

Computer Architecture and system programming

Kristoffer Klokke

2022

Contents

1	The basics	4
1.1	Structure and Function	4
1.2	Gates, memory cells, Chips, and Multichip modules	5
1.3	Processor architecture	6
1.4	Embedded systems	6
2	Performance	6
2.1	Measuring performance	7
3	Digital logic	9
3.1	Boolean algebra	9
3.2	Karnaugh maps	10
3.3	Quine-McCluskey method	11
3.4	Circuits	12
3.4.1	Multiplex	12
3.4.2	Decoders and encoders	13
3.4.3	Read-only Memory	13
3.4.4	Sequential circuits	13
3.4.5	Programmable logic devices	13
4	Instruction sets	14
4.1	Machine instructions	14
4.2	Types of operands	17
4.3	Types of operations	17
5	Addressing modes and Formats	18
5.1	Addressing modes	18
5.2	Addressing in x86	20
5.3	Addressing in ARM	21
5.4	Instruction formats	21
5.4.1	x86 instruction format	22
5.4.2	ARM instruction format	23
6	Assembly language concepts	24
6.1	Assembly language elements	25
6.1.1	Pseudo-instructions	25
6.1.2	Macro definitions	26
6.2	Types of assemblers	28
6.3	Loading and linking	29

7	Assembly language x86	29
7.1	Directives	30
7.2	Starting the code	30
7.3	Registers	30
7.4	System calls	31
7.5	Jumps	32
7.6	Stack and function calls	32
8	Computer Arithmetic	33
8.1	Sign-magnitude representation	33
8.2	Twos compliment	33
8.2.1	Twos compliment arithmetic	33
8.2.2	Twos compliment multiplication	34
8.2.3	Twos compliment division	35
8.3	Floating-point representation	35

1 The basics

Computer architecture: attributes of a system visible to the programmer, such as instruction set architecture (ISA) which defines opcodes, registers, instruction and data memory.

Computer organization: operational units and their interconnections, which are the behind the scenes of the architecture.

1.1 Structure and Function

A computer can perform 4 basic functions:

- Data processing - manipulate data in some form
- Data storage - in every computer some form of storage is needed even if it just temporary
- Data movement - data movement can be in many forms but most clear is the data movement from the input/output (I/O) referred to as peripheral
- Control - a control unit which can orchestrate the performance and functional parts of the computer

This therefore creates a computer structure of: CPU, Main memory, I/O, System bus.

The CPU are here a unit consisting of:

- Control unit - Control the operations sent to the CPU
- Arithmetic and logic unit (ALU) - perform the data processing
- Registers - storage for the CPU
- CPU interconnection - communication between the different units in the CPU

Some processors have multiple levels of cache where the higher level the faster yet smaller cache.

Some CPU may also have multiple cores which consist of:

- ISU (instruction sequence unit) - controls instructions sequence and allows for an out-of-order (OOO) sequence

- IFB (instruction fetch and branch) and ICM (instruction cache and merge) - These two subunits contain the 128-kB instruction cache, branch prediction logic, instruction fetching controls, and buffers.
- IDU (instruction decode unit) - fed from the IFU buffer it parses and decodes architecture operation codes
- LSU (load-store unit) - contains L1 data cache and controls data flow between L1 and L2 cache
- XU (translation unit) - Translate logical addresses into physical addresses
- PC (core pervasive unit) - Collects instrument data and errors
- FXU (fixed-point unit) - executes fixed point arithmetic operations
- VFU (vector and floating-point unit) - Handles all binary and hexadecimal floating point operations and fixed-point multiplication
- RU (recovery unit) - Keep a copy of the complete state in case of recovery
- COP (dedicated co-processor) - data compression and encryption functions for each core
- L2D - data cache for memory traffic
- L2I - instruction cache

1.2 Gates, memory cells, Chips, and Multichip modules

The only two required components for a digital computer are: gates and memory cells

A gate is a component which implements a boolean or logical function, ex AND gate.

A memory cell can be in two states at all time on or off and in this way save a bit.

A transistor is the electric based implementation of a gate or memory cell

1.3 Processor architecture

The Intel x86 by the complex instruction set computers (CISCs).

Unlike ARM which is based on reduced instruction set computer (RISC).

1.4 Embedded systems

These are system which are general purpose, but system where hardware and software (embedded system (OS)) are coupled together.

Theses system is found everywhere, and often working with the external environment via sensors.

Due to the software only having one purpose they are more efficient in both energy and processing power.

An embedded system may use a general purpose chip but most use a dedicated processor with specific number of needed tasks.

These dedicated chips often take form in microcontrollers which are sos called computers on a chips, small chips which have the same requirements for the 4 basic functions of a computer.

Deeply embedded systems are microcontrolelrs with burnt in programs and no interactions with the user.

2 Performance

In order to achieve the processing power of today different methods are being used to keep task comming to the CPU

- Pipelining - An execution of an instruction has multiple parts like: fetching instruction, decoding opcode, fetch operands and so on. Pipelining handles multiple executions by handling a different parts in every task.
- Branch prediction - By looking ahead in the instruction code, a prediction of needed instructions and buffers can be fetched beforehand.
- Superscalar execution - Multiple instructions are performed every processor clock cycle.
- Data flow analysis - By observing which instructions depend on other instruction result, a new order of instruction are made.
- Speculative execution - By using branch prediction and data flow analysis, the CPU speculates on upcoming instruction and execute them.

To create a new and faster CPU there are three approaches:

- Increase speed by reducing size of the chip, and therefore reducing the travel time of information
- Speed up cache size, to reduce to waiting time on slow data transformation
- Change processor organization and architecture to allow for better parallelism

But by increasing the speed and lowering the size of transistor, it creates new problem such as: Power density becoming higher and making it harder to dissipate heat, slower electron flow due to smaller connection which creates more resistance, Memory access speed which is a common constraint for CPUs.

Therefore a more modern solution is multi core processor designs, such more cores with shared cache can archive more speed.

Amdahls law describe how multicore can speedup a process as followed

$$Speedup = \frac{1}{(1 - f) + \frac{f}{N}}$$

Where f is the code which can infinitely be parallelizable and N is the number of cores.

But this should be taken with a grain of salt due to in a real environment other processes are able to make use of extra cores in case of a non parallelizable task.

A simple way to measure required speed is Little's Law which is

$$L = \lambda W$$

Where L is the average number of unit in the system at any time, λ is average rate of items which arrive per unit time and W is a number of unit time.

2.1 Measuring performance

Clock speed are a way of measuring the speed of electric pulses in the CPU measured in Hz, The time between a clock tick is called cycle time.

Average cycle per instruction (CPI) is the average weighted number of cycles needed for every available instruction.

$$CPI = \frac{\sum_{i=0}^n (CPI_i \cdot I_i)}{I_c}$$

Where CPI_i is the number of operation for the instruction i and I_i is the number of the instruction. I_c is the total number of operations.

With this the process time can be calculated in two ways

$$T = I_c \cdot CPI \cdot \tau$$

$$T = I_C \times [p + (m \times l)] \times \tau$$

I_C is instruction count, $\tau = 1/f$ where f is clock frequency, p number of processor cycles for decode and execute, m number of memory references, k ratio between memory cycle time and processor cycle time.

Often the millions of instruction per second (MIPS) is used which can be found with:

$$MIPS = \frac{f}{CPI \times 10^6} = \frac{I_c}{T \cdot 10^6}$$

Which also can be found in variations with floating point operations (MFLOPS)

This is a flawed measurement due to different architectures like RISC and CISC where RISC will always have an advantage due to the reduced instruction set.

A good benchmark should be:

- Written in high level language to make portability high
- Is representative of a kind of programming domain
- Easily measured
- Wide distribution

SPEC is a standard for benchmarking which uses these terms:

- Benchmark - program written in high level and able to compile and execute on every computer which implements the compiler
- System under test - the tested computer system
- Reference machine - the reference scores from a chosen machine to compare current result to
- Base metric - strict guidelines for compilation in order to be able to compare results
- Peak metric - Optimized settings for the given system

- Speed metric - The total time of execute a compiled benchmark
- Rate metric - the number of tasks which can be completed in a given amount of time

3 Digital logic

3.1 Boolean algebra

Boolean algebra are algebra based on only the values 1 or 0.

It consist of variables and the operations AND (\cdot), OR ($+$) and NOT (\bar{b}) in that precedence.

Another often usefull operators are XOR (\oplus), NAND ($\overline{b \cdot a}$) and NOR ($\overline{a + b}$) Set operation may also be performed on sets of boolean, where union is or, intersect is and. When applies the operation is performed on each bit one by one.

And then the universal set will just be a set of 0's.

For more info, checkout my logical proposition repo.

These gates only have two output and one output except the NOT gate, but any number of inputs is possible and some gates may have two outputs where one is negated.

When designing af circuit the fewer amount of gates the simpler and to complete possible operation the following set combinations are possible:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR

When writing circuit, it can be done in two forms, sum of products, where product expression are multiplied, and product of sums (POS) where sum are multiplied.

Both forms may not be the most simple form, but SOP uses only NAND, NOT and OR gates and POS uses only OR, AND, and NOT.

To simplify a circuit there are different methods

- Algebraic simplifications - This can be done with indentities, which can simplify the expressions


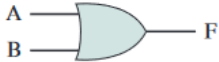
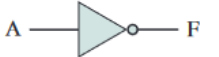

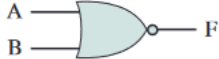

Name	Graphical Symbol	Algebraic Function	Truth Table															
AND		$F = A \cdot B$ or $F = AB$	<table><tr><th>\overline{A}</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	\overline{A}	B	F	0	0	0	0	1	0	1	0	0	1	1	1
\overline{A}	B	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = A + B$	<table><tr><th>\overline{A}</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	\overline{A}	B	F	0	0	0	0	1	1	1	0	1	1	1	1
\overline{A}	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$F = \overline{A}$ or $F = A'$	<table><tr><th>\overline{A}</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	\overline{A}	F	0	1	1	0									
\overline{A}	F																	
0	1																	
1	0																	
NAND		$F = \overline{AB}$	<table><tr><th>\overline{A}</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	\overline{A}	B	F	0	0	1	0	1	1	1	0	1	1	1	0
\overline{A}	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = \overline{A + B}$	<table><tr><th>\overline{A}</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	\overline{A}	B	F	0	0	1	0	1	0	1	0	0	1	1	0
\overline{A}	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$F = A \oplus B$	<table><tr><th>\overline{A}</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	\overline{A}	B	F	0	0	0	0	1	1	1	0	1	1	1	0
\overline{A}	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

Figure 1: Figure of gates and their logical operation

- Karnaugh Maps - k-maps are a method which can help simplifying which variables are the out depended upon

3.2 Karnaugh maps

This method works by taking 2 to 4 variables from a truthstable or function. They are then setup in a grid such all possibilities are accounted for.

So for one side describing one variable there are 2 possibilites and a row which describes two variables there will be 4 possible outcomes.

For each row/column combination the function or truthtable are used to determine if the cell is 0 or 1.

Afterwards each 1 is circled in groups of powers of 2, so 1,2,4,8 or so on. A circle can not be cross or contain 0.

For each circle the depending non changing variables are used in an and form and may be negated if the input was a consisten 0.

For each circle the found AND expression is added with or to eachoter.

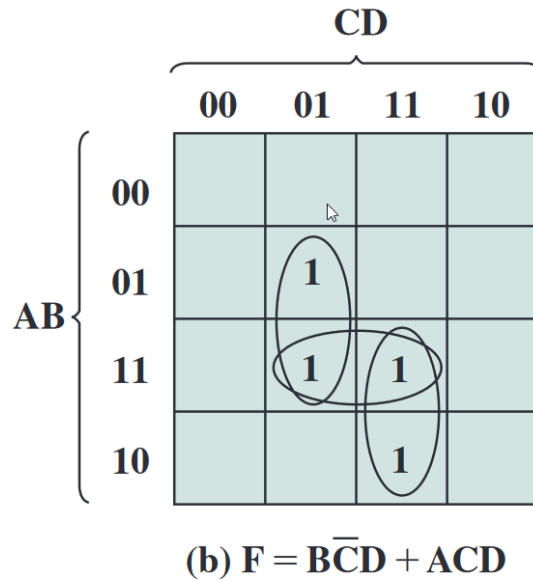


Figure 2: Example of kmap and the output function

3.3 Quine-McCluskey method

This is a more suitable method for SOP which have more than 4 variables. The method works by first creating a table with each collection of products on each row and in every column is the variables and in each cell are the needed value for the term to turn true.

The table is then ordered such the row with most 0's is at the top at the row with most 1's are at the bottom.

Then every row is compared to every row starting at the top, and if a row exist with only one column difference, the difference variable is eliminated and the rest of the variables are added to a new list.

Then every element in the list is done with same procedure, and new found objects are added to the list. Then the same step is repeated with every new element until there is no new elements.

Then every element from the list is added to a table as rows and the original terms are added as columns.

Then an X is placed in every cell where the row product is contained in the column. Then circle every X which is alone in a column, and square every X which are in a row with a circle.

Those rows with a marked X are now needed for the minimal expression.

Product Term	Index	A	B	C	D	
$\bar{A} B \bar{C} D$	1	0	0	0	1	✓
$\bar{A} B C \bar{D}$	5	0	1	0	1	✓
$\bar{A} B C D$	6	0	1	1	0	✓
$\bar{A} B \bar{C} \bar{D}$	12	1	1	0	0	✓
$\bar{A} \bar{B} C D$	7	0	1	1	1	✓
$\bar{A} \bar{B} C \bar{D}$	11	1	0	1	1	✓
$\bar{A} \bar{B} \bar{C} D$	13	1	1	0	1	✓
$\bar{A} \bar{B} \bar{C} \bar{D}$	15	1	1	1	1	✓

$\bar{A} \bar{C} D$ $\bar{A} B D$ $\bar{A} B C$ $\bar{A} B \bar{C}$ $B \bar{C} D$ $A \bar{C} D$ $A B \bar{C}$ $B \bar{C} \bar{D}$

Product Term	A	B	C	D	
$\bar{A} \bar{C} D$	0		0	1	
$\bar{A} B D$	0	1		1	✓
$\bar{A} B C$	0	1	1		
$\bar{A} B \bar{C}$	1	1	0		
$B \bar{C} D$		1	1	1	✓
$A \bar{C} D$	1		1	1	
$A B \bar{C}$	1	1		1	✓
$B \bar{C} \bar{D}$		1	1	1	✓

$B \bar{D}$ $B \bar{C} D$ $B \bar{C} \bar{D}$

	$A B C D$	$A B \bar{C} D$	$A B \bar{C} \bar{D}$	$A \bar{B} C D$	$\bar{A} B C D$	$\bar{A} B \bar{C} D$	$\bar{A} B \bar{C} \bar{D}$	$\bar{A} \bar{B} C D$
$B \bar{D}$	X	X			X		X	
$\bar{A} \bar{C} D$							X	⊗
$\bar{A} B C$					X	⊗		
$A B \bar{C}$		X	⊗					
$A \bar{C} D$	X			⊗				

Figure 3: Example of the method on

$$F = A B C D + A B \bar{C} D + A B \bar{C} \bar{D} + \bar{A} B C D + \bar{A} B \bar{C} D + \bar{A} B \bar{C} \bar{D} + \bar{A} \bar{B} C D + \bar{A} \bar{B} \bar{C} D$$

Which result in the list $F = A B \bar{C} + A \bar{C} D + \bar{A} B C + \bar{A} \bar{C} D$

3.4 Circuits

3.4.1 Multiplex

Multiplex is a circuit of which a number of inputs label D_0, D_1, \dots, D_N is wired to an output F .

To controle which input determine the F value, the required number of selection inputs are used called S_1, S_2, \dots, S_N .

So for a 4 input it would require two S inputs.

3.4.2 Decoders and encoders

A decoder is a circuit with a number of output lines, with only one asserted at the time.

In general a decoder has n input and 2^n outputs and can be useful for writing a specific sequence of bits according to a simple code.

An encoder will then be the inverse of the decoder.

3.4.3 Read-only Memory

As in the name this is memory, which can only be read from and are not programmable.

This is implemented using a decoder and a set of OR gates.

This is done by the a number inputs representing the placement of data and the OR gates each give out the value at the address.

3.4.4 Sequential circuits

Unlike combinational circuits as above a sequential circuits outputs are depending on current inputs and the current state.

The most simple form is a flip-flop, which is able to store one bit of data.

There are different types of flip flops with different properties The SR flip flop can not have both S and R be 1 but is the most simple, with a on (S) and off (R)

The D flip flop has a single switch for both on and off

The JK flip flop can have both inputs be 1 and will simply result in Q being 0

These flip flops can then be made in parallel forming a register, or by as a shift register which has flip flops in series and are sending data down the series at each clock cycle with only input at the front.

Another use case is a series of flip flops which creates a ripple counter or asynchronous counter, which when incremented the effect ripples through all the other flip flops.

Synchronous counters have the clock going into every flip flop. It can be observed that when counting in binary the first bit, simply flips on every count, the next bits flip when the right bit is 1.

3.4.5 Programmable logic devices

PLD are general-purpose chips.

There are different types of PLD

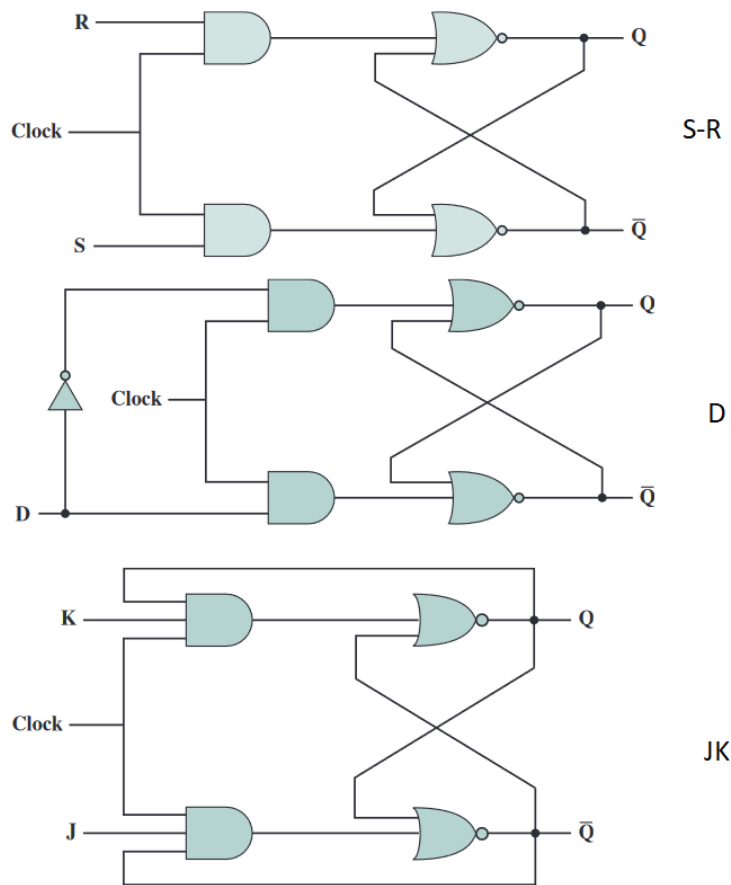


Figure 4: Different types of flip flops

- PLA - Programmable logic array, is circuit which allows a number of inputs in both normal and negated form, which goes into an array of and gates, which output are wired up to an OR array into the outputs. This takes advantage of SOP binary form
- FPGA - Field programmable gate array, is a circuit consisting of a logic block, which are programmable using flip flops, I/O blocks and interconnect which connects the I/O block to the internal logic blocks

4 Instruction sets

4.1 Machine instructions

A machine instruction consist of the following:

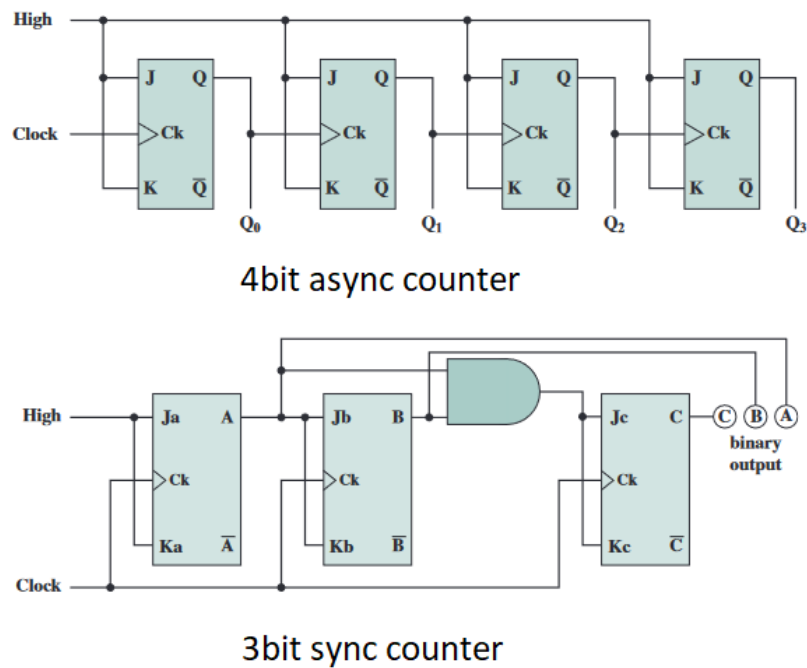


Figure 5: Two implmentations of binary counters

- Operation code - the code which describe the operation to be performed called opcode
- Source operand reference - The operation may one or more source reference for the operation
- Result operand reference - The reference to the result of the operation if it exist
- Next instruction reference - The reference which hold the next instruction for after the execution

The next instruction reference and result reference can reference memory in different sources:

- Main or virtual memory
- Processor register - in some cases one or more registers may contain memory addresses which can be reference by the register name
- Immediate - The reference may be contained in the current instruction
- I/O device

When referencing instruction the opcode is most often represented as abbreviation called mnemonics, such as ADD

Instructions can be of the following types

- Data processing - Arithmetic and logic instructions
- Data storage - movement of data from and to registers and memory locations
- Data movement - I/O instructions
- Control - Test and branch instructions

Instructions may be designed to use

- 0 references - This will then use the stack for references
- 1 reference - Performs the instruction on and saves in a common accumulator register or something alike which the instruction is used upon
- 2 references - Performs the instruction and saves it in the first register
- 3 references - Performs instruction and first two references and saves in the last reference

When designing a set of instruction there are multiple questions have to be considered

- Operation repertoire - How many and which operations should be in the set
- Data types - Which data types should be available
- Instruction format - Instruction length number of addresses, size of various fields and so on
- Registers - Number of registers which should be referenceable and what their use should be
- Addressing - In which mode an address of an operand is specified

4.2 Types of operands

For numbers there are 3 different types

- Binary integer or binary fixed point - The classic integer in binary form
- Binary floating point - here the first bit mean the sign(1 = negative) the next 8 bits are the exponent and the next 23 are the mantisse for the 32 bit version
- Packed decimal - used to avoid a lot of conversion, such every decimal is represented with 4 bits, and (1101) means - and (1100) means +

For representing characters 8 binary bits can be used with standards by ASCII, which dictates what the different binary combination represent.

The x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quadword), and 128 (double quadword)

ARM processors support data types of 8 (byte), 16 (halfword), and 32 (word) bits in length.

4.3 Types of operations

A useful and typical categorization is the following:

- Data transfer - Calculate the memory address based on address mode, if virtual translate to real memory, determine if data is not cached and issue a command to the memory module.
- Arithmetic - Different kind of mathematic operations and may include data movement for the operation
- Logical - Logical operations such as XOR, right shift, left shift or rotate on binary data
- Conversion - Conversion between binary and decimal as well as operation conversion between 8 bit and such
- I/O - Instructions for data movement in and out of the system
- System control - Privilege function often reserved to operation system, such as alter register control or modifying storage protection key.

- Transfer of control - Operations for changing execution with: branching on condition (if and loops), skip on condition (skip out loop or flow), call a block of code and return to current code (function calling) is done by calling it and pushing parameters and return to stack in a stack frame.

When using conditions it refers to one of the architectures flags, which may be raised during instructions.

For x86 the call specifically does

- Push the return point on the stack
- Push the current frame pointer on the stack
- Copy the stack pointer as the new value of the frame pointer
- Adjust the stack pointer to allocate a frame

This can be done manually by instructions or by the ENTER instruction though it takes 10 cycles instead of 6.

MMX instructions are also exclusive to x86 and are instruction which can operate on multiple smaller data set by combining them into 32 or 64 bit chunks.

This allows the instruction to work in parallel on things like image processing where pixels are gathered in larger chunks.

5 Addressing modes and Formats

5.1 Addressing modes

Addressing modes are different methods of getting an address which can be send to the accumulator.

Most often a system implements two modes and the selected method is either in the opcode or in the memory field.

- Immediate - The address is in the instruction, needs little memory but has size of address is limited to operand
- Direct - The instruction contains a memory location which value is the address, memory location is limited to operand size.

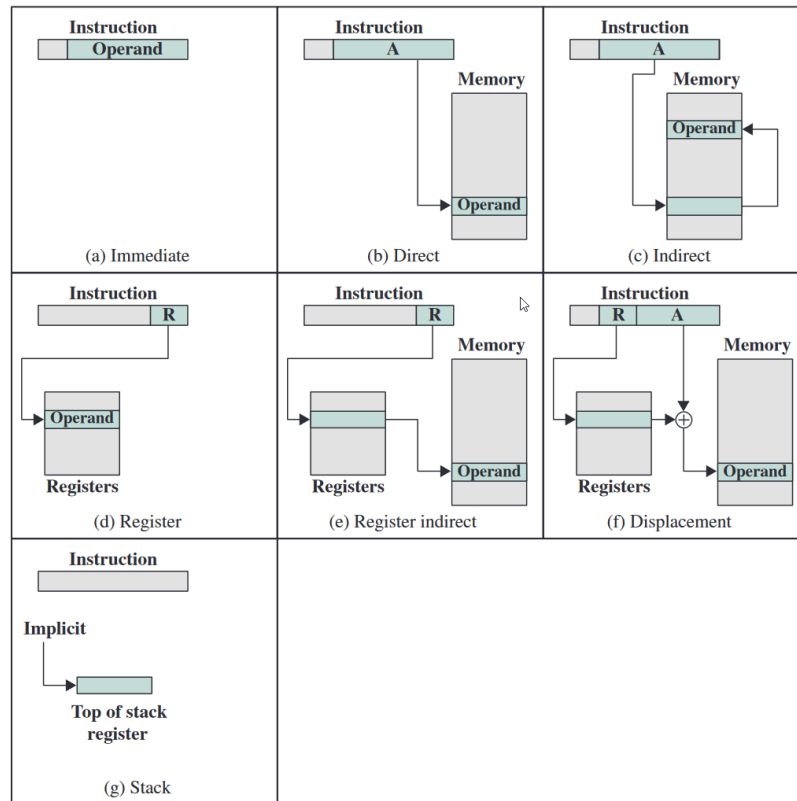


Figure 6: Illustrations of addressing modes

- Indirect - The instruction contains memory locations which contains memory location of which value is the address, first operand location is then low such it can contain a higher memory location
- Register - Same as direct but the operand refers to a register instead, registers are faster in case of reuse and requires a smaller address
- Register indirect - Same as indirect but with registers
- Displacement - Operands are both a register address and a value which is added to the register value to find memory location containing address.
- Stack - Instead of instruction including a memory reference the stack is used to operate

Displacement can be used in different ways

- Relative - The register used is the program counter, such the memory location is relative to the current with the offset of the other operand.

- Base register - Uses the base registers value added with the other operand
- Indexing - Register contains offset and other operand is memory location, often used for loops, and autoindex may be enabled where it automatically in a cycle increment.

For autoindex, the new index is saved in the memory location, this is either done before preindexing or after the indirection it is postindexing.

5.2 Addressing in x86

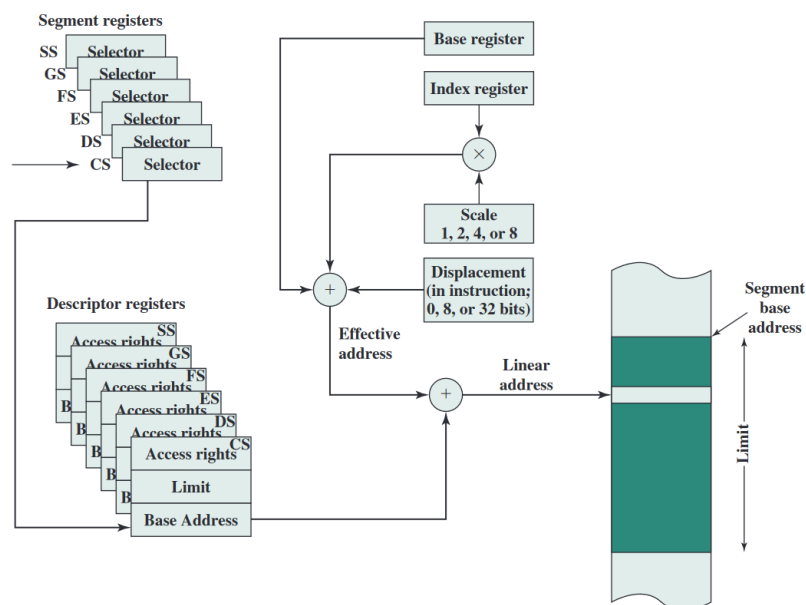


Figure 7: Addressing modes in x86 and how it is calculated

In x86 there are segment registers, these registers hold an index of the descriptor registers.

The descriptor registers hold 3 values: Access right to the data, how much data there is and where the first part of the data is located.

In addition there is a base register and index register.

For register operand mode - either a 32 bit register, 16 bit register or 8 bit register can be used for data transfer, arithmetic and logical instructions.

Displacement mode are not often used due to the long length up to 32 bits and are mostly used for global variables.

For the rest of the addressing modes the memory location is referenced by

the segment and the offset in the segment.

Displacement mode with base is used for local variables, index of arrays and large record pointers.

5.3 Addressing in ARM

On arm there are three alternatives to indexing due to no index register being a thing.

Offset - The instruction holds the offset as well as the register with the address.

Preindex and postindex are offset but with the writing to the register either pre or post indexing.

5.4 Instruction formats

The length of an instruction is determined by a lot of factors, the longer the easier to program but more space.

The length should be equal to memory-transfer or a multiple the length to ensure integral number during a fetch cycle.

The length should also be optimized to be shorter due to memory most often being a bottleneck in speed.

The instruction length should also be a multiple of 8 due to the character length, and how that relates to the word length such a word contain an integral number of characters.

The allocation of bits is also determined by multiple factors

The more opcodes the more readable code and often less code, but some opcodes may be determined by the operands also.

The amount of registers also affect the amount of bits required to describe a register. The ideal amount is found to be between 8 and 32.

These registers can also be in sets such they can be determined with smaller amount of bits ex 2 sets of 8 requires 3 bits of data.

The operands also need a good amount of data for addresses, not directly addresses but rather a big range for the displacement.

Variable length instruction are variable length in the bit allocation in instruction which solves some of the many problems but requires more complexity in the processor and multiple instruction may be fetched at once.

5.4.1 x86 instruction format

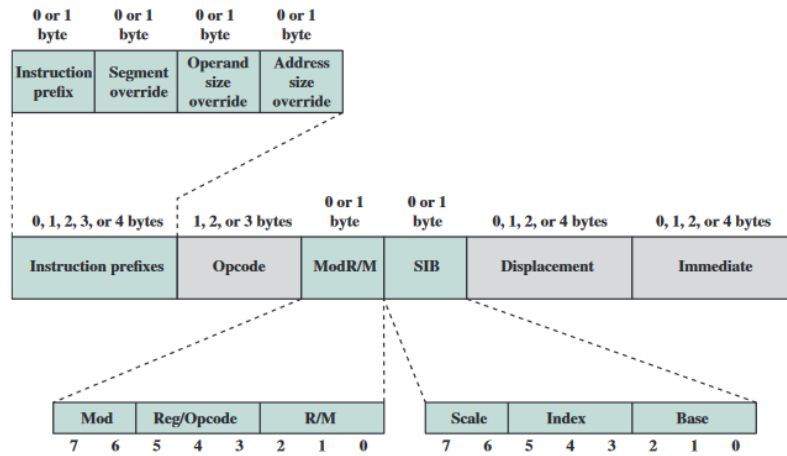


Figure 8: x86 instruction format and it lengths

- **Prefixes**
- Instruction prefixes - There are two available prefixes, LOCK which ensures exclusive use of shared memory in multiprocessor environments, and repeat prefixes which can be one of five
 - REP - repeats instruction until register CX is equal to zero
 - REPE/REPNE - repeats until value of ZF flag
 - REPZ/REPNZ - repeats until RCX, ECX or CV is equal 0
- Segment override - Overrides which segment register an instruction should use
- Operand size - Switches between 16 or 32 bits operands
- Address size - Switches between 16 or 32 bit address generation
- **Instruction**
- Opcode - 1 - 3 bytes of length may also include bits about: if data is byte- or fullsize, direction of data operation, immediate data field must be sign extended
- ModR/M - The location of the first operand (address mode or register) and the location of the second operand (a register) if required by the instruction. Or an extra bit for the opcode (opcode extension)

- SIB - If a specified address mode need more data the SIB contain: The scale field for scaled index (2 bits), index field (3 bits) for index register and base field (3 bits) for base register.
- Displacement - If displacement addressing mode is used, an 8-, 16-, or 32-bit signed integer displacement is specified
- Immediate - Provides the value of an 8-, 16-, or 32-bit operand

5.4.2 ARM instruction format

	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
Data processing immediate shift	cond	0	0	0	opcode				S	Rn				Rd				shift amount				shift	0	Rm								
Data processing register shift	cond	0	0	0	opcode				S	Rn				Rd				Rs				0	shift	1	Rm							
Data processing immediate	cond	0	0	1	opcode				S	Rn				Rd				rotate				immediate										
Load/store immediate offset	cond	0	1	0	P	U	B	W	L	Rn				Rd				immediate														
Load/store register offset	cond	0	1	1	P	U	B	W	L	Rn				Rd				shift amount				shift	0	Rm								
Load/store multiple	cond	1	0	0	P	U	S	W	L	Rn				register list																		
Branch/branch with link	cond	1	0	1	L	24-bit offset																										

- S = For data processing instructions, signifies that the instruction updates the condition codes
- S = For load/store multiple instructions, signifies whether instruction execution is restricted to supervisor mode
- P, U, W = bits that distinguish among different types of addressing mode
- B = Distinguishes between an unsigned byte (B==1) and a word (B==0) access
- L = For load/store instructions, distinguishes between a Load (L==1) and a Store (L==0)
- L = For branch instructions, determines whether a return address is stored in the link register

Figure 9: ARM instruction formats

- Immediate constants - 8 bit constant value which can be rotated by 4 bit to create constants
- Thumb instruction set - A subset of instruction which have been decreased to 16 bit instead of 32 bit. This is done by
 - Thumb instruction are unconditional, so the conditional field is not used and all arithmetic thumbs update the conditions flag, so not flag bits are needed
 - The limited amount of opcodes only require 2 bits and 3 bit type field
 - Thumb instructions only references registers r0 to r7 so only 3 bit is required

- Thumb-2 instruction set - This is the bridge between ARM and thumb instruction which makes it possible to combine both instruction for more compact and performed code

6 Assembly language concepts

Assembler - compiler for assembly to object code

Linker - Combines one or more files containing object code into a single loadable file or executable code

Loader - Copies an executable into memory for execution

Object code - Step between assembly and executable code Symbolic program

- A static somewhat like assembly language with static memory

Assembly uses symbolic addresses for data for non static references.

The downsides of writing in assembly instead of high level language

- development time takes much longer
- Reliability and security are lower due to no compiler which can warn or throw errors of bad or unsecure code
- Debugging and verifying take longer due to more code resulting in more places it can go wrong
- Maintainability is low due to the often spaghetti like structure of assembly
- Portability is only on same platform
- HLL language have access to use intrinsic functions so assembly is not required for device drivers and other system code
- Compilers are so good now that it is often harder to write better assembly than the compiler

But there are upsides

- A compiled assembly code can be verified
- To create a compiler assembly is required
- Embedded systems may not be able to have a compiler and therefore need all code in assembly

- Hardware drivers and system code are often easier to program with assembly due to hll's not having access to hardware or registers or so on
- Accessing instructions that are not accessible from hhl
- Code size are often smaller with self written assembly
- Writing in assembly makes it possible to optimize more in speed than a compilers general optimization

6.1 Assembly language elements

An assembly statement consist of: label, mnemonic, operand and comment

- Label (Optional) - Equivalent to address used for code references
- Mnemonic - Name of operand or function , symbolic name representing an opcode
- Operand(s) - zero or more operands may be given representing either a immediate value, a register value or a memory location
- Comment (Optional) - Text which are ignored by help the programmer in x86 it starts with a semicolon

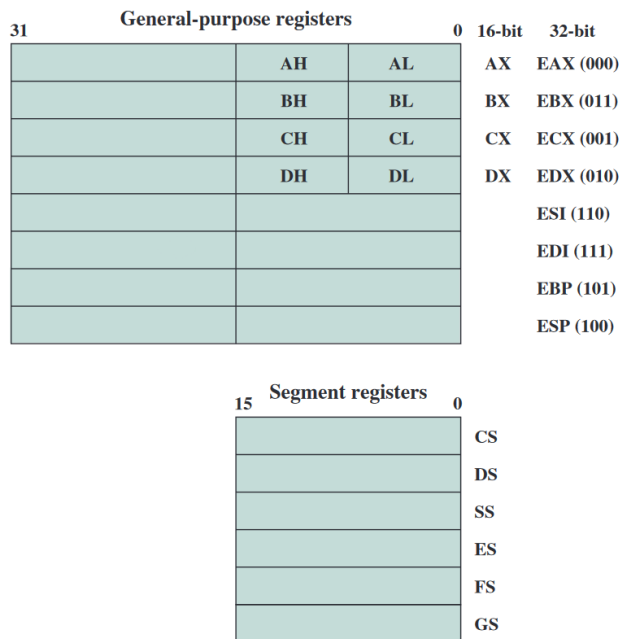
When referencing data in operands there are different methods as discussed. For register addresses the register name is used. For immediate addressing uses an indication that the value is encoded. Examples are H for hexadecimal, B for binary and decimal has no suffix

Ex. 100B is read as a binary value

For direct addressing expressed as a dispalcement from the DS segment.

6.1.1 Pseudo-instructions

Pseudo instruction are not x86 machine instruction but are still placed in the instruction field.



General Purpose Registers - Conventional Use

EAX	Accumulator
EBX	Used to address memory or hold data
ECX	Used as a Counter
EDX	Acts as Accumulator's backup
ESI	Instruction source pointer
EDI	Instruction destination pointer
EBP	Stack base pointer
ESP	Used only as a Stack pointer

Figure 10: x86 register names and their uses

6.1.2 Macro definitions

Macros are a subroutine of code, and can be used multiple times like a call instruction.

The main difference is macro expansion which actually inserts the macro at every instance to get rid of overhead time of instruction switching.

This will result in a larger code but more clean code.

Therefore a macro should only be a small bit of code.

Macros can both be defined on single lines as - `%DEFINE A(X) = 1 + 8 + X` and then be referred to as - `MOV AX, A(8)`

For a multiline macro

`%MACRO jNAMEj jnumber of paramsj`

Unit	Letter
byte	B
word (2 bytes)	W
double word (4 bytes)	D
quad word (8 bytes)	Q
ten bytes	T

Name	Description	Example
DB, DW, DD, DQ, DT	Initialize locations	L6 DD 1A92H ;doubleword at L6 initialized to 1A92H
RESB, RESW, RESD, RESQ, REST	Reserve uninitialized locations	BUFFER RESB 64 ;reserve 64 bytes starting at BUFFER
INCBIN	Include binary file in output	INCBIN "file.dat" ; include this file
EQU	Define a symbol to a given constant value	MSGLEN EQU 25 ;the constant MSGLEN equals decimal 25
TIMES	Repeat instruction multiple times	ZEROBUF TIMES 64 DB 0 ;initialize 64-byte buffer to all zeros

Figure 11: Directives unit and letter and what the different pseudo instructions do

```
PUSH EBP;  
SUB EBP, %1 ; first parameter  
%ENDMACRO
```

6.2 Types of assemblers

- Cross-assembler - Runs on another computer host to then be transferred to the target machine
- Resident assembler - Host and target are the same
- Macroassembler - Allows the user to define sequences of instructions as macros
- Microassembler - Used to write microprograms which define the instruction set for a microprogrammed computer
- Meta-assembler - Can handle multiple instruction sets
- One-pass assembler - Produces machine code from a single pass of the assembly code
- Two-pass assembler - Makes two passes to produce machine code

Two pass works by first finding and creating a symbol table of all symbols and their given location counter (LC).

In the second pass the following is done

- Translate the mnemonic into binary opcode
- Use opcode to analyze the instruction
- Translate each operand to appropriate register or memory code
- Translate immediate value to binary
- Translate labels reference to LC
- Set any other bit given in the instruction

The assembler also has a zeroth pass where it reads all macros which are defined at the top, such it can expand on first pass.

For a one pass compiler the trouble is keeping reference while translating which is done by, when a new label is found the following is done

- Leaves the instruction operand field empty in the assembled binary instruction
- The symbol used as an operand is entered in the symbol table and flagged as undefined
- The address of the operand field is added to a list of forward references associated with the symbol table entry

6.3 Loading and linking

Linker - Finds references to other code or data and links the references between object code (non linked machine code)

A linker which produces a single module from the main module and its references to other modules is called the linkage editor

Load-time dynamic linking keeps all references to external modules and in run time when the external module is used it is loaded into main memory and the reference is updated.

This also makes the libraries updatable and be in files themselves, ex in windows they are in dynamical-link libraries (DLLs), but this can also lead to DLL hell where two executables expect two different versions of the same DLL file.

Loader - loads the final machine code into main memory

There are 3 types of loading

- Absolute loading - Requires all modules are in the same location in memory, and references are absolute, this has many disadvantages with everything having to be in main memory at once and the absolute nature makes it near impossible to make correct references
- Relocatable loading - Every reference is relative to some point in the program, such that point location is just added to every other reference
- Dynamic run-time loading - To ensure that the program has not been moved since first loading, the reference is first calculated with the offset when the reference is needed

7 Assembly language x86

A list of every instruction can be found [here](#)

The structure using `at&t` is mnemonic source, destination.

The registers start with % and literal values start with \$

For comments # is used.

For 64 bit instruction, the opcode ends in q

7.1 Directives

Directives are part of code to initialize data or write the code.

There are 3 types of sections .text, .bss and .data

.data is for storing data and .bss is for allocating data.

.text is for the code and global variables.

Other forms of directives are for storing or allocating data.

This can be done by: .space *numBytes*; (Byte 8, Word 16, Doubleword 32, Quadword 64, Double quadword 128)

Or with text and label: myString: .ascii "Hello World!"

7.2 Starting the code

The assembler looks for the start of the code by:

```
01 | section .text
02 |     global _start
03 | _start:
04 |     ...
```

7.3 Registers

[h!] From the existing registers the same space is actually used.

The different registers and their space can be seen in the figure.

In some instructions two registers may be used for wider range indicated by reg1:reg2 ex: RDX:RAX

- RAX - accumulator for arithmetic operations
- RBX - Base, holding a pointer to data
- RCX - Counter used in shift/rotate instructions and loops
- RDX - Data, used in arithmetic operations and I/O operations
- RSI - Source index, a pointer to data source in stream operations
- RDI - Destination index, a pointer to data destination

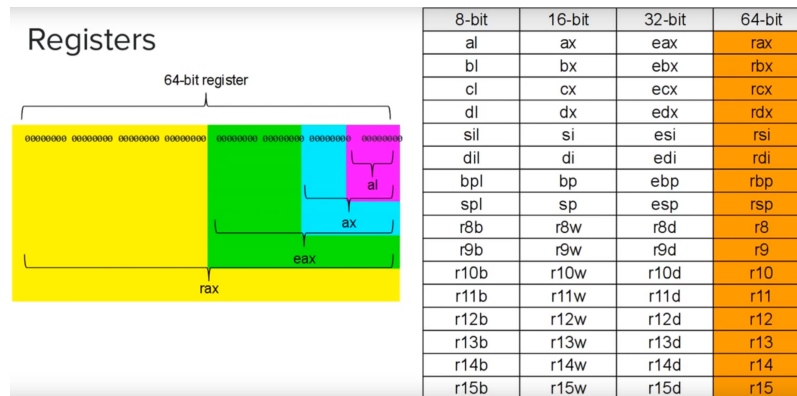


Figure 12: The different registers in x86 assembly

- R8-R15 - New registers in 64 bit
- RSP - Stack pointer, points to top of stack
- RBP - Strack frame base, pointer to base of current stack frame
- RIP - Instruction pointer

7.4 System calls

To make system calls the call and its arguments are putted into the following registers.

- ID - rax
- 1 - rdi
- 2 - rsi
- 3 - rdx
- 4 - r10
- 5 - r8
- 6 - r9

Every system call and their ids can be found [here](#).

When every value is set the instruction 'syscall' can be called.

To end a program a `exit` system call is made with argument 0 symbolising the error code.

```
01 | section .text
02 |     global _start
03 | _start:
04 |     movq $60, \%rax \#exit id is 60
05 |     movq $0, \%rdi
06 |     syscall
```

7.5 Jumps

First the instruction `cmpq jarg1, jarg2` is ran to set the flag. Then one of the following condition jumps can be made

- `je: arg1 == arg2`
- `jne: arg1 != arg2`
- `jg: arg1 < arg2`
- `jge: arg1 <= arg2`
- `jl: arg1 > arg2`
- `jle: arg1 >= arg2`

7.6 Stack and function calls

The stack goes from higher address to lower address.

So when a new thing is pushed on the stack the RSP is decreased by 8 for each stack.

When a function is called the stack is used in the following order to create a stack frame.

- Argument in reverse order
- Current instruction pointer
- Local variables created in function

The called function is responsible for popping off the local variables and instruction pointer.

The calling function has to pop the arguments.

For first 6 integer/pointer arguments are in registers
RDI, RSI, RDX, RCX, R8, and R9
If more arguments are needed to stack is used.
RAX is used for return value.
For saving old register values they may be pushed to stack before beginning
to function call.
To call a function: `call jlabelj`
And to return back from call: `ret`

8 Computer Arithmetic

The core of the computer is the arithmetic unit (ALU)
Representing positive or negative a fixed number a bits is used to represent
the number and the right most bit is a sign bit.
This causes some drawbacks, mainly arithmetics are not as simple and check-
ing for zero is also not as easy.

8.1 Sign-magnitude representation

The left most bit represent the sign if 1 then it is negative if 0 then positive.
Drawbacks are bad arithmetic and hard to check for zero.

8.2 Twos compliment

Twos compliment counter act this by positive must the first bit be 0.
Twos compliment of size n is defined such if the most significant bit is 1, 2^n
is subtracted to make it more simple.
When negating twos compliment the most significant digit is flipped.
The two edge cases 0 negation and the largest number negation, here a carry
has to be remembered an used.

8.2.1 Twos compliment arithmetic

Doing addition is the same as binary, but the two most significant bits are
not added rather XOR'ed
For subtraction a negation is done and then an addition
The addition is therefore done by a single addition unit taking two registers,
saving the result in one of them or a third register, and indicate a possible

overflow in a flag.

8.2.2 Twos compliment multiplication

For multiplication $n \times m$ bits, the n bits are run through for every iteration a shift right of m is done. If the bit iteration in n is 1, m is added to the summation. For multiplication of twos compliment, the multiplicand is in register M , multiplier in Q and two extra registers are needed A with start value of 0, and a single bit register we call Q_{-1} to represent it being to the right of iteration of Q .

Then Q is iterated through and looking at both Q_i and Q_{i-1} .

If the two bits differ then, if Q_{i-1} is 1 then M is added to A , and if Q_{i-0} is 0 then M is subtracted from A .

If the two bits are the same, or an addition/subtraction is done, then A and Q is cyclic shifted to the right as one unit, and Q_{-1} is what would have shifted into -1

This is repeated n times where n is the length of the multiplier.

The result will be AQ .

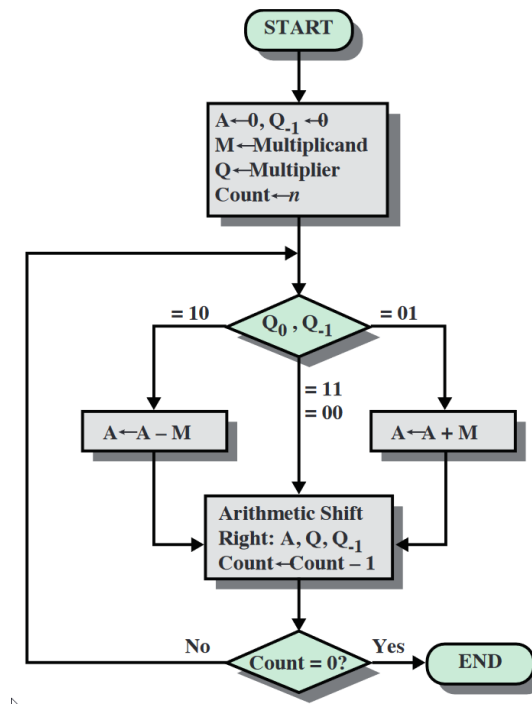


Figure 13: Twos complement multiplication diagram

8.2.3 Twos compliment division

For division it differs not far, where M is the divisor, Q is the dividend and a summation register A with initial value 0.

First A and Q is cyclic shifted left as one unit

Then M is subtracted to A .

If A is larger than zero (rightmost bit is 0) then set $Q_0 = 1$ otherwise set $Q_0 = 0$ and add M to A .

The remainder will be in A and the quotient is in Q .

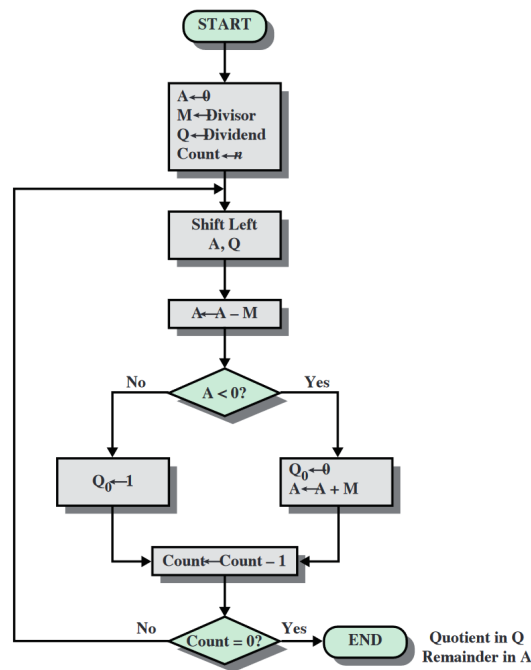


Figure 14: Twos complement division diagram

8.3 Floating-point representation

For a typical 32 bit format of a floating point will be

1 bit for sign,

8 bits for exponent,

23 bits for mantissa

$sign \cdot mantissa \cdot 2^{exponent}$

For 64 the exponent is 11, and 128 15bits

The exponent has a default bias being the negative value of half the range.

This giving the values: It can here be seen that the density of representable

Parameter	Format		
	Binary32	Binary64	Binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	-126	-1022	-16382
Approx normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16382}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Figure 15: Limits for the different floating point formats

numbers is lower on higher numbers.

When working with floating-point arithmetic one of the following may happen

- Exponent overflow
- Exponent underflow ending up being reported as 0
- Significand underflow/overflow ending up rounding such nothing really is changing

Subtraction and addition can be split into four groups

- Check for zeros - Change sign if subtraction, and check if one is equal to 0 such the result can be returned
- Align the significands - The lower number is shifted to the right and exponent is incremented, such the least significant bits are may lost
- Add or subtract the signifands - If overflow/underflow occur the exponent is either incremented or decremented
- Normalize the result - left shift until the left most digit is not zero and decrementing the exponent

Multiplication

- Check for zeros
- Exponents are added together

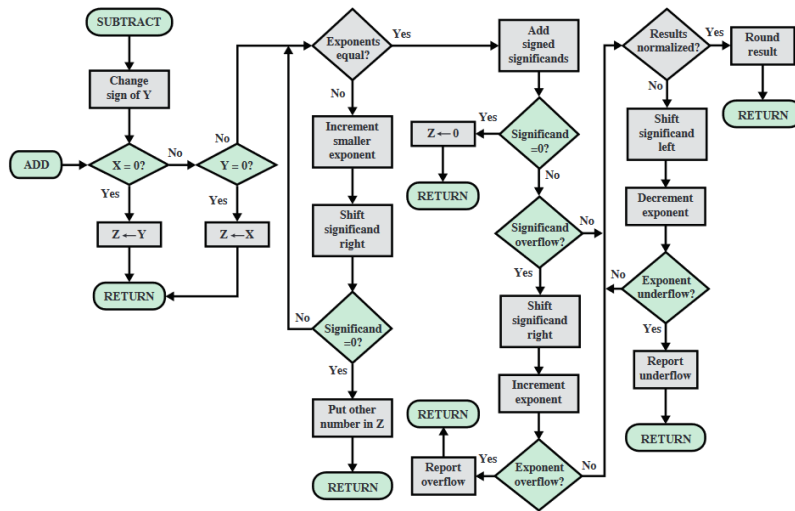


Figure 16: Floating point addition and subtraction diagram

- Significands are multiplied as normal integers
- Normalize the result

Pretty much the same for division, but instead exponents are subtracted and significands are divided as normal integers.

Guard bits are extra zeros added to the end of the significand, such at left shift there will be less lost of significant bits.

When the result is put into the given format rounding may be needed, there are 4 different approaches

- Round to nearest representable number, does require some extra statement for the exact between of two representable numbers, standard is towards even
- Round towards $+\infty$
- Round towards $-\infty$
- Round towards 0

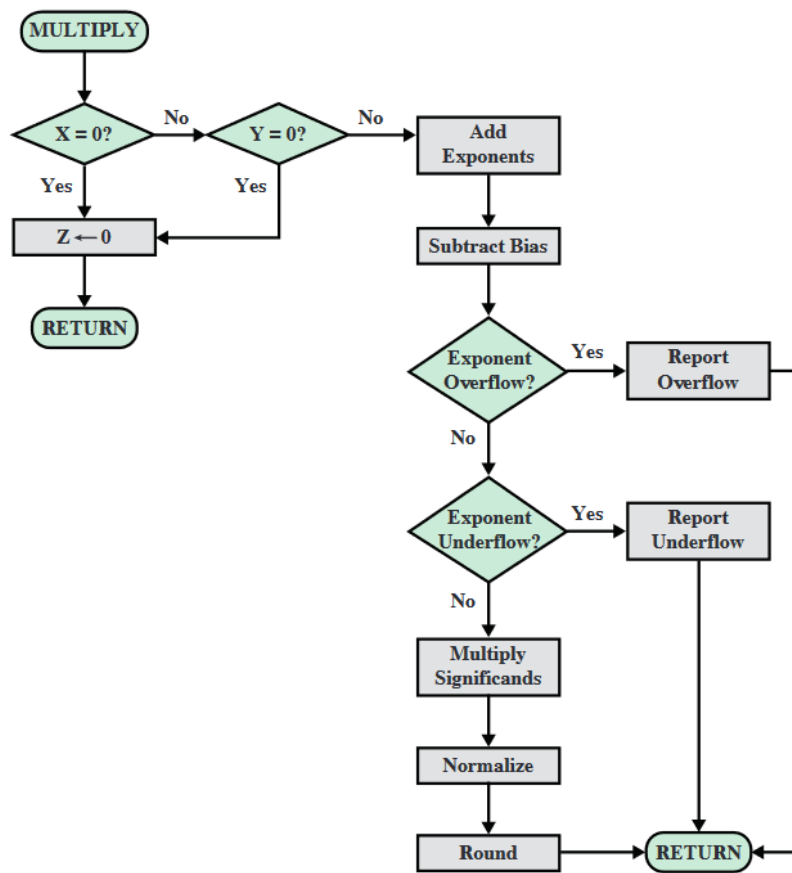


Figure 17: Floating point multiplication diagram