Computertechnik

Jil Zerndt, Lucien Perret
January 2025

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

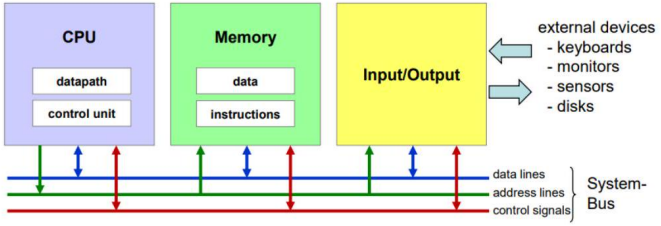
von Neumann Architecture

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

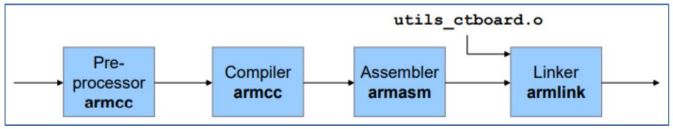
Key concepts for working with memory

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

```c
// source.c
#include <stdio.h>
#define MAX 100

int main(void) {
    printf("Max is %d\n", MAX);
    return 0;
}
```

After preprocessing (.i):

```c
// 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
```

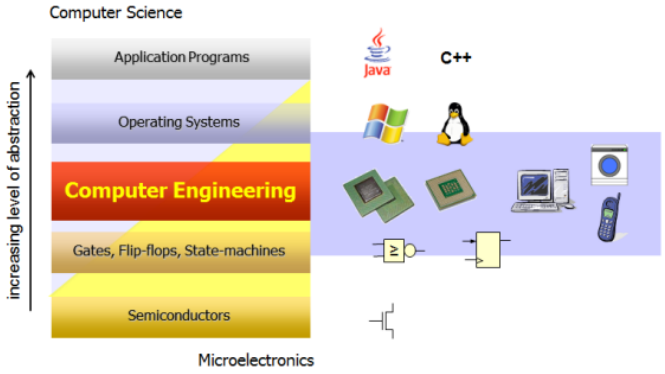
Host vs Target Development

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

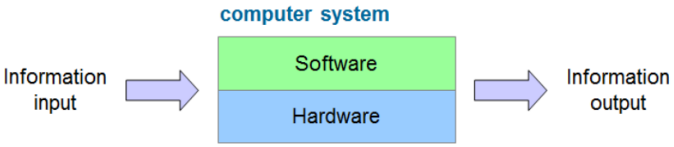
Examples

Was ist Computertechnik?



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:

```c
// Example showing interaction between components
int main(void) {
    int data;              // Uses Memory
    scanf("%d", &data);    // Uses I/O
    data = data * 2;       // Uses CPU/ALU
    printf("%d", data);    // Uses I/O
    return 0;
}
```

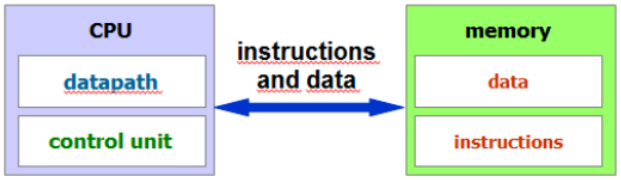
Steuereinheit einer CPU

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):

```c
#include <stdio.h>
#define MAX 100

int main(void) {
    printf("Max is %d\n", MAX);
    return 0;
}
```

2. After preprocessing (.i):

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

3. Assembly output (.s):

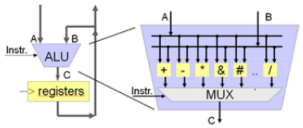
```
    AREA |.text|, CODE, READONLY
    EXPORT main
main
    PUSH    {LR}
    LDR     R0, =string1
    MOV     R1, #100
    BL      printf
    MOVS    R0, #0
    POP     {PC}
string1
    DCB     "Max is %d\n",0
    ALIGN
```

Operationstypen CPU

Welche Operationstypen werden im Allgemeinen von einer CPU unterstützt?

Datapath

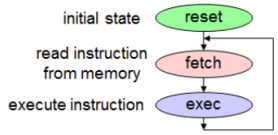
- ALU: Arithmetic and Logic Unit
  - performs arithmetic/logic operations



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

Control Unit

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



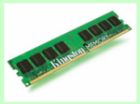
- types of operations
  - data transfer: registers ←→ memory
  - arithmetic and logic operations
  - jumps

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:

```
; RAM access example
    LDR     R0, =ram_data     ; Load RAM address
    LDR     R1, [R0]          ; Read from RAM

; ROM access example
    LDR     R0, =rom_const    ; Load ROM address
    LDR     R1, [R0]          ; Read from ROM

; Flash memory programming
    LDR     R0, =FLASH_BASE   ; Flash memory base
    LDR     R1, =0x12345678   ; Data to write
    ; Flash write sequence would go here

section_data
ram_data    SPACE   4         ; RAM variable
rom_const   DCD     0xFF      ; ROM constant
```

Computer System Debugging

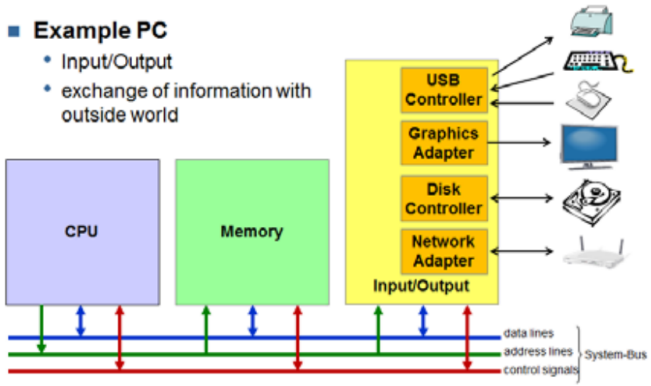
Steps for debugging computer system issues:

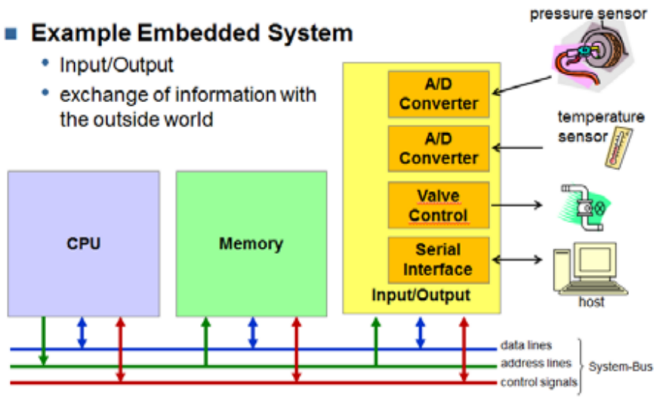
1. Hardware level:
   - Check power and connections
   - Verify clock signals
   - Test memory access
   - Check I/O interfaces
2. Software level:

```
    ; Debug example
    PUSH    {R0-R3, LR}       ; Save registers
    BL      print_debug       ; Call debug routine
    ; Check specific values
    LDR     R0, =debug_var
    LDR     R1, [R0]
    CMP     R1, #expected
    BNE     error_handler
    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.





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

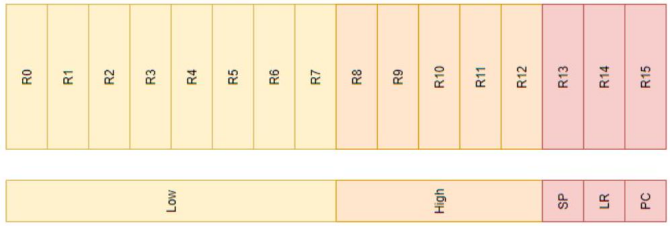
Core Architecture Overview

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)**: **S**tack **P**ointer - temporary storage
  - Points to current stack position
  - Must be word-aligned (multiple of 4)
- **R14 (LR)**: **L**ink **R**egister - return address from procedures
  - Stores return address for function calls
  - Can be saved to stack for nested calls
- **R15 (PC)**: **P**rogram **C**ounter - 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:

```
MOVS    R0, #0xFF     ; R0 = 255 (max unsigned 8-bit)
ADDS    R0, #1        ; R0 = 0, Z=1, C=1 (overflow)

MOVS    R0, #0x7F     ; R0 = 127 (max signed 8-bit)
ADDS    R0, #1 ; R0 = 128, N=1, V=1 (signed overflow)

MOVS    R0, #5
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:

```
; Register operations
ADDS    R0, R1, R2    ; R0 = R1 + R2
MOVS    R0, R1        ; R0 = R1
ANDS    R0, R1        ; R0 = R0 & R1
; Immediate values
MOVS    R0, #100      ; Load immediate value
ADDS    R0, R0, #1    ; Add immediate
CMP     R0, #10       ; Compare with immediate
; Memory access
LDR     R0, [R1]      ; Load from memory
STR     R0, [R1, #4]  ; Store with offset
LDRB    R0, [R1]      ; Load byte
```

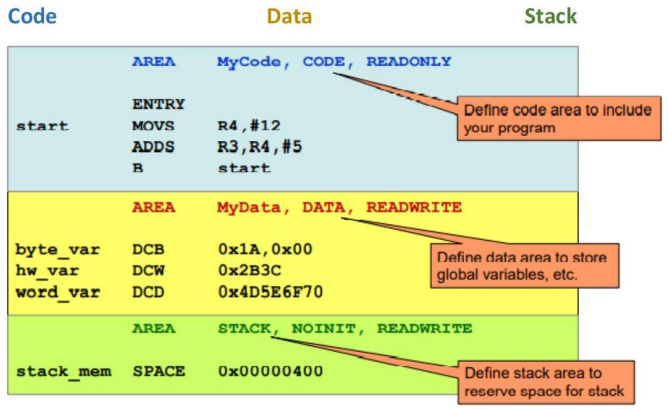
Basic Assembly Program Structure

Example of a simple assembly program:

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

Assembly Program Sections

Program memory organized in sections:



Code Section (CODE):

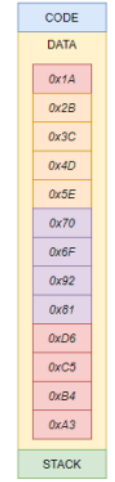
- Contains program instructions
- Usually read-only
- Placed in Flash memory
- Can contain constants (literal pool)

Data Section (DATA):

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

Stack Section: (STACK)

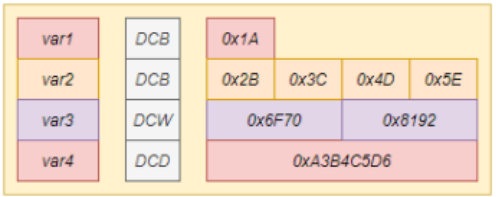
- Dynamic memory allocation
- Used for local variables
- Function call management
- Grows downward in memory



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:

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

Creating Assembly Programs Steps for creating an assembly program:

1. Define program sections: (CODE, DATA)

```
    AREA    |.text|, CODE, READONLY
    AREA    |.data|, DATA, READWRITE
```

2. Declare external symbols: (IMPORT/EXPORT)

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

3. Define data:
   - Define initialized data using DCx directives
   - Reserve uninitialized data using SPACE

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

4. Write program code using proper instruction syntax:

```
    AREA    |.text|, CODE, READONLY
    ENTRY                   ; Program entry
main
    ; Your code here
    END
```

5. End program with END directive!

Common Assembly Patterns

1. Loop with counter:

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

2. Memory copy:

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

3. Function call with parameters:

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

Complete Program Example

Program to sum array elements:

```
    AREA    |.text|, CODE, READONLY
    EXPORT  array_sum

array_sum
    MOVS    R2, #0          ; Initialize sum
    MOVS    R3, #0          ; Initialize index
loop
    LDR     R1, [R0, R3]    ; Load array element
    ADDS    R2, R2, R1      ; Add to sum
    ADDS    R3, R3, #4      ; Next element
    CMP     R3, #16         ; Check if done
    BLT     loop            ; Continue if not
    MOVS    R0, R2          ; Return sum
    BX      LR              ; Return
    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:

```
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?

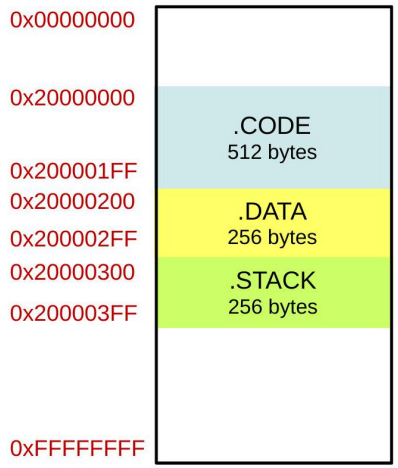
- **CODE** (read-only → RAM or ROM):
  - Machine instructions
  - Constants
- **DATA** (read-write → RAM):
  - Global variables
  - Static variables
  - Heap in C
- **STACK** (read-write → RAM):
  - Procedure calls
  - Passing of parameters
  - Local variables

Memory Areas/Memory Map

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

- Code: 0×20000000 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:

```
AREA    |.text|, CODE, READONLY
; Code section starts here

AREA    |.data|, DATA, READWRITE
; Data section starts here

; Stack section typically defined in startup code
Stack_Size     EQU     0x400    ; 1KB
AREA    STACK, NOINIT, READWRITE, ALIGN=3
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 Little Endian representation:

Address	Byte content	Variable
0x2FFF´FFF8	0xE2	Var4 (16-bit short)
0x2FFF´FFF9	01100010 (binary)	
0x2FFF´FFFC	0x65	Var3 (32-bit unsigned)
0x2FFF´FFFD	10101101	
0x2FFF´FFFE	0xA3	
0x2FFF´FFFF	0x82	

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)
$0 \times 3000'0005$	0x03	Var1 (char)

Var1 $= 0 \times 03 = 3d$ (it is 8-bit unsigned)
Var2 $= 0 \times 92 = 146$ d (it is 8 -bit unsigned)
Var3 $= 0 \times 82A3AD65 = +2'191'764$ '837d (it is 32-bit unsigned!!)
Var4 $= 0 \times 62E2 = +25$ '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	0×92	Var2 (unsigned char)
0x3000'0005	0×03	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

```
; MOV examples
MOVS    R1, #0xFF        ; Load immediate 255
MOVS    R2, R1           ; Copy R1 to R2
; LDR examples
LDR     R1, =0x12345678  ; Load 32-bit constant
LDR     R2, [R1]         ; Load from address in R1
LDR     R3, [R1, #4]     ; Load with offset
LDR     R4, [R1, R2]     ; Load with register offset
; Byte/Half-word loads
LDRB    R1, [R2]         ; Load unsigned byte
LDRSB   R1, [R2]         ; Load signed byte
LDRH    R1, [R2]         ; Load unsigned half-word
LDRSH   R1, [R2]         ; Load signed half-word
```

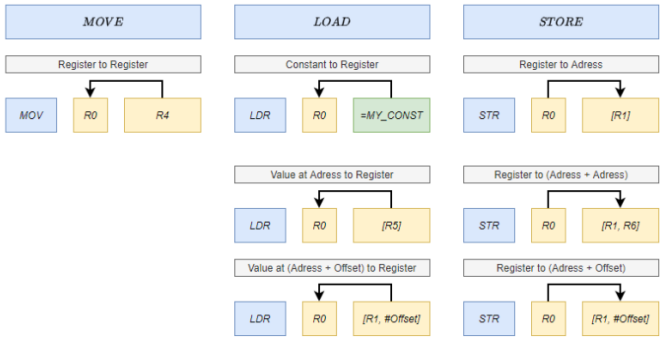
Store Instructions

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:

1. Determine required data size (byte, half-word, word)
2. Choose appropriate load/store instruction
3. Calculate correct memory address
4. Consider alignment requirements
5. 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:

```
;Load operations
MOVS R1, #42       ;Load immediate value
MOVS R2, R1        ;Copy register
LDR  R3, =0x1234   ;Load 32-bit constant
LDR  R4, [R3]      ;Load from memory
LDRB R5, [R3, #1] ;Load byte with offset

;Store operations
STR  R1, [R2]      ;Store word
STRB R1, [R2, #4] ;Store byte with offset
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
                    . . .
                                                    Not content of my_array,
                                                    but address of my_array
                    ; load base and offset registers
0000007C 4906       LDR     R1,=my_array    ; load address of array
0000007E 4B07       LDR     R3,=0x08

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

- 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:

```
; Access array[i] where i is in R1
; Array base address in R0

; For byte array
LDRB    R2, [R0, R1]    ; R2 = array[i]

; For half-word array
LSLS    R2, R1, #1      ; R2 = i * 2
LDRH    R3, [R0, R2]    ; R3 = array[i]

; For word array
LSLS    R2, R1, #2      ; R2 = i * 4
LDR     R3, [R0, R2]    ; R3 = array[i]
```

Multiple Register Transfer

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:

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

Multiple Data Transfer
Loading/Storing multiple registers:

```
; Store multiple registers
PUSH    {R0-R3, LR}     ; Push registers to stack

; Load multiple registers
POP     {R0-R3, PC}     ; Pop and return

; Load multiple memory locations
LDM     R0!, {R1-R4}    ; Load 4 words, update R0

; Store multiple memory locations
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:

```
    ; Load 96-bit value from memory
    ; R3(MSW), R2, R1(LSW) contain result
    ; Memory address in R6
    LDM     R6, {R1-R3}     ; Load all words at once

    ; Alternative using individual loads:
    LDR     R1, [R6]        ; Load LSW
    LDR     R2, [R6, #4]    ; Load middle word
    LDR     R3, [R6, #8]    ; Load MSW
```

2. Storing 96-bit value:

```
    ; Store 96-bit value to memory
    ; R3(MSW), R2, R1(LSW) contain data
    ; Memory address in R6
    STM     R6, {R1-R3}     ; Store all words at once

    ; Alternative using individual stores:
    STR     R1, [R6]        ; Store LSW
    STR     R2, [R6, #4]    ; Store middle word
    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:

```
    LDR     R1, =0xFF55AAB0  ; Pseudo instruction
    ; Assembler converts to:
    LDR     R1, [PC, #offset]
    ...
    DCD     0xFF55AAB0       ; In literal pool
```

Pseudo Instruction vs Direct Load The difference between LDR forms:

```
    LDR     R5, mylita      ; Loads value at mylita
    LDR     R5, =mylita     ; Loads address of mylita

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:

```
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:

```
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:

```
; Copy contents with flags unchanged
MOV     R3, R1           ; For any registers

; Copy contents and update flags
MOVS    R3, R1           ; Only for low registers
```

2. Loading immediate values:

```
; Small values (flags unchanged)
MOV     R0, #42          ; 8-bit immediate

; Larger values
LDR     R0, =0x12345678 ; 32-bit value using literal
    pool

; For high registers
LDR     R0, =value       ; Load into low register first
MOV     R8, R0           ; Then move to high register
```

3. Memory access:

```
; Load from memory
LDR     R0, [R1]         ; Word from address in R1
LDRB    R0, [R1]         ; Byte from address in R1

; Store to memory
STR     R0, [R1]         ; Word to address in R1
STRB    R0, [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 RO with $0 \times AA$ (flags unchanged)
  `LDR R0,=0xAA`
- Initialize R1 with 234 (flags modified)
  `MOVS R1,#234`
- Initialize R4 with $0 \times 55AACC$
  `LDR R4,=0x55AACC`
- Copy contents of R9 in R3
  `MOV R3, R9`
- Initialize R10 with $0 \times 345678$
  `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 × 2
   - Word arrays: offset = index × 4
2. Choose appropriate instructions:

```
    ; Array base in R0, index in R1

    ; For byte array
    LDRB    R2, [R0, R1]     ; index offset

    ; For half-word array
    LSLS    R2, R1, #1       ; multiply index by 2
    LDRH    R3, [R0, R2]     ; load half-word

    ; For word array
    LSLS    R2, R1, #2       ; multiply index by 4
    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, $= 0 \times FF$	R0 = 0x000000FF
	LDR	R1, lit1	R1 = 0x0000EFAA
	LDR	R2, lit2	R2 = 0x00012345
	LDR	R3, $= 0 \times 55AA$	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 = 0xFFFFFF89
	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

```
LDR     R0,=0x20001000      ; R0 = 0x20001000
MOVS    R1,#42              ; R1 = 0x0000002A
STR     R1,[R0]     ; Mem[0x20001000] = 0x0000002A
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

```
; At address 0x1000:
LDR     R0,[PC,#12]         ; PC = 0x1004, loads from
    0x1010
...
ALIGN   4
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

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

```
    LDR     R0,=0x12345678      ; Load constant from
        literal pool
    LDR     R1,myvar            ; Load value at myvar
    LDR     R2,=myvar           ; Load address of myvar

myvar   DCD     0x11223344      ; Define variable
```

Handling Pointers

Example C code and assembly implementation:

```
uint32_t x;
uint32_t y;
uint32_t *xp;

void main(void) {
    x = 3;
    xp = &x;
    y = *xp;
}
```

Assembly implementation:

```
    AREA    myVar, DATA, READWRITE
X   DCD     0
Y   DCD     0
XP  DCD     0

    AREA    myCode, CODE, READONLY
main
    ; Get addresses
    LDR     R3, =X      ; Address of X
    LDR     R4, =Y      ; Address of Y
    LDR     R5, =XP     ; Address of XP

    ; x = 3
    MOVS    R0, #3      ; Load immediate value
    STR     R0, [R3]    ; Store in X

    ; xp = &x
    STR     R3, [R5]    ; Store address of X in XP

    ; y = *xp
    LDR     R2, [R5]    ; Load address from XP
    LDR     R1, [R2]    ; Load value from address
    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** → Rd = Rn + Rm
  - Updates flags, only low registers
- **ADD Rd, Rm** → Rd = Rd + Rm
  - No flag updates, can use high registers
- **ADDS Rd, #imm** → Rd = Rd + immediate
  - 8-bit immediate value only

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

Subtraction Operations

Subtraction instructions and their uses:

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

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

Multiplication

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

Arithmetic Operations

Steps for arithmetic operations:

1. Determine if operation is signed or unsigned
2. 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
3. Consider potential carry/overflow conditions
4. For multi-word operations: see KR **Multi-Word Operations**
5. 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)

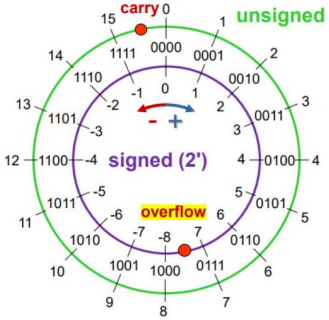
Signed Operations:

- 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 → Carry

```
  1 1 0 1   13d
  0 1 1 1    7d
1 1 1 1
1 0 1 0 0   20d → 16d + 4d
```

Subtraction: C = 0 → Borrow

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

  0 1 1 0    6d
  0 0 1 0    2d = TC(14d)
0 1 1 0
0 1 0 0 0    8d → - 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

```
; Pattern for modifying memory value
LDR     Rx, =variable     ; Get address
LDR     Ry, [Rx]          ; Load value
; Perform operation
STR     Ry, [Rx]          ; Store result
; For 16-bit values
LDRH/STRH                 ; Use half-word instructions
; For 8-bit values
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:

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

Example:

```
    ; Unsigned overflow detection
    ADDS    R0, R1          ; Perform addition
    BCS     overflow        ; Branch if carry set

    ; Signed overflow detection
    ADDS    R0, R1          ; Perform addition
    BVS     overflow        ; Branch if overflow set
```

Flag Usage

Examples of flag behavior:

```
; Zero flag example
MOVS    R0, #5
SUBS    R0, #5          ; Z=1 (result is zero)

; Negative flag example
MOVS    R0, #1
SUBS    R0, #2          ; N=1 (result is negative)

; Carry flag example
MOVS    R0, #0xFF
ADDS    R0, #1          ; C=1 (unsigned overflow)

; Overflow flag example
MOVS    R0, #0x7F       ; Max positive 8-bit
ADDS    R0, #1          ; V=1 (signed overflow)
```

Examples for basic arithmetic operations

Basic Arithmetic Operations

Given variables in data section:

```
    AREA    progData, DATA, READWRITE
Zahl1   DCD     0       ; 32 bit
Zahl2   DCD     0       ; 32 bit
Zahl3   DCW     0       ; 16 bit
Zahl4   DCW     0       ; 16 bit
Zahl5   DCB     0       ; 8 bit
Zahl6   DCB     0       ; 8 bit
```

Basic register operations:

```
; R0 = R1 + R2
ADDS    R0, R1, R2

; R0 = R1 + R2 + R3
ADDS    R0, R1, R2
ADDS    R0, R0, R3

; R8 = R8 + 1
MOVS    R0, #1
ADD     R8, R8, R0

; R8 = -R8
MOV     R7, R8
RSBS    R7, #0
MOV     R8, R7
```

```
; R1 = R2 + 200
MOVS    R1, #200
ADD     R1, R1, R2
; Not ADDS R1,R2,#200
; invalid, rd = rn

; R10 = R8 - R7
MOV     R0, R8
SUBS    R0, R0, R7
MOV     R10, R0
```

Operations involving memory variables:

```
; Zahl1 = Zahl1 - 1
LDR     R1, =Zahl1      ; Get address
LDR     R0, [R1]        ; Load value
SUBS    R0, #1          ; Decrement
STR     R0, [R1]        ; Store back

; Zahl1 = Zahl1 + Zahl2
LDR     R0, =Zahl1
LDR     R1, [R0]        ; Load Zahl1
LDR     R2, =Zahl2
LDR     R3, [R2]        ; Load Zahl2
ADDS    R1, R1, R3      ; Add
STR     R1, [R0]        ; Store result

; Zahl3 = Zahl3 - Zahl4 (unsigned)
LDR     R0, =Zahl3
LDRH    R1, [R0]        ; Load 16-bit
LDR     R2, =Zahl4
LDRH    R3, [R2]
SUBS    R1, R1, R3
STRH    R1, [R0]        ; Store 16-bit

; Zahl1 = R5 * Zahl3 (unsigned)
LDR     R0, =Zahl1
LDR     R1, =Zahl3
LDRH    R2, [R1]
MULS    R2, R5, R2
STR     R2, [R0]
```

Multi-Word Arithmetic

Multi-Word Arithmetic Guidelines for operations on large numbers:

1. Addition sequence:

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

2. Subtraction sequence:

```
    ; 64-bit subtraction (R1:R0 - R3:R2)
    SUBS    R0, R2          ; Subtract low words
    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

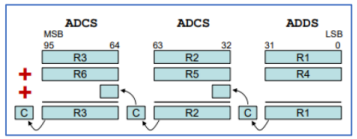
3. Storing results:
- Use STM for efficient multi-word storage
- Maintain proper alignment
- Consider word order (endianness)

Example 96-bit addition:

```
    ; Load values
    LDR     R0, =value1
    LDM     R0!, {R1-R3}    ; First number
    LDR     R0, =value2
    LDM     R0!, {R4-R6}    ; Second number

    ; Add with carry
    ADDS    R1, R1, R4      ; Low word
    ADCS    R2, R2, R5      ; Middle word
    ADCS    R3, R3, R6      ; High word

    ; Store result
    LDR     R0, =result
    STM     R0!, {R1-R3}
```

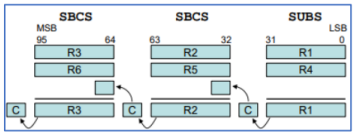
Important considerations:
- Always handle carry/borrow for multi-word operations
- Consider register restrictions (high vs low)
- Pay attention to word size when loading/storing
- Use appropriate instructions for signed vs unsigned operations

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




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

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




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

64-bit Operations

Given 64-bit variables:

```
    AREA    progData, DATA, READWRITE
Long1   DCD     0       ; low word
        DCD     0       ; high word
Long2   DCD     0       ; low word
        DCD     0       ; high word
Long3   DCD     0       ; low word
        DCD     0       ; high word
```

64-bit addition:

```
; Long3 = Long1 + Long2
LDR     R0, =Long1
LDM     R0!, {R1-R4}        ; Load all words
ADDS    R1, R1, R3          ; Add low words
ADCS    R2, R2, R4          ; Add high words with carry
LDR     R0, =Long3
STM     R0!, {R1, R2}       ; Store result
```

64-bit subtraction:

```
; Long3 = Long2 - Long1
LDR     R0, =Long1
LDM     R0!, {R1-R4}        ; Load all words
SUBS    R3, R3, R1          ; Subtract low words
SBCS    R4, R4, R2          ; Subtract high words with
    borrow
LDR     R0, =Long3
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

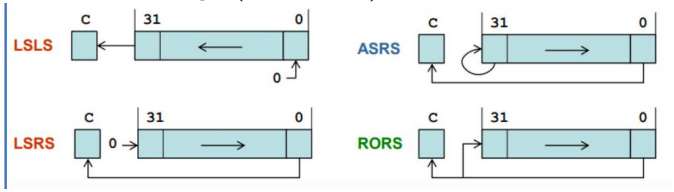
Common logic operations:

```
; Logic operations
ANDS R0, R1          ; R0 = R0 AND R1
BICS R0, R1          ; R0 = R0 AND NOT R1
EORS R0, R1          ; R0 = R0 XOR R1
MVNS R0, R1          ; R0 = NOT R1
ORRS R0, R1          ; R0 = R0 OR R1

; Shift operations
LSLS R0, R1, #2      ; R0 = R1 << 2 (multiply by 4)
LSRS R0, R1, #1      ; R0 = R1 >> 1 (divide by 2)
ASRS R0, R1, #2      ; R0 = R1 >> 2 (signed divide by 4)
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:

```
; Multiplication by powers of 2
LSLS    R0, R0, #1       ; R0 = R0 * 2
LSLS    R0, R0, #2       ; R0 = R0 * 4
LSLS    R0, R0, #3       ; R0 = R0 * 8

; Division by powers of 2
LSRS    R0, R0, #1       ; R0 = R0 / 2 (unsigned)
ASRS    R0, R0, #1       ; R0 = R0 / 2 (signed)

; Multiply by 10 (8 + 2)
LSLS    R1, R0, #3       ; R1 = R0 * 8
ADDS    R0, R0, R1       ; R0 = R0 + (R0 * 8) = R0 * 9
ADDS    R0, R0, R0       ; R0 = R0 * 2 = R0 * 10
```

Using Logic and Shift Instructions

Steps for bit manipulation:

1. Identify required operation (AND, OR, XOR, NOT, shift)
2. Choose appropriate instruction
3. 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:

1. Setting specific bits:

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

2. Clearing specific bits:

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

3. Inverting specific bits:

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

4. Testing bits:

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

Bit Manipulation

```
; Set bits 0 and 4
MOVS    R1, #0x11        ; Mask: 0001 0001
ORRS    R0, R1           ; Set bits in R0

; Clear bits 1 and 5
MOVS    R1, #0x22        ; Mask: 0010 0010
BICS    R0, R1           ; Clear bits in R0

; Toggle bits 2,3,4
MOVS    R1, #0x1C        ; Mask: 0001 1100
EORS    R0, R1           ; Toggle bits in R0

; Test bit 3
MOVS    R1, #0x08        ; Mask: 0000 1000
ANDS    R2, R0, R1       ; Test bit
BEQ     bit_is_clear     ; Branch if bit was 0
```

Casting, Sign Extension and Type Conversion

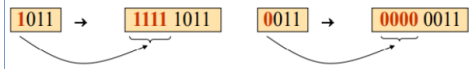
Integer Casting

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
	0x00	0	0
	...	...	...
	0x7F	127	127
	0x80	128	-128
	...	...	...
	0xFF	255	-1

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

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

Sign Extension Instructions

Instructions for extending smaller values:

SXTB: Sign 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

UXTB: Zero extend byte to word

- Takes lowest byte
- Sets bits 31-8 to zero

UXTH:
Zero extend half-word to word

- Takes lowest half-word
- Sets bits 31-16 to zero

Sign Examples

```
; Sign extension examples
SXTB    R0, R1           ; Sign extend byte
SXTH    R0, R1           ; Sign extend half-word

; Zero extension examples
UXTB    R0, R1           ; Zero extend byte
UXTH    R0, R1           ; Zero extend half-word

; Manual sign extension
LSLS    R0, R0, #24      ; Shift left 24 bits
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

```
; Extend 8-bit to 32-bit unsigned
MOVS    R0, #0xFF       ; Load 8-bit value
UXTB    R2, R0          ; Unsigned extension

; Manual zero extension
LDRB    R0, [R1]        ; Load byte, top bits zero
LSLS    R0, #24         ; Move to top byte
LSRS    R0, #24         ; Logical shift back
```

2. For signed to larger signed:
   - Use sign extension (SXTB, SXTH)
   - Or use LSLS followed by ASRS

```
; Extend 8-bit to 32-bit signed
MOVS    R0, #0xFF       ; Load 8-bit value
SXTB    R1, R0          ; Signed extension

; Manual sign extension
LDRSB   R0, [R1]        ; Load with sign extend
LDRB    R0, [R1]        ; Load byte
LSLS    R0, #24         ; Move to top byte
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:

```
    ; Truncate 32-bit to 8-bit
    MOVS    R1, #0xFF       ; Create mask
    ANDS    R0, R1          ; Truncate to 8 bits

    ; Store 32-bit value as 8-bit
    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):

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

2. Selective bit manipulation:

```
; Set bits 3..0 to 1, bits 7..4 to 0
; Invert bits 17-16, keep others unchanged
MOVS    R0, #0xF        ; Pattern for bits 3..0
ORRS    R1, R0          ; Set bits 3..0 to 1
MOVS    R0, #0xF0       ; Pattern for bits 7..4
BICS    R1, R0          ; Clear bits 7..4
MOVS    R0, #0x30
LSLS    R0, #12         ; 0x30000 shift 4 nibbles = 12bit
EORS    R1, R0          ; Invert bits 17-16
```

Multiplication Using Shifts

Multiply R7 by 43 using two different methods:

1. Using multiplication instruction:

```
MOVS    R0, #43         ; Load constant
MULS    R7, R0, R7      ; Multiply
```

2. Using shifts and additions:

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

Reverse Engineering

Convert assembly to C code:
Given C variables:

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

Example 1:

```
; Assembly code
LDR     R0, =ux         ; Load address
LDR     R1, [R0]        ; Load value
LSLS    R1, R1, #1      ; Shift left by 1
LDR     R2, =uy         ; Load uy address
LDR     R3, [R2]        ; Load uy value
ADDS    R3, R1          ; Add shifted ux
LSLS    R3, R3, #3      ; Shift left by 3
STR     R3, [R2]        ; Store back in uy
```

Equivalent C code:

```
uy = ((ux << 1) + uy) << 3;
// or:
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:

```
ux: DCD ? ; uint32_t ux
uy: DCD ? ; uint32_t uy
uz: DCD ? ; uint32_t uz
```

Example 2:

```
; Assembly code
LDR     R0, =ux         ; Load address
LDR     R1, [R0]        ; Load value
LDR     R2, =uy         ; Load address
LDR     R3, [R2]        ; Load value
LDR     R4, =uz         ; Load address
LDR     R5, [R4]        ; Load value
LSRS    R1, R1, #3      ; Shift right by 3
LSLS    R3, R3, #4      ; Shift left by 4
ORRS    R1, R3          ; OR to combine
MVNS    R1, R1          ; Invert bits
ANDS    R1, R5          ; AND with uz
STR     R1, [R4]        ; Store back in uz
```

Equivalent C code:

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

Type Casting

Examples of explicit casting in C:

1. Unsigned to signed casting:

```
uint8_t ux = 100;           // 0x64 or 0110 0100
int8_t sx = (int8_t)ux;     // Same bit pattern
                            // 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:

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

Branch Instructions

Overview Branch Instructions

Branch instructions control program flow:

Type:

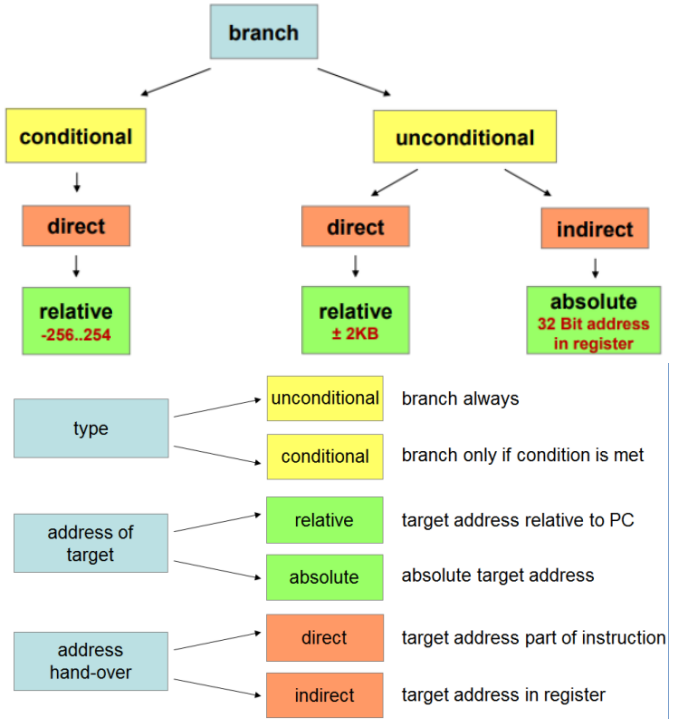
- **Unconditional:** Always taken
- **Conditional:** Branch if condition met

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 dependant 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:

```
    ; Signed comparisons
    LDR     R1, =0xFFFFFFFF
    CMP     R1, #1          ; Compare with immediate
    BLT     is_less         ; Branch if less than

    ; Unsigned comparisons
    CMP     R1, #1
    BLO     is_below        ; Branch if lower

    ; Bit testing
    LDR     R5, =0x0040FFFF
    TST     R3, R5          ; Test bits
    BNE     bits_set        ; Branch if any bit set
    BEQ     bits_clear      ; Branch if all bits clear
```

Branch Pattern Recognition

Common branch patterns and their uses:

1. Basic branching:

```
    ; Simple jump
    B       target          ; Unconditional
    BEQ     target          ; Branch if equal
    BNE     target          ; Branch if not equal

    ; Conditional execution
    CMP     R0, #value
    BCC     target          ; Branch if carry clear
    BGT     target          ; Branch if greater than
```

2. Jump tables:

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

3. Infinite loops:

```
endless B       endless      ; Simple infinite loop

loop    ; Do something
        B       loop         ; Loop forever
```

Conditional Branches with Flag Testing

Example tracking non-taken branches:

```
    MOVS    R0, #0          ; Clear mask

    ; Test carry flag
    ADDS    R1, R1, #5
    BCS     taken
    ADDS    R0, R0, #0x01   ; Mark if not taken
taken

    ; Test equality
    ANDS    R1, R1, R5
    BEQ     equal
    ADDS    R0, R0, #0x02   ; Mark if not taken
equal

    ; Test overflow
    SUBS    R2, R2, R5
    BVS     overflow
    ADDS    R0, R0, #0x04   ; Mark if not taken
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 $\underline{\mathbf{11-16}}$ (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          MOVS     R1, #3
17          LSLS     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  LDR  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, =0x7FFFFFFF
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)

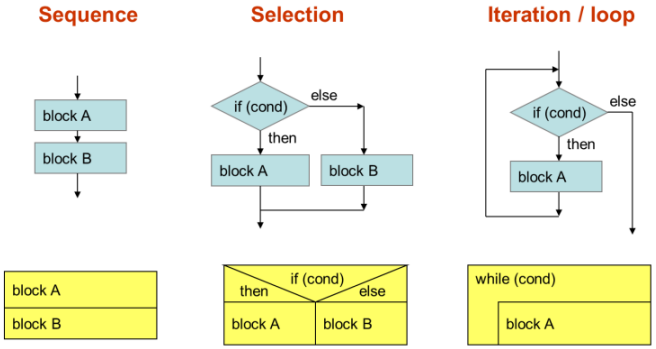
Control Structures

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

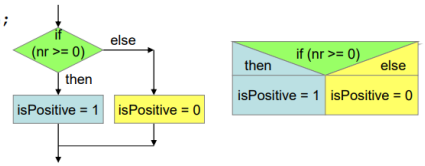
Important considerations:

- 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:

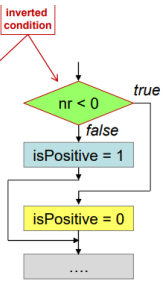
```
int32_t nr, isPositive;
...
if (nr >= 0) {
    isPositive = 1;
}
else {
    isPositive = 0;
}
```




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

Assume: nr in R1, isPositive in R2

```
    CMP   R1,#0x00
    BLT   else
    MOVS  R2,#1
    B     end_if
else
    MOVS  R2,#0
end_if
    ....
```



Selection Implementation

1. Simple if-then:

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

2. if-then-else:

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

```
    CMP     R0, #value      ; Test condition
    BNE     else_part       ; Branch if false
then_part
    ; Then code
    B       endif           ; Skip else
else_part
    ; Else code
endif
    ; Continue execution
```

3. Nested if:

```
    if (x > 0) {
        if (y > 0) {
            x = y;
        }
    }
```

```
    CMP     R0, #0          ; Check x > 0
    BLE     endif_outer
    CMP     R1, #0          ; Check y > 0
    BLE     endif_inner
    MOVS    R0, R1          ; x = y
endif_inner
endif_outer
```

Selection Implementation with multiple Conditions

1. Multiple conditions (AND):

```
if (condA && condB) {    //then-part
} else {                 //else-part
}
```

```
    ; Test first condition
    CMP     Rx, #valA
    B<cc>   else_label          ; Branch if first fails
    ; Test second condition
    CMP     Ry, #valB
    B<cc>   else_label          ; Branch if second fails

    ; Then part
    <then instructions>
    B       endif_label

else_label
    ; Else part
    <else instructions>

endif_label
```

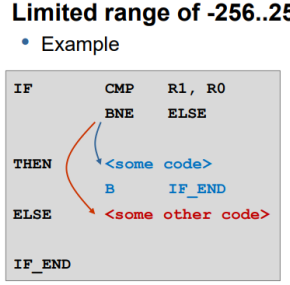
2. Multiple conditions (OR):

```
if (condA || condB) {    //then-part
} else {                 //else-part
}
```

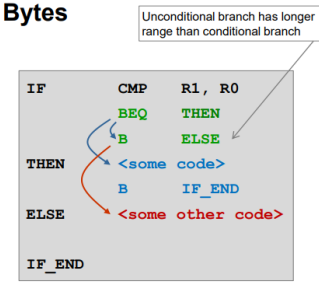
```
    ; Test first condition
    CMP     Rx, #valA
    B<cc>   test_second         ; Try second if first
        fails
    B       then_label          ; First succeeded

test_second
    CMP     Ry, #valB
    B<cc>   else_label          ; Branch if both fail

then_label
    ; Then part
    <then instructions>
    B       endif_label

else_label
    ; Else part
    <else instructions>

endif_label
```

Limitations of Conditional Branches

- **Limited range of -256..254 Bytes**
  - Example



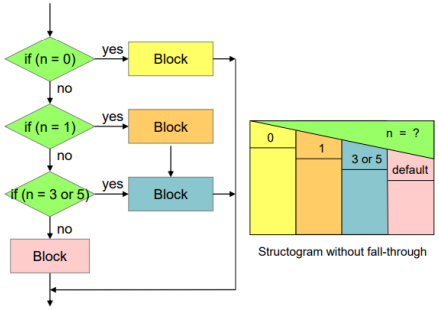
Simple code for the case when <some code> is short



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;
}
```



Structogram without fall-through

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;
}
```

```
NR_CASES      EQU   6
case_switch   CMP   R1, #NR_CASES
              BHS   case_default
              LSLS  R1, #2   ; * 4
              LDR   R7, =jump_table
              LDR   R7, [R7, R1]
              BX    R7
case_0        ADDS  R2, R2, #17
              B     end_sw_case
case_1        ADDS  R2, R2, #13
case_3_5      ADDS  R2, R2, #37
              B     end_sw_case
case_default  MOVS  R2,#0
end_sw_case   ...

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

Assume: n in R1
result in R2

Switch Implementation

1. Range check and table access:

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

2. Jump table structure:

```
jump_table
    DCD     case_0          ; Case 0 handler
    DCD     case_1          ; Case 1 handler
    DCD     default_case    ; Default handler
    ; ... more cases
```

3. Case handlers:

```
case_0
    ; Handle case 0
    B       switch_end
case_1
    ; Handle case 1
    B       switch_end
default_case
    ; Handle default case
switch_end
```

Switch Statement Implementation C code example:

```
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;
}
```

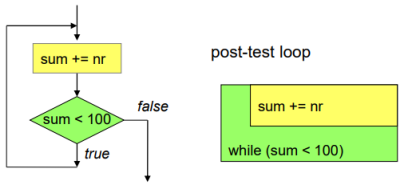
Assembly implementation with jump table:

```
NR_CASES     EQU     6
case_switch  CMP     R1, #NR_CASES
             BHS     case_default
             LSLS    R1, #2          ; * 4
             LDR     R7, =jump_table
             LDR     R7, [R7, R1]
             BX      R7

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

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

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);
```

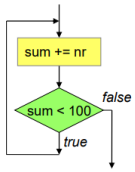


post-test loop


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

Assume: nr in R1
sum in R2

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



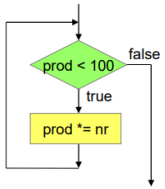
Do-While loop

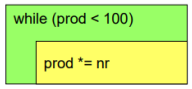
```
1     ; do { x++; } while (x < 10);
2 do_loop
3     ADDS    R0, #1          ; x++
4     CMP     R0, #10         ; Check x < 10
5     BLT     do_loop         ; Continue if true
```

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;
}
```

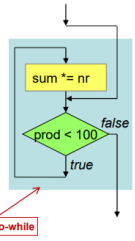





```
int32_t nr;
int32_t prod;
...
prod = 1;
while (prod < 100) {
    prod *= nr;
}
```

Assume: nr in R1
prod in R2

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



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          ; Check i == 2
4     BEQ       outer_continue  ; Skip if i == 2
5
6     MOVS      R1, #0          ; j = 0
7 inner_loop
8     CMP       R1, #1          ; Check j == 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:

```
    MOVS    R2, #0          ; Index
loop
    LDRB    R3, [R0, R2]    ; Load char
    CMP     R3, #0          ; Check end
    BEQ     done            ; Exit if terminator
    ; Process character
    ADDS    R2, R2, #1      ; Next char
    B       loop
```

2. Character transformation:

```
    ; Check character range
    CMP     R3, #lower_bound
    BLO     skip            ; Below range
    CMP     R3, #upper_bound
    BHI     skip            ; Above range

    ; Transform character
    ADDS    R3, #offset     ; Apply offset

skip
    STRB    R3, [R1, R2]    ; Store result
```

3. String copy:

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

Important considerations:

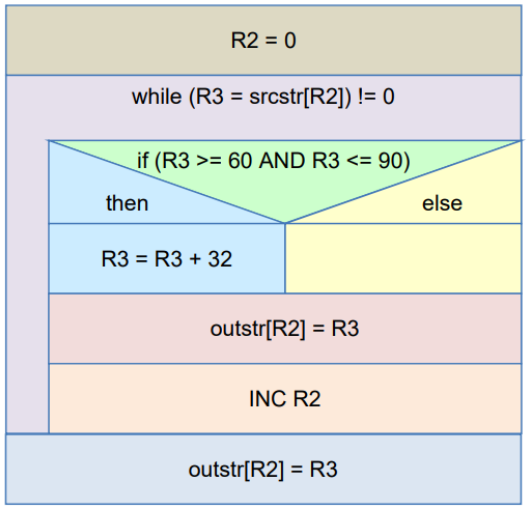
- Choose appropriate conditional branches
- Consider signed vs unsigned comparisons
- Handle edge cases and termination
- Maintain proper register allocation
- Document complex control flow

String Processing Loop

Converting string to uppercase:

```
    AREA    progCode, CODE, READONLY
    THUMB
main
    PROC
    EXPORT  main
    LDR     R0, =srcstr     ; Source string
    LDR     R1, =outstr     ; Output string
    MOVS    R2, #0          ; Initialize index

cond
    LDRB    R3, [R0, R2]    ; Load character
    CMP     R3, #0          ; Check for end
    BEQ     endloop         ; Exit if done
    CMP     R3, #60         ; Check if < 'a'
    BLO     store           ; Skip if not lowercase
    CMP     R3, #90         ; Check if > 'z'
    BHI     store           ; Skip if not lowercase
    ADDS    R3, R3, #32     ; Convert to uppercase

store
    STRB    R3, [R1, R2]    ; Store character
    ADDS    R2, R2, #1      ; Next character
    B       cond            ; Continue loop

endloop
    STRB    R3, [R1, R2]    ; Store terminator
    ENDP

srcstr  DCB     "This IS mY TestStriNG", 0
    AREA    progData, DATA, READWRITE
outstr  SPACE   50
```

Structogram:



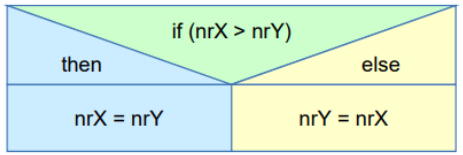
Result: "THIS IS MY TESTSTRING"

Examples

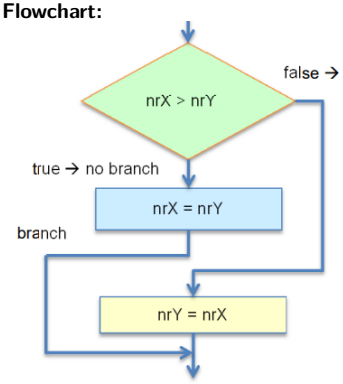
Selection Structures

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

Structogram:



Flowchart:



C code:

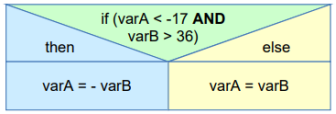
```
uint8_t nrX = ...;
uint8_t nrY = ...;
if (nrX > nrY) {
    nrX = nrY;
} else {
    nrY = nrX;
}
```

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:

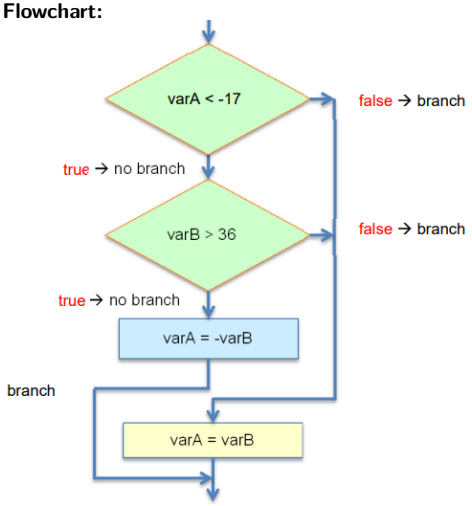
Structogram:



C code:

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

Flowchart:

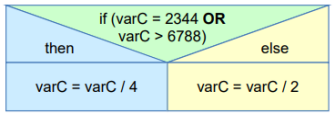


Assembly code:

```
        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:

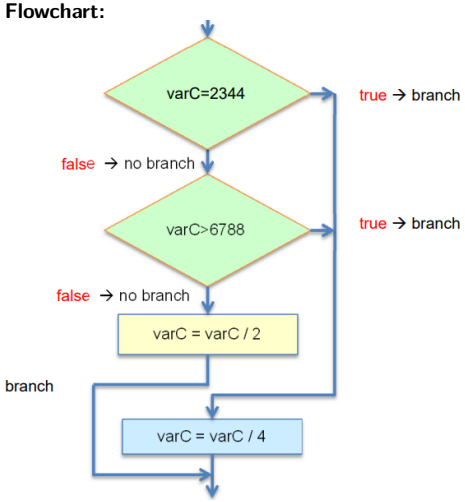
Structogram:



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
        SXTH  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:

```
    AREA    progCode, CODE, READONLY
    THUMB
main
    PROC
    EXPORT  main

    LDR     R6, =i          ; R6 = address of i
    LDR     R0, [R6]        ; R0 = value at i
    LDR     R7, =count      ; R7 = address of count
    LDR     R1, [R7]        ; R1 = value at count
    B       cond

loop
    ADDS    R0, R0, #1      ; Increment i
    ADDS    R1, R1, #1      ; Increment count
cond
    CMP     R0, #10
    BLT     loop            ; Branch if i < 10

    STR     R0, [R6]        ; Store final i
    STR     R1, [R7]        ; Store final count
    ENDP
```

Compiler-optimized version:

```
    LDR     r1, [pc, #20]   ; Load address
    MOVS    r0, #0          ; Initialize counter
    STR     r0, [r1, #0]    ; Store i
    LDR     r2, [r1, #4]    ; Load count

increment
    ADDS    r0, r0, #1      ; i++
    ADDS    r2, r2, #1      ; count++
    CMP     r0, #10         ; Check condition
    BLT     increment       ; Loop if i < 10
    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:

1. Define subroutine entry point with label
2. Save registers that will be modified
   - Use PUSH at start
   - Include LR if calling other subroutines
3. Implement subroutine logic
4. Restore registers in reverse order
   - Use POP before return
   - Can return using POP ..., PC if LR was saved
5. 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, =0xEEFF1020
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:

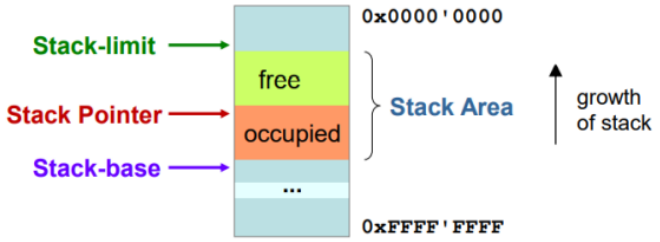
- **Stack Area** (Section): Continuous RAM section
- **Stack Pointer (SP)**: R13, points to last written value
- **Direction**: Full-descending (grows toward lower addresses)
- **Alignment**: Word-aligned (4 bytes)
- **Data Size**: 32-bit words only

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

```
ADDR_LED_31_0   EQU     0x60000100
LED_PATTERN     EQU     0xA55A5AA5

subrExample     PUSH    {R4,R5,LR}

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

                BL      write7seg

                POP     {R4,R5,PC}
```

Save LR and registers used by subroutine

Call another subroutine

Restore registers and PC

```
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:

```
func    PUSH    {LR}        ; Save return address
        ; Function body
        POP     {PC}        ; Return
```

2. Function with locals:

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

3. Function with parameters:

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

Stack Frame
Example of complete function:

```
; int calc(int a, int b, int c)
; a in R0, b in R1, c in R2
calc    PUSH    {R4-R6, LR} ; Save registers
        ; Save parameters
        MOVS    R4, R0      ; Save a
        MOVS    R5, R1      ; Save b
        MOVS    R6, R2      ; Save c
        ; Call helper function
        MOVS    R0, R4      ; First param
        BL      helper      ; Call helper
        ; Continue calculation
        ADDS    R0, R5      ; Add b
        ADDS    R0, R6      ; Add c

        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:

```
    ; Save registers and create frame
    PUSH    {R4-R7, LR}     ; Save registers
    SUB     SP, SP, #frame_size  ; Allocate space
    SUB     SP, SP, #locals ; Allocate local vars

    ; Initialize frame if needed
    MOV     R4, #0          ; Clear locals
    STR     R4, [SP, #0]    ; Initialize var1
    STR     R4, [SP, #4]    ; Initialize var2
```

3. Stack frame access:

```
    ; Access local variables
    LDR     R0, [SP, #offset1]  ; Load local1
    STR     R1, [SP, #offset2]  ; Store to local2

    ; Access parameters
    LDR     R0, [SP, #20]   ; First stack parameter
    ; Access parameters beyond R0-R3
    LDR     R0, [SP, #param_offset] ; Load param
```

4. Frame cleanup/Function epilogue:

```
    ; Deallocate frame and restore
    ADD     SP, SP, #frame_size  ; Remove locals
    ADD     SP, SP, #locals ; Deallocate locals
    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:

```
func    ; Function prologue
    PUSH    {R4-R8, LR}     ; Save registers
    SUB     SP, SP, #12     ; Allocate locals
    ; Access local variables relative to SP
    STR     R0, [SP, #0]    ; Local var 1
    STR     R1, [SP, #4]    ; Local var 2
    STR     R2, [SP, #8]    ; Local var 3
    ; Function body
    BL      other_func      ; Call another function
    ; Function epilogue
    ADD     SP, SP, #12     ; Deallocate locals
    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
   - Stack grows downward in memory
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:

```
main    ; SP = 0x20002000
        LDR     R0, =0x11111111
        LDR     R1, =0x22222222
        BL      funcA           ; Z1
        B       endless

funcA   PUSH    {R0, R1, LR}    ; Z2
        BL      funcB
        POP     {R0, R1, PC}

funcB   PUSH    {R4, LR}        ; Z3
        MOVS    R4, #42
        POP     {R4, PC}
```

Stack contents at Z1, Z2, Z3:

```
Z1: Stack is empty, SP = 0x20002000

Z2: Stack from top (higher address):
0x20001FF4: Return addr to main
0x20001FF0: 0x22222222 (R1)
0x20001FEC: 0x11111111 (R0)
SP = 0x20001FEC

Z3: Stack from top (higher address):
0x20001FE4: Return addr to funcA
0x20001FE0: R4 value
0x20001FF4: Return addr to main (from Z2)
0x20001FF0: 0x22222222 (R1 from Z2)
0x20001FEC: 0x11111111 (R0 from Z2)
SP = 0x20001FE0
```

Parameter Passing

Parameter Passing Methods

Data can be passed between functions through:

- **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 dara area
- Overhead to access variable
- Error prone, unmaintainable

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

AREA exCode,CODE,...
  ...
  LDR     R4,=param1        ; caller
  MOVS    R5,#0x03
  STRB    R5,[R4]
  BL      double_g
  LDR     R4,=result
  LDRB    ...,[R4]
  ...

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

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

pass by value:

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

AREA exCode,CODE,...
  ...
  MOVS  R1,#0x03            ; caller
  BL    double
  MOVS  ...,R0
  ...

double                      ; callee function double
  LSLS  R0,R1,#1
  BX    LR
```

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** ($\leq$ 32 bits → smaller than word size):
  – Return in R0
  – Zero/sign extend to word if needed
- **Word** (32 bits): return in R0
- **Double Word** (64 bits): return in R0/R1
- **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

Complete stack frame layout:

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

Steps for calling functions:

Caller's responsibilities:

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

Callee's responsibilities:

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

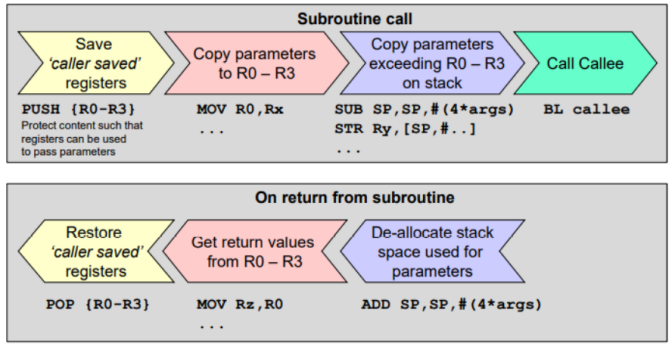
Important considerations:

- 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.



Function Parameter Guidelines

Best practices for parameter passing:

1. Register Usage:

- R0-R3: First four parameters
- R0: Return value
- R4-R11: Preserve if used

2. Stack Usage:

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

3. Memory Structures:

- Pass pointers for large structures
- Use registers for small values
- Consider alignment requirements

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:

```
    ; Save any needed registers
    PUSH    {R4-R6, LR}       ; Save registers

    ; Load parameters into R0-R3
    MOV     R0, R4            ; First parameter
    MOV     R1, R5            ; Second parameter
    MOV     R2, R6            ; Third parameter

    ; Call function
    BL      function

    ; Save return value if needed
    MOV     R7, R0            ; Save result

    ; Restore registers
    POP     {R4-R6, PC}       ; Return
```

2. Callee responsibilities:

```
function
    ; Save any registers we'll modify
    PUSH    {R4, LR}

    ; Process parameters in R0-R3
    ; Put return value in R0

    ; Restore registers
    POP     {R4, PC}
```

3. Reference parameter handling:

```
    ; Loading from pointer
    LDR     R2, [R0]          ; Get value at address
    ; Storing to pointer
    STR     R2, [R1]          ; Store to address
    ; Incrementing pointer
    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:

```
// Incorrect swap - pass by value
void swap_bad(int32_t c, int32_t d) {
    int32_t temp = c;
    c = d;
    d = temp;
}
// Correct swap - pass by reference
void swap_good(int32_t *c, int32_t *d) {
    int32_t temp = *c;
    *c = *d;
    *d = temp;
}
int32_t main(void) {
    int32_t a = 3, b = 5;

    swap_bad(a, b);   // Doesn't work
    swap_good(&a, &b); // Works correctly
}
```

Assembly implementation of swap_good:

```
swap_good
    PUSH    {LR}              ; Save return address

    LDR     R2, [R0]          ; Load *c into R2
    LDR     R3, [R1]          ; Load *d into R3

    STR     R3, [R0]          ; Store R3 to *c
    STR     R2, [R1]          ; Store R2 to *d

    POP     {PC}              ; Return
```

Complex Parameter Example

Function with mixed parameter types:

```
typedef struct {
    int32_t x;
    int32_t y;
} point_t;

int32_t calculate(point_t* p, int32_t scale,
                  int32_t* result);
```

Assembly implementation:

```
; R0 = point_t* p
; R1 = scale
; R2 = result pointer
calculate
    PUSH    {R4-R5, LR}     ; Save registers

    ; Load structure members
    LDR     R4, [R0, #0]    ; Load p->x
    LDR     R5, [R0, #4]    ; Load p->y

    ; Perform calculation
    MULS    R4, R1, R4      ; x * scale
    MULS    R5, R1, R5      ; y * scale

    ; Store result
    STR     R4, [R2, #0]    ; *result = x
    ADDS    R0, R4, R5      ; Return sum

    POP     {R4-R5, PC}     ; Return
```

Data Structure Access

Working with structures and arrays:

```
typedef struct {
    uint32_t minutes;
    uint32_t seconds;
} time_t;

time_t time;
```

Assembly implementation:

```
    ; Access structure members
    LDR     R0, =time         ; Get structure address
    LDR     R1, [R0, #0]      ; Load minutes
    LDR     R2, [R0, #4]      ; Load seconds

    ; Modify structure
    ADDS    R2, #1            ; Increment seconds
    CMP     R2, #60           ; Check for overflow
    BLT     store_back
    MOVS    R2, #0            ; Reset seconds
    ADDS    R1, #1            ; Increment minutes
store_back
    STR     R1, [R0, #0]      ; Store minutes
    STR     R2, [R0, #4]      ; Store seconds
```

Recursive Function Example

Factorial calculation showing stack usage:

```
int32_t fakultaet_recursive(int32_t n) {
    if(n < 2) {
        return 1;
    } else {
        return n * fakultaet_recursive(n-1);
    }
}

int32_t main(void) {
    int32_t n = 20;
    int32_t result = fakultaet_recursive(n);
}
```

Assembly implementation showing stack growth:

```
fakultaet_recursive
    PUSH    {R4, LR}          ; Each call adds 8 bytes
    MOV     R4, R0            ; Save n

    CMP     R0, #2            ; Check base case
    BLT     return_one

    SUB     R0, R0, #1        ; n-1
    BL      fakultaet_recursive
    MUL     R0, R4, R0        ; n * result

    POP     {R4, PC}          ; Return

return_one
    MOV     R0, #1            ; Return 1
    POP     {R4, PC}          ; Return
```

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

Function Call Example

C function and its parameter passing:

```c
uint32_t logical_and(uint32_t a, uint32_t b, uint32_t
    c) {
    return a & b & c;
}

int32_t main(void) {
    uint32_t x = 0x11223344;  // In R4
    uint32_t y = 0xFFFF0000;  // In R5
    uint32_t z = 0x33661122;  // In R6
    uint32_t result;          // In R7

    result = logical_and(x, y, z);
}
```

Beim Start des Programmes wird die Variable x in R4, y in R 5 und z in R 6 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. $R6 \rightarrow R2, R5 \rightarrow R1, R4 \rightarrow R0$

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:

```
main
    ; Initial register setup
    LDR     R4, =0x11223344   ; x
    LDR     R5, =0xFFFF0000   ; y
    LDR     R6, =0x33661122   ; z

    ; Parameter passing
    MOV     R0, R4            ; a = x
    MOV     R1, R5            ; b = y
    MOV     R2, R6            ; c = z
    BL      logical_and
    MOV     R7, R0            ; result = return value

logical_and
    AND     R0, R0, R1        ; a & b
    AND     R0, R0, R2        ; & c
    BX      LR                ; Return
```

Modular Coding and Linking

From source code to executable Program

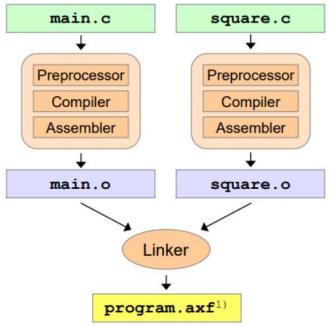
Compile/Asseble each Module: Results in an object file for each module
Link all object files together: Creates a single executable file

main.c

```
...

uint32_t square(uint32_t v);

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

square.c

```
...

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



Tool Chain Components

Essential tools for development:

- **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 an 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
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
a       DCD     0x00000005
b       DCD     0x00000007
```

usable outside of module main

from module square

Working with Linkage in ARM Assembly

1. **Exporting symbols:** Use EXPORT directive before symbol definition
   - Only export symbols needed by other modules
   - Can export both code and data symbols
2. **Importing symbols:** Use IMPORT directive before symbol usage
   - Import must match exactly with export name
   - Import before first use of symbol
3. **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):

```
    AREA |.text|, CODE, READONLY
    EXPORT add_and_square
    IMPORT square_value     ; From module 2

add_and_square PROC
    PUSH    {R4, LR}
    ADDS    R4, R0, R1      ; Add parameters
    MOVS    R0, R4          ; Prepare for square
    BL      square_value    ; Call imported function
    POP     {R4, PC}
    ENDP
    END
```

Module 2 (square.s):

```
    AREA |.text|, CODE, READONLY
    EXPORT square_value

square_value PROC
    PUSH    {LR}
    MULS    R0, R0, R0      ; Square the input
    POP     {PC}
    ENDP
    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:

```
    AREA |.text|, CODE, READONLY
    EXPORT module_init     ; Public interface
    EXPORT module_process

module_init PROC
    PUSH    {LR}
    BL      internal_init ; Private implementation
    POP     {PC}
    ENDP

module_process PROC
    ; Public processing function
    ENDP

internal_init PROC
    ; Private initialization
    ENDP
```

2. Shared data structures:

```
    AREA myData, DATA, READWRITE
    EXPORT module_state    ; Public state

module_state              ; Accessible by other modules
    SPACE   16

private_buffer            ; Internal buffer
    SPACE   32
```

3. Constants and configuration:

```
    AREA myConstants, DATA, READONLY
    EXPORT CONFIG_VERSION
    EXPORT MAX_BUFFER_SIZE

CONFIG_VERSION
    DCD     0x00000001

MAX_BUFFER_SIZE
    DCD     0x00000100

internal_config           ; Private configuration
    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;
}
```

square = external linkage

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

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

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

```c
int global_var;           // External linkage
void global_func(void);   // External linkage
```

Internal Linkage:

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

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

No Linkage:

- Local variables and function parameters
- Scope limited to block

```c
void func(void) {
    int local_var;       // No linkage
    static int static_var; // Internal linkage
}
```

Module Interface Design

Example of a well-designed modular system:

```c
// sensor.h - Public interface
typedef struct {
    uint32_t id;
    uint32_t value;
} sensor_reading_t;

void sensor_init(uint32_t id);
sensor_reading_t sensor_read(void);

// sensor.c - Implementation
static uint32_t current_sensor_id;
static uint32_t last_reading;

void sensor_init(uint32_t id) {
    current_sensor_id = id;
    last_reading = 0;
}

sensor_reading_t sensor_read(void) {
    sensor_reading_t reading;
    reading.id = current_sensor_id;
    reading.value = /* Read hardware */;
    last_reading = reading.value;
    return reading;
}
```

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

File sections:

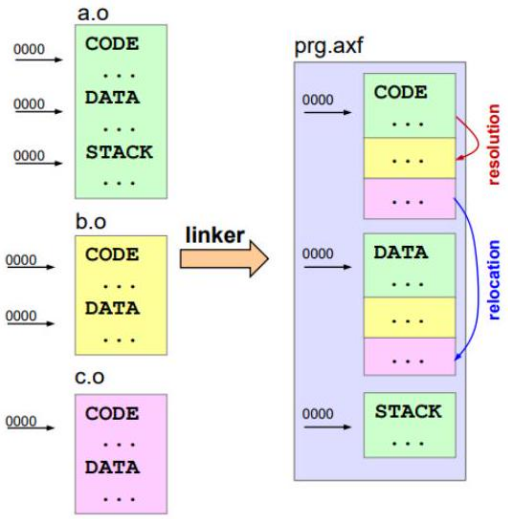
```
; 1. '.text' section (Code):
0x00000000: 4604   MOV      r4,r0
0x00000002: 0040   LSLS     r0,r0,#1
0x00000004: 4420   ADD      r0,r4
; 2. '.data' section:
0x00000000: Initial values for global data
; 3. Symbol table:
;  #  Name       Value    Type    Binding
   6  myFunc     0x0000   CODE    Global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

```
    ; In module1.s
    AREA |.text|, CODE, READONLY
    EXPORT func1
func1
    ; function code

    ; In module2.s
    AREA |.text|, CODE, READONLY
    IMPORT func1
    BL      func1    ; Reference to resolve
```

2. Relocation:

```
    ; Before relocation
    BL      func1    ; Relative offset

    ; After relocation
    BL      0x08000234  ; Absolute address
```

Linking Multiple Modules

Example of linking multiple assembly modules:

```
; math_ops.s
    AREA |.text|, CODE, READONLY
    EXPORT add_values
    EXPORT multiply_values

add_values PROC
    ADDS R0, R0, R1
    BX LR
    ENDP

multiply_values PROC
    MULS R0, R1, R0
    BX LR
    ENDP
    END

; main.s
    AREA |.text|, CODE, READONLY
    IMPORT add_values
    IMPORT multiply_values
    EXPORT main

main PROC
    PUSH {R4,LR}
    MOVS R0, #5
    MOVS R1, #3
    BL add_values       ; Call add_values
    MOVS R4, R0         ; Save result
    MOVS R0, #6
    MOVS R1, #2
    BL multiply_values ; Call multiply_values
    ADD R0, R0, R4     ; Combine results
    POP {R4,PC}
    ENDP
    END
```

Creating and Using Libraries

Library Creation and Use

Steps for creating and using libraries:

1. Create library source files:

```
// lib.h
void lib_func(int x);

// lib.c
void lib_func(int x) {
    // Implementation
}
```

2. Compile to object files:

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

3. Create static library:

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

4. Link with library:

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

Creating Static Libraries

Creating and using a static library:

1. Create object files:

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

2. Create static library:

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

3. Link with library:

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

Example C code for the library:

```
// math_ops.h
uint32_t add_values(uint32_t a, uint32_t b);
uint32_t multiply_values(uint32_t a, uint32_t b);

// math_ops.c
uint32_t add_values(uint32_t a, uint32_t b) {
    return a + b;
}

uint32_t multiply_values(uint32_t a, uint32_t b) {
    return a * b;
}
```

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):

```
// module.h
#ifndef MODULE_H_
#define MODULE_H_

// Public type definitions
typedef struct {
    uint32_t x;
    uint32_t y;
} point_t;

// Public function declarations
void init_module(void);
uint32_t process_point(point_t* p);

#endif
```

3. Create implementation files (.c):

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

// Private/static functions
static void helper_function(void) {
    // Internal implementation
}

// Public function implementations
void init_module(void) {
    // Initialize module state
}

uint32_t process_point(point_t* p) {
    helper_function();
    return p->x + p->y;
}
```

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:

```
    AREA myData, DATA, READWRITE
    EXPORT global_counter     ; Public variable
global_counter
    DCD     0x00000000

local_counter                 ; Private variable
    DCD     0x00000000

    AREA myCode, CODE, READONLY
    EXPORT init_counter       ; Public function
    EXPORT increment

init_counter PROC             ; Can be called from other
    modules
    PUSH    {LR}
    LDR     R0, =global_counter
    MOVS    R1, #0
    STR     R1, [R0]
    POP     {PC}
    ENDP

increment PROC                ; Can be called from other
    modules
    PUSH    {LR}
    BL      internal_helper ; Private function call
    POP     {PC}
    ENDP

internal_helper PROC          ; Private function
    PUSH    {LR}
    LDR     R0, =global_counter
    LDR     R1, [R0]
    ADDS    R1, #1
    STR     R1, [R0]
    POP     {PC}
    ENDP
```

Module Interface Example

```
    ; Module A - Defining function
    AREA myCode, CODE, READONLY
    EXPORT myFunction     ; Make available externally
myFunction
    PUSH    {LR}
    ; function code here
    POP     {PC}

    ; Module B - Using function
    AREA myCode, CODE, READONLY
    IMPORT myFunction     ; Use external function

    BL      myFunction    ; Call the function
```

Data Symbol Linkage

Example showing data symbol linkage:
Header file (constants.s):

```
    AREA myData, DATA, READONLY
    EXPORT MAX_VALUE
    EXPORT MIN_VALUE

MAX_VALUE
    DCD     0x000000FF    ; Public constant
MIN_VALUE
    DCD     0x00000000    ; Public constant

internal_value            ; Private constant
    DCD     0x00000055
    END
```

Usage file (process.s):

```
    AREA |.text|, CODE, READONLY
    IMPORT MAX_VALUE
    IMPORT MIN_VALUE

validate_value PROC
    PUSH    {LR}
    LDR     R1, =MAX_VALUE
    LDR     R1, [R1]      ; Load max value
    CMP     R0, R1
    BGT     invalid       ; Above max
    LDR     R1, =MIN_VALUE
    LDR     R1, [R1]      ; Load min value
    CMP     R0, R1
    BLT     invalid       ; Below min
    MOVS    R0, #1        ; Valid
    POP     {PC}
invalid
    MOVS    R0, #0        ; Invalid
    POP     {PC}
    ENDP
    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	set PRIMASK	CPSID[1)] i	__disable_irq();
- Enable	clear PRIMASK	CPSIE[1)] i	__enable_irq();

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

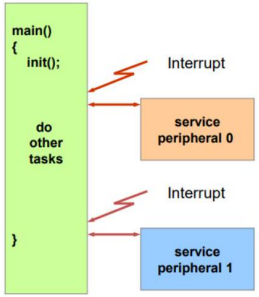
Key concepts for interrupt handling:
- **Interrupt Sources**:
  - Hardware interrupts from peripherals
  - Software interrupts (SVC)
  - System exceptions
- **Configuration Steps**:
  - Enable specific interrupt source
  - Install interrupt handler
  - Configure interrupt priority
  - Enable global interrupts
- **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

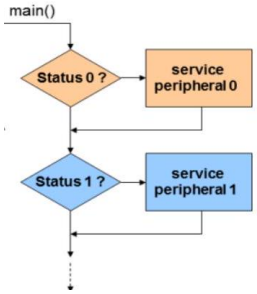


Polling

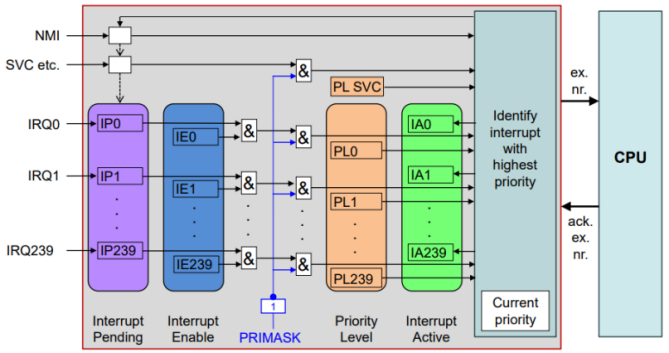
Periodic query of status information

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

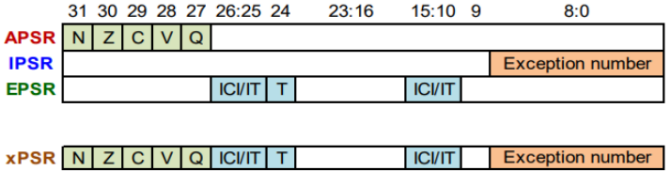


Interrupt Control



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

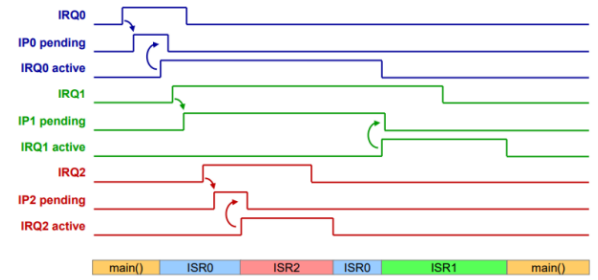
Interrupt priority handling:
- **Priority Levels**:
  - 0-255 (lower number = higher priority)
  - Fixed priorities for system exceptions
  - Programmable priorities for IRQs
- **Preemption**:
  - Higher priority interrupts can preempt lower
  - Same priority follows FIFO

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



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 (IRQn=1), but interrupts are disabled (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:

```
SETENA0 EQU 0xE000E100    ; Enable interrupts
CLRENA0 EQU 0xE000E180    ; Disable interrupts
; Enable IRQ3
LDR     R0, =SETENA0
MOVS    R1, #(1<<3)
STR     R1, [R0]
; Disable IRQ3
LDR     R0, =CLRENA0
MOVS    R1, #(1<<3)
STR     R1, [R0]
```

2. Pending Registers:

```
SETPEND0 EQU 0xE000E200   ; Set pending
CLRPEND0 EQU 0xE000E280   ; Clear pending
; Set IRQ3 pending
LDR     R0, =SETPEND0
MOVS    R1, #(1<<3)
STR     R1, [R0]
; Clear IRQ3 pending
LDR     R0, =CLRPEND0
MOVS    R1, #(1<<3)
STR     R1, [R0]
```

Exception Vector Table Setup and usage:

Vector table structure and Handler implementation:

```
    AREA RESET, DATA, READONLY
__Vectors
    DCD     __initial_sp        ; Top of Stack
    DCD     Reset_Handler       ; Reset
    DCD     NMI_Handler         ; NMI
    DCD     HardFault_Handler   ; Hard Fault
    DCD     0                   ; Reserved
    DCD     0                   ; Reserved
    ; ... more vectors
    DCD     IRQ0_Handler        ; IRQ0
    DCD     IRQ1_Handler        ; IRQ1
```

```
    AREA |.text|, CODE, READONLY

IRQ0_Handler PROC
    EXPORT IRQ0_Handler
    PUSH    {R4-R7,LR}
    ; Handle interrupt
    POP     {R4-R7,PC}
    ENDP
```

NVIC Configuration

Complete interrupt system setup:

```
void init_timer_interrupt(void) {
    // Configure timer peripheral
    TIM2->PSC = 7199;          // Prescaler
    TIM2->ARR = 9999;          // Auto-reload value
    TIM2->DIER |= 1;           // Enable interrupt
    TIM2->CR1 |= 1;            // Enable timer

    // Configure NVIC
    NVIC_SetPriority(TIM2_IRQn, 2);  // Set priority
    NVIC_EnableIRQ(TIM2_IRQn);       // Enable IRQ

    __enable_irq();            // Enable global
        interrupts
}

// Timer 2 interrupt handler
void TIM2_IRQHandler(void) {
    if (TIM2->SR & 1) {        // Check update flag
        // Handle interrupt
        TIM2->SR &= ~1;        // Clear flag
    }
}
```

Assembly equivalent:

```
TIM2_BASE     EQU     0x40000000
TIM_SR        EQU     0x10
TIM_DIER      EQU     0x0C
SETENA0       EQU     0xE000E100

    ; Enable timer interrupt
    LDR     R0, =TIM2_BASE
    LDR     R1, [R0, #TIM_DIER]
    ORRS    R1, #1
    STR     R1, [R0, #TIM_DIER]

    ; Enable in NVIC
    LDR     R0, =SETENA0
    MOVS    R1, #1
    LSLS    R1, #28          ; IRQ28
    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 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:

```
// Global time counters accessed by ISR
volatile uint32_t minutes = 0;
volatile uint32_t hours = 0;

// Timer ISR updates counters
void TIM2_IRQHandler(void) {
    minutes++;
    if (minutes >= 60) {
        minutes = 0;
        hours++;
    }
    // Clear interrupt flag
}

// Main code reading counters
void read_time(uint32_t *min, uint32_t *hr) {
    // Disable interrupts to read consistent values
    __disable_irq();
    *min = minutes;
    *hr = hours;
    __enable_irq();
}
```

Assembly equivalent:

```
read_time
    PUSH    {R4-R5, LR}
    CPSID   i                ; Disable interrupts

    LDR     R4, =minutes     ; Load minutes
    LDR     R2, [R4]
    STR     R2, [R0]         ; Store to output

    LDR     R5, =hours       ; Load hours
    LDR     R3, [R5]
    STR     R3, [R1]         ; Store to output

    CPSIE   i                ; Enable interrupts
    POP     {R4-R5, PC}
```

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:

```
handler_name
    PUSH    {R4-R6}          ; Save used registers

    ; Check interrupt flags
    ; Handle interrupt condition
    ; Clear interrupt flags

    POP     {R4-R6}          ; Restore registers
    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:

```
    ; Disable all interrupts
    CPSID   i

    ; Critical section code
    ; Access shared resources

    ; Enable interrupts
    CPSIE   i
```

Timer Interrupt Configuration

Configuring Timer 2 interrupt:

1. Handler definition:

```
    AREA    handlers, CODE, READONLY

    EXPORT  TIM2_IRQHandler
TIM2_IRQHandler
    PUSH    {R4-R6}          ; Save registers
    ; Handle interrupt
    POP     {R4-R6}          ; Restore registers
    BX      LR               ; Return
```

2. Enable interrupt (IRQ28):

```
    AREA    startup, CODE, READONLY

SETENA0     EQU     0xE000E100  ; Interrupt enable
    register

    ; Enable Timer 2 interrupt
    LDR     R7, =SETENA0
    MOVS    R6, #1           ; Set bit
    LSLS    R6, #28          ; Shift to IRQ28
    STR     R6, [R7]         ; Enable interrupt
```


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:

```
void init_timer_interrupt(void) {
    // Enable timer interrupt
    NVIC_EnableIRQ(TIM2_IRQn);

    // Set priority
    NVIC_SetPriority(TIM2_IRQn, 2);

    // Configure timer
    // ...

    // Enable global interrupts
    __enable_irq();
}
```

Data Consistency

Example protection:

```
void update_shared_data(void) {
    __disable_irq();          // Critical section start
    shared_var++;             // Update shared data
    __enable_irq();           // Critical section end
}
```

Nested Interrupts Example

Implementation with different priorities:

```
// Initialize interrupts
void init_interrupts(void) {
    // Enable interrupts
    NVIC_EnableIRQ(IRQ0_IRQn);
    NVIC_EnableIRQ(IRQ1_IRQn);

    // Set priorities
    NVIC_SetPriority(IRQ0_IRQn, 1); // Higher
    NVIC_SetPriority(IRQ1_IRQn, 2); // Lower

    // Enable global interrupts
    __enable_irq();
}

// Higher priority ISR
void IRQ0_Handler(void) {
    // Handle high priority interrupt
    // Can't be interrupted by IRQ1
}

// Lower priority ISR
void IRQ1_Handler(void) {
    // Handle low priority interrupt
    // Can be interrupted by IRQ0
}
```


Kurzfristiges Aus- und Einschalten von Interrupts

Enabling von 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**

RISC

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

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

CISC

- One of the operands of an instruction may directly be a memory location

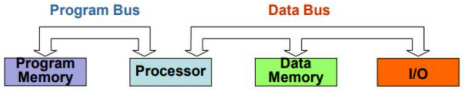
```
MOV  AX, [Credit]
ADD  [Balance], AX
```

Von Neumann Architectures

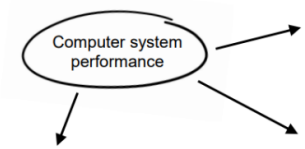
- Single memory for program and data
- Single bus system between CPU and memory

Harvard Architecture:

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



How to increase system speed?



External factors:

- Better Compilers
- Better Algorithms

CPU improvements:

- Clock Speed
- Cache Memory
- Multiple Cores / Threads
- Pipelined Execution
- Branch Prediction
- Out of Order Execution
- Instruction Set Architecture

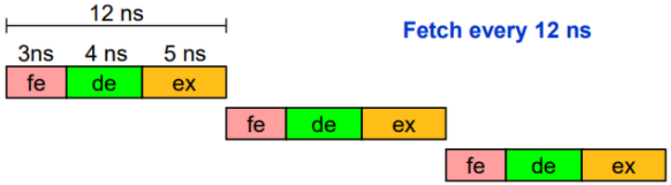
System level factors:

- Special Purpose Units
  e.g. Crypto, Video, AI
- Multiple Processors
- Bus Architecture
  e.g. von Neumann / Harward
- Faster components (e.g. SSDs, 1000Base-T, etc.)

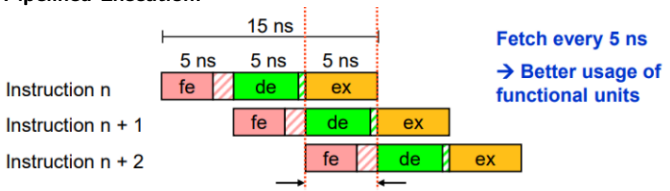
Pipelining

Sequential vs. Pipelined Execution

Sequential Execution:



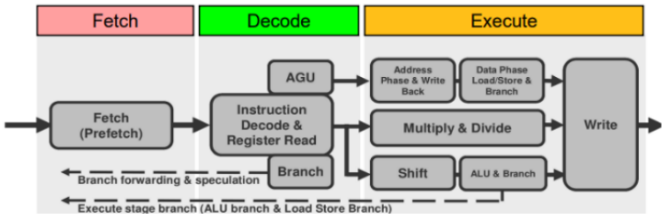
Pipelined Execution:



Longest stage (ex) defines pipeline cycle: 5 ns

Pipelining

Process of fetching next instruction while current one decodes:



Advantages:

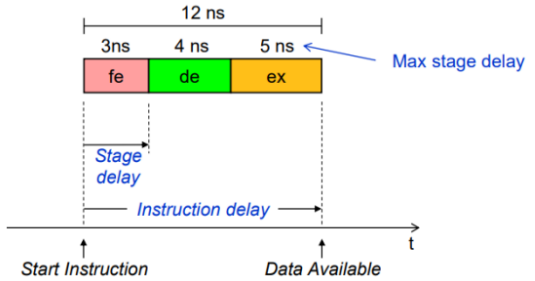
- Uniform execution time per stage
- Significant performance improvement
- Simpler hardware per stage allows higher clock rates

Disadvantages:

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

Pipeline Stages (Example)

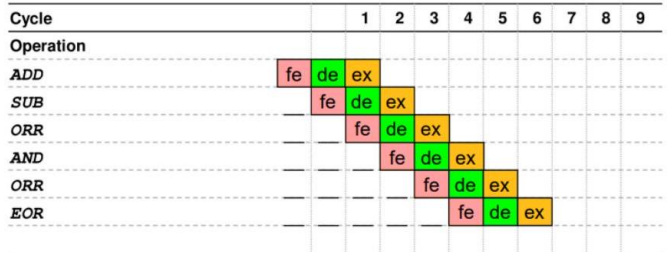
- Fetch (Fe): Read instruction - 3ns
- Decode (De): Process instruction - 4ns
- Execute (Ex): Execute and writeback - 5ns



Pipeline Execution

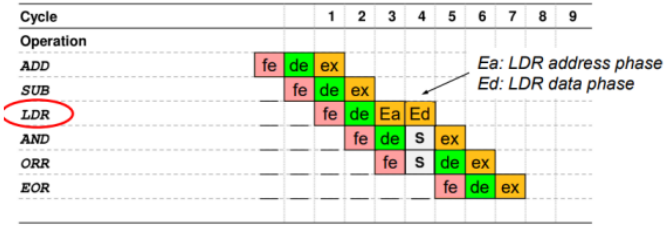
Optimal Case:

- 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

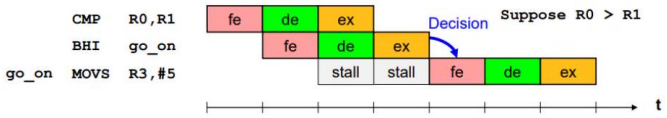
- Stage delays: Fe=3ns, De=4ns, Ex=5ns
- Without pipeline: 12ns per instruction
- With pipeline: 5ns per instruction after filling
- Performance improvement: 2.4×

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

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



Reduce control hazards: Use loop fusion!

Pipeline Hazard Analysis

1. Structural Hazard:

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

2. Data Hazard:

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

3. Control Hazard:

```
CMP   R0, #0       ; Compare
BEQ   target       ; Branch if equal
ADD   R1, R2, R3   ; May be unnecessary
SUB   R4, R5, R6   ; May be unnecessary
target
; 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
- "Meltdwon" 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; 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:

```c
typedef struct {
    int *array;
    int start;
    int end;
    long sum;
} task_t;

void* partial_sum(void* arg) {
    task_t* task = (task_t*)arg;
    long sum = 0;

    for (int i = task->start; i < task->end; i++) {
        sum += task->array[i];
    }
    task->sum = sum;
    return NULL;
}
// Split work across multiple cores
void parallel_sum(int* array, int size, int num_cores)
    {
    task_t tasks[MAX_CORES];
    int chunk = size / num_cores;
    // Create tasks for each core
    for (int i = 0; i < num_cores; i++) {
        tasks[i].array = array;
        tasks[i].start = i * chunk;
        tasks[i].end = (i == num_cores-1) ?
                       size : (i+1) * chunk;
        // Launch task on core i
        launch_on_core(i, partial_sum, &tasks[i]);
    }
    // Wait for all cores and combine results
    long total = 0;
    for (int i = 0; i < num_cores; i++) {
        wait_for_core(i);
        total += tasks[i].sum;
    }
    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:

```c
// Scalar addition
void add_arrays_scalar(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}

// SIMD/Vector addition (conceptual)
void add_arrays_vector(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i += 4) {
        // Load 4 elements at once
        vector_t va = vector_load(&a[i]);
        vector_t vb = vector_load(&b[i]);
        // Add 4 pairs of elements simultaneously
        vector_t vc = vector_add(va, vb);
        // Store 4 results at once
        vector_store(&c[i], vc);
    }
}
```

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):

```c
#ifndef _MODULE_H
#define _MODULE_H

typedef enum {
    STATE_A = 0x00,
    STATE_B = 0x01
} module_state_t;

void module_init(void);
void module_set_state(module_state_t state);
module_state_t module_get_state(void);

#endif /* _MODULE_H */
```

2. Implementation File (module.c):

```c
#include "module.h"

static module_state_t current_state;
static void helper_function(void);

void module_init(void) {
    current_state = STATE_A;
}

void module_set_state(module_state_t state) {
    current_state = state;
    helper_function();
}

module_state_t module_get_state(void) {
    return current_state;
}

static void helper_function(void) {
    // Implementation
}
```

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