7. Thumb 명령어 세트

- 전체 요약

The Thumb instruction set addresses the issue of code density. It may be viewed as a compressed form of a subset of the ARM instruction set. Thumb instructions map onto ARM instructions, and the Thumb programmer's model maps onto the ARM programmer's model. Implementations of Thumb use dynamic decompression in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor.

Thumb is not a complete architecture; it is not anticipated that a processor would execute Thumb instructions without also supporting the ARM instruction set. Therefore the Thumb instruction set need only support common application functions, allowing recourse to the full ARM instruction set where necessary (for instance, all exceptions automatically enter ARM mode).

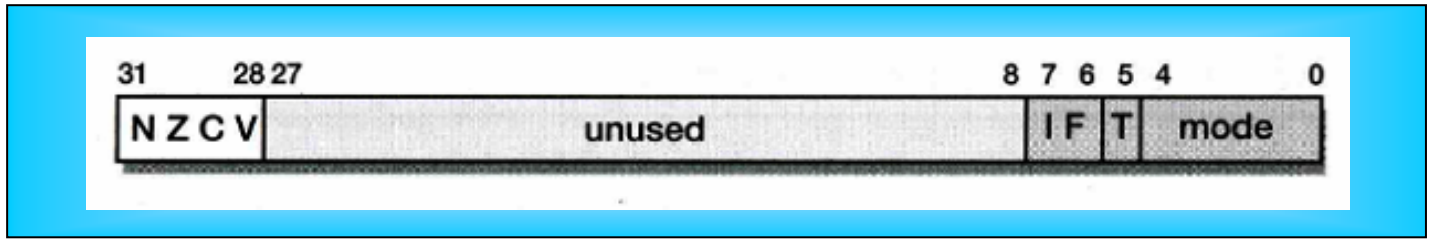
Thumb is fully supported by ARM development tools, and an application can mix ARM and Thumb subroutines flexibly to optimize performance or code density on a routine-by-routine basis.

This chapter covers the Thumb architecture and implementation, and suggests the characteristics of applications that are likely to benefit from using Thumb. In the right application, use of the Thumb instruction set can improve power-efficiency, save cost and enhance performance all at once.

7.1 CPSR 레지스터의 Thumb 비트

▷ ARM 프로세서는 32 비트 ARM 명령어와 16 비트 Thumb 명령어를 교대로 실행함 (ARM7TDMI와 같이 T가 포함된 프로세서만 Thumb 명령어를 실행)

▷ 특정 시간에서의 명령어 코드는 CPSR 레지스터의 T 비트에 의해 결정, T 비트가 1이면 프로세서는 명령어 코드를 Thumb 명령어를 간주하고 T비트가 0이면 명령어 코드를 ARM 명령어로 간주함)

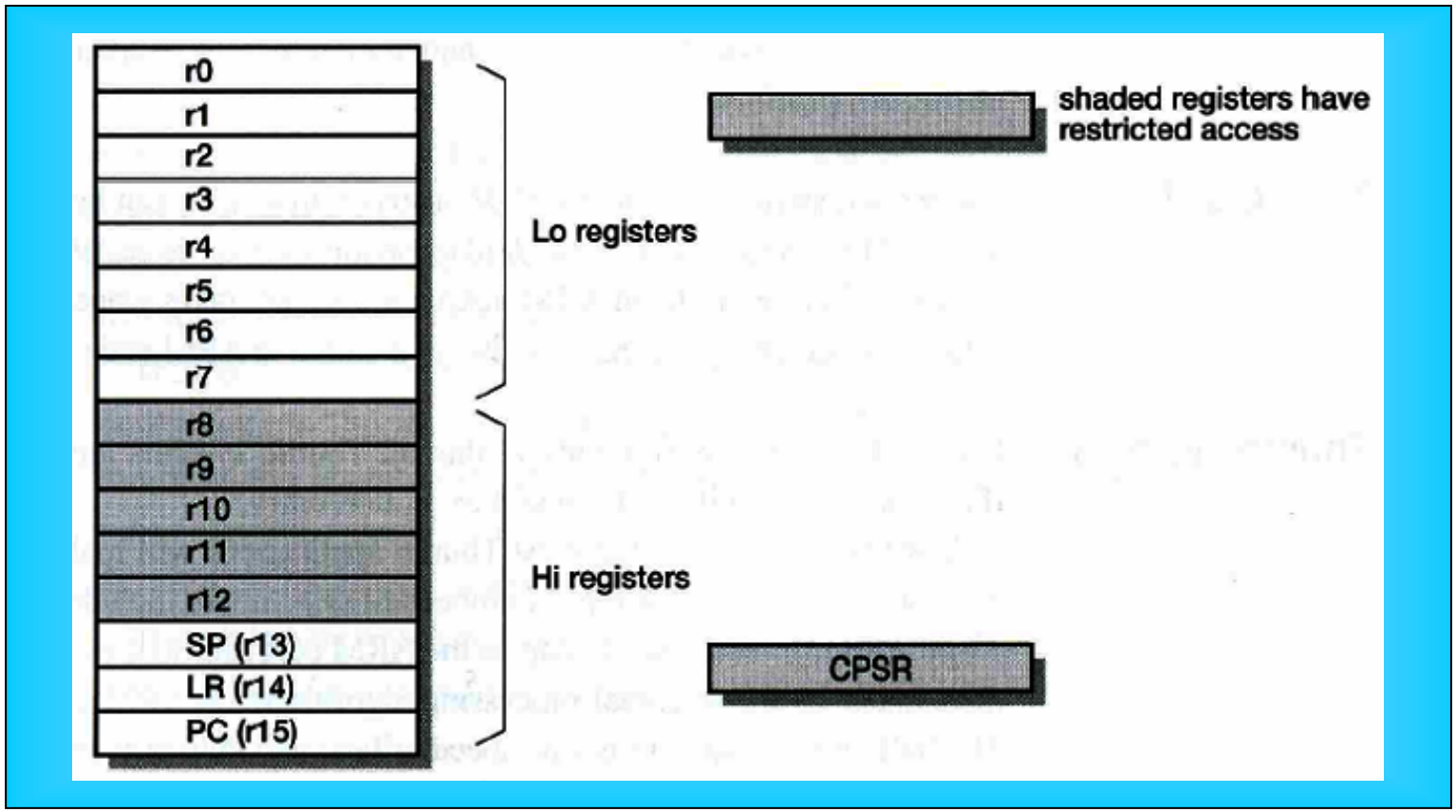


▷ ARM 명령어 실행에서 Thumb 명령어 실행으로의 전환은 BX 명령어를 사용 (오퍼랜드 레지스터의 최하위 비트가 1이면 T 비트는 1이 되고 Thumb 명령어 실행 모드로 전환됨)

▷ Thumb 명령어 실행에서 ARM 명령어 실행으로의 전환은 Thumb BX 명령어나 예외 처리에 의해 전환됨, 예외 루틴은 반드시 ARM 코드에서 처리됨

7.2 Thumb programmer's model

- Thumb 레지스터 사용



▷ 8개의 Lo 레지스터 (r0-r7)은 모든 명령어에서 사용 가능하지만, r8-r12는 특별한 경우에만 사용 가능, r13, r14, r15는 SP, LR, PC로 사용됨

- Thumb 명령어와 ARM 명령어 실행 비교

  ▷ Thumb 명령어 길이는 16 비트이고 ARM 명령어와 같이 load-store 구조를 가짐, 8 비트, 16 비트, 32 비트 데이터 형을 지원함

  ▷ ARM 명령어는 조건부로 실행할 수 있지만 대부분의 Thumb 명령어는 무조건적으로 실행됨

  ▷ ARM 데이터 처리 명령은 3 주소를 사용하지만 Thumb 데이터 처리 명령은 64 비트 곱셈을 제외하고 2 주소를 사용

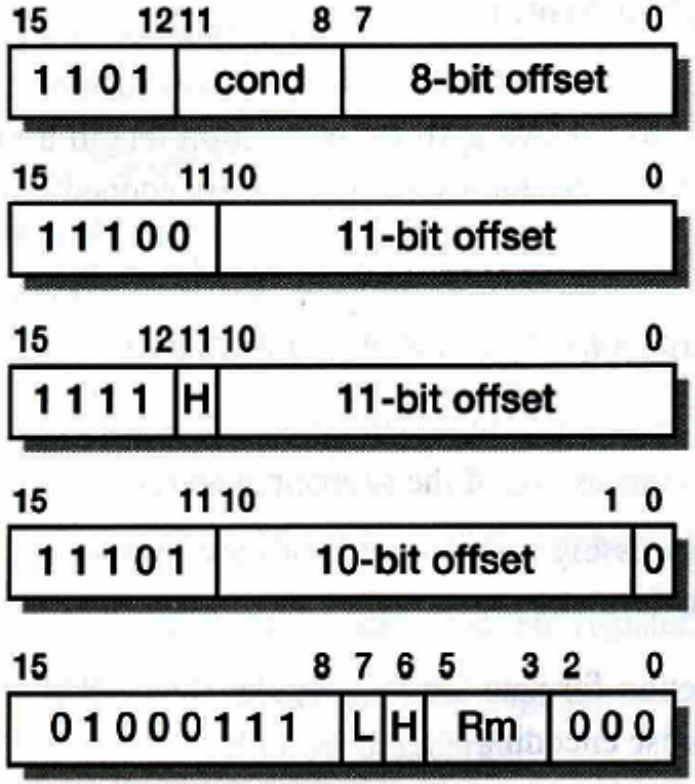
  ▷ Thumb 명령어 형식은 ARM 명령어 형식보다 덜 규칙적임

- Thumb 예외 처리

  ▷ 예외가 발생하면 ARM 실행 모드로 전환하고 ARM 프로그램 모델에서 예외를 처리함, 예외가 발생한 명령어에 따라 CPSR의 T 비트는 SPSR에 저장됨

  ▷ 예외 복귀가 Thumb 명령어이면 복귀 주소의 offset 조정이 달라져야 하지만 Thumb 구조는 ARM 복귀 offset과의 조화를 위해 자동으로 조정되기 때문에 동일한 ARM 복귀 명령어를 사용

7.3 Thumb 분기 명령어



(1) `B<cond> <label>`

```
B<cond> <label> ; format 1  - Thumb target
```

(2) `B <label>`

```
B       <label> ; format 2  - Thumb target
```

(3) `BL <label>`

```
BL      <label> ; format 3  - Thumb target
```

(3a) `BLX <label>`

```
BLX     <label> ; format 3a - ARM target
```

(4) `B{L}X Rm`

```
B{L}X   Rm      ; format 4  - ARM or Thumb target
```

1. short conditional branches to control (for example) loop exit;
2. medium-range unconditional branches to 'goto' sections of code;
3. long-range subroutine calls.

▷ Thumb 명령어 offset은 half-word이므로 1 비트 left shift한 후에 부호 확장하여 사용됨

▷ Branch and link 형식은 이동 범위가 넓어 16 비트 형식으로 표현하기 어렵기 때문에 동일한 형식의 두 개의 명령어를 사용하여 표현 (22 비트 half word offset을 사용하여 ±4M 바이트 이내의 이동 범위를 지원)

▷ 형식 3과 3a의 분기 주소와 복귀 주소의 계산

```
1. (H=0)        LR := PC + (sign-extended offset shifted left 12 places);
2. (H=1)        PC := LR + (offset shifted left 1 place);
                LR := oldPC + 3.

1. (BL, H=0)    LR := PC + (sign-extended offset shifted left 12 places);
2. (BLX)        PC := LR + (offset shifted left 1 place) & 0xfffffffc;
                LR := oldPC + 3;
                the Thumb bit is cleared.
```

▷ 형식 4의 명령어인 경우 r14에 (다음 Thumb 명령어주소+1)이 저장됨

▷ 서브루틴 호출 및 복귀

Functions that are called only from the same instruction set can use the conventional `BL` call and `MOV pc, r14` or `LDMFD sp!,{..,pc}` (in Thumb code, `POP {...,pc}`) return sequences.

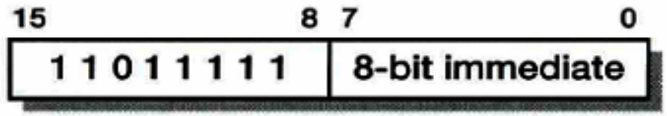
Functions that can be called from the opposite instruction set or from either instruction set can return with `BX lr` or `LDMFD sp!,{...,rN}; BX rN` (in Thumb code, `POP {...,rN}; BX rN`).

ARM processors that support architecture v5T can also return with `LDMFD sp!, {...,pc}` (in Thumb code, `POP {...,pc}`) as these instructions use the bottom bit of the loaded PC value to update the Thumb bit, but this is not supported in architectures earlier than v5T.

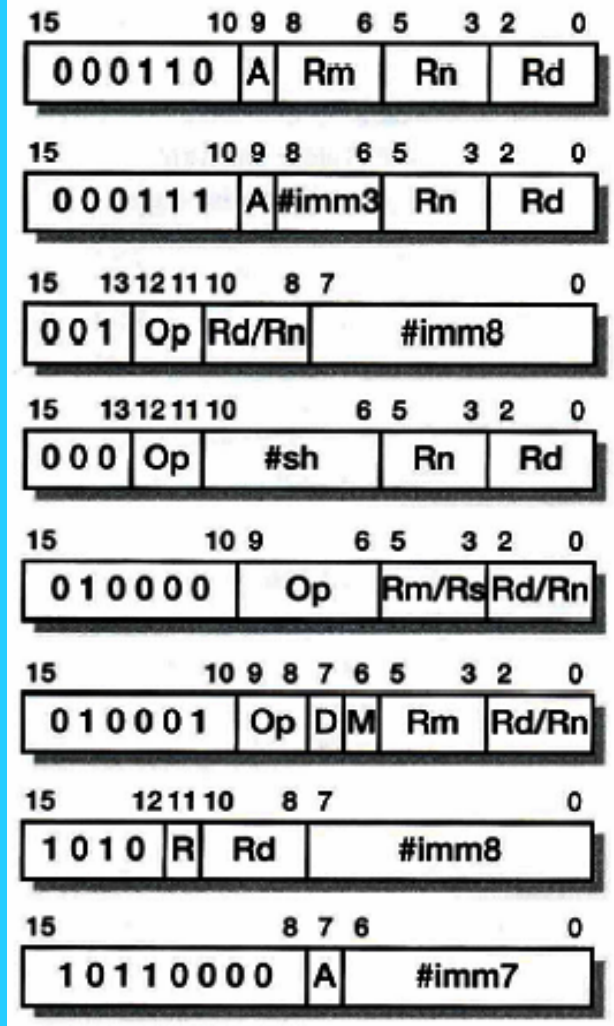
7.4 Thumb 소프트웨어 인터럽트 명령어

- The address of the next Thumb instruction is saved in r14_svc.
- The CPSR is saved in SPSR_svc.
- The processor disables IRQ, clears the Thumb bit and enters supervisor mode by modifying the relevant bits in the CPSR.
- The PC is forced to address 0x08.

```
SWI     <8-bit immediate>
```



7.5 Thumb 데이터 처리 명령어



(1) ADD|SUB Rd,Rn,Rm

```
<op>    Rd, Rn, Rm          ; <op> = ADD|SUB
```

(2) ADD|SUB Rd,Rn,#imm3

```
<op>    Rd, Rn, #<#imm3>    ; <op> = ADD|SUB
```

(3) <Op> Rd/Rn,#imm8

```
<op>    Rd|Rn, #<#imm8>     ; <op> = ADD|SUB|MOV|CMP
```

(4) LSL|LSR|ASR Rd,Rn,#shift

```
<op>    Rd, Rn, #<#sh>      ; <op> = LSL|LSR|ASR
```

(5) <Op> Rd/Rn,Rm/Rs

```
<op>    Rd|Rn, Rm|Rs        ; <op> = MVN|CMP|CMN|..
; ..TST|ADC|SBC|NEG|MUL|LSL|LSR|ASR|ROR|AND|EOR|ORR|BIC
```

(6) ADD|CMP|MOV Rd/Rn,Rm

```
<op>    Rd|Rn, Rm           ; <op> = ADD|CMP|MOV
                                     (Hi regs)
```

(7) ADD Rd,SP|PC,#imm8

```
ADD     Rd, SP|PC, #<#imm8>
```

(8) ADD|SUB SP,SP,#imm7

```
<op>    SP, SP, #<#imm7>    ; <op> = ADD|SUB
```

▷ Thumb 명령어는 ALU 동작과 자리 이동을 독립된 명령어로 분리

▷ Thumb 명령어와 등가인 ARM 명령어

ARM instruction				Thumb instruction		
MOVS	Rd,	#<#imm8>	;	MOV	Rd,	#<#imm8>
MVNS	Rd,	Rm	;	MVN	Rd,	Rm
CMP	Rn,	#<#imm8>	;	CMP	Rn,	#<#imm8>
CMP	Rn,	Rm	;	CMP	Rn,	Rm
CMN	Rn,	Rm	;	CMN	Rn,	Rm
TST	Rn,	Rm	;	TST	Rn,	Rm
ADDS	Rd,	Rn, #<#imm3>	;	ADD	Rd,	Rn, #<#imm3>
ADDS	Rd,	Rd, #<#imm8>	;	ADD	Rd,	#<#imm8>
ADDS	Rd,	Rn, Rm	;	ADD	Rd,	Rn, Rm
ADCS	Rd,	Rd, Rm	;	ADC	Rd,	Rm
SUBS	Rd,	Rn, #<#imm3>	;	SUB	Rd,	Rn, #<#imm3>
SUBS	Rd,	Rd, #<#imm8>	;	SUB	Rd,	#<#imm8>
SUBS	Rd,	Rn, Rm	;	SUB	Rd,	Rn, Rm
SBCS	Rd,	Rd, Rm	;	SBC	Rd,	Rm
RSBS	Rd,	Rn, #0	;	NEG	Rd,	Rn
MOVS	Rd,	Rm, LSL #<#sh>	;	LSL	Rd,	Rm, #<#sh>

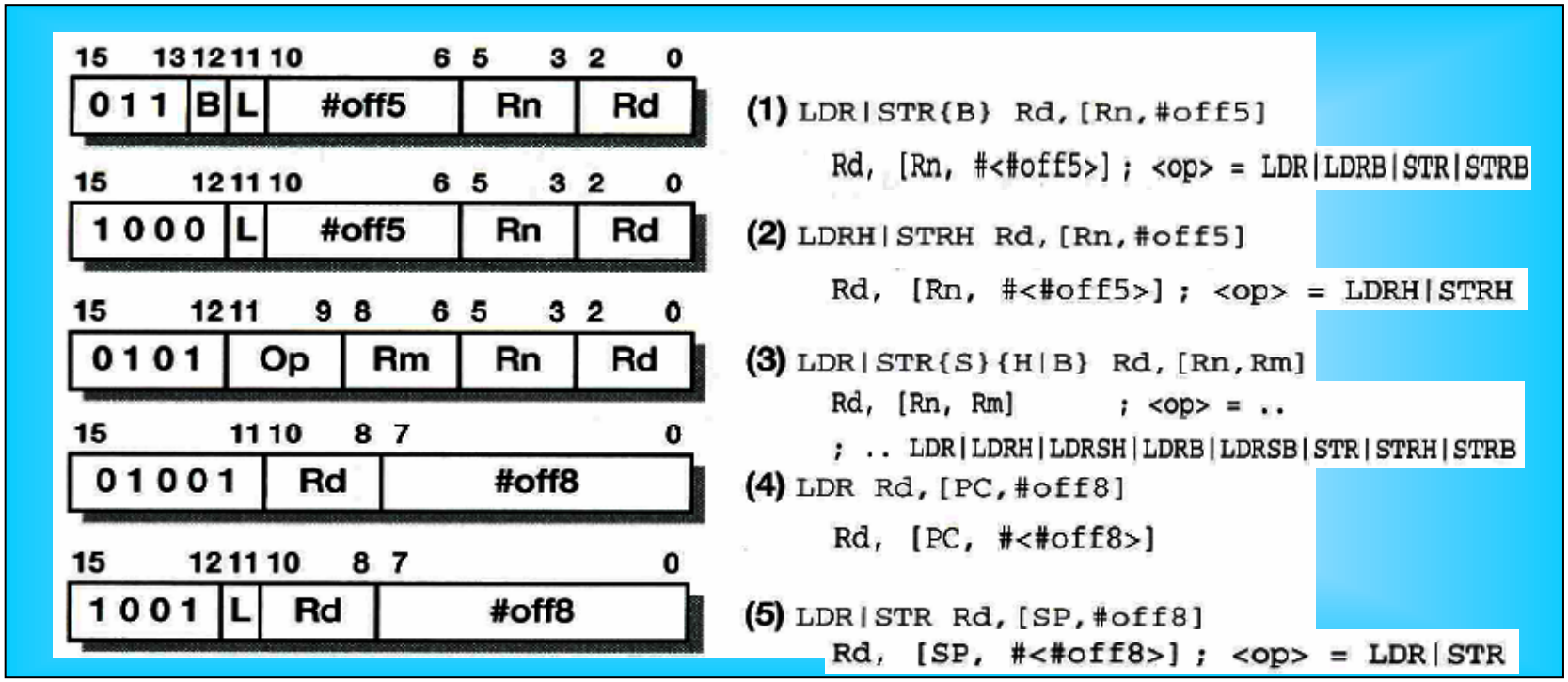
ARM instruction				Thumb instruction			
MOVS	Rd,	Rd,	LSL Rs	; LSL	Rd,	Rs	
MOVS	Rd,	Rm,	LSR #<#sh>	; LSR	Rd,	Rm,	#<#sh>
MOVS	Rd,	Rd,	LSR Rs	; LSR	Rd,	Rs	
MOVS	Rd,	Rm,	ASR #<#sh>	; ASR	Rd,	Rm,	#<#sh>
MOVS	Rd,	Rd,	ASR Rs	; ASR	Rd,	Rs	
MOVS	Rd,	Rd,	ROR Rs	; ROR	Rd,	Rs	
ANDS	Rd,	Rd,	Rm	; AND	Rd,	Rm	
EORS	Rd,	Rd,	Rm	; EOR	Rd,	Rm	
ORRS	Rd,	Rd,	Rm	; ORR	Rd,	Rm	
BICS	Rd,	Rd,	Rm	; BIC	Rd,	Rm	
MULS	Rd,	Rm,	Rd	; MUL	Rd,	Rm	

ARM instruction				Thumb instruction			
ADD	Rd,	Rd,	Rm	; ADD	Rd,	Rm (1/2 Hi regs)	
CMP	Rn,	Rm		; CMP	Rn,	Rm (1/2 Hi regs)	
MOV	Rd,	Rm		; MOV	Rd,	Rm (1/2 Hi regs)	
ADD	Rd,	PC,	#<#imm8>	; ADD	Rd,	PC,	#<#imm8>
ADD	Rd,	SP,	#<#imm8>	; ADD	Rd,	SP,	#<#imm8>
ADD	SP,	SP,	#<#imm7>	; ADD	SP,	SP,	#<#imm7>
SUB	SP,	SP,	#<#imm7>	; SUB	SP,	SP,	#<#imm7>

▷ 대부분의 Thumb 데이터 처리 명령어는 'Lo' 레지스터를 사용하고 조건 코드 비트에 영향을 주지만 'Hi' 레지스터를 사용하는 명령어는 CMP 명령을 제외하고 조건 코드 비트에 영향을 주지 않음

▷ 1 or 2 Hi regs는 'Hi' 레지스터에 속하는 1, 2개의 레지스터를 가져야 함

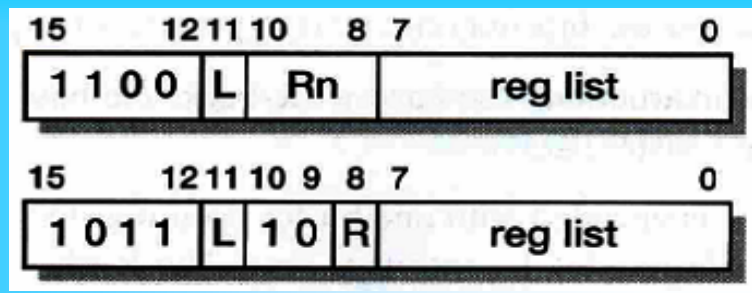
7.6 Thumb 싱글 레지스터 데이터 이동 명령어



▷ signed 오퍼랜드는 base plus 레지스터 어드레싱을 사용하고 unsigned 오퍼랜드는 base plus offset 또는 레지스터 어드레싱을 사용함

▷ 5 비트 offset은 load/store byte/half word/word 명령에서 32, 64, 128 바이트 변위를 나타냄, auto-indexing이 지원이 안됨

7.7 Thumb 다중 레지스터 데이터 이동 명령어



(1) LDMIA|STMIA Rn!, {<reg list>}

(2) POP|PUSH {<reg list>{,R}}

<reg list> is a list of registers and register ranges from r0 to r7.

```
LDMIA   Rn!, {<reg list>}
STMIA   Rn!, {<reg list>}
POP     {<reg list>{, pc}}
PUSH    {<reg list>{, lr}}
```

▷ 블록 복사는 LDMIA, STMIA 어드레싱 모드만 사용하여야 하고 베이스 레지스터와 레지스터 항목에는 'Lo' 레지스터 (r0-r7)만 사용되어야 함

▷ 스택 어드레싱은 full descending 모드만 지원되고 PUSH, POP 명령 사용

▷ write-back (auto-index)이 지원되기 때문에 베이스 레지스터가 레지스터 항목에 포함되어서는 안됨

▷ 등가적인 ARM 명령어

Block copy:

```
LDMIA   Rn!, {<reg list>}
STMIA   Rn!, {<reg list>}
```

Pop:

```
LDMFD   SP!, {<reg list>{, pc}}
```

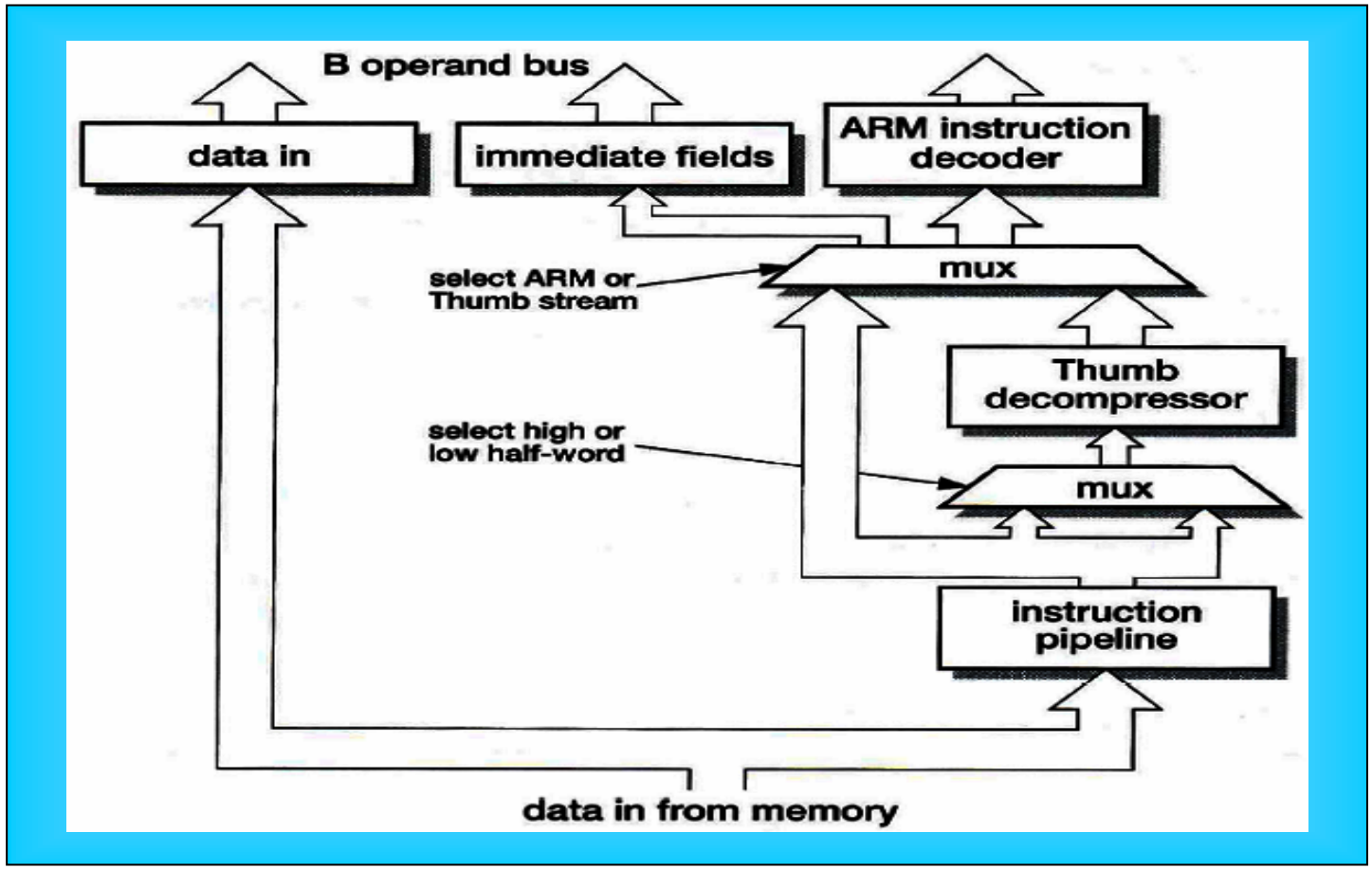
Push:

```
STMFD   SP!, {<reg list>{, lr}}
```

▷ v5T 구조에서는 복원되는 PC의 최하위 비트가 T 비트에 자동으로 update됨

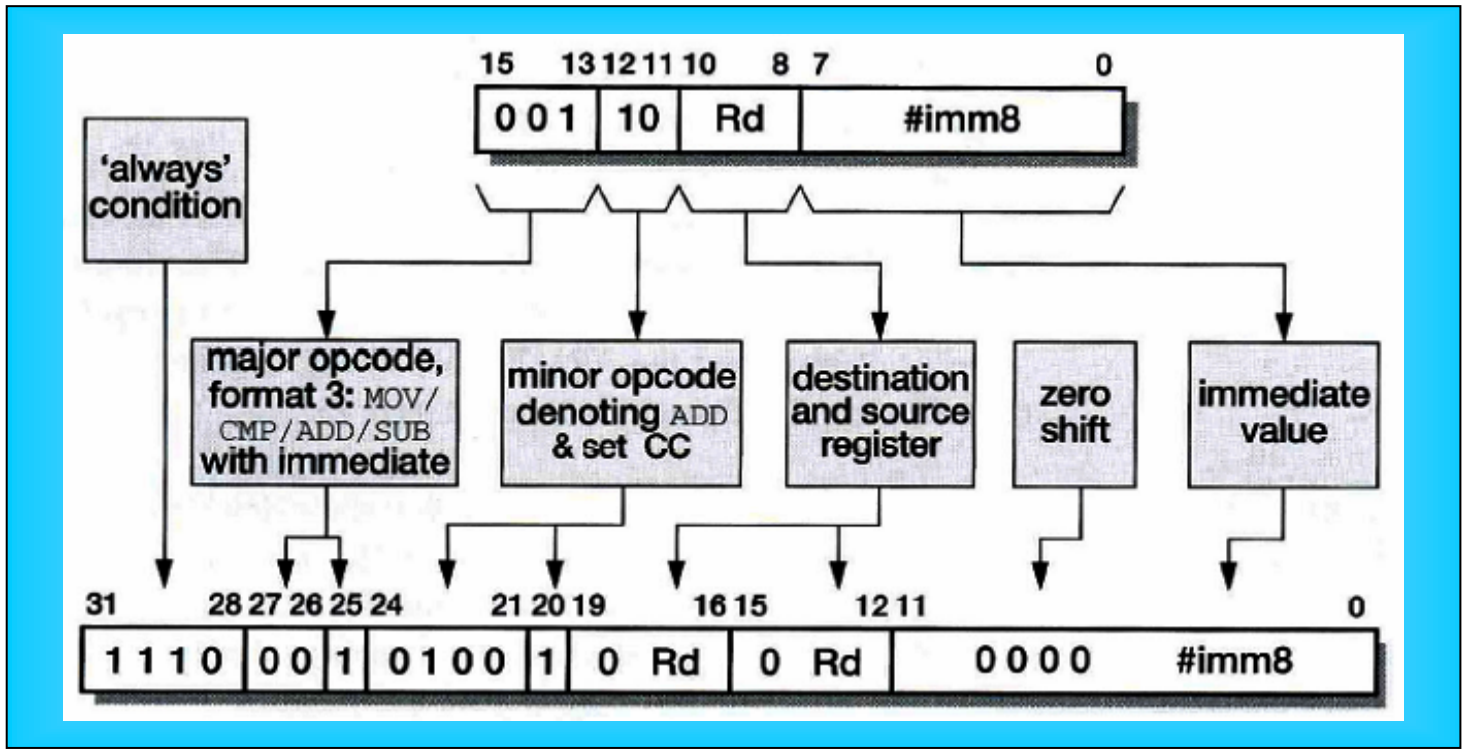
7.9 Thumb 구현

▷ Thumb 명령어 실행을 위해 파이프라인에 Thumb 명령어를 ARM 명령어로 번역해주는 decompressor가 포함되어야 함



▷ Decompressor 로직은 명령어 decoder와 직렬로 연결되고 decode 사이클의 phase 1에서 명령어를 decomressing을 할 수 있기 때문에 추가적인 파이프라인 사이클 없이 Thumb 명령어를 실행할 수 있음

▷ Thumb 명령어 ‘ADD Rd, #imm8’을 ARM 명령어 ‘ADDS Rd, Rd, #imm8’로의 mapping 과정



▷ 조건 실행 Thumb 명령어는 분기 명령어만 있으므로 ARM 명령어로 변환될 때 조건은 ‘always’로 설정됨

▷ 2 주소 형식은 3 주소 형식으로 변환되어야 함

7.10 Thumb applications

▷ Thumb 코드와 ARM 코드 비교

- The Thumb code requires 70% of the space of the ARM code.
- The Thumb code uses 40% more instructions than the ARM code.
- With 32-bit memory, the ARM code is 40% faster than the Thumb code.
- With 16-bit memory, the Thumb code is 45% faster than the ARM code.
- Thumb code uses 30% less external memory power than ARM code.

▷ 성능이 중요한 경우 32 비트 메모리 시스템과 ARM 코드를 사용하고 비용과 전력 소모가 중요한 경우 16 비트 메모리 시스템과 Thumb 코드를 사용하여야 함

7.11 Example and exercises

```
        AREA    HelloW,CODE,READONLY
SWI_WriteC      EQU     &0
SWI_Exit        EQU     &11
        ENTRY
START   ADR     r1, TEXT
LOOP    LDRB    r0, [r1], #1
        CMP     r0, #0
        SWINE   SWI_WriteC
        BNE     LOOP
        SWI     SWI_Exit
TEXT    =       "Hello World",&0a,&0d,0
        END
```

ARM 코드 : 명령어 6개 + 데이터 14 바이트

= 6*4 + 14 = 38 바이트

Thumb 코드 : 명령어 8개 + 데이터 14 바이트

= 8*2 + 14 = 30 바이트

```
        AREA    HelloW_Thumb,CODE,READONLY
SWI_WriteC      EQU     &0
SWI_Exit        EQU     &11
        ENTRY
        CODE32
        ADR     r0, START+1
        BX      r0
        CODE16
START   ADR     r1, TEXT
LOOP    LDRB    r0, [r1]
        ADD     r1, r1, #1
        CMP     r0, #0
        BEQ     DONE
        SWI     SWI_WriteC
        B       LOOP
DONE    SWI     SWI_Exit
        ALIGN
TEXT    DATA
        =       "Hello World",&0a,&0d,&00
        END
```