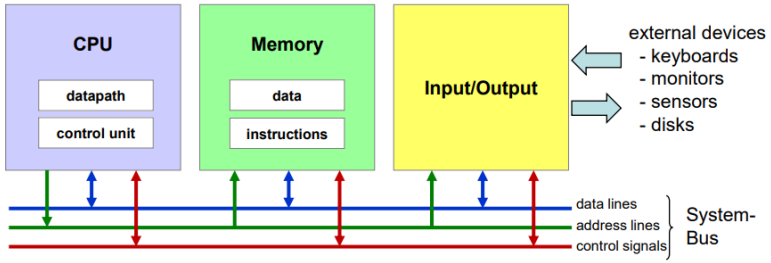
Computer Engineering

Hardware

- CPU — **C**entral **P**rocessing **U**nit
- Memory — Stores instructions and data
- Input / Output — Interface to external devices
- System-Bus — Electrical connection of blocks



Datapath

- ALU — Arithmetic and Logic Unit
- Registers — Fast but limited storage inside CPU

Control Unit

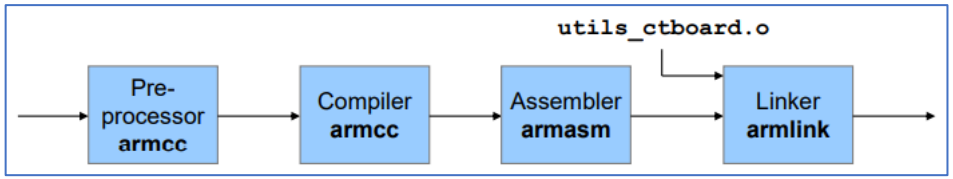
- Finite State Machine — Reads and executes instructions
- Types of instructions — Data transfer, Arithemtic, logical and jumps

Memory

- A set of storage cells
- Smallest addressable unit
- 2^N addresses
  - RAM — read/write
  - ROM — read

Software

From C to executable



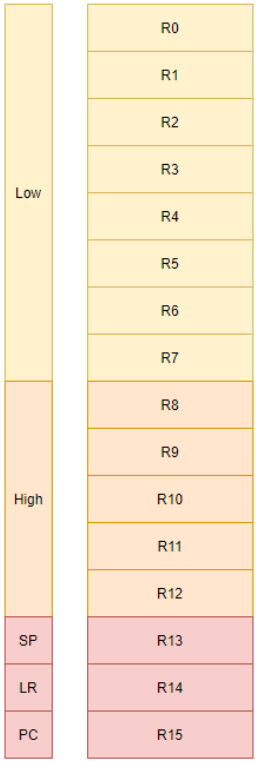
1. Preprocessor
   - Text processing
   - Pasting of *#include* files
   - Replacing macros (*#define*)
2. Compiler
   - Translate CPU-independent C-code into CPU-specific assembly code
3. Assembler
   - Translate to machine instructions
   - Result: Relocatable object file
   - Binary file → not readable with text editor
4. Linker
   - Merge object files
   - Resolve dependencies and cross-references
   - Create executable

Pascal Isliker

Cortex-M Architecture

Registers

- 16 Core Registers
- 32-Bit wide
- R0 – R7 Lower Registers
- R8 – R12 Higher Registers
- R13 **S**tack **P**ointer Temp Storage
- R14 **L**ink **R**egister Return from Procs
- R15 **P**rogram **C**ounter Addr of next Instr.



ALU

- 32-Bit wide processing unit

APSR (Flag Register)

- N **N**egative
- Z **Z**ero
- C **C**arry
- V O**v**erflow

Instruction Set

- 16-Bit Thumb instruction encoding

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

Instruction Types

- Data transfer Move, Load and Store
- Data processing Arithmetic, Logical and Shift operations
- Control flow Branches and functions

Assembly Program Structure

Code **Data** **Stack**

```
            AREA    MyCode, CODE, READONLY
            ENTRY
start       MOVS    R4,#12
            ADDS    R3,R4,#5
            B       start

            AREA    MyData, DATA, READWRITE
byte_var    DCB     0x1A,0x00
hw_var      DCW     0x2B3C
word_var    DCD     0x4D5E6F70

            AREA    STACK, NOINIT, READWRITE
stack_mem   SPACE   0x00000400
```

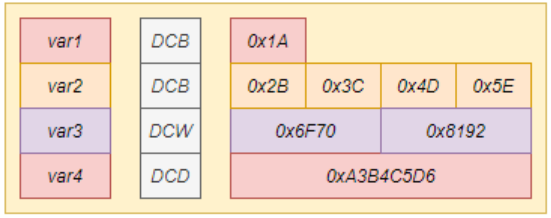
Define code area to include your program

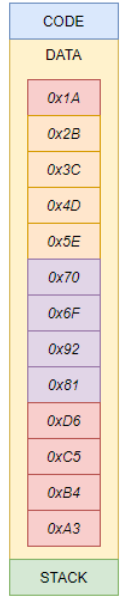
Define data area to store global variables, etc.

Define stack area to reserve space for stack

Directives for *initialized* data

- DCB **B**ytes
- DCW Half-**W**ords
- DCD **W**ords





Directives for *uninitialized* data

- SPACE Bytes to be reserved

Pascal Isliker

Data Transfer Instructions

Loading Data

- MOVS
  - Reg to Reg *MOVS R1, R2*
  - 8-Bit Literal *MOVS R1, #0x1C*
  - Constant *MOVS R1, #MyConst*
- LDR
  - 32-Bit Literal *LDR R1, #0xA1B2C3D4*
  - Literal + Offset *LDR R1, [PC, #12]*
  - Constant *LDR R1, =MyConst*
  - Reg Value *LDR R1, [R2]*
- LDRB
  - Load Register Byte
  - Bits 31 to 8 set to zero
- LDRH
  - Load Register Half-word
  - Bits 31 to 16 set to zero

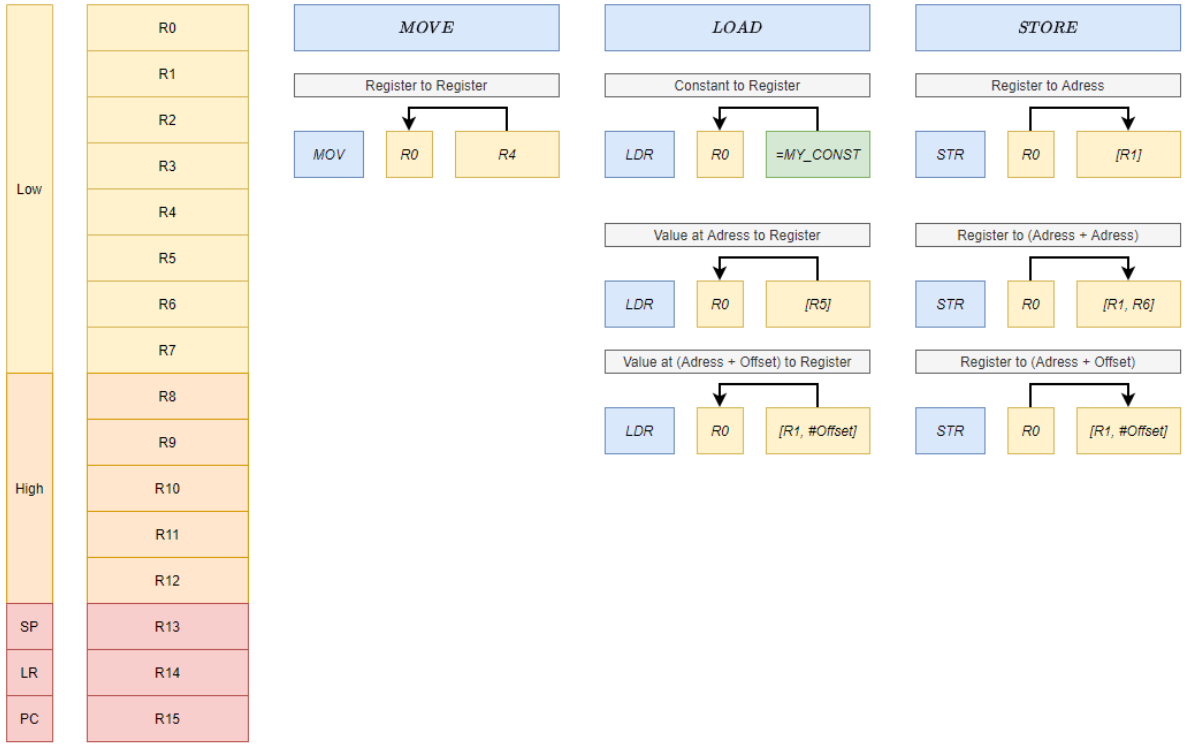
Load Array

- my_array = 3 * 4 Bytes
- Instructions = 5 * 2 Bytes
- Literals (0x08) = 1 * 4 Bytes

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

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

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

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

Not content of my_array, but address of my_array



Storing Data

- STR
  - Value from Register *STR R1, [R2]*
  - Value from Reg + Offset *STR R1, [R2, #0x04]*
- STRB
  - Store Register Byte (Low 8 bits of register stored)
- STRH
  - Store Register Half-word (Low 15 bits of register stored)

Pascal Isliker

Arithmetic Operations

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

Instructions ending with with «S» allow flag modification

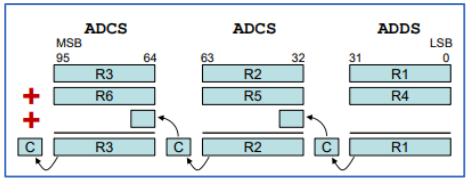
- ADDS
- SUBS
- MOVS
- LSLS

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

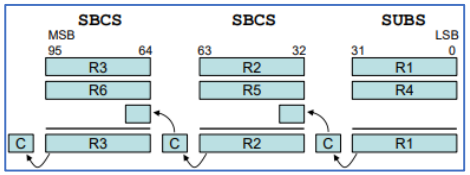
- ADD / ADDS Addition $A + B$
- ADCS Addition with Carry $A + B + c$
- ADR Address to Register $PC + A$
- SUB / SUBS Subtraction $A - B$
- SBCS Subtraction with carry (borrow) $A - B - !\,c$
- RSBS Reverse Subtract (negative) $-1 \cdot A$
- MULS Multiplication $A \cdot B$

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

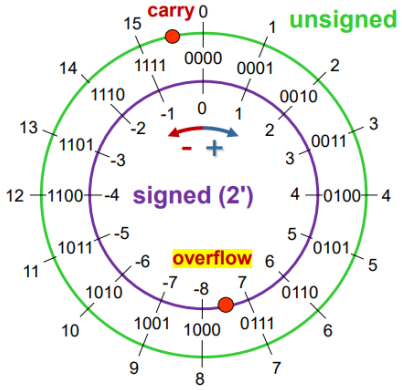
- Addition C = 1 → Carry

```
  1 1 0 1   13d
  0 1 1 1    7d
1 1 1 1
-----------------
1 0 1 0 0   20d → 16d + 4d
```

- Subtraction C = 0 → Borrow

```
6d - 14d = 0110b - 1110b = 0110b + 0010b

  0 1 1 0    6d
  0 0 1 0    2d = TC(14d)
0 1 1 0
-----------------
0 1 0 0 0   8d → - 16d + 8d
```

sign



Pascal Isliker

Branch Instructions

Unconditional Branches

- B (immediate) *B label*
  - Direct
  - Relative
- BX (**B**ranch and E**x**change) *BX R0*
  - Branch and Exchange
  - Indirect
  - Absolute

Conditional Branches

Flag-dependent and arithmetic branches

- Indirect
- Absolute

Overview

Type

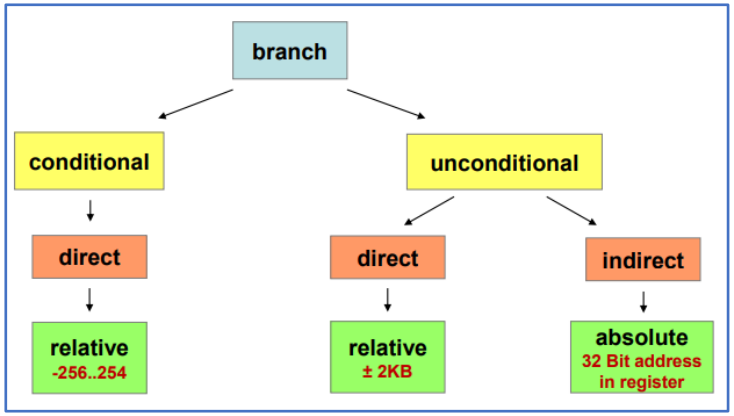
- Unconditional — Branch always
- Conditional — Branch if condition is met

Address hand-over

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

Address of target

- Relative — Target address relative to PC
- Absolute — Absolute address



Flag dependent instruction

Unsigned

- **H**igher and **L**ower

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

- **G**reater and **L**ess

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

Compare and Test

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

Pascal Isliker

Logic and Shift Instructions / Integer Casting

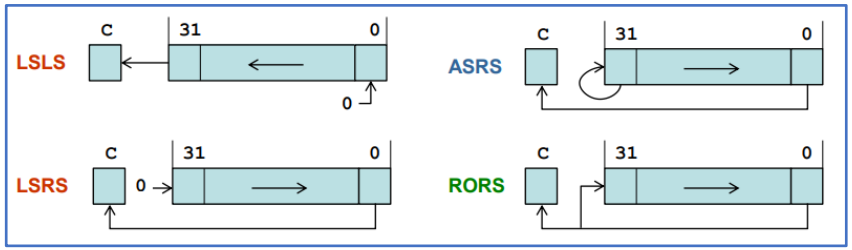
Logical Instructions

The following instruction only affect N and Z flags

• ANDS	Bitwise AND	Rdn & Rm	a & b
• BICS	Bit Clear	Rdn & !Rm	a & ~b
• EORS	XOR	Rdn $ Rm	a ^ b
• MVNS	Bitwise NOT	!Rm	~a
• ORRS	Bitwise OR	Rdn # Rm	a | b

Shift Instructions

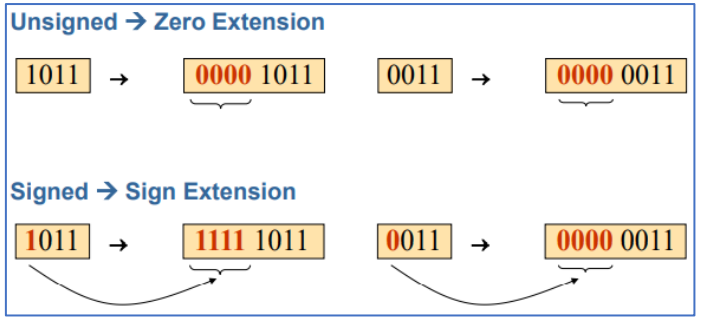
• LSLS	Logical Shift Left	$2^n \cdot Rn \quad 0 \rightarrow LSB$
• LSRS	Logical Shift Right	$2^{-n} \cdot Rn \quad 0 \rightarrow MSB$
• ASRS	Arithmetic Shift Right	$R^{-n} \cdot \pm A \quad MSB \rightarrow MSB$
• RORS	Rotate Right	$LSB \rightarrow MSB$



Sign-Extension

Add additional bits

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



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	unsigned	signed
	0x00	0	0
	...	...	...
	0x7F	127	127
	0x80	128	-128
	...	...	...
	0xFF	255	-1

16-bit	hex	unsigned	signed
	0x0000	0	0
	...	...	...
	0x7FFF	32'767	32'767
	0x8000	32'768	-32'768
	...	...	...
	0xFFFF	65'535	-1

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

Pascal Isliker

Structured Programming – Control Structures

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) {
case 0:
    result += 17;
    break;
case 1:
    result += 13;
    //fall through
case 3: case 5:
    result += 37;
    break;
default:
    result = 0;
}
```

Assume: n in R1, result in R2

```
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
```

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

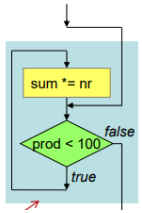
```
        MOVS  R2,#0
loop    ADDS  R2,R2,R1
        CMP   R2,#100
        BLT   loop
        ....
```

- 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 Stack

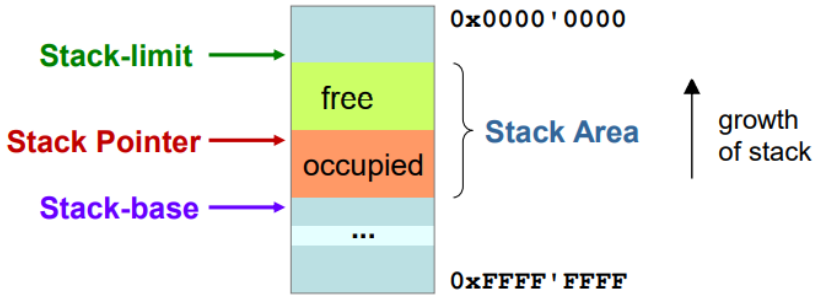
Subroutine Call and Return

- Label with Name (MulBy3)
- Return Statement (*BX LR*)

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

Stack

- Stack Area (Section) — Continuous area of RAM
- Stack Pointer (SP) — R13 → points to last written data value
- PUSH {…} — Decrement SP and store words
- POP {…} — Read words and increment SP
- Direction on ARM — full-descending stack
- Alignment — word-aligned
- Only words — 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]
00000006 9602      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 — Caller and Callee use the same register
- Global variables — Shared variables in data area
- 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 Registers

- By value
  - Efficient and simple
  - Limited number of registers
- By reference
  - Allows passing of larger structures

Register / "pass by value"

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

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

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

Passing through global variables

- Shared variables in data area
- Overhead to access variable
- Error prone, unmaintainable

Global Variables

```
AREA exData,DATA,...
param1  SPACE 1
result  SPACE 1

AREA exCode,CODE,...
  ...
  LDR     R4,=param1    ; caller
  MOVS    R5,#0x03      ; caller
  STRB    R5,[R4]       ; caller
  BL      double_g      ; caller
  LDR     R4,=result    ; caller
  LDRB    ...,[R4]      ; caller
  ...

double_g                ; callee function double_g
  LDR     R4,=param1    ; callee function double_g
  LDRB    R1,[R4]       ; callee function double_g
  LSLS    R0,R1,#1      ; callee function double_g
  LDR     R4,=result    ; callee function double_g
  STRB    R0,[R4]       ; callee function double_g
  BX      LR            ; callee function double_g
```

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 — zero or sign extend to word
- Word — return in R0
- Double word — return in R0 / R1
- 128-Bit — return in R0 – R3

Returning composite data types

- Up to 4 bytes — return in R0
- Larger than 4 bytes — stored in data area

Register Usage

Register	Synonym	Role
r0	a1	Argument / result / scratch register 1
r1	a2	Argument / result / scratch register 2
r2	a3	Argument / scratch register 3
r3	a4	Argument / scratch register 4
r4	v1	Variable register 1
r5	v2	Variable register 2
r6	v3	Variable register 3
r7	v4	Variable register 4
r8	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[1)]
r13	SP	
r14	LR	
r15	PC	

Cortex-M0: Registers r8 – r11 have limited set of instructions. Therefore, they are often not used by compilers.

Register contents might be modified by callee

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

Restore 'caller saved' registers	Get return values from R0 – R3	De-allocate stack space used for parameters
`POP {R0-R3}`	`MOV Rz,R0` `...`	`ADD SP,SP,#(4*args)`

Pascal Isliker

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

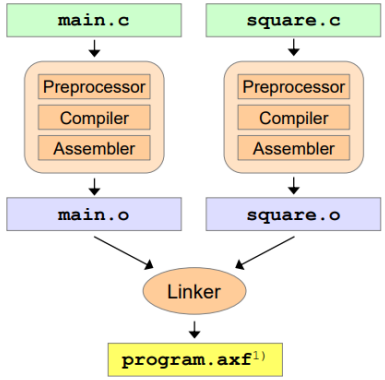
- Results in one executable file

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 developers working on the same source repository
Useful partitioning and structuring of the programs	Eases reuseing of modules
Individual verification of each module	Benefits all users of the module
Providing libraries of types and functions	For reuse instead of reinvention
Mixing of modules that are programmed in various languages	E.g. mix C and assembly language modules
Only compile the changed modules	Speeds up compilation time

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
```


Linker Input - Object files

Code section	Code and constant data of the module, base at address 0x0
Data section	All global variables of the module, based at address 0x0
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

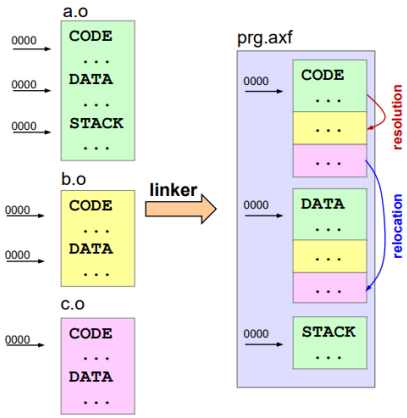
ARM tool chain uses ELF for object files

Linker tasks

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

Linker Output

- AXF = **A**RM e**X**ecutable **F**ile



Pascal Isliker

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	Restart of processor
• NMI	Non-maskable Interrupt (cannot be ignored)
• Faults	Undefined instructions
• System Level Calls	OS calls – Instructions SVC and PendSV

PRIMASK
- Single bit controlling all maskable interrupts

PRIMASK

		Assembly	C
- Disable	set PRIMASK	CPSID[1] i	__disable_irq();
- Enable	clear PRIMASK	CPSIE[1] i	__enable_irq();

On reset PRIMASK = 0 → enabled

Interrupt-Driven I/O

Main program

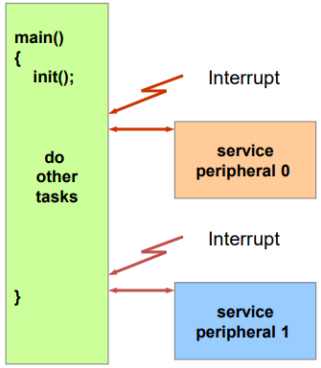
- Initializes peripherals
- Afterwards it executes other tasks
- Peripherals signal when they require SW attention
- Events interrupt program execution

Advantage

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

Disadvantages

- No synchronization
- Difficult debugging



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

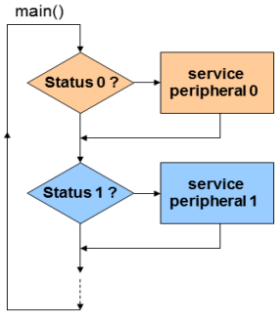
ISR Return

- Use BX LR or POP {…, PC}
- Loading EXC Return into PC
  - Restores R0-R3, R12, LR, PC and xPSR from Stack

Polling

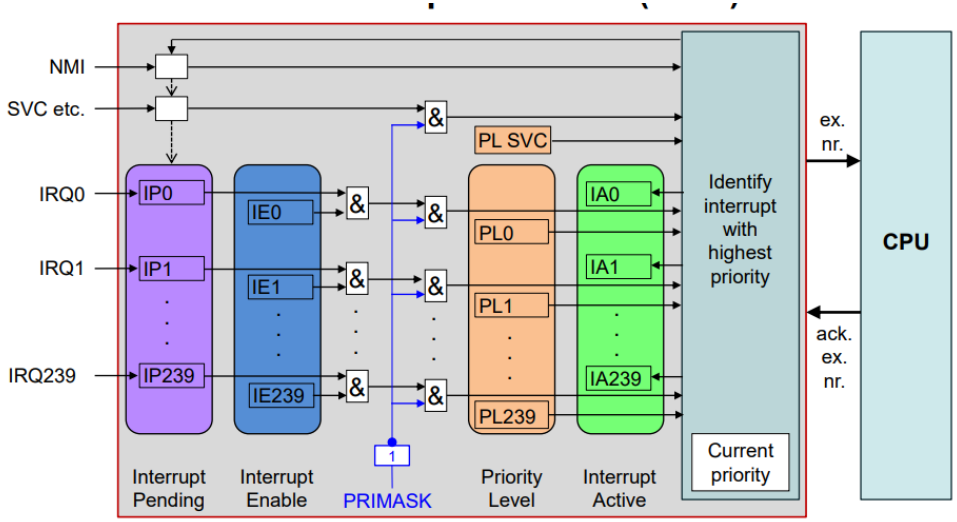
Periodic query of status information

- 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



Pascal Isliker

Interrupt Control

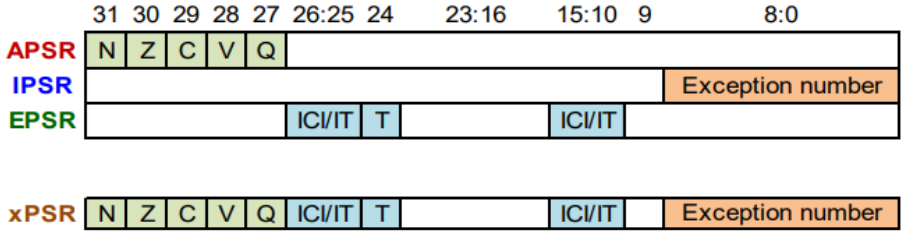


Expection States

- Inactive
  - Not active and not pending
- Pending
  - Exception is waiting tob e serviced b CPU
  - An interrupt event occurred (IRQn=1) but interrupts are disabled (PRIMASK)
- Active
  - Exception is being serviced by the CPU but has not completed
- Active and pending
  - Exception is being serviced by the CPU and there is a pending exception fort he same source

Program Status Registers PSRs

- IPSR Interrupt Program Status Register
- EPSR Execution Program Status Register
- APSR< Application Program Status Register
- xPSR Combination fo all three PSRs



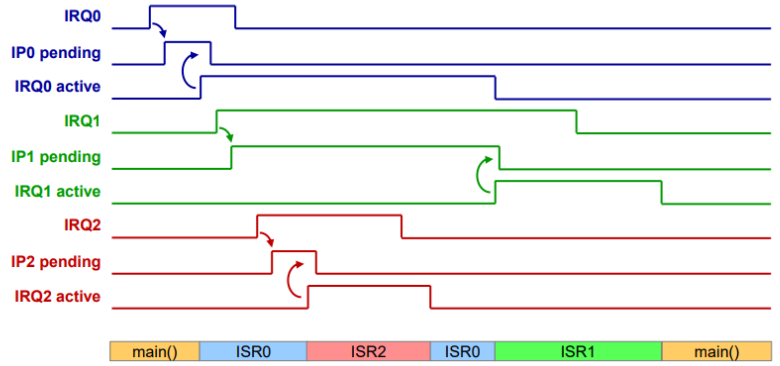
Nested Exceptions

Example Priorities

- ISR1 does not preempt ISR0
- ISR2 preempts ISR0

assuming

IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



Improving System Performance

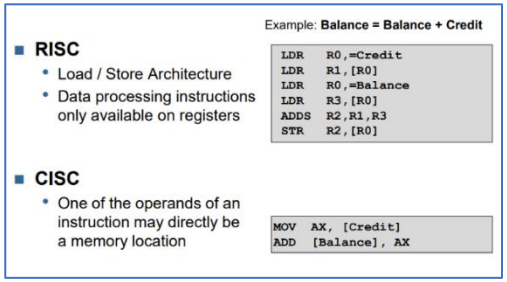
Speed vs Low Power

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 = **R**educed **I**nstruction **S**et **C**omputer

- 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)

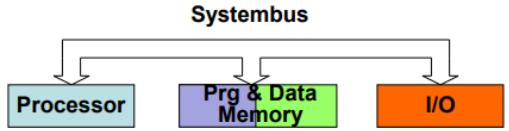


CISC = **C**omplex **I**nstruction **S**et **C**omputer

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

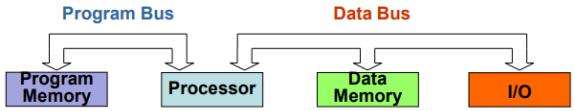
Von Neuman Arhcitecture

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

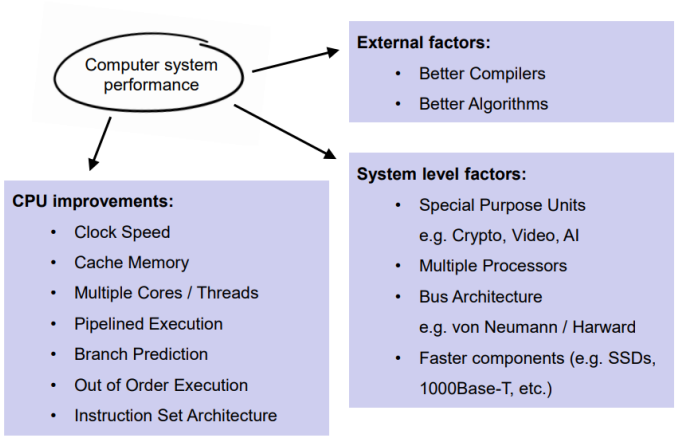


Harvard Architecture

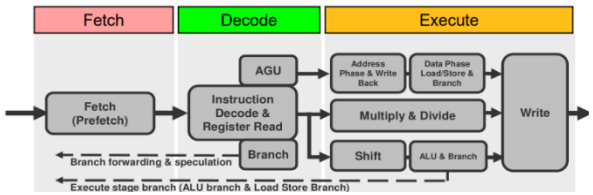
- «Mark I» at Harvard University
- Separate memories for program and data
- Two sets of addresses/data buses between CPU and memory



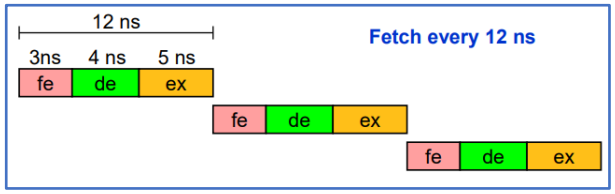
How to Increase System Speed?

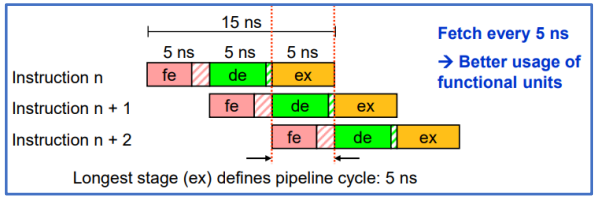


Fetching the next instruction, while the current one decodes



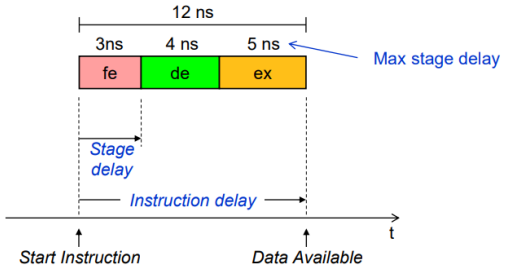
Sequential vs. Pipelined





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



Instructions per second

Without pipelining

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

With pipelining

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

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

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

Optimal pipelining

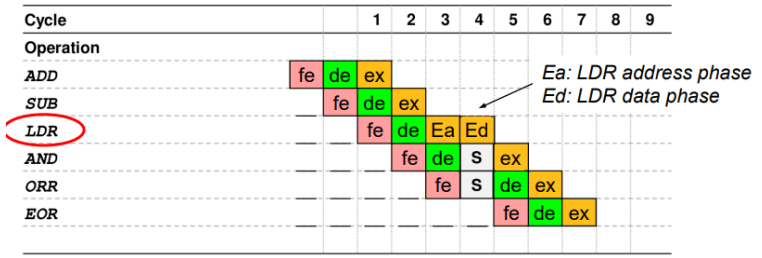
- All operations here are on registers
- In this example it takes 6 clock cycles to execute 6 instructions
- Clock cycles per instruction (CPI) = 1

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	

Pascal Isliker

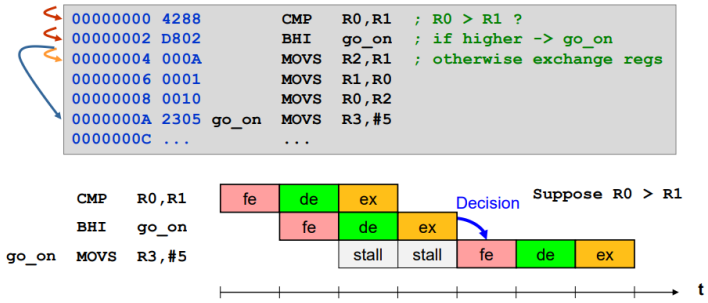
Special situation: LDR

- 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:



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