CT1 Summary (beliebig lang)

21. September, 2024; rev. 17. Januar 2025

Linda Riesen (rieselin)

1 Vorlesung 01

1965: The number of transistors per IC doubles every yearßomewhat slower since 1975 (doubles every 18 months, i.e. exponential growth)

1.1 Function of a Simple Computer System (explain and outline)

Computer System

- processes input
- takes decisions based on the outcome
- and outputs the processed information

Hardware and software work together → application

- often a common hardware is used for many different applications
- application is defined by the software (e.g. controls for washing machines, vending machines ...)

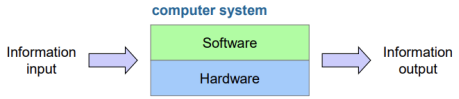


Abbildung 1: Computer System

Properties

- instructions and data are stored in the same memory
- datapath executes arithmetic and logic operations and holds intermediate results
- control unit reads and interprets instructions and controls their execution

1.2 Four Main Hardware Components and their functions

- **CPU** Central Processing Unit or processor
- **Memory** stores instructions and data
- **Input / Output** interface to external devices
- **System-Bus** electrical connection of blocks

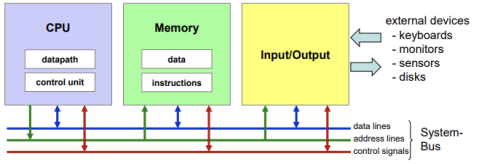


Abbildung 2: Hardware Componets

1.2.1 CPU

Datapath

- ALU: Arithmetic and Logic Unit : performs arithmetic/logic operations
- registers
  - fast but limited storage inside CPU
  - hold intermediate results
- 4 / 8 / 16 / 32 / 64 bits wide

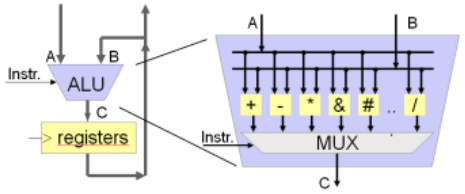


Abbildung 3: Data Path

Control Unit

- Finite State Machine (FSM) : reads and executes instructions
- types of operations
  - data transfer: registers memory
  - arithmetic and logic operations
  - jumps

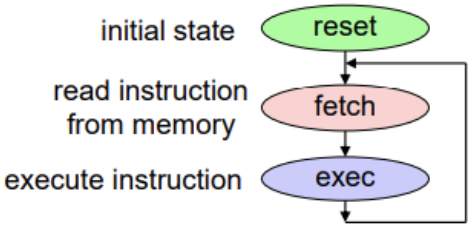


Abbildung 4: Finite State Machine

1.2.2 Memory

- a set of storage cells : 8 bit $\rightarrow$ 1 byte
- smallest addressable unit
  - one byte
  - **one address per byte**
- 2^N addresses
  - from 0 to $2^N - 1$
  - can be read and sometimes written
  - RAM Random Access Memory read/write
  - ROM Read Only Memory read

Main Memory (Arbeitsspeicher)

- central memory
- connected through System-Bus
- access to individual bytes
- volatile (flüchtig)
  - SRAM – Static RAM
  - DRAM – Dynamic RAM
- non-volatile (nicht-flüchtig)
  - ROM factory programmed
  - flash in system programmable

Secondary Storage

- long term or peripheral storage
- connected through I/O-Ports
- access to blocks of data
- non-volatile
- slower but lower cost
  - magnetic: hard disk, tape, floppy
  - semiconductor: solid state disk
  - optical: CD, DVD
  - mechanical: punched tape/card

1.2.3 System-Bus

- CPU writes or reads data from/to memory or I/O
- address lines
  - CPU drives the desired address onto the address lines
    - to which address does the CPU write?
    - from which address does the CPU read?
    - analogy → address on an envelope of a letter
  - number of addresses = 2^n → n = number of address lines
    - n = 16 → 2^{16} = 65'536 addresses → 64 KBytes
    - n = 20 → 2^{20} = 1'048'576 addresses → 1 MBytes

Control Signals

- CPU tells whether the access is **read** or **write**.
- CPU tells when address and data lines are valid ⇒ bus timing.

Data Lines

- Transfer of data:
  - Analogy: the letter that's inside the envelope.
  - **Write**: CPU provides data ⇒ memory receives data.
  - **Read**: CPU receives data ⇐ memory provides data.
- 4/8/16/32/64 data lines.

1.3 four translation steps from source code in C to exec. program

- Programmer writes main.c in text editor
- main.c is stored in ASCII / Unicode format on disk
- From C to executable: Translation of main.c into machine language

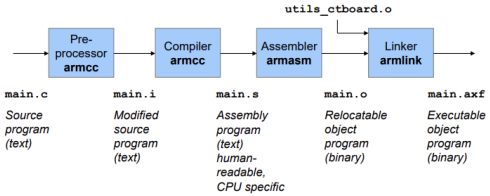


Abbildung 5: From C to executable

Preprocessor

- Text Processing
- Pasting of #include files
- Replacing macros (#define)

Compiler

- Translate CPU-independent C-code into CPU-specific assembly code

Assembler

- Translate to machine instructions
- Result: Relocatable object file
- Binary file → not readable with text editor, use Hex Dump

Linker

- Merge object files
- Resolve dependencies and cross-references
- Create executable

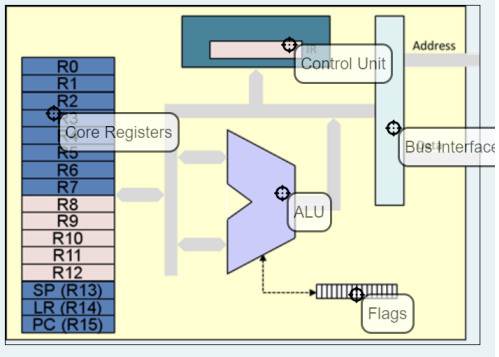


Abbildung 6: CPU Components

2 Vorlesung 02

2.1 CPU Model

2.1.1 Instruction Set Architecture (ISA)

What the programmer sees of a computer
Depends on not just available Instructions:

- Instruction Set (Available instructions)
- Processing width (8-bit/16-bit/32-bit?)
- Register Set (How many registers of what size)
- Addressing modes (How can memory and IO be accessed)
- ARM Cortex

2.1.2 CPU Components

- 16 Core Registers (each 32 Bit)

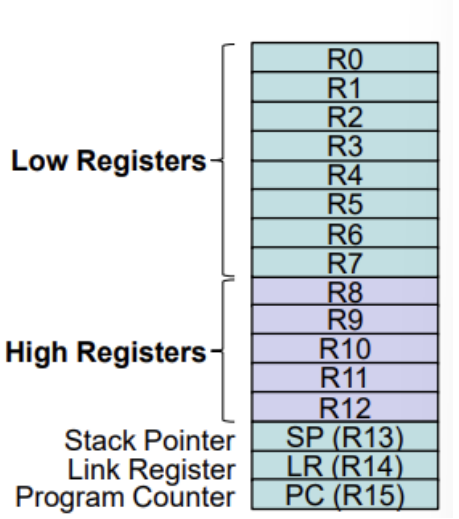


Abbildung 7: Core Registers

- 32 bit ALU (Rechenwerk)

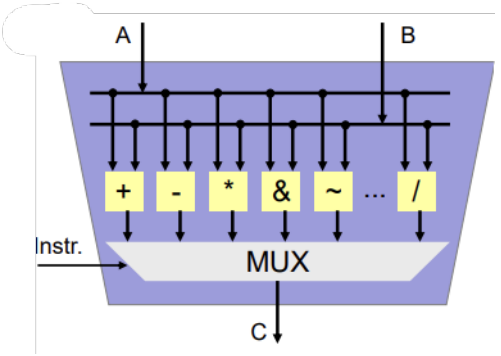


Abbildung 8: ALU

- Flags (APSR)
  - **Borrow (Unterlauf bei Subtraktion):**
    - Entsteht bei einer vorzeichenlosen Subtraktion, wenn der Minuend kleiner als der Subtrahend ist.

    - Bedingung: Minuend $<$ Subtrahend oder indirekt durch Borrow $=$ ¬Carry.
  - **Carry (Überlauf bei Addition):**
    - Entsteht bei einer vorzeichenlosen Addition, wenn das Ergebnis die maximale Wortgröße überschreitet.
    - Bedingung: Ergebnis $\geq 2^n$ (für n-Bit Werte) oder bei Subtraktion Borrow $= 1$.
  - **Overflow (Überlauf bei vorzeichenbehafteten Operationen):**
    - Entsteht bei vorzeichenbehafteten Operationen, wenn das Ergebnis außerhalb des darstellbaren Bereichs liegt.
    - Bedingung bei Addition:

      $$\text{Overflow} = (\text{Vorzeichen}_{\text{Operand1}} = \text{Vorzeichen}_{\text{Operand2}}) \wedge (\text{Vorzeichen}_{\text{Ergebnis}} \neq \text{Vorzeichen}_{\text{Op}}$$

  - **Negative (Negativ-Flag, Vorzeichen-Bit):**
    - Entsteht bei vorzeichenbehafteten Operationen, wenn das höchste Bit (MSB) des Ergebnisses 1 ist.
    - Bedingung: Negative $= (\text{Ergebnis}[n-1] = 1)$ für ein n-Bit Ergebnis.
  - **Zero (Null-Flag):**
    - Entsteht, wenn das Ergebnis einer Operation exakt 0 ist.
    - Bedingung: Ergebnis $= 0$.
  - N (Negative)
  - Z (Zero)
  - C (Carry)
  - V (Overflow)
- Control Unit with Instruction Register (IR)
  - Machine code (opcode) of instruction that is currently being executed
  - Controls execution flow based on instruction in IR
  - Generates control signals for all other CPU components
- Bus Interface (Tor Zur Welt)
  - Interface between internal CPU bus and external system-bus
  - contains registers to store addresses

2.1.3 Instruction Set

Processors Interpret binary coded instructions (but that is hard to read therefore instructions in human readable text form (assembly)
For specific Instructions see extra doc (CT1 Handout)

- Label (optional)
- Operands
- Instruction (Mnemonic)
- Comment (optional)

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
			; store result in R0
	LDR	R2,=0x2000	; load 0x2000 into R2
	STR	R0,[R2]	; store content of R0 at
			; the address given by R2

Abbildung 9: Assembly Program

- Label = noch unbekannte Adresse

Instruction Types Thumb encoded (16 bit jeweils)

- Data transfer
  - Copy content of one register to another register
  - Load registers with data from memory
  - Store register contents into memory
- Data processing
  - Arithmetic operations → + - * / ...
  - Logic operations → AND, OR, ...
  - Shift / rotate operations
- Control flow

  - Branches
  - Function calls

- Miscellaneous (various)

2.2 Program Execution

CODE: Startadresse: 0x20001800
Länge: 1024 Bytes (0x400)
Endadresse: 0x20001800 + 0x400 - 1 = 0x20001BFF

DATA: Startadresse: 0x20001C00 (direkt nach CODE)
Länge: 256 Bytes (0x100)
Endadresse: 0x20001C00 + 0x100 - 1 = 0x20001CFF

STACK: Startadresse: 0x20001D00 (direkt nach DATA)
Länge: 512 Bytes (0x200)
Endadresse: 0x20001D00 + 0x200 - 1 = 0x20001EFF

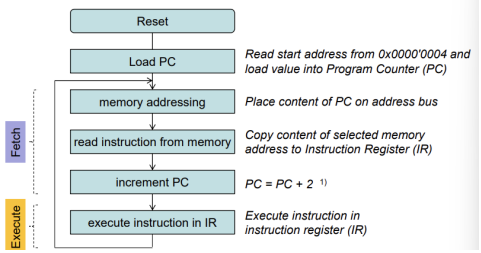


Abbildung 10: Program Execution Sequence

2.3 Memory Map

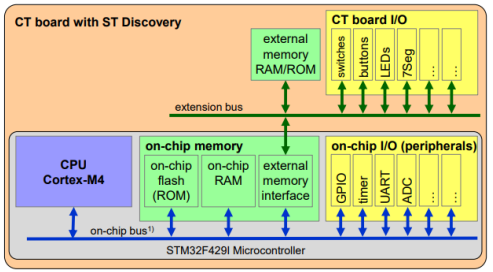


Abbildung 11: Hardware View

2.3.1 Memory Layout

- System Address Map
- Graphical layout of main memory
- Linear array of bytes
- What is located where (at which address) in memory?
  - Location of RAM (readable and writable)
  - Location of ROM (only readable)
  - Location of I/O registers

2.3.2 Address Allocation

First ARM Policies, then ST design decisions, then CT Board Design decisions

2.3.3 CT Board

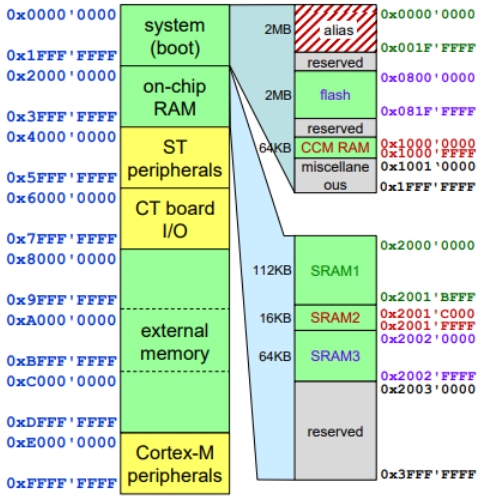


Abbildung 12: CT Board Addresses

3 Vorlesung 03

3.1 Transfer Types

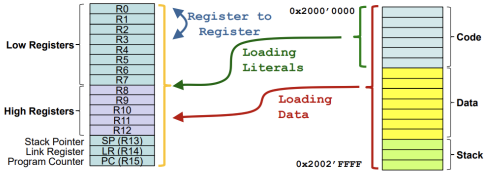


Abbildung 13: 4 Transfer Types

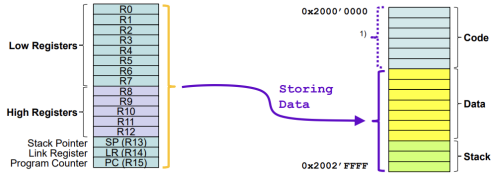


Abbildung 14: Storing Data Type 4

3.2 Casting: Sign Extension

0x82 (8-bit signed)

- **Binärdarstellung:**

  $$1000\ 0010$$

  Das MSB (1) zeigt, dass die Zahl negativ ist.

- **Interpretation als 8-Bit signed:** Verwenden des Zweierkomplements:

  $$-\left(2^7 - (0b000\ 0010)\right) = -\left(128 - 2\right) = -126$$

- **Sign-Extension auf 16-Bit:** Das MSB bleibt 1, also wird mit 1 aufgefüllt:

  $$1111\ 1111\ 1000\ 0010$$

- **Hexadezimaldarstellung:**

  $$0xFF82$$

3.3 Most common Codewords

- MOV / MOVS (register)
  - Copy register value to other register
  - MOV $\rightarrow$ low and high registers
  - MOVS $\rightarrow$ only low registers S = update of flags 3)
- MOVS (immediate data)

  - Copy immediate 8-bit value (literal) to register (only low registers)
  - 8-bit literal is part of opcode (imm8)
  - Register-bits 31 to 8 set to 0
- EQU - Assembler Directive
  - Symbolic definition of literals and constants
  - Comparable to #define in C
- Load - LDR (literal)
  - Indirect access relative to PC 1)
  - PC offset <imm>
  - If PC not word-aligned (align on next upper word-address)
- LDR (immediate offset) – general
  - Indirect addressing
    - Immediate offset <imm>
    - Offset range 0 - 124d (0x7C) 1)
  - Only low registers
- LDRB (register/immediate offset)
  - Load Register Byte
  - Register bits 31 to 8 set to zero
- LDRH (register/immediate offset)
  - Load Register Half-word
  - Register bits 31 to 16 set to zero
- LDRSB
  - Load Register Signed Byte
  - Sign extend
    - Register bits 31 to 8 set or reset depending on bit 7
- LDRSH
  - Load Register Signed Half-word
  - Sign extend
    - Register bits 31 to 16 set or reset depending on bit 15
- STR (immediate offset)
  - Indirect addressing with immediate offset
    - Offset range 0 - 124d (0x7C) 1)
  - Only low registers
- STR (register offset)
  - Indirect addressing with offset register
    - Offset register $\rightarrow$ index
  - Only low registers
- STRB (immediate/register offset)
  - Store Register Byte
  - Low 8 bits of register stored
- STRH (immediate/register offset)
  - Store Register Half-word
  - Low 15 bits of register stored

4 Vorlesung 04 & Vorlesung 05

Instructions to process data in the ALU

- **arithmetic** Addition, Subtraction, Multiplication, Division
- **logic** NOT, AND, OR, XOR
- **shift** Shift left/right. Fill with 0 or MSB
- **rotate** Cyclic shift left/right: What drops out enters on the other side

4.1 Data flow (Cortex-M0)

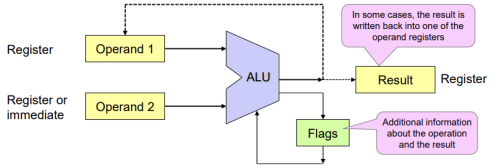
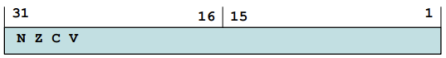


Abbildung 15: Dataflow Cortex-M0



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

Abbildung 16: Cortex-M0 Flags

As Processor does not know signed/unsigned => always calculates both

Mnemonic	Instruction	Function
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 - NOT(c) [1]
RSBS	Reverse Subtract (negative)	-1 • A
MULS	Multiplication	A • B

Abbildung 17: Cortex-M0 Instructions

ADD / ADDS (Summary)

- ADD Flags not changed
- ADDS Flags changed according to Operation Result

Instr	Rd	Rn	Rm	imm	Restrictions
ADD	R0-R15	R0-R15	R0-R15	-	Rd and Rn must specify the same register. Rn and Rm must not both specify the PC (R15)
ADDS	R0-R7	R0-R7	-	0 - 7	-
ADDS	R0-R7	R0-R7	-	0 - 255	Rd and Rn must specify the same register
ADDS	R0-R7	R0-R7	R0-R7	-	-

Abbildung 18: ADD / ADDS Summary

- **unsigned** Interpretation
  - Program must check carry flag (C) after operation
  - C = 1 for Addition C = 0 for Subtraction
    - Result cannot be represented (not enough digits / no negative numbers) Full turn on number circle must be added or subtracted => odometer effect
  - Overflow flag (V) irrelevant
- **signed** Interpretation
  - Program must check overflow flag (V) after operation
  - V = 1 means
    - Not enough digits available to represent the result
    - Full turn on number circle must be added or subtracted => odometer effect
  - Carry flag (C) irrelevant

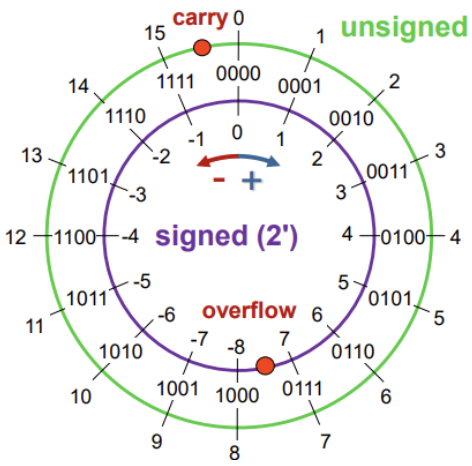


Abbildung 19: Addition Clock = Subtraction Clock

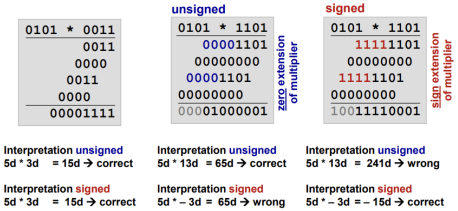


Abbildung 20: Signed and Unsigned Multiplication

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

Arithmetic Instructions

- `ADD/ADDS` `ADCS` `ADR`
  `SUB/SUBS` `SBCS` `RSBS`
  `MULS`

5 Vorlesung 06

5.1 Shift / Rotate

Input als einzelne Bits ansehen (nicht wie arithmetic wo das ganze als zahl angesehen wird)

Mnemonic	Instruction	Function	C-Operator
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

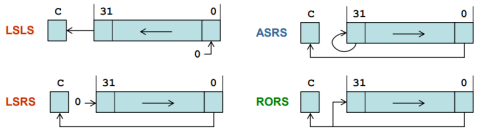
flags N = result<31> 1)
Z = 1 if result = 0
Z = 0 otherwise
C and V unchanged

Abbildung 21: Bitwise Operations

All these operations affect the flags: (Only low registers)

- N and Z according to result
- C and V unchanged

Mnemonic	Instruction	Function
LSLS	Logical Shift Left	2^n • Rn 0 → LSB
LSRS	Logical Shift Right	2^{-n} • Rn 0 → MSB
ASRS	Arithmetic Shift Right	2^{-n} • ±A MSB → MSB
RORS	Rotate Right	LSB → MSB



Note: rotate left does not exist

Abbildung 22: Shift / Rotate Instruction

flags N = result<31>

- Z = 1 if result = 0
- Z = 0 otherwise
- C = last bit shifted out
- V unchanged

5.2 Branch Instructions

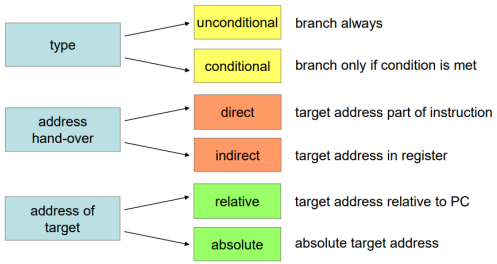


Abbildung 23: Overview Branch Instructions

5.2.1 Unconditional Branches

- B: direct, relative
- BX: (Branch and Exchange) indirect, absolute

5.2.2 Conditional Branches

- Flag dependent branches: Based on one specific flag
- Arithmetic branches: Based on one or more flags

	Symbol	Condition	Flag
B	EQ	Equal	Z == 1
B	NE	Not equal	Z == 0
B	CS	Carry set	C == 1
B	CC	Carry clear	C == 0
B	MI	Minus/negative	N == 1
B	PL	Plus/positive or zero	N == 0
B	VS	Overflow	V == 1
B	VC	No overflow	V == 0

Abbildung 24: Conditional Branches: Flag Dependent

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

Abbildung 25: Arithmetic Unsigned (Add B everywhere)

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

Abbildung 26: Arithmetic Signed (Add B everywhere)

5.2.3 Compare and Test

CMP (compare)

- Same as SUBS, but without
- storing a result!
- Compare 2 operands
  - Higher/lower?
  - Greater/less?
  - Equal?
- Only flags are affected!
- Registers unchanged
- T2 also higher registers

CMN Compare

- Same as ADDS, but without storing result!
- Compare 2 operands negative
- Only flags are affected!
- Registers unchanged
- Read CMN as: Is content of Rm equal to 2's complement of Rn?

TST Test

- Is a specific bit set?
- Logical AND without storing result
- Registers unchanged
- Changes only flags N and Z (C and V unchanged)

6 Vorlesung 07

Structured Programming: (standard is sequence)

- Selection: if – then - else
- Loops: Do – While / While / For (Pre-/Post-Test Loop)
- Switch Statements (case)

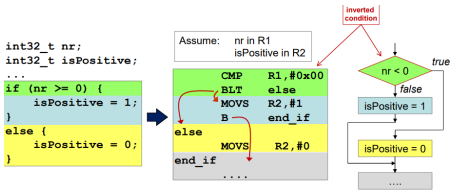


Abbildung 27: if-then-else

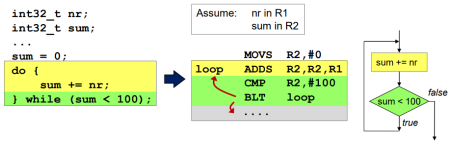


Abbildung 28: Do-While Loop

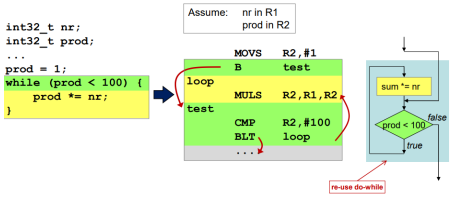


Abbildung 29: while Loop (or for loops translated to this too)

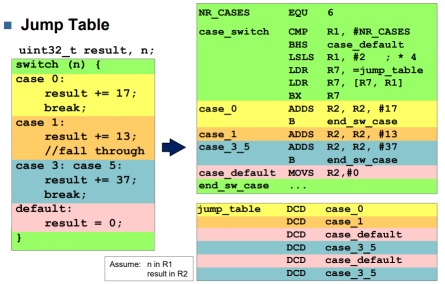


Abbildung 30: Switch Statements

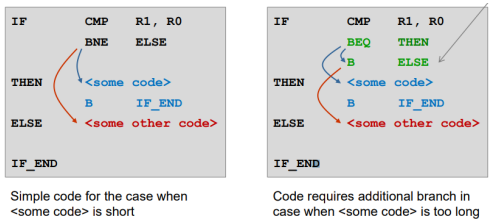


Abbildung 31: Limited Range of-256..254 Bytes

7 Vorlesung 08 (09 was zwprüfung)

7.1 Subroutine = Procedure = Function = Method

7.1.1 ARM Terms:

Routine, subroutine: A fragment of program to which control can be transferred that, on completing its task, returns control to its caller at an instruction

following the call. Routine is used for clarity where there are nested calls: a routine is the caller and a subroutine is the callee.
Procedure: A routine that returns no result value.
Function: A routine that returns a result value.

7.1.2 Functionality of Subroutines

- **PC**: Zeigt auf die Adresse, von der der nächste Befehl gelesen wird
- **SP**: Zeigt auf die Adresse, an der sich die zuletzt auf den Stack geschriebenen Daten befinden
- **LR**: Hierin befindet sich die Adresse von der, nach Beenden einer Funktion, fortgefahren wird.
- **Storing Data:** Writing data from a CPU register into a memory location, often using specific instructions to transfer values to slower, larger storage.
- **Loading Literals:** Directly placing a constant value into a register to initialize or update it for immediate use in computations.
- **Register-to-Register:** Transferring data directly between two CPU registers, enabling fast operations without accessing memory.
- **Loading Literals (Expanded):** Utilizing immediate operands in load instructions to efficiently set large or small constants in a register.
- Call Save PC to Link Register (LR) (Instruction BL / BLX)
- Return Restore PC from LR

7.1.3 Structure

- Label with Name (e.g. MulBy3)
- Return Statement (BX LR)
- Assembler Directives
- PROC / ENDP
- FUNCTION / ENDFUNC

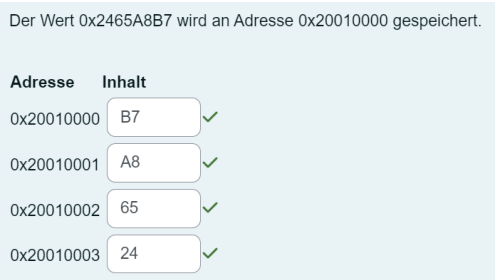


Abbildung 32: Little Endian Usage

7.1.4 Nested Subroutines

to implement (nested) subroutines in assembly: Stack is needed as LR would be overwritten
Nested subroutines => save LR on stack

7.2 Stack

- Methods: Push() & Pop()
- Lowest register stored to lowest address (little endian) Eine Multibyte Darstellung bei der das LSByte an der unteren Adresse liegt
- Number of PUSHs- Number of POPs"
- Stack-limit < SP < stack-base
- Last-in First-Out

Wenn der Speicherinhalt als big-endian abgelegt ist, bedeutet das, dass der höchstwertige Byte (Most Significant Byte, MSB) an der niedrigsten Adresse gespeichert wird und der niedrigstwertige Byte (Least Significant Byte, LSB) an der höchsten Adresse.

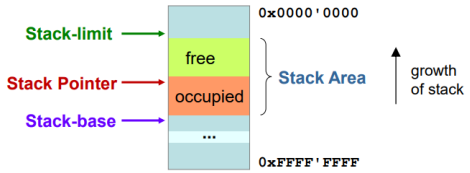


Abbildung 33: Stack Structure

8 Vorlesung 10: Parameter Passing

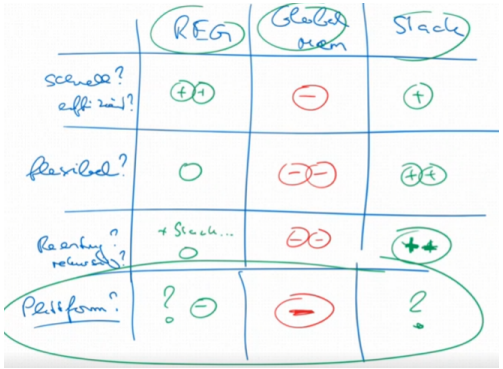


Abbildung 34: Parameter Passing Options

- R0-R3: First to fourth parameters.
- R4-R7: Callee-saved registers.
- R8-R12: Temporary registers (not saved by the callee).
- R13 (SP): Stack pointer.
- R14 (LR): Link register (return address).
- R15 (PC): Program counter.
- R0: Return value (if 32-bit).
- R1: Return value (if part of a larger value).

Register	Responsibility	Saved by/Restored by
R0-R3 (function call)	Holds first four arguments	Saved by **caller** (if needed)
R4-R7 (function call)	Used for callee-saved data	Saved by **callee**
R0-R3 (return)	Holds return values	**Not restored** (caller-saved)
R4-R7 (return)	Must be preserved by callee	Restored by **callee**
LR (function call)	Holds return address	Saved by **caller** (if needed)
LR (return)	Return address to caller	Restored by **callee**

Abbildung 35: ARM Procedure CallStandards

- Where?
  - Register
    - Caller and Callee use the same register
  - Global variables
    - Shared variables in data area (section)
    - Error prone, unmaintainable (no encapsulation, many dependencies)
  - Stack
    - Caller => PUSH parameter on stack
    - Callee => access parameter through LDR <Rt>,[SP,#<imm>]
- How?
  - pass by value: Handover the value
  - pass by reference: Handover the address to a value

- **Parameters**
  - Caller copies arguments to R0 to R3
  - Caller copies additional parameters to stack
- **Returning fundamental data types**
  - Smaller than word — zero or sign extend to word; return in R0
  - Word — return in R0
  - Double-word — return in R0 / R1 [1)]
  - 128-bit — return in R0 – R3 [1)]
- **Returning composite data types** ***(structs, arrays, ...)***
  - Up to 4 bytes — return in R0
  - Larger than 4 bytes — stored in data area; address passed as extra argument at function call

Abbildung 36: ARM Procedure Call Standard

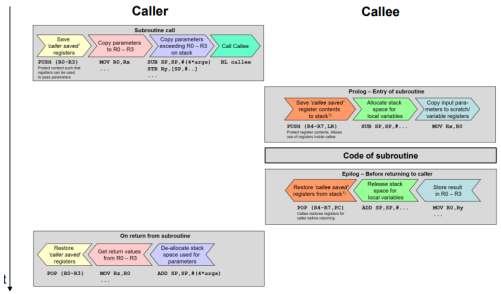
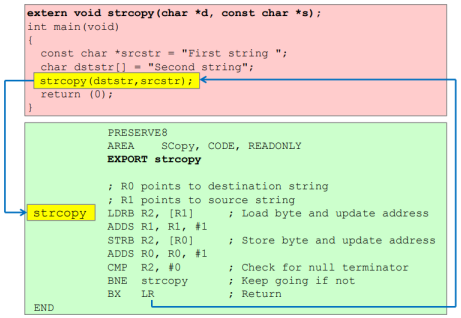


Abbildung 37: Flow of Execution for a Subroutine Call



```
extern void strcopy(char *d, const char *s);
int main(void)
{
  const char *srcstr = "First string ";
  char dststr[] = "Second string";
  strcopy(dststr,srcstr);
  return (0);
}
```

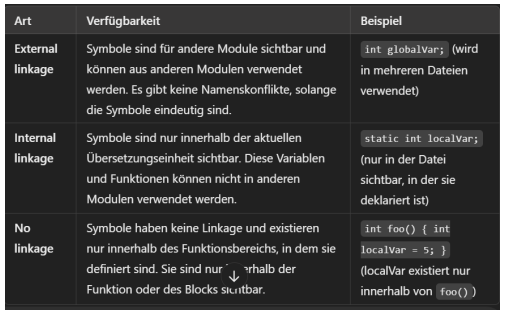
```
            PRESERVE8
            AREA    SCopy, CODE, READONLY
            EXPORT strcopy

            ; R0 points to destination string
            ; R1 points to source string
strcopy     LDRB R2, [R1]     ; Load byte and update address
            ADDS R1, R1, #1
            STRB R2, [R0]     ; Store byte and update address
            ADDS R0, R0, #1
            CMP  R2, #0       ; Check for null terminator
            BNE  strcopy      ; Keep going if not
            BX   LR           ; Return
 END
```

Abbildung 38: Calling Assembly Subroutines from C

9 Vorlesung 11: Modular Coding/Linking

9.1 Translation Steps

- Compile/assemble each module
  - Results in an object file for each module (module.o)
- Link all object files
  - Results in one executable file



Art	Verfügbarkeit	Beispiel
External linkage	Symbole sind für andere Module sichtbar und können aus anderen Modulen verwendet werden. Es gibt keine Namenskonflikte, solange die Symbole eindeutig sind.	`int globalVar;` (wird in mehreren Dateien verwendet)
Internal linkage	Symbole sind nur innerhalb der aktuellen Übersetzungseinheit sichtbar. Diese Variablen und Funktionen können nicht in anderen Modulen verwendet werden.	`static int localVar;` (nur in der Datei sichtbar, in der sie deklariert ist)
No linkage	Symbole haben keine Linkage und existieren nur innerhalb des Funktionsbereichs, in dem sie definiert sind. Sie sind nur innerhalb der Funktion oder des Blocks sichtbar.	`int foo() { int localVar = 5; }` (localVar existiert nur innerhalb von `foo()`)

Abbildung 39: External, Internal and No Linkage

9.2 ARM assambly IMPORT and EXPORT

- Linkage control
  - EXPORT declares a symbol for use by other modules
  - IMPORT declares a symbol from another module for use in this module
- Internal symbols
  - Neither EXPORT nor IMPORT
  - Defined in this module
  - Can only be used within this module

9.3 Linker

9.3.1 Linker Tasks

- Merge code sections
  - Place all data sections of the individual object files into one data section of the executable file
- Merge data sections
  - Place all code sections of the individual object files into one code section of the executable file

- Symbol resolution: References to other modules
  - Search missing addresses of used external symbols
- Address relocation: Adapt to new positions of symbols
  - Adjust used addresses since merging the sections invalidated the original addresses

9.3.2 .LST File

- Das .lst-File wird nach dem Compilieren vom Assembler erzeugt.
- Aus dem .lst-File kann man bereits exakt lesen, wieviel Platz die AREAs eines Moduls benötigen werden.
- Im .lst-File können einige Instruktionen noch nicht fertig assembliert sein, weil dem Assembler gewisse Informationen fehlen.

9.3.3 .MAP File

- Die Bezeichnung 'map' (=Karte) bezieht sich darauf, dass hier dokumentiert ist, welches Symbol sich wo im Speicher befindet.
- Das .map-File enthält Informationen über den Linking-Prozess.
- Im .map-File sind die Adressen aller Objekte wie AREAs, Variablen und Funktionen so gelistet, wie sie auch im Debugger erscheinen.

9.3.4 Linker Input: Object Files / Linker Output: Executable File

Contain all compiled data of a module (Code, Data - Section, Symbol, Relocation - Table)

9.4 Tool Chain

The set of tools that is required to create from source code an executable for a given environment

- Native tool chain
  - Builds for the same architecture where it runs on
- Cross compiler tool chain
  - Builds for another architecture than the one it runs on
  - E.g. build in KEIL (on Windows) for the CT Board (bare-metal ARM)

9.5 Libraries

- Static libraries
  - Executable is completely linked with a static library at link time
  - The resulting executable is self contained
  - No need for any other libraries at run time
- Dynamic or shared libraries
  - Executable is not linked with a dynamic library at link time
  - The resulting executable is not self contained
    - Needs other libraries at run time
    - Loader of hosting OS links at load time with the shared libraries

9.6 Debugging

- Single stepping
  - Support by the HW (stops processor, provides register access)
  - Support by SW (swap instructions with a breakpoint instruction)
- Source level debugging
  - Source level debugging needs mapping between machine address and source code line and memory locations and source code lines
  - Mapping information is often provided in object fiels (e.g. ELF files)

10 Vorlesung 12 Exceptional Control Flow

10.1 Polling (Periodic Query of Status Information

Reading of status registers in loop, Synchronous with main program

Advantages

- Simple and straightforward
- Implicit synchronization
- Deterministic
- No additional interrupt logic required

Disadvantages

- Busy wait => wastes CPU time
- Reduced throughput
- Long reaction times

10.2 Interrupt Driven I/O

10.2.1 Ausschalten und Einschalten von Interrupts

Ausschalten von Interrupts:

- C: `__disable_irq();`
- Assembly (ARM): `CPSID I`

Einschalten von Interrupts:

- C: `__enable_irq();`
- Assembly (ARM): `CPSIE I`

Main: Initializes peripherals, Afterwards it executes other tasks, Peripherals signal when they require software attention (phone call analogy), Events interrupt program execution

Advantages

- No busy wait => better use of CPU time
- Short reaction times

Disadvantages

- No synchronization (between main program and ISRs)
- Difficult debugging
- Chage of Program Flow

10.3 Exceptions Cortex M3/M4

Exception Number	Exception Type	Priority	Description
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

Abbildung 40: System Exceptions

10.4 Interrupt System Cortex-M3/M4

Nested Vectored Interrupt Controller (NVIC) 240 sources can trigger exception => high level signal on IRQx, Forwards respective exception number to CPU

- enable and disable interrupts
- set and clear interrupts by software
- prioritize exceptions (same priority => lower number)

CPU

- Calculates vector table address based on exception number
- Uses address to read vector from memory
- Stores context on stack (interrupt can take place on any time, saves R0-R3, PC, LR, PSRs)
- Loads vector into PC => branch to ISR

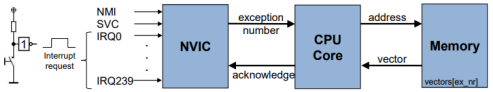


Abbildung 41: Interrupt Process

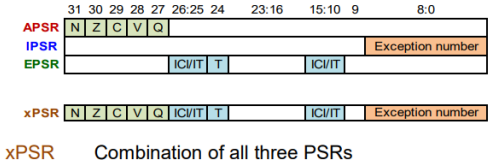


Abbildung 42: PSRs

Program Status Registers (PSRs)

10.5 NVIC

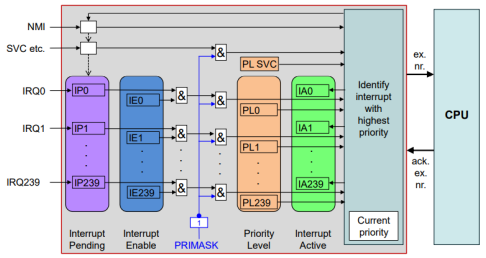


Abbildung 43: NVIC Aufbau

11 Vorlesung 13

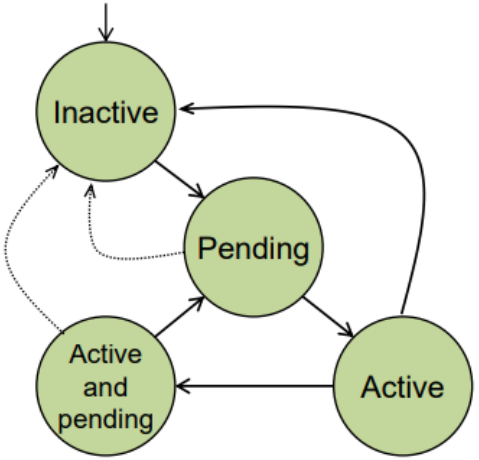


Abbildung 44: Exception States (active and pending = pending exception for same source)

11.1 Interrupt Control

11.1.1 Interrupt Pending Registers

- Trigger hardware interrupt by software => set pending bit
- Cancel a pending interrupt => clear pending bit

11.1.2 Interrupt Active Status Registers

- Read-only
- Corresponding bit is set when ISR starts
- Corresponding bit is cleared when interrupt return is executed

11.1.3 Masking of Inputs

General Masking of Interrupts (PRIMASK)

- Single bit controlling all maskable interrupts
- Disable set PRIMASK
- Enable clear PRIMASK
- On reset PRIMASK = 0 => enabled

Non-Maskable Interrupt (NMI) Power-fail, emergency button, watchdog, ...

11.1.4 Interrupt Enable Registers

Individual masking of interrupt sources

- IEn cleared pending bit not forwarded
- IEn set interrupt enabled

11.2 Register View

- 1 Bit for each IRQ
- Registers are written as 32-bit words
- Only bits with value “1” have an effect, “0” is ignored
- Separate registers to read and write

11.3 Nested Exceptions

Preemption Assigned priority level for each exception Fixed priorities: Reset (-3), NMI (-2), hard fault (-1) (others are programmable) 4-bit priority level 0x0 – 0xF

11.4 Special Interrupt Situations

- IRQ request stays active: Interrupt becomes pending again
- Common Configuration on Microcontrollers: IRQ Set by Hardware – Cleared by Software
- Multiple IRQ request pulses before entering ISR: Treated as single interrupt, Events are lost

11.5 CMSIS (Cortex Microcontroller Software Interface Standard)

```
void NVIC_EnableIRQ(IRQn_t IRQn)                 Enable IRQn
void NVIC_DisableIRQ(IRQn_t IRQn)                Disable IRQn

uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)        Return true (IRQ-Number) if IRQn is pending
void NVIC_SetPendingIRQ (IRQn_t IRQn)            Set IRQn pending
void NVIC_ClearPendingIRQ (IRQn_t IRQn)          Clear IRQn pending status
uint32_t NVIC_GetActive (IRQn_t IRQn)            Return '1' if active bit of IRQn is set. '0' otherwise

void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)  Set priority for IRQn
uint32_t NVIC_GetPriority (IRQn_t IRQn)          Read priority of IRQn

void NVIC_SystemReset (void)                     Reset the system
```

Abbildung 45: NVIC Control

11.6 Data Consistency

2 Datenzugriffe auf die selbe Struktur erreichen nicht zwingend dieselbe struktur

- Data structure must not be changed during output
- Disable Interrupts during output
- Multitasking problem

12 Vorlesung 14 Improving System Performance

12.1 Aspects of Optimization

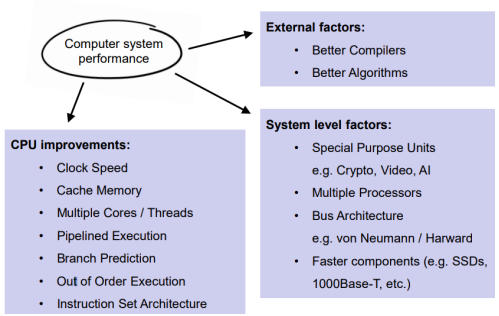


Abbildung 46: Computer system performance

12.2 Bus Architectures

von Neumann Architecture:

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

Harvard Architecture

- Separate memories for program and data
- Two sets of address/data buses between CPU and memory

12.3 Instruction Set Architectures (ISA)

12.3.1 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 accesses,
- CPU works at full-speed on registers
- Load / Store Architecture
- Data processing instructions only available on registers
- fixed length instructions

12.3.2 CISC (Complex Instruction Set Computer)

One of the operands of an instruction may directly be a memory location variable length/complexity Less program memory needed with complex instructions Short programs may work faster with less memory accesses

12.4 Pipelining

fetch the next instruction, while the current one decodes: $instructions/s = 1/maxStageDelay$ (without: $Instructions/s = 1/instructionDelay$)

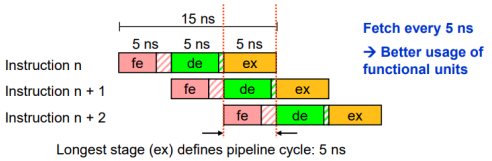


Abbildung 47: Pipelined execution

- Advantages
  - All stages are set to the same execution time
  - Massive performance gain
  - Simpler hardware at each stage allows for a higher clock rate
- Disadvantages
  - A blocking stage blocks whole pipeline

- Multiple stages may need to have access to the memory at the same time

12.4.1 Special Situation: LDR

- 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

12.5 Parallel Computing

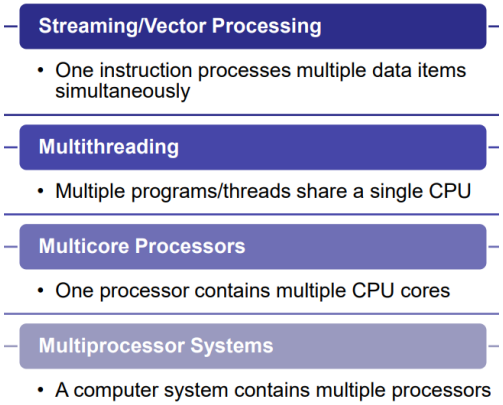


Abbildung 48: Parallel Computing

12.5.1 Parallelism on CPU / Processor Level

Multicore processor

- All on one chip
- Less traffic (cores integrated on one chip)
- Possibility to share memories on-chip
- Cheaper

Multiprocessor

- Multiple Chips
- Longer distances between CPUs
- More expensive