Differnt parts of an assembly instruction:

```
Label
Instruction (Mnemonic)
Operands
Comment
```

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)

n	2^n	n	2^n	n	2^n
0	1	11	2,048	22	4,194,304
1	2	12	4,096	23	8,388,608
2	4	13	8,192	24	16,777,216
3	8	14	16,384	25	33,554,432
4	16	15	32,768	26	67,108,864
5	32	16	65,536	27	134,217,728
6	64	17	131,072	28	268,435,456
7	128	18	262,144	29	536,870,912
8	256	19	524,288	30	1,073,741,824
9	512	20	1,048,576	31	2,147,483,648
10	1,024	21	2,097,152	32	4,254,967,296

Function	IEC 60617-12 *since 1997*	US ANSI 91 *1984*	DIN 40700 *until 1976*
AND	&		
OR	≥ 1		
Buffer	1		
XOR	= 1		
NOT	1		
NAND	&		
NOR	≥ 1		
XNOR	= 1		

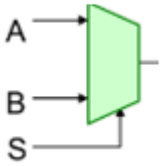
+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

X

Sum

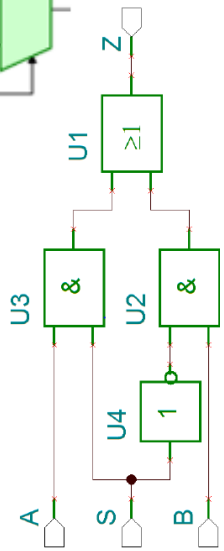
Y

Multiplexer



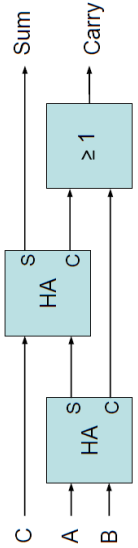
Z=(A&S)#(B&!S)

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

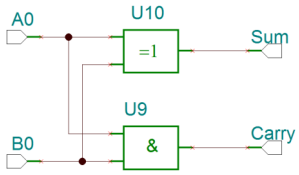


1-Bit Full-Adder

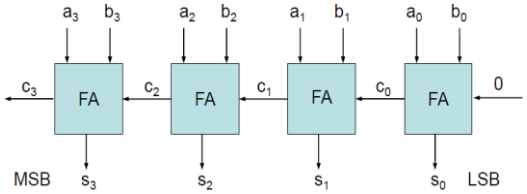
C	A	B	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



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

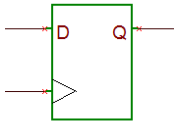
Period T vs. frequency f

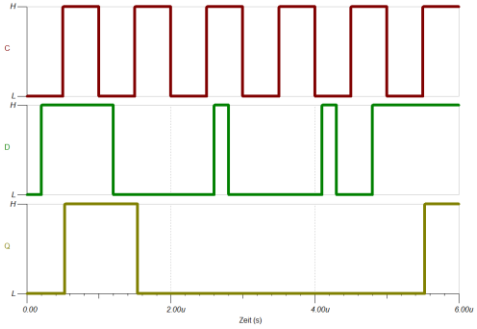
- **Period T**: measured in seconds (s)
- **Frequency f**: measured in Hertz (1/s), i.e. number of cycles per second

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



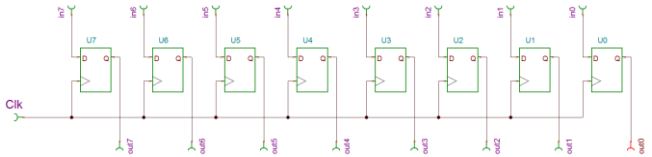


Counter

- 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

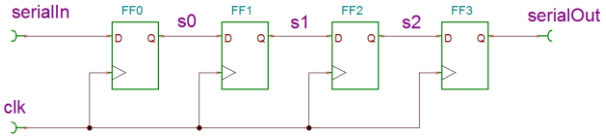
(Parallel) register

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



Applications Shift Registers

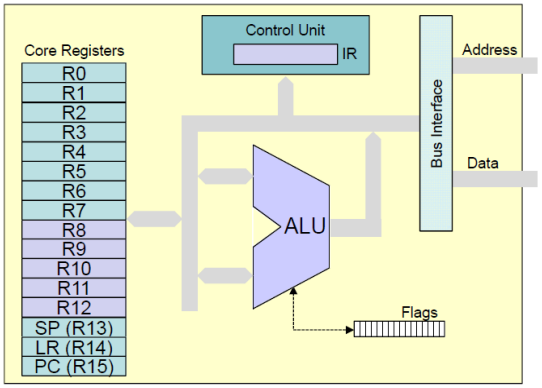
- Ethernet and USB
  - convert serial bit streams to parallel and vice versa
  - serial requires less connections but processer works on parallel data
- Mobile phones, Ethernet, miscellaneous interfaces
  - shift register with feedback for error detection
  - program development: set breakpoints, monitor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

Moore machine

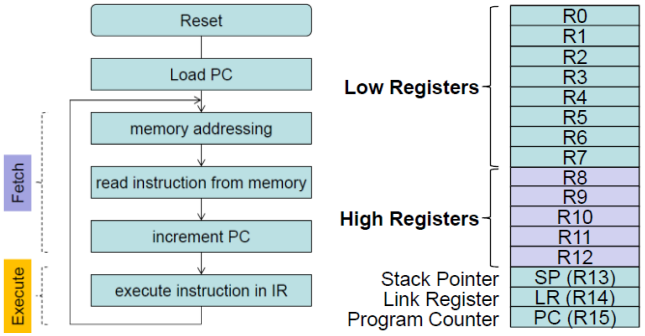
state influenced by inputs

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

LDRH -> **SXTH 16-Bit Sign Extend!**

LDRB -> **SXTB 8-Bit Sign Extend!**

Or use LDRSB
LDRSH
LDRx with sign extension

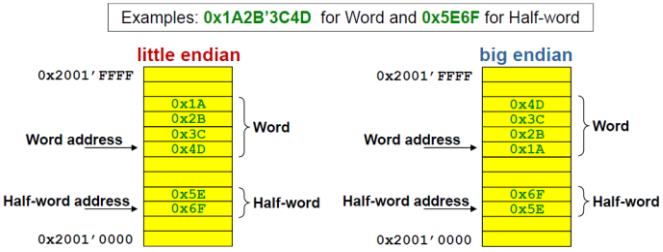
Integer Types

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

DCB offset 1
DCW offset 2
DCD offset 4

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

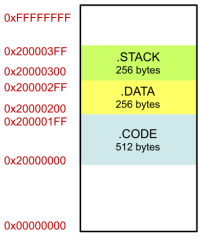


Alignment

- 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

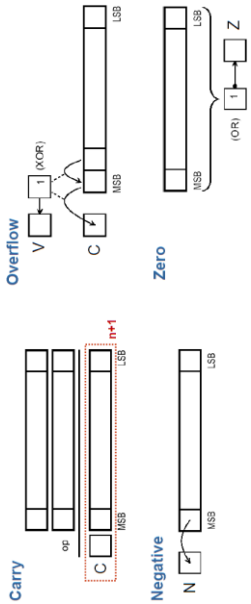
- Load/Store Architecture (ARM Cortex-M)
  - Memory accessed only with load/store operations
  - Usual steps for data processing
    - Load operands from memory to register
    - Execute operation → result in register
    - Store result from register to memory
- Register Memory Architecture e.g. Intel 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

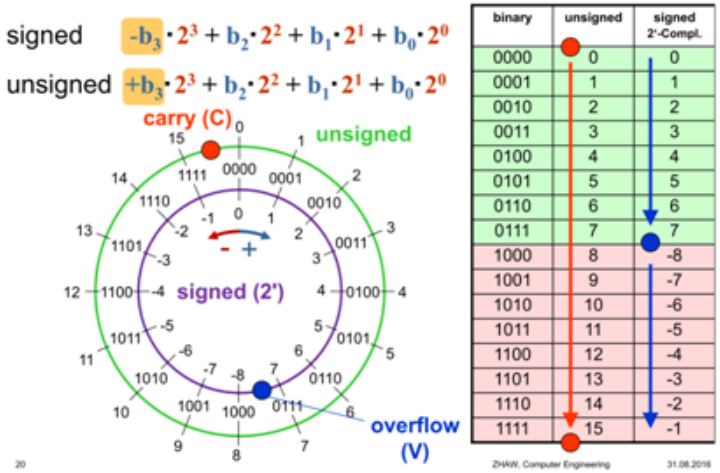


APSR: Application Program Status Register

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



2' Complement

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

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

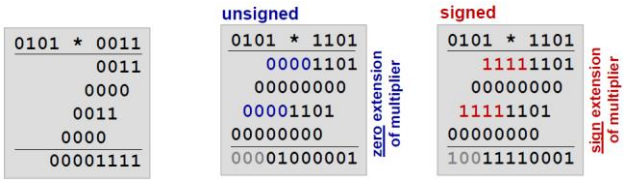
- There is no subtraction! Use addition of 2' complement instead
- unsigned
  - Use 2' complement as well
  - Example 4-Bit unsigned: $12 - 3 = 9$
  - Attention
    - $C = 1 \rightarrow$ NO borrow
    - $C = 0 \rightarrow$ borrow
  - Borrow
    - operation yields negative result → cannot be represented 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

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)

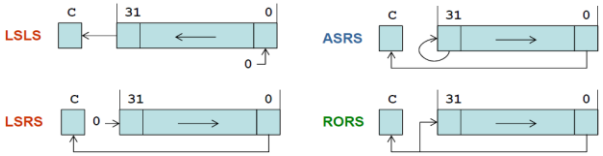
Expression	Type	Evaluation
$0 == 0U$	unsigned	1
$-1 < 0$	signed	1
$-1 < 0U$	unsigned	0
$2'147'483'647 > -2'147'483'647 - 1$	signed	1
$2'147'483'647U > -2'147'483'647 - 1$	unsigned	0
$2'147'483'647 > (int)2'147'483'648U$	signed	1
$-1 > -2$	signed	1
$(unsigned) - 1 > -2$	unsigned	1

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

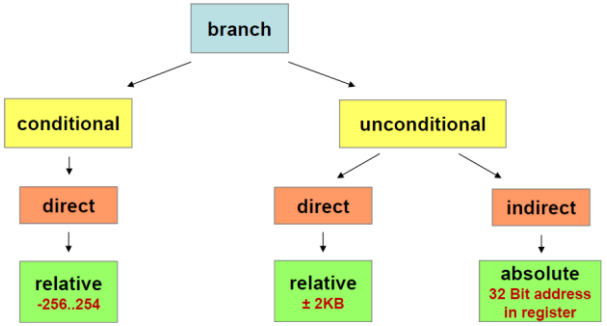


Multiplication with Constants using LSLS and ADDS

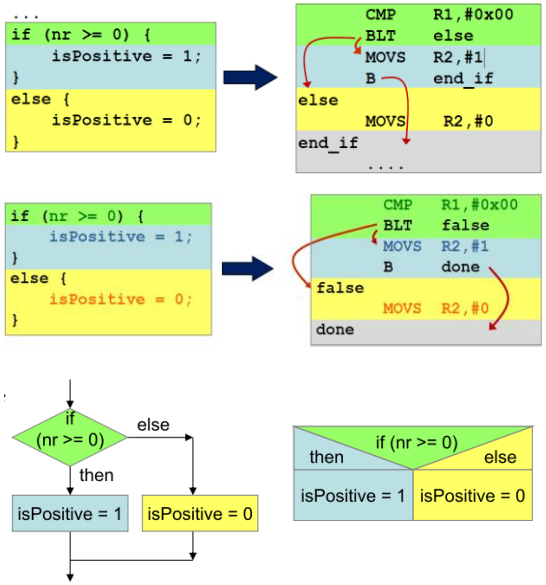
- Example: Multiplication with 13
  - Constant shown as power of 2: $13 = 8 + 4 + 1$
  - $R0 = 13 * R1 \rightarrow R0 = (1 + 4 + 8) * R1 = R1 + 4 * R1 + 8 * R1$
- MOVS R0, R1 ; R0=R1
- LSLS R1, R1, #2; 4•R1
- ADDS R0, R0, R1 ; R0=R0+4•R1
- LSLS R1, R1, #1 ; 2•R1->8•R1
- ADDS R0, R0, R1 ; R0=R0+8•R1

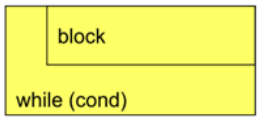
Branch Instructions Properties

- 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

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

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

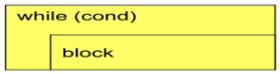
```
do {
    sum += nr;
} while (sum < 100);
```

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

While Loops

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

```
        MOVS   R2,#1
        B      test
loop
        MULS   R2,R1,R2
test
        CMP    R2,#100
        BLT    loop
        ...
```




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

```
        B      test
loop
        MULS   R2,R1,R2
test
        CMP    R2,#100
        BLT    loop
        ...
```

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
```

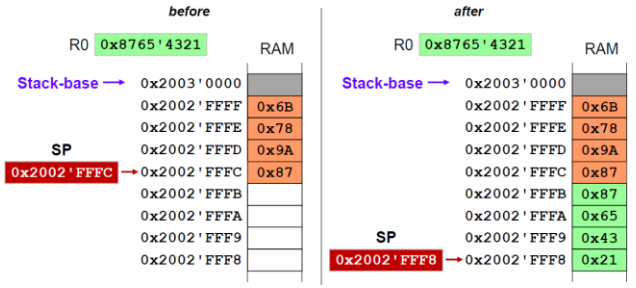
Subroutine / Procedures / Functions / Methods

- 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.

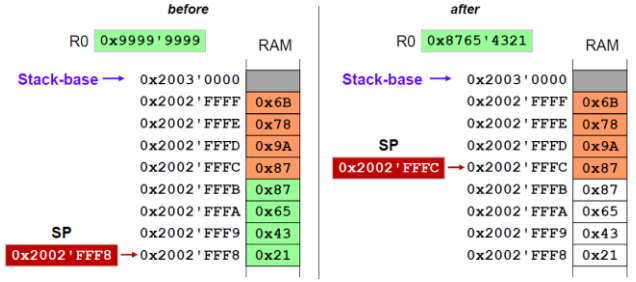
Stack

- **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

```
ADDR_LED_31_0   EQU   0x60000100
LED_PATTERN     EQU   0xA55A5AA5

subrExample     PUSH  {R4,R5,LR}

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

                BL    write7seg

                POP   {R4,R5,PC}
```

Save LR and registers used by subroutine

Call another subroutine

Restore registers and PC

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)

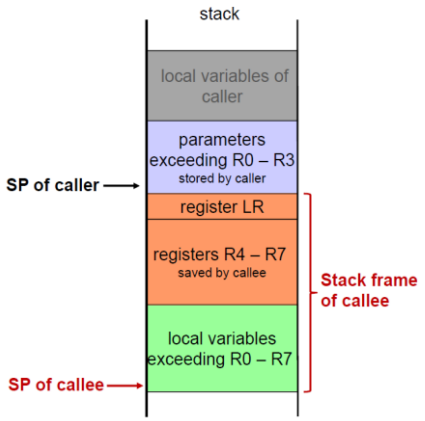
- Scratch Register
  - Used to hold an intermediate value during a calculation
  - Usually, such values are not named in the program source and have a limited lifetime
- Variable Register
  - A register used to hold the value of a variable, usually one local to a routine, and often named in the source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

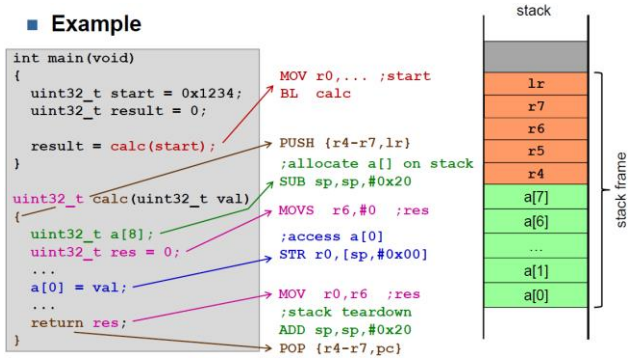
1. Subroutine call
   a. PUSH R0 – R3
   b. Copy parameters to R0 – R3
   c. Copy parameters exceeding R0 – R3 on stack
   d. Call Callee
2. On return from subroutine
   a. POP R0 – R3
   b. Get return values from R0 – R3

Subroutine Structure – Callee Side

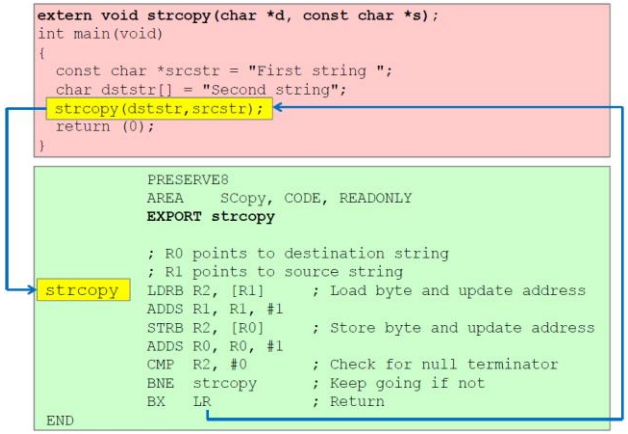
1. Prolog – Entry of subroutine
   a. Save callee saved register contents to stack
   b. Allocate stack space for local variables
   c. Copy input parameters to scratch/variable registers
2. Epilog – Before returning to caller
   a. Restore callee saved registers from stack
   b. Release stack space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



Functions - Stack Frame Example C – Assembler



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

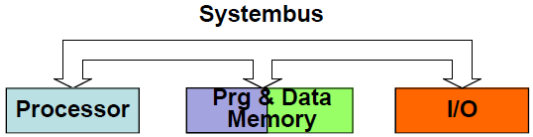
- Interrupt Pending Registers
  - Trigger hardware interrupt by software → set pending bit
  - Cancel a pending interrupt → clear pending bit
- Interrupt Active Status Registers
  - Read-only
  - Corresponding bit is set when ISR starts
  - Corresponding bit is cleared when interrupt return is executed
- 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

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

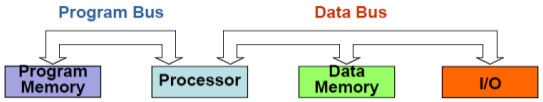
von Neumann Architecture

- 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



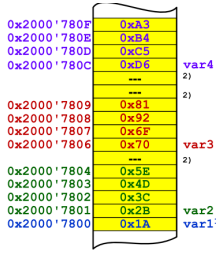
Instructions per second (IPS)

- 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

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
```



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

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

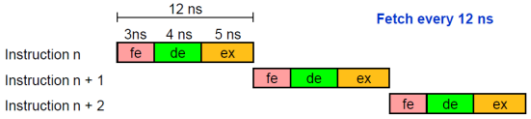
CISC vs. RISC

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

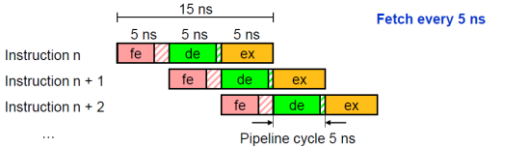
Pipelining Principle

- 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



DCD vs. EQU loading:

```
LDR  Rx,mylita   ;DCD value of mylita !use next one
LDR  Rx,=mylita  ;DCD address of mylita

LDR  Rx,=CONST_A ;EQU
MOVS Rx,#CONST_A ;EQU
```

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
```

Test if IRQ3 is active

```
PL_IRQ3  EQU   0xE000E403
         ...

         LDR   R0, =PL_REG_IRQ3
         MOVS  R1, #0x50
         STRB  R1, [R0]
```

```
ACTIVE0 EQU 0xE000E300
         ...

         LDR  R0, =ACTIVE0
         MOVS R1, #0x08
         LDR  R2, [R0]
         TST  R1, R2
         BEQ  ...
```

Delete Pending IRQ3

```
CLRPENDO EQU 0xE000E280
         ...

         LDR  R0,=CLRPENDO
         MOVS      R1,#0x08
         STR  R1,[R0]
```

Trigger IRQ3 by Software

```
SETPENDO  EQU 0xE000E200
          ...

          LDR  R0,=SETPENDO
          MOVS R1,#0x08
          STR  R1,[R0]
```

ARM and ST use different names for the same register

Disable IRQ3

```
CLRENA0 EQU   0xE000E180
        ...

        LDR   R0, =CLRENA0
        MOVS  R1, #0x08
        STR   R1, [R0]
```

Enable IRQ3

```
SETENA0 EQU   0xE000E100
        ...

        LDR   R0, =SETENA0
        MOVS  R1, #0x08
        STR   R1, [R0]
```

```
        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	

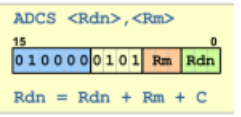
Inhalt

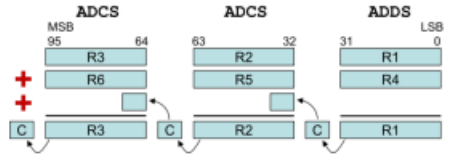
AGBs:
Durch die Verwendung dieses Dokuments erkläre ich (Verwender dieses Dokuments) mich damit einverstanden, Kay ein Bierchen zu spendieren.

Disclaimer:
Alle Angaben (ausgenommen die AGBs) sind ohne Gewähr.

Multi-Word Addition ADCS

- Example: Addition of two 96-bit Operands

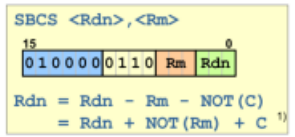


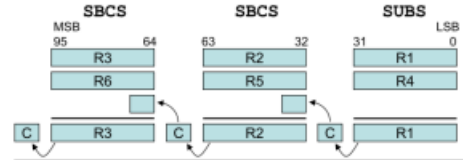



```
; Operand A:        96-bit in R3(MSW),R2,R1(LSW)
; Operand B:        96-bit in R6(MSW),R5,R4(LSW)
; Result = A + B:  96-bit in R3(MSW),R2,R1(LSW)
00000028 1909    ADDS    R1,R1,R4
0000002A 416A    ADCS    R2,R2,R5
0000002C 4173    ADCS    R3,R3,R6
```

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
```

1) –Rm – NOT(C)
= (NOT(Rm) + 1) – NOT(C)
= NOT(Rm) + C

The CPU actually calculates Rdn + NOT(Rm) + C

```
                        AREA    my_data, DATA, READWRITE
00000000 FEDCBA98 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

Gegeben sind die beiden folgenden Halfword-Tabellen

```
TABLELENGTH   EQU     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
```