

# ARM7, Cortex M3 Programming

## COE718: Embedded Systems Design

<http://www.ecb.torontomu.ca/~courses/coe718/>

**Dr. Gul N. Khan**

<http://www.ecb.torontomu.ca/~gnkhan>

*Electrical, Computer and Biomedical Engineering*

**Toronto Metropolitan University**

---

## Overview

- ARM Cortex-M\* Programming
- Data Processing & Load/Store Instructions
- Control Instruction and Conditional Execution - IT Instructions
- Functional Call and Return
- Temporary Variables

Text by Lewis: Part of Chapters 6, 7 and Data Sheets

Text by M. Wolf: part of Chapters/Sections 2.1, 2.2 and 2.3

# ARM Registers and Programming Model

**ARM  
Mode:**  
**15 general  
purpose  
registers**

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13: Stack Pointer (SP)
R14: Link Register (LR)
R15: Program Counter (PC)

**Thumb  
Mode:**

**8 general  
purpose  
registers**

**7 “high”  
registers**

**r8-R12 only  
accessible  
with MOV,  
ADD, or  
CMP**

# ARM Data Processing Instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	$Rd := Rn \text{ AND } Op2$
0001	EOR	Logical bit-wise exclusive OR	$Rd := Rn \text{ EOR } Op2$
0010	SUB	Subtract	$Rd := Rn - Op2$
0011	RSB	Reverse subtract	$Rd := Op2 - Rn$
0100	ADD	Add	$Rd := Rn + Op2$
0101	ADC	Add with carry	$Rd := Rn + Op2 + C$
0110	SBC	Subtract with carry	$Rd := Rn - Op2 + C - 1$
0111	RSC	Reverse subtract with carry	$Rd := Op2 - Rn + C - 1$
1000	TST	Test	Scc on $Rn \text{ AND } Op2$
1001	TEQ	Test equivalence	Scc on $Rn \text{ EOR } Op2$
1010	CMP	Compare	Scc on $Rn - Op2$
1011	CMN	Compare negated	Scc on $Rn + Op2$
1100	ORR	Logical bit-wise OR	$Rd := Rn \text{ OR } Op2$
1101	MOV	Move	$Rd := Op2$
1110	BIC	Bit clear	$Rd := Rn \text{ AND NOT } Op2$
1111	MVN	Move negated	$Rd := \text{NOT } Op2$

# Bitwise Instructions

<i>Bitwise Instructions</i>	<i>Operation</i>	<i>{S}</i>	<i>&lt;op&gt;</i>	<i>Notes</i>
AND $R_d, R_n, <op>$	$R_d \leftarrow R_n \& <op>$	NZC	imm. const. -or- reg{,<shift>}	
ORR $R_d, R_n, <op>$	$R_d \leftarrow R_n   <op>$	NZC		
EOR $R_d, R_n, <op>$	$R_d \leftarrow R_n \wedge <op>$	NZC		
BIC $R_d, R_n, <op>$	$R_d \leftarrow R_n \& \sim <op>$	NZC		
ORN $R_d, R_n, <op>$	$R_d \leftarrow R_n   \sim <op>$	NZC		
MVN $R_d, R_n$	$R_d \leftarrow \sim R_n$	NZC		

# Shift Instructions

<i>&lt;shift&gt;</i>	<i>Meaning</i>	<i>Notes</i>
LSL #n	Logical shift left by n bits	Zero fills; $0 \leq n \leq 31$
LSR #n	Logical shift right by n bits	Zero fills; $1 \leq n \leq 32$
ASR #n	Arithmetic shift right by n bits	Sign extends; $1 \leq n \leq 32$
ROR #n	Rotate right by n bits	$1 \leq n \leq 32$
RRX	Rotate right w/C by 1 bit	

# Load/Store Instructions

<i>Load/Store Memory</i>	<i>Operation</i>	<i>Notes</i>
LDR $R_d, <mem>$	$R_d \leftarrow mem_{32}[address]$	
LDRB $R_d, <mem>$	$R_d \leftarrow mem_8[address]$	Zero fills
LDRH $R_d, <mem>$	$R_d \leftarrow mem_{16}[address]$	Zero fills
LDRSB $R_d, <mem>$	$R_d \leftarrow mem_8[address]$	Sign extends
LDRSH $R_d, <mem>$	$R_d \leftarrow mem_{16}[address]$	Sign extends
LDRD $R_t, R_{t2}, <mem>$	$R_{t2}.R_t \leftarrow mem_{64}[address]$	Addr. Offset must be imm.

<i>Load/Store Memory</i>	<i>Operation</i>	<i>Notes</i>
STR $R_d, <mem>$	$R_d \rightarrow mem_{32}[address]$	
STRB $R_d, <mem>$	$R_d \rightarrow mem_8[address]$	
STRH $R_d, <mem>$	$R_d \rightarrow mem_{16}[address]$	
STRD $R_t, R_{t2}, <mem>$	$R_{t2}.R_t \rightarrow mem_{64}[address]$	Addr. Offset must be imm.

# Loading Constants

**MOV**  $r_d, \textit{constant}$

- Works for 0 - 255 and “some” others

**MVN**  $r_d, \textit{constant}; r_d \leftarrow \sim \textit{constant}$

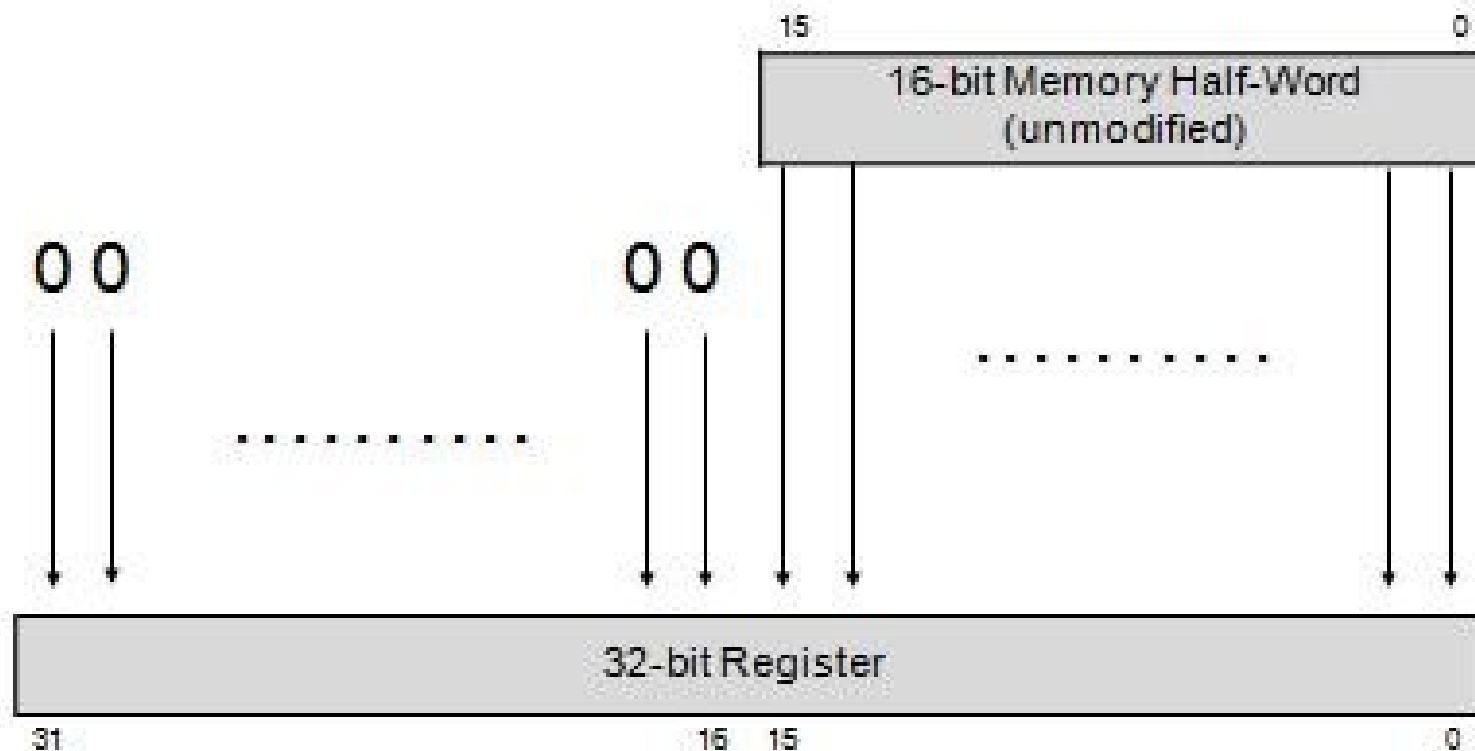
- Effectively doubles the # of constants
- Assembler converts MOV w/neg. const to MVN

**LDR**  $r_d, =\textit{constant}$

- An assembler pseudo-op, not an instruction
- Converted to MOV or MVN if possible
- Else converts to LDR  $r_d, [pc, \#imm]$

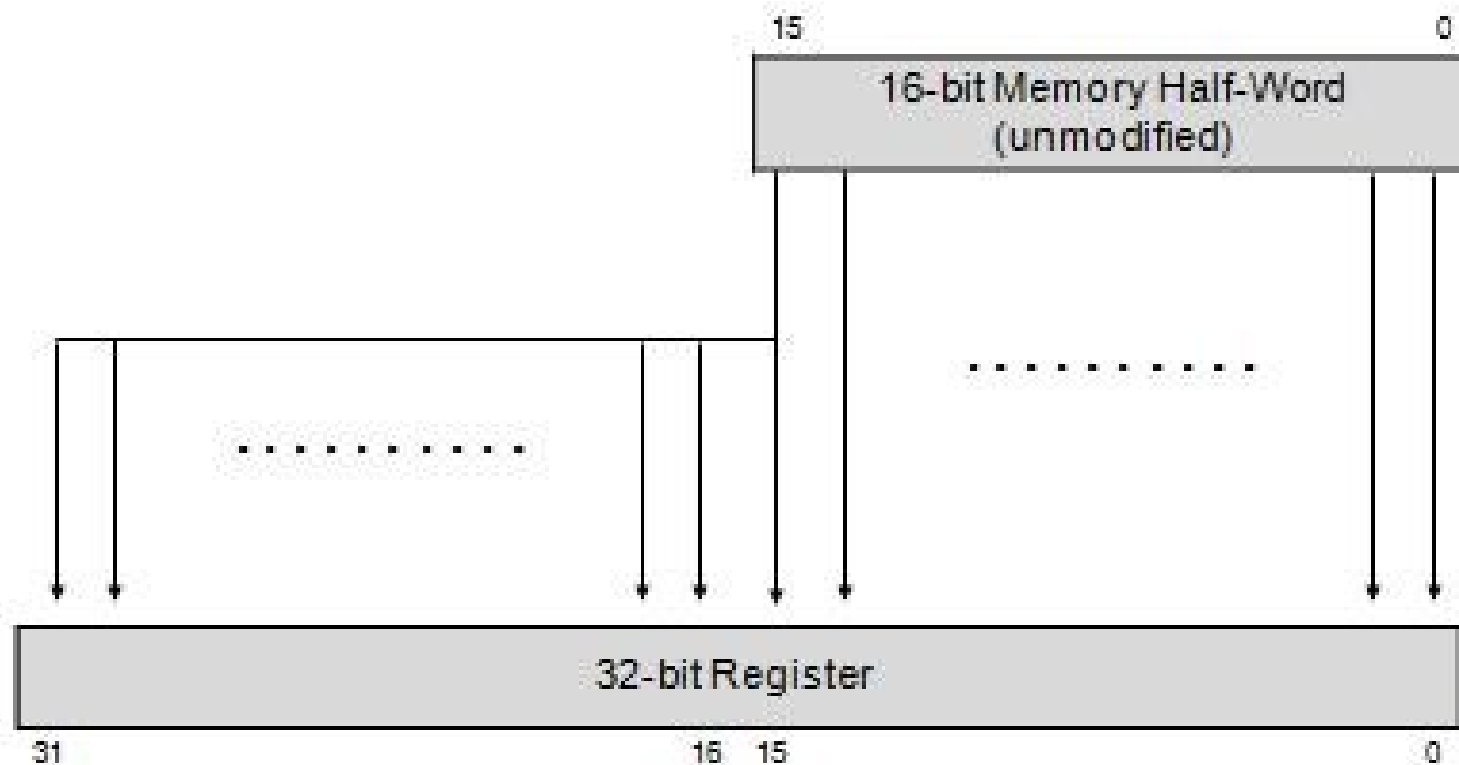
# LDRH

## (Load Halfword)



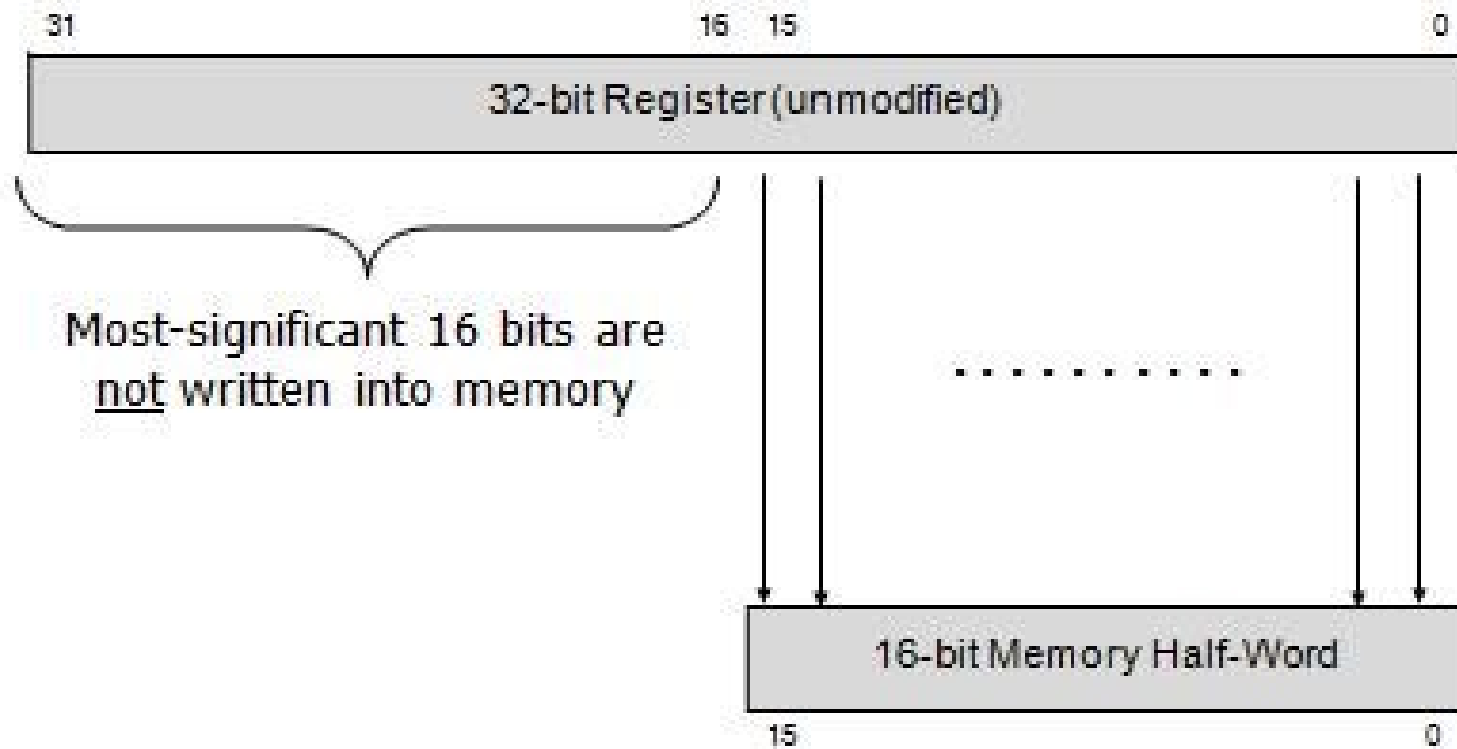
# LDRSH

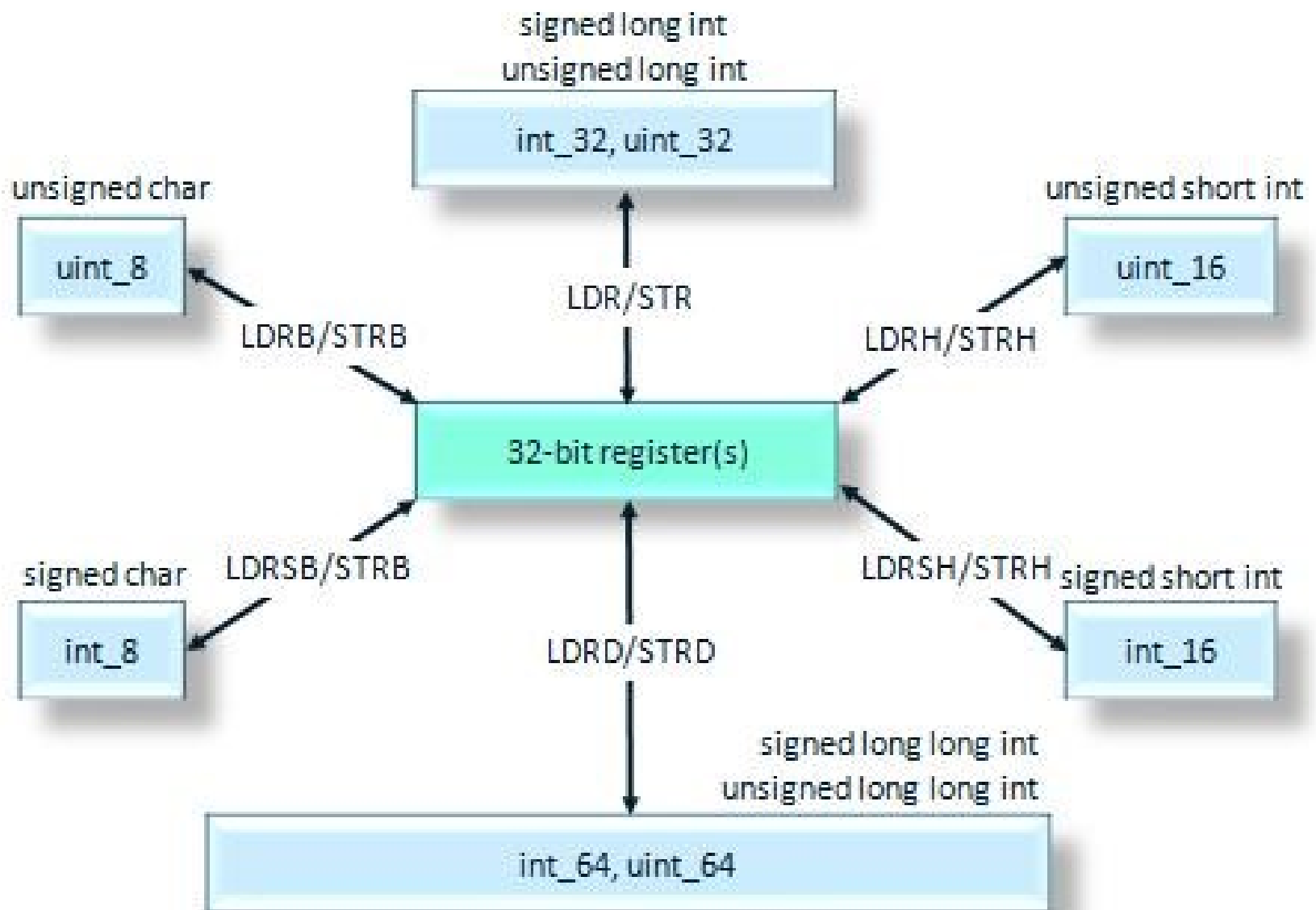
(Load Signed Halfword)



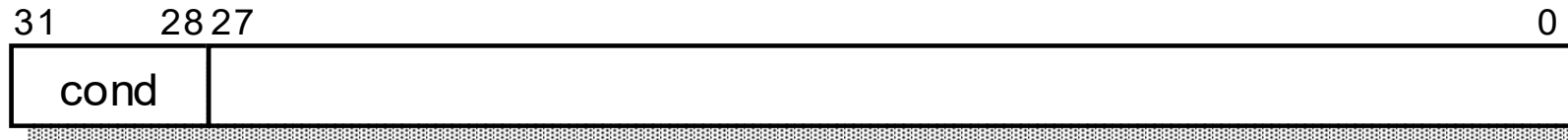


# STRH (Store Halfword)





# The ARM Condition Code Field



## ARM condition codes

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

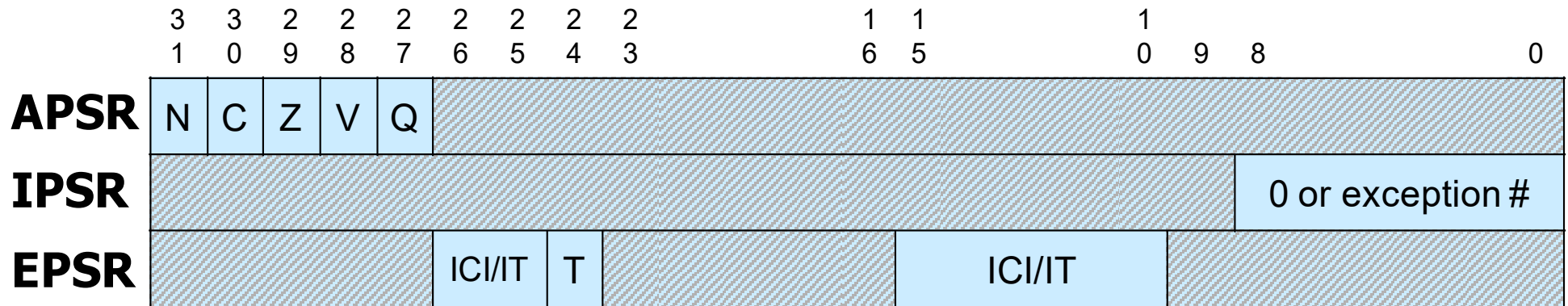
# Branch Instructions

<i>Branch Instructions</i>	<i>Operation</i>	<i>{S}</i>	<i>Notes</i>
B{c} label	$PC \leftarrow PC + \text{imm}$	n/a	“c” is an <i>optional</i> condition code
BL label	$PC \leftarrow PC + \text{imm};$ $LR \leftarrow \text{rtn adr}$	n/a	Subroutine call
BX reg	$PC \leftarrow \text{reg}$	n/a	
CBZ $R_n$ ,label	If $R_n=0$ , $PC \leftarrow PC + \text{imm}$	n/a	Cannot append condition code to CBZ
CBNZ $R_n$ ,label	If $R_n \neq 0$ , $PC \leftarrow PC + \text{imm}$	n/a	Cannot append condition code to CBNZ
ITc <sub>1</sub> c <sub>2</sub> c <sub>3</sub> cond	Each $c_i$ is one of T, E, or <i>empty</i>	n/a	Controls 1-4 instructions in “IT block”

# Branch Conditions

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

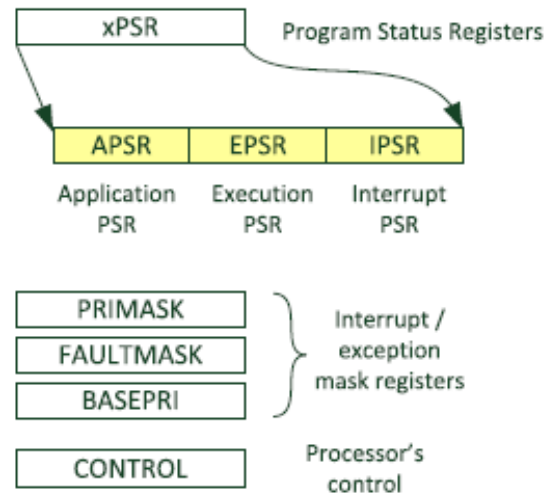
# Status Registers (xPSR)



Bits	Name	Description
31	N	Negative (bit 31 of result is 1)
30	C	Unsigned Carry
29	Z	Zero or Equal
28	V	Signed Overflow

**Most important for application programming**

## Special Registers



	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/T	T		GE*	ICI/T		Exception Number				

\*GE is available in ARMv7E-M processors such as the Cortex-M4. It is not available in the Cortex-M3 processor.

Bit	Description
N	Negative flag
Z	Zero flag
C	Carry (or NOT borrow) flag
V	Overflow flag
Q	Sticky saturation flag (not available in ARMv6-M)
GE[3:0]	Greater-Than or Equal flags for each byte lane (ARMv7E-M only; not available in ARMv6-M or Cortex <sup>®</sup> -M3).
ICI/T	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit for conditional execution (not available in ARMv6-M).
T	Thumb state, always 1; trying to clear this bit will cause a fault exception.
Exception Number	Indicates which exception the processor is handling.

# PSR: Program Status Register

Divided into 3-bit fields

- Application Program Status Register (APSR)
- Interrupt Program Status Register (IPSR)
- Execution Program Status Register (EPSR)

Q-bit is the sticky saturation bit and supports two rarely used instructions (SSAT and USAT)

*SSAT{cond} Rd, #sat, Rm{, shift}*

- IPSR holds the exception number of exception processing.
- ICI/IT bits hold the state information for IT block instructions or instructions that are suspended during interrupt processing.
- T bit = 1, indicates Thumb instructions.



# SSAT: Saturate Instruction

- Consider two numbers 0xFFFF FFFE and 0x0000 0002. A 32-bit mathematical addition would result in 0x1 0000 0001 which contain 9 hex digits or 33 binary bits. If the same arithmetic is done in a 32-bit processor, ideally the carry flag will be set and the result in the register will be 0x0000 0001.
- If the operation was done by any comparison instruction this would not cause any harm but during any addition operation this may lead to un-predictable results if the code is not designed to handle such operations. Saturate arithmetic says that when the result crosses the extreme limit the value should be maintained at the respective maximum/minimum (in our case result will be maintained at 0xFFFF FFFF which is the largest 32-bit number).
- Saturate instructions are very useful in implementing certain DSP algorithms like audio processing where we have a cutoff high in the amplitude. For instance, the highest amplitude is expressed by a 32-bit value and if my audio filter gives an output more than this I need not to programmatically monitor the result. Rather the value automatically saturates to the max limit.
- Also a new flag field called 'Q' has been added to the ARM processor to show us if there had been any such saturation taken place or the natural result itself was the maximum.

# SSAT or USAT Instructions

$op\{cond\} Rd, \#n, Rm \{, shift \#s\}$

$op = SSAT$  Saturates a signed value to a signed range.

$USAT$  Saturates a signed value to an unsigned range.

**Cond** condition code

**Rd** Specifies the destination register.

**n** Specifies the bit position to saturate to:

$n$  ranges from 1 to 32 for SSAT

$n$  ranges from 0 to 31 for USAT.

**Rm** Register containing the value to saturate.

$shift \#s$  optional shift applied to  $Rm$  before saturating.

These instructions saturate to a signed or unsigned  $n$ -bit value.

SSAT instruction applies the specified shift, then saturates to the signed range  $-2^{n-1} \leq x \leq 2^{n-1}-1$ .

The USAT instruction applies the specified shift, then saturates to the unsigned range  $0 \leq x \leq 2^n-1$ .

# SSAT or USAT Instructions

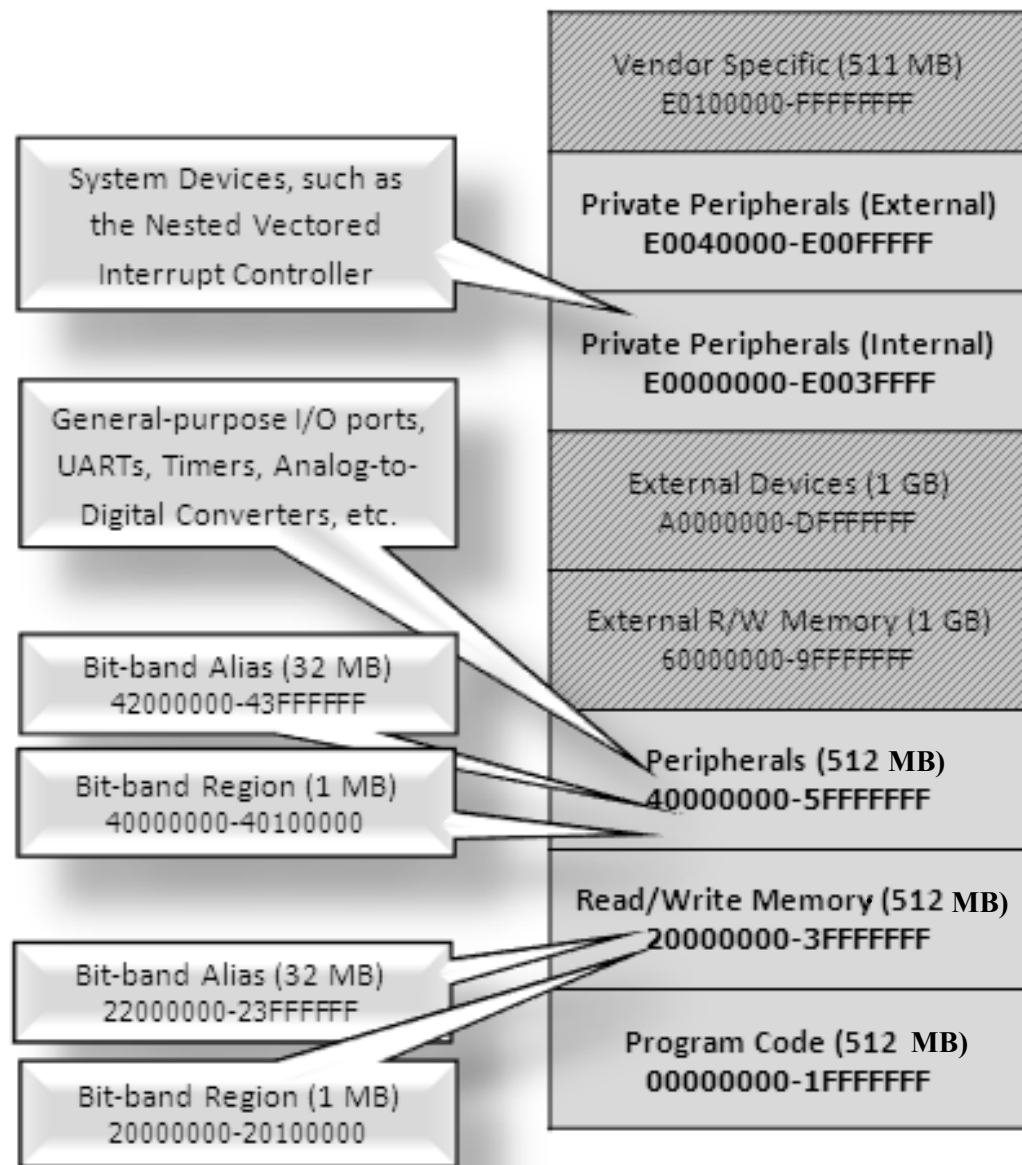
If the returned result is different from the value to be saturated, it is called *saturation*.

If saturation occurs, the instruction sets the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged.

## Examples

```
SSAT    R7, #16, R7, LSL #4  
        ;
```

```
USATNE  R0, #7, R5    ; Conditionally saturate value in R5 as an  
                        ; unsigned 7 bit value and write it to R0.
```



# ARM Cortex-M3 Memory

# Bit Banding

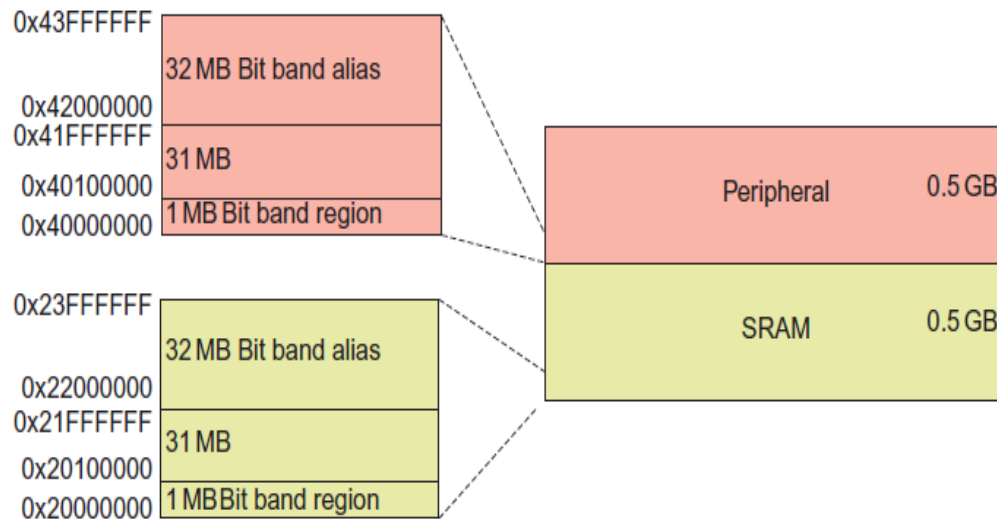
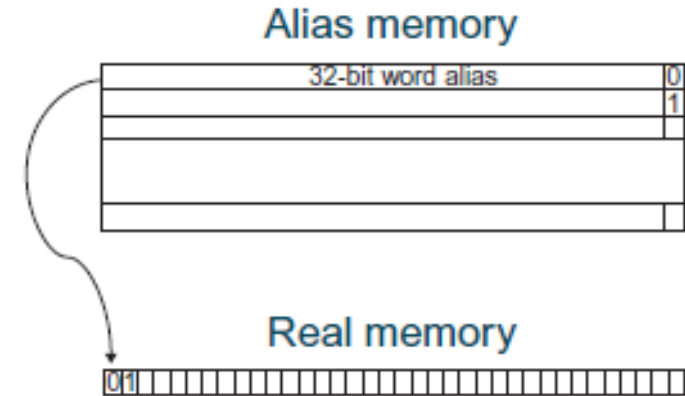
- Memory mapped I/O, 4GB memory address space organized in bytes.
- 4GB is very large for small embedded applications.
- Bit-banding happens by taking advantage of this large memory space.
- Uses two different regions of the address space to refer the same physical data in the memory.
- In primary bit-band region each address corresponds to single data byte.
- In the bit-band alias each address corresponds to 1-bit of the same data.
- It allows the access of a bit of data (read or write) by a single instruction.
- Two bit band alias regions can be used to access individual status and control bit of I/O devices or to implement a set of 1-bit Boolean flags that can be used to implement a set of mutex objects.
- Bit-band hardware does not allow interruption of read-modify write.

Bit\_band alias address = Bit\_band base + 128 × word\_offset + 4 × bit #

If bit-5 at address 20001000<sub>16</sub> is to be accessed, the bit-band alias will be  
 $22000000_{16} + (128_{10} \times 1000_{16}) + (4 \times 5) = 22080014_{16}$

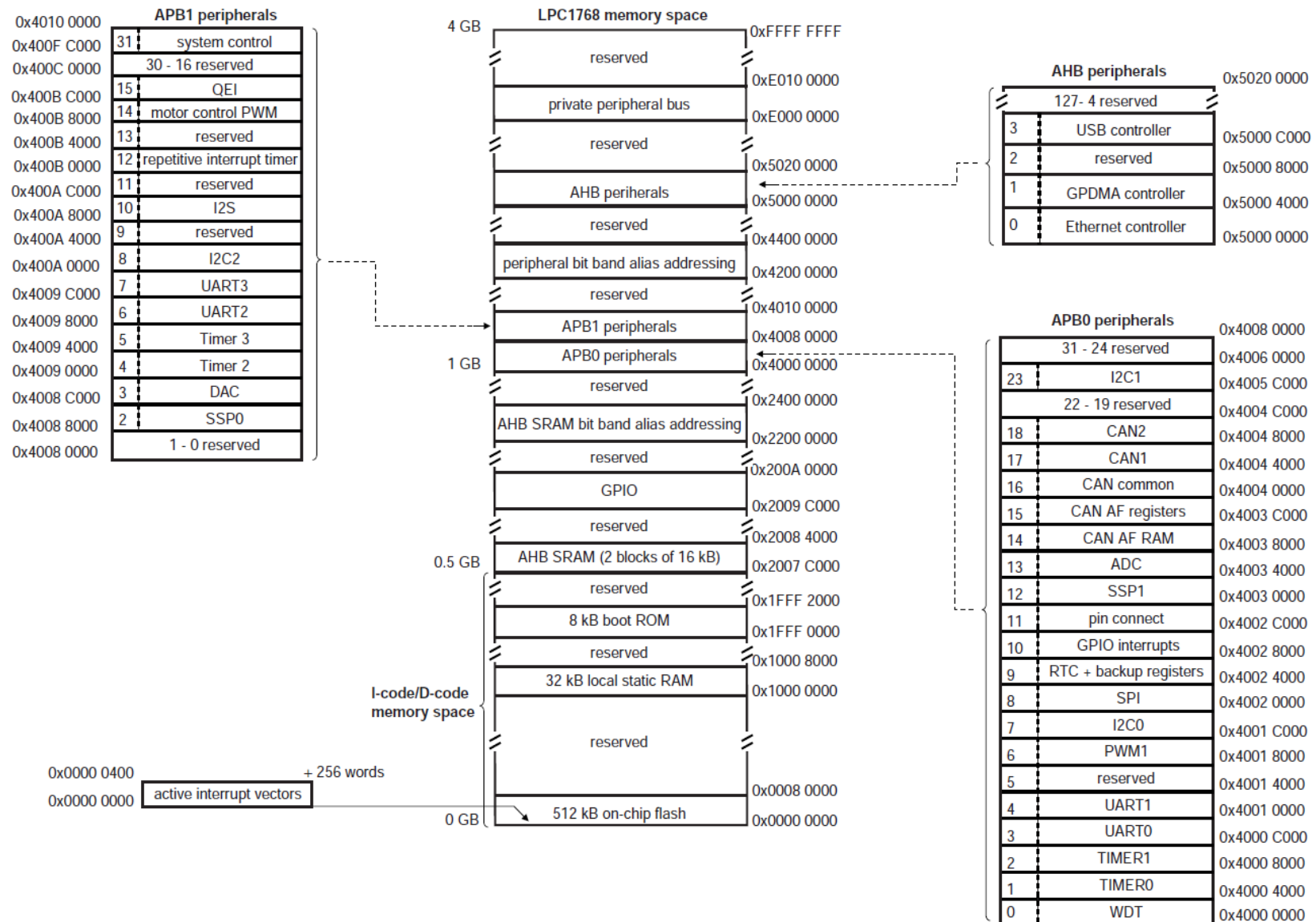
# Bit Banding

Address 0x20000000 = SRAM  
 0x40000000 = Peripheral = external RAM  
 devices, memory vendor specific, etc.



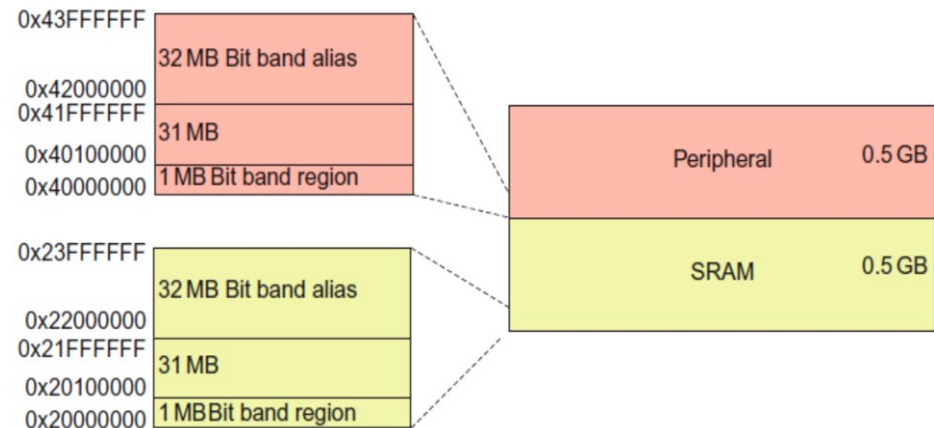
- \* One bit is addresses by its own 32-bit (word) in a separate part of memory (bit-band region)
- \* Bit-banding is for 2 predefined memory regions:
  - first 1MB of SRAM,
  - first 1MB of peripheral region
- \* To access each bit individually, we need to access a memory region referred to as the bit-band alias region.

# Bit Banding



# Bit Banding

Question: Find bit band word address for:  
SRAM address 0x2008C000, bit 3.  
Use equations (2) and (1):



## Bit Band Word Address =

Bit Band Alias Base Address + (Byte\_Offset \* 32) + (Bit Number \* 4) (1)

Byte\_Offset = Bit's Bit Band Base Address - Bit Band Base Address (2)

where: **Byte\_Offset**

Bit's Bit Band Base Address - the base address for the targeted SRAM or peripheral register  
(The Effective Address of the Port) (= real address)

**Bit Band Base Address:** for SRAM = 0x20000000, for Peripherals = 0x40000000

**Bit Band Alias Base Address:**

for SRAM = 0x22000000, for Peripherals = 0x42000000

**Bit Number:** the bit position of the targeted register (i.e. pin of the port)



# Bit Banding Example

Peripheral address 0x400ABC00, bit 8

Steps for bit banding:

1. Calculate the Word Address:

2. Define a Pointer to the Address:

```
#define BIT_ADDR= (*(
```

3. Assign a Value to the Port Bit:

```
int main(void) {
```

```
...
```

```
}
```

# Conditional Execution

ADD instruction with the EQ condition appended.

This instruction will only be executed when the zero flag in the *cpsr* is set;

ADDEQ    r0, r1, r2    ; r0 = r1 + r2 if zero flag is set

```
while (a!=b) {  
    ; Greatest Common Divisor Algorithm  
    if (a > b) a -= b; else b -= a;  
}
```

Register *r1* represent *a* and register *r2* represent *b*.

```
gcd      CMP r1, r2  
         BEQ complete  
         BLT lessthan  
         SUB r1, r1, r2  
         B gcd  
lessthan SUB r2, r2, r1  
         B gcd  
complete  
...  
This dramatically reduces  
the number of instructions
```

```
gcd CMP r1, r2  
    SUBGT r1, r1, r2  
    SUBLT r2, r2, r1  
    BNE gcd  
complete  
...
```

```
gcd CMP r1, r2  
  
    SUBGT r1, r1, r2  
    SUBLT r2, r2, r1  
    BNE gcd  
complete  
...
```

# IT (If-Then)

**IT (If-Then)** instruction makes up to four following instructions (the *IT block*) conditional. The conditions can be all the same, or some of them can be the logical inverse of the others.

**IT {*x* {*y* {*z*} } } {*cond*}**

where: *x*: specifies the condition switch for the second instruction in the IT block.

*y*: specifies condition switch for the third instruction in the IT block

*z*: specifies condition switch for the fourth instruction in the IT block

***cond***: specifies the condition for first instruction in the IT block

Condition switch for 2nd, 3rd & 4<sup>th</sup> instruction in the IT block either:

- T Then. Applies the condition *cond* to the instruction.
- E Else. Applies the inverse condition of *cond* to the instruction.

The instructions (including branches) in the IT block, except the BKPT instruction, must specify the condition in the {*cond*} part of their syntax.

# IT (If-Then) instruction

- You do not need to write IT instructions in your code.
- The assembler generates them automatically according to the conditions specified on the following instructions.
- Writing the IT instructions ensures that you consider the placing of conditional instructions, and the choice of conditions.
- When assembling to ARM code, the assembler performs the same checks, but does not generate any IT instructions.
- With the exception of CMP, CMN, and TST, the 16-bit instructions that normally affect the condition code flags, do not affect them in IT block.
- A BKPT instruction in an IT block is always executed, so it does not need a condition in the *{cond}* part of its syntax. The IT block continues from the next instruction.
- Conditional branches inside an IT block have a longer branch range than those outside the IT block.

# IT (If-Then) instruction

The following instructions are not permitted in an IT block:

- IT
- CBZ and CBNZ
- TBB and TBH
- CPS, CPSID and CPSIE
- SETEND.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC is only permitted in an IT block if it is the last instruction in the block.
- You cannot branch to any instruction in an IT block, unless when returning from an exception handler.

## Architectures

- This 16-bit Thumb instruction is available in ARMv6T2 and above.
- In ARM code, IT is a pseudo-instruction that does not generate any code.

## IT Examples

```
ITTE    NE          ; IT can be omitted
ANDNE   r0,r0,r1    ; 16-bit AND, not ANDS
ADDsNE  r2,r2,#1     ; 32-bit ADDS (16-bit ADDS dos'nt set flags in IT)
MOVEQ   r2,r3        ; 16-bit MOV
```

```
ITT     AL          ; emit 2 non-flag setting 16-bit instructions
ADDAL   r0,r0,r1    ; 16-bit ADD, not ADDS
SUBAL   r2,r2,#1     ; 16-bit SUB, not SUB
ADD     r0,r0,r1     ; expands into 32-bit ADD, and is not in IT block
```

```
ITT     EQ
MOVEQ   r0,r1
BEQ     dloop        ; branch at end of IT block is permitted
```

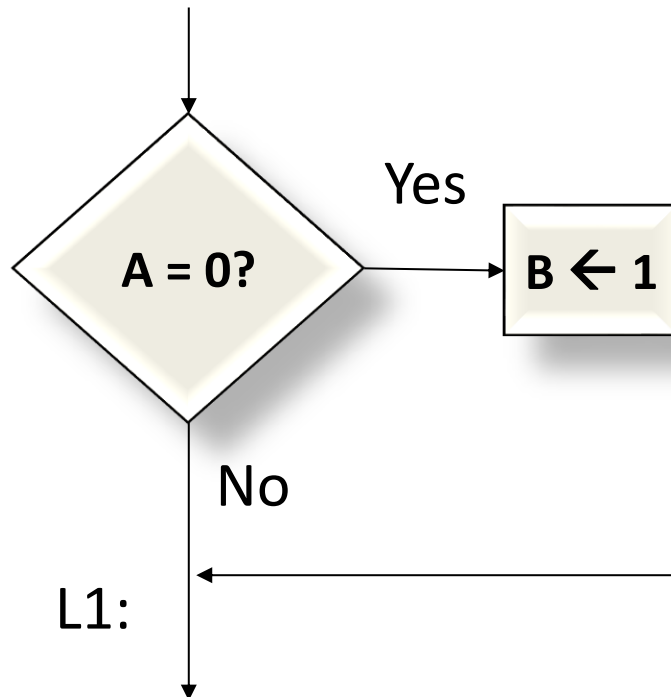
```
ITT     EQ
MOVEQ   r0,r1
BKPT    #1           ; BKPT always executes
ADDEQ   r0,r0,#1
```

## Incorrect example

```
IT     NE
ADD    r0,r0,r1; syntax error: no condition code used in IT
```

# if-then statement

if (a == 0) b = 1 ;

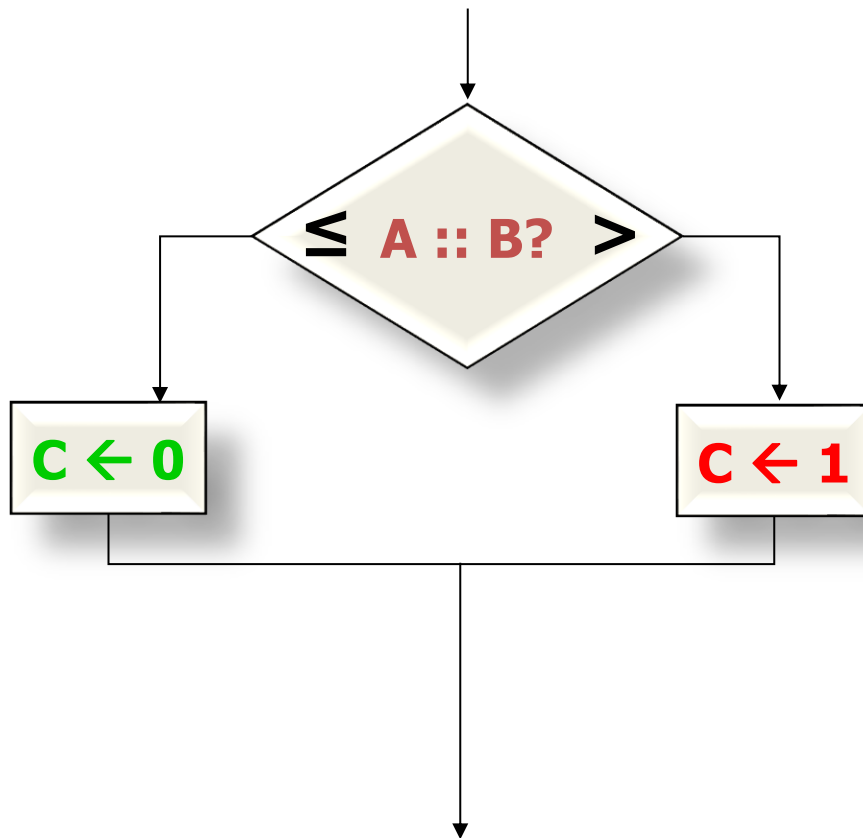


```
LDR    R0,A
CMP     R0,#0
BNE     L1
LDR     R0,#1
STR     R0,B
```

**L1:** ...

- or -

# if-then-else statement



```
LDR    R0,A
LDR    R1,B
CMP    R0,R1
BLE    L1
LDR    R0,=1
B      L2
L1:    LDR    R0,=0
L2:    STR    R0,C
...
```

- or -

```
LDR    R0,A
LDR    R1,B
CMP    R0,R1
ITE    GT
LDRGT  R0,=1
LDRLE  R0,=0
STR    R0,C
```



# An ITTE Block

```
if (R1<R2) then
    R2=R2-R1
    R2=R2/2
else
    R1=R1-R2
    R1=R1/2
```

```
CMP      R1, R2 ; If R1 < R2 (less than)
ITTEE    LT      ; then execute instruction 1 and 2
           ; (indicated by T)
           ; else execute instruction 3 and 4
           ; (indicated by E)
SUBLT.W  R2,R1   ; 1st instruction
LSRLT.W  R2,#1   ; 2nd instruction
SUBGE.W  R1,R2   ; 3rd instruction (notice the GE is
           ; opposite of LT)
LSRGE.W  R1,#1   ; 4th instruction
```

# Conditional Execution

- ARM allows non-control flow based instructions to be appended with conditional codes.
- It allows for more efficient coding and processor performance.

## Conditional Instruction Method

CMP	r2, #5	//if (a <= 5)
MOVLE	r2, #10	//a = 10;
MOVGT	r2, #1	//else a = 1;

## Non-Conditional Method

# Loops: Variable #Iterations

GCD (a, b) – Greatest Common Divisor

<b>while (a != b) {</b>		<b>LDR</b>	<b>R0,a</b>
<b>if (a &gt; b) a = a – b ;</b>		<b>LDR</b>	<b>R1,b</b>
<b>else b = b – a ;</b>	<b>top:</b>	<b>CMP</b>	<b>R0,R1</b>
<b>}</b>		<b>BEQ</b>	<b>done</b>
		<b>ITE</b>	<b>GT</b>
		<b>SUBGT</b>	<b>R0,R0,R1</b>
		<b>SUBLE</b>	<b>R1,R1,R0</b>
		<b>B</b>	<b>top</b>
	<b>done:</b>		
			<b>; R0 = R1 = GCD(a,b)</b>

# ARM Procedure Call Standard

Register Number	APCS Name	APCS Role
0	a1	argument 1 / integer result / scratch register
1	a2	argument 2 / scratch register
2	a3	argument 3 / scratch register
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4	register variable
8	v5	register variable
9	sb/v6	static base / register variable
10	sl/v7	stack limit / stack chunk handle / reg. variable
11	fp	frame pointer
12	ip	scratch register / new-sb in inter-link-unit calls
13	sp	lower end of current stack frame
14	lr	link address / scratch register
15	pc	program counter

# Function Call and Return

## Function Call: “BL function”

- Loads program counter (pc) with entry point address of function.
- Saves return address in the link register.

## Function Return: “BX lr”

- copies link register back into program counter.

```
void enable(void) ;
```

```
...
```

```
enable() ;
```

```
...
```



*Compiler*

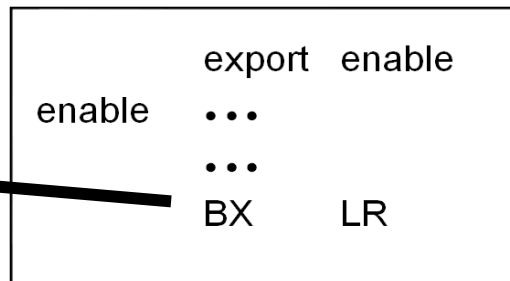
```
...
```

```
BL
```

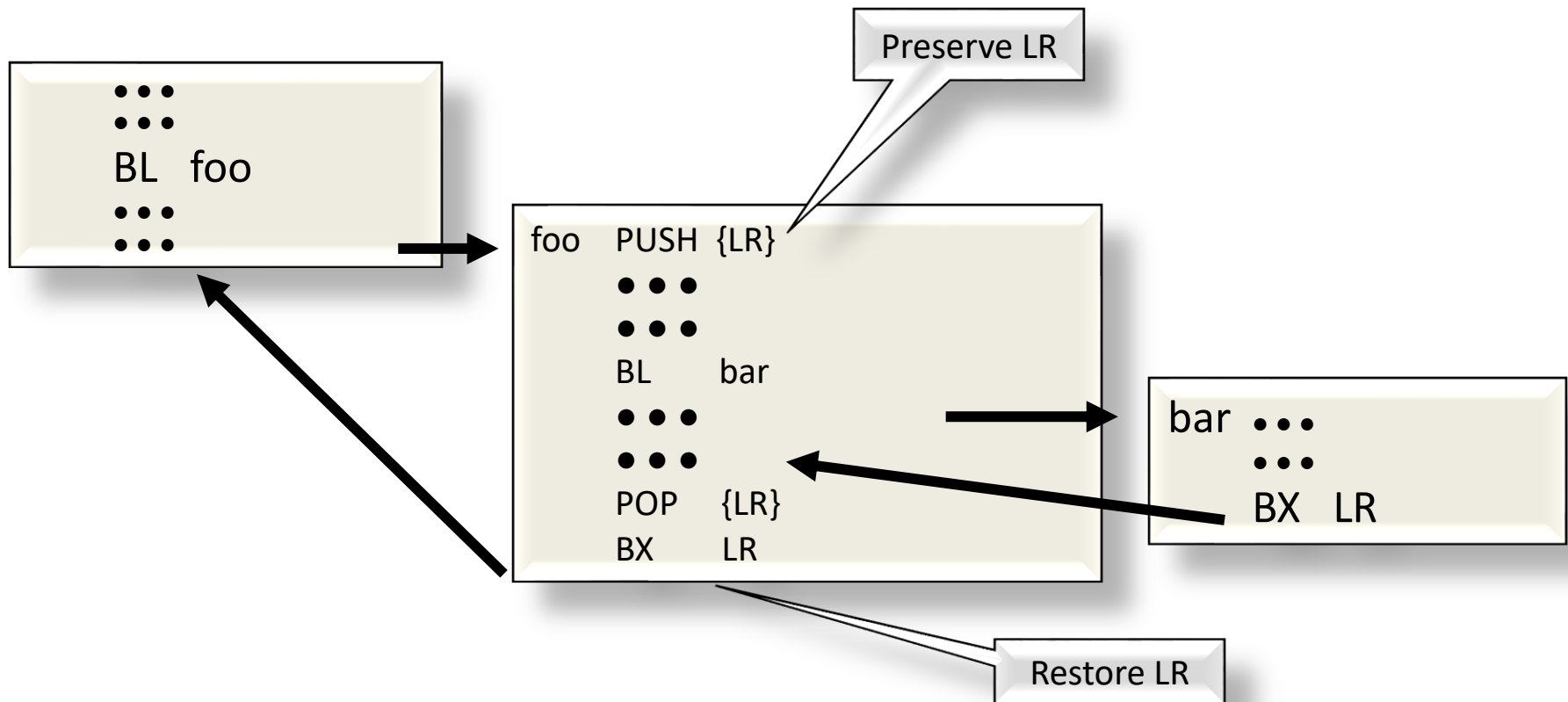
enable



```
...
```



# Function Call and Return



# Function Call and Return

```
int32_t random(void) ;  
...  
numb = random() ;  
...
```

 *Compiler*

...

BL random

STR R0,numb

...

export random  
random ...  
...  
MOV R0, ...  
BX LR

# Temporary Variables

r0 - r3 (those not used for parameters)

Must preserve and restore around any call

r4 – r8 (must always preserve and restore)

## Temporaries in Registers

<p>func1 ...</p> <p>...</p> <p>No function calls; OK to use r0 – r3</p> <p>...</p> <p>BX lr</p>	<p>func2 <b>PUSH {r4,..,r8}</b></p> <p>...</p> <p>...</p> <p>; registers r4 – r8 may be in use ; by the function that called this ; function, so their values must ; be preserved if these registers ; are used here.</p> <p>...</p> <p>...</p> <p><b>POP {r4,..,r8}</b></p> <p>BX lr</p>	<p>func3 <b>PUSH {lr,..}</b></p> <p>...</p> <p>; Since functions are not required ; to preserve r0 – r3, then if used ; here, you must preserve/restore ; their values wherever this function ; calls other functions.</p> <p>...</p> <p><b>PUSH {r0,..,r3}</b></p> <p><b>BL func4</b></p> <p>POP {r0,..,r3}</p> <p>...</p> <p><b>POP {lr,..}</b></p> <p>BX lr</p>
---	---	--



# Use of special registers: Example

```
proc_example( )  ----- ;  
..... ; other code
```

```
Void proc_example( ) {  
    int a = b + 1;  
}
```

## The assembly:

proc\_example ; LR = PC i.e. MOV R14, R15 ---- to get to this subroutine,  
we have a return address in LR

PUSH {R1} ; R13 = R13 - 4, Memory[R13] = R1

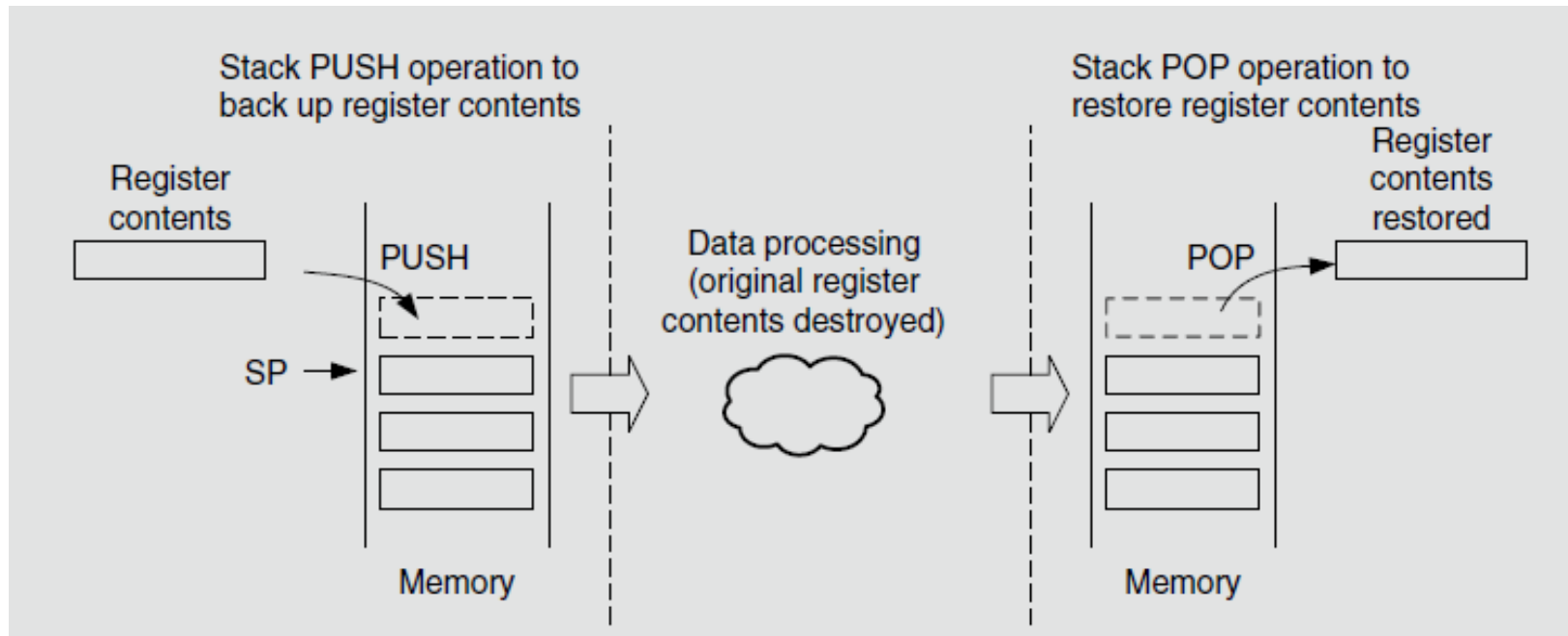
ADD R3, R1, #1

.....

POP {R1} ; R1 = Memory[R13] and R13 = R13 + 4;

BX R14 (i.e., link register)

# Use of special registers: Example



What's happening concurrently:

IF | ID | EXE

fetch = fetch instructions,  $PC = PC + 4$

To access PC, use the MOV instruction

`0x1000 MOV R0, PC ;`

# Temporary Variables

```
void Exchange(int *pItem1, int *pItem2)
{
    int temp1 = *pItem1 ;
    int temp2 = *pItem2 ;
    *pItem1 = temp2 ;
    *pItem2 = temp1 ;
}
```

EXPORT Exchange

    ; r0 = pItem1

    ; r1 = pItem2

```
Exchange    LDR    r2,[r0]    ; r2 = temp1
            LDR    r3,[r1]    ; r3 = temp2
            STR    r3,[r0]
            STR    r2,[r1]
            BX     lr
```