

04

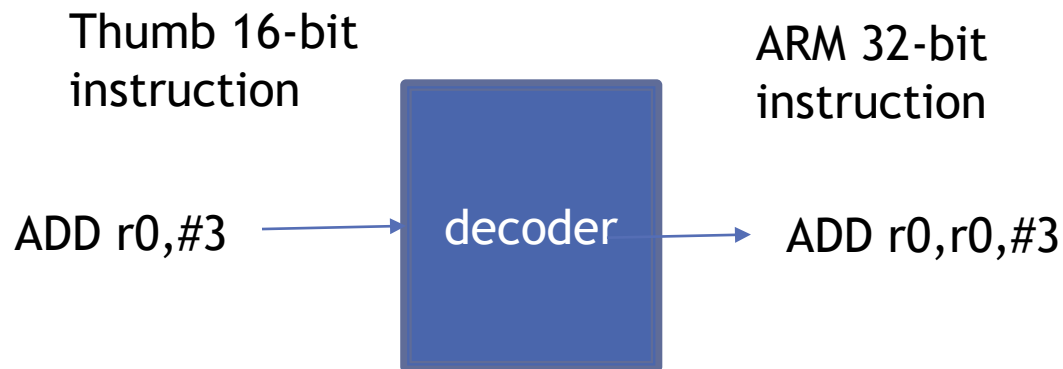
THUMB INSTRUCTION SET

마이크로프로
세서응용

Handong
Global
University

Thumb is 16-bit instruction set

- Thumb encodes a subset of 32-bit ARM instructions into 16-bit instruction set space
 - Each Thumb instruction is related to a 32-bit ARM instruction
- Thumb has higher code density than ARM:
 - Thumb implementation usually uses 30% less memory than equivalent ARM implementation
 - use Thumb for memory-constrained systems



Thumb has higher code density

ARM code

ARMDivide

; IN: r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

```
      MOV    r3,#0
loop  SUBS    r0,r0,r1
      ADDGE  r3,r3,#1
      BGE    loop
      ADD    r2,r0,r1
```

$5 \times 4 = 20$ bytes

Thumb code

ThumbDivide

; IN: r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

```
      MOV    r3,#0
loop  ADD     r3,#1
      SUB     r0,r1
      BGE     loop
      SUB     r3,#1
      ADD     r2,r0,r1
```

$6 \times 2 = 12$ bytes

Thumb Instruction set List

Mnemonics	THUMB ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
ASR	v1	arithmetic shift right
B	v1	branch relative
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v2	breakpoint instructions
BL	v1	relative branch with link
BLX	v2	branch with link and exchange
BX	v1	branch with exchange
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit integers
EOR	v1	logical exclusive OR of two 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1	load a single value from a virtual address in memory
LSL	v1	logical shift left
LSR	v1	logical shift right
MOV	v1	move a 32-bit value into a register
MUL	v1	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
NEG	v1	negate a 32-bit value
ORR	v1	logical bitwise OR of two 32-bit values
POP	v1	pops multiple registers from the stack
PUSH	v1	pushes multiple registers to the stack
ROR	v1	rotate right a 32-bit value
SBC	v1	subtract with carry a 32-bit value
STM	v1	store multiple 32-bit registers to memory
STR	v1	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
TST	v1	test bits of a 32-bit value

Thumb Instruction Encoding(1)

Instruction classes (indexed by <i>op</i>)				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
LSL		LSR		0	0	0	0	<i>op</i>	<i>immed5</i>				<i>Lm</i>			<i>Ld</i>					
ASR				0	0	0	1	0	<i>immed5</i>				<i>Lm</i>			<i>Ld</i>					
ADD		SUB		0	0	0	1	1	0	<i>op</i>	<i>Lm</i>		<i>Ln</i>			<i>Ld</i>					
ADD		SUB		0	0	0	1	1	1	<i>op</i>	<i>immed3</i>		<i>Ln</i>			<i>Ld</i>					
MOV		CMP		0	0	1	0	<i>op</i>	<i>Ld/Ln</i>			<i>immed8</i>									
ADD		SUB		0	0	1	1	<i>op</i>	<i>Ld</i>			<i>immed8</i>									
AND		EOR		0	1	0	0	0	0	0	0	<i>op</i>	<i>Lm/Ls</i>			<i>Ld</i>					
ASR		ADC		0	1	0	0	0	0	0	1	<i>op</i>	<i>Lm/Ls</i>			<i>Ld</i>					
TST		NEG		0	1	0	0	0	0	1	0	<i>op</i>	<i>Lm</i>			<i>Ld/Ln</i>					
ORR		MUL		0	1	0	0	0	0	1	1	<i>op</i>	<i>Lm</i>			<i>Ld</i>					
CPY	Ld, Lm			0	1	0	0	0	1	1	0	0	0	<i>Lm</i>			<i>Ld</i>				
ADD		MOV	Ld, Hm		0	1	0	0	0	1	<i>op</i>	0	0	1	<i>Hm & 7</i>			<i>Ld</i>			
ADD		MOV	Hd, Lm		0	1	0	0	0	1	<i>op</i>	0	1	0	<i>Lm</i>			<i>Hd & 7</i>			
ADD		MOV	Hd, Hm		0	1	0	0	0	1	<i>op</i>	0	1	1	<i>Hm & 7</i>			<i>Hd & 7</i>			
CMP				0	1	0	0	0	1	0	1	0	1	<i>Hm & 7</i>			<i>Ln</i>				
CMP				0	1	0	0	0	1	0	1	1	0	<i>Lm</i>			<i>Hn & 7</i>				
CMP				0	1	0	0	0	1	0	1	1	1	<i>Hm & 7</i>			<i>Hn & 7</i>				
BX		BLX		0	1	0	0	0	1	1	1	<i>op</i>	<i>Rm</i>			0 0 0					
LDR	Ld, [pc, # <i>immed</i> *4]			0	1	0	0	1	<i>Ld</i>			<i>immed8</i>									
STR		STRH		0	1	0	1	0	<i>op</i>		<i>Lm</i>		<i>Ln</i>			<i>Ld</i>					
LDR		LDRH		0	1	0	1	1	<i>op</i>		<i>Lm</i>		<i>Ln</i>			<i>Ld</i>					
STR		LDR	Ld, [Ln, # <i>immed</i> *4]		0	1	1	0	<i>op</i>	<i>immed5</i>				<i>Ln</i>			<i>Ld</i>				
STRB		LDRB	Ld, [Ln, # <i>immed</i>]		0	1	1	1	<i>op</i>	<i>immed5</i>				<i>Ln</i>			<i>Ld</i>				
STRH		LDRH	Ld, [Ln, # <i>immed</i> *2]		1	0	0	0	<i>op</i>	<i>immed5</i>				<i>Ln</i>			<i>Ld</i>				

Thumb Instruction Encoding(2)

Instruction classes (indexed by op)

STR | LDR Ld, [sp, #immed*4]

```
ADD  Ld, pc, #immed*4 |
```

```
ADD Ld, sp, #immed*4
```

```
ADD sp, #immed*4 | SUB sp,
    #immed*4
```

SXTH | SXTB | UXTH | UXTB

REV	REV16		REVSH
-----	-------	--	-------

PUSH		POP
------	--	-----

SETEND LE | SETEND BE

CPSIE | CPSID

BKPT immmed8

```
STMIA | LDMIA Ln!, {register-list}
```

```
B<cond> instruction_address+
    4+offset*2
```

Undefined and expected to remain so

SWI immmed8

```
B instruction_address+4+offset*2
```

```
BLX ((instruction+4+
      (poff<<12)+offset*4) &~ 3)
```

This must be preceded by a branch prefix instruction.

This is the branch prefix instruction. It must be followed by a relative BL or BLX instruction.

BL instruction+4+ (poff<<12)+
offset*2 This must be preceded by a
branch prefix instruction.

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

1	0	0	1	<i>op</i>	<i>Ld</i>			<i>immed8</i>							
1	0	1	0	<i>op</i>	<i>Ld</i>			<i>immed8</i>							
1	0	1	1	0	0	0	0	<i>op</i>	<i>immed7</i>						
1	0	1	1	0	0	1	0	<i>op</i>	<i>Lm</i>			<i>Ld</i>			
1	0	1	1	1	0	1	0	<i>op</i>	<i>Lm</i>			<i>Ld</i>			
1	0	1	1	<i>op</i>	1	0	<i>R</i>	<i>register_list</i>							
1	0	1	1	0	1	1	0	0	1	0	1	<i>op</i>	0	0	0
1	0	1	1	0	1	1	0	0	1	1	<i>op</i>	0	<i>a</i>	<i>i</i>	<i>f</i>
1	0	1	1	1	1	1	0	<i>immed8</i>							
1	1	0	0	<i>op</i>	<i>Ln</i>			<i>register_list</i>							
1	1	0	1	<i>cond</i> < 1110				signed 8-bit offset							
1	1	0	1	1	1	1	0	<i>x</i>							
1	1	0	1	1	1	1	1	<i>immed8</i>							
1	1	1	0	0	signed 11-bit <i>offset</i>										
1	1	1	0	1	unsigned 10-bit <i>offset</i>										0
1	1	1	1	0	signed 11-bit prefix offset <i>poff</i>										
1	1	1	1	1	unsigned 11-bit <i>offset</i>										

Thumb Register Usage

registers	Access
r0-r7	fully accessible
r8-r12	only accessible by MOV, ADD, CMP
r13 (sp)	limited accessibility
r14 (lr)	limited accessibility
r15 (pc)	limited accessibility
cpsr	only indirect access
spsr	no access

- No direct access to cpsr or spsr
- in order to alter cpsr or spsr, you must switch into ARM state to use MSR and MRS
- No coprocessor instruction in Thumb state

ARM-Thumb interworking(1)

- The method of linking ARM and Thumb code together
- ATPCS ARM and Thumb procedure call standards
 - Call Thumb from ARM routine, core has to change state
 - State change is shown in T bit in CPSR
 - The BX and BLX instruction cause a switch between ARM and Thumb state
 - ARM BX instruction enters **Thumb state** only if **bit 0 of the address in Rn is set to binary 1**,
otherwise it enters **ARM**

ARM-Thumb Interworking(2)

● ARM state to Thumb state Switch

start:

```
; ARM instructions here  
; Prepare the address of the Thumb function, setting the LSB to 1  
ldr r0, =thumb_function+1  
; Branch to the address in r0 and switch to Thumb mode  
bx r0  
; More ARM instructions (if needed)
```

```
.thumb
```

```
.align 2
```

```
thumb_function: ;
```

```
; Thumb instructions here  
; Return from the function (switching back to ARM mode if necessary)  
bx lr
```

- The ARM code prepares to switch to Thumb state by loading the address of the Thumb function into a register (**r0**) and sets the **LSB to 1**.
- the **bx r0** instruction performs the branch and switches the processor to Thumb state because the LSB of the address in **r0** is set
- The **bx lr** instruction at the end of the **thumb_function** ensures that the processor returns to the previous state (ARM or Thumb), depending on the lr state.

BX/BLX instruction

Use BX

```
; ARM code
CODE32 ; word aligned
LDR r0, =thumbCode+1
; to enter Thumb state

MOV lr, pc ; set the return address
BX r0 ; branch to Thumb code & mode
; continue here

; Thumb code
CODE16 ; halfword aligned
thumbCode
ADD r1, #1
BX lr ; return to ARM code & state
```

Use BLX

```
; ARM code
CODE32 ; word aligned
LDR r0, =thumbCode+1
; enter Thumb state

BLX r0 ; Jump to Thumb state
; continue here

; Thumb code
CODE16 ; halfword aligned
thumbCode
ADD r1, #1
BX lr ; return to ARM code & state
```

Branch Thumb Instructions

- The conditional Branch instruction is the only conditionally executed instruction in Thumb state

- Syntax; B <cond> label

B label

BL label

B	branch	$pc = label$
BL	branch with link	$pc = label$ $lr = (\text{instruction address after the BL}) + 1$

- Return from BL subroutine call ($pc \leftarrow lr$)
 - MOV pc, lr
 - BX lr
 - POP {pc}

Thumb Data Processing Instructions

- a subset of ARM data processing instructions

- Syntax:

`<ADC|ADD|AND|BIC|EOR|MOV|MUL|MVN|NEG|ORR|SBC|SUB> Rd, Rm`

`<ADD|ASR|LSL|LSR|ROR|SUB> Rd, Rn #immediate`

`<ADD|MOV|SUB> Rd, #immediate`

`<ADD|SUB> Rd, Rn, Rm`

`ADD Rd, pc, #immediate`

`ADD Rd, sp, #immediate`

`<ADD|SUB> sp, #immediate`

`<ASR|LSL|LSR|ROR> Rd, Rs`

`<CMN|CMP|TST> Rn, Rm`

`CMP Rn, #immediate`

`MOV Rd, Rn`

Thumb data processing instructions

- Most Thumb instructions operates on low registers and
 - Updates the cpsr
 - Use two operands
 - No conditional execution

- The Exceptions are:
 - ✓ allow to use higher registers (r8-r14) and pc
 - ✓ do not update cpsr when using higher regs.
 - ✓ But, CMP instructions always updates the cpsr



```
MOV    Rd,Rn
ADD    Rd,Rm
CMP    Rn,Rm
ADD    sp, #immediate
SUB    sp, #immediate
ADD    Rd,sp,#immediate
ADD    Rd,pc,#immediate
```


Thumb single register Load-Store Instructions

- addressing mode
 - [Rn, Rm]: base reg + offset reg
 - [Rn, #imm5] : base reg + 5bit-imm
 - [pc|sp, #imm5] : pc or sp + 5bit imm
- 5bit-immediate offsets are multiplied by 1, 2, or 4 for byte access, 16-bit access, and 32-bit accesses

Syntax: <LDR|STR>{<B|H>} Rd, [Rn,#immediate]
LDR{<H|SB|SH>} Rd,[Rn,Rm]
STR{<B|H>} Rd,[Rn,Rm]
LDR Rd,[pc,#immediate]
<LDR|STR> Rd,[sp,#immediate]

LDR	load word into a register	$Rd \leftarrow mem32[address]$
STR	save word from a register	$Rd \rightarrow mem32[address]$
LDRB	load byte into a register	$Rd \leftarrow mem8[address]$
STRB	save byte from a register	$Rd \rightarrow mem8[address]$
LDRH	load halfword into a register	$Rd \leftarrow mem16[address]$
STRH	save halfword into a register	$Rd \rightarrow mem16[address]$
LDRSB	load signed byte into a register	$Rd \leftarrow SignExtend(mem8[address])$
LDRSH	load signed halfword into a register	$Rd \leftarrow SignExtend(mem16[address])$

Thumb Multiple-Register Load-Store Instructions

- Only support IA (increment after) mode

Syntax : <LDM|STM>IA Rn!, {low Register list}

LDMIA	load multiple registers	$\{Rd\}^{*N} \leftarrow \text{mem32}[Rn + 4 * N], Rn = Rn + 4 * N$
STMIA	save multiple registers	$\{Rd\}^{*N} \rightarrow \text{mem32}[Rn + 4 * N], Rn = Rn + 4 * N$

PRE

r1 = 0x00000001

r2 = 0x00000002

r3 = 0x00000003

r4 = 0x9000

- N: Number of registers in list

- Ex)

STMIA r4!, {r1, r2, r3}

POST mem32[0x9000] = 0x00000001

mem32[0x9004] = 0x00000002

mem32[0x9008] = 0x00000003

r4 = 0x900c

Thumb Stack Instructions

- Stack Pointer is r13 in Thumb mode and sp is automatically updated
- List of registers is limited to the low registers (r0 - r7) and lr (r14)

Syntax: POP {low_register_list[, pc]}
 PUSH {low_register_list[, lr]}

POP	pop registers from the stacks	$Rd^N \leftarrow \text{mem32}[sp + 4 * N], \text{ sp} = \text{sp} + 4 * N$
PUSH	push registers on to the stack	$Rd^N \rightarrow \text{mem32}[sp + 4 * N], \text{ sp} = \text{sp} - 4 * N$

- **POP {reg_list}**
 - synonym of **LDMFD** sp!, {reg_list} = synonym of **LDMIA**
- **PUSH {reg_list}**
 - synonym of **STMFD** sp!, {reg_list} = synonym of **STMDB**

reg_ist: r0-r7, lr, pc

- registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

Thumb Stack Instructions

- Example

```
; Call subroutine  
BL    ThumbRoutine  
; continue
```

ThumbRoutine

```
PUSH    {r1, lr}  
MOV     r0, #2  
POP     {r1, pc}
```

push with link

; enter subroutine

; return from subroutine

pop and return

17

Thumb SWI instruction

- Similar to ARM SWI instruction
- But, it differs in that ...
 - SWI number is limited to 0~255
 - not conditionally executed

Syntax: SWI immediate

SWI	software interrupt	<i>lr_svc</i> = address of instruction following the SWI <i>spsr_svc</i> = <i>cpsr</i> <i>pc</i> = vectors + 0x8 <i>cpsr</i> mode = SVC <i>cpsr</i> I = 1 (mask IRQ interrupts) <i>cpsr</i> T = 0 (ARM state)
-----	--------------------	--

Ex)

```
PRE    cpsr = nzcVqifT_USER
        pc = 0x00008000
        lr = 0x003ffffff ; lr = r14
        r0 = 0x12
```

0x00008000

SWI 0x45

```
POST   cpsr = nzcVqIfT_SVC
        spsr = nzcVqifT_USER
        pc = 0x00000008
        lr = 0x00008002
        r0 = 0x12
```


Thumb Instruction Summary

- Thumb instructions are 16-bits in length
- Thumb provides about 30% better code density than ARM code
- ARM-Thumb interworking uses BX and BLX instruction
- In Thumb, only branch instructions are executed conditionally
- Shift operations are separate instructions
- No Thumb instructions to access the coprocessors, cpsr, spsr

THUMB-2

Introduction of Thumb-2

- Issues of using ARM and Thumb instruction sets
 - The 16-bit Thumb instructions had too many limitations.
 - Switching between Thumb and ARM ISA added extra cycles.
 - Cortex-M4 (ARMv7 architecture, our target device) only execute Thumb-2 instructions.

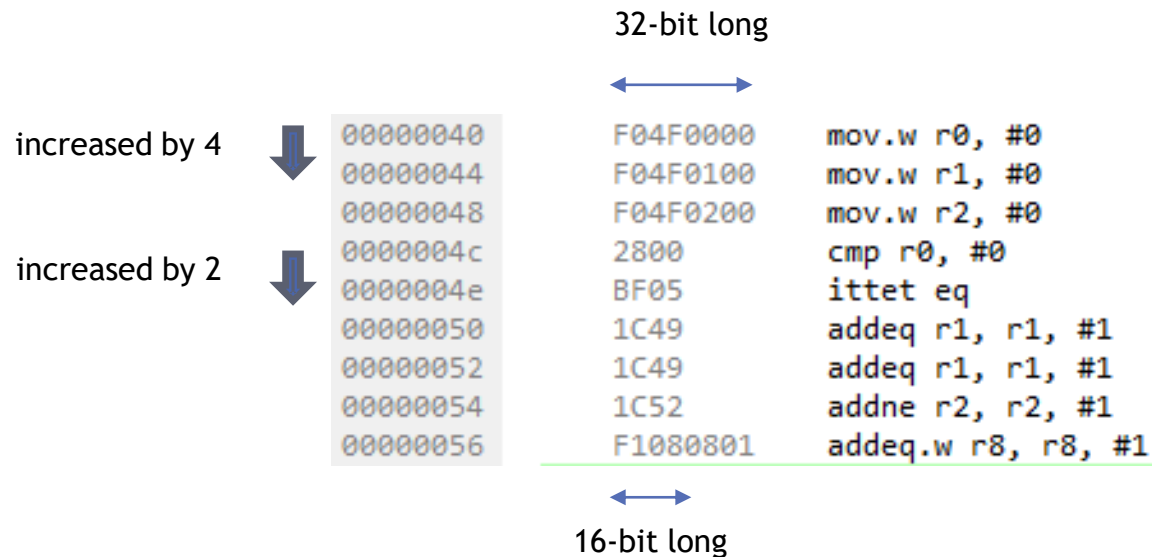
Architectures and Instruction Sets

Version	Example Core	ISA
v4T	ARM7TDMI, ARM9TDMI	ARM, Thumb
v5TE	ARM946E-S, ARM966E-S	ARM, Thumb
v5TEJ	ARM926EJ-S, ARM1026EJ-S	ARM, Thumb
v6	ARM1136 J(F)-S	ARM, Thumb
v6T2	ARM1156T2(F)-S	ARM, Thumb-2
v6-M	Cortex-M0, Cortex-M1	Thumb-2 subset
v7-A	Cortex-A5, Cortex-A8, Cortex-A12, Cortex-A15	ARM, Thumb-2
v7-R	Cortex-R4, Cortex-R5, Cortex-R7	ARM, Thumb-2
v7-M	Cortex-M3	Thumb-2
v7E-M	Cortex-M4	Thumb-2

Thumb-2 Instruction Format has 16-bits and 32-bits

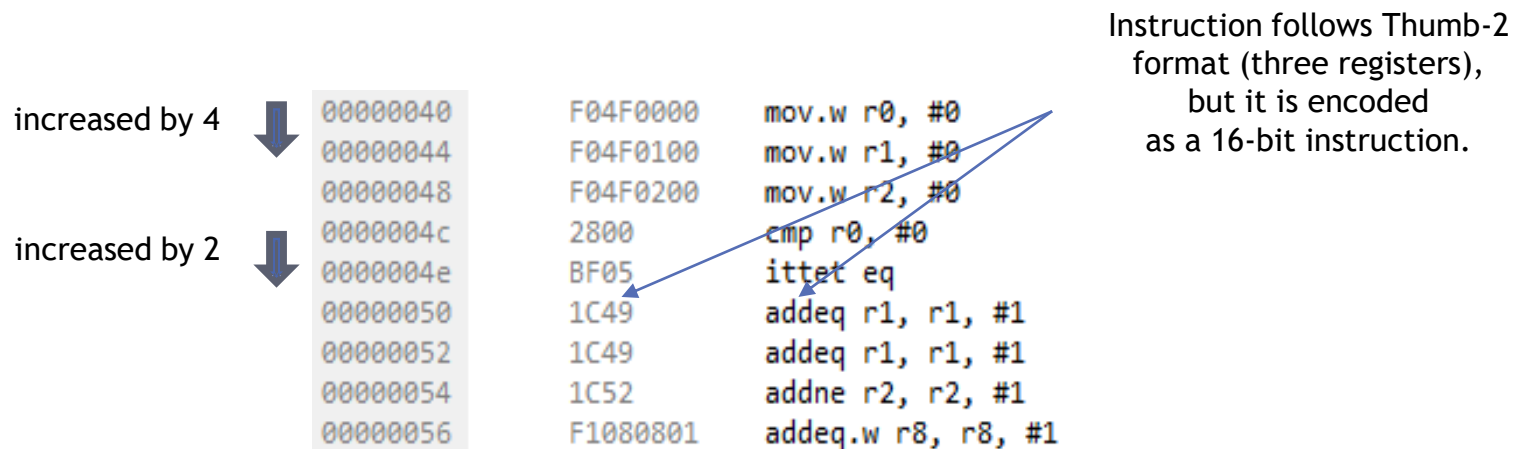
- Thumb-2

- A superset of Thumb
 - ✓ It contains all 16-bit Thumb instructions.
- Almost all the functionality of the ARM ISA is covered.
- Combination of both 16-bit and 32-bit instructions
 - ✓ Thumb-2 delivers overall code density comparable with Thumb, together with the performance levels associated with the ARM ISA.



Thumb-2 Instruction Encoding

- Thumb-2
 - The Thumb-2 instruction encodings are rather messy in order to squeeze as many useful instructions as possible into the 16-bit space.
 - ✓ For example, the ADD instruction has a 16-bit encoding for the three-register version, provided all the registers are the *low registers* (r0-r7).
 - ✓ Most Thumb 32-bit instructions cannot use the *pc* register as a source or destination register.



Now, interpret this as a 32-bit Thumb instruction.

Thumb-2 Instruction Format


- 32-bit Thumb-2 instructions
 - As 16-bit Thumb instructions are covered in the previous section, here we will see only the 32-bit Thumb-2 instructions only.

	hw1																hw2																	
	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		
Data processing: immediate, including bitfield, and saturate	1	1	1	1	0												0																	
Data processing, no immediate operand	1	1	1			1	0	1																										
Load and store single data item, memory hints	1	1	1	1	1	0	0																											
Load and Store, Double and Exclusive, and Table Branch	1	1	1	0	1	0	0			1																								
Load and Store Multiple, RFE and SRS	1	1	1	0	1	0	0			0																								
Branches, miscellaneous control	1	1	1	1	0												1																	
Coprocessor	1	1	1			1	1	1	1																									

If-Then Instruction

- Most 32-bit Thumb-2 instructions are unconditional.
 - while nearly every instruction can be made conditional in ARM ISA, Thumb-2 externalized the condition with the if-then instruction (IT) which acts like a conditional prefix to the next instruction.

IT{pattern} {cond}



```
; classic ARM
addge r0, r1, r2 ; r0 = r1 + r2 if ge condition is set

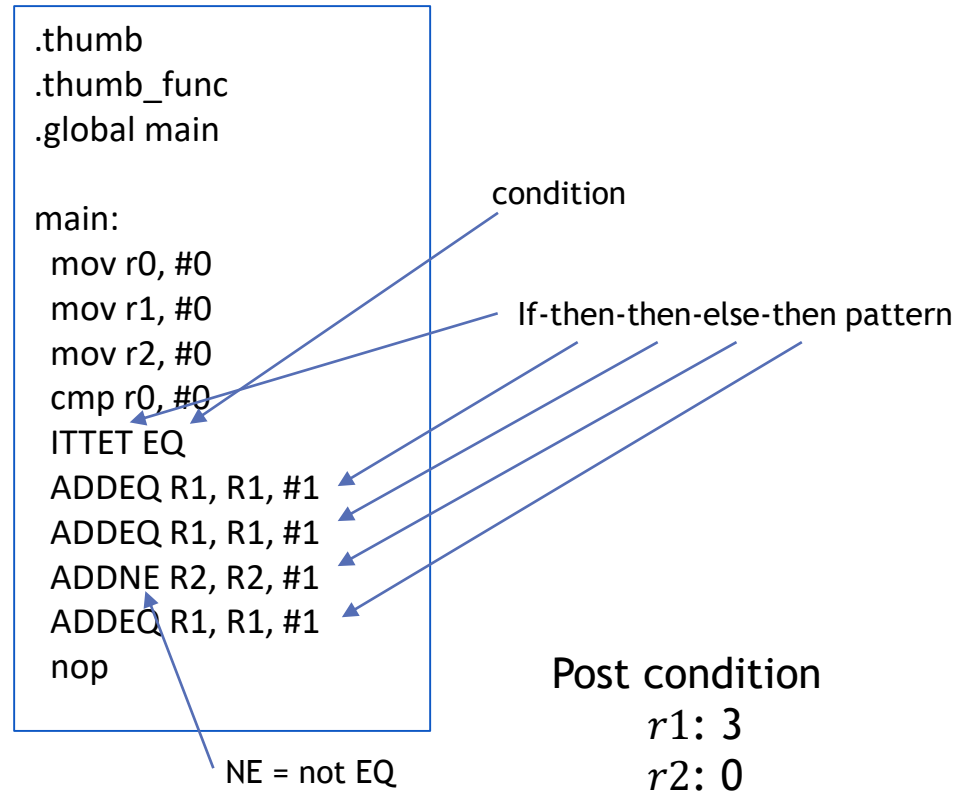
; Thumb-2
it ge          ; next instruction executes if ge
addge r0, r1, r2 ; r0 = r1 + r2
```

*ge: Greater than or Equal to

If-Then

- The if-then instruction can conditionalize up to four instructions according to pattern.
- Conditions

Condition	Meaning	Evaluation	Notes
EQ	equal	Z = 1	
NE	not equal	Z = 0	
CS	carry set	C = 1	
HS	high or same		unsigned greater than or equal
CC	carry clear	C = 0	
LO	low		unsigned less than
MI	minus	N = 1	signed negative
PL	plus	N = 0	signed positive or zero
VS	overflow set	V = 1	signed overflow
VC	overflow clear	V = 0	no signed overflow
HI	high	C = 1 and Z = 0	unsigned greater than
LS	low or same	C = 0 or Z = 1	unsigned less than or equal
GE	greater than or equal	N = V	signed greater than or equal
LT	less than	N ≠ V	signed less than
GT	greater than	Z = 0 and N = V	signed greater than
LE	less than or equal	Z = 1 or N ≠ V	signed less than
AL	always	always true	unconditional



Addressing Modes

- Register indirect
 - `ldr r0, [r1]` ; $r0 = *r1$
- Register with immediate offset
 - `ldr r0, [r1, #imm]` ; $r0 = *(r1 + imm)$
 - `ldr r0, [r1, #-imm]` ; $r0 = *(r1 - imm)$
- Register with register offset
 - `ldr r0, [r1, r2]` ; $r0 = *(r1 + r2)$
 - `ldr r0, [r1, -r2]` ; $r0 = *(r1 - r2)$

Addressing Modes

- Register with scaled register offset
 - `ldr r0, [r1, r2, LSL #2]` ; $r0 = *(r1 + (r2 \ll 2))$
 - `ldr r0, [r1, -r2, ASR #1]` ; $r0 = *(r1 - (r2 \gg 1))$ signed shift
 - Barrel Shifter operations

Mnemonic	Meaning	Range	Notes
	Do nothing		No scaling applied
<code>LSL #imm</code>	Logical shift left	$1 \leq \text{imm} \leq 31$	Shift left with zero-fill
<code>LSR #imm</code>	Logical shift right	$1 \leq \text{imm} \leq 32$	Shift right with zero-fill
<code>ASR #imm</code>	Arithmetic shift right	$1 \leq \text{imm} \leq 32$	Shift right with sign-fill
<code>ROR #imm</code>	Rotate right	$1 \leq \text{imm} \leq 31$	32-bit rotation
<code>RRX</code>	Rotate right extended	1	33-bit rotation (carry is the extra bit)

Addressing Modes

- Pre-indexed

- `ldr r0, [r1, #4]!` ; $r1 = r1 + 4$
 ; $r0 = *r1$
- `ldr r0, [r1, r2, lsl #2]!` ; $r1 = r1 + (r2 \ll 2)$
 ; $r0 = *r1$

- Post-indexed

- `ldr r0, [r1], #4` ; $r0 = *r1$
 ; $r1 = r1 + 4$
- `ldr r0, [r1], r2, lsl #2` ; $r0 = *r1$
 ; $r1 = r1 + (r2 \ll 2)$

Single-instruction Constants

- Easy case
 - `movs Rd, #imm8` ; Rd = imm8 and set some flags
- Related reading
 - 4096 could be $1 \ll 12$, or $2 \ll 11$, up to $128 \ll 5$.
 - The Thumb-2 encoding exploits this redundancy by requiring that the 8-bit value being shifted have a 1 in bit 7.
 - `mov Rd, #imm12` ; Rd = decode(imm12)
 - `movs Rd, #imm12` ; Rd = decode(imm12), set some flags
 - `mvn Rd, #imm12` ; Rd = ~decode(imm12)
 - `mvns Rd, #imm12` ; Rd = ~decode(imm12), set some flags
 - `mov Rd, #imm16` ; Rd = imm16
 - `movt Rd, #imm16` ; Rd[31:16] = imm16
; Rd[15: 0] unchanged
 - ✓ replacing the upper 16 bits of a register

Arithmetic

- The general format of three-register instructions in Thumb-2
 - `op Rd, Rn, op2` ; op2 can be `#imm12`, `Rm`, or `Rm` with a shift
- Basic arithmetic operations
 - `add Rd, Rn, op2` ; `Rd = Rn + op2`
 - `adc Rd, Rn, op2` ; `Rd = Rn + op2 + carry`
 - `sub Rd, Rn, op2` ; `Rd = Rn - op2`
 - `sbc Rd, Rn, op2` ; `Rd = Rn - op2 - !carry`
 - `rsb Rd, Rn, op2` ; `Rd = op2 - Rn`
 - `rsc Rd, Rn, op2` ; `Rd = op2 - Rn - !carry`
 - `mov Rd, #imm8` ; `Rd = imm8 (0 to 255)`
 - `mov Rd, op2` ; `Rd = op2`
 - `mvn Rd, op2` ; `rd = ~op2`
 - All support the `S` suffix.

Arithmetic

- Purpose for setting flags
 - `cmp Rn, op2` ; Set flags for $Rn - op2$
 - `cmn Rn, op2` ; Set flags for $Rn + op2$
- Multiplication ($32 \times 32 \rightarrow 32$ multiplies)
 - `mul Rd, Rn, Rm` ; $Rd = Rn * Rm$
 - `muls Rd, Rn, Rm` ; $Rd = Rn * Rm$, set partial flags
 - `mla Rd, Rm, Rs, Rn` ; $Rd = (Rm * Rs) + Rn$
 - `mls Rd, Rm, Rs, Rn` ; $Rd = Rn - (Rm * Rs)$

Arithmetic

- Multiplication ($32 \times 32 \rightarrow 64$ multiplies)
 - umull Rdlo, Rdhi, Rm, Rs ; Rdhi:Rdlo = Rm * Rs (unsigned)
 - smull Rdlo, Rdhi, Rm, Rs ; Rdhi:Rdlo = Rm * Rs (signed)
 - umlal Rdlo, Rdhi, Rm, Rs ; Rdhi:Rdlo = Rdhi:Rdlo + Rm * Rs (unsigned)
 - smlal Rdlo, Rdhi, Rm, Rs ; Rdhi:Rdlo = Rdhi:Rdlo + Rm * Rs (signed)
 - umaal Rdlo, Rdhi, Rm, Rs ; Rdhi:Rdlo = Rdhi + Rdlo + Rm * Rs (unsigned)
- Division
 - udiv Rd, Rn, Rm ; Rd = Rn / Rm (unsigned)
 - sdiv Rd, Rn, Rm ; Rd = Rn / Rm (signed)
 - mls Rr, Rq, Rm, Rn ; Rr = Rn - (Rq * Rm) = Rn % Rm

Bitwise Operations

- Bitwise operations

- and Rd, Rn, op2 ; Rd = Rn & op2
- orr Rd, Rn, op2 ; Rd = Rn | op2
- eor Rd, Rn, op2 ; Rd = Rn ^ op2
- mvn Rd, op2 ; Rd = ~op2
- bic Rd, Rn, op2 ; Rd = Rn & ~op2
- orn Rd, Rn, op2 ; Rd = Rn | ~op2
- all support the S suffix

- Bit-testing purpose

- teq Rn, op2 ; set flags for Rn ^ op2
- tst Rn, op2 ; set flags for Rn & op2

- For bitwise operations that set flags, the negative (N) and zero (Z) flags reflect the result, the carry (C) flag reflects any shifting that occurred during the calculation of op2, and the overflow (V) flag is unchanged.

Bit Shifting and Bitfield Access

- Bit shifting instructions

- `lsr Rd, Rn, #imm5` ; `Rd = Rn >> imm5` (unsigned)
- `lsr Rd, Rn, Rm` ; `Rd = Rn >> (Rm & 0xFF)` (unsigned)
- `asr Rd, Rn, #imm5` ; `Rd = Rn >> imm5` (signed)
- `asr Rd, Rn, Rm` ; `Rd = Rn >> (Rm & 0xFF)` (signed)
- `lsl Rd, Rn, #imm5` ; `Rd = Rn << imm5`
- `lsl Rd, Rn, Rm` ; `Rd = Rn << (Rm & 0xFF)`
- `ror Rd, Rn, #imm5` ; `Rd = rotate_right(Rn, imm5)`
- `ror Rd, Rn, Rm` ; `Rd = rotate_right(Rn, Rm & 0xFF)`
- `rrx Rd, Rn` ; `temp = Rn`
; `Rd = (carry << 31) | (temp >> 1)`
; `carry = temp & 1`
- all support the S suffix

Bit Shifting and Bitfield Access

- Bitfield manipulation

- `bfc Rd, #lsb, #w ; Rd[lsb+w-1:lsb] = 0`
- `bfi Rd, Rn, #lsb, #w ; Rd[lsb+w-1:lsb] = Rn[w-1:0]`
- `ubfx Rd, Rn, #lsb, #w ; Rd = Rn[lsb+w-1:lsb], zero-extended`
- `sbfx Rd, Rn, #lsb, #w ; Rd = Rn[lsb+w-1:lsb], sign-extended`

- Ex)

- ✓ Use

- `bfc r0, r0, #14, #18 ; r0 = r0 & 0x0003FFFF`

- ✓ Do not use

- `and r0, r0, #0x0003FFFF ; not a valid instruction`

- ✓ 0x0003FFFF cannot be encoded.

- There are too many 1-bits for it to be encoded as a shifted 8-bit value, and there are too many 0-bits for it to be encoded as the inverse of a shifted 8-bit value.

Memory Access and Alignment

- load and store instructions for memory access
 - LDR Rd, [...] ; load word
 - STR Rd, [...] ; store word

 - LDRD Rd, Rd2, [...] ; load doubleword into Rd and Rd2
 - LDRH Rd, [...] ; load halfword, zero-extended
 - LDRSH Rd, [...] ; load halfword, sign-extended
 - LDRB Rd, [...] ; load byte, zero-extended
 - LDRSB Rd, [...] ; load byte, sign-extended

 - STRD Rd, Rd2, [...] ; store doubleword from Rd and Rd2
 - STRH Rd, [...] ; store halfword
 - STRB Rd, [...] ; store byte

Memory Access and Alignment

- Loading and storing multiple registers
 - ; load multiple registers starting at Rn
 - ldm Rn, { registers }
 - ; load multiple registers starting at Rn, and update Rn
 - ldm Rn!, { registers }
 - ; store multiple registers ending at Rn
 - stm Rn, { registers }
 - ; store multiple registers ending at Rn, and update Rn
 - stm Rn!, { registers }
- The registers are stored with the lowest-numbered register at the lowest address, and subsequent registers in adjacent memory locations.

Memory Access and Alignment

- PUSH and POP instructions
 - ; push multiple registers
 - push { registers } ; stm sp!, { registers }
 - ; pop multiple registers
 - pop { registers } ; ldm sp!, { registers }

Control Transfer

- Direct relative branch
 - b label ; unconditional branch
 - The relative branch instruction can be conditionalized on the status flags:

Condition	Meaning	Evaluation	Notes
EQ	equal	$Z = 1$	
NE	not equal	$Z = 0$	
CS	carry set	$C = 1$	
HS	high or same		unsigned greater than or equal
CC	carry clear	$C = 0$	
LO	low		unsigned less than
MI	minus	$N = 1$	signed negative
PL	plus	$N = 0$	signed positive or zero
VS	overflow set	$V = 1$	signed overflow
VC	overflow clear	$V = 0$	no signed overflow
HI	high	$C = 1$ and $Z = 0$	unsigned greater than
LS	low or same	$C = 0$ or $Z = 1$	unsigned less than or equal
GE	greater than or equal	$N = V$	signed greater than or equal
LT	less than	$N \neq V$	signed less than
GT	greater than	$Z = 0$ and $N = V$	signed greater than
LE	less than or equal	$Z = 1$ or $N \neq V$	signed less than
AL	always	always true	unconditional

Control Transfer

- Branch with link
 - ; branch and link, stay in Thumb-2
 - bl label ; lr = next instruction + 1
 ; execution resumes at label
 - ; branch and link with exchange, switch to classic ARM
 - blx label ; lr = next instruction + 1
- Branch with exchange
 - ; branch with exchange
 - bx Rn ; switch to classic ARM if Rn is even
 ; execution resumes at Rn & ~1
 - ; branch and link with exchange
 - blx Rn ; lr = next instruction + 1
 ; switch to classic ARM if Rn is even
 ; execution resumes at Rn & ~1

Manipulating Flags

- Two instructions for accessing the flags register directly
 - `mrs Rd, apsr` ; Rd = APSR
 - `msr apsr, Rd` ; APSR = Rd
 - ✖ Thumb-2 instructions are designed to manipulate the processor's state through flags contained in the *apsr* (Application Program Status Register), instead of directly manipulating the *cpsr*.
- The format of APSR

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0									
N	Z	C	V	Q								GE[3:0]																		

- The N, Z, C, and V flags are updated by arithmetic operations.
- The GE flags are updated by SIMD operations.
- The Q flag is set when a saturating arithmetic operation overflows, and the only way to clear it is to issue an MSR instruction.

Manipulating Flags

- How to set the *aspr* indirectly
 - `cmp r0, r0` ; sets $N = 0$, $Z = 1$, $C = 1$, $V = 0$
 - `adds r0, r0, #0` ; $C = 0$
 - `teq r0, r0` ; sets $N = 0$, $Z = 1$, C and V unchanged
 - `cmp sp, #1` ; force nonzero result

Miscellaneous Instructions

- How to set the *aspr* indirectly
 - `clz Rd, Rm` ; Rd = number of leading zeroes in Rm
 - `rbit Rd, Rm` ; Rd = Rm bitwise reversed
 - `rev Rd, Rm` ; Rd = Rm byte-wise reversed
 - `rev16 Rd, Rm` ; Rd[31:24] = Rm[23:16]
; Rd[23:16] = Rm[31:24]
; Rd[15: 8] = Rm[7: 0]
; Rd[7: 0] = Rm[15: 8]
 - `revsh Rd, Rm` ; Rd[31:8] = Rm[7:0] sign extended
; Rd[7:0] = Rm[15:8]
 - `yield` ; yield to other threads
 - `wfi` ; wait for interrupt
 - `svc #imm8` ; system call
 - `bkpt #imm8` ; software breakpoint
 - `udf #imm8` ; undefined opcode
 - `Nop` ; no operation

Thumb-2 Architecture Constraints

- Almost all the functionality of the ARM ISA is covered by the Thumb ISA. Apart from the absence of a condition field, the main exceptions are:
 - ARM instructions with no Thumb-2 equivalent (for more info: [link](#))
 - ✓ RSC, SWP, SWPB
 - New functionality introduced with Thumb-2
 - ✓ BL and BLX, and all the other 32-bit Thumb instructions, can only take exceptions on their start address.
 - ✓ SDIV: Signed Divide, UDIV: Unsigned Divide.
 - 32-bit Thumb instructions with less functionality than ARM instructions.
 - ✓ Please refer to this [link](#).

References for Thumb-2

- ARM and Thumb-2 Instruction Set Reference
 - <https://developer.arm.com/documentation/qrc0001/latest/>
- ARM Architecture Reference Manual Thumb-2 Supplement
 - <https://class.ece.iastate.edu/cpre288/resources/docs/Thumb-2SupplementReferenceManual.pdf>