Introduction to Embedded Systems

Unit 1.3: Overview of Programming Languages, Assembly Programming, and ARM Architecture and Assembly

Alexander Yemane
BCIT
INCS 3610
Fall 2025

Programming Languages

Levels of program code

- A programmer can write programs for a computer in the following ways:
  - **High-level language**
    - Level of abstraction is closer to problem domain
    - High-level language programs are translated into assembly code or machine code
  - **Assembly language**
    - Is a lower-level programming language that is machine-dependent
    - Symbolic representation of instructions that computer hardware understands and obeys
    - Assembly language programs are translated into machine code
  - **Machine language**
    - Is the lowest level of computer software, i.e. zeros and ones
    - Directly executed by a computer system

Programmer’s view

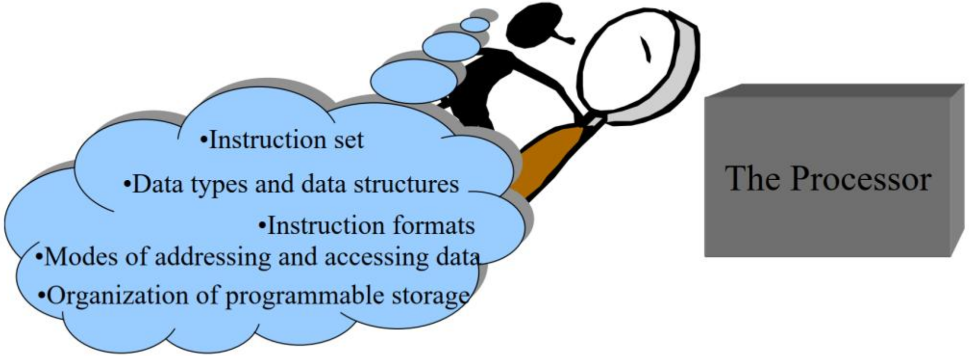
- Programmers don’t need a detailed understanding of architecture to write programs
- Most development today done using high-level languages (C, C++, Java, etc.)
  - Development is faster
  - Maintenance is easier
  - Programs are portable
- But some assembly level programming may still be necessary
  - E.g. Drivers: portion of program that communicates with and/or controls (drives) another device

Programmer’s view: why assembly language?

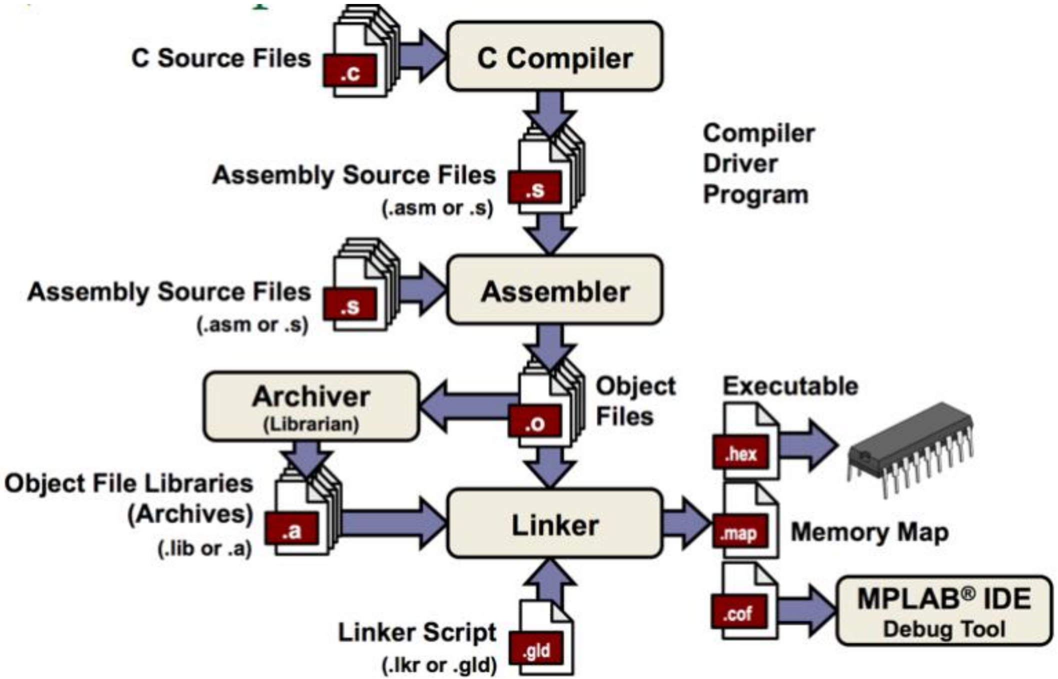
- Accessibility to system hardware
- Space and time efficiency
- Compiler design and optimizations

Programmer’s view: ISA as the manual

- Particularly, embedded system designers use the *instruction set architecture* (ISA) to help determine which processor is the best solution.
  - Does the processor provide specialized instructions that are useful?
  - Does the processor provide optimal ways to implement the functions of the application?



Integrated development environment



Integrated development environment

- Implementation phase
  - *Compilers* translate structured programs into assembly (or machine) programs
  - *Assemblers* translate assembly instructions to binary machine instructions
  - A *linker* allows a programmer to create a program in separately-assembled files
    - Separating object and library modules help make the compilation process more efficient
- Verification phase
  - *Debuggers* and *profilers* help programmers evaluate and correct their programs
  - *Emulators* support debugging of the program while it executes on the target processor.

Example C, assembly, and machine programs

C Program

```c
int main(void){
  int i;
  int total = 0;
  for (i = 0; i < 10; i++) {
    total += i;
  }
  while(1); // Dead loop
}
```

Compile

Assembly Program

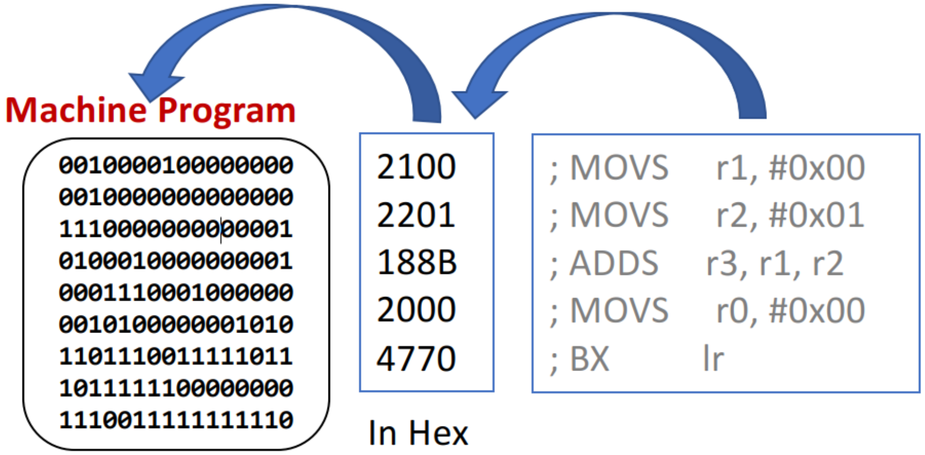
```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

Assemble

Machine Program

```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

Example C, assembly, and machine programs



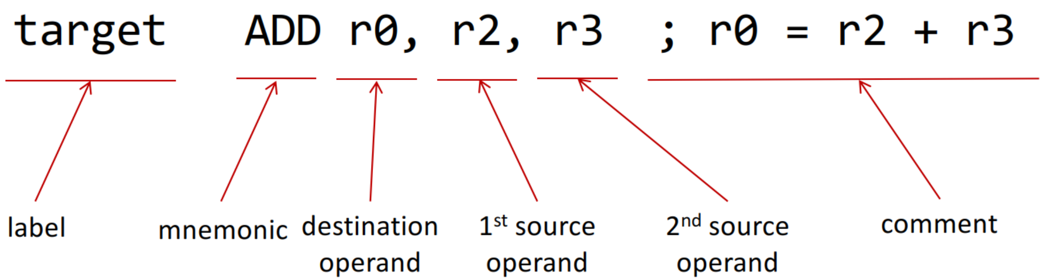
Assembly Language Programming

Assembly languages

- There are many different types of assembly languages
  - x86 – used in most modern Intel PCs
  - ARM – used in smartphones, tablets, embedded systems (Raspberry Pi)
  - AVR – used in embedded systems (Arduino Uno)
  - etc.

Assembly language instructions

- Instructions in assembly language programs follow a similar *instruction format*:

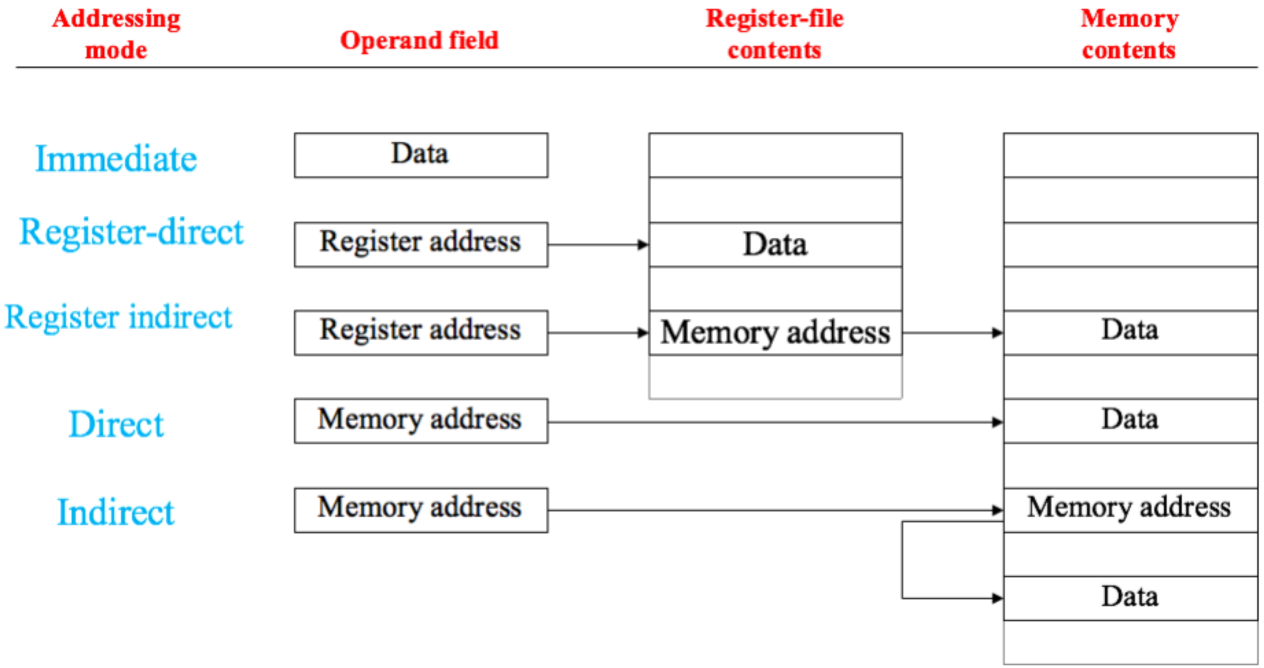


Assembly language instruction format

```
label            mnemonic operand1, operand2, operand3    ; comments
```

- The *label* is a symbolic reference to this instruction's address in memory.
- The *mnemonic* (or *op-code*) represents the operation to be performed.
- The number of *operands* varies, depending on each specific instruction. Some instructions have no operands at all.
    - operand1 is typically the destination register, and operand2 and operand3 are source operands
    - operand2 is usually a register
    - operand3 can represent many different things, depending on the instruction.
- Everything after the semicolon is a *comment*, which is ignored by the assembler.

Instruction operands and addressing modes



Assembler directives

- There are some "instructions" that aren't actual instructions
- Tell the assembler to do something as opposed to the processor, e.g. to allocate space or define types

```
// Setup a numerical constant for use later
.equ    PADS_BANK0_BASE, 0x4001c000

// Allocate 4-bytes in memory, setting the value to 0
.word   0x00000000
```

Assembler directives

- `.equ` – Symbolic name for a 32-bit value
- `.cpu` – Declare the CPU this assembly is for
- `.thumb_func` – Declare that this is thumb assembly
- `.global` – Export a symbol globally to the linker

- `.word` – Allocate 4-bytes here
- `.space` – Allocate a certain amount of space
- `.align` – Align this memory location by a certain multiple

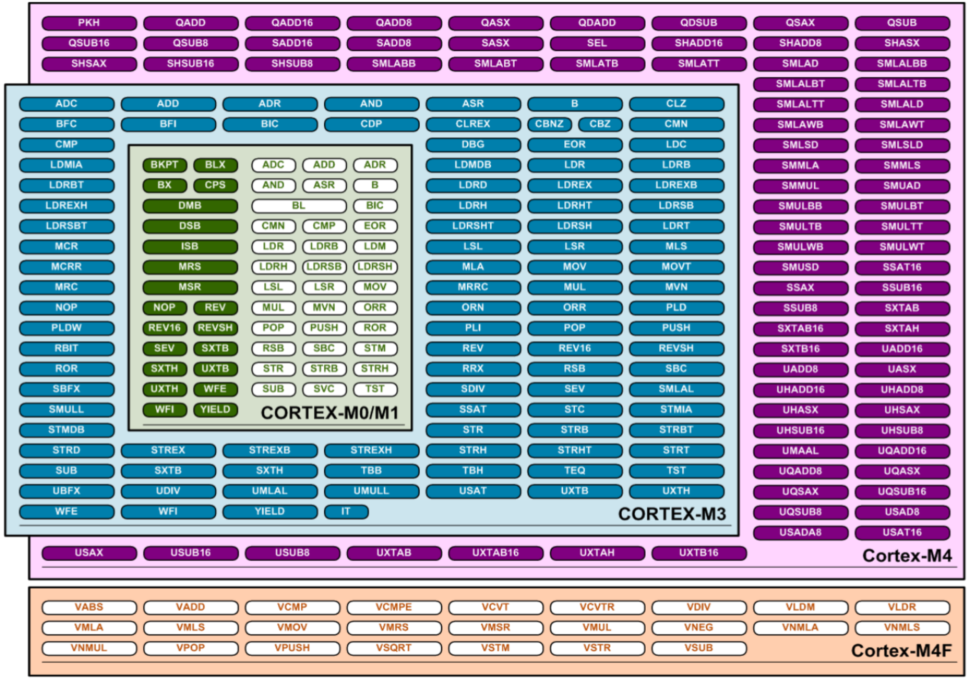
- `.section` – Start a new section
  - `.data` – Store this in the data segment
  - `.bss` – Store this in the bss segment (uninitialized)
  - `.text` – Store this in the text segment

ARM Features and Instruction Set

ARM instruction sets

- Modern ARM processors have several instruction sets:
  - The fully-featured 32-bit *ARM* instruction set
  - The more restricted, but space efficient, 16-bit *Thumb* instruction set
  - The newer mixed 16/32-bit *Thumb-2* instruction set
    - Thumb-2 is the progression of Thumb
    - It improves performance while keeping the code density tight by allowing a mixture of 16- and 32-bit instructions.
  - The 64-bit *ARM* instruction set

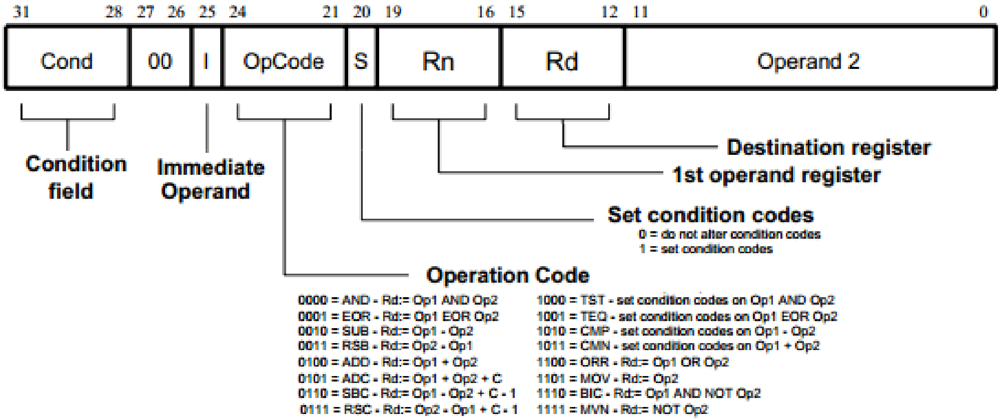
Assembly instruction sets for Cortex-M



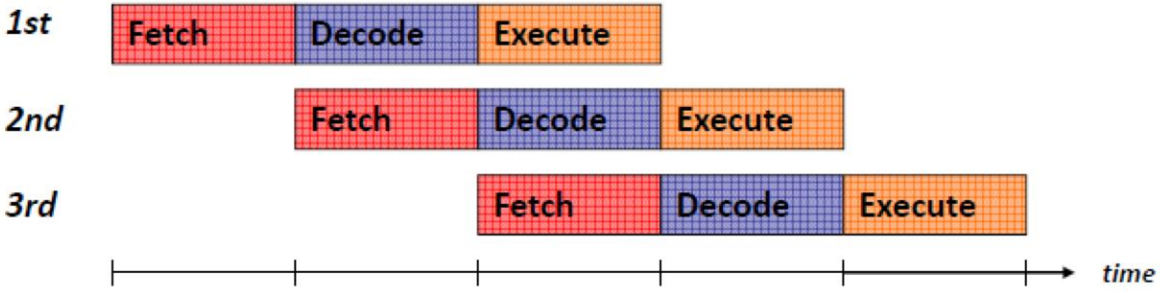
ARM instruction format

- We will concentrate on one instruction set, ARM7
- Datasheet available at: https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf

- Features multiple load and store operations (ARM does not support 'memory-to-memory' operations)
- Fixed length 32-bit instructions
- All instructions can be executed conditionally

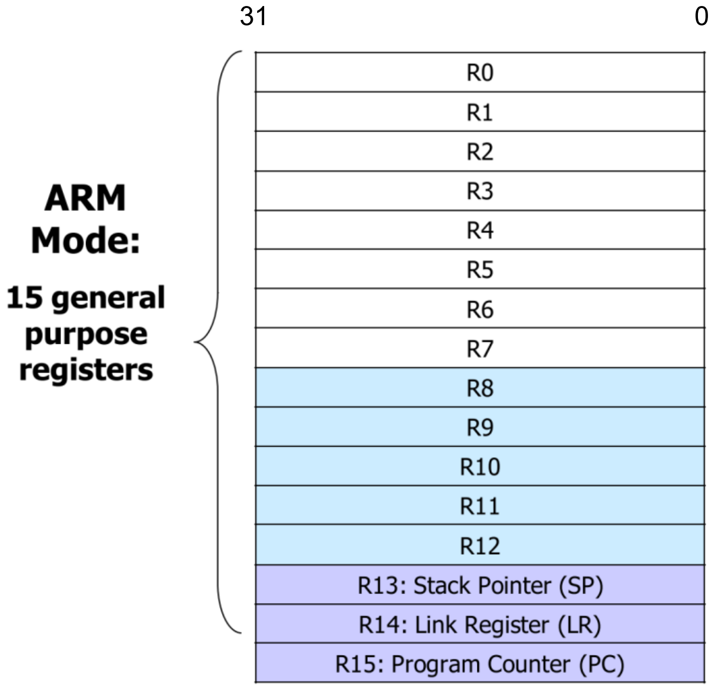


ARM7 pipeline execution

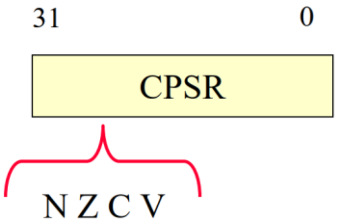


- 3-stage pipeline design allows effective throughput to increase to one instruction per clock cycle
- Allows the next instruction to be fetched while still decoding or executing the previous instructions

ARM registers (user mode)

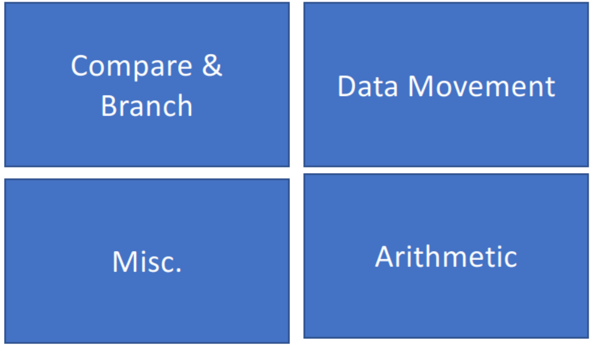


Current Program Status Register



Assembly instructions supported

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
  - Load, Store, Move
- Compare and branch
  - Compare, Test, If-then, Branch, Compare and branch on zero
- Miscellaneous (mainly control flow)
  - Breakpoints, Wait for events, Interrupt enable/disable, Data memory barrier, Data synchronization barrier



Data movement and load instructions

Remember:
R has two components:
- Register Address
- Register Content

```
MOV – Move
MVN – Move NOT
LDR - Load
```

```
MOV Rn, #imm          ; Load a (small) immediate value
MOV Rn, Rm            ; Copy one register to another
LDR Rn, [Rm]          ; Rn = value pointed by Rm
LDR Rn, [Rm,#4]       ; Rn = *(Rm+4) – offset can be +/-
LDR Rn, [Rm,Rp]       ; Rn = *(Rm+Rp) - offset can be +/-
```

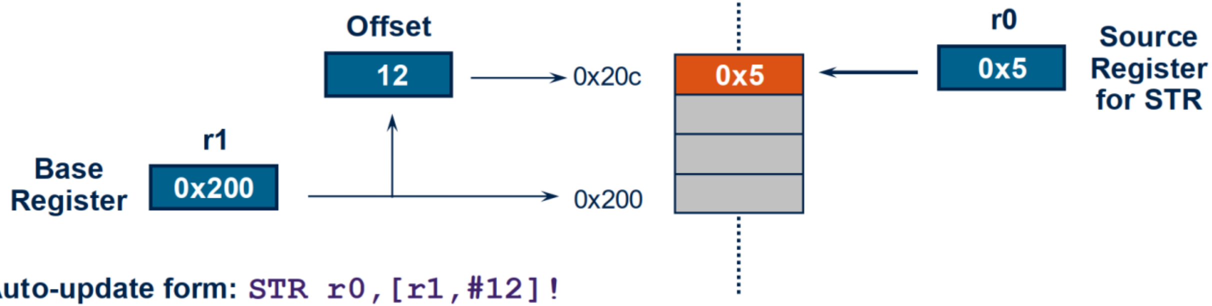
Loading large immediate values

- You can’t put a 32-bit immediate value into the instruction
- So you need to store in memory somewhere and load it
- An assembler directive and pseudo-instruction can help

```
; Put this at the top of the program
.equ SOME_NAME 0x12345678

; Put this down in the code
LDR R0, =SOME_NAME
```

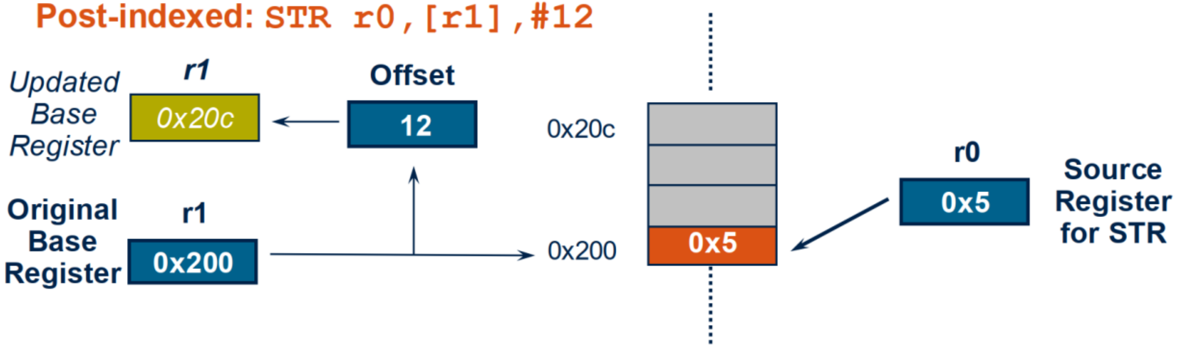
Offsets: choice of pre-indexed or post-indexed addressing

- **Pre-indexed:** `STR r0,[r1,#12]`

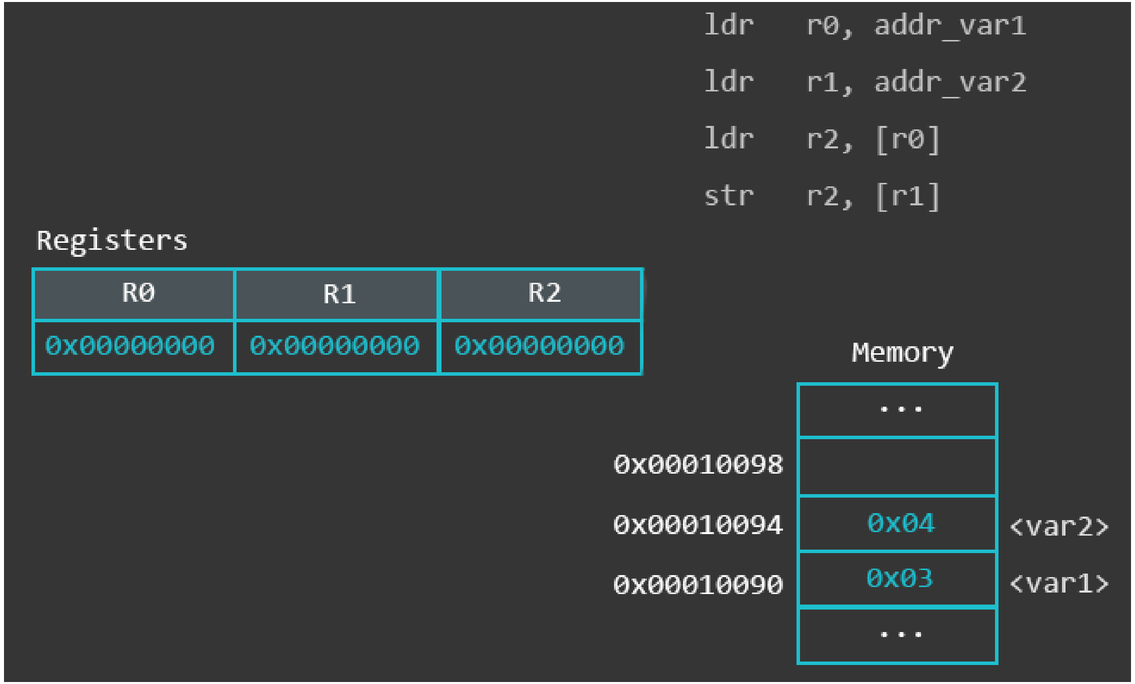


Auto-update form: `STR r0,[r1,#12]!`

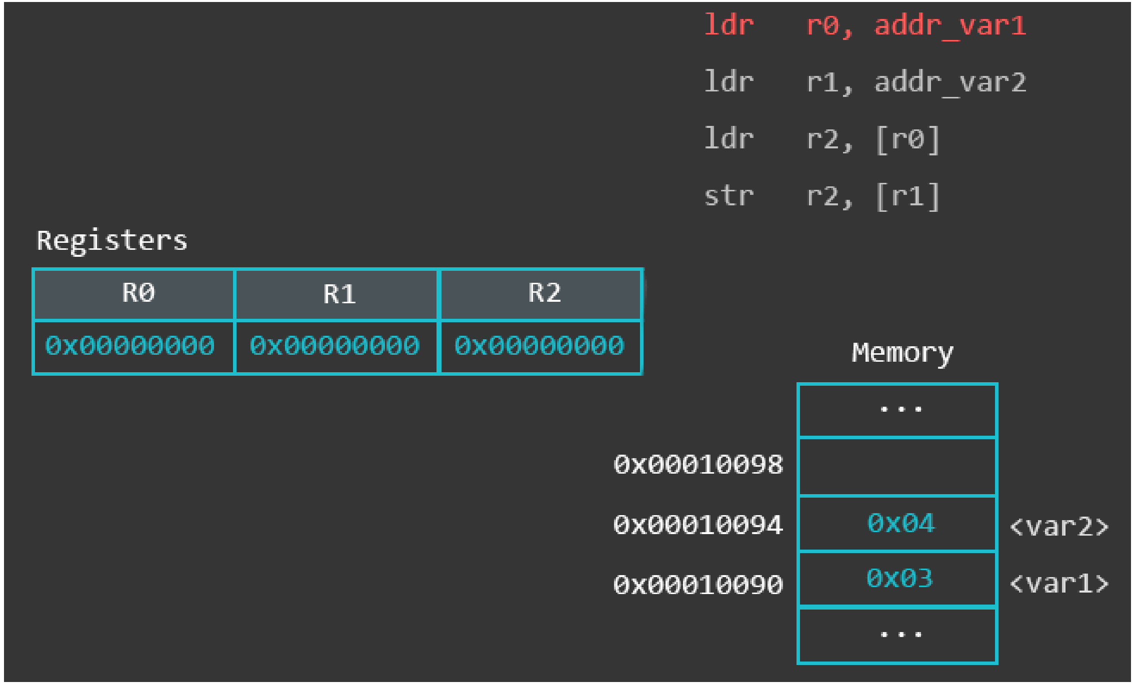
- **Post-indexed:** `STR r0,[r1],#12`



LDR example #1



LDR example #1



LDR example #1

```
ldr    r0, addr_var1
ldr    r1, addr_var2
ldr    r2, [r0]
str    r2, [r1]
```

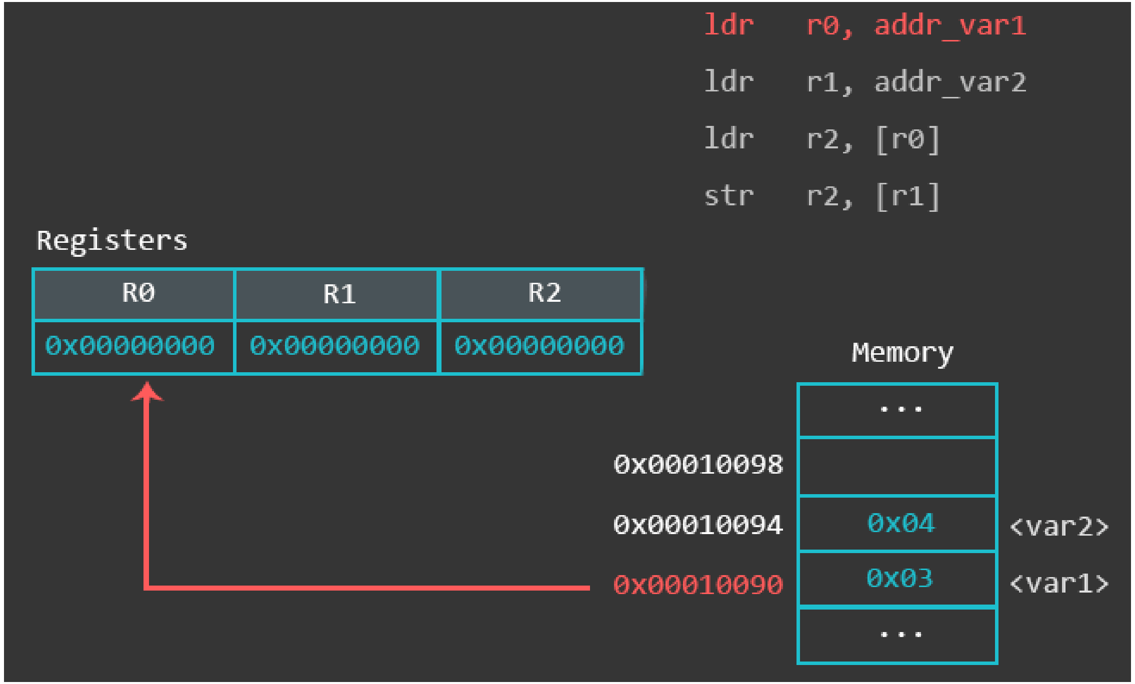
Registers

R0	R1	R2
0x00000000	0x00000000	0x00000000

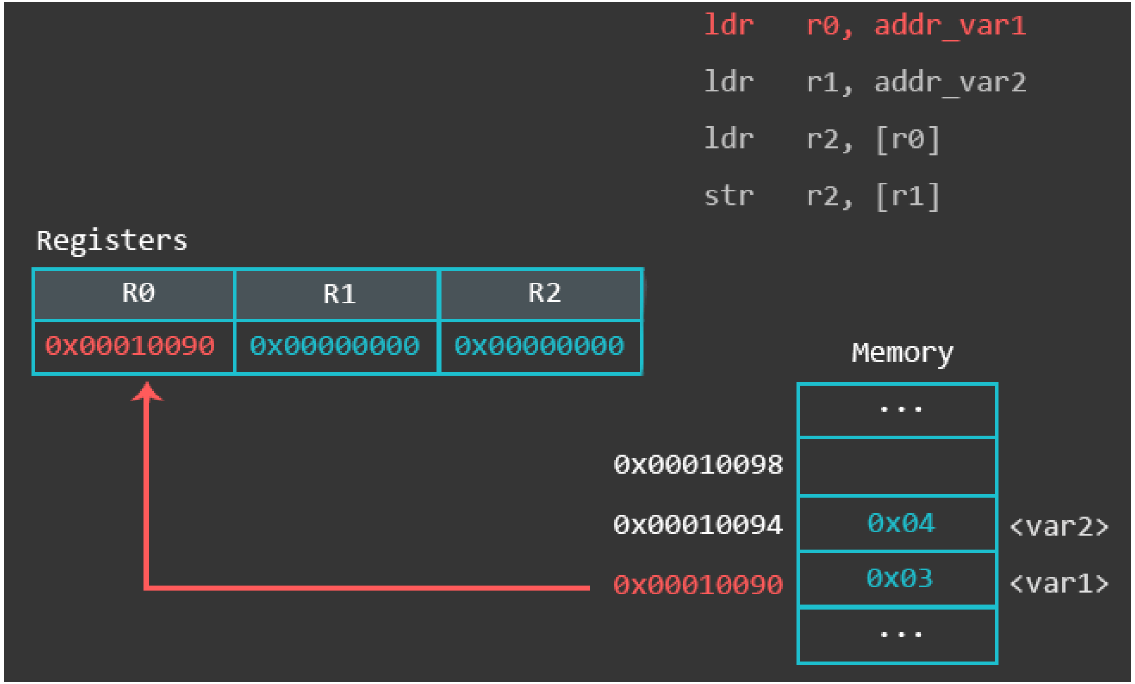
Memory

	...	
0x00010098		
0x00010094	0x04	<var2>
0x00010090	0x03	<var1>
	...	

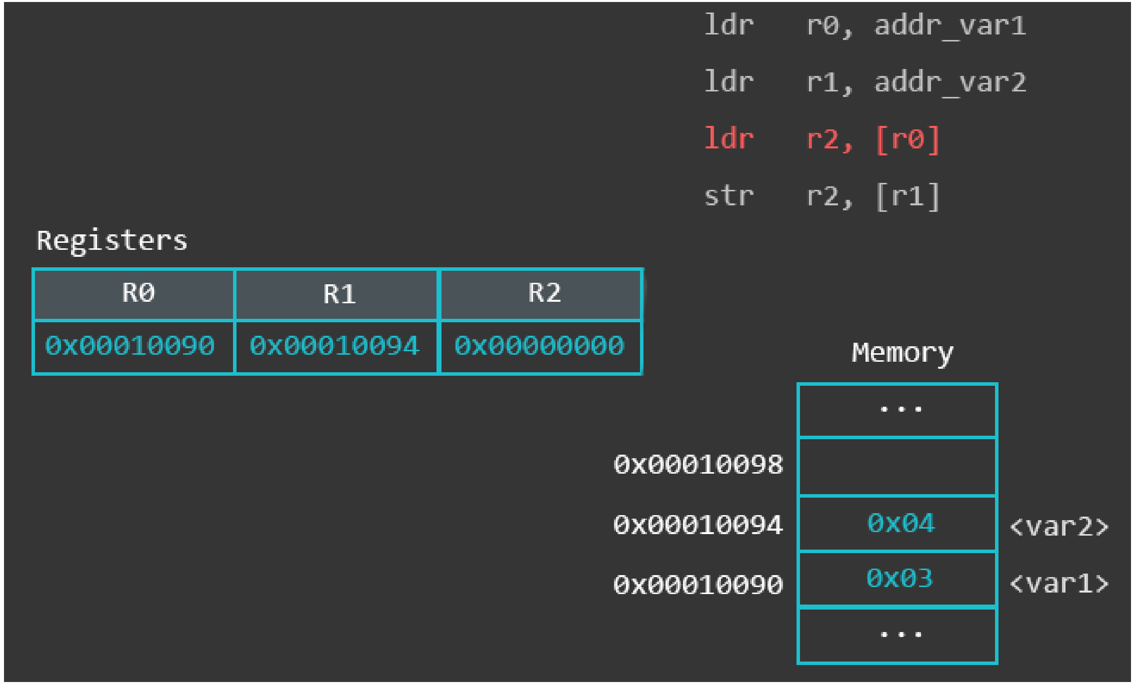
LDR example #1



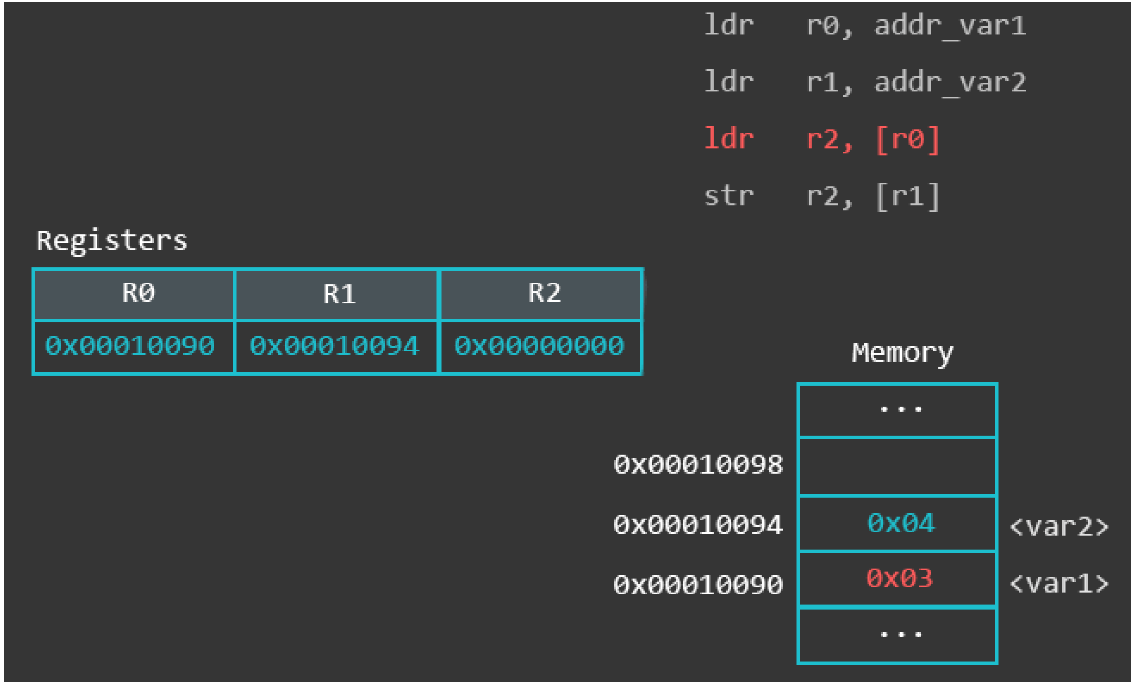
LDR example #1



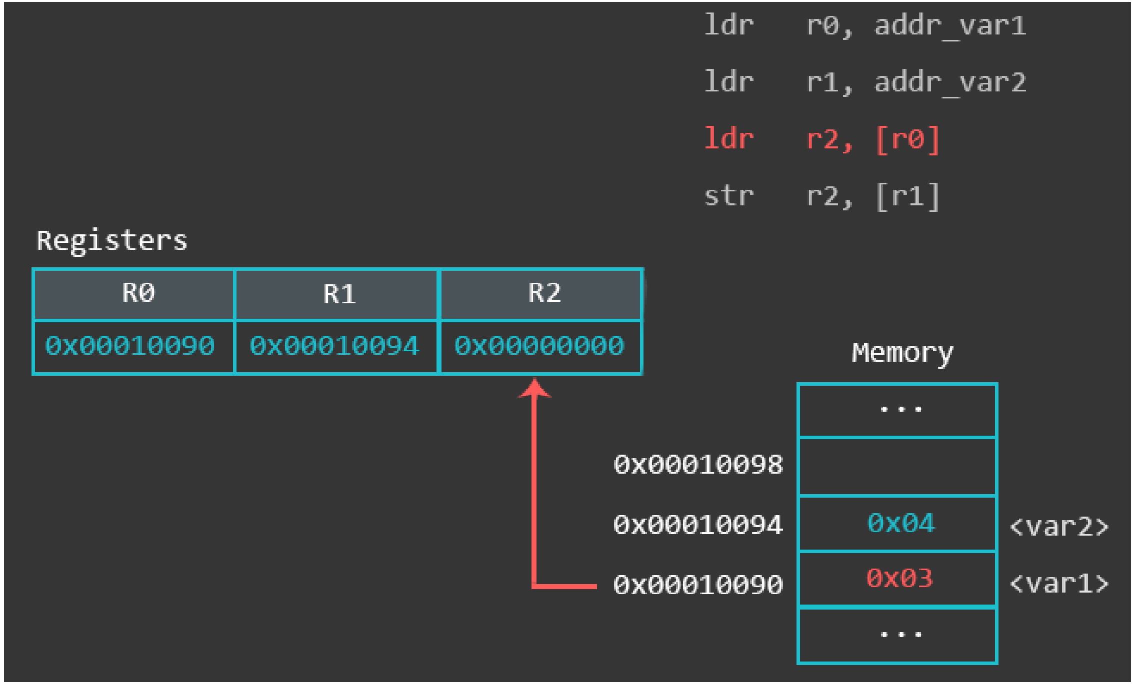
LDR example #2



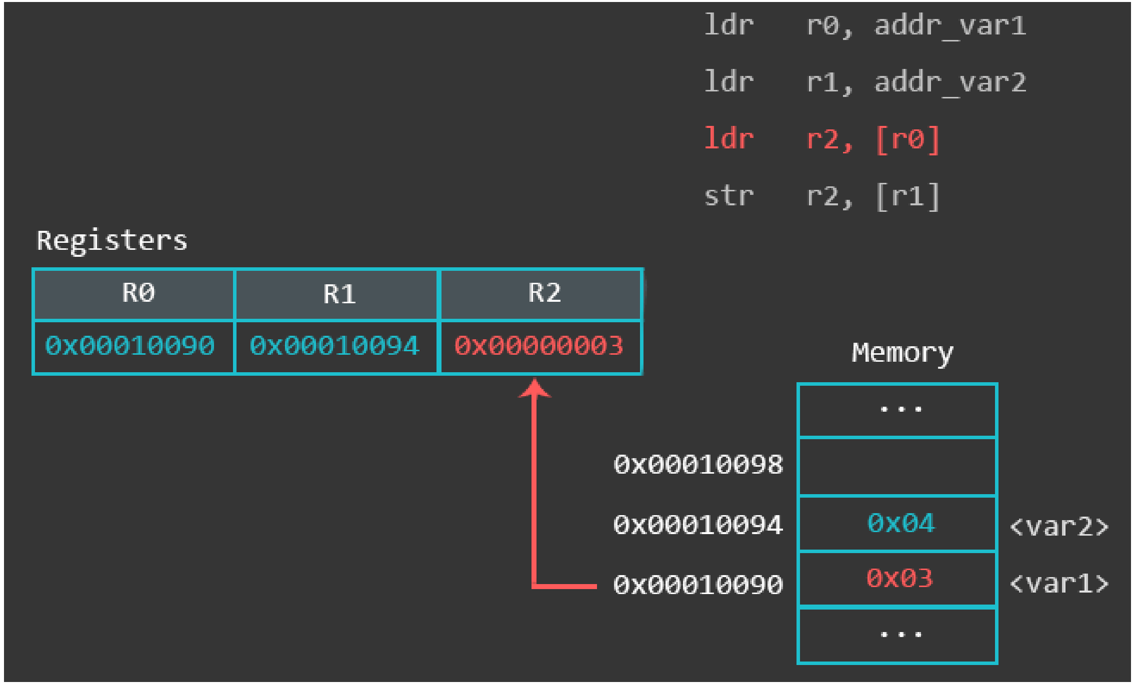
LDR example #2



LDR example #2



LDR example #2

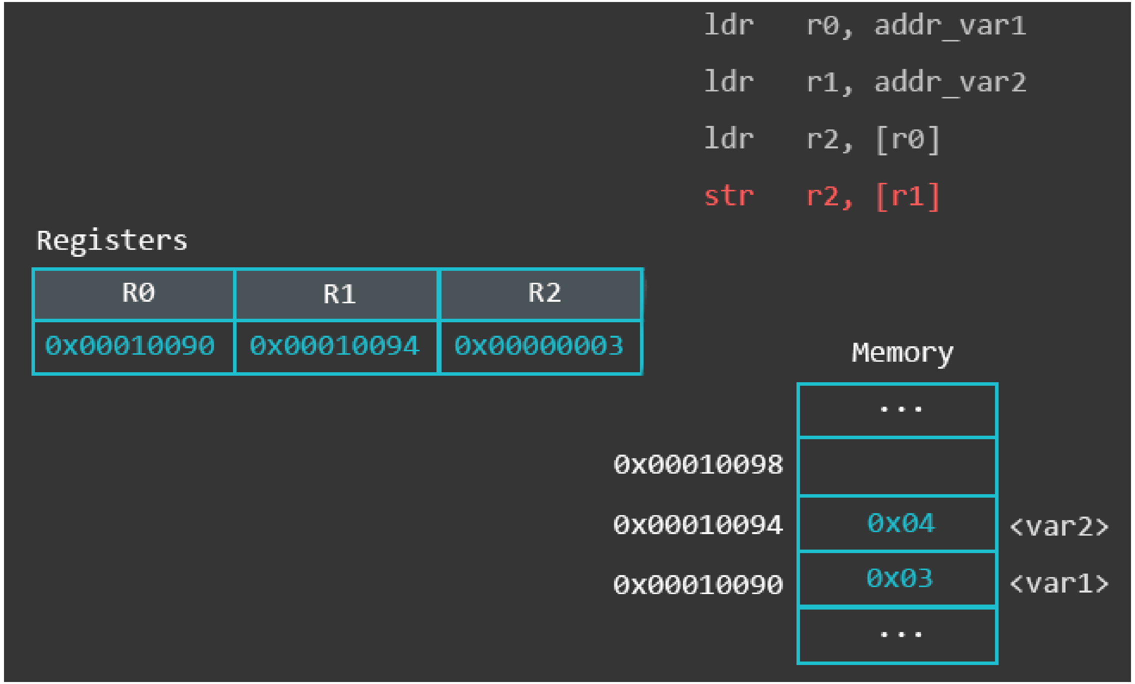


Store instructions

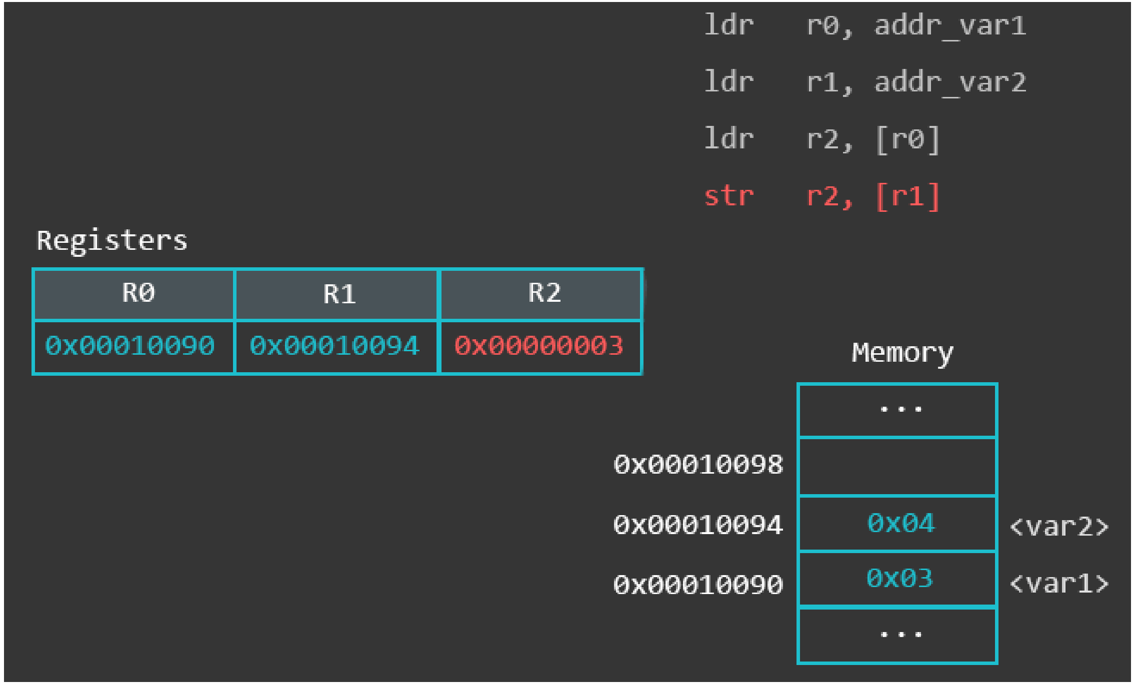
STR – Store

```
STR Rd, [Rn]          ; *Rn = Rd
STR Rd, [Rn,#N]       ; *(Rn+N) = Rd
STR Rd, [Rn,Rm]       ; *(Rn+Rm)= Rd
```

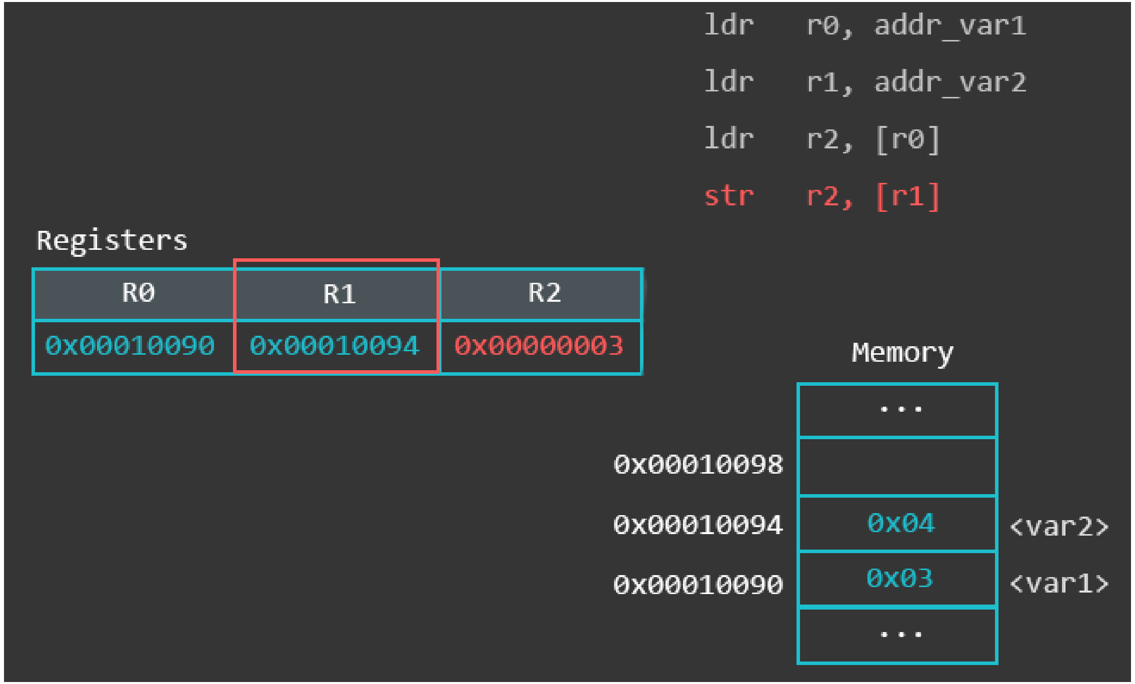
STR example


ldr r0, addr_var1
ldr r1, addr_var2
ldr r2, [r0]
str r2, [r1]
Registers
R0
R1
R2
0x00010090
0x00010094
0x00000003
Memory
...
0x00010098
0x00010094
0x04
<var2>
0x00010090
0x03
<var1>
...

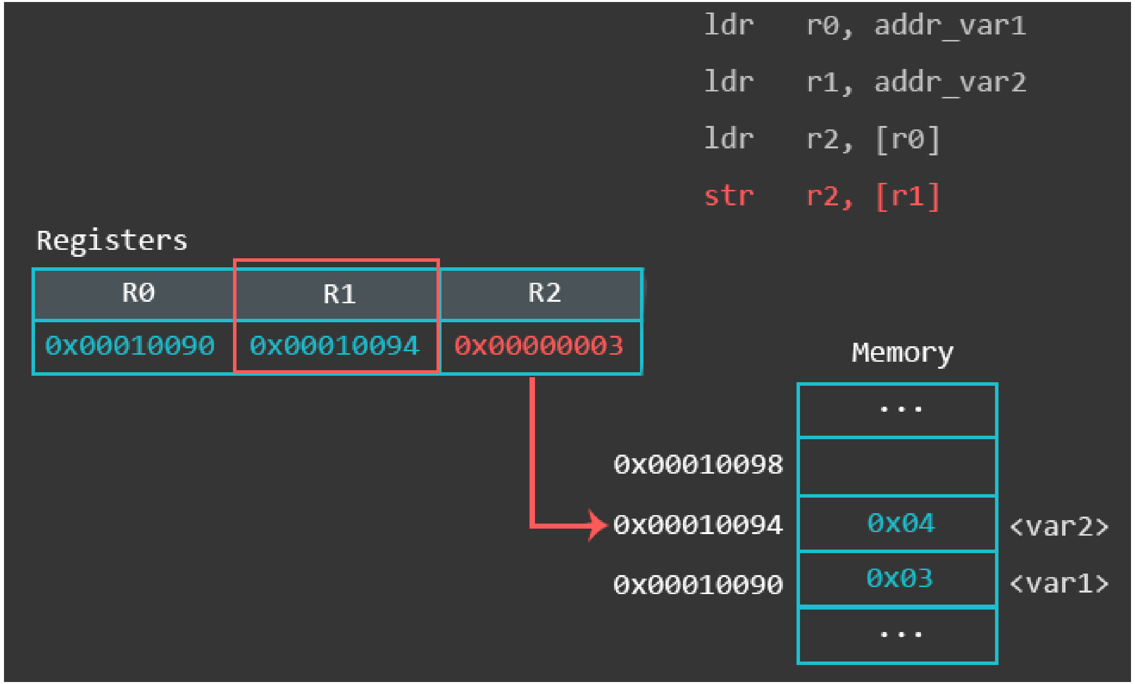
STR example


ldr r0, addr_var1
ldr r1, addr_var2
ldr r2, [r0]
str r2, [r1]
Registers
R0
R1
R2
0x00010090
0x00010094
0x00000003
Memory
...
0x00010098
0x00010094
0x04
<var2>
0x00010090
0x03
<var1>
...

STR example

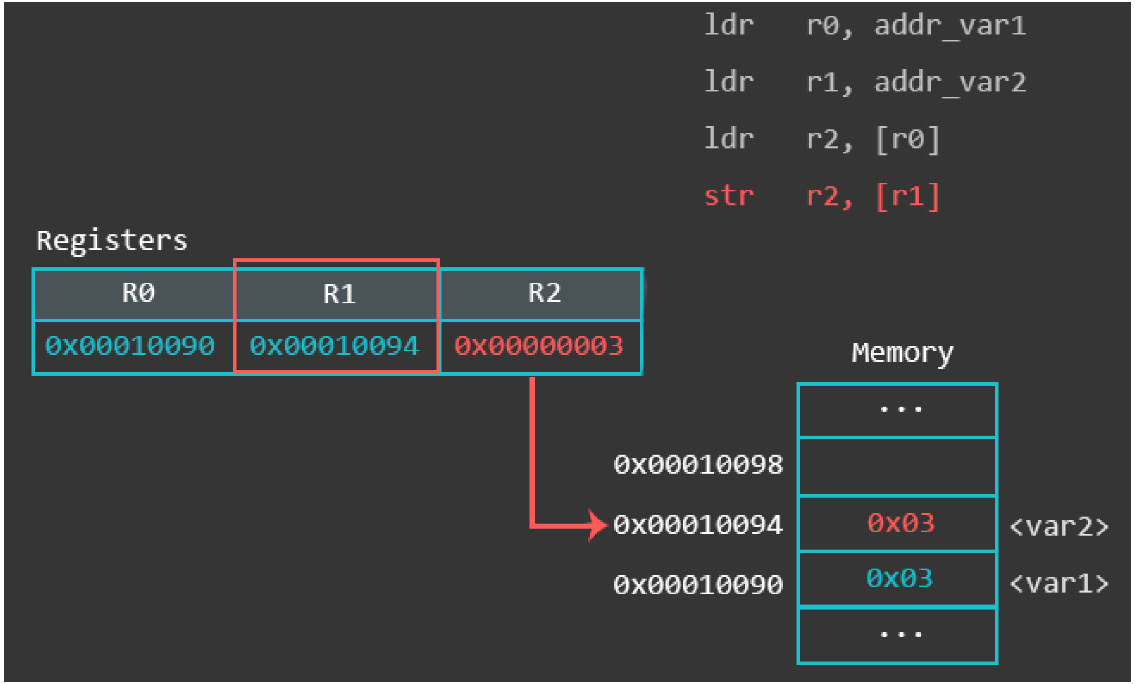

ldr r0, addr_var1
ldr r1, addr_var2
ldr r2, [r0]
str r2, [r1]
Registers
R0
R1
R2
0x00010090
0x00010094
0x00000003
Memory
...
0x00010098
0x00010094
0x04
<var2>
0x00010090
0x03
<var1>
...

STR example



ldr r0, addr_var1
ldr r1, addr_var2
ldr r2, [r0]
str r2, [r1]
Registers
R0
R1
R2
0x00010090
0x00010094
0x00000003
Memory
...
0x00010098
0x00010094
0x04
<var2>
0x00010090
0x03
<var1>
...

STR example


ldr r0, addr_var1
ldr r1, addr_var2
ldr r2, [r0]
str r2, [r1]
Registers
R0
R1
R2
0x00010090
0x00010094
0x00000003
Memory
...
0x00010098
0x00010094
0x03
<var2>
0x00010090
0x03
<var1>
...

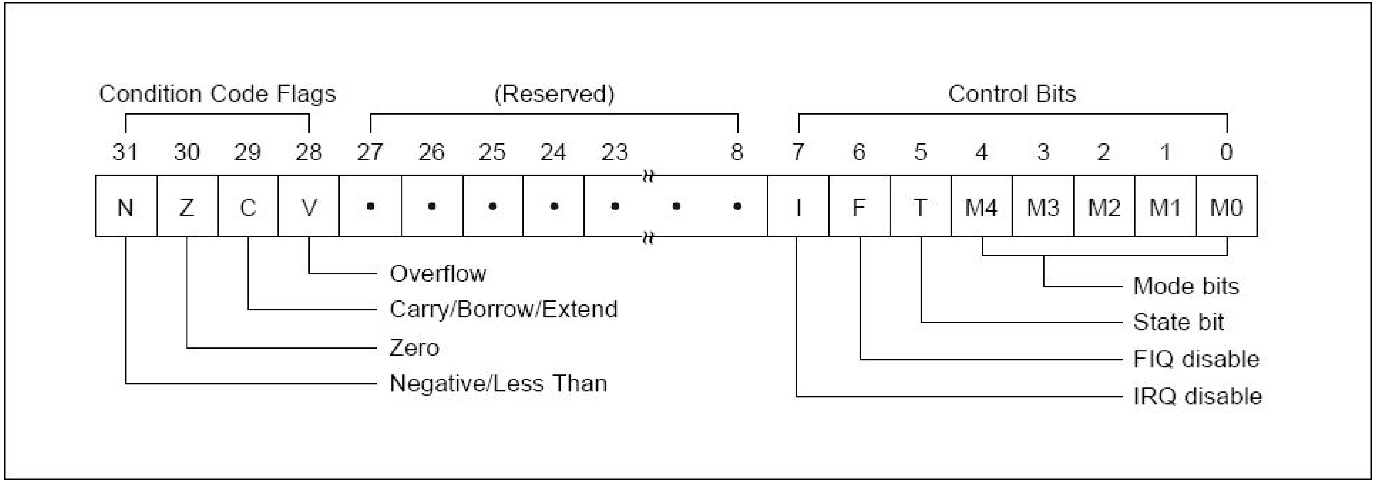
Preface to arithmetic operations: CPSR condition code flags

- After an addition/subtraction operation, certain status flags in the CPSR can be set

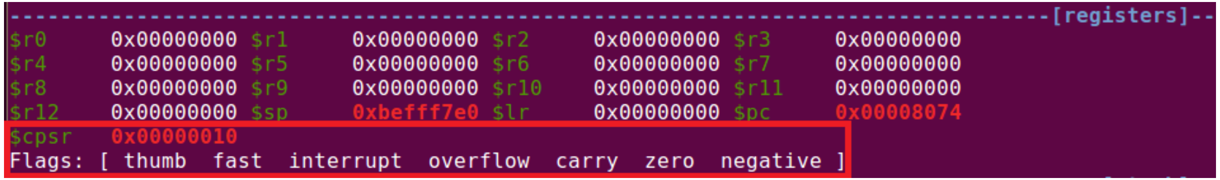
Bit	Name	Meaning after add or sub
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow
C	carry	unsigned overflow

- C bit set after an unsigned addition if the answer is “wrong”
- C bit cleared after an unsigned subtract if the answer is “wrong”
- V bit set after a signed addition or subtraction if the answer is “wrong”

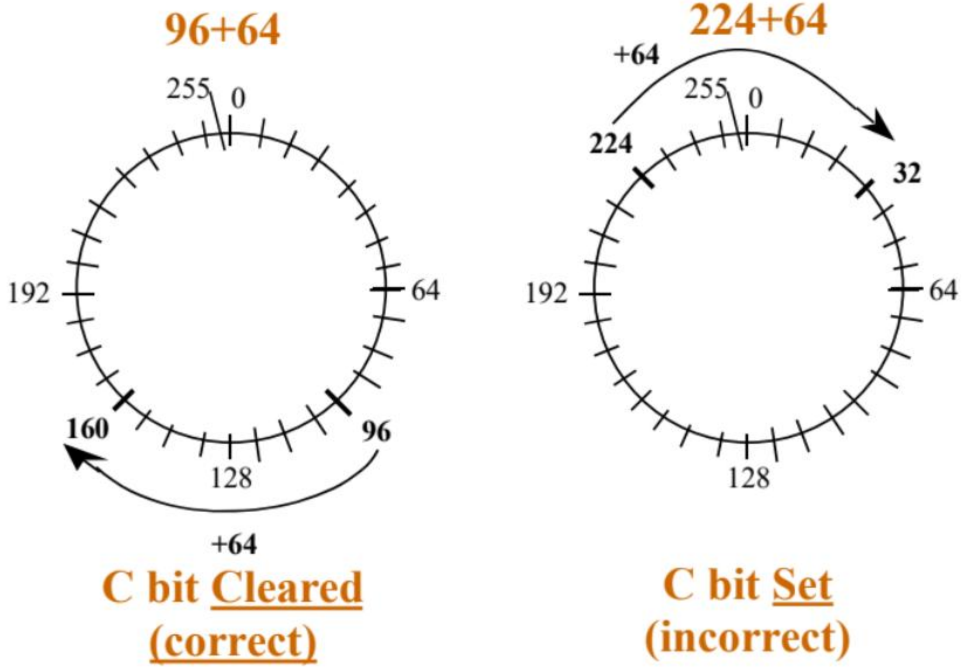
CSPR flags and control bits



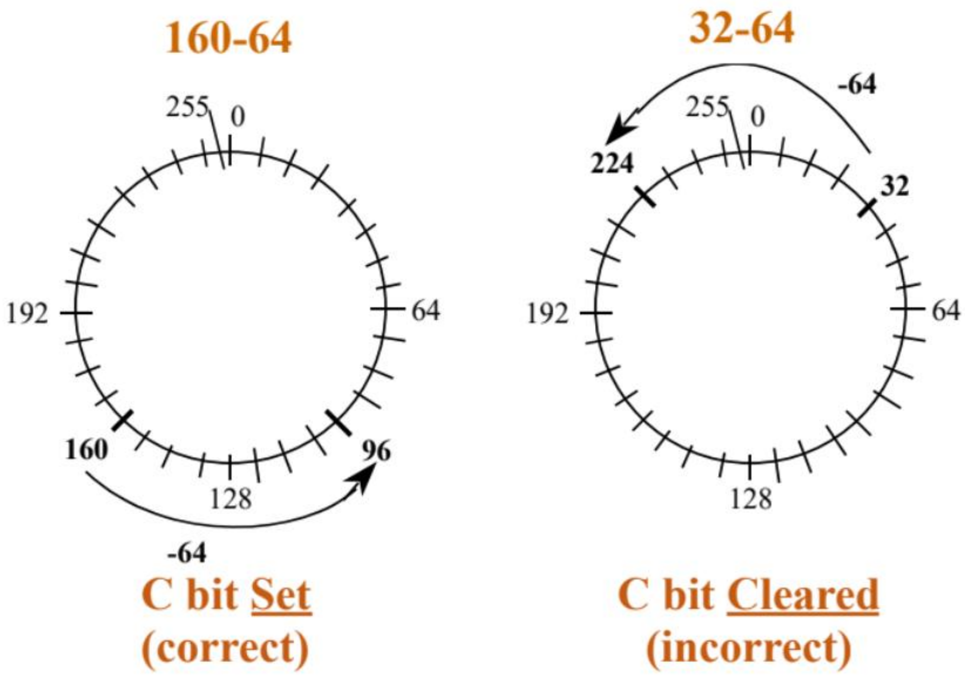
View in debugger (gdb):



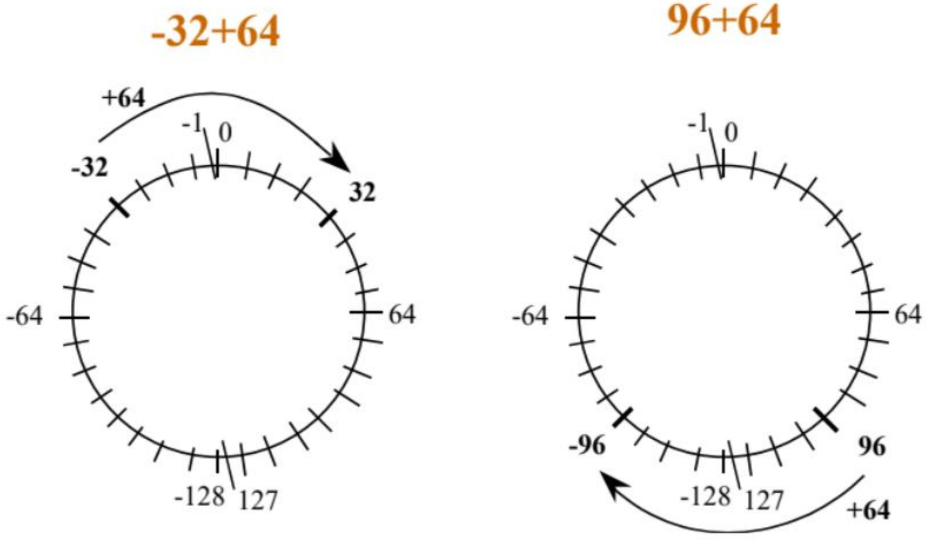
Unsigned addition carry bit



Unsigned subtraction carry bit



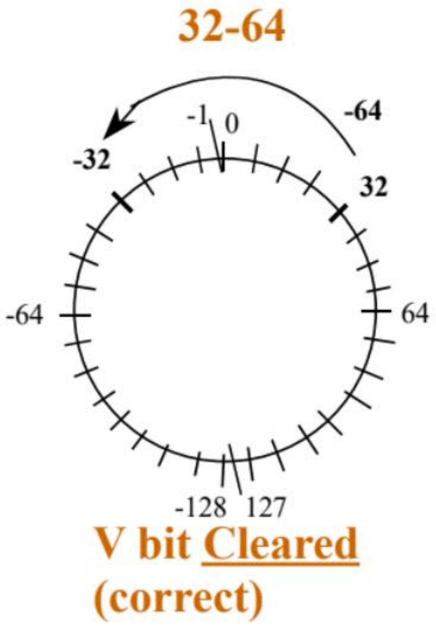
Signed addition overflow bit

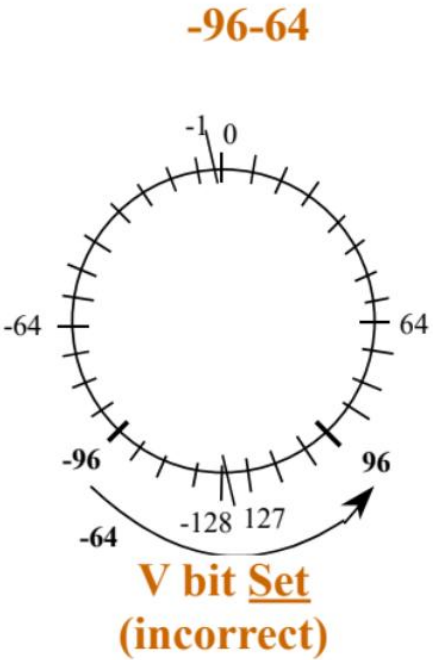


V bit Cleared
(correct)

V bit Set
(incorrect)

Signed subtraction overflow bit





ARM condition codes

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

ARM condition codes

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

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

```
CMP    r3,#0
 ADDNE r0,r1,r2
```

- Flags can be optionally set by using “S” (CMP does not need “S”)

```
loop
  …
  SUBS r1,r1,#1
  BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

Addition instructions

ADD – Add

ADC – Add with carry

```
ADD Rd, Rn, #imm          ; Rd = Rn + imm
ADD Rd, Rd, #imm          ; Rd = Rd + imm
ADD Rd, Rn, Rm            ; Rd = Rn + Rm
ADC Rd, Rn, Rm            ; Rd = Rn + Rm + C
```

Subtraction instructions

SUB - Subtract

SBC – Subtract with carry

RSB – Reverse subtract

RSC - Reverse subtract with carry

```
SUB Rd, Rn, #imm          ; Rd = Rn - imm
SUB Rd, Rn, Rm            ; Rd = Rn – Rm
SBC Rd, Rn, Rm            ; Rd = Rn – Rm + C - 1
RSB Rd, Rn, Rm            ; Rd = Rm - Rn
```

Multiplication instructions

`MUL – Multiply`

`MLA – Multiply and accumulate`

```
MUL Rd, Rm, Rn          ; Rd = Rm * Rn
MLA Rd, Rm, Rn, Rp      ; Rd = Rm * Rn + Rp
```

- Immediate second operands are not supported
- There is no division instruction, instead use repeated subtraction or shift/multiply tricks
    - ARMv7-R introduced signed divide (`SDIV`) and unsigned divide (`UDIV`) instructions

Bitwise logic instructions

AND – Logic AND

ORR – Logic OR

EOR – Logic exclusive OR (XOR)

BIC – Bitwise clear

Remember:
Bitwise operators -> bits
Boolean operators -> words

```
AND Rd, Rd, Rm          ; Rd = Rd & Rm
ORR Rd, Rd, Rm          ; Rd = Rn | Rm
EOR Rd, Rd, Rm          ; Rd = Rn ^ Rm
BIC Rd, Rd, Rm          ; Rd = Rn & (~Rm)
```

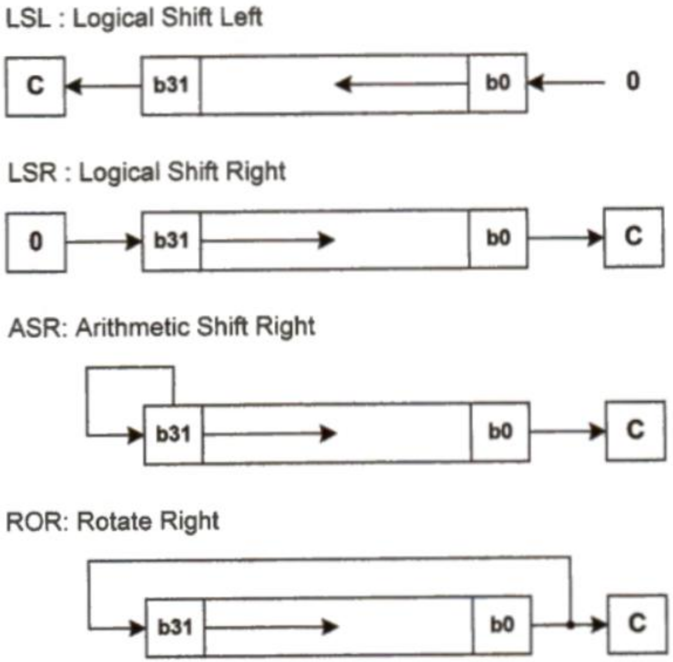
Comparison instructions

- Want to set condition codes without doing an actual operation?
- Try compare or test
- Performs an operation, sets the condition codes, but throws away the result

```
CMP Rn, Rd                    ; Performs SUB, ignores result
CMN Rn, Rd                    ; Performs ADD, ignores result

TST Rn, Rd                    ; Performs AND, ignores result
TEQ Rn, Rd                    ; Performs EOR, ignores result
```

Shift instructions

```
LSL Rd, Rm, #imm5
LSL Rd, Rm, Rs

LSR Rd, Rm, #imm5
LSR Rd, Rm, Rs

ASR Rd, Rm, #imm5
ASR Rd, Rm, Rs

ROR Rd, Rm, Rs
```



Branching instructions

- These instructions change flow of control
- Loops, if statements, switch and case statements can be implemented with branching

```
B - Branch


B target          ; Branch to a label called target
BL target         ; Branch to subroutine called target
                    (returning implemented by restoring the PC from LR)
BX Rm             ; Branch to location specified by Rm
```

Conditional branching instructions

- You can branch based on the condition codes

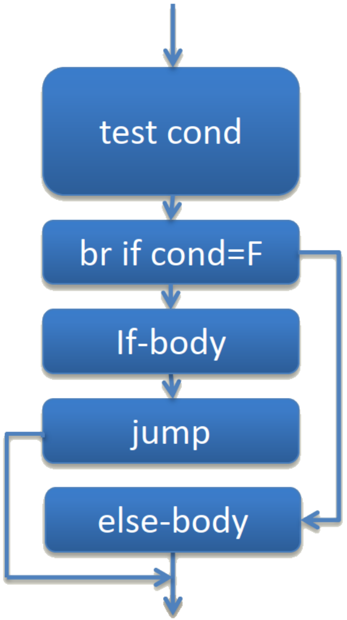
Compare	Signed	Unsigned
==	EQ	EQ
≠	NE	NE
>	GT	HI
≥	GE	HS
<	LT	LO
≤	LE	LS

Compare	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS

If-else statement example #1

```c
if (a < 100) {
      a++;
} else {
      a = -100;
}
```

```
        CMP R0, #100
        BGE else
        ADD R0, R0, #1
        B skip
else:
        MOV R0, #-100
skip:
        // whatever is next
```

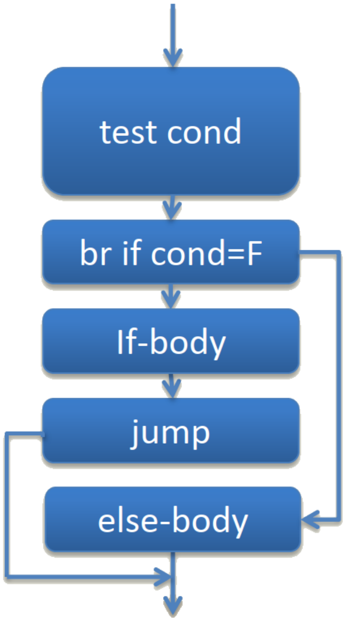


If-else statement example #2

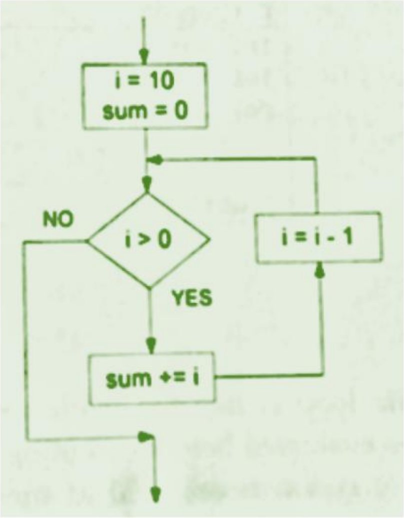
```
int max, a, b;

if (a < b)
    max = b;
else
    max = a;
```

Assume:
a in R0,
b in R1,
max address in R2
(all signed)

```
    CMP R0, R1
    BGE else

    STR R1, [R2, #0]
    B endif

else:
    STR R0, [R2, #0]

endif:
    // whatever is next
```



While loop example

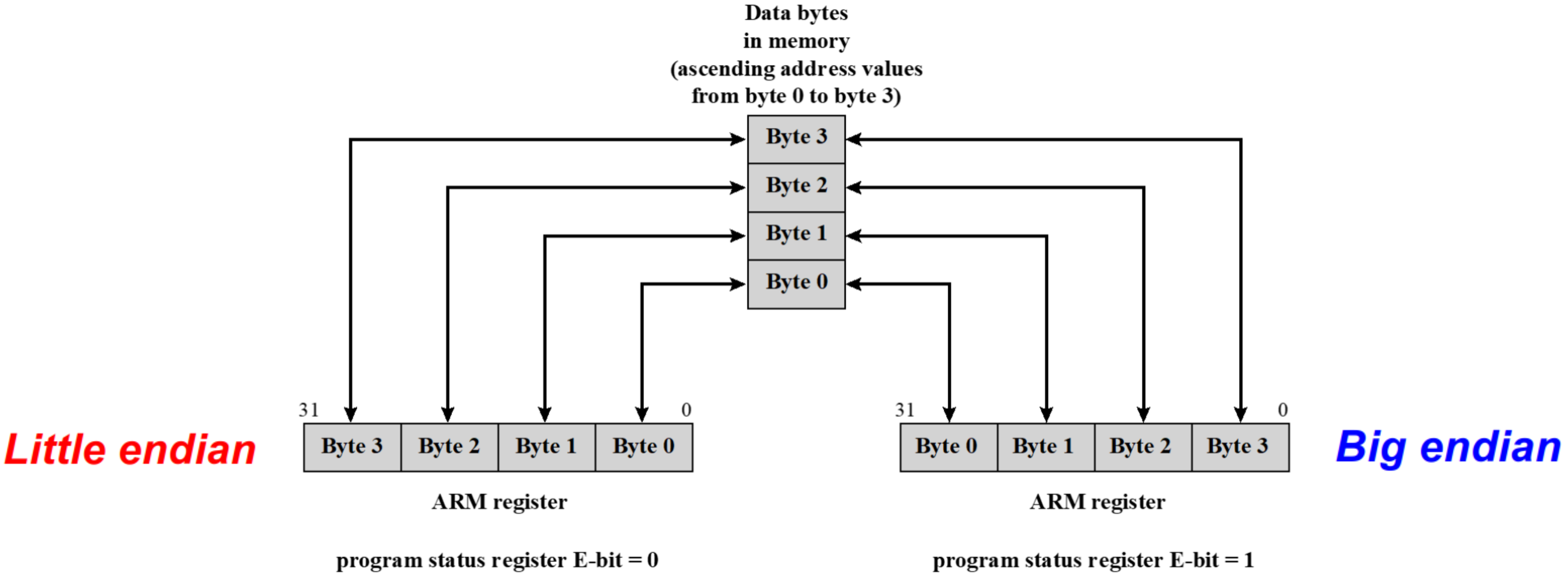



```
C Program
int i = 10;
int sum = 0;

while( i > 0 ){
  sum += i;
  i--;
}
```

```
        ;IMPEMENTING    WHILE-LOOP
        ;WHILE  I>0  --> SUM = SUM+1, I=I-1
        MOV     R0, #5   ;R0=1
        MOV     R1, #0       ;SUM
B_LOOP CMP      R0,#0
        BLE     B_ENDLOOP
        ADD     R1,R1,R0
        SUB     R0,R0,#1
        B     B_LOOP
B_ENDLOOP
```

Endianness

- ARM is little endian by default. It can be set to big endian mode via the CPSR E-bit



GNU Debugger (GDB)

- GNU debugger is a command line debugging tool
- Graphical frontends and extensions are available

```
Breakpoint 1, 0x00008054 in _start ()
---------------------------------------------------------------------------------[ registers ]-
$r0   : 0x00000000
$r1   : 0x00000000
$r2   : 0x00000000
$r3   : 0x00000000
$r4   : 0x00000000
$r5   : 0x00000000
$r6   : 0x00000000
$r7   : 0x00000000
$r8   : 0x00000000
$r9   : 0x00000000
$r10  : 0x00000000
$r11  : 0x00000000
$r12  : 0x00000000
$sp   : 0xbefff850 -> 0x00000001
$lr   : 0x00000000
$pc   : 0x00008054 -> <_start+0> push {r11,  lr}
$cpsr : [thumb fast interrupt overflow carry zero negative]
-------------------------------------------------------------------------------------[ stack ]-
0xbefff850|+0x00: 0x00000001    <-$sp
0xbefff854|+0x04: 0xbefff94e -> "/home/pi/lab/gdb-example"
0xbefff858|+0x08: 0x00000000
0xbefff85c|+0x0c: 0xbefff967 -> "TERM=vt100"
0xbefff860|+0x10: 0xbefff972 -> "SHELL=/bin/bash"
0xbefff864|+0x14: 0xbefff982 -> 0x5f474458
0xbefff868|+0x18: 0xbefff9d1 -> "LC_ALL=en_US.UTF-8"
0xbefff86c|+0x1c: 0xbefff9e4 -> "USER=pi"
------------------------------------------------------------------------------[ code:armv4t ]-
       0x803c                  andeq  r8,  r0,  r0
       0x8040                  andeq  r8,  r0,  r0
       0x8044                  muleq  r0,  r4,  r0
       0x8048                  muleq  r0,  r4,  r0
       0x804c                  andeq  r0,  r0,  r5
       0x8050                  andeq  r8,  r0,  r0
->    0x8054 <_start+0>       push   {r11,  lr}
       0x8058 <_start+4>       add    r11,  sp,  #0
       0x805c <_start+8>       sub    sp,  sp,  #16
       0x8060 <_start+12>      mov    r0,  #1
       0x8064 <_start+16>      mov    r1,  #2
       0x8068 <_start+20>      bl     0x8074 <max>
-----------------------------------------------------------------------------------[ threads ]-
[#0] Id 1, Name: "gdb-example", stopped, reason: BREAKPOINT
-------------------------------------------------------------------------------------[ trace ]-
[#0] 0x8054->Name: _start()
-----------------------------------------------------------------------------------------------
gef>
```

GDB commands

- Start gdb using:
  - gdb <binary>
- Pass initial commands for gdb through a file
  - gdb <binary> –x <ini|ile>
- For help
  - help
- To start running the program
  - run or r <argv>

GDB commands

```
b   main
run
display/10i   $pc
display/x   $r0
display/x   $r1
display/x   $r2
display/x   $r3
display/x   $r4
display/x   $r5
display/x   $r6
display/x   $r7
display/x   $r11
display/32xw   $sp
display/32xw   $cpsr
```

- display/{format string} — prints the expression following the command every time debugger stops
- {format string} include two things:
  - Count — repeat specified number of {size} elements
  - Format — format of how whatever is displayed
- x (hexadecimal), o(octal), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string).
- Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).

GDB examining variables/memory

- Similar to display, to look at contents of memory
- Use "examine" or "x" command
  - x/32xw <memory location> to see memory contents at memory location, showing 32 words
  - x/5s <memory location> to show 5 strings (null terminated) at a particular memory location
  - x/10i <memory location> to show 10 instructions at particular memory location
- The “print” or “p” command evaluates an expression

```
pwndbg> x/10s *((char **)environ)
0x7fffffffe221: "SHELL=/bin/bash"
0x7fffffffe231: "SESSION_MANAGER=local/revbox:@/tmp/.ICE-unix/
0x7fffffffe283: "QT_ACCESSIBILITY=1"
0x7fffffffe296: "COLORTERM=truecolor"
0x7fffffffe2aa: "XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg"
0x7fffffffe2d7: "XDG_MENU_PREFIX=gnome-"
0x7fffffffe2ee: "GNOME_DESKTOP_SESSION_ID=this-is-deprecated"
0x7fffffffe31a: "MY_SHELL=/bin/sh"
0x7fffffffe32b: "GNOME_SHELL_SESSION_MODE=ubuntu"
0x7fffffffe34b: "SSH_AUTH_SOCK=/run/user/1001/keyring/ssh"
```

```
pwndbg> p $ebp
$1 = (void *) 0xffffce68
pwndbg> p &buffer
$2 = (char (*)[38]) 0xffffce3a
pwndbg> p /d 0xffffce68 - 0xffffce3a
$3 = 46
```

GDB breakpoints

- To put breakpoints
  (stop execution on a certain line)
  - b <function name>
  - b *<instruction address>
  - b <filename:line number>
  - b <line number>
- To show breakpoints
  - info b
- To remove breakpoints
  - clear <function name>
  - clear *<instruction address>
  - clear <filename:line number>
  - clear <line number>

GDB stepping

- To step one instruction
  - stepi or si
- To continue till next breakpoint
  - continue or c
- To see backtrace
  - backtrace or bt

Try it for yourself: VisUAL2

- Download VisUAL2 ARM emulator at https://scc416.github.io/Visual2-doc/download
- Or navigate to CPUlator simulator at https://cpulator.01xz.net/?sys=arm
- Write and run some simple programs