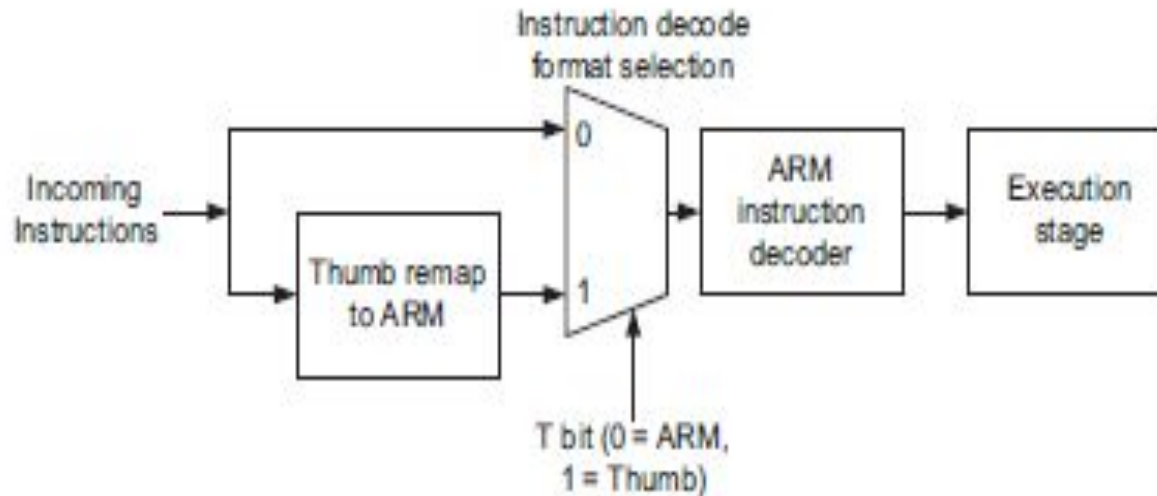


UNIT III

Instruction Sets

Background of ARM and Thumb Instruction Set



Unit-III-Instruction Set

Background of ARM and Thumb Instruction Set

- The early ARM processors use a 32-bit instruction set called the ARM instructions.
- The 32-bit ARM instruction set is powerful and provides good performance,
- but at the same time it often requires larger program memory when compared to 8-bit and 16-bit processors.
- This was and still is an issue, as memory is expensive and could consume a considerable amount of power.

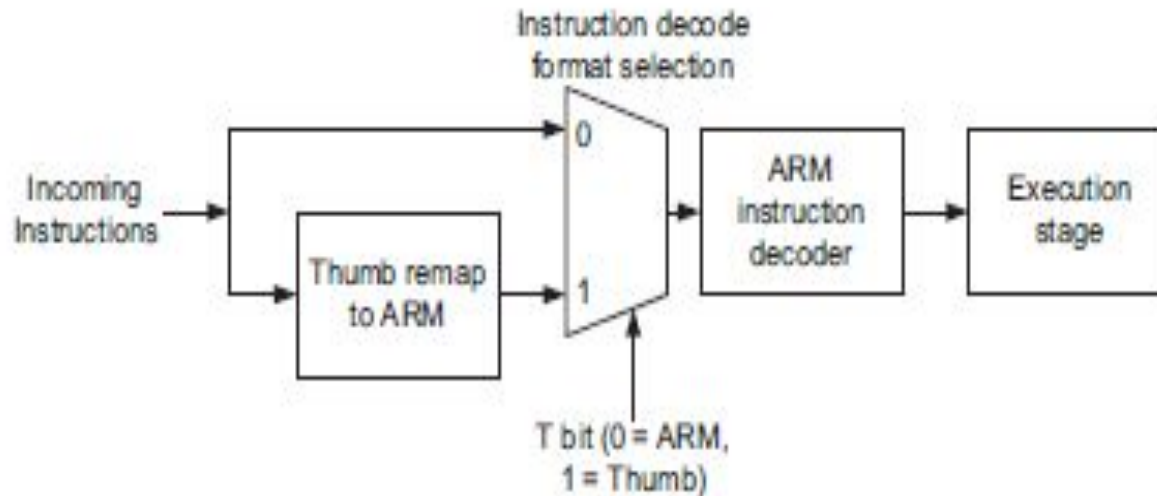
Background of ARM and Thumb Instruction Set

- In 1995, ARM introduced the ARM7TDMI processor, adding a new 16-bit instruction set called the Thumb instruction set.
- The ARM7TDMI supports both ARM instructions and Thumb instructions, and a state-switching mechanism is used to allow the processor to decide which instruction decode scheme should be used (Figure 5.1).

Background of ARM and Thumb Instruction Set

- The Thumb instruction set provides a subset of the ARM instructions.
- By itself it can perform most of the normal functions, but interrupt entry sequence and boot code must still be in ARM state.
- Nevertheless, most processing can be carried out using Thumb instructions and interrupt handlers could switch themselves to use the Thumb state, so the ARM7TDMI processor provides excellent code density when compared to other 32-bit RISC architectures.

Background of ARM and Thumb Instruction Set



Thumb Code

- Thumb code provides a code size reduction of approximately 30% compared to the equivalent ARM code.
- However, it has some impact on the performance and can reduce the performance by 20%.
- On the other hand, in many applications, the reduction of program memory size and the low-power nature of the ARM7TDMI processor made it extremely popular with portable electronic devices like mobile phones and microcontrollers.

Thumb Code and Unified Assembler Language (UAL)

- Traditionally, programming of the ARM processors in Thumb state is done with the Thumb Assembly syntax.
- To allow better portability between architectures and to use a single assembly language syntax between different ARM processors with various architectures, recent ARM development tools have been updated to support the Unified Assembler Language (UAL).
- For users who have used ARM7TDMI in the past, the most noticeable differences are the following:

Background of ARM and Thumb

Thumb Code and Unified Assembler Language (UAL)

Traditionally, programming of the ARM processors in Thumb state is done with the Thumb Assembly syntax. To allow better portability between architectures and to use a single assembly language syntax between different ARM processors with various architectures, recent ARM development tools have been updated to support the Unified Assembler Language (UAL). For users who have used ARM7TDMI in the past, the most noticeable differences are the following:

- Some data operation instructions use three operands even when the destination register is the same as one of the source registers. In the past (pre-UAL), syntax might only use two operands for the same instructions.
- The “S” suffix becomes more explicit. In the past, when an assembly program file was assembled into Thumb code, most data operations were implied as instructions that updated the APSR; as a result, the “S” suffix was not essential. With the UAL syntax, instructions that update the APSR should have the “S” suffix to clearly indicate the expected operation. This prevents program code from failing when being ported from one architecture to another.

For example, a pre-UAL ADD instruction for 16-bit Thumb code is

```
ADD    R0, R1    ; R0 = R0 + R1, update APSR
```

With UAL syntax, this should be written as

```
ADDS   R0, R0, R1    ; R0 = R0 + R1, update APSR
```

Unified Assembly Language(UAL)

- Some data operation instructions use three operands even when the destination register is the same as one of the source registers.
- In the past (pre-UAL), syntax might only use two operands for the same instructions.
- The “S” suffix becomes more explicit. In the past, when an assembly program file was assembled into Thumb code, most data operations were implied as instructions that updated the APSR; as a result, the “S” suffix was not essential.
- With the UAL syntax, instructions that update the APSR should have the “S” suffix to clearly indicate the expected operation.
- This prevents program code from failing when being ported from one architecture to another.

UAL

- For example, a pre-UAL ADD instruction for 16-bit Thumb code is
- `ADD R0, R1 ; R0 = R0 + R1, update APSR`
- With UAL syntax, this should be written as
- `ADDS R0, R0, R1 ; R0 = R0 + R1, update APSR`

16/32 bit Thumb Instructions

Table 5.1: 16-Bit Thumb Instructions Supported on the Cortex-M0 Processor

16-Bit Thumb Instructions Supported on Cortex-M0									
ADC	ADD	ADR	AND	ASR	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EOR	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSL	LSR	MOV	MVN	MUL	NOP	ORR	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBC	SEV	STM	STR	STRH	STRB
SUB	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

The Cortex-M0 processor also supports a number of 32-bit Thumb instructions from Thumb-2 technology (Table 5.2):

- MRS and MSR special register access instructions
- ISB, DSB, and DMB memory synchronization instructions
- BL instruction (BL was supported in traditional Thumb instruction set, but the bit field definition was extended in Thumb-2)

Table 5.2: 32-Bit Thumb Instructions Supported on the Cortex-M0 Processor

32-Bit Thumb Instructions Supported on Cortex-M0					
BL	DSB	DMB	ISB	MRS	MSR

Cortex –M0 processor supports 32-bit Thumb Instructions.

Instruction		DMB
Function	Data Memory Barrier	
Syntax	DMB	
Note	Ensures that all memory accesses are completed before new memory access is committed	

Instruction		DSB
Function	Data Synchronization Barrier	
Syntax	DSB	
Note	Ensures that all memory accesses are completed before the next instruction is executed	

Instruction		ISB
Function	Instruction Synchronization Barrier	
Syntax	ISB	
Note	Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions	

Commonly used Directives to insert data into the program and Suffixes

Table 5.3: Commonly Used Directives for Inserting Data into a Program

Type of Data to Insert	ARM Assembler	GNU Assembler
Word	DCD (e.g., DCD 0x12345678)	.word / .4byte (e.g., .word 0x012345678)
Half word	DCW (e.g., DCW 0x1234)	.hword / .2byte (e.g., .hword 0x01234)
Byte	DCB (e.g., DCB 0x12)	.byte (e.g., .byte 0x012)
String	DCB (e.g., TXT DCB "Hello\n", 0)	.ascii / .asciz (with NULL termination) (e.g., .ascii "Hello\n") .byte 0 /* add NULL character */ (e.g., .asciz "Hello\n")
Instruction	DCI (e.g., DCI 0xBE00 ; Breakpoint-BKPT 0)	.word / .hword (e.g., .hword 0xBE00 /* Breakpoint (BKPT 0) */)

Use of a Suffix

In the assembler for ARM processors, some instructions can be followed by suffixes. For Cortex-M0, the available suffixes are shown in Table 5.4.

Table 5.4: Suffixes for Cortex-M0 Assembly Language

Suffix	Descriptions
S	Update APSR (flags); for example, <code>ADDS R0, R1</code> ; this ADD operation will update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Conditional execution. EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. On the Cortex-M0 processor, these conditions can only be applied to conditional branches. For example, <code>BEQ label</code> ; branch to label if equal

Conditional Suffixes

Table 5.7: Condition Suffixes for Conditional Branch

Suffix	Branch Condition	Flags (APSR)
EQ	Equal	Z flag is set
NE	Not equal	Z flag is cleared
CS/HS	Carry set / unsigned higher or same	C flag is set
CC/LO	Carry clear / unsigned lower	C flag is cleared
MI	Minus / negative	N flag is set (minus)
PL	Plus / positive or zero	N flag is cleared
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared ($N == V$)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set ($N != V$)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared ($Z == 0$ and $N == V$)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set ($Z == 1$ or $N != V$)

Instruction List

The instructions in the Cortex-M0 processor can be divided into various groups based on functionality:

- Moving data within the processor
- Memory accesses
- Stack memory accesses
- Arithmetic operations
- Logic operations
- Shift and rotate operations
- Extend and reverse ordering operations
- Program flow control (branch, conditional branch, and function calls)
- Memory barrier instructions
- Exception-related instructions
- Other functions

Instruction List

The instructions in the Cortex-M0 processor can be divided into various groups based on functionality:

- Moving data within the processor
- MOV (move) instruction can be used to transfer data between two registers, or it can be used to put an immediate constant value into a register.

Instruction List

The instructions in the Cortex-M0 processor can be divided into various groups based on functionality:

- `MOVS R0, #0x12` ; Set R0 = 0x12 (hexadecimal)
- `MOVS R1, #'A'` ; Set R1 = ASCII character A

Moving Data within the Processor

Instruction	MOV
Function	Move register into register
Syntax (UAL)	MOV <Rd>, <Rm>
Syntax (Thumb)	MOV <Rd>, <Rm> CPY <Rd>, <Rm>
Note	Rm and Rn can be high or low registers CPY is a pre-UAL synonym for MOV (register)

Instruction	MOVS/ADDS
Function	Move register into register
Syntax (UAL)	MOVS <Rd>, <Rm> ADDS <Rd>, <Rm>, #0
Syntax (Thumb)	MOVS <Rd>, <Rm>
Note	Rm and Rn are both low registers APSR.Z, APSR.N, and APSR.C (for ADDS) update

Instruction	MOV
Function	Move immediate data (sign extended) into register
Syntax (UAL)	MOVS <Rd>, #immed8
Syntax (Thumb)	MOV <Rd>, #immed8
Note	Immediate data range 0 to +255 APSR.Z and APSR.N update

Register	Value
Core	
R0	0x0000001A
R1	0x0000001A
R2	0x00000061
R3	0x00000041
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001D0
xPSR	0x01000000
Banked	
System	
Internal	
Mode	Thread

22: stop B stop

⇒ 0x000001D0 E7FE B 0x000001D0

__aeabi_uidiv:

0x000001D2 2200 MOVS r2, #0x00

HALFWORD.asm

startup_NUC1xx.s

system_NUC1xx.c

basic Instructions.asm

```

4          AREA |.text|, CODE, READONLY
5
6          EXPORT __main
7          ; Start of CODE area
8          __main
9
10         MOVS r0, #0x12
11         MOVS r1, r0 ;Transferring from source to destination with in the processor
12         MOVS r2, # 'a' ;Ascii
13         MOVS r3, # 'A' ;Ascii
14         ADDS r0, r0, #0x08; updation
15         CPY r1, r0
16
17
18
19
20

```

DCD Inserting the word

- PRESERVE8 ; Indicate the code here preserve
- ; 8 byte stack alignment
- THUMB ; Indicate THUMB code is used
- AREA |.text|, CODE, READONLY
-
- EXPORT __main
- ; Start of CODE area
- __main
- LDR R3,=MY_NUMBER ; Get the memory location of MY_NUMBER
- LDR R4, [R3] ; Read the value 0x12345678 into R4
-
- ;LDR R0,=HELLO_TEXT ; Get the starting address of HELLO_TEXT
-

DCD Inserting the word

- ;BL PrintText ; Call a function called PrintText to
- ; display string
- ALIGN 4
- MY_NUMBER DCD 0x1234567
- ;HELLO_TEXT DCB “Hello\n”, 0 ; Null terminated string
- stop B stop
- END

Move between Special Registers and Registers

Instruction	MRS
Function	Move Special Register into register
Syntax	MRS <Rd>, <SpecialReg>
Note	Example: MRS R0, CONTROL; Read CONTROL register into R0 MRS R9, PRIMASK; Read PRIMASK register into R9 MRS R3, xPSR; Read xPSR register into R3

Table 5.5: Special Register Symbols for MRS and MSR Instructions

Symbol	Register	Access Type
APSR	Application Program Status Register (PSR)	Read/Write
EPSR	Execution PSR	Read only
IPSR	Interrupt PSR	Read only
IAPSR	Composition of IPSR and APSR	Read only
EAPSR	Composition of EPSR and APSR	Read only
IEPSR	Composition of IPSR and EPSR	Read only
XPSR	Composition of APSR, EPSR, and IPSR	Read only
MSP	Main stack pointer	Read/Write
PSP	Process stack pointer	Read/Write
PRIMASK	Primary exception mask register	Read/Write
CONTROL	CONTROL register	Read/Write in Thread mode Read only in Handler mode

Instruction	MSR
Function	Move register into Special Register
Syntax	MSR <SpecialReg>, <Rd>
Note	Example: MSR CONTROL, R0; Write R0 into CONTROL register MSR PRIMASK, R9; Write R9 into PRIMASK register

SUB and ADC

Instruction	ADC
Function	Add with carry and update APSR
Syntax (UAL)	ADCS <Rd>, <Rm>
Syntax (Thumb)	ADC <Rd>, <Rm>
Note	$Rd = Rd + Rm + \text{Carry}$ Rd and Rm are low registers.

Instruction	SUB
Function	Subtract two registers
Syntax (UAL)	SUBS <Rd>, <Rn>, <Rm>
Syntax (Thumb)	SUB <Rd>, <Rn>, <Rm>
Note	$Rd = Rn - Rm$, APSR update. Rd, Rn, Rm are low registers.

Instruction	SUB
Function	Subtract a register with an immediate constant
Syntax (UAL)	SUBS <Rd>, <Rn>, #immed3 SUBS <Rd>, #immed8
Syntax (Thumb)	SUB <Rd>, <Rn>, #immed3 SUB <Rd>, #immed8
Note	$Rd = Rn - \text{ZeroExtend}(\#immed3)$, APSR update, or $Rd = Rd - \text{ZeroExtend}(\#immed8)$, APSR update. Rd, Rn are low registers.

Move between Special Registers and Registers

The screenshot displays a debugger interface with a register list on the left and an assembly window on the right.

Register List:

Register	Value
R0	0x70000000
R1	0xE0000000
R2	0x90000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R...	0x00000000
R...	0x00000000
R...	0x00000000
R...	0x20000420
R...	0x000000DD
R...	0x000001CC
x...	0x91000000
Bank...	
System	
Internal	
M Thread	

Assembly Window:

13: stop B stop

0x000001CC E7FE B 0x000001CC

0x000001CE 0000 MOVS r0,r0

0x000001D0 0000 MOVS r0,r0

HALFWORD.asm startup_NUC1xx.s system_NUC1xx.c basicinstruction2.asm

```
1 PRESERVE8 ; Indicate the code here preserve
2 ; 8 byte stack alignment
3 THUMB ; Indicate THUMB code is used
4 AREA |.text|, CODE, READONLY
5
6 EXPORT __main
7 ; Start of CODE area
8 __main
9 ldr r0,=0x70000000
10 ADDS r1,r0,r0
11 MRS r2,APSR
12
13 stop B stop
14 END
```

Move between Special Registers and Registers, ADCS and SBCS

The screenshot displays a debugger interface with a register list on the left and an assembly window on the right.

Register List (Left):

Register	Value
R0	0x20000000
R1	0xE0000000
R2	0x90000000
R3	0x20000000
R4	0x30000000
R5	0x00000234
R6	0x00001000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R...	0x00000000
R...	0x00000000
R...	0x00000000
R...	0x20000420
R...	0x000000DD
R...	0x000001DE
x...	0x21000000

Assembly Window (Right):

Files: HALFWORD.asm, startup_NUC1xx.s, system_NUC1xx.c, basicinstruction2.asm, basicInstructions.

```
3          THUMB          ; Indicate THUMB code is used
4          AREA          |.text|, CODE, READONLY
5
6          EXPORT  __main
7          ; Start of CODE area
8          __main
9          ldr r0,=0x70000000
10         ADDS r1,r0,r0
11         MRS r2,APSR
12         ldr r0,=0x90000000
13         ADCS r0,r0,r0;source and destination should be same
14         MRS r4,APSR
15         ldr r5,=0x00001234
16         ldr r6,=0x00001000
17         SBCS r5,r5,r6;source and destination should be same
18         MRS r3,APSR
19
```

Disassembly View (Top Right):

Address	Instruction	Comment
0x000001DE	E7FE	B 0x000001DE
0x000001E0	0000	MOVS r0,r0
0x000001E2	7000	STRB r0,[r0,#0x001]

Memory Access Instructions

Table 5.6: Memory Access Instructions for Various Transfer Sizes

Transfer Size	Unsigned Load	Signed Load	Signed/Unsigned Store
Word	LDR	LDR	STR
Half word	LDRH	LDRSH	STRH
Byte	LDRB	LDRSB	STRB

For memory read operations, the instruction to carry out single accesses is LDR (load):

Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, <Rm>] ; Word read LDRH <Rt>, [<Rn>, <Rm>] ; Half Word read LDRB <Rt>, [<Rn>, <Rm>] ; Byte read
Note	Rt = memory[Rn + Rm] Rt, Rn and Rm are low registers

The Cortex-M0 processor also supports immediate offset addressing modes:

Instruction	LDR/LDRH/LDRB
Function	Read single memory data into register
Syntax	LDR <Rt>, [<Rn>, #immed5] ; Word read LDRH <Rt>, [<Rn>, #immed5] ; Half Word read LDRB <Rt>, [<Rn>, #immed5] ; Byte read
Note	Rt = memory[Rn + ZeroExtend(#immed5 << 2)] ; Word Rt = memory[Rn + ZeroExtend(#immed5 << 1)] ; Half word Rt = memory[Rn + ZeroExtend(#immed5)] ; Byte Rt and Rn are low registers

Memory Access Instructions

__main

```
LDR r0,=0x20000000 ; Source address
LDR r1,=0x20000100 ; Destination address
LDR r2, =10 ; number of bytes to copy
LDR r4,=0x20000050 ;
```

copy_loop

```
LDRB r3, [r0] ; read 1 byte
ADDS r0, r0, #1 ; increment source pointer
STRB r3, [r1] ; write 1 byte
ADDS r1, r1, #1 ; increment destination pointer
SUBS r2, r2, #1 ; decrement loop counter
BNE copy_loop ; loop until all data copied
LDR r2, =10 ; number of bytes to copy
LDR r0,=0x20000000 ; Source address
LDR r2, =10 ; number of bytes to copy
LDR r4,=0x20000050 ;
```

Memory Access Instructions

copy_loop1

LDRH r3, [r0] ; read 1 byte

ADDS r0, r0, #2 ; increment source pointer

STRH r3, [r4] ; write 1 byte

ADDS r4, r4, #2 ; increment destination

pointer

SUBS r2, r2, #2 ; decrement loop counter

BNE copy_loop1 ; loop until all data

copied

stop B stop

END

Memory Access Instructions (LDRH)

Core	
R0	0x20000002
R1	0x2000010A
R2	0x0000000A
R3	0x00000101
R4	0x20000050
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001E6
+ xPSR	0x01000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	155

The screenshot shows the Keil uVision IDE with the assembly code for the 'copy' function. The code is as follows:

```

9  __main
10
11
12      LDR r0,=0x20000000 ; Source address
13      LDR r1,=0x20000100 ; Destination address
14      LDR r2, =10 ; number of bytes to copy
15      LDR r4,=0x20000050 ;
16  copy_loop
17      LDRB r3, [r0] ; read 1 byte
18      ADDS r0, r0, #1 ; increment source pointer
19      STRB r3, [r1] ; write 1 byte
20      ADDS r1, r1, #1 ; increment destination pointer
21      SUBS r2, r2, #1 ; decrement loop counter
22      BNE copy_loop ; loop until all data copied
23      LDR r2, =10 ; number of bytes to copy
24      LDR r0,=0x20000000 ; Source address
25      LDR r2, =10 ; number of bytes to copy
26      LDR r4,=0x20000050 ;
  
```

The 'Memory 3' window is open, showing the memory address 0x20000050. The memory content is displayed in hexadecimal and ASCII format. The first row shows 0x20000050: 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00. The second row shows 0x2000005F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00. The third row shows 0x2000006E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00. The fourth row shows 0x2000007D: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00. The fifth row shows 0x2000008C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00.

Memory Access Instructions (LDRB)

R0	0x20000005
R1	0x20000105
R2	0x00000005
R3	0x00000003
R4	0x20000050
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x00001D6
xPSR	0x21000000
+ Banked	
+ System	
+ Internal	
Mode	Thread
Stack	MSP
States	91
Sec	0.00000758

The screenshot shows the Keil uVision IDE interface. The main editor displays the assembly file "copy_50_bytes.asm". At the top, there are tabs for various files: "copy_50_bytes.asm", "LDR.asm", "basicinstruction2.asm", "ADR.asm", "ASR.asm", "Illustrationof ANDORMVN.asm", "system_NUC1xx.c", and "startup_NUC...". The assembly code includes directives like PRESERVE8, THUMB, AREA, EXPORT __main, and comments describing stack alignment and loop logic. A memory dump window titled "Memory 3" is open on the right side, displaying hexadecimal addresses from 0x20000100 to 0x2000013C along with their corresponding byte values in a grid format.

```
22:      BNE copy_loop ; loop until all data copied  
→ 0x000001D6 D1F9      RNF       0x000001CC
```

File Explorer:
copy_50_bytes.asm, LDR.asm, basicinstruction2.asm, ADR.asm, ASR.asm, Illustrationof ANDORMVN.asm, system_NUC1xx.c, startup_NUC...

Assembly Code:

```
1    PRESERVE8 ; Indicate the code here preserve  
2    ; 8 byte stack alignment  
3          THUMB        ; Indicate THUMB code is used  
4    AREA |.text|, CODE, READONLY  
5  
6    EXPORT __main  
7    ; Start of CODE area  
8  
9    __main  
10  
11  
12           LDR r0,=0x20000000 ; Source address  
13     LDR r1,=0x20000100 ; Destination address  
14     LDR r2, =10 ; number of bytes to copy  
15     LDR r4,=0x20000050 ;  
16   copy_loop  
17     LDRB r3, [r0] ; read 1 byte  
18     ADDS r0, r0, #1 ; increment source pointer  
19     STRB r3, [r1] ; write 1 byte  
20     ADDS r1, r1, #1 ; increment destination pointer  
21     SUBS r2, r2, #1 ; decrement loop counter  
→ 22     BNE copy_loop ; loop until all data copied  
23     LDR r2, =10 ; number of bytes to copy  
24     LDR r0,=0x20000000 ; Source address  
25     LDR r2, =10 ; number of bytes to copy  
26     LDR r4,=0x20000050 ;  
27   copy_loop1
```

Memory Window (Memory 3):

Address
0x20000100: 01 01 02 02 03 00 00 00 00 00 00 00 00 00 00 00
0x2000010F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000011E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000012D: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000013C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Navigation: Memory 2, **Memory 3**

ADR,ADC(difference)

Instruction	ADR (ADD)
Function	Add an immediate constant with PC to a register without updating APSR
Syntax (UAL)	ADR <Rd>, <label> (normal syntax) ADD <Rd>, PC, #immed8 (alternate syntax)
Syntax (Thumb)	ADR <Rd>, (normal syntax) ADD <Rd>, PC, #immed8 (alternate syntax)
Note	$Rd = (PC[31:2] \ll 2) + \text{ZeroExtend}(\#immed8 \ll 2)$. This instruction is useful for locating a data address within the program memory near to the current instruction. The result address must be word aligned. Rd is a low register.

Instruction	ADC
Function	Add with carry and update APSR
Syntax (UAL)	ADCS <Rd>, <Rm>
Syntax (Thumb)	ADC <Rd>, <Rm>
Note	$Rd = Rd + Rm + \text{Carry}$ Rd and Rm are low registers.

Move between Special Registers and Registers

- $0x90000000 + 0x90000000$ Result = $0x30000000$, $N = 0$, $Z = 0$, $C = 1$, $V = 1$
- $0x00001234 - 0x00001000$ Result $\frac{1}{4}$ $0x00000234$, $N = 0$, $Z = 0$, $C = 1$, $V = 0$

SUB, SBC, RSB, MULS

Instruction SUB	
Function	Subtract SP by an immediate constant
Syntax (UAL)	SUB SP, SP, #immed7
Syntax (Thumb)	SUB SP, #immed7
Note	SP = SP - ZeroExtend(#immed7 << 2). This instruction is useful for C functions to adjust the SP for local variables.

Instruction SBC	
Function	Subtract with carry (borrow)
Syntax (UAL)	SBCS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	SBC <Rd>, <Rm>
Note	Rd = Rd - Rm - Borrow, APSR update. Rd and Rm are low registers.

Instruction RSB	
Function	Reverse Subtract (negative)
Syntax (UAL)	RSBS <Rd>, <Rn>, #0
Syntax (Thumb)	NEG <Rd>, <Rn>
Note	Rd = 0 - Rm, APSR update. Rd and Rm are low registers.

Instruction MUL	
Function	Multiply
Syntax (UAL)	MULS <Rd>, <Rm>, <Rd>
Syntax (Thumb)	MUL <Rd>, <Rm>
Note	Rd = Rd * Rm, APSR.N, and APSR.Z update. Rd and Rm are low registers.

SBCS and SUBS

Core	
R0	0xFFFFFFFF
R1	0x00000000
R2	0x00000003
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001D0
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	52
Sec	0.00000433

```
0x000001D2 0000    MOVS    r0,r0
0x000001D4 0001    MOVS    r1,r0

system_NUC1xx.c  startup_NUC1xx.s  SBCS.asm  copy_50_bytes.asm

1 ;64 bit subtraction
2
3     PRESERVE8 ; Indicate the code here preserve
4 ; 8 byte stack alignment
5     THUMB     ; Indicate THUMB code is used
6     AREA     |.text|, CODE, READONLY
7
8     EXPORT   __main
9 ; Start of CODE area
10
11
12 __main
13     LDR r0,=0x00000001 ; X_Low(X = 0x0000000100000001)
14     LDR r1,=0x00000001 ; X_High
15     LDR r2,=0x00000003 ; Y_Low(Y = 0x0000000000000003)
16     LDR r3,=0x00000000 ; Y_High
17     SUBS r0,r0,r2 ; lower 32-bit
18     SBCS r1,r1,r3 ; upper 32-bit
19 stop B stop
20     END
```

CMP and CMN

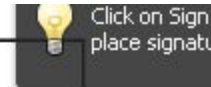
place signature

Instruction		CMP
Function	Compare	
Syntax (UAL)	CMP <Rn>, #immed8	
Syntax (Thumb)	CMP <Rn>, #immed8	
Note	Calculate $Rd - \text{ZeroExtended}(\#immed8)$, APSR update but subtract result is not stored. Rn is a low register.	

Instruction		CMN
Function	Compare negative	
Syntax (UAL)	CMN <Rn>, <Rm>	
Syntax (Thumb)	CMN <Rn>, <Rm>	
Note	Calculate $Rn - \text{NEG}(Rm)$, APSR update but subtract result is not stored. Effectively the operation is an ADD.	

Operations

BIC, MVN, TST



Instruction BIC	
Function	Logical Bitwise Clear
Syntax (UAL)	BICS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	BIC <Rd>, <Rm>
Note	Rd = AND(Rd, NOT(Rm)), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction MVN	
Function	Logical Bitwise NOT
Syntax (UAL)	MVNS <Rd>, <Rm>
Syntax (Thumb)	MVN <Rd>, <Rm>
Note	Rd = NOT(Rm), APSR.N, and APSR.Z update. Rd and Rm are low registers.

Instruction TST	
Function	Test (bitwise AND)
Syntax (UAL)	TST <Rn>, <Rm>
Syntax (Thumb)	TST <Rn>, <Rm>
Note	Calculate AND(Rn, Rm), APSR.N, and APSR.Z update, but the AND result is not stored. Rd and Rm are low registers.

- BIC(1100,0011))
- =AND(1101,not(0011))
- =AND(1101,1100)
- 1101
- 1100
- -----
- **1100**
- BIC(1100,1011))
- =AND(1101,not(1011))
- =AND(1101,0100)
- 1101
- 0100
- -----
- **0100**

BICS,MVN

The image shows a debugger interface with two main panels. The left panel displays the state of the processor's core registers, and the right panel shows the assembly code being executed.

Core Registers:

Register	Value
R0	0x00000030
R1	0x000000C0
R2	0x0000003F
R3	0x000000C0
R4	0xFFFFFFFF
R5	0x000000C0
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001D6
xPSR	0x81000000

Below the registers, the system state is shown:

Mode	Thread
Stack	MSP
States	51
Sec	0.00000425

Assembly Code:

```
0x000001D4 43EC    MVNS    r4,r5
20: stop B stop
0x000001D6 E7FE    B       0x000001D6

BICMVN.asm*  system_NUC1xx.c  startup_NUC1xx.s  Text2  copy_50_bytes.asm

1      PRESERVE8 ; Indicate the code here preserve
2      ; 8 byte stack alignment
3      THUMB      ; Indicate THUMB code is used
4      AREA      |.text|, CODE, READONLY
5
6      EXPORT    __main
7      ; Start of CODE area
8
9      __main
10
11     LDR r0,=0xF0 ; and(r0,not(r1))
12     LDR r1,=0xC0
13     BICS r0,r0,r1;30
14     LDR r2,=0xFF ;
15     LDR r3,=0xC0;and(r2,not(r3))3F
16     BICS r2,r2,r3
17     LDR r4,=0xFF ;
18     LDR r5,=0xC0;3F
19     MVNS r4,r5; 0xC0 = 0x000000C0
20     stop B stop
21     END
```

Extend and Reverse Ordering Operations

Instruction	REV (Byte-Reverse Word)
Function	Byte Order Reverse
Syntax	REV <Rd>, <Rm>
Note	Rd = {Rm[7:0], Rm[15:8], Rm[23:16], Rm[31:24]} Rd and Rm are low registers.

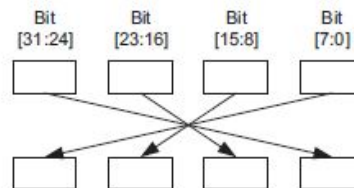
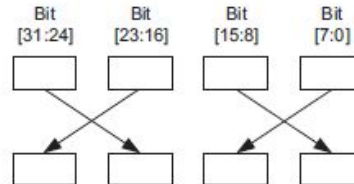


Figure 5.7:
REV operation.

Instruction	REV16 (Byte-Reverse Packed Half Word)
Function	Byte Order Reverse within half word
Syntax	REV16 <Rd>, <Rm>
Note	Rd = {Rm[23:16], Rm[31:24], Rm[7:0], Rm[15:8]} Rd and Rm are low registers.



REV and REV16

- PRESERVE8
- THUMB
- AREA |.TEXT|, CODE, READONLY
- EXPORT __main
- __main
- LDR R0,=0x11223344
- REV R1, R0
- LDR R0,=0x11223344
- REV16 R1, R0
- stop B stop
- END
-

Registers	
Register	Value
Core	
R0	0x11223344
R1	0x44332211
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000E9
R15 (PC)	0x000000C4
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread

Disassembly			
8: stop B stop			
⇒ 0x000000C4	E7FE	B	0x000000C4
0x000000C6	0000	MOVS	r0, r0
0x000000C8	3344	ADDS	r3, r3, #0x44
0x000000CA	1122	ASRS	r2, r4, #4
<			
<div> <div>startup_NUC1xx.s</div> <div>system_NUC1xx.c</div> <div>QB6.asm</div> <div>QA6.asm</div> </div>			
1	PRESERVE8		
2	THUMB		
3	AREA .TEXT , CODE, READONLY		
4	EXPORT __main		
5	__main		
6	LDR R0, =0x11223344		
7	REV R1, R0		
⇒ 8	stop B stop		
9	END		
10			

D:\Documents and Settings\admin\Desktop\Anu 140-1\Anu 140\NuvotonPlatform_KeilSample\EduSampleCode\1MS16CS023\A6.uvproj - µVision4

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
Core	
R0	0x11223344
R1	0x22114433
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000ED
R15 (PC)	0x000000C8
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	42
Sec	0.00000350

Disassembly

10: stop B stop

Address	Disassembly	Comment
0x000000C8	E7FE B	0x000000C8
0x000000CA	0000	MOVS r0,r0
0x000000CC	3344	ADDS r3,r3,#0x44
0x000000CE	1122	ASRS r2,r4,#4

Trace Windows

Show or hide the trace windows

startup_NUC1xx.s system_NUC1xx.c QB6.asm QA6.asm

```
1 PRESERVE
2 THUMB
3 AREA |.TEXT|, CODE, READONLY
4 EXPORT __main
5 __main
6 LDR R0,=0x11223344
7 REV R1, R0
8 LDR R0,=0x11223344
9 REV16 R1, R0
10 stop B stop
11 END
12
```

Command

```
*** error 65: access violation at 0x50000100 : no 'write' permission
*** error 65: access violation at 0x50000024 : no 'write' permission
*** error 65: access violation at 0x50000100 : no 'write' permission
```

Call Stack + Locals

Name	Location/Value	Type
------	----------------	------

ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE DEFINE

Show or hide the trace windows

Simulation t1: 0.00000350 sec

start Latest News, Breakin... GOPI (H:) The.Definitive.Guide... D:\Documents and Se... untitled - Paint Microsoft PowerPoint ... 2:20 PM

REVSH, SXTB, SXTH

Instruction	REVSH (Byte-Reverse Signed Half Word)
Function	Byte order reverse within lower half word, then sign extend result
Syntax	REVSH <Rd>, <Rm>
Note	Rd = SignExtend({Rm[7:0], Rm[15:8]}) Rd and Rm are low registers.

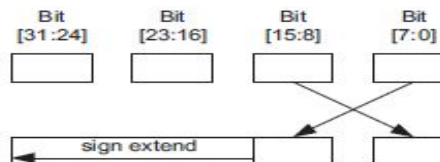


Figure S.9:
REVSH operation.

These reverse instructions are usually used for converting data between little endian and big endian systems.

The SXTB, SXTH, UXT, and UXTH instructions are used for extending a byte or half word data into a word. They are usually used for data type conversions.

Instruction	SXTB (Signed Extended Byte)
Function	SignExtend lowest byte in a word of data
Syntax	SXTB <Rd>, <Rm>
Note	Rd = SignExtend(Rm[7:0]) Rd and Rm are low registers.

Instruction	SXTH (Signed Extended Half Word)
Function	SignExtend lower half word in a word of data
Syntax	SXTH <Rd>, <Rm>
Note	Rd = SignExtend(Rm[15:0]) Rd and Rm are low registers.

UXTB, UXTH

Instruction	UXTB (Unsigned Extended Byte)
Function	Extend lowest byte in a word of data
Syntax	UXTB <Rd>, <Rm>
Note	Rd = ZeroExtend(Rm[7:0]) Rd and Rm are low registers.

Instruction Set 97

Instruction	UXTH (Unsign Extended Half Word)
Function	Extend lower half word in a word of data
Syntax	UXTH <Rd>, <Rm>
Note	Rd = ZeroExtend(Rm[15:0]) Rd and Rm are low registers.

With SXTB or SXTH, the data are extended using bit[7] or bit[15] of the input data, whereas for UXTB and UXTH, the data are extended using zeros. For example, if R0 is 0x55AA8765, the result of these extended instructions is

```
SXTB   R1, R0      ; R1 = 0x00000065
SXTH   R1, R0      ; R1 = 0xFFFF8765
UXTB   R1, R0      ; R1 = 0x00000065
UXTH   R1, R0      ; R1 = 0x00008765
```

SXTH,SXTB,UXTB,UXTH

Register Window:

Register	Value
R0	0x55AA8765
R1	0x00000065
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x00000109
R15 (PC)	0x000000CC
xPSR	0x41000000

Assembly Window:

```
1      PRESERVE8
2      THUMB
3      AREA |.TEXT|, CODE, READONLY
4      EXPORT __main
5      __main
6      LDR R0,=0x11223344
7      REV R1, R0
8      LDR R0,=0x11223344
9      REV16 R1, R0
10     LDR R0,=0x55AA8765
11     SXTB R1, R0 ; R1 = 0x00000065
12     LDR R0,=0xAA558765
13     SXTB R1, R0 ;
14     LDR R0,=0x55AA8765
15     SXTH R1, R0 ; R1 = 0xFFFF8765
16     LDR R0,=0xAA558765
17     SXTH R1, R0
18     UXTB R1, R0 ; R1 = 0x00000065
19     UXTH R1, R0 ; R1 = 0x00008765
20     stop B stop
21     END
22
```

SXTH with 15bit 1(8765)

Core	
R0	0x55AA8765
R1	0xFFFF8765
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x00000100
R15 (PC)	0x000000D4
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	51
Sec	0.00000425

```
0x000000D4 4803      LDR      R0,[PC,#20] ; 0x000000D0
17:          SXTH R1, R0
0x000000D6 B201      SXTH      r1,r0
18:          UXTB R1, R0 ; R1 <= 0x00000065

startup_NUC1xx.s  system_NUC1xx.c  QB6.asm  QA6.asm
1      PRESERVE8
2      THUMB
3      AREA    |.TEXT|, CODE, READONLY
4      EXPORT __main
5      __main
6      LDR R0,=0x11223344
7      REV R1, R0
8      LDR R0,=0x11223344
9      REV16 R1, R0
10     LDR R0,=0x55AA8765
11     SXTB R1, R0 ; R1 <= 0x00000065
12     LDR R0,=0xAA556765
13     SXTB R1, R0 ;
14     LDR R0,=0x55AA8765
15     SXTH R1, R0 ; R1 <= 0xFFFF8765
16     LDR R0,=0xAA558765
17     SXTH R1, R0
18     UXTB R1, R0 ; R1 <= 0x00000065
19     UXTB R1, R0 ; R1 <= 0x00008765
```


SXTH with 15 bit 0(6765)

Core

R0	0xAA558765
R1	0x00006765
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x00000111
R15 (PC)	0x000000D6
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	53
Sec	0.00000442

Assembly code (main):

```
1 PRESERVE8
2 THUMB
3 AREA |.TEXT|, CODE, READONLY
4 EXPORT __main
5 __main
6 LDR R0,=0x11223344
7 REV R1, R0
8 LDR R0,=0x11223344
9 REV16 R1, R0
10 LDR R0,=0x55AA8765
11 SXTB R1, R0 ; R1 ← 0x00000065
12 LDR R0,=0xAA556765
13 SXTB R1, R0 ;
14 LDR R0,=0x55AA6765
15 SXTH R1, R0 ; R1 ← 0xFFFF8765
16 LDR R0,=0xAA558765
17 SXTH R1, R0
18 UXTB R1, R0 ; R1 ← 0x00000065
19 UXTH R1, R0 ; R1 ← 0x000008765
20 stop B stop
21 END
22
```


UXTBwith zero extended

Core

R0	0xAA558765
R1	0x00000065
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x00000115
R15 (PC)	0x000000DC
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	57
Sec	0.00000475

Assembly code:

```
21: UXTH R1, RO ; R1 <= 0x00008765
0x000000DE B281 UXTH r1,r0
22: stop B stop

1 PRESERVE8
2 THUMB
3 AREA |.TEXT|, CODE, READONLY
4 EXPORT __main
5 __main
6 LDR RO,=0x11223344
7 REV R1, RO
8 LDR RO,=0x11223344
9 REV16 R1, RO
10 LDR RO,=0x55AA8765
11 SXTB R1, RO ; R1 <= 0x00000065
12 LDR RO,=0xAA556765
13 SXTB R1, RO ;
14 LDR RO,=0x55AA6765
15 SXTH R1, RO ; R1 <= 0xFFFF8765
16 LDR RO,=0xAA558765
17 SXTH R1, RO
18 LDR RO,=0xAA558765
19 UXTB R1, RO ; R1 <= 0x00000065
20 LDR RO,=0xAA558765
21 UXTH R1, RO ; R1 <= 0x00008765
22 stop B stop
23 END
24
```

UXTH with after 15 bit extended by zero

The screenshot displays a debugger interface with two main panels. The left panel shows the register state for a core, and the right panel shows the assembly code being executed.

Register State (Left Panel):

Register	Value
R0	0xAA558765
R1	0x00008765
R2	0x50000024
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x00000115
R15 (PC)	0x000000E0
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP
States	60
Sec	0.00000500

Assembly Code (Right Panel):

```
0x000000E0 E7FE B 0x000000E0
0x000000E2 0000 MOVs r0,r0
0x000000E4 3344 ADDS r3,r3,#0x44
0x000000E6 1122 ASRS r2,r4,#4

1 PRESERVE8
2 THUMB
3 AREA |.TEXT|, CODE, READONLY
4 EXPORT __main
5 __main
6 LDR R0,=0x11223344
7 REV R1, R0
8 LDR R0,=0x11223344
9 REV16 R1, R0
10 LDR R0,=0x55AA8765
11 SXTB R1, R0 ; R1 <= 0x00000065
12 LDR R0,=0xAA556765
13 SXTB R1, R0 ;
14 LDR R0,=0x55AA6765
15 SXTH R1, R0 ; R1 <= 0xFFFF8765
16 LDR R0,=0xAA558765
17 SXTH R1, R0
18 LDR R0,=0xAA558765
19 UXTB R1, R0 ; R1 <= 0x00000065
20 LDR R0,=0xAA558765
21 UXTH R1, R0 ; R1 <= 0x000008765
22 stop B stop
23 END
24
```

Shift and Rotate Operations

Shift and Rotate Operations

Page 99 of 100

The Cortex-M0 also supports shift and rotate instructions. It supports both arithmetic shift operations (the datum is a signed integer value where MSB needs to be reserved) as well as logical shift.

Instruction	ASR
Function	Arithmetic Shift Right
Syntax (UAL)	ASRS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ASR <Rd>, <Rm>
Note	Rd = Rd >> Rm, last bit shift out is copy to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

Instruction	ASR
Function	Arithmetic Shift Right
Syntax (UAL)	ASRS <Rd>, <Rm>, #immed5

(Continued)

ASR

When ASR is used, the MSB of the result is unchanged, and the Carry flag is updated using the last bit shifted out (Figure 5.3).

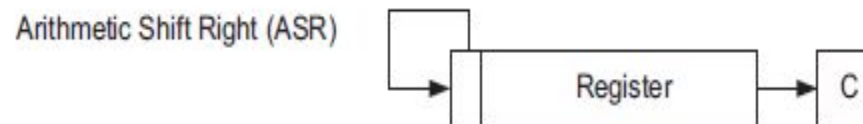


Figure 5.3:
Arithmetic Shift Right.

ASR

- PRESERVE8 ; Indicate the code here preserve
- ; 8 byte stack alignment
- THUMB ; Indicate THUMB code is used
- AREA |.text|, CODE, READONLY
- EXPORT __main
- ; Start of CODE area
- __main
- LDR r2,=0x00000080;
- ASRS r0,r2,#04
- LDR r2,=0x80000000;
- ASRS r0,r2,#04
- stop B stop
- END

ASR

Core Register Map:

Register	Address
R0	0xF8000000
R1	0x00000000
R2	0x80000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001CC
xPSR	0x81000000

ASR.asm Assembly Code:

```
0x000001CE 0000 MOVS r0,r0
0x000001D0 0080 LSLS r0,r0,#2

1 PRESERVE8 ; Indicate the code here preserve
2 ; 8 byte stack alignment
3 THUMB ; Indicate THUMB code is used
4 AREA |.text|, CODE, READONLY
5 EXPORT __main
6 ; Start of CODE area
7 __main
8 LDR r2,=0x00000080;
9 ASRS r0,r2,#04
10 LDR r2,=0x80000000;
11 ASRS r0,r2,#04
12 stop B stop
13 END
14
```


Core	
R0	0xF8000000
R1	0x00000000
R2	0x80000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001CC
xPSR	0x81000000
N	1
Z	0
C	0
V	0
T	1
ISR	0
Banked	

```

0x000001CC E7FE      B      0x000001CC
0x000001CE 0000      MOVs    r0,r0
0x000001D0 0080      lsls    r0,r0,#2

```

asr.asm* NUC1xx.h startup_NUC1xx.s system_NUC1xx.c

```

1      PRESERVE8 ; Indicate the code here preserve
2      ; 8 byte stack alignment
3      THUMB      ; Indicate THUMB code is used
4      AREA      |.text|, CODE, READONLY
5      EXPORT    __main
6      ; Start of CODE area
7      __main
8      LDR r2,=0x00000080;
9      ASRS r0,r2,#04
10     LDR r2,=0x80000000;// (1111,1000,0000....)N=1
11     ASRS r0,r2,#04
12     stop B stop
13     END
14

```

LSLS and LSRS

- For logical shift operations, the instructions are LSL (Figure 5.4) and LSR (Figure 5.5).

Logical Shift Left (LSL)



Figure 5.4:
Logical Shift Left.

Logical Shift Right (LSR)



LSLS

- PRESERVE8 ; Indicate the code here preserve
- ; 8 byte stack alignment
- THUMB ; Indicate THUMB code is used
- AREA |.text|, CODE, READONLY
- EXPORT __main
- ; Start of CODE area
- __main
- LDR r0,=0x80000001;
- LSLS r2,r0,#01
- LDR r0,=0x80000001;
- LSLS r2,r0,#02
- stop B stop
- END

Registers

Register	Value
Core	
R0	0x80000001
R1	0x00000000
R2	0x00000002
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13...	0x20000420
R14...	0x000000DD
R15...	0x000001CA
xPSR	
xPSR	0x21000000
N	0
Z	0
C	1
V	0

Disassembly

11: LSLS r2,r0,#02

⇒ 0x000001CA 0082 LSLS r2,r0,#2

12: stop B stop

0x000001CC F3FF B 0x000001CC

asr.asm*

NUC1xx.h

startup_NUC1xx.s

system_NUC1xx.c

```

1 PRESERVE8 ; Indicate the code here preserve
2 ; 8 byte stack alignment
3 THUMB ; Indicate THUMB code is used
4 AREA |.text|, CODE, READONLY
5 EXPORT __main
6 ; Start of CODE area
7 __main
8 LDR r0,=0x80000001;
9 LSLS r2,r0,#01//0000,.....,0010) C=1
10 LDR r0,=0x80000001;
11 LSLS r2,r0,#02
12 stop B stop
13 END
14

```

Register	Value
Core	
R0	0x80000001
R1	0x00000000
R2	0x00000004
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 ...	0x20000420
R14 ...	0x000000DD
R15 ...	0x000001CC
xPSR	0x01000000
N	0
Z	0
C	0
V	0
T	1
I...	0
+ Banked	

```

11:                                LSLS r2,r0,#02
0x000001CA 0082                LSLS    r2,r0,#2
    12: stop B stop
0x000001CC F2FF                B        0x000001CC
<

asr.asm*  NUC1xx.h  startup_NUC1xx.s  system_NUC1xx.c

1      PRESERVE8 ; Indicate the code here preserve
2      ; 8 byte stack alignment
3      THUMB      ; Indicate THUMB code is used
4      AREA      |.text|, CODE, READONLY
5      EXPORT    __main
6      ; Start of CODE area
7      __main
8      LDR r0,=0x80000001;
9      LSLS r2,r0,#01//0000,.....,0010) C=1
10     LDR r0,=0x80000001;
11     LSLS r2,r0,#02 (0000,0000,.....,0100) C=0
12     stop B stop
13     END
14

```

LSRS

- PRESERVE8 ; Indicate the code here preserve
- ; 8 byte stack alignment
- THUMB ; Indicate THUMB code is used
- AREA |.text|, CODE, READONLY
- EXPORT __main
- ; Start of CODE area
- __main
- LDR r0,=0x80000001;
- LSRS r2,r0,#31
- LDR r0,=0x80000001;
- LSRS r2,r0,#30
- stop B stop
- END

Core	
R0	0x80000001
R1	0x00000000
R2	0x00000001
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001C8
xPSR	0x01000000
N	0
Z	0
C	0
V	0
T	1
ISR	0

```

0x000001C8 4801      LDR      r0,[pc,#4] ; @0x000001D0
11:          LSRS   r2,r0,#30
0x000001CA 0F82      LSRS   r2,r0,#30
<

asr.asm  NUC1xx.h  startup_NUC1xx.s  system_NUC1xx.c

1  PRESERVE8 ; Indicate the code here preserve
2      ; 8 byte stack alignment
3      THUMB ; Indicate THUMB code is used
4      AREA |.text|, CODE, READONLY
5      EXPORT __main
6      ; Start of CODE area
7  __main
8      LDR r0,=0x80000001;
9      LSRS r2,r0,#31
10     LDR r0,=0x80000001;
11     LSRS r2,r0,#30
12 stop B stop
13     END

```

Register	Value
Core	
R0	0x80000001
R1	0x00000000
R2	0x00000002
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001CC
xPSR	0x01000000
N	0
Z	0
C	0
V	0
T	1
ISR	0
Banked	

```

12: stop B stop
0x000001CC E7FE      B      0x000001CC
0x000001CE 0000      MOVS    r0,r0
0x000001D0 0001      MOVS    r1,r0

```

asr.asm NUC1xx.h startup_NUC1xx.s system_

```

1      PRESERVE8 ; Indicate the code here pres
2      ; 8 byte stack alignment
3      THUMB      ; Indicate T
4      AREA      |.text|, CODE, REA
5      EXPORT    __main
6      ; Start of CODE area
7      __main
8      LDR r0,=0x80000001;
9      LSRS r2,r0,#31
10     LDR r0,=0x80000001;
11     LSRS r2,r0,#30
12     stop B stop
13     END

```

RORS



Figure 5.6:
Rotate Right.

rotate right.

Instruction	ROR
Function	Rotate Right
Syntax (UAL)	RORS <Rd>, <Rd>, <Rm>
Syntax (Thumb)	ROR <Rd>, <Rm>
Note	Rd = Rd rotate right by Rm bits, last bit shifted out is copied to APSR.C, APSR.N and APSR.Z are also updated. Rd and Rm are low registers.

RORS

- PRESERVE8 ; Indicate the code here preserve
- ; 8 byte stack alignment
- THUMB ; Indicate THUMB code is used
- AREA |.text|, CODE, READONLY
- EXPORT __main
- ; Start of CODE area
- __main
- LDR r0,=0x80000001;
- MOVS r2,#31;32-1
- RORS r0,r2
- LDR r0,=0x80000001;
- MOVS r2,#30
- RORS r0,r2
- stop B stop
- END

Register	Value
Core	
R0	0x00000003
R1	0x00000000
R2	0x0000001F
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 ...	0x20000420
R14 ...	0x000000DD
R15 ...	0x000001CA
xPSR	0x01000000
N	0
Z	0
C	0
V	0
T	1
I...	0
Banked	

```

0x000001C8 41D0      RORS    r0,r0,r2
11:                LDR r0,=0x80000001;
0x000001CA 4802      LDR     r0,[pc,#8] ; @0x000001D4
12:                MOVS    r2,#30

```

asr.asm NUC1xx.h startup_NUC1xx.s system_NUC1xx.c

```

1      PRESERVE8 ; Indicate the code here preserve
2      ; 8 byte stack alignment
3      THUMB      ; Indicate THUMB code is
4      AREA      |.text|, CODE, READONLY
5      EXPORT    __main
6      ; Start of CODE area
7      __main
8      LDR r0,=0x80000001;
9      MOVS r2,#31;32-1
10     RORS r0,r2
11     LDR r0,=0x80000001;
12     MOVS r2,#30
13     RORS r0,r2
14     stop B stop
15     END

```

Register	Value
Core	
R0	0x00000006
R1	0x00000000
R2	0x0000001E
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001D0
xPSR	0x01000000
N	0
Z	0
C	0
V	0
T	1

14: stop B stop

```

0x000001D0 E7FE      B      0x000001D0
0x000001D2 0000      MOVS    r0,r0
0x000001D4 0001      MOVS    r1,r0

```

asr.asm

NUC1xx.h

startup_NUC1xx.s

system_NUC1xx.c

```

1      PRESERVE8 ; Indicate the code here preserve
2      ; 8 byte stack alignment
3      THUMB      ; Indicate THUMB code i:
4      AREA      |.text|, CODE, READONLY
5      EXPORT __main
6      ; Start of CODE area
7      __main
8      LDR r0,=0x80000001;
9      MOVs r2,#31;32-1
10     RORS r0,r2
11     LDR r0,=0x80000001;
12     MOVs r2,#30
13     RORS r0,r2
14     stop B stop
15     END

```

Branch and Branch under condition

Program Flow Control

There are five branch instructions in the Cortex-M0 processor. They are essential for program flow control like looping and conditional execution, and they allow program code to be partitioned into functions and subroutines.

Instruction	B (Branch)
Function	Branch to an address (unconditional)
Syntax	B <label>
Note	Branch range is +/- 2046 bytes of current program counter

Instruction	B<cond> (Conditional Branch)
Function	Depending of APSR, branch to an address
Syntax	B<cond> <label>
Note	Branch range is +/- 254 bytes of current program counter. For example, CMP R0, #0x1 ; Compare R0 with 0x1 BEQ process1 ; Branch to process1 if R0 equal 1

Condition Suffixes.

Table 5.7: Condition Suffixes for Conditional Branch

Suffix	Branch Condition	Flags (APSR)
EQ	Equal	Z flag is set
NE	Not equal	Z flag is cleared
CS/HS	Carry set / unsigned higher or same	C flag is set
CC/LO	Carry clear / unsigned lower	C flag is cleared
MI	Minus / negative	N flag is set (minus)
PL	Plus / positive or zero	N flag is cleared
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared ($N == V$)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set ($N != V$)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared ($Z == 0$ and $N == V$)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set ($Z == 1$ or $N != V$)

place signature on

The loop will execute three times. The third time, PC is 1 before the SUBS instruction. After

BL and BX

Instruction		BL (Branch and Link)
Function	Branch to an address and store return address to LR. Usually use for function calls, and can be used for long-range branch that is beyond the branch range of branch instruction (B <label>).	
Syntax	BL <label>	
Note	Branch range is $\pm 16\text{MB}$ of current program counter. For example, BL functionA ; call a function called functionA	

Instruction		BX (Branch and Exchange)
Function	Branch to an address specified by a register, and change processor state depending on bit[0] of the register.	
Syntax	BX <Rm>	
Note	Because the Cortex-M0 processor only supports Thumb code, bit[0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will generate a fault exception.	

BLX-when working with a function pointers

Instruction BLX (Branch and Link with Exchange)	
Function	Branch to an address specified by a register, save return address to LR, and change processor state depending on bit[0] of the register.
Syntax	BLX <Rm>
Note	Because the Cortex-M0 processor only supports Thumb code, the bit [0] of the register content (Rm) must be set to 1, otherwise it means it is trying to switch to the ARM state and this will create a fault exception.

BLX is used when a function call is required but the address of the function is held inside a register (e.g., when working with function pointers).

Memory Barrier Instructions

Instruction DMB	
Function	Data Memory Barrier
Syntax	DMB
Note	Ensures that all memory accesses are completed before new memory access is committed

Instruction DSB	
Function	Data Synchronization Barrier
Syntax	DSB
Note	Ensures that all memory accesses are completed before the next instruction is executed

Instruction ISB	
Function	Instruction Synchronization Barrier
Syntax	ISB
Note	Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Exception-Related Instructions

The Cortex-M0 processor provides an instruction called supervisor call (SVC). This instruction causes the SVC exception to take place immediately if the exception priority level is higher than the current level.

Instruction SVC	
Function	Supervisor call
Syntax	SVC # <immed8> SVC <immed8>
Note	Trigger the SVC exception. For example, SVC #3 ; SVC instruction, with parameter, equals 3. Alternative syntax without the “#” is also allowed. For example, SVC 3 ; this is the same as SVC #3.

Instruction CPS	
Function	Change processor state: enable or disable interrupt
Syntax	CPSIE I ; Enable Interrupt (Clearing PRIMASK) CPSID I ; Disable Interrupt (Setting PRIMASK)
Note	PRIMASK only blocks external interrupts, SVC, PendSV, SysTick. But it does not block NMI and the hard fault handler.

Sleep Mode Feature Related Instructions

Instruction		WFI
Function	Wait for Interrupt	
Syntax	WFI	
Note	Stops program execution until an interrupt arrives or until the processor enters a debug state.	



Instruction		WFE
Function	Wait for Event	
Syntax	WFE	
Note	If the internal event register is set, it clears the internal event register and continues execution. Otherwise, stop program execution until an event (e.g., an interrupt) arrives or until the processor enters a debug state.	

Send Event to Wake up other processor

The Send Event (SEV) instruction is normally used in multiprocessor systems to wake up other processors that are in sleep mode by means of the WFE instruction. For single-processor systems, where the processor does not have a multiprocessor communication interface or the multiprocessor communication interface is not used, the SEV can only affect the local event register inside the processor itself.

Instruction		SEV
Function	Send event to all processors in multiprocessing environment (including itself)	
Syntax	SEV	
Note	Set local event register and send out an event pulse to other microprocessor in a multiple processor system	

Get the code

Other Instructions

Instruction		NOP
Function		No operation
Syntax		NOP
Note		The NOP instruction takes one cycle minimum on Cortex-M0. In general, delay timing produced by NOP instruction is not guaranteed and can vary among different systems (e.g., memory wait states, processor type). If the timing delay needs to be accurate, a hardware timer should be used.

Instruction		BKPT
Function		Breakpoint
Syntax		BKPT #<immed8> BKPT <immed8>
Note		BKPT instruction can have an 8-bit immediate data field. The debugger can use this as an identifier for the BKPT. For example, BKPT #0 ; breakpoint, with immediate field equal zero Alternative syntax without the “#” is also allowed. For example, BKPT 0 ; This is the same as BKPT #0.

Instruction		YIELD
Function		Indicate task is stalled
Syntax		YIELD
Note		Execute as NOP on the Cortex-M0 processor

LDR using Label

Pseudo Instruction	LDR
Function	Load a 32-bit immediate data into register Rd
Syntax	LDR <Rd>, =immed32
Note	This is translated to a PC-related load from a literal pool. For example, LDR R0, =0x12345678 ; Set R0 to hexadecimal value 0x12345678 LDR R1, =10 ; Set R1 to decimal value 10 LDR R2, ='A' ; Set R2 to character 'A'



Click on Sign to add text and place signature on a PDF File

Pseudo Instruction	LDR
Function	Load a data in specified address (label) into register
Syntax	LDR <Rd>, label
Note	The address of label must be word aligned and should be closed to the current program counter. For example, you can put a data item in program ROM using DCD and then access this data item using LDR. LDR R0, CONST_NUM ; Load CONST_NUM (0x17) in R0 ... ALIGN 4 ; make sure next data are word aligned CONST_NUM DCD 0x17 ; Put a data item in program code

Other pseudo instructions depend on the tool chain being used. For more information, please