

CSCI 6461 Computer Systems Architecture

Lecture 2

ARM7TDMI overview

ARM7TDMI

- The ARM7TDMI core is the industry's most widely used 32-bit & 64-bit embedded RISC microprocessor solution
- Optimized for cost and power-sensitive applications,
- ARM7TDMI solution provides
 - low power consumption,
 - small size,
 - high performance needed in portable, embedded applications

ARM7TDMI

ARM - **A**dvanced **RISC** **M**achine

T - supports both ARM (32-bit) and Thumb (16-bit) instruction sets

D - Contains Debug extensions

M - Enhanced (relative to earlier ARM cores) 32x8 Multiplier block

I : EmbeddedICE macrocell on-chip logic to support debug operations. (In-circuit emulator)

RISC Architectures

- ARM is a reduced instruction set computer (**RISC**) architecture.
- The ARM instruction set includes only simple, commonly used instructions.
- The number of instructions is kept small
 - so that the hardware required to decode the instruction and its operands can be simple, small, and fast.
- Ideal for mobile platforms

CISC

- More elaborate operations that are less common are performed using sequences of multiple simple instructions.
- Architectures with many complex instructions, such as Intel's x86 architecture, are complex instruction set computers (**CISC**).

Small instruction sets

- One CISC operation translates into many, possibly even hundreds, of simple instructions in a RISC machine.
- The cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.
- A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small.

CISC vs RISC

- The CISC approach attempts to minimize the number of instructions per program
 - sacrificing the number of cycles per instruction.
- RISC does the opposite, reducing the cycles per instruction
 - at the cost of the number of instructions per program.
- Memory to memory load and store vs
 - Register to register load and store

Encoding Instructions

- An instruction set with 64 simple instructions would need 6 bits to encode the operation.
 - 000000 -> 111111
- An instruction set with 256 complex instructions would need ? bits of encoding per instruction.
- In a CISC machine, even though the complex instructions may be used only rarely,
 - they add overhead to **all** instructions, even the simple ones.

ARM processor modes

Exception modes	Mode	Description	
	Supervisor (SVC)	Entered on reset and when a Software Interrupt (SWI) instruction is executed	Privileged modes
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a low priority (normal) interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most applications/OS tasks run	Unprivileged mode

It is possible to make mode changes under software control, but most are normally caused by external conditions or exceptions.

User Mode and Interrupts

- Most application programs will execute in **User** mode.
- The other modes are known as **privileged** modes
- They provide a way to service exceptions or to access protected resources,
 - such as bits that disable sections of the core.
- ARM7TDMI has two types of interrupts:
 - fast interrupts
 - lower priority interrupts

Interrupt Types

- Think of the fast interrupt as one that might be used to indicate high priority events
- These events need an immediate response
 - the machine is about to lose power in a few milliseconds!
- Lower priority interrupts might be used for indicating that a peripheral needs to be serviced,
 - a user has typed a key on the keyboard
 - or clicked the mouse button

Abort Mode

- Abort mode allows the processor to recover from exceptional conditions such as a
 - memory access to an address that doesn't physically exist, for either an instruction or data.
- The processor will switch to Undefined mode when it sees an instruction in the pipeline that it does not recognize.
- It is now the programmer's (or the operating system's) responsibility to determine how the machine should recover from this error

ARM Registers

A brief overview

ARM7TDMI Registers

The ARM7TDMI processor has a total of **37** registers:

- 30 general-purpose registers
- 6 status registers
- A Program Counter register
- The general-purpose registers are 32 bits wide
- They are named R0, R1, etc.

ARM7TDMI Registers


- The registers are arranged in partially overlapping banks,
- meaning that you as a programmer see a different register bank for each processor mode.
- At any one time, you get 15 general-purpose registers (R0 - R14),
 - one or two status registers,
- and the Program Counter (PC or R15) are visible.

ARM7TDMI Registers

The registers have the same names, but depending on the **mode**, they are physically different registers.

Mode					
User/System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

 = *banked register*

A note on FIQ

FIQ mode is often used in real-time systems for tasks like:

- Handling critical hardware interrupts (e.g., timers, DMA controllers).
- Low-latency operations in embedded systems, such as motor control or high-speed communication.

Banked Registers

- In User/System mode, you have registers R0 to R14, a Program Counter, and a **Current Program Status Register** (CPSR / APSR) available.
- If the processor were to suddenly change to Abort mode, it would swap, or bank out, registers R13 and R14 with different R13 and R14 registers.
- The largest number of registers swapped occurs when the processor changes to FIQ (Fast Interrupt Request) mode.

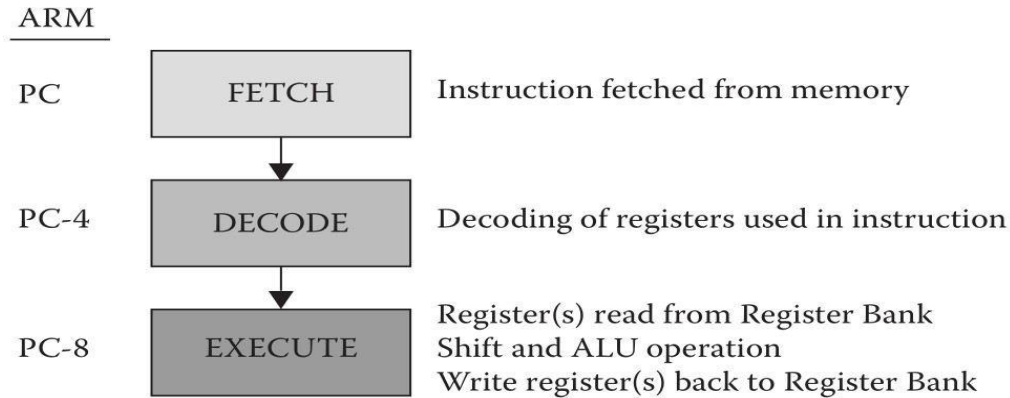
Banked Registers

- During an interrupt, it is normally necessary to drop everything you're doing and begin to work on one task: namely, saving the state of the machine and transition to handling the interrupt code quickly.
- Rather than moving data from all the registers on the processor to external memory, the machine swaps certain registers with new ones to allow the programmer access

Pipelined Architecture

Step by step processing

FDE Cycle



The ARM7TDMI is a pipelined architecture meaning that while one instruction is being fetched, another is being decoded, and yet another is being executed.

PC and CPSR

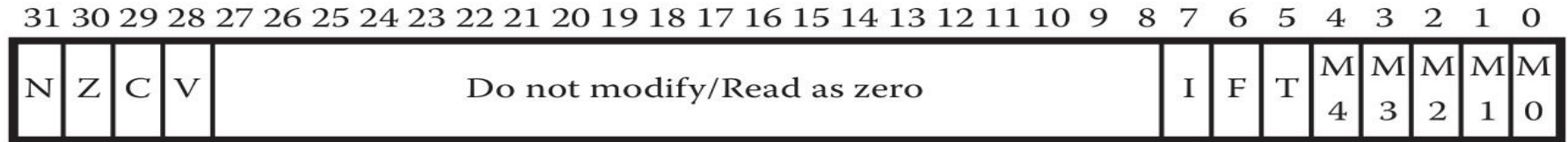
- The address of the instruction that is being **fetch**ed (not the one being executed) is contained in the Program Counter
- The Current Program Status Register (CPSR) contains
 - condition code flags,
 - interrupt enable flags,
 - the current mode,
 - and the current state.

SPSR - Saved Registers

- Each privileged mode (except System mode) has a Saved Program Status Register (SPSR) that is used to preserve the value of the CPSR when an exception occurs.
- Since User mode and System mode are not entered on any exception, they do not have an SPSR, and a register to preserve the CPSR is not required.
- In later ARM versions, the CPSR is called the Application Program Status Register (APSR).

CPSR & SPSR

The format of the Current Program Status Register and the Saved Program Status Register:



→ The condition code flags in the CPSR can be altered by arithmetic and logical instructions, such as subtractions, logical shifts, and rotations.

NZCV Condition Flags

- N - Negative. The result of the operation was negative.
- Z - Zero. The result of the operation was Zero. Can occur when two values being compared are the same, or when you have decremented a value and the result is zero.
- C - Carry (unsigned). The result of the operation cannot fit in the available number of bits. For A64 code using x registers, this flag is set when the result of an operation is 65-bits long.
- V - Overflow (signed). When an operation on two numbers of the **same sign** produce a number of the **opposite** sign. Here, the MSB has been toggled.

Why do we care about carries?

Consider the following
5 bit register example
We want to add
 $28 + 6 = 34$

$$28 + 6$$

Carry	1	1	1	0	0
-------	---	---	---	---	---

	1	1	1	0	0		28
+	0	0	1	1	0		+ 6
<hr/>							
	1	0	0	0	1	0	2

Extra bit is
discarded.

5-bit result

$$28 + 6 = 2$$

Carry = 1

CPSR & SPSR

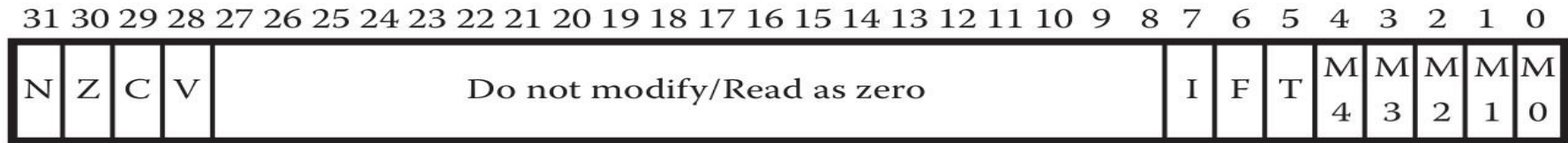
By allowing these bits to be used with all the instructions on the ARM7TDMI, the processor can conditionally execute an instruction.

The bottom eight bits of a status register (the mode bits M[4:0], I, F, and T) are known as the **control** bits.

Control Bits

- The I and F bits are the interrupt disable bits, which disable interrupts in the processor if they are set.
- The **I** bit controls the IRQ interrupts (normal priority interrupt).
- The **F** bit controls the FIQ interrupts (fast/high priority interrupt).
- The **T** bit is a **status** bit, meant only to indicate the state of the machine, so as a programmer you would only read this bit, not write to it.
 - If the bit is set to 1, the core is executing Thumb code, which consists of 16-bit instructions.
 - The processor changes between ARM and Thumb state via a special instruction.
- These control bits can be altered by software only when the processor is in a privileged mode.

Mode Bits



The Mode Bits

xPSR[4:0]

10000

10001

10010

10011

10111

11011

11111

Mode

User mode

FIQ mode

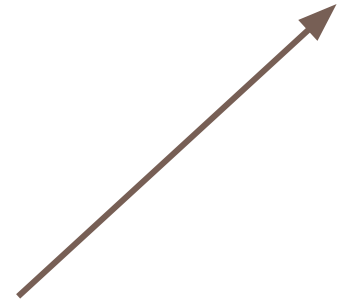
IRQ mode

Supervisor mode

Abort mode

Undefined mode

System mode



ARM, Thumb, and Thumb-2 instructions

ARM instructions are **32 bits wide** .

Thumb instructions are **16 bits wide** to achieve higher code density.

They are identical to regular ARM instructions but generally have limitations.

ARM instruction	ADD	R0, R0, R2
Thumb instruction	ADD	R0, R2

Almost all ARM instructions have Thumb equivalents.

Thumb instructions

- Because Thumb instructions are less powerful,
 - more are required to write an equivalent program.
- However, the instructions are half as long,
 - giving overall Thumb code size of about 65% of the ARM equivalent.

Thumb instructions

- The Thumb instruction set is valuable not only to reduce the size and cost of code storage memory,
 - but also to allow for an inexpensive 16-bit bus to instruction memory and to reduce the power consumed by fetching instructions from the memory
- Thumb instruction encoding is more complex and irregular than ARM instructions to pack as much useful information

Thumb and Thumb-2

- ARM subsequently refined the Thumb instruction set and added a number of 32-bit Thumb-2 instructions
 - to boost performance of common operations and to allow any program to be written in Thumb mode
- Thumb-2 is a superset of Thumb instructions,
 - including new 32-bit instructions for more complex operations.

Thumb and Thumb-2

- In other words, Thumb-2 is a combination of both 16-bit and 32-bit instructions.
- Some cores, such as the Cortex-M3 and M4,
 - only execute Thumb-2 instructions
 - => there are no ARM instructions at all.

Some more instructions

- Digital Signal Processing instructions
- Power Saving instructions
- Virtualization instructions
- SIMD instructions

We look look at some of these in later lectures

Two's Complement

How do we represent negative numbers

Negative Numbers

- There have been many theories about how best to represent negative numbers in computer systems.
- One idea - dedicate the **MSB** to represent the sign
- So in an 5-bit number (3)
 - $00011 = 3_{10}$
 - $10011 = -3_{10}$
- this scheme fails in a few places
 - it allows us to represent 0 and negative 0 (00000 and 10000)

Addition ... fail

$$\begin{array}{r} 00011 \\ + 10011 \\ \hline 10110 \end{array}$$

- So this is -6_{10} which is wrong - it should be 0
- So we would need special hardware to deal with negative numbers

Special Hardware

- Special additional hardware for **add/mul/sub** is highly undesirable
- So what we need is a scheme for representing negative numbers with the following property
 - No special/additional hardware required for **add/mul/sub**
 - No contradictory positive and negative 0
 - Normal performance / cycle speeds

Solution - two's complement

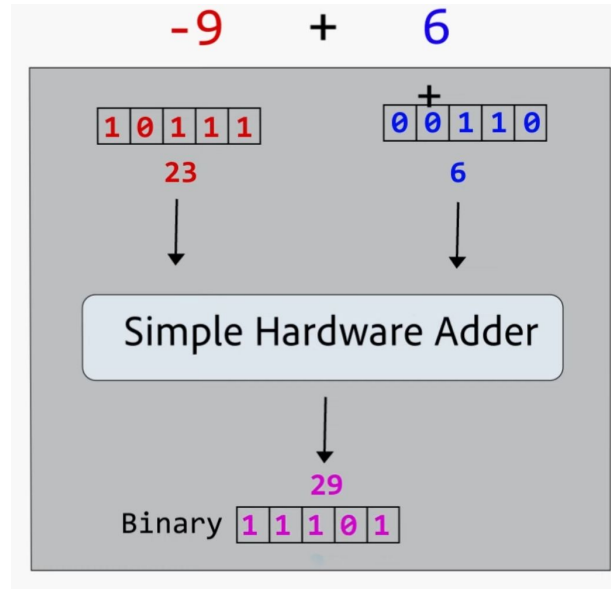
Steps:

- Take the binary representation of the positive number and invert all the bits (eg, 9_{10})
 $01001 \Rightarrow 10110$
- Add one
 - $10110 \Rightarrow 10111$

Does it work?

Lets try $-9_{10} + 6_{10}$

```
  10111
+ 00110
-----
  11101
```



But is 11101
correct for
-3?

Let's check ...
& tell me what
you get