

MICROPROCESSOR AND COMPUTER ARCHITECTURE

UNIT-1

basic Processor architecture & design

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

introduction

TEXTBOOKS

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

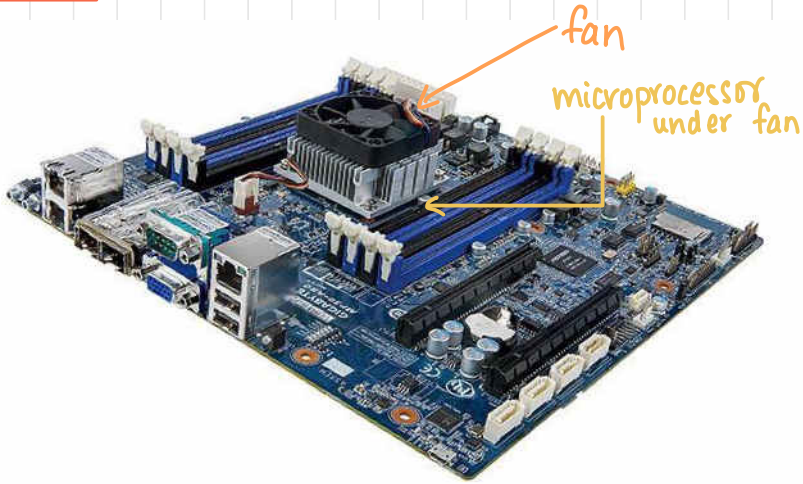
MPCA

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) faster than GPU
 - 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

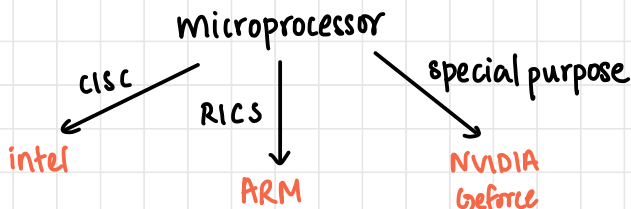
Motherboard



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.

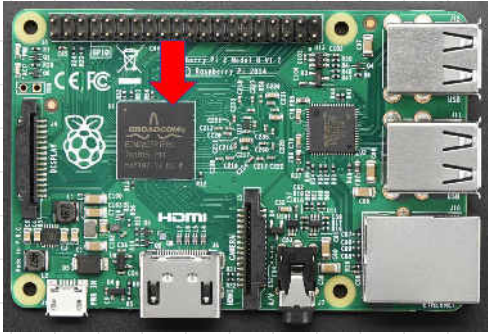
CISC

- 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.

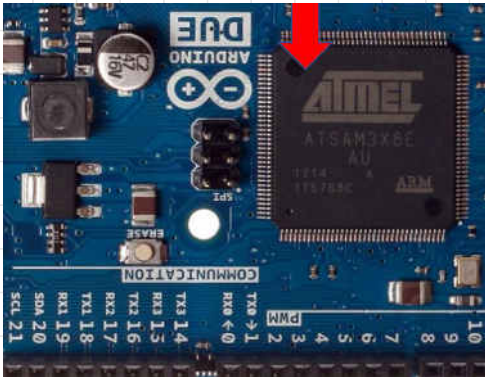
ARM (ACORN RISC MACHINE → ADVANCED RISC MACHINE)

- 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.

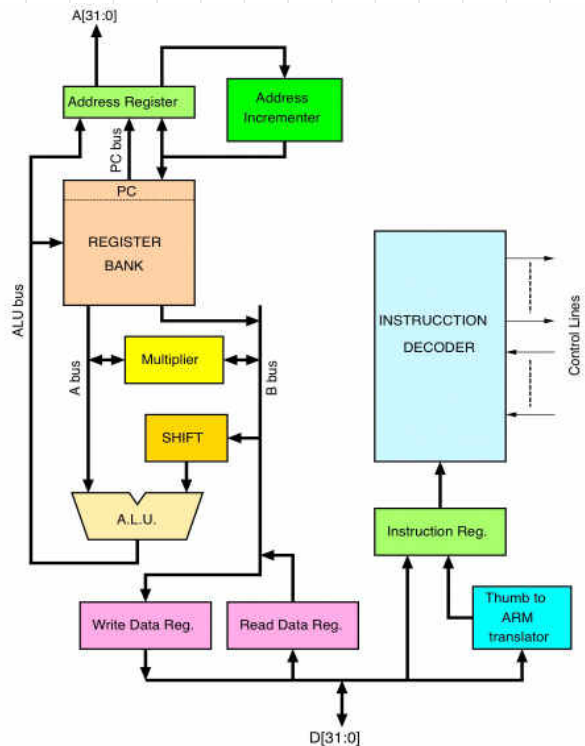
Version 4T: Thumbing: 16 bit instructions can be compressed in a 32 bit processor, thus enabling more instructions to be packed in the same memory, thereby increasing the code density.

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

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

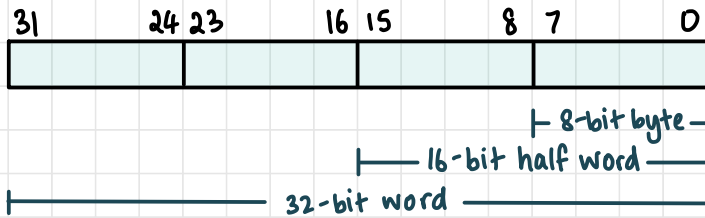
ARM7TDMI Processor

- 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
- 32x8 multiplier
- Barrel shifter



Data Sizes and Instruction Sets

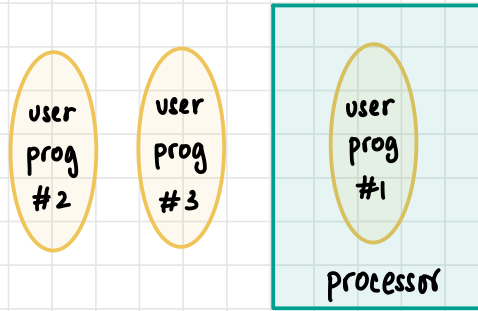
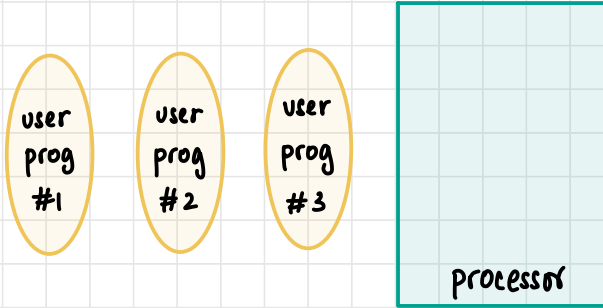
- 32-bit architecture ; byte-addressible
- **Byte:** 8-bit
Halfword: 16-bit ; aligned on 2-byte boundary
Word: 32-bit ; aligned on 4-byte boundary
- Two instruction sets:
 - 32-bit **ARM** instruction set
 - 16-bit **Thumb** instruction set



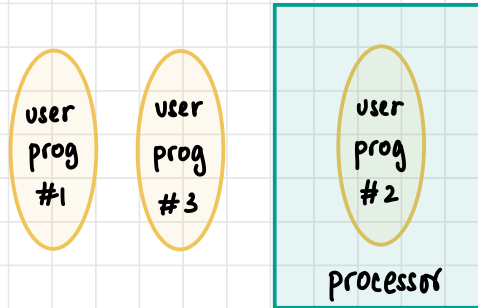
Processor Modes

- **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
- **Undefined:** handle undefined instructions
- **System:** privileged mode using same registers as user modes

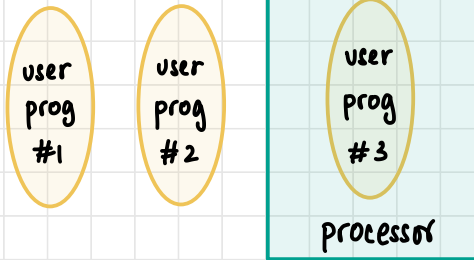
- interrupt service routines for each interrupt
- FIQ/IRQ pins on microprocessor
- eg: processor executing programs



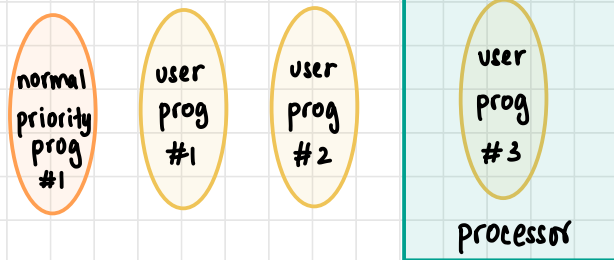
User prog #1
executing
user progs #2
and #3 waiting



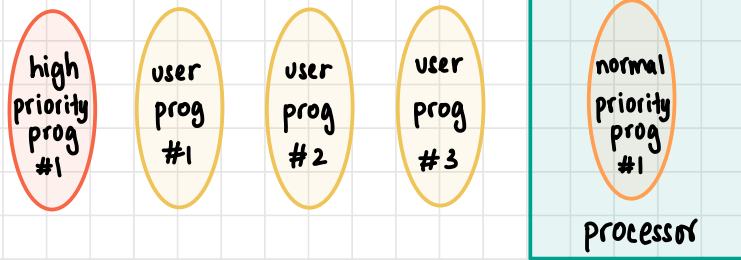
User prog #2
executing
context switching
between user
prog #1 and user
prog #2



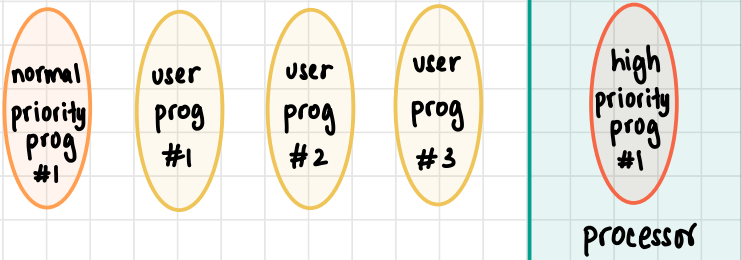
user prog #3
executing
context switching
between user
prog #2 and user
prog #3



priority prog
to be executed



priority progs
executed with
context switching
high priority prog to
be executed



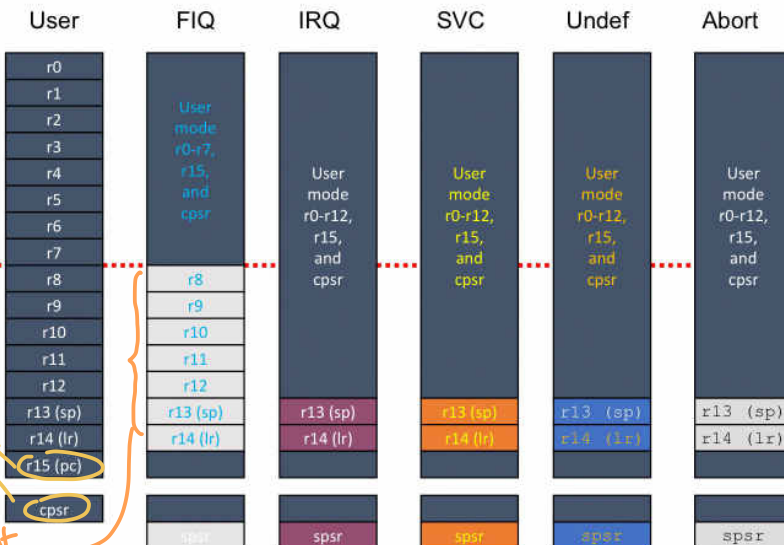
Register Bank

- 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
- Context Switch: CPSR contents moved to dedicated SPSR
- 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)

Register Organisation Summary

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

totally 37 regs



Thumb state Low registers

Thumb state High registers

common to all modes

different set of regs (not user)

5 spsr regs (one for each mode)

Note: System mode uses the User mode register set

Visible and Banked Out Registers

Current Visible Registers

User Mode

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

Banked out Registers

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

Banked out Registers

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

Current Visible Registers

IRQ Mode

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

Banked out Registers

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

Banked out Registers

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

Current Visible Registers

Abort Mode

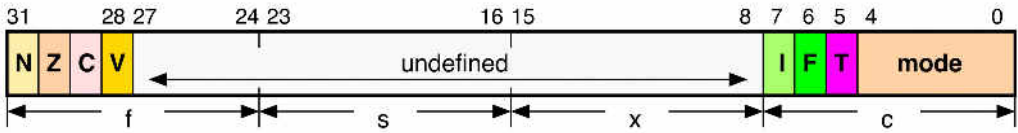
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
spsr
spcr

Banked out Registers

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

Status Register

- decision making using status register



- f: flag bits
 - s: status bits
 - x: extension bits
 - c: control bits
- } future enhancements;
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

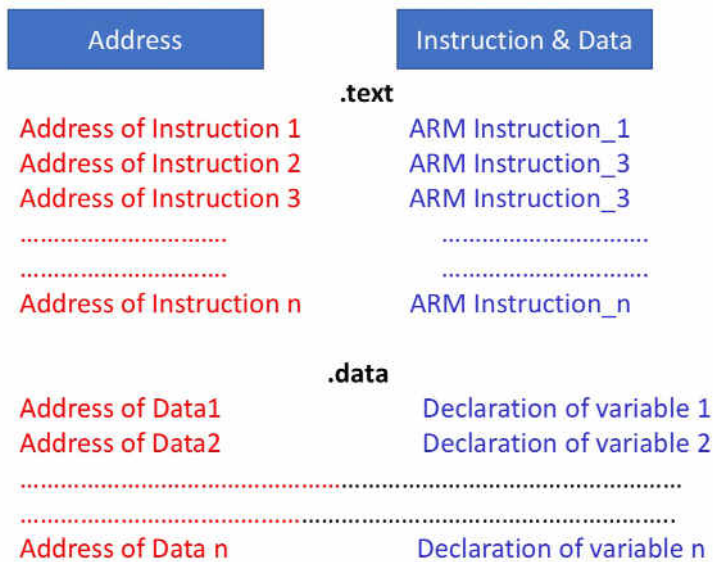
- Thumb is 16-bit instruction set
 - optimised for code density from C code
 - subset of ARM functionality
 - only low registers r0 to r7 used

- 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



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

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

step info

addr of inst

The screenshot shows the ARM Simulator interface with several panels:

- RegistersView:** Shows registers R0 through R15. R0 is highlighted with a value of 00000000. A red arrow points to the 'Flags' section below, which includes: Negative (N): 0, Zero (Z): 0, Carry (C): 0, Overflow (V): 0, IRQ Disable: 1, FIQ Disable: 1, Thumb (T): 0, CPU Mode: System, and 0x0000001F.
- CodeView:** Shows assembly code for 'p1.s'. The first three lines are highlighted: 'MOV R0, #10' at address 00001000:33A0000A, 'MOV R1, #20' at 00001004:33A01014, and 'ADD R2, R0, R1' at 00001008:30802001. A green arrow points to the address of the first instruction.
- MemoryView:** Shows a memory dump starting at 00000000. The first few lines are filled with 00000000, and subsequent lines are filled with 0xFFFFFFFF. A red arrow points to the 'Flags' section.
- StackView:** Shows a stack dump starting at 000113B0:01010101, with values ranging from 000113B0:01010101 to 00011420:????????.

flags

Data & Text

p1.s

.TEXT

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

.DATA

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

The screenshot shows the ARM Simulator interface with the following components:

- Registers View:** Shows registers R0 through R15 with values 0x00000000.
- Code View:** Displays the assembly code:

```
.TEXT
00001000:R3A01014 MOV R0, #10
00001006:R0802001  MOV R1, #20
0000100C:R0802001  ADD R2, R0, R1

.DATA
0000100C:R08000012 A: .WORD 0X12...
```
- Memory View:** Shows a memory dump starting at address 0000100C. The dump shows the word 0x00000012 at the end of the first instruction, which is the value of the constant 12 stored in little-endian format.
- Stack View:** Shows memory addresses from 00013000 to 00014500, mostly containing 0xFFFFFFFF.

↑ 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

The screenshot shows the ARM simulator interface with several panels:

- RegistersView:** Shows the CPSR register with the following status:
 - Negative(N): 1
 - Zero(Z): 0
 - Carry(C): 0
 - Overflow(O): 0
 - IRQ Disable: 1
 - FIQ Disable: 1
 - Thumb(T): 0
 - CPU Mode: System
- CodeView:** Shows the assembly code:

```
.TEXT  
00001000: E3A0000A  MOV R0, #10  
00001004: E3A00014  MOV R1, #20  
00001008: E0B02001  SUBS R2, R0, R1  
  
.DATA  
0000100C: 00000012  A: .WORD 0X12...
```
- MemoryView:** Shows memory at address 0000100C containing the value 00000012.
- StackView:** Shows a stack of memory addresses from 000113B0 to 00011450, mostly containing 0xFFFFFFFF.

flags reflected in CPSR

Load-Store Architecture

Address	Instruction	Meaning
	.text	
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, ascii

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

Syntax error if 2 imm or imm before reg

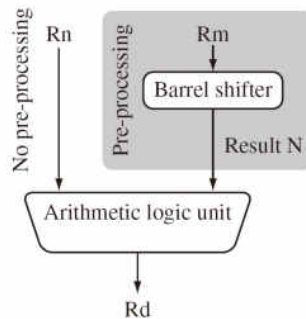
Operand2 - Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

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 - R_n , second operand is sent to ALU via barrel shifter



Data Movement

- MOV - operand2
- MVN - operand 2
 ↖ move negation
- no operand1
- $\langle \text{Operation} \rangle \{ \langle \text{cond} \rangle \} \{ S \} R_d, \text{Operand2}$

.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

```
R0      : 0000000a
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) : 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

0x600000df

Arithmetic Operations

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

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

- 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

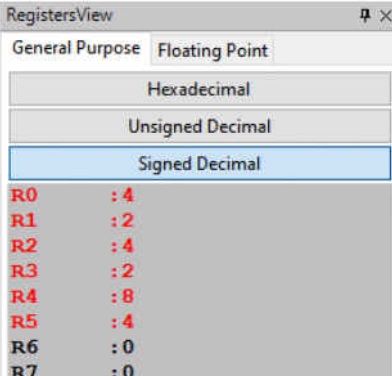
Operand2 and Carry in are inverted and fed to adder or Invert Operand2 first and subtract carry later

Wrong Application of SBC

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

should use SUBS

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



RegistersView

General Purpose Floating Point

Hexadecimal

Unsigned Decimal

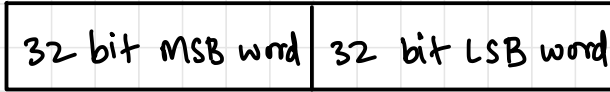
Signed Decimal

R0	: 4
R1	: 2
R2	: 4
R3	: 2
R4	: 8
R5	: 4
R6	: 0
R7	: 0

Multiword Arithmetic

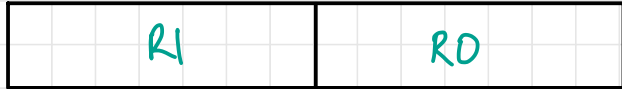
- operations for more than 32 bit operands, ADC, SBC, RSC are used

64-bit word



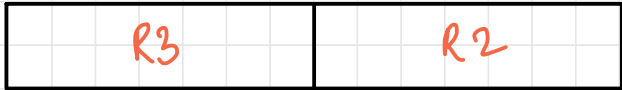
- add 2 64-bit words

ADDS R4, R0, R2

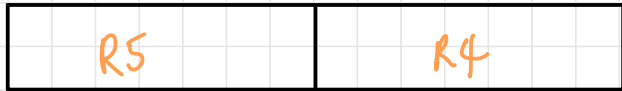


+

ADC R5, R1, R3



result

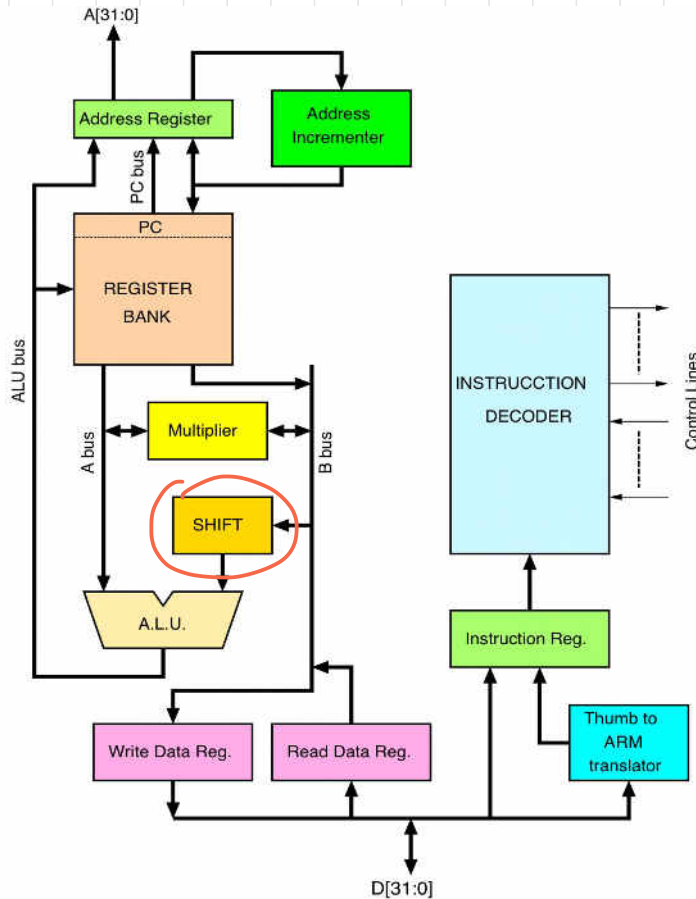


- subtract one 96-bit word from another

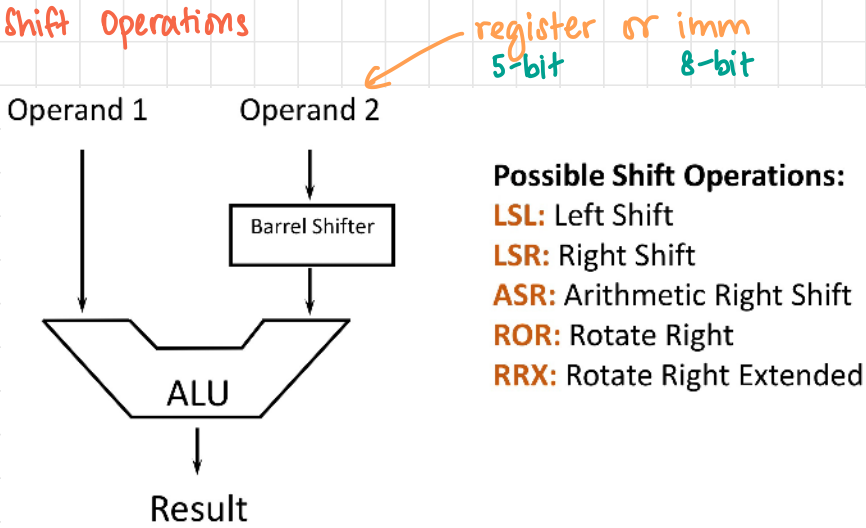
SUBS R3, R6, R9
SBCS R4, R7, R10
SBC R5, R8, R11

Barrel Shifter

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



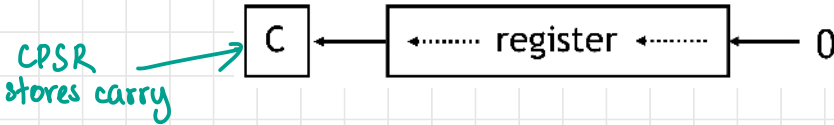
Barrel Shift Operations



• 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

The screenshot shows the ARMSim# interface with three main panes:

- RegistersView:** Lists registers R0 through R15. R2 is highlighted with a handwritten arrow and the text "R2 unchanged". R15 (pc) is 00001008.
- CodeView:** Shows assembly code for p2.o:

```
.TEXT
00001000:E3A02030    MOV R2, #0X00000030
00001004:E1A00102    MOV R0, R2, LSL #2
```
- MemoryView1:** Shows memory addresses from 00001000 to 00001060. The address 00001000 is selected.
- StackView:** Shows memory addresses from 000113B0 to 00011450, mostly containing 000113B0:81818181.

Logical Shift Right

- LSR

not
sign
extended



Syntax

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

before

11111111 11111111 11111111 11010000

after

00111111 11111111 11111111 11110100

- divided by 2^n

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

Output

$$3 + 256 \div 32$$
$$R3 = 11$$

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

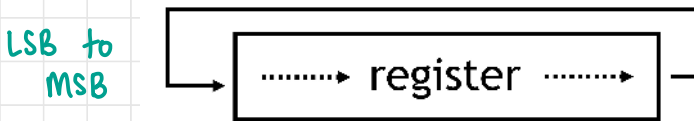
11111111 11111111 11111111 11010000

after

11111111 11111111 11111111 11110100

Rotate Right

- ROR



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

CPSR reflects

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
.END
```

R0-R1 is negative

```
R0 :00000019
R1 :00000100
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) :0000100e
-----
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
```

256 = 00000100
25 = 00000019
-25 = ffffffff e7

256 = 0'0'0'0'0'1'0'0
- 25 = f'f'f'f'f'f'e'7

0'0'0'0'0'0'e'7 → 231

①
↑
carry

Example 3

```
.TEXT
    MOV R0, #256
    MOV R1, #256
    CMP R0, R1
.END
```

```
R0 :00000100
R1 :00000019
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):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

```
R0 :00000100
R1 :00000100
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):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 : ffffffff
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): 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 : ffffffff
R1 : ffffffff
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): 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 (s1) : 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 (~R2)

BIC

Example 1

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

→ mask for which bits should get cleared

→ 0x 10011010

```
.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
```

complementz

```
R0 : 00000005
R1 : 00000006
R2 : 00000004
R3 : 00000007
R4 : 00000003
R5 : fffffffa
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/interworking-arm-and-thumb/basic-assembly-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:
```



backward branching

Conditional Branch

```
MOV R0, #0
```

```
loop:  
  ADD R0, R0, #1  
  CMP R0, #10 ← CPSR  
  BNE loop
```



branch if not equal

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
...
sub: ... @ sub entry point
...
MOV PC, LR @ return
```

would have been bypassed in regular B instruction

move LR contents back to PC (return)

Example 1

```
.TEXT
MOV R0, #5
MOV R1, #5
CMP R0, R1
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
```

RegistersView

General Purpose Floating Point

Hexadecimal
Unsigned Decimal
Signed Decimal

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

CodeView

p3.o

```
.TEXT
00001000:E3A00005 MOV R0, #5
00001004:E3A01005 MOV R1, #5
00001006:E1500001 CMP R0, R1
0000100C:0B000000 BLEQ label
00001010:E3A02006 MOV R2, #6

label:
00001014:E3A03014 MOV R3, #20
00001018:E1ADF00E MOV PC, LR...
```

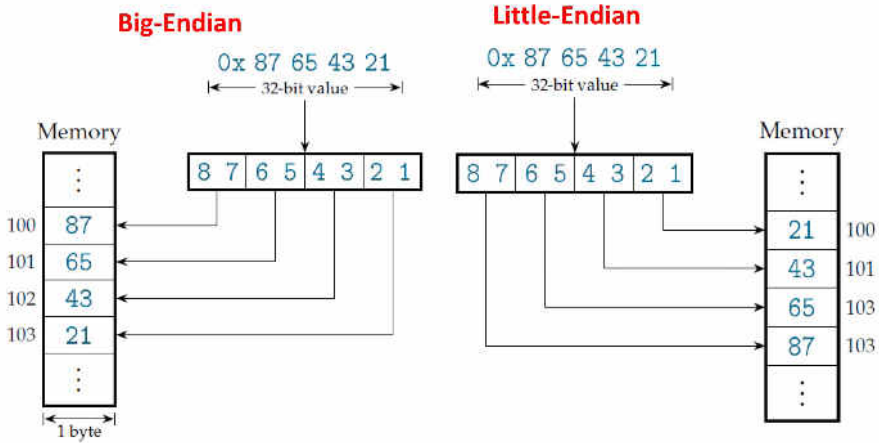
PC contents in LR

Data Transfer Instructions

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

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
0xFFFFFFF	00
0xFFFFFFF	00
0xFFFFFFF	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

$\langle \text{LDR/STR} \rangle \{ \langle \text{cond} \rangle \} \{ \text{B} \} \text{ (Rd), Addressing}$

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
Let B →	0x014	
	0x018	30
Let A →	0x01C	40
	0x020	50
	0x024	60

```
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]
```

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

Executed in ARMSim

.DATA

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

The screenshot displays the ARM simulator interface with three main panels:

- RegistersView:** Shows the state of 16 registers. R15 (PC) is highlighted in red and contains the value 00001010.
- CodeView:** Shows the assembly code. The instruction `STR R5, [R3]` at address 0000100C is highlighted in blue.
- MemoryView0:** Shows memory contents starting at address 0000101C. The value at 0000101C is 0000000A, which corresponds to the value of register R5.

Register	Value
R0	:00001018
R1	:00000000
R2	:00000000
R3	:0000101c
R4	:00000000
R5	:0000000a
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)	:00001010

Address	Instruction
00001000	E59F0008 LDR R0, =A
00001004	E5905000 LDR R5, [R0]
00001008	E59F3004 LDR R3, =B
0000100C	E5835000 STR R5, [R3]

Address	Value	Value	Value	Value	Value
0000101C	0000000A	81818181	81818181	81818181	81818181
00001030	81818181	81818181	81818181	81818181	81818181
00001044	81818181	81818181	81818181	81818181	81818181

Example 2

$$B = A + 9$$

.TEXT

LDR R0, =A

LDR R1, [R0]

ADD R2, R1, #9

LDR R3, =B

STR R2, [R3]

operator only supported
by LDR/STR instructions,
not ADD

← adds 9 to 0xA

.DATA

A: .WORD 10

B: .WORD

The screenshot shows a debugger interface with three main panes:

- RegistersView:** Shows the state of registers R0 through R15. R0 is 0000101c, R1 is 0000000a, R2 is 00000013, R3 is 00001020, R4 is 00000000, R5 is 00000000, R6 is 00000000, R7 is 00000000, R8 is 00000000, R9 is 00000000, R10 (s1) is 00000000, R11 (fp) is 00000000, R12 (ip) is 00000000, R13 (sp) is 00011400, R14 (lr) is 00000000, and R15 (pc) is 00001014.
- CodeView:** Shows assembly code for p1.o. The .TEXT section includes:
 - 00001000:E59F000C LDR R0, =A
 - 00001004:E5901000 LDR R1, [R0]
 - 00001008:E2812009 ADD R2, R1, #9
 - 0000100C:E59F3004 LDR R3, =B
 - 00001010:E5832000 STR R2, [R3]The .DATA section includes:
 - 0000101C:0000000A A: .WORD 10
 - B: .WORD...
- MemoryView0:** Shows memory at address 00001020. The value is 00000013. The memory is displayed in a table with columns for address and hex values.

Example 2

$C = A + B$ (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
B: .HWORD 20
C: .HWORD
```

} half word

.END

The screenshot shows a debugger interface with three main panes:

- RegistersView:** Shows the state of registers R0 through R15. R15 (PC) is highlighted in red and contains the value 0000101c.
- CodeView:** Displays the assembly code for the program. The .DATA section is highlighted, showing A: .HWORD 10, B: .HWORD 20, and C: .HWORD. The .TEXT section shows the assembly instructions: LDR R0, =A; LDR R1, =B; LDR R3, =C; LDRH R4, [R0]; LDRH R5, [R1]; ADD R6, R4, R5; STRH R6, [R3].
- MemoryView:** Shows the memory at address 00001028. The word size is set to 16Bit. The memory contains the values 000A 0014 001E 8181 8181. A red arrow points to the value 001E, which is highlighted in red. A handwritten note "little endian" is written above the memory view.

(see e-bit)

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: .ASCII "HELLO WORLD"
```

The screenshot displays a debugger interface with three main windows:

- RegistersView:** Shows the state of registers R0 through R15. R0 is 00001010, and R15 (pc) is 00001008. The CPSR Register is 0.
- CodeView:** Shows assembly code for p1.o. The .TEXT section contains:

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

The .DATA section contains:

```
MYSTR: .ASCII "HELLO WORLD"
```
- OutputView:** Shows the console output:

```
Loading assembly language  
Execution starting ...  
HELLO WORLD  
Execution ending, Instruct  
Instructions per second:63
```

At the bottom, the **MemoryView1** window shows the memory address 00001010 with a word size of 8Bit, displaying the hex value 48 45 4C 4C 4F 20 57 4F 52 4C, which corresponds to the ASCII string "HELLO WORL".

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:

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

pre indexing w/o writeback ←

- Register:

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

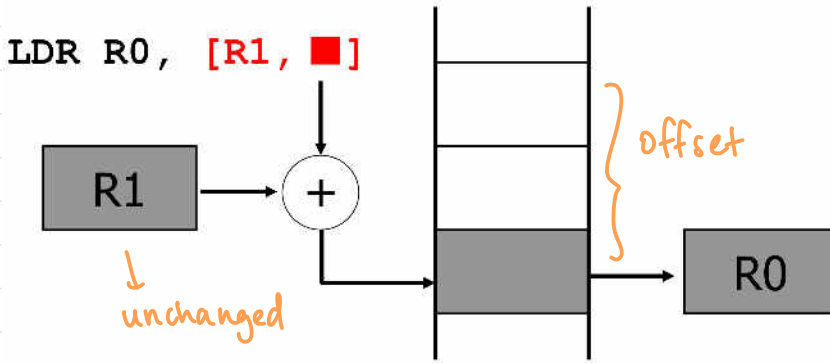
• Scaled Register type

has to be reg for scaled

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

PRE-INDEXING WITHOUT WRITEBACK

```
LDR R0, [R1, #4]
```



The screenshot shows a debugger interface with three panes:

- RegistersView:** Lists registers R0 through R15. R15 (pc) is highlighted with the value 00001008.
- CodeView:** Shows assembly code for `p1.o`. The instructions are:
 - `00001000:E59F0004 LDR R0, =A`
 - `00001004:E5901004 LDR R1, [R0, #4]` (The `#4` is circled in yellow and labeled "bytes")
 - `00001008:EF000011 SWI 0x11`
- MemoryView1:** Shows memory at address 00001010. The data is displayed in little endian format: `0A 00 00 00 14 00 00 00 1E 00 ...`. A yellow arrow points to the `0A` byte.

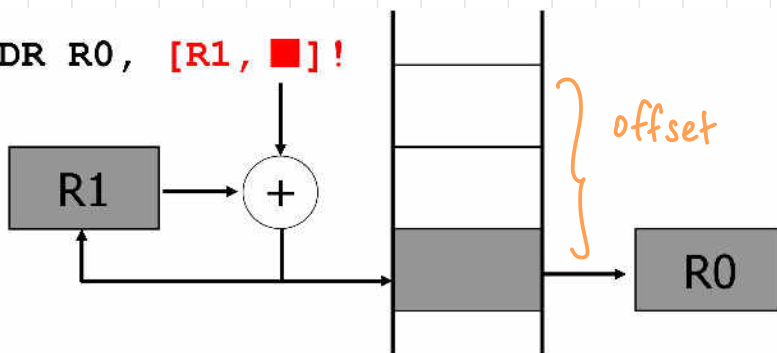
#4: 4 bytes
(#2: useless value)

AUTO INDEXING

PRE-INDEXING WITH WRITEBACK

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

LDR R0, [R1, ■]!



RegistersView

General Purpose Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : 00001014
R1 : 00000014
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

CodeView

p1.o

```
.TEXT
00001000:E59F0004    LDR R0, =A
00001004:E5B01004    LDR R1, [R0, #4]!
00001008:EF000011    SWI 0X11

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

.END
```

MemoryView1

00001010

Word Size

8Bit

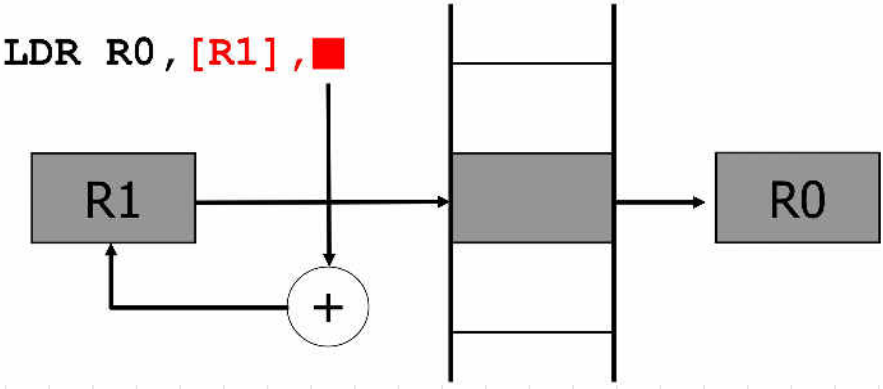
16Bit

32Bit

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

POST INDEXING

```
LDR R0, [R1], #4
```



The screenshot shows a debugger interface with three main windows:

- RegistersView:** Shows the state of registers R0 through R15. R0 is 00001014, R1 is 0000000a, and R15 (pc) is 00001008. The CPSR Register shows Negative (N) as 0 and Zero (Z) as 0.
- CodeView:** Shows assembly code for p1.o. The current instruction is LDR R1, [R0], #4 at address 00001004. The code includes .TEXT, .DATA, and .END sections.
- MemoryView1:** Shows memory contents starting at address 00001010. The word size is set to 32Bit. The memory contains values 0000000A, 00000014, 0000001E, 00000028, 00000032, 81818181, and 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 STM instructions
↳ load multiple registers

Syntax

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

A

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

Load at once

.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

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

Store at Once

The screenshot shows a debugger interface with three main windows:

- RegistersView:** Shows the state of registers R0 through R15. R15 (PC) is highlighted in red and set to 00001010. Below the registers, the CPSR register is shown with various flags like Negative (N), Zero (Z), Carry (C), etc.
- CodeView:** Displays assembly code for a file named p1.o. The code includes:


```

      .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:** Shows a memory dump starting at address 00001018. The dump shows five columns of 32-bit words. The values at addresses 00001028, 00001032, and 00001036 are highlighted in red, corresponding to the values 0000000A, 00000014, and 0000001E.

With Writeback

R8: 1024 (wb)

.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
```

The RegistersView window shows the following register values:

- R0: 0000000a
- R1: 00000014
- R2: 0000001e
- 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 : 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
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:E8B80007      LDMIA R8!, {R0, R1, R2} @ 1
0000100C:E8A90007      STMIA 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

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 (s1): 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      LDMDIA R8!, {R0, R1, R2} @
0000100C:E9A90007      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

00001018

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}
STMDA R7!, {R0-R2}
```

.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 (s1)	: 00000000	
R11 (fp)	: 00000000	
R12 (ip)	: 00000000	
R13 (sp)	: 00011400	
R14 (lr)	: 00000000	
R15 (pc)	: 00001014	

50
higher addr:
higher reg

The screenshot shows a debugger interface with three main panes:

- RegistersView:** Shows the state of 16 registers (R0-R15) and the CPSR Register. R9 is highlighted in red with the value 00001034. R15 (pc) is 00001018.
- CodeView:** Shows assembly code for the current instruction. The selected instruction is `STMDB R9!, {R0-R2}` at address 00001014. The code includes `.TEXT` and `.DATA` sections.
- MemoryView0:** Shows a memory dump starting at address 00001018. The selected word at 00001018 is 0000001E. The dump shows a sequence of 16 words, with some values highlighted in red.

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(s1):00000000
R11(fp):00000000
R12(ip):00000000
R13(sp):00011400
R14(lr):00000000
R15(pc):00001014
```

RegistersView Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : 00000014
 R1 : 0000001e
 R2 : 00000028
 R3 : 00000000
 R4 : 00000000
 R5 : 00000000
 R6 : 00000000
 R7 : 81818181
 R8 : 00001024
 R9 : 00001034
 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

CodeView

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: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 Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : e9290007
 R1 : 00001018
 R2 : 0000102c
 R3 : 00000000
 R4 : 00000000
 R5 : 00000000
 R6 : 00000000
 R7 : 00000000
 R8 : 0000100c
 R9 : 00001020
 R10 (s1) : 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	STMFD	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 displays three debugger windows:

- RegistersView:** Shows registers R0 through R15. R15 (pc) is highlighted with the value 00001010.
- CodeView:** Shows assembly code for the .TEXT section. The instruction STMEA R13, {R0, R1, R2} is highlighted in blue.
- StackView:** Shows memory addresses from 000013B0 to 00001410. The address 00001400 is highlighted.

Example 2

.TEXT

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

The screenshot displays a debugger interface with three main panels:

- RegistersView:** Shows the state of registers R0 through R15. R13 (sp) is highlighted in red and contains the value 0000140c. R14 (lr) contains 00000000, and R15 (pc) contains 00001010.
- CodeView:** Shows assembly code for the .TEXT section. The instruction at address 00001010 is highlighted: SWI 0X11...
- StackView:** Shows the stack memory layout, with addresses ranging from 000013BC to 0000141C. The address 0000140c is highlighted, corresponding to the stack pointer value in R13.

Additional details include the 'MemoryView' panel at the bottom showing address 00001028 and 'Word Size' options (8bit, 16bit, 32bit).

Example 3

.TEXT

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

RegistersView

Register	Value
R0	:00000001
R1	:00000002
R2	:00000003
R3	:00000001
R4	:00000002
R5	:00000003
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)	:00001014

CodeView

```

p1.o
    .TEXT
00001000:E3A00001    MOV R0, #1
00001004:E3A01002    MOV R1, #2
00001008:E3A02003    MOV R2, #3
0000100C:E9AD0007    STMFA R13!, {R0, R1, R2}
00001010:E93D0038    LDMFA R13!, {R3, R4, R5}
00001014:EF000011    SWI 0X11...
  
```

StackView

```

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
000013F8:81818181
000013FC:81818181
00001400:00000001
00001404:00000002
00001408:00000003
0000140C:81818181
00001410:81818181
  
```

(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 (IB)
pop, dec (DA)

RegistersView

Register	Value
R0	:00000001
R1	:00000002
R2	:00000003
R3	:00000001
R4	:00000002
R5	:00000003
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)	:00001014

CodeView

```

p1.o
    .TEXT
00001000:E3A00001    MOV R0, #1
00001004:E3A01002    MOV R1, #2
00001008:E3A02003    MOV R2, #3
0000100C:E9AD0007    STMFA R13!, {R0, R1, R2}
00001010:E93D0038    LDMFA R13!, {R3, R4, R5}
00001014:EF000011    SWI 0X11...
  
```

StackView

```

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
000013F8:81818181
000013FC:81818181
00001400:00000001
00001404:00000002
00001408:00000003
00001410:81818181
00001414:81818181
  
```

(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 displays a debugger interface with three main panels:

- RegistersView:** Shows the state of registers R0 through R15. R0 is 00000001, R1 is 00000002, R2 is 00000003, R3 is 00000001, R4 is 00000002, R5 is 00000003, and R13 (sp) is 00001400.
- CodeView:** Shows the assembly code for the example. The code is located at address 00001000 and includes instructions: MOV R0, #1; MOV R1, #2; MOV R2, #3; STMED R13!, {R0, R1, R2}; LDMED R13!, {R3, R4, R5}; and SWI 0X11...
- StackView:** Shows the stack memory starting at address 000013B0. The stack contains the 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, and 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}
```

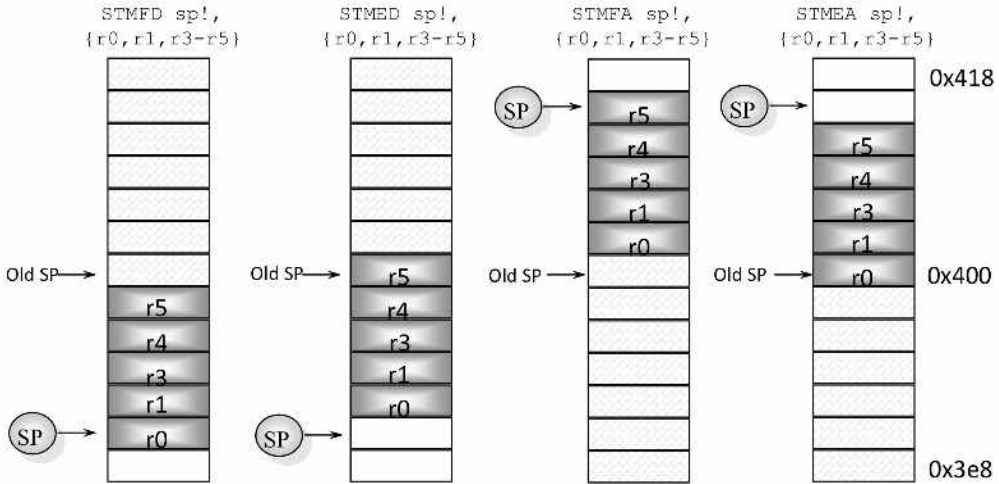
```
SWI 0X11
```

The screenshot displays a debugger interface with three main panels:

- RegistersView:** Shows the state of registers R0 through R15. R0 is 00000001, R1 is 00000002, R2 is 00000003, R3 is 00000001, R4 is 00000002, R5 is 00000003, and R15 (pc) is 00001014.
- CodeView:** Shows assembly code for the .TEXT section. The current instruction is SWI 0X11... at address 00001014:EF000011.
- StackView:** Shows memory addresses from 000013B0 to 00001410, all containing the value 81818181.

The MemoryView1 panel at the bottom shows the address 00001028 and the Word Size set to 32Bit.

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	
0x0020	Procedure: Instruction 1
0x0024	Instruction 2
0x0028	Instruction 3
0xxxxx	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
```

The screenshot shows a debugger window with two panes. The left pane displays the register file, and the right pane displays the assembly code.

Register File:

Register	Value
R0	:0000000a
R1	:0000001e
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

Assembly Code:

```
.TEXT
00001000:E3A0000A  MOV R0, #10
00001004:E2801014  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 Memory/View1 pane shows the current instruction at address 00001028.

RegistersView

General Purpose Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : 0000000a
 R1 : 0000001e
 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) : 0000100c
 R15 (pc) : 00001014

CodeView

p1.o

```

.TEXT
00001000:E3A0000A    MOV R0, #10
00001004:E2801014    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
  
```

MemoryView1

00001028

Word Size: 8Bit 16Bit 32Bit

RegistersView

General Purpose Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : 0000000a
 R1 : 0000001e
 R2 : 00000005
 R3 : 00000000
 R4 : 00000000
 R5 : 00000008
 R6 : 00000000
 R7 : 00000000
 R8 : 00000000
 R9 : 00000000
 R10 (s1) : 00000000
 R11 (fp) : 00000000
 R12 (ip) : 00000000
 R13 (sp) : 00001400
 R14 (lr) : 0000100c
 R15 (pc) : 0000100c

CodeView

p1.o

```

.TEXT
00001000:E3A0000A    MOV R0, #10
00001004:E2801014    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
  
```

MemoryView1

00001028

Word Size: 8Bit 16Bit 32Bit

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
```

The screenshot displays a debugger interface with three main windows: RegistersView, CodeView, and StackView.

RegistersView: Shows the state of various registers. R15 (pc) is highlighted in red and contains the value 00001018. Other registers like R0, R1, R2, etc., contain zero or specific values.

CodeView: Displays assembly code. The .TEXT section includes instructions for MOV, STMFD, BL, LDR, and STR. The ADDFUNC section includes LDMFD, ADD, and MOV. The .DATA section defines variable A as .WORD 0.

StackView: Shows memory addresses from 00001380 to 00001400, all containing zero.

MemoryView: Shows memory addresses from 00001028 to 000010AB. Address 00001028 contains 00000003. Address 00001029 contains 81818181. Address 00001030 contains 81818181. Address 00001031 contains 81818181. Address 00001032 contains 81818181. Address 00001033 contains 81818181. Address 00001034 contains 81818181. Address 00001035 contains 81818181. Address 00001036 contains 81818181. Address 00001037 contains 81818181. Address 00001038 contains 81818181. Address 00001039 contains 81818181. Address 0000103A contains 81818181. Address 0000103B contains 81818181. Address 0000103C contains 81818181. Address 0000103D contains 81818181. Address 0000103E contains 81818181. Address 0000103F contains 81818181. Address 00001040 contains 81818181. Address 00001041 contains 81818181. Address 00001042 contains 81818181. Address 00001043 contains 81818181. Address 00001044 contains 81818181. Address 00001045 contains 81818181. Address 00001046 contains 81818181. Address 00001047 contains 81818181. Address 00001048 contains 81818181. Address 00001049 contains 81818181. Address 0000104A contains 81818181. Address 0000104B contains 81818181.

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 panels:

- RegistersView:** Lists registers R0 through R15. R0-R15 are shown in hexadecimal. Below the registers, various system flags and CPU mode are displayed.
- CodeView:** Shows assembly code for a function named `MULFUNC`. The code includes instructions like `MOV R0, #1`, `MOV R1, #2`, `MOV R2, #3`, `STMPD R13!, {R0, R1, R2}`, `BL MULFUNC`, `LDR R6, =A`, `STR R7, [R6]`, and `SWI 0x11`.
- StackView:** Shows a stack of memory addresses, with `00001408` highlighted.
- MemoryView1:** Shows a memory dump starting at `00001048`. The dump shows a sequence of 32-bit values, with `00000009` at the first address.

MULTIPLICATION INSTRUCTIONS

MUL	Multiply	32-bit result
MLA	Multiply accumulate	32-bit result
UMULL	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

long data (64) (handwritten note pointing to UMULL)

2 registers (handwritten note in a bracket next to the 64-bit result entries)

MUL

MUL{<cond>}{S} **Rd, Rf, Rs** @ Rd = (Rf * 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 displays two windows from a debugger. The left window, titled 'RegistersView', shows the state of 16 registers. Registers R0 through R15 are listed, with their values in hexadecimal. R0 is 00000003, R1 is 00000002, and R2 is 00000006. The right window, titled 'CodeView', shows the assembly code for the example. The code is as follows:

```
p1.o
      .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 a debugger interface with three main windows: RegistersView, CodeView, and MemoryView.

- RegistersView:** Shows the state of 16 registers (R0-R15). R0-R3 contain values 00000000, 00000000, 00001058, and 00000000 respectively. R4-R15 contain values from 0000001c to 00011400. Below the registers, the CPSR register is shown with fields like Negative(N):0, Zero(Z):1, Carry(C):1, Overflow(O):0, IRQ Disable:1, FIQ Disable:1, and Thumb(T):0.
- CodeView:** Displays assembly code for a program named 'pl.o'. The code includes:
 - Initialization: `LDR R0, =A`, `LDR R1, =B`, `LDR R2, =C`, `MOV R3, #3`, `MOV R4, #0`.
 - Loop: `LOOP:` `LDR R5, [R0], #4` (with a handwritten note "post indexing" and an arrow pointing to the register update), `LDR R6, [R1], #4`, `MUL R7, R5, R6`, `ADD R4, R4, R7`, `SUB R3, R3, #1`, `CMP R3, #0`, `BNE LOOP`.
 - Final instruction: `STR R4, [R2]`.
- MemoryView:** Shows memory addresses from 00001040 to 00001090. The values at 00001050, 00001060, 00001070, 00001080, and 00001090 are all 81818181. The value at 00001058 is 0000001c, which corresponds to the value in register R4.

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
```

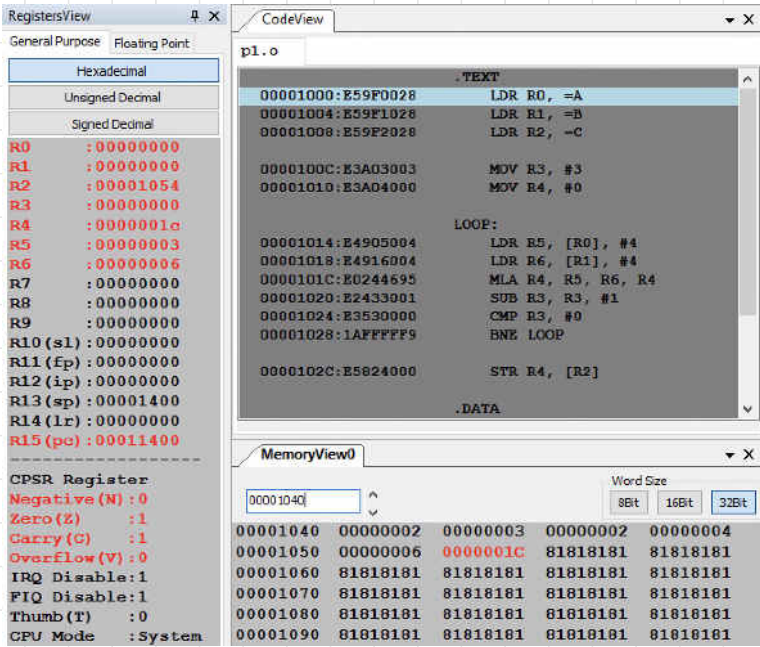
replaces
MUL &
ADD



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

.DATA

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



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 displays a debugger interface with two main windows: RegistersView and CodeView.

RegistersView: Shows the state of various registers. R0 is 0x00000000, R1 is 0x400000df, and R15 (pc) is 0x00010008. The CPSR Register is also visible with fields like Negative (N), Zero (Z), Carry (C), and Overflow (V).

CodeView: Shows assembly code for a file named 'pl.o'. The code includes:
00001000:E3800000 MOVS R0, #0
00001004:E10F1000 MRS R1, CPSR
00001008:EF000011 SWI 0X11...

MemoryView: Shows a memory dump starting at address 00001040. The dump consists of a grid of 4 columns of 32-bit words, with values ranging from 0x81818181 to 0x01818181.

MSR

move to status register from reg (write)

Example:

MSR CPSR_field, R0

MSR SPSR_field, R1

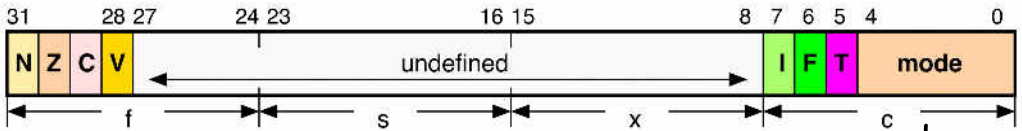
_field

_c: Control Field (0:7)

_f: Flag Field(24:31)

_x: Extension (8:15)

_s: Status (16:23)



Example 1

.TEXT

MOV R0, #0XF000000

MSR CPSR_f, R0

SWI 0X11

mode:
1000 - user
1111 - system
(page 13)

0xF = 1111

only first 8 bits

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 two main windows. The 'RegistersView' window on the left displays the state of ARM registers. Register R0 is highlighted in red and contains the value 00000005. Register R1 contains 00000000. The 'MemoryView0' window on the right shows the assembly code and memory dump for a file named 'p1.s'. The assembly code is as follows:

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

.DATA
00001010:          A: .WORD 6
```

The memory dump at the bottom shows the address 00001010 containing the value 00000006, which corresponds to the data at label A. The word size is set to 32bit.

The screenshot shows a debugger interface with two main panes. The top pane, titled 'RegistersView', displays the state of registers R0 through R15. R0 is highlighted in red and contains the value 00000006. The bottom pane, titled 'MemoryView0', shows the current instruction at address 00001010: SWP RO, RO, [R1]. The assembly code in the background includes:

```

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

.DATA
00001010:          A: .WORD 6

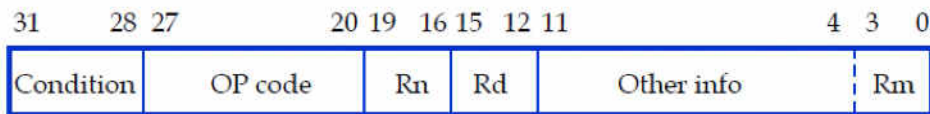
```

ENCODING

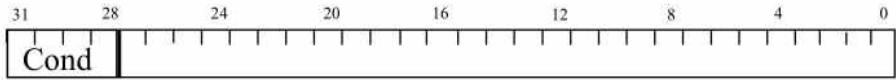
instructions

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

Instruction Format

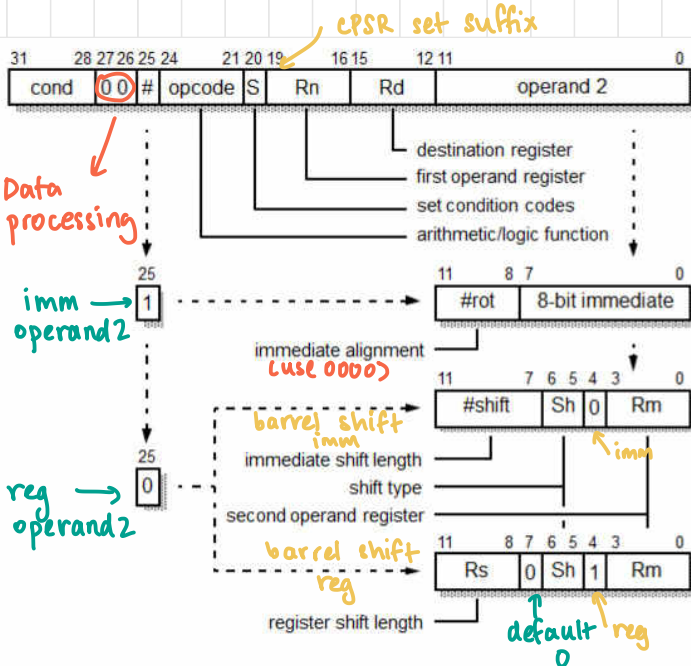


Condition Field



- 0000 = EQ - Z set (equal)
- 0001 = NE - Z clear (not equal)
- 0010 = HS / CS - C set (unsigned higher or same)
- 0011 = LO / CC - C clear (unsigned lower)
- 0100 = MI - N set (negative)
- 0101 = PL - N clear (positive or zero)
- 0110 = VS - V set (overflow)
- 0111 = VC - V clear (no overflow)
- 1000 = HI - C set and Z clear (unsigned higher)
- 1001 = LS - C clear or Z (set unsigned lower or same)
- 1010 = GE - N set and V set, or N clear and V clear (>or=)
- 1011 = LT - N set and V clear, or N clear and V set (>)
- 1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)
- 1101 = LE - Z set, or N set and V clear, or N clear and V set (<, or =)
- 1110 = AL - always → like no cond.
- 1111 = NV - reserved.

DATA PROCESSING INSTRUCTION

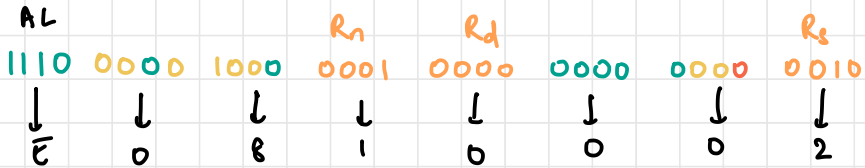
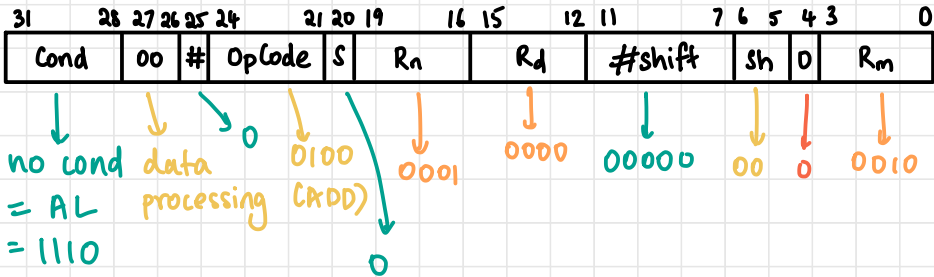


Opcode [24:21]	Mnemonic
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

operations

Example 1

ADD R0, R1, R2



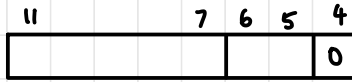
0xE0810002

ARM sim

```
.TEXT
00001000:E0810002    ADD R0, R1, R2
```

shift type & amount

IMMEDIATE

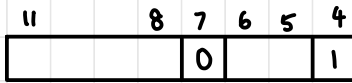


↓
shift amount
5-bit unsigned

↘ shift type

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

REGISTER



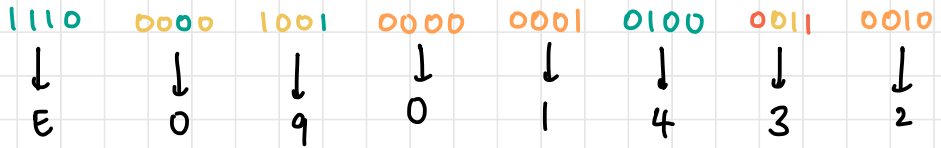
↓ shift register
shift amount
specified in reg

↓ default
↘ shift type

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

Example 2

ADDS R1, R0, R2, LSR R4

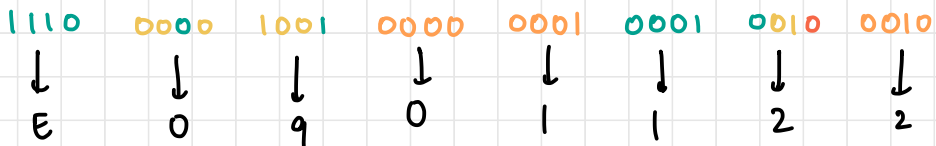
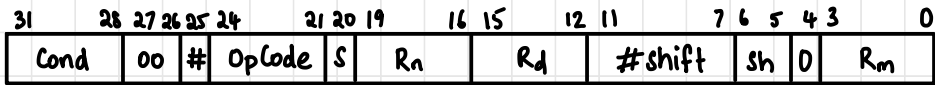


0xE0901432

```
.TEXT  
00001000:E0901432    ADDS R1, R0, R2, LSR R4
```

Example 3

ADDS R1, R0, R2, LSR #2

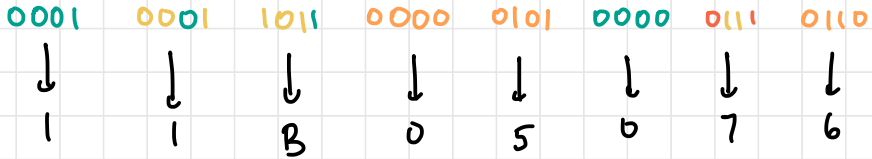
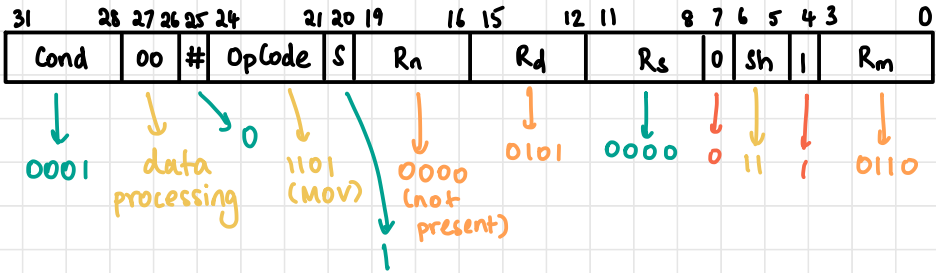


0xE0901122

```
.TEXT  
00001000:E0901122    ADDS R1, R0, R2, LSR #2
```

Example 4

MOVNES R5, R6, ROR R0

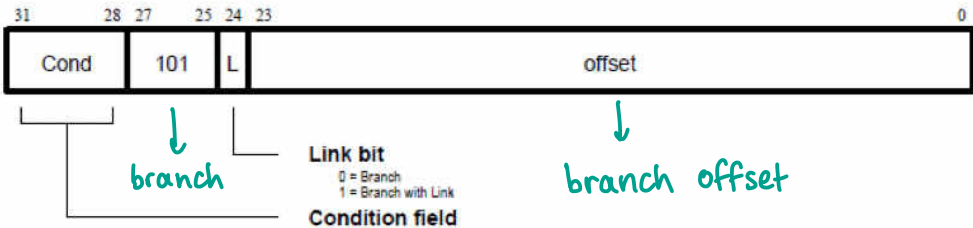


0x11B05076

```

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

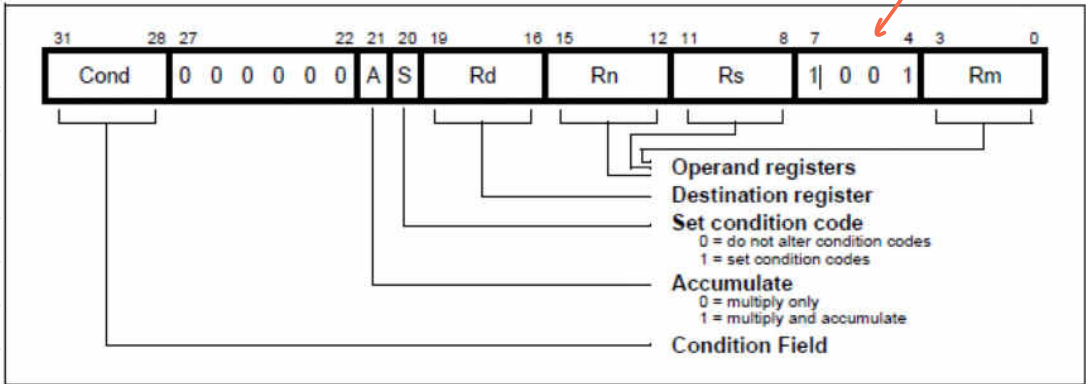
BRANCH INSTRUCTIONS



BL, B

MULTIPLICATION INSTRUCTIONS

multiplication

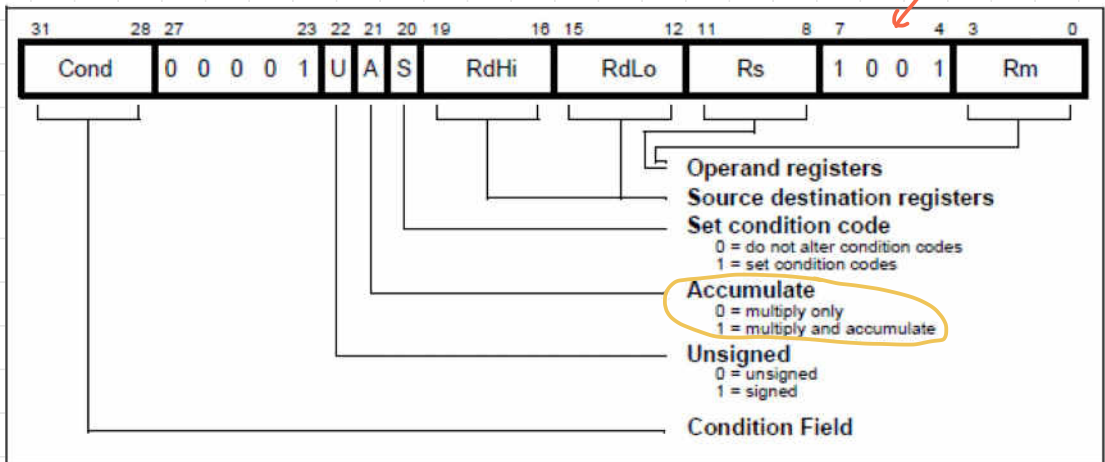


$MUL\{\langle cond \rangle\}\{S\} Rd, Rm, Rs @ Rd = (Rm * Rs)$

$MLA\{\langle cond \rangle\}\{S\} Rd, Rm, Rs, Rn @ Rd = Rm * Rs + Rn$

LARGE MULTIPLICATIONS

multiplication



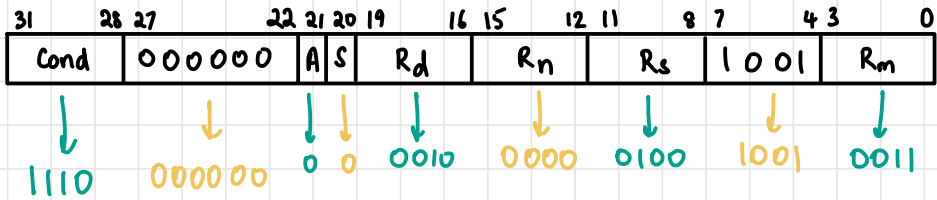
SMULL, UMLAL

$\langle SMLAL/SMULL/UMLAL/UMULL \rangle\{\langle cond \rangle\}\{S\} RdLo, RdHi, Rm, Rs$

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



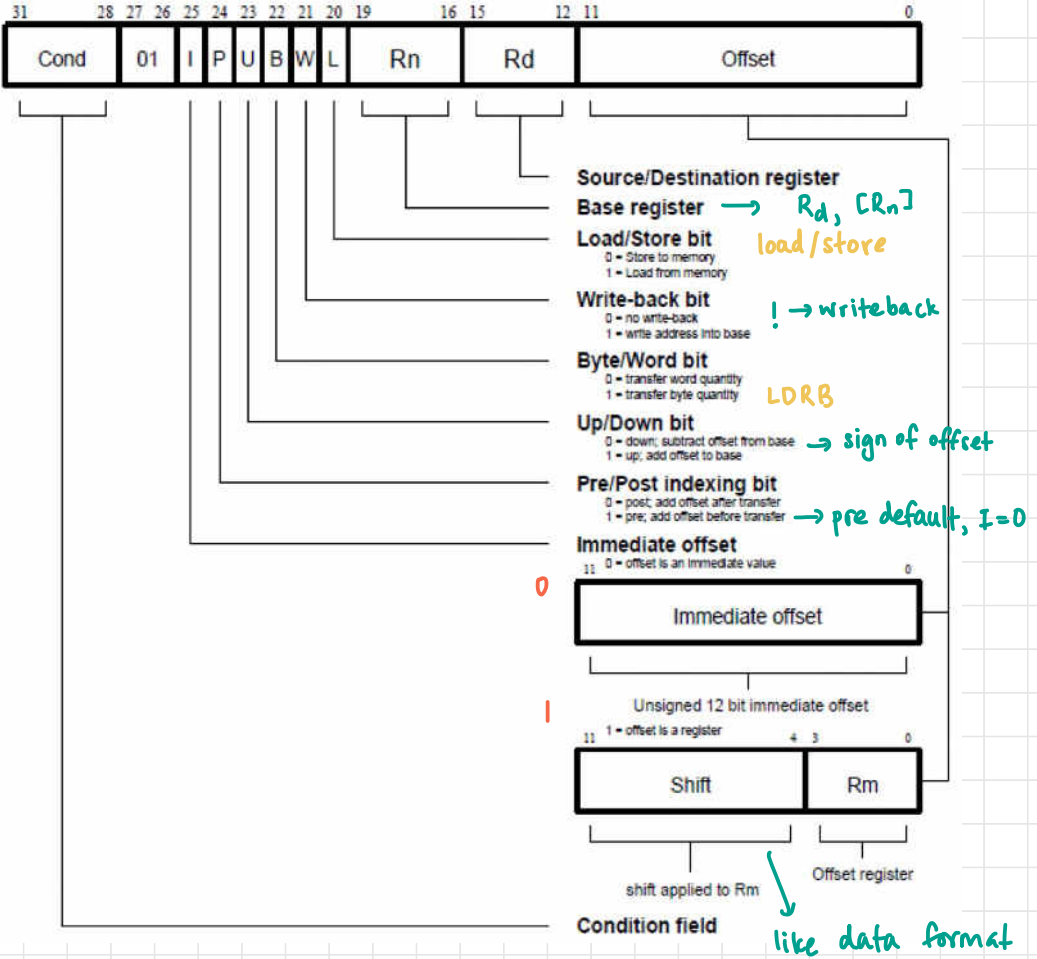
0xE0020493

```

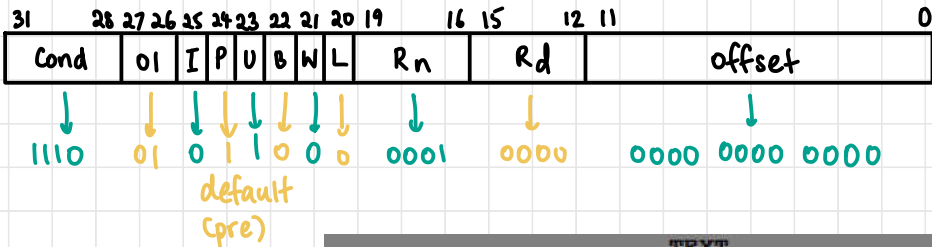
.TEXT
00001000:E0020493      MUL R2, R3, R4

```

DATA TRANSFER INSTRUCTION



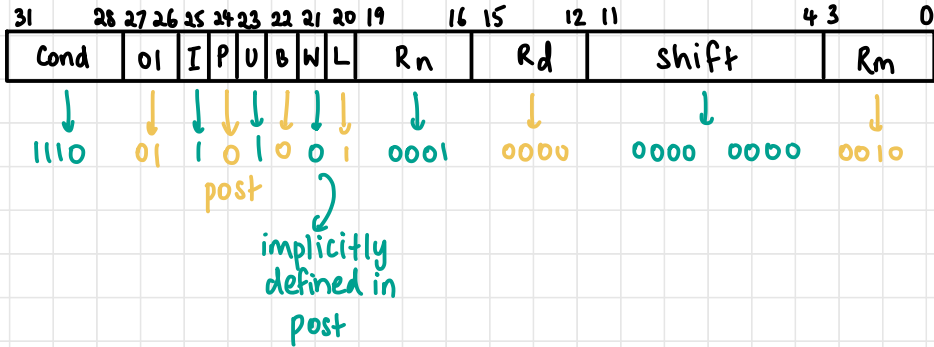
STR R0, [R1] default: pre increment no wb



0xE5810000

```
.TEXT
00001000:E5810000 STR R0, [R1]
```

LDR R0, [R1], R2



0xE6910002

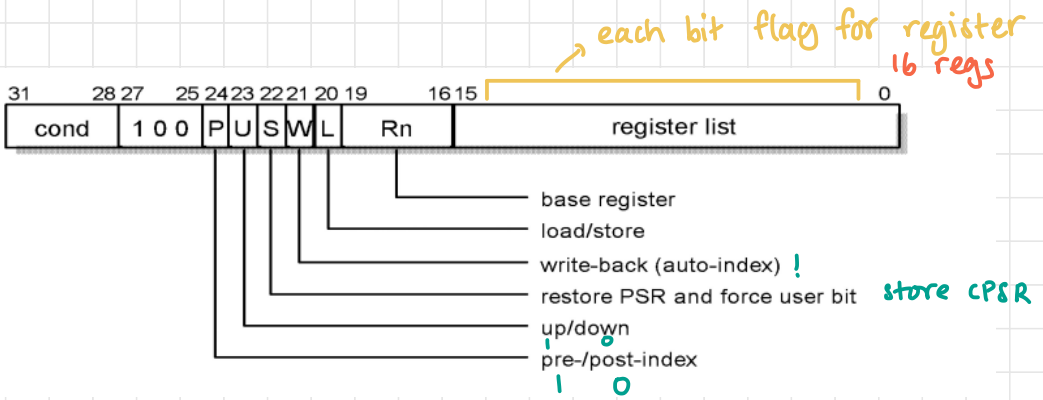
```
.TEXT
00001000:E6910002    LDR R0, [R1], R2
```

BLOCK TRANSFER

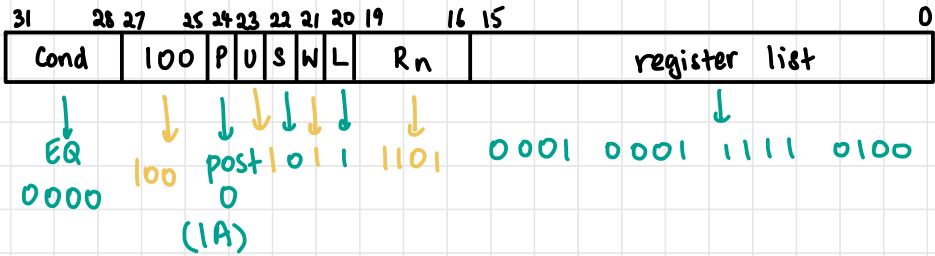
Values

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}

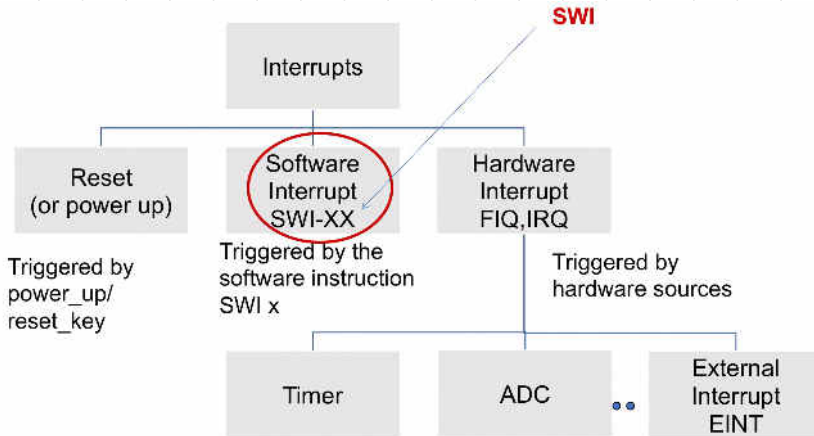


0x08BD11F4

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

INTERRUPTS

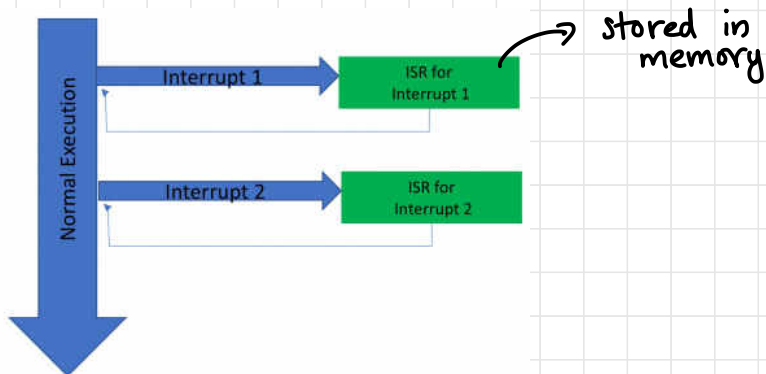
- normal program flow is disrupt
- interrupt service routine (ISR)



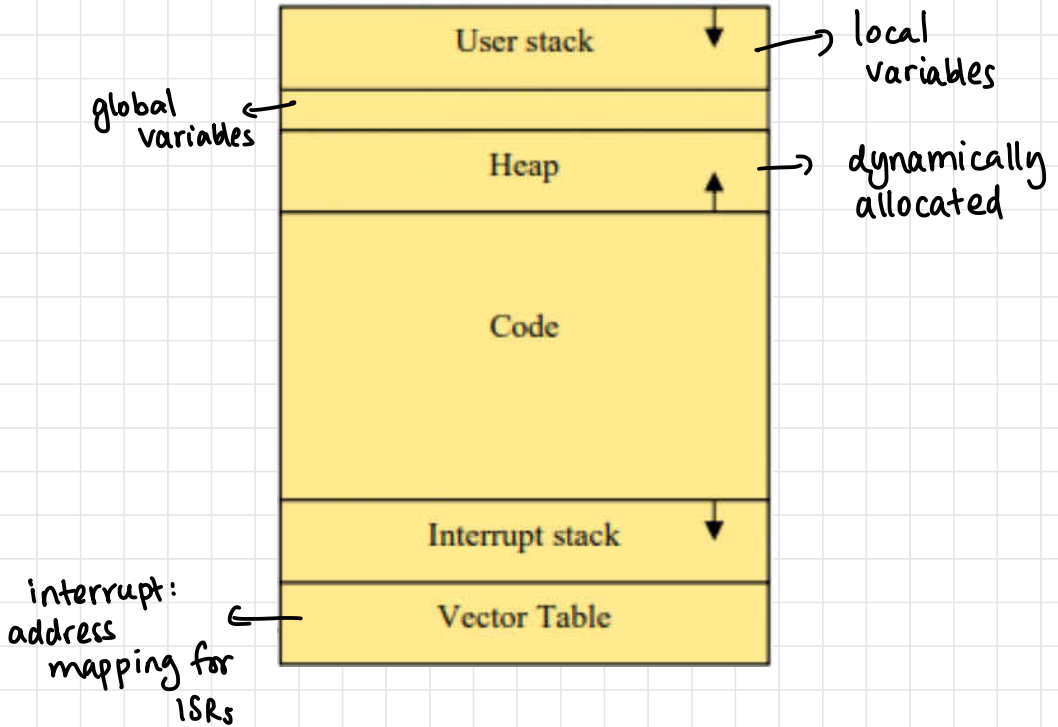
Serving Interrupts

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

Event Driven Tasks Execution

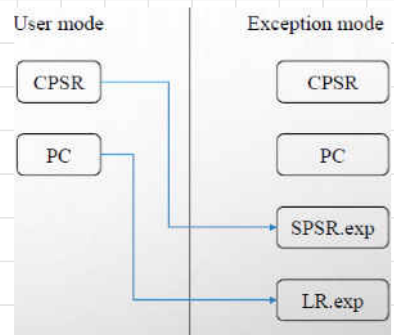


Main Memory

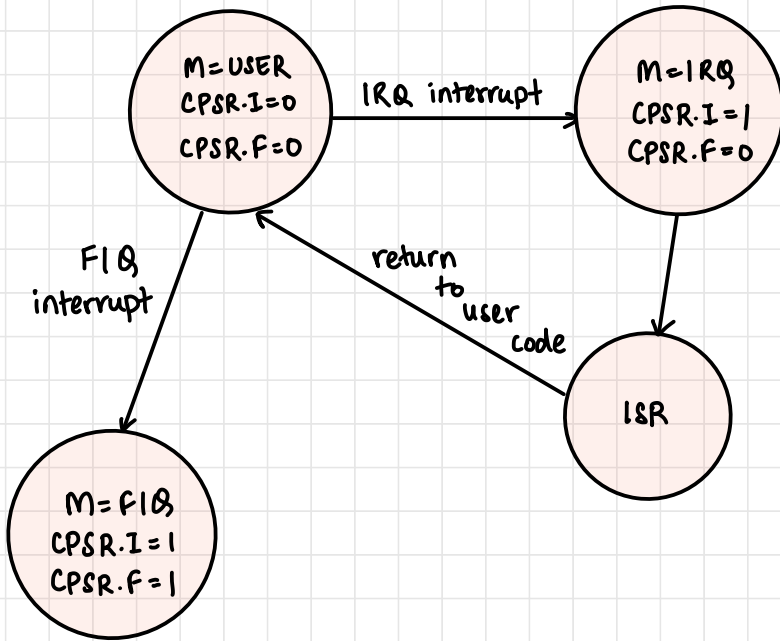


ARM Exception Handling

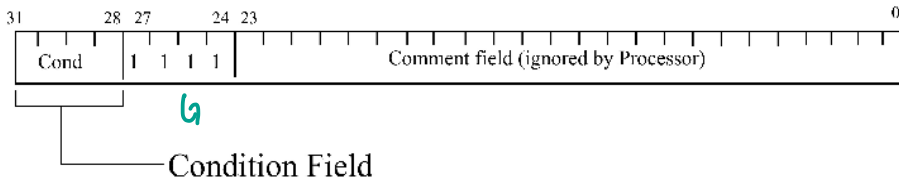
- CPSR \rightarrow SPSR of exception mode
- PC \rightarrow LR of exception mode
- CPSR set to exception mode
- PC \rightarrow address of exception handler



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_MeAlloc
swi 0x13	Deallocate 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