A computer system is a device that

- processes input
- takes decisions based on the outcome
- and outputs the processed information

Hardware Components

- **CPU**: Central Processing Unit or processor
  - **Datapath**
    - **ALU (Arithmetic and Logic Unit)**: performs arithmetic/logic operations
    - **Registers**: fast but limited storage inside CPU, hold intermediate results
  - **Control Unit**
    - **Finite State Machine (FSM)**: reads and executes instructions
    - **types of operations**: data transfer: registers ↔ memory / arithmetic and logic operations /jumps
- **Memory**: stores instructions and data
  - **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
- **Input / Output**: interface to external devices
- **System-Bus**: electrical connection of blocks
  - **address lines**: CPU drives the desired address onto the address lines, number of addresses $= 2^n | n = number\ of\ address\ lines$
  - **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

From C to executable: example with gcc (The GNU Compiler Collection)

hello.c: Source program (text) >> Preprocessor **cpp** > *hello.i*: Modified source program (text) >> Compiler **cc1** > *hello.s*: Assembly program (text) human-readable, CPU specific >> Assembler **as** > *hello.o*: Relocatable object program (binary) >> Linker **ld** > *hello*: Executable object program (binary)

Host and Target

- Software development on host
  - Compiler/Assembler/Linker on host
  - Loader on target loads executable 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

Benefits of knowing Assembler

- 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

Combinational Logic

- Logic states in a binary system
  - Outputs change depending on inputs and internal logic functions
  - System has no memory, i.e. there is no storage element
  - For n inputs there are 2^n possible input combinations
  - The only timing influence are internal delays
  - Outputs are stable after a delay

Functions / symbols set

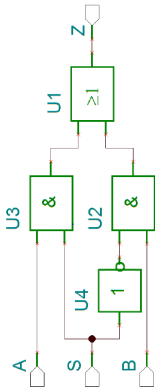
Function	Text
AND	Z=A&B
OR	Z=A#B
Buffer	Z=A
XOR	Z=A$B
NOT	Z=!A
NAND	Z=!(A&B)
NOR	Z=!(A#B)
XNOR	Z=!(A$B)

Function	IEC 60617-12 *since 1997*	US ANSI 91 *1984*	DIN 40700 *until 1976*
AND			
OR			
Buffer			
XOR			
NOT			
NAND			
NOR			
XNOR			

Multiplexer

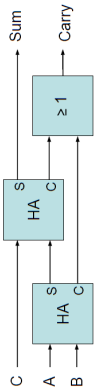
Z=(A&S)#(B&!S)

S	A	B	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

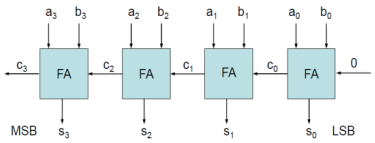


1-Bit Full-Adder

C	A	B	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



4-Bit Adder



Sequential Logic

- General form → Finite State Machine (FSM)
  - Contains memory, i.e. storage of system state
  - Outputs depend on inputs and internal state
  - Next system state depends on current state and inputs → clock

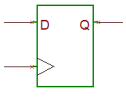
Period T vs. frequency f

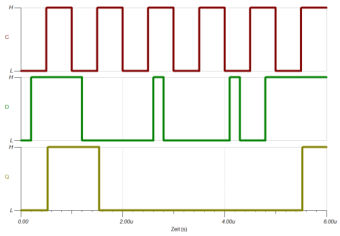
- **Period T**: measured in seconds (s)
- **Frequency f**: measured in Hertz (1/s), i.e. number of cycles per second

T	F
$1\ s$	$1\ Hz$
$1\ ms = 10^{-3}\ s$	$1\ kHz = 10^{3}\ Hz$
$1\ us = 10^{-6}\ s$	$1\ MHz = 10^{6}\ Hz$
$1\ ns = 10^{-9}\ s$	$1\ GHz = 10^{9}\ Hz$

D-Flip-Flop

- Edge triggered storage element
  - rising edge of C → current value at input D is stored (Q=D)
  - other times → no change of Q
- Basic block of all our sequential circuits
- n flip-flops can represent 2^n states



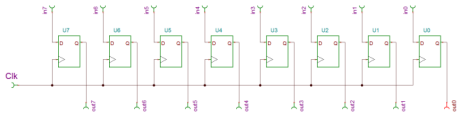


Counter

- Simple form of sequential logic (finite state machine)
- State changes with rising clock edge
- Next state depends only on current state (sequence of states cannot be influenced from the outside)
- Outputs depend only on internal state, not on any inputs

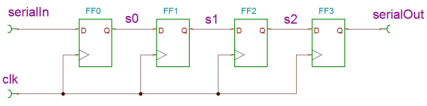
(Parallel) register

input and output are parallel

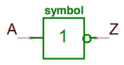


Shift Register

- Chain of connected D-flip-flops
  - Output of FFX is connected to input of FFX+1
  - Input of first FF→ serial input of shift register
  - Output of last FF → serial output of shift register
  - Often parallel reading of data possible through s0, s1,…,sx
  - Parallel write requires multiplexer on each FF input



Inverter



logic equation: Z = !A

truth table

A	Z
0	1
1	0

AND

Z = A & B

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

NAND → NOT AND = AND with inverter

Z = !(A & B)

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

Buffer

Z = A

A	Z
0	0
1	1

OR

Z = A # B

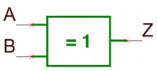
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

NOR → NOT OR = OR with inverter

Z = !(A # B)

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

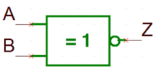
EXOR



Z = A $ B

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

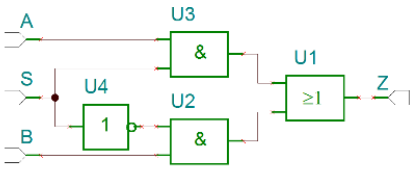
EXNOR



Z = !(A $ B)

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

Example: Multiplexer

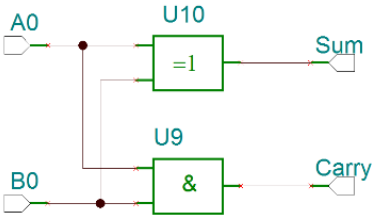


Z = ?

S	A	B	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

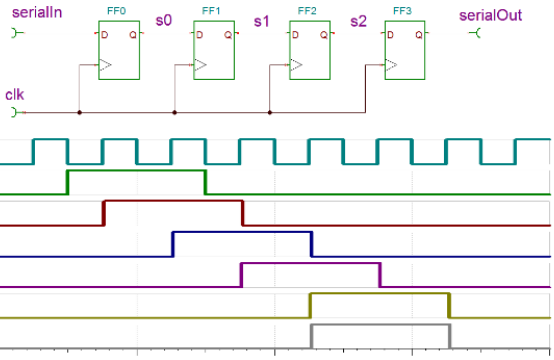
1-Bit Half-Adder

- **Addition of two 1-Bit inputs**

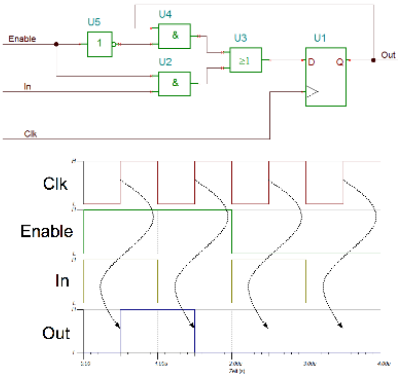


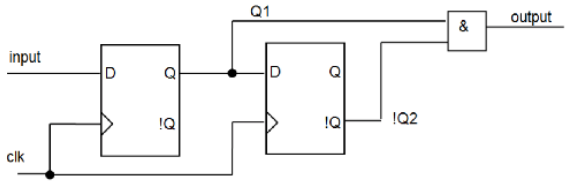
A0	B0	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- **Timing diagram**
  - Input pattern is repeated with a delay on the output of each FF

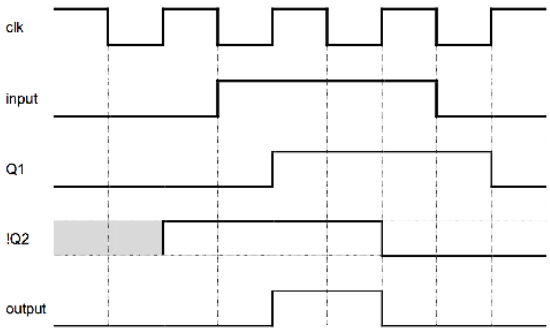


Example: Flip-flop with multiplexer





Ergänzen Sie den folgenden zeitlichen Signalverlauf:



Applications Shift Registers

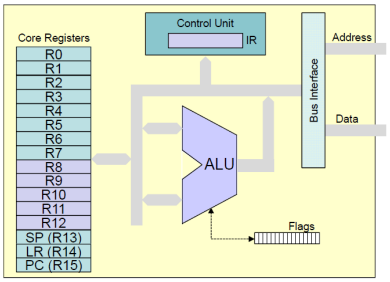
- Ethernet and USB
  - convert serial bit streams to parallel and vice versa
  - serial requires less connections but processer works on parallel data
- Mobile phones, Ethernet, miscellaneous interfaces
  - shift register with feedback for error detection
  - program development: set breakpoints, monitor internal states
  - verification
  - production test → does each transistor / gate work?

Moore machine

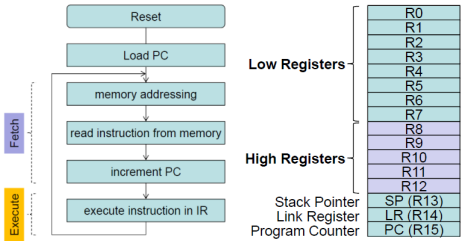
state influenced by inputs

CPU Model

- CPU Components
  - **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
  - 32-bit **ALU**
  - **Flags (APSR)**: N=Negative, Z=Zero, C=Carry, V=Overflow
  - **Control Unit** with IR
    - 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
  - (Instruction Register)
  - **Bus Interface**: Interface between internal CPU bus and external system-bus, contains registers to store addresses



Program Execution Sequence



Instruction Types

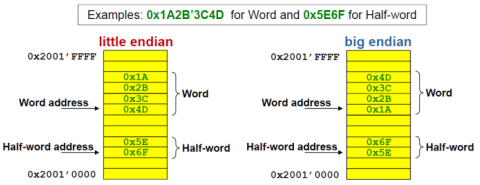
- Data transfer
  - Copy content of one register to another register
  - Load registers with data from memory
  - Store register contents into memory
- Data processing
  - Arithmetic operations → + - * / …
  - Logic operations → AND, OR, …
  - Shift / rotate operations
- Control Flow
  - Branches
  - Function calls
- Miscellaneous

Integer Types

C-Type	Size	Term	inttypes.h / stdint.h
unsigned char	8 Bit	Byte	uint8_t
unsigned short	16 Bit	Half-word	uint16_t
unsigned int	32 Bit	Word	uint32_t
unsigned long	32 Bit	Word	uint32_t
unsigned long long	64 Bit	Double-word	uint64_t
signed char	8 Bit	Byte	int8_t
signed short	16 Bit	Half-word	int16_t
signed int	32 Bit	Word	int32_t
signed long	32 Bit	Word	int32_t
signed long long	64 Bit	Double-word	int64_t

Little / big endian

- 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

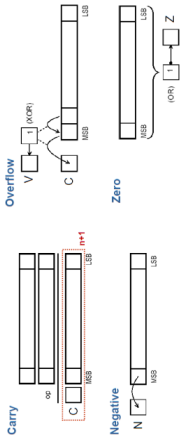


Alignment

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

Object File Sections

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



Load/Store vs. Register Memory

- 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

Types of data transfer instructions

- Register to register
- Loading data
- Loading literals
- Storing data

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

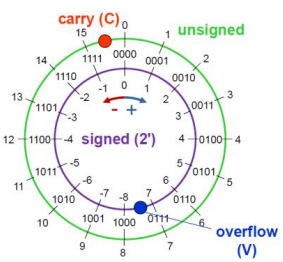
APSR: Application Program Status Register

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

Arithmetic Instructions

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-NOT(c)
RSBS	Reverse Subtract	(negative)-1•A
MULS	Multiplication	A•B

Negative Numbers



2' Complement

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

OC: 1' complement, TC: 2' complement

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

Addition

- Unsigned
  - C=1 indicates carry
  - V irrelevant
  - Addition of 2 big numbers can yield a small result
- Signed
  - V=1 indicates overflow
  - possible with same „sign"
  - C irrelevant

Subtraction

- There is no subtraction! Use addition of 2' complement instead
- unsigned
  - Use 2' complement as well
  - Example 4-Bit unsigned: $12 - 3 = 9$
  - Attention
    - $C = 1 \rightarrow$ NO borrow
    - $C = 0 \rightarrow$ borrow
  - Borrow
    - operation yields negative result → cannot be represented in unsigned
    - in multi-word operations, missing digits are borrowed from more significant word
  - 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

Assembly Program

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

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

Memory address *Opcode* *Listfile*

```
00000000 20A5      demoprg MOVS   R0,#0xA5    ; copy 0xA5 into R0
00000002 2111              MOVS   R1,#0x11    ; copy 0x11 into R1
00000004 1840              ADDS   R0,R0,R1    ; add contents of R0 and R1
                                              ; 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
0000000A 00002000
```

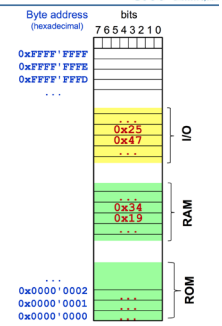
Binary ← *generated by Assembler (tool)* — *Source code*

Memory Map

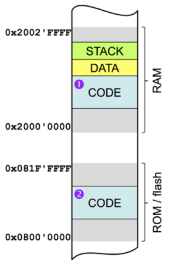
Memory Layout

- 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 bottom and highest address at the top.



- **CODE**
  - Read-only → RAM or ROM
  - Instructions (opcodes)
  - Literals [1]
- **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

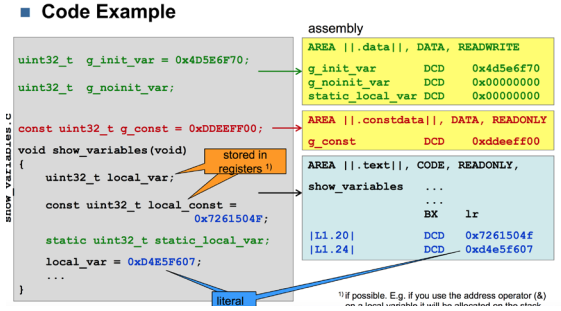


[1] Literal: a fixed/constant value in source code

[2] The data section is sometimes further sub-divided into a read-only and a read-write part

Object File Sections

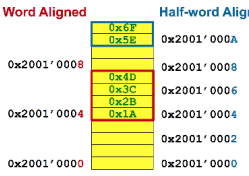
Code Example



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

Alignment

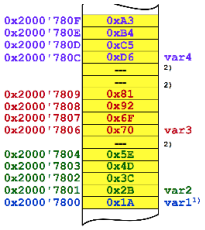
- Half-word aligned — Variables aligned on even addresses
- Word aligned — Variables aligned on addresses that are divisible by four



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
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

- Directives for uninitialized data
  - `SPACE` or `%` with number of bytes to be reserved

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

Example 1 ADDITION

The problem: P = Q + R + S
Let Q = 2, R = 4, S = 5. Assume that r1 = Q, r2 = R, r3 = S. The result Q will go in r0.

The Code

```
ADD r0,r1,r2      ;add Q to R and put in P
ADD r0,r0,r3      ;add S to P and put the result in P
```

Example 2 ADDITION

This problem is the same as Example 1. P = Q + R + S
Once again, let Q = 2, R = 4, S = 5 and assume r1 = Q, r2 = R, r3 = S. In this case, we will put the data in mer
the form of constants before the program runs.

The Code

```
MOV  r1,#Q      ;load Q into r1
MOV  r2,#R      ;load R into r2
MOV  r3,#S      ;load S into r3
ADD  r0,r1,r2   ;Add Q to R
ADD  r0,r0,r3   ;Add S to (Q + R)
```

Example 3 ADDITION

The problem once again is P = Q + R + S. As before, Q = 2, R = 4, S = 5 and we assume that r1 = Q, r2 = R, r3 = S.

In this case, we will put the data in memory as constants before the program runs. We first use the load register, `LDR r1,Q` instruction to load register r1 with the contents of memory location Q. This instruction *does not exist* and is not part of the ARM's instruction set. However, the ARM assembler automatically changes it into an actual instruction. We call `LDR r1,Q` a *pseudoinstruction* because it behaves like a real instruction. It is indented to make the life of a programmer happier by providing a shortcut.

The Code

```
LDR   r1,Q      ;load r1 with Q
LDR   r2,R      ;load r2 with R
LDR   r3,S      ;load r3 with S
ADD   r0,r1,r2  ;add Q to R
ADD   r0,r0,r3  ;add in S
STR   r0,Q      ;store result in Q
```

Example 4 ADDITION

The problem

P = Q + R + S where Q = 2, R = 4, S = 5. In this case we are going to use register indirect addressing to access the variables. That is, we have to set up a pointer to the variables and access them via this pointer.

The Code

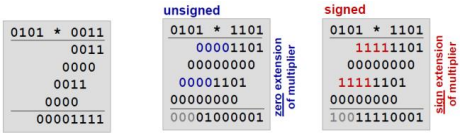
```
ADR   r4,TheData   ;r4 points to the data area
LDR   r1,[r4,#Q]   ;load Q into r1
LDR   r2,[r4,#R]   ;load R into r2
LDR   r3,[r4,#S]   ;load S into r3
ADD   r0,r1,r2     ;add Q and R
ADD   r0,r0,r3     ;add S to the total
STR   r0,[r4,#P]   ;save the result in memory
```

Interpretation of carry / borrow flags in 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

Multiplication

- Result requires twice as many binary digits
- Signed and unsigned multiplication are different



C operations with unsigned and signed operators

If one of the operands is unsigned, C performs an implicit cast for the signed values to unsigned

Example n = 32: signed ∈ [-2'147'483'648, 2'147'483'647]

Can lead to strange results (red lines)

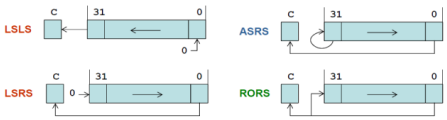
Expression	Type	Evaluation
$0 == 0U$	unsigned	1
$-1 < 0$	signed	1
$-1 < 0U$	unsigned	0
$2'147'483'647 > -2'147'483'647 - 1$	signed	1
$2'147'483'647U > -2'147'483'647 - 1$	unsigned	0
$2'147'483'647 > (int)2'147'483'648U$	signed	1
$-1 > -2$	signed	1
$(unsigned) - 1 > -2$	unsigned	1

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

Shift / Rotate Instructions

Note: rotate left does not exist

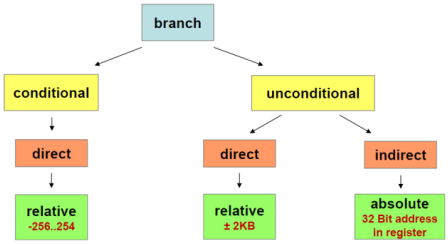


Multiplication with Constants using LSLS and ADDS

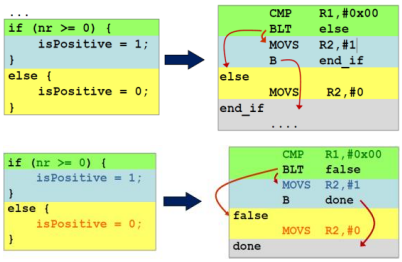
- Example: Multiplication with 13
  - Constant shown as power of 2: $13 = 8 + 4 + 1$
  - $R0 = 13 * R1 \rightarrow R0 = (1 + 4 + 8) * R1 = R1 + 4 * R1 + 8 * R1$
- MOVS R0, R1 ; R0=R1
- LSLS R1, R1, #2; 4•R1
- ADDS R0, R0, R1 ; R0=R0+4•R1
- LSLS R1, R1, #1 ; 2•R1->8•R1
- ADDS R0, R0, R1 ; R0=R0+8•R1

Branch Instructions Properties

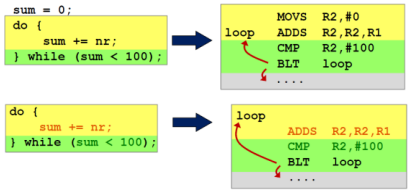
- type
  - **unconditional**: branch always
  - **conditional**: branch only if condition is met
- address of target
  - **relative**: target address relative to PC
  - **absolute**: absolute target address
- address hand-over
  - **direct**: target address part of instruction
  - **indirect**: target address in register



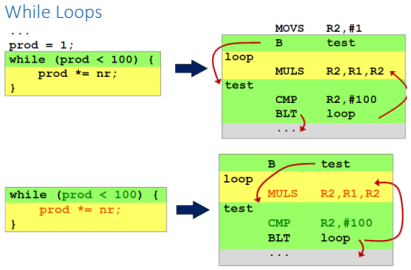
if – then – else



Do-While Loops

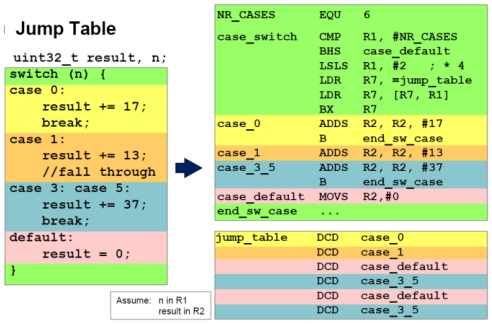


While Loops



Switch Statements

Jump Table



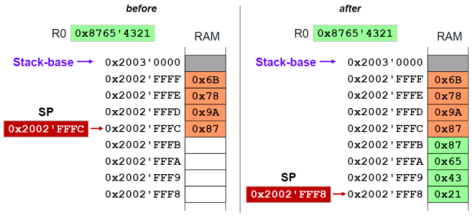
Subroutine / Procedures / Functions / Methods

- 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
- 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.

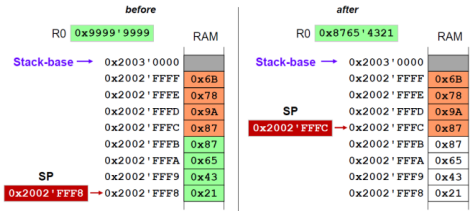
Stack

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

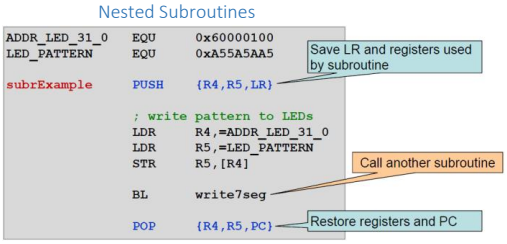
Example: PUSH {R0}



Example: POP {R0}



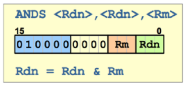
Nested Subroutines



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)
```

- **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?**

Bit Manipulations

- **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
```

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

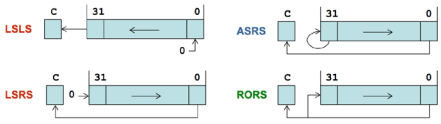
  - Divide by 2^n

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

LSRS and ASRS differ!

- **Overview**

Mnemonic	Instruction	Function
LSLS	Logical Shift Left	2^n • Rn 0 → LSB
LSRS	Logical Shift Right	2^{-n} • Rn 0 → MSB
ASRS	Arithmetic Shift Right	2^{-n} • ±A MSB → MSB
RORS	Rotate Right	LSB → MSB



- **signed ←→ unsigned**

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

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

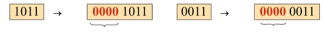
Casts in red area

→ Small negative numbers turn into large positive numbers

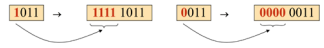
→ Large positive numbers turn into small negative numbers

binary	unsigned	signed 2' 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

- **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 R3
```

Unconditional Branches

- **Direct, relative branch** → `B label`

```
00000002 2112      back      MOVS    R1,#0x12
00000004 2029                MOVS    R0,#0x29
00000006 E003                B       forward
00000008 BF00                NOP
0000000A BF00                NOP
0000000C BF00                NOP
0000000E BF00                NOP
00000010 1A09      forward   SUBS    R1,R1,R0
00000012 E7F6                B       back
```

Forward

```
  0x0000'0006   Address of current instruction
+ 0x0000'0004   1)
+ 0x0000'0006   (0x003 << 1) = 0x006
  0x0000'0010
```

Back

```
  0x0000'0012   Address of current instruction
+ 0x0000'0004   1)
+ 0xFFFF'FFEC   (0x7F6 << 1) = 0xFEC
  0x0000'0002                = -20d
                sign-extended
```

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 and N == V
LE	Signed less than or equal	Z == 1 or N != V

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

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

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

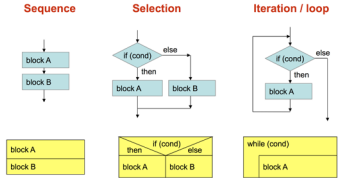
Arithmetic - unsigned

- 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

- **Program flow can be represented with three elements**

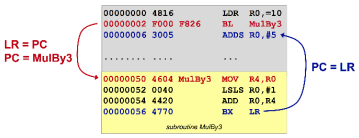


Example

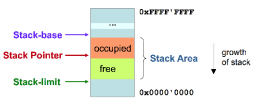
The following example shows a subroutine, `doadd`, that adds the values of two arguments and returns a result in `R0`:

```
        AREA    subrout, CODE, READONLY     ; Name this block of code
        ENTRY                               ; Mark first instruction to execute
start   MOV     r0, #10                     ; Set up parameters
        MOV     r1, #3
        BL      doadd                       ; Call subroutine
stop    MOV     r0, #0x18                   ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                   ; ARM semihosting (formerly SWI)
doadd   ADD     r0, r0, r1                  ; Subroutine code
        BX      lr                          ; Return from subroutine
        END                                 ; Mark end of file
```

- **Change of control flow**
  - Call Save PC to Link Register (LR)
  - Return Restore PC from LR

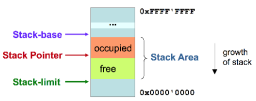


- **Implementation**
  - Stack Area (Section) Continuous area of RAM
  - Stack Pointer - SP R13 → points to last written data value
  - PUSH { ... } Decrement SP and store word(s)
  - POP { ... } Read word(s) and increment SP
  - Direction on ARM "grows" from higher towards lower addresses → full-descending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



Assembler Directives

- 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
```

ABI – Application Binary Interface

- Specification to which independently produced relocatable object files must conform to be statically linkable and executable
  - Function calls
  - Parameter passing
  - In which binary format should information be passed

Parameter Passing

- Where?
  - **Register**: Caller and Callee 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
    - Values in agreed registers,
    - Efficient and simple
    - Limited number of registers
  - **pass by reference**: Handover the address to a value
    - Pass reference (= address) of data structure in register
    - Allows passing of larger structures
  - **Passing through Global Variables** (don't do this!)
    - Shared variables in data area
    - High Overhead in Caller and Callee to access the variable
    - Error-prone, unmaintainable

ARM Procedure Call Standard

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

Callee must preserve contents of these registers (Callee saved)

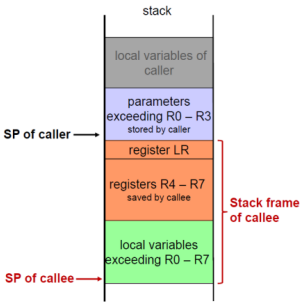
- Scratch Register
  - Used to hold an intermediate value during a calculation
  - Usually, such values are not named in the program source and have a limited lifetime
- Variable Register
  - A register used to hold the value of a variable, usually one local to a routine, and often named in the source 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
  - **128-bit**: return in R0 – R3
- 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

Subroutine Call – Caller Side

1. Subroutine call
   a. PUSH R0 – R3
   b. Copy parameters to R0 – R3
   c. Copy parameters exceeding R0 – R3 on stack
   d. Call Callee
2. On return from subroutine
   a. POP R0 – R3
   b. Get return values from R0 – R3

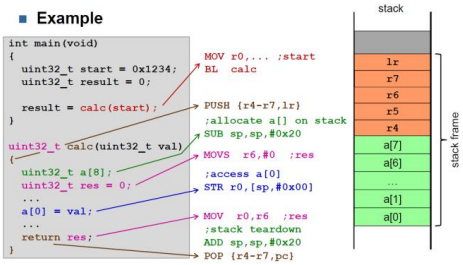
Subroutine Structure – Callee Side

1. Prolog – Entry of subroutine
   a. Save callee saved register contents to stack
   b. Allocate stack space for local variables
   c. Copy input parameters to scratch/variable registers
2. Epilog – Before returning to caller
   a. Restore callee saved registers from stack
   b. Release stack space for local variables
   c. Store result in R0 – R3Stack Frame

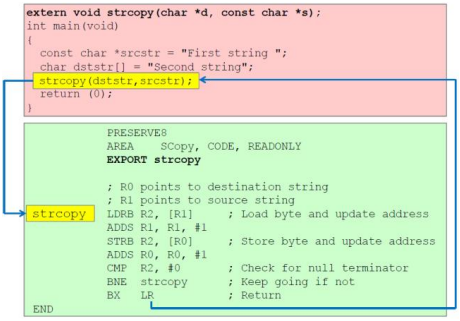


Functions - Stack Frame Example C – Assembler

- **Example**



Calling Assembly Subroutines from C



Polling

- 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

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-System Cortex-M3/M4

- Nested Vectored Interrupt Controller (NVIC)
  - Many sources can trigger exception with high level signal, e.g. 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 (Causes branch to ISR)

Initialization of Interrupt IRQ0_Handler example

```
                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
  - Stores xPSR, PC, LR, R12, R0 – R3 on stack
    - Program Status Registers (PSRs)
      - APSR Application Program Status Register
      - IPSR Interrupt Program Status Register
      - EPSR Execution Program Status Register
  - 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)

Exception states

- **Inactive**: Exception is not active and not pending
- **Pending**: Exception is waiting to be serviced by CPU (E.g. an interrupt event occurred (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

NVIC Registers

- Interrupt Pending Registers
  - Trigger hardware interrupt by software → set pending bit
  - Cancel a pending interrupt → clear pending bit
- Interrupt Active Status Registers
  - Read-only
  - Corresponding bit is set when ISR starts
  - Corresponding bit is cleared when interrupt return is executed
- Interrupt Enable Registers
  - Individual masking of interrupt sources
    - IEn cleared pending bit not forwarded
    - IEn set interrupt enabled
- Priority Level Registers
  - The lower a priority level, the greater the priority
  - 4-bit priority level 0x0 – 0xF

Register / "pass by value"

```
AREA exData,DATA,READWRITE
  ...
AREA exCode,CODE,READONLY
  ...
  MOVS  R1,#0x03
  BL    double
  MOVS  ...,R0
  ...

double FUNCTION
  LSLS  R0,R1,#1
  BX    LR
  ENDFUNC
```

- **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"

```
TABLE_LENGTH    EQU     16

AREA exData,DATA,READWRITE
p1Table SPACE TABLE_LENGTH

AREA exCode,CODE,READONLY
  ... 1)
  LDR   R0,=p1Table
  MOVS  R1,#TABLE_LENGTH
  BL    doubleTableValues
  ...

doubleTableValues FUNCTION
      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
  ENDFUNC
```

- **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

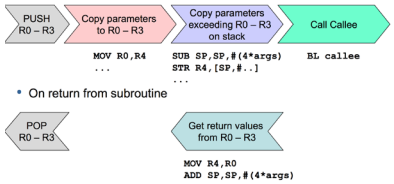
Global Variables

```
AREA exData,DATA,READWRITE
param1  SPACE 1
result  SPACE 1

AREA exCode,CODE,READONLY
  ...
  LDR     R4,=param1
  MOVS    R5,#0x03
  STRB    R5,[R4]
  BL      double_g
  LDR     R4,=result
  LDRB    ...,[R4]
  ...
double_g PROC
  LDR     R4,=param1
  LDRB    R1,[R4]
  LSLS    R0,R1,#1
  LDR     R4,=result
  STR     R0,[R4]
  BX      LR
  ENDP
```

- **Shared variables in data area**
  - param1 Caller → procedure
  - result Return value
- **High Overhead in Caller and Callee to access the variable**
- **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

Subroutine Call – Caller Side

- Subroutine call



- On return from subroutine

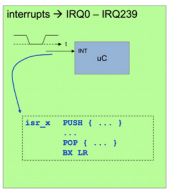
Functions - Stack Frame

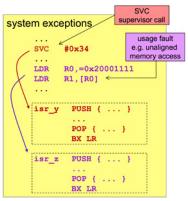
- **Example**

```
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}
```

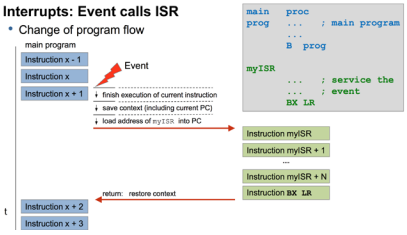
Examples





Interrupts: Event calls ISR

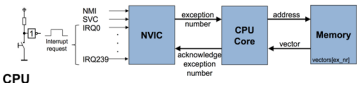
- Change of program flow




```
main    proc
prog    ...     ; main program
        ...
        B   prog

myISR
        ...     ; service the
        ...     ; event
        BX LR
```

Nested Vectored Interrupt Controller (NVIC)

- Many sources can trigger exception with high level signal, e.g. IRQx
- Forwards respective exception number to CPU



CPU

Which ISR Shall the Processor Call?

- Each exception has a different ISR

Memory Addr.	31 … 0	Exception Nr.	
0x0000'03FC	IRQ239	255	Interrupts 0 – 239 IRQn = Exception n + 16
0x0000'0044	IRQ1	17	
0x0000'0040	IRQ0	16	
0x0000'003C	SysTick	15	System Exceptions 1 – 15
0x0000'0038	PendSV	14	
0x0000'002C	SVC	11	
0x0000'000C	Hard Fault	3	
0x0000'0008	NMI	2	
0x0000'0004	Reset	1	
0x0000'0000	Top of Stack	0	

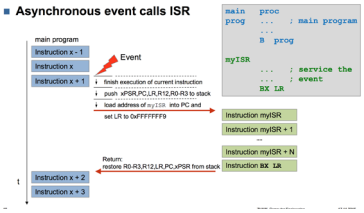
Vector Table (Cortex-M3/M4)

- **Initialization**

```
; Vector Table Mapped to Address 0 at Reset
        AREA  RESET, DATA, READONLY

__Vectors       DCD     __initial_sp        ; Top of Stack
                DCD     Reset_Handler       ; Reset Handler
                DCD     NMI_Handler         ; NMI Handler
                DCD     HardFault_Handler   ; Hard Fault Handler
                DCD     ...                 } System
                DCD     ...                 } Exceptions
                                            } 15 vectors
                ; Interrupts
                DCD     IRQ0_Handler        ; ISR for IRQ0      }
                DCD     IRQ1_Handler        ; ISR for IRQ1      } Interrupts
                DCD     ...                 }

                AREA  SOURCE_CODE, CODE, READONLY
IRQ0_Handler    PUSH { ... }
                ...         ; interrupt service
                POP { ... }
                BX LR
```

Storing the Context

- **Asynchronous event calls ISR**




```
main    proc
prog    ...     ; main program
        ...
        B   prog

myISR
        ...     ; service the
        ...     ; event
        BX LR
```

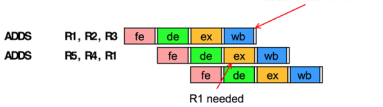
Pipeline Hazards

- **Control Hazards**
  - Occur with branches. The processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline at "fetch" stage
- **Data Hazards**
  - Occur when instructions use data, that is modified in different stages of a pipeline
    - read after write (RAW), a true dependency
    - write after read (WAR), an anti-dependency
    - write after write (WAW), an output dependency
- **Structural Hazards**
  - Occur when a part of the processor's hardware is needed by two or more instructions at the same time

Data Hazards

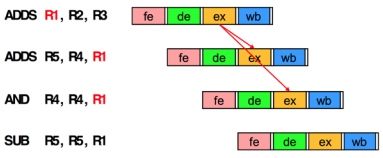
- **Example: read after write, RAW**
  - Not occurring on 3-stages pipeline → Not on Cortex M3
  - On 4-stages pipeline: Fetch, Decode, Execute, Write Back
  - Example:

```
ADDS    R1, R2, R3  ; R2 + R3 → R1
ADDS    R5, R4, R1  ; R4 + R1 → R5
```



Data Hazards

- **Solution: Forwarding**
  - Result can be forwarded to the ex phase of the next instruction

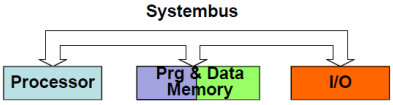


Nested Exceptions: Preemption

- 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

von Neumann Architecture

- 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



Harvard Architecture

- Separate memories for program and data
- Two sets of address/data buses between CPU and memory

Instructions per second (IPS)

- Without pipelining: $IPS = \frac{1}{instruction\ delay}$
- With pipelining: $IPS = \frac{1}{\max stage\ delay}$
  - Pipeline needs to be filled first
  - After filling, instructions are executed after every stage

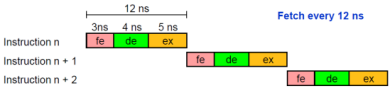
CISC vs. RISC

Complex Instruction Set Computer (CISC)	Reduced Instruction Set Computer (RISC)
Traditional memory access	Load-store architecture
Complex addressing	Only simple addressing
High code density	More lines of code
Most compilers support only sub set of instructions: Powerful instructions are often not used at all	Reduced instruction set: Requires less hardware, allows higher clock rates and use Silicon area for more registers
Often require manual optimization of assembly code for embedded systems	Allow effective compiler optimization with limited, generic instructions
Program needs to wait for external memory	Program works on registers at fullspeed

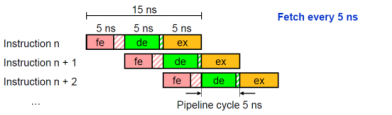
Pipelining Principle

- Advantages
  - All stages are set to the same execution time
  - Massive performance gain
  - Simpler hardware at each stage allows for a higher clock rate
- General
  - Number of execution stages is design decision
  - Typically: 2 – 12 Stages

Sequential execution



Pipelined execution



Code Example Interrupts

```
; -- Constants ------------------------------------------------
                AREA myCode, CODE, READONLY
                THUMB
REG_GPIOA_IDR   EQU  0x40020010
REG_CT_LEDL     EQU  0x60000100
REG_CT_LEDH     EQU  0x60000102
REG_CT_7SEG     EQU  0x60000114
REG_SETENA0     EQU  0xe000e100
PATTERN_CW      EQU  0x00ff
PATTERN_CCW     EQU  0xff00
; -- Main -----------------------------------------------------
main    PROC
        EXPORT main
        BL   init_analog
        BL   init_control
        BL   init_measurement
            ; Configure NVIC (enable interrupt channel)
        LDR  R0,=REG_SETENA0  ; load addr for enabeling IRQs
        MOVS R1,#0x80         ; prepare part 1 of mask
        LSLS R1,#21           ; shift to get 0x08000000
        ADDS R1,#0x40         ; R1 = 0x08000040
        STR  R1,[R0]          ; enable IRQ6
            ; Initialize variables
        LDR  R0,=direction    ; load address of direction
        MOVS R1,#0            ; R1 = default value
        STRH R1,[R0]          ; set default direction
        LDR  R0,=counter      ; load address of counter
        STRH R1,[R0]          ; set default counter
        LDR  R0,=speed        ; load address of speed
        STRH R1,[R0]          ; set default speed
loop
        BL   do_analog        ; Read + output motor control
        BL   do_input         ; Check DIPSW and react
            ; Display speed and direction
        LDR  R0,=direction    ; load address of direction
        LDRH R0,[R0]          ; read current direction
        LDR  R1,=REG_CT_LEDL  ; load address of LEDs
        STRH R0,[R1]          ; write direction to LEDs
        LDR  R0,=speed        ; load address of speed
        LDRH R0,[R0]          ; read current speed
        LDR  R1,=REG_CT_7SEG  ; load address of SEG7 display
        STRH R0,[R1]          ; write speed to SEG7 display
        B    loop
        ENDP
; Handler for EXTI0 interrupt ------------------------------
EXTI0_IRQHandler    PROC
        EXPORT EXTI0_IRQHandler  ; export routine
        PUSH {LR}                ; push LR
        LDR  R0,=REG_GPIOA_IDR   ; load addr of GPIO register
        LDRB R0,[R0]             ; read GPIO bits
        MOVS R1,#0x08            ; mask for bit nr 3
        MOVS R2,#0xff            ; R2 = singal for 8 full LEDs
        ANDS R0,R0,R1            ; maks GPIO bits
        CMP  R0,R1               ; compare R0 with R1
        BNE  next
        LSLS R2,#8               ; leftshift R2 -> R2 = 0xff00
next
```



```
        LDR  R0,=direction      ; get address of direction
        STRH R2,[R0]            ; store current direction in
        LDR  R0,=counter        ; get address of counter
        LDRH R1,[R0]            ; load current counter
        ADDS R1,R1,#1           ; increment current counter
        STRH R1,[R0]            ; store new counter
        BL   clear_IRQ_EXTI0    ; clear active flag
        POP  {PC}               ; return
        ENDP
; Handler for TIM2 interrupt -------------------------------
TIM2_IRQHandler    PROC
        EXPORT TIM2_IRQHandler  ; export routine
        PUSH {LR}               ; push LR
        LDR  R0,=counter        ; load address of counter
        LDR  R1,=speed          ; load address of speed
        LDRH R2,[R0]            ; read current counter
        STRH R2,[R1]            ; save current counter as speed
        MOVS R2,#0              ; reset R2
        STRH R2,[R0]            ; reser counter
        BL   clear_IRQ_TIM2     ; clear active flag
        POP  {PC}               ; return
        ENDP
        ALIGN
; -- Variables -----------------------------------------------
                AREA myVars, DATA, READWRITE
direction       SPACE  2  ; space for halfword for direction
counter         SPACE  2  ; space for halfword for counter
speed           SPACE  2  ; space for halfword for speed
; -- End of file ---------------------------------------------
        END
```

Musterlösung Prüfung 1

Aufgabe 1

Im Register R1 steht ein Wert. Schreiben Sie ein Code Fragment in Assembler, welches (ohne andere Bits zu verändern) die folgenden Operationen auf R1 durchführt:

a) Bits 5 und 2 setzen

```
MOVS    R2,#0x24
ORRS    R1,R1,R2
```

b) Bits 6 und 3 löschen

```
MOVS    R2,#0x48
BICS    R1,R1,R2
```

oder

```
MOVS    R2,#0B7
ANDS    R1,R1,R2
```

c) Bits 6 und 4 invertieren

```
MOVS    R2,#50
EORS    R1,R1,R2
```

Aufgabe 2

Ein Assemblerprogramm verwendet folgende Speicherbereiche:
CODE Beginn bei Adresse 0x08001000, Länge 1024 Bytes
DATA Beginn bei Adresse 0x20030400, Länge 512 Bytes
STACK Beginn lückenlos bei nächster Adresse nach DATA, Länge 256 Bytes
Zeichnen Sie die Bereiche in der gegebenen Memory Map ein. Beschriften Sie jeweils die tiefste und die höchste physikalische Adresse jedes Bereiches.

0x2003FFFF

0x200306FF	STACK
0x20030600	256 Bytes
0x200305FF	DATA
0x20030400	512 Bytes
0x080013FF	CODE
0x08001000	1024 Bytes

0x08000000

Aufgabe 6

Für Berechnungen wurden für die folgenden unsigned Variablen Speicherplatz reserviert:

```
DATA
Zahl1   DCB ?
Zahl2   DCB ?
```

Codieren Sie die folgenden Ausdrücke in Assembler möglichst effektiv:

a) Zahl1 = Zahl1 Zahl2 (3 Punkte)

```
LDR     R7,=Zahl1
LDRB    R1,[R7]
LDR     R2,=Zahl2
LDRB    R2,[R2]
SUBS    R1,R1,R2   ; Alternativ : SUBS R1,R2
STRB    R1,[R7]
```

b) Zahl1 = Zahl1 + 42 (2 Punkte)

```
LDR     R7,=Zahl1
LDRB    R1,[R7]
ADDS    R1,#42
STRB    R1,[R7]
```

Aufgabe 8

In den Registern R2, R3 und R4 stehen vorzeichenlose 32 Bit Zahlen. Schreiben Sie ein ARM Assemblerfragment, welches die drei Zahlen addiert. Das 64 Bit breite Resultat soll in den Registern R1:R0 liegen.

```
MOVS    R1,#0
MOVS    R5,#0
ADDS    R0,R2,R3
ADCS    R1,R5
ADDS    R0,R0,R4
ADCS    R1,R5
```

Aufgabe 9

Betrachten Sie die folgende Assemblersequenz. Welcher Wert (Hexadezimal) steht nach der Ausführung der letzten Instruktion in den Registern R1 und R2?

```
MOV     R1,#0xC4
MOV     R2,R1
MVN     R1,R1
RSBS    R2,R2,#0
```

R1 = 3Bh ; binär 1100'0100 -> 0011'1011 = 3B
R2 = 3Ch ; NOT AL+1 (Zweierkomplement)

Musterlösung Prüfung 2

Aufgabe 1

Für die Vorzeichenbehafteten 8-bit-Werte R0, R1 und R2 sollen gleichnamige Register verwendet werden. R1 und R2 enthalten bereits die Daten.

a) Codieren Sie den folgenden Code in Assembler:
if (R1 < R2) {R0 = R1;} else {R0 = R2;}

```
        CMP     R1,R2
        BGE     else
then
        MOV     R0,R1
        B       endif
else
        MOV     R0,R2
endif
```

b) Was müssen Sie ändern, wenn unsigned statt signed verwendet wird?

```
BHS     then    ;unsigned higher or same
```

c) Welche Flags warden in a) verwendet, welche in b)?
BLT: N!=V; BLO: C==0

Aufgabe 2

Es seien zwei Variablen wie folgt deklariert:

```
int32_t i,count;
```

Implementieren Sie folgenden for-Schleife in Assembler:

```
for (i=0;i<10;i++) {count++;}
```

Nehmen Sie an, dass i vereits in R0 liegt und count in R1. Beide sind signed.

```
            B       testCond
loopStart   ADDS    R1,#1
            ADDS    R0,#1
testCond    CMP     R0,#10
            BLT     loopStart
```

Aufgabe 4

Schreiben Sie die folgenden Assemblerfragmente:

a) Falls der signed Wert im Register R1 grösser als 37d ist, so soll das Register R1 auf den Wert 20d gesetzt werden. Andernfalls soll es den bestehenden Wert behalten.

```
            CMP     R1,#37
            BLE     endcmp
            MOV     R1,20d
endcmp
```

b) Falls Bit 3 im Register R1 gesetzt ist (=1), soll der Inhalt des Registers R0 um eins nach links verschoben werden. Andernfalls soll nichts geschehen. Alle anderen Register sollen nicht verändert werden.

```
            MOVS    R2,#0x08
            TST     R1,R2
            BNE     endtest
            LSLS    R0,#1
endtest
```

Aufgabe 5

Gegeben ist der folgende C Code:

```
unit8_t ucx = 155;
int8_t cx = (int8_t) ucx;
```

Als welche Dezimalzahl wird der Inhalt der Variable cx nach dem Cast interpretiert?

-101d

Aufgabe 6

Gegeben sind die beiden folgenden Halfword-Tabellen

```
TABLELENGTH     EUQ     32
                AREA    MyAsmVar, DATA, READWRITE
srcTable        SPACE   TABLELENGTH
destTable       SPACE   TABLELENGTH
```

Schreiben Sie ein Assemblerfragment, welches in einer Schleife alle Werte aus der Tabelle srcTable liest, verdoppelt und an der gleichen Position in destTable abspeichert.

```
            MOVS    R0,#0
            LDR     R7,=srcTable
            LDR     R6,=destTable
loop        LDRH    R2,[R7,R0]
            LSLS    R2,R2,#1
            STRH    R2,[R6,R0]
            ADDS    R0,#2
            CMP     R0,TABLELENGTH
            BNE     loop
```

Aufgabe 7

Die Register R1, R7 und R0 enthalten die folgenden Datenwerte:

R1: 0x23B107A4
R7: 0x200048D0
R0: 0x0000000C

Nun wird folgender Befehl ausgeführt:

STR R1,[R7,R0]

Geben Sie an, auf welcher Speicheradresse, welcher Datenwert abgelegt wird:

Speicheradresse	Datenwert
0x200048DF	0x23
0x200048DE	0xB1
0x200048DD	0x07
0x200048DC	0xA4

Aufgabe 9

Zu Beginn des folgenden Programmes steht der Stackpointer SP auf 0x20000200 (entspricht SP0). Bestimmen Sie zum angegeben Zeitpunkt den Inhalt des Stacks. Geben Sie dabei den Bytewert für jede einzelne Adresse an. Notieren Sie rechts daneben den Stackpointer mit dem Index aus dem Kommentar (SP3-SP4). Niteren Sie ganz rechts, um welchen Inhalt es sich handelt.

```
LDR     R0,=0x12341111
LDR     R1,=0x00121001
PUSH    {R0,R1}             ;SP1
MOV     R2,SP
SUB     SP,SP,#4            ;SP2
PUSH    {R2}                ;SP3
LDR     R3,=0x01020304
STR     R3,[Sp,#4]
POP     {SP}
```

Adresse	Byte	SP	Inhalt
0x2000'0200		SP0	
0x2000'01FF	00		
0x2000'01FE	12		
0x2000'01FD	10		R1
0x2000'01FC	01		
0x2000'01FB	12		
0x2000'01FA	34		
0x2000'01F9	11		R0
0x2000'01F8	11	SP1/SP4	
0x2000'01F7	01		
0x2000'01F6	02		
0x2000'01F5	03		R3
0x2000'01F4	04	SP2	
0x2000'01F3	20		
0x2000'01F2	00		
0x2000'01F1	01		R2/SP1
0x2000'01F0	F8	SP3	

Dec	Hex	Oct	Bin
0	0	000	00000000
1	1	001	00000001
2	2	002	00000010
3	3	003	00000011
4	4	004	00000100
5	5	005	00000101
6	6	006	00000110
7	7	007	00000111
8	8	010	00001000
9	9	011	00001001
10	A	012	00001010
11	B	013	00001011
12	C	014	00001100
13	D	015	00001101
14	E	016	00001110
15	F	017	00001111
16	10	020	00010000
17	11	021	00010001
18	12	022	00010010
19	13	023	00010011
20	14	024	00010100
21	15	025	00010101
22	16	026	00010110
23	17	027	00010111
24	18	030	00011000
25	19	031	00011001
26	1A	032	00011010
27	1B	033	00011011
28	1C	034	00011100
29	1D	035	00011101
30	1E	036	00011110
31	1F	037	00011111
32	20	040	00100000
33	21	041	00100001
34	22	042	00100010
35	23	043	00100011
36	24	044	00100100
37	25	045	00100101
38	26	046	00100110
39	27	047	00100111
40	28	050	00101000
41	29	051	00101001
42	2A	052	00101010
43	2B	053	00101011
44	2C	054	00101100
45	2D	055	00101101
46	2E	056	00101110
47	2F	057	00101111
48	30	060	00110000
49	31	061	00110001
50	32	062	00110010
51	33	063	00110011
52	34	064	00110100
53	35	065	00110101
54	36	066	00110110
55	37	067	00110111
56	38	070	00111000
57	39	071	00111001
58	3A	072	00111010
59	3B	073	00111011
60	3C	074	00111100
61	3D	075	00111101
62	3E	076	00111110
63	3F	077	00111111

Dec	Hex	Oct	Bin
64	40	100	01000000
65	41	101	01000001
66	42	102	01000010
67	43	103	01000011
68	44	104	01000100
69	45	105	01000101
70	46	106	01000110
71	47	107	01000111
72	48	110	01001000
73	49	111	01001001
74	4A	112	01001010
75	4B	113	01001011
76	4C	114	01001100
77	4D	115	01001101
78	4E	116	01001110
79	4F	117	01001111
80	50	120	01010000
81	51	121	01010001
82	52	122	01010010
83	53	123	01010011
84	54	124	01010100
85	55	125	01010101
86	56	126	01010110
87	57	127	01010111
88	58	130	01011000
89	59	131	01011001
90	5A	132	01011010
91	5B	133	01011011
92	5C	134	01011100
93	5D	135	01011101
94	5E	136	01011110
95	5F	137	01011111
96	60	140	01100000
97	61	141	01100001
98	62	142	01100010
99	63	143	01100011
100	64	144	01100100
101	65	145	01100101
102	66	146	01100110
103	67	147	01100111
104	68	150	01101000
105	69	151	01101001
106	6A	152	01101010
107	6B	153	01101011
108	6C	154	01101100
109	6D	155	01101101
110	6E	156	01101110
111	6F	157	01101111
112	70	160	01110000
113	71	161	01110001
114	72	162	01110010
115	73	163	01110011
116	74	164	01110100
117	75	165	01110101
118	76	166	01110110
119	77	167	01110111
120	78	170	01111000
121	79	171	01111001
122	7A	172	01111010
123	7B	173	01111011
124	7C	174	01111100
125	7D	175	01111101
126	7E	176	01111110
127	7F	177	01111111

Dec	Hex	Oct	Bin
128	80	200	10000000
129	81	201	10000001
130	82	202	10000010
131	83	203	10000011
132	84	204	10000100
133	85	205	10000101
134	86	206	10000110
135	87	207	10000111
136	88	210	10001000
137	89	211	10001001
138	8A	212	10001010
139	8B	213	10001011
140	8C	214	10001100
141	8D	215	10001101
142	8E	216	10001110
143	8F	217	10001111
144	90	220	10010000
145	91	221	10010001
146	92	222	10010010
147	93	223	10010011
148	94	224	10010100
149	95	225	10010101
150	96	226	10010110
151	97	227	10010111
152	98	230	10011000
153	99	231	10011001
154	9A	232	10011010
155	9B	233	10011011
156	9C	234	10011100
157	9D	235	10011101
158	9E	236	10011110
159	9F	237	10011111
160	A0	240	10100000
161	A1	241	10100001
162	A2	242	10100010
163	A3	243	10100011
164	A4	244	10100100
165	A5	245	10100101
166	A6	246	10100110
167	A7	247	10100111
168	A8	250	10101000
169	A9	251	10101001
170	AA	252	10101010
171	AB	253	10101011
172	AC	254	10101100
173	AD	255	10101101
174	AE	256	10101110
175	AF	257	10101111
176	B0	260	10110000
177	B1	261	10110001
178	B2	262	10110010
179	B3	263	10110011
180	B4	264	10110100
181	B5	265	10110101
182	B6	266	10110110
183	B7	267	10110111
184	B8	270	10111000
185	B9	271	10111001
186	BA	272	10111010
187	BB	273	10111011
188	BC	274	10111100
189	BD	275	10111101
190	BE	276	10111110
191	BF	277	10111111

Dec	Hex	Oct	Bin
192	C0	300	11000000
193	C1	301	11000001
194	C2	302	11000010
195	C3	303	11000011
196	C4	304	11000100
197	C5	305	11000101
198	C6	306	11000110
199	C7	307	11000111
200	C8	310	11001000
201	C9	311	11001001
202	CA	312	11001010
203	CB	313	11001011
204	CC	314	11001100
205	CD	315	11001101
206	CE	316	11001110
207	CF	317	11001111
208	D0	320	11010000
209	D1	321	11010001
210	D2	322	11010010
211	D3	323	11010011
212	D4	324	11010100
213	D5	325	11010101
214	D6	326	11010110
215	D7	327	11010111
216	D8	330	11011000
217	D9	331	11011001
218	DA	332	11011010
219	DB	333	11011011
220	DC	334	11011100
221	DD	335	11011101
222	DE	336	11011110
223	DF	337	11011111
224	E0	340	11100000
225	E1	341	11100001
226	E2	342	11100010
227	E3	343	11100011
228	E4	344	11100100
229	E5	345	11100101
230	E6	346	11100110
231	E7	347	11100111
232	E8	350	11101000
233	E9	351	11101001
234	EA	352	11101010
235	EB	353	11101011
236	EC	354	11101100
237	ED	355	11101101
238	EE	356	11101110
239	EF	357	11101111
240	F0	360	11110000
241	F1	361	11110001
242	F2	362	11110010
243	F3	363	11110011
244	F4	364	11110100
245	F5	365	11110101
246	F6	366	11110110
247	F7	367	11110111
248	F8	370	11111000
249	F9	371	11111001
250	FA	372	11111010
251	FB	373	11111011
252	FC	374	11111100
253	FD	375	11111101
254	FE	376	11111110
255	FF	377	11111111