First ARM® Program

First Program

```
AREA  Prog1, CODE, READONLY      ;directive for assembler
     ENTRY                       ;entry point for program
     MOV    r0, #0x11            ;move a hex 11 into r0
     MOV    r1, r0, LSL #1       ;shift r0 one bit left
                                 ; and move result to r1
     MOV    r2, r1, LSL #1       ;shift r1 one bit left
                                 ; and move to r2
stop B      stop                 ;branch to stop
END                              ;directive for assembler
```

What is the value in r2 after the program reaches stop?

First Program - AREA

```
AREA   Prog1, CODE, READONLY    ;directive for assembler
```

- AREA – indicates a section of code of type DATA
- Prog1 – the name of the program
- READONLY – default, can not be overwritten

The AREA directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

First Program - ENTRY

```
ENTRY                          ;entry point for program
```

The **ENTRY** directive declares an entry point to a program.

Syntax **ENTRY** Usage

You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time. You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

First Program – 1st MOV

```
MOV   r0, #0x11          ;move a hex 11 into r0
```

The `MOV` instruction places a copy of the operand (in this case immediate value hexadecimal `0x11` into register r0. `0x11` is the 32-bit string:

0000 0000 0000 0000 0000 0000 0001 0001

The `#` means that the constant is an immediate operand that is coded into the machine instruction for `MOV`

First Program – 2nd MOV

```
MOV   r1, r0, LSL #1     ;shift r0 one bit left
                         ; and move result to r1
```

The `MOV` instruction places a copy of the operand (in this case r0 left shifted by 1 bit) into r1. In binary, this is the string:

0000 0000 0000 0000 0000 0000 0010 0010

`LSL` is "logical left shift", vacates bits are replaced by 0s.

`LSL` can also be used as a standalone instruction

Due to unique ARM architecture, this is an efficient way to do both a multiply (by certain values) and a `MOV` at the same time.

First Program – 3rd MOV

```
MOV  r2, r1, LSL #1     ;shift r1 one bit left
                        ;       and move to r2
```

The MOV instruction places a copy of the operand (in this case r1 left shifted by 1 bit) into r2. In binary, this is the string:

0000 0000 0000 0000 0000 0000 0100 0100

LSL is "logical left shift", vacates bits are replaced by 0s.

LSL can also be used as a standalone instruction

Due to unique ARM architecture, this is an efficient way to do both a multiply (by certain values) and a MOV at the same time.

First Program – B

```
stop B      stop                ;branch to stop
```

'stop' is a user-defined label, B is the ARM branch (ie jump or go to) instruction. This causes an infinite loop as in each clock cycle the PC is updated to contain the address represented by 'stop'.

First Program – END

```
END                                ;directive for assembler
```

`END` is a directive that indicates to the assembler that the file containing all the `AREA`s for this program are complete. It directs the assembler to halt processing of the file content into an object file.

Using ADS 1.2.1

- Create a separate subdirectory for each program you are working on.
- Use a TEXT editor to create the source file. You can use the one built in to ADS, or you can use a tools such as DOS edit, Windows Notepad, MAC OS textedit, or even MS Word – but you MUST save the file as a text file (not RTF or doc, etc)
- Follow the directions in Lab 1 to create a project, assemble the code, and use the debugger to download it into the Evaluator7T board
- Use the debugger to run the code and observe the memory and register content. You can single step and set break points.

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
    CMP   r3,#0                  CMP   r3,#0
    BEQ   skip                   ADDNE r0,r1,r2
    ADD   r0,r1,r2
skip
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
    …
    SUBS r1,r1,#1     ← decrement r1 and set flags
    BNE  loop         ← if Z flag clear then branch
```


Condition Codes

- The possible condition codes are listed below
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Conditional Codes Examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

Conditional Codes Examples

- Use a sequence of several conditional instructions

```
  if (a==0) func(1);
    CMP      r0,#0
    MOVEQ    r0,#1
    BLEQ     func
```

- Set the flags, then use various condition codes

```
  if (a==0) x=0;
  if (a>0)  x=1;
    CMP      r0,#0
    MOVEQ    r1,#0
    MOVGT    r1,#1
```

- Use conditional compare instructions

```
  if (a==4 || a==10) x=0;
    CMP      r0,#4
    CMPNE    r0,#10
    MOVEQ    r1,#0
```

Second ARM® Program

Second Program

- Calculates the factorial of a value n:

$$n! = \prod_{i=1}^{n} i = (n)(n-1)(n-2)\ldots(2)(1)$$

- Loads value of n into registers r6 and r4
- Uses a conditional loop to do each multiply (`MULNE`)
- r6 accumulates the factorial value
- r4 is decremented for each multiply
- when r4 is zero, the computation is complete and r6 contains $n!$
- Uses the Zero flag (`Z`) to control the loop branching

Second Program

```
        AREA Prog2, CODE, READONLY
        ENTRY
        MOV      r6, #10      ;r6<-n value
        MOV      r4, r6       ;r4<-n value
loop    SUBS     r4, r4, #1   ;r4,<-r4-1
        ;mult only if Z=0, Z flag is set
        ;based on previous subtraction
        MULNE    r7, r6, r4   ;r7<-r6*r4
        MOV      r6, r7     ;update latest
                            ;value to r6
        BNE      loop       ;if Z=1, go to loop
stop    B        stop       ;calculation done
        END
```


Second Program

```
loop    SUBS     r4, r4, #1   ;r4,<-r4-1
```

Placing the 'S' at the end of the SUB instruction indicates that the processor is to update the condition flags in CPSR based on the outcome of the instruction. Some architectures automatically update flags after each operation, but not in this case – it must be done explicitly.

'loop' is a user-defined label that is used as a target for the branch instruction (BNE).

Second Program

```
MULNE    r7, r6, r4  ;r7<-r6*r4
```

'NE' is short for 'Not Equal'. To determine if to values are equal, a subtraction is performed, and if the result is zero, the Z flag is set (Z=1) and the values are equal. Putting NE at the end of the MUL instruction makes it a *conditional instruction*. If Z=0 indicating 'not equal', the multiplication is performed. If Z=1 indicating 'equal' a 'no-operation, nop) is executed in place of the MUL.

Second Program

```
BNE      loop     ;if Z=1, go to loop
```

The Z flag is still set based on the SUBS command. In other architectures, where flags are set after every arithmetic instruction, this might not be the case since a MUL occurred after the SUB. Here, we can still rely on the Z flag for the branch that was updated by the SUB instruction.

Third Program

- Exchanging the content of registers r0 and r1
- In high-level languages, this is often done using a temporary or third storage space
- Can also use the logical Exclusive-OR operation
- Recall the XOR operation:

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

$$A \oplus A = 0$$

$$A \oplus 1 = \overline{A}$$

$$A \oplus 0 = A$$

Third Program

- `EOR` is bitwise XOR of register content
- Swapping can be accomplished using three `EOR` instructions:

$$A_1 \leftarrow A \oplus B$$

$$B_{final} \leftarrow A_1 \oplus B = (A \oplus B) \oplus B = A$$

$$A_{final} \leftarrow A_1 \oplus B_{final} = (A \oplus B) \oplus (A) = B$$

Third Program

```
        AREA    Prog3, CODE, READONLY
        ENTRY
        LDR       r0, =0xF631024C
        LDR       r1, =0x17539ABD
        EOR       r0, r0, r1
        EOR       r1, r0, r1
        EOR       r0, r0, r1
stop    B    stop
        END
```


Third Program

```
LDR        r0, =0xF631024C
```

This loads a constant into r0. Before we used a `MOV` with a constant preceded by #. This is called a "pseduo-instruction". Because the immediate filed is limited in size in the `MOV` instruction format, `0xF631024C` cannot fit into the field and we must use the pseudo-instruction form to load this constant.

We will discuss this in more detail later in class.

What is the result of EOR of `0xF631024C` with `0x17539ABD` ?

Third Program

```
EOR Calculation:

0xF631024C:
1111 0110 0011 0001 0000 0010 0100 1100

0x17539ABD:
0001 0111 0101 0011 1001 1010 1011 1101

0xF631024C EOR 0x17539ABD:
1110 0001 0110 0010 1001 1000 1111 0001
  0xE16298F1
```


ARM® Assembler Language

Constant Values

- Constants can be Expressed as Numeric Values or Character Strings
  - Decimal: 1324
  - Hexadecimal: 0x3DE2 (32-bit value, zero-padded)
  - General: n_xxxx (n is base in [2,9], xxx is digit string)
  - Character: 'V' (enclosed in single quote)
  - String: "Hello world!\n"
- Control Characters Specified as in C Language
- Single Quote: ' \' '
- Dollar Sign or Double Quote: Use Two in a row "$$" is a SINGLE Dollar Sign; " "" " is a single Double Quote

Predefined Register Names

- r0-r15 and R0-R15
- a1-a4 (argument, result, or scratch registers, synonyms for r0 to r3)
- v1-v8 (variable registers, r4 to r11)
- sb and SB (static base, r9)
- sl and SL (stack limit, r10)
- fp and FP (frame pointer, r11)
- ip and IP (intra-procedure-call scratch register, r12)
- sp and SP (stack pointer, r13)
- lr and LR (link register, r14)
- pc and PC (program counter, r15).

Predefined Register Names

Predeclared program status register names

The following program status register names are predeclared:

- `cpsr` and `CPSR` (current program status register)
- `spsr` and `SPSR` (saved program status register).

Predeclared floating-point register names

The following floating-point register names are predeclared:

- `f0-f7` and `F0-F7` (FPA registers)
- `s0-s31` and `S0-S31` (VFP single-precision registers)
- `d0-d15` and `D0-D15` (VFP double-precision registers).

Predeclared coprocessor names

The following coprocessor names and coprocessor register names are predeclared:

- `p0-p15` (coprocessors 0-15)
- `c0-c15` (coprocessor registers 0-15).

Format of Source Line

```
{symbol} {instruction|directive|pseudo-instruction} {;comment}
```

- All 3 Portions Optional (indicated by {})
- Instructions CANNOT start in first line, must be at least 1 space
- Directives may be in upper or lower case but CANNOT mix cases
- symbol is usually a label and MUST begin in first column – cannot contain white space or tab

Labels

```
{symbol} {instruction|directive|pseudo-instruction} {;comment}
```

- Can use Upper/Lower Case or Numeric Characters or Underscore
- Can't Use Numbers as First Character
- Symbol Names are Case Sensitive
- Symbol Names must be Unique (within Scope)
- No Predefined Names Allowed

Directives

- Directives are NOT ARM Instructions
- Directives tell the Assembler how to Process the Source file
- These Notes are for the ADS Tools NOT the Keil Tools
- Frequently Used Directives in Hohl Book, p. 51
- Suggest Making Directives and Instructions in Opposite Case for Readability

Common Directives

Directive	Comment
AREA	Defines Block of data or code
RN	Equates a Register with a name
EQU	Equates a Symbol to a Numeric Constant
ENTRY	Declares an Entry Point to a Program
DCB	Allocates one or more Bytes of memory. It also specifies initial runtime contents of memory.

Common Directives

Directive	Comment
DCW	Allocates one or more Halfwords (16 bits) of memory. It also specifies initial runtime contents of memory.
DCD	Allocates one or more Words (32 bits) of memory. It also specifies initial runtime contents of memory.
ALIGN	Aligns data or code to a specific boundary
SPACE	Reserves a zeroed block of memory of a certain size.
LTORG	Assigns starting point of a literal pool.
END	Designates end of source file

ARM Instruction Set

Mnemonic	ISA Version	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
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values

ARM Instruction Set (cont)

Mnemonic	ISA Version	Description
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction

ARM Instruction Set (cont)

Mnemonic	ISA Version	Description
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLAL*xy*	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAW*y*	v5E	signed multiply accumulate instruction $(((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$

ARM Instruction Set (cont)

Mnemonic	ISA Version	Description
SMUL*xy*	v5E	signed multiply instructions $(16 \times 16 = 32\text{-bit})$
SMULW*y*	v5E	signed multiply instruction $((32 \times 16) \gg 16 = 32\text{-bit})$
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
UMULL	v3M	unsigned multiply long $(32 \times 32 = 64\text{-bit})$