Computertechnik

Jil Zerndt, Lucien Perret
May 2024

Computer Engineering

What is Computer Engineering?

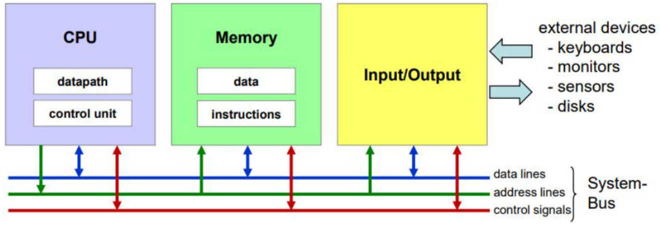
Computer Engineering is where microelectronics and software meet. It involves:

- 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:

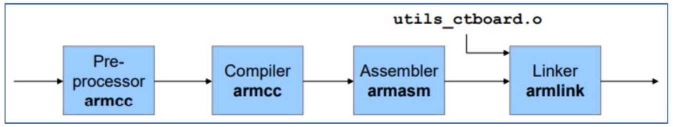
- **Core Registers**: Fast but limited storage inside CPU
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations
- **Control Unit**: Reads and executes instructions
- **Bus Interface**: Connects CPU to system bus

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

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

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

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

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:

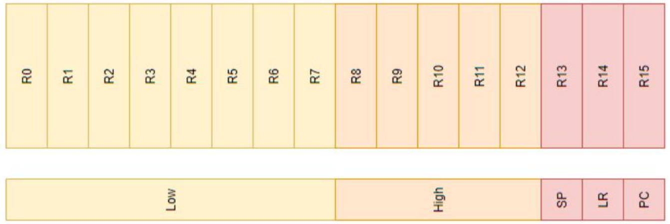
- Arithmetic operations (add, subtract, multiply)
- Logic operations (AND, OR, XOR)
- Compare operations
- Shift and rotate operations

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

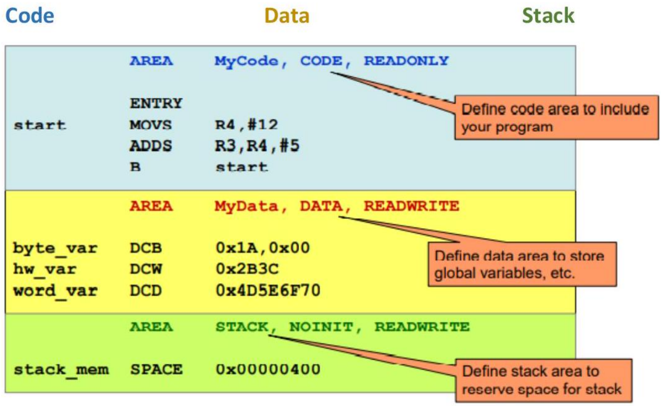
Basic Assembly Program Structure

Example of a simple assembly program:

```
Label    Instr.   Operands     Comments
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 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:

```
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 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

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

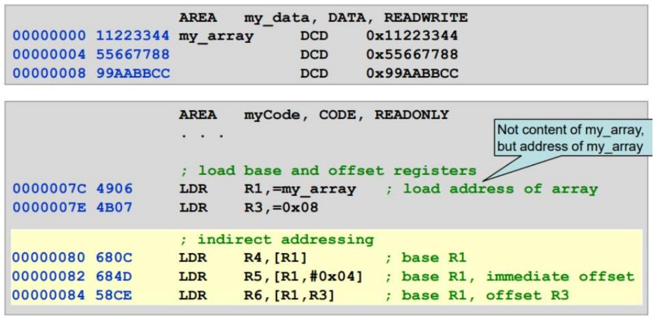
Store Instructions

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

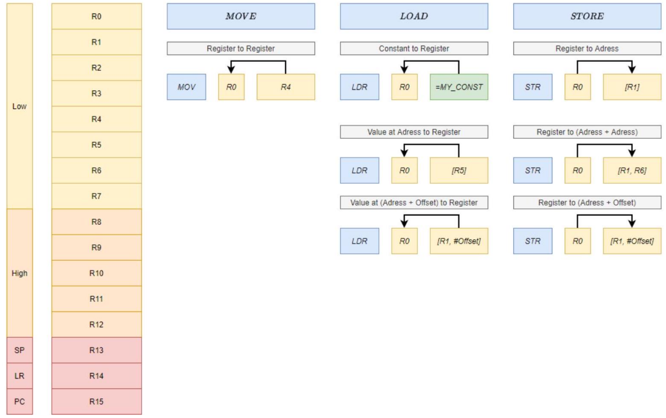
Memory Access Example

Loading and storing array elements:



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:

```
;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
```

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:

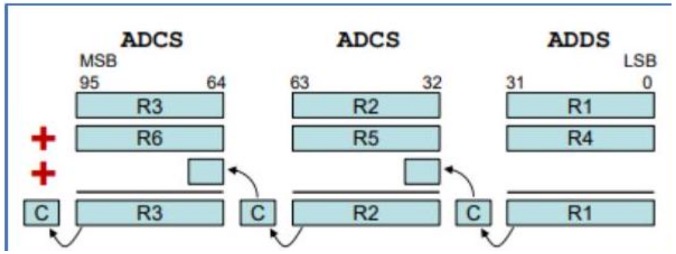
- Addition: C = 1 indicates carry (result too large)
- Subtraction: C = 0 indicates borrow (result negative)

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

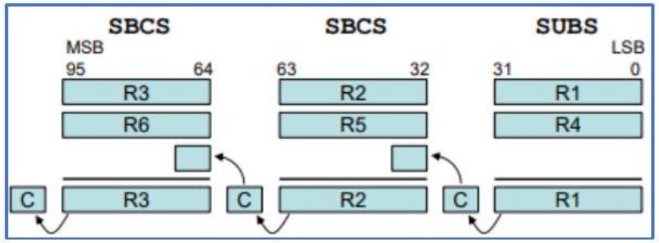
Adding 96-bit values using ADCS:




```
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
```

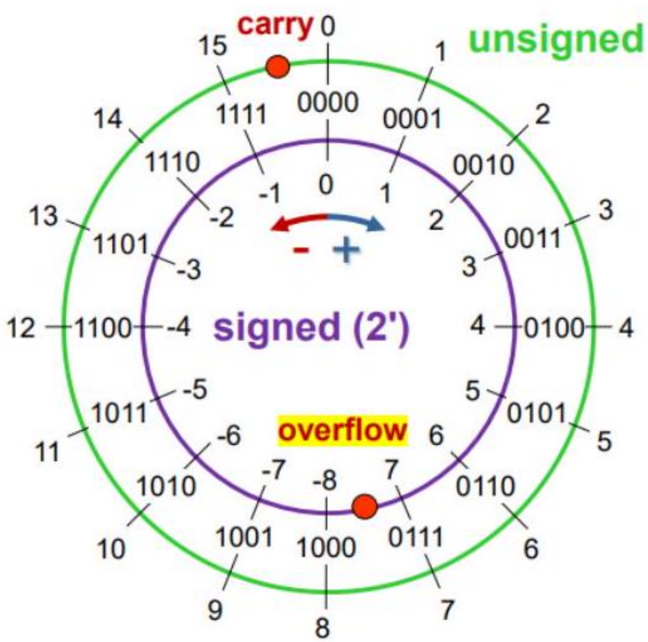
Addition and Subtraction Examples

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

Logic, Shift and Rotate Instructions

Logic Instructions

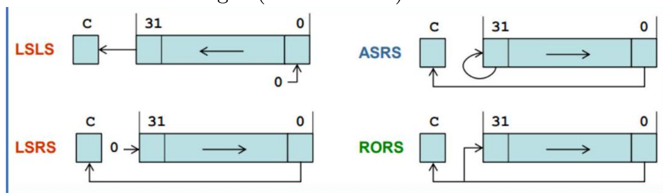
Base logic operations (affect only N and Z flags):

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

Shift and Rotate Instructions

Shift operations for binary manipulation:

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



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:

```
    ; 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

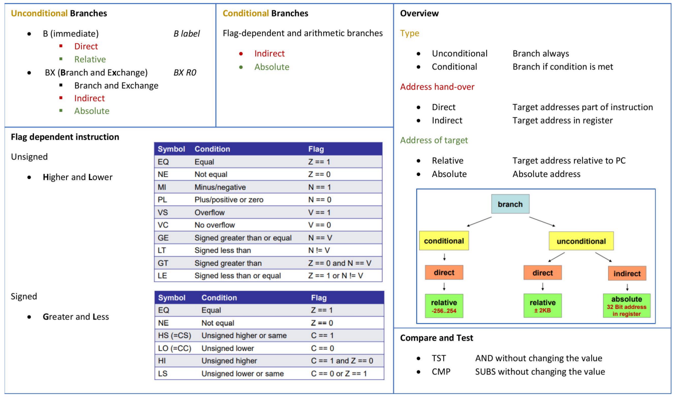
Steps for bit manipulation:

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

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

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

Overview

Type

- Unconditional — Branch always
- Conditional — Branch if condition is met

Address hand-over

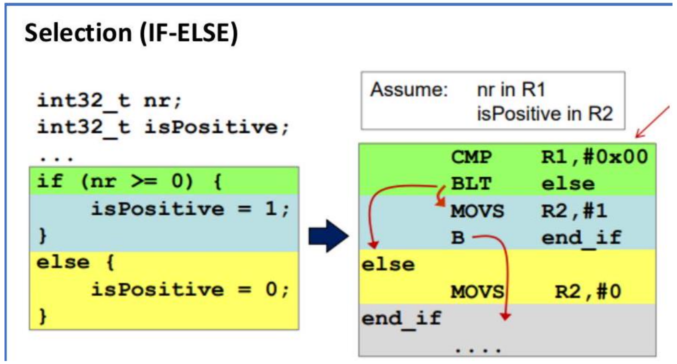
- Direct — Target addresses part of instruction
- Indirect — Target address in register

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



Switch Statement Implementation

C code example:

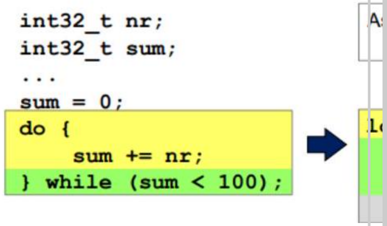
```c
uint32_t result, n;
switch (n) {
    case 0:
        result += 17;
        break;
    case 1:
        result += 13;
        //fall through
    case 3:
    case 5:
        result += 37;
        break;
    default:
        result = 0;
}
```

Assembly implementation with jump table:

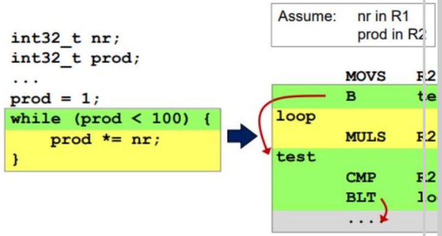
```
NR_CASES     EQU    6
case_switch  CMP    R1, #NR_CASES
             BHS    case_default
             LSLS   R1, #2          ; * 4
             LDR    R7, =jump_table
             LDR    R7, [R7, R1]
             BX     R7

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

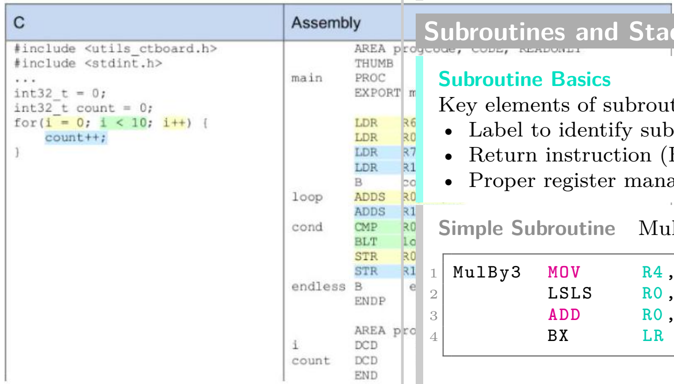
Three main types of loops:



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
```

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:

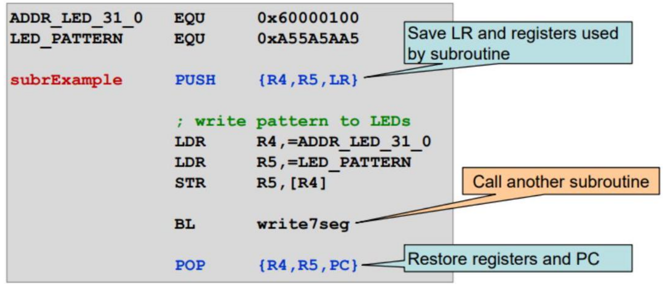
- **Stack Area**: Continuous RAM section
- **Stack Pointer (SP)**: R13, points to last written value
- **Direction**: Full-descending (grows toward lower addresses)
- **Alignment**: Word-aligned (4 bytes)
- **Data Size**: 32-bit words only

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



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

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:

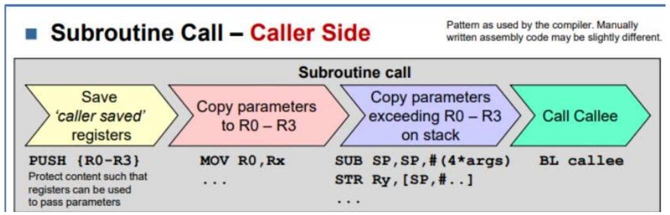
- First four arguments use R0-R3
- Additional parameters go on stack

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:

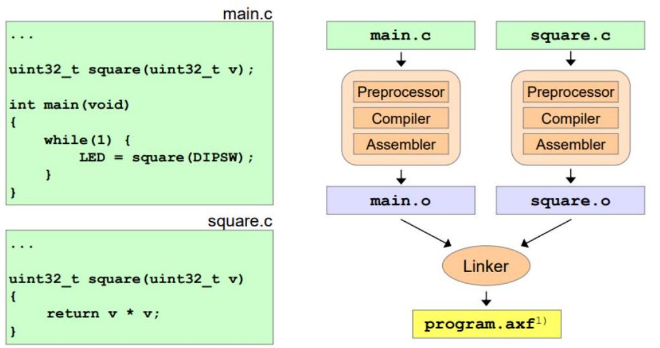
- Avoid global variables for parameter passing
- Use registers for efficiency
- Follow ARM calling convention strictly
- Consider stack usage in recursive functions

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

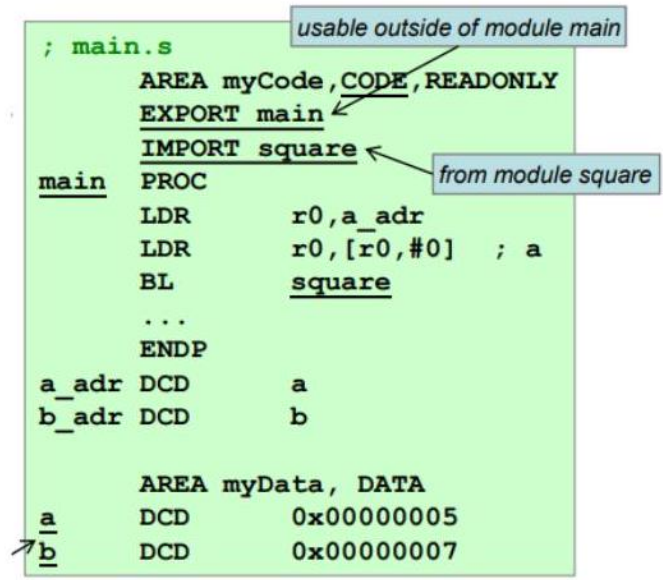
Key advantages:

- **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:

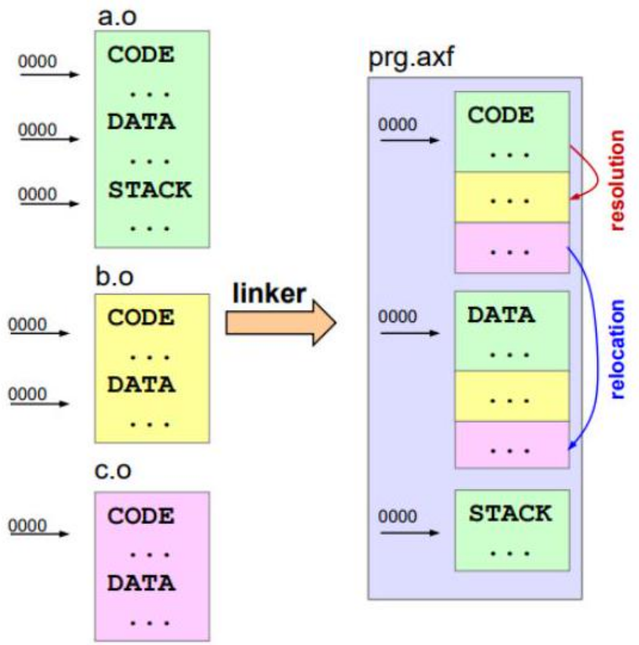
- **Code Section**:
  - Program code and constants
  - Based at address 0x0
- **Data Section**:
  - Global variables
  - Based at address 0x0
- **Symbol Table**:
  - All symbols and their attributes
  - Global/local status
  - References to external symbols
- **Relocation Table**:
  - Instructions for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

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

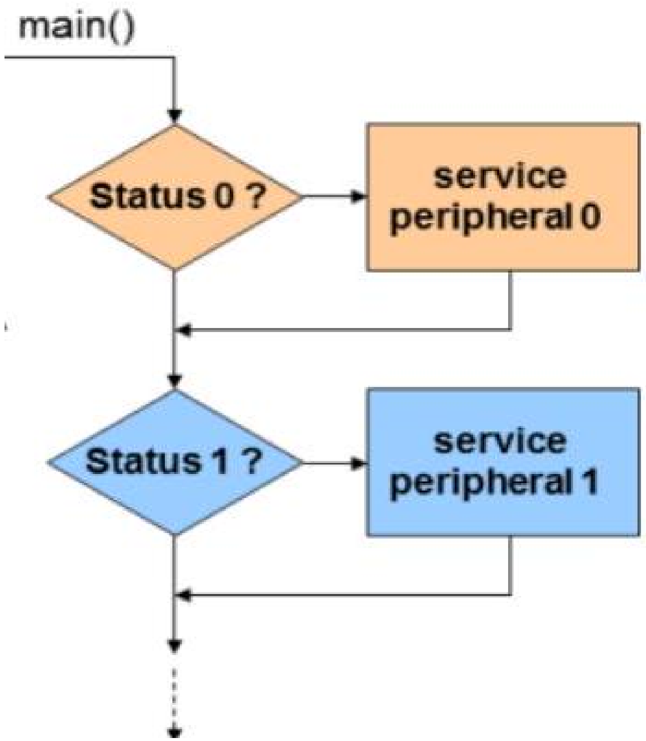
ISR Exit:

- 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

Increasing System Performance

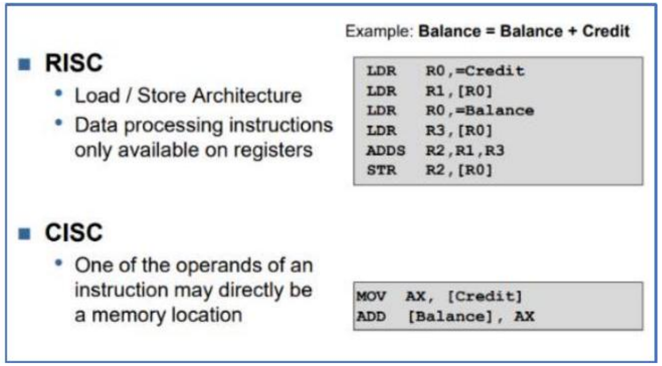
Performance Optimization Trade-offs

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



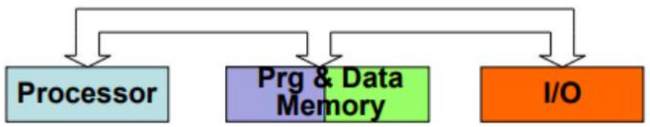
CISC (Complex Instruction Set Computer):

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

Computer Architectures

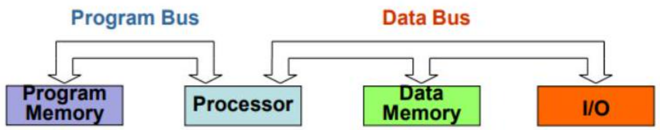
Von Neumann Architecture:

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



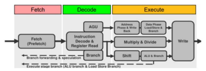
Harvard Architecture:

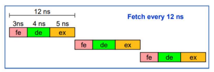
- Separate program and data memories
- Two sets of address/data buses
- Originally from Harvard Mark I

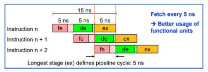


Pipelining

Process of fetching next instruction while current one decodes:

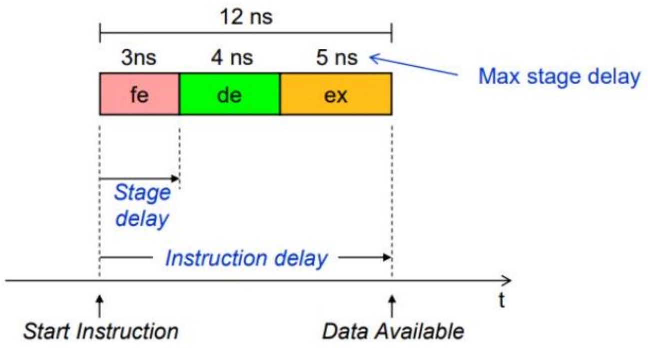






Pipeline Stages (Example):

- 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)

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

LDR Special Case:

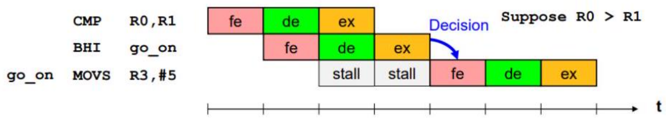
- 6 instructions in 7 cycles due to memory access
- Pipeline stalls for memory read
- CPI = 1.2

Pipeline Hazards and Optimization

Control Hazards:

- Branch decisions in execute stage
- Pipeline stalls for taken branches

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



Optimization Techniques:

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

Optimization Limits:

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

Parallel Computing

Different approaches to parallelism:

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

Optimizing System Performance

Steps for performance optimization:

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

Examples