Computer Systems 1

Dr. John T. O'Donnell

Joao Almeida-Domingues[*]

University of Glasgow

January 7th, 2019 – May 24th, 2019

Contents

[*]2334590D@student.gla.ac.uk

1 Analogue and Digital Representation

1.1 *Computer Systems*

Lecture 1
January 8th, 2019

1.1 definition. Digital Circuits Electronic systems which use very large numbers of only a few types of components, which when connected the right way create incredibly complex , useful behaviour

1.2 definition. Machine Language Programming language executed directly by the computer hardware, which serves as the interface between low-level circuits and high-level software. Therefore, it is simple enough so that a digital circuit can be designed to execute it, yet also powerful enough that high-level languages can be converted into it

1.2 *Data Representation*

There are several digital data types which need somehow be represented in hardware. This is done by encoding electrical signals (voltage). It is by manipulating these voltages in the hardware that computations are able to be performed. There are 2 ways of achieving this:

1. **Analogue**

   1.3 definition. The variables in the calculation are *analogue* to the physical signals being measured (e.g. $1 \iff 3\text{v}$)

   There are some advantages to this method, such as fast calculation of differential equations. Unfortunately, it also has significant drawbacks: limited precision; errors accumulate; difficult to represent non-numerical data

2. **Digital**

   1.4 definition. As indicated by the root of the word, it makes use of digits. It is by counting that calculations can be performed.

   This addresses some of the drawbacks of analogue computing. For example, one can always add more digits when higher precision is required; Handles noise and errors better; Easy representation of most data types

1.5 definition. Bit Unit of information used in digital circuits. It represents either a 0 or a 1

1.6 remark. Different circuits use different voltages, but the more simpler and reliable ones tend to use just two clearly distinct voltages to represent each value $(0, 1)$

1.7 definition. Flip Flop The simplest circuit with memory. Basic element of computer memory

1.8 definition. Byte 8 bits

Note that just like in any other positional number system, the number of possible representations grows exponentially with the number of digits added. In the binary system we only have $0, 1$. But by using basic knowledge of combinatorics, one observes that for every new position n the number of possible orderings doubles. It follows, for example, that for an 8 bit binary number one can represent up to $2^8 = 256$ different values ($0 \rightarrow 255$). In general, for a k bit number:

$$2^k \text{ possible values. From 0 until } 2^k - 1$$

1.9 notation. By convention, 4 bits are separated by a space for readability

$$0000\ 0000$$

1.10 definition. Word For convenience, for larger numbers of bits, the term *word* is used. The specific number of bits which a word represent changes. For this class: $16, 32, 64 =$ short w., w., long w.

Lecture 2
January 10th, 2019

2 Number Systems

2.1 *Conversion*

Conversions between different bases are easily achieved, by keeping in mind the simple fact that each position is able to represent a minimum value, and a maximum value, every time that value is exceeded , we need another "slot" to reset the counting with the "symbols" (i.e the digits) available in that system. To make it clear, let's first look at the decimal system, as first learned in primary school. 19_{10} can be thought of $10 + 9$, which in turn can be thought of 9 single units + 1 single "10" unit. Since 9 is the highest "unit symbol" available in the decimal number system we need another slot "the tenths". Transitioning from 9 to "10". Note how the "counter" was reset to 0 and we now start counting again from 1, but in the tenths and in the units slot. We don't need to add another slot until all digits, in all possible combinations, have been used.

Similarly, binary/base 2, simply means that for every "slot" we have only two symbols at our disposal, 0 and 1. And instead of thinking of the slots as $10^{ths}, 100^{ths}, \ldots$ we think in terms of powers of 2.

128	64	32	16		8	4	2	1	
2^7	2^6	2^5	2^4		2^3	2^2	2^1	2^0	
0	0	0	0		0	1	1	1	$= 4 + 2 + 1 = 7_{10}$

From decimal to binary the process is similar, but in reverse. If a primary school child was given a decimal table, and was asked to fill it with 150 , she would obviously not start by the units and start incrementing 1-by-1. She would notice that she'll need at least the 1 unit of "100" (10^3) and immediately put a 1 in the 3rd slot, and then notice that she'll need 5 "10s" and so on. This is trivial, and similarly trivial it is for base 2. Say you're given 74_{10}. What is the highest slot you can immediately fill? Well, 128 it's too high, the one before it is $64 = 2^7$ (position 7). That works, but it still leaves us with 10 units to be represented. Repeating the process, we see that we'll need an $8 = 2^3$ (position 3) and a 2 (position 2) until all units have been distributed amongst the "slots". Giving us:

0100 1010

This process is less straightforward for other bases, simply because we're used to think in base10. But the method is similar for all. Another useful base is **base 16**, where we use the letters A - F to represent the numbers from $10 \rightarrow 15$. This has the obvious advantage of reducing the number of "symbols"/characters needed for representing a binary number. For example: an 8 bit number can be easily represented by just 2 characters, with plenty leftover, since $16^2 = 256$ possible values.

2.1 notation. For this class "$" is used to represent hexadecimal numbers

The simplest way to convert from Binary to Hex is simply to note that, we need 4 bits to represent 1 hexadecimal bit in its entirety (i.e. 0 - F) . Hence we split the number into 4 bit chunks and convert them into the corresponding hex character. Such that the first 4 bits correspond to the hex uni bit , the next 4 bits to the "16^{ths}" , etc.

0	1	0	0	1	0	1	0
4				A			

The reverse is done in the same way as in the decimal above, but again working only in 4 bit chunks

2.2 remark. Note however that it is always more natural to filter through decimal instead of thinking "$2^4 - 1 = \mathrm{F}$"

2.2 *Operations*

Multiplication and Addition are performed much like in the same way we perform operations in decimal, again keeping in mind the digits available to work within each system, every time we exceed the limit we must carry onto the next.

Subtraction can be seen as a special case of addition with negative numbers, i.e. $3 - 2 = 3 + (-2)$. Binary numbers cannot represent negative numbers however, so we use **2's complement**. A 2c number is negative if its leftmost digit is 1

2.3 remark. Note that this does not affect the number of possible values that can still be represented, but it reduces the possible positive number representations in half, since for every positive number there will be a corresponding negative.

except from the lowest possible negative, which positive counterpart will always need an extra bit, since we're counting from 0

The general conversion method is straight forward:

1. Invert the bits
2. Add 1

It can be useful to think of the rightmost digit of an n bit 2c number as always representing the sum of the highest possible negative number ($-2^{(n-1)}$) with an $n-1$ bits positive number. The n^{th} bit is just the sign bit

2.4 remark. Note that, by adding all the positives we'll get 0, in which case the sign bit is turned off. We can think of 0 as belonging to the positives in this case, since it has the sign bit off. But when not considering it, even though we can represent $2^{(n-1)}$ negative numbers we can only represent $2^{(n-1)}-1$ positive ones. Since the "n^{th}" would in effect be 0.

2.5 example.

1	0	1	0	
-8	$+0$	$+2$	$+0$	$=-6$

Since it is a 2c number we know we can represent up to $2^3 = 8$ different values in each "direction" (considering 0 a positive number). So we have -8 and then we need to find out the 3bit binary. ($0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 = 2$). And then add them

It is clear from the example above how subtraction and division can then be performed just like addition

3 Logic Gates

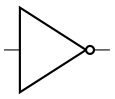
Lecture 3
January 15th, 2019

3.1 definition. Logic Gate A physical, basic component, which takes 1+ input(s) and performs a boolean function (i.e the result is either T/F)

1. **Inverter**

   3.2 definition. Takes an input and returns its opposite

   3.3 notation. `inv`



**alternatively $\bar{p}$*

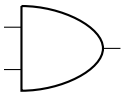
p	$\neg p^*$
1	0
0	1

2. **2-Input AND**

   3.4 definition. Returns true *iff* the two inputs are true

   3.5 notation. `and2`

the 2 used in the gates' names just means that 2 inputs are passed



p	q	$p \wedge q$
1	0	0
1	1	1
0	0	0
0	1	0

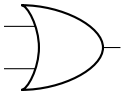
   3.6 remark. Note that for a n input truth-table, there are 2^n possible input combos

   3.7 remark. An easy way to construct truth tables is to start half(1)-half(0), then $\frac{1}{4}$, $\frac{1}{8}, \cdots$. So for a 3 input table, one first put 4 1s, 4 0s. Then, half of that, i.e. 2 1s 2 0s 2×, and finally 1, 0 , 8×

3. **2-input OR**

   3.8 definition. Returns true if any one of the two inputs are true, or if both are true

   3.9 notation. `or2`



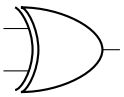
p	q	$p \vee q$
1	0	1
1	1	1
0	0	0
0	1	1

4. **2-input XOR**

   3.10 definition. Returns true *iff* one of the two inputs is true

   3.11 notation. `xor2`

*⊕



p	q	$p \veebar * q$
1	0	1
1	1	0
0	0	0
0	1	1

3.1 *Combinational Circuits*

3.12 definition. Combinational Circuits Their output depends only on their current input. There's no feedback loop, or "memory" state

For complex circuits it is hard to keep track of every input and output by just connecting single logic gates. It is useful to create *black box circuits* to abstract some of the complexity out. These components, regardless of what logic gates are used to built them*, always perform the same operation (e.g. choosing between two inputs).

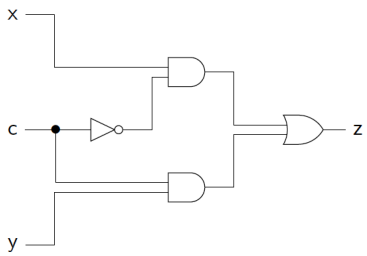
*hence the name black box

1. **Multiplexer**

   3.13 notation. `mux1`

   3.14 definition. Hardware equivalent of the *"if-then-else"* statement. Chooses between 2 values (x, y) given a third (c). If c y, else x

   3.15 remark. Every decision a computer makes comes down to a multiplexer

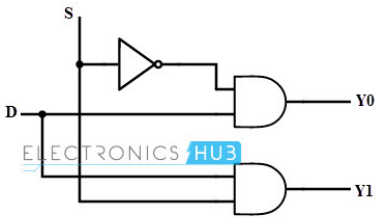


c	x	y	z
1	1	1	1
1	1	0	0
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	0
0	0	0	0

2. **Demultiplexer**

3.16 definition. The opposite of a multiplexer, it takes a single input converts it into several outputs. It its used to select where to send the input data

3.17 remark. For 2^n possible outputs , n selection signals/lines are required (i.e. 1-to-2 dmux requires 1 control signal)

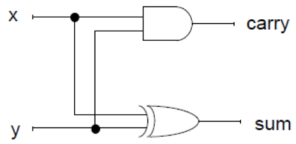


S	I	Y_1	Y_0
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

3. **Half-Adder**

3.18 definition. Adds two bits using a 2-bit (carry,sum) representation.

Note that there's only a carry if both inputs are 1, hence we represent this by an and2 gate, while the sum can be represented by an xor2

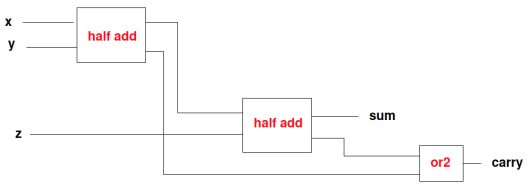


x	y	$c(x \wedge y)$	$s(x \veebar y)$
1	1	1	0
1	0	0	1
0	1	1	1
0	0	0	0

4. **Full-Adder**

3.19 definition. Adds three bits (inputs + carry) using a 2-bit (carry,sum) representation.

Note that there's only a carry when 2+ inputs are 1, and there's only a sum iff there are an odd number of inputs equal to 1



x	y	z	c	s
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

4 Boolean Algebra & Arithmetics

Lecture 4
January 17th, 2019

4.1 *Laws*

4.1 definition. Idempotence operation that can be applied several times without changing the result

$$x \vee x = x \qquad x \wedge x = x$$

4.2 definition. Commutative operations can be performed in any order

$$x \vee y = y \vee x \qquad x \wedge y = y \wedge x$$

4.3 definition. Associative the order in which the operations are performed does not matter as long as the sequence of the operands is not changed

$$x \vee (y \vee z) = (x \vee y) \vee z$$
$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

Proof.

x	y	z	y ∨ z	x ∨ y	x ∨ (y ∨ z)	(x ∨ y) ∨ z
1	1	1	1	1	1	1
1	1	0	1	1	1	1
1	0	1	1	1	1	1
1	0	0	0	1	1	1
0	1	1	1	1	1	1
0	1	0	1	1	1	1
0	0	1	1	0	1	1
0	0	0	0	0	0	0

QED

4.4 definition. Distributive & Absorption

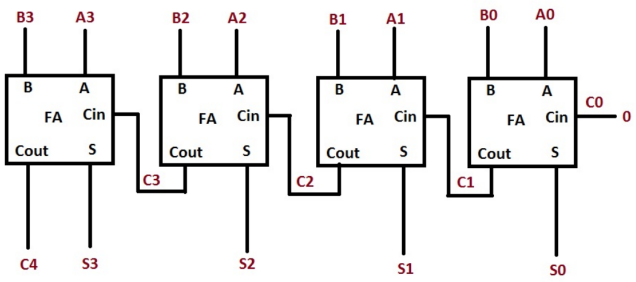
$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \wedge z)$$

$$x \wedge (x \vee y) = x$$
$$x \vee (x \wedge y) = x$$

4.5 remark. Useful for prove of correctness

4.2 *Arithmetic: Adding 2 Integers*

4.6 definition. 4-bit Ripple Carry Adder Uses 4 full adders to add two 4-bit numbers



4.7 remark. Note how the carry from each previous operation is passed onto the next, just like how one does addition manually

4.8 remark. Subtraction can be done much in the some way, by first applying 2's c to one of the inputs

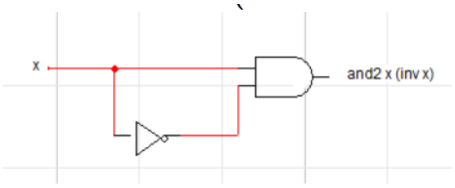
5 Synchronous Circuits

Given that logic gates are physical devices, there is a time delay between an input change and the new output. This makes the circuit stop obeying the boolean laws

5.1 remark. the clock running faster than it should may lead to invalid outputs, but if it is running slower it simply leads to loss of computational performance/speed

5.2 remark. Even though the time delay for a gate is marginal, for a whole circuit the delays add up

5.3 definition. Gate Delay Time taken for a gate to respond to a change of input with the correct output



5.4 example. The expected output of the following circuit would be 0, however when changing from, 1 to 0, the delay of the `inv` gate will cause both inputs of the and gate to be 1, which in turn will erroneously output a 1

5.5 definition. State The stored contents of the memory elements of a circuit, at a given point in time, is collectively referred to as the circuit's state and contains all the information about the past to which the circuit has access

5.6 definition. Delay Flip Flop (dff) component which endows a circuit with state. It has 2 inputs (saved value, clock tick) and 1 output (state value)

Every time the `dff` receives a clock signal, it updates its state value . Since the physical components deal with analogue signals, the *clock tick* can not be represented discretely. The workaround is to have the `dff` treat the voltage spike as the tick

There is only one clock signal for all flip flops, hence they are all updated simultaneously

5.7 remark. The output of a `dff` is independent of its input

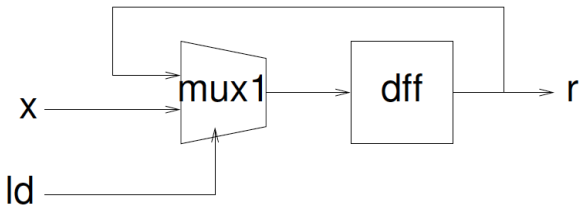
5.8 definition. Synchronous Circuit

1. Every flip flop must be directly connected to a global clock
2. No logical functions can be applied to the clock signal
3. Every clock tick much reach all flip flops simultaneously
4. Every feedback loop must pass through a flip flop
5. The inputs to the circuit remain stable throughout clock cycles

The cycle must be long enough to allow for all signals to become valid

At every clock tick the `dff` state gets updated, in order to store its state for longer we can use a register (`reg1`). If the control input is 1, then the value from the `dff` is loaded into the `reg1` otherwise it remains the same. The conditional is implemented by an `mux1`

5.9 definition. Register Allows the bit to be recorded between cycles, until a new value is loaded. It takes 2 inputs (`ld` =1,0 and the value bit) and it outputs the state bit




```
dff_input = mux1 ld old_state x
```

The register can be simulated by way of a simulation table:

Cycle	Inputs		State	Internal
	ld	x	r	dff_input
0	1	1	?	**1**
1	1	0	**1**	0
2	0	1	0	**0**
3	1	1	**0**	1

6 Register Transfer Machine

Lecture 5
January 25th, 2019

The purpose of an RTM is to use the memory provided by the registers we've learned about in the last lecture to allow simple assignment statements to memory and arithmetic operations to be carried out. To be able to achieve this in a digital circuit, we'll need (1) simple statements (2) A small number of types of statement with a fixed form

6.1 definition. Opcode portion of a machine language instruction that specifies the operation to be performed

6.2 definition. Instruction a group of several bits in a computer program that contains an opcode and usually one or more memory addresses

6.3 notation. Assignment `x := y`

6.4 example. Performing an assignment operation, between the value 2 and the Register 1:

```
R1 := 2
```

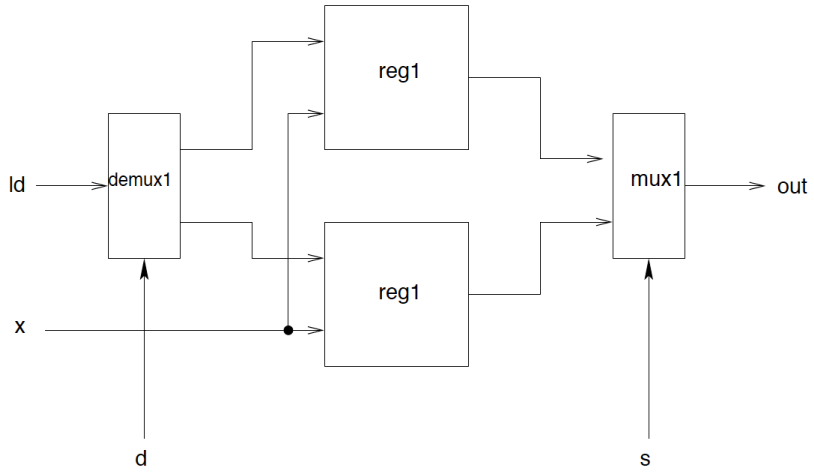
6.1 *Register File*

6.5 definition. Register File Array of registers (5.9)

The register file circuit enables the user to:

1. Specify, address and read out a specific register

2. Specify and load a value into a new address



Inner Workings

- The operation to be performed is controlled by *control signals*. These include:
  - the "on"/"off" signal or the "load" signal `ld`
  - the memory addresses to the individual registers `d, s`
  - "read"/"write"
- The input passed onto the circuit `x` is loaded into one of the registers **only** when a clock tick occurs **and** if `ld =1`.
- During the cycle select operations may occur, and in the end data is output from a specific register using a memory address passed as another control signal `s`

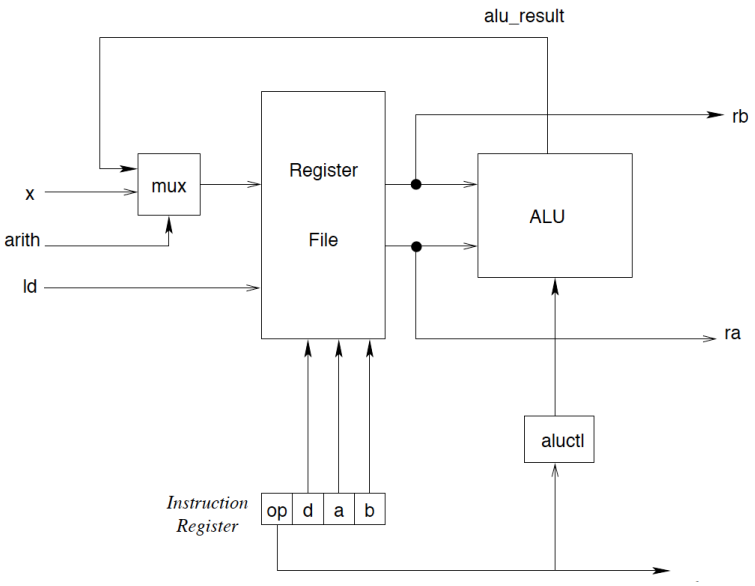
6.6 remark. Note that only the register at address `d` will be modified, since the demultiplexer (3.16) will make sure that any other register will have a load control of 0

6.7 remark. Note also that since a `dmux` (3.16) is able to generate 2^n outputs from n inputs, it follows that for controlling a k bit register only $\sqrt{k}$ address bits are required

6.8 remark. When more than one address needs to be read out, we can pass more source addresses (`s1 s2 s3`) into the multiplexer

6.2 *RTM Circuit*

Note that the register file allow us to update and read values from inputs, but our intention is to also be able to perform simple arithmetic operations on the values in the registers. This can be done by connecting an adder and a register file in a feedback loop, i.e. the n outputs of the register file, become the n inputs of the adder


alu_result
rb
x
mux
Register
File
ALU
arith
ld
ra
aluctl
Instruction
Register
op
d
a
b

Inner Workings

- A multiplexer is added, which allow the user to decide wether to pass onto the register a new value or to pass the result from the adder (hence the feedback)
- If the select signal is on `arith = 1`, then the file register circuit loads into the register [d] the result of the adder in the way describe above
- Otherwise, it simply loads the new external output

```
if ld
    then if arith
        then reg[d] := reg[sa] + reg[sb]
        else reg[d] := x
```

So now we have a 3rd useful feature provided by the RTM (on top of the 2 previously seen provided by the register), the possibility of adding two numbers *in memory*. So the following is now possible :

1. Assign an external input to a register `R[address] = constant`
2. Adding (or subtracting) two values in memory `R[a] = R[b] + R[c]*`

the two values being added can be from the same register R[a] = R[b] + R[b] or R[a] = R[a] + R[b]

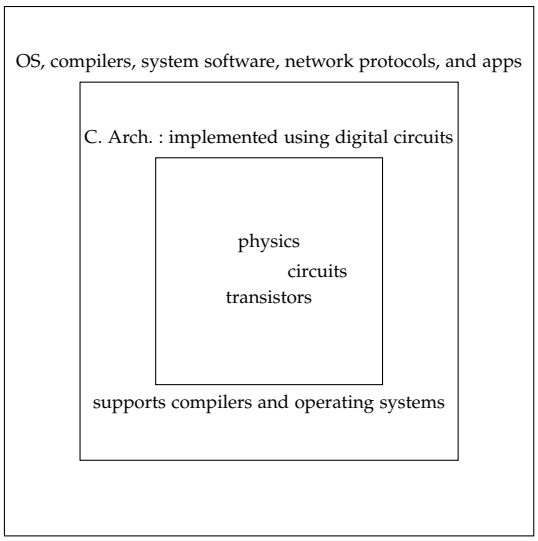
6.9 remark. Note however that the following is not possible *within one cycle* `R[a] = c + R[b]`, since `c` is not in memory

7 Computer Architecture

Lecture 6
January 29th, 2019

7.1 definition. Computer Architecture Defines the structure and the machine language of a computer

7.2 remark. Computer Systems' Abstraction Hierarchy



7.1 *Machine Language*

7.3 definition. Machine Language The language made up of binary-coded instructions that is used directly by the computer

7.4 example.
```
01010000 : operation specifier
(4-bit opcode , 1-bit register specifier , 3-bit addressing mode specifier)
0000000001001000 : operand ($0048 = char "H")
```

7.5 remark. Each CPU has its own machine language

High-Level programming languages are *compiled* (converted) into machine language by a compiler. This has the obvious advantage of making programming languages extremely versatile, since one language can be used across many machines, as long as a compiler for it exists

7.6 definition. Compiler Software responsible for translating HLL (nowadays) directly into ML

The first abstraction from ML was what is know as an *assembly language*, where letter codes represent instructions, instead of bits.

7.7 definition. Assembly Language A low-level programming language in which a mnemonic represents each of the machine-language instructions for a particular computer

7.8 definition. Assembler Translates AL into ML

7.9 remark. For this course we'll use a research architecture developed at the university : `Sigma16`

`Sigma16` features:

1. it has only 16 16-bit registers
2. uses reduced instruction set

7.2 *Main Subsystems*

Register File

There are 16 , 16-bit registers. The register file is a volatile memory, used for storing variables for easy access like intermediate values in a calculation

7.10 notation. `Rn` represents the register `n`

7.11 notation. `$0000` represents the data to be stored in `hex`

7.12 remark. `R0` is reserved for the number 0 , and `R15` for additional transient information (e.g. overflow?)

Arithmetic Logic Unit - ALU

7.13 definition. ALU the circuit that performs arithmetic operations and logical operations (comparison)

Memory

7.14 definition. Memory large collection of words

The memory is much larger (65,536 addresses) and slower, and no arithmetic takes place in it. Instead, it is used for long-term storage.

7.3 *Input/Output - I/O*

7.15 definition. Input Unit A device that accepts data to be stored in memory

7.16 definition. Output Unit A device that prints, displays data or copies it to another device

7.4 *RTM Instructions*

Remember from above (6.2) that the RTM was able to do two things (1) add two numbers; (2) store a number. The following instructions , are how this can be done in `Sigma16`

1. **Arithmetic:** `op Rst, Rvar1, Rvar2` $\implies$ `Rst := Rvar1 op Rvar2`

   7.17 notation. `add , sub , mul , div`

   7.18 example. `add R1,R2,R3 ; R1 := R2 + R3`

   7.19 remark. Note that the `;` is used to terminate a sentence, everything after it, in the same line is a comment

2. **Memory Access:**

   - `load Rn, x[R0]` copies `x` from memory into `Rn`
   - `store Rn, x[R0]` copies the word in `Rn` into the var `y` in memory
   - `lea Rn, 23[R0]` loads a constant into `Rn`

   7.20 remark. Note that usually the storage address is the first "argument" but when storing this is reversed

3. **Execution:** `trap R0,R0,R0` halts the program

4. **Variable Definition:** `x data 98` creates a variable `x` with an initial value of 98

   7.21 remark. `data` statements must come after all the instructions in the program

   7.22 example. A program that adds two integers:

```
load R1,x[R0] ; R1 := x
load R2,y[R0] ; R2 := y
add R3,R1,R2 ; R3 := x + y
store R3,z[R0] ; z := x + y
trap R0,R0,R0 ; terminate

;

x data 23
y data 14
z data 99
```

8 Control Structures

Lecture 7
January 31st, 2019

8.1 definition. Control Structures Statements that control the flow of execution

- Conditionals
- Loops
- Structuring Computation: functions, recursion, procedures

8.1 *Low Level Constructs*

The `jump <label>` statements transfer control to the line of code with the label passed. In `Sigma16` there are also two *conditional* jump instructions

`jumpf R1 , <label>[R0]` = if R1 is false, goto label

`jumpt R1 , <label>[R0]` = if R1 is true, goto label

8.2 remark. Labels must start with a letter, and must be at the beginning of the line

We can compare expressions using the following:

`cmplt R1 R2` ; returns true if R1 $<$ R2

`cmpeq R1 R2` ; returns true if R1 = R2

`cmpgt R1 R2` ; returns true if R1 $>$ R2

8.2 *Compilation Patterns*

It is easier to translate high-level languages into machine language, if it is done as a two-step process:

1. Translate complex statements to pseudo-code
2. Translate pseud-code to machine language

8.3 example. Translating *if $x < y$ then $a = 1$ else $a = 2$*

1.Pseudo

```
R3 := (x<y) ; assign boolean to R3
jumpf R4, else[R0] ; jump to label "else" if false
a := 1 ;
jump halt[R0]; jump to label "halt"
else a := 2;

halt trap R0 R0 R0; halt
```

2.Low Level

See slides for more examples of common idioms

```
lea R1, 10[R0] ;
store R1, x[R0] ;

lea R2, 5[R0] ;
store R2, y[R0] ;

load R1, x[R0] ;
load R2, y[R0];
cmplt R3, R1,R2 ; R3 := (x<y)

jumpf R3, else[R0] ;

lea R4, 1[R0];
store R4, a[R0];

jump halt[R0];

else lea R4, 2[R0];
store R4, a[R0];

halt trap R0 R0 R0;
```

9 Stored Program Computer

Lecture 8
February 5th, 2019

9.1 definition. SPC Both the program and the data are stored in the main memory

In Sigma16 there are 3 types of instructions, each with its own standard format

The third will covered in future lectures

9.2 definition. Instruction Format systematic way to represent an instruction using a string of bits

9.1 *Machine Language*

9.3 remark. Hexadecimal is used to represent each field

RRR

RRR instructions consist of 1 operation and 3 register operands (destination + operands). The assembler reserves 4bits for each field, so that one instruction has a total of 16bits = 1word

9.4 notation. Standard Names

op : op-code

d : destination register address

a : operand register address 1

b : operand register address 2

9.5 example. `add R13,R6,R9` becomes `0d69`

RX

RX instructions consist of 1 register - *destination register* - and a memory location composed of the "label" used for the address - *displacement* - and the *index register*

2 words are needed to represent it: the first has the 4 standard fields and the second contains the displacement

9.6 example. Let `x` be at the memory address `0008`, then `load R1,x[R0]` becomes `f101 0008`

9.7 remark. In this case `f` is always used as op and `b` indicates which operation is being executed

9.2 *Assembler: Memory Allocation*

1. The assembler keeps track of where to put the next line of code, by maintaining a variable with a 0-based index - *location counter*
2. It reads each line of code , if a label is value it maps onto the current location and stores it in the *symbol table*; it also decides how many memory an instruction requires and *adds this to the counter*
3. Reads the program a second time
4. Generates the code for each statement, looking up the stored value for labels when needed

Adder Program (7.22) in ML

address	contents	Meaning
0000	f101	first word of load R1,x[R0]
0001	0008	second word: address of x
0002	f201	first word of load R2,y[R0]
0003	0009	second word: address of y
0004	0312	add R3,R1,R2
0005	f302	first word of store R3,z[R0]
0006	000a	second word: address of z
0007	d000	trap R0,R0,R0
0008	0017	x = 23
0009	000e	y = 14
000a	0063	z = 99

9.3 *Control Registers*

CR are registers which the machine uses to keep track of what it is doing, but which are usually not accessible in the program

9.8 definition. Program Counter contains the address of the next line to be executed

9.9 definition. Instruction Register contains the instruction being executed

9.10 definition. Address Register for RX only; contains the address of the second operand

10 Arrays

Lecture 9
February 7th, 2019

10.1 definition. Data Structure Larger data entity composed of many smaller individual entities *(e.g. arrays, dictionaries, etc.)*

10.2 definition. Array/Vector sequence of indexed values

10.3 notation. x[i] represents the i^{th} element of the x array

Arrays can be represented in a computer by placing its elements in consecutive memory locations. The address of the array is equal to that of its first element `x[0]`. Every other element can be found by simple *address arithmetic*, i.e.

```
x[i] = x+i
```

10.4 notation. Defining the array:

```
; only x[0] has a label
x     data    13    ; x[0]
       data    189  ; x[1]
```

10.1 *Indexed Addressing*

Since the address of the array is given by the element at index 0, a given element is accessed by its *effective address*: the displacement value is added to the index register

10.5 example. `x[R4]` where `x` represents the address of the element labeled `x` - *displacement*; `R4` is the *index register*

10.6 remark. This is the reason why `[R0]` is used for simple variables

10.7 definition. Addressing Mode scheme for specifying the address of data *(e.g. displacement[index])*

10.8 example. Using Effective Addresses `x[i] := x[i] + 50`

```
lea    R1,50[R0]    ; R1 := 50
load   R5,i[R0]      ; R5 := i
load   R6,x[R5]     ; R6 := x[i]
add    R6,R6,R1    ; R6 := x[i] + 50
store  R6,x[R5]     ; x[i] := x[i] + 50
```

10.2 *Array Transversal & For Loops*

10.9 example. !!!ADD COMPLETE PROGRAM INSTEAD!!!

11 Records and Pointers

Lecture 10
February 12th, 2019

11.1 *Compilation Patterns*

Patterns make it easier to translate common idioms. When trying to perform a common task, all one needs to do is pay attention at (1) what common fixed parts you can reuse (2) what varies, i.e. what is specific to the task

11.1 example. While Loop

```
; < >  changeable parts

; High level

    while <bool> do
        <S>

; Low Level Pattern

        label1
            if <bool> = False goto label2
                <Statement>
                goto label1
        label2
```

11.2 *Pointers*

Pointers are a sort of "variable", which hold the address of an object. There are two operators, one which returns the address and the other which returns the value of the object

11.2 notation. Pointer : &

11.3 notation. Pointer's Value : *

```
lea R5,x[R0] ; R5 := &x ; address of x is stored in R5
load R5,x[R0] ; R5 := *x ; value at address [x + 0] stored in R5
```

Comparing accessing a variable directly vs by pointer

```
; directly

lea R1,5[R0] ; R1 := 5 (constant)
load R2,x[R0] ; R2 := x
add R2,R2,R1 ; R2 := x + 5
store R2,x[R0] ; x := x + 5

; pointer

; R3 := &x
lea R3,x[R0] ; R3 := &x

; Add 5 to whatever word R3 points to
lea R1,5[R0] ; R1 := 5 (constant)
load R4,0[R3] ; R4 := *R3
add R4,R4,R1 ; R4 := *R3 + 5
store R4,0[R3] ; *R3 := *R3 + 5
```

11.3 *Records*

Records are essentially an array where every entry is labeled. You can think of it as equivalent to a python dictionary. Each entry can be accessed directly, but the easiest way is to write a block of code using pointers, which can then be reused for all entries

11.4 example. Given a record labeled from A to C

```
; Set x as the current record by making R3 point to it
lea R3,x[R0] ; R3 := &x

; Perform the calculation on the record that R3 points
load R1,1[R3] ; R1 := (*R3).B
load R2,2[R3] ; R2 := (*R3).C
add R1,R1,R2 ; R1 := (*R3).B + (*R3).C
store R1,0[R3] ; *R3.A := (*R3).B + (*R3).C
```

11.4 *Requestes to OS*

Many operations cannot be perform directly by a user program, the main reason being that this would compromise the system's integrity. In order to perform those "sensitive" operations *(e.g. kill exec, read/write, memory allocation, etc)* there are instructions reserved which transfer control to the OS

11.5 remark. The specific codes used to make a request are defined by the OS, not by the hardware. This is the main reason of why compiled programs are not compatible across systems

Termination : `trap R0 R0 R0`

Write : `trap R1 R2 R3`

- R1 : the code that indicates a write request
- R2 : address of the first word of string to write
- R3 : the length of string

```
;To write a string named out, we use (1) lea to load a
    constant, (2) lea to load the address of an array,
    (3) load to get a variable

; write out (size = k)
lea R1,2[R0] ; trap code: write
lea R2,animal[R0] ; address of string to print
load R3,k[R0] ; string size = k
trap R1,R2,R3 ; write out (size = k)
trap R0,R0,R0 ; terminate
k data 4 ; length of animal

; animal = string "cat"
animal
data 99 ; character code for 'c'
data 97 ; character code for 'a'
data 116 ; character code for 't'
data 10 ; character code for newline
```

12 Procedures and Call Stack

Lecture 11
February 14th, 2019

12.1 definition. Procedure Block of code that performs common tasks

Instead of repeating code we can write a block of code and call it when necessary by restructuring the flow of the program. Once the procedure finishes the task, the value can be returned and the execution is transferred back to the main flow

12.2 remark. To avoid breaking the main flow, it is convenient to write the procedure someplace else with a label

Implementation

Note that even though the call is always made to the same place, different calls must return to different places, so it is not possible to just label the caller

statement. In order to remember the call origin the statement's address is saved into a register using the `jal` instrunction

12.3 example. Implementing Call `jal R13, fn[R0] ; saves fn address in R13` and Return `jump 0[R13]`

Passing Parameters

12.4 remark. By convention, arguments and results are passed in R1

```
;Main program
load R1,x[R0] ; arg = x
jal R13,work[R0] ; result = work (x)
...
load R1,y[R0] ; arg = y
jal R13,work[R0] ; result = work (y)
...

; Function work (x) = 1 + 7*x
work lea R2,7[R0] ; R7 = 2
lea R3,1[R0] ; R3 = 1
mul R1,R1,R2 ; result = arg * 7
add R1,R3,R1 ; result = 1 + 7*arg
jump 0[R13] ; return
```

Recursive Calls

Note that if a procedure is called recursively, then registers might be overwritten. In order to execute recursive calls, we need to *save the state* of the caller. This can be achieved, by using a *call stack*

12.5 definition. Stack Data container which only supports *push* and *pop*, hence at any given time, only the top element is accessible

Because most procedures usually need several registers, recursive calls would quickly exhaust the number of available registers. Hence the state needs to be saved in memory:

By convention, R13 will always be used as the return

1. At the start of the procedure, there must be a store instruction for its return address
2. At the end of the procedure, the return address must be loaded and passed to jump

Call Stack

12.6 definition. Call Stack Data container which stores all the information about calls and returns

Every call triggers the following to be stored in memory:

a pointer to the previous stack frame (needed for pop)

the return address

the saved registers

local variables (frees up working memory for the procedure)

We implement the stack by dedicating `R14` to the stack pointer , saving the word with the info at `0[R14] + 1` at call time, and reseting it when returning. (i.e. `0[R14] -1`)

12.7 remark. Note that `R14` must remain unchanged, otherwise the pointer to the stack will be overwritten

13 Variables

Lecture 12
February 19th, 2019

Variables are distinguished according to 3 features:

1. Lifetime: When it is created and destroyed
2. Scope: From where it can be accessed
3. Location: Address in memory

13.1 *Static*

Static Variables are created at the start of the program, and retain their values until the program terminates. The variables defined using the `data` instruction, initialized after the trap instruction are static

To avoid the duplication of instructions, and to allow for executable code to be shared, variables and code are often separated.

13.1 definition. Code Segment Shareable, read-only part of the program

13.2 definition. Data Segment Non-Shareable, read-write part of the program

13.2 *Local*

Local Variables on the other hand are defined within the program (e.g. within a function, a block etc.). Local variables have only one name, but there may be many instances of it (e.g. recursive calls). Hence, instead of storing them in memory using `data`, they are kept in *stack frames*

Local Variables can be accessed by addressing them *relative* to the stack frame

13.3 *Dynamic*

Dynamic Variables are created explicitly, and therefore they are not limited to blocks of code, and need not follow the push-pop order of the stack. Hence, they can neither be kept in the stack frame nor in the static data segment. They are instead kept in the *heap*

13.3 definition. Heap Data structure composed of a very large number of small objects. It contains a *free space list* - a data structure which points to all the free words of memory

When the variable is created a small chunk of memory from the heap points to the object, once the variable is destroyed the memory pointer is released back into the *free space list*

13.4 *Call Stack*

????

14 Linked Lists

14.1 definition. Linked List Linear chain of nodes

14.2 definition. Node Record with two fields: (1) *value* - word containing info (2) *next* - pointer to the next node

14.3 remark. The last element of a linked list has *value* = *nil*

As seen above, we can access the fields of a record using pointers' arithmetic. Hence, for a given pointer `p`

```
; load the pointer to the node into R1
load R1,p[R0] ; R1 := p
; load the value field at position 0 into R2
load R2,0[R1] ; R2 := (*p).value
; load the next field at position 1 into R3
load R3,1[R1] ; R3 := (*p).next
```

14.1 *Basic Operations*

14.4 remark. Assuming that `p,q` are in memory. Normally, these will be kept in registers so as to minimize the number of `load,store` instructions

Is Empty?

```
load R1,p[R0]
cmpeq R2,R1,R0
jumpt R2,pIsEmpty[R0]
; Not empty Body
...
; Empty Body
pIsEmpty ...
```

Return Value

```
load R1,p[R0] ; R1 := p
load R2,0[R1] ; R2 := *p.value
store R2,x[R0] ; x := *p.value
```

Return Next

```
load R1,p[R0] ; R1 := p
load R2,1[R1] ; R2 := *p.value
store R2,q[R0] ; q := *p.value
```

14.5 remark. Nothing new! Just like accessing a record,`value` has an offset of 0 and `next` has an offset of 1

14.2 *Transversing*

Using a while loop, with the stoping condition of *pointer* $\neq 0$. Since the last element of the list is nil

14.3 *Cons*

14.6 definition. Cons A term from functional languages. To *cons x onto y* loosely means to create a new object from y by "appending" an object x of the same type

14.7 example. Given a list `p = [1,2] , q = [3,1,2] ; cons(3, p)`

```
cons (x, p)
        { q := newnode ();
        (*q).value := x;
        (*q).next := p;
        return q;
        }
```

14.8 remark. Note that p is unchanged. A new node is created, which points to p with the new added value

memory management too detailed?

14.4 *Lists Vs Arrays*

Feature	Array	List
Direct Element Access	Yes (with arbitrary index)	No (only to element with pointer)
Traversing	Yes (for loop i++)	Yes (set pointer to p.next, until p is nil)
Memory p/el	= element	= el + node
Flexibility	Fixed & Fully Allocated	Variable & Dynam Allocated

14.5 *General Data Structures*

By using more pointers, more general structures can be created. For example: (1) Double Linked List (points back-forward) (2) Circular (no nill valued node)

14.6 *Abstract Data Type*

14.9 definition. ADT A data type defined by its behaviour (semantics) , i.e. the operations it supports. Thought of from the point of view of the user instead of the implementer

An example of this type of structure is a stack. The core idea behind it is to support "push-pop" of data, but it can be implemented in more than one way, depending on what is required by the program. For example, instead of array, a linked list can be used:

```
; List Stack

Empty stack as nil
Push x is implemented by stack := cons (x, stack)
Pop x is implemented by stack := (*stack).next

; Array Stack

; PUSH
; stack[stTop] := R1; stTop := stTop + 1

push load R1,x[R0] ; R1 := x
load R2,stTop[R0] ; R2 := stTop
store R1,stack[R2] ; stack[stTop] := x
lea R3,1[R0] ; R3 := constant 1
add R2,R2,R3 ; R2 := stTop + 1
store R2,stTop[R0] ; stTop := stTop + 1

; POP

; pop the stack, store top element into x
; stTop := stTop - 1; x := stack[stTop]

pop load R2,stTop[R0] ; R2 := stTop
lea R3,1[R0] ; R3 := constant 1
sub R2,R1,R3 ; R2 := stTop - 1
load R1,stack[R2] ; R1 := stack[stTop-1]
store R1,x[R0] ; x := stack[stTop-1]
store R2,stTop[R0] ; stTop := stTop - 1
```

14.7 *Error Checking*

The previous implementation does not check for errors, which might cause the program to crash or to output the wrong data. In order to prevent that, programs need to be able to check and handle errors.

- Error message
- Error code to caller, and transfer control
- Throw an exception
- Terminate Execution

15 Programming Techniques

15.1 *Compound Boolean Expressions: Short Circuit*

15.1 definition. Short Circuit When two conditions need to be satisfied, if the first fails, we can save resources by skipping the evaluation of the second

```
; while i<n && x[i]>0 do S
; if not (i<n && x[i]>0) then goto loopDone

load R1,i[R0] ; R1 := i
load R2,n[R0] ; R2 := n
cmplt R3,R1,R2 ; R3 := i<n
jumpf R3,loopDone[R0] ; if not (i<n) then goto
   loopDone
load R3,x[R1] ; R3 := x[i] safe because i<n
cmpgt R4,R3,R0 ; R4 := x[i]>0
jumpf R4,loopDone[R0] ; if not (x[i]>0) then goto
   loopDo
```

15.2 *Condition Code*

Instead of storing the comparison boolean in a register and jump, it is possible to use the instruction `cmp` which stores the result - *condition code* - in `R15` and then one of the condition jump operators (e.g. `jumplt`)

15.3 *Repeat-until Loop*

```
; stopping condition checked at the end

repeat
        {S1; S2; S3}
until i>n;
```

16 Arrays & Pointers

16.1 *Accessing Elements with Pointers*

```
lea R1,x[R0] ; point p to first el of array x store in
    R1
load R2,0[R1] ; access the value p points to store in
   R2
lea R1,1[R1] ; p++
```

16.1 remark. Note that the arithmetic happens on the pointers

16.2 *Sum*

```
; R1 = p = pointer to current element of array x
; R2 = q = pointer to end of array x
; R3 = sum of elements of array x

        lea R1,x[R0] ; p := &x
        lea R2,xEnd[R0] ; q := %xEnd
        add R3,R0,R0 ; sum := 0
sumLoop
        cmplt R4,R1,R2 ; R4 := p<q
        jumpf sumLoopDone ; if not p<q then goto
           sumLoopDone
        load R4,0[R1] ; R4 := *p (this is current
           element of x)
        add R3,R3,R4 ; sum := sum + *p
        lea R1,1[R1] ; p := p+1 (point to next element
            of x)
        jump sumLoop[R0] ; goto sumLoop
sumLoopEnd

x       data 23 ; first element of x
        data 42 ; next element of x
        data 19 ; last element of x
xEnd
```

16.2 remark. It is easier to use ArraySum with pointers for records

16.3 *Traverse*

```
; LOW LEVEL

; sum := 0;
; p := &RecordArray;
; q := &RecordArrayEnd;
; RecordLoop
; if (p<q) = False then goto recordLoopDone;
; *p.fieldA := *p.fieldB + *p.fieldC;
; sum := sum + *p.fieldA;
; p := p + RecordSize;
; goto recordLoop;
; RecordLoopDone

; Sigma16

; R1 = sum
```

```
; R2 = p (pointer to current element)
; R3 = q (pointer to end of array)
; R4 = RecordSize

        lea R1,0[R0] ; sum := 0
        lea R2,RecordArray[R0] ; p := &RecordArray;
        lea R3,RecordArrayEnd[R0] ; q := &RecordArray;
        load R4,RecordSize[R0] ; R4 := RecordSize
RecordLoop
        cmplt R5,R2,R3 ; R5 := p<q
        jumpf R5,RecordLoopDone[R0] ; if (p<q) = False
            then goto RecordLoopDone
        load R5,1[R2] ; R5 := *p.fieldB
        load R6,2[R2] ; R6 := *p.fieldC
        add R7,R5,R6 ; R7 := *p.fieldB + *p.fieldC
        store R7,0[R2] ; *p.fieldA := *p.fieldB + *p.
           fieldC
        add R1,R1,R7 ; sum := sum + *p.fieldA
        add R2,R2,R4 ; p := p + RecordSize
        jump RecordLoop[R0] ; goto RecordLoop
RecordLoopDone
```

16.4 *Stack Overflow*

16.3 remark. `R12`, `R11` registers dedicated to store info for procedures. 12 - *stack top* , 11 - *stack limit*

Initialize Stack

```
lea R14,CallStack[R0] ; initialise stack pointer
store R0,0[R14] ; main program dynamic link = nil
lea R12,1[R14] ; initialise stack top
load R1,StackSize[R0] ; R1 := stack size
add R11,R14,R1 ; StackLimit := &CallStack + StackSize
```

Call, Return, Stack Overflow

16.4 remark. arguments are passed in the first registers

```
; Create stack frame
store R14,0[R12] ; save dynamic link
add R14,R12,R0 ; stack pointer := stack top
lea R12,6[R14] ; stack top := stack ptr + frame cmp
   R12,R11 ; stack top ~ stack limit
jumpgt StackOverflow[R0] ; if top>limit then goto
   stack store R13,1[R14] ; save return address
store R1,2[R14] ; save R1
store R2,3[R14] ; save R2
store R3,4[R14] ; save R3
store R4,5[R14] ; save R4

; return
load R1,2[R14] ; restore R1
load R2,3[R14] ; restore R2
load R3,4[R14] ; restore R3
load R13,1[R14] ; restore return address
load R14,0[R14] ; pop stack frame
jump 0[R13] ; return

; If the stack is full and a procedure is called, this
    is a fatal error. Error message is printed
; StackOverflow

lea R1,2[R0]
lea R2,StackOverflowMessage[R0]
lea R3,15[R0] ; string length
trap R1,R2,R3 ; print "Stack overflow\n"
trap R0,R0,R0 ; halt

StackOverflowMessage
        data 83 ; 'S'
        data 116 ; 't'
        data 97 ; 'a'
        data 99 ; 'c'
        data 107 ; 'k'
...
```

17 Nested Conditionals

Lecture 14
February 28th, 2019

17.1 *Jump Tables*

A bunch of *if-then* or *case* statements are inefficient if we're looking to match a numeric code directly. This can instead be done by constructing an array made of jumps to certain conditional statements which match the code passed as an argument. A sort of table of "tasks"

```
; Jump to operation specified by code
add R4,R4,R4 ; code := 2*code
lea R5,CmdJumpTable[R0] ; R5 := pointer to jump table
add R4,R5,R4 ; address of jump to operation
jump 0[R4] ; jump to jump to operation

CmdJumpTable
        jump CmdTerminate[R0] ; code 0: terminate the
           program
        jump CmdInsert[R0] ; code 1: insert
        jump CmdDelete[R0] ; code 2: delete
        jump CmdSearch[R0] ; code 3: search
        jump CmdPrint[R0] ; code 4: print

CmdDone
        load R5,InputPtr[R0]
        lea R6,3[R0]
        add R5,R5,R6
        store R5,InputPtr[R0]
        jump CommandLoop[R0]
```

17.1 remark. If the code passed does not exist in the table, program will jump to random instruction. Error Check by comparing code with least/largest value in table

References

wikipedia

URL: https://en.wikipedia.org/wiki/Wikipedia.

Basic Electronics Tutorials and Revision **basic˙electronics**

Basic Electronics Tutorials and Revision. URL: https://www.electronics-tutorials.ws/.

Dale et al.: Computer Science Illuminated **csIlluminated**

Nell Dale and John Lewis. *Computer Science Illuminated*. Jones and Bartlett Learning, 2016.

Petzold: Code: the hidden language of computer hardware and software **petzold˙2001**

Charles Petzold. *Code: the hidden language of computer hardware and software*. Microsoft Press, 2001.

Velleman: How to prove it: a structured approach **velleman˙2009**

Daniel J. Velleman. *How to prove it: a structured approach*. Cambridge University Press, 2009.