

Computer Engineering

- Computer Engineering** is where microelectronics and software meet:
- Architecture and organization of computer systems
 - Combines hardware and software to implement a computer
 - Applications in embedded systems, information technology, and technical/scientific tools
 - Historical development spanning over 70 years:
 - 1940s: Relay/vacuum tubes
 - 1950s: Transistors
 - 1970s: Integrated circuits (CMOS)
 - Present: Complex microprocessors with billions of transistors

von Neumann Architecture

The fundamental architecture used in most computers:

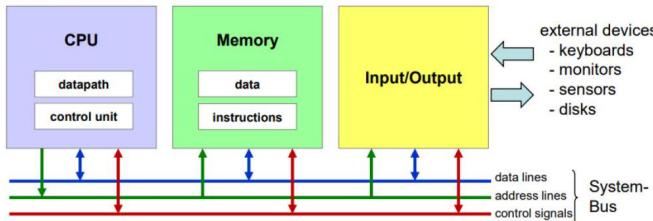
- Single memory for both data and instructions
- Sequential instruction execution
- Components: Control unit, ALU, memory, Input/Output
- Key limitation: Memory bottleneck ("von Neumann bottleneck")

Hardware

Basic Hardware Components

A computer system consists of four fundamental components:

- **CPU (Central Processing Unit)**: Processes instructions and data
- **Memory**: Stores instructions and data
- **Input/Output**: Interface to external devices
- **System Bus**: Electrical connection between components
 - Address lines: Select memory location
 - Data lines: Transfer data (8/16/32/64 bits)
 - Control signals: Coordinate operations



CPU Components The CPU contains several key components:

Datapath:

- **Core Registers**: Fast but limited storage inside CPU
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations

Control Unit:

- Finite State Machine: Reads and executes instructions
- Controls program flow and manages instruction pipeline

Bus Interface: Connects CPU to system bus

Memory

A set of storage cells and the smallest addressable unit is a byte.
 2^N addresses:

- RAM (Random Access Memory): read/write
- ROM (Read-Only Memory): read-only

Memory Types

- **Main Memory (Arbeitsspeicher)**:
 - Connected through System-Bus
 - Access to individual bytes
 - Volatile:
 - * SRAM (Static RAM) - faster, more expensive
 - * DRAM (Dynamic RAM) - needs refresh, cheaper
 - Non-volatile:
 - * ROM - factory programmed
 - * Flash - in-system programmable
- **Secondary Storage**:
 - Connected through I/O
 - Access to blocks of data
 - Non-volatile
 - Examples: HDD, SSD, CD, DVD
 - Slower but cheaper than main memory

Memory Addressing

- Each byte in memory has a unique address
- Address space depends on address bus width:
 - 8-bit address bus: 256 bytes (2^8)
 - 16-bit address bus: 64 KB (2^{16})
 - 32-bit address bus: 4 GB (2^{32})
- Memory map shows allocation of address ranges

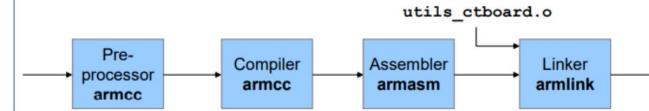
Key concepts for working with memory

1. Memory terms:
 - **Word**: A 32-bit memory unit
 - **Half-word**: A 16-bit memory unit
 - **Word Alignment**: Address is multiple of word size (4)
2. Endianness handling:
 - **Little endian**: LSByte at lower address
 - **Big endian**: MSByte at lower address

Program Translation Process

from C to executable

Translation from source code to executable involves four steps:



1. **Preprocessor**: Text processing
 - Includes header files (#include)
 - Expands macros (#define)
 - Output: Modified source program (.i)
2. **Compiler**: Translates C to assembly
 - CPU-specific code generation
 - Optimization (if enabled)
 - Output: Assembly program (.s)
3. **Assembler**: Converts assembly to machine code
 - Creates relocatable object file
 - Generates symbol table
 - Output: Binary object file (.o)
4. **Linker**: Merges object files into executable
 - Resolves dependencies
 - Relocates addresses
 - Links with libraries
 - Output: Executable file (.axf)

Program Compilation Process

To compile and link a program:

1. Create source files (.c) and header files (.h)
 2. Run preprocessor to expand includes and macros
 3. Compile source files to object files
 4. Link object files and libraries
 5. Test executable
- Common compiler flags:
- **-c**: Compile only, don't link
 - **-o**: Specify output file name
 - **-O[0-3]**: Optimization level
 - **-g**: Include debug information

Simple Program Translation - From Source to Executable

```
// source.c
#include <stdio.h>
#define MAX 100
int main(void) {
    printf("Max is %d\n", MAX);
    return 0;
}
```

After preprocessing (.i):

```
// Contents of stdio.h included here
int main(void) {
    printf("Max is %d\n", 100);
    return 0;
}
```

Assembly output (.s):

```
AREA .text!, CODE, READONLY
EXPORT main
main
PUSH {LR}
LDR R0, =string1
LDR R1, =100
BL printf
MOVS R0, #0
POP {PC}
ALIGN
string1 DCB "Max is %d\n",0
END
```

Host vs Target Development

When developing for embedded systems:

- **Host**: Development computer where code is written and compiled
- **Target**: Embedded system where code will run
- **Cross-compilation**: Compiling on host for different target architecture
- **Tool chain**: Complete set of development tools (compiler, linker, debugger)

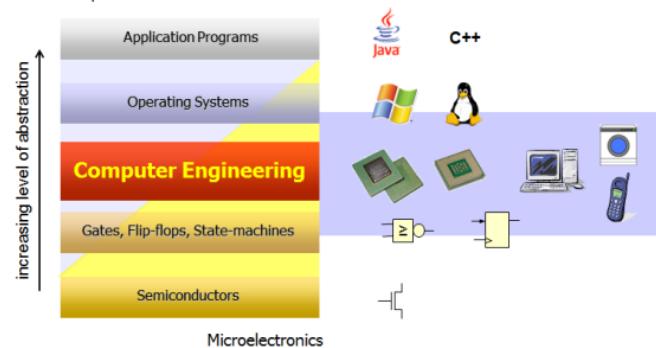
Understanding assembly language is important because it:

- Helps understand machine-level operation
- Aids in debugging and optimization
- Required for system programming
- Essential for security analysis

Examples

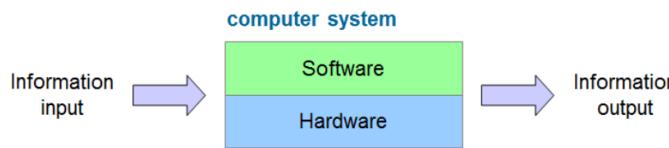
Was ist Computertechnik?

Computer Science



Struktur eines Computersystems

Skizzieren Sie die Struktur eines Computersystems:



Komponenten eines Computersystems

Nennen Sie 4 grundlegende Komponenten eines Computersystems und beschreiben Sie die Aufgaben jeder Komponente:

- **CPU (Central Processing Unit)** oder Prozessor: Führt Anweisungen und Daten aus
- **Memory**: Speichert Anweisungen (Instructions) und Daten
- **Input/Output**: Eingabe/Ausgabe Interface zu externen Geräten
- **System Bus**: Elektrische Verbindung von Funktionseinheiten

Computer System Components

Examples of basic computer system components and their interactions:

```

1 // Example showing interaction between components
2 int main(void) {
3     int data;          // Uses Memory
4     scanf("%d", &data); // Uses I/O
5     data = data * 2;   // Uses CPU/ALU
6     printf("%d", data); // Uses I/O
7
8 }
```

Steuereinheit einer CPU

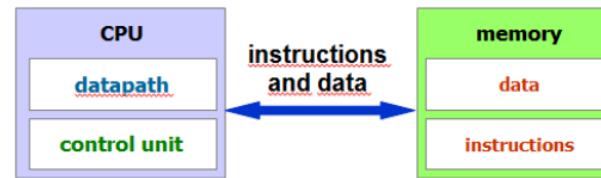
Beschreiben Sie die Aufgaben der Steuereinheit einer CPU:

Die Steuereinheit liest, interpretiert und führt Anweisungen aus. Sie steuert den Programmablauf und verwaltet den Befehlsablauf.

von Neumann Architecture

Erklären Sie die von Neumann Architektur:

- **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



Programmübersetzung

Nennen und erklären Sie die vier Schritte der Programmübersetzung.

Welche Output Dateien werden erzeugt?

Preprocessor:

- Verarbeitet die Preprocessor Statements (z.B. #include, #define)
- Textprocessing: Ersetzen und Ergänzen von Inhalten
- **Output:** Textfile mit modifiziertem C Source Code (.i)

Compiler:

- Übersetzt C Code in prozessorspezifische, symbolische Assemblerbefehle
- Optimierung (falls aktiviert)
- **Output:** Textfile mit menschenlesbarem Assemblercode (.s)

Assembler:

- Übersetzt Assemblercode in binäre Maschinenbefehle
- Erzeugt ein Objectfile mit Maschinenbefehlen (relocatable Object File (.o))

Linker:

- Fügt mehrere Objectfiles zu einem ausführbaren Objectfile zusammen
- Löst die entsprechenden Referenzen zwischen den einzelnen Objectfiles auf (Resolution)
- Passt Referenzen an die tatsächlichen Speicheradressen an (Relocation)
- **Output:** Ausführbares Objectfile - Executable (.axf)

Program Translation Process

Example of program translation stages:

1. Source code (.c):

```

1 #include <stdio.h>
2 #define MAX 100
3
4 int main(void) {
5     printf("Max is %d\n", MAX);
6     return 0;
7 }
```

2. After preprocessing (.i):

```

1 // stdio.h contents included here
2 int main(void) {
3     printf("Max is %d\n", 100);
4     return 0;
5 }
```

3. Assembly output (.s):

```

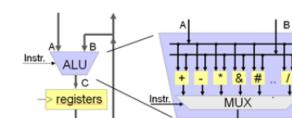
1 AREA .text!, CODE, READONLY
2 EXPORT main
3 main
4     PUSH    {LR}
5     LDR     R0, =string1
6     MOV     R1, #100
7     BL      printf
8     MOVS   R0, #0
9     POP     {PC}
10 string1
11     DCB    "Max is %d\n", 0
12     ALIGN
```

Operationstypen CPU

Welche Operationstypen werden im Allgemeinen von einer CPU unterstützt?

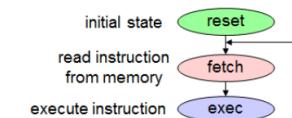
Datapath

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



Control Unit

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



registers

- fast but limited storage inside CPU
- hold intermediate results

■ 4 / 8 / 16 / 32 / 64 bits wide

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

Speicher Typen

Erklären Sie die Unterschiede zwischen Haupt- und Sekundärspeicher.

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



Memory Types and Organization

Example of different memory access patterns:

```

1 ; RAM access example
2 LDR    R0, =ram_data      ; Load RAM address
3 LDR    R1, [R0]           ; Read from RAM
4
5 ; ROM access example
6 LDR    R0, =rom_const     ; Load ROM address
7 LDR    R1, [R0]           ; Read from ROM
8
9 ; Flash memory programming
10 LDR   R0, =FLASH_BASE    ; Flash memory base
11 LDR   R1, =0x12345678    ; Data to write
12 ; Flash write sequence would go here
13
14 section_data
15 ram_data    SPACE    4      ; RAM variable
16 rom_const   DCD     0xFF    ; ROM constant

```

Computer System Debugging

Steps for debugging computer system issues:

- Hardware level:
 - Check power and connections
 - Verify clock signals
 - Test memory access
 - Check I/O interfaces
- Software level:

```

1 ; Debug example
2 PUSH   {R0-R3, LR}        ; Save registers
3 BL     print_debug         ; Call debug routine
4 ; Check specific values
5 LDR   R0, =debug_var
6 LDR   R1, [R0]
7 CMP   R1, #expected
8 BNE   error_handler
9 POP   {R0-R3, PC}

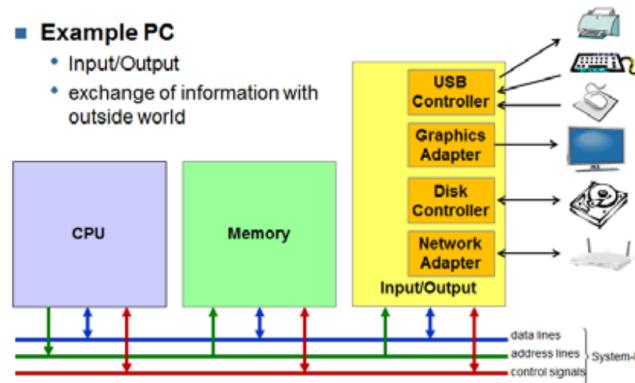
```

PC vs. Embedded System

Erklären Sie den Unterschied zwischen einem Personal Computer (PC) und einem Embedded System bezüglich Aufbau, Anwendung und Programmausführung.

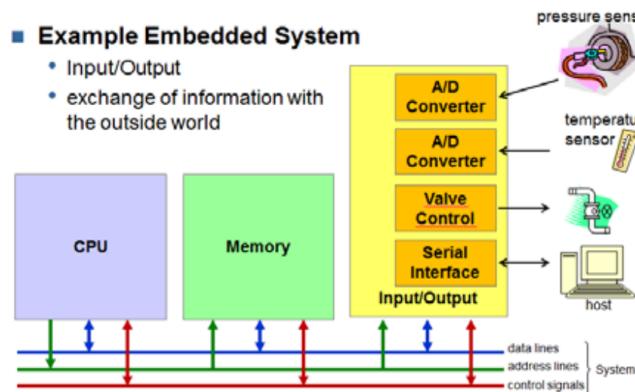
Example PC

- Input/Output
- exchange of information with outside world



Example Embedded System

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



Wichtigkeit dieses Kurses

Wieso ist Wissen um Assemblerprogrammierung wichtig?

- Mit Hilfe von Assembler können wir verstehen, was auf der Maschinenebene abläuft
- Verhalten von Programmen mit Fehlern besser verstehen
- Erhöhen der Performance:
 - vorhandene und nicht vorhandene Optimierungen durch Compiler verstehen
 - Ursachen für ineffiziente Programme finden und beheben
- Implementieren von System Software: Boot Loader, Betriebssysteme, Interrupt Service Routinen
- Lokalisieren und vermeiden von Sicherheitslücken (z.B. Buffer Overflows)

Cortex-M Architecture

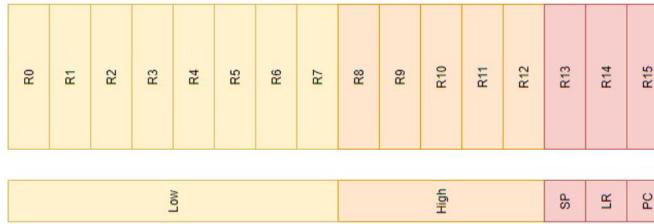
Core Architecture Overview

The ARM Cortex-M is a 32-bit processor architecture designed for embedded systems:

- Load/store architecture
- 32-bit data path
- Thumb instruction set
- Hardware multiply and optional divide
- Harvard architecture variant (separate instruction and data buses)
- Designed for embedded applications:
 - Low cost and power consumption
 - Real-time capabilities
 - Interrupt handling
 - Debug support

Registers

The Cortex-M has 16 core registers, each 32-bit wide:



- **R0-R7:** Lower registers - general purpose
 - Used by most instructions
 - Parameter passing in functions (R0-R3)
 - Results returned in R0
- **R8-R12:** Higher registers - general purpose
 - Limited instruction support
 - Often used for temporary storage
- **R13 (SP): Stack Pointer** - temporary storage
 - Points to current stack position
 - Must be word-aligned (multiple of 4)
- **R14 (LR): Link Register** - return address from procedures
 - Stores return address for function calls
 - Can be saved to stack for nested calls
- **R15 (PC): Program Counter** - address of next instruction
 - Points to next instruction
 - Auto-incremented during execution

Arithmetic Logic Unit (ALU)

32-bit wide processing Unit and supports:

- Arithmetic operations:
 - Addition (ADD, ADC)
 - Subtraction (SUB, SBC)
 - Multiplication (MUL)
 - Division (Optional)
- Logic operations:
 - AND, ORR, EOR (XOR)
 - BIC (Bit Clear)
 - MVN (NOT)
- Shift and rotate operations
- Compare operations

APSR (Flag Register)

The Application Program Status Register (APSR) contains flags:

- **N:** Set when result is negative (bit 31 = 1)
- **Z:** Set when result is zero
- **C:** Set on carry or borrow
- **V:** Set on signed overflow

Instruction suffix 'S' (e.g., ADDS) updates these flags.

Flag Usage Examples

After arithmetic operations with 'S' suffix:

```

1 MOVS  R0, #0xFF    ; R0 = 255 (max unsigned 8-bit)
2 ADDS  R0, #1       ; R0 = 0, Z=1, C=1 (overflow)
3
4 MOVS  R0, #0x7F    ; R0 = 127 (max signed 8-bit)
5 ADDS  R0, #1       ; R0 = 128, N=1, V=1 (signed overflow)
6
7 MOVS  R0, #5
8 SUBS  R0, #10     ; R0 = -5, N=1, C=0 (borrow)

```

Instruction Set

The Cortex-M uses 16-bit Thumb instructions:

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

Main instruction types:

- **Data Transfer:** Move, Load, Store operations
 - MOV/MOVS - Register to register
 - LDR/STR - Memory access
 - PUSH/POP - Stack operations
 - LDM/STM - Multiple register transfer
- **Data Processing:** Arithmetic, logical, shift operations
 - ADD/SUB - Arithmetic
 - AND/ORR/EOR - Logical
 - LSL/LSR/ASR - Shifts
 - CMP/CMN - Compare
- **Control Flow:** Branch and function calls
 - B - Branch
 - BL - Branch with Link
 - BX - Branch and Exchange
 - Conditional variants (BEQ, BNE, etc.)

Common Instruction Formats

Register operations:

```

1 ; Register operations
2 ADDS  R0, R1, R2    ; R0 = R1 + R2
3 MOVS  R0, R1        ; R0 = R1
4 ANDS  R0, R1        ; R0 = R0 & R1
5 ; Immediate values
6 MOVS  R0, #100      ; Load immediate value
7 ADDS  R0, R0, #1     ; Add immediate
8 CMP   R0, #10        ; Compare with immediate
9 ; Memory access
10 LDR   R0, [R1]      ; Load from memory
11 STR   R0, [R1], #4  ; Store with offset
12 LDRB  R0, [R1]      ; Load byte

```

Basic Assembly Program Structure

Example of a simple assembly program:

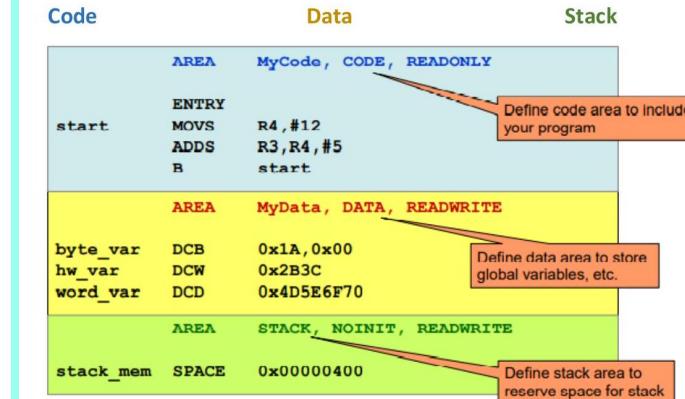
```

1 Label  Instr.  Operands  Comments
2 demoprg MOVS    R0 ,#0xA5 ; copy 0xA5 into R0
3                   MOVS    R1 ,#0x11 ; copy 0x11 into R1
4                   ADDS    R0 ,R0 ,R1 ; add R0 and R1, store in R0

```

Assembly Program Sections

Program memory organized in sections:



Code Section (CODE):

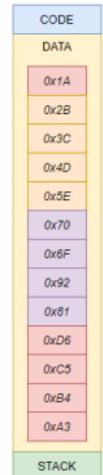
- Contains program instructions
- Usually read-only
- Placed in Flash memory
- Can contain constants (literal pool)

Data Section (DATA):

- Contains global/static variables
- Read-write access
- Placed in RAM
- Initialized at startup

Stack Section: (STACK)

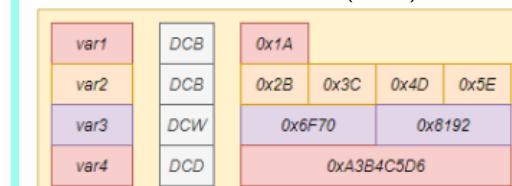
- Dynamic memory allocation
- Used for local variables
- Function call management
- Grows downward in memory



Initialized vs uninitialized Data

Directives for initialized data:

- DCB: Define Constant Byte (8-bit)
- DCW: Define Constant Half-Word (16-bit)
- DCD: Define Constant Word (32-bit)



Directive for uninitialized data:

- SPACE: Reserve specified number of bytes

Data Definition

Memory layout for different data types:

```
1 var1 DCB 0x1A ; single byte
2 var2 DCB 0x2B,0x3C,0x4D,0x5E ; byte array
3 var3 DCW 0x6F70,0x8192 ; half-words
4 var4 DCD 0xA3B4C5D6 ; word
5 data SPACE 100 ; reserve 100 bytes
```

Creating Assembly Programs

Steps for creating an assembly program:

1. Define program sections: (CODE, DATA)

```
1 AREA .text, CODE, READONLY
2 AREA .data, DATA, READWRITE
```

2. Declare external symbols: (IMPORT/EXPORT)

```
1 IMPORT external_func ; External function
2 EXPORT my_function ; Public function
```

3. Define data:

- Define initialized data using DCx directives
- Reserve uninitialized data using SPACE

```
1 AREA .data, DATA, READWRITE
2 var1 DCD 0x1234 ; Word
3 array SPACE 100 ; Reserve space
```

4. Write program code using proper instruction syntax:

```
1 AREA .text, CODE, READONLY
2 ENTRY ; Program entry
3 main
4 ; Your code here
5 END
```

5. End program with END directive!

Common Assembly Patterns

1. Loop with counter:

```
1 MOVS R0, #0 ; Initialize counter
2 loop ; Loop body
3 ADDS R0, #1 ; Increment
4 CMP R0, #10 ; Check condition
5 BLT loop ; Branch if less than
```

2. Memory copy:

```
1 ; R0 = source, R1 = destination, R2 = count
2 copy_loop
3 LDR R3, [R0], #4 ; Load and increment
4 STR R3, [R1], #4 ; Store and increment
5 SUBS R2, #1 ; Decrement counter
6 BNE copy_loop ; Continue if not done
```

3. Function call with parameters:

```
1 MOVS R0, #1 ; First parameter
2 MOVS R1, #2 ; Second parameter
3 BL function ; Call function
4 ; Result in R0
```

Complete Program Example

Program to sum array elements:

```
1 AREA .text, CODE, READONLY
2 EXPORT array_sum
3
4 array_sum
5 MOVS R2, #0 ; Initialize sum
6 MOVS R3, #0 ; Initialize index
7 loop
8 LDR R1, [R0, R3] ; Load array element
9 ADDS R2, R2, R1 ; Add to sum
10 ADDS R3, R3, #4 ; Next element
11 CMP R3, #16 ; Check if done
12 BLT loop ; Continue if not
13 MOVS R0, R2 ; Return sum
14 BX LR ; Return
15 END
```

Examples

CPU Components

Nennen Sie die Hauptkomponenten der M0 CPU und erklären Sie ihre Funktion:

- **Core Registers:**
 - 13 x 32-bit Register für temporäre Speicherung (R0 – R12)
 - 1 x 32-bit Register für Stack Pointer (SP) (R13)
 - 1 x 32-bit Register für Link Register (LR) (R14)
 - 1 x 32-bit Register für Program Counter (PC) (R15)
- **ALU (Arithmetic Logic Unit):**
 - Data processing unit for arithmetic and logic operations
- **Flags:**
 - Processor Status Register: Indicates the state of the processor
- **Control Unit with Instruction Register:**
 - Controls execution of an instruction based on the machine code currently stored in the Instruction Register (IR)
- **Bus Interface:**
 - Interface between CPU and external System Bus
 - Bridge between internal and external bus

Special Purpose Registers

Beschreiben Sie die Funktion folgender Register:

- **PC (Program Counter):**
 - Points to the address where the instructions will next be read
- **SP (Stack Pointer):**
 - Points to the memory addresses where the elements are written/read from the stack
- **LR (Link Register):**
 - Used to keep track of the positions where to jump back (e.g. routines)

Program Counter Initialization

Warum wird der PC bei Reset auf einen definierten Wert initialisiert (während andere CPU Register undefinierte Werte haben können)?

So that fetching of the first instruction can always start at the same (known and predictable) place.

M0 Instruction Types

Name 3 instruction types of the M0 CPU:

- Data transfer
- Data processing
- Flow control

Instruction Execution Analogy

Ein Prozessor führt eine Liste von Instruktionen in einer vordefinierten Reihenfolge aus. Finden Sie Analogien aus dem Alltag:

- Baker/Koch, der einem Rezept folgt
- Musiker, der nach Noten spielt
- Pilot, der vor dem Start eine Checkliste durchgeht

Assembly Code Structure

Name the different parts of a line in assembly code:

```
1 label MOVS R0, #42 ; This is a comment
```

Components:

- Label
- Mnemonic (instruction)
- Operands
- Comment

Terms in Memory

Explain the following terms:

- **Fetch:** Get the instruction from code memory
- **Execute:** Do what the instructions say
- **Word:** A 32-bit memory unit (for the MO)
- **Half-word:** A 16-bit memory unit (for the MO)
- **Little endian:** A multi-byte representation where the LSByte is at the lower address
- **Big endian:** A multi-byte representation where the MSByte is at the lower address
- **Word Alignment:** The address of the multi-byte element is a multiple of the word length (4 for the MO)
- **Data transfer:** Move data between registers and memory
- **Data processing:** Perform arithmetic and logic operations
- **Flow control:** Change the order of execution
- **Stack:** Memory area for procedure calls and local variables
- **Heap:** Memory area for dynamic memory allocation
- **Global variables:** Variables accessible from all functions
- **Local variables:** Variables accessible only within a function
- **Static variables:** Variables with fixed memory allocation
- **Constants:** Fixed values used in the program
- **Code section:** Memory area for instructions
- **Data section:** Memory area for variables
- **Stack section:** Memory area for runtime data
- **Memory map:** Graphical layout showing addresses and sizes of elements
- **Memory areas:** Sections of memory for different purposes
- **Memory sections:** Organized parts of memory for program use
- **Memory ranges:** Address ranges for different memory sections

Memory Map Usage

What is a memory map? What is it used for?

- It is a graphical layout (map) showing the addresses and sizes of elements that communicate with the CPU (memories, Inputs, Outputs)
- The memory map helps users to know where each element is (e.g. when writing the appropriate drivers)

Memory Areas

Which 3 memory areas (sections) can be differentiated for a program?

- **CODE** (read-only → RAM or ROM):
 - Machine instructions
 - Constants

- **DATA** (read-write → RAM):
 - Global variables
 - Static variables
 - Heap in C

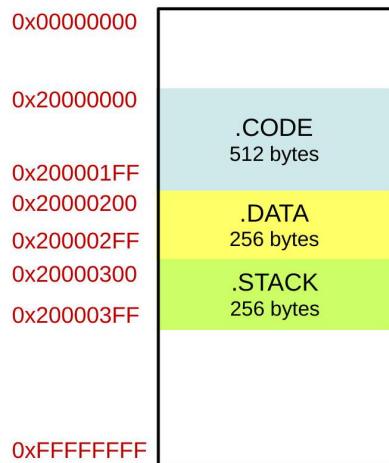
- **STACK** (read-write → RAM):
 - Procedure calls
 - Passing of parameters
 - Local variables

Memory Areas/Memory Map

Assume that the following memory areas are used when executing a program:

- Code: 0x20000000 to 0x200001FF
- Data: 0x20000200 to 0x200002FF
- Stack: 0x20000300 to 0x200003FF

Draw an appropriate memory map and draw the three sections. Label the first and last addresses for each area. How many storage locations does each of the areas contain?



Memory Section Organization

Guidelines for organizing program memory sections:

1. Determine required sections:
 - Code section for instructions
 - Data section for variables
 - Stack section for runtime data
2. Calculate section sizes:
 - Add up all code space needs
 - Account for all global/static variables
 - Estimate maximum stack depth
3. Assign memory ranges:

```

1 AREA    .text,  CODE,  READONLY
2 ; Code section starts here
3
4 AREA    .data,   DATA,  READWRITE
5 ; Data section starts here
6
7 ; Stack section typically defined in startup code
8 Stack_Size EQU    0x400   ; 1KB
9 AREA    STACK,  NOINIT, READWRITE, ALIGN=3
10 Stack_Mem SPACE   Stack_Size

```

Memory Addressing

Wie viele Memory Bytes können mit verschiedenen Adressbreiten adressiert werden?

- 8 bit → 256 Bytes (2^8)
- 16 bit → 64 KBytes = 65'536 Bytes (2^{16})
- 32 bit → 4 GBytes = 4'294'967'296 Bytes (2^{32})

Memory addressing

How many byte positions can be addressed by the M0? Which positions in the memory map need to be occupied? Explain your answer.

The M0 has a 32-Bit address bus. Therefore, it can address 4 GByte = 2^{32} Bytes. Positions needed for the initialization of the processor (at Boot) must be covered by the proper elements (memory). Otherwise, there will be no correct start.

Memory Values Example

A program has variables represented as below in the memory map. Determine the decimal values assuming LittleEndian representation:

Address	Byte content	Variable
0x2FFF'FFF8	0xE2	Var4 (16-bit short)
0x2FFF'FFF9	01100010 (binary)	
0x2FFF'FFFC	0x65	Var3 (32-bit unsigned integer)
0x2FFF'FFFD	10101101(binary)	
0x2FFF'FFFE	0xA3	
0x2FFF'FFFF	0x82	
0x3000'0000	0xA2	Var5 (32-bit integer)
0x3000'0001	34	
0x3000'0002	0x54	
0x3000'0003	0xFF	
0x3000'0004	0x92	Var2 (unsigned char)
0x3000'0005	0x03	Var1 (char)

Solutions:

- Var4 = 0x62E2 = +25'314d (16-bit signed)
- Var3 = 0x82A3AD65 = +2'191'764'837d (32-bit unsigned)

Little Endian vs. Big Endian

A program (code in C) has variables represented as below in the memory map. Determine the decimal values of the variables Var1 ... Var5. Assume Little Endian representation.

Address	Byte content (decimal, hex, binary)	Variable
0x2FFF'FFF7	0x45	
0x2FFF'FFF8	0xE2	Var4 (16-bit short)
0x2FFF'FFF9	01100010 (binary)	
0x2FFF'FFFA	213	
0x2FFF'FFFB	25	
0x2FFF'FFFC	0x65	Var3 (32-bit unsigned integer)
0x2FFF'FFFD	10101101(binary)	
0x2FFF'FFFE	0xA3	
0x2FFF'FFFF	0x82	
0x3000'0000	0xA2	Var5 (32-bit integer)
0x3000'0001	34	
0x3000'0002	0x54	
0x3000'0003	0xFF	
0x3000'0004	0x92	Var2 (unsigned char)
0x3000'0005	0x03	Var1 (char)

Var1 = 0 × 03 = 3d (it is 8-bit unsigned)

Var2 = 0 × 92 = 146 d (it is 8-bit unsigned)

Var3 = 0 × 82A3AD65 = +2'191'764'837d (it is 32-bit unsigned!!)

Var4 = 0 × 62E2 = +25'314d (it is 16-bit signed)

Var5 = 0xFF5422A2 = -1'1263'326 (it is 32-bit signed)

(-2'147'483'648 + 2'136'220'322)

How would the same variable values be stored on a Big Endian platform?
Fill in the table.

Address	Byte content (decimal, hex, binary)	Variable
0x2FFF'FFF7	0x45	
0x2FFF'FFF8	01100010 (binary)	Var4 (short)
0x2FFF'FFF9	0xE2	
0x2FFF'FFFA	213 (or 25)	
0x2FFF'FFFB	25 (or 213)	
0x2FFF'FFFC	0x82	Var3 (32-bit unsigned integer)
0x2FFF'FFFD	0xA3	
0x2FFF'FFFE	10101101(binary)	
0x2FFF'FFFF	0 × 65	
0x3000'0000	0xFF	Var5 (32-bit integer)
0x3000'0001	0x54	
0x3000'0002	34	
0x3000'0003	0xA2	
0x3000'0004	0x92	Var2 (unsigned char)
0x3000'0005	0x03	Var1 (char)

We are not told if 25 / 213 form a unit or not. Therefore, it can be both ways.

Data Transfer

Data Transfer Overview

- ARM Cortex-M uses a load/store architecture:
- Memory can only be accessed through load and store instructions
 - All other operations work on registers
 - Data processing only between registers
 - Various addressing modes for flexible memory access:
 - Immediate offset: Fixed displacement from base
 - Register offset: Variable displacement using register
 - Pre-indexed: Address calculated before access
 - Post-indexed: Address calculated after access
 - Steps: Load operands → Execute → Store result

Two main approaches to memory access, the other one:

Register Memory Architecture (e.g., Intel x86):

- Operations can use memory operands directly
- Results can be written directly to memory
- More flexible but more complex instructions

Load Instructions

Main load instructions for moving data into registers:

- MOVS** (Move and Set flags):
 - Register to Register: MOVS R1, R2
 - 8-bit immediate: MOVS R1, #0x1C
 - Constant: MOVS R1, #MyConst
 - Limitations: Only 8-bit immediates, only low registers
- LDR** (Load Register):
 - 32-bit literal: LDR R1, #0xA1B2C3D4
 - PC-relative: LDR R1, [PC, #12]
 - Pseudo instruction: LDR R1, =MyConst
 - Register indirect: LDR R1, [R2]
 - Immediate offset: LDR R1, [R2, #4]
 - Register offset: LDR R1, [R2, R3]
- LDRB** (Load Register Byte):
 - Loads 8-bit value
 - Bits 31 to 8 are set to zero (Zero extension to 32 bits)
 - Common for arrays of bytes
- LDRH** (Load Register Half-word):
 - Loads 16-bit value
 - Bits 31 to 16 are set to zero (Zero extension to 32 bits)
 - Common for arrays of half-words
- LDRSB/LDRSH** (Load Signed Register Byte/Half-word):
 - Sign extension to 32 bits
 - Used for signed small integers

Load Instruction Examples

```

1 ; MOV examples
2 MOVS R1, #0xFF          ; Load immediate 255
3 MOVS R2, R1              ; Copy R1 to R2
4 ; LDR examples
5 LDR R1, =0x12345678     ; Load 32-bit constant
6 LDR R2, [R1]              ; Load from address in R1
7 LDR R3, [R1, #4]          ; Load with offset
8 LDR R4, [R1, R2]          ; Load with register offset
9 ; Byte/Half-word loads
10 LDRB R1, [R2]            ; Load unsigned byte
11 LDRSB R1, [R2]           ; Load signed byte
12 LDRH R1, [R2]            ; Load unsigned half-word
13 LDRSH R1, [R2]           ; Load signed half-word

```

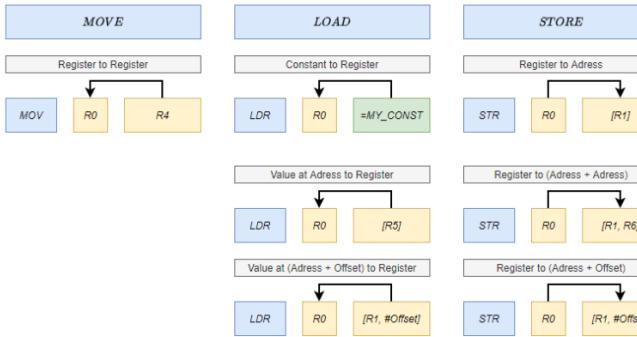
Store Instructions

Instructions for storing data from registers to memory:

- STR** (Store Register):
 - Basic store: STR R1, [R2]
 - With immediate offset: STR R1, [R2, #0x04]
 - With register offset: STR R1, [R2, R3]
 - Word-aligned addresses only
- STRB** (Store Register Byte):
 - Stores lowest 8 bits of register
 - No alignment requirements
- STRH** (Store Register Half-word):
 - Stores lowest 16 bits of register
 - Must be half-word aligned

Memory Access

Memory Layout for array elements and instructions:



Size considerations:

- Array elements: 3 * 4 Bytes
- Instructions: 5 * 2 Bytes
- Literals (0x08): 1 * 4 Bytes

Memory Access Patterns in Load/Store Architecture

Steps for accessing memory:

- Determine required data size (byte, half-word, word)
- Choose appropriate load/store instruction
- Calculate correct memory address
- Consider alignment requirements
- Load/store data using proper addressing mode

Memory Alignment Important alignment rules:

- Word access (LDR/STR):**
 - Address must be multiple of 4
 - Misaligned access causes fault
- Half-word access (LDRH/STRH):**
 - Address must be multiple of 2
- Byte access (LDRB/STRB):**
 - No alignment requirements
- Stack operations:**
 - SP must be word-aligned
 - PUSH/POP automatically maintain alignment

Basic Data Transfer Operations Common data transfer operations:

```

1 ; Load operations
2 MOVS R1, #42          ; Load immediate value
3 MOVS R2, R1            ; Copy register
4 LDR R3, =0x1234        ; Load 32-bit constant
5 LDR R4, [R3]           ; Load from memory
6 LDRB R5, [R3, #1]       ; Load byte with offset
7
8 ; Store operations
9 STR R1, [R2]           ; Store word
10 STRB R1, [R2, #4]      ; Store byte with offset
11 STRH R1, [R2, R3]      ; Store half-word with register offset

```

Common Data Transfer Patterns

1. Copy memory block:

```

; R0 = source, R1 = dest, R2 = count
loop
  LDR R3, [R0], #4        ; Load and increment
  STR R3, [R1], #4        ; Store and increment
  SUBS R2, #1             ; Decrement counter
  BNE loop                ; Continue if not zero

```

2. Initialize memory block:

```

; R0 = start, R1 = value, R2 = count
loop
  STR R1, [R0], #4        ; Store and increment
  SUBS R2, #1             ; Decrement counter
  BNE loop                ; Continue if not zero

```

3. Search memory:

```

; R0 = start, R1 = value to find, R2 = count
loop
  LDR R3, [R0], #4        ; Load and increment
  CMP R3, R1              ; Compare with value
  BEQ found               ; Branch if found
  SUBS R2, #1             ; Decrement counter
  BNE loop                ; Continue if not zero
not_found
  ; Handle not found case
found
  ; Handle found case

```

Register Access

For operations involving high registers (R8-R15):

- Use MOV for register transfers (works with all registers but only for reg/reg transfers)
- Cannot use MOVS with high registers

Arrays

Memory Access Loading and storing array elements:

```

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

AREA myCode, CODE, READONLY
...
; load base and offset registers
LDR R1,=my_array ; load address of array
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

```

- `my_array` = 3 * 4 Bytes
- Instructions = 5 * 2 Bytes
- Literals (`0x08`) = 1 * 4 Bytes

Accessing Array Elements

Steps for array access:

1. Calculate element offset:
 - Byte array: offset = index
 - Half-word array: offset = index * 2
 - Word array: offset = index * 4
2. Choose appropriate instruction:
 - LDRB/STRB for byte arrays
 - LDRH/STRH for half-word arrays
 - LDR/STR for word arrays

Example implementation:

```

1 ; Access array[i] where i is in R1
2 ; Array base address in R0
3
4 ; For byte array
5 LDRB  R2, [R0, R1] ; R2 = array[i]
6
7 ; For half-word array
8 LSLS  R2, R1, #1    ; R2 = i * 2
9 LDRH  R3, [R0, R2] ; R3 = array[i]
10
11 ; For word array
12 LSLS  R2, R1, #2    ; R2 = i * 4
13 LDR   R3, [R0, R2] ; R3 = array[i]

```

Multiple Register Transfer

LDM (Load Multiple) and STM (Store Multiple):

- Load/store multiple registers in one instruction
- More efficient than individual loads/stores
- Used for stack operations (PUSH/POP)
- Register list specified in curly braces

Example:

```

1 LDM   R0!, {R1-R4} ; Load 4 consecutive words
2 STM   R0!, {R1-R4} ; Store 4 consecutive words

```

Multiple Data Transfer Loading/Storing multiple registers:

```

1 ; Store multiple registers
2 PUSH  {R0-R3, LR} ; Push registers to stack
3
4 ; Load multiple registers
5 POP   {R0-R3, PC} ; Pop and return
6
7 ; Load multiple memory locations
8 LDM   R0!, {R1-R4} ; Load 4 words, update R0
9
10 ; Store multiple memory locations
11 STM   R0!, {R1-R4} ; Store 4 words, update R0

```

Multi-Word Data Transfer

For transferring data larger than 32 bits:

1. Loading 96-bit value:

```

1 ; Load 96-bit value from memory
2 ; R3(MSW), R2, R1(LSW) contain result
3 ; Memory address in R6
4 LDM   R6, {R1-R3} ; Load all words at once
5
6 ; Alternative using individual loads:
7 LDR   R1, [R6]      ; Load LSW
8 LDR   R2, [R6, #4]   ; Load middle word
9 LDR   R3, [R6, #8]   ; Load MSW

```

2. Storing 96-bit value:

```

1 ; Store 96-bit value to memory
2 ; R3(MSW), R2, R1(LSW) contain data
3 ; Memory address in R6
4 STM   R6, {R1-R3} ; Store all words at once
5
6 ; Alternative using individual stores:
7 STR   R1, [R6]      ; Store LSW
8 STR   R2, [R6, #4]   ; Store middle word
9 STR   R3, [R6, #8]   ; Store MSW

```

Stack Operations

Stack Access Instructions

Special variants of LDM/STM for stack operations:

- **PUSH {register list}:**
 - Decrements SP
 - Stores registers
 - Example: `PUSH {R0-R3, LR}`
- **POP {register list}:**
 - Loads registers
 - Increments SP
 - Example: `POP {R0-R3, PC}`

Important considerations:

- Always check alignment requirements
- Be aware of endianness (STM32 is little-endian)
- Consider using multiple register transfer for efficiency
- Manage literal pool placement in code
- Stack operations must maintain SP word alignment

Pseudo Instructions

LDR Pseudo Instructions

The LDR pseudo instruction `LDR Rx, =value` is expanded by the assembler:

1. For literal values:
 - Assembler creates 'literal pool' at convenient location
 - Allocates and initializes memory with DCD directive
 - Uses PC-relative addressing to access value
2. For addresses:
 - Places address in literal pool
 - Generates PC-relative load instruction

Example:

```

1 LDR   R1, =0xFF55AAB0 ; Pseudo instruction
2 ; Assembler converts to:
3 LDR   R1, [PC, #offset]
4 ...
5 DCD   0xFF55AAB0 ; In literal pool

```

Pseudo Instruction vs Direct Load The difference between LDR forms:

```

1 LDR   R5, mylita      ; Loads value at mylita
2 LDR   R5, =mylita     ; Loads address of mylita
3
4 mylita DCD 0xFF001122 ; Data definition

```

First instruction loads `0xFF001122`, second loads address of `mylita`.

Pseudo-Instructions

A pseudo-instruction like `LDR Rn, =literal`:

- Does not directly translate to machine code
- Is expanded by assembler into actual instructions
- Decomposed into:
 - DCD directive for place reservation
 - LDR instruction to load from that position
- Can handle:
 - Address literals (position in memory)
 - Constant literals (known at assembly time)

Examples

Load/Store Architecture What is a load/store architecture?

- Data processing is between registers
- Transfer of data from and to the external memory is done using load (memory to register) or store (register to memory) instructions

MOV vs MOVS Instructions

Key differences between MOV and MOVS:

- **MOV**: Transfer does NOT affect flags (low and high registers)
- **MOVS**: Transfer affects flags (only low registers)

Initializing Registers

Different ways to initialize a register with immediate value:

1. Using MOVS:

```
1 MOVS R0, #42 ; Limited to 8-bit values
```

Advantages:

- Value is in instruction
- Less memory space needed

Disadvantages:

- Limited to 8-bit values
- Only works with low registers

2. Using LDR with PC-relative addressing:

```
1 LDR R0, [PC, #12] ; Can load 32-bit values
```

Advantages:

- Can load larger values (up to 32-bit)
- Works with any register

Disadvantages:

- Takes more space in memory
- Requires literal pool management

Data transfer instruction for high registers

Which data transfer instructions should you use if at least one high register is an operand?

MOV works for all registers (but only for reg/reg transfers)

Initializing low registers

List different ways of initializing a low register with an immediate value. What are the advantages/disadvantages?

MOVS <Rd>,#<imm8>

Value to load is in instruction

Less memory space but limited to 8-bit values.

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

Use PC/offset combination to point to the value to load.

Can be used to load larger values (up to 32-bit). Takes more space in memory.

Register Transfer Operations

Steps for data transfer operations:

1. Moving between registers:

```
1 ; Copy contents with flags unchanged
2 MOV R3, R1 ; For any registers
3
4 ; Copy contents and update flags
5 MOVS R3, R1 ; Only for low registers
```

2. Loading immediate values:

```
1 ; Small values (flags unchanged)
2 MOV R0, #42 ; 8-bit immediate
3
4 ; Larger values
5 LDR R0, =0x12345678 ; 32-bit value using literal
   pool
6
7 ; For high registers
8 LDR R0, =value ; Load into low register first
9 MOV R8, R0 ; Then move to high register
```

3. Memory access:

```
1 ; Load from memory
2 LDR R0, [R1] ; Word from address in R1
3 LDRB R0, [R1] ; Byte from address in R1
4
5 ; Store to memory
6 STR R0, [R1] ; Word to address in R1
7 STRB R0, [R1] ; Byte to address in R1
```

Assembly Instructions for Memory Access

Write down the assembly instructions to perform the following actions

- Copy contents of R1 in R3 (flags unchanged)
MOV R3, R1
- Initialize R0 with $0 \times AA$ (flags unchanged)
LDR R0,=0xAA
- Initialize R1 with 234 (flags modified)
MOVS R1,#234
- Initialize R4 with $0 \times 55AA$
LDR R4,=0x55AAC
- Copy contents of R9 in R3
MOV R3, R9
- Initialize R10 with 0×345678
LDR R0,=0x345678 ; R0 as example.
Another low register is ok: MOV R10, R0
- Copy contents of R8 in R9
MOV R9, R8

Array Access

Guidelines for working with arrays:

1. Calculate element offset:

- Byte arrays: offset = index
- Half-word arrays: offset = index $\times 2$
- Word arrays: offset = index $\times 4$

2. Choose appropriate instructions:

```
1 ; Array base in R0, index in R1
2
3 ; For byte array
4 LDRB R2, [R0, R1] ; index offset
5
6 ; For half-word array
7 LSLS R2, R1, #1 ; multiply index by 2
8 LDRH R3, [R0, R2] ; load half-word
9
10 ; For word array
11 LSLS R2, R1, #2 ; multiply index by 4
12 LDR R3, [R0, R2] ; load word
```

3. Consider alignment:

- Words must be aligned to 4-byte boundary
- Half-words must be aligned to 2-byte boundary
- Bytes have no alignment requirement

Important considerations:

- Always check alignment requirements
- Be aware of endianness (STM32 is little-endian)
- Consider using multiple register transfer for efficiency
- Manage literal pool placement carefully
- Stack operations must maintain SP word alignment

Pseudo-Instructions

What is a pseudo-instruction?

Explain what is done with a <LDR Rn, =literal> pseudo-instruction.

A pseudo-instruction does not directly translate in machine code. It is an instruction that is interpreted (or expanded) by the assembler (the tool) to generate the needed machine code instruction(s).

When we run the assembler (the tool), the <LDR Rn, =literal> instruction is decomposed into a place reservation of the type DCD and an indirect load of the type LDR <Rt>,[PC,#<imm>] to get the contents of the reserved position and write it in Rn

Literal can be an address (a position in the memory)

Literal can be a constant (the value is known at assembling time)

Register Content and Memory Positions ex 7 from exercise sheet
Whenever possible, work out the contents of registers or memory positions that have changed.

again	LDR	R0, = 0 × FF	R0 = 0x000000FF
	LDR	R1, lit1	R1 = 0x0000EFAA
	LDR	R2, lit2	R2 = 0x00012345
	LDR	R3, = 0 × 55AA	R3 = 0x000055AA
	MOV	R4, R1	R4 = 0x0000EFAA
	MOV	R4, R2	R4 = 0x00012345
	MOV	R6, R3	R6 = 0x000055AA
	MOVS	R7, #04	R7 = 0x00000004
	LDR	R0, =lit1	R0 = address of lit1 (Address is not known)
	LDR	R1, =lit2	R1 = address of lit2
	LDR	R2, =lit3	R2 = address of lit3
	LDRB	R5, [R2]	R5 = 0x00000097
	LDRH	R6, [R2,#2]	R6 = 0x0000008F
	LDR	R2, [R0]	R2 = 0x0000EFAA
	LDR	R3, [R0, #4]	R3 = 0x00012345
	STR	R3, [R1]	lit2 = 0x00012345
	STR	R2, [R1, #8]	lit4 = 0x0000EFAA
	STR	R4, [R1, R7]	lit3 = 0x00012345 (writes to (address of lit2 + R7))
	LDR	R5, =lit5	R5 = address of lit5
	MOVS	R0, #0	R0 = 0x00000000
	ADDS	R7, R0, #1	R7 = 0x00000001
	LDRSB	R6, [R5, R0]	R6 = 0xFFFFFFF89
	LDRSH	R6, [R5, R7]	Problem: Unaligned memory access
	B	Again	
lit1	DCD	0xEFAA	
lit2	DCD	0x12345	
lit3	DCD	0x8F1097	
lit4	DCD	0xFF76552F	
lit5	DCD	0xAA654389	
lit6	DCD	0x0165	
Var1	DCD	0x23	
Var2	DCD	0x24	
Var3	DCD	0 × 23455678	
Var4	DCD	0xE4568900	

Memory Access Patterns Tracking memory and register contents:

1. Track register contents:
 - Note initial values and update after each instruction
 - Pay attention to flags if MOVS/ADDS used
2. Track memory changes: Consider byte ordering! (little-endian)
 - Note memory layout before operations
 - Update after each store instruction

Memory Operation Example

```

1 LDR   R0,=0x20001000 ; R0 = 0x20001000
2 MOVS  R1,#42          ; R1 = 0x0000002A
3 STR   R1,[R0]          ; Mem[0x20001000] = 0x0000002A
4 LDRB  R2,[R0]          ; R2 = 0x0000002A (byte access)

```

PC-Relative Calculations ex. 8 from exercise sheet

Steps for calculating PC-relative offsets:

1. Start with instruction address
2. Add 4 to get effective PC value
3. Calculate target address
 - Must be word-aligned (multiple of 4)
 - Within ±1KB range
4. Compute offset = target - (PC + 4)

PC-Relative Example

```

1 ; At address 0x1000:
2 LDR   R0,[PC,#12]      ; PC = 0x1004, loads from
                           0x1010
3 ...
4 ALIGN 4
5 DCD   0x12345678      ; At 0x1010

```

Array Access ex. 10 from exercise sheet

Steps for array manipulation:

1. Calculate array element offset
 - Byte arrays: index * 1
 - Half-word arrays: index * 2
 - Word arrays: index * 4
2. Access elements
 - LDRB/STRB for bytes
 - LDRH/STRH for half-words
 - LDR/STR for words

Array Example

```

1 LDR   R0,=array        ; Get array base address
2 MOVS  R1,#1            ; Array index
3 LSLS  R2,R1,#2         ; Multiply index by 4 (word
                           array)
4 LDR   R3,[R0,R2]       ; Load array[1]

```

Literal Pool Usage ex. 11 from exercise sheet

Steps for working with literals:

1. Direct load of value
 - LDR Rx,=value for constants
 - Value stored in literal pool
2. Load address of label
 - LDR Rx,label loads value at label
 - LDR Rx,=label loads address of label

Literal Example

```

1 LDR   R0,=0x12345678  ; Load constant from
                           literal pool
2 LDR   R1,myvar         ; Load value at myvar
3 LDR   R2,=myvar        ; Load address of myvar
4
5 myvar  DCD 0x11223344 ; Define variable

```

Handling Pointers

Example C code and assembly implementation:

```

1 uint32_t x;
2 uint32_t y;
3 uint32_t *xp;
4
5 void main(void) {
6     x = 3;
7     xp = &x;
8     y = *xp;
9 }

```

Assembly implementation:

```

1 AREA  myVar, DATA, READWRITE
2 X     DCD 0
3 Y     DCD 0
4 XP    DCD 0
5
6 AREA  myCode, CODE, READONLY
7 main
8 ; Get addresses
9 LDR   R3, =X           ; Address of X
10 LDR   R4, =Y           ; Address of Y
11 LDR   R5, =XP          ; Address of XP
12
13 ; x = 3
14 MOVS R0, #3           ; Load immediate value
15 STR  R0, [R3]          ; Store in X
16
17 ; xp = &x
18 STR  R3, [R5]          ; Store address of X in XP
19
20 ; y = *xp
21 LDR  R2, [R5]          ; Load address from XP
22 LDR  R1, [R2]          ; Load value from address
23 STR  R1, [R4]          ; Store in Y

```

Arithmetic Operations

Basic Arithmetic Instructions

Core arithmetic operations:

- 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 ($A - B - !c$)
- RSBS:** Reverse Subtract ($-1 \cdot A$)
- MULS:** Multiplication ($A \cdot B$)

Addition Operations

Addition instructions and their uses:

- ADDS Rd, Rn, Rm \rightarrow Rd = Rn + Rm**
 - Updates flags, only low registers
- ADD Rd, Rm \rightarrow Rd = Rd + Rm**
 - No flag updates, can use high registers
- ADDS Rd, #imm \rightarrow Rd = Rd + immediate**
 - 8-bit immediate value only

```
1 ; Different ADD variants
2 ADDS R1, R2, R3      ; R1 = R2 + R3, update flags
3 ADD  R8, R9           ; R8 = R8 + R9, no flags
4 ADDS R1, #255         ; R1 = R1 + 255, update flags
```

Subtraction Operations

Subtraction instructions and their uses:

- SUBS Rd, Rn, Rm \rightarrow Rd = Rn - Rm**
 - Updates flags, only low registers
- SUBS Rd, #imm \rightarrow Rd = Rd - immediate**
 - 8-bit immediate value
- RSBS Rd, Rn, #0 \rightarrow Rd = -Rn** (2's complement)
 - Special case for negation

```
1 ; Different SUB variants
2 SUBS R1, R2, R3      ; R1 = R2 - R3
3 SUBS R1, #100        ; R1 = R1 - 100
4 RSBS R1, R2, #0       ; R1 = -R2
```

Multiplication

```
1 ; Basic multiplication
2 MULS R0, R1, R0      ; R0 = R1 * R0
3 ; Multiply by constant using shifts
4 LSLS R0, R0, #2       ; R0 = R0 * 4
5 ; Multiply by 10 (8 + 2)
6 LSLS R1, R0, #3       ; R1 = R0 * 8
7 LSLS R2, R0, #1       ; R2 = R0 * 2
8 ADDS R0, R1, R2       ; R0 = R0 * 10
```

Arithmetic Operations

Steps for arithmetic operations:

- Determine if operation is signed or unsigned
- Choose appropriate instruction (with or without 'S')
 - Use ADDS/SUBS for low registers with flags
 - Use ADD/SUB for high registers
 - For immediate values > 8-bit, load to register first
- Consider potential carry/overflow conditions
- For multi-word operations: see KR **Multi-Word Operations**
- Check relevant flags after operation: see **Processor Status Flags**

Signed vs. Unsigned Arithmetic

Two's Complement

For negative numbers:

- Two's complement: $A = !A + 1$ (Invert all bits and add 1 to result)
- Used for representing signed numbers
- Enables using same hardware for addition and subtraction

Carry and Overflow

Unsigned Operations:

- Addition: C = 1 indicates carry (result too large for available bits)
- Subtraction: C = 0 indicates borrow (result negative)

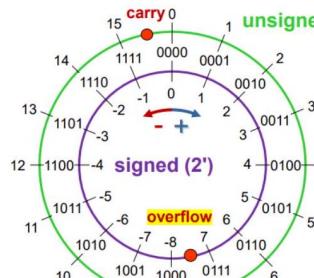
Signed Operations:

- Addition: V = 1 if overflow with operands of same sign
- Subtraction: V = 1 if overflow with operands of opposite signs

Number Circles and Two's Complement

Understanding arithmetic wrap-around:

- Fixed number of bits creates circular number space
- Addition moves clockwise on number circle
- Subtraction moves counter-clockwise



Addition: C = 1 \rightarrow Carry

1	1	0	1	13d
0	1	1	1	7d
1	1	1	1	

1 0 1 0 0 20d \rightarrow 16d + 4d

Subtraction: C = 0 \rightarrow Borrow

6d	-	14d	= 0110b	- 1110b = 0110b + 0010b
0	1	1	0	6d
0	1	0	0	2d = TC(14d)
0	1	1	0	

0 1 0 0 0 8d \rightarrow - 16d + 8d

Integer Ranges by Word Size

8-bit integers:

- Unsigned: 0 to 255 (0x00 to 0xFF)
- Signed: -128 to 127 (0x80 to 0x7F)

16-bit integers:

- Unsigned: 0 to 65,535 (0x0000 to 0xFFFF)
- Signed: -32,768 to 32,767 (0x8000 to 0x7FFF)

32-bit integers:

- Unsigned: 0 to 4,294,967,295 (0x00000000 to 0xFFFFFFFF)
- Signed: -2,147,483,648 to 2,147,483,647 (0x80000000 to 0x7FFFFFFF)

Memory Operations for Arithmetic Operation Patterns

```
1 ; Pattern for modifying memory value
2 LDR  Rx, =variable    ; Get address
3 LDR  Ry, [Rx]          ; Load value
4 ; Perform operation
5 STR  Ry, [Rx]          ; Store result
6 ; For 16-bit values
7 LDRH/STRH              ; Use half-word instructions
8 ; For 8-bit values
9 LDRB/STRB              ; Use byte instructions
```

Flag Usage and Overflow Detection

Processor Status Flags

APSR (Application Program Status Register) contains important flags affected by arithmetic operations:

- N (Negative):** Set when result's MSB = 1, used for signed operations
- Z (Zero):** Set when result = 0, used for both signed/unsigned
- C (Carry):** Set when unsigned overflow occurs
- V (Overflow):** Set when signed overflow occurs

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

Instructions ending with 'S' modify these flags:

- ADDS, SUBS, MOVS, LSLS

Overflow Detection

Steps to detect overflow in arithmetic operations:

- For unsigned arithmetic (using C flag):
 - Addition: Check C flag (C=1 means overflow)
 - Subtraction: Check C flag (C=0 means underflow)
- For signed arithmetic (using V flag):
 - Addition: Check V flag for same-sign operands
 - Subtraction: Check V flag for opposite-sign operands

Example:

```
1 ; Unsigned overflow detection
2 ADDS R0, R1           ; Perform addition
3 BCS   overflow        ; Branch if carry set
4
5 ; Signed overflow detection
6 ADDS R0, R1           ; Perform addition
7 BVS   overflow        ; Branch if overflow set
```

Flag Usage Examples of flag behavior:

```
1 ; Zero flag example
2 MOVS R0, #5            ; Perform addition
3 SUBS R0, #5            ; Z=1 (result is zero)
4
5 ; Negative flag example
6 MOVS R0, #1            ; Perform addition
7 SUBS R0, #2            ; N=1 (result is negative)
8
9 ; Carry flag example
10 MOVS R0, #0xFF
11 ADDS R0, #1            ; C=1 (unsigned overflow)
12
13 ; Overflow flag example
14 MOVS R0, #0x7F
15 ADDS R0, #1            ; V=1 (signed overflow)
```

Examples for basic arithmetic operations

Basic Arithmetic Operations

Given variables in data section:

```

1 AREA    progData, DATA, READWRITE
2 Zahl1   DCD    0          ; 32 bit
3 Zahl2   DCD    0          ; 32 bit
4 Zahl3   DCW    0          ; 16 bit
5 Zahl4   DCW    0          ; 16 bit
6 Zahl5   DCB    0          ; 8 bit
7 Zahl6   DCB    0          ; 8 bit

```

Basic register operations:

```

1 ; R0 = R1 + R2
2 ADDS   R0, R1, R2
3
4 ; R0 = R1 + R2 + R3
5 ADDS   R0, R1, R2
6 ADDS   R0, R0, R3
7
8 ; R8 = R8 + 1
9 MOVS   R0, #1
10 ADD    R8, R8, R0
11
12 ; R8 = -R8
13 MOV    R7, R8
14 RSBS   R7, #0
15 MOV    R8, R7

```

```

1 ; R1 = R2 + 200
2 MOVS   R1, #200
3 ADD    R1, R1, R2
4 ; Not ADDS R1,R2,#200
5 ; invalid, rd = rn
6
7 ; R10 = R8 - R7
8 MOV    R0, R8
9 SUBS   R0, R0, R7
10 MOV   R10, R0

```

Operations involving memory variables:

```

1 ; Zahl1 = Zahl1 - 1
2 LDR   R1, =Zahl1      ; Get address
3 LDR   R0, [R1]         ; Load value
4 SUBS  R0, #1          ; Decrement
5 STR   R0, [R1]         ; Store back
6
7 ; Zahl1 = Zahl1 + Zahl2
8 LDR   R0, =Zahl1
9 LDR   R1, [R0]         ; Load Zahl1
10 LDR  R2, =Zahl2
11 LDR  R3, [R2]         ; Load Zahl2
12 ADDS R1, R1, R3      ; Add
13 STR  R1, [R0]         ; Store result
14
15 ; Zahl3 = Zahl3 - Zahl4 (unsigned)
16 LDR  R0, =Zahl3
17 LDRH R1, [R0]         ; Load 16-bit
18 LDR  R2, =Zahl4
19 LDRH R3, [R2]
20 SUBS R1, R1, R3
21 STRH R1, [R0]         ; Store 16-bit
22
23 ; Zahl1 = R5 * Zahl3 (unsigned)
24 LDR  R0, =Zahl1
25 LDR  R1, =Zahl3
26 LDRH R2, [R1]
27 MULS R2, R5, R2
28 STR  R2, [R0]

```

Multi-Word Arithmetic

Multi-Word Arithmetic

Guidelines for operations on large numbers:

1. Addition sequence:

```

1 ; 64-bit addition (R1:R0 + R3:R2)
2 ADDS  R0, R2           ; Add low words
3 ADCS  R1, R3           ; Add high words with carry

```

2. Subtraction sequence:

```

1 ; 64-bit subtraction (R1:R0 - R3:R2)
2 SUBS  R0, R2           ; Subtract low words
3 SBCS  R1, R3           ; Subtract high words with borrow

```

3. Important considerations:

- Start with least significant words
- Use carry-aware instructions for higher words
- Ensure proper register allocation
- Track flags through entire operation

Multi-Word Operations Multi-Register Operations

Guidelines for multi-word arithmetic:

1. Loading values:

- Use LDM to load multiple words efficiently
- Consider register allocation carefully for all parts
- Keep track of which registers hold which parts
- Use temporary registers for high register operations

2. Performing operations:

- Start with least significant words
- Use ADDS/SUBS for first word
- Use ADCS/SBCS for subsequent words
- Pay attention to carry/borrow propagation
- Chain operations for complex expressions

3. Storing results:

- Use STM for efficient multi-word storage
- Maintain proper alignment
- Consider word order (endianness)

Example 96-bit addition:

```

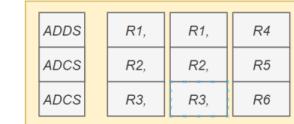
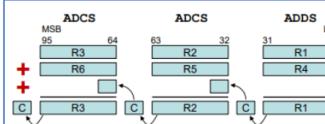
1 ; Load values
2 LDR   R0, =value1
3 LDM   R0!, {R1-R3}      ; First number
4 LDR   R0, =value2
5 LDM   R0!, {R4-R6}      ; Second number
6
7 ; Add with carry
8 ADDS  R1, R1, R4        ; Low word
9 ADCS  R2, R2, R5        ; Middle word
10 ADCS R3, R3, R6        ; High word
11
12 ; Store result
13 LDR   R0, =result
14 STM   R0!, {R1-R3}

```

Important considerations:

- Always handle carry/borrow for multi-word operations
- Consider register restrictions (high vs low)
- Pay attention to word size when loading/storing
- Use appropriate instructions for signed vs unsigned operations

Multi-Word Addition Adding 96-bit values using ADCS:

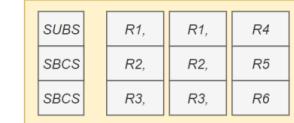
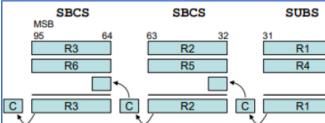


```

1 ADDS R1, R1, R4           ; Add least significant words
2 ADCS R2, R2, R5           ; Add middle words with carry
3 ADCS R3, R3, R6           ; Add most significant words with carry

```

Multi-Word Subtraction Subtracting 96-bit values using SBCS:



```

1 SUBS R1, R1, R4           ; Subtract least significant words
2 SBCS R2, R2, R5           ; Subtract middle words with borrow
3 SBCS R3, R3, R6           ; Subtract most significant words with borrow

```

64-bit Operations

Given 64-bit variables:

```

1 AREA    progData, DATA, READWRITE
2 Long1   DCD    0          ; low word
3 DCD    0          ; high word
4 Long2   DCD    0          ; low word
5 DCD    0          ; high word
6 Long3   DCD    0          ; low word
7 DCD    0          ; high word

```

64-bit addition:

```

1 ; Long3 = Long1 + Long2
2 LDR   R0, =Long1
3 LDM   R0!, {R1-R4}      ; Load all words
4 ADDS R1, R1, R3        ; Add low words
5 ADCS R2, R2, R4        ; Add high words with carry
6 LDR   R0, =Long3
7 STM   R0!, {R1, R2}     ; Store result

```

64-bit subtraction:

```

1 ; Long3 = Long2 - Long1
2 LDR   R0, =Long1
3 LDM   R0!, {R1-R4}      ; Load all words
4 SUBS R3, R3, R1        ; Subtract low words
5 SBCS R4, R4, R2        ; Subtract high words with borrow
6 LDR   R0, =Long3
7 STM   R0!, {R3, R4}     ; Store result

```

Logic, Shift and Rotate Instructions

Logic Instructions

Base logic operations (affect only N and Z flags):

- ANDS:** Bitwise AND (Rdn & Rm, a & b)
- BICS:** Bit Clear (Rdn & !Rm, a & b)
- EORS:** Exclusive OR (Rdn \$Rm, a ^ b)
- MVNS:** Bitwise NOT (!Rm, a)
- ORRS:** Bitwise OR (Rdn # Rm, a | b)

Logical Operations Common logic operations:

```

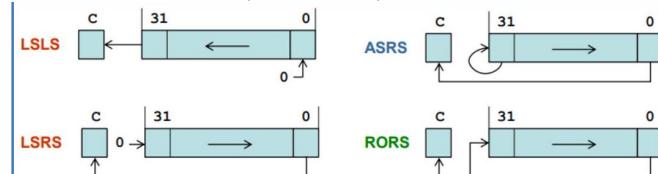
1 ; Logic operations
2 ANDS R0, R1           ; RO = RO AND R1
3 BICS R0, R1           ; RO = RO AND NOT R1
4 EORS R0, R1           ; RO = RO XOR R1
5 MVNS R0, R1           ; RO = NOT R1
6 ORRS R0, R1           ; RO = RO OR R1
7
8 ; Shift operations
9 LSLS R0, R1, #2        ; RO = R1 << 2 (multiply by 4)
10 LSRS R0, R1, #1       ; RO = R1 >> 1 (divide by 2)
11 ASRS R0, R1, #2       ; RO = R1 >> 2 (signed divide by 4)
12 RORS R0, R1, #1       ; Rotate R1 right by 1 bit

```

Shift and Rotate Instructions

Shift operations for binary manipulation:

- LSLS:** Logical Shift Left ($2^n \cdot Rn$, 0 → LSB)
- LSRS:** Logical Shift Right ($2^{-n} \cdot Rn$, 0 → MSB)
- ASRS:** Arithmetic Shift Right (R^{-n} , ±MSB → MSB)
- RORS:** Rotate Right (LSB → MSB)



Shift Operations for Arithmetic Using shifts for multiplication and division:

```

1 ; Multiplication by powers of 2
2 LSLS  R0, R0, #1      ; RO = RO * 2
3 LSLS  R0, R0, #2      ; RO = RO * 4
4 LSLS  R0, R0, #3      ; RO = RO * 8
5
6 ; Division by powers of 2
7 LSRS  R0, R0, #1      ; RO = RO / 2 (unsigned)
8 ASRS  R0, R0, #1      ; RO = RO / 2 (signed)
9
10 ; Multiply by 10 (8 + 2)
11 LSLS  R1, R0, #3      ; R1 = RO * 8
12 ADDS  R0, R0, R1      ; RO = RO + (RO * 8) = RO * 9
13 ADDS  R0, R0, R0      ; RO = RO * 2 = RO * 10

```

Using Logic and Shift Instructions

- Identify required operation (AND, OR, XOR, NOT, shift)
- Choose appropriate instruction
- Consider effect on flags if relevant

For shifts:

- LSLS for multiplication by 2^n
- LSRS for unsigned division by 2^n
- ASRS for signed division by 2^n

For logic:

- ANDS for bit masking
- ORRS for bit setting
- BICS for bit clearing
- EORS for bit toggling

Flag Behavior with Logic Instructions

Logic instructions only affect N and Z flags:

- N flag:** Set to bit 31 of result (MSB)
 - Z flag:** Set if result is zero
 - C, V flags:** Unchanged
- Special case for shift/rotate:
- C flag:** Set to last bit shifted out
 - N,Z flags:** Set based on result
 - V flag:** Unchanged

Bit Manipulation Techniques

Common patterns for bit manipulation:

- Setting specific bits:

```

1 MOVS  R0, #pattern    ; Create bit pattern
2 ORRS  target, R0       ; OR to set bits

```

- Clearing specific bits:

```

1 MOVS  R0, #pattern    ; Create bit pattern
2 BICS  target, R0       ; Clear selected bits

```

- Inverting specific bits:

```

1 MOVS  R0, #pattern    ; Create bit pattern
2 EORS  target, R0       ; XOR to invert bits

```

- Testing bits:

```

1 MOVS  R0, #pattern    ; Create bit pattern
2 ANDS  R1, target, R0   ; AND to test bits
3 ; Check flags for result

```

Bit Manipulation

```

1 ; Set bits 0 and 4
2 MOVS  R1, #0x11        ; Mask: 0001 0001
3 ORRS  R0, R1           ; Set bits in R0
4
5 ; Clear bits 1 and 5
6 MOVS  R1, #0x22        ; Mask: 0010 0010
7 BICS  R0, R1           ; Clear bits in R0
8
9 ; Toggle bits 2,3,4
10 MOVS  R1, #0x1C       ; Mask: 0001 1100
11 EORS  R0, R1           ; Toggle bits in R0
12
13 ; Test bit 3
14 MOVS  R1, #0x08        ; Mask: 0000 1000
15 ANDS  R2, R0, R1       ; Test bit
16 BEQ   bit_is_clear     ; Branch if bit was 0

```

Casting, Sign Extension and Type Conversion

Integer Casting

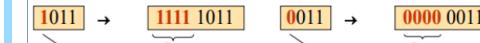
Extension (adding bits):

- Zero Extension** (unsigned):
 - Fill left bits with zero
 - Example: 1011 → 00001011
- Sign Extension** (signed):
 - Copy sign bit to the left
 - Example: 1011 → 11111011

Unsigned → Zero Extension



Signed → Sign Extension



Truncation: Cast cuts out the left most bits

- Signed: May change sign
- Unsigned: Results in modulo operation

Integer Ranges based on word size

8-bit	hex	unsigned	signed	16-bit	hex	unsigned	signed
0x00		0	0	0x0000		0	0
...
0x7F	127	127	127	0x7FFF	32'767	32'767	32'767
0x80	128	-128	-128	0x8000	32'768	-32'768	-32'768
...
0xFF	255	-1	-1	0xFFFF	65'535	-1	-1

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

Sign Extension Instructions

Instructions for extending smaller values:

- SXTB:** Sign extend byte to word
- UXTB:** Zero extend byte to word

- Takes lowest byte
- Copies bit 7 to bits 31-8

SXTH:

Sign extend half-word to word

- Takes lowest half-word
- Copies bit 15 to bits 31-16

- UXTH:** Zero extend half-word to word
- Takes lowest half-word
- Sets bits 31-16 to zero

Sign Examples

```

1 ; Sign extension examples
2 SXTB  R0, R1           ; Sign extend byte
3 SXTH  R0, R1           ; Sign extend half-word
4
5 ; Zero extension examples
6 UXTB  R0, R1           ; Zero extend byte
7 UXTH  R0, R1           ; Zero extend half-word
8
9 ; Manual sign extension
10 LSLS  R0, R0, #24      ; Shift left 24 bits
11 ASRS  R0, R0, #24      ; Arithmetic shift right 24

```

Type Conversion Guidelines Steps for safe type conversion:

1. For unsigned to larger unsigned:

- Use zero extension (UXTB, UXTH)
- Or use LSLS followed by LSRS

```

1 ; Extend 8-bit to 32-bit unsigned
2 MOVS R0, #0xFF      ; Load 8-bit value
3 UXTB R2, R0          ; Unsigned extension
4
5 ; Manual zero extension
6 LDRB R0, [R1]        ; Load byte, top bits zero
7 LSLS R0, #24         ; Move to top byte
8 LSRS R0, #24         ; Logical shift back

```

2. For signed to larger signed:

- Use sign extension (SXTB, SXTH)
- Or use LSLS followed by ASRS

```

1 ; Extend 8-bit to 32-bit signed
2 MOVS R0, #0xFF      ; Load 8-bit value
3 SXTB R1, R0          ; Signed extension
4
5 ; Manual sign extension
6 LDRSB R0, [R1]       ; Load with sign extend
7 LDRB R0, [R1]         ; Load byte
8 LSLS R0, #24         ; Move to top byte
9 ASRS R0, #24         ; Arithmetic shift back

```

3. Reducing size (truncation):

- Use AND with appropriate mask
- Or store using STRB/STRH
- Check for potential data loss

Example:

```

1 ; Truncate 32-bit to 8-bit
2 MOVS R1, #0xFF      ; Create mask
3 ANDS R0, R1          ; Truncate to 8 bits
4
5 ; Store 32-bit value as 8-bit
6 STRB R0, [R1]        ; Store byte

```

Important considerations:

- Always consider signedness of values
- Check for potential carry/overflow in arithmetic shifts
- Remember carry flag behavior in shifts
- Use appropriate extension for data type
- Consider performance impact of shifts vs multiply
- Be careful with bit patterns crossing byte boundaries
- Document complex bit manipulations clearly

Examples

Logical Instructions

Bit manipulation examples:

1. Inverting register contents (one's complement):

```
1 MVNS R1, R1          ; Invert all bits in R1
```

2. Selective bit manipulation:

```

1 ; Set bits 3..0 to 1, bits 7..4 to 0
2 ; Invert bits 17-16, keep others unchanged
3 MOVS R0, #0xF          ; Pattern for bits 3..0
4 ORRS R1, R0            ; Set bits 3..0 to 1
5 MOVS R0, #0xFO         ; Pattern for bits 7..4
6 BICS R1, R0            ; Clear bits 7..4
7 MOVS R0, #0x30         ; 
8 LSLS R0, #12            ; 0x30000 shift 4 nibbles = 12bit
9 EORS R1, R0            ; Invert bits 17-16

```

Multiplication Using Shifts

Multiply R7 by 43 using two different methods:

1. Using multiplication instruction:

```

1 MOVS R0, #43          ; Load constant
2 MULS R7, R0, R7        ; Multiply

```

2. Using shifts and additions:

```

1 MOVS R7, R0            ; Copy original value
2 LSLS R0, R0, #1         ; *2
3 ADDS R7, R7, R0         ; 
4 LSLS R0, R0, #2         ; *8=4*2
5 ADDS R7, R7, R0         ; 
6 LSLS R0, R0, #2         ; *32=8*4
7 ADDS R7, R7, R0         ; Total = 1 + 2 + 8 + 32 = 43

```

Reverse Engineering

Convert assembly to C code:

Given C variables:

```
1 uint32_t ux, uy, uz; // Variables in memory
```

Example 1:

```

1 ; Assembly code
2 LDR R0, =ux            ; Load address
3 LDR R1, [R0]             ; Load value
4 LSLS R1, R1, #1          ; Shift left by 1
5 LDR R2, =uy            ; Load uy address
6 LDR R3, [R2]             ; Load uy value
7 ADDS R3, R1              ; Add shifted ux
8 LSLS R3, R3, #3          ; Shift left by 3
9 STR R3, [R2]             ; Store back in uy

```

Equivalent C code:

```

1 uy = ((ux << 1) + uy) << 3;
2 // or:
3 uy = 8 * (2 * ux + uy);

```

Reverse Engineering 2

Convert assembly to C code:

Die Speicherstellen im Assembler werden den ursprünglichen Variablen im C-Programm wie folgt zugeordnet:

```

1 ux: DCD ? ; uint32_t ux
2 uy: DCD ? ; uint32_t uy
3 uz: DCD ? ; uint32_t uz

```

Example 2:

```

1 ; Assembly code
2 LDR R0, =ux            ; Load address
3 LDR R1, [R0]             ; Load value
4 LDR R2, =uy            ; Load address
5 LDR R3, [R2]             ; Load value
6 LDR R4, =uz            ; Load address
7 LDR R5, [R4]             ; Load value
8 LSRS R1, R1, #3          ; Shift right by 3
9 LSLS R3, R3, #4          ; Shift left by 4
10 ORRS R1, R3             ; OR to combine
11 MVNS R1, R1             ; Invert bits
12 ANDS R1, R5             ; AND with uz
13 STR R1, [R4]             ; Store back in uz

```

Equivalent C code:

```

1 uz = ~(ux >> 3) | (uy << 4) & uz;
2 // or:
3 uz = ~(ux / 8) | (uy * 16) & uz;

```

Type Casting Examples of explicit casting in C:

1. Unsigned to signed casting:

```

1 uint8_t ux = 100;           // 0x64 or 0110 0100
2 int8_t sx = (int8_t)ux;     // Same bit pattern
3                                         // Interpreted as 100

```

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

100d -> ux hat Speicherinhalt 0x64 oder Binär 0110'0100

sx hat denselben Speicherinhalt, wird aber als signed interpretiert. Da das höchstwertigste Bit nicht gesetzt ist, hat die Interpretation aber in diesem Fall keinen Einfluss.

sx wird ebenfalls als 100d interpretiert

2. Signed to unsigned casting:

```

1 int8_t sx = -10;           // 0xF6 or 1111 0110
2 uint8_t ux = (uint8_t)sx;    // Same bits but unsigned
3                                         // 246 (128+64+32+16+4+2)

```

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

-10d -> sx hat Speicherinhalt 0xF6 oder Binär 1111'0110 (-128+64+32+16+4+2)

ux hat denselben Speicherinhalt, wird aber als unsigned interpretiert. So ergibt sich 128 + 64 + 32 + 16 + 4 + 2 = 246d

Branch Instructions

Overview Branch Instructions

Branch instructions control program flow:

Type:

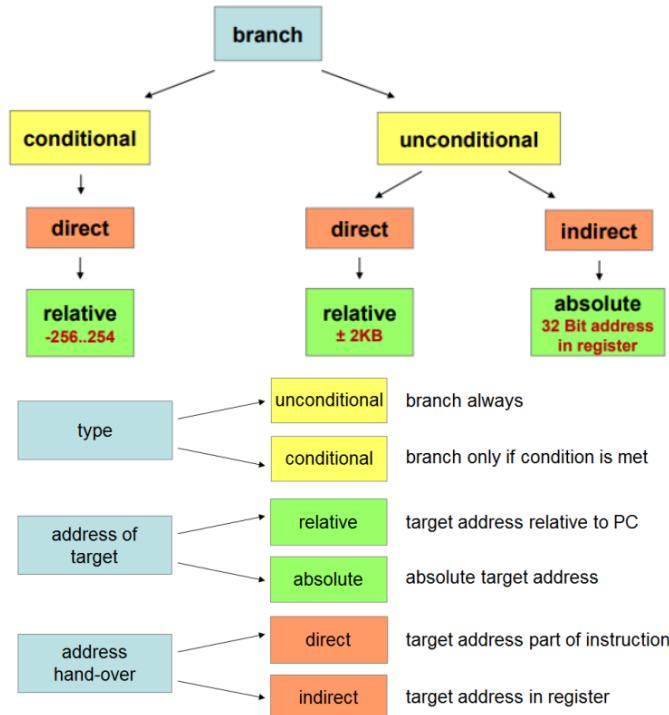
- **Unconditional:** Always taken
- **Conditional:** Branch if condition met

Address hand-over:

- **Direct:** Target addresses part of instruction
- **Indirect:** Target address in register

Address of Target:

- **Relative:** Target address relative to PC
- **Absolute:** Complete (absolute) target address



Types of Branches

Unconditional Branches:

- B (immediate) → B label
 - **Direct**
 - **Relative**
- BX (Branch and Exchange) → BX R0
 - **Indirect**
 - **Absolute**
- BL (Branch with Link) → BL label
 - **Indirect**
 - **Absolute**

Conditional Branches:

Flag-dependent: BEQ, BNE, BCS, BCC, etc.

Arithmetic: BHI, BLS, BGE, BLT, etc.

- **Indirect**
- **Absolute**

Flag dependent instructions

Unsigned: Higher and Lower

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

Signed: Greater and Less

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

Compare and Test

- TST: AND without changing the value
- CMP: SUB without changing the value

Comparison Instructions Using CMP and TST instructions:

```

1 ; Signed comparisons
2 LDR R1, =0xFFFFFFFF
3 CMP R1, #1           ; Compare with immediate
4 BLT is_less          ; Branch if less than
5
6 ; Unsigned comparisons
7 CMP R1, #1
8 BLO is_below         ; Branch if lower
9
10 ; Bit testing
11 LDR R5, =0x0040FFFF
12 TST R3, R5           ; Test bits
13 BNE bits_set         ; Branch if any bit set
14 BEQ bits_clear       ; Branch if all bits clear
15
16 ; Test overflow
17 SUBS R2, R2, R5
18 BVS overflow
19 ADDS R0, R0, #0x04   ; Mark if not taken
  
```

Branch Pattern Recognition

Common branch patterns and their uses:

1. Basic branching:

```

1 ; Simple jump
2 B target          ; Unconditional
3 BEQ target        ; Branch if equal
4 BNE target        ; Branch if not equal
5
6 ; Conditional execution
7 CMP R0, #value
8 BCC target        ; Branch if carry clear
9 BGT target        ; Branch if greater than
  
```

2. Jump tables:

```

1 ; Load table base
2 LDR R0, =jumptable
3 ; Calculate offset
4 LSLS R1, R1, #2      ; Multiply index by 4
5 ADDS R0, R0, R1      ; Add to base
6 ; Jump to handler
7 LDR R0, [R0]          ; Load address
8 BX R0                ; Branch to handler
  
```

3. Infinite loops:

```

1 endless B endless ; Simple infinite loop
2
3 loop ; Do something
4 B loop            ; Loop forever
  
```

Conditional Branches with Flag Testing

Example tracking non-taken branches:

```

1 MOVS R0, #0          ; Clear mask
2
3 ; Test carry flag
4 ADDS R1, R1, #5
5 BCS taken
6 ADDS R0, R0, #0x01  ; Mark if not taken
7 taken
8
9 ; Test equality
10 ANDS R1, R1, R5
11 BEQ equal
12 ADDS R0, R0, #0x02  ; Mark if not taken
13 equal
14
15 ; Test overflow
16 SUBS R2, R2, R5
17 BVS overflow
18 ADDS R0, R0, #0x04  ; Mark if not taken
19 overflow
  
```

Result in R0 shows which branches were not taken.

Examples

Unconditional Branch Flow The execution starts at line 10.

Example of branching sequence and jump table:

1. List the sequence of branch instructions that end in an infinite loop.

Do this by stating the branches in tabular form: from - to.

E.g. the first branch is 11 – 16 (branch unconditionally from line 11 to line 16).

2. At which line does the execution sequence finally loop forever?

```

1 Label1 LDR R0, =Label5
2 Label2 BX R0
3 Jumptable
4     DCD Case0
5     DCD Case1
6     DCD Case2
7     DCD Case3
8 Label5 LDR R0, =Label6
9     B Label2
10 Label6 LDR R2, =Jumptable
11     ADDS R2, R2, #4
12 Label4 LDR R2, [R2]
13     BX R2
14 Case0 B Case0           ; Infinite loop
15 Case1 LDR R2, =Jumptable
16     MOVIS R1, #3
17     LSIS R1, R1, #2
18     ADDS R2, R2, R1
19     B Label14
20 Case2 B Label1
21 Case3 B Case0

```

Solution:

1. Branch sequence:

10 – 16, 17 – 11, 11 – 18, 21 – 23, 27 – 20, 21 – 29, 29 – 22, 22 – 22...

2. Loop at line 22

Conditional Branch Flow The execution starts at line 10.

1. List which branch instructions jump to the given label.

Do this by stating the branches in tabular form: from - to.

2. What is the final value in R0 as hexadecimal value?

```

10    LDR R1, =0xFFFFFFFF-5
11    LDR R2, -10
12    LDR R3, =0x2341

13
14    MOVS R0, #0          ; mask of branches that are not taken
15
16 Label11 ADDS R1, R1, #5
17     BCS Label12
18     ADDS R0, R0, #0x01      ; set flag if no branch
19 Label12 ADDS R1, R1, #1
20     BCS Label13
21     ADDS R0, R0, #0x02      ; set flag if no branch
22 Label13 ADDS R5, =0x0F18C
23     ANDS R1, R1, R5
24     BEQ Label21
25     ADDS R0, R0, #0x04      ; set flag if no branch
26
27 Label21 LDR R5, =2000
28     SUBS R2, R2, R5
29     BVS Label22
30     ADDS R0, R0, #0x08      ; set flag if no branch
31 Label22 LDR R5, =0x7FFFFFFF
32     SUBS R2, R2, R5
33     BVS Label23
34     ADDS R0, R0, #0x10      ; set flag if no branch
35 Label23 ADDS R2, R2, #1
36     BMI Label31
37     ADDS R0, R0, #0x20      ; set flag if no branch
38
39 Label31 LSRS R3, R3, #1
40     BCS Endless
41     ADDS R0, R0, #0x40      ; set flag if no branch
42
43 Endless B Endless

```

Solution:

1. 20 – 22, 24 – 27, 33 – 35, 40 – 43

2. 0×29 (binary 0010 '1001)

Comparison Instructions

The execution starts at line 10.

1. List which branch instructions jump to the given label.

Do this by stating the branches in tabular form: from - to.

2. What is the final value in R0 as hexadecimal value?

```

10    LDR R1, =0xFFFFFFFF
11    LDR R2, =0x80000000
12    LDR R3, =0x9CFA0000
13    LDR R4, =0xC2350000

14
15    MOVS R0, #0          ; mask of branches that are not taken
16
17 Label11 CMP R1, #1
18     BLT Label12
19     ADDS R0, R0, #0x01      ; set flag if no branch
20 Label12 CMP R1, #1
21     BLO Label21
22     ADDS R0, R0, #0x02      ; set flag if no branch
23
24 Label21 LDR R5, =0xFFFFFFFF
25     CMP R2, R5
26     BGT Label22
27     ADDS R0, R0, #0x04      ; set flag if no branch
28 Label22 LDR R5, =0x7FFFFFFF
29     CMP R2, R5
30     BHI Label31
31     ADDS R0, R0, #0x08      ; set flag if no branch
32
33 Label31 LDR R5, =0x0040FFFF
34     TST R3, R5
35     BNE Label32
36     ADDS R0, R0, #0x10      ; set flag if no branch
37 Label32 TST R3, R5
38     BEQ Label41
39     ADDS R0, R0, #0x20      ; set flag if no branch
40
41 Label41 LDR R5, =0x02100000
42     TST R4, R5
43     BEQ Label42
44     ADDS R0, R0, #0x40      ; set flag if no branch
45 Label42 LDR R5, =0x10080000
46     TST R4, R5
47     BEQ Endless
48     ADDS R0, R0, #0x80      ; set flag if no branch
49
50 Endless B Endless

```

Solution:

1. 18 – 20, 30 – 33, 35 – 37, 47 – 50

2. 0×66 (binary 0110 '0110)

Control Structures

Control Structure Types

Three fundamental types of control structures:

1. Sequence:

- Linear execution of instructions
- No branching or decisions
- Operations performed in order

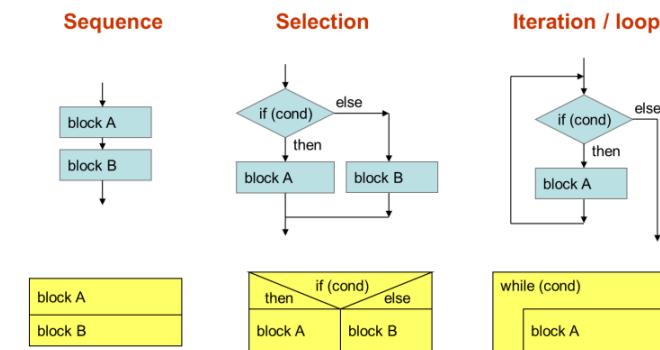
2. Selection:

- Conditional execution (if-then-else)
- Branch based on condition
- Different paths based on test

3. Iteration:

- Repeated execution (loops)
- Condition-controlled repetition
- Fixed number of repetitions

Program flow can be represented with three elements:



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

Implementing Control Structures

Steps for implementing control structures:

1. Choose appropriate control structure:

- If-then-else for simple decisions
- Switch for multiple cases with same variable
- Loops for repeated operations

2. For switches:

- Create jump table
- Calculate offset based on case value
- Handle default case

3. For loops:

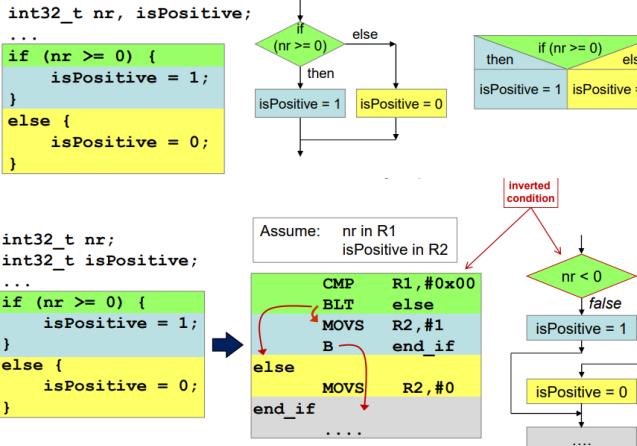
- Initialize counter/condition
- Place condition check appropriately
- Ensure proper exit condition
- Update variables correctly

Important considerations:

- Consider branch range limitations
- Be aware of condition flag changes
- Handle corner cases in comparisons
- Plan for proper loop termination
- Document complex branching logic

Selection Structures

IF-ELSE Compiler translates **selection** into assembly code using conditional and unconditional jumps:



Selection Implementation

1. Simple if-then:

```

1 ; if (x > 0) { x++; }
2   CMP R0, #0           ; Compare x with 0
3     BLE endif          ; Skip if x <= 0
4     ADDS R0, #1          ; x++
5   endif
  
```

2. if-then-else:

```

1 if (condition) { //then-part
2 } else {          //else-part
3 }
  
```

Assembly code:

```

1   CMP R0, #value      ; Test condition
2   BNE else_part       ; Branch if false
3   then_part
4     ; Then code
5     B endif            ; Skip else
6   else_part
7     ; Else code
8   endif
9     ; Continue execution
  
```

3. Nested if:

```

1 if (x > 0) {
2   if (y > 0) {
3     x = y;
4   }
5 }
  
```

Assembly code:

```

1   CMP R0, #0           ; Check x > 0
2   BLE endif_outer      ; If false, skip inner
3   CMP R1, #0           ; Check y > 0
4   BLE endif_inner      ; If false, skip code
5   MOVS R0, R1           ; x = y
6   endif_inner
7   endif_outer
  
```

Selection Implementation with multiple Conditions

1. Multiple conditions (AND):

```

1 if (condA && condB) { //then-part
2 } else {               //else-part
3 }

1 ; Test first condition
2 CMP Rx, #valA
3 B<cc> else_label      ; Branch if first fails
4 ; Test second condition
5 CMP Ry, #valB
6 B<cc> else_label      ; Branch if second fails
7
8 ; Then part
9 <then instructions>
10 B endif_label
11
12 else_label
13 ; Else part
14 <else instructions>
15
16 endif_label
  
```

2. Multiple conditions (OR):

```

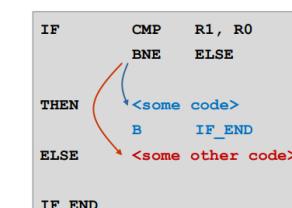
1 if (condA || condB) { //then-part
2 } else {               //else-part
3 }

1 ; Test first condition
2 CMP Rx, #valA
3 B<cc> test_second      ; Try second if first
4   fails
5 B then_label           ; First succeeded
6
7 test_second
8   CMP Ry, #valB
9   B<cc> else_label      ; Branch if both fail
10
11 then_label
12   ; Then part
13   <then instructions>
14   B endif_label
15
16 else_label
17   ; Else part
18   <else instructions>
19
endif_label
  
```

Limitations of Conditional Branches

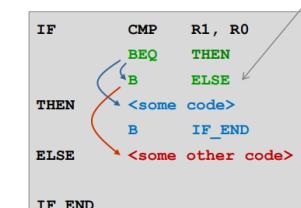
Limited range of -256..254 Bytes

- Example



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

Unconditional branch has longer range than conditional branch

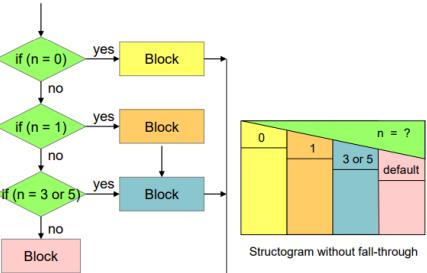


Code requires additional branch in case when <some code> is too long

Switch-Case

```
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;
}
```



Switch Statement Implementation C code example:

```
1 uint32_t result, n;
2 switch (n) {
3     case 0:
4         result += 17;
5         break;
6     case 1:
7         result += 13;
8         //fall through
9     case 3:
10    case 5:
11        result += 37;
12        break;
13    default:
14        result = 0;
15 }
```

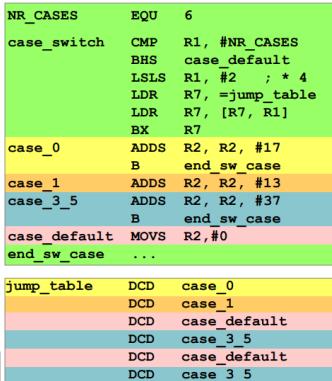
Assembly implementation with jump table:

```
1 NR_CASES      EQU      6
2 case_switch   CMP      R1, #NR_CASES
3                      BHS      case_default
4                      LSLS     R1, #2 ; * 4
5                      LDR      R7, =jump_table
6                      LDR      R7, [R7, R1]
7                      BX       R7
8
9 case_0        ADDS     R2, R2, #17
10 case_1       ADDS     R2, R2, #13
11 case_3_5     ADDS     R2, R2, #37
12                      B       end_sw_case
13
14 case_default MOVS     R2, #0
15 end_sw_case ...
16
17 jump_table   DCD      case_0
18                      DCD      case_1
19                      DCD      case_default
20                      DCD      case_3_5
21                      DCD      case_default
22                      DCD      case_3_5
```

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



Switch Implementation

1. Range check and table access:

```
1    CMP    R0, #MAX_CASES ; Check range
2    BHS    default_case ; If too high, default
3    LSLS   R0, #2        ; Multiply by 4
4    LDR    R1, =jump_table ; Load table address
5    ADD    R1, R0        ; Add offset
6    LDR    R1, [R1]      ; Load target address
7    BX     R1        ; Branch to case
```

2. Jump table structure:

```
1 jump_table
2     DCD    case_0      ; Case 0 handler
3     DCD    case_1      ; Case 1 handler
4     DCD    default_case ; Default handler
5
; ... more cases
```

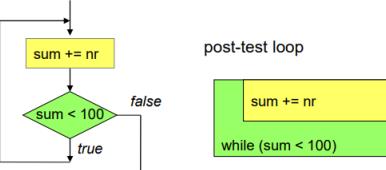
3. Case handlers:

```
1 case_0
2     ; Handle case 0
3     B      switch_end
4 case_1
5     ; Handle case 1
6     B      switch_end
7 default_case
8     ; Handle default case
9 switch_end
```

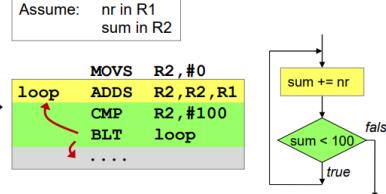
Loops

Do-While (Post-Test Loop) : Compiler translates **post-test** loops into assembly code using conditional branches:

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



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

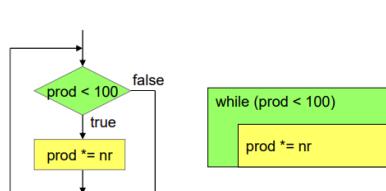


Do-While loop

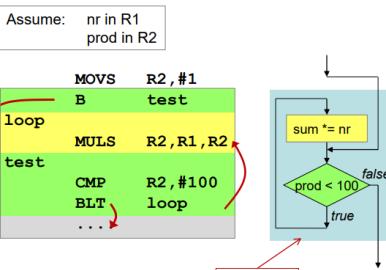
```
1 ; do { x++; } while (x < 10);
2 do_loop
3 ADDS R0, #1 ; x++
4 CMP R0, #10 ; Check x < 10
5 BLT do_loop ; Continue if true
```

While (Pre-Test Loop) : Compiler translates **pre-test** loops into assembly code reusing structure of do-while:

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



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



While loop

```
1 ; while (x < 10) { x++; }
2 B while_cond ; Jump to condition
3 while_loop
4 ADDS R0, #1 ; x++
5 while_cond
6 CMP R0, #10 ; Check x < 10
7 BLT while_loop ; Continue if true
```

For Loop (Pre-Test Loop)

For-Loops are converted into while-loops by the compiler. break and continue statements require special treatment.

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

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

For loop

```
1 ; for (i = 0; i < 10; i++)
2 MOVS R0, #0 ; i = 0
3 B for_cond
4 for_loop
5 ; Loop body
6 ADDS R0, #1 ; i++
7 for_cond
8 CMP R0, #10 ; Check i < 10
9 BLT for_loop ; Continue if true
```

Complex Control Structure

Implementing nested loops with conditions:

```
1 for (i = 0; i < 5; i++) {
2     if (i == 2) continue;
3     for (j = 0; j < 3; j++) {
4         if (j == 1) break;
5         sum += i + j;
6     }
7 }
```

```
1 MOVS R0, #0 ; i = 0
2 outer_loop
3 CMP R0, #2
4 BEQ outer_continue ; Skip if i == 2
5 ...
6 MOVS R1, #0 ; j = 0
7 inner_loop
8 CMP R1, #1
9 BEQ outer_continue ; Break to outer loop
10 ...
11 ADDS R2, R0, R1 ; Calculate i + j
12 ADDS R4, R4, R2 ; Add to sum
13 ...
14 ADDS R1, #1 ; j++
15 CMP R1, #3 ; Check j < 3
16 BLT inner_loop ; Continue inner loop
17 ...
18 outer_continue
19 ADDS R0, #1 ; i++
20 CMP R0, #5 ; Check i < 5
21 BLT outer_loop ; Continue outer loop
```

String Processing

String Processing Patterns

Common patterns for string manipulation:

1. String traversal:

```

1   MOVS   R2, #0          ; Index
2   loop
3   LDRB   R3, [R0, R2]    ; Load char
4   CMP    R3, #0          ; Check end
5   BEQ    done            ; Exit if terminator
6   ; Process character
7   ADDS   R2, R2, #1      ; Next char
8   B      loop

```

2. Character transformation:

```

1   ; Check character range
2   CMP    R3, #lower_bound
3   BLO    skip             ; Below range
4   CMP    R3, #upper_bound
5   BHI    skip             ; Above range
6
7   ; Transform character
8   ADDS   R3, #offset       ; Apply offset
9
10  skip
11  STRB   R3, [R1, R2]     ; Store result

```

3. String copy:

```

1   MOVS   R2, #0          ; Index
2   copy_loop
3   LDRB   R3, [R0, R2]    ; Load source
4   STRB   R3, [R1, R2]    ; Store to dest
5   ADDS   R2, R2, #1      ; Next char
6   CMP    R3, #0          ; Check end
7   BNE    copy_loop       ; Continue if not done

```

Important considerations:

- Choose appropriate conditional branches
- Consider signed vs unsigned comparisons
- Handle edge cases and termination
- Maintain proper register allocation
- Document complex control flow

String Processing Loop

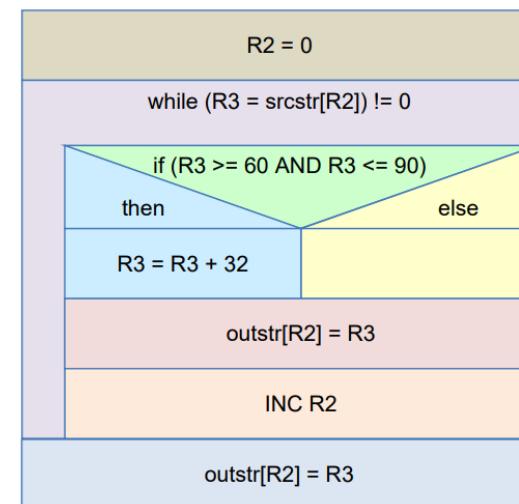
Converting string to uppercase:

```

1   AREA   progCode, CODE, READONLY
2   THUMB
3   main
4   PROC
5   EXPORT main
6   LDR    R0, =srcstr      ; Source string
7   LDR    R1, =outstr       ; Output string
8   MOVS   R2, #0           ; Initialize index
9
10  cond
11  LDRB   R3, [R0, R2]    ; Load character
12  CMP    R3, #0           ; Check for end
13  BEQ    endloop          ; Exit if done
14  CMP    R3, #60          ; Check if < 'a'
15  BLO    store             ; Skip if not lowercase
16  CMP    R3, #90          ; Check if > 'z'
17  BHI    store             ; Skip if not lowercase
18  ADDS   R3, R3, #32      ; Convert to uppercase
19
20  store
21  STRB   R3, [R1, R2]    ; Store character
22  ADDS   R2, R2, #1      ; Next character
23  B      cond              ; Continue loop
24
25  endloop
26  STRB   R3, [R1, R2]    ; Store terminator
27  ENDP
28
29  srcstr  DCB   "This IS mY TestString", 0
30  AREA   progData, DATA, READWRITE
31  outstr  SPACE  50

```

Structogram:

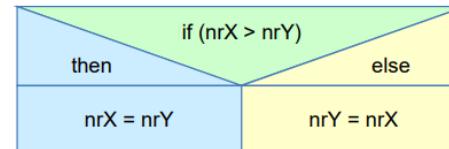


Result: "THIS IS MY TESTSTRING"

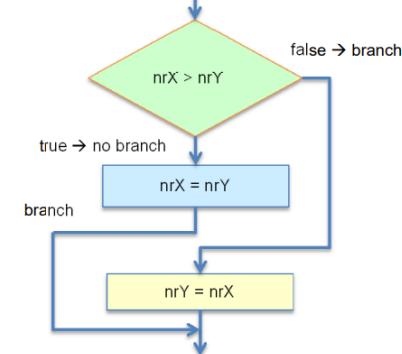
Examples

Selection Structures If-Then-Else with unsigned 8-bit variables:

Structogram:



Flowchart:



C code:

```

uint8_t nrX = ...;
uint8_t nrY = ...;
if (nrX > nrY) {
    nrX = nrY;
} else {
    nrY = nrX;
}

```

Assembly code:

```

AREA progCode, CODE, READONLY
THUMB

main PROC
EXPORT main

    LDR R6,=nrX ; R6 = address of nrX
    LDRB R0,[R6] ; R0 = byte stored at nrX
    LDR R7,=nrY ; R7 = address of nrY
    LDRB R1,[R7] ; R1 = byte stored at nrY
    CMP R0, R1
    BLS else ; *unsigned* comparison
    STRB R1,[R6] ; store nrY value at nrX
    B endif
    else
    STRB R0,[R7] ; store nrX value at nrY
endif

endless B      endless
ENDP

AREA progData, DATA, READWRITE

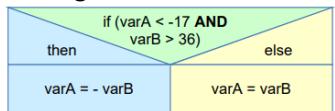
nrX  DCB   0x01 ; some 8 bit value
nrY  DCB   0xFF ; some other 8 bit value

END

```

Selection Structures If-Then-Else with signed 8-bit variables:

Structogram:

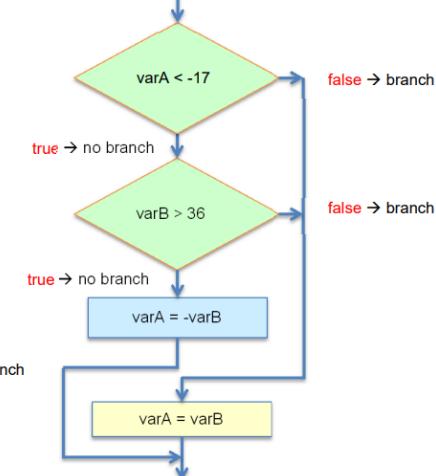


C code:

```

int8_t varA = ...;
int8_t varB = ...;
if (varA < -17 && varB > 36) {
    varA = -varB;
} else {
    varA = varB;
}
  
```

Flowchart:



Assembly code:

```

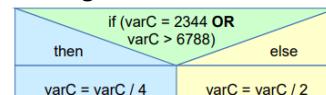
AREA progCode, CODE, READONLY
THUMB
main PROC
EXPORT main

LDR R6,=varA ; R6=address of varA
LDRB R0,[R6] ; R0=byte stored at varA
SXTB R0,R0 ; extend signed varA
LDR R7,=varB ; R7=address of varB
LDRB R1,[R7] ; R1=byte stored at varB
SXTB R1,R1 ; extend signed varB
MOVS R2,#17 ; +17
RSBS R2,R2 ; -17
CMP R0,R2
BGE else ; *signed* comparison
CMP R1,#36
BLE else ; *signed* comparison
RSBS R1,R1,#0 ; R1=-R1
STRB R1,[R6] ; varA = -varB
B endif
else STRB R1,[R6] ; varA = varB
endif
endless B endless
ENDP

AREA progData, DATA, READWRITE
varA DCB 123 ; some 8 bit value
varB DCB 45 ; some other 8 bit value
END
  
```

Selection Structures If-Then-Else with signed 16-bit variables:

Structogram:

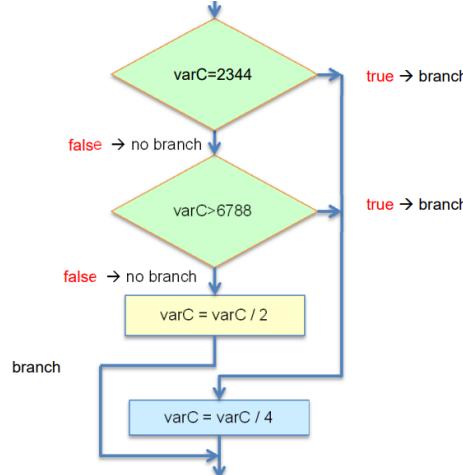


C code:

```

int16_t varC = ...;
if (varC == 2344 || varC > 6788) {
    varC = varC / 4;
} else {
    varC = varC / 2;
}
  
```

Flowchart:



Assembly code:

```

AREA progCode, CODE, READONLY
THUMB
main PROC
EXPORT main
LDR R7,=varC ; R7=address of varC
LDRH R0,[R7] ; R0=value stored at varC
SXTB R0,R0 ; extend signed varC
LDR R2,=2344
CMP R0,R2
BEQ then ; *signed* comparison
LDR R2,=6788
CMP R0,R2
BGT then ; *signed* comparison
ASRS R0,R0,#1 ; R0 = R0 / 2
STRH R0,[R7] ; varC = varC / 2
B endif
then ASRS R0,R0,#2 ; R0 = R0 / 4
STRH R0,[R7] ; varC = varC / 4
endif
endless B endless
ENDP

AREA progData, DATA, READWRITE
varC DCW 1234 ; some 16 bit value
END
  
```

For-Loop Implementation

Example for-loop in C and assembly:

```

// C code with volatile variables
volatile int32_t i = 0;
volatile int32_t count = 0;
for(i = 0; i < 10; i++) {
    count++;
}
  
```

Assembly implementation:

```

1 AREA progCode, CODE, READONLY
2 THUMB
3 main
4 PROC
5 EXPORT main
6 LDR R6, =i ; R6 = address of i
7 LDR R0, [R6] ; R0 = value at i
8 LDR R7, =count ; R7 = address of count
9 LDR R1, [R7] ; R1 = value at count
10 B cond
11
12 loop
13 ADDS R0, R0, #1 ; Increment i
14 ADDS R1, R1, #1 ; Increment count
15 cond
16 CMP R0, #10
17 BLT loop ; Branch if i < 10
18
19 STR R0, [R6] ; Store final i
20 STR R1, [R7] ; Store final count
21
22 ENDP
  
```

Compiler-optimized version:

```

1 LDR r1, [pc, #20] ; Load address
2 MOVS r0, #0 ; Initialize counter
3 STR r0, [r1, #0] ; Store i
4 LDR r2, [r1, #4] ; Load count
5
6 increment
7 ADDS r0, r0, #1 ; i++
8 ADDS r2, r2, #1 ; count++
9 CMP r0, #10 ; Check condition
10 BLT increment ; Loop if i < 10
11 STM r1!, {r0, r2} ; Store final values
  
```

Subroutines and Stack

Subroutine

- Subroutines Key elements of subroutines:
 - Label to identify subroutine entry point
 - Return instruction (BX LR) to exit
 - Proper register management

Call and Return Mechanism Basic subroutine mechanics:

- BL (Branch with Link):**
 - Stores current PC in LR (R14)
 - Branches to subroutine address
 - Direct and relative addressing
- BLX (Branch with Link and Exchange):**
 - Similar to BL but with register-specified target
 - Indirect and absolute addressing
- Return:** Using BX LR or POP ..., PC if LR was saved

Subroutine Calling Convention and Register Usage

- Calling Convention:**
 - Parameters passed in R0-R3
 - Return value in R0
 - Link Register (LR) for return address
 - Stack for additional parameters/locals
- Register Usage:**
 - R0-R3: Parameters and scratch
 - R4-R11: Must be preserved
 - R12: IP (scratch)
 - R13: SP (stack pointer)
 - R14: LR (link register)
 - R15: PC (program counter)

Subroutine Call and Return Multiply by 3 implementation:

```

1 MulBy3 MOV R4, R0 ; Save input value
2 LSLS R0, #1 ; Multiply by 2
3 ADD R0, R4 ; Add original value
4 BX LR ; Return

```

```

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

```

in detail:

- Label with name **MulBy3**
- Return Statement **BX LR**

Using Subroutines and Stack Steps for implementing subroutines:

- Define subroutine entry point with label
- Save registers that will be modified
 - Use PUSH at start
 - Include LR if calling other subroutines
- Implement subroutine logic
- Restore registers in reverse order
 - Use POP before return
 - Can return using POP ..., PC if LR was saved
- Return using BX LR if LR wasn't saved

- Important considerations:
- Always maintain stack alignment
 - Match PUSH/POP pairs exactly
 - Be careful with SP manipulation
 - Consider nesting depth for stack space

Subroutine Implementation

Guidelines for implementing subroutines:

1. Basic subroutine:

```

1 proc_name
2     PUSH {LR} ; Save return address
3     ; Subroutine code
4     POP {PC} ; Return

```

2. With register preservation:

```

1 proc_name
2     PUSH {R4-R7, LR} ; Save modified registers
3     ; Subroutine code using R4-R7
4     POP {R4-R7, PC} ; Restore and return

```

3. With local variables:

```

1 proc_name
2     PUSH {R4, LR} ; Save registers
3     SUB SP, SP, #8 ; Allocate locals
4     ; Use [SP] to [SP, #4] for locals
5     ADD SP, SP, #8 ; Deallocate locals
6     POP {R4, PC} ; Restore and return

```

Nested Subroutine Calls

Example of multiple nested calls with stack manipulation:

```

1 AREA progCode, CODE, READONLY
2 THUMB
3 main
4     LDR R1, =0x10203040 ; Initial values
5     LDR R2, =0x50607080
6     BL procA ; Call procA
7     BL procB ; Call procB
8     B endless
9
10 procA
11     PUSH {R1, R2} ; Save registers
12     LDR R1, =0xAABBCCDD ; New values
13     LDR R2, =0xEFF1020
14     POP {R1, R2} ; Restore registers
15     BX LR ; Return
16
17 procB
18     PUSH {R1, R2, LR} ; Save including LR
19     LDR R1, =0x11223344 ; New values
20     LDR R2, =0x55667788
21     BL procC ; Call procC
22     POP {R1, R2, PC} ; Return by popping PC
23
24 procC
25     PUSH {R1, R2, LR} ; Save registers
26     LDR R1, =0x11111111 ; New values
27     LDR R2, =0x22222222
28     BL procD ; Call procD
29     POP {R1, R2, PC} ; Return by popping PC

```

Stack contents at key points:

- After procA PUSH: R1(0x10203040), R2(0x50607080)
- After procB PUSH: R1, R2, LR(ret_addr)
- After procC PUSH: R1(0x11223344), R2(0x55667788), LR(ret_addr)

Stack

Stack characteristics:

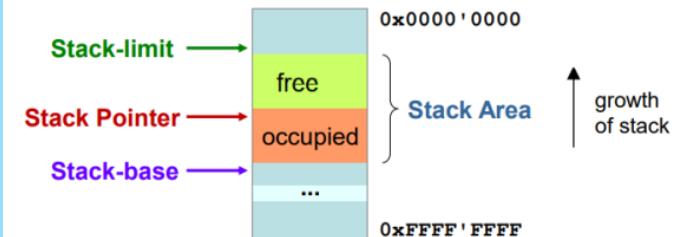
- Stack Area** (Section): Continuous RAM section
- Stack Pointer (SP)**: R13, points to last written value
- Direction**: Full-descending (grows toward lower addresses)
- Alignment**: Word-aligned (4 bytes)
- Data Size**: 32-bit words only

Main operations:

- PUSH**: Decrements SP, then stores words
- POP**: Loads words, then increments SP

Stack constraints:

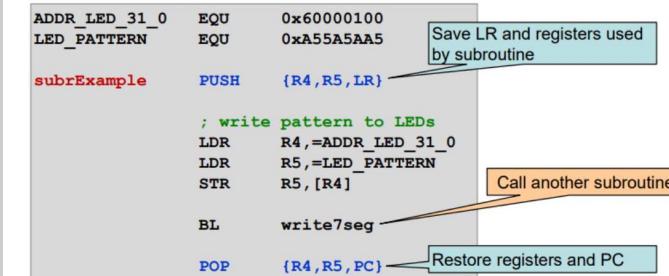
- Number of PUSH and POP operations must match
- SP must stay between stack-limit and stack-base
 - **Stack-limit** ≤ SP ≤ **Stack-base**



Stack Instructions Special stack manipulation instructions:

- ADD/SUB SP**:
 - Immediate offset 0-508
 - Must be multiple of 4
- SP-relative LDR/STR**:
 - Immediate offset 0-1020
 - Used for frame access
- PUSH/POP**:
 - Multiple register transfer
 - Maintains alignment
 - Can include PC/LR

PUSH/POP Implementation



```

1 ; PUSH {R2,R3,R6}
2 SUB SP, SP, #12 ; Reserve stack space
3 STR R2, [SP] ; Store R2
4 STR R3, [SP, #4] ; Store R3
5 STR R6, [SP, #8] ; Store R6
6
7 ; POP {R2,R3,R6}
8 LDR R2, [SP] ; Restore R2
9 LDR R3, [SP, #4] ; Restore R3
10 LDR R6, [SP, #8] ; Restore R6
11 ADD SP, SP, #12 ; Free stack space

```

Stack Operations and Functions

Stack Operations

Common stack manipulation patterns:

- **Register Save/Restore:**
 - PUSH/POP for callee-saved registers
 - Multiple register transfer
- **Local Variables:**
 - SUB SP to allocate space
 - Access via SP-relative addressing
 - ADD SP to deallocate space
- **Return Handling:**
 - Save LR if making calls
 - Return via BX LR or POP {PC}
 - Use PC in POP list when LR saved

Function Implementation Patterns

1. Simple function:

```

1 func    PUSH    {LR}          ; Save return address
2           ; Function body
3     POP    {PC}          ; Return

```

2. Function with locals:

```

1 func    PUSH    {R4, LR}      ; Save registers
2           SUB    SP, #8        ; Space for locals
3           ; Function body
4     ADD    SP, #8        ; Remove locals
5     POP    {R4, PC}      ; Return

```

3. Function with parameters:

```

1           ; R0-R3 = first 4 parameters
2           ; [SP] = fifth parameter
3 func    PUSH    {R4-R6, LR}   ; Save registers
4           LDR    R4, [SP, #16] ; Load 5th param
5           ; Function body
6     POP    {R4-R6, PC}      ; Return

```

Stack Frame Example of complete function:

```

1 ; int calc(int a, int b, int c)
2 ; a in R0, b in R1, c in R2
3 calc   PUSH    {R4-R6, LR}   ; Save registers
4           ; Save parameters
5     MOVS   R4, R0          ; Save a
6     MOVS   R5, R1          ; Save b
7     MOVS   R6, R2          ; Save c
8           ; Call helper function
9     MOVS   R0, R4          ; First param
10    BL     helper          ; Call helper
11           ; Continue calculation
12    ADDS   R0, R5          ; Add b
13    ADDS   R0, R6          ; Add c
14
15     POP    {R4-R6, PC}      ; Return

```

Stack usage considerations:

- Monitor stack depth in nested calls
- Always maintain 8-byte alignment for SP
- Consider register usage to minimize stack operations
- Be aware of stack space in interrupt handlers
- Document stack requirements for functions

Stack Frame

Stack Frame Structure

Components of a stack frame:

- **Saved Registers:**
 - Caller-saved (R0-R3, R12)
 - Callee-saved (R4-R11)
 - Link register (LR)
- **Local Variables:**
 - Allocated on stack if needed
 - Word-aligned access
- **Parameters:**
 - Beyond R0-R3 if needed
 - Pushed by caller

Stack Frame Layout/Management

Steps for function prologue and epilogue, and guidelines for managing stack frames:

1. Frame structure:

- Previous stack frame
- Return address (LR)
- Saved registers
- Local variables
- Parameters for called functions

2. Frame creation/Function prologue:

```

1           ; Save registers and create frame
2     PUSH    {R4-R7, LR}      ; Save registers
3     SUB    SP, SP, #frame_size ; Allocate space
4     SUB    SP, SP, #locals   ; Allocate local vars
5
6           ; Initialize frame if needed
7     MOV    R4, #0            ; Clear locals
8     STR    R4, [SP, #0]      ; Initialize var1
9     STR    R4, [SP, #4]      ; Initialize var2

```

3. Stack frame access:

```

1           ; Access local variables
2     LDR    R0, [SP, #offset1] ; Load local1
3     STR    R1, [SP, #offset2] ; Store to local2
4
5           ; Access parameters
6     LDR    R0, [SP, #20]     ; First stack parameter
7           ; Access parameters beyond R0-R3
8     LDR    R0, [SP, #param_offset] ; Load param

```

4. Frame cleanup/Function epilogue:

```

1           ; Deallocate frame and restore
2     ADD    SP, SP, #frame_size ; Remove locals
3     ADD    SP, SP, #locals   ; Deallocate locals
4     POP    {R4-R7, PC}      ; Restore and return

```

Important considerations:

- Maintain 8-byte stack alignment
- Save LR before any BL instructions
- Properly pair PUSH/POP operations
- Document stack frame layout
- Track stack depth in nested calls

Stack Frame Management

Stack frame creation and cleanup:

```

1 func    ; Function prologue
2     PUSH    {R4-R8, LR}      ; Save registers
3     SUB    SP, SP, #12       ; Allocate locals
4     ; Access local variables relative to SP
5     STR    R0, [SP, #0]      ; Local var 1
6     STR    R1, [SP, #4]      ; Local var 2
7     STR    R2, [SP, #8]      ; Local var 3
8     ; Function body
9     BL     other_func      ; Call another function
10    ; Function epilogue
11     ADD    SP, SP, #12       ; Deallocate locals
12     POP    {R4-R8, PC}      ; Restore and return

```

Examples

Stack Analysis in Nested Subroutine Calls

from exercise sheet 8

Steps to analyze stack content in nested subroutine calls:

1. **Track initial stack pointer value:** Note starting SP value
2. **For each PUSH instruction:** Subtract 4 bytes per register
 - Write values in order specified
 - Remember which subroutine saved what
3. **For subroutine calls (BL):** LR gets set to return address
 - If LR needs to be saved (nested calls), it must be PUSHed
4. **Keep track through nested calls:**
 - Each nested level maintains its own stack frame
 - Stack unwinds in reverse order when returning

Simple Stack Analysis

Consider this simple nested call sequence:

```

1 main   ; SP = 0x20002000
2     LDR    R0, =0x11111111
3     LDR    R1, =0x22222222
4     BL     funcA          ; Z1
5     B     endless
6
7 funcA  PUSH    {R0, R1, LR} ; Z2
8     BL     funcB          ; Z2
9     POP    {R0, R1, PC}
10    funcB  PUSH    {R4, LR}   ; Z3
11    MOVS   R4, #42
12    POP    {R4, PC}

```

Stack contents at Z1, Z2, Z3:

```

1 Z1: Stack is empty, SP = 0x20002000
2
3 Z2: Stack from top (higher address):
4 0x20001FF4: Return addr to main
5 0x20001FF0: 0x22222222 (R1)
6 0x20001FEC: 0x11111111 (R0)
7 SP = 0x20001FEC
8
9 Z3: Stack from top (higher address):
10 0x20001FE4: Return addr to funcA
11 0x20001FE0: R4 value
12 0x20001FF4: Return addr to main (from Z2)
13 0x20001FF0: 0x22222222 (R1 from Z2)
14 0x20001FEC: 0x11111111 (R0 from Z2)
15 SP = 0x20001FEC

```

Parameter Passing

Parameter Passing Methods

- Registers:** Fast, limited number available
→ Caller and Callee use the same register
- Global Variables:** Shared memory space
- Stack:**
 - Caller: PUSH parameters onto stack
 - Callee: Access parameter via LDR from stack

Global variable approach

NOT recommended!!

- Shared variables in data area
- Overhead to access variable
- Error prone, unmaintainable

```

AREA exData,DATA,...
param1 SPACE 1
result SPACE 1
AREA exCode,CODE,...
    LDR R4,=param1
    MOVS R5,#0x03
    STRB R5,[R4]
    BL double_g
    LDR R4,=result
    LDRB ...
caller
double_g
    LDR R4,=param1
    LDRB R1,[R4]
    LSLS R0,R1,#1
    LDR R4,=result
    STRB R0,[R4]
    BX LR
callee
function
double_g
  
```

Register-based approach (preferred): There are two main approaches in Register-based approach: by value or by reference

```

1 func  PUSH {R4, LR} ; Save registers
2      ; R0 contains input parameter
3      MOV R4, R0 ; Save parameter
4      ; Process value in R4
5      MOV R0, R4 ; Set return value
6      POP {R4, PC} ; Restore and return
  
```

Parameter Passing by Value vs. Reference

- Pass by Value:**
 - Copies value to function
 - Changes don't affect original
 - Default in C
 - Example:
Simple types, integers
 - Limited numbers of registers
- Pass by Reference:**
 - Passes memory address
 - Changes affect original value
 - In C: Using pointers
 - Example:
Arrays, large structures

Example implementation:

```

1 ; Pass by value
2 func1 PUSH {LR}
3      ADDS R0, #1 ; Modify parameter
4      POP {PC} ; Original unchanged
5 ; Pass by reference
6 func2 PUSH {LR}
7      LDR R1, [R0] ; Load from address
8      ADDS R1, #1 ; Modify value
9      STR R1, [R0] ; Store back to address
10     POP {PC} ; Original changed
  
```

ARM Procedure Call Standard

Parameter Passing:

- Caller copies parameters from R0 to R3
- Caller copies additional parameters to stack
- Return Values:**
 - Small Values** (≤ 32 bits → smaller than word size):
 - Return in R0
 - Zero/sign extend to word if needed
 - Word** (32 bits): return in R0/R1
 - Double Word** (64 bits): return in R0/R1/R2/R3
 - 128-bit Values**: return in R0-R3
 - Composite data types** (structs, arrays):
 - Up to 4 bytes: return in R0
 - Larger: return pointer in R0 (stored in data area)

Register Usage:

- R0-R3: Arguments/results (caller-saved)
- R4-R11: Local variables (callee-saved)
- R12: IP - scratch register
- R13: SP - stack pointer
- R14: LR - link register
- R15: PC - program counter

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)

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

Stack Frame Organization

Current Frame:

- Arguments 5+
- Return address (LR)
- Saved registers (R4-R11)
- Local variables
- Temporary storage

Previous Stack Frame:

- Local variables
- Saved registers

Next Frame:

- Space for called functions

Implementing Function Calls

Caller's responsibilities:

- Place parameters in R0-R3
- Push additional parameters on stack
- Save caller-saved registers if needed

Steps for calling functions:

Callee's responsibilities:

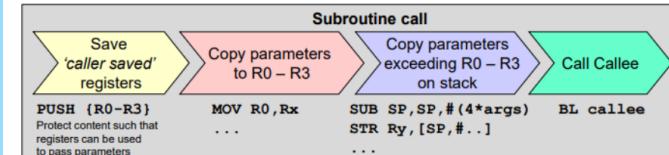
- Save callee-saved registers used
- Save LR if making other calls
- Process parameters
- Place return value in R0
- Restore saved registers

Important considerations:

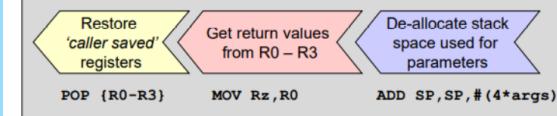
- Avoid global variables for parameter passing
- Use registers for efficiency
- Follow ARM calling convention strictly
- Consider stack usage in recursive functions

Subroutine Call Caller Side

Pattern as used by the compiler. Manually written code may differ.



On return from subroutine



Function Parameter Guidelines

Best practices for parameter passing:

- 1. Register Usage:**
 - R0-R3: First four parameters
 - R0: Return value
 - R4-R11: Preserve if used
- 3. Memory Structures:**
 - Pass pointers for large structures
 - Use registers for small values
 - Consider alignment requirements

2. Stack Usage:

- Additional parameters pushed right to left
- Maintain 8-byte alignment
- Caller responsible for cleaning up stack

Example implementation:

```

1 ; void func(int a, int b, int c, int d, int e)
2 ; First four params in R0-R3, fifth on stack
3 func  PUSH {R4-R6, LR} ; Save registers
4      ; Save parameters
5      MOV R4, R0 ; Save a
6      MOV R5, R1 ; Save b
7      MOV R6, R2 ; Save c
8      ; R3 contains d
9      LDR R0, [SP, #16] ; Load e from stack
10
11     ; Function body
12     POP {R4-R6, PC} ; Return
  
```

Reentrancy

Handling recursive function calls:

- Each call needs its own data set
- Registers/globals get overwritten
- Solution: ARM Procedure Call Standard

Recursive Function Implementation

Factorial calculation:

```

1 ; uint32_t factorial(uint32_t n)
2 ; Input in R0, result in R0
3 factorial
4     PUSH {R4, LR} ; Save registers
5     MOVS R4, R0 ; Save n
6     CMP R4, #1 ; Check base case
7     BLE fact_end ; Return 1 if n <= 1
8
9     SUBS R0, R4, #1 ; n-1
10    BL factorial ; Recursive call
11    MULS R0, R4, R0 ; n * factorial(n-1)
12
13 fact_end
14     POP {R4, PC} ; Restore and return
  
```

Examples

ARM Parameter Passing

Key rules for parameter passing:

- **Register Parameters:**
 - First four parameters in R0-R3
 - Additional parameters on stack
 - Return value in R0 (or R0/R1 for 64-bit)
- **Stack Parameters:**
 - Pushed right-to-left
 - 8-byte aligned
 - Caller responsible for cleanup
- **Return Values:**
 - 32-bit or less in R0
 - 64-bit in R0 and R1
 - Larger values via pointer

Parameter Passing Guidelines

Rules for implementing function calls:

1. Caller responsibilities:

```

1 ; Save any needed registers
2 PUSH {R4-R6, LR} ; Save registers
3
4 ; Load parameters into R0-R3
5 MOV R0, R4 ; First parameter
6 MOV R1, R5 ; Second parameter
7 MOV R2, R6 ; Third parameter
8
9 ; Call function
10 BL function
11
12 ; Save return value if needed
13 MOV R7, R0 ; Save result
14
15 ; Restore registers
16 POP {R4-R6, PC} ; Return

```

2. Callee responsibilities:

```

function
1 ; Save any registers we'll modify
2 PUSH {R4, LR}
3
4 ; Process parameters in R0-R3
5 ; Put return value in R0
6
7 ; Restore registers
8 POP {R4, PC}

```

3. Reference parameter handling:

```

1 ; Loading from pointer
2 LDR R2, [R0] ; Get value at address
3 ; Storing to pointer
4 STR R2, [R1] ; Store to address
5 ; Incrementing pointer
6 ADD R0, R0, #4 ; Next word

```

Important considerations:

- Track register usage and preservation
- Consider stack alignment requirements
- Be aware of parameter passing limits
- Handle return values consistently
- Monitor stack growth in recursion

Pass by Value vs Reference Parameter passing issues:

```

1 // Incorrect swap - pass by value
2 void swap_bad(int32_t c, int32_t d) {
3     int32_t temp = c;
4     c = d;
5     d = temp;
6 }
7 // Correct swap - pass by reference
8 void swap_good(int32_t *c, int32_t *d) {
9     int32_t temp = *c;
10    *c = *d;
11    *d = temp;
12 }
13 int32_t main(void) {
14     int32_t a = 3, b = 5;
15
16     swap_bad(a, b); // Doesn't work
17     swap_good(&a, &b); // Works correctly
18 }

```

Assembly implementation of swap_good:

```

1 swap_good
2     PUSH {LR} ; Save return address
3
4     LDR R2, [R0] ; Load *c into R2
5     LDR R3, [R1] ; Load *d into R3
6
7     STR R3, [R0] ; Store R3 to *c
8     STR R2, [R1] ; Store R2 to *d
9
10    POP {PC} ; Return

```

Complex Parameter Example Function with mixed parameter types:

```

1 typedef struct {
2     int32_t x;
3     int32_t y;
4 } point_t;
5
6 int32_t calculate(point_t* p, int32_t scale,
7                     int32_t* result);

```

Assembly implementation:

```

1 ; R0 = point_t* p
2 ; R1 = scale
3 ; R2 = result pointer
4 calculate
5     PUSH {R4-R5, LR} ; Save registers
6
7     ; Load structure members
8     LDR R4, [R0, #0] ; Load p->x
9     LDR R5, [R0, #4] ; Load p->y
10
11    ; Perform calculation
12    MULS R4, R1, R4 ; x * scale
13    MULS R5, R1, R5 ; y * scale
14
15    ; Store result
16    STR R4, [R2, #0] ; *result = x
17    ADDS R0, R4, R5 ; Return sum
18
19    POP {R4-R5, PC} ; Return

```

Data Structure Access Working with structures and arrays:

```

1 typedef struct {
2     uint32_t minutes;
3     uint32_t seconds;
4 } time_t;
5
6 time_t time;

```

Assembly implementation:

```

1 ; Access structure members
2 LDR R0, =time ; Get structure address
3 LDR R1, [R0, #0] ; Load minutes
4 LDR R2, [R0, #4] ; Load seconds
5
6 ; Modify structure
7 ADDS R2, #1 ; Increment seconds
8 CMP R2, #60 ; Check for overflow
9 BLT store_back
10 MOVS R2, #0 ; Reset seconds
11 ADDS R1, #1 ; Increment minutes
12 store_back
13 STR R1, [R0, #0] ; Store minutes
14 STR R2, [R0, #4] ; Store seconds

```

Recursive Function Example

Factorial calculation showing stack usage:

```

1 int32_t fakultaet_recursive(int32_t n) {
2     if(n < 2) {
3         return 1;
4     } else {
5         return n * fakultaet_recursive(n-1);
6     }
7 }
8
9 int32_t main(void) {
10     int32_t n = 20;
11     int32_t result = fakultaet_recursive(n);
12 }

```

Assembly implementation showing stack growth:

```

1 fakultaet_recursive
2     PUSH {R4, LR} ; Each call adds 8 bytes
3     MOV R4, R0 ; Save n
4
5     CMP R0, #2 ; Check base case
6     BLT return_one
7
8     SUB R0, R0, #1 ; n-1
9     BL fakultaet_recursive
10    MUL R0, R4, R0 ; n * result
11
12    POP {R4, PC} ; Return
13
14 return_one
15    MOV R0, #1 ; Return 1
16    POP {R4, PC} ; Return

```

Maximum stack size = 8 bytes * 19 calls = 152 bytes

Function Call Example

C function and its parameter passing:

```
1 uint32_t logical_and(uint32_t a, uint32_t b, uint32_t
2     c) {
3     return a & b & c;
4 }
5 int32_t main(void) {
6     uint32_t x = 0x11223344; // In R4
7     uint32_t y = 0xFFFF0000; // In R5
8     uint32_t z = 0x33661122; // In R6
9     uint32_t result; // In R7
10
11     result = logical_and(x, y, z);
12 }
```

Beim Start des Programmes wird die Variable x in R4, y in R5 und z in R6 abgelegt. Die Variable result wird in R7 abgelegt.

1. Welche Schritte führt der Caller (main) vor dem Aufruf der Funktion logical_and() durch? Wie werden die Parameter übergeben?

Die Variablen in den Registern R4 bis R6 werden nach R0 bis R2 kopiert und so der Funktion übergeben. R6 → R2, R5 → R1, R4 → R0

2. Wie gibt die Funktion logical_and() den Rückgabewert zurück?

Der Rückgabewert wird via R0 zurückgegeben.

3. Welche Operation führt der Call nach dem Aufruf der Funktion logical_and() durch?

Der Rückgabewert wird von R0 nach R7 kopiert

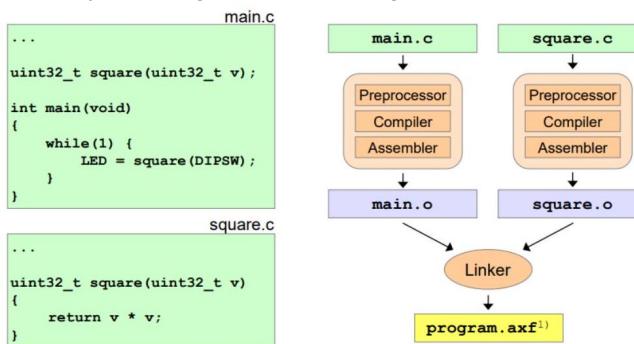
Assembly implementation showing parameter passing:

```
1 main
2 ; Initial register setup
3 LDR    R4, =0x11223344 ; x
4 LDR    R5, =0xFFFF0000 ; y
5 LDR    R6, =0x33661122 ; z
6
7 ; Parameter passing
8 MOV    R0, R4           ; a = x
9 MOV    R1, R5           ; b = y
10 MOV   R2, R6           ; c = z
11 BL    logical_and
12 MOV    R7, R0           ; result = return value
13
14 logical_and
15 AND    R0, R0, R1       ; a & b
16 AND    R0, R0, R2       ; & c
17 BX    LR               ; Return
```

Modular Coding and Linking

From source code to executable Program

Compile/Assemble each Module: Results in an object file for each module
Link all object files together: Creates a single executable file



Tool Chain Components

- Compiler (armcc):**
 - Translates C to assembly
 - Performs optimizations
 - Generates object files
- Assembler (armasm):**
 - Processes assembly code
 - Creates object files
 - Handles directives
- Linker (armlink):**
 - Combines object files
 - Resolves references
 - Creates executable
- Library Manager (armar):**
 - Creates/maintains libraries
 - Adds/removes object files
 - Archives multiple objects

Guidelines for Modular Programming

- Key design principles:
- High Cohesion:** Group related functionality together
 - Each module fulfills a single defined task
 - Lean external interface
 - Low Coupling:** Minimize dependencies between modules
 - Clear and minimal interfaces
 - Easy to modify individual modules
 - Information Hiding:** Split interface from implementation
 - Don't expose unnecessary details
 - Maintain freedom to change internals

Benefits of Modular Programming

Key advantages:

Team Development: Multiple developers working on same codebase

- Clear ownership of modules

Code Organization: Logical partitioning/grouping of functionality

- Easier code reuse and better maintainability/understandability

Development Efficiency:

- Individual module testing
- Faster compilation (only recompile changed modules)
- Reusable library creation

Language Integration: Mix C and assembly modules

- Language-specific optimizations (best of both worlds)

Important considerations:

- Use consistent naming conventions
- Document module interfaces clearly
- Consider initialization dependencies
- Test modules independently
- Maintain version control
- Document build requirements

Module Interfaces and Linkage

Module Linkage

Keywords for controlling module interfaces:

- EXPORT:** Make symbol available to other modules
- IMPORT:** Use symbol from another module
- Internal symbols:**
Neither IMPORT nor EXPORT

```

; main.s
AREA myCode, CODE, READONLY
EXPORT main
IMPORT square <-- from module square

main PROC
    LDR r0, a_adr
    LDR r0, [r0, #0] ; a
    BL square
    ...
ENDP

a_adr DCD a
b_adr DCD b

AREA myData, DATA
DCD 0x00000005
DCD 0x00000007

```

Working with Linkage in ARM Assembly

- Exporting symbols:** Use EXPORT directive before symbol definition
 - Only export symbols needed by other modules
 - Can export both code and data symbols
- Importing symbols:** Use IMPORT directive before symbol usage
 - Import must match exactly with export name
 - Import before first use of symbol
- Internal symbols:** No EXPORT/IMPORT directive
 - Only visible within current module
 - Used for module-private implementation

Multiple Module Linkage

Example of two interacting modules:

Module 1 (math_ops.s):

```

1 AREA .text!, CODE, READONLY
2 EXPORT add_and_square
3 IMPORT square_value ; From module 2
4
5 add_and_square PROC
6 PUSH {R4, LR}
7 ADDS R4, R0, R1 ; Add parameters
8 MOVS R0, R4 ; Prepare for square
9 BL square_value ; Call imported function
10 POP {R4, PC}
11 ENDP
12 END

```

Module 2 (square.s):

```

1 AREA .text!, CODE, READONLY
2 EXPORT square_value
3
4 square_value PROC
5 PUSH {LR}
6 MULS R0, R0, R0 ; Square the input
7 POP {PC}
8 ENDP
9 END

```

Important considerations for ARM assembly linkage:

- Only export symbols that are part of the module's interface
- Keep internal helper functions and data private
- Use consistent naming conventions for public/private symbols
- Document the purpose and usage of exported symbols
- Consider implications of global data on reentrancy

Common Linkage Patterns in ARM Assembly

1. Interface functions:

```

1 AREA .text!, CODE, READONLY
2 EXPORT module_init ; Public interface
3 EXPORT module_process

4 module_init PROC
5 PUSH {LR}
6 BL internal_init ; Private implementation
7 POP {PC}
8 ENDP

11 module_process PROC
12 ; Public processing function
13 ENDP

15 internal_init PROC
16 ; Private initialization
17 ENDP

```

2. Shared data structures:

```

1 AREA myData, DATA, READWRITE
2 EXPORT module_state ; Public state
3
4 module_state SPACE 16 ; Accessible by other modules
5
6 private_buffer SPACE 32 ; Internal buffer
7
8

```

3. Constants and configuration:

```

1 AREA myConstants, DATA, READONLY
2 EXPORT CONFIG_VERSION
3 EXPORT MAX_BUFFER_SIZE
4
5 CONFIG_VERSION
6 DCD 0x00000001
7
8 MAX_BUFFER_SIZE
9 DCD 0x00000010
10
11 internal_config ; Private configuration
12 DCD 0x00000010

```

Library Usage Example

Linkage:

```

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

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

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

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

```

Linkage in C

Linkage Types in C Three types of linkage:

External Linkage:

- Global names available to all modules
- Default for functions and global variables

```
1 int global_var;           // External linkage
2 void global_func(void);   // External linkage
```

Internal Linkage:

- Names only available within module
- Created using 'static' keyword

```
1 static int module_var;    // Internal linkage
2 static void local_func(void); // Internal linkage
```

No Linkage:

- Local variables and function parameters
- Scope limited to block

```
1 void func(void) {
2     int local_var;          // No linkage
3     static int static_var; // Internal linkage
4 }
```

Module Interface Design

Example of a well-designed modular system:

```
1 // sensor.h - Public interface
2 typedef struct {
3     uint32_t id;
4     uint32_t value;
5 } sensor_reading_t;
6
7 void sensor_init(uint32_t id);
8 sensor_reading_t sensor_read(void);
9
10 // sensor.c - Implementation
11 static uint32_t current_sensor_id;
12 static uint32_t last_reading;
13
14 void sensor_init(uint32_t id) {
15     current_sensor_id = id;
16     last_reading = 0;
17 }
18
19 sensor_reading_t sensor_read(void) {
20     sensor_reading_t reading;
21     reading.id = current_sensor_id;
22     reading.value = /* Read hardware */;
23     last_reading = reading.value;
24     return reading;
25 }
```

Linker Input and Output

Linker Input - Object Files

- **Code Section:** Based at address 0x0 – Program code and constant data (READONLY) of the module
- **Data Section:** Based at address 0x0 – all global variables and initialized data
- **Symbol Table:** References to external symbols – All symbols with their attributes like global/local status
- **Relocation Table:** Instructions for adjusting addresses:
 - which bytes of the data and code sections need to be modified (and how) after merging the sections in the linking process
 - Applied during linking process

ARM tool chain uses ELF (Executable and Linkable Format)

Object File Structure

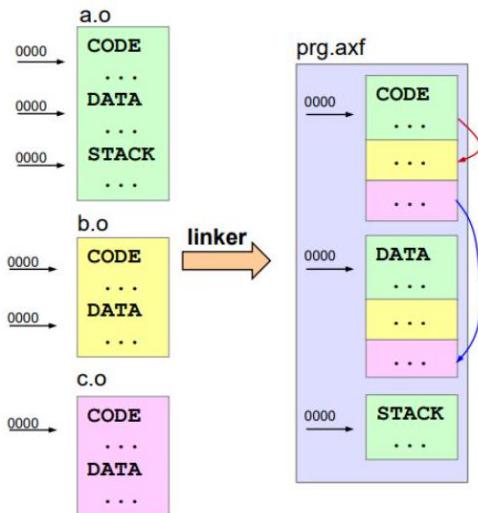
```
1 ; 1. '.text' section (Code):
2 0x00000000: 4604 MOV r4,r0
3 0x00000002: 0040 LSLS r0,r0,#1
4 0x00000004: 4420 ADD r0,r4
5 ; 2. '.data' section:
6 0x00000000: Initial values for global data
7 ; 3. Symbol table:
8 ; # Name      Value  Type   Binding
9 6 myFunc   0x0000 CODE   Global
10 7 extVar    0x0000 DATA   Reference
11 ; 4. Relocation entries:
12 Offset   Type      Symbol
13 0x0006 R_ARM_REL32 extVar
```

Linker Operation

Linker tasks:

- Merge object file code sections
- Merge object file data sections
- Symbol Resolution: Resolve symbol references between modules
- Address relocation: Relocate addresses to final positions

Linker output: AXF = ARM executable file



Symbol Resolution and Relocation

Steps in linking process:

1. Symbol Resolution:

```
1 ; In module1.s
2 AREA .text, CODE, READONLY
3 EXPORT func1
4 func1
5 ; function code
6
7 ; In module2.s
8 AREA .text, CODE, READONLY
9 IMPORT func1
10 BL func1 ; Reference to resolve
```

2. Relocation:

```
1 ; Before relocation
2 BL func1 ; Relative offset
3
4 ; After relocation
5 BL 0x08000234 ; Absolute address
```

Linking Multiple Modules

Example of linking multiple assembly modules:

```
1 ; math_ops.s
2 AREA .text, CODE, READONLY
3 EXPORT add_values
4 EXPORT multiply_values
5
6 add_values PROC
7 ADDS R0, R0, R1
8 BX LR
9 ENDP
10
11 multiply_values PROC
12 MULS R0, R1, R0
13 BX LR
14 ENDP
15 END
16
17 ; mains.s
18 AREA .text, CODE, READONLY
19 IMPORT add_values
20 IMPORT multiply_values
21 EXPORT main
22
23 main PROC
24 PUSH {R4,LR}
25 MOVS R0, #5
26 MOVS R1, #3
27 BL add_values ; Call add_values
28 MOVS R4, R0 ; Save result
29 MOVS R0, #6
30 MOVS R1, #2
31 BL multiply_values ; Call multiply_values
32 ADD R0, R0, R4 ; Combine results
33 POP {R4,PC}
34 ENDP
35 END
```

Creating and Using Libraries

Library Creation and Use

Steps for creating and using libraries:

1. Create library source files:

```
1 // lib.h
2 void lib_func(int x);
3
4 // lib.c
5 void lib_func(int x) {
6     // Implementation
7 }
```

2. Compile to object files:

```
1 armcc -c lib.c -o lib.o
```

3. Create static library:

```
1 armar --create libmy.a lib.o
```

4. Link with library:

```
1 armlink main.o libmy.a -o program.axf
```

Creating Static Libraries

Creating and using a static library:

1. Create object files:

```
1 armcc -c math_ops.c -o math_ops.o
2 armcc -c string_ops.c -o string_ops.o
```

2. Create static library:

```
1 armar --create libutils.a math_ops.o string_ops.o
```

3. Link with library:

```
1 armlink main.o libutils.a -o program.axf
```

Example C code for the library:

```
1 // math_ops.h
2 uint32_t add_values(uint32_t a, uint32_t b);
3 uint32_t multiply_values(uint32_t a, uint32_t b);
4
5 // math_ops.c
6 uint32_t add_values(uint32_t a, uint32_t b) {
7     return a + b;
8 }
9
10 uint32_t multiply_values(uint32_t a, uint32_t b) {
11     return a * b;
12 }
```

Creating a Multi-Module Project

Steps for creating a modular program:

1. Design module structure:

- Identify clear module boundaries and responsibilities
- Define external interfaces between modules
- Plan dependencies between modules

2. Create header files (.h):

```
1 // module.h
2 #ifndef MODULE_H_
3 #define MODULE_H_
4
5 // Public type definitions
6 typedef struct {
7     uint32_t x;
8     uint32_t y;
9 } point_t;
10
11 // Public function declarations
12 void init_module(void);
13 uint32_t process_point(point_t* p);
14
15#endif
```

3. Create implementation files (.c):

```
1 // module.c
2 #include "module.h"
3
4 // Private/static functions
5 static void helper_function(void) {
6     // Internal implementation
7 }
8
9 // Public function implementations
10 void init_module(void) {
11     // Initialize module state
12 }
13
14 uint32_t process_point(point_t* p) {
15     helper_function();
16     return p->x + p->y;
17 }
```

4. Create makefile/build configuration:

- Define compilation and linking rules
- Specify dependencies between modules
- Configure optimization and debug settings

Examples

Creating Modular Programs

Steps for modular development:

1. Design module structure:
 - Identify clear boundaries
 - Define interfaces
2. Create individual modules:
 - Declare IMPORT/EXPORT
 - Implement functionality
3. Compile modules separately
4. Link modules:
 - Resolve references
 - Create executable
5. Test integrated system

Module Linkage Example

Example implementations:

```
1  AREA myData, DATA, READWRITE
2  EXPORT global_counter      ; Public variable
3  global_counter
4  DCD    0x00000000
5
6  local_counter             ; Private variable
7  DCD    0x00000000
8
9  AREA myCode, CODE, READONLY
10 EXPORT init_counter       ; Public function
11 EXPORT increment
12
13 init_counter PROC         ; Can be called from other
   modules
14   PUSH {LR}
15   LDR  R0, =global_counter
16   MOVS R1, #0
17   STR  R1, [R0]
18   POP  {PC}
19 ENDP
20
21 increment PROC            ; Can be called from other
   modules
22   PUSH {LR}
23   BL   internal_helper ; Private function call
24   POP  {PC}
25 ENDP
26
27 internal_helper PROC      ; Private function
28   PUSH {LR}
29   LDR  R0, =global_counter
30   LDR  R1, [R0]
31   ADDS R1, #1
32   STR  R1, [R0]
33   POP  {PC}
34 ENDP
```

Module Interface Example

```
1  ; Module A - Defining function
2  AREA myCode, CODE, READONLY
3  EXPORT myFunction          ; Make available externally
4  myFunction
5  PUSH {LR}
6  ; function code here
7  POP {PC}
8
9  ; Module B - Using function
10 AREA myCode, CODE, READONLY
11 IMPORT myFunction          ; Use external function
12
13 BL myFunction             ; Call the function
```

Data Symbol Linkage

Example showing data symbol linkage:

Header file (constants.s):

```
1  AREA myData, DATA, READONLY
2  EXPORT MAX_VALUE
3  EXPORT MIN_VALUE
4
5  MAX_VALUE
6  DCD    0x000000FF      ; Public constant
7  MIN_VALUE
8  DCD    0x00000000      ; Public constant
9
10 internal_value           ; Private constant
11 DCD    0x00000055
12 END
```

Usage file (process.s):

```
1  AREA .text, CODE, READONLY
2  IMPORT MAX_VALUE
3  IMPORT MIN_VALUE
4
5  validate_value PROC
6   PUSH {LR}
7   LDR  R1, =MAX_VALUE
8   LDR  R1, [R1]          ; Load max value
9   CMP  R0, R1
10  BGT  invalid          ; Above max
11  LDR  R1, =MIN_VALUE
12  LDR  R1, [R1]          ; Load min value
13  CMP  R0, R1
14  BLT  invalid          ; Below min
15  MOVS R0, #1            ; Valid
16  POP  {PC}
17 invalid
18  MOVS R0, #0            ; Invalid
19  POP  {PC}
20 ENDP
21 END
```

Exceptional Control Flow

Exception Types Two main categories of exceptions:

Interrupt Sources:

- Peripherals signal to CPU that an event needs immediate attention
- Can alternatively be generated by software request
- Asynchronous to instruction execution

System Exceptions:

- **Reset:** Processor restart
- **NMI:** Non-maskable Interrupt (cannot be ignored)
- **Faults:** Undefined instructions, errors
- **System Calls:** OS calls - Instructions SVC and PendSV

Interrupt Control

PRIMASK register controls interrupt handling:

- Single bit controls all maskable interrupts
- Reset state: PRIMASK = 0 (interrupts enabled)
- Control methods:
 - Assembly: CPSID i (disable), CPSIE i (enable)
 - C: `_disable_irq()`, `_enable_irq()`

PRIMASK

- Single bit controlling all maskable interrupts



	Assembly	C
- Disable	<code>CPSID¹⁾ i</code>	<code>_disable_irq();</code>
- Enable	<code>CPSIE¹⁾ i</code>	<code>_enable_irq();</code>

On reset PRIMASK = 0 → enabled

Context Storage Interrupt handling requires automatic context saving

Timing: Interrupts can occur at any time (e.g. between instructions)

- CPU must save current state before handling
- Context includes registers, flags, and program counter

ISR call:

- requires automatic save off lags and caller saved registers
- Stores on stack:
 - xPSR, PC, LR, R12
 - R0-R3 (caller-saved registers)
- Stores EXC_RETURN in LR

ISR return:

- Use BX LR or POP ..., PC
- Loading EXC return into PC → restores from stack:
 - R0-R3, R12, LR, PC, xPSR

Basic ISR Implementation

```

1 ; Interrupt Service Routine
2 EXPORT MyISR
3 MyISR
4 PUSH {R4-R7, LR} ; Save registers
5
6 ; Handle interrupt here
7 ; R0-R3 already saved automatically
8
9 POP {R4-R7, PC} ; Restore and return

```

Interrupt Handling

Interrupt Basics

Key concepts for interrupt handling:

- **Interrupt Sources:**
 - Hardware interrupts from peripherals
 - Software interrupts (SVC)
 - System exceptions
- **Configuration Steps:**
 - Enable specific interrupt source
 - Install interrupt handler
 - Configure interrupt priority
 - Enable global interrupts
- **Handler Requirements:**
 - Predefined names from vector table
 - No return values allowed
 - Must clear interrupt flags
 - Save/restore used registers

Interrupt-Driven I/O

- Hardware-triggered event handling
- Asynchronous to main program

Main program:

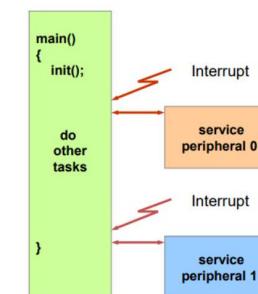
- Initializes peripherals, afterwards executes other tasks
- Peripherals signal when they require SW attention
- Events interrupt program execution

Advantages:

- Efficient CPU usage (no busy waiting)
- Quick response times
- Better system throughput

Disadvantages:

- No synchronization between main program and ISR
- More complex implementation and harder to debug
- Timing less predictable



Polling

Periodic query of status information

Main program:

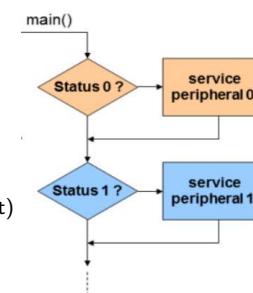
- Reading of status registers in loop
- Continues execution if no event
- Handles event if detected
- Synchronous with main program

Advantages:

- Simple and straightforward
- Implicit synchronization
- Predictable timing (deterministic)
- No additional interrupt logic required

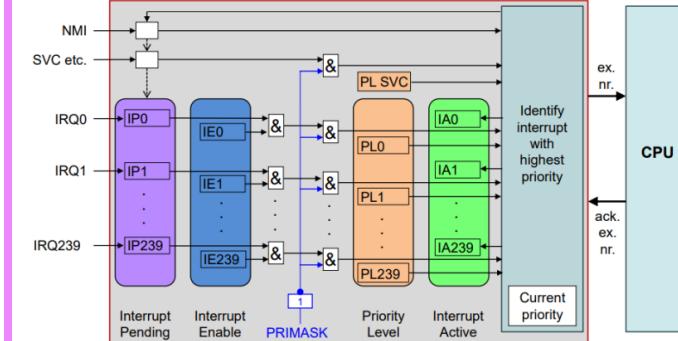
Disadvantages:

- CPU wastes time waiting (busy wait)
- Reduced system throughput
- Longer response times



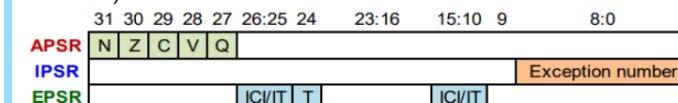
Interrupt Control

Interrupt Control



Program Status Registers (PSR)

- IPSR: Interrupt Program Status Register
- EPSR: Execution Program Status Register
- APSR: Application Program Status Register
- xPSR: Extended Program Status Register (combination of all three above)



Priority System Interrupt priority handling:

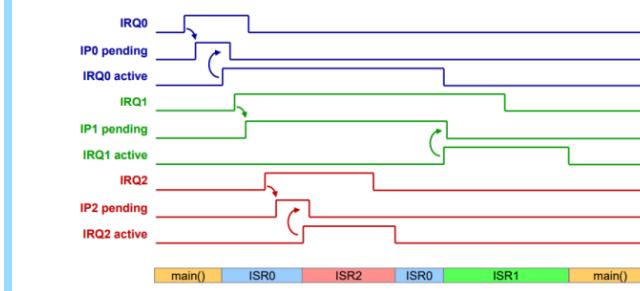
- **Priority Levels:**
 - 0-255 (lower number = higher priority)
 - Fixed priorities for system exceptions
 - Programmable priorities for IRQs
- **Preemption:**
 - Higher priority interrupts can preempt lower
 - Same priority follows FIFO

Nested Exceptions

Example Priorities

- ISR1 does not preempt ISR0
- ISR2 preempts ISR0

assuming		
IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



Nested Vectored Interrupt Controller (NVIC)

NVIC (Nested Vectored Interrupt Controller)

Interrupt/Exception States:

- Inactive:** Not active and not pending
- Pending:** Waiting to be serviced by CPU or: Interrupt event occurred ($\text{IRQn}=1$), but interrupts are disabled ($\text{PRIMASK}=1$)
- Active:** Currently being serviced by CPU but not yet completed
- Active and Pending:**
Being serviced with new request pending for same source

Control Registers:

- Interrupt Enable (IE)
- Interrupt Pending (IP)
- Interrupt Active (IA)
- Priority Level (PL)

Interrupt Control Registers

Important NVIC registers:

1. Enable/Disable Registers:

```

1 SETENA0 EQU 0xE000E100 ; Enable interrupts
2 CLRENA0 EQU 0xE000E180 ; Disable interrupts
3 ; Enable IRQ3
4 LDR R0, =SETENA0
5 MOVS R1, #(1<<3)
6 STR R1, [R0]
7 ; Disable IRQ3
8 LDR R0, =CLRENA0
9 MOVS R1, #(1<<3)
10 STR R1, [R0]

```

2. Pending Registers:

```

1 SETPENDO EQU 0xE000E200 ; Set pending
2 CLRPENDO EQU 0xE000E280 ; Clear pending
3 ; Set IRQ3 pending
4 LDR R0, =SETPENDO
5 MOVS R1, #(1<<3)
6 STR R1, [R0]
7 ; Clear IRQ3 pending
8 LDR R0, =CLRPENDO
9 MOVS R1, #(1<<3)
10 STR R1, [R0]

```

Exception Vector Table

Setup and usage:

Vector table structure and Handler implementation:

```

1 AREA RESET, DATA, READONLY
2 --Vectors
3 DCD __initial_sp ; Top of Stack
4 DCD Reset_Handler ; Reset
5 DCD NMI_Handler ; NMI
6 DCD HardFault_Handler ; Hard Fault
7 DCD 0 ; Reserved
8 DCD 0 ; Reserved
9 ; ... more vectors
10 DCD IRQ0_Handler ; IRQ0
11 DCD IRQ1_Handler ; IRQ1

```

```

1 AREA .text!, CODE, READONLY
2
3 IRQ0_Handler PROC
4 EXPORT IRQ0_Handler
5 PUSH {R4-R7,LR}
6 ; Handle interrupt
7 POP {R4-R7,PC}
8 ENDP

```

NVIC Configuration

Complete interrupt system setup:

```

1 void init_timer_interrupt(void) {
2     // Configure timer peripheral
3     TIM2->PSC = 7199;           // Prescaler
4     TIM2->ARR = 9999;          // Auto-reload value
5     TIM2->DIER |= 1;           // Enable interrupt
6     TIM2->CR1 |= 1;            // Enable timer
7
8     // Configure NVIC
9     NVIC_SetPriority(TIM2_IRQn, 2); // Set priority
10    NVIC_EnableIRQ(TIM2_IRQn);      // Enable IRQ
11
12    __enable_irq();               // Enable global
13        interrupts
14
15 // Timer 2 interrupt handler
16 void TIM2_IRQHandler(void) {
17     if (TIM2->SR & 1) {          // Check update flag
18         // Handle interrupt
19         TIM2->SR &= ~1;          // Clear flag
20     }
21 }

```

Assembly equivalent:

```

1 TIM2_BASE EQU 0x40000000
2 TIM_SR EQU 0x10
3 TIM_DIER EQU 0x0C
4 SETENA0 EQU 0xE000E100
5
6 ; Enable timer interrupt
7 LDR R0, =TIM2_BASE
8 LDR R1, [R0, #TIM_DIER]
9 ORRS R1, #1
10 STR R1, [R0, #TIM_DIER]
11
12 ; Enable in NVIC
13 LDR R0, =SETENA0
14 MOVS R1, #1
15 LSLS R1, #28 ; IRQ28
16 STR R1, [R0]

```

Important considerations:

- Always clear interrupt flags
- Minimize time in interrupt handlers
- Protect shared data access
- Consider interrupt priorities
- Avoid deadlocks with nested interrupts

Data Consistency

Data Consistency

Handling shared data access:

- Race Conditions:**
 - Main program and ISR accessing same data
 - Interrupts during multi-step operations
- Solutions:**
 - Disable interrupts during critical sections
 - Use atomic operations
 - Implement proper synchronization

Data Consistency Protection

Protecting shared data access:

```

1 // Global time counters accessed by ISR
2 volatile uint32_t minutes = 0;
3 volatile uint32_t hours = 0;
4
5 // Timer ISR updates counters
6 void TIM2_IRQHandler(void) {
7     minutes++;
8     if (minutes >= 60) {
9         minutes = 0;
10        hours++;
11    }
12    // Clear interrupt flag
13 }
14
15 // Main code reading counters
16 void read_time(uint32_t *min, uint32_t *hr) {
17     // Disable interrupts to read consistent values
18     __disable_irq();
19     *min = minutes;
20     *hr = hours;
21     __enable_irq();
22 }

```

Assembly equivalent:

```

1 read_time
2     PUSH {R4-R5, LR}
3     CPSID i ; Disable interrupts
4
5     LDR R4, =minutes ; Load minutes
6     LDR R2, [R4]
7     STR R2, [R0] ; Store to output
8
9     LDR R5, =hours ; Load hours
10    LDR R3, [R5]
11    STR R3, [R1] ; Store to output
12
13    CPSIE i ; Enable interrupts
14    POP {R4-R5, PC}

```

Implementing Interrupt Handlers

Implementing Interrupt Handlers

1. Define interrupt vector
2. Save necessary context
3. Handle the interrupt
4. Clear interrupt flag
5. Restore context
6. Return from interrupt

- Important considerations:
- Keep ISRs short
 - Handle critical tasks only
 - Be aware of nested interrupts
 - Protect shared resources

Interrupt Handler Implementation

Guidelines for implementing interrupt handlers:

1. Handler structure:

```

1  handler_name
2  PUSH   {R4-R6}          ; Save used registers
3
4  ; Check interrupt flags
5  ; Handle interrupt condition
6  ; Clear interrupt flags
7
8  POP    {R4-R6}          ; Restore registers
9  BX     LR               ; Return from handler

```

2. Register preservation:

- R0-R3: Automatically saved by hardware
- R4-R11: Must be preserved if used
- R12: Can be used freely
- LR: Contains special EXC_RETURN value

3. Critical section handling:

```

1  ; Disable all interrupts
2  CPSID  i
3
4  ; Critical section code
5  ; Access shared resources
6
7  ; Enable interrupts
8  CPSIE  i

```

Timer Interrupt Configuration Configuring Timer 2 interrupt:

1. Handler definition:

```

1  AREA   handlers, CODE, READONLY
2
3  EXPORT  TIM2_IRQHandler
4  TIM2_IRQHandler
5  PUSH   {R4-R6}          ; Save registers
6  ; Handle interrupt
7  POP    {R4-R6}          ; Restore registers
8  BX     LR               ; Return

```

2. Enable interrupt (IRQ28):

```

1  AREA   startup, CODE, READONLY
2
3  SETENA0 EQU      0xE000E100 ; Interrupt enable
4  register
5
6  ; Enable Timer 2 interrupt
7  LDR    R7, =SETENA0
8  MOVS  R6, #1           ; Set bit
9  LSLS  R6, #28          ; Shift to IRQ28
10  STR   R6, [R7]         ; Enable interrupt

```

Exception Handling in C

CMSIS Functions for Interrupt Control

Standard CMSIS functions for interrupt handling:

- NVIC_EnableIRQ(IRQn): Enable specific interrupt
- NVIC_DisableIRQ(IRQn): Disable specific interrupt
- NVIC_SetPendingIRQ(IRQn): Set interrupt pending
- NVIC_ClearPendingIRQ(IRQn): Clear pending status
- NVIC_SetPriority(IRQn, priority): Set priority
- NVIC_GetPriority(IRQn): Read priority

Example usage:

```

1 void init_timer_interrupt(void) {
2     // Enable timer interrupt
3     NVIC_EnableIRQ(TIM2_IRQn);
4
5     // Set priority
6     NVIC_SetPriority(TIM2_IRQn, 2);
7
8     // Configure timer
9     // ...
10
11    // Enable global interrupts
12    __enable_irq();
13 }

```

Data Consistency Example protection:

```

1 void update_shared_data(void) {
2     __disable_irq();           // Critical section start
3     shared_var++;             // Update shared data
4     __enable_irq();           // Critical section end
5 }

```

Nested Interrupts Example Implementation with different priorities:

```

1 // Initialize interrupts
2 void init_interrupts(void) {
3     // Enable interrupts
4     NVIC_EnableIRQ(IRQ0_IRQn);
5     NVIC_EnableIRQ(IRQ1_IRQn);
6
7     // Set priorities
8     NVIC_SetPriority(IRQ0_IRQn, 1); // Higher
9     NVIC_SetPriority(IRQ1_IRQn, 2); // Lower
10
11    // Enable global interrupts
12    __enable_irq();
13 }
14
15 // Higher priority ISR
16 void IRQ0_Handler(void) {
17     // Handle high priority interrupt
18     // Can't be interrupted by IRQ1
19 }
20
21 // Lower priority ISR
22 void IRQ1_Handler(void) {
23     // Handle low priority interrupt
24     // Can be interrupted by IRQ0
25 }

```

Examples

Kurzfristiges Aus- und Einschalten von Interrupts

Enabling of Interrupt Quellen, wie in Aufgabe 1 behandelt, dient zur initialen Konfiguration von Interrupts. Im Betrieb ist es aber unter Umständen nötig alle Interrupts kurzfristig auszuschalten und danach wieder einzuschalten.

1. Was kann ein Grund sein für ein solches kurzfristiges Aus- und wieder Einschalten?

Data Konsistenz: z.B. eine Interrupt Service Routine eines Timers unterhält zwei Zähler, einen für Minuten und einen für Stunden. Beim Übergang von Minute 59 zu 0 wird der Stundenzähler um eins erhöht. Eine andere Routine welche diese beiden Zähler ausliest muss dafür sorgen dass kein Interrupt passiert zwischen dem Zugriff auf diese beiden Zähler.

2. Wie schalten Sie alle Interrupts in Assembler aus? Wie ein?

Ausschalten:

CPSID i

Einschalten:

CPSIE i

3. Wie schalten Sie alle Interrupts in C aus? Wie ein?

Ausschalten:

`_disable_irq();`

Einschalten:

`_enable_irq();`

ISR (Interrupt Service Routines)

Der ARM Prozessor rettet beim Abarbeiten eines Interrupts gewisse Register auf den Stack bevor die ISR (Interrupt Service Routine) ausgeführt wird - und restauriert diese nach Beendigung der ISR automatisch.

1. Wenn Sie in Ihrer ISR die Register R0-R6 verwenden, welche dieser Register müssen Sie auf den Stack pushen weil sie nicht schon automatisch vorher gerettet wurden?

PUSH {R4-R6} ; R0-R3 wurden schon automatisch gerettet

2. Wie Unterscheidet sich in der Programmierung eine ISR von einer „normalen“Funktion?

Jede ISR hat einen vordefinierten Namen (von der Vectors Tabelle vorgegeben).

Eine ISR kann keine Werte zurückgeben (ist in C eine void iss_name(void) Funktion).

Eine ISR muss gegebenenfalls den Interrupt zurücksetzen so dass er nicht permanent feuert.

Increasing System Performance

Performance Optimization Trade-offs

Optimizing for	Drawbacks on
Higher speed	Power, cost, chip area
Lower cost	Speed, reliability
Zero power consumption	Speed, cost
Super reliable	Chip area, cost, speed
Temperature range	Power, cost, lifetime

Instruction Set Architectures

RISC (Reduced Instruction Set Computer):

- Few instructions with uniform format
- Fast decoding, simple addressing
- Less hardware → higher clock rates
- More chip space for registers (up to 256)
- Load-store architecture reduces memory access
- CPU works at full speed on registers
- Enables shorter, efficient pipelines (instruction size/duration)

CISC (Complex Instruction Set Computer):

- More complex and more instructions
- Lower memory usage for complex programs
- Potential performance gain for short programs
- More complex hardware required

Example: Balance = Balance + Credit

```
LDR R0, =Credit
LDR R1, [R0]
LDR R0, =Balance
LDR R3, [R0]
ADDS R2, R1, R3
STR R2, [R0]
```

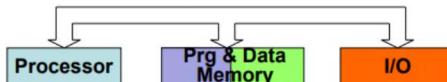
RISC

- Load / Store Architecture
- Data processing instructions only available on registers

```
MOV AX, [Credit]
ADD [Balance], AX
```

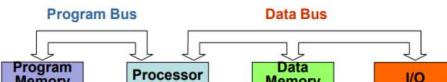
Von Neumann Architectures

- Single memory for program and data
- Single bus system between CPU and memory

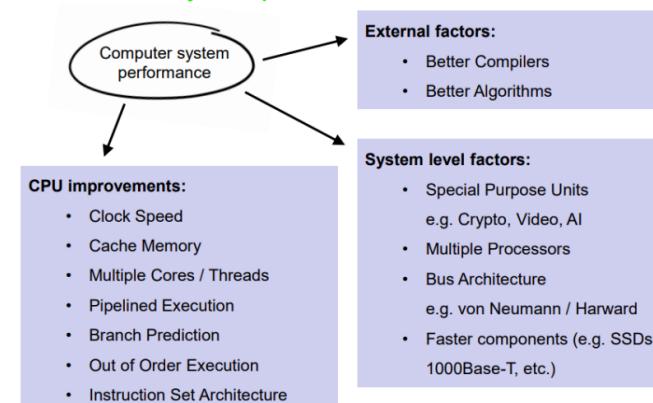


Harvard Architecture:

- Separate program and data memories
- Two sets of address/data buses
- Originally from Harvard Mark I



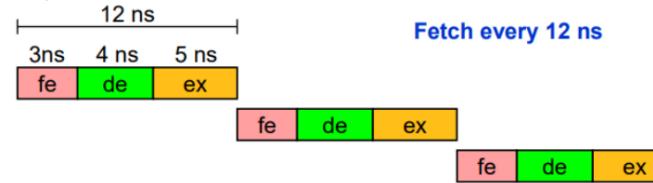
How to increase system speed?



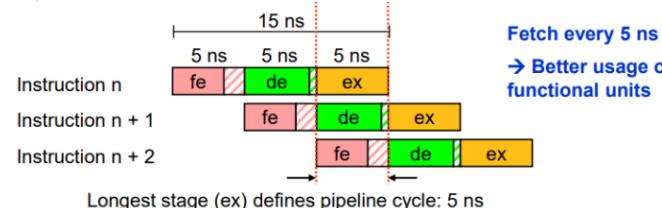
Pipelineing

Sequential vs. Pipelined Execution

Sequential Execution:

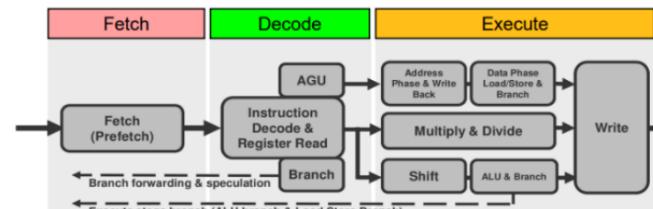


Pipelined Execution:



Pipelineing

Process of fetching next instruction while current one decodes:



Advantages:

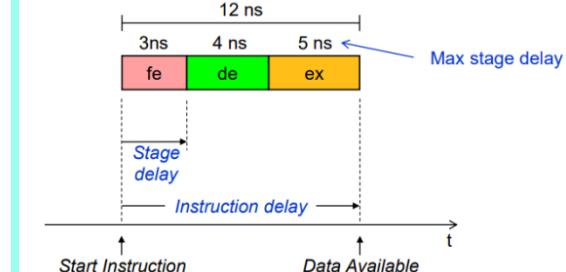
- Uniform execution time per stage
- Significant performance improvement
- Simpler hardware per stage allows higher clock rates

Disadvantages:

- Blocking stages affect whole pipeline
- Possible Memory access conflicts between stages

Pipeline Stages (Example)

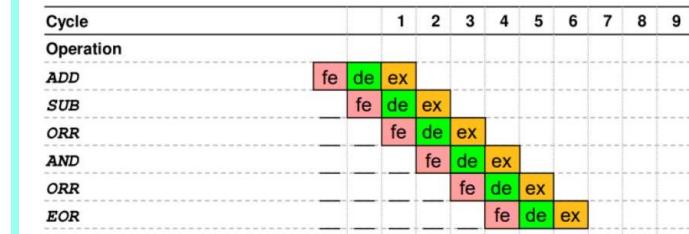
- Fetch (Fe): Read instruction - 3ns
- Decode (De): Process instruction - 4ns
- Execute (Ex): Execute and writeback - 5ns



Pipeline Execution

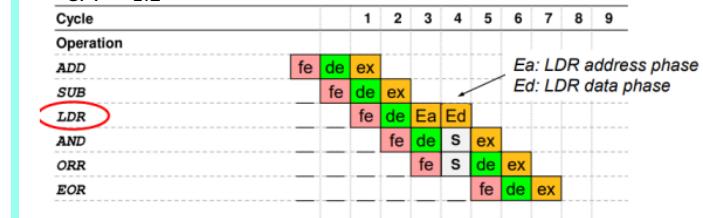
Optimal Case:

- Register-only operations
- 6 instructions in 6 cycles
- CPI = 1 (Cycles Per Instruction)



LDR Special Case:

- 6 instructions in 7 cycles due to memory access
- Pipeline stalls for memory read
- CPI = 1.2



Pipeline important definitions

- CPI: Cycles per instruction
- IPC: Instructions per cycle
- Fe, De, Ex: Stage delays
- Cycles: Number of cycles for instruction
- Performance: $\frac{1}{\text{Total delay}}$
- Throughput: $\frac{1}{\text{Max stage delay}}$
- Initial latency: Number of cycles until pipeline is filled
- Performance improvement: $\frac{\text{Without pipeline delay}}{\text{With pipeline delay}}$
- Pipeline stalls: Delay due to memory access

Pipeline Performance Calculation

For a processor with n pipeline stages:

Without pipelining:

- Time per instruction = Sum of all stage delays
- Performance (Instructions/Second) =

$$\frac{1}{\text{Total delay}}$$

With pipelining:

- Time per instruction = Longest stage delay
- Initial latency = n cycles
- Throughput (Instructions/Second) =

$$\frac{1}{\text{Max stage delay}}$$

Note: Pipeline must be filled first! After filling, instructions are executed after every stage.

Analyzing Pipeline Performance

Steps for calculating pipeline performance:

1. Calculate performance without pipelining:

- Total delay = Sum of all stage delays
- Performance = $\frac{1}{\text{Total delay}}$

Example for 3 stages:

- Fetch: 3ns, Decode: 4ns, Execute: 5ns
- Total delay = 12ns
- Performance = 83.3 MIPS

2. Calculate performance with pipelining:

- Critical path = Longest stage delay
- Initial latency = Number of stages
- Throughput = $\frac{1}{\text{Max stage delay}}$

Example for 3 stages:

- Max stage delay = 5ns (Execute)
- Initial latency = 3 cycles
- Throughput = 200 MIPS

3. Calculate performance improvement:

$$\text{Improvement} = \frac{\text{Pipelined throughput}}{\text{Non-pipelined throughput}}$$

Example:

$$\frac{200}{83.3} = 2.4 \times \text{speedup}$$

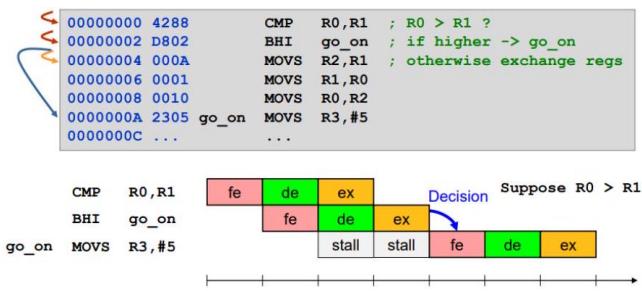
Pipeline Performance Calculation

- Stage delays: Fe=3ns, De=4ns, Ex=5ns
- Without pipeline: 12ns per instruction
- With pipeline: 5ns per instruction after filling
- Performance improvement: 2.4×

Pipeline Hazards and Optimization

Control Hazards:

- Branch/jump decisions in execute stage (stage 3)
- Worst case scenario: conditional branch taken
- Pipeline stalls for taken branches



Reduce control hazards: Use loop fusion!

Pipeline Hazard Analysis

1. Structural Hazard:

```
1 ; Multiple instructions need memory access
2 LDR R0, [R1] ; Load from memory
3 LDR R2, [R3] ; Load from memory - must wait
4 ; Pipeline stalls because memory system can't handle both loads
```

2. Data Hazard:

```
1 ADDS R0, R1, R2 ; R0 gets new value
2 ADDS R3, R0, R4 ; Uses R0 before ready
3 ; Second instruction must wait for first to complete
```

3. Control Hazard:

```
1 CMP R0, #0 ; Compare
2 BEQ target ; Branch if equal
3 ADD R1, R2, R3 ; May be unnecessary
4 SUB R4, R5, R6 ; May be unnecessary
5 target
6 ; Pipeline must flush if branch taken
```

Pipeline Optimization

Optimization Techniques:

- Branch prediction based on history
 - Store last decisions made for each conditional branch
 - probability is high that the same decision will be made again
- Instruction prefetch
 - Fetch several instructions in advance
 - reduces pipeline stalls
 - better use of system bus
 - possibility of "Out-of-order execution"
- Out-of-order execution
 - If one instruction stalls, it might be possible to already execute next instruction
 - requires complex hardware

Optimization Limits:

- Complex optimizations → severe security problems
- Instructions executed, that would throw access violations under "In Order" circumstances
- "Meltdown" and "Spectre" attacks: allow process to access memory of other processes

Loop Optimization Techniques

Steps for optimizing loops:

1. Loop Fusion:

```
// Before optimization - two loops
for (i = 0; i < N; i++) {
    a[i] = b[i] * c[i];
}
for (i = 0; i < N; i++) {
    d[i] = a[i] + c[i];
}

// After fusion - single loop
for (i = 0; i < N; i++) {
    a[i] = b[i] * c[i];
    d[i] = a[i] + c[i];
}
```

2. Loop Unrolling:

```
// Before unrolling
for (i = 0; i < N; i++) {
    sum += array[i];
}

// After unrolling by factor 4
for (i = 0; i < N-3; i += 4) {
    sum += array[i];
    sum += array[i+1];
    sum += array[i+2];
    sum += array[i+3];
}
// Handle remaining elements
for (i = N-3; i < N; i++) {
    sum += array[i];
}
```

3. Memory Access Optimization:

- Align data to cache line boundaries
- Use sequential memory access patterns
- Minimize cache misses

Parallel Computing

Parallel Computing

Different approaches to parallelism:

- **Streaming/Vector Processing:** Single instruction processes multiple data items simultaneously
- **Multithreading:** Multiple programs/threads share a single CPU
- **Multicore:** One processor with multiple CPU cores
- **Multiprocessor:** A computer system containing multiple processors

Multicore vs Multiprocessor Key differences:

- **Multicore:**
 - Multiple CPU cores on single chip
 - Shared cache and memory interface
 - Lower communication overhead
 - More power efficient
- **Multiprocessor:**
 - Multiple separate CPU chips
 - Independent caches
 - Higher communication overhead
 - More scalable for large systems

Multi-Core Programming Example of parallel task processing:

```
1 typedef struct {
2     int *array;
3     int start;
4     int end;
5     long sum;
6 } task_t;
7
8 void* partial_sum(void* arg) {
9     task_t* task = (task_t*)arg;
10    long sum = 0;
11
12    for (int i = task->start; i < task->end; i++) {
13        sum += task->array[i];
14    }
15    task->sum = sum;
16    return NULL;
17 }
18 // Split work across multiple cores
19 void parallel_sum(int* array, int size, int num_cores)
20 {
21     task_t tasks[MAX_CORES];
22     int chunk = size / num_cores;
23     // Create tasks for each core
24     for (int i = 0; i < num_cores; i++) {
25         tasks[i].array = array;
26         tasks[i].start = i * chunk;
27         tasks[i].end = (i == num_cores-1) ?
28                         size : (i+1) * chunk;
29         // Launch task on core i
30         launch_on_core(i, partial_sum, &tasks[i]);
31     }
32     // Wait for all cores and combine results
33     long total = 0;
34     for (int i = 0; i < num_cores; i++) {
35         wait_for_core(i);
36         total += tasks[i].sum;
37     }
38     return total;
}
```

Parallel Processing Models

SISD (Single Instruction Single Data):

- Traditional sequential processing
- One instruction processes one data item
- Example: Basic scalar processor

SIMD (Single Instruction Multiple Data):

- Vector processing
- One instruction processes multiple data items
- Examples: MMX, SSE, AVX instructions

MIMD (Multiple Instruction Multiple Data):

- True parallel processing
- Multiple processors execute different instructions
- Example: Multicore systems

SIMD Operations

Example of vectorized operations:

```
1 // Scalar addition
2 void add_arrays_scalar(int *a, int *b, int *c, int n) {
3     for (int i = 0; i < n; i++) {
4         c[i] = a[i] + b[i];
5     }
6 }
7
8 // SIMD/Vector addition (conceptual)
9 void add_arrays_vector(int *a, int *b, int *c, int n) {
10    for (int i = 0; i < n; i += 4) {
11        // Load 4 elements at once
12        vector_t va = vector_load(&a[i]);
13        vector_t vb = vector_load(&b[i]);
14        // Add 4 pairs of elements simultaneously
15        vector_t vc = vector_add(va, vb);
16        // Store 4 results at once
17        vector_store(&c[i], vc);
18    }
19 }
```

Conclusion on Performance Optimization

Performance Growth Overview

Historical development:

- Early improvements:
 - Increasing clock frequencies
 - Better manufacturing processes
 - Smaller transistor sizes
- Modern improvements:
 - Advanced architectural concepts (RISC, Pipelining)
 - Multiple cores
 - Specialized hardware units
- Current limitations:
 - Power density
 - Heat dissipation
 - Memory wall
 - Parallelization overhead

Optimizing System Performance

Steps for performance optimization:

1. Analyze performance bottlenecks
2. Choose appropriate architecture:
 - RISC vs CISC based on application
 - Consider memory architecture
3. Implement pipelining:
 - Balance stage delays
 - Handle hazards appropriately
4. Consider parallelization options
5. Evaluate security implications

System Level Optimization

Different approaches to improve performance:

- **External Factors:**
 - Better compiler optimization
 - Improved algorithms
 - Efficient software design
- **System Level Factors:**
 - Special Purpose Units (e.g., Crypto, Video)
 - Multiple Processors
 - Bus Architecture optimization
 - Faster peripheral components
- **CPU Improvements:**
 - Increased Clock Speed
 - Cache Memory
 - Multiple Cores
 - Pipeline Optimization
 - Branch Prediction
 - Out-of-Order Execution

Performance Optimization Guidelines

Steps for system optimization:

1. **Analyze Requirements:**
 - Performance targets
 - Power constraints
 - Cost limitations
 - Reliability needs
2. **Choose Architecture:**
 - RISC vs CISC
 - Memory architecture
 - Pipeline depth
 - Parallelization approach
3. **Optimize Implementation:**
 - Balance pipeline stages
 - Implement hazard handling
 - Consider branch prediction
 - Optimize memory access
4. **Security Considerations:**
 - Evaluate optimization risks
 - Consider side-channel attacks
 - Balance performance and security

Coding Guidelines

Guidelines Purpose

Coding guidelines are essential for:

- Reducing the number of bugs through:
 - Improved robustness
 - Better correctness
 - Enhanced maintainability
- Facilitating code reading within a team
- Improving portability for code reuse

Code Structure and Organization

Key guidelines for code organization:

1. Code Appearance:

- Indentation: 4 spaces, no tabs
- Maximum of 80 characters per line
- No more than one statement per line
- Use parentheses to aid clarity (don't rely on operator precedence)

2. Braces Usage:

- Non-function statement blocks (if, else, switch, for, while, do):
 - Opening brace last on line
 - Closing brace first on line
- Functions:
 - Opening brace beginning of next line
 - Closing brace first on line
- Always use braces for single statements and empty statements

Function Requirements

Guidelines for function implementation:

- Keep functions short and focused (max 50 lines)
- Do just one thing
- No more than 5-10 local variables
- No more than 3 parameters
- Function prototypes must include parameter names with data types
- Maximum of 3 levels of indentation
- Single exit point at bottom of function
- Use const for call-by-reference parameters that shouldn't be modified

Return Value Conventions

Guidelines for function return values:

- Return values must be checked by the caller
- For functions named as actions/commands:
 - Return error-code integer (0 success, -Exxx failure)
 - Use Posix error codes where possible
 - Document custom error codes in header files
- For predicate functions:
 - Return boolean success value
 - Example: pci_dev_present() returns 1 for success, 0 for failure
- Computation functions return actual results
 - Indicate failure through out-of-range values
 - Use NULL or ERR_PTR for pointer returns

Naming Conventions

Rules for naming different code elements:

- Macro names (#define): All uppercase letters
- Function/variable names: No uppercase letters
- Use descriptive names for functions, globals, and important locals
- Use underscores to separate words (e.g., count_active_users())
- Short names acceptable for auxiliary locals (e.g., i for loop counters)
- Don't encode types in names - let compiler do type checking

Comments and Documentation

Guidelines for code documentation:

- All comments must be in English
- C99 comments (//) are allowed
- Explain WHAT code does, not HOW
- Don't repeat what the statement says in comments
- Comments shall never be nested
- Document all assumptions in comments
- Interface documentation in header files only
- Comment function prototypes in header, not in source file

Type Usage

Rules for data type usage:

- Use fixed-width C99 types from stdint.h
 - uint8_t/int32_t instead of unsigned char/int
- Restrict char type to string operations
- No bit-fields within signed integer types
- Don't use bitwise operators on signed integers
- Don't mix signed/unsigned in comparisons
- Use 'U' suffix for unsigned decimal constants
- One data declaration per line for clarity

Header File Organization

Requirements for header files:

- One header file per module
- Include preprocessor guards against multiple inclusion
- Only add immediately needed #includes
- No variable definitions/declarations
- Keep interface minimal - only expose necessary functions
- Document all public functions in header
- Private functions must be declared static
- Function prototypes in module interface

Module Design

Example of proper module organization:

1. Header File (module.h):

```
1 #ifndef _MODULE_H
2 #define _MODULE_H
3
4 typedef enum {
5     STATE_A = 0x00,
6     STATE_B = 0x01
7 } module_state_t;
8
9 void module_init(void);
10 void module_set_state(module_state_t state);
11 module_state_t module_get_state(void);
12
13 #endif /* _MODULE_H */
```

2. Implementation File (module.c):

```
1 #include "module.h"
2
3 static module_state_t current_state;
4 static void helper_function(void);
5
6 void module_init(void) {
7     current_state = STATE_A;
8 }
9
10 void module_set_state(module_state_t state) {
11     current_state = state;
12     helper_function();
13 }
14
15 module_state_t module_get_state(void) {
16     return current_state;
17 }
18
19 static void helper_function(void) {
20     // Implementation
21 }
```

Important considerations:

- Coding guidelines are subjective
- Different organizations have different standards
- Follow existing project conventions
- Guidelines help maintain consistency
- Use automated checks and peer reviews
- Document any deviations from standards