ARM®

The ARM Architecture

- **Introduction to ARM Ltd**

  Programmers Model

  Instruction Set

  System Design

  Development Tools

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

One of the most licensed and thus widespread processor cores in the world

Used in PDA, cell phones, multimedia players, handheld game console, digital TV and cameras

ARM7: GBA, iPod

ARM9: NDS, PSP, Sony Ericsson, BenQ

ARM11: Apple iPhone, Nokia N93, N800

75% of 32-bit embedded processors

Used especially in portable devices due to its low power consumption and reasonable performance

Intellectual Property

- **ARM provides hard and soft views to licencees**
  - RTL and synthesis flows
  - GDSII layout

- **Licencees have the right to use hard or soft views of the IP**
  - soft views include gate level netlists
  - hard views are DSMs

- **OEMs must use hard views**
  - to protect ARM IP

Introduction to ARM Ltd

- **Programmers Model**

Instruction Sets

System Design

Development Tools

Data Sizes and Instruction Sets

ARM

- **The ARM is a 32-bit architecture.**

- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- **Most ARM's implement two instruction sets**
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

- **Jazelle cores can also execute Java bytecode**

- **The ARM has seven basic operating modes:**

  - **User** : unprivileged mode under which most tasks run

  - **FIQ** : entered when a high priority (fast) interrupt is raised

  - **IRQ** : entered when a low priority (normal) interrupt is raised

  - **Supervisor** : entered on reset and when a Software Interrupt instruction is executed

  - **Abort** : used to handle memory access violations

  - **Undef** : used to handle undefined instructions

 - **System** : privileged mode using the same registers as user mode

Register Organization Summary

User	FIQ	IRQ	SVC	Undef	Abort
r0	User mode r0-r7, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8				
r9	r9				
r10	r10				
r11	r11				
r12	r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
r15 (pc)					
cpsr					
	spsr	spsr	spsr	spsr	spsr

Thumb state Low registers

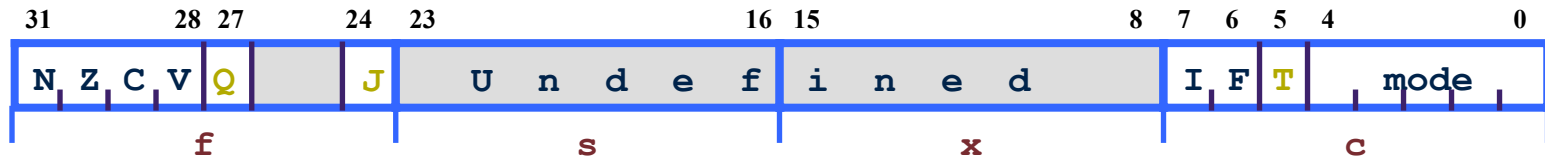**Thumb state High registers**
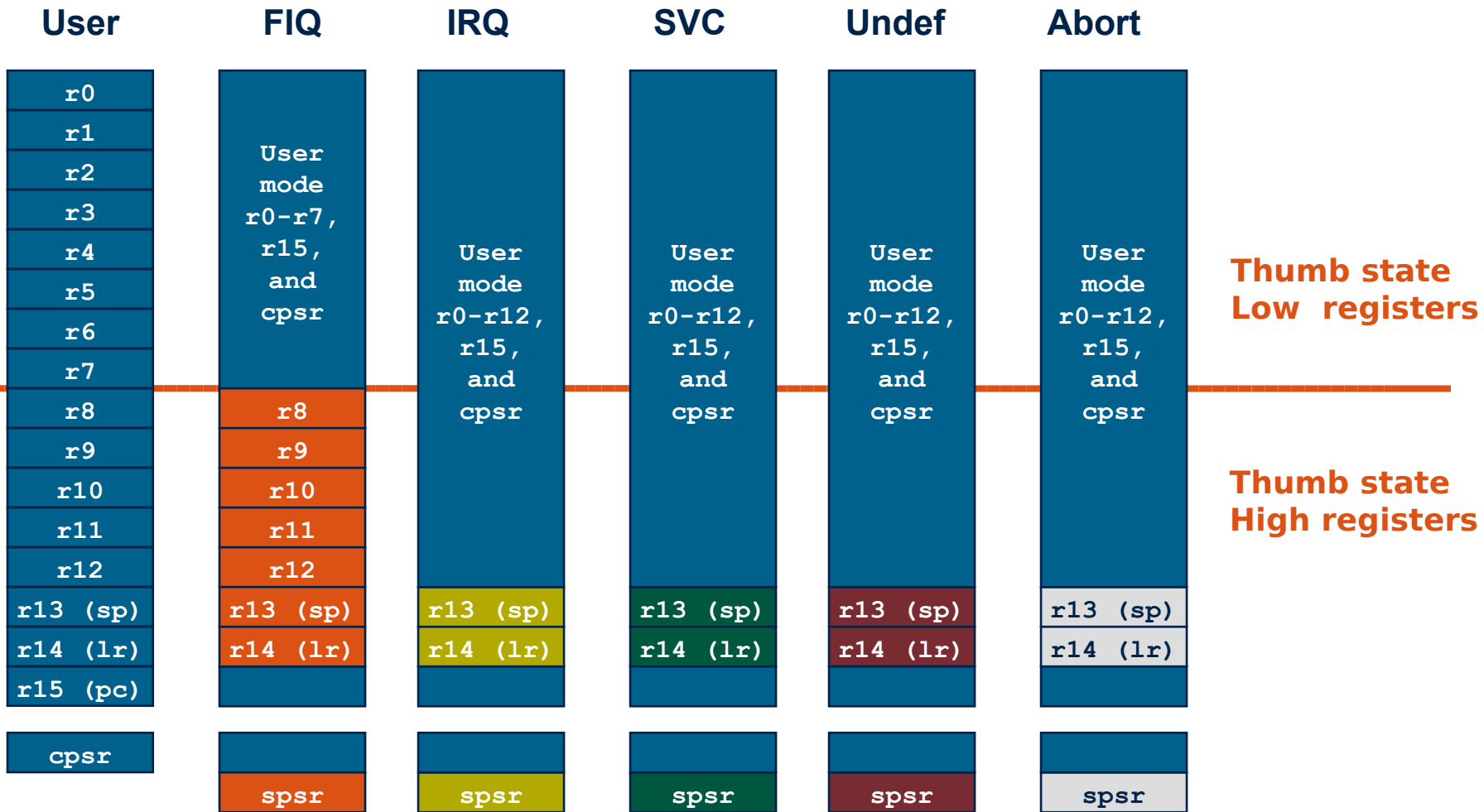
Note: System mode uses the User mode register set

ARM

- **ARM has 37 registers all of which are 32-bits long.**
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers

- **The current processor mode governs which of several banks is accessible. Each mode can access**
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr

- **Privileged modes (except System) can also access**
  - a particular spsr (saved program status register)

Program Status Registers

```
 31      28 27      24 23          16 15              8 7 6 5 4        0
┌──────────┬───┬──────┬──────────────┬────────────────┬───┬─┬────────┐
│ N Z C V  │ Q │    J │ U n d e f i  │ n e d          │I F│T│ mode   │
└──────────┴───┴──────┴──────────────┴────────────────┴───┴─┴────────┘
      f                      s               x               c
```

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
- Architecture 5TE/J only
- Indicates if saturation has occurred

J bit
- Architecture 5TEJ only
- J = 1: Processor in Jazelle state

Interrupt Disable bits.
- I = 1: Disables the IRQ.
- F = 1: Disables the FIQ.

T Bit
- Architecture xT only
- T = 0: Processor in ARM state
- T = 1: Processor in Thumb state

Mode bits
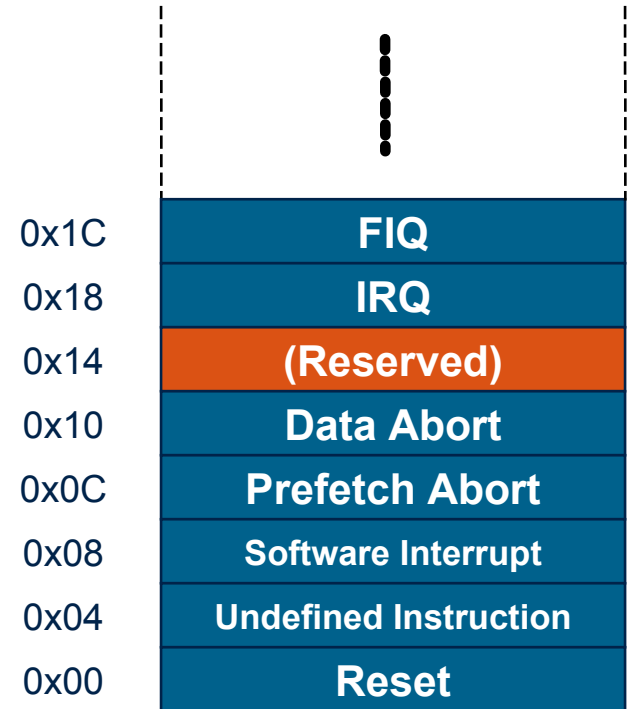- Specify the processor mode

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
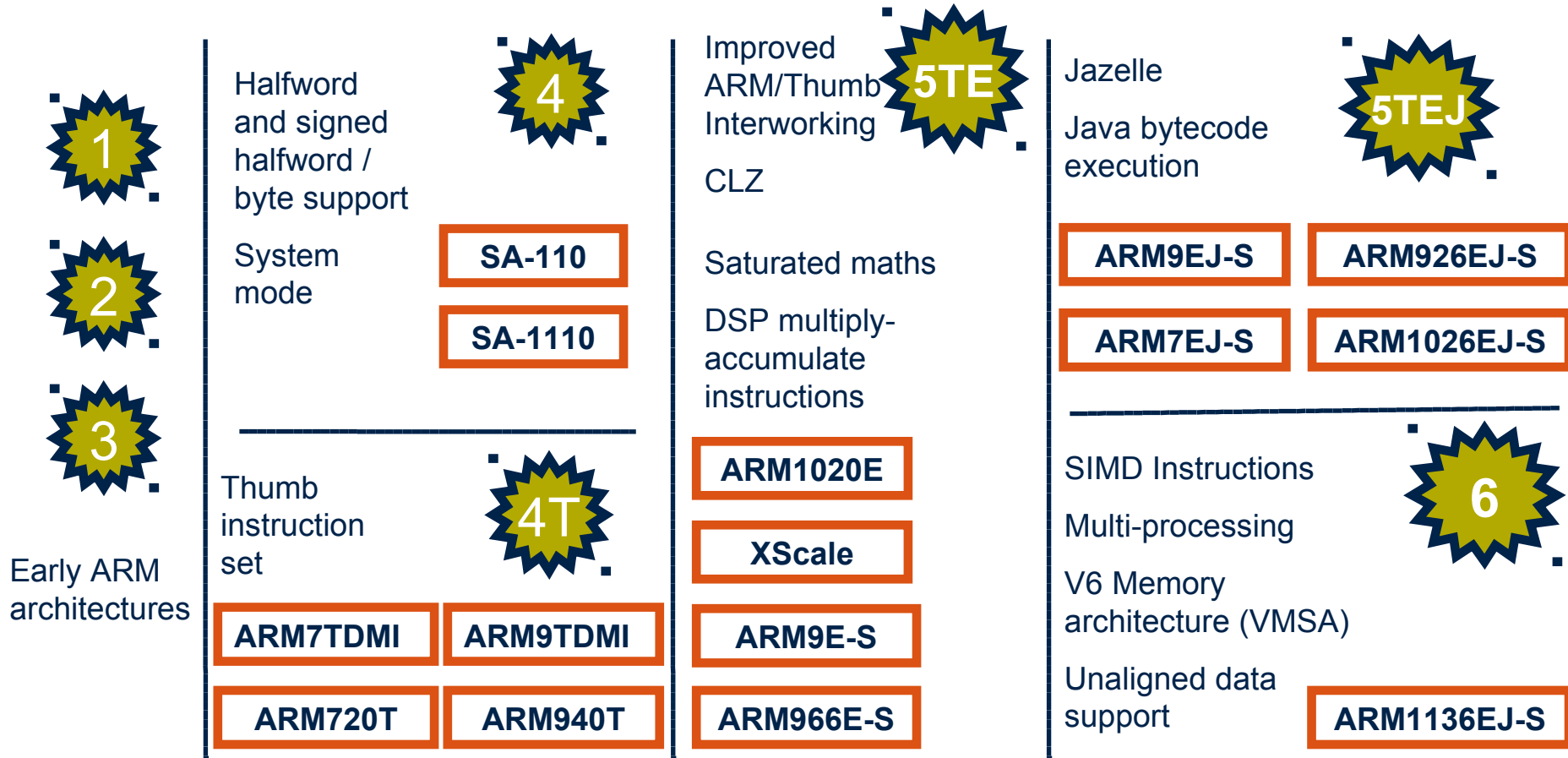
When an exception occurs, the ARM:

- Copies CPSR into SPSR_<mode>
- Sets appropriate CPSR bits
  - Change to ARM state
  - 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>

This can only be done in ARM state.

Address	
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

Vector table can be at `0xFFFF0000` on ARM720T and on ARM9/10 family devices

1

2

3

Early ARM architectures

Halfword and signed halfword / byte support

System mode

4

| SA-110 |

| SA-1110 |

Thumb instruction set

4T

| ARM7TDMI | ARM9TDMI |

| ARM720T | ARM940T |

Improved ARM/Thumb Interworking

5TE

CLZ

Saturated maths

DSP multiply-accumulate instructions

| ARM1020E |

| XScale |

| ARM9E-S |

| ARM966E-S |

Jazelle

Java bytecode execution

5TEJ

| ARM9EJ-S | ARM926EJ-S |

| ARM7EJ-S | ARM1026EJ-S |

SIMD Instructions

6

Multi-processing

V6 Memory architecture (VMSA)

Unaligned data support

| ARM1136EJ-S |

ARMxyzTDMIEJFS

x: series

y: MMU

z: cache

T: Thumb

D: debugger

M: Multiplier

I: EmbeddedICE (built-in debugger hardware)

E: Enhanced instruction

J: Jazelle (JVM)

F: Floating-point

S: Synthesizible version (source code version for EDA tools)

Introduction to ARM Ltd

Programmers Model

- **Instruction Sets**

System Design

Development Tools

ARM

- **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.
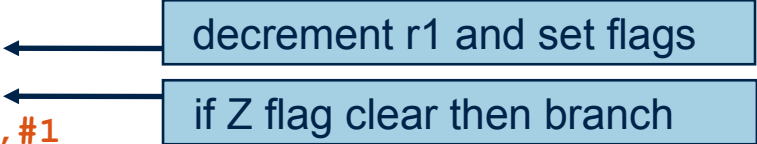
```
CMP    r3,#0                          CMP    r3,#0
 BEQ   skip                           ADDNE r0,r1,r2
 ADD   r0,r1,r2
skip
```

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

```
loop                    decrement r1 and set flags

   …                    if Z flag clear then branch
   SUBS r1,r1,#1
   BNE loop
```

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	

Examples of conditional execution

- **Use a sequence of several conditional instructions**

  ```
  if (a==0) func(1);
      CMP       r0,#0
      MOVEQ     r0,#1
      BLEQ      func
  ```
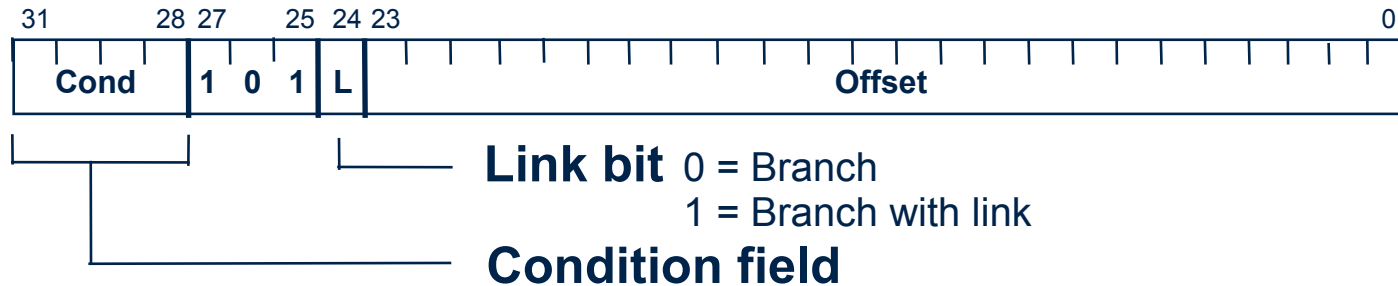
- **Set the flags, then use various condition codes**

  ```
  if (a==0) x=0;
  if (a>0)  x=1;
      CMP       r0,#0
      MOVEQ     r1,#0
      MOVGT     r1,#1
  ```

- **Use conditional compare instructions**

  ```
  if (a==4 || a==10) x=0;
      CMP       r0,#4
      CMPNE     r0,#10
      MOVEQ     r1,#0
  ```

Branch instructions

- **Branch :**   `B{<cond>} label`

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

31	28	27	25	24	23	0

```
  Cond    1  0  1  L              Offset
```

Link bit 0 = Branch
              1 = Branch with link

Condition field

- **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?

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:**

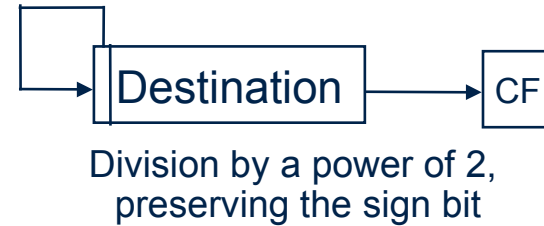 `<Operation>{<cond>}{S} Rd, Rn, Operand2`

  - 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.**
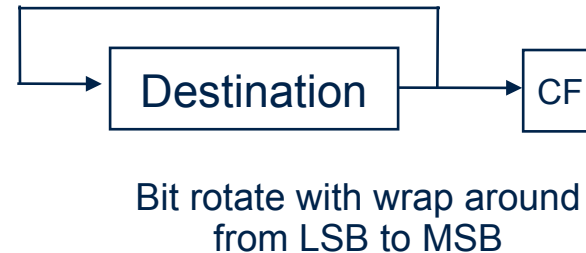
ARM

LSL : Logical Left Shift

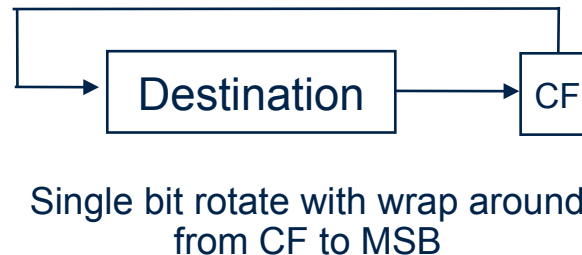Multiplication by a power of 2

LSR : Logical Shift Right

Division by a power of 2
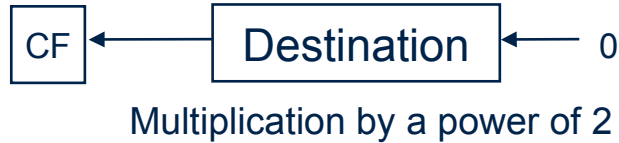
ASR: Arithmetic Right Shift

Division by a power of 2, preserving the sign bit

ROR: Rotate Right

Bit rotate with wrap around from LSB to MSB

RRX: Rotate Right Extended

Single bit rotate with wrap around from CF to MSB

Operand 1   **Operand 2** ← **Register, optionally with shift operation**

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Barrel Shifter

ALU

Result

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers
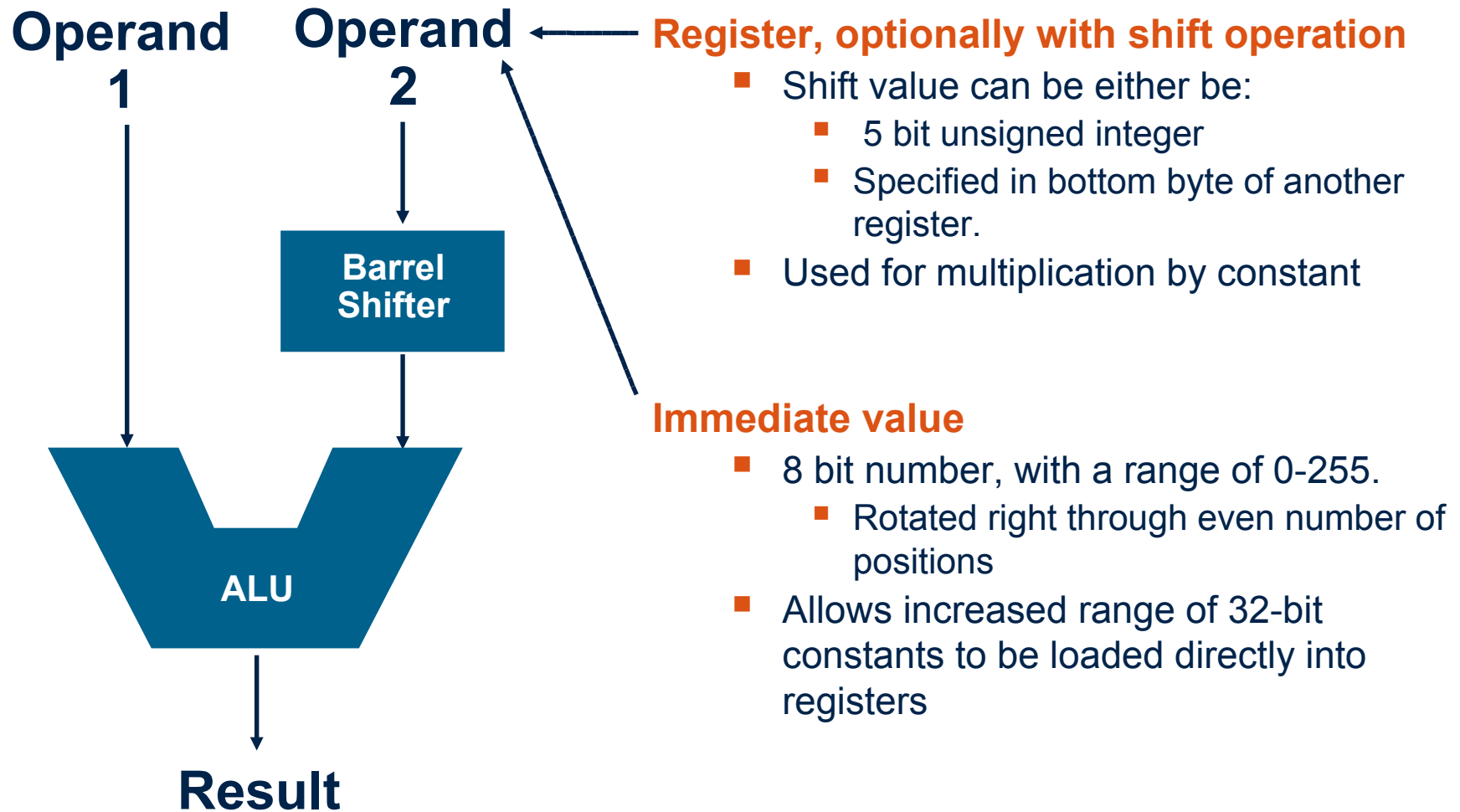
- **No ARM instruction can contain a 32 bit immediate constant**
  - All ARM instructions are fixed as 32 bits long

- **The data processing instruction format has 12 bits available for operand2**
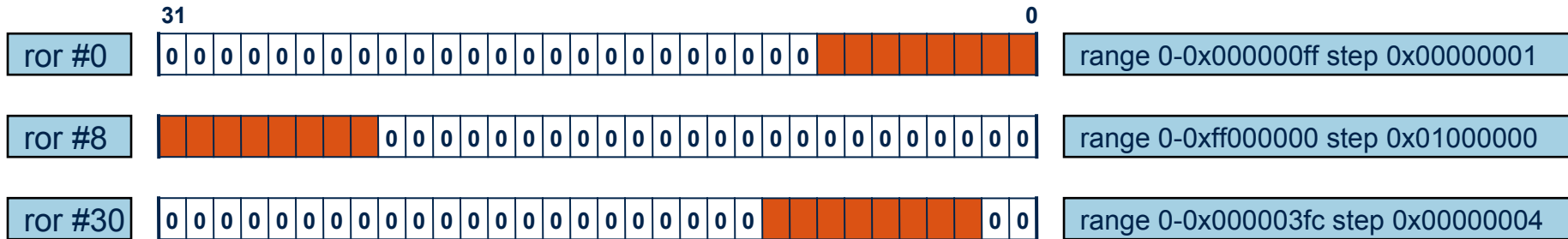
```
     11      8 7            0
_____
            | rot |  immed_8  |
_____
         x2    |      |
               ↓      ↓
          ┌─────────────┐
          │  Shifter    │
          │    ROR      │
          └─────────────┘
                 ↓
```
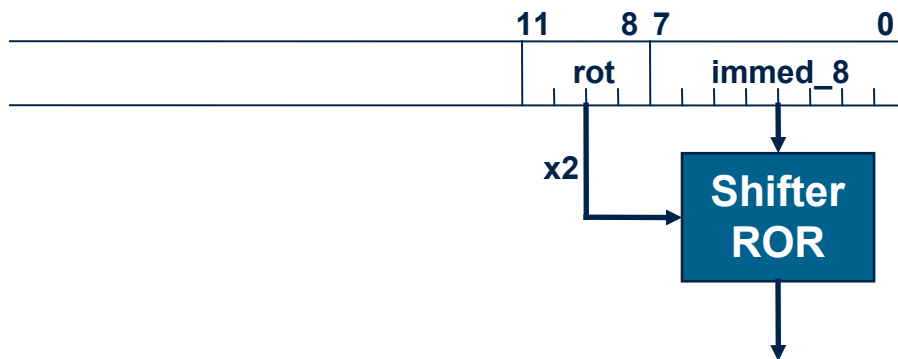
Quick Quiz:
`0xe3a004ff`
`MOV r0, #???`

- **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 shifted by an even number of bit positions".**

Immediate constants (2)

- **Examples:**

	31		0
ror #0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ▮▮▮▮▮▮▮▮		range 0-0x000000ff step 0x00000001
ror #8	▮▮▮▮▮▮▮▮ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		range 0-0xff000000 step 0x01000000
ror #30	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ▮▮▮▮▮▮▮▮ 0 0		range 0-0x000003fc step 0x00000004

- **The assembler converts immediate values to the rotate form:**
    - `MOV r0,#4096          ; uses 0x40 ror 26`
    - `ADD r1,r2,#0xFF0000 ; uses 0xFF ror 16`

- **The bitwise complements can also be formed using MVN:**
    - `MOV r0, #0xFFFFFFFF            ; assembles to MVN r0,#0`

- **Values that cannot be generated in this way will cause an error.**

Loading 32 bit constants

- **To allow larger constants to be loaded, the assembler offers a pseudo-instruction:**
  - `LDR rd, =const`

- **This will either:**
  - Produce a **MOV** or **MVN** instruction to generate the value (if possible).

  or
  - Generate a **LDR** instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).

- **For example**
  - `LDR r0,=0xFF`   =>   `MOV r0,#0xFF`
  - `LDR r0,=0x55555555`   =>   `LDR r0,[PC,#Imm12]`
                                       `...`
                                       `...`
                                       `DCD 0x55555555`

- **This is the recommended way of loading constants into a register**

- **Syntax:**
  - MUL{<cond>}{S} Rd, Rm, Rs   Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn   Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs RdHi,RdLo := (Rm*Rs)+RdHi,RdLo

- **Cycle time**
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for "long"

- **Above are "general rules" - refer to the TRM for the core you are using for the exact details**

Single register data transfer

LDR	**STR**	Word
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- **Memory system must support all access sizes**

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

  e.g. **LDREQB**

- **Address accessed by LDR/STR is specified by a base register plus an offset**

- **For word and unsigned byte accesses, offset can be**
    - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).
        ```
        LDR r0,[r1,#8]
        ```
    - A register, optionally shifted by an immediate value
        ```
        LDR r0,[r1,r2]
        LDR r0,[r1,r2,LSL#2]
        ```

- **This can be either added or subtracted from the base register:**
    ```
    LDR r0,[r1,#-8]
    LDR r0,[r1,-r2]
    LDR r0,[r1,-r2,LSL#2]
    ```

- **For halfword and signed halfword / byte, offset can be:**
    - An unsigned 8 bit immediate value (ie 0-255 bytes).
    - A register (unshifted).

- **Choice of *pre-indexed* or *post-indexed* addressing**

- **Pre-indexed: `STR r0,[r1,#12]`**

Offset

| 12 | → 0x20c

r0

Source Register for STR 0x5

0x5

r1

Base Register 0x200 → 0x200

Auto-update form: `STR r0,[r1,#12]!`

- **Post-indexed: `STR r0,[r1],#12`**

Updated Base Register **r1** 0x20c ← **Offset** 12

0x20c

Original Base Register **r1** 0x200 → 0x200

r0

Source Register for STR 0x5

0x5
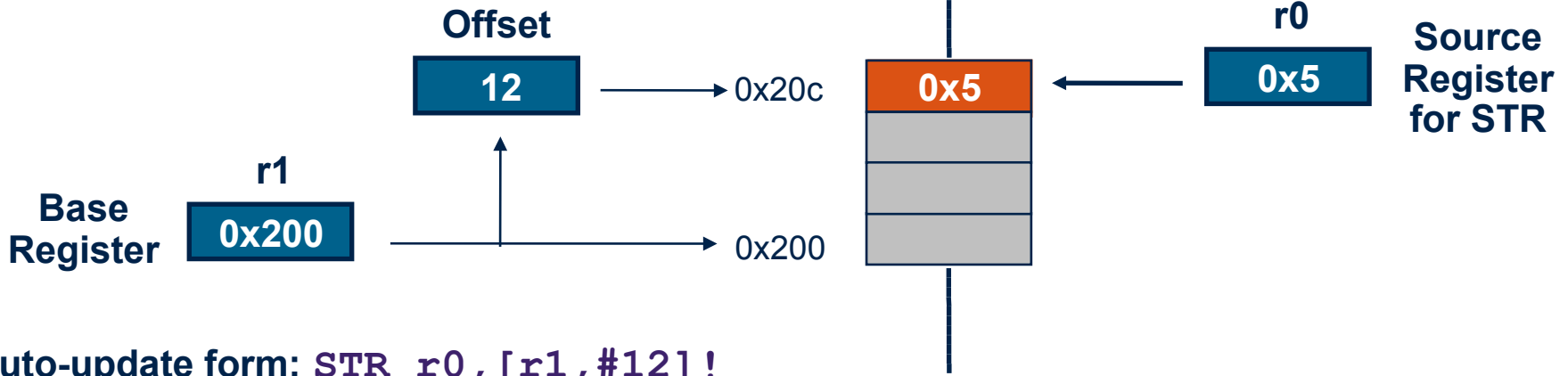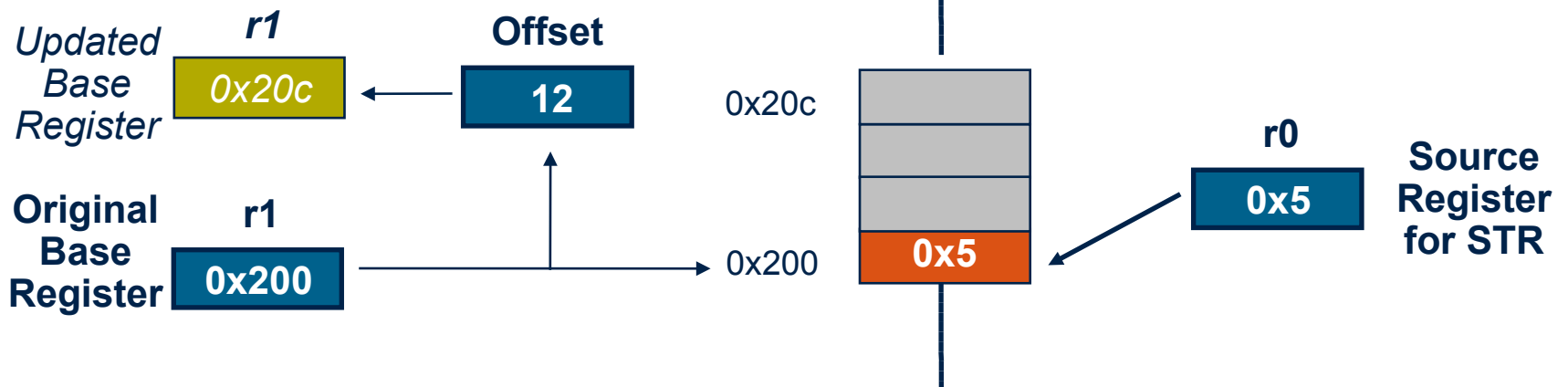
- **Syntax:**

  `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>

- **4 addressing modes:**

`LDMIA` / `STMIA`	increment after
`LDMIB` / `STMIB`	increment before
`LDMDA` / `STMDA`	decrement after
`LDMDB` / `STMDB`	decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```

Base Register (Rb) r10

Increasing Address

Software Interrupt (SWI)

31	28	27	24	23		0
Cond		1 1 1 1		SWI number (ignored by processor)		

Condition Field
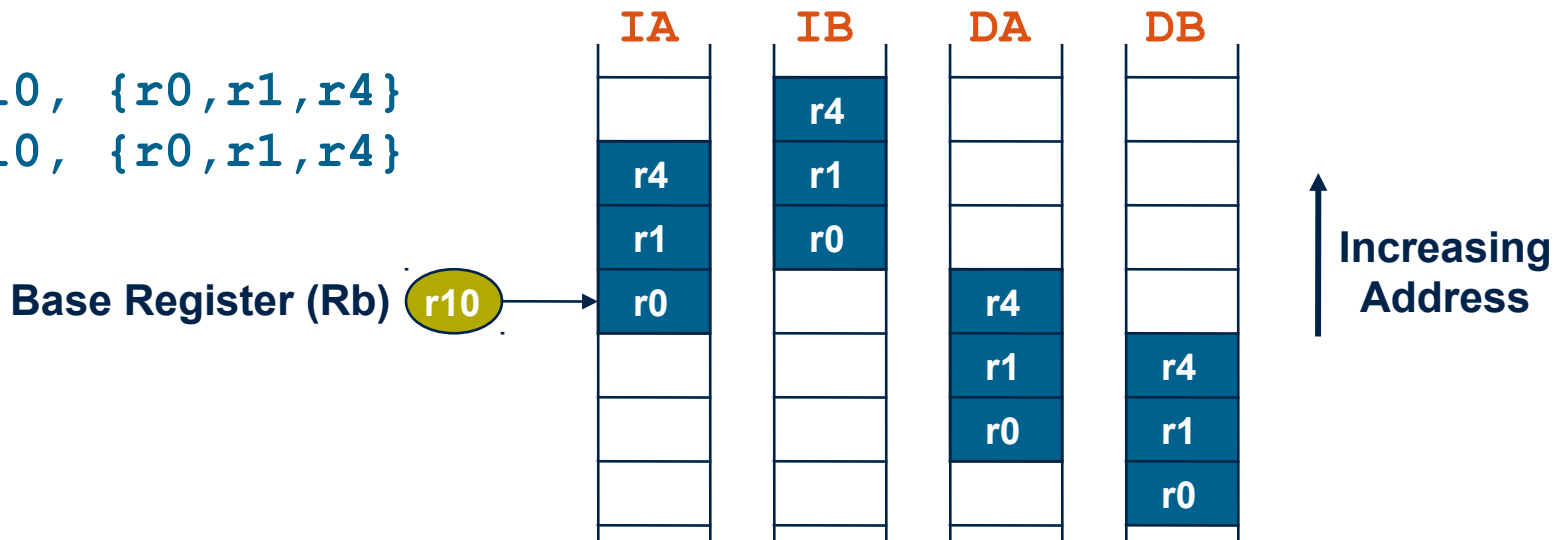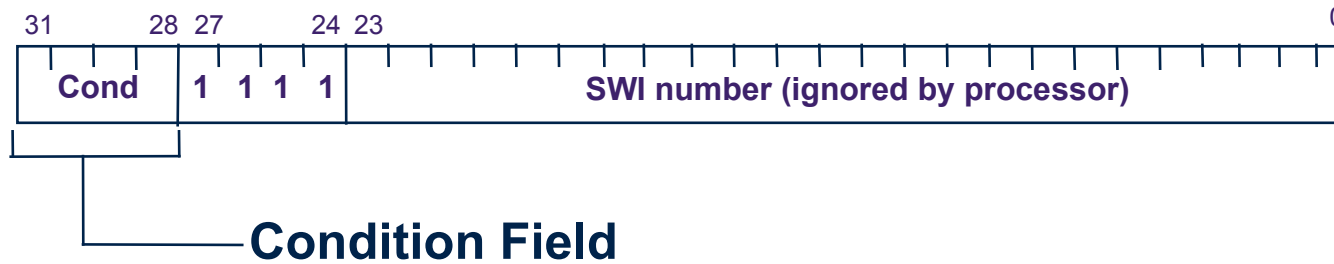
- **Causes an exception trap to the SWI hardware vector**

- **The SWI handler can examine the SWI number to decide what operation has been requested.**

- **By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.**

- **Syntax:**
  - `SWI{<cond>} <SWI number>`

31			28	27		24	23				16	15			8	7	6	5	4		0
N	Z	C	V	Q		J	U	n	d e f	i n e d		i n	e	d		I	F	T		mode	
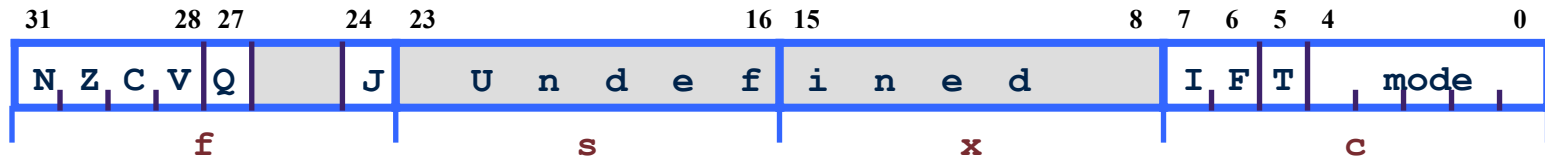
| f | s | x | c |

- **MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register.**

- **Syntax:**
    - **MRS**`{<cond>} Rd,<psr>`   `; Rd = <psr>`
    - **MSR**`{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm`

  where
    - `<psr> = CPSR or SPSR`
    - `[_fields] = any combination of 'fsxc'`

- **Also an immediate form**
    - **MSR**`{<cond>} <psr_fields>,#Immediate`

- **In User Mode, all bits can be read but only the condition flags (_f) can be written.**

ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.

- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked

func1

func2

```
BL func1
```

```
STMFD sp!,
{regs,lr}
   :
BL func2
   :
LDMFD sp!,
{regs,pc}
```

```
MOV pc, lr
```

ARM
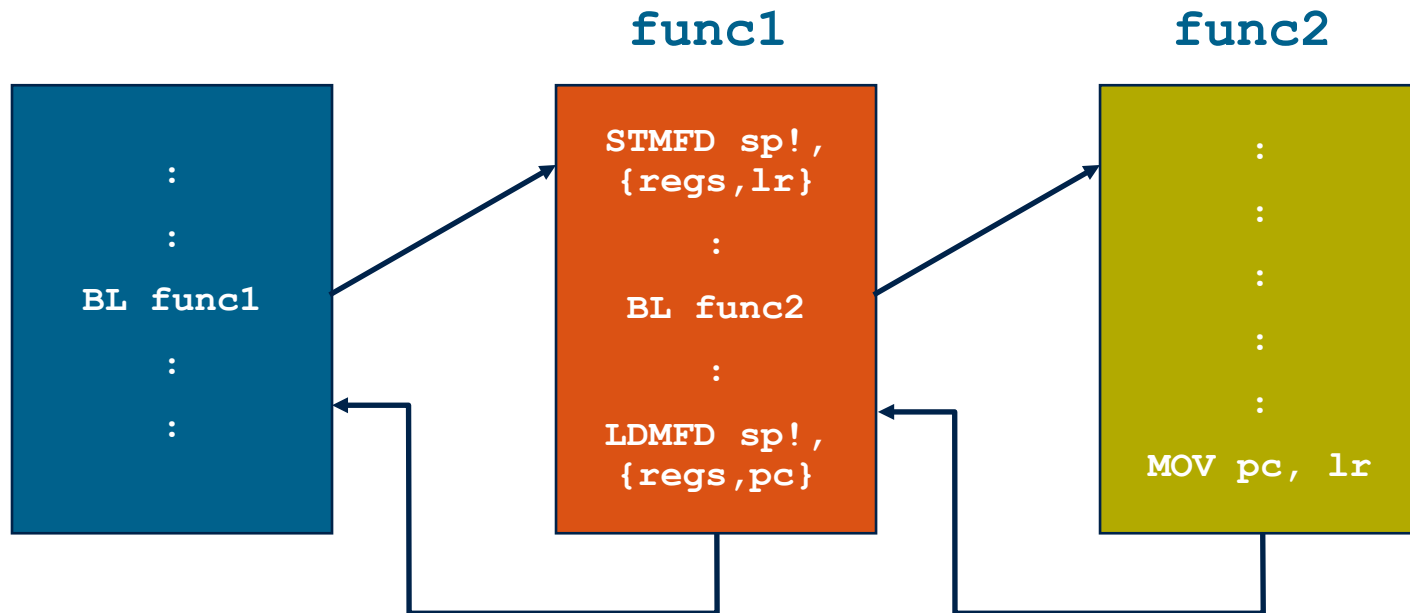
- **Thumb is a 16-bit instruction set**
  - Optimised for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set

- **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

Introduction

Programmers Model

Instruction Sets

- **System Design**

Development Tools

ARM

16 bit RAM

8 bit ROM

32 bit RAM

Interrupt Controller

nIRQ   nFIQ

ARM Core

Peripherals

I/O
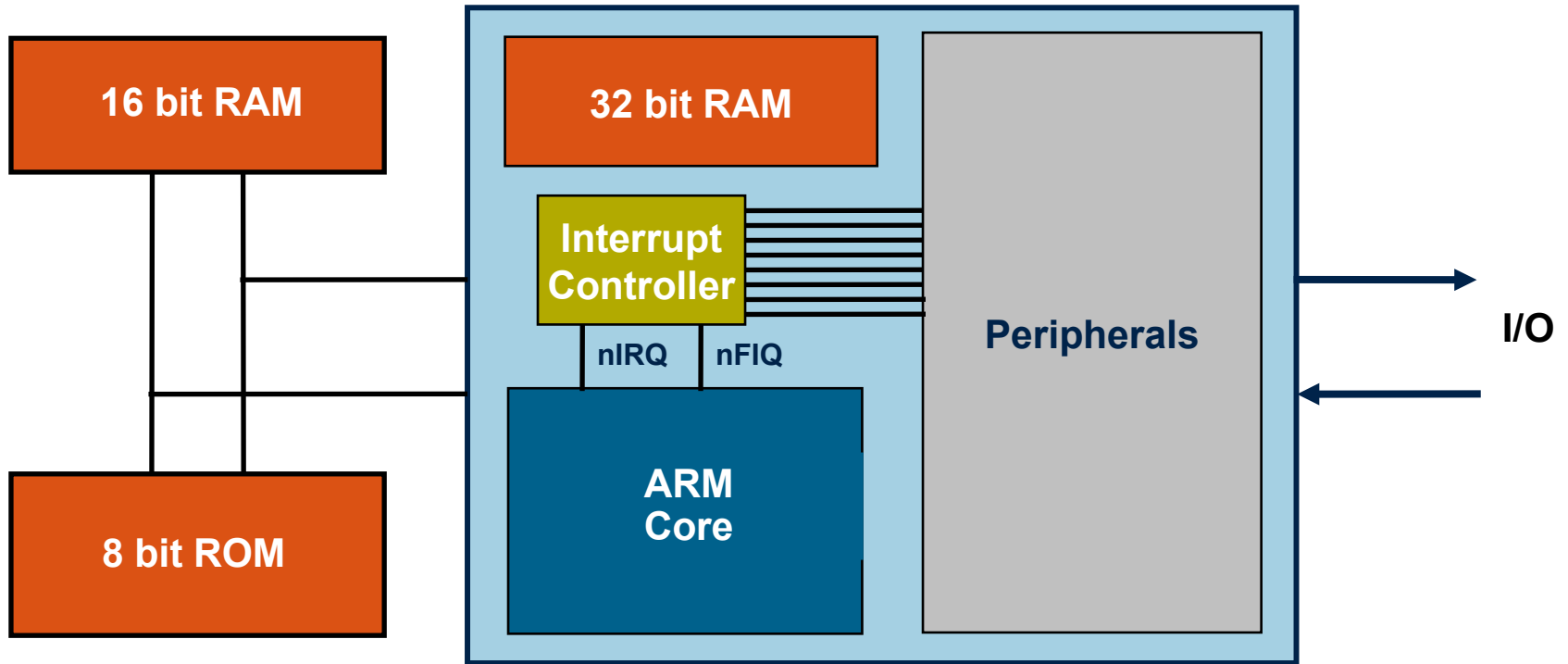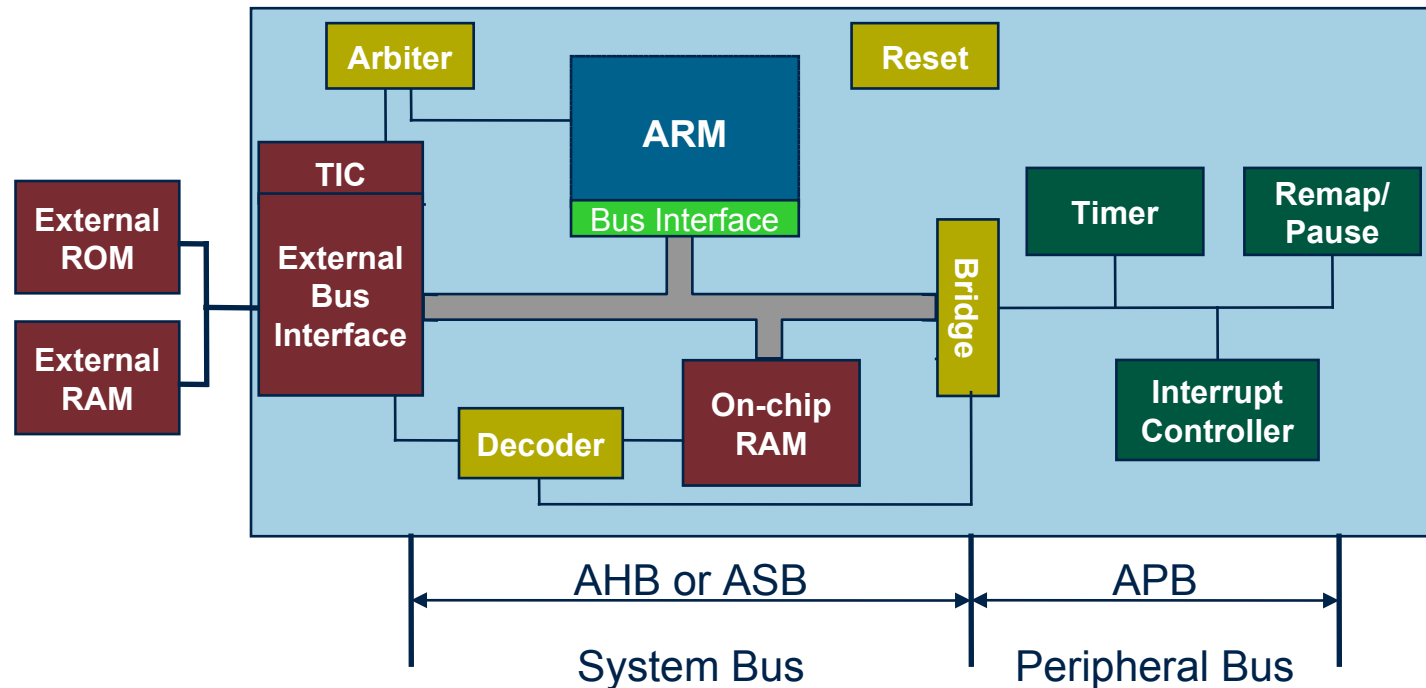
- **AMBA**
  - Advanced Microcontroller Bus Architecture
- **ADK**
  - Complete AMBA Design Kit
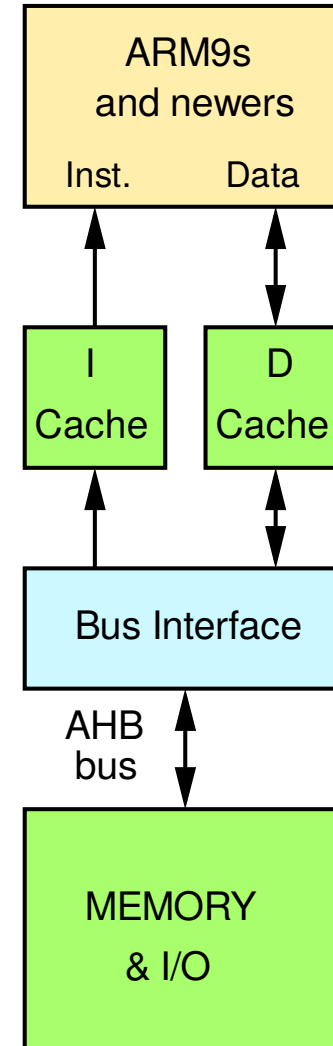
- **ACT**
  - AMBA Compliance Testbench
- **PrimeCell**
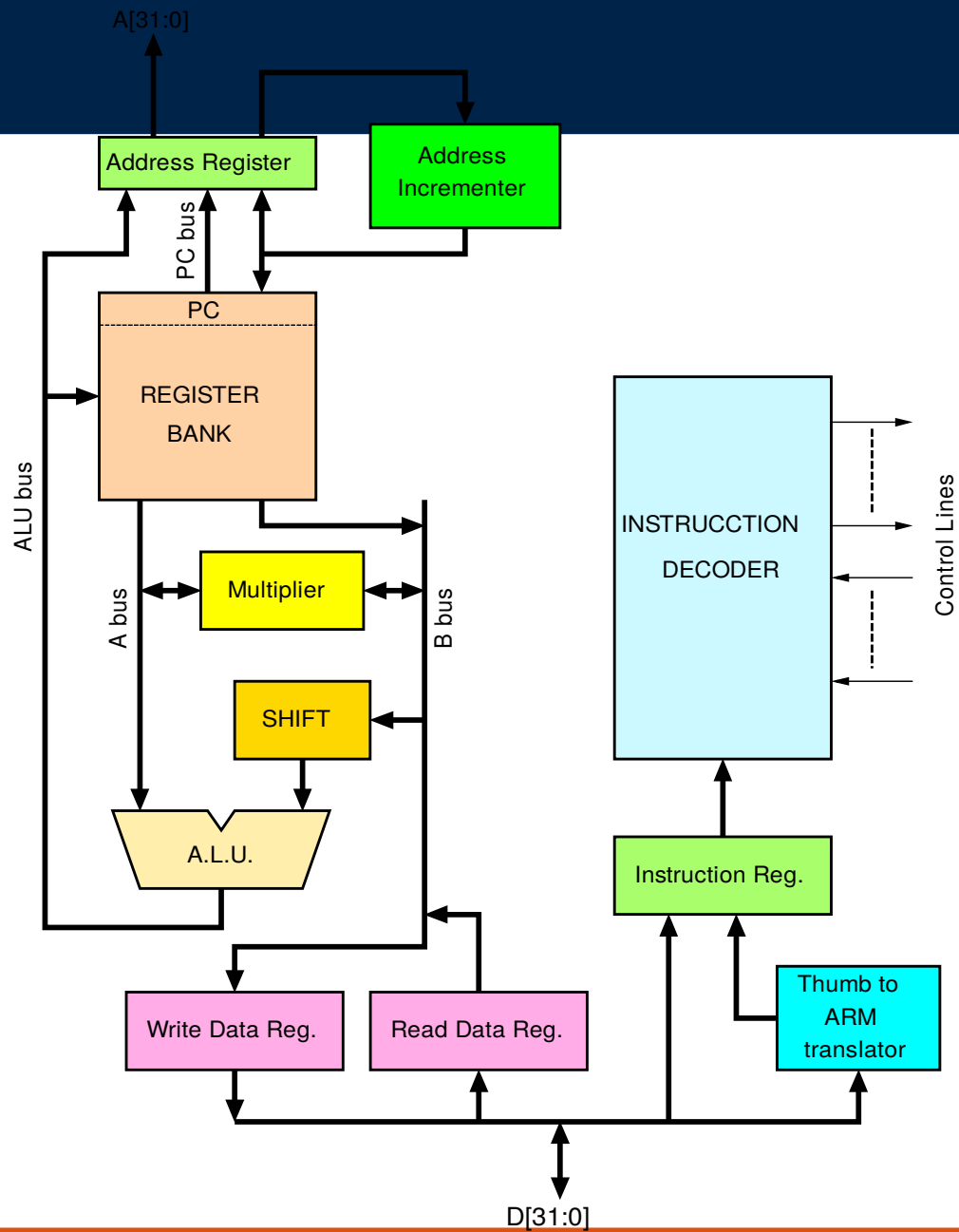 - ARM's AMBA compliant peripherals

Von Neumann
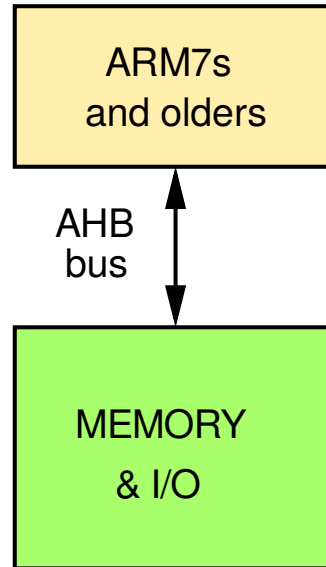
Harvard

Memory-mapped I/O:
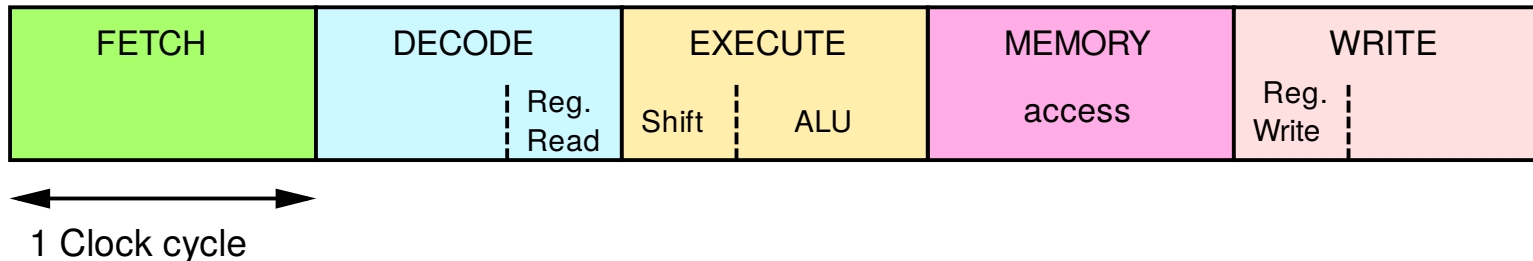
- No specific instructions for I/O (use Load/Store instr. instead)
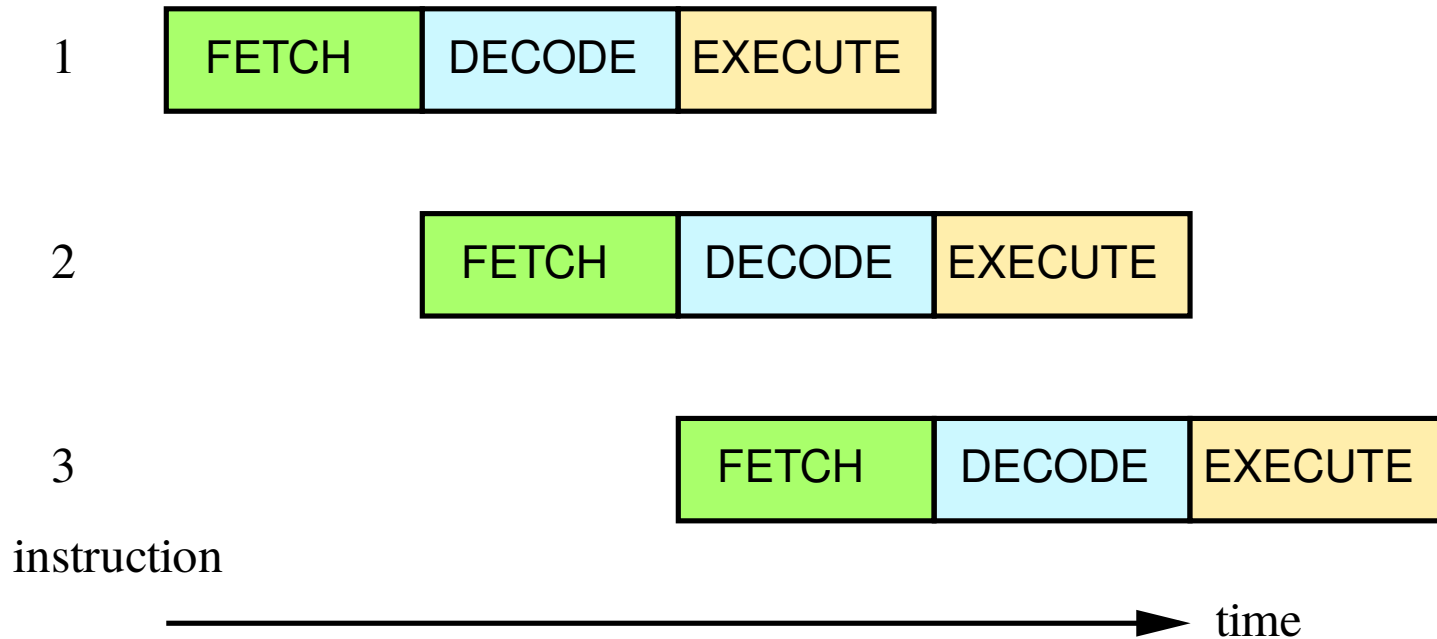- Peripheral's registers at some memory addresses

ARM Pipelining examples

ARM7TDMI Pipeline

FETCH	DECODE	EXECUTE			
		Reg. Read	Shift	ALU	Reg. Write

←— 1 Clock cycle —→

ARM9TDMI Pipeline

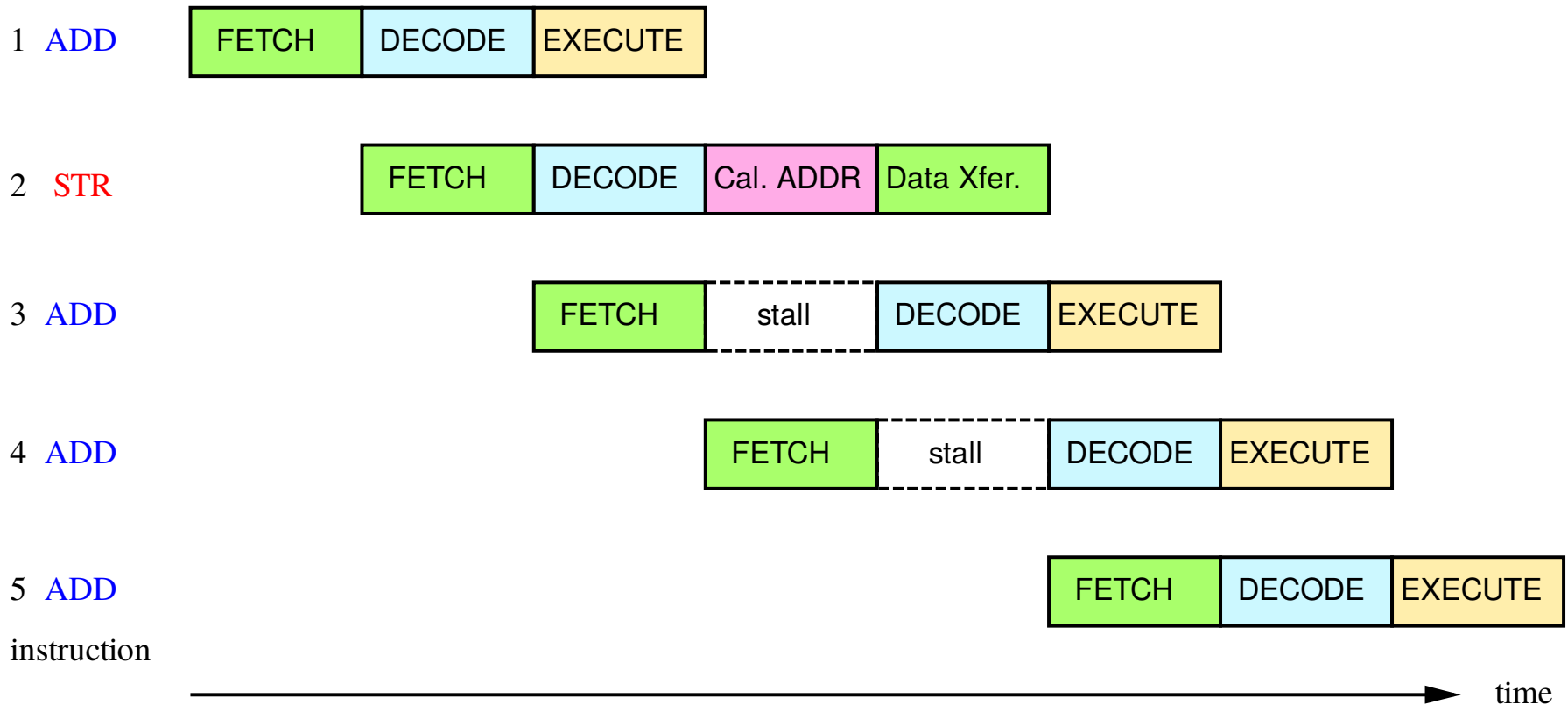| FETCH	DECODE		EXECUTE		MEMORY access	WRITE
		Reg. Read	Shift	ALU		Reg. Write
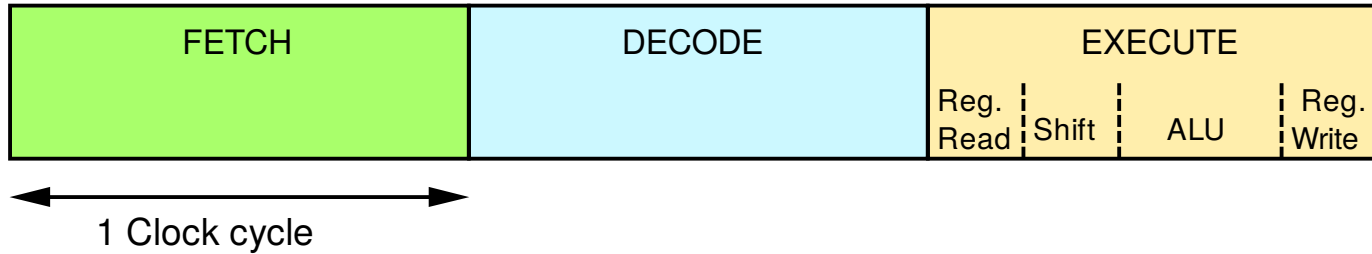
←— 1 Clock cycle —→

- **Fetch**: Read Op-code from memory to internal Instruction Register

- **Decode**: Activate the appropriate control lines depending on Opcode

- **Execute**: Do the actual processing

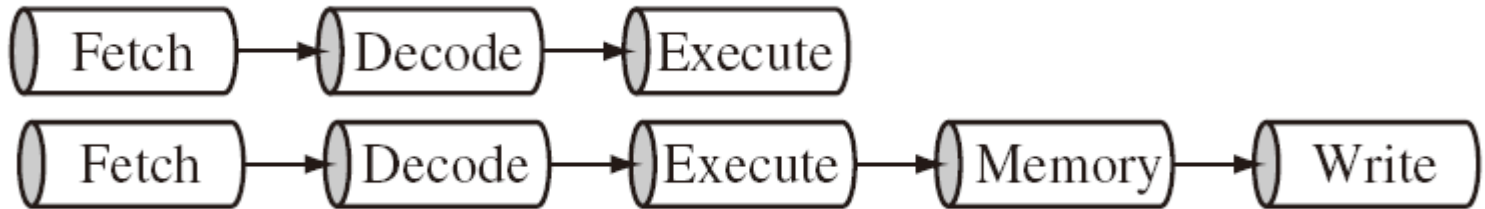- Simple instructions (like ADD) Complete at a rate of one per cycle
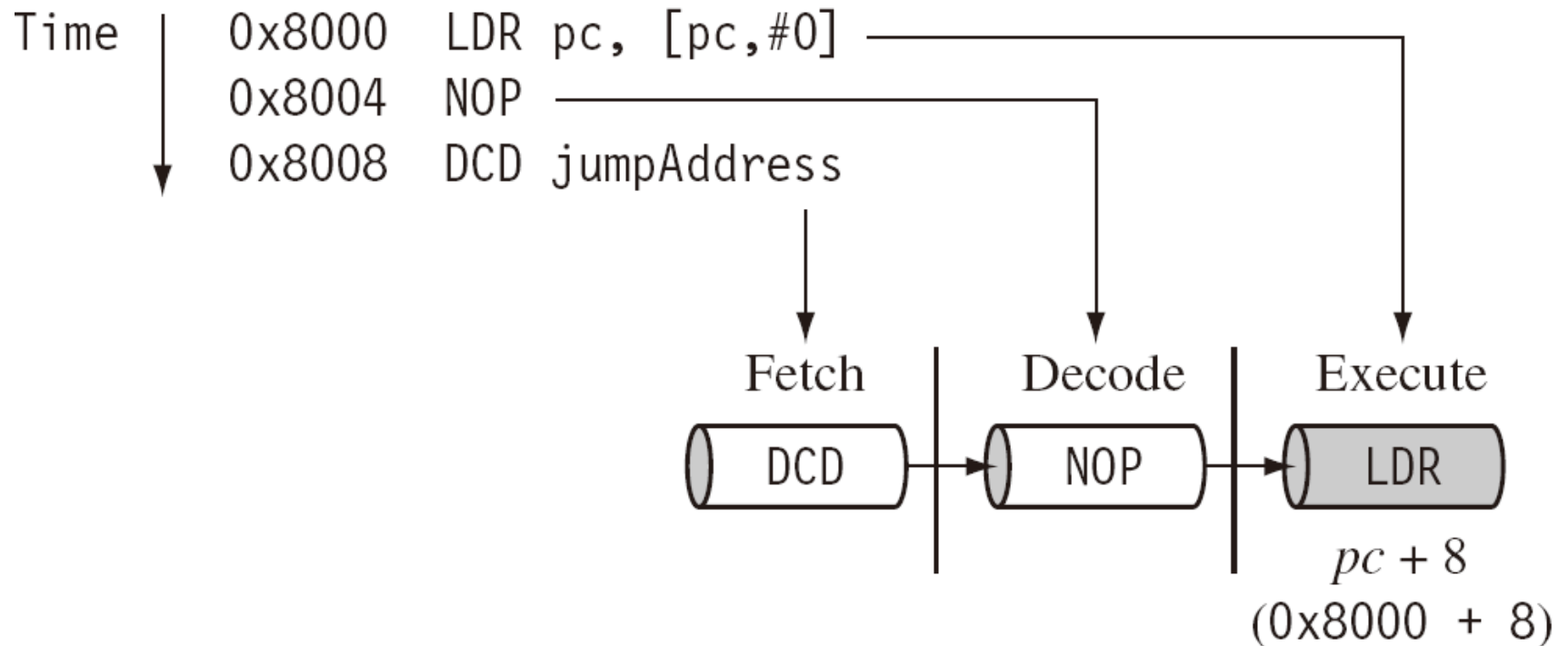
• More complex instructions:

1 ADD	FETCH	DECODE	EXECUTE	
2 STR	FETCH	DECODE	Cal. ADDR	Data Xfer.
3 ADD	FETCH	stall	DECODE	EXECUTE
4 ADD	FETCH	stall	DECODE	EXECUTE
5 ADD	FETCH	DECODE	EXECUTE	

instruction

time →

STR : 2 effective clock cycles (+1 cycle)

ARM7

ARM9

In execution, pc always 8 bytes ahead


```
Time    0x8000   LDR pc, [pc,#0]
        0x8004   NOP
        0x8008   DCD jumpAddress
```

Fetch — DCD

Decode — NOP

Execute — LDR

$pc + 8$
$(0x8000 + 8)$

Cortex-A8

- **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

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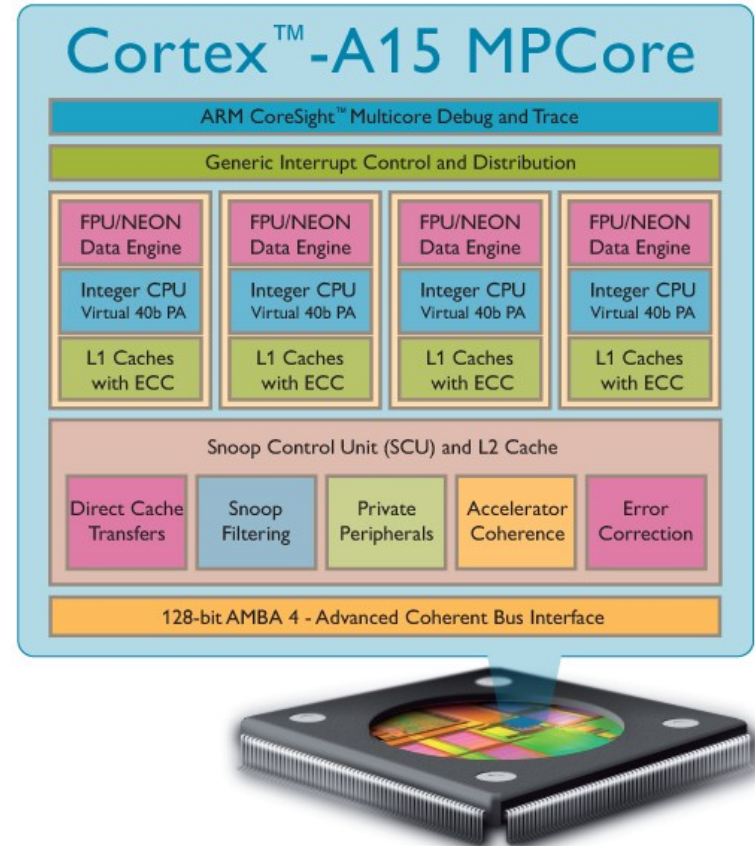 - NEON and VFP implemented at end of pipeline

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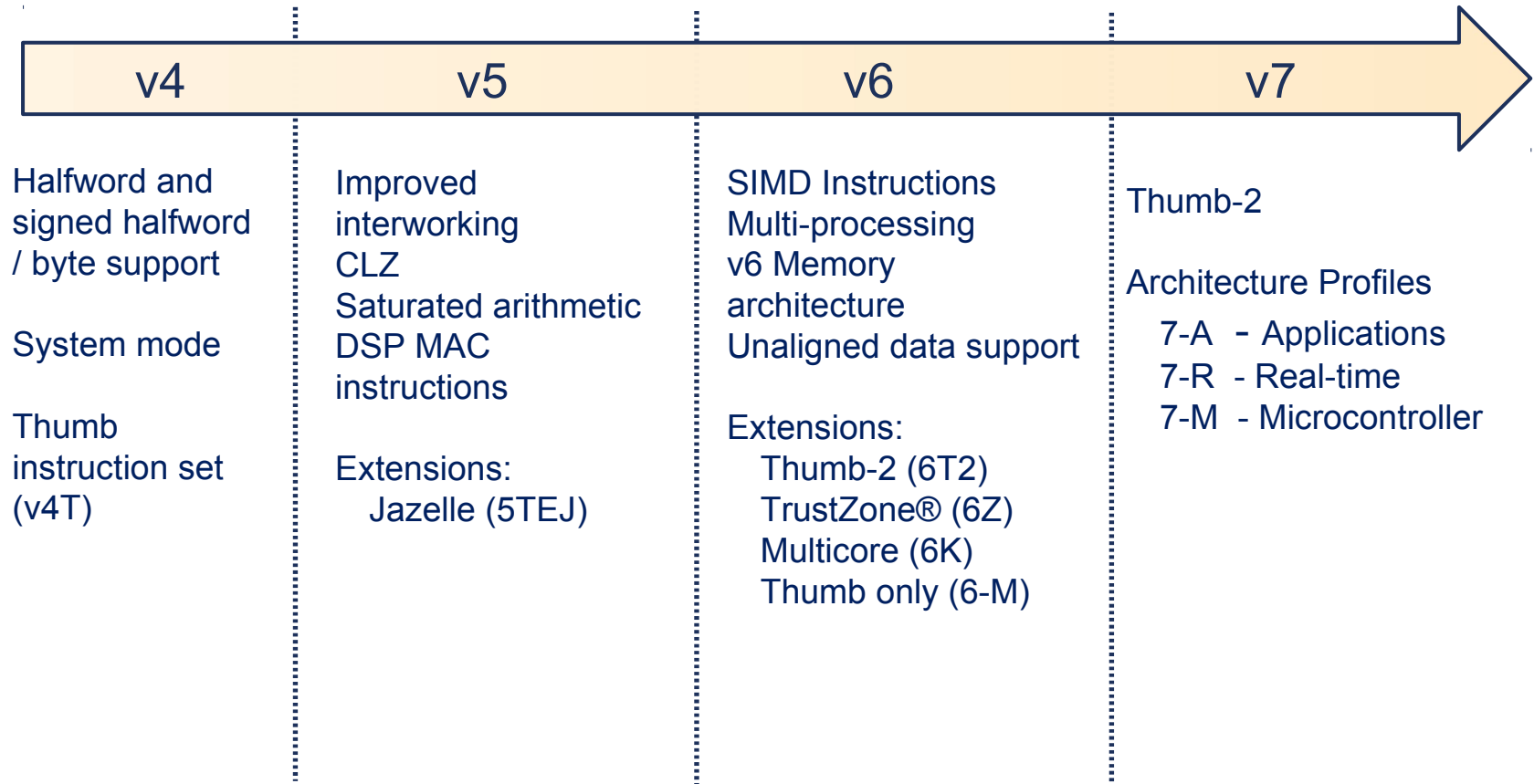 - 4-way set-associative
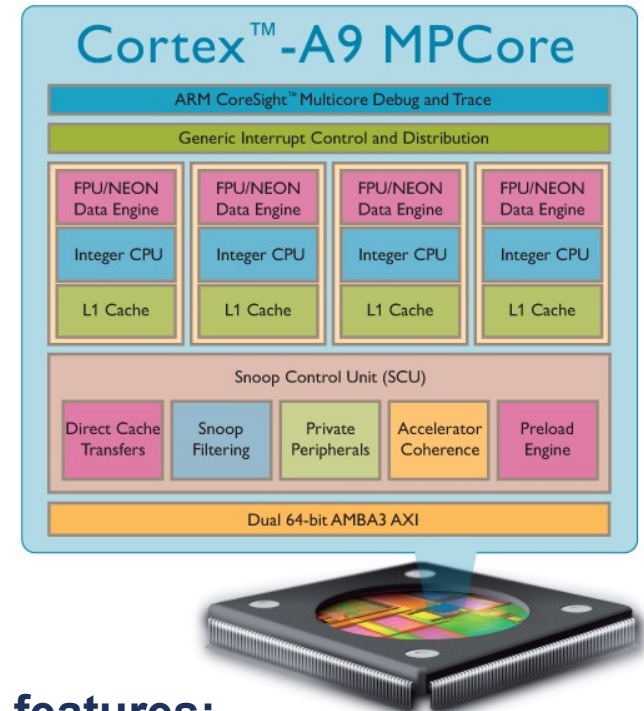
- **Optional features:**
  - PTM instruction trace interface
  - IEM power saving support
  - Full Jazelle DBX support
  - VFPv3-D16 Floating-Point Unit (FPU) or NEON™ media processing engine

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**

Development of the ARM Architecture

v4	v5	v6	v7
Halfword and signed halfword / byte support	Improved interworking	SIMD Instructions	Thumb-2
	CLZ	Multi-processing	
System mode	Saturated arithmetic	v6 Memory architecture	Architecture Profiles
	DSP MAC instructions	Unaligned data support	7-A - Applications
Thumb instruction set (v4T)			7-R - Real-time
	Extensions:	Extensions:	7-M - Microcontroller
	Jazelle (5TEJ)	Thumb-2 (6T2)	
		TrustZone® (6Z)	
		Multicore (6K)	
		Thumb only (6-M)	

- **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

THE ARCHITECTURE FOR THE DIGITAL WORLD®

ARM®

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?