

Computer Engineering

What is Computer Engineering?

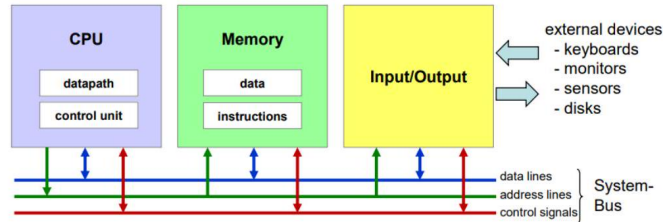
Computer Engineering is where microelectronics and software meet. It involves:

- Architecture and organization of computer systems
- Combines hardware and software to implement a computer
- Applications in embedded systems, information technology, and technical/scientific tools

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



CPU Components

The CPU contains several key components:

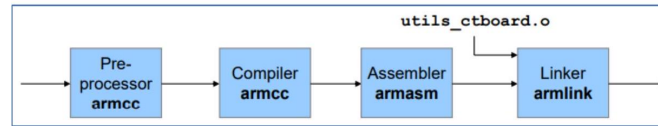
- **Core Registers**: Fast but limited storage inside CPU
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations
- **Control Unit**: Reads and executes instructions
- **Bus Interface**: Connects CPU to system bus

Memory Types

- **Main Memory (Arbeitsspeicher)**:
 - Connected through System-Bus
 - Access to individual bytes
 - Volatile: SRAM, DRAM
 - Non-volatile: ROM, Flash
- **Secondary Storage**:
 - Connected through I/O
 - Access to blocks of data
 - Non-volatile
 - Examples: HDD, SSD, CD, DVD

Program Translation Process

Translation from source code to executable involves four steps:



1. **Preprocessor**:
 - Text processing
 - Includes header files
 - Expands macros
 - Output: Modified source program
2. **Compiler**:
 - Translates C to assembly
 - CPU-specific code generation
 - Output: Assembly program
3. **Assembler**:
 - Converts assembly to machine code
 - Creates relocatable object file
 - Output: Binary object file
4. **Linker**:
 - Merges object files
 - Resolves dependencies
 - Creates executable program
 - Output: Executable file

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

Simple Program Translation - From Source to Executable

```

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

```

Translation steps:

1. Preprocessor expands include and replaces MAX with 100
2. Compiler converts to assembly language
3. Assembler creates object file
4. Linker combines with C library to create executable

Computer Engineering

What is Computer Engineering?

Computer Engineering is where microelectronics and software meet. It involves:

- 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

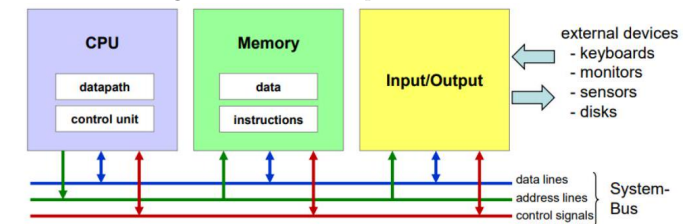
Early Computing Milestones

- 1943: First electronic computer ENIAC (18,000 tubes, 30 tons, 140 kW)
- 1947: First transistor at Bell Labs
- 1971: First microprocessor Intel 4004 (2300 transistors)
- 1978: Intel 8086 (29,000 transistors)
- Modern processors: Over 5 billion transistors

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:

- **Core Registers**: Fast but limited storage inside CPU
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations
- **Control Unit**:
 - Reads and executes instructions
 - Controls program flow
 - Manages instruction pipeline
- **Bus Interface**: Connects CPU to system bus

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")

Memory Types

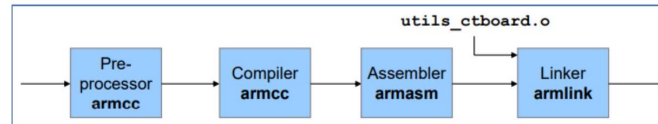
- **Main Memory (Arbeitsspeicher):**
 - Connected through System-Bus
 - Access to individual bytes
 - Volatile:
 - * SRAM (Static RAM) - faster, more expensive
 - * DRAM (Dynamic RAM) - needs refresh, cheaper
 - Non-volatile:
 - * ROM - factory programmed
 - * Flash - in-system programmable
- **Secondary 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

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

```
1 // source.c
2 #include <stdio.h>
3 #define MAX 100
4
5 int main(void) {
6     printf("Max is %d\n", MAX);
7     return 0;
8 }
```

After preprocessing (`.i`):

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

Assembly output (`.s`):

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

Core Registers

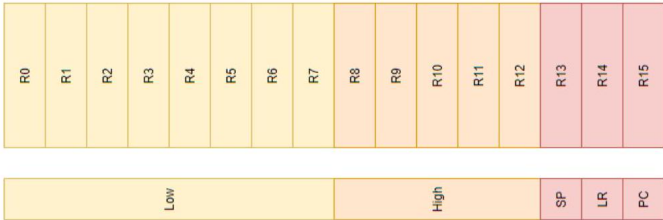
- The Cortex-M has 16 core registers, each 32-bit wide:
- **R0-R7**: Low registers - general purpose
 - **R8-R12**: High registers - general purpose
 - **R13 (SP)**: Stack Pointer - temporary storage
 - **R14 (LR)**: Link Register - return address from procedures
 - **R15 (PC)**: Program Counter - address of next instruction

ALU and Flags

- The Arithmetic Logic Unit (ALU) is 32-bit wide and supports:
- Arithmetic operations (add, subtract, multiply)
 - Logic operations (AND, OR, XOR)
 - Compare operations
 - Shift and rotate operations
- The Application Program Status Register (APSR) contains flags:
- **N**: Negative result
 - **Z**: Zero result
 - **C**: Carry from operation
 - **V**: Overflow occurred

Instruction Set

The Cortex-M uses 16-bit Thumb instructions:



Main instruction types:

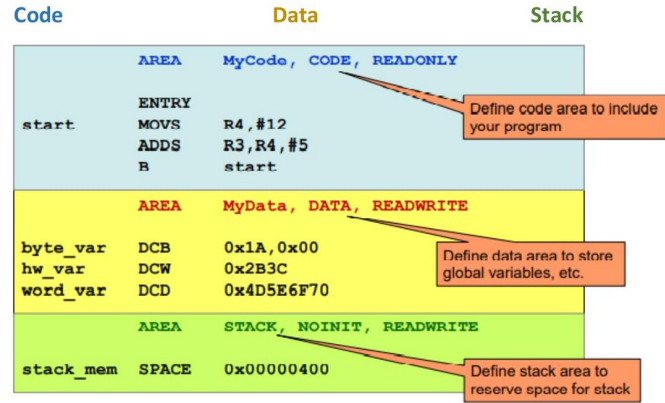
- **Data Transfer**: Move, Load, Store operations
- **Data Processing**: Arithmetic, logical, shift operations
- **Control Flow**: Branch and function calls

Basic Assembly Program Structure Example of a simple assembly program:

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

Assembly Program Sections

Program memory is organized in sections:



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 to create an assembly program:

1. Define program sections (CODE, DATA)
2. Declare any external symbols (IMPORT/EXPORT)
3. Define initialized data using DCx directives
4. Reserve uninitialized data using SPACE
5. Write program code using proper instruction syntax
6. End program with END directive

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

Core Registers

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

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

ALU and Flags

The Arithmetic Logic Unit (ALU) is 32-bit wide 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

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.

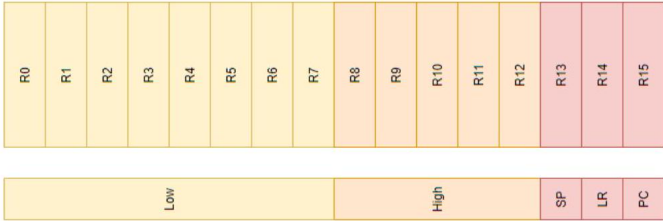
Flag Usage Examples

After arithmetic operations with 'S' suffix:

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

Instruction Set

The Cortex-M uses 16-bit Thumb instructions:



Main instruction types:

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

Common Instruction Formats

Register operations:

```
1 ADDS R0, R1, R2 ; R0 = R1 + R2
2 MOVS R0, R1 ; R0 = R1
3 ANDS R0, R1 ; R0 = R0 & R1
```

Immediate values:

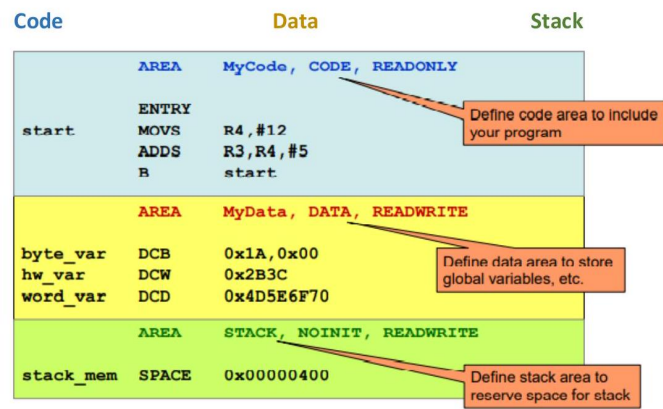
```
1 MOVS R0, #100 ; Load immediate value
2 ADDS R0, R0, #1 ; Add immediate
3 CMP R0, #10 ; Compare with
    immediate
```

Memory access:

```
1 LDR R0, [R1] ; Load from memory
2 STR R0, [R1, #4] ; Store with offset
3 LDRB R0, [R1] ; Load byte
```

Assembly Program Sections

Program memory is 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:

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

Writing Assembly Programs

Steps for creating an assembly program:

1. Setup sections:

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

2. Declare external symbols:

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

3. Define data:

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

4. Write code:

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

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
```


Common Assembly Patterns

1. Loop with counter:

```
1  MOVS    R0, #0           ; Initialize counter
2  loop
3      ; Loop body
4      ADDS    R0, #1         ; Increment
5      CMP     R0, #10        ; Check condition
6      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
5          increment
6      SUBS    R2, #1          ; Decrement counter
7      BNE     copy_loop      ; Continue if not
8      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
```

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
- Various addressing modes for flexible memory access

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
- **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]
- **LDRB** (Load Register Byte):
 - Loads 8-bit value
 - Bits 31 to 8 are set to zero
- **LDRH** (Load Register Half-word):
 - Loads 16-bit value
 - Bits 31 to 16 are set to zero

Store Instructions

Instructions for storing data from registers to memory:

- **STR** (Store Register):
 - Basic store: STR R1, [R2]
 - With offset: STR R1, [R2, #0x04]
- **STRB** (Store Register Byte):
 - Stores lowest 8 bits of register
- **STRH** (Store Register Half-word):
 - Stores lowest 16 bits of register

Memory Access Example

Loading and storing array elements:

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

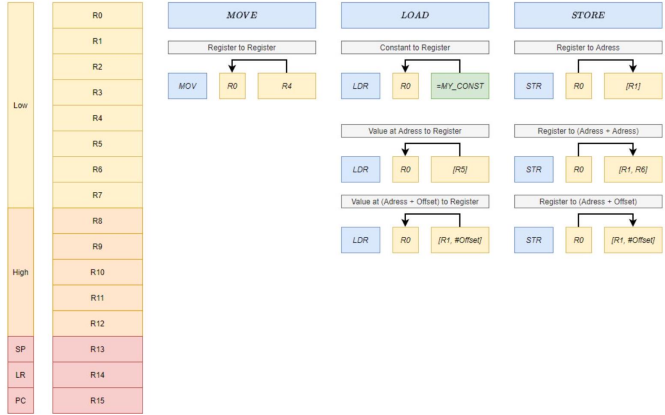
AREA myCode, CODE, READONLY
. . .

; load base and offset registers
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
```

Memory Layout Example

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

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

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
```

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
- 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

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/LDRH** (Load Byte/Half-word):
 - Zero extension to 32 bits
 - Common for arrays of bytes/shorts
- **LDRSB/LDRSH** (Load Signed 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
5 ; LDR examples
6 LDR R1, =0x12345678 ; Load 32-bit
7 ; constant
8 LDR R2, [R1] ; Load from address
9 ; in R1
10 LDR R3, [R1, #4] ; Load with offset
11 LDR R4, [R1, R2] ; Load with
12 ; register offset
13
14 ; Byte/Half-word loads
15 LDRB R1, [R2] ; Load unsigned byte
16 LDRSB R1, [R2] ; Load signed byte
17 LDRH R1, [R2] ; Load unsigned
18 ; half-word
19 LDRSH R1, [R2] ; Load signed
20 ; 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

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 LSL R2, R1, #1 ; R2 = i * 2
9 LDRH R3, [R0, R2] ; R3 = array[i]
10
11 ; For word array
12 LSL R2, R1, #2 ; R2 = i * 4
13 LDR R3, [R0, R2] ; R3 = array[i]
```

Multiple Data Transfer Loading/Storing multiple registers:

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

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

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
5 ; increment
6 SUBS R2, #1 ; Decrement counter
7 BNE loop ; Continue if not
8 ; zero
```

2. Initialize memory block:

```
1 ; R0 = start, R1 = value, R2 = count
2 loop
3 STR R1, [R0], #4 ; Store and
4 ; increment
5 SUBS R2, #1 ; Decrement counter
6 BNE loop ; Continue if not
7 ; 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
8 ; zero
9 not_found
10 ; Handle not found case
11 found
12 ; Handle found case
```

Load/Store vs Register Memory Architecture

Two main approaches to memory access:

- **Load/Store Architecture (ARM Cortex-M):**
 - Memory accessed only with load/store operations
 - Data processing only between registers
 - Steps: Load operands → Execute → Store result
- **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

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.

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
```

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 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

Arithmetic Operations

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

Instructions ending with 'S' modify these flags:

- ADDS, SUBS, MOVS, LSLs

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 ($A \cdot B$)

Two's Complement

For negative numbers:

- Two's complement: $A = !A + 1$
- Used for representing signed numbers
- Enables using same hardware for addition and subtraction

Carry and Overflow

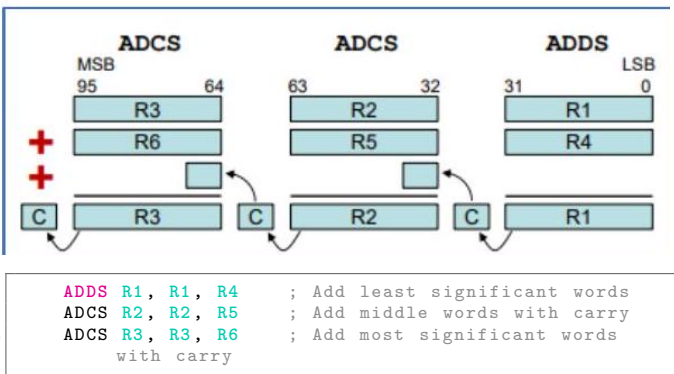
Unsigned Operations:

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

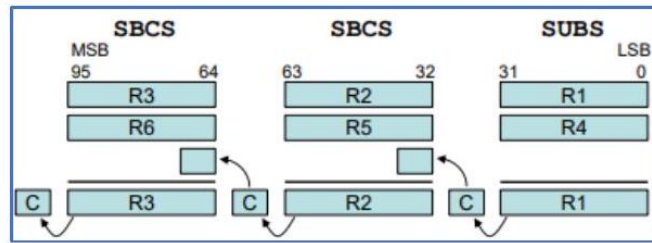
Signed Operations:

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

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



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



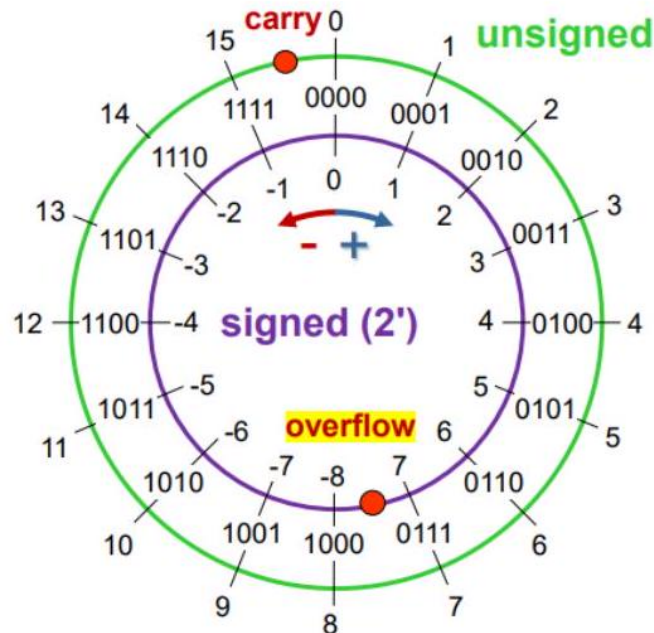
```
1 SUBS R1, R1, R4 ; Subtract least significant words
2 SBCS R2, R2, R5 ; Subtract middle words with borrow
3 SBCS R3, R3, R6 ; Subtract most significant words with borrow
```

Addition and Subtraction Examples Addition with carry (13d + 7d):

```
1101 (13d)
0111 (7d)
----
1 0100 (20d = 16d + 4d)
```

Subtraction with borrow (6d - 14d):

```
0110 (6d)
+ 0010 (TC of 14d)
----
1000 (8d - 16d = -8d)
```



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

ALU Operation Fundamentals

The Arithmetic Logic Unit (ALU) processes data:

- 32-bit wide data processing unit
- Fixed point operations only (no floating point)
- Flag outputs indicate operation results
- Can perform both signed and unsigned operations
- Same hardware used for addition and subtraction

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**
 - $Rd = Rd + \text{immediate}$
 - 8-bit immediate value only

Example encodings:

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

Subtraction Operations

Subtraction instructions and their uses:

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

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
```

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
4 set
5 ; Signed overflow detection
6 ADDS R0, R1 ; Perform addition
7 BVS overflow ; Branch if
8 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)
```

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
- Two's complement:
 - Invert all bits (one's complement)
 - Add 1 to result
 - Enables using addition hardware for subtraction

Example for 4-bit numbers:

Positive: 0000 to 0111 (0 to 7)

Negative: 1000 to 1111 (-8 to -1)

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

Multiplication

Simple multiplication examples:

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

Logic, Shift and Rotate Instructions

Logic Instructions

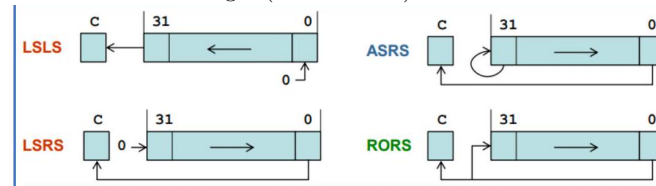
Base logic operations (affect only N and Z flags):

- **ANDS**: Bitwise AND ($R_{dn} \& R_m, a \& b$)
- **BICS**: Bit Clear ($R_{dn} \& !R_m, a \& b$)
- **EORS**: Exclusive OR ($R_{dn} \oplus R_m, a \wedge b$)
- **MVNS**: Bitwise NOT ($!R_m, a$)
- **ORRS**: Bitwise OR ($R_{dn} \# R_m, a \vee b$)

Shift and Rotate Instructions

Shift operations for binary manipulation:

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



Integer Casting

Extension (adding bits):

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

Truncation (removing bits):

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

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)

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
```

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 MOVBS R1, #0x11 ; Mask: 0001 0001
3 ORRS R0, R1 ; Set bits in R0
```

2. Clear specific bits:

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

3. Toggle specific bits:

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

4. Test specific bits:

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

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
8 (unsigned)
9 ASRS R0, R0, #1 ; R0 = R0 / 2
10 (signed)
11
12 ; Multiply by 10 (8 + 2)
13 LSLS R1, R0, #3 ; R1 = R0 * 8
14 ADDS R0, R0, R1 ; R0 = R0 + (R0 *
8) = R0 * 9
15 ADDS R0, R0, R0 ; R0 = R0 * 2 = R0
* 10
```

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

Type Conversion Examples Examples of common type conversions:

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

Example:

```
1 ; Convert 8-bit to 32-bit
2 MOVBS R0, #0xFF ; Load 8-bit value
3 SXTB R1, R0 ; Signed extension
4 UXTB R2, R0 ; Unsigned extension
5
6 ; Truncate 32-bit to 8-bit
7 MOVBS R1, #0xFF ; Create mask
8 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

Control Structures

Branch Instructions

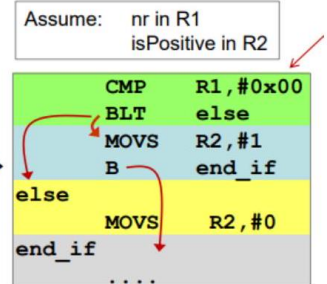
Branch instructions control program flow:

Unconditional Branches	Conditional Branches	Overview
<ul style="list-style-type: none">• B (immediate)<ul style="list-style-type: none">• Direct• Relative• BX (Branch and Exchange)<ul style="list-style-type: none">• Branch and Exchange• Indirect• Absolute	<ul style="list-style-type: none">• Flag-dependent and arithmetic branches<ul style="list-style-type: none">• Indirect• Absolute	<p>Type</p> <ul style="list-style-type: none">• Unconditional: Branch always• Conditional: Branch if condition is met <p>Address hand-over</p> <ul style="list-style-type: none">• Direct: Target addresses part of instruction• Indirect: Target address in register <p>Address of target</p> <ul style="list-style-type: none">• Relative: Target address relative to PC• Absolute: Absolute address <p>Compare and Test</p> <ul style="list-style-type: none">• TST: AND without changing the value• CMP: SUBS without changing the value

Selection (IF-ELSE)

```
int32_t nr;
int32_t isPositive;
```

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



Switch Statement Implementation C code example:

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

Assembly implementation with jump table:

```
1 NR_CASES EQU 6
2 case_switch CMP R1, #NR_CASES
3             BHS case_default
4             LSLS R1, #2 ; * 4
5             LDR R7, =jump_table
6             LDR R7, [R7, R1]
7             BX R7
8
9 case_0 ADDS R2, R2, #17
10      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 MOVSW 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:

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

Do-While (Post-Test Loop):

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

While (Pre-Test Loop):

C	Assembly
<pre>#include <utils_ctboard.h> #include <stdint.h> ... int32_t = 0; int32_t count = 0; for(i = 0; i < 10; i++) { count++; }</pre>	<pre>AREA pro THUMB PROC EXPORT m main LDR R6, =0 LDR R0, =0 LDR R7, =10 LDR R1, =0 B CC loop ADDS R0, R0, #1 ADDS R1, R1, #1 cond CMP R0, R7 BLT loop STR R1, [R6, #0] endless B ENDP i count AREA pro DCD 0 DCD 0 END</pre>

For Loop (Pre-Test Loop):

Implementing Control Structures

Steps for implementing control structures:

- Choose appropriate control structure:
 - If-then-else for simple decisions
 - Switch for multiple cases with same variable
 - Loops for repeated operations
- For switches:
 - Create jump table
 - Calculate offset based on case value
 - Handle default case
- For loops:
 - Initialize counter/condition
 - Place condition check appropriately
 - Ensure proper exit condition
 - Update variables correctly

Basic Control Structures Example implementations:

```
1 ; If-then-else
2 CMP R0, #0 ; Compare value
3 BEQ else_label ; Branch if equal
4 ; then code
5 B endif_label
6 else_label
7 ; else code
8 endif_label
9
10 ; While loop
11 B while_cond ; Jump to condition
12 while_loop
13 ; loop body
14 while_cond
15 CMP R0, #10 ; Check condition
16 BLT while_loop ; Branch if less than
17
18 ; Do-while loop
19 do_loop
20 ; loop body
21 CMP R0, #10 ; Check condition
22 BLT do_loop ; Branch if less than
```

Branch Instruction Types

Classification of branch instructions:

1. Based on Condition:

• Unconditional:

- B - Branch always
- BL - Branch with Link
- BX - Branch and Exchange

• Conditional:

- Flag-dependent (EQ, NE, CS, CC, etc.)
- Arithmetic (HI, LS, GE, LT, etc.)

2. Based on Target Address:

- Direct:** Target address in instruction
- Indirect:** Target address in register
- Relative:** Offset from current PC
- Absolute:** Complete target address

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
```

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
```

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
```

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
18      loop
19
20      ADDS    R2, R0, R1      ; Calculate i + j
21      ADDS    R4, R4, R2      ; Add to sum
22
23      ADDS    R1, #1          ; j++
24      CMP     R1, #3          ; Check j < 3
25      BLT     inner_loop      ; Continue inner
26      loop
27
28      ADDS    R0, #1          ; i++
29      CMP     R0, #5          ; Check i < 5
30      BLT     outer_loop      ; Continue outer
31      loop
```

Subroutines and Stack

Subroutine Basics

Key elements of subroutines:

- Label to identify subroutine entry point
- Return instruction (BX LR) to exit
- Proper register management

Simple Subroutine 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
```

Stack Operations

Stack characteristics:

- **Stack Area:** 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

```
ADDR_LED_31_0 EQU 0x60000100
LED_PATTERN EQU 0xA55A5AA5

subrExample    PUSH    {R4,R5,LR}    ; Save LR and registers used
                                           by subroutine
                                           ; write pattern to LEDs
                                           LDR     R4,=ADDR_LED_31_0
                                           LDR     R5,=LED_PATTERN
                                           STR     R5,[R4]
                                           BL      write7seg
                                           POP     {R4,R5,PC}    ; Restore registers and PC
```

Stack Operations Implementation PUSH 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
```

POP implementation:

```
1      ; POP {R2,R3,R6}
2      LDR     R2, [SP]             ; Restore R2
3      LDR     R3, [SP, #4]         ; Restore R3
4      LDR     R6, [SP, #8]         ; Restore R6
5      ADD     SP, SP, #12          ; Free stack space
```

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

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
7     address
8     BL      proc_b            ; Call proc_b
9     POP     {PC}              ; Return to main
10
11 proc_b
12     PUSH    {LR}              ; Save return
13     address
14     BL      proc_c            ; Call proc_c
15     POP     {PC}              ; Return to proc_a
16
17 proc_c
18     ; Do something
19     BX      LR                ; Return to proc_b
```

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 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

Function Implementation Patterns

Common implementation patterns:

1. Simple function:

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

2. Function with locals:

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

3. Function with parameters:

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

Stack Frame 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

Parameter Passing

Parameter Passing Methods

Data can be passed between functions through:

- **Registers:** Fast, limited number available
- **Global Variables:** Shared memory space
- **Stack:**
 - Caller: PUSH parameters onto stack
 - Callee: Access via LDR from stack

ARM Procedure Call Standard

Parameter Passing:

- First four arguments use R0-R3
- Additional parameters go on stack

Return Values:

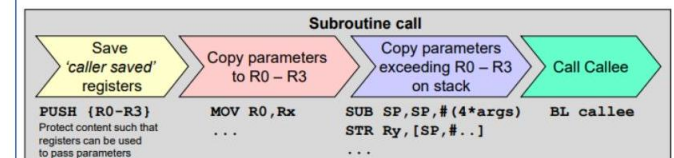
- **Small Values** (≤ 32 bits):
 - Return in R0
 - Zero/sign extend if needed
- **Double Word** (64 bits): R0/R1
- **128-bit Values:** R0-R3
- **Larger Values:**
 - Store in memory
 - Return pointer in R0

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

Subroutine Call – Caller Side

Pattern as used by the compiler. Manually written assembly code may be slightly different.



Reentrancy

Handling recursive function calls:

- Each call needs its own data set
- Registers/globals get overwritten
- Solution: Use stack for local storage

Parameter Passing Methods Global variable approach (not recommended):

```
1 .data
2 value DCD 0 ; Global variable
3
4 .text
5 func LDR R0, =value ; Load address
6      LDR R1, [R0] ; Get value
7      ; Process value
8      STR R1, [R0] ; Store result
```

Register-based approach (preferred):

```
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
```

Implementing Function Calls

Steps for calling functions:

- 1. Caller’s responsibilities:
 - Place parameters in R0-R3
 - Push additional parameters on stack
 - Save caller-saved registers if needed
- 2. 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

Parameter Passing by Value vs. Reference

Two main approaches:

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

Example implementation:

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

Data Structure Access Working with structures and arrays:

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

Assembly implementation:

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

Stack Frame Organization

Complete stack frame layout:

- Previous Stack Frame:
 - Local variables
 - Saved registers
- Current Frame:
 - Arguments 5+
 - Return address (LR)
 - Saved registers (R4-R11)
 - Local variables
 - Temporary storage
- Next Frame:
 - Space for called functions

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     MOV 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
```

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
5         ; Save parameters
6         MOV     R4, R0      ; Save a
7         MOV     R5, R1      ; Save b
8         MOV     R6, R2      ; Save c
9         ; R3 contains d
10        LDR     R0, [SP, #16] ; Load e from
11                stack
12
13        ; Function body
14
15        POP     {R4-R6, PC} ; Return
```

Complex Parameter Example Function with mixed parameter types:

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

Assembly implementation:

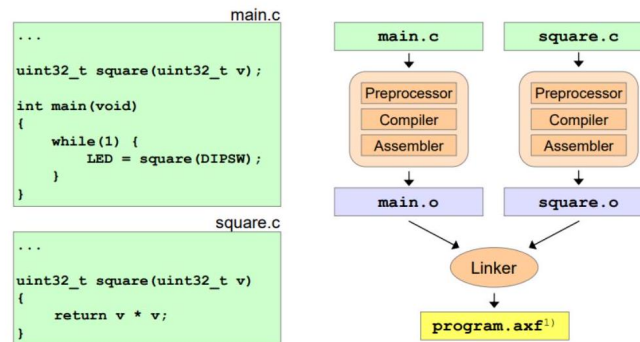
```
1 ; R0 = point_t* p
2 ; R1 = scale
3 ; R2 = result pointer
4 calculate
5     PUSH    {R4-R5, LR} ; Save registers
6
7     ; Load structure members
8     LDR     R4, [R0, #0] ; Load p->x
9     LDR     R5, [R0, #4] ; Load p->y
10
11    ; Perform calculation
12    MULS    R4, R1, R4    ; x * scale
13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

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

```
1 ; main.s
2
3 AREA myCode, CODE, READONLY
4 EXPORT main
5 IMPORT square
6
7 main PROC
8     LDR     r0, a_adr
9     LDR     r0, [r0, #0] ; a
10    BL      square
11
12    ...
13    ENDP
14
15 a_adr DCD    a
16 b_adr DCD    b
17
18
19 AREA myData, DATA
20
21 a     DCD    0x00000005
22 b     DCD    0x00000007
```

Object Files

ELF format contains:

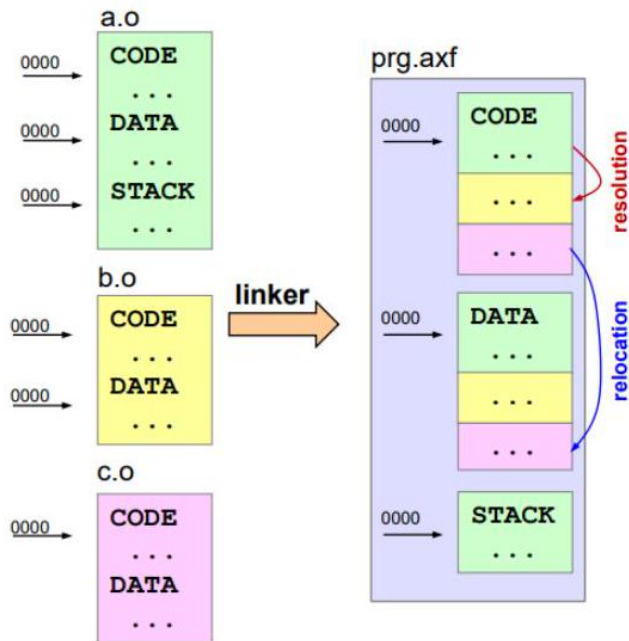
- **Code Section:**
 - Program code and constants
 - Based at address 0x0
- **Data Section:**
 - Global variables
 - Based at address 0x0
- **Symbol Table:**
 - All symbols and their attributes
 - Global/local status
 - References to external symbols
- **Relocation Table:**
 - Instructions for adjusting 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

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

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:

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

2. Relocation:

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

Linkage Types in C

Three types of linkage:

• External Linkage:

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

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

• Internal Linkage:

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

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

• No Linkage:

- Local variables and function parameters
- Scope limited to block
- Example:

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

Object File Structure Example of complete object file:

```
1 File sections:
2 1. '.text' section (Code):
3 0x00000000: 4604 MOV    r4,r0
4 0x00000002: 0040 LSLS   r0,r0,#1
5 0x00000004: 4420 ADD     r0,r4
6
7 2. '.data' section:
8 0x00000000: Initial values for global data
9
10 3. Symbol table:
11 # Name      Value      Type      Binding
12 6 myFunc     0x0000  CODE     Global
13 7 extVar     0x0000  DATA    Reference
14
15 4. Relocation entries:
16 Offset      Type          Symbol
17 0x0006      R_ARM_REL32  extVar
```

Library Creation and Use

Steps for creating and using libraries:

1. Create library source files:

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

2. Compile to object files:

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

3. Create static library:

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

4. Link with library:

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

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

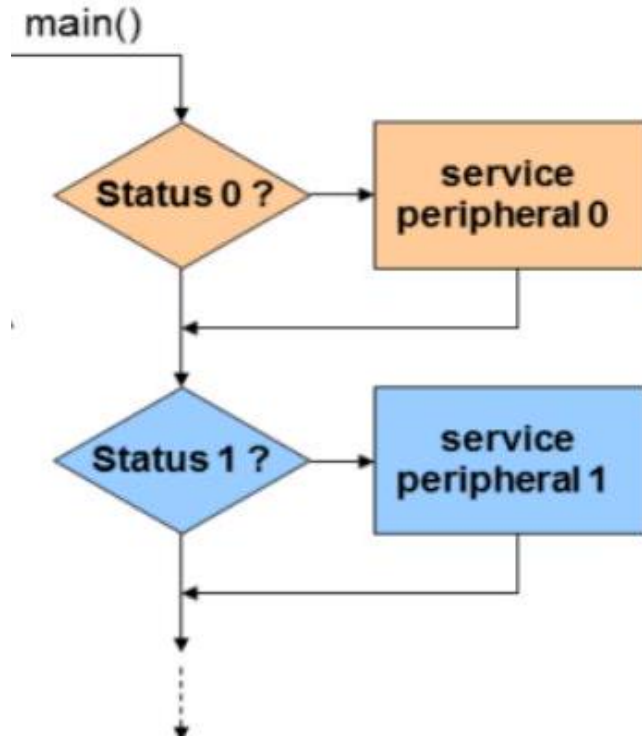
ISR Exit:

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

Polling vs Interrupts

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

```
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
```

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:

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

2. Pending Registers:

```
1 SETPEND0 EQU 0xE000E200 ; Set pending
2 CLRPEND0 EQU 0xE000E280 ; Clear pending
3
4 ; Set IRQ3 pending
5 LDR R0, =SETPEND0
6 MOVS R1, #(1<<3)
7 STR R1, [R0]
8
9 ; Clear IRQ3 pending
10 LDR R0, =CLRPEND0
11 MOVS R1, #(1<<3)
12 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:

```
1 // Set priority for IRQ3
2 NVIC_SetPriority(IRQ3_IRQn, 2);
3
4 // Get priority
5 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
16void IRQ0_Handler(void) {
17    // Handle high priority interrupt
18    // Can't be interrupted by IRQ1
19}
20
21// Lower priority ISR
22void 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
3     section start
4     shared_var++;    // Update shared
5     data
6     __enable_irq();  // Critical section
7     end
8 }
```

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}
```

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

■ RISC

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

■ CISC

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

Example: **Balance = Balance + Credit**

```

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

```

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

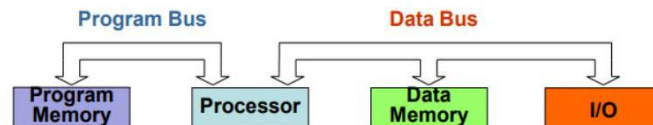
Von Neumann Architecture:

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



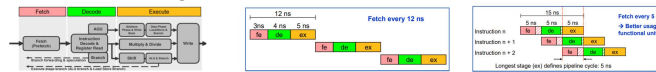
Harvard Architecture:

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



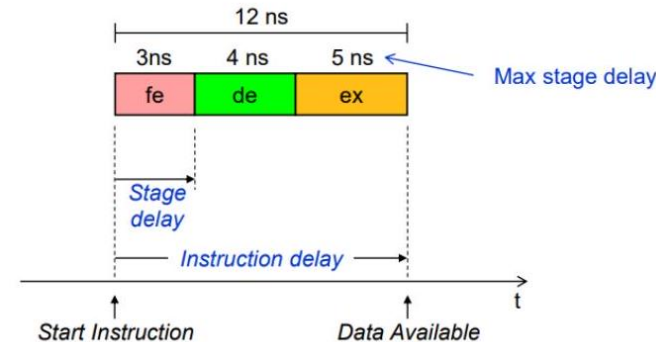
Pipelining

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

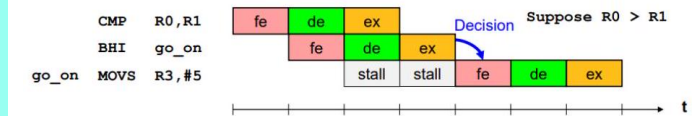
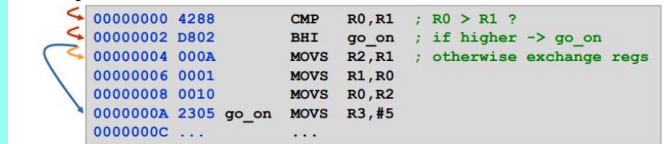
LDR Special Case:

- 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



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
- $Performance = \frac{1}{Total\ delay}$

With pipelining:

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

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

