UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

1918 · 2018

forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 2

Review – Background & Number Representation
Hersiening – Agtergrond & Getalle Voorstelling

Dr Rensu Theart & Dr Lourens Visagie

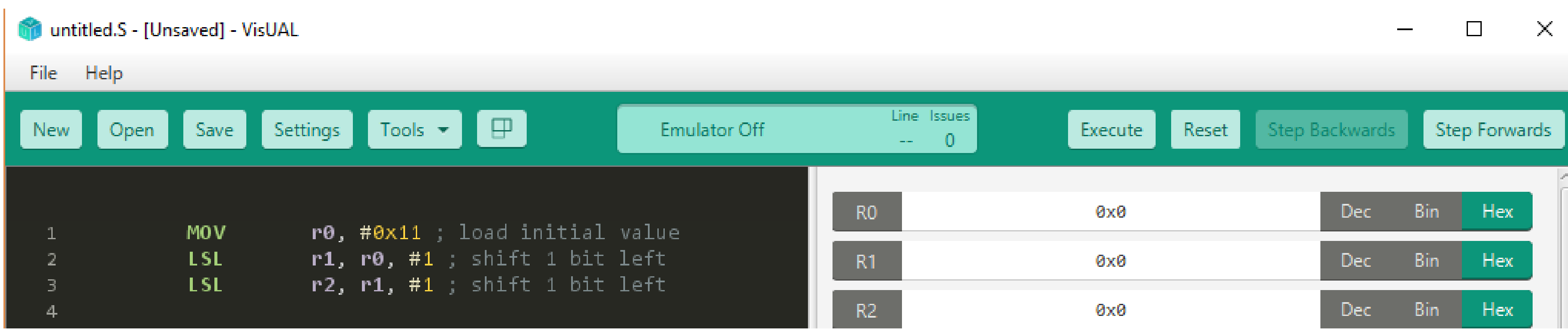
Lecture Overview

- The software we will use in this module
- Executing instructions on a CPU
- Number representation / Getalle voorstelling
  - Number systems (binary, hexadecimal)
  - Integer & Signed integer (2's complement)
  - Floating point
  - Boolean type
- Data types
- Characters and strings

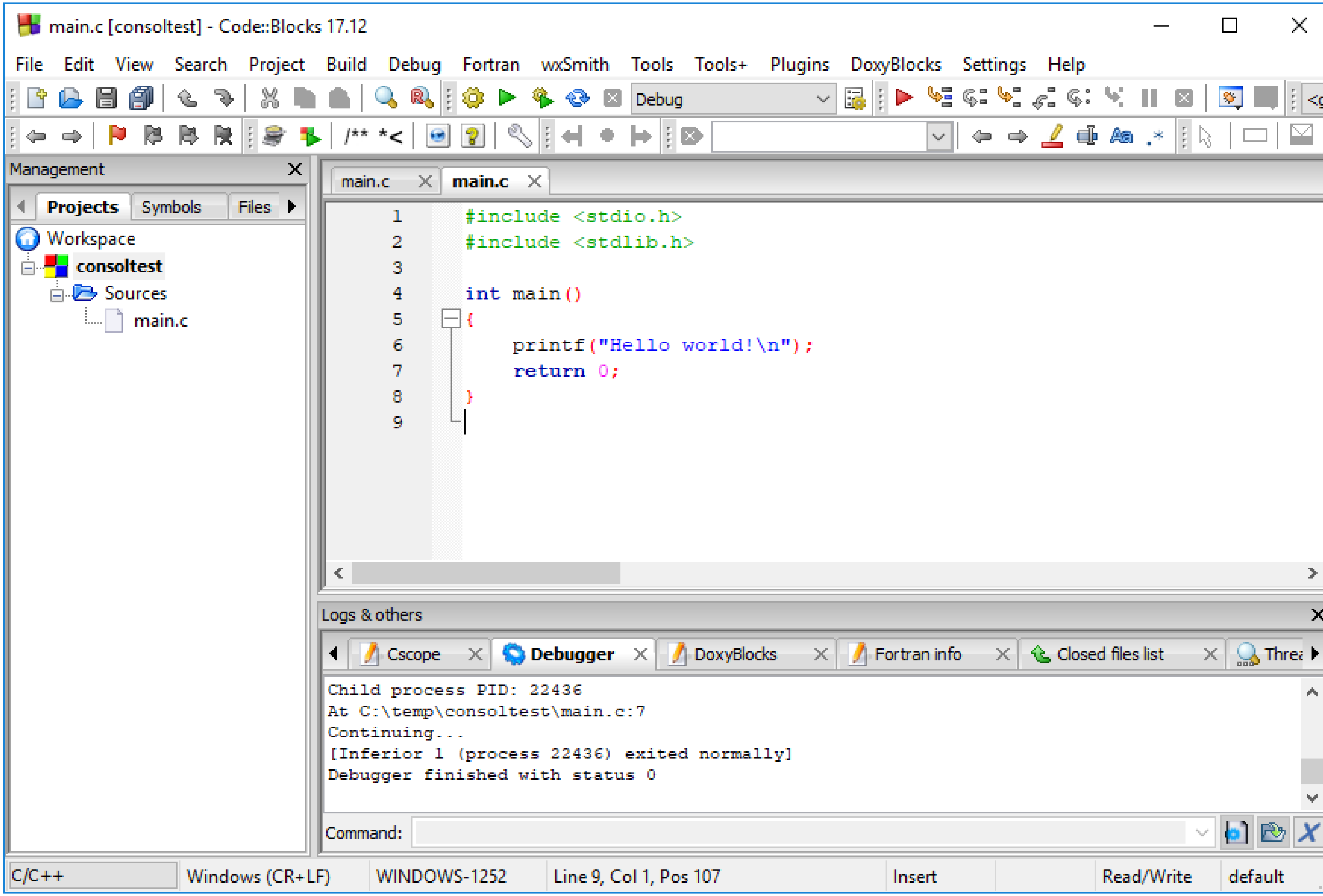
VisUAL

ARM emulator

- Developed by Imperial College London (https://salmanarif.bitbucket.io/visual/)
- Supports UAL instructions (not all op-codes supported – see website for documentation)
- Press CTRL+Space in a line for help on that instruction

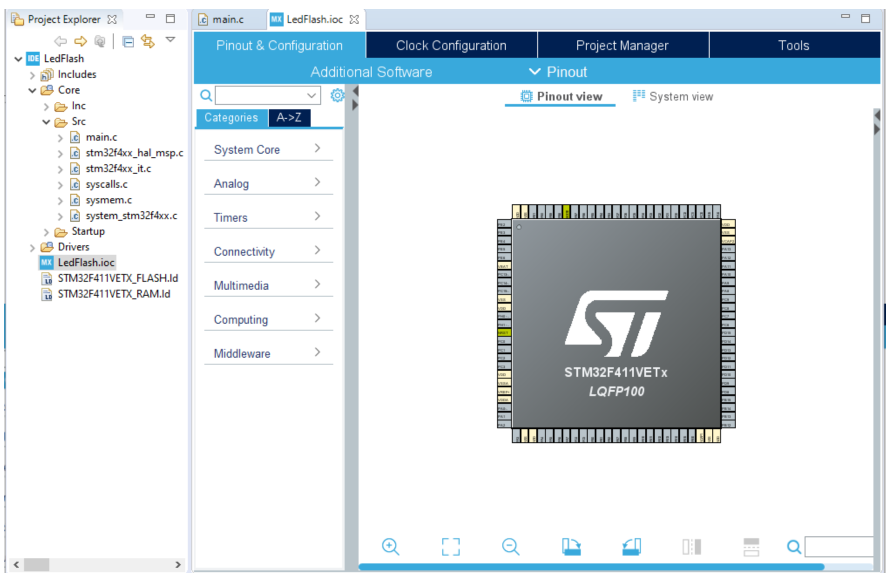


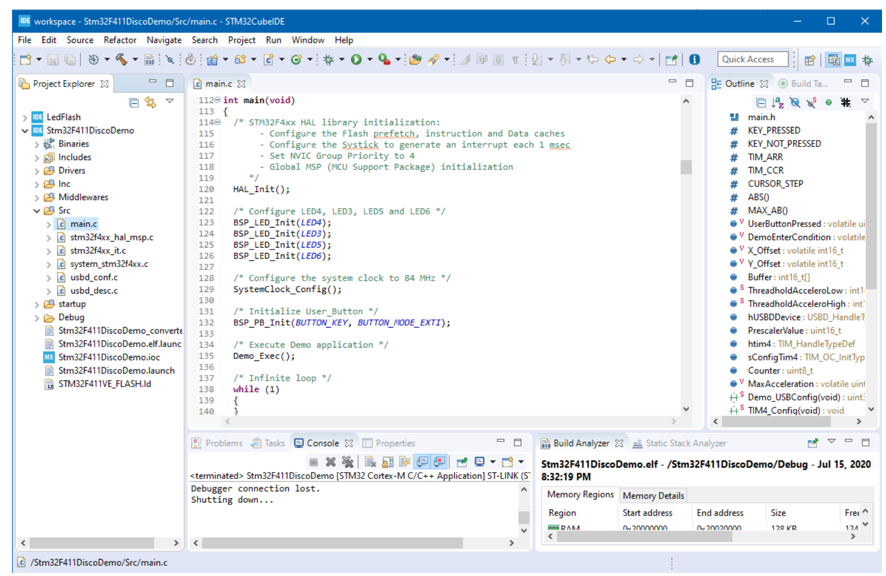
Code::Blocks C IDE



STM32CubeIDE

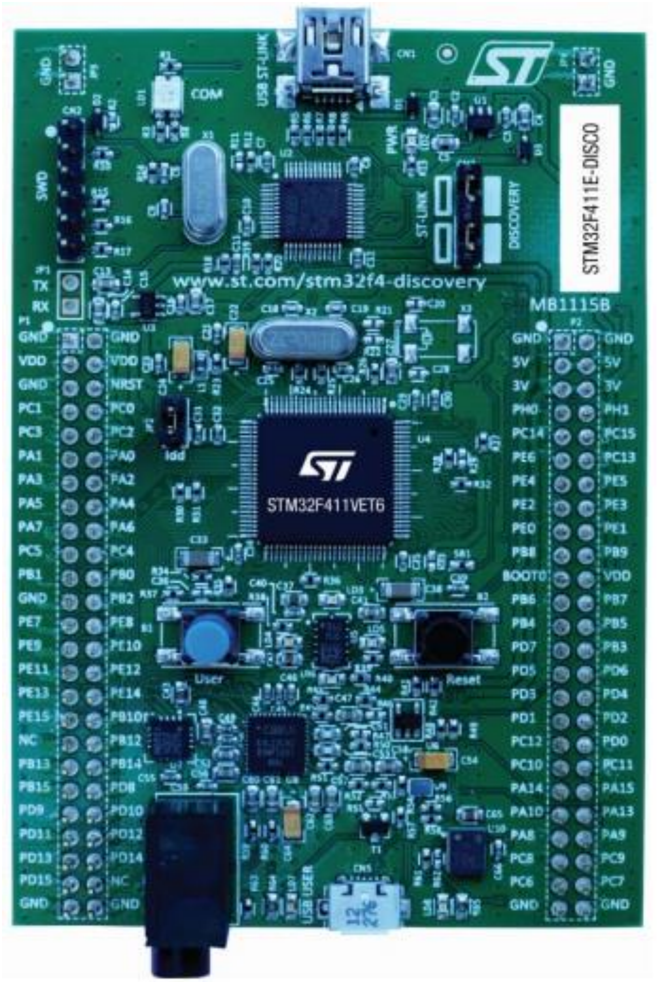
- STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors.
- It is based on the ECLIPSE™ framework and GCC toolchain for the development, and GDB for the debugging.





Development board

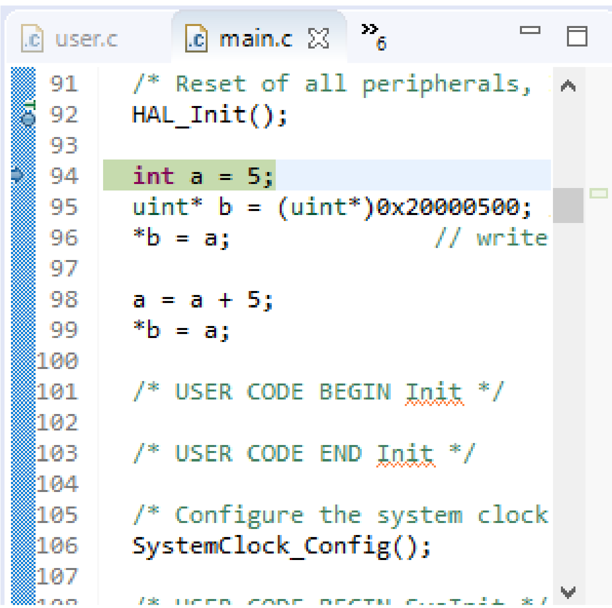
- Development board for the **STM32F411VET6** microcontroller (STM32F411 Discovery Kit)
- Board and microcontroller manufactured by ST Microelectronics
- In the microcontroller:
  - **ARM Cortex-M4** with floating point unit
  - 512 kb flash memory (program memory)
  - 128 kb SRAM
- Other stuff on the board:
  - 3D MEMS gyroscope, linear accelerometer, magnetic field sensor
  - MEMS microphone, audio digital-to-analog converter (DAC) and amplifier
  - LEDs, pushbuttons



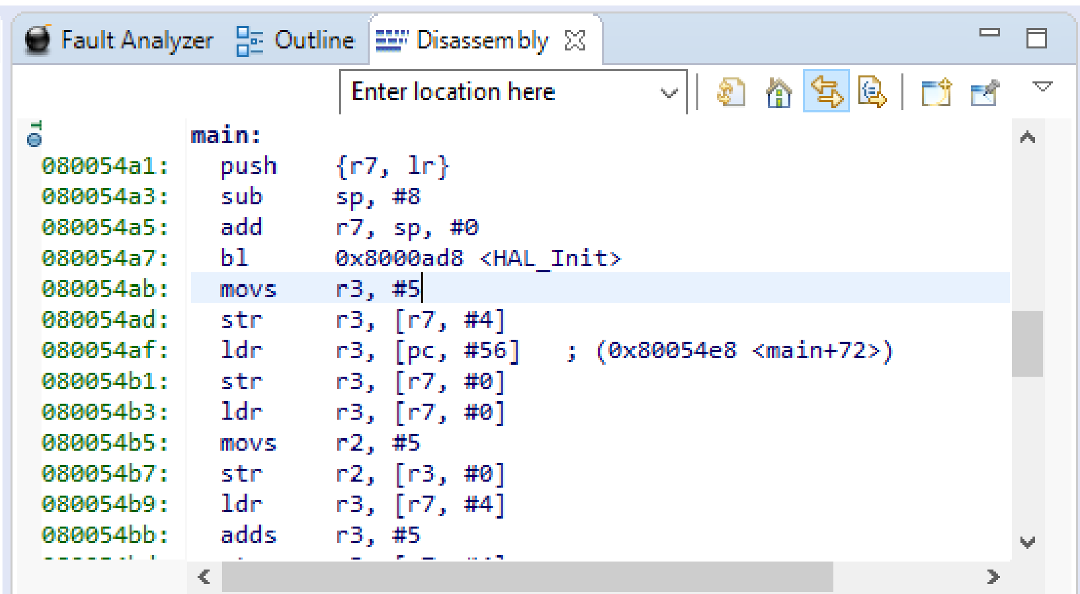
- Can be ordered from [HERE]() or [HERE]() or [HERE]()

C vs Assembly

- Higher level programs (C, Java etc) will eventually translate into many simple machine instructions



C-code



Equivalent assembly
(generated backwards from
compiled machine instructions)

- What does the machine code look like?

Architecture / Argitektuur

- The **architecture** is the programmer's view of a computer.
- It is defined by the **instruction set** (language) and **operand locations** (registers and memory).
- Many different architectures exist, such as ARM, x86, MIPS, SPARC…
- We will specifically look at the ARM instruction set, although the concepts of assembly programming apply generally between the different instruction sets.
- ARM is a family of CPUs based on the **RISC** (reduced instruction set computer) architecture.
- ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

Instructions and operands / Instruksies en operande

- The first step in understanding any computer architecture is to learn its language.
- The words in a computer's language are called **instructions**.
- The computer's vocabulary is called the **instruction set**.
- All programs running on a computer use the same basic instruction set, which are instructions such as *add*, *subtract*, and *branch*.
- Computer instructions indicate both the **operation** to perform and the **operands** to use.
- The operands may come from memory, from registers, or from the instruction itself.

$$\underbrace{\texttt{ADD}}_{\text{Operation}}\ \underbrace{\texttt{R0, R1, \#5}}_{\text{Operands}}$$

Machine language and assembly language

- Computer hardware only understand 1's and 0's, so instructions are encoded as binary numbers in a format called **machine language**.
- The ARM architecture represent each instruction as a **32-bit word**.
- Microprocessors are digital systems that read and execute machine language instructions.
  - These instructions are directly implemented in logic circuits.
- Machine language is hard to read for humans, so we prefer to represent the instructions in a symbolic format called **assembly language**.

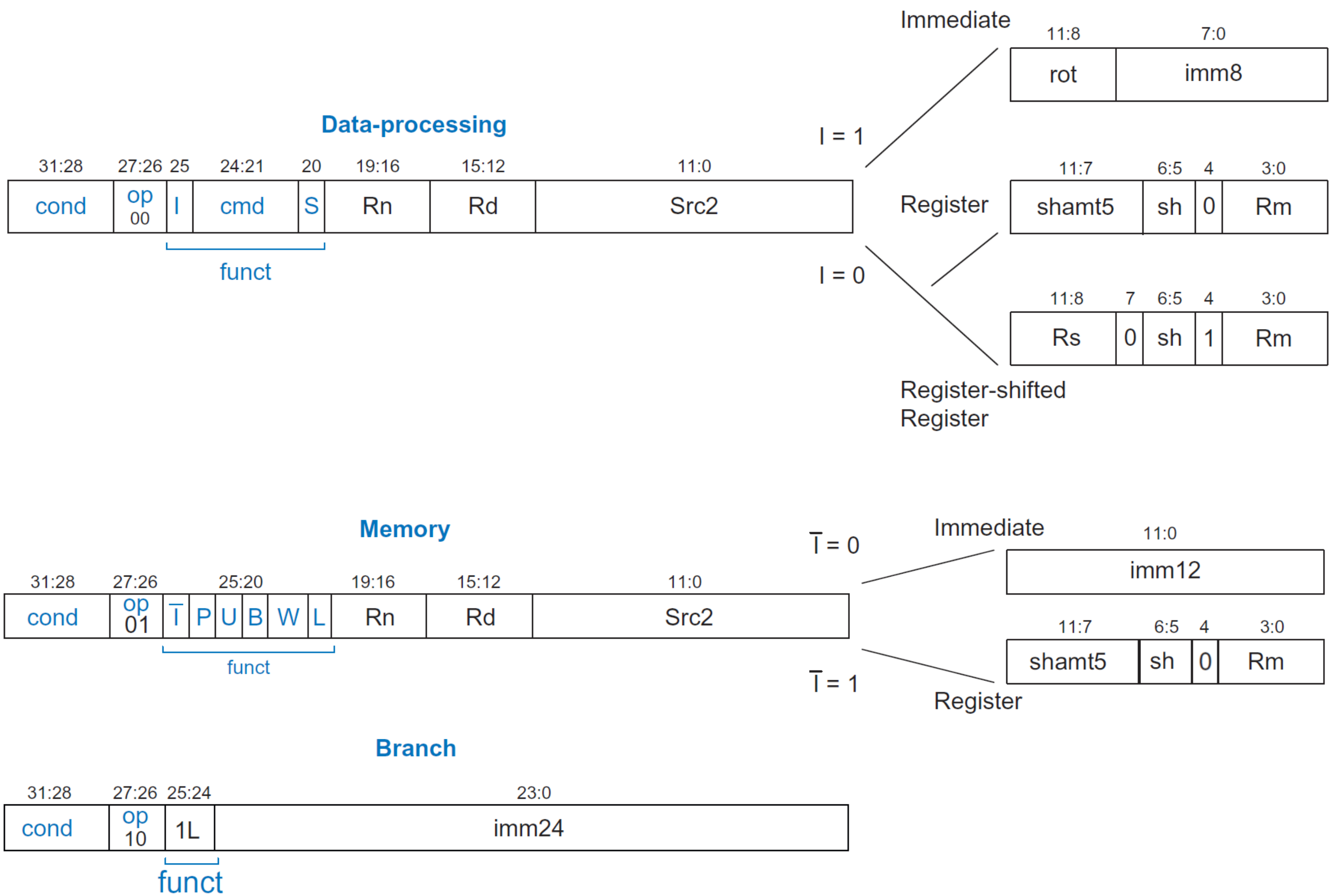
For example:

```
MOV R1, R2 ;move data from one place to another
```

This same instruction translates into machine language as:

$$(11100001101000000001000000000010)_2 = (E1A01002)_{16}$$

Machine code / Masjien kode



Machine language and Integrated Circuits

C program

```c
int myadd(int a, int b)
{
    return a+b;
}
```

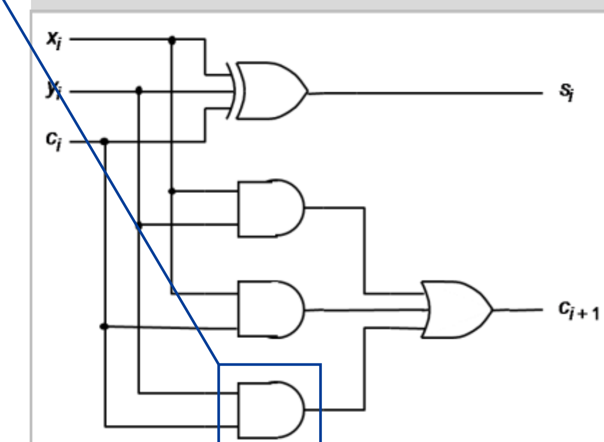
Assembly code

```
myadd:
    .fnstart
    add     r0, r0, r1
    bx      lr
    .fnend
```

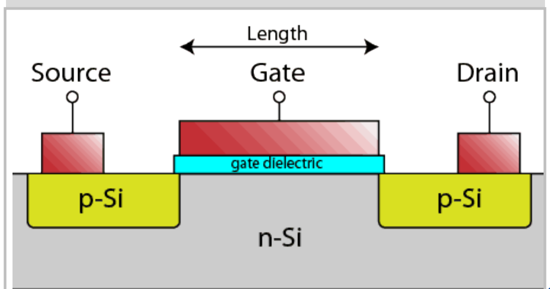
Machine instruction

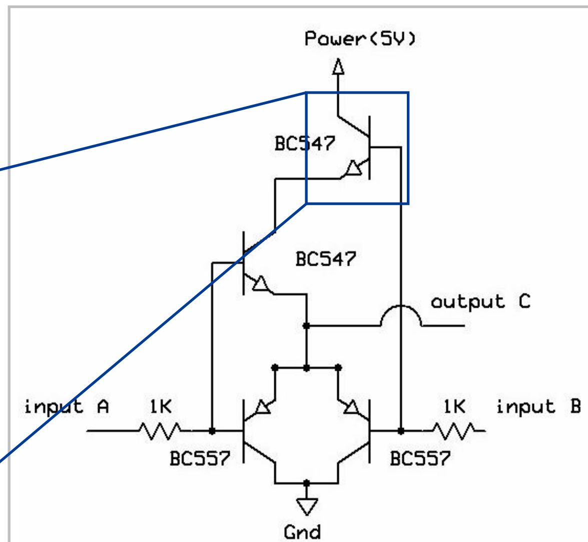
31–28	27–26	25	24–21	20	19–16	15–12	11–0
cond	0 0	I	opcode	S	Rn	Rd	shifter_operand

Full Adder circuit



Transistor

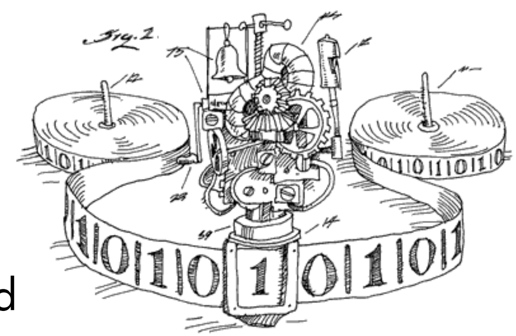


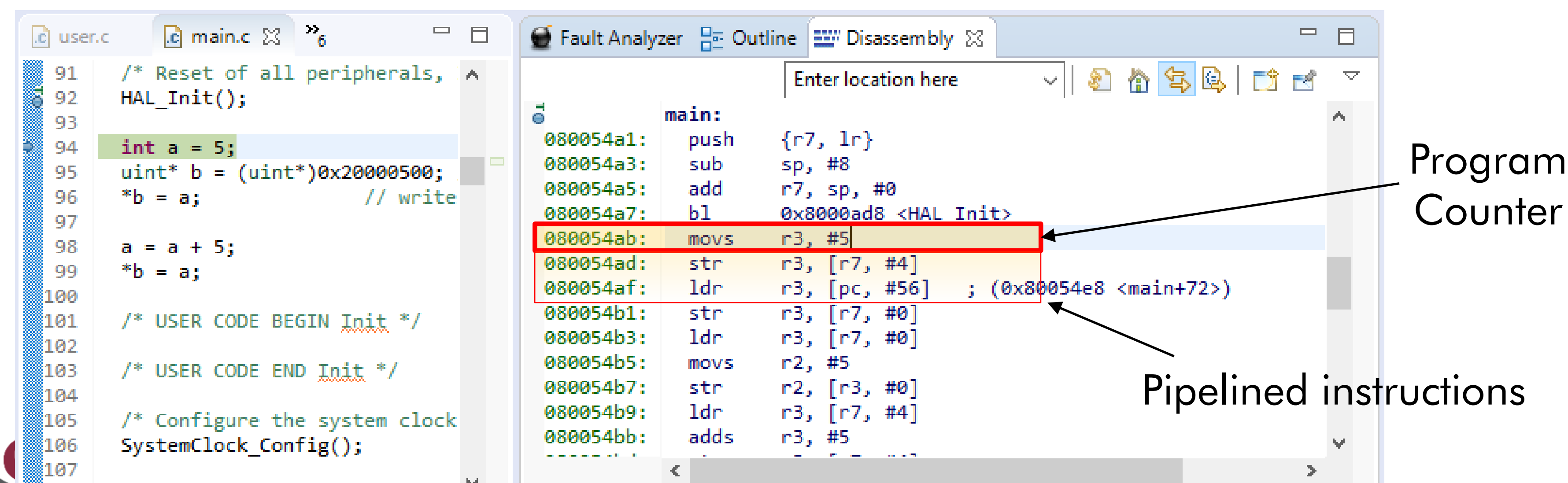


AND gate

Instruction loading

- We have to give the CPU the "sequence-of-CPU-machine-instructions" to make it work.
  - This is called the **binary program**. Or compiled program. (sometimes the words "firmware" or "image" are also used).
- Think of the program instructions as a sequence on a tape.
- Think of the CPU as a sequential processing machine with a single-instruction view.
- The "thing" inside the CPU that holds the position of the current "viewed" instruction is the **Program Counter (PC)**, which is stored in a register.
- High-performance CPUs will have a processing **pipeline** – start pre-processing following instructions.





Instruction loading - Memory

- The CPU will execute (run) a binary program after the program has been placed into memory
  - Remember CPU does not have its own memory.
  - Microcontrollers (=CPU + other stuff in the same integrated device) might have integrated memory, but it is still outside of the CPU.
  - Lots of types (RAM, ROM, Flash etc.). More on this later.
- For now: memory = a matrix of storage elements that allows data to be written and read at specific addresses, and each address refers to a **byte** (8 bits) of data.
- Even though data is byte addressed, CPUs can sometimes read/write more than 8-bits at a time – often referred to as a **word.**

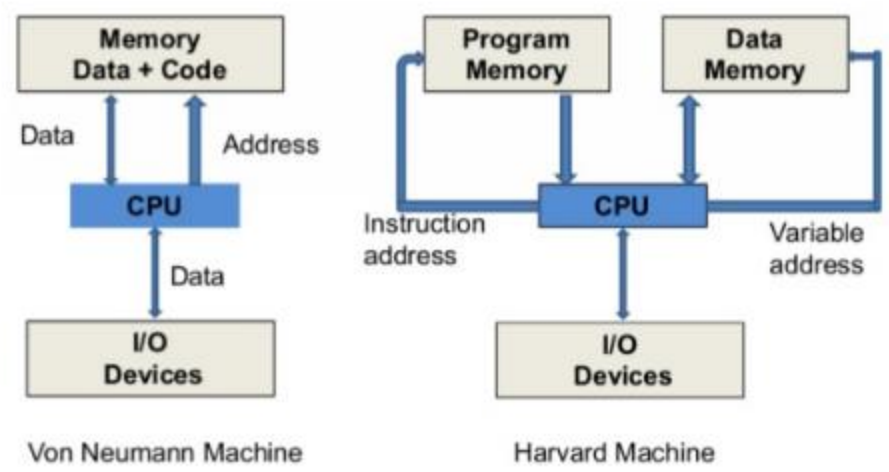
Byte Address	Data
0	0xAA
1	0xBB
2	0xCC
3	0xDD
4	0xEE
5	0xFF
6	0x11
7	0x22
…	…

Word Address	32-bit data
0	0xDDCCBBAA
4	0x2211FFEE
…	…

Little-Endian

Memory / Geheue

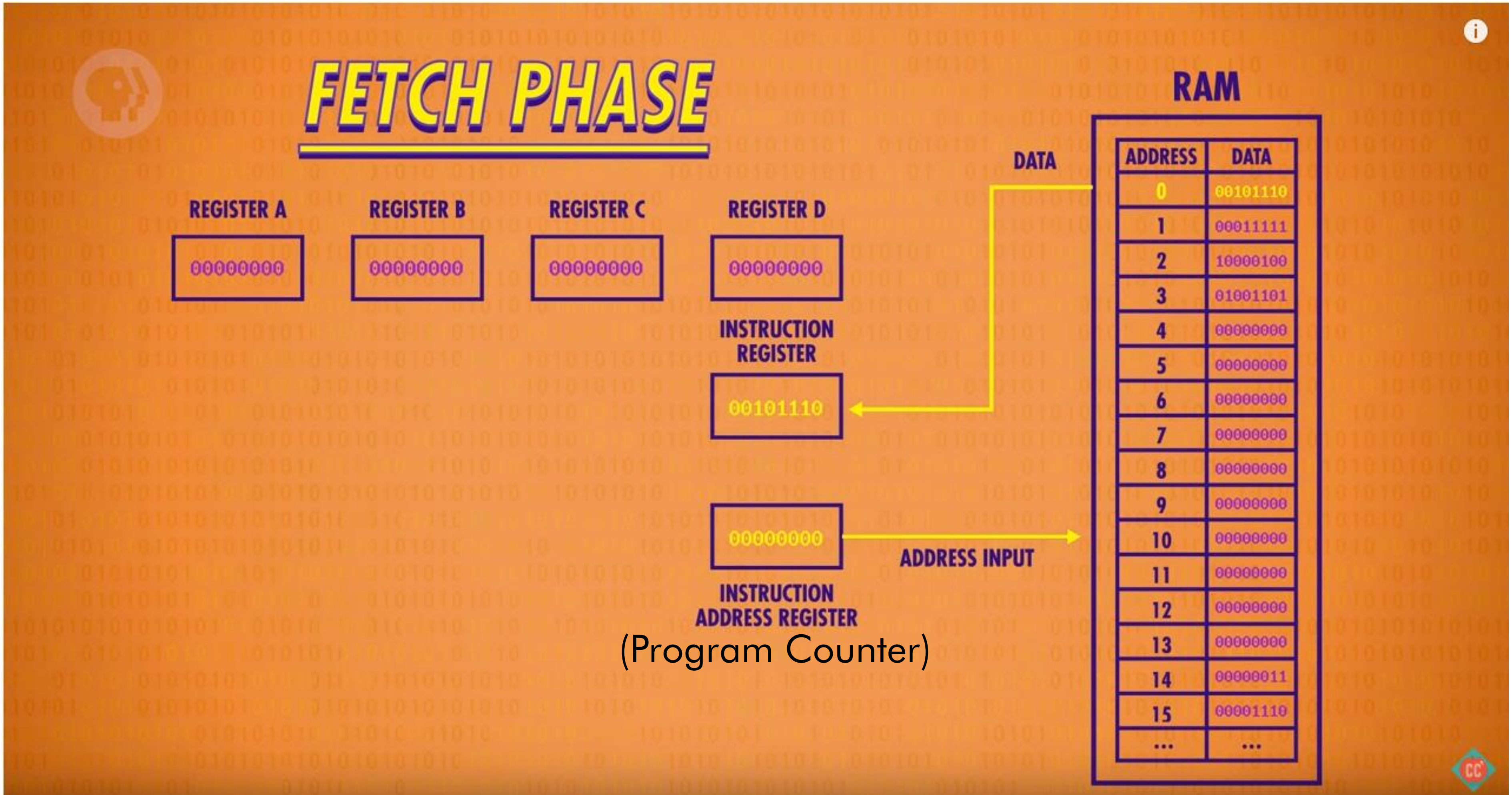
- **Program memory** is normally only read not written to.
- **Data memory,** memory that the program alters as a result of instructions being executed, is used for variables.
- Program and data memory can be physically different memory chips (Harvard Machine) or the same (Von Neumann Machine).



https://vivadifferences.com/5-major-difference-between-von-neumann-and-harvard-architecture/

Fetching instructions

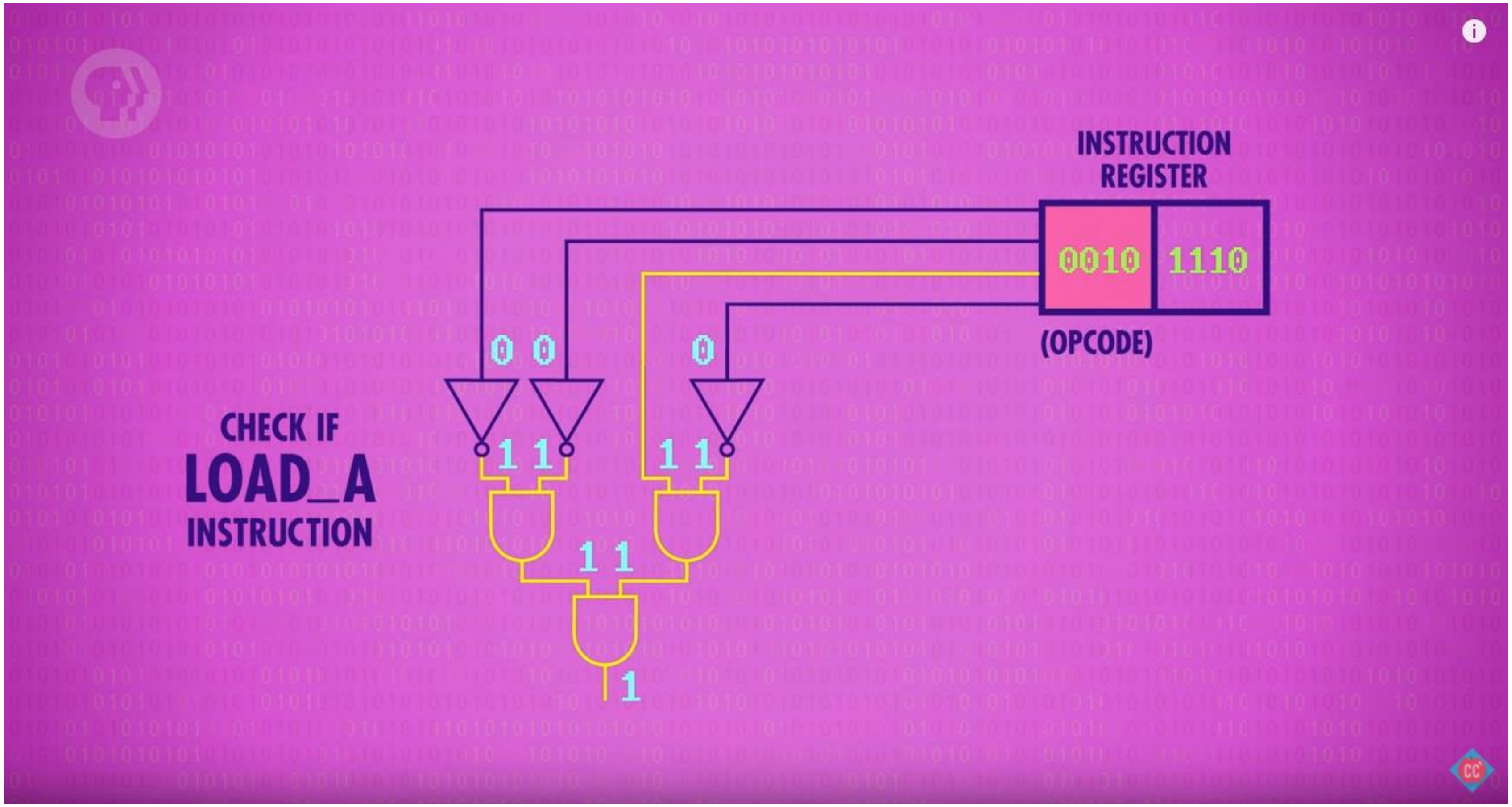
Instructions are *fetched* from memory at the current program counter



(Program Counter)

Decoding instructions

- Logic circuits in the CPU will *Decode* the instruction (check if the instruction bit pattern matches)



Executing instructions

- The relevant logic gates will then *Execute* the instruction.
- Depending on the instruction, this might have various effects:
  - Some instructions will alter the memory contents (write to memory at specific addresses)
  - Some instructions may change the Program Counter (will cause the CPU to fetch instructions from somewhere else, i.e. a function)
  - (Normally the PC is simply incremented so that instructions are executed in the order they are stored in memory)

Simplified pipelined processor

Note that the ARM Cortex-M4 **only** has the Fetch, Decode and Execute states

From CS214

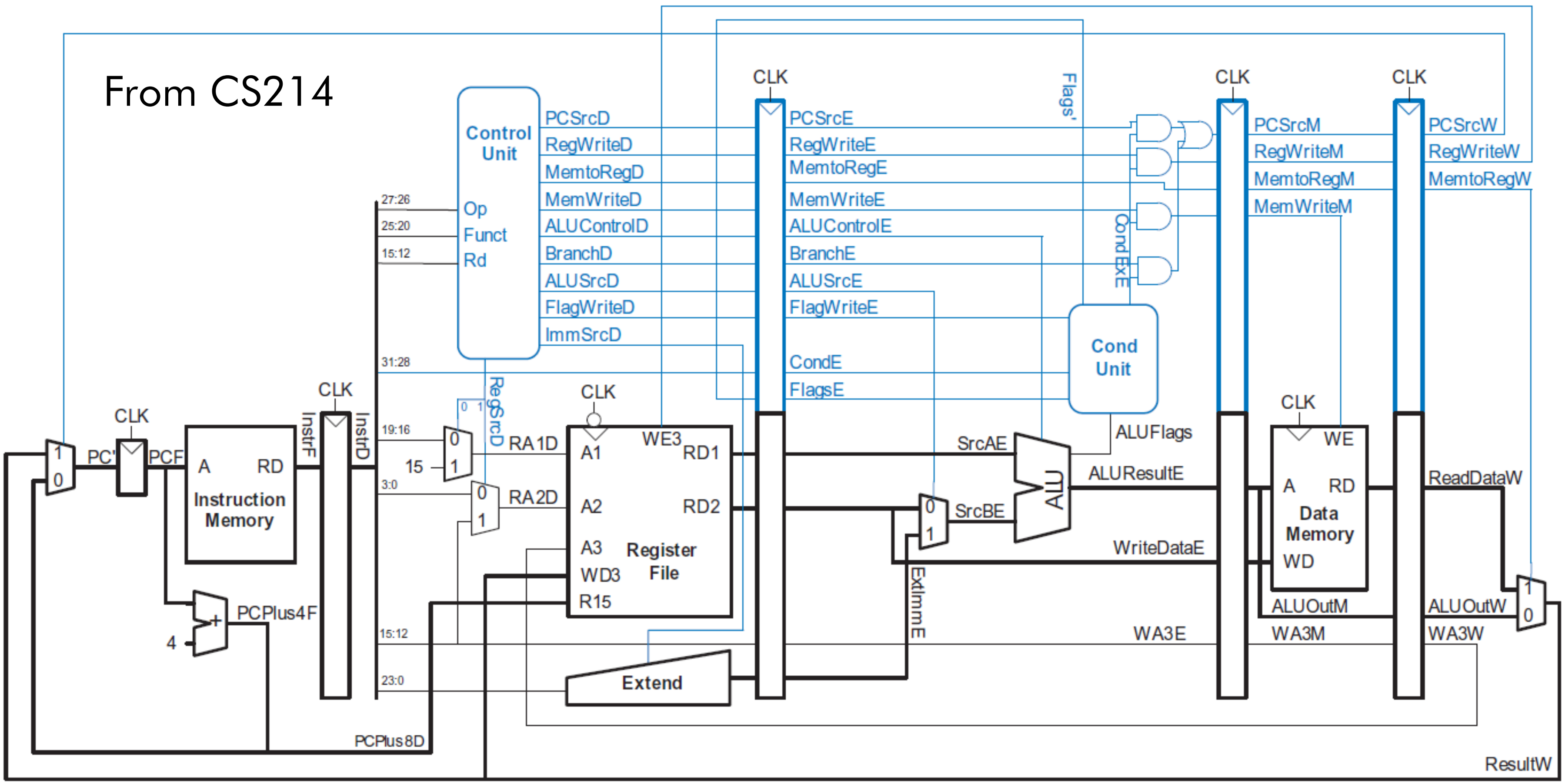


Figure 7.47 Pipelined processor with control

Fetch Decode Execute Memory Write Back

Processor Cycles per instruction

Even though ARM is a RISC architecture, not all instructions execute in a single processor cycle (since no Memory and Writeback pipeline stages).

Cortex-M series processors

Table 3-1 Processor instruction set summary

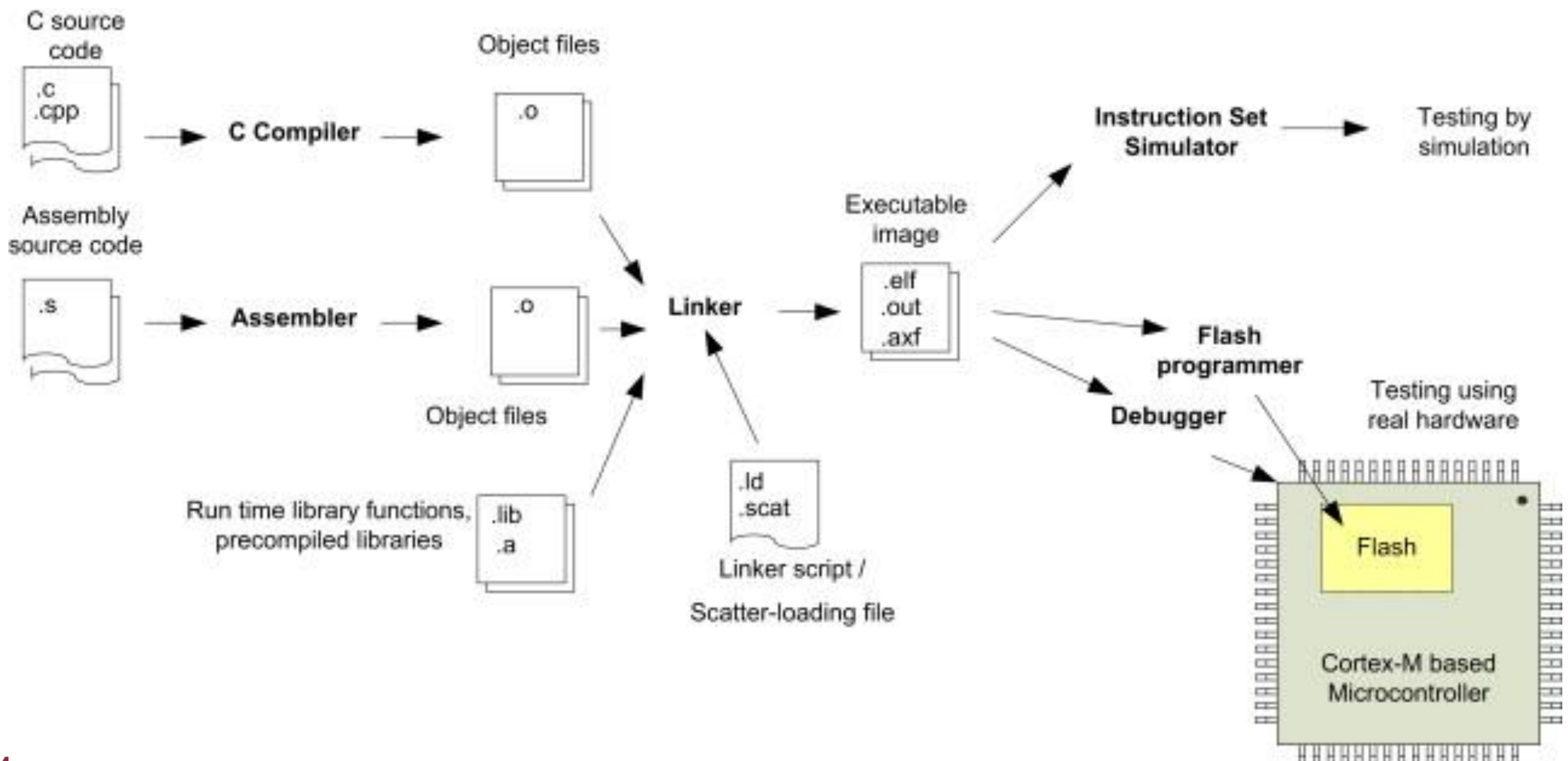
Operation	Description	Assembler	Cycles	Notes
Move	Register	MOV Rd, <op2>	1	
	16-bit immediate	MOVW Rd, #<imm>	1	
	Immediate into top	MOVT Rd, #<imm>	1	
	To PC	MOV PC, Rm	1 + P	
Add	Add	ADD Rd, Rn, <op2>	1	
	Add to PC	ADD PC, PC, Rm	1 + P	
	Add with carry	ADC Rd, Rn, <op2>	1	
	Form address	ADR Rd, <label>	1	
Load	Word	LDR Rd, [Rn, <op2>]	2	
	To PC	LDR PC, [Rn, <op2>]	2 + P	
	Halfword	LDRH Rd, [Rn, <op2>]	2	
	Byte	LDRB Rd, [Rn, <op2>]	2	
	Signed halfword	LDRSH Rd, [Rn, <op2>]	2	
	Signed byte	LDRSB Rd, [Rn, <op2>]	2	

Instructions that alter the PC results in performance hit because the processing pipeline must be “flushed”

ARM vs C – Simple program instruction

Program instructions for the microcontroller (**machine instructions**) is also just a pattern of bits, with specific interpretation

Masjien instruksies is ook net 'n patroon van bisse met spesifieke interpretasie

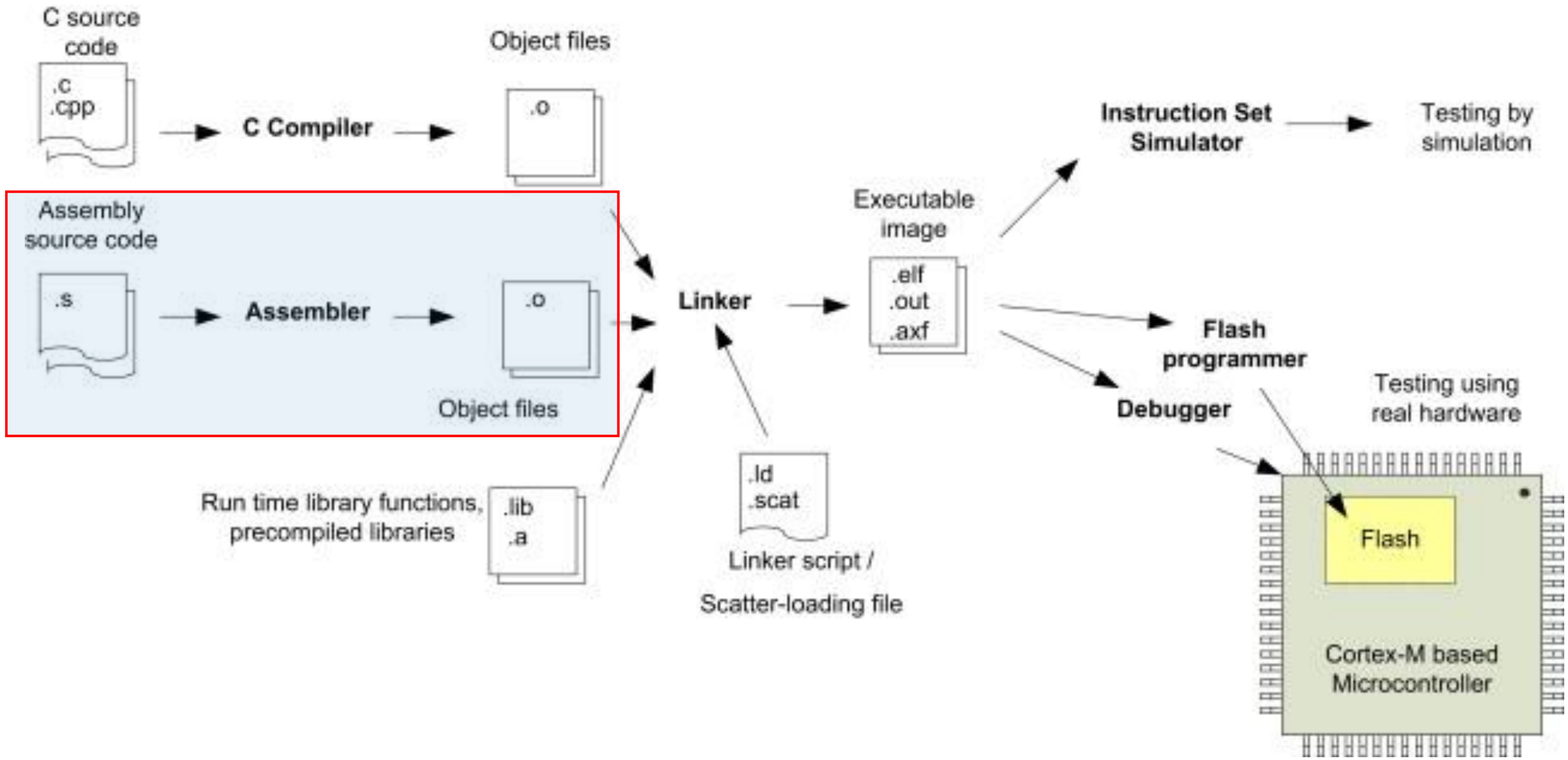


https://www.sciencedirect.com/topics/engineering/memory-layout

ARM vs C – Simple program instruction

The **Assembler** translates assembly programs into machine code

Die **saamsteller** verander saamsteltaal programme na masjien kode

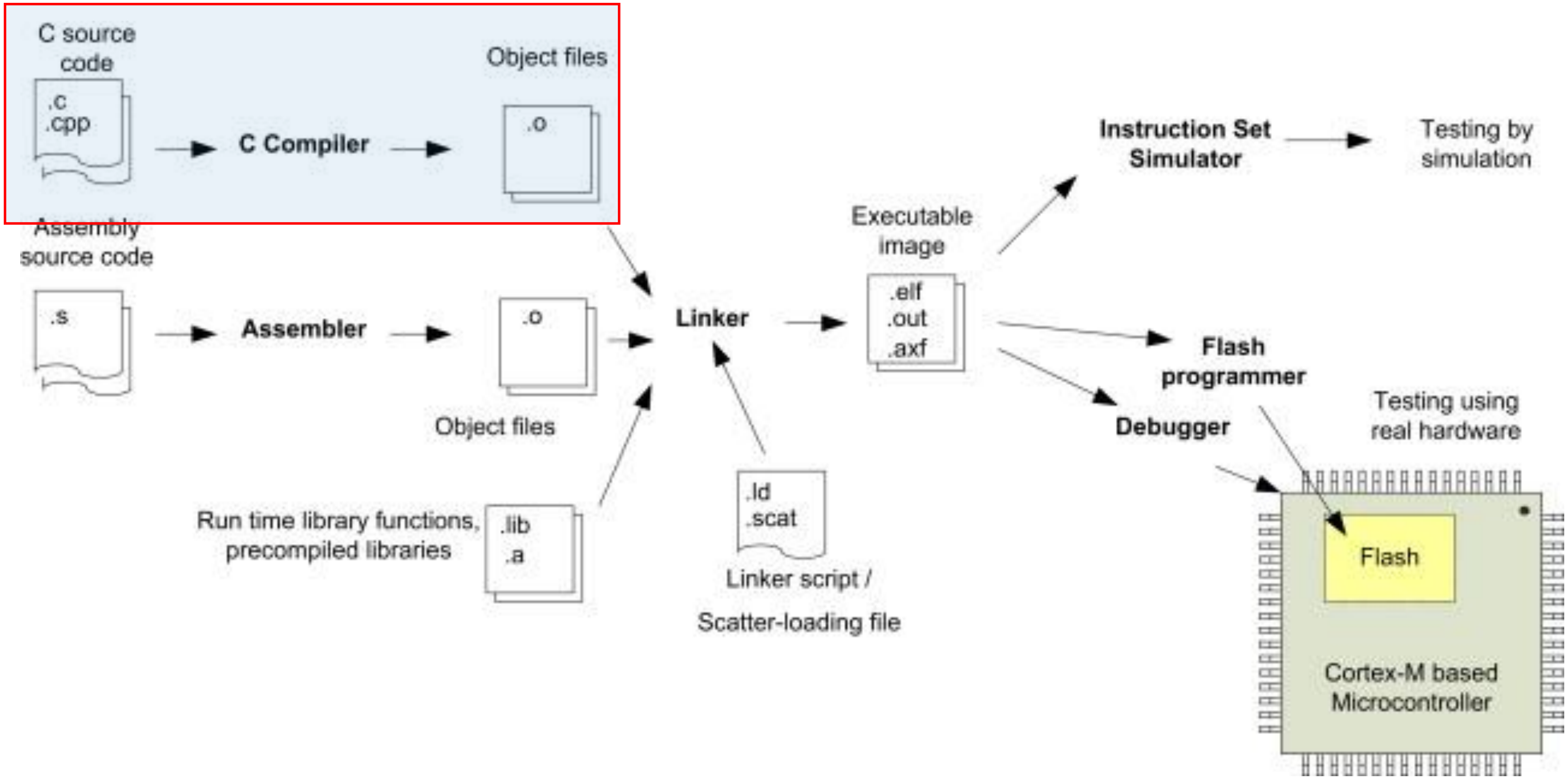


https://www.sciencedirect.com/topics/engineering/memory-layout

ARM vs C – Simple program instruction

The **Compiler** translates C or C++ code into machine code

Die **kompileerder**(?) verander C or C++ kode na masjien kode

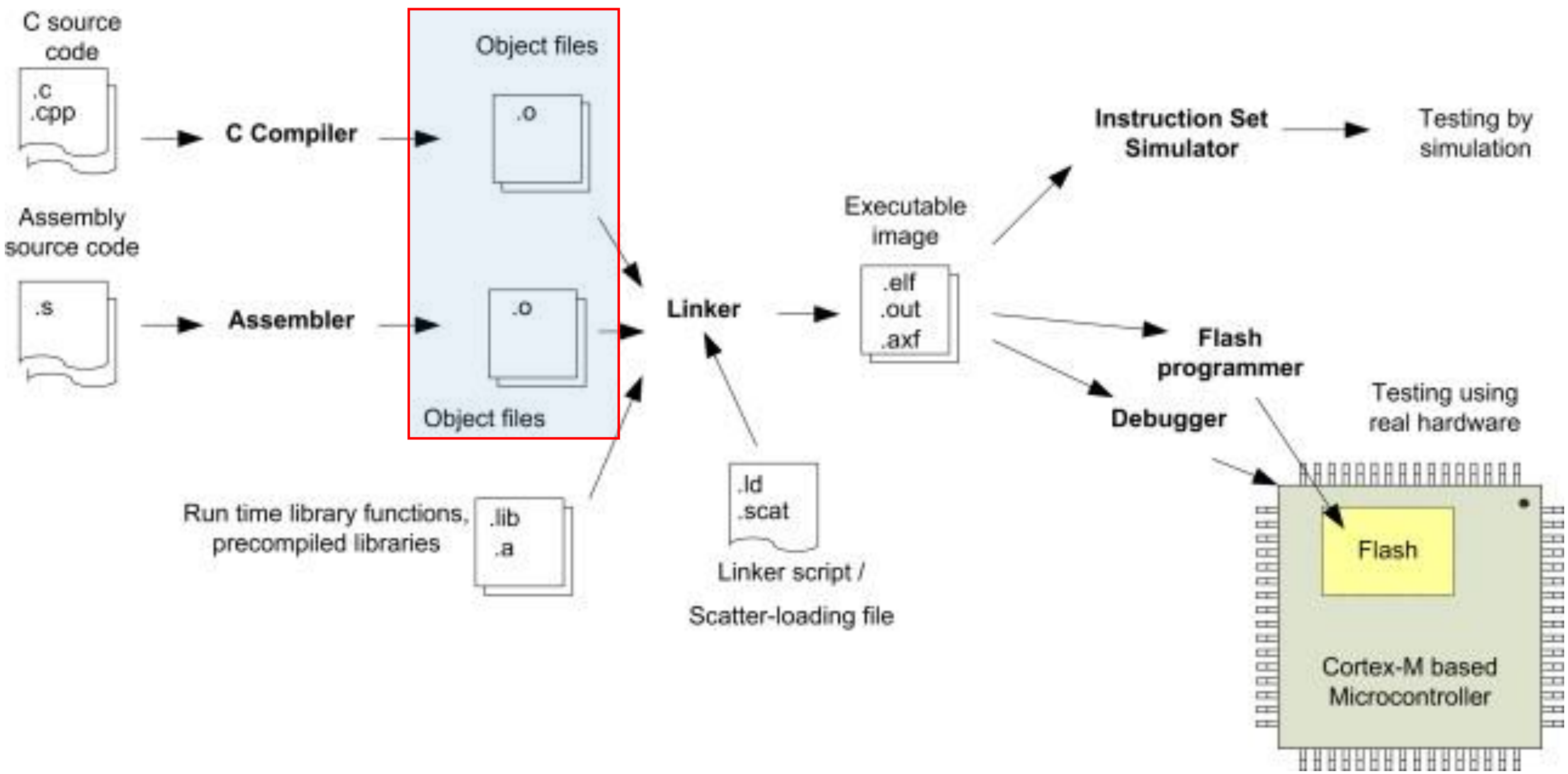


https://www.sciencedirect.com/topics/engineering/memory-layout

ARM vs C – Simple program instruction

- **Object files** contain machine code, debug information and symbols, relocation and linker information

  Objek lêers bevat masjien kode, simbole en fout-opsporing-, herposisionerings en koppel-inligting.

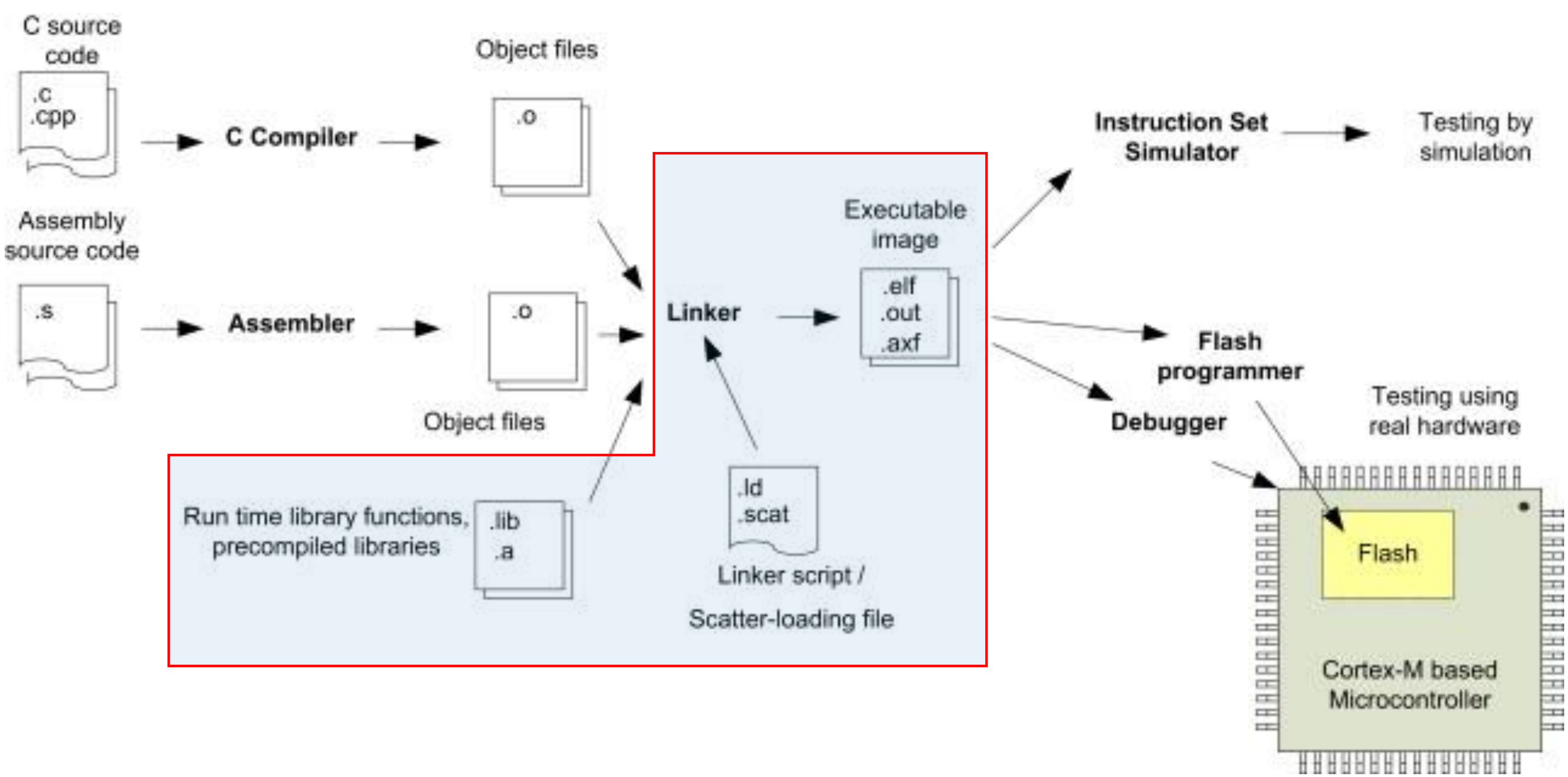


https://www.sciencedirect.com/topics/engineering/memory-layout

ARM vs. C – Simple program instruction

The **Linker** combines object files with library files into an executable file

Die **koppelaar**(?) komineer objek lêers en biblioteek leers om uitvoerbare program kode te lewer



https://www.sciencedirect.com/topics/engineering/memory-layout

ARM vs. C - Simple program instructions

- The C-code can be translated to machine code as follows:

`int i;`
`i = i + 1;`
	`080054ae:    ldr     r3, [r7, #4]`
	`080054b0:    adds    r3, #1`
	`080054b2:    str     r3, [r7, #4]`

Addresses of instructions

Equivalent assembly from machine instructions

```
ldr     r3, [r7, #4]
adds    r3, #1
str     r3, [r7, #4]
```

LDR = load. Load current value of `i` from memory (R7 + 4)

Add 1 to the current value of `i`

STR = Store. Store new value for `i` back to memory

ARM Assembly program – Populate array

- Simple loop that stores incrementing numbers to memory

```c
uint8_t data1[100];
int i;
for (i = 0; i < 100; i++)
{
    data1[i] = (uint8_t)i;
}
```

```
data1       FILL        100             ;Allocate 100 bytes of data memory

            LDR         R1,=data1
            MOV         R2, #0
loop
            CMP         R2, #100
            BGE         the_end
            STRB        R2, [R1]
            ADD         R1, R1, #1
            ADD         R2, R2, #1
            B           loop
the_end
```

Start address: 0x100 End address: 0x1100

Word Address	Byte 3	Byte 2	Byte 1	Byte 0	Word Value
0x100	0x3	0x2	0x1	0x0	0x3020100
0x104	0x7	0x6	0x5	0x4	0x7060504
0x108	0xB	0xA	0x9	0x8	0xB0A0908
0x10C	0xF	0xE	0xD	0xC	0xF0E0D0C
0x110	0x13	0x12	0x11	0x10	0x13121110
0x114	0x17	0x16	0x15	0x14	0x17161514
0x118	0x1B	0x1A	0x19	0x18	0x1B1A1918
0x11C	0x1F	0x1E	0x1D	0x1C	0x1F1E1D1C
0x120	0x23	0x22	0x21	0x20	0x23222120
0x124	0x27	0x26	0x25	0x24	0x27262524

ARM Assembly program – Populate array

- Some notes:
  - We used R2 for the variable i. i is not stored in memory.
  - Should a variable always be stored in memory?
  - If the C compiler (translated C code into machine instructions) is used to compile the code below, how will it differ from ASM code?
  - Answer: Compiler optimization settings

- We used R2 to temporarily contain an address – the address that we are writing to.

```
data1       FILL        100

            LDR         R1,=data1
            MOV         R2, #0
loop
            CMP         R2, #100
            BGE         the_end
            STRB        R2, [R1]
            ADD         R1, R1, #1
            ADD         R2, R2, #1
            B           loop
the_end
```

```c
uint8_t data1[100];
int i;
for (i = 0; i < 100; i++)
{
    data1[i] = (uint8_t)i;
}
```

Number systems intro / Getallestelsels intro

- Refers to the numerical representation of a value.
- In digital logic we only work with to states – HIGH and LOW.
- It is clumsy to work with numbers between 0 – 9 (decimal) to perform mathematics with a two state (binary) system.
- We use different number systems to make life easier when working with and designing digital systems.

- Rekenaars gebruik binêre voorstelling vir alles. Heksadesimaal word gebruik vir leesbaarheid en gerief

Decimal numbers / Desimale getalle

- Just as you (probably) have ten fingers, there are ten decimal digits from 0 to 9. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- Each digit (right to left) represents a multiple of a power of 10.
- Decimal numbers are also called **base-10** (or radix-10) numbers.

Indicating the **base** of the number

1000's	100's	10's	1's
9	7	4	2

$$9742_{10} = \underbrace{9 \times 10^3}_{\text{nine thousands}} + \underbrace{7 \times 10^2}_{\text{seven hundreds}} + \underbrace{4 \times 10^1}_{\text{four tens}} + \underbrace{2 \times 10^0}_{\text{two ones}}$$

- An N-digit decimal number represents one of 10^N possibilities. This is called the **range** of the number.
  - Three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

Binary numbers / Binêre getalle

- Binary digits or "bits" represent one of two values, 0 or 1.
- Each bit (right to left) represent a multiple of a power of 2.
- Binary numbers are **base-2**.

16's	8's	4's	2's	1's
1	0	1	1	0

$$10110_2 = \underbrace{1 \times 2^4}_{\text{one sixteen}} + \underbrace{0 \times 2^3}_{\text{no eight}} + \underbrace{1 \times 2^2}_{\text{one four}} + \underbrace{1 \times 2^1}_{\text{one two}} + \underbrace{0 \times 2^0}_{\text{no one}} = 22_{10}$$

- An N-bit binary number represents one of 2^N possibilities.
  - A four-bit binary number represents one of 16 possibilities. For positive numbers, this is the range of 0 to 15.
- Binary numbers are convenient for logic circuits in computers, since they can represent 2 states, i.e. low and high voltages.

Hexadecimal numbers / Heksadesimale getalle

- Decimal numbers are useful, because they are used by people.
- Binary numbers are useful, because they are used by computers.
- Hexadecimal (**base-16**) numbers are useful, because they provide a shorthand notation for binary numbers.
- Each hexadecimal digit represents a group of four bits (a nibble).
- Hexadecimal numbers use the **digits 0 to 9** along with the **letters A to F**, to represent a total of 16 possible values (from 0 to 15).
- Each digit (right to left) represents a multiple of a power of 16.

256's	16's	1's
2	E	D

$$2ED_{16} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{10}$$

two two hundred fifty six's

fourteen sixteens

thirteen ones

Conversion table / Omskakelings tabel

Table 1.2 **Hexadecimal number system**

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Remember:
In C-code, prepend hexadecimal literals with '0x'

0xFF = $FF_{16} = 1111\ 1111_2$

0xA5 = $A5_{16} = 1010\ 0101_2$

In C-kode word heksadesimale konstantes voorafgegaan deur '0x'

Binary representation / Binêre voorstelling

- In memory (SRAM, registers, etc) all we have is a pattern of bits.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0				0				E				8				9				0				0				0			

- Is it
  - A number? (signed, unsigned, floating point)
  - Character?
  - Machine instruction?
- The computer does not know what it is! To the computer this is simply a bit pattern.
- How the data in memory (SRAM, register, etc.) is interpreted depends on the current state of the CPU, and which instruction is about to be executed – it depends on you!

Integers / Heelgetalle

- The maximum integer number that can be represented depends on the number of bits being used
- Unsigned numbers range from 0 to 2^n-1
- Signed numbers range from -2^{n-1} to $+2^{n-1}-1$

Length	No. of bits	Unsigned range	Signed range
Byte	8	0 to 255	-128 to 127
Half-word	16	0 to 65535	-32768 to 32767
Word	32	0 to 4,294,967,295	–2,147,483,648 to 2,147,483,647
Double word	64	0 to $2^{64}-1$	-2^{63} to $2^{63}-1$

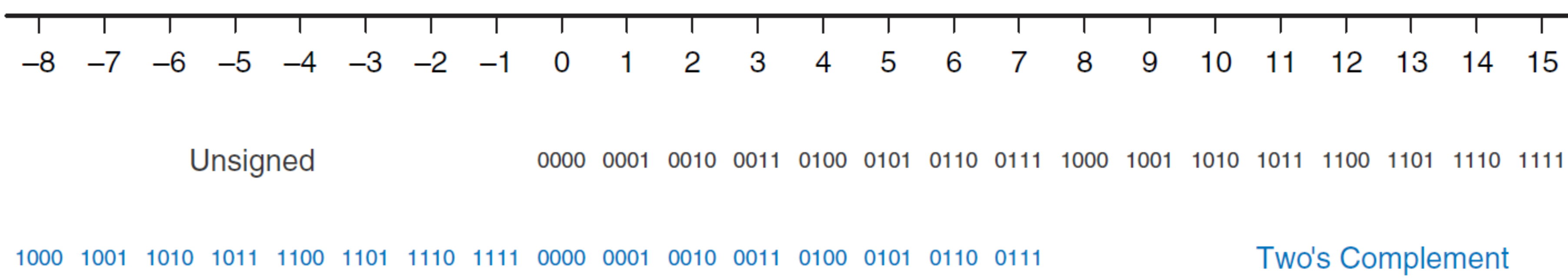
- Heelgetalle sonder teken bis kan strek van 0 tot 2^n-1. Heelgetalle met teken-bis kan strek van -2^{n-1} to $+2^{n-1}-1$ (waar daar n bisse gebruik word vir die voorstelling)

Integers / Heelgetalle

- Signed integer numbers (numbers which can be positive or negative) are represented using 2s complement / Heelgetalle wat positief of negatief kan wees word deur 2s komplement voorgestel

7	6	5	4	3	2	1	0
Sign	Number						

- The processor will perform all arithmetic as if the number is unsigned. The difference is in the **interpretation** of the bit pattern.
- 2s complement are identical to unsigned binary numbers, but the most significant bit has a weight of -2^{N-1} instead of 2^{N-1}.
  - The most positive number is: $01...111_2 = 2^{N-1}-1$
  - The most negative number is: $10...000_2 = -2^{N-1}$



C number types / C getal tipes

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`unsigned short`	16	0	$2^{16} - 1 = 65{,}535$
`long`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`unsigned long`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

C number types / C getal tipes

- How to declare C variables for portable code:
- Platform provides a stdint.h header file, for instance:

```c
typedef signed char int8_t;
typedef short int   int16_t;
typedef int         int32_t;
typedef unsigned char      uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int       uint32_t;
```

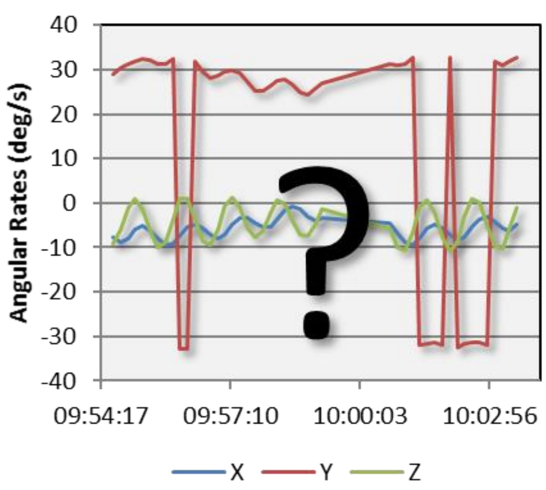
- In your source file:

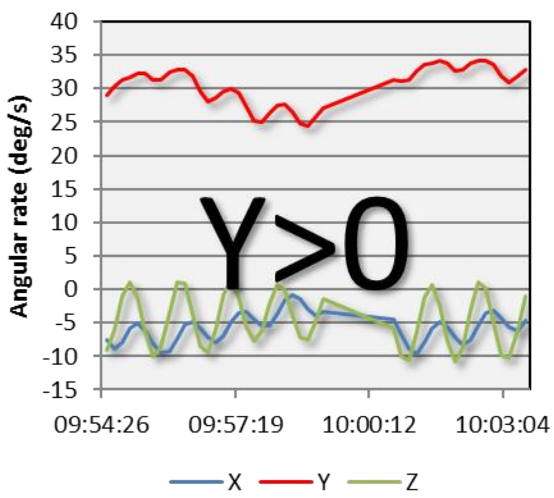
```c
#include <stdint.h>

uint32_t variable = 5;  // guaranteed to be a 32-bit int
```

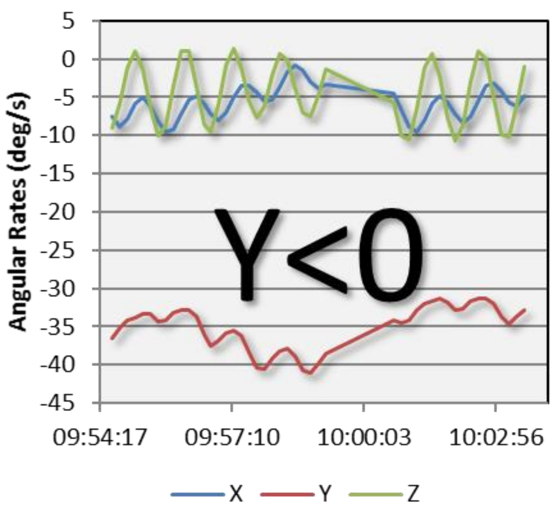
Integers / Heelgetalle

Example – Satellite spin rate / Voorbeeld – satelliet spin tempo

- Satellite angular rate vector (how fast it is spinning about each axis) telemetry uses 16-bit signed integer with x1000 scaling (fixed point decimal representation)
- Angular rate can vary from -32.768 to 32.767 deg/s
- Initial angular rates were quite a lot higher than expected. Values "jumping" between +32 and -32.



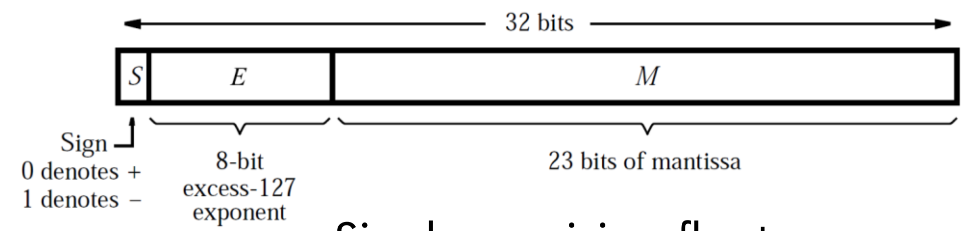




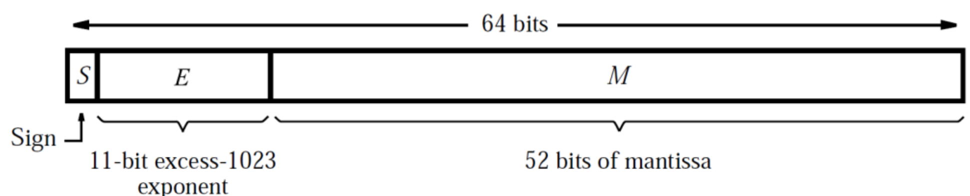
- What is the actual sign of the Y component?

Floating point numbers / Wisselpunt getalle

- Increase the dynamic range with which a number can be stored – use scientific notation / Verhoog die dinamiese bereik waarbinne 'n getal gestoor kan word – gebruik wetenskaplike notasie
- Value = $(-1)^s \times (1.M) \times 2^{E-b}$
- S = Sign bit
- M = Mantissa (fraction)
- E = Exponent
- b = Bias/excess



Single precision float



Double precision float

- Single precision provides typically 6–9 digits of numerical precision, while double precision gives 15–17.
- Convenient online converter:
  http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

Characters / Karakters

- Numbers (bit patterns) can also be interpreted as characters (each character has a unique code) / Getalle kan ook geinterpreteer word as karakter kodes
- ASCII used most commonly / ASCII word algemeen gebruik
- ASCII code for '0' is 48, '1' is 49 …
- ASCII code for 'A' is 65, 'B' is 66 …

```
char string1[] = "abc";
unsigned char string2[] = {97, 98, 99, 0};
```

- Memory contents of string1 and string2 will be the same

ASCII Table

Table 6.5 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Special Characters / Spesiale karakters

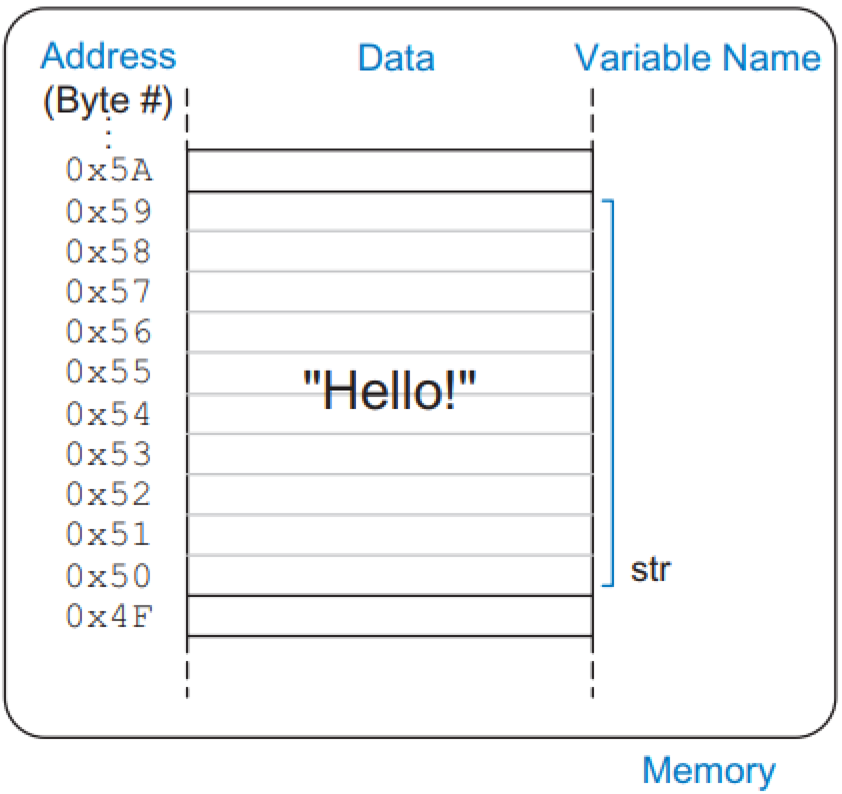
- ASCII special character codes

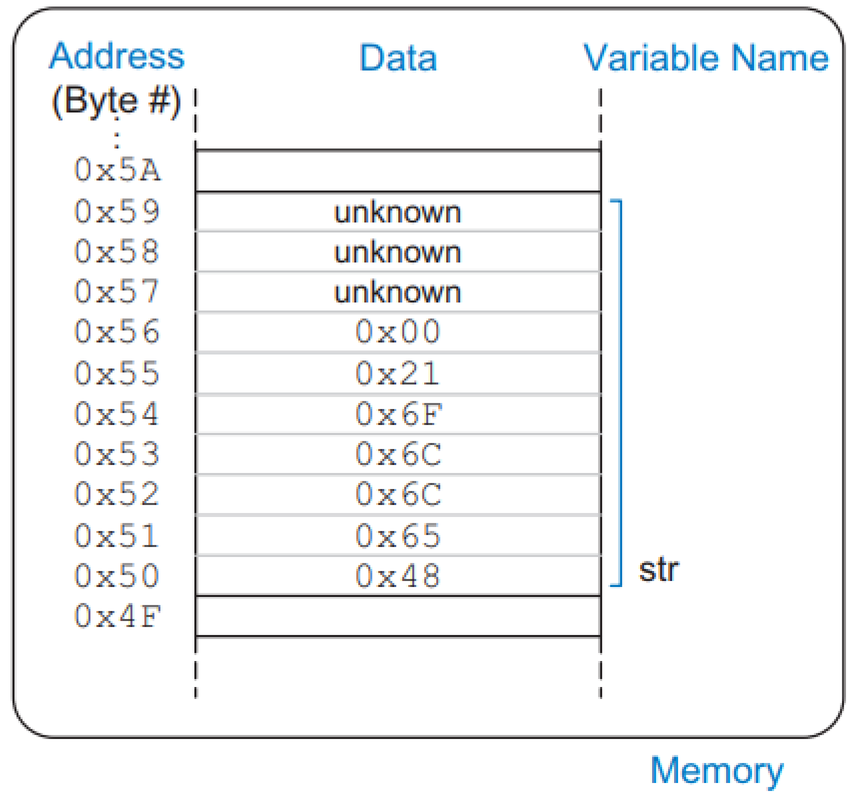
Special Character	Hexadecimal Encoding	Description
`\r`	0x0D	carriage return
`\n`	0x0A	new line
`\t`	0x09	tab
`\0`	0x00	terminates a string
`\\`	0x5C	backslash
`\"`	0x22	double quote
`\'`	0x27	single quote
`\a`	0x07	bell

C-string / C-stringe

- Strings in C are character arrays with “null termination” – a byte value of 0.

```c
char greeting[10] = "Hello!";
```





Boolean type / Boolse tipe

C89 does not have built-in Boolean types (*)

```c
#include <stdbool.h>

bool conditionA = false;
bool conditionB = true;

if (conditionA && conditionB)
    do_stuff();
```

In stdbool.h:

```c
#define true   1
#define false  0

#define bool _Bool
```

Is equivalent to (*)

```c
int conditionA = 0;
int conditionB = 1;

if (conditionA && conditionB)
    do_stuff();
```

*sort of. Depends how the compiler implements the _Bool type of C99 standard of C language

sizeof(…)

What does sizeof() return in C?

- Allocated storage size for variable
- sizeof(unsigned char) → 1
- sizeof(uint32_t) → 4
- sizeof(_Bool) → implementation specific
- sizeof(int) → implementation specific

For arrays and pointers:

- char data[40];
- char* ddata = data;
- sizeof(data) → 40
- sizeof(ddata) → num of bytes to store a pointer (address, e.g. 32 bits)

NB! sizeof() is not an actual function. It is an operator. There is no code that will calculate how many elements there are in your array. It simply asks the compiler to replace the "sizeof(…)" text with its implementation for that variable