# Computertechnik

Jil Zerndt, Lucien Perret
May 2024

## Computer Engineering

### What is Computer Engineering?

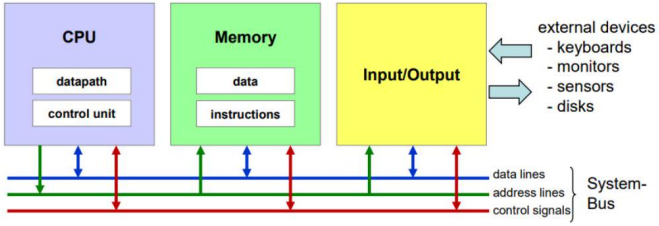
Computer Engineering is where microelectronics and software meet. It involves:

- 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:

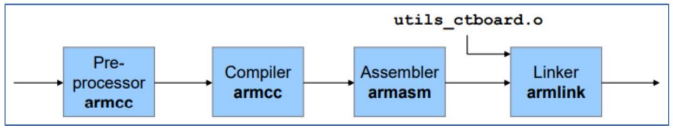
- **Core Registers**: Fast but limited storage inside CPU
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logic operations
- **Control Unit**: Reads and executes instructions
- **Bus Interface**: Connects CPU to system bus

### 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:

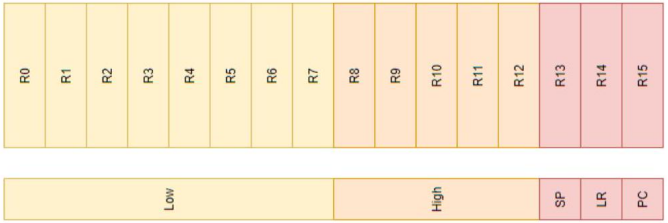
- Arithmetic operations (add, subtract, multiply)
- Logic operations (AND, OR, XOR)
- Compare operations
- Shift and rotate operations

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

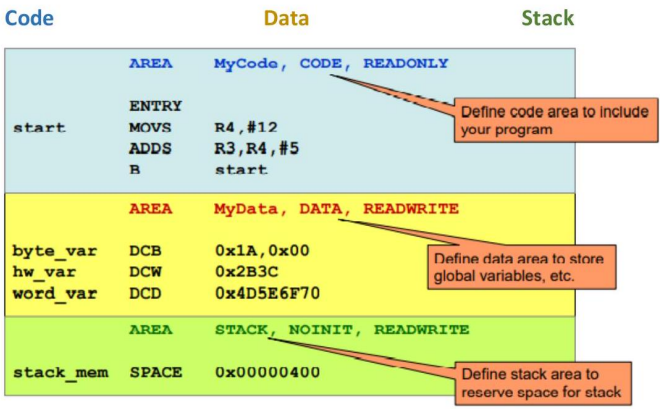
### Basic Assembly Program Structure

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:

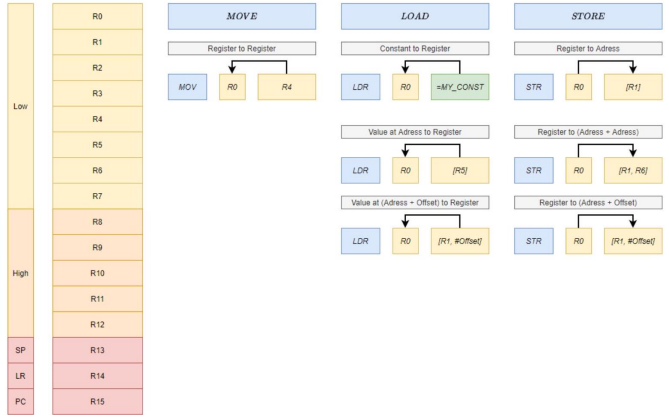
```
                    AREA    my_data, DATA, READWRITE
00000000 11223344 my_array      DCD     0x11223344
00000004 55667788               DCD     0x55667788
00000008 99AABBCC               DCD     0x99AABBCC
```

```
                    AREA    myCode, CODE, READONLY
                    . . .
                                        Not content of my_array, but address of my_array
                    ; load base and offset registers
0000007C 4906       LDR     R1,=my_array    ; load address of array
0000007E 4B07       LDR     R3,=0x08

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

### 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

## Arithmetic Operations

Flags (APSR = N, Z, C, V)

Instructions ending with with «S» allow flag modification

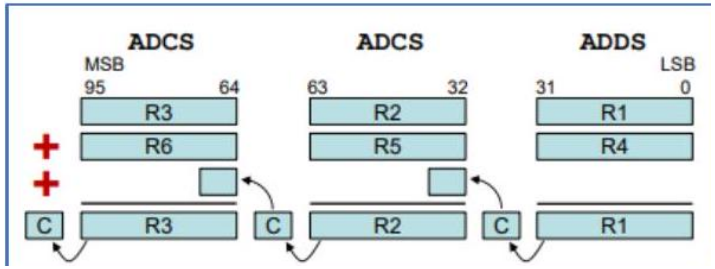
- ADDS
- SUBS
- MOVS
- LSLS

| Flag | Meaning | Action | Operands |
|---|---|---|---|
| Negative | MSB =1 | N = 1 | signed |
| Zero | Result = 0 | Z = 1 | signed, unsigned |
| Carry | Carry | C = 1 | unsigned |
| Overflow | Overflow | V = 1 | signed |

## Overview

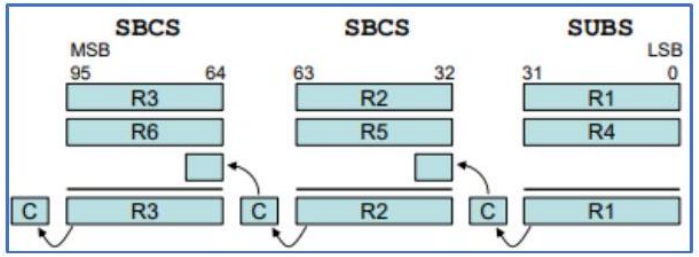
- ADD / ADDS
- ADCS Addition with Carry
- ADR Address to Register
- SUB / SUBS
- SBCS
- RSBS
- MULS

Subtraction
Subtraction with carry (borrow)
Reverse Subtract (negative)
Multiplication
$A + B$
$A + B + c$
$PC + A$
$A - B$
$A - B - !c$
$-1 \cdot A$
$A \cdot B$
Multi-Word Addition with ADCS




| | | | |
|---|---|---|---|
| ADDS | R1, | R1, | R4 |
| ADCS | R2, | R2, | R5 |
| ADCS | R3, | R3, | R6 |

Multi-Word Subtraction with SBCS




| | | | |
|---|---|---|---|
| SUBS | R1, | R1, | R4 |
| SBCS | R2, | R2, | R5 |
| SBCS | R3, | R3, | R6 |

## Negative Number

- 2' Complement $A = !A + 1$

## Carry and Overflow

unsigned

- Addition → C = 1 → carry result too large for available bits
- Subtraction → C = 0 → borrow result less than zero → no negative numbers

signed

- Addition → potential overflow in case of operands with equal signs
- Subtraction → potential overflow in case of operands with opposite signs

## Addition and Subtraction

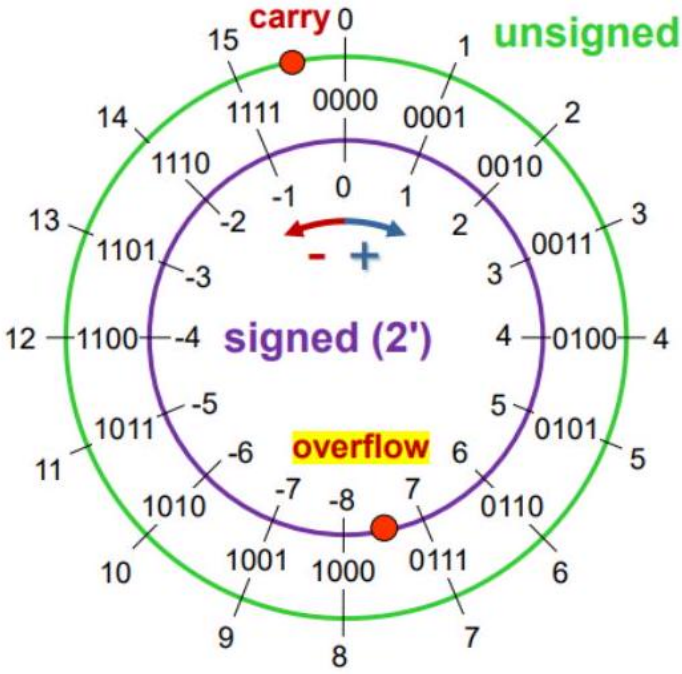
- Addition C = 1 → Carry

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 0 | 1 | $13d$ |
| 0 | 1 | 1 | 1 | $7d$ |
| 1 | 1 | 1 | 1 | |
| 1 | | | | |
| 1 | 0 | 1 | 0 | 0 |

- Subtraction C = 0 → Borrow sign
  $6d - 14d = 0110b - 1110b = 0110b + 0010b$

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | $6d$ |
| 0 | 0 | 1 | 0 | $2d = \text{TC}(14d)$ |

0100




## Logic, Shift and Rotate Instructions

## Logical Instructions

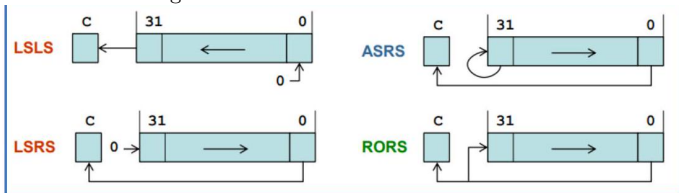
The following instruction only affect N and Z flags

- ANDS
- BICS
- EORS
- MVNS Bitwise NOT
- ORRS

Bitwise OR
Rdn # Rm
Rdn & Rm
Rdn & ! Rm $a \& \sim b$
Rdn $ Rm $a \hat{} b$
Rm a
a | b

Shift Instructions

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




## Sign-Extension

Add additional bits

- Unsigned zero extension fill left bits with zero
- Signed sign extension copy sign bit to the left

| Unsigned →∼ Zero Extension | | | | |
|---|---|---|---|---|
| 1011 → | 00001011 | 0011 | → | 00000011 |
| Signed → Sign Extension | | | | |
| 1011 → | 11111011 | 0011 | → | 00000011 |

## Truncation

Cast cuts out the left most digits

- Signed possible change of sign
- Unsigned results in module operation

## Integer ranges based on word sizes

| 8-bit | hex $0 \times 00$ | unsigned | signed | 16-bit | hex 0×0000 | unsigned |
|---|---|---|---|---|---|---|
| | | | | | | |
| | 0x75 | 127 | 127 | | 0x7FFF | F 32'767 |
| | 0x80 | 128 | -128 | | 0x8000 | 032 · 768 |
| | | . . . | -. | | . . | . . |
| | 0xFF | 255 | -1 | | 0xFFFF | F 65'535 |

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

# Control Structures

## Branch Instructions

**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

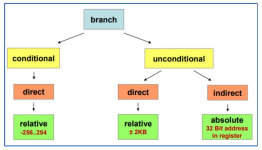
- Unconditional — Branch always
- Conditional — Branch if condition is met

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

## Selection (IF-ELSE)

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

Assume: nr in R1, isPositive in R2

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

## Switch

### - Jump Table

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

| | | |
|---|---|---|
| NR_CASES case_switch | EQU | 6 |
| | 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 |
| 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 |

## Loops

- Do while: Post-Test Loop

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

Assume: nr in R1, sum in R2

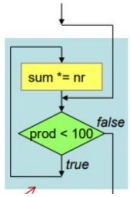
```
    MOVS  R2,#0
loop
    ADDS  R2,R2,R1
    CMP   R2,#100
    BLT   loop
    ....
```

- While = Pre-Test Loop

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

Assume: nr in R1, prod in R2

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




- For = Pre-Test Loop

**C**

```
#include <utils_ctboard.h>
#include <stdint.h>
...
int32_t i = 0;
int32_t count = 0;
for(i = 0; i < 10; i++) {
    count++;
}
```

**Assembly**

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

        LDR   R6,=i     ; R6=address of i
        LDR   R0,[R6]   ; R0=value at i
        LDR   R7,=count ; R7=address of count
        LDR   R1,[R7]   ; R1=value at count
        B     cond
loop    ADDS  R0,R0,#1
        ADDS  R1,R1,#1
cond    CMP   R0, #10
        BLT   loop      ; *signed* comparison
        STR   R0,[R6]   ; store final i
        STR   R1,[R7]   ; store final count
endless B     endless
        ENDP

        AREA progData, DATA, READWRITE
i       DCD   0
count   DCD   0
        END
```

# Subroutines and the Stack

## Subroutine Call and Return

- Label with Name (MulBy3)
- Return Statement (BX 1$R$ )

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

Stack

- Stack Area (Section)
- Stack Pointer (SP)
- PUSH {...}
- POP {...}
- Direction on ARM
- Alignment
- Only words

Continuous area of RAM
R13 → points to last written data value
Decrement SP and store words
Read words and increment SP
full-descending stack
word-aligned
32-Bit

## Stack - Push and Pop

- Number of Pushs = Number of Pops
- Stack-limit < SP < 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

PUSH {R2,R3,R6}

| | | | |
|---|---|---|---|
| 00000000 | B083 | SUB | SP,SP,#12 |
| 00000002 9200 | STR | R2,[SP] | |
| 00000004 | 9301 | STR | R3,[SP,#4] |
| 000000069602 | STR | R6,[SP,#8] | |

## POP {R2,R3,R6}

| | | | |
|---|---|---|---|
| 00000008 9A00 | LDR | R2,[SP] |
| 0000000A 9B01 | LDR | R3,[SP,#4] |
| 0000000C 9B02 | LDR | R6,[SP,#8] |
| 0000000E B003 | ADD | SP,SP,#12 |

## Parameter Passing

### Where

- Register
- Global variables
- Stack
- Caller: PUSH parameter on stack
- Callee: Access parameter through LDR

### Reentrancy

- Recursive Function Calls
- Registers and gobal variables are overwritten
- Requires an own set of data for each call
- Solution:
- ARM Procedure Call Standard

### Passing through global variables

- Shared variables in data area
- Overhead to access variable
- Error prone, unmaintainable
- By reference
- Allows passing of larger structures

## ARM Procedure call Standard

### Parameters

- Caller copies arguments From R0 to R3
- Caller copies additional parameters to stack

Returning fundamental data types

- Smaller than word
- Word
- Double word
- 128-Bit
  zero or sign extend to word return in RO return in RO / R1 return in RO - R3

Returning composite data types

- Up to 4 bytes
  return in RO
- Larger than 4 bytes stored in data area

## Register / "pass by value"

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

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

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

Register Usage

| Register | Synonym | Role |
|---|---|---|
| ro | a1 | Argument / result / scratch regist |
| r1 | a2 | Argument / result / scratch regist |
| I2 | a3 | Argument / scratch register 3 |
| r3 | a4 | Argument / scratch register 4 |
| 4τ | v1 | Variable register 1 |
| r5 | v2 | Variable register 2 |
| r6 | v3 | Variable register 3 |
| r7 | v4 | Variable register 4 |
| 8τ | v5 | Variable register 5 |
| r9 | v6 | Variable register 6 |
| r10 | v7 | Variable register 7 |
| r11 | v8 | Variable register 8 |
| r12 | IP | Intra-Procedure-call scratch register |
| r13 | SP | |
| r14 | LR | |

Register contents
might be modified
might be modified
Callee must preserve contents
of these registers (Callee saved)

## Subroutine Call – Caller Side

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

Subroutine call

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

On return from subroutine

## Modular Coding and Linking

## Modular Coding / Linking

## From source code to executable program

Compile / assemble each module

- Results in an object file for each module

Link all object files

- Results in one executable file

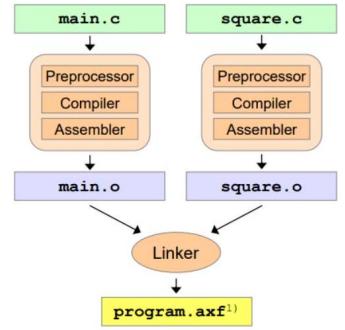
main.c

```
...
uint32_t square(uint32_t v);

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

square.c

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




## Managing complexity by modular programming

| Topic | Benefits |
|---|---|
| Enable working in teams | Multiple devel repository |
| Useful partitioning and structuring of the programs | Eases reuseing c |
| Individual verification of each module | Benefits all user |
| Providing libraries of types and functions | For reuse instea |
| Mixing of modules that are programmed in various languages | E.g. mix C and |
| Only compile the changed modules | Speeds up comp |

## ARM assembly IMPORT and EXPORT keywords

Linkage control

- EXPORT for use by other module
- IMPORT from another module for use in this module Internal symbols
- Neither IMPORT nor EXPORT

```
; main.s
        AREA myCode,CODE,READONLY
        EXPORT main
        IMPORT square
main    PROC
        LDR     r0,a_adr
        LDR     r0,[r0,#0]   ; a
        BL      square
        ...
        ENDP
a_adr   DCD     a
b_adr   DCD     b

        AREA myData, DATA
a       DCD     0x00000005
b       DCD     0x00000007
```

*usable outside of module main*

*from module square*

## Linker Input - Object files

Code section Code and constant data of the module, base at address $0 \times 0$
Data section All global variables of the module, based at address $0 \times 0$
Symbol table All symbols with their attributes like global/local, reference
Relocation table

- Which bytes oft he data and code section need to be adjusted (and how) after merging the sections in the linking process

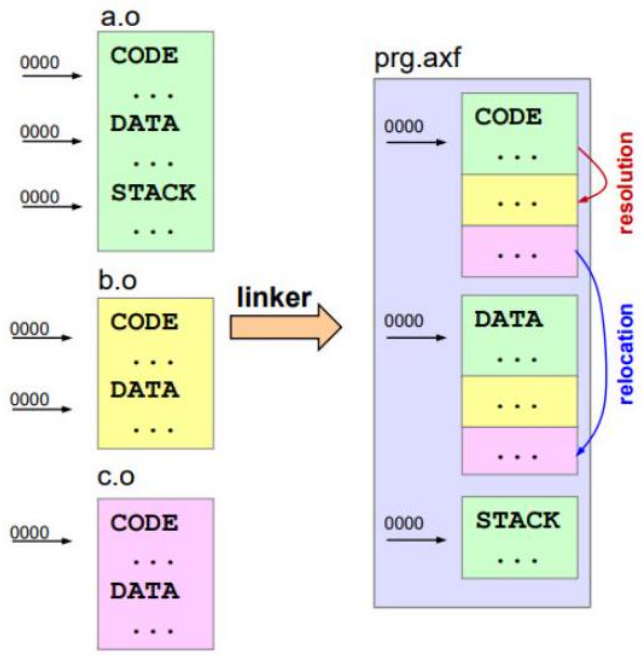
ARM tool chain uses ELF for object files

## Linker tasks

- Merge object file code sections
- Merge object file data sections
- Symbol resolution
- Address relocation

## Linker Output

- AXF = **ARM** eXecutable File




## Exceptional Control Flow

## Interrupt sources

- Perfipherals signal to CPU that an event needs immediate attention
- Can alternativly be generated by software request
- Asynchronous to instruction execution

## System exceptions

- Reset
- NMI
- Faults
- System Level Calls

Restart of processor
Non-maskable Interrupt (cannot be ignored) Undefined instructions OS calls - Instructions SVC and PendSV

PRIMASK
- Single bit controlling all maskable interrupts

| | | |
|---|---|---|
| - Disable<br>- Enable | set PRIMASK clear PRIMASK | Assembly<br>CPSID [1] CPSIE [1] |

On reset PRIMASK = 0 → enabled

## Storing the context

Interrupt event can take place at any time

- E.g. between TST and BEQ instructions
- ISR call requires automatic save off lags and caller saved registers

ISR call

- Stores xPSR, PC, LR, R12, R0-R3 on Stack
- Stores EXC_RETURN to LR

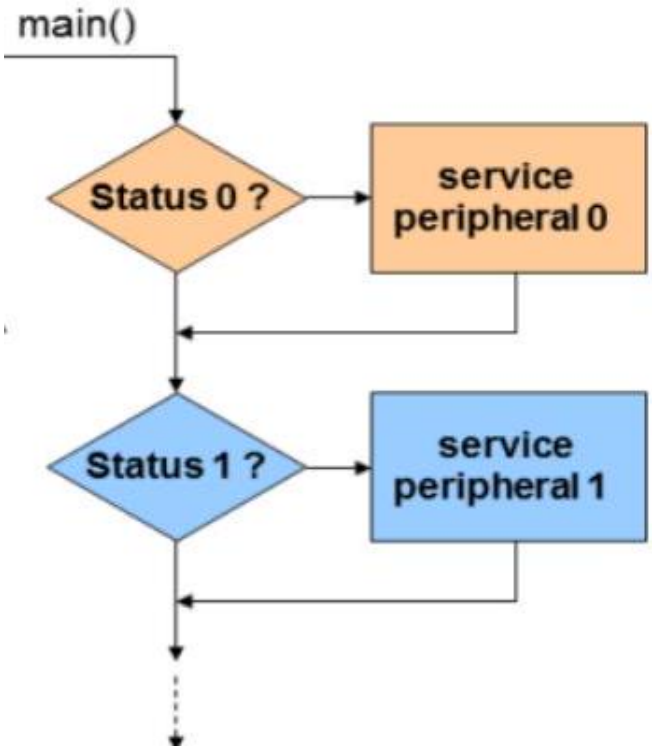
ISR Return

- Use BX LR or POP {..., PC}
- Loading EXC Return into PC
- Restores RO-R3, R12, LR, PC and xPSR from Stack

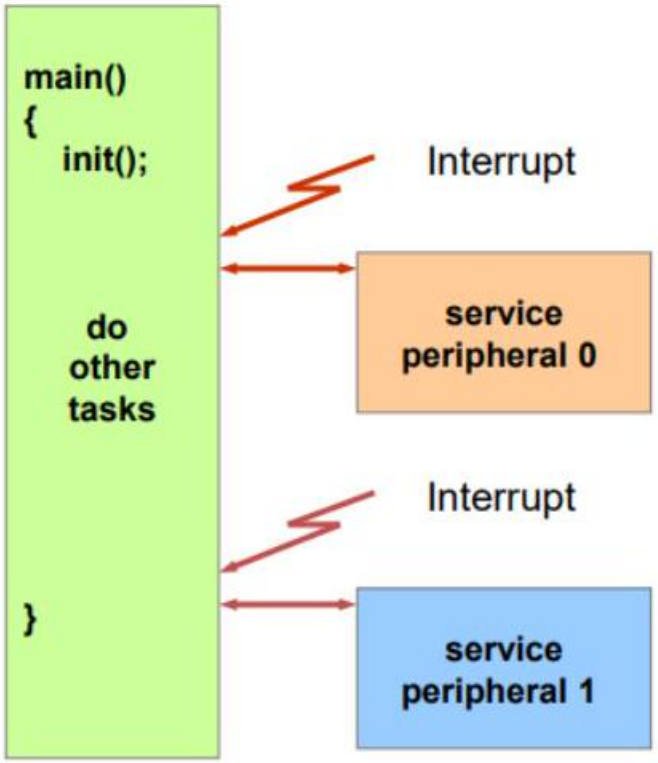
## Polling

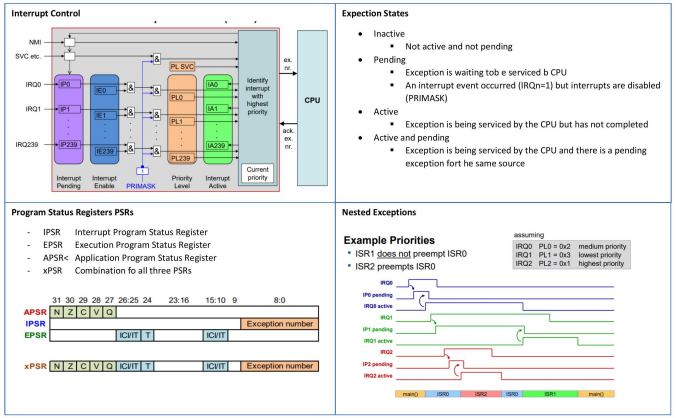
Periodic query of status information

- Reading of status registers in loop
- Synchronous with main program
- Advantages
- Simple straightforward
- Implicit synchronisation
- Deterministic




- No additional interrupt logic required
- Disadvantages
- Busy wait -> wastes CPU time
- Reduced throughput
- Long reaction time
- No synchronization
- Difficult debugging




## Increasing System Performance

| Speed vs Low Power Aspects of Optimization | |
|---|---|
| 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 |

## RISC = Reduced Instruction Set Computer

- Few instructions, unique instruction format
- Fast decoding, simple addressing
- Less hardware -> allows higher clock rates
- More chip space for registers (up to 256!)
- Load-store architecture reduces memory access,

CPU works at full-speed on registers

- Higher clock frequencies
- Easy and shorter pipelines (instructio size / duration)

Example: **Balance = Balance + Credit**

**RISC**

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

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

**CISC**

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

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

CISC = Complex Instruction Set Computer

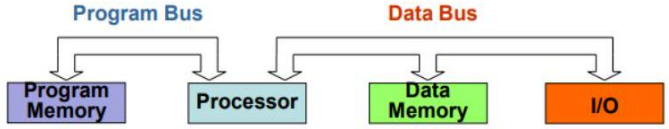
- More complex and more instructions
- Less program memory needed with complex instructions
- Short programs may work faster with less memory accesses

## Von Neuman Arhcitecture

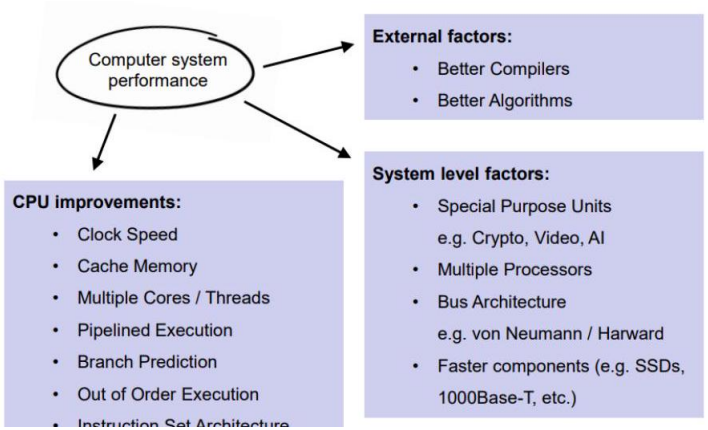
- Same memory holds program and data
- Single bus system between CPU and memory Systembus


## Harvard Architecture

- «Mark I» at Harvard University
- Separate memories for program and data
- Two sets of addresses/data buses between CPU and memory

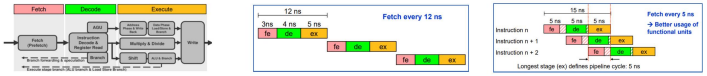



How to Increase System Speed?



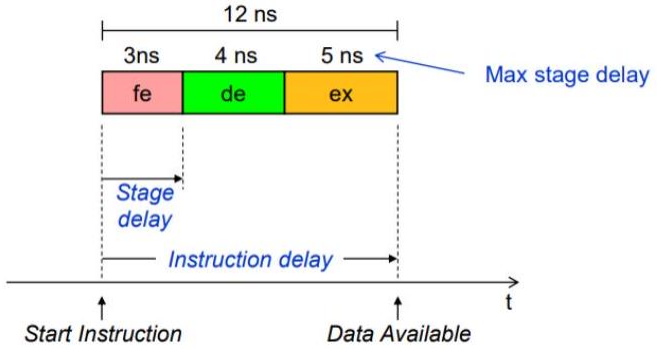

Fetching the next instruction, while the current one decodes

Sequential vs. Pipelined



## Timings and definitions (Example)

- Fe: fetch Read instructions 3 ns
- De: decode Decode instruction, read register or memory 4 ns
- Ex: execute Execute instruction, write back result 5 ns




## Advantages of pipelining

- All stages are set tot he same execution time
- Massive performance gain
- Simpler hardware at each stage allows for a higher clock rate

## Disadvantages

- A blocking stage blocks while pipeline
- Multiple stages may need to have access to the memory at the same time

## Instructions per second

Without pipelining

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

With pipelining

- Pipeline needs to be filled first
- After filling, instructions are executed after every stage

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

## Optimal pipelining

- All operations here are on registers
- In this example it takes 6 clock cycles to execute 6 instructions
- Clock cycles per instruction (CPI) = 1

| 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 | |

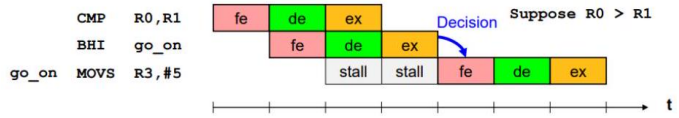
## Special situation: LDR

- In this example it takes 7 clock cycles to execute 6 instructions
- Read cycle must complete on the bus before LDR instruction can complete
- Next 2 instructions must wait one pipeline cycle ( S = stall)
- Clock cycles per Instruction (CPI) = 1.2

## Control Hazards

- Branch / jump decisions occur in stage 3 (ex)
- Worst case scenario - conditional branch taken:

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




## Reduce control hazards

- Loop fusion reduces control hazards

## Ideas to further improve pipelining

- Branch prediction
- Store last decisions made for each conditional branch
- -> probability is high that the same decision is taken again
- Instruction prefetch
- Fetch several instructions in advance
- -> better use of system bus
- -> possibility of «Out of Order Execution»
- Out of Order Execution
- If one instruction stalls, it might be possible to already execute the next instruction

## Limits of optimization

- Complex optimizations -> sever security problems
- Instructions executed, that would throw access violations under «In Order» circumstances.
- «Meltdown» and «Spectre» attacks: allow a process to access the data of another process

## Parallel Computing

- Streaming / Vector processing One instruction processes multiple data items simultaneously
- Multithreading Multiple programs/threads share a single CPU
- Multicore Processors One processor contains multiple CPU cores
- Multiprocessor Systems A computer system contains multiple processors