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
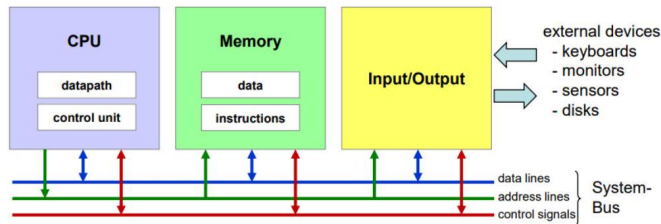
CPU Components
The CPU contains several key components:
Datapath:
- **Core Registers**: Fast but limited storage inside CPU
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations

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

Program Translation Process
from C to executable

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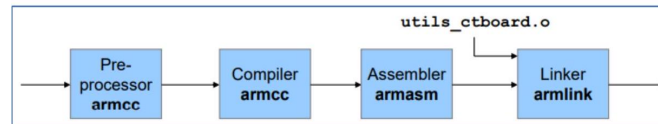
1. **Preprocessor**:
   - Text processing
   - Includes header files (#include)
   - Expands macros (#define)
   - Output: Modified source program (.i)
2. **Compiler**:
   - Translates C to assembly
   - CPU-specific code generation
   - Optimization (if enabled)
   - Output: Assembly program (.s)
3.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

Program Compilation Process
To compile and link a program:
1. Create source files (.c) and header files (.h)
2. Run preprocessor to expand includes and macros
3. Compile source files to object files
4. Link object files and libraries
5. Test executable

Common compiler flags:
- -c: Compile only, don't link
- -o: Specify output file name
- -O[0-3]: Optimization level
- -g: Include debug information

Simple Program Translation - From Source to Executable

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

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

After preprocessing (.i):

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

Assembly output (.s):

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

Host vs Target Development

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

Understanding assembly language is important because it:
- Helps understand machine-level operation
- Aids in debugging and optimization
- Required for system programming
- Essential for security analysis

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:
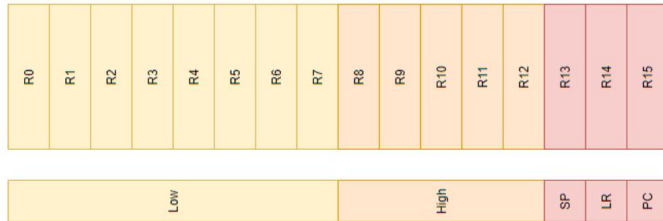
- **R0-R7**: Lower registers - general purpose
  – Used by most instructions
  – Parameter passing in functions (R0-R3)
  – Results returned in R0
- **R8-R12**: Higher registers - general purpose
  – Limited instruction support
  – Often used for temporary storage
- **R13 (SP)**: **S**tack **P**ointer - 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:

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

Instruction Set
The Cortex-M uses 16-bit Thumb instructions:

Label	Instr.	Operands	Comments
demoprg	MOVS	R0,#0xA5	; copy 0xA5 into register R0
	MOVS	R1,#0x11	; copy 0x11 into register R1
	ADDS	R0,R0,R1	; add contents of R0 and R1

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

### Common Instruction Formats
Register operations:

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

Basic Assembly Program Structure
Example of a simple assembly program:

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

Assembly Program Sections
Program memory organized in sections:
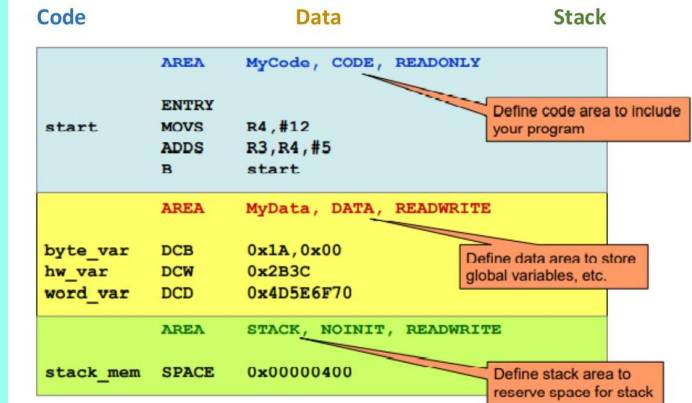
Code Section (CODE):
- Contains program instructions
- Usually read-only
- Placed in Flash memory
- Can contain constants
  (literal pool)

Data Section (DATA):
- Contains global/static variables
- Read-write access
- Placed in RAM
- Initialized at startup

Stack Section: (STACK)
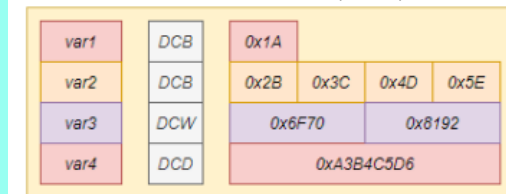- Dynamic memory allocation
- Used for local variables
- Function call management
- Grows downward in memory

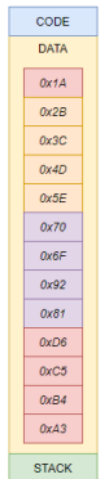### Initialized vs uninitialized Data
Directives for initialized data:
- **DCB**: Define Constant Byte (8-bit)
- **DCW**: Define Constant Half-Word (16-bit)
- **DCD**: Define Constant Word (32-bit)

Directive for uninitialized data:
- **SPACE**: Reserve specified number of bytes

Data Definition   Memory layout for different data types:

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

Creating Assembly Programs   Steps for creating an assembly program:
1. Define program sections: (CODE, DATA)

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

2. Declare external symbols: (IMPORT/EXPORT)

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

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

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

4. Write program code using proper instruction syntax:

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

5. End program with END directive!

Common Assembly Patterns
1. Loop with counter:

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

2. Memory copy:

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

3. Function call with parameters:

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

Complete Program Example
Program to sum array elements:

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

ADD COMPLETE EXAMPLES HERE!

Data Transfer

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

Two main approaches to memory access, the other one:
Register Memory Architecture (e.g., Intel x86):
- Operations can use memory operands directly
- Results can be written directly to memory
- More flexible but more complex instructions

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

Load Instruction Examples

```
 1  ; MOV examples
 2  MOVS    R1, #0xFF        ; Load immediate 255
 3  MOVS    R2, R1           ; Copy R1 to R2
 4  ; LDR examples
 5  LDR     R1, =0x12345678  ; Load 32-bit constant
 6  LDR     R2, [R1]         ; Load from address in R1
 7  LDR     R3, [R1, #4]     ; Load with offset
 8  LDR     R4, [R1, R2]     ; Load with register offset
 9  ; Byte/Half-word loads
10  LDRB    R1, [R2]         ; Load unsigned byte
11  LDRSB   R1, [R2]         ; Load signed byte
12  LDRH    R1, [R2]         ; Load unsigned half-word
13  LDRSH   R1, [R2]         ; Load signed half-word
```
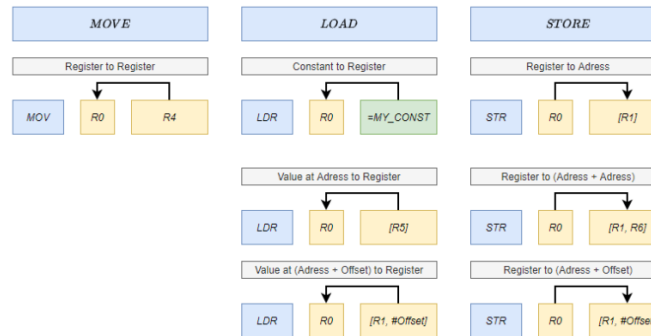
Store Instructions
Instructions for storing data from registers to memory:
- **STR** (Store Register):
  - Basic store: STR R1, [R2]
  - With immediate offset: STR R1, [R2, #0x04]
  - With register offset: STR R1, [R2, R3]
  - Word-aligned addresses only
- **STRB** (Store Register Byte):
  - Stores lowest 8 bits of register
  - No alignment requirements
- **STRH** (Store Register Half-word):
  - Stores lowest 16 bits of register
  - Must be half-word aligned

Memory Access

Memory Layout
for array elements and instructions:

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

Memory Access Patterns
in Load/Store Architecture
Steps for accessing memory:
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:

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

Common Data Transfer Patterns
1. Copy memory block:

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

2. Initialize memory block:

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

3. Search memory:

```
 1      ; R0 = start, R1 = value to find, R2 = count
 2  loop
 3      LDR    R3, [R0], #4    ; Load and increment
 4      CMP    R3, R1          ; Compare with value
 5      BEQ    found           ; Branch if found
 6      SUBS   R2, #1          ; Decrement counter
 7      BNE    loop            ; Continue if not zero
 8  not_found
 9      ; Handle not found case
10  found
11      ; Handle found case
```

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,
                ; load base and offset registers  but address of my_array]
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:

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

Multiple Register Transfer

Multiple Register Transfer
LDM (Load Multiple) and STM (Store Multiple):
- Load/store multiple registers in one instruction
- More efficient than individual loads/stores
- Used for stack operations (PUSH/POP)
- Register list specified in curly braces
Example:

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

Multiple Data Transfer Loading/Storing multiple registers:

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

Multi-Word Data Transfer
For transferring data larger than 32 bits:
1. Loading 96-bit value:

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

2. Storing 96-bit value:

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

Stack Operations

Stack Access Instructions
Special variants of LDM/STM for stack operations:
- **PUSH {register list}**:
  - Decrements SP
  - Stores registers
  - Example: PUSH {R0-R3, LR}
- **POP {register list}**:
  - Loads registers
  - Increments SP
  - Example: POP {R0-R3, PC}

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

Pseudo Instructions

LDR Pseudo Instructions
The LDR pseudo instruction LDR Rx, =value is expanded by the assembler:
1. For literal values:
- Assembler creates 'literal pool' at convenient location
- Allocates and initializes memory with DCD directive
- Uses PC-relative addressing to access value
2. For addresses:
- Places address in literal pool
- Generates PC-relative load instruction
Example:

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

Pseudo Instruction vs Direct Load The difference between LDR forms:

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

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

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

Example encodings:

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

Subtraction Operations   Subtraction instructions and their uses:

- **SUBS Rd, Rn, Rm**
  - Rd = Rn - Rm
  - Updates flags
  - Only low registers

- **RSBS Rd, Rn, #0**
  - Rd = -Rn (2's complement)
  - Special case for negation

- **SUBS Rd, #imm**
  - Rd = Rd - immediate
  - 8-bit immediate value

Example encodings:

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

Multiplication   Simple multiplication examples:

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

Signed vs. Unsigned Arithmetic

Arithmetic Operations   Steps for arithmetic operations:
1. Determine if operation is signed or unsigned
2. Choose appropriate instruction (with or without 'S')
3. Consider potential carry/overflow conditions
4. For multi-word operations:
   - Start with least significant words
   - Use carry-aware instructions for higher words
   - Track flags through operation
5. Check relevant flags after operation **FLAGS ON NEXT PAGE**

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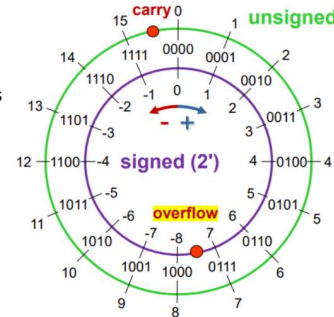
Carry and Overflow
Unsigned Operations:
- Addition: C = 1 indicates carry (result too large for available bits)
- Subtraction: C = 0 indicates borrow (result negative)

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)

Multi-Word Arithmetic
Guidelines for operations on large numbers:
1. Addition sequence:

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

2. Subtraction sequence:

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

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

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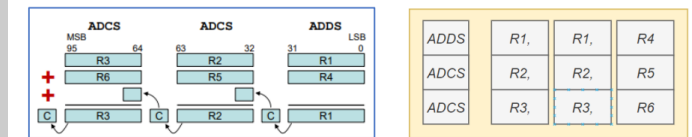

```
1  ADDS R1, R1, R4    ; Add least significant words
2  ADCS R2, R2, R5    ; Add middle words with carry
3  ADCS R3, R3, R6    ; Add most significant words with
       carry
```

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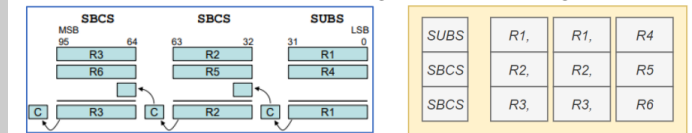

```
1  SUBS R1, R1, R4    ; Subtract least significant words
2  SBCS R2, R2, R5    ; Subtract middle words with borrow
3  SBCS R3, R3, R6    ; Subtract most significant words
       with borrow
```

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:

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

Flag Usage   Examples of flag behavior:

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

Logic, Shift and Rotate Instructions

Logic Instructions

Base logic operations (affect only N and Z flags):
- **ANDS**: Bitwise AND (Rdn & Rm, a & b)
- **BICS**: Bit Clear (Rdn & !Rm, a & b)
- **EORS**: Exclusive OR (Rdn $Rm, a ∧ b)
- **MVNS**: Bitwise NOT (!Rm, a)
- **ORRS**: Bitwise OR (Rdn # Rm, a | b)

Logical Operations   Common logic operations:

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

Shift and Rotate Instructions

Shift operations for binary manipulation:
- **LSLS**: Logical Shift Left ($2^n \cdot Rn$, $0 \rightarrow$ LSB)
- **LSRS**: Logical Shift Right ($2^{-n} \cdot Rn$, $0 \rightarrow$ MSB)
- **ASRS**: Arithmetic Shift Right (R^{-n}, $\pm$MSB $\rightarrow$ MSB)
- **RORS**: Rotate Right (LSB $\rightarrow$ MSB)
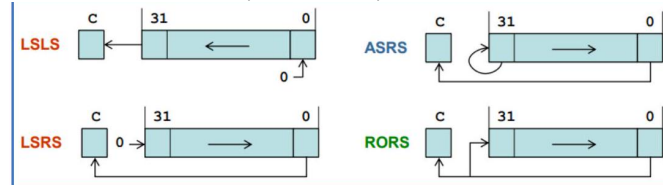
Shift Operations for Arithmetic   Using shifts for multiplication and division:

```
1  ; Multiplication by powers of 2
2  LSLS    R0, R0, #1       ; R0 = R0 * 2
3  LSLS    R0, R0, #2       ; R0 = R0 * 4
4  LSLS    R0, R0, #3       ; R0 = R0 * 8
5
6  ; Division by powers of 2
7  LSRS    R0, R0, #1       ; R0 = R0 / 2 (unsigned)
8  ASRS    R0, R0, #1       ; R0 = R0 / 2 (signed)
9
10 ; Multiply by 10 (8 + 2)
11 LSLS    R1, R0, #3       ; R1 = R0 * 8
12 ADDS    R0, R0, R1       ; R0 = R0 + (R0 * 8) = R0 * 9
13 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
4. For shifts:
   - LSLS for multiplication by 2^n
   - LSRS for unsigned division by 2^n
   - ASRS for signed division by 2^n
5. 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 flag**: Unchanged
- **V flag**: 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 operations on individual bits:

1. Set specific bits:

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

2. Clear specific bits:

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

3. Toggle specific bits:

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

4. Test specific bits:

```
1     ; Test bit 3
2     MOVS    R1, #0x08        ; Mask: 0000 1000
3     ANDS    R2, R0, R1       ; Test bit
4     BEQ     bit_is_clear     ; Branch if bit was 0
```
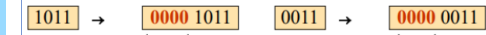
Casting, Sign Extension and Type Conversion

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

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

Type Conversion Guidelines

Steps for safe type conversion:

1. For unsigned to larger unsigned:
- Use zero extension (UXTB, UXTH)
- Or use LSLS followed by LSRS

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

3. Reducing size (truncation):
- Use AND with appropriate mask
- Or store using STRB/STRH
- Check for potential data loss

Example:

```
; Convert 8-bit to 32-bit
MOVS    R0, #0xFF        ; Load 8-bit value
SXTB    R1, R0           ; Signed extension
UXTB    R2, R0           ; Unsigned extension

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

Important considerations:
- Always consider signedness of values
- Check for potential overflow in arithmetic shifts
- Remember carry flag behavior in shifts
- Use appropriate extension for data type
- Consider performance impact of shifts vs multiply

Branch Instructions

Overview Branch Instructions
Branch instructions control program flow:
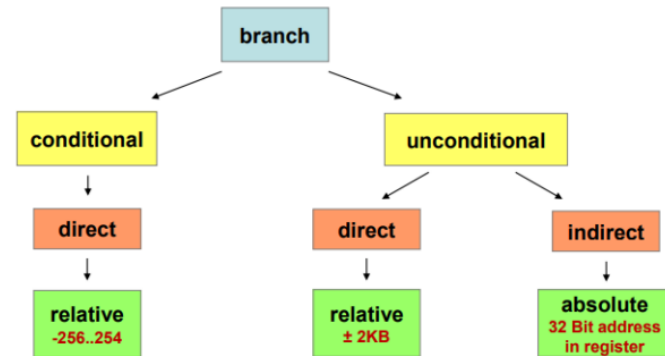
Type:
- **Unconditional:** Always taken
- **Conditional:** Branch if condition met

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

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**

Branch Instructions

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

Control Structures

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

IF-ELSE

```
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

Guidelines for implementing if-then-else structures:

1. Simple if-then:

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

2. if-then-else:

```
1       ; if (x > y) { x = y; } else { y = x; }
2       CMP     R0, R1          ; Compare x and y
3       BLE     else_part       ; Branch if x <= y
4       MOVS    R0, R1          ; Then part: x = y
5       B       endif           ; Skip else part
6 else_part
7       MOVS    R1, R0          ; Else part: y = x
8 endif
```

3. Nested if:

```
1       ; if (x > 0) {
2       ;       if (y > 0) {
3       ;           x = y;
4       ;       }
5       ; }
6       CMP     R0, #0          ; Check x > 0
7       BLE     endif_outer
8       CMP     R1, #0          ; Check y > 0
9       BLE     endif_inner
10      MOVS    R0, R1          ; x = y
11 endif_inner
12 endif_outer
```

Switch-Case

■ Jump Table

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

}
```

Assume: n in R1
         result in R2

```
NR_CASES        EQU     6

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

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

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

Switch Implementation

Steps for implementing switch statements:

1. Range check and table access:

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

2. Jump table structure:

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

3. Case handlers:

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

Switch Statement Implementation   C code example:

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

Assembly implementation with jump table:

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

Loop Types   Three main types of loops:

Do-While (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
        ....
```
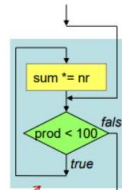
While (Pre-Test Loop):

```
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
        ...
```

```
sum *= nr
prod < 100   false
             true
```

For Loop (Pre-Test Loop):

C	Assembly

`#include <utils_ctboard.h>`
`#include <stdint.h>`
`...`
`int32_t = 0;`
`int32_t count = 0;`
`for(i = 0; i < 10; i++) {`
`    count++;`
`}`
	` ` `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`
	` ` `ADDS  R1,R1,#1`
	`cond` `CMP   R0, #10`
	` ` `BLT   loop      ; *signed* comparison`
	` ` `STR   R0,[R6]    ; store final i`
	` ` `STR   R1,[R7]    ; store final count`
	`endless B    endless`
	` ` `ENDP`
	` `
	` ` `AREA progData, DATA, READWRITE`
	`i` `DCD    0`
	`count` `DCD    0`
	` ` `END`

Loop Implementation
Templates for different loop types:

1. 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
```

2. 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
```

3. 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      ; }
8
9      MOVS    R0, #0           ; i = 0
10 outer_loop
11     CMP     R0, #2           ; Check i == 2
12     BEQ     outer_continue   ; Skip if i == 2
13
14     MOVS    R1, #0           ; j = 0
15 inner_loop
16     CMP     R1, #1           ; Check j == 1
17     BEQ     outer_continue   ; Break to outer loop
18
19     ADDS    R2, R0, R1       ; Calculate i + j
20     ADDS    R4, R4, R2       ; Add to sum
21
22     ADDS    R1, #1           ; j++
23     CMP     R1, #3           ; Check j < 3
24     BLT     inner_loop       ; Continue inner loop
25
26 outer_continue
27     ADDS    R0, #1           ; i++
28     CMP     R0, #5           ; Check i < 5
29     BLT     outer_loop       ; Continue outer loop
```

Subroutines and Stack

Subroutine

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

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
```

in detail:
- Label with name **MulBy3**
- Return Statement **BX LR**

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

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

Nested Subroutine Calls   Example of nested calls:

```
1  main
2      BL      proc_a          ; Call proc_a
3      ; continue main
4
5  proc_a
6      PUSH    {LR}            ; Save return address
7      BL      proc_b          ; Call proc_b
8      POP     {PC}            ; Return to main
9
10 proc_b
11     PUSH    {LR}            ; Save return address
12     BL      proc_c          ; Call proc_c
13     POP     {PC}            ; Return to proc_a
14
15 proc_c
16     ; Do something
17     BX      LR              ; Return to proc_b
```
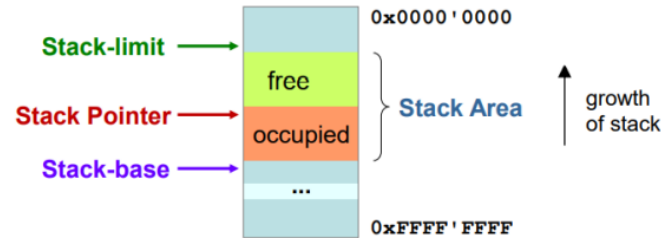
Stack

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
  - $\rightarrow$ Stack-limit $\leq$ SP $\leq$ Stack-base

Stack Instructions

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

PUSH/POP Implementation

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

Stack 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 Management
Steps for function prologue and epilogue:
1. Function Prologue:

```
1      PUSH    {R4-R7, LR}     ; Save registers
2      SUB     SP, SP, #locals ; Allocate local vars
```

2. Function Epilogue:

```
1      ADD     SP, SP, #locals ; Deallocate locals
2      POP     {R4-R7, PC}     ; Restore and return
```

3. Stack frame access:

```
1      ; Access local variables
2      STR     R0, [SP, #0]    ; First local
3      STR     R1, [SP, #4]    ; Second local
4
5      ; Access parameters
6      LDR     R0, [SP, #20]   ; First stack parameter
```

Stack Frame Layout   Example of complete function:

```
1  ; int calc(int a, int b, int c)
2  ; a in R0, b in R1, c in R2
3  calc    PUSH    {R4-R6, LR} ; Save registers
4
5          ; Save parameters
6          MOVS    R4, R0      ; Save a
7          MOVS    R5, R1      ; Save b
8          MOVS    R6, R2      ; Save c
9
10         ; Call helper function
11         MOVS    R0, R4      ; First param
12         BL      helper      ; Call helper
13
14         ; Continue calculation
15         ADDS    R0, R5      ; Add b
16         ADDS    R0, R6      ; Add c
17
18         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

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

Function Implementation Patterns

Common 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
```

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

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

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

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

Parameter Passing by Value vs. Reference

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

pass by value:

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

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

double
    LSLS   R0,R1,#1
    BX     LR
```
(caller / callee function double)

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
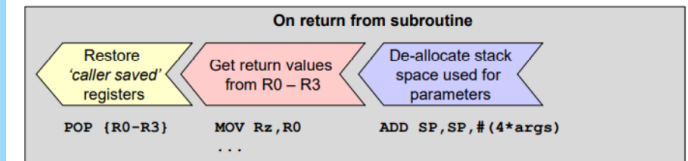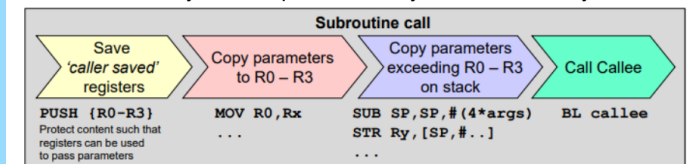- Place return value in R0
- Restore saved registers

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.

Subroutine call

Save 'caller saved' registers	Copy parameters to R0 – R3	Copy parameters exceeding R0 – R3 on stack	Call Callee
PUSH {R0-R3} Protect content such that registers can be used to pass parameters	MOV R0,Rx ...	SUB SP,SP,#(4*args) STR Ry,[SP,#..] ...	BL callee

On return from subroutine

Restore 'caller saved' registers	Get return values from R0 – R3	De-allocate stack space used for parameters
POP {R0-R3}	MOV Rz,R0 ...	ADD SP,SP,#(4*args)

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
```

Complex Parameter Example   Function with mixed parameter types:

```c
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:

```c
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
```

Modular Coding and Linking

Modular Programming Overview
Program code is divided into modules with:
- Each source file compiled into separate object file
- All object files linked into single executable
- Clear interfaces between modules
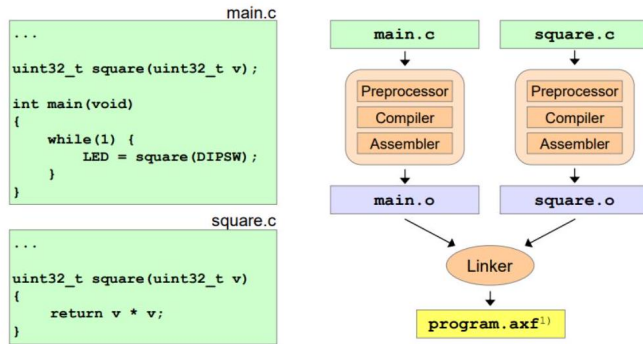

Benefits of Modular Programming
Key advantages:
- **Team Development**:
  - Multiple developers working on same codebase
  - Clear ownership of modules
- **Code Organization**:
  - Logical partitioning of functionality
  - Easier code reuse
- **Development Efficiency**:
  - Individual module testing
  - Faster compilation (only changed modules)
  - Reusable library creation
- **Language Integration**:
  - Mix C and assembly modules
  - Language-specific optimizations

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
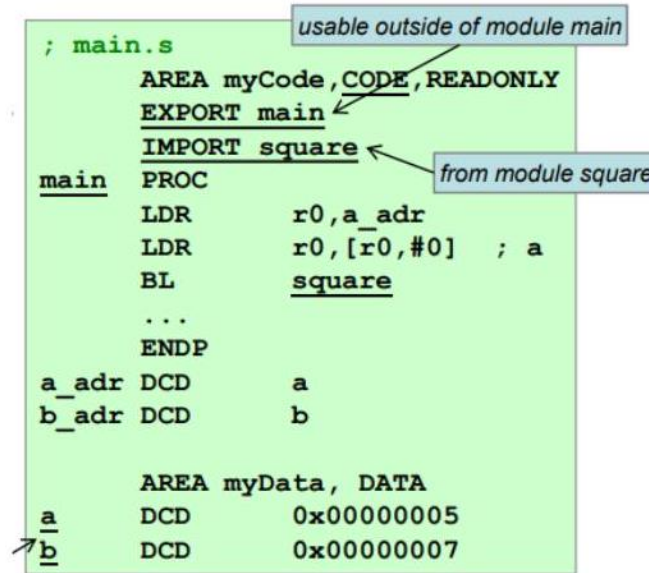
Object Files
ELF format contains:
- **Code Section**:
  - Program code and constants
  - Based at address 0x0
- **Data Section**:
  - Global variables
  - Based at address 0x0
- **Symbol Table**:
  - All symbols and their attributes
  - Global/local status
  - References to external symbols
- **Relocation Table**:
  - Instructions for adjusting addresses
  - Applied during linking process

Linker Operation
Main tasks:
- Merge code sections from all objects
- Merge data sections from all objects
- Resolve symbol references between modules
- Relocate addresses to final positions

Output is ARM Executable File (AXF):
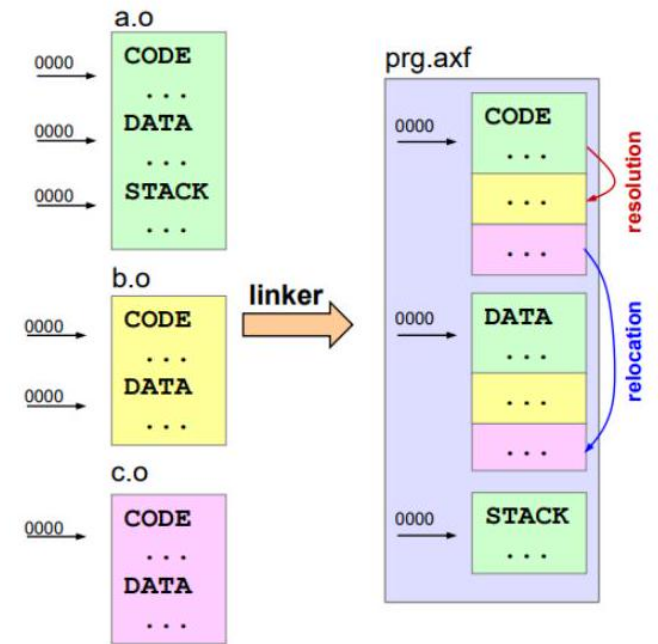
Module Interface Example

```
     ; 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
```

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

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

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
```

Linkage Types in C

Three types of linkage:
- **External Linkage**:
  - Global names available to all modules
  - Default for functions and global variables
  - Example:

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

- **Internal Linkage**:
  - Names only available within module
  - Created using 'static' keyword
  - Example:

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

- **No Linkage**:
  - Local variables and function parameters
  - Scope limited to block
  - Example:

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

Object File Structure   Example of complete object file:

```
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
```

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
```

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

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

Exceptional Control Flow

Exception Types
Two main categories of exceptions:
Interrupt Sources:
- Peripherals requesting immediate CPU attention
- Software-generated interrupts
- Asynchronous to instruction execution

System Exceptions:
- **Reset**: Processor restart
- **NMI**: Non-maskable Interrupt (cannot be ignored)
- **Faults**: Undefined instructions, errors
- **System Calls**: OS services (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()

Context Storage
Interrupt handling requires automatic context saving:
ISR Entry:
- Stores on stack:
  - xPSR, PC, LR, R12
  - R0-R3 (caller-saved registers)
- Stores EXC_RETURN in LR

ISR Exit:
- Via BX LR or POP ..., PC
- Restores from stack:
  - R0-R3, R12, LR, PC
 - xPSR

Interrupt Handling

Polling Approach

- Periodic status register checks
- Synchronous with main program
- **Advantages:**
  - Simple implementation
  - Predictable timing
  - No extra hardware needed
- **Disadvantages:**
  - CPU wastes time waiting
  - Reduced system throughput
  - Longer response times
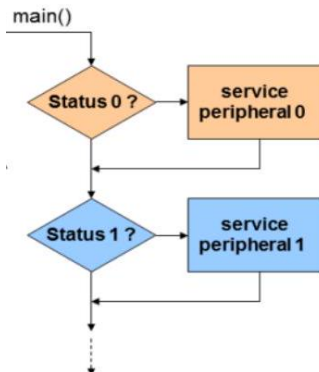
Interrupt Approach

- Hardware-triggered event handling
- Asynchronous to main program
- **Advantages:**
  - Efficient CPU usage
  - Quick response times
  - Better system throughput
- **Disadvantages:**
  - More complex implementation
  - Harder to debug
  - Timing less predictable
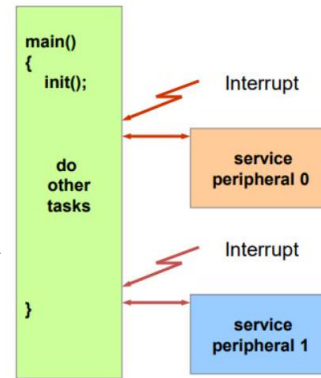
Basic ISR Implementation

```
    ; Interrupt Service Routine
    EXPORT MyISR
MyISR
    PUSH    {R4-R7, LR}    ; Save registers

    ; Handle interrupt here
    ; R0-R3 already saved automatically

    POP     {R4-R7, PC}    ; Restore and return
```

Implementing Interrupt Handlers
Steps for 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

NVIC (Nested Vectored Interrupt Controller)
Key components and functionality:
- **Interrupt States**:
  - **Inactive**: Not active and not pending
  - **Pending**: Waiting to be serviced
  - **Active**: Currently being serviced
  - **Active and Pending**: Being serviced with new request
- **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]
```

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

Example priority setting:

```
// Set priority for IRQ3
NVIC_SetPriority(IRQ3_IRQn, 2);

// Get priority
uint32_t prio = NVIC_GetPriority(IRQ3_IRQn);
```

Exception Vector Table

Setup and usage:
1. Vector table structure:

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

2. Handler implementation:

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

Nested Interrupts Example   Implementation with different priorities:

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

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

Example protection:

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

CMSIS Functions for Interrupt Control

Standard CMSIS functions for interrupt handling:
- `NVIC_EnableIRQ(IRQn)`: Enable specific interrupt
- `NVIC_DisableIRQ(IRQn)`: Disable specific interrupt
- `NVIC_SetPendingIRQ(IRQn)`: Set interrupt pending
- `NVIC_ClearPendingIRQ(IRQn)`: Clear pending status
- `NVIC_SetPriority(IRQn, priority)`: Set priority
- `NVIC_GetPriority(IRQn)`: Read priority

Example usage:

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

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

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
```
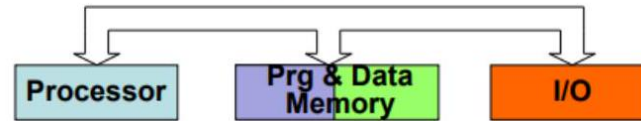
CISC (Complex Instruction Set Computer):
- More complex instruction set
- Lower memory usage for programs
- Potential performance gain for short programs
- More complex hardware required
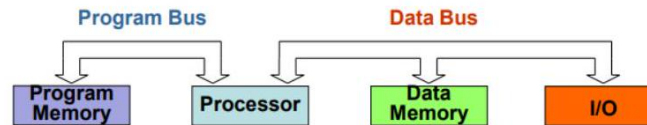
Computer Architectures

Von Neumann Architecture:
- Single memory for program and data
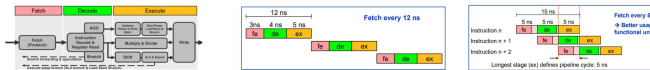- Single bus system between CPU and memory

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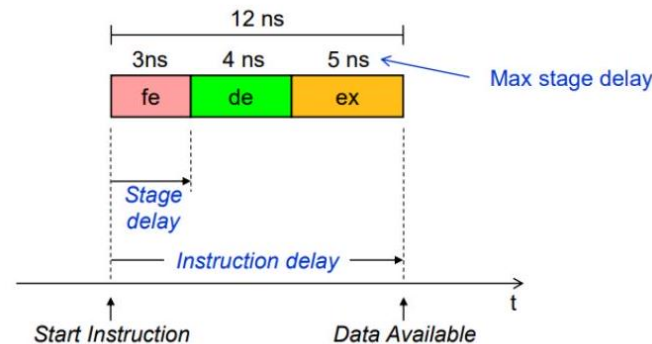- Originally from Harvard Mark I

Pipelining

Process of fetching next instruction while current one decodes:

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

Advantages:
- Uniform execution time per stage
- Significant performance improvement
- Simpler hardware per stage

Disadvantages:
- Blocking stages affect whole pipeline
- Memory access conflicts between stages

Pipeline Performance

Without pipelining:

$$\frac{\text{Instructions}}{\text{second}} = \frac{1}{\text{Instruction delay}}$$

With pipelining:

$$\frac{\text{Instructions}}{\text{second}} = \frac{1}{\text{Max stage delay}}$$

Note: Pipeline must be filled first

Pipeline Execution

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

| Cycle		1	2	3	4	5	6	7	8	9
Operation										
ADD		fe	de	ex						
SUB			fe	de	ex					
ORR				fe	de	ex				
AND					fe	de	ex			
ORR						fe	de	ex		
EOR							fe	de	ex	

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

Pipeline Hazards and Optimization

Control Hazards:
- Branch decisions in execute stage
- 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 ...         ...
```
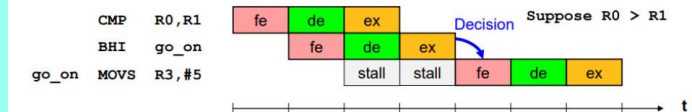

Optimization Techniques:
- Branch prediction based on history
- Instruction prefetch
- Out-of-order execution

Optimization Limits:
- Security vulnerabilities (Meltdown, Spectre)
- Complex optimizations increase risk

Parallel Computing

Different approaches to parallelism:
- **Vector Processing**: Single instruction processes multiple data
- **Multithreading**: Multiple threads share CPU
- **Multicore**: Multiple CPU cores on one chip
- **Multiprocessor**: Multiple CPUs in system

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

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

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

Pipeline Performance Calculation

For a processor with n pipeline stages:
Without pipelining:
- Time per instruction = Sum of all stage delays
- Performance = $\frac{1}{\text{Total delay}}$

With pipelining:
- Time per instruction = Longest stage delay
- Initial latency = n cycles
- Throughput = $\frac{1}{\text{Max stage delay}}$

Example 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   Three types of pipeline hazards:
1. **Structural Hazards**:

```
1  LDR   R0, [R1]     ; Needs memory access
2  LDR   R2, [R3]     ; Also needs memory access
3  ; Memory system can't handle both at once
```

2. **Data Hazards**:

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

3. **Control Hazards**:

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

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

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

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