03 | SUN JUNE - 2018

Microcontrollers & Embedded System.
Module-2.

Introduction to the ARM Instruction set:

- *** → Data processing Instructions.
- → Programme Instruction
- → Software Interrupt Instructions.
- → Programs Status Register Instructions
- * → Coprocessor Instructions
- → Loading Constants

04 | MON

ARM programming using Assembly language

- → Writing Assembly code.
- * → Profiling and cycle counting.
- * → Instruction scheduling.
- * → Register Allocation.
- → Conditional Execution.
- → Looping.
- → Constructs.

05 | TUE JUNE - 2018

Introduction:-

The hexadecimal numbers are represented with the prefix 0x and binary numbers with prefix 0b

```
PRE <pre-conditions>
  <Instructions>
POST <post-conditions>
```

In the pre & post-conditions, memory is denoted as mem<data_size>[address]

Example: mem 32[1024]

06 | WED

ARM instruction process data held in registers and memory is accessed only with load and store instructions

ARM instructions commonly take two or three operands

Instruction Syntax: ADD r3, r1, r2
Destination register Rd: r3.
Source register1 Rn: r1.
Source register 2 Rm: r2.

Data Processing Instructions:

The data processing instructions manipulate data within registers.

They are
* MOVE
* ARITHMETIC
* LOGICAL
* COMPARISON and
* MULTIPLY Instruction.

Most data processing instructions can process one of their operands using the barrel shifter.

The S suffix on data processing instruction

 then it updates the flags in CPSR. Move and logical operations update carry flag C, negative flag N, zero flag Z.

The C flag is set from the result of the barrel shifter as the last bit shifted out.

- N flag is set to bit 31 of the result
- Z flag is set if the result is zero.

→ MOVE Instruction:

This Instruction copies N to Rd, where N is a register or immediate value. It is useful for setting inital values and transfering

data b/w registers

syntax: <instruction>{<cond>}{S} Rd, N

MOV - Move a 32 bit value into a register
→ Rd = N.

MVN - move the NOT of the 32 bit value into a register → Rd = ~N.

Example:

```
PRE  r5=5
     r7=8
MOV  r7,r5 ; Let r7=r5
POST r5=5
     r7=5
```


Barrel shifter:-

- Data processing instructions are processed within the arithmetic logic unit (ALU)
- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source register left or right by a specific number of positions before it enters the ALU.

• Pre-processing or shift occurs within the cycle time of the instructions.

→ Shift increases the power and flexibility of data processing operations.

→ This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.

• There are data processing instructions that do not use the barrel shift, for example:

 the MUL (multiply), CLZ (count leading zeros) & QADD (signal saturated 32-bit add), instructions.

• The data flow b/w ALU & barrel shifter (Below figure).

~~use apply to logical~~

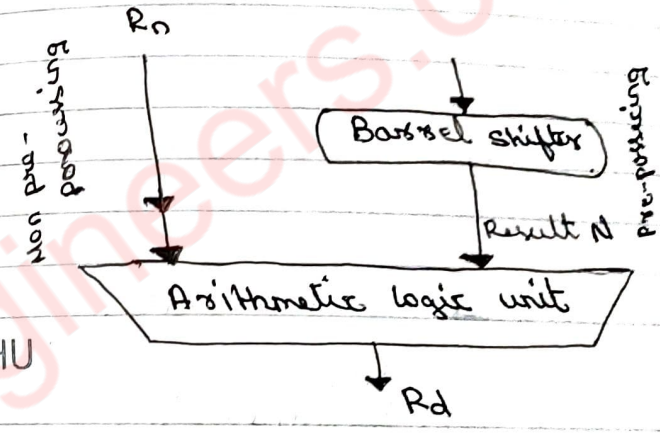
• Register R_m enters the ALU without any pre-processing of registers

• We apply LSL (logical shift left) to R_m before moving it to R_d. It is same like applying C language shift operator to the

register

• The MOV instruction copies the shift operator result N into register Rd.

N → represents the result of the LSL operation



Barrel shifter and ALU.

Barrel shifter Operations

LSL - logical shift left - xLSLy - x<<y
LSR - logical shift right - xLSRy - unsigned(x>>y)
ASR - arithmetic right shift - xASRy - (signed) x>>y
ROR - rotate right - xRORy - x>>y | x<<(32-y)
RRX - rotate right extended - xRRX
(c flag << 31) | (unsigned) x >> 1)

Example:-

PRE r5=5

r7 = 8

MOV r7, r5, LSL#2

POST

r5 = 5

r7 = 20.

Barrel shifter Operation syntax for data processing Instructions

Immediate → # immediate.

16 | SAT Register → Rm.

logical shift left by immediate → Rm, LSL # shift_imm.

Logical shift left by register → Rm, LSL Rs.

Arithmetic shift right by immediate → Rm, ASR # shift_imm

Arithmetic shift right by register → Rm, ASR Rs.

Rotate right by register → Rm, ROR Rs.

Rotate right with extend → Rm, RRX.

Arithmetic Instructions.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N.

we use three operands :- Rd - destination register other two or source operands In two source operand one must be register other can be register or immediate data.

ADC - add two 32-bit values and carry - Rd = Rn+N+Carry

ADD - add two 32-bit value - Rd = Rn+N

RSB - reverse subtract of two 32-bit values - Rd = N-Rn

RSC - reverse subtract with carry of two 32-bit values - Rd = N-Rn-!CF.

SBC - subtract with carry of two 32-bit values - Rd = Rn-N-!CF.

SUB - subtract two 32-bit => Rd = Rn-N.

The arithmetic instruction implement addition and subtraction of 32-bit signed & unsigned.

Examples:-

1) PRE r0 = 0X00000000
r1 = 0X00000002
r2 = 0X00000001
SUB r0, r1, r2
POST r0 = 0X00000001

2) PRE r0 = 0X00000000
r1 = 0X00000077
RSB r0, r1, #0
POST r0 = -r1 = 0XFFFFFF89

3) PRE cpsr = nzcvqiFt_USER
r1 = 0X00000001
SUBS r1, r1, #1 ; r1=r1-1
POST cpsr = nzcvqiFt_USER
r1 = 0X00000000

Logical Instructions:-

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

Logical instructions perform bitwise logical operations on the two source registers.

AND - logical bitwise AND of two 32-bit values => Rd = Rn & N.

ORR - logical bitwise OR of two 32-bit values => Rd = Rn | N.

EOR - logical exclusive OR of two 32-bit values => Rd = Rn ^ N.

BIC - logical bit clear (AND NOT) => Rd = Rn & ~N.

Example:- PRE r0 = 0X00000000
r1 = 0X02040608
r2 = 0X10305070
ORR r0, r1, r2
POST r0 = 0X12345678

BIC Instruction:- (Bit clear instruction)

=> It is combination of AND and NOT

Syntax: BIC {<cond>}{S} Rd, Rn, N.

Example: PRE r1 = 0b1111
r2 = 0b1010
BIC r0, r1, r2
POST r0 = 0b0101
This is equivalent to
Rd = Rn AND NOT(N).

Purpose of BIC instruction :-

Example :- (binary) (hexadecimal form)

r1 = 0b 1100 r1 = 0XC

r2 = 0b 1000

BIC r0, r1, r2

r0 = 0b 0100 r0 = 0X4 = 4

So BIC instruction is used to clear (i.e to ~~make~~ make zero) particular bit in a source data.

Comparison Instructions

Syntax : <instruction>{<cond>}Rn, N

CMN – compare negated - flags set as a result of Rn + N

CMP – Compare - flags set as a result of Rn - N

TEQ - Test for equality of two 32-bit values - flags set as a result of Rn ^ N

TST – test bits of a 32-bit value - flags set as a result of Rn & N.

The compare instructions are used to compare or test a register with a 32-bit value. They update the cpsr flag bits according to the result, but do not affect other register. It is not required to apply the S suffix for comparison instruction to update the flag.

Example :-

CMP{<cond>} Rn, N

PRE

cpsr = nzcvqiFt_USER

r1 = 20

r2 = 30

CMP r1, r2

POST

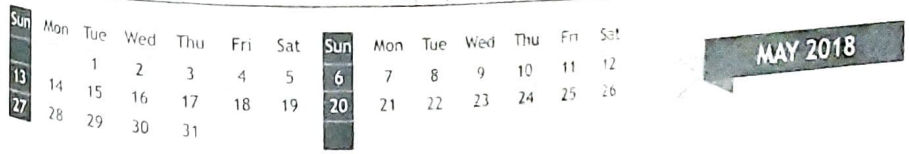
cpsr = NzcvqiFt_USER

Working

CMP means subtraction take place

Observe the above example

r1 - r2 => 20 - 30 = -10 so conditional code is NzcvqiFt_USER.



Example 2:

N	Z	C	V
0	1	0	0

PRE

CPSR = nZCvqiFt_USER

R1 = 30

R2 = 30

CMP R1, R2

POST

CPSR = nZCvqiFt_USER.

Multiply Instruction (32 bit):

Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn

MUL{<cond>}{S} Rd, Rm, Rs

MLA - multiply and accumulate → Rd = (Rm * Rs) + Rn

MUL - multiply → Rd = Rm * Rs

Multiply instructions multiply the contents of a pair of registers and depending upon the instruction, accumulate in register.

Example:

PRE R0 = 0x00000000

R1 = 0x00000002

R2 = 0x00000002

MUL R0, R1, R2 ; R0 = R1 * R2

POST

R0 = 0x00000004

R1 = 0x00000002

R2 = 0x00000002

Multiplication of 64 bit.

Syntax: <instruction>{<cond>} RdLO, RdHi, Rm, Rs.

SMLAL - Signed multiply accumulate long
→ [RdHi, RdLO] = [RdHi, RdLO] + (Rm * Rs).

SMULL - Signed multiply long
→ [RdHi, RdLO] = Rm * Rs.

UMLAL - Unsigned multiply accumulate long
→ [RdHi, RdLO] = [RdHi, RdLO] + (Rm * Rs).

UMULL - unsigned multiply long
→ [RdHi, RdLO] = Rm * Rs.

Note

The number of cycles taken to execute a multiply instruction depends on the processor implementation.

For some implementation the cycle timing also depends on the value of Rs.

Example:

The instruction multiplies registers r2 & r3 & places the result into register r0 & r1.

Register r0 contains the lower 32-bits, & register r1 contains the higher 32 bits of the 64-bit result.

```
PRE   r0 = 0x00000000
      r1 = 0x00000000
      r2 = 0xf0000002
      r3 = 0x00000002
UMULL r0, r1, r2, r3 ; [r1, r0] = r2 * r3
POST
      r0 = 0xe0000004 ; = RdLO
      r1 = 0x00000001 ; = RdHi
```


01 | SUN

JULY - 2018

The long multiply instruction produce a 64-bit result. The result is too large to fit a single 32-bit register so result is placed in two registers namely

RdHi - holds higher 32-bits

RdLO - holds lower 32-bits

∴ 32 + 32 bits ⇒ 64 bits.

Refer Textbook for more Examples:-

02 | MON

* Branch Instructions :-

There are 4 branch Instruction
- B → Branch
- BL → Branch with link
- BX → Branch exchange
- BLX → Branch exchange with link

→ A branch instruction changes the flow of execution or is used to call a routine
→ Branch instruction allows programmer
- to skip set of instructions from execution
- to repeat execution of set of instruction

→ Branching changes content of PC register.

→ A branch can be conditional or unconditional
→ Conditional branches get executed only when condition is true.

Syntax:

B{<cond>} label
BL{<cond>} label
BX{<cond>} Rm
BLX{<cond>} label/Rm

Conditional is optional.

B → Branch — PC = label

BL → Branch with link → PC = label

lr = address of the next instruction after the BL

(lr = link register).

BX → Branch exchange

→ PC = Rm & 0xfffffffe, T = Rm & 1

BLX → Branch exchange with link

→ PC = label, T = 1

→ PC = Rm & 0xfffffffe, T = Rm & 1

→ lr = address of the next instruction after BL

where.

T refers to Thumb bit in cpsr.

When instructions set T, the ARM switches to thumb state.

Example for forward and Backward Branches:

1) B forward

```
ADD r1, r2, #4
ADD r0, r6, #2
ADD r3, r7, #4
forward
SUB r1, r2, #4
```

2) backward

```
ADD r1, r2, #4
SUB r1, r2, #4
ADD r4, r6, r7
B backward.
```

forward & Backward are labels.

↳ it skips three instructions.

2015 backward branch forms infinite loop.

BL instruction is similar to B instruction but overwrites the link register (lr) with return address. It performs subroutine call.

Example:

```
BL      subroutine      ; branch to subroutine
CMP r1, #5              ; compare r1 with 5
MOVEQ r1, #0            ; if (r1 == 5) then r1 = 0.
  :
  :
subroutine
<subroutine code>
MOV pc, lr              ; return by moving pc = lr
```

BX & BLX are third type of Branch instruction

BX uses an absolute address stored in register Rm. It primarily used to branch to and from Thumb code. The T bit in the cpsr is updated by the least significant bit of branch register.

BLX is same as BX additionally set the link register with the return address.

Load - Store Instructions.

Types

→ Single - register transfer, → mutiple - register transfer, → swap.

- Single - Register Transfer:

→ These instructions are used for moving a single data item in and out of a register.

→ The data types supported are signed & unsigned, half-words (16-bit) and bytes.

Syntax: <LDR|STR>{<cond>}{B} Rd, addressing[1]
LDR{<cond>}SB|H|SH Rd, addressing[2]
STR{<cond>}H Rd, addressing[2]

LDR - load word into register → Rd ← mem32[address]
STR - Save byte or word from a register
→ Rd → mem32[address]
LDRB - load byte into a register → Rd ← mem8[addr]
STRB - Save byte from a register
→ Rd → mem8[address].
LDRH - load halfword into a register
→ Rd ← mem16[address]
STRH - Save halfword into a register
→ Rd → mem16[address].

LDRSB – load signal byte into register
Rd ← Sign Extend (mem8[address]).

LDRSH – load signed halfword into a register
→ Rd ← Sign Extend (mem16[address]).

LDR & STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.

Example:-

LDR r0,[r1] ; =LDR r0,[r1,#0]

Load register r0 with the contents of the memory address pointed to by register r1.

STR r0,[r1] ; = STR r0,[r1,#0]

Store the contents of register r0 to the memory address pointed by r1.

29 Sunday

Register r1 is called base address register.

Single - Register Load - store addressing Modes:

There are three types of load - store instruction

- pre - index with write back
- pre - index
- post - index

→ Indexing addressing methods.

Index methods:

Index method	Data	Base address register	Example
Preindex with writeback	mem[base + offset]	base + offset	LDR r0, [r1, #4]!
pre index	mem[base + offset]	not updated	LDR r0, [r1, #4]
post index	mem[base]	base + offset	LDR, r0, [r1], #4

pre index with writeback calculates an address from a base register plus address offset & then updates that address base register with the new address.

Pre index is same as pre index writeback but does not update the address base register.

Post index only updates the address base register after the address is used.

Example:-

PRE r0 = 0x00000000
r1 = 0x00090000
mem 32[0x00009000] = 0x01010101
mem 32[0x00009004] = 0x02020202
LDR r0, [r1, #4]!

Pre indexing with writeback
POST (1) r0 = 0x02020202
r1 = 0x00009004
LDR r0, [r1, #4]

Pre-indexing
POST (2) r0 = 0x02020202
r1 = 0x00009000
LDR r0, [r1, #4]

Post-indexing
POST (3) r0 = 0x01010101
r1 = 0x00009004

8
Wednesday
July

189-176 • WK 28

JUNE 2015
M T W T F S S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

Addressing mode for single-Register:

Addressing mode and index method	Addressing syntax
Pre-index with immediate offset	[Rn, # +/- offset_12]
Preindex with register offset	[Rn, +/- Rm]
Preindex with scaled register offset	[Rn, +/- Rm, → shift # shift_imm]
Pre-index writeback with immediate offset	[Rn, # +/- offset_12]!
Pre index writeback with register offset	[Rn, +/- Rm]!
Immediate postindexed	[Rn], # +/- Offset_12
Register post index	[Rn], +/- Rm
Scaled register postindex	[Rn] +/- Rm, shift # shift_imm

- A signed offset or register is denoted by +/- identifying that it is either a positive or negative offset from the base address Register R_n.

- Immediate means the address is calculated using the base address register & a 12-bit offset encoded in the instruction.

- Scaled means the address is calculated using the base address register & a barrel shift operation.

- Multiple - Register Transfer

→ It can transfer data b/w multiple register & memory in a single instruction.

→ efficient from single-register transfers for moving blocks of data, saving and restoring context & stacks.

→ It can increase interrupt latency

→ Transfer occurs from a base address register R_n pointing into memory

10 Friday July

191-174 • WK 28

JUNE 2015

M	T	W	T	F	S
1	2	3	4	5	6
8	9	10	11	12	13
15	16	17	18	19	20
22	23	24	25	26	27
29	30				

Syntax : <LDM/STM>{<cond>} <addressing mode> Rn{!}, <registers>{^}

LDM – load multiple register
– {Rd}^N ← mem32[start address + 4*N]

STM – Save multiple register
– {Rd}^N → mem32[start address + 4*N]

where, N – Number of registers in the list of register.

Addressing mode for Load-store Multiple Instruction:

Addressing mode	→ Description	→ start address	→ end address	→ Rn!
IA	→ Increment after	→ Rn	→ Rn+4*N-4	→ Rn+4*N
IB	→ Increment before	→ Rn+4	→ Rn+4*N	→ Rn+4*N
DA	→ decrement after	→ Rn-4*N+4	→ Rn	→ Rn-4*N
DB	→ decrement before	→ Rn-4*N	→ Rn-4	→ Rn-4*N

- Any subset of the current bank of register can be transferred to memory or fetched from memory.
- Rn - determines the source or destination address for a load-store multiple instruction.

Example :-

PRE mem32[0x80018] = 0x03
mem32[0x80014] = 0x02
mem32[0x80010] = 0x01
r0 = 0x00080010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000

LDMIA r0!,{r1-r3}
POST r0 = 0x0008001c
r1 = 0x00000001
r2 = 0x00000002
r3 = 0x00000003.

13 Monday July

194 171 • WK 29

JUNE 20

M	T	W	T	F	S	S
1	2	3	4	5	6	
8	9	10	11	12	13	
15	16	17	18	19	20	
22	23	24	25	26	27	
29	30					

Stack Operation :-

- The pop operation uses a load- ~~store~~ multiple instructions.
- The push operation uses a store multiple instruction.

When using a stack you have to decide whether the stack will grow up or down in memory

A stack is either -

ascending (A) - grows towards higher memory address

descending (D) - grows towards lower memory address.

When you use a full stack (F), the stack pointer SP points to an address that is the last used for full location.

If we an Empty stack (E) the SP points to an address that is the ~~last~~ first unused or empty location.

Addressing methods for stack operation.

Addressing modes	Description	POP	LDM	Push	STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMD
FD	full descending	LDMFD	LDMIA	STMFD	STMI
EA	empty ascending	LDMEA	LDMDB	STMEA	STMI
ED	empty descending	LDMED	LDMIB	STMED	STMD

2015

Example:- Using STMED.

PRE $r1 = 0x00000002$
$r4 = 0x00000003$
$SP = 0x00080010$
STMED $sp!, \{r1, r4\}$
POST $r1 = 0x00000002$
$r4 = 0x00000003$
$SP = 0x00080008$

When handling a checked stack there are three attributes that need to be preserved: the stack base, stack pointer, stack limit.

* Swap Instruction.

Syntax: SWP{B}{<cond>} Rd, Rm, [Rn]

SWP - swap a word b/w memory and a register

tmp = mem32[Rn]
mem32[Rn] = Rm
Rd = tmp

SWPB - swap a byte b/w memory and a register

tmp = mem8[Rn]
mem8[Rn] = Rm
Rd = tmp

→ It is a special-case in load-store instruction
→ It swaps the contents of memory with the contents of a register.

→ It is an atomic operation - it reads & writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until till completes.

Example:-

PRE mem 32 [0x9000] = 0x12345678
R0 = 0x00000000
R1 = 0x11112222
R2 = 0x00009000
SWP R0, R1, [R2]

POST mem 32 [0x9000] = 0x11112222
R0 = 0x12345678
R1 = 0x11112222
R2 = 0x00009000

17
Friday
July
198-167 • Wk 29

JUNE 20
M T W T F
1 2 3 4 5
8 9 10 11 12
15 16 17 18 19
22 23 24 25 26
29 30

Software Interrupt Instruction.

A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for application to call OS routines

Syntax : SWI{<cond>} SWI_number.

SWI – Software interrupt – lr_svc = address of instruction following SWI

SPSR_SVC = CPSR
PC = vectors + 0x8
CPSR mode = SVC
CPSR I = 1 (mask IRQ interrupts).

example :- PRE
CPSR = nzcVqift_USER
PC = 0x00008000
lr = 0x003fffff ; lr = r14
r0 = 0x12
0x00008000 SWI 0x123456
POST
CPSR = nzcVqIft_SVC
SPSR = nzcVqift_USER
PC = 0x00000008
lr = 0x00008004
r0 = 0x12.

2015

When the processor executes an SWI instruction, it sets the program counter PC to the offset 0x8 in the vector table. The instruction also forces the processor mode to SVC, which allows an OS routine to be called in a privileged mode.

Since SWI instructions are used to call OS routines, you need some form of parameter passing, This is achieved using registers.

Program Status Register Instruction.

The ARM instruction set provides two instructions to directly control a program status register

→ The MSR instruction transfers the contents of either the cpsr or spsr into register

→ The MSR instruction transfers the contents of a register into the cpsr or spsr

→ Together these instructions are used to read & write the cpsr and spsr.

Syntax: MRS{<cond>} Rd, <cpsr|spsr>
MSR{<cond>} <cpsr|spsr> - <fields>, Rm
MSR{<cond>} <cpsr|spsr> - <fields>, #immediate.

* Coprocessor Instructions:

Coprocessor instruction are used to extend the instruction set.

- A coprocessor can either provide additional computation capability. or be used to control the memory subsystem including caches and memory management.
- A coprocessor instructions include data processing, register transfer, and memory transfer instructions.

Syntax: CDP{<cond>} cp, opcode1, Cd, Cn{opcode2}
<MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm
{, opcode2}
<LDC|STC>{<cond>} cp, Cd, addressing.

CDP - coprocessor data processing - perform an operation in a coprocessor.

MRC MCR - coprocessor register transfer - move data to/from coprocessor register.

LDC STC → coprocessor memory transfer - load and store blocks of memory to/from a coprocessor.

- In the syntax of the coprocessor instructions.

→ The Cp field represents the coprocessor number blw p0 and p15.
→ The opcode fields describes the operation to take place on the coprocessor.
→ The Cn, Cm and Cd fields describe registers within the coprocessor.

- The coprocessor operations and registers depends on the specific coprocessor you are using.

- Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control and identification registers

Example :-

CP15 register being copied into a general - purpose register.

MRC p15, 0, r10, c0, c0, 0.

LOADING CONSTANTS.

Syntax: LDR Rd, = constant
ADR Rd, label.

LDR - load constant pseudo instruction
Rd = 32-bit constant.

ADR - load address pseudo instruction
Rd = 32-bit relative address.

There is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant. To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

⇒ The first pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.

⇒ The second pseudo-instruction writes a relative address into a register, which will be encoded using a PC-relative expression.

Example:-
LDR instruction loading 32-bit constant 0xff00ffff into register r0.

```
LDR r0, [PC, # constant_number - 8 - {PC}]
  :
  :
constant_number
  DCD  0xff00ffff.
```

ARM programming Using Assembly Language:

Introduction:-

Writing assembly by hand gives you direct control of three optimization tools that you cannot explicitly use of writing C source.

Instruction scheduling:- Reordering the instruction in a code sequence to avoid processor stalls. Since ARM implementations are pipelined, the timing of an instruction can be affected by neighbouring instruction.

Register allocation :- Deciding how variables should be allocated to ARM registers & or stack locations for maximum performance.

Conditional execution :- Accessing the full range of ARM condition code & conditional instructions.

Writing Assembly Code.

Converting C program to assembly code.

```
# include <stdio.h>
 int square (int i)
 int main (void)
   {
      int i;
      for (i=0; i<10; i++)
        {
          printf(" square of %d\n", i, square(i));
        }
   }
   int square (int i)
     {
       return i*i;
     }
```

Assembly program.

```
AREA |.text|, CODE, READONLY

  EXPORT square

square
        MUL  r1, r0, r0
        MOV  r0, r1
        MOV  pc, lr
        END
```

Area - it is directive names the area or code section that the code lives in. In previous code we define a read-only code area called .text.

EXPORT – directive makes the symbol square available for external linking.

END - directive marks the end of the assembly file.

Example:-
When calling ARM code from C compiled as thumb, the only change required is to change the return instruction to a BX.

```
Area |.text|, CODE, READONLY

    EXPORT  square

square

    MUL   r1, r0, r0
    MOV   r0, r1
    BX    lr
    END.
```

Profiling and Cycle Counting.

The first stage of any optimization process is to identify the critical routines and measure there current performance. A profilier is a tool that measures the proportion of time or processing cycle spent in each subroutine. You use a profiler to identify the most critical routines. A cycle counter measures the number of cycles taken by a specific routine. You can measure your sucess by using a cycle counter to benchmark a given subroutine before and after an optimization.

The ARM simulator used by the ADS.
1. debugger is called the ARMulator and provides profiling and cycle counting features.

1 Saturday August 213-152 • WK 31

The ARMulator profiler works by sampling the program counter ~~of~~ ~~cycles~~ PC at regular intervals.

The profiler identifies the function the PC points to and updates a hit counter for each function it encounters.

ARM implementation do not normally contain cycle-counting hardware, so to easily measure cycle counts, you should use an ARM debugger with ARM simulator.

INSTRUCTION SCHEDULING.

The time taken to execute instructions depends on the implementation pipeline.

Instructions that are conditional on the value of the ARM condition codes in the cpsr take one cycle if the condition is not met.

2 Sunday If the condition is met, then following rules apply:-

- ALU operations such as addition, subtraction & logical operation take place in one cycle. This include a shift by an immediate value. If you use a register - shift, then add one cycle. If the instruction writes to pc, then 2 cycle.

2015

- Load instructions that load N 32-bit words of memory such as LDR & LDM take N cycles to issue, but the result of the last word loaded is not available on the following cycle. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an uncached system or a cache hit for a cached system. An LDM of a single value is exceptional, taking two cycles. If the instruction loads pc, then add two cycles.

- Load instructions that load 16-bit or 8-bit data such as LDRB, LDRSB, LDRH & LDRSH take one-cycle to issue. The updated load address is available on the next cycle.

- Branch instructions takes 3 cycles.

- Store instructions that store N values take N cycles. This assumes zero-wait-state memory for an uncached system, or a cache hit or write buffer with N free entries for a cached system.

- Multiply instructions take a varying number of cycles depending on the value of the second operand.

4
Tuesday
August

216-149 • WK 32

JULY 20
M T W T F S
1 2 3 4
6 7 8 9 10 11
13 14 15 16 17 18
20 21 22 23 24 25
27 28 29 30 31

ARM9TDMI pipeline executing in ARM state.

- The ARM9TDMI processor five operations in parallel.

→ Fetch → Fetch from memory the instruction at address pc. The instruction is loaded into the core and then processed down the core pipeline.

→ Decode → Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if are not available via one of the forwarding paths.

→ ALU → Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address pc-8 (ARM state) or pc-4 (Thumb state). Normally this involves calculating the answer for a data processing operation, or the address for a load, store or branch operations. Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations takes several ALU cycles.

2015

→ LS1: Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.

→ LS2: Extract and zero- or sign-extend the data loaded by a byte or halfword load instruction. If the instruction is not a load of an 8-bit byte or 16-bit halfword item, then this stage has no effect.

Example:-

This example shows the case where there is no interlock

```
ADD   r0,r0,r1
ADD   r0,r0,r2
```

This instruction pair take two cycles. The ALU calculate r0 + r1 in one cycle. Therefore this result is available for the ALU to calculate r0 + r2 in the second cycle.

Scheduling of load instructions:

Load instructions occur frequently in compiled code, accounting for approximately one third of all instructions. Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance. The compiler attempts to schedule the code as best it can, but the aliasing problems of C that we looked at in Section 5.6 limits the available optimizations. The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.

Example of a memory-intensive task. The following function, str_tolower, copies a zero-terminated string of characters from in to out. It converts the string to lowercase in the process.

```
void str_tolower(char *out, char *in)
{
    unsigned int c;
    do
    {
        c = *(in++)
        if(c >= 'A' && c <= 'Z')
        {
            c = c + ('a' - 'A');
        }
```

```
    *(out++) = (char) c
  } while (c);
}
```

The ADS1.1 compiler generates the following compiled output. Notice that the compiler optimizes the condition (c >= 'A' && c <= 'z') to the check that 0 <= c - 'A' <= 'z' - 'A'. The compiler can perform this check using a single unsigned comparison.

```
str_tolower

    LDRB   r2, [r1], #1
    SUB    r3, r2, #0x41
    CMP    r3, #0x19
    ADDLS  r2, r2, #0x20
    STRB   r2, [r0], #1
    CMP    r2, #0
    BNE    str_tolower
    MOV    pc, r14.
```

Unfortunately, the SUB instruction uses the value of c directly after the LDRB instruction that loads c, consequently, the ARM9TDMI pipeline will stall for two cycles. The compiler can't do any better since everything following the load of c depends on its value. However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly. We call these methods load scheduling by preloading and unrolling.

8 Saturday August 220-145 • WK 32

Load Scheduling by preloading:

In this method of load scheduling, we load the data required for the loop at the end of the previous loop, rather than at the beginning of the current loop. To get performance improvement with little increases in code size, we don't unroll the loop.

Example:-

This assembly applies the preload method to the str_tolower function.

```
out     RN 0
in      RN 1
c       RN 2
t       RN 3
        ;

loop
        SUB   t, c, #'A'
        CMP   t, #'Z' - 'A'
        ADDLS c, c, #'a' - 'A'
        STRB  c, [out], #1
        TEQ   c, [in], #1
        BNE   loop
        MOV   pc, lr
```

9 Sunday

The scheduled version is one instruction longer than the C version, but we save two cycles for each inner loop iteration. This reduces the loop from 11 cycles per character to 9 cycles per character on an ARM9TDMI, giving a 1.22 times speed improvement.

The ARM architecture is particularly well suited to this type of preloading because instructions can be executed conditionally. Since loop i loading the data for loop-i+1 there is always a problem with the first & last loop. No byte load occurs on the last loop.

Load Scheduling by Unrolling.

This method of load scheduling works by unrolling and then interleaving the body of the loop.

for example:- we can perform loop iterations i, i+1, i+2 interleaved. When the result of an operation from loop i is not ready, we can perform an operation from loop i+1 that avoids waiting for the loop i result.

This loop is the most efficient implementation we've looked at so far. The implementation requires seven cycles per character on ARM9TDMI. This gives a 1.57 times speed increase over the original STR loader.

Again it is the conditional nature of the ARM instruct
that makes this possible. We use conditional instructi
to avoid storing characters that are past the
end of the string.

However, the improvement in Example 1.10 does
have some costs. The routine is more than double
the code size of the original implementation.
We have assumed that you can read up to
two characters beyond the end of the input string.
which may not be true if the string is right at
the end of available RAM. where reading off the end
will cause a data abort. Also performance can be
slower for very short strings because (1) stacking
& causes additional function call overhead &
(2) the routine may process up two characters
pointlessly, before discovering that they lie
beyond the end of the string.

You should use this form of scheduling
by unrolling for time-critical parts of an
application where you know the data size is large
If you also know the size of the size of the
data at compile time, you can remove the problem
of reading beyond the end of the array.

```
out     RN 0 ;
in      RN 1 ;
ca0     RN 2 ;
t       RN 3 ;
ca1     RN 12;
ca2     RN 14;
        ;void str_tolower_unrolled(char *out, char *in)
        str_tolower_unrolled
        STMFD  sp!, {lr}    ;
loop_next3
        LDRB  ca0, [in], #1
        LDRB  ca1, [in], #1
        LDRB  ca2, [in], #1
        SUB   t, ca0, #'A'
        CMP   t, #'z' - 'A'
ADDLS   ca1, ca1, #'a' - 'A'
        SUB   t, ca2, #'A'
        CMP   t, #'z' - 'A'
ADDLS   ca2, ca2, #'a' - 'A'
        STRB  ca0, [out], #1
        TEQ   ca0, #0
        STRNEB  ca1, [out], #1
        TEQNE   ca1, #0
        STRNEB  ca2, [out], #1
        TEQNE   ca2, #0
        BNE     loop_next3
        LDMFD   sp!, {pc}
```

REGISTER ALLOCATION:-

We can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the stack pointer r13 and the program counter r15. For a function to be ATPCS compliant it must preserve the callee values of registers r4 to r11. ATPCS also specifies that the stack should be eight-byte aligned. Therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers.

```
routine_name
        STMFD sp!, {r4-r12, lr}
        LDMFD sp!, {r4-r12, pc}
```

Our only purpose in stacking r12 is to keep the stack eight-byte aligned. You need not stack r12 if your routine doesn't call other ATPCS routines. For ARMv5 and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an ARMv4T processor, then modify the template as follows:

routine_name
STMFD sp!, {r4-r12, lr}
; body of routine
LDMFD sp! {r4-r12, lr}
BX lr

In this section we look at how best to allocate variables to register numbers for register intensive tasks, how to use more than 14 local variables, & how to make the best use of the 14 available registers.

Allocating Variables to Register Numbers:

When we write an assembly routine, it is best to start by using names for variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.

For the most part ARM operations are Orthogonal with respect to register numbers. In other words, specific register numbers do not have specific roles.

• Argument registers. The ATPCS convention defines that the first four arguments to a function are placed in registers r0 to r3. Further arguments are placed on the stack. The return value must be placed in r0.

• Registers used in a load or store multiple. Load & store multiple instructions LDM and STM operate on a list of registers in order of ascending register number. If r0 and r1 appear in the register list, then the processor will always load or store r0 using a lower address than r1 & so on...

• Load and store double word. The LDRD & STRD instructions introduced in ARMv5E operate on a pair of registers with sequential register numbers, Rd & Rd+1. Futhermore, Rd must be an even register number.

16 Sunday

Example:-

The assembly shows our final shift-bits routine. It uses all 14 available ARM registers

Kr RN lr
shift_bits

```
        STMFD   sp!,{r4-r11, lr}
        RSB     Kr, K, #32
        MOV     y_0, #0
loop
        LDMIA   in!,{x_0-x_7}
        ORR     y_0, y_0, x_0, LSL K
        MOV     y_1, x_0, LSR Kr
        ORR     y_1, y_1, x_1, LSL K
        MOV     y_2, x_1, LSR Kr
        ORR     y_2, y_2, x_2, LSL K
        MOV     y_2, x_1, LSR Kr
        ORR     y_2, y_2, x_2, LSL K
        MOV     y_3, x_2, LSR Kr
        ORR     y_3, y_3, x_3, LSL K
        MOV     y_4, x_3, [illegible] LSR Kr
        ORR     y_4, y_4, x_4, LSL K
        MOV     y_5, x_4, LSR Kr
        ORR     y_5, y_5, x_5, LSL K
        MOV     y_6, x_5, LSR Kr
        ORR     y_6, y_6, x_6, LSL K
        MOV     y_7, x_6, LSR Kr
        ORR     y_7, y_7, x_7, LSL K
        STMIA   out!,{y_0-y_7}
        MOV     y_0, x_7, LSR Kr
        SUBS    N, N, #256
        BNE     loop
        MOV     r0, y_0
        LDMFD   sp!{r4-r11, pc}
```

2015

Using More than 14 Local Variables.

If you need more than 14 local 32-bit variables, then you must store some variables on the stack. The standard procedure is to work outwards from the inner-most loop of the algorithm, since the innermost loop has the greatest performance impact.

Example:

```
nested_loops
        STMFD  sp! {r4-r11, lr}
loop1
        STMFD  sp!, {loop1 registers}
loop2
        STMFD  sp!, {loop2 registers}
loop3
        B{cond} loop3
        LDMFD  sp!, {loop2 registers}
        B{cond} loop2
        LDMFD  sp!, {loop1 registers}
        B{cond} loop1
        LDMFD  sp!, {r4-r11, pc}.
```

CONDITIONAL EXECUTION:

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one 15 conditional codes, If you don't specify a condition the assembler defaults to execute always condition(AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four conditional flags N, Z, C, V stored in the cpsr register.

By default, ARM instructions do not update the N, Z, C, V flags in the ARM cpsr. For most instructions, to update these flags you append an S suffix to the instruction mnemonic. Exceptions to this are comparison instructions that do not write to a ~~determine~~ destination register. Their sole purpose is to update the flags and so they don't require the S suffix.

By combining conditional execution and conditional setting of flags, you can implement simple if statement without need of Branches

Example:- Assembly using conditional execution

```
CMP  i,#10
ADDLO  c,i,#'0'
ADDHS  c,i,#'A'-10.
```

Conditional execution is even more powerful for cascading conditions.

Example: 2 . Identifies if c is vowel

```
TEQ   c, #'a'
TEQNE c, #'e'
TEQNE c, #'i'
TEQNE c, #'o'
TEQNE c, #'u'
ADDEQ vowel, vowel, #1
```

As soon as one of the TEQ comparisons detects a match, the z flag is set in the cpsr. The following TEQNE instructions have no effect as they are conditional on Z=0.

The next instruction to have effect is the ADDEQ that increments vowel.

LOOPING CONSTRUTS.

Most routines critical to performance will contain a loop. We saw in instruction scheduling that on the ARM loops are fatest when they count down towards zero. This section describes how to implement these loops efficiently in assembly.

Decremented Counted Loops:

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with i=0. An efficient implementation is

```
    MOV i,N
loop
    ;loops body goes here & i=N, N-1 ....
    SUBS i,i,#1
    BGT loop.
```

The loop overhead consits of a subtraction setting the condition codes followed by a conditional branch. On ARM7 & ARM9 this overhead costs four cycle per loop. If i is an array index, then you may want to count down from N-1 to 0 inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch.

```
    SUBS i,N,#1
loop
    SUBS i,i,#1
    BGE loop
```

Unrolled Counted Loops:

Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome.

Function sets N bytes of memory at address to the byte value c. The function needs to be efficient, so we will look at how to unroll the loop without placing extra restrictions on the input operands.

Void my-memset (char *s, int c, unsigned int N)

To be efficient for large N, we need to write multiple bytes at a time using STR or STM instructions. Therefore our first ~~to~~ task is to align the array pointers. However, it is only worth us doing this if N is sufficiently large. Let's assume we can choose a threshold value T1 & only bother to align the array when $N \geq T1$. Clearly $T1 \geq 3$ as there is no point in aligning if we don't have four bytes to write!

Now suppose we have aligned the array s. We can use store multiples ~~of eight words to set~~ to set memory efficiently. Example:- we can use a loop of four store multiples of eight words each to set 128 bytes on each loop. However, it will only be worth doing this if $N \geq T2 \geq 128$, where

T2 is another threshold to be determined later on. Finally, we are left with N<T2 bytes to set. We can write bytes in blocks of four using STR until N<4. Then we can finish by writing bytes singly with STRB to the end of the array.

Multiple Nested Loops:

Example:- The matrix using this single counter in register count:

```
R     EQU 40
S     EQU 40
T     EQU 40
a     RN 0  ; Points to an rows x columns matrix
b     RN 1  ;  ———— 11 ————
c     RN 2  ;  ———— 11 ————
sum   RN 3
bval  RN 4
cval  RN 12
count RN 14

matrix_mul
        STMFD sp!, {r4, lr}
        MOV   count, #(R-1)

loopi
        ADD   count, count, #(T-1)<<8
```

loop j

```
ADD     count, count, #(S-1)<<16
MOV     sum, #0
```

loop K

```
LDR     bval, [b], #4
LDR     cval, [c], #4*T
[illegible]
SUBS    count, count, #1<<16
MLA     sum, bval, cval, sum
BPL     loop_K
STR     sum, [a], #4
SUB     c, c, #4*S*T
ADD     C, C, #4
ADDS    count, count, #(1<<16)-(1<<8)
SUBPL   b, b, #4*S
BPL     loop_j
SUB     c, c, #4*T
ADDS    count, count, #(1>>8)-1
BPL     loop_i
LDMFD   sp! {r4, pc}
```

Here as you observe the assembly program you can see 3 loops are use loopi, loopj, loopk.

So we can say it is a nested loop.

Negative indexing.

This loop structure counts from -N to 0 (inclusive or exclusive) in steps of size STEP

```
        RSB   i,N,#0
loop
        ADDS  i,i,#STEP
        BLT   loop.
```

Logarithmic Indexing

This loop structure count down from 2^N to 1 powers of two.

For example. if N=4 then it counts 16,8,4,2,1.

```
        MOV   i,#1
        MOV   i,i,LSL N
loop
        MOVS  i,i,LSR#1
        BNE   loop.
```

Programs:

1. Demonstrating logical operation

```
AREA LOGIC, CODE, READONLY
ENTRY
 LDR  R0,=5
 LDR  R1,=3
 AND  R4,R0,R1
 ORR  R5,R0,R1
 EOR  R6,R0,R1
 BIC  R7,R0,R1
END.
```

2. find factorial of a given number. (subroutine).

```
 AREA FACTORIAL, CODE, READONLY
 ENTRY
START  LDR R0,#5
       BL FACT
       LDR R4,=DST
       STR R5,[R4]
STOP   B STOP
FACT
       MOVS R1,R0
       MOVEQ R5,#1
LOOP
```

30 Sunday

```
SUBNES  R1,R1,#1
MULNE   R0,R1,R0
BNE     LOOP
MOV     R5,R0
MOV     PC,R14
AREA  FACT, DATA, READWRITE
DST     DCD 0
END.
```

3. To find Fibonacci series

```
AREA  FIB, CODE, READONLY
ENTRY
MOV R0,#00
SUB R0,R0,#01
MOV R1,#01
MOV R4,#05
LDR R2,=FIB0
BACK ADD R0,R1
STR R0,[R2]
ADD R2,#04
MOV R3,R0
MOV R0,R1
MOV R1,R3
CMP R4,#00
BNE BACK
STOP B STOP
AREA FIBONACCI, DATA, READWRITE
END.
```