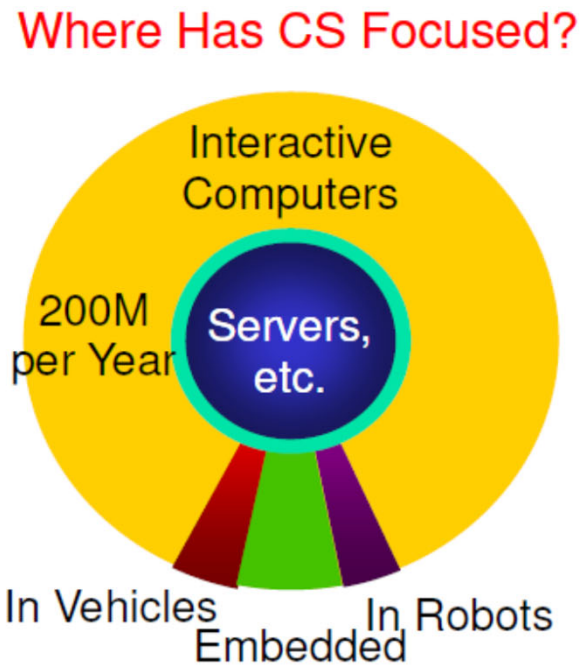
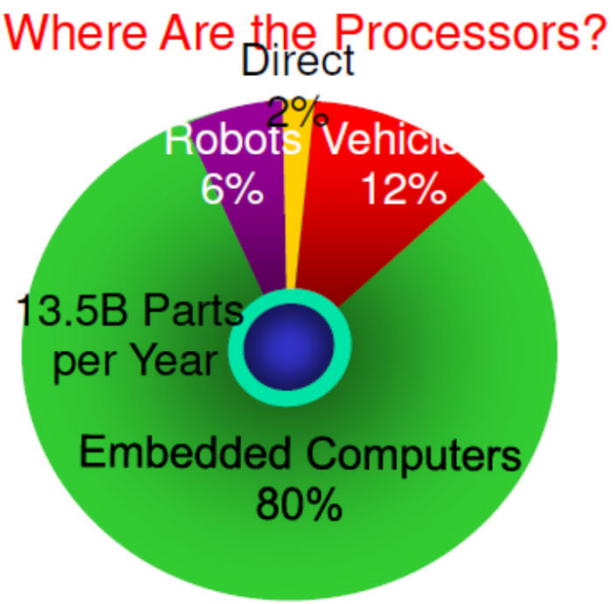
Embedded Systems: Introduction

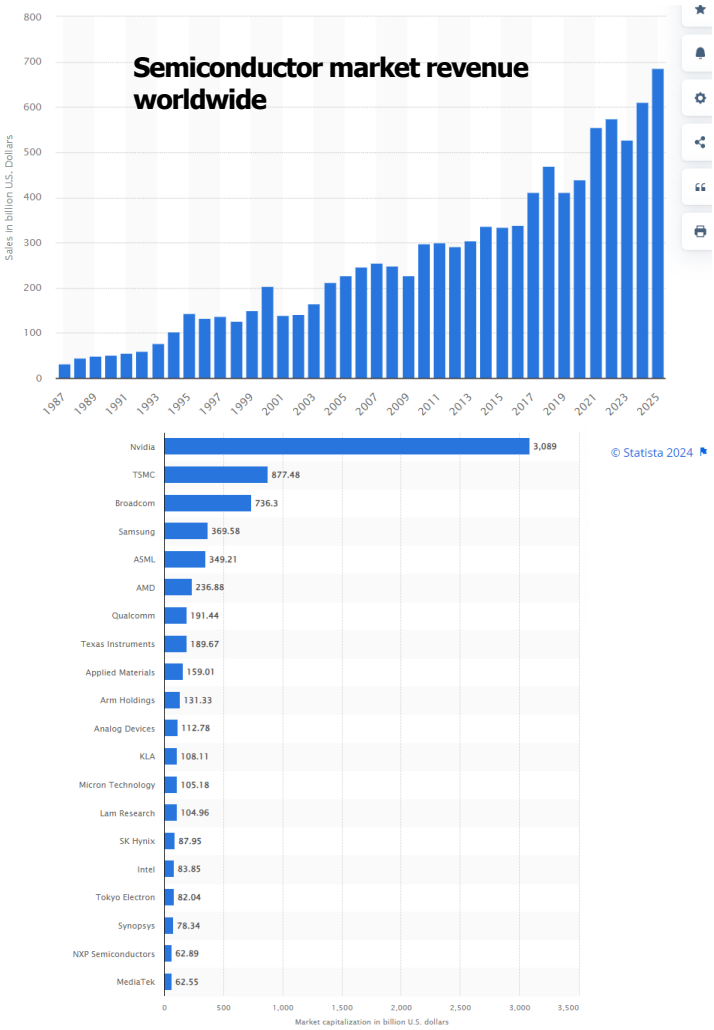
Edited: Waqas Majeed

Why Embedded Systems?

Estimated 98% of 20 Billion CPUs produced in 2015 used for embedded apps





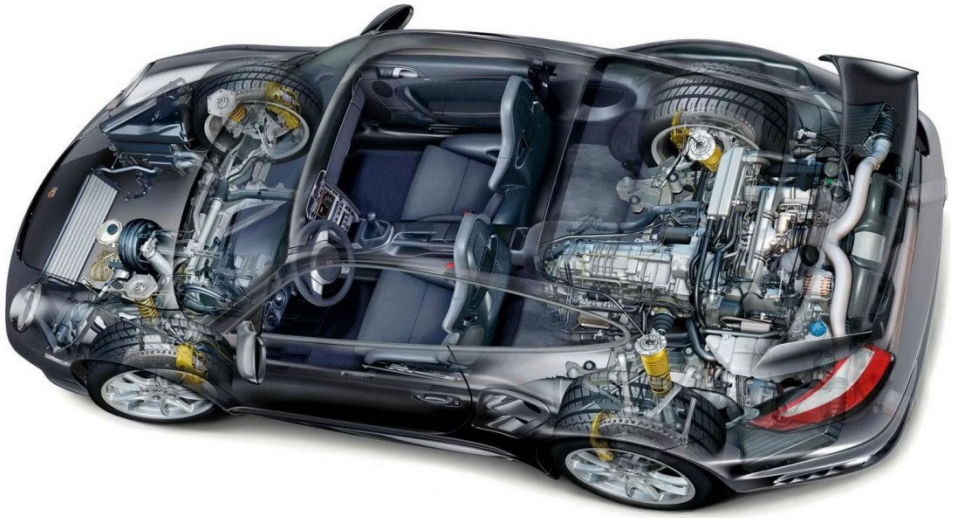


Aug 2024, Statista

Look for the CPUs…the Opportunities Will Follow!

Source: DARPA/Intel (Tennenhouse)

Embedded Systems



Computers Comparison

Embedded Systems

- Special purpose HW and embedded OS.
- Few applications that are known at design-time.
- Not programmable by end user.
- Fixed run-time requirements (additional computing power not useful).
- e.g. MCU, PLC, …
- Package
- Criteria:
  - cost
  - power consumption
  - Predictability (speed)
  - Performance (Constrained)

General Purpose Computers

- Generic HW and general purpose OS
- Broad class of applications.
- Programmable by end user.
- Faster is better.
- Criteria:
  - cost
  - average speed
- Examples: PCs, Workstations, Servers
- Modular

ASIC uC FPGA
Speed: High
Cost (prototyping): High
Cost (mass production): Low
Power Consumption: Low
Customizability: Low
Usability: Hard

Embedded Systems Comparison

ASIC

- Large production volumes

Microcontroller (MCU)

- CPU with RAM, ROM, I/O ports and other peripheral devices integrated as one, to perform specific tasks.
- Assembly, GP Programming Languages

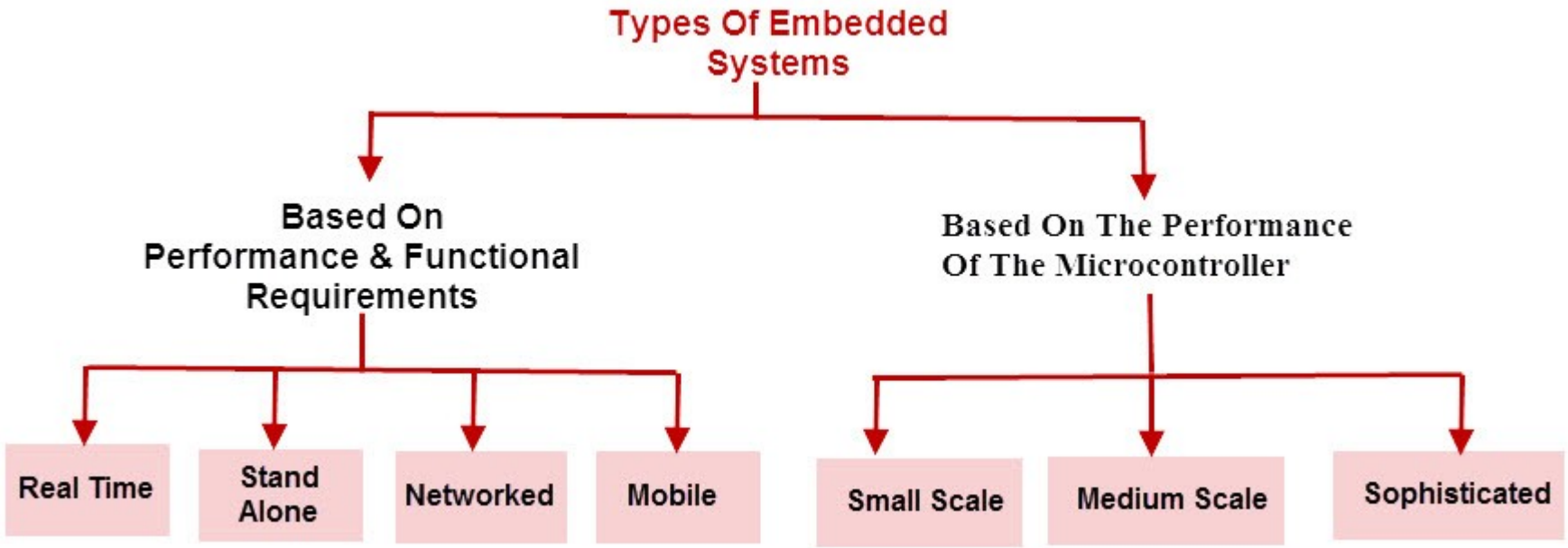
Field Programmable Gate Array (FPGA)

- GP IC for embedded applications
- Co-designing SW and HW
- An integrated circuit that contains millions of logic gates that can be electrically re-configured, through programmable interconnects to perform the required task.
- HDL (VHDL, Verilog) Programming
- Small scale use, prototyping

SoC

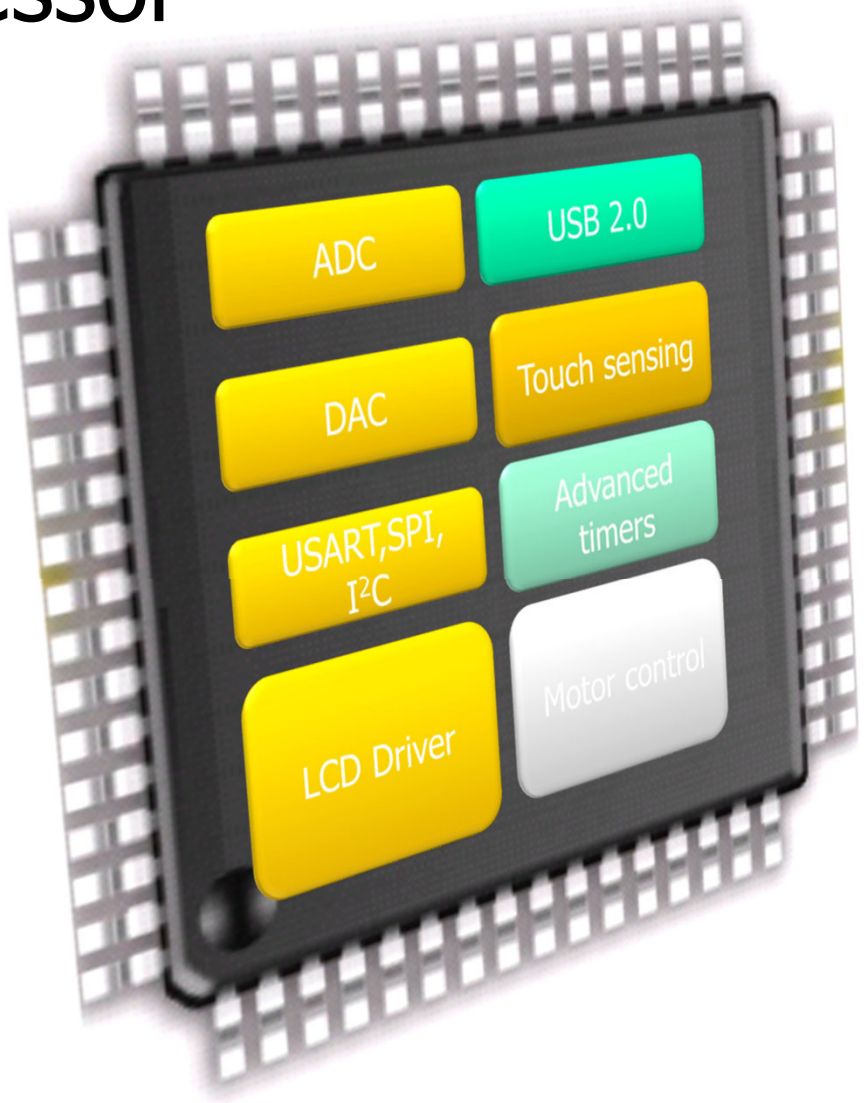
- CPU+FPU+iGPU+Cache/s all in one
- e.g. Raspberry Pi SBC:
  - DRAM+SDC Flash Storage+xDisplay+GPIO+USBs
  - Comm (WiFi, BT, Ethernet)

Classification of Embedded Systems



Why ARM processor

- As of 2005, **98%** of the more than one billion mobile phones sold each year used ARM processors
- As of 2009, ARM processors accounted for approximately **90%** of all embedded 32-bit RISC processors
- In 2010 alone, **6.1 billion** ARM-based processors, representing **95%** of smartphones, **35%** of digital televisions and set-top boxes and **10%** of mobile computers
- As of 2016, over 80 billion ARM processors have been produced



Laboratory Main Component

- We will be working with a state-of-the-art ARM processor:
  - STM32L476VGT6 microcontroller
  - On-board USB interface
  - Mbed-enabled
  - LCD Display 4 x 24 seg
  - 7 LEDs
  - Pushbutton + Joystick
  - Stereo Audio DAC
  - 9 axis Inertial Motion Sensor (IMU)
  - 128Mbit flash memory

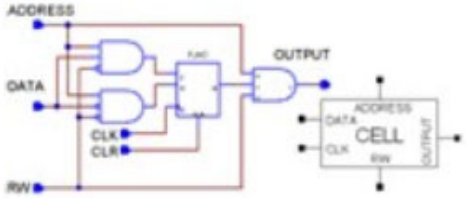
http://www.st.com/en/evaluation-tools/32l476gdiscovery.html

Why do we learn Assembly?

- Hardware/processor specific code
  - Processor booting code
  - Device drivers
  - A test-and-set atomic assembly instruction can be used to implement locks, mutex and semaphores.
- Cost-sensitive applications
  - Embedded devices, where the size of code is limited, washing machine controller, automobile controllers
- **The best applications are written by those who've mastered assembly language or fully understand the low-level implementation of the high-level language statements they're choosing.**

Embedded System Design Flow

Planning & Architect (modeling & simulation)



ASIC/SoC design

Hardware Design with CAD tools

ASIC/SoC chip

System prototype board

Hardware debugging & Software development

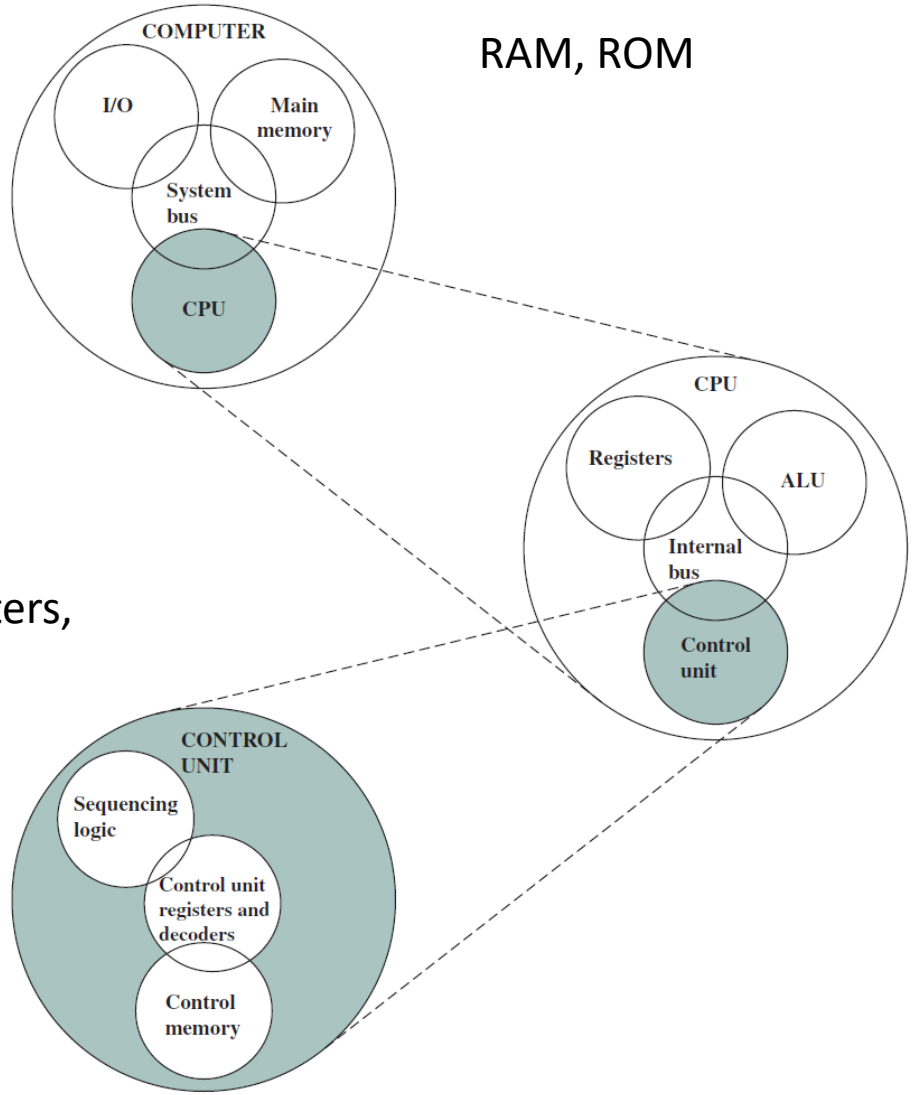
Final product

Chapter 1
Introduction to Computer Systems and Computer Programs

Edited: Waqas Majeed

Structure and Function

RAM, ROM

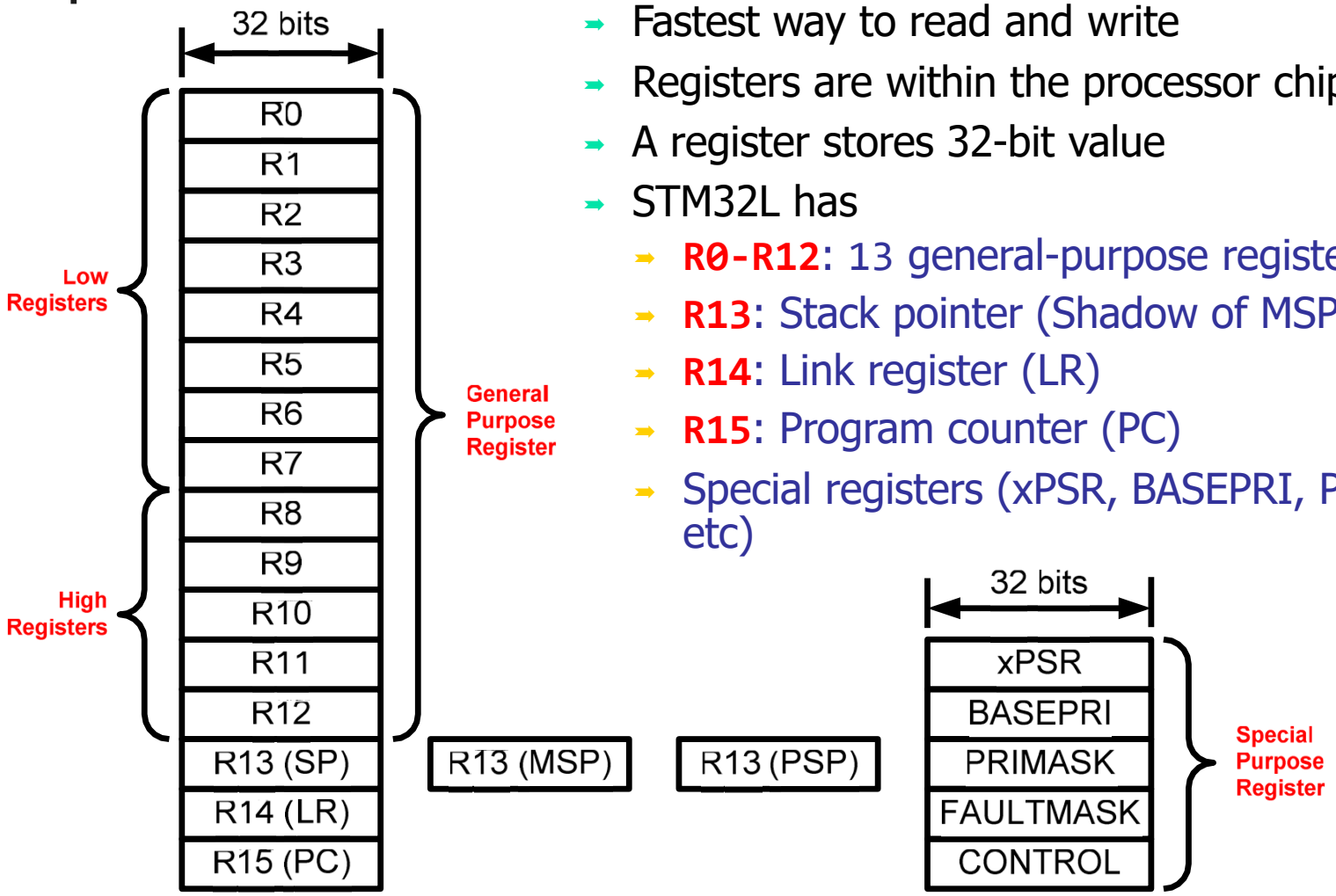


logic gates, adder,

flip-flops, counters, multiplexer,

Figure 1.1 The Computer: Top-Level Structure

ARM Processor Registers

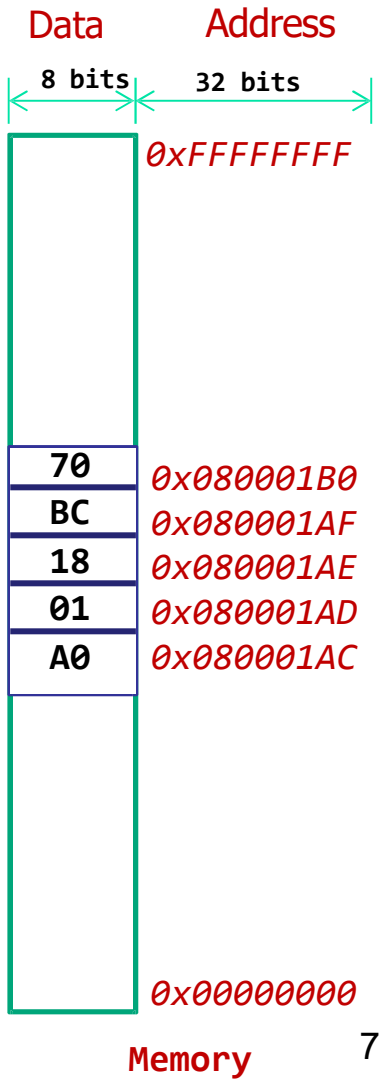


- Fastest way to read and write
- Registers are within the processor chip
- A register stores 32-bit value
- STM32L has
  - **R0-R12**: 13 general-purpose registers
  - **R13**: Stack pointer (Shadow of MSP or PSP)
  - **R14**: Link register (LR)
  - **R15**: Program counter (PC)
  - Special registers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

Thread mode
SPSEL-control
0 or 1

Memory

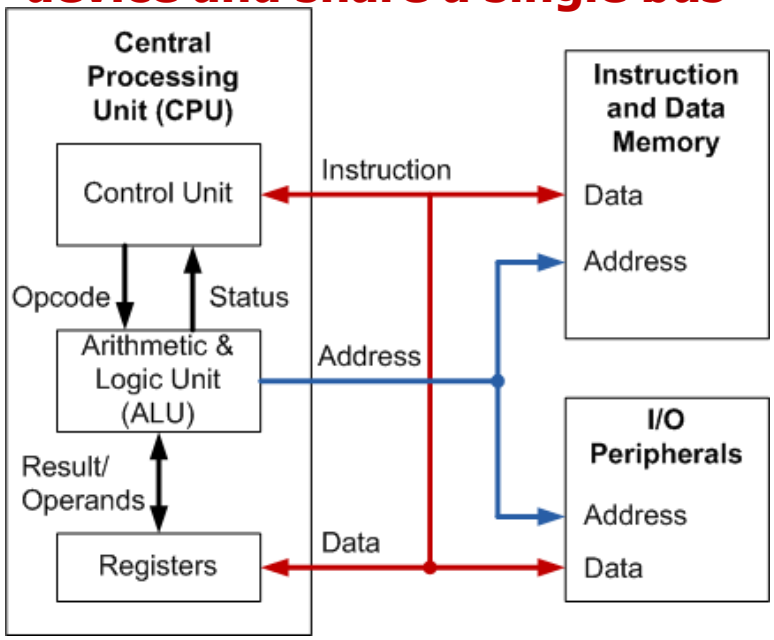
- Memory is arranged as a series of "locations"
  - Each location has a unique "address"
  - Each location holds a byte (**byte-addressable**)
  - e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112
- The number of locations in memory is limited
  - e.g. 4 GB of RAM
  - 1 Gigabyte (GB) = 2^{30} bytes
  - 2^{32} locations ➔ 4,294,967,296 locations!
- Values stored at each location can represent either program data or program instructions
  - e.g. the value `0x70` might be the code used to tell the processor to add two values together



Computer Architecture

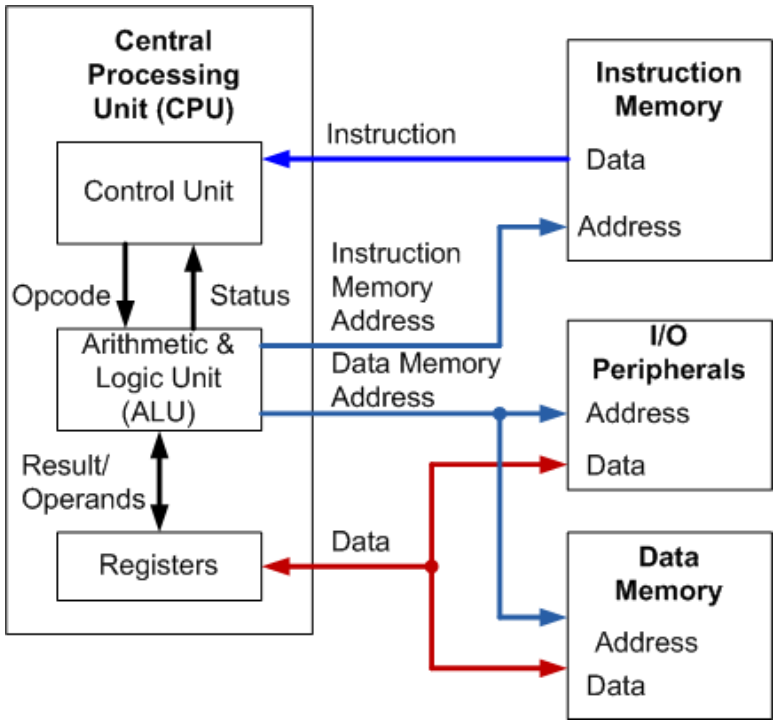
Von Neumann

Instructions and data are stored in the same memory device and share a single bus



Harvard

Data and instructions are stored in separate memory devices. Each has its own buses.



How Programs are Built

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
```

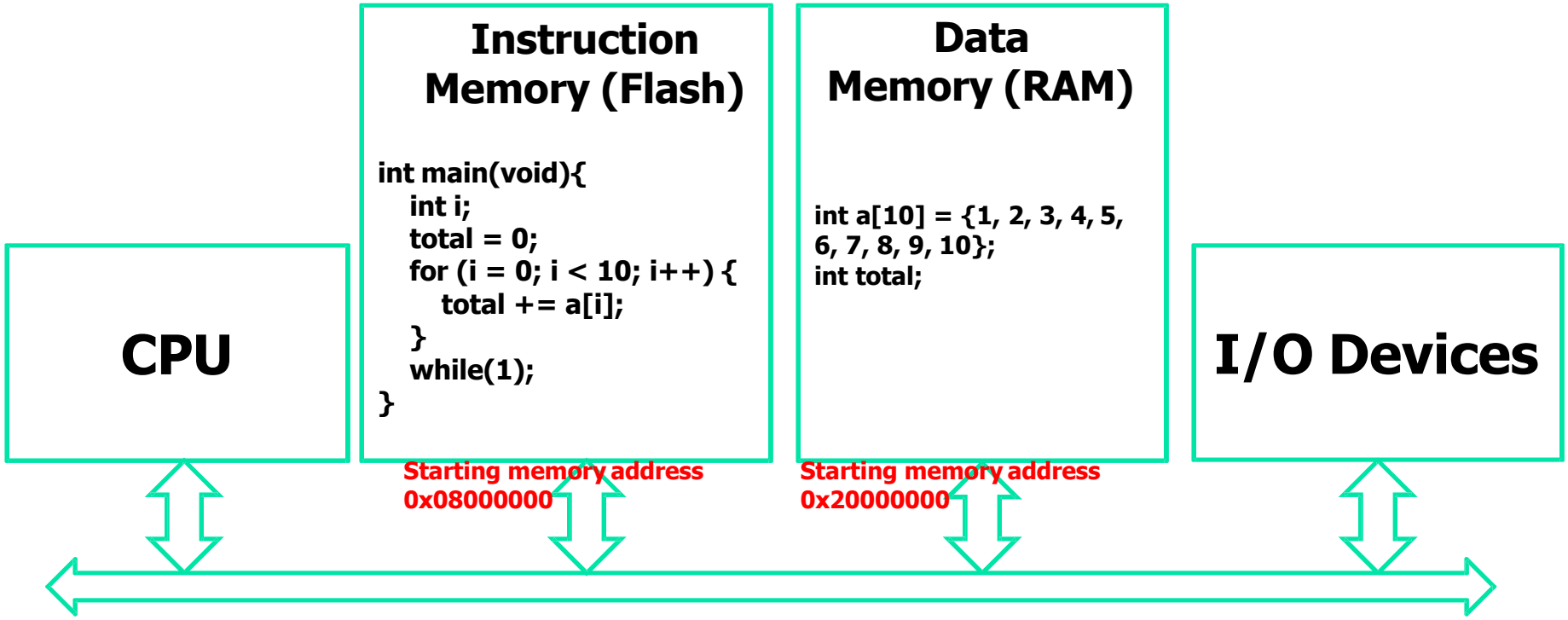
Assemble →

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 machine instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

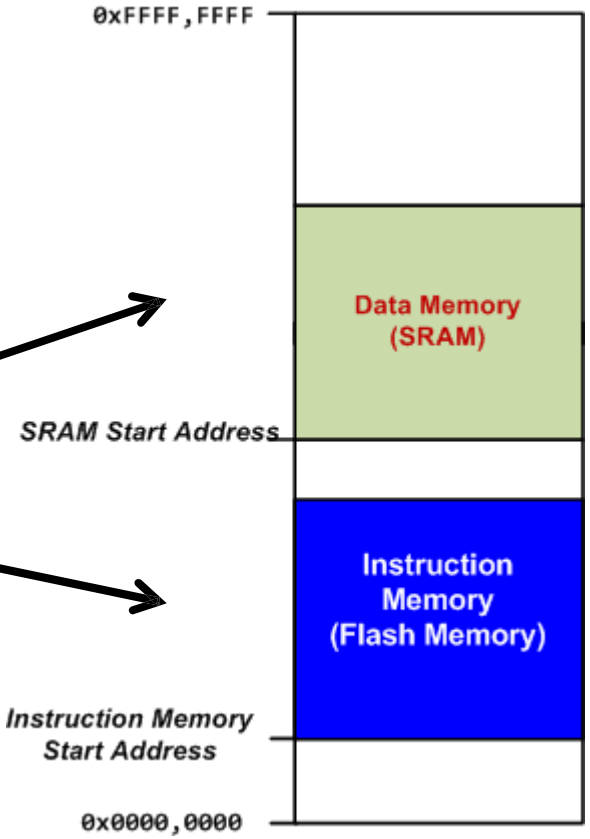
Example: Calculate the Sum of an Array

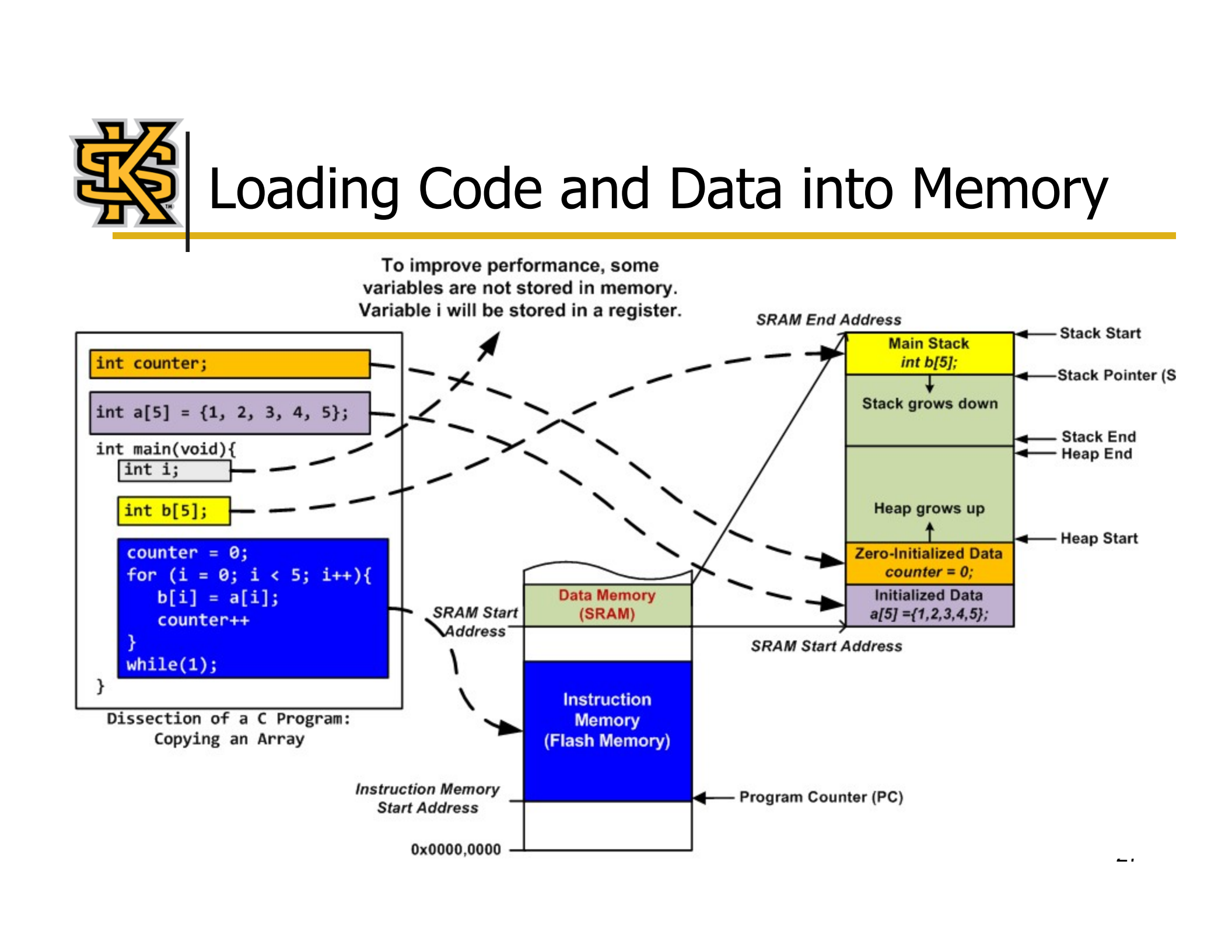


Loading Code and Data into Memory

```
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
   int i;

   int b[5];

   counter = 0;
   for (i = 0; i < 5; i++){
      b[i] = a[i];
      counter++
   }
   while(1);
}
```

C Program:
Copying an Array

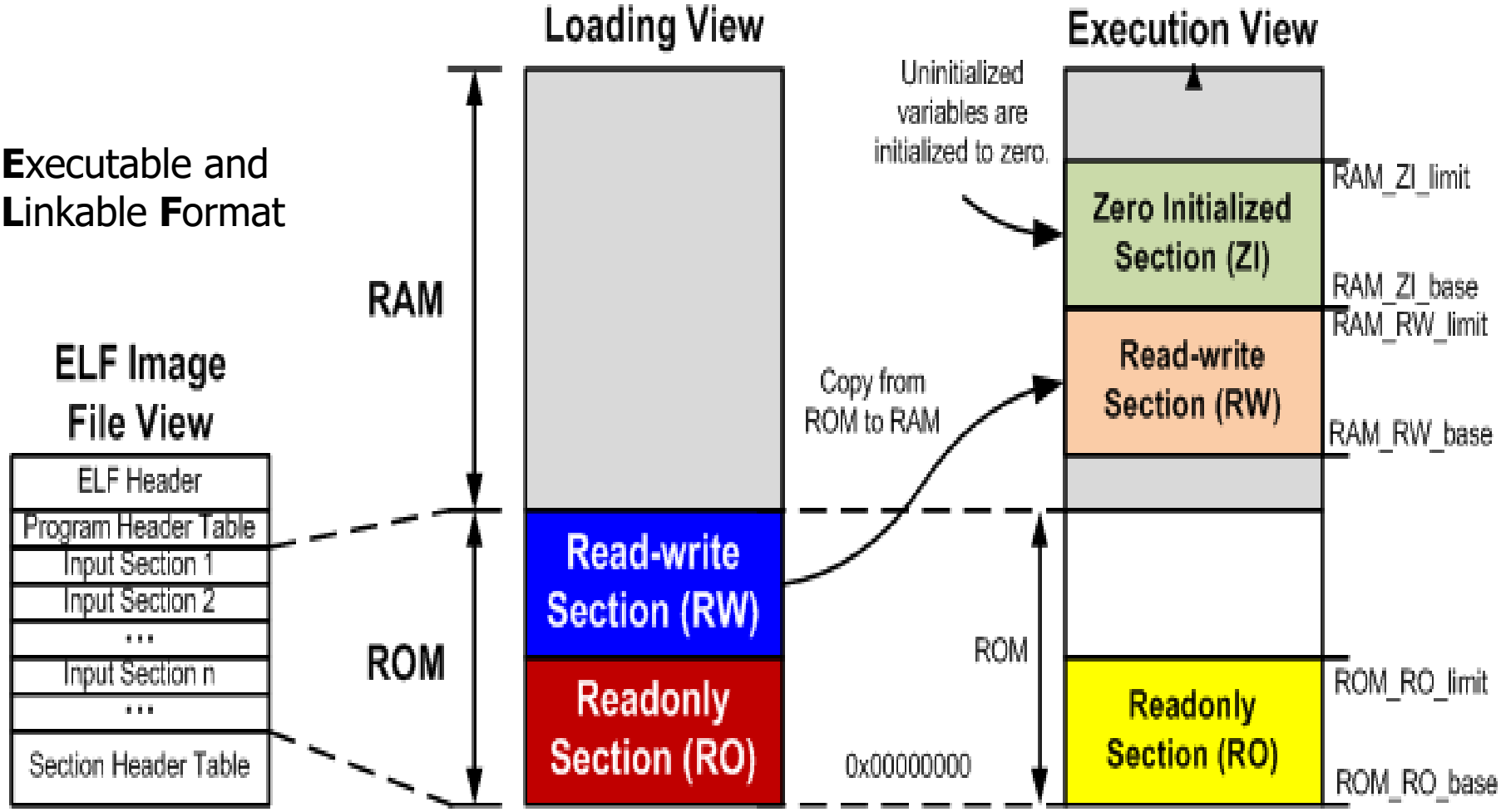




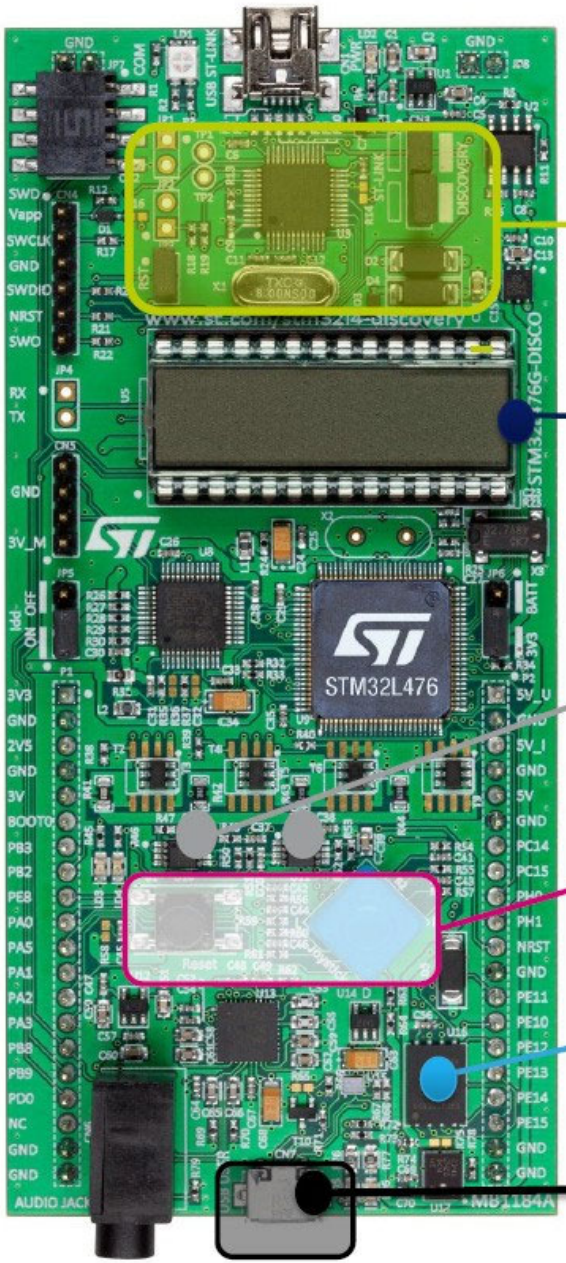
Loading Code and Data into Memory

View of a Binary Program

Executable and
Linkable **F**ormat



STM32L476 Discovery – HMI



Integrated ST-Link/V2-1 (for programming and debugging)

LCD 96 segments

Motion Mems (9-axis)

push buttons and joystick, 2 color LEDs

Quad SPI NOR Flash 16 MB

USB OTG connector

from st.com

STM32L476 Discovery - Audio and connector

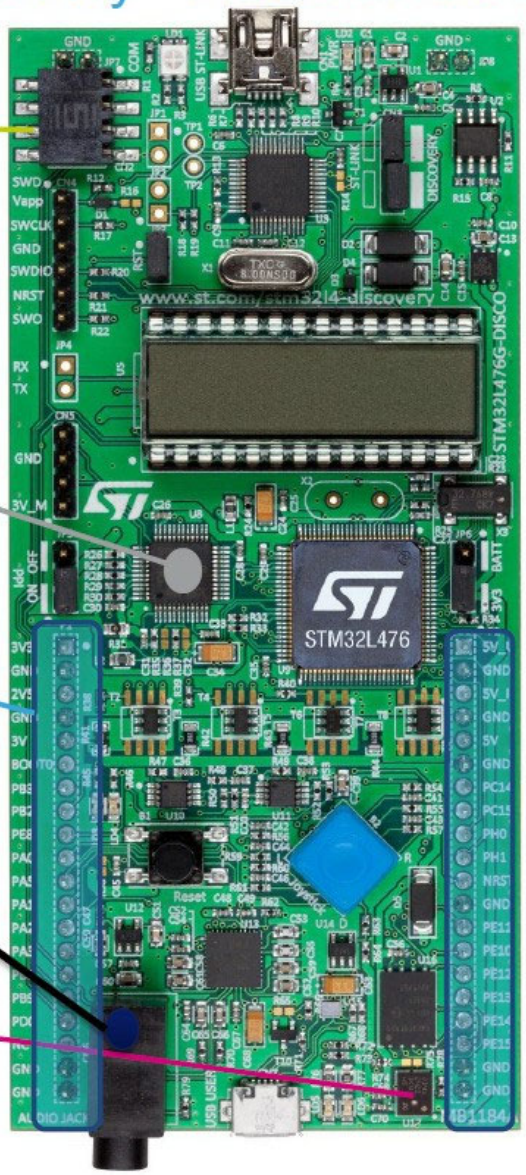
APC connector (for Apple connector)

MFX to auto-measure power consumption

Direct access to all MCU I/Os

Audio Codec and 3.5 mm connector

Microphone Mems



life.augmented

from st.com

STM32L4

Parallel Interface
FSMC 8-/16-bit (TFT-LCD, SRAM, NOR, NAND)

Display
LCD driver 8 x 40

Timers
17 timers including:
2 x 16-bit advanced motor control timers
2 x ULP timers
7 x 16-bit-timers
2 x 32-bit timers

I/Os
Up to 114 I/Os
Touch-sensing controller

Cortex-M4
80 MHz
FPU
MPU
ETM

DMA

ART Accelerator™

Up to 1-Mbyte Flash with ECC Dual Bank

128-Kbyte RAM

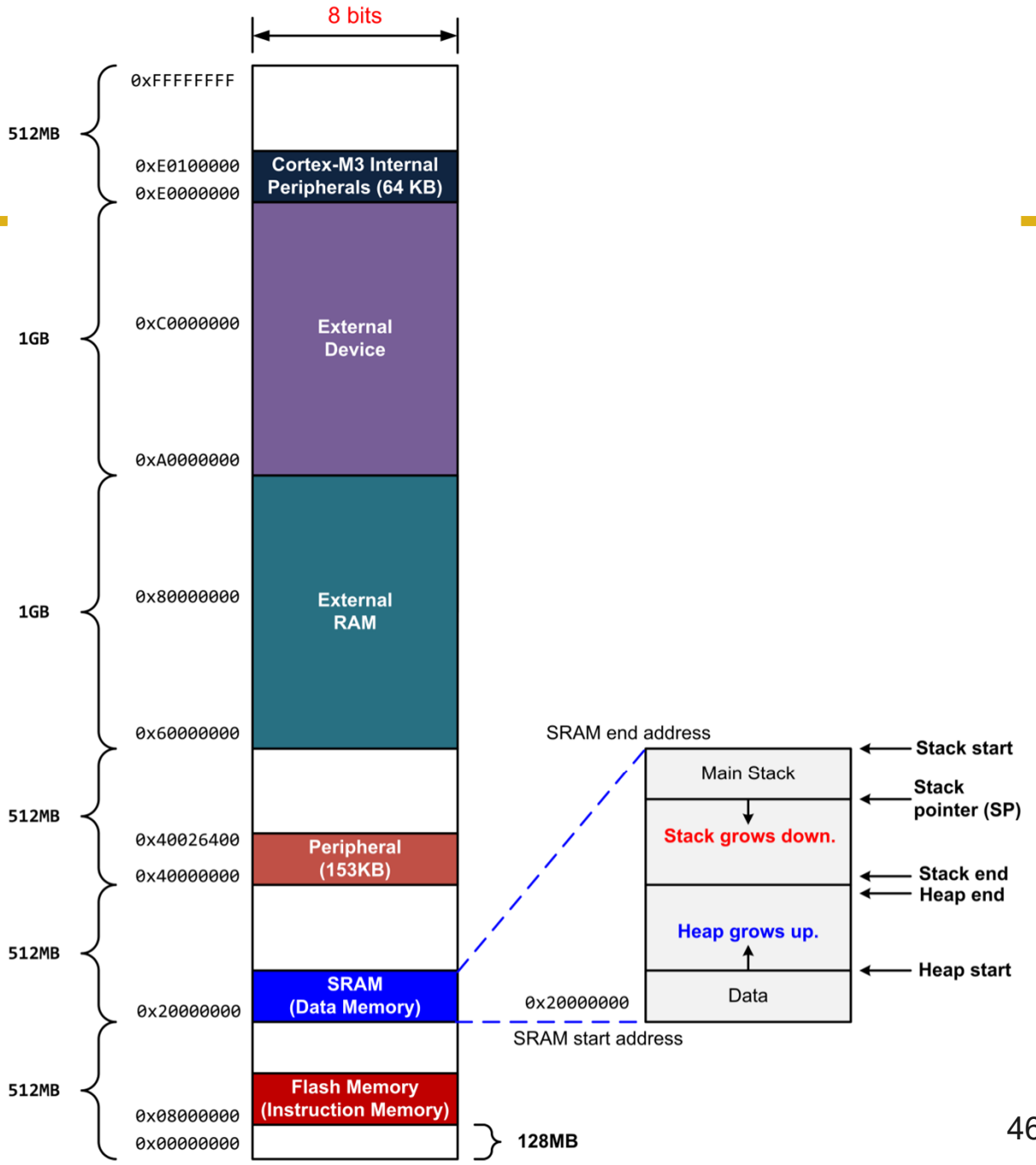
Connectivity
USB OTG,
1x SD/SDIO/MMC, 3 x SPI, 3 x I²C, 1x CAN, 1 x Quad SPI,
5 x USART + 1 x ULP

Digital
AES (256-bit), TRNG, 2 x SAI, DFSDM (8 channels)

Analog
3 x 16-bit ADC, 2 x DAC,
2 x comparators, 2 x op amps
1 x temperature sensor

from st.com

Memory Map


8 bits
0xFFFFFFFF
512MB
0xE0100000
0xE0000000
Cortex-M3 Internal Peripherals (64 KB)
1GB
0xC0000000
External Device
0xA0000000
1GB
0x80000000
External RAM
0x60000000
512MB
0x40026400
Peripheral (153KB)
0x40000000
512MB
SRAM (Data Memory)
0x20000000
512MB
Flash Memory (Instruction Memory)
0x08000000
0x00000000
128MB
SRAM end address
Main Stack
Stack grows down.
Heap grows up.
Data
0x20000000
SRAM start address
Stack start
Stack pointer (SP)
Stack end
Heap end
Heap start

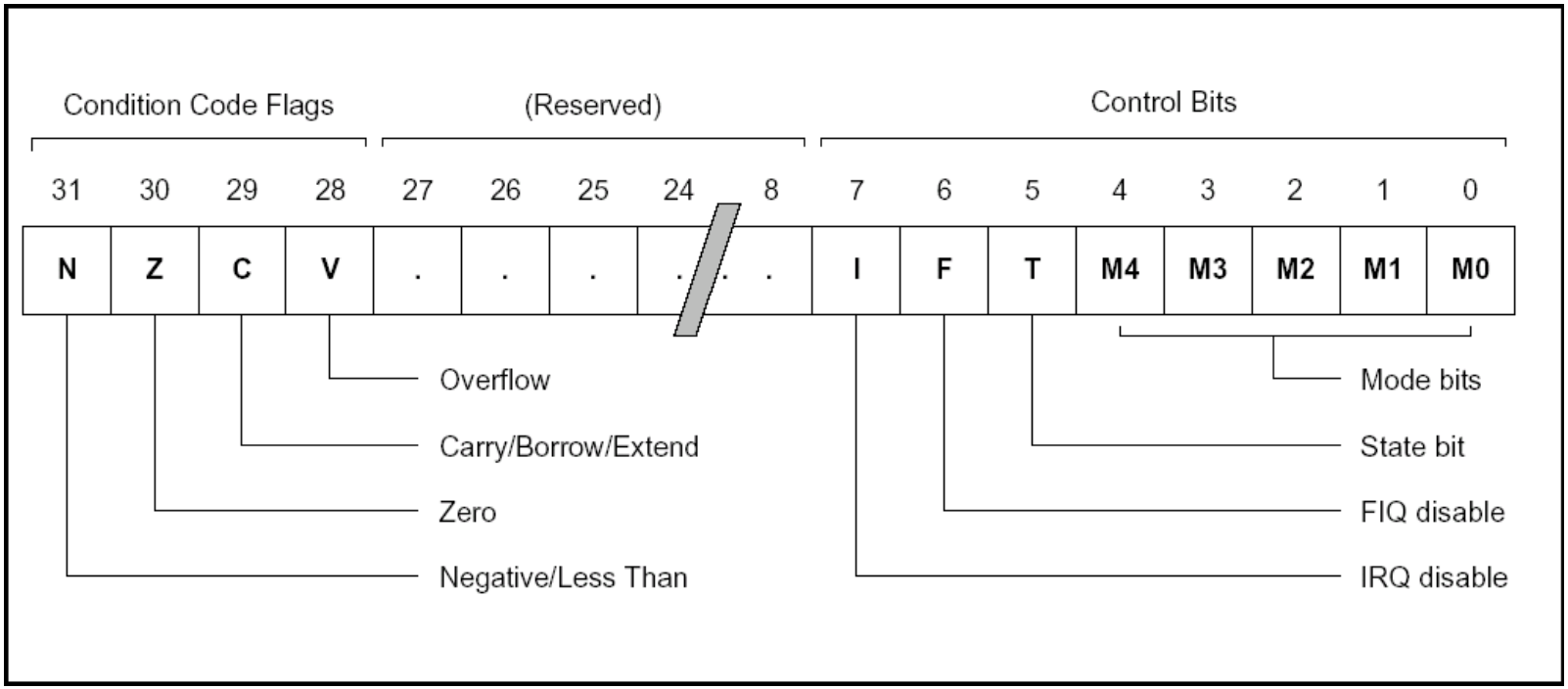
Embedded Systems with ARM Cortex-M Microcontrollers
in Assembly Language and C

Chapter 2
Data Representation

Edited: Waqas Majeed

ARM Current Program Status Register

CPSR



ARM CPSR Format

- N (Negative)
- Z (Zero)
- C (Carry)
- V (oVerflow)
- M0-M4 mode – control processor mode (see manual)
- T – control instruction set
  - T = 1 – instruction stream is 16-bit Thumb instructions
  - T = 0 – instruction stream is 32-bit ARM instructions
- I, F – interrupt enables

Signed Integer Representation Overview

- Three ways to represent signed binary integers:
  - Signed magnitude
    - $\text{value} = (-1)^{\text{sign}} \times \text{Magnitude}$
  - One's complement ($\tilde{\alpha}$)
    - $\alpha + \tilde{\alpha} = \mathbf{2^n - 1}$
  - Two's complement ($\bar{\alpha}$)
    - $\alpha + \bar{\alpha} = \mathbf{2^n}$

	Sign-and-Magnitude	One's Complement	Two's Complement
Range	$[-2^{n-1}+1, 2^{n-1}-1]$	$[-2^{n-1}+1, 2^{n-1}-1]$	$[-2^{n-1}, 2^{n-1}-1]$
Zero	Two zeroes (± 0)	Two zeroes (± 0)	One zero
Unique Numbers	$2^n - 1$	$2^n - 1$	2^n

Signed Integer Representation
Method 3: Two's Complement

Assume a four-bit system:

Expression	Result	Carry ?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	Yes(unsigned), No(signed)
1100 + 1110	1010	Yes	No	No(unsigned), Yes(signed)
1100 + 1010	0110	Yes	Yes	No

Why use Two's Complement

Two's complement simplifies hardware

Operation	Are signed and unsigned operations the same?
Addition	Yes
Subtraction	Yes
Multiplication	Yes if the product is required to keep the same number of bits as operands
Division	No

Condition Codes

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 is set upon an **unsigned** addition if the answer is wrong
C is cleared upon an **unsigned** subtract if the answer is wrong
V is set upon a **signed** addition or subtraction if the answer is wrong

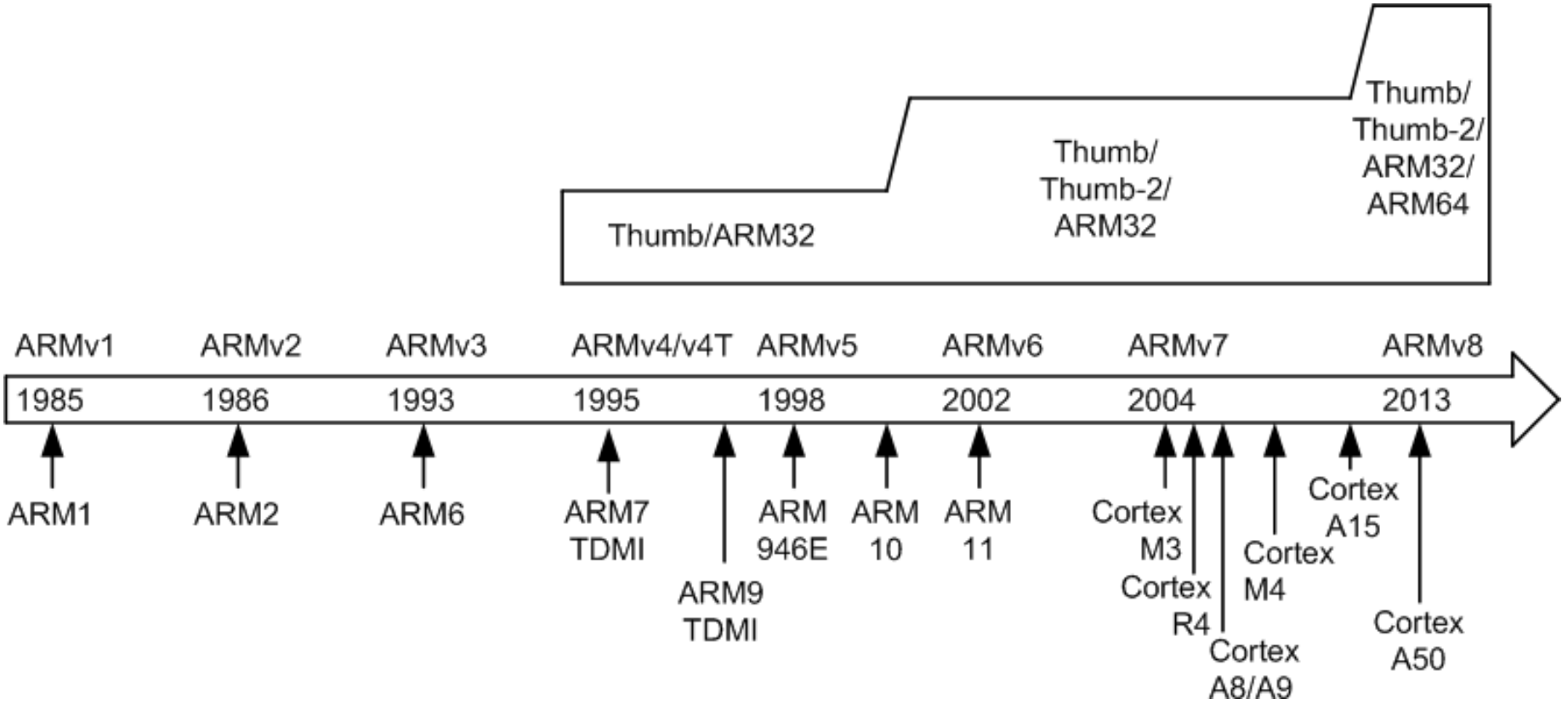
Why do we care about these bits?

Embedded Systems with ARM Cortex-M Microcontrollers
in Assembly Language and C

Chapter 3
ARM Instruction Set Architecture

Edited: Waqas Majeed

History

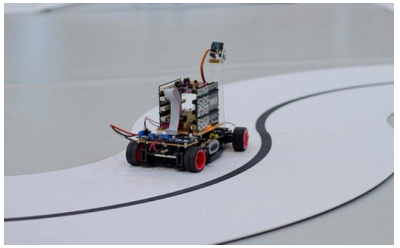

Thumb/ARM32
Thumb/
Thumb-2/
ARM32
Thumb/
Thumb-2/
ARM32/
ARM64
ARMv1
1985
ARM1
ARMv2
1986
ARM2
ARMv3
1993
ARM6
ARMv4/v4T
1995
ARM7
TDMI
ARM9
TDMI
ARMv5
1998
ARM
946E
ARM
10
ARMv6
2002
ARM
11
ARMv7
2004
Cortex
M3
Cortex
R4
Cortex
A8/A9
Cortex
M4
ARMv8
2013
Cortex
A15
Cortex
A50

ARM Cortex Processors

- ARM Cortex-A family:
  - **A**pplications processors
  - Support OS and high-performance applications
  - Such as Smartphones, Smart TV

- ARM Cortex-R family:
  - **R**eal-time processors with high performance and high reliability
  - Support real-time processing and mission-critical control

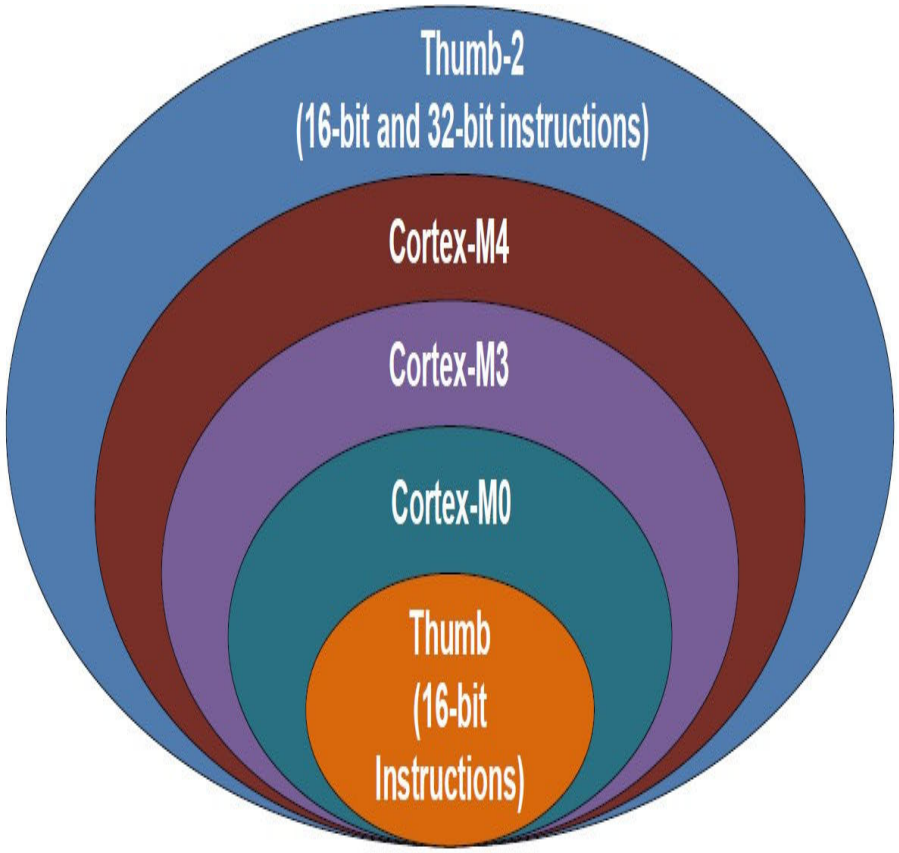
- ARM Cortex-M family:
  - **M**icrocontroller
  - Cost-sensitive, support SoC

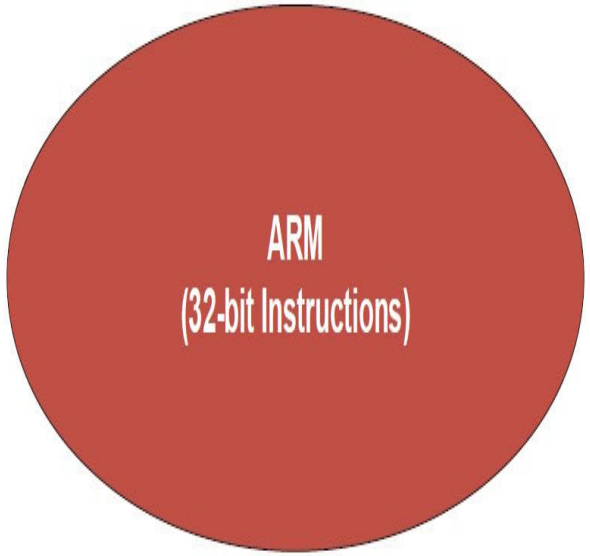


ARM Assembly Instruction Sets

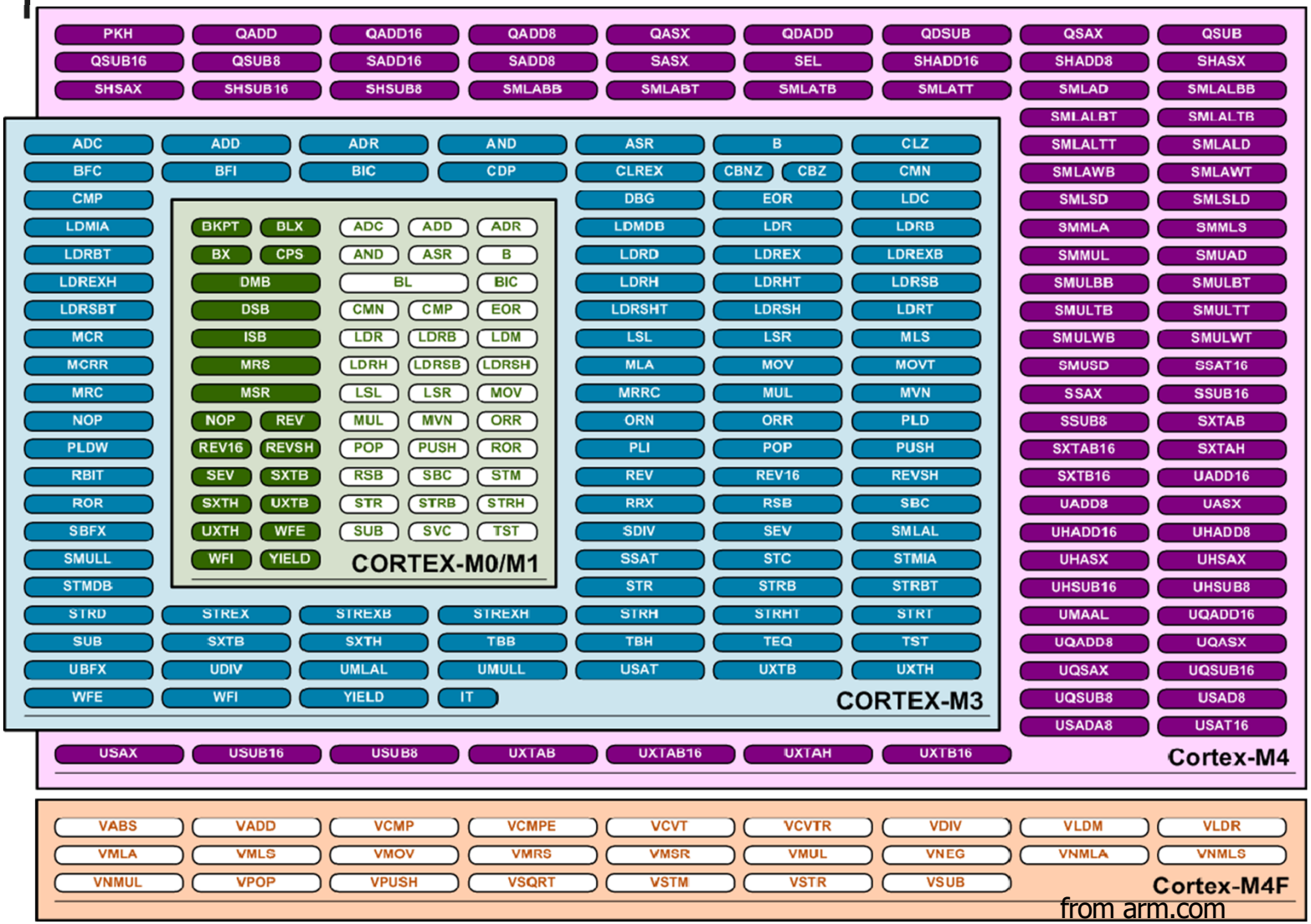
- ARM processors support 4 different assembly instruction sets
  1. Thumb – 16-bit length, improve code density
  2. ARM32 – 32-bit length, more flexible and faster than Thumb, lower code density
  3. Thumb-2 – Mixture of 16-bit and 32-bit instructions, compromise between Thumb and ARM32
  4. ARM64 – 64-bit length, used for desktops and servers

Instruction Sets





Instruction Sets

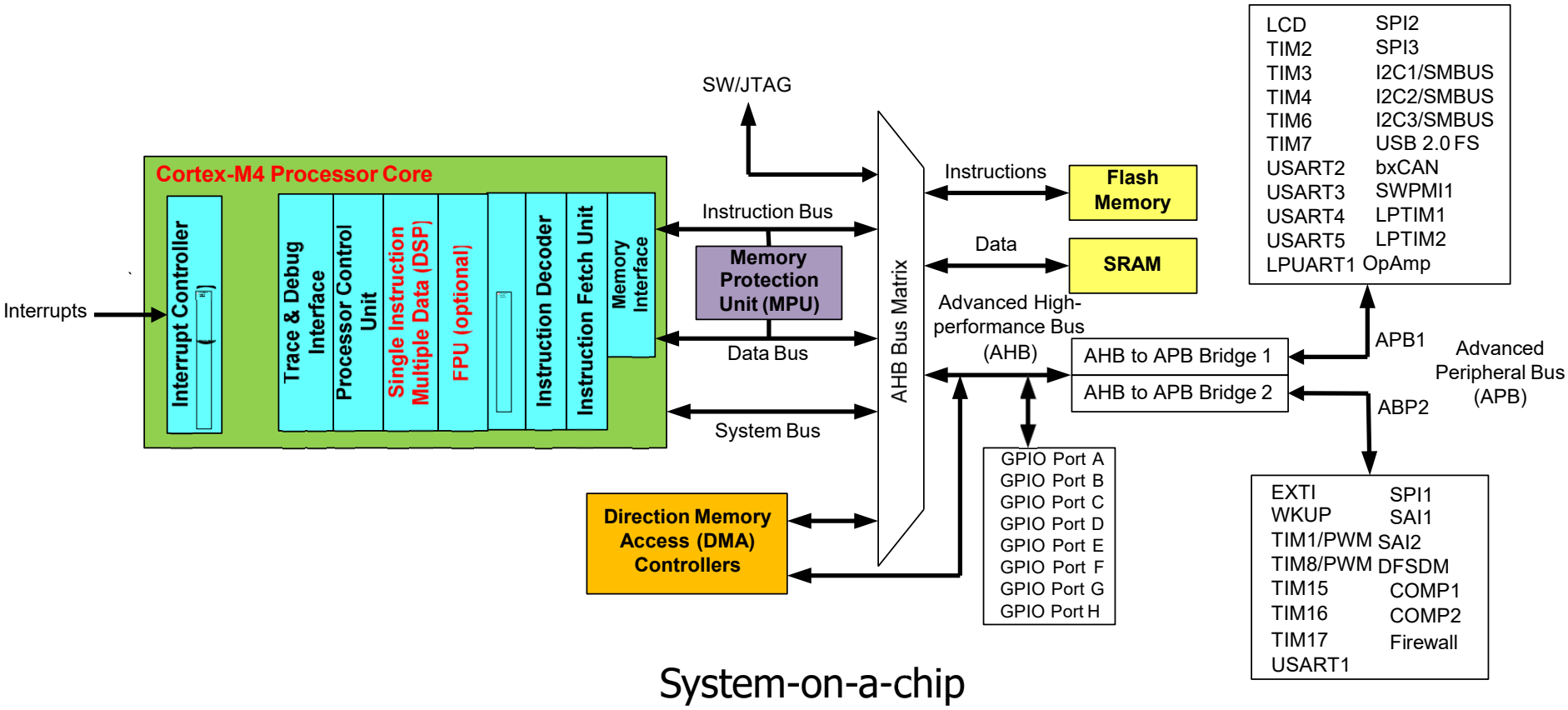


from arm.com

ARM Instruction Set Architecture (ISA)

- ARM processors have a load-store instruction set architecture
  - ALU cannot directly use data stored in memory as operands.
  - ALU source /destination operands must be general-purpose registers.
  - In order to access/modify data in memory, load and store instructions must be used.

ARM Cortex-M4 Organization (STM32L4)



System-on-a-chip

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
  - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier

ARM Instruction Format

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

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

label

mnemonic

destination operand

1st source operand

2nd source operand

comment

Assembly Directives

- Directives are **NOT** instructions. Instead, they are used to provide key information to compiler.

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Directive: Data Allocation

- Data allocation directives are used to reserve space in data memory for variables and set their initial content.

Directive	Description	Memory Space
DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values
DCQ	Define Constant	Reserve 64-bit values
DCFS	Define single-precision floating-point numbers	Reserve 32-bit values
DCFD	Define double-precision floating-point numbers	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve a number of zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Chapter 4
ARM Arithmetic and Logic Instructions

Edited: Waqas Majeed

Overview: Arithmetic and Logic Instructions

- Shift
  - LSL (logic shift left), LSR (logic shift right), ASR (arithmetic shift right), ROR (rotate right), RRX (rotate right with extend)
- Logic
  - AND (bitwise and), ORR (bitwise or), EOR (bitwise exclusive or), ORN (bitwise or not), MVN (move not)
- Bit set/clear
  - BFC (bit field clear), BFI (bit field insert), BIC (bit clear), CLZ (count leading zeroes)
- Bit/byte reordering
  - RBIT (reverse bit order in a word), REV (reverse byte order in a word), REV16 (reverse byte order in each half-word independently), REVSH (reverse byte order in each half-word independently)
- Addition
  - ADD, ADC (add with carry)
- Subtraction
  - SUB, RSB (reverse subtract), SBC (subtract with carry)
- Multiplication
  - MUL (multiply), MLA (multiply-accumulate), MLS (multiply-subtract), SMULL (signed long multiply-accumulate), SMLAL (signed long multiply-accumulate), UMULL (unsigned long multiply-subtract), UMLAL (unsigned long multiply-subtract)
- Division
  - SDIV (signed), UDIV (unsigned)
- Saturation
  - SSAT (signed), USAT (unsigned)
- Sign extension
  - SXTB (signed), SXTH, UXTB, UXTH
- Bit field extract
  - SBFX (signed), UBFX (unsigned)
- Syntax

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

Example: Add

- Unified Assembler Language (UAL) Syntax

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

- Traditional Thumb Syntax

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

Program Status Register (PSR)

- Cortex-M processors have 5 status flags
  - Negative (**N**)
    - Set if ALU result is negative.
  - Zero (**Z**)
    - Set if ALU result is zero.
  - Overflow (**V**)
    - Set if overflow occurs for signed addition/subtraction.
  - Carry (**C**)
    - Set (=1) if carry occurs for unsigned addition and set (=0) if borrow occurs for unsigned subtraction.
  - Saturation (**Q**)
    - Set if SSAT or USAT instruction causes saturation.

Program Status Register (PSR)

- Application PSR (**APSR**), Interrupt PSR (**IPSR**), Execution PSR (**EPSR**)

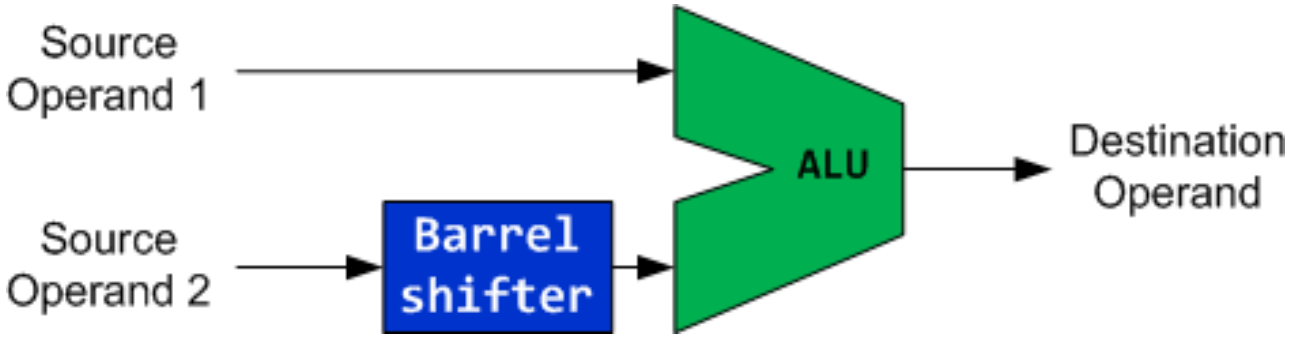
	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

- Combine them together into one register (**PSR**)
- Use PSR in code

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
PSR	N	Z	C	V	Q	ICI/IT	T		GE	ICI/IT		Exception Number				

Note: GE flags are only available on Cortex-M4 and M7

Barrel Shifter

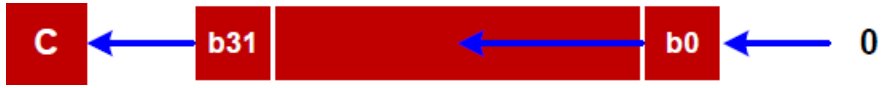


- The 2nd operand of ALU has a special hardware called Barrel shifter that enables quick shift and rotate operations on 2nd operand
- Example:

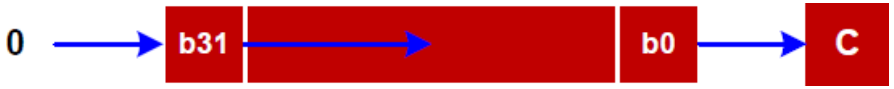
```
ADD r1, r0, r0, LSL #3 ; r1 = r0 + r0 << 3 = 9 x r0
```

Shift and Rotate Instructions

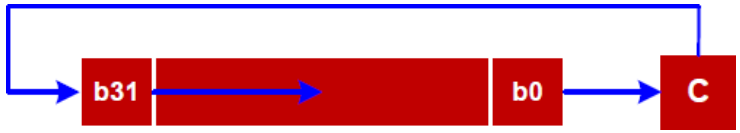
Logical Shift Left (**LSL**)



Logical Shift Right (**LSR**)

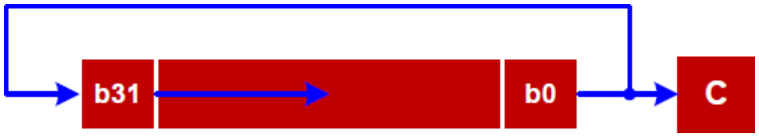


Rotate Right Extended (**RRX**)



Arithmetic Shift Right (**ASR**)

Rotate Right (**ROR**)



Why is there rotate right but no rotate left?

Rotate left can be replaced by a rotate right with a different rotate offset

Rotate left *n* bits = Rotate right 32 - *n*

Barrel Shifter

- Examples:
  - **ADD r1, r0, r0, LSL #3**

    ; r1 = r0 + r0 << 3 = r0 + 8 × r0

  - **ADD r1, r0, r0, LSR #3**

    ; r1 = r0 + r0 >> 3 = r0 + r0/8 (unsigned)

  - **ADD r1, r0, r0, ASR #3**

    ; r1 = r0 + r0 >> 3 = r0 + r0/8 (signed)

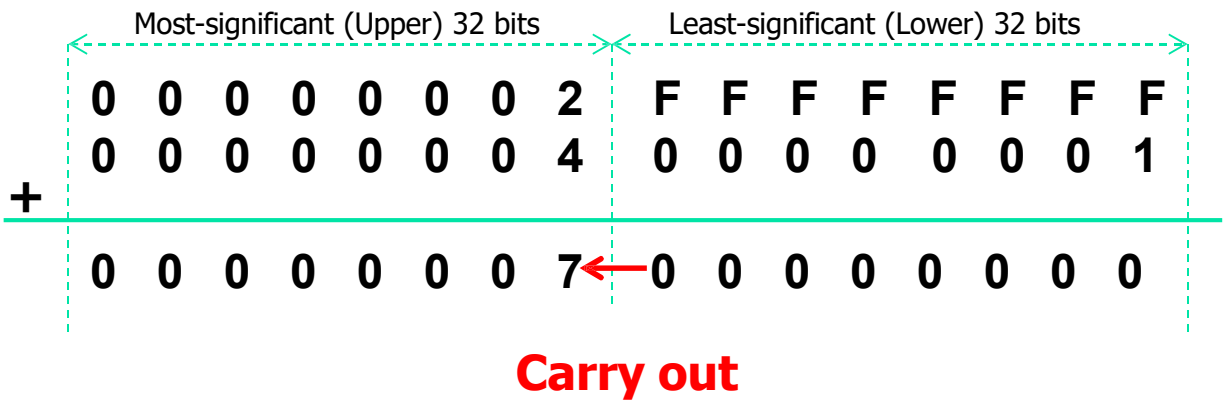
- Use Barrel shifter to speed up the application

```
ADD r1, r0, r0, LSL #3  <=>  MOV r2, #9      ; r2 = 9
                             MUL r1, r0, r2 ; r1 = r0 * 9
```

Arithmetic Operations

ADD {Rd,} Rn, Op2	Add. Rd ← Rn + Op2
ADC {Rd,} Rn, Op2	Add with carry. Rd ← Rn + Op2 + Carry
SUB {Rd,} Rn, Op2	Subtract. Rd ← Rn - Op2
SBC {Rd,} Rn, Op2	Subtract with carry. Rd ← Rn - Op2 + Carry - 1
RSB {Rd,} Rn, Op2	Reverse subtract. Rd ← Op2 - Rn
MUL {Rd,} Rn, Rm	Multiply. Rd ← (Rn × Rm)[31:0]
MLA Rd, Rn, Rm, Ra	Multiply with accumulate. Rd ← (Ra + (Rn × Rm))[31:0]
MLS Rd, Rn, Rm, Ra	Multiply and subtract, Rd ← (Ra – (Rn × Rm))[31:0]
SDIV {Rd,} Rn, Rm	Signed divide. Rd ← Rn / Rm
UDIV {Rd,} Rn, Rm	Unsigned divide. Rd ← Rn / Rm
SSAT Rd, #n, Rm {,shift #s}	Signed saturate
USAT Rd, #n, Rm {,shift #s}	Unsigned saturate

Example: 64-bit Addition



- A register can only store 32 bits
- A 64-bit integer needs two registers
- Split 64-bit addition into two 32-bit additions

Saturating Instruction: SSAT and USAT

- Syntax:
  - op{cond} Rd, #n, Rm{, shift}
- SSAT saturates a signed value to the signed range $-2^{n-1} \le x \le 2^{n-1}-1$.

$$SAT(x) = \begin{cases} 2^{n-1}-1 & if\ x > 2^{n-1}-1 \\ -2^{n-1} & if\ x < 2^{n-1} \\ x & otherwise \end{cases}$$

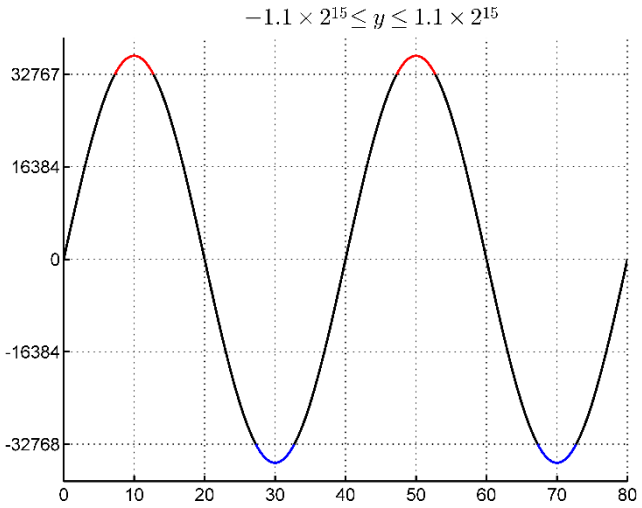
- USAT saturates a signed value to the unsigned range $0 \le x \le 2^{n}-1$.

$$USAT(x) = \begin{cases} 2^{n-1}-1 & if\ x > 2^{n-1}-1 \\ 0 & if\ x \le 0 \\ x & otherwise \end{cases}$$

- Examples:
  - `SSAT r2, #11, r1` ; output range: $-2^{10} \le r2 \le 2^{10}-1$
  - `USAT r2, #11, r3` ; output range: $0 \le r2 \le 2^{11}-1$

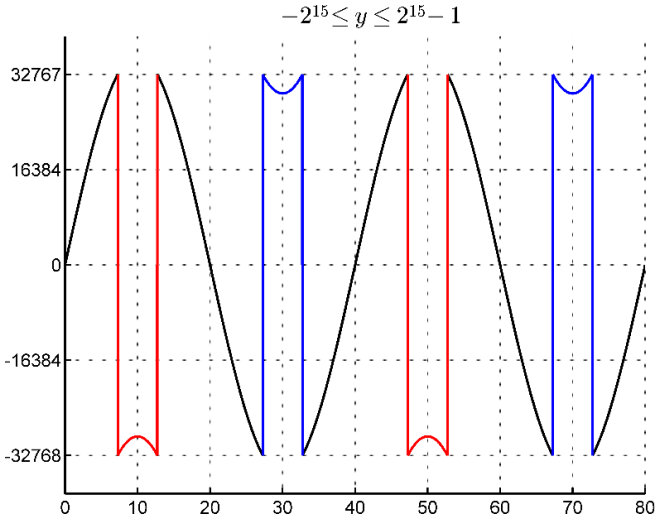
Example of Saturation

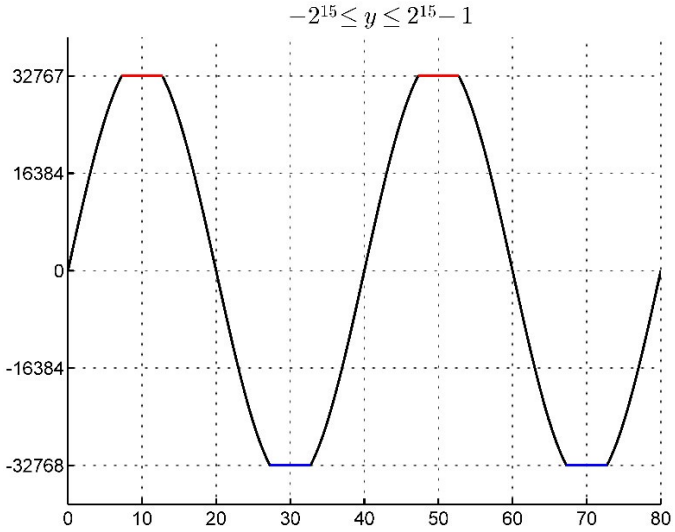
Assume data are limited to **16** bits



Without saturation

With saturation





Bitwise Logic

AND {Rd,} Rn, Op2	Bitwise logic AND. Rd ← Rn & operand2
ORR {Rd,} Rn, Op2	Bitwise logic OR. Rd ← Rn | operand2
EOR {Rd,} Rn, Op2	Bitwise logic exclusive OR. Rd ← Rn ^ operand2
ORN {Rd,} Rn, Op2	Bitwise logic OR NOT. Rd ← Rn | (NOT operand2)
BIC {Rd,} Rn, Op2	Bit clear. Rd ← Rn & NOT operand2
BFC Rd, #lsb, #width	Bit field clear. Rd[(width+lsb–1):lsb] ← 0
BFI Rd, Rn, #lsb, #width	Bit field insert. Rd[(width+lsb–1):lsb] ← Rn[(width-1):0]
MVN Rd, Op2	Move NOT, logically negate all bits. Rd ← 0xFFFFFFFF EOR Op2

Bit Operators (&, |, ~) vs Boolean Operators (&& ,| |, !)

A && B	Boolean and	**A & B**	Bitwise and
A||B	Boolean or	**A|B**	Bitwise or
!B	Boolean not	**~B**	Bitwise not

- The Boolean operators perform word-wide operations, not bitwise
- For example
  - "0x10 & 0x01" = 0x00, but "0x10 && 0x01" = 0x01.
  - "~0x01" = 0xFFFFFFFE, but "!0x01" = 0x00.

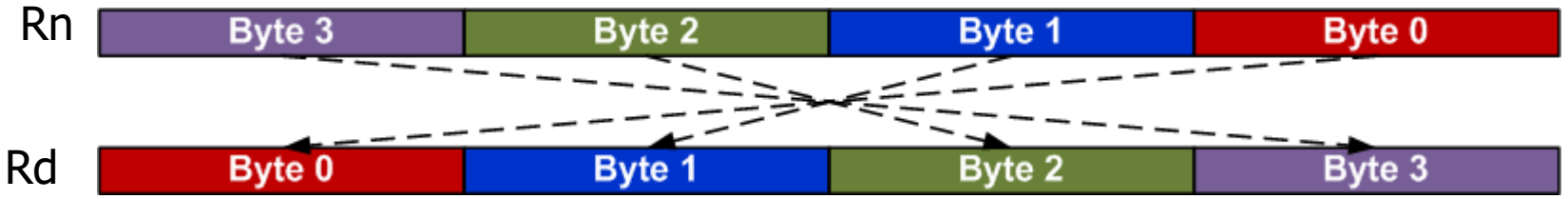
Bitwise Logic Instructions

- Logic operations with **S** suffix (ANDS, ORRS, EORS, ORNS and MVNS) will update N, Z and C flags only.
- C flag is updated when 2nd operand of a logic operation
  - Is a constant with MSB = 1.
  - Is shifted resulting shift operation results in C = 1.

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31−i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] && 0xFFFF

REV Rd, Rn



Example:

```
LDR R0, =0x12345678
REV R1, R0     ; R1 = 0x78563412
```

Sign and Zero Extension

SXTB {Rd,} Rm {,ROR #n}	Sign extend a byte. Rd[31:0] ← Sign Extend((Rm ROR n)[7:0])
SXTH {Rd,} Rm {,ROR #n}	Sign extend a half-word. Rd[31:0] ← Sign Extend((Rm ROR n)[15:0])
UXTB {Rd,} Rm {,ROR #n}	Zero extend a byte. Rd[31:0] ← Zero Extend((Rm ROR n)[7:0]) n))[7:0])
UXTH {Rd,} Rm {,ROR #n}	Zero extend a half-word. Rd[31:0] ← Zero Extend((Rm ROR n)[15:0]) n))[15:0])

```
LDR R0, =0x55AA8765
SXTB R1, R0    ; R1 = 0x00000065
SXTH R1, R0    ; R1 = 0xFFFF8765
UXTB R1, R0    ; R1 = 0x00000065
UXTH R1, R0    ; R1 = 0x00008765
```

Data Comparison

CMP Rn, Op2	Compare	Set NZCV flags on $Rn - Op2$
CMN Rn, Op2	Compare negative	Set NZCV flags on $Rn + Op2$
TST Rn, Op2	Test	Set NZCV flags on $Rn\ AND\ Op2$
TEQ Rn, Op2	Test Equivalence	Set NZCV flags on $Rn\ EOR\ Op2$

Data Movement between Registers

- 2 categories of instructions for moving data between registers
  - Move data between 2 general-purpose registers.
  - Move data between a general-purpose register and a special-purpose register.

Data Movement between Registers

MOV	Rd ← operand2
MVN	Rd ← NOT operand2
MRS Rd, spec_reg	Move from special register to general register
MSR spec_reg, Rm	Move from general register to special register

```
MOV r4, r5              ; Copy r5 to r4
MVN r4, r5              ; r4 = bitwise logical NOT of r5
MOV r1, r2, LSL #3      ; r1 = r2 << 3
MOV r0, PC              ; Copy PC (r15) to r0
MOV r1, SP              ; Copy SP (r14) to r1
```

Move Immediate Number to Register

MOVW Rd, #imm16	**Move Wide**, Rd ← #imm16
MOVT Rd, #imm16	**Move Top**, Rd ← #imm16 << 16
MOV Rd, #const	**Move**, Rd ← const

Example: Load a 32-bit number into a register

```
MOVW r0, #0x4321      ; r0 = 0x00004321
MOVT r0, #0x8765      ; r0 = 0x87654321
```

Order does matter!

- **MOVW** will zero the upper halfword
- **MOVT** won't zero the lower halfword

```
MOVT r0, #0x8765      ; r0 = 0x8765xxxx
MOVW r0, #0x4321      ; r0 = 0x00004321
```

Bit Field Extract

- These instructions extract adjacent bits from 1 register and place them in another

SBFX Rd, Rn, #lsb, #width	Signed Bit Field Extract $Rd[(width-1):0] \leftarrow Rn[(width+lsb-1):lsb]$ $Rd[31:width] \leftarrow Replicate(Rn[(width+lsb-1)])$
UBFX Rd, Rn, #lsb, #width	Unsigned Bit Field Extract $Rd[(width-1):0] \leftarrow Rn[(width+lsb-1):lsb]$ $Rd[31:width] \leftarrow Replicate(0)$

```
; Assume r3 = 0x1234CDEF

UBFX r4, r3, #4, #8   ; r4 = 0x000000DE (zero extension)
SBFX r4, r3, #4, #8   ; r4 = 0xFFFFFFDE (sign extension)
```

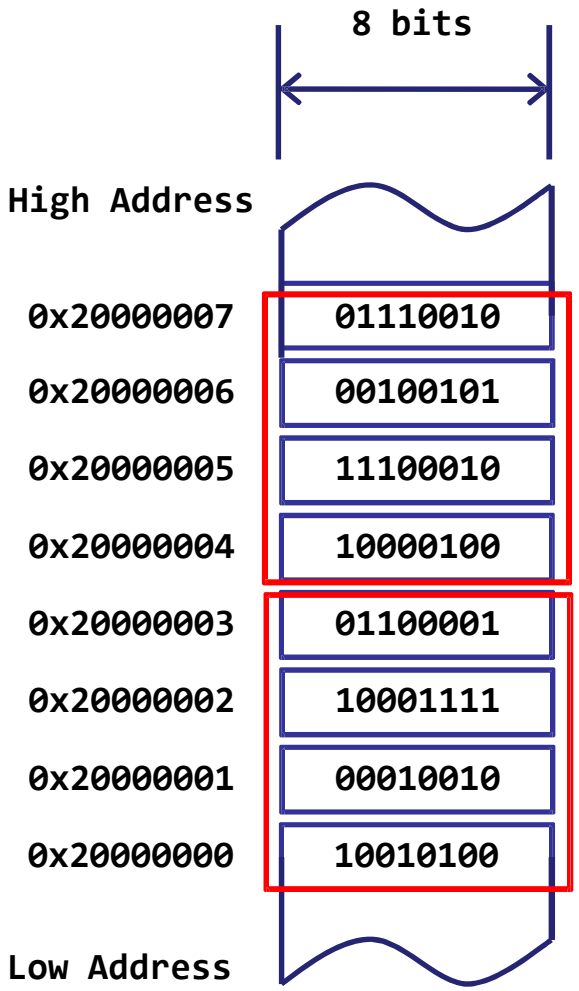
Chapter 5
Memory Access

Edited: Waqas Majeed

Logical View of Memory

- When we refer to memory locations by address, we can only do so in units of bytes, halfwords or words
- Words
  - 32 bits = 4 bytes = 1 word = 2 halfwords
  - In diagram on the right, we have two words at addresses:
    - 0x20000000
    - 0x20000004
  - Can you store a word anywhere? **NO.**
  - A word can only be stored at an address that's divisible by 4.
  - Memory address of a word is the lowest address of all four bytes in that word.

Word-address mod 4 = 0



Single register data transfer

- You can load/store data of different sizes between memory and registers

LDR	Load Word
LDRB	Load Byte
LDRH	Load Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword

STR	Store Word
STRB	Store Lower Byte
STRH	Store Lower Halfword

Address

- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes).

```
LDR r0,[r1,#8]
```

  - A register, optionally shifted by an immediate value

```
LDR r0,[r1,r2]       ;r1 = Memory.word[r1+r2]
LDR r0,[r1,r2,LSL#2];r1 = Memory.word[r1+4*r2]
```

- This can be either added or subtracted from the base register:

```
LDR r0,[r1,#-8]
LDR r0,[r1,-r2]
LDR r0,[r1,-r2,LSL#2]
```

- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (i.e. 0-255 bytes).
  - A register (unshifted).
- Choice of pre-indexed or post-indexed addressing

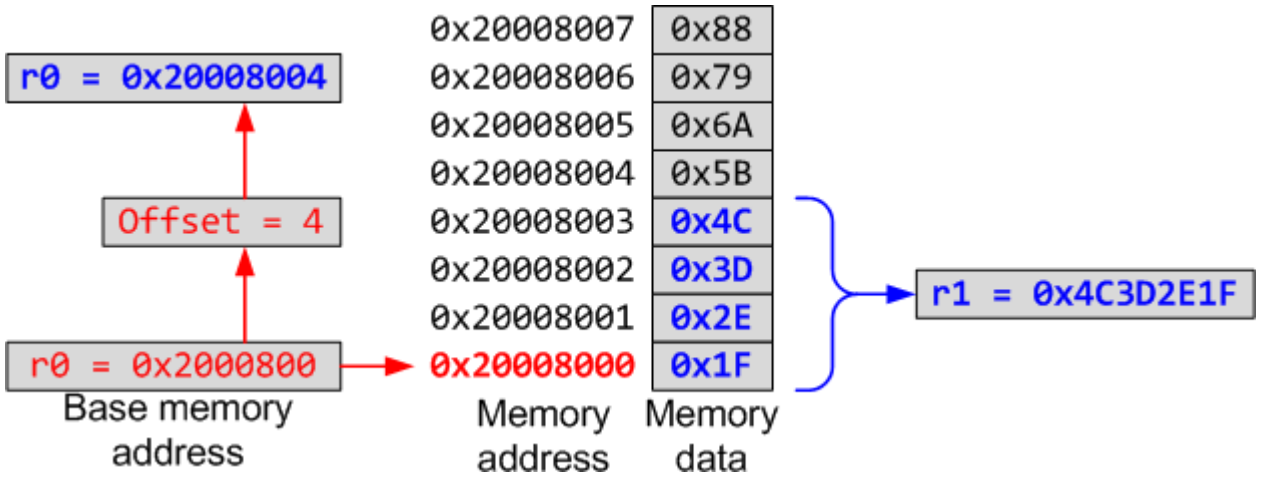
Pre-index

Pre-Index: LDR r1, [r0, #4]

Offset range is -255 to +255

Post-index

Post-Index: LDR r1, [r0], #4



Offset range is -255 to +255

Summary of Pre-index and Post-index

Index Format	Example	Equivalent
Pre-index	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged
Pre-index with update	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-index	LDR r1, [r0], #4	r1 ← memory[r0] r0 ← r0 + 4

Offset range is -255 to +255

Load/Store Multiple Registers

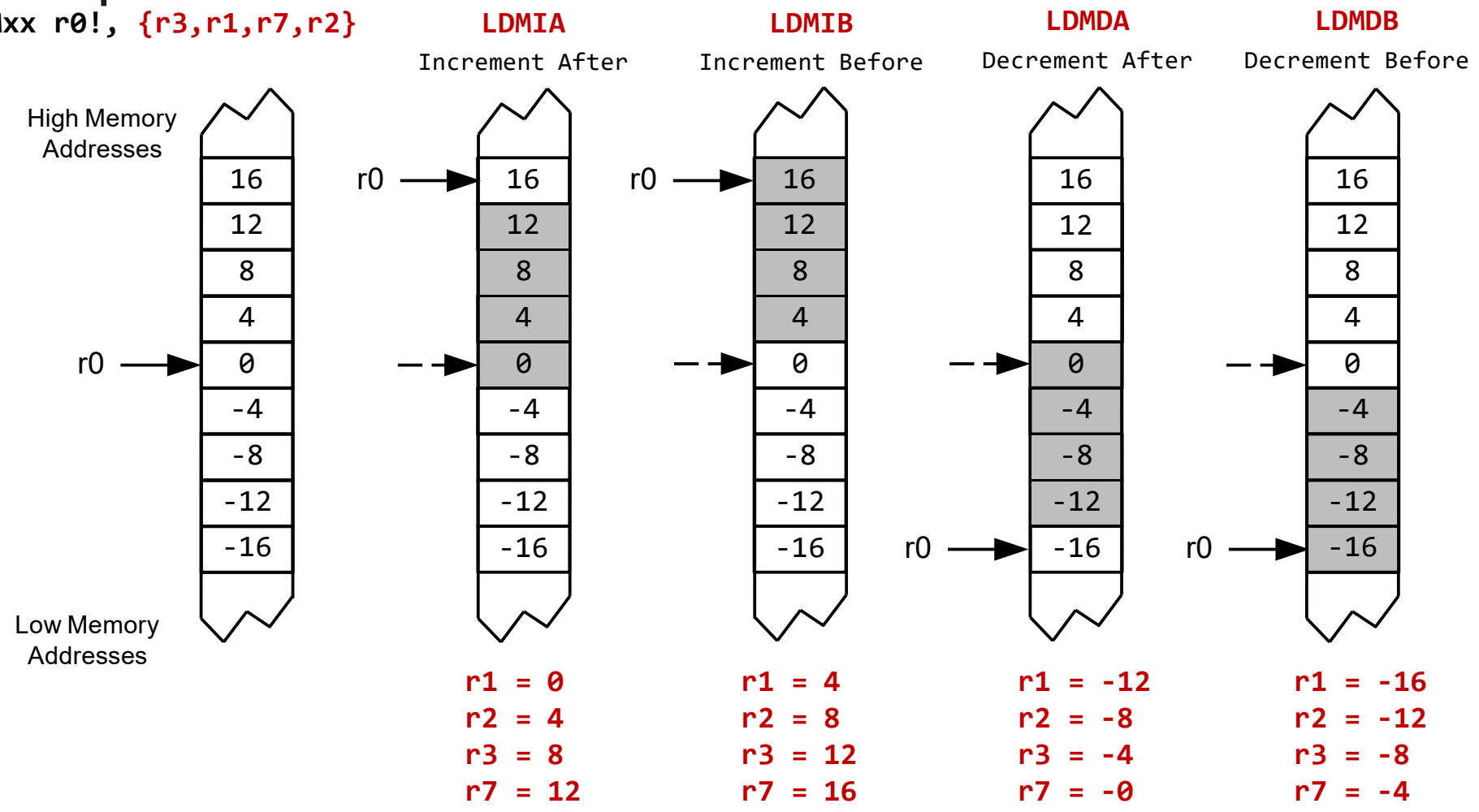
```
STMxx rn{!}, {register_list}
LDMxx rn{!}, {register_list}
```

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

- **IA**: address is incremented by 4 after a word is loaded or stored.
- **IB**: address is incremented by 4 before a word is loaded or stored.
- **DA**: address is decremented by 4 after a word is loaded or stored.
- **DB**: address is decremented by 4 before a word is loaded or stored.

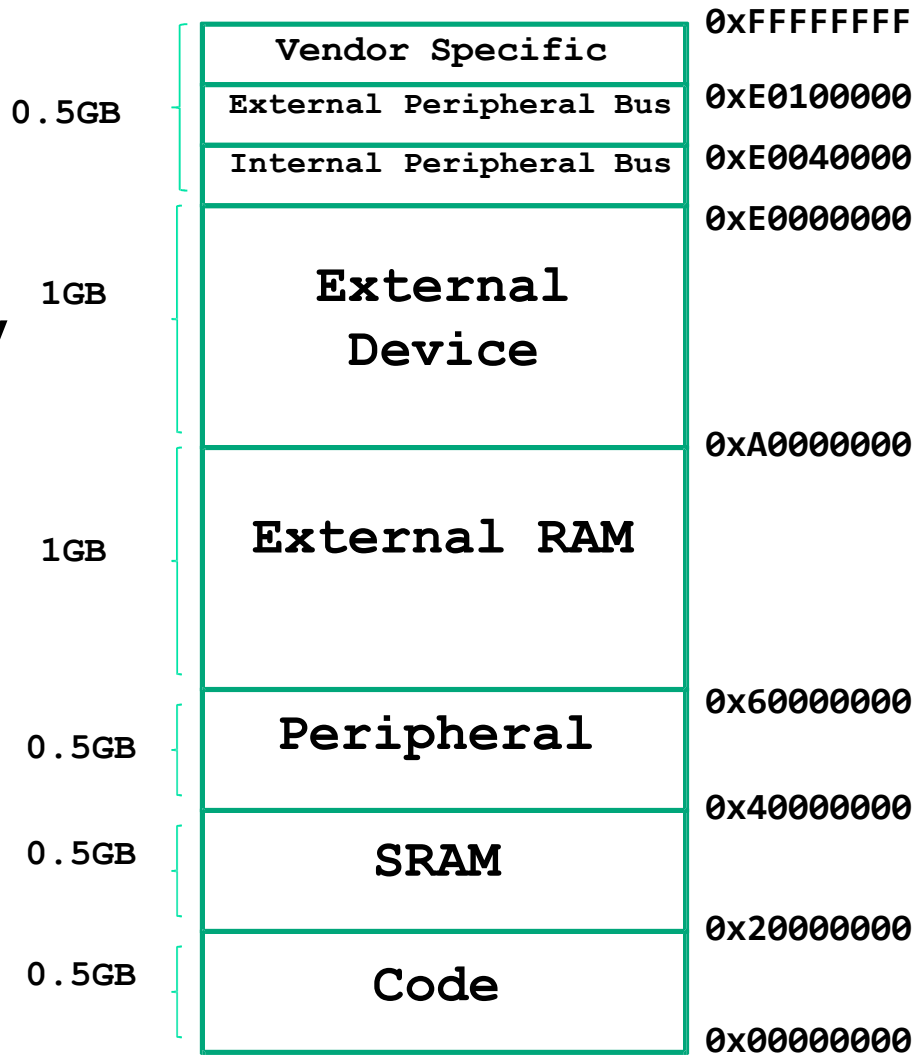
Load Multiple Registers

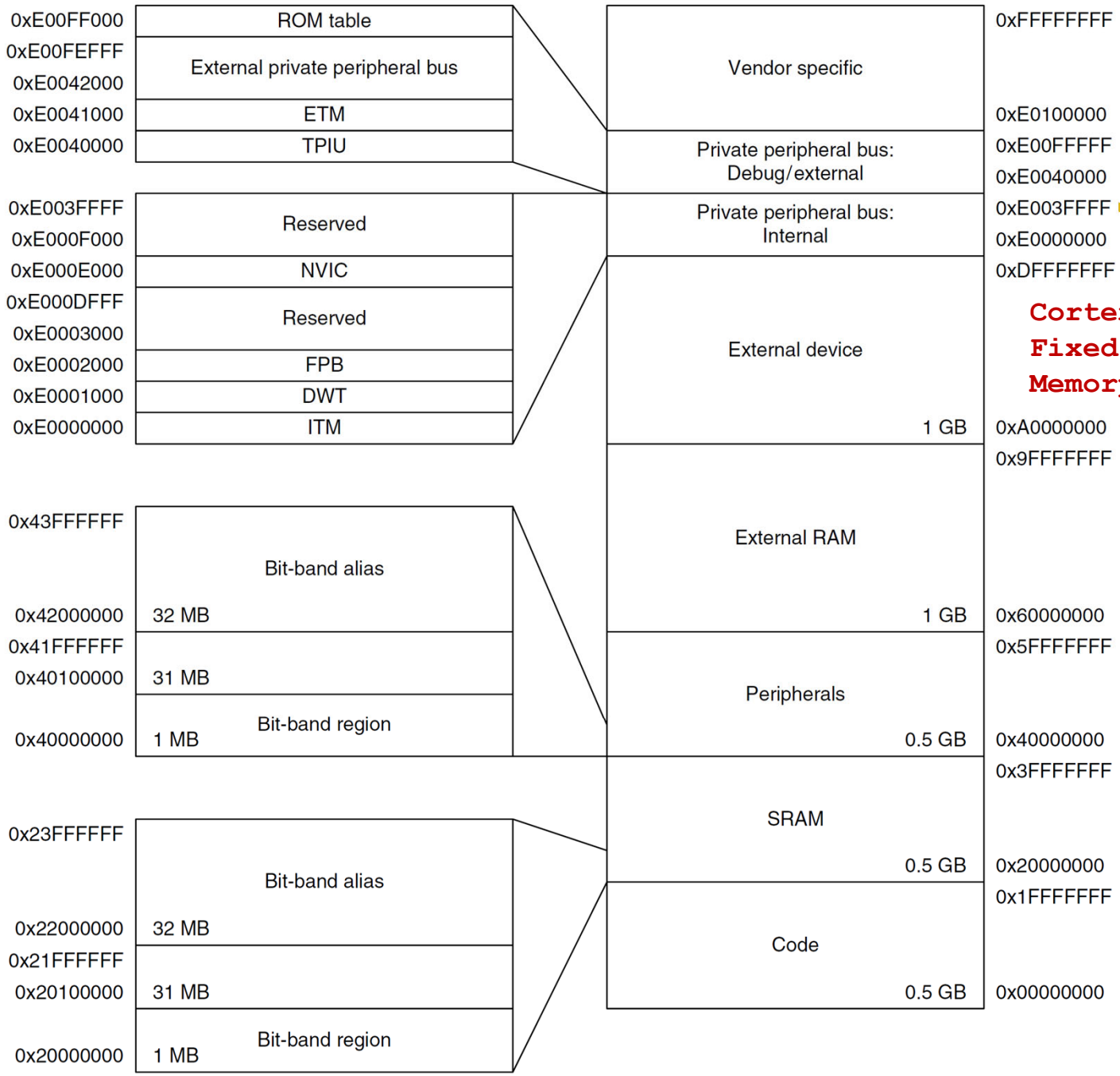
`LDMxx r0!, {r3,r1,r7,r2}`



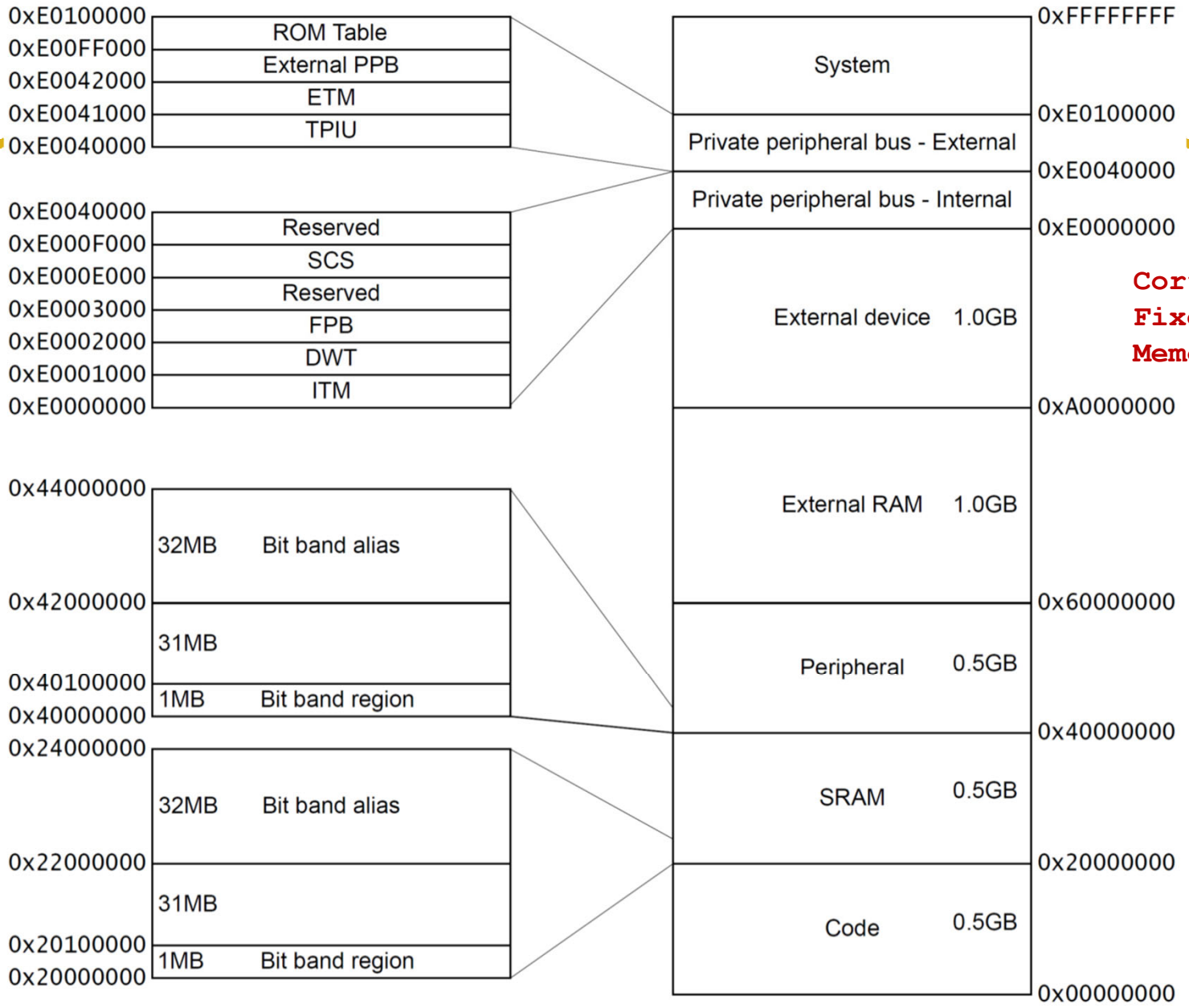
Cortex-M3 & Cortex-M4 Memory Map

- 32-bit Memory Address
- 2^{32} bytes of memory space (4 GB)
- Modified Harvard architecture:
  - Physically separated instruction memory and data memory
  - Separate buses
  - Unified address space





Cortex-M3 Fixed Memory Map



Cortex-M4 Fixed Memory Map

Chapter 6
Flow Control in Assembly

Edited: Waqas Majeed

Comparison Instructions

Instruction	Operands	Brief description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N,Z,C,V
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

- The only effect of the comparisons is to **update the condition flags**.
  - No need to set S bit.
  - No need to specify Rd.
- Operations are:
  - CMP operand1 - operand2, but result not written
  - CMN operand1 + operand2, but result not written
  - TEQ operand1 ^ operand2, but result not written
  - TST operand1 & operand2, but result not written
- Examples:
  - `CMP r0, r1`
  - `TST r2, #5`

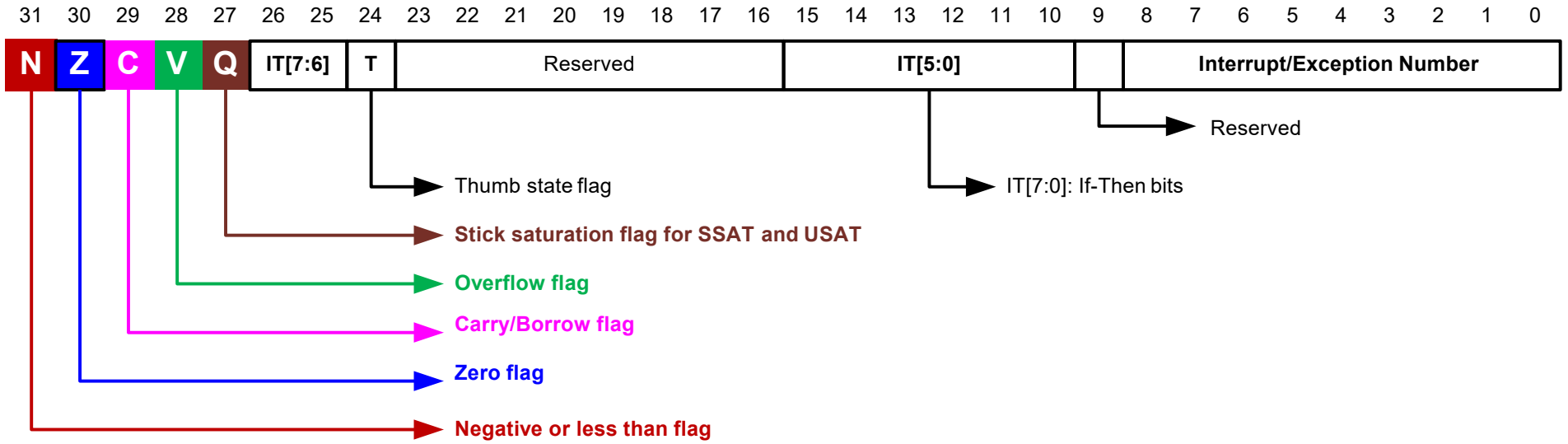
Condition Codes

- Status of a condition code depends on the status of N, Z, C, and V flags.

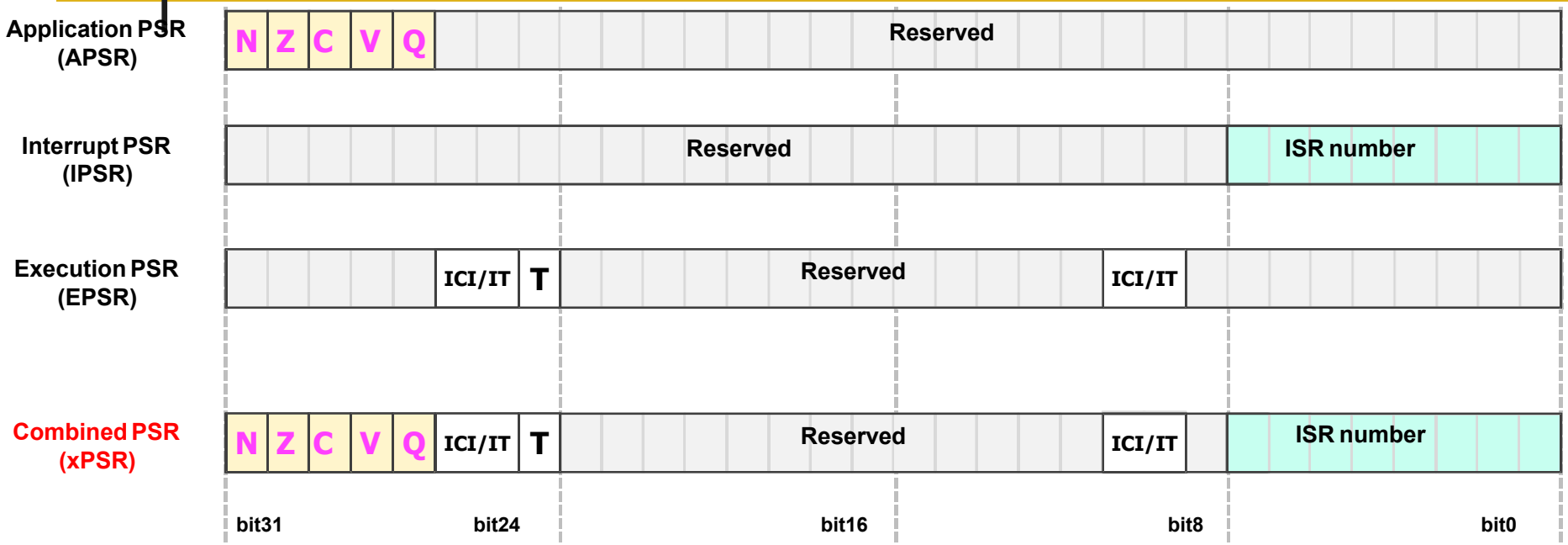
Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Negative	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Signed Greater or equal	N=V
LT	Signed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

Note AL is the default and does not need to be specified

Combined Program Status Registers (xPSR)



Combined Program Status Registers (xPSR)



XPSR = APSR + IPSR + EPSR

Example: GE - Signed Greater or Equal (N == V)

CMP r0, r1

We in fact perform subtraction r0 – r1, without saving the result.

	N = 0	N = 1
V = 0	• No overflow, implying the result is correct. • The result is non-negative, • Thus r0 – r1 ≥ 0, i.e., r0 ≥ r1	• No overflow, implying the result is correct. • The result is negative. • Thus r0 – r1 < 0, i.e., r0 < r1
V = 1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as non-negative and in fact it should be negative. • Thus r0 – r1 < 0 in reality, i.e., r0 < r1	• Overflow occurs, implying the result is incorrect. • The result is mistakenly reported as negative and in fact it should be non-negative. • Thus r0 – r1 ≥ 0 in reality., i.e. r0 ≥ r1

Conclusions:

- If N == V, then it is signed greater or equal (GE).
- Otherwise, it is signed less than (LT)

Branch Instructions

Instruction	Operands	Brief description	Flags
B	label	Branch	-
BL	label	Branch with Link	-
BLX	Rm	Branch indirect with Link	-
BX	Rm	Branch indirect	-

- B label: causes a branch to label.
- BL label: instruction copies the address of the next instruction into **r14** (**lr**, the link register), and causes a branch to label.
- BX Rm: branch to the address held in Rm
- BLX Rm: copies the address of the next instruction into **r14** (**lr**, the link register) and branch to the address held in Rm

Branch With Link

- The "Branch with link (BL)" instruction implements a subroutine call by writing PC – 4 into the LR of the current bank.
  - i.e. the address of the next instruction following the branch with link instruction (allowing for the pipeline).
- To return from subroutine, simply need to restore the PC from the LR:
  - `MOV pc, lr`
  - Again, pipeline has to refill before execution continues.
- The "Branch" instruction does not affect LR.

Signed vs. Unsigned

Conditional codes applied to branch instructions

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

Branch Instructions

	Instruction	Description	Flags tested
Unconditional Branch	B label	Branch to label	
Conditional Branch	BEQ label	Branch if EQual	Z = 1
	BNE label	Branch if Not Equal	Z = 0
	BCS/BHS label	Branch if unsigned Higher or Same	C = 1
	BCC/BLO label	Branch if unsigned LOwer	C = 0
	BMI label	Branch if MInus (Negative)	N = 1
	BPL label	Branch if PLus (Positive or Zero)	N = 0
	BVS label	Branch if oVerflow Set	V = 1
	BVC label	Branch if oVerflow Clear	V = 0
	BHI label	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS label	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE label	Branch if signed Greater or Equal	N = V
	BLT label	Branch if signed Less Than	N != V
	BGT label	Branch if signed Greater Than	Z = 0 & N = V
	BLE label	Branch if signed Less than or Equal	Z = 1 or N = !V

Which is Greater: 0xFFFFFFFF or 0x00000001?

It's **software's responsibility** to tell computer how to interpret data:

- If written in C, declare the signed vs unsigned variable
- If written in Assembly, use signed vs unsigned branch instructions

```
signed int x, y ;
x = -1;
y = 1;
if (x > y)
   ...
```

```
    MOVS r6, #0xFFFFFFFF ;x
    MOVS r5, #0x00000001 ;y
    CMP  r5, r6
    BLE  Then_Clause
    ...
```

Check Z=1 or N!=V

BLE: Branch if less than or equal, signed ≤

```
unsigned int x, y ;
x = 4294967295;
y = 1;
if (x > y)
   ...
```

```
    MOVS r6, #0xFFFFFFFF ;x
    MOVS r5, #0x00000001 ;y
    CMP  r5, r6
    BLS  Then_Clause
    ...
```

Check C=0 or Z=1

BLS: Branch if lower or same, unsigned ≤

Combination

Instruction	Operands	Brief description	Flags
CBZ	Rn, label	Compare and Branch if Zero	-
CBNZ	Rn, label	Compare and Branch if Non Zero	-

- Except that it does not change the condition code flags, CBZ Rn, label is equivalent to:

```
CMP Rn, #0
BEQ label
```

- Except that it does not change the condition code flags, CBNZ Rn, label is equivalent to:

```
CMP  Rn, #0
BNE  label
```

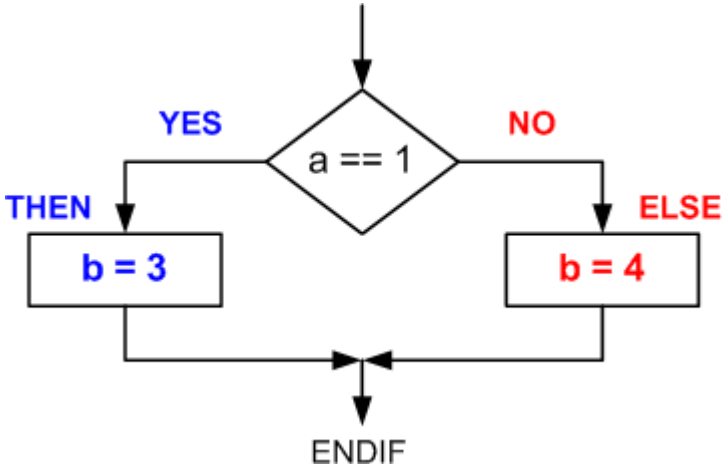
Conditional Execution

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

If-then-else

C Program

```
if (a == 1)
    b = 3;
else
    b = 4;
```

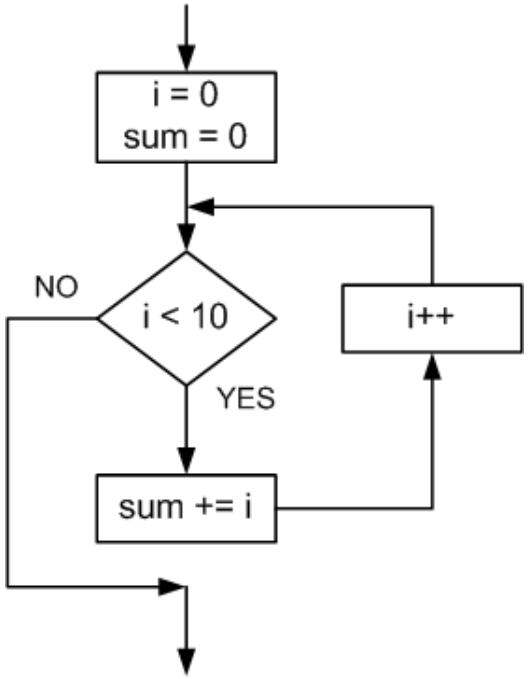



```
        ; r1 = a, r2 = b
        CMP r1, #1    ; compare a and 1
        BNE else      ; go to else if a ≠ 1
then    MOV r2, #3    ; b = 3
        B   endif     ; go to endif
else    MOV r2, #4    ; b = 4
endif
```

BNE: Branch not equal, Z=0

For Loop

```
C Program
int i;
int sum = 0;
for(i = 0; i < 10; i++){
  sum += i;
}
```



Implementation 1:

```
        MOV r0, #0   ; i
        MOV r1, #0   ; sum

        B   check
loop    ADD r1, r1, r0
        ADD r0, r0, #1
check   CMP r0, #10
        BLT loop
endloop
```

Example 1: Greatest Common Divider (GCD)

Euclid's Algorithm

```
uint32_t a, b;
while (a != b ) {
    if (a > b)
       a = a – b;
    else
       b = b – a;
}
```

```
gcd   CMP r0, r1
      SUBHI r0, r0, r1
      SUBLO r1, r1, r0
      BNE gcd
```

```
          ; suppose r0 = a and r1 = b
gcd       CMP r0, r1          ; a > b?
          BEQ end             ; if a = b, done
          BLO less            ; a < b
          SUB r0, r0, r1      ; a = a – b
          B   gcd
less      SUB r1, r1, r0      ; b = b – a
          B   gcd
end
```

Break and Continue

C Program	Assembly Program

`int i;`
`int sum = 0;`
 	
`for(i = 0; i < 10; i++) {`
`  if (i == 5)`
`  continue;`
`  sum += i;`
`}`
	`  MOVS  r0, #0  ; i = 0`
	`  MOVS  r1, #0  ; sum = 0`
	
	`loop  CMP  r0, #10`
	`  BGE  endloop`
	`  CMP  r0, #5`
	`  ADDNE  r1, r1, r0  ; sum += i`
	`  ADD  r0, r0, #1  ; i++`
	`  B  loop`
	`endloop`

Switch Statement Example

C Program

```
unsigned int score;
char grade;

switch(score){
    case 10:

    case 9:
      grade = 'A';
      break;

    case 8:
      grade = 'B';
      break;

    case 7:
      grade = 'C';
      break;

    case 6:
      grade = 'D';
      break;

    default:
      grade = 'F';
      break;

}
```

Assembly Program

```
        ; r0 = numeric score (0 ≤ r0 ≤10)
        ; r1 = letter grade

        CMP  r0, #5
        BLS  default    ; branch if unsigned r0 ≤ 5
        SUBS r2, r0, #6    ; r2 holds branch index

        ; pc = pc + 4 + 2 x BranchTable[r2]
        TBB [pc, r2]

BranchTable
        DCB (case_6 - BranchTable)/2     ; index = 0
        DCB (case_7 - BranchTable)/2     ; index = 1
        DCB (case_8 - BranchTable)/2     ; index = 2
        DCB (case_10_9 - BranchTable)/2  ; index = 3
        DCB (case_10_9 - BranchTable)/2  ; index = 4

        ALIGN

case_10_9
        MOV  r1, #0x41     ; ASCII 'A' = 0x41
        B    exit

case_8
        MOV  r1, #0x42     ; ASCII 'B' = 0x42
        B    exit
```

Switch Statement Example Cont'd

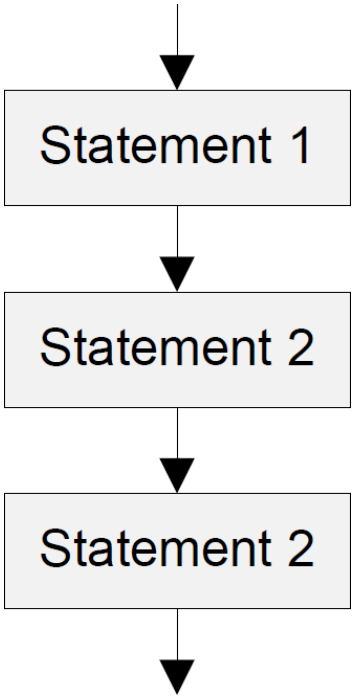
C Program	Assembly Program

unsigned int score;
char grade;
 	
switch(score){
  case 10:
 	
  case 9:
  grade = 'A';
  break;
 	
  case 8:
  grade = 'B';
  break;
 	
  case 7:
  grade = 'C';
  break;
 	
  case 6:
  grade = 'D';
  break;
 	
  default:
  grade = 'F';
  break;
 	
}
	case_7
	  MOV r1, #0x43 ; ASCII 'C' = 0x43
	  B exit
	
	case_6
	  MOV r1, #0x44 ; ASCII 'D' = 0x44
	  B exit
	
	default
	  MOV r1, #0x46 ; ASCII 'F' = 0x46
	
	exit

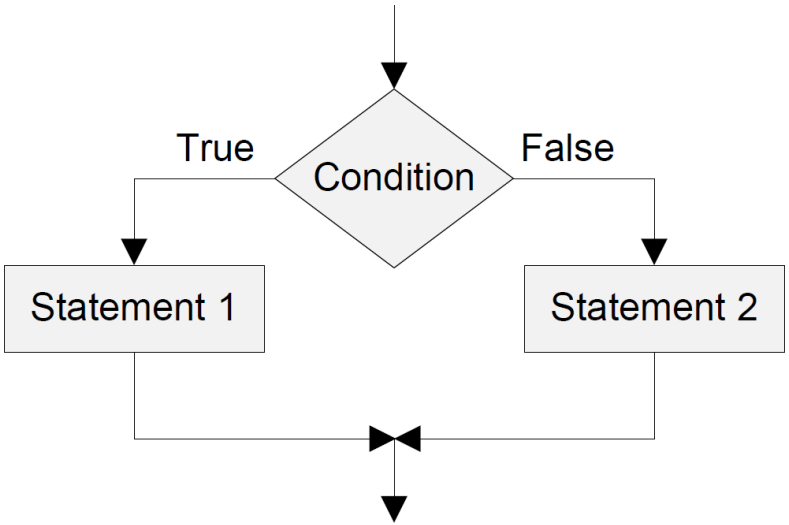
Chapter 7
Structured Programming

Edited: Waqas Majeed

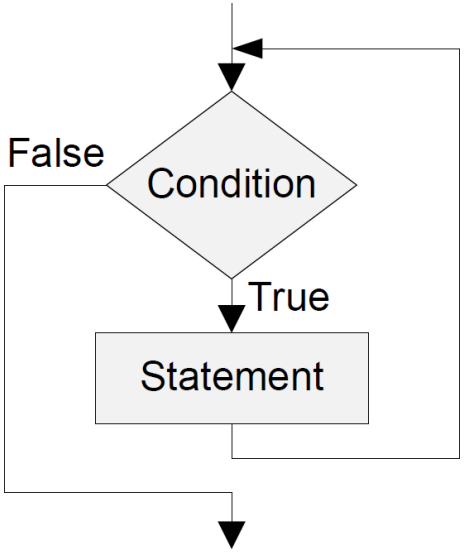
Basic Control Structures



Sequence Structure



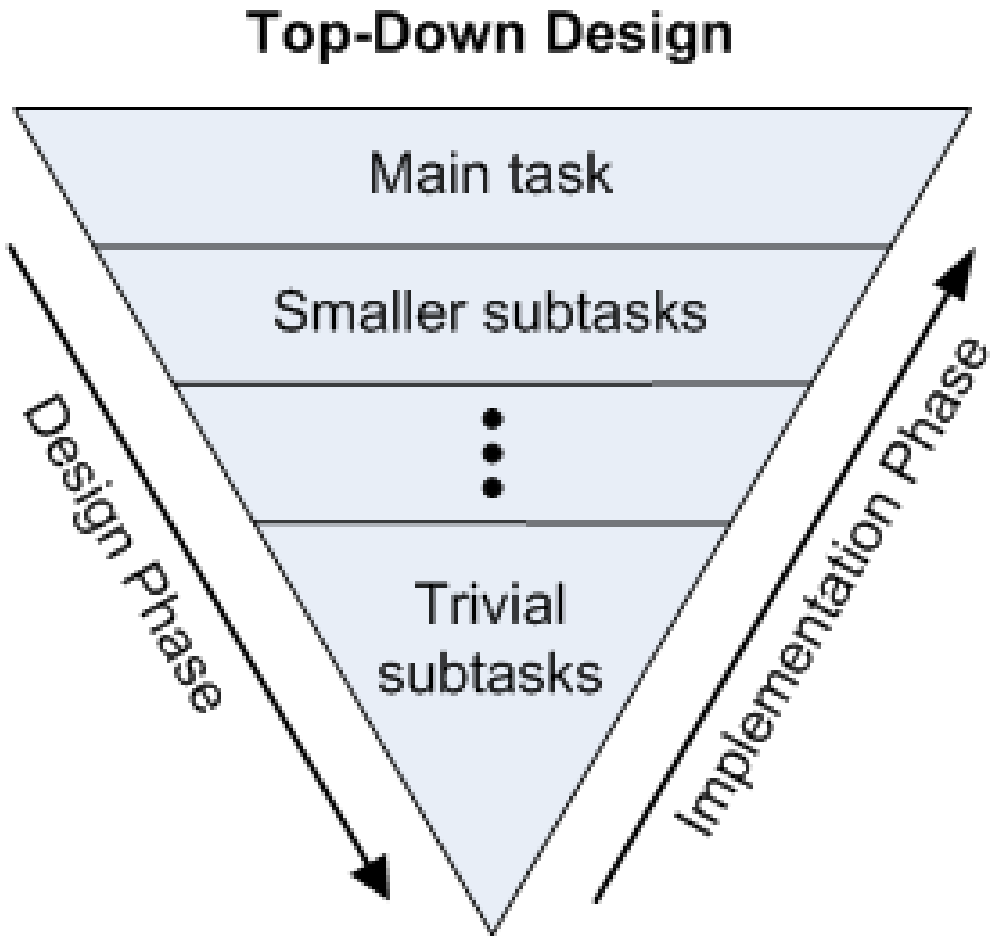
Selection Structure



Loop Structure

- Each structure has only 1 entry point and 1 exit point.
- Control structure can be nested or embedded in another structure to form a compound structure.

Top-Down Design



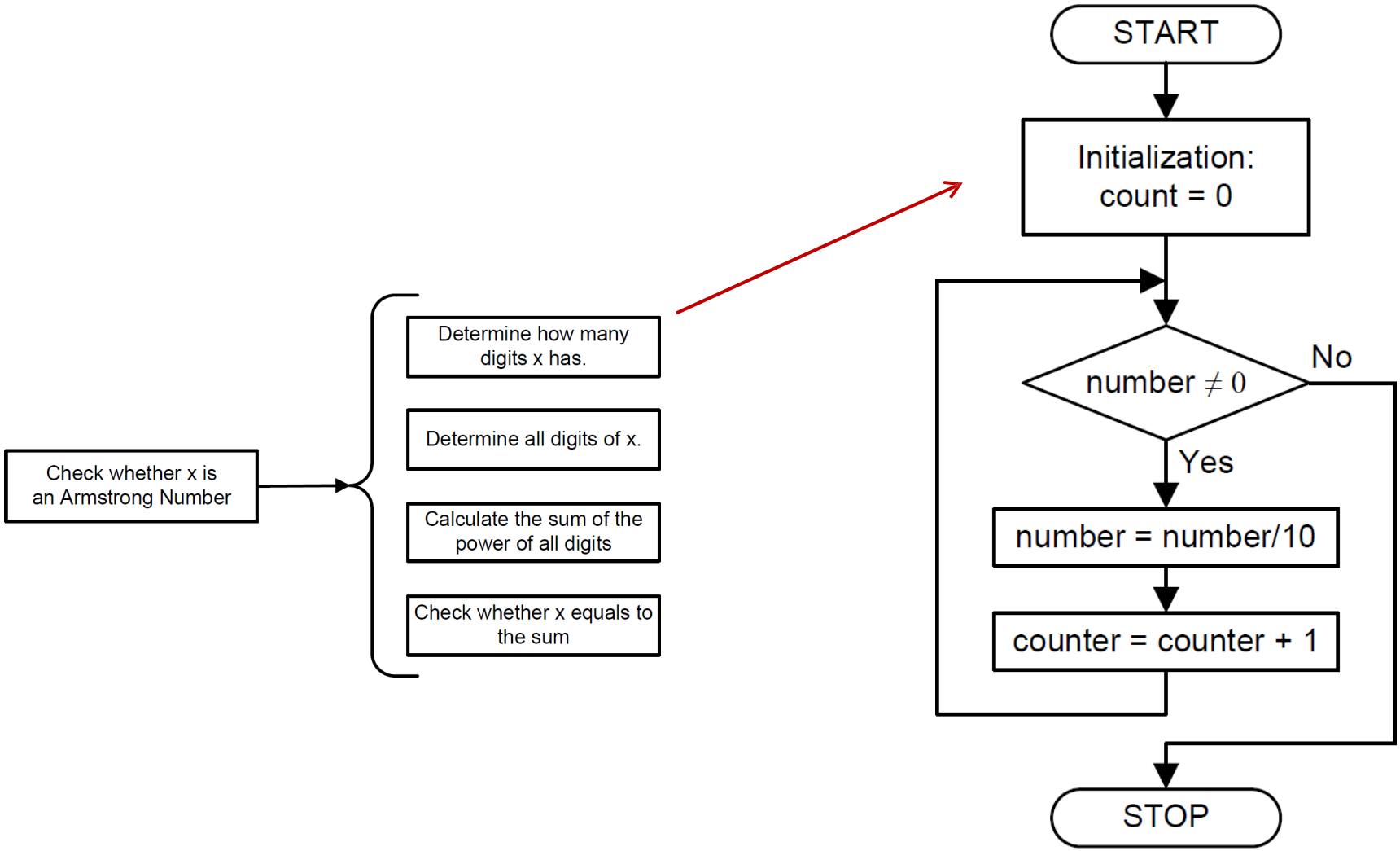
Structured Programming

- Programming using nested control structures as well as subroutines in high level languages is called **structured programming**.

Structured Programming in Assembly

- Structured programming in assembly involves 2 steps:
  1. Top-down Logical Design – Perform stepwise refinement and construct program flow using high level structures.
  2. Implement the high level structures identified in Step 1 in assembly language.

Top-Down Design Example:
Counting digits



Top-Down Design Example: Program Implementation

C Program	Assembly Program

`int main(void) {`
`  int number, sum, r;`
`  int flag, t;`
	`AREA Armstrong, CODE, READONLY`
	`EXPORT __main`
	`ENTRY`
	`__main PROC`

`  number = 371;`
`  sum = 0;`
`  t = number;`
	`LDR  r0, =371  ; number to be checked`
	`MOV  r4, #0  ; sum = 0`
	`MOV  r1, r0  ; save a copy, r1=number`

`  while( t != 0  ) {`
`  r = t % 10;`
`  sum = sum + r*r*r;`
`  t = t / 10;`
`  }`
	`MOV  r3, #10`
	`loop  CBZ  r1, check  ; if t = 0, exit loop`
	`SDIV  r6, r1, r3  ; r2 = remainder`
	`MLS  r2, r3, r6, r1 ; r2 = r1 – 10*r6`
	`MUL  r3, r2, r2  ; remainder^2`
	`MLA  r4, r2, r2, r4 ; r4 = r2*r2 + r4`
	`MOVS  r3, #10`
	`SDIV  r1, r1, r3  ; t = t/10`
	`CBNZ  loop`

`  if( number == sum )`
`  flag = 1;`
`  else`
`  flag = 0;`
	`check  CMP  r0, r4`
	`yes  MOVEQ  r0, #1  ; Armstrong`
	`no  MOVNE  r0, #0  ; not Armstrong`

`  while(1);`
`}`
	`stop  B  stop`
	`ENDP`
	`END`

Register Reuse

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r4, =C
    LDR r5, [r4]
    LDR r6, =D
    LDR r7, [r6]
    ADD r1, r3, r5
    SUB r1, r1, r7
    LDR r0, =A
    STR r1, [r0]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

Lifetime of r2 (LDR r2, =B … LDR r3, [r2])
Lifetime of r4 (LDR r4, =C … LDR r5, [r4])
Lifetime of r6 (LDR r6, =D … LDR r7, [r6])
Lifetime of r3 (LDR r3, [r2] … ADD r1, r3, r5)
Lifetime of r0 (LDR r0, =A … STR r1, [r0])

8 registers used

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r2, =C
    LDR r5, [r2]
    LDR r2, =D
    LDR r7, [r2]
    ADD r3, r3, r5
    SUB r3, r3, r7
    LDR r2, =A
    STR r3, [r2]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

Lifetime of r2 (LDR r2, =B … LDR r7, [r2])
Lifetime of r2 (LDR r2, =A … STR r3, [r2])

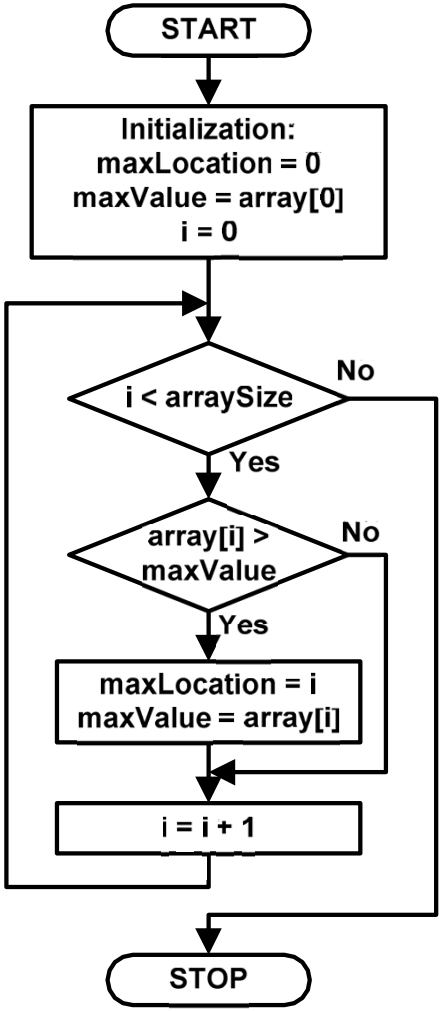
4 registers used

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r2, =C
    LDR r5, [r2]
    LDR r2, =D
    LDR r2, [r2]
    ADD r3, r3, r5
    SUB r3, r3, r2
    LDR r2, =A
    STR r3, [r2]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

Reuse r2 (LDR r2, [r2] … SUB r3, r3, r2)

3 registers used

Finding Max of an Array



Finding Max of an Array

C Program	Assembly Program

`int array[10] = {-1, 5, 3, 8, 10, 23, 6, 5, 2, -10};`
 	
`int size = 10;`
	`AREA myData, DATA`
	`ALIGN`
	`array DCD -1,5,3,8,10,23,6,5,2,-10`
	`size DCD 10`

`int main(void) {`
`  int i, maxLocation, maxValue;`
	`AREA findMax, CODE`
	`EXPORT __main`
	`ALIGN`
	`ENTRY`
	`__main PROC`
	`; Identify the array size`
	`LDR r3, =size`
	`LDR r3, [r3] ; array size`
	`SUB r3, r3, #1`

`// Initialize max and location`
`maxLocation = 0;`
`maxValue = array[0];`
	`; Initialize max value and location`
	`LDR r4, =array`
	`LDR r0, [r4] ; r0 = default max`
	`MOV r1, #0 ; r1 = max location`

`// loop through the array`
`for (i = 0; i < size; i++){`
`  if (array[i] > maxValue) {`
`  maxValue = array[i];`
`  maxLocation = i;`
`  }`
`}`
	`; loop over the array`
	`MOV r2, #0 ; loop index i`
	`loop CMP r2, r3 ; compare i & size`
	`BGE stop ; stop if i ≥ size`
	`LDR r5, [r4,r2,LSL #2] ; array[i]`
	`CMP r5, r0 ; compare with max`
	`MOVGT r0, r5 ; update max value`
	`MOVGT r1, r2 ; update location`
	`ADD r2, r2, #1 ; update index i`
	`B loop`

`while(1); //dead loop`
`}`
	`stop B stop ; dead loop`
	`ENDP`
	`END`

Chapter 8
Subroutines

Edited: Waqas Majeed

Introduction

- 2 special considerations when using subroutines in assembly
  1. Preserve and recover the caller's environment.
     - Subroutine must push registers at beginning of subroutine.
     - Subroutine must pop registers at end of subroutine.
  2. Embedded application binary interface (EABI) has specifications for passing input arguments to a subroutine and returning the result back to the caller.
     - Follow EABI to make assembly subroutines compatible with C programs.

Calling a Subroutine

- Assembly program can use branch and link (BL) instruction to call a subroutine.
- BL instruction performs 2 operations:
  1. Store memory address of instruction after BL instruction in LR (PC + 4) → referred to as **return address**
  2. Place memory address of 1st instruction of subroutine in PC.

BL and BX

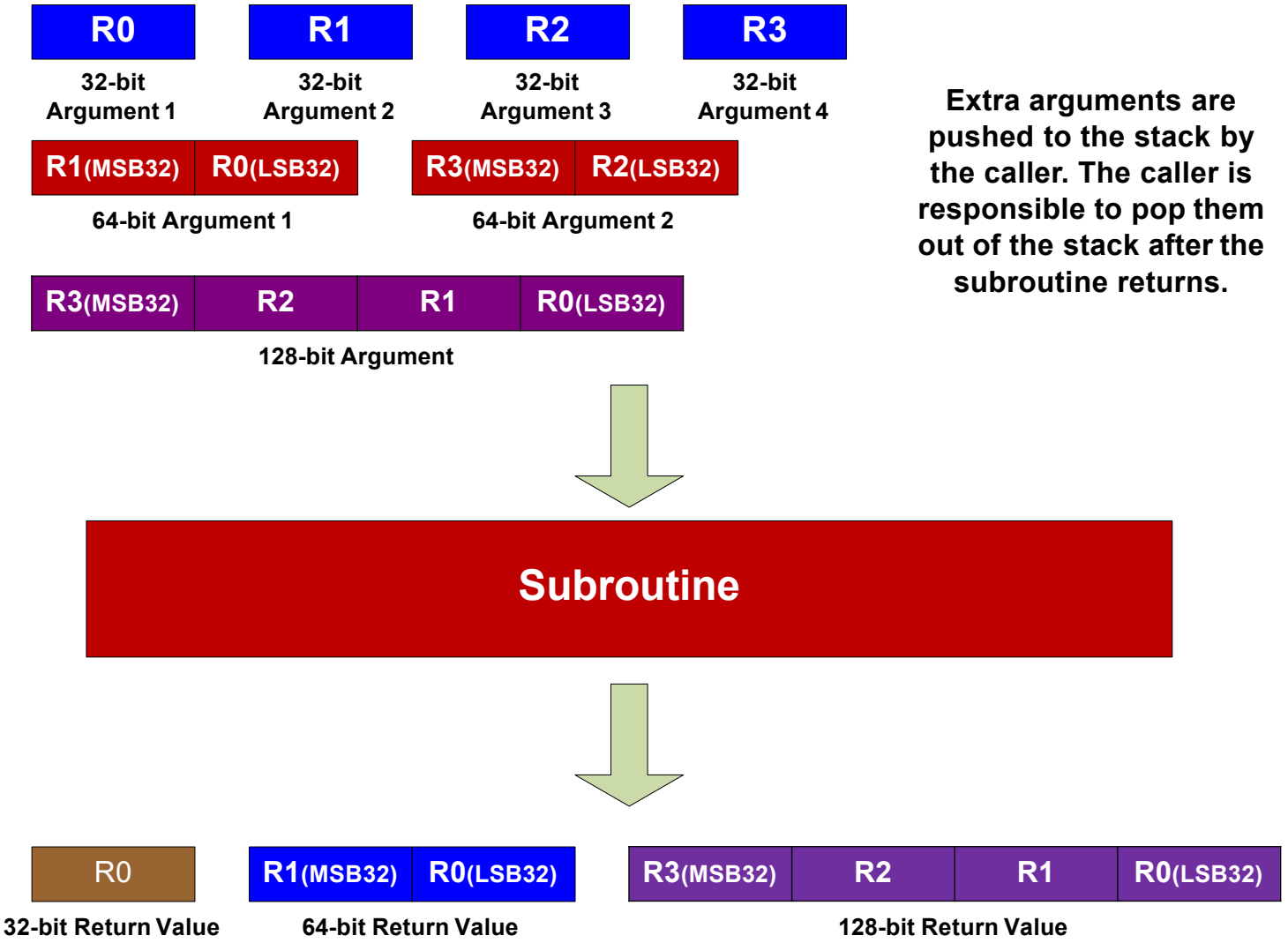
```
void enable(void) ;


   • • •
   enable() ;
   • • •
```

Compiler

```
  • • •
BL enable
  • • •
```

```
export  enable
enable
          • • •
          • • •
          BX  LR
```

Passing Arguments via Registers

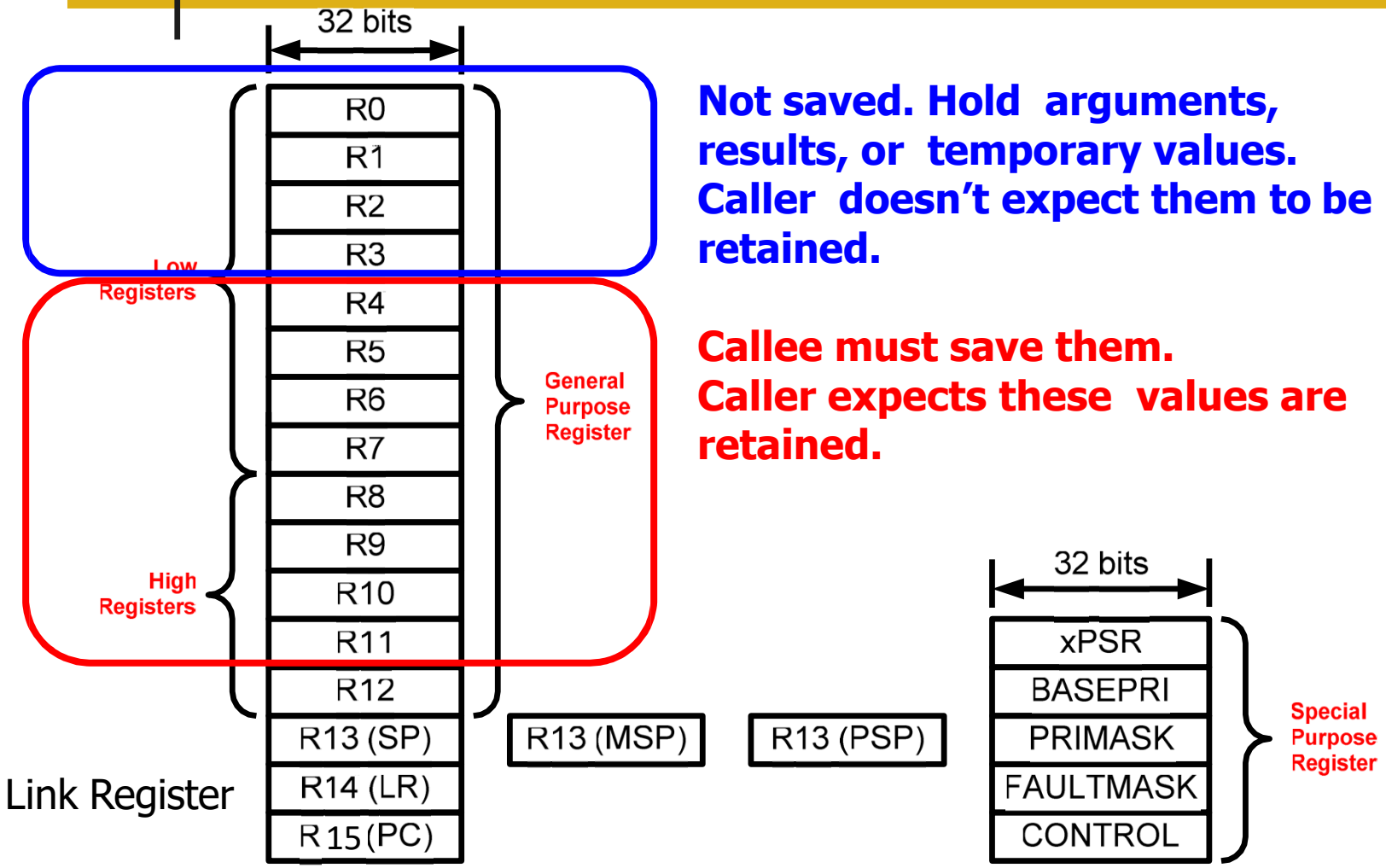


Extra arguments are pushed to the stack by the caller. The caller is responsible to pop them out of the stack after the subroutine returns.

ARM Procedure Call Standard

Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	No	
r2	Argument 3	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	No	If more than 4 arguments, use the stack
r4	General-purpose V1	Yes	Variable register 1 holds a local variable.
r5	General-purpose V2	Yes	Variable register 2 holds a local variable.
r6	General-purpose V3	Yes	Variable register 3 holds a local variable.
r7	General-purpose V4	Yes	Variable register 4 holds a local variable.
r8	General-purpose V5	YES	Variable register 5 holds a local variable.
r9	Platform specific/V6	No	Usage is platform-dependent.
r10	General-purpose V7	Yes	Variable register 7 holds a local variable.
r11	General-purpose V8	Yes	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	No	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Yes	SP has to be the same after a subroutine has completed.
r14 (LR)	Link register	No	LR does not have to contain the same value after a subroutine has completed.
r15 (PC)	Program counter	N/A	Do not directly change PC

Link Register

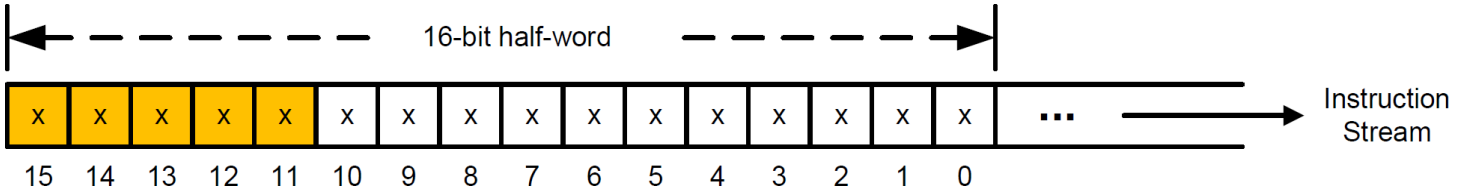


Not saved. Hold arguments, results, or temporary values. Caller doesn't expect them to be retained.

Callee must save them. Caller expects these values are retained.

Realities

- PC is always incremented by **4**.
  - Each time, 4 bytes are fetched from the instruction memory
  - It is either two 16-bit instructions or one 32-bit instruction

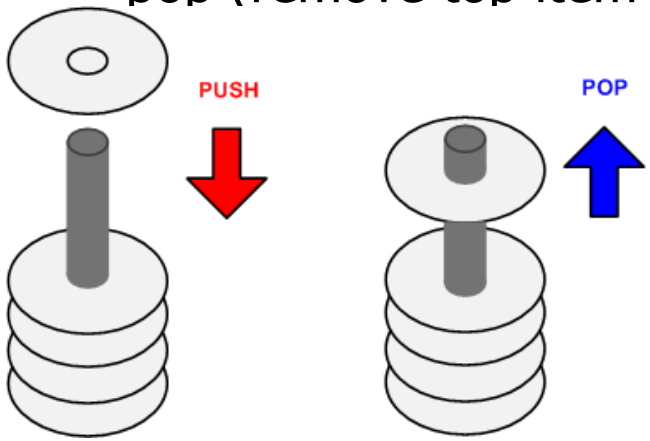


If bit [15-11] = `11101`, `11110`, or `11111`, then, it is the first half-word of a 32-bit instruction. Otherwise, it is a 16-bit instruction.

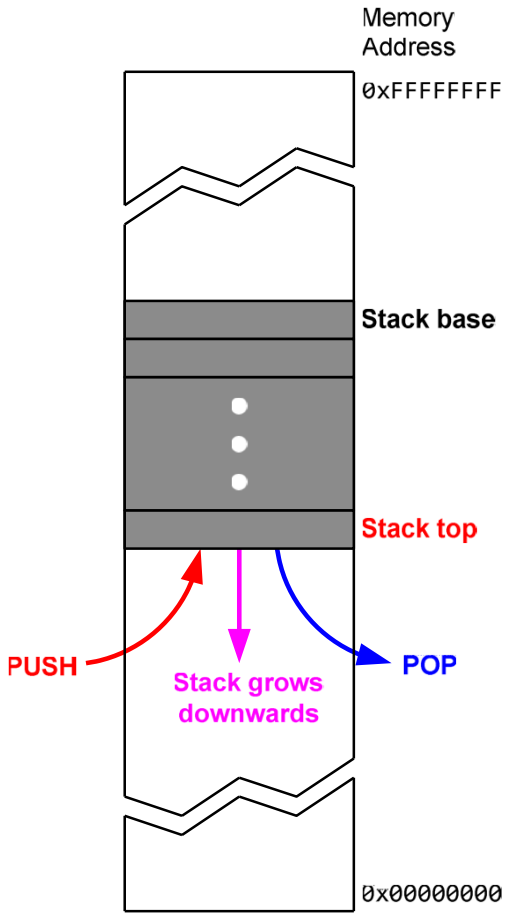
- The least significant bit of LR is always **1** for ARM Cortex-M
  - This bit is used to control the processor mode:
    - 0 = ARM, 1 = THUMB
  - Cortex-M only supports THUMB.

Stack

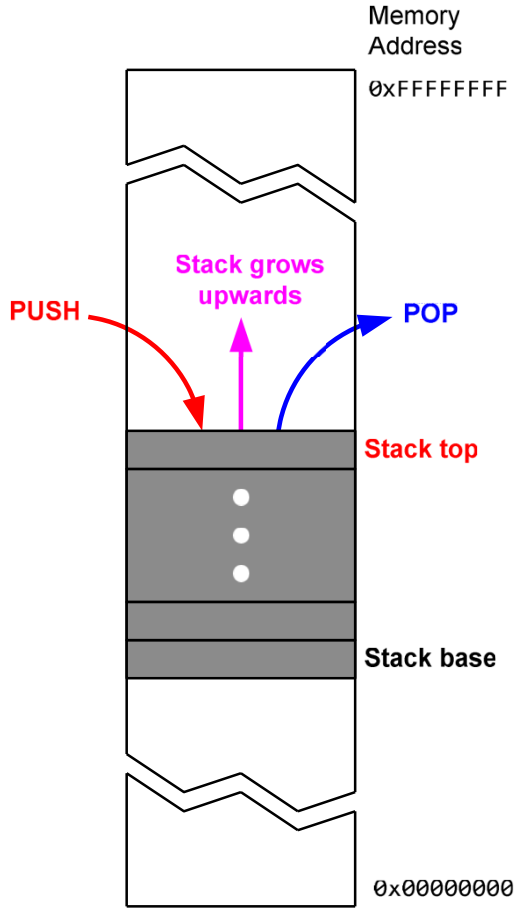
- A **Last-In-First-Out** data structure
- Only allow to access the most recently added item
  - Also called the top of the stack
- Key operations:
  - push (add item to stack)
  - pop (remove top item from stack)



Stack Growth Convention: Ascending vs Descending

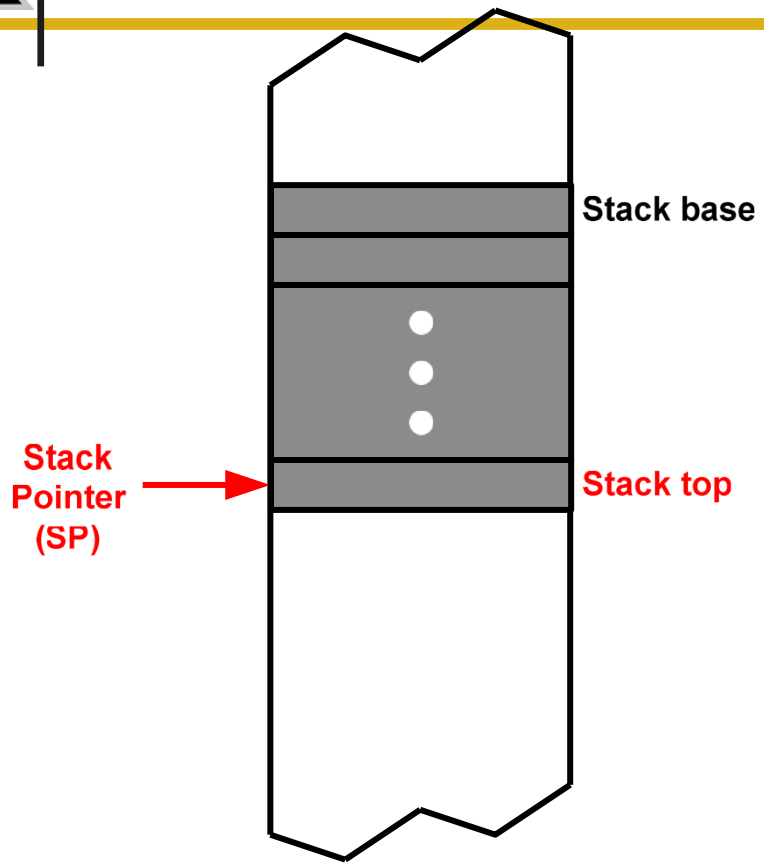


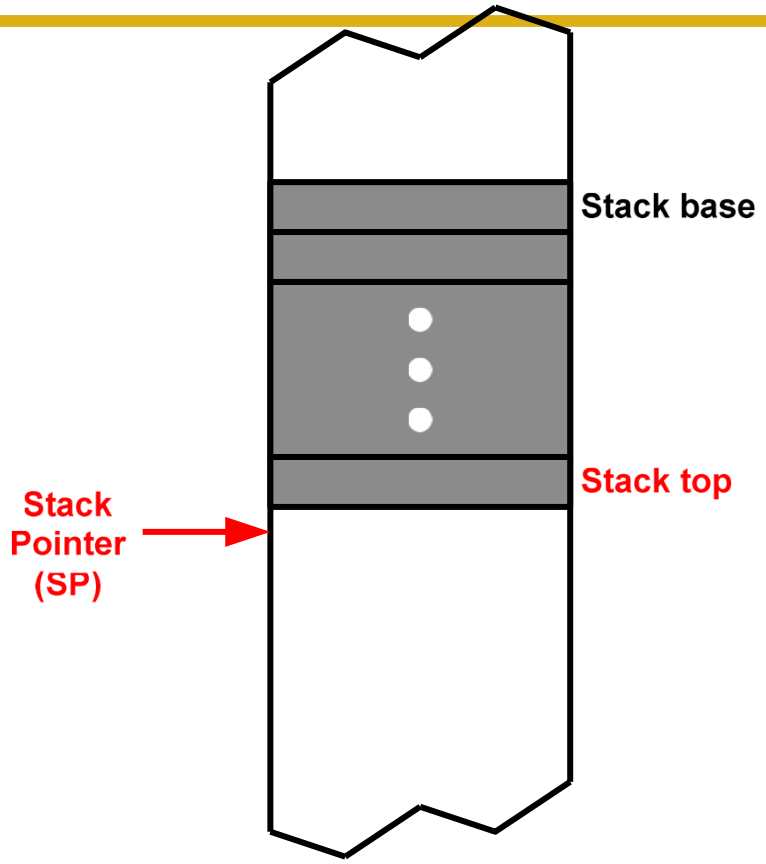
Descending stack: Stack grows towards low memory address



Ascending stack: Stack grows towards high memory address

Stack Growth Convention:
Full vs Empty



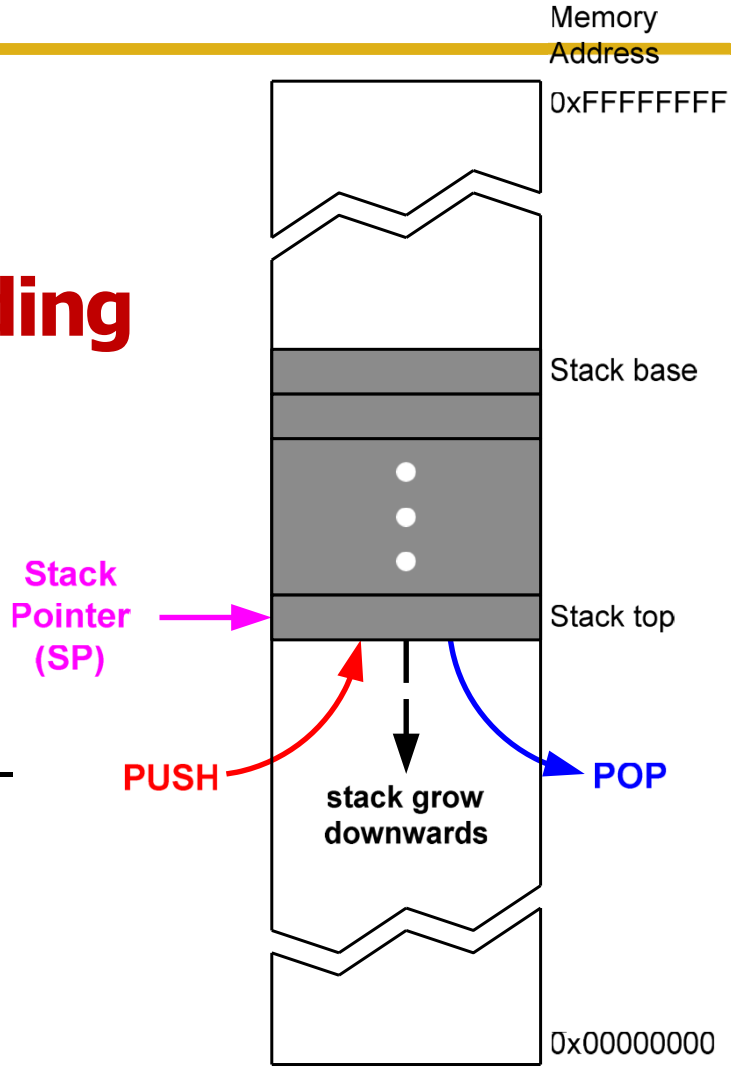


Full stack: SP points to the last item pushed onto the stack

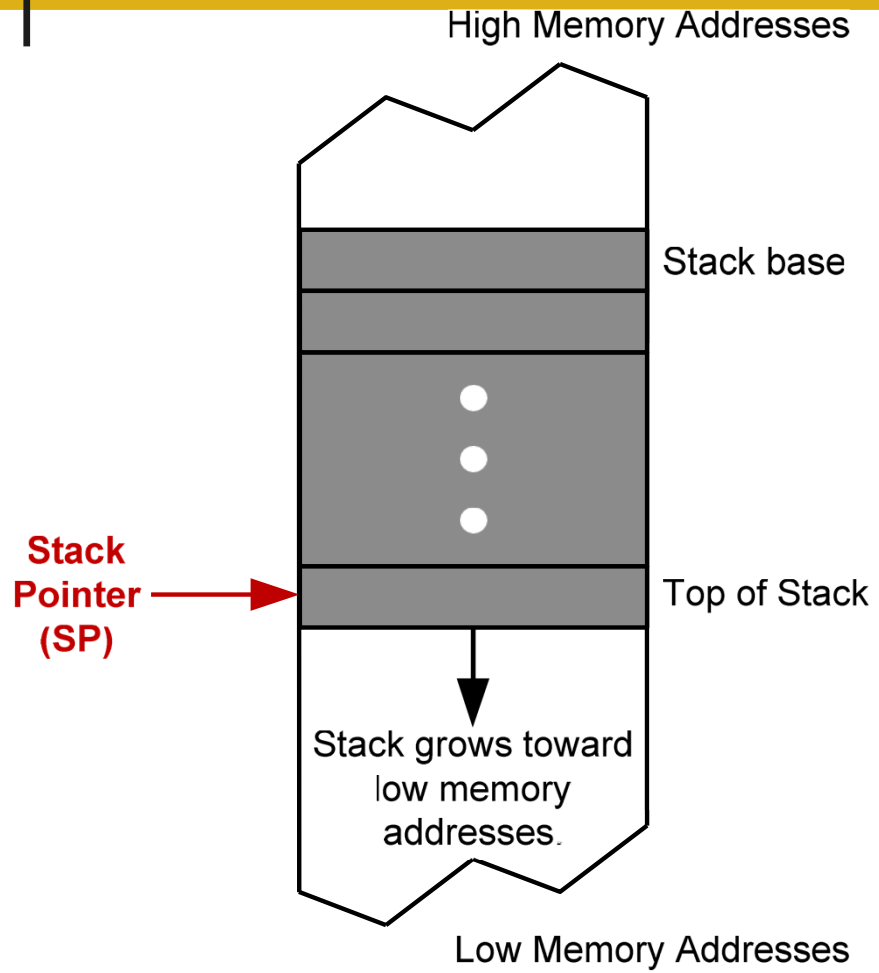
Empty stack: SP points to the next free space on the stack

Cortex-M Stack

- Stack pointer (SP) = R13
- Cortex-M uses **full descending stack**
- stack pointer
  - decremented on **PUSH**
  - incremented on **POP**
  - SP starts at `0x20000200` for STM32-Discovery



Full Descending Stack



PUSH {register_list}
equivalent to:
STMDB SP!, {register_list}

DB: Decrement Before

POP {register_list}
equivalent to:
LDMIA SP!, {register_list}

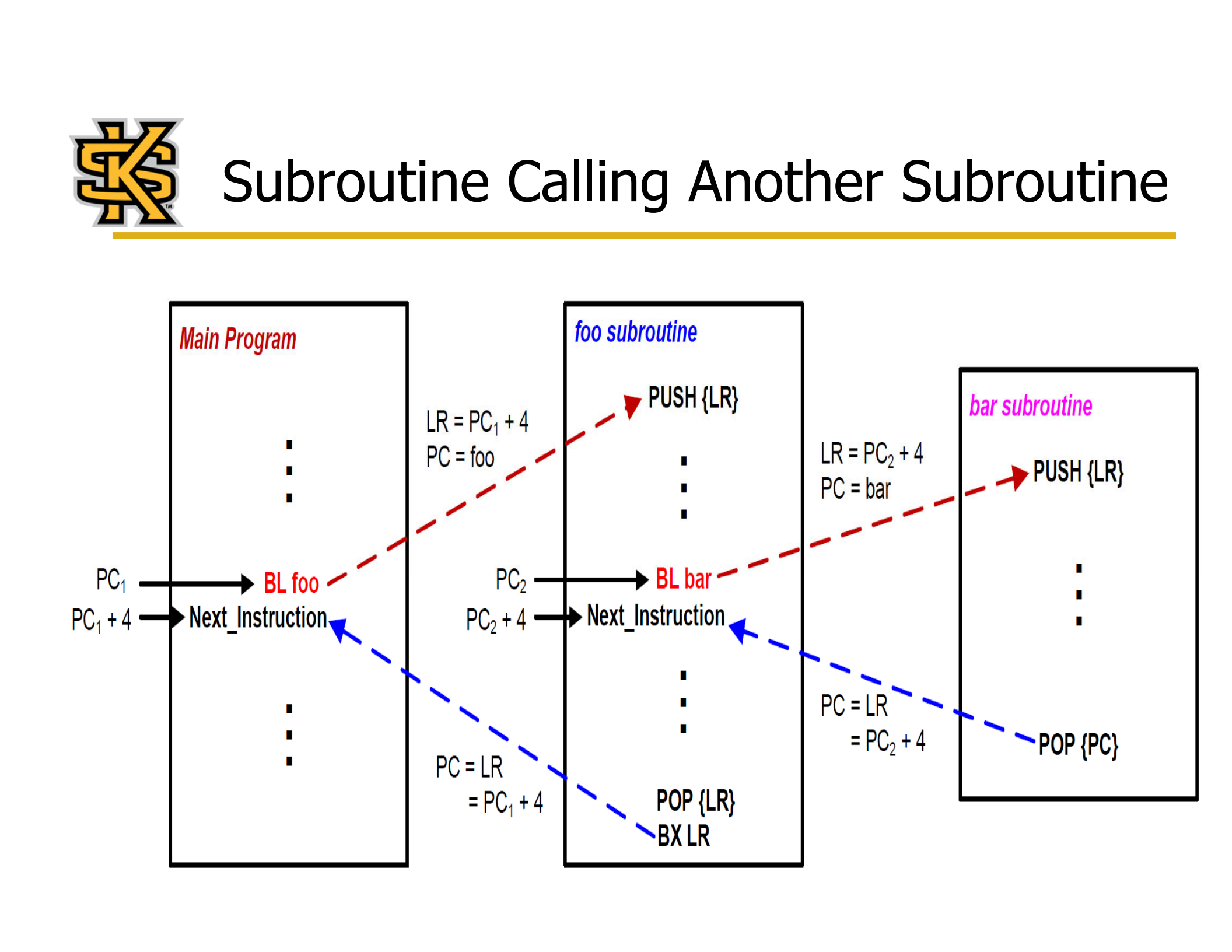
IA: Increment After

Stack Implementation via STM and LDM

Stock Name	Push		Pop	
	Equivalent	Alternative	Equivalent	Alternative
Full Descending(FD)	STMFD SP!,list	STMDB SP!,list	LDMFD SP!,list	LDMIA SP!,list
Empty Descending(ED)	STMED SP!,list	STMDA SP!,list	LDMED SP!,list	LDMIB SP!,list
Full Ascending(FA)	STMFA SP!,list	STMIB SP!,list	LDMFA SP!,list	LDMDA SP!,list
Empty Ascending(EA)	STMEA SP!,list	STMIA SP!,list	LDMEA SP!,list	LDMDB SP!,list

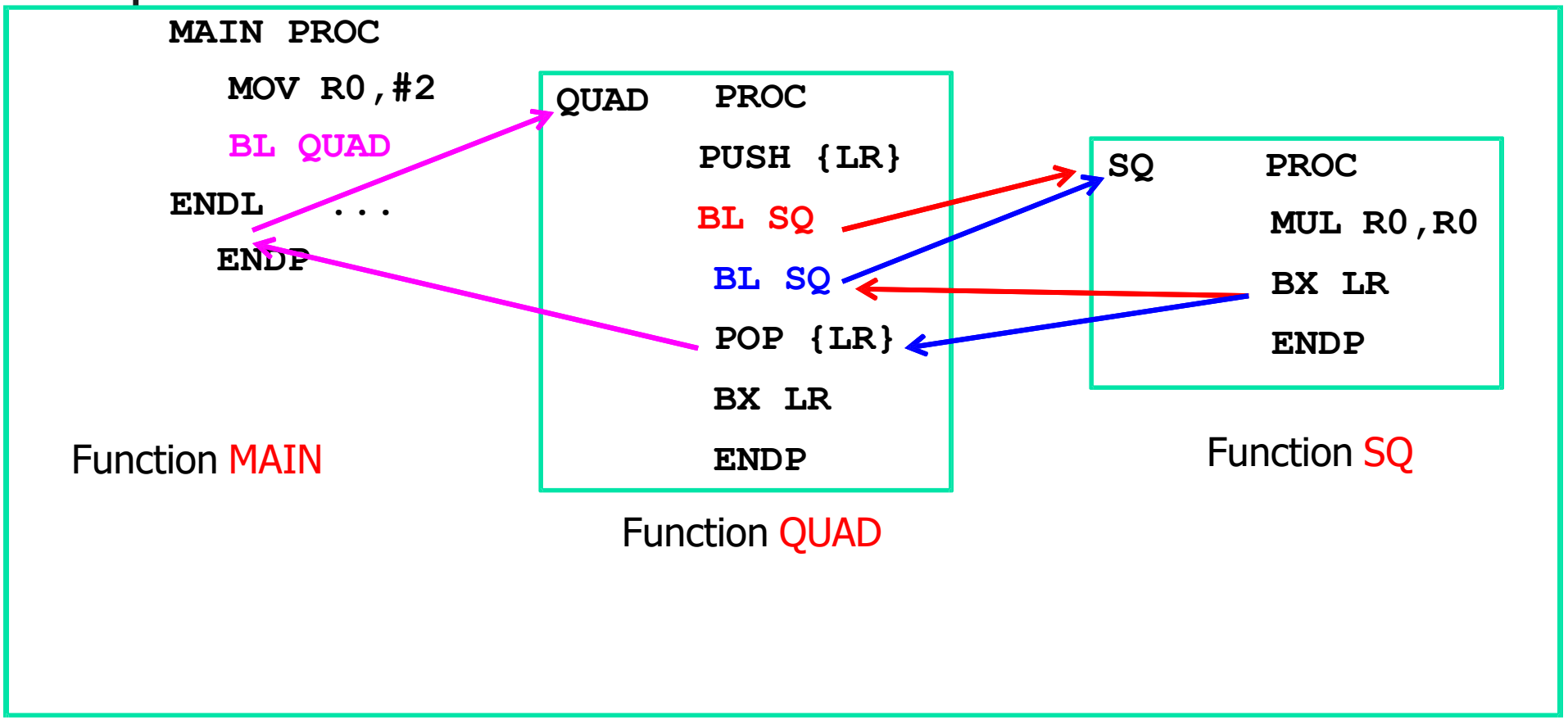
Stacks and Subroutines

- Stack can also be used to return from a subroutine.
- This is done by using PUSH {LR} at beginning of subroutine followed by POP {PC} at end of subroutine.
- This approach is required when a subroutine calls another subroutine.
- Under these circumstances, the LR will be overwritten in the caller subroutine.



Subroutine Calling Another Subroutine

Subroutine Calling Another Subroutine



2 Ways to Effectively Return Within Nested Subroutines

Method 1

```
PUSH {LR}
…
POP {LR}
BX LR
```

Method 2

```
PUSH {LR}
…
POP {PC}
```

Initializing the stack pointer (SP)

- Before using the stack, software has to define stack space and initialize the stack pointer (SP).
- The assembly file startup.s defines stack space and initialize SP.

Recursive Functions

- A recursive function is one that solves its task by calling itself on smaller pieces of data.
- An effective tactic is to
  - divide a problem into sub-problems of the same type as the original,
  - solve those sub-problems, and
  - combine the results

Recursion vs Iteration

- Any problem that can be solved recursively can also be solved iteratively (using loop).

- Recursive functions vs Iterative functions
  - Cons:
    - Recursive functions are slow
    - Recursive function take more memory
  - Pros
    - Recursive functions resembles the problem more naturally
    - Recursive function are easier to program, and debug.

Recursive Functions

- push LR (& working registers) onto stack before nested call
- pop LR (& working registers) off stack after nested return