# Module-5

Introduction to THUMB instruction set
Efficient C Programming

# Introduction to THUMB instruction set

- Introduction
- THUMB register usage
- ARM – THUMB interworking
- Other branch instructions
- Data processing instructions
- Stack instructions
- Software interrupt instructions

# Introduction

- Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space.
- Thumb has higher *code density*—the space taken up in memory by an executable program—than ARM.
- For memory-constrained embedded systems, for example, mobile phones and PDAs, code density is very important.
- On average, a Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation.
- Each Thumb instruction is related to a 32-bit ARM instruction.

# THUMB register usage

Summary of Thumb register usage.

| Registers | Access |
|---|---|
| *r0–r7* | fully accessible |
| *r8–r12* | only accessible by `MOV`, `ADD`, and `CMP` |
| *r13 sp* | limited accessibility |
| *r14 lr* | limited accessibility |
| *r15 pc* | limited accessibility |
| *cpsr* | only indirect access |
| *spsr* | no access |

# ARM-Thumb Interworking

- *ARM-Thumb interworking* is the name given to the method of linking ARM and Thumb code together for both assembly and C/C++.
- ATPCS defines the ARM and Thumb procedure call standards.
- To call a Thumb routine from an ARM routine, the core has to change state.
- This state change is shown in the *T* bit of the *cpsr*.
- The BX and BLX branch instructions cause a switch between ARM and Thumb state while branching to a routine.

# Other Branch Instructions

- There are two variations of the standard branch instruction, or B.
- The conditional branch instruction is the only conditionally executed instruction in Thumb state.

```
Syntax: B<cond> label
        B label
        BL label
```

| | | |
|---|---|---|
| B | branch | *pc* = *label* |
| BL | branch with link | *pc* = *label*<br>*lr* = (instruction address after the BL) + 1 |

# Data Processing Instructions

- The data processing instructions manipulate data within registers.
- They include move instructions, arithmetic instructions, shifts, logical instructions, comparison instructions, and multiply instructions.
- The Thumb data processing instructions are a subset of the ARM data processing instructions.
- These instructions follow the same style as the equivalent ARM instructions.
- Most Thumb data processing instructions operate on low registers and update the *cpsr*.

# Thumb ADD Instruction: Example

This example shows a simple Thumb **ADD** instruction. It takes two low registers *r1* and *r2* and adds them together. The result is then placed into register *r0*, overwriting the original contents. The *cpsr* is also updated.

```
PRE     cpsr = nzcvIFT_SVC
        r1 = 0x80000000
        r2 = 0x10000000

        ADD     r0, r1, r2

POST    r0 = 0x90000000
        cpsr = NzcvIFT_SVC
```

# Separate Barrel Shift Operations

Thumb deviates from the ARM style in that the barrel shift operations (ASR, LSL, LSR, and ROR) are separate instructions. This example shows the logical left shift (LSL) instruction to multiply register *r2* by 2.

```
PRE     r2 = 0x00000002
        r4 = 0x00000001

        LSL     r2, r4

POST    r2 = 0x00000004
        r4 = 0x00000001
```

# Single-Register Load-Store Instructions

- The Thumb instruction set supports load and storing registers, or LDR and STR.

| | | |
|---|---|---|
| LDR | load word into a register | *Rd <- mem32[address]* |
| STR | save word from a register | *Rd -> mem32[address]* |
| LDRB | load byte into a register | *Rd <- mem8[address]* |
| STRB | save byte from a register | *Rd -> mem8[address]* |
| LDRH | load halfword into a register | *Rd <- mem16[address]* |
| STRH | save halfword into a register | *Rd -> mem16[address]* |
| LDRSB | load signed byte into a register | *Rd <- SignExtend(mem8[address])* |
| LDRSH | load signed halfword into a register | *Rd <- SignExtend(mem16[address])* |

# Multiple-Register Load-Store Instructions

- The Thumb versions of the load-store multiple instructions are reduced forms of the ARM load-store multiple instructions.
- They only support the increment after (IA) addressing mode.

```
Syntax : <LDM|STM>IA Rn!, {low Register list}
```

| | | |
|---|---|---|
| LDMIA | load multiple registers | {Rd}*N <- mem32[Rn + 4*N], Rn = Rn + 4*N |
| STMIA | save multiple registers | {Rd}*N -> mem32[Rn + 4*N], Rn = Rn + 4*N |

# Stack Instructions

- The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

```
Syntax: POP {low_register_list{, pc}}
        PUSH {low_register_list{, lr}}
```

| | | |
|---|---|---|
| POP | pop registers from the stacks | Rd*N <- mem32[sp+4*N], sp=sp+4*N |
| PUSH | push registers on to the stack | Rd*N -> mem32[sp+4*N], sp=sp-4*N |

- The interesting point to note is that there is no stack pointer in the instruction. This is because the stack pointer is fixed as register *r13* in Thumb operations and *sp* is automatically updated.

# Software Interrupt Instruction

- Similar to the ARM equivalent, the Thumb software interrupt (SWI) instruction causes a software interrupt exception.
- If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception.
- The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent.
- It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

# Summary

- All Thumb instructions are 16 bits in length.
- Thumb provides approximately 30% better code density over ARM code.
- Most code written for Thumb is in a high-level language such as C and C++.
- ATPCS defines how ARM and Thumb code call each other, called *ARM-Thumb interworking*.
- In Thumb, only the branch instructions are conditionally executed. The barrel shift operations (ASR, LSL, LSR, and ROR) are separate instructions.
- The multiple-register load-store instructions only support the increment after (IA) addressing mode.
- The Thumb instruction set includes POP and PUSH instructions as stack operations. These instructions only support a full descending stack.
- There are no Thumb instructions to access the coprocessors, cpsr, and spsr.

# Efficient C Programming

- Overview of C Compilers and optimization
- Basic C Data types
- C looping structures

# Overview of C Compilers and optimization

- Optimizing code takes time and reduces source code readability.
- It’s only worth optimizing functions that are frequently executed and important for performance.
- C compilers have to translate your C function literally into assembler so that it works for all possible inputs.
- In practice, many of the input combinations are not possible or won’t occur.
- Let’s start by looking at an example of the problems the compiler faces.

# Example: memclr function

- The **memclr** function clears N bytes of memory at address data.

```
void memclr(char *data, int N)
{
  for (; N>0; N--)
  {
    *data=0;
    data++;
  }
}
```

- No matter how advanced the compiler, it does not know whether N can be 0 on input or not.
- Therefore the compiler needs to test for this case explicitly before the first iteration of the loop.
- The compiler doesn't know whether the data array pointer is four-byte aligned or not.
- If it is four-byte aligned, then the compiler can clear four bytes at a time using an int store rather than a char store.
- Nor does it know whether N is a multiple of four or not.
- The compiler must be conservative and assume all possible values for N and all possible alignments for data.

# Overcoming problems faced by compilers

- To write efficient C code, you must be aware of areas where the C compiler has to be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of a specific C compiler.
- Most of this section covers the first two points above and should be applicable to any ARM C compiler.

# Basic C Data Types

- Let’s start by looking at how ARM compilers handle the basic C data types.
- There are also differences between the addressing modes available when loading and storing data of each type.
- ARM processors have 32-bit registers and 32-bit data processing operations.
- The ARM architecture is a RISC load/store architecture.
- In other words you must load values from memory into registers before acting on them.
- There are no arithmetic or logical instructions that manipulate values in memory directly.

# Local Variable Types

- ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data.
- However, most ARM data processing operations are 32-bit only.
- For this reason, you should use a 32-bit datatype, int or long, for local variables wherever possible.
- Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value.
- To see the effect of local variable types, let’s consider a simple example.