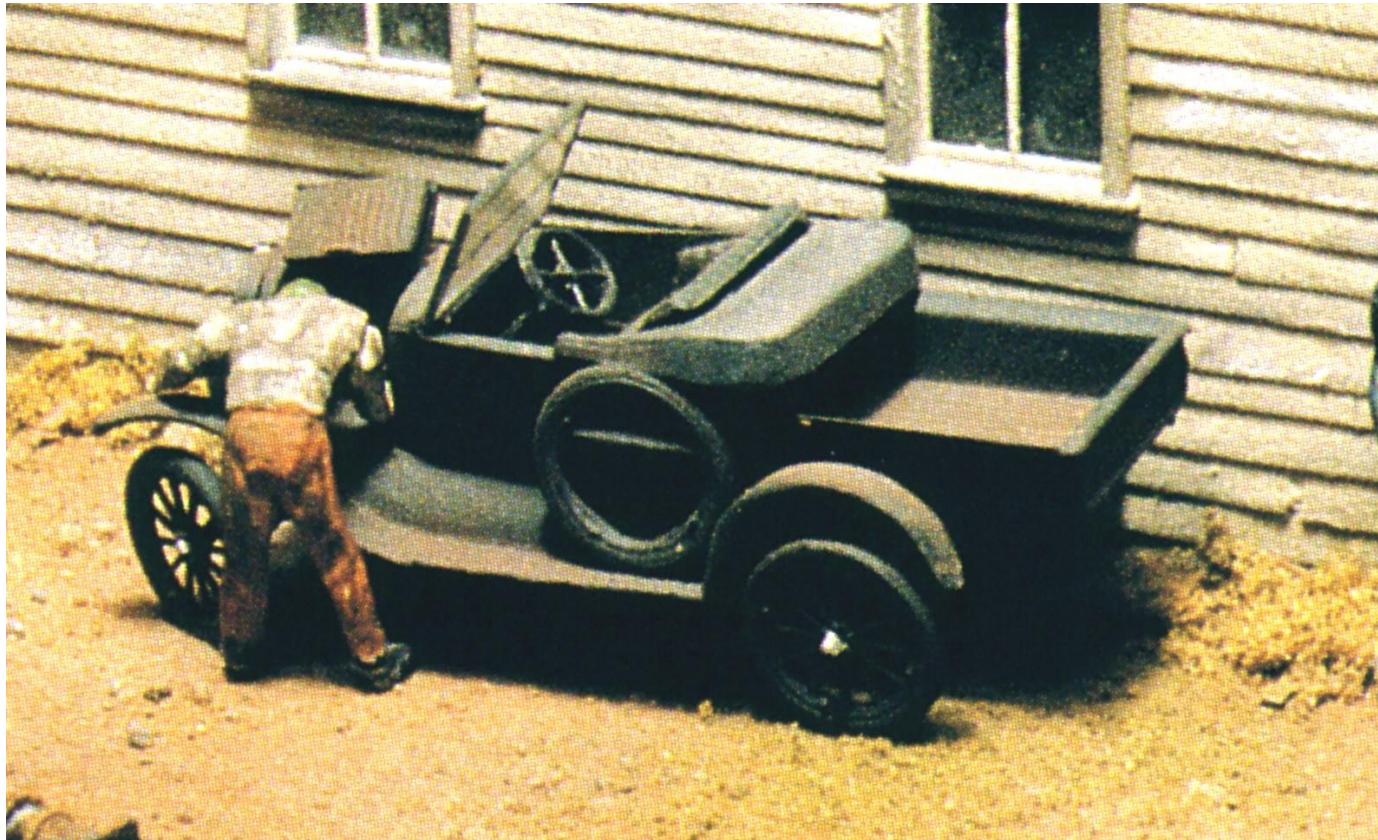


Computer Engineering

CT Team: A. Gieriet, J. Gruber, B. Koch, M. Loeser, M. Meli,
M. Rosenthal, M. Ostertag, A. Rüst, J. Scheier

Motivation

- See what's inside



Motivation

“I think there is a world market for maybe five computers.”

Thomas Watson, IBM, 1943

“Computers in the future may weigh no more than 1.5 tons.”

Popular Mechanics, 1949

“The number of transistors per IC doubles every year.”

Gordon Moore, Fairchild, 1965

“There is no reason for any individual to have a computer in his home.”

Ken Olson, DEC, 1977

“640K ought to be enough for anybody.”

Bill Gates, Microsoft, 1981

Todays Agenda

- What is Computer Engineering?
- Course Content and Organization
- Computer History
- Properties of a Computer System
- von Neumann Architecture
- Hardware Components
 - CPU, Memory, Input/Output, System Bus
- Software Aspects
 - from C to executable
- Interaction of Hardware and Software

What is Computer Engineering?

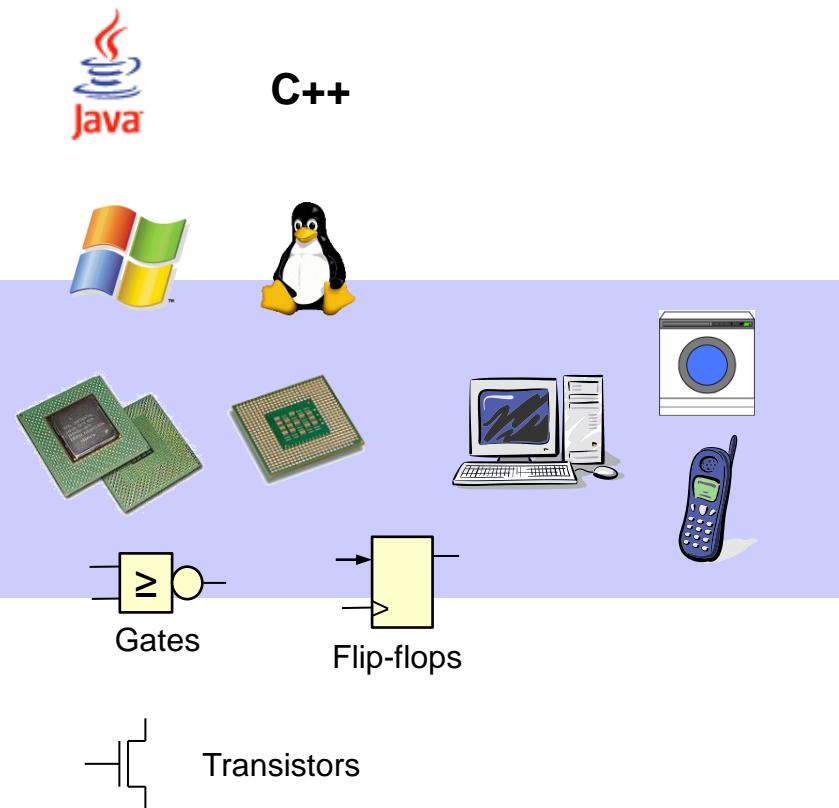
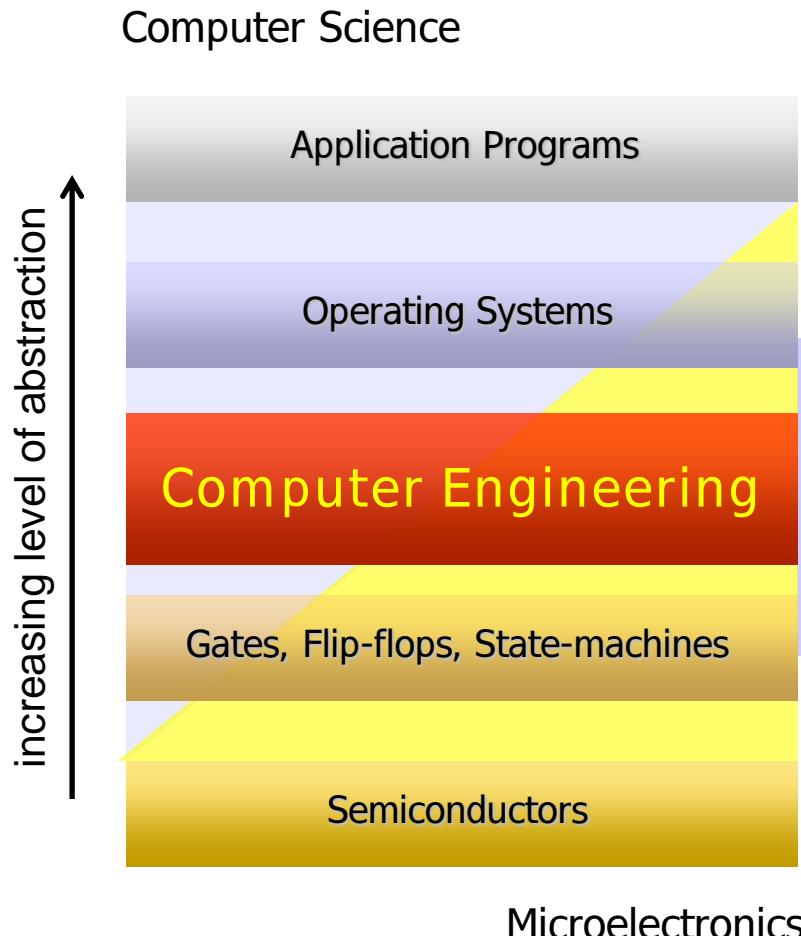
■ Computer Engineering (Technische Informatik)

- architecture and organization of computer systems
- combines hardware and software to implement a computer

■ Where Microelectronics and Software meet

- 70 years of computer hardware
 - 1940s relay / vacuum tubes
 - 1950s transistors
 - 1970s integrated circuits (CMOS¹)
- 40 years of software → Computer Science
 - Assembly Language ("Assembler")
 - High Level Language (e.g. C, ...)
 - Object Oriented Programming (C++, Java, ...)
 - Visual Programming (Model Driven Design)

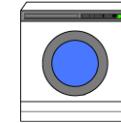
What is Computer Engineering?



Applications

■ Embedded Systems

- often part of a larger system
- control of devices, facilities, processes
- wireless sensor networks (WSN)



■ Information Technology

- communication networks
- processing of data
- multimedia



■ Tools

- support of technical and scientific activities
- simulation and modeling
- logging and analysis of measurement data



Objectives CT 1

After the course you will be able to

- describe the architecture and the operation of a basic computer system and a processor
- to explain how instructions are executed
- to describe the main architectures and performance features of processors as well as the concept of pipelining
- to comprehend how structures in C are compiled into executable object code and to use this knowledge to eliminate programming errors and to optimize program performance
- to develop, debug and verify basic hardware-oriented programs in C and in assembly language
- to explain the concept of interrupts and exceptions and to implement basic interrupt applications
- to find their way in other microprocessor systems

Course Content CT 1

■ Organization of computer systems

- Representation of information
- Program translation
- Architecture: CPU, Memory, I/O, Bus

■ CPU: Principle of Operation

- Instruction set
- Program execution
- Memory map, little endian vs. big endian

■ Data transfer

- Addressing modes
- Integer data types, arrays, pointers

■ Arithmetic and logic operations

- Computing with the ALU
- Integer casting

■ Control flow

- Compare and jump instructions
- Structured programming

■ Machine code

- Encoding of instructions and operands

■ Subroutines/functions

- Parameter passing

■ Exceptional Control Flow

- Hardware interrupts, interrupt service routine, vector table
- Exceptions (Traps)

■ Linking

- Address Resolution and Relocation
- Linker Map and Symbol Table
- Static Linking vs. Dynamic Linking

■ Processor architectures

- von Neumann vs. Harvard, Pipelining

■ Hardware-oriented programming labs

- Working with cross-compiler, assembler, linker, loader and debugger

Objectives for Today's Lesson

You will be able to

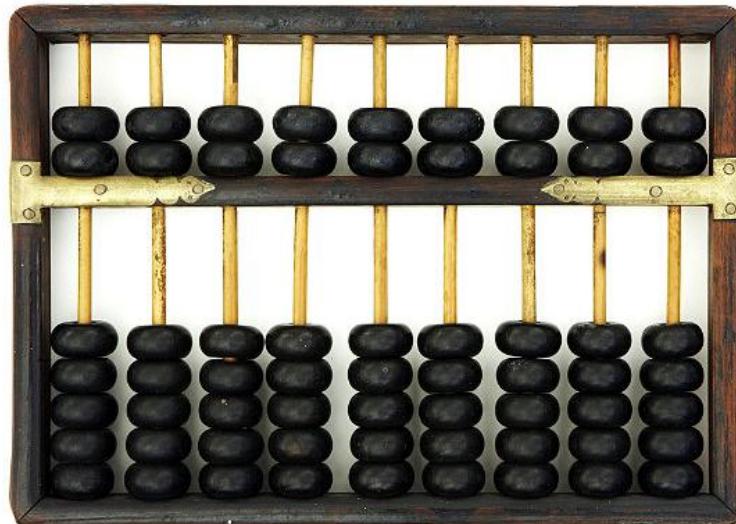
- outline and explain the function of a simple computer system
- name the four main hardware components of a computer system and to describe their functions
- describe different forms of memory and storage
- recall and explain the four translation steps from source code in C to an executable program
- comprehend the use of target and host during development
- explain why knowledge of assembly language is important

Computer History

■ Support for calculations

- Babylonian / Chinese
- John Napier

between 1000 und 500 BC: Abacus
beginning AD 1600:
tables for multiplications and logarithms



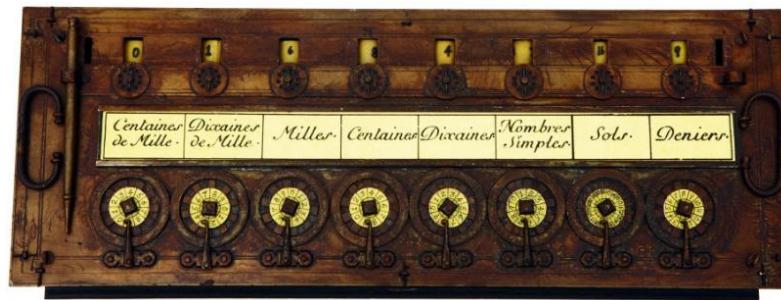
Abacus from www.computerhistory.org



Napier's Bones from www.computerhistory.org

■ First mechanical computers: + - (* /)

- Leonardo da Vinci (1452 - 1519)
 - around 1500 → rebuilt successfully in 1967
- Wilhelm Schickard (1592 - 1635)
 - around 1625 → no preserved originals, rebuilt
- **Blaise Pascal (1623 - 1662)**
 - around 1640 → arithmetic machine (Pascaline)
- Gottfried von Leibnitz (1646 - 1716)
 - enhancement of arithmetic machine



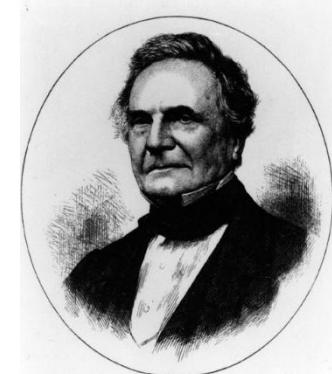
replica of a Pascaline from www.computerhistory.org

Computer History



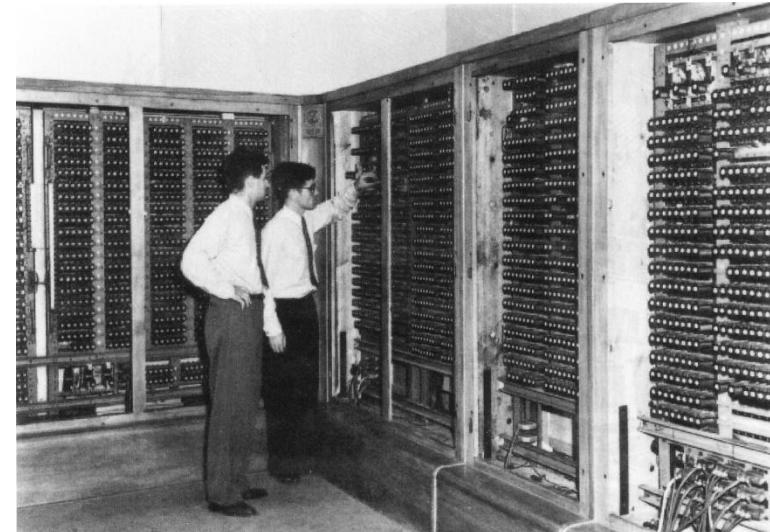
■ First mechanical computer in today's sense

- Charles Babbage
 - around 1822 "Difference Engine", not completed
 - replaced by "Analytical Machine"
- Ada Lovelace
 - Mathematician
 - wrote programs for the Analytical Machine
 - Daughter of Lord Byron



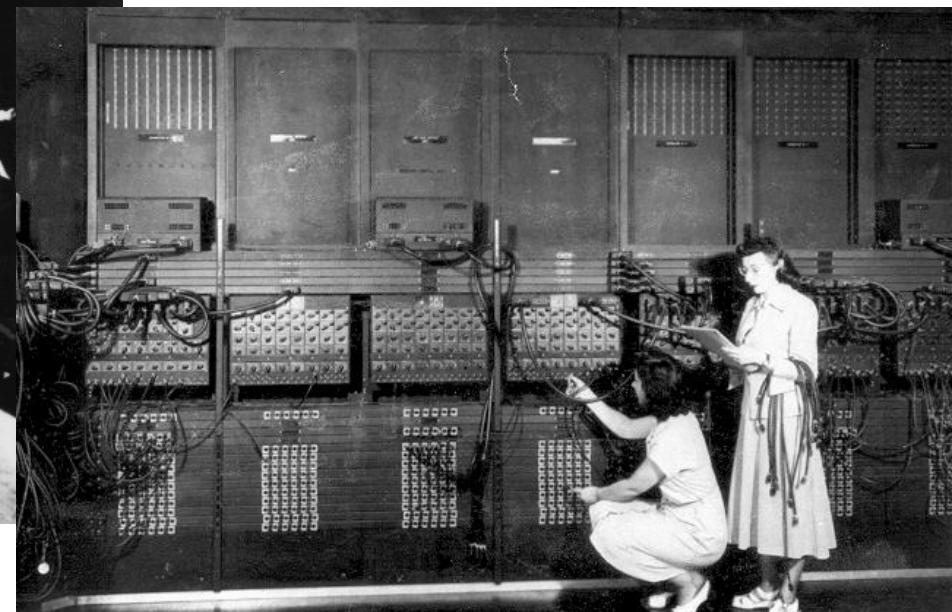
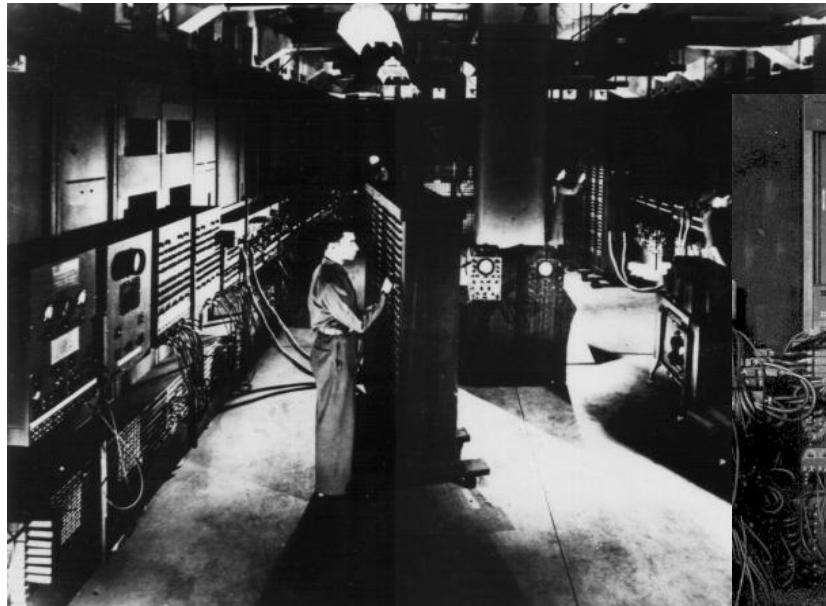
■ First electromechanical computers

- Howard H. Aiken
 - Harvard Mark 1, between 1939 and 1944
 - consisting of switches, relays
 - around 750'000 components:
 - 15m x 2.4m x 0.6m, 4500 kg
- Konrad Zuse, Germany
 - Z3, built in 1941 in Berlin
 - 1944 destroyed by bombing
 - work on Z4 started around 1943
 - used at the ETH from 1950 on



■ First electronic "general purpose" computer

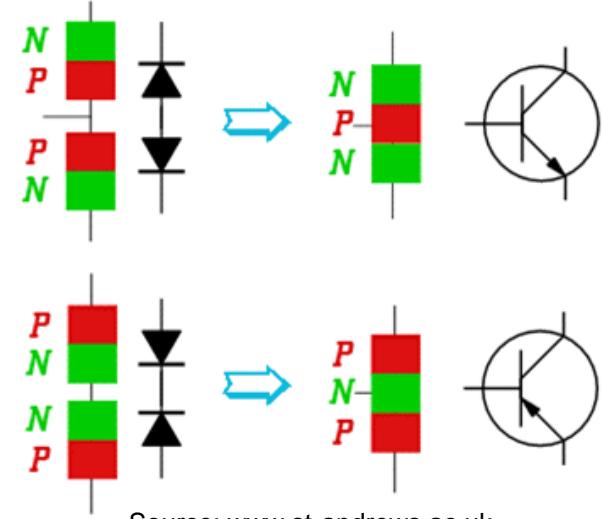
- J. Presper Eckert and John Mauchly, Univ. of Pennsylvania
 - ENIAC, 1944 → Electronic Numerical Integrator And Calculator
 - around 18'000 tubes, 30 tons, 140 kW, 5'000 additions / s, 1400 m²



Computer History

■ First transistors

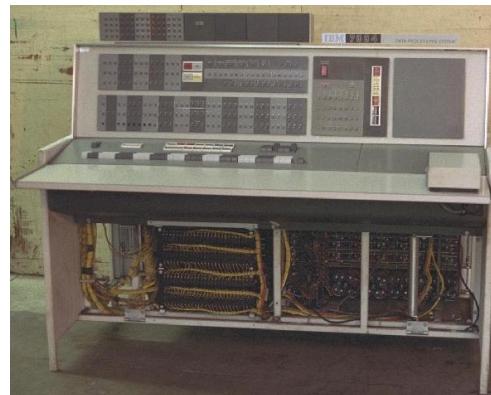
- 1926, patent by Julius Edgar Lilienfeld
- 1947, Germanium transistor
 - W. Shockley, W. Brattain, J. Bardeen
- 1950, Bipolar Transistor
 - William Shockley



Source: www.st-andrews.ac.uk

■ Early transistor-based computers

- around 1957
 - DEC PDP-1
 - IBM 7000
 - NCR & RCA



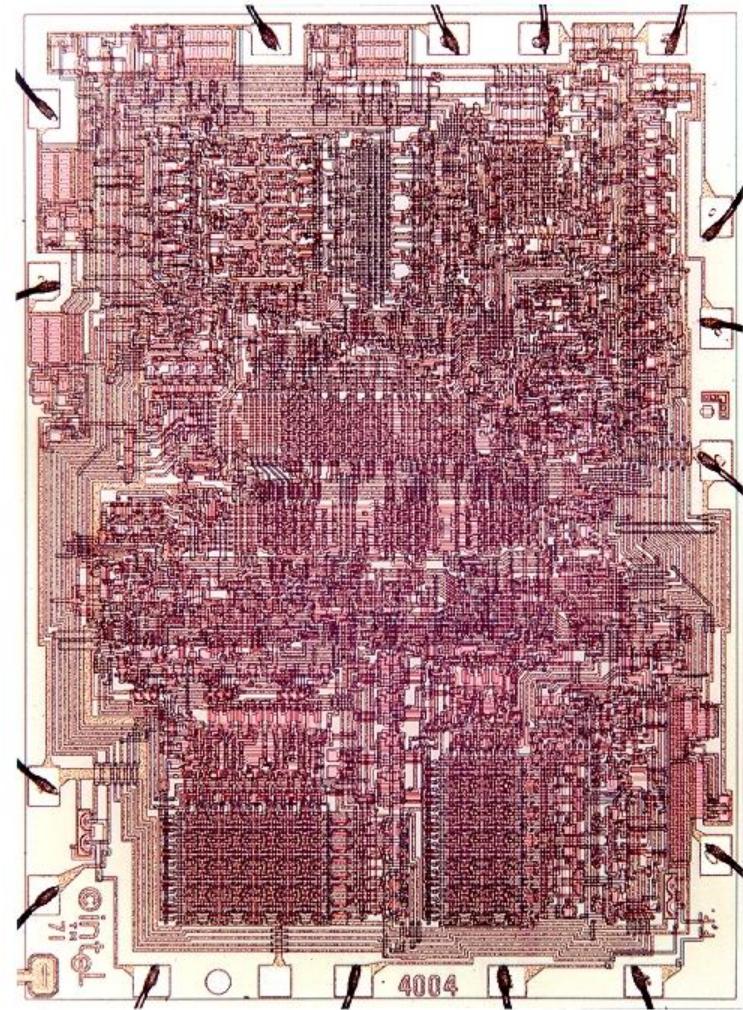
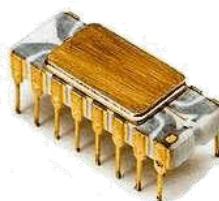
■ Early integrated circuits (IC)

- 1958, Jack Kilby at Texas Instruments (TI)
 - based on an idea from 1952
 - several components on the same substrate
- 1963, Fairchild, "the 907 device"
 - 2 logic gates
- 1967, Fairchild, "Micromosaic"
 - several 100 transistors
- 1970, Fairchild
 - first 256-bit static RAM



■ Early microprocessors

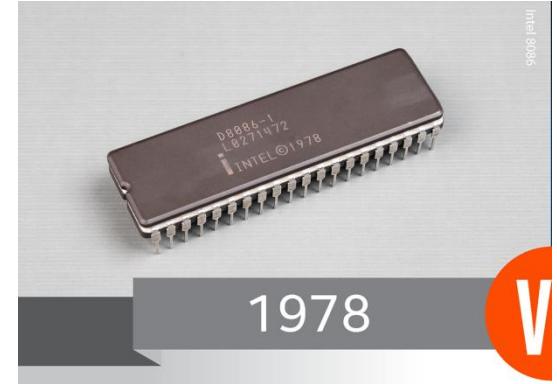
- 1971, Intel 4004
 - all CPU components on a single chip
 - 4 Bit, 2300 transistors
 - 12mm² (3x4 mm)
- 1972, Intel 8008
 - 8-bit version of the 4004



2018: 40 Years 8086

■ 1978: Intel 8086

- 29000 transistors
- ~33 mm²
- 5 MHz
- 0,33 MIPS



■ 2018 Intel Core i7 8086K multi core

- 6 Cores
- ~149 mm²
- 4 GHz
- 14 nm gate length
- 2000 – 5000 mio. transistors (estimated)
- > 100'000 MIPS



Source: Intel

<https://www.heise.de/newsticker/meldung/40-Jahre-8086-der-Prozessor-der-die-PC-Welt-veraenderte-4074260.html>

Where are we today?

■ AMD Ryzen 2 Architecture

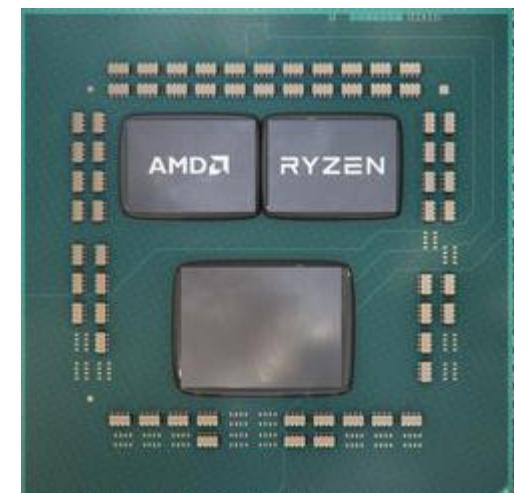
- Up to 64 cores
- 7 nm (cores) / 12 nm (interconnect) gate length
- Example: AMD Ryzen 9 3950X, 16 cores (Q3/2019)
 - ~9'890'000'000 transistors
 - ~273 mm²
 - Up to 4.7 GHz

■ Area

- $A_{i7} = \sim 23 \cdot A_{4004}$

■ Transistors

- $T_{i7} = \sim 4'300'000 \cdot T_{4004}$

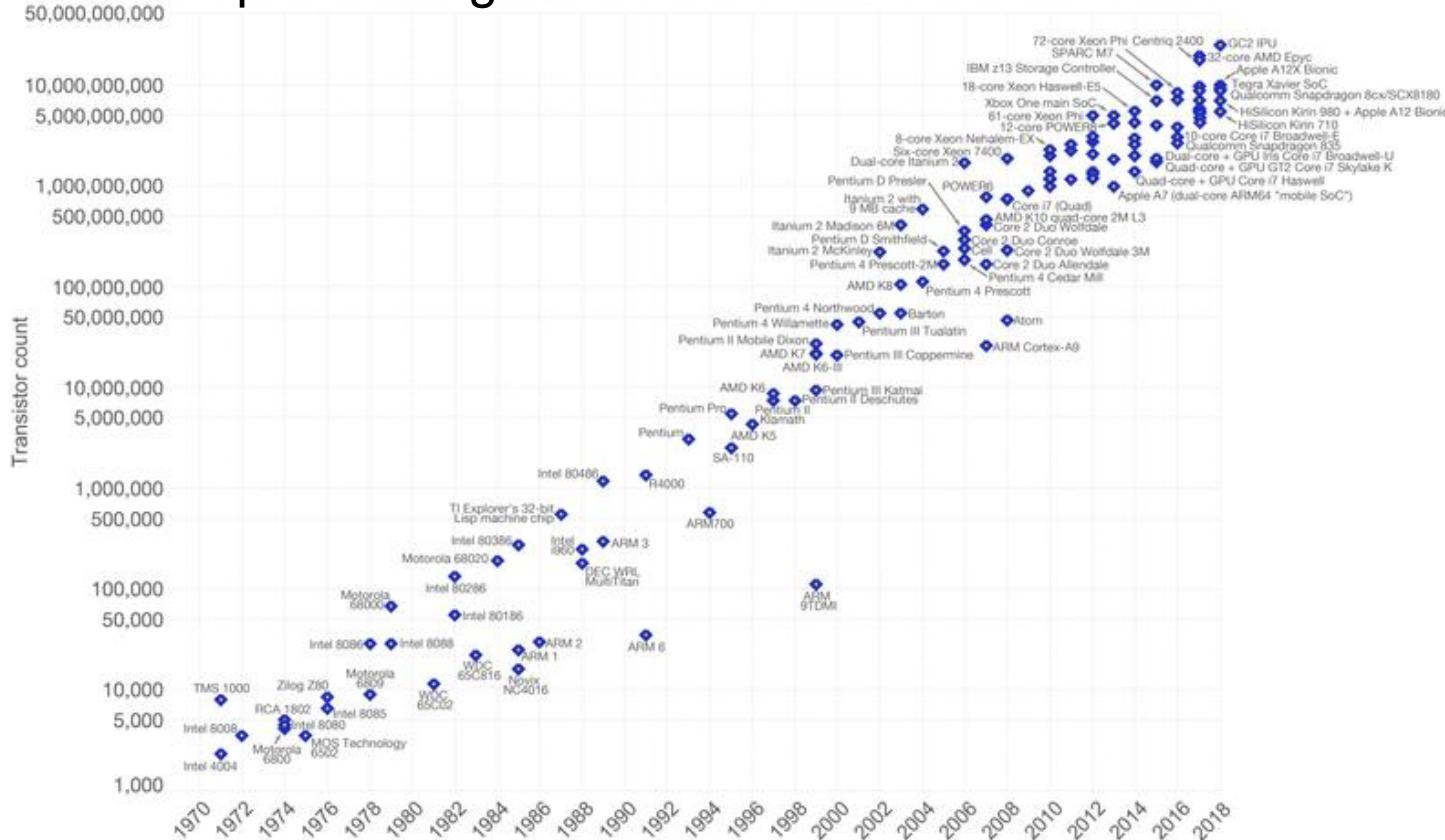


Source: AMD

Moore's Law

■ Gordon Moore, Intel

- 1965: "The number of transistors per IC doubles every year"
 - somewhat slower since 1975 → doubles every 18 months, i.e. exponential growth



Computer Engineering Today

Moore's Law lebt

Halbleiter-Trends und neue High-End-Chips

Der Forschungschef des Chip-Auftragfertigers TSMC skizziert auf dem Hot Chips Symposium die Zukunft der Halbleiterindustrie. Und ein Riesen-Chip für künstliche Intelligenz nutzt fast einen ganzen Wafer.



Geht es nach Philip Wong, Forschungschef des weltgrößten Chip-Auftragfertigers TSMC, dann gilt das Moore'sche Gesetz auch noch im Jahr 2050. Die Metrik, die Wong für seine Aussage nutzt, ist die Packdichte der Transistoren. In der Vergangenheit war die Strukturbreite die gängige Stellschraube für immer dichter gepackte Transistoren, und an ihr werde man in absehbarer Zeit auch weiterhin drehen. 5-nm-Chips seien bereits als Prototypen verfügbar und danach werde es feinere Fertigungsprozesse geben. Wenn die Strukturgrößen in den Bereich weniger Atomlagen schrumpfen, verändert sich allerdings das Verhalten bisheriger Transistortypen zum Schlechteren. Bislang noch nicht verwendete Materialien wie MoS₂, WSe₂ oder WS₂ zeigten hingegen auch bei Dicken von einem Nanometer und darunter noch gute Resultate. Auch den schon länger bekannten Kohlenstoffnanoröhren wird weiterhin viel geforscht. Zusätzlich kämen weitere Verbesserungen hinzu. Als aktuelles Beispiel nannte Wong FinFETs, also dreidimensionale (Gate-)Strukturen auf dem Wafer. Den aktuellen Trend, statt großer monolithischer Dies mehrere kleinere Chiplets zu einem großen Ganzen zusammenzuschalten - wie AMD es beispielsweise bei den jüngst erschienenen Zen-2-CPU's tut -, sieht Wong als einen weiteren Kniff. Die Idee sei aber alles

Flexiprozessor

Das Potenzial von RISC-V-Prozessoren

Die offene CPU-Architektur RISC-V steckt zwar erst in wenigen Chips, begeistert aber viele Entwickler. Denn RISC-V eignet sich für Einsatzbereiche vom Mikrocontroller bis zum Supercomputer und verspricht höhere Sicherheit.



Bild: Albert Hulm
Das Interesse an RISC-V vereint so unterschiedliche Firmen und Institutionen wie Google, russische Waffenhersteller, die NSA, den chinesischen Geheimdienst, Blockchain-Chiphersteller, unterfinanzierte Unis und europäische Supercomputer-Forscher. Manche wollen Lizenzgebühren sparen, andere neuartige Prozessortechnik entwickeln, wieder andere suchen sichere Chips ohne Hintertüren. RISC-V verspricht, solche Anforderungen zu erfüllen, und wird von CPU-Experten heiß diskutiert - obwohl es bisher erst sehr wenige RISC-V-Chips gibt. Der folgende Überblick zeigt, was die

Source:

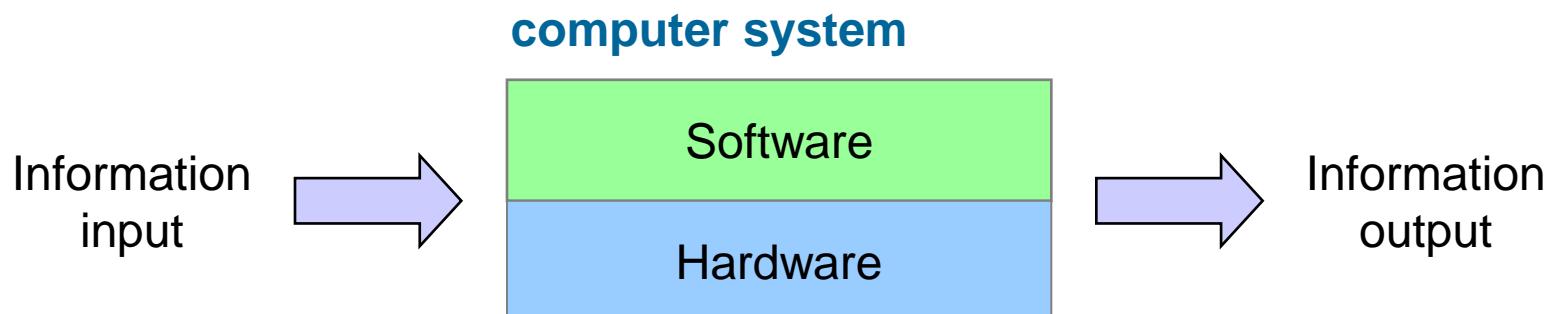
c't Heft 19/2019

S. 46-47 / News - Prozessor-Entwicklung

S. 134-139 / Hintergrund - Prozessortechnik

Properties of a Computer System

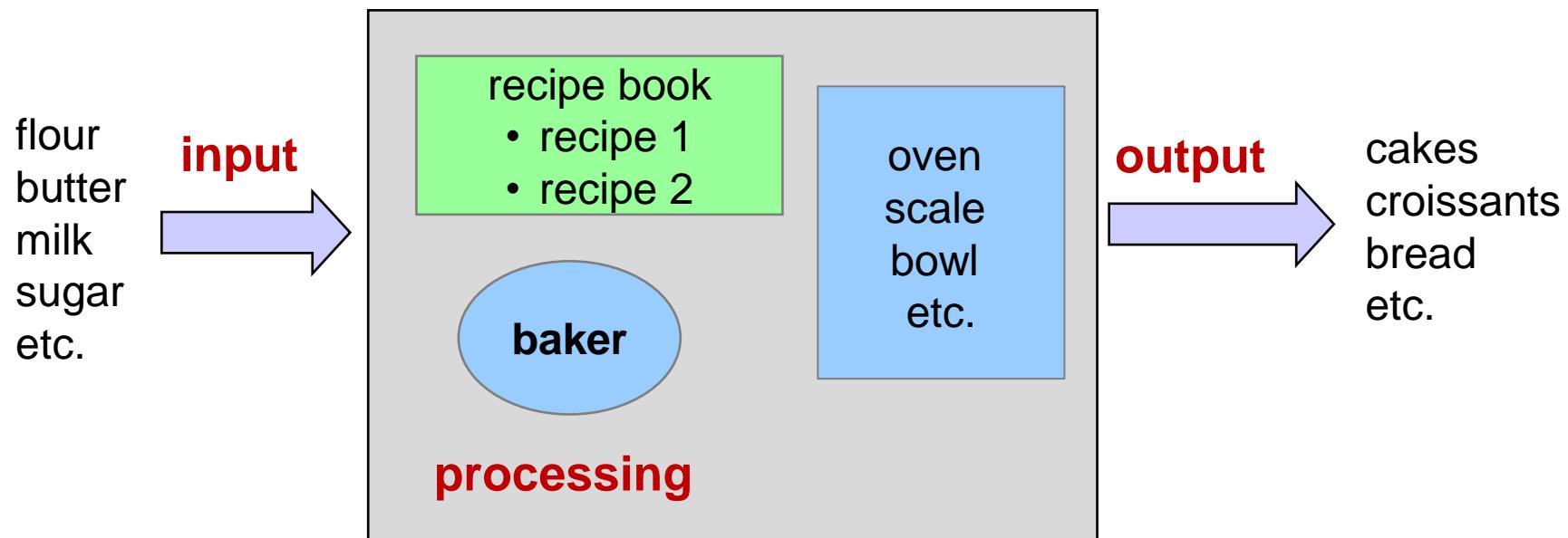
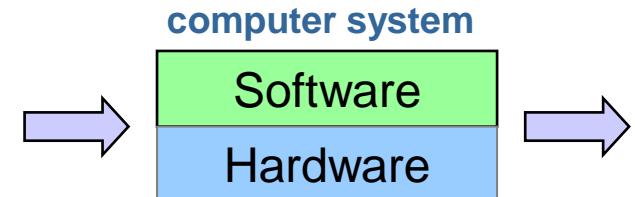
- **A computer system is a device that**
 - processes input
 - takes decisions based on the outcome
 - and outputs the processed information
- **Hardware and software work together → application**
 - often a common hardware is used for many different applications
 - application is defined by the software
 - e.g. controls for washing machines, vending machines,



Properties of a Computer System

■ Analogy → bakery

- baker ↔ processor
- recipe book ↔ software
- tools ↔ HW-resources



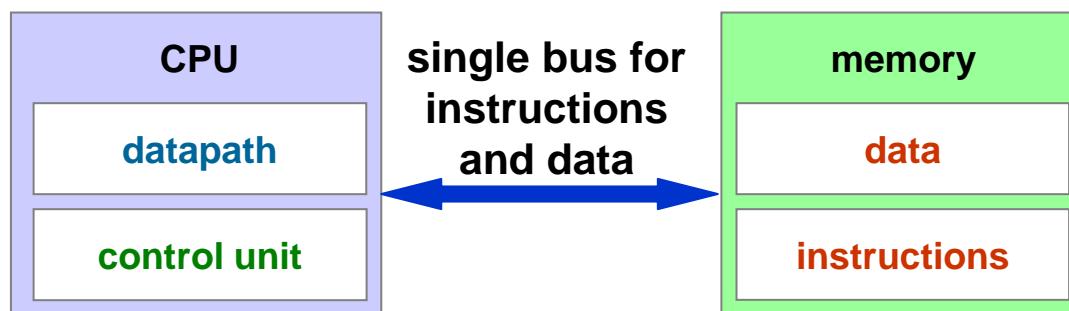
von Neumann Architecture

- Many of today's computers are based on ideas of John von Neumann in the year 1945
- Properties

- **instructions** and **data** are stored in the same memory
- **datapath** executes arithmetic and logic operations and holds intermediate results
- **control unit** reads and interprets instructions and controls their execution



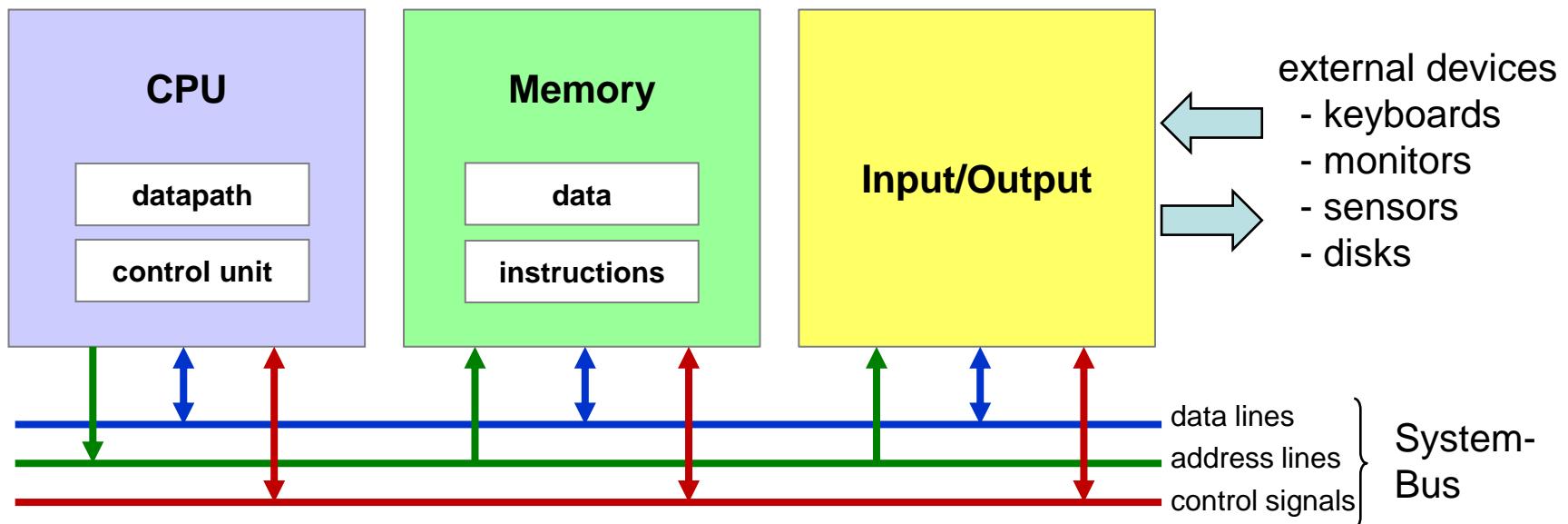
Von Neumann in the 1940s
en.wikipedia.org



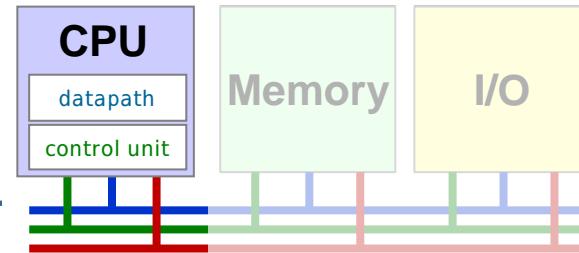
Hardware Components

- **CPU**
- **Memory**
- **Input / Output**
- **System-Bus**

Central Processing Unit or processor
stores instructions and data
interface to external devices
electrical connection of blocks

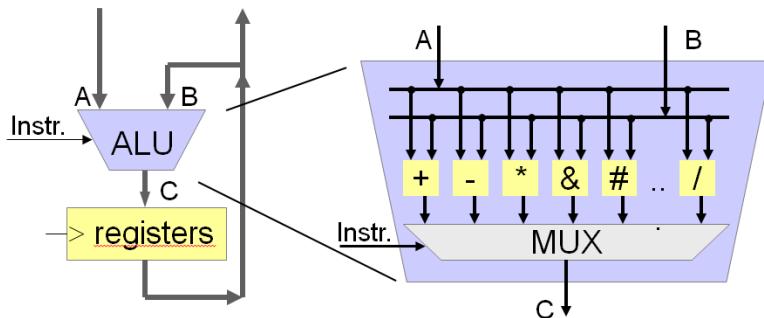


HW Components: CPU



Datapath

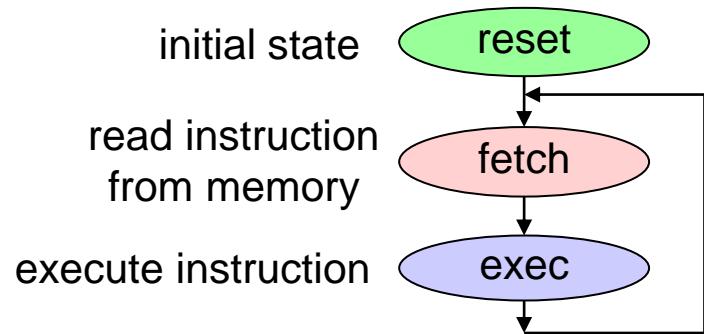
- ALU: Arithmetic and Logic Unit
 - performs arithmetic/logic operations



- registers
 - fast but limited storage inside CPU
 - hold intermediate results
- 4 / 8 / 16 / 32 / 64 bits wide

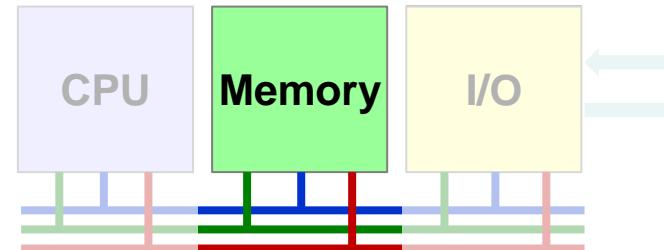
Control Unit

- Finite State Machine (FSM)
 - reads and executes instructions



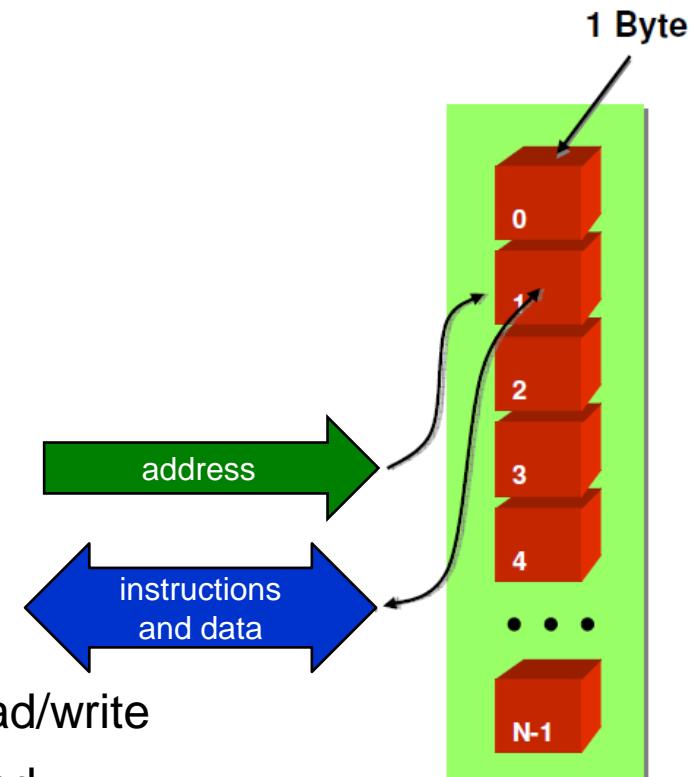
- types of operations
 - data transfer: registers \leftrightarrow memory
 - arithmetic and logic operations
 - jumps

HW Components

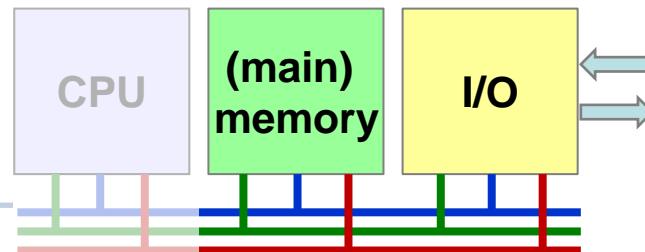


■ Memory

- a set of storage cells
 - 8 bit → 1 byte
- smallest addressable unit
 - one byte
 - one address per byte
- 2^N addresses
 - from 0 to 2^N-1
 - can be read and sometimes written
 - RAM Random Access Memory read/write
 - ROM Read Only Memory read

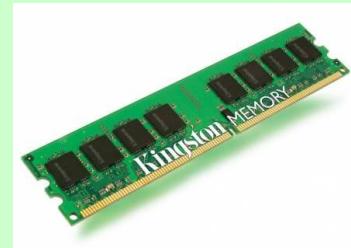


HW Components



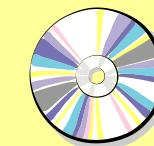
Main memory - Arbeitsspeicher

- central memory
- connected through System-Bus
- access to individual bytes
- **volatile (flüchtig)**
 - SRAM – Static RAM
 - DRAM – Dynamic RAM
- **non-volatile (nicht-flüchtig)**
 - ROM factory programmed
 - flash in system programmable



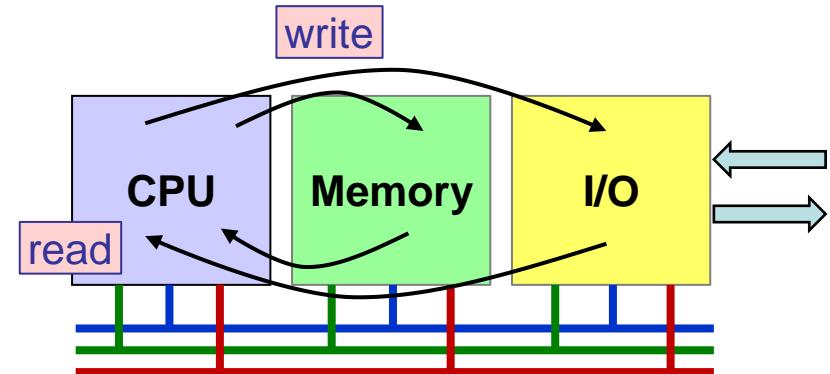
Secondary storage

- long term or peripheral storage
- connected through I/O-Ports
- access to blocks of data
- **non-volatile**
- slower but lower cost
 - magnetic hard disk, tape, floppy
 - semiconductor solid state disk
 - optical CD, DVD
 - mechanical punched tape/card



■ System-Bus

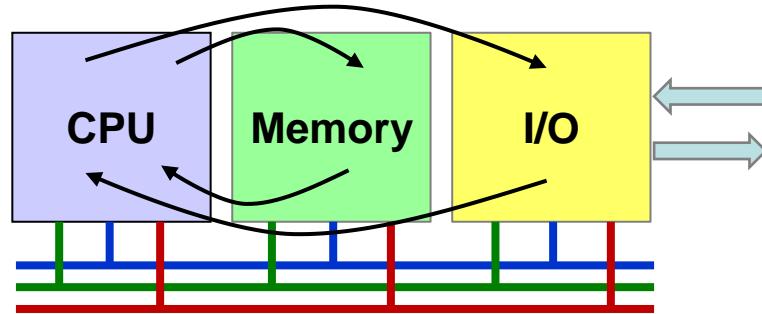
- CPU writes or reads data from/to memory or I/O



- **address lines**

- CPU drives the desired address onto the address lines
 - ▶ to which address does the CPU write?
 - ▶ from which address does the CPU read?
 - ▶ analogy → address on an envelope of a letter
- number of addresses = 2^n → n = number of address lines
 - ▶ n = 16 → $2^{16} = 65'536$ addresses → 64 KBytes
 - ▶ n = 20 → $2^{20} = 1'048'576$ addresses → 1 MBytes

■ ... System-Bus

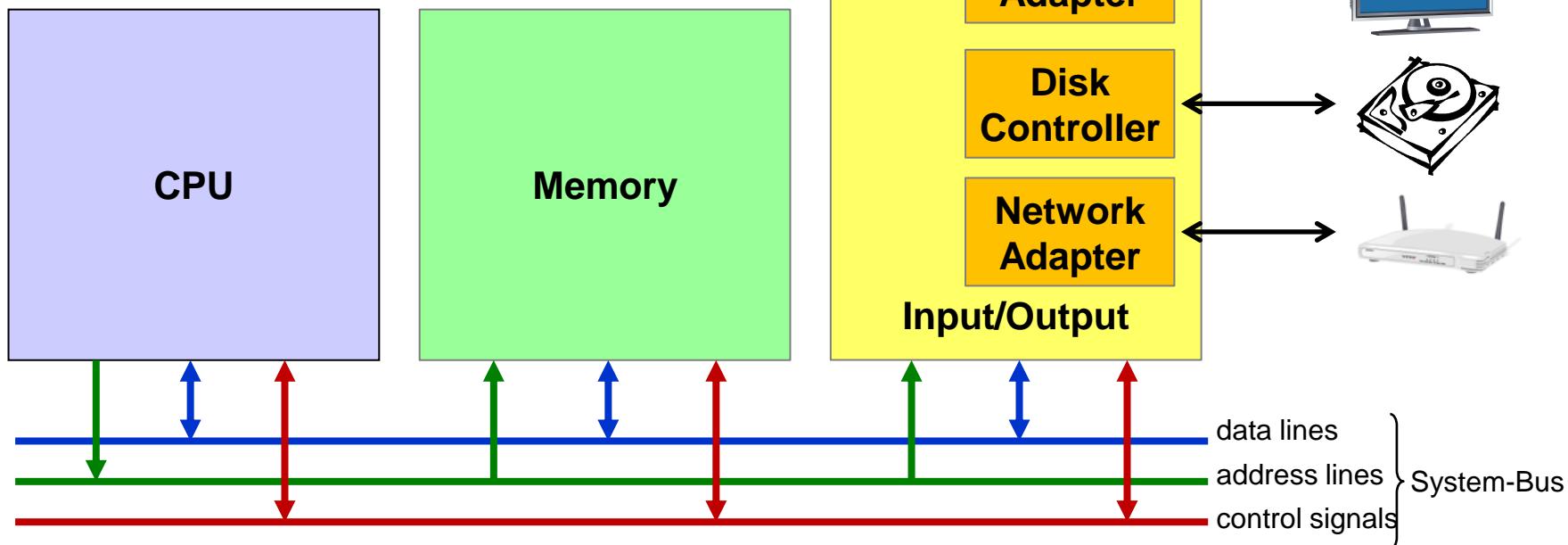


- **control signals**
 - CPU tells whether the access is read or write
 - CPU tells when address and data lines are valid → bus timing
- **data lines**
 - transfer of data
 - ▶ analogy the letter that's inside the envelope
 - ▶ write CPU provides data → memory receives data
 - ▶ read CPU receives data ← memory provides data
 - 4/8/16/32/64 data lines

HW Components

■ Example PC

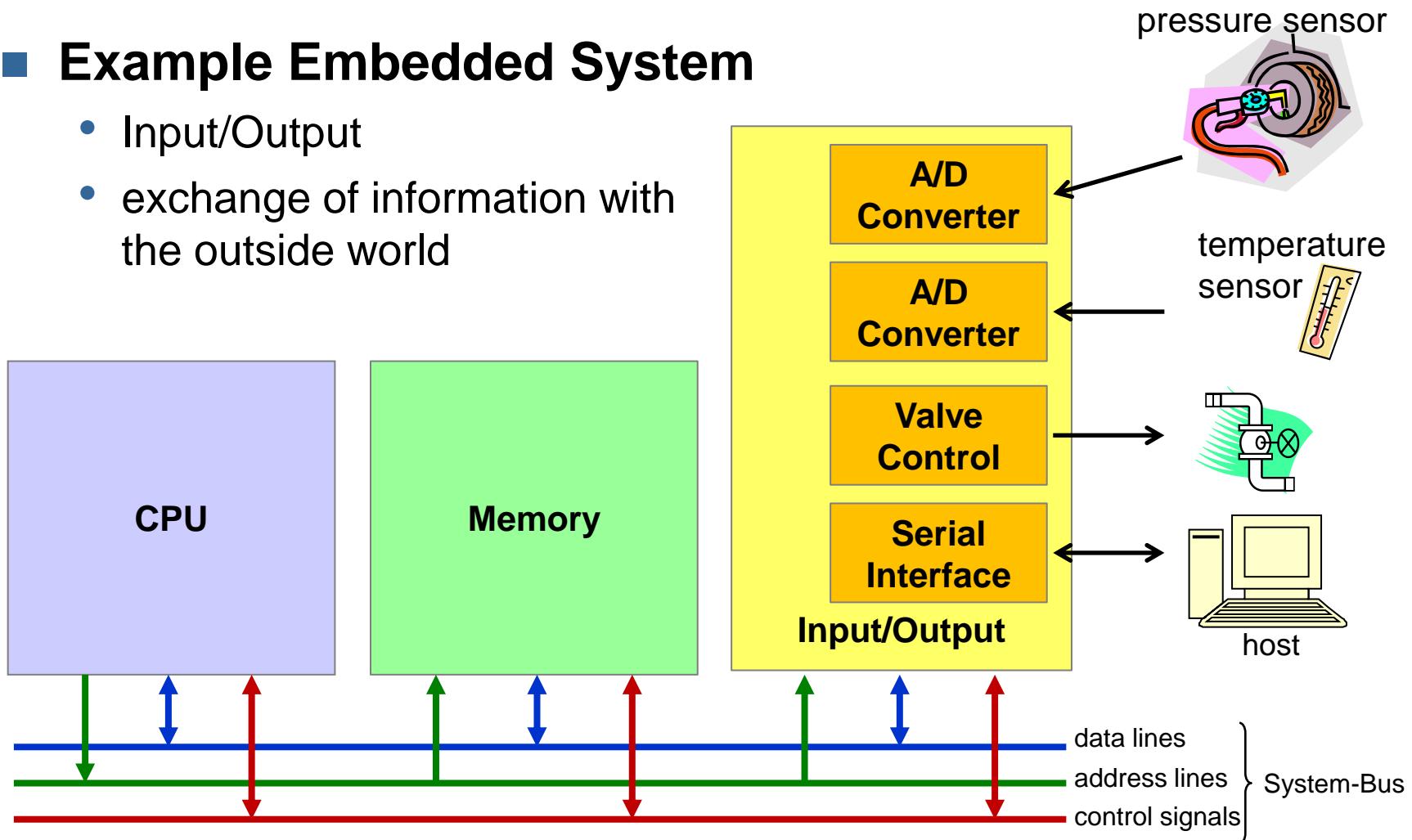
- Input/Output
- exchange of information with outside world



HW Components

■ Example Embedded System

- Input/Output
- exchange of information with the outside world



So far

- CPU reads instructions from memory and executes them

But

- How to process a program in a high level language like C so that a CPU can interpret the instructions?
- What is needed for a program in C to allow execution on a CPU?
- What does the path from the C source code to the executable object file look like?

Software Aspects

■ Programmer writes `main.c` in text editor

```
main.c

#include "utils_ctboard.h"

#define LED_ADDR      0x60000100
#define LED_VALUE     0x12

int main(void)
{
    while (1) {
        write_byte(LED_ADDR, LED_VALUE);
    }
}
```

calls function
`write_byte()`
from module
`utils_ctboard`

Software Aspects

- **main.c is stored in ASCII / Unicode format on disk**

```
#include "utils_ctboard.h"

#define LED_ADDR      0x60000100
#define LED_VALUE     0x12

int main(void)
{
    while (1) {
        write_byte(LED_ADDR, LED_VALUE);
    }
}
```

→ 0x23
i → 0x69
n → 0x6E
c → 0x63
....

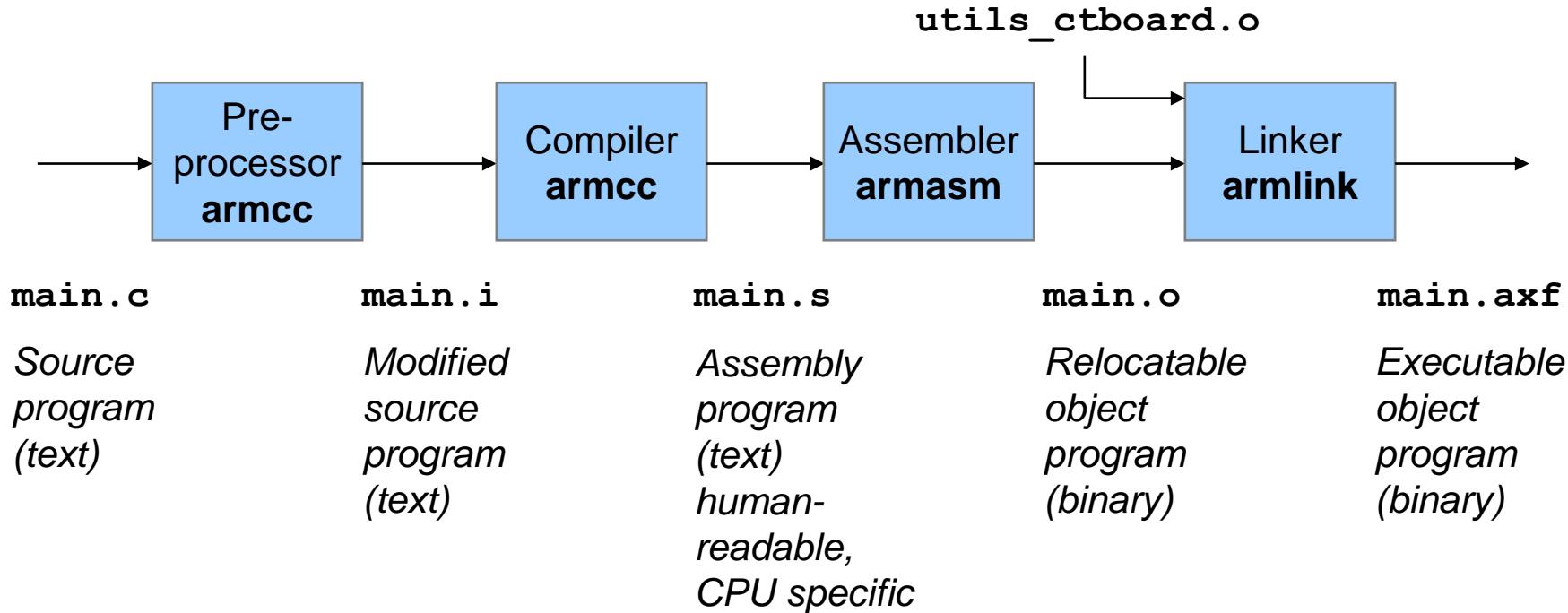
23696E636C75646520227574696C735F
6374626F6172642E68220D0A0D0A2364
6566696E65204C45445F414444522020
.....

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	Ø	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	:	91	5B	[123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	Ø	127	7F	Ø

Software Aspects

■ From C to executable

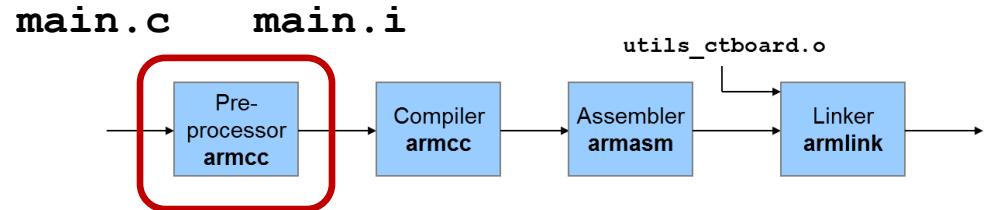
- Translation of `main.c` into machine language



Software Aspects

■ Preprocessor

- Text processing
- Pasting of #include files
- Replacing macros (#define)



pasting included content
of included files

```
#include "utils_ctboard.h"

#define LED_ADDR      0x60000100
#define LED_VALUE     0x12

int main(void)
{
    while (1) {
        write_byte(LED_ADDR, LED_VALUE);
    }
}
```

`main.c`

```
void write_byte(uint32_t address,
                uint8_t data);

int main(void)
{
    while (1) {
        write_byte(0x60000100, 0x12);
    }
}
```

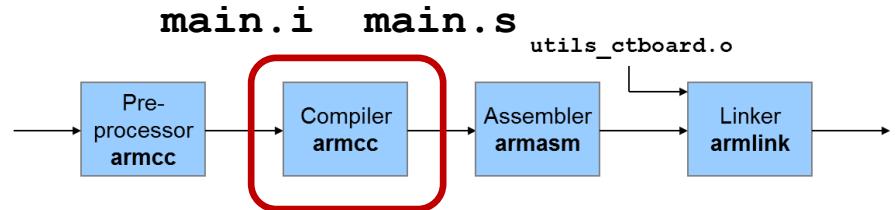
`main.i`

replacing macros

Software Aspects

Compiler

- Translate CPU-independent C-code into CPU-specific assembly code



main.c

```
void write_byte(uint32_t address,
                uint8_t data);

int main(void)
{
    while (1) {
        write_byte(0x60000100, 0x12);
    }
}
```

human readable

main.s

```
AREA .text, CODE, READONLY

main      PROC
          B           endloop

mloop     MOVS      r1,#0x12
          LDR       r0,L112
          BL        write_byte

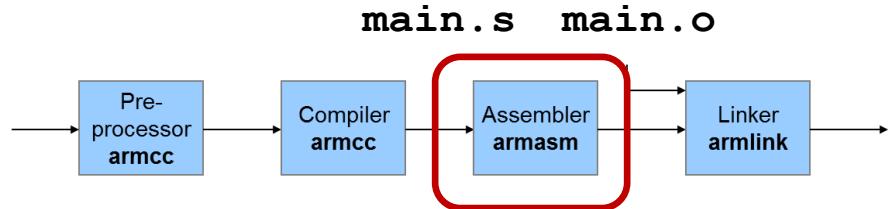
endloop   B           mloop
          ENDP

L112     DCD      0x60000100
```

Software Aspects

■ Assembler

- Translate to machine instructions
 - Result: Relocatable object file
 - Binary file → not readable with text editor, use Hex Dump



```
main.s
AREA .text, CODE, READONLY

main      PROC
          B          endloop

mloop     MOVS      r1,#0x12
          LDR       r0,L112
          BL        write_byte

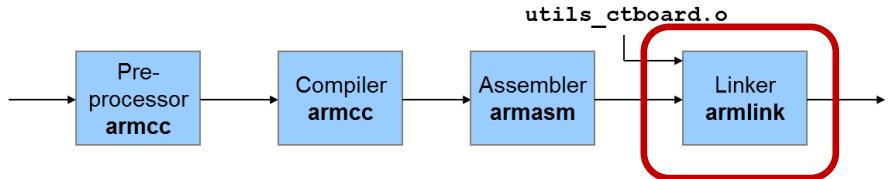
endloop   B          mloop
          ENDP

L112     DCD      0x60000100
```

Software Aspects

■ Linker

- armcc



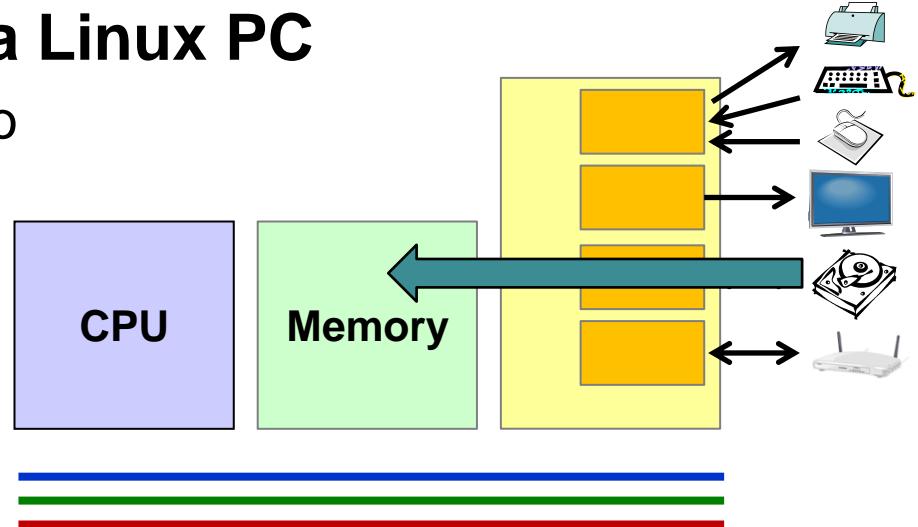
main.axf

remember: `main()` calls function
`write_byte()` from module
`utils_ctboard`

Software Aspects

■ Program execution on a Linux PC

- load executable `hello` into memory and execute it

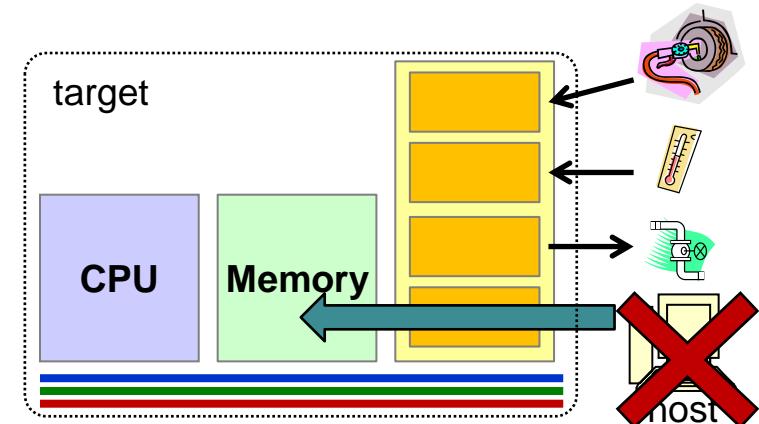
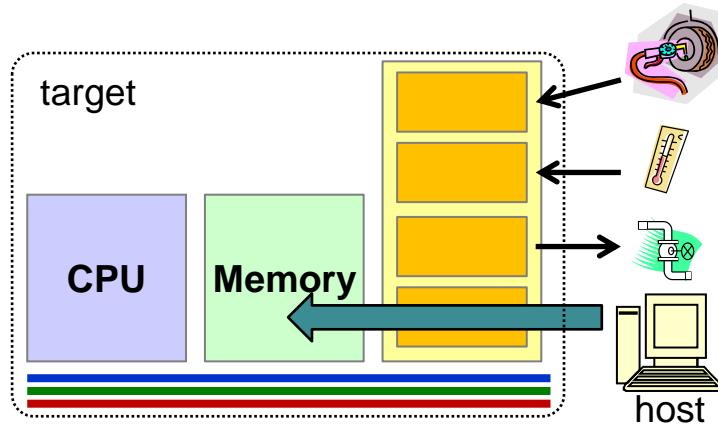


- typing `./hello` in a shell
 - transfers the executable from disk to memory (RAM)
- operating system
 - creates a new process
 - jumps to start of `main()` function and begins execution

Software Aspects

■ Program execution on Small Scale Embedded System

- Host vs. Target



Software development on host

- Compiler/Assembler/Linker on host
- Loader on target loads executable (e.g. *.axf) from host to RAM
- Loader copies executable from RAM into non-volatile memory (FLASH)
→ **Firmware Update**

System operation without host

- Loader jumps to `main()` and starts execution
- Instruction fetch often takes place directly from FLASH

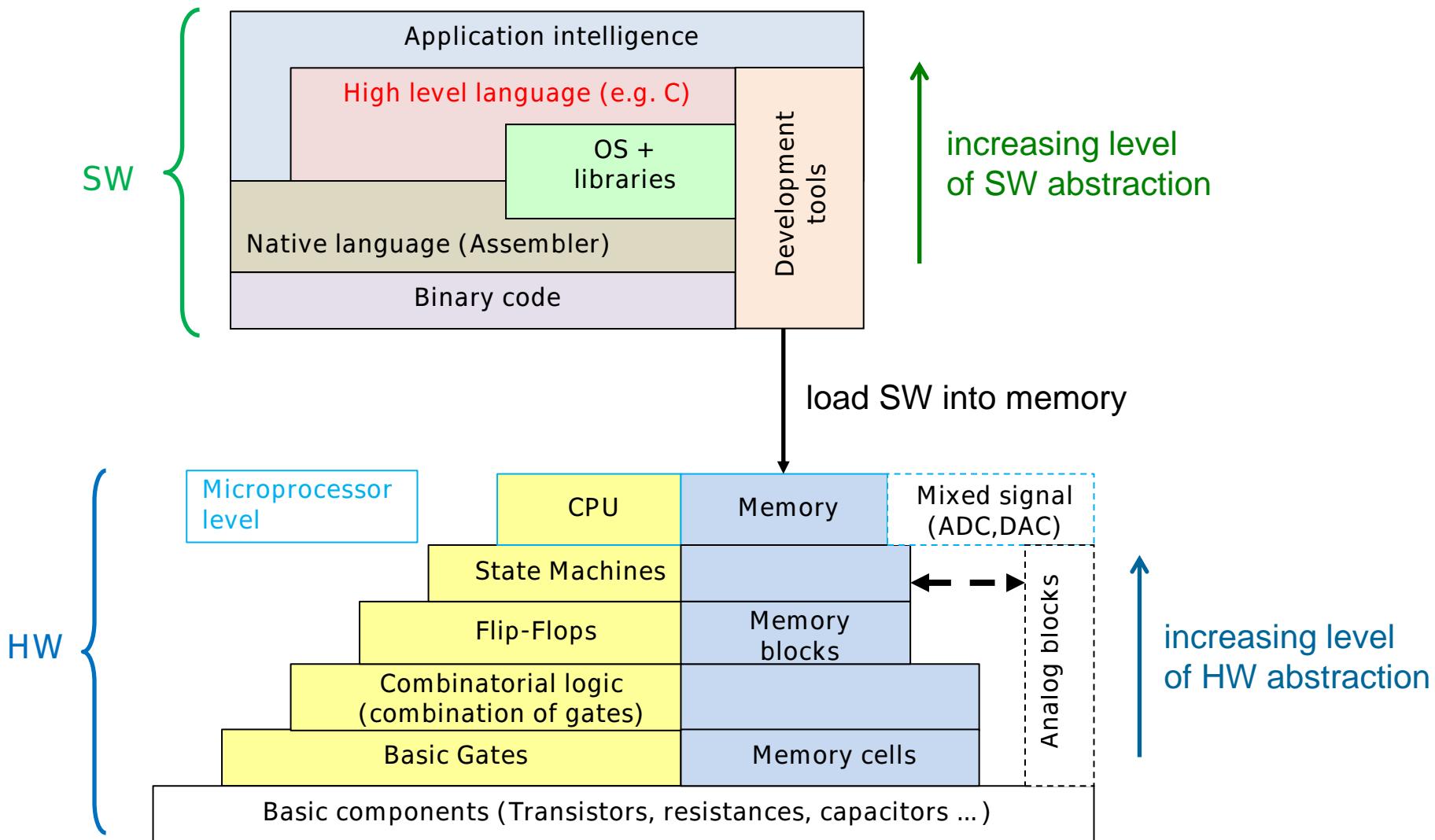
■ Why learn assembly language?

- few engineers write assembly code
 - use of High Level Languages (HLL) and compilers more efficient

■ But

- assembly language yields understanding on machine level
 - understanding helps to avoid programming errors in HLL
- increase performance
 - understand compiler optimizations
 - find causes for inefficient code
- implement system software
 - boot Loader, operating systems, interrupt service routines
- localize and avoid security flaws
 - e.g. buffer overflow

Interaction of HW and SW



Conclusion

- **Computer system → hardware and software**
- **Hardware**
 - CPU Central Processing Unit or microprocessor (μ P)
 - memory stores instructions and data
 - I/O input and output devices
 - system bus electrical connection of blocks
- **Software**
 - source code in high level language (C)
 - assembly code → machine-oriented, human readable
 - object code → machine instructions in binary without libraries
 - executable → executable object file including libraries
- **Target vs. Host**

Cortex-M Architecture

Computer Engineering 1

Motivation

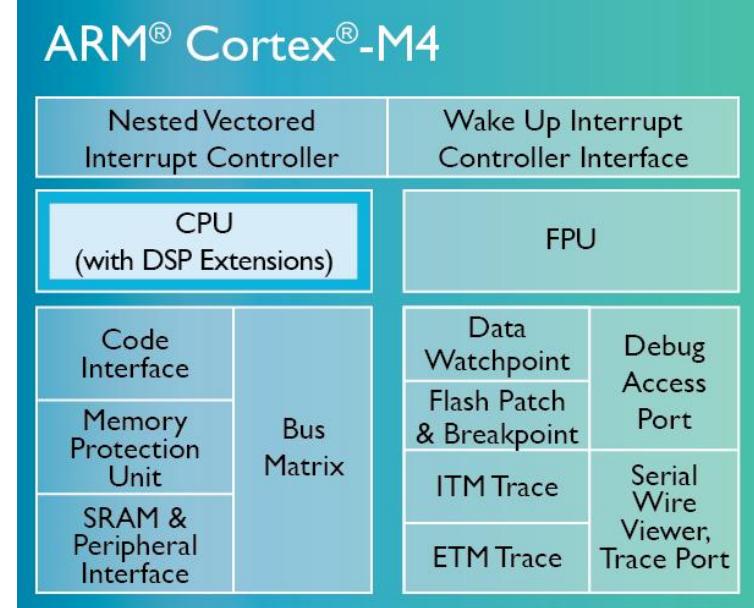
■ ARM (Cortex-M)

1985



2014

1'400 x
transistors
36 x
clock speed



ARM1
25k Transistors
5 MHz

ARMv7 - Cortex-M4
35 Mio Transistors
180 MHz
4 mm²

Agenda

- **Hardware Platform**
 - STM32F4 and evaluation board, ARM Processor Portfolio
- **CPU Model**
 - Register, ALU, Flags, Control Unit
- **Instruction Set**
 - Assembly, Instruction Types, Cortex-M0
- **Program Execution**
 - Fetch and Execute
- **Memory Map**
 - ARM, ST, CT-Board
- **Integer Types**
 - Sizes, Little Endian vs. Big Endian, Alignment
- **Object File Sections**
 - Code, Data, Stack

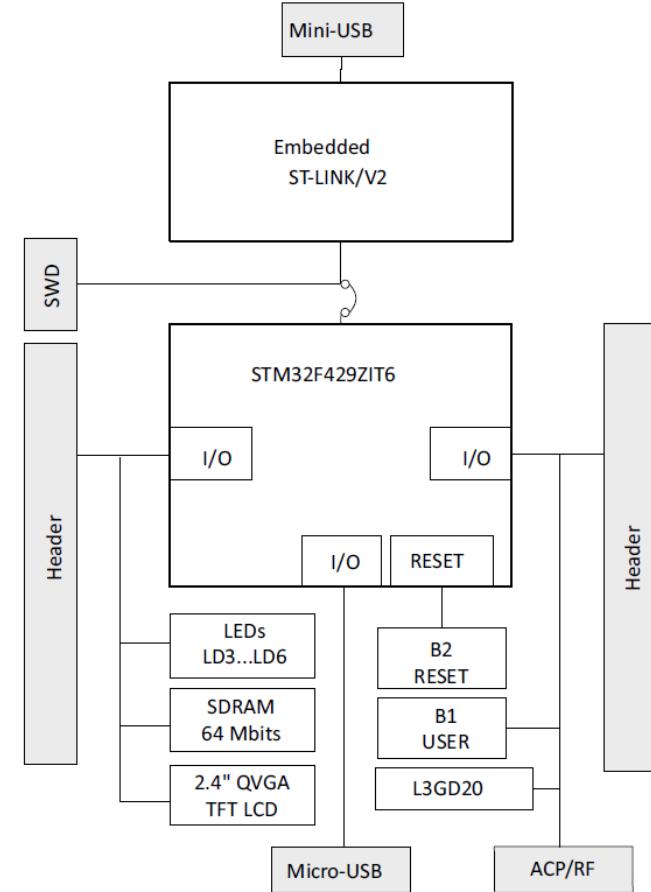
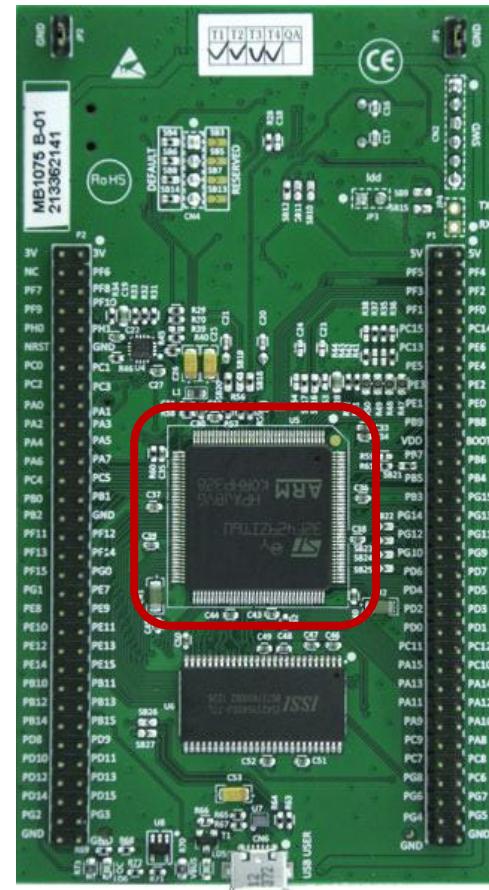
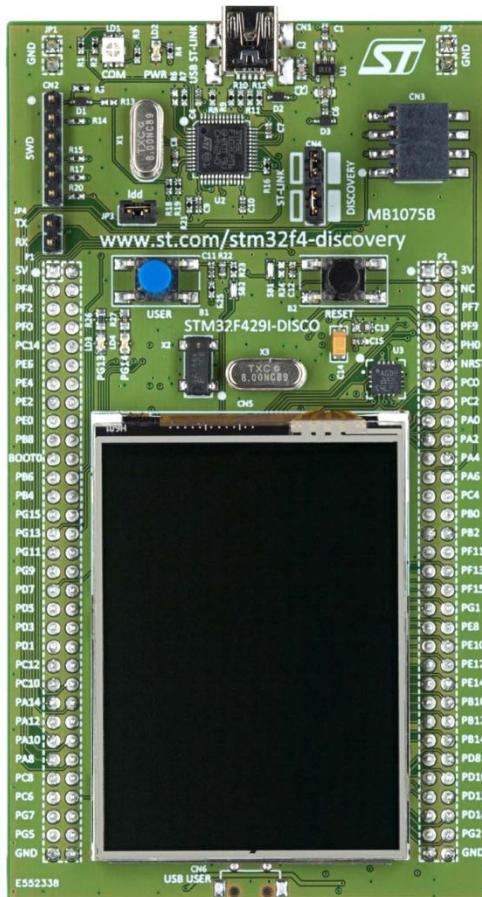
Learning Objectives

At the end of this lesson you will be able

- to describe what an 'Instruction Set Architecture' is
- to outline the Cortex-M architecture and enumerate the main components and their functions
- to enumerate the instruction type categories
- to understand the structure of the Cortex-M instruction set
- to explain how a processor executes a program
- to recall the registers of the Cortex-M, their layout and their functions
- to explain and draw a memory map
- to calculate the size in bytes for a memory block given by its start and end address
- to determine the end address of a memory block given by the start address and the number of bytes
- to understand that sizes of integer types in C depend on the architecture and that portability can be enhanced by using the C99 types in stdint.h
- to explain the difference between 'little endian' and 'big endian' and to show how multi-byte integer values are mapped to individual bytes
- to list the three typical memory sections of an object file and to explain their content

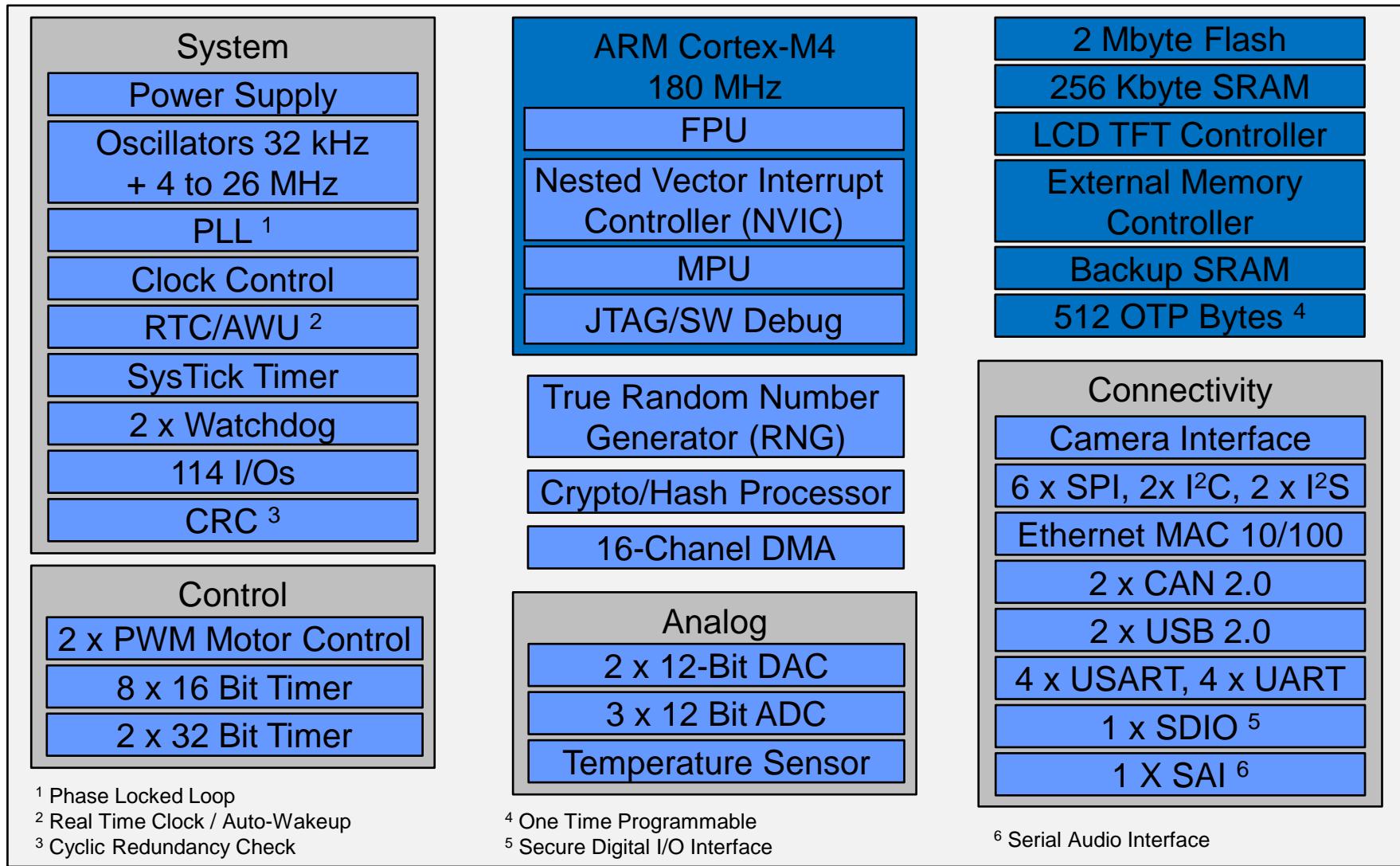
Hardware Platform: STM32F4-Discovery

■ Evaluation board STM32F4-DISCO

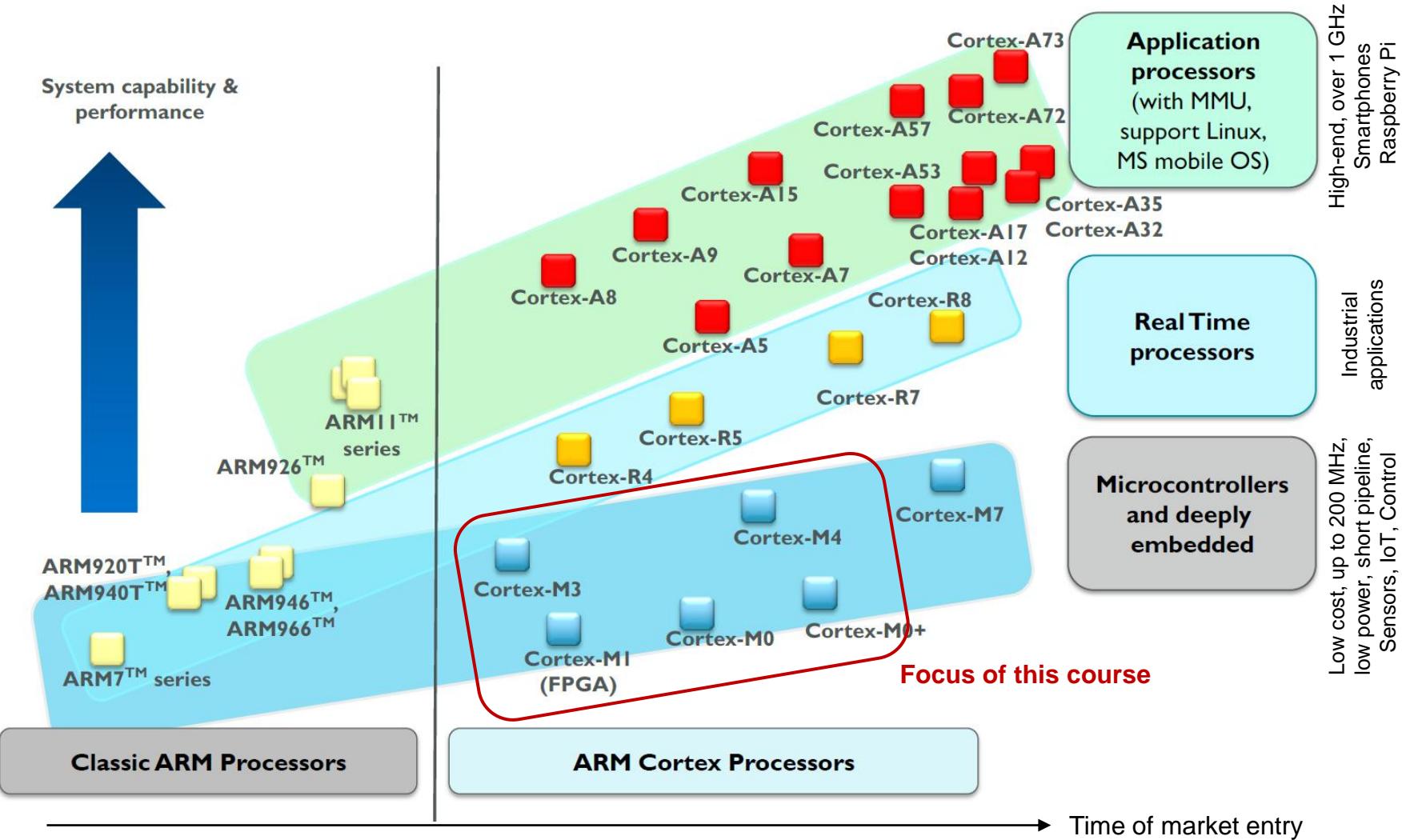


source: STMicroelectronics

Hardware Platform: STM32F429I



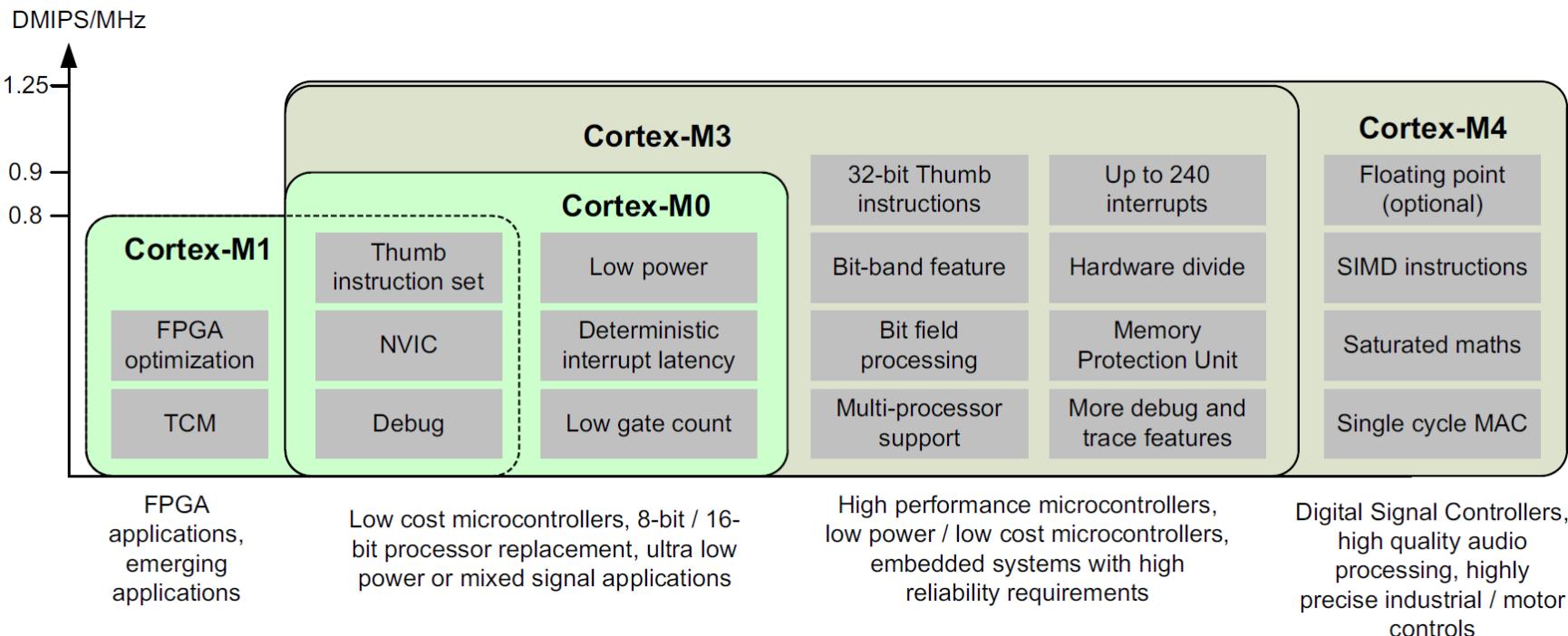
Hardware Platform: ARM Processor Portfolio



Hardware Platform

■ Cortex-M Processor Family

- Hardware used in this course is a Cortex-M4
- But most of the time we only use the simpler Cortex-M0 subset



■ Instruction Set Architecture (ISA) What the programmer sees of a computer

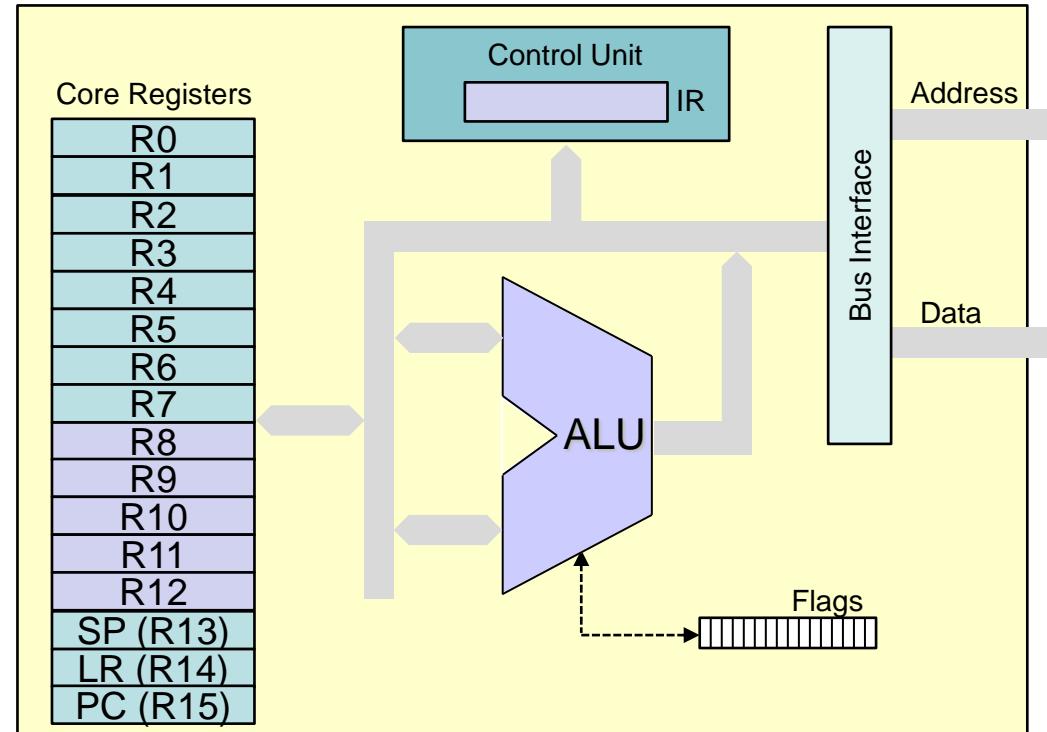
→ CT1

- Instruction Set
 - Available instructions?
- Processing width
 - 8-bit/16-bit/32-bit?
- Register set
 - How many registers? Which size?
- Addressing modes
 - How can memory and IO be accessed?
- ARM Cortex-M
 - ARMv6-M → Cortex-M0
 - ARMv7-M → Cortex-M3/M4 (Superset of ARMv6-M)

CPU Model

■ CPU Components

- Core Registers
- 32-bit ALU
- Flags (APSR)
- Control Unit with IR (Instruction Register)
- Bus Interface

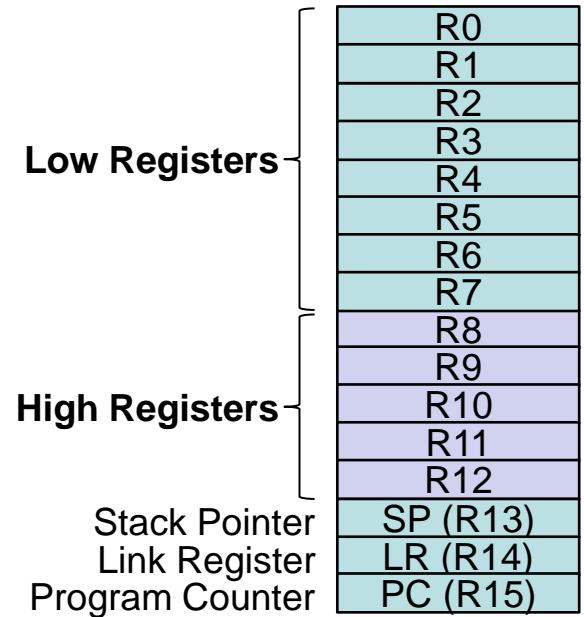


simple CPU model based on the Programmers'
Model of the ARM Cortex-Mx CPUs¹⁾

¹⁾ without pipelining

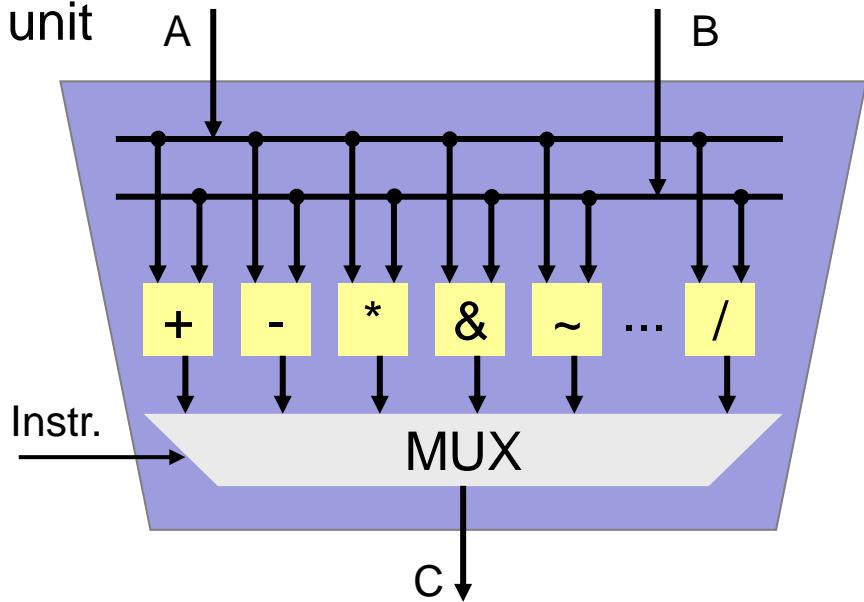
■ 16 Core Registers

- Each 32-bit wide
- 13 General-Purpose Registers
 - Low Registers R0 – R7
 - High Registers R8 – R12
 - Used for temporary storage of data and addresses
- Program Counter (R15)
 - Address of next instruction
- Stack Pointer (R13)
 - Last-In First-Out for temporary data storage
- Link Register (R14)
 - Return from procedures



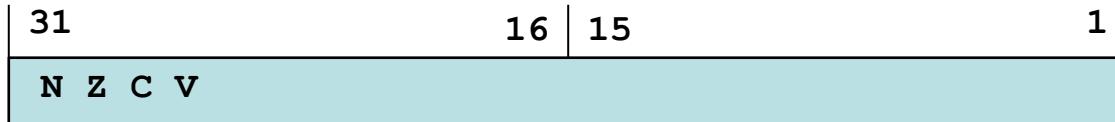
■ ALU – Arithmetic Logic Unit

- 32-bit wide data processing unit
 - inputs A and B
 - result C
- integer arithmetic
 - addition / subtraction
 - multiplication / division
 - sign extension
- logic operations
 - AND, NOT, OR, XOR
- shift/rotate
 - left / right



■ APSR¹⁾ or Flag-Register

- Bits set by CPU based on results in ALU



N	Negative
Z	Zero
C	Carry
V	Overflow

As seen in the IDE

Register	Value
Core	
R0	0x20000078
R1	0x20000278
R2	0x20000278
R3	0x20000278
R4	0x08000860
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x08000860
R11	0x08000860
R12	0x00000000
R13 (SP)	0x20000678
R14 (LR)	0x08000243
R15 (PC)	0x08000270
xPSR	0x41000000
N	0
Z	1
C	0
V	0

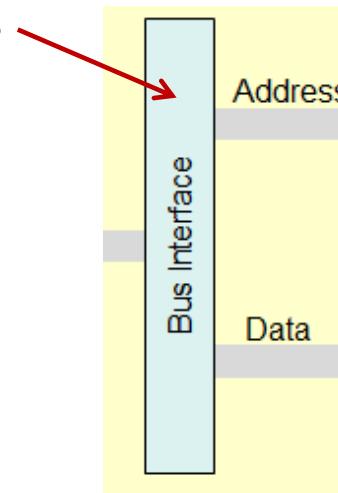
¹⁾ APSR: Application Processor Status Register

■ Control Unit

- Instruction Register (IR)
 - Machine code (opcode) of instruction that is currently being executed
- Controls execution flow based on instruction in IR
- Generates control signals for all other CPU components

■ Bus Interface

- Interface between internal CPU bus and external system-bus
 - contains registers to store addresses



■ Processors interpret binary coded instructions

- But binary is hard for programming
- Therefore instructions in human readable text form
 - → assembly
- Assembler (tool) does the translation
 - assembly → binary

ADDS R0 ,R0 ,R1



0001'1000'0100'0000
=
0x1840

Instruction Set

■ Assembly Program

- Label (optional)
- Operands
- Instruction (Mnemonic)
- Comment (optional)

Label	Instr.	Operands	Comments
demoprg	MOVS MOVS ADDS LDR STR	R0 ,#0xA5 R1 ,#0x11 R0 ,R0 ,R1 R2 ,=0x2000 R0 , [R2]	; copy 0xA5 into register R0 ; copy 0x11 into register R1 ; add contents of R0 and R1 ; store result in R0 ; load 0x2000 into R2 ; store content of R0 at ; the address given by R2

■ Instruction Types

- **Data transfer**

- Copy content of one register to another register
- Load registers with data from memory
- Store register contents into memory

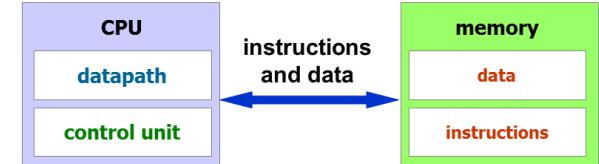
- **Data processing**

- Arithmetic operations → + - * / ...
- Logic operations → AND, OR, ...
- Shift / rotate operations

- **Control flow**

- Branches
- Function calls

- **Miscellaneous (various)**



Type	Frequency
Data transfer	43%
Control flow	23%
Arithmetic	15%
Compare	13%
Logical	5%
Miscellaneous	1%

Data processing

Instruction Set

Instructions Cortex-M

ARMv7-M
Architecture

ARMv6-M
Architecture

CT1
Instruction Set Cortex-M0
=
Subset Cortex-M3/4

Cortex-M4 FPU							
VABS	VADD	VCMP	VCMPE	VCVT	VCVTR	VDIV	VLDM
VLDR	VMLA	VMLS	VMOV	VMRS	VMSR	VMUL	VNEG
VNMLA	VMMLS	VNMUL	VPOP	VPUSH	VSQRT	VSTM	VSTR
VSUB	VFMA	VFMS	VFNMA	VFNMS			
Cortex-M4							
PKH	QADD	QADD16	QADD8	QASX	QDADD	QDSUB	QSAX
QSUB	QSUB16	QSUB8	SADD16	SADD8	SASX	SEL	SHADD16
SHADD8	SHASX	SHSAX	SHSUB16	SHSUB8	SMLABB	SMLABT	SMLATB
SMLATT	SMLAD	SMLALBB	SMLALBT	SMLALTB	SMLALTT	SMLALD	SMLAWB
SMLAWT	SMLSD	SMLS LD	SMLLA	SMMLS	SMMUL	SMUAD	SMULBB
Cortex-M3							
ADC	ADD	ADR	AND	ASR	B	SMULBT	SMULLT
CLZ	BFC	BFI	BIC	CDP	CLREX	SMULTB	SMULWT
CBNZ	CBZ	CMN	CMP	DBG	EOR	SMULWB	SMUSD
LDMIA	LDMDB	LDR	LDRB	LDRBT	LDRD	SSAT16	SSAX
LDREX	LDREXB	LDREXH	LDRH	LDRHT	LDRSB	SSUB16	SSUB8
LDRSBT	LDRSHT	LDRSH	LDRT	MCR	LSL	SXTAB	SXTAB16
LSR	MCRR	MLS	MLA	MOV	MOVT	SXTAH	SXTB16
MRC	MRRC	MUL	MVN	NOP	ORN	UADD16	UADD8
ORR	PLD	PLDW	PLI	POP	PUSH	UASX	UHADD16
RBIT	REV	REV16	REVSH	ROR	RRX	UHADD8	UHASX
Cortex-M0/M1							
BKPT	BLX	ADC	ADD	ADR	SDIV	SEV	SMLAL
BX	CPS	AND	ASR	B	SMULL	SSAT	STC
DMB		BL		BIC	STMIA	STMDB	STR
DSB	CMN	CMP	EOR		STRB	STRBT	STRD
ISB	LDR	LDRB	LDM		STREX	STREXB	STREXH
MRS	LDRH	LDRSB	LDRSH		STRH	STRHT	STRT
MSR	LSL	LSR	MOV		SUB	SXTB	SXTH
NOP	REV	MUL	MVN	ORR	TBB	TBH	TEQ
REV16	REVSH	POP	PUSH	ROR	TST	UBFX	UDIV
SEV	SXTB	RSB	SBC	STM	UMLAL	UMULL	USAT
SXTH	UXTB	STR	STRB	STRH	UXTB	UXTH	WFE
UXTH	WFE	SUB	SVC	TST	WFI	YIELD	IT

Instruction Set

■ Overview Cortex-M0

Instruction Type	Instructions
Move	MOV, MOVS
Load/Store	LDR, LDRB, LDRH, LDRSB, LDRSH, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Branch	B, BL, B{cond}, BX, BLX
Stack	POP, PUSH
Processor State	BKPT, CPS, MRS, MSR, SVC
No Operation	NOP
Hint / Synchronization	DMB, DSB, ISB, SEV, WFE, WFI, YIELD

Instruction Set

■ Cortex-M0: 16-bit Thumb instruction encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
0	0	0	opcode	imm5				Rm	Rd		shift by immediate, move register									
0	0	0	1	1	0	opc	Rm		Rn	Rd		add/subtract register								
0	0	0	1	1	1	opc	imm3			Rn	Rd		add/subtract immediate							
0	0	1	opcode	Rdn			imm8						add/sub./comp./move immediate							
0	1	0	0	0	0	0	opcode			Rm	Rd		data-processing register							
0	1	0	0	0	1	opcode	DN	Rm		Rd		special data processing								
0	1	0	0	0	1	1	L	Rm		0		0	0	0	branch/exchange					
0	1	0	0	1	Rt		PC-relative imm8						load from literal pool							
0	1	0	1	opcode		Rm		Rn	Rt		load/store register offset									
0	1	1	B	L	imm5				Rn	Rt		load/store word/byte imm. offset								
1	0	0	0	L	imm5				Rn	Rt		load/store halfword imm. offset								
1	0	0	1	L	Rt		SP-relative imm8						load from or store to stack							
1	0	1	0	SP	Rd		imm8						add to SP or PC							
1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	miscellaneous				
1	1	0	0	L	Rn		register list						load/store multiple							
1	1	0	1	cond			imm8						conditional branch							
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x	undefined instruction				
1	1	0	1	1	1	1	1	imm8						service (system) call						
1	1	1	0	0	imm11						unconditional branch									
1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	32-bit instruction				
1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	32-bit instruction				

Note: There are some inconsistencies in ARM's use of the notation Rd (destination) and Rt (target). The slides use the ARMv6-M Architecture Reference Manual (ARM DDI 0419C (ID092410)) as a reference.

Instruction Set

■ Cortex-M0 Example Program

- Assembler converts each Assembly instruction to 16-bit opcode
- Memory addresses in steps of 2 (Zweierschritte)
 - Reason: opcodes are 16-bit (two bytes) long, e.g. **0x20A5**

Memory address	Opcode	demoprg	MOVS R0, #0xA5 ; copy 0xA5 into R0
00000000	20A5	MOVS R1, #0x11 ; copy 0x11 into R1	
00000002	2111	ADDS R0, R0, R1 ; add contents of R0 and R1	
00000004	1840		; store result in R0
00000006	4A00	LDR R2, =0x2000 ; load address into R2	
00000008	6010	STR R0, [R2] ; store content of R0 at	
			; the address given by R2
1)	0000000A	00002000	

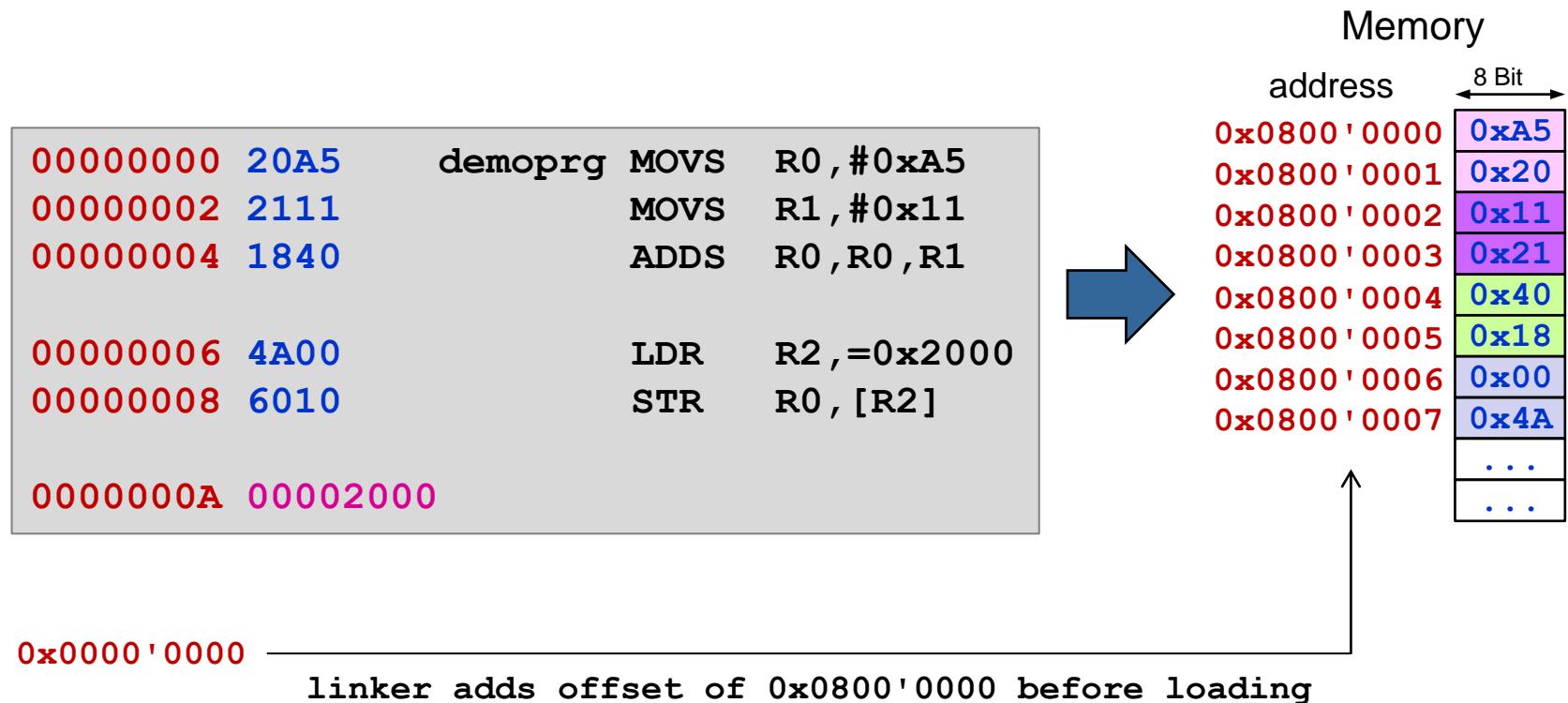
1) Assembler places constant for the LDR instruction at the end of the code

Binary ← generated by Assembler (tool) **Source code**

Instruction Set

■ Load program into memory

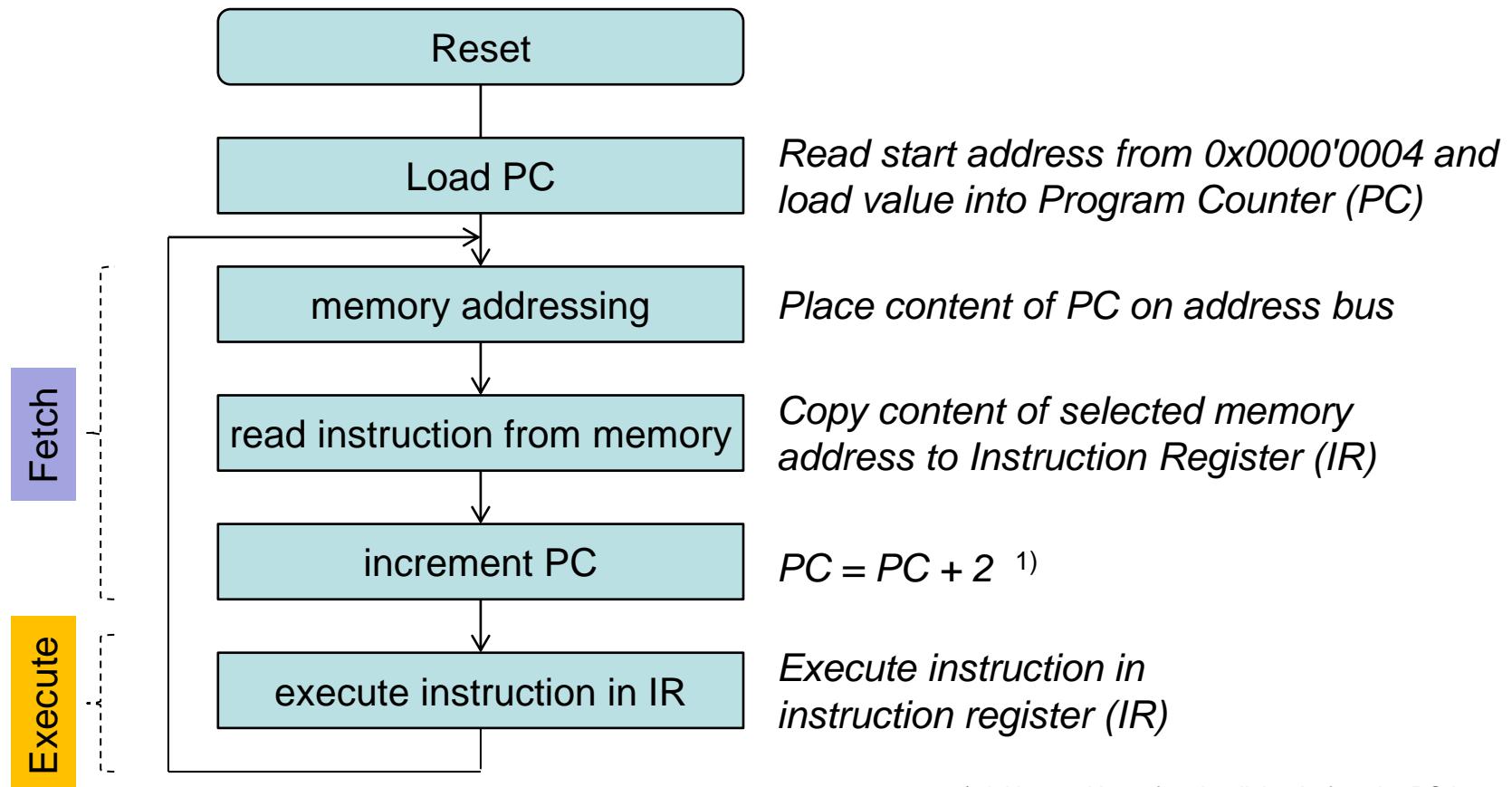
- Example assumes code area in memory at address **0x0800'0000** ¹⁾



1) address of code area depends on individual system

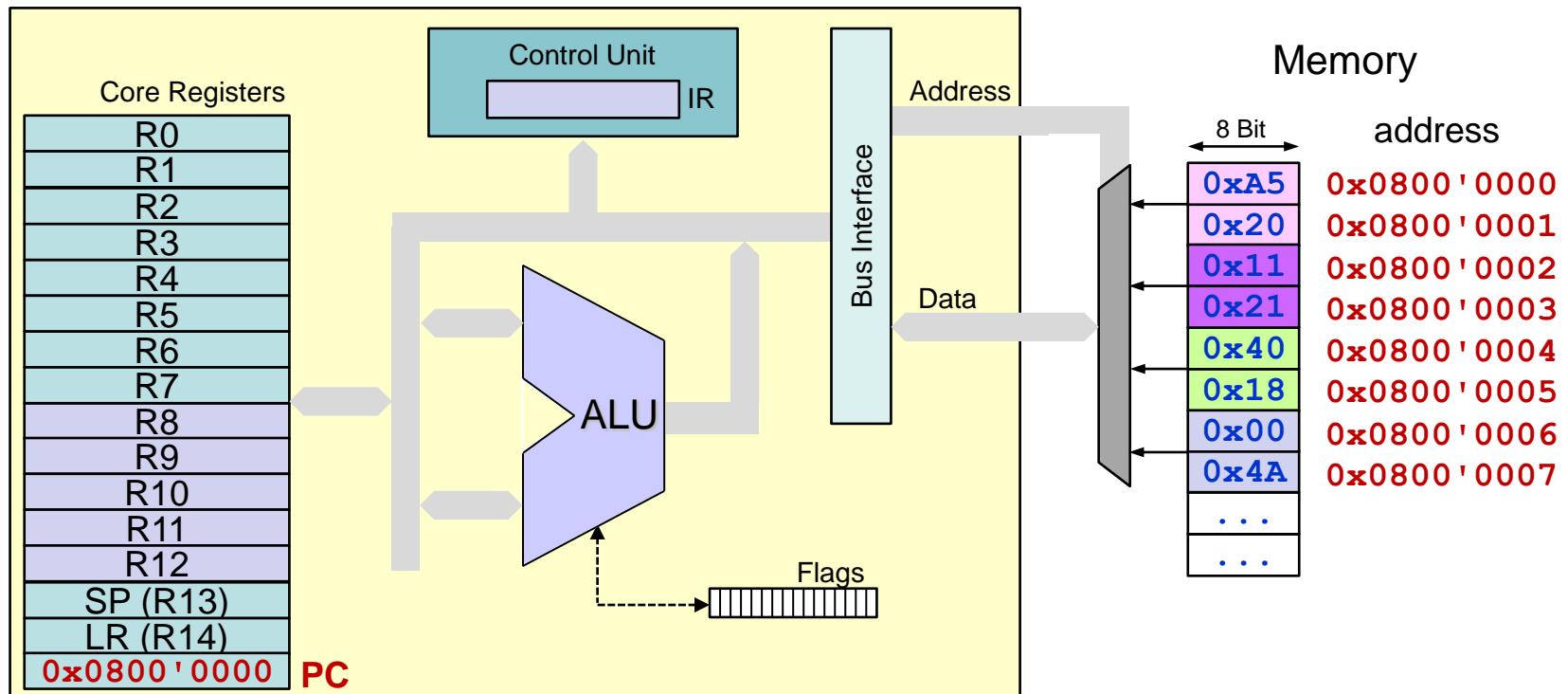
Program Execution

■ Sequence



Program Execution

- "Reset": Read 0x0000'0004 → PC

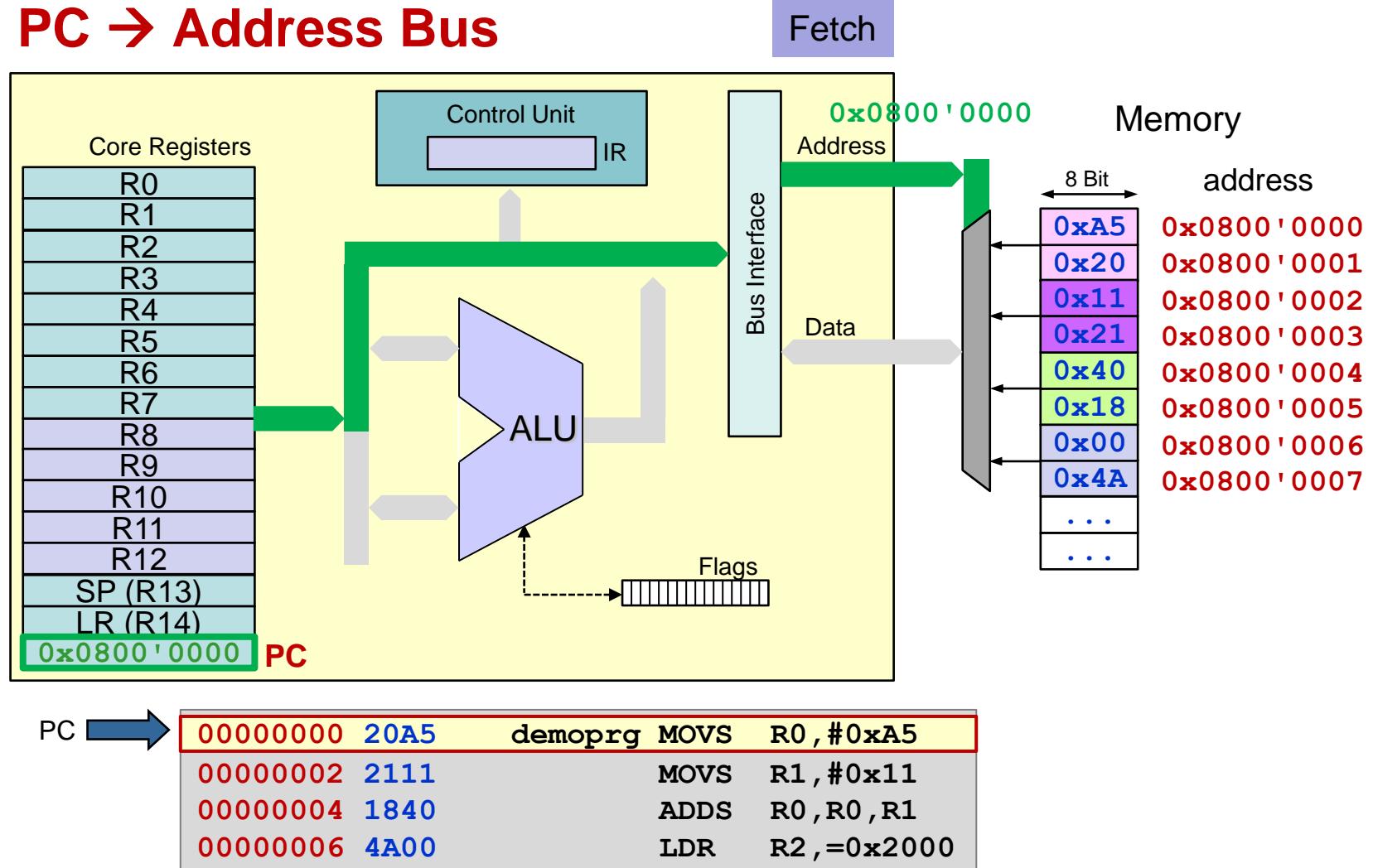


PC

00000000	20A5	demoprg	MOVS	R0 , #0xA5
00000002	2111		MOVS	R1 , #0x11
00000004	1840		ADDS	R0 , R0 , R1
00000006	4A00		LDR	R2 , =0x2000

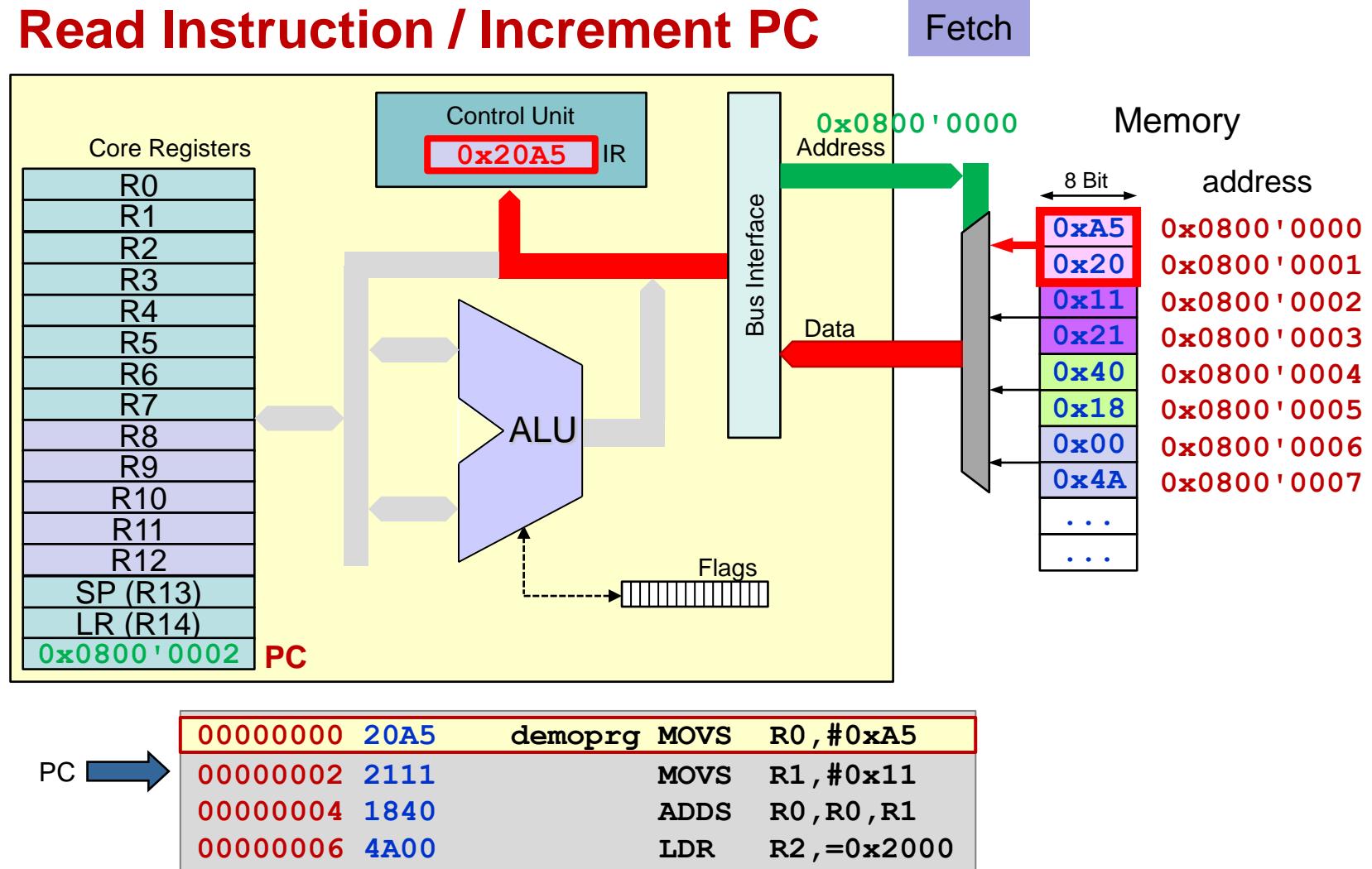
Program Execution

■ PC → Address Bus



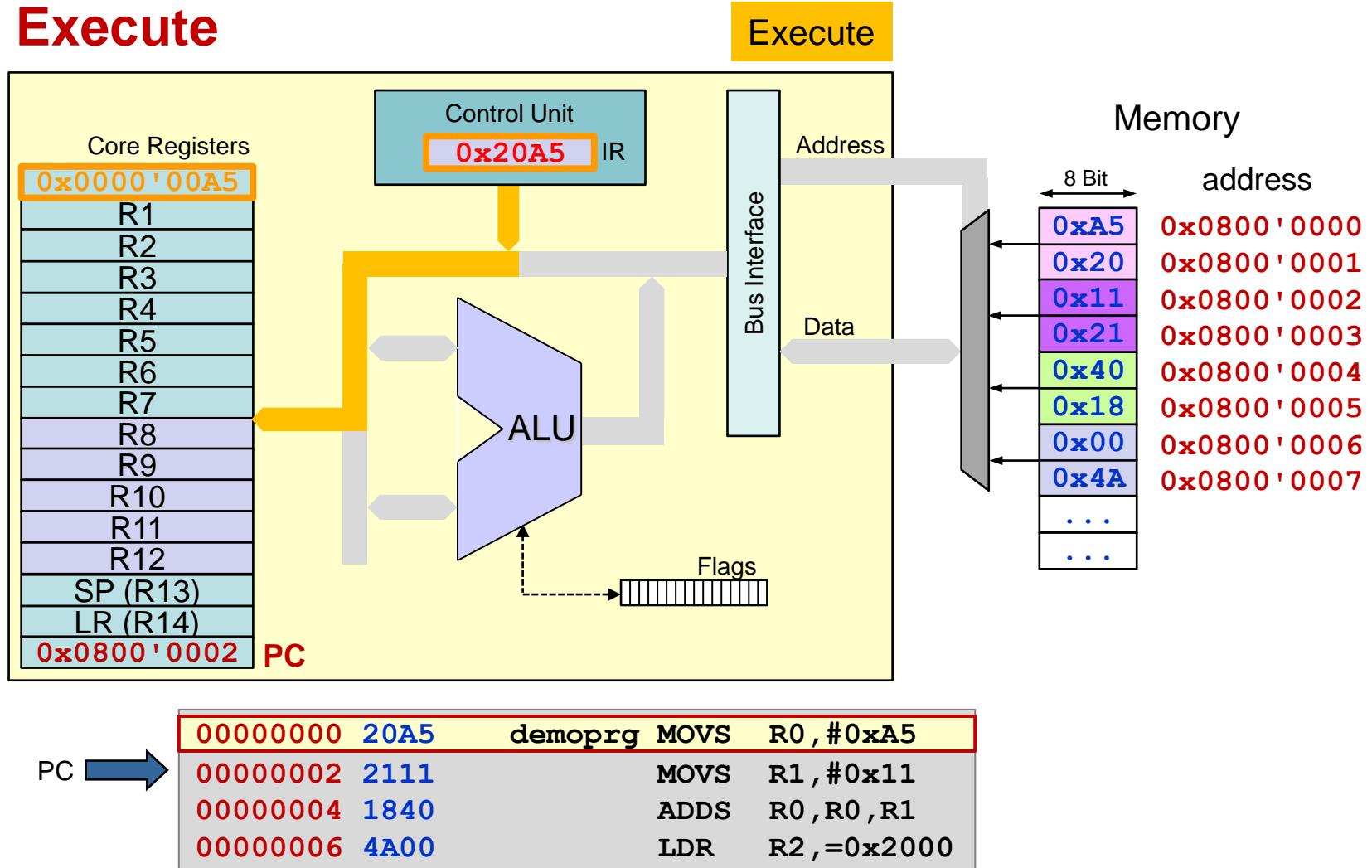
Program Execution

■ Read Instruction / Increment PC



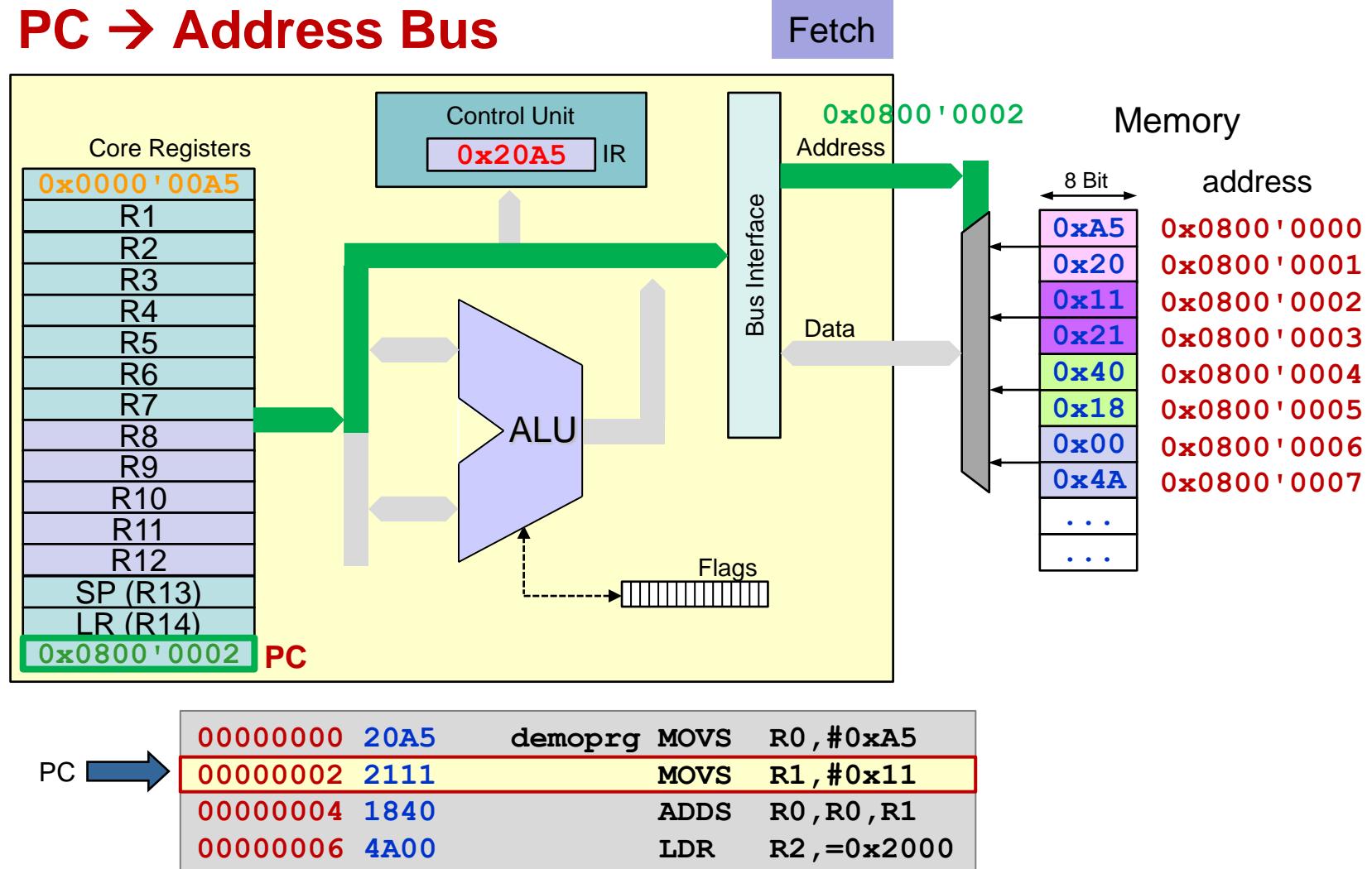
Program Execution

Execute



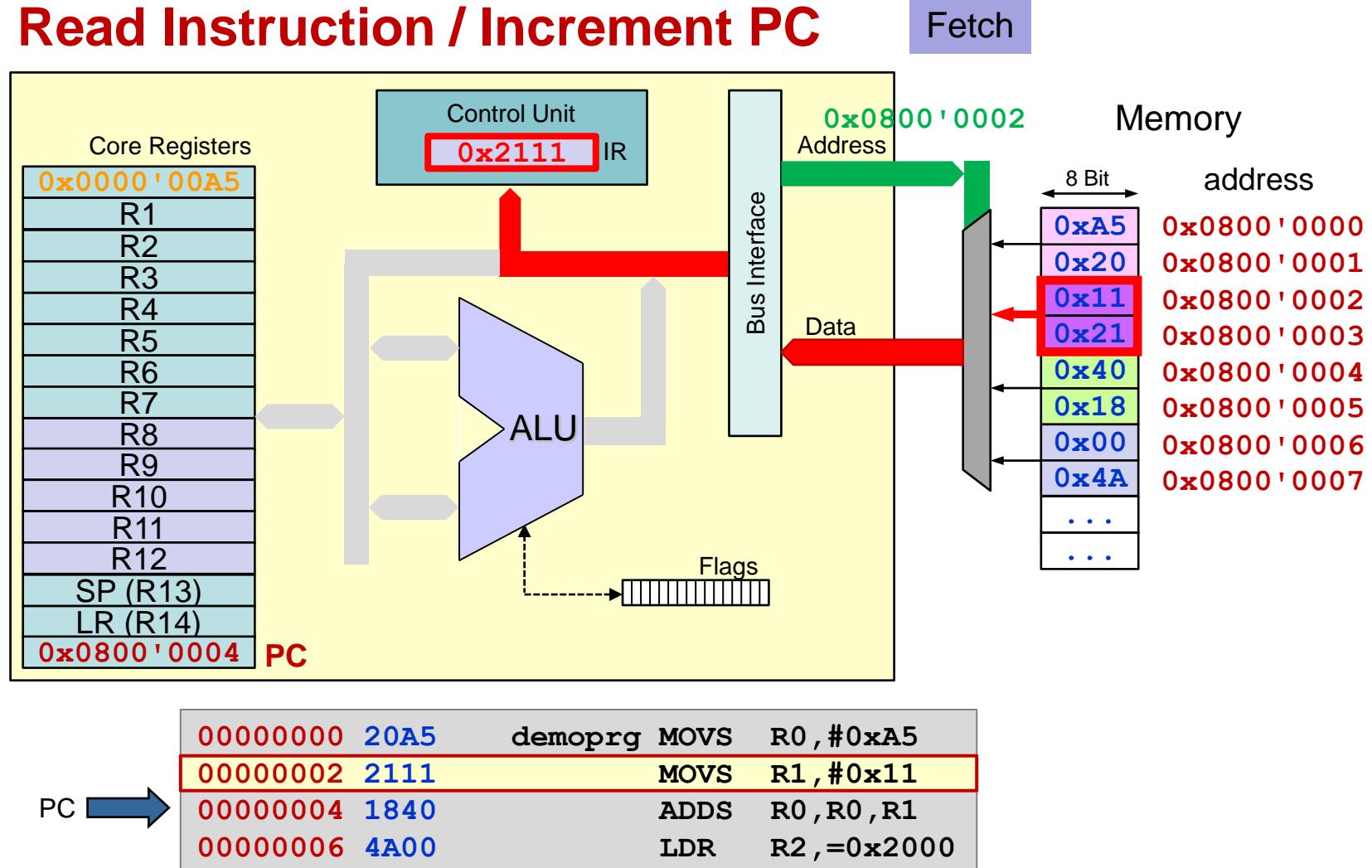
Program Execution

■ PC → Address Bus



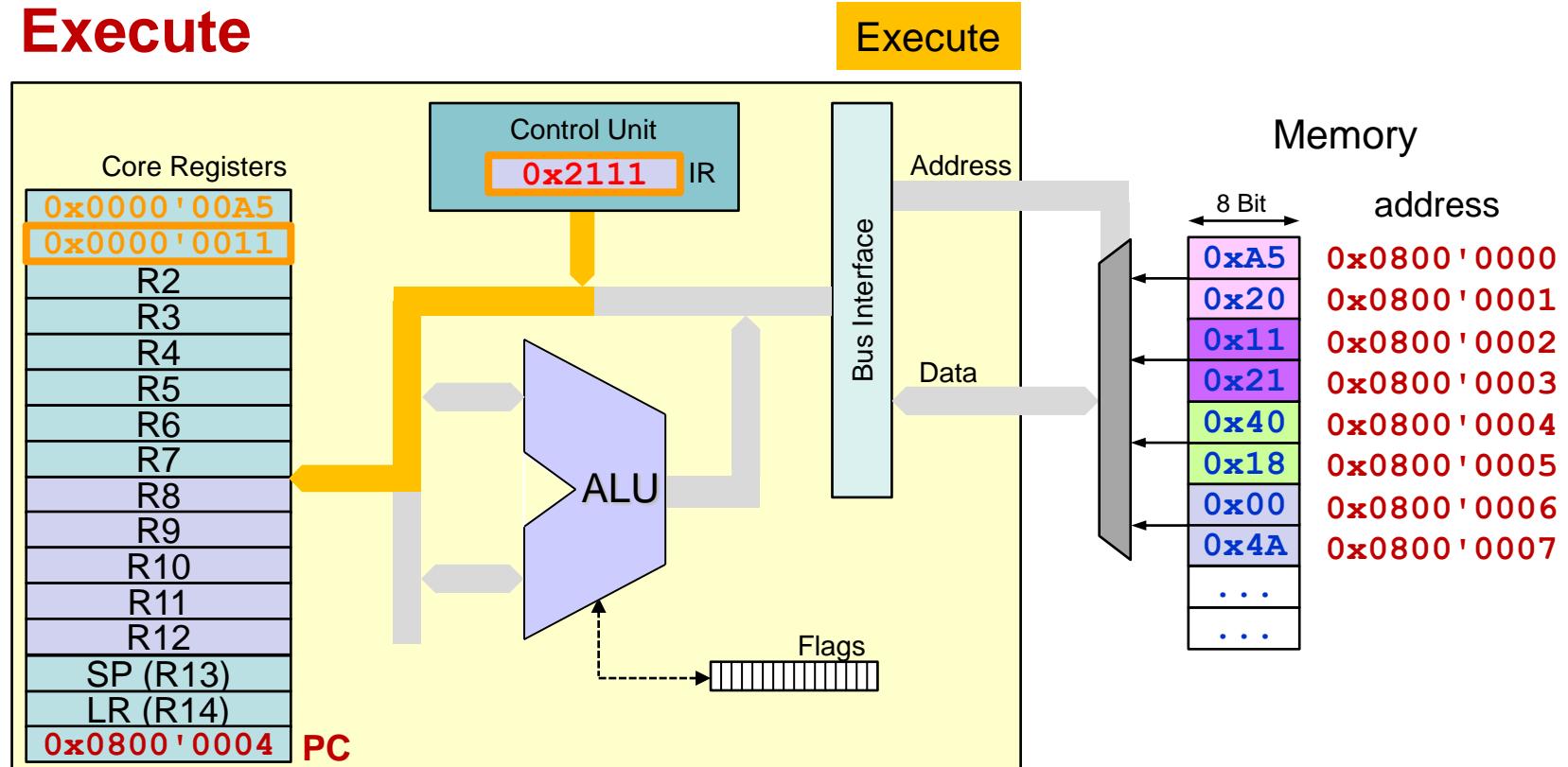
Program Execution

■ Read Instruction / Increment PC



Program Execution

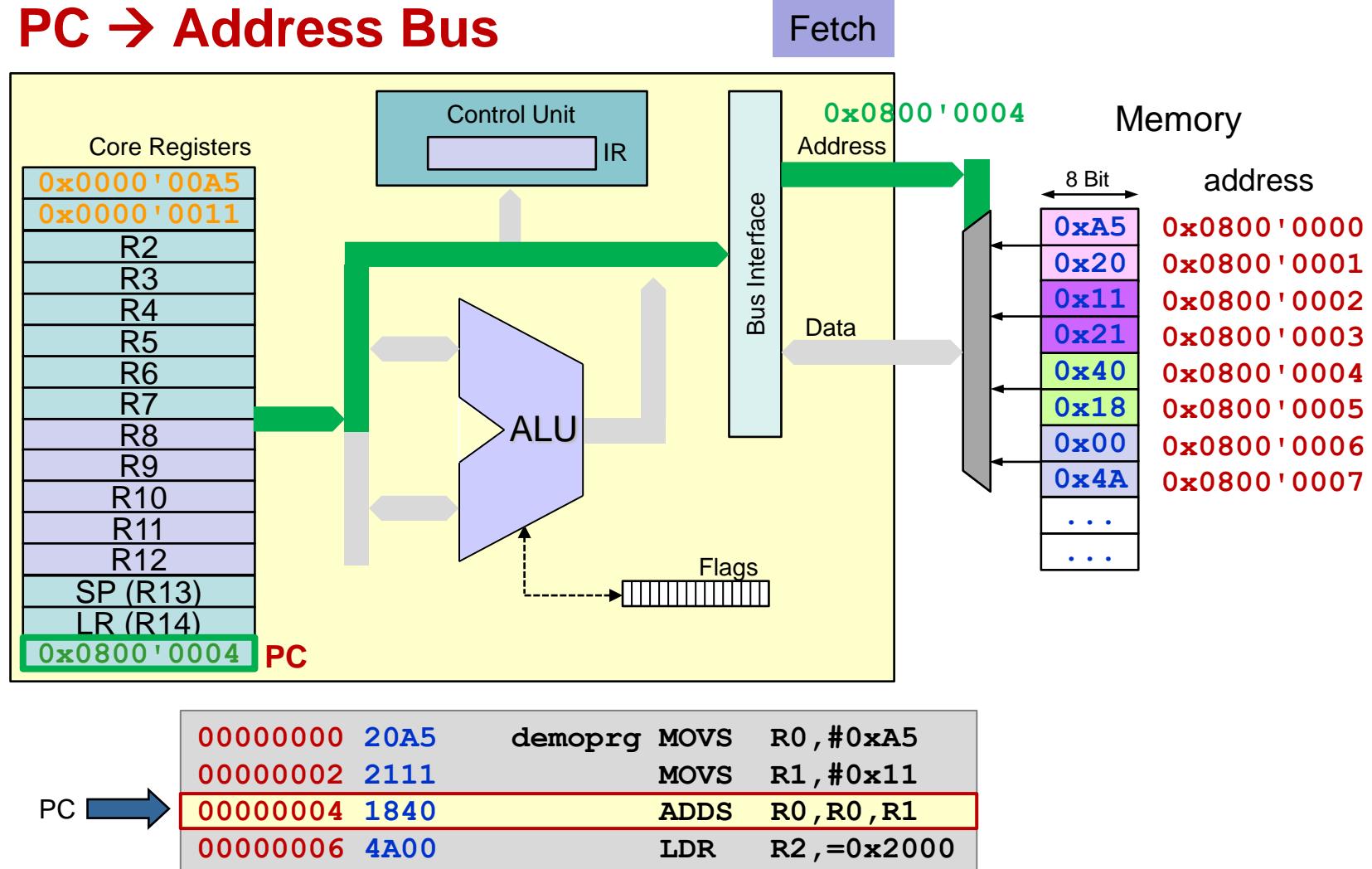
Execute



00000000	20A5	demoprg	MOVS	R0 , #0xA5
00000002	2111		MOVS	R1 , #0x11
00000004	1840		ADDS	R0 , R0 , R1
00000006	4A00		LDR	R2 , =0x2000

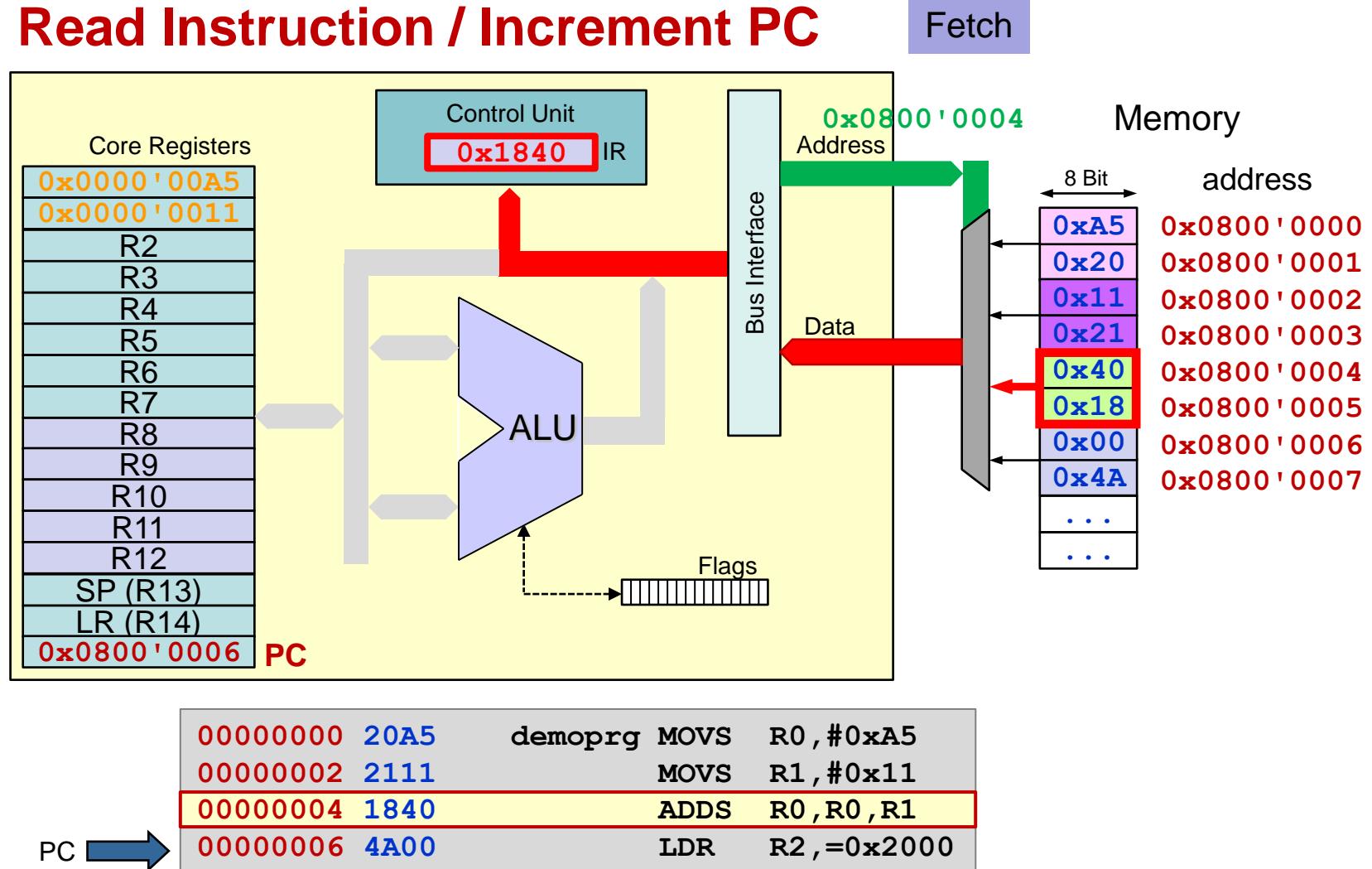
Program Execution

■ PC → Address Bus



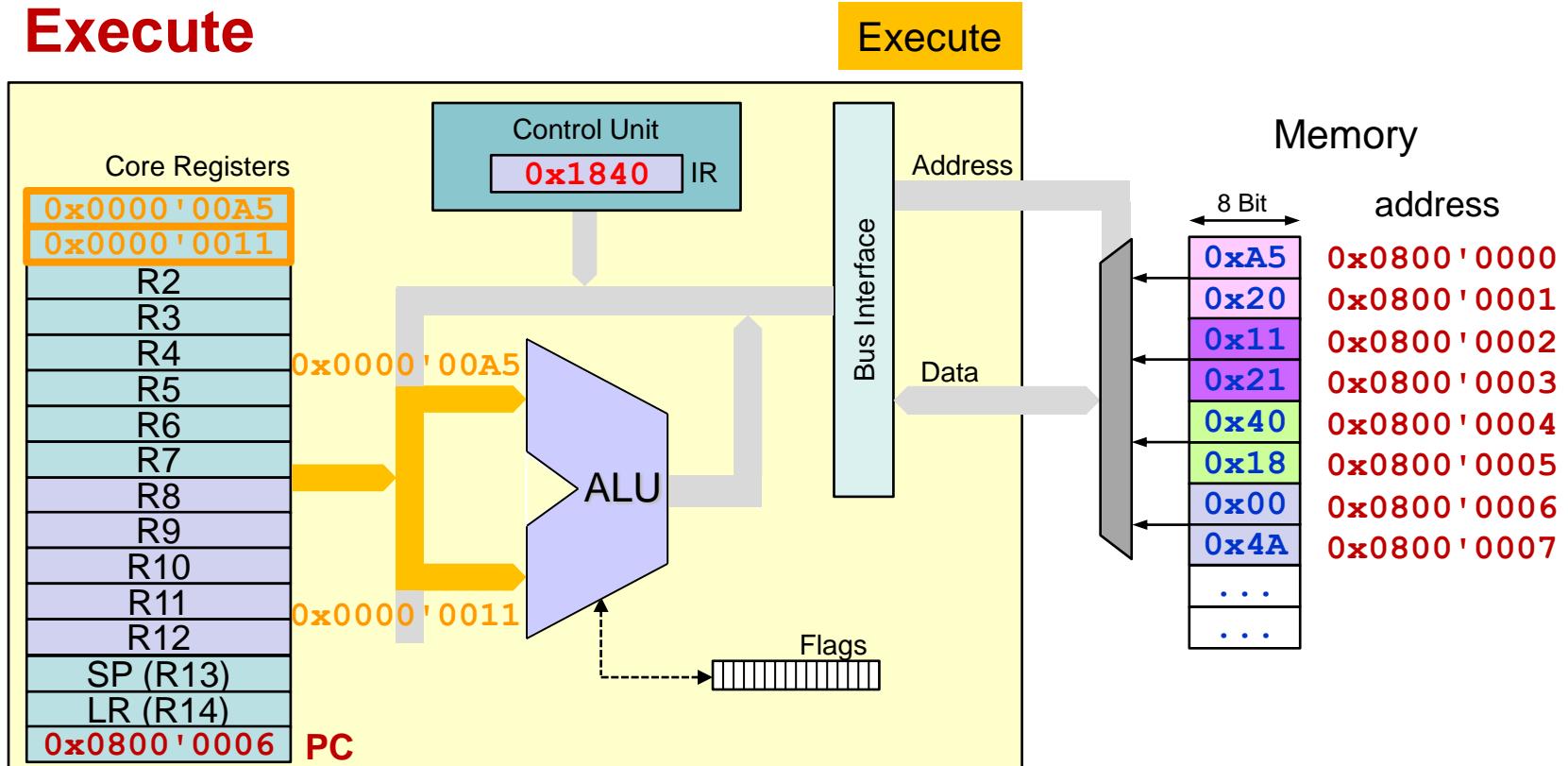
Program Execution

■ Read Instruction / Increment PC



Program Execution

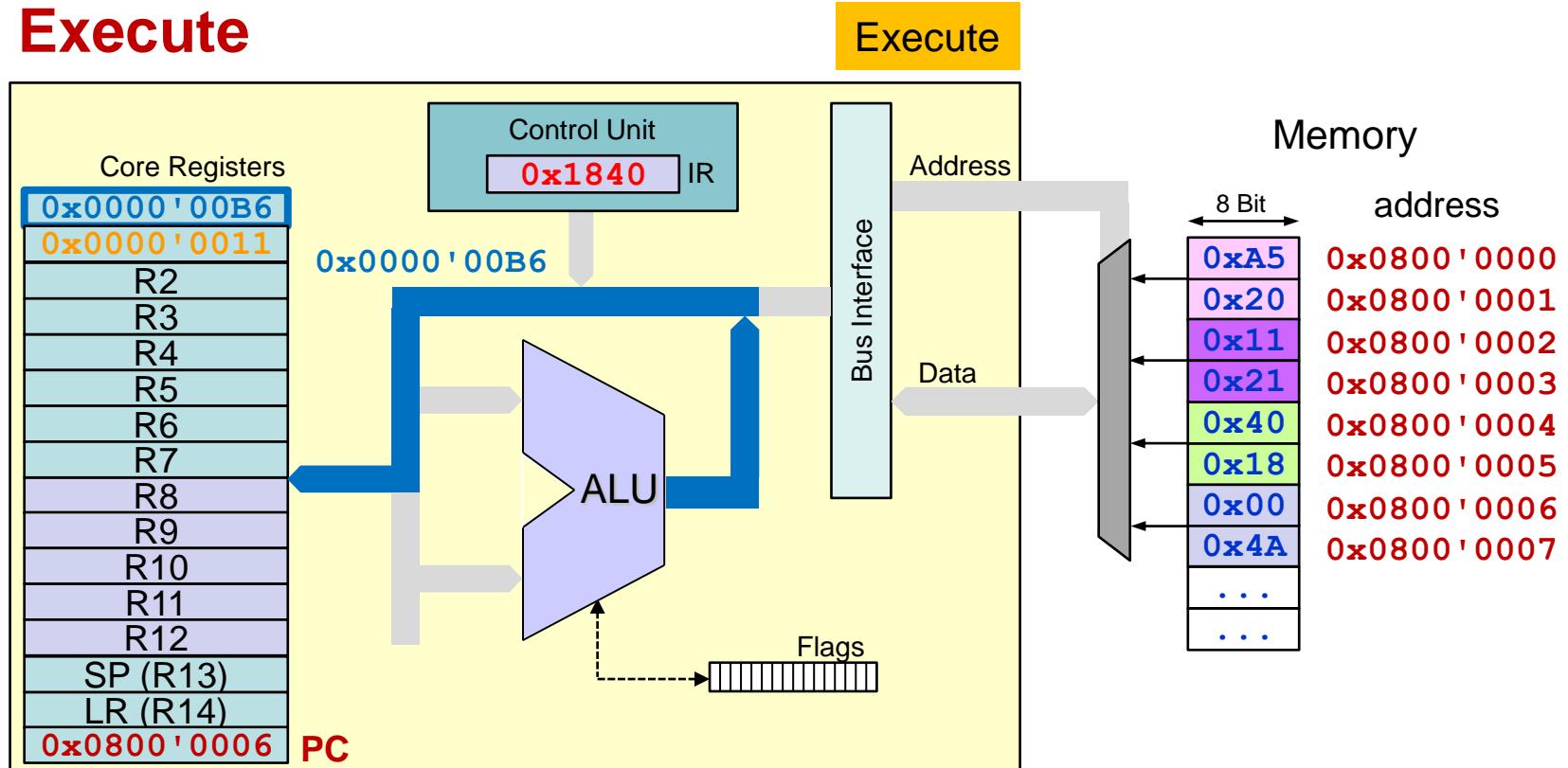
Execute



00000000	20A5	demoprg	MOVS	R0 , #0xA5
00000002	2111		MOVS	R1 , #0x11
00000004	1840		ADDS	R0 , R0 , R1
00000006	4A00		LDR	R2 , =0x2000

Program Execution

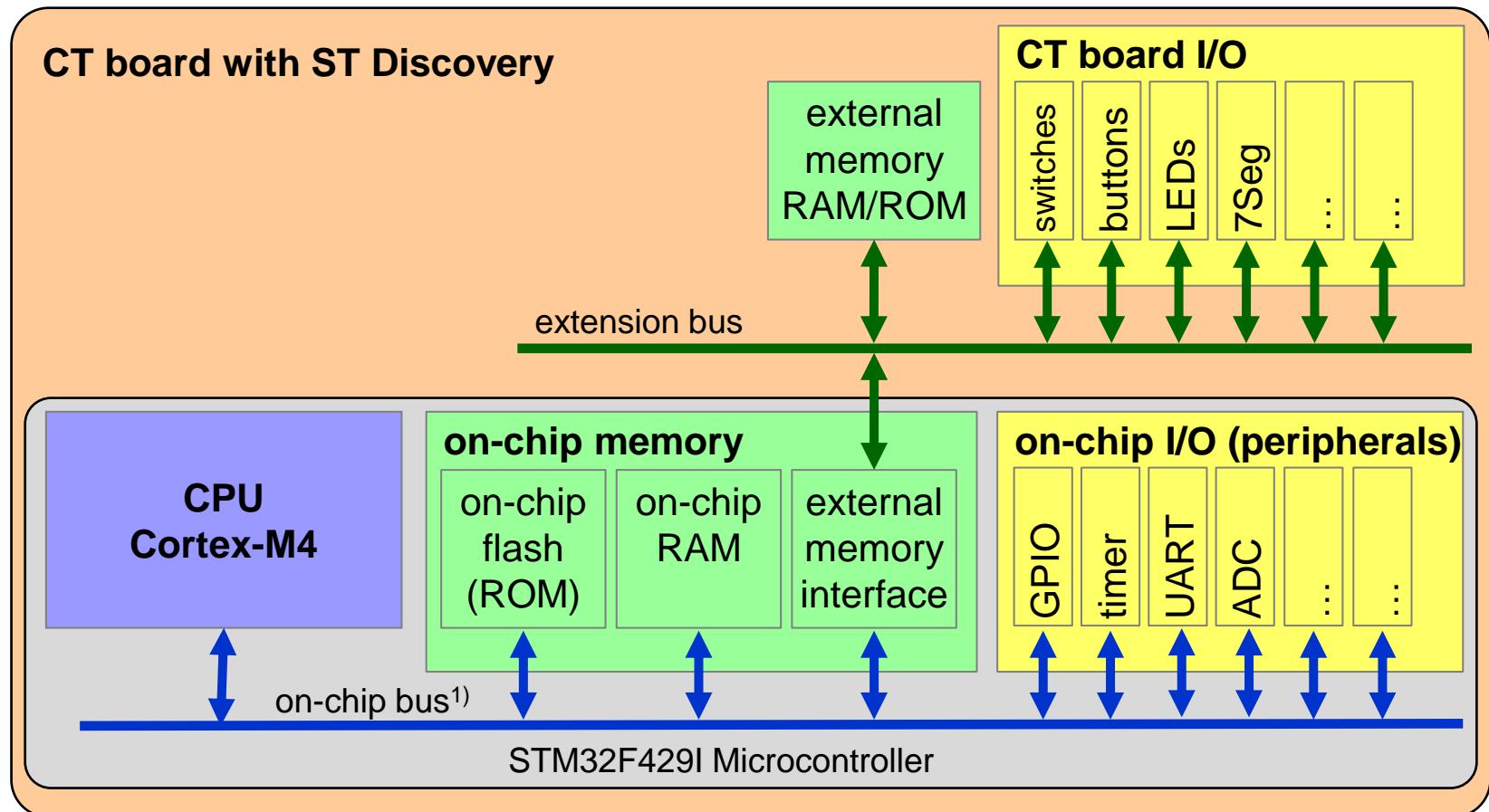
Execute



00000000	20A5	demoprg	MOVS	R0 , #0xA5
00000002	2111		MOVS	R1 , #0x11
00000004	1840		ADDS	R0 , R0 , R1
00000006	4A00		LDR	R2 , =0x2000

Memory Map

■ Hardware View



¹⁾ The implementation partitions the on-chip bus into several busses called AHBx and APBx

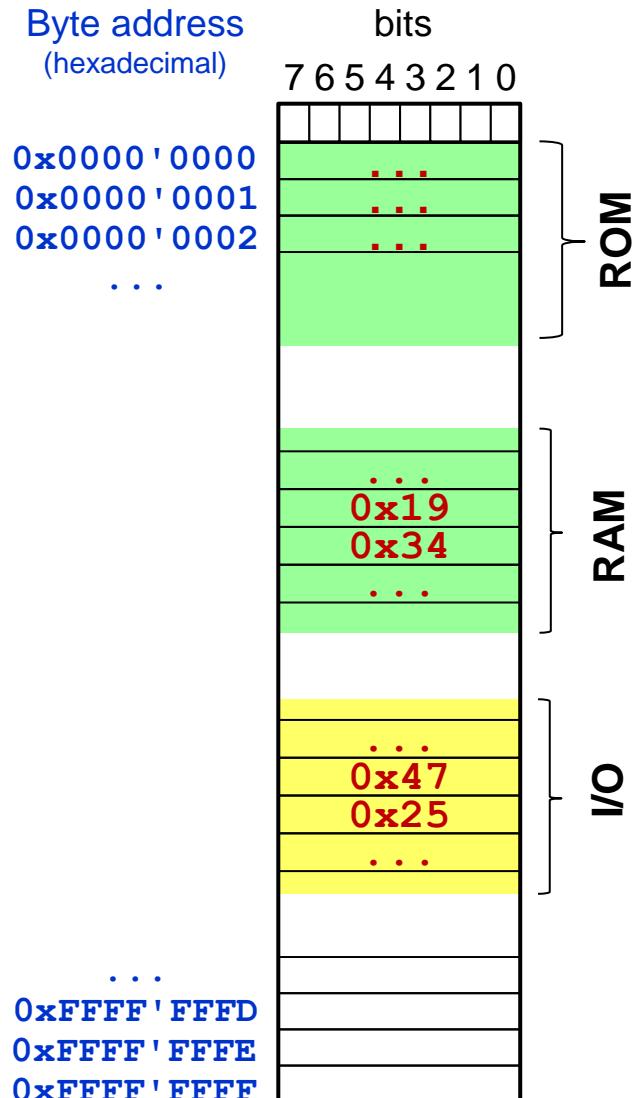
Memory Map

■ Memory Layout

- System Address Map
- Graphical layout of main memory
- Linear array of bytes
- What is located where (at which address) in memory?
 - Location of RAM (readable and writable)
 - Location of ROM (only readable)
 - Location of I/O registers

Memory maps in CT1/CT2 will be drawn with lowest address at the top and highest address at the bottom. This simplifies work with assembly listing and tables.

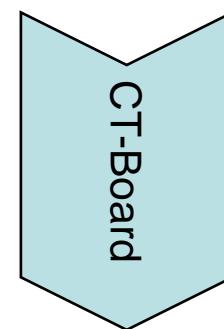
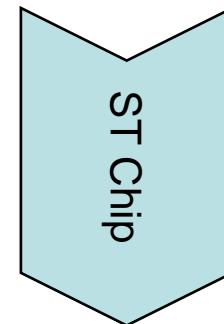
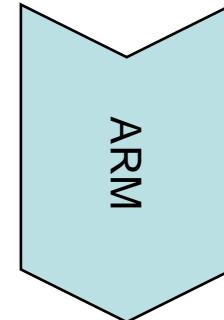
Caution: ARM and ST documentation are the other way round.
Lowest address at the bottom and highest address at top.



Memory Map

■ Address Allocation

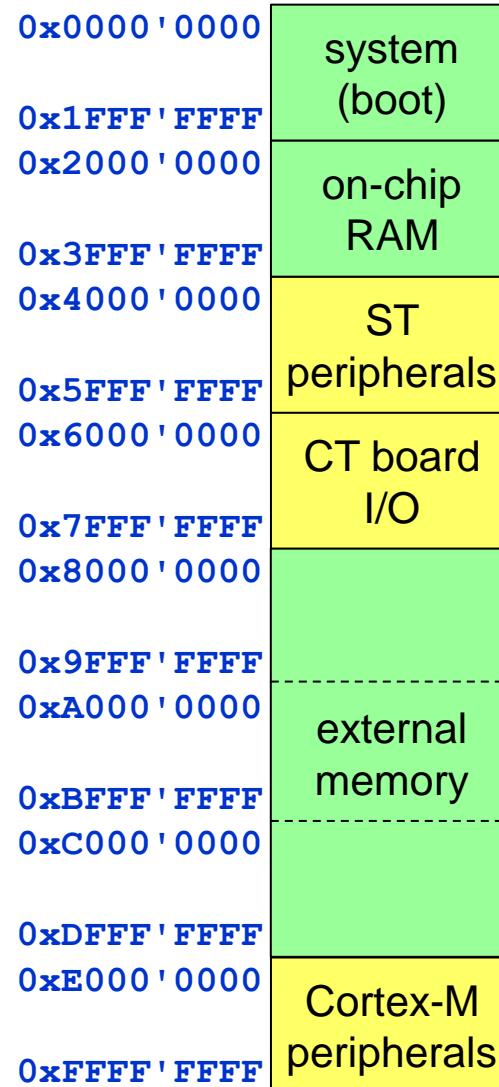
- ARM policies
 - Cortex-M specific
 - guide lines for chip manufacturer
- ST design decisions
 - chip specific
 - number and size of on-chip RAMs
 - size of flash
 - control register for peripherals
- CT board design decisions
 - board specific
 - LEDs, switches, etc



Memory Map

■ CT-Board

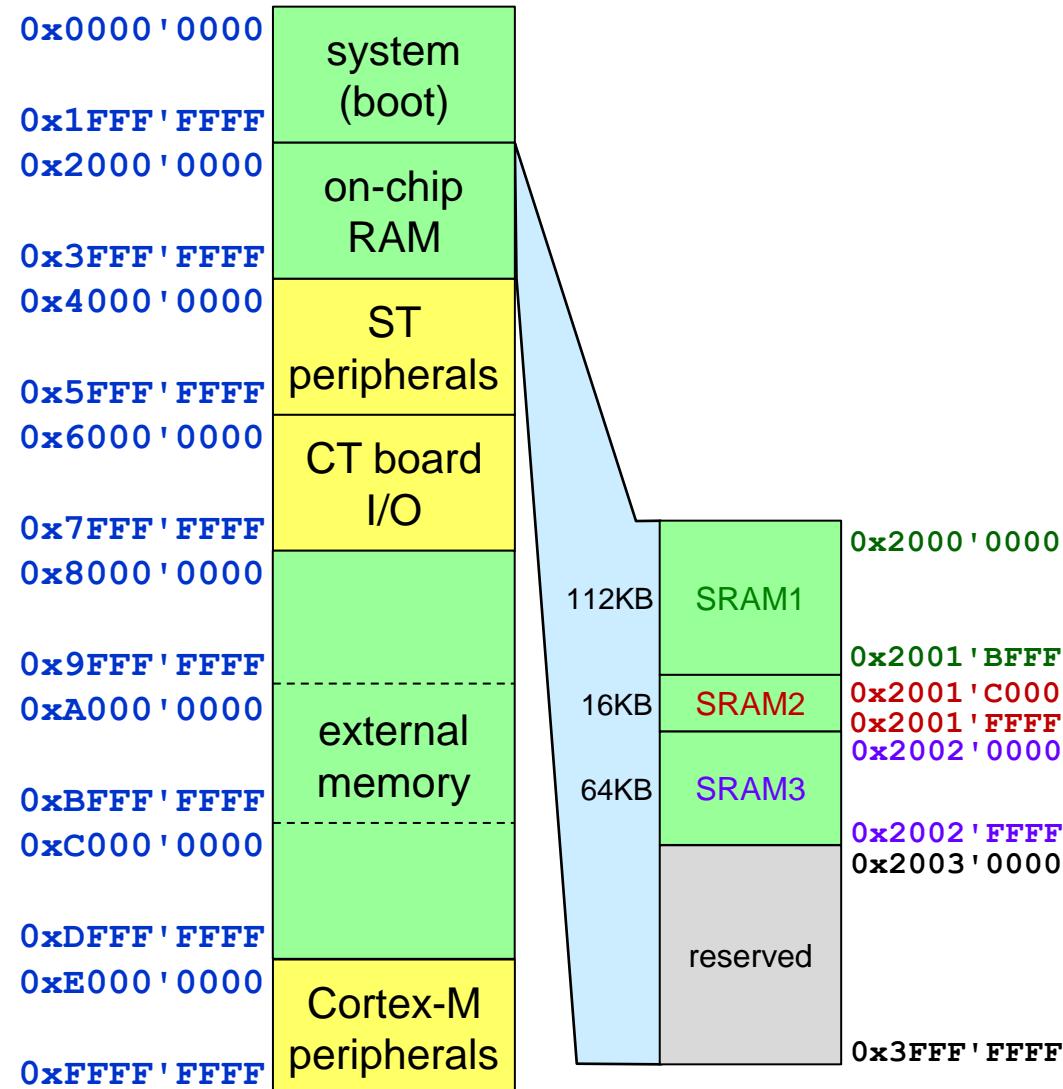
- Address space
4 GByte = 2^{32}
- From **0x0000'0000** to **0xFFFF'FFFF**
- Partitioned into
8 blocks of
512-MByte each



Memory Map

■ ST chip specific

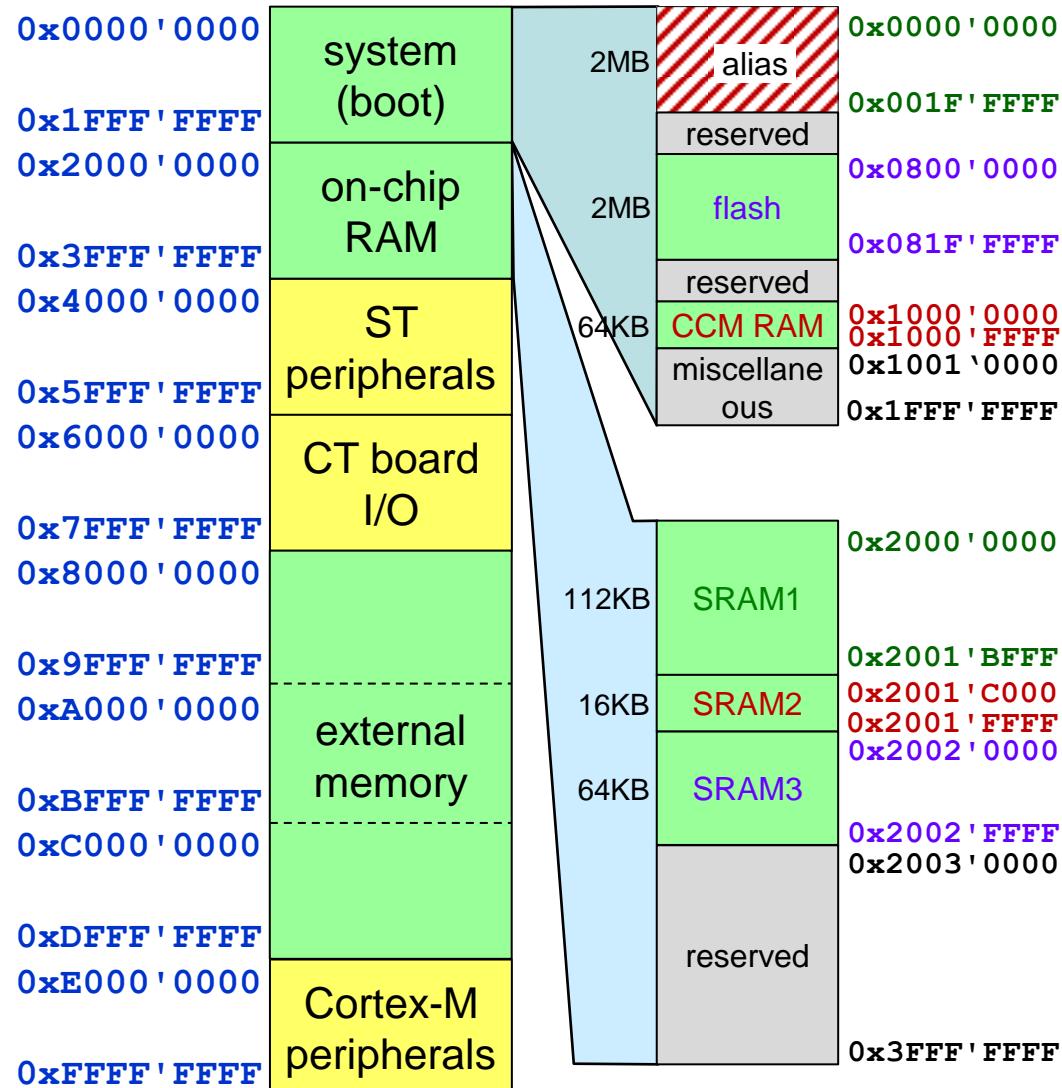
- SRAM1
 - 112 KByte
- SRAM2
 - 16 KByte
- SRAM3
 - 64 KByte



Memory Map

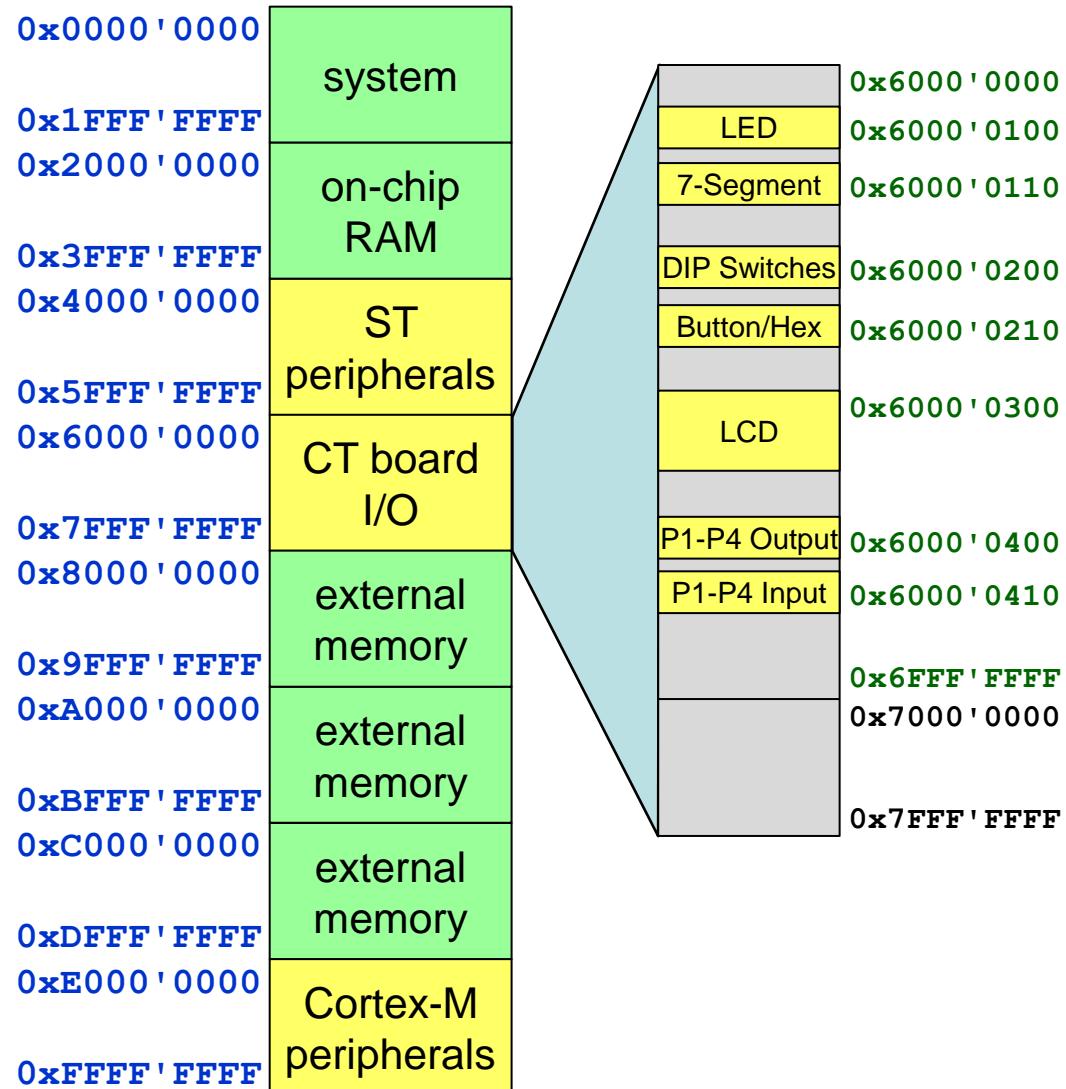
■ ST chip specific

- Flash
 - non-volatile memory
- CCM RAM
 - core coupled memory
 - very fast RAM
- Alias
 - user configurable mirror
 - physical memory can appear at two locations



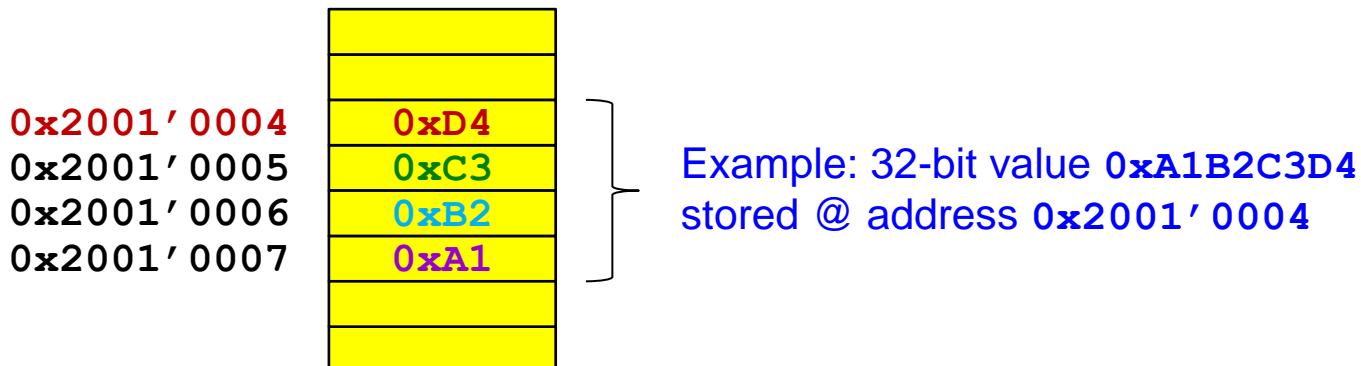
Memory Map

■ CT-Board I/O



■ Multi-byte Integer Types

- Usually memory is organized in bytes
 - one address per byte → reasons: space and history
- An integer type often requires several bytes
 - e.g. 4 byte addresses are required to store a 32-bit integer



Integer Types

■ Integer Types in C

- Sizes of integer types are platform dependent

Sizes in byte

8051

char	1
short	2
int	2
long int	4
char *	2

Cortex-Mx: Keil (ARM)

char	1
short	2
int	4
long int	4
long long int	8
void *	4

x86-64 (i7): gcc

char	1
short	2
int	4
long int	8
long long int	8
void *	8

Integer Types

■ ARM Cortex-M

C99 / specified width



C-Type – unsigned integers	Size	Term	inttypes.h / stdint.h
<code>unsigned char</code>	8 Bit	Byte	<code>uint8_t</code>
<code>unsigned short</code>	16 Bit	Half-word	<code>uint16_t</code>
<code>unsigned int</code>	32 Bit	Word	<code>uint32_t</code>
<code>unsigned long</code>	32 Bit	Word	<code>uint32_t</code>
<code>unsigned long long</code>	64 Bit	Double-word	<code>uint64_t</code>

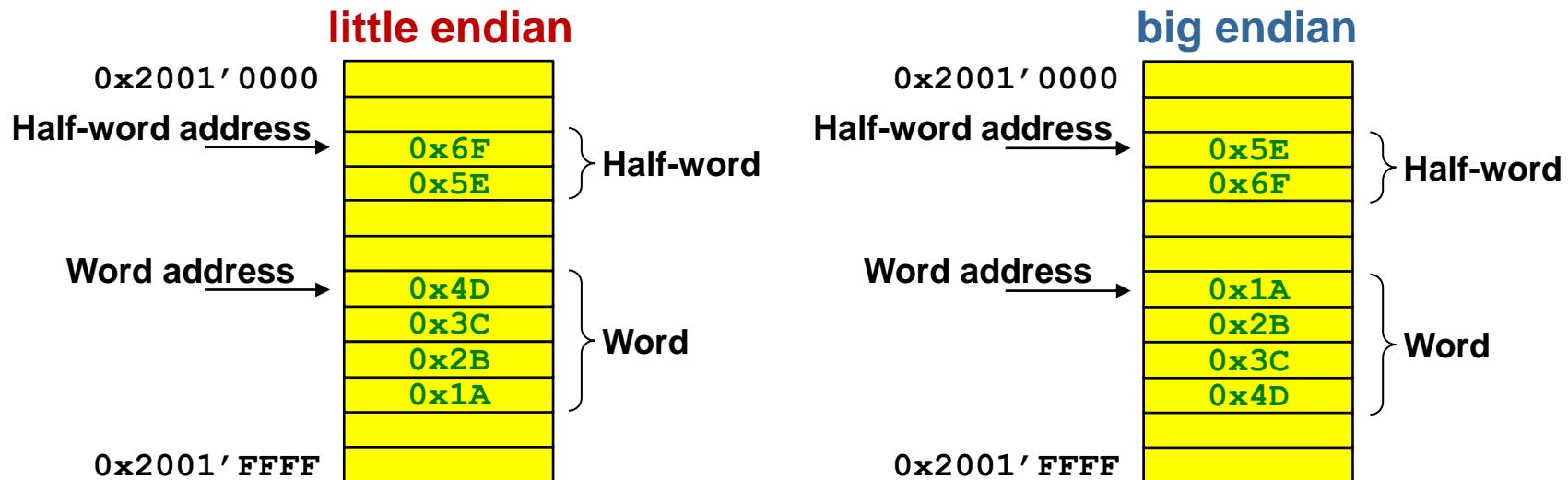
C-Type – signed integers	Size	Term	inttypes.h / stdint.h
<code>signed char</code>	8 Bit	Byte	<code>int8_t</code>
<code>short</code>	16 Bit	Half-word	<code>int16_t</code>
<code>int</code>	32 Bit	Word	<code>int32_t</code>
<code>long</code>	32 Bit	Word	<code>int32_t</code>
<code>long long</code>	64 Bit	Double-word	<code>int64_t</code>

Integer Types

■ How are groups of bytes arranged in memory?

- little endian → *least significant byte at lower address*
e.g. Intel x86, Altera Nios, ST ARM (STM32)
- big endian → *most significant byte at lower address*
e.g. Freescale (Motorola), PowerPC

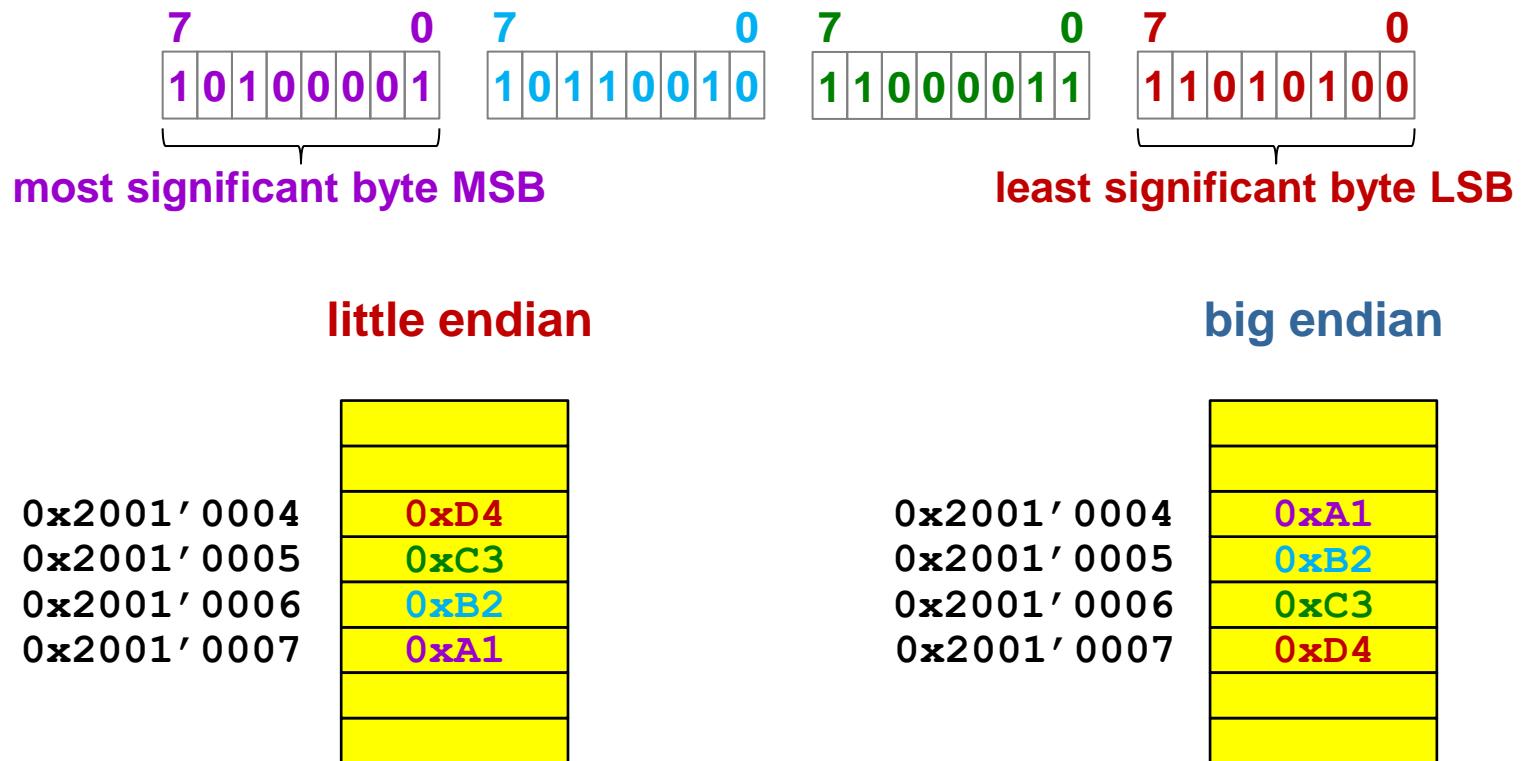
Examples: **0x1A2B'3C4D** for Word and **0x5E6F** for Half-word



Integer Types

■ Example

- Store Word **0xA1B2' C3D4** at Address **0x2001' 0004**



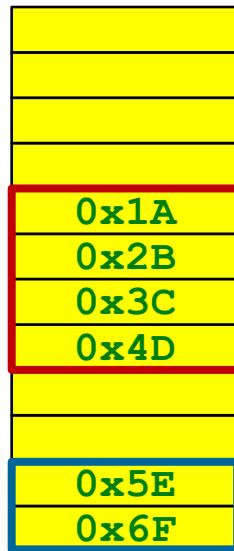
Integer Types

■ Alignment

- Half-word aligned Variables aligned on even addresses
- Word aligned Variables aligned on addresses that are divisible by four

Word Aligned

0x2001' 0000



0x2001' 0004

0x2001' 0008

Half-word Aligned

0x2001' 0000

0x2001' 0002

0x2001' 0004

0x2001' 0006

0x2001' 0008

0x2001' 000A

Object File Sections

■ CODE

- Read-only → RAM or ROM
- Instructions (opcodes)
- Literals ¹⁾

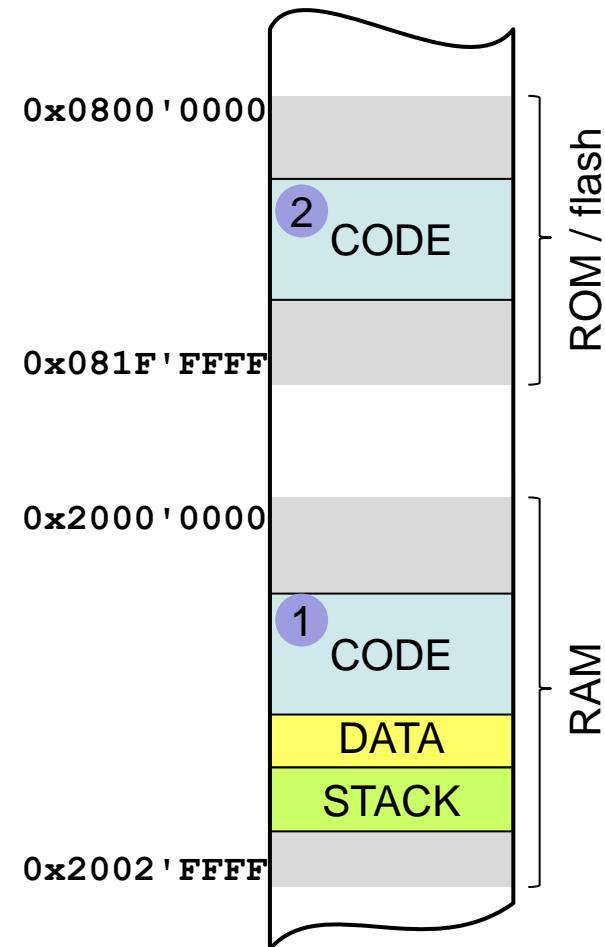
1 2

■ DATA ²⁾

- Read-write → RAM
- Global variables
- *static* variables in C
- Heap in C → `malloc()`

■ STACK

- Read-write → RAM
- Function calls / parameter passing
- Local variables and local constants



¹⁾ Literal: a fixed/constant value in source code

Object File Sections

■ Assembly Program Structure

- AREA directive

	AREA	MyCode, CODE, READONLY	
	ENTRY		Define code area to include your program
start	MOVS	R4, #12	
	ADDS	R3, R4, #5	
	B	start	
	AREA	MyData, DATA, READWRITE	
byte_var	DCB	0x1A, 0x00	Define data area to store global variables, etc.
hw_var	DCW	0x2B3C	
word_var	DCD	0x4D5E6F70	
	AREA	STACK, NOINIT, READWRITE	
stack_mem	SPACE	0x00000400	Define stack area to reserve space for stack

Memory Map / Object File Sections

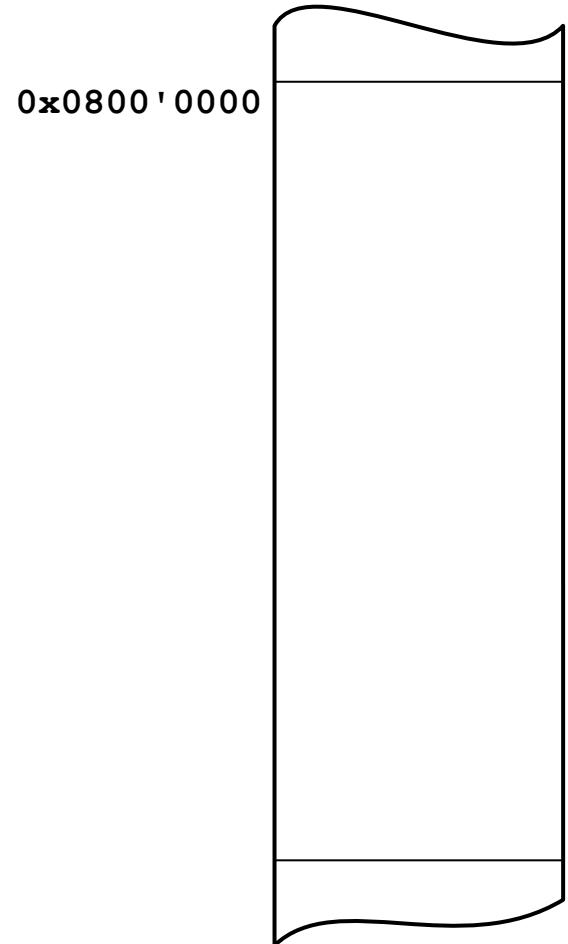
Assume

- A program uses the following memory segments during execution

- Code 0x0800'1000 to 0x0800'17FF
- Data 0x2001'0000 to 0x2001'01FF
- Stack 0x2001'0200 to 0x2001'05FF

- Exercise

- Draw the memory map with the three sections
- For each section mark the first and the last address with its value
- How many memory cells (bytes) does each section contain?
- For each section: In which STM32F4 memory is it located?

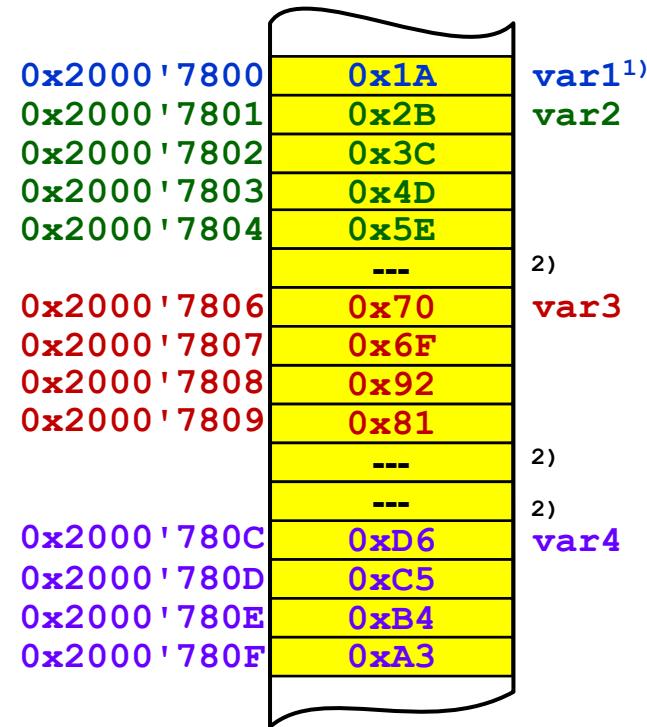


Object File Sections

■ Memory Allocation in Assembly

- Directives for initialized data
 - **DCB** bytes
 - **DCW** half-words (half-word aligned)
 - **DCD** words (word aligned)
 - Can be located in **DATA** or **CODE** area

```
AREA example1, DATA, READWRITE
var1    DCB   0x1A
var2    DCB   0x2B, 0x3C, 0x4D, 0x5E
var3    DCW   0x6F70, 0x8192
var4    DCD   0xA3B4C5D6
```



¹⁾ if we assume that `example1` starts at `0x2000'7800`

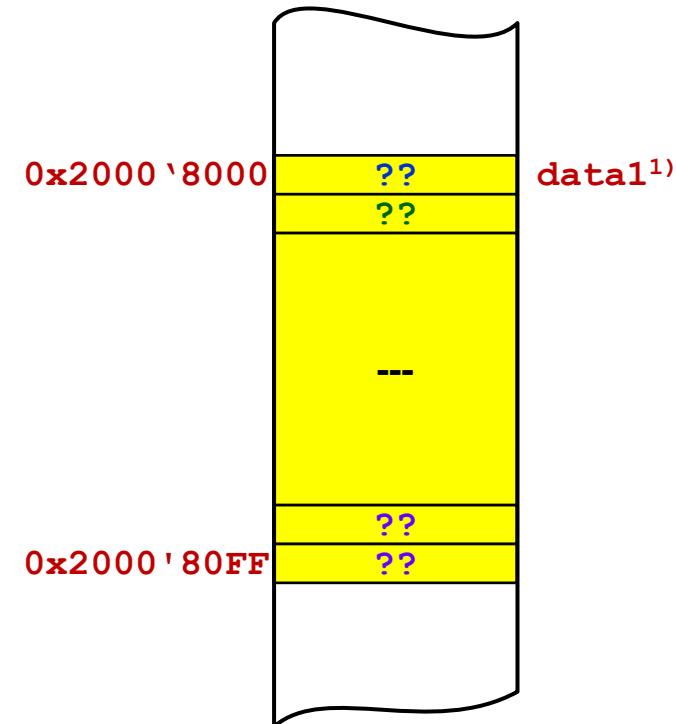
²⁾ Padding bytes introduced for alignment

Object File Sections

■ Memory Allocation in Assembly

- Directives for uninitialized data
 - **SPACE** or % with number of bytes to be reserved
 - Reserves number of bytes without initializing them

```
AREA example2, DATA, READWRITE
data1 SPACE 256
```



¹⁾ if we assume that example2 starts at **0x2000`8000**

Object File Sections

■ Code Example

show_variables.c

```
uint32_t g_init_var = 0x4D5E6F70;

uint32_t g_noinit_var;

const uint32_t g_const = 0xDDDEFF00;

void show_variables(void)
{
    uint32_t local_var;

    const uint32_t local_const =
                    0x7261504F;

    static uint32_t static_local_var;

    local_var = 0xD4E5F607;
    ...
}
```

Into which sections will the colored objects be allocated?

Object File Sections

■ Code Example

show_variables.c

```
uint32_t g_init_var = 0x4D5E6F70;
```

global variable with initialization

```
uint32_t g_noinit_var;
```

global variable with no init value
specified in C source code

```
const uint32_t g_const = 0xDDDEFF00;
```

global constant

```
void show_variables(void)  
{
```

local variable

```
    uint32_t local_var;
```

```
    const uint32_t local_const =
```

local constant

```
        0x7261504F;
```

```
    static uint32_t static_local_var;
```

literal

```
    local_var = 0xD4E5F607;
```

local variable with qualifier static

```
    ...
```

literal

Literal A fixed value in source code **without** an associated symbol name

Constant "variable" that cannot be changed after first assignment

Object File Sections

Code Example

show_variables.c

```
uint32_t g_init_var = 0x4D5E6F70;  
  
uint32_t g_noinit_var;  
  
const uint32_t g_const = 0xDDDEFF00;  
  
void show_variables(void)  
{  
    uint32_t local_var;  
  
    const uint32_t local_const =  
        0x7261504F;  
  
    static uint32_t static_local_var;  
  
    local_var = 0xD4E5F607;  
    ...  
}
```

assembly

```
AREA GlobalVars, DATA, READWRITE  
  
g_init_var DCD 0x4d5e6f70  
g_noinit_var DCD 0x00000000  
static_local_var DCD 0x00000000
```

```
AREA MyConsts, DATA, READONLY  
  
g_const DCD 0xdddeff00
```

```
AREA MyCode, CODE, READONLY,  
  
show_variables ...  
...  
BX lr  
  
lit1 DCD 0x7261504F  
lit2 DCD 0xd4e5f607
```

stored in
registers¹⁾

literal

¹⁾ if possible. E.g. if you use the address operator (&) on a local variable it will be allocated on the stack.

Conclusion

■ Components Cortex-M CPU

- Core Registers: R0-R12, SP, LR, PC
- 32-bit ALU
- Flags (APSR)
- Control Unit with IR (Instruction Register)
- Bus Interface

■ Instruction Types

- Data transfer, data processing, control flow

■ Program Execution

- Fetch – Execute

■ Memory Map

■ Integer Types

- Size depends on architecture → use C99 types for portability
- 'Little Endian' vs. 'Big Endian', alignment

■ Object File Sections

- CODE, DATA, STACK

Data Transfer Instructions

Computer Engineering 1

Motivation

■ How do I "move" data?



Type	Distribution
Data transfers	43%
Control flow	23%
Arithmetic	15%
Compare	13%
Logic	5%
Others	1%

- CPU → CPU register to register
- memory/IO → CPU load from memory/IO
- CPU → memory/IO store to memory/IO

Agenda

- Data Transfers
- Register to Register
- Loading Literals
- Loading Data
- Storing Data
- Loading/Storing Multiple Registers
- The C Perspective
 - Arrays
 - Pointers

For information on individual instructions covered in these slides:
See also Quick Reference Card for Thumb 16-bit Instruction Set

Learning Objectives

At the end of this lesson you will be able

- to enumerate the 4 transfer types of the Cortex-M0
- to read a Cortex-M0 assembly program with data transfer instructions
- to write assembly programs with the major Cortex-M0 data transfer instructions
- to encode and decode data transfer instructions to/from binary machine code
- to apply the EQU assembler directive
- to explain the concept of a 'literal pool' and to apply the 'LDR Rd, =literal' pseudo instruction
- to understand PC relative and indirect addressing (including offsets)
- to explain how arrays are stored in memory and how array elements can be accessed
- to understand how a compiler translates array accesses in a C-program to assembly
- to explain how a C-compiler implements pointers and address operators in assembly language

■ Load/Store Architecture (ARM Cortex-M)

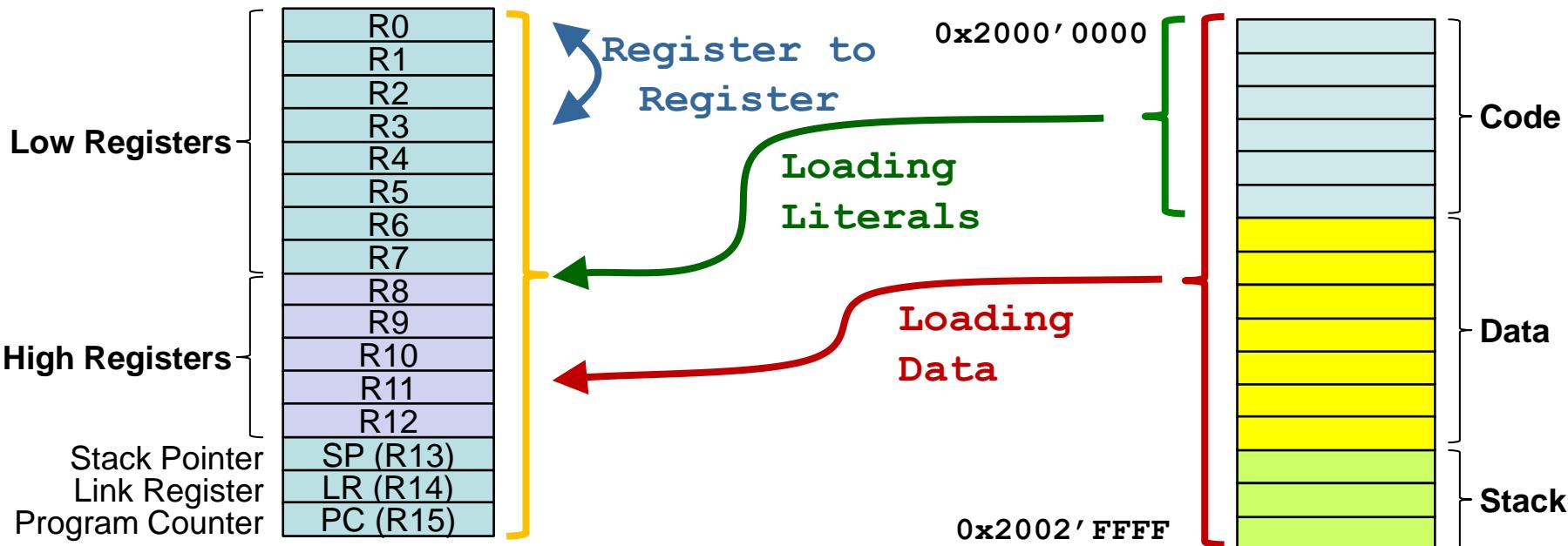
- Memory accessed only with load/store operations
- Usual steps for data processing
 - **Load** operands from memory to register
 - Execute operation → result in register
 - **Store** result from register to memory

■ Register Memory Architecture e.g. Intel x86

- One of the operands can be located in memory
- Result can be directly written to memory

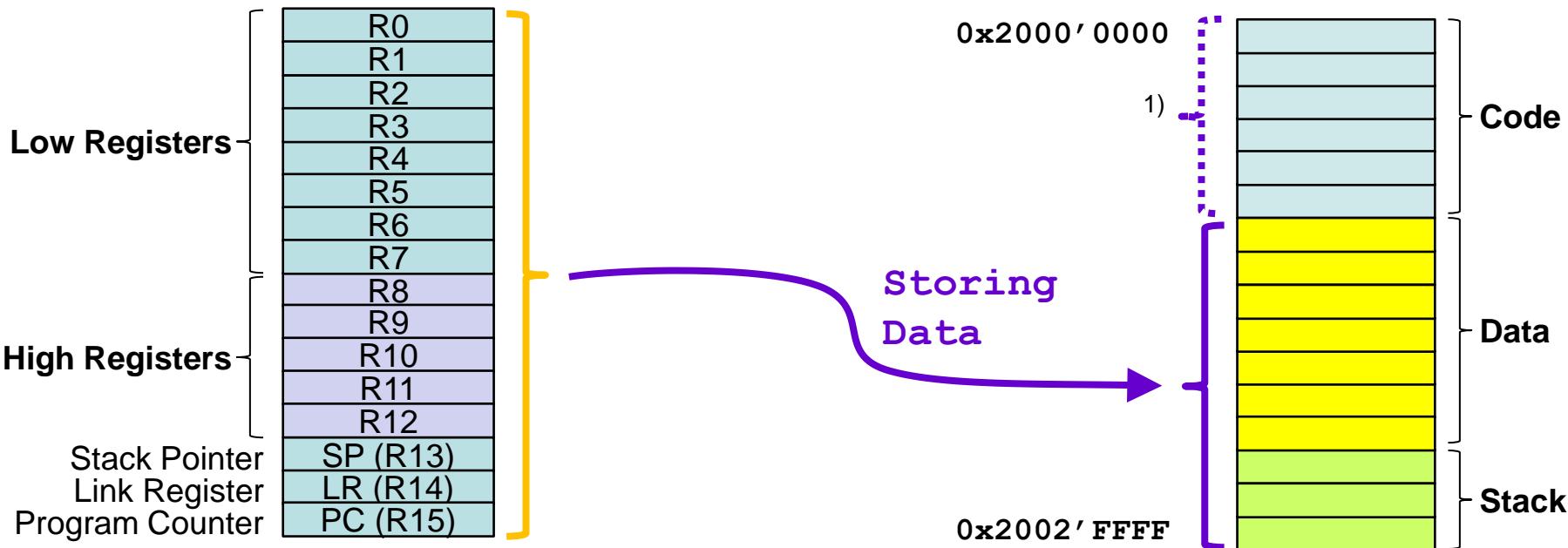
Data Transfers

■ Transfer Types



Data Transfers

■ Transfer Types (continued)

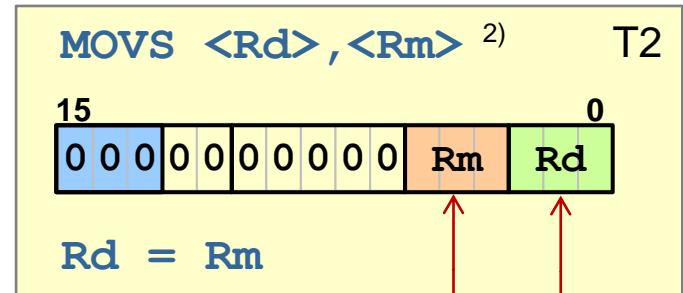
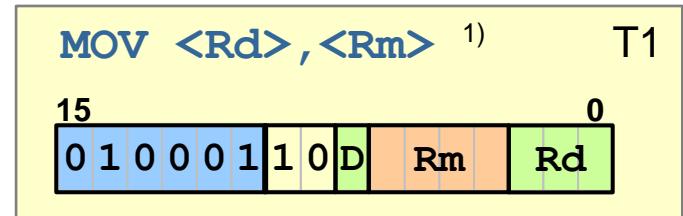
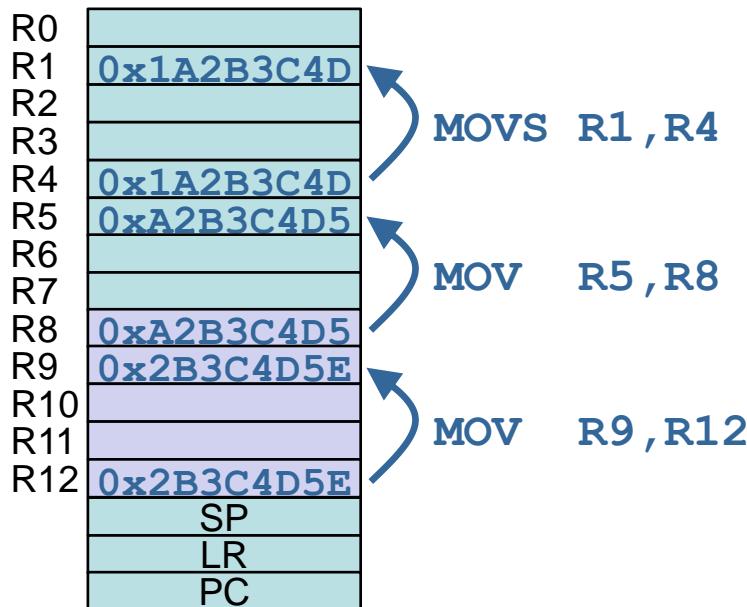


¹⁾ Code region is typically read-only

Register to Register

■ MOV / MOVS (register)

- Copy register value to other register
- MOV → low and high registers
- MOVS → only low registers
S = update of flags ³⁾



3-bit → low registers only

- 1) Instruction group "special data processing"
see table in lecture 2
- 2) Instruction group "shift by immediate, move register"
see table in lecture 2
- 3) 'S' stands for status register

Register to Register

■ Examples MOV / MOVS

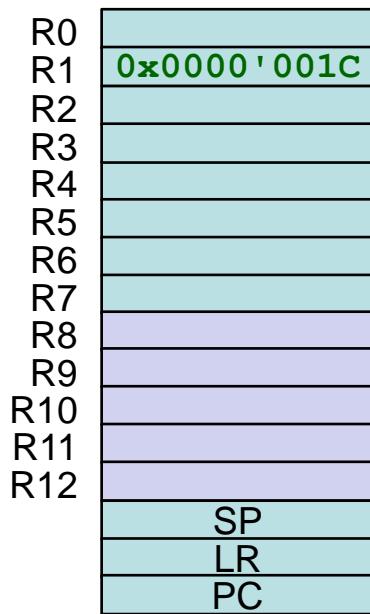
- Copy value of register Rm to Rd
 - **MOV <Rd>, <Rm>** high and low registers allowed
 - **MOVS <Rd>, <Rm>** restricted to low registers

address	opcode	instruction	comment
00000002	4621	MOV R1,R4	; low reg to low reg
00000004	4641	MOV R1,R8	; high reg to low reg
00000006	4688	MOV R8,R1	; low reg to high reg
00000008	46C8	MOV R8,R9	; high reg to high reg
0000000A	0021	MOVS R1,R4	; low reg to low reg
		;MOVS R1,R8	; not possible: high reg
		;MOVS R8,R1	; not possible: high reg
		;MOVS R8,R9	; not possible: high reg

Loading Literals

■ MOVS (immediate data)

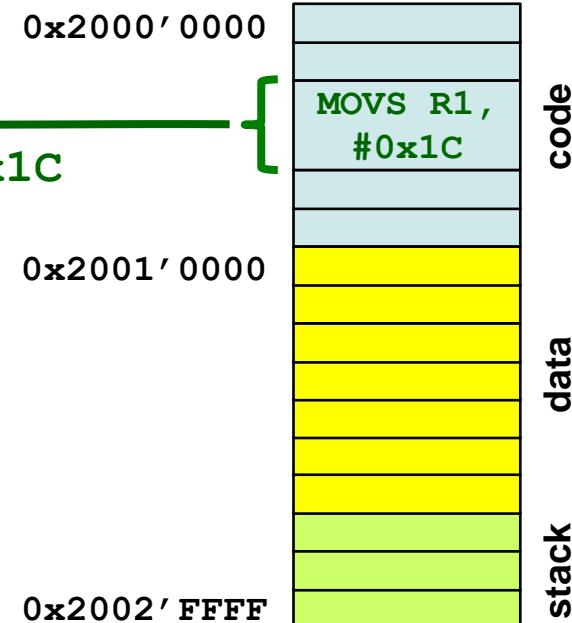
- Copy immediate 8-bit value (literal) to register (only low registers)
- 8-bit literal is part of opcode (imm8)
- Register-bits 31 to 8 set to 0



MOVS <Rd>, #<imm8>

15	0 0 1 0 0	Rd	imm8	0
----	-----------	----	------	---

Rd = <imm8>



Loading Literals

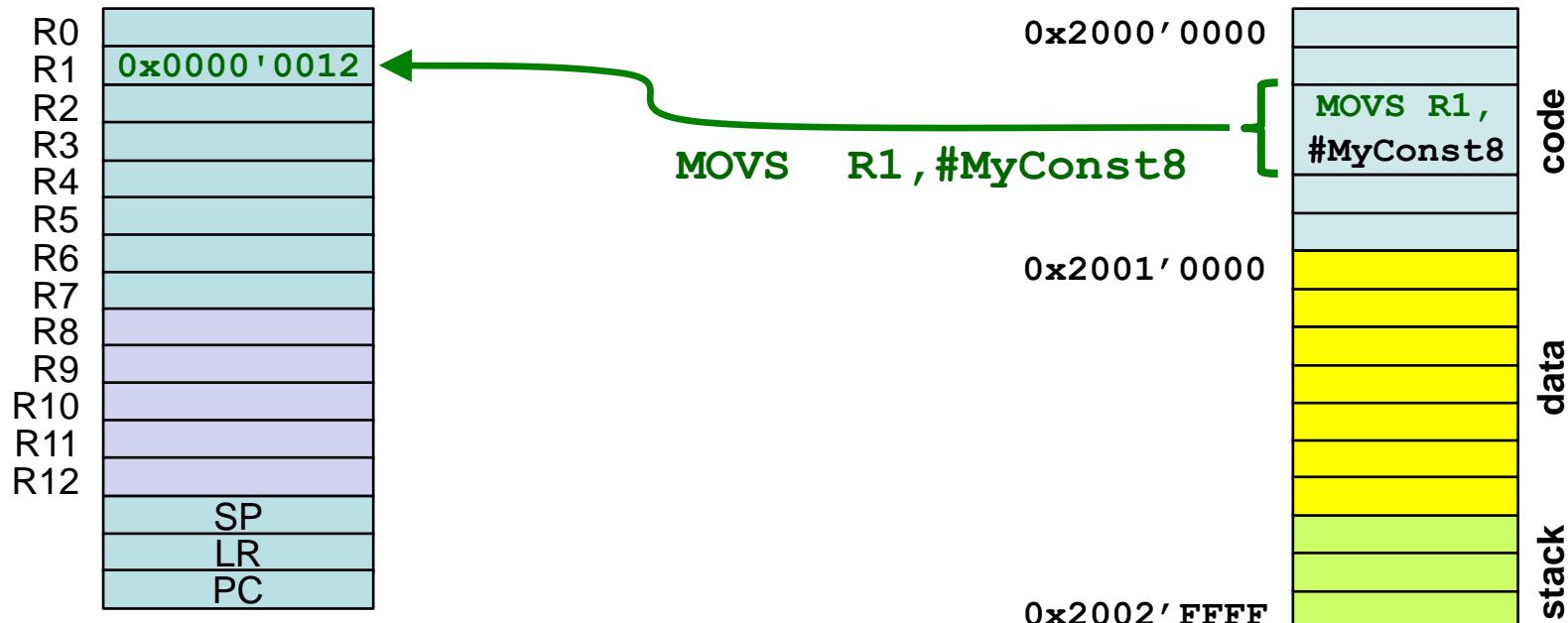
■ EQU - Assembler Directive

- Symbolic definition of literals and constants
- Comparable to `#define` in C

Example:

MOVS with symbolic definition of literal

```
MY_CONST8 EQU 0x12
          MOVS R1, #MY_CONST8
```



Loading Literals

■ Example MOVS (immediate data)

- Immediate 8-bit → 0 to 255d

```
MY_CONST8      EQU      0xCD          ; assembler directive
                ; does not generate opcode

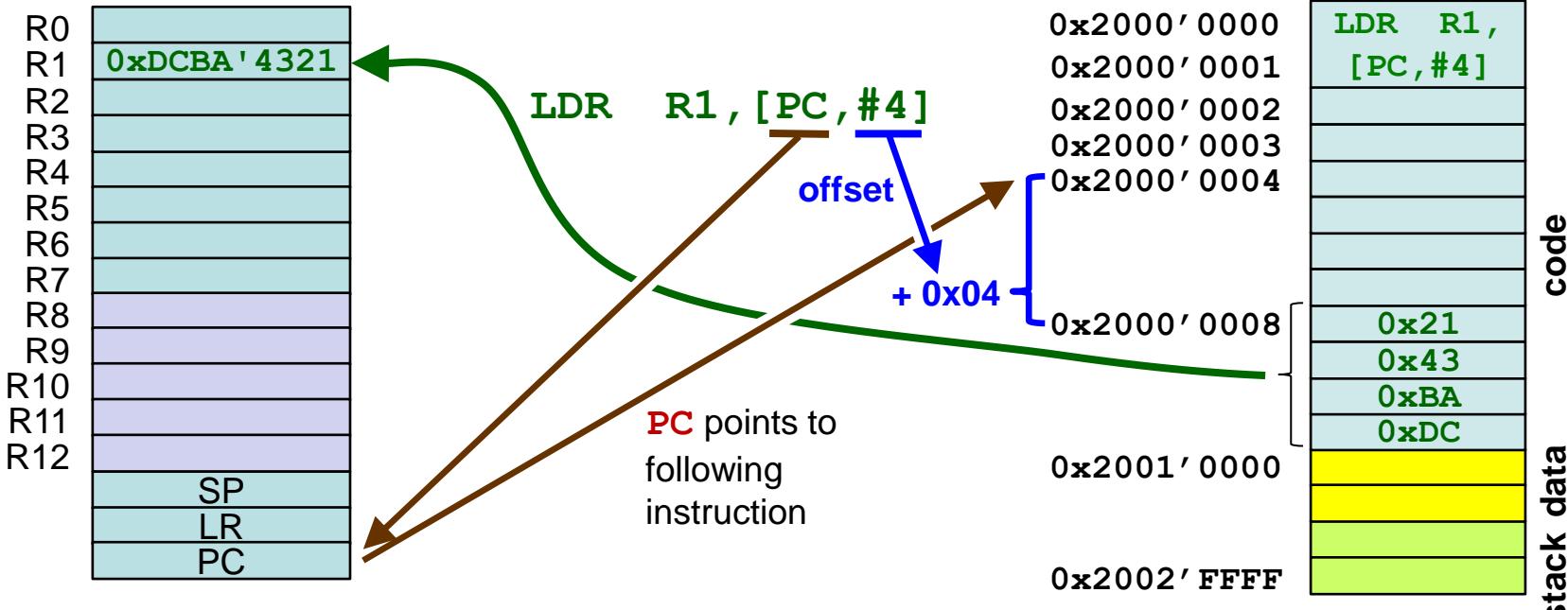
0000000C 21FF  MOVS      R1,#0xFF    ; immediate hex to low reg
0000000E 240C  MOVS      R4,#12      ; immediate dec to low reg
00000010 27CD  MOVS      R7,#MY_CONST8 ; literal with symbolic name

;MOVS      R8,#MY_CONST8   high reg not possible
;MOVS      R1,#0x100      immediate out of range
```

Loading Literals

■ Load - LDR (literal)

- Indirect access relative to PC ¹⁾
- PC offset <imm>
- If PC not word-aligned
 - align on next upper word-address



¹⁾ offsets are positive multiples of 4 in the range of 0 to 0x3FC, i.e. 1020d

Loading Literals

■ Example LDR (literal)

- Offset in bytes vs. imm8 in words → <imm> = imm8:00¹⁾
 - Line1: assembler converts <imm> 0x08 to imm8 = 0x02
- Literals on lines 7 and 8
- PC
 - points to following instruction
 - if not word-aligned → align on next upper word-address

1	00000014	4902	ldr_lit	LDR	R1,[PC,#0x08]	; hex offset
2	00000016	4A03		LDR	R2,[PC,#12]	; dec offset
3	00000018	4B01		LDR	R3,myLit	
4	0000001A	4C01		LDR	R4,myLit	
5	0000001C	4D00		LDR	R5,myLit	
6	0000001E	E003		B	ldr_lit2	
7	00000020	12345678	myLit	DCD	0x12345678	
8	00000024	9ABCDEF0		DCD	0x9ABCDEF0	

Pseudo instruction:
Assembler converts to
LDR R3,[PC,#0x04]

¹⁾ offset = imm8 << 2 i.e. imm8 * 4

Loading Literals

■ Examples LDR (literal)

same code as previous slide

- Which values are being loaded into registers R1 to R5?

1	00000014	4902	ldr_lit	LDR	R1, [PC, #0x08] ; hex offset
2	00000016	4A03		LDR	R2, [PC, #12] ; dec offset
3	00000018	4B01		LDR	R3, myLit
4	0000001A	4C01		LDR	R4, myLit
5	0000001C	4D00		LDR	R5, myLit
6	0000001E	E003		B	ldr_lit2
7	00000020	12345678	myLit	DCD	0x12345678
8	00000024	9ABCDEF0		DCD	0x9ABCDEF0

Use a symbol: Let assembler do the calculation

Branch prevents interpretation of literals as instructions

Line 1

PC = 0x00000016 --> word_aligned 0x00000018

Load literal from 0x00000018 + 0x08 = 0x00000020

R1 = 0x12345678

Line 2

PC = 0x00000018 --> word_aligned 0x00000018

Load literal from 0x00000018 + 12d = 0x00000024

R2 = 0x9ABCDEF0

¹⁾ Word alignment of PC causes the same offset in lines 3 and 4, although they are accessing the same literal

Loading Literals

■ Pseudo Instruction

LDR Rd, =literal

- Assembler
 - creates 'literal pool' at convenient code location
 - allocates and initializes memory in 'literal pool' → DCD
 - uses LDR (literal) instruction and calculates offset ¹⁾

assembly code as written
by human programmer

LDR R1, =0x20000012

code generated
by assembler (tool)

LDR R1, [PC, #68]

...

DCD

...

DCD

0x20000012

DCD

...

'Literal Pool' at end of code block

'literal pool' → group of literals

¹⁾ on M3/M4 the assembler may use a MOV instruction with an immediate value instead

Loading Literals

■ Pseudo Instruction Examples

- Warning: difference between lines 7 and 8

```
1 CONST_A EQU 0x000000AA
2 CONST_B EQU 0xBBCCDDEE
3
4 0000003C 4F03 LDR R7 ,=0x12
5 0000003E 4F04 LDR R7 ,=CONST_A
6 00000040 4F04 LDR R7 ,=CONST_B
7 00000042 4D01 LDR R5 ,mylita
8 00000044 4D04 LDR R5 ,=mylita
9 00000046 E02F B wherever
10
11 00000048 FF001122 mylita DCD 0xFF001122
12
13 0000004C 00000012
14 00000050 000000AA CONST_A and CONST_B
15 00000054 BBCCDDEE
16 00000058 00000000 space where the address of mylita
                           will be stored after linking
```

Literal Pool

load 0xFF001122 from address 0x00000048

load address of mylita from address 0x00000058

CONST_A and CONST_B

space where the address of mylita will be stored after linking

Loading Literals

■ Pseudo Instruction Examples

LDR R5,mylita

→ LDR R5, [PC,#...]

The **value** 0xFF001122 at label **mylita** is loaded into R5

LDR R5,=0x20003000

→ LDR R5, [PC,#...]

Space is allocated in literal pool. The **value** 0x20003000 is stored into this location and loaded into R5 from there.

LDR R5,=CONST_A

→ LDR R5, [PC,#...]

Space is allocated in literal pool. The **value** 0x000000AA, defined through **EQU**, is stored into this location and loaded into R5 from there.

LDR R5,=mylita

→ LDR R5, [PC,#...]

Space is allocated in literal pool. The address of **mylita** is stored into this location and loaded into R5 from there.

```
CONST_A EQU 0x000000AA
AREA example, CODE, ...
Start

mylita B wherever
mylita DCD 0xFF001122
```

Literal pool will be created here

= sign tells the compiler to allocate and initialize space in literal pool. → Programmer does not have to type DCD lines.

Loading Literals

■ C Example

C-Code

```
static uint32_t g;

void lit_example(void) {
    uint32_t a;
    uint32_t b;
    const uint32_t c = 0x04;
    uint32_t *p;

    a = 0x05;
    b = 0xABCD1)E12;
    p = &g;
    ...
}
```

Loading Literals

■ C Example

C-Code

```
static uint32_t g;

void lit_example(void) {
    uint32_t a;
    uint32_t b;
    const uint32_t c = 0x04;
    uint32_t *p;

    a = 0x05;
    b = 0xABCDE12; 1)
    p = &g;
    ...
}
```

Compiler assigns
variables to registers

a → R0	p → R2
b → R1	c → R3

Assembly

```
4904 → LDR R1,[PC,#0x10]
4a04 → LDR R2,[PC,#0x10]
```

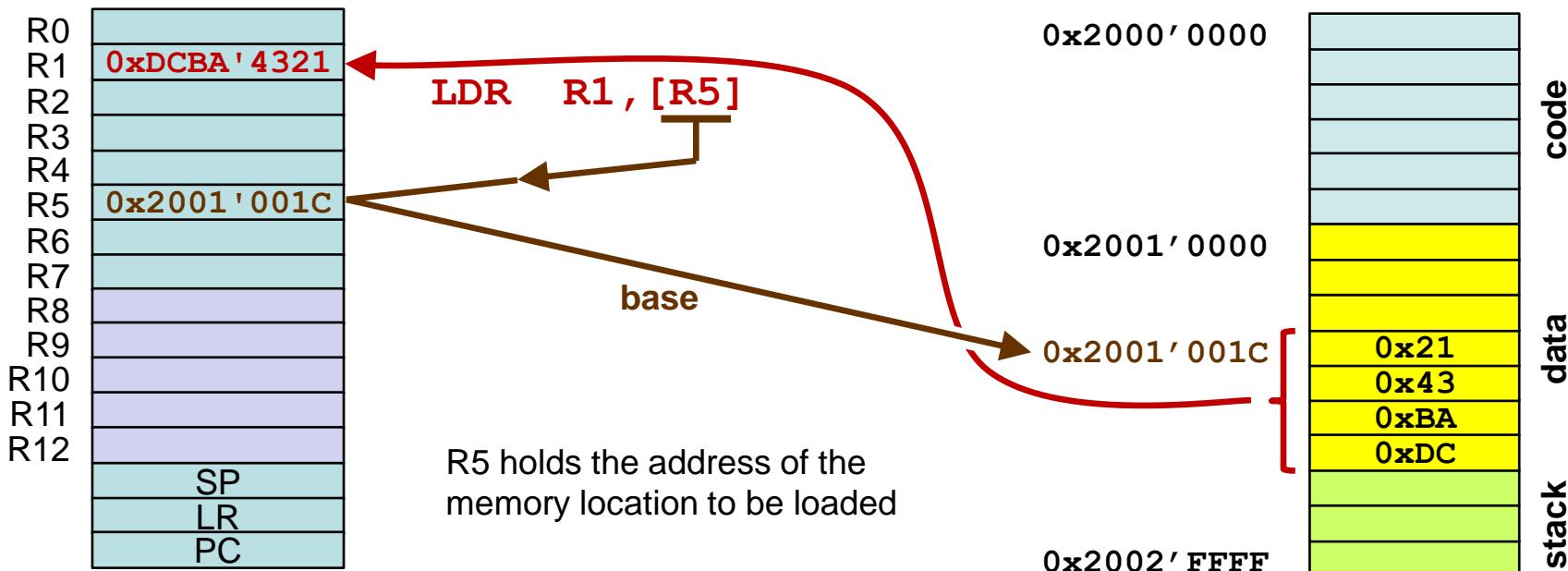
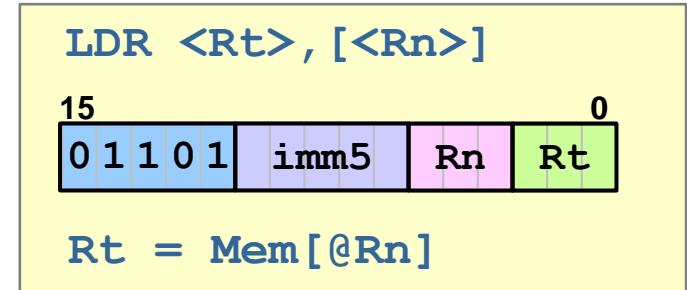
```
AREA MyCode, CODE, ...
...
000002 2304      MOVS  R3,#4
000004 2005      MOVS  R0,#5
000006 4904      LDR   R1,lit1
000008 4a04      LDR   R2,adr_g 2)
...
000018          lit1  DCD   0xabcdef12
00001C          adr_g DCD   g 3)
AREA MyData, DATA, ...
g           DCD   0x00000000
```

- 1) Compiler translates `b = 0xABCDE12`; to `LDR R1,lit1`
Assembler then translates `LDR R1,lit1` to `LDR R1,[PC,#0x10]`
- 2) loads the address of g
- 3) DCD stores the address of g (not the content of g)
Compiler does not use the pseudo-instruction `LDR R2,=g`, which would mean the same.

Loading Data

■ LDR (immediate offset) – imm5 = 0¹⁾

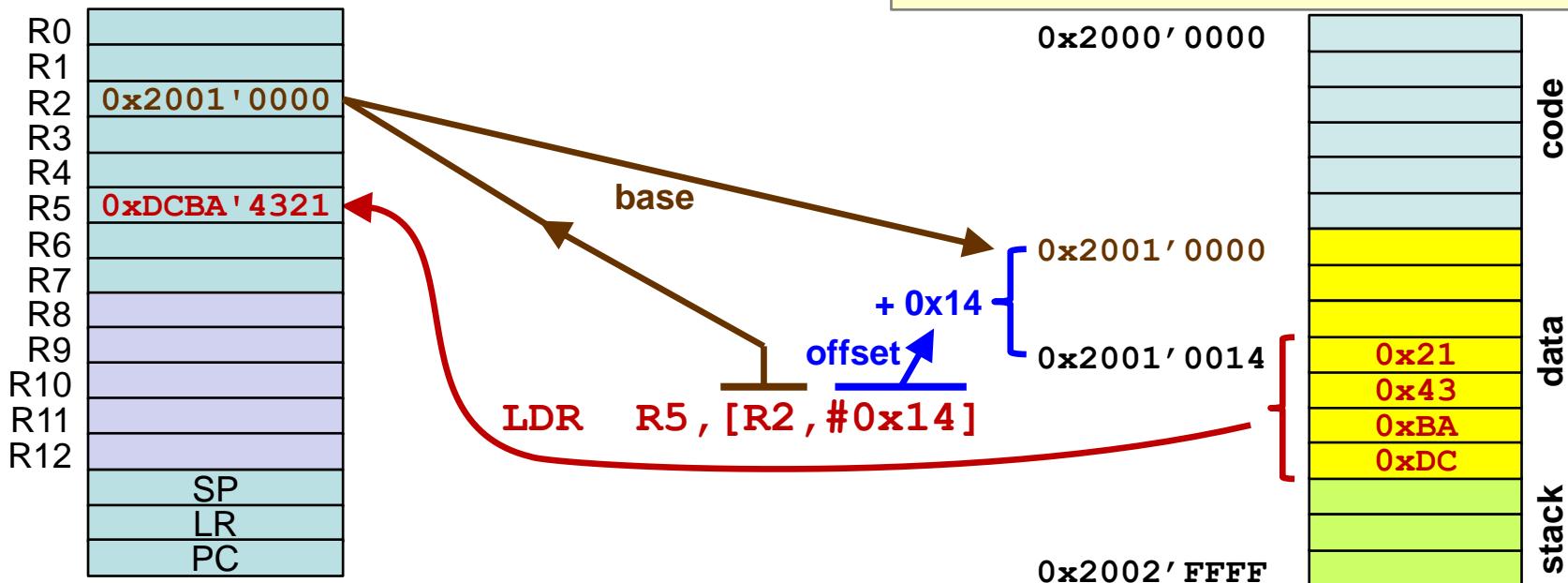
- Indirect addressing → [...]
 - Rn → memory address
- Only low registers



Loading Data

■ LDR (immediate offset) – general

- Indirect addressing
 - Immediate offset <imm>
 - Offset range 0 - 124d (0x7C)¹⁾
 - Only low registers

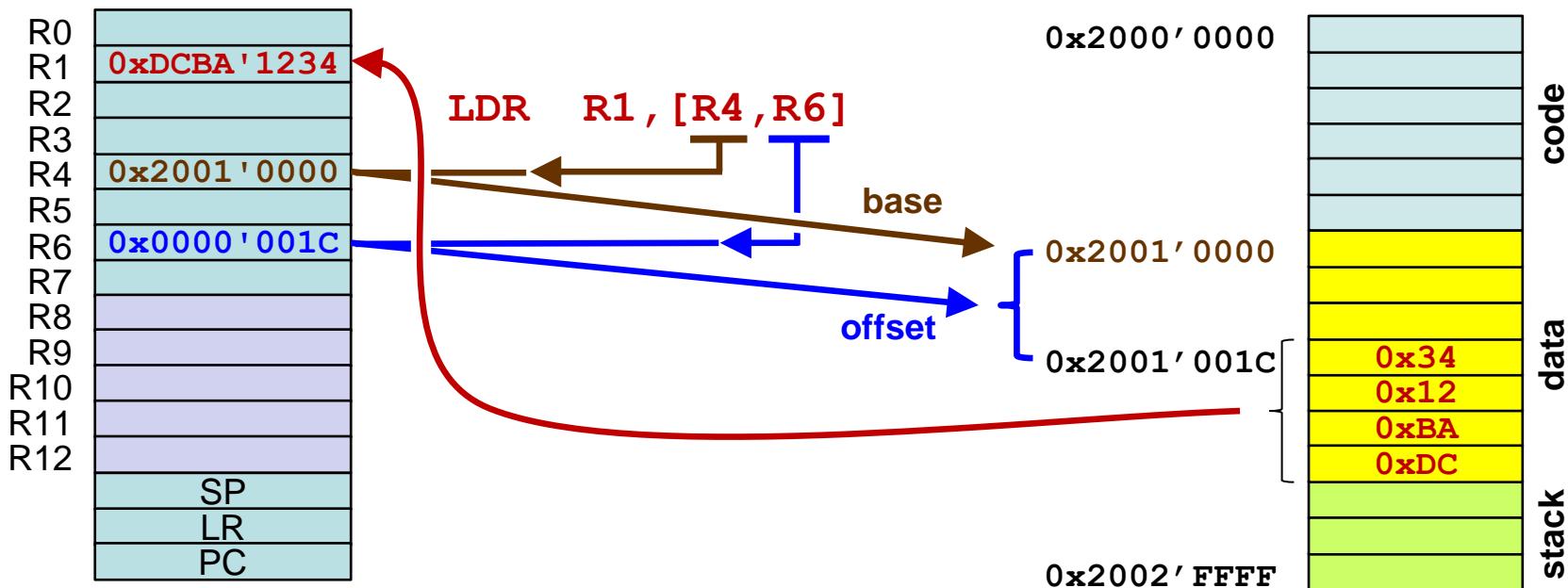
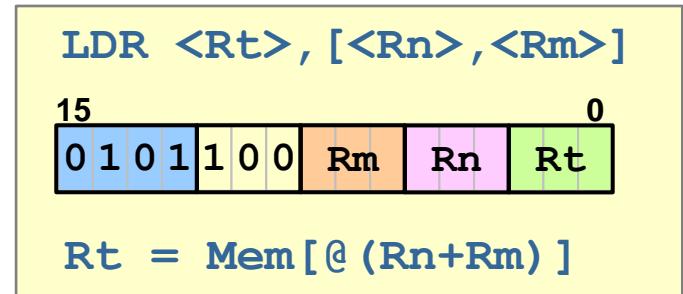


1) positive values, multiples of 4 only

Loading Data

■ LDR (register offset)

- Indirect addressing with offset register
 - offset = unsigned
- Only low registers



Loading Data

■ Examples

- Content of R4, R5, R6 after execution?

```
        AREA    my_data, DATA, READWRITE
00000000 11223344 my_array      DCD    0x11223344
00000004 55667788           DCD    0x55667788
00000008 99AABBCC           DCD    0x99AABBCC
```

```
        AREA    myCode, CODE, READONLY
        . . .
; load base and offset registers
0000007C 4906    LDR    R1,=my_array ; load address of array
0000007E 4B07    LDR    R3,=0x08

; indirect addressing
00000080 680C    LDR    R4,[R1]      ; base R1
00000082 684D    LDR    R5,[R1,#0x04] ; base R1, immediate offset
00000084 58CE    LDR    R6,[R1,R3]   ; base R1, offset R3
```

Not content of my_array,
but address of my_array

Loading Data

■ LDRB (register/immediate offset)

- Load Register Byte
 - Register bits 31 to 8 set to zero

■ LDRH (register/immediate offset)

- Load Register Half-word
 - Register bits 31 to 16 set to zero

LDRB <Rt>, [<Rn>, #<imm>]

15	0	
0 1 1 1 imm5	Rn	Rt

```
<imm> = imm5  
Rt = Byte[@(Rn+<imm>)]
```

LDRB <Rt>, [<Rn>, <Rm>]

15 0

0	1	0	1	1	1	0	Rm	Rn	Rt
---	---	---	---	---	---	---	----	----	----

Rt = Byte[@(Rn+Rm)]

LDRH <Rt>, [<Rn>, #<imm>]

15	0
1 0 0 0 1 imm5 Rn Rt	

<imm> = imm5:0

Rt = Hw[@(Rn+imm5)]

LDRH <Rt>, [<Rn>, <Rm>]

$$R_t = H_w[\alpha(R_n + R_m)]$$

Loading Data

■ LDRSB

- Load Register Signed Byte
- Sign extend
 - Register bits 31 to 8 set or reset depending on bit 7

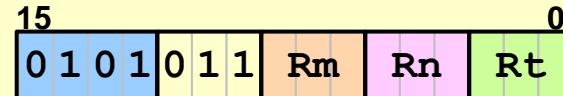
■ LDRSH

- Load Register Signed Half-word
- Sign extend
 - Register bits 31 to 16 set or reset depending on bit 15

■ Sign Extension

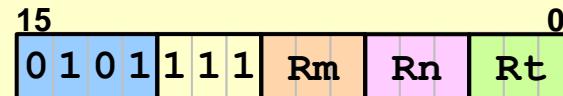
- See slides on casting

LDRSB <Rt>, [<Rn>, <Rm>]



Rt = sign_extend(Byte[@(Rn+Rm)])

LDRSH <Rt>, [<Rn>, <Rm>]

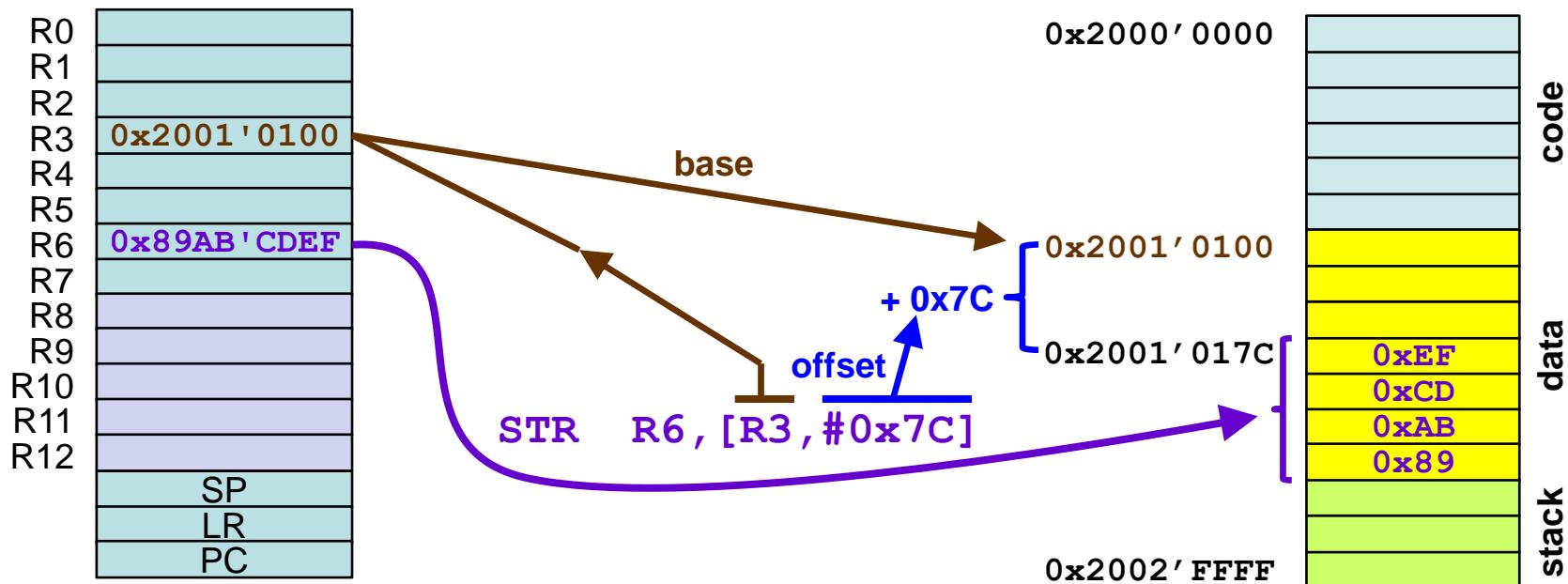


Rt = sign_extend(Hw[@(Rn+Rm)])

Storing Data

■ STR (immediate offset)

- Indirect addressing with immediate offset
 - Offset range 0 - 124d (0x7C)¹⁾
 - Only low registers

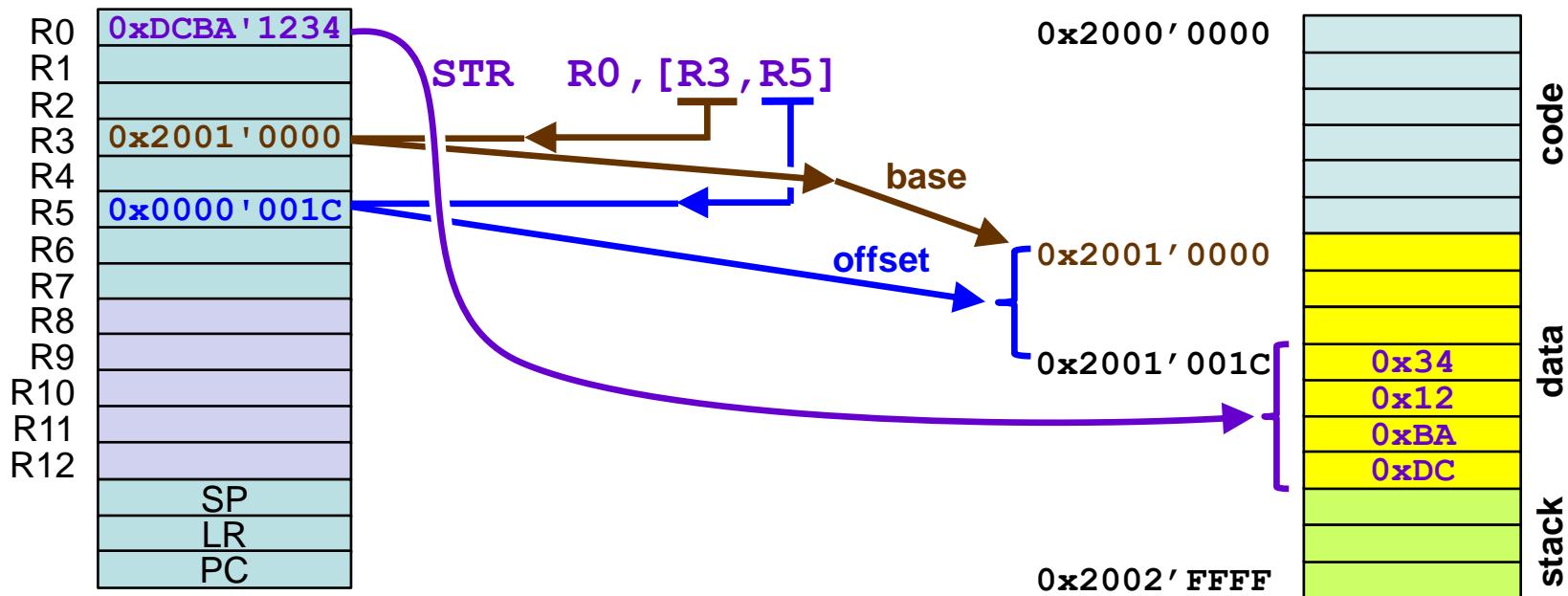
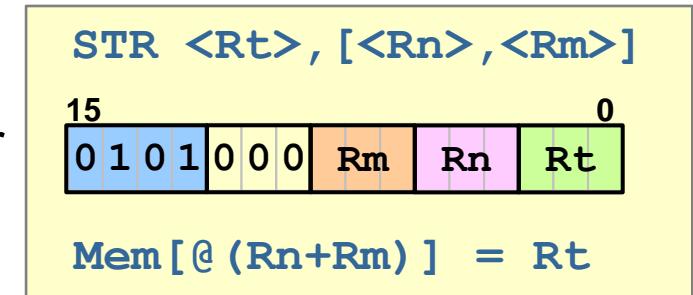


1) positive values, multiples of 4 only

Storing Data

■ STR (register offset)

- Indirect addressing with offset register
 - Offset register → index
- Only low registers



Storing Data

```
CONST_C EQU 0xCCCCCCCC  
CONST_D EQU 0xDDDDDDDD  
CONST_E EQU 0xEEEEEEEE
```

■ Example

	AREA	progData2, DATA, READWRITE
data_array	SPACE	256
000000A4 4904	LDR	R1 ,=CONST_C
000000A6 4A05	LDR	R2 ,=CONST_D
000000A8 4B05	LDR	R3 ,=CONST_E
000000AA 4F06	LDR	R7 ,=data_array
000000AC 4E06	LDR	R6 ,=0x08
000000AE 6039	STR	R1 , [R7]
000000B0 607A	STR	R2 , [R7 ,#0x04]
000000B2 51BB	STR	R3 , [R7 ,R6]
000000B4 E00A	B	ldm_ex
000000B6 0000	ALIGN	4
000000B8 CCCCCCCC		
000000BC DDDDDDDD		
000000C0 EEEEEEEE		
000000C4 00000000		
000000C8 00000008		

store CONST_C in memory at address of data_array

store CONST_D in memory at address of data_array + 0x04

store CONST_E in memory at address of data_array + 0x08

storage space in literal pool for address of data_array

Storing Data

■ STRB (immediate/register offset)

- Store Register Byte
- Low 8 bits of register stored

STRB <Rt>, [<Rn>, #<imm>]

15	0	1	1	1	0	imm5	Rn	Rt	0
----	---	---	---	---	---	------	----	----	---

<imm> = imm5

Byte[@(Rn+<imm>)] = Rt(7:0)

STRB <Rt>, [<Rn>, <Rm>]

15	0	1	0	1	0	1	0	Rm	Rn	Rt	0
----	---	---	---	---	---	---	---	----	----	----	---

Byte[@(Rn+Rm)] = Rt(7:0)

STRH <Rt>, [<Rn>, #<imm>]

15	0	1	0	0	0	0	imm5	Rn	Rt	0
----	---	---	---	---	---	---	------	----	----	---

<imm> = imm5:0

Hw[@(Rn+<imm>)] = Rt(15:0)

STRH <Rt>, [<Rn>, <Rm>]

15	0	1	0	1	0	0	1	Rm	Rn	Rt	0
----	---	---	---	---	---	---	---	----	----	----	---

Hw[@(Rn+Rm)] = Rt(15:0)

Summary Data Transfer

<code>MOVS <Rd>, <Rm></code>	Register to register	
<code>MOVS <Rd>, #<imm8></code>	Loading literals	8-bit literal
<code>LDR <Rt>, [PC, #<imm>]</code>		32-bit literal, PC-relative
<code>LDR <Rt>, [<Rn>, #<imm>]</code> also <code>LDRB</code> and <code>LDRH</code>	Loading data	Register indirect with immediate offset
<code>LDR <Rt>, [<Rn>, <Rm>]</code> also <code>LDRB</code> , <code>LDRH</code> , <code>LDRSB</code> and <code>LDRSH</code>		Register indirect with register offset
<code>STR <Rt>, [<Rn>, #<imm>]</code> also <code>STRB</code> and <code>STRH</code>	Storing data	Register indirect with immediate offset
<code>STR <Rt>, [<Rn>, <Rm>]</code> also <code>STRB</code> and <code>STRH</code>		Register indirect with register offset

Loading/Storing Multiple Registers

■ LDM 1)

For information only

- Load Multiple Registers
- Rn: Base address

1) LDMIA (Load Multiple Increment After) and LDFM (Load Multiple from Full Descending stacks) are aliases for LDM

LDM <Rn>! ,<registers>

LDM <Rn> ,<registers>



Registers in reg_list
are loaded from memory
starting at address in Rn

000000CC	4A06	LDR	R2 ,=ldm_const
000000CE	CAE6	LDM	R2 ,{R1 ,R2 ,R5-R7}
000000D0	E00C	B	ldm_ex2
000000D2	0000		
000000D4	AAAAAAA	ldm_const	
		DCD	0xAAAAAAA
000000D8	BBBBBBBB	DCD	0xBBB BBBB
000000DC	CCCCCCCC	DCD	0xCC CCCCCCCC
000000E0	DDDDDDDD	DCD	0xDD DDDDDDDD
000000E4	EEEEEEEE	DCD	0xEE EEEEEE
000000E8	00000000		

R1 = 0xAAAA'AAAA
R2 = 0xBBB BBBB'BBBB
R5 = 0xCCCC'CCCC
R6 = 0xDDDD'DDDD
R7 = 0xEEEE'EEEE

Loading/Storing Multiple Registers

■ STM 1)

For information only

- Store Multiple Registers
- Rn: Base address

1) STMIA (Store Multiple Increment After) and STMEA (Store Empty Ascending) are aliases for LDM

STM <Rn>! ,<registers>



Registers in `reg_list`
are stored to memory
starting at address in `Rn`

<code>data_array</code>	<code>AREA progData2, DATA, READWRITE</code>
	<code>SPACE 256</code>

<code>000000CE 4C01</code>	<code>LDR R4, =data_array</code>
<code>000000D0 C4E6</code>	<code>STM R4!, {R1,R2,R5-R7}</code>
<code>000000D2 E001</code>	<code>B stm_cont</code>
<code>000000D4 00000000</code>	

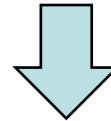
`R1 → @data_array`
`R2 → @data_array + 0x04`
`R5 → @data_array + 0x08`
`R6 → @data_array + 0x0C`
`R7 → @data_array + 0x10`

The C Perspective: Arrays

■ Array of Bytes

C-code

```
static uint8_t byte_array[] =  
    {0xAA, 0xBB, 0xCC, 0xDD,  
     0xEE, 0xFF};
```



assembly

```
byte_array  
    DCB    0xAA, 0xBB, 0xCC, 0xDD  
    DCB    0xEE, 0xFF
```

address	index
0x2001'0000	0
0x2001'0001	1
0x2001'0002	2
0x2001'0003	3
0x2001'0004	4
0x2001'0005	5

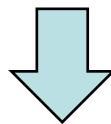
assuming that `byte_array` starts at 0x2001'0000

The C Perspective: Arrays

■ Array of Half-words

C-code

```
static uint16_t halfword_array[] =  
    {0x0011, 0x2233,  
     0x4455, 0x6677,  
     0x8899, 0xAABB};
```



assembly

```
halfword_array  
    DCW    0x0011,0x2233  
    DCW    0x4455,0x6677  
    DCW    0x8899,0xAABB
```

address	index
0x2001'0000	0
0x2001'0002	1
0x2001'0004	2
0x2001'0006	3
0x2001'0008	4
0x2001'000A	5
0x2001'000C	
0x2001'000E	

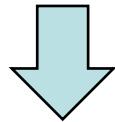
assuming that `halfword_array` starts at 0x2001'0000

The C Perspective: Arrays

■ Array of Words

C-code

```
static uint32_t word_array[] =  
    {0xFFEEDDCC,  
     0xBA9988,  
     0x77665544,  
     0x33221100};
```



assembly

word_array	DCD	0xFFEEDDCC
	DCD	0xBA9988
	DCD	0x77665544
	DCD	0x33221100

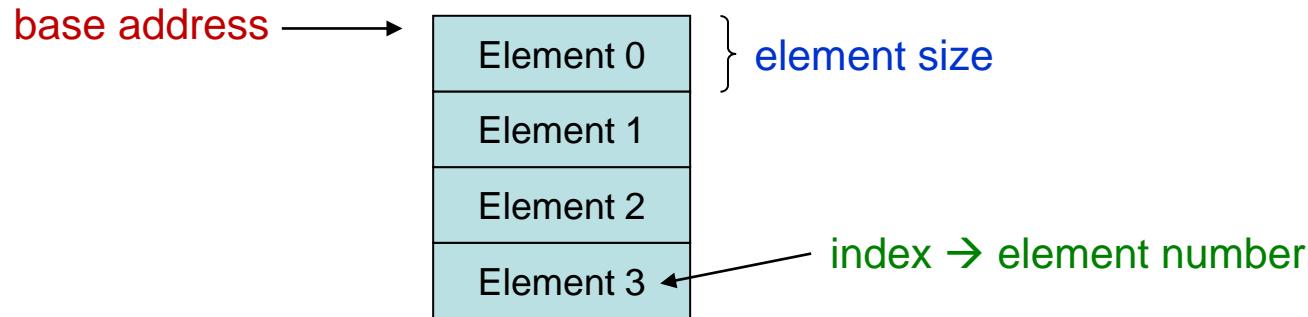
address	index
0x2001'0000	0
0x2001'0004	
0x2001'0008	
0x2001'000C	
0x2001'000F	3

assuming that `word_array` starts at 0x2001'0000

The C Perspective: Arrays

■ Accessing array elements

- element address = **base address** + **element size** • **index**



- element sizes in bytes
 - word 4
 - half-word 2
 - byte 1

The C Perspective: Arrays

■ Example: array access

C-Code

```
static uint8_t byte_array[] =  
    {0xAA, 0xBB, 0xCC, 0xDD,  
     0xEE, 0xFF};  
  
void access_byte_array(void)  
{  
    ...  
    byte_array[3] = 0x12;  
    ...  
}
```

Assembly

```
AREA MyData, DATA, READWRITE  
byte_array DCB 0xaa,0xbb  
            DCB 0xcc,0xdd  
            DCB 0xee,0xff
```

```
AREA MyCode, CODE, READONLY  
access_byte_array
```

```
    ...  
    1 MOVS      r0,#0x12  
    2 LDR       r1,adr_b  
    3 STRB      r0,[r1,#3]  
    ...
```

```
adr_b      DCD      byte_array
```

- 1 Load value to be stored into R0
- 2 Load base address from label below ¹⁾
- 3 Store R0 to base address plus offset

The C Perspective: Arrays

■ Array access (word)

C-Code

```
static uint32_t word_array[] =  
    {0xFFEEDDCC,  
     0xBBAA9988,  
     0x77665544,  
     0x33221100};  
  
void access_word_array(void)  
{  
    ...  
    word_array[3] = 0xAABBCCDD;  
    ...  
}
```

Assembly

```
AREA MyData, DATA, READWRITE  
word_array    DCD    0xffeeddcc  
                DCD    0xbbaa9988  
                DCD    0x77665544  
                DCD    0x33221100  
  
AREA MyCode, CODE, READONLY  
access_word_array  
    ...  
    1 LDR    r0, lit_1  
    2 LDR    r1, adr_w  
    3 STR    r0, [r1, #0xc]  
    ...  
lit_1        DCD    0xaabbccdd  
adr_w        DCD    word_array
```

- 1 Load literal from label → R0
- 2 Load base address from label → R1
- 3 Store R0 to base address plus offset
 $offset (0xC) = \text{element size (4)} * \text{index (3)}$

The C Perspective: Pointer

■ Pointer and Address Operator

C-Code

```
void pointer_example(void)
{
    static uint32_t x;
    static uint32_t *xp;

    xp = &x;
    *xp = 0x0C;
}
```

- 1 Load address of x → R0
- 2 Load address of xp → R1
- 3 Store R0 (*i.e. address of x*) in xp variable
(indirect memory access through R1)
- 4 Load immediate value 0x0C → R0
- 5 Load content of xp → R1
i.e. address of x is now in R1
- 6 Store R0 at address given by R1

Assembly

```
AREA MyData, DATA, READWRITE
x          DCD      0x00000000
xp         DCD      0x00000000
```

```
AREA MyCode, CODE, READONLY
pointer_example
...
```

- 1 LDR r0,adr_x
- 2 LDR r1,adr_xp
- 3 STR r0,[r1,#0]
- 4 MOVS r0,#0xc
- 5 LDR r1,[r1,#0]
- 6 STR r0,[r1,#0]

```
...
adr_x      DCD      x
adr_xp     DCD      xp
```

The C Perspective: Pointer

■ Memory Mapped I/O

- Write to LEDs on CT Board

C-Code

```
void main(void)
{
    volatile uint32_t *p;

    p = (volatile uint32_t *)
        0x60000100;
    *p = 0x1A2B3C4D;
}
```

Assembly

```
AREA MyCode, CODE, READONLY

                    LDR      r0,led_adr
                    LDR      r1,led_val
                    STR      r1,[r0, #0]

led_val     DCD 0x1A2B3C4D
led_adr     DCD 0x60000100
```

The local variable p is kept in register r0, not in memory

Conclusion

■ Data transfers

- Register to Register **MOV/MOVS (register)**
- Loading Literals **MOVS (immediate data)**
- Loading Data **LDR (PC-relative/literal-pool)**
- Storing Data **LDR (immediate/register offset)**
-
-
-
-

■ Addressing Modes

- PC Relative **[PC,#0x12]**
- Indirect Addressing **[R1], [R2,#0x12], [R5,R6]**

■ Arrays

- Element address = **base address + element size • index**
- Accessed with data transfer instructions

■ Volatile

- Use for accessing memory mapped items

Loading Data

```
R4 = 0x11223344
R5 = 0x55667788
R6 = 0x99AABBCC
```

■ Examples

- Content of R4, R5, R6 after execution?

```

      AREA   my_data, DATA, READWRITE
00000000 11223344 my_array      DCD    0x11223344
00000004 55667788              DCD    0x55667788
00000008 99AABBCC              DCD    0x99AABBCC

```

```

      AREA   myCode, CODE, READONLY
      . . .
; load base and offset registers
0000007C 4906    LDR    R1,=my_array ; load address of array
0000007E 4B07    LDR    R3,=0x08

; indirect addressing
00000080 680C    LDR    R4,[R1]      ; base R1
00000082 684D    LDR    R5,[R1,#0x04] ; base R1, immediate offset
00000084 58CE    LDR    R6,[R1,R3]   ; base R1, offset R3

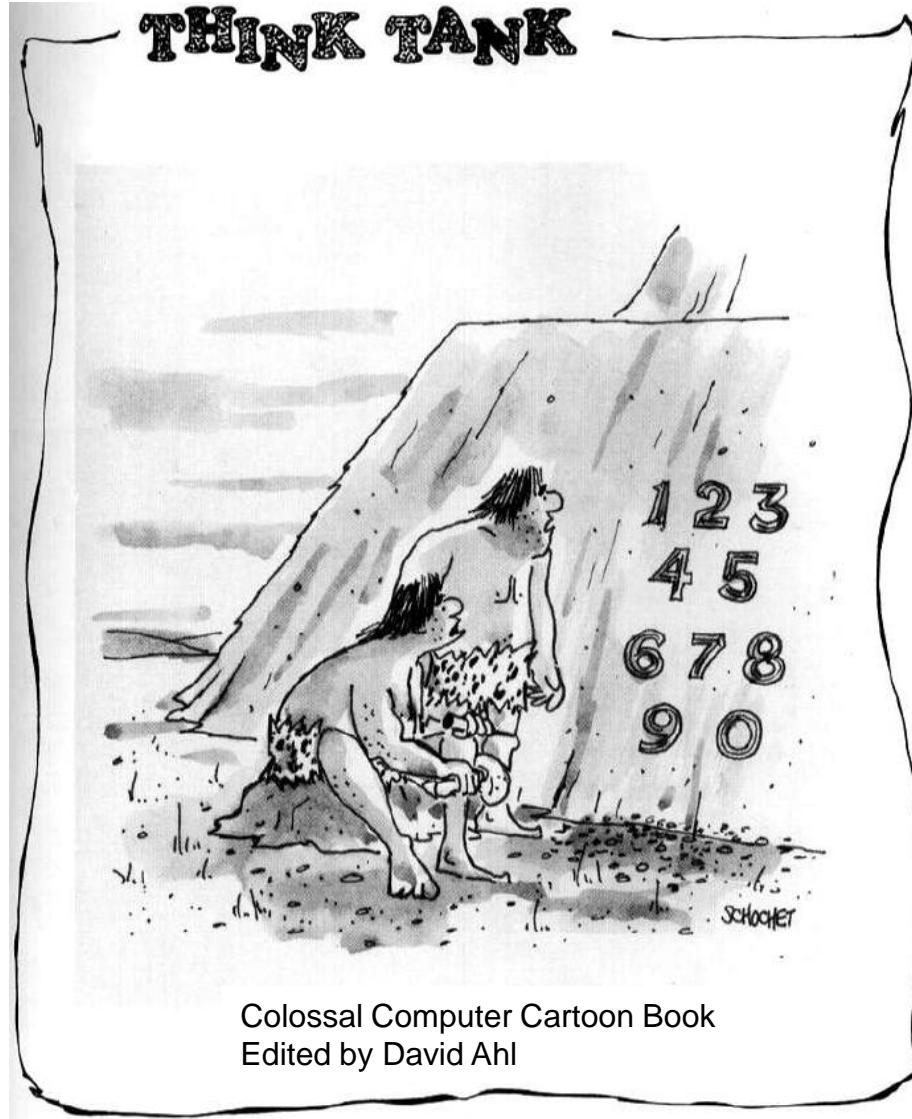
```

Not content of my_array,
but address of my_array

Arithmetic Operations

Computer Engineering 1

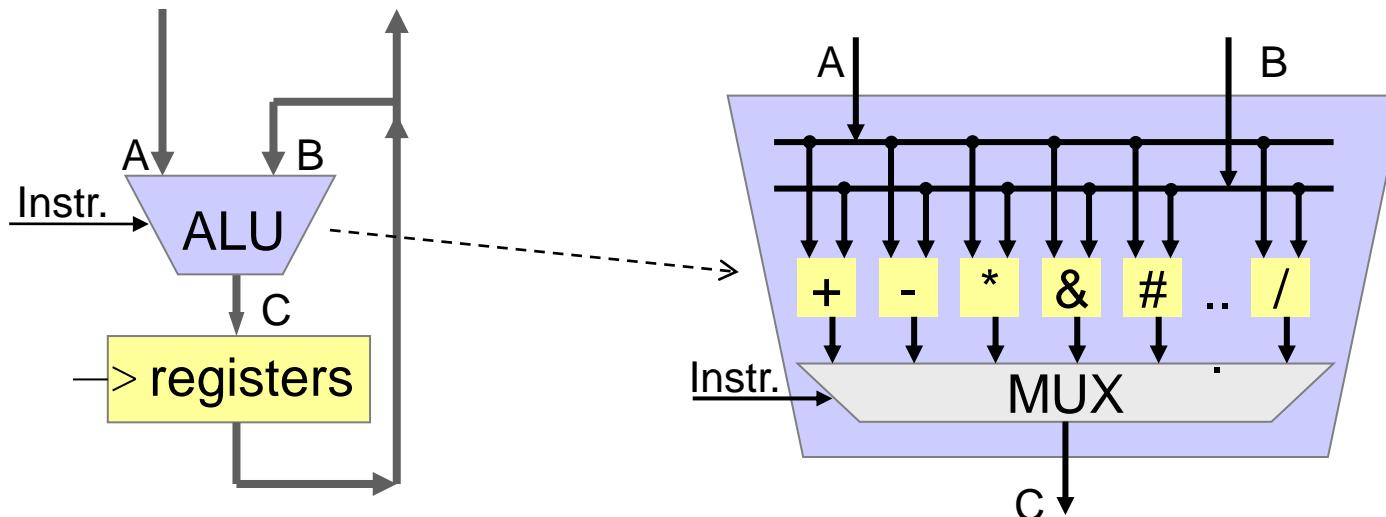
Motivation



"I call them numbers, you can add them, subtract them, multiply them, divide them ... find their square root ..."

Motivation

- Instructions to process data in the ALU
 - **arithmetic** Addition, Subtraction, Multiplication, Division
 - **logic** NOT, AND, OR, XOR
 - **shift** Shift left/right. Fill with 0 or MSB
 - **rotate** Cyclic shift left/right: What drops out enters on the other side



■ How often is the for loop executed?

- For `#define INCREMENT 1`
- For `#define INCREMENT 10`

```
...
int main(void) {
    uint8_t uc;
    uint32_t count = 0;

    for (uc = 0; uc < 255; uc += INCREMENT) {
        printf("hello again %d \n",uc);
        count++;
    }
    printf("Loop executed %d times\n",count);
}
```

Agenda

- **Cortex-M0**
 - Data flow
 - Flags
 - Overview of arithmetic instructions
- **Add Instructions Cortex-M0**
- **Negative Numbers**
- **Addition**
- **Subtraction**
- **Subtract Instructions Cortex-M0**
- **Multi-Word Arithmetic**
- **Multiplication**

Learning Objectives

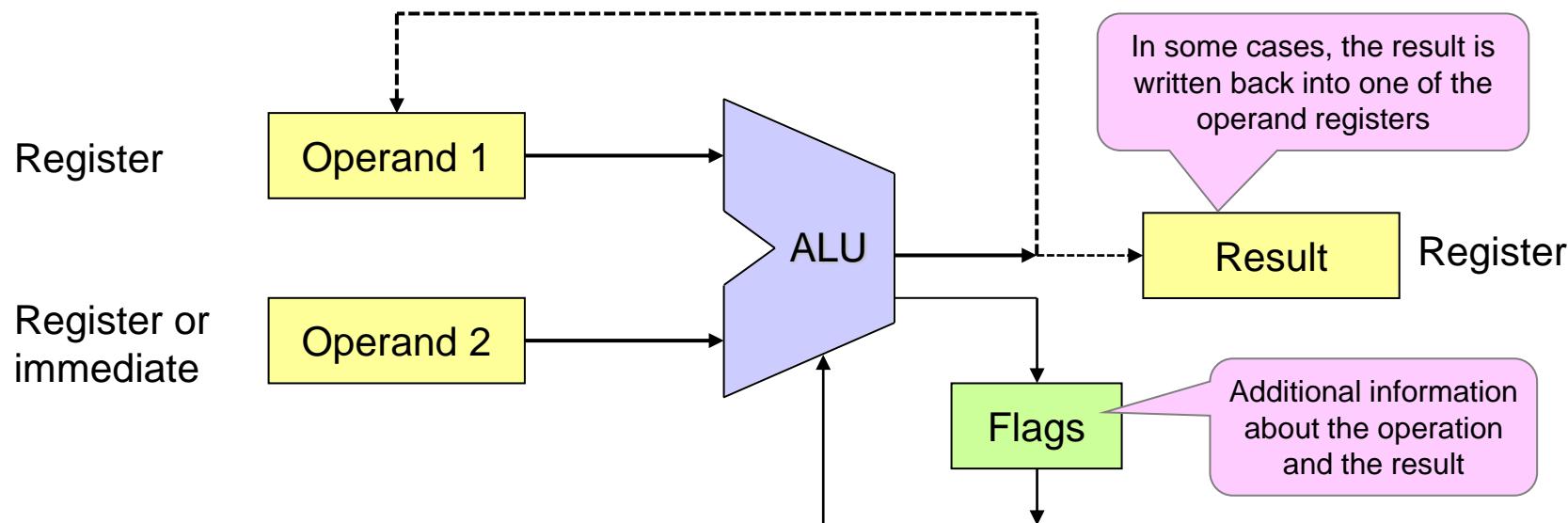
At the end of this lesson you will be able

- to enumerate and apply the Cortex-M0 arithmetic instructions
- to interpret Cortex-M0 assembly programs with arithmetic instructions
- to enumerate and explain the meaning of the Cortex-M0 Flags (N, Z, C, V)
- to carry out additions and subtractions of signed and unsigned integers and to explain the operations with the circle of numbers
- to calculate and interpret carry/borrow and overflow/underflow
- to determine (with the help of documents) the state of Cortex-M0 Flags (N, Z, C, V) after an arithmetic instruction
- to describe how addition and subtraction are done in hardware (in the ALU)
- to program integer calculations with operands that exceed the number of bits available in the ALU
- to explain how numbers in two's complements representation are multiplied

Data flow (Cortex-M0)

■ Operands and results stored in registers

- Exception: Immediate operations with data in OP-code
- Load/store architecture



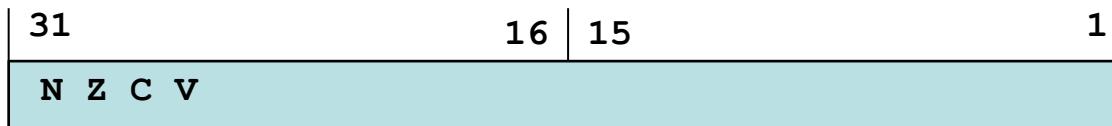
Examples

ADDS	R0 , R0 , R1	; R0 = R0 + R1	result back in R0
ADDS	R0 , #0x34	; R0 = R0 + 0x34	
SUBS	R3 , R4 , #5	; R3 = R4 - 0x05	result in other reg

Flags

■ Cortex-M0

- APSR: Application Program Status Register



Flag	Meaning	Action	Operands
Negative	MSB = 1	N = 1	signed
Zero	Result = 0	Z = 1	signed , unsigned
Carry	Carry	C = 1	unsigned
Overflow	Overflow	V = 1	signed

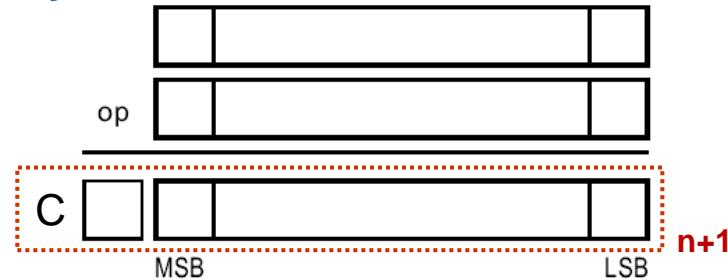
- Processor does not know whether the user is working with **unsigned (C)** or with **signed (V)** numbers
 - Processor therefore always calculates C and V!

Flags

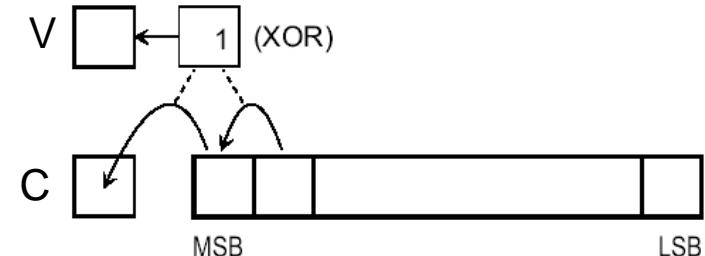
■ Cortex-M0

- Instructions ending with ',S' allow modification of flags
- Examples: **MOVS**, **ADDS**, **SUBS**, ...

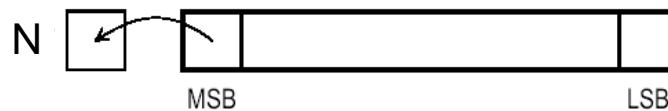
Carry



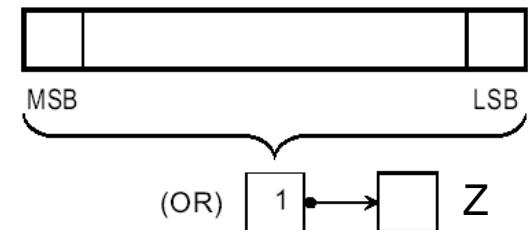
Overflow



Negative



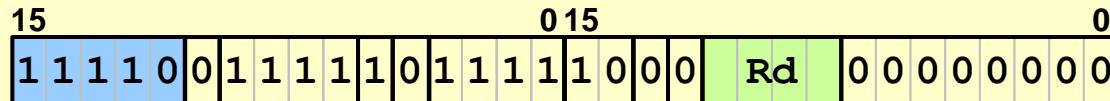
Zero



■ Instructions to access APSR

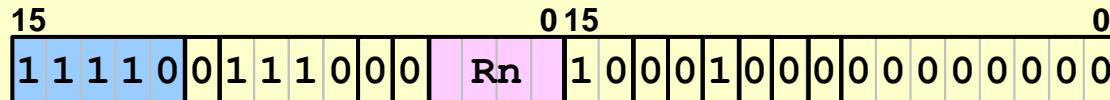
- **MRS** Move to register from special register (APSR)
- **MSR** Move to special register (APSR) from register
- 32-bit opcode

MRS <Rd>, APSR



Rd = APSR

MSR APSR, <Rn>



APSR = Rn

Arithmetic Instructions

■ Overview Cortex-M0

Mnemonic	Instruction	Function
ADD / ADDS	Addition	$A + B$
ADCS	Addition with carry	$A + B + c$
ADR	Address to Register	$PC + A$
SUB / SUBS	Subtraction	$A - B$
SBCS	Subtraction with carry (borrow)	$A - B - \text{NOT}(c)$ ¹⁾
RSBS	Reverse Subtract (negative)	$-1 \bullet A$
MULS	Multiplication	$A \bullet B$

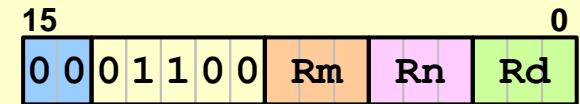
¹⁾ borrow = NOT(c)

Add Instructions Cortex-M0

■ ADDS (register)

- Update of flags
- Result and two operands
- Only low register

ADDS <Rd>, <Rn>, <Rm>



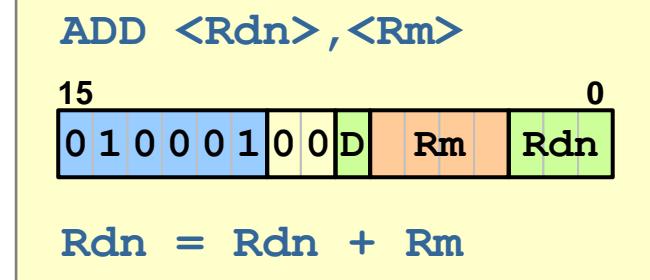
$$Rd = Rn + Rm$$

00000002	18D1	ADDS	R1, R2, R3	
00000004	1889	ADDS	R1, R1, R2	
00000006	1889	ADDS	R1, R2	; the same (dest = R1)
00000008		;ADDS	R9, R2	; not possible (high reg)
00000008		;ADDS	R1, R10	; not possible (high reg)

Add Instructions Cortex-M0

■ ADD (register)

- No update of Flags
- High or low register
- <Rdn> → result and operand
 - I.e. same register for operand and result!



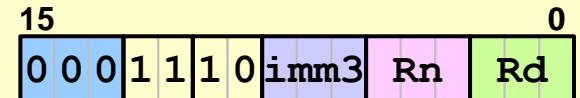
00000008	4411	ADD	R1, R1, R2	; low regs
0000000A	44D1	ADD	R9, R9, R10	; high regs
0000000C	44D1	ADD	R9, R10	; the same (dest = R9)
0000000E	4411	ADD	R1, R1, R2	
0000000E		; ADD	R1, R2, R3	; not possible

Add Instructions Cortex-M0

■ ADDS (immediate) – T1

- Update of flags
- Two different¹⁾ low registers and immediate value 0 - 7

ADDS <Rd>, <Rn>, #<imm3>



Rd = Rn + <imm3>

00000010 1D63	ADDS	R3, R4, #5	
00000012	;ADDS	R3, R4, #8	; out of range immediate
00000012	;ADDS	R10, R11, #5	; not possible (high reg)

¹⁾ If the same register is used for Rd and Rn, the assembler will choose the encoding T2 on the next slide

Add Instructions Cortex-M0

■ ADDS (immediate) – T2

- Update of flags
- Low register with immediate value 0 - 255d
- <Rdn> → Result and operand in same register

ADDS <Rdn>, #<imm8>



Rdn = Rdn + <imm8>

```
00000012 33F0    ADDS    R3,R3,#240
00000014 33F0    ADDS    R3,#240      ; the same (dest = R3)
00000016          ;ADDS    R8,R8,#240 ; not possible (high reg)
00000016          ;ADDS    R3,#260      ; out of range immediate
```

Add Instructions Cortex-M0

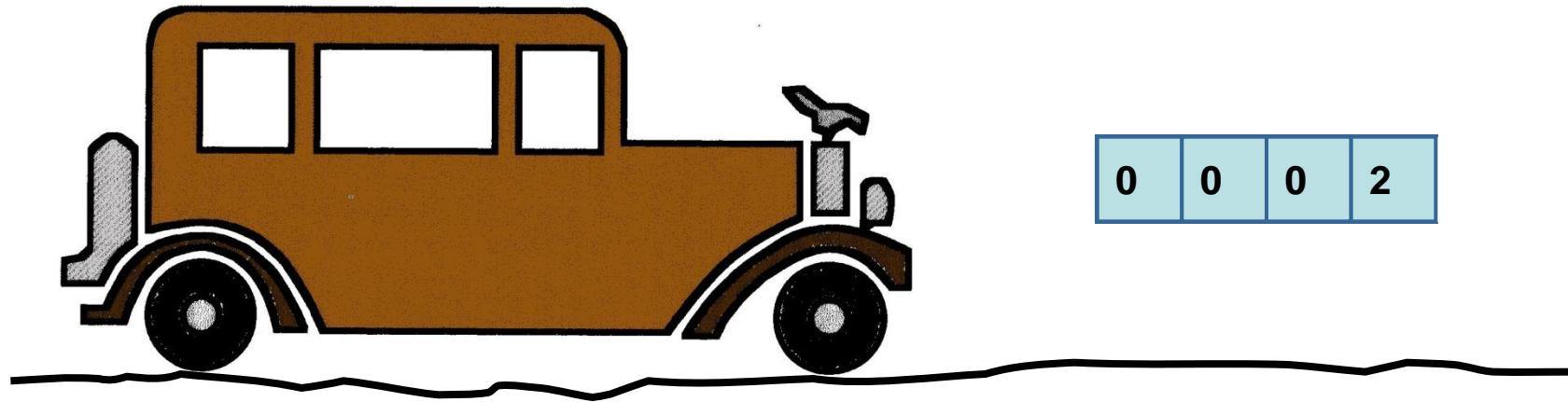
■ ADD / ADDS (Summary)

- ADD Flags not changed
- ADDS Flags changed according to Operation Result

Instr	Rd	Rn	Rm	imm	Restrictions
ADD	R0-R15	R0-R15	R0-R15	-	Rd and Rn must specify the same register. Rn and Rm must not both specify the PC (R15)
ADDS	R0-R7	R0-R7	-	0 - 7	-
ADDS	R0-R7	R0-R7	-	0 - 255	Rd and Rn must specify the same register
ADDS	R0-R7	R0-R7	R0-R7	-	-

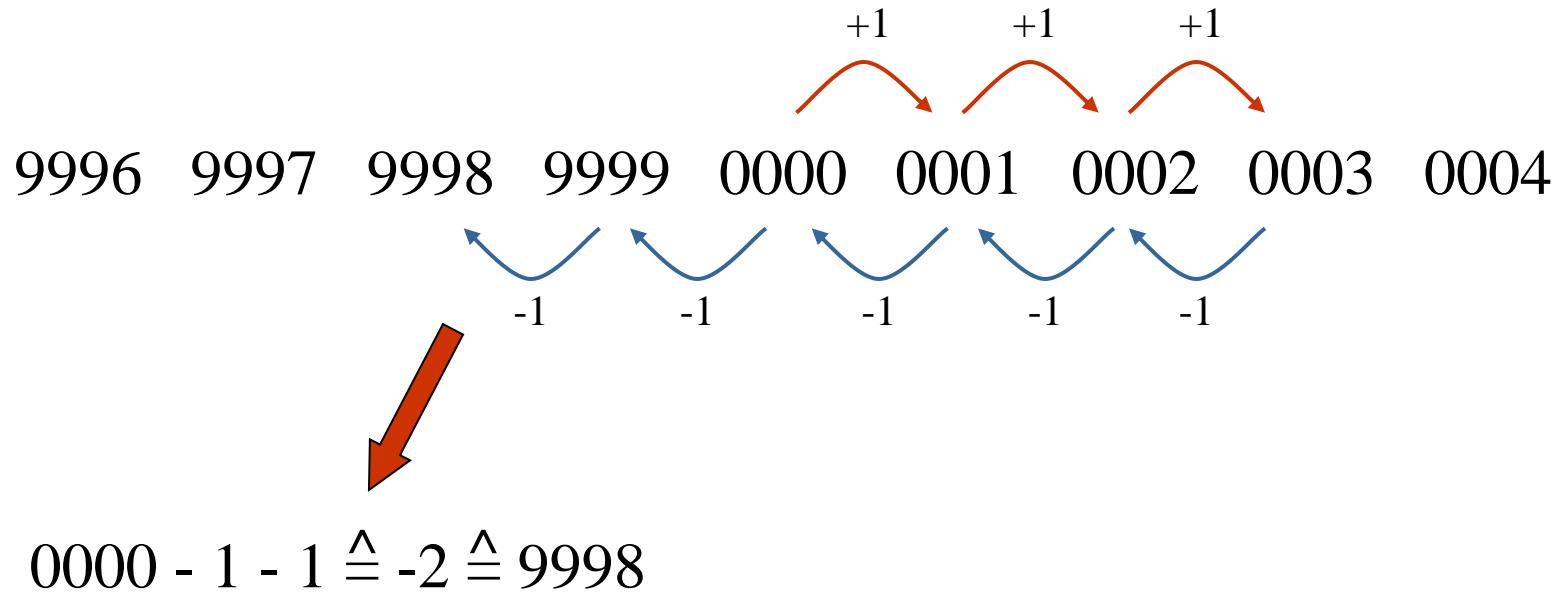
Negative Numbers

- Lord Ardry's new Rolls Royce



Negative Numbers

■ What happened?



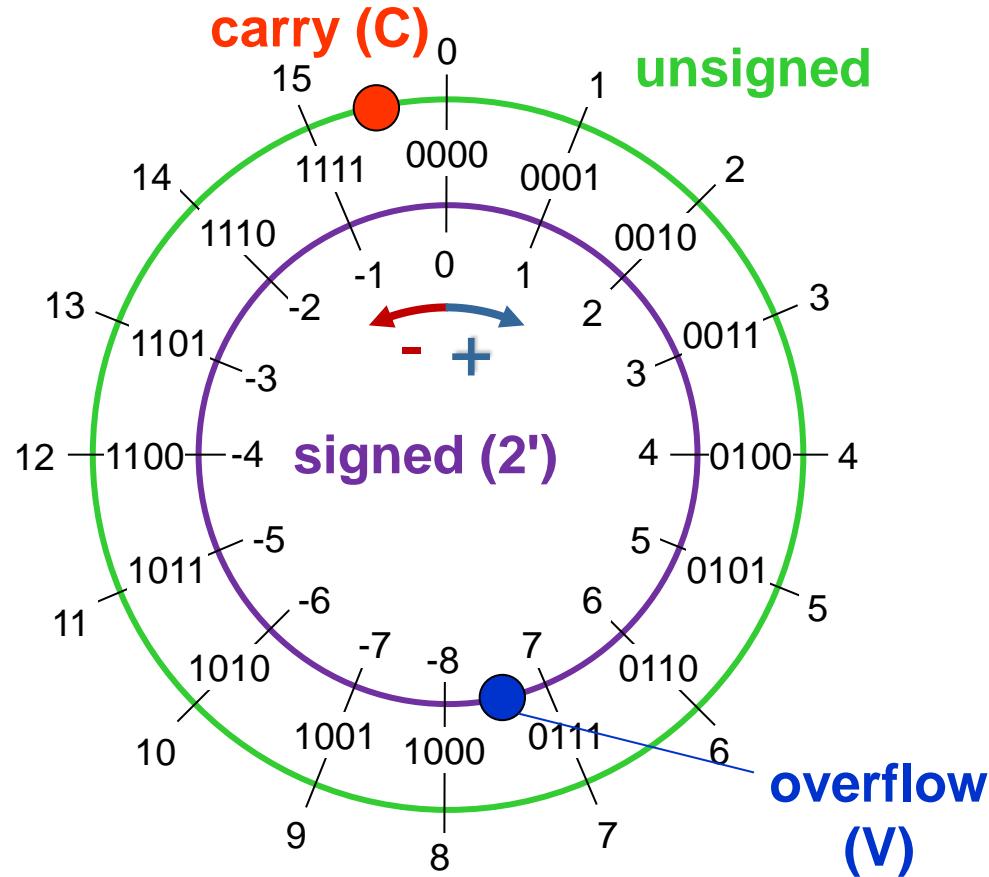
possible because

- number of digits is finite
- respectively given by the word size

Negative Numbers

signed $-b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

unsigned $+b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

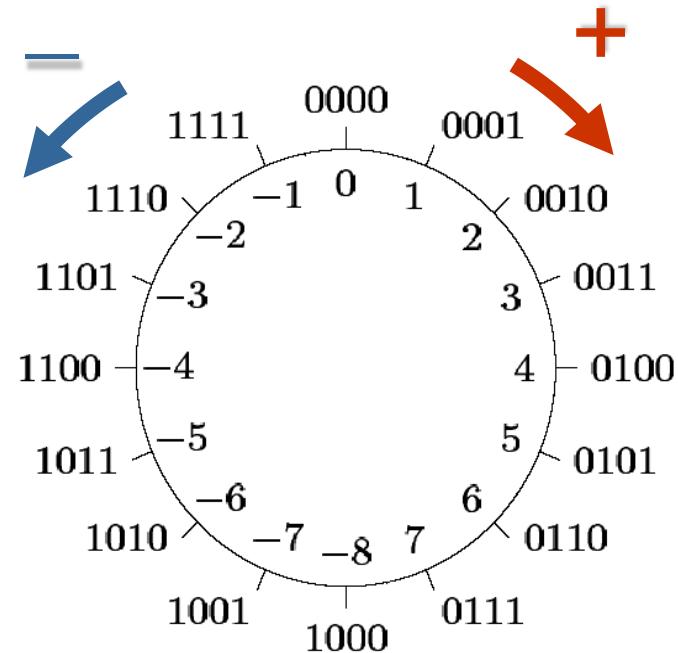


binary	unsigned	signed 2'-Compl.
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Negative Numbers

■ Numbers with finite number of digits

- can be represented on a circle
- Addition
 - Clockwise
- Subtraction
 - Counter-clockwise



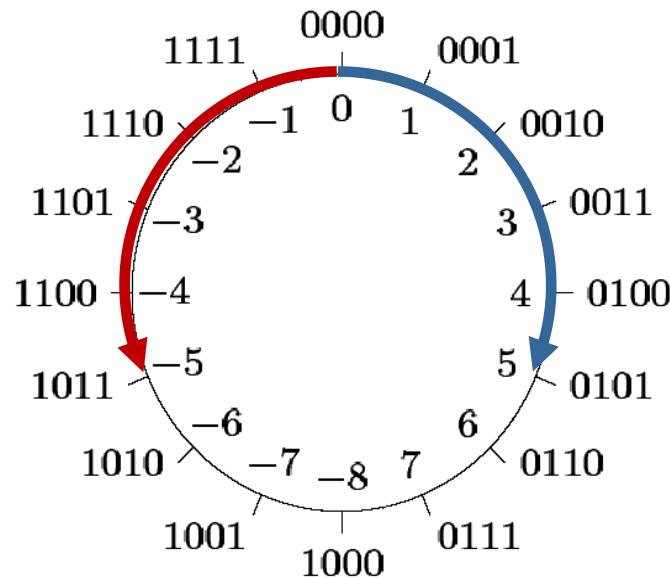
Negative Numbers

■ Negation of a number

- Idea: $-a = 0 - a$
- Starting at 0000 enter the absolute value of a counter-clockwise
- Example: $5d = 0101 \rightarrow -5d = 1011$

written calculation

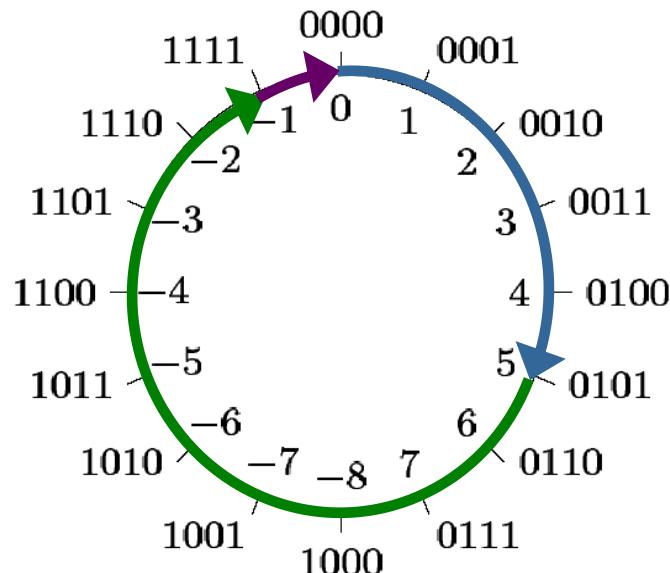
$$\begin{array}{r} 0000 \\ - 0101 \\ \hline 1\ 1\ 1\ 1 \\ \hline 1011 \end{array}$$



Negative Numbers

■ 2' Complement

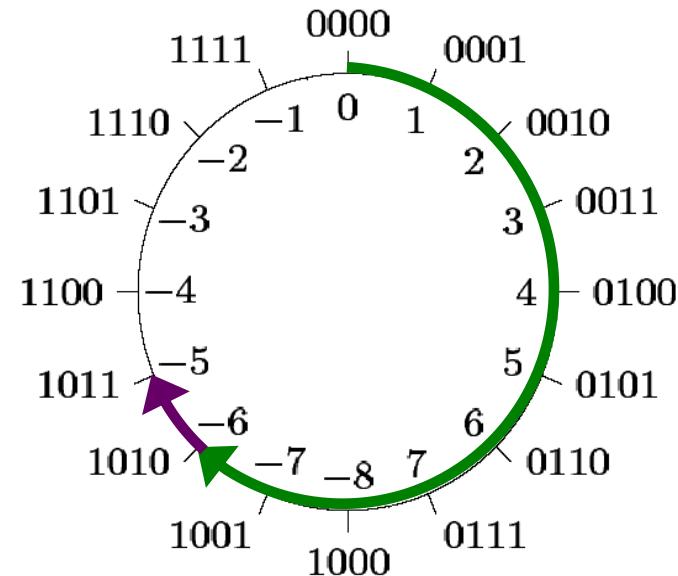
$$a - a = 0 \leftrightarrow a + OC(a) + 1 = 0$$



OC(a) is the bit-wise
inverse of a

$$\begin{array}{r} 1111 \\ - 0101 \\ \hline 0000 \\ \text{OC}(0101) = \underline{\underline{1010}} \end{array}$$

$$-a = OC(a) + 1 = TC(a)$$



$$\begin{aligned} -5d &= TC(0101) = \\ &1010 + 1 = 1011 \end{aligned}$$

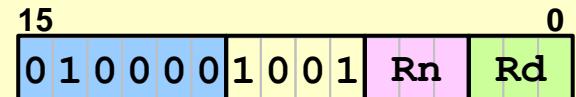
OC: 1' complement
TC: 2' complement

Negative Numbers

■ Instruction RSBS

- *Reverse Subtract*
- Generates 2' complement
- Updates flags
- Only low register possible

RSBS <Rd>, <Rn>, #0



$$Rd = 0 - Rn$$

00000022	4251	RSBS	R1, R2, #0
00000024	427F	RSBS	R7, R7, #0
00000026	427F	RSBS	R7, #0 ; the same (dest = R7)
00000028		;RSBS	R8, R1, #0 ;not possible (high reg)
00000028		;RSBS	R1, R8, #0 ;not possible (high reg)

Negative Numbers

■ Example: Instruction RSBS

C-Code

```
int32_t intA = 3;  
int32_t intB;  
  
int32_t invertSign(void)  
{  
    ...  
    intB = -intA;  
    ...  
}
```

Assembler-code

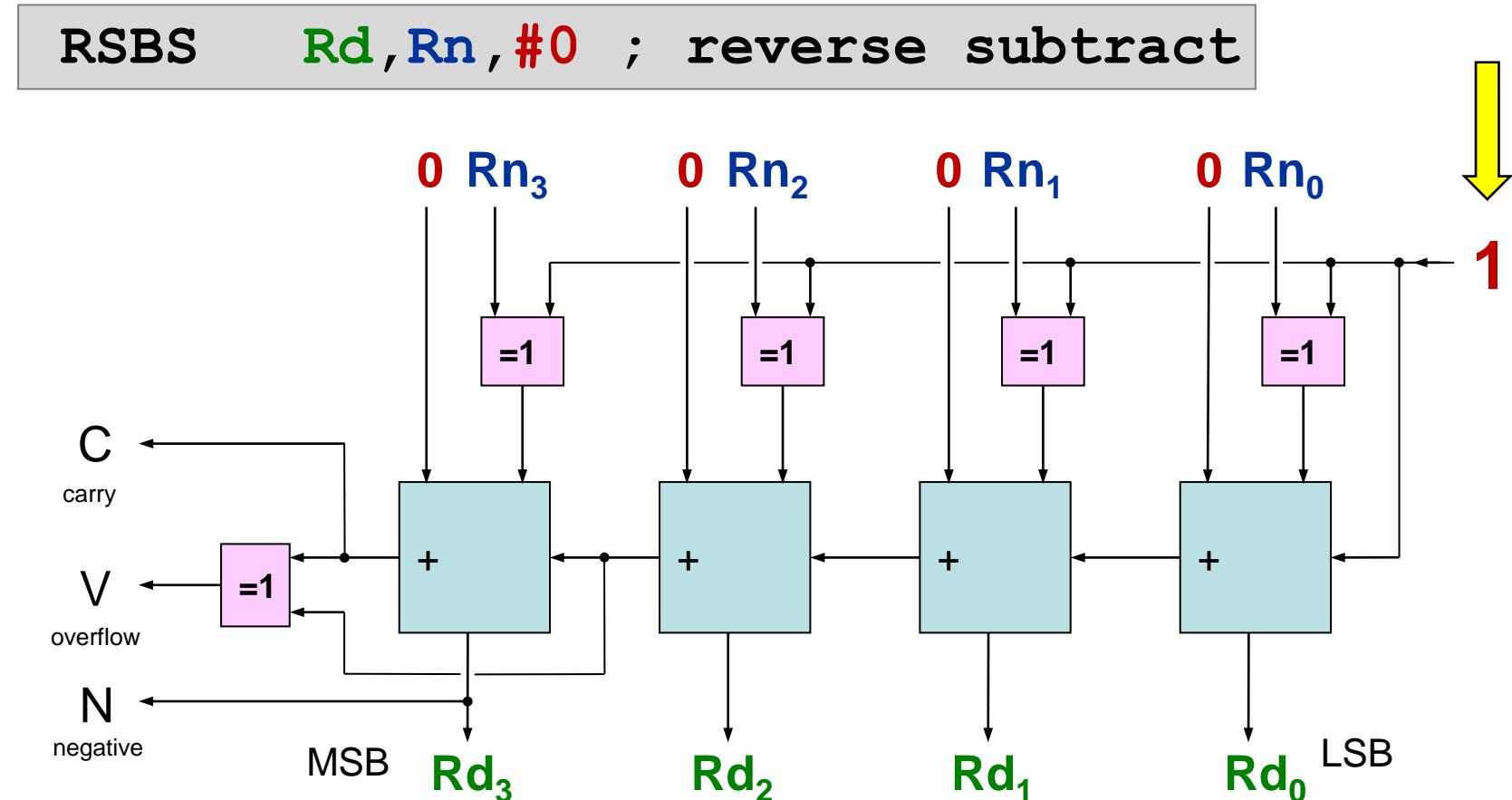
```
AREA MyCode, CODE, READONLY  
    ...  
    LDR    R0,=intA    ①  
    LDR    R0,[R0,#0]  
    RSBS  R0,R0,#0  
    LDR    R1,=intB    ①  
    STR    R0,[R1,#0]  
    ...  
  
AREA MyData, DATA, READWRITE  
intA  DCD 0x00000003  
intB  DCD 0x00000000
```



The compiler does not use the pseudo-instruction LDR in this case

Negative Numbers

■ 2' complement in Hardware (ALU)



Addition

■ Unsigned

- C = 1 indicates carry
- V irrelevant
- Addition of 2 big numbers
→ can yield a small result

Programmer interprets register content
either as signed or as unsigned.
CPU does not care!

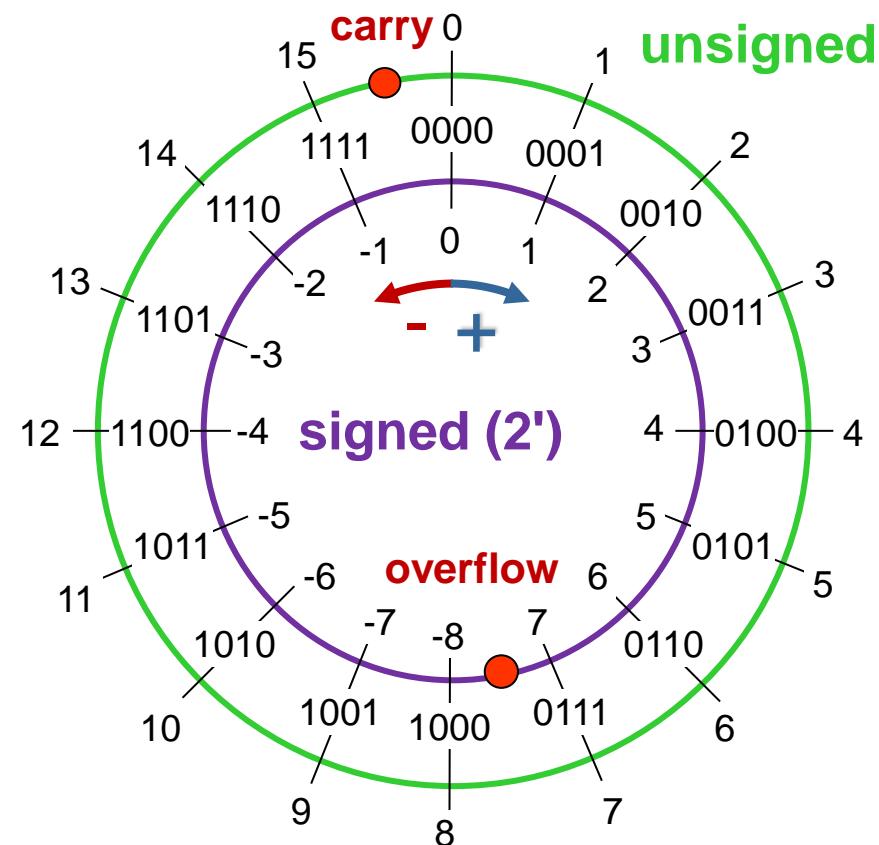
ADDS R1 ,R2 ,R3

■ Signed

- V = 1 indicates overflow
- possible with same „sign“



- C irrelevant

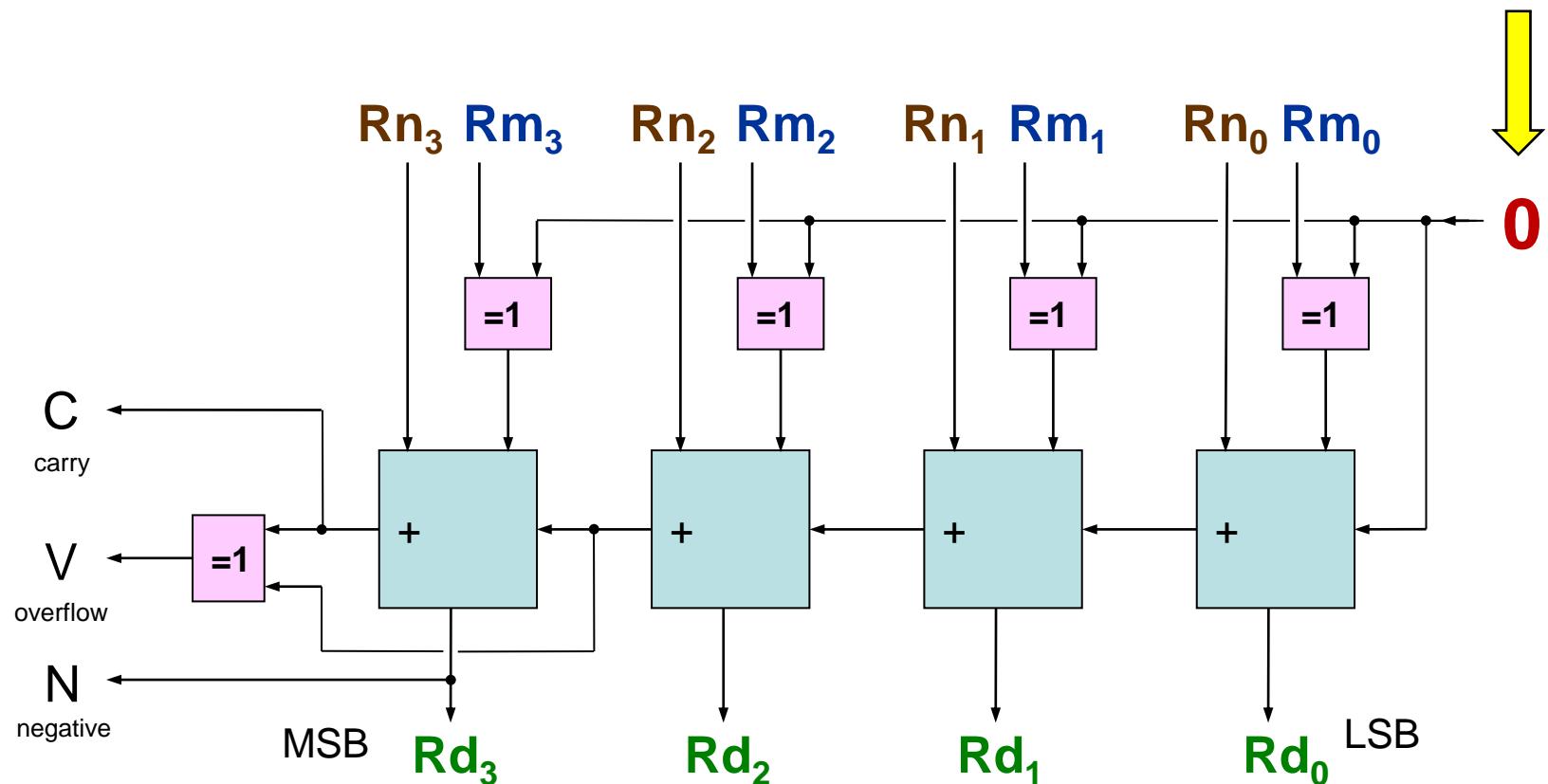


Addition

■ Addition in Hardware (ALU)

ADDS

Rd , Rn , Rm

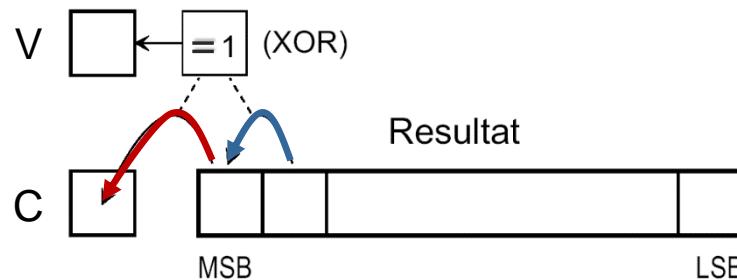


Addition

Signed

- Overflow

carry to MSB has different value than
carry from MSB



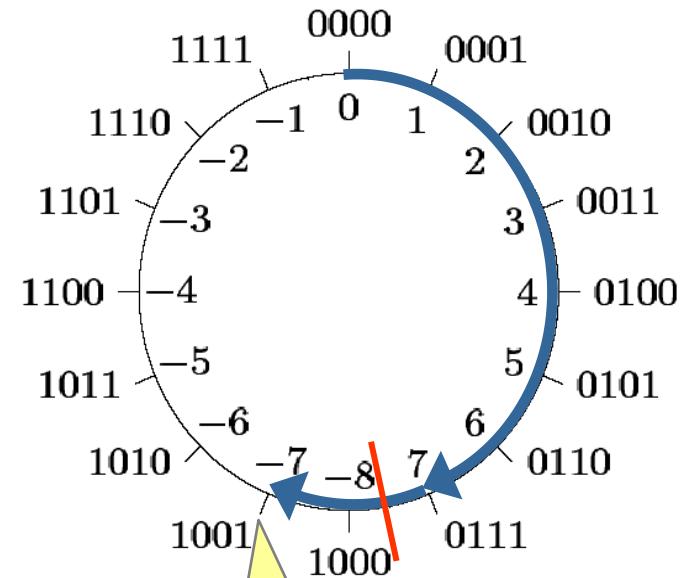
- Examples for 4 cases

$$\begin{array}{r} \textcolor{red}{0} \textcolor{blue}{1} \\ \boxed{\begin{array}{r} 0111 \\ +0010 \\ \hline 1001 \end{array}} \\ \hline V = 1 \end{array}$$

$$\begin{array}{r} \textcolor{red}{1} \textcolor{blue}{0} \\ \begin{array}{r} 1010 \\ +1101 \\ \hline 0111 \end{array} \\ \hline V = 1 \end{array}$$

$$\begin{array}{r} \textcolor{red}{0} \textcolor{blue}{0} \\ \begin{array}{r} 0001 \\ +1110 \\ \hline 1111 \end{array} \\ \hline V = 0 \end{array}$$

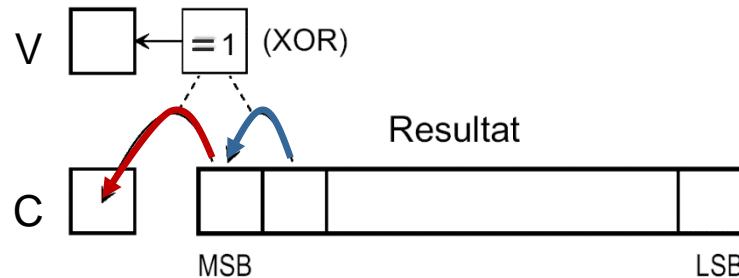
$$\begin{array}{r} \textcolor{red}{1} \textcolor{blue}{1} \\ \begin{array}{r} 0011 \\ +1111 \\ \hline 0010 \end{array} \\ \hline V = 0 \end{array}$$



Addition

Signed

- Overflow carry to MSB has different value than
 carry from MSB



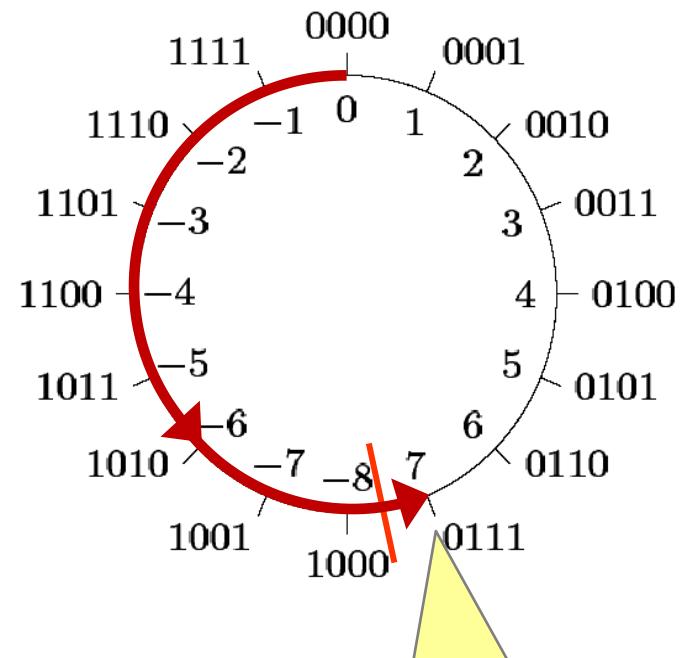
- Examples for 4 cases

$$\begin{array}{r}
 0111 \\
 +0010 \\
 \hline
 1001
 \end{array}$$

$$\begin{array}{r}
 1010 \\
 +1101 \\
 \hline
 0111
 \end{array}$$

$$\begin{array}{r}
 0001 \\
 +1110 \\
 \hline
 0\ 0\ 0\ 0 \\
 \hline
 1111
 \end{array}$$

$$\begin{array}{r}
 \textcolor{red}{1} \textcolor{blue}{1} \\
 \\
 \textbf{0011} \\
 + \textbf{1111} \\
 \hline
 \textbf{0010}
 \end{array}$$



Number -9 cannot be represented

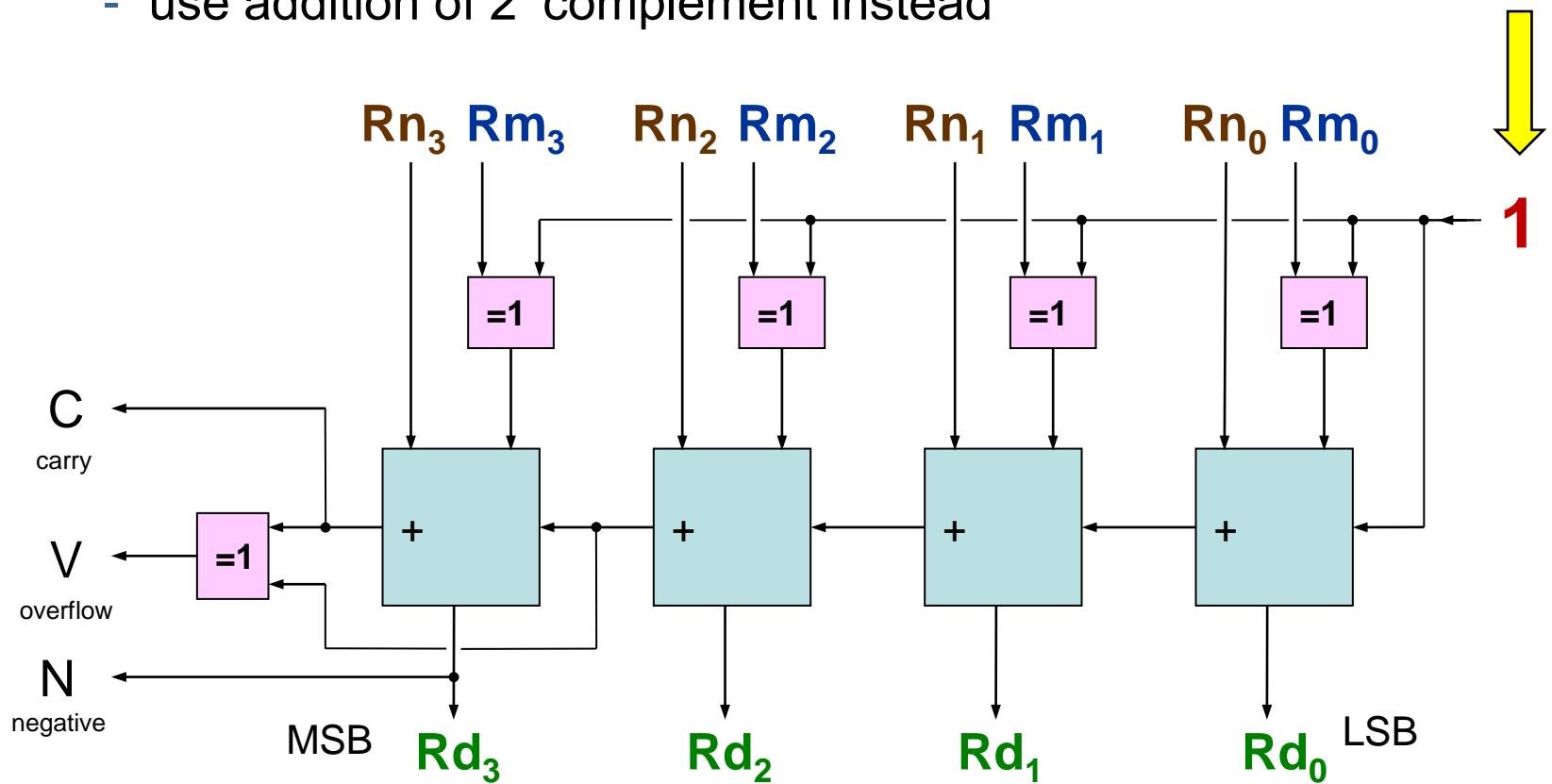
Subtraction

■ Subtraction in Hardware (ALU)

SUBS

Rd , Rn , Rm

- There is no subtraction!
 - use addition of 2' complement instead



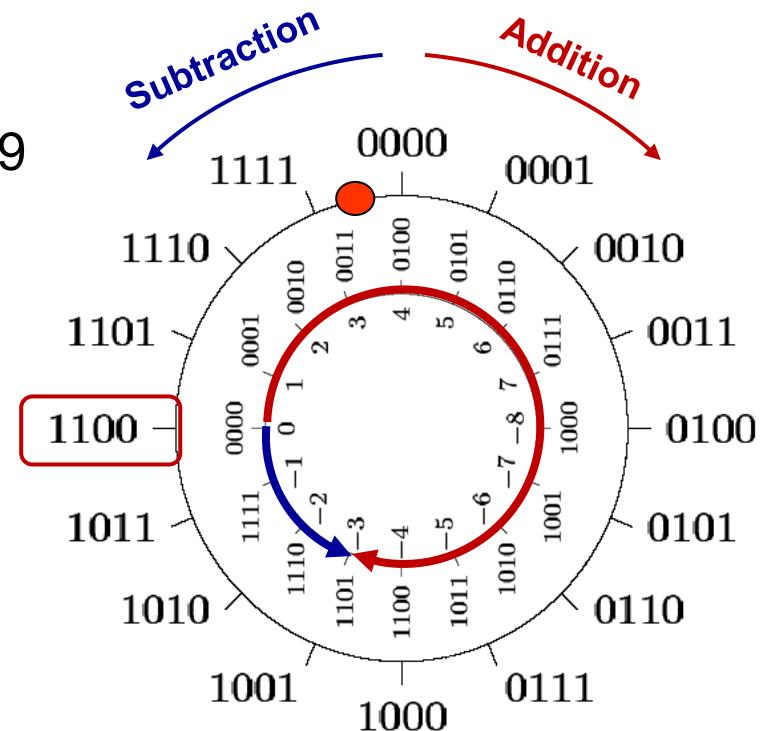
Subtraction

■ **unsigned**

- Use 2' complement as well
 - Example 4-Bit unsigned: $12 - 3 = 9$

$ \begin{array}{r} 12 \\ - 3 \\ \hline 9 \end{array} $	$ \begin{array}{r} 1100 \\ - 0011 \\ \hline 1001 \end{array} $	$ \begin{array}{r} 1100 \\ + 1101 \\ \hline 1\ 1001 \end{array} $
human		computer

- Attention
 - $C = 1 \rightarrow \underline{\text{NO}}$ borrow
 - $C = 0 \rightarrow \text{borrow}$
 - Borrow
 - operation yields negative result \rightarrow cannot be represented in unsigned
 - in multi-word operations missing digits are borrowed from more significant word



Subtraction

■ Unsigned

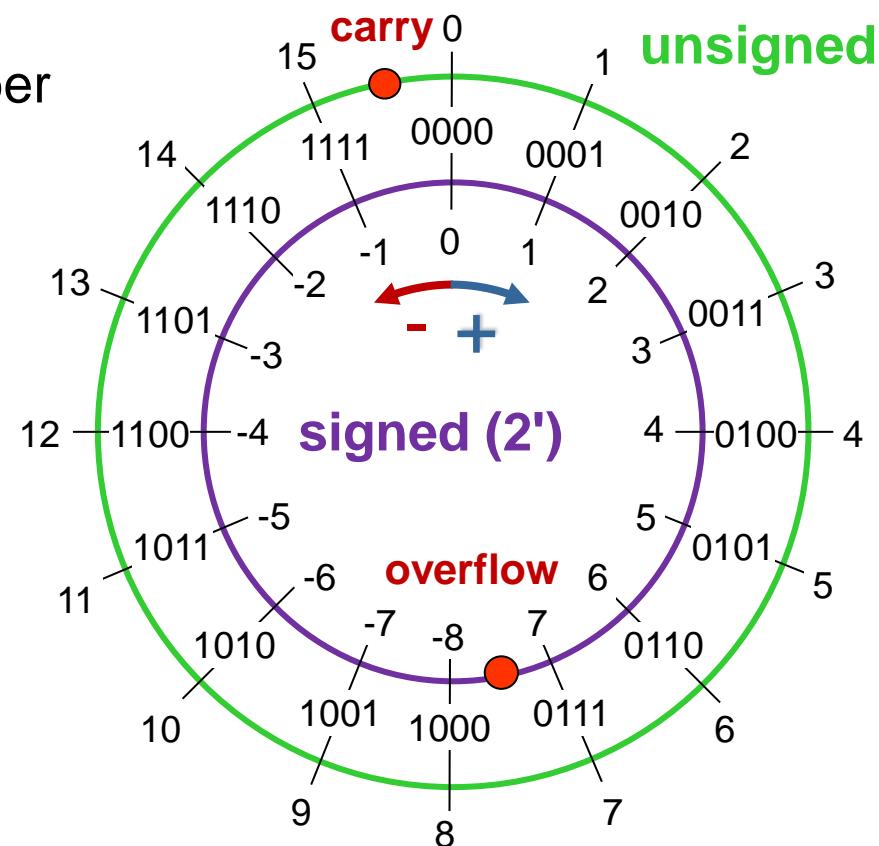
- C = 0 indicates borrow
- V irrelevant
- Subtraction from a small number
→ can yield a big result

■ Signed

- V = 1 indicates overflow or underflow
 - Possible with opposite signs
 - NOT possible when operands have same sign
- C irrelevant

Programmer interprets register content either as signed or as unsigned.
CPU does not care!

SUBS R1 ,R2 ,R3



Subtraction Operations Cortex-M0

■ SUBS (register)

- Updates flags
- Result and 2 operands
- Only low register

SUBS <Rd>, <Rn>, <Rm>

15 0 0 0 1 1 0 1 0
Rm Rn Rd

$$\begin{aligned} \text{Rd} &= \text{Rn} - \text{Rm} \\ &= \text{Rn} + \text{NOT}(\text{Rm}) + 1 \end{aligned}$$

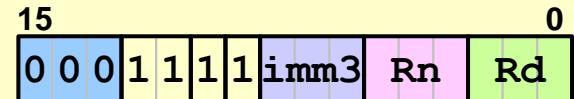
00000016	1AD1	SUBS	R1, R2, R3	
00000018	1B64	SUBS	R4, R4, R5	
0000001A	1B64	SUBS	R4, R5	; the same (dest = R4)
0000001C				
0000001C		;SUBS	R8, R4, R5	; not possible (high reg)
0000001C		;SUBS	R4, R8, R5	; not possible (high reg)
0000001C		;SUBS	R4, R5, R8	; not possible (high reg)

Subtraction Operations Cortex-M0

■ SUBS (immediate) – T1

- Updates flags
- 2 different low registers and immediate value 0 - 7d

SUBS <Rd>, <Rn>, #<imm3>



$$\begin{aligned} \text{Rd} &= \text{Rn} - \text{imm3} \\ &= \text{Rn} + \text{NOT} \text{imm3} + 1 \end{aligned}$$

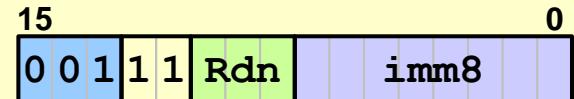
0000001C 1F63	SUBS	R3, R4, #5	
0000001E	;SUBS	R3, R4, #8	; out of range immediate
0000001E	;SUBS	R10, R11, #5	; not possible (high reg)

Subtraction Operations Cortex-M0

■ SUBS (immediate) – T2

- Updates flags
- One low register and immediate value 0 - 255d
- <Rdn> → Result and operand
→ same register for result and operand

SUBS <Rdn>, #<imm8>



$$\begin{aligned} \text{Rdn} &= \text{Rdn} - \text{imm8} \\ &= \text{Rdn} + \text{NOT}\text{imm8} + 1 \end{aligned}$$

```
0000001E 3BF0    SUBS      R3,R3,#240
00000020 3BF0    SUBS      R3,#240      ; the same (dest = R3)
00000022        ;SUBS     R8,R8,#240      ; not possible (high reg)
00000022        ;SUBS     R3,#260      ; out of range immediate
```

Addition / Subtraction

■ Interpretation of carry flag

- Addition

$$C = 1 \rightarrow \text{Carry}$$

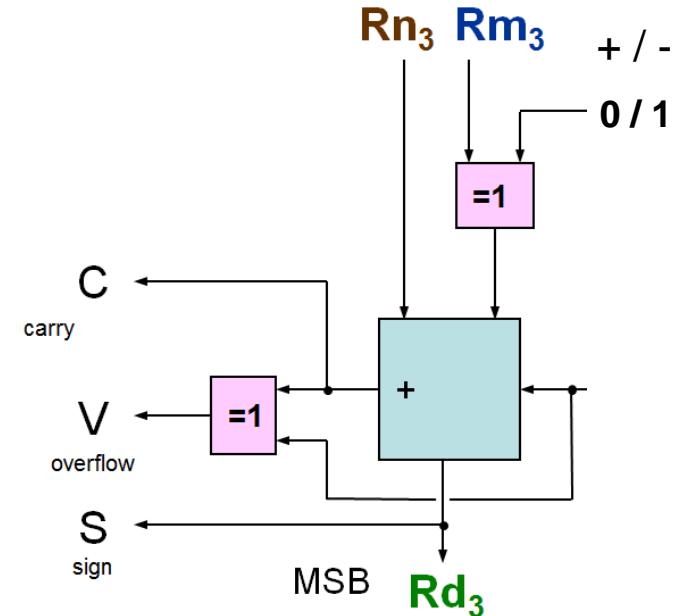
$$\begin{array}{r} 1101 & 13d \\ 0111 & 7d \\ 1111 \\ \hline 10100 & 20d \rightarrow 16d + 4d \end{array}$$

- Subtraction

$$C = 0 \rightarrow \text{Borrow}$$

$$6d - 14d = 0110b - 1110b = 0110b + 0010b$$

$$\begin{array}{r} 0110 & 6d \\ 0010 & 2d = TC(14d) \\ 0110 \\ \hline 01000 & 8d \rightarrow - 16d + 8d \end{array}$$



Addition / Subtraction

■ **unsigned** Interpretation

- Program must check **carry flag (C)** after operation
- **C = 1 for Addition C = 0 for Subtraction**
 - Result cannot be represented (not enough digits / no negative numbers)
 - Full turn on number circle must be added or subtracted
 - odometer effect
- Overflow flag (V) irrelevant

■ **signed** Interpretation

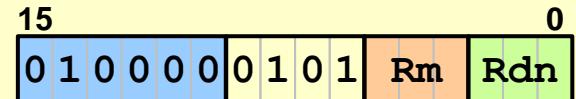
- Program must check **overflow flag (V)** after operation
- **V = 1** means
 - Not enough digits available to represent the result
 - Full turn on number circle must be added or subtracted
 - odometer effect
- Carry flag (C) irrelevant

Multi-Word Arithmetic

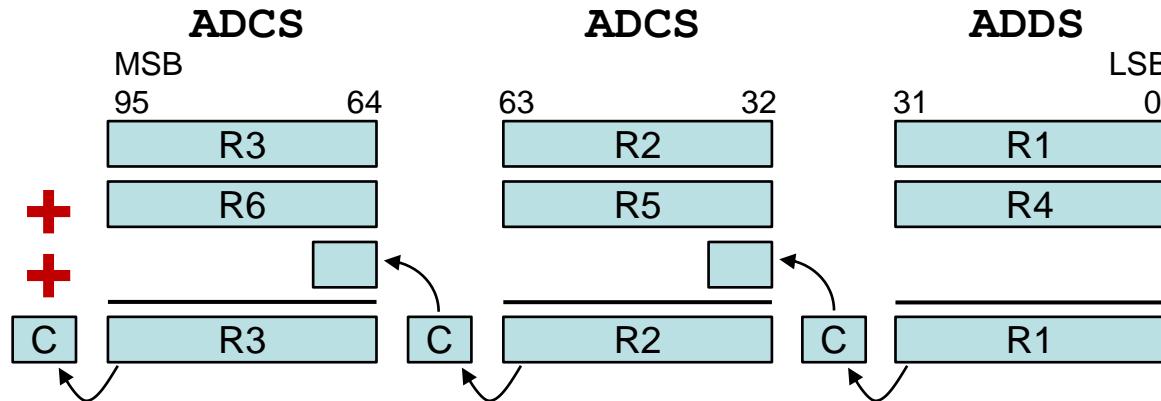
■ Multi-Word Addition ADCS

- Example: Addition of two 96-bit Operands

ADCS <Rdn>, <Rm>



$$Rdn = Rdn + Rm + C$$



; Operand A: 96-bit in R3 (MSW), R2, R1 (LSW)

; Operand B: 96-bit in R6 (MSW), R5, R4 (LSW)

; Result = A + B: 96-bit in R3 (MSW), R2, R1 (LSW)

00000028 1909 ADDS R1, R1, R4

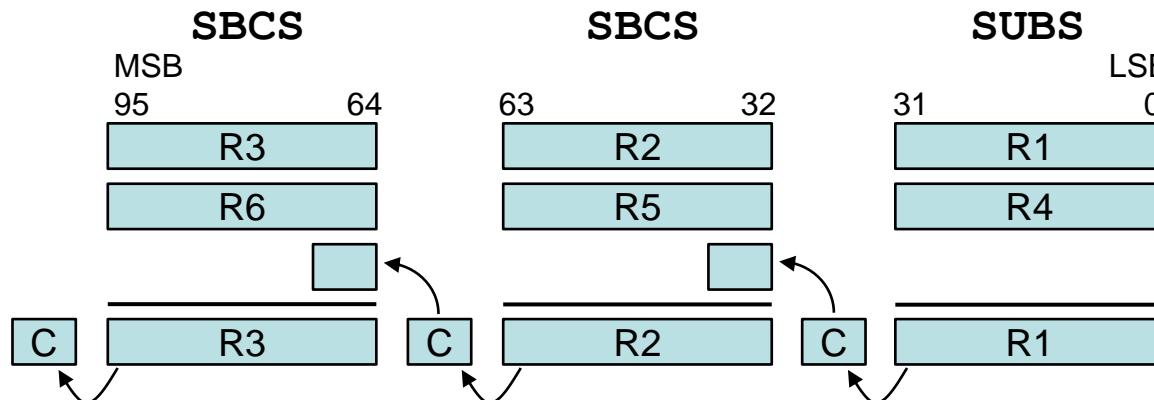
0000002A 416A ADCS R2, R2, R5

0000002C 4173 ADCS R3, R3, R6

Multi-Word Arithmetic

■ Multi-Word Subtraction SBCS

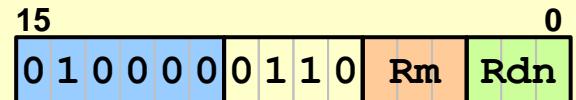
- Example: Subtraction of two 96-bit Operands



```
; Operand A: 96-bit in R3 (MSW), R2, R1 (LSW)
; Operand B: 96-bit in R6 (MSW), R5, R4 (LSW)
; Result A - B: 96-bit in R3 (MSW), R2, R1 (LSW)

0000002E 1B09 SUBS    R1, R1, R4
00000030 41AA SBCS    R2, R2, R5
00000032 41B3 SBCS    R3, R3, R6
```

SBCS <Rdn>, <Rm>



$$\begin{aligned} Rdn &= Rdn - Rm - \text{NOT}(C) \\ &= Rdn + \text{NOT}(Rm) + C \end{aligned}$$

¹⁾

SUBS

LSB

¹⁾ $-Rm - \text{NOT}(C)$
 $= (\text{NOT}(Rm) + 1) - \text{NOT}(C)$
 $= \text{NOT}(Rm) + C$

The CPU actually calculates $Rdn + \text{NOT}(Rm) + C$

Multiplication

■ MULS (register)

- Flags
 - N and Z updated
 - C and V unchanged
- Operands
 - Rn multiplicand
 - Rdm multiplier
 - only low registers
- Result
 - **Rdm contains only lowest 32 bits of product**

MULS <Rdm>, <Rn>, <Rdm>

15	0 1 0 0 0 0	1 1 0 1	Rn	Rdm	0
----	-------------	---------	----	-----	---

Rdm = Rn * Rdm

0000002E 4351 MULS R1,R2,R1

00000030 ;MULS R1,R1,R2 ; not possible: destination and
00000030 ; 2nd source must be same
00000030 ;MULS R1,R8,R1 ; not possible: high reg

Multiplication

- Result requires twice as many binary digits
 - Example 4-bit * 4-bit → 8-bit result
- Signed and unsigned multiplication are different

$$\begin{array}{r} 0101 * 0011 \\ \hline 0011 \\ 0000 \\ 0011 \\ 0000 \\ \hline 00001111 \end{array}$$

unsigned

$$\begin{array}{r} 0101 * 1101 \\ \hline 00001101 \\ 00000000 \\ 00001101 \\ 00000000 \\ \hline 00001000001 \end{array}$$

zero extension of multiplier

signed

$$\begin{array}{r} 0101 * 1101 \\ \hline 11111101 \\ 00000000 \\ 11111101 \\ 00000000 \\ \hline 10011110001 \end{array}$$

sign extension of multiplier

Interpretation **unsigned**
 $5d * 3d = 15d \rightarrow \text{correct}$

Interpretation **unsigned**
 $5d * 13d = 65d \rightarrow \text{correct}$

Interpretation **unsigned**
 $5d * 13d = 241d \rightarrow \text{wrong}$

Interpretation **signed**
 $5d * 3d = 15d \rightarrow \text{correct}$

Interpretation **signed**
 $5d * -3d = -15d \rightarrow \text{wrong}$

Interpretation **signed**
 $5d * -3d = -15d \rightarrow \text{correct}$

■ MULS on Cortex-M0

- Rn (32-bit) * Rdm (32-bit) → Rdm (32-bit)
 - Upper 32-bit of result are lost!
 - Lower 32-bit are the same for unsigned and signed
- unsigned → watch out if result requires more than 32 bits
- signed → part with sign bit is missing

■ Processor Arithmetic

- Odometer effect because of finite word length

■ Processors do **not** distinguish **signed** and **unsigned**

- User (resp. compiler) has to know, whether he is working with signed or unsigned numbers
- Processor always calculates flags for both cases
 - carry (C) unsigned
 - overflow (V) signed
 - negative (N)

Conclusion

■ **unsigned**

- Addition → $C = 1 \rightarrow$ carry
result too large for available bits
- Subtraction → $C = 0 \rightarrow$ borrow
result less than zero → no negative numbers

■ **signed**

- Addition → potential overflow in case of operands with equal signs
- Subtraction → potential overflow in case of operands with opposite signs

■ **Arithmetic Instructions**

- | | | |
|-------------------|-------------|-------------|
| • ADD/ADDS | ADCS | ADR |
| SUB/SUBS | SBCS | RSBS |
| MULS | | |

Motivation

■ How often is the for loop executed?

- For `#define INCREMENT 1` → 255 times
- For `#define INCREMENT 10` → infinite times
 ≥ 255 never reached

```
...
int main(void) {
    uint8_t uc;
    uint32_t count = 0;

    for (uc = 0; uc < 255; uc += INCREMENT) {
        printf("hello again %d \n", uc);
        count++;
    }
    printf("Loop executed %d times\n", count);
}
```

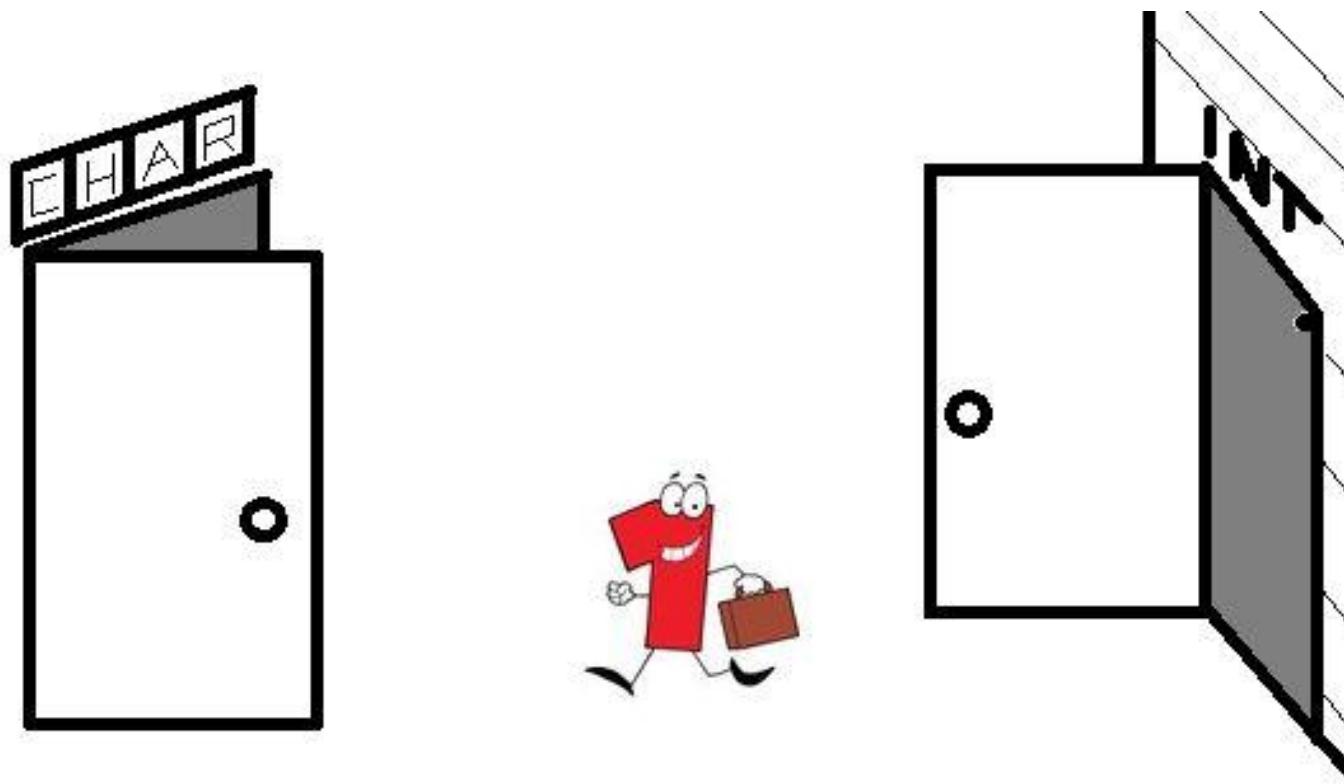
Why?

```
.....
hello again 210
hello again 220
hello again 230
hello again 240
hello again 250
hello again 4
hello again 14
hello again 24
hello again 34
hello again 44
.......
```

Integer Casting in C

Computer Engineering 1

Motivation



<http://www.instructables.com>

Agenda

- Type conversion
 - signed ↔ unsigned
 - Extension
 - Truncation

Learning Objectives

At the end of this lesson you will be able

- to explain the casting of integer types in C
- to apply the assembly instructions associated with casting
- to say how a given memory content is interpreted for different integer types in C
- to give the memory content after storing different C integer types

Integer Casting in C

■ Integer ranges based on word sizes

8-bit	hex	unsigned	signed
	0x00	0	0
...
0x7F	127	127	127
0x80	128	-128	-128
...
0xFF	255	-1	-1

16-bit	hex	unsigned	signed
	0x0000	0	0
...
0x7FFF	32'767	32'767	32'767
0x8000	32'768	-32'768	-32'768
...
0xFFFF	65'535	-1	-1

32-bit	hex	unsigned	signed
	0x0000 0000	0	0
...
0x7FFF'FFFF	2'147'483'647	2'147'483'647	2'147'483'647
0x8000'0000	2'147'483'648	-2'147'483'648	-2'147'483'648
...
0xFFFF'FFFF	4'294'967'295	-1	-1

Integer Casting in C: Type conversion

■ signed ↔ unsigned

<code>int8_t</code>	\leftrightarrow	<code>uint8_t</code>
<code>int16_t</code>	\leftrightarrow	<code>uint16_t</code>
<code>int32_t</code>	\leftrightarrow	<code>uint32_t</code>
<code>int64_t</code>	\leftrightarrow	<code>uint64_t</code>

■ Extension

<code>int_8</code>	\rightarrow	<code>int16_t</code>	\rightarrow	<code>int32_t</code>	\rightarrow	<code>int64_t</code>	signed
<code>uint_8</code>	\rightarrow	<code>uint16_t</code>	\rightarrow	<code>uint32_t</code>	\rightarrow	<code>uint64_t</code>	unsigned

■ Truncation

<code>int64_t</code>	\rightarrow	<code>int32_t</code>	\rightarrow	<code>int16_t</code>	\rightarrow	<code>int_8</code>	signed
<code>uint64_t</code>	\rightarrow	<code>uint32_t</code>	\rightarrow	<code>uint16_t</code>	\rightarrow	<code>uint_8</code>	unsigned

Integer Casting in C

■ signed \leftrightarrow unsigned

signed $-b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

unsigned $+b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Casts in red area

→ Small negative numbers turn
into large positive numbers

→ Large positive numbers turn
into small negative numbers

binary	unsigned	signed 2^4 compl.
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Integer Casting in C

■ Casting

unsigned	→	signed
signed	→	unsigned

Bit representation stays the same,
but interpretation changes

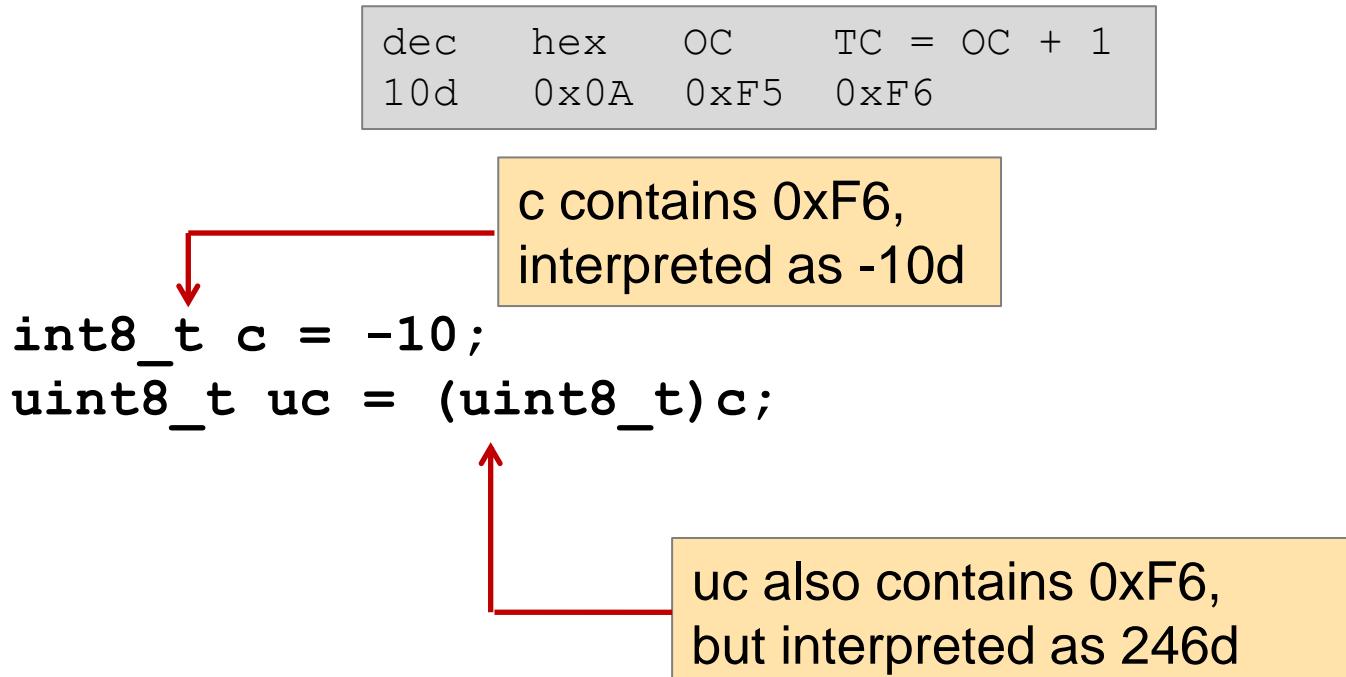
■ Example 4-Bit

- 1011b → Interpretation as unsigned 11d
 → Interpretation as signed -5d

Integer Casting in C

■ Example 1: signed 8-bit → unsigned 8-bit

- Bit representation stays the same, interpretation changes



Integer Casting in C

- **Example 2: signed 32-bit → unsigned 32-bit**
 - Bit representation stays the same, interpretation changes

dec	hex	OC	TC = OC + 1
9133d	0x0000'23AD	0xFFFF'DC52	0xFFFF'DC53

```
int32_t i = -9133;  
uint32_t ui = (uint32_t)i;
```

i contains 0xFFFF'DC53,
interpreted as -9133d

ui also contains 0xFFFF'DC53,
but interpreted as 4'294'958'163d

Explicit cast is not even required

```
uint32_t ui2 = i; // implicit cast
```

Integer Casting in C

■ Example 3: Cast unsigned 32-bit → signed 32-bit

- Bit representation stays the same, interpretation changes

dec	hex
4'294'964'632d	0xFFFF'F598

uix contains 0xFFFF'F598,
interpreted as 4'294'964'632

```
uint32_t uix = 4294964632;  
int32_t ix = uix; // implicit cast
```

ix also contains 0xFFFF'F598,
but interpreted as -2664d

Integer Casting in C

- If one of the operands is unsigned, C performs an implicit cast for the signed values to unsigned
 - Example n = 32: signed $\in [-2^{147}483'648, 2^{147}483'647]$
 - Can lead to strange results (red lines)

Expression	Type	Evaluation
$0 == 0U$	unsigned	1
$-1 < 0$	signed	1
$-1 < 0U$	unsigned	0
$2^{147}483'647 > -2^{147}483'647 - 1$	signed	1
$2^{147}483'647U > -2^{147}483'647 - 1$	unsigned	0
$2^{147}483'647 > (int) 2^{147}483'648U$	signed	1
$-1 > -2$	signed	1
$(unsigned) -1 > -2$	unsigned	1

Examples: R. Bryant, D. O'Hallaron

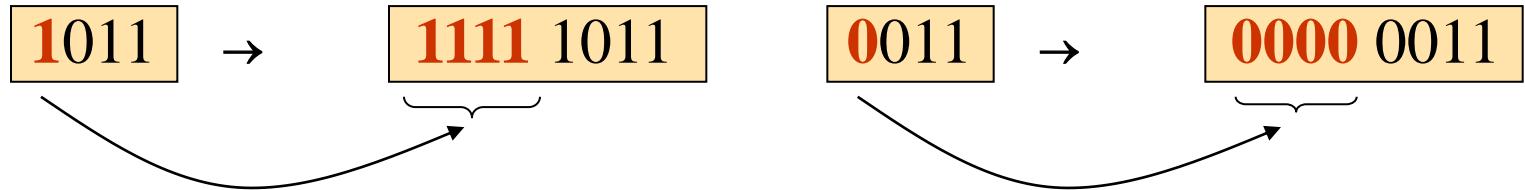
Integer Casting

■ Extension: 4 Bit → 8 Bit

- Unsigned → Zero Extension



- Signed → Sign Extension



■ **Sign Extension Cortex-M0 (signed values)**

- Extend word-length without changing value
- **SXTB** Extends an 8-bit value to a 32-bit value
- **SXTH** Extends a 16-bit value to a 32-bit value

■ **Zero Extension Cortex-M0 (unsigned values)**

- Extend word-length, fill with zeroes
- **UXTB** Extends an 8-bit value to a 32-bit value
- **UXTH** Extends a 16-bit value to a 32-bit value

Examples

```
SXTB  R3, R10 ; Extract lowest byte of the value in R10,  
           ; sign extend it and write the result to R3  
UXTH  R2, R3  ; Extract lower two bytes of the value in R3,  
           ; zero extend it and write the result to R2
```

Integer Casting: Sign Extension

■ Example Sign Extension

```
int16_t sx = 15213;  
int32_t ix = (int32_t)sx;  
  
int16_t sy = -15213;  
int32_t iy = (int32_t)sy;
```

	dec	hex	bin		
sx	15213	0x3B6D		00111011	01101101
ix	15213	0x0000'3B6D	00000000 00000000	00111011	01101101
sy	-15213	0xC493		11000100	10010011
iy	-15213	0xFFFF'C493	11111111 11111111	11000100	10010011

signed Integer Types: from small to large

→ **Sign bit is copied to the left**

Integer Casting: Truncation

- **Truncation: Reduce number of digits**
 - Cast cuts the left most digits
- **Unsigned** → modulo Operation

```
uint32_t x = 287962;  
uint16_t sx = (uint16_t)x;  
uint32_t y = (uint32_t)sx;
```

0x000464DA →	287'962
0x64DA →	25'818
0x000064DA →	25'818

- **Signed** → possible change of sign!

```
int32_t x = 53191;  
int16_t sx = (int16_t)x;  
int32_t y = (int32_t)sx;
```

0x0000CFC7 →	53'191
0xCFC7 →	-12'345
0xFFFFFCFC7 →	-12'345

Conclusion

■ Integer Casts

- Type Conversions

■ signed – unsigned

- Small negative numbers correspond to large positive numbers

■ Extensions

- Add additional bits

- signed	sign extension	copy sign bit to the left
- unsigned	zero extension	fill left bits with zero

■ Truncations

- Cut left most digits

- signed	possible change of sign
- unsigned	results in modulo operation

Logic and Shift/Rotate Instructions

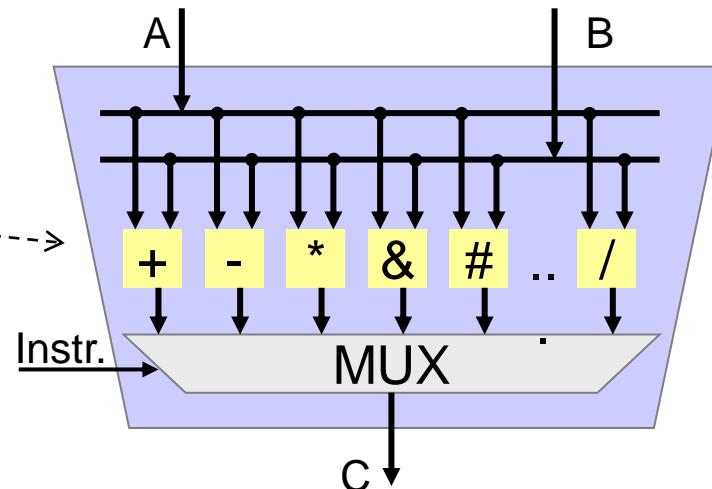
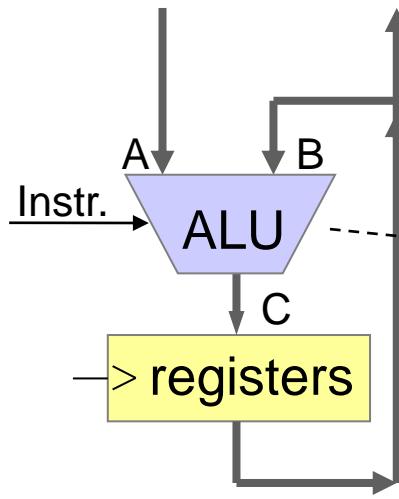
Computer Engineering 1

- **So far, operands were bytes, half-words or words**
 - How can we work on a unit as small as a bit?
 - How can we look at a bit or a pattern of bits within a larger unit?
 - How can we modify a particular bit or a pattern of bits without affecting other bits?
- **Logic operations**
 - Help deal efficiently with elements at bit level
 - Enable the implementation of algorithms where Boolean operations are needed
- **Like in HW, shift operations can enable**
 - Division and multiplication
 - Test of certain bits
 - Communication, ...

Motivation

■ Instructions to process data in ALU

- **arithmetic** Addition, Subtraction, Increment, Decrement, Multiplication, Division
- **logic** NOT, AND, OR, XOR
- **shift** Left/right shift. Fill up with 0 or MSB
- **rotate** Cyclic left/right shift: What falls out enters on the other side.



Agenda

- **Logic Instructions**
 - Bit manipulations
- **Shift/Rotate Instructions**
- **Multiplication with a Constant**

Learning Objectives

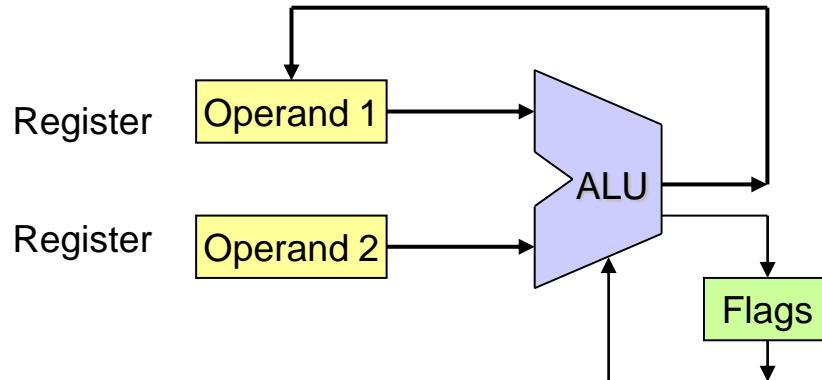
At the end of this lesson you will be able

- to enumerate and to apply the ARM instructions for logic as well as for shift/rotate operations
- to determine (with the help of documents) the state of the ARM Flags (N, Z, C, V) after the execution of an instruction
- to understand and interpret ARM assembly programs with logic and shift/rotate operations
- to explain bit manipulations based on examples
- to set, clear or invert one or several bits in a bit pattern
- to implement a multiplication with a constant in assembly using shift and add instructions

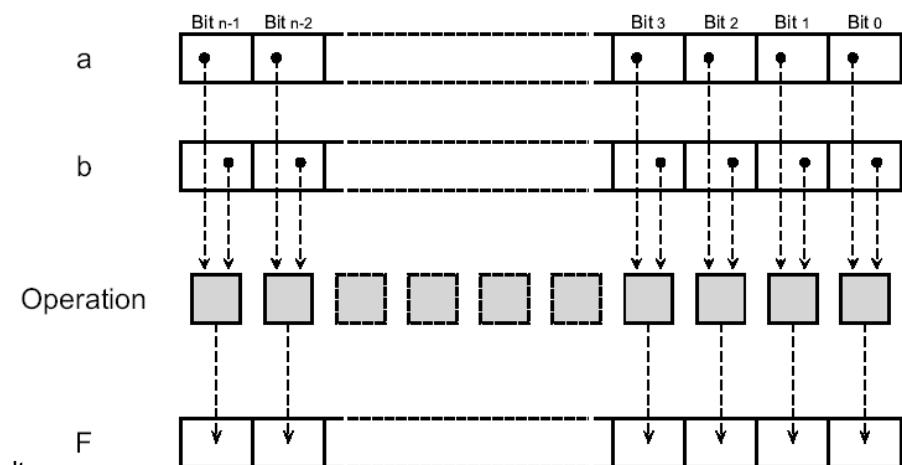
Logic Instructions

■ Overview

Mnemonic	Instruction	Function	C-Operator	
ANDS	Bitwise AND	Rdn & Rm	a & b	flags N = result<31> ¹⁾ Z = 1 if result = 0 Z = 0 otherwise C and V unchanged
BICS	Bit Clear	Rdn & !Rm	a & ~b	
EORS	Exclusive OR	Rdn \$ Rm	a ^ b	
MVNS	Bitwise NOT	!Rm	~a	
ORRS	Bitwise OR	Rdn # Rm	a b	



¹⁾ i.e. N is equal to bit 31 of the result
MSB shows the sign of the result



Logic Instructions

■ Bitwise Operations

ANDS <Rdn>, <Rdn>, <Rm>

15	0 1 0 0 0 0	0 0 0 0	Rm	Rdn	0
----	-------------	---------	----	-----	---

$Rdn = Rdn \& Rm$

BICS <Rdn>, <Rdn>, <Rm>

15	0 1 0 0 0 0	1 1 1 0	Rm	Rdn	0
----	-------------	---------	----	-----	---

$Rdn = Rdn \& !Rm$

EORS <Rdn>, <Rdn>, <Rm>

15	0 1 0 0 0 0	0 0 0 1	Rm	Rdn	0
----	-------------	---------	----	-----	---

$Rdn = Rdn \$ Rm$

MVNS <Rd>, <Rm>

15	0 1 0 0 0 0	1 1 1 1	Rm	Rd	0
----	-------------	---------	----	----	---

$Rd = !Rm$

ORRS <Rdn>, <Rdn>, <Rm>

15	0 1 0 0 0 0	1 1 0 0	Rm	Rdn	0
----	-------------	---------	----	-----	---

$Rdn = Rdn \# Rm$

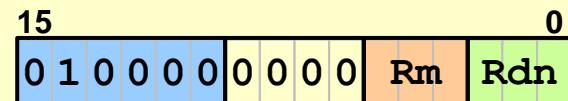
- All these operations affect the flags:
 - N and Z according to result
 - C and V unchanged
- Only low registers

Logic Instructions

■ Example

- Update flags N and Z
 - Only low registers possible!

ANDS <Rdn>, <Rdn>, <Rm>



Rdn = Rdn & Rm

00000002	4011	ANDS	R1,R1,R2	; R1 = R1 AND R2
00000004	4011	ANDS	R1,R2	; the same (dest = R1)
00000006	4337	ORRS	R7,R7,R6	; R7 = R7 OR R6
00000008	4063	EORS	R3,R3,R4	; R3 = R3 EXOR R4
0000000A	4388	BICS	R0,R0,R1	; R0 = R0 AND NOT(R1)
0000000C	43D1	MVNS	R1,R2	; R1 = NOT(R2)

■ Bit Manipulations (Cortex-M0)

- **Clear bits**, e.g. clear bits **5** and **1** in register R1

```
MOVS    R2, #0x22      ; 00100010b
BICS    R1, R1, R2
```

- **Set bits**, e.g. set bits **6** und **3** in register R1

```
MOVS    R2, #0x48      ; 01001000b
ORRS    R1, R1, R2
```

- **Invert bits**, e.g. invert bits **4**, **3** and **2** in register R1

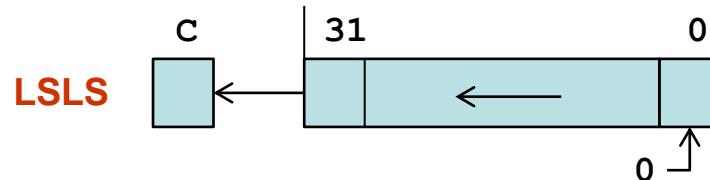
```
MOVS    R2, #0x1C      ; 00011100b
EORS    R1, R1, R2
```

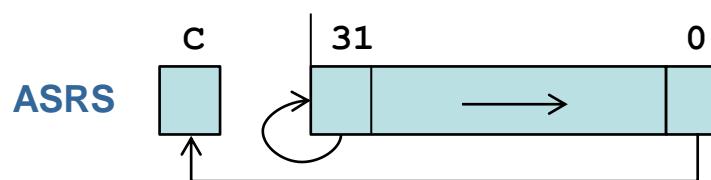
- What happens to the other bits?
- Compute the value of R1 after execution of all lines assuming that R1 contains **0xAF08'24A3** at the start.
- What are the values of the flags after execution?

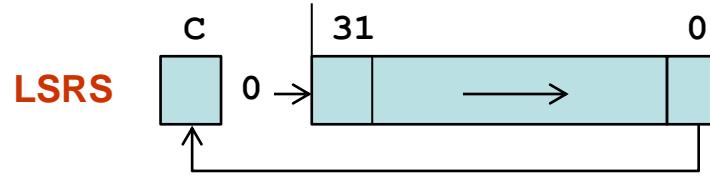
Shift / Rotate Instructions

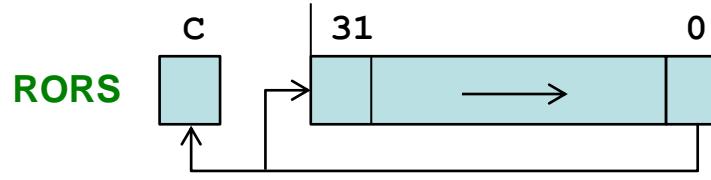
■ Overview

Mnemonic	Instruction	Function
LSLS	Logical Shift Left	$2^n \cdot Rn$ $0 \rightarrow \text{LSB}$
LSRS	Logical Shift Right	$2^{-n} \cdot Rn$ $0 \rightarrow \text{MSB}$
ASRS	Arithmetic Shift Right	$2^{-n} \cdot \pm A$ $\text{MSB} \rightarrow \text{MSB}$
RORS	Rotate Right	$\text{LSB} \rightarrow \text{MSB}$

LSLS 

ASRS 

LSRS 

RORS 

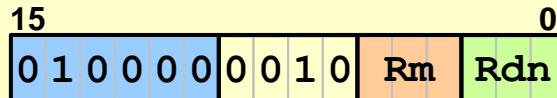
Note: rotate left does not exist

Shift / Rotate Instructions

■ Opcodes (register)

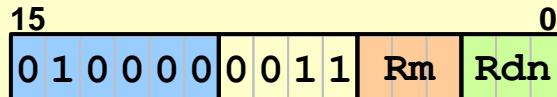
- Low registers only
- Flags affected

LSLS <Rdn>, <Rdn>, <Rm>



Rdn = shift Rdn left
by Rm<7:0> bits,
fill with zeros²⁾

LSRS <Rdn>, <Rdn>, <Rm>



Rdn = shift Rdn right
by Rm<7:0> bits,
fill with zeros²⁾

flags N = result<31>¹⁾

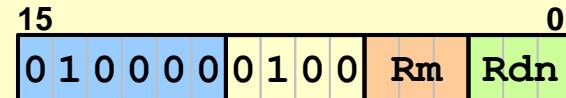
Z = 1 if result = 0

Z = 0 otherwise

C = last bit shifted out

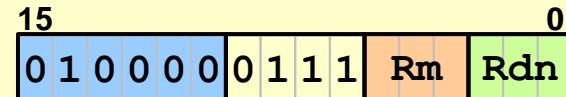
V unchanged

ASRS <Rdn>, <Rdn>, <Rm>



Rdn = shift Rdn right
by Rm<7:0> bits,
fill with MSB²⁾

RORS <Rdn>, <Rdn>, <Rm>



Rdn = cyclic rotate right
by Rm<7:0> bits

¹⁾ i.e. N is equal to bit 31 of the result / MSB shows the sign of the result

²⁾ see previous slide

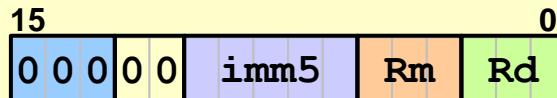
Shift / Rotate Instructions

■ Opcodes (immediate 0 – 31d)

- low registers only

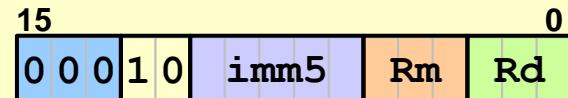
flags	N = result<31> ¹⁾ Z = 1 if result = 0 Z = 0 otherwise C = last bit shifted out V unchanged
-------	---

LSLS <Rd>, <Rm>, #<imm5>



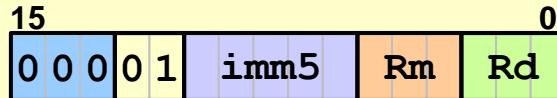
Rd = shift Rm left
by <imm5> bits
fill with zeros

ASRS <Rd>, <Rm>, #<imm5>



Rd = shift Rm right
by <imm5> bits
fill with MSB

LSRS <Rd>, <Rm>, #<imm5>



Rd = shift Rm right
by <imm5> bits
fill with zeros

- RORS (immediate) not supported
- LSRS/ASRS
 - <imm5> = 0 not allowed
- LSLS
 - C unaffected if <imm5> = 0

Shift / Rotate Instructions

■ Examples

00000000 4904	LDR	R1 ,=0xCCCCCCCC
00000002 2203	MOVS	R2 ,#3
00000004 4B04	LDR	R3 ,=0x66666666
00000006 4C05	LDR	R4 ,=0x99999999
00000008 25E3	MOVS	R5 ,#0xE3
0000000A 4111	ASRS	R1 ,R1 ,R2 ; register
0000000C 111B	ASRS	R3 ,R3 ,#4 ; immediate
0000000E 41D4	RORS	R4 ,R4 ,R2 ; register
00000010 00ED	LSLS	R5 ,R5 ,#3 ; immediate

- What are the values of R1 – R5 after execution?
- What are the values of the flags?

Shift / Rotate Instructions

■ Examples Shift

- Multiply with 2^n

unsigned / signed

```
LSLS R0,R1,#1 ; *2
LSLS R0,R1,#2 ; *4
LSLS R0,R1,#3 ; *8
```

LSLS for signed and unsigned

- Divide by 2^n

unsigned

```
LSRS R0,R1,#1 ; /2
LSRS R0,R1,#2 ; /4
LSRS R0,R1,#3 ; /8
```

signed

```
ASRS R0,R1,#1 ; /2
ASRS R0,R1,#2 ; /4
ASRS R0,R1,#3 ; /8
```

LSRS and ASRS differ!

Shift / Rotate Instructions

■ Multiplication with Constants using LSLS and ADDS

- Example: Multiplication with 13

- Constant shown as power of 2: $13 = 8 + 4 + 1$
- $R0 = 13 \cdot R1 \rightarrow R0 = (1 + 4 + 8) \cdot R1 = R1 + 4 \cdot R1 + 8 \cdot R1$

```
MOVS  R0, R1      ; R0 = R1
LSLS  R1, R1, #2   ; 4 • R1
ADDS  R0, R0, R1   ; R0 = R0 + 4 • R1
LSLS  R1, R1, #1   ; 2 • R1 -> 8 • R1
ADDS  R0, R0, R1   ; R0 = R0 + 8 • R1
```

Conclusion

■ Logic Instructions

- ANDS, BICS, EORS, MVNS, ORRS

■ Shift/Rotate Instructions

- LSLS, LSRS, ASRS, RORS

Shift / Rotate Instructions

■ Examples

00000000 4904	LDR	R1 ,=0xCCCCCCCC
00000002 2203	MOVS	R2 ,#3
00000004 4B04	LDR	R3 ,=0x66666666
00000006 4C05	LDR	R4 ,=0x99999999
00000008 25E3	MOVS	R5 ,#0xE3

0000000A 4111	ASRS	R1 ,R1 ,R2 ; register
0000000C 111B	ASRS	R3 ,R3 ,#4 ; immediate
0000000E 41D4	RORS	R4 ,R4 ,R2 ; register
00000010 00ED	LSLS	R5 ,R5 ,#3 ; immediate

R1	0xF999'9999		
R2	0x0000'0003		
R3	0x0666'6666		
R4	0x3333'3333		
R5	0x0000'0718		
N = 0	Z = 0	C = 0	V = unknown

Branch Instructions

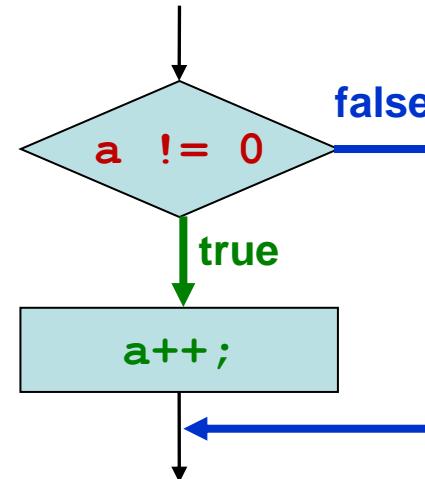
Computer Engineering 1

Motivation

■ Branches may change the PC

- "Non-linear" execution of programs
- Taking decisions

```
uint32_t a;  
...  
if (a != 0) {  
    a++;  
}  
...
```



Agenda

- **Overview Branch Instructions** ¹⁾
- **Unconditional Branches**
 - B → direct, relative
 - BX → indirect, absolute
- **Conditional Branches**
 - Flag dependent branches
 - Arithmetic branches
 - signed vs. unsigned
- **Compare and Test**
 - **CMP** and **CMN**
 - **TST**

Learning Objectives

At the end of this lesson you will be able

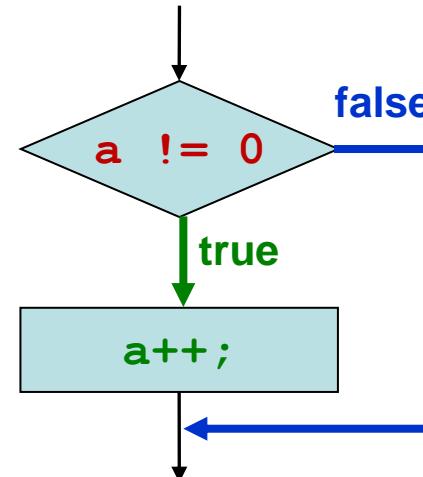
- to explain what branch instructions are and how they work
- to classify a given branch instruction with regard to
 - conditional / unconditional
 - relative / absolute
 - direct / indirect
- to apply and discuss the different branch instructions
- to determine based on the settings of the flags whether a conditional branch is taken or not
- to distinguish, apply and explain the instructions **CMP**, **CMN** and **TEST**

Motivation

■ Branches may change the PC

- "Non-linear" execution of programs
- Taking decisions

```
uint32_t a;  
...  
if (a != 0) {  
    a++;  
}  
...
```

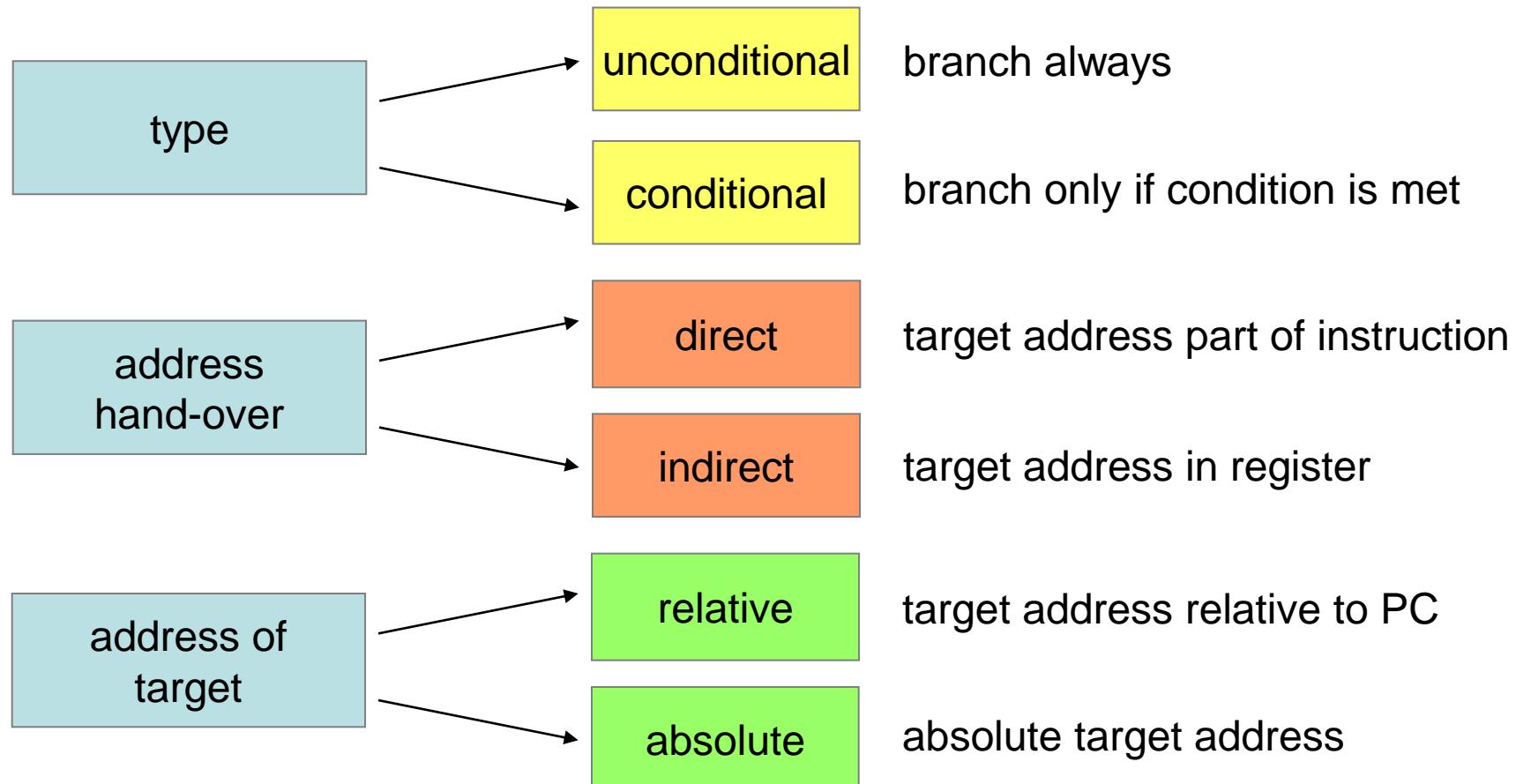


modify PC based
on decision

0x20000030 3800	SUBS	r0,#0
0x20000032 d000	BEQ	end_if
0x20000034 1c40	ADDS	r0,r0,#1
0x20000036 ...	end_if	...
0x20000038		

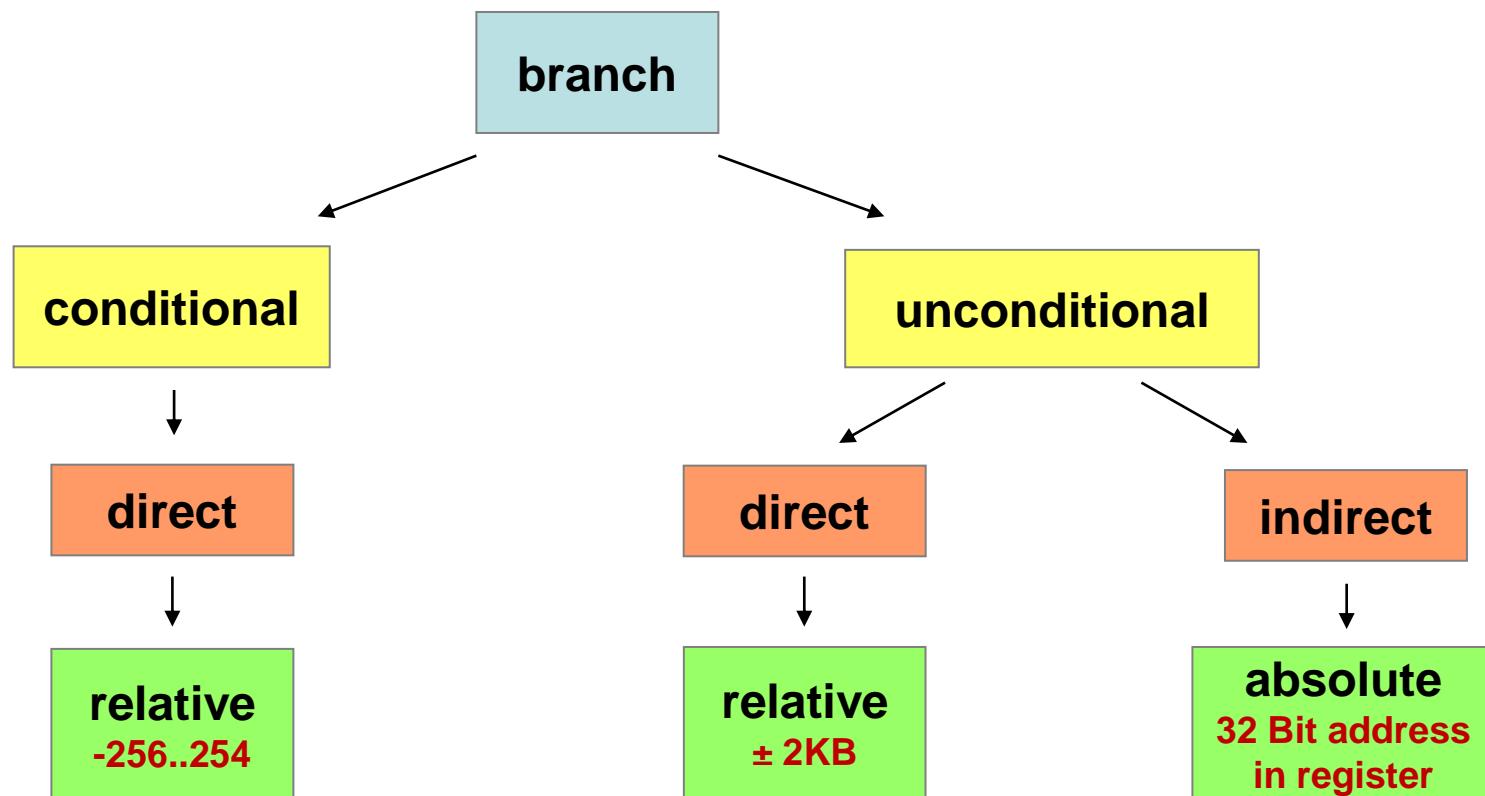
Overview Branch Instructions

■ Properties



Overview Branch Instructions

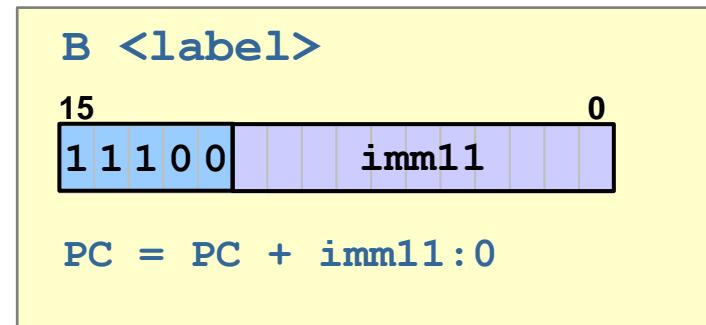
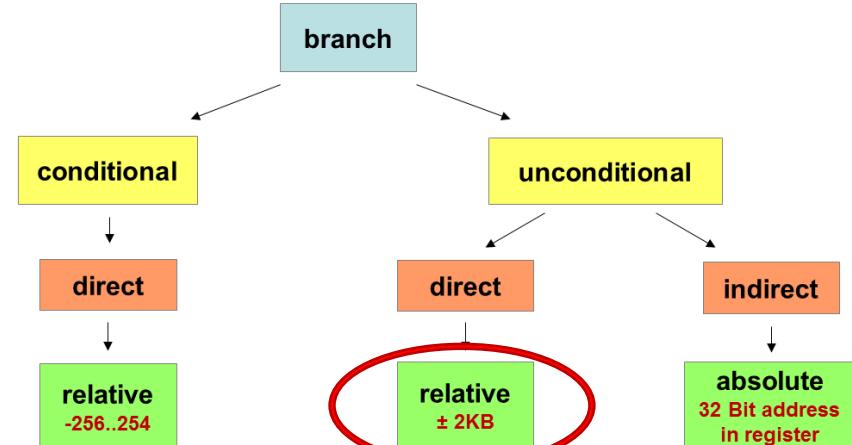
■ Overview ARMv6-M (Cortex-M0)



Unconditional Branches

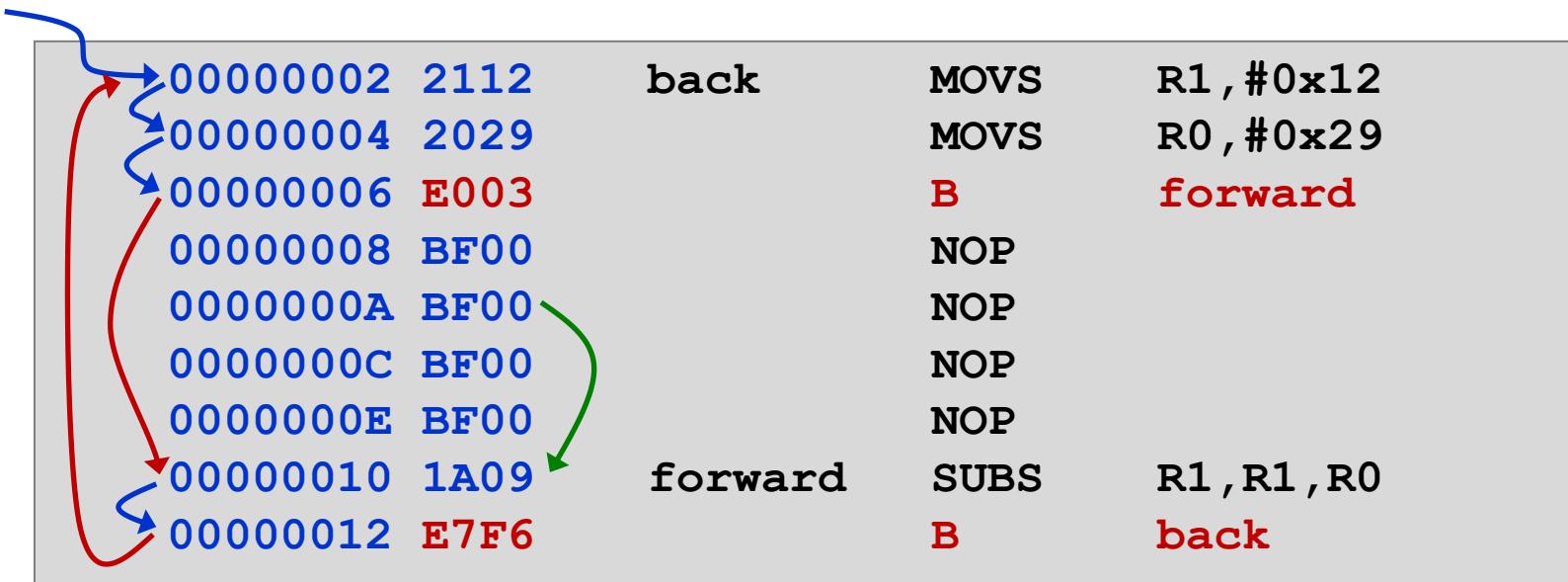
■ B (immediate)

- Unconditional
- Direct
- Relative (to PC)
 - imm11:0
 - Offsets from -2048d to +2046d



Unconditional Branches

■ Direct, relative branch → B label



Forward

$$\begin{aligned} & 0x0000'0006 \quad \text{Address of current instruction} \\ & + 0x0000'0004 \quad ^1) \\ & + \underline{0x0000'0006} \quad (0x003 \ll 1) = 0x006 \\ & \underline{0x0000'0010} \end{aligned}$$

Back

$$\begin{aligned} & 0x0000'0012 \quad \text{Address of current instruction} \\ & + 0x0000'0004 \quad ^1) \\ & + \underline{0xFFFF'FFEC} \quad (0x7F6 \ll 1) = 0xFEC \\ & \underline{0x0000'0002} \end{aligned}$$

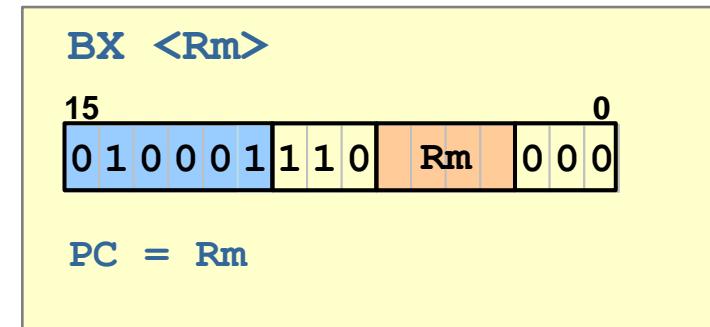
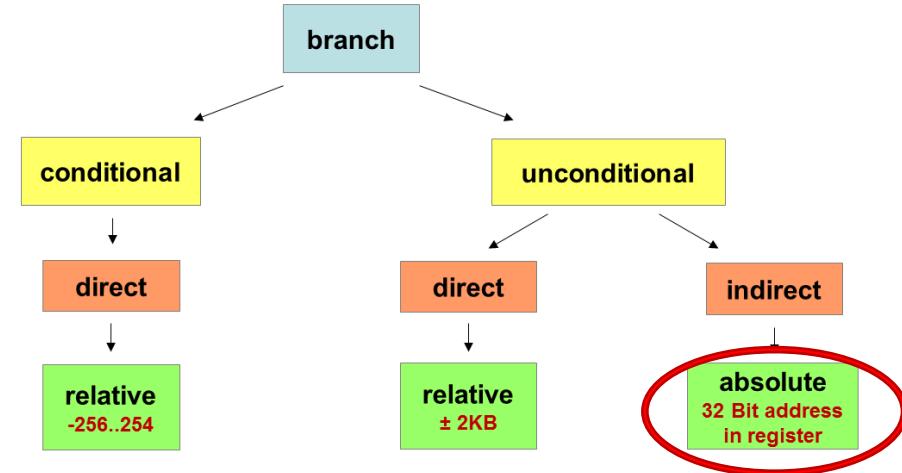
= -20d
sign-extended

¹⁾ Because of pipeline, PC is always **current address plus 4**

Unconditional Branches

BX

- Branch and Exchange
 - Register Rm holds target address
 - Unconditional
 - Indirect
 - Absolute



Unconditional Branches

■ Indirect, absolute branch → BX R0

00000014	4802	LDR	R0, =jmpaddr
00000016	4700	BX	R0
00000018	BF00	NOP	
0000001A	BF00	NOP	
0000001C	3013	jmpaddr ADDS	R0, R0, #0x13
0000001E	BF00		NOP

- jmpaddr = 0x0000'001C → R0
- R0 → PC

Conditional Branches

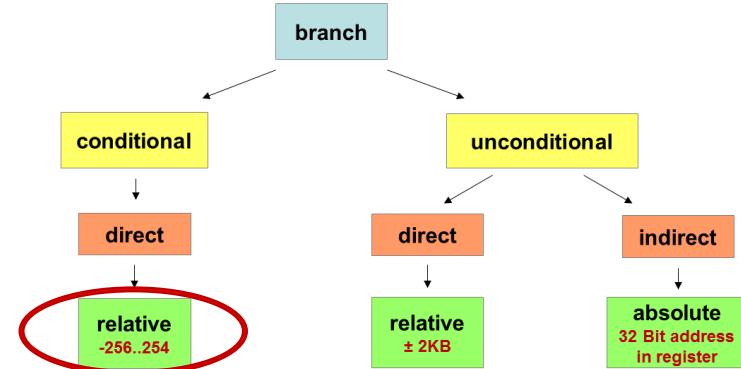
■ Flag-dependent branches

- Based on **one specific flag**

■ Arithmetic branches

- Based on one or more flags
 - e.g. after an arithmetic instruction
- **unsigned** operations
 - higher and lower
- **signed** operations
 - greater and less

■ Conditional branches are always **relative** on ARM



Conditional Branches

■ Flag-dependent

Symbol	Condition	Flag
EQ	Equal	$Z == 1$
NE	Not equal	$Z == 0$
CS	Carry set	$C == 1$
CC	Carry clear	$C == 0$
MI	Minus/negative	$N == 1$
PL	Plus/positive or zero	$N == 0$
VS	Overflow	$V == 1$
VC	No overflow	$V == 0$

source: Joseph Yiu: *The definite Guide to the ARM Cortex M3*, Page 63

Conditional Branches

■ Arithmetic - unsigned

- higher and lower

Symbol	Condition	Flag
EQ	Equal	$Z == 1$
NE	Not equal	$Z == 0$
HS (=CS)	Unsigned higher or same	$C == 1$
LO (=CC)	Unsigned lower	$C == 0$
HI	Unsigned higher	$C == 1$ and $Z == 0$
LS	Unsigned lower or same	$C == 0$ or $Z == 1$

source: Joseph Yiu: *The definite Guide to the ARM Cortex M3*, Page 63

Conditional Branches

■ Arithmetic - signed

- greater and less

Symbol	Condition	Flag
EQ	Equal	$Z == 1$
NE	Not equal	$Z == 0$
MI	Minus/negative	$N == 1$
PL	Plus/positive or zero	$N == 0$
VS	Overflow	$V == 1$
VC	No overflow	$V == 0$
GE	Signed greater than or equal	$N == V$
LT	Signed less than	$N != V$
GT	Signed greater than	$Z == 0 \text{ and } N == V$
LE	Signed less than or equal	$Z == 1 \text{ or } N != V$

source: Joseph Yiu: *The definite Guide to the ARM Cortex M3*, Page 63

Conditional Branches

■ Opcodes

- imm8:0
 - Offset from -256d to +254d

cond	short	Flag
0000	EQ	Z == 1
0001	NE	Z == 0
0010	CS/HS	C == 1
0011	CC/LO	C == 0
0100	MI	N == 1
0101	PL	N == 0
0110	VS	V == 1
0111	VC	V == 0

cond	short	Flag
1000	HI	C == 1 and Z == 0
1001	LS	C == 0 or Z == 1
1010	GE	N == V
1011	LT	N != V
1100	GT	Z == 0 and N == V
1101	LE	Z == 1 or N != V
1110	AL	always
1111	--	--

B<c> <label>

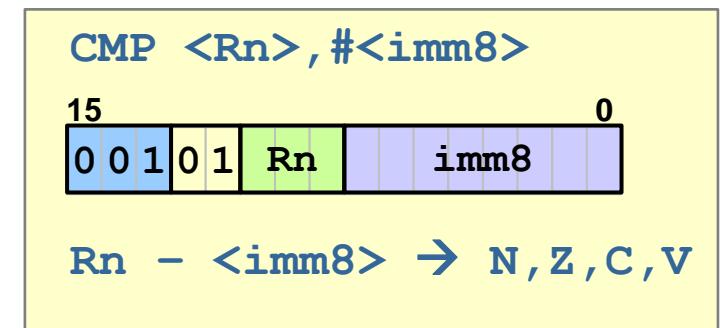
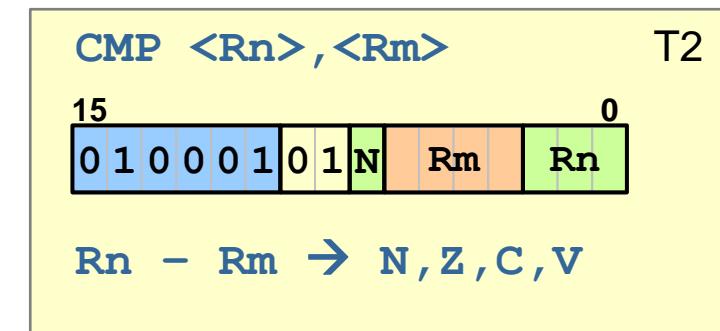
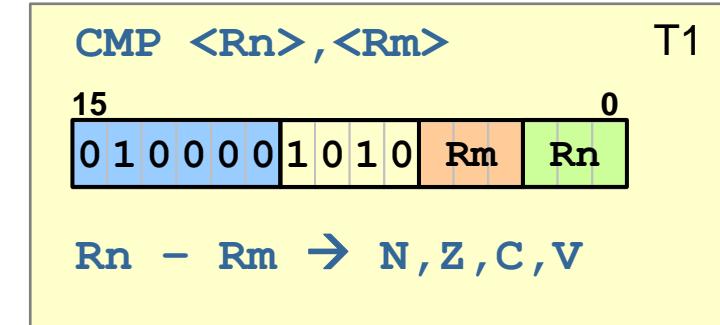


```
if (cond) then  
    PC = PC + imm8:0
```

Compare and Test

■ CMP

- Same as **SUBS**, but without storing a result!
- Compare 2 operands
 - Higher/lower?
 - Greater/less?
 - Equal?
- Only flags are affected!
- Registers unchanged
- T2 also higher registers



Compare and Test

■ CMP

- **CMP** does not change registers
- Example



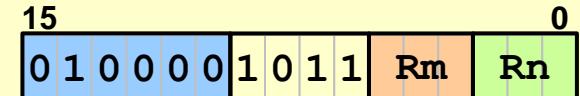
00000000	4288	CMP	R0,R1	; R0 > R1 ?
00000002	D802	BHI	go_on	; if higher -> go_on
00000004	000A	MOVS	R2,R1	; otherwise exchange regs
00000006	0001	MOVS	R1,R0	
00000008	0010	MOVS	R0,R2	
0000000A	2305	go_on	MOVS	R3,#5
0000000C

Compare and Test

■ CMN

- Same as **ADDS**, but without storing result!
- Compare 2 operands negative
- Only flags are affected!
- Registers unchanged
- Read **CMN** as
 - Is content of Rm equal to 2's complement of Rn?

CMN <Rn>, <Rm>



Rn + Rm → N, Z, C, V

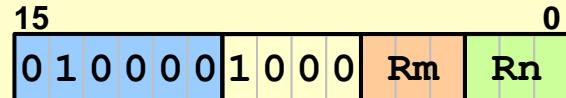
00000000 42C8	CMN	R0,R1 ; R0 equal minus R1?
00000002 D002	BEQ	next

Compare and Test

■ TST

- Is a specific bit set?
- Logical AND without storing result
- Registers unchanged
- Changes only flags N and Z
 - C and V unchanged

TST <Rn>, <Rm>



Rn & Rm → N, Z

```
SWITCH_ADDRESS      EQU    0x60000200
S3_MASK            EQU    0x00000008
00000000 4903      LDR    R1,=SWITCH_ADDRESS
00000002 6808      LDR    R0,[R1]           ; read switch data
00000004 4A03      LDR    R2,=S3_MASK
00000006 4210      TST    R0,R2             ; bit S3 = 1 ?
00000008 D101      BNE    s3_equal_one     ; branch if Z = 0
0000000A ...      s3_equal_zero
                                ...
...                  ...
...                  ...      s3_equal_one
                                ...
```

Exercise

■ Which branches are taken?

Instruction	Z	C	N	V	“Taken” / “Not Taken”?
BNE label	1	0	0	0	
BLO label	0	0	0	0	
BHI label	0	1	0	0	
BLT label	0	0	1	1	
BLE label	1	0	1	1	

“Taken”

“Not Taken”

Execution is continued at the indicated label

Execution is continued at the instruction following the branch instruction

■ Branch Instructions Change PC

- „Decision Making“ and Control Flow

■ Branch Instructions

- unconditional, relative, direct **B** $PC = PC \pm 2KB$
- unconditional, absolute, indirect **BX** $PC = Rm$
- conditional, relative, direct **Bxx** $PC = PC-256; PC+254$

■ Compare and Test

- **CMP**, **CMN** → **SUBS**, **ADDS** without result, but flags are set!
- **TST** → **AND** without result, but flags are set!

Structured Programming – Control Structures

Computer Engineering 1

Motivation



Spaghetti code

From Wikipedia, the free encyclopedia.

Spaghetti code is a pejorative term for code with a complex and tangled control structure, especially one using many [GOTOs](#), exceptions, or other "unstructured" branching constructs. It is named after [spaghetti](#) because a diagram of program flow tends to look like that. Nowadays it is preferable to use so-called [structured programming](#).

Also called [kangaroo code](#) because such code has so many jumps in it.

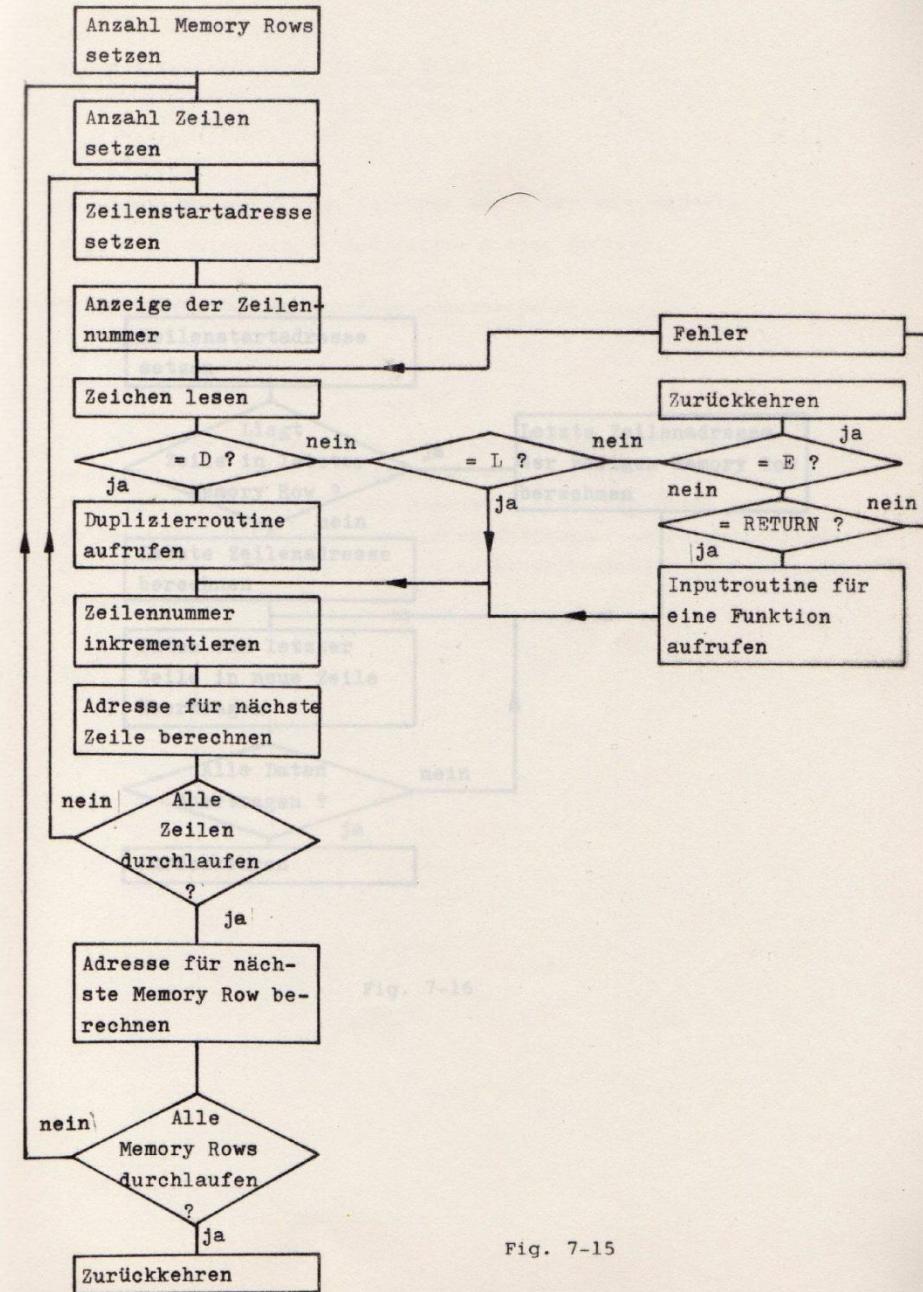


Fig. 7-15

Agenda

- **Structured Programming**
- **Selection**
 - if – then - else
- **Loops**
 - Do – While
 - While
 - For
- **Switch Statements**

Learning Objectives

At the end of this lesson you will be able

- to explain the basic concepts of structured programming
- to enumerate and explain the basic elements of a structogram
- to comprehend how a C-compiler implements control structures in assembly language
 - if-then-else
 - do-while loops
 - while loops
 - for loops
 - switch statements
- to program basic structograms in assembly language

Why Structured Programming ?

■ Rules for the structure of a program

- Patterns for control structures
 - Sequence
 - Selection if - then - else
 - Iteration / Loop for, while, do - while
- Compilers generate code-blocks based on these patterns

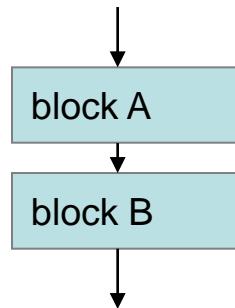
■ Supports program development

- Clarity
- Documentation
- Maintenance
- Allows to program on a higher level of abstraction

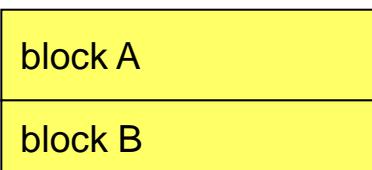
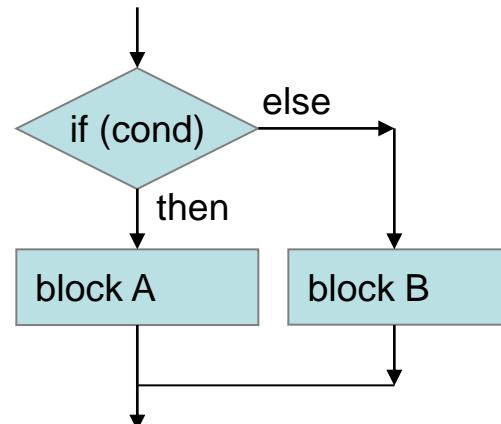
Structured Programming

- Program flow can be represented with three elements

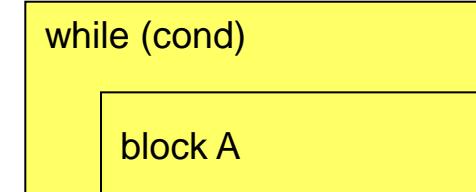
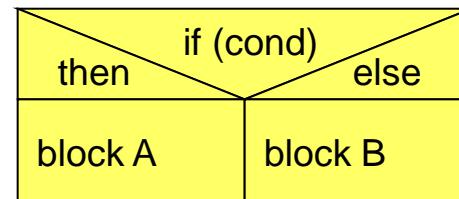
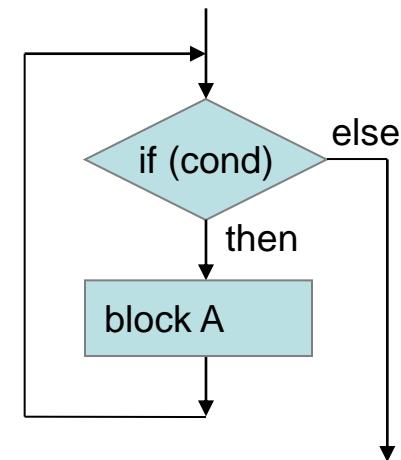
Sequence



Selection



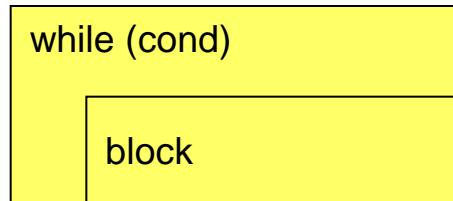
Iteration / loop



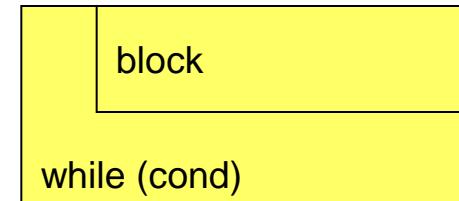
Further Structograms

■ Iteration

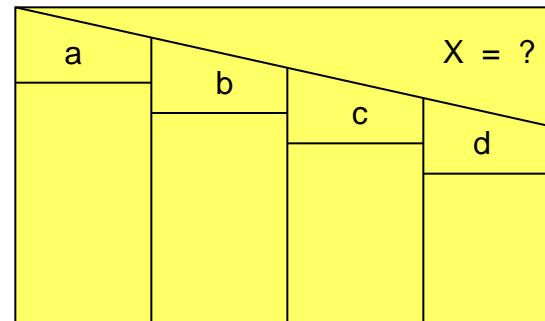
pre-test loop



post-test loop



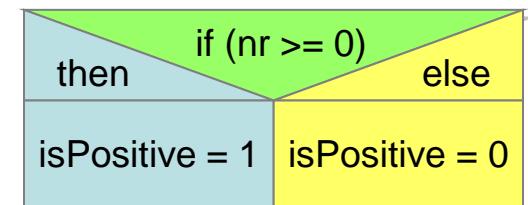
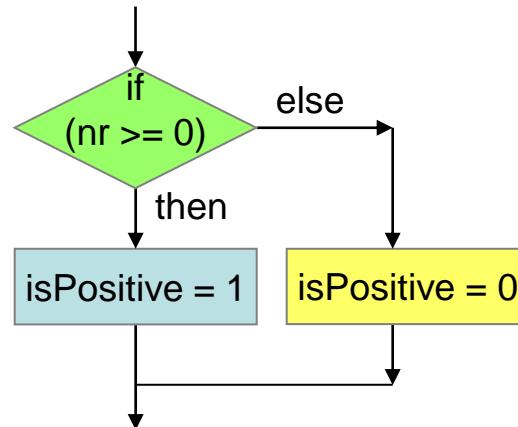
■ Switch statement (case)



Selection

■ if(...) – then - else

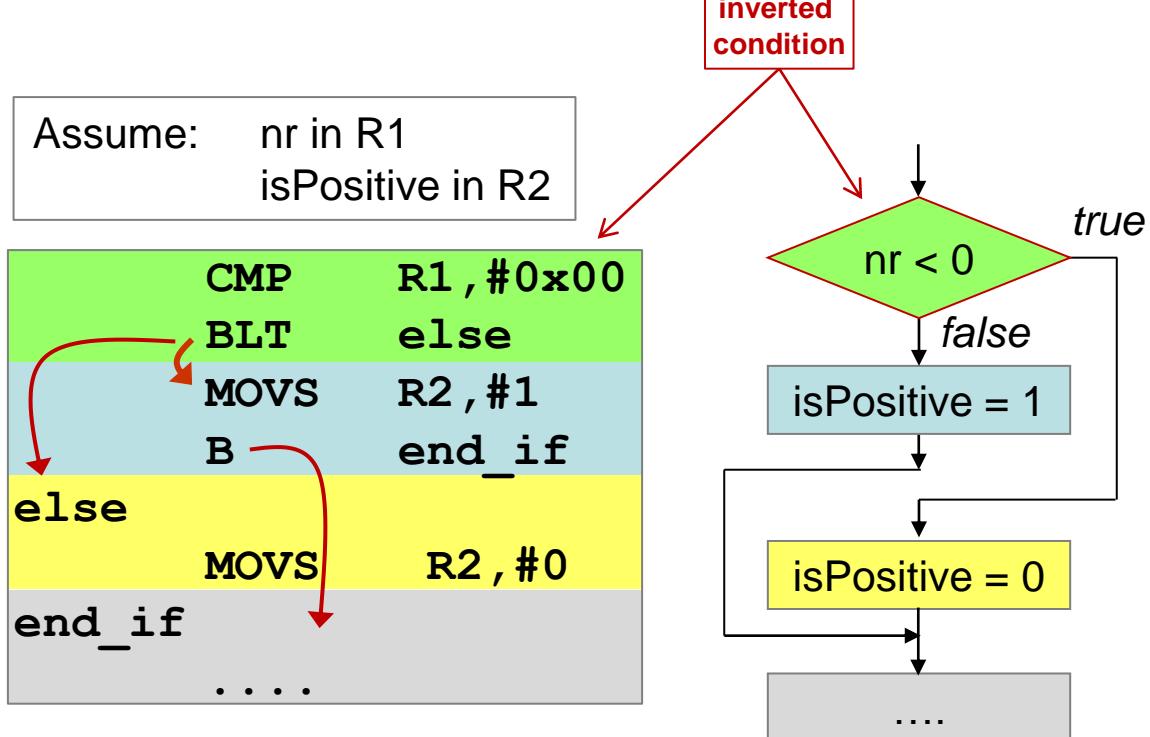
```
int32_t nr, isPositive;  
...  
if (nr >= 0) {  
    isPositive = 1;  
}  
else {  
    isPositive = 0;  
}
```



Selection: if – then – else

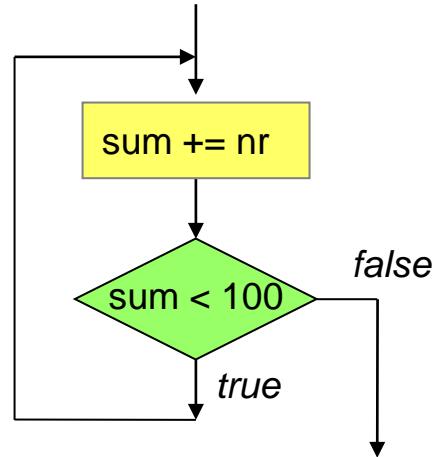
- Compiler translates *selection* into assembly code
 - uses conditional and unconditional jumps

```
int32_t nr;
int32_t isPositive;
...
if (nr >= 0) {
    isPositive = 1;
}
else {
    isPositive = 0;
}
```

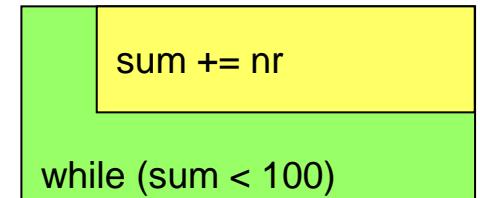


Loops: Do-While Loops

```
int32_t nr;  
int32_t sum;  
...  
sum = 0;  
do {  
    sum += nr;  
} while (sum < 100);
```



post-test loop



Loops: Do-While Loops

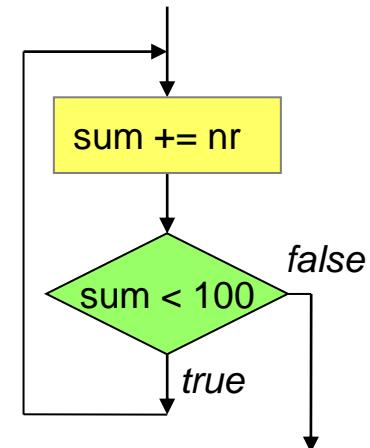
- Compiler translates *post-test loop* to assembly code

```
int32_t nr;  
int32_t sum;  
  
...  
sum = 0;  
do {  
    sum += nr;  
} while (sum < 100);
```

Assume: nr in R1
sum in R2

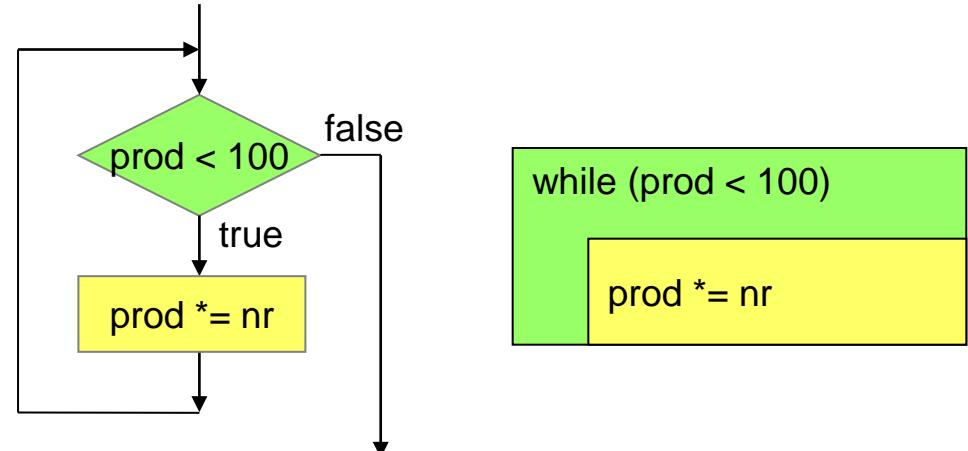


```
MOVS  R2 ,#0  
loop  ADDS  R2 ,R2 ,R1  
      CMP   R2 ,#100  
      BLT   loop  
      ...
```



Loops: While Loops

```
int32_t nr;  
int32_t prod;  
...  
prod = 1;  
while (prod < 100) {  
    prod *= nr;  
}
```

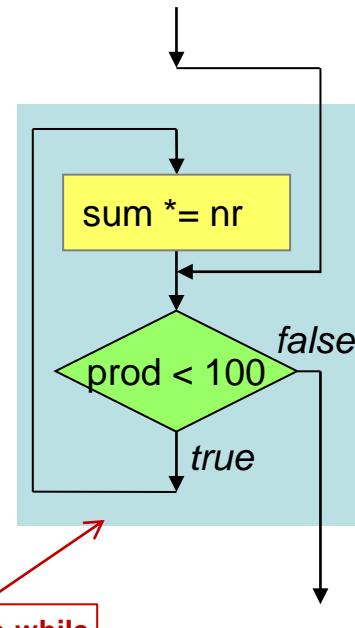
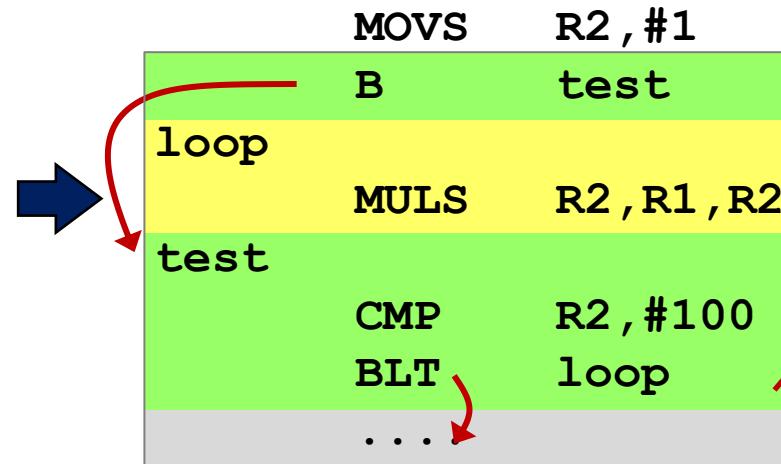


Loops: While Loops

- Compiler translates *pre-test loop* to assembly code
 - Re-using structure of do-while (pre-test loop)

```
int32_t nr;  
int32_t prod;  
...  
prod = 1;  
while (prod < 100) {  
    prod *= nr;  
}
```

Assume: nr in R1
prod in R2



Loops: For Loops

- For Loops are converted into While Loops
 - break/continue statements require special treatment

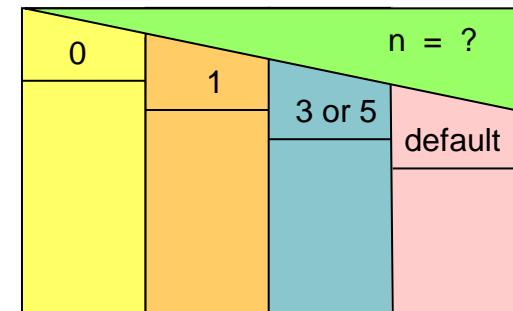
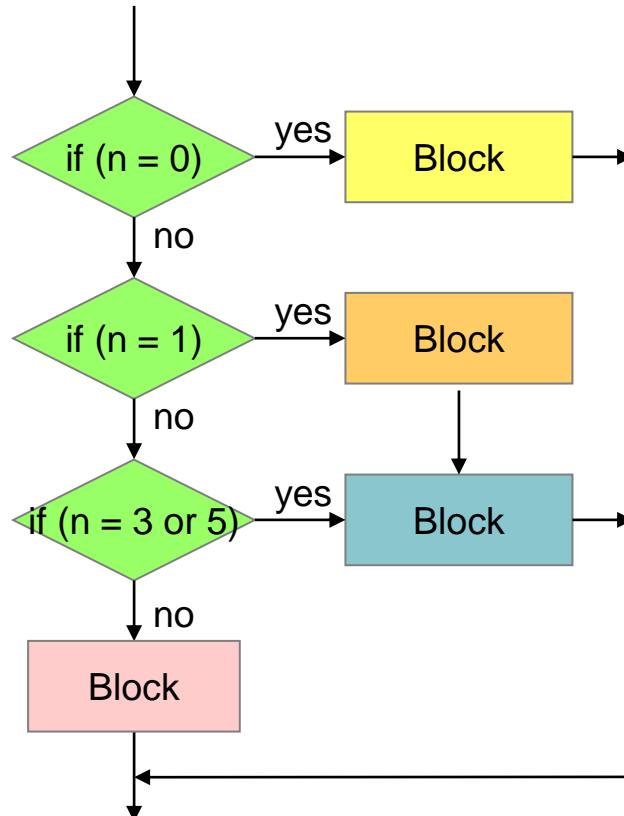
```
for (init-expr; test-expr; update-expr)
    body-block
```



```
init-expr;
while (test-expr) {
    body-block
    update-expr;
}
```

Switch Statements

```
uint32_t result, n;  
  
switch (n) {  
    case 0:  
        result += 17;  
        break;  
    case 1:  
        result += 13;  
        //fall through  
    case 3: case 5:  
        result += 37;  
        break;  
    default:  
        result = 0;  
}
```



Structogram without fall-through

Switch Statements

■ Jump Table

```
uint32_t result, n;  
switch (n) {  
    case 0:  
        result += 17;  
        break;  
    case 1:  
        result += 13;  
        //fall through  
    case 3: case 5:  
        result += 37;  
        break;  
    default:  
        result = 0;  
}
```

Assume: n in R1
result in R2

NR_CASES EQU 6

case_switch CMP R1, #NR_CASES
BHS case_default
LSLS R1, #2 ; * 4
LDR R7, =jump_table
LDR R7, [R7, R1]
BX R7

case_0 ADDS R2, R2, #17
B end_sw_case
case_1 ADDS R2, R2, #13
case_3_5 ADDS R2, R2, #37
B end_sw_case
case_default MOVS R2, #0
end_sw_case ...

jump_table DCD case_0
DCD case_1
DCD case_default
DCD case_3_5
DCD case_default
DCD case_3_5

Limitations of Conditional Branches

■ Limited range of -256..254 Bytes

- Example

```
IF           CMP      R1, R0
             BNE      ELSE
             THEN     <some code>
             B       IF_END
             ELSE     <some other code>
             IF_END
```

Simple code for the case when
<some code> is short

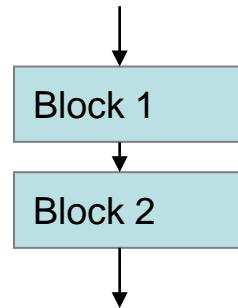
```
IF           CMP      R1, R0
             BEQ      THEN
             B       ELSE
             THEN     <some code>
             B       IF_END
             ELSE     <some other code>
             IF_END
```

Code requires additional branch in
case when <some code> is too long

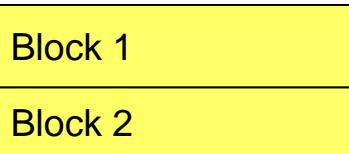
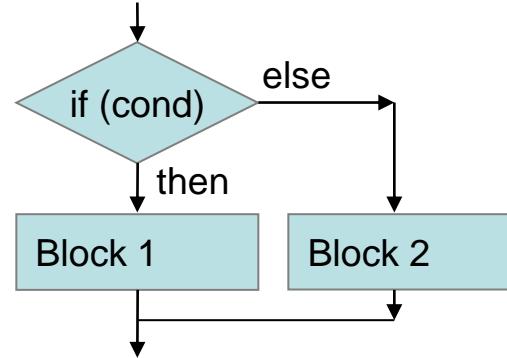
Conclusion

- Program flow can be represented with three elements

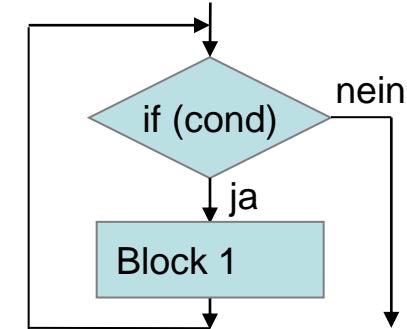
Sequence



Selection



Iteration/loop



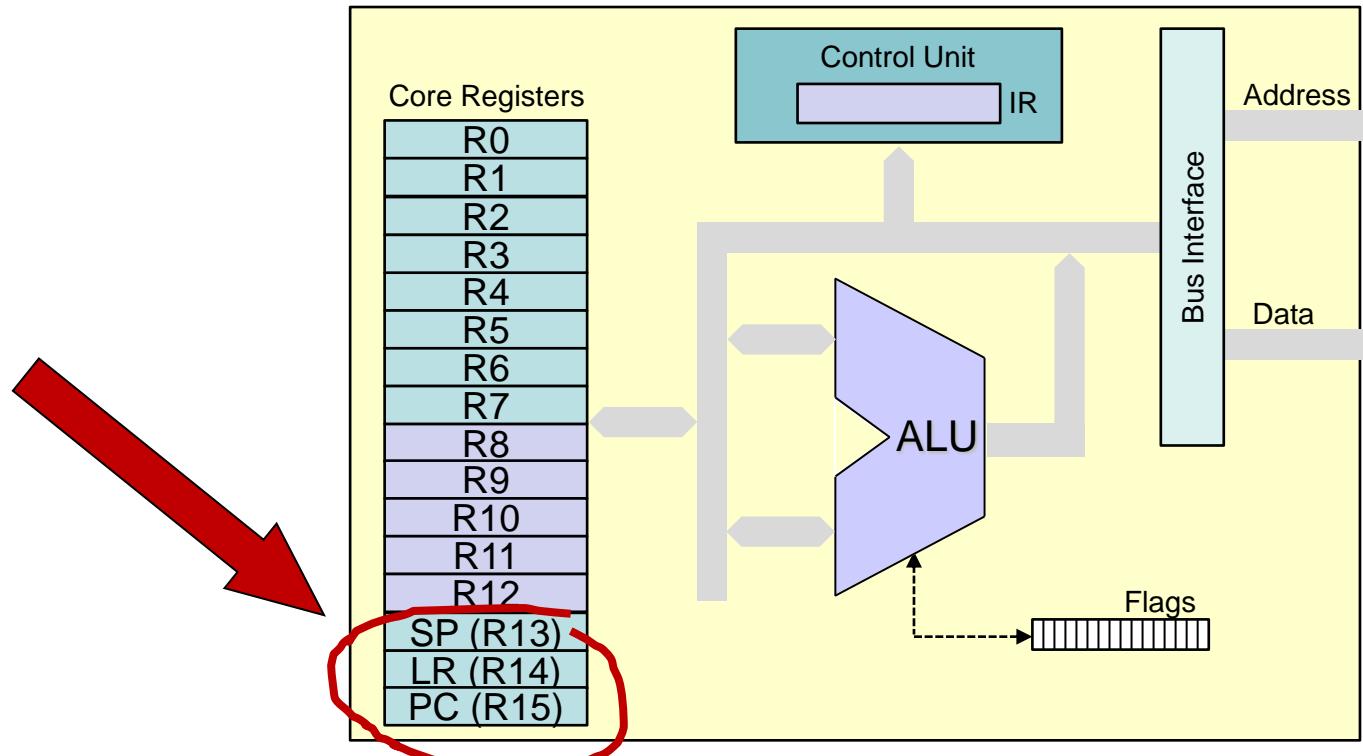
- High level programming language provides these control structures
- Compiler translates control structures to assembly using conditional and unconditional jumps

Subroutines and Stack

Computer Engineering 1

Motivation

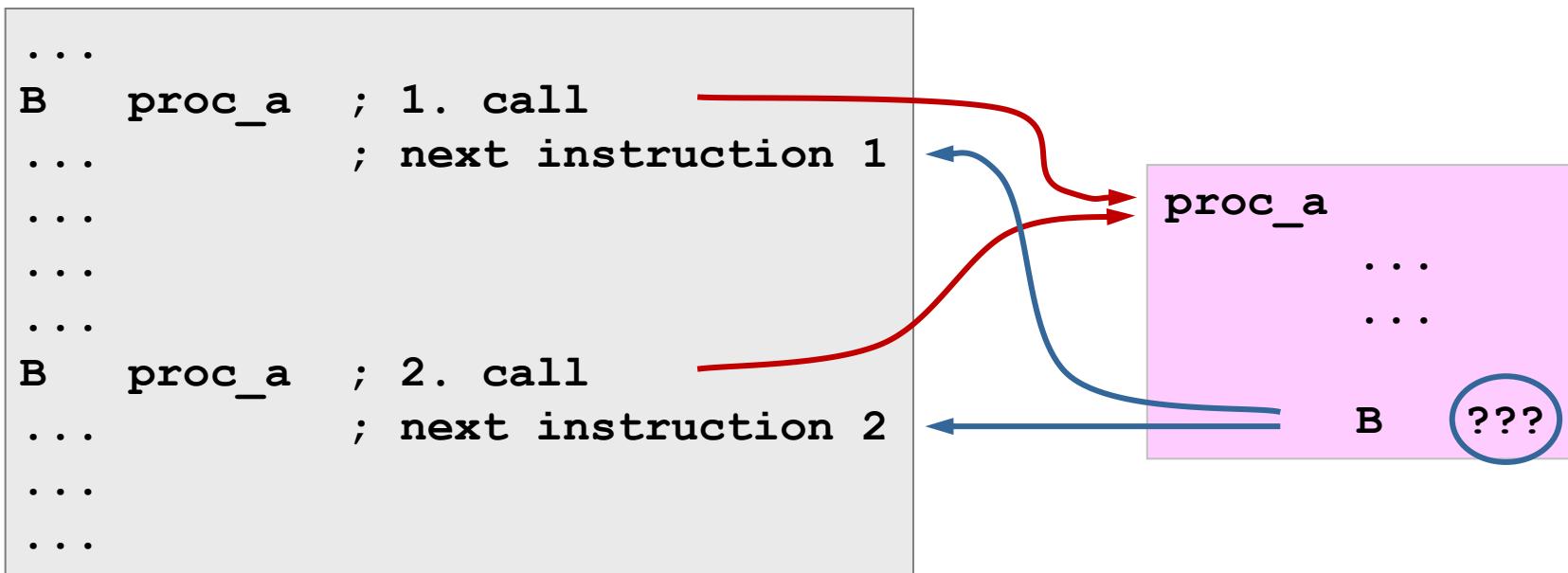
■ Do you remember?



Motivation

main program

subroutine



Agenda

- **Terminology**
- **Subroutine Call and Return**
- **Nested Subroutine Calls**
- **Stack**
- **ARM: PUSH and POP**
- **Nested Subroutines (revisited)**
- **Instructions using SP**
- **Assembler Directives**

Learning Objectives

At the end of this lesson you will be able

- to explain and discuss the term subroutine
- to comprehend and explain how a subroutine call and return are implemented on ARM Cortex-M
- to implement (nested) subroutines in assembly
- to explain how a processor stack works
- to determine the content of the stack for a given assembly program with nested subroutine calls

Terminology

■ Subroutine / Procedure / Function / Method

- Sequence of instructions to solve a subtask
- Called by "name"
- Interface and functionality known
- Internal design and implementation are hidden
 - information hiding
- Can be called from miscellaneous places in the program

■ Why Subroutines?

- Basic element of structured programming
- Reuse of the same implementation → less mistakes
- Simplifies verification and maintenance
- Requires less memory
 - only one instance for several calls

■ Terms used by ARM

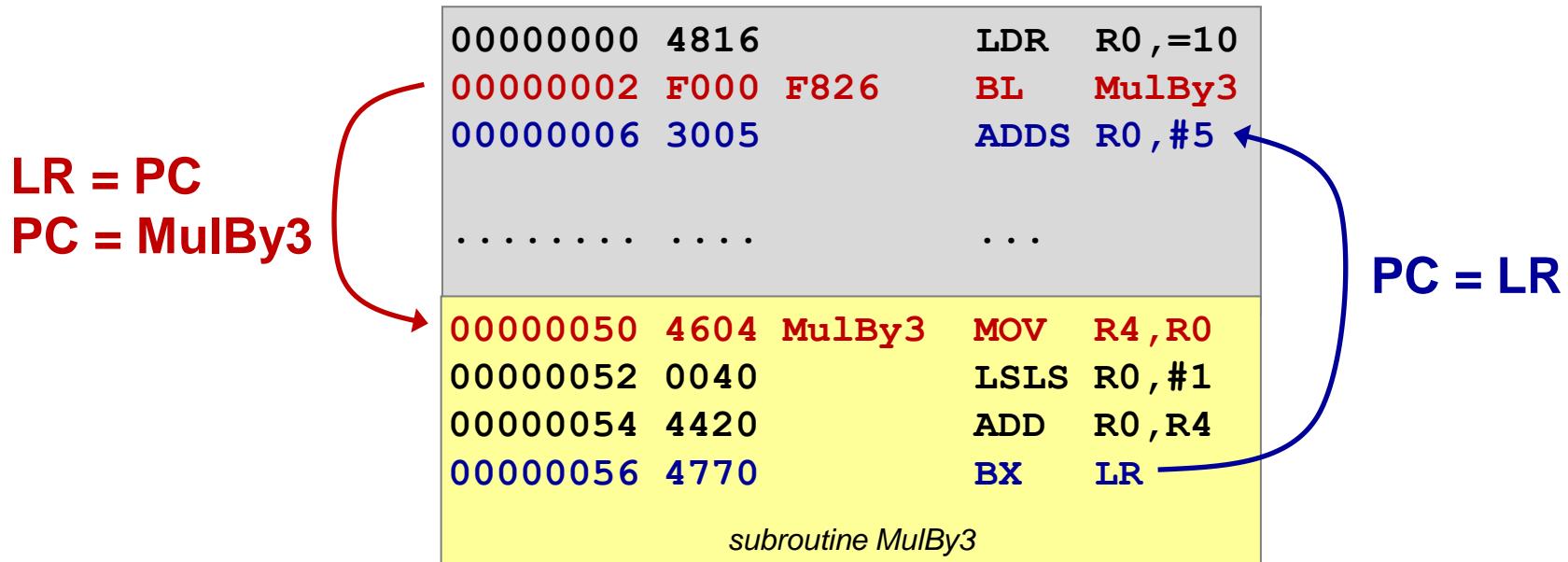
- Routine, subroutine
 - A fragment of program to which control can be transferred that, on completing its task, returns control to its caller at an instruction following the call. *Routine* is used for clarity where there are nested calls: a routine is the *caller* and a subroutine is the *callee*.
- Procedure
 - A routine that returns no result value.
- Function
 - A routine that returns a result value.

source: *Procedure Call Standard for the ARM Architecture*
ARM IHI 0042E, 30th November 2012

Subroutine Call and Return

■ Change of control flow

- Call Save PC to Link Register (LR)
- Return Restore PC from LR



Subroutine Call and Return

■ Structure of Subroutine

- Label with Name
 - e.g. **MulBy3**
- Return Statement
 - **BX LR**

```
00000050 4604 MulBy3    MOV   R4 ,R0
00000052 0040                LSLS  R0 ,#1
00000054 4420                ADD   R0 ,R4
00000056 4770                BX    LR
```

Subroutine Call and Return

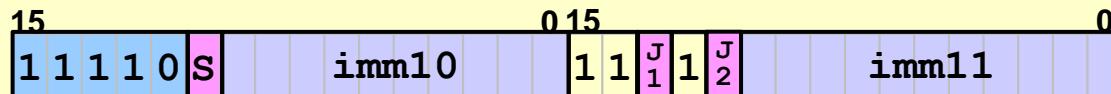
■ BL <label>

- Store current PC in LR
- Branch to <label>
 - $PC = PC +/- \text{offset}$
 - offset range -16'777'216 to 16'777'214

Subroutine call through a label

- unconditional
- **relative**
- direct

BL <label>



```
I1 = NOT(J1 EOR S); I2 = NOT (J2 EOR S)
<imm> = S:I1:I2:imm10:imm11:0
LR = PC (LSB set to '1')
PC = PC + <imm>
```

Subroutine Call and Return

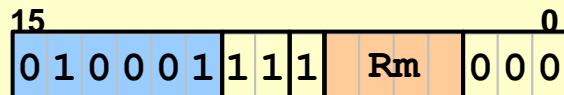
■ BLX (register)

- Store current PC in LR
- Address of subroutine in register
- Branch
 - PC = register
 - Branch address from 0 to 2^{32}

Subroutine call with address in register

- unconditional
- absolute
- indirect

BLX <Rm>



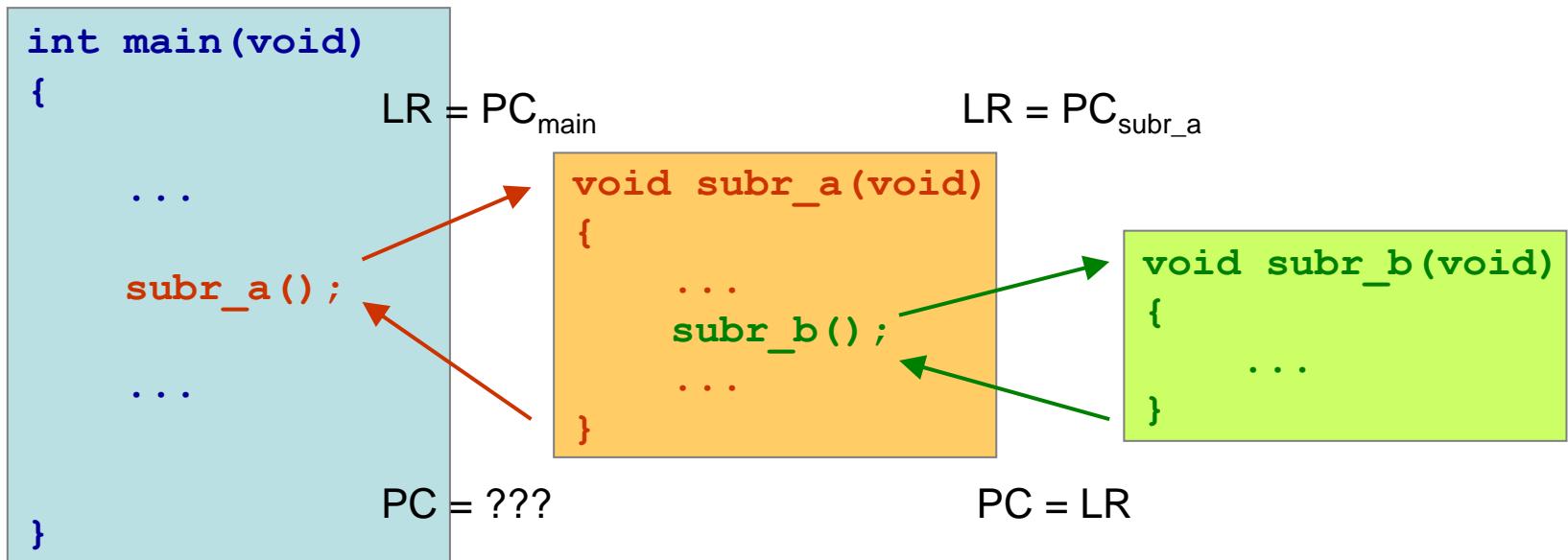
LR = PC - 2 (LSB set to '1')

PC = Rm

Nested Subroutine Calls

■ Nested Subroutine (Function) Calls

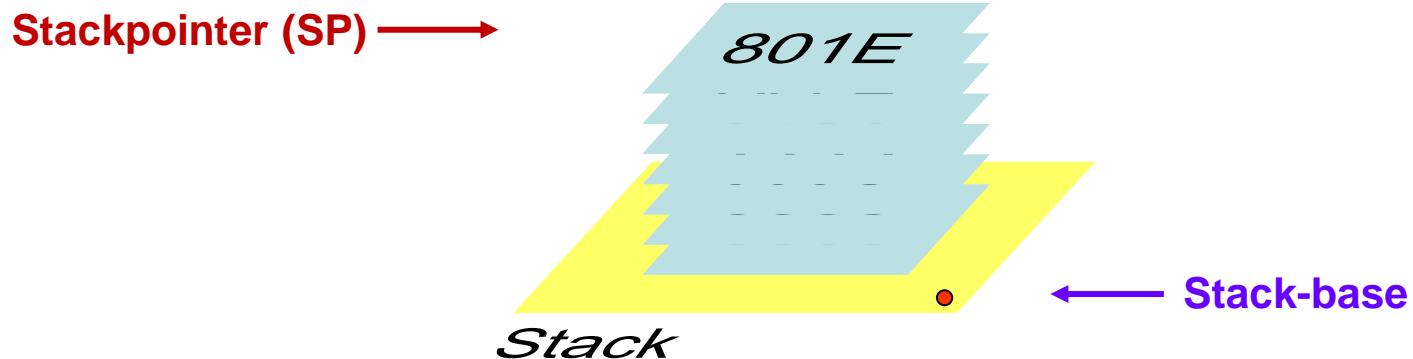
- How do we do that with a single LR?



Stack

■ Stack as Object

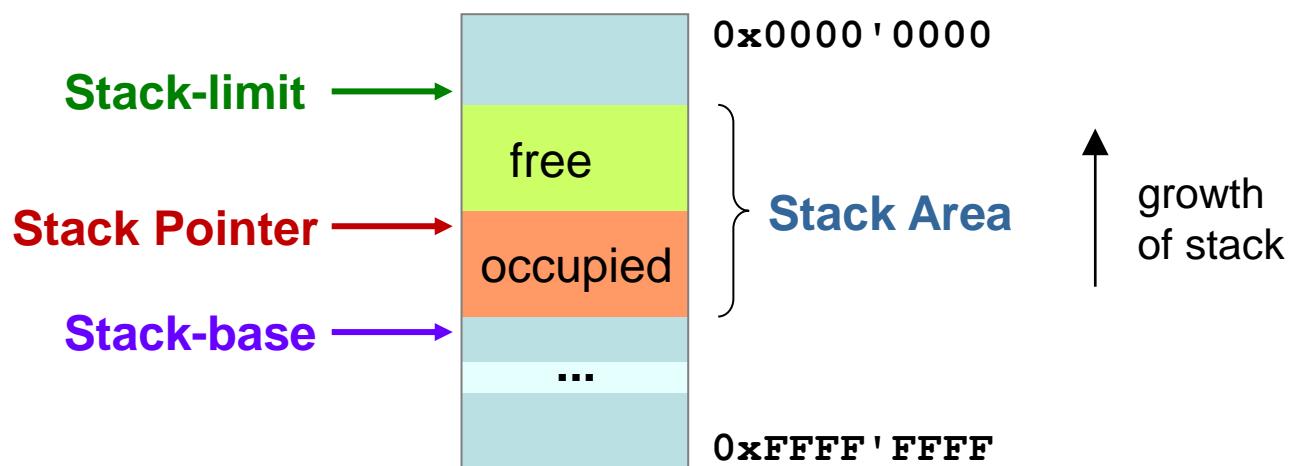
- Methods
 - `PUSH()` and `POP()`
- Data
 - pushed (written) on top of the stack
 - popped (fetched, read) from the top of the stack → LIFO¹⁾



¹⁾ Last In First Out

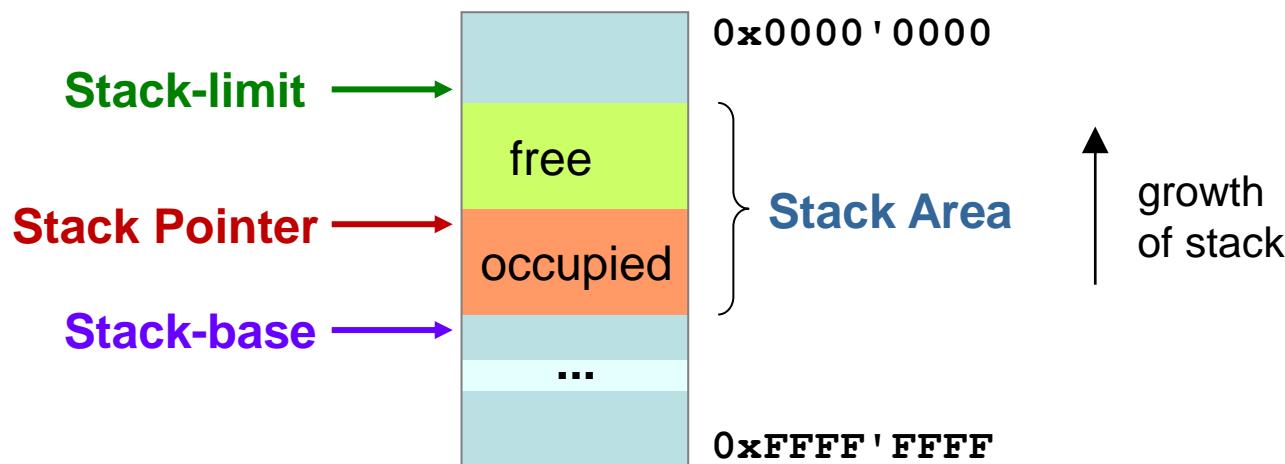
■ Implementation

- **Stack Area (Section)** Continuous area of RAM
- **Stack Pointer (SP)** R13 → points to last written data value
- **PUSH { ... }** Decrement SP and store word(s)
- **POP { ... }** Read word(s) and increment SP
- Direction on ARM "grows" from higher towards lower addresses → full-descending stack
- Alignment Stack operations are word-aligned



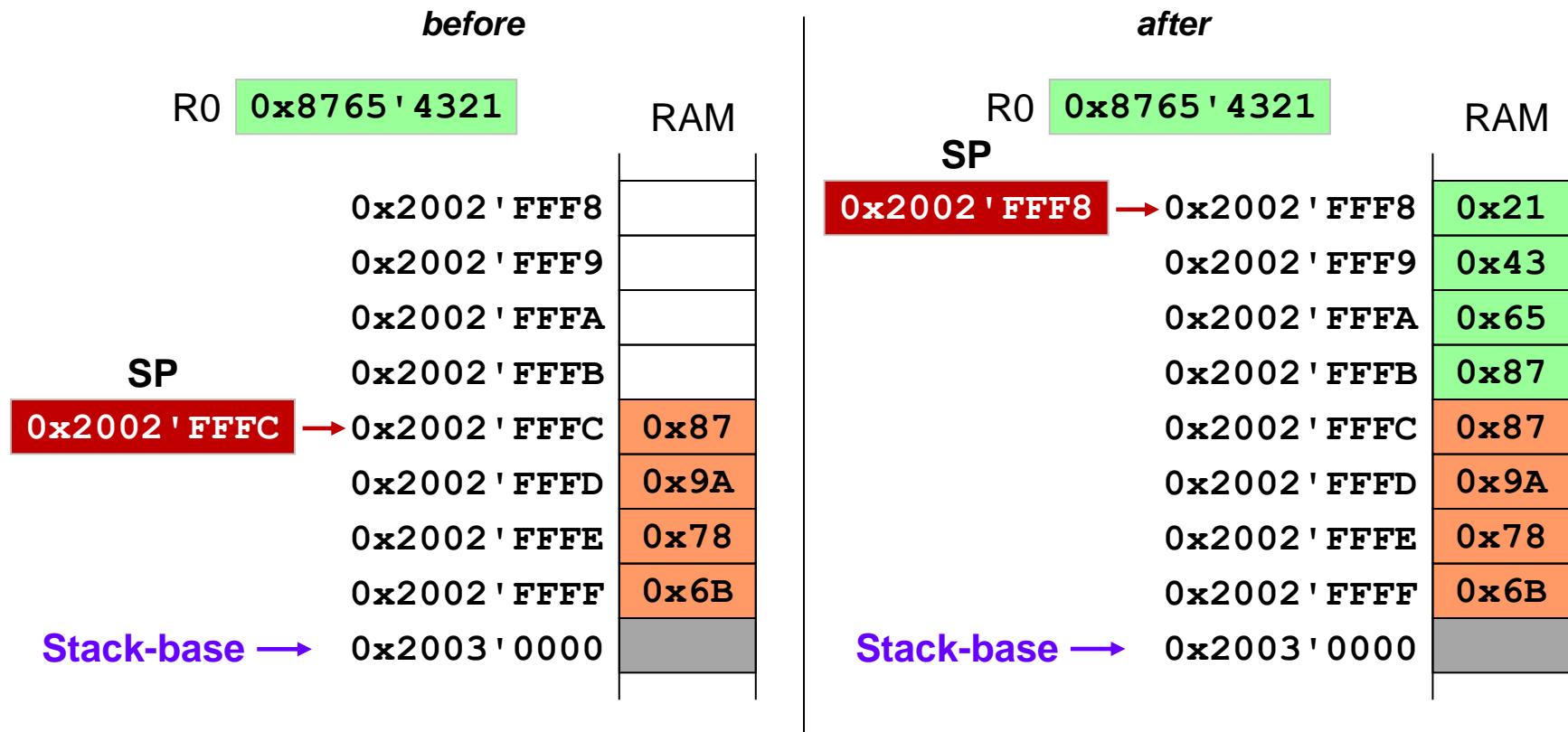
■ Initialization

- Processor fetches initial value of SP (**Stack-base**) at reset
 - from address 0x0000'0000
- **Stack-base** is right above the stack area
 - SP is decremented before writing the first word



ARM: PUSH and POP

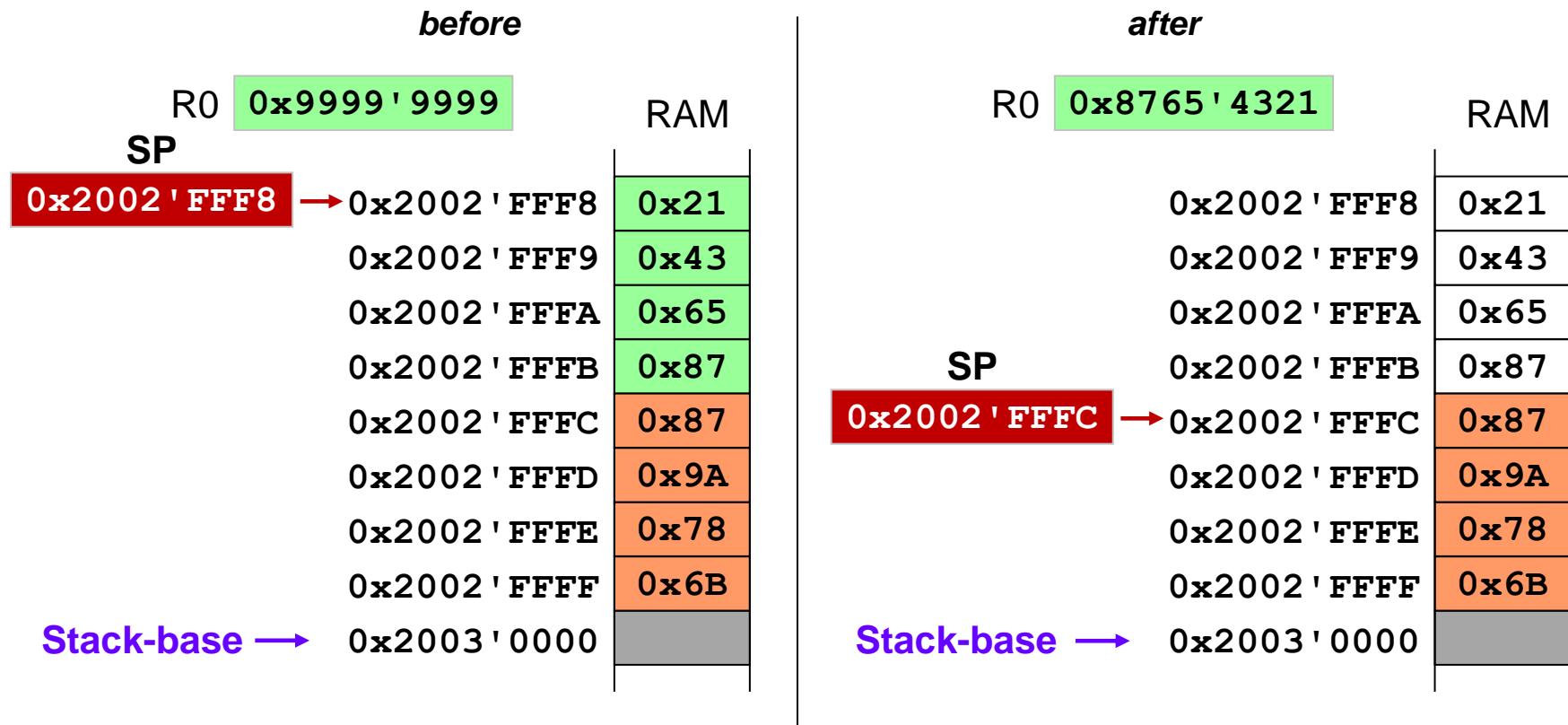
■ Example: PUSH {R0}



SP points to last value that has been written

ARM: PUSH and POP

■ Example: POP {R0}

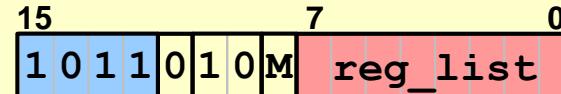


ARM: PUSH and POP

■ PUSH

- **registers**
 - One or more registers to be stored
 - Low registers → `reg_list` = one bit per register
 - LR (R14) → M-bit → No other high registers
 - Lowest register stored first (lowest address)

PUSH {registers}



```
addr = SP - 4*BitCount(M::reg_list)

for i = 0 to 7
    if reg_list<i> == '1' then
        Mem[addr,4] = R[i]
        addr = addr + 4

    if (M == '1') then
        Mem[addr] = LR

SP = SP - 4*BitCount(M::reg_list)
```

00000000 B480	PUSH	{R7}
00000002 B43A	PUSH	{R1, R3, R4, R5}
00000004 B43A	PUSH	{R1, R3-R5}
00000006 B500	PUSH	{LR}
00000008 B580	PUSH	{R7, LR}

M::reg_list = 0x03A
= 0'0011'1010b

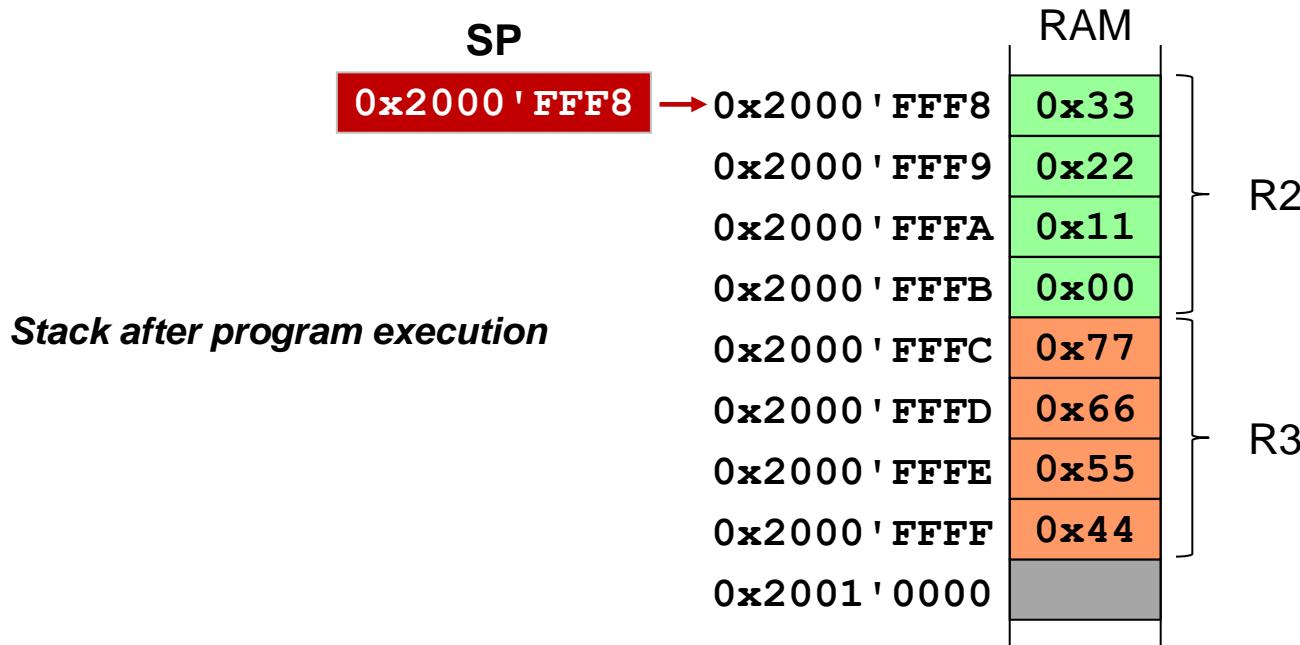
ARM: PUSH and POP

■ Storage Order PUSH

- Lowest register
 - stored to lowest address ¹⁾

Example

LDR	R1 ,=0x20010000
MOV	SP ,R1
LDR	R2 ,=0x00112233
LDR	R3 ,=0x44556677
PUSH	{R2 ,R3 }



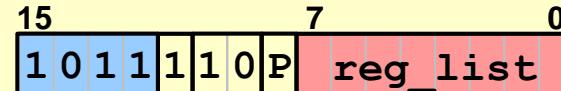
¹⁾ The lowest register is stored first.

ARM: PUSH and POP

■ POP

- **registers**
 - One or more registers to be restored
 - Low registers → `reg_list` = one bit per register
 - PC (R15) → P-bit
 - No other high registers
 - Lowest register reloaded first

POP {registers}



```
addr = SP

for i = 0 to 7
    if reg_list<i> == '1' then
        R[i] = Mem[addr,4]
        addr = addr + 4

if (P == '1') then
    PC = Mem[addr]

SP = SP + 4*BitCount(P::reg_list)
```

00000000 BC80	POP	{R7}
00000002 BC3A	POP	{R1, R3, R4, R5}
00000004 BC3A	POP	{R1, R3-R5}
00000006 BD00	POP	{PC}
00000008 BD80	POP	{R7, PC}

P::reg_list = 0x03A
= 0'0011'1010b

■ ARM Stack

- Only Words → 32-bit
- Pushing and popping of half-words and bytes not possible
- I.e. $SP \bmod 4 = 0$ → word aligned

■ "Number of PUSHs" = "Number of POPs"

■ **Stack-limit < SP < stack-base**

- Stack size has to fit program requirements

Nested Subroutines (revisited)

■ Save LR on Stack

```
ADDR_LED_31_0    EQU      0x60000100
LED_PATTERN       EQU      0xA55A5AA5
Save LR and registers used
by subroutine

subrExample      PUSH     {R4,R5,LR}

; write pattern to LEDs
LDR    R4,=ADDR_LED_31_0
LDR    R5,=LED_PATTERN
STR    R5,[R4]

BL     write7seg
Call another subroutine

POP   {R4,R5,PC}
Restore registers and PC
```

Please note: **BX LR** is not required here, as we directly restore the PC using **POP**

Instructions using SP

■ Add to / subtract from SP

- Immediate offset <imm>
 - Offset range 0 – 1020d and 0 – 508d respectively

load stack pointer
plus offset into
register

allocate (SUB) /
deallocate (ADD)
memory on stack

ADD (SP plus immediate)

ADD <Rd>, SP, #<imm>

15	1010	1	Rd	imm8	0
----	------	---	----	------	---

`<imm> = imm8:00`
`Rd = SP + <imm>`

ADD SP,SP,#<imm>

Memory dump:

15	0	
1 0 1 1	0 0 0 0 0	imm7

`<imm> = imm7:00`
`SP = SP + <imm>`

SUB (SP minus immediate)

high registers as destination are
not supported

SUB SP,SP,#<imm>

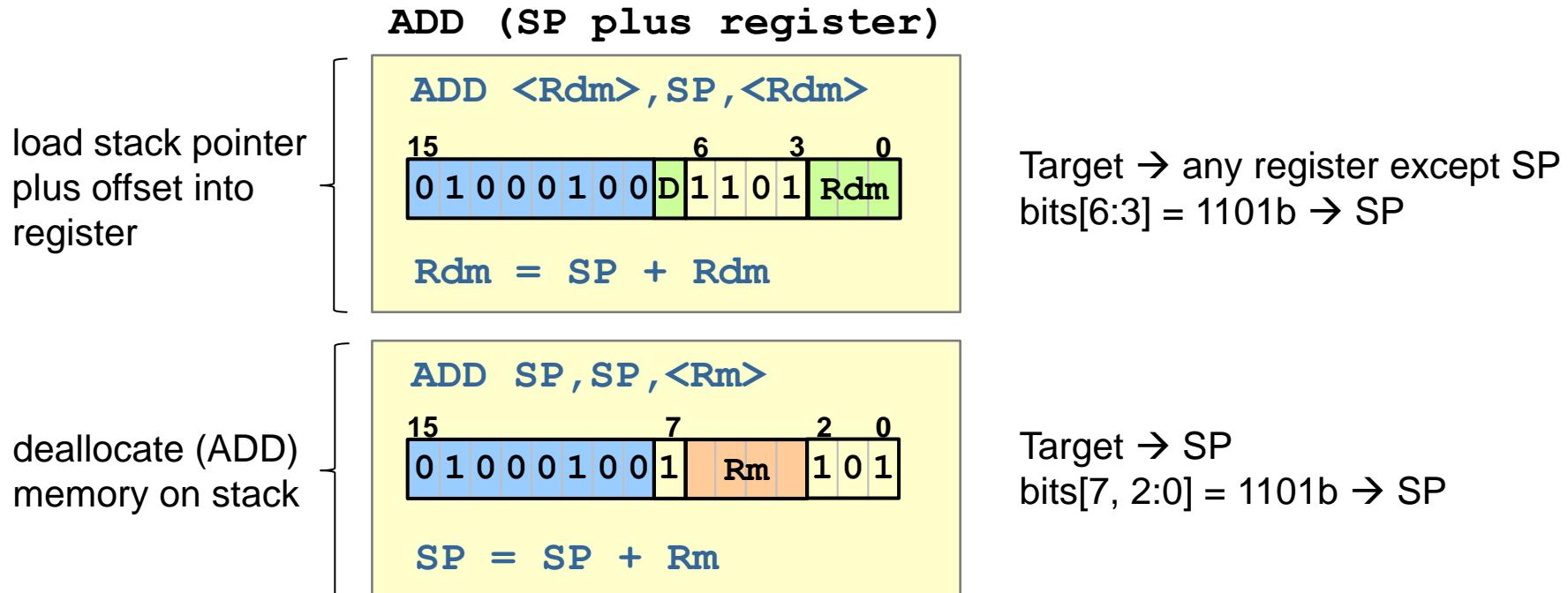
A 16-bit register diagram. The bits are numbered from 15 to 0. Bits 15 through 8 are grouped together and labeled "imm7". Bits 7 through 0 are individual squares.

`<imm> = imm7:00`
`SP = SP - <imm>`

no flags will be changed

Instructions using SP

■ Add to SP (Register)



Target → any register except SP
bits[6:3] = 1101b → SP

Target → SP
bits[7, 2:0] = 1101b → SP

Instructions using SP

■ Instructions with Opcodes previously covered

CMP SP, Rm

CMP Rn, SP

MOV SP, Rm

MOV Rd, SP

Instructions using SP

■ Accessing Memory using SP

- Immediate offset <imm>
- Offset range 0 – 1020d
- Word transfers

LDR (immediate) T2

LDR <Rt>, [SP, #<imm>]

15				0
1	0	0	1	1

<imm> = imm8:00

Rt = Mem[SP + <imm>]

STR (immediate) T2

STR <Rt>, [SP, #<imm>]

15				0
1	0	0	1	0

<imm> = imm8:00

Mem[SP + <imm>] = Rt

Instructions using SP

■ Using other instructions to implement¹⁾

- PUSH {R2,R3,R6}

00000000	B083	SUB	SP , SP , #12
00000002	9200	STR	R2 , [SP]
00000004	9301	STR	R3 , [SP , #4]
00000006	9602	STR	R6 , [SP , #8]

- POP {R2,R3,R6}

00000008	9A00	LDR	R2 , [SP]
0000000A	9B01	LDR	R3 , [SP , #4]
0000000C	9B02	LDR	R6 , [SP , #8]
0000000E	B003	ADD	SP , SP , #12

■ Assembler Directives

- PROC / ENDP
- FUNCTION / ENDFUNC

■ Mark start and end of a procedure / function

- Used by debugger (tool)
 - Buttons "step over" and "step out"
- Structure code for reader

```
subrExample      PROC
                  PUSH      { . . . , LR }
                  ...
                  ...
                  POP       { . . . , PC }
ENDP
```

Conclusions

■ Subroutines

- Structured programming
 - Avoids duplicated code / clear interface
- Call and return on ARM:
 - **BL <label>** and **BX LR** / **POP {PC}**
- Nested subroutines → save LR on stack

■ Stack

- Continuous area of memory → Last-in First-Out
- **PUSH** und **POP**
- ARM
 - Full-descending stack
 - SP points to last entry that has been written
 - grows from higher towards lower addresses

Parameter Passing

Computer Engineering 1

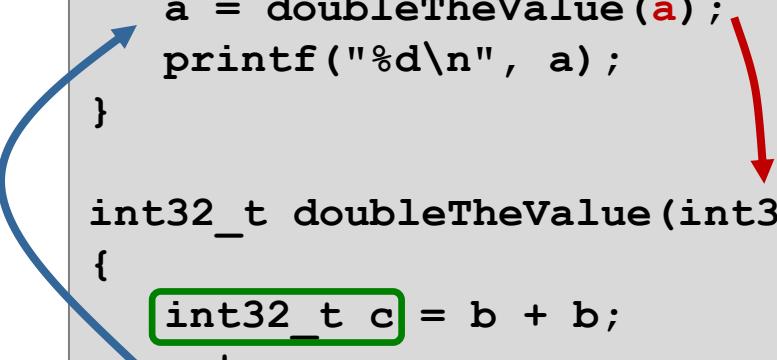
Motivation

```
#include <stdio.h>

int32_t doubleTheValue(int32_t b);

int32_t main(void)
{
    int32_t a = 5;
    a = doubleTheValue(a);
    printf("%d\n", a);
}

int32_t doubleTheValue(int32_t b)
{
    int32_t c = b + b;
    return c;
}
```



- How does `main()` pass the value of **a** to the function?
- Where is the local variable **c** stored?
- How is the value of **c** returned to `main()`?

Agenda

- Parameter Passing
- Passing through Registers
- Passing through Global Variables
- Reentrancy
- ARM Procedure Call Standard
- Functions - Stack Frame
- Calling Assembly Subroutines from C

Learning Objectives

At the end of this lesson you will be able

- to explain and classify the different possibilities to pass data between different parts of the program
- to outline what an Application Binary Interface is
- to name the roles of the different registers in the ARM Procedure Call Standard
- to enumerate and describe the operations of the caller of a subroutine
- to summarize the structure of a subroutine and describe what happens in the prolog and epilog respectively
- to explain, interpret and discuss stack frames
- to access elements of a stack frame in assembly
- to understand the build-up and tear-down of stack-frames
- to call an assembly subroutine from a C program

Parameter Passing

■ Where?

- **Register**
 - Caller and Callee¹⁾ use the same register
- **Global variables**
 - Shared variables in data area (section)
- **Stack**
 - Caller → PUSH parameter on stack
 - Callee → access parameter through `LDR <Rt>, [SP, #<imm>]`

■ How?

- **pass by value**
 - Handover the value
- **pass by reference**
 - Handover the address to a value

¹⁾ Caller: the routine that calls the subroutine; Callee: the subroutine being called

Passing through Registers

Register / "pass by value"

```
AREA exData,DATA,...  
...  
  
AREA exCode,CODE,...  
...  
MOVS R1,#0x03  
BL double  
MOVS ...,R0  
...  
  
double  
LSLS R0,R1,#1  
BX LR
```

caller

callee
function
double

- **Values in agreed registers, e.g.**
 - **R1** Parameter:
Caller → function
 - **R0** Return value of function
- **Efficient and simple**
- **Limited number of registers**
 - How do we pass tables and structs?

Passing through Registers

Register / "pass by reference"

```
TLENGTH      EQU      16
AREA exData,DATA,...  
p1Table SPACE TLENGTH

AREA exCode,CODE,...  
    ...  
    1)  
    LDR    R0 ,=p1Table  
    MOVS   R1 ,#TLENGTH  
    BL     doubleTableValues  
    ...  
  
doubleTableValues  
    MOVS   R2 ,#0  
loop  LDRB   R4 ,[R0 ,R2]  
    LSLS   R4 ,R4 ,#1  
    STRB   R4 ,[R0 ,R2]  
    ADDS   R2 ,#1  
    CMP    R2 ,R1  
    BLO    loop  
    BX     LR
```

caller

callee
function
doubleTableValues

- Pass reference (= address) of data structure in register
- Allows passing of larger structures
- Example
 - Function `doubleTableValues`
 - doubles each value in the table
 - **R0** Caller passes address of `p1Table`
 - **R1** Caller passes length of table (pass by value)

¹⁾ Filling the table with values is not shown in the code

Passing through Global Variables

Global Variables

```
AREA exData,DATA,...
```

caller

```
param1 SPACE 1
result SPACE 1
```

```
AREA exCode,CODE,...
```

function double_g

```
...
LDR      R4 ,=param1
MOVS    R5 ,#0x03
STRB    R5 ,[R4]
BL      double_g
LDR      R4 ,=result
LDRB    ... ,[R4]
...

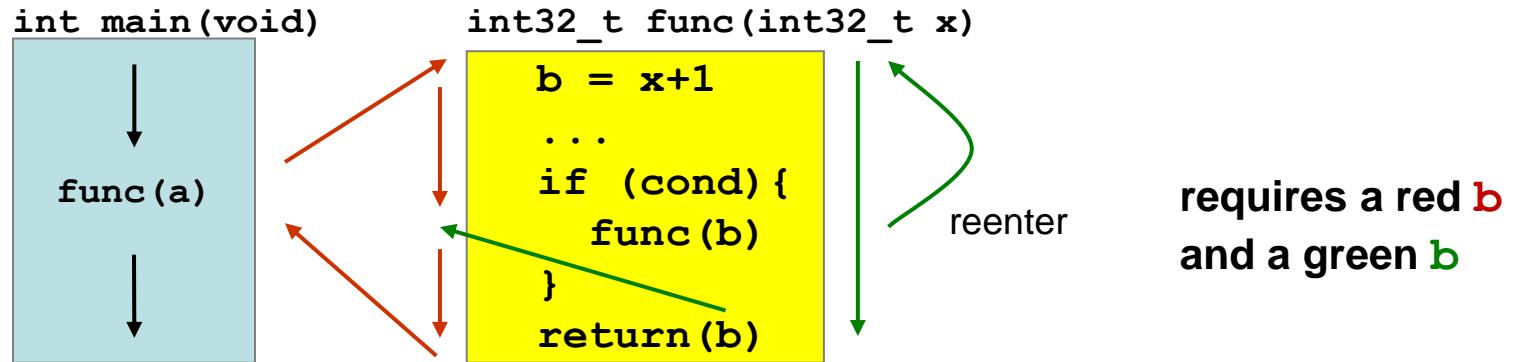
double_g
LDR      R4 ,=param1
LDRB    R1 ,[R4]
LSLS    R0 ,R1 ,#1
LDR      R4 ,=result
STRB    R0 ,[R4]
BX      LR
```

- **Shared variables in data area**
 - **param1** Caller → procedure
 - **result** Return value
- **Overhead to access variable**
 - In Caller and Callee
- **Error-prone, unmaintainable**
 - No encapsulation,
 - Many dependencies
 - Multiple use of the same variable
 - Where is the variable written?
 - Who is allowed to read variable?
 - Requires unique variable names
 - Challenge if there is a large number of modules

Reentrancy

■ Recursive Function Calls?

- Registers and global variables are overwritten
- Requires an own set of data for each call
 - Parameters / Local variables



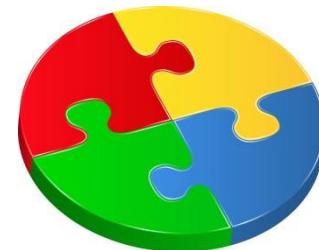
■ Solution

- Combined use of registers and stack for parameter passing
- See ARM Procedure Call Standard

ARM Procedure Call Standard

■ Procedure Call Standard for the ARM architecture (AAPCS)

- http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf
- Part of the ABI (Application Binary Interface) for the ARM Architecture
- ABI → Specification to which independently produced relocatable object files must conform to be statically linkable¹⁾ and executable
 - Function calls
 - Parameter passing
 - Binary formats of information



source: colourbox

“The AAPCS defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine (caller) and a called routine (callee).”

→ Enables interaction of code produced by different compilers.

¹⁾ Linking will be covered later

ARM Procedure Call Standard

■ AAPCS specifies

- Most of the items have already been covered in this course

Layout of data

- Size, alignment, layout of fundamental data types

Register Usage

- What are the registers used for

Memory Sections and Stack

- Code, read-only data, read-write data, stack, heap

Stack

- Full-descending, word-aligned, ...

Subroutine Calls

- Mechanism using LR and PC

Result Return

- Returning arguments through r0 (and r1 – r3)

Parameter Passing

- Passing arguments in r0-r3 and on stack

ARM Procedure Call Standard

■ Register Usage

Register	Synonym	Role
r0	a1	Argument / result / scratch register 1
r1	a2	Argument / result / scratch register 2
r2	a3	Argument / scratch register 3
r3	a4	Argument / scratch register 4
r4	v1	Variable register 1
r5	v2	Variable register 2
r6	v3	Variable register 3
r7	v4	Variable register 4
r8	v5	Variable register 5
r9	v6	Variable register 6
r10	v7	Variable register 7
r11	v8	Variable register 8
r12	IP	Intra-Procedure-call scratch register ¹⁾
r13	SP	
r14	LR	
r15	PC	

Register contents
might be modified
by callee

Cortex-M0: Registers
r8 – r11 have limited set of
instructions. Therefore,
they are often not used by
compilers.

Callee must
preserve contents
of these registers
(Callee saved)

■ Scratch Register

- Used to hold an intermediate value during a calculation
- Usually, such values are not named in the program source and have a limited lifetime

■ Variable Register

- A register used to hold the value of a variable, usually one local to a routine, and often named in the source code.
- Cortex-M0 registers R8 – R11 (v5 – v8) are often unused
 - as they are accessible only by few instructions

■ Argument, Parameter

- Used interchangeably
- Formal parameter of a subroutine

■ Parameters

- Caller copies arguments to R0 to R3
- Caller copies additional parameters to stack

■ Returning fundamental data types

- Smaller than word zero or sign extend to word; return in R0
- Word return in R0
- Double-word return in R0 / R1 ¹⁾
- 128-bit return in R0 – R3 ¹⁾

■ Returning composite data types (*structs, arrays, ...*)

- Up to 4 bytes return in R0
- Larger than 4 bytes stored in data area; address passed as extra argument at function call

¹⁾ Least significant word stored in lowest register

ARM Procedure Call Standard

■ Example

```

void caller(void)
{
    uint32_t p = 4;
    uint32_t q = 5;
    uint32_t r = 6;
    uint32_t sum;

    sum = callee(p,q,r);
}
  
```

```

uint32_t callee(uint32_t a,
                uint32_t b,
                uint32_t c)
{
    return a + b + c;
}
  
```

MOVS r4,#4
 MOVS r5,#5
 MOVS r6,#6 } local variables
 R4 – R6

MOV r2,r6
 MOV r1,r5
 MOV r0,r4 } copy parameters
 to R0 – R2

BL callee
 MOV r7,r0 } copy return value
 to local variable

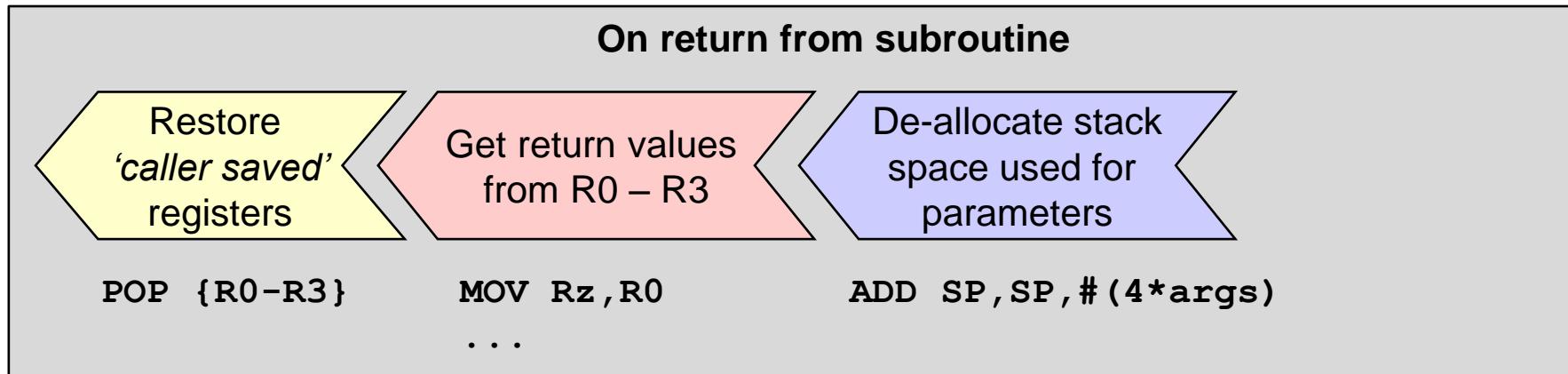
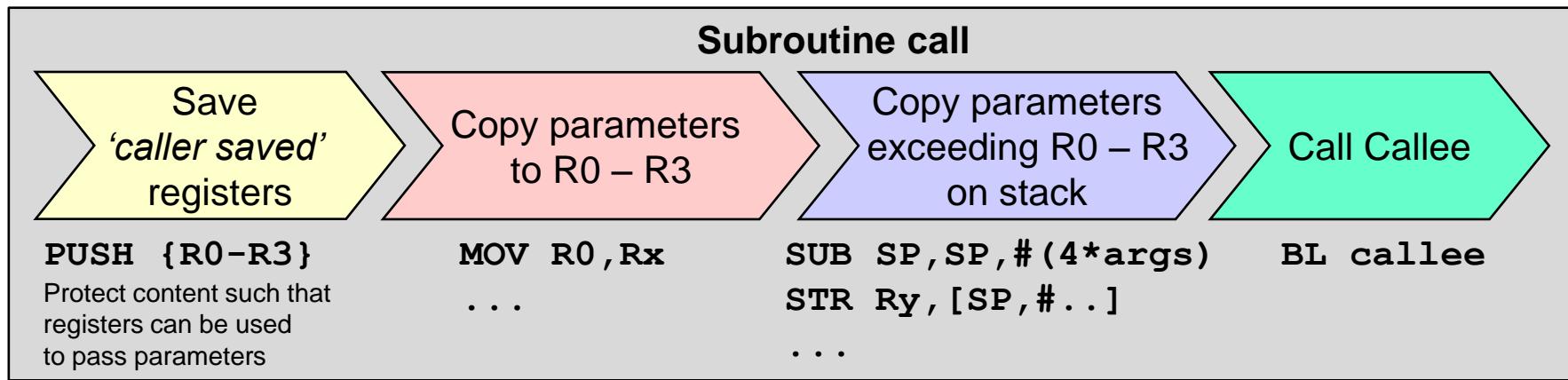
callee PROC
 ADDS r0,r0,r1
 ADDS r0,r0,r2
 BX lr } callee uses
 own copies
 of parameters

PUSH and POP are omitted in the example

ARM Procedure Call Standard

■ Subroutine Call – Caller Side

Pattern as used by the compiler. Manually written assembly code may be slightly different.



ARM Procedure Call Standard

■ Subroutine Structure – Callee Side

Pattern as used by the compiler. Manually written assembly code may be slightly different.

Prolog – Entry of subroutine

Save 'callee saved'
register contents
to stack¹⁾

PUSH {R4-R7,LR}

Protect register contents. Allows
use of registers inside callee.

Allocate stack
space for
local variables

SUB SP,SP,#...

Copy input para-
meters to scratch/
variable registers

MOV Rx,R0

Epilog – Before returning to caller

Restore 'callee saved'
registers from stack¹⁾

POP {R4-R7,PC}

Callee restores registers for
caller before returning.

Release stack
space for
local variables

ADD SP,SP,#...

Store result
in R0 – R3

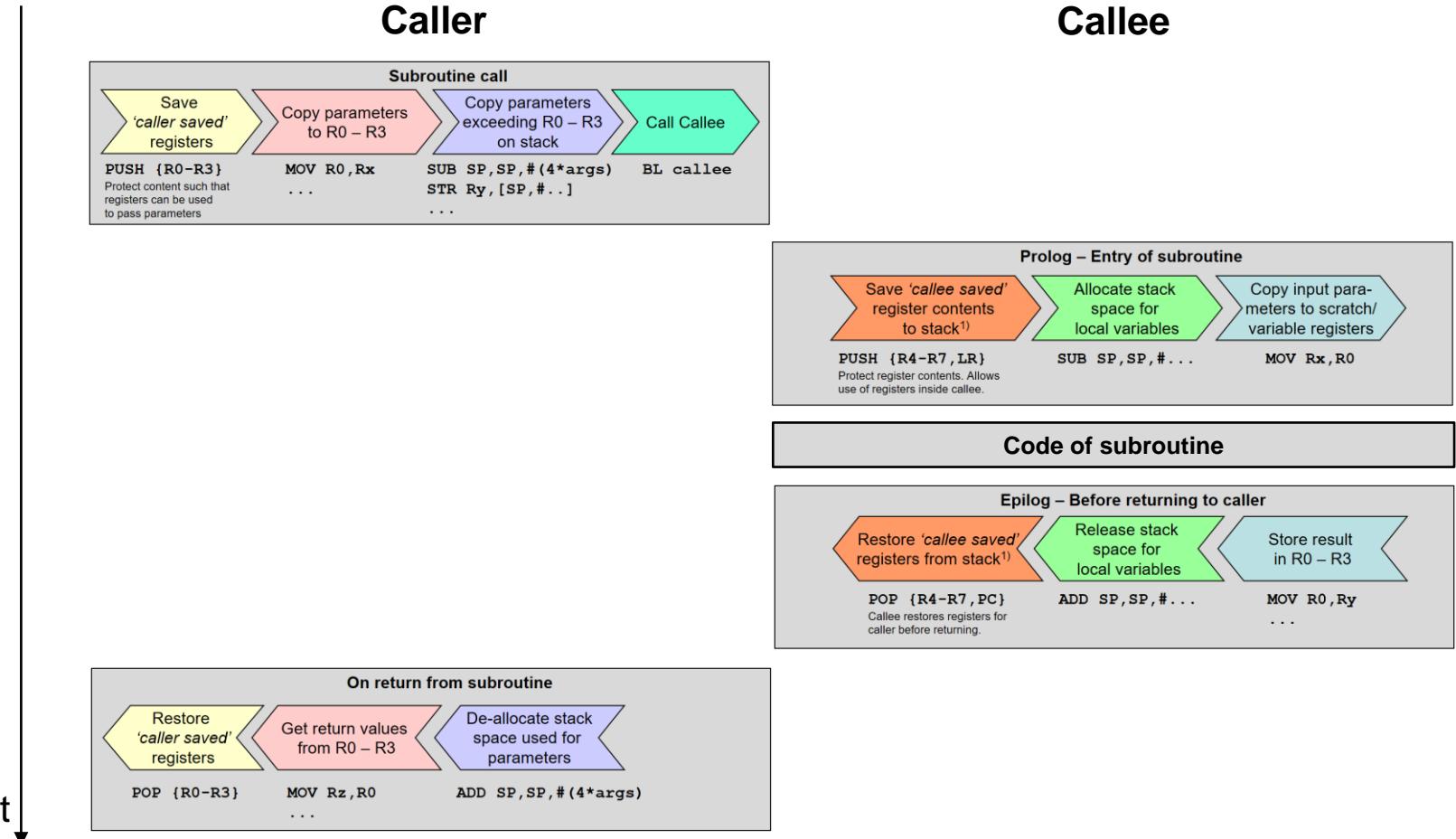
MOV R0,Ry

...

¹⁾ only registers modified by the callee are saved and restored. r8 – r11 are often unused by the compiler.

ARM Procedure Call Standard

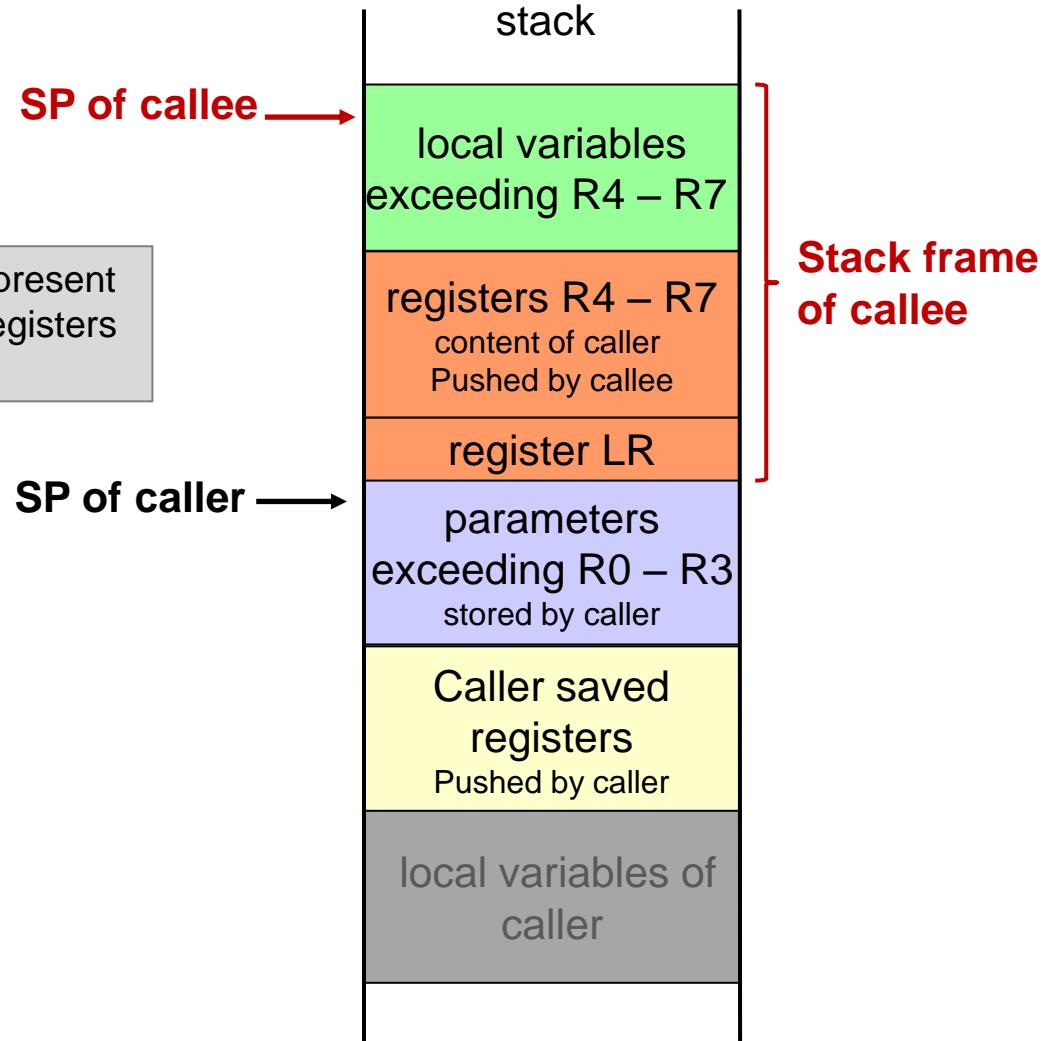
■ Summary – Flow of Execution for a Subroutine Call



Functions - Stack Frame

■ Stack Frame

Each field may or may not be present depending on the number of registers required by the function

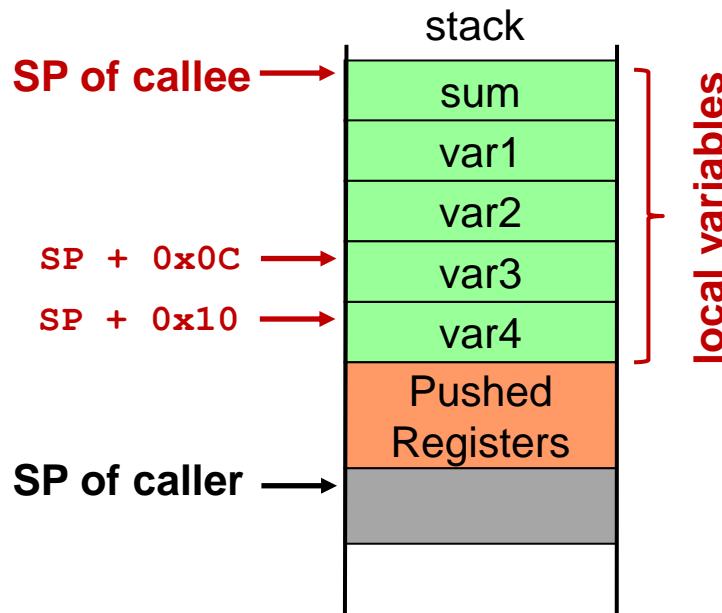


Functions – Stack Frame

■ Access to Local Variables on Stack

- Example using word sized variables

Assume that the compiler has allocated the local variables as follows:



Compiler uses indirect addressing relative to stack pointer (SP)

```
sum = var3 + var4;  
LDR R0, [SP, #0x0c] var3  
LDR R1, [SP, #0x10] var4  
ADDS R0, R1, R0  
STR R0, [SP, #0x00] sum
```

R0 and R1 used as scratch register

Functions - Stack Frame

■ Functions in C

- Subroutine with return value (usually in register R0 / R1)
- Function Parameters
 - Always "pass by value"
 - Copy is being passed (not the original)
 - "pass by reference" only possible through use of pointers
 - Pointer itself passed by value
 - Registers R0 – R3
 - starting with the first argument in R0
 - Stack if more space is required
- Local Variables
 - In registers R4 – R7
 - On stack if more space is required or address operator (&) is used

Functions - Stack Frame

■ Example

```

int main(void)
{
    uint32_t start = 0x1234;
    uint32_t result = 0;

    result = calc(start);
}

uint32_t calc(uint32_t val)
{
    uint32_t a[8];
    uint32_t res = 0;
    ...
    a[0] = val;
    ...
    return res;
}
  
```

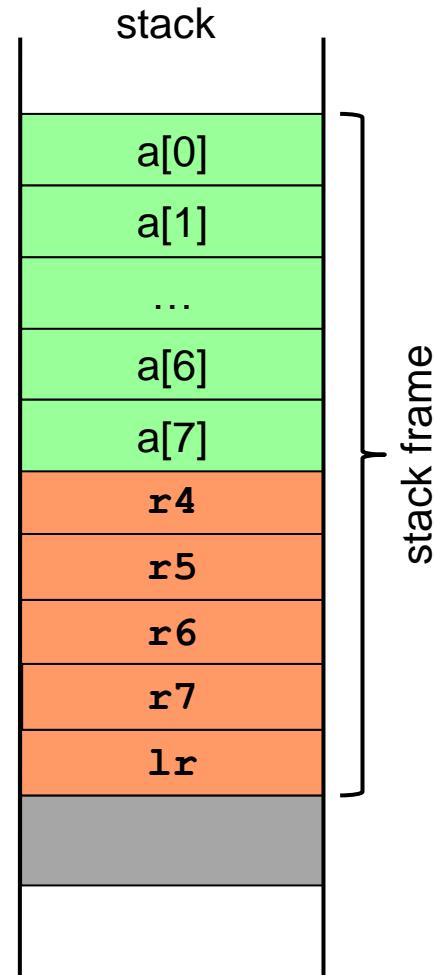
MOV r0,... ;start
 BL calc

PUSH {r4-r7,lr}
 ;allocate a[] on stack
 SUB sp,sp,#0x20

MOVS r6,#0 ;res
 ;access a[0]
 STR r0,[sp,#0x00]

MOV r0,r6 ;res
 ;stack teardown
 ADD sp,sp,#0x20

POP {r4-r7,pc}



Functions - Stack Frame

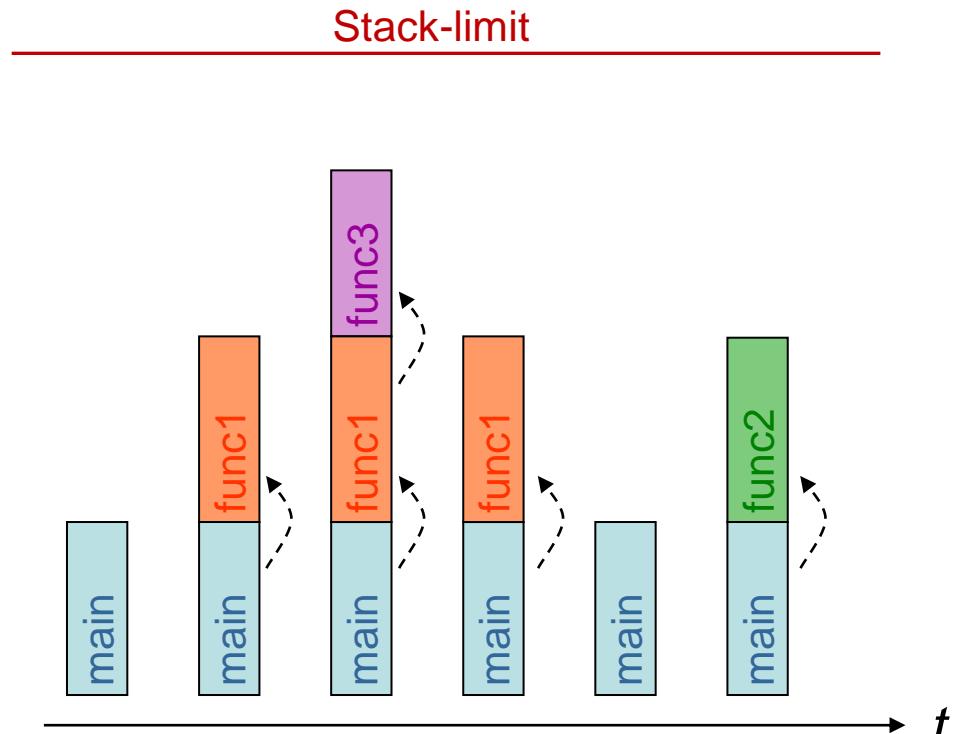
■ Build-up and tear-down of stack frames

```
int32_t main(void)
{
    int16_t x, y;
    x = func1(...);
    y = func2(...)
}

int16_t func1(...)
{
    z = func3(...)
    ...
}

int16_t func2(...) {...}

int16_t func3(...) {...}
```



Calling Assembly Subroutines from C

```
extern void strcpy(char *d, const char *s);
int main(void)
{
    const char *srcstr = "First string ";
    char dststr[] = "Second string";
    strcpy(dststr,srcstr);
    return (0);
}
```

```
PRESERVE8
AREA      SCopy, CODE, READONLY
EXPORT strcpy

strcpy:
    ; R0 points to destination string
    ; R1 points to source string
    LDRB R2, [R1]          ; Load byte and update address
    ADDS R1, R1, #1
    STRB R2, [R0]          ; Store byte and update address
    ADDS R0, R0, #1
    CMP   R2, #0            ; Check for null terminator
    BNE   strcpy            ; Keep going if not
    BX    LR                ; Return
END
```

Example : from ARM Ltd.

Conclusions

■ Parameter Passing

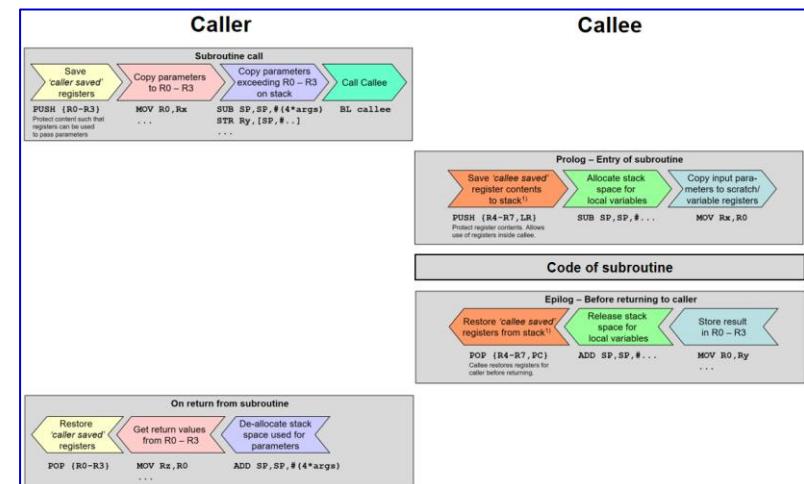
- Through registers
- Through global variables
- On the stack (stack frame)

■ Procedure Call Standard

- R0 – R3 parameters / scratch → callee may modify these registers
- R4 – R7 local variables → callee restores potentially modified registers
- R0 – R3 Return

■ Stack Frame

- Nesting of subroutines
- Stack frame is constructed at function call
- Stack frame is deconstructed when function terminates (returns)



Modular Coding/Linking

Computer Engineering 1

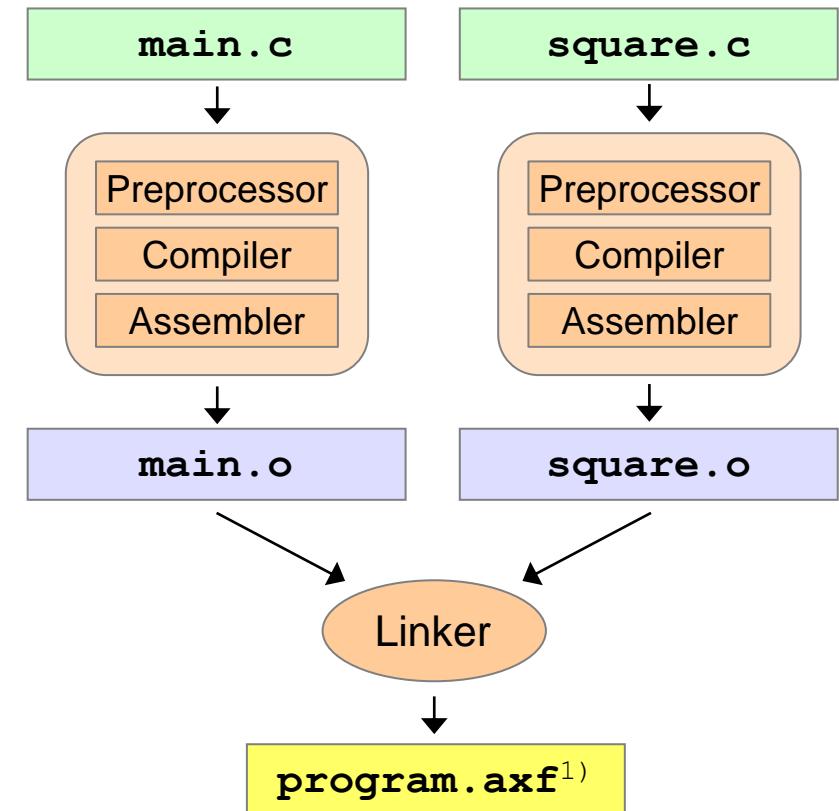
Motivation

■ From source code to executable program

```
main.c
...
uint32_t square(uint32_t v);

int main(void)
{
    while(1) {
        LED = square(DIPSW);
    }
}

square.c
...
uint32_t square(uint32_t v)
{
    return v * v;
}
```



1) AXF file extension indicates ARM Executable File Format - other environments may have other executable formats

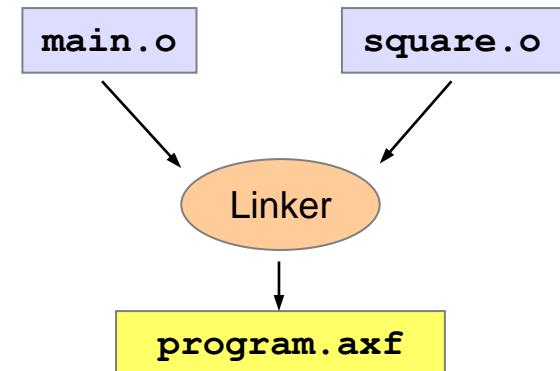
Agenda

■ Modular programming

- Why modular programming
- Some guidelines for designing modular programs

■ From source code to the executable program

- Source code anatomy
- Linker
 - Merging code sections
 - Merging data sections
 - Symbol resolution
 - Symbol relocation



■ Tools, libraries, debugging

- Cross compiler tool chain
- Static libraries versus dynamic libraries
- Source level debugging



Learning Objectives

At the end of this lesson you will be able

- to explain the concepts behind modular programming
- to appropriately partition C and assembly programs into modules
- to explain the steps involved from source to the executable program
- to interpret map files of object files and executable programs
- to explain the main tasks of a linker: merging, resolution, relocation
- to explain the rules the linker applies for resolution and relocation
- to explain the difference between static and dynamic linking
- to explain the concept of source level debugging

Modular Programming – Overview

■ Why modular programming?

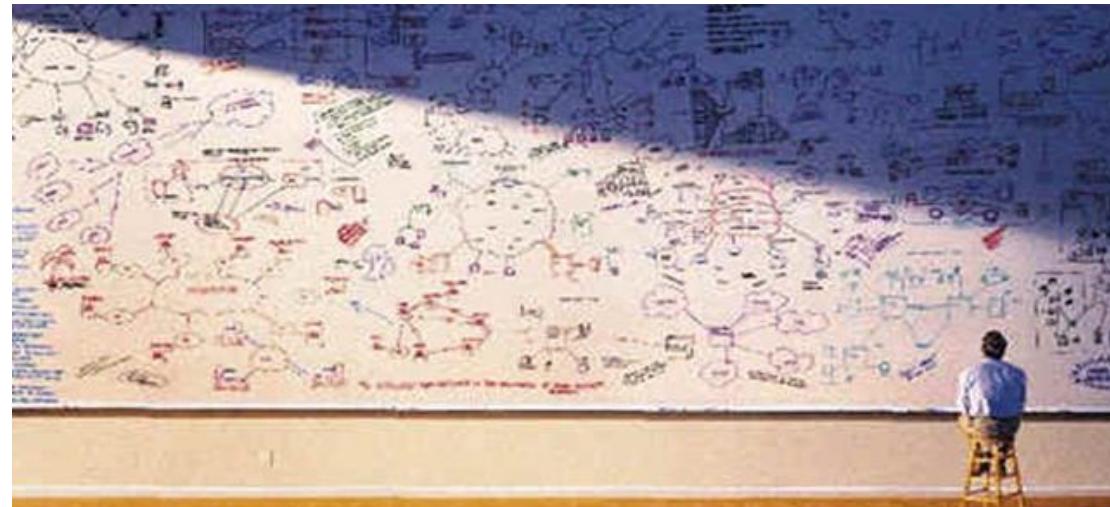
- To manage complexity!

■ Basic rules

- Group together what belongs together
- Split what does not belong together
- Don't repeat code
 - Make modules, types, and functions instead
 - Enables reusing of existing code

■ Intellectual effort

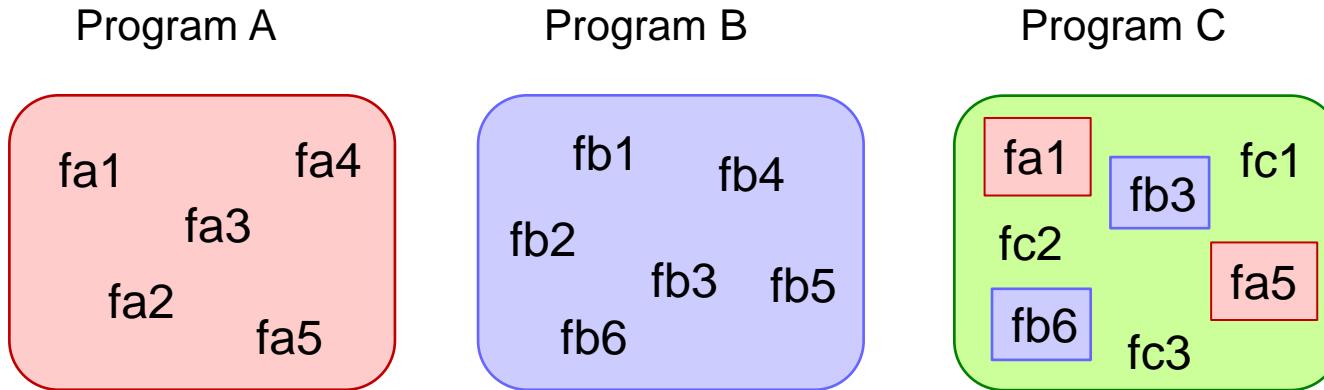
- There is no golden rule but established practice



1) Image source: <http://www.marketingsavant.com/wp-content/uploads/2010/06/Complexity650300.jpg>

Modular Programming

■ Example: Problem of non-modular programming



- Three single-module programs – no shared code
 - Program C needs parts from program A and B
 - Program C has copies of parts from A and B
 - Changing code and fixing bugs requires effort on multiple sources

Modular Programming

■ Managing complexity by modular programming

Topic	Benefits
Enable working in teams	Multiple developers working on the same source repository
Useful partitioning and structuring of the programs	Eases reusing of modules
Individual verification of each module	Benefits all users of the module
Providing libraries of types and functions	For reuse instead of reinvention
Mixing of modules that are programmed in various languages	E.g. mix C and assembly language modules
Only compile the changed modules	Speeds up compilation time

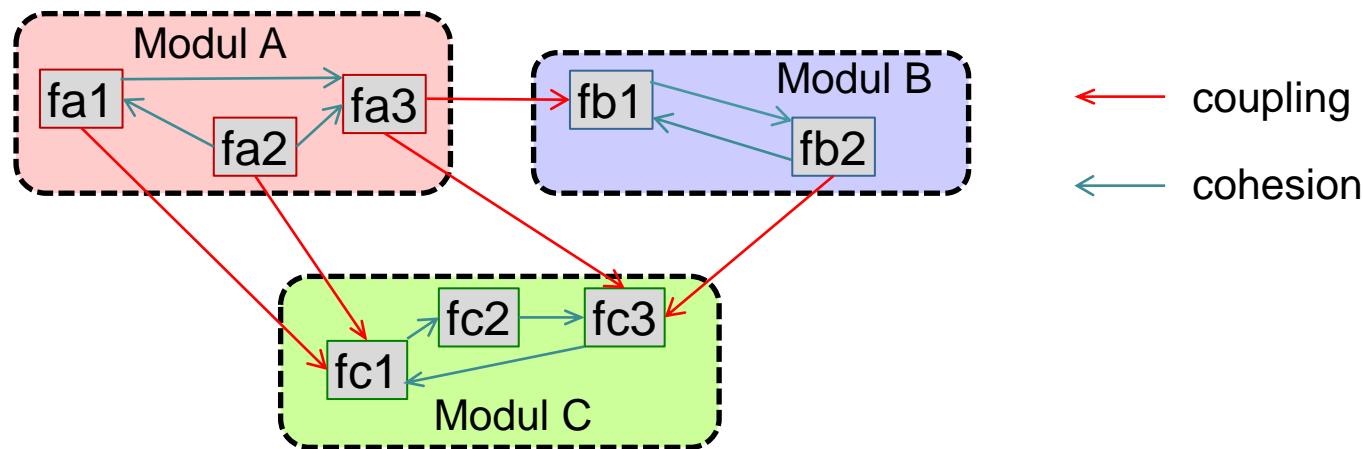
Modular Programming – Guidelines

■ High module cohesion

- Group together what belongs together
- Lean external interface
- Idea: each module fulfills a **single** defined task

■ Low module coupling

- Split what does not belong together
- Little dependencies between modules



Modular Programming – Guidelines

■ Divide and conquer

- Partition functionality into manageable chunks
- Hierarchical design

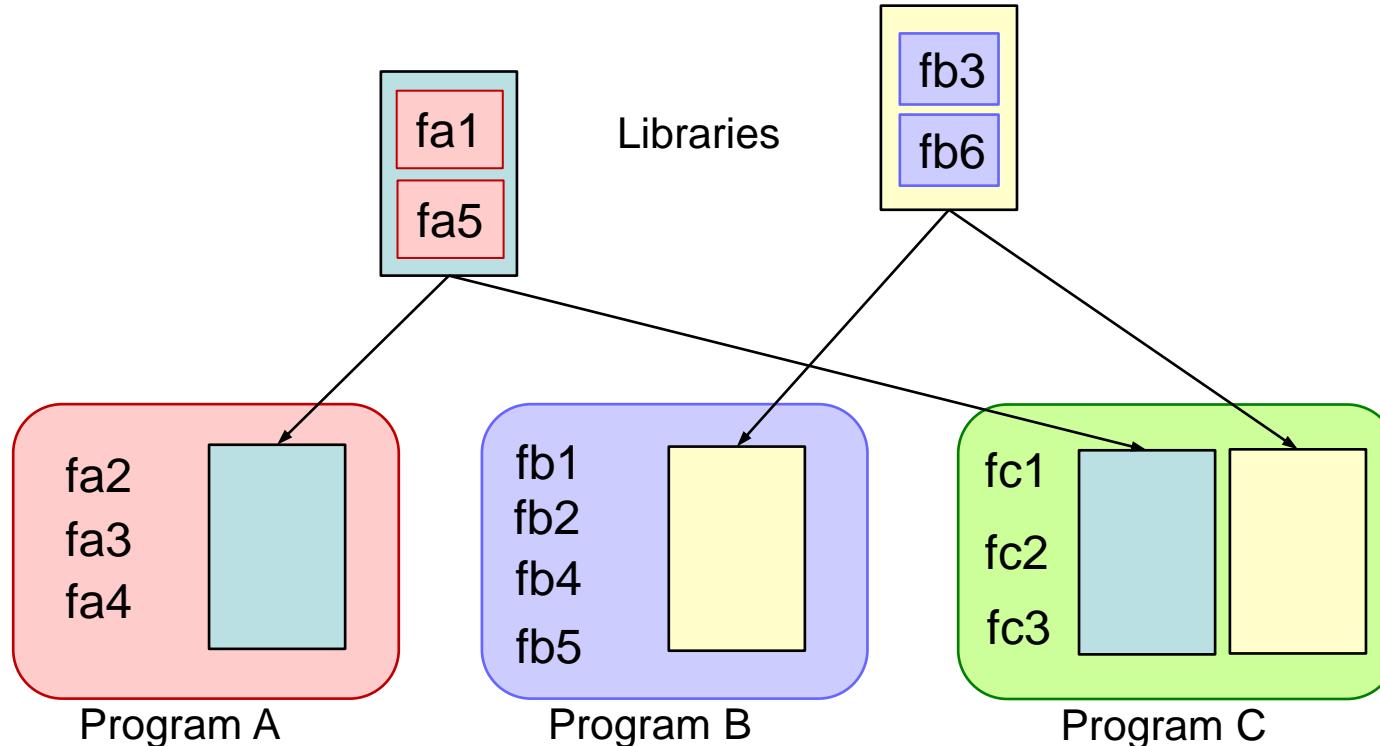
■ Information hiding

- Split interface from implementation
- Do not disclose unnecessary details
- Maintain freedom to change implementation details

Modular Programming – Guidelines

■ Reuse

- Libraries of functions and types to enable reuse



Modular Programming – Design

■ Module design and implementation

- Definition of module interface
 - Defines what functionality is available to the client of the code
- Implementation of module
 - Provides the functionality behind the interface
 - An interface may have alternative implementations
- Individual testing of each module
 - Modules can be tested individually if designed accordingly
- Reuse of existing modules in new modules
 - Modules should be designed such that they can be reused

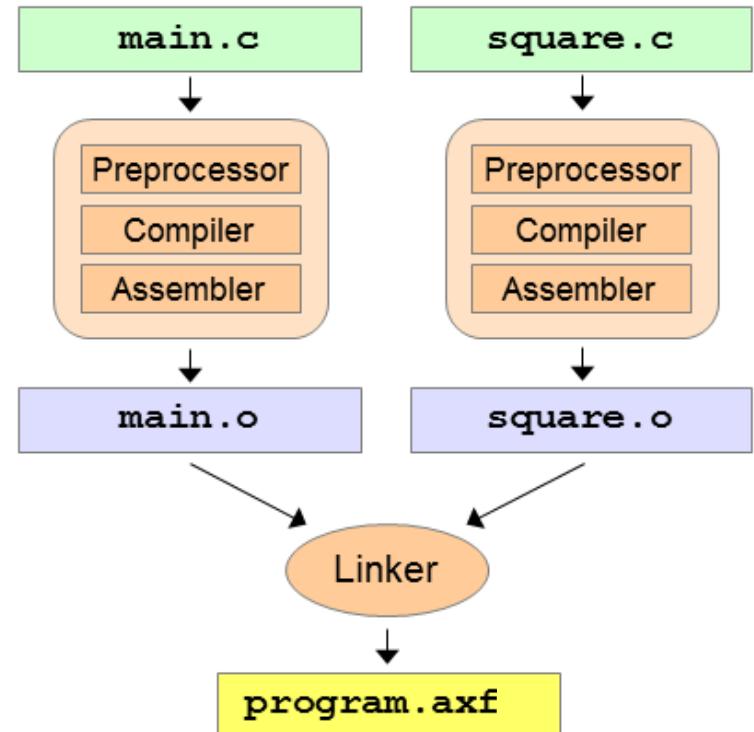
.h file

.c file

Modular Programming – Translation

■ Translation steps

- Compile/assemble each module
 - Results in an object file for each module (module.o¹)
- Link all object files
 - Results in one executable file²



1) The file extension may vary depending on the environment

2) See later slides of this lecture

■ Partitioning into modules

- Modular programming: the whole source code base is split into multiple source files¹⁾
- Each source file defines a module
- Each source file gets translated into one object file
- The object files get linked into an executable file

■ Implications for source code

- C declarations and definitions
- Header files to share commonly used declarations
- Linkage of declarations and definitions
- From C-declarations and C-definitions to assembly symbols
- From assembly symbols to object file symbols

1) Source files are the ones that hold the core code of the module - header files are discussed later in this lecture

■ Challenge

- Modular programming requires concepts where
 - Types, functions and variables may be defined in other modules than where they are used
 - Consistency of types, functions and variables is maintained across module boundaries

■ Solution

- Terminology: declarations and definitions
- Declared-before-used
- One-definition-rule

Source Code Anatomy

■ C: Declaration vs. definition

- Declaration

- Specifies how a name can be used¹⁾

```
uint32_t square(uint32_t v); // square function defined elsewhere
extern uint32_t counter;     // counter variable defined elsewhere
struct S;                   // struct S type defined elsewhere
```

- Definition²⁾

- Where a function is given with its body
- Where memory is allocated for a variable
- A struct type with its members

```
uint32_t square(uint32_t v) { ... } // square function definition
uint32_t counter;                // counter variable definition
struct S { ... };               // struct S type definition
```

1) Function and type declarations do not need the «extern» keyword

2) Each definition is implicitly also a declaration of the given name

■ Some C rules: what is legal and what is illegal code

- Names declared before use

- Each name must be declared before it can be used
- Note: a definition is also a declaration

```
uint32_t square(uint32_t v); // square is declared before use
...
out = square(in); // square is known at the point of use
```

- One-definition-rule

- A variable or function may be declared multiple times
- But may be defined only once in the same scope¹⁾

```
uint32_t in = 5; // legal first definition of the variable in
uint32_t in = 5; // illegal second definition of in
```

1) The exact rules are more elaborate

Source Code Anatomy

■ Challenge: reuse of declarations

- Declared-before-used may result in repeating declarations in multiple source files

```
// program_A.c

// declaration of square
uint32 t square(uint32 t v);
...
int main(void) {
    // use of square
    res = square(a) + b;
    ...
}
```

```
// program_B.c

// declaration of square
uint32 t square(uint32 t v);
...
int main(void) {
    // use of square
    y = square(x);
    ...
}
```

■ Maintenance issues

- Duplicated declarations are a consistency problem

Source Code Anatomy

■ Solution: use of header files

- Use a single header file instead of duplicating declarations
- Avoids copy/paste of declarations
- Maintains consistency over time

```
// square.h
#ifndef _SQUARE_H_ // incl.-guard
#define _SQUARE_H_ // guard

// declaration of square
uint32_t square(uint32_t v);

#endif // end of incl.-guard
```

```
// square.c
#include "square.h"

// definition of square
uint32_t square(uint32_t v)
{
    return v*v;
}
```

- Usage through `#include` preprocessor directive

```
// program_A.c
#include "square.h"
int main(void) {
    res = square(a) + b;
    ...
}
```

```
// program_B.c
#include "square.h"
int main(void) {
    y = square(x);
    ...
}
```

Source Code Anatomy

- **Challenge: Which names can be used by other modules?**¹⁾
 - How to provide names that can be used by other modules?
 - How to inhibit use of internal names by other modules?
- **Solution: Concept of linkage in C¹⁾**
 - External linkage
 - The global name is externally available for use in any modules
 - E.g. a function or a global variable
 - Internal linkage
 - The global name is only internally available for use in this module
 - E.g. a function or a global variable
 - No linkage
 - Any name that is not in the global space

1) I.e. names that are subject to symbol resolution in the linker process

Source Code Anatomy

■ Example: Internal and external linkage (C)

- All global names have external linkage unless defined **static**

```
// square.c
...
uint32_t square(uint32_t v) {
    return v*v;
}
```

```
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
    uint32_t res;
    res = square(a) + b;
    ...
}
```

square = external linkage

a = internal linkage
b = internal linkage
main = external linkage
res = no linkage
square = external linkage¹⁾

1) square has external linkage (no static keyword), but no definition in main.c – needs to be resolved by the linker

Source Code Anatomy

■ From C declaration/definition to assembly symbol

- Names given in C translate into symbols in assembly
- C-definitions with external linkage translate into EXPORT symbols in assembly
- C-declarations with external linkage which are used but not defined in the module translate into IMPORT symbols in assembly

```
// square.c
...
uint32_t square(uint32_t v) {
    return v*v;
}
```



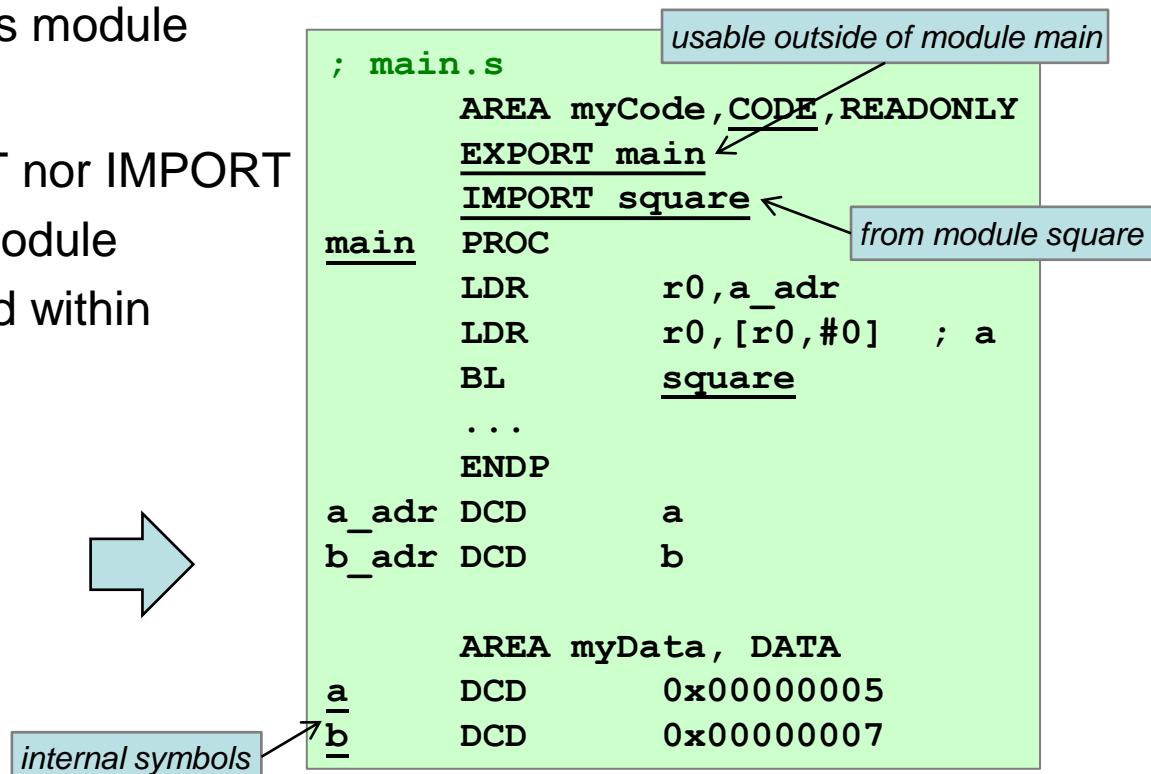
```
; square.s
AREA myCode, CODE, READONLY
EXPORT square
square PROC
    MOV      r1,r0
    MULS   r0,r1,r0
    BX      lr
    ENDP
    END
```

Source Code Anatomy

■ ARM assembly IMPORT and EXPORT keywords

- Linkage control
 - EXPORT declares a symbol for use by other modules
 - IMPORT declares a symbol from another module for use in this module
 - Internal symbols
 - Neither EXPORT nor IMPORT
 - Defined in this module
 - Can only be used within this module

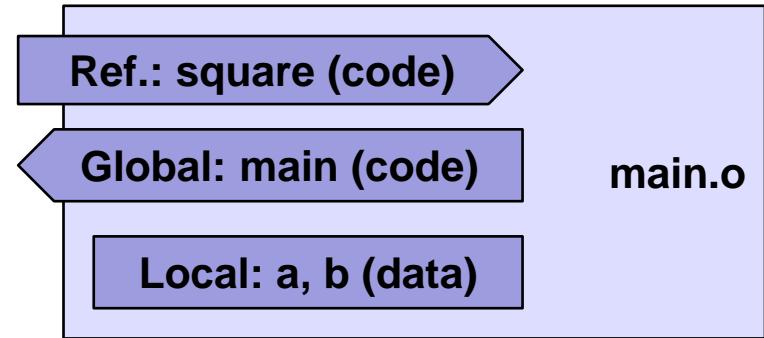
```
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
    uint32_t res;
    res = square(a) + b;
    ...
}
```



Source Code Anatomy

■ From assembly symbols to object file symbols

```
IMPORT square
...
EXPORT main
main: ...
...
a: ...
b: ...
```



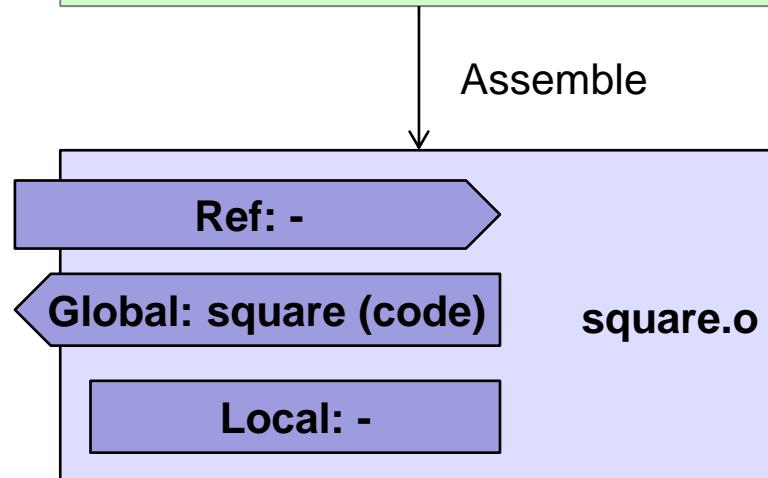
- References
 - Imported symbols from assembly code translate to global reference symbols in the object file
- Global
 - Exported symbols from assembly code translate to global symbols in the object file
- Local
 - Internal symbols from assembly code translate to local symbols in the object file

Source Code Anatomy

■ Example: Assembly to object file

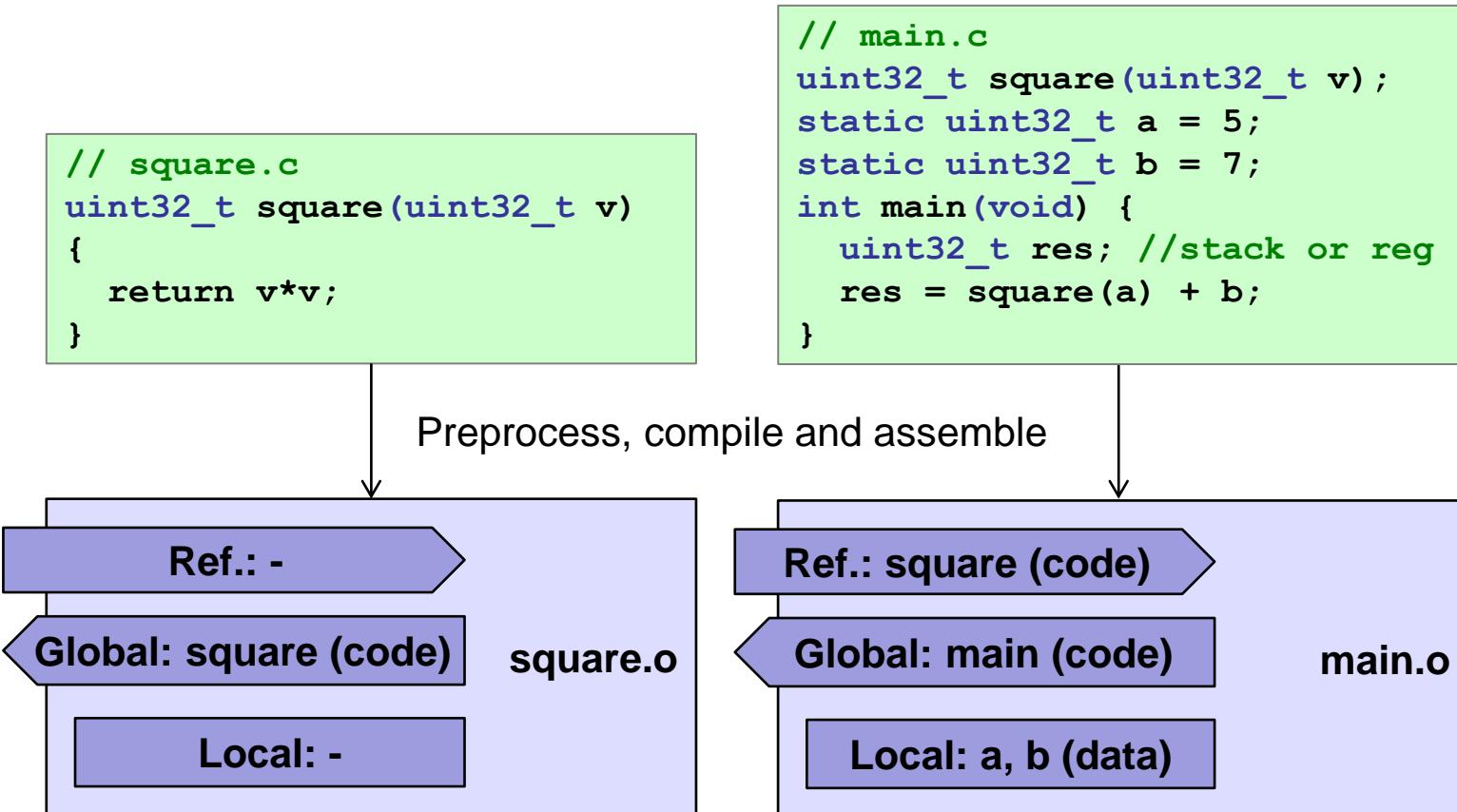
- Exported
 - Code symbol **square**
- Referenced/Imported
 - None
 - No external symbol used
- Local
 - None
 - No internal symbol defined

```
; square.s
AREA myCode,CODE,READONLY
EXPORT square
square PROC
    MOV      r1,r0
    MULS   r0,r1,r0
    BX      lr
ENDP
END
```



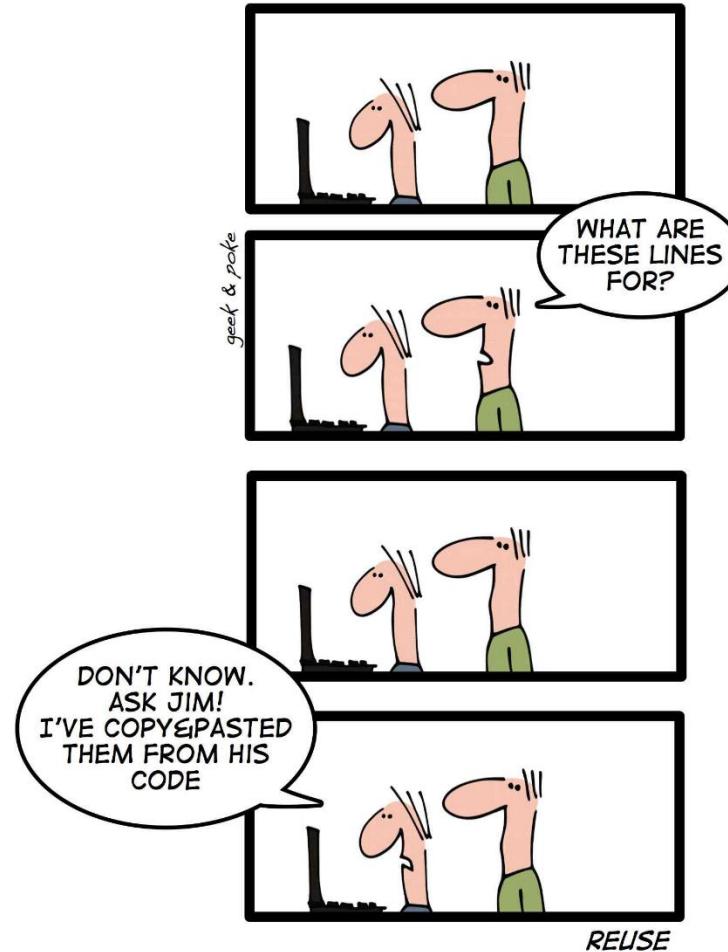
Source Code Anatomy

■ Example: C to object file



Motivation

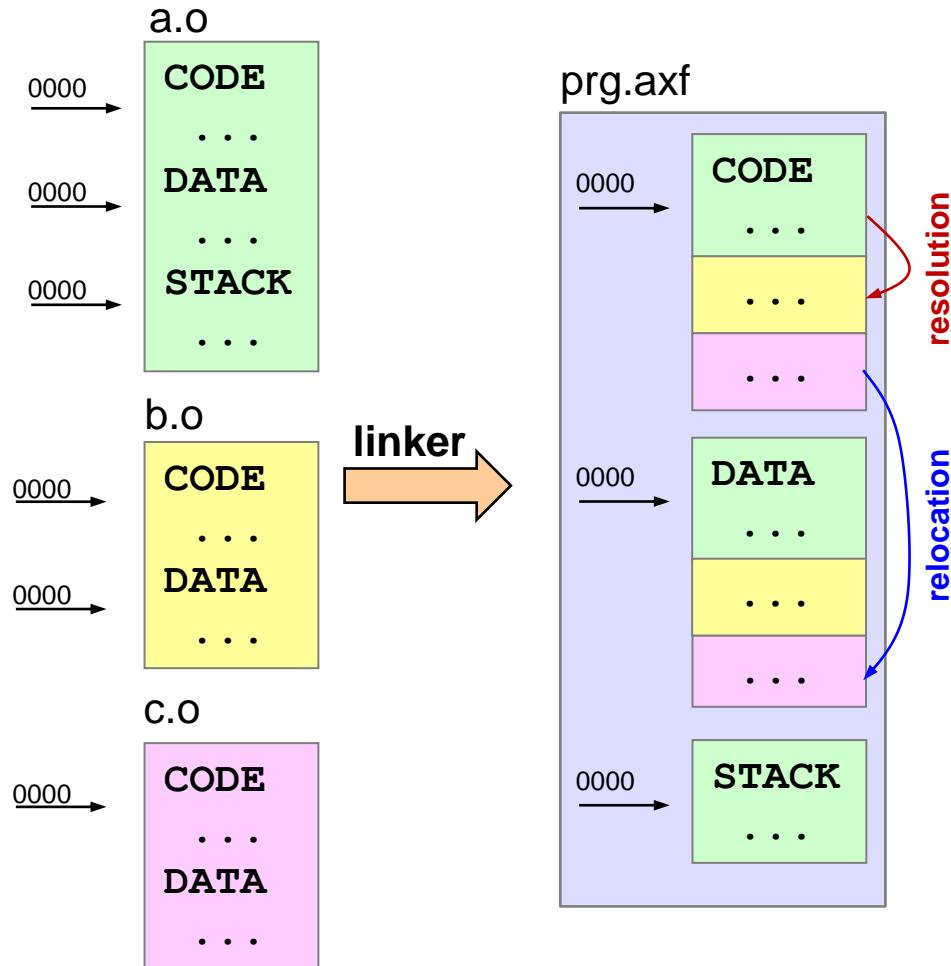
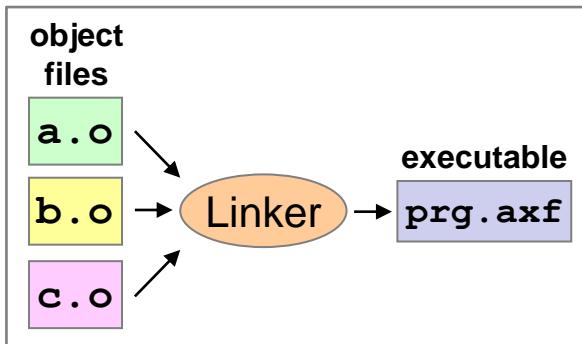
SIMPLY EXPLAINED



Linker – Overview

■ Linker tasks

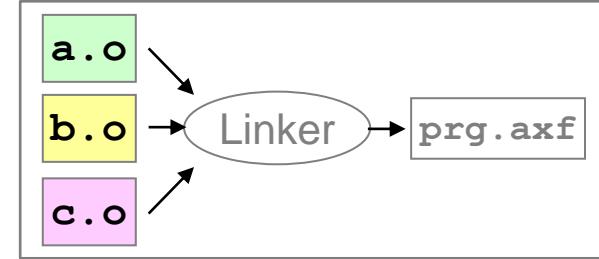
- Merge code sections
- Merge data sections
- Symbol resolution
 - References to other modules
- Address relocation
 - Adapt to new positions of symbols



Linker Input: Object Files

■ Object files

- Contain all compiled data of a module
 - Code section
 - ▶ Code and constant data of the module, based at address 0x0
 - Data section
 - ▶ All global variables of the module, based at address 0x0
 - Symbol table
 - ▶ All symbols with their attributes like global/local, reference, etc.
 - Relocation table
 - ▶ Which bytes of the data and code section need to be adjusted (and how) after merging the sections in the linking process



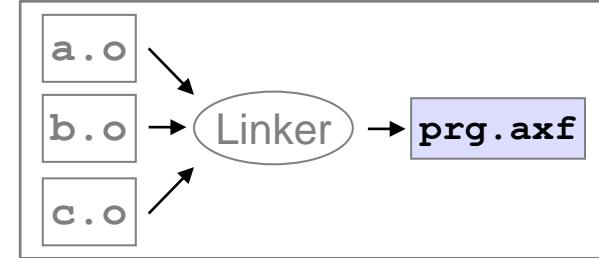
■ ARM tool chain uses ELF for object files

- ELF = Executable and Linkable Format
 - Includes the above mentioned sections as well as further sections (e.g. string tables, debugging information, etc.)

Linker Output: Executable File

■ Executable file

- Contains all linked data of the program
 - Code section
 - ▶ Code and constant data of the program
 - Data section
 - ▶ All global variables of the program
 - Symbol table
 - ▶ All symbols with their attributes like global/local, etc.
- If the program is loaded before execution (by a loader of the hosting operating system), there might still be¹⁾
 - Unresolved symbols for linking with shared (dynamic linked) libraries
 - A relocation table to move the program/data to fixed locations



■ ARM tool chain uses ELF for executable file

- File extension: AXF = ARM eXecutable FFile

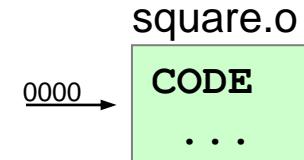
1) In CT1/CT2, we have no hosting OS, so we place the program sections at fixed memory location s – no loader involved

Example ELF Object File

■ square.o¹⁾

- File section #1: code section, at base address **0x00000000**
- File section #5: symbol table: **square = global code symbol**
- No data section (has no global variables)
- No relocation section (no referenced symbols in code/data)

```
File Type: ET_REL (Relocatable object) (1)
...
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
  Address: 0x00000000
square
  0x00000000: 4601      .F      MOV      r1,r0
  0x00000002: 4608      .F      MOV      r0,r1
  0x00000004: 4348      HC     MULS    r0,r1,r0
  0x00000006: 4770      pG     BX     lr
...
** Section #5 '.symtab' (SHT_SYMTAB)
#  Symbol Name          Value      Bind Sec  Type  Vis   Size
=====
6  square              0x00000001  Gb     1  Code  Hi    0x8
```



1) The output is obtained with: `c:\Keil_v5\ARM\ARMCC\bin\fromelf.exe --text -c -d -r -s -z square.o`

Example ELF Object File

■ main.o (part I)

- File section #1: code section, at base address **0x00000000**
 - **0x00000002:** LDR r0, =a (address **a** stored at 0x14)
 - **0x0000000a:** LDR r1, =b (address **b** stored at 0x18)
 - BL **square** calls a dummy address until linked
- File section #4: data section, at base address **0x00000000**

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
  Address: 0x00000000
main
    0x00000000: b510      ..      PUSH    {r4,lr}
    0x00000002: 4804      .H      LDR     r0,[pc,#16] ; LDR r0, =a
    0x00000004: 6800      .h      LDR     r0,[r0,#0]  ; r0 = value at a
    0x00000006: f7fffffe ....    BL      square   ; BL needs adjustment
    0x0000000a: 4903      .I      LDR     r1,[pc,#12] ; LDR r1, =b
    0x0000000c: 6809      .h      LDR     r1,[r1,#0]  ; r1 = value at b
    0x0000000e: 1844      D.      ADDS   r4,r0,r1
    0x00000010: 2000      .      MOVS   r0,#0
    0x00000012: bd10      ..      POP    {r4,pc}

square requires resolution → 0x00000006: f7fffffe .... BL square ; BL needs adjustment
addresses of a and b require relocation → 0x00000014: 00000000 .... DCD   0 ; address a will be stored here
                                         0x00000018: 00000000 .... DCD   0 ; address b will be stored here
** Section #4 '.data' (SHT_PROGBITS) [SHF_ALLOC + SHF_WRITE]
  Address: 0x00000000
    0x00000000: 00000005          ; value at a = 5
    0x00000004: 00000007          ; value at b = 7
```

main.o

CODE

DATA

0000 →

0000 →

0000 →

0000 →

Example ELF Object File

■ main.o (part II)

- File section #6: symbols:
 - **a**: local data section symbol, at offset 0x00000000
 - **b**: local data section symbol, at offset 0x00000004
 - **main**: global code section symbol, at offset 0x00000000
(LSB set: Thumb code)
 - **square**: global code section symbol, referenced
(no definition in main.o)

```
...
** Section #6 '.syms' (SHT_SYMTAB)
# Symbol Name           Value      Bind Sec Type Vis Size
=====
7  a                   0x00000000  Lc     4  Data  De   0x4
8  b                   0x00000004  Lc     4  Data  De   0x4
11 main                0x00000001  Gb     1  Code  Hi   0x14
12 square              0x00000000  Gb   Ref  Code  Hi
...
...
```

Example ELF Object File

■ main.o (part III)

- File section #7: relocation table:
 - Relocation at code address **0x00000006**:
 - Modify the **BL call** to branch to the symbol **square**
 - Relocation at code address **0x00000014**:
 - Set the **absolute 32 bit** value of the symbol **a**
 - Relocation at code address **0x00000018**:
 - Set the **absolute 32 bit** value of the symbol **b**

Relocation table section

```
...
** Section #7 '.rel.text' (SHT_REL)
#      Offset    Relocation Type          Wrt Symbol
=====
0  0x00000006    10 R_ARM_THM_CALL        12 square
1  0x00000014      2 R_ARM_ABS32         7 a
2  0x00000018      2 R_ARM_ABS32         8 b
...
```

Affected code section locations

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
...
0x00000006:  f7fffffe  ....   BL     square       ; BL needs adjustment
...
0x00000014:  00000000  ....   DCD    0 ; address a will be stored here
0x00000018:  00000000  ....   DCD    0 ; address b will be stored here
```

Tasks of a Linker – Overview

■ Tasks of a Linker

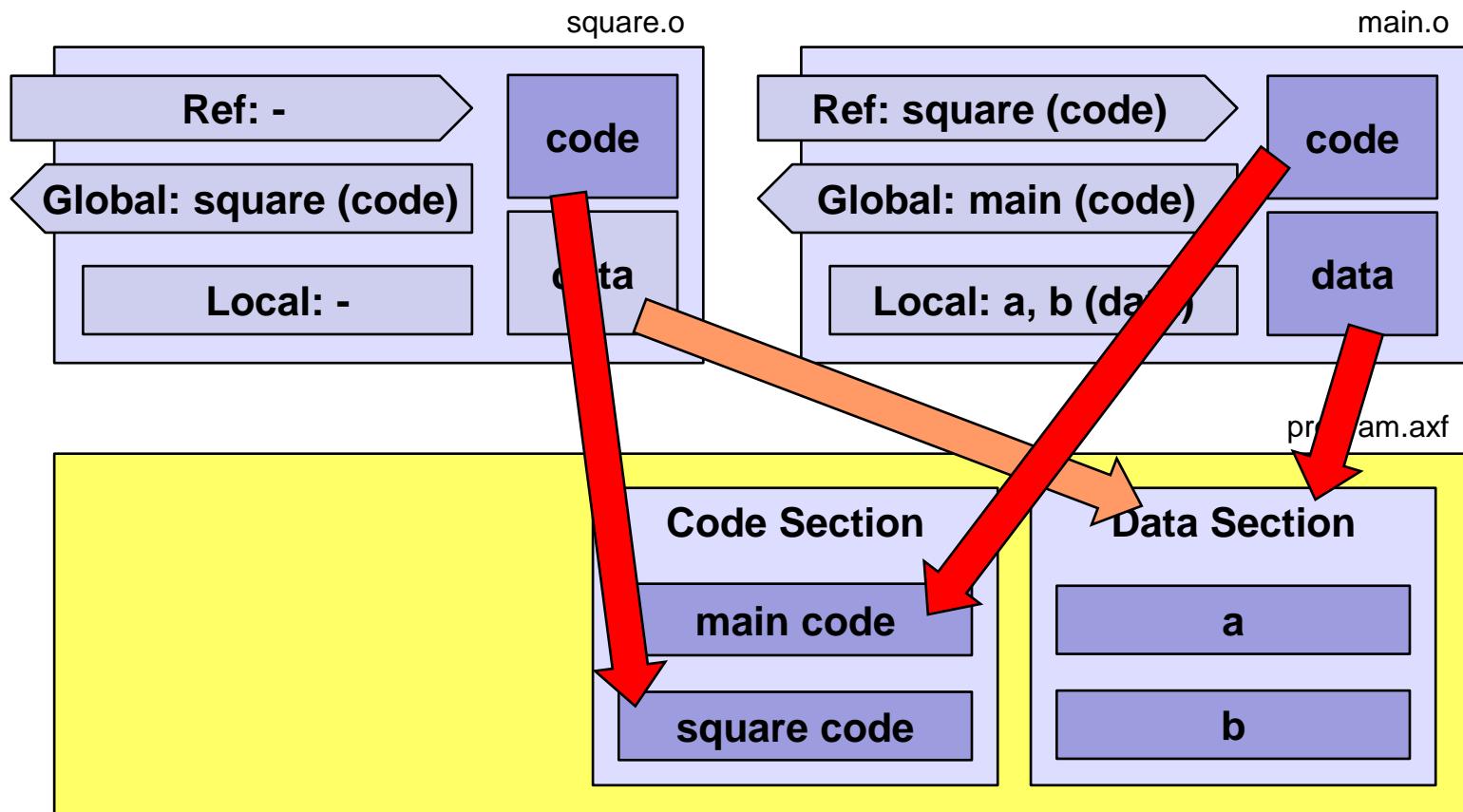
- **Merge object file data sections**
 - Place all data sections of the individual object files into one data section of the executable file
- **Merge object file code sections**
 - Place all code sections of the individual object files into one code section of the executable file
- **Resolve used external symbols**
 - Search missing addresses of used external symbols
- **Relocate addresses**
 - Adjust used addresses since merging the sections invalidated the original addresses¹⁾

1) The linker places the sections depending on the target system, the given command line arguments and the scatter file (if given). A scatter file tells at which absolute addresses the code and data sections are placed. In an OS-hosted environment, addresses for code and data sections are at a fixed “virtual” address. The loader then places the sections into suitable virtual process memory.

Tasks of a Linker

■ Merge data sections and code sections of the modules

- Place one section after the other
 - Note: in this example, square.o has no data section



Tasks of a Linker – Example

■ Example: Merge code sections

- Merging code sections of main.o and square.o
 - Offset for first code section is **0x00000000** (main.o)
 - Offset for next code section is **0x00000001C** (square.o)

0x00000000 B510	PUSH	{r4,lr}	
0x00000002 4804	LDR	r0,[pc,#16]	
0x00000004 6800	LDR	r0,[r0,#0x00]	
0x00000006 f7fffffe	BL	square	
0x0000000A 4903	LDR	r1,[pc,#12]	
0x0000000C 6809	LDR	r1,[r1,#0x00]	
0x0000000E 1844	ADDS	r4,r0,r1	
0x00000010 2000	MOVS	r0,#0x00	
0x00000012 BD10	POP	{r4,pc}	
0x00000014 00000000	DCD	0x00000000	
0x00000018 00000000	DCD	0x00000000	
 			code (main.o)
0x00000001C 4601	MOV	r1,r0	
0x0000001E 4608	MOV	r0,r1	
0x00000020 4348	MULS	r0,r1,r0	
0x00000022 4770	BX	lr	
			code (square.o)

- No resolution nor relocation done yet

Tasks of a Linker – Example

■ Example: Merge data sections

- Merging data sections of main.o and square.o
 - Offset for first data section is **0x00000000** (main.o)
 - There is no further data section (square.o has no global data)

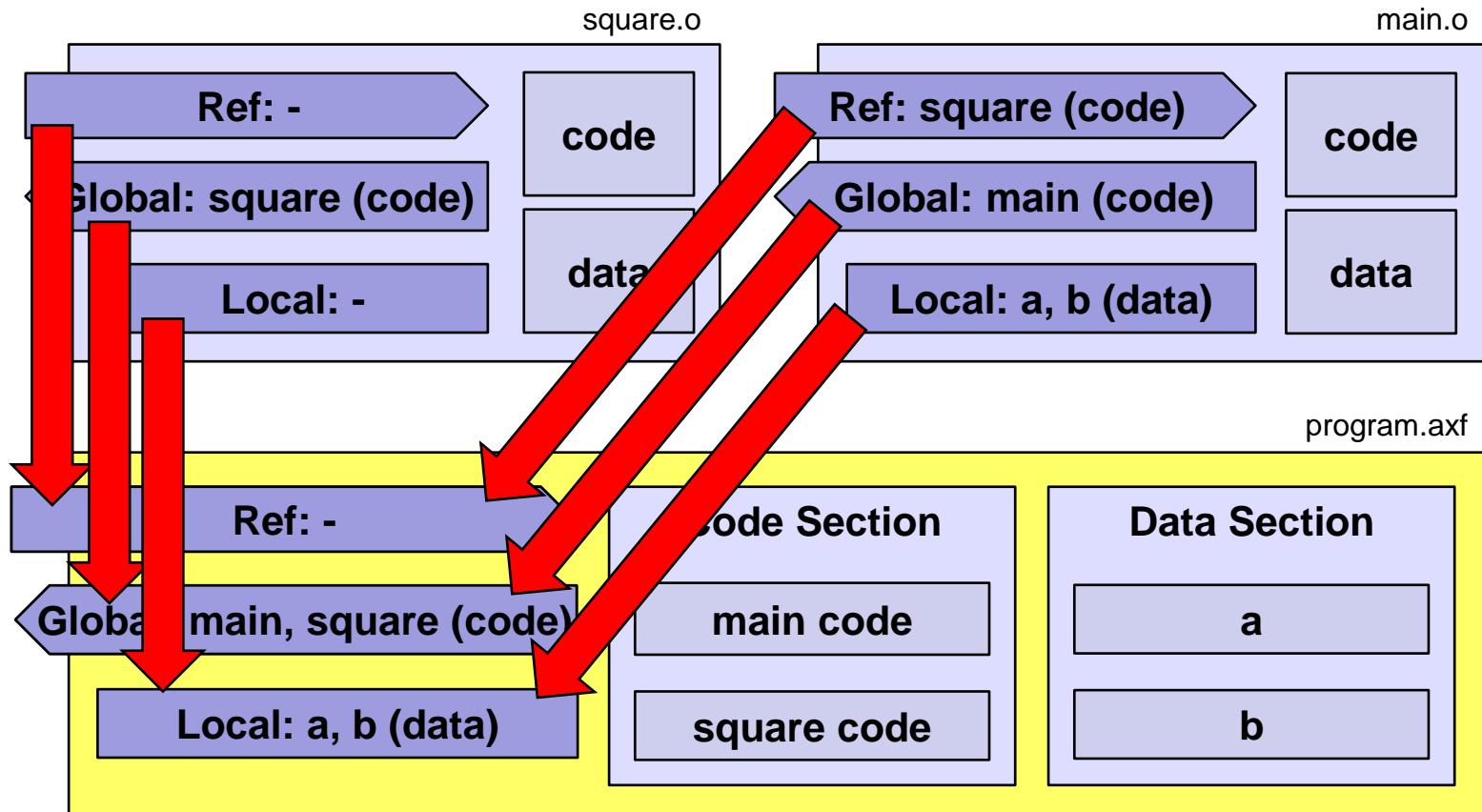
0x00000000 05000000 DCD 5	data (main.o)
0x00000004 07000000 DCD 7	

- No resolution nor relocation done yet

Tasks of a Linker – Resolution

■ Resolve referenced symbols

- Merge the symbol tables
- Resolve references within symbol table



Tasks of a Linker – Example

■ Example: Resolve symbols

- Merging symbol table sections of main.o and square.o

#	Symbol Name	Value	Bind	Sec	Type	Vis	Size	
<hr/>								
7	a	0x00000000	Lc	4	Data	De	0x4	
8	b	0x00000004	Lc	4	Data	De	0x4	
11	main	0x00000001	Gb	1	Code	Hi	0x14	
12	square	0x00000000	Gb	Ref	Code	Hi		

#	Symbol Name	Value	Bind	Sec	Type	Vis	Size	
<hr/>								
6	square	0x00000001	Gb	1	Code	Hi	0x8	

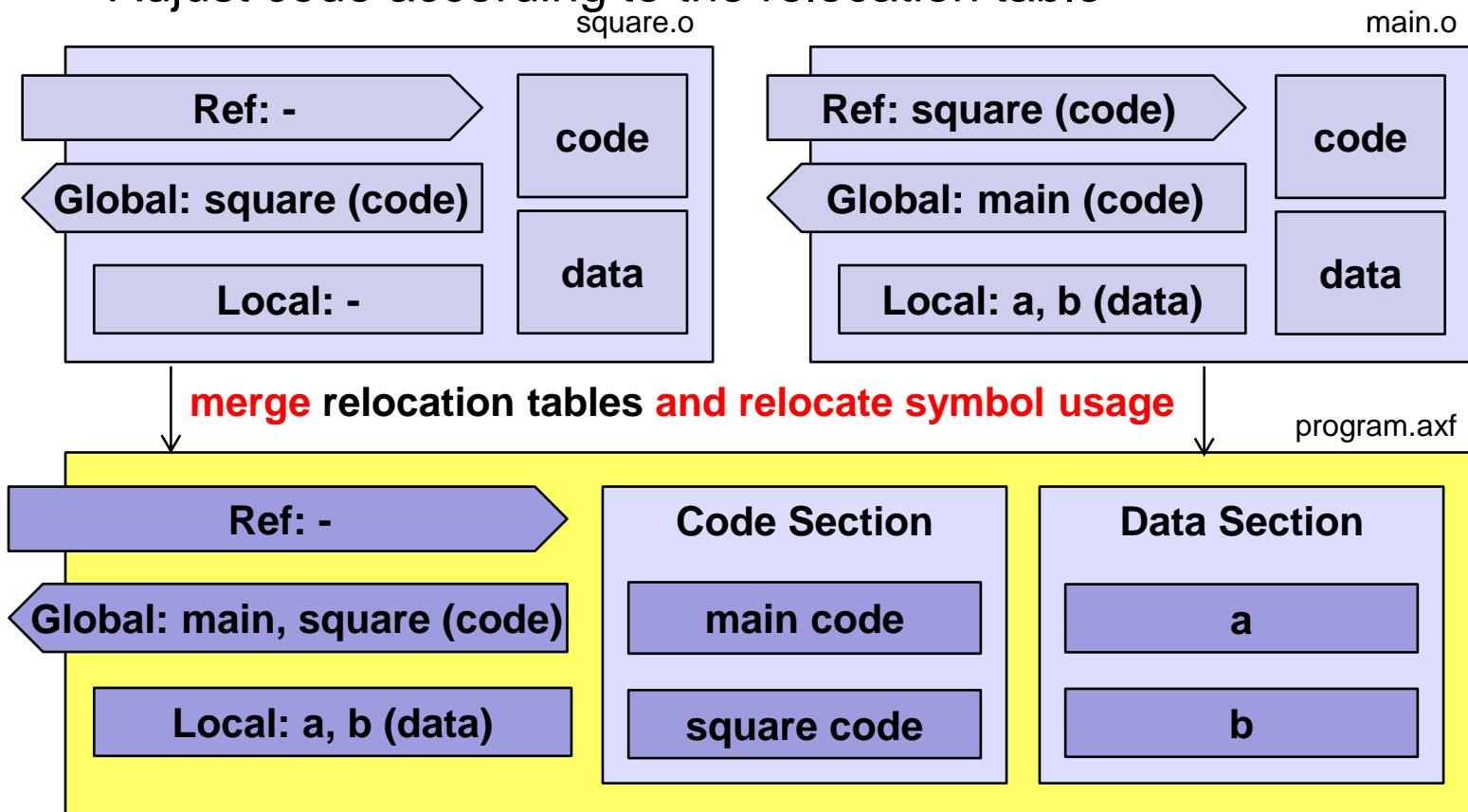
#	Symbol Name	Value	Bind	Sec	Type	Vis	Size	
<hr/>								
20	a	0x00000000	Lc	4	Data	De	0x4	(main.o)
21	b	0x00000004	Lc	4	Data	De	0x4	(main.o)
186	main	0x00000001	Gb	1	Code	Hi	0x14	(main.o)
187	square	0x00000000	Gb	1	Code	Hi	0x8	(square.o)

- The relative values of the symbols within the modules are not yet relocated to global addresses. Therefore, the linker needs to remember for which module/section the relative address is given
- No relocation done yet

Tasks of a Linker – Relocation

■ Relocate usage of symbols

- Merge the relocation tables
- Relocate code and data section, symbols, relocation table
- Adjust code according to the relocation table



Tasks of a Linker – Example

■ Example: Relocate sections and usage of symbols (I)

- Merging relocation table sections of main.o and square.o
 - square.o has no relocation table

#	Offset	Relocation Type	Wrt Symbol	merged relocation
0	0x00000006	10 R_ARM_THM_CALL	12 square	(main.o)
1	0x00000014	2 R_ARM_ABS32	7 a	(main.o)
2	0x00000018	2 R_ARM_ABS32	8 b	(main.o)

- The relative address of the relocation table within the modules is not yet adjusted to global addresses, therefore, needs to remember for which module and which section the relative address is given

Tasks of a Linker – Example

■ Example: Relocate sections and usage of symbols (II)

1) Relocate

- Sections
- Symbols
- Relocation offsets

Relocation calculations

- new value = global base + merge offset + module relative offset
- E.g. symbol `b`:
 - ▶ global base = internal SRAM = 0x20000000
 - ▶ merge offset = 1st in merged data section = 0x00000000
 - ▶ module relative offset = b is the 2nd variable after a = 0x00000004
 - ▶ new value for symbol b = 0x20000004
- E.g. symbol `square` if user code (like main) starts at 0x08000254
 - ▶ 0x08000254 + 0x0000001C + 0x00000000 = 0x08000270

2) Adjust the code according to the relocation table

Tasks of a Linker – Example

■ Example: Relocate sections and usage of symbols (III)

- Relocated code sections

```
0x08000254 B510    PUSH    {r4,lr}          ; main
0x08000256 4804    LDR     r0,[pc,#16]
...
0x08000270 4601    MOV     r1,r0          ; square
0x08000272 4608    MOV     r0,r1
...
```

- Relocated data sections

```
0x20000000 00000005 DCD    5          ; value of a
0x20000004 00000007 DCD    7          ; value of b
```

- Relocated symbols

20 a	0x20000000	Lc	4	Data	De	0x4
21 b	0x20000004	Lc	4	Data	De	0x4
186 main	0x08000255	Gb	1	Code	Hi	0x14
187 square	0x08000270	Gb	1	Code	Hi	0x8

- Relocated relocation table entries

0 0x0800025A	10 R_ARM_THM_CALL	12 square
1 0x08000268	2 R_ARM_ABS32	7 a
2 0x0800026C	2 R_ARM_ABS32	8 b

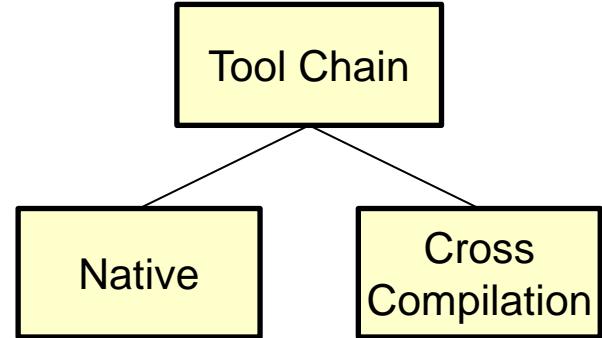
- Adjusted code locations according to relocation table

```
...
0x0800025A F000F809 BL.W    square      ; 0x08000270
...
0x08000268 20000000 DCD    0x20000000
0x0800026C 20000004 DCD    0x20000004
```

Tool Chain, Libraries, Debugging

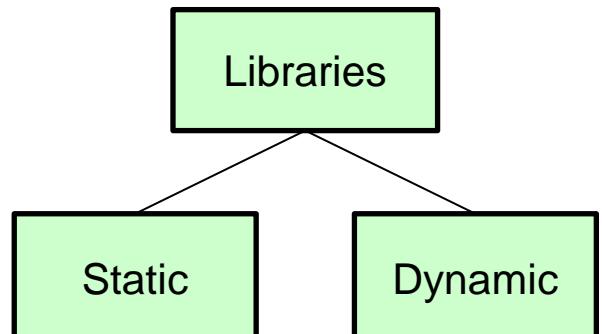
■ Tool chain

- Native tool chain
- Cross compiler tool chain



■ Libraries

- Libraries
 - Collection of object files
- Static libraries
 - Linked into the executable at link time
- Dynamic or shared libraries
 - Executable is linked at loading time with the shared library



■ Debugging

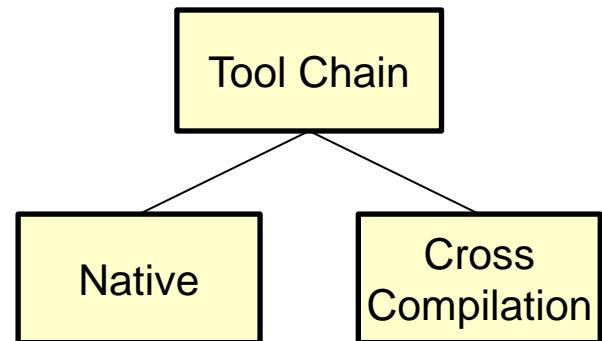
- Single step and breakpoints
- Source level debugger

A screenshot of a debugger interface. On the left, there is a memory dump window showing memory addresses 123 to 128. A red dot is placed over address 123. To the right of the dump is a column of assembly code:

```
123 REG_TIMx_PSC(TIM4_BASE) = (F_84MHZ / F_10KHZ)-1;  
124 REG_TIMx_ARR(TIM4_BASE) = F_10KHZ - 1;  
125 REG_TIMx_CR1(TIM4_BASE) = 0x0;  
126 REG_TIMx_CR1(TIM4_BASE) |= DOWNCOUNT;  
127 REG_TIMx_DIER(TIM4_BASE) |= UIE;  
128 REG_TIMx_CR1(TIM4_BASE) |= CEN;
```

■ Tool chain

- Minimal view
 - The set of tools that is required to create from source code an executable for a given environment
- Native tool chain
 - Builds for the same architecture where it runs on
- Cross compiler tool chain
 - Builds for another architecture than the one it runs on
 - E.g. build in KEIL (on Windows) for the CT Board (bare-metal ARM)
- Professional view: there is more than the “compiler & linker”:
 - Editing tools (IDE), revision control tools, documentation tools, testing tools, build tools, deployment tools, issue tracking tools, ...



Cross Compiler Tool Chain

■ KEIL IDE – Integrated Development Environment

- UI Frontend for editing, compiling and debugging



■ Cross compiler tool chain

- Produces executable programs for a target system which is different from the host system of the tool chain (e.g. compile on a Windows PC for an ARM platform).
- Behind the scene, KEIL IDE employs a cross compilation tool chain

– armcc	ARM C/C++ Compiler (including Preprocessor)
– armasm	ARM Assembler Compiler
– armar	ARM Library manager
– armlink	ARM Linker
– fromelf	ARM Image conversion and dumper tool

■ Libraries in general

- Collection of object files
- May speedup linking
 - May provide an overall prepared (sorted) symbol table
- Linking with libraries may result in smaller code
 - Libraries may provide only the really needed parts of the sections
 - Linking with plain object files always links all and the whole sections
- Created by a librarian tool (e.g. armar for our environment)
- May replace one library by another one
 - E.g. at evaluation time have a working model, at production time have a high-performance library of the same functionality

■ Static libraries

- Executable is completely linked with a static library at link time
- The resulting executable is self contained
 - No need for any other libraries at run time
- Benefit
 - Self contained
 - No version issues in the run environment
 - No support needed from any hosting OS
- Drawback
 - Larger executables compared to dynamically linked libraries
 - No possibility to share common code between different executables
 - Cannot replace broken shared code with a new version of the library

■ KEIL/ARM

- Static libraries are used in the ARM cross compilation environment

■ Dynamic or shared libraries

- Executable is not linked with a dynamic library at link time
- The resulting executable is not self contained
 - Needs other libraries at run time
 - Loader of hosting OS links at load time with the shared libraries
- Benefit
 - Smaller executables compared with static libraries
 - Can replace shared libraries
- Drawback
 - May result in versioning problems at load time
 - ▶ Well known “DLL-Hell” from MS Windows environment

■ Windows/Unix/OSx

- With PC OS support you have generally both libraries
 - Static: libX.a, dynamic: libX.so (Unix/Linux/OSx), libX.dll (Windows)

■ Single stepping

- Support by the HW (stops processor, provides register access)
- Support by SW (swap instructions with a breakpoint instruction)

■ Source level debugging

- Source level debugging needs mapping between
 - Machine address and source code line
 - Memory locations and source code types
- Mapping information is often also provided in object files (e.g. in the ELF files)
 - Also depends on all linking steps (merging section, resolve symbols, relocate symbol usage)
 - On Windows, this information is provided in a separate file (PDB)

Conclusions

■ Modular programming

- Crucial concept of software development
- There is no golden rule but established practice
- C supports this by use of header files and compilation into object files

■ Linker

- Combines object files into executable by merging sections, resolve referenced symbols and relocating all symbols and code
- Is ignorant of the used programming language

■ Tools

- Tool chains and further tools build the working environment

■ Debugging

- Source-level debugging is a crucial tool to analyze and fix bugs

Example Source Code

```
// square.h
...
// declaration of square
uint32_t square(uint32_t v);
...
```

```
// square.c
#include "square.h"

// definition of square
uint32_t square(uint32_t v)
{
    return v*v;
}
```

square = *external linkage*

```
// main.c
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
    uint32_t res;
    res = square(a) + b; //use
    ...
}
```

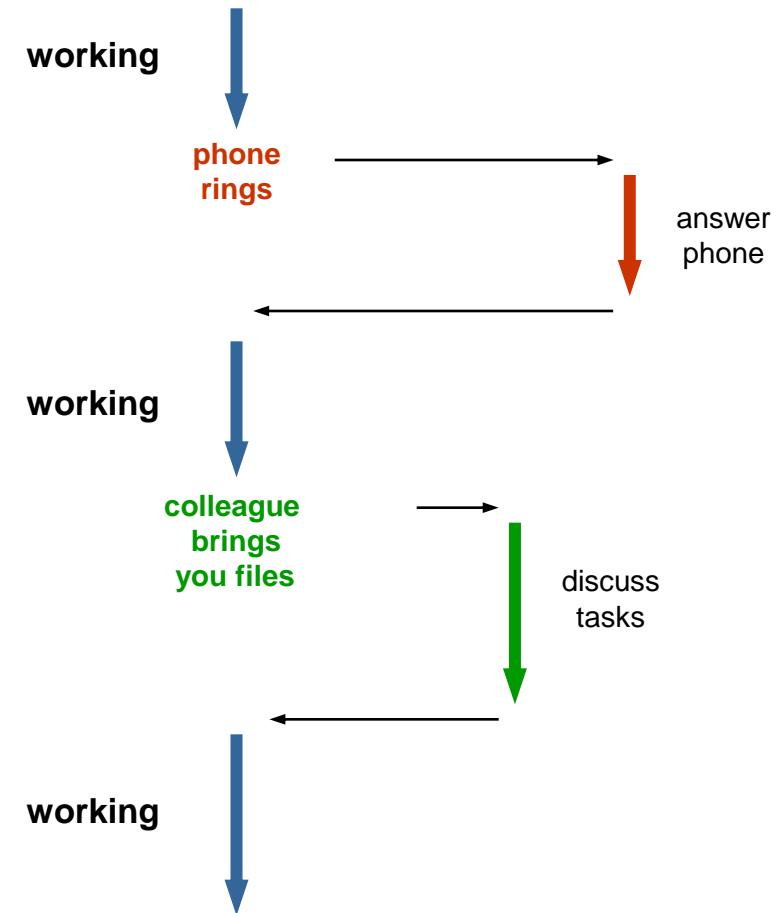
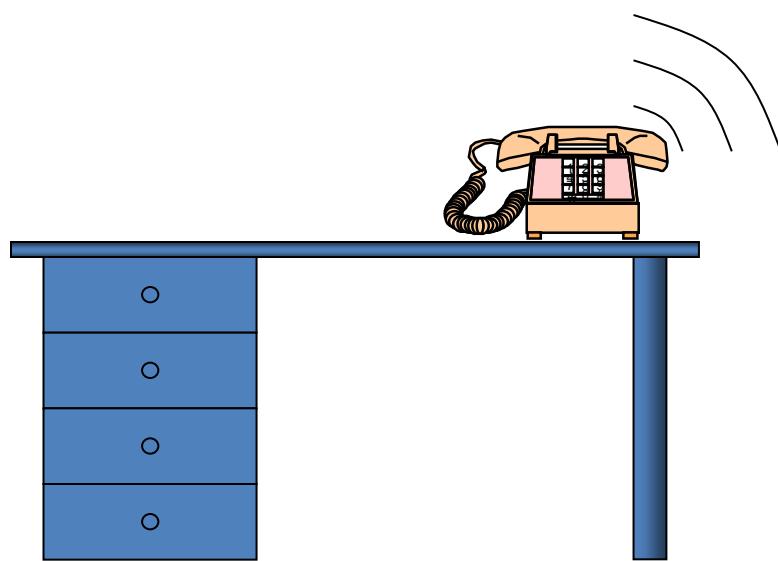
a = *internal linkage*
b = *internal linkage*
main = *external linkage*
res = *no linkage*
square = *external linkage*

Exceptional Control Flow

Computer Engineering 1

Motivation

■ You are working at your desk!



- **Computers work concurrently on several tasks**
 - Often wait until result is available, e.g. network request
 - Fill idle time with useful tasks → efficient use of processing power
 - Interrupts signal, that result of a request is available
 - Requires switching between programs
- **Unexpected events – exceptions**
 - Division by 0, unaligned accesses, etc.
 - Must be caught and corresponding actions need to be triggered
- **Time critical events with high priority**
 - Interrupt running program → process event

Agenda

- **Polling**
- **Interrupt-Driven I/O**
- **Exceptions Cortex-M3/M4**
- **Interrupt-System Cortex-M3/M4**
- **Vector Table**
- **Storing the Context**
- **External Interrupt Pins**
- **Interrupt Control**
- **Nested Exceptions**
- **Special Interrupt Situations**
- **CMSIS Functions**
- **Data Consistency**

The lecture does not cover all the features of Cortex-M3/4 exception processing, in particular the following simplifications apply:

- No distinction between handler mode and thread mode
- No distinction between preempt priority and sub-priority fields
- Tail chaining and late arrival are not covered
- Instructions that are abandoned and restarted/resumed after interrupt service are not covered (e.g. LDM)

Learning Objectives

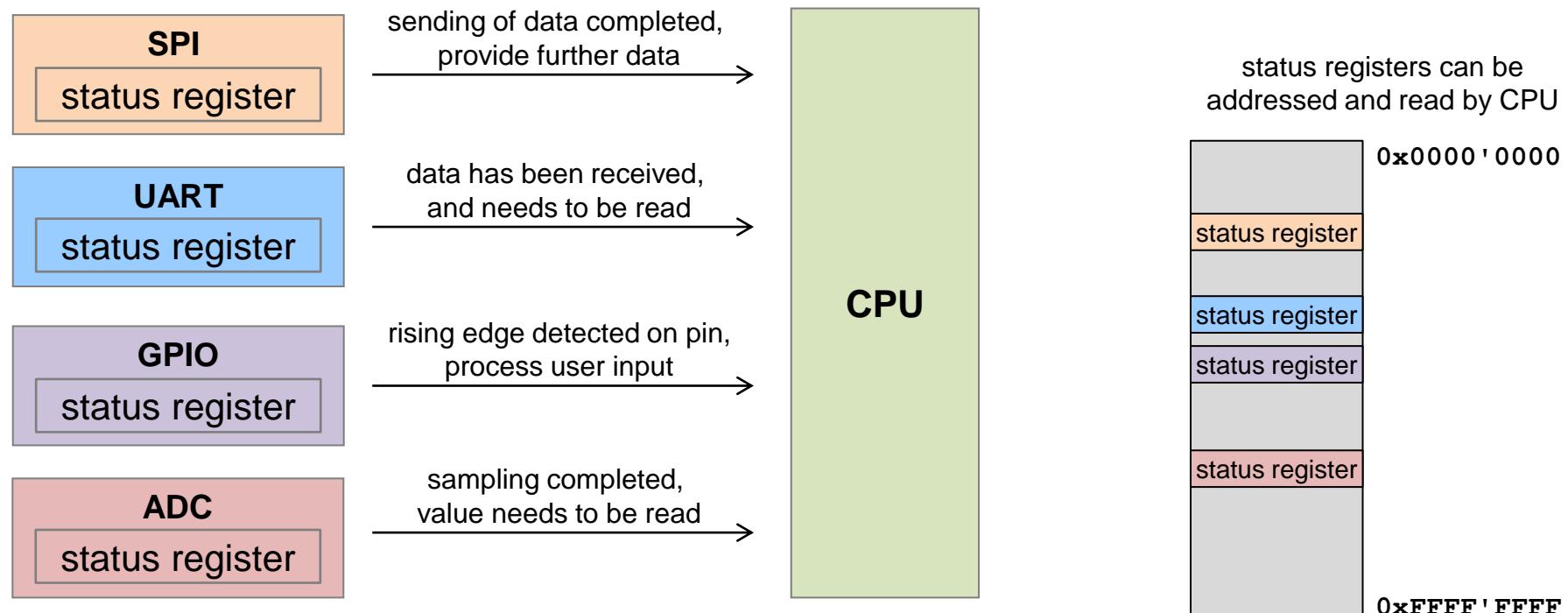
At the end of this lesson you will be able

- to explain advantages and disadvantages of polling and interrupt-driven I/O
- to distinguish the different types of exceptions on a Cortex-M3/M4
- to explain how the Cortex-M3/M4 recognizes and processes exceptions
- to explain the vector table of the Cortex-M3/M4
- to understand the basic functionality of the Nested Vectored Interrupt Controller (NVIC)
 - to enable and disable interrupts
 - to set and clear interrupts by software
 - to prioritize exceptions
 - to know how programmed priorities influence preemption of service routines
 - to explain how simultaneously pending interrupts are processed
- to implement a simple interrupt service routine in Cortex-M assembly
- to explain potential data consistency issues due to interrupts and to give potential examples

CPU Has to Act on Events

■ Peripherals signal events to CPU

- Something happened that requires a CPU service → call a subroutine

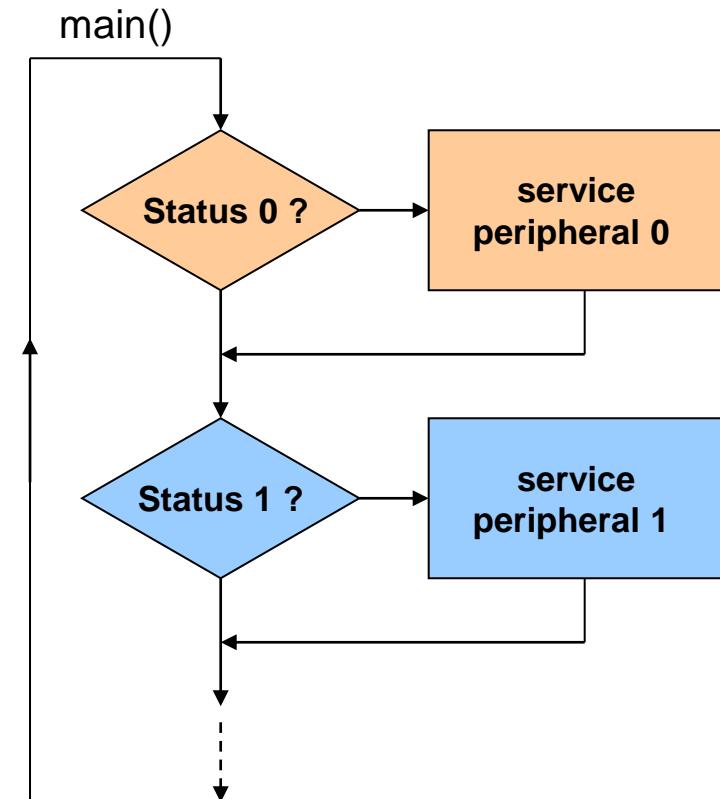


SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
GPIO	General Purpose Input Output
ADC	Analog-Digital Converter

■ Periodic Query of Status Information

- Reading of status registers in loop
- Synchronous with main program
- Advantages
 - Simple and straightforward
 - Implicit synchronization
 - Deterministic
 - No additional interrupt logic required
- Disadvantages
 - Busy wait → wastes CPU time
 - Reduced throughput
 - Long reaction times

in case of many I/O devices or
if the CPU is working on other tasks



¹⁾ to poll → abfragen

Polling

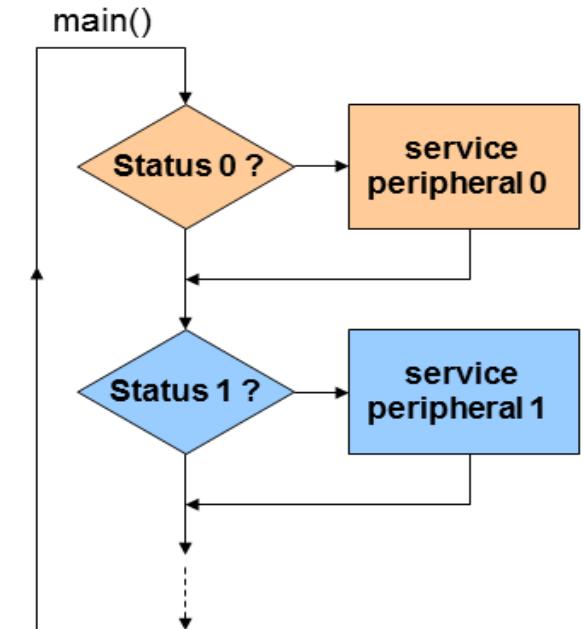
■ Architecture

```
while(1)
{
    if ( read_byte(ADDR_BUTTON_A) ) {
        execute_task_A();
    }

    if ( read_byte(ADDR_BUTTON_B) ) {
        execute_task_B();
    }

    if ( read_byte(ADDR_BUTTON_C) ) {
        execute_task_C();
    }

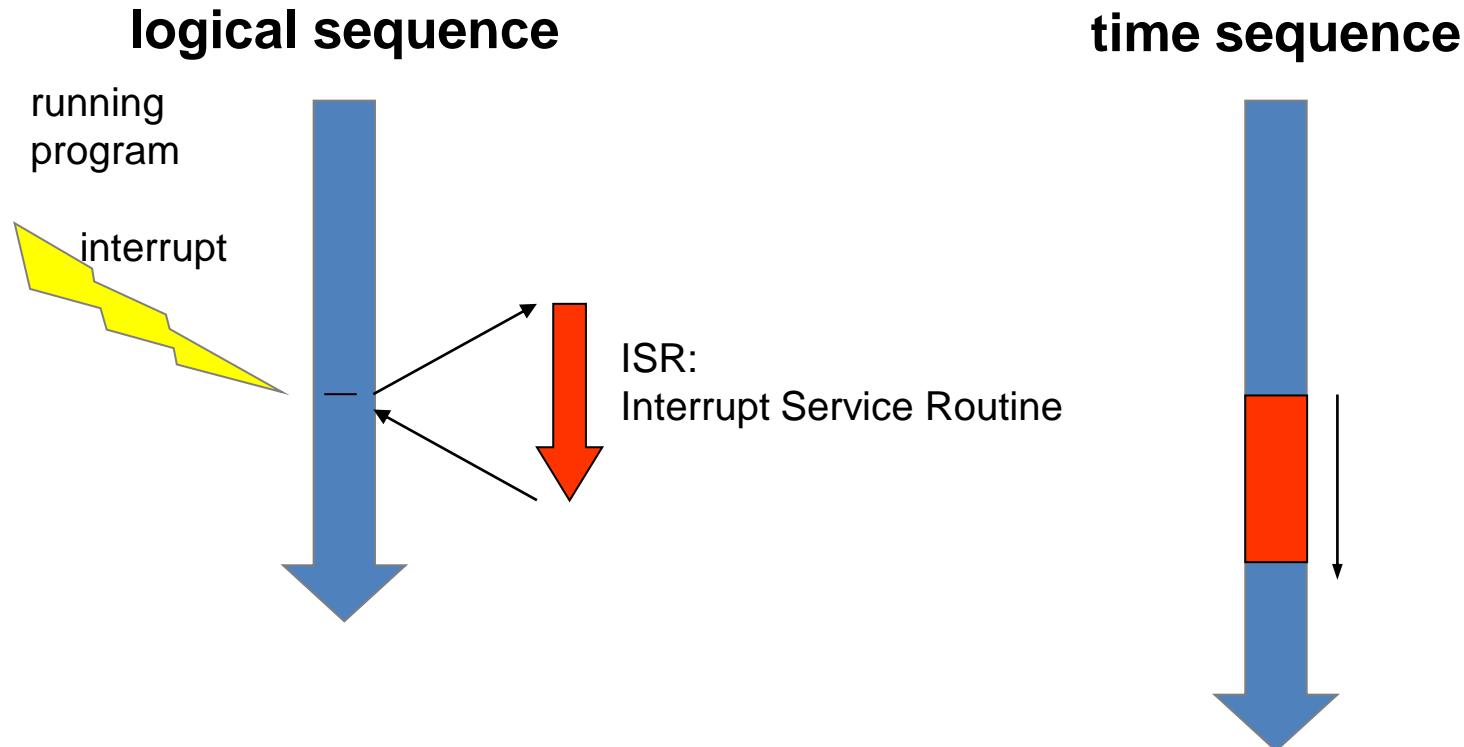
    ...
}
```



Interrupt-Driven I/O

■ Alternative to polling

Interrupt: Sudden change of program flow due to an event



Interrupt-Driven I/O

■ Main program

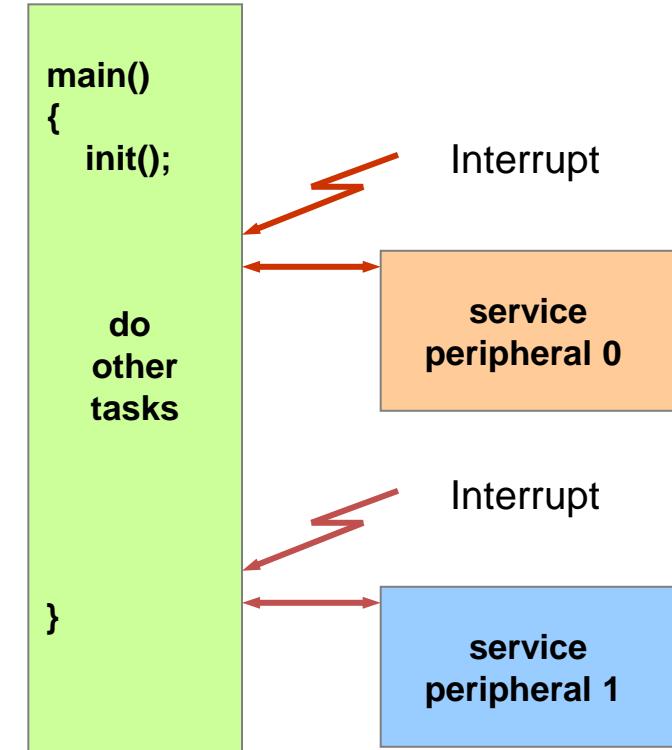
- Initializes peripherals
- Afterwards it executes other tasks
- Peripherals signal when they require software attention (phone call analogy)
- Events interrupt program execution

■ Advantage

- No busy wait → better use of CPU time
- Short reaction times

■ Disadvantages

- No synchronization (between main program and ISRs)
- Difficult debugging



■ **Interrupt sources: IRQ0 – IRQ239**

- Peripherals¹⁾ signal to CPU that an event needs immediate attention
- Can alternatively be generated by software request
- Asynchronous to instruction execution

■ **System exceptions**

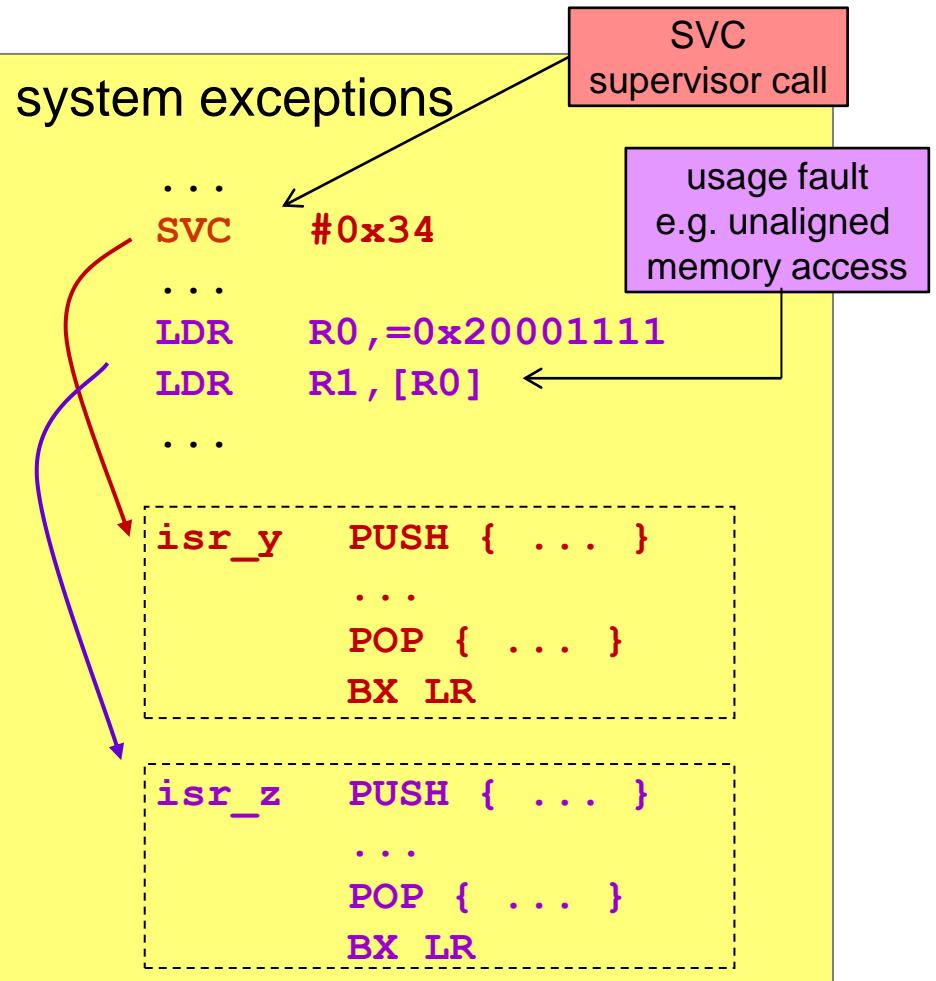
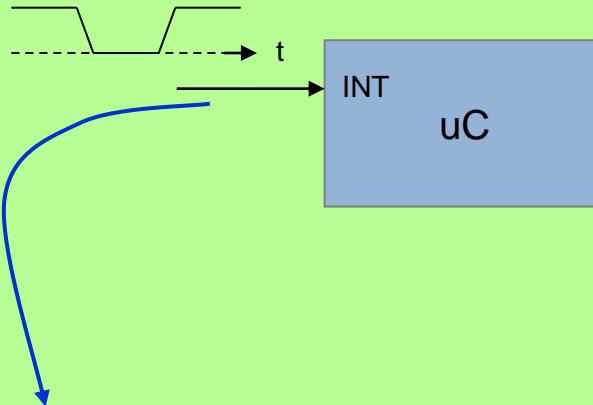
- Reset
 - Restart of processor
- NMI – Non-maskable Interrupt
 - Condition that cannot be ignored, e.g. a critical hardware error
- Faults
 - E.g. undefined instructions, unaligned accesses, etc.
- System Level Calls
 - Operating system (OS) calls – Instruction SVC and PendSV

¹⁾ GPIO (general purpose input/output), timer, serial interfaces, etc

Exceptions Cortex-M3/M4

■ Examples for Exceptions

interrupts → IRQ0 – IRQ239



Exceptions Cortex-M3/M4

■ System Exceptions

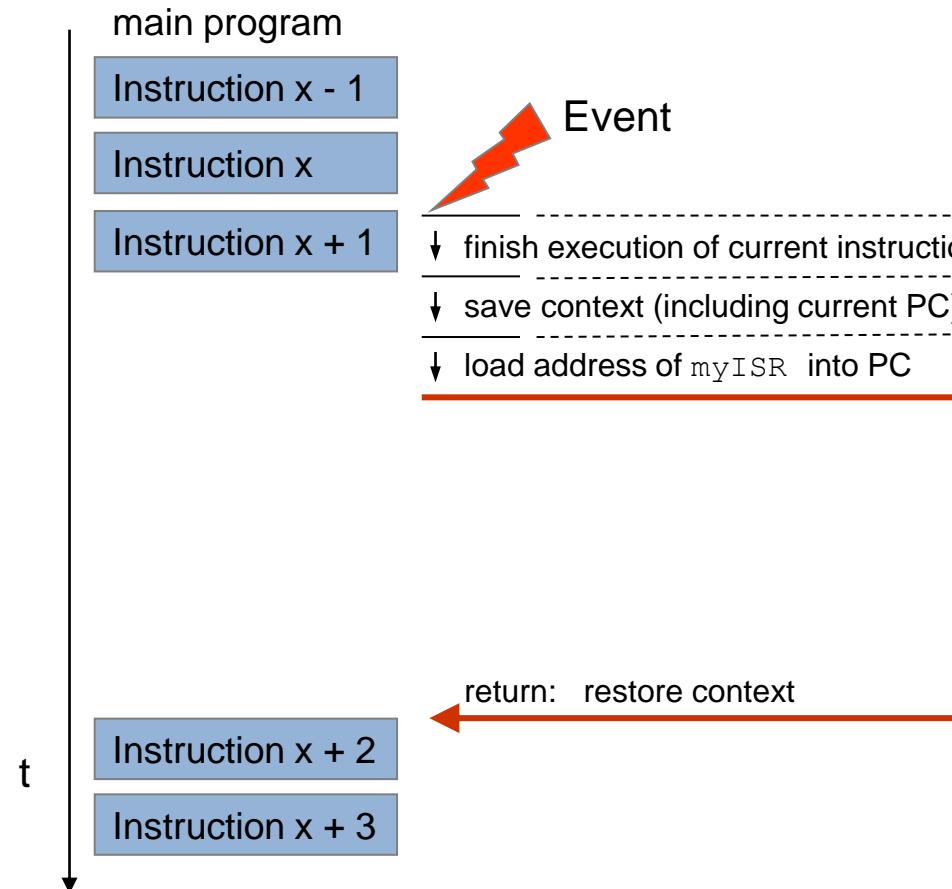
Exception Number	Exception Type	Priority	Description
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	-1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called <i>prefetch abort</i> if it is an instruction fetch or <i>data abort</i> if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7-10	Reserved	NA	-
11	SVCALL	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	-
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

source: "The definitive Guide to the Cortex-M3"

Exceptions Cortex-M3/M4

■ Interrupts: Event calls ISR

- Change of program flow



```
main    proc
prog    ...      ; main program
...
B  prog

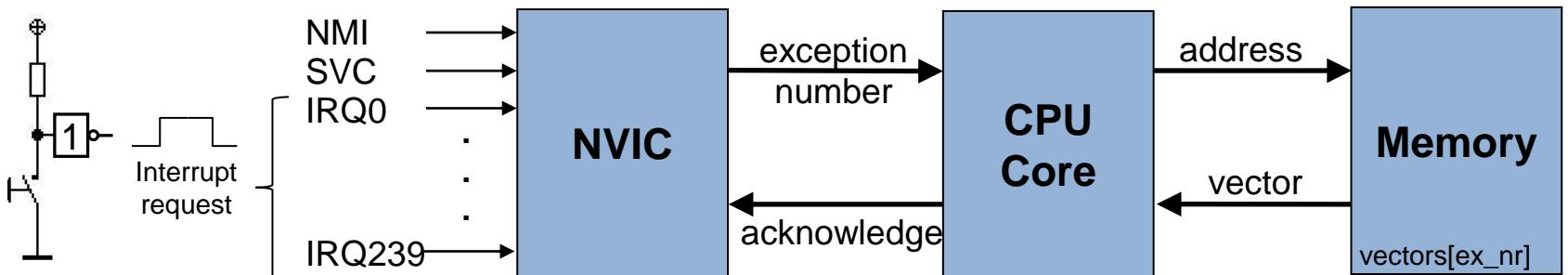
myISR
...
...      ; service the
...      ; event
BX LR
```

```
Instruction myISR
Instruction myISR + 1
...
Instruction myISR + N
Instruction BX LR
```

Interrupt-System Cortex-M3/M4

■ Nested Vectored Interrupt Controller (NVIC)

- 240 sources can trigger exception → high level signal on IRQx
- Forwards respective exception number to CPU



■ CPU

- Calculates vector table address based on exception number
- Uses address to read vector from memory
- Stores context on stack
- Loads vector into PC → branch to ISR

Simultaneous exceptions from different sources will be covered later

Vector Table (Cortex-M3/M4)

■ Which ISR Shall the Processor Call?

- Each exception has a different ISR

Memory Addr.	31	0	Exception Nr.
0x0000'0000	Top of Stack		0
0x0000'0004	Reset		1
0x0000'0008	NMI		2
0x0000'000C	Hard Fault		3
...
0x0000'002C	SVC		11
...
0x0000'0038	PendSV		14
0x0000'003C	SysTick		15
0x0000'0040	IRQ0		16
0x0000'0044	IRQ1		17
...
0x0000'03FC	IRQ239		255

System Exceptions
1 – 15

Interrupts 0 – 239
IRQn → Exception Nr. (n + 16)

Example:
IRQ3 → Exception Nr. 19

Vector Table (Cortex-M3/M4)

■ Initialization

```
; Vector Table Mapped to Address 0 at Reset
        AREA  RESET, DATA, READONLY

0  __Vectors      DCD    __initial_sp          ; Top of Stack
1          DCD    Reset_Handler           ; Reset Handler
2          DCD    NMI_Handler            ; NMI Handler
3          DCD    HardFault_Handler     ; Hard Fault Handler
...          DCD    ...
...          DCD    ...

; Interrupts
16         DCD    IRQ0_Handler          ; ISR for IRQ0
17         DCD    IRQ1_Handler          ; ISR for IRQ1
...         DCD    ...
```

System Exceptions 15 vectors

Interrupts

```
AREA  SOURCE_CODE, CODE, READONLY
IRQ0_Handler PUSH { ... }

...           ; interrupt service

POP { ... }
BX  LR
```

Storing the Context

■ Interrupt event can take place at any time

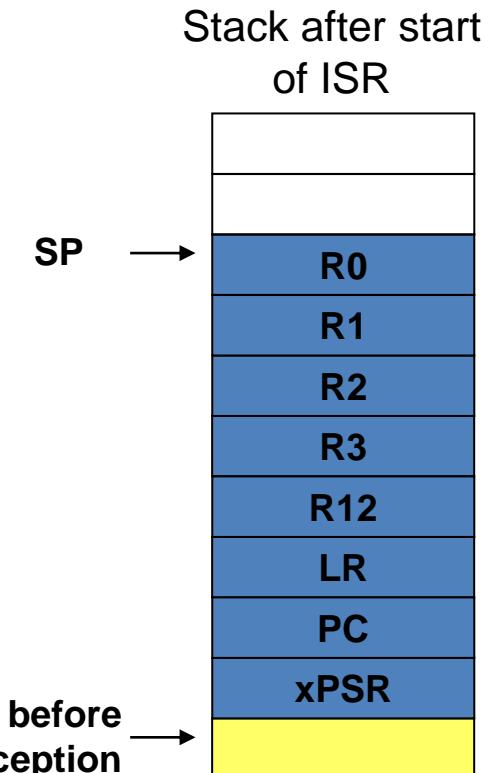
- E.g. between **TST** and **BEQ** instructions
 - ISR call requires automatic save of flags and caller saved registers

■ ISR Call

- Stores xPSR, PC, LR, R12, R0 – R3 on stack
- Stores EXC_RETURN¹⁾ to LR

■ ISR Return

- Use **BX LR** or **POP {..., PC}**²⁾
- Loading EXC_RETURN¹⁾ into PC
 - restores R0 – R3, R12, LR, PC and xPSR from stack



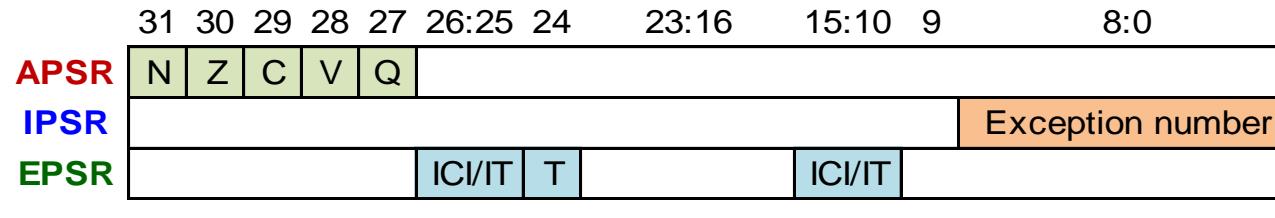
¹⁾ EXC_RETURN = 0xFFFF'FFF9

²⁾ if LR has been saved with PUSH {..., LR} in ISR

Storing the Context

■ Program Status Registers (PSRs)

- **APSR** Application Program Status Register
- **IPSR** Interrupt Program Status Register
- **EPSR** Execution Program Status Register



- **xPSR** Combination of all three PSRs

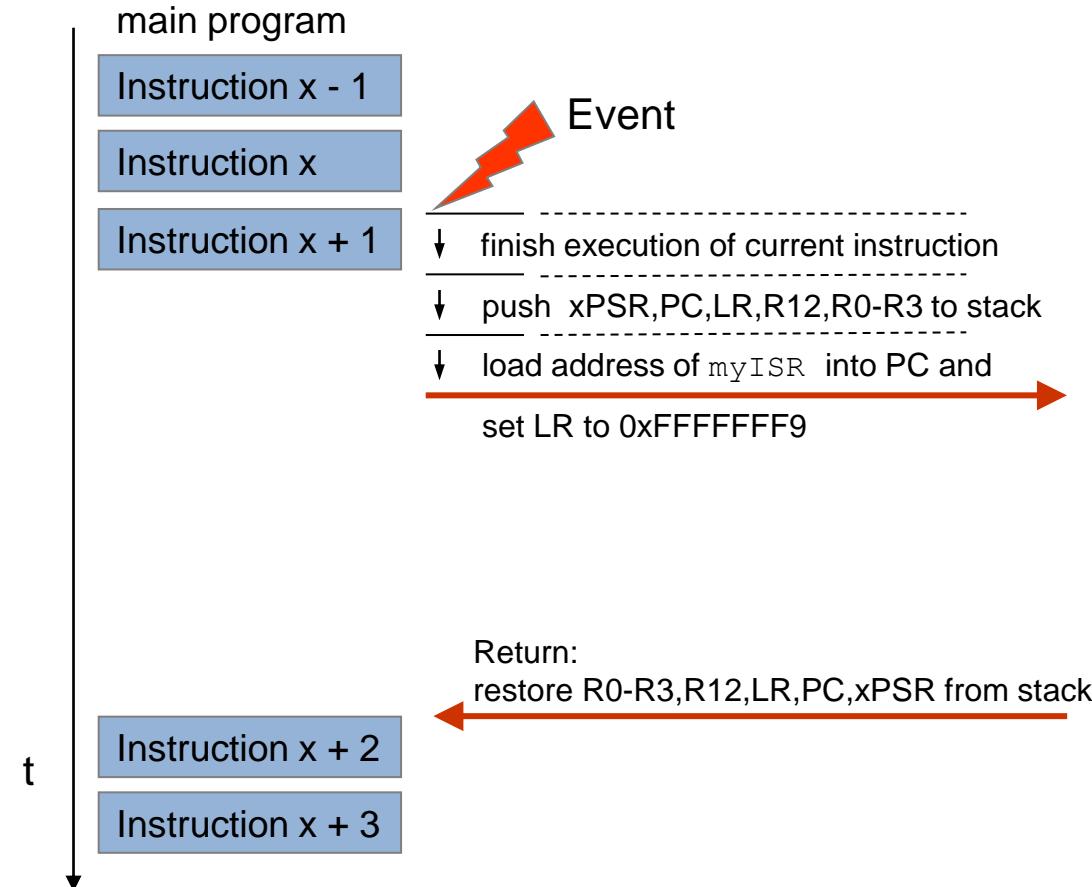
Exception number indicates which exception the processor is handling

ICI/IT Bits used in case of Interrupt-Continuable Instructions e.g. LDM/STM
or conditional execution of instructions (IF-THEN)

T Thumb mode. Always one

Storing the Context

■ Asynchronous event calls ISR



```
main    proc
prog    ...      ; main program
...
B  prog

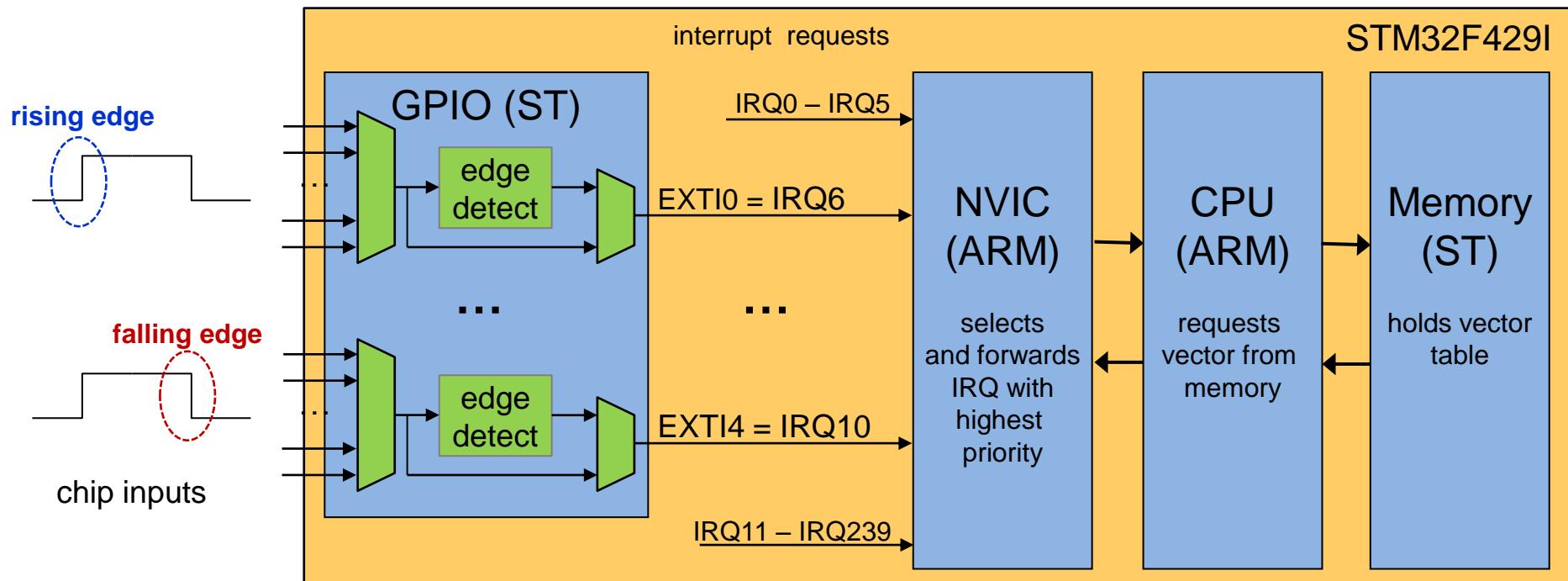
myISR
...
...      ; service the
...      ; event
BX LR

Instruction myISR
Instruction myISR + 1
...
Instruction myISR + N
Instruction BX LR
```

External Interrupt Pins

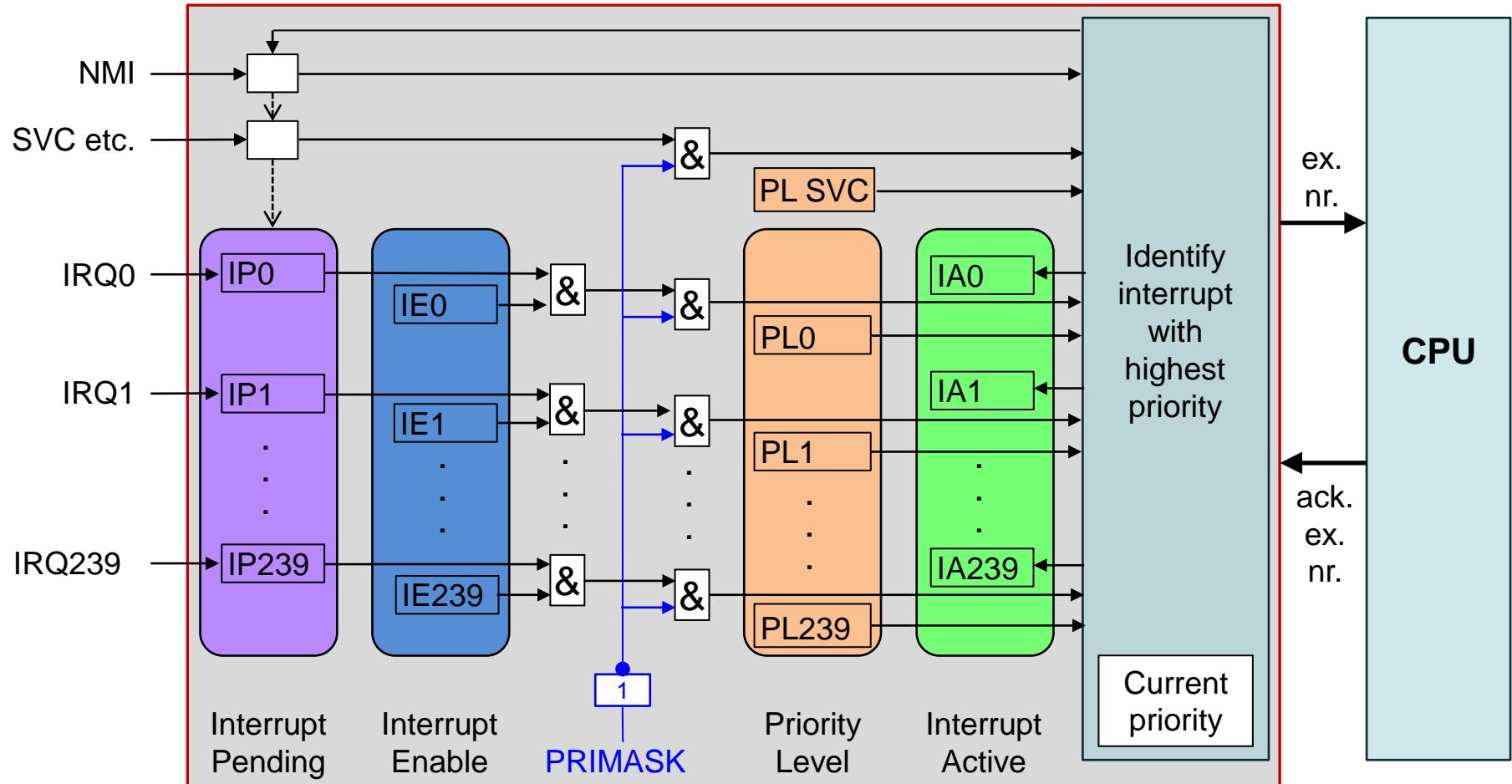
■ Chip Vendor (ST) adds GPIO logic

- EXTI0 through EXTI4 connected to IRQ6 through IRQ10
 - IRQ0 – IRQ5 / IRQ11 – IRQ239 used for other sources e.g. SPI, UART, ADC
- Select chip input for each EXTIx line
- Select level or edge



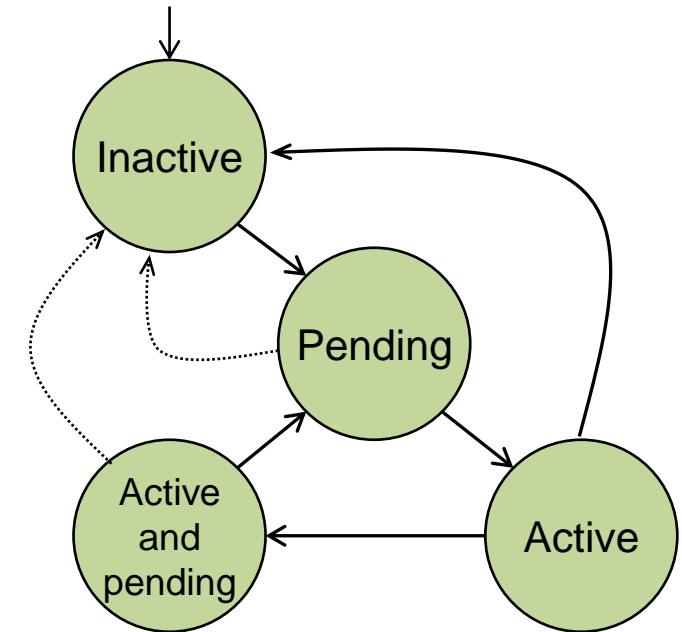
Interrupt Control

■ Nested Vectored Interrupt Controller (NVIC)



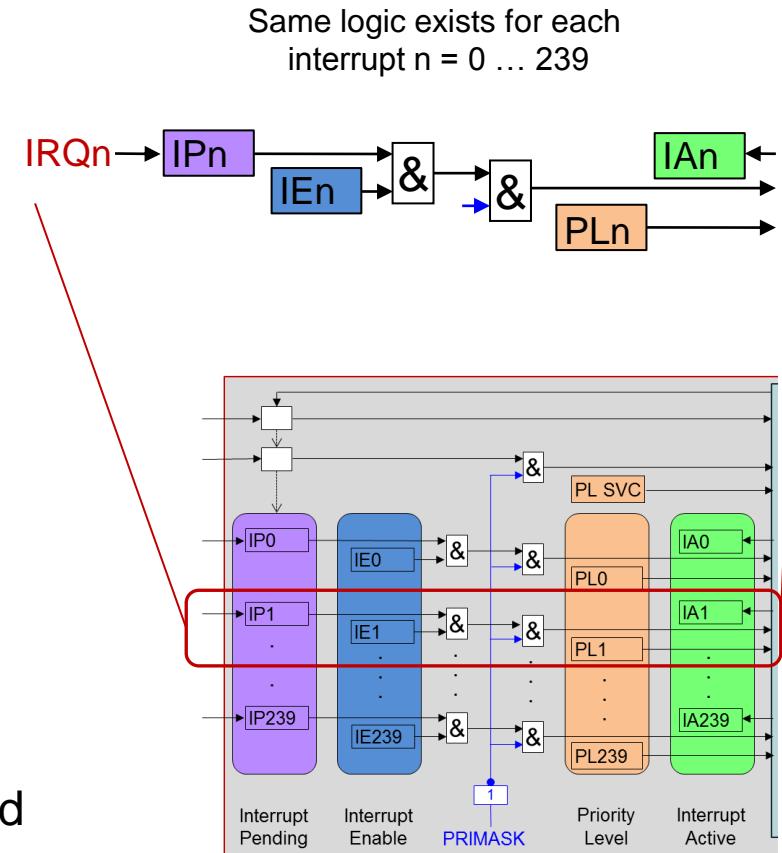
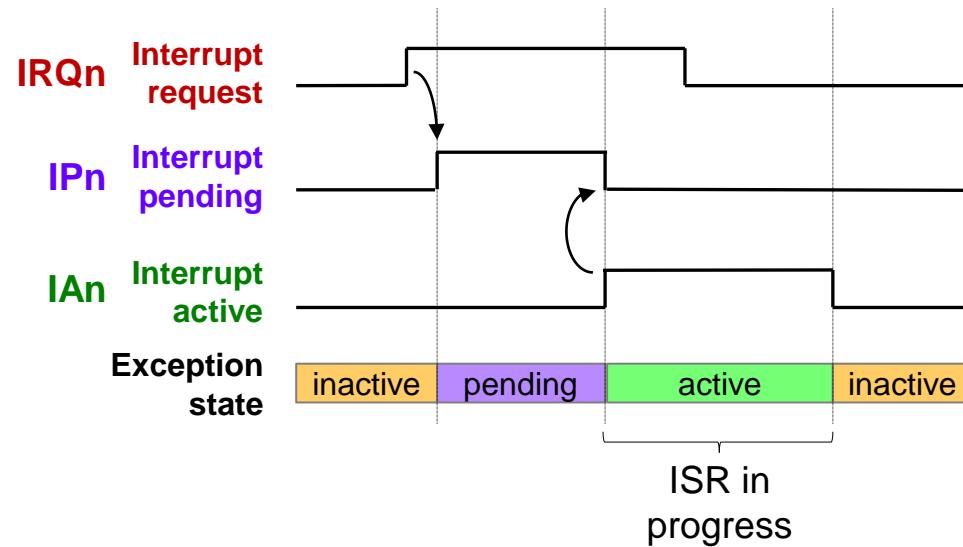
■ Exception States (for each interrupt source, e.g. IRQ17)

- Inactive
 - Exception is not active and not pending
- Pending
 - Exception is waiting to be serviced by CPU
 - E.g. an interrupt event occurred ($\text{IRQn} = 1$) but interrupts are disabled (PRIMASK)
- Active
 - Exception is being serviced by the CPU but has not completed
- Active and pending
 - Exception is being serviced by the CPU and there is a pending exception from the same source



Interrupt Control

■ IRQ Inputs and Pending Behavior



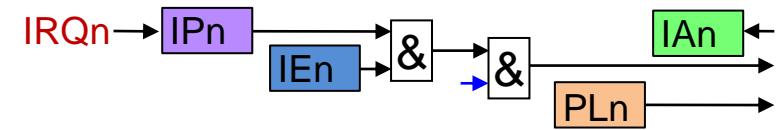
- High level on IRQn sets pending bit (IPn)
- Active Bit (IA)
 - Set as soon as exception is being serviced
 - Resets pending bit

¹⁾ IRQn is set by hardware and usually reset by software

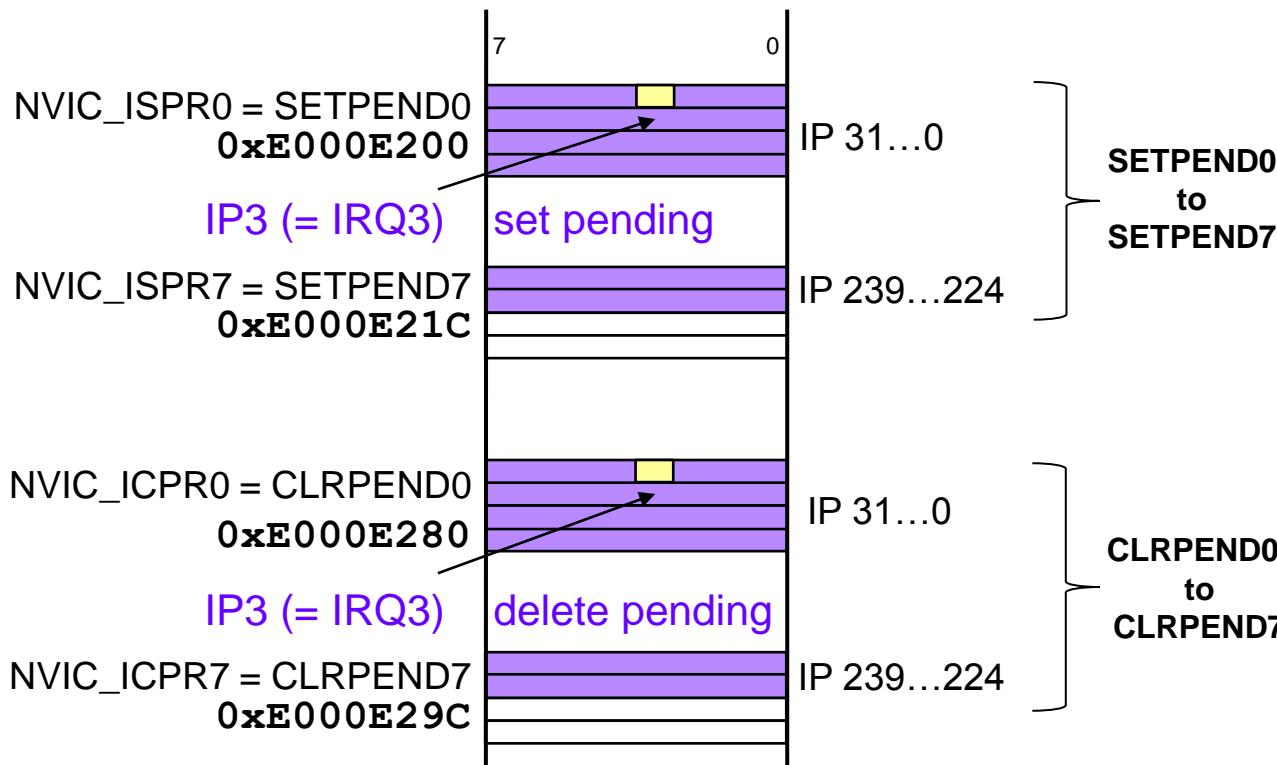
Interrupt Control

■ Interrupt Pending Registers

- Trigger hardware interrupt by software
- Cancel a pending interrupt



→ set pending bit
→ clear pending bit



Trigger IRQ3 by Software

```
SETPEND0 EQU 0xE000E200
...
LDR R0,=SETPEND0
MOVS R1,#0x08
STR R1,[R0]
```

Delete Pending IRQ3

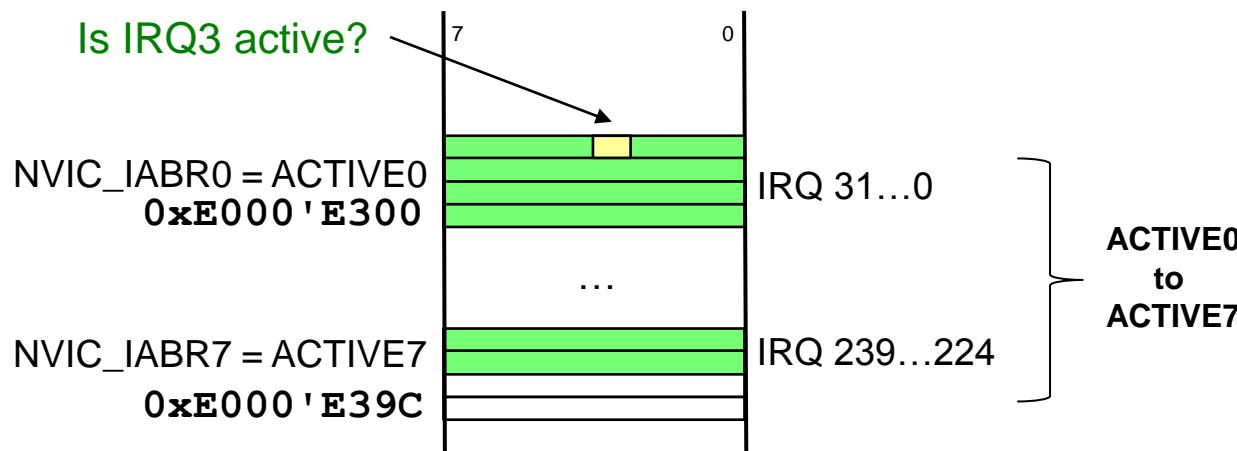
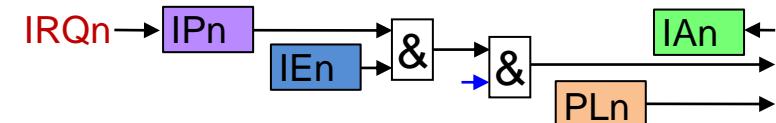
```
CLRPEND0 EQU 0xE000E280
...
LDR R0,=CLRPEND0
MOVS R1,#0x08
STR R1,[R0]
```

ARM and ST use different names for the same register

Interrupt Control

■ Interrupt Active Status Registers

- Read-only
- Corresponding bit is set when ISR starts
- Corresponding bit is cleared when interrupt return is executed



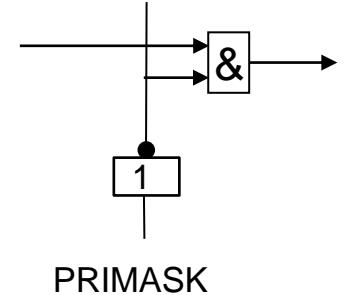
Test if IRQ3 is active

```
ACTIVE0 EQU 0xE000E300
...
LDR R0, =ACTIVE0
MOVS R1, #0x08
LDR R2, [R0]
TST R1, R2
BEQ ...
```

Interrupt Control

■ General Masking of Interrupts

- PRIMASK
 - Single bit controlling all maskable interrupts



- Disable set PRIMASK
- Enable clear PRIMASK

Assembly

`CPSID1) i`
`CPSIE1) i`

C

`_disable_irq();`
`_enable_irq();`

- On reset PRIMASK = 0 → enabled

■ Non-maskable Interrupt (NMI)

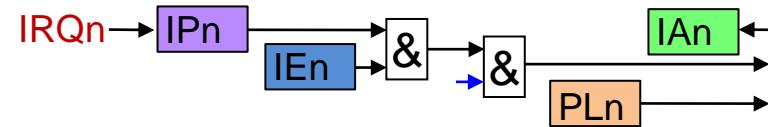
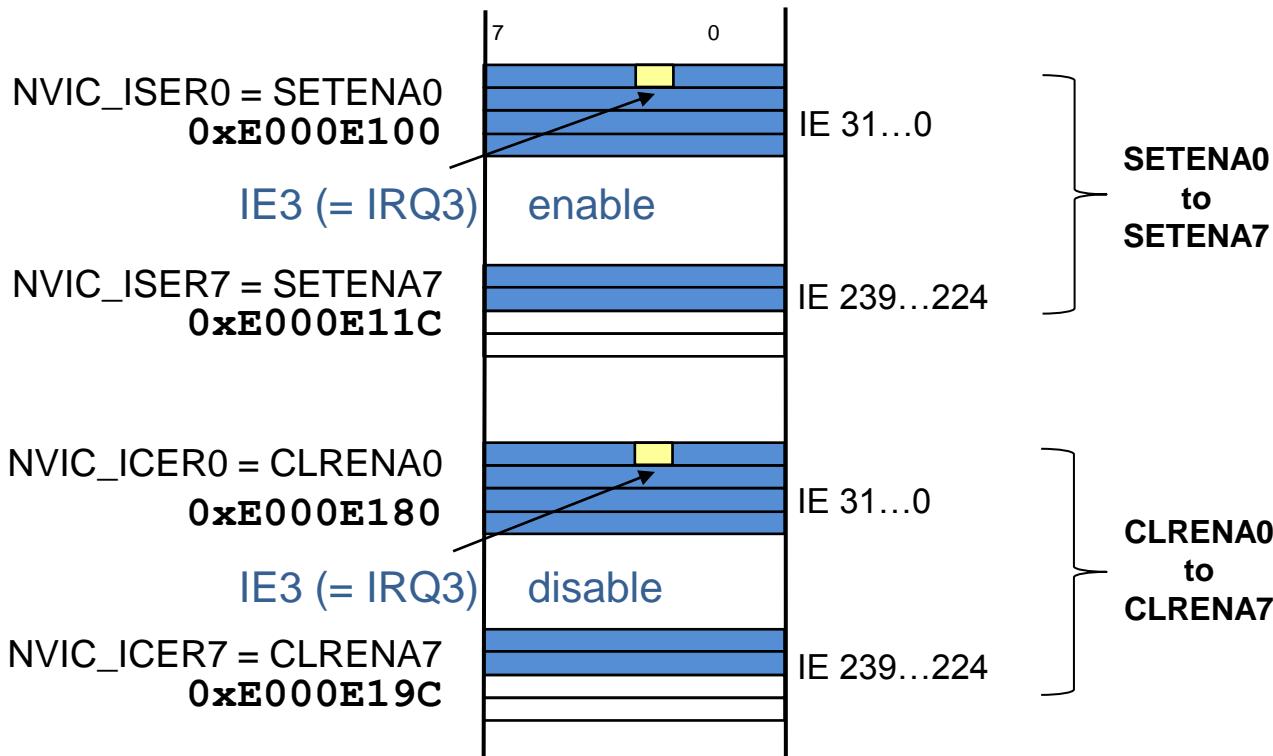
- Power-fail, emergency button, watchdog, ...

¹ CPSIE/D = Change Processor State Interrupt Enable / Disable

Interrupt Control

■ Interrupt Enable Registers

- Individual masking of interrupt sources
 - IEn cleared pending bit not forwarded
 - IEn set interrupt enabled



CLRENA and SETENA registers can be read by software and will return the settings of the IE bits

Enable IRQ3

```
SETENA0 EQU 0xE000E100
...
LDR R0, =SETENA0
MOVS R1, #0x08
STR R1, [R0]
```

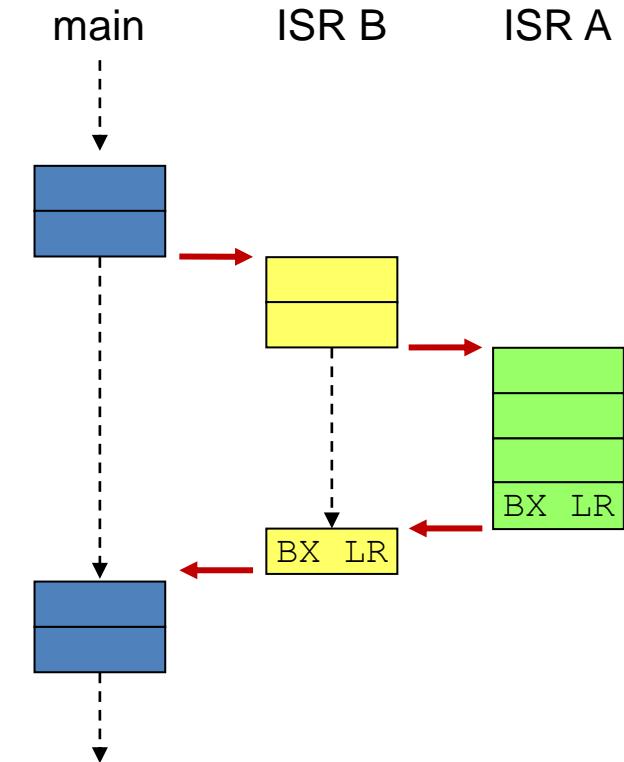
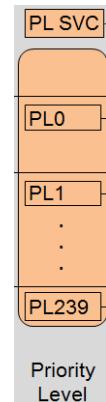
Disable IRQ3

```
CLRENA0 EQU 0xE000E180
...
LDR R0, =CLRENA0
MOVS R1, #0x08
STR R1, [R0]
```

Nested Exceptions

■ Preemption

- Service routine A temporarily interrupts service routine B
- Assigned priority level for each exception
 - Levels define whether A can preempt B
- Fixed priorities
 - Reset (-3), NMI (-2), hard fault (-1)
- All other priorities are programmable

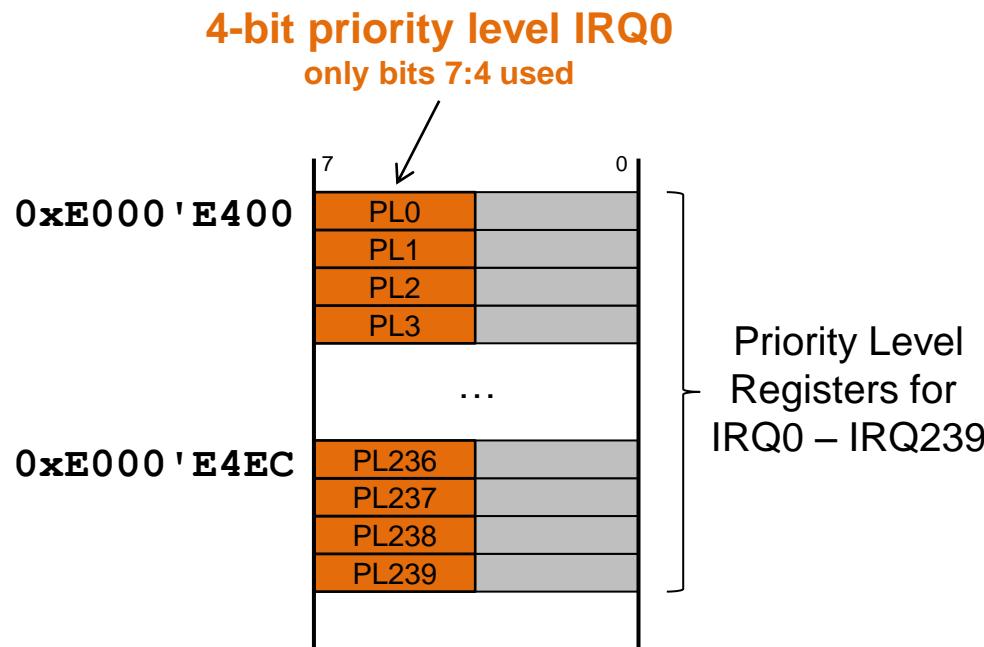


if A has been programmed
to higher priority than B

Nested Exceptions

■ Priority Level Registers

- The lower a priority level, the greater the priority
- 4-bit priority level ¹⁾ 0x0 – 0xF



Set priority of IRQ3 to 5

```
PL_IRQ3 EQU 0xE000E403
...
LDR R0, =PL_REG_IRQ3
MOVS R1, #0x50
STRB R1, [R0]
```

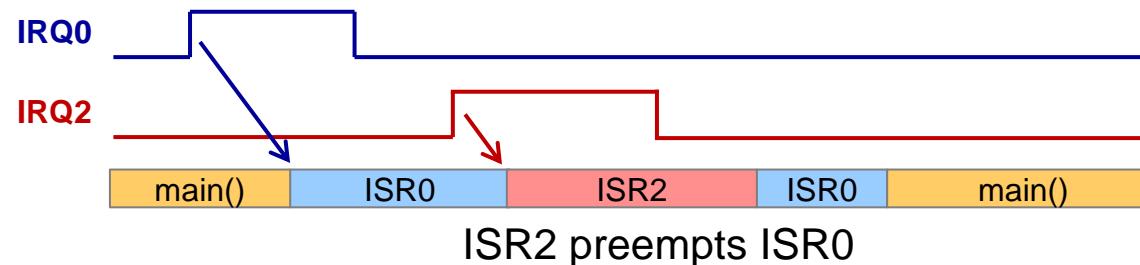
priority level registers for system exceptions
0xE000'ED18 – 0xE000'ED23

1) Cortex-M3/4 can handle up to 8 bits but STM uses only 4
Registers are called NVIC_IPRx on ST

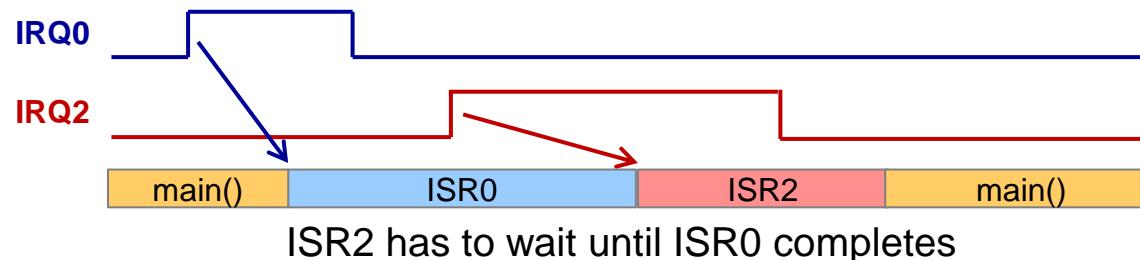
Nested Exceptions

■ IRQ request while other ISR is running

- ISR0 is executing (triggered by IRQ0)
- Interrupt request IRQ2 arrives
- $PL2 < PL0 \rightarrow$ IRQ2 has higher priority than IRQ0



- $PL2 \geq PL0 \rightarrow$ IRQ2 has lower or same priority than IRQ0



Nested Exceptions

■ Two or more pending interrupts

- (1) IRQ requests arrive simultaneously
- (2) More than one request has been waiting for a higher priority ISR to complete

- NVIC will evaluate the priorities of the pending interrupts
 - Selects IRQx with highest priority i.e.
 - ▶ with lowest priority level value (PL)
 - ▶ with lowest number (in case of identical PL values)
 - Other interrupts remain pending

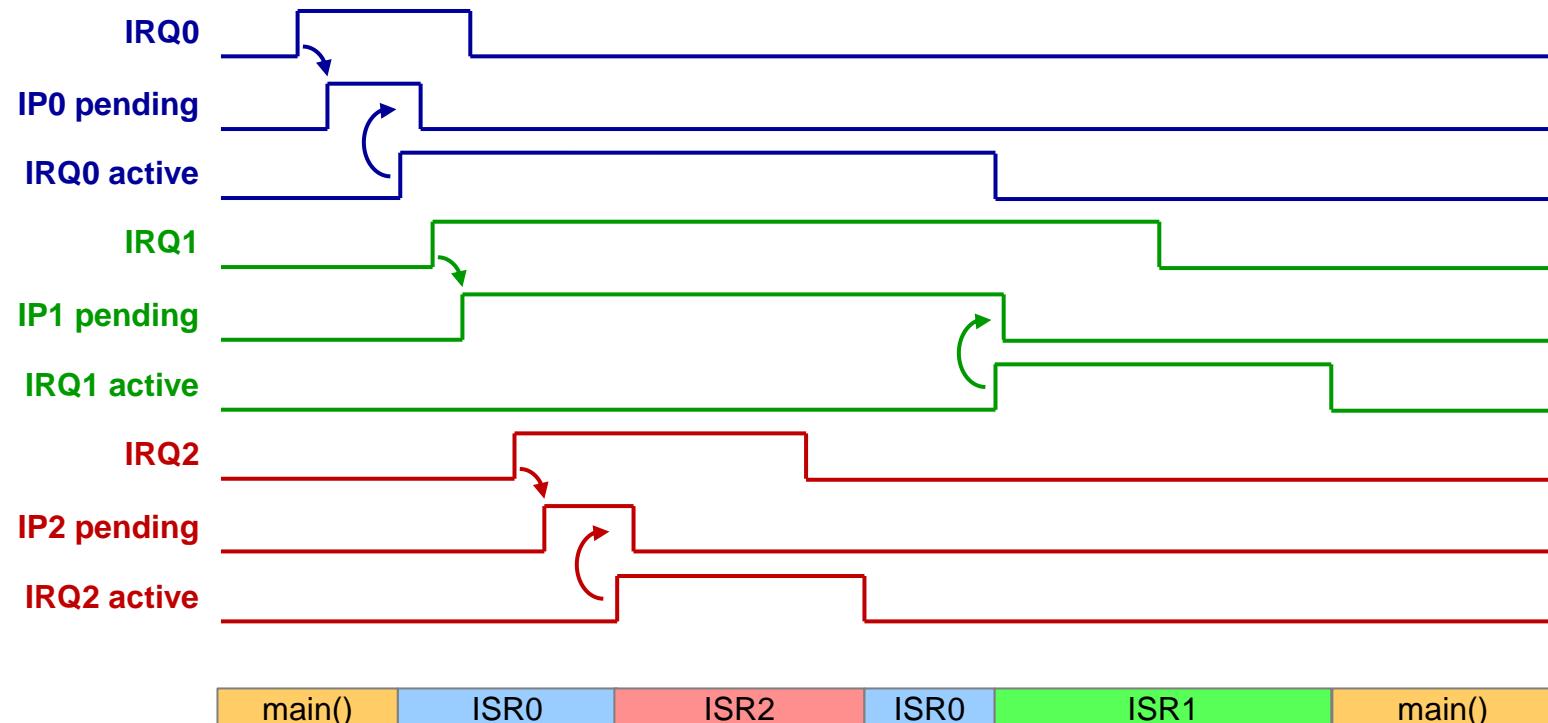
Nested Exceptions

■ Example Priorities

- ISR1 does not preempt ISR0
- ISR2 preempts ISR0

assuming

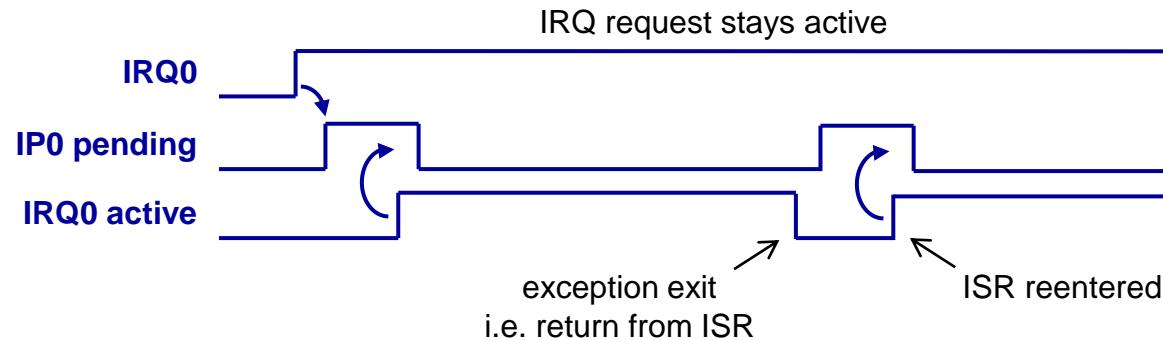
IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



Special Interrupt Situations

■ IRQ request stays active

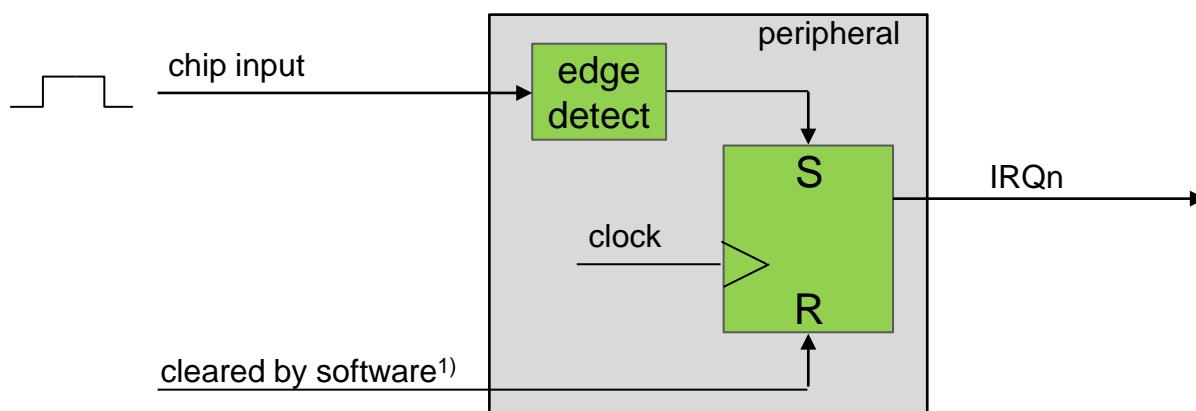
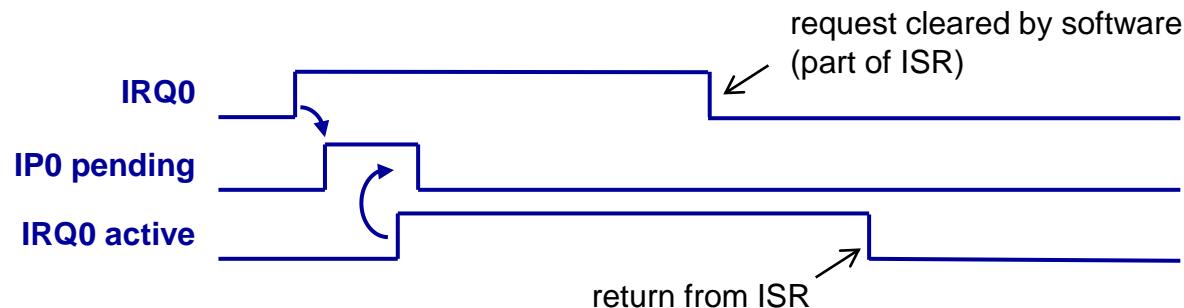
- Interrupt becomes pending again



Special Interrupt Situations

■ Common Configuration on Microcontrollers

- IRQ Set by Hardware – Cleared by Software

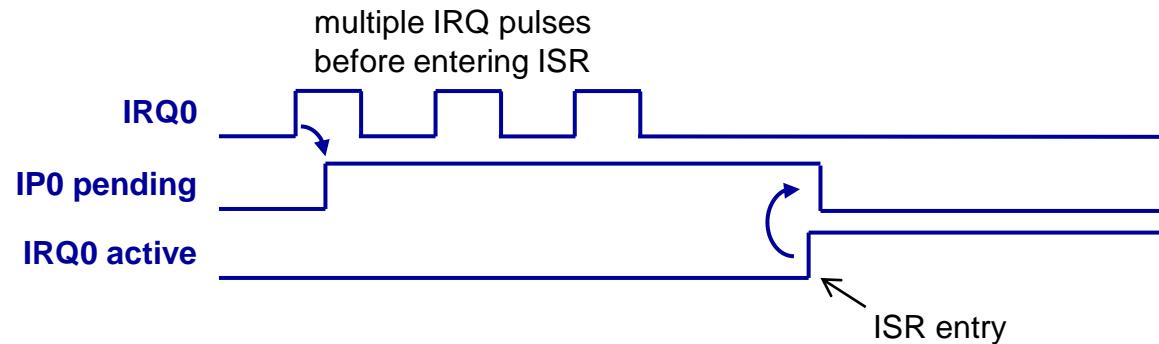


1) The software clears the bit at the correct position at the defined register address

Special Interrupt Situations

■ Multiple IRQ request pulses before entering ISR

- Treated as single interrupt
- Events are lost



■ What is CMSIS?

- Cortex Microcontroller Software Interface Standard
- Vendor-independent hardware abstraction layer for Cortex-M
- Defines generic tool interfaces
- ISO/IEC C code cannot directly access some Cortex-M3 instructions
 - intrinsic functions required

CMSIS Functions

■ NVIC Control

```
void NVIC_EnableIRQ(IRQn_t IRQn)  
void NVIC_DisableIRQ(IRQn_t IRQn)
```

Enable IRQn
Disable IRQn

```
uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)  
void NVIC_SetPendingIRQ (IRQn_t IRQn)  
void NVIC_ClearPendingIRQ (IRQn_t IRQn)  
uint32_t NVIC_GetActive (IRQn_t IRQn)
```

Return true (IRQ-Number) if IRQn is pending
Set IRQn pending
Clear IRQn pending status
Return '1' if active bit of IRQn is set. '0' otherwise

```
void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority) Set priority for IRQn  
uint32_t NVIC_GetPriority (IRQn_t IRQn) Read priority of IRQn
```

```
void NVIC_SystemReset (void)
```

Reset the system

Data Consistency

■ Example

```
typedef struct {
    uint8_t minutes;
    uint8_t seconds;
} time_t;

static time_t time = { 0, 0 };

int main(void)
{
    while (1) {

        write_byte(ADDR_LED_7_0,
                   time.seconds);
        write_byte(ADDR_LED_15_8,
                   time.minutes);

    }
}
```



```
void IRQ1_Handler(void)
{
    time.seconds++;
    if (time.seconds > 59) {
        time.seconds = 0;
        time.minutes++;
    }
}
```

time = { 15, 59 }
1) Output 59 → LED_7_0
2) Interrupt → time = { 16, 0 }
3) Output 16 → LED_15_8
→ display: 16 59 !!!

Data Consistency

- Data structure must not be changed during output
- Disable Interrupts during output
- Multitasking problem

Conclusions

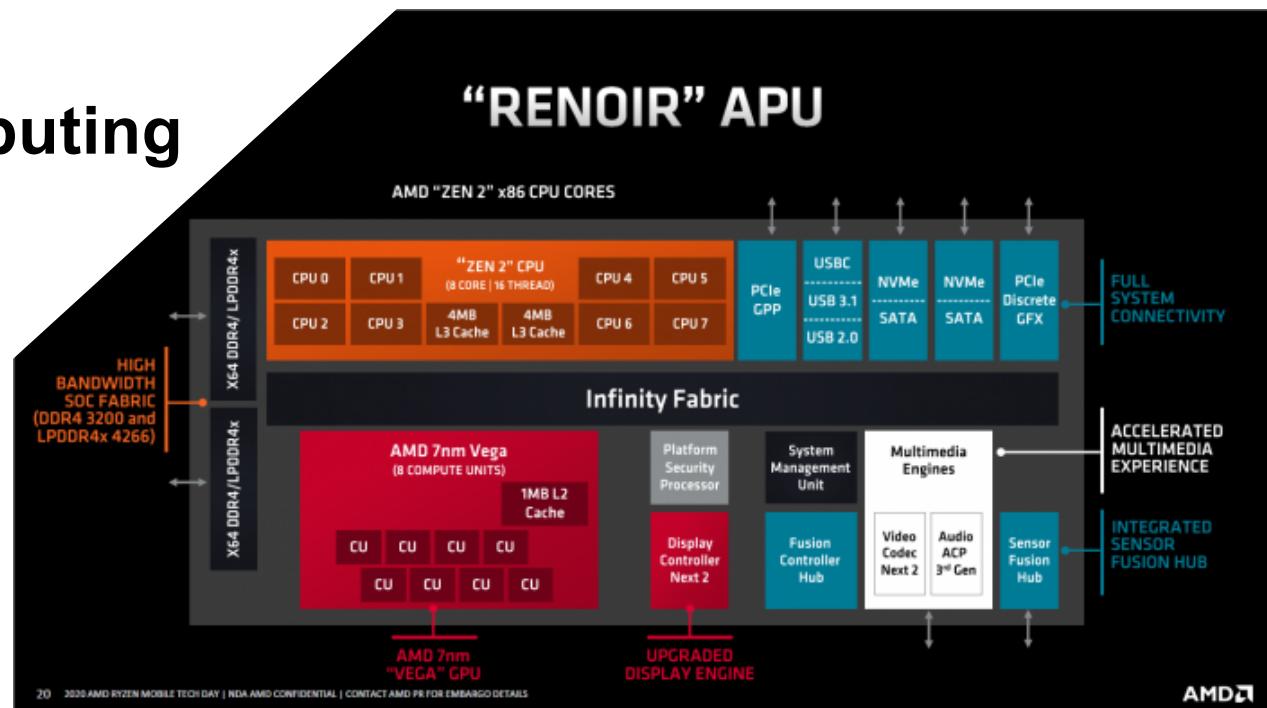
- **Polling vs. Interrupt-driven I/O**
- **Exceptions**
 - System exceptions: Reset, NMI, Faults, System Level Calls
 - Interrupts IRQ0 – IRQ239
- **Nested Vectored Interrupt Controller**
 - Vector Table holds addresses of interrupt service routines (ISR)
 - Save context including xPSR
 - Masking of interrupts
 - Pending and active bits
 - Priority levels
 - IRQn: Often hardware set; software reset
- **Data Consistency Issues**

Improving System Performance

Computer Engineering 1

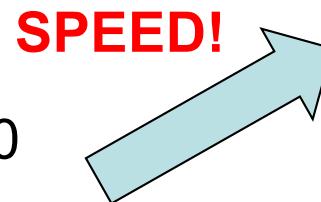
Agenda

- Motivation
- Aspects of Optimization
- Bus Architectures
- Instruction Set Architectures (ISA)
- Pipelining
- Parallel Computing



■ Intel Pentium E5800

- 3.2 GHz
- 2 Cores
- 2 MB cache
- Released Q4 2010
- **PassMark: 1184**
- **TDP: 65 W**



■ Intel Celeron G4930

- 3.2 GHz
- 2 Cores
- 2 MB cache
- Q2 2019
- **PassMark: 2628**
- TDP: 54 W

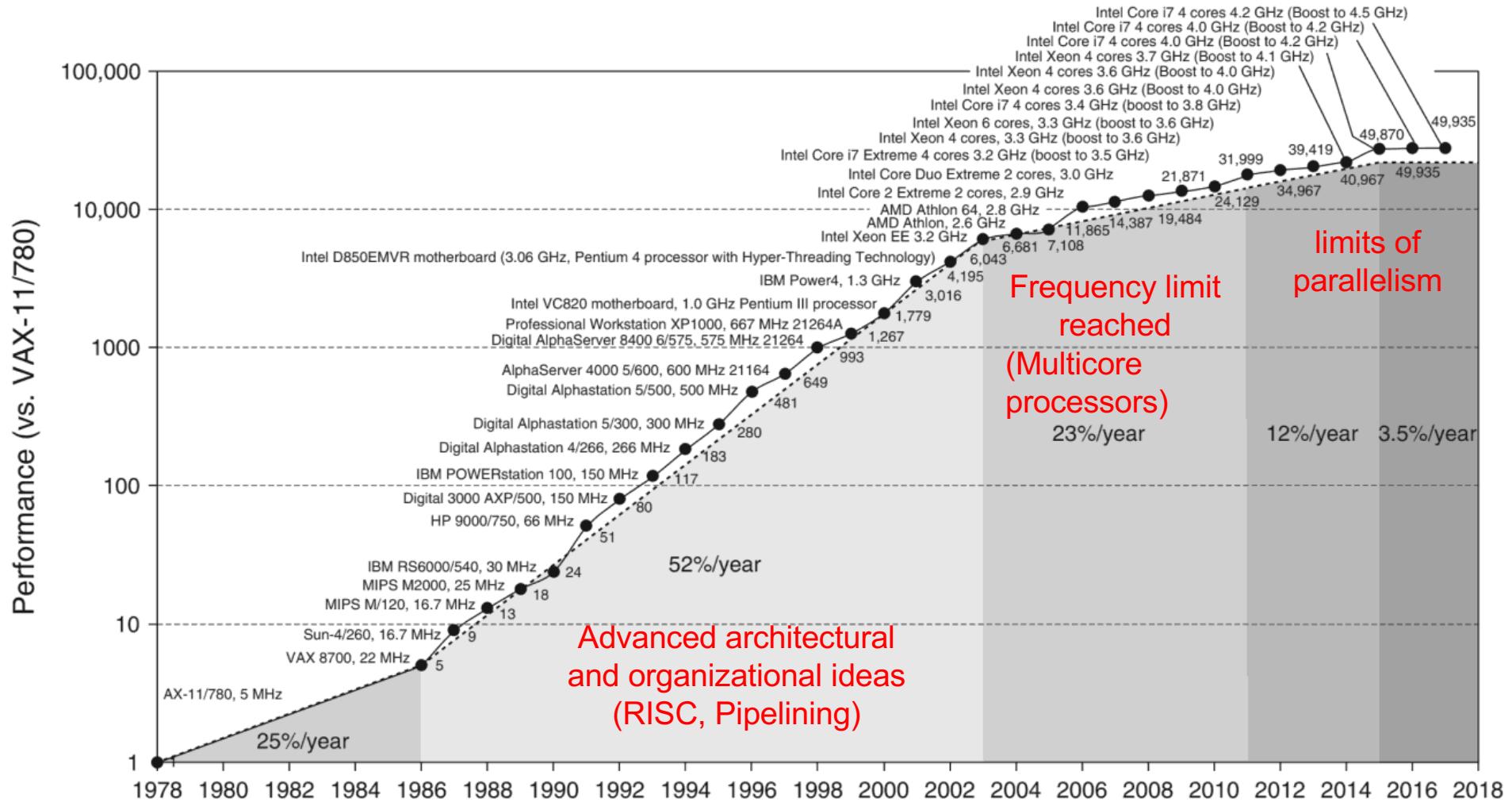
■ AMD Ryzen R1102G

- 1.2 GHz
- 2 Cores
- 4 MB cache
- Q1 2020
- PassMark: ~1600
- **TDP: 6 W**



Motivation

Growth in processor performance



Source: John L. Hennessy, David A. Patterson (2017). Computer Architecture: A Quantitative Approach. 6th edition.

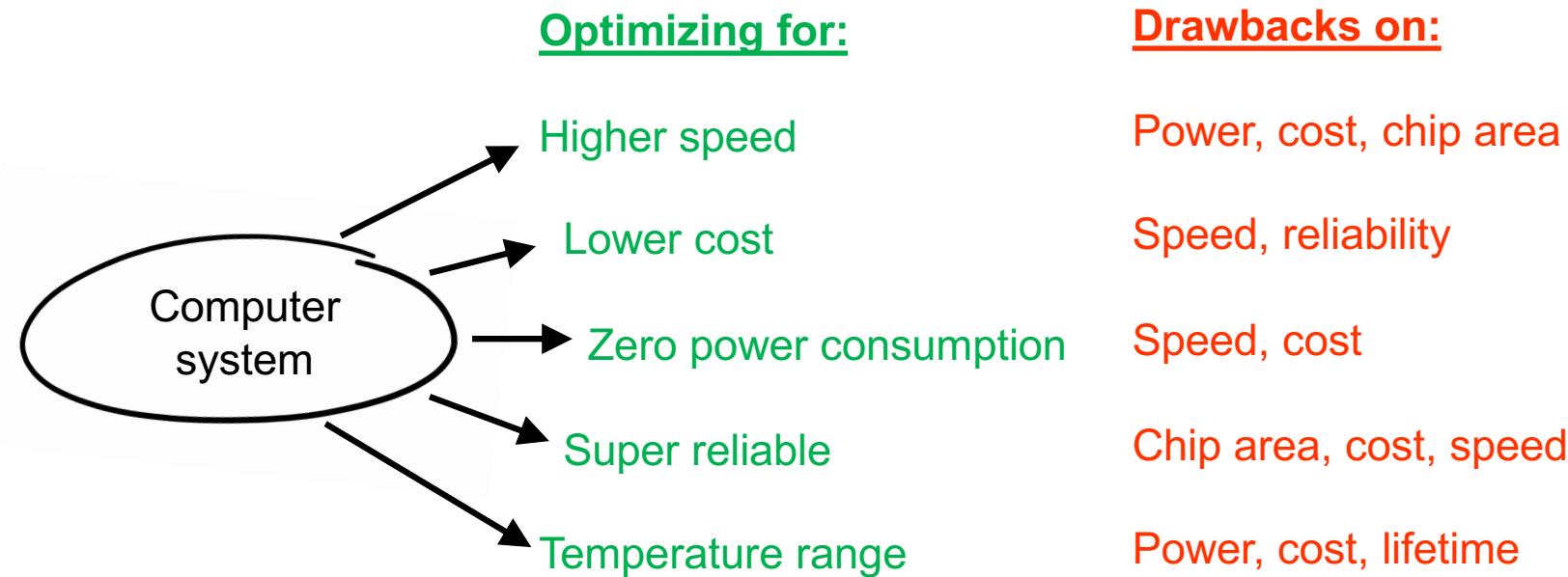
Learning Objectives

■ At the end of this lesson you will be able

- to explain different types of bus architectures
- to understand the difference between von Neumann and Harvard architecture
- to understand RISC and CISC paradigms
- to describe the idea of pipelining
- to calculate processing performance improvement through pipelining
- to describe the basics of parallel computing

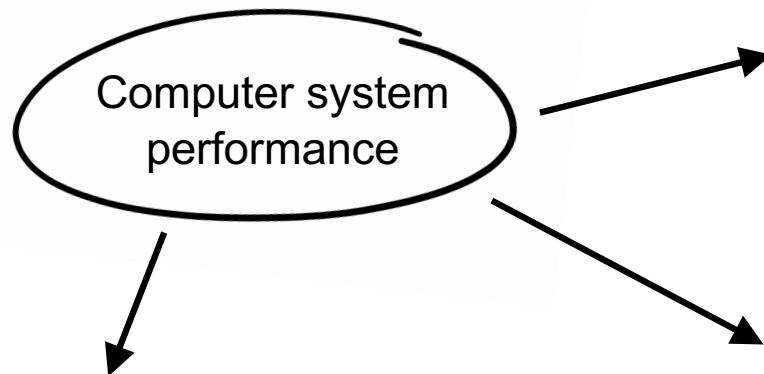
Aspects of Optimization

■ ‘Wishlist of Optimization’



- This lecture: Increasing system/CPU speed
- Some other topics: Covered in ‘Microcomputer Systems 1’ (MC1)

How to Increase System Speed?



CPU improvements:

- Clock Speed
- Cache Memory
- Multiple Cores / Threads
- Pipelined Execution
- Branch Prediction
- Out of Order Execution
- Instruction Set Architecture

External factors:

- Better Compilers
- Better Algorithms

System level factors:

- Special Purpose Units
 - e.g. Crypto, Video, AI
- Multiple Processors
- Bus Architecture
 - e.g. von Neumann / Harvard
- Faster components (e.g. SSDs, 1000Base-T, etc.)

How is the connection between the CPU and memory organized?

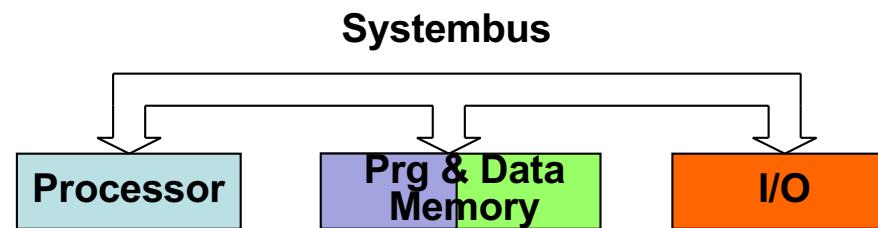
A single system bus: simple, cheaper slower

Multiple system buses: faster more complexity

Bus Architecture

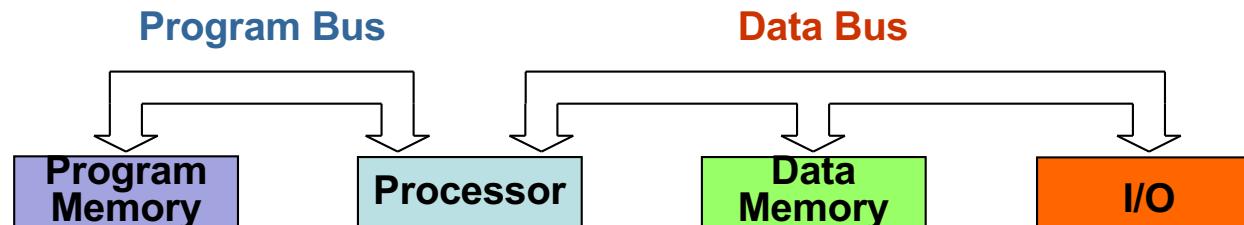
■ von Neumann Architecture

- Same memory holds program and data
- Single bus system between CPU and memory



■ Harvard Architecture

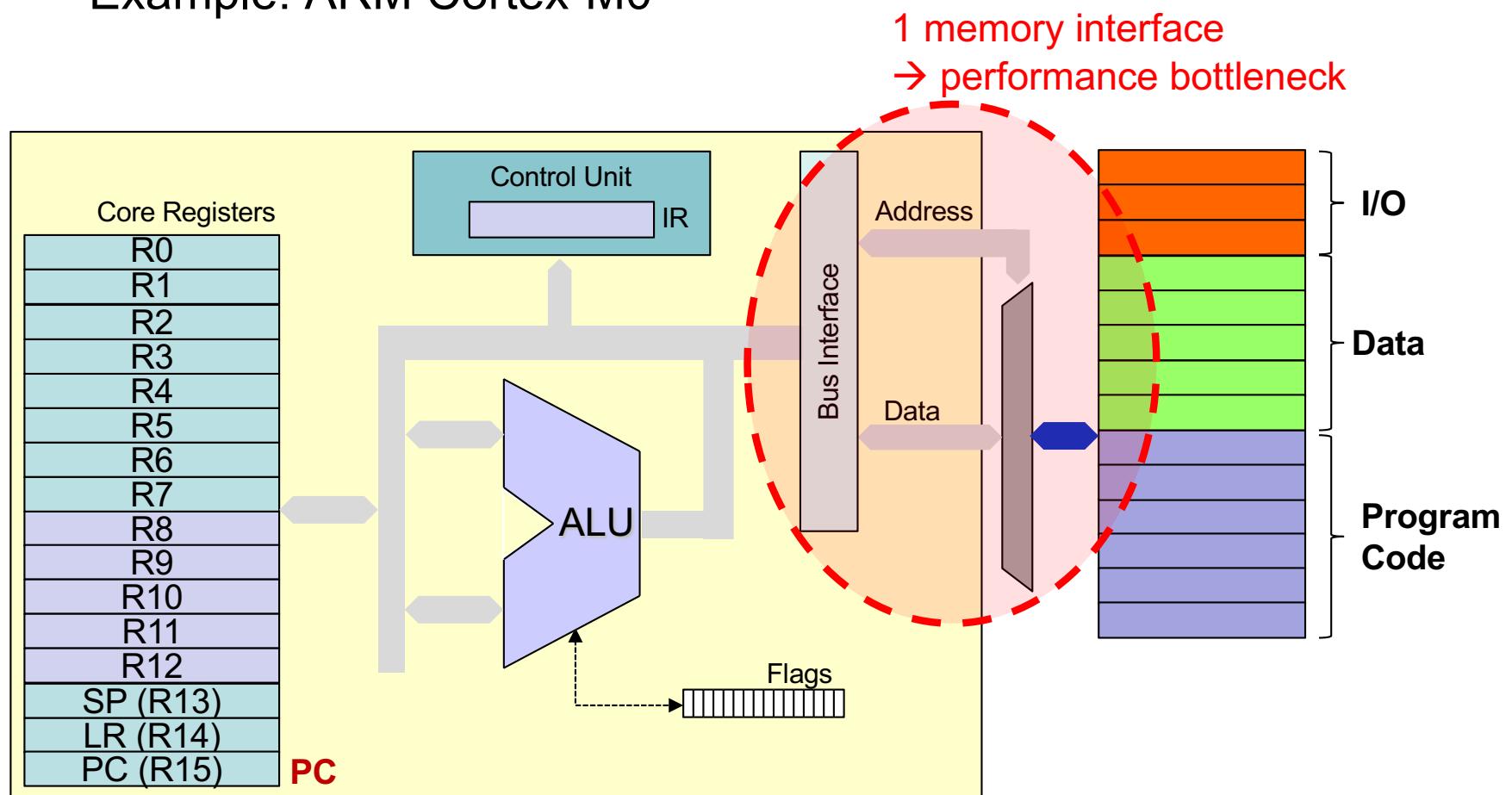
- “Mark I” at Harvard University (Howard Aiken, 1939-44)
- Separate memories for program and data
- Two sets of address/data buses between CPU and memory



Bus Architecture

■ von Neumann Architecture

- Example: ARM Cortex-M0

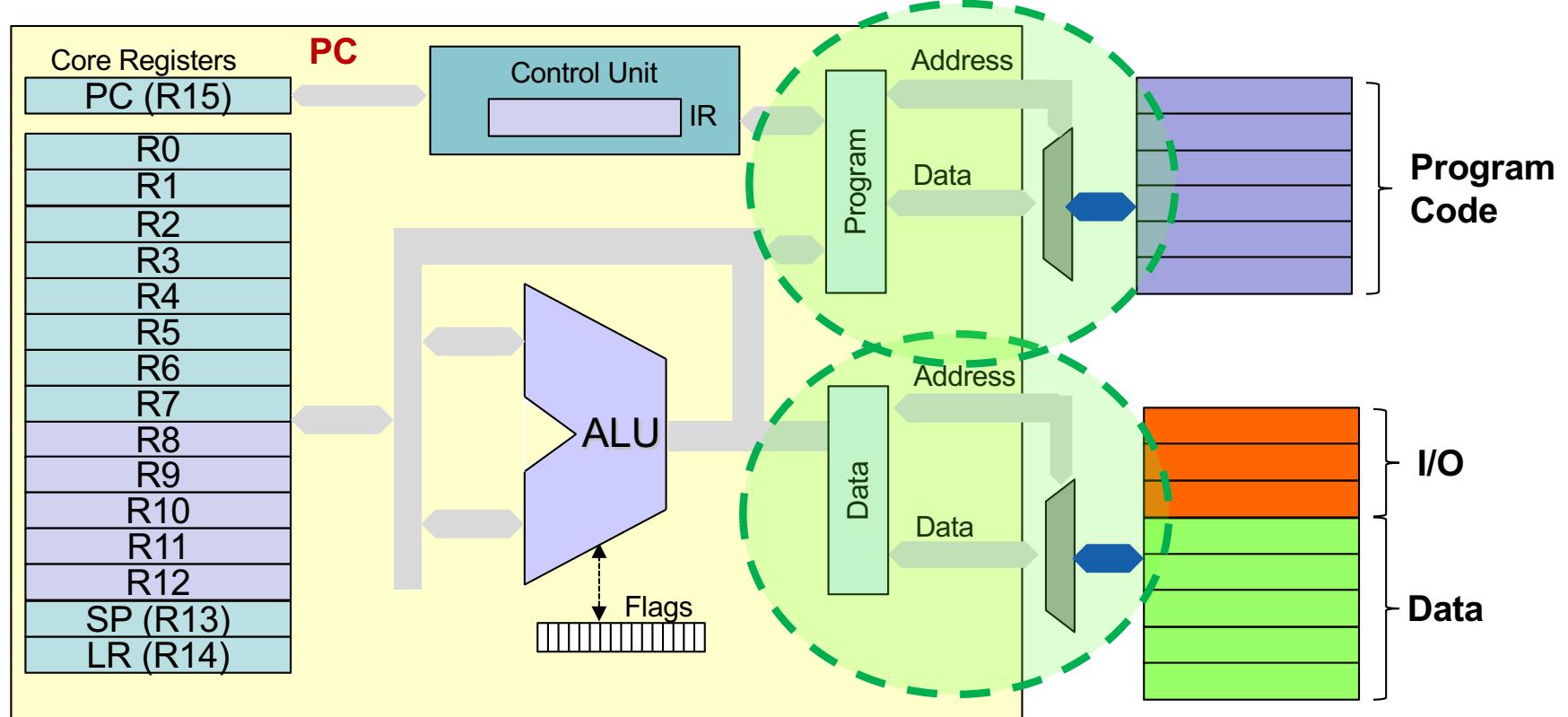


Bus Architecture

■ Harvard Architecture

- Example: ARM Cortex-M3/M4

2 memory interfaces
→ twice the access speed



Powerful, complex instructions?

or

Simple, highly optimizable instructions?

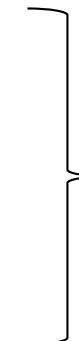
(or something in between?)

■ First CPUs

- Small address bus, small data bus (8 bit)
- Few registers, no cache, pipelining, co-processors, multi-core
- Machine code as finite state machine: Shift, multiply, etc. in loops

■ Consequences for machine code

- Differing complexity
- Different length
- Variable execution time
- Individual addressing / arguments



for different instructions

→ More and more complex instructions (CISC)

Instruction Set Architecture

- **John Cocke (IBM, 1974) proved that**
 - 80% of work was done using only 20% of the instructions
 - Most (complex) instructions were not used at all by compilers
- **RISC (Reduced Instruction Set Computer)**
 - Few instructions, unique instruction format
 - Fast decoding, simple addressing
 - less hardware → allows higher clock rates
 - more chip space for registers (up to 256!)
 - Load-store architecture reduces memory accesses, CPU works at full-speed on registers



Acorn Archimedes A3000
Source: Wikipedia Commons

Remark: Word “CISC” was introduced with “RISC”

Instruction Set Architecture

■ RISC

- Load / Store Architecture
- Data processing instructions only available on registers

Example: **Balance = Balance + Credit**

```
LDR  R0 ,=Credit
LDR  R1 ,[R0]
LDR  R0 ,=Balance
LDR  R3 ,[R0]
ADDS R2 ,R1 ,R3
STR  R2 ,[R0]
```

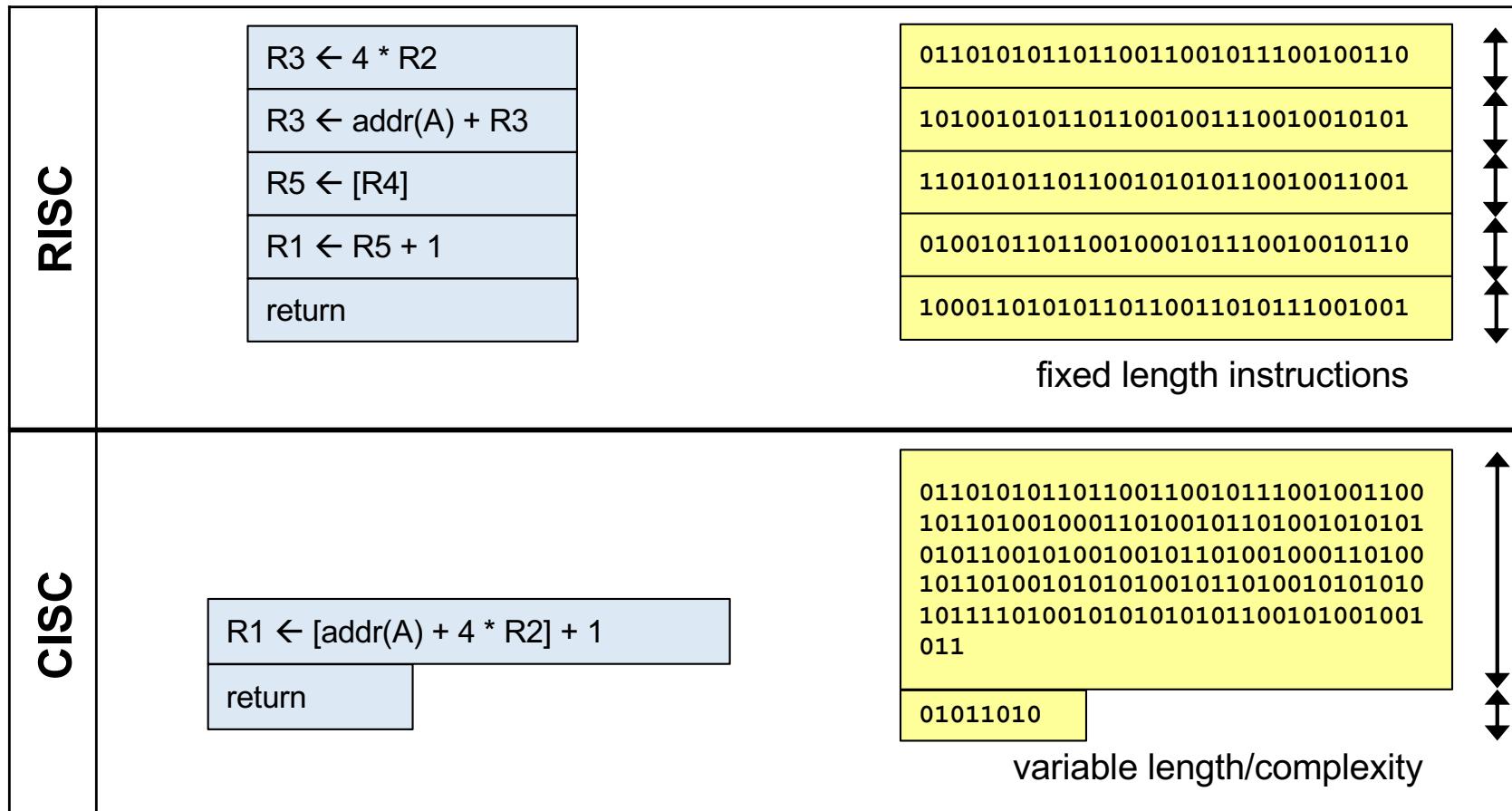
■ CISC

- One of the operands of an instruction may directly be a memory location

```
MOV  AX, [Credit]
ADD  [Balance], AX
```

Instruction Set Architecture

■ Instruction / opcode complexity



No RISC – No Fun

■ **RISC advantages**

- Simple and fast instruction decoding
- More registers/cache on silicon area (less memory access)
- Several data paths possible (superscalar computers)
- Higher clock frequencies
- Effective compiler optimization with limited, generic instructions
- Easy pipelining, shorter pipelines (instruction size / duration)

■ **CISC advantages**

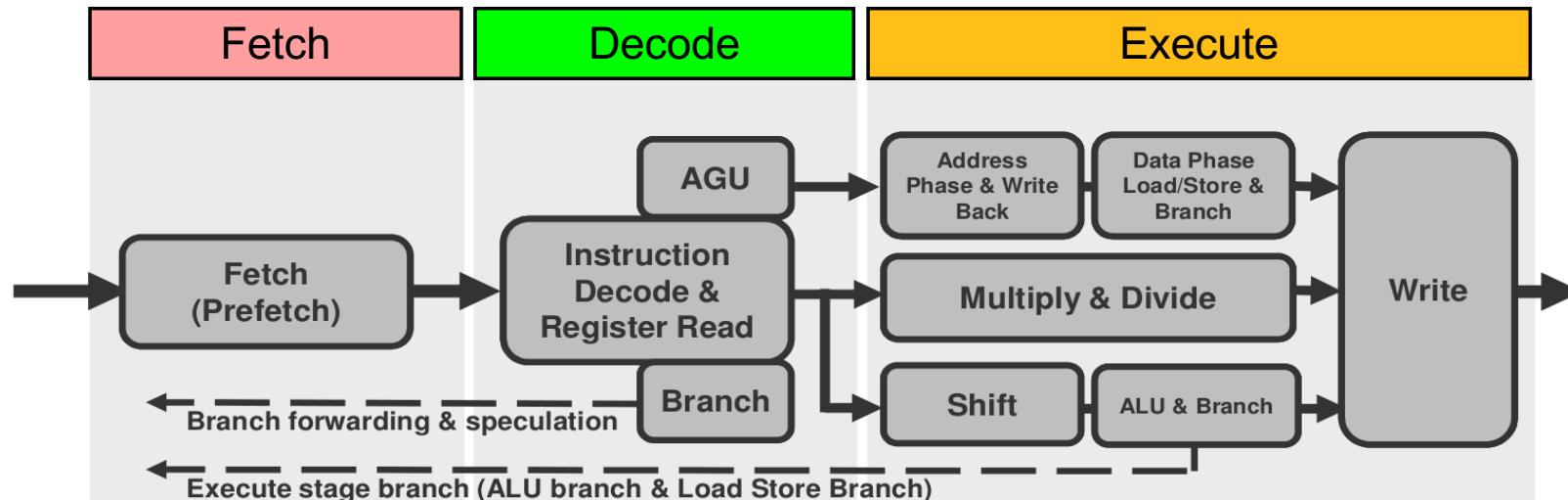
- Less program memory needed with complex instructions
- Short programs may work faster with less memory accesses

■ **CISC and RISC more and more indistinguishable**

- Modern x86 CPUs convert instructions (CISC to RISC)

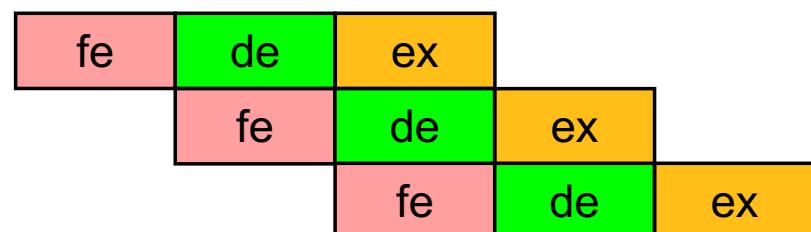
Pipelining

■ ARM Cortex-M3 sequence:



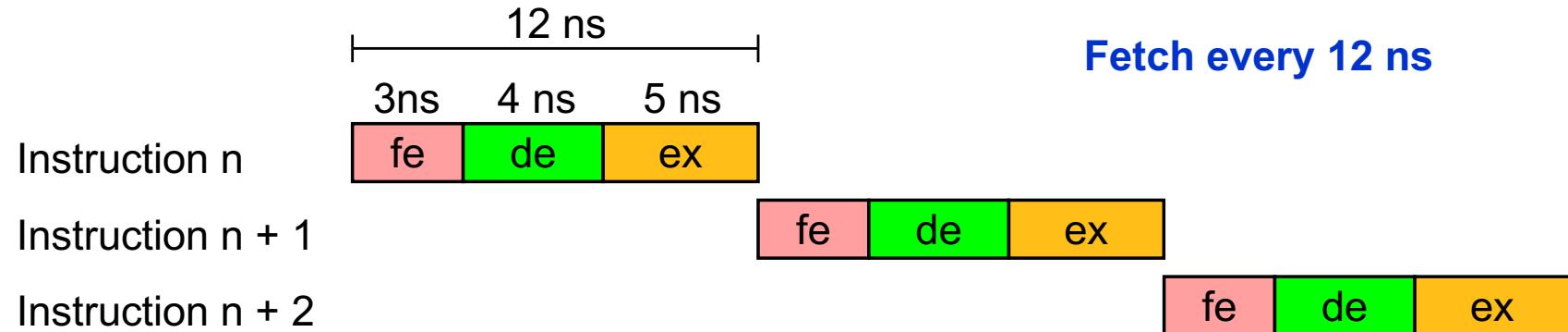
Source: ARM

■ Idea: Why not fetch the next instruction, while the current one decodes?

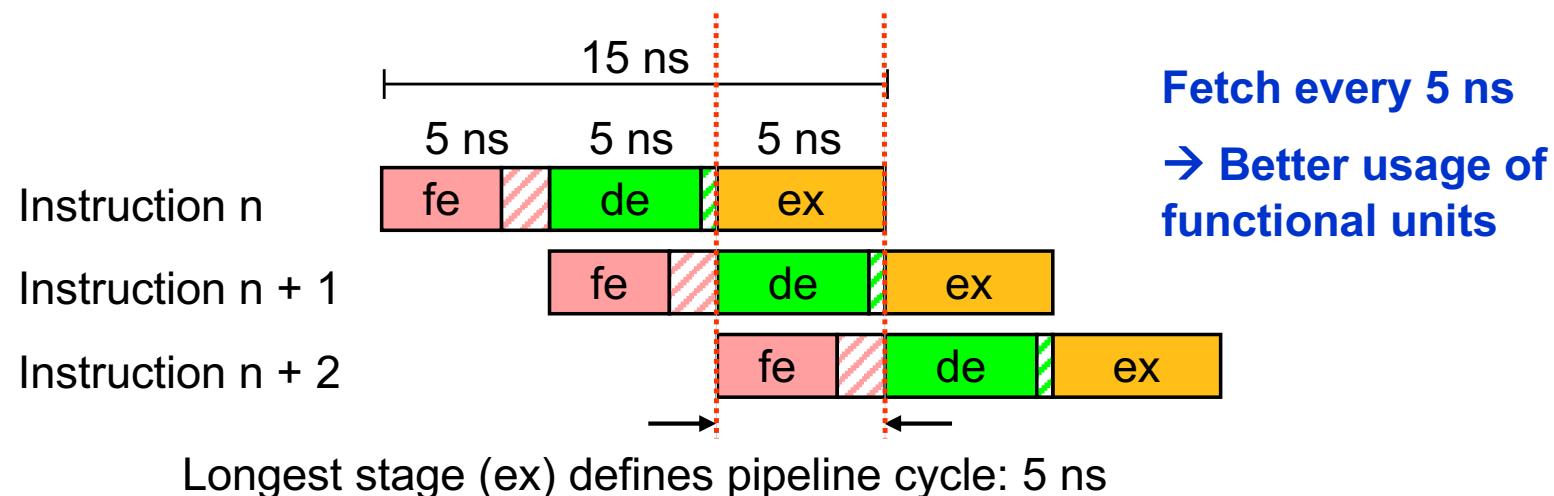


Pipelining

■ Sequential execution



■ Pipelined execution



Pipelining

■ Instructions per second

- Without pipelining

$$\text{Instructions per second} = \frac{1}{\text{Instruction delay}}$$

- With pipelining
 - Pipeline needs to be filled first
 - After filling, instructions are executed after every stage

$$\text{Instructions per second} = \frac{1}{\text{Max stage delay}}$$

■ Example Cortex-M3

- Without pipelining: $1/12\text{ns} = 83,3$ million instructions per second
- With pipelining: $1/5\text{ns} = 200$ million instructions per second

Pipelining

■ Advantages

- All stages are set to the same execution time
- Massive performance gain
- Simpler hardware at each stage allows for a higher clock rate

■ Disadvantages

- A blocking stage blocks whole pipeline
- Multiple stages may need to have access to the memory at the same time

■ General

- Number of execution stages is design decision
- Typically 3 (ARM Cortex M4) – 20 stages (Pentium 4)

Pipelining

■ Optimal Pipelining

- All operations here are on registers (single cycle execution)
- In this example it takes 6 clock cycles to execute 6 instructions
- Clock cycles per instruction (CPI) = 1

Cycle		1	2	3	4	5	6	7	8	9
Operation										
<i>ADD</i>		fe	de	ex						
<i>SUB</i>			fe	de	ex					
<i>ORR</i>				fe	de	ex				
<i>AND</i>					fe	de	ex			
<i>ORR</i>						fe	de	ex		
<i>EOR</i>							fe	de	ex	

Pipelining

■ Special situation: LDR

- In this example it takes 7 clock cycles to execute 6 instructions
- Read cycle must complete on the bus before LDR instruction can complete
- Next 2 instructions must wait one pipeline cycle ($S = \text{stall}$)
- Clock cycles per instruction (CPI) = 1.2

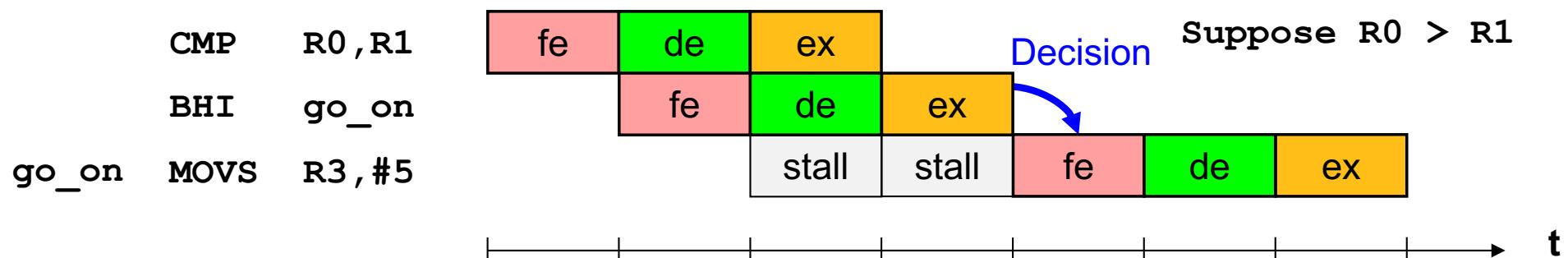
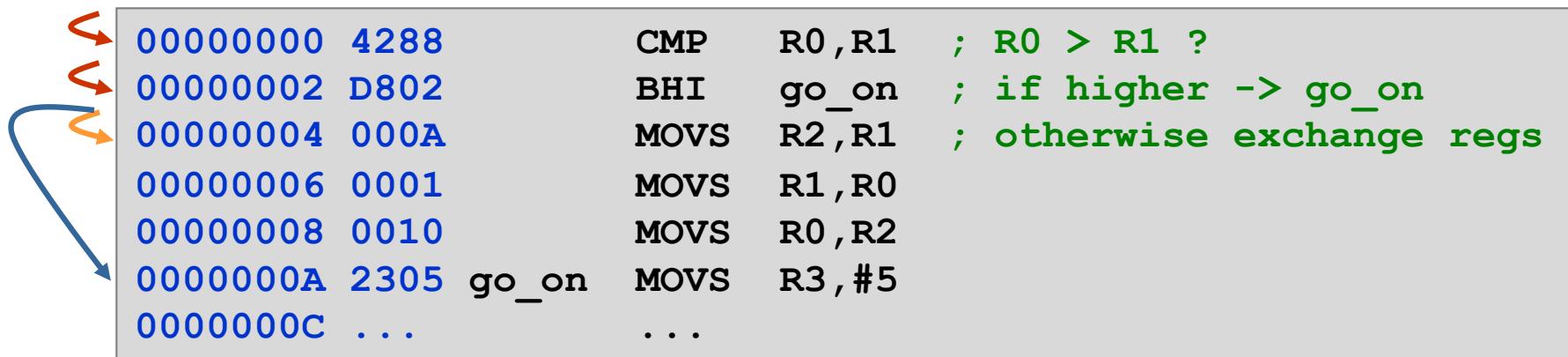
Cycle		1	2	3	4	5	6	7	8	9
Operation										
ADD		fe	de	ex						
SUB			fe	de	ex					
LDR	circled		fe	de	Ea	Ed				
AND			fe	de	S	ex				
ORR			fe	S	de	ex				
EOR				fe	de	ex				

*Ea: LDR address phase
Ed: LDR data phase*

Pipelining

■ Control Hazards

- Branch / jump decisions occur in stage 3 (ex)
- Worst case scenario – conditional branch taken:



Pipelining

■ Reduce control hazards

- Loop fusion reduces control hazards
- Simple example for illustration

```
/* Two loops → double number of control hazards*/
for (i = 0; i < N; i++) {
    a[i] = 1/b[i] * c[i];
}
for (i = 0; i < N; i++) {
    d[i] = a[i] + c[i];
}

/* Better performance: Fusion into one loop */
for (i = 0; i < N; i++) {
    a[i] = 1/b[i] * c[i];
    d[i] = a[i] + c[i];
}
```

2*N Conditional
Branches

N Conditional
Branches

Pipelining

■ Ideas to further improve pipelining:

- Branch prediction:
 - Store last decision made for each conditional branch
→ probability is high that the same decision is taken again
- Instruction prefetch:
 - Fetch several instructions in advance
→ better use of the system bus
→ possibility of 'Out of Order Execution'
- Out of Order Execution:
 - If one instruction stalls (e.g. a LDR from slow memory),
it might be possible to already execute the next instruction

Pipelining

■ Limits of optimization

- Complex Optimizations → severe security problems (e.g. Out of Order Execution)
- Instructions (speculatively) executed, that would throw access violations under ‘In Order’ circumstances.
- ‘Meltdown’ and ‘Spectre’ attacks: allow a process to access the data of another process.



Parallel Computing

Streaming/Vector Processing

- One instruction processes multiple data items simultaneously

Multithreading

- Multiple programs/threads share a single CPU

Multicore Processors

- One processor contains multiple CPU cores

Multiprocessor Systems

- A computer system contains multiple processors

Parallel Computing

■ Instructions working on more than one data item

		Data streams	
		Single	Multiple
Instruction streams	Single	SISD: Single processor with one instruction, data stream	SIMD: Data Vectors, all data streams react to one instruction
	Multiple	(no examples)	MIMD: Multiprocessor with SIMD-instructions

Source: David A. Patterson, John L. Hennessy

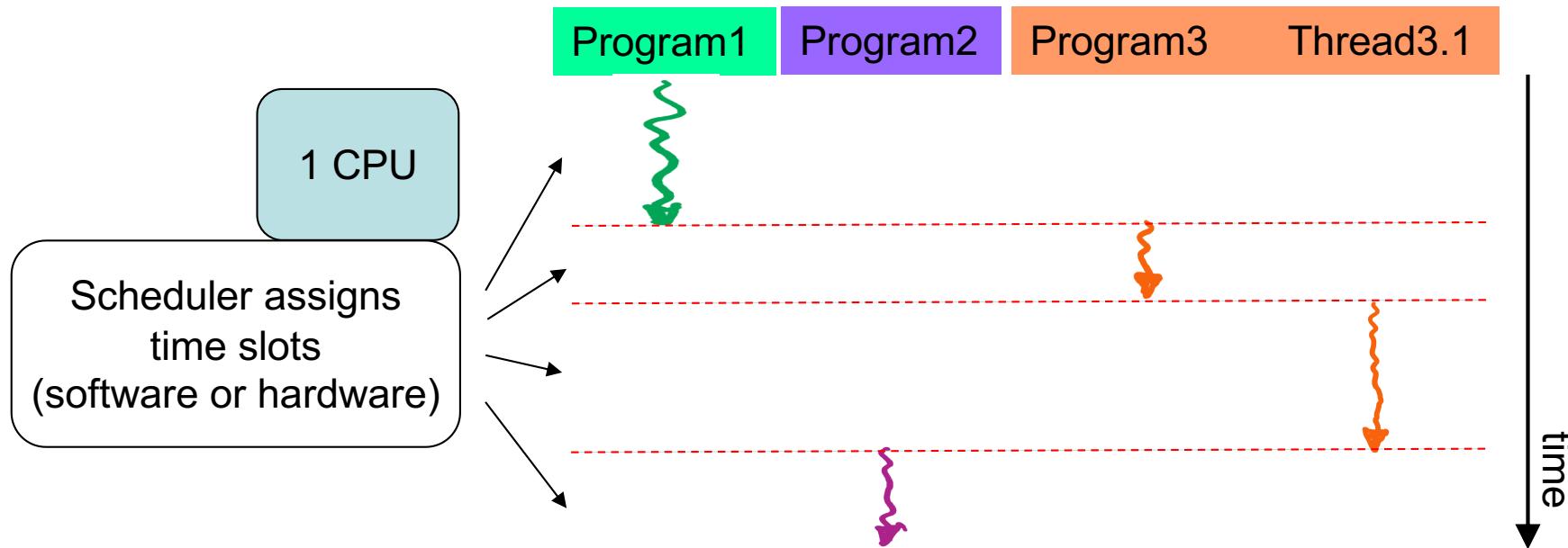
■ SIMD examples (x86):

MMX (Multimedia Extension), AMD 3D-Now,
SSE (Streaming SIMD Extensions),
AVX (Advanced Vector Extensions)

Parallel Computing

■ Multithreading

- Scheduler: Assigns time slots to programs/threads
- Programs/threads only **seem** to run in parallel (1 CPU)



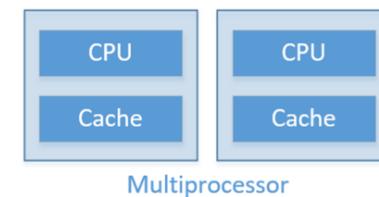
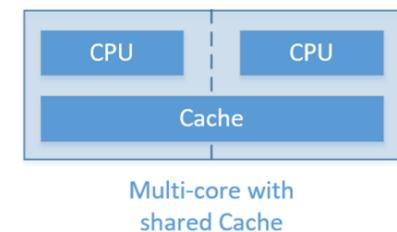
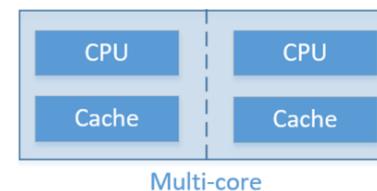
Remark: More on Multithreading in MC1

Parallel Computing

■ Parallelism on CPU / processor level:

- Multicore processor
 - All on one chip
 - Less traffic (cores integrated on one chip)
 - Possibility to share memories on-chip
 - Cheaper

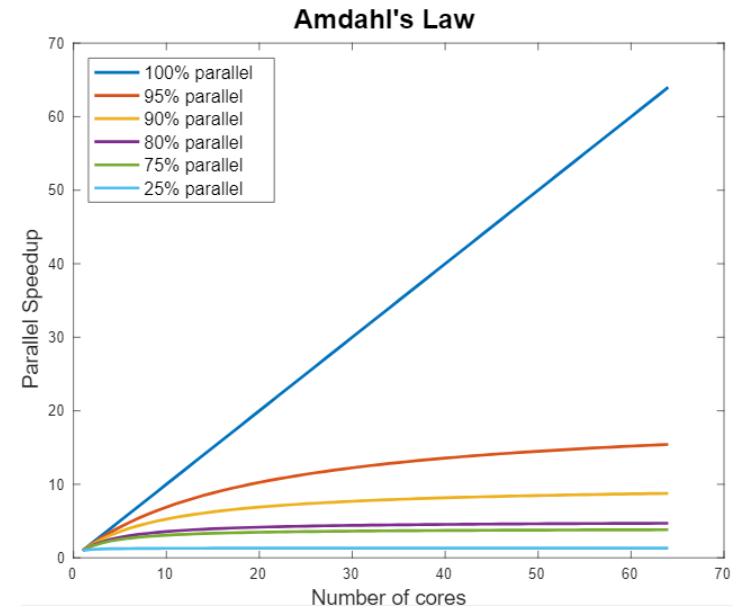
- Multiprocessor
 - Multiple Chips
 - Longer distances between CPUs
 - More expensive



Source: <https://www.queryhome.com/tech/106320/what-the-difference-between-multicore-and-multiprocessor>

Conclusion

- CPU clock frequency not the only factor
- Different ways to increase processor performance
 - RISC / CISC
 - Pipelining
 - Out-of-order execution
 - Parallel computing
- Today's CPUs combine most of these technologies
- Other components with influence
 - SSD, memory, interfaces, algorithms ...



Source: <https://researchcomputingservices.github.io/parallel-computing/02-speedup-limitations/>