Kombinatorische Logik

AND — & — Z = A & B

Invertor — Z = ! A

Buffer — Z = A

OR — ≥1 — Z = A # B

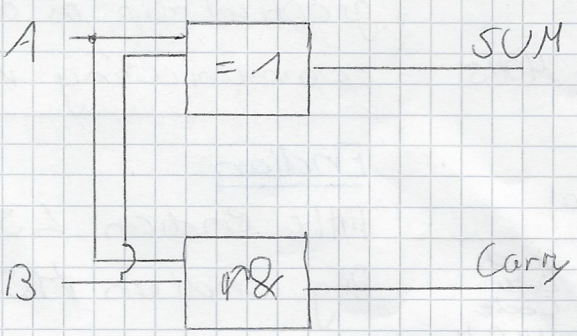
NAND — & o — Z = ! (A & B)

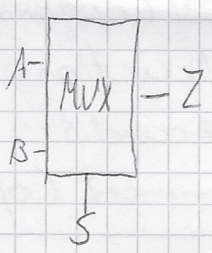
NOR — ≥1 o — Z = ! (A # B)

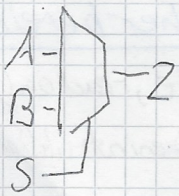
XOR — =1 — Z = A $ B

XNOR — =1 — Z = ! (A $ B)

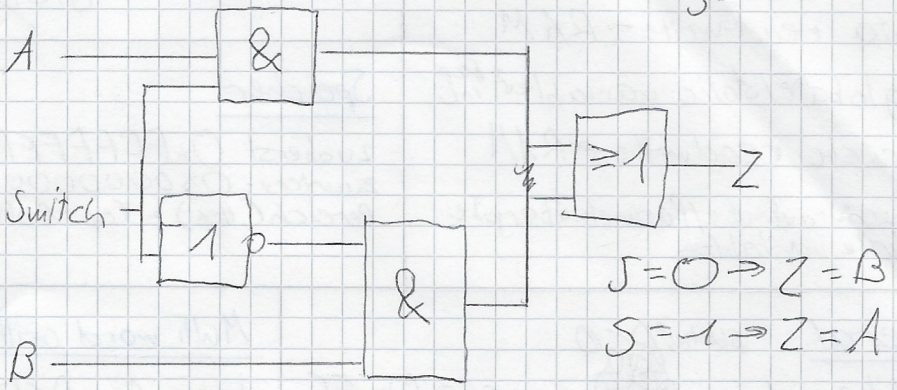
1 Bit Halfaddierer







Multiplexer



S = 0 → Z = B

S = 1 → Z = A

Komb vs sequenz. Logik

seq: memory

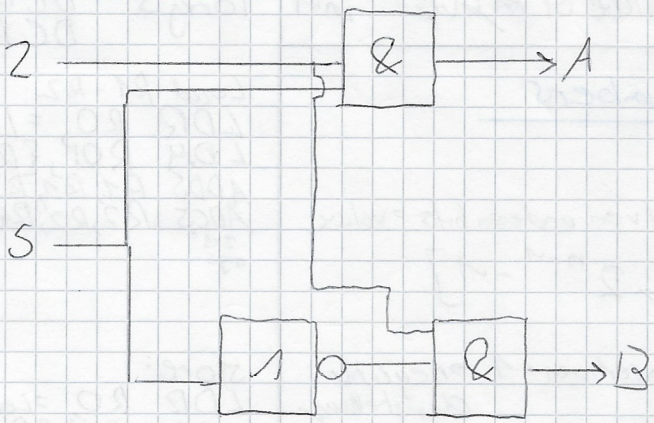
output: input + memory

= komb + clock

kom: no memory

output: input dep.

Demultiplexer



A = (Z & S)

B = (Z & ! S)

M0 CPU

Core Registers 16 × 32bit Registers

ALU Rechenwerk log + arithmetische Operationen

Control Unit (IR) enthält OP code von momentaner auszuführender Aktion

Bus Interface Schnittstelle zu externem System Bus

Flags Status von letzter ALU Rechnung

Registers

R0 - R7 Low Regs

R8 - R12 High Regs

R15 PC Program Counter

R13 SP Stack Pointer

R14 LR (Link Register Rücksprungadresse)

ASPR Reg: N Z C V

N neg, Z zero, C carry, V overflow

Assembly instruction

Label (optional)
instruction 2.B MOVS
Operands
comment (optional)

memory map

graphical map of addresses and sizes of elements communicating with CPU

Endian

little endian LSB an tiefster Stelle in Address
Big endian MSB an tiefster Stelle in Address

little → big : Swap addresses; value in address stay the same

Speicherbereiche

Stack
Data
Code

Code, read only RAM/ROM
Maschineninstruktionen, Konstanten
Data read write → RAM
- globale/static variablen + heap in C
Stack read write → RAM
- Prozeduraufrufe/Parameterübergabe
lokale variablen

Speicher

zuoberst 0xFFFFFFFF
zuunterst 0x00000000
Bereich (Hex) = Top - Bottom + 1

Load/Store

access of memory via load/store

Literal

global, "DCD"
=literal DCD myvar = 0x55

LDR R1, =myvar
=0x55
loads the value of myvar in R1

Multi word arithmetic

```
long1  DCD  ...  ;low
       DCD       ;high
long2  DCD       low
       DCD       high
long3  DCD       low
       DCD       high
```

Load R1 + R2
LDR R0, =Long1
LDM R0!, {R1, R4} → Starts reading in R0 and saves in R1–R4
ADDS R1, R2, R3 // lower
ADCS R2, R2, R4 // higher part with carry (2+3 args) → R1 long1 low, R2 long1 high, R3 long2 low, R4 long2 high

store:
LDR R0, =Long3
STM R0!, {R1, R2}

Negative Numbers

0 = pos

minus
± MSB + Wert von anderen bits = value

anzeigbar $[-2^{n-1}, 2^{n-1}-1]$

2er Komplement : invertieren + 1 → negative Darstellung

RSBS in Assembler

ASPR Add/Sub

unsigned: C check after operation
- addition C = 1 → falsche Ergebnisse
sub C = 0
→ ganzen Zahlenkreis addieren/subtrahieren

signed: V check after operation
V = 1 → not enough digits
full turn of numbercycle + oder -
C irrelevant

ASPR

Addition unsigned: C = 0 → Borrow
V irrelevant
Sub von small may result in big

signed V = 1 → over/underflow
not possible when operands same sign
C irrelevant

MULS <Rdm>, <Rn>, <Rdm>

Rdm = Rn * Rdm
Flags N, Z update
C, V unchanged

MULS von Hand

unsigned: add Left zeros
signed: add left ones

Casting

Casting von negativen Zahlen → Fehler
small pos negative → grosse positive
high positive → small negative

Shift

low regs only

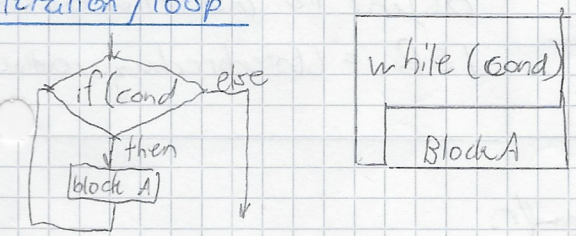
LSLS logical Shift Left — leftmost in carry, rightfill with 0's
$2^n \cdot R_n$

LSRS Logical Shift Right — rightmost in carry, leftfill with 0's
$2^{-n} \cdot R_n$

ASRS Arithmetic Shift Right — rightmost in carry, leftfill with MSB
$2^{-n} \cdot \pm A$

RORS rotate right (carrussel) — rightmost in Carray, leftfill with prev. LSB

Iteration / loop



Extension

unsigned: left fill with 0's
signed: left fill with 1's

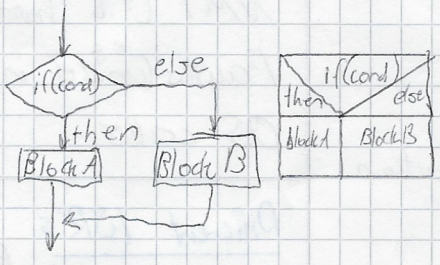
SXTB 8 → 32 } signed
SXTH 16 → 32 }
UXTB 8 - 32 } unsigned
UXTH 16 - 32 }

conditional Branches

CMP = SUBS without just flags
CMN = ADDS just flags

CMP R1, R0
B... siehe Anhänge Folien

Selektion (IF)

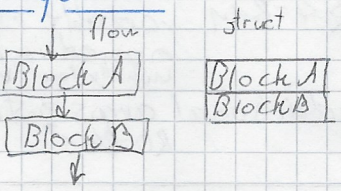


Logic operations

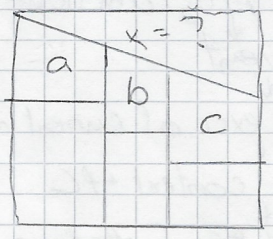
only low Registers <Rd> <Rn> <Rm>
changes N and Z

		Rd=
ANDS	bitwise AND	A&B
EORS	bitwise XOR	A$B
MVNS	bitwise NOT	!A
ORRS	bitwise OR	A#B
BICS	bitclear	A&!B

Sequence



Switch



Ass if then else

```
CMP R1, #0x00
BLT false      } if
MOVS R2, #1    } else
B done
false: MOVS R2, #0x0  } then block
done: ...
```

Ass do while loop

```
Loop: ADDS R2,R2,R1
      CMP R2, #100
      BLT loop
      ...          } end d while
```

While loops

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

Compare

CMP <x>, <y> → x - y
CMN x, y → x + y
y = 2er(-x)?

TST only changes N,Z,C not V
logical AND
x y is specific bit set

Subtraction

C=0 → borrow=1
x - y → x + (2er comp y)
x ≥ y → carry

negativieren

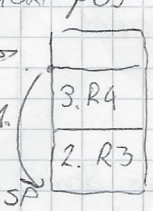
RSBS <Rd> <Rm>

Stack

SP pointer, an unterste Adr
LR = Adresse nach Branch

grows top → botton
Stackpointer on current pos

push {R3, R4} →



push LR
pop PC

pop {R3, R4}
= pop first value into R3
second into R4

add sub with imm 7 possible from stackpointer
BLX store PC in LR

Functions / Subroutines

```
Progc
progc.. endp
function..endfunc
functions have return value
```

ASS EQU

```
TABLELENGTH EQU 32
→TABLELENGTH kann als var gebraucht werden

ADDS R0, R0, TABLELENGTH
```

Ass for loops

translated to while

```
for(init; test, updat-)
    body
→
init
while(test)
    body
    update
```

Parameter Passing

R0-R4 Arguments/scratch
R0-R1 results
R0-R4 might be modified by callee
R4-R11 Callee must preserve content

scratch: hold imm values during calculation
usually unnamed

R4-R11 variable register named in code
usually local variable

additional args → stack
R0-R4 ↗

return values

kleiner als word → extend to word

Double word → R0 R1
LSW MSW

128 bit R0-R4
LSW MSW

composite: up to 32 Bytes in R0
> 4 Bytes → stored in data area, address passed as argument at function call

Interrupt event

main program flow
↓ event
finish exec of current instruction
save context + PC
load PC of attention needing code into PC
do stuff
BX LR
restore context
continue mainflow

System exceptions

Reset
NMI (z.b hardware)
Faults (e.g. under instruction)
OS calls

Interrupts IRQ0 - IRQ239

signal CPU that event need attention

async to instruction exec.

ISR = Unterbrechungsroutine

Different ISR's

	Exception Nr.	
IRQ239	255	Interrupts 0-239
...	...	
IRQ1	17	IRQn = Exceptions n + 16
IRQ0	16	
Systick	15	System exceptions
...	...	
hard fault	3	
NMI	2	
Reset	1	tiefere Nummer = höhere Priorität
Top of stack	0	

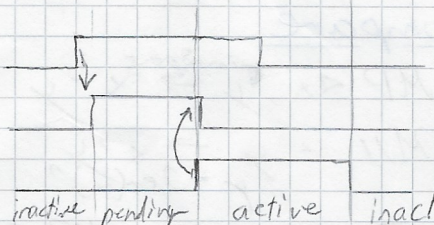
Program Status Registers

APSR APSR Flags
IPSR Interrupt Program status register
EPSR Execution program status register
xPSR combo of all 3 above

Exception states

inactive : not pending
pending : waiting to be served by CPU
active being serviced by cpu but not completed
active + pending : being service + other exception of same
No waiting

IRQ Interrupt request
IPr Interrupt pending
IAn Interrupt Active



sonst nacheinander gleich IRQ → 2. wird vergessen

Masking interrupts

CPSIE (enable)
CPSID (disable)
Change Processor state Interrupt

Nested Exceptions

interrupts of exceptions only possible if following exception has higher priority (lower number) compared to current one. Otherwise waiting to completion of current interrupt.

multiple IRQ request pulses before entering ISR → ignored, just 1 interrupt

CMSIS

Cortex Microcontroller Interface Standart

vendor independent hardware abstraction layer.

Interrupts adv. to polling

no busy wait / short reaction time to 1st I/O

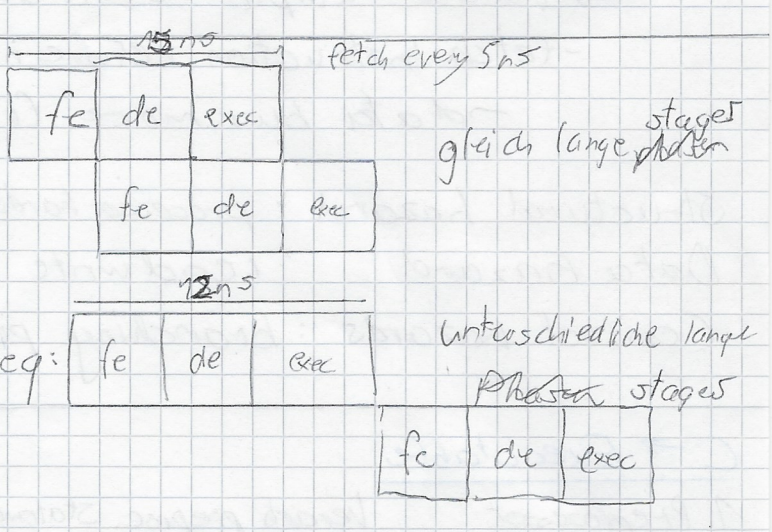
Architecture

Complex instruction set Computer CISC	Reduced instruction set Computer RISC
- traditional memory access	- load/store architecture
- complex addressing	- simple addressing
- High code density	- more lines of code
- most compilers do not support all operations → powerful instructions may not be used	- reduced instruction set → less hardware - higher clockrates
- often require manual optimizing of assembly code for embedded systems	- allows effective compiler optimization
- program needs to wait for external memory	- Program works on registers at full speed

Pipelining

fetch → decode → execute

Reset
↓
memory addressing
↓
read instruction from mem } fetch
↓
inc. PC++
↓
Execute instruction in IR } Execute



Vorteil: massiv perf. gain.
simpler hardware → higher clockrate
all stages same length

general: Number of exec stations is design decisions
typ 2 - 12 stages

instructions/sec sequencial $\frac{1}{\text{Dauer aller Phasen (instruction delay)}}$

pipelay $\frac{1}{\text{stage delay}}$

problem with branches: to outcome of branch for new fetch needed but branch result not known jet

→ remember if branch was taken last time. change prediction after 2 successive mis-predictions

worst case: sequentiell behandeln

pipelining II

instruction use data that is used in different stages

read after	write	RAW	true dependency
write after	read	WAR	anti dependency
write after	write	WAW	output dependency

ex: in execution stage value of ~~written~~ previously write back instruction is needed. Both stages happen at same time

→ forward solution of pre current instruction to the next before writing

→ execution stage of next instruction and write back of current can happen simultaneously

part of processor ~~Bereich~~ hardware is needed by more than one instruction at same time

→ -fetch multiple instruction at once
- fetch instruction before needed
- data bus move force

Structural hazard : processor hardware stuff
Data hazard : read write stuff
Control hazards : branching problem

C → Executable

1. Preprozessor — verarb preproc. statements (#define #include ersetzen) → Textfile mit mod. C Source
2. Compiler — C source → Assemblerbefehle → Textfile mit Assemblercode
3. Assembler — Assembercode → Maschinenbefehle → Binäres Objektfile mit Maschinenbefehlen
4. Linker — mehrere .o files → 1 executable object file auflösen Referenzen → binäres, executable Objektfile

branching poss. direkt/indirekt