

■ Introduction

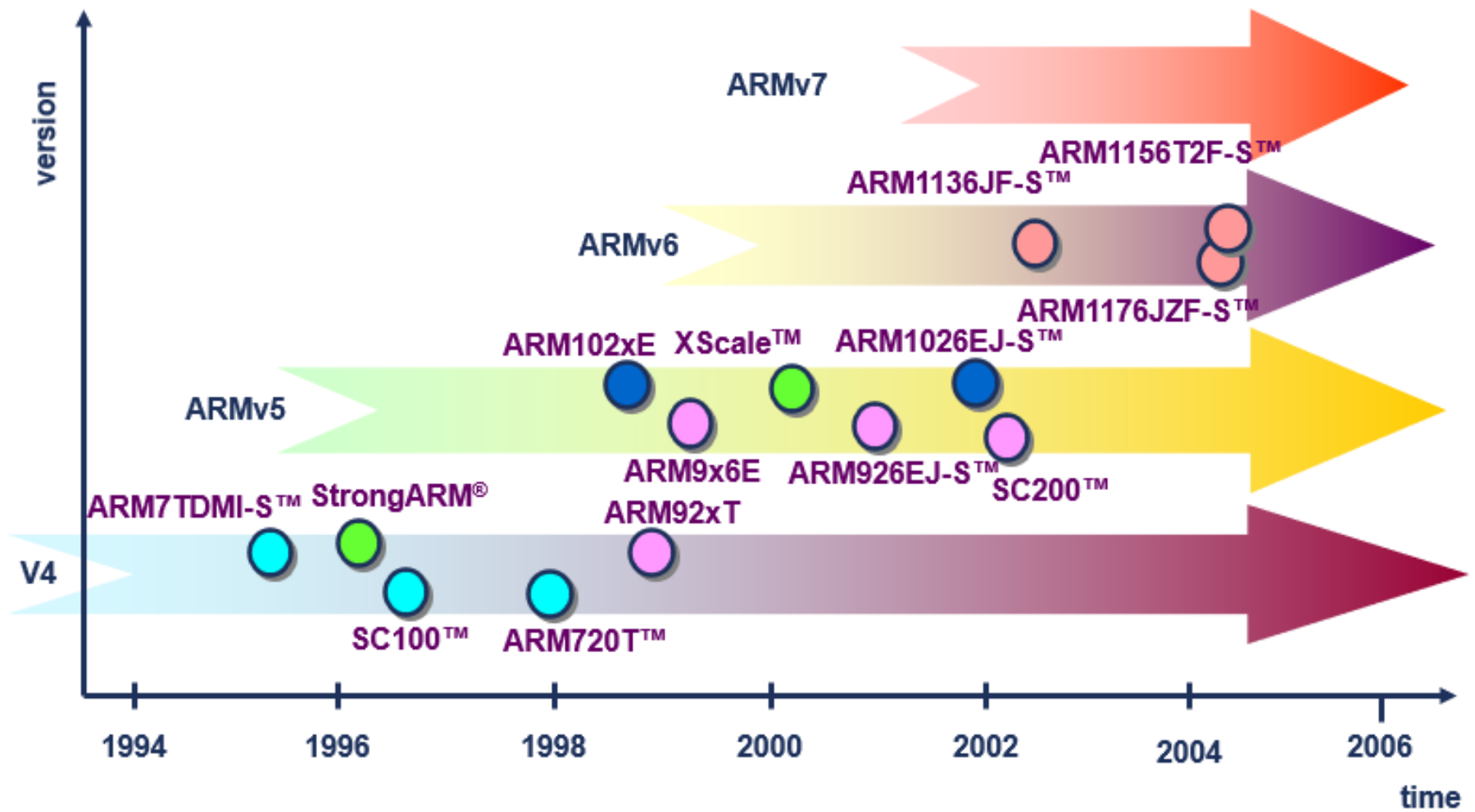
Architecture

Programmers Model

Instruction Set

- **ARM (Acorn RISC Machine)** started as a new, powerful, CPU design for the replacement of the 8-bit 6502 in Acorn Computers (Cambridge, UK, 1985)
- **First models had only a 26-bit program counter, limiting the memory space to 64 MB** (not too much by today standards, but a lot at that time).
- **1990 spin-off: ARM renamed Advanced RISC Machines**
- **ARM now focuses on Embedded CPU cores**
 - IP licensing: Almost every silicon manufacturer sells some microcontroller with an ARM core. Some even compete with their own designs.
 - Processing power with low current consumption
 - Good MIPS/Watt figure
 - Ideal for portable devices
 - Compact memories: 16-bit opcodes (Thumb)
- **New cores with added features**
 - Harvard architecture (ARM9, ARM11, Cortex)
 - Floating point arithmetic
 - Vector computing (VFP, NEON)
 - Java language (Jazelle)

- **32-bit CPU**
- **3-operand instructions (typical):** **ADD Rd,Rn,Operand2**
- **RISC design...**
 - Few, simple, instructions
 - Load/store architecture (instructions operate on registers, not memory)
 - Large register set
 - Pipelined execution
- **... And some very specific details**
 - No stack. Link register instead
 - PC as a regular register
 - Conditional execution of all instructions
 - Flags altered or not by data processing instructions (selectable)
 - Concurrent shifts/rotations (at the same time of other processing)
 - ...



Introduction

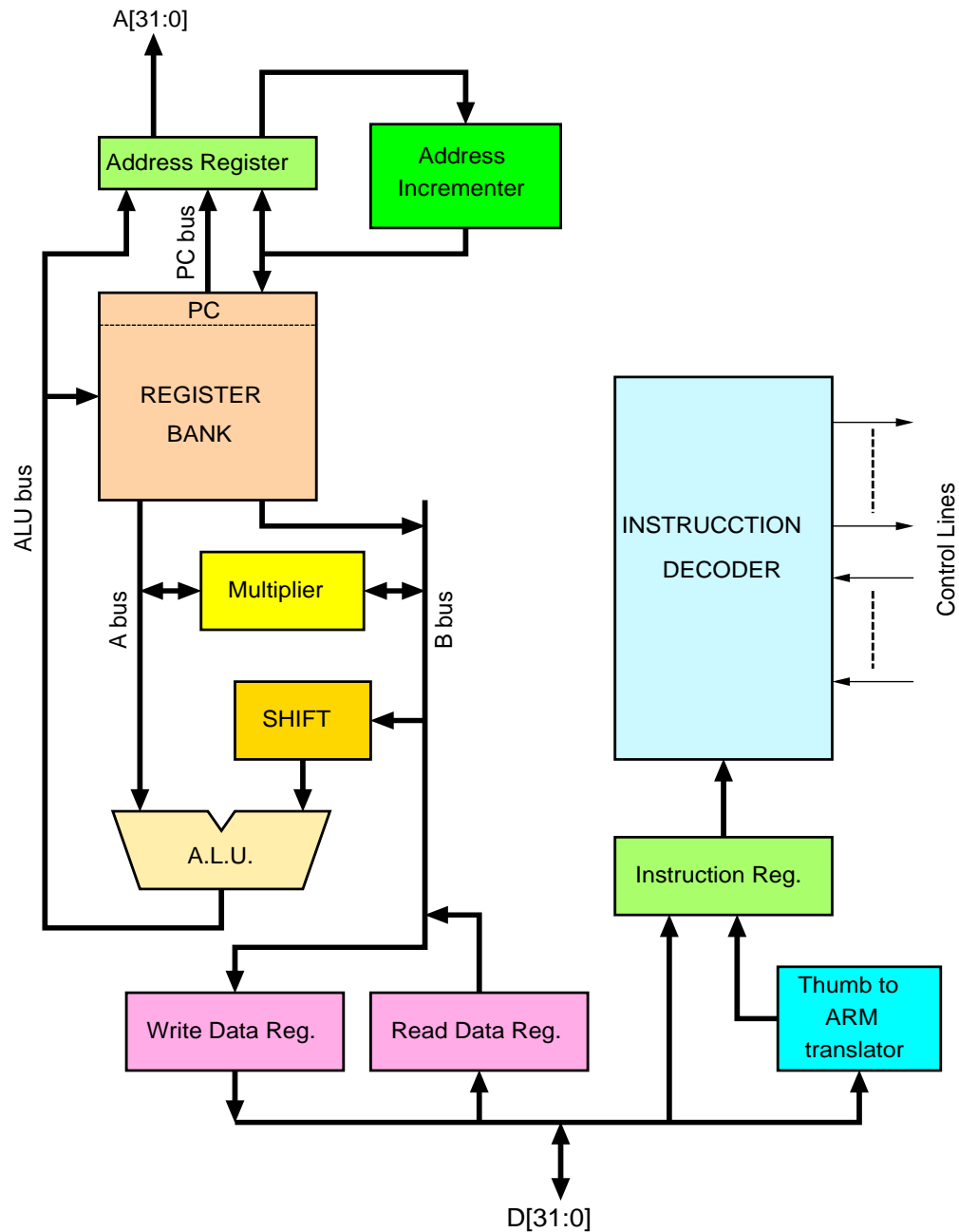
■ **Architecture**

Programmers Model

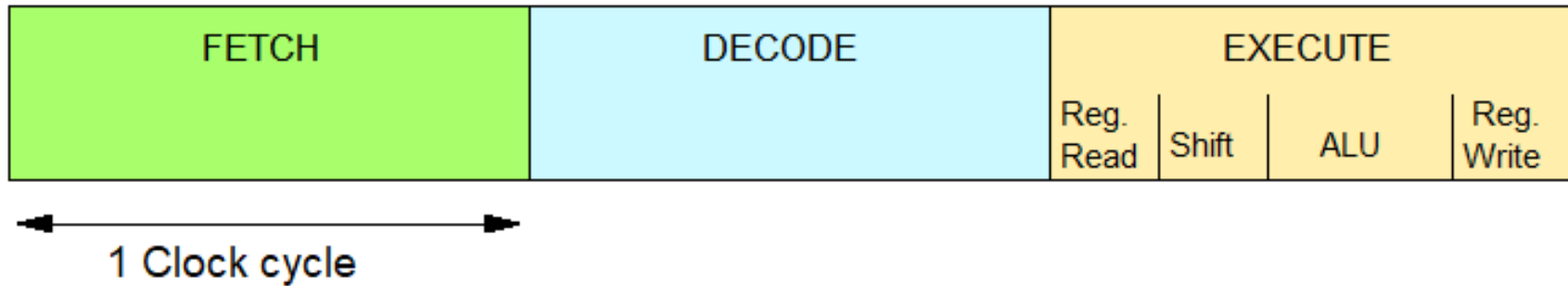
Instruction Set

ARM7TDMI

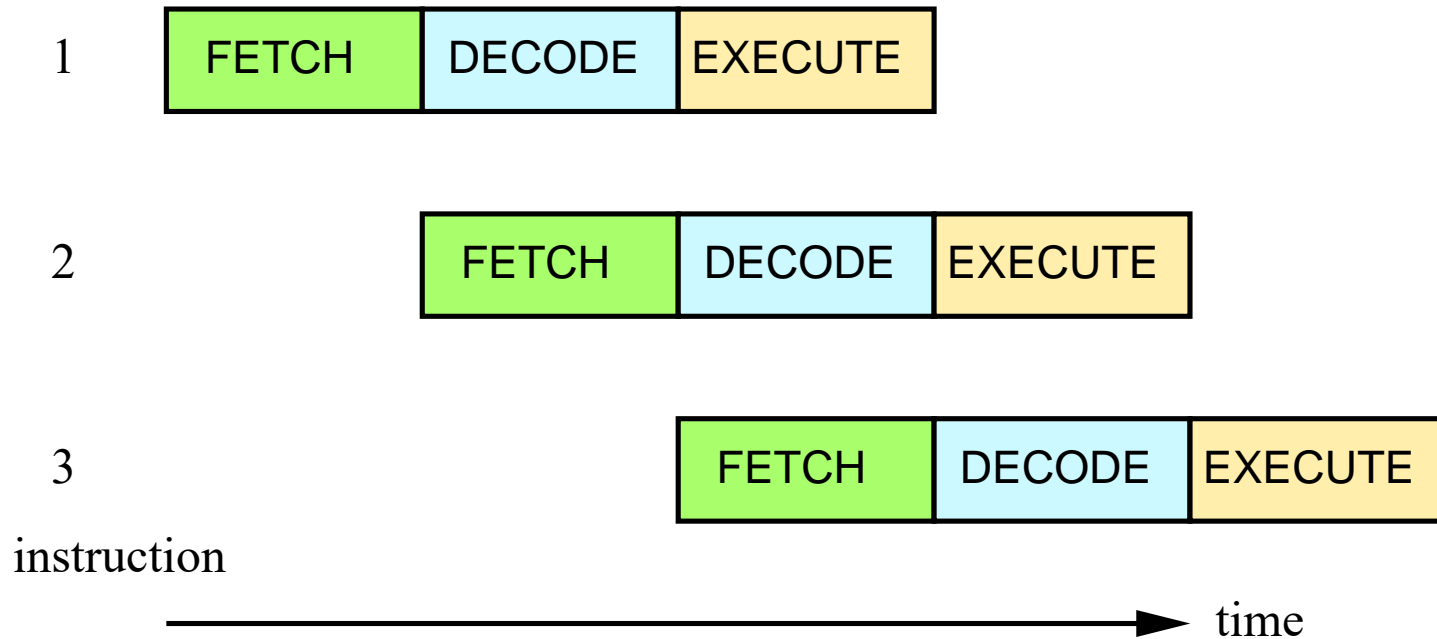
Block Diagram



ARM7TDMI Pipeline



- **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

Introduction

Architecture

■ **Programmers Model**

Instruction Set

- 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

- **The ARM has seven 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
 - **SVC** : (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

- **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)

Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr

Banked out Registers

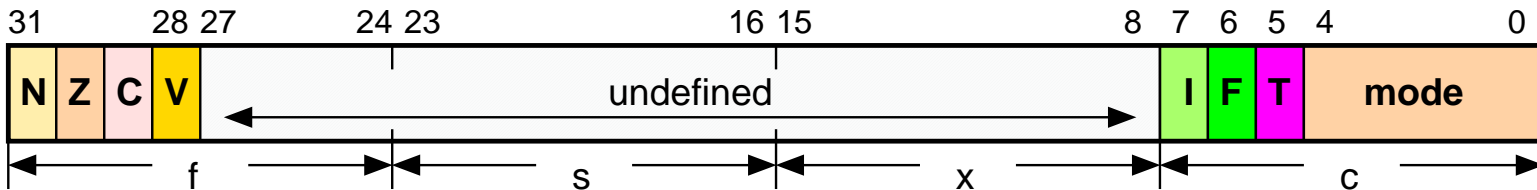
User, SYS	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

■ Special function registers:

- **PC** (R15): Program Counter. Any instruction with PC as its destination register is a program branch
- **LR** (R14): Link Register. Saves a copy of PC when executing the BL instruction (subroutine call) or when jumping to an exception or interrupt routine
 - It is copied back to PC on the return from those routines
- **SP** (R13): Stack Pointer. There is **no stack** in the ARM architecture. Even so, R13 is usually reserved as a pointer for the program-managed stack
- **CPSR** : Current Program Status Register. Holds the visible status register
- **SPSR** : Saved Program Status Register. Holds a copy of the previous status register while executing exception or interrupt routines
 - It is copied back to CPSR on the return from the exception or interrupt
 - No SPSR available in User or System modes

User, SYS	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

Note: System mode uses the User mode register set



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

Mode bits

10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

Never change T directly (use BX instead)

Changing T in CPSR will lead to unexpected behavior due to pipelining

Tip: Don't change undefined bits.

This allows for code compatibility with newer ARM processors

- **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] and bits [1:0] are zero
 - Due to pipelining, the PC points 8 bytes ahead of the current instruction, or 12 bytes ahead if current instruction includes a register-specified shift

- **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] and bit [0] is zero

Introduction


Architecture

Programmers Model

- **Instruction Set (for ARM state)**

- 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.

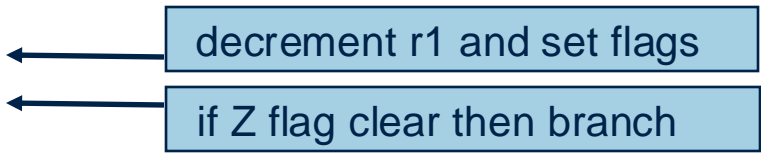
```
CMP    r3, #0
BEQ    skip
ADD    r0, r1, r2
skip
```



```
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” (comparisons always set the flags).

```
loop
...
SUBS  r1, r1, #1
BNE  loop
```



- The 15 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	

C source code

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

ARM instructions

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

■ 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
```

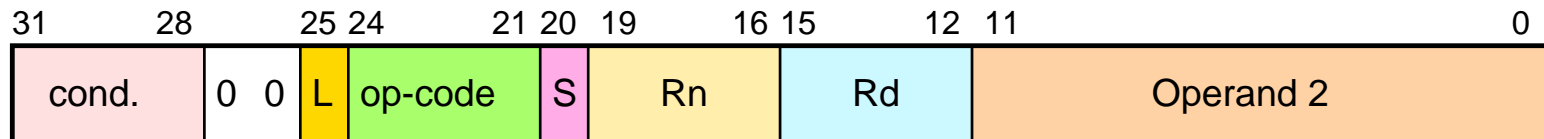
■ Use conditional compare instructions

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

■ 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.



L, Literal: 0: Operand 2 from register, 1: Operand 2 immediate

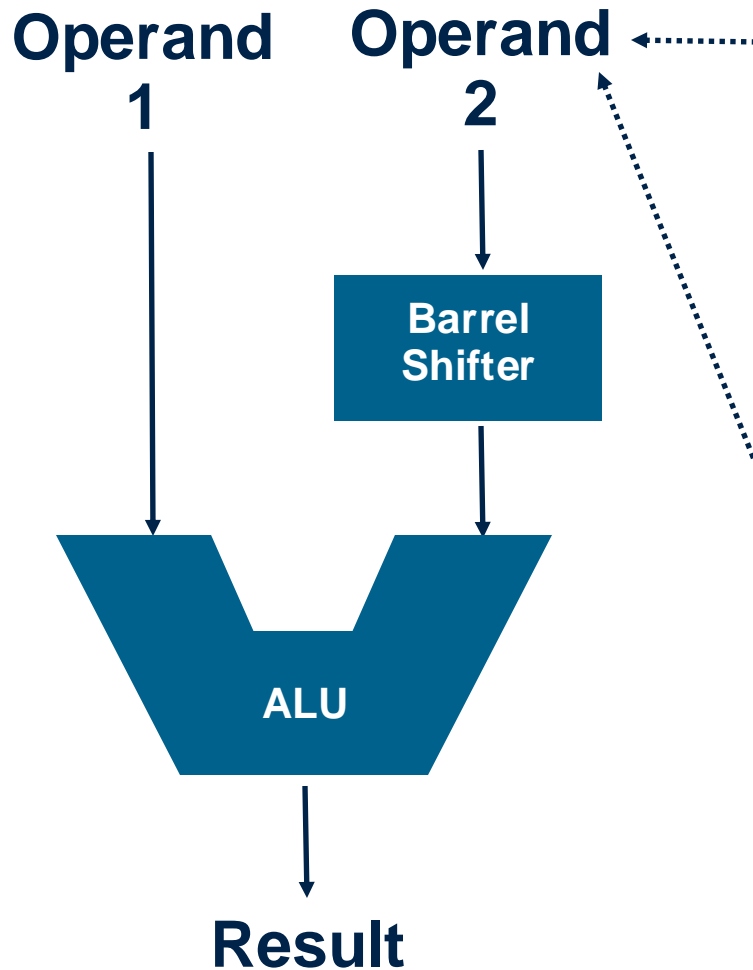
■ Syntax:

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

- {S} means that the Status register is going to be updated
- Comparisons always update the status register. Rd is not specified
- Data movement does not specify Rn

■ Second operand is sent to the ALU via barrel shifter.

Using the Barrel Shifter: The Second Operand



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 a power of 2

Example: `ADD R1, R2, R3, LSL #2`

$(R2 + R3 * 4) \rightarrow R1$

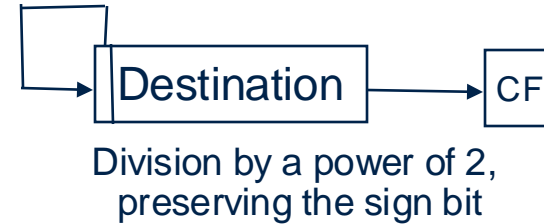
Immediate value

- 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

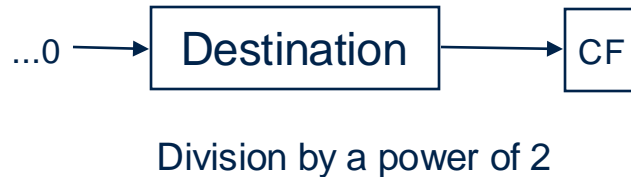
LSL : Logical Left Shift



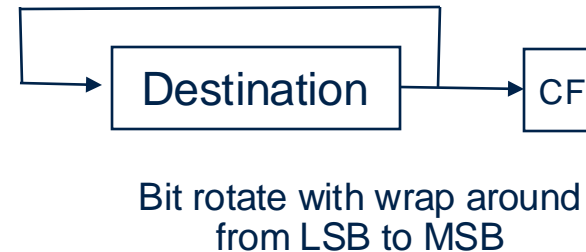
ASR: Arithmetic Right Shift



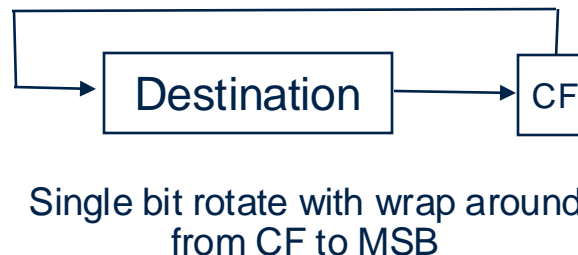
LSR : Logical Shift Right



ROR: Rotate Right



RRX: Rotate Right Extended



- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
 - `LDR rd, =const` (notice the “=” sign)
- 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` \Rightarrow `MOV r0,#0xFF`
 - `LDR r0,=0x55555555` \Rightarrow `LDR r0,[PC, #Imm12]`
 - ...
 - ...
 - `DCD 0x55555555`
- This is the recommended way of loading constants into a register

- **Flags are changed only if the S bit of the op-code is set:**
Mnemonics ending with “s”, like “movs”, and comparisons: **cmp**, **cmn**, **tst**, **teq**
- **N and Z have the expected meaning for all instructions**
 - **N**: bit 31 (sign) of the result
 - **Z**: set if result is zero
- **Logical instructions (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN)**
 - **V**: unchanged
 - **C**: from **barrel shifter** if **shift $\neq 0$** . Unchanged otherwise
- **Arithmetic instructions (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN)**
 - **V**: Signed overflow from **ALU**
 - **C**: Carry (bit 32 of result) from **ALU**
- **When PC is the destination register (exception return)**
 - CPSR is copied from **SPSR**. This includes all the flags.
 - No change in **user** or **system** modes

Example: **SUBS** **PC,LR,#4** @ return from IRQ

Operations are:

ADD	$\text{operand1} + \text{operand2}$
ADC	$\text{operand1} + \text{operand2} + \text{carry}$
SUB	$\text{operand1} - \text{operand2}$
SBC	$\text{operand1} - \text{operand2} + \text{carry} - 1$
RSB	$\text{operand2} - \text{operand1}$
RSC	$\text{operand2} - \text{operand1} + \text{carry} - 1$

Syntax:

$\langle \text{Operation} \rangle \{ \langle \text{cond} \rangle \} \{ S \} \text{ Rd, Rn, Operand2}$

Examples

```
ADD r0, r1, r2
SUBGT r3, r3, #1
RSBLES r4, r5, #5
```

The only effect of the comparisons is to **UPDATE THE CONDITION FLAGS**. Thus no need to set S bit.

Operations are:

CMP operand1 - operand2, but result not written
CMN operand1 + operand2, but result not written
TST operand1 AND operand2, but result not written
TEQ operand1 EOR operand2, but result not written

Syntax:

<Operation>{<cond>} Rn, Operand2

Examples:

CMP r0, r1
TSTEQ r2, #5

Operations are:

AND operand1 AND operand2
EOR operand1 EOR operand2
ORR operand1 OR operand2
BIC operand1 AND NOT operand2 [ie bit clear]

Syntax:

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

Examples:

AND r0, r1, r2
BICEQ r2, r3, #7
EORS r1,r3,r0

Operations are:

MOV operand2
MVN NOT operand2

Note that these make no use of operand1.

Syntax:

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

Examples:

MOV r0, r1
MOVS r2, #10
MVNEQ r1,#0

■ Syntax:

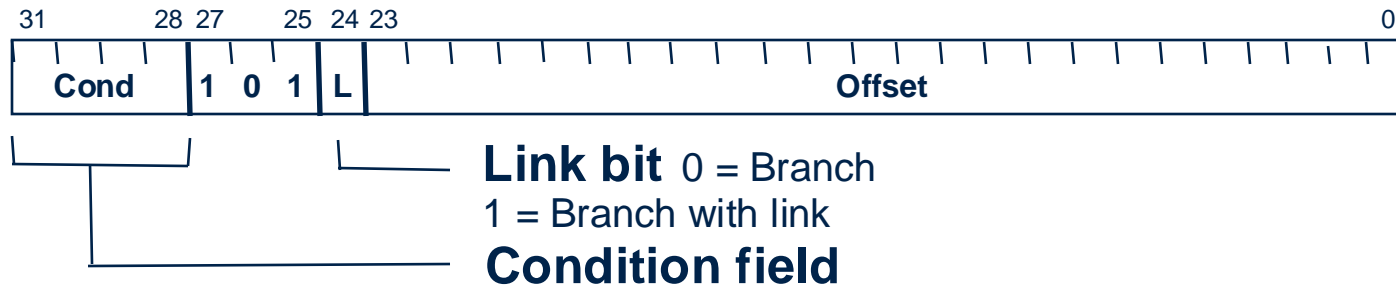
- | | |
|---|--|
| ■ MUL {<cond>}{S} Rd, Rm, Rs | $Rd = Rm * Rs$ |
| ■ MLA {<cond>}{S} Rd, Rm, Rs, Rn | $Rd = (Rm * Rs) + Rn$ |
| ■ [U S]MULL {<cond>}{S} RdLo, RdHi, Rm, Rs | $RdHi, RdLo := Rm * Rs$ |
| ■ [U S]MLAL {<cond>}{S} RdLo, RdHi, Rm, Rs | $RdHi, RdLo := (Rm * Rs) + RdHi, RdLo$ |

■ 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

- **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 or absolute address branches?
solution: `LDR PC,...`

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

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

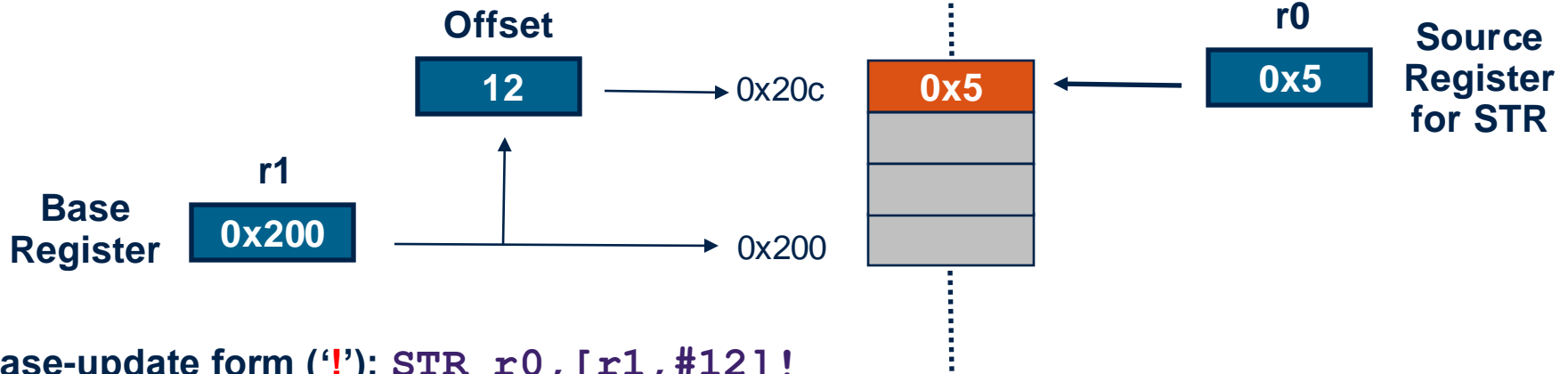
- **Syntax:**

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

e.g. **LDREQB**

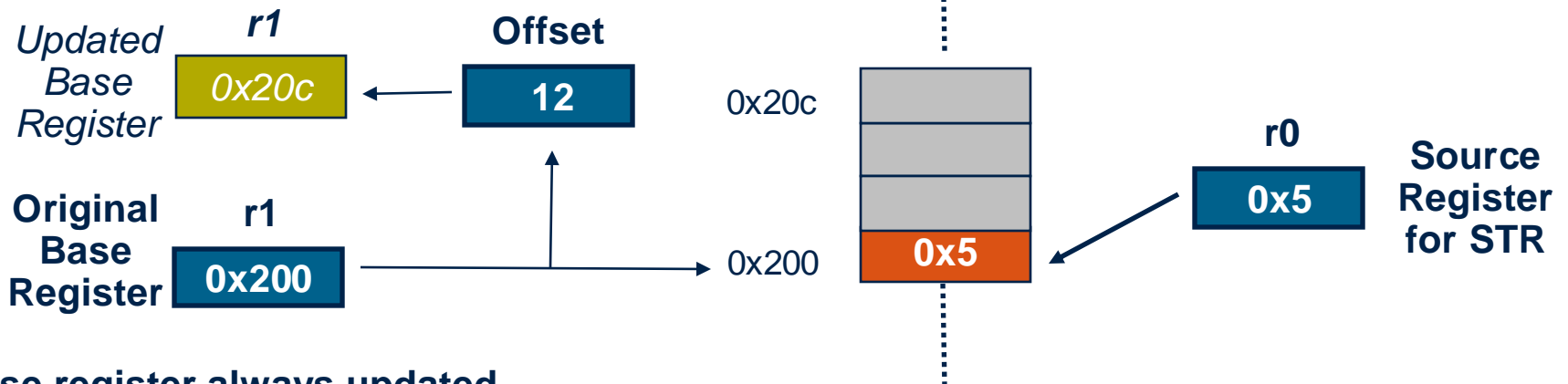
- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
 - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).
`LDR r0, [r1, #8]`
 - A register, optionally shifted by an immediate value
`LDR r0, [r1, r2]`
`LDR r0, [r1, r2, 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]`



Base-update form ('!'): `STR r0, [r1, #12] !`

■ Post-indexed: `STR r0, [r1], #12`



Base register always updated

- Load/Store Multiple 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**

Base-update possible:
LDM r10!, {r0-r6}

