MORE ON ASSEMBLY PROGRAMMING

EL-GY 6483 Real Time Embedded Systems

A SIMPLE C PROGRAM

```
int total;

int i;

total = 0;

for (i = 10; i > 0; i--) {

total += i;

}
```

ARM EQUIVALENT

```
MOV R0, #0 ; R0 accumulates total

MOV R1, #10 ; R1 counts from 10 to 1

again ADD R0, R0, R1

SUBS R1, R1, #1

BNE again

halt B halt ; infinite loop to stop
```

MEMORY INSTRUCTIONS

```
addInts MOV R4, #0

addLoop LDR R2, [R0]

ADD R4, R4, R2

ADD R0, R0, #4

SUBS R1, R1, #1

BNE addLoop
```

MORE ADDRESSING MODES

```
addInts MOV R4, #0

addLoop SUBS R1, R1, #1

LDR R2, [R0, R1, LSL #2]

ADD R4, R4, R2

BNE addLoop
```

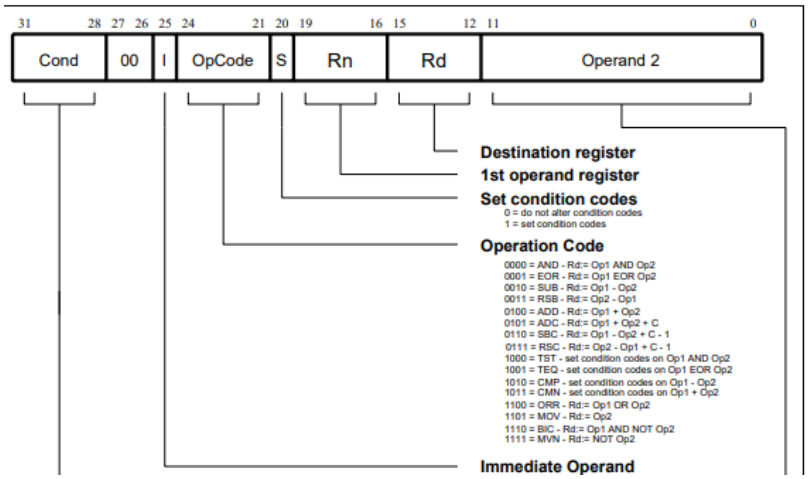
WHAT DOES THE ASSEMBLER DO?

Translate

MOV R3, R9

into

1110 0001 1010 0000 0011 0000 0000 1001



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2													Data processing

1101	LE	Signed less than or equal	Z set or N is no
1110	AL	Always	any
1111	NV	Never (do not use!)	none

HELLO WORLD FOR ARM ASSEMBLY

```
.global _start
_start:
MOV R7, #4
MOV R0, #1
MOV R2, #12
LDR R1, =string
SWI 0
MOV R7, #1
SWI 0
.data
string:
.ascii ”Hello World\n”
```

CALLING ARM ASSEMBLY FROM C

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{ const char *srcstr = ”First string - source ”;
char dststr[] = ”Second string - destination ”;
/* dststr is an array since we’re going to change it */
printf(”Before copying:\n”);
printf(” %s\n %s\n”,srcstr,dststr);
strcopy(dststr,srcstr); // in strcopy.s
printf(”After copying:\n”);
printf(” %s\n %s\n”,srcstr,dststr);
return (0);
}
```

CALLING ARM ASSEMBLY FROM C

```
PRESERVE8
AREA SCopy, CODE, READONLY
EXPORT strcopy
strcopy         ; R0 points to destination string.
                ; R1 points to source string.
LDRB R2, [R1],#1 ; Load byte and update address.
STRB R2, [R0],#1 ; Store byte and update address.
CMP R2, #0      ; Check for null terminator.
BNE strcopy     ; Keep going if not.
BX lr           ; Return.
END
```

INLINE ASSEMBLY EXAMPLES

Calling assembly inline:

/* NOP example */

__asm(”mov r0,r0”);

```
/* Rotating bits example */
asm("mov %[result], %[value], ror #1" : [result] "=r" (y) : [value] "r" (x));
```

Each *asm* statement is divided by colons into up to four parts:

1. The assembler instructions, defined in a single string literal:

   ```
   "mov %[result], %[value], ror #1"
   ```

2. An optional list of output operands, separated by commas. Each entry consists of a symbolic name enclosed in square brackets, followed by a constraint string, followed by a C expression enclosed in parentheses. Our example uses just one entry:

   ```
   [result] "=r" (y)
   ```

3. A comma separated list of input operands, which uses the same syntax as the list of output operands. Again, this is optional and our example uses one operand only:

   ```
   [value] "r" (x)
   ```

Variable Constraints

Constraint	Used for	Range
a	Simple upper registers	r16 to r23
b	Base pointer registers pairs	y,z
d	Upper register	r16 to r31
e	Pointer register pairs	x,y,z
G	Floating point constant	0.0
I	6-bit positive integer constant	0 to 63
J	6-bit negative integer constant	-63 to 0
K	Integer constant	2
L	Integer constant	0
l	Lower registers	r0 to r15
M	8-bit integer constant	0 to 255
n	16-bit integer constant?	
N	Integer constant	-1
O	Integer constant	8, 16, 24
P	Integer constant	1
q	Stack pointer register	SPH:SPL
r	Any register	r0 to r31
t	Temporary register	r0
v	32-bit integer constant?	
w	Special upper register pairs	r24, r26, r28, r30
x	Pointer register pair X	x (r27:r26)
y	Pointer register pair Y	y (r29:r28)
z	Pointer register pair Z	z (r31:r30)

Modifier	Specifies
=	Write-only operand, usually used for all output operands.
+	Read-write operand (not supported by inline assembler)
&	Register should be used for output only

```
asm volatile("swap %0" : "=r" (value) : "0" (value));
```

INLINE ASSEMBLY EXAMPLE

```
int foo(int x, int y)
{
__asm
{
SUBS x,x,y
BEQ end
}
return 1;
end:
return 0;
}
```

ARM CALLING CONVENTION

Register	Role
r0-r3	arguments passed, r0 holds result
r4-r10	local variables, callee-save
r11	frame pointer
r12	intra-procedure-call scratch register
r13	stack pointer
r14	link register
r15	program counter

Additional parameters and return values may be passed via stack.