

MICROPROCESSOR AND COMPUTER ARCHITECTURE

UNIT-1

basic Processor
architecture & design

introduction

TEXTBOOKS

1. "Computer Organization and Design" - Patterson, Hennessy,
5th edition
2. "ARM System on a Chip" - Steve Furber , 2nd edition

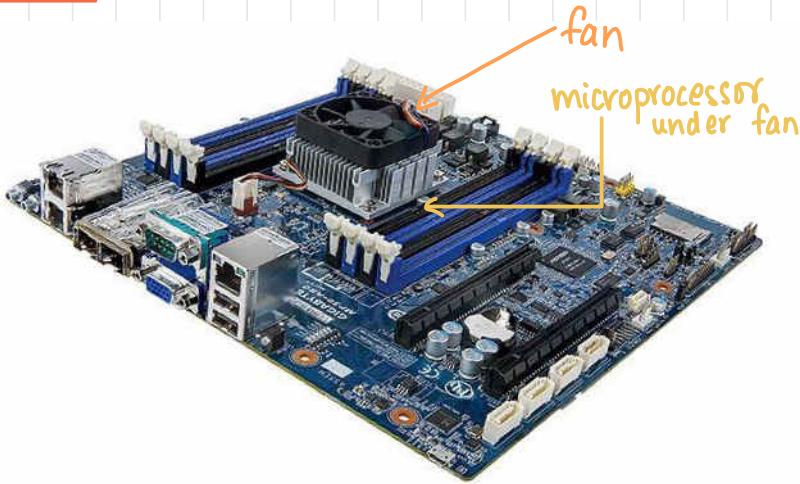
MpCA

1. Programmer
2. Operating System : resource manager
3. Compiler : converts high level to machine lang (irrespective of system)
4. Hardware (Processor, I/O, Memory) : computation,

MICROPROCESSOR

- Single chip implementation of CPU
- Not all microprocessors are CPUs
 - GPU (image processing)
 - TPU (tensors, matrix multiplication for image processing)
 - NPU (neural nets, ML)
- Multipurpose programmable Devices
- Multi-core processor: core has processing unit, registers, cache; receives instructions from single computing task

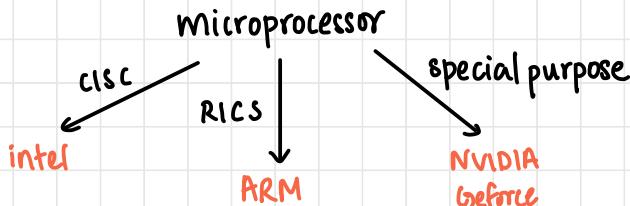
Motherboard



Evolution of Intel Microprocessors

Intel Pentium Dual-Core	2006 - 2009
Intel Pentium (2009)	2009–present
Intel Core	2006 - 2008
Intel Core 2	2006 - 2011
Intel Core i3	2010–present
Intel Core i5	2009–present
Intel Core i7	2008–present
Intel Core i7 (Extreme Edition)	2011–present
Intel Core i9	2018–present
Intel Core i9 (Extreme Edition)	Q3 2017–present

Microprocessor Classification



- Differences in

- 1) Instruction set: set of instructions that microprocessor can execute
- 2) Bandwidth: no. of bits processed in each instruction
- 3) Clock speed: instructions per second

Instruction Set Architecture (ISA)

1. CISC - Complex Instruction Set Computer

- 1970s - computer memory expensive
- each instruction complex; code is short but complex to reduce memory usage

```
mov ax, 10
mov bx, 5
mul bx, ax
```

single instruction

2. RISC - Reduced Instruction Set Computer

- simple instructions for single, simple tasks
- separate instructions for load and store
- numbers of lines of code increased

```
mov ax, 0
mov bx, 10
mov cx, 5
Begin add ax, bx
loop Begin
```

multiple instructions

RISC vs CISC

RISC

- Simple instructions, few in number.
- Fixed length instructions.
- Multiple register sets.
- Three operands per instruction.
- Parameter passing through register windows.
- Single-cycle instructions.
- Hardwired control.
- Highly pipelined.
- Complexity in compiler.
- Only **LOAD/STORE** instructions access memory.
- Few addressing modes.

CISC

- Many complex instructions.
- Variable length instructions.
- Single register set.
- One or two register operands per instruction.
- Parameter passing through memory.
- Multiple cycle instructions.
- Microprogrammed control.
- Less pipelined.
- Many instructions can access memory.
- Many addressing modes.

ARM (ACORN RISC MACHINE → ADVANCED RISC MACHINE)

- 32-bit embedded RISC
- High performance
- Low power
- Low system cost
- Licensed and fabricated

- Raspberry Pi - ARM Cortex-A72 (microprocessor)



- Arduino Due - ARM Cortex-M3 CPU (microcontroller)



• ARM Architecture versions

Version 1 (ARM1): 26 bit addressing, no coprocessor.

Version 2 (ARM2): Includes a 32 bit result multiply coprocessor

Version 2as (ARM3 & 250):

Version 3 (ARM6, 7, 8): 32 bit addressing

Version 4 (Strong ARM, ARM9): Half word load/store instructions were provided.

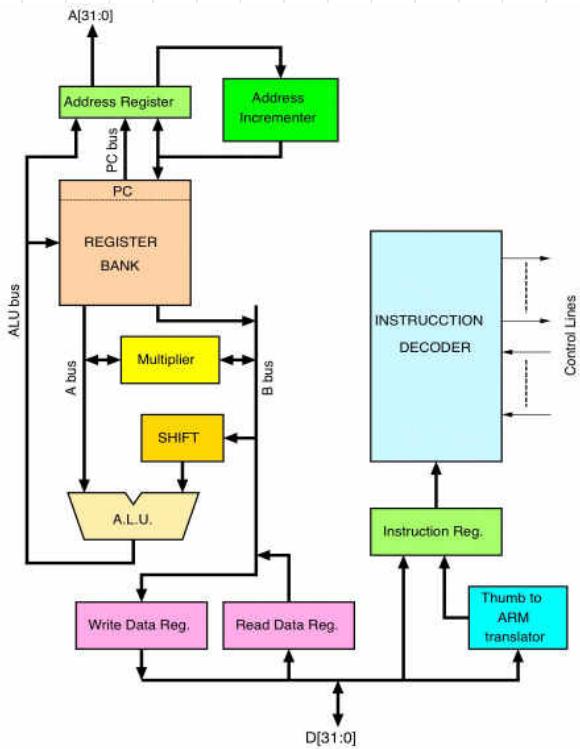
Version 4T: Thumbing: 16 bit instructions can be compressed in a 32 bit processor, thus enabling more instructions to be packed in the same memory, thereby increasing the code density.

Version 5T and 5TE (ARM10): 5TE: thumb extension- built for powerful computations.

CORTEX-M, CORTEX-R, CORTEX-A (32 Bit and 64 bit), NEOVERSE

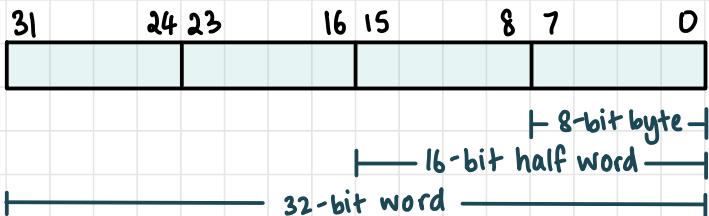
ARM7TDMI Processor

- Low-end ARM core for mobile phones
- TDMI
 - T: thumbing, 16-bit instruction set
 - D: on-chip debug support
 - M: enhanced multiplier
 - I: embedded ICE hardware
- Von Neumann Architecture (unlike Harvard; common memory for data and instructions)
- 3-stage pipeline (fetch, decode, execute)
- 32-bit data bus
- 32-bit address bus
- 37 32-bit registers
- 32-bit ARM instruction set
- 16-bit THUMB instruction set
- 32×8 multiplier
- Barrel shifter



Data Sizes and Instruction Sets

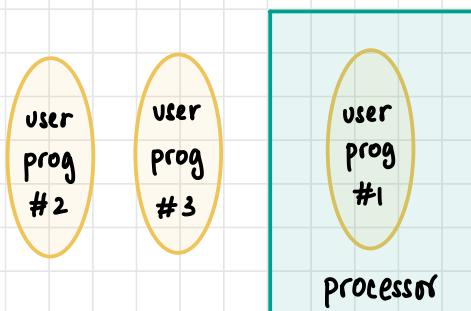
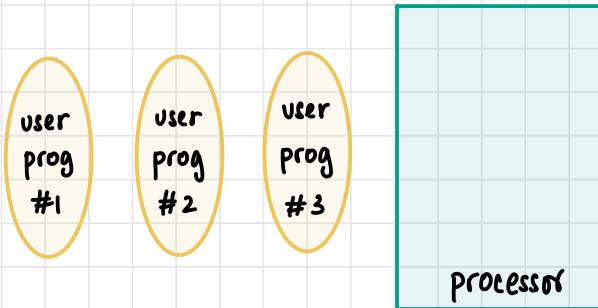
- 32-bit architecture ; byte-addressable
- **Byte:** 8-bit
Halfword: 16-bit ; aligned on 2-byte boundary
Word: 32-bit; aligned on 4-bit boundary
- Two instruction sets:
 - 32-bit ARM instruction set
 - 16-bit Thumb instruction set



Processor Modes

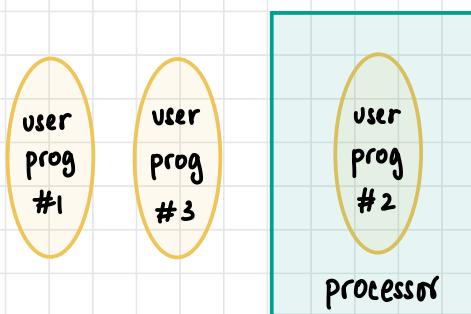
- **User:** unprivileged mode (low priority) ; most tasks
- **FIQ:** high priority (fast) interrupt raised **fast interrupt request**
- **IRQ:** low priority (normal) interrupt raised **interrupt request**
- **Supervisor:** entered on reset and when software interrupt instruction executed
- **Abort:** memory access violations
- **Undef:** handle undefined instructions
- **System:** privileged mode using same registers as user modes

- interrupt service routines for each interrupt
- FIQ/IRQ pins on microprocessor
- eg: processor executing programs



User prog #1
executing

user progs #2
and #3 waiting



User prog #2
executing

context switching
between user
prog #1 and user
prog #2

user
prog
#1

user
prog
#2

user
prog
#3

processor

User prog #3
executing

context switching
between user
prog #2 and user
prog #3

normal
priority
prog
#1

user
prog
#1

user
prog
#2

user
prog
#3

processor

priority prog
to be executed

high
priority
prog
#1

user
prog
#1

user
prog
#2

user
prog
#3

normal
priority
prog
#1

processor

priority progs
executed with
context switching

high priority prog
to be executed

normal
priority
prog
#1

user
prog
#1

user
prog
#2

user
prog
#3

high
priority
prog
#1

processor

Register Bank

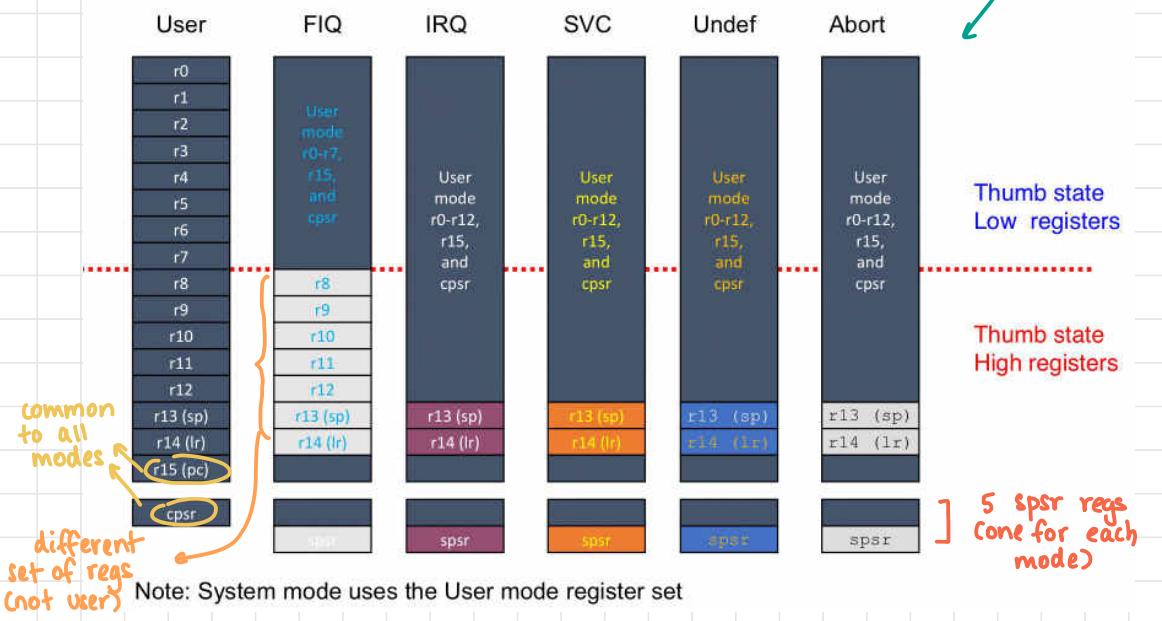
- ARM has 37 registers of 32 bit length
 - 1 dedicated PC
 - 1 dedicated Current Program Status Register (CPSR)
 - 5 dedicated Saved Program Status Registers (SPSR)
 - 30 general purpose registers
- Processor mode governs which bank of several banks of registers is accessible
 - set of r0 - r12 registers - available to user (not all 30)
 - r13 (stack pointer, SP) and r14 (link register, LR)
 - r15 (Program counter, PC)
 - current program status register (CPSR)

context switch:
cpcr
contents moved to
dedicated SPSR

Register Organisation Summary

- same colour (bg): same regs ; common for modes

totally
37 regs



Visible and Banked Out Registers

Current Visible Registers

User Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

Banked out Registers

FIQ IRQ SVC Undef Abort

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

spsr spsr spsr spsr

Current Visible Registers

SVC Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

Banked out Registers

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

spsr spsr spsr spsr

Current Visible Registers

FIQ Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

Banked out Registers

User IRQ SVC Undef Abort

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

spsr spsr spsr spsr

Current Visible Registers

SVC Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

Banked out Registers

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

spsr spsr spsr spsr

Current Visible Registers

Undef Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

Banked out Registers

User FIQ IRQ SVC Abort
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

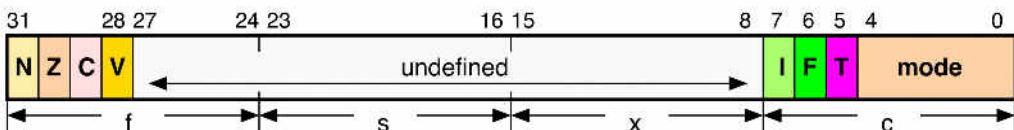
Banked out Registers

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr

spsr spsr spsr spsr

Status Register

- decision making using status register



- f: flag bits
 - s: status bits } future enhancements; - x: extension bits } no use now
- N: negative set to 1 if result of prev. instruction is negative
Z: zero set to 1 if result of prev. instruction is zero
C: carry set to 1 if result of arith or shifter produces a carry-out
V: overflow 1 if prev. instruction produces an overflow into sign bit

- Mode bits

10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

- Interrupt Disable bits

I = 1: disables IRQ,
F = 1: disables FIQ,

- T Bit (arch with Thumb mode only)

T = 0: processor in ARM state
T = 1: processor in Thumb state

- Thumb is 16-bit instruction set
 - optimised for code density from C code
 - subset of ARM functionality
 - only low registers r0 to r7 used
- I & F bits (illegal)

I F

- 1 1 FIQ is served, IRQ and FIQ is disabled
 1 0 IRQ is served, IRQ is disabled & FIQ is enabled
0 1 FIQ is served, IRQ is enabled & FIQ is disabled (Not Allowed)
 0 0 USER program is served, IRQ and FIQ both are enabled

ARM Program Structure

Address	Instruction & Data
Address of Instruction 1	ARM Instruction_1
Address of Instruction 2	ARM Instruction_3
Address of Instruction 3	ARM Instruction_3
.....
.....
Address of Instruction n	ARM Instruction_n
.text	
Address of Data1	Declaration of variable 1
Address of Data2	Declaration of variable 2
.....
.....
Address of Data n	Declaration of variable n
.data	

Note: Blue color depict the code written by the programmer
 Red color depict the address assigned during execution

p1.s

only .TEXT

.TEXT

```
MOV R0, #10  
MOV R1, #20  
ADD R2, R0, R1
```

ARM Simulator

step info

https://webhome.cs.uvic.ca/~nigelh/ARMSim-V2.1/index.html

ARMSim - The ARM Simulator Dept. of Computer Science

File View Cache Debug Watch Help

RegistersView x General Purpose Floating Point

CodeView x p1.s .TEXT
00001000:53A0000A MOV R0, #10
00001004:53A00104 MOV R1, #20
00001008:30000001 ADD R2, R0, R1

MemoryView x 00000000 Word Size: 8bit 16bit 32bit

CPSR Register
Negative (N) : 0
Zero (Z) : 0
Carry (C) : 0
Overflow (V) : 0
IRQ Disable: 1
FIQ Disable: 1
Thumb (T) : 0
CPU Mode : System

0x0000000E

Data & Text

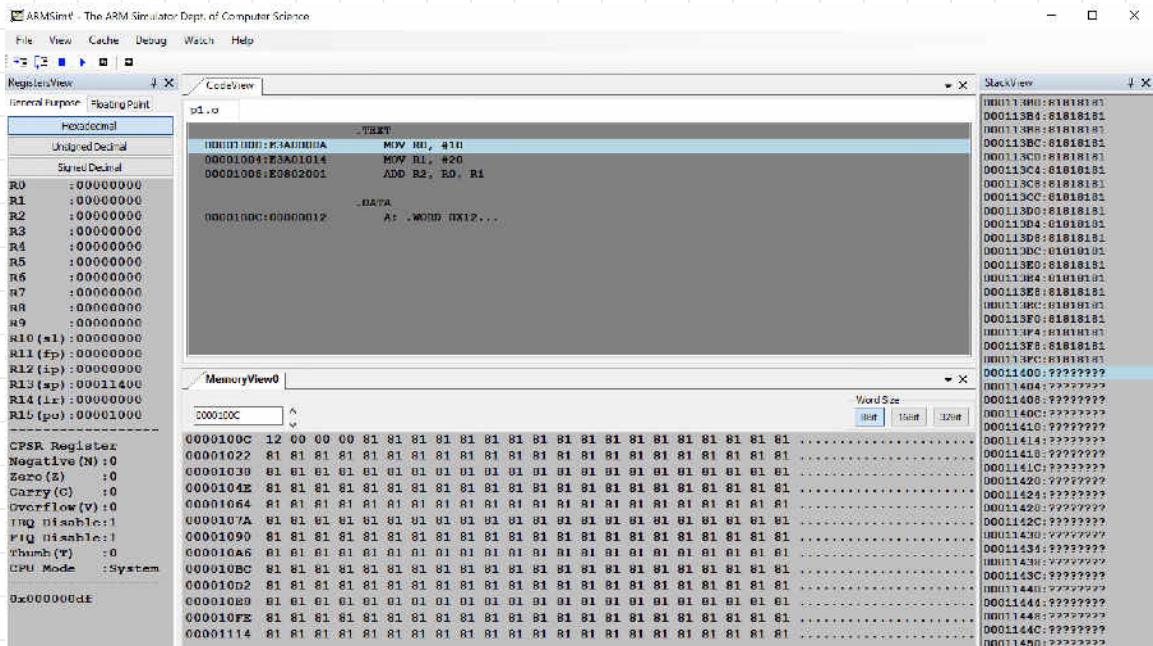
p1.s

.TEXT

```
MOV R0, #10  
MOV R1, #20  
ADD R2, R0, R1
```

.DATA

```
A: .WORD 0X12
```



↑ little endian

Status stored in CPSR - add S to end of instruction

p1.s

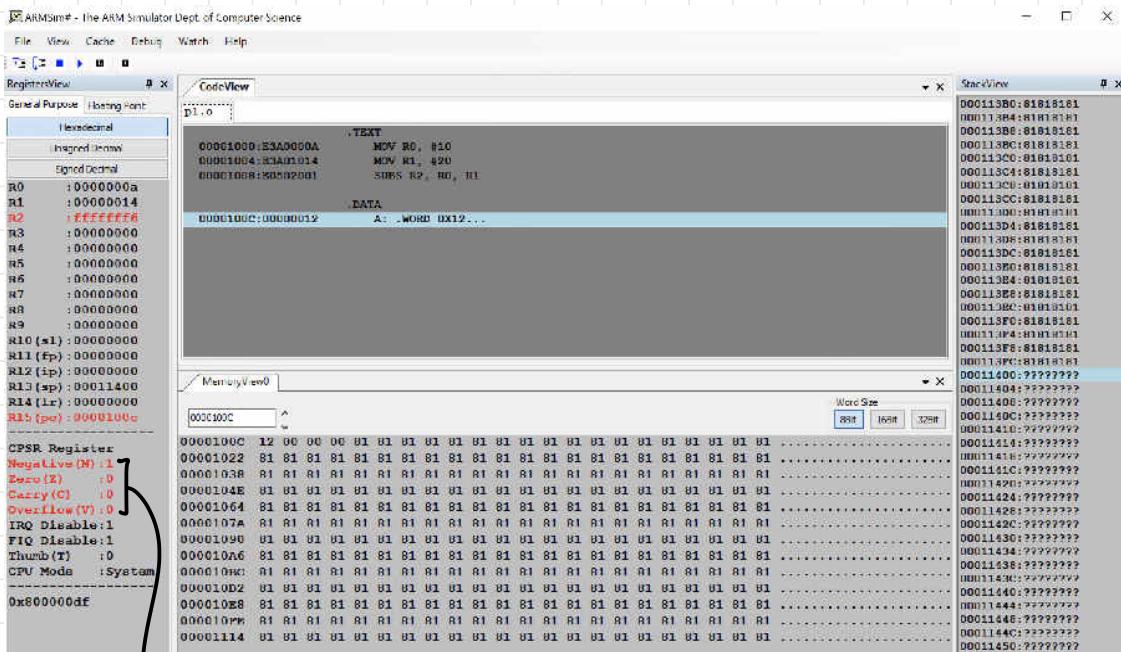
.DATA

A: .WORD 0X12

.TEXT

MOV R0, #10
MOV R1, #20
SUBS R2, R0, R1

stores in CPSR



flags reflected in CPSR

Load-Store Architecture

Address	Instruction	Meaning
00001000:EF9F0014	LDR R0, =a	Load the Address of a to R0
00001004:EF9F1014	LDR R1,=b	Load the Address of b to R1
00001008:EF9F3014	LDR R3,=c	Load the Address of c to R3
0000100C:E5D14000	LDR R4, [r1]	Load the value (100) to R4
00001010:E5D05000	LDR R5, [r0]	Load the value (200) to R5
00001014:E0846005	Add R6, R4, R5	Add R4 & R5
00001018:E00360B0	STR R6, [r3]	Store the result(300) in the address specified in R3
	.data	
00001028:	a: .word 100	Variable a of data type word
00001029:	b: word 200	Variable b of data type word
0000102A:	c: word 0	Variable c of data type word

} datatype
can be
word, byte,
halfword,
asciz

Features of ARM Instructions

- 32 bit instructions
- load-store architecture
- most instructions are 3-address instructions
- conditional execution of each instruction
- can load/store multiple reg at once
- can combine ALU and shift operation
- no memory-memory operations

Instruction Format

MNEMONIC{condition}{S} {Rd}, Operand1, Operand2

MNEMONIC - Short name of the instruction. *Eg: ADD, SUB....*

{condition} - Condition that is needed to be met in order for the instruction to be executed *Eg: EQ, MI, GT, LT, LE, AL, NE*

{S} - An optional suffix. If S is specified, the condition flags are updated on the result of the operation. *Eg: To set N, O, C, V of CPSR*

{Rd} - Register (destination) for storing the result of the instruction

Operand1 - First operand. Either a register or an immediate value

Operand2 - Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

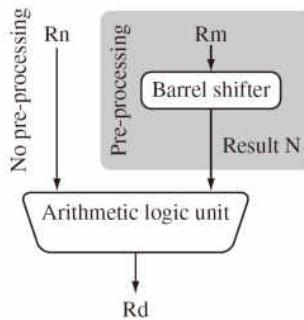
syntax error
if 2 imm
or imm
before reg

ARM Conditional Codes

Code	Meaning (for cmp or subs)	Flags Tested
eq	Equal.	Z==1
ne	Not equal.	Z==0
cs or hs	Unsigned higher or same (or carry set).	C==1
cc or lo	Unsigned lower (or carry clear).	C==0
mi	Negative. The mnemonic stands for "minus".	N==1
pl	Positive or zero. The mnemonic stands for "plus".	N==0
vs	Signed overflow. The mnemonic stands for "V set".	V==1
vc	No signed overflow. The mnemonic stands for "V clear".	V==0
hi	Unsigned higher.	(C==1) && (Z==0)
ls	Unsigned lower or same.	(C==0) (Z==1)
ge	Signed greater than or equal.	N==V
lt	Signed less than.	N!=V
gt	Signed greater than.	(Z==0) && (N==V)
le	Signed less than or equal.	(Z==1) (N!=V)
al (or omitted)	Always executed.	None tested.

Data Processing Instructions

- largest family of ARM instructions
- contains: arithmetic, comparisons, logical, data movement
- load/store architecture
- specific instructions
- first operand is always a register - Rn, second operand is sent to ALU via barrel shifter



Data Movement

- MOV - operand2
- MVN - operand 2
 - move negation
- no operand1
- <Operation> {<cond>} {S} Rd, Operand2

.TEXT

```
MOV R0, #10  
MOV R1, #10  
CMP R0, R1
```

not stored;
only CPSR

.DATA

```
A: .WORD 0X12
```

- MOV r0, r1
- MOVS r2, #10
- MVNEQ r1, #0

.text

```
;ADD 2 numbers loaded from register  
MOV r0, #0x80  
MOV r1, #20  
ADD r2, r0, r1
```

```
;Sub 2 numbers loaded from register  
SUB r4, r1, r0  
SUB r3, r0, r1  
.end
```

Arithmetic Operations

- ADD: operand1 + operand2
- SUB: operand1 - operand2
- RSB: operand2 - operand1 ← reverse subtraction

<Operation> {<cond>} {S} Rd, Rn, Operand2

R0	:0000000a
R1	:0000000a
R2	:00000000
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00000000
R9	:00000000
R10(sl)	:00000000
R11(fp)	:00000000
R12(ip)	:00000000
R13(sp)	:00011400
R14(lr)	:00000000
R15(pc)	:0000100c
<hr/>	
CPSR Register	
Negative(N)	:0
Zero(Z)	:1
Carry(C)	:1
Overflow(V)	:0
IRQ Disable	:1
FIQ Disable	:1
Thumb(T)	:0
CPU Mode	:System
<hr/>	
0x600000df	

- With carry

- ADC:** Operand1 + Operand 2 + CPSR.c
- SBC:** Operand 1 – Operand2 – NOT(CPSR.c)
or
Operand1 - Operand2 + carry -1
- RSC:** Operand2-Operand 1 - NOT(CPSR.c)
or
Operand2 – Operand1 + carry -1

- for SBC and RSC

Operand2 and Carry in are inverted and fed to adder or Invert Operand2 first and subtract carry later

Wrong Application of SBC

```
.text
MOV r0, #4
MOV r1, #2
SUB r3,r0,r1
SBC r2, r0, r1
.end
```

should use SUBC

```
.text
MOV r0, #4
MOV r1, #2
SUBS r3,r0,r1
SBC r2, r0, r1
.end
```

RegistersView	
General Purpose Floating Point	
Hexadecimal	
Unsigned Decimal	
Signed Decimal	
R0	:4
R1	:2
R2	:4
R3	:2
R4	:8
R5	:4
R6	:0
R7	:0

Multword Arithmetic

- operations for more than 32 bit operands, ADC, SBC, RSC are used

64-bit word

32 bit MSB word	32 bit LSB word
-----------------	-----------------

- add 2 64-bit words

ADDS R4, R0, R2

R1	R0
----	----

+

ADC R5, R1, R3

R3	R2
----	----

result

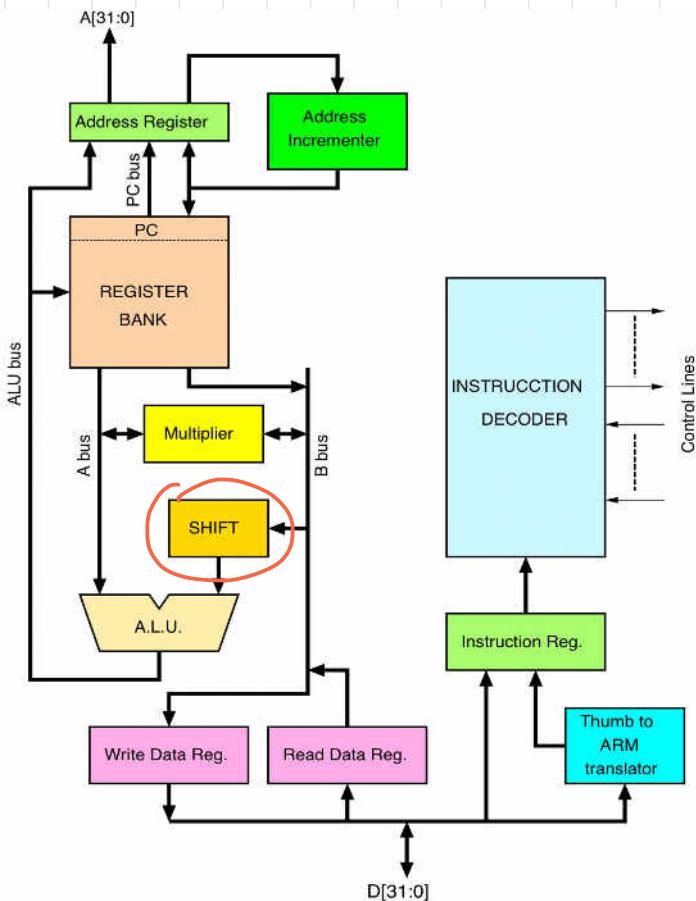
R5	R4
----	----

- subtract one 64-bit word from another

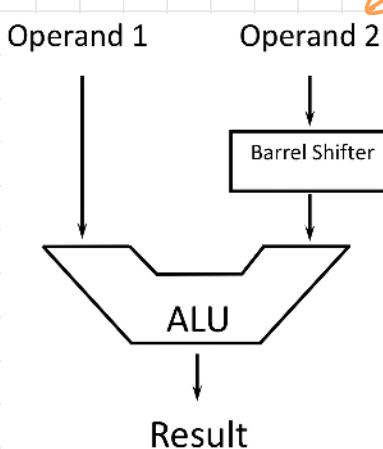
SUBS R3, R6, R9
SBCS R4, R7, R10
SBC R5, R8, R11

Barrel Shifter

- 3-stage pipeline
- fetch, decode, execute
- no actual shift instruction; instead only barrel shifter with shifts as part of other instructions



Barrel Shift Operations



register or imm
5-bit 8-bit

Possible Shift Operations:

LSL: Left Shift

LSR: Right Shift

ASR: Arithmetic Right Shift

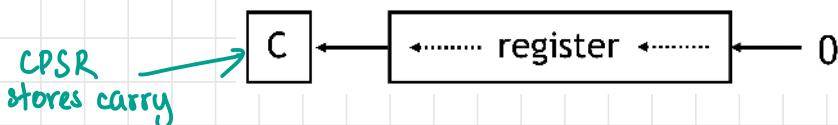
ROR: Rotate Right

RRX: Rotate Right Extended

- default: LSL#0

— Logical Shift Left —

- LSL



Syntax

.TEXT

```
MOV R2, #0X00000030
MOV R0, R2, LSL #2
```

imm - left shift by 2 bits

.TEXT

```
MOV R2, #0X00000030
MOV R3, #0X00000002
MOV R0, R2, LSL R3
```

register

before

00000000 00000000 00000000 00110000

after

00000000 00000000 00000000 11000000

- multiplied by 2^n

ARMSim# - The ARM Simulator Dept. of Computer Science

File View Cache Debug Watch Help

RegistersView CodeView StackView

General Purpose Floating Point

Hexadecimal

RegistersView

R0 : 00000000 R2 unchanged

R1 : 00000000 R3 : 00000030

R2 : 00000000 R4 : 00000000

R5 : 00000000 R6 : 00000000

R7 : 00000000 R8 : 00000000

R9 : 00000000 R10 (s1) : 00000000

R11 (fp) : 00000000 R12 (ip) : 00000000

R13 (sp) : 000011400 R14 (lr) : 00000000

R15 (pc) : 000001008

CPSR Register Negative(N) : 0 Zero(Z) : 0 Carry(C) : 0 Overflow(V) : 0 IRQ Disable: 1 FIQ Disable: 1 Thumb(T) : 0 CPU Mode : System

MemoryView1

Word Size: 8Bit 16Bit 32Bit

00001000 30 20 A0 E3 02 01 A0 E1 0 ä.. á

00001008 81 81 81 81 81 81 81 81

00001010 81 81 81 81 81 81 81 81

00001018 81 81 81 81 81 81 81 81

00001020 81 81 81 81 81 81 81 81

00001028 81 81 81 81 81 81 81 81

00001030 81 81 81 81 81 81 81 81

00001038 81 81 81 81 81 81 81 81

00001040 81 81 81 81 81 81 81 81

00001048 81 81 81 81 81 81 81 81

00001050 81 81 81 81 81 81 81 81

00001058 81 81 81 81 81 81 81 81

00001060 81 81 81 81 81 81 81 81

.TEXT

00001000:E3A02030 MOV R2, #0X00000030

00001004:E1A00102 MOV R0, R2, LSL #2

000113B0:81818181
000113B4:81818181
000113B8:81818181
000113BC:81818181
000113C0:81818181
000113C4:81818181
000113C8:81818181
000113CC:81818181
000113D0:81818181
000113D4:81818181
000113D8:81818181
000113DC:81818181
000113E0:81818181
000113E4:81818181
000113E8:81818181
000113EC:81818181
000113F0:81818181
000113F4:81818181
000113F8:81818181
000113FC:81818181
00011400:????????
00011404:????????
00011408:????????
0001140C:????????
00011410:????????
00011414:????????
00011418:????????
0001141C:????????
00011420:????????
00011424:????????
00011428:????????
0001142C:????????
00011430:????????
00011434:????????
00011438:????????
0001143C:????????
00011440:????????
00011444:????????
00011448:????????
0001144C:????????
00011450:????????

— Logical Shift Right —

- LSR



Syntax

.TEXT

```
MOV R2, #0xFFFFFFFFD0  
MOV R0, R2, LSR #2
```

before

11111111 11111111 11111111 11010000

after

00111111 11111111 11111111 11110100

- divided by 2^n

Output

.text

```
MOV R0, #3  
MOV R1, #256  
ADD R3, R0, R1, LSR #5
```

.end

$$3 + 256 \div 32$$

$$R3 = 11$$

Before R1: 00000000 00000000 000000 1000 0001

After Right Shift R1: 00000000 00000000 000000 0000 1000 (8)

Arithmetic Shift Right

- ASR



Syntax

```
.text  
MOV R2, #0xffffffffd0  
MOV R1, R2, ASR #2
```

before

11111111 11111111 11111111 11010000

after

11111111 11111111 11111111 11110100

Rotate Right

- ROR

LSB to
MSB



Syntax

```
MOV R2, #0xffffffffd5  
MOV R0, R2, ROR #2
```

before

11111111 11111111 11111111 11010101

after

(01111111 11111111 11111111 11110101)

Rotate Right Extended

• RRX

- rotates number right by 1 bit, moves LSB to carry bit and moves carry bit to MSB



Syntax

MOV R0, R2, RRX

Multiplication by 2^n (1,2,4,8,16,32..)

MOV Ra, Rb, LSL #n

Multiplication by 2^{n+1} (3,5,9,17..)

ADD Ra,Ra,Ra,LSL #n

Multiplication by 2^{n-1} (3,7,15..)

RSB Ra,Ra,Ra,LSL #n

Multiplication by 6

ADD Ra,Ra,Ra,LSL #1 ; multiply by 3

MOV Ra,Ra,LSL#1 ; and then by 2

Multiply by 10 and add in extra number

ADD Ra,Ra,Ra,LSL#2 ; multiply by 5

ADD Ra,Rc,Ra,LSL#1 ; multiply by 2 and add in next digit

Comparison Instructions

Syntax

<Operation> {<cond>} Rn, Operand2

{S} not needed as CPSR
reflects changes anyway

compare ↗ CMP R1, R2 @ set cc on R1-R2

compare negation ↗ CMN R1, R2 @ set cc on R1+R2

test ↗ TST R1, R2 @ set cc on R1 and R2

test equality ↗ TEQ R1, R2 @ set cc on R1 xor R2

CMP

Example 1

.TEXT

```
MOV R0, #25
MOV R1, #256
CMP R0, R1
```

→ R0-R1 is negative

.END

R0	:00000019
R1	:00000100
R2	:00000000
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00000000
R9	:00000000
R10(sl)	:00000000
R11(fp)	:00000000
R12(ip)	:00000000
R13(sp)	:00011400
R14(lr)	:00000000
R15(pc)	:0000100c

CPSR Register

Negative(N)	:1
Zero(Z)	:0
Carry(C)	:0
Overflow(V)	:0
IRQ Disable	:1
FIQ Disable	:1
Thumb(T)	:0
CPU Mode	:System

negative

Example 2

.TEXT

```
MOV R0, #256
MOV R1, #25
CMP R0, R1
```

.END

$$\begin{array}{rcl} 256 & = & 00000100 \\ 25 & = & 00000019 \\ -25 & = & \text{ffffffe7} \end{array}$$

$$\begin{array}{rcl} 256 & = & \begin{smallmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \end{smallmatrix} \\ -25 & = & \underline{\begin{smallmatrix} & f & f & f & f & f & e & 1 \end{smallmatrix}} \\ & = & \begin{smallmatrix} 1 \\ 0 & 0 & 0 & 0 & 0 & e & 7 \end{smallmatrix} \end{array} \quad \rightarrow 231$$

(1) *carry*

R0	:	00000100
R1	:	00000100
R2	:	00000000
R3	:	00000000
R4	:	00000000
R5	:	00000000
R6	:	00000000
R7	:	00000000
R8	:	00000000
R9	:	00000000
R10(sl)	:	00000000
R11(fp)	:	00000000
R12(ip)	:	00000000
R13(sp)	:	00011400
R14(lr)	:	00000000
R15(pc)	:	0000100c

CPSR Register	
Negative(N)	:0
Zero(Z)	:0
Carry(C)	:1
Overflow(V)	:0
IRQ Disable	:1
FIQ Disable	:1
Thumb(T)	:0
CPU Mode	:System

2's comp
subtraction

Example 3

.TEXT

```
MOV R0, #256
MOV R1, #256
CMP R0, R1
```

.END

R0	:	00000100
R1	:	00000100
R2	:	00000000
R3	:	00000000
R4	:	00000000
R5	:	00000000
R6	:	00000000
R7	:	00000000
R8	:	00000000
R9	:	00000000
R10(sl)	:	00000000
R11(fp)	:	00000000
R12(ip)	:	00000000
R13(sp)	:	00011400
R14(lr)	:	00000000
R15(pc)	:	0000100c

CPSR Register	
Negative(N)	:0
Zero(Z)	:1
Carry(C)	:1
Overflow(V)	:0
IRQ Disable	:1
FIQ Disable	:1
Thumb(T)	:0
CPU Mode	:System

TST & TEQ

- test and test equivalence

Example 1

.TEXT

```
MOV R0, #-5
MOV R1, #5
TEQ R0, R1
ADDEQ R3, R0, R1
```

.END

R0	:fffffb
R1	:00000005
R2	:00000000
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00000000
R9	:00000000
R10(sl)	:00000000
R11(fp)	:00000000
R12(ip)	:00000000
R13(sp)	:00011400
R14(lr)	:00000000
R15(pc)	:0000100c

CPSR Register
Negative(N) :1
Zero(Z) :0
Carry(C) :0
Overflow(V) :0
IRQ Disable:1
FIQ Disable:1
Thumb(T) :0
CPU Mode :System

Example 2

.TEXT

```
MOV R0, #-5
MOV R1, #-5
TEQ R0, R1
ADDEQ R3, R0, R1
```

.END

R0	:fffffb
R1	:fffffb
R2	:00000000
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00000000
R9	:00000000
R10(sl)	:00000000
R11(fp)	:00000000
R12(ip)	:00000000
R13(sp)	:00011400
R14(lr)	:00000000
R15(pc)	:0000100c

CPSR Register
Negative(N) :0
Zero(Z) :1
Carry(C) :0
Overflow(V) :0
IRQ Disable:1
FIQ Disable:1
Thumb(T) :0
CPU Mode :System

Example 3

check if even

.TEXT

MOV R0, #12

TST R0, #1

.END

R0	: 0000000c
R1	: 00000000
R2	: 00000000
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10(sl)	: 00000000
R11(fp)	: 00000000
R12(ip)	: 00000000
R13(sp)	: 00011400
R14(lr)	: 00000000
R15(pc)	: 00001008

CPSR Register	
Negative(N)	: 0
Zero(Z)	: 1
Carry(C)	: 0
Overflow(V)	: 0
IRQ Disable	: 1
FIQ Disable	: 1
Thumb(T)	: 0
CPU Mode	: System

if zero = 1,
no. is even

Logical Operations

- AND
- EOR
- ORR
- BIC ← bit clear

Syntax

<Operation> {<cond>} {S} Rd, Rn, Operand2

AND R0, R1, R2 @ R0 = R1 and R2

ORR R0, R1, R2 @ R0 = R1 or R2

EOR R0, R1, R2 @ R0 = R1 xor R2

BIC R0, R1, R2 @ R0 = R1 and (\sim R2)

BIC

Example 1

```
MOV R1, #0x11111111  
MOV R2, #0x01100101  
BIC R0, R1, R2
```

0x 10011010

mask for which
bits should get cleared

.TEXT

```
MOV R0, #5  
MOV R1, #6  
AND R2, R0, R1      @ Logical AND  
ORR R3, R0, R1      @ Logical OR  
EOR R4, R0, R1      @ Logical XOR  
MVN R5, R0          @ Complement
```

.END

complements

R0	:00000005
R1	:00000006
R2	:00000004
R3	:00000007
R4	:00000003
R5	: ffffffffa
R6	:00000000
R7	:00000000
R8	:00000000
R9	:00000000
R10 (s1)	:00000000
R11 (fp)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:00001018

CPSR Register	
Negative (N)	:0
Zero (Z)	:0
Carry (C)	:0
Overflow (V)	:0
IRQ Disable	:1
FIQ Disable	:1
Thumb (T)	:0
CPU Mode	:System

Flow Control Instructions

- B{<cond>} Label
- BL{<cond>} Label
- BX{<cond>} Rm
- BLX{<cond>} Rm

Reference:

<https://developer.arm.com/documentation/dui0040/d/introducing-arm-and-thumb/basic-arm-language-interworking/the-branch-exchange-instruction>

B	Branch	Program Counter = Label
BL	Branch & Link	Step1: PC will be copied to R14 the Link Register (LR) before branch is taken. Step2: Program Counter = Label
BX	Branch Exchange	Used for changing ARM to Thumb mode or from Thumb mode to ARM mode.
BLX	Branch Exchange with link	

Branch Instruction

Unconditional Branch

B label

.

.

.

label:



Conditional Branch

MOV R0, #0

loop:

ADD R0, R0, #1

CMP R0, #10

BNE loop

CPSR

branch if not equal

backward
branching

Branch and Link

- BL instruction saves return address to R14 (lr) from PC

PC stores addr of CMP, moves to lr

BL sub	@ call sub
CMP R1, #5	@ return to here
MOVEQ R1, #0	
...	

would have been bypassed
in regular B instruction

sub: ...	@ sub entry point
...	

MOV PC, LR @ return

move LR contents back to PC (return)

Example 1

.TEXT

```
MOV R0, #5
MOV R1, #5
CMP R0, R1 ← satisfied
BEQ label
MOV R2, #6
```

label:

```
MOV R3, #20
```

R0	:	00000005
R1	:	00000005
R2	:	00000000
R3	:	00000000
R4	:	00000000
R5	:	00000000
R6	:	00000000
R7	:	00000000
R8	:	00000000
R9	:	00000000
R10 (s1)	:	00000000
R11 (fp)	:	00000000
R12 (ip)	:	00000000
R13 (sp)	:	00011400
R14 (lr)	:	00000000
R15 (pc)	:	00001014

CPSR Register
Negative (N) : 0
Zero (Z) : 1
Carry (C) : 1
Overflow (V) : 0
IRQ Disable : 1
FIQ Disable : 1
Thumb (T) : 0
CPU Mode : System

Example 2

.TEXT

```
MOV R0, #5  
MOV R1, #5  
CMP R0, R1  
BLEQ label  
MOV R2, #6
```

label:

```
MOV R3, #20  
MOV PC, LR
```

The RegistersView pane shows the state of general-purpose registers (R0-R15) and CPSR. The PC register is highlighted in red. The CodeView pane shows the assembly code for the .TEXT section of p3.o.

Register	Value
R0	: 00000005
R1	: 00000005
R2	: 00000000
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10 (s1)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00011400
R14 (lr)	: 00001010
R15 (pc)	: 00001014

CPSR Register:
Negative(N): 0
Zero(Z): 1
Carry(C): 1
Overflow(V): 0

```
.TEXT  
00001000:E3A00005 MOV R0, #5  
00001004:E3A01005 MOV R1, #5  
00001008:E1500001 CMP R0, R1  
0000100C:0B000000 BLEQ label  
00001010:E3A02006 MOV R2, #6  
label:  
00001014:E3A03014 MOV R3, #20  
00001016:E1A0F005 MOV PC, LR...
```

PC contents in LR

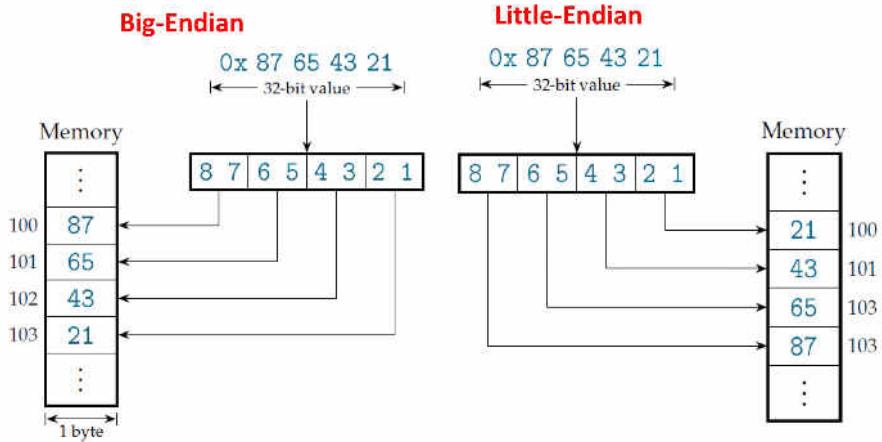
Data Transfer Instructions

- Memory to register
- Register to memory
- Load-store architecture
- Memory: array of 0 to $2^{32}-1$
- word, half-word, byte

A memory array diagram showing 16 bytes of memory starting at address 0x00000000. The bytes are arranged in a grid:

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
0xFFFFFFFFFD	00
0xFFFFFFFFFE	00
0xFFFFFFFFFF	00

Big Endian and Little Endian



Load / Store

- Move data between memory and registers
- Single register load / store: LDR, STR
- Multiple register load / store or Block Transfer: LDM, STM

— Single Register Load/Store —

Syntax

<LDR/STR>{<cond>} {B} *Rd* *Addressing* ↗ *destination reg* ↗ *memory address*

LDR: mem to reg
STR: reg to mem

Instructions

LDR	Load word into register
STR	Save byte or word from register
LDRB	Load byte into register
STRB	Save byte from Register

Half-words

LDR{<cond>}**SB/H/SH** Rd, Addressing
STR{<cond>}**H** Rd, Addressing

signed byte *half word* *signed HW*

LDRH	Load half word into register
STRH	Save half word from a register
LDRSB	Load signed byte into register
LDRSH	Load signed halfword into a Register

No STRSB/STRSH since STRB/STRH stores both signed/unsigned ones

Example 1

copy A = B

Memory

addr	data
0x010	10
0x014	
0x018	30
0x01C	40
0x020	50
0x024	60

Let B



Let A



LDR R0, =A;
LDR R5, [R0];

@ R0 = 0x1C - address of A
@ Copy data from R0 into R5 (40)

LDR R3, =B;
STR R5, [R3];

@ R3 = 0x14 - address of B
@ copy contents of R5 (40) into R3's data

- The `[]` operator is similar to C/C++'s `*` operator (pointer dereference) and the `=` operator is similar to `&` (address of)
- `int *R0 = &A;` // similar to this
`int R5 = *R0;` // but not high level

.TEXT

```
LDR R0, =A  
LDR R5, [R0]
```

Executed in ARMsim

```
LDR R3, =B  
STR R5, [R3]
```

.DATA

```
A: .WORD 10  
B: .WORD
```

The screenshot shows the ARMsim interface with three main windows:

- RegistersView**: Shows general purpose registers R0-R15 and floating-point register FPC. R0 contains 00001018, R15 (pc) is highlighted in red at 00001010.
- CodeView**: Displays assembly code in the p4.o file. The code is:

```
.TEXT  
00001000:E59F0008 LDR R0, =A  
00001004:E5905000 LDR R5, [R0]  
  
.DATA  
00001008:E59F3004 LDR R3, =B  
0000100C:E5835000 STR R5, [R3]  
  
A: .WORD 10  
B: .WORD
```
- MemoryView**: Shows memory starting at address 0000101c. The word at 0000101c is highlighted in red as 0000000A, while others are 81818181. Word size is set to 32Bit.

Example 2

$$B = A + 9$$

.TEXT

```
LDR R0, =A
LDR R1, [R0]
ADD R2, R1, #9
```

operator only supported
by LDR/STR instructions,
not ADD

```
LDR R3, =B
STR R2, [R3]
```

.DATA

```
A: .WORD 10
B: .WORD
```

The screenshot shows a debugger interface with three main panes:

- RegistersView**: Shows general purpose registers (R0-R15) and floating-point registers (fp, ip, sp, lr, pc). Most registers are at their initial value (0). R15 (pc) is highlighted in red and shows the value 00001014.
- CodeView**: Displays assembly code for the file p1.o. The code consists of three sections: .TEXT, .DATA, and .BSS. The .TEXT section contains the instructions for loading A and B into registers, performing the addition, and storing the result. The .DATA section defines variables A and B. The .BSS section is empty.
- MemoryView**: Shows memory starting at address 00001020. The first word (00001020) is highlighted in blue and shows the value 00000013. Subsequent words show the values 81818181, indicating the binary representation of the constant 10.

Example 2

$$C = A + B \quad (\text{half word})$$

.TEXT

```
LDR R0, =A
LDR R1, =B
LDR R3, =C
LDRH R4, [R0]
LDRH R5, [R1]
ADD R6, R4, R5
STRH R6, [R3]
```

.DATA

A: .HWORD 10]	half word
B: .HWORD 20		
C: .HWORD		

.END

The screenshot shows the following windows:

- RegistersView**: Shows general purpose registers R0-R15 with their values in hexadecimal, decimal, and signed decimal formats.
- CodeView**: Shows the assembly code for the program p1.o. The memory locations 00001000-E59F0014 through 00001018-E1C360B0 are highlighted in blue. The labels .TEXT, .DATA, and .END are also visible.
- MemoryView1**: Shows the memory dump in little endian format. The address 00001028 is selected. The data bytes are shown as pairs of hex digits: 00001028 000A 0014 001E 8181 8181 00001032 8181 8181 8181 8181 8181. An arrow points from the text "little endian" to the memory dump window.

Annotations:

- An orange bracket groups the three lines of the .DATA section (A, B, C) under the label "half word".
- The text "little endian" is written in red above the MemoryView window.
- The text "(see e-bit)" is written in red in the bottom right corner.

— string —

- Datatype: ASCII " " (byte)
- string must always be loaded into R0
- SWI : software interrupt — calls interrupt service routine (stored in a table)
- SWI 0x02 — print onto stdout
0x11 — normal exit from prog

.TEXT

```
LDR R0, =MYSTR  
SWI 0X02  
SWI 0X11
```

.DATA

```
MYSTR: .ASCIZ "HELLO WORLD"
```

The screenshot shows a debugger interface with four main windows:

- RegistersView**: Shows general purpose registers (R0-R15) in hexadecimal, all set to 0.
- CodeView**: Displays assembly code:

```
.TEXT  
00001000:E59F0004 LDR R0, =MYSTR  
00001004:EF000002 SWI 0X02  
00001008:EF000011 SWI 0X11  
.DATA  
00001010:4C4C4548 MYSTR: .ASCIZ "HELLO WORLD".  
:4F57204F  
:00444C52
```
- OutputView**: Shows the execution log:

```
Loading assembly language  
Execution starting ...  
HELLO WORLD  
Execution ending, Instruct  
Instructions per second: 63
```
- MemoryView1**: Shows memory dump at address 00001010:

Word Size	8Bit	16Bit	32Bit
00001010	48 45 4C 4C 4F 20 57 4F 52 4C HELLO WORL		
0000101A	44 00 81 81 81 81 81 81 81 81 D.....		
00001024	81 81 81 81 81 81 81 81 81 81		

ADDRESSING OR INDEXING

- Arrays

1. Pre-Indexing Without Writeback

LDR Rd, [Rn, OFFSET]

mem loc
first increment
pc, then store
incremented

value of Rn unchanged

2. Pre-Indexing With Writeback

LDR Rd, [Rn, OFFSET] !

value of Rn changed

3. Post-Indexing

LDR Rd, [Rn], OFFSET

first stored in Rd,
then Rn incremented

implicitly
incremented

offset

- Immediate:

pre indexing w/o writeback

LDR, R0, [R1, #4] @mem [R1+4]

- Register:

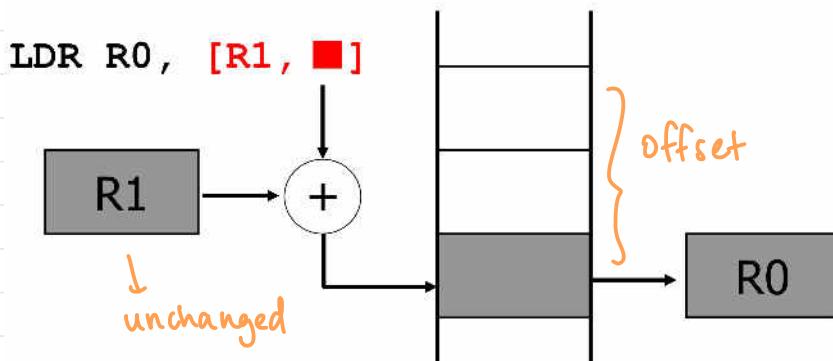
LDR, R0, [R1, R2] @mem [R1+R2]

- Scaled Register type has to be reg for scaled

LDR R0, [R1, R2, LSL #2] @mem [R1+4*R2]

PRE-INDEXING WITHOUT WRITEBACK

LDR R0, [R1, #4]



RegistersView

General Purpose Floating Point

	Hexadecimal
R0	: 00001010
R1	: 00000114
R2	: 00000000
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10 (s1)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00011400
R14 (lr)	: 00000000
R15 (pc)	: 00001008

CPSR Register

Negative (N) : 0

Zero (Z) : 0

Carry (C) : 0

CodeView

```
.TEXT
00001000:E59F0004 LDR R0, =A
00001004:E5901004 LDR R1, [R0, #4]
00001008:EF000011 SWI 0X11

.DATA
00001010:0000000A A: .WORD 10, 20, 30, 40, 50
:00000014
:0000001E
:00000028
:00000032

.END
```

little endian

Word Size: 8Bit 16Bit 32Bit

00001010	0A 00 00 00	14 00 00 00	1E 00
0000101A	00 00 28 00	00 00 32 00	00 00 ..(....2...
00001024	81 81 81 81	81 81 81 81

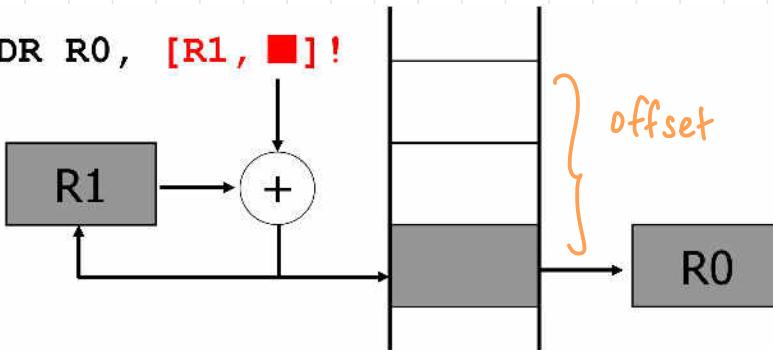
#4: 4 bytes
(#2: useless value)

AUTO INDEXING

PRE-INDEXING WITH WRITEBACK

LDR R0, [R1, #4]! (faster)

LDR R0, [R1, █]!



RegistersView CodeView MemoryView1

General Purpose Floating Point

Hexadecimal Unsigned Decimal Signed Decimal

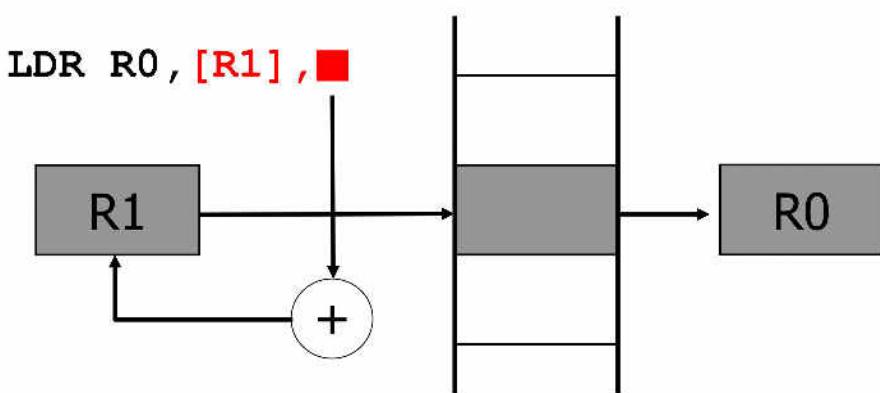
R0 : 00001014	00001000:E59F0004	.TEXT
R1 : 00000014	00001004:E5B01004	LDR R0, =A
R2 : 00000000	00001008:EF000011	LDR R1, [R0, #4]!
R3 : 00000000		SWI 0X11
R4 : 00000000		
R5 : 00000000		
R6 : 00000000		
R7 : 00000000		
R8 : 00000000		
R9 : 00000000		
R10 (sl) : 00000000		
R11 (fp) : 00000000		
R12 (ip) : 00000000		
R13 (sp) : 00011400		
R14 (lr) : 00000000		
R15 (pc) : 00001008		

CPSR Register Word Size
Negative (N) : 0 8Bit 16Bit 32Bit
Zero (Z) : 0

00001010	0000000A	00000014	0000001E	00000028
00001020	00000032	81818181	81818181	81818181

POST INDEXING

LDR R0, [R1], #4



RegistersView X

General Purpose Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : 00001014	
R1 : 0000000a	
R2 : 00000000	
R3 : 00000000	
R4 : 00000000	
R5 : 00000000	
R6 : 00000000	
R7 : 00000000	
R8 : 00000000	
R9 : 00000000	
R10 (s1) : 00000000	
R11 (fp) : 00000000	
R12 (ip) : 00000000	
R13 (sp) : 00011400	
R14 (lr) : 00000000	
R15 (pc) : 00001008	

CodeView X

p1.o

00001000:E59F0004	.TEXT
00001004:E4901004	LDR R0, =A
00001008:EF000011	LDR R1, [R0], #4
	SWI 0X11

.DATA

00001010:0000000A A: .WORD 10, 20, 30, 40, 50

:00000014

:0000001E

:00000028

:00000032

.END

MemoryView1 X

Word Size

8Bit 16Bit 32Bit

00001010	^
00001010 0000000A 00000014 0000001E 00000028	
00001020 00000032 81818181 81818181 81818181	

BLOCK TRANSFER

- Multiple register load/store
- Transfer 10, 20, 30 to registers
- Single register load-store

.text

LDR R4, =A

LDR R1, [R4], #4

LDR R2, [R4], #4

LDR R3, [R4], #4

} post indexing

LDR R4, =B

STR R1, [R4], #4

STR R2, [R4], #4

STR R3, [R4], #4

} post indexing

.data

A: .word 10, 20, 30, 40, 50

B: .word

- Quite tedious
- LDM and SDM instructions
 - load multiple registers

Syntax

<LDM/STM> {cond} <Addressing Mode>Rn {!}, Registers

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1010	40
0x1020	50
0x1024	60

direction of transfer is different

Addressing Mode

Addressing Mode	Meaning
IA	Increase after
IB	Increase before
DA	Decrease after
DB	Decrease before

moving through memory

Registers

LDM <IA/IB/DA/DB> Rn, {R1,R2,R3}

or

LDMIA <IA/IB/DA/DB> Rn, {R1-R3}

Warning if order wrong, not error

inclusive

LDMIA

.TEXT

LDR R8, =A

LDR R9, =B

LDMIA R8, {R0, R1, R2}

STMIA R9, {R0-R2}

.DATA

A: .WORD 10, 20, 30, 40, 50

B: .WORD

Load at once

R0	:0000000a
R1	:00000014
R2	:0000001e
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00001018
R9	:0000102c
R10 (s1)	:00000000
R11 (f1)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:0000100c

Store at Once

RegistersView X

General Purpose Floating Point

	Hexadecimal
R0	: 0000000a
R1	: 00000014
R2	: 0000001e
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00001018
R9	: 0000102c
R10 (sl)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00011400
R14 (lr)	: 00000000
R15 (pc)	: 00001010

CPSR Register

Negative(N) : 0
Zero(Z) : 0
Carry(C) : 0
Overflow(V) : 0
IRQ Disable: 1
FIQ Disable: 1
Thumb(T) : 0
CPU Mode : System

CodeView X

p1.o

```

.TEXT
00001000:E59F8008    LDR R8, =A
00001004:E59F9008    LDR R9, =B

00001008:E8980007    LDMIA R8, {R0, R1, R2}   @ W
0000100C:E8890007    STMIA R9, {R0-R2}

.DATA
00001018:0000000A    A: .WORD 10, 20, 30, 40, 50
                      :00000014
                      :0000001E
                      :00000028
                      :00000032

B: .WORD...

```

MemoryView0 X

Word Size: 32Bit

00001018	^	▼		
00001018	0000000A	00000014	0000001E	00000028
00001028	00000032	0000000A	00000014	0000001E
00001038	81818181	81818181	81818181	81818181
00001048	81818181	81818181	81818181	81818181
00001058	81818181	81818181	81818181	81818181
00001068	81818181	81818181	81818181	81818181
00001078	81818181	81818181	81818181	81818181
00001088	81818181	81818181	81818181	81818181

With Writeback

.TEXT

LDR R8, =A

LDR R9, =B

LDMIA R8!, {R0, R1, R2}

STMIA R9!, {R0-R2}

.DATA

A: .WORD 10, 20, 30, 40, 50

B: .WORD

R8: 1024 (wb)

R0	: 0000000a
R1	: 00000014
R2	: 0000001e
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00001024
R9	: 0000102c
R10 (sl)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00011400
R14 (lr)	: 00000000
R15 (pc)	: 0000100c

RegistersView CodeView

General Purpose		Floating Point	
Hexadecimal			
Unsigned Decimal			
Signed Decimal			
R0	:0000000a		
R1	:00000014		
R2	:0000001e		
R3	:00000000		
R4	:00000000		
R5	:00000000		
R6	:00000000		
R7	:00000000		
R8	:00001024		
R9	:00001038		
R10 (s1)	:00000000		
R11 (fp)	:00000000		
R12 (ip)	:00000000		
R13 (sp)	:00011400		
R14 (lr)	:00000000		
R15 (pc)	:00001010		

CPSR Register		Word Size		
Negative (N)	:0	8Bit	16Bit	32Bit
Zero (Z)	:0			
Carry (C)	:0			
Overflow (V)	:0			
IRQ Disable	:1			
FIQ Disable	:1			
Thumb (T)	:0			
CPU Mode	:System			

MemoryView0

00001018	0000000A	00000014	0000001E	00000028
00001028	00000032	0000000A	00000014	0000001E
00001038	81818181	81818181	81818181	81818181
00001048	81818181	81818181	81818181	81818181
00001058	81818181	81818181	81818181	81818181
00001068	81818181	81818181	81818181	81818181
00001078	81818181	81818181	81818181	81818181
00001088	81818181	81818181	81818181	81818181

LDMIB

- increment before store and load

.TEXT

LDR R8, =A

LDR R9, =B

LDMIB R8!, {R0, R1, R2}

STMIB R9!, {R0-R2}

.DATA

A: .WORD 10, 20, 30, 40, 50

B: .WORD

R0	:00000014
R1	:0000001e
R2	:00000028
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00001024
R9	:0000102c
R10 (s1)	:00000000
R11 (fp)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:0000100c

RegistersView

General Purpose Floating Point

Hexadecimal
Unsigned Decimal
Signed Decimal
R0 : 00000014
R1 : 0000001e
R2 : 00000028
R3 : 00000000
R4 : 00000000
R5 : 00000000
R6 : 00000000
R7 : 00000000
R8 : 00001024
R9 : 00001038
R10(sl) : 00000000
R11(fp) : 00000000
R12(ip) : 00000000
R13(sp) : 00011400
R14(lr) : 00000000
R15(pc) : 00001010

CPSR Register

Negative(N) : 0
Zero(Z) : 0
Carry(C) : 0
Overflow(V) : 0
IRQ Disable:1
FIQ Disable:1
Thumb(T) : 0
CPU Mode : System

CodeView

p1.o

```
.TEXT
00001000:E59F8008 LDR R8, =A
00001004:E59F9008 LDR R9, =B

00001008:E9B80007 LDMIB R8!, {R0, R1, R2}
0000100C:E9A90007 STMIB R9!, {R0-R2}

.DATA
00001018:0000000A A: .WORD 10, 20, 30, 40, 50
:00000014
:0000001E
:00000028
:00000032

B: .WORD ...

MemoryView0
```

Word Size: 8Bit 16Bit 32Bit

00001018	0000000A	00000014	0000001E	00000028
00001028	00000032	81818181	00000014	0000001E
00001038	00000028	81818181	81818181	81818181
00001048	81818181	81818181	81818181	81818181
00001058	81818181	81818181	81818181	81818181
00001068	81818181	81818181	81818181	81818181
00001078	81818181	81818181	81818181	81818181
00001088	81818181	81818181	81818181	81818181

LDMDA

- decrement after
- word: #4 (4 bytes)

TEXT

LDR R8, =A
LDR R9, =B

LDR R6, [R8], #16
LDR R7, [R9], #12

LDMDA R6!, {R0, R1, R2}
STMADA R7!, {R0-R2}

50
higher addr:
higher reg

DATA

A: .WORD 10, 20, 30, 40, 50
B: .WORD

R0	: 0000001e	30
R1	: 00000028	40
R2	: 00000032	50
R3	: 00000000	
R4	: 00000000	
R5	: 00000000	
R6	: 00000000	
R7	: 81818181	
R8	: 00001024	
R9	: 00001040	
R10(sl)	: 00000000	
R11(fp)	: 00000000	
R12(ip)	: 00000000	
R13(sp)	: 00011400	
R14(lr)	: 00000000	
R15(pc)	: 00001014	

RegistersView X

General Purpose Floating Point

Hexadecimal
Unsigned Decimal
Signed Decimal
R0 : 0000001e
R1 : 00000028
R2 : 00000032
R3 : 00000000
R4 : 00000000
R5 : 00000000
R6 : 00000000
R7 : 81818181
R8 : 00001024
R9 : 00001034
R10 (sl) : 00000000
R11 (fp) : 00000000
R12 (ip) : 00000000
R13 (sp) : 00011400
R14 (lr) : 00000000
R15 (pc) : 00001018

CPSR Register

Negative(N) : 0
Zero(Z) : 0
Carry(C) : 0
Overflow(V) : 0
IRQ Disable: 1
FIQ Disable: 1
Thumb(T) : 0
CPU Mode : System

CodeView X

p1.o

```
.TEXT
00001000:E59F8010 LDR R8, =A
00001004:E59F9010 LDR R9, =B

00001008:E4987010 LDR R7, [R8], #16
0000100C:E499700C LDR R7, [R9], #12

00001010:E8380007 LDMDB R8!, {R0, R1, R2}
00001014:E8290007 STMDB R9!, {R0-R2}

.DATA
00001020:000000A A: .WORD 10, 20, 30, 40, 50
:00000014
:0000001E
:00000028
:00000032

B: .WORD...
```

MemoryView0 X

Word Size: 32Bit

00001018	^			
00001018	00001020	00001034	0000000A	00000014
00001028	0000001E	00000028	00000032	81818181
00001038	0000001E	00000028	00000032	81818181
00001048	81818181	81818181	81818181	81818181
00001058	81818181	81818181	81818181	81818181
00001068	81818181	81818181	81818181	81818181
00001078	81818181	81818181	81818181	81818181
00001088	81818181	81818181	81818181	81818181

— LDMDB —

- decrement before
- word: #4 (4 bytes)

.TEXT

LDR R8, =A
LDR R9, =B

LDR R7, [R8], #16
LDR R7, [R9], #12

LDMDB R8!, {R0, R1, R2}
STMDB R9!, {R0-R2}

.DATA

A: .WORD 10, 20, 30, 40, 50
B: .WORD

R0	: 00000014	20
R1	: 0000001e	30
R2	: 00000028	40
R3	: 00000000	
R4	: 00000000	
R5	: 00000000	
R6	: 00000000	
R7	: 81818181	
R8	: 00001024	
R9	: 00001040	
R10 (sl)	: 00000000	
R11 (fp)	: 00000000	
R12 (ip)	: 00000000	
R13 (sp)	: 00011400	
R14 (lr)	: 00000000	
R15 (pc)	: 00001014	

RegistersView

General Purpose Floating Point

	Hexadecimal
R0	:00000014
R1	:000001e
R2	:00000028
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:81818181
R8	:00001024
R9	:00001034
R10 (sl)	:00000000
R11 (fp)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:00001018

CPSR Register

Negative(N) :0
Zero(Z) :0
Carry(C) :0
Overflow(V) :0
IRQ Disable:1
FIQ Disable:1
Thumb(T) :0
CPU Mode :System

CodeView

p1.o

```
.TEXT
00001000:E59F8010    LDR R8, =A
00001004:E59F9010    LDR R9, =B

00001008:E94987010   LDR R7, [R8], #16
0000100C:E499700C   LDR R7, [R9], #12

00001010:E9380007   LDMDB R8!, {R0, R1, R2}
00001014:E9290007   STMDB R9!, {R0-R2}

.DATA
00001020:0000000A    A: .WORD 10, 20, 30, 40, 50
:00000014
:0000001E
:00000028
:00000032

B: .WORD...
```

MemoryView0

Word Size: 8Bit 16Bit 32Bit

00001018	^			
00001018	00001020	00001034	0000000A	00000014
00001028	0000001E	00000028	00000032	00000014
00001038	0000001E	00000028	81818181	81818181
00001048	81818181	81818181	81818181	81818181
00001058	81818181	81818181	81818181	81818181
00001068	81818181	81818181	81818181	81818181
00001078	81818181	81818181	81818181	81818181
00001088	81818181	81818181	81818181	81818181

Without Moving R8, R9

RegistersView

General Purpose Floating Point

	Hexadecimal
R0	:e9290007
R1	:00001018
R2	:00001024
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:0000100c
R9	:00001020
R10 (sl)	:00000000
R11 (fp)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:00001010

CPSR Register

Negative(N) :0
Zero(Z) :0
Carry(C) :0
Overflow(V) :0
IRQ Disable:1
FIQ Disable:1
Thumb(T) :0
CPU Mode :System

CodeView

p1.o

```
.TEXT
00001000:E59F8008    LDR R8, =A
00001004:E59F9008    LDR R9, =B

00001008:E9380007   LDMDB R8!, {R0, R1, R2}
0000100C:E9290007   STMDB R9!, {R0-R2}

.DATA
00001018:0000000A    A: .WORD 10, 20, 30, 40, 50
:00000014
:0000001E
:00000028
:00000032

B: .WORD...
```

MemoryView0

Word Size: 8Bit 16Bit 32Bit

00001008	^			
00001008	E9380007	E9290007	00001018	0000102c
00001018	0000000A	00000014	E9290007	00001018
00001028	0000102c	81818181	81818181	81818181
00001038	81818181	81818181	81818181	81818181
00001048	81818181	81818181	81818181	81818181
00001058	81818181	81818181	81818181	81818181
00001068	81818181	81818181	81818181	81818181
00001078	81818181	81818181	81818181	81818181

overwriting

ADDRESSING MODES

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	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

BLOCK TRANSFER - STACK

- FILO fashion
- R13: stack pointer (top of stack)
- mainly used in procedural calls
- stack can grow upward or downward (based on mode)

Addr	data
0x1010	
0x1014	
0x1018	10
0x101C	20
0x1010	30
0x1020	40
0x1024	50
0x1028	60
0x102C	
0x1030	
0x1034	

Syntax

<LDM/STM> <Addressing Mode>R13{!}, Registers

STM: push onto stack reg to mem (stack)
LDM: pop from stack mem (stack) to reg

mode	POP	=LDM	PUSH	=STM
Full ascending (FA)	LDMFA	LDMDA	STMFA	STMIB
Full descending (FD)	LDMFD	LDMIA	STMF	STMDB
Empty ascending (EA)	LDMEA	LDMDB	STMEA	STMIA
Empty descending (ED)	LDMED	LDMIB	STMED	STMDA

- use R13 - stack pointer

↑ either works
(functionally equivalent)

— (LDM | STM) EA —

Example 1

.TEXT

```

MOV R0, #1
MOV R1, #2
MOV R2, #3
STMEA R13, {R0, R1, R2}
SWI 0X11

```

push, inc

The screenshot shows a debugger interface with four main windows:

- RegistersView**: Shows general purpose registers (R0-R15) in Hexadecimal, Unsigned Decimal, and Signed Decimal formats. R13 is highlighted.
- CodeView**: Shows assembly code for the .TEXT section. The instruction `STMEA R13, {R0, R1, R2}` is highlighted.
- StackView**: Shows memory starting at address 000013B0, where all bytes are 81 (hex).
- MemoryView1**: Shows memory starting at address 00001028, with word size set to 32Bit. The value 00001400 is highlighted.

Example 2

.TEXT

```

MOV R0, #1
MOV R1, #2
MOV R2, #3
STMEA R13!, {R0, R1, R2}
SWI 0X11

```

The screenshot shows a debugger interface with three main windows:

- RegistersView**: Shows general purpose and floating point registers in Hexadecimal, Unsigned Decimal, and Signed Decimal formats. The R0, R1, and R2 registers are highlighted in red.
- CodeView**: Displays assembly code with memory addresses. The assembly code is identical to Example 2.
- StackView**: Shows the stack memory starting at address 000013BC, containing mostly zeros with some scattered ones.

Example 3

.TEXT

```

MOV R0, #1
MOV R1, #2
MOV R2, #3
STMEA R13!, {R0, R1, R2}
LDMEA R13!, {R3, R4, R5}
SWI 0X11

```

The screenshot shows a debugger interface with three main panes:

- RegistersView**: Shows general purpose and floating point registers in decimal, hexadecimal, and signed decimal formats. The R15 register (pc) is highlighted in red and contains the value 00001014.
- CodeView**: Displays assembly code for the file b1.o. The code includes:
 - MOV R0, #1
 - MOV R1, #2
 - MOV R2, #3
 - STMFA R13!, {R0, R1, R2}
 - LDMFA R13!, {R3, R4, R5}
 - SWI 0X11...
- StackView**: Shows the stack memory starting at address 000013B0, containing multiple entries of the byte sequence 81818181.

— (LDM / STM) FA —

.TEXT

```

MOV R0, #1
MOV R1, #2
MOV R2, #3
STMFA R13!, {R0, R1, R2}
LDMFA R13!, {R3, R4, R5}
SWI 0X11

```

inc, push (1B)
pop, dec (DA)

The screenshot shows a debugger interface with three main panes:

- RegistersView**: Shows general purpose and floating point registers in decimal, hexadecimal, and signed decimal formats. The R15 register (pc) is highlighted in red and contains the value 00001014.
- CodeView**: Displays assembly code for the file b1.o. The code includes:
 - MOV R0, #1
 - MOV R1, #2
 - MOV R2, #3
 - STMFA R13!, {R0, R1, R2}
 - LDMFA R13!, {R3, R4, R5}
 - SWI 0X11...
- StackView**: Shows the stack memory starting at address 000013B0, containing multiple entries of the byte sequence 81818181.

(LDM/STM) ED

Example 1

.TEXT

```
MOV R0, #1
MOV R1, #2
MOV R2, #3
STMED R13!, {R0, R1, R2}
LDMED R13!, {R3, R4, R5}
SWI 0X11
```

The screenshot shows a debugger interface with three main panes:

- RegistersView**: Shows general purpose registers (R0-R15) in Hexadecimal, Unsigned Decimal, and Signed Decimal formats. Values are: R0: 00000001, R1: 00000002, R2: 00000003, R3: 00000001, R4: 00000002, R5: 00000003, R6: 00000000, R7: 00000000, R8: 00000000, R9: 00000000, R10 (sl): 00000000, R11 (fp): 00000000, R12 (ip): 00000000, R13 (sp): 00001400, R14 (lr): 00000000, R15 (pc): 00001014.
- CodeView**: Displays assembly code for the file p1.o. The code is:

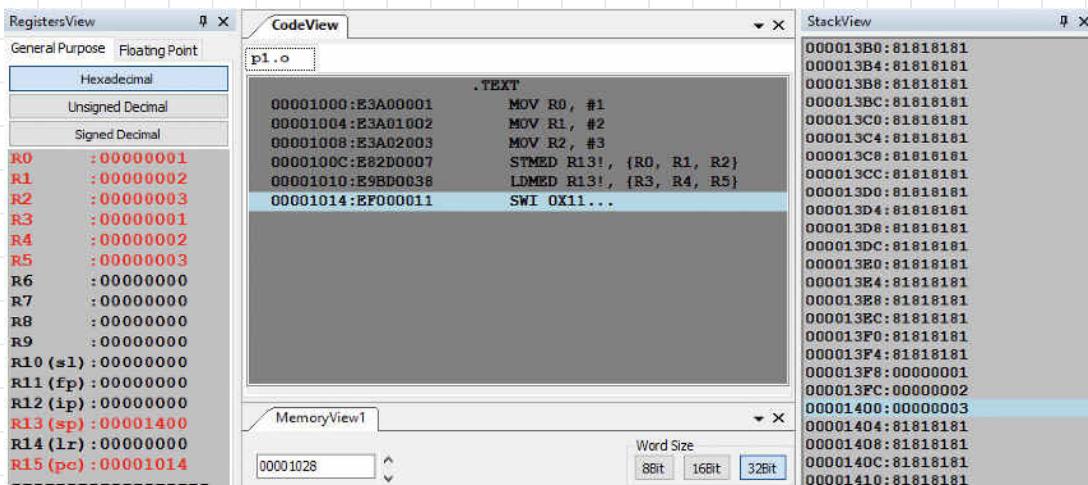
```
.TEXT
00001000:E3A00001    MOV R0, #1
00001004:E3A01002    MOV R1, #2
00001008:E3A02003    MOV R2, #3
0000100C:E82D0007    STMED R13!, {R0, R1, R2}
00001010:E9BD0038    LDMED R13!, {R3, R4, R5}
00001014:EF000011    SWI 0X11...
```
- StackView**: Shows memory starting at address 000013B0. The stack grows downwards, with values: 000013B0:81818181, 000013B4:81818181, 000013B8:81818181, 000013BC:81818181, 000013C0:81818181, 000013C4:81818181, 000013C8:81818181, 000013CC:81818181, 000013D0:81818181, 000013D4:81818181, 000013D8:81818181, 000013DC:81818181, 000013E0:81818181, 000013E4:81818181, 000013E8:81818181, 000013EC:81818181, 000013F0:81818181, 000013F4:81818181, 000013F8:00000001, 000013FC:00000002, 00001400:00000003, 00001404:81818181, 00001408:81818181, 0000140C:81818181, 00001410:81818181.

(LDM/STM)FD

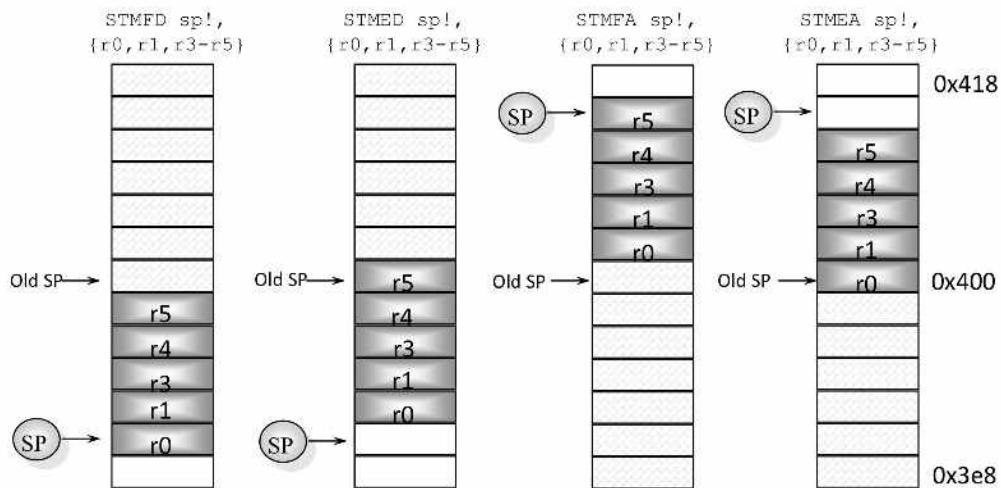
- mostly used

.TEXT

```
MOV R0, #1
MOV R1, #2
MOV R2, #3
STMFD R13!, {R0, R1, R2}
LDMFD R13!, {R3, R4, R5}
SWI 0X11
```



Stack Load/Store Instructions



PROCEDURE call

- Perform link operation before branching (BL)
- Current PC value (R15) stored in R14 before the branch is taken (implicitly by BL instruction)
- Returning back: MOV R15, R14 or MOV PC, LR or BX LR
- Like a function

Main Procedure	
0x0000	Instruction 1
0x0004	Instruction 2
0x0008	Instruction 3
0x000C	BL Procedure
0x0010	Instruction 4
0x0014	Instruction 5

	Instruction Last

Called Procedure	
Procedure:	Instruction 1
	Instruction 2
	Instruction 3
	MOV PC LR
	or
	BX LR

Example 1

.TEXT

```
MOV R0, #10
ADD R1, R0, #20
BL FUNCTION
MOV R2, #50
SWI 0X11
```

unparameterised

FUNCTION:

```
MOV R5, #8
SUB R2, R5, #3
MOV PC, LR
```

General Purpose	Floating Point
Hexadecimal	
Unsigned Decimal	
Signed Decimal	
R0 : 0000000a	
R1 : 0000000e	
R2 : 00000000	
R3 : 00000000	
R4 : 00000000	
R5 : 00000000	
R6 : 00000000	
R7 : 00000000	
R8 : 00000000	
R9 : 00000000	
R10 (sl) : 00000000	
R11 (fp) : 00000000	
R12 (ip) : 00000000	
R13 (sp) : 00001400	
R14 (lr) : 00000000	
R15 (pc) : 00001008	

The screenshot shows a debugger interface with two main panes. The left pane displays assembly code for a function named 'p1.o' with the following instructions:

```
.TEXT
00001000:E3A0000A    MOV R0, #10
00001004:E2B01014    ADD R1, R0, #20
00001008:EB000001    BL FUNCTION
0000100C:E3A02032    MOV R2, #50
00001010:EF000011    SWI 0X11

FUNCTION:
00001014:E3A05008    MOV R5, #8
00001018:E2452003    SUB R2, R5, #3
0000101C:E1A0F00E    MOV PC, LR
```

The right pane shows a memory view with the address 00001028 and word size set to 32Bit. The memory content at 00001028 is shown as 00001008.

RegistersView X

General Purpose Floating Point

	Hexadecimal
R0	: 0000000a
R1	: 0000001e
R2	: 00000000
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10 (sl)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00001400
R14 (lr)	: 0000100c
R15 (pc)	: 00001014

CodeView X

p1.o

```
.TEXT
00001000:E3A0000A    MOV R0, #10
00001004:E2801014    ADD R1, R0, #20
00001008:EB000001    BL FUNCTION
0000100C:E3A02032   MOV R2, #50
00001010:EP000011    SWI 0x11

FUNCTION:
00001014:E3A05008    MOV R5, #8
00001018:E2452003    SUB R2, R5, #3
0000101C:E1A0F00E    MOV PC, LR
```

MemoryView X

Word Size: 32Bit

00001028

RegistersView X

General Purpose Floating Point

	Hexadecimal
R0	: 0000000a
R1	: 0000001e
R2	: 00000005
R3	: 00000000
R4	: 00000000
R5	: 00000008
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10 (sl)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00001400
R14 (lr)	: 0000100c
R15 (pc)	: 0000100c

CodeView X

p1.o

```
.TEXT
00001000:E3A0000A    MOV R0, #10
00001004:E2801014    ADD R1, R0, #20
00001008:EB000001    BL FUNCTION
0000100C:E3A02032   MOV R2, #50
00001010:EP000011    SWI 0x11

FUNCTION:
00001014:E3A05008    MOV R5, #8
00001018:E2452003    SUB R2, R5, #3
0000101C:E1A0F00E    MOV PC, LR
```

MemoryView X

Word Size: 32Bit

00001028

Example 2

parameterised: push parameters onto stack

.TEXT

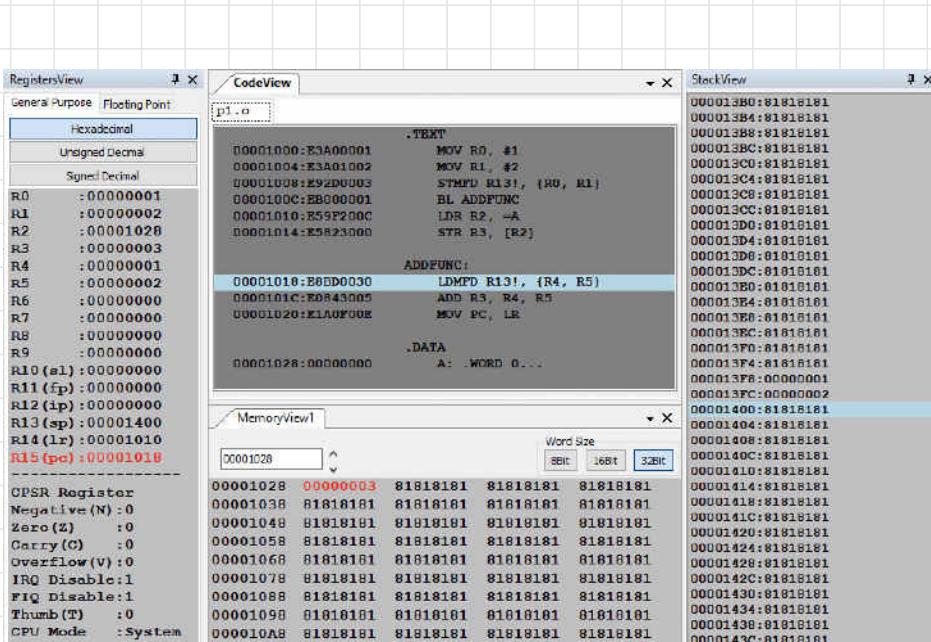
```
MOV R0, #1
MOV R1, #2
STMFD R13!, {R0, R1}
BL ADDFUNC
LDR R2, =A
STR R3, [R2]
```

ADDFUNC:

```
LDMFD R13!, {R4, R5}  
ADD R3, R4, R5  
MOV PC, LR
```

.DATA

A: .WORD 0



nested Procedure calls

@MUL(ADD(A, B), C) $(A+B)*C$

.TEXT

```
MOV R0, #1  
MOV R1, #2  
MOV R2, #3
```

```
STMFD R13!, {R0, R1, R2}
```

```
BL MULFUNC
```

```
LDR R6, =A  
STR R7, [R6]  
SWI 0X11
```

MULFUNC:

```
LDMFD R13!, {R3, R4, R5}  
STMFD R13!, {R3, R4, LR}  
BL ADDFUNC  
LDMFD R13!, {LR}  
MUL R7, R8, R5  
MOV PC, LR
```

ADDFUNC:

```
LDMFD R13!, {R6, R7}  
ADD R8, R6, R7  
MOV PC, LR
```

.DATA

```
A: .WORD 0
```

The screenshot shows a debugger interface with three main panes:

- RegistersView**: Shows general purpose registers R0-R15 and CPSR Register.
- CodeView**: Displays assembly code for a function named p1.o. The code includes instructions like @MUL (ADD(A, B), C), .TEXT, MOV R0, #1, MOV R1, #2, MOV R2, #3, STMPD R13!, (R0, R1, R2), BL MULFUNC, LDR R6, =A, STR R7, [R6], SWI 0X11, and MULFUNC:.
- StackView**: Shows the stack memory starting at address 000013B0, containing various memory addresses and their values.

MULTIPLICATION INSTRUCTIONS

MUL	Multiply	32-bit result
MLA	Multiply accumulate	32-bit result
UMULL <i>long data (64)</i>	Unsigned multiply	64-bit result
UMLAL	Unsigned multiply accumulate	64-bit result
SMULL	Signed multiply	64-bit result
SMLAL	Signed multiply accumulate	64-bit result

2 registers

MUL

$MUL\{<\text{cond}>\}\{S\} \text{Rd}, \text{Rf}, \text{Rs}$ @ $\text{Rd} = (\text{Rf} * \text{Rs})_{[31:0]}$

must be different registers

reg only, not immediate

Example 1

```
.TEXT
    MOV R0, #3
    MOV R1, #2
    MUL R2, R0, R1
    SWI 0X11
```

The screenshot shows two windows from a debugger. The left window is 'RegistersView' showing general purpose registers:

Register	Value
R0	: 00000003
R1	: 00000002
R2	: 00000006
R3	: 00000000
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10 (s1)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00001400
R14 (lr)	: 00000000
R15 (pc)	: 0000100c

The right window is 'CodeView' showing assembly code:

```
.TEXT
00001000:E3A00003      MOV R0, #3
00001004:E3A01002      MOV R1, #2
00001008:E0020190      MUL R2, R0, R1
0000100C:EF000011      SWI 0X11
```

Example 2

$$C = C + a[i] * b[i]$$

.TEXT

```
LDR R0, =A
LDR R1, =B
LDR R2, =C
```

```
MOV R3, #3
MOV R4, #0
```

LOOP:

```
LDR R5, [R0], #4
LDR R6, [R1], #4
MUL R7, R5, R6
ADD R4, R4, R7
SUB R3, R3, #1
CMP R3, #0
BNE LOOP
```

```
STR R4, [R2]
```

.DATA

```
A: .WORD 1, 2, 3
B: .WORD 2, 4, 6
C: .WORD
```

The screenshot shows the following details:

- RegistersView:** Shows registers R0 through R15. R0-R3 are initialized to 0, R4 to 1, R5 to 3, R6 to 0, and R15 (pc) to 000011400.
- CodeView:** Shows assembly code for the program. A handwritten note "post indexing" with a checkmark is placed next to the instruction LDR R5, [R0], #4. The code includes:


```
.TEXT
00001000: E59F002C LDR R0, =A
00001004: E59F102C LDR R1, =B
00001008: E59F202C LDR R2, =C
0000100C: B3A00003 MOV R3, #3
00001010: E3A04000 MOV R4, #0
LOOP:
00001014: E49US004 LDR R5, [R0], #4
00001018: E4916004 LDR R6, [R1], #4
0000101C: B0070695 MUL R7, R5, R6
00001020: E0844007 ADD R4, R4, R7
00001024: E2433001 SUB R3, R3, #1
00001028: B3530000 CMP R3, #0
0000102C: 1AFFFFF8 BNE LOOP
00001030: B5B24000 STR R4, [R2]
```
- MemoryView:** Shows memory starting at address 00001040. The first four bytes (00001040-00001043) are 81818181, representing the value 1 in decimal.

MLA

<MLA> Rd, Rf, Rn, Rm @ Rd = Rf*Rn + Rm

Example 1

$$c = c + a[i] * b[i]$$

.TEXT

```
LDR R0, =A  
LDR R1, =B  
LDR R2, =C
```

```
MOV R3, #3  
MOV R4, #0
```

LOOP:

```
LDR R5, [R0], #4  
LDR R6, [R1], #4  
MLA R4, R5, R6, R4  
SUB R3, R3, #1  
CMP R3, #0  
BNE LOOP
```

replaces
MUL &
ADD

```
STR R4, [R2]
```

.DATA

```
A: .WORD 1, 2, 3  
B: .WORD 2, 4, 6  
C: .WORD
```

RegistersView

General Purpose		Floating Point
Hexadecimal		
Unsigned Decimal		
Signed Decimal		
R0 : 00000000		
R1 : 00000000		
R2 : 00001054		
R3 : 00000000		
R4 : 0000001c		
R5 : 00000003		
R6 : 00000006		
R7 : 00000000		
R8 : 00000000		
R9 : 00000000		
R10 (s1) : 00000000		
R11 (fp) : 00000000		
R12 (ip) : 00000000		
R13 (sp) : 00001400		
R14 (lr) : 00000000		
R15 (pc) : 00011400		

CPSR Register

Negative (N) : 0	
Zero (Z) : 1	
Carry (C) : 1	
Overflow (V) : 0	
IRQ Disable : 1	
FIQ Disable : 1	
Thumb (T) : 0	
CPU Mode : System	

CodeView

```

p1.o
    .TEXT
00001000:E59F0028 LDR R0, =A
00001004:E59F1028 LDR R1, =B
00001008:E59F2028 LDR R2, =C

0000100C:E3A03003 MOV R3, #3
00001010:E3A04000 MOV R4, #0

LOOP:
00001014:E4905004 LDR R5, [R0], #4
00001018:E4916004 LDR R6, [R1], #4
0000101C:E0244695 MLA R4, R5, R6, R4
00001020:E2433001 SUB R3, R3, #1
00001024:E3530000 CMP R3, #0
00001028:1AFFFFFF9 BNE LOOP

0000102C:E5824000 STR R4, [R2]

    .DATA

```


MemoryView0

Word Size	8Bit	16Bit	32Bit
00001040			
00001040 00000002 00000003 00000002 00000004			
00001050 00000006 0000001C 81818181 81818181			
00001060 81818181 81818181 81818181 81818181			
00001070 81818181 81818181 81818181 81818181			
00001080 81818181 81818181 81818181 81818181			
00001090 81818181 81818181 81818181 81818181			

SMLAL / SMULL / UMLAL / UMULL

Syntax

0-31 32-63

<SMLAL/SMULL/UMLAL/UMULL>{cond}{S} 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)

PSR Instructions

MRS

move to reg from status register (read)

MRS R0, CPSR
MRS R1, SPSR

Example 1

.TEXT

```
MOVS R0, #0
MRS R1, CPSR
SWI 0X11
```

The screenshot shows a debugger interface with three main windows:

- RegistersView**: Shows general purpose registers R0-R15 and the CPSR register. R0 is set to 0, R1 to 4000000f, and CPSR has various flags set.
- CodeView**: Displays assembly code for the .TEXT section of file p1.o. It includes the instructions MOVS R0, #0, MRS R1, CPSR, and SWI 0X11.
- MemoryView**: Shows memory starting at address 00001040. The first few bytes are 81 81 81 81, corresponding to the value 0x4000000f.

MSR

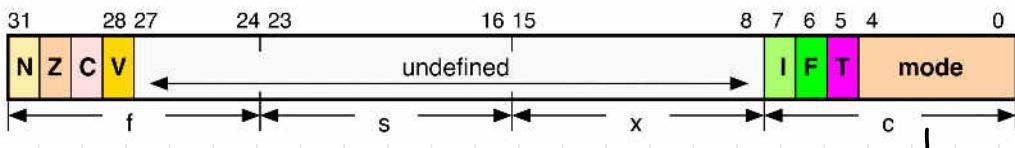
move to status register from reg (write)

Example:

MSR CPSR_field, R0
MSR SPSR_field, R1

_field

_c: Control Field (0:7)
_f: Flag Field(24:31)
_x: Extension (8:15)
_s:Status (16:23)



Example 1

.TEXT

MOVS R0, #0XF0000000
MSR CPSR_f, R0
SWI 0X11

mode:
10000 - user
11111 - system
(page 13)

General Purpose Floating Point

Hexadecimal
Unsigned Decimal
Signed Decimal
R0 : 10000000
R1 : 00000000
R2 : 00000000
R3 : 00000000
R4 : 00000000
R5 : 00000000
R6 : 00000000
R7 : 00000000
R8 : 00000000
R9 : 00000000
R10 (s1) : 00000000
R11 (fp) : 00000000
R12 (ip) : 00000000
R13 (sp) : 00001400
R14 (lr) : 00000000
R15 (pc) : 00001008

p1.o

```
00001000:E3B0020F    .TEXT
00001004:E128E200    MOVS R0, #0XF0000000
00001008:EP000011    MSR CPSR_f, R0
                      SWI 0X11...
```

MemoryView0

Word Size
00001040 01818181 01818181 01818181 01818181
00001050 01818181 01818181 01818181 01818181
00001060 01818181 01818181 01818181 01818181
00001070 01818181 01818181 01818181 01818181
00001080 01818181 01818181 01818181 01818181
00001090 01818181 01818181 01818181 01818181
000010A0 01818181 01818181 01818181 01818181
000010B0 01818181 01818181 01818181 01818181

0XF = 1111 {

only first bits → 0xf00000df

SWP

swap memory and register value

Syntax

SWP <Swap Destination>, <Original>, [<address>]

↓ deprecated in ARMv6 and v7 ↓ same: swap occurs

Example

.TEXT

```
MOV R0, #5  
LDR R1, =A  
SWP R0, R0, [R1]
```

.DATA

```
A: .WORD 6
```

The screenshot shows a debugger interface with three main windows:

- RegistersView**: Shows general purpose registers R0-R15 and floating-point registers F0-F15. R0 is set to 00000005.
- Stack View**: Shows the stack pointer (R13) at address 00005400, containing the value 00000000.
- MemoryView0**: Shows memory starting at address 00001010, with values 81818181 repeated across the visible range.

The assembly code in the **p1.s** file is:

```
.TEXT  
00001000: E3A00005 MOV R0, #5  
00001004: E59F1000 LDR R1, =A  
00001008: E1010090 SWP R0, R0, [R1]  
.DATA  
00001010: A: .WORD 6
```

The screenshot shows the QEMU debugger interface with three main windows:

- RegistersView**: Shows general purpose and floating point registers in decimal, hexadecimal, and signed decimal formats. The PC register (R15) is highlighted in red with the value **R15 (pc): 0000100c**.
- p1.s**: Assembly code window showing the following instructions:

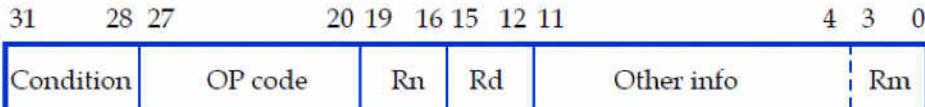

```
.TEXT
00001000:E3A00005    MOV R0, #5
00001004:E59F1000    LDR R1, =A
00001008:E1010090    SWP R0, R0, [R1]

.TEXT
00001010:             A: .WORD 6
```
- MemoryView0**: Memory dump window showing the memory starting at address 00001010. The word size is set to 32Bit. The memory dump shows the bytes: 00000005 81818181 81818181 81818181.

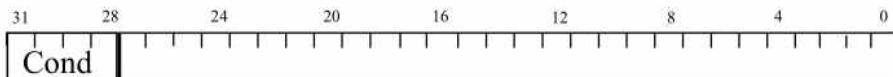
ENCODING instructions

OPcode{condition}{S} Rd, Operand1, Operand2

Instruction Format



Condition Field



0000 = EQ - Z set (equal)

1001 = LS - C clear or Z (set unsigned lower or same)

0001 = NE - Z clear (not equal)

1010 = GE - N set and V set, or N clear and V clear ($>=$)

0010 = HS / CS - C set (unsigned higher or same)

1011 = LT - N set and V clear, or N clear and V set ($>$)

0011 = LO / CC - C clear (unsigned lower)

1100 = GT - Z clear, and either N set and V set, or N clear and V set ($>$)

0100 = MI - N set (negative)

1101 = LE - Z set, or N set and V clear, or N clear and V set ($<$, or $=$)

0101 = PL - N clear (positive or zero)

1110 = AL - always → like no cond.

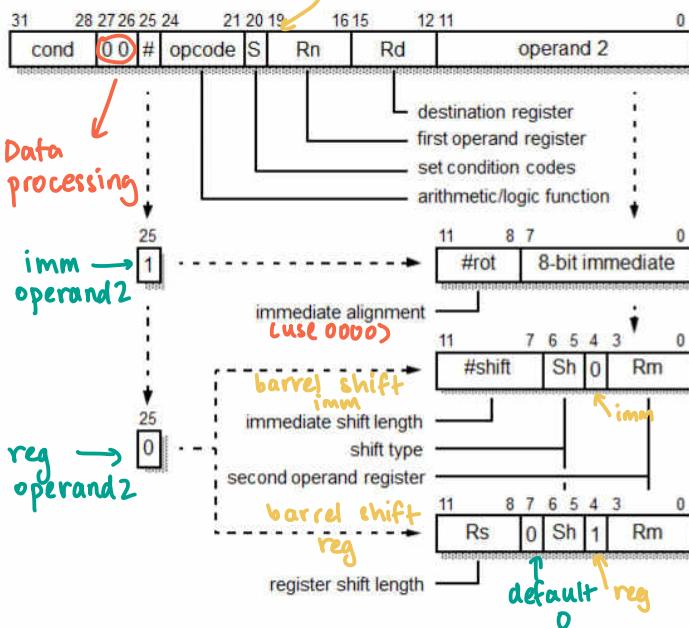
0110 = VS - V set (overflow)

1111 = NV - reserved.

1000 = HI - C set and Z clear
(unsigned higher)

DATA PROCESSING INSTRUCTION

CPSR set suffix



Opcode [24:21]	Mnemonic
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

operations

Example 1

ADD R0, R1, R2

31	28	27	26	25	24	21	20	19	16	15	12	11	7	6	5	4	3	0
Cond	00	#	Opcode	S	Rn		Rd		#shift		Sh	0	Rm					
no cond	00	0	0100	0	0001		0000		000000		00	0	0010					

= AL
= 1110

data processing (ADD)

0

AL	Rn	Rd	Rs
1110 0000 1000	0001	0000	0000 0000 0010
↓ ↓ ↓ ↓	0 1 0 0	0 0 0 2	

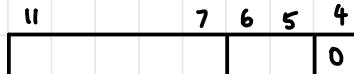
0xE0810002

ARM sim

.TEXT	
00001000:E0810002	ADD R0, R1, R2

shift type & amount

IMMEDIATE



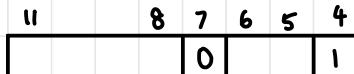
shift amount

5-bit unsigned

shift type

- 00 — logical left
- 01 — logical right
- 10 — arithmetic right
- 11 — rotate right

REGISTER



shift register

shift amount
specified in reg

default

shift type

- 00 — logical left
- 01 — logical right
- 10 — arithmetic right
- 11 — rotate right

Example 2

ADDS R1, R0, R2, LSR R4

31	28	27	26	25	24	21	20	19	16	15	12	11	7	6	5	4	3	0
Cond	00	#	Opcode	S	Rn		Rd		Rs		0	sh	1		Rm			



1110 0000 1001 0000 0001 0100 0011 0010
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
E 0 9 0 1 4 3 2

0xE0901432

.TEXT

00001000:E0901432

ADDS R1, R0, R2, LSR R4

Example 3

ADDS R1, R0, R2, LSR #2

31	28	27	26	25	24	21	20	19	16	15	12	11	7	6	5	4	3	0
Cond	00	#	Opcode	S	Rn		Rd		#shift	sh	0		Rm					



1110 0000 1001 0000 0001 0001 0010 0010
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
E 0 9 0 1 1 2 2

0xE0901122

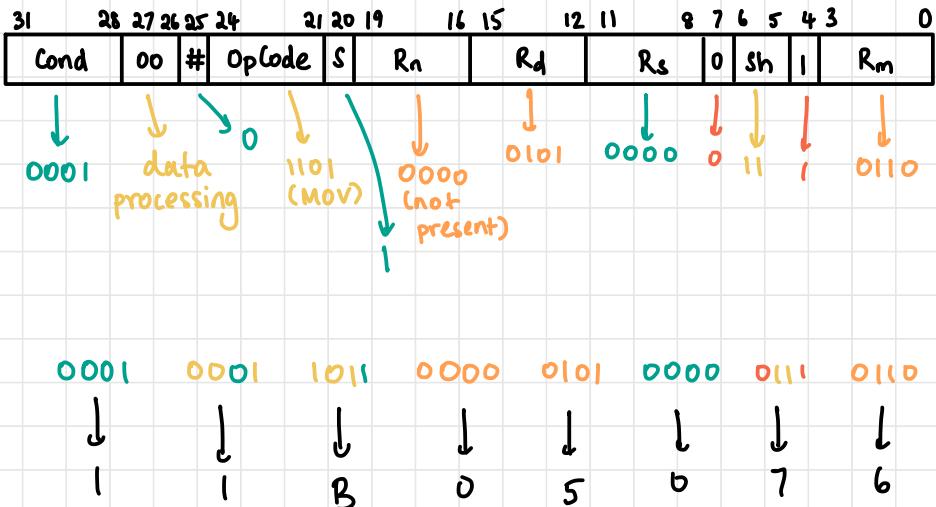
.TEXT

00001000:E0901122

ADDS R1, R0, R2, LSR #2

Example 4

Rd Rm Rs
MOVNES R5, R6, ROR R0



0x11B05076

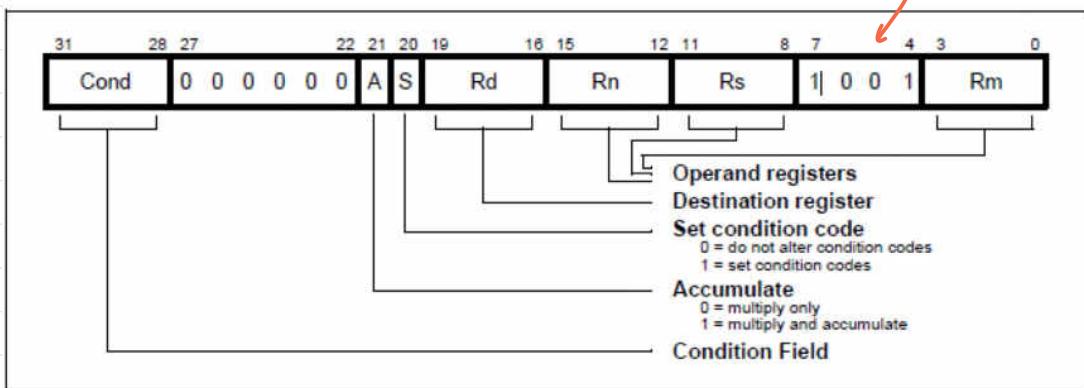
```
.TEXT
00001000:11B05076      MOVNES R5, R6, ROR R0
```

— BRANCH INSTRUCTIONS —



BL, B

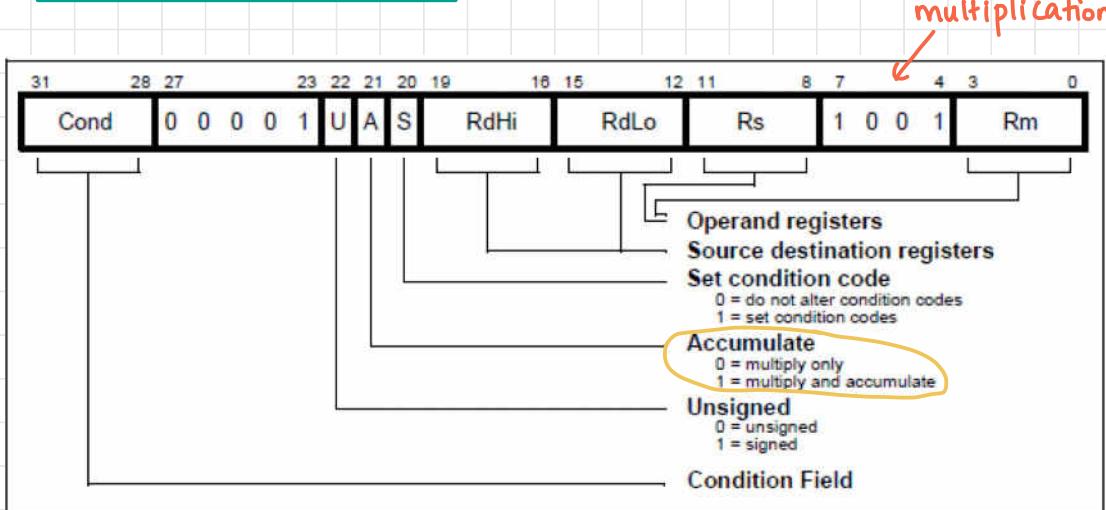
MULTIPLICATION INSTRUCTIONS



`MUL{<cond>}{S} Rd, Rm, Rs @ Rd = (Rm * Rs)`

`MLA{<cond>}{S} Rd, Rm, Rs, Rn @ Rd = Rm*Rs+Rn`

LARGE MULTIPLICATIONS



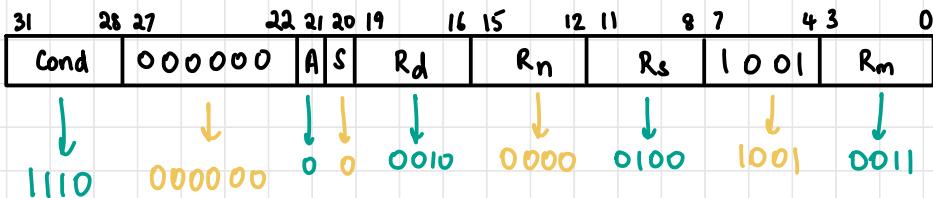
SMULL, UMLAL

`<SMLAL/SMULL/UMLAL/UMULL>{cond}{S} 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	$[RdHi, RdLo] = Rm * Rs$

Example 1

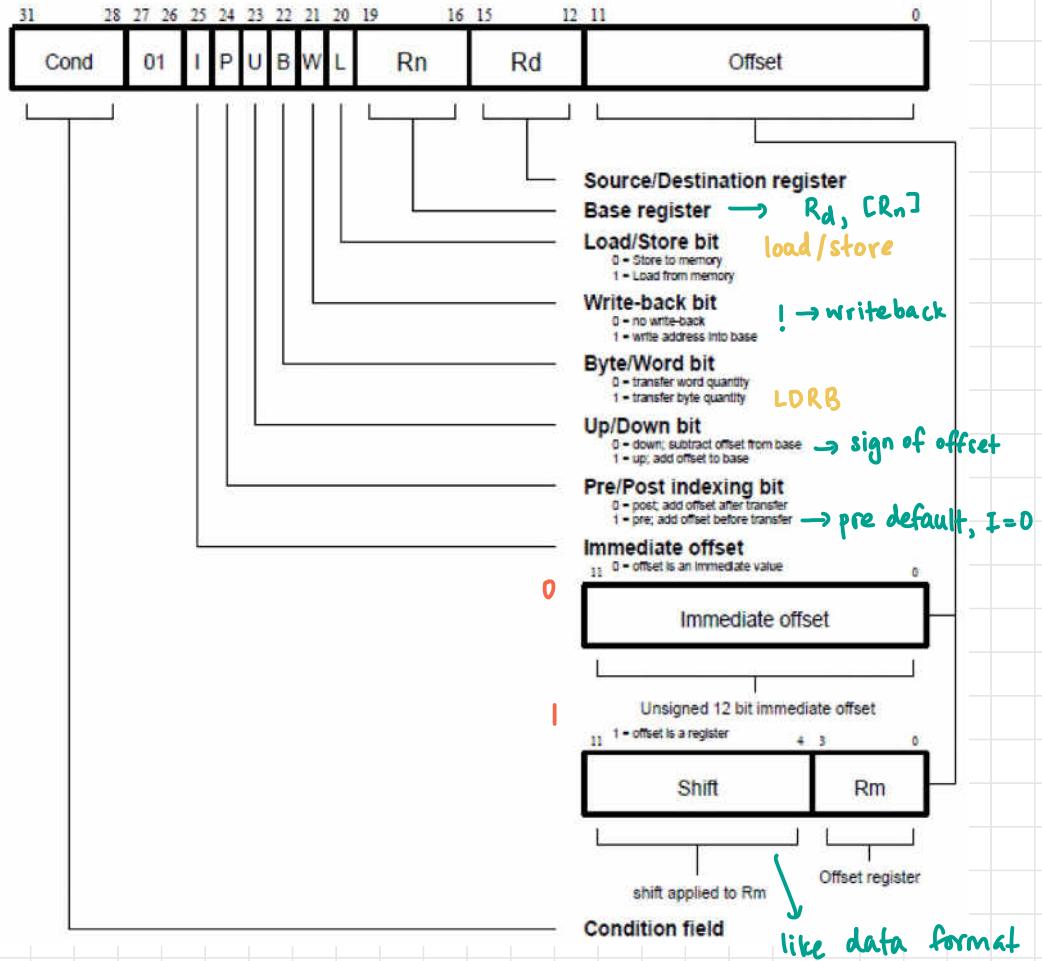
MUL R2, R3, R4



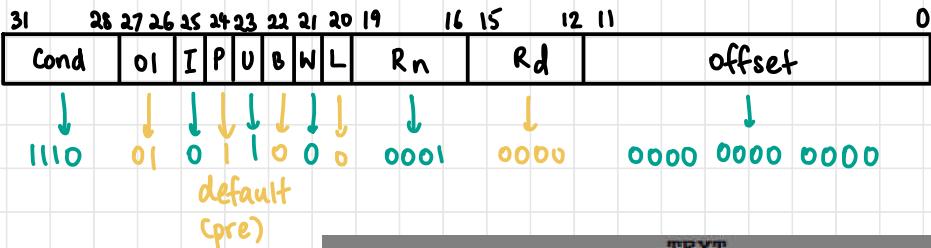
0xE0020493

```
.TEXT
00001000:E0020493      MUL R2, R3, R4
```

DATA TRANSFER INSTRUCTION



STR RO, [R1] default: pre increment no wb



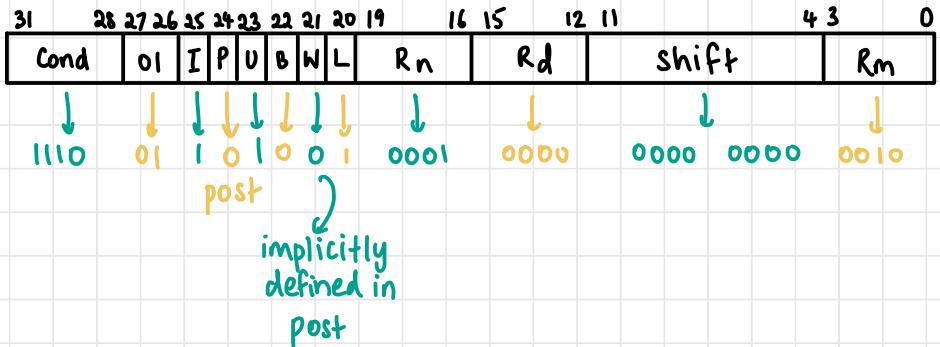
0xE5810000

00001000:E5810000

.TEXT

STR R0, [R1]

LDR R0, [R1], R2



0xE6910002

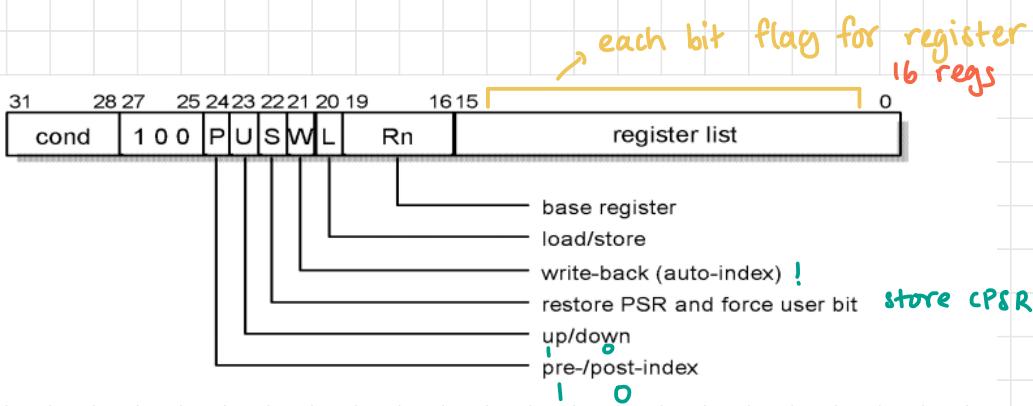
```
.TEXT
00001000:E6910002      LDR R0, [R1], R2
```

BLOCK TRANSFER

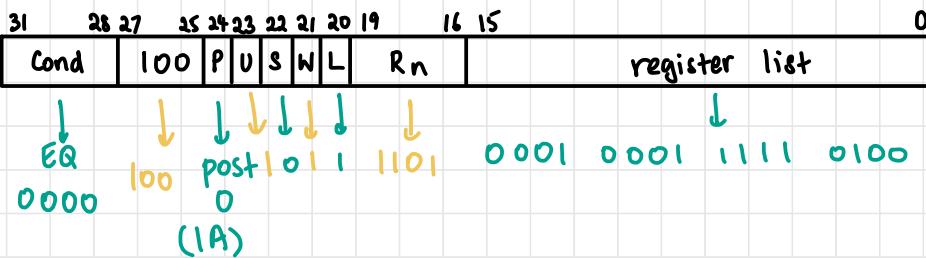
Values

Name	Stack	Other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

<LDM/STM> {cond} <Addressing Mode>Rn {!}, Registers



LDMEQIA R13!, {R4, R5-R8, R12, R2}

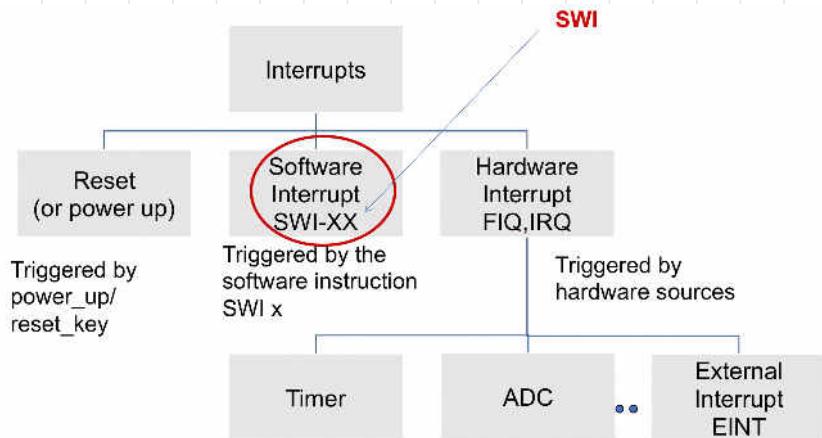


0x08BD11F4

.TEXT	00001000:08BD11F4	LDMEQIA R13!, {R4, R5-R8, R12, R2}
-------	-------------------	------------------------------------

INTERRUPTS

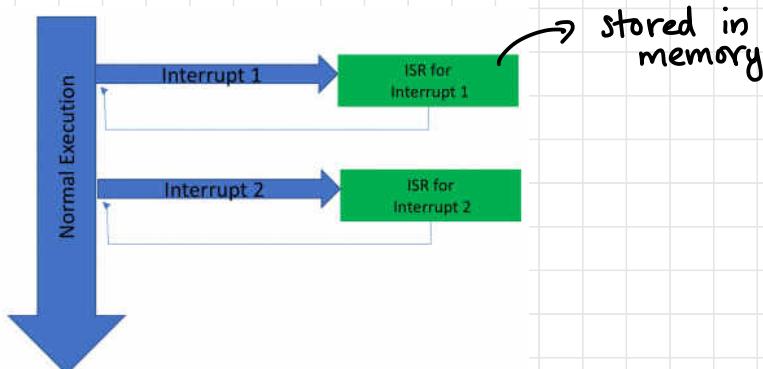
- normal program flow is disrupt
- interrupt service routine (ISR)



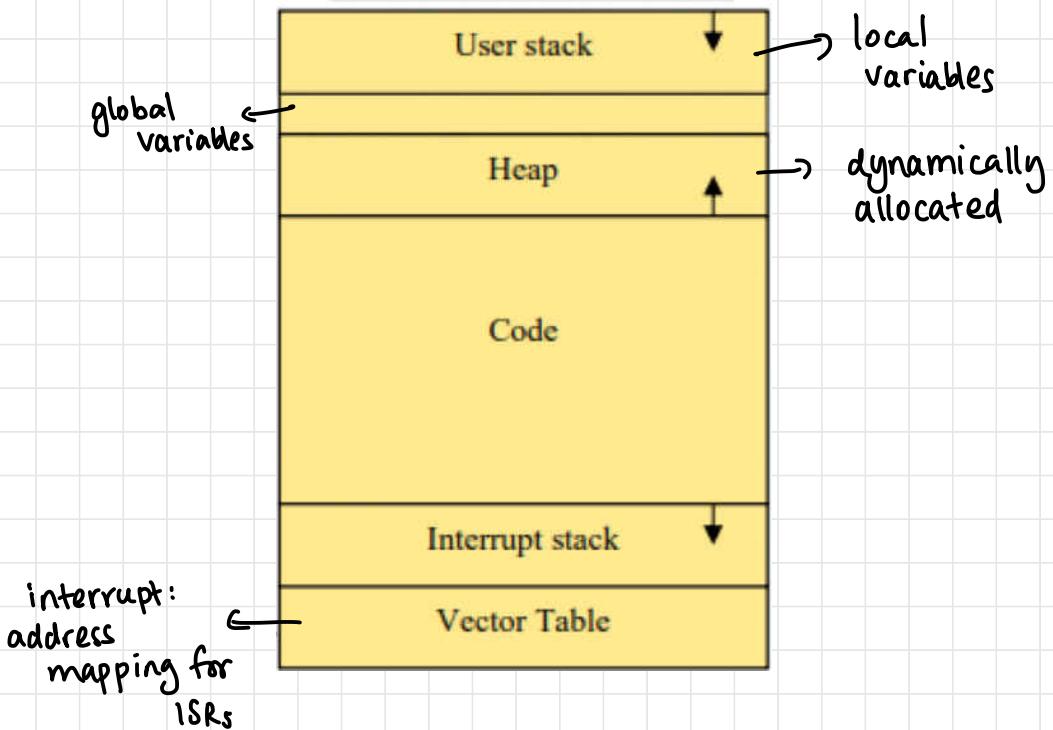
Serving Interrupts

- priority-based: polling
- all devices checked
- slow process

Event Driven Tasks Execution

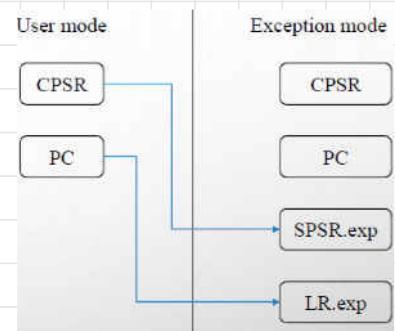


Main Memory

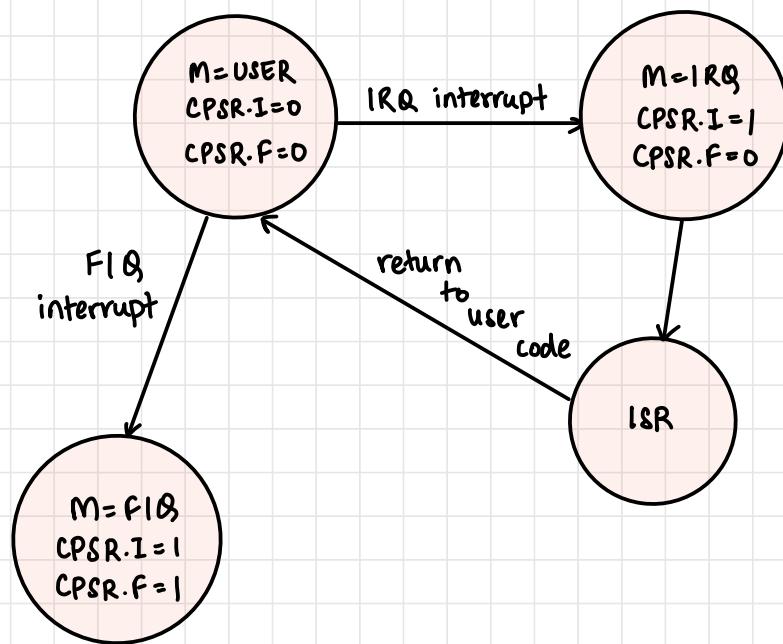


ARM Exception Handling

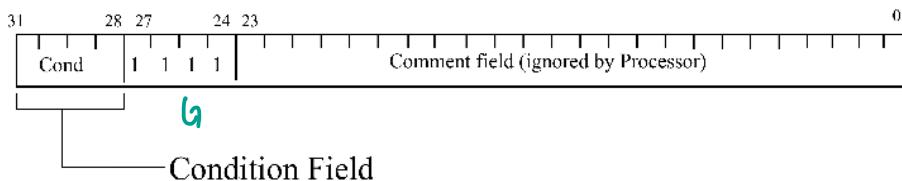
- CPSR → SPSR of exception mode
- PC → LR of exception mode
- CPSR set to exception mode
- PC → address of exception handler



FSM



software INTERRUPT



Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x00	Display Character on Stdout	r0: the character		SWI_PrChr
swi 0x02	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
swi 0x11	Halt Execution			SWI_Exit
swi 0x12	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	SWI_MAlloc
swi 0x13	Dallocate All Heap Blocks			SWI_DAlloc
swi 0x66	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	SWI_Open
swi 0x68	Close File	r0: file handle		SWI_Close
swi 0x69	Write String to a File or to Stdout	r0: file handle or Stdout r1: address of a null terminated ASCII string		SWI_PrStr

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x6a	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	SWI_RdStr
swi 0x6b	Write Integer to a File	r0: file handle r1: integer		SWI_Print
swi 0x6c	Read Integer from a File	r0: file handle	r0: the integer	SWI_RdInt
swi 0x6d	Get the current time (ticks)		r0: the number of ticks (milliseconds)	SWI_Timer