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, 5^{th} edition

2. "ARM System on a Chip" - Steve Furbur, 2^{nd} 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

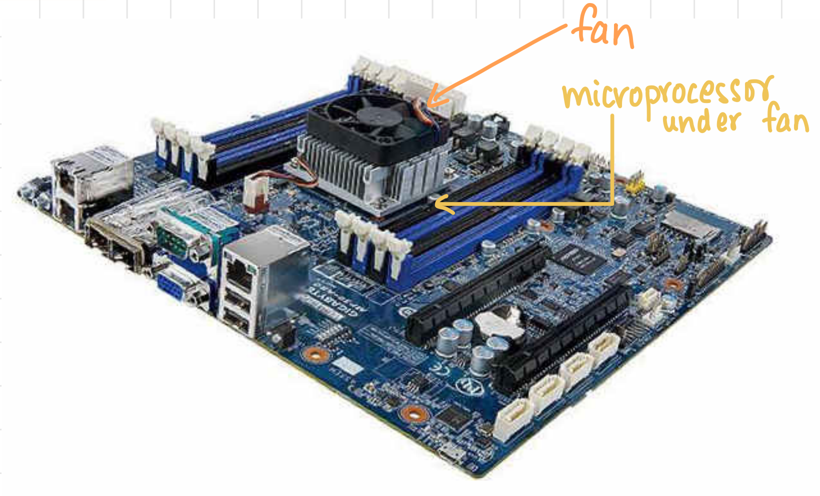
- Single chip implementation of CPU

- Not all microprocessors are CPUs
    - GPU (image processing)
    - TPU (tensors, matrix multiplication for image processing) faster than GPU
    - 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

microprocessor

- CISC → intel
- RICS → ARM
- special purpose → NVIDIA Geforce

- 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

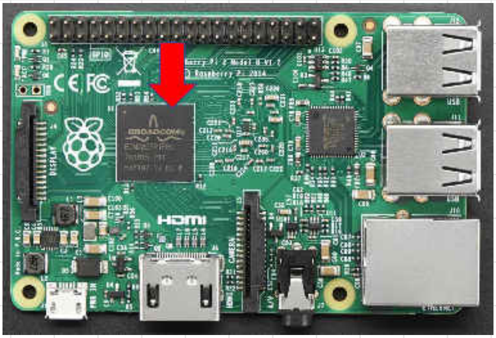
RISC vs CISC

RISC	CISC
• Simple instructions, few in number.	• Many complex instructions.
• Fixed length instructions.	• Variable length instructions.
• Multiple register sets.	• Single register set.
• Three operands per instruction.	• One or two register operands per instruction.
• Parameter passing through register windows.	• Parameter passing through memory.
• Single-cycle instructions.	• Multiple cycle instructions.
• Hardwired control.	• Microprogrammed control.
• Highly pipelined.	• Less pipelined.
• Complexity in compiler.	• Many instructions can access memory.
• Only **LOAD/STORE** instructions access memory.	• Many addressing modes.
• Few addressing modes.	

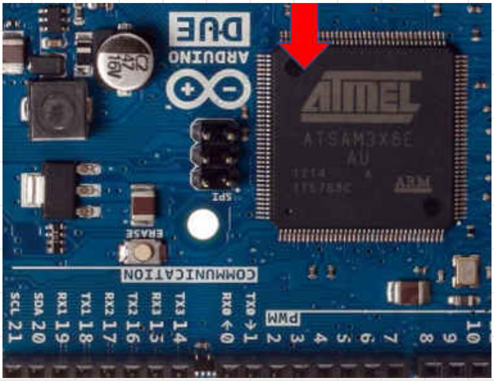
ARM (ACORN RISC MACHINE → ADVANCED RISC MACHINE)

- 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)

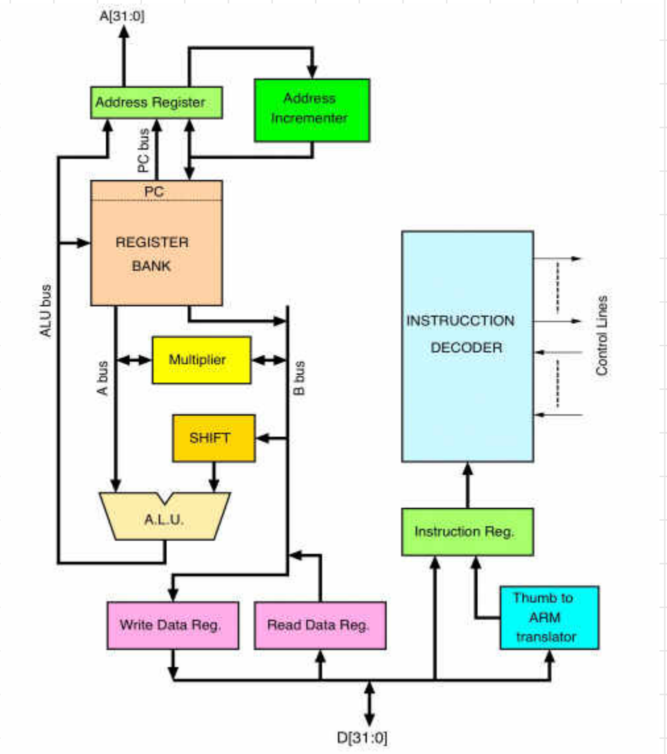


- ARM Architecture versions

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

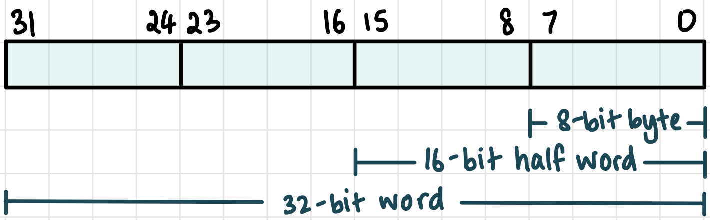


Data Sizes and Instruction Sets

- 32-bit architecture ; byte-addressible

- Byte: 8-bit
  Halfword: 16-bit ; aligned on 2-byte boundary
  Word: 32-bit; aligned on 4-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

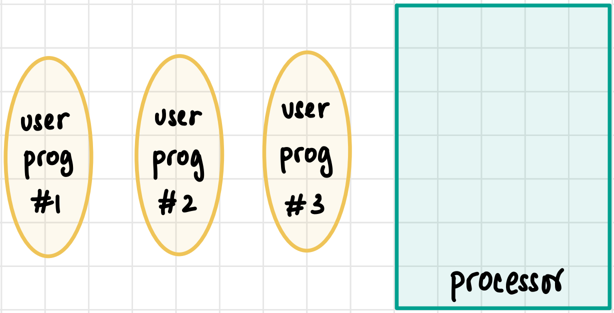
- Supervisor: entered on reset and when software interrupt instruction executed

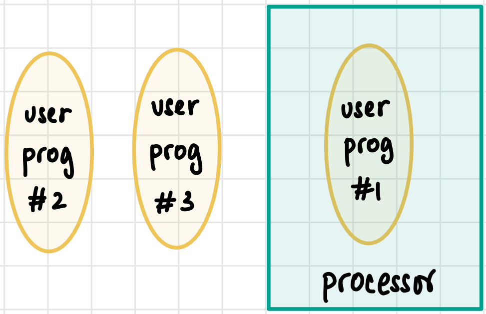
- Abort: memory access violations

- Undef: handle undefined instructions

- System: privileged mode using same registers as user modes

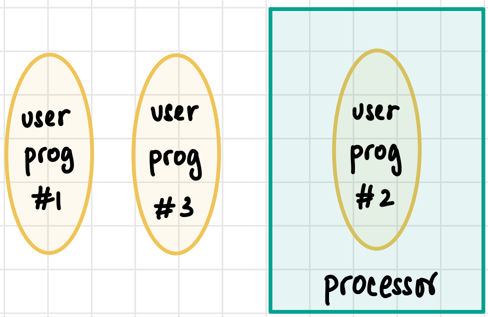
- interrupt service routines for each interrupt
- FIQ/IRQ pins on microprocessor
- eg: processor executing programs





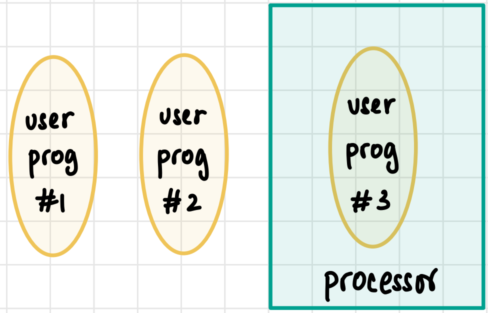
user prog #1 executing

user progs #2 and #3 waiting



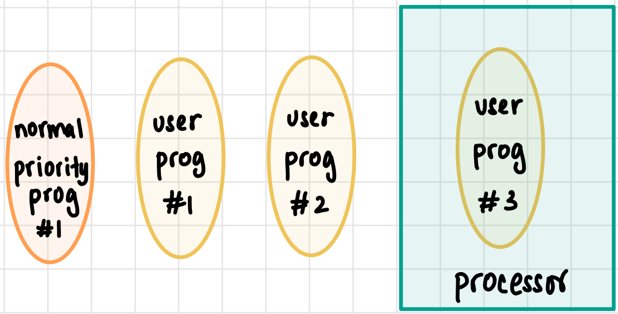
user prog #2 executing

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

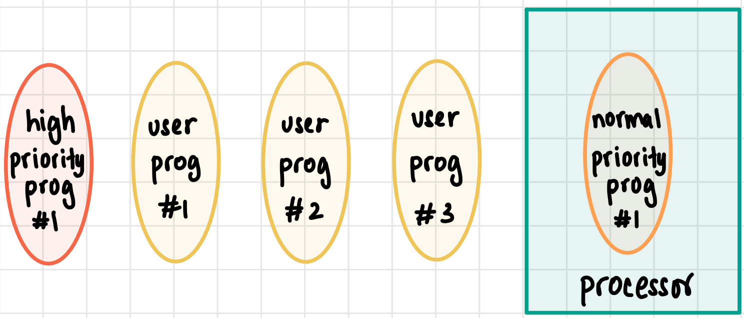


user prog #3 executing

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

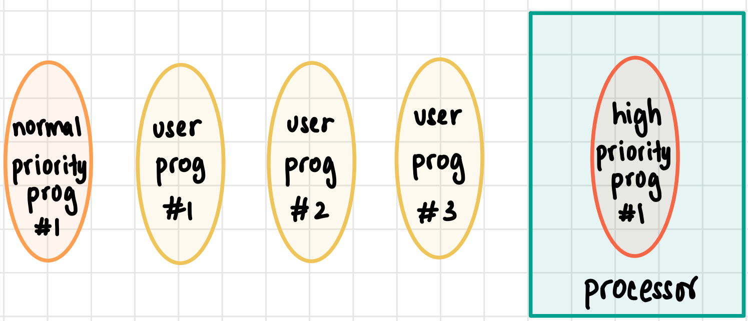


priority prog to be executed



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

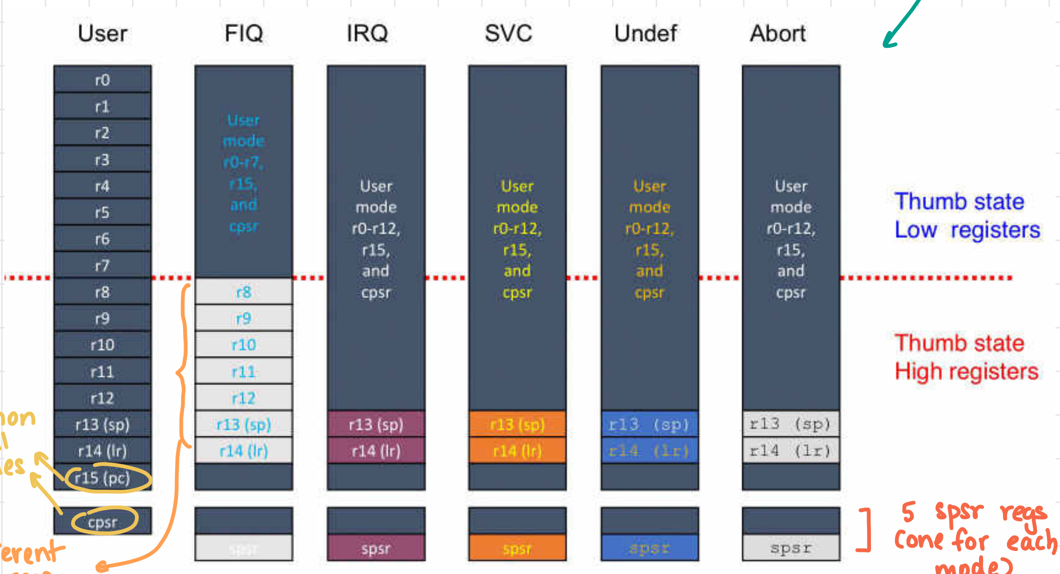
context switch: CPSR contents moved to dedicated SPSR

- 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



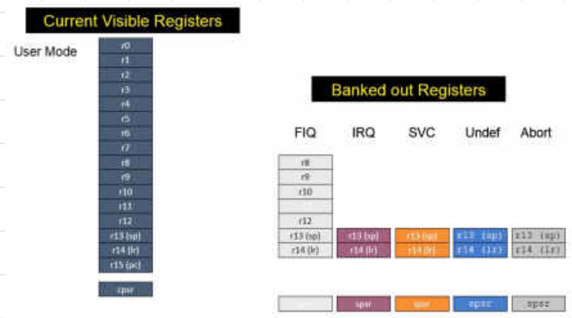
common to all modes

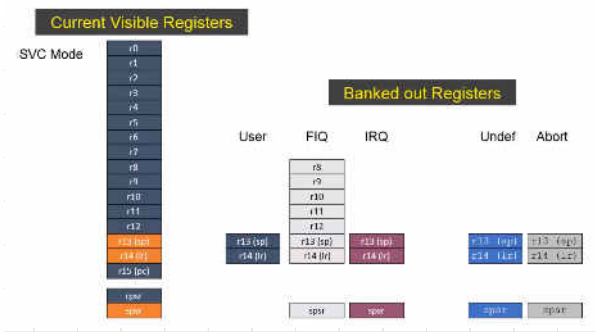
different set of regs (not user)

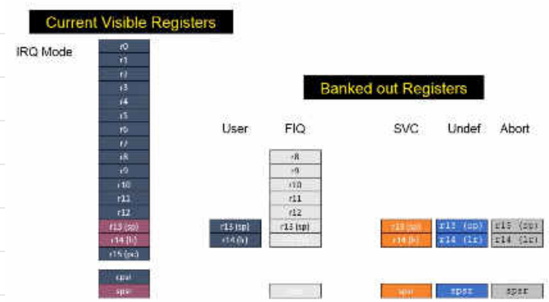
5 spsr regs (one for each mode)

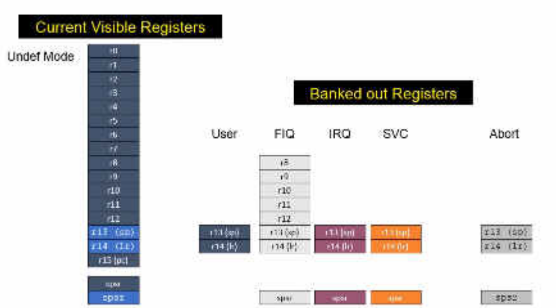
Note: System mode uses the User mode register set

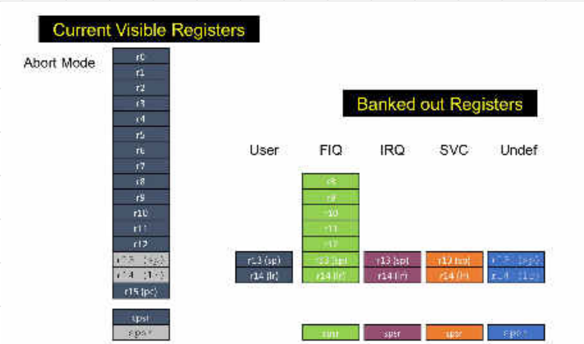
Visible and Banked Out Registers





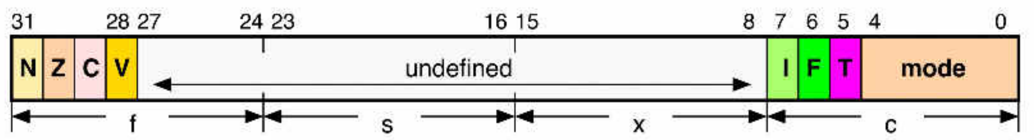






Status Register

- decision making using status register



- f: flag bits
  s: status bits } future enhancements; no use now
  x: extension bits
  c: control bits

- 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
	.text
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
	.data
Address of Data1	Declaration of variable 1
Address of Data2	Declaration of variable 2
....	....
....	....
Address of Data n	Declaration of variable n

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

```
p1.s

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

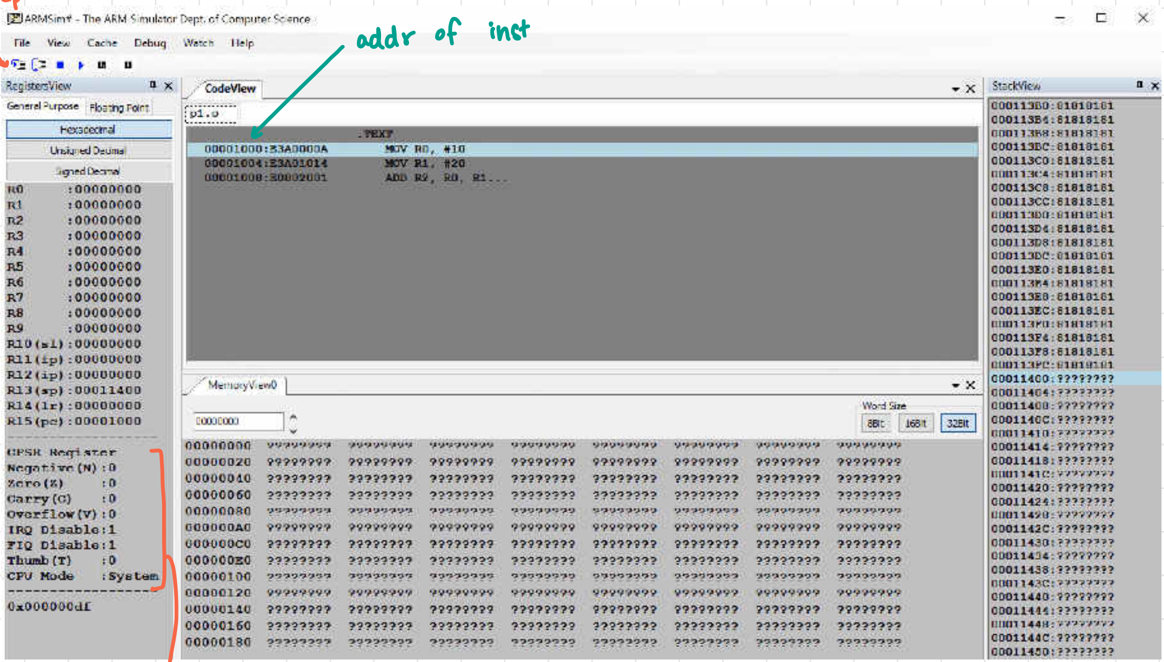
only .TEXT

ARM Simulator

step info

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

addr of inst



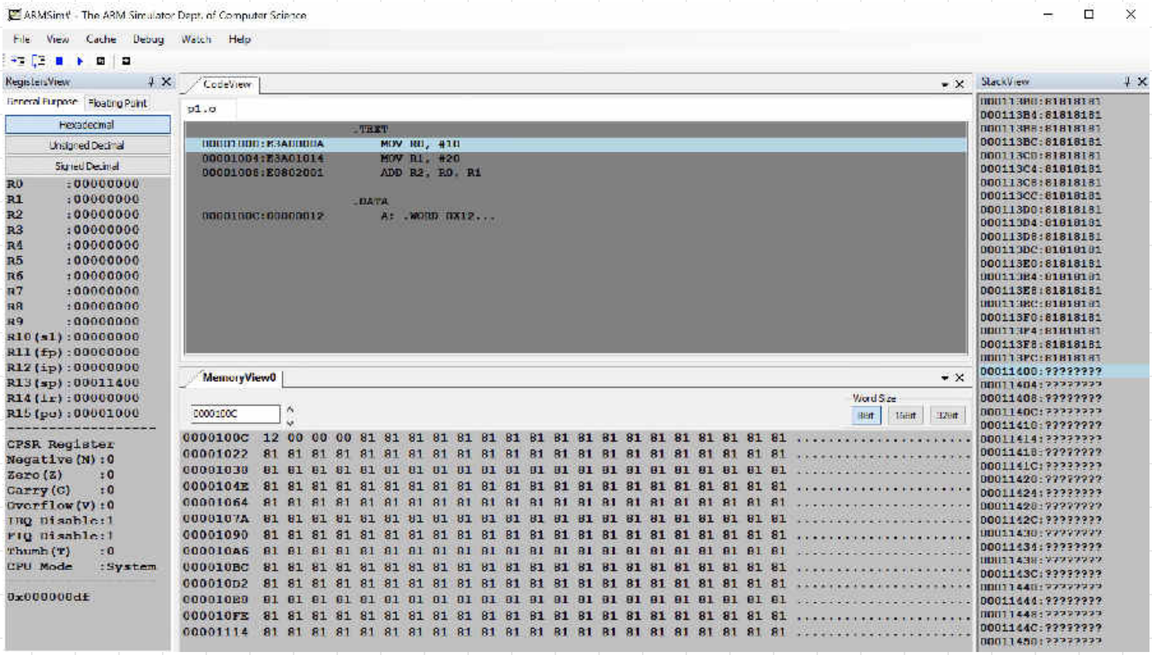
flags

Data & Text

p1.s

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

.DATA
    A: .WORD 0X12
```



↑ little endian

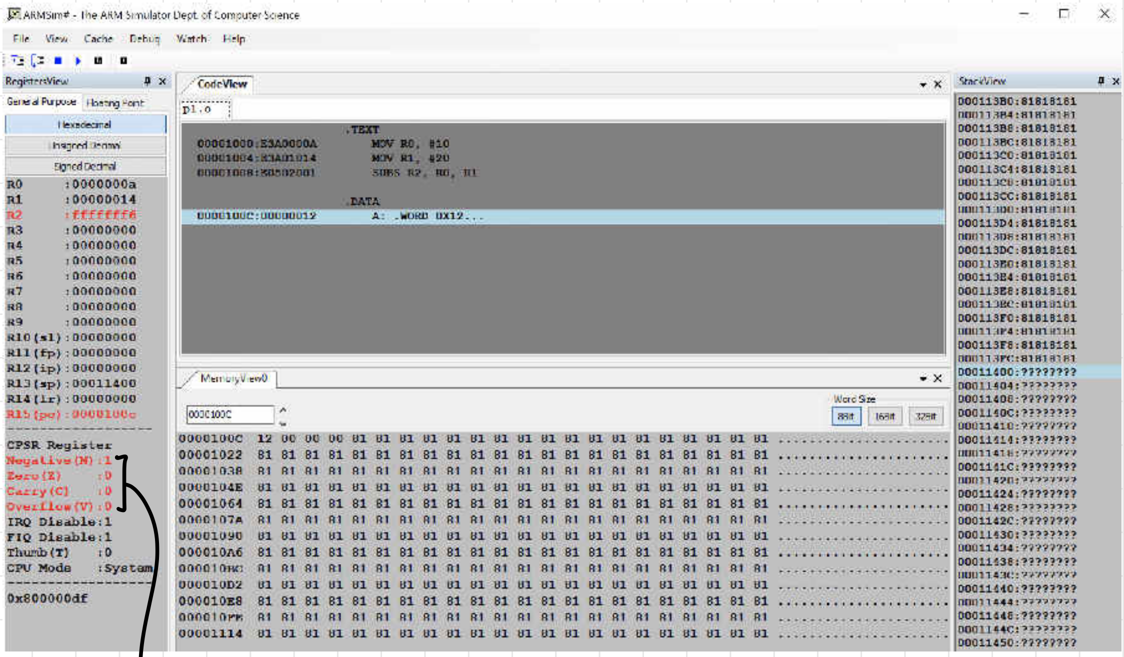
Status stored in CPSR - add S to end of Instruction

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
	.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, 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

syntax error if 2 imm or imm before reg

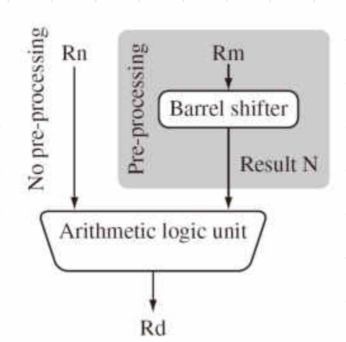
Operand2 - Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

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: arithmatic, 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

```
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
------------------
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
```

```
.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
```

Arithmatic 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

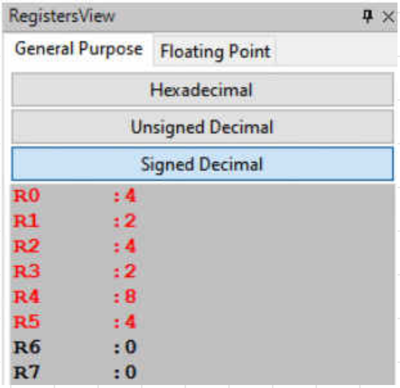
  Operand2 and Carry in are inverted and fed to adder or Invert Operand2 first and subtract carry later

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
```



Multiword Arithmatic

- 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

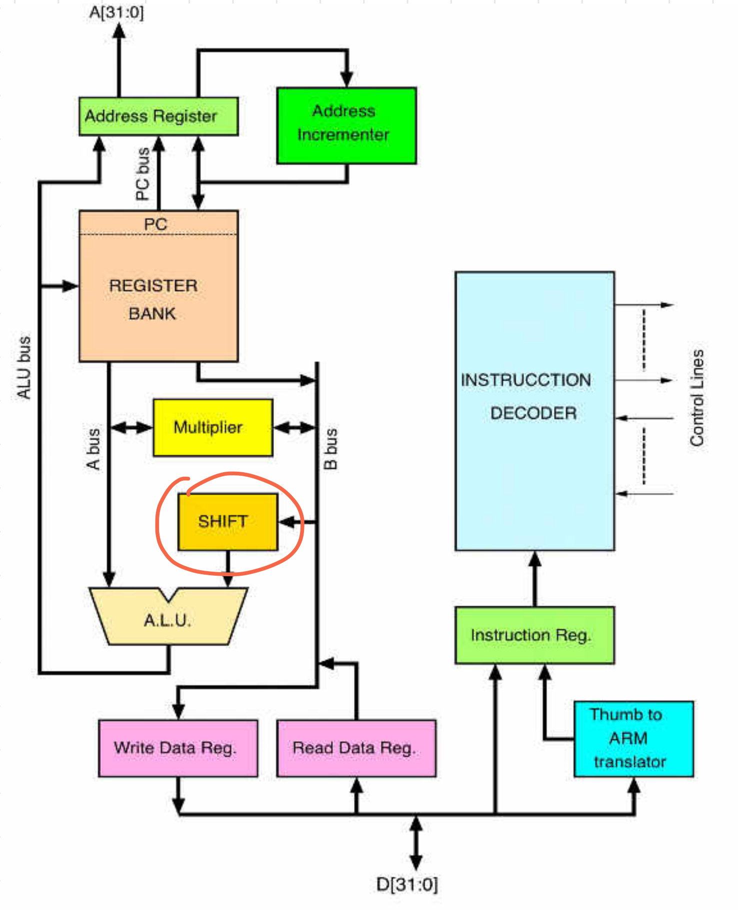
R5	R4

- subtract one 96-bit word from another

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

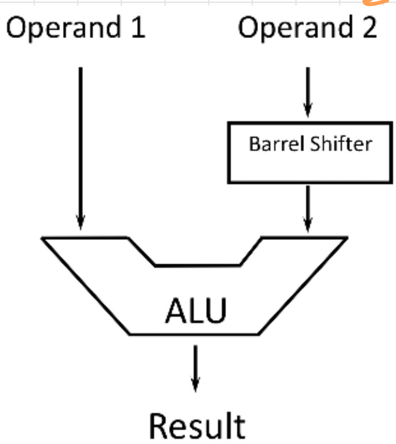
Barrel Shifter

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



Barrel Shift Operations

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

CPSR
stores carry

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

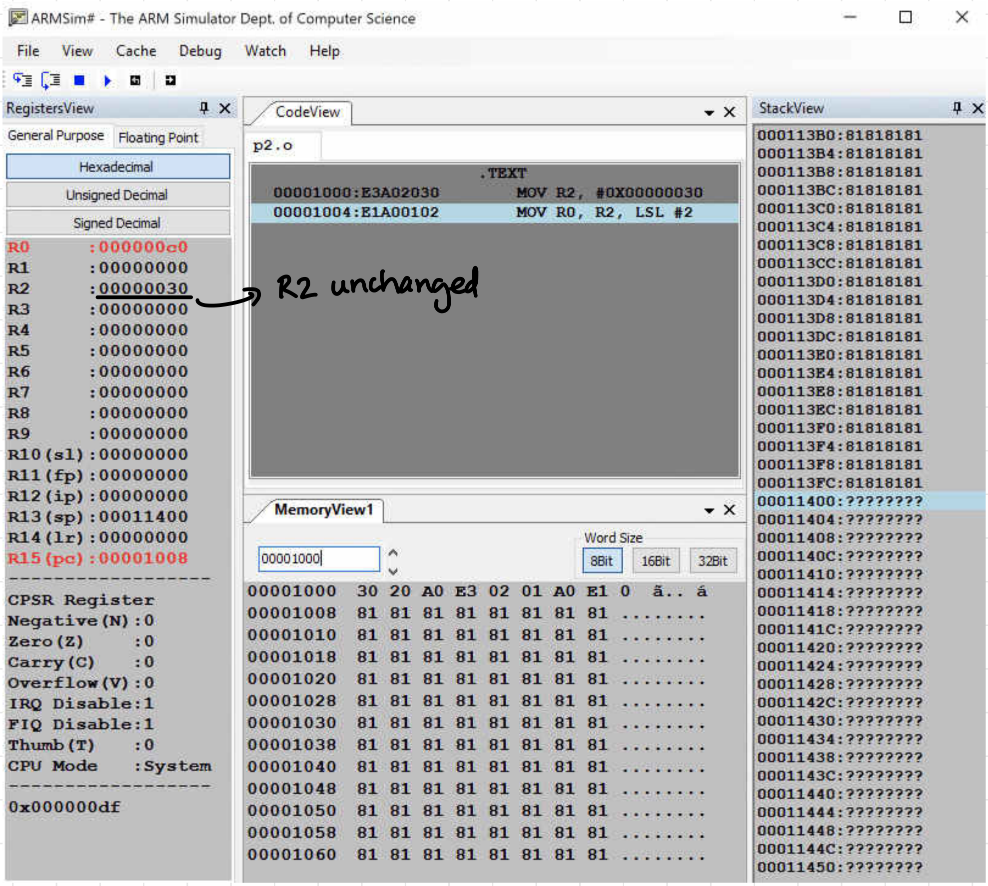
before

00000000 00000000 00000000 00110000

after

00000000 00000000 00000000 11000000

- multiplied by 2^n



Logical Shift Right

- LSR

Syntax

```
.TEXT
    MOV R2, #0XFFFFFFD0
    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

$3 + 256 \div 32$

$R3 = 11$

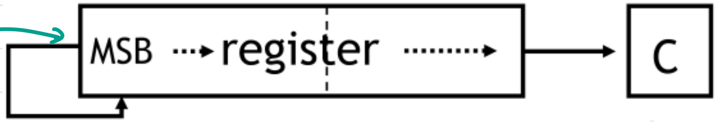
Before R1: 00000000 00000000 000000 1000 0001

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

Arithmatic Shift Right

- ASR

sign extended



Syntax

```
.text
    MOV R2, #0xffffffd0
    MOV R1, R2, ASR #2
```

before

11111111 11111111 11111111 11010000

after

11111111 11111111 11111111 11110100

Rotate Right

- ROR

LSB to MSB

Syntax

```
MOV R2, #0xffffffd5
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

{S} not needed as CPSR reflects changes anyway

```
<Operation> {<cond>} Rn, Operand2
```

CPSR reflects

compare

compare negation

test

test equality

```
CMP R1, R2     @ set cc on R1-R2
CMN R1, R2     @ set cc on R1+R2
TST R1, R2     @ set cc on R1 and R2
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(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
```

```
R0       :00000100
R1       :00000019
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

256 = 00000100
25 = 00000019
-25 = ffffffe7

$$
\begin{array}{rr}
256 & 00000100 \\
-\ 25 & ffffffe7 \\
\hline
 & 1\ 000000e7
\end{array}
$$

000000e7 → 231

carry

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       :fffffffb
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       :fffffffb
R1       :fffffffb
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 (~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
```

complements

```
R0        :00000005
R1        :00000006
R2        :00000004
R3        :00000007
R4        :00000003
R5        :fffffffa
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)   :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:
```

Conditional Branch

```
MOV R0, #0

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

backward branching

CPSR

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
```

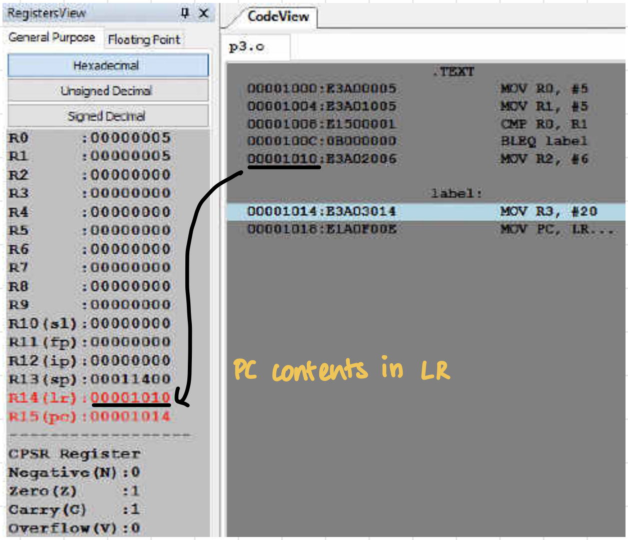
satisfied

```
R0      :00000005
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) :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
```



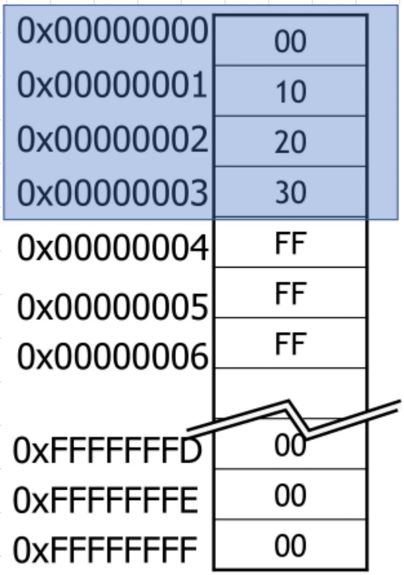
Data Transfer Instructions

- Memory to register
- Register to memory

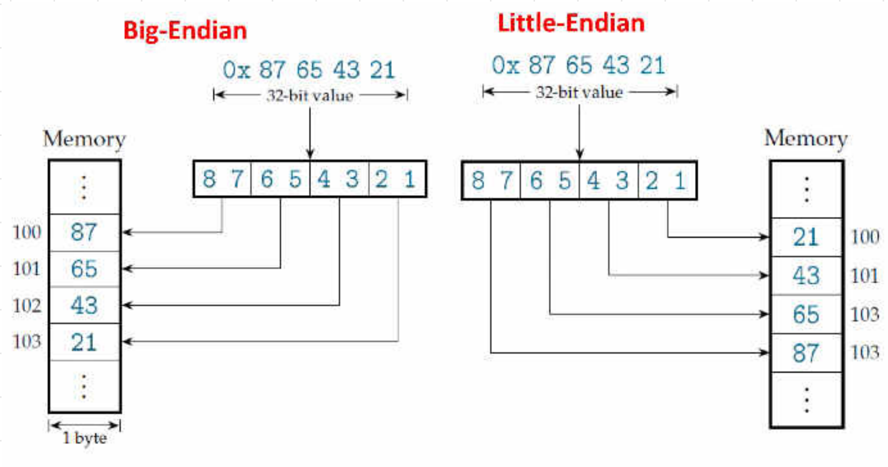
- Load-store architecture

- Memory: array of 0 to $2^{32}-1$

- word, half-word, byte



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

destination reg

Syntax

<LDR/STR>{<cond>}{B} Rd, Addressing

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

signed byte   half word   signed HW

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

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 ;     @ R0 = 0x1C - address of A
LDR  R5, [R0] ;   @ Copy data from R0 into R5 (40)

LDR  R3, =B;      @ R3 = 0x14 - address of B
STR  R5, [R3];    @ 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)

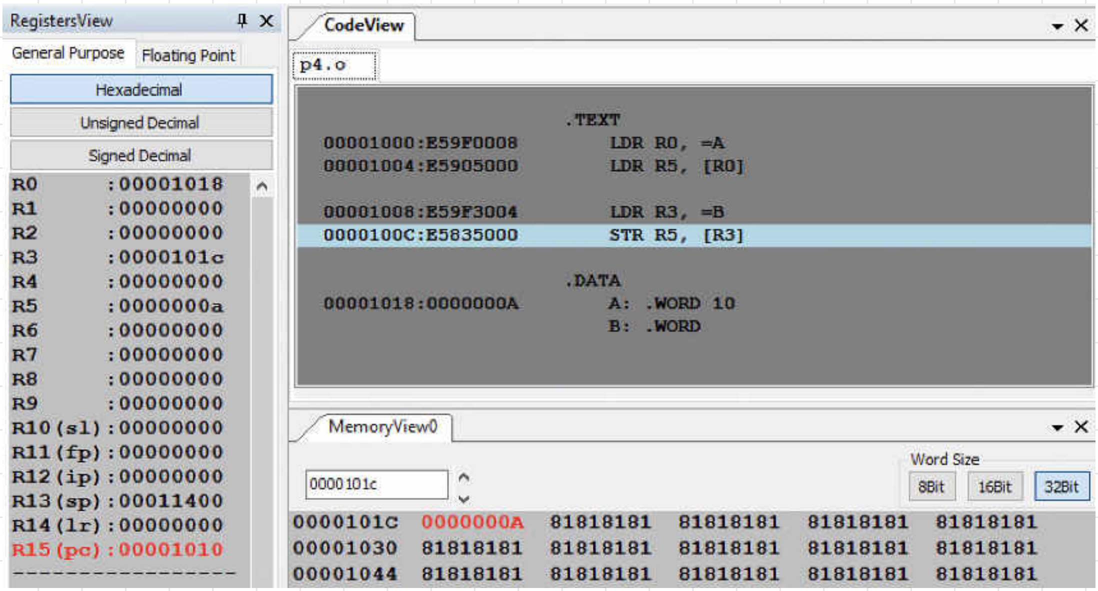
- ```
  int *R0 = &A;     // similar to this
  int R5 = *R0;     // but not high level
  ```
  ```

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

    LDR R3, =B
    STR R5, [R3]

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

Executed in ARMSim



Example 2

B = A+9

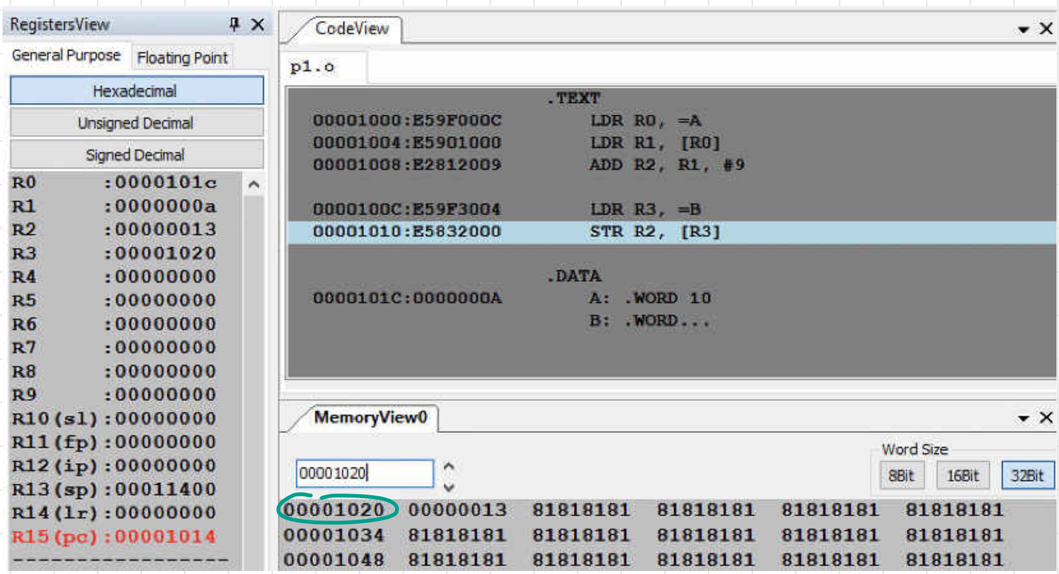
```
.TEXT
    LDR R0, =A
    LDR R1, [R0]
    ADD R2, R1, #9

    LDR R3, =B
    STR R2, [R3]

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

operator only supported by LDR/STR instructions not ADD

adds 9 to 0xA

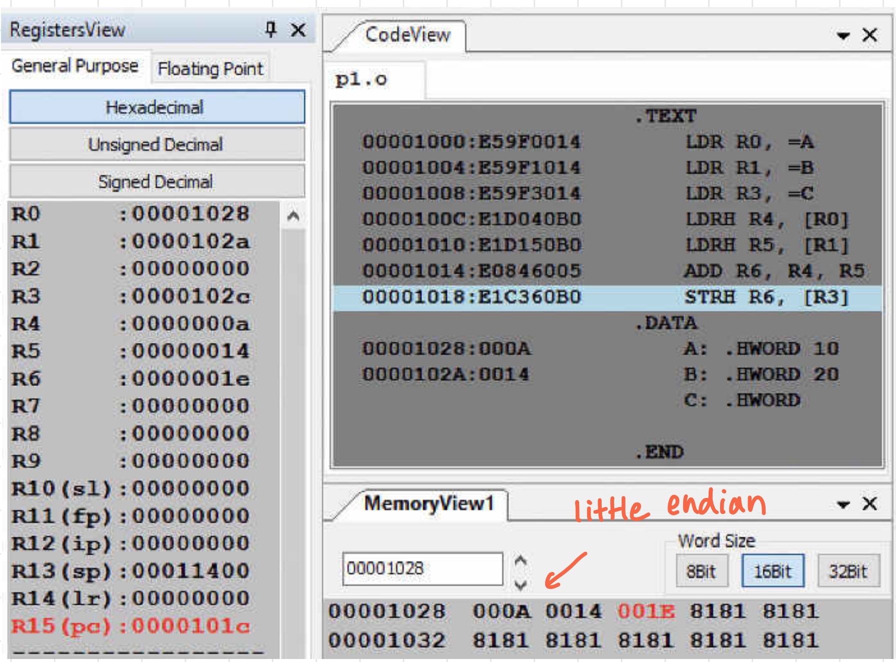


Example 2

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
.END
```

half word

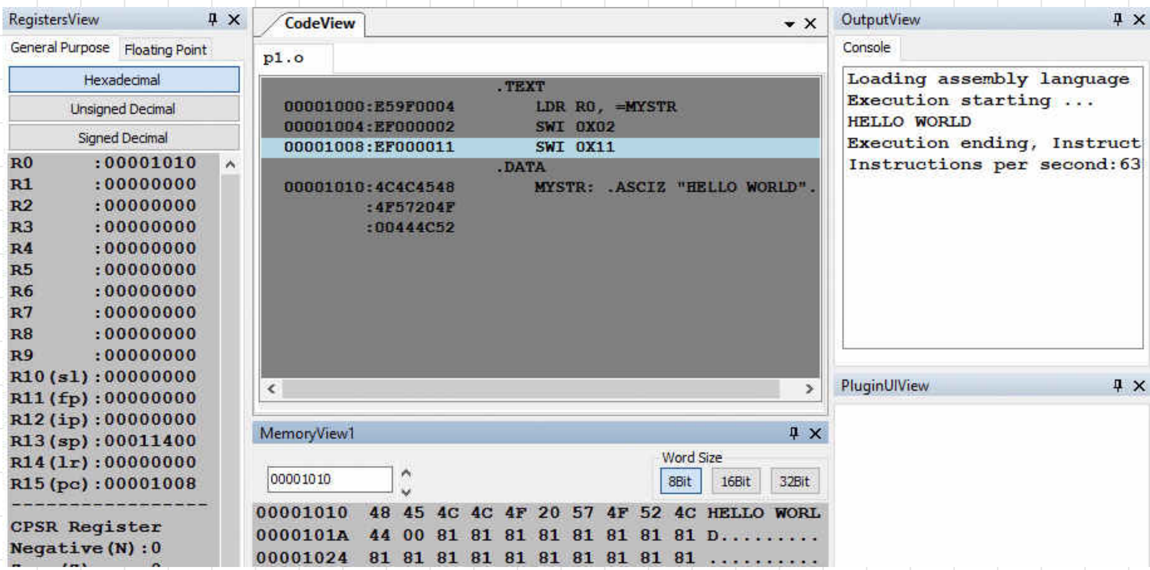


(see 8-bit)

String

- Datatype: ASCIZ " " (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"
```



ADDRESSING OR INDEXING

- Arrays

1. Pre-Indexing Without Writeback

mem loc

first increment pc, then store incremented

```
LDR Rd, [Rn, OFFSET]
```

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]
```

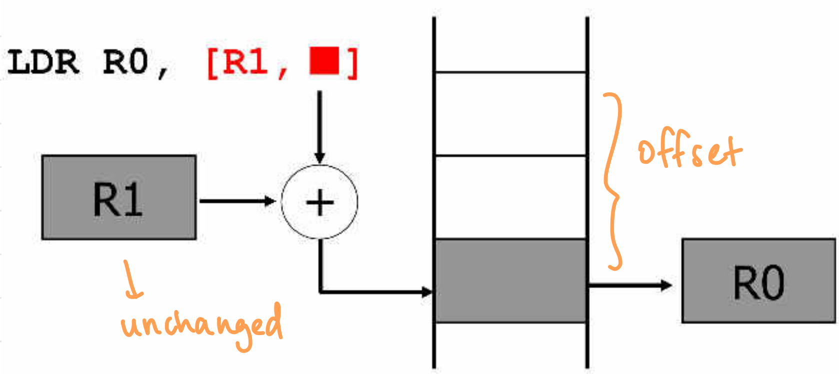
- Scaled Register type

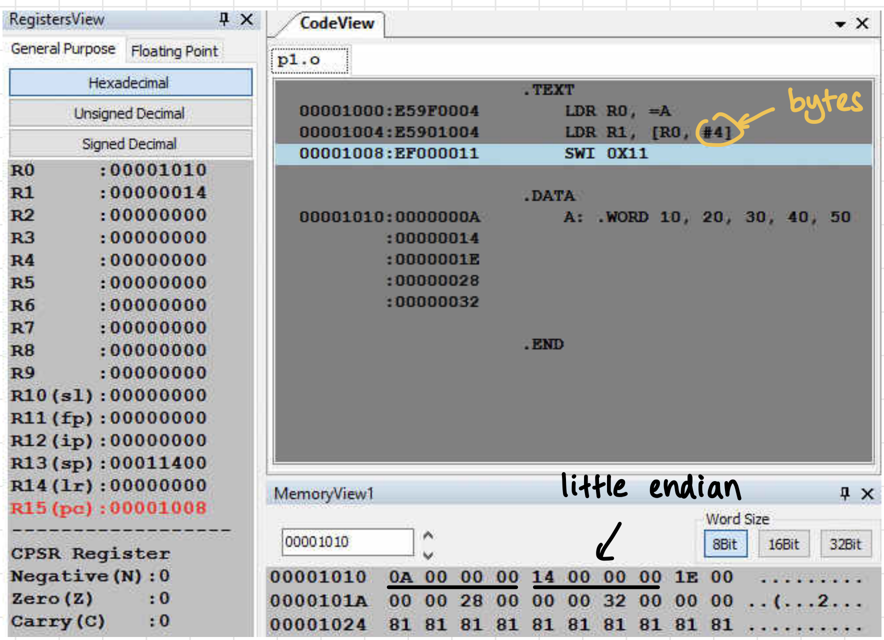
has to be reg for scaled

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

PRE-INDEXING WITHOUT WRITEBACK

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





#4: 4 bytes

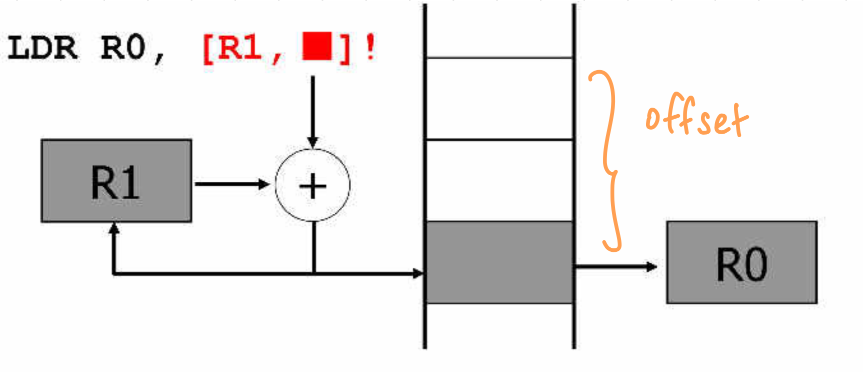
(#2: useless value)

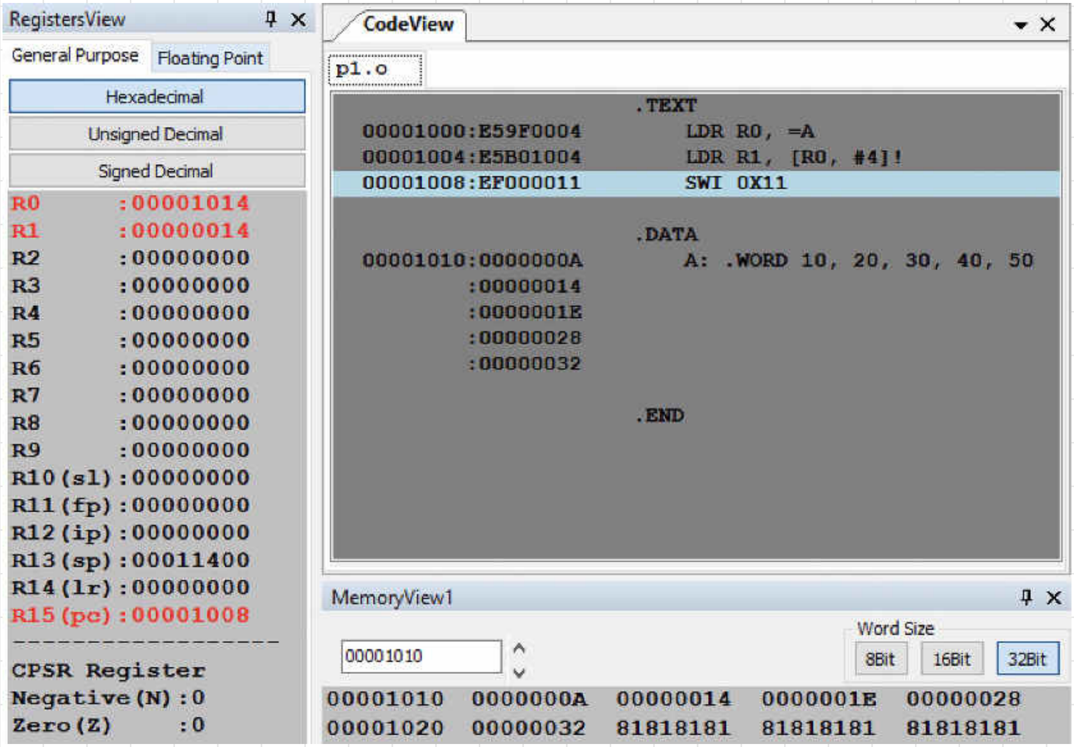
AUTO INDEXING

PRE-INDEXING WITH WRITEBACK

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

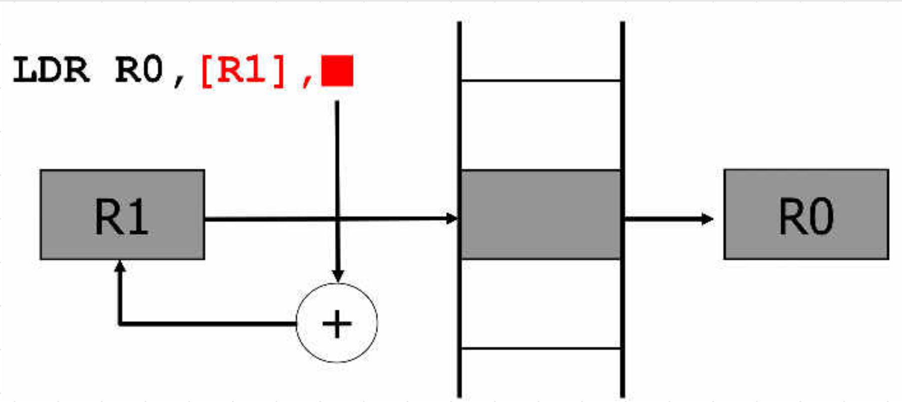
(faster)

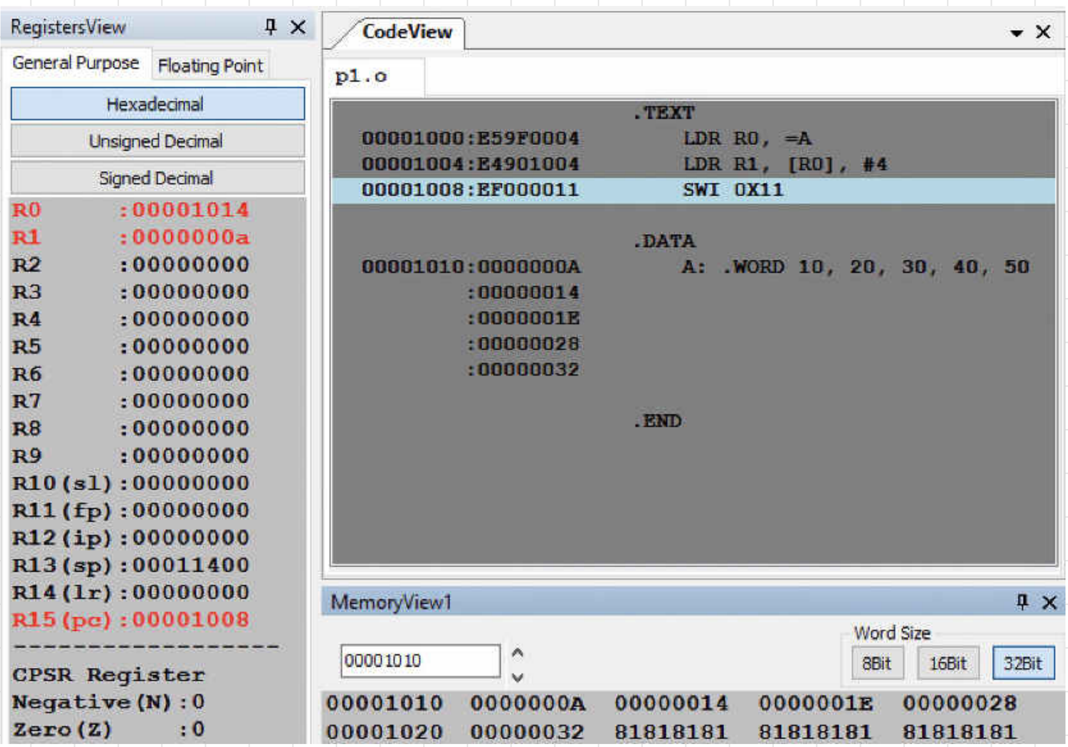




POST INDEXING

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





BLOCK TRANSFER

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

A

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

```
.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

direction of transfer is different

Syntax

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

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

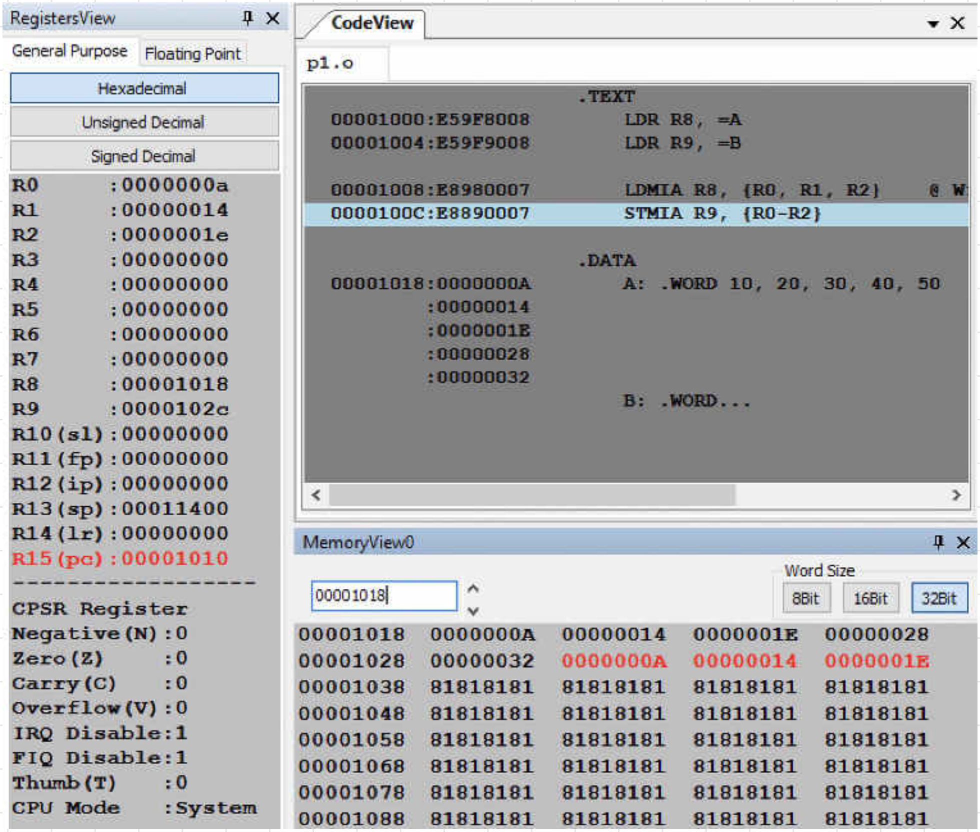
```
.TEXT
    LDR R8, =A
    LDR R9, =B

    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 (sl)	:00000000
R11 (fp)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:0000100c

Store at Once



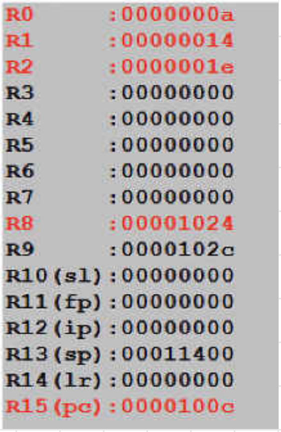
With Writeback

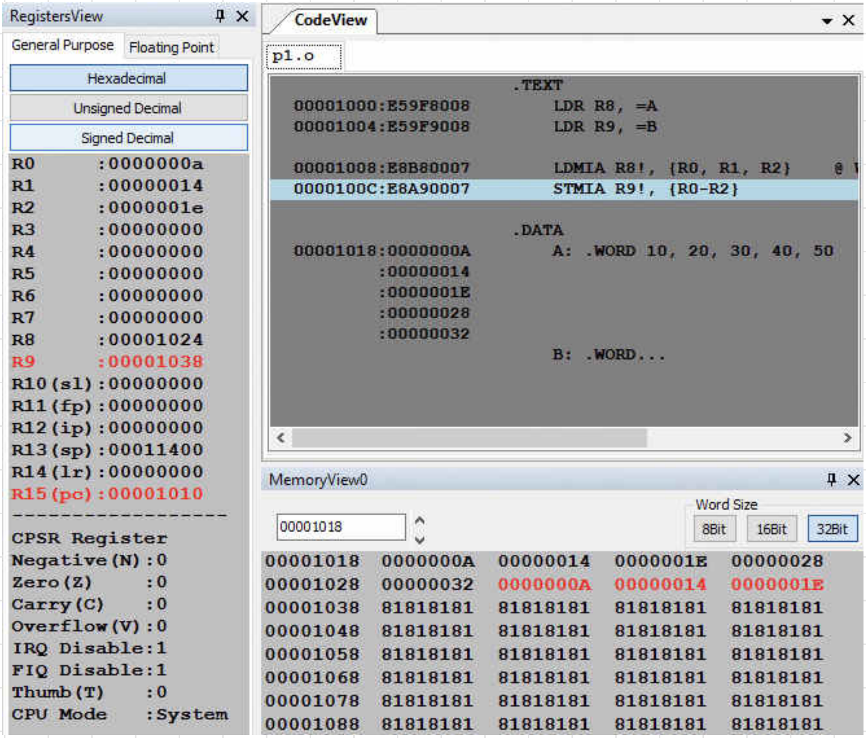
```
.TEXT
    LDR R8, =A
    LDR R9, =B

    LDMIA R8!, {R0, R1, R2}
    STMIA R9!, {R0-R2}

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

R8: 1024 (wb)





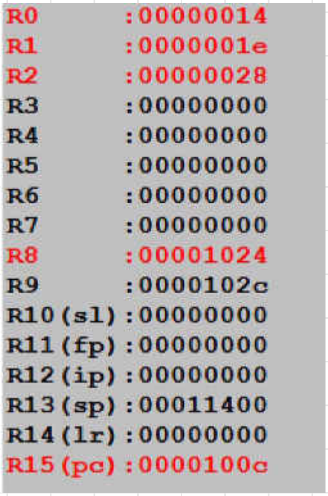
LDMIB

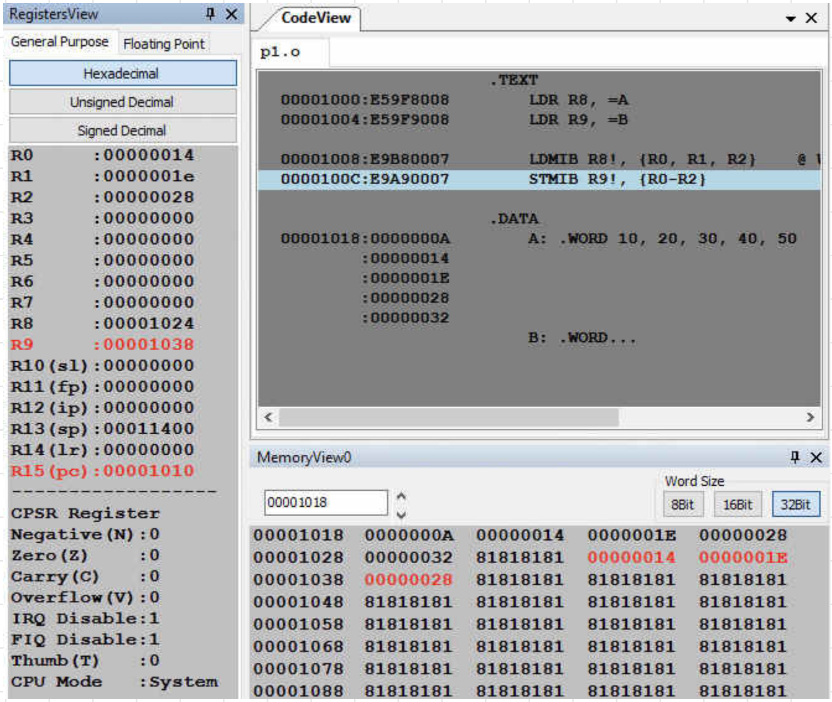
- increment before store and load

```
.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
```





LDMDA

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

```
.TEXT
    LDR R8, =A
    LDR R9, =B

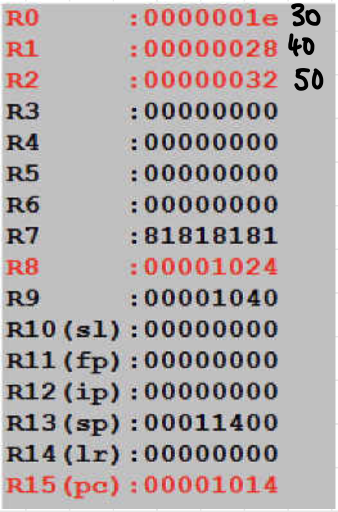
    LDR R6, [R8], #16
    LDR R7, [R9], #12

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

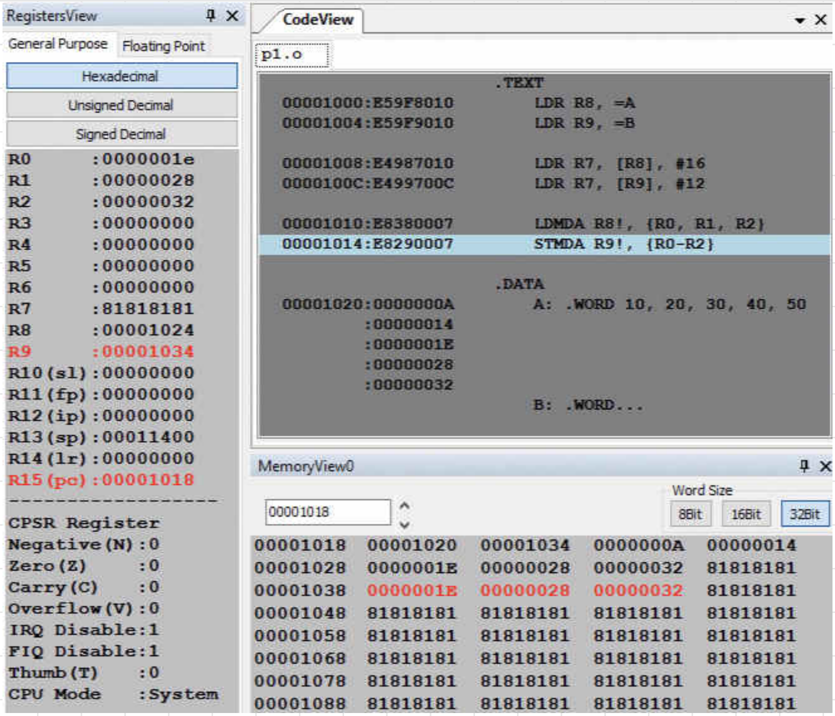
.DATA
    A: .WORD 10, 20, 30, 40, 50
    B: .WORD
```

50

higher addr: higher reg



30
40
50



LDMDB

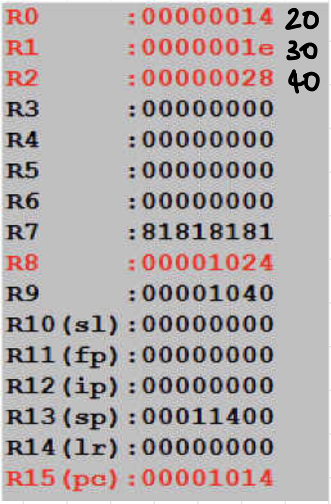
- decrement before
- word: #4 (4 bytes)

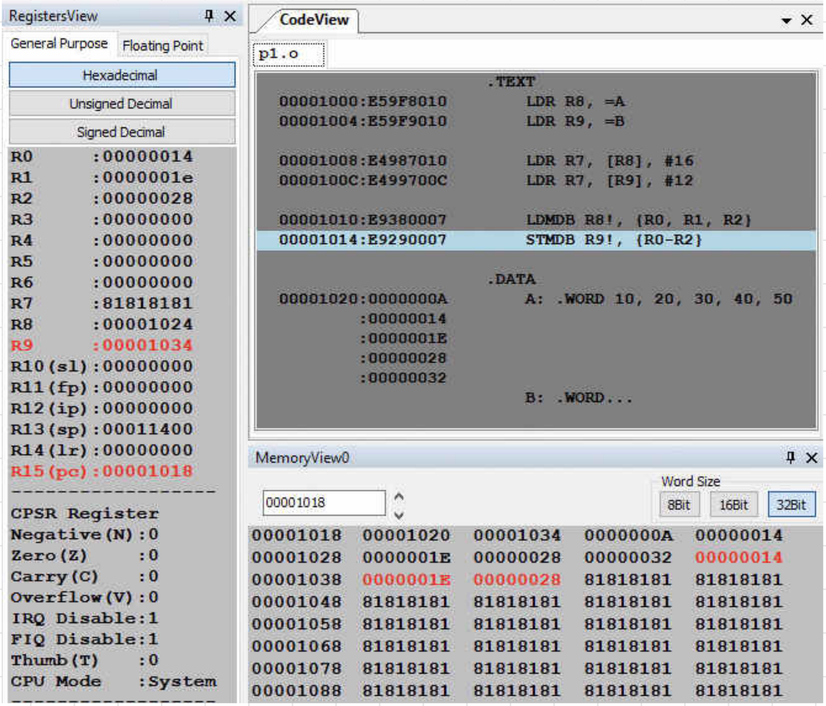
```
.TEXT
    LDR R8, =A
    LDR R9, =B

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

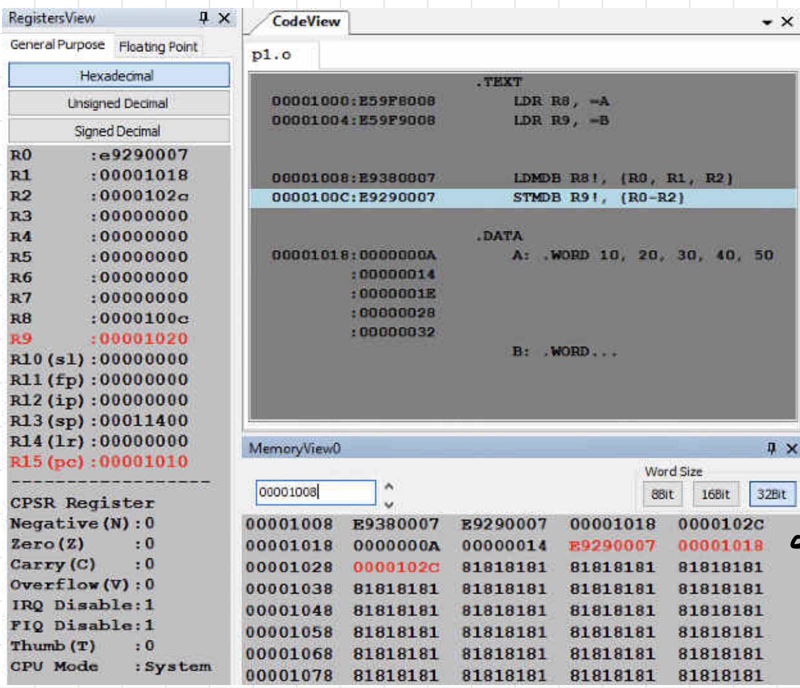
    LDMDB R8!, {R0, R1, R2}
    STMDB R9!, {R0-R2}

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





Without Moving R8, R9



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

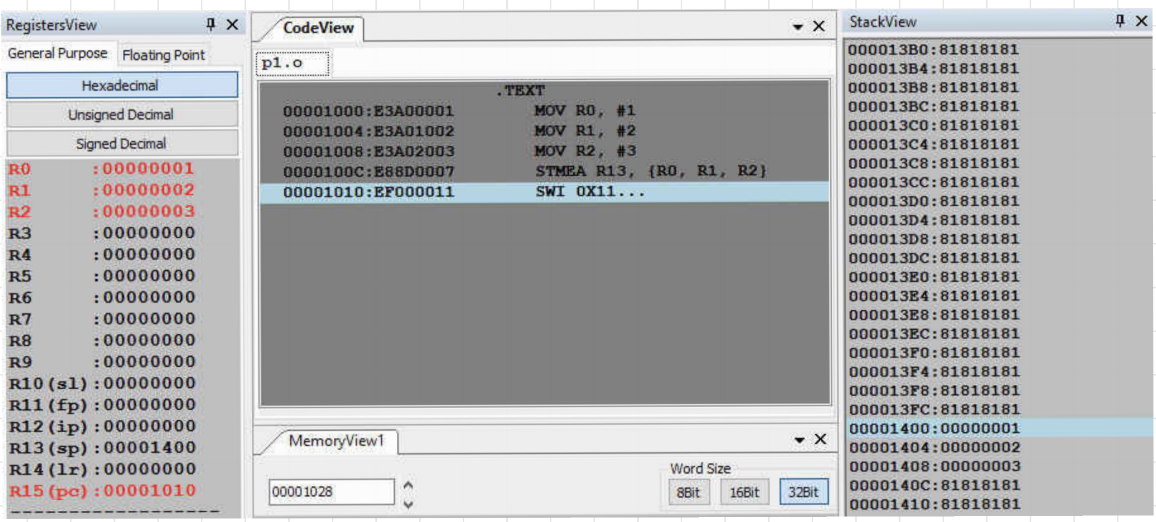
either works (functionally equivalent)

(LDM / STM) EA

Example 1

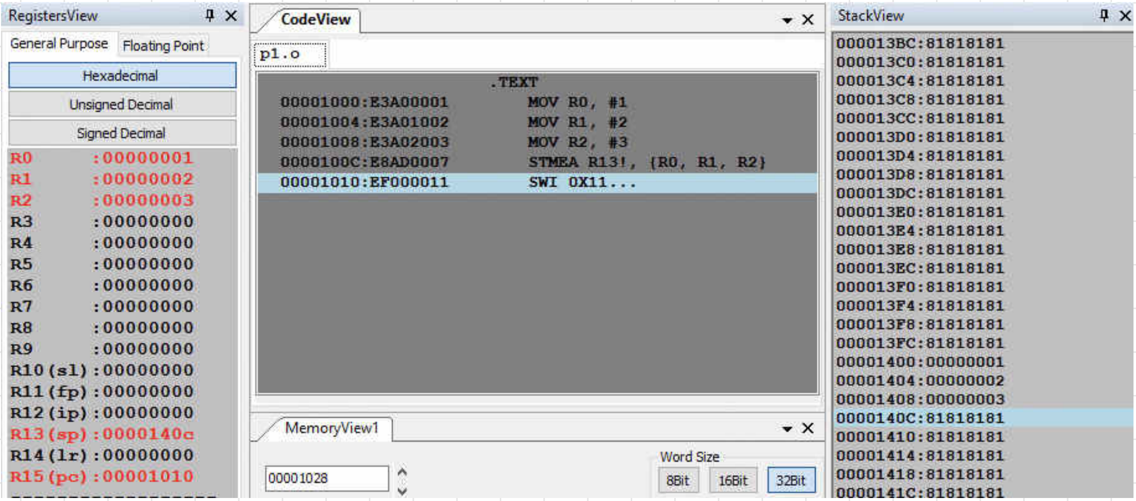
push, inc

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



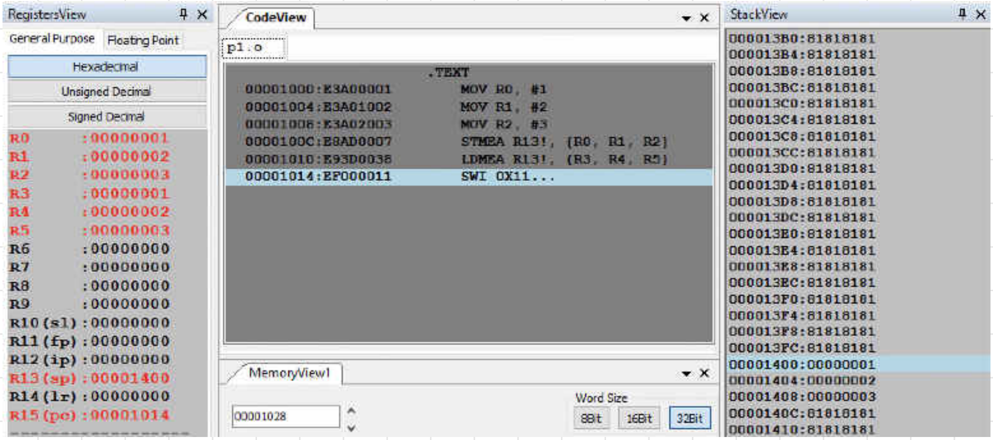
Example 2

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



Example 3

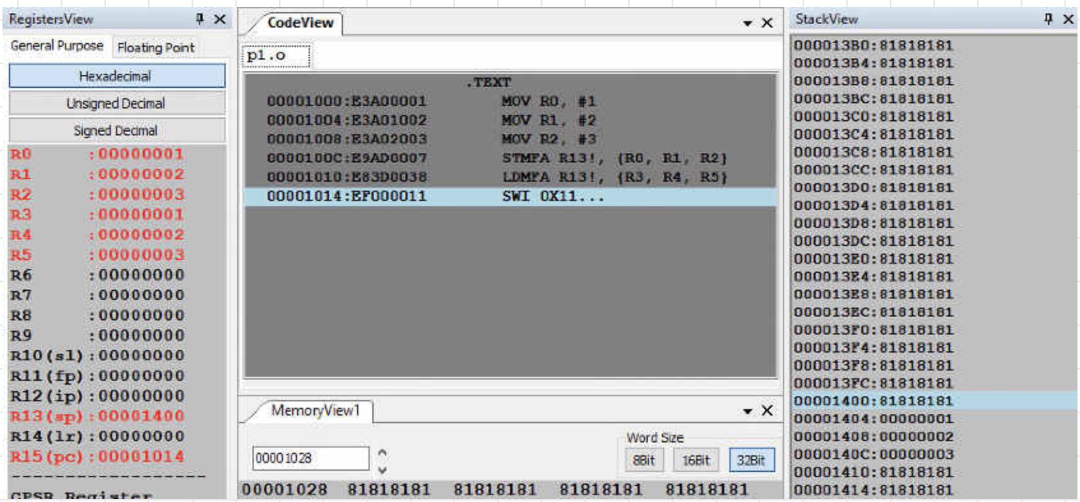
```
.TEXT
    MOV R0, #1
    MOV R1, #2
    MOV R2, #3
    STMEA R13!, {R0, R1, R2}
    LDMEA R13!, {R3, R4, R5}
    SWI 0X11
```



(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
```

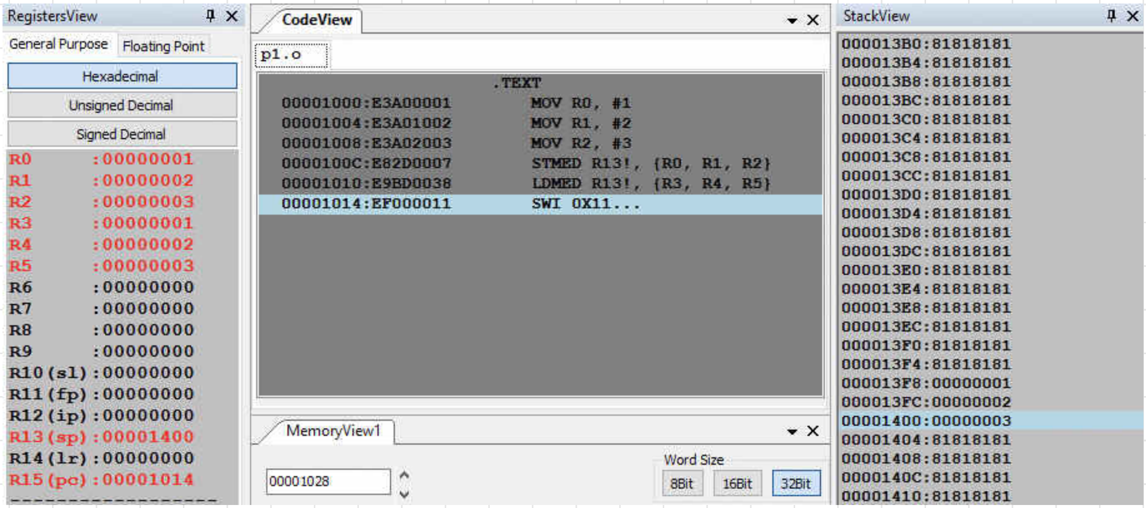
inc, push (IB)
pop, dec (DA)



(LDM | STM) ED

Example 1

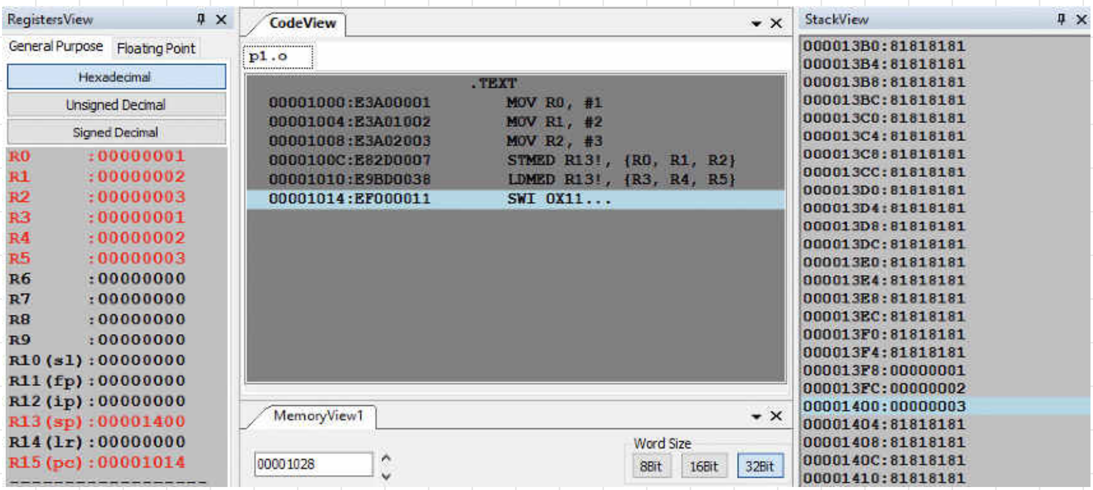
```
.TEXT
    MOV R0, #1
    MOV R1, #2
    MOV R2, #3
    STMED R13!, {R0, R1, R2}
    LDMED R13!, {R3, R4, R5}
    SWI 0X11
```



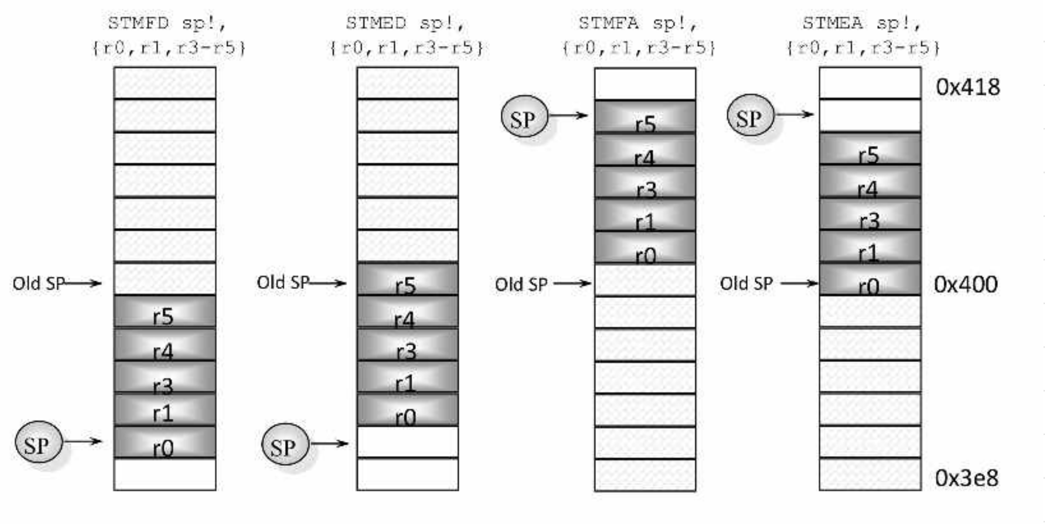
(LDM | STM) FD

- 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

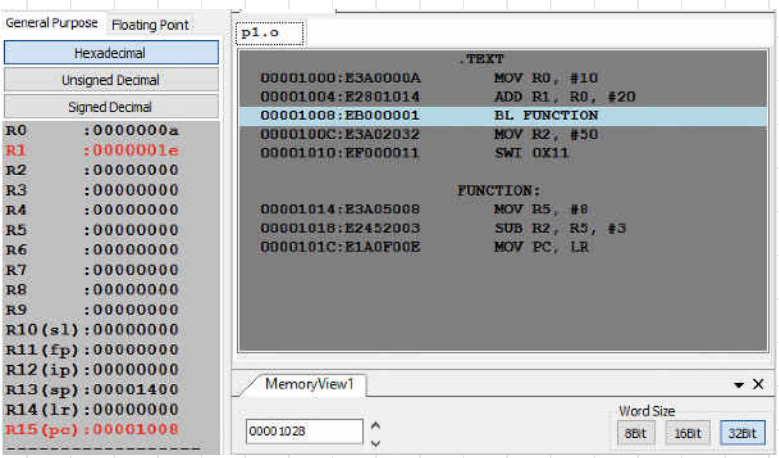
Called Procedure	
0x0020	Procedure: Instruction 1
0x0024	Instruction 2
0x0028	Instruction 3
0xxxxx	MOV PC LR or BX LR

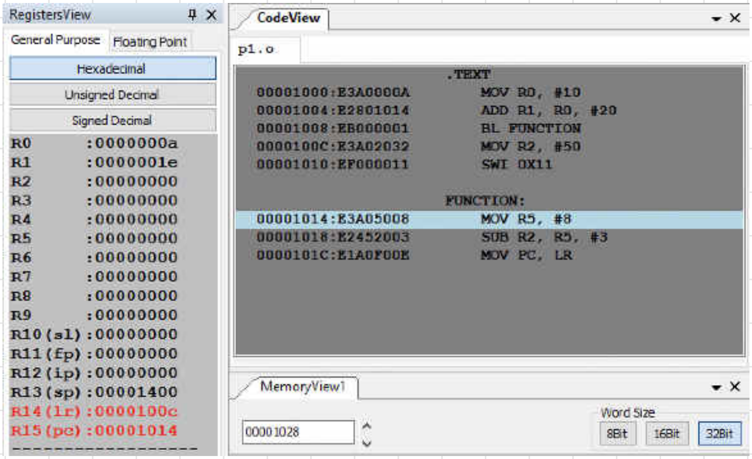
Example 1

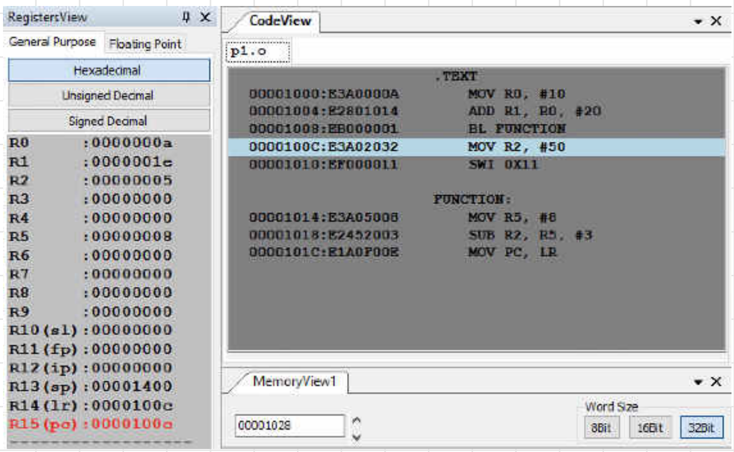
```
.TEXT
    MOV R0, #10
    ADD R1, R0, #20
    BL FUNCTION
    MOV R2, #50
    SWI 0X11

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

unparameterised







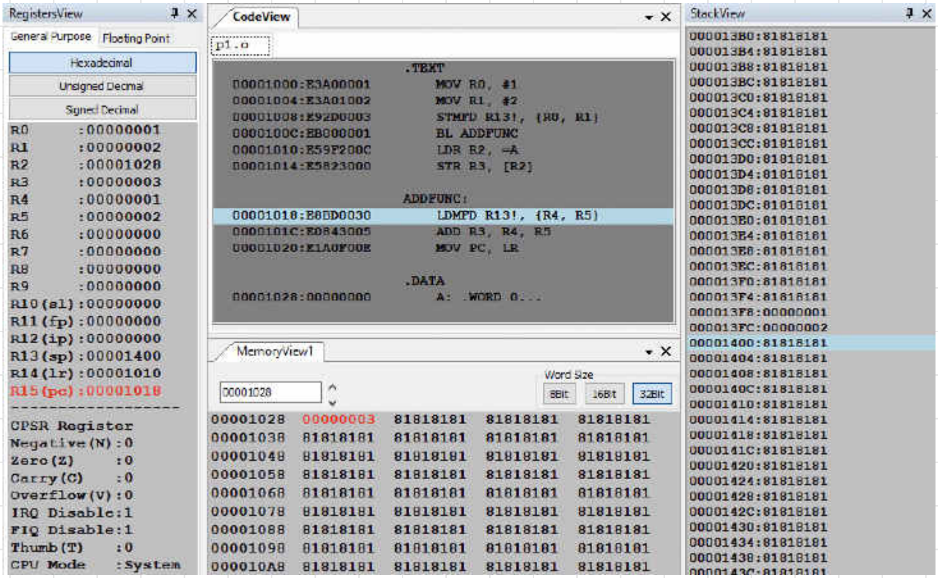
Example 2

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}

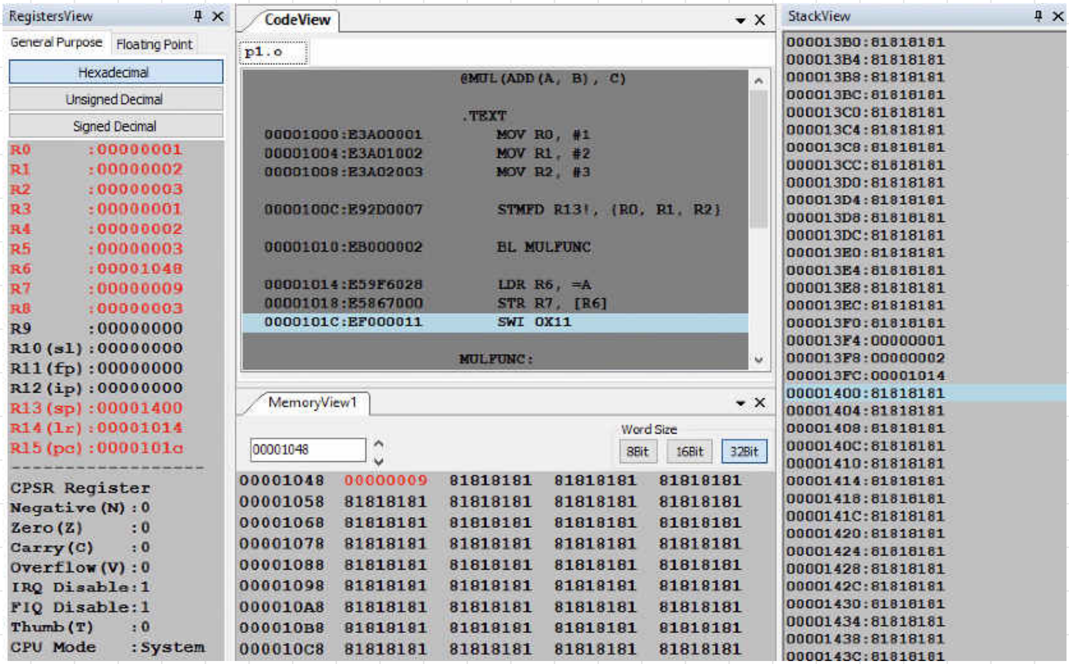
    BL MULFUNC

    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
```



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)

2 registers

MUL

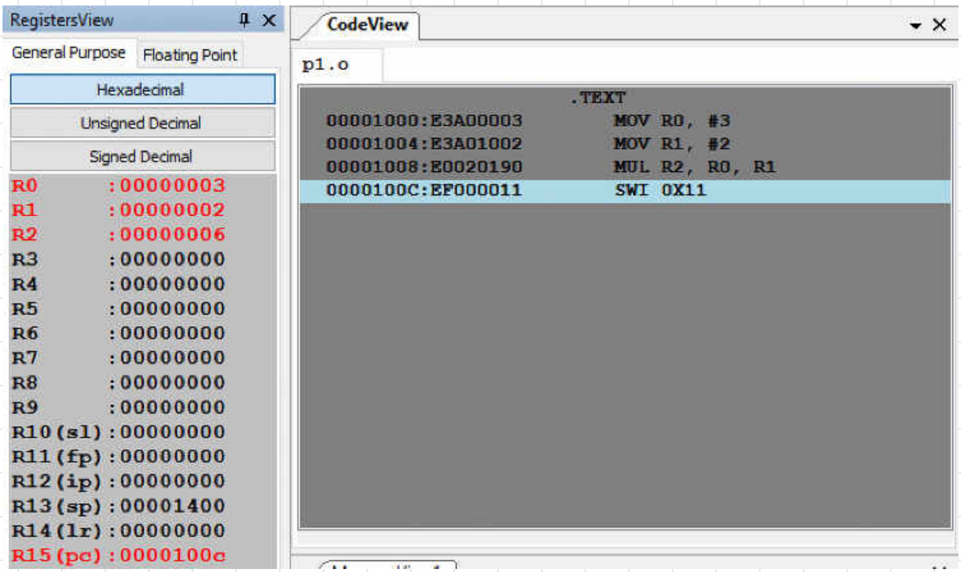
```
MUL{<cond>}{S} Rd, Rf, Rs    @ Rd = (Rf * Rs)[31:0]
```

must be different registers

reg only, not immediate

Example 1

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



Example 2

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

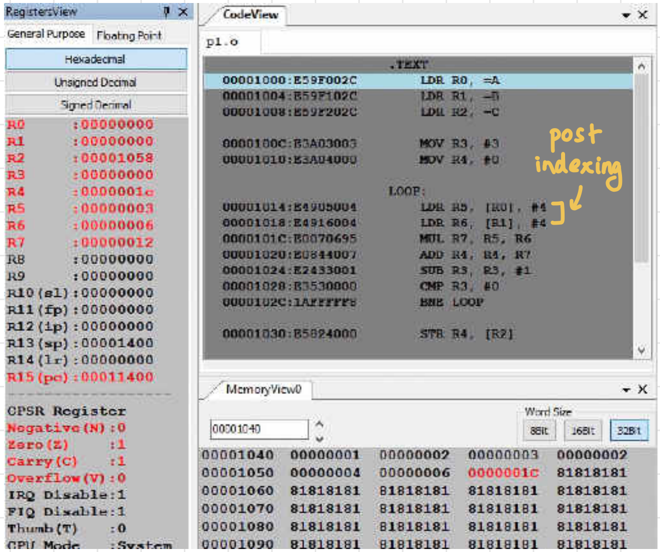
```
.TEXT
    LDR R0, =A
    LDR R1, =B
    LDR R2, =C

    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
```



MLA

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

Example 1

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

replaces MUL & ADD

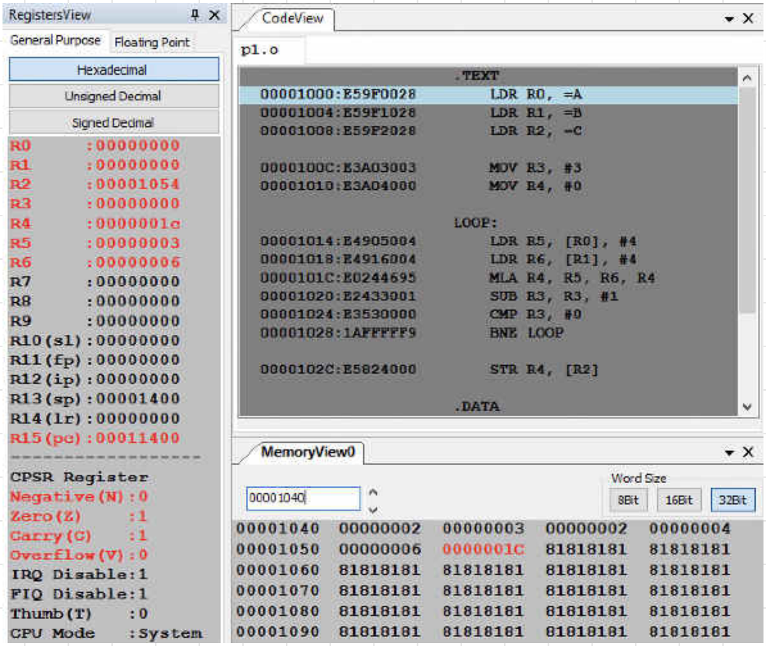
```
.TEXT
    LDR R0, =A
    LDR R1, =B
    LDR R2, =C

    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

    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

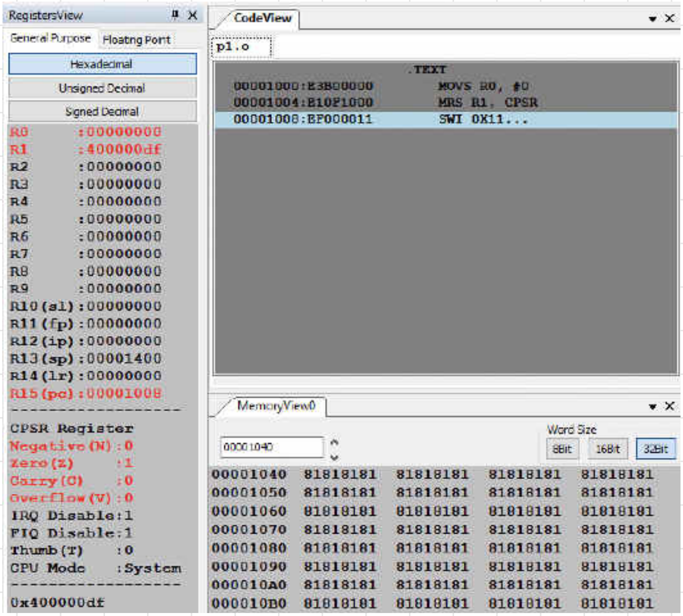
MRS

move to reg from status register (read)

```
MRS R0, CPSR
MRS R1, SPSR
```

Example 1

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



MSR

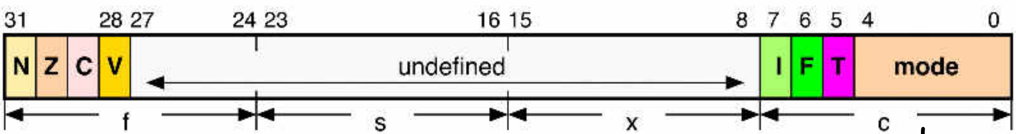
move to status register from reg (write)

Example:

```
MSR CPSR_field, R0
MSR SPSR_field, R1
```

_field

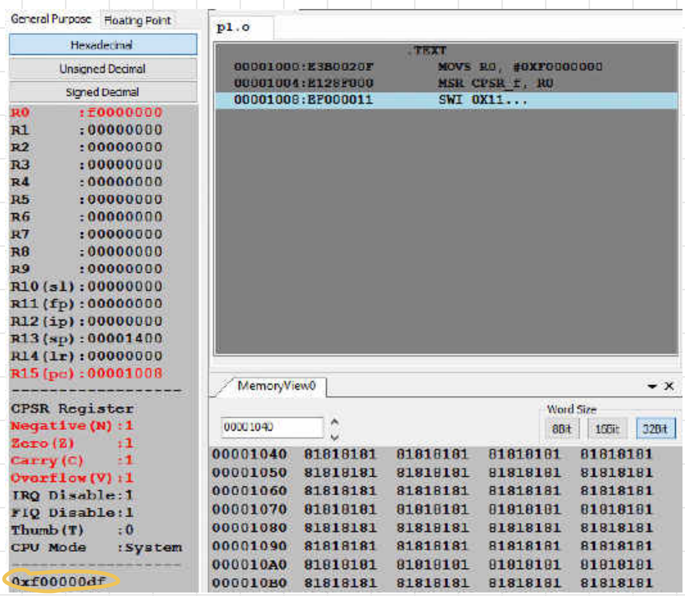
_c: Control Field (0:7)
_f: Flag Field(24:31)
_x: Extension (8:15)
_s:Status (16:23)



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

Example 1

```
.TEXT
    MOVS R0, #0XF0000000
    MSR CPSR_f, R0
    SWI 0X11
```



0xF = 1111

only first 8 bits

SWP

swap memory and register value

Syntax

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

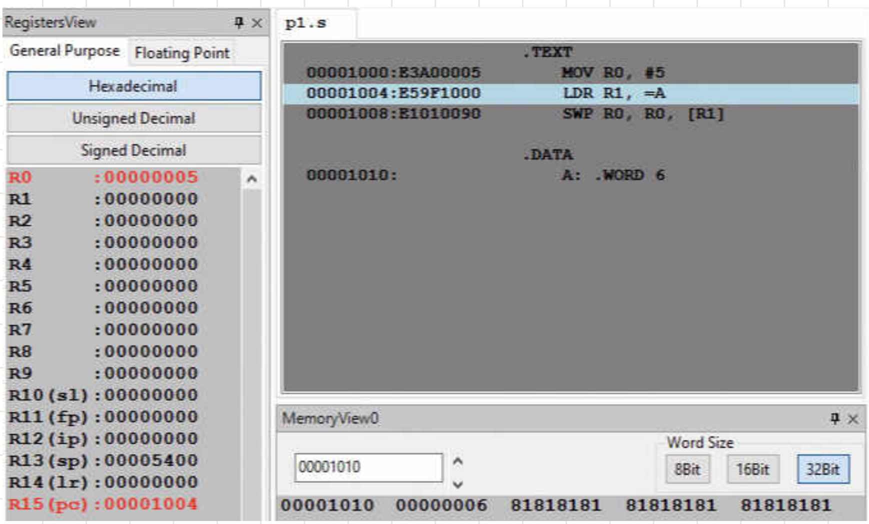
deprecated in ARMv6 and v7

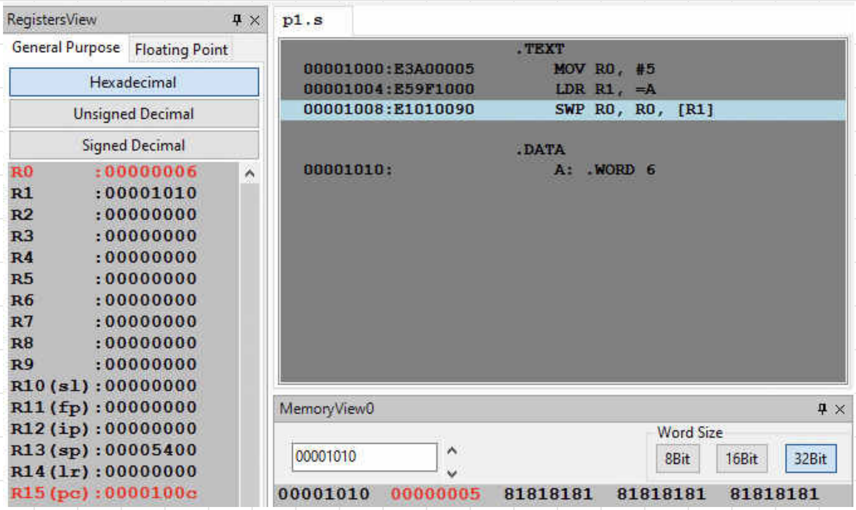
same: swap occurs

Example

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

.DATA
    A: .WORD 6
```

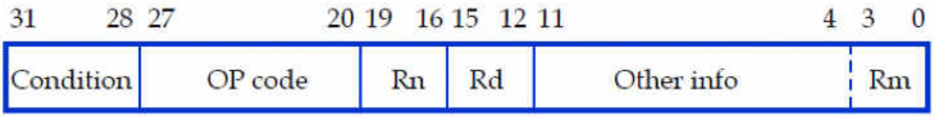




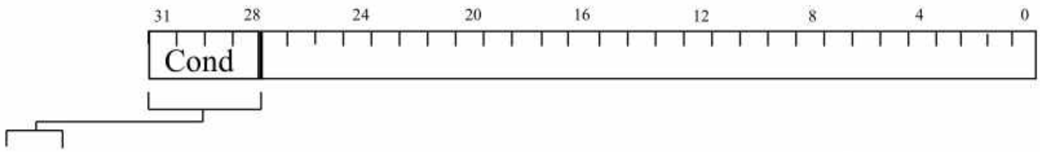
ENCODING *instructions*

```
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 (>)

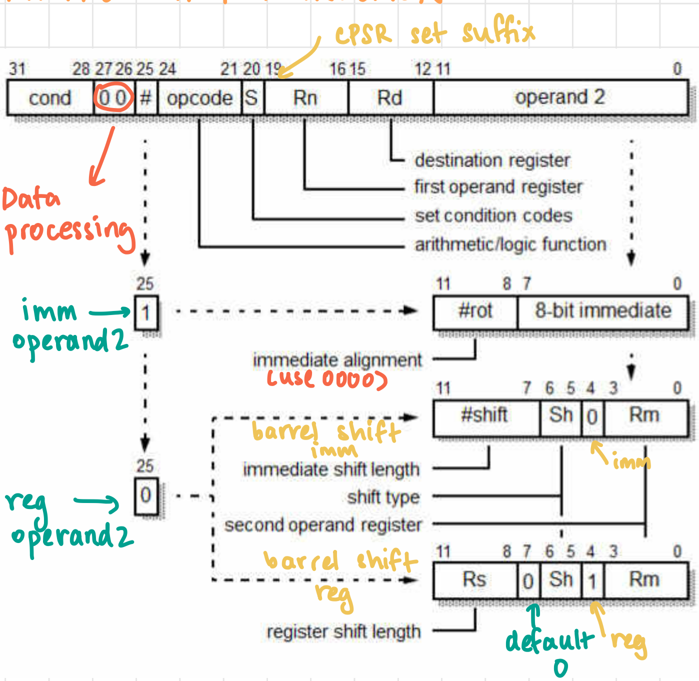
1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

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

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 = AL = 1110	data processing	0	0100 (ADD)	0	0001	0000	00000	00	0	0010

AL			Rn	Rd			Rs
1110	0000	1000	0001	0000	0000	0000	0010
E	0	8	1	0	0	0	2

0x E0810002

ARM sim

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

shift type & amount

IMMEDIATE

11 – 7	6 – 5	4
		0

shift amount
5-bit unsigned

shift type
00 — logical left
01 — logical right
10 — arithmatic right
11 — rotate right

REGISTER

11 – 8	7	6 – 5	4
	0		1

shift register
shift amount specified in reg

default

shift type
00 — logical left
01 — logical right
10 — arithmatic right
11 — rotate right

Example 2

ADDS R1, R0, R2, LSR R4

31–28	27–26	25	24–21	20	19–16	15–12	11–8	7	6–5	4	3–0
Cond	00	#	OpCode	S	Rn	Rd	Rs	0	Sh	1	Rm
no cond = AL = 1110	data processing	0	0100 (ADD)	1	0000	0001	0100	0	01	1	0010

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
no cond = AL = 1110	data processing	0	0100 (ADD)	1	0000	0001	00010	01	0	0010

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

MOVNES R5, R6, ROR R0

(Rd: R5, Rm: R6, Rs: R0)

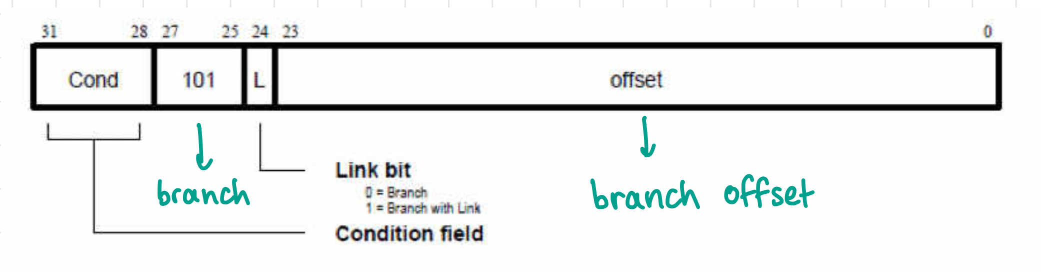
31–28	27–26	25	24–21	20	19–16	15–12	11–8	7	6–5	4	3–0
Cond	00	#	OpCode	S	Rn	Rd	Rs	0	Sh	1	Rm
0001	data processing	0	1101 (MOV)	1	0000 (not present)	0101	0000	0	11	1	0110

0001	0001	1011	0000	0101	0000	0111	0110
1	1	B	0	5	0	7	6

0x 11B05076

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

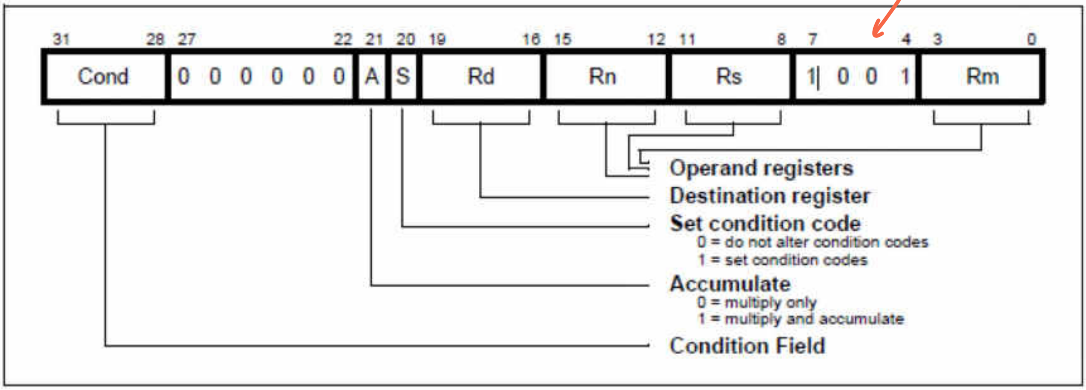
BRANCH INSTRUCTIONS



BL , B

MULTIPLICATION INSTRUCTIONS

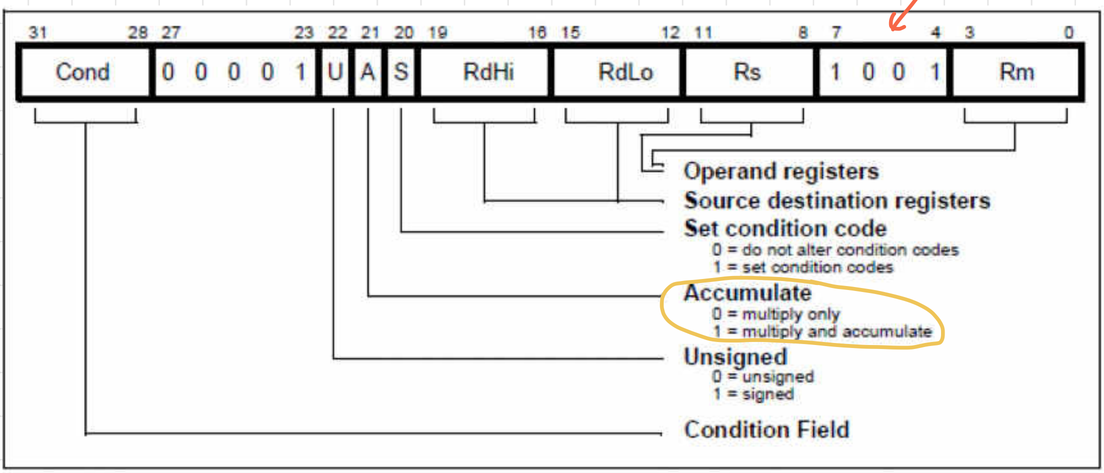
multiplication




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

LARGE MULTIPLICATIONS

multiplication



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

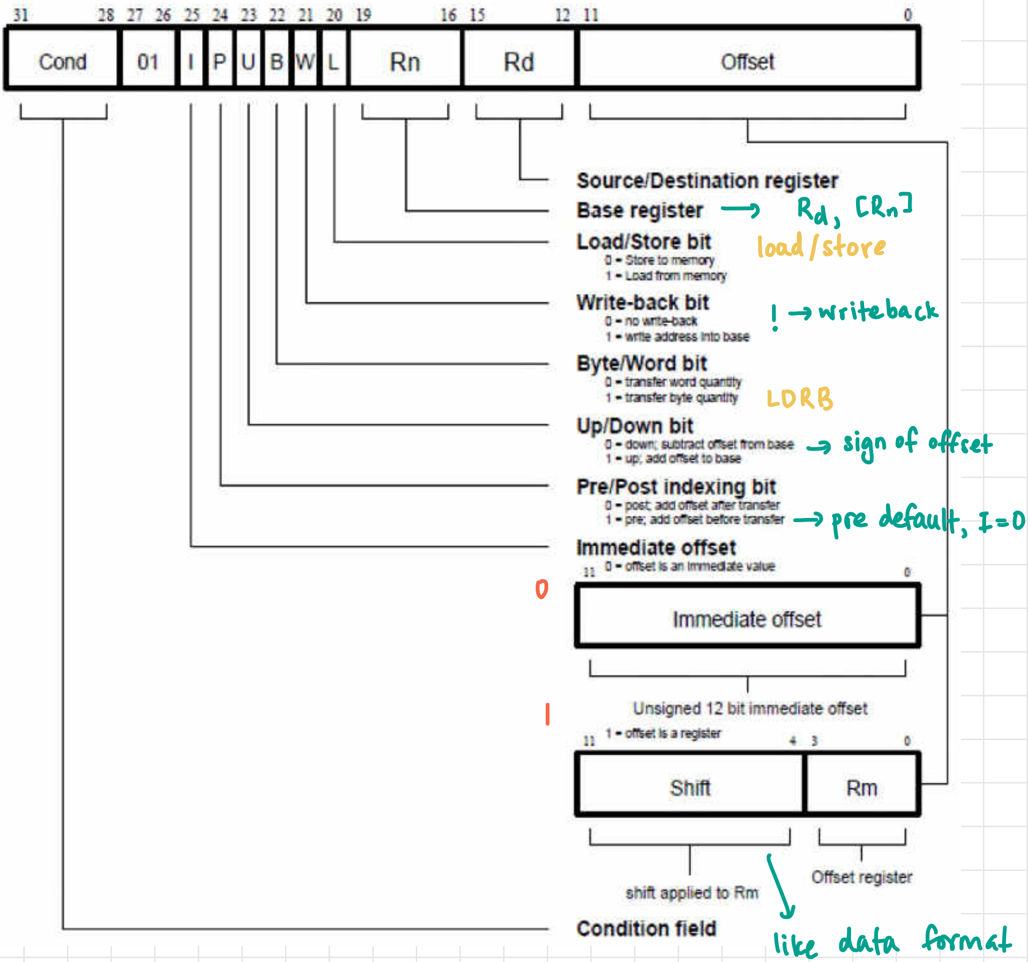
31 28	27 22	21	20	19 16	15 12	11 8	7 4	3 0
Cond	000000	A	S	Rd	Rn	Rs	1001	Rm
↓	↓	↓	↓	↓	↓	↓	↓	↓
1110	000000	0	0	0010	0000	0100	1001	0011

0xE0020493

.TEXT

00001000:E0020493 MUL R2, R3, R4

DATA TRANSFER INSTRUCTION



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

31–28	27–26	25	24	23	22	21	20	19–16	15–12	11–0
Cond	01	I	P	U	B	W	L	Rn	Rd	offset
1110	01	0	1	1	0	0	0	0001	0000	0000 0000 0000
			default (pre)							

0x E5810000

.TEXT	
00001000:E5810000	STR R0, [R1]

LDR R0, [R1], R2

31–28	27–26	25	24	23	22	21	20	19–16	15–12	11–4	3–0
Cond	01	I	P	U	B	W	L	Rn	Rd	Shift	Rm
1110	01	1	0	1	0	0	1	0001	0000	0000 0000	0010

post

implicitly defined in post

0xE6910002

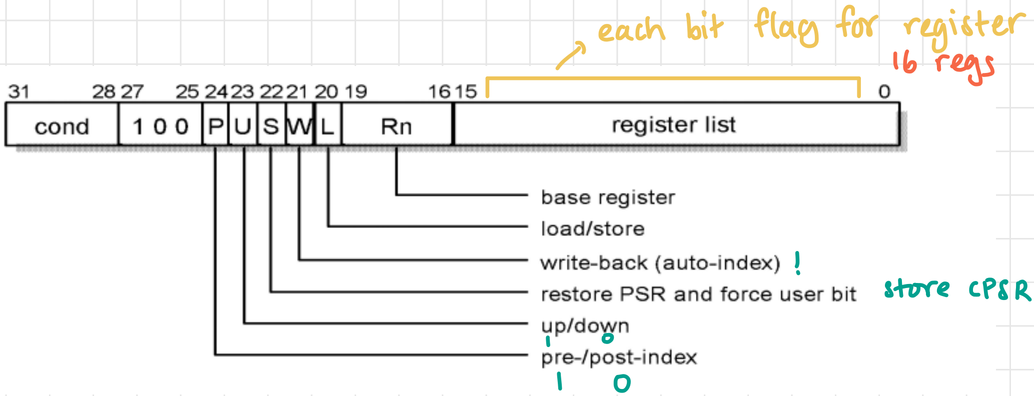
.TEXT	
00001000:E6910002	LDR R0, [R1], R2

BLOCK TRANSFER

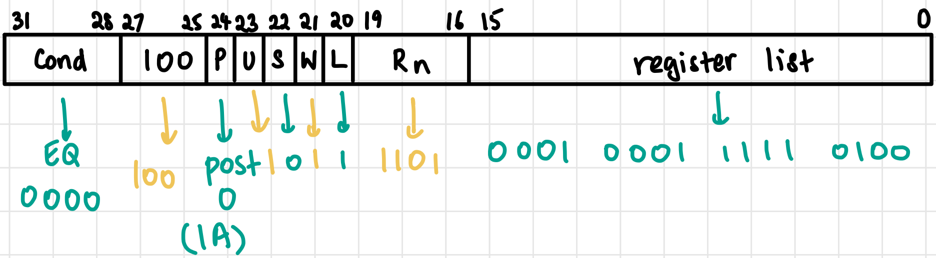
Values

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}

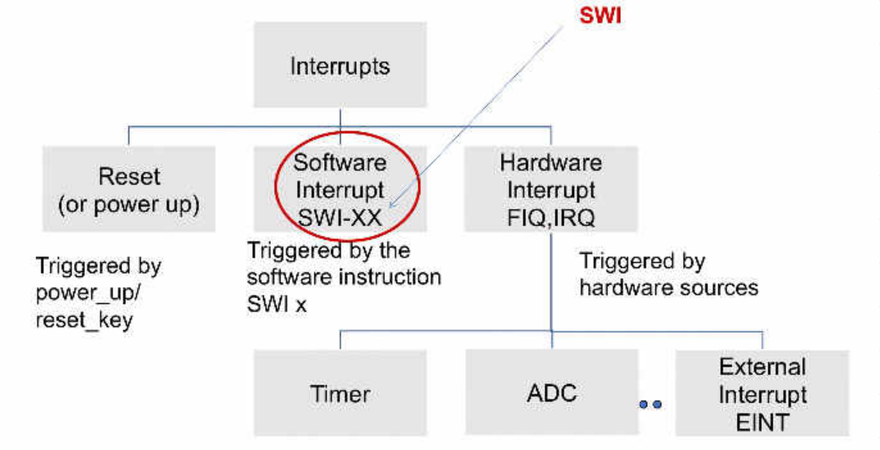


0x08BD11F4

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

INTERRUPTS

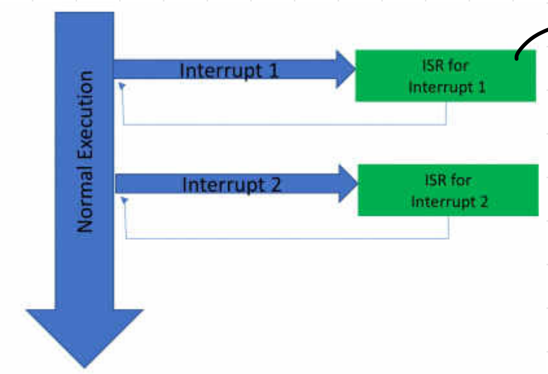
- normal program flow is disrupt
- interrupt service routine (ISR)



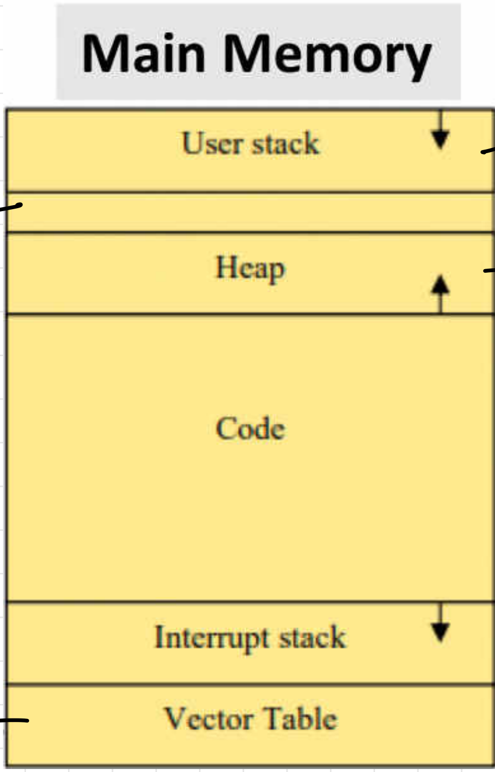
Serving Interrupts

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

Event Driven Tasks Execution



stored in memory



local variables

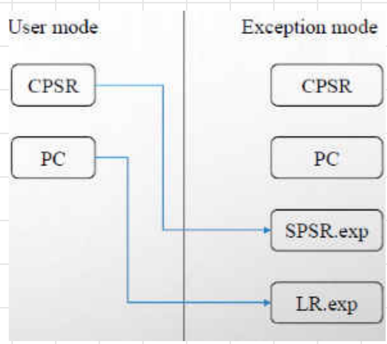
global variables

dynamically allocated

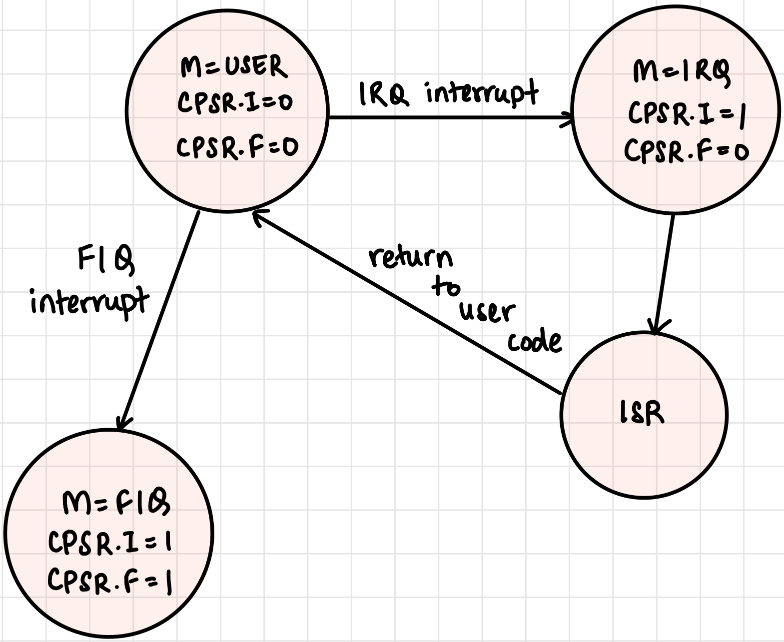
interrupt: address mapping for ISRs

ARM Exception Handling

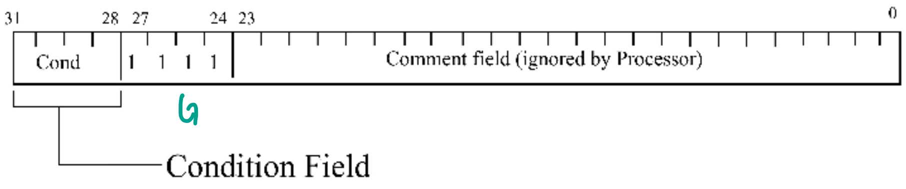
- CPSR ⟶ SPSR of exception mode
- PC ⟶ LR of exception mode
- CPSR set to exception mode
- PC ⟶ address of exception handler



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_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 handleor 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