



고려대학교
KOREA UNIVERSITY

COSE321 Computer Systems Design

Lecture 8. Thumb2

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

Thumb, Thumb2

- **Thumb**

- 16-bit instruction set for improved code density
- Released in 1994 since ARM7TDMI

- **Thumb2**

- Introduced in 2003 since ARM1156
- Major enhancement to Thumb
- Extend 16-bit instruction set (**Thumb**) with additional 32-bit instructions; Thus, a variable-length instruction set
- Most Thumb instructions are **unconditional**
 - **Most ARM instructions are conditional**
 - **Thumb2 introduces a conditional execution instruction (IT: If-Then)**
- Goal is to achieve code density similar to Thumb with performance similar to ARM

- **ThumbEE (Thumb Execution Environment)** (we don't cover this...)

- First appeared in 2005 (in Cortex-A8)
- 4th instruction set state, making small changes to Thumb2
- ARM **deprecates** any use of ThumbEE ISA and ARMv8 removes support for ThumbEE

Table A2-1 J and T bit encoding in ISETSTATE

J	T	Instruction set state
0	0	ARM
0	1	Thumb2
1	0	Jazelle
1	1	ThumbEE

CPSR

- Current Program Status Register (CPSR) is accessible in all modes
- Contains all condition flags, interrupt disable bits, the current processor mode

Table A2-1 J and T bit encoding in ISETSTATE

J	T	Instruction set state
0	0	ARM
0	1	Thumb2
1	0	Jazelle
1	1	ThumbEE

Program status registers

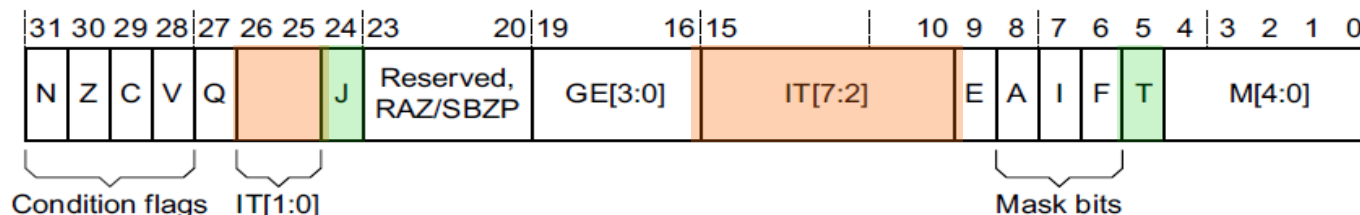
The *Current Program Status Register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *Saved Program Status Register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

Note

User mode and System mode do not have an SPSR, because they are not exception modes. All instructions that read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

Format of the CPSR and SPSRs

The CPSR and SPSR bit assignments are:



Interworking

- In ARM, the **instruction set change** from ARM to Thumb2 or Thumb2 to ARM is called **interworking**
- Switching is done via BX or BLX instruction
 - BLX label
 - Always changes the instruction set; It changes a processor in ARM state to Thumb2 state, or a processor in Thumb2 state to ARM state
 - Use BX lr to return
 - BLX Rm
 - If bit[0] of Rm is 0, the processor changes to, or remain in ARM state
 - If bit[0] of Rm is 1, the processor changes to, or remain in Thumb2 state
 - Use BX lr to return
 - BX Rm
 - If bit[0] of Rm is 0, the processor changes to, or remain in ARM state
 - If bit[0] of Rm is 1, the processor changes to, or remain in Thumb2 state

BLX, BX

A8.8.25 BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, and changes instruction set from ARM to Thumb, or from Thumb to ARM.

A8.8.26 BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address and instruction set specified by a register.

A8.8.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

	Caller	Callee	
ISA Instruction Set Architecture)	Arm	Arm	①
		Thumb2	②
	Thumb2	Arm	③
		Thumb2	④

BLX, BX

BLX label	Branch with Link and Exchange Instruction <ul style="list-style-type: none">Used to call a subroutineAlways change Instruction set between ARM and Thumb2	BLX foo // Destination is a function <i>foo()</i>
BLX Rm	Branch and Exchange Instruction <ul style="list-style-type: none">Used to call a subroutineBranch to a target location in RmInstruction set (ARM, Thumb2) is specified in Rm[0]<ul style="list-style-type: none">If Rm[0] is 0, CPU goes to or remains in ARM stateIf Rm[0] is 1, CPU goes to or remains in Thumb2 state	ldr r3, =foo blx r3 // Destination is in r3 register
BX Rm	Branch and Exchange Instruction <ul style="list-style-type: none">Branch to a target location in RmInstruction set (ARM, Thumb2) is specified in Rm[0]<ul style="list-style-type: none">If Rm[0] is 0, CPU goes to or remains in ARM stateIf Rm[0] is 1, CPU goes to or remains in Thumb2 state	bx r3 // Destination is in r3 register

AAPCS (Procedure Call Standard for ARM Architecture)

5.1.1 Core registers

There are 16, 32-bit core (integer) registers visible to the ARM and Thumb instruction sets. These are labeled r0-r15 or R0-R15. Register names may appear in assembly language in either upper case or lower case. In this specification upper case is used when the register has a fixed role in the procedure call standard. *Table 2, Core registers and AAPCS usage* summarizes the uses of the core registers in this standard. In addition to the core registers there is one status register (CPSR) that is available for use in conforming code.

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

```
int foo(int a, unsigned b)
{
    int c;

    c = a + (int) b;
    return c;
}
```

- Arguments are passed via r0, r1, r2, and r3
- Results are passed via r0 and r1

'BLX label' example

(Arm → Arm , Arm → Thumb2)

Arm code

```
mov r0, #1

// Switch from ARM to Thumb2
blx thumb_C_test // branch to C code (Thumb2 mode)

// Switch (?) from ARM to ARM
blx arm_C_test // The spec says blx always changes the instruction set.
// But, it does NOT. It seems that the compiler changes it to BL instruction
// since the pragma in C Code indicates the code is for ARM

mov r1, r0 // expecting r0 = 0x5555_5555 according to the calling convention
```

Thumb2 code

```
#pragma GCC target ("thumb")
int thumb_C_test(int a)
{
    int c;

    c = a + 0x11223343;

    return c;
}
```

Arm code

```
#pragma GCC target ("arm")
int arm_C_test(int a)
{
    int c;

    c = a + 0x44332211;

    return c;
}
```


Compiled Version

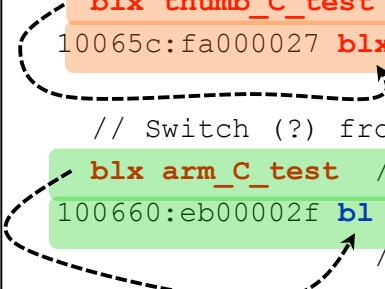
```
forever:

    mov r0, #1
100658:e3a00001 movr0, #1

    // Switch from ARM to Thumb2
    blx thumb_C_test // branch to C code (Thumb2 mode)
10065c:fa000027 blx 100700 <thumb_C_test>

    // Switch (?) from ARM to ARM
    blx arm_C_test // The spec says blx always changes the instruction set.
100660:eb00002f bl 100724 <arm_C_test>
    // But, it does NOT. It seems that the compiler changes it to BL instruction
    // since the pragma in C Code indicates the code is for ARM

    mov r1, r0 // expecting r0 = 0x5555_5555 according to the calling convention
100664:e1a01000 movr1, r0
```



The diagram illustrates the flow of execution between two branch instructions. A dashed arrow originates from the `blx thumb_C_test` instruction at address 10065c and points to the `blx arm_C_test` instruction at address 100660. This indicates that the first branch instruction successfully branches to the second instruction, despite the comment suggesting it should switch to Thumb2 mode.

'BLX Rm' example

(Arm → Thumb2)

The screenshot shows a disassembler window with the file 'csd_asm.S'. The main window displays assembly code. An orange dashed box highlights the ARM code section, which includes instructions to move r0 to r1, load thumb_Assem_test into r0, and branch to r0. A green dashed box highlights the Thumb2 code section, which starts with a .thumb directive, defines a function, and contains a loop. Annotations include a green arrow pointing from the 'blx r0' instruction to the Thumb2 code, and an orange arrow pointing from the 'blx thumb_Assem_test' instruction to the 'b forever' instruction. The right-hand pane shows the 'Registers' window with a table of register values.

Arm code

```
mov r1, r0    // expecting r0 = 0x5555_5555 atc
// Use either (a) 2 insts (ldr, blx) or (b) blx
// (a) Switch from ARM to Thumb2,
ldr r0,=thumb_Assem_test
blx r0

// (b) Switch from ARM to Thumb2
blx thumb_Assem_test // branch to assembly code

b forever
```

Thumb2 code

```
.thumb
.thumb_func
thumb_Assem_test:
// If-Then conditional execution in Thumb2
mov r0, #1
mov r1, #2
bx lr
```

Registers

Name	Hex
r0	00100679
r1	55555555
r2	11223344
r3	55555555
r4	00000044
r5	00000055
r6	00000066
r7	00000077
r8	00000088
r9	00000099
r10	000000aa
r11	000000bb
r12	000000cc
sp	00101bd0
lr	00100664
pc	0010066c
cpsr	000000df

'BLX Rm' Example

(Arm → Arm , Arm → Thumb2)

Thumb2 code

```
#pragma GCC target ("thumb")

int sum2(int n1, int n2) {
    return (n1 + n2);
}

int sub2(int n1, int n2) {
    return (n1 - n2);
}
```

Arm code

```
#pragma GCC target ("arm")

int and2(int n1, int n2) {
    return (n1 & n2);
}

int arm_C_test_fp(char op) {
    int res;

    int (*fp) (int, int);

    switch (op) {
        case '+': fp = sum2; break; // Thumb2 function
        case '-': fp = sub2; break; // Thumb2 function
        default: fp = and2; // ARM function
    }

    res = fp(10, 13);

    return (res);
}
```

Compiled code

```
switch (op) {
10076c: e55b300d    ldrb    r3, [fp, #-13]
100770: e353002b    cmp    r3, #43 ; 0x2b
100774: 0a000002    beq    100784 <arm_C_test_fp+0x2c>
100778: e353002d    cmp    r3, #45 ; 0x2d
10077c: 0a000004    beq    100794 <arm_C_test_fp+0x3c>
100780: ea000007    b      1007a4 <arm_C_test_fp+0x4c>
        case '+': fp = sum2; break; // Thumb2 function
100784: e30036cd    movw   r3, #1741 ; 0x6cd ①
100788: e3403010    movt   r3, #16
10078c: e50b3008    str    r3, [fp, #-8]
100790: ea000006    b      1007b0 <arm_C_test_fp+0x58>
        case '-': fp = sub2; break; // Thumb2 function
100794: e30036e9    movw   r3, #1769 ; 0x6e9 ②
100798: e3403010    movt   r3, #16
10079c: e50b3008    str    r3, [fp, #-8]
1007a0: ea000002    b      1007b0 <arm_C_test_fp+0x58>
        default: fp = and2; // ARM function
1007a4: e3003728    movw   r3, #1832 ; 0x728 ③
1007a8: e3403010    movt   r3, #16
1007ac: e50b3008    str    r3, [fp, #-8]
}

res = fp(10, 13);
1007b0: e51b3008    ldr    r3, [fp, #-8]
1007b4: e3a0100d    mov    r1, #13
1007b8: e3a0000a    mov    r0, #10
1007bc: e12fff33    blx    r3 ④
1007c0: e50b000c    str    r0, [fp, #-12]
```

'BLX Rm' Example

(Thumb2 → Arm)

Disassembly | csd_asm.S

Thumb2 code

```
.thumb
.thumb_func
thumb_Assem_test:
    // If-Then conditional execution in Thumb2
    mov r0, #1

    push {lr}
    // Switch from Thumb2 to ARM
    // Check if LSB of r14 (lr) is set to 1 after blx.
    blx arm_C_test

    pop {lr}

    // to see if thumb2 has a dedicated register set
    // it turn out to be wrong... That is, ARM and Thumb share
```

Registers

Name	Hex
r4	00000044
r5	00000055
r6	00000066
r7	00000077
r8	00000088
r9	00000099
r10	000000aa
r11	000000bb
r12	000000cc
sp	00101bcc
lr	00100670
pc	0010067e
cpsr	000000ff

Before executing
'blx arm_C_test'

Disassembly | csd_asm.S | csd_main.c

Arm code

```
#pragma GCC target ("thumb")

int thumb_C_test(int a)
{
    int c;

    c = a + 0x11223343;

    return c;
}

#pragma GCC target ("arm")

int arm_C_test(int a)
{
    int c;

    c = a + 0x44332211;

    return c;
}
```

Registers

Name	Hex	D
r4	00000044	6
r5	00000055	8
r6	00000066	1
r7	00000077	1
r8	00000088	1
r9	00000099	1
r10	000000aa	1
r11	00101bc8	1
r12	000000cc	2
sp	00101bb4	1
lr	00100683	1
pc	00100734	1
cpsr	000000df	2
n	0	0
z	0	0
c	0	0
v	0	0

After executing
'blx arm_C_test'

'BX Rm' Example

(Arm → Thumb2)

```
#pragma GCC target ("arm")

int arm_C_test(int a)
{
    100724:  e52db004    push    {fp}          ; (str fp, [sp, #-4]!)
    100728:  e28db000    add fp, sp, #0
    10072c:  e24dd014    sub sp, sp, #20
    100730:  e50b0010    str r0, [fp, #-16]
        int c;

        c = a + 0x44332211;
    100734:  e51b2010    ldr r2, [fp, #-16]
    100738:  e3023211    movw   r3, #8721     ; 0x2211
    10073c:  e3443433    movt   r3, #17459    ; 0x4433
    100740:  e0823003    add r3, r2, r3
    100744:  e50b3008    str r3, [fp, #-8]

        return c;
    100748:  e51b3008    ldr r3, [fp, #-8]
}
    10074c:  e1a00003    mov r0, r3
    100750:  e24bd000    sub sp, fp, #0
    100754:  e49db004    pop {fp}          ; (ldr fp, [sp], #4)
    100758:  e12ffffe    bx lr
```

IT (If-Then) Instruction

Syntax: IT {T|E} {T|E} {T|E} <cond>

- Makes the next 1-4 instructions conditional
- Any condition code may be used
- Doesn't affect condition flags
- 16-bit instructions in block **do not** affect condition flags (except CMP, CMN & TST)
- 32-bit instructions in block **do** affect condition flags (normal rules apply)
- Current 'if-then status' stored in CPSR
 - Conditional block may be safely interrupted and returned to
 - Not recommended to branch into or out of 'if-then' block

```
if (r0 == 0)
    r0 = *r1 + 2;
else
    r0 = *r2 + 4;
```

```
CMP    r0, #0    // if
ITTEE EQ      // if
LDREEQ    r0, [r1]
ADDEEQ    r0, #2
LDRNE    r0, [r2]
ADDNE    r0, #4
```

Example #1

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

```
CMP    r0, #0    // if
ITE   EQ        // if
ADDEQ   r1, #1
ADDNE  r2, #1
```

Example #2

① C pseudo code

```
if (r0 == r1) then
    add r0, r0, #2
    add r1, r1, #2
else
    add r0, r0, #4
    add r1, r1, #4
```

```
// If-Then conditional execution in Thumb2
mov r0, #1
mov r1, #2
cmp r0, r1
ittee eq
addeq r0, #2
addeq r1, #2
addne r0, #4
addne r1, #4
```

“then” statements

“else” statements

② Assembly equivalent



```
// If-Then conditional execution in Thumb2
mov r0, #1
100770: f04f 0001  mov.w  r0, #1
mov r1, #2
100774: f04f 0102  mov.w  r1, #2
cmp r0, r1
100778: 4288      cmp r0, r1
ittee eq
10077a: bf07      ittee  eq
addeq r0, #2
10077c: 3002      addeq  r0, #2
addeq r1, #2
10077e: 3102      addeq  r1, #2
addne r0, #4
100780: 3004      addne  r0, #4
addne r1, #4
100782: 3104      addne  r1, #4
```

③ Compiled binary

Example #3

1

One "then" statement

```
iteeq gt // "greater than" condition for signed numbers
addgt r0, r2, #2
addle r1, r3, #2
suble r10, r12, #2
suble r11, r13, #2
```

Three "else" statements

2

One "then" statement

```
it hs // "higher or same" condition for unsigned numbers
addhs r0, r2, #2
```

```
iteeq gt // "greater than" condition for signed numbers
100784: bfcf iteq gt
addgt r0, r2, #2
100786: lc90 addgt r0, r2, #2
addle r1, r3, #2
100788: lc99 addle r1, r3, #2
suble r10, r12, #2
10078a: flac 0a02 suble.w s1, ip, #2
suble r11, r13, #2
10078e: flad 0b02 suble.w fp, sp, #2
```

```
it hs // "higher or same" condition for unsigned numbers
100792: bf28 it cs
addhs r0, r2, #2
100794: lc90 addcs r0, r2, #2
```

Conditional Execution

- ARM vs Thumb2 -

CMP r0, r2

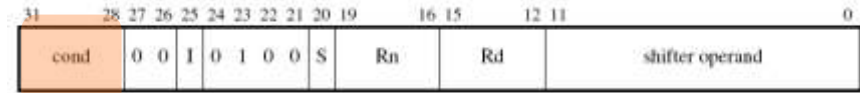
LDR^{EQ} r0, [r1]

ADD^{GT} r0, r0, #2

0xe1500001

0x05910000

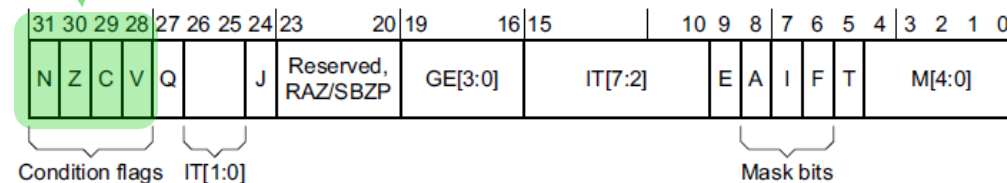
0xc2800002



ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADD can optionally update the condition code flags, based on the result.

The CPSR and SPSR bit assignments are:

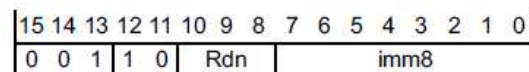


ARM

Encoding T2

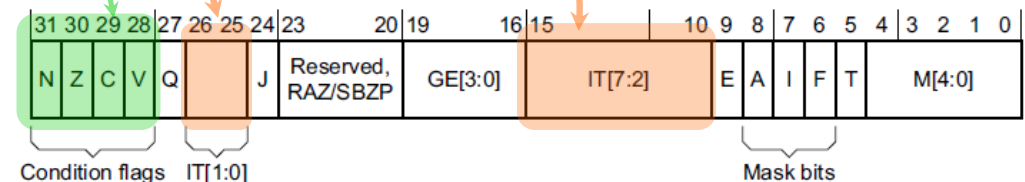
ADDS <Rdn>, #<imm8>

ADD<c> <Rdn>, #<imm8>



ARMv4T, ARMv5T*, ARMv6*, ARMv7

The CPSR and SPSR bit assignments are:



CPSR'IT[7:0] = {firstcond . Mask}

Thumb2

CMP r0, r1

ITTEE EQ

ADDEQ r0, #2

ADDEQ r1, #2

ADDNE r0, #4

ADDNE r1, #4

0x4288

0xbf07

0x3002

0x3102

0x3004

0x3104

IT (If-Then) Instruction

- If-Then makes up to 4 following instructions conditional

Encoding T1 ARMv6T2, ARMv7

IT{<x>{<y>{<z>}}} <firstcond>

Not permitted in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	firstcond				mask			

CPSR'IT[7:0] = {firstcond . Mask}

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

Related encodings See *If-Then, and hints* on page A6-229.

Assembler syntax

IT{<x>{<y>{<z>}}}{<q>} <firstcond>

where:

<x>	The condition for the second instruction in the IT block.
<y>	The condition for the third instruction in the IT block.
<z>	The condition for the fourth instruction in the IT block.
<q>	See <i>Standard assembler syntax fields</i> on page A8-287. An IT instruction must be unconditional.
<firstcond>	The condition for the first instruction in the IT block. See Table A8-1 on page A8-288 for the range of conditions available, and the encodings.

Each of <x>, <y>, and <z> can be either:

T	Then. The condition for the instruction is <firstcond>.
E	Else. The condition for the instruction is the inverse of <firstcond>. The condition code is the same as <firstcond>, except that the least significant bit is inverted. E must not be specified if <firstcond> is AL.

IT (If-Then) Instruction Format

ITTEE EQ

Encoding T1 ARMv6T2, ARMv7
IT{<x>{<y>{<z>}}} <firstcond>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	firstcond				mask			

CPSR'IT[7:0] = {firstcond . Mask}

- firstcond[3:0] = EQ = 4'b0000
- mask[3:0] = 4'b0111

ITTEE EQ

Table A8-2 Determination of mask field

<x>	<y>	<z>	mask[3]	mask[2]	mask[1]	mask[0]
Omitted	Omitted	Omitted	1	0	0	0
T	Omitted	Omitted	firstcond[0]	1	0	0
E	Omitted	Omitted	NOT firstcond[0]	1	0	0
T	T	Omitted	firstcond[0]	firstcond[0]	1	0
E	T	Omitted	NOT firstcond[0]	firstcond[0]	1	0
T	E	Omitted	firstcond[0]	NOT firstcond[0]	1	0
E	E	Omitted	NOT firstcond[0]	NOT firstcond[0]	1	0
T	T	T	firstcond[0]	firstcond[0]	firstcond[0]	1
E	T	T	NOT firstcond[0]	firstcond[0]	firstcond[0]	1
T	E	T	firstcond[0]	NOT firstcond[0]	firstcond[0]	1
E	E	T	NOT firstcond[0]	NOT firstcond[0]	firstcond[0]	1
T	T	E	firstcond[0]	firstcond[0]	NOT firstcond[0]	1
E	T	E	NOT firstcond[0]	firstcond[0]	NOT firstcond[0]	1
T	E	E	firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1
E	E	E	NOT firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1

Condition Codes (pg# A8-288)

top 3 bits of the condition code



Table A8-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) ^a	Condition flags
0000	EQ	Equal	Equal	$Z == 1$
0001	NE	Not equal	Not equal, or unordered	$Z == 0$
0010	CS ^b	Carry set	Greater than, equal, or unordered	$C == 1$
0011	CC ^c	Carry clear	Less than	$C == 0$
0100	MI	Minus, negative	Less than	$N == 1$
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	$N == 0$
0110	VS	Overflow	Unordered	$V == 1$
0111	VC	No overflow	Not unordered	$V == 0$
1000	HI	Unsigned higher	Greater than, or unordered	$C == 1$ and $Z == 0$
1001	LS	Unsigned lower or same	Less than or equal	$C == 0$ or $Z == 1$
1010	GE	Signed greater than or equal	Greater than or equal	$N == V$
1011	LT	Signed less than	Less than, or unordered	$N != V$
1100	GT	Signed greater than	Greater than	$Z == 0$ and $N == V$
1101	LE	Signed less than or equal	Less than, equal, or unordered	$Z == 1$ or $N != V$
1110	None (AL) ^d	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.

b. HS (unsigned higher or same) is a synonym for CS.

c. LO (unsigned lower) is a synonym for CC.

d. AL is an optional mnemonic extension for always, except in IT instructions. For details see [IT](#) on page A8-390.

IT (If-Then) field in CPSR

- IT[7:0] in CPSR holds the If-Then execution state bits for IT instruction
 - It applies to the IT block of 1 ~ 4 instructions immediately following the IT instruction
 - IT[7:5] holds the base condition** for the current IT block
 - The base condition is the **top 3 bits of the condition code** specified by the <firstcond> field of the IT instruction
 - IT[4:0] encodes the size of the IT block & LSB of the condition code**
 - The size of the block (#insts) is implied by the position of the least significant 1 in this field
 - LSB of condition code for each instruction in the block

ITTEE EQ

Table A2-2 Effect of IT execution state bits

IT bits ^a						Note
[7:5]	[4]	[3]	[2]	[1]	[0]	
cond_base	P1	P2	P3	P4	1	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	1	0	Entry point for 3-instruction IT block
cond_base	P1	P2	1	0	0	Entry point for 2-instruction IT block
cond_base	P1	1	0	0	0	Entry point for 1-instruction IT block
000	0	0	0	0	0	Normal execution, not in an IT block

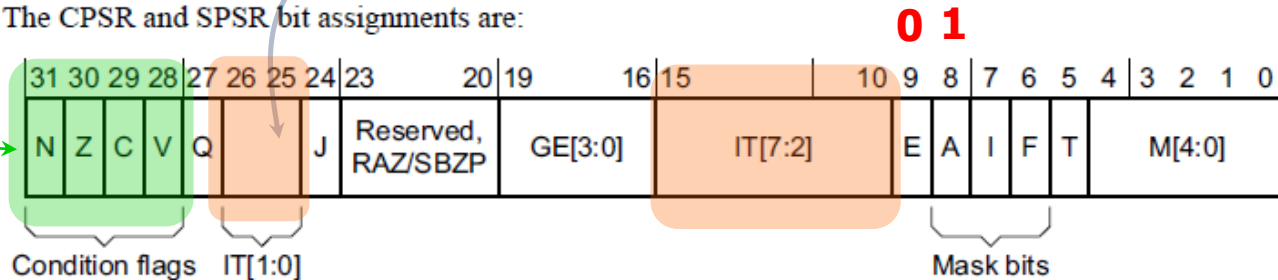
a. Combinations of the IT bits not shown in this table are reserved.

Lab Code Example

```

mov r0, #1
CPSR = 0x2000_01FF (IT = 8'b0000_0000) after cmp r0, #0
CPSR = 0x2600_05FF (IT = 8'b0000_0111) after ittee eq
CPSR = 0x2400_0DFF (IT = 8'b0000_1110) after addeq r0, #2
CPSR = 0x2000_1DFF (IT = 8'b0001_1100) after addeq r1, #2
CPSR = 0x2000_19FF (IT = 8'b0001_1000) after addne r0, #4
CPSR = 0x2000_01FF (IT = 8'b0000_0000) after addne r1, #4
  
```

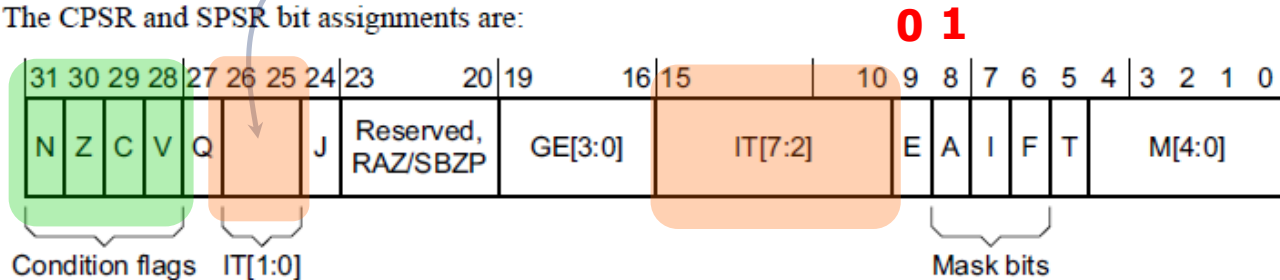
The CPSR and SPSR bit assignments are:



Lab Code Example

CPSR = 0x2000_01FF (IT = 8'b0000_0000) after `mov r0, #1`
 CPSR = (IT = 8'b0000_0000) after `cmp r0, #0`
 CPSR = (IT =) after `ittee ge`
 CPSR = (IT =) after `addge r0, #2`
 CPSR = (IT =) after `addge r1, #2`
 CPSR = (IT =) after `addlt r0, #4`
 CPSR = (IT =) after `addlt r1, #4`

The CPSR and SPSR bit assignments are:



Backup Slides

Instruction Encoding in ARM and Thumb

- In ARMv7 Architecture Reference Manual,
 - T1, T2, T3 ... for the first, second, third, and any additional Thumb encodings
 - A1, A2, A3 ... for the first, second, third, and any additional ARM encodings
 - E1, E2, E3 ... for the first, second, third, and any additional ThumbEE encodings that are not also Thumb encodings

Example

A8.8.4 ADD (immediate, Thumb)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

ADDS <Rd>, <Rn>, #<imm3>

Outside IT block.

ADD<C> <Rd>, <Rn>, #<imm3>

Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn		Rd			

d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

ADDS <Rdn>, #<imm8>

Outside IT block.

ADD<C> <Rdn>, #<imm8>

Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

Encoding T3 ARMv6T2, ARMv7

ADD{S}<C>.W <Rd>, <Rn>, #<const>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3				Rd				imm8							

if Rd == '1111' && S == '1' then SEE CMN (immediate);

if Rn == '1101' then SEE ADD (SP plus immediate);

d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = ThumbExpandImm(i:imm3:imm8);

if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;

Encoding T4 ARMv6T2, ARMv7

ADDW<C> <Rd>, <Rn>, #<imm12>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							

A8.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<C> Is an optional field. It specifies the condition under which the instruction is executed. See [Conditional execution on page A8-288](#) for the range of available conditions and their encoding. If <C> is omitted, it defaults to *always* (AL).

<q> Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

.N Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

.W Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description.

———— **Note** ————

When assembling to the ARM instruction set, the .N qualifier produces an assembler error and the .W qualifier has no effect.

—————