About The ARM Processor

Advanced Risc Machine

Yongjin Kim
CASP Lab. Hanyang Univ.
yjkim@casp.hanyang.ac.kr
ARM Ltd

- Founded in November 1990
  - Spun out of Acorn Computers

- Designs the ARM range of RISC processor cores

- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers
  - ARM does not fabricate silicon itself

- Also develop technologies to assist with the design-in of the ARM architecture
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc

- 1985
  - Acorn Computer Group develops the world's first commercial RISC processor
- 1987
  - Acorn's ARM processor debuts as the first RISC processor for low-cost PCs
- 1990
  - Advanced RISC Machines (ARM) spins out of Acorn and Apple Computer's collaboration efforts with a charter to create a new microprocessor standard. VLSI Technology becomes an investor and the first licensee
- 1993
  - ARM introduces the ARM7™ core
- 1994
  - Samsung license ARM technology
  - ARM introduces the ARM7500™ "system-chip" for multimedia applications
- 1995
  - ARM's Thumb® architecture extension gives 32-bit RISC performance at 16-bit system cost and offers industry-leading code density
  - ARM launches Software Development Toolkit
  - First StrongARM™ core from Digital Semiconductor and ARM
  - ARM extends family with ARM8™ high-performance solution
  - ARM launches the ARM7100™ "PDA-on-a-chip"

1996
- ARM and VLSI Technology introduce the ARM810™ microprocessor
- ARM announces ARM7500FETM multimedia "system-chip" for the Network Computer
- Virtual Socket Interface Alliance (VSIA) is formed, with ARM as a charter member
- ARM and Microsoft work together to extend Windows CE to the ARM architecture

1997
- ARM and Sun offer direct JavaOS support for ARM RISC architecture
- ARM introduces high-performance, embedded software modem solutions
- ARM9TDMI™ family announced

1998
- Qualcomm license ARM technology
- ARM joins Bluetooth™ consortium
- Intel licenses StrongARM microprocessor family
- ARM introduces ARM910™ and ARM920™ solutions for Windows CE applications
- ARM develops synthesisable version of the ARM7TDMI® core
- ARM announces ARM10™ Thumb family of processors
- ARM establishes consortium for Windows CE
- ARM Partners ship more than 50 million ARM Powered® products
ARM HISTORY (1999 – 2001)

1999
- ARM announces synthesizable ARM9E™ processor with enhanced signal processing
- ARM works with Microsoft to optimize Windows Media Audio
- ARM introduces MP3 and Dolby Digital Software solutions

2000
- ARM launches SecurCore™ family for smartcards
- ARM introduces Jazelle™ technology for Java™ applications
- ARM introduces ARM922T™ core
- Lucent ships ARM10™ silicon
- ARM supports Intel's launch of ARM architecture-compliant XScale™ microarchitecture

2001
- ARM announces new ARMv6 architecture
- ARM announces the ARM VFP9-ST™ and ARM VFP10™ vector floating-point coprocessors
- ARM introduces the ARM926EJ-S™ soft macrocell, the ARM7EJ™ core, and the Jazelle Technology Enabling Kit™ (JTEK)SavaJe and Symbian license Jazelle™ software for incorporation into their operating systems
- ARM announces the PrimeXsys™ platforms.
- ARM introduces the SecurCore SC200™ and SC210™ microprocessor cores
- ARM announces MOVE™ multimedia acceleration technology for wireless devices
ARM POWERED PRODUCTS

- Samsung ML5100A
- Diamond Multimedia Rio 600
- JVC "Pixstar" GC-X1
- Alba Bush Internet TV
- 3Com 10/100 PCI NIC
- Nintendo Gameboy Advance
- Iomega HipZip
- Sony MZ-R90 MiniDisc
- Lexmark Z52 Color Jetprinter
- Ericsson R380
- HP Jornada 820
- Psion Revo Plus
- Nokia 8810
- HP CapShare
- Nokia Mediamaster
ARM FEATURE

- RISC (Reduced Instruction Set Computer) architecture
- General purpose 32-bit microprocessors
- Very low power consumption and price
- Big/Little Endian Mode
  - Intel x86 – Little Endian
  - Motorola – Big Endian
- Fast Interrupt Response
  - FIQ Mode
- Excellent High Level Language Support
  - C Language
  - Auto Increment/Decrement addressing mode
- Simple and powerful Instruction Set
  - Load/Store multiple instructions
  - Conditional execution
## ARM ARCHITECTURE

<table>
<thead>
<tr>
<th>Architecture</th>
<th>THUMB</th>
<th>DSP</th>
<th>Jazelle</th>
<th>Media</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4T</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V5TE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V5TEJ</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>V6</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
ARM ARCHITECTURE

- v4T
  - **Thumb** instruction set
  - Short 16-bit instructions
  - Typical memory savings of up to 35%
- 5TE
  - **DSP** instruction set
  - 70% speed up (audio DSP applications)
- v5TEJ
  - **Jazelle** technology for java
  - 80% reduction
- V6
  - **SIMD** extensions
The Thumb® Architecture Extension

- Excellent code-density for minimal system memory size and cost
- 32-bit performance from 8 or 16-bit memory on an 8 or 16-bit bus for low system cost.
ARM DSP Instruction Set Extensions

- Single-cycle 16x16 and 32x16 MAC implementations
- 2-3 x DSP performance improvement on ARM9TDMI based CPU products
- Zero overhead saturation extension support
- New instructions to load and store pairs of registers, with enhanced addressing modes
- New CLZ instruction improves normalisation in arithmetic operations and improves divide performance
- Full support in the ARMv5TE and ARMv6 architecture
The ARM Jazelle technology provides a highly-optimized implementation of the Java Virtual Machine (JVM), speeding up execution times and providing consumers with an enriched user experience on their mobile devices.
ARM Media Extensions

- **Overview**
  - streaming media performance (film, video phone, music and more),
  - more human-oriented interfaces (voice and handwriting recognition)

- **Feature**
  - 2-4x performance increase for audio and video processing
  - Simultaneous computation of 2x16-bit or 4x8-bit operands
  - Fractional arithmetic
  - User definable saturation modes (arbitrary word-width)
  - Dual 16x16 multiply-add/subtract
  - 32x23 fractional MAC
  - Simultaneous 8/16-bit select operations
ARM7 FAMILY (60-100MIPS)

- ARM740T
  - Embedded RTOS core
  - 60-75Mhz (0.18um w/c)
  - Hard Macro IP

- ARM7-S
  - Base Integer core
  - 100Mhz (0.18um w/c)
  - Synthesizable (Soft IP)

- ARM7
  - Base Integer core (Hard Macro IP)
  - 60-100Mhz (0.18um w/c)
  - NEW: low-voltage layout

- ARM7TDMI
  - ARM v4T ISA
  - Von Neumann Architecture
  - Thumb Support
  - Debug Support
  - 3-Stage Pipeline
  - 0.9 MIPS/Mhz

- ARM7TDMI-S
  - Platform OS core
  - 60-75Mhz (0.18um w/c)
  - 0.65mW/Mhz
  - Hard Macro IP

- ARM720T
  - ARM v4T ISA
  - Von Neumann Architecture
  - Thumb Support
  - Debug Support
  - 3-Stage Pipeline
  - 0.9 MIPS/Mhz
ARM9 FAMILY (200 MIPS)

- **ARM922T**
  - Platform OS processor core
  - Up to 200Mhz (0.18um w/c)
  - 0.8mW/Mhz

- **ARM920T**
  - Platform OS processor core
  - Up to 200Mhz (0.18um w/c)
  - 0.8mW/Mhz

- **ARM940T**
  - Embedded RTOS processor core
  - Up to 180Mhz (0.18um w/c)
  - 0.85mW/Mhz

- **ARM9TDMI**
  - Base Integer processor core
  - Up to 220Mhz (0.18um w/c)
  - 0.3mW/Mhz

- **ARM9T**
  - Hard macro IP
  - ARM v4T ISA
  - Harvard architecture
  - Thumb Support
  - Debug Support
  - 5-Stage Pipeline
  - Dual Caches
  - AMBA ASB bus
  - 1.1 MIPS/Mhz

- **PU**
  - 4K Caches
  - ARM9T
  - ASB i/face

- **MMU**
  - 16K Caches
  - ARM9T
  - EMT7 i/face
  - ASB i/face
ARM9E FAMILY (200 MIPS + DSP)

ARM946E-S
- Embedded OS processor core
  - 160MIPS (0.18um w/c)
  - ~140K gates + SRAM

Flexible SRAMs
- ARM9E-S
- ETM9 i/face
- AHB i/face

Flexible Caches
- ARM9E-S
- EMT9 i/face
- AHB i/face

ARM966E-S
- Embedded processor core
  - 180MIPS (0.18um w/c)
  - ~110K gates + SRAM

ARM9E-S
- Base Integer processor core
  - 180MIPS (0.18um w/c)
  - ~65K gates

PU
- TCM i/face
- Flexible Caches
- ARM9E-S
- EMT9 i/face
- AHB i/face

MMU
- TCM i/face
- Flexible Caches
- ARM9E-S
- EMT7 i/face
- AHB i/face

Jazelle™ Enhanced

ARM926EJ-S
- Platform OS processor core
  - 200MIPS (0.18um w/c)
  - ~190K gates + SRAM

- Synthesizable (Soft IP)
- ARM v5TE ISA
- Thumb Support
- Debug Support
- 5-Stage Pipeline
- Dual Caches
- AMBA AHB bus
ARM10E FAMILY (390 MIPS+DSP)

- **ARM1020E**
  - Platform OS core
  - 300Mhz (0.15um w/c)
  - Hard Macro IP

- **ARM946E-S**
  - Platform OS core
  - 300Mhz (0.15um w/c)
  - Hard Macro IP

- **ARM966E-S**
  - 600Mflops Vector floating Point coprocessor

- **VFP10**
  - Jazelle™ Enhanced

- **ARM10J**
  - 16K/32K Cache
  - EMT10 i/face
  - Dual AHB

- **MMU**
  - 16K/32K Cache
  - EMT10 i/face
  - Dual AHB

- **ARM966E-S**
  - Platform OS core
  - >400Mhz (0.13um w/c)
  - Hard Macro IP

- **ARM946E-S**
  - Platform OS core
  - 300Mhz (0.15um w/c)
  - Hard Macro IP

- **ARM966E-S**
  - 600Mflops Vector floating Point coprocessor

- **ARM966E-S**
  - ARM v5TE ISA
  - Thumb Code Compression
  - Debug Support
  - 6-Stage Pipeline
  - Dual Cache architecture
  - Dual 64-bit AHB bus
  - Optional VFP coprocessor
  - AMBA AHB bus
  - 1.3 MIPS/Mhz
  - Low power: 0.7mW/MIPS
ARM11 FAMILY (<400 MIPS+DSP+ Jazelle™)

ARM1136JF-S
- Platform OS core
- 600Mflops Vector floating
- Point coprocessor

ARM1136J-S
- Platform OS core
- Up to 1Ghz (0.15um w/c)

ARM11
- Platform OS core
- Up to 1Ghz (0.15um w/c)

- ARM v6 ISA
- Thumb Code Compression
- DSP extensions
- 8-Stage Pipeline (35% enhanced)
- SIMD media processing
- Multi-port 64bit memory system
- Optional VFP coprocessor
- TCM (scratch-pad) memory
- Low power: 0.4mW/Mhz
<table>
<thead>
<tr>
<th>MIPS</th>
<th>Basic Cores</th>
<th>Un-cached Cores For Hard real-time Applications</th>
<th>Cached Cores With Embedded real-time applications</th>
<th>Cached Cores With full MMU OS Support for e.g. winCE, Symbian OS, Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>ARM9TDMI</td>
<td>ARM1136(J)-S</td>
<td>ARM1136(J)-S</td>
<td>ARM1020E</td>
</tr>
<tr>
<td>200</td>
<td>ARM9TDMI</td>
<td>ARM966E-S</td>
<td>ARM940T</td>
<td>ARM920T</td>
</tr>
<tr>
<td></td>
<td>ARM9E-S</td>
<td></td>
<td>ARM946E-S</td>
<td>ARM926E-S</td>
</tr>
<tr>
<td>100</td>
<td>ARM7TDMI</td>
<td></td>
<td>ARM740T</td>
<td>ARM720T</td>
</tr>
<tr>
<td></td>
<td>ARM7TDMI-S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ARM7TDMI Core diagram
ARM

INSTRUCTIONS
Registers

- 31 general-purpose 32-bit Registers
- 6 Status Registers
- 6 mode banked (Remapping)
- 16 visible Register (R0 – R15)
  - R0-R13: General purpose registers (R13: Stack Pointer (SP))
  - R14: Link register (LR)
  - R15: Program counter (PC)
  - CPSR: Current program status register
  - SPSR: Saved program status register
    (only accessible in privileged modes)
- Conditional execution of instructions
  - ARM: which executes 32-bit, word-aligned ARM instructions
  - Thumb: which operates with 16-bit, halfword-aligned THUMB instruction (PC uses bit 1 to select between alternate halfwords)
Program Status Register Format

<table>
<thead>
<tr>
<th>Condition Code Flags</th>
<th>Reserved</th>
<th>Control Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27</td>
<td>8 7 6 5 4</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td>N Z C V</td>
<td></td>
<td>I F T M M M M</td>
</tr>
</tbody>
</table>

- **Condition Code Flags** (arithmetic and logical operations)
  - N: Negative/Less Than
  - Z: Zero
  - C: Carry/Borrow/Extend
  - V: Overflow

- **Control bits**
  - I: IRQ disable (Active Set)
  - F: FIQ disable (Active Set)
  - T: State bit (1: Thumb state 0: ARM state)

- **Mode bits**
  - User, FIQ, IRQ, Supervisor, Abort, Undefined
# Processor Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Value M[4:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User(usr)</td>
<td>10000</td>
<td>Normal program execute mode</td>
</tr>
<tr>
<td>FIQ(fiq)</td>
<td>10001</td>
<td>Supports a high-speed data transfer or channel process</td>
</tr>
<tr>
<td>IRQ(irq)</td>
<td>10010</td>
<td>Used for general purpose interrupt handling</td>
</tr>
<tr>
<td>Supervisor(svc)</td>
<td>10011</td>
<td>A protected mode for the operating system</td>
</tr>
<tr>
<td>Abort(abt)</td>
<td>10111</td>
<td>Implements virtual memory and/or memory protection</td>
</tr>
<tr>
<td>Undefined(und)</td>
<td>11011</td>
<td>Supports software emulation of hardware coprocessors</td>
</tr>
<tr>
<td>System(sys)</td>
<td>1111</td>
<td>Runs privileged operating system tasks</td>
</tr>
</tbody>
</table>
Register organization in the ARM state

<table>
<thead>
<tr>
<th>User</th>
<th>FIQ</th>
<th>Supervisor</th>
<th>Abort</th>
<th>IRQ</th>
<th>Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R8_fiq</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
<td>R9_fiq</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
</tr>
<tr>
<td>R10</td>
<td>R10_fiq</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
</tr>
<tr>
<td>R11</td>
<td>R11_fiq</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>r12_fiq</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
</tr>
<tr>
<td>R13(SP)</td>
<td>R13_fiq</td>
<td>R13 svc</td>
<td>R13_abt</td>
<td>R13_irq</td>
<td>R13_und</td>
</tr>
<tr>
<td>R14(LR)</td>
<td>R14_fiq</td>
<td>R14 svc</td>
<td>R14_abt</td>
<td>R14_irq</td>
<td>R14_und</td>
</tr>
</tbody>
</table>

CPSR  CPSR  CPSR  CPSR  CPRR  CPSR

SPSR_fiq  SPSR_svc  SPSR_abt  SPSR_irq  SPSR_und
## Register organization in the Thumb state

<table>
<thead>
<tr>
<th>User</th>
<th>FIQ</th>
<th>Supervisor</th>
<th>Abort</th>
<th>IRQ</th>
<th>Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>Not Used</td>
<td>Not Used</td>
<td>Not Used</td>
<td>Not Used</td>
<td>Not Used</td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>SP_fiq</td>
<td>SP_svc</td>
<td>SP_abt</td>
<td>SP_irq</td>
<td>SP_und</td>
</tr>
<tr>
<td>LR</td>
<td>LR_fiq</td>
<td>LR_svc</td>
<td>LR_abt</td>
<td>LR_irq</td>
<td>LR_und</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>CPSR</td>
<td>CPSR</td>
<td>CPSR</td>
<td>CPSR</td>
<td>CPRR</td>
<td>CPSR</td>
</tr>
<tr>
<td>SPSR_fiq</td>
<td>SPSR_svc</td>
<td>SPSR_abt</td>
<td>SPSR_irq</td>
<td>SPSR_und</td>
<td>SPSR_und</td>
</tr>
</tbody>
</table>
Exception

- Interrupts, traps, supervisor calls

- Exceptions arise whenever the normal flow of program has to be halted temporarily
CISC and RISC

Pt = Ti + 2Ti + 3Ti+ … + nTi

(Pt : 프로그램 실행 시간, Ti : 명령어의 실행시간, n : 명령어의 수)

Ti를 줄이자 RISC
N을 줄이자 CISC

CISC (Complex Instruction Set Computing) 구조
- 단순한 명령 처리에서 복잡한 명령 수행까지 하나의 명령집합으로 실행할 수 있도록 여러 개의 명령어를 가진 구조
- 논리 회로 증가 → 명령어마다 실행속도가 틀리다
- 과다한 Memory Access로 속도 저하

RISC (Reduced Instruction Set Computing) 구조
- 1975년 IBM 윌슨 연구소의 John Cocke
- 명령의 종류를 최소한으로 줄임 → 논리회로 줄어듬
- 여러 개의 명령을 여러 개의 연산회로에 나누어 처리
- 파이프 라인 구조 재택
Interrupt Concept

- Hardware Interrupt
  - 외부의 사건을 CPU에게 알리는 방편
  - 인터럽트가 걸리면 CPU가 자동으로 반응하게 됨
  - 의외 - CPU가 실행중인 프로그램의 영향에서 벗어나게 함.

- Software Interrupt
  - 프로그램에서 명령에 의해 실행됨
  - 동작방법은 하드웨어 인터럽트와 유사함
  - Interrupt 서비스루틴 - 모든 program이 공유, kernel level의 프로그램

- 임 출력 연산을 한 군데서 제어하는데 유용함
  - 임 출력 장치로부터의 사건발생(키보드, 마우스, 디스크 등)
  - 비정상적인 긴급한 상황의 발생 (전원 고장, 등)
  - 불법 명령의 수행 금지( 0 으로 나누기, 없는 명령 수행 시도 등)
Interrupt - ISR 기본 시나리오

1. CPU상에서 프로그램 A가 수행 중
2. 인터럽트가 걸린 사실을 CPU가 감지
3. 현재 수행중인 명령(Ic)의 실행을 완료
4. 수행 중인 프로그램 A의 상태를 PCB나 스택에 저장
   - 프로세스 상태(process state)
     - 프로세스가 다시 복원되는데 필요한 data 집합(PC, 플래그 레지스터, 범용 레지스터 등)
     - PCB(process control block)
5. 인터럽트 서비스 루틴을 수행
   - interrupt service routine = interrupt routine = interrupt handler
6. 인터럽트 서비스 루틴의 수행을 완료
7. 수행하던 프로그램 A의 명령 Ic의 다음 명령을 수행함
   혹은, 인터럽트 서비스 루틴에서 지정한 다른 프로그램 수행
   ex) 프린터 종이가 다 떨어졌을 때, 배터리 전원이 부족한 상황
Interrupt - 용어

- Interrupt source
  - Input pin, counter, Serial port 등과 같이 Interrupt를 발생시킬 수 있는 것들

- Interrupt vector
  - Interrupt 발생시 분기 장소(Address)

- Interrupt service routine(ISR)
  - 실제 Interrupt 처리 프로그램

- Interrupt priority
  - Interrupt가 동시에 발생 할 때, 어떤 Interrupt를 먼저 처리 할 것인가를 결정하는 것
  - Interrupt priority가 높은 것이 낮은 것보다 먼저 발생한다
## Exception Vector

<table>
<thead>
<tr>
<th>Address</th>
<th>Exception</th>
<th>Mode on entry</th>
<th>Priorities</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Reset</td>
<td>Supervisor</td>
<td>1</td>
<td>When the nRESET signal goes LOW</td>
</tr>
<tr>
<td>0x00000004</td>
<td>Undefined Instruction</td>
<td>Undefined</td>
<td>6</td>
<td>When ARM comes across an undefined instruction</td>
</tr>
<tr>
<td>0x00000008</td>
<td>Software Interrupt</td>
<td>Supervisor</td>
<td>6</td>
<td>Software Program Interrupt. System call</td>
</tr>
<tr>
<td>0x0000000C</td>
<td>Abort (Pre-fetch)</td>
<td>Abort</td>
<td>5</td>
<td>When the current memory access cannot be completed</td>
</tr>
<tr>
<td>0x00000010</td>
<td>Abort (Data)</td>
<td>Abort</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0x00000014</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000018</td>
<td>IRQ</td>
<td>IRQ</td>
<td>4</td>
<td>The IRQ exception is a normal interrupt caused by a Low level on the nIRQ input. R13 – R14 remapped</td>
</tr>
<tr>
<td>0x0000001C</td>
<td>FIQ</td>
<td>FIQ</td>
<td>3</td>
<td>The FIQ exception is a fast interrupt caused by a Low level on the nFIQ input . R8 – R14 remapped</td>
</tr>
</tbody>
</table>
Exception Flow – Reset

nReset

R14_svc = PC
SPSR_svc = CPSR

Supervisor Mode Enter

M [4:0] = 10011
IRQ Mode ( M[4:0] = 10010 )
ARM state ( T = 0 )
FIQ Disable ( F = 1 )
IRQ Disable ( I = 1 )

Reset Vector Branch

PC = 0x00

Fetch

Next Instruction fetch

ARM State

Reset Exception Flow
Exception Flow – IRQ

**User Mode (R0-R15)**

- **IRQ Event**
- **Register Remapping**
  - R13 to R13_irq
  - R14 to R14_irq
- **Return Address**
  - (PC + 4) → R14_irq
- **IRQ Mode**
  - CPSR → SPSR_irq
  - IRQ Mode (M[4:0] = 10010)
  - ARM state (T = 0)
  - IRQ Disable (I = 1)
- **IRQ Vector Branch**
  - PC → 0x18

** IRQ Exception Entry **

** IRQ Service Routine execution **

** IRQ SR Entry **

** IRQ SR Exit **

** Exit **

** User Mode **

- **SPSR_irq → CPSR**

** User Branch **

- **R14_irq → PC**

** User Mode (R0-R15) **

** IRQ Exception Exit **

** IRQ Exception Flow **
Instructions Feature

- 32 bits wide
- 4-byte boundaries
- The Load-Store architecture
- 3-address data processing instructions
- Conditional execution of every instruction
- The inclusion of very powerful load and store multiple register instructions
- The ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle
- Open instruction set extension through the coprocessor instruction set. Including adding new registers and data types to the programmer’s model
- A very dense 16-bit compressed representation of the instruction set in the Thumb
### Instruction Format

<table>
<thead>
<tr>
<th>Data processing and FSR transfer</th>
<th>Cond</th>
<th>0 0 1</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>Cond</td>
<td>0 0 0 0 0 0 A S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs</td>
<td>1 0 0 1</td>
<td>Rm</td>
</tr>
<tr>
<td>Multiply long</td>
<td>Cond</td>
<td>0 0 0 0 1 U A S</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rn</td>
<td>1 0 0 1</td>
<td>Rm</td>
</tr>
<tr>
<td>Single data swap</td>
<td>Cond</td>
<td>0 0 0 0 1 0 B 0 0</td>
<td>Rn</td>
<td>Rd</td>
<td>0 0 0 0 1 0 0 0 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Branch and exchange</td>
<td>Cond</td>
<td>0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1</td>
<td>Rn</td>
<td>Rd</td>
<td>0 0 0 0 1 1 S H 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Halfword data transfer, register offset</td>
<td>Cond</td>
<td>0 0 0 P U O W L</td>
<td>Rn</td>
<td>Rd</td>
<td>0 0 0 0 1 S H 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>Halfword data transfer, immediate offset</td>
<td>Cond</td>
<td>0 0 0 P U 1 W L</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
<td>1 S H 1</td>
<td>Offset</td>
</tr>
<tr>
<td>Single data transfer</td>
<td>Cond</td>
<td>0 1 1 P U B W L</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undefined</td>
<td>Cond</td>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Block data transfer</td>
<td>Cond</td>
<td>1 0 0 P U S W L</td>
<td>Rn</td>
<td></td>
<td>Register list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>Cond</td>
<td>1 0 1 L</td>
<td></td>
<td></td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coprocessor data transfer</td>
<td>Cond</td>
<td>1 1 0 P U N W L</td>
<td>Rn</td>
<td>CRd</td>
<td>CP#</td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>Coprocessor data operation</td>
<td>Cond</td>
<td>1 1 1 0</td>
<td>CP Opc</td>
<td>CRn</td>
<td>CRd</td>
<td>CP#</td>
<td>CP 0</td>
</tr>
<tr>
<td>Coprocessor register transfer</td>
<td>Cond</td>
<td>1 1 1 0</td>
<td>CP Opc L</td>
<td>CRn</td>
<td>Rd</td>
<td>CP#</td>
<td>CP 1</td>
</tr>
<tr>
<td>Software interrupt</td>
<td>Cond</td>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td>Ignored by processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


ADD R1, R2, R3 ; R1: = R2 + R3

Note:
1) Operand 1 is always a register
2) Operand 2 can be a register or an immediate value
Branch Instruction (B, BL)

- [31:28] Cond : Condition Field
- [27:25] 101 : B instruction
- [24] L bit
  - 0 – Branch (B)
  - 1 – Branch with Link (BL)
- [23:0] Offset
  - A signed 2’s complement 24 bit offset
  - Branches beyond +/- 32Mbytes (Because PC increases 2 words (8 bytes))

Examples:
- B label ; branch unconditionally to label
- BCC label ; branch to label if carry flag is clear
- BEQ label ; branch to label if zero flag is set
- MOV PC, #0 ; R15 = 0, branch to location zero
- BL func ; subroutine cal to function
- func MOV PC, LR ; R15 = R14, return to instruction after the BL
- MOV LR, PC ; store the address of the instruction after the next one into R14 ready to return
- LDR PC, =func ; load a 32-bit value into the program counter
Data Processing Instructions Format

- [31:28] Cond : Condition Field
- [25] I : Immediate Operand
  - 0 : Operand 2 is Register
  - 1 : Operand 2 is an immediate value
- [24:21] Opcode : Operation Codes (Data Processing Instruction)
- [20] S : Set condition codes (CPSR effect)
  - 0 : Do not after condition codes
  - 1 : Set condition codes
- [19:16] Rn : Source Register Operand 1 (always Register)
- [15:12] Rd : Destination Register
- [11:0] Operand 2 : Operand 2 type selection
[11:0] Operand 2 : Operand 2 type selection

0: Register

[3:0] Rm 2nd Operand register

Shift Operation

11  7  6  5  4  11  8  7  6  5  4
Count  Type  Rs  0  Type  1

[11:7] Shift amount
5 bit unsigned integer
[6:5] Shift type

Shift amount specified in
Bottom-byte of Rs

[6:5] Shift type

Shift Type Operation

<table>
<thead>
<tr>
<th>Value</th>
<th>Instruction</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Logical left</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>Logical right</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>Arithmetic right</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>Rotate right</td>
<td>ROR</td>
</tr>
</tbody>
</table>

1: immediate value

11  8  7  0
Rotate  Imm

[11:8] Rotate
Shift applied to Imm
[7:0]
Unsigned 8 bit immediate value

• 8 bit immediate value is zero extended
to 32 bits, And then subject to a rotate	right by twice the value in the rotate field

Op1( Rn )  Operand2

Barrel Shifter

ALU

Rd
Shift Operations

- **LSL #5**: Shifts the bits to the left by 5 positions.
- **LSR #5**: Shifts the bits to the right by 5 positions.
- **ASR #5, positive operand**: Arithmetic right shift by 5 positions, preserving the sign bit.
- **ASR #5, negative operand**: Arithmetic right shift by 5 positions, rotating the sign bit.
- **ROR #5**: Rotates the bits right by 5 positions.
## Opcode : Operation Codes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
<td>AND</td>
<td>Rd : = Op1 And Op2</td>
</tr>
<tr>
<td>0001</td>
<td>EOR</td>
<td>Exclusive</td>
<td>Rd: = Op1 XOR Op2</td>
</tr>
<tr>
<td>0010</td>
<td>SUB</td>
<td>Subtract</td>
<td>Rd: = Op1 – Op2</td>
</tr>
<tr>
<td>0011</td>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>Rd: = Op2 – Op1</td>
</tr>
<tr>
<td>0100</td>
<td>ADD</td>
<td>Add</td>
<td>Rd: = Op1 + Op2</td>
</tr>
<tr>
<td>0101</td>
<td>ADC</td>
<td>Add with carry</td>
<td>Rd: = Op1 + Op2 + C</td>
</tr>
<tr>
<td>0110</td>
<td>SBC</td>
<td>Subtract with carry</td>
<td>Rd: = Op1 – Op2 + C – 1</td>
</tr>
<tr>
<td>0111</td>
<td>RSC</td>
<td>Reverse Subtract with carry</td>
<td>Rd: = Op2 – Op1 + C – 1</td>
</tr>
<tr>
<td>1000</td>
<td>TST</td>
<td>Test bits</td>
<td>Op1 AND OP2 → CPSR</td>
</tr>
<tr>
<td>1001</td>
<td>TEQ</td>
<td>Test bitwise equality</td>
<td>Op1 XOR OP2 → CPSR</td>
</tr>
<tr>
<td>1010</td>
<td>CMP</td>
<td>Compare</td>
<td>Op1 – Op2 → CPSR</td>
</tr>
<tr>
<td>1011</td>
<td>CMN</td>
<td>Compare Negavive</td>
<td>Op1 + Op2 → CPSR</td>
</tr>
<tr>
<td>1100</td>
<td>ORR</td>
<td>Logical (inclusive) OR</td>
<td>Rd: = Op1 OR Op2</td>
</tr>
<tr>
<td>1101</td>
<td>MOV</td>
<td>Mov register or constant</td>
<td>Rd: = Op2</td>
</tr>
<tr>
<td>1110</td>
<td>BIC</td>
<td>Bit Clear</td>
<td>Rd: Op1 AND (NOT Op2)</td>
</tr>
<tr>
<td>1111</td>
<td>MVN</td>
<td>Move Not</td>
<td>Rd: = NOT Op2</td>
</tr>
</tbody>
</table>
## Condition Code Summary

<table>
<thead>
<tr>
<th>Code</th>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Z Set</td>
<td>equal</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Z Clear</td>
<td>not equal</td>
</tr>
<tr>
<td>0010</td>
<td>CS</td>
<td>C set</td>
<td>unsigned higher or same</td>
</tr>
<tr>
<td>0011</td>
<td>CC</td>
<td>C Clear</td>
<td>unsigned lower</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>N Set</td>
<td>negative</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>M Clear</td>
<td>positive or zero</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>V Set</td>
<td>overflow</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>V Clear</td>
<td>no overflow</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>C Set and Z Clear</td>
<td>unsigned higher</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>C Clear or Z Set</td>
<td>unsigned lower or same</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>N Set and V Set or N Clear and V Clear</td>
<td>greater or equal</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>N Set and V Clear or N Clear and V Set</td>
<td>less than</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Z Clear and ether N Set and V Set or N Clear and V Clear</td>
<td>greater than</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Z Set or N Set and V Clear or N Clear and V Set</td>
<td>less than or equal</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>( Ignored )</td>
<td>Always</td>
</tr>
</tbody>
</table>
Data Processing Instruction Summary (1)

- **Arithmetic operations**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R0, R1, R2</td>
<td>R0: = R1 + R2</td>
</tr>
<tr>
<td>ADC R0, R1, R2</td>
<td>R0: = R1 + R2 + C</td>
</tr>
<tr>
<td>SUB R0, R1, R2</td>
<td>R0: = R1 − R2</td>
</tr>
<tr>
<td>SUC R0, R1, R2</td>
<td>R0: = R1 − R2 + C −1</td>
</tr>
<tr>
<td>RSB R0, R1, R2</td>
<td>R0: = R2 − R1</td>
</tr>
<tr>
<td>RSC R0, R1, R2</td>
<td>R0: = R2 − R1 + C −1</td>
</tr>
</tbody>
</table>

- **Bit-wise logical Operations**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND R0, R1, R2</td>
<td>R0: = R1 and R2</td>
</tr>
<tr>
<td>ORR R0, R1, R2</td>
<td>R0: = R1 or R2</td>
</tr>
<tr>
<td>EOR R0, R1, R2</td>
<td>R0: = R1 xor R2</td>
</tr>
<tr>
<td>BIC R0, R1, R2</td>
<td>R0: = R1 and not R2</td>
</tr>
</tbody>
</table>

- **Register movement operations**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R2</td>
<td>R0: = R2</td>
</tr>
<tr>
<td>MVN R0, R2</td>
<td>R0: = not R2</td>
</tr>
</tbody>
</table>
Data Processing Instruction Summary (2)

- **Comparison operations**
  - CMP R1, R2 ; set cc on R1 – R2
  - CMN R1, R2 ; set cc on R1 + R2
  - TSTR1, R2 ; set cc on R1 and R2
  - TEQ R1, R2 ; set cc on R1 xor R2

- **Immediate operands**
  - ADD R3, R3, #1 ; R3: = R3 + 1
  - AND R8, R7, #&ff ; R8: = R7

- **Shifted register operands**
  - ADD R3, R2, R1, LSL #3 ; R3: = R2 + (8 x R1)
  - ADD R5, R5, R3, LSL R2 ; R5: = R5 + (R3 x 2^R2)

- **Setting the condition codes**
  - ADDS R2, R2, R10 ; R3: = R2 + (8 x R1)
  - ADC R3, R3, R1 ; R3: = R2 + (8 x R1)
PSR Transfer Instructions (1)

**MRS (Transfer PSR contents to a register)**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>0 0 0 0 1 0</td>
<td>PS</td>
<td>0 0 1 1 1 1</td>
<td>Rd</td>
<td>0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [31:28] Cond : Condition Field
- [22] PS : Source PSR
  - 0 = CPSR
  - 1 = SPSR_<current mode_>
- [15:12] Rd : Destination Register

**MSR (Transfer register contents to PSR)**

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>0 0 0 0 1 0</td>
<td>PS</td>
<td>0 0 1 1 1 1</td>
<td>Rd</td>
<td>0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [31:28] Cond : Condition Field
- [22] PS : Destination PSR
  - 0 = CPSR
  - 1 = SPSR_<current mode_>
- [3:0] Rd : Source Register
PSR Transfer Instructions (2)

- **Cond**: Condition Field
  - [31:28] Cond

- **I**: Immediate Operand
  - 0 = Source operand is a register
  - 1 = SPSR_<current mode>

- **Pd**: Destination PSR
  - 0 = CPSR
  - 1 = SPSR_<current mode>

- **Source operand**
  - [11:0]

[11:4] Source operand is an immediate value
[3:0] Source Register

[11:8] Shift applied to Imm
[7:0] Unsigned 8 bit immediate value
PSR Transfer Instructions (3)

**Example**

The flowing sequence performs a mode change:

```assembly
MSR  R0, CPSR ; Take a copy of the CPSR
BIC  R0, R0, #0x1F ; Clear the mode bits
ORR  R0, R0, #new_mode ; Select new mode
MSR  CPSR, R0 ; Write back the modified CPSR
```

The following instruction sets the N,Z,C and V flags:

```assembly
MSR  CPSR_flg, #0xF0000000 ; Set all the flags regardless of their
                           ; previous state (does not affect any
                           ; control bits)
```
Multiply and Multiply-Accumulate (MUL, MLA)

- [31:28] Cond : Condition Field
- [21] A : Accumulate
  - 0 : Multiply only
  - 1 : Multiply and Accumulate
- [20] S : Set condition codes (CPSR effect)
  - 0 : Do not after condition codes
  - 1 : Set condition codes
- [19:16] Rd : Destination Register (always Register)
- [15:12] Rn : Operand Register
- [11:8] Rs : Operand Register
- [3:0] Rm : Operand Register

- Multiply form
  - Rd : Rm x Rs
- Multiply and Accumulate form
  - Rd : Rm x Rs + Rn
Multiply Long and Multiply-Accumulate Long (MULL, MUAL)

- [31:28] Cond : Condition Field
- [21] U : Unsigned
  - 0 : Unsigned
  - 1 : Signed
- [21] A : Accumulate
  - 0 : Multiply only
  - 1 : Multiply and Accumulate
- [20] S : Set condition codes (CPSR effect)
  - 0 : Do not after condition codes
  - 1 : Set condition codes
- [19:16] RdHi : Source Destination Register (always Register)
- [15:12] RdLo : Source Destination Register (always Register)
- [11:8] Rs : Operand Register
- [3:0] Rm : Operand Register

- **Multiply Long Form**
  - RdHi, RdLo : Rm x Rs
- **Multiply and Accumulate Long Form**
  - RdHi, RdLo : Rm x Rs + RdHi,RdLo
## MUL and MLA Instructions Example

- **Example**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>R4, R2, R1</td>
<td>Set R4 to value of R2 multiplied by R1</td>
</tr>
<tr>
<td>MULS</td>
<td>R4, R2, R1</td>
<td>R4 = R2 x R1, set N and Z flags</td>
</tr>
<tr>
<td>MLA</td>
<td>R7, R8, R9, R3</td>
<td>R7 = R8 x R9 + R3</td>
</tr>
<tr>
<td>SMULL</td>
<td>R4, R8, R2, R3</td>
<td>R4 = bits 0 to 31 of R2 x R3, R8 = bits 32 to 63 of R2 x R3</td>
</tr>
<tr>
<td>UMULL</td>
<td>R6, R8, R0, R1</td>
<td>R8, R6 = R0 x R1</td>
</tr>
<tr>
<td>UMLAL</td>
<td>R5, R8, R0, R1</td>
<td>R8, R5 = R0 x R1 + R8, R5</td>
</tr>
</tbody>
</table>
Data transfer instructions – Single Data Transfer (LDR, STR)

- **[31:28] Cond**: Condition Field
- **[25] I**: Immediate
- **[24] P**: Pre/Post Indexing Bit
  - 0 = Post: add offset after transfer
  - 1 = Pre: add offset before transfer
- **[23] U**: Up/Down Bit
  - 0 = Down: Subtract offset from base
  - 1 = Up: add offset to base
- **[22] B**: Byte/Word Bit
  - 0 = Transfer word quantity
  - 1 = Transfer byte quantity
- **[21] Write-back Bit**
  - 0 = No write-back
  - 1 = Write address into base
- **[20] Load/Store Bit**
  - 0 = Store to memory
  - 1 = Load from memory
- **[19:16] Base Register**
- **[15:12] Source/Destination Registers**

**Offset Field**

- **[11:0] Offset**

**Immediate Field**

- **[11:0] Immediate offset**

**Shift Field**

- **[11:4] Shift applied to Rm**
- **[3:0] Offset register**
### LDR and STR Transfer Mode

<table>
<thead>
<tr>
<th>MODE</th>
<th>Effective Address</th>
<th>Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Rn]</td>
<td>Rn</td>
<td>None</td>
</tr>
<tr>
<td>[Rn, ± expression]</td>
<td>Rn, ± expression</td>
<td>Pre-indexed</td>
</tr>
<tr>
<td>[Rn, ± Rm]</td>
<td>Rn, ± Rm</td>
<td>Pre-indexed</td>
</tr>
<tr>
<td>[Rn, ± Rm, shift cnt]</td>
<td>Rn, ± (Rm shift by cnt)</td>
<td>Pre-indexed</td>
</tr>
<tr>
<td>[Rn], ± expression</td>
<td>Rn</td>
<td>Post-indexed</td>
</tr>
<tr>
<td>[Rn], ± Rm</td>
<td>Rn</td>
<td>Post-indexed</td>
</tr>
<tr>
<td>[Rn], ± Rm, shift cnt</td>
<td>Rn</td>
<td>Post-indexed</td>
</tr>
</tbody>
</table>

# Rn : base address register  
Rm : offset(register), signed value  
Expression : immediate value(12bit -4095 - +4096 )  
Shift : LSL, LSR, ASR, ROR  
cnt : 1 – 31 value
LDR and STR Address Mode

**Pre-indexed Addressing Mode**
- Rn : Base Address
- Rn ± Offset
- Offset : 12 bit immediate value or Register value
- Register Offset : Shift operation (optional)

Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R0, [R1]</td>
<td>R1:base, Offset : no</td>
</tr>
<tr>
<td>LDR R0, [R1, #132]</td>
<td>R1:base, Offset : #132</td>
</tr>
<tr>
<td>STR R0, [R1, R2]</td>
<td>R1:base, Offset : R2</td>
</tr>
<tr>
<td>LDR R0, [R1,R2, LSL #2]</td>
<td>R1:base, Offset : R2 &lt;&lt; 2</td>
</tr>
</tbody>
</table>

**Post-indexed Addressing Mode**
- Rn : Base Address Register
- Write-Back Function

Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R0, [R1], R2</td>
<td>1. R0 = [R1], 2. R1 = R1 + R2</td>
</tr>
<tr>
<td>STR R0, [R1], #20</td>
<td>1. [R1] = R0, 2. R1 = R1 + 20</td>
</tr>
</tbody>
</table>

**Relative Addressing Mode**
Write-Back

- Base address register = Base address register + Offset value
- Sign : !
- Example
  
  ```
  LDR R0, =table_end
  LDR R1, =table
  MOV R2, #0
  Loop STR R2, [R0, #-4]! ; Write-Back function
  ADD R2, R2, #1
  CMP R0, R1
  BNE loop...
  ...
  ALIGN
  table % table_length+4
  table_end
  ```

- 예를 들어 1000번지부터 2개의 entry를 가정한다면, table은 1000이고 table_end는 1008이 될 것이다. 이 때 STR 명령에서 1008을 베이스 번지로 하고, -4를 음식으로 설정 했으므로 실제로 R2값은 1004번지부터 4바이트를 값을 읽어 기록하게 되고, 이후에 Write-Back 음선이 적용되어 해당 명령이 끝나면 R0에 1004(1008-40)로 값이 변경 됩니다.
### Block data transfer (LDM, STM)

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:28</td>
<td>Condition Field</td>
</tr>
<tr>
<td>24</td>
<td>Pre/Post Indexing Bit</td>
</tr>
<tr>
<td>23</td>
<td>Up/Down Bit</td>
</tr>
<tr>
<td>22</td>
<td>PSR &amp; Force User Bit</td>
</tr>
<tr>
<td>21</td>
<td>Write-back Bit</td>
</tr>
<tr>
<td>20</td>
<td>Load/Store Bit</td>
</tr>
<tr>
<td>19:16</td>
<td>Base Register</td>
</tr>
</tbody>
</table>

- **[31:28] Cond**: Condition Field
- **[24] P**: Pre/Post Indexing Bit
  - 0 = Post: add offset after transfer
  - 1 = Pre: add offset before transfer
- **[23] U**: Up/Down Bit
  - 0 = Down: Subtract offset from base
  - 1 = Up: add offset to base
- **[22] P**: PSR & Force User Bit
  - 0 = Do not load PSR or user mode
  - 1 = Load PSR or force user mode
- **[21] Write-back Bit**
  - 0 = No write-back
  - 1 = Write address into base
- **[20] Load/Store Bit**
  - 0 = Store to memory
  - 1 = Load from memory
- **[19:16] Base Register**
Multiple register transfer addressing modes

STMIA R9!,{R0, R1, R5}

STMIB R9!,{R0, R1, R5}

STMDA R9!,{R0, R1, R5}

STMDB R9!,{R0, R1, R5}
The mapping between the stack and block copy views of the load and store multiple instructions

<table>
<thead>
<tr>
<th></th>
<th>Ascending</th>
<th>Descending</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full</td>
<td>Empty</td>
</tr>
<tr>
<td>Increment</td>
<td>Before</td>
<td>STMIB</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>STMIA</td>
</tr>
<tr>
<td>Decrement</td>
<td>Before</td>
<td>LDMDB</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>LDMDA</td>
</tr>
</tbody>
</table>
Directives

- Data structure definitions and allocation of space for data
- Partitioning of files into logical subdivisions
- Error reporting and control of assembly listing
- Definition of symbols
- Conditional and repetitive assembly, and inclusion of subsidiary files.
# directive

The # directive describes space within a storage map that has been defined using the ^ directive

- **Usage**
  - If a storage map is set by a ^directive that specifies a base-register, the base register is implicit in all labels defined by following 3 directives, until the next ^ directive.

- **Example**
  ```
  ^ 0, R9 ; set @ to the address stored in R9
  # 4 ; increment @ by 4 bytes
  Lab# 4 ; set Lab to the address [R9 + 4]
  ; and then increment @ by 4 bytes
  LDR R0, Lab ; equivalent to LDR R0, [R9, #4]
  
  ; @ - the value of the storage location counter
  ```
% directive

The % directive reserves a zeroed block of memory.

Example

```
AREA MyData, DATA, READWRITE
Data1 % 255 ; defines 255 bytes of zeroed store
```
The IF directive introduces a condition that is used to decide whether to assemble a sequence of instructions and/or directives.

**Usage**
- Use IF with ENDIF, and optionally with ELSE, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition.

**Example**

```
[ Version = "1.0"         ; IF ...
  ; code and / or
  ; directives
|                         ; ELSE ...
  ; code and / or
  ; directives
]                         ; ENDIF
```
ALIGN directive

By default, the ALIGN directive aligns the current location within the code to a word (4-byte) boundary.

Usage

- Use ALIGN to ensure that your code is correctly aligned.

Example

```assembly
AREA Example, CODE, READONLY

start LDR R6, =label1
DCB 1 ; pc misaligned
ALIGN ; ensures that label1 addresses the following instruction

label1 MOV R5, #0x5

AREA cacheable, CODE, ALIGN=4

rout1 ; code ; aligned on 16-byte boundary
; code
MOV pc, lr ; aligned only on 4-byte boundary
ALIGN 16 ; now aligned on 16-byte boundary

rout2 ; code
```
The DATA directive informs the assembler that a label is a data-in-code label. This means that the label is the address of data within a code segment.

**Usage**
- You must use the DATA directive when you define data in a Thumb code area with any of the data-defining directives such as DCD, DCB, and DCW

**Example**

```
AREA example, CODE

Thumb_fn ; code

; code
MOV pc, lr

Thumb_Data DATA
DCB 1, 3, 4
```
DCB(=) DCW DCD(&) directives

- Directives allocates one or more bytes (halfword, word) of memory

Usage
- You must use the DATA directive if you use DCB(DCW, DCW) to define labeled data within Thumb code.

Example

C_string DCB "C_string", 0
 AREA MiscData, DATA, READWRITE
Data DCW -225, 2*number
 DCW number + 4

data1 DCD 1, 5, 20 ; Defines 3 words containing decimal values 1, 5, and 20
data2 DCD mem06 ; Defines 1 word containing the address of the label ; mem06
data3 DCD glb + 4 ; Defines 1 word containing 4 + the value of glb
GBLA directive

The GBLA directive declares and initializes a global arithmetic variable. The range of values that arithmetic variables may take is the same as that of numeric expressions.

Usage
- Using GBLA for a variable that is already defined re-initializes the variable to 0.

Example

```
GBLA objectsize ; declare the variable name
Objectsize SETA 0xff ; set its value
% objectsize ; quote the variable
```
The GET directive includes a file within the file being assembled. The included file is assembled. INCLUDE is a synonym for GET.

Usage

- GET is useful for including macro definitions, EQUs, and storage maps in an assembly.

Example

```
AREA Example, CODE, READONLY
GET file1.s ; include file1 if it exists
; in the current place
GET c:\project\file2.s ; includes files
```
Instruction Example(1)

AREA HelloW, Code, READONLY

SWI_WriteC EQU &0
SWI(EXIT) EQU &11

ENTRY

START ADR R1, TEXT
LOOP LDRB R0, [R1], #1
CMP R0, #0
SWINE SWI_WriteC
BNE LOOP
SWI SWI_EXIT

TEXT = “Hello World”, &0a,
&0d, 0

END

; declare code area
; output character in R0
; finish program
; code entry point
; R1 -> “Hello World”
; get the next byte
; check for text end
; if not end print …
; … end loop back
; end of execution
; end of program source
Instruction Example(2)

AREA Blkcpy, Code, READONLY

SWI_WriteC EQU &0
SWI_Exit EQU &1

ENTRY
ADR R1, TABLE1
ADR R2, TABLE2
ADR R3, T1END

LOOP1
LDR R0, [R1], #4
STR R0, [R2], #4
CMP R1, R3
BLT R1, R3
ADR R1, TABLE2

LOOP2
LDRB R0, [R1], #1
CMP R0, #0
SWINE SWI_WriteC
BNE LOOP2
SWI SWI_Exit

TABLE1 = "This is the right string!", &0a, &0d, 0
T1END
ALIGN
TABLE2 = "This is the wrong string!", 0
END

; output character in R0
; finish program
; code entry point
; R0 -> TABLE1
; R1 -> TABLE2
; get TABLE1 1st word
; copy into TABLE2
; finished?
; if not, do more
; R1 -> TABLE2
; get next byte
; check for text end
; if not end, print …
; … and loop back
; finish
; ensure word alignment
**Instruction Example(3)**

<table>
<thead>
<tr>
<th>AREA</th>
<th>Hex_Out, CODE, READONLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI_WriteC</td>
<td>EQU &amp;0</td>
</tr>
<tr>
<td>SWI.Exit</td>
<td>EQU &amp;11</td>
</tr>
</tbody>
</table>

**ENTRY**

- LDR R1, VALUE
- BL HexOut
- SWI SWI.Exit

**VALUE**

- DCD &12345678

**HexOut**

- MOV R2, #8

**LOOP**

- MOV R0, R1, LSR #28
- CMP R0, #9
- ADDGT R0, R0, "A"-10
- ADDLE R0, R0, "0"
- SWI SWI_WriteC
- MOV R1, R1, LSL #4
- SUBS R2, R2, #1
- BNE LOOP
- MOV pc, R14

**END**

- output character in R0
- finish program
- code entry point
- get value to print
- call hexadecimal output
- finish
- test value
- nibble count = 8
- get top nibble
- 0-9 or A-F?
- ASCII alphabetic
- ASCII numeric
- print character
- shift left one nibble
- decrement nibble count
- if more do next nibble
- return
Instruction Example(4)

AREA        Text_Out, CODE, READONLY
SWI_WriteC  EQU &0
SWI.Exit     EQU &11

ENTRY
BL           TextOut
            =
            "Test string", &0a, &0d, 0
ENTRY
BL           TextOut
            =
            "Test string", &0a, &0d, 0
ALIGN
SWI          SWI.Exit
TextOut      LDRB  R0, [R14], #1
            CMP   R0, #0
            SWINE SWI_WriteC
            BNE   TextOut
            ADD   R14, R14, #3
            BIC   R14, R14, #3
            MOV   pc, r14
END

; output character in R0
; finish program
; code entry point
; print following string
; finish
; get next character
; test for end mark
; if not end, print …
; … and loop
; pass next word boundary
; round back to boundary
; return
```c
#include <stdio.h>

void my_strcpy(char *src, char *dst)
{
    int ch;
    __asm
    {
        loop:
        LDRB    ch, [src], #1
        STRB    ch, [dst], #1
        CMP     ch, #0
        BNE     loop
    }
}

int main(void)
{
    char a[] = "hello world!";
    char b[20];
    __asm
    {
        MOV R0, a
        MOV R1, b
        BL     my_strcpy, {R0, R1}, {}, {}
    }
    printf("Original string = %s\n", a);
    printf("Copied string = s\n", b);
    return 0;
}
```