ARM Cortex-M3/M4 Instruction Set & Architecture
Cortex M4 block diagram

<table>
<thead>
<tr>
<th>ISA Support</th>
<th>Thumb® / Thumb-2</th>
</tr>
</thead>
</table>
| DSP Extensions       | Single cycle 16/32-bit MAC
                       | Single cycle dual 16-bit MAC
                       | 8/16-bit SIMD arithmetic
                       | Hardware Divide (2-12 Cycles) |
| Floating Point Unit | Single precision floating point unit
                       | IEEE 754 compliant |
| Pipeline            | 3-stage + branch speculation |
| Performance Efficiency | 3.40 CoreMark/MHz* |
| Performance Efficiency | Without FPU: 1.25 / 1.52 / 1.91 DMIPS/MHz**
                       | With FPU: 1.27 / 1.55 / 1.95 DMIPS/MHz** |
| Memory Protection   | Optional 8 region MPU with sub regions and background region |
| Interrupts          | Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts |
| Interrupt Priority Levels | 8 to 256 priority levels |
| Wake-up Interrupt Controller | Up to 240 Wake-up Interrupts |
| Sleep Modes         | Integrated WiFi and WFE instructions and Sleep On Exit capability.
                       | Sleep & Deep Sleep Signals.
                       | Optional Retention Mode with ARM Power Management Kit |
| Bit Manipulation    | Integrated Instructions & Bit Banding |
| Debug               | Optional JTAG & Serial-Wire Debug Ports. Up to 8 Breakpoints and 4 Watchpoints. |

Cortex M4
Embedded processor for DSP with FPU
(b) The Cortex-M4 ISA is enhanced efficient DSP features including extended single-cycle cycle 16/32-bit multiply-accumulate (MAC), dual 16-bit MAC instructions, optimized 8/16-bit SIMD arithmetic and saturating arithmetic instructions
# ARM Cortex-M Series Family

<table>
<thead>
<tr>
<th>Processor</th>
<th>ARM Architecture</th>
<th>Core Architecture</th>
<th>Thumb°</th>
<th>Thumb°-2</th>
<th>Hardware Multiply</th>
<th>Hardware Divide</th>
<th>Saturated Math</th>
<th>DSP Extensions</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-M0</td>
<td>ARMv6-M</td>
<td>Von Neumann</td>
<td>Most</td>
<td>Subset</td>
<td>1 or 32 cycle</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M0+</td>
<td>ARMv6-M</td>
<td>Von Neumann</td>
<td>Most</td>
<td>Subset</td>
<td>1 or 32 cycle</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M1</td>
<td>ARMv6-M</td>
<td>Von Neumann</td>
<td>Most</td>
<td>Subset</td>
<td>3 or 33 cycle</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M3</td>
<td>ARMv7-M</td>
<td>Harvard</td>
<td>Entire</td>
<td>Entire</td>
<td>1 cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cortex-M4</td>
<td>ARMv7E-M</td>
<td>Harvard</td>
<td>Entire</td>
<td>Entire</td>
<td>1 cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Optional</td>
</tr>
</tbody>
</table>
ARMv7 M (Thumb-2) features

- Mix of 16 and 32b instructions
- 1.2 CPI
- 26% higher code density ARM32
- 25% speed improvement over Thumb16

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>16b x 16b</td>
<td>32b</td>
<td>1</td>
</tr>
<tr>
<td>32b x 16b</td>
<td>32b</td>
<td>1</td>
</tr>
<tr>
<td>32b x 32b</td>
<td>32b</td>
<td>1</td>
</tr>
<tr>
<td>32b x 32b</td>
<td>64b</td>
<td>3-7*</td>
</tr>
</tbody>
</table>
Thumb-2

• Mixes 16 and 32 bits instructions
  – Enhancements: e.g. UDIV, SDIF division, bit-field operators UFBX, BFC, BFE, wrt traditional ARMv4T
  – No need to mode switch, can be mixed freely

• **Not** backwards binary compatible
  – But porting is «easy»
Cortex-M4 Processor Overview

• Cortex-M4 Processor
  – Introduced in 2010
  – Designed with a large variety of highly efficient signal processing features
  – Features extended single-cycle multiply accumulate instructions, optimized SIMD arithmetic, saturating arithmetic and an optional Floating Point Unit.

• High Performance Efficiency
  – 1.25 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz) at the order of µWatts / MHz

• Low Power Consumption
  – Longer battery life – especially critical in mobile products

• Enhanced Determinism
  – The critical tasks and interrupt routines can be served quickly in a known number of cycles
Cortex-M4 Processor Features

- 32-bit Reduced Instruction Set Computing (RISC) processor
- Harvard architecture
  - Separated data bus and instruction bus
- Instruction set
  - Include the entire Thumb®-1 (16-bit) and Thumb®-2 (16/32-bit) instruction sets
- 3-stage + branch speculation pipeline
- Performance efficiency
  - 1.25 – 1.95 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz)
- Supported Interrupts
  - Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
  - 8 to 256 interrupt priority levels
Cortex-M4 Processor Features

• Supports Sleep Modes
  – Up to 240 Wake-up Interrupts
  – Integrated WFI (Wait For Interrupt) and WFE (Wait For Event) Instructions and Sleep On Exit capability (to be covered in more detail later)
  – Sleep & Deep Sleep Signals
  – Optional Retention Mode with ARM Power Management Kit

• Enhanced Instructions
  – Hardware Divide (2-12 Cycles)
  – Single-Cycle 16, 32-bit MAC, Single-cycle dual 16-bit MAC
  – 8, 16-bit SIMD arithmetic

• Debug
  – Optional JTAG & Serial-Wire Debug (SWD) Ports
  – Up to 8 Breakpoints and 4 Watchpoints

• Memory Protection Unit (MPU)
  – Optional 8 region MPU with sub regions and background region
Cortex-M4 Processor Features

- Cortex-M4 processor is designed to meet the challenges of low dynamic power constraints while retaining light footprints
  - 180 nm ultra low power process – 157 µW/MHz
  - 90 nm low power process – 33 µW/MHz
  - 40 nm G process – 8 µW/MHz

<table>
<thead>
<tr>
<th>ARM Cortex-M4 Implementation Data</th>
<th>Process</th>
<th>Dynamic Power</th>
<th>Floorplanned Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>180ULL (7-track, typical 1.8v, 25C)</td>
<td>157 µW/MHz</td>
<td>0.56 mm²</td>
</tr>
<tr>
<td></td>
<td>90LP (7-track, typical 1.2v, 25C)</td>
<td>33 µW/MHz</td>
<td>0.17 mm²</td>
</tr>
<tr>
<td></td>
<td>40G 9-track, typical 0.9v, 25C</td>
<td>8 µW/MHz</td>
<td>0.04 mm²</td>
</tr>
</tbody>
</table>
Cortex-M4 Block Diagram

- ARM Cortex-M4 Microprocessor
  - Optional WIC
  - Nested Vector Interrupt Controller (NVIC)
  - Optional FPU
  - Processor core
  - Optional Memory protection unit
  - Optional Embedded Trace Macrocell
  - Optional Serial Wire Viewer
  - Optional Debug Access Port
  - Optional Flash patch
  - Optional Data watchpoints
  - Bus matrix
  - Code interface
  - SRAM and peripheral interface
Cortex-M4 Block Diagram

• Bus interconnect
  – Allows data transfer to take place on different buses simultaneously
  – Provides data transfer management, e.g. a write buffer, bit-oriented operations (bit-band)
  – May include bus bridges (e.g. AHB-to-APB bus bridge) to connect different buses into a network using a single global memory space
  – Includes the internal bus system, the data path in the processor core, and the AHB LITE interface unit

• Debug subsystem
  – Handles debug control, program breakpoints, and data watchpoints
  – When a debug event occurs, it can put the processor core in a halted state, where developers can analyse the status of the processor at that point, such as register values and flags
Cortex-M4 Block Diagram

• Nested Vectored Interrupt Controller (NVIC)
  – Up to 240 interrupt request signals and a non-maskable interrupt (NMI)
  – Automatically handles nested interrupts, such as comparing priorities between interrupt requests and the current priority level

• Wakeup Interrupt Controller (WIC)
  – For low-power applications, the microcontroller can enter sleep mode by shutting down most of the components.
  – When an interrupt request is detected, the WIC can inform the power management unit to power up the system.

• Memory Protection Unit (optional)
  – Used to protect memory content, e.g. make some memory regions read-only or preventing user applications from accessing privileged application data
3-Stage Pipeline ARM Organization

- **Register Bank**
  - 2 read ports, 1 write ports, access any register
  - 1 additional read port, 1 additional write port for r15 (PC)

- **Barrel Shifter**
  - Shift or rotate the operand by any number of bits

- **ALU**

- **Address register and incrementer**

- **Data Registers**
  - Hold data passing to and from memory

- **Instruction Decoder and Control**
3-Stage Pipeline (1/2)

- **Fetch**
  - The instruction is fetched from memory and placed in the instruction pipeline

- **Decode**
  - The instruction is decoded and the datapath control signals prepared for the next cycle

- **Execute**
  - The register bank is read, an operand shifted, the ALU result generated and written back into destination register
3-Stage Pipeline (2/2)

- At any time slice, 3 different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operations.
- When the processor is executing data processing instructions, the latency = 3 cycles and the throughput = 1 instruction/cycle.
- There are exceptions: multi-cycle instructions and branches.
Data Processing Instruction

All operations take place in a single clock cycle

(a) register - register operations

(b) register - immediate operations
Data Transfer Instructions

- Computes a memory address similar to a data processing instruction
- Load instruction follows a similar pattern except that the data from memory only gets as far as the ‘data in’ register on the 2nd cycle and a 3rd cycle is needed to transfer the data from there to the destination register

(a) 1st cycle - compute address

(b) 2nd cycle - store data & auto-index
Branch Instructions

- The third cycle, which is required to complete the pipeline refilling, is also used to mark the small correction to the value stored in the link register in order that is points directly at the instruction which follows the branch.
Branch Pipeline Example

- Breaking the pipeline
- Two clock stalls $\rightarrow$ IPC goes down
Pipeline summary

Harvard architecture
   Separate Instruction & Data buses
   enable parallel fetch & store
Advanced 3-Stage Pipeline
   Includes Branch Forwarding & Speculation
Additional Write-Back via Bus Matrix
Decoding Thumb

Instruction 1: Fetch, Decode, Execute
Instruction 2: Fetch, Decode, Execute
Instruction 3: Fetch, Decode, Execute
Instruction 4: Fetch, Decode, Execute
Instruction Prefetch & Execution

Handles mix of 16+32b instructions which can be misaligned in word address

Branch speculation
The ARM has seven basic operating modes:
- Each mode has access to:
  - Its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor (SVC)</td>
<td>Entered on reset and when a Software Interrupt instruction (SWI) is executed</td>
</tr>
<tr>
<td>FIQ</td>
<td>Entered when a high priority (fast) interrupt is raised</td>
</tr>
<tr>
<td>IRQ</td>
<td>Entered when a low priority (normal) interrupt is raised</td>
</tr>
<tr>
<td>Abort</td>
<td>Used to handle memory access violations</td>
</tr>
<tr>
<td>Undef</td>
<td>Used to handle undefined instructions</td>
</tr>
<tr>
<td>System</td>
<td>Privileged mode using the same registers as User mode</td>
</tr>
<tr>
<td>User</td>
<td>Mode under which most Applications / OS tasks run</td>
</tr>
</tbody>
</table>
Operating Modes

User mode:
- Normal program execution mode
- System resources unavailable
- Mode changed by exception only

Exception modes:
- Entered upon exception
- Full access to system resources
- Mode changed freely

<table>
<thead>
<tr>
<th>Modes (Thread out of reset)</th>
<th>Operations (privilege out of reset)</th>
<th>Stacks (Main out of reset)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Handler</strong></td>
<td>Privileged execution Full control</td>
<td>Main Stack Used by OS and Exceptions</td>
</tr>
<tr>
<td>- An exception is being processed</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td>Privileged/Unprivileged</td>
<td>Main/Process</td>
</tr>
<tr>
<td>- No exception is being processed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Normal code is executing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode</th>
<th>Priority</th>
<th>IV Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>Undefined</td>
<td>6</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Software interrupt</td>
<td>Supervisor</td>
<td>6</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>5</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
<td>2</td>
<td>0x00000010</td>
</tr>
<tr>
<td>Interrupt</td>
<td>IRQ</td>
<td>4</td>
<td>0x00000018</td>
</tr>
<tr>
<td>Fast interrupt</td>
<td>FIQ</td>
<td>3</td>
<td>0x0000001C</td>
</tr>
</tbody>
</table>

Table 1 - Exception types, sorted by Interrupt Vector addresses
Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Functions (and banked registers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R1</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R2</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R3</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R4</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R5</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R6</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R7</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R8</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R9</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R10</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R11</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R12</td>
<td>General-purpose register</td>
</tr>
<tr>
<td>R13 (MSP)</td>
<td>Main Stack Pointer (MSP)</td>
</tr>
<tr>
<td>R13 (PSP)</td>
<td>Process Stack Pointer (PSP)</td>
</tr>
<tr>
<td>R14</td>
<td>Link Register (LR)</td>
</tr>
<tr>
<td>R15</td>
<td>Program Counter (PC)</td>
</tr>
</tbody>
</table>

Low registers:

High registers:
ARM Registers

• 31 general-purpose 32-bit registers
• 16 visible, R0 – R15
• Others speed up the exception process
ARM Registers (2)

• Special roles:
  – Hardware
    • R14 – Link Register (LR): optionally holds return address for branch instructions
    • R15 – Program Counter (PC)
  – Software
    • R13 - Stack Pointer (SP)
ARM Registers (3)

• Current Program Status Register (CPSR)
• Saved Program Status Register (SPSR)
• On exception, entering mod mode:
  – (PC + 4) → LR
  – CPSR → SPSR_mod
  – PC ← IV address
  – R13, R14 replaced by R13_mod, R14_mod
  – In case of FIQ mode R7 – R12 also replaced
### Special Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xPSR</td>
<td>Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number</td>
</tr>
<tr>
<td>PRIMASK</td>
<td>Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault</td>
</tr>
<tr>
<td>FAULTMASK</td>
<td>Disable all interrupts except the NMI</td>
</tr>
<tr>
<td>BASEPRI</td>
<td>Disable all interrupts of specific priority level or lower priority level</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Define privileged status and stack pointer selection</td>
</tr>
</tbody>
</table>
Memory map

- Statically defined memory map (faster addr decoding) 4GB of address space
Bit Banding

- Fast single-bit manipulation: 1MB → 32MB aliased regions in SRAM & Peripheral space

**Traditional bit manipulation method**

```
LDR R0, =0x200FFFFF ; Setup address
MOV R2, #0x4 ; Setup data
LDR R1, [R0] ; Read
ORR R1, R2 ; Modify bit
STR R1, [R0] ; Write back result
```

**Direct, single cycle access with bit banding**

```
LDR R0, =0x23FFFFFFC ; Setup address
MOV R1, #0x1 ; Setup data
STR R1, [R0] ; Write
```

Diagram showing the process of reading from SRAM, masking and modifying the bit element, and writing back to SRAM.
Cortex M3/M4 Instruction Set
Major Elements of ISA
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianness

Endianness

32-bits
Traditional ARM instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Shift & ALU operation in single clock cycle
- Access memory with load and store instructions only
  - Load/Store multiple register
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the ‘S’ suffix to the instruction
Thumb-2

• Original 16-bit Thumb instruction set
  – a subset of the full ARM instructions
  – performs similar functions to selective 32-bit ARM instructions but in 16-bit code size
• For ARM instructions that are not available
  – more 16-bit Thumb instructions are needed to execute the same function compared to using ARM instructions
  – but performance may be degraded
• Hence the introduction of the Thumb-2 instruction set
  – enhances the 16-bit Thumb instructions with additional 32-bit instructions
• All ARMv7 chips support the Thumb-2 (& ARM) instruction set
  – but Cortex-M3 supports only the 16-bit/32-bit Thumb-2 instruction set
16bit Thumb-2

Some of the changes used to reduce the length of the instructions from 32 bits to 16 bits:

- reduce the number of bits used to identify the register
  - less number of registers can be used
- reduce the number of bits used for the immediate value
  - smaller number range
- remove options such as ‘S’
  - make it default for some instructions
- remove conditional fields (N, Z, V, C)
- no conditional executions (except branch)
- remove the optional shift (and no barrel shifter operation)
  - introduce dedicated shift instructions
- remove some of the instructions
  - more restricted coding
Thumb-2 Implementation

- The 32-bit ARM Thumb-2 instructions are added through the space occupied by the Thumb BL and BLX instructions

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hw1</td>
<td></td>
<td>Hw2</td>
</tr>
</tbody>
</table>

32-bit Thumb-2 Instruction format

- The first Halfword (Hw1)
  - determines the instruction length and functionality
- If the processor decodes the instruction as 32-bit long
  - the processor fetches the second halfword (hw2) of the instruction from the instruction address plus two
Unified Assembly Language

- UAL supports generation of either Thumb-2 or ARM instructions from the same source code
  - same syntax for both the Thumb code and ARM code
  - enable portability of code for different ARM processor families
- Interpretation of code type is based on the directive listed in the assembly file
- Example:
  - For GNU GAS, the directive for UAL is `.syntax unified`
- For ARM assembler, the directive for UAL is THUMB
32bit Instruction Encoding

Example: ADD instruction format

- ARM 32-bit encoding for ADD with immediate field

Typical settings:
- Major opcode = 00 (this indicates data operation instructions)
- Minor opcode = 0100 (specifically, 100 ⇒ ADD instruction)
- Immediate flag = 1 (immediate field in operand 2)
- Set status flag = 1 (set carry flag after operation)
ARM and 16-bit Instruction Encoding

**ARM 32-bit encoding:** `ADDS r1, r1, #2`

![ARM 32-bit encoding diagram](image)

- Equivalent 16-bit Thumb instruction: `ADD r1, #2`
  - No condition flag
  - No rotate field for the immediate number
  - Use 3-bit encoding for the register
  - Shorter opcode with implicit flag settings (e.g. the set status flag is always set)
Application Program Status Register (APSR)

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:
  
  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then \( N = 1 \) if the result is negative and \( N = 0 \) if it is positive or zero.
  
  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
  
  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.
  
  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.
  
  Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.
Updating the APSR

- **SUB Rx, Ry**
  - $Rx = Rx - Ry$
  - APSR unchanged

- **SUBS**
  - $Rx = Rx - Ry$
  - APSR N or Z bits might be set

- **ADD Rx, Ry**
  - $Rx = Rx + Ry$
  - APSR unchanged

- **ADDS**
  - $Rx = Rx + Ry$
  - APSR C or V bits might be set
Conditional Execution

• Each data processing instruction prefixed by condition code
• Result – smooth flow of instructions through pipeline
• 16 condition codes:

<table>
<thead>
<tr>
<th>EQ</th>
<th>equal</th>
<th>MI</th>
<th>negative</th>
<th>HI</th>
<th>unsigned higher</th>
<th>GT</th>
<th>signed greater than</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>not equal</td>
<td>PL</td>
<td>positive or zero</td>
<td>LS</td>
<td>unsigned lower or same</td>
<td>LE</td>
<td>signed less than or equal</td>
</tr>
<tr>
<td>CS</td>
<td>unsigned higher or same</td>
<td>VS</td>
<td>overflow</td>
<td>GE</td>
<td>signed greater than or equal</td>
<td>AL</td>
<td>always</td>
</tr>
<tr>
<td>CC</td>
<td>unsigned lower</td>
<td>VC</td>
<td>no overflow</td>
<td>LT</td>
<td>signed less than</td>
<td>NV</td>
<td>special purpose</td>
</tr>
</tbody>
</table>
Conditional Execution

- Every ARM (32 bit) instruction is conditionally executed.
- The top four bits are ANDed with the CPSR condition codes, if they do not match the instruction is executed as NOP.
- The AL condition is used to execute the instruction irrespective of the value of the condition code flags.
- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. Ex: SUBS r1,r1,#1.
- Conditional Execution improves code density and performance by reducing the number of forward branch instructions.

<table>
<thead>
<tr>
<th>Normal</th>
<th>Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP r3,#0</td>
<td>CMP r3,#0</td>
</tr>
<tr>
<td>BEQ skip</td>
<td>ADDNE r0,r1,r2</td>
</tr>
<tr>
<td>ADD r0,r1,r2</td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>addne r0,r1,r2</td>
</tr>
</tbody>
</table>

- Normal
- Conditional
Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by post-fixing them with the appropriate condition code.
  - This can increase code density and increase performance by reducing the number of forward branches.
    
    | Instruction | Description |
    |-------------|-------------|
    | CMP r0, r1  | r0 - r1, compare r0 with r1 and set flags |
    | ADDGT r2, r2, #1 | if > r2=r2+1 flags remain unchanged |
    | ADDLE r3, r3, #1 | if <= r3=r3+1 flags remain unchanged |

- By default, data processing instructions do not affect the condition flags but this can be achieved by post fixing the instruction (and any condition code) with an “S”

```
loop
ADD r2, r2, r3  \( \Rightarrow r2=r2+r3 \)
SUBS r1, r1, #0x01 \( \Rightarrow \) decrement r1 and set flags
BNE loop \( \Rightarrow \) if Z flag clear then branch
```
### Conditional execution examples

<table>
<thead>
<tr>
<th>C source code</th>
<th>ARM instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (r0 == 0)</td>
<td>unconditional</td>
</tr>
<tr>
<td>{</td>
<td>CMP r0, #0</td>
</tr>
<tr>
<td>r1 = r1 + 1;</td>
<td>BNE else</td>
</tr>
<tr>
<td>} else</td>
<td>ADD r1, r1, #1</td>
</tr>
<tr>
<td>{</td>
<td>B end</td>
</tr>
<tr>
<td>r2 = r2 + 1;</td>
<td>else</td>
</tr>
<tr>
<td>}</td>
<td>ADD r2, r2, #1</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conditional</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp r0, #0</td>
<td></td>
</tr>
<tr>
<td>ADDEQ r1, r1, #1</td>
<td></td>
</tr>
<tr>
<td>ADDNE r2, r2, #1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- 5 instructions
- 5 words
- 5 or 6 cycles
- 3 instructions
- 3 words
- 3 cycles
ARM Instruction Set (3)

Data processing instructions

Block transfer instructions

Multiply instructions

Data transfer instructions

Branching instructions

Software interrupt instructions

Data Movement: 45.28%  Logical: 3.91%
Flow Control: 28.73%  Shift: 2.92%
Arithmetic: 10.75%  Bit Manipulation: 2.05%
Compare: 5.92%  I/O & Others: 0.44%
### Structural view of ARM ISA

<table>
<thead>
<tr>
<th>Condition</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift Amount</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data processing immediate shift</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous instructions:</strong></td>
<td><strong>See Figure 3-3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>data processing register shift</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous instructions:</strong></td>
<td><strong>See Figure 3-3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multipies, extra load/stores:</strong></td>
<td><strong>See Figure 3-2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data processing immediato</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Undefined instruction</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Move immediate to status register</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Load/store immediate offset</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load/store register offset</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Undefined instruction</strong></td>
<td><strong>cond [1]</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Undefined instruction [4,7]</strong></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Load/store multiple</strong></td>
<td><strong>cond [1]</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Undefined instruction [4]</strong></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Branch and branch with link</strong></td>
<td><strong>cond [1]</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch and branch with link and change to Thumb [4]</strong></td>
<td><strong>cond [1]</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Coprocessor load/store and double register transfers [4]</strong></td>
<td><strong>cond [5]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Coprocessor data processing</strong></td>
<td><strong>cond [6]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Coprocessor register transfers</strong></td>
<td><strong>cond [5]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Software interrupt</strong></td>
<td><strong>cond [1]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Undefined instruction [4]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Processing Instructions

• Arithmetic and logical operations

• 3-address format:
  – Two 32-bit operands
    (op1 is register, op2 is register or immediate)
  – 32-bit result placed in a register

• Barrel shifter for op2 allows full 32-bit shift within instruction cycle
Data Processing Instructions (2)

• Arithmetic operations:
  – ADD, ADDC, SUB, SUBC, RSB, RSC

• Bit-wise logical operations:
  – AND, EOR, ORR, BIC

• Register movement operations:
  – MOV, MVN

• Comparison operations:
  – TST, TEQ, CMP, CMN
Data Processing Instructions (3)

Conditional codes

+ Data processing instructions

+ Barrel shifter

= Powerful tools for efficient coded programs
Data Processing Instructions (4)

e.g.:

if (z==1) R1=R2+(R3*4)

compiles to

EQADDS R1,R2,R3, LSL #2

( SINGLE INSTRUCTION ! )
Multiply Instructions

- Integer multiplication (32-bit result)
- Long integer multiplication (64-bit result)
- Built in Multiply Accumulate Unit (MAC)
- Multiply and accumulate instructions add product to running total
Saturated Arithmetic

The QADD and QSUB instructions apply the specified add or subtract, and then saturate the result to the signed range $-2^{n-1} \leq x \leq 2^{n-1}-1$.

For signed $n$-bit saturation, this means that:
- if the value to be saturated is less than $-2^{n-1}$, the result returned is $-2^{n-1}$
- if the value to be saturated is greater than $2^{n-1}-1$, the result returned is $2^{n-1}-1$
- otherwise, the result returned is the same as the value to be saturated.

For unsigned $n$-bit saturation, this means that:
- if the value to be saturated is less than 0, the result returned is 0
- if the value to be saturated is greater than $2^{n-1}$, the result returned is $2^{n-1}$
- otherwise, the result returned is the same as the value to be saturated.

If the returned result is different from the value to be saturated, it is called saturation. If saturation occurs, the instruction sets the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. To clear the Q flag to 0, you must use the MSR instruction, see MSR on page 186.

To read the state of the Q flag, use the MRS instruction, see MRS on page 185.
## Multiply Instructions

- **Instructions:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Bit Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>32-bit</td>
</tr>
<tr>
<td>MULA</td>
<td>Multiply accumulate</td>
<td>32-bit</td>
</tr>
<tr>
<td>UMULL</td>
<td>Unsigned multiply</td>
<td>64-bit</td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unsigned multiply accumulate</td>
<td>64-bit</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed multiply</td>
<td>64-bit</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed multiply accumulate</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
MUL, MULA

- Multiply, multiply accumulate

![Diagram of MUL and MLA instructions]

MUL\{cond\}\{S\} Rd, Rm, Rs
MLA\{cond\}\{S\} Rd, Rm, Rs, Rn

{cond}
set condition codes if S present

{S}
Rd, Rm, Rs and Rn are expressions evaluating to a register number other than R15.

two-character condition mnemonic. See Table 4-2: Condition code summary on page 4-5.
Data Transfer Instructions

- Load/store instructions
- Used to move signed and unsigned Word, Half Word and Byte to and from registers
- Can be used to load PC
  (if target address is beyond branch instruction range)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load Word</td>
<td>STR</td>
<td>Store Word</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Half Word</td>
<td>STRH</td>
<td>Store Half Word</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load Signed Half Word</td>
<td>STRSH</td>
<td>Store Signed Half Word</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Byte</td>
<td>STRB</td>
<td>Store Byte</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load Signed Byte</td>
<td>STRSB</td>
<td>Store Signed Byte</td>
</tr>
</tbody>
</table>
(op2): Memory Addressing Mode

Used when accessing memory

Reading (Loading) data from memory:
\[ \text{DESTINATION} \leftarrow \text{M(SOURCE)} \]
\text{DESTINATION} must be a register \text{SOURCE} is any (op2) value

\[ \text{LDR } r1, [r12] \quad \text{R1} \leftarrow \text{M(R12)} \]

Writing (Storing) data into memory
\[ \text{M(DESTINATION)} \leftarrow \text{SOURCE} \]
\text{SOURCE} must be a register \text{DESTINATION} is any (op2) value

\[ \text{STR } r1, [r12] \quad \text{M(R12)} \leftarrow \text{R1} \]

Store is the only ARM instruction to place the \text{SOURCE} before the \text{DESTINATION}
Memory Addressing (Syntax)

- **Offset Addressing**
  
  \[
  [Rn, \ (value)] \quad \text{Offset Immediate}
  
  [Rn, \ Rm] \quad \text{Offset Register}
  
  [Rn, \ Rm, \ (shift) \ (value)] \quad \text{Offset scaled}
  
- **Pre-Index Addressing**
  
  \[
  [Rn, \ (value)]! \quad \text{Pre-Index Immediate}
  
  [Rn, \ Rm]! \quad \text{Pre-Index Register}
  
  [Rn, \ Rm, \ (shift) \ (value)]! \quad \text{Pre-Index scaled}
  
- **Post-Index Addressing**
  
  \[
  [Rn], \ (value) \quad \text{Post-Index Immediate}
  
  [Rn], \ Rm \quad \text{Post-Index Register}
  
  [Rn], \ Rm, \ (shift) \ (value) \quad \text{Post-Index scaled}
Memory Addressing (RTL)

- **Offset Addressing:** `LDR R0, [R1, R2]
  \[(op2) \leftarrow R1 + R2 \]
  \[MBR \leftarrow M((op2)) \]
  \[R0 \leftarrow MBR \]

- **Pre-Index Addressing:** `LDR R0, [R1, R2]!
  \[(op2) \leftarrow R1 + R2 \]
  \[R1 \leftarrow (op2) \]
  \[MBR \leftarrow M((op2)) \]
  \[R0 \leftarrow MBR \]

- **Post-Index Addressing:** `LDR R0, [R1], R2
  \[(op2) \leftarrow R1 \]
  \[R1 \leftarrow R1 + R2 \]
  \[MBR \leftarrow M((op2)) \]
  \[R0 \leftarrow MBR \]`
Memory Addressing (RTL)

- **Offset Addressing:** \( \text{LDR R0, [R1, R2]} \)
  
  \[
  (op2) \gets R1 + R2 \\
  \text{MBR} \gets M((op2)) \\
  \text{R0} \gets \text{MBR}
  \]

- **Pre-Index Addressing:** \( \text{LDR R0, [R1, R2]} \)
  
  \[
  (op2) \gets R1 + R2 \\
  \text{R1} \gets (op2) \\
  \text{MBR} \gets M((op2)) \\
  \text{R0} \gets \text{MBR}
  \]

- **Post-Index Addressing:** \( \text{LDR R0, [R1], R2} \)
  
  \[
  (op2) \gets R1 \\
  \text{R1} \gets R1 + R2 \\
  \text{MBR} \gets M((op2)) \\
  \text{R0} \gets \text{MBR}
  \]
Memory Addressing (RTL)

- **Offset Addressing:**
  
  \[ \text{LDR } \text{R0}, \ [\text{R1, R2}] \]
  
  \[
  (op2) \leftarrow R1 + R2 \\
  \text{MBR} \leftarrow M((op2)) \\
  \text{R0} \leftarrow \text{MBR}
  \]

- **Pre-Index Addressing:**
  
  \[ \text{LDR } \text{R0}, \ [\text{R1, R2}]! \]
  
  \[
  (op2) \leftarrow R1 + R2 \\
  \text{R1} \leftarrow (op2) \\
  \text{MBR} \leftarrow M((op2)) \\
  \text{R0} \leftarrow \text{MBR}
  \]

- **Post-Index Addressing:**
  
  \[ \text{LDR } \text{R0}, \ [\text{R1}], \text{R2} \]
  
  \[
  (op2) \leftarrow R1 \\
  \text{R1} \leftarrow R1 + R2 \\
  \text{MBR} \leftarrow M((op2)) \\
  \text{R0} \leftarrow \text{MBR}
  \]
Memory Addressing (RTL)

- **Offset Addressing:** \( \text{LDR} \ \text{R0, [R1, R2]} \)
  
  \[
  \begin{align*}
  \text{op2} & \leftarrow R1 + R2 \\
  \text{MBR} & \leftarrow M((\text{op2})) \\
  \text{R0} & \leftarrow \text{MBR}
  \end{align*}
  \]

- **Pre-Index Addressing:** \( \text{LDR} \ \text{R0, [R1, R2]}! \)
  
  \[
  \begin{align*}
  \text{op2} & \leftarrow R1 + R2 \\
  \text{R1} & \leftarrow (\text{op2}) \\
  \text{MBR} & \leftarrow M((\text{op2})) \\
  \text{R0} & \leftarrow \text{MBR}
  \end{align*}
  \]

- **Post-Index Addressing:** \( \text{LDR} \ \text{R0, [R1], R2} \)
  
  \[
  \begin{align*}
  \text{op2} & \leftarrow R1 \\
  \text{R1} & \leftarrow R1 + R2 \\
  \text{MBR} & \leftarrow M((\text{op2})) \\
  \text{R0} & \leftarrow \text{MBR}
  \end{align*}
  \]
<offset> options

• An immediate constant
  – #10

• An index register
  – <Rm>

• A shifted index register
  – <Rm>, LSL #<shift>
Block Transfer Instructions

- Load/Store Multiple instructions (LDM/STM)
- Whole register bank or a subset copied to memory or restored with single instruction
Swap Instruction

• Exchanges a word between registers
  • Two cycles but single atomic action
• Support for RT semaphores
Modifying the Status Registers

- Only indirectly
- *MSR* moves contents from CPSR/SPSR to selected GPR
- *MRS* moves contents from selected GPR to CPSR/SPSR
- Only in privileged modes
Branching Instructions

- **Branch (B):**
  jumps forwards/backwards up to 32 MB

- **Branch link (BL):**
  same + saves (PC+4) in LR

- Suitable for function call/return

- Condition codes for conditional branches
Program: sum16.s

7    Main
8    LDR    R0, =Data1  ;load the address of the lookup table
9    EOR    R1, R1, R1 ;clear R1 to store sum
10   LDR    R2, Length ;init element count
11   Loop
12   LDR    R3, [R0]  ;get the data
13   ADD    R1, R1, R3 ;add it to r1
14   ADD    R0, R0, #+4 ;increment pointer
15   SUBS   R2, R2, #1 ;decrement count with zero set
16   BNE    Loop ;if zero flag is not set, loop
19
22   Table  DCW &2040 ;table of values to be added
24   DCW    &1C22
28   TablEnd DCD 0
29
31   Length DCW (TablEnd - Table) / 4 ;because we’re having to align
Program: sum16.s

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LDR</td>
<td>R0, =Data1</td>
<td>;load the address of the lookup table</td>
</tr>
<tr>
<td>9</td>
<td>EOR</td>
<td>R1, R1, R1</td>
<td>;clear R1 to store sum</td>
</tr>
<tr>
<td>10</td>
<td>LDR</td>
<td>R2, Length</td>
<td>;init element count</td>
</tr>
<tr>
<td>11</td>
<td>Loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LDR</td>
<td>R3, [R0]</td>
<td>;get the data</td>
</tr>
<tr>
<td>13</td>
<td>ADD</td>
<td>R1, R1, R3</td>
<td>;add it to r1</td>
</tr>
<tr>
<td>14</td>
<td>ADD</td>
<td>R0, R0, +#4</td>
<td>;increment pointer</td>
</tr>
<tr>
<td>15</td>
<td>SUBS</td>
<td>R2, R2, #1</td>
<td>;decrement count with zero set</td>
</tr>
<tr>
<td>16</td>
<td>BNE</td>
<td>Loop</td>
<td>;if zero flag is not set, loop</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Table</td>
<td>DCW &amp;2040</td>
<td>;table of values to be added</td>
</tr>
<tr>
<td>24</td>
<td>DCW</td>
<td>&amp;1C22</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>TablEnd</td>
<td>DCD 0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Length</td>
<td>DCW (TablEnd - Table) / 4</td>
<td>;because we’re having to align</td>
</tr>
<tr>
<td></td>
<td>EOR</td>
<td>Quick way of setting R1 to zero</td>
<td></td>
</tr>
</tbody>
</table>
Program: sum16.s

7   Main
8       LDR   R0, =Data1  ;load the address of the lookup table
9       EOR   R1, R1, R1  ;clear R1 to store sum
10      LDR   R2, Length  ;init element count
11   Loop
12      LDR   R3, [R0]   ;get the data
13      ADD   R1, R1, R3  ;add it to r1
14      ADD   R0, R0, #+4 ;increment pointer
15      SUBS  R2, R2, #1  ;decrement count with zero set
16      BNE   Loop       ;if zero flag is not set, loop
19
22   Table  DCW   &2040  ;table of values to be added
24      DCW   &1C22
28   TablEnd DCD   0
29
31   Length  DCW   (TablEnd - Table) / 4 ;because we're having to align

Loop   Label the next instruction
Program: sum16.s

7 Main
8 LDR R0, =Data1 ;load the address of the lookup table
9 EOR R1, R1, R1 ;clear R1 to store sum
10 LDR R2, Length ;init element count
11 Loop
12 LDR R3, [R0] ;get the data
13 ADD R1, R1, R3 ;add it to r1
14 ADD R0, R0, +#4 ;increment pointer
15 SUBS R2, R2, #1 ;decrement count with zero set
16 BNE Loop ;if zero flag is not set, loop

22 Table DCW &2040 ;table of values to be added
24 DCW &1C22
28 TablEnd DCD 0
29
31 Length DCW (TablEnd - Table) / 4 ;because we’re having to align
ADD Move pointer (R0) to next word
Program: sum16.s

7  Main
8  LDR R0, =Data1 ;load the address of the lookup table
9  EOR R1, R1, R1 ;clear R1 to store sum
10 LDR R2, Length ;init element count
11 Loop
12 LDR R3, [R0] ;get the data
13 ADD R1, R1, R3 ;add it to r1
14 ADD R0, R0, #+4 ;increment pointer
15 SUBS R2, R2, #1 ;decrement count with zero set
16 BNE Loop ;if zero flag is not set, loop
19
22 Table DCW &2040 ;table of values to be added
24 DCW &1C22
28 TableEnd DCD 0
31 Length DCW (TableEnd - Table) / 4 ;because we’re having to align

LDR/ADD Using Post-index addressing we can remove the ADD:
   LDR R3, [R0], #4
Program: sum16.s

7  Main
8     LDR  R0, =Data1 ;load the address of the lookup table
9     EOR  R1, R1, R1 ;clear R1 to store sum
10    LDR  R2, Length ;init element count
11   Loop
12    LDR  R3, [R0] ;get the data
13    ADD  R1, R1, R3 ;add it to r1
14    ADD  R0, R0, #+4 ;increment pointer
15 SUBS  R2, R2, #1 ;decrement count with zero set
16    BNE  Loop ;if zero flag is not set, loop
19
22   Table  DCW  &2040 ;table of values to be added
24    DCW  &1C22
28  TablEnd  DCD  0
29
31   Length  DCW  (TablEnd - Table) / 4 ;because we’re having to align

SUBS  Subtract and set flags
Decrement loop counter, R2
### Program: sum16.s

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Main</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>LDR R0, =Data1</td>
<td>load the address of the lookup table</td>
</tr>
<tr>
<td>9</td>
<td>EOR R1, R1, R1</td>
<td>clear R1 to store sum</td>
</tr>
<tr>
<td>10</td>
<td>LDR R2, Length</td>
<td>init element count</td>
</tr>
<tr>
<td>11</td>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LDR R3, [R0]</td>
<td>get the data</td>
</tr>
<tr>
<td>13</td>
<td>ADD R1, R1, R3</td>
<td>add it to r1</td>
</tr>
<tr>
<td>14</td>
<td>ADD R0, R0, #+4</td>
<td>increment pointer</td>
</tr>
<tr>
<td>15</td>
<td>SUBS R2, R2, #1</td>
<td>decrement count with zero set</td>
</tr>
<tr>
<td>16</td>
<td>BNE Loop</td>
<td>if zero flag is not set, loop</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Table DCW &amp;2040</td>
<td>table of values to be added</td>
</tr>
<tr>
<td>24</td>
<td>DCW &amp;1C22</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>TablEnd DCD 0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Length DCW (TablEnd - Table) / 4</td>
<td>because we’re having to align</td>
</tr>
<tr>
<td></td>
<td>BNE Branch to Loop if counter is not equal to zero</td>
<td></td>
</tr>
</tbody>
</table>
Program: sum16.s

7  Main
8    LDR R0, =Data1 ; load the address of the lookup table
9    EOR R1, R1, R1 ; clear R1 to store sum
10   LDR R2, Length ; init element count
11  Loop
12   LDR R3, [R0] ; get the data
13   ADD R1, R1, R3 ; add it to r1
14   ADD R0, R0, #+4 ; increment pointer
15   SUBS R2, R2, #1 ; decrement count with zero set
16   BNE Loop ; if zero flag is not set, loop

22  Table DCW &2040 ; table of values to be added
24   DCW &1C22
28  TablEnd DCD 0

31  Length DCW (TablEnd - Table) / 4 ; because we’re having to align

DCW Assembler will calculate the length of data table for me
Program: sum16b.s

8  Main
9    LDR  R0, =Data1 ; load the address of the lookup table
10   EOR  R1, R1, R1 ; clear R1 to store sum
11   LDR  R2, Length ; init element count
12   CMP  R2, #0 ; zero length table?
13   BEQ  Done ; yes => skip over sum loop
14  Loop
15   LDR  R3, [R0] ; get the data that R0 points to
16   ADD  R1, R1, R3 ; add it to R1
17   ADD  R0, R0, #+4 ; increment pointer
18   SUBS R2, R2, #0x1 ; decrement count with zero set
19   BNE  Loop ; if zero flag is not set, loop
20  Done
21   STR  R1, Result ; otherwise done - store result
22   SWI  &11
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Main</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>LDR R0, =Data1</td>
<td>;load the address of the lookup table</td>
</tr>
<tr>
<td>10</td>
<td>EOR R1, R1, R1</td>
<td>;clear R1 to store sum</td>
</tr>
<tr>
<td>11</td>
<td>LDR R2, Length</td>
<td>;init element count</td>
</tr>
<tr>
<td>12</td>
<td>CMP R2, #0</td>
<td>;zero length table ?</td>
</tr>
<tr>
<td>13</td>
<td>BEQ Done</td>
<td>;yes =&gt; skip over sum loop</td>
</tr>
<tr>
<td>14</td>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>LDR R3, [R0]</td>
<td>;get the data that R0 points to</td>
</tr>
<tr>
<td>16</td>
<td>ADD R1, R1, R3</td>
<td>;add it to R1</td>
</tr>
<tr>
<td>17</td>
<td>ADD R0, R0, #+4</td>
<td>;increment pointer</td>
</tr>
<tr>
<td>18</td>
<td>SUBS R2, R2, #0x1</td>
<td>;decrement count with zero set</td>
</tr>
<tr>
<td>19</td>
<td>BNE Loop</td>
<td>;if zero flag is not set, loop</td>
</tr>
<tr>
<td>20</td>
<td>Done</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>STR R1, Result</td>
<td>;otherwise done - store result</td>
</tr>
<tr>
<td>22</td>
<td>SWI &amp;11</td>
<td></td>
</tr>
</tbody>
</table>

EOR Quick way of setting R1 to zero
Program: sum16b.s

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Main</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>LDR</td>
<td>load the address of the lookup table</td>
</tr>
<tr>
<td>10</td>
<td>EOR</td>
<td>clear R1 to store sum</td>
</tr>
<tr>
<td>11</td>
<td>LDR</td>
<td>init element count</td>
</tr>
<tr>
<td>12</td>
<td>CMP</td>
<td>zero length table ?</td>
</tr>
<tr>
<td>13</td>
<td>BEQ</td>
<td>done if zero flag is not set, loop</td>
</tr>
<tr>
<td>14</td>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>LDR</td>
<td>get the data that R0 points to</td>
</tr>
<tr>
<td>16</td>
<td>ADD</td>
<td>add it to R1</td>
</tr>
<tr>
<td>17</td>
<td>ADD</td>
<td>increment pointer</td>
</tr>
<tr>
<td>18</td>
<td>SUBS</td>
<td>decrement count with zero set</td>
</tr>
<tr>
<td>19</td>
<td>BNE</td>
<td>if zero flag is set, loop</td>
</tr>
<tr>
<td>20</td>
<td>Done</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>STR</td>
<td>otherwise done - store result</td>
</tr>
<tr>
<td>22</td>
<td>SWI</td>
<td></td>
</tr>
</tbody>
</table>

CMP

Is table length zero?
## Program: sum16b.s

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Main</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>LDR R0, =Data1</td>
<td>load the address of the lookup table</td>
</tr>
<tr>
<td>10</td>
<td>EOR R1, R1, R1</td>
<td>clear R1 to store sum</td>
</tr>
<tr>
<td>11</td>
<td>LDR R2, Length</td>
<td>init element count</td>
</tr>
<tr>
<td>12</td>
<td>CMP R2, #0</td>
<td>zero length table?</td>
</tr>
<tr>
<td>13</td>
<td>BEQ Done</td>
<td>yes =&gt; skip over sum loop</td>
</tr>
<tr>
<td>14</td>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>LDR R3, [R0]</td>
<td>get the data that R0 points to</td>
</tr>
<tr>
<td>16</td>
<td>ADD R1, R1, R3</td>
<td>add it to R1</td>
</tr>
<tr>
<td>17</td>
<td>ADD R0, R0, #+4</td>
<td>increment pointer</td>
</tr>
<tr>
<td>18</td>
<td>SUBS R2, R2, #0x1</td>
<td>decrement count with zero set</td>
</tr>
<tr>
<td>19</td>
<td>BNE Loop</td>
<td>if zero flag is not set, loop</td>
</tr>
<tr>
<td>20</td>
<td>Done</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>STR R1, Result</td>
<td>otherwise done - store result</td>
</tr>
<tr>
<td>22</td>
<td>SWI &amp;11</td>
<td></td>
</tr>
</tbody>
</table>

BEQ Skip zero length tables
Protects from processing an empty list
Program: sum16b.s

8  Main
9   LDR   R0, =Data1 ;load the address of the lookup table
10  EOR   R1, R1, R1 ;clear R1 to store sum
11  LDR   R2, Length ;init element count
12  CMP   R2, #0 ;zero length table ?
13  BEQ   Done ;yes => skip over sum loop
14  Loop
15  LDR   R3, [R0] ;get the data that R0 points to
16  ADD   R1, R1, R3 ;add it to R1
17  ADD   R0, R0, #+4 ;increment pointer
18  SUBS  R2, R2, #0x1 ;decrement count with zero set
19  BNE   Loop ;if zero flag is not set, loop
20  Done
21  STR   R1, Result ;otherwise done - store result
22  SWI   &11

Using Post-index addressing we can remove the ADD:
LDR  R3, [R0] + #4
Program: sum16b.s

8   Main
9    LDR    R0, =Data1 ;load the address of the lookup table
10   EOR    R1, R1, R1 ;clear R1 to store sum
11   LDR    R2, Length ;init element count
12   CMP    R2, #0    ;zero length table ?
13   BEQ    Done    ;yes => skip over sum loop

14   Loop
15   LDR    R3, [R0]  ;get the data that R0 points to
16   ADD    R1, R1, R3 ;add it to R1
17   ADD    R0, R0, +#4 ;increment pointer
18   SUBS   R2, R2, +#0x1 ;decrement count with zero set
19   BNE    Loop      ;if zero flag is not set, loop

20  Done
21   STR    R1, Result ;otherwise done - store result
22   SWI    &11

SUBS/BNE  Decrement counter and branch to Loop if not zero
IF-THEN Instruction

• Another alternative to execute conditional code is the new 16-bit IF-THEN (IT) instruction
  – no change in program flow
  – no branching overhead
• Can use with 32-bit Thumb-2 instructions that do not support the ‘S’ suffix
• Example:
  
  \[
  \begin{align*}
  &\text{CMP R1, R2} \quad ; \text{If R1 = R2} \\
  &\text{IT EQ} \quad ; \text{execute next (1st)} \\
  &\quad ; \text{instruction} \\
  &\text{ADDEQ R2, R1, R0} \quad ; \text{1st instruction}
  \end{align*}
  \]
• The conditional codes can be extended up to 4 instructions
Software Interrupt

• *SWI* instruction
  – Forces CPU into supervisor mode
  – Usage: SWI #n

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>Opcode</td>
<td>Ordinal</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Maximum $2^{24}$ calls
• Suitable for running privileged code and making OS calls
## Barrier instructions

- Useful for multi-core & Self-modifying code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMB</td>
<td>Data memory barrier; ensures that all memory accesses are completed before new memory access is committed</td>
</tr>
<tr>
<td>DSB</td>
<td>Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions</td>
</tr>
</tbody>
</table>