ARM Processors and Architectures

Tratto in parte da
ARM University Program

ARM

- ARM was developed at Acorn Computer Limited of Cambridge, UK (between 1983 & 1985)
  - RISC concept introduced in 1980 at Stanford and Berkeley
- ARM founded in November 1990
  - **A**dvanced **R**ISC **M**achines

- Best known for its range of RISC processor cores designs
  - Other products – fabric IP, software tools, models, cell libraries - to help partners develop and ship ARM-based SoCs
- ARM does *not* manufacture silicon
  - Licensed to partners to develop and fabricate new micro-controllers
    - Soft-core

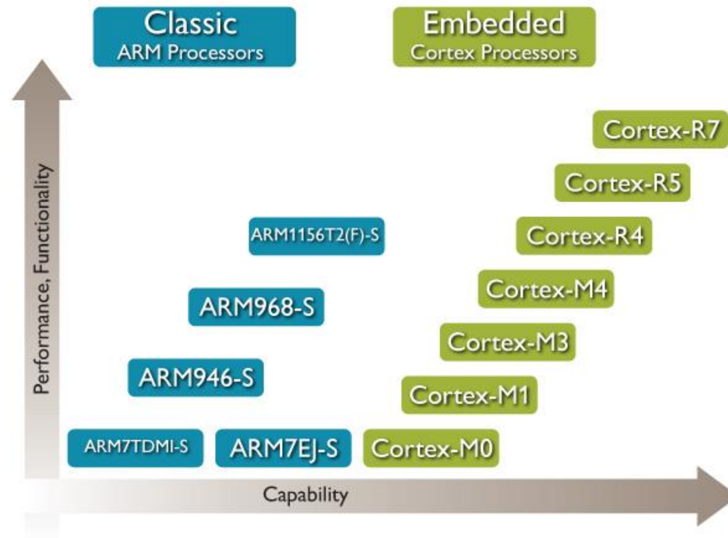
ARM architecture

- Based upon RISC Architecture with enhancements to meet requirements of embedded applications
  - A large uniform register file
  - Load-store architecture
  - Fixed length instructions
  - 32-bit processor
  - Good speed/power
  - High code density

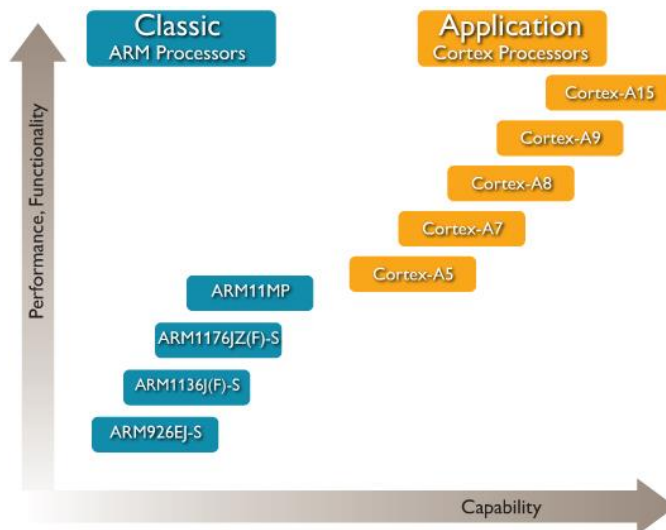
Enhancement to Basic RISC

- Control over ALU and shifter for every data processing operations
- Auto-increment and auto-decrement addressing modes
  - To optimize program loops
- Load/Store multiple data instructions
  - To maximize data throughput
- Conditional execution of instructions
  - To maximize execution throughput

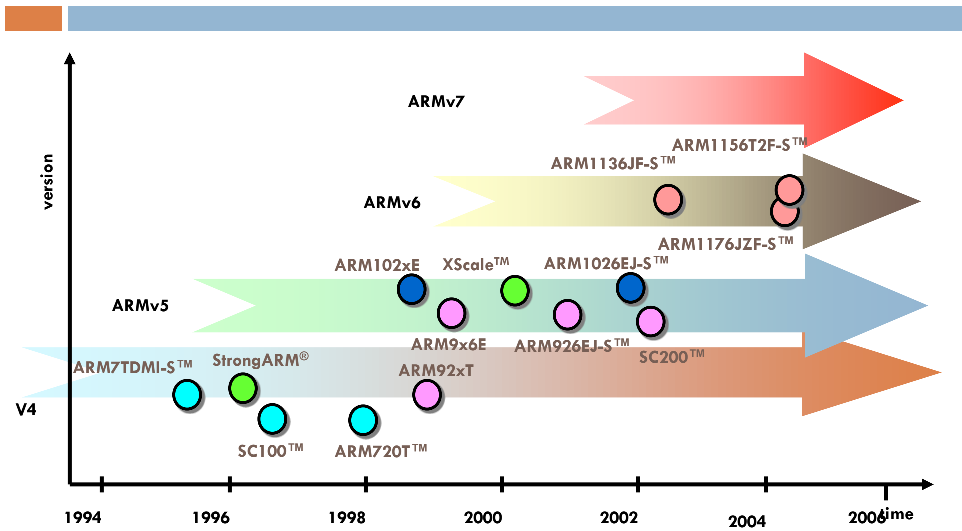
Embedded Processors



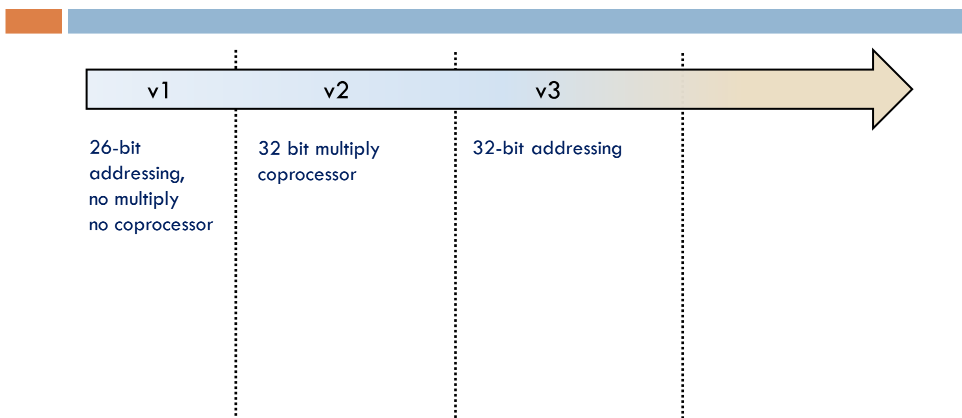
Application Processors



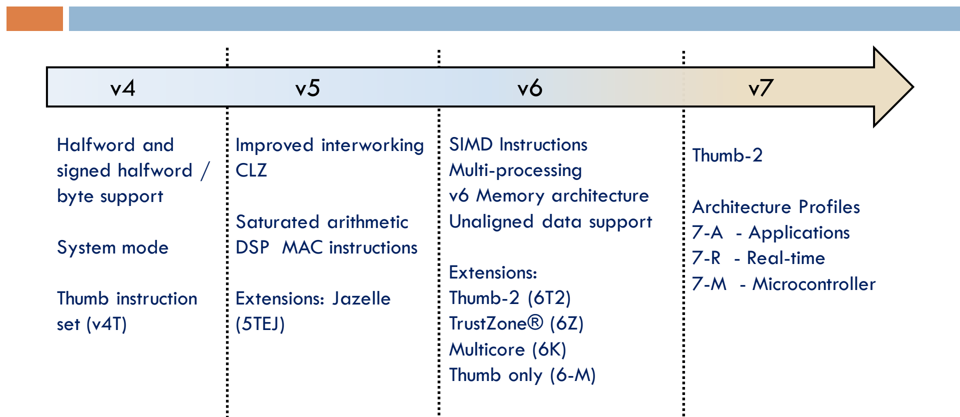
Architecture Revisions



Development of the ARM Architecture



Development of the ARM Architecture

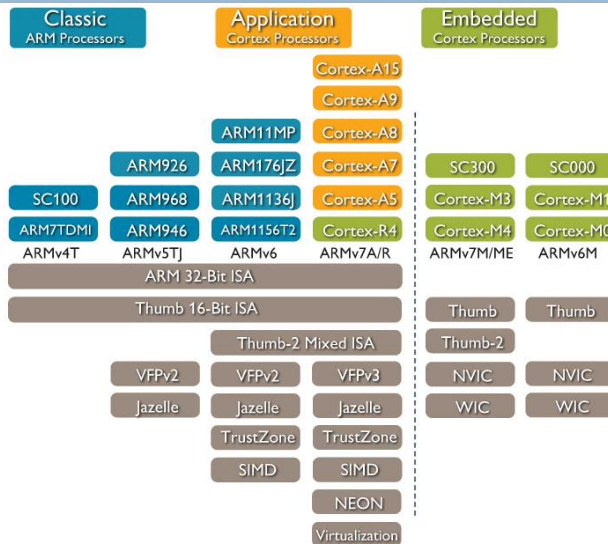


- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

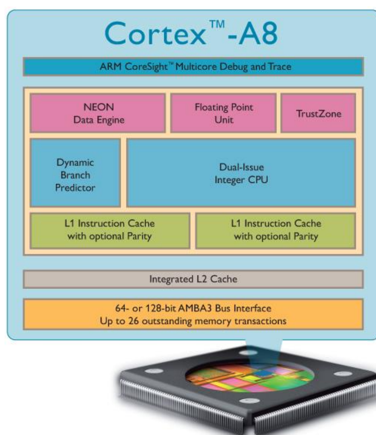
Architecture ARMv7 profiles

- Application profile (ARMv7-A)
  - Memory management support (MMU)
  - Highest performance at low power
    - Influenced by multi-tasking OS system requirements
  - TrustZone and Jazelle-RCT for a safe, extensible system
  - e.g. Cortex-A5, Cortex-A9
- Real-time profile (ARMv7-R)
  - Protected memory (MPU)
  - Low latency and predictability 'real-time' needs
  - Evolutionary path for traditional embedded business
  - e.g. Cortex-R4
- Microcontroller profile (ARMv7-M, ARMv7E-M, ARMv6-M)
  - Lowest gate count entry point
  - Deterministic and predictable behavior a key priority
  - Deeply embedded use
  - e.g. Cortex-M3

Which architecture is my processor?



Cortex-A8



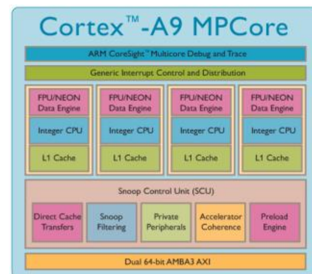
- **Dual-issue, super-scalar 13-stage pipeline**
  - Branch Prediction & Return Stack
  - NEON and VFP implemented at end of pipeline

- ARMv7-A Architecture
  - Thumb-2
  - Thumb-2EE (Jazelle-RCT)
  - TrustZone extensions
- Custom or synthesized design
- MMU
- 64-bit or 128-bit AXI Interface
- L1 caches
  - 16 or 32KB each
- Unified L2 cache
  - 0-2MB in size
  - 8-way set-associative

- **Optional features**
  - VFPv3 Vector Floating-Point
  - NEON media processing engine

Cortex-A9

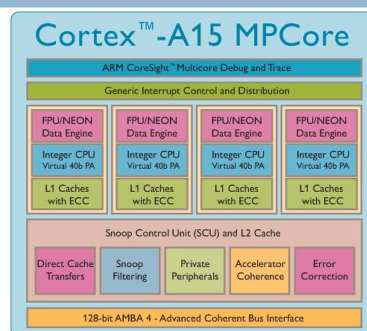
- ARMv7-A Architecture
  - Thumb-2, Thumb-2EE
  - TrustZone support
- Variable-length Multi-issue pipeline
  - Register renaming
  - Speculative data prefetching
  - Branch Prediction & Return Stack
- 64-bit AXI instruction and data interfaces
- TrustZone extensions
- L1 Data and Instruction caches
  - 16-64KB each
  - 4-way set-associative



- **Optional features:**
  - PTM instruction trace interface
  - IEM power saving support
  - Full Jazelle DBX support
  - VFPv3-D16 Floating-Point Unit (FPU) or NEON™ media processing engine

Cortex-A15 MPCore

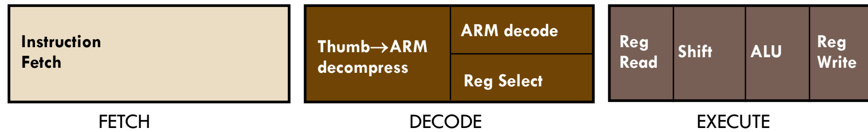
- 1-4 processors per cluster
- Fixed size L1 caches (32KB)
- Integrated L2 Cache
  - 512KB – 4MB
- System-wide coherency support with AMBA 4 ACE
- Backward-compatible with AXI3 interconnect
- Integrated Interrupt Controller
  - 0-224 external interrupts for entire cluster
- CoreSight debug
- Advanced Power Management



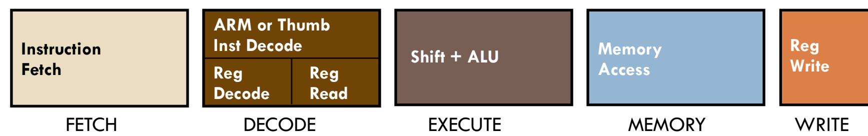
- **Large Physical Address Extensions (LPAE) to ARMv7-A Architecture**
- **Virtualization Extensions to ARMv7-A Architecture**

Pipeline changes for ARM9TDMI

ARM7TDMI

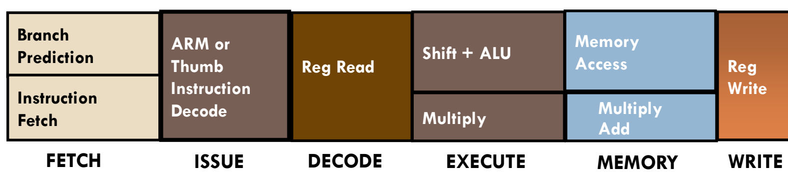


ARM9TDMI

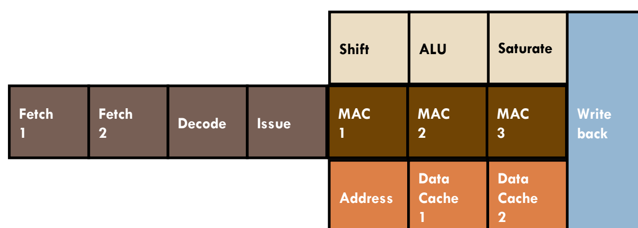


ARM10 vs. ARM11 Pipelines

ARM10



ARM11



Data Sizes and Instruction Sets

- ARM is a 32-bit load / store RISC architecture
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide
  - Most instructions execute in a single cycle
- When used in relation to ARM cores
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
  - **Doubleword** means 64 bits (eight bytes)

Data Sizes and Instruction Sets

- ARM cores implement two basic instruction sets
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set
- Depending on the core, may also implement other instruction sets
  - **VFP** instruction set – 32 bit (vector) floating point instructions
  - **NEON** instruction set – 32 bit SIMD instructions
  - **Jazelle-DBX** - provides acceleration for Java VMs (with additional software support)
  - **Jazelle-RCT** - provides support for interpreted languages

Core Data Path

- Data items are placed in register file
  - No data processing instructions directly manipulate data in memory
- Instructions typically use two source registers and a single destination register
- A barrel shifter on the data path can preprocess data before it enters ALU
- Increment/decrement logic can update register content for sequential access independent of ALU

Basic ARM Organization

Registers

- General Purpose registers hold either data or address
- All registers are of 32 bits
- In user mode 16 data registers and 2 status registers are visible
- Data registers: r0 to r15
  - r13, r14, and r15 perform special functions
    - r13: stack pointer
    - r14: link register
    - r15: program counter

Registers

- Depending upon context, registers r13 and r14 can also be used as GPR
- Any instruction which use r0 can as well be used with any other GPR (r1-r13)
- Two status registers
  - CPSR: Current Program Status Register
  - SPSR: Saved Program Status Register

Processor Modes

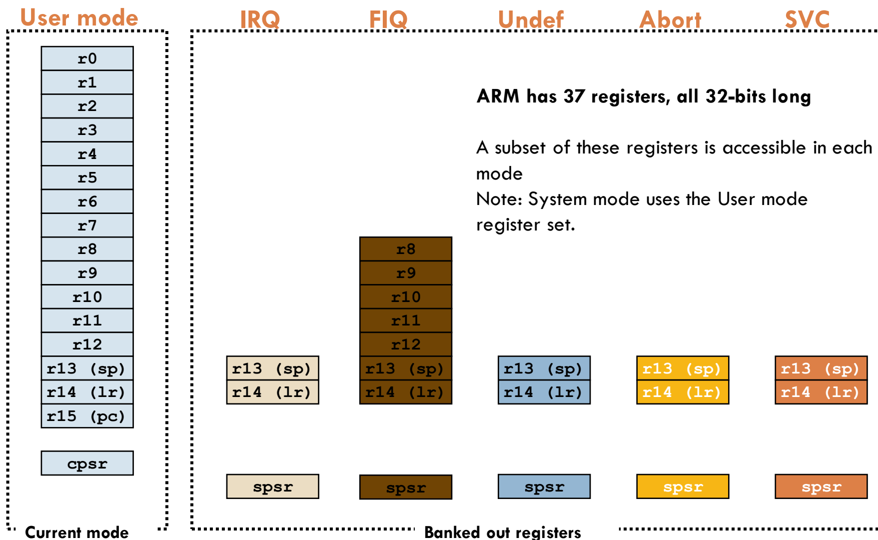
- Processor modes determine
  - Which registers are active
  - Access right to CPSR registers itself
- Each processor mode is either
  - Privileged: full read-write access to the CPSR
  - Non-privileged: only read access to the control field of CPSR but read-write access to the condition flags

Processor Modes

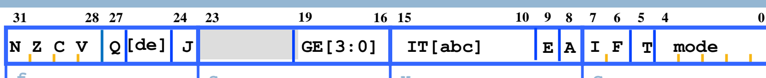
- ARM has seven basic operating modes
  - Each mode has access to its own space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

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

The ARM Register Set



Program Status Registers



- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- Sticky Overflow flag - Q flag
  - Indicates if saturation has occurred
- SIMD Condition code bits – GE[3:0]
  - Used by some SIMD instructions
- IF THEN status bits – IT[abcde]
  - Controls conditional execution of Thumb instructions

- T bit
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- J bit
  - J = 1: Processor in Jazelle state
- Mode bits
  - Specify the processor mode
- Interrupt Disable bits
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- E bit
  - E = 0: Data load/store is little endian
  - E = 1: Data load/store is bigendian
- A bit
  - A = 1: Disable imprecise data aborts

Program Counter (r15)

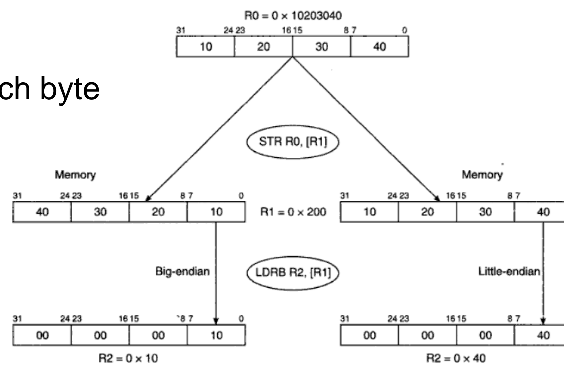
- When the processor is executing in ARM state:
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)
- When the processor is executing in Thumb state:
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)
- When the processor is executing in Jazelle state:
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Mode Changing

- Mode changes by writing directly to CPSR or by hardware when the processor responds to exception or interrupt
- To return to user mode a special return instruction is used that instructs the core to restore the original CPSR and banked registers

ARM Memory Organization

- Can be configured as
  - ➔Little Endian
  - ➔Big Endian
- Addresses are for each byte



ARM Instruction Set

Instructions

- Instruction process data held in registers and access memory with load and store instructions
- Classes of instructions
  - Data processing
  - Branch instructions
  - Load-store instructions
  - Software interrupt instruction
  - Program status register instructions

Features of ARM Instruction Set

- 3-address data processing instructions
- Conditional execution of every instruction
- Load and store multiple registers
- Shift, ALU operation in a single instruction
- Open instruction set extension through the co-processor instruction

ARM Data Types

- Word is 32 bits long
- Word can be divided into four 8-bit bytes
- ARM addresses can be 32 bit long
- Address refers to byte
- Can be configured at power-up as either little- or big-endian mode

Data Processing Instructions

- Consist of :
  - Arithmetic: **ADD ADC SUB SBC RSB RSC**
  - Logical: **AND ORR EOR BIC**
  - Comparisons: **CMP CMN TST TEQ**
  - Data movement: **MOV MVN**
- These instructions only work on registers, NOT memory.
- Syntax:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter.
- Suffix *S* on data processing instructions updates flags in CPSR

Data Processing Instructions

- Operands are 32-bit wide
  - ➔Come from registers of specified as literal in the instruction itself
- Second operand sent to ALU via barrel shifter
- 32-bit result placed in register
  - ➔Long multiply instruction produces 64-bit result

Move instruction

- MOV Rd, N
  - ➔Rd: destination register
  - ➔N: can be an immediate value or source register
  - ➔Example: MOV r7, r5
- MVN Rd, N
  - ➔Move into Rd not of the 32-bit value from source

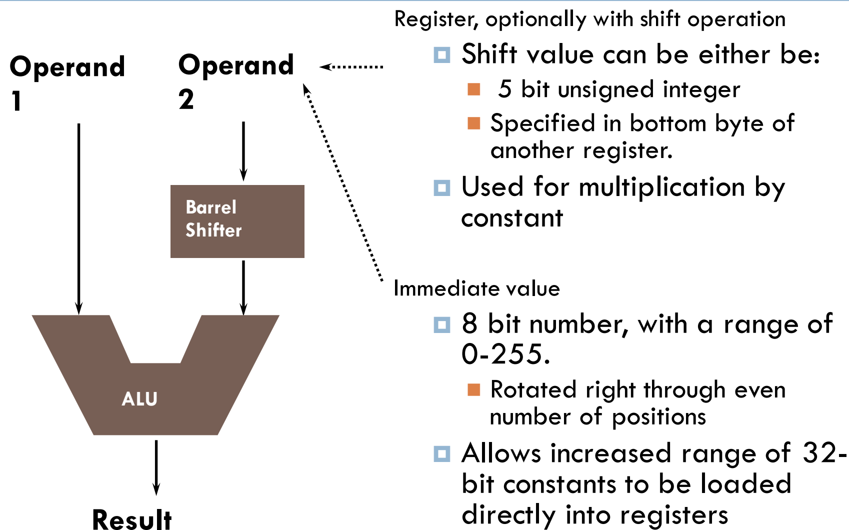
Using Barrel Shifter

- Enables shifting 32-bit operand in one of the source registers left or right by a specific number of positions within the cycle time of instruction
- Basic Barrel shifter operations
  - ➔Shift left, shift right, rotate right
- Facilitate fast multiply, division and increases code density
- Example: MOV r7, r5, LSL #2
  - ➔Multiplies content of r5 by 4 and puts result in r7

Arithmetic Instructions

- Implements 32-bit addition and subtraction
  - ➔3-operand form
  - ➔Examples
    - ✓SUB r0, r1, r2
      - – Subtract value stored in r2 from that of r1 and store in r0
    - ✓SUBS r1, r1, #1
      - – Subtract 1 from r1 and store result in r1 and update Z and C flags

Using a Barrel Shifter:The 2nd Operand



Data Processing Exercise

1. How would you load the two's complement representation of -1 into Register 3 using one instruction?

2. Implement an ABS (absolute value) function for a registered value using only two instructions.

3. Multiply a number by 35, guaranteeing that it executes in 2 core clock cycles.

Data Processing Solutions

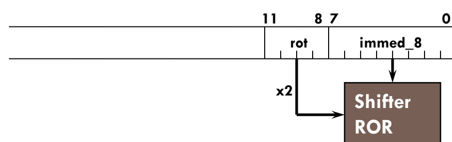
```
1. MOVN  r6, #0

2. MOVS  r7,r7          ; set the flags
   RSBMI r7,r7,#0       ; if neg, r7=0-r7

3. ADD   r9,r8,r8,LSL #2     ; r9=r8*5
   RSB   r10,r9,r9,LSL #3    ; r10=r9*7
```

Immediate constants

- No ARM instruction can contain a 32 bit immediate constant
  - All ARM instructions are fixed as 32 bits long
- The data processing instruction format has 12 bits available for operand2



Quick Quiz:

```
0xe3a004ff
MOV r0, #???
```

- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2
- Rule to remember is

  "8-bits rotated right by an even number of bit positions"

Multiply and Divide

- There are 2 classes of multiply - producing 32-bit and 64-bit results
- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles
  - `MUL r0, r1, r2           ; r0 = r1 * r2`
  - `MLA r0, r1, r2, r3       ; r0 = (r1 * r2) + r3`
- 64-bit multiply instructions offer both signed and unsigned versions
  - For these instruction there are 2 destination registers
  - `[U|S]MULL r4, r5, r2, r3     ; r5:r4 = r2 * r3`
  - `[U|S]MLAL r4, r5, r2, r3     ; r5:r4 = (r2 * r3) + r5:r4`
- Most ARM cores do not offer integer divide instructions
  - Division operations will be performed by C library routines or inline shifts

Logical Instructions

- Bit wise logical operations on the two source registers
  - ➔AND, OR, Ex-OR, bit clear
  - ➔Example: BIC r0, r1, r2
    - ✓R2 contains a binary pattern where every binary 1 in r2 clears a corresponding bit location in register r1
    - ✓Useful in manipulating status flags and interrupt masks

Compare Instructions

- Enables comparison of 32 bit values
  - Updates CPSR flags but do not affect other registers
  - Examples
    - ✓CMP r0, r9
      - Flags set as a result of r0 - r9
    - ✓TEQ r0, r9
      - Flags set as a result r0 ex-or r9
    - ✓TST r0, r9
      - Flags set as a result of r0 & r9

Instruction Set basics

- The ARM Architecture is a **Load/Store** architecture
  - No direct manipulation of memory contents
  - Memory must be loaded into the CPU to be modified, then written back out
- Cores are either in ARM *state* or Thumb *state*
  - This determines which instruction set is being executed
  - An instruction must be executed to switch between states
- The architecture allows programmers and compilation tools to reduce branching through the use of conditional execution
  - Method differs between ARM and Thumb, but the principle is that most (ARM) or all (Thumb) instructions can be executed conditionally.

Load-Store Instructions

- Transfer data between memory and processor registers
  - Single register transfer
    - Data types supported are signed and unsigned words (32 bits), half-word, bytes
  - Multiple-register transfer
    - Transfer multiple registers between memory and the processor in a single instruction
  - Swap
    - Swaps content of a memory location with the contents of a register

Single Transfer Instructions

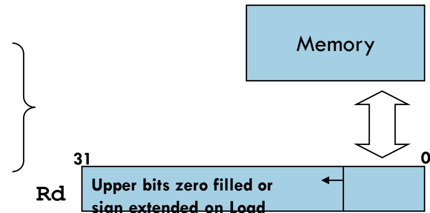
LDR R0, [R8]	load content of memory location pointed to by $R8$ into $R0$
LDR R0, [R1, -R2]	load content of memory location pointed to by $R1 - R2$ into $R0$
LDR R0, [R1, +4]	load content of memory location pointed to by $R1 + 4$ into $R0$
LDR R0, [R1, +4]!	load content of memory location pointed to by $R1 + 4$ into $R0$, $R1$ is also incremented by 4
LDR R0, [R1], +16	Loads $R0$ from memory location pointed to by $R1$, then adds 16 to $R1$

LDR	Load word	STR	Store word
LDRH	Load half word	STRH	Store half word
LDRSH	Load signed half word	STRSH	Store signed half word
LDRB	Load byte	STRB	Store byte
LDRSB	Load signed byte	STRSB	Store signed byte

Single Access Data Transfer

- Use to move data between one or two registers and memory

LDRD	STRD	Doubleword
LDR STR		Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load



- Syntax:
  - `LDR{<size>}{<cond>} Rd, <address>`
  - `STR{<size>}{<cond>} Rd, <address>`

- Example:

```
LDRB r0, [r1] ; load bottom byte of r0 from the
              ; byte of memory at address in r1
```

Single Transfer Instructions

- Load & Store data on a boundary alignment
  - ➔LDR, LDRH, LDRB
    - ✓Load (word, half-word, byte)
  - ➔STR, STRH, STRB
    - ✓Store (word, half-word, byte)
- Supports different addressing modes
  - ➔Register indirect: LDR r0, [r1]
  - ➔Immediate: LDR r0, [r1,#4]
    - ✓12-bit offset added to the base register
  - ➔Register operation: LDR r0, [r1,-r2]
    - ✓Address calculated using base register and another register

More Addressing Modes

- Scaled
  - Address is calculated using the base address register and a barrel shift operation
- Pre & Post Indexing
  - Pre-index with write back: LDR r0, [r1,#4]!
    - Updates the address base register with new address
  - Post index: LDR r0, [r1], #4
    - Updates the address register after address is used

Example

- Pre-indexing with write back
- LDR r0, [r1,#4]!
  - Before instruction execution
    - r0 = 0x00000000 r1 = 0x00009000
      Mem32[0x00009000] = 0x01010101
      Mem32[0x00009004] = 0x02020202
  - After instruction execution
    - r0 = 0x02020202
      r1 = 0x00009004

Multiple Register Transfer

- Load-store multiple instructions transfer multiple register contents between memory and the processor in a single instruction
- More efficient for moving blocks of memory and saving and restoring context and stack
- These instructions can increase *interrupt latency*
  - ➔Instruction executions are not interrupted by ARM

Multiple Byte Load-Store

- Any subset of current bank of registers can be transferred to memory or fetched from memory
  - ➔LDM
  - ➔SDM
- The base register Rn determines source or destination address

Multiple Register Data Transfer

- **These instructions move data between multiple registers and memory**
- **Syntax**
  - `<LDM|STM>{<addressing_mode>}{<cond>} Rb{!}, <register list>`
- **4 addressing modes**
  - Increment after/before
  - Decrement after/before

(IA) IB DA DB

Base Register (Rb) r10

r4 r1 r0

Increasing Address

- **Also**
  - **PUSH/POP**, equivalent to **STMDB/LDMIA** with **SP!** as base register
- **Example**
  - `LDM    r10, {r0,r1,r4}      ; load registers, using r10 base`
  - `PUSH   {r4-r6,pc}           ; store registers, using SP base`

Example

- Moving a large data block

```
; R12 points to the start if the source data
; R14 points to the end of the source data
; R13 points to the start of the destination data

Loop    LDMIA  R12!, {R0-R11}     ; load 48 bytes...
        STMIA  R13!, {R0-R11}     ; ...and store them
        CMP    R12, R14           ; check for the end
        BNE    Loop               ; and loop until done
```

Addressing Modes

- LDMIA|IB|DA|DB   ex: LDMIA Rn!, {r1-r3}
- STMIA|IB|DA|DB

Stack Processing

- A stack is implemented as a linear data structure which grows up (ascending) or down (descending)
- Stack pointer hold the address of the current top of the stack

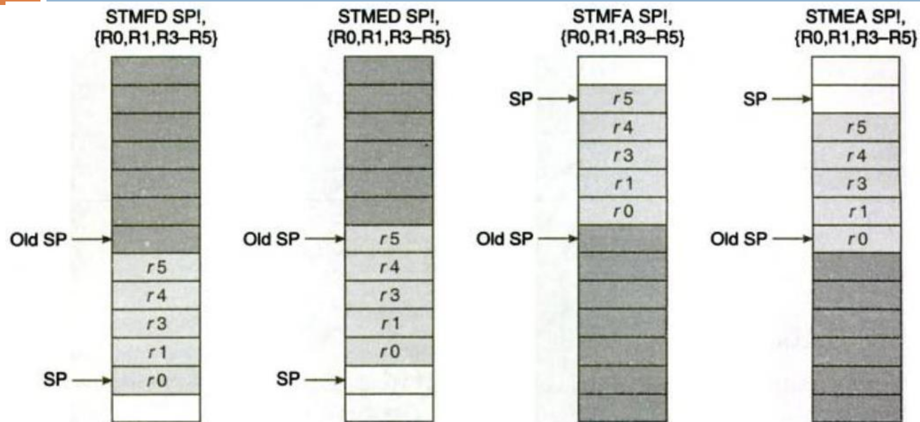
Modes of Stack Operation

- ARM multiple register transfer instructions support
  - Full ascending: grows up, SP points to the highest address containing a valid item
  - Empty ascending: grows up, SP points to the first empty location above stack
  - Full descending: grows down, SP points to the lowest address containing a valid data
  - Empty descending: grows down, SP points to the first location below the stack

Some Stack Instructions

- Full Ascending
  - LDMFA: translates to LDMIA (POP)
  - STMFA: translates to STMIB (PUSH)
  - SP points to last item in stack
- Empty Descending
  - LDMED: translates to LDMIB (POP)
  - STMED: translates to STMIA (PUSH)
  - SP points to first unused location

Stack Example



SWAP Instruction

- Special case of load store instruction
- Swap instructions
  - ➔SWP: swap a word between memory and register
  - ➔SWPB: swap a byte between memory and register
- Useful for implementing synchronization primitives like semaphore

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
  CMP    r3,#0
  BEQ    skip
  ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
ADDNE  r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
  …
  SUBS r1,r1,#1    ; decrement r1 and set flags
  BNE  loop        ; if Z flag clear then branch
```

Condition Codes

- The possible condition codes are listed below
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Conditional execution examples

C source code | **ARM instructions**

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

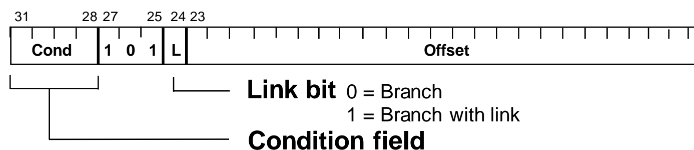
conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

Branch instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

Branch instruction

- Branch instruction: B label
  - Example: B forward
  - Address label is stored in the instruction as a signed pc-relative offset
- Conditional Branch: B<cond> label
  - Example: BNE loop
  - Branch has a condition associated with it and executed if condition codes have the correct value

Example: Block Memory Copy

```
Loop  LDMIA    r9!, {r0-r7}
      STMIA    r10!, {r0-r7}
      CMP      r9, r11
      BNE      Loop
```

- r9 points to source of data, r10 points to start of destination data, r11 points to end of the source

Branch & Link Instruction

- Perform a branch, save the address following the branch in the link register, r14
- Example: BL subroutine
- For nested subroutine, push r14 and some work registers required to be saved onto a stack in memory

```
        BL    Sub1
        …
Sub1    STMFD R13!,{R0-R2,R14}
        …
        BL    Sub2
        LDMFD R13!,{R0-R2,PC}
```

Subroutine Return Instructions

- No specific instructions
  - Example

```
sub:
    …
    MOV  PC, r14
```

- When return address has been pushed to stack

```
sub:
    …
    LDMFD  r13!, {r0-r12,PC}
```

Register Usage

	Register	

Arguments into function
Result(s) from function
otherwise corruptible
(Additional parameters passed on stack)
	`r0`
	`r1`
	`r2`
	`r3`	

Register variables
Must be preserved
	`r4`
	`r5`
	`r6`
	`r7`
	`r8`
	`r9/sb`
	`r10/sl`
	`r11`
		
		- Stack base
		- Stack limit if software stack checking selected

Scratch register (corruptible)	`r12`	

Stack Pointer
Link Register
Program Counter
	`r13/sp`
	`r14/lr`
	`r15/pc`
		- SP should always be 8-byte (2 word) aligned
		- R14 can be used as a temporary once value stacked

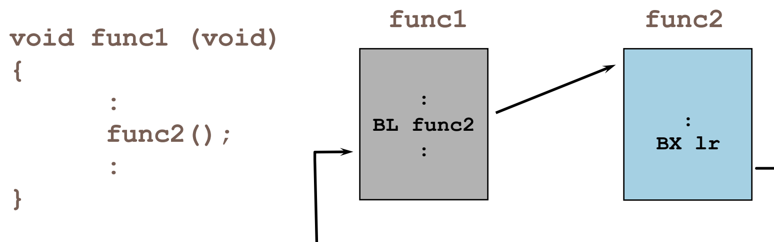
The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see **AAPCS**)

CPSR flags may be corrupted by function call. Assembler code which links with compiled code must follow the AAPCS at external interfaces

The AAPCS is part of the new ABI for the ARM Architecture

Subroutines

- Implementing a conventional subroutine call requires two steps
  - Store the return address
  - Branch to the address of the required subroutine
- These steps are carried out in one instruction, `BL`
  - The return address is stored in the link register (`lr/r14`)
  - Branch to an address (range dependent on instruction set and width)
- Return is by branching to the address in `lr`



Supervisor Call (SVC)

```
SVC{<cond>} <SVC number>
```

- **Causes an SVC exception**
- **The SVC handler can examine the SVC number to decide what operation has been requested**
  - But the core ignores the SVC number
- **By using the SVC mechanism, an operating system can implement a set of privileged operations (system calls) which applications running in user mode can request**
- **Thumb version is unconditional**

Exception Handling

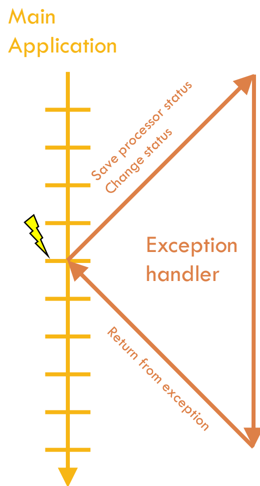
- When an exception occurs, the core…
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state (if appropriate)
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to…
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>
- Cores can enter ARM state or Thumb state when taking an exception
  - Controlled through settings in CP15
- Note that v7-M and v6-M exception model is different

Address	Vector
	⋮
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Supervisor Call
0x04	Undefined Instruction
0x00	Reset

Vector Table

Vector table can also be at `0xFFFF0000` on most cores

Exception handling process

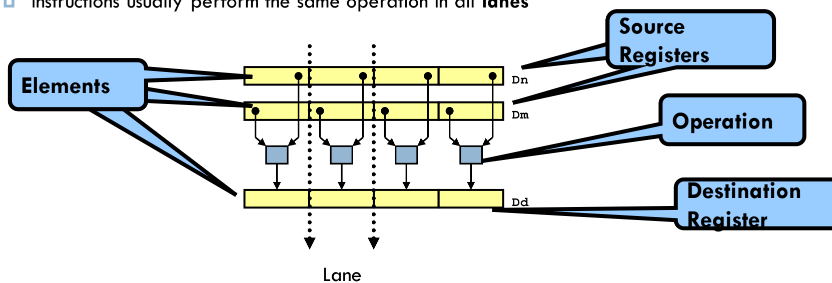


1. Save processor status
   - Copies `CPSR` into `SPSR_<mode>`
   - Stores the return address in `LR_<mode>`
   - Adjusts LR based on exception type
2. Change processor status for exception
   - Mode field bits
   - ARM or Thumb state
   - Interrupt disable bits (if appropriate)
   - Sets PC to vector address
3. Execute exception handler
   - <users code>
4. Return to main application
   - Restore `CPSR` from `SPSR_<mode>`
   - Restore PC from `LR_<mode>`

- 1 and 2 performed automatically by the core
- 3 and 4 responsibility of software

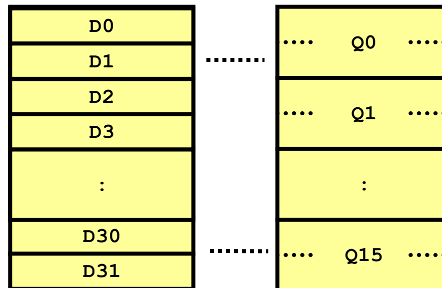
What is NEON?

- NEON is a wide SIMD data processing architecture
  - Extension of the ARM instruction set (v7-A)
  - 32 x 64-bit wide registers (can also be used as 16 x 128-bit wide registers)
- NEON instructions perform "Packed SIMD" processing
  - Registers are considered as **vectors** of **elements** of the same data type
  - Data types available: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions usually perform the same operation in all **lanes**



NEON Coprocessor registers

- NEON has a 256-byte register file
  - Separate from the core registers (r0-r15)
  - Extension to the VFPv2 register file (VFPv3)
- Two different views of the NEON registers
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)
- Enables register trade-offs
  - Vector length can be variable
  - Different registers available



NEON vectorizing example

- How does the compiler perform vectorization?

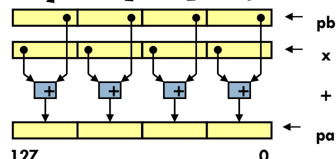
```
void add_int(int * __restrict pa,
             int * __restrict pb,
             unsigned int n, int x)
{
  unsigned int i;
  for(i = 0; i < (n & ~3); i++)
    pa[i] = pb[i] + x;
}
```

1. Analyze each loop:
   - Are pointer accesses safe for vectorization?
   - What data types are being used? How do they map onto NEON vector registers?
   - Number of loop iterations

2. Unroll the loop to the appropriate number of iterations, and perform other transformations like pointerization

```
void add_int(int *pa, int *pb,
             unsigned n, int x)
{
  unsigned int i;
  for (i = ((n & ~3) >> 2); i; i--)
  {
    *(pa + 0) = *(pb + 0) + x;
    *(pa + 1) = *(pb + 1) + x;
    *(pa + 2) = *(pb + 2) + x;
    *(pa + 3) = *(pb + 3) + x;
    pa += 4; pb += 4;
  }
}
```

3. Map each unrolled operation onto a NEON vector lane, and generate corresponding NEON instructions

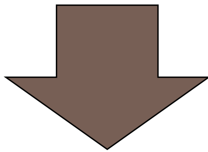


Thumb

- Thumb is a 16-bit instruction set
  - Optimised for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has additional execution state - Thumb
  - Switch between ARM and Thumb using **BX** instruction

ADDS r2,r2,#1

32-bit ARM Instruction



ADD r2,#1

16-bit Thumb Instruction

For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used