

ECE3849  
D-Term 2021

Real Time Embedded Systems

Module 5 Part 3

# Module 5 Part 3 Overview

- ARM
  - ARM business model and advantages.
  - ARM CPU architecture.
  - Instruction execution and pipelining.
  - ARMv7-CPU Registers.
  - Memory Organization.
  - Endianness.

# ARM Holdings Business Model

- **ARM Holdings Ltd. in Cambridge UK**
  - Designs components of ARM processors for others to build.
  - They do not fabricate or sell chips.
  - They license the Intellectual property (IP) for the components and the instruction set.
  - The IP comes in three different forms,
    - Synthesizable cores (Verilog).
    - Black box cores (fully laid out for use in ASICs).
    - Architecture licenses (License use of architecture, write your own HDL to customize as necessary) .
- **Customers build systems around these designs and customize the IP to optimize their system.**
  - When an application uses ARM it simply means it is using the ARM IP and instruction set.
  - No generalized performance is given as each customer implementation will be a little different.
  - This is contrary to the x86 model where actual devices are sold and device performance is often well benchmarked.

# Types of Arm Processors

- **A-Profile**
  - High-Performance applications and operating systems.
  - Servers, mobile phones, automotive applications, IoT Gateways.
- **R-Profile**
  - Real-Time, High-Performance applications
  - Used when deterministic response is required.
  - Medical equipment, vehicle steering and breaking, media players, networking and data storage.
- **M-Profile**
  - Discrete Processing and Microcontrollers.
  - Optimized for size and low power.
  - Small sensors, communication modules, smart home products.
- **The Tiva C Series devices, integrate a Cortex-M4F processor.**

# The ARM Advantage

- ARM is a 32-bit RISC architecture
  - RISC = Reduced instruction set computer.
- Reduced Instruction Set requires fewer transistors
  - Designs have a smaller die -> less expensive, smaller size.
  - Lower power allowing for mobility and smaller heat sinking solutions.
  - Makes them popular for embedded systems.
  - High code density (less memory needed to store program).
- Business model
  - Low-cost, flexible licensing.
  - Large, mature software ecosystem.
  - Scalable: High and lower performance and price options.
    - Low-end ARM cost comparable to other microcontrollers.
    - High-end ARM performance similar to x86.
  - Highly customizable to the designs specific needs.

# ARMv7 Implementations

- Large CPU register set.
- Load-store architecture.
  - Data is loaded into the CPU registers. (load instruction).
  - Operated on by instruction.
  - Written back to memory (store instruction).
- Instruction Length
  - Fixed length 32-bit instructions (originally).
  - Thumb-2 instruction sets inter-mixes 16-bit long instructions with 32-bit long instructions.
    - Reduces the size of your program / increases code density.
- 3-address instructions.
  - 2 input addresses and 1 output address.
  - Example:  $r1 = r2 + r3$ .
- Most instructions can be completed one per cycle
- In special cases, multiple operations can be executed in one instruction.
  - A Load or store to multiple registers in one instruction.
  - Shift or rotate and then execute arithmetic or logic operation.
    - Example shift by 4 and add with second number can be done in a single instruction.
  - Can check condition execution status and perform an operation based on result in a single instruction.

# CPU Specific Code Optimization

- Optimized code will minimize latency and execution time.
  - For standard performance application, compilers will take the C code, optimize it to the specific CPU and generate the instructions for you.
  - For critical real-time operation, it maybe required to write instruction level / assembly code yourself.
    - Writing in assembly optimizes the code to your specific application and minimize execution time.
    - It provides the coder more control over what is implemented.
- To write assembly code, you need to know the specific instruction set and understand the CPU architecture.
  - CPU architecture includes all the functional features exposed to the software.

# Instruction Execution

## • Instruction execution

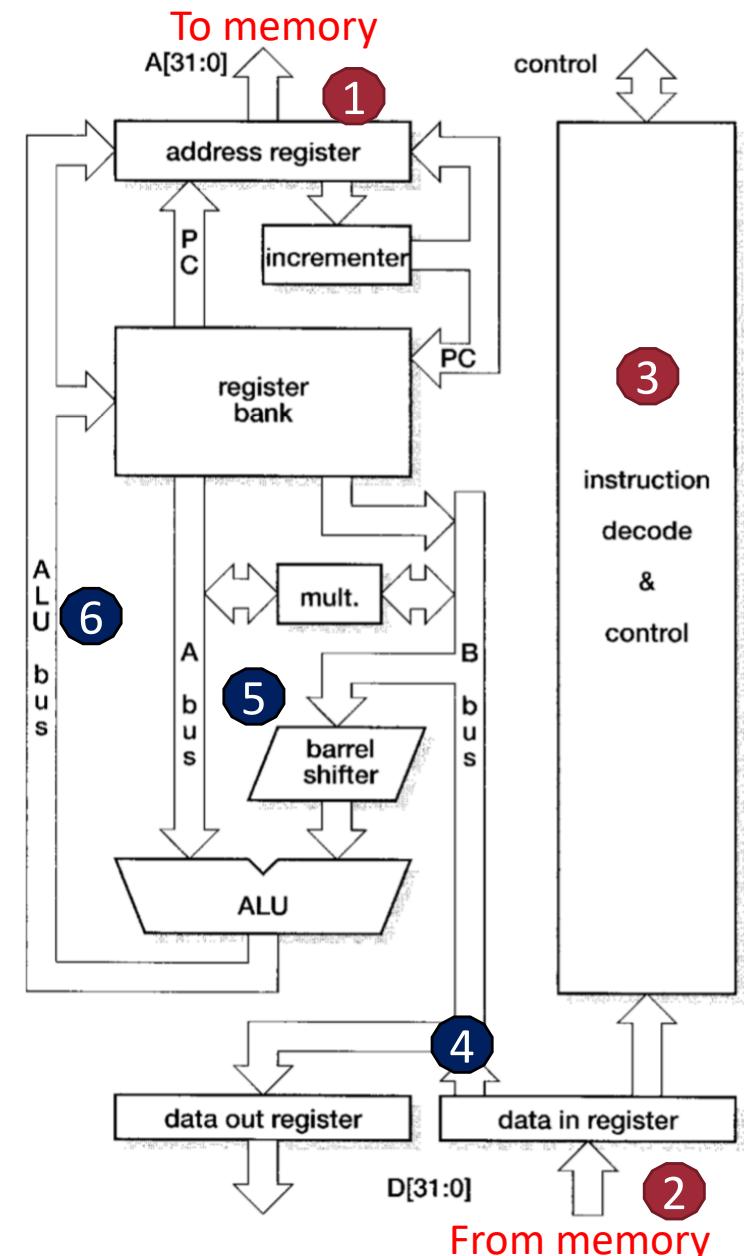
- The program counter (PC) contains the address of the instruction being executed.
  - It initializes to the start of your program and increments as each instruction is executed.
  - There are 4 bytes in an instruction, so PC increments by 4.

- 1 The instruction is read from memory (fetched).
- 2 The instruction is returned to the instruction decode and control block.
- 3 The control output is used in the execution stage of the instruction.

## • Once the instruction is decoded, the instruction is executed.

- To execute an instruction its arguments must be stored in the register bank. Example  $c = a + b$ .
    - Arguments a and b must be stored in the register bank to execute the instruction.
- 4 Prior to issuing the add instruction, load instruction(s) must be performed to fetch the data from memory
  - 5 When the add instruction is executed, the arguments are placed on the A and/or B bus and the ALU performs the add.
  - 6 The result is placed on the ALU bus to be stored in the register bank.

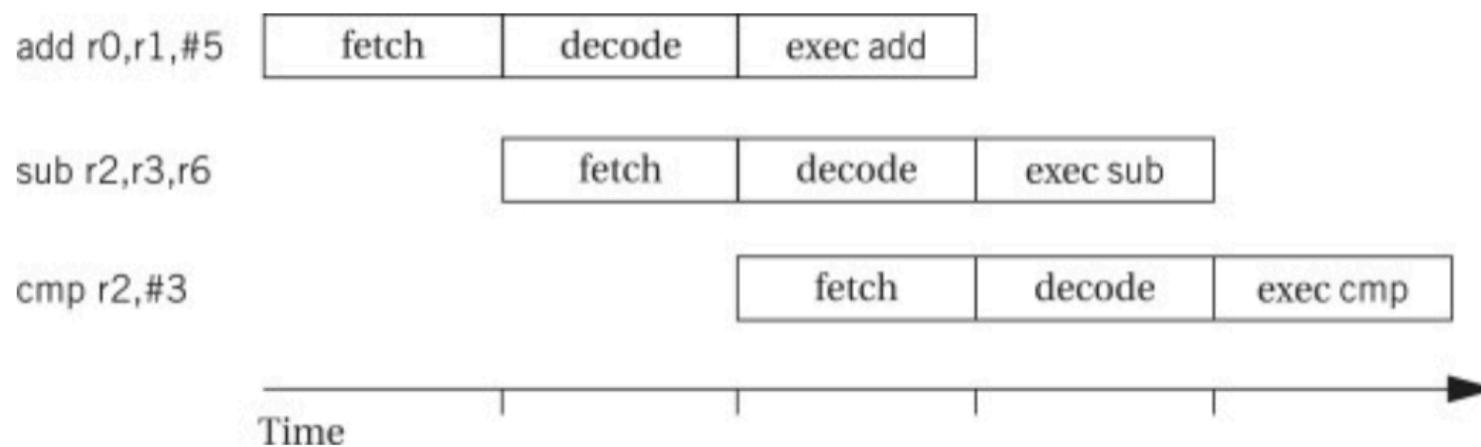
Example: Early ARM CPU block diagram (ARM7).



from S. Furber, "ARM system-on-chip architecture"

# Instruction Pipelining

- There are three steps to the instruction processing.
  - Fetching the instruction from memory.
  - Decoding the instruction.
  - Executing the instruction.
- For single cycle execution of instructions, a 3 stage pipeline is implemented.
  - This allows three instructions to be processed concurrently.
    - The latency of the instruction is 3 clock cycles, but once the pipeline is full an instruction is completed every cycle.
  - Below shows an example.
    - The add instruction is executing.
    - The sub function is decoding.
    - The cmp function is being fetched.



# ARMv7-M CPU Registers

- The CPU contains its own local memory in the form of 32-bit registers.
  - Registers are not part of the the memory address space.
  - Local variables can be stored in registers.
  - There are 16 numbered registers, r0 – r15, and a few special purpose registers.
- r0 – r12 are general purpose registers for temporary storage.
  - Mostly used for integer data and memory addresses.
  - r0-r12 are accessible by all 32-bit instructions.
  - r8-r12 are not accessible by all 16-bit instructions.
- r13 is the stack pointer (SP)
  - The stack stores local variables, function arguments and CPU context.
  - One of two stack pointers is visible at a time depending on the mode of the CPU.
    - MSP - main stack pointer (RTOS and ISRs).
    - PSP – process stack pointer (application / task code).
- r14 is the link register (LR)
  - Holds the return address of a function upon entry.
  - Can be used as a general purpose register if the return address is saved on the stack.
- r15 is the program counter (PC)
  - Indicates where the executing instruction is in memory.
  - Writing to PC results in a branch.

# Special Purpose Registers

- The Program Status Register (PSR) functionality is split between several registers, APSR, IPSR and EPSR.
- **APSR – Application Program Status Register**
  - Contains conditional flags to allow conditional execution of instructions.
  - Based on the result of an arithmetic, logic, or shift operation, a flag will be set if a given condition is met.
  - There are five flags that can be set.
    - N = negative flag
    - Z = zero flag
    - C = Carry flag (unsigned overflow on add, last bit shifted out)
    - V = Signed overflow flag
    - Q = Sticky saturation flag ( not used commonly)
- **IPSR – Interrupt Program Status Register**
  - Interrupt number when executing an ISR
- **EPSR – Execution Program Status Register**
  - Indicates which instruction set you are using.
- **PRIMASK, FAULTMASK, BASEPRI registers**
  - Allow priority boosting during exception conditions.
- **CONTROL - Control bits.**
  - Selects which stack to use MSP or PSP and execution privilege in thread mode.

# Memory Organization

- Flat 32-bit addressing space
  - Smallest addressable unit = byte (8 bits).
  - Address space size =  $2^{32}$  bytes = 4GB.
  - All I/O and peripherals are memory mapped.
- ARMv7-M architecture does not support virtual addressing.
  - All memory addresses are physical and with direct access to memory and peripherals.
- Some data may require alignment when stored in memory.
  - 32-bit (4-byte) words may need to be aligned on 4-byte boundaries.
    - With the least two significant address bits set to 0, example addresses 0, 4, 8, C.
    - Words at address A include bytes A, A+1, A+2 and A+3.
  - 16-bit (2-byte) words may need to be aligned on 2-byte boundaries.
    - Half-words least significant bit address bit is set to 0, example addresses 0, 2, 4, etc...
    - Half-words at address A contain the bytes at address A and A+1.
  - Some memory access instructions support unaligned accesses, some do not.
    - This is normally taken care of by the compiler.
  - Stack pointer (SP) must always point to word-aligned data (4-byte aligned)

# Word Alignment

- Accessing 32-bit data that is word aligned at address 0x14
  - The data is stored in addresses,
    - 0x14 = 0101\_00.
    - 0x15 = 0101\_01.
    - 0x16 = 0101\_10.
    - 0x17 = 0101\_11.
  - Only the bottom two bits change for each byte address.
  - This means that all four bytes can be accessed using only the top 30-bits. All four can be written or read in one shot by accessing address 0x14.
- Accessing 32-bits of data that is unaligned at address 0x13.
  - The data is stored in addresses,
    - 0x13 = 0100\_11.
    - 0x14 = 0101\_00.
    - 0x15 = 0101\_01.
    - 0x16 = 0101\_10.
  - Now we would need at least two accesses.
    - One to access the data at 0x13.
    - The second to access to the rest of the words starting at 0x14.
- If data is always aligned it may cause memory to be used inefficiently.
  - For example, if we word aligned two consecutive characters (1 byte) at addresses 0x10 and 0x14, then address 0x11-0x13 and 0x15-0x17 would be unused causing 8-bytes of memory to be allocated when only 2 are needed.
  - If they are byte aligned at 0x10 and 0x11 then only 2-bytes are used.

# Endianness

- Endianness is the byte ordering within a specific word address.
  - Little endian stores the least significant byte at the lowest address location.
    - Better for math functions with carry operations
  - Big endian stores the most significant byte at the lowest address location. (network order)
    - Makes finding the sign easier as it is always in bit 7 of the lowest address independent of word size. Similar advantages for compares.
- The ARM default is little-endian.

