# ARM Processors ITMC301 Architecture Performance - L2 Fall 2023

By: Dr. Abdussalam Nuri Baryun

Mobile Computing Department, Faculty of Advance Technology, University of Tripoli

## ARM use load-store

- The ARM processor, like all RISC processors, uses a *load-store architecture*.
- This means it has two instruction types for transferring data in and out of the processor: load instructions copy data from memory to registers in the core, and store instructions copy data from registers to memory.
- There are no data processing instructions that directly manipulate data in memory.

## What is instruction set architecture? (ISA)

- The operation codes encodings of instructions that are executed by an Arm-based processor. Also includes other aspects as well such as the exception model, system programming features, memory management and suchlike.
- ISA is a legal contract, between hardware and software.

## How do you decide which instructions to include in an ISA?

- When we are looking at requests from partners or from internal research to add a new instruction, we go through quite a long process of trying to justify that instruction, or, quite commonly, a set of instructions rather than a single instruction.
- We have to show that it gives us some real benefit in performance, the performance of your code running on that CPU. Or maybe not performance.
- Maybe it's security you're trying to achieve. But it has to give you some really concrete benefit that is worth the cost of adding all of the software, the validation software, the implementation costs for all of the different implementations, compiler support, so on and so forth.

## What is the difference between an ISA and a microarchitecture?

- The difference between ISA and the microarchitecture is that the ISA is an abstract concept.
- It defines a set of instruction encodings which software can use, and which hardware has to recognize and implement.
- How that is implemented is a choice for the microarchitecture. So the instruction set architecture is fixed, it's defined by Arm.
- The microarchitecture is defined by whatever team of people is designing that CPU.
- For microarchitecture, there are many different approaches to implementing the Arm architecture.

## Microarchitecture

- Defined by whatever team of people is designing that CPU. And there are many different approaches to implementing the Arm architecture, from very small, efficient cores with in-order pipelines up to very high-performance, state-of-the-art, out-of-order execution, and everywhere in between.

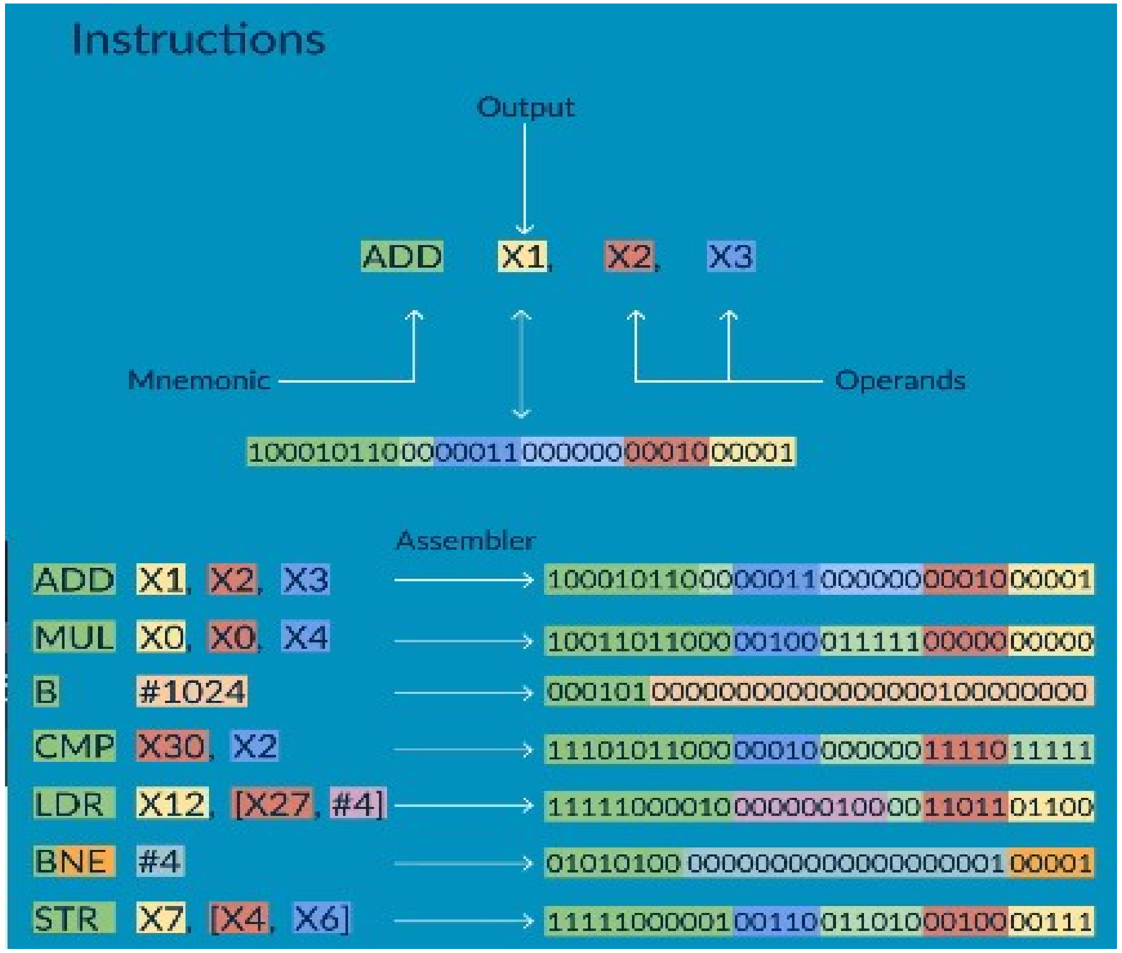
- So the microarchitecture is implementation-specific, the architecture is generic, **and software written for the architecture should run on any microarchitecture.**

- Q: Why does Arm produce processors with different instruction sets or/and different architecture?
  - Arm supports multiple instruction sets.
  - Some of that is to do with legacy: you can't abandon your legacy software, your legacy ecosystem.
  - As the architecture has advanced and we've introduced major new instruction sets, we still have to continue to support old software. It takes years, maybe 10 years to move the software ecosystem to a major new ISA.
- Therefore, the **Answer** is there are different instruction sets for reasons of the market that they're trying to address, and then there are different instruction sets because, to support legacy software and new developed technologies.

- **For example, AArch64, which is the 64-bit architecture that with introduced with ARMv8,**
- also supported the AArch, what we called AArch32, the old 32-bit architecture that was implemented in the ARMv7 architecture, <u>and prior to that including the Arm and the Thumb instruction sets.</u>

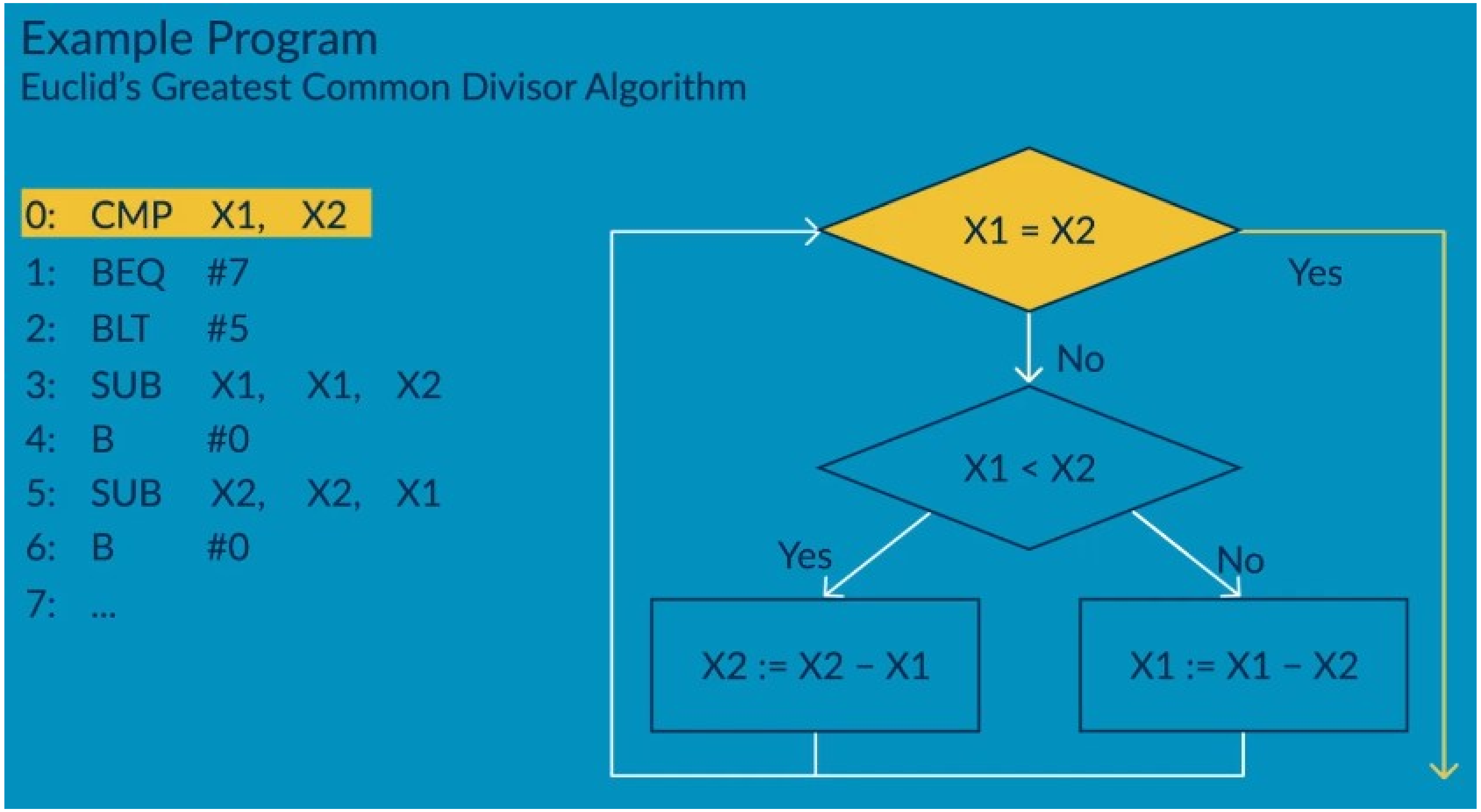
# Translate to binary by the microprocessor assembler

- Using sequence of instructions we can build programs



| | |
|---|---|
| ADD X1, X2, X3 | 10001011000000110000000001000001 |
| MUL X0, X0, X4 | 10011011000001000111110000000000 |
| B #1024 | 00010100000000000000000100000000 |
| CMP X30, X2 | 11101011000000100000001111011111 |
| LDR X12, [X27, #4] | 11111000010000000100001101101100 |
| BNE #4 | 01010100000000000000000000100001 |
| STR X7, [X4, X6] | 11111000001001100110100010000111 |

## Example



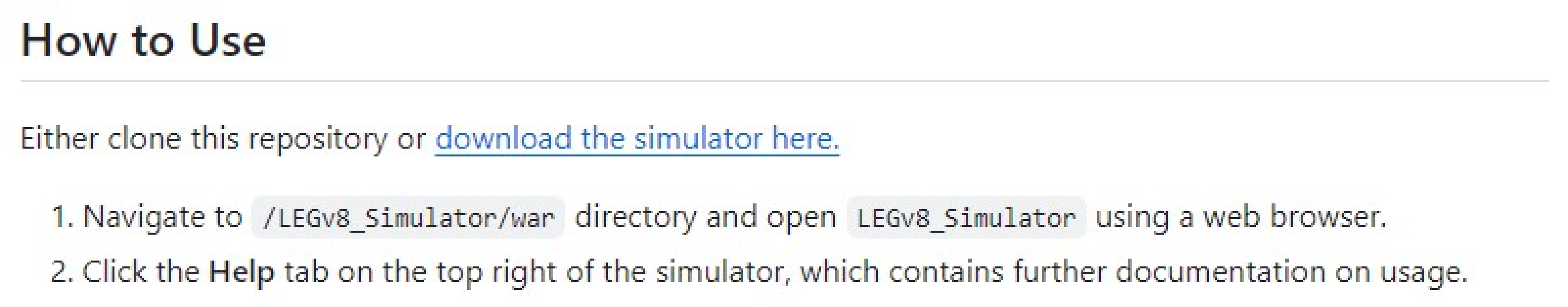
Example Program

Euclid's Greatest Common Divisor Algorithm

```
0:  CMP   X1,  X2
1:  BEQ   #7
2:  BLT   #5
3:  SUB   X1,  X1,  X2
4:  B     #0
5:  SUB   X2,  X2,  X1
6:  B     #0
7:  ...
```

# Lab: Asim (ARM64 simulator)

- https://github.com/arm-university/Graphical-Micro-Architecture-Simulator



## How to Use

Either clone this repository or download the simulator here.

1. Navigate to `/LEGv8_Simulator/war` directory and open `LEGv8_Simulator` using a web browser.
2. Click the **Help** tab on the top right of the simulator, which contains further documentation on usage.

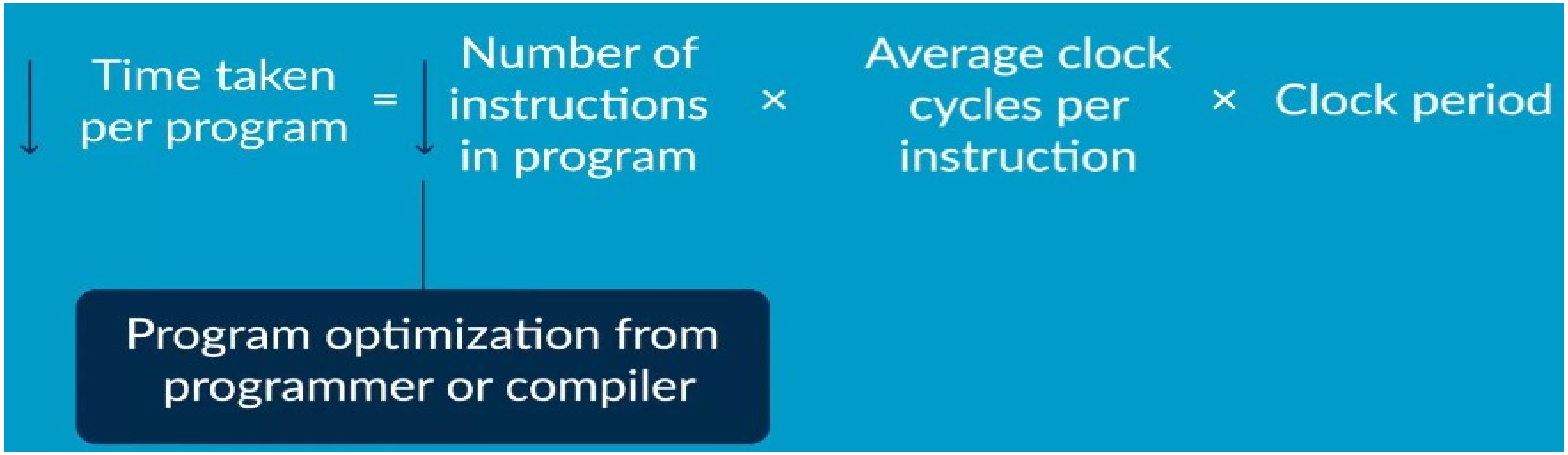
## Microprocessor Performance

- The performance depends on both ARM architecture and microarchitecture.

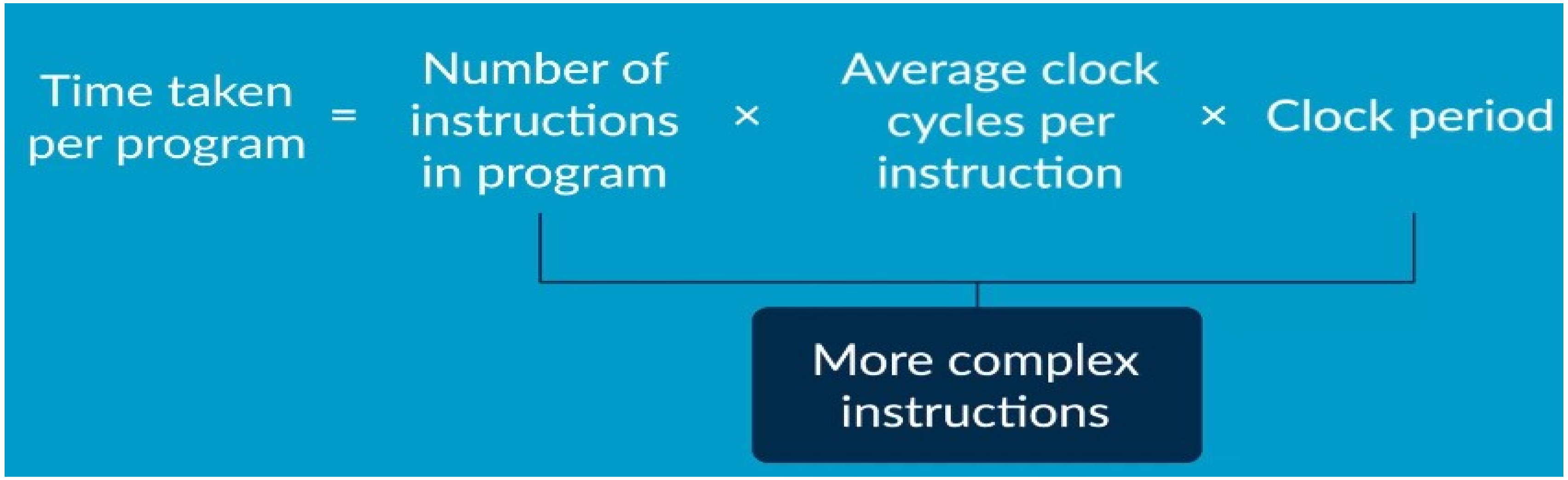
$$\text{Time taken per instruction} = \text{Average clock cycles per instruction} \times \text{Clock period}$$

$$\text{Time taken per program} = \text{Number of instructions in program} \times \overbrace{\boxed{\text{Average clock cycles per instruction} \times \text{Clock period}}}^{\text{Time taken per instruction}}$$

## Two possible ways with trade-offs

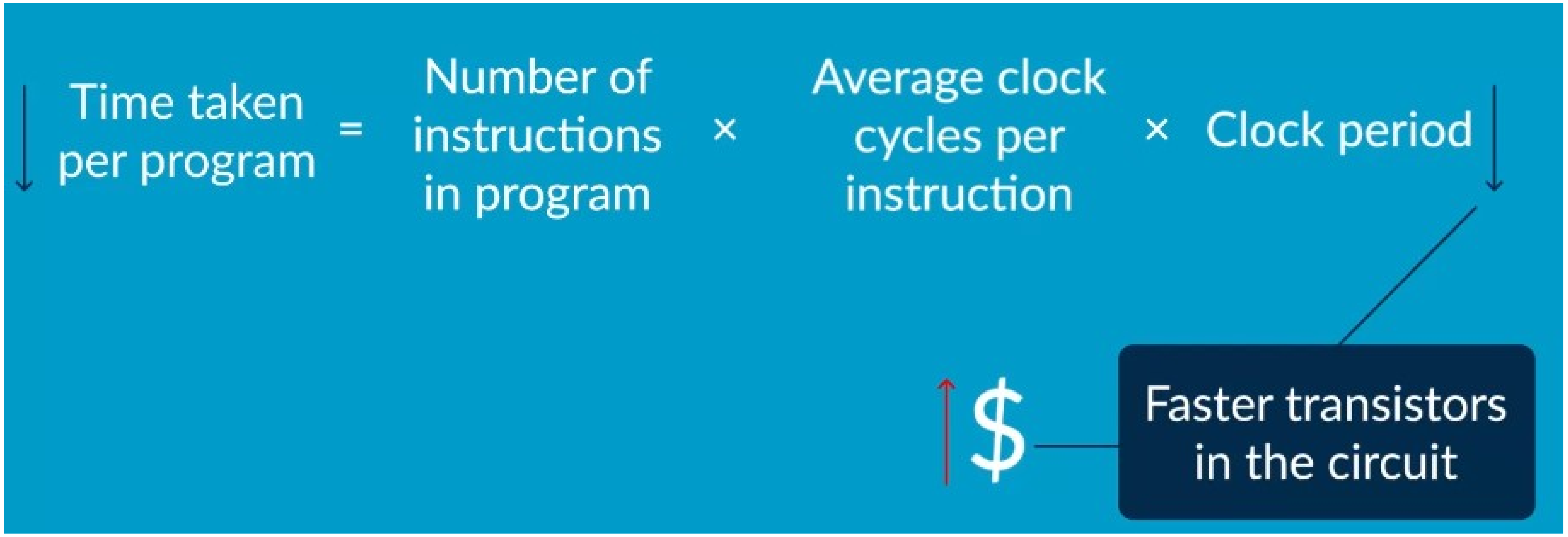


Time taken per program
=
Number of instructions in program
×
Average clock cycles per instruction
×
Clock period
Program optimization from programmer or compiler

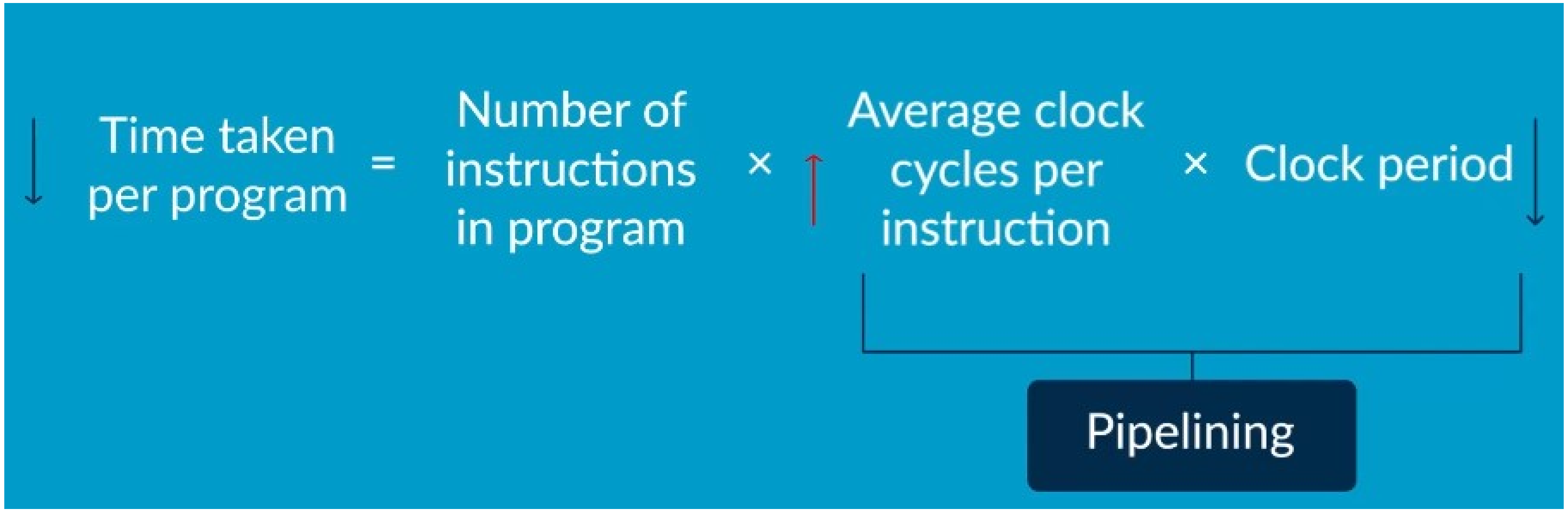


Time taken per program
=
Number of instructions in program
×
Average clock cycles per instruction
×
Clock period
More complex instructions

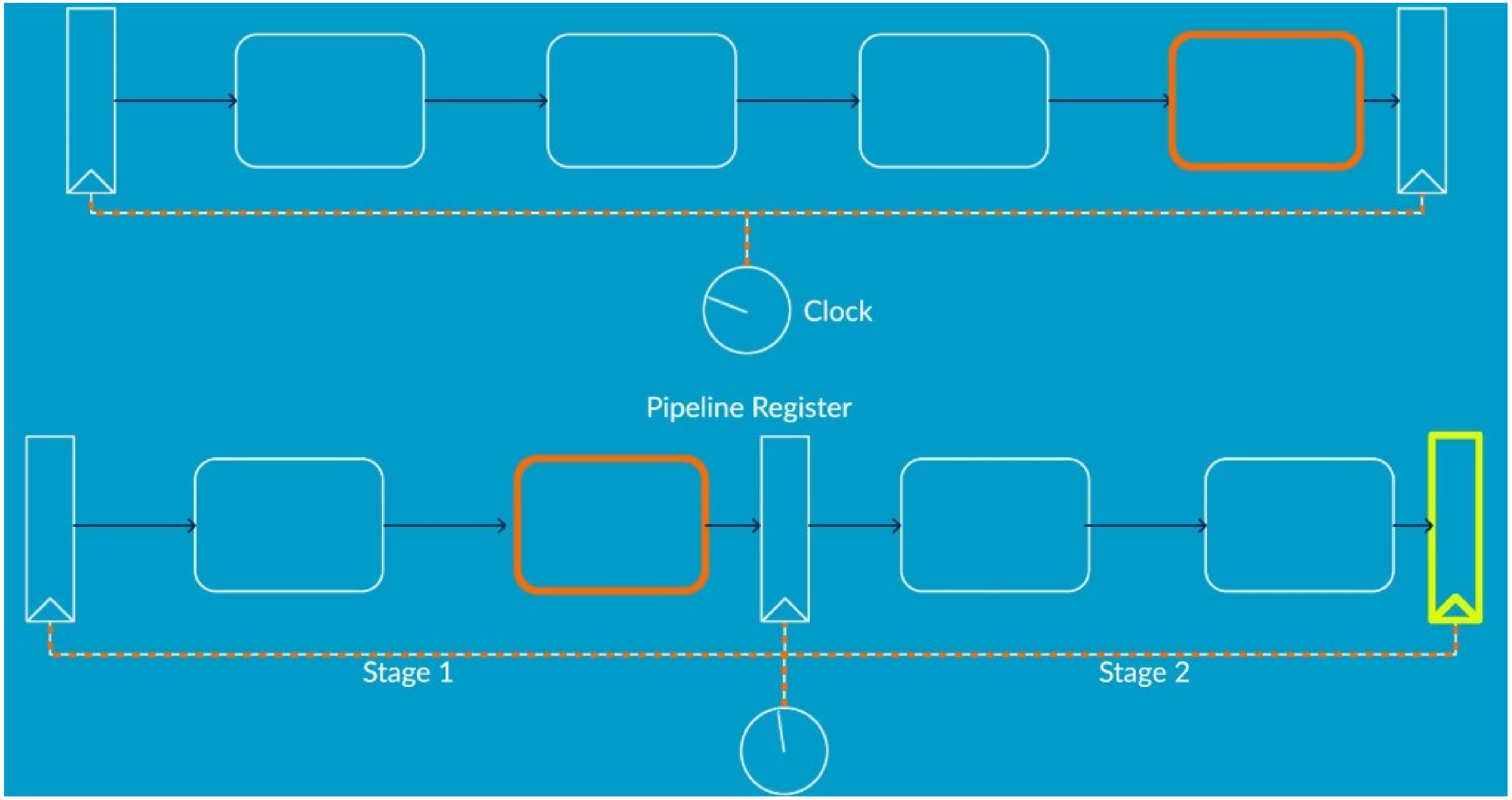
## Third optional way


↓ Time taken per program
=
Number of instructions in program
×
Average clock cycles per instruction
×
Clock period ↓
↑ $
Faster transistors in the circuit

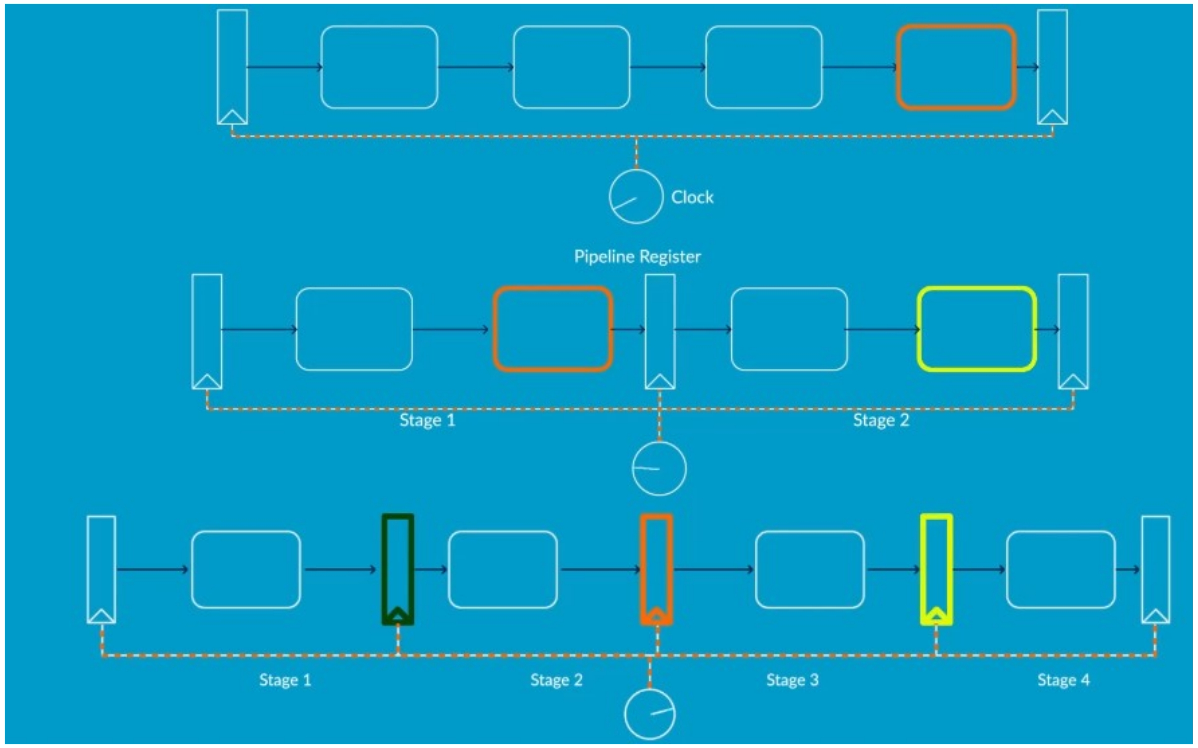
## Using then pipelining


↓ Time taken per program
=
Number of instructions in program
×
↑ Average clock cycles per instruction
×
Clock period ↓
Pipelining

## Example: having two stage will increase avg cycle per instruction by 2

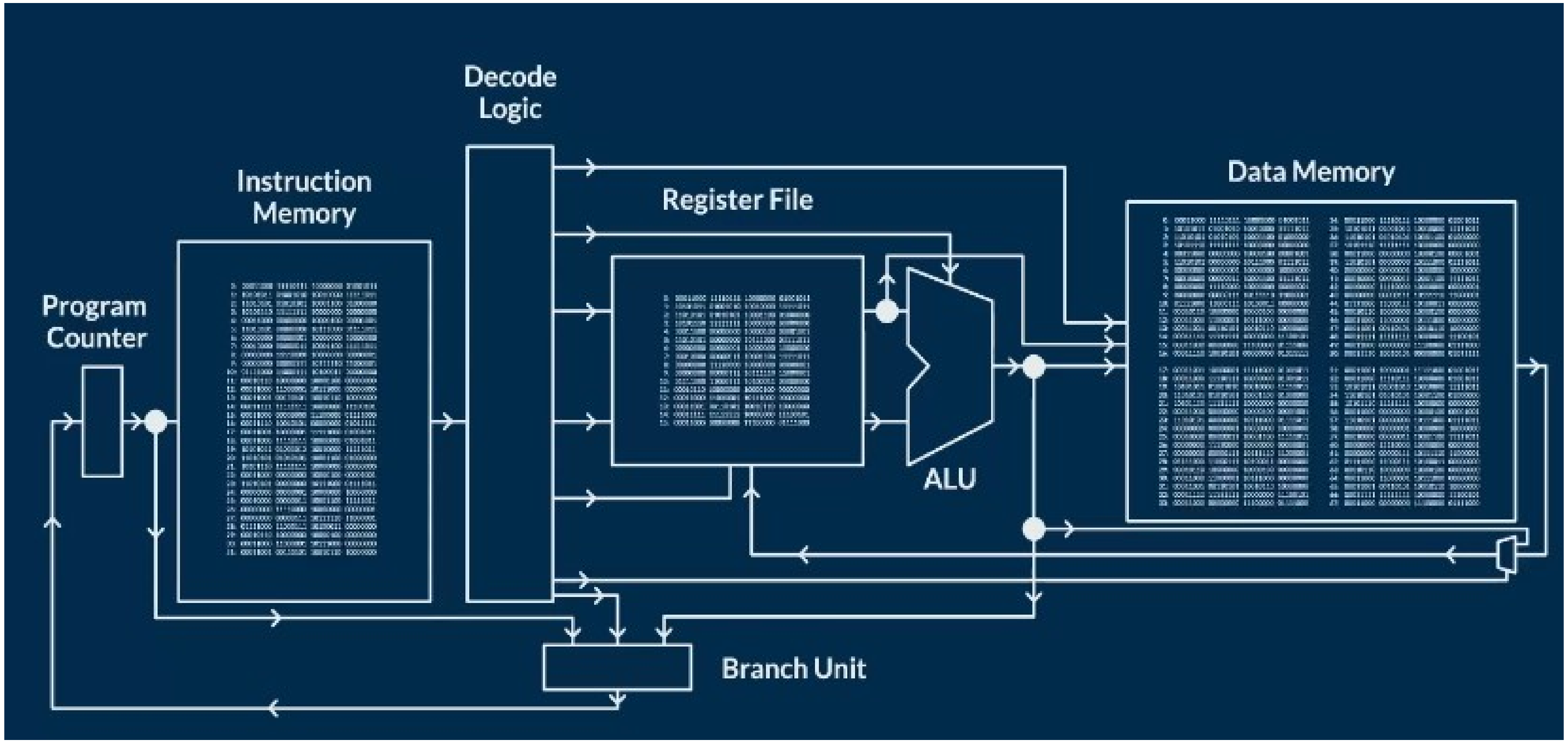


- Increasing stages for pipelining by using registers can increase performance.

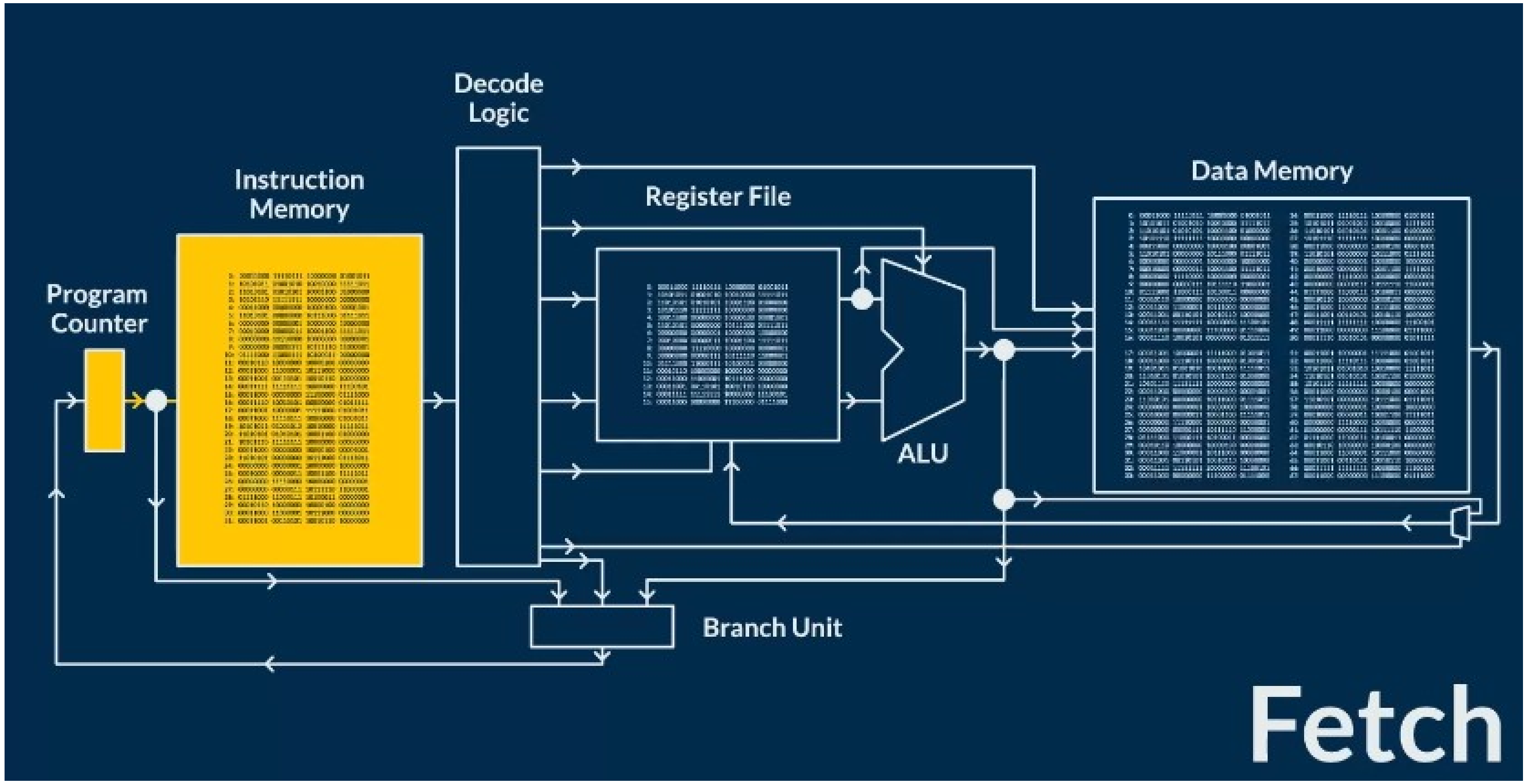


# Components of ARM microprocessor

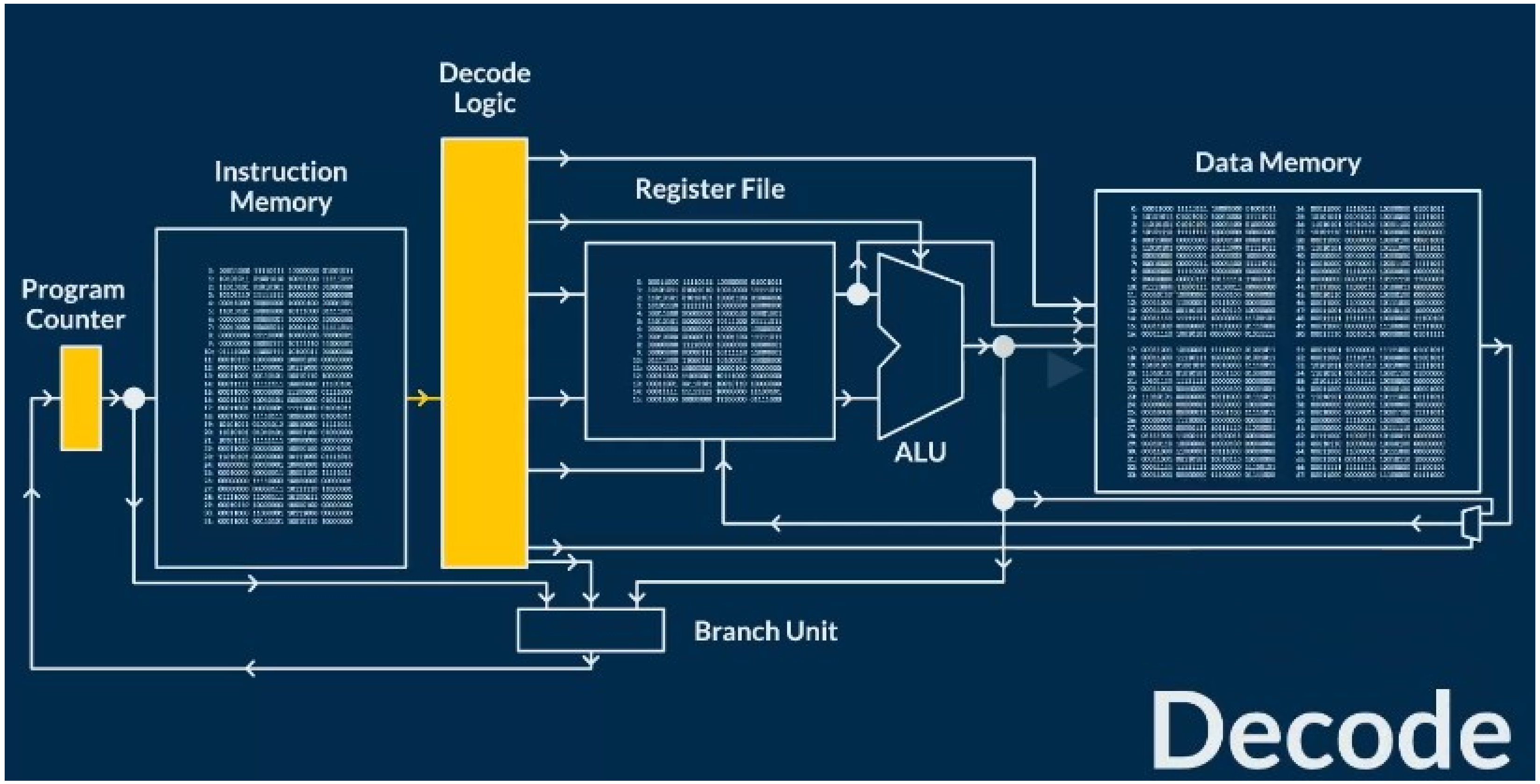
- Processing without pipelining



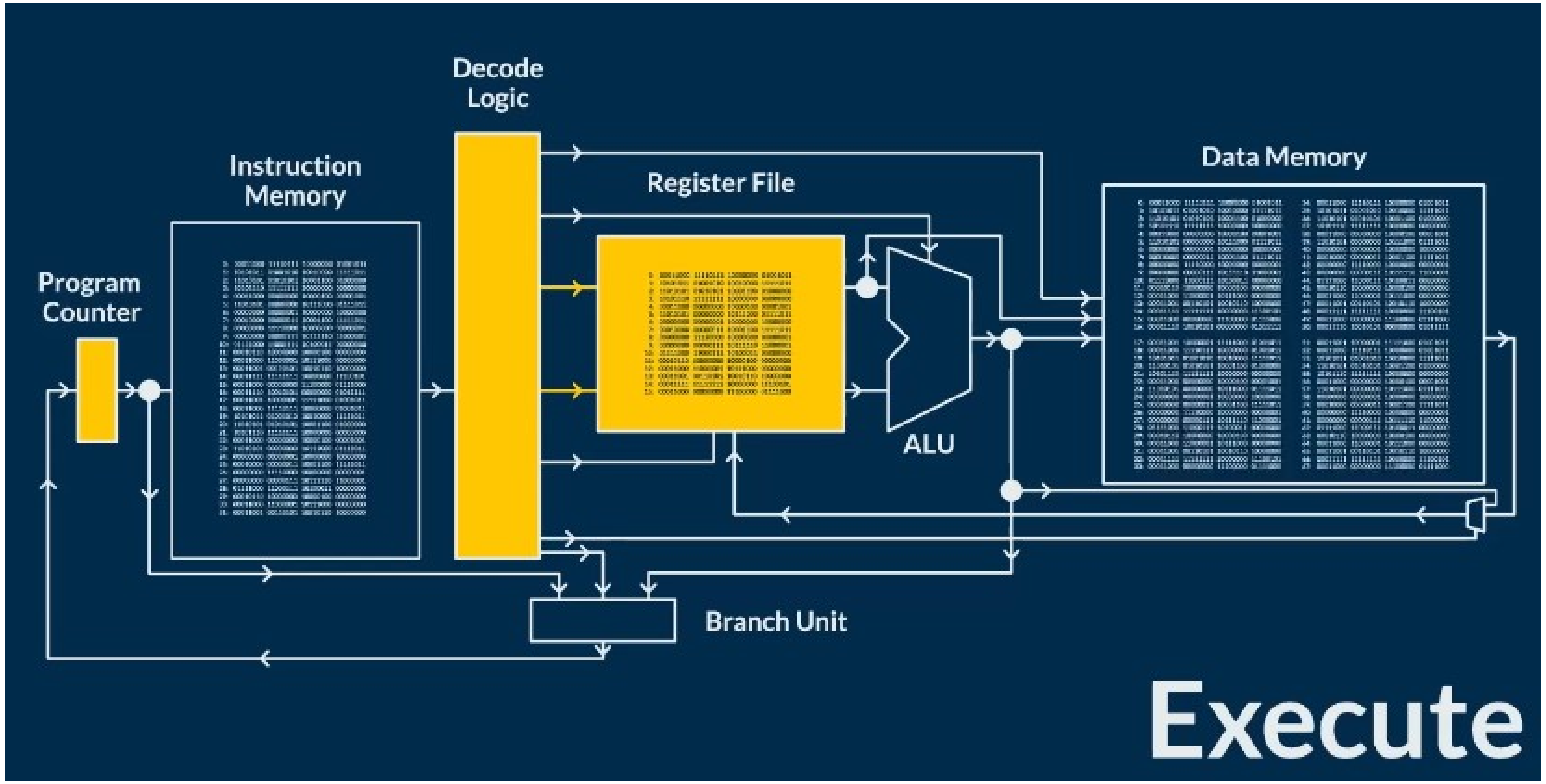
## Process 1 (Fetch)



# Process 2 (Decode)

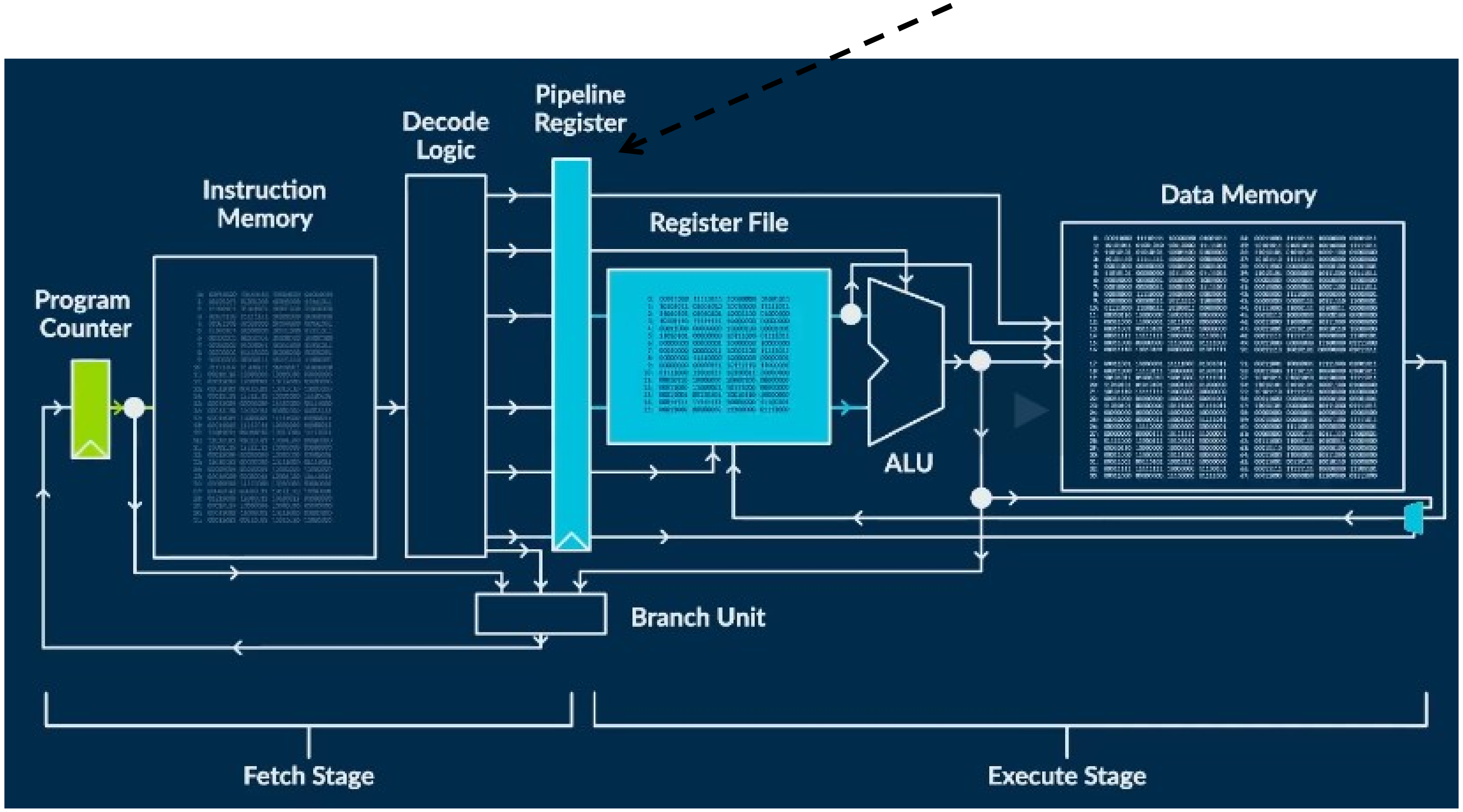


# Process 3 (Execute)

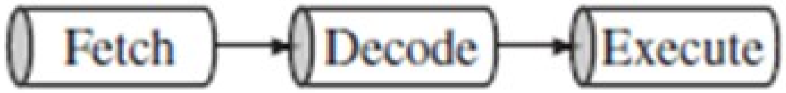


# Pipelining: Two stage

- Using pipeline register (shown in blue)



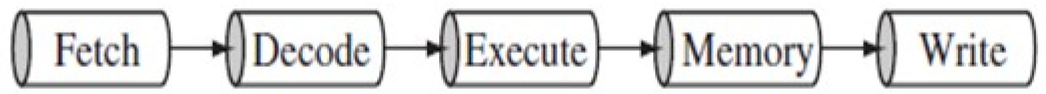
## Example: ARM7 three stage pipeline



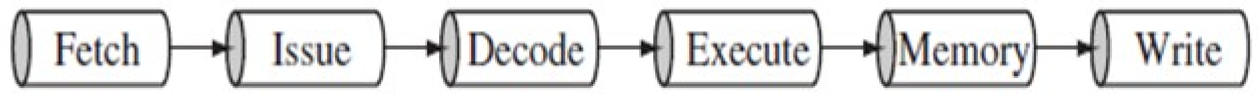
ARM7 Three-stage pipeline.

- *Fetch* loads an instruction from memory.
- *Decode* identifies the instruction to be executed.
- *Execute* processes the instruction and writes the result back to a register.

# Example: ARM9 and ARM10



ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

# Comparing with different pipelines

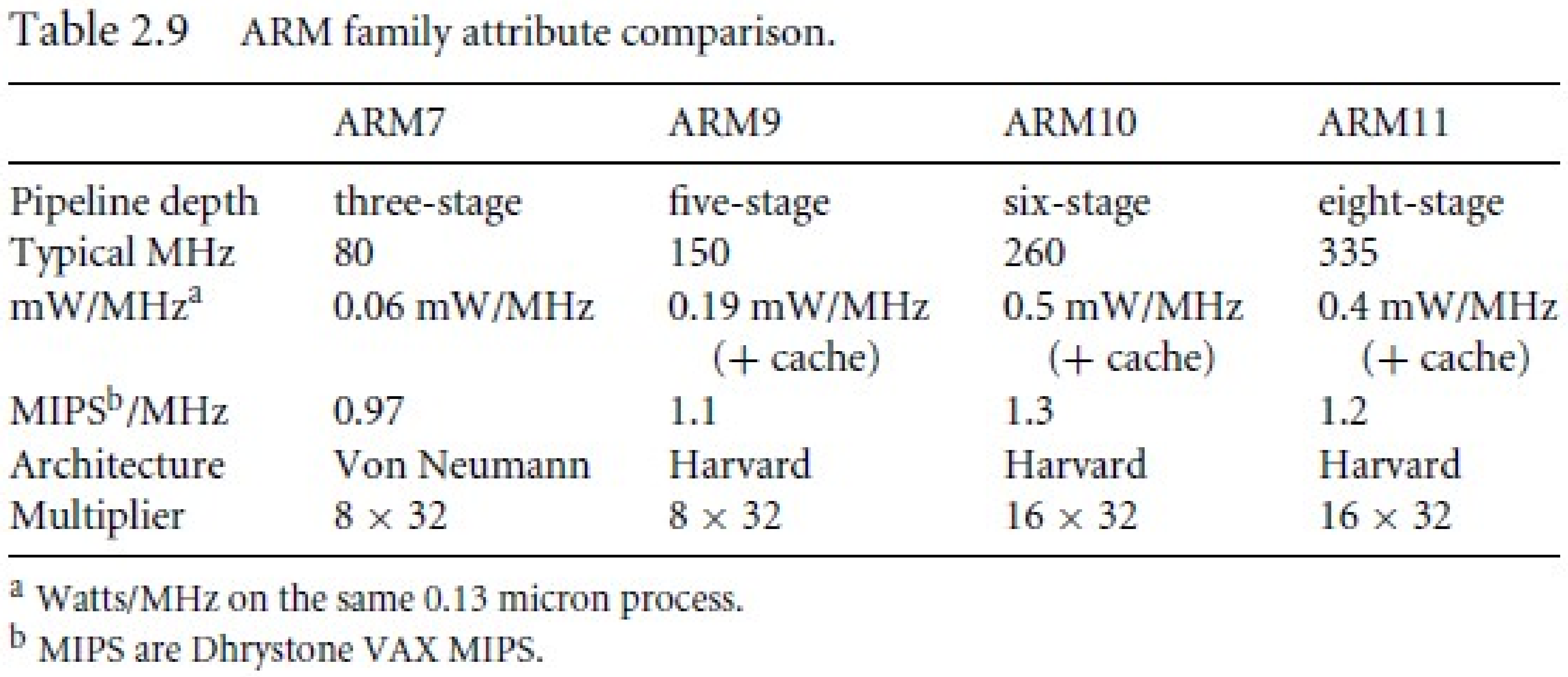


Table 2.9 ARM family attribute comparison.

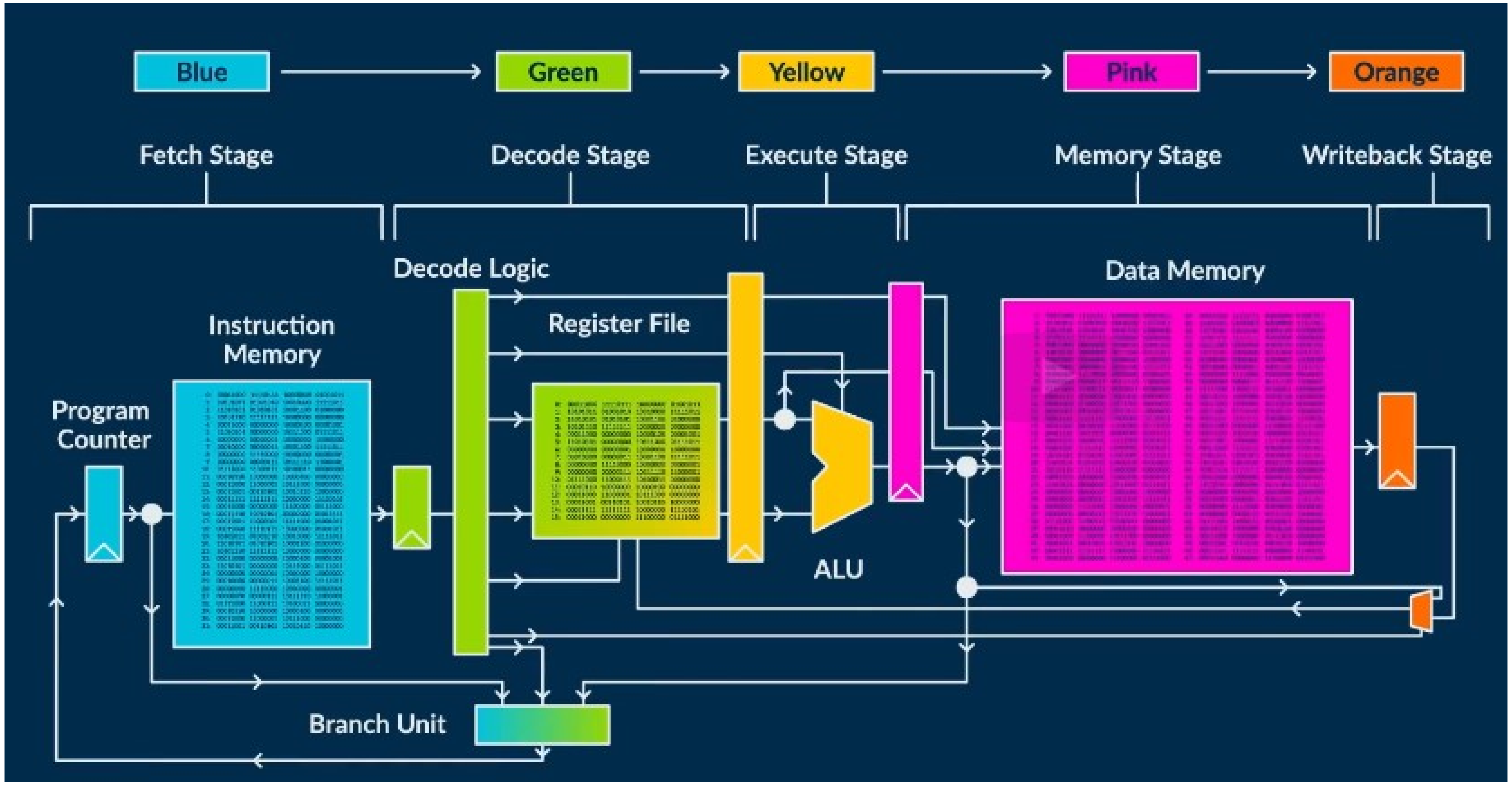
| | ARM7 | ARM9 | ARM10 | ARM11 |
|---|---|---|---|---|
| Pipeline depth | three-stage | five-stage | six-stage | eight-stage |
| Typical MHz | 80 | 150 | 260 | 335 |
| mW/MHz[a] | 0.06 mW/MHz | 0.19 mW/MHz (+ cache) | 0.5 mW/MHz (+ cache) | 0.4 mW/MHz (+ cache) |
| MIPS[b]/MHz | 0.97 | 1.1 | 1.3 | 1.2 |
| Architecture | Von Neumann | Harvard | Harvard | Harvard |
| Multiplier | 8 × 32 | 8 × 32 | 16 × 32 | 16 × 32 |

[a] Watts/MHz on the same 0.13 micron process.

[b] MIPS are Dhrystone VAX MIPS.

# Pipelining: Five stages

- Using four pipeline registers ( ▮ )



## The Reference:

[1] A., Sloss, et al., ARM System Developer's Guide.

[2] ARM documents, www.arm.com