161 (64-bit quad core 16-stage pipeline out-of-order RISC-V processor)

2. System IPs

- ERPLIC - Interrupt Controller
- ERDMA - Direct Memory Access Controller

3. Peripheral IPs

- EROTG1 – USB2.0 On-the-Go Controller
- ERUSBHC – USB2.0 Host Controller
- ERUSB2 – USB2.0 Function Controller
- ERMAC - Ethernet Media Access Controller (10/100 Mbps)
- ERGMAC - Gigabit Ethernet Media Access Controller
- ERTIMER - Configurable Timer
- ER16C450 – UART
- ERSPI - SPI Controller
- ERI2C - I2C Controller
- ERPWM - PWM Controller
- ERGPIO - GPIO Controller
- ERSDHOST - SD Host Controller
- ERQSPI - QSPI Controller
- ERSMC - Static Memory Controller (SRAM/NOR)

Products

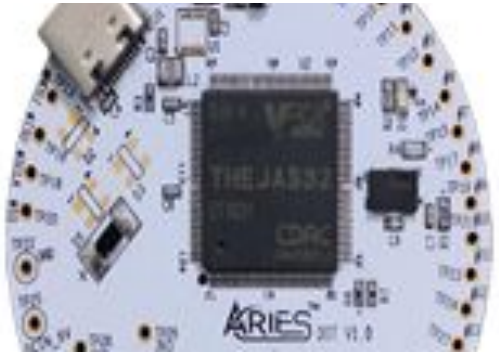
- **THEJAS32 SoC -** is based on the VEGA ET1031 processor, which is a 32 bit single core in-order, 3-stage pipeline processor.

Development Boards

ARIES v2.0

ARIES MICRO v1.0

ARIES IOT v2.0



RISCV Simulators

- RISC-V Interpreter (cornell.edu)
- Ripes

Levels of optimization

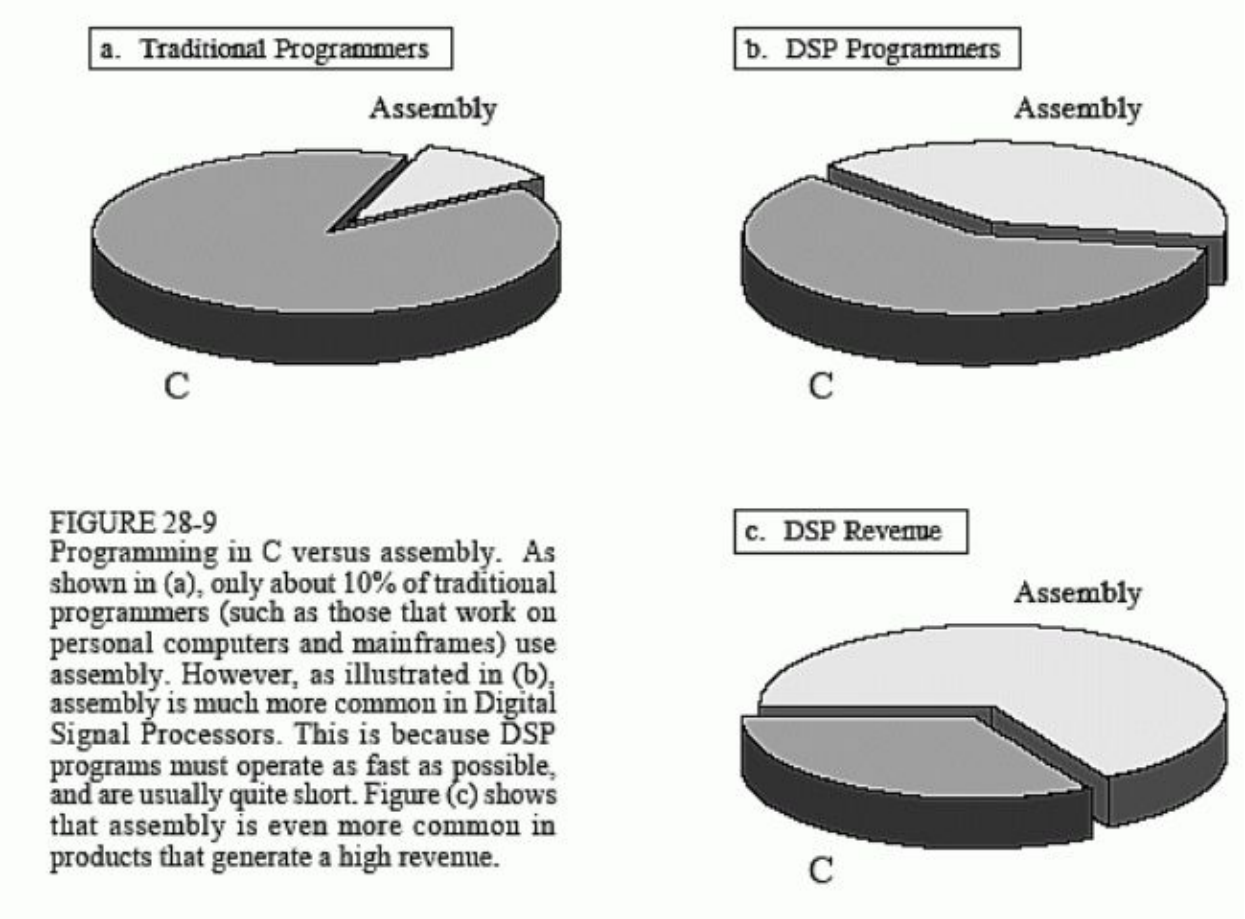


FIGURE 28-9
Programming in C versus assembly. As shown in (a), only about 10% of traditional programmers (such as those that work on personal computers and mainframes) use assembly. However, as illustrated in (b), assembly is much more common in Digital Signal Processors. This is because DSP programs must operate as fast as possible, and are usually quite short. Figure (c) shows that assembly is even more common in products that generate a high revenue.

Simple Example – vector multiplication

C – Program

```
001     #define LEN 20
002       float dm x[LEN];
003       float pm y[LEN];
004       float result;
005
006     main()
007
008     {
009       int n;
010       float s;
011       for (n=0;n<LEN;n++)
012         s += x[n]*y[n];
013       result = s
014     }
```

Assembly Program

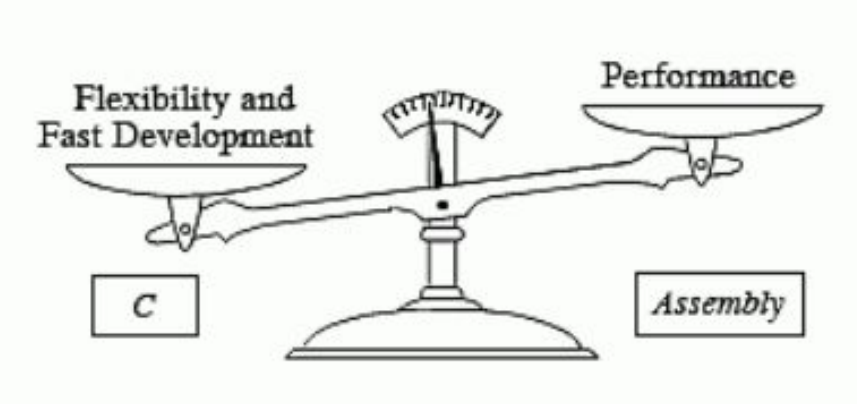
```
001     i12 = _y;                       /* i12 points to beginning of y[ ] */
002     i4 = _x;                        /* i4 points to beginning of x[ ] */
003
004     lcntr = 20, do (pc,4) until lce;  /* loop for the 20 array entries */
005       f2 = dm(i4,m6);               /* load the x[ ] value into register f2 */
006       f4 = pm(i12,m14);             /* load the y[ ] value into register f4 */
007       f8 = f2*f4;                   /* multiply the two values, store in f8 */
008       f12 = f8 + f12;               /* add the product to the accumulator in f12 */
009
010     dm(_result) = f12;              /* write the accumulator to memory */
```

Optimized ASM code

```
001    i12 = _y;                                   /* i12 points to beginning of y[ ] */
002    i4 = _x;                                    /* i4 points to beginning of x[ ] */
003
004    f2 = dm(i4,m6), f4 = pm(i12,m14)            /* prime the registers */
005    f8 = f2*f4, f2 = dm(i4,m6), f4 = pm(i12,m14);
006
007    lcntr = 18, do (pc,1) until lce;            /* highly efficient main loop */
008    f12 = f8 + f12, f8 = f2*f4, f2 = dm(i4,m6), f4 = pm(i12,m14);
009
010    f12 = f8 + f12, f8 = f2*f4;                 /* complete the last loop */
011    f12 = f8 + f12;
012
013    dm(_result) = f12;                          /* store the result in memory */
```

Comparison

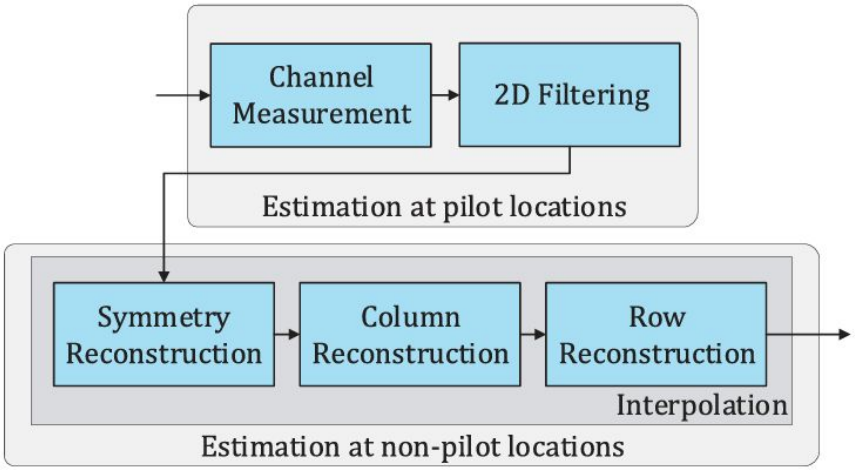
- Number of clock cycles required by the unoptimized and the optimized programs
  - Unoptimized code
    - 20 loops, with four actions being required in each loop
    - 80 clock cycles to carry out the actions within the loops, plus 5 clock cycles of overhead, for a total of 85 clock cycles
  - Optimized code
    - 18 loops in 18 clock cycles,
    - 11 clock cycles of overhead to prime the registers and complete the last loop.
    - This results in a total execution time of 29 clock cycles,
    - about three times faster than the brute force method



Case Study 1

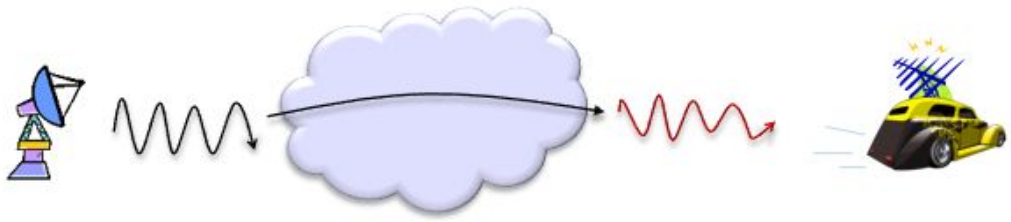
Channel Estimation for Advanced 5G/6G

The process of figuring out channel characteristics is called Channel Estimation.



Channel do the following to the signal going through

- attenuate
- phase-shift
- add noise



How much of the followings are added to the signal at a specific frequency and a specific time ?

- attenuate
- phase-shift
- add noise

What if the frequency and time changes ?

Processing of finding the answer to these questions is called

Channel Estimation

Channel Estimation for Advanced 5G/6G

- Downlink peak data rates for 5G/6G (3rd Generation Partnership Project (3GPP) standard model) – 7.3 Gb/s - 7.5 Gb/s.
- With growing data rates latency is a challenge
- Implementation of channel estimation on hardware platforms is a challenge with processing complexities.

Channel Estimation Methods	Computational Complexity
LS	$O(N\log_2 N)$
DNT [33]	$O(N_P)+O(N\log_2 N)$
WNT [37]	$O(N_P)+O(N\log_2 N)$
IMOT [36]	$O(LN_P)+O(N\log_2 N)$
AFS	$O(N\log_2 N)$
AFS-DT	$O(N_P)+O(N\log_2 N)$

Research problem and Innovation – CE for 5G/6G

- Linear interpolation expression:

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,l_{pi}) + \frac{\hat{\boldsymbol{H}}(k,l_{p(i+1)}) - \hat{\boldsymbol{H}}(k,l_{pi})}{l_{p(i+1)} - l_{pi}} \cdot (l - l_{pi}),$$

Implementing this expression:

- Step 1 – Develop C or python program
- Step 2 – Simulate the program and verify functionality
- Step 3 – Convert this C code to Assembly code or Op code
- Step 4 – Implement the code on the hardware
- Step 5 – Estimate the processing time, latency, throughput, power
- Step 6 – Optimize

Operations required

- We see that there ar
  - a division (div),
  - 3× subtraction (sub),
  - a multiplication (mpy),
  - an addition (add) and
  - at least 2× fixed point type cast (cast).

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,l_{pi}) + \frac{\hat{\boldsymbol{H}}(k,l_{p(i+1)}) - \hat{\boldsymbol{H}}(k,l_{pi})}{l_{p(i+1)} - l_{pi}} \cdot (l - l_{pi}),$$

k ∈ {1, 2,...,N}, and column index l ∈ { lpi + 1,lpi + 2,...,lp(i+1) − 1}.

Optimization process – human intervention

- Step 1
  - lpi = 0 is the time index of the left known column and lp(i+1) = Rps is the time index of the right known column, where Rps is the row pilot spacing for that block.

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,l_{pi}) + \frac{\hat{\boldsymbol{H}}(k,l_{p(i+1)}) - \hat{\boldsymbol{H}}(k,l_{pi})}{l_{p(i+1)} - l_{pi}} \cdot (l - l_{pi}),$$

k ∈ {1, 2,...,N}, and column index l ∈ { lpi + 1,lpi + 2,...,lp(i+1) − 1}.

- Step 2
  - the column index changes to l ∈ 1,2,...,Rps − 1
- Step 3
  - Rps is small in g

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,0) + \left(\hat{\boldsymbol{H}}(k,R_{ps}) - \hat{\boldsymbol{H}}(k,0)\right) \cdot \frac{l}{R_{ps}}$$

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,0) + \left(\hat{\boldsymbol{H}}(k,R_{ps}) - \hat{\boldsymbol{H}}(k,0)\right) \cdot \boldsymbol{x}(l),$$

Operations required are
- a sub,
- a mpy,
- an add and
- a cast per data item

Optimization process – human intervention

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}\left(k, l_{pi}\right) + \frac{\hat{\boldsymbol{H}}\left(k, l_{p(i+1)}\right) - \hat{\boldsymbol{H}}\left(k, l_{pi}\right)}{l_{p(i+1)} - l_{pi}} \cdot \left(l - l_{pi}\right),$$

$$\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,0) + \left(\hat{\boldsymbol{H}}\left(k, R_{ps}\right) - \hat{\boldsymbol{H}}(k,0)\right) \cdot \boldsymbol{x}(l),$$

k ∈ {1, 2,...,N}, and column index l ∈ { lpi + 1,lpi + 2,...,lp(i+1) − 1}.

Requires multiply-accumulate (mac) operation efficiently

- Step 4
  - Recursive expression $\hat{\boldsymbol{H}}(k,l) = \hat{\boldsymbol{H}}(k,l-1) + \left(\hat{\boldsymbol{H}}\left(k, R_{ps}\right) - \hat{\boldsymbol{H}}(k,0)\right) \cdot \dot{\boldsymbol{x}}(l),$

Algorithm 1: RR Block High Throughput Variant

input: Known columns $\hat{\boldsymbol{H}}(:,0)$, $\hat{\boldsymbol{H}}\left(:, R_{ps}\right)$ and LUT $\dot{x}$
output: Interpolated block $\hat{\boldsymbol{H}}(:,:)$
// Data registers v_1, v_2
// ACC register w_1

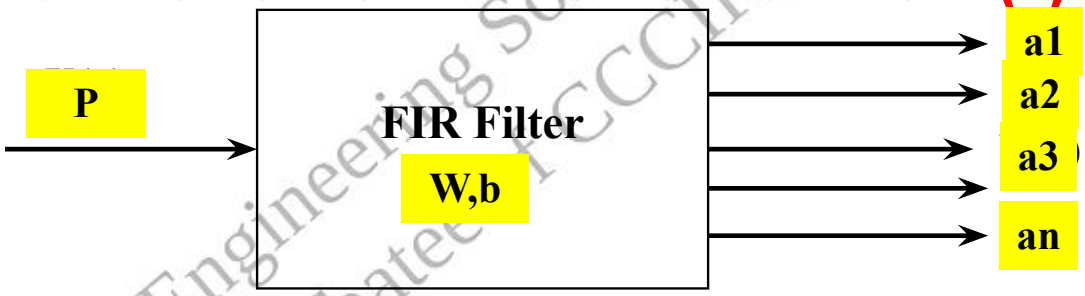
```
for k ← 0 to N − 1 do
    v1 = sub(Ĥ(k, Rps), Ĥ(k, 0));
    w1 = cast(Ĥ(k, 0));
    for l ← 1 to Rps − 1 do
        w1 = mac(w1, v1, ẋ);
        v2 = cast(w1);
        Ĥ(k, l) = v2;
    end
end
```

Implementation	Processing Load		Delay
	$\left[\frac{cycles}{RB \cdot OFDM}\right]$	$\left[\frac{cycles}{data\ vector}\right]$	$\left[\frac{cycles}{RB \cdot OFDM}\right]$
Algorithm 1 (19)	1.52	1.9	$1.52(R_{ps}-1)$
Algorithm 2 (17)	2.04	2.55	2.04

Case study 2

ANN and Filter

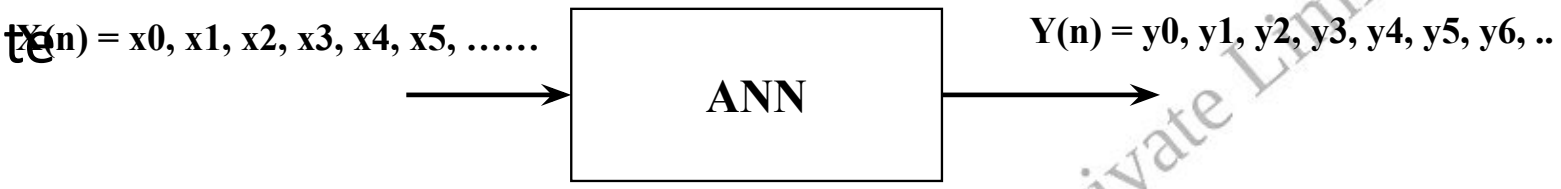
- Filter has input **x(n)**, filter coefficient **h(n),** output **y(n)**
- ANN has input **P**, filter coefficient **W, b** and output **a**
- **Output in filter is Y = X*H □ realized using multiplier and adder**
- **Output in ANN is a = P.W □ realized using multiplier and adder**
  - **Additional macro unit is network activation function f(x)**



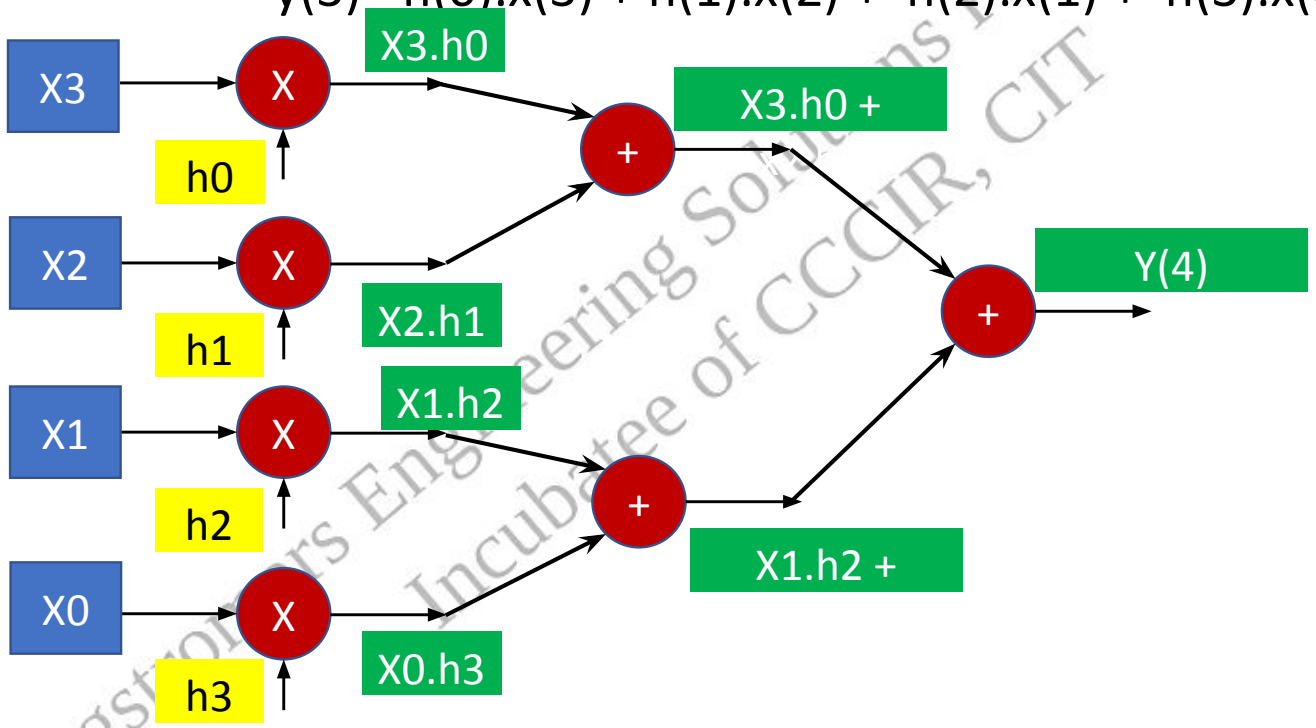
$$y[n] = x[n] * h[n] = \sum_{m=-\infty}^{\infty} x[m] \cdot h[n-m] = \sum_{m=-\infty}^{\infty} h[m] \cdot x[n-m]$$

Filter/ANN basics

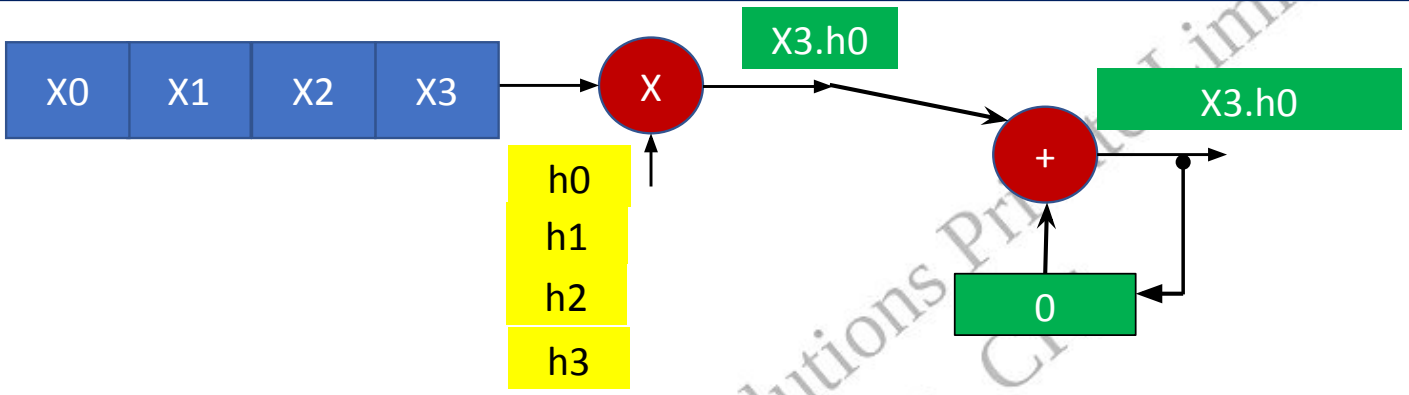
- To compute



$$y(3)= h(0).x(3) + h(1).x(2) + h(2).x(1) + h(3).x(0)$$



Fully Serial



Clock 1,2 output = x3.h0+0

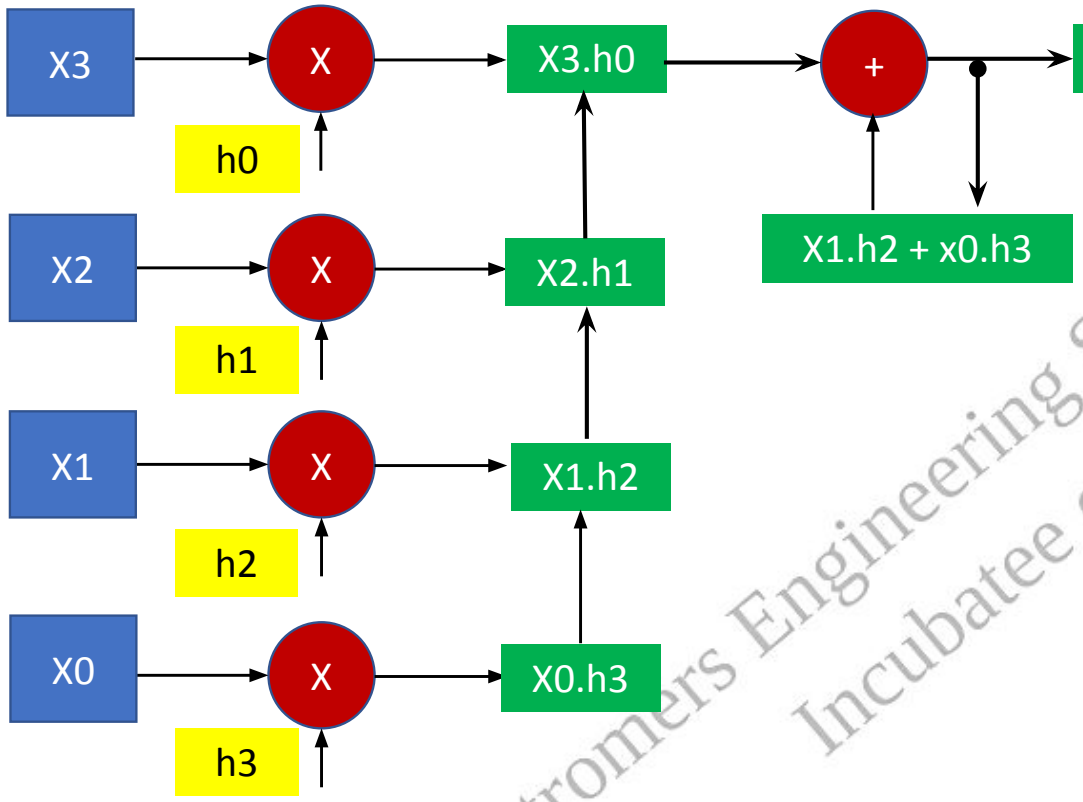
Clock 3,4 output = x3.h0 + x2h1

Clock 5,6 output = x3.h0 + x2h1 + x1h2

Clock 7,8 output = x3.h0 + x2h1 + x1h2 + x0h3

Datapath	Fully parallel	Fully serial
X	4	1
+	3	1
Reg	15	11
Clocks	3	8

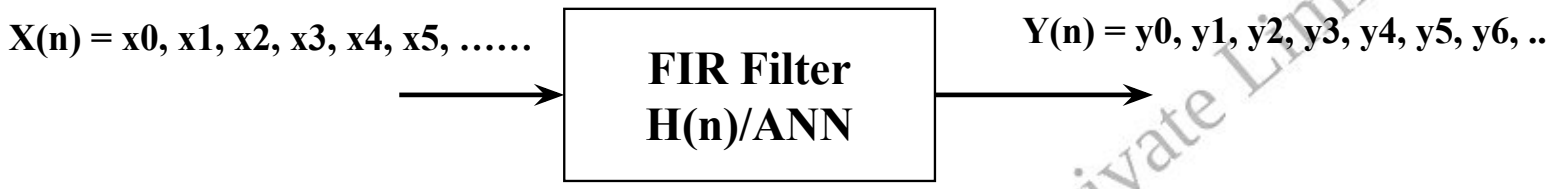
Partially Serial



Data path	Fully parallel	Fully serial	Partially Serial
X	4	1	4
+	3	1	1
Reg	15	11	14
Clocks	3	8	5

Clock 1, All multiplications
Clock 2,, output = x3.h0 + 0
Clock 3, output = x3.h0 + x2h1
Clock 4, output = x3.h0 + x2h1 + x1h2
Clock 5, output = x3.h0 + x2h1 + x1h2 + x0h3

Case Study 2 - Optimization



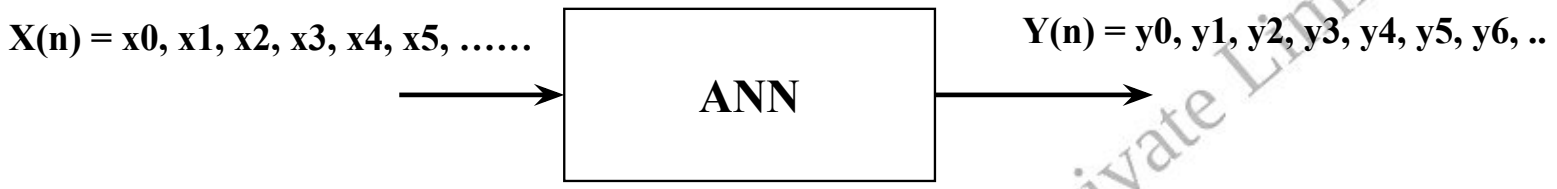
Order h(n)	Filter coefficient
Ho	18
H1	18
H2	24
H3	24
H4	56
H5	31
H6	31
H7	56

$$y(0)= h(0).x(0) + h(1).x(1) + h(2).x(2) + h(3).x(3)+ h(4).x(4) + h(5).x(5) + h(6).x(6) + h(7).x(7)+$$

$$y(1)= h(0).x(1) + h(1).x(2) + h(2).x(3) + h(3).x(4)+ h(4).x(5) + h(5).x(6) + h(6).x(7) + h(7).x(8)+$$

Data path	Fully parallel	Fully serial	Partially Serial
X	8	1	8
+	7	1	1
Reg	27	19	26
Clocks	4	18	9

Case Study 2



Order h(n)	Filter coefficient
Ho	18
H1	18
H2	24
H3	24
H4	56
H5	31
H6	31
H7	56

$$y(0)= h(0).x(0) + h(1).x(1) + h(2).x(2) + h(3).x(3)+ h(4).x(4) + h(5).x(5) + h(6).x(6) + h(7).x(7)+$$

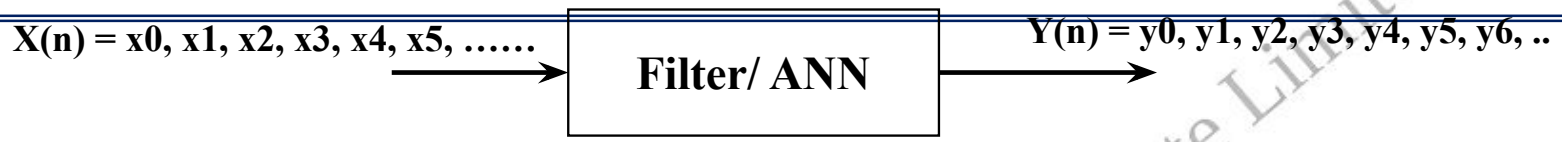
Substituting filter coefficients

$$y(0)= 18.\ x(0) + 18\ .x(1) + 24\ .x(2) + 24\ .x(3)+ 56\ .x(4) + 31\ .x(5) + 31\ .x(6) + 56\ .x(7)+$$

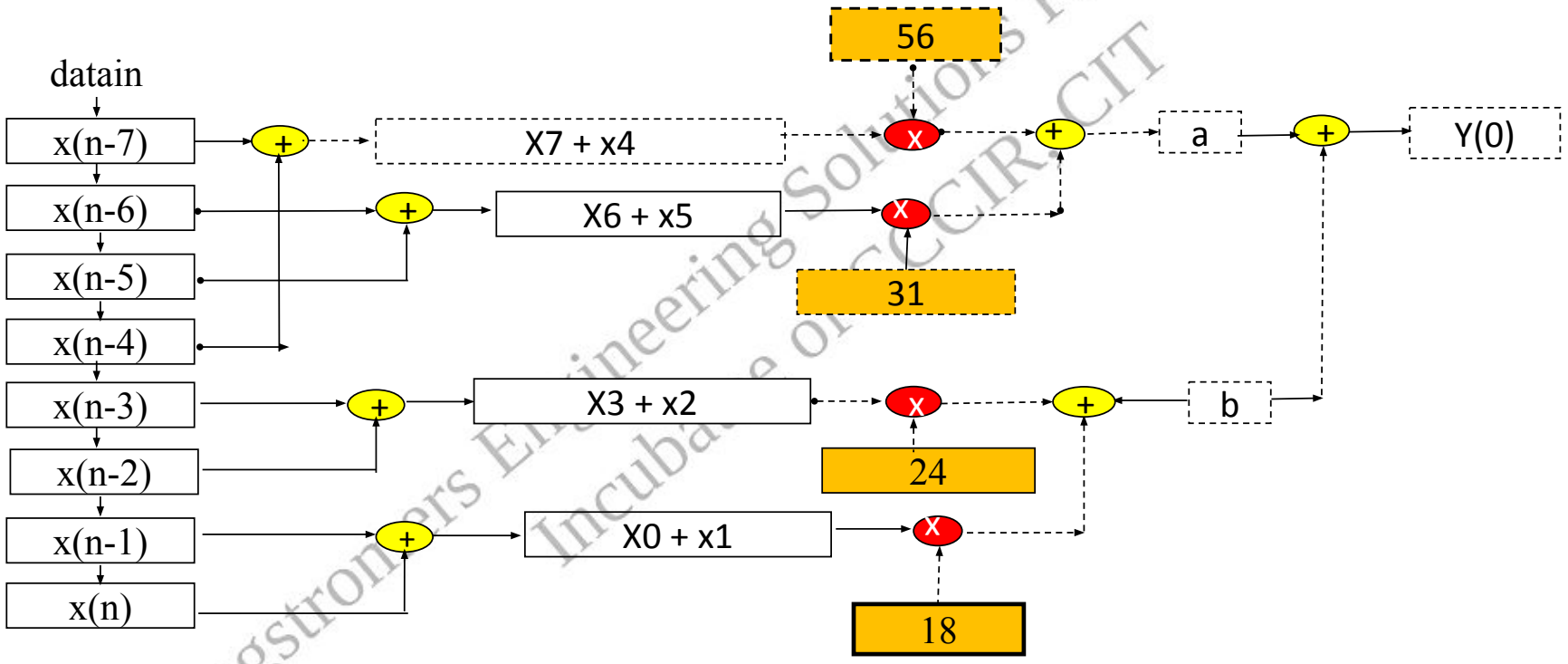
Rearranging the terms

$$y(0)= 18\ [x(0) + x(1)] + 24\ [x(2) + x(3)] + 56\ [x(4) + x(7)]+ 31\ [x(5) + x(6)]$$

Case Study 2



y(0)= 18 [x(0) + x(1)] + 24 [x(2) + x(3)] + 56 [x(4) + x(7)]+ 31 [x(5) + x(6)]



Comparison of all Designs

Datapath	Fully parallel	Fully serial	Partially Serial	New design
X	8	1	8	4
+	7	1	1	7
Reg	27	19	26	19
Clocks	4	18	9	4

Further Optimization

y(0)= 18 [x(0) + x(1)] + 24 [x(2) + x(3)] + 56 [x(4) + x(7)]+ 31 [x(5) + x(6)]

y(0) = 18 [a1] + 24 [a2] + 56 [a3]+ 31 [a4]

Decimal No.	Binary No.					
	32	16	8	4	2	1
18	0	1	0	0	1	0
24	0	1	1	0	0	0
56	1	1	1	0	0	0
31	0	1	1	1	1	1
32	1	0	0	0	0	0

- **Lets assume a1 = 2, a2 = 7, a3 = 1, a4 = 4**
- **y(0) = 36 + 168 + 56 + 124 = 384**
- **y(0) = 36 + 168 + 56 + 128 = 388**

Further Optimization – ZERO PAD METHOD

y(0) = 18 [2] + 24 [7] + 56 [1]+ 32 [4]

Decimal No.	Binary No.					
	32	16	8	4	2	1
2	0	0	0	0	1	0
7	0	0	0	1	1	1
1	0	0	0	0	0	1
4	0	0	0	1	0	0

Decimal No.	Binary No.					
	32	16	8	4	2	1
18	0	1	0	0	1	0
24	0	1	1	0	0	0
56	1	1	1	0	0	0
32	1	0	0	0	0	0

$2 \times 18 = 36 \square 2 \times (1x2^4 + 1x2^1) \square 2 \times (1x2^4) + 2 \times (1x2^1)$

2 = 00 00 10

2 x 2 = 00 01 00

2 x 4 = 00 10 00

2 x 8 = 01 00 00

2 x 16 = 10 00 00

00 00 10 **00 00**

00 00 10 **0**

0	0	0	0	0	1	0	**0**	**0**	**0**	**0**
+										
			0	0	0	0	0	1	0	**0**
0	0	0	0	0	1	0	**0**	**1**	**0**	**0**

1024	512	256	128	64	32	16	**8**	**4**	**2**	**1**

Further Optimization – ZERO PAD METHOD

y(0) = 18 [2] + 24 [7] + 56 [1]+ 32 [4]

Decimal No.	Binary No.					
	32	16	8	4	2	1
2	0	0	0	0	1	0
7	0	0	0	1	1	1
1	0	0	0	0	0	1
4	0	0	0	1	0	0

Decimal No.	Binary No.					
	32	16	8	4	2	1
18	0	1	0	0	1	0
24	0	1	1	0	0	0
56	1	1	1	0	0	0
32	1	0	0	0	0	0

$7 \times 24 = 168 \rightarrow 7 \times (1x2^4 + 1x2^3) \rightarrow 7 \times (1x2^4) + 7 \times (1x2^3)$

00 01 11

00 01 11 **00 00**

0 00 11 1 **000**

	0	0	0	0	1	1	1	**0**	**0**	**0**	**0**
+	0	0	0	0	0	1	1	1	**0**	**0**	**0**
	0	0	0	1	0	1	0	**1**	**0**	**0**	**0**
	1024	512	256	128	64	32	16	**8**	**4**	**2**	**1**

Further Optimization – ZERO PAD METHOD

y(0) = 18 [2] + 24 [7] + 56 [1]+ 32 [4]

Decimal No.	Binary No.					
	32	16	8	4	2	1
2	0	0	0	0	1	0
7	0	0	0	1	1	1
1	0	0	0	0	0	1
4	0	0	0	1	0	0

Decimal No.	Binary No.					
	32	16	8	4	2	1
18	0	1	0	0	1	0
24	0	1	1	0	0	0
56	1	1	1	0	0	0
32	1	0	0	0	0	0

$1 \times 56 = 56 \square 1 \times (1 \times 2^4 + 1 \times 2^4 + 1 \times 2^3) \square 1 \times (1 \times 2^5) + 1 \times (1 \times 2^4) + 1 \times (1 \times 2^3)$

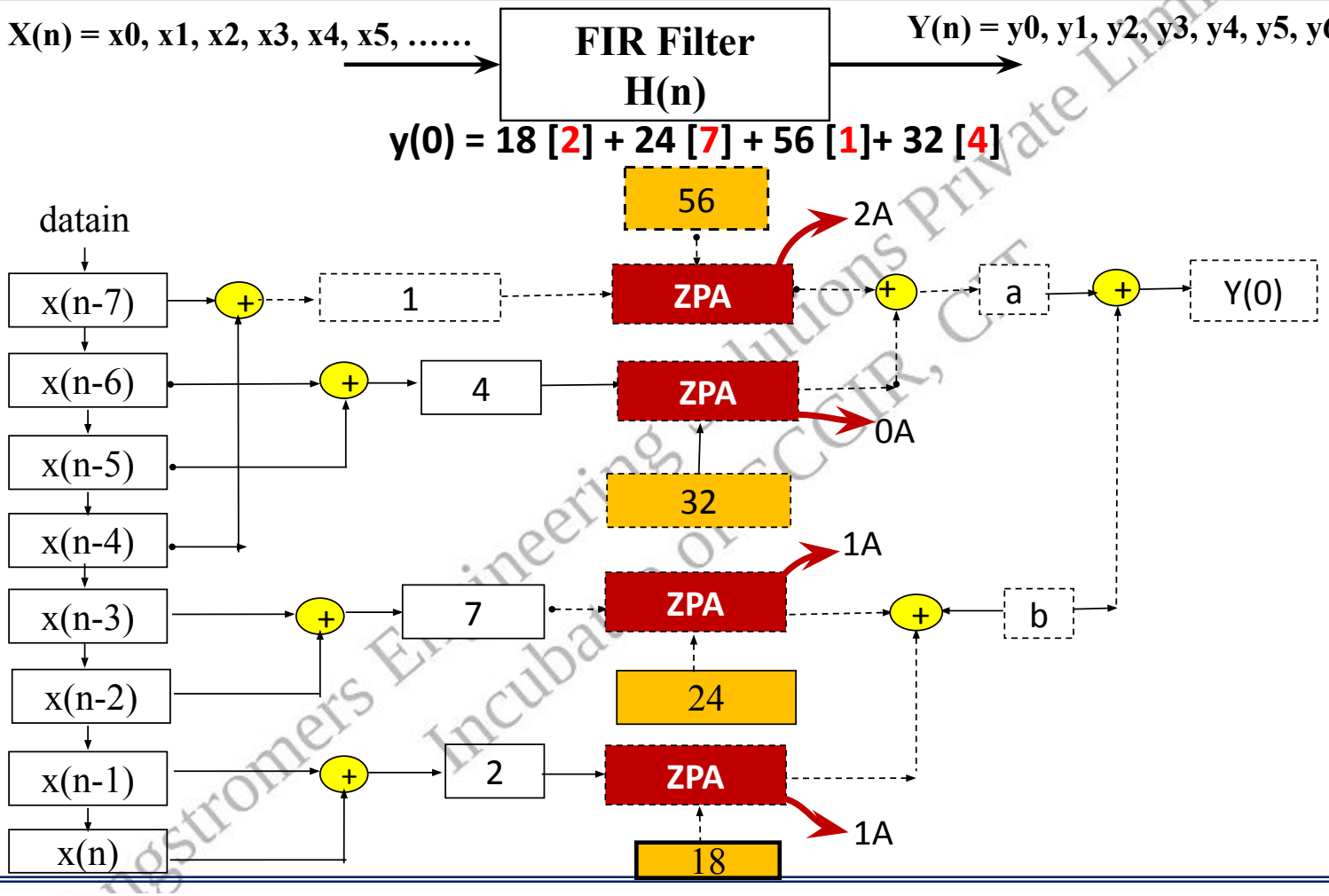
↓

00 00 01

00 00 01 **00 00 0** 0 00 00 1 **0000** 0 00 00 1 **000**

	0	0	0	0	0	0	1	**0**	**0**	**0**	**0**	**0**
	0	0	0	0	0	0	0	1	**0**	**0**	**0**	**0**
+	0	0	0	0	0	0	0	0	1	**0**	**0**	**0**
	0	0	0	0	0	0	1	1	1	**0**	**0**	**0**
	1024	512	256	128	64	32	16	**8**	**4**	**2**	**1**	

Further Optimization using ZERO PAD



Comparison of all Designs

Data path	Fully parallel	Fully serial	Partially Serial	New design	Improved Design
X	8	1	8	4	0
+	7	1	1	7	11
Reg	27	19	26	19	19
Clocks	4	18	9	4	4


Thank You