

Tutorial 9: Instruction Encoding and Decoding

Problem 1: Pseudo Instruction

Consider the ARM assembly code in Figure 1 Answer the following questions:

What is the real ARM instruction corresponding to the pseudo instruction at label L2?

What is the real ARM instruction corresponding to the pseudo instruction at label L3?

What is the real ARM instruction corresponding to the pseudo instruction at label L4?

What is the content of the register r0 in instruction (str r0, [r2, #0])?

```
L1:    ldr    r0, data+4
L2:    ldr    r1, data+8
      ldr    r0, [r0, #0]
      ldr    r1, [r1, #0]
      add   r0, r0, r1
L3:    adr    r2, data
L4:    mov    r3, #0x00ffffff
      add   r0, r0, r3
      str   r0, [r2, #0]
ldmfd sp!, {pc}

data:
      .word 0x20000
      .word 0x300
      .word 0x400
```

Figure 1: Sample ARM Assembly Code

The assembler will turn the pseudo instruction at label L1 (`ldr r0, data+4`) into a real Load/Store instruction of the type (`ldr Rdest, [Rbase, #offset]`). Since the data at the label `data+4` is 11 words away from the instruction (`ldr r0, data+4`), the word address for (`data+4`) is at an offset of $(11 - 2) \times 4 = 36$ bytes from the current value of the register PC. We subtract 2 from 11 because the current value of register PC is two instructions away from the current instruction being executed (due to pipelining). So, the replacement instruction will be (`ldr r0, [pc, #36]`).

The pseudo instruction at label L2 (`ldr r1, data+8`) will similarly convert to (`ldr r1, [pc, #36]`).

The pseudo instruction at label L3 (`adr r2, data`) should load the address of the word at the label (`data`) into R2. Since at this point PC is 3 words away from the address of (`data`) the corresponding real instruction is (`add r0, pc, #12`).

The instruction at label L4 (`mov r3, #0x00ffffff`), is not a valid instruction as an immediate value (`0x00ffffff`) does not fit in 8 bits, or a rotated version of an 8-bit number (rotation in steps of two bits in a length of 32). However, the complement of (`0x00ffffff`) which is (`0xff000000`) fits in 8 bits (rotated in right direction by 8 bits). So, the instruction (`mov r3, #0x00ffffff`) will get converted to (`mvn r3, #0xff000000`) by the assembler. However, note that an instruction like (`mov r3, #0x00ffffff0`) where the immediate value requires more than 12 bits, no matter how it is manipulated, will cause the assembler error.

It is tempting to assume that at the point of the execution of the instruction (`str r0, [r2, #0]`) the content of the register R0 is $(=R0 + R1 + R3) (=0x200 + 0x300 + 0x00ffffff) (=0x10006ff)$. However, note that instructions (`ldr r0, [r0, #0]`) and (`ldr r0, [r0, #0]`) use ($R0 = 0x300$) and ($R0 = 0x400$) as addresses (pointers) in memory. What is stored at these addresses is not known. Therefore, at the point of the execution of the instruction (`str r0, [r2, #0]`) the content of the register R0 is unknown.

Problem 2: Instruction decoding

Convert the the ARM machine code in Figure 2 to a corresponding C Program

0:	e5902000
4:	e3a00000
8:	e1a03000
c:	e1530002
10:	a1a0f00e
14:	e0800003
18:	e2833001
1c:	e1530002
20:	baffffffb
24:	e1a0f00e

Figure 2: The Sample ARM Machine Code

Figure 3 is the binary representation of the code segment in Figure 2, with condition and instruction type fields separated out.

		bits	bits	
		31-28	27-25	
0:	e5902000	1110	010	110010000001000000000000
4:	e3a00000	1110	001	110100000000000000000000
8:	e1a03000	1110	000	110100000001100000000000
c:	e1530002	1110	000	101010011000000000000010
10:	a1a0f00e	1010	000	1101000001111000000001110
14:	e0800003	1110	000	010000000000000000000011
18:	e2833001	1110	001	0100000110011000000000001
1c:	e1530002	1110	000	101010011000000000000010
20:	baffffffb	1011	101	011111111111111111111011
24:	e1a0f00e	1110	000	1101000001111000000001110

Figure 3: The Sample ARM Machine Code in Binary

By looking at the bits (27 - 25), we identify the instruction type. Figure 4 separates the instructions into 3 categories; Load/Store, Data Processing and Control.

		31-28	27-25	
Load Store Category:				
0:	e5902000	1110	010	110010000001000000000000
Data Processing Category:				
4:	e3a00000	1110	001	110100000000000000000000
8:	e1a03000	1110	000	110100000001100000000000
c:	e1530002	1110	000	101010011000000000000010
10:	a1a0f00e	1010	000	1101000001111000000001110
14:	e0800003	1110	000	010000000000000000000011
18:	e2833001	1110	001	0100000110011000000000001
1c:	e1530002	1110	000	101010011000000000000010
Control Category:				
20:	baffffffb	1011	101	011111111111111111111011
Data Processing Category:				
24:	e1a0f00e	1110	000	1101000001111000000001110

Figure 4: The sample ARM Machine Code

Next, we decode the instructions in each category one by one.

Figure 5 demonstrate the decoding process for the single instruction in the data processing category.

```

Load Store Category:
0:  e5902000  1110  010  11001000000100000000000000

14=Always  load/store immediate Pre index up (+ve offset) word transfer
1110      010                1      1      0

no write back load Base reg=0 Destination reg=2  offset = 0
0         1    0000      0010      000000000000

Decoded Instruction:
0:  ldr r2, [r0, #0]

```

Figure 5: The Decoding Process for the Load/Store Instruction

Figure 6 demonstrates the decoding process for the instructions in the data processing category. We divide these instructions into immediate and register sub-categories.

```

Data Processing Category Immediate:
          Cond      opcode S  Rn  Rd  Immediate Value
4:  e3a00000  1110  001  1101  0 0000 0000 000000000000
18: e2833001  1110  001  0100  0 0011 0011 000000000001

Decoding Process:
4:  e3a00000  14=Al 001 13=mov 0=(no S) 0=r0 0=r0 0=0
18: e2833001  14=AL 001 4=add 0=(no S) 3=r3 3=r3 1=1

Decoded Instructions:
4:  mov  r0, #0
18: add  r3, r3, #1

Data Processing Category Register:
          Cond      opcode S  Rn  Rd  Shift fields      Rm
8:  e1a03000  1110  000  1101  0 0000 0011 00000000      0000
c:  e1530002  1110  000  1010  1 0011 0000 00000000      0010
10: a1a0f00e  1010  000  1101  0 0000 1111 00000000      1110
14: e0800003  1110  000  0100  0 0000 0000 00000000      0011
1c: e1530002  1110  000  1010  1 0011 0000 00000000      0010
24: e1a0f00e  1110  000  1101  0 0000 1111 00000000      1110

Decoding Process:
8:  e1a03000  14=AL 000 13=mov 0=(no S) 0=r0 3=r3 0=(no shift) 0=r0
c:  e1530002  14=AL 000 10=cmp 1=(S) 3=r3 0=r0 0=(no shift) 2=r2
10: a1a0f00e  10=GE 000 13=mov 0=(no S) 0=r0 15=r15 0=(no shift) 14=r14
14: e0800003  14=AL 000 4=add 0=(no S) 0=r0 0=r0 0=(no shift) 3=r3
1c: e1530002  14=AL 000 10=cmp 1=(S) 3=r3 0=r0 0=(no shift) 2=r2
24: e1a0f00e  14=AL 000 13=mov 0=(no S) 0=r0 15=r15 0=(no shift) 14=r14

Decoded Instructions:
8:  mov  r3, r0
c:  cmp  r3, r2
10: movge r15, r14
14: add  r0, r0, r3
1c: cmp  r3, r2
24: mov  r15, r14

```

Figure 6: The Decoding Process for the Data Processing Instructions

Figure 7 demonstrates the decoding process for the single instruction in the control category.

Control Category:		Cond	Branch	Signed offset in word
20:	baffffffb	1011	101	0 111111111111111111111111011
Decoding Process:				
20:	baffffffb	11=LT	101	0=Branchonly -5= go back 5 words (20 bytes) from current PC, or 3 words (12 bytes) from current instruction.
Decoded Instructions:				
ble 0x14				

Figure 7: The Decoding Process for the Data Processing Instructions

Putting back everything together, we will have the ARM assembly code in Figure 8

0:	e5902000	ldr	r2, [r0]
4:	e3a00000	mov	r0, #0
8:	e1a03000	mov	r3, r0
c:	e1530002	cmp	r3, r2
10:	a1a0f00e	movge	r15, r14
14:	e0800003	add	r0, r0, r3
18:	e2833001	add	r3, r3, #1
1c:	e1530002	cmp	r3, r2
20:	baffffffb	blt	0x14
24:	e1a0f00e	mov	r15, r14

Figure 8: The Decoded ARM Assembly Code

The ARM assembly code in Figure 8 can be re-written as symbolic ARM assembly code as show in Figure 9.

sum_arr:			
	ldr	a3, [a1]	
	mov	a1, #0	
	mov	a4, a1	
	cmp	a4, a3	
Loop:	movge	pc, lr	
	add	a1, a1, a4	
	add	a4, a4, #1	
	cmp	a4, a3	
	blt	Loop	
	mov	pc, lr	

Figure 9: The Decoded ARM Symbolic Assembly Code

The symbolic ARM assembly code in Figure 9 can be re-written as a C code as show in Figure 10.

int sum_arr(int *num)
{
int n, s=0;
for (n=0; n<(*num); n++)
{
s = s + n;
}
return s;
}

Figure 10: The Decoded C Code

The C function in Figure 10 corresponds to the algorithm in (1).

$$s = \sum_{i=0}^{N-1} i \quad (1)$$

Problem 3: Literal Pool

Consider the C function in Figure 11. Produce the symbolic and true ARM assembly and machine versions of this code.

```
int sum(int s)
{
  int n = 0x00ff00ff,
  int m = 0xee00ee00;
  return (s + m + n);
}
```

Figure 11: C-code for the Demonstration of the Literal Pool

The code in Figure 12 is the symbolic ARM assembly version of the C code in Figure 11.

```
ldr a2, =0x00ff00ff ; pseudo instruction
ldr a3, =0xee00ee00 ; pseudo instruction
add a2, a2, a3
add a1, a1, a2
```

Figure 12: Symbolic Assembly Code for the Demonstration of the Literal Pool

The code in Figure 13 is the real ARM assembly version of the symbolic code in Figure 12.

```
ldr r1, [pc, #8]; real instruction
ldr r2, [pc, #8]; real instruction
add r1, r1, r2
add r0, r0, r1
L_Pool:
.word 0x00ff00ff
.word 0xee00ee00
```

Figure 13: Real Assembly Code for the Demonstration of the Literal Pool

The code in Figure 14 is the real machine version of the real ARM assembly code in Figure 13.

```
0: e59f1008 ldr r1, [pc, #8]; real instruction
4: e59f2008 ldr r2, [pc, #8]; real instruction
8: e0811002 add r0, r0, r2
c: e0800001 add r0, r0, r1
L_Pool:
.word 0x00ff00ff
.word 0xee00ee00
```

Figure 14: Machine Code for the Demonstration of the Literal Pool.