BLM4021 Gömülü Sistemler

Doç. Dr. Ali Can KARACA

ackaraca@yildiz.edu.tr

Yıldız Teknik Üniversitesi – Bilgisayar Mühendisliği

Sunum 5 – ARM Komut Setleri ve Assembly Kodları

- ARM komut setleri
- Assembly kodları
- Visual ARM (ARM Assembly Simulator)
- Bazı Örnekler

Ders Materyalleri

Gerekli Kaynaklar:

- Derek Molloy, Exploring Raspberry Pi: Interfacing to the Real World with Embedded Linux, Wiley, 2016.
- M. Wolf, Computers as Components: Principles of Embedded Computing System Design, Elsevier, 2008.

Yardımcı Kaynaklar:

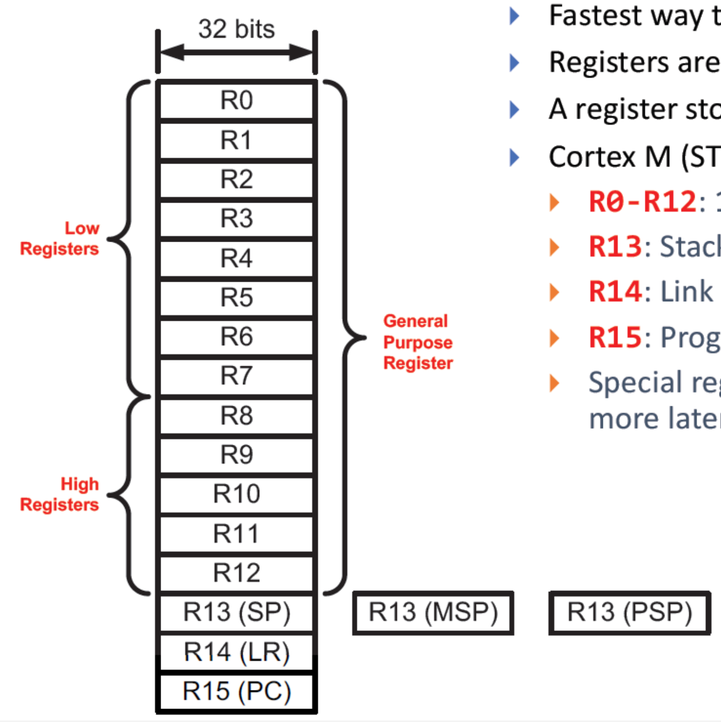
- P. Membrey, D. Hows, Learn Raspberry Pi 2 with Linux and Windows 10, Apress, 2015.
- Ali Saidi, The ARM architecture slide.
- Farid Farahmand, Chapter- 3, Embedded Systems with ARM Cortex-M, 2018.
- O. Urhan, Gömülü Sistem Lisansüstü Ders Notları, 2018.
- V. Weawer, ECE 471 – Embedded Systems Lecture 3, 2020.

Haftalık Konular

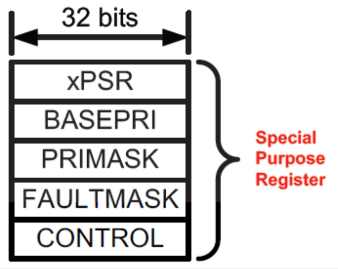
Hafta	Teorik	Laboratuvar
1	Giriş ve Uygulamalar, Mikroişlemci, Mikrodenetleyici ve Gömülü sistem kavramlarının açıklanması	Grupların oluşturulması & Kitlerin Testi
2	Bir Tasarım Örneği, Mikroişlemci, Mikrodenetleyici, DSP, FPGA, ASIC kavramları	Kitlerin gruplara dağıtımı + Raspberry Pi Kurulumu
3	16, 32 ve 64 bitlik mikrodenetleyiciler, pipeline, PIC ve MSP430 özellikleri	Raspberry Pi ile Temel Konfigürasyon
4	ARM tabanlı mikrodenetleyiciler ve özellikleri	---- Resmi Tatil ---
5	ARM Komut setleri ve Assembly Kodları-1	Uygulama 1 – Raspberry Pi ile Buzzer Uygulaması
6	ARM Komut setleri ve Assembly Kodları-2, Raspberry Pi vers. ve GPIO'ları	Uygulama 2 – Raspberry Pi ile Ivme ve Gyro Uygulaması
7	Seri Haberleşme Protokolleri (SPI, I2C ve CAN)	Uygulama 3 – Raspberry Pi ile Motor Kontrol Uygulaması
8	*Vize Sınavı*	
9	Veri toplama; algılayıcı, örnekleme teoremi, ADC, DAC	Proje Soru-Cevap Saati - 1
10	Zamanlayıcı ve kesmeler	Proje Soru-Cevap Saati -2
11	Görüntü, Ses ve Video	Proje kontrolü-1
12	Gerçek Zaman Sistemlerinde temel kavramlar	Proje Kontrolü-2
13	Gerçek zaman İşletim Sistemleri	Mazeret sebepli son proje kontrollerinin yapılması
14	Nesnelerin İnterneti	
15	*Final Sınavı*	

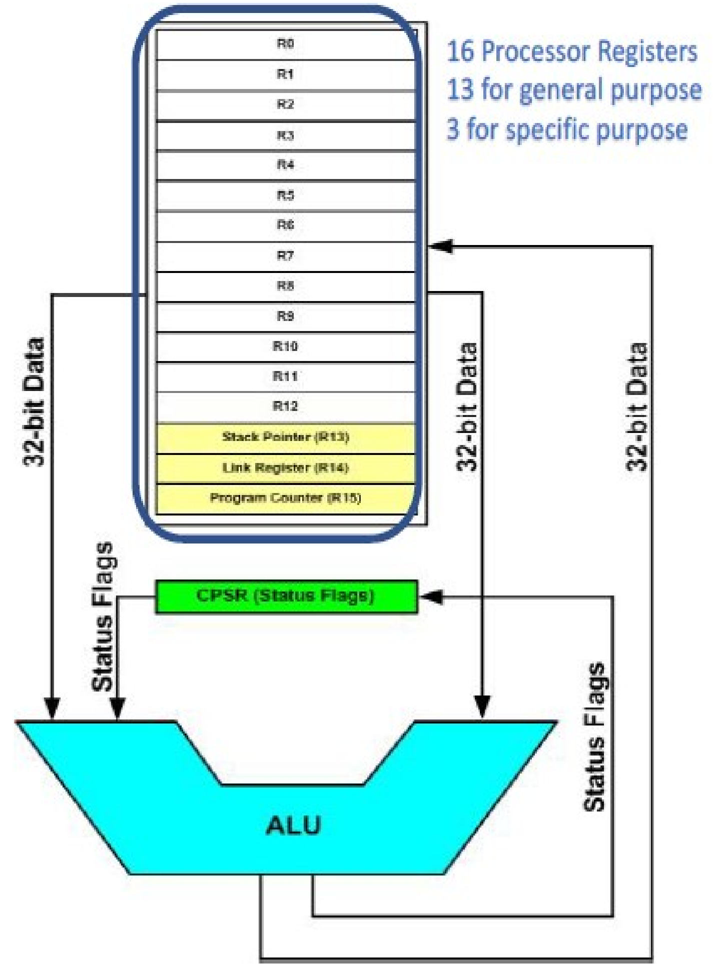
For more details -> Bologna page: http://www.bologna.yildiz.edu.tr/index.php?r=course/view&id=9463&aid=3

Hatırlatma: ARM Kaydedicileri



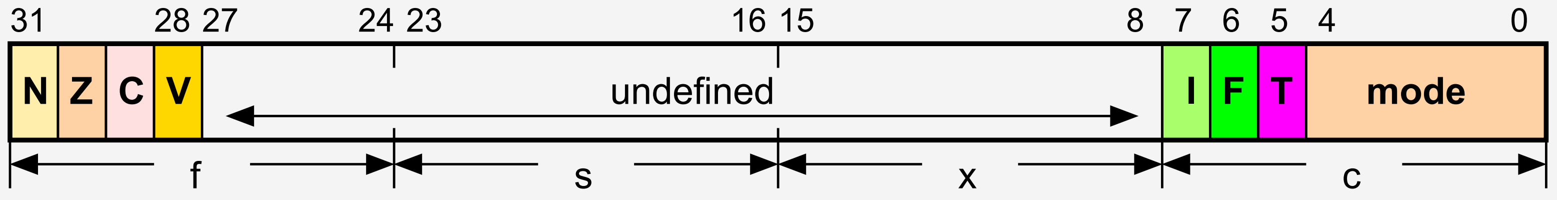
- Fastest way to read and write
- Registers are within the processor chip
- A register stores 32-bit value
- Cortex M (STM32L) has
  - **`R0-R12`**: 13 general-purpose registers
  - **R13**: Stack pointer (Shadow of MSP or PSP)
  - **R14**: Link register (LR)
  - **R15**: Program counter (PC)
  - Special registers (xPSR, BASEPRI, PRIMASK, etc.)- more later





Credit by Farih Farahmand

Hatırlatma: Program Durum Kaydedicileri (CPSR)



- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

Never change T directly (use BX instead)

Changing T in CPSR will lead to unexpected behavior due to pipelining

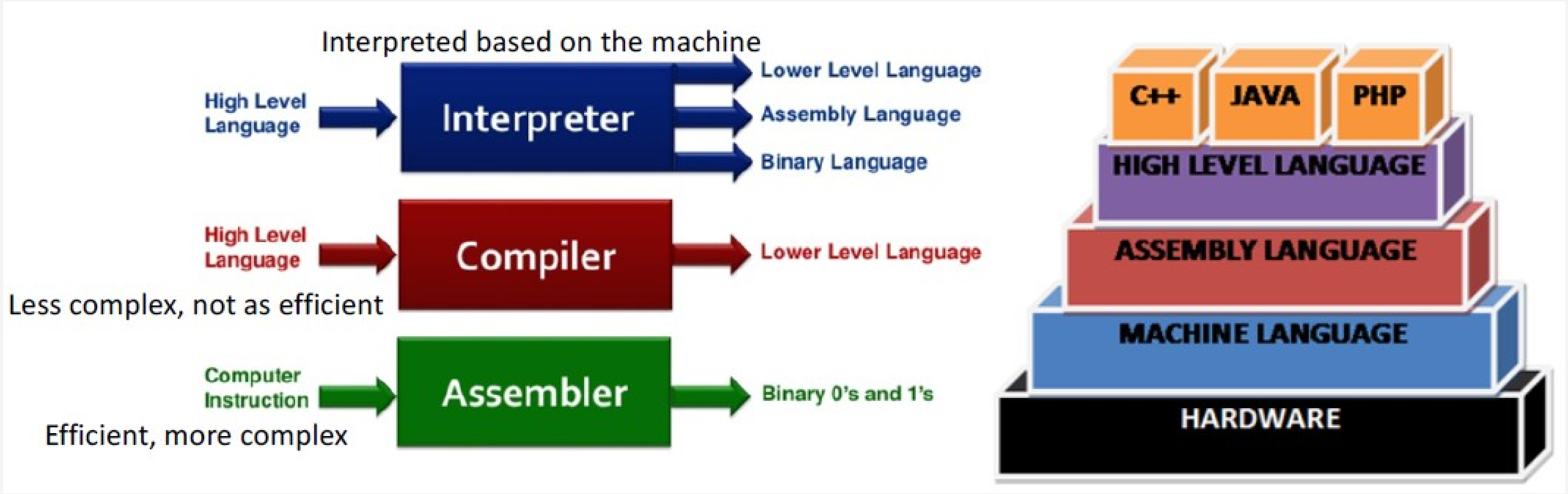
Tip: Don't change undefined bits.

This allows for code compatibility with newer ARM processors

- **Mode bits**

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

Programming Language – Interpreter vs. Compiler



Interpreter: Translates program one statement at a time.

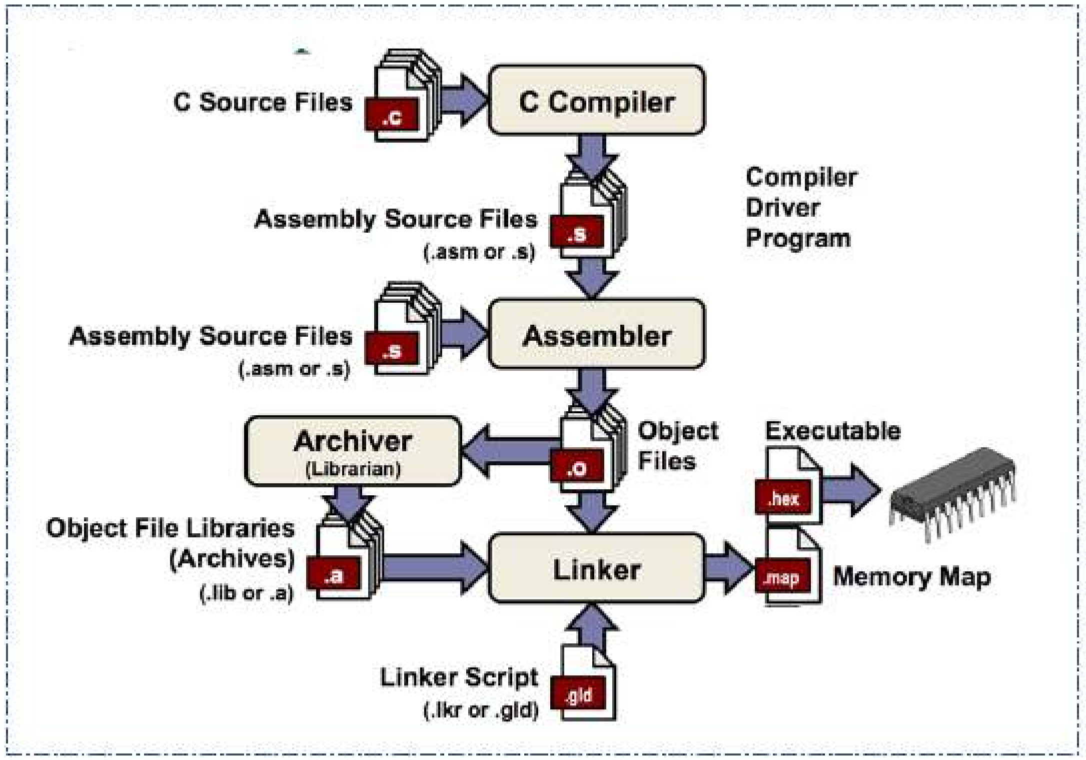
- Continues translating the program until the first error is met.
- Programming language like Python, Ruby.

Compiler: Scan the entire program and translates it a whole into machine code.

- It generates the error message only after scanning the whole program.
- Programming language like C, C++.

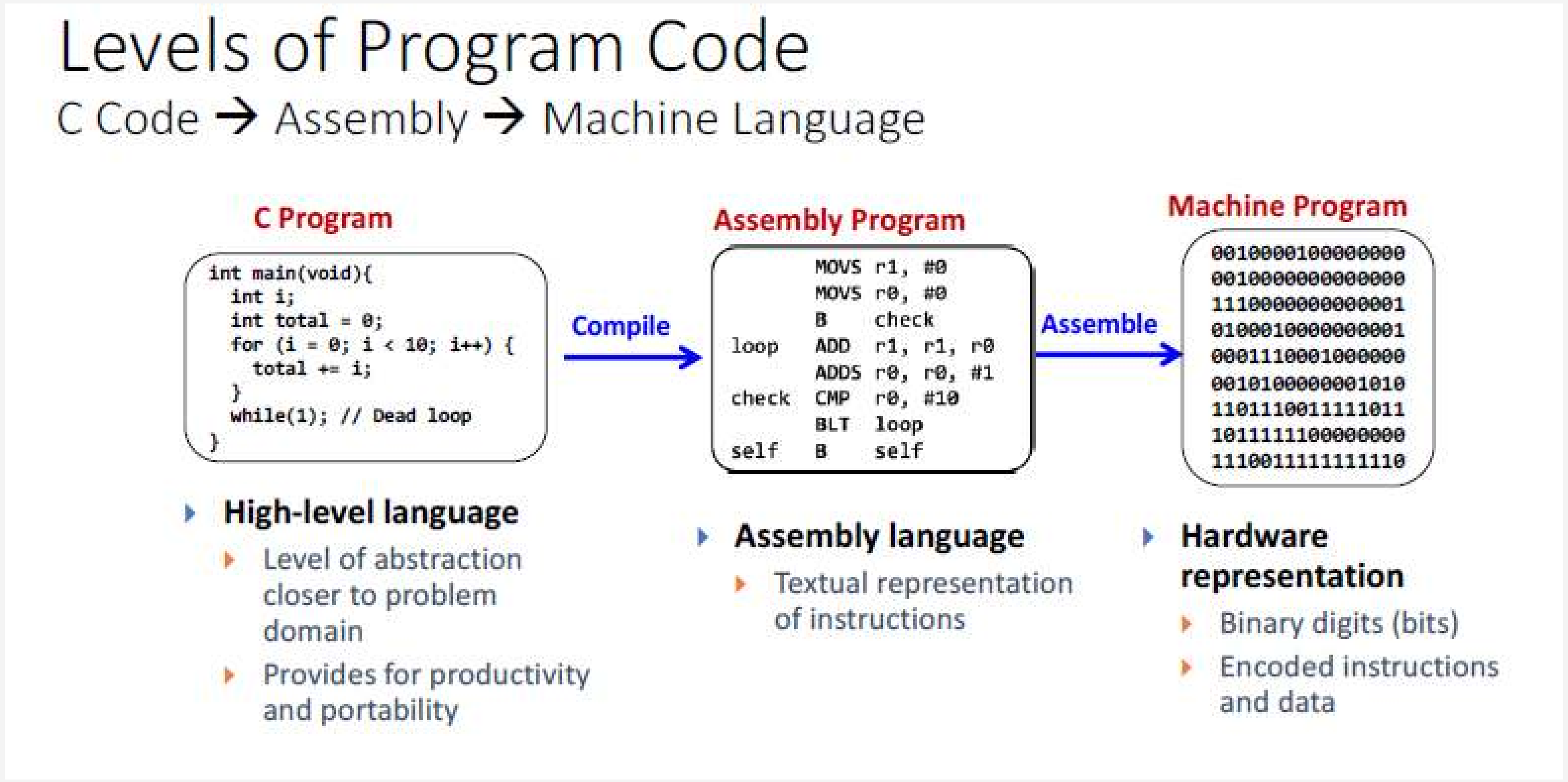
Assemblers and C Compilers

- Compiler (gcc): .c → .s
  - translates high-level language to assembly language
- Assembler (as): .s → .o
  - translates assembly language to machine language
- Archiver (ar): .o → .a
  - collects object files into a single library
- Linker (ld): .o + .a → a.out
  - builds an executable file from a collection of object files
- Execution (execlp)
  - loads an executable file into memory and starts it


C Source Files
.c
C Compiler
Compiler Driver Program
Assembly Source Files (.asm or .s)
.s
Assembly Source Files (.asm or .s)
.s
Assembler
Archiver (Librarian)
.o
Object Files
Executable
.hex
Object File Libraries (Archives) (.lib or .a)
.a
Linker
.map
Memory Map
Linker Script (.lkr or .gld)
.gld

IDE - Integrated Development Environment

Credit by Farid Farahmand



Levels of Program Code

C Code → Assembly → Machine Language

C Program

```
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
```

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

Credit by Farid Farahmand

ARM Assembly

ARM processors have two general purpose instruction sets:

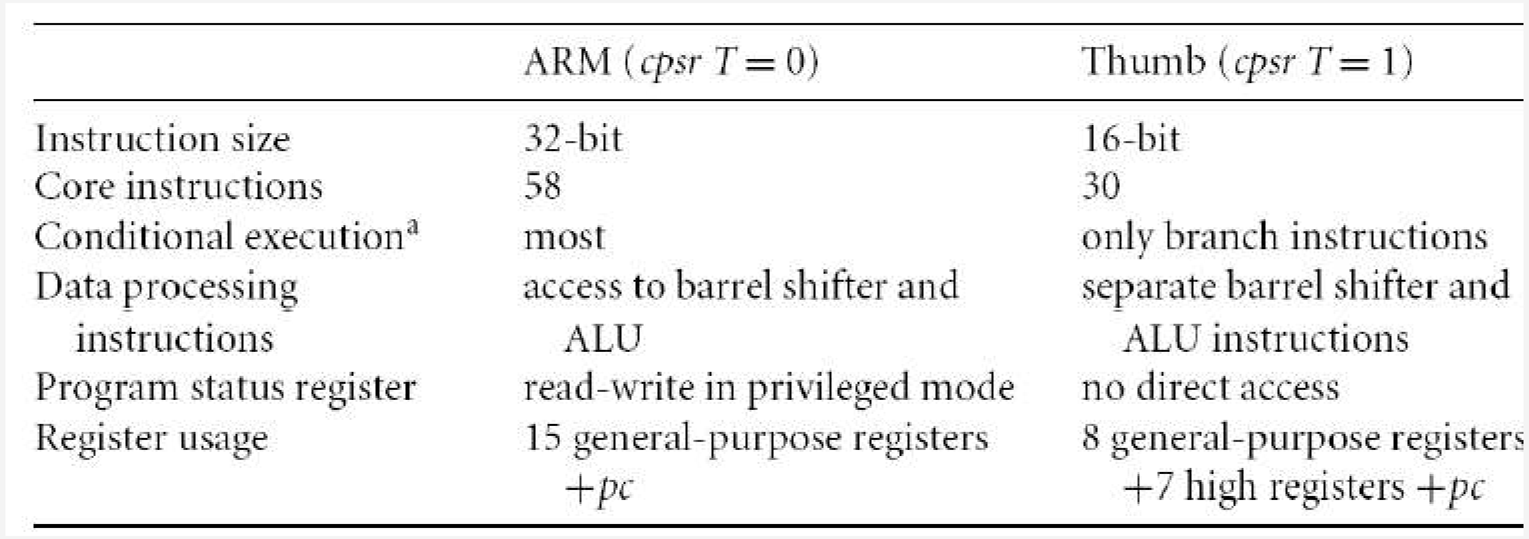
- 32-bit ARM instruction set,
- Space efficient 16-bit Thumb/Thumb-2 instruction set

In newer versions, there are also the following instruction sets:

- Neon 64/128-bit SIMD instruction set,
- VFP vector floating point instruction set

http://www.davespace.co.uk/arm/introduction-to-arm/instruction-sets.html

ARM Instructions & Thumb Instructions

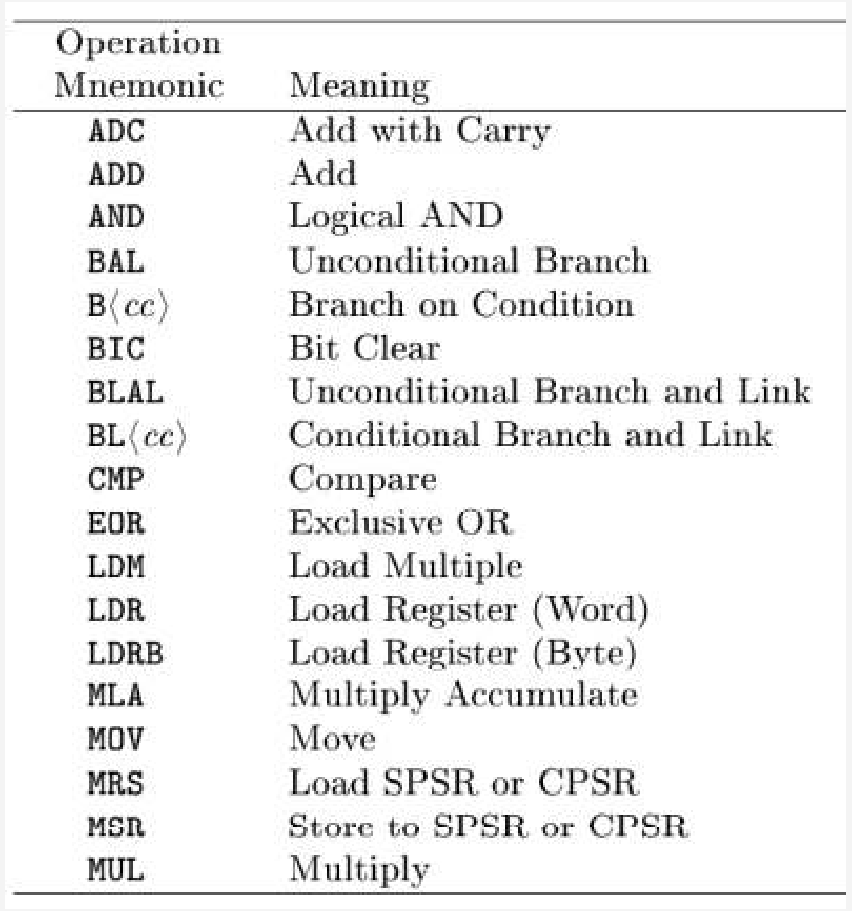


	ARM ($cpsr$ $T=0$)	Thumb ($cpsr$ $T=1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers $+pc$	8 general-purpose registers $+7$ high registers $+pc$

Credit by Yung-Yu Chuang

Assembly Instructions

- Arithmetic & Logic (such as Add, Substract, Multiply, Shift),
- Data Transfer (Load, Move or Store),
- Compare and branch (Branch, If, Test, Compare),
- Other instructions (interrupt, data memory barrier)…

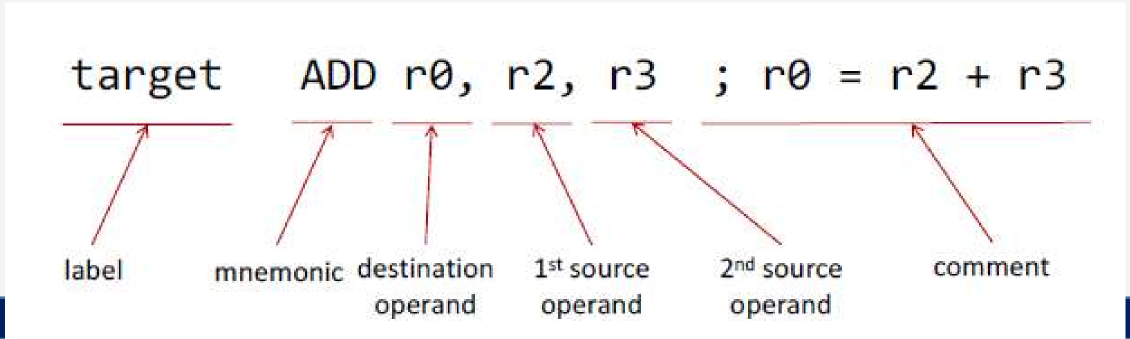


Operation Mnemonic	Meaning
ADC	Add with Carry
ADD	Add
AND	Logical AND
BAL	Unconditional Branch
B$\langle cc \rangle$	Branch on Condition
BIC	Bit Clear
BLAL	Unconditional Branch and Link
BL$\langle cc \rangle$	Conditional Branch and Link
CMP	Compare
EOR	Exclusive OR
LDM	Load Multiple
LDR	Load Register (Word)
LDRB	Load Register (Byte)
MLA	Multiply Accumulate
MOV	Move
MRS	Load SPSR or CPSR
MSR	Store to SPSR or CPSR
MUL	Multiply

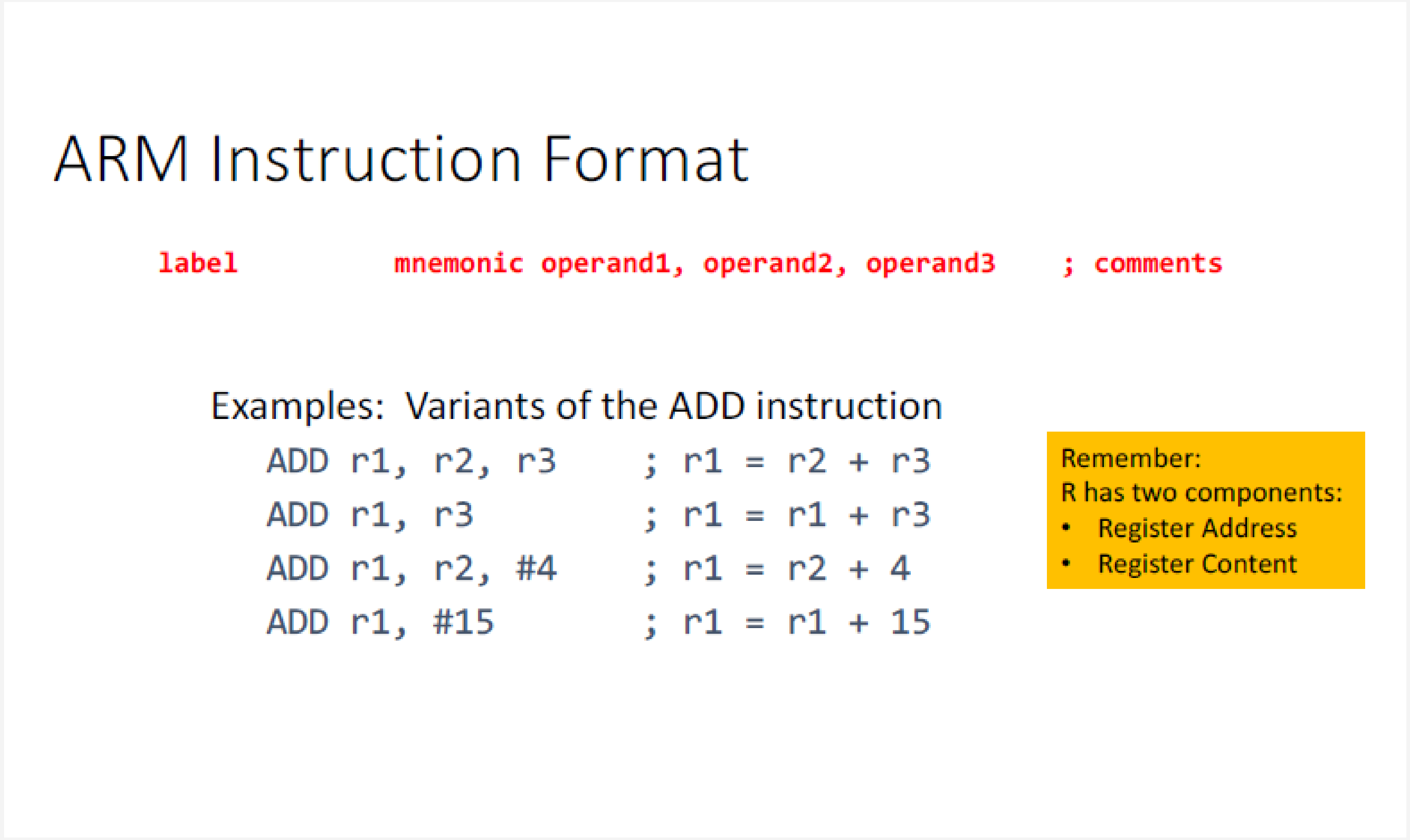
Instruction Syntax in ARM

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

- Label is a reference to the **memory address** of this instruction.
- **Mnemonic** represents the operation to be performed (ADD, SUB, etc.).
- The number of **operands** varies, depending on each specific instruction. Some instructions have no operands at all.
  - Typically, operand1 is the **destination** register, and operand2 and operand3 are source operands.
  - operand2 is usually a register.
  - operand3 may be a register, an immediate number, a register shifted to a constant amount of bits, or a register plus an offset (used for memory access).
- Everything after the semicolon ";" is a comment, which is an annotation explicitly declaring programmers' intentions or assumptions.


```
target   ADD r0, r2, r3  ; r0 = r2 + r3
```

Credit by Farid Farahmand



ARM Instruction Format

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

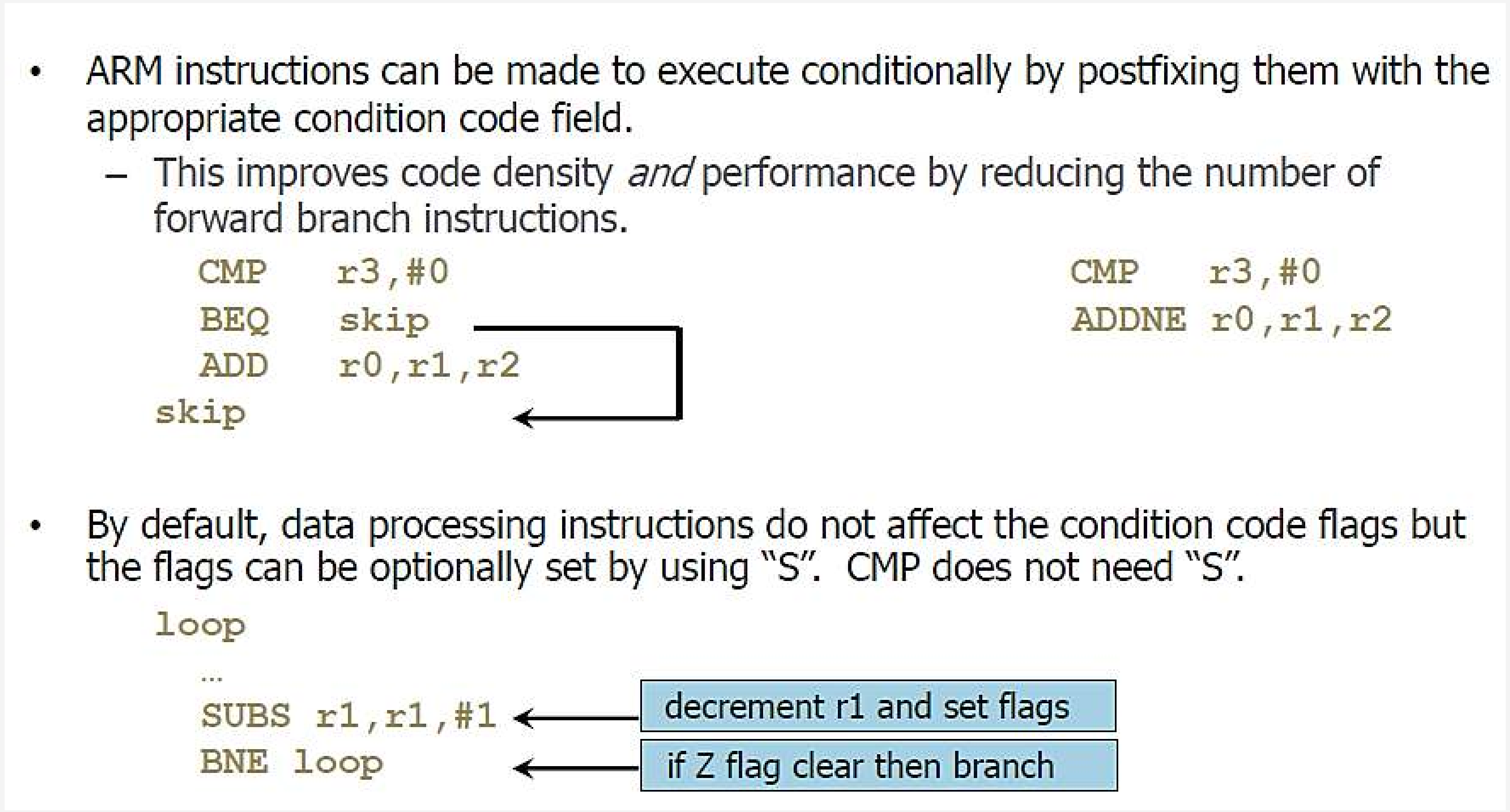
Examples: Variants of the ADD instruction

```
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r3          ; r1 = r1 + r3
ADD r1, r2, #4      ; r1 = r2 + 4
ADD r1, #15         ; r1 = r1 + 15
```

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

Credit by Farid Farahmand

Conditional Execution and Flags



- 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
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

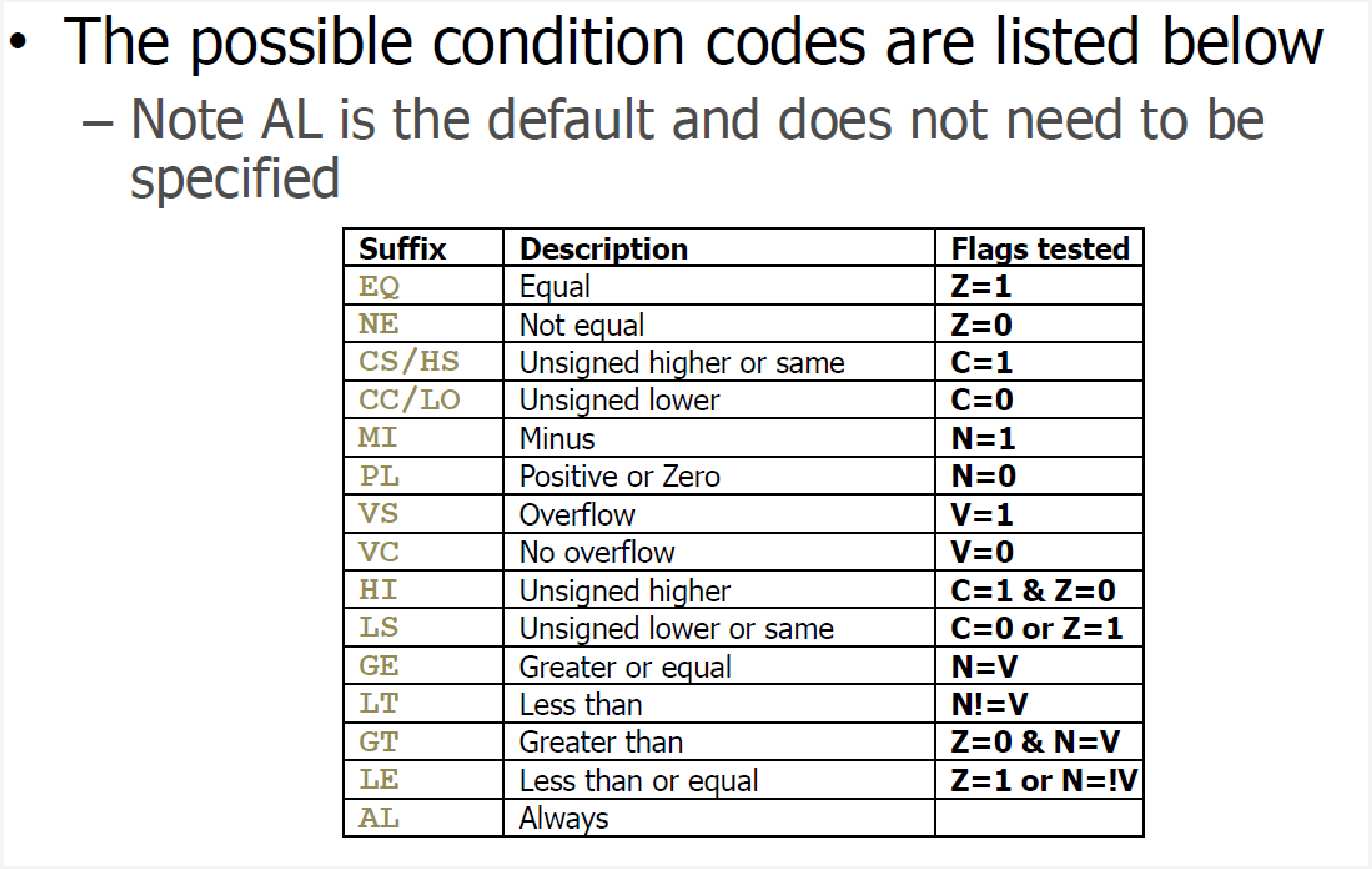
```
loop
    …
    SUBS r1,r1,#1
    BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

Credit by Joseph Zambreno

Conditional Execution and Flags



- The possible condition codes are listed below
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Credit by Joseph Zambreno

Conditional Execution and Flags

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

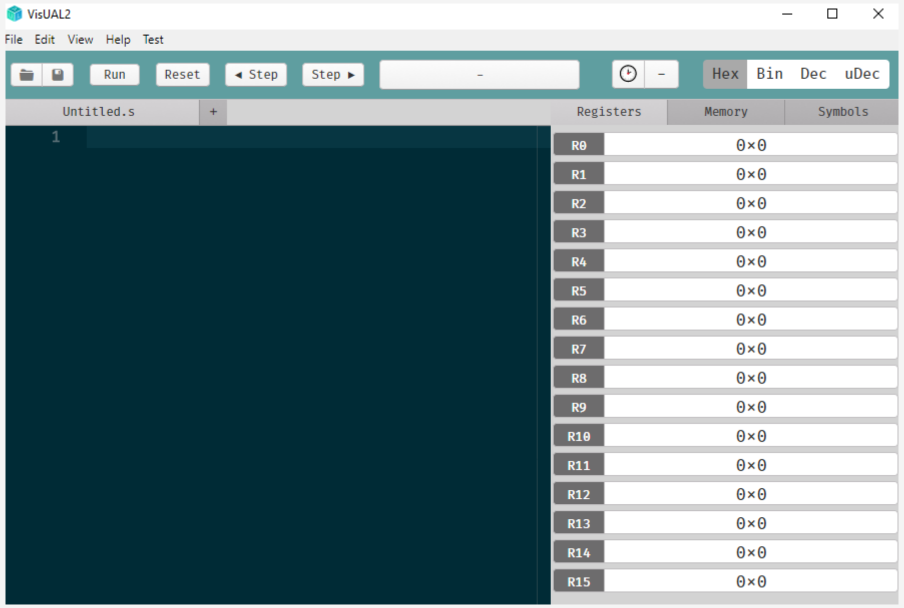
Credit by Joseph Zambreno

Let's Practice…

Visual ARM Emulator

https://salmanarif.bitbucket.io/visual/supported_instructions.html

Download link: https://github.com/tomcl/V2releases



R0	0x0
R1	0x0
R2	0x0
R3	0x0
R4	0x0
R5	0x0
R6	0x0
R7	0x0
R8	0x0
R9	0x0
R10	0x0
R11	0x0
R12	0x0
R13	0x0
R14	0x0
R15	0x0

Department of Electrical and Electronic Engineering of Imperial College London

The Last Example

Example-1:

```
        CMP r0, #0
        BNE else
        ADD r1, r1, #1
        B finish
 else
        ADD r2, r2, #1
 finish
```

Example-2:

```
        CMP r0, #0
        BNE else
        ADDS r1, r1, #1
        B finish
 else
        ADDS r2, r2, #1
 finish
```

Example-3:

```
CMP       r0,#0
MOVEQ     r1,#20
MOVGT     r1,#13
```

```
if (a==0) x=0;
  if (a>0)  x=20;
```

Example-4:

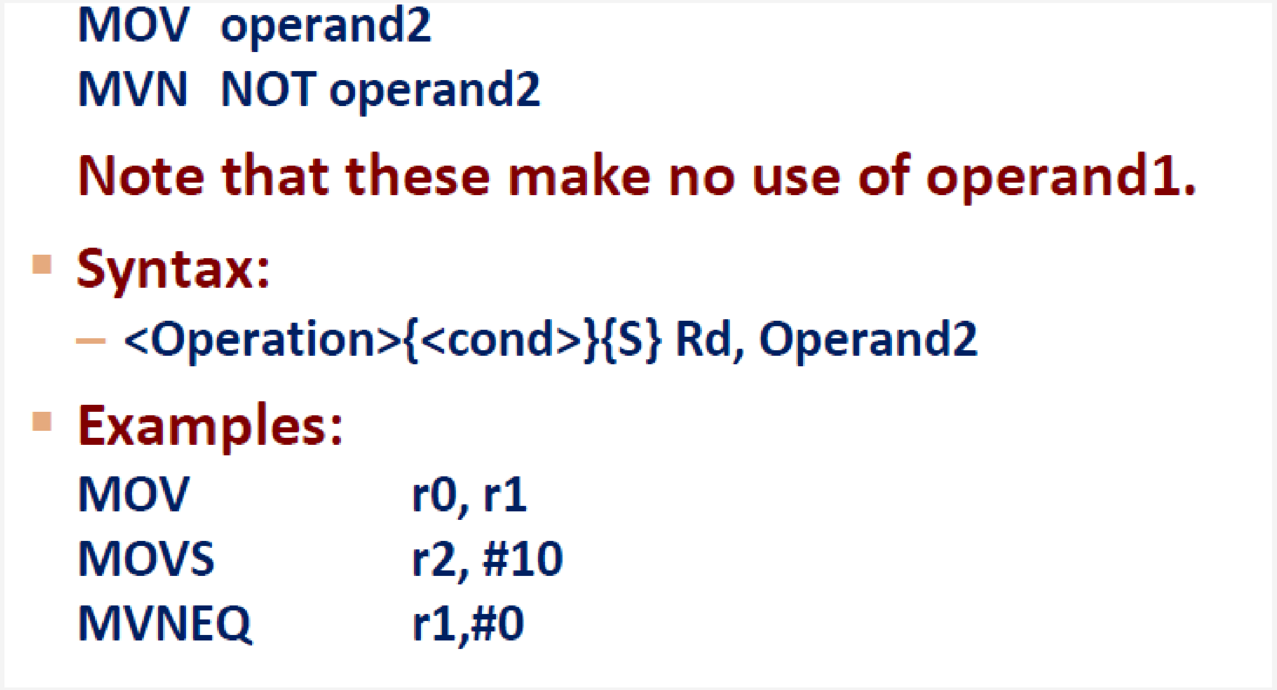
```
MOV r0,#4
CMP r0,#4
CMPNE r0,#10
MOVEQ r1,#2
```

```
if (a==4 || a==10) x=0;
```

All supported instruction set and explanations:

https://salmanarif.bitbucket.io/visual/supported_instructions.html

Instructions: Data Movement



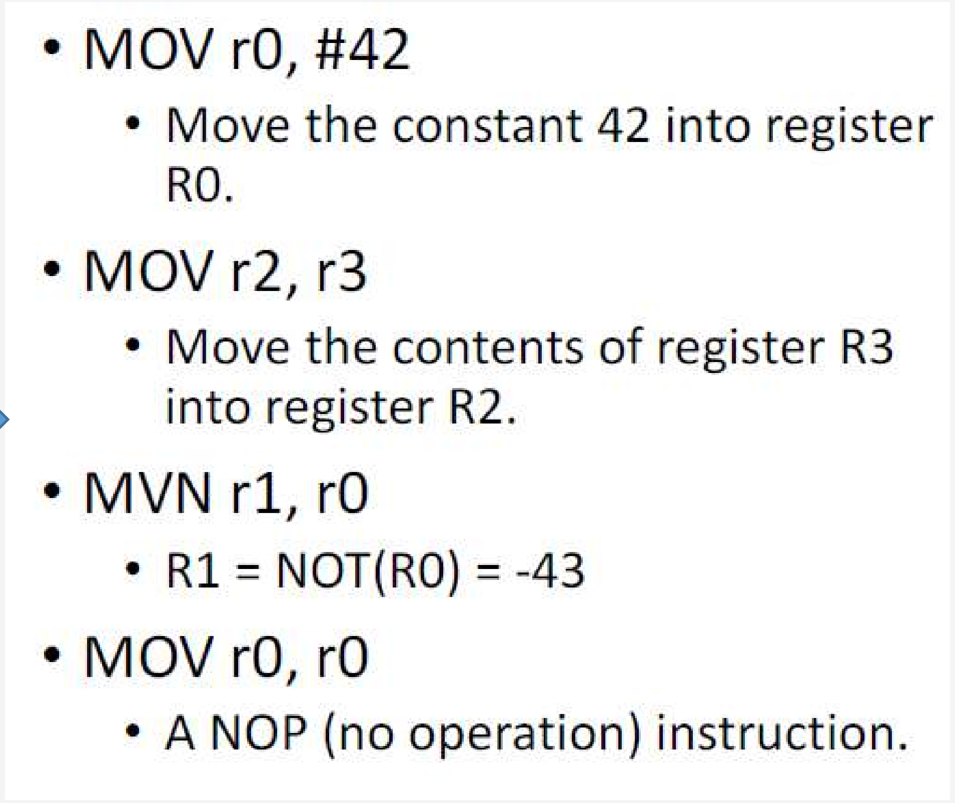
MOV   operand2
MVN   NOT operand2

Note that these make no use of operand1.

- **Syntax:**
  – <Operation>{<cond>}{S} Rd, Operand2
- **Examples:**

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

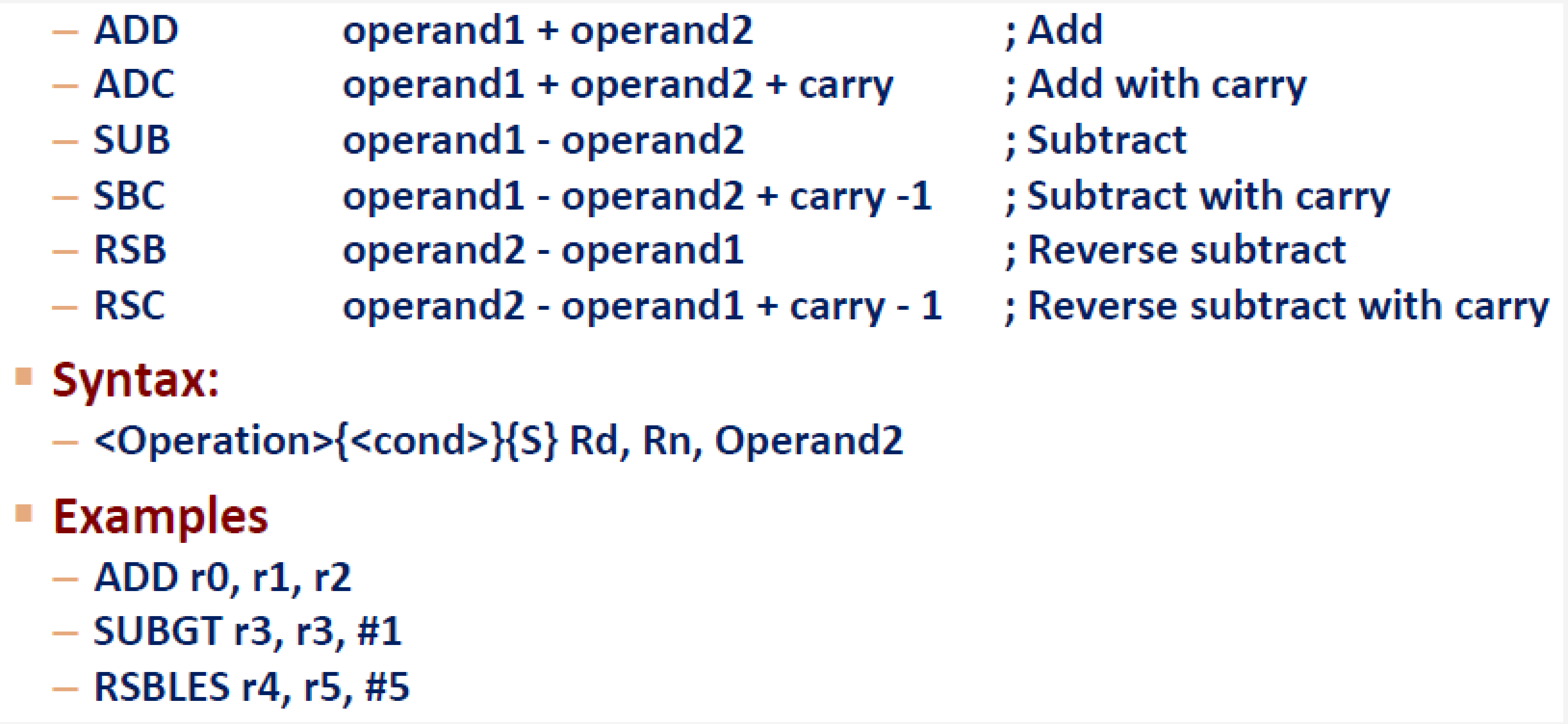
Credit by Yung-Yu Chuang



- MOV r0, #42
  - Move the constant 42 into register R0.
- MOV r2, r3
  - Move the contents of register R3 into register R2.
- MVN r1, r0
  - R1 = NOT(R0) = -43
- MOV r0, r0
  - A NOP (no operation) instruction.

Credit by Farid Farahmand

Instructions: Arithmetic Operations



– ADD	operand1 + operand2	; Add
– ADC	operand1 + operand2 + carry	; Add with carry
– SUB	operand1 - operand2	; Subtract
– SBC	operand1 - operand2 + carry -1	; Subtract with carry
– RSB	operand2 - operand1	; Reverse subtract
– RSC	operand2 - operand1 + carry - 1	; Reverse subtract with carry

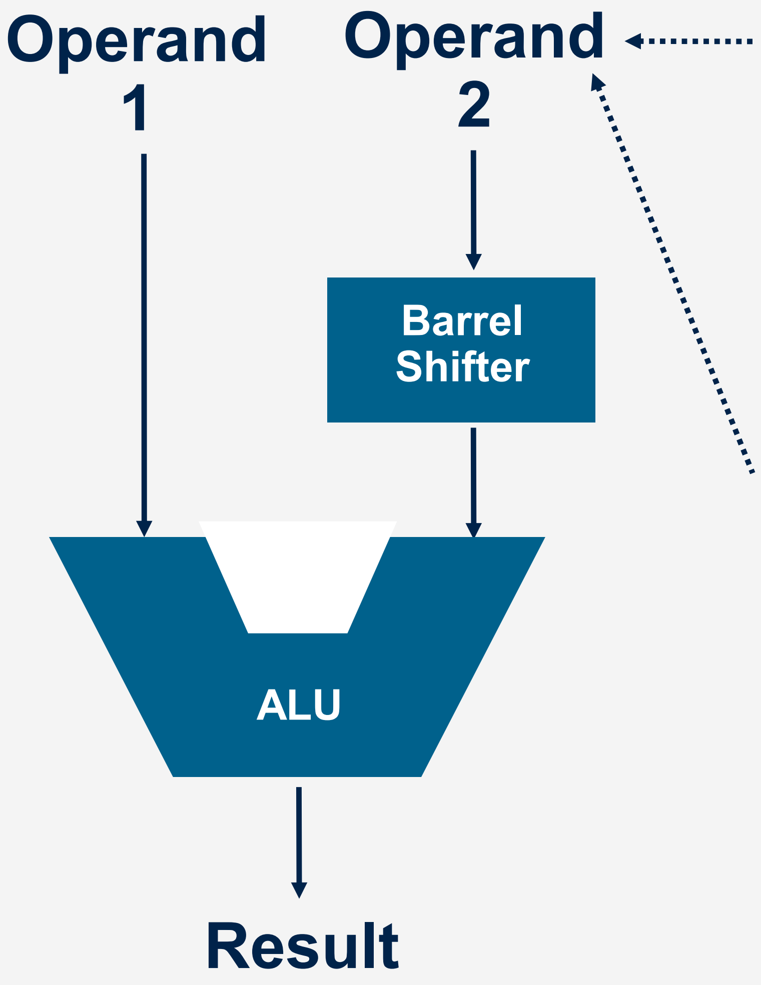
- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- **Examples**
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Credit by Yung-Yu Chuang

- ADD r0, r1, r2
  - R0 = R1 + R2
- SUB r5, r3, #10
  - R5 = R3 – 10
- RSB r2, r5, #0xFF00
  - R2 = 0xFF00 – R5

Credit by Farid Farahmand

Instructions: Shifting & Adding Operations (Barrel Shifter)



Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by a power of 2

Example: ADD R1, R2, R3, LSL #2

(R2 + R3*4) -> R1

```
MOV r3, #2
MOV r2, #5
ADD r1, r2, r3, lsl #2
```

Credit by Farid Farahmand

Instructions: Barrel Shifter

- Using a multiplication instruction to multiply by a constant means first loading the constant into a register and then waiting a number of internal cycles for the instruction to complete.
- A more optimum solution can often be found by using some combination of MOVs, ADDs, SUBs and RSBs with shifts.
  - Multiplications by a constant equal to a ((power of 2) ± 1) can be done in one cycle.

```
MOV R2, R0, LSL #2        ; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD R9, R5, R5, LSL #3    ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB R9, R5, R5, LSL #3    ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB R10, R9, R8, LSR #4   ; R10 = R9 - R8 / 16
MOV R12, R4, ROR R3       ; R12 = R4 rotated right by value of R3
```

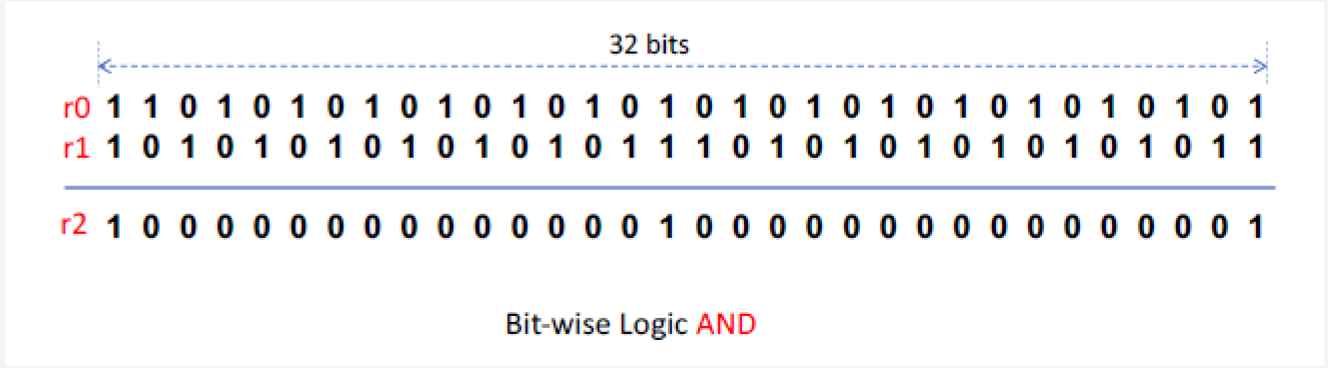
Credit by Mark McDermott

Instructions: Logical Operations

AND	operand1 AND operand2
EOR	operand1 EOR operand2
ORR	operand1 OR operand2
ORN	operand1 NOR operand2
BIC	operand1 AND NOT operand2 [ie bit clear]

- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- **Examples:**

```
AND   r0, r1, r2
BICEQ r2, r3, #7
EORS  r1,r3,r0
```



```
              32 bits
r0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
r1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
r2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Bit-wise Logic AND

- AND r8, r7, r2
  - R8 = R7 & R2
- ORR r11, r11, #1
  - R11 |= 1
- BIC r11, r11, #1
  - R11 &= ~1
- EOR r11, r11, #1
  - R11 ^= 1

Credit by Farid Farahmand

Instructions: Comparison Operations

- The only effect of the comparisons is to update the condition flags. Thus no need to set S bit.
- Operations are:

– CMP	operand1 - operand2	; Compare
– CMN	operand1 + operand2	; Compare negative
– TST	operand1 AND operand2	; Test
– TEQ	operand1 EOR operand2	; Test equivalence

- Syntax:
  - <Operation>{<cond>} Rn, Operand2
- Examples:

– CMP	r0, r1
– TSTEQ	r2, #5

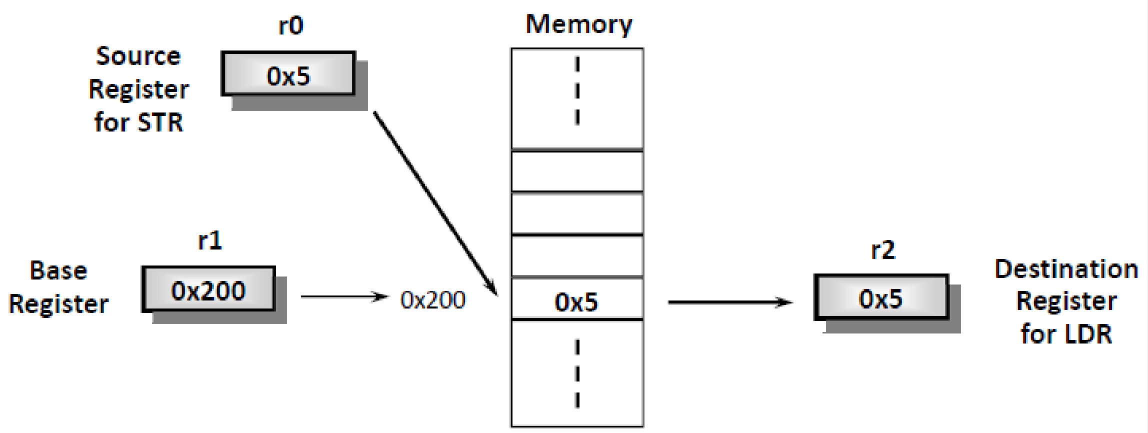
- CMP r0, #42
  - Compare R0 to 42.
- CMN r2, #42
  - Compare R2 to -42.
- TST r11, #1
  - Test bit zero.
- TEQ r8, r9
  - Test R8 equals R9.
- SUBS r1, r0, #42
  - Compare R0 to 42, with result.

Credit by Mark McDermott

Instructions: Single register data transfer

- **The basic load and store instructions are:**
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- **Syntax:**
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

```
STR r0, [r1]        ; Store contents of r0 to location pointed to
                    ; by contents of r1.
LDR r2, [r1]        ; Load r2 with contents of memory location
                    ; pointed to by contents of r1.
```



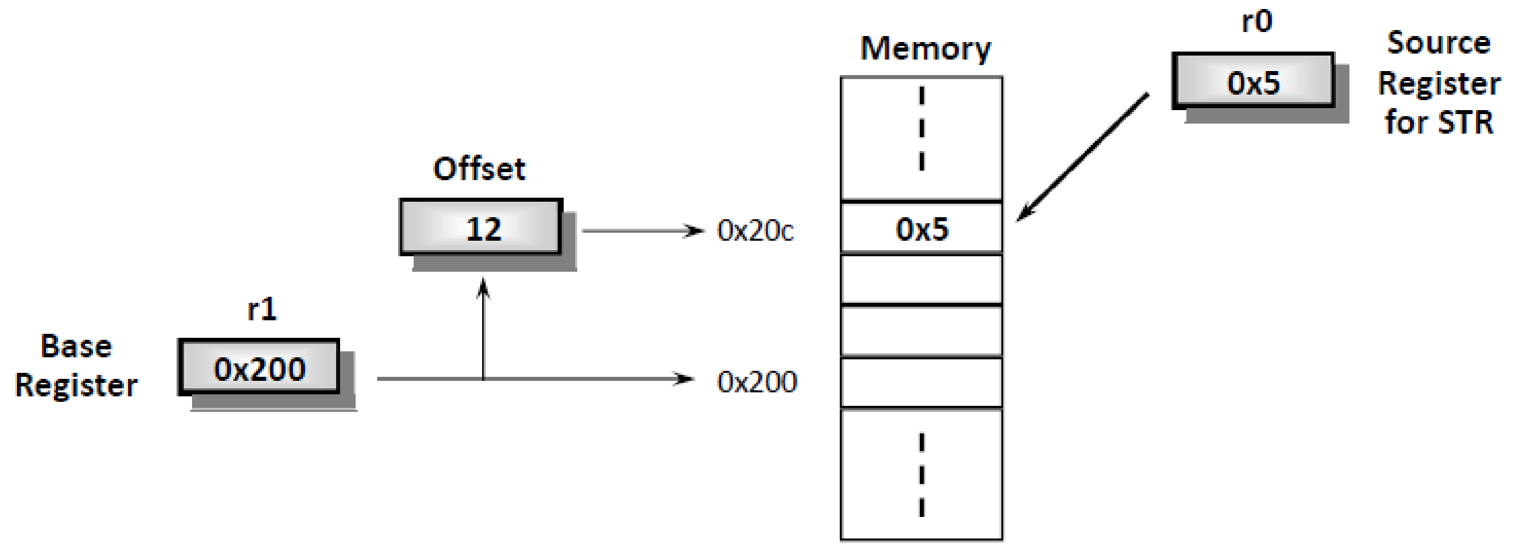
MOV r1,#0x200
MOV r0,#5
STR r0,[r1]

Credit by Mark McDermott

LDR: Pre-indexed Adressing

Credit by Mark McDermott

- **The basic load and store instructions are:**
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB
- **Example: STR r0, [r1,#12]**



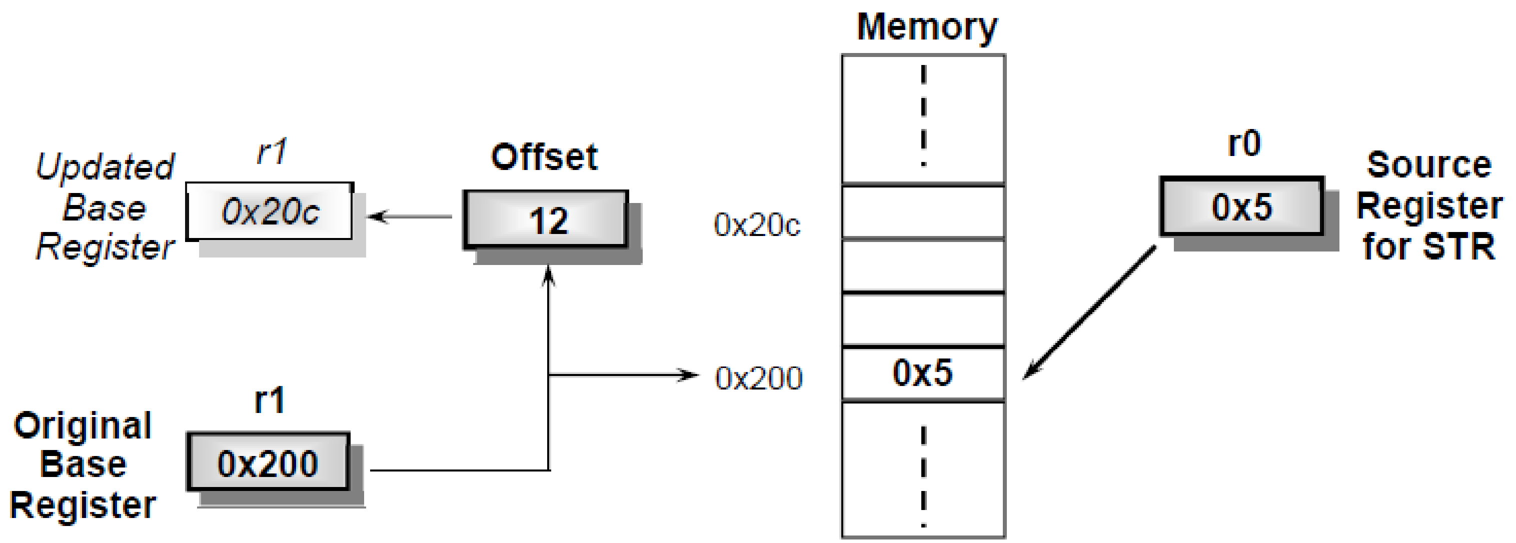
  - To store to location 0x1f4 instead use: STR r0, [r1,#-12]
  - To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]!
  - If r2 contains 3, access 0x20c by multiplying this by 4:
    - STR r0, [r1, r2, LSL #2]

LDR: Post-indexed Adressing

Credit by Mark McDermott

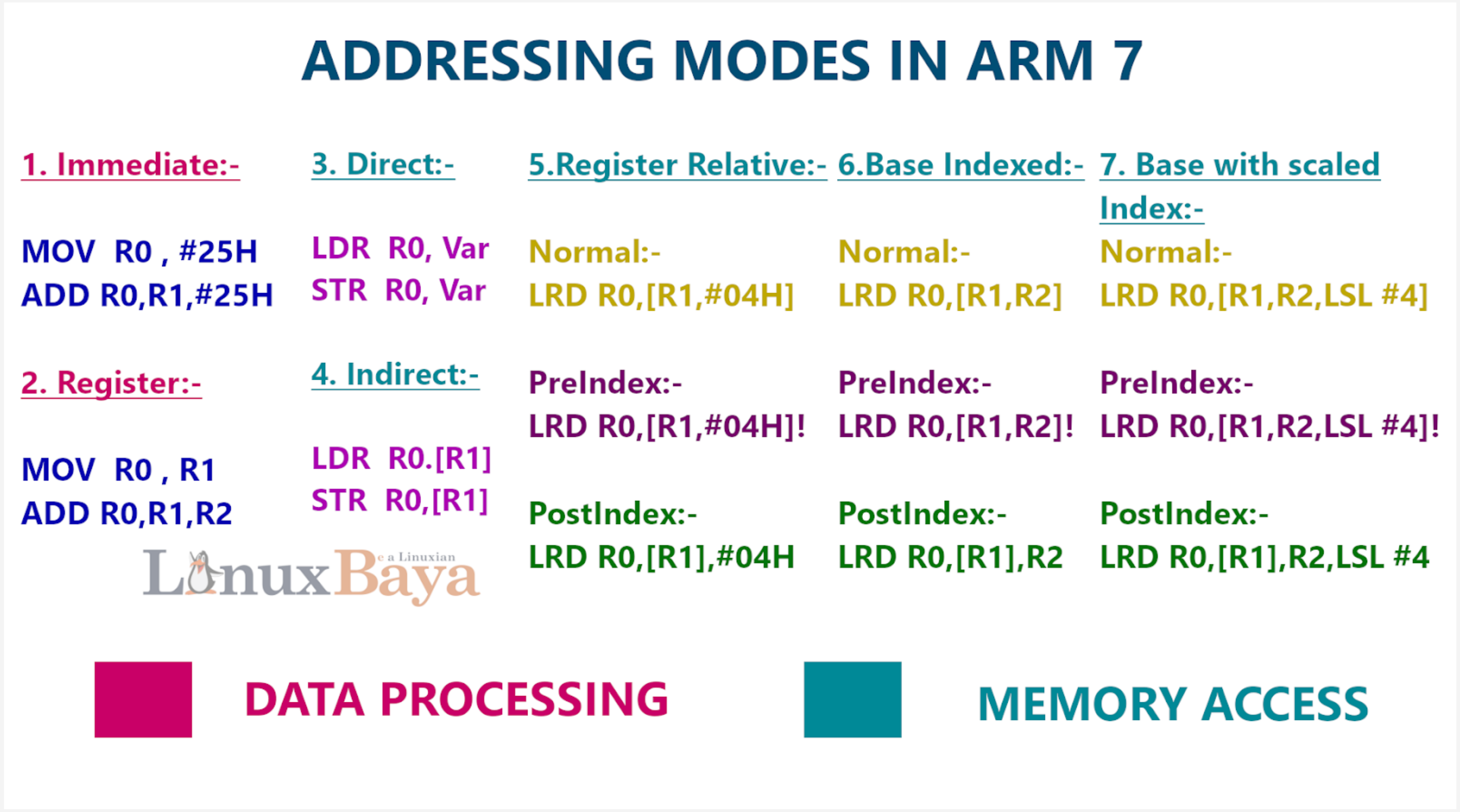
- **The basic load and store instructions are:**
  - Load and Store Word or Byte
    - LDR / STR / LDRB / STRB

- **Example: STR r0, [r1], #12**



  - To auto-increment the base register to location 0x1f4 instead use:
    - STR r0, [r1], #-12
  - If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:
    - STR r0, [r1], r2, LSL #2

```
MOV r2,#0x200
MOV r3,#10
MOV r4,#20
STR r3,[r2],#4
STR r4,[r2]
```



ADDRESSING MODES IN ARM 7

1. Immediate:-

MOV R0 , #25H
ADD R0,R1,#25H

2. Register:-

MOV R0 , R1
ADD R0,R1,R2

3. Direct:-

LDR R0, Var
STR R0, Var

4. Indirect:-

LDR R0.[R1]
STR R0,[R1]

5.Register Relative:-

Normal:-
LRD R0,[R1,#04H]

PreIndex:-
LRD R0,[R1,#04H]!

PostIndex:-
LRD R0,[R1],#04H

6.Base Indexed:-

Normal:-
LRD R0,[R1,R2]

PreIndex:-
LRD R0,[R1,R2]!

PostIndex:-
LRD R0,[R1],R2

7. Base with scaled Index:-

Normal:-
LRD R0,[R1,R2,LSL #4]

PreIndex:-
LRD R0,[R1,R2,LSL #4]!

PostIndex:-
LRD R0,[R1],R2,LSL #4

🟥 DATA PROCESSING 🟦 MEMORY ACCESS

Example: C Assignment

- **C:**

```
z = (a << 2) |  (b & 15);
```

- **Assembler:**

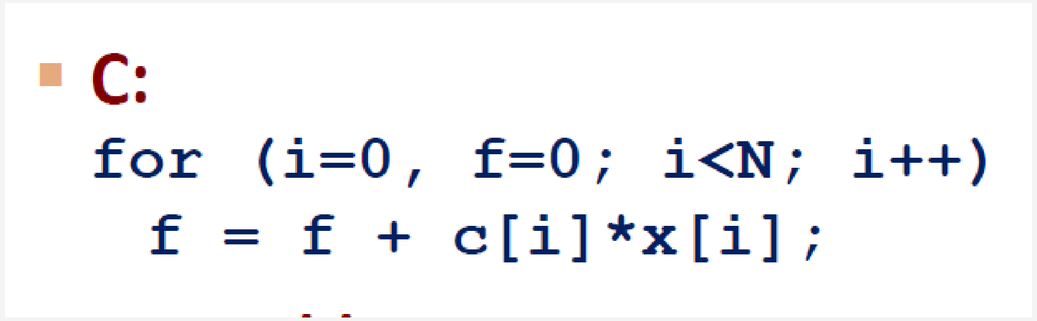
```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
ADR r4,z ; get address for z
STR r1,[r4] ; store value for z
```

DCD: Defined Word value storage area

```
aVal DCD 8
bVal DCD 0

        ADR r4, aVal
        LDR r0, [r4]
        MOV r0, r0, lsl #2
```

FIR Filter



- **C:**

```
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];
```

????