UNIT NO 6

Subject: Processor Architecture
(214451)

Current Trends in Processor Architecture

BY
Prof Anjali Hudedamani
Assistant Professor AIT

RISC

- **Definition of RISC**
  - Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.
- **Evolution/History.**
  - The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC.
- **Certain design features have been characteristic of most RISC processors**
  - (1**) One Cycle Execution** Time. RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called ;
  - (2) **Pipelining.** A technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
  - (3) **Large Number of Registers**. The RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

Difference Between RISC and CISC

S.No	Feature	RISC	CISC
1.	**Instruction Set**	Small and simple set of instructions	Large and complex set of instructions
2.	**Instruction Length**	Typically fixed-length	Variable-length
3.	**Execution Time**	Most instructions execute in **one clock cycle**	Instructions may take **multiple clock cycles**
4.	**Memory Access**	Load/Store architecture (only LOAD & STORE access memory)	Many instructions can access memory directly
5.	**Complexity**	Moved to software (compiler)	More complexity handled in hardware
6.	**Code Size**	Larger (more instructions needed)	Smaller (fewer complex instructions)
7.	**Pipelining**	Easier to implement due to uniform instruction size	More difficult due to varying instruction sizes
8.	**Hardware Requirements**	Simpler hardware, easier to design and scale	More complex hardware
9.	**Examples**	ARM, MIPS, RISC-V	Intel x86, AMD64

Von Neumann vs Harvard Architecture

S.No.	Feature	Von Neumann Architecture	Harvard Architecture
1.	**Memory**	Single memory for **instructions and data**	**Separate memories** for instructions and data
2.	**Bus System**	Single bus for **data and instructions**	Separate buses for **data and instructions**
3.	**Access Speed**	Slower (since instructions and data share the same bus)	Faster (instructions and data can be accessed simultaneously)
4.	**Design Simplicity**	Simpler and cheaper to design	More complex and expensive
5.	**Memory Access Bottleneck**	Prone to **bottleneck** (Von Neumann Bottleneck)	No bottleneck due to separate pathways
6.	**Flexibility**	More flexible, can modify instructions as data	Less flexible, data and instructions are strictly separated
7.	Usage	Used in **general-purpose computers**	Used in **embedded systems, microcontrollers, DSPs**

RISC Philosophy

- The RISC philosophy is implemented with four major design rules:
  - 1**. Instructions**
    - RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction
  - 2. **Pipelines**
    - The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput.
  - 3. **Registers**
    - RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations.
  - 4. **Load-store architecture**
    - The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.

The ARM Design Philosophy

- The **ARM (Advanced RISC Machine)** design philosophy is centered around efficiency, simplicity, and performance-per-watt. It follows the **RISC (Reduced Instruction Set Computing)** model, which emphasizes a streamlined, highly optimized instruction set to achieve greater performance and energy efficiency. Here's a breakdown of the core principles behind ARM's design philosophy:
  - **1. Simplicity Through RISC Architecture**
    - ARM uses a **reduced set of instructions** compared to CISC (like x86).
    - Instructions are simple and execute in a single clock cycle, where possible.
    - This simplicity leads to **faster execution** and easier pipeline optimization.
  - **2. High Performance Per Watt**
    - ARM is optimized for **low power consumption**, which is crucial for mobile and embedded systems.
    - It achieves high performance while using **minimal energy**, which is why ARM dominates smartphones, tablets, and IoT devices.
  - **3. Load/Store Architecture**
    - ARM separates memory access from computation.
    - Only **load** and **store** instructions can access memory; all other operations work on CPU registers.
    - This approach reduces complexity and increases speed.
  - **4. Orthogonality and Uniformity**
    - Instructions are designed to be **consistent and flexible**.
    - Most operations can be performed on any register, and many instructions support optional features like condition codes.
    - This leads to a **cleaner instruction set** and easier compiler optimization.
  - **5. Scalability and Modularity**
    - ARM is designed to be **highly scalable**, from tiny microcontrollers (like Cortex-M) to powerful CPUs (like Cortex-A).
    - The architecture is **modular**, allowing vendors to choose only the components (MMU, FPU, etc.) they need.
  - **6. Licensing and Customization**
    - ARM Holdings licenses the architecture rather than manufacturing chips.
    - Partners like Apple, Qualcomm, and Samsung can **customize** ARM cores or build their own based on ARM's ISA.
    - This results in a wide variety of implementations optimized for specific needs

ARM Design Philosophy

- The ARM core is not a pure RISC architecture because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far.
- In today's systems the key is not raw processor speed but total effective system performance and power consumption.
- Instruction Set for Embedded Systems:
  - The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:
  - Variable cycle execution for certain instructions—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses, which increases performance since sequential memory accesses are often faster than random accesses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.
  - Inline barrel shifter leading to more complex instructions—The inline barrel shifter is a hardware component that pre processes one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density.
  - Thumb 16-bit instruction set—ARM enhanced the processor corecalled Thumb that permits the ARM core to execute either 16- or 32-bit instructions.

ARM Design Philosophy

- The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.
  - Conditional execution—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.
  - Enhanced instructions—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations and saturation. These instructions allow a faster-performing ARM processor in some cases to replace the traditional combinations of a processor plus a DSP.
- These additional features have made the ARM processor one of the most commonly used 32-bit embedded processor cores.
- Many of the top semiconductor companies around the world produce products based around the ARM processor.

Main features of ARM Processor

- **Multiprocessing Systems –**
  ARM processors are designed so that they can be used in cases of multiprocessing systems where more than one processors are used to process information. First AMP processor introduced by name of ARMv6K had ability to support 4 CPUs along with its hardware.

- **Tightly Coupled Memory –**
  Memory of ARM processors is tightly coupled. This has very fast response time. It has low latency (quick response) that can also be used in cases of cache memory being unpredictable.

- **Memory Management –**
  ARM processor has management section. This includes Memory Management Unit and Memory Protection Unit. These management systems become very important in managing memory efficiently.

- **Thumb-2 Technology –**
  Thumb-2 Technology was introduced in 2003 and was used to create variable length instruction set. It extends 16-bit instructions of initial Thumb technology to 32-bit instructions. It has better performance than previously used Thumb technology.

Main features of ARM Processor

- **One cycle execution time –**
  ARM processor is optimized for each instruction on CPU. Each instruction is of fixed length that allows time for fetching future instructions before executing present instruction. ARM has CPI (Clock Per Instruction) of one cycle.

- **Pipelining –**
  Processing of instructions is done in parallel using pipelines. Instructions are broken down and decoded in one pipeline stage. The pipeline advances one step at a time to increase throughput (rate of processing).

- **Large number of registers –**
  Large number of registers are used in ARM processor to prevent large amount of memory interactions. Registers contain data and addresses. These act as local memory store for all operations.

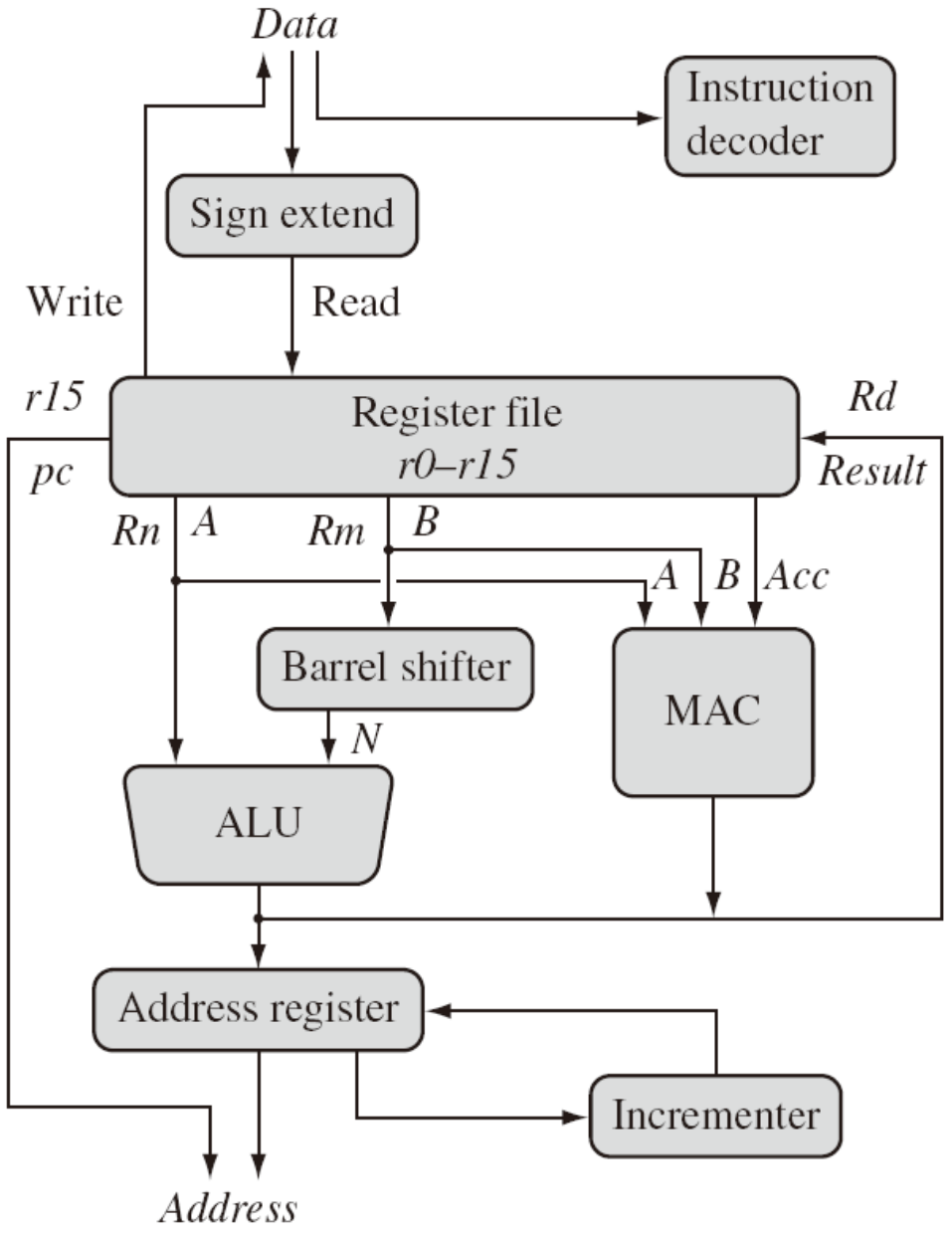
ARM powered products

ARM processors

- A simple but powerful design
- A whole family of designs sharing similar design principles and a common instruction set

ARM Core Dataflow Model

- The **ARM Core Dataflow Model** describes how data moves through the ARM processor core during instruction execution.
- It focuses on the paths data takes between various components such as the registers, ALU (Arithmetic Logic Unit), memory, and other units inside the processor.



ARM Core Data Flow

1. **Instruction Fetch Unit**
   - Fetches instructions from memory (usually from the instruction cache).
   - Passes them to the decode unit.
2. **Instruction Decode Unit**
   - Decodes instructions to determine operation type and involved operands.
   - Sets control signals for the rest of the datapath.
3. **Register File**
   - Contains general-purpose registers (e.g., R0–R15 for ARMv7).
   - Source operands are read from here.
   - Destination operands are written back here.
4. **ALU / Execution Units**
   - Performs arithmetic or logical operations.
   - Also includes shifters, multipliers, and sometimes a floating-point unit (FPU).
   - Takes inputs from the register file or immediate values.
   - Sends results back to registers or memory.
5. **Memory Access Unit**
   - Handles load/store instructions.
   - Accesses memory for reading (load) or writing (store) via the data bus.
   - Includes cache interfaces and MMU (Memory Management Unit).
6. **Write-Back Unit**
   - Takes the result from ALU/memory and writes it back into the destination register.
7. **Control Unit**
   - Coordinates all the above components.
   - Generates the necessary control signals based on the instruction type and processor state.

Typical Dataflow in an ARM Instruction

Typical Dataflow in an ARM Instruction (e.g., ADD R1, R2, R3):
1.Fetch: ADD R1, R2, R3 is fetched from memory.
2.Decode: Control unit identifies it's an ADD instruction.
3.Read Operands: Values from R2 and R3 are read from the register file.
4.Execute: ALU adds the values.
5.Write-Back: Result is written to R1 in the register file.

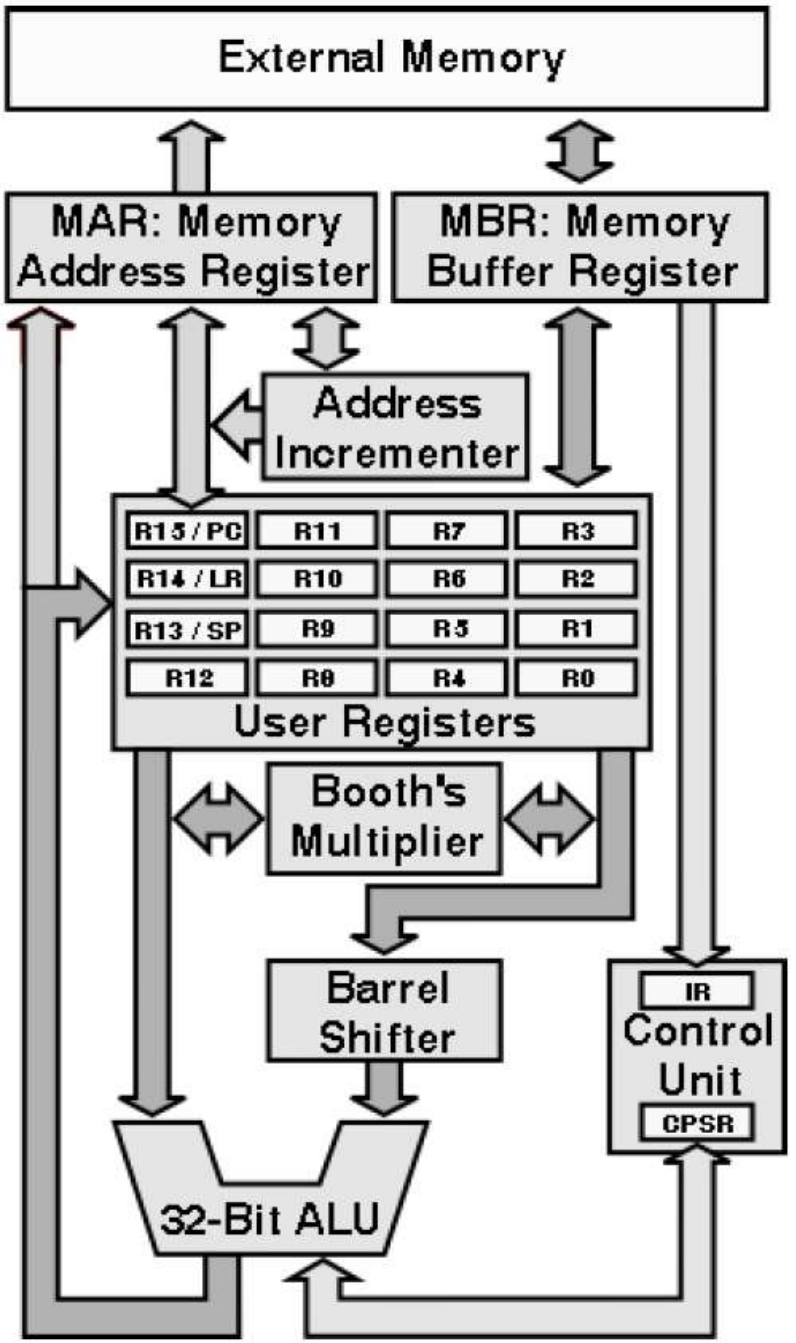
- **Pipeline Stages and Dataflow (for ARM Cortex series, like Cortex-M or Cortex-A):**
  - ARM cores are pipelined, which means multiple instructions are processed simultaneously at different stages:
- **5-stage pipeline (ARM7TDMI):**
  - Fetch → Decode → Execute → Memory → Write-back
- **8-stage or deeper (Cortex-A):**
  - Includes instruction issue, rename, multiple execution units, etc

ARM Programmer's Model

- This is essentially what a **software developer (especially embedded devs)** needs to know when writing code for an ARM core. It includes things like **registers, modes, memory, and exceptions** — basically, **how the processor looks and behaves from the programmer's perspective**.
- The **Programmer's Model** includes:
  - **Registers** (general-purpose, special-purpose)
  - **Processor Modes**
  - **Program Counter (PC) and Stack Pointer (SP)**
  - **Status Registers (CPSR/SPSR)**
  - **Memory Model (flat memory, MMU)**
  - **Exception Handling**
  - **Instruction Set (ARM, Thumb, AArch64)**

ARM archit

- Load/store architecture
- A large array of uniform registers
- Fixed-length 32-bit instructions
- 3-address instructions



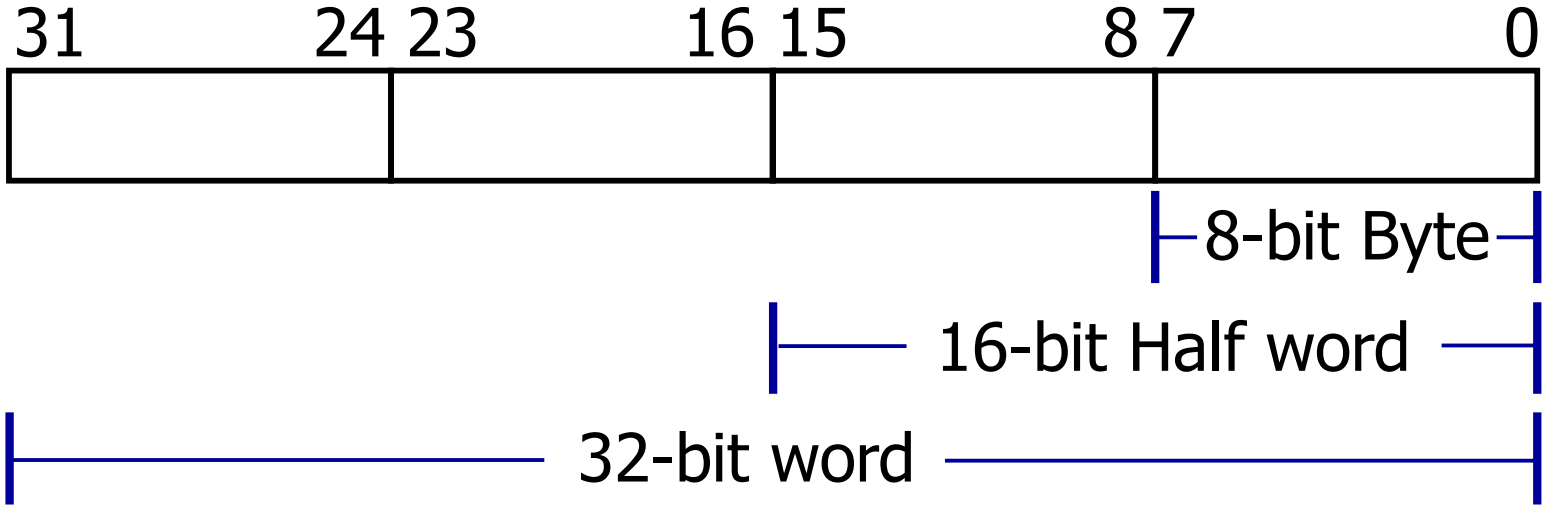
Registers

- General purpose registers hold either data or an address.
- All registers are 32-bit in size.
- 18 active registers
- 16 data registers – r0 to r15
- 2 process status registers
  - CPSR – Current Program Status Register
  - SPSR – Saved Program Status Register
- r13, r14, r15 have special functions
- R13 – stack pointer (sp) and stores the head of the stack in the current processor mode
- R14 – link register (lr) where the core puts the return address whenever it calls a subroutine.
- R15 – programme counter (pc) and contains the address of the next instruction to be fetched by the processor.
- r13 and r14 can also be used as general purpose register as these registers are banked during a processor mode change.
- The registers r0 to r13 are orthogonal. Any instruction that you can apply to r0, you can equally apply to other registers.
- There are instructions that treat r14 and r15 in a special way.

Registers

- Only 16 registers are visible to a specific mode. A mode could access
  - A particular set of r0-r12- General use
  - r13 (sp, Stack Pointer)
  - r14 (lr, Link Register)- holds return address for subroutines
  - r15 (pc, Program Counter)
  - Current program status register (CPSR)
  - The uses of r0-r13 are orthogonal

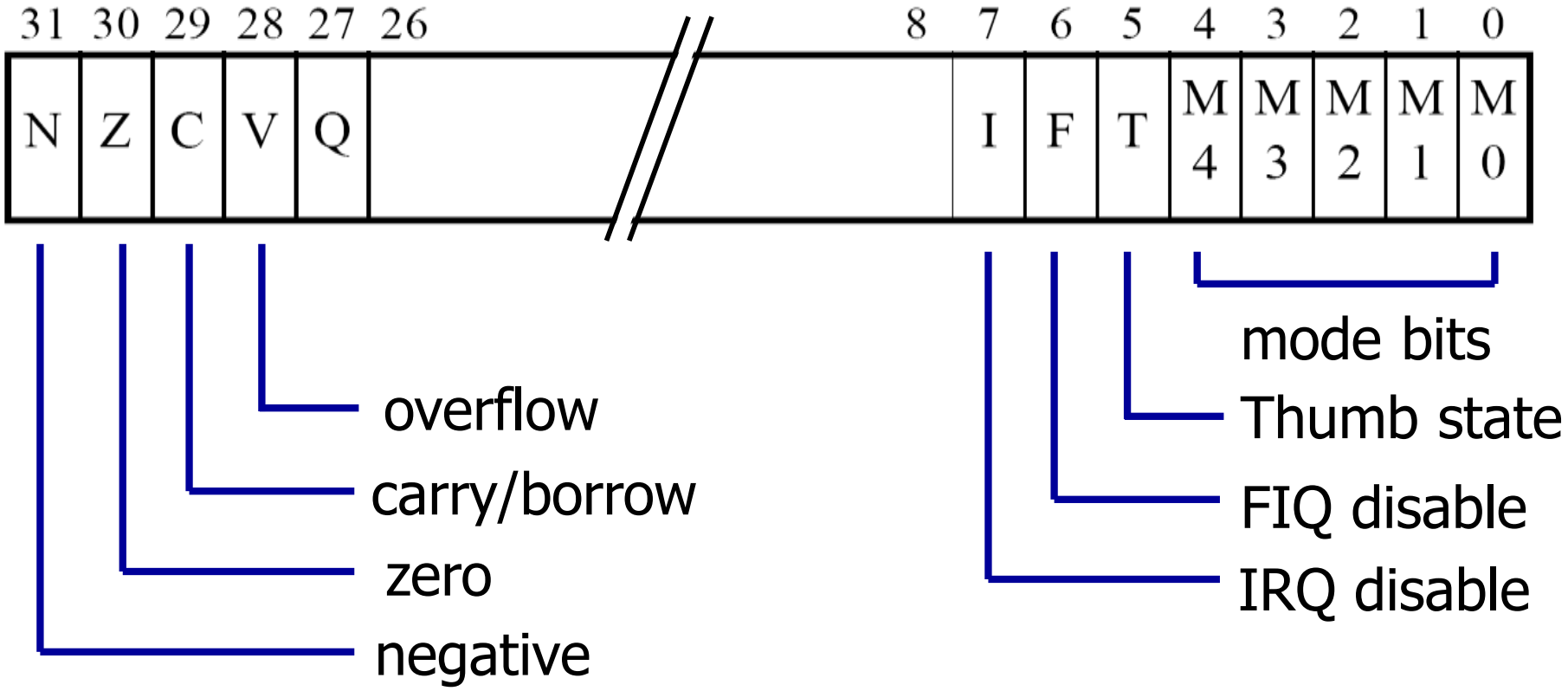
General-Purpose Registers



- 6 data types (signed/unsigned)
- All ARM operations are 32-bit. Shorter data types are only supported by data transfer operations.

Status Registers

Current Program Status Register (CPSR):



- Condition flags: N (negative), Z (zero), C (carry), V (overflow)
- Control bits: processor mode, interrupt disables (I, F), Thumb bit (T)

SPSR (Saved Program Status Register):

- Used to save the CPSR when handling exceptions.

R13, R14, and R15 – Special Purpose Registers

Register	Name	Purpose
R13	**SP (Stack Pointer)**	Points to the top of the stack **Used for stack operations** – function calls, interrupts, local variables. Different **processor modes** (like IRQ, FIQ, Supervisor) have **banked R13**, meaning each mode has its **own stack pointer**. The stack grows **downward** (from high memory to low memory). Essential for function calls, context switching, and exception handling.
R14	**LR (Link Register)**	Stores return address for functions **Holds the return address** when a subroutine or function is called. When you use BL (Branch with Link), the address of the **next instruction** is stored in R14. You return from a subroutine by copying R14 into PC: assembly MOV PC, LR Like R13, it can be **banked** for different modes to allow nested exceptions or interrupts.
R15	**PC (Program Counter)**	Holds address of current/next instruction Always points to the **current instruction** being executed. In ARM, it may appear to be ahead by 8 bytes (because of pipelining). In Thumb mode, it's ahead by 4 bytes. You can jump to a different part of the code by **writing a new value to R15**, like: assembly MOV PC, R0 ; jump to address in R0

R13, R14, and R15 – Special Purpose Registers

These three registers form the **core of control flow** in ARM:

- R13 manages where local data is stored during function calls.
- R14 handles return addresses automatically with BL, BX LR, etc.
- R15 controls program execution – changing it = jumping.

They are critical for implementing **call stacks**, **function calls, interrupts, and multitasking.**

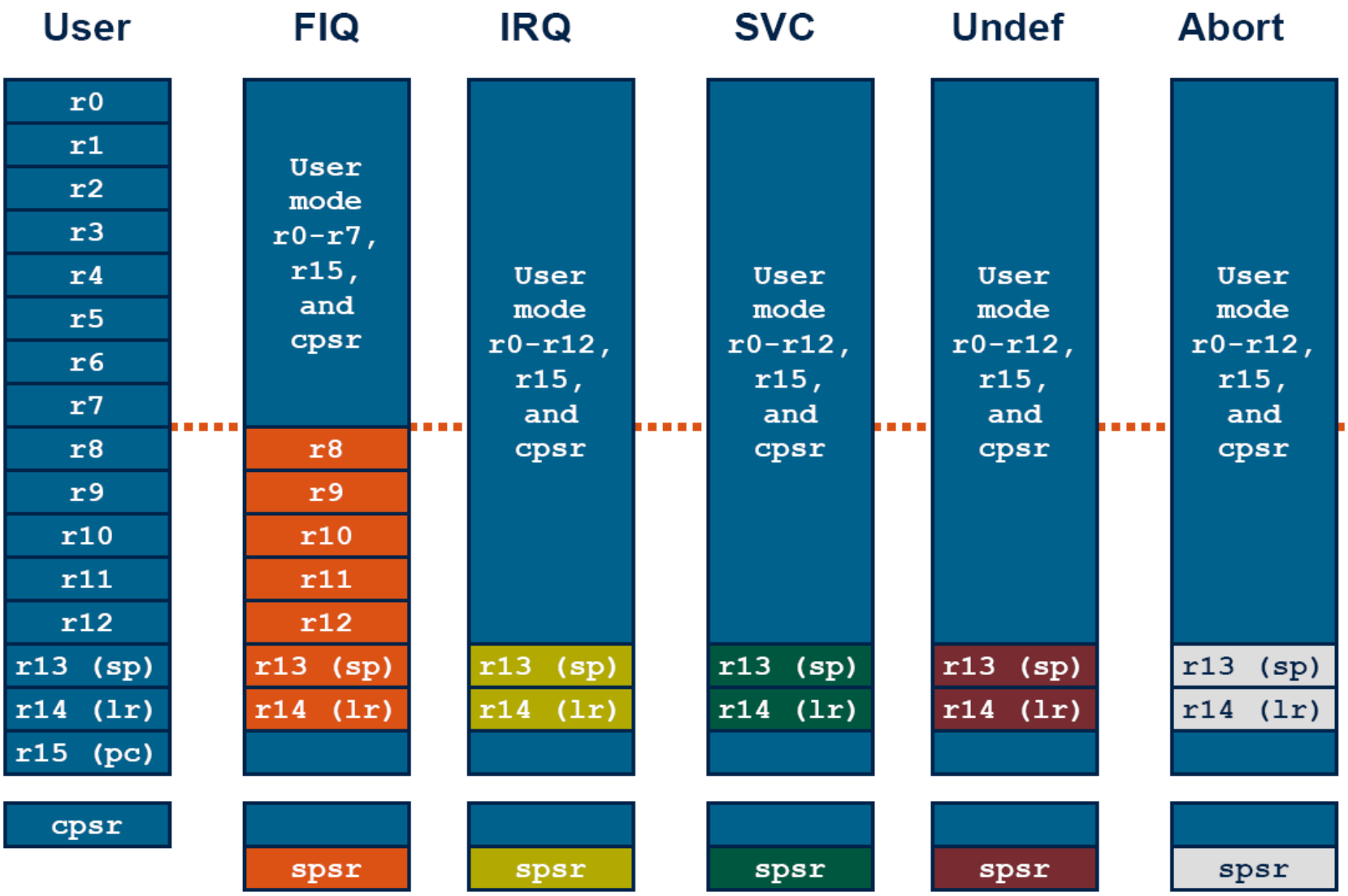
Memory Model

- **Flat address space**: 32-bit (AArch32) or 64-bit (AArch64) addressable.
- Can access:
  - Code
  - Data
  - Stack
  - Memory-mapped peripherals
- Memory access types:
  - Byte, halfword (16-bit), word (32-bit), doubleword (64-bit)
  - ARMv8 introduces **virtual memory and MMU (Memory Management Unit)**

Processor modes

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks

Processor Modes: Register Organization



Each mode may bank certain registers, meaning it uses a private copy (like SP, LR, SPSR).

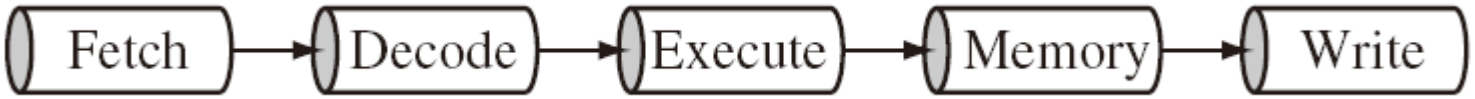
Instruction Sets

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

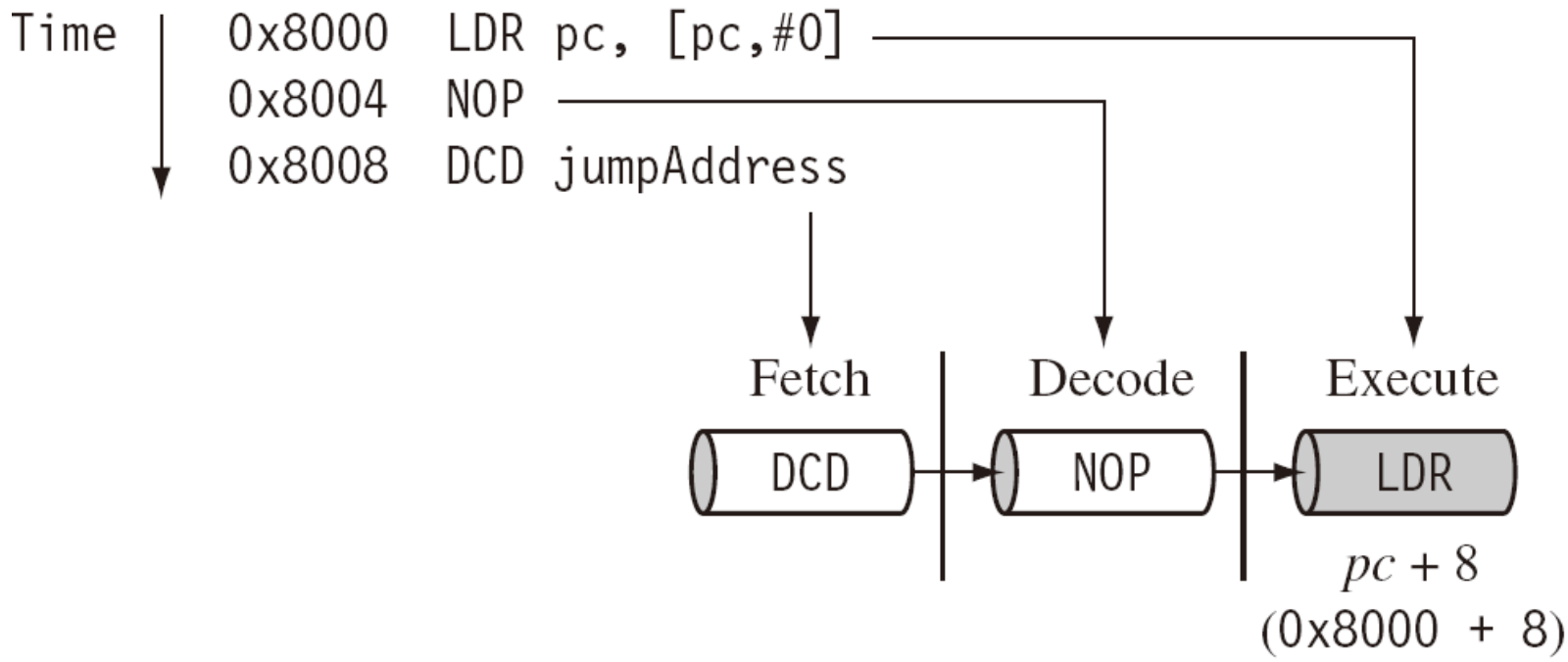
	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

Pipeline

ARM7 

ARM9 

In execution, pc always moves 8 bytes ahead



Pipeline

- Execution of a branch or direct modification of pc causes ARM core to flush its pipeline
- ARM10 starts to use branch prediction
- An instruction in the execution stage will complete even though an interrupt has been raised. Other instructions in the pipeline are abondond.

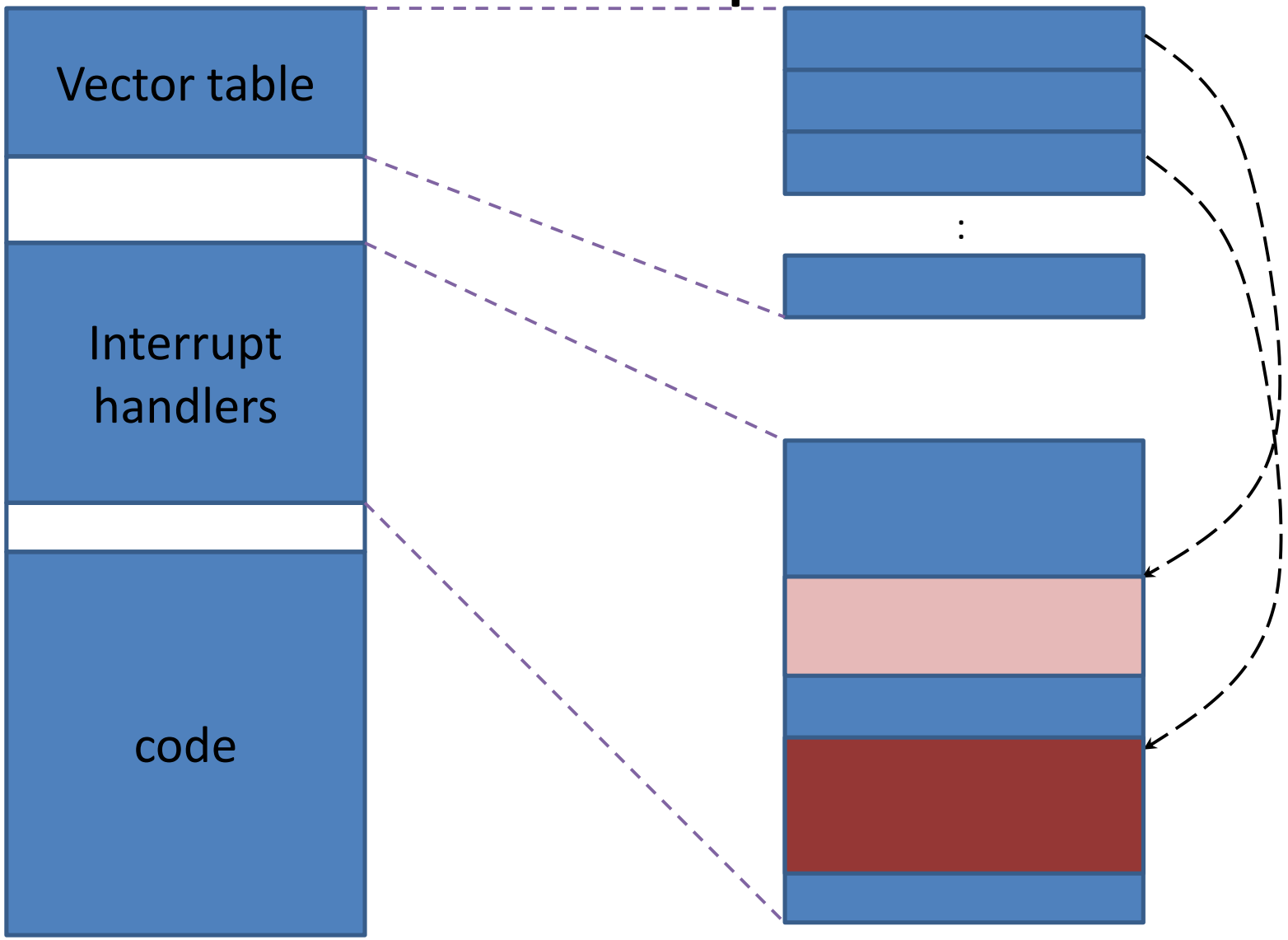
Memory Model

- **Flat address space**: 32-bit (AArch32) or 64-bit (AArch64) addressable.
- Can access:
  - Code
  - Data
  - Stack
  - Memory-mapped peripherals
- Memory access types:
  - Byte, halfword (16-bit), word (32-bit), doubleword (64-bit)
  - ARMv8 introduces **virtual memory and MMU (Memory Management Unit)**

Exception Handling

- When an interrupt or fault occurs:
- The processor:
  - Switches to a specific mode (IRQ, FIQ, etc.)
  - Saves context (PC, CPSR)
  - Jumps to an exception vector
- ARM exceptions:
  - Reset
  - Undefined instruction
  - Software interrupt (SWI/SVC)
  - Prefetch/Data abort
  - IRQ/FIQ

Interrupts



Interrupts

Exception/interrupt	Shorthand	Address
Reset	RESET	0x00000000
Undefined instruction	UNDEF	0x00000004
Software interrupt	SWI	0x00000008
Prefetch abort	PABT	0x0000000c
Data abort	DABT	0x00000010
Reserved	—	0x00000014
Interrupt request	IRQ	0x00000018
Fast interrupt request	FIQ	0x0000001c

Naming ARM

- ARMxyzTDMIEJFS
  - x: series
  - y: MMU
  - z: cache
  - T: Thumb
  - D: debugger
  - M: Multiplier
  - I: EmbeddedICE (built-in debugger hardware)
  - E: Enhanced instruction
  - J: Jazelle (JVM)
  - F: Floating-point
  - S: Synthesizible version (source code version for EDA tools)

Popular ARM Architectures

- **ARM7**
  - **Simplest of the three.**
  - No cache, simple pipeline.
  - Very power-efficient — still found in **low-cost microcontrollers** (e.g., LPC2000 series).
  - Doesn't support Linux without heavy modifications (due to no MMU).
  - 3 pipeline stages (fetch/decode/execute)
  - High code density/low power consumption
  - One of the most used ARM-version (for low-end systems)
  - All ARM cores after ARM7TDMI include TDMI even if they do not include TDMI in their labels
- **ARM9**
  - **Better performance** due to 5-stage pipeline (fetch/decode/execute/memory/write).
  - **Separate instruction and data caches**, which reduces memory bottlenecks.
  - Popular in **early smartphones** and embedded Linux systems (e.g., routers).
  - MMU support enables **full Linux support**.
  - Compatible with ARM7
  - Separate instruction and data cache
- **ARM11**
  - Big jump in **performance and features**.
  - Supports **ARMv6**: SIMD instructions, media processing, and **TrustZone** for security.
  - Deeper pipeline = higher clock speeds.
  - Was used in early **smartphones** (e.g., **first-gen iPhone, Nokia N95**).
  - Last generation before Cortex-A took over.

ARM Family Comparison

ARM family attribute comparison.

year	1995	1997	1999	2003
	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.

ARM Bus Technology

- Embedded systems use different bus technologies. The Peripheral Component Interconnect (PCI) bus connects devices such as video card and disk controllers to the X 86 processor buses. This is called External or off chip bus technology.
- Embedded devices use an on-chip bus that is internal to the chip and allows different peripheral devices to be inter-connected with an ARM core.
- There are two different types of devices connected to the bus
  - 1. Bus Master
  - 2. Bus Slave

- **Bus Master: A logical device capable of initiating a data transfer with another device across the same bus (ARM processor core is a bus Master).**
- **Bus Slave: A logical device capable only of responding to a transfer request from a bus master device (Peripherals are bus slaves)**
- Generally a Bus has two architecture levels
  - **Physical lever: Which covers electrical characteristics a bus width (16, 32, 64 bus).**
  - **Protocol level: which deals with protocol?**
- NOTE: - ARM is primarily a design company. It seldom implements the electrical characteristics of the bus, but it routinely specifies the bus protocol

AMBA (Advanced Microcontroller Bus Architecture) Bus protocol

- The **Advanced Microcontroller Bus Architecture (AMBA)** is a set of protocols developed by ARM for interconnecting various components in a microcontroller or microprocessor system. The AMBA bus protocol is widely used in ARM-based systems to facilitate communication between different components such as the CPU, memory, and peripherals AMBA Bus was introduced in 1996 and has been widely adopted as the On Chip bus architecture used for ARM processors.
- AMBA AHB (Advanced High-performance Bus):
  - AHB is used for high-performance and high-throughput data transfers between the CPU, memory, and peripherals.
  - It's a system bus designed to support high-speed operation and low-latency data transfer.
  - AHB is a pipelined bus that supports multiple data transactions simultaneously, making it suitable for high-performance systems.
- AMBA APB (Advanced Peripheral Bus):
  - APB is a simpler bus designed for connecting low-speed peripherals to the system.
  - It is a low-power, low-performance bus that is ideal for peripherals that do not require high bandwidth.
  - APB is often used for simple devices like timers, UARTs, and GPIOs.
- AMBA AXI (Advanced eXtensible Interface):
  - AXI is the most advanced protocol in the AMBA family, designed to provide high-bandwidth, low-latency communication.
  - AXI supports high-performance transactions and includes features like out-of-order transactions, multiple data channels, and burst transfers.
  - It is designed for systems that require a high level of data throughput, such as in multimedia or networking applications.

Applications of AMBA

- **ARM-based SoCs (System on Chips)**:
  - AMBA is widely used in ARM-based systems, including mobile devices, embedded systems, and networking equipment.
- **High-performance computing**:
  - AXI is often used in systems requiring high throughput, such as in data centers and multimedia processing systems.
- **Embedded systems**:
  - AHB and APB are typically used in embedded systems where low power consumption and efficient peripheral communication are required.
- In summary, the AMBA bus protocol is crucial for managing the communication between components in ARM-based systems, offering a flexible, scalable, and efficient method of interconnecting devices. Its different bus protocols (AHB, APB, and AXI) cater to varying performance and power requirements of different system components