Computertechnik

Jil Zerndt, Lucien Perret
January 2025

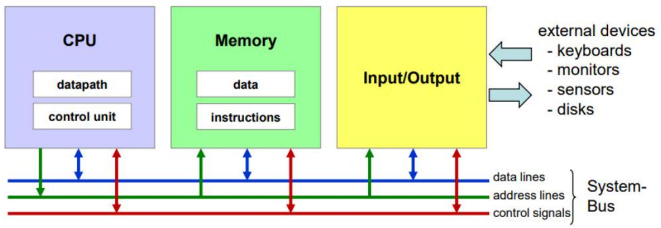
Computer Engineering

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

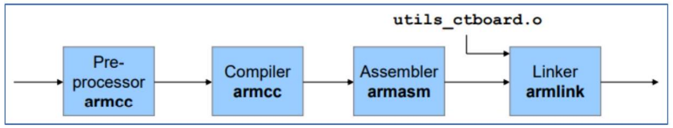
Memory
A set of storage cells and the smallest addressable unit is a byte.
2^N addresses:
- RAM (Random Access Memory): read/write
- ROM (Read-Only Memory): read-only

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

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

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

After preprocessing (.i):

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

Assembly output (.s):

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

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

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

Cortex-M Architecture

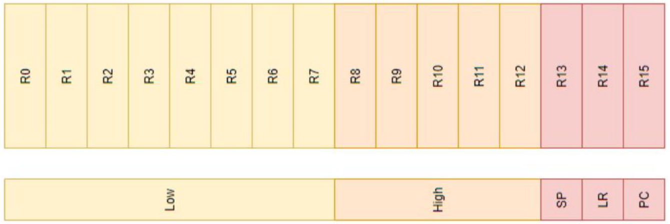
Core Architecture Overview

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

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

Registers

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



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

Arithmetic Logic Unit (ALU)

32-bit wide processing Unit and supports:

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

APSR (Flag Register)

The Application Program Status Register (APSR) contains flags:

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

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

Flag Usage Examples

After arithmetic operations with 'S' suffix:

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

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

MOVS    R0, #5
SUBS    R0, #10       ; R0 = -5, N=1, C=0 (borrow)
```

Instruction Set

The Cortex-M uses 16-bit Thumb instructions:

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

Main instruction types:

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

Common Instruction Formats

Register operations:

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

Basic Assembly Program Structure

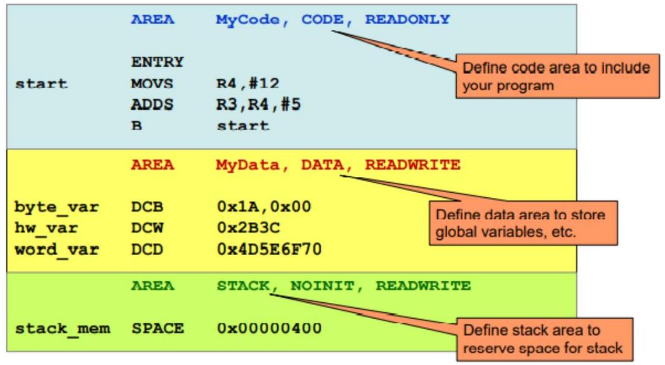
Example of a simple assembly program:

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

Assembly Program Sections

Program memory organized in sections:

Code **Data** **Stack**



Code Section (CODE):

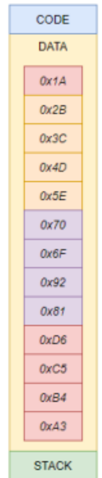
- Contains program instructions
- Usually read-only
- Placed in Flash memory
- Can contain constants (literal pool)

Data Section (DATA):

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

Stack Section: (STACK)

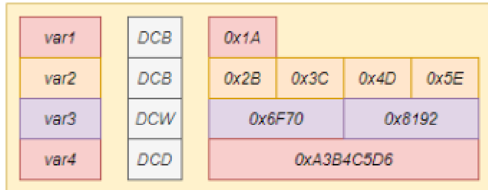
- Dynamic memory allocation
- Used for local variables
- Function call management
- Grows downward in memory



Initialized vs uninitialized Data

Directives for initialized data:

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



Directive for uninitialized data:

- **SPACE**: Reserve specified number of bytes

Data Definition Memory layout for different data types:

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

Creating Assembly Programs Steps for creating an assembly program:

1. Define program sections: (CODE, DATA)

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

2. Declare external symbols: (IMPORT/EXPORT)

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

3. Define data:

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

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

4. Write program code using proper instruction syntax:

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

5. End program with END directive!

Common Assembly Patterns

1. Loop with counter:

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

2. Memory copy:

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

3. Function call with parameters:

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

Complete Program Example
Program to sum array elements:

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

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

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

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

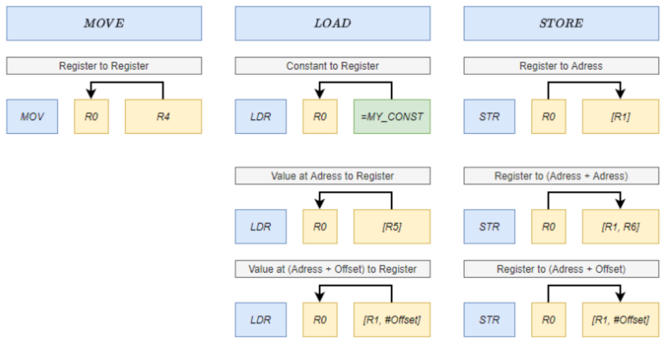
Store Instructions

Instructions for storing data from registers to memory:

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

Memory Access

Memory Layout for array elements and instructions:



Size considerations:

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

Memory Access Patterns in Load/Store Architecture (ARM Coretx-M)

Steps for accessing memory:

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

Memory Alignment Important alignment rules:

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

Basic Data Transfer Operations Common data transfer operations:

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

;Store operations
STR  R1, [R2]       ;Store word
STRB R1, [R2, #4]   ;Store byte with offset
STRH R1, [R2, R3]   ;Store half-word with register offset
```

Common Data Transfer Patterns

1. Copy memory block:

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

2. Initialize memory block:

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

3. Search memory:

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

Arrays

Memory Access Loading and storing array elements:

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

```
                   AREA    myCode, CODE, READONLY
                   . . .

                   ; load base and offset registers
0000007C 4906      LDR     R1,=my_array    ; load address of array
0000007E 4B07      LDR     R3,=0x08

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

Not content of my_array, but address of my_array

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

Example implementation:

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

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

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

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

Multiple Register Transfer

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

Example:

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

Multiple Data Transfer Loading/Storing multiple registers:

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

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

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

; Store multiple memory locations
STM     R0!, {R1-R4}     ; Store 4 words, update R0
```

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

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

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

2. Storing 96-bit value:

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

    ; Alternative using individual stores:
    STR     R1, [R6]         ; Store LSW
    STR     R2, [R6, #4]     ; Store middle word
    STR     R3, [R6, #8]     ; Store MSW
```

Stack Operations

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

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

Pseudo Instructions

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

Example:

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

Pseudo Instruction vs Direct Load The difference between LDR forms:

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

mylita  DCD     0xFF001122   ; Data definition
```

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

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:

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

Subtraction Operations

Subtraction instructions and their uses:

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

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

Multiplication

Simple multiplication examples:

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

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: $A = !A + 1$
- 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)

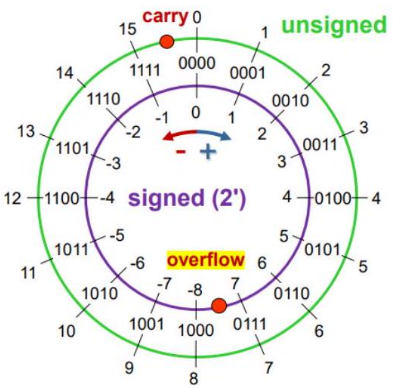
Signed Operations:

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

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:

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

2. Subtraction sequence:

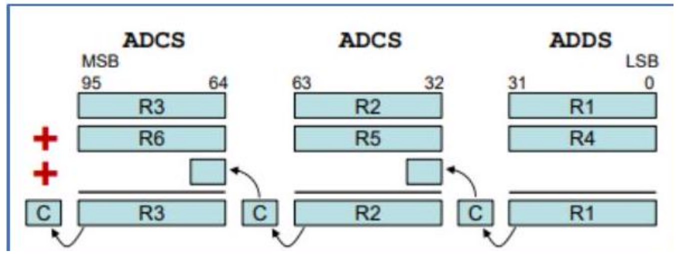
```
    ; 64-bit subtraction (R1:R0 - R3:R2)
    SUBS    R0, R2              ; Subtract low words
    SBCS    R1, R3              ; Subtract high words with
        borrow
```

3. Important considerations:

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

Multi-Word Addition

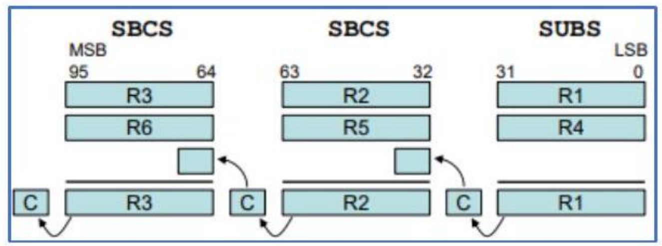
Adding 96-bit values using ADCS:




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

Multi-Word Subtraction

Subtracting 96-bit values using SBCS:




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

Flag Usage and Overflow Detection

Processor Status Flags

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

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

Instructions ending with 'S' modify these flags:

- ADDS, SUBS, MOVS, LSLS

Overflow Detection

Steps to detect overflow in arithmetic operations:

1. For unsigned arithmetic (using C flag):

- Addition: Check C flag (C=1 means overflow)
- Subtraction: Check C flag (C=0 means underflow)

2. For signed arithmetic (using V flag):

- Addition: Check V flag for same-sign operands
- Subtraction: Check V flag for opposite-sign operands

Example:

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

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

Flag Usage Examples of flag behavior:

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

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

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

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

Logic, Shift and Rotate Instructions

Logic Instructions

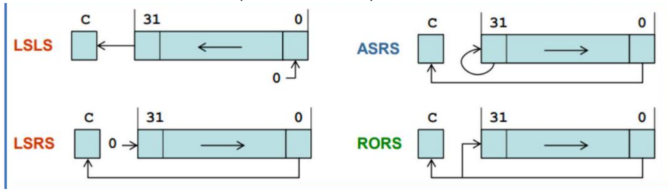
Base logic operations (affect only N and Z flags):

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

Shift and Rotate Instructions

Shift operations for binary manipulation:

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



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

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:

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

; Shift operations
LSLS R0, R1, #2      ; R0 = R1 << 2 (multiply by 4)
LSRS R0, R1, #1      ; R0 = R1 >> 1 (divide by 2)
ASRS R0, R1, #2      ; R0 = R1 >> 2 (signed divide by 4)
RORS R0, R1, #1      ; Rotate R1 right by 1 bit
```

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:

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

2. Clear specific bits:

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

3. Toggle specific bits:

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

4. Test specific bits:

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

Shift Operations for Arithmetic

Using shifts for multiplication and division:

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

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

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

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:

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

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

    ; Manual sign extension
    LSLS    R0, R0, #24     ; Shift left 24 bits
    ASRS    R0, R0, #24     ; Arithmetic shift right 24
```

Type Conversion Guidelines

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

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:

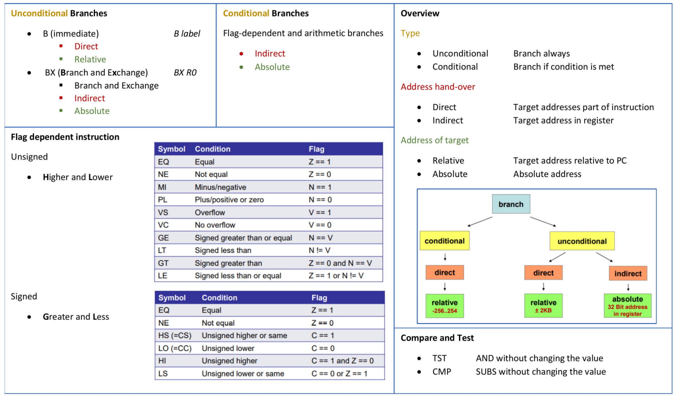
```
    ; Convert 8-bit to 32-bit
    MOVS    R0, #0xFF       ; Load 8-bit value
    SXTB    R1, R0          ; Signed extension
    UXTB    R2, R0          ; Unsigned extension

    ; 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

Control Structures

Branch Instructions

Branch instructions control program flow:



Unconditional Branches

- B (immediate) — *B label*
  - Direct
  - Relative
- BX (Branch and Exchange) — *BX R0*
  - Branch and Exchange
  - Indirect
  - Absolute

Conditional Branches

Flag-dependent and arithmetic branches

- Indirect
- Absolute

Overview

Type

- Unconditional — Branch always
- Conditional — Branch if condition is 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 — Absolute address

Compare and Test

- TST — AND without changing the value
- CMP — SUBS without changing the value

Flag dependent instruction

Unsigned

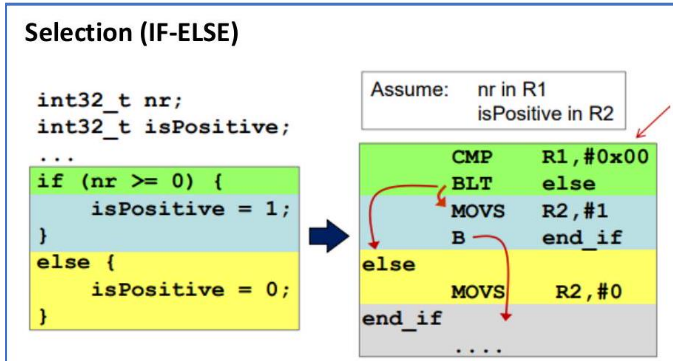
- Higher and Lower

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



Switch Statement Implementation

C code example:

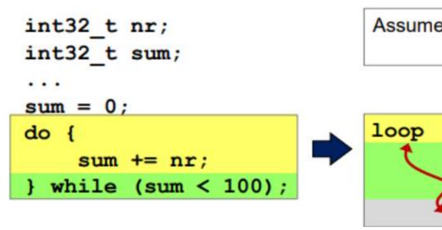
```c
uint32_t result, n;
switch (n) {
    case 0:
        result += 17;
        break;
    case 1:
        result += 13;
        //fall through
    case 3:
    case 5:
        result += 37;
        break;
    default:
        result = 0;
}
```

Assembly implementation with jump table:

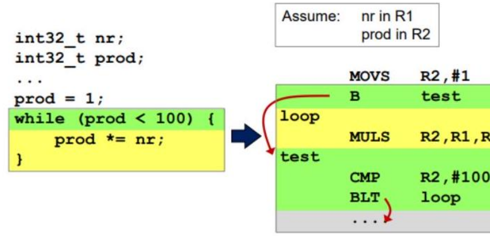
```
NR_CASES     EQU     6
case_switch  CMP     R1, #NR_CASES
             BHS     case_default
             LSLS    R1, #2          ; * 4
             LDR     R7, =jump_table
             LDR     R7, [R7, R1]
             BX      R7

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

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

Loop Types

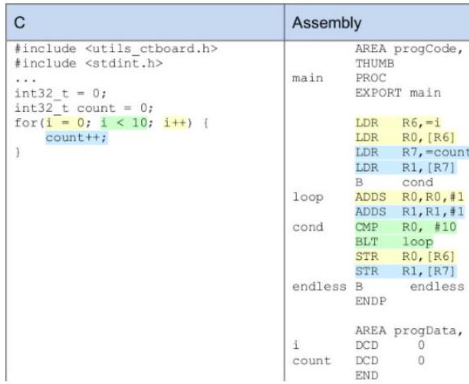
Three main types of loops:



Do-While (Post-Test Loop):



While (Pre-Test Loop):



For Loop (Pre-Test Loop):

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

Basic Control Structures

Example implementations:

```
    ; If-then-else
    CMP     R0, #0      ; Compare value
    BEQ     else_label  ; Branch if equal
    ; then code
    B       endif_label
else_label
    ; else code
endif_label

    ; While loop
    B       while_cond  ; Jump to condition
while_loop
    ; loop body
while_cond
    CMP     R0, #10     ; Check condition
    BLT     while_loop  ; Branch if less than

    ; Do-while loop
do_loop
    ; loop body
    CMP     R0, #10     ; Check condition
    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:

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

2. if-then-else:

```
    ; if (x > y) { x = y; } else { y = x; }
    CMP     R0, R1          ; Compare x and y
    BLE     else_part       ; Branch if x <= y
    MOVS    R0, R1          ; Then part: x = y
    B       endif           ; Skip else part
else_part
    MOVS    R1, R0          ; Else part: y = x
endif
```

3. Nested if:

```
    ; if (x > 0) {
    ;     if (y > 0) {
    ;         x = y;
    ;     }
    ; }
    CMP     R0, #0          ; Check x > 0
    BLE     endif_outer
    CMP     R1, #0          ; Check y > 0
    BLE     endif_inner
    MOVS    R0, R1          ; x = y
endif_inner
endif_outer
```

Loop Implementation

Templates for different loop types:

1. While loop:

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

2. Do-while loop:

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

3. For loop:

```
    ; for (i = 0; i < 10; i++)
    MOVS    R0, #0          ; i = 0
    B       for_cond
for_loop
    ; Loop body
    ADDS    R0, #1          ; i++
for_cond
    CMP     R0, #10         ; Check i < 10
    BLT     for_loop        ; Continue if true
```

Switch Implementation

Steps for implementing switch statements:

1. Range check and table access:

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

2. Jump table structure:

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

3. Case handlers:

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

Complex Control Structure Implementing nested loops with conditions:

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

    MOVS    R0, #0          ; i = 0
outer_loop
    CMP     R0, #2          ; Check i == 2
    BEQ     outer_continue  ; Skip if i == 2

    MOVS    R1, #0          ; j = 0
inner_loop
    CMP     R1, #1          ; Check j == 1
    BEQ     outer_continue  ; Break to outer loop

    ADDS    R2, R0, R1      ; Calculate i + j
    ADDS    R4, R4, R2      ; Add to sum

    ADDS    R1, #1          ; j++
    CMP     R1, #3          ; Check j < 3
    BLT     inner_loop      ; Continue inner loop

outer_continue
    ADDS    R0, #1          ; i++
    CMP     R0, #5          ; Check i < 5
    BLT     outer_loop      ; Continue outer 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:

```
MulBy3  MOV     R4, R0       ; Save input value
        LSLS    R0, #1       ; Multiply by 2
        ADD     R0, R4       ; Add original value
        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}

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

                BL      write7seg

                POP     {R4,R5,PC}
```

Save LR and registers used by subroutine

Call another subroutine

Restore registers and PC

Stack Operations Implementation PUSH implementation:

```
    ; PUSH {R2,R3,R6}
    SUB     SP, SP, #12      ; Reserve stack space
    STR     R2, [SP]         ; Store R2
    STR     R3, [SP, #4]     ; Store R3
    STR     R6, [SP, #8]     ; Store R6
```

POP implementation:

```
    ; POP {R2,R3,R6}
    LDR     R2, [SP]         ; Restore R2
    LDR     R3, [SP, #4]     ; Restore R3
    LDR     R6, [SP, #8]     ; Restore R6
    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:

```
main
    BL      proc_a              ; Call proc_a
    ; continue main

proc_a
    PUSH    {LR}                ; Save return address
    BL      proc_b              ; Call proc_b
    POP     {PC}                ; Return to main

proc_b
    PUSH    {LR}                ; Save return address
    BL      proc_c              ; Call proc_c
    POP     {PC}                ; Return to proc_a

proc_c
    ; Do something
    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:

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

2. Function Epilogue:

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

3. Stack frame access:

```
    ; Access local variables
    STR     R0, [SP, #0]        ; First local
    STR     R1, [SP, #4]        ; Second local

    ; Access parameters
    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:

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

2. Function with locals:

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

3. Function with parameters:

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

Stack Frame Layout

Example of complete function:

```
; int calc(int a, int b, int c)
; a in R0, b in R1, c in R2
calc    PUSH    {R4-R6, LR} ; Save registers

        ; Save parameters
        MOVS    R4, R0      ; Save a
        MOVS    R5, R1      ; Save b
        MOVS    R6, R2      ; Save c

        ; Call helper function
        MOVS    R0, R4      ; First param
        BL      helper      ; Call helper

        ; Continue calculation
        ADDS    R0, R5      ; Add b
        ADDS    R0, R6      ; Add c

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

Stack usage considerations:

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

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:

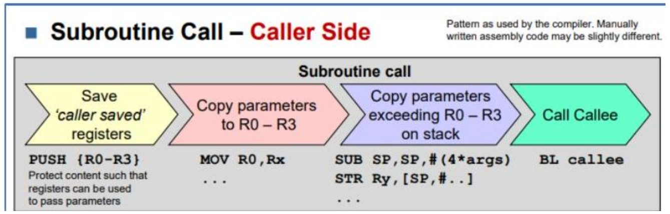
- First four arguments use R0-R3
- Additional parameters go on stack

Return Values:

- **Small Values** ($\leqslant$ 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



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):

```
    .data
value   DCD     0           ; Global variable

    .text
func    LDR     R0, =value  ; Load address
        LDR     R1, [R0]    ; Get value
        ; Process value
        STR     R1, [R0]    ; Store result
```

Register-based approach (preferred):

```
func    PUSH    {R4, LR}    ; Save registers
        ; R0 contains input parameter
        MOV     R4, R0      ; Save parameter
        ; Process value in R4
        MOV     R0, R4      ; Set return value
        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:

```
; Pass by value
func1   PUSH    {LR}
        ADDS    R0, #1          ; Modify parameter
        POP     {PC}            ; Original unchanged

; Pass by reference
func2   PUSH    {LR}
        LDR     R1, [R0]        ; Load from address
        ADDS    R1, #1          ; Modify value
        STR     R1, [R0]        ; Store back to address
        POP     {PC}            ; Original changed
```

Data Structure Access

Working with structures and arrays:

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

time_t time;
```

Assembly implementation:

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

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

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:

```
; uint32_t factorial(uint32_t n)
; Input in R0, result in R0
factorial
    PUSH    {R4, LR}        ; Save registers
    MOVS    R4, R0          ; Save n
    CMP     R4, #1          ; Check base case
    BLE     fact_end        ; Return 1 if n <= 1

    SUBS    R0, R4, #1      ; n-1
    BL      factorial       ; Recursive call
    MULS    R0, R4, R0      ; n * factorial(n-1)

fact_end
    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:

```
; void func(int a, int b, int c, int d, int e)
; First four params in R0-R3, fifth on stack
func    PUSH    {R4-R6, LR} ; Save registers

        ; Save parameters
        MOV     R4, R0      ; Save a
        MOV     R5, R1      ; Save b
        MOV     R6, R2      ; Save c
        ; R3 contains d
        LDR     R0, [SP, #16] ; Load e from stack

        ; Function body

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

Complex Parameter Example

Function with mixed parameter types:

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

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

Assembly implementation:

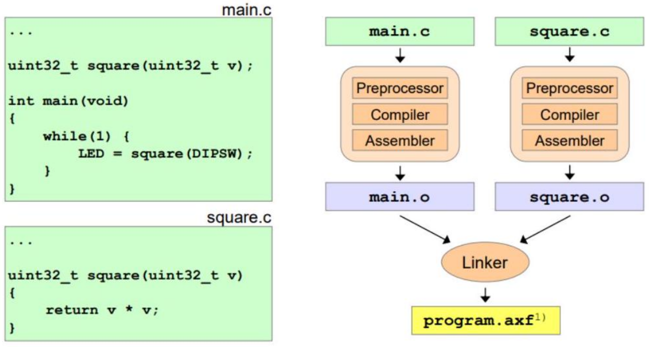
```
; R0 = point_t* p
; R1 = scale
; R2 = result pointer
calculate
    PUSH    {R4-R5, LR}     ; Save registers

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

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

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

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

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

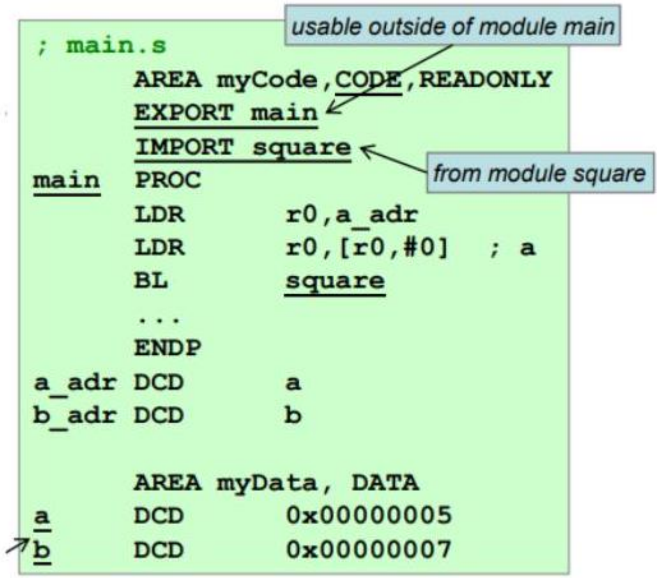
Key advantages:

- **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:

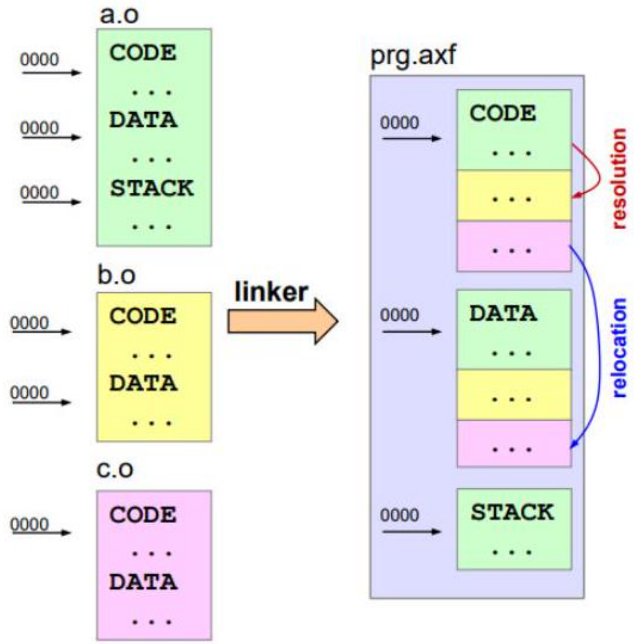
- **Code Section**:
  - Program code and constants
  - Based at address 0x0
- **Data Section**:
  - Global variables
  - Based at address 0x0
- **Symbol Table**:
  - All symbols and their attributes
  - Global/local status
  - References to external symbols
- **Relocation Table**:
  - Instructions for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

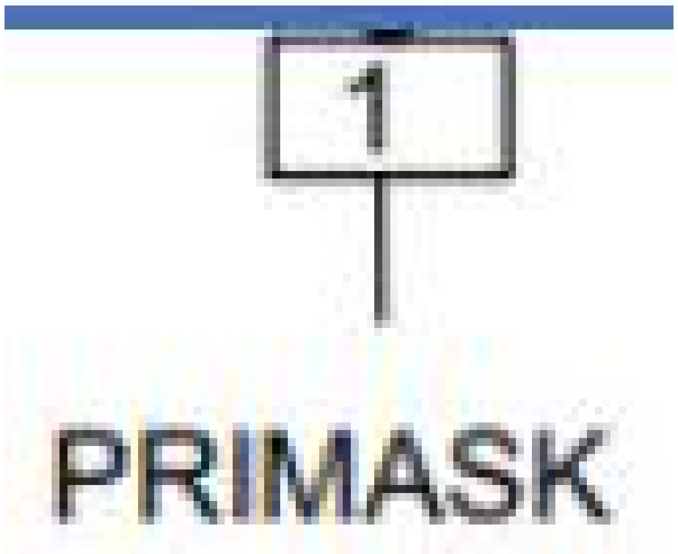
System Exceptions:

- **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

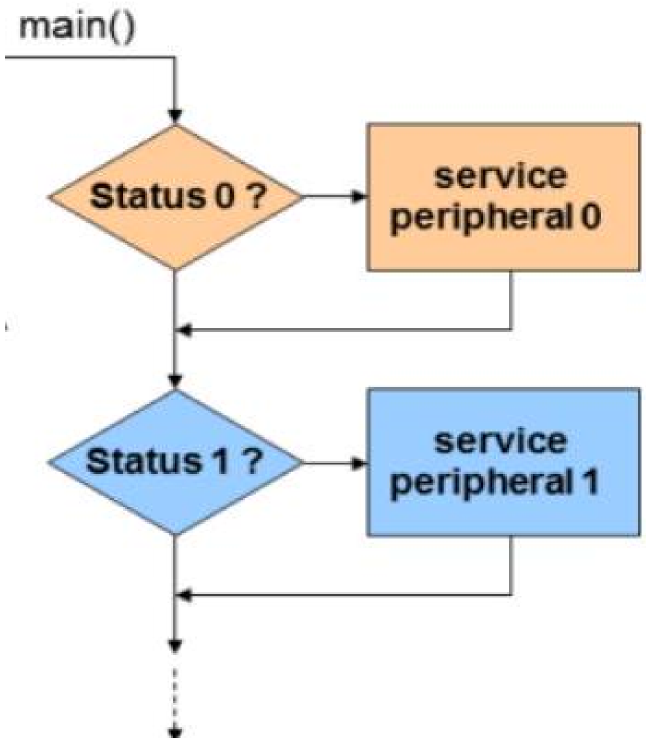
ISR Exit:

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

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

```
    ; 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:

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

2. Handler implementation:

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

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

Nested Interrupts Example

Implementation with different priorities:

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

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

    // Enable global interrupts
    __enable_irq();
}

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

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

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:

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

CMSIS Functions for Interrupt Control

Standard CMSIS functions for interrupt handling:

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

Example usage:

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

    // Set priority
    NVIC_SetPriority(TIM2_IRQn, 2);

    // Configure timer
    // ...

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

Increasing System Performance

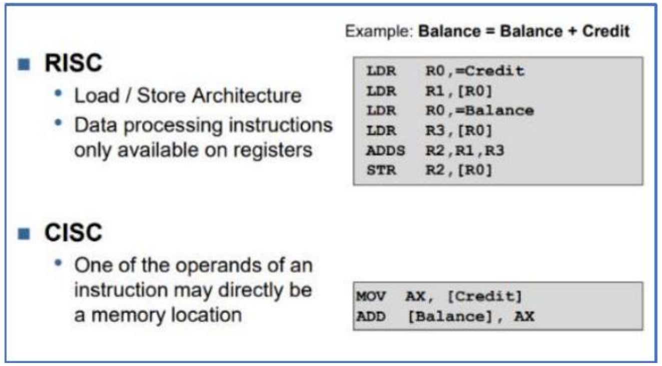
Performance Optimization Trade-offs

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



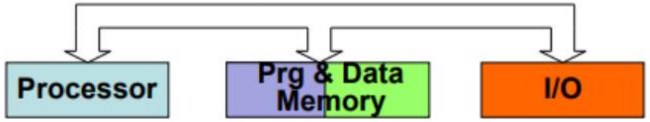
CISC (Complex Instruction Set Computer):

- More complex instruction set
- Lower memory usage for programs
- Potential performance gain for short programs
- More complex hardware required

Computer Architectures

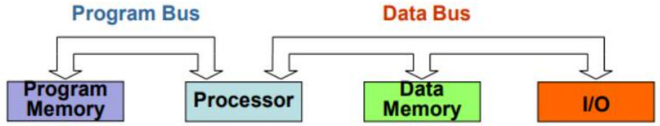
Von Neumann Architecture:

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



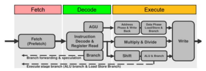
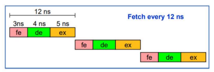
Harvard Architecture:

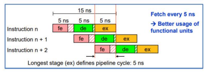
- Separate program and data memories
- Two sets of address/data buses
- Originally from Harvard Mark I



Pipelining

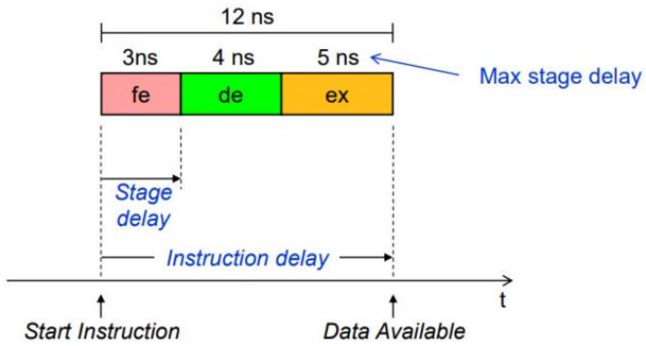
Process of fetching next instruction while current one decodes:



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:

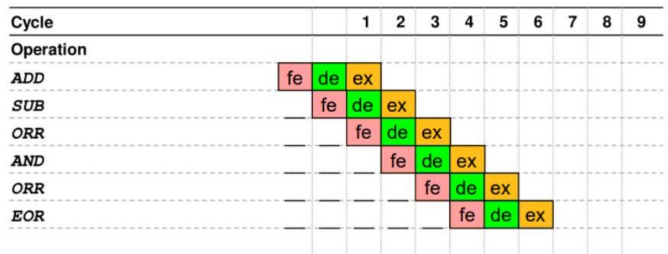
$$\frac{\text{Instructions}}{\text{second}} = \frac{1}{\text{Max stage delay}}$$

Note: Pipeline must be filled first

Pipeline Execution

Optimal Case:

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



LDR Special Case:

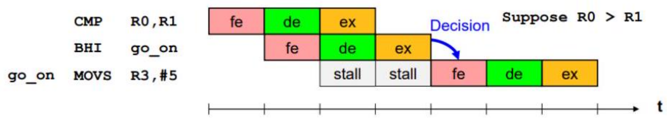
- 6 instructions in 7 cycles due to memory access
- Pipeline stalls for memory read
- CPI = 1.2

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**:

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

2. **Data Hazards**:

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

3. **Control Hazards**:

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

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

Examples