

Computer Architecture and system programming

Kristoffer Klokke

2022

Contents

1	The basics	6
1.1	Structure and Function	6
1.2	Gates, memory cells, Chips, and Multichip modules	7
1.3	Processor architecture	8
1.4	Embedded systems	8
2	Performance	8
2.1	Measuring performance	9
3	Digital logic	11
3.1	Boolean algebra	11
3.2	Karnaugh maps	12
3.3	Quine-McCluskey method	13
3.4	Circuits	14
3.4.1	Multiplex	14
3.4.2	Decoders and encoders	15
3.4.3	Read-only Memory	15
3.4.4	Sequential circuits	15
3.4.5	Programmable logic devices	15
4	Instruction sets	16
4.1	Machine instructions	16
4.2	Types of operands	19
4.3	Types of operations	19
5	Addressing modes and Formats	20
5.1	Addressing modes	20
5.2	Addressing in x86	22
5.3	Addressing in ARM	23
5.4	Instruction formats	23
5.4.1	x86 instruction format	24
5.4.2	ARM instruction format	25
6	Assembly language concepts	26
6.1	Assembly language elements	27
6.1.1	Pseudo-instructions	27
6.1.2	Macro definitions	28
6.2	Types of assemblers	30
6.3	Loading and linking	31

7	Assembly language x86	31
7.1	Directives	32
7.2	Starting the code	32
7.3	Registers	32
7.4	System calls	33
7.5	Jumps	34
7.6	Stack and function calls	34
8	Computer Arithmetic	36
8.1	Sign-magnitude representation	37
8.2	Twos compliment	37
8.2.1	Twos compliment arithmetic	37
8.2.2	Twos compliment multiplication	37
8.2.3	Twos compliment division	38
8.3	Floating-point representation	38
9	The Memory Hierarchy: Locality and Performance	42
9.1	Locality	42
9.2	Memory systems	42
9.3	Multilevel Memory Hierarchy	43
10	Cache	44
10.1	Elements of cache design	44
10.2	Types of cache misses	44
10.3	Associative mapping	45
10.4	Set Association mapping	45
10.5	Replacement algorithms	45
10.6	Write Policy	46
10.7	Line size	46
10.8	Inclusion policy	47
10.9	Cache timing models	47
10.10	Performance modeling of multilevel memory hierarchy	47
11	Internal Memory	48
11.1	Semiconductor main memory	48
11.1.1	Error correction	49
11.1.2	Synchronous DRAM	49
11.1.3	DDR DRAM	49

12 External Memory	50
12.1 Magnetic Disk	50
12.1.1 Performance	51
12.2 RAID	51
12.2.1 Hamming code	52
12.3 Solid State Drives	52
12.4 Optical memory	53
13 C programming language	53
13.1 IO	55
13.2 Memory allocation	55
14 I/O	56
14.1 Programmed I/O	56
14.2 Interrupt-driven I/O	57
14.3 Direct Memory Access	58
14.4 Direct cache access	58
14.5 I/O channels and processors	58
15 Link layer	59
15.1 ARP	59
15.2 Ethernet	59
16 Processor Structure and Function	60
16.1 Processor Organization	60
16.2 Register organization	60
16.3 Instruction Cycle	61
16.4 Instruction Pipelining	62
16.4.1 Performance	63
16.4.2 Branch prediction	63
16.5 Processor Organization for Pipelining	64
17 Reduced Instruction Set Computers	64
17.1 Instruction execution characteristics	64
17.2 Large register files	65
17.3 Advantages of RISC over CISC	66
17.4 Pipelining improvements	68
18 Instruction level parallelism and superscalar processors	69
18.1 Constraints	69
18.2 Design issues	69

19 Parallel Processing	70
19.1 Symmetric Multiprocessors	71
19.2 Cache Coherence and the MESI Protocol	72
19.2.1 Software Solutions	72
19.2.2 Hardware Solutions	73
19.2.3 The MESI Protocol	73
19.3 Multithreading and Chip Multiprocessors	75
19.3.1 Approaches to Explicit Multithreading	75
19.4 Clusters	76
19.5 Nonuniform memory access	76
20 Hardware Performance Issues	77
20.1 Multicore Organization	77
20.2 Heterogeneous Multicore Organization	78
21 Control Unit Operation and Microprogrammed Control	78
21.1 Micro operations	78
21.2 Control of the Processor	80
21.2.1 Control signals	80
21.3 Control Unit Implementation	81

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 times 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 so called computers on a chips, small chips which have the same requirements for the 4 basic functions of a computer.

Deeply embedded systems are microcontrollers 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 coming 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 is 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 Littles 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 several 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 tic 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 operations 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 is algebra based on only the values 1 or 0.

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

Another often useful 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 a circuit, the fewer number 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 is multiplied, and product of sums (POS) where sum is multiplied.

Both forms may not be the simplest 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 identities, which can simplify the expressions


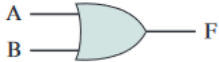
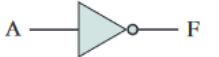

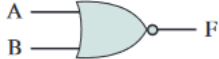

Name	Graphical Symbol	Algebraic Function	Truth Table															
AND		$F = A \cdot B$ or $F = AB$	<table><tr><th>\bar{A}</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	\bar{A}	B	F	0	0	0	0	1	0	1	0	0	1	1	1
\bar{A}	B	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = A + B$	<table><tr><th>\bar{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>	\bar{A}	B	F	0	0	0	0	1	1	1	0	1	1	1	1
\bar{A}	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$F = \bar{A}$ or $F = A'$	<table><tr><th>\bar{A}</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	\bar{A}	F	0	1	1	0									
\bar{A}	F																	
0	1																	
1	0																	
NAND		$F = \overline{AB}$	<table><tr><th>\bar{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>	\bar{A}	B	F	0	0	1	0	1	1	1	0	1	1	1	0
\bar{A}	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = \overline{A + B}$	<table><tr><th>\bar{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>	\bar{A}	B	F	0	0	1	0	1	0	1	0	0	1	1	0
\bar{A}	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$F = A \oplus B$	<table><tr><th>\bar{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>	\bar{A}	B	F	0	0	0	0	1	1	1	0	1	1	1	0
\bar{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 possibilities and a row which describes two variables there will be 4 possible outcomes.

For each row/column combination the function or truthstable 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 cannot 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 consistent 0.

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

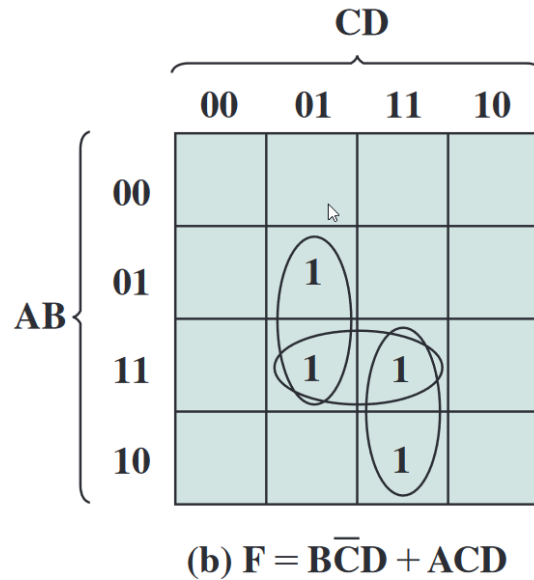


Figure 2: Example of kmap and the output function

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 exists 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 newfound 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	✓
$\bar{A} B C D$	7	0	1	1	1	✓
$\bar{A} \bar{B} C D$	11	1	0	1	1	✓
$A B \bar{C} D$	13	1	1	0	1	✓
$A B C D$	15	1	1	1	1	✓

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

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

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

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

Figure 3: Example of the method on

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

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

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 control 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 a number of 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 simplest form is a flip-flop, which can store one bit of data.

There are different types of flip flops with different properties The SR flip flop cannot have both S and R be 1 but is the simplest, 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 flips 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 flips when the right bit is 1.

3.4.5 Programmable logic devices

PLD are general-purpose chips.

There are different types of PLD

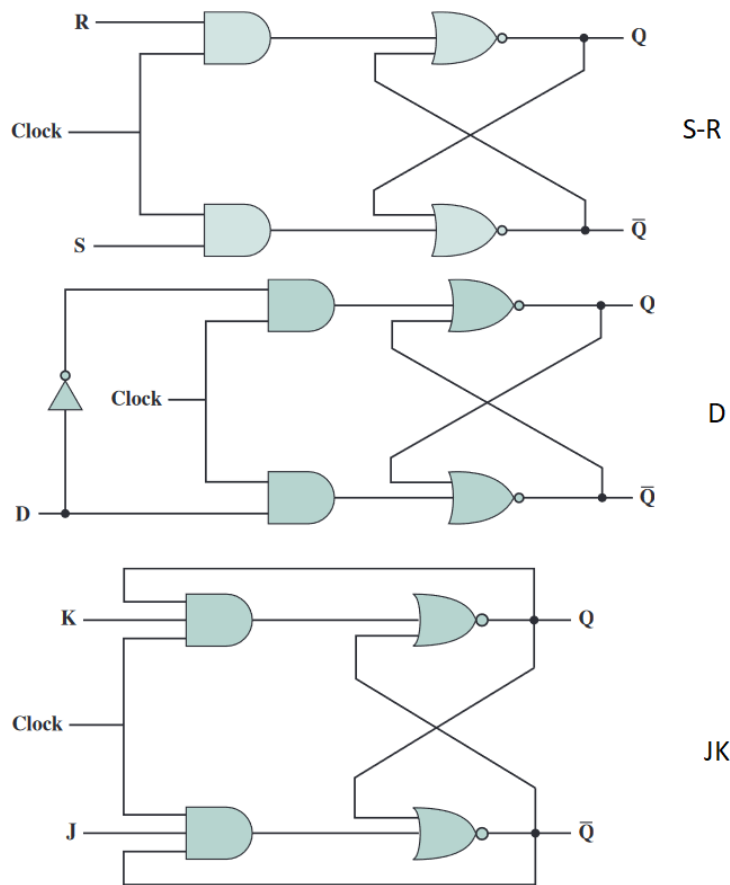


Figure 4: Different types of flip flops

- 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 consists of the following:

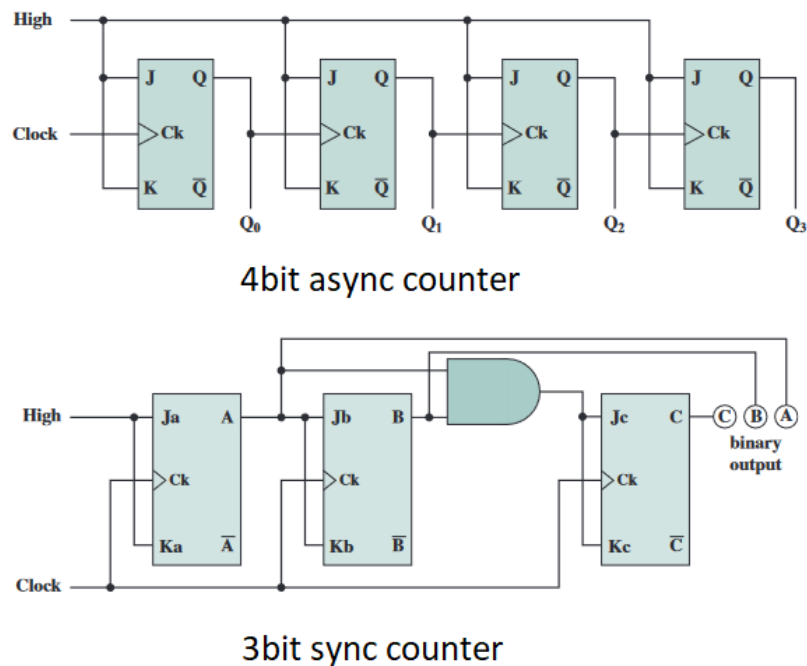


Figure 5: Two implementations of binary counters

- 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 exists
- 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 field 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 means the sign(1 = negative) the next 8 bits are the exponent and the next 23 are the mantissa 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 issues 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.

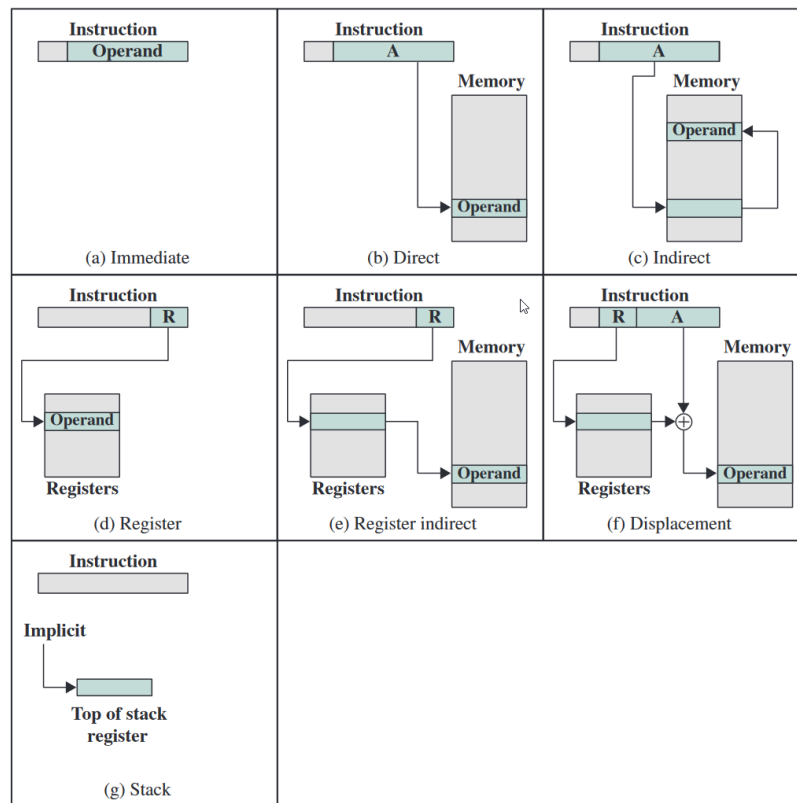


Figure 6: Illustrations of addressing modes

- 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

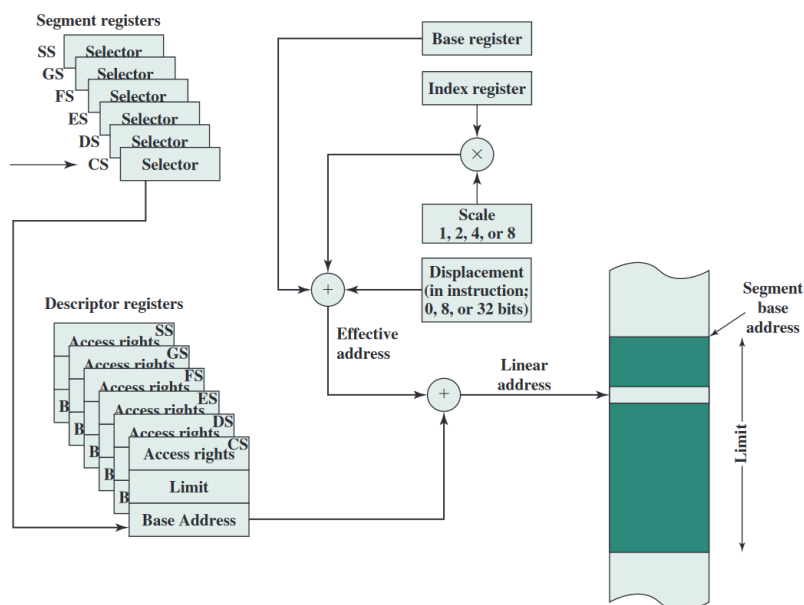


Figure 7: Addressing modes in x86 and how it is calculated

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 if 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 contains 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 number of registers also affect the number 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 instructions 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

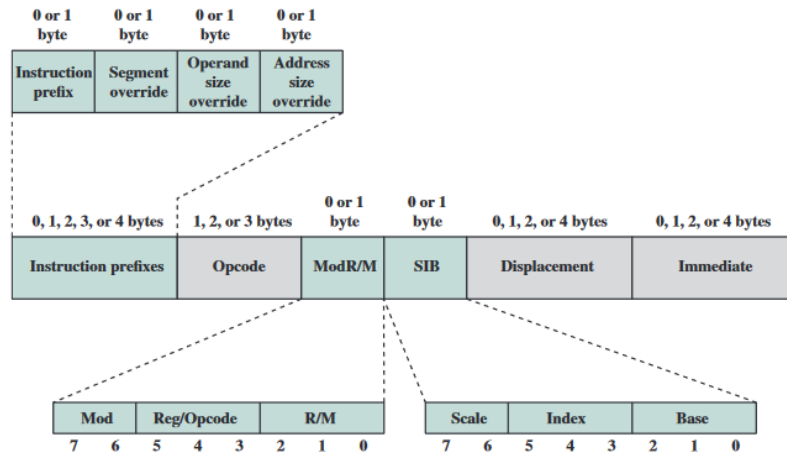


Figure 8: x86 instruction format and its lengths

- **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 on of 5
 - 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 full-size, 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 needs 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 operated 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 instructions 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 number of opcodes only require 2 bits and 3-bit type field
 - Thumb instructions only references register 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
- Probability 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 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 hll
- 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 consists 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

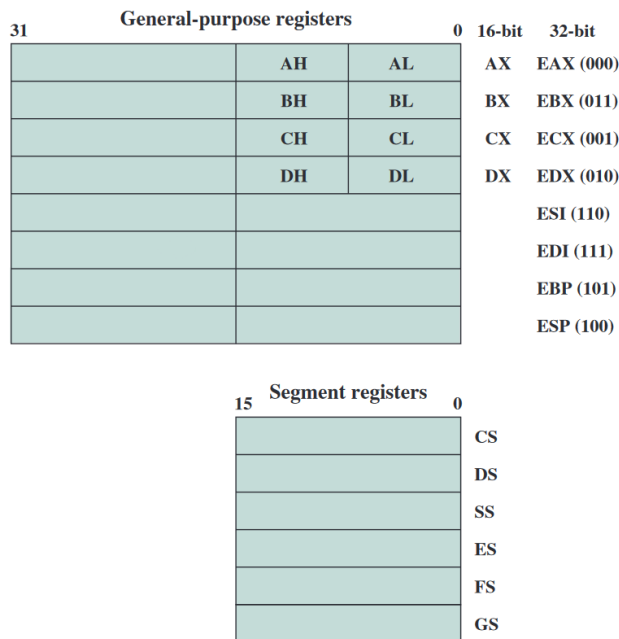
When referencing data in operands there are different methods as discussed. For register addresses the register name is used. For immediate addressing uses an indication that the value is encoded. Examples are H for hexadecimal, B for binary and decimal has no suffix

Ex. 100B is read as a binary value

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

6.1.1 Pseudo-instructions

Pseudo instructions 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 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 instruction 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
- Macro assembler - Allows the user to define sequences of instructions as macros
- Micro assembler - 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 references are 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 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

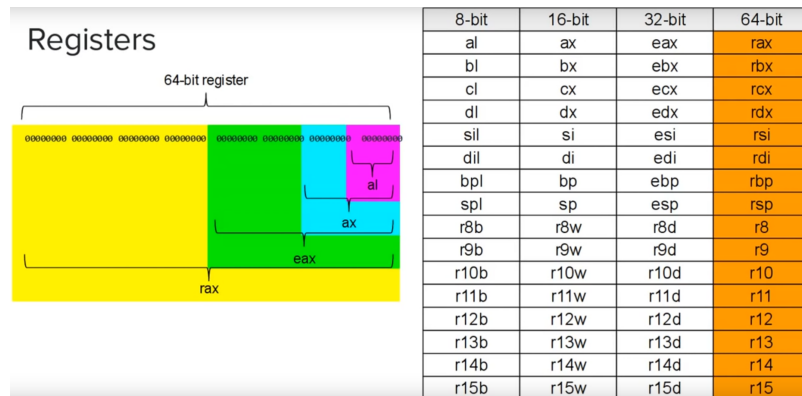


Figure 12: The different registers in x86 assembly

- R8-R15 - New registers in 64 bit
- RSP - Stack pointer, points to top of stack
- RBP - Stack 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 an exit system call is made with argument 0 symbolizing 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 conditions 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 must pop the arguments.

For first 6 integer/pointer arguments are in registers

RDI, RSI, RDX, RCX, R8, and R9

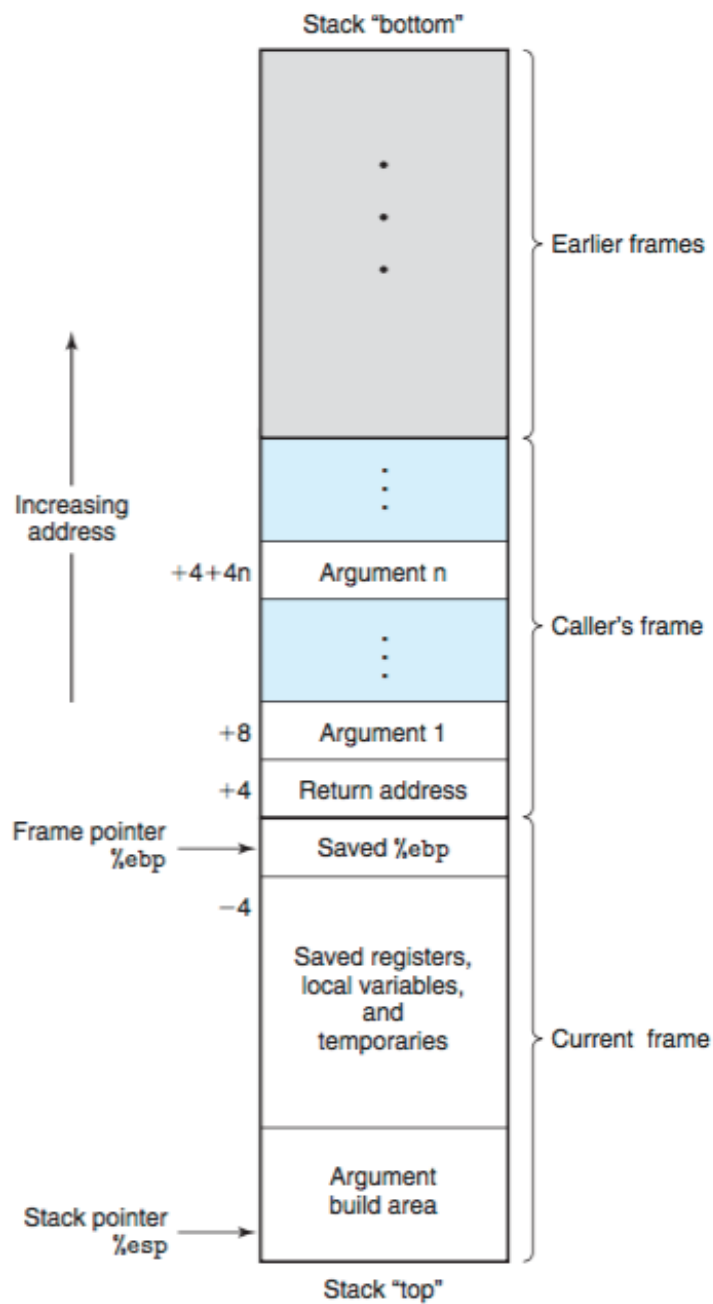
If more arguments are needed to stack is used.

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 jlabel`

And to return 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 arithmetic is not as simple and checking 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 simpler.

When negating twos compliment the most significant digit is flipped.

The two edge cases 0 negation and the largest number negation, here a carry must be remembered and 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 .

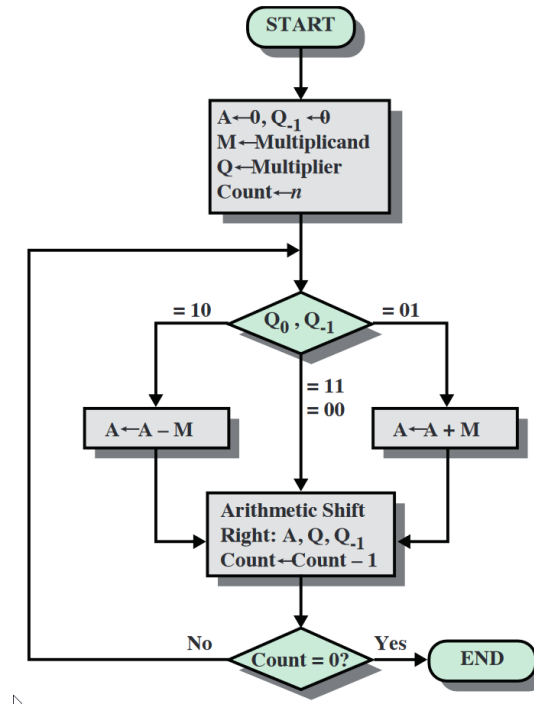


Figure 13: Two's complement multiplication diagram

8.2.3 Two's complement 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 .

8.3 Floating-point representation

For a typical 32-bit format of a floating point will be
1 bit for sign,

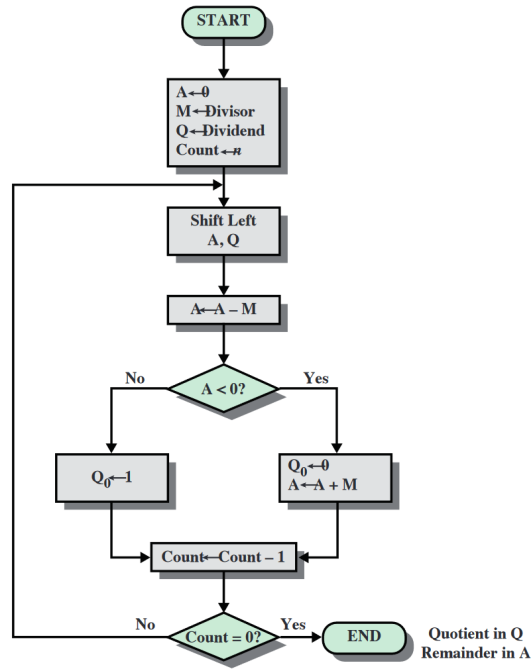


Figure 14: Two's complement division diagram

8 bits for exponent,

23 bits for mantissa

$sign1.mantissa \cdot 2^{exponent}$

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 significands - 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

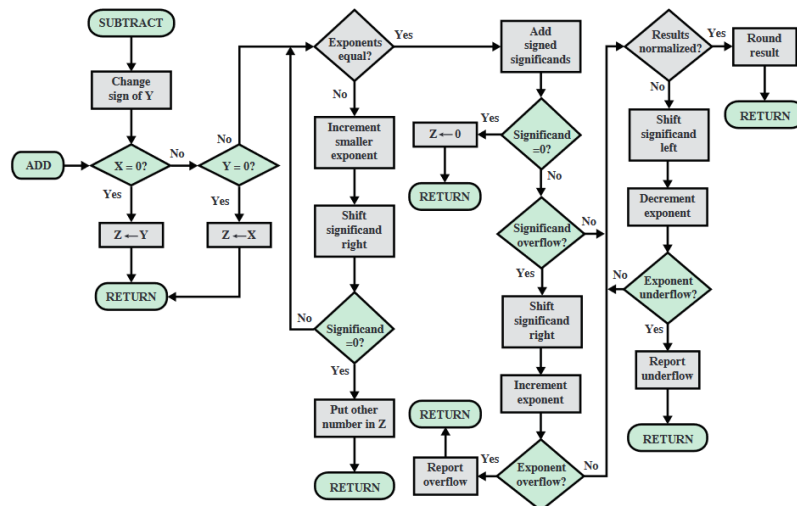


Figure 16: Floating point addition and subtraction diagram

Multiplication

- Check for zeros
- Exponents are added together
- Significands are multiplied as normal integers
- Normalize the result

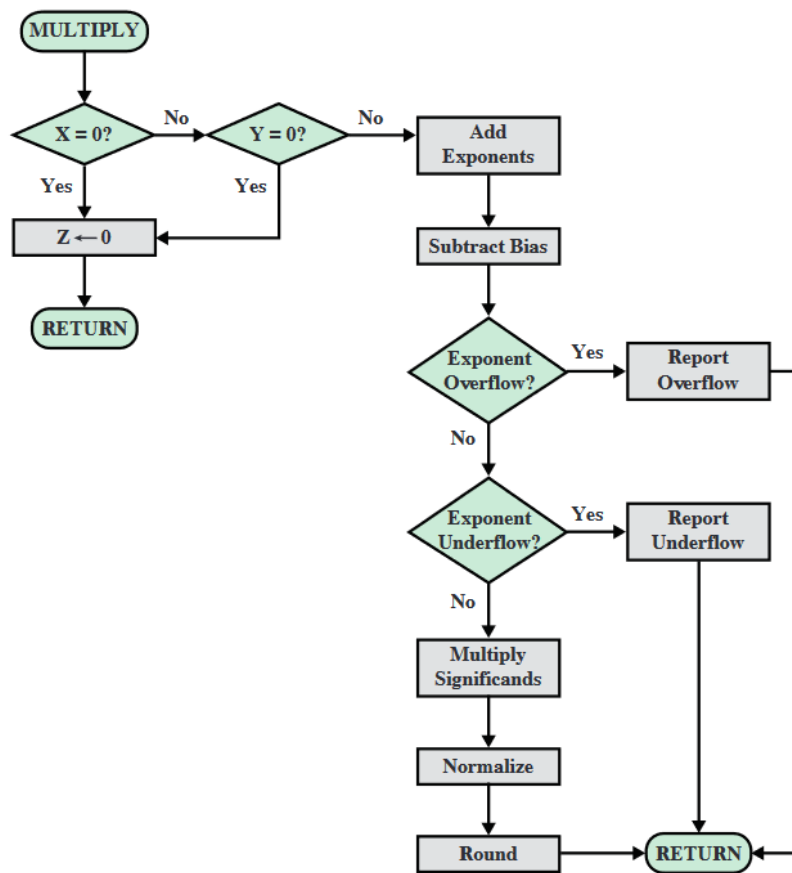


Figure 17: Floating point multiplication diagram

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 significant, such at left shift there will be less loss 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

9 The Memory Hierarchy: Locality and Performance

9.1 Locality

Locality is the act of caching memory or instructions.
Locality comes in two types

- Temporal Locality - Recent used data which is saved in case of reuse
- Spacial Locality - Data around gathered data

Temporal locality is very useful since many studies have found that many programs tend to use a small amount of data a lot and have access to a larger amount of data.

Likewise with spacial is the probability high especially for code since its most linear that surrounding data is useful in the future.

9.2 Memory systems

Glossary

- Word - A unit of length depending on system intel x86 32 bit
- Addressable units - The amount of addresses
- Unit of transfer - Number of bits written or read to/from main memory
- Sequential access - Data is stored in linear form in records which is moved around in units
- Direct access - Memory is assigned to a physical address
- Random access - Data is assigned to unique addressing mechanism

- Associative - Type of random-access memory which compares address data
- Access time (latency) - Time to perform a read or write operation
- Memory cycle time - Time to reset memory in RAM
- Transfer rate - the speed of which data is transferred denoted $T_n = T_a + \frac{n}{R}$, T_n =average time to read/write n bits, T_a =average access time, n = number of bits, R = transfer rate

9.3 Multilevel Memory Hierarchy

The memory hierarchy is used to distribute data such larger data capacity has slower access time, to compensate cost.

The hierarchy is groped as

- Registers
- On-chip cache
- Off-chip cache
- Main memory
- Flash memory
- Disk
- Off-line storage

on-chip cache is often implemented using SRAM and off-chip cache using eDRAM.

Cache unlike other memory is not visible to the programmer and is hardware controlled.

External nonvolatile memory is referred to as secondary,- or auxiliary memory For implementing a memory hierarchy, the following must be supported

- Locality
- Inclusion - Given data in one level the same data must be exist as a copy on all higher levels
- Coherence - If data is modified all copies must also be updated

10 Cache

Cache is used for temporary storage for faster access.

When data is not present in the cache a copy is read into the cache from which the CPU read the data.

The cache consists of blocks which is the minimum size of data transfer to the cache.

The blocks are contained in lines which can hold one block.

A tag is associated with the line for addressing purposes.

Each line also includes a control bit indicating if the data in memory has been modified.

A transfer is often less than a line which typically is 128 bytes and a block size being 2 bytes.

Cache is split into data and instructions, since it optimizes for old reusable data.

10.1 Elements of cache design

High performance computing (HPC) deals with super computers which require a whole different approaches to cache.

A program uses virtual addresses which is translated to physical by the memory management unit (MMU).

Cache can therefore save the virtual address or the physical.

By having the virtual a translation, it is not needed so it is faster, but must be flushed for every new program and bits to the virtual address for different applications.

Since there are fewer cache lines than memory block a selection and organization has been done.

The simplest is direct mapping which simply takes the main memory block modulo with the available cache lines which is the address.

10.2 Types of cache misses

- Compulsory Miss - Not present
- Conflict Miss - Data was overwritten
- Capacity Miss - The working set is larger than cache resulting in constant overwriting

10.3 Associative mapping

Associative mapping allows a physical address to be placed in any line of cache.

This therefore requires every line to be checked for data in cache.

For the SRAM is used which is more expensive and contains less storage but can in parallel in one clock cycle compare each line.

If the Main memory size is $= 2^k$ then k is the physical memory bits.

The block Size $= 2^b$ and b is the block offset.

The tag bits are then P.A. bits - Block offset.

10.4 Set Association mapping

This is a medium between associative and direct mapping.

By dividing the cache into sets which is directly mapped to memory from which the sets are associative.

Calculating in this can be done as:

Line numbers: lines

MM address: x bit system

Block size

offset $= \log(\text{blocksize})$

set number bits $= \text{index} = \log(\text{line numbers}) / (\log(\text{number of sets associated}))$

tag $= \log(\text{MM}) - \text{offset} - \text{index}$

This then constructs the address with the same address size as the rest of the system but in the form

Tag— Index — Offset

So, the address the first bits will be the tag, the next number of bits will be the index (i.e., the set it associated with), and the last bits will be the offset in the data. The index will then be at a set in which the cpu can search the tag.

10.5 Replacement algorithms

Associative and set associative require a replacement algorithm for choosing which cache to be overwritten.

Least recently used (LRU) keeps a list of most recently used cache, when cache is used it is moved on top.

When overwriting something the bottom is used.

First in first out (FIFO) keeps a list of the longest staying cache and overwrites the oldest.

Least frequently used (LFU) uses a list of counter of use and replaces the

least used cache.

10.6 Write Policy

When writing cache back to memory there are two cases

The memory has not been modified

The memory has been modified.

Memory may have been overwritten by the I/O or other caches.

Write through every update in cache is updated in memory which requires a lot of memory traffic.

Write back requires every write to happen in cache, when something is updated a cache, a dirty bit is set from which memory can be updated

In case of write miss there are two approaches

- Write allocate - The block containing the word to be written is fetched from memory to cache to then write
- No write allocate - The block containing the word is written directly in memory

No write allocate is most often used with write through, but the amount of moving proves inefficient.

Write back and write allocate is most used, due writing to both cache and memory due to in a miss it will hit cache next time and setting the dirty bit for the block

In case of multiple devices with cache a cache coherency is needed, approaches can be

- Bus watching with write through - Each cache controller check memory of cached data and if updated invalidates local cache
- Hardware transparent - hardware is used to update modified memory in all caches
- Noncacheable memory - shared memory is not cacheable

10.7 Line size

The larger the line size the higher chance is relevant adjacent data is fetched.

But larger lines overwrite old cache and less blocks can be in cache at once.

For larger lines the adjacent words become less likely to be used.

An optimum is found of 8 to 64 bytes.

10.8 Inclusion policy

Inclusive policy dictates that data in one cache is guaranteed to be in all lower levels of cache.

This simplifies searching since the can be done in smallest fastest caches first and moving up.

Exclusive policy dictates that cache is guaranteed to not be in lower levels of cache, this is done to not waste space but requires more intense searching.

Non inclusive policy dictates that nothing is guaranteed about placement of cache.

10.9 Cache timing models

t_{ct} = compare time of tag address and tag value in cache

t_{rl} = time to read a line from cache to retrieve data block in cache

t_{xb} = time needed to transmit byte or word to the processor

t_{hit} = time spend on cache level in case of hit

t_{miss} = time spend in cache in case of miss

Direct mapped cache

$$t_{hit} = t_{rl} + t_{xb}$$

$$t_{miss} = t_{rl} + t_{et}$$

Associative set cache

$t_{hit} = t_{rl} + t_{xb} + (1 - F_p)t_{et}$ where F_p is the fraction of time that the way prediction succeeds

$$t_{miss} = t_{rl} + t_{et}$$

For the associate cache it is the same as set but $t_{miss} = t_{rl} + t_{et}$

10.10 Performance modeling of multilevel memory hierarchy

For finding the average time to access an item it can be expressed as:

$$T_s = H \cdot T_1 + (1 - H) \times (T_1 + T_2)$$

$$T_s = T_1 + (1 - H) \cdot T_2$$

T_1 = Access time of fastest memory

T_2 = Access time to second level of memory

H = hit ratio

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H) \frac{T_2}{T_1}}$$

$\frac{T_1}{T_s}$ = Access efficiency.

11 Internal Memory

11.1 Semiconductor main memory

When talking about memory it is referred to as RAM (Random Access Memory) RAM comes in two types of DRAM and SRAM

DRAM (Dynamic) uses capacitors to store charge, the charge can then be interpreted as 1 if charge is above a threshold.

The memory cell uses a transistor such two lines comes in a bit line and address line, the address line starts the circuit, if bit line is off it can be charged by the capacitor,

If the bit line is one the capacitor will be charged.

The dynamic name comes from the need to recharge due to the slow loss of charge in the capacitors.

SRAM (Static) uses flip flops to store memory and therefore is digital instead of analog

DRAM is cheaper but SRAM is faster and therefore used for cache memory. ROM (Read Only Memory) must advantage of being hard coded such it is nonvolatile and fast.

PROM is the programmable version but more expensive.

Read Most Memory is for memory which is most read version of this includes

- EPROM - Erasable Programmable Read Only Memory is a programmable ROM which can be deleted using UV lighting
- EEPROM - Electrical EPROM can be erased with electricity instead of UV and is faster
- Flash memory - Is the fastest EPROM and provides eraseability down to chunks but not bytes and has high density

The architecture of the memory cell is controlled by a chip.

The architecture decisions may include how big chunks the memory cell should be ex 8 Mbit chip organized in 1M x 8 memory cells.

This chips uses a multiplexer to convert the numbering of the cell into an electrical signal to the cell.

The chip then has input to whether a read or write should be performed and the output.

The architecture can therefore also chain multiple chips together for larger amount of memory but keeping clustered memory cells.

eDRAM is memory between the memory and chip often named L4 cache.

11.1.1.1 Error correction

For error correction parity bits are used.

These are extra bits used to ensure the stored bits are correct.

When writing or reading to memory, is the memory inserted into a function.

This function generates check bits in a given way.

When reading the checkbits and the memory is gathered, the memory is once again inserted into the function and the new checkbits is XOR'ed with the old creating the syndrome.

If an error has occurred to the syndrome value, will it be equal to the bit which needs to be flipped.

This is an example of single-error-correcting (SEC) code.

Most often does semiconductor have both SEC and double-error-correction code which requires a bit more.

11.1.2 Synchronous DRAM

Normal DRAM when getting request for data will the CPU have to check up if it has been delivered.

Using synchronous the CPU can instruct on which next cycle the memory should be available.

SDRAM also makes use of multiple bank (set of chips) to perform parallel data management.

SDRAM also allows certain length of burst of data to be written into the bus. Therefore, making SDRAM fast at larger chunks of data.

11.1.3 DDR DRAM

DDR (Double Data Rate) is a spin on the SDRAM such data is gathered both on up and down clock cycles.

DDR also uses a higher clock rate for the BUS to increase transfer rate.

DDR also have a prefetch buffer on the chip, the memory buffer can get both buffer and memory on 1 cycle therefore making it possible for the BUS having the double clock rate.

12 External Memory

12.1 Magnetic Disk

Magnetic disks consist of a recording medium.

This medium is read by the head consisting of magnetoresistive (MR) sensor or written to by a magnetic coil.

The medium is split up into tracks at different radius and these tracks contain sectors at variable length.

To compensate for different speed at the inner track and outside track the bits are written with different intervals, this therefore result in the outer track stores the same data size the same as inner track.

Therefore, making the density the limit of data.

Multizone recording is a way to divide tracks into zones which have roughly the same data density. This allows the data density to be higher than the outer the tracks are.

A sector of data contains

- Gap
- Sync - Indicate start of sector and include timing alignment
- Address mark - Sector data including number, location and status
- Data
- ECC - error correction code

The format is in two ways the old 512 bytes where 50 is ECC and 15 is GAP, SYNC and Address mark.

The new format is 4k bytes, 100 ECC and 15 GAP, SYNC and Address mark. Different types of magnetic disk

- Fixed head disk - A head for each track
- Movable head disk - Head moves in between tracks
- Nonremovable - Stationary disk unlike portable versions
- Double sided - Magnetic coding on both sides of the medium
- Multiple platters - Stacked mediums and reader for more data
- Types of head
 - Air gap - Where air is present between reader and medium

- No gap - The medium is more robust and allows reader to be on medium
- Winchester - A foil rest on the medium when still and in spin the air pressure lifts the foil

12.1.1 Performance

$$t_B = t_S + t_L + t_T$$

t_B - bloc access time

t_S - Seek time, head to track

t_L - Latency time, time for sector to reach head

$$t_L = \frac{1}{2r}$$

r - Rotation speed in RPS t_T - transfer time from memory to BUS

$$t_T = \frac{b}{rN}$$

b - Number of bytes transferred N - Number of bytes on track Rotational positional sensing (RPS) is a server feature which tries to gain IO access when the data is about to read, if not available it will take a rotation and try again.

12.2 RAID

Redundant Array of Independent Disks comes in seven level 0 - 6

RAID uses a set of physical drives and creates logical drive.

The data is then distributed across the devices called stripping.

RAID also provides redundancy method for drive failure in some levels.

The RAID then also gives the ability of faster data transfer through multiple IO's

0. Level - The data is striped out on all disk without any redundancy. The data is striped such continuous data is spread out as much, such a single chunk can be split up into every drive.
1. Level - The data has a full duplicate, makes it two times the speed to get data, relative fast write since it depends only on slow drive, easy backup, lots of required capacity
2. Level - The data is stripped down to byte level, gives fast read and error correction and uses hamming code for parity, require log x extra disk where x is number is disk.

3. Level - The data is stripped down to bytes, parity is on a single drive where parity bit of the same strip place, fast read due to byte level strip
4. Level - The data is striped in larger chunks, parity is depending on strips therefore making write time slower, for high I/O request fast but not big transfer
5. Level - Like level 4 but larger strip to prevent I/O bottle neck in RAID 4
6. Level - Like Level 5 but another disk is also used with another parity method such two drives can fail

12.2.1 Hamming code

Hamming code works by every 2^n bit is a parity bit and everything else is data bit.

The parity bit works by being 1 if a selection of data bit is odd and 0 if even. Comes in different formats where for the 7-bit version is

p1,p2,d1,p3,d2,d3,d4

Where

p1 = d1,d2,d4

p2 = d1,d3,d4

p3 = d2,d3,d4

A single error can then be detected and deducted for which bit is flipped

12.3 Solid State Drives

SSD make use of the NAND flash.

The SSD has:

- Higher input/output operations per second (OPS)
- More durability to physical factors
- Longer lifespan due to no moving parts
- Lower power consumption
- Lower access time and latency rate

The SSD also contains beside the flash

- Controller for interface and firmware execution

- Addressing
- Data buffer/cache
- Error correction logic and detection

The downside of the SSD is scattering problem. When data is scattered into multiple cell as consequence of filling up cell the write and read time will slow down.

The cell is also typically limited around 100,000 writes.

12.4 Optical memory

CDs and CD-ROMs are optical disk used for storing data.

With techniques like the HDD but with optical lasers.

Uses variable rotation speed, with a laser at a constant linear velocity.

The optical disk is engraved such at flat level represent 0 and changes in level represent 1.

The data is split into block of

- Sync - byte of 0, 10 bytes of 1 and a byte of 0
- Header - represent the following data and if error correction is at place
- Data
- Auxiliary - either extra data or error correction according to header

It has the advantage of cheap mass production and ability of storage of data and saving of data as backup.

CD-Recordable (CD-R) is a CD with a consumer-friendly write

CD-Rewriteable (CD-RW) can be written to multiple times using a phase changeable crystal, with a limited amount of erase cycles

Digital Versatile Disk (DVD) is a more compact solution to CDs with capacity up to 4.7 GB

Blue ray used like DVD a finer laser resulting in more density up to 25 GB.

13 C programming language

C is a performance constrained language which focuses on minimizing run-time overhead.

It is in direct control of memory and compiles to native code.

Was designed for implementing UNIX

C has no support for

- Objects
- Exceptions
- Function overloading
- range check on arrays
- Garbage collection
- Limited type safety

To combat no run time error sanitizers are used. These run before compiling checking for essentially run time errors.

Compiling C is done using gcc, this can be to object files using -c or directly to executable

To get warnings -Wall and for more warning -Wextra.

For debug -g is used.

So, a good compile command should be:

gcc -Wall -Wextra -g -fsanitize=address -fsanitize=leak -fsanitize=undefined

Valgrind is an alternative to sanitizers which runs the program in a virtual a machine using:

valgrind -leak-check=full -track-origins=yes ./myProg arg1 arg2

The C program uses a main in the form `int main(int argc, char **argv)` or with just void.

An empty parameter means the function takes any number of parameters.

Lines starting with # is used for preprocessing such as inclusions or macros

Macros simply takes blindly and copies the macro into the wanted places.

Macros are made as: `#define PI 3.14` and undefined by `#undef PI`

Preprocessing can also use conditions like `#ifdef DEBUG` where -DEBUG is an argument to gcc

For negative variables signed versions is used

short	int	long	pointer	long long	Label	Examples
—	16	—	16	—	IP16	PDP-11 Unix (1973)
16	16	32	16	—	IP16L32	PDP-11 Unix (1977)
16	16	32	32	—	I16LP32	Macintosh, Windows
16	32	32	32	—	ILP32	IBM 370, VAX Unix
16	32	32	32	64	ILP32LL64	Win32, Unix (1990s)
16	32	32	64	64	IL32LLP64	Win64
16	32	64	64	64	I32LP64	Unix (most of them)
16	64	64	64	64	ILP64	HAL
64	64	64	64	64	SILP64	UNICOS

For error handling goto are often used as exceptions.

Arrays are initialed by: `int arr[10] = {1,2,3}` where the rest is just zeros.

Pointers are made by: `*p = &i` where `i` may be `i = 42`
 Using `**p = &p` will then be the pointer to the value of `i` i.e., 42
 Additions to pointers is allowed and good use for array pointers.
 Strings are made by `char str[] = "Hello"`

13.1 IO

`getchar` and `putchar` for single character input output to `std`
`printf` writes string to `stdout` including formatting.
`scanf` is the read equivalent
`s` version of `printf` and `scanf` does not include length check.

Format specifier	Meaning
<code>%c</code>	Print a single character
<code>%i, %d</code>	<code>printf</code> a decimal integer
<code>%u</code>	print an unsigned decimal integer
<code>%x</code>	print an unsigned hexadecimal integer using a,b,c,d,e,f
<code>%X</code>	<code>printf</code> an unsigned hexadecimal integers using A,B,C,D,E,F
<code>%o</code>	print an unsigned octal integer
<code>%f</code>	print a floating point number
<code>%e</code>	print the floating point number in the exponential format
<code>%E</code>	Same as the <code>%e</code> . but it is print E for exponent
<code>%g</code>	print the floating point number in float <code>%f</code> or <code>%e</code> format whichever is shorter.
<code>%%</code>	print the <code>%</code> sign.
<code>%s</code>	print the string
<code>%p</code>	print the pointer

13.2 Memory allocation

Effective type is the type which the object relies on.
`malloc` is used for allocation space, with no effective type until written, and returns a pointer to start of allocated space.
`sizeof` returns the size of an object type, ex `sizeof(int)`
`free` function free up the space after use and takes a pointer to the space which the free up.
`calloc` allocates space and fille with zeros
`realloc` is reallocating memory from one place to new place. The old pointer should never be used.

14 I/O

I/O modules contain logic for communicating between peripherals and system bus.

The module is required to handle peripheral data format and mismatch between data transfer rate of processor and peripherals.

Peripherals are split up into three categories: Human readable, machine readable, and communication for remote devices.

Control logic associated with the I/O module controls peripherals in response to direction from IO module.

Transducers convert electrical signal to digital input, often with a buffer most often 8 to 16 bits or larger if block oriented.

The functions of the IO modules come in categories

- Control and timing - coordinate flow of traffic between internal and external resources
- Processor communication - Command decoding from processor to peripherals, data transfer between bus and peripherals, status reporting of devices, address recognition of peripherals
- Device communication - Commands, status information and data
- Data buffering - buffer data in case of different speeds
- Error detection - Error detection of mechanical and electrical malfunctions and error detection code

Three techniques are possible for I/O operations

- Programmed I/O - The program gets full control of IO, the processor must wait for next execution with IO and check when status is open for when finished
- Interrupt-driven I/O - Uses programmed I/O but interrupts the processor when done, such it can handle the data
- Direct memory access (DMA) - Uses interrupt driven IO but can transfer data to main memory itself

14.1 Programmed I/O

IO commands consist of which IO module, which peripheral and what command.

The commands are classified as

- Control - activate peripheral and issue a command
- Test - Test condition of IO module and peripherals
- Read - Obtain data from peripheral and place in internal buffer
- Write - Take data from system bus and transmit to peripheral

For addressing a peripheral there are two ways, memory mapped I/O uses a single address space for memory location and IO devices, such peripherals are treated as memory locations.

The other is isolated I/O uses a single read line and single write line on bus, such it specifies whether a memory location or peripheral is being addressed.

14.2 Interrupt-driven I/O

The main idea is the IO module will interrupt the processor once done. When the IO module completes an operation, the following steps is done.

1. I/O device issue interrupt signal to processor
2. Processor finishes what currently is going on and responds
3. Processor test for interrupt to find module and issue an acknowledgement signal to remove interrupt signal
4. Prepare to work with IO module by saving next instruction in the program status word (PSW), and registers onto the system control stack
5. Next instruction for the processor is set to the interrupt handling program
6. The interrupt is then handled where the whole IO is handled
7. Registers and PSW instruction is retrieved

For the processor to find the IO module which raised an interrupt there are different design approaches

Multiple interrupt lines between processor and IO modules but is impractical to dedicate multiple bus lines.

Software poll essentially ask every module until it find the correct one, but it is slow.

Daisy chain is hardware poll which chain every IO module and an ack is sent to all, from which it will be answered by the correct one, from which it can

be identified called a vector.

Bus arbitration also use vectored interrupts as in daisy chain which uses the bus to place the vector.

14.3 Direct Memory Access

The problem with Interrupt and programmed IO is the limited speed of which the processor can check for interrupt and the processor being tied up on IO transfers.

The DMA module is a module which connects to the IO and handles IO processes.

The DMA then only uses the bus when data must be transferred and then steals cycles from the processor to use the system bus.

The DMA is commanded by sending a read or write control line, the address of the wanted IO device, starting point of read / write, and the amount to write / read.

The DMA module is then able to handle memory and IO directly without the processor, and once done sends an interrupt signal to the processor.

The DMA module can be connected to IO via the system bus, the IO can be connected directly to the DMA module, or the IO is on a bus between the DMA and other IOs.

14.4 Direct cache access

Problem with DMA is it is too slow for some speeds today.

DCA uses the last level cache of the processor instead of main memory.

This can be seen when using networking at high speeds

The multiple unwrapping of protocols would result in many back and forward interaction between cache and main memory.

14.5 I/O channels and processors

An IO channel is an extension to DMA which make it able to itself execute IO instruction, such the CPU does not do any IO processing.

Instruction is stored in main memory and executed by a special purpose processor for the IO channel.

The instruction then includes the needed information to perform the IO action.

There are two common types of IO channels.

Selector channel controls multiple high-speed devices at one at a time.

A multiplexor channel works with a larger set of device at the same time with slower data throughput but can chain reads from different devices.

15 Link layer

15.1 ARP

Address Resolution protocol is a protocol for getting MAC addresses of local devices.

In case of LAN connected through a router the router will include an ARP module and an adapter

The router is then capable to both forward ARP messages and share its own ARP map of MAC addresses.

15.2 Ethernet

Ethernet was a standard which dominated the LAN setups using a hub which forward not packets but rather bits into all connected devices.

This was hub-based star topology, and the problem is collision may occur in case of sending while forwarding.

The switch therefore replaced the hub where collision would not occur.

The ethernet frame consist of

- Data field - 46 bytes to 1500 bytes- carries the ip datagram
- Destination address - 6 bytes - MAC address of destination
- Source address - 6 bytes- MAC address of source
- Type field - 2 bytes - type of payload in data field may not be IP datagram
- Cyclic redundancy check (CRC) - 4 bytes
- Preamble - 8 bytes - 7 repeats of 10101010 and then 10101011 used to synchronize timing in case of not perfect target rate of a router

In case of error from error check the package is dropped with no alert.

16 Processor Structure and Function

16.1 Processor Organization

A processor is required to

- Fetch instruction - read instruction
- Interpret instruction - instruction is decoded
- Fetch data - potential required read for instruction
- Process data - Arithmetic or logical operation on data
- Write data - write result from instruction

Moreover, does it need memory for last and next instruction and temporary storage for execution.

The processors major components are

- Arithmetic and logic unit (ALU) - computation
- Control unit (CU) - data movement
- Registers
- Internal processor bus - transfer of data between ALU and registers

16.2 Register organization

Registers perform two roles

User-visible registers - used by the programmer for temp storage

Control and status registers - used by privileged operating system program to control execution

The user visible registers come in different categories

- General purpose
- Data - may only hold data and not part of calculation
- Address - general purpose but may be devoted to addressing modes like
 - Segment pointers - points to base of segments
 - Index registers
 - Stack pointer - point to top of stack

- Condition codes - written to by hardware visible by user, indicates things like overflow

The most essential register for operation is

- Program counter (PC) - instruction to be fetched
- Instruction register (IR) - most recent instruction
- Memory address register (MAR) - address of a location in memory
- Memory buffer register (MBR) - word of data to be written to memory

Bus connects directly to MAR and MBR.

The ALU is directly connected to MBR may be through a buffer

The program status word (PSW) contains status information with common field as

- Sign - sign of result of recent execution
- Zero - if result is 0
- Carry - if operation resulted in carry
- Equal - if test results in equal
- Overflow - if arithmetic resulted in overflow
- Interrupt - If interrupts are enabled or disabled
- Supervisor - If the process is in privileged mode

16.3 Instruction Cycle

In general terms the instruction cycle goes as: Fetch - instruction read from memory at place PC with MAR onto the bus

Control unit request read data into IR and PC is incremented by 1 for next fetch.

The control unit check if IR is using indirect addressing if so, moves it into MBR and request a memory read.

16.4 Instruction Pipelining

Instruction pipelining is the idea of running instruction as a set of operations, such instruction can be running semi parallel such an operation is always occupied.

The simplest pipeline is fetch stage and execution stage, where when the execution takes place, the fetch can get the next instruction ready called fetch overlap.

A latch is then used in between the stages to forward data at clock cycles.

A problem is execution is slower than fetching therefore the process can be subdivided further

- Fetch instruction (FI) - read expected instruction
- Decode instruction (DI) - Decode instruction
- Calculate operands (CO) - Calculate address of operands
- Fetch operands (FO) - Fetch each operand
- Execute instruction (EI) - Execute instruction
- Write operand (WO) - Write result

In this way the duration is close the equal.

Conflict may occur if Co stage depends on data written by previous instruction, or likewise data conflicts known as pipeline bubble.

There are three types of hazards

- Resource hazards - Two or more instructions need the same resource also known as structural hazard
- Data hazards - Conflict in the access of an operand location, which is further divided into three types
 - Read after write (RAW) or true dependency - read takes place before the write operation is complete avoided by reassigning or renaming registers
 - Write after read (WAR) or anti dependency - write is performed before read operation takes place
 - Write after write (WAW) - Two instruction both write to same register
- Control hazards - Wrong prediction of branching therefore fetch must be discarded

The process is not further subdivided due to overhead time in moving data and the amount and complexity of control logic.

16.4.1 Performance

The cycle time of an instruction pipeline is determined by

$$\tau = \max_i(\tau_i) + d$$

$$\tau = \tau_m + d \quad 1 \leq i \leq k$$

Where

τ_i = time delay of the circuitry in the i th stage of the pipeline

τ_m = maximum stage delay

k = number of stages in instruction pipeline

d = time delay of a latch, most often equal clock pulse

The total time required for a pipeline with k stages and n instruction will be

$$T_{k,n} = [k + (n - 1)]\tau$$

The time benefit of more stages can be described as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

16.4.2 Branch prediction

Another problem is branches where the instruction cannot be fetched before execution.

For this there is different prediction techniques

- Multiple streams - Create fetch stream for both branch and normal execution
- Prefetch branch target - Fetch both branch and normal execution
- Loop buffer - A buffer containing the next sequential addresses, and is checked to see if branch is already in fetched buffer, ideal for loops and if else if else statements
- Branch prediction - different techniques for guessing what execution will be taken

- Predict never taken - branch never taken
- Predict always taken - branch is always taken
- Predict by opcode - certain branch opcodes dictates if branch is expected
- Taken/not taken switch - Each branch has an associated bits for history of branch to predict branching
- Branch history table - cache memory with entries of: address of branch — some number of bits which record history — information such as branch address

The branch prediction Taken/not taken switch and branch history table is dynamic prediction and refined version is referred to as two level or correlation-based branch history.

Delayed branch is rearranging instruction automatically such branch instruction occur later than desired.

16.5 Processor Organization for Pipelining

To improve pipelining the processors L1 cache is separated into I cache and D cache to remove conflicts.

Likewise, is EXS divided into different units allowing pipelines to send instructions to different units while other are in use.

A buffer can also be applied for operands to EX units until operands are available.

17 Reduced Instruction Set Computers

17.1 Instruction execution characteristics

For Computers the most expensive things are software, therefore high-level languages (HLL) have been developed to speed up programming.

This have in result created a semantic gap where the HLLs does not reflect computer architecture.

One way to close the gap is to create a large instruction set matching the HLL and dozens of addressing modes.

A different approach is making the architecture simpler but faster rather than complex.

The most used operations in HLLs are simple movement of data, to speed up more registers can be available

Likewise, are conditional statements (IF, LOOP) very used, making sequence mechanism of the instruction set important.

The most time-consuming part of programs is procedure calls, to speed it up two observations can be made

- Most calls are with fewer than 6 arguments - making the number of words required per procedure activation small
- Most calls have a narrow depth of invocation - making the operand references highly localized

17.2 Large register files

Two approaches are taken to maximize register usage

In software where the most frequent data is put into registers.

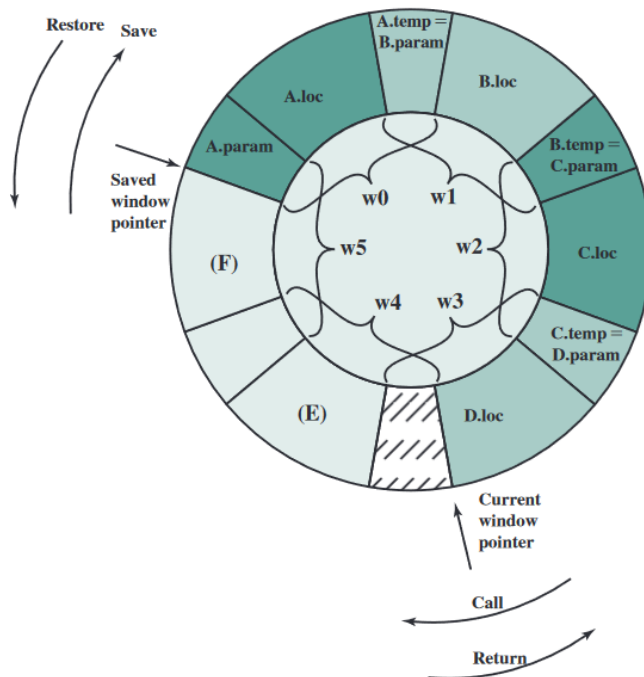
And in hardware is more registers.

Since most variables are local scalars, registers can be split up into windows of registers for a procedure.

Therefore, by calling a procedure the registers do not need to be saved to memory.

For parsing and returning data a shared window can be used where parameters and returns can be used, such no movement of data is needed.

In this way the registers can be implemented as a circular buffer



For global variables the most frequent can be assigned registers and less used can be in memory.

This approach makes the registers into a small very, somewhat inefficient due to windows likely not being used up, fast buffer.

For the software approach the compiler can give every variable a virtual register.

The virtual registers are then put into a map with as nodes and being connected if used in the same procedure.

Then a graph coloring where to adjacent nodes can have the same color can be done with a maximum of n colors being equal to number of registers.

Then each color can be assigned a register, such some variables can share the same register.

17.3 Advantages of RISC over CISC

The general trend is richer instruction sets, with the reasoning being simpler compiler and improve performance.

Making the compiler gets harder due to more complex cases to use new instructions.

It is expected that fewer instruction is needed for a program hence making it smaller.

The smaller size makes the program more cacheable and faster since fewer

instructions is fetched.

But studies show that for most cases it only a 10% reduction.

This is due to CISC favoring simple instructions and RISC using more register addresses than memory addresses.

The more complex instruction in CISC does save some clock cycles but makes the control unit more complex and make execution time of simple instruction higher.

It has also been found that the speedup comes in hand with the complex control unit since it can act as a cache.

RISC is characterized by

- Machine cycle - A cycle is defined as, fetch two operands from register, perform ALU operation, store result in a register, hence making execution faster since a microprogram control store is not accessed in execution
- Register to Register - Only Load and Store operations which simplifies control unit
- Simple addressing modes - RISC only include simple register addressing and may have displacement and PC-relative
- Simple instruction formats - Instructions format is a fixed length with fixed field location such as opcode, making decode of field parallelizable and simplify control unit, the instruction is also aligned such it does not cross page boundaries

Interrupts is also more effective since it can happen in between operations and not wait for large complex operations.

RISC is characterized by

- A single instruction size
- That size is most often 4 bytes
- A small number of data addressing modes
- No indirect addressing
- No operation that combine load/store with arithmetic
- No more than one memory addressed operand per instruction
- Does not support arbitrary alignment of data for load/store operations

- Maximum number of uses of MMU for a data address in an instruction
- Number of bits for integer register specifier equal to five or more
- Number of bits for floating point register specifier equal to four or more

RISC can also easier be pipelined since simple instruction are easier to split into stages.

Delayed branch is used in case of dependencies, that effects does not take place before after execution of instruction.

Delayed load continues execution while waiting for load until an execution need loading data.

Loop unrolling is used in case of loop which may be more effective unrolled such the body is copied number of used times

17.4 Pipelining improvements

By merging the instruction decode ID and operand fetch (OF) into a single ID stage will improve speed especially in RISC since most memory is in register and little time is spend in either.

Memory operand fetch is referred to as the load/store unit (LSU)

The instruction buffer works such instruction and be prefetched and buffered in case of a L1 cache miss and more time is spent on a fetch.

In case of branch the buffer is flushed but speed is still improved.

The prodecoder (PD) offloads some task from the ID stage to avoid bottleneck.

The PD is between L2 cache and L1 instruction cache, since the slow L2 cache time it can spend it on decoding.

Store buffer store data to be written, in case of newly written data instead of waiting for write and load it can be fetched from the store buffer.

The reservation station is such the ID can buffer up instruction and does not have to wait for the functional unit (FU) to finish

Dedicated reservation station has a buffer for each FU, the slots in the reserve station acts as a virtual FU.

This is also referred to as an instruction window.

Data forwarding makes data input to the reservation station, to combat read after write delays.

Reorder buffer allows instruction to be executed out of order (OoOE).

18 Instruction level parallelism and superscalar processors

Superscalar refers to a machine that is designed to improve the performance of executing scalar instructions.

The approach is multiple pipelines.

Unlike traditional scalar organization, which uses a single pipeline.

Using hardware and compiler the parallel execution is done without violating the intent of the program.

The performance is measured in issue rate which is the number of instructions issued per instruction cycle.

For operations which must be done in order a reorder buffer is used.

Superpipelined is an alternative way to improve performance by dividing pipeline stages into greater numbers such two stages can be done pr cycle, thus increasing the temporal parallelism.

18.1 Constraints

Instruction level parallelism is the degree to which on average the instructions of a program can be executed in parallel.

This is constrained by

- True data dependency / Read after write (RAW) dependency
- Procedural dependency - Branch problem
- Resource conflict - Two execution using the same resource
- Output dependency / Write after write (WAW)
- Antidependency / Write after read (WAR) dependency

18.2 Design issues

The degree of instruction level parallelism is determined by the frequency of true data dependencies and procedural dependencies in the code.

Also determined by the operation latency, i.e., the time until the result of an instruction is available for use as an operand in a subsequent instruction.

Machine parallelism measures the ability of the processor to take advantage of instruction level parallelism, i.e., the number of parallel pipelines.

Instruction issue refer to the process of init instruction execution.

Instruction issue policy is the protocol used to issue instructions.

When issuing processes there are three types of ordering

- The order in which instructions are fetched
- The order in which instructions are executed
- The order in which instructions update the contents of register and memory locations

Grouped superscalar instruction issue policies can be categories as:

- In-order issue within order completion - sequential execution and write result in the same order
- In order issue with out of order completion - out of order execution but the write result is the same as sequential, so allowing longer taking executions to be started earlier
- Out of order issue with out of order completion - the decode and execute stages is decoupled such instruction is put into an instruction window and fetched if they do not need a dependency

Register renaming is used such values in dependent registers is not changed. This directly combat WAR and WAW.

When a new register value is created a new register is allocated for that value. A reference to the new register is then stored in the instruction.

19 Parallel Processing

Parallel processing is categorized into

- Single instruction, single data (SISD) stream - single processor with single stream of input
- Single instruction, multiple data (SIMD) stream - multiple processes managed in a single instruction
- Multiple instruction, single data (MISD) stream - a sequence of data transmitted into multiple instructions
- Multiple instruction, multiple data (MIMD) stream - Processors with each data streams

MIMD is general purpose.

MIMD can be subdivided into:

Symmetric multiprocessor (SMP) - processors share a single memory or pool of memory through bus

Nonuniform memory access (NUMA) - Access time to memory may differ from processor to processor

Cluster - a collection of independent uniprocessors or SMPs

19.1 Symmetric Multiprocessors

SMP have the following characteristics

- Two or more similar processors of comparable capability
- Share the same memory and I/O facilities through a bus or internal connection scheme with approximately the same access time
- All processors can perform the same functions
- Controlled by an integrated operating system

SMP have the following advantages

- Performance - Allow portions of work to be performed parallel
- Availability - Failure of a single processor does not halt
- Incremental growth - Performance can be increased by adding processors incrementally
- Scaling - Pricing will be lowered by vendors due to large range of products and prices

A time-shared bus is used for DMA by having the features

- Addressing - Distinguish between different modules (processors)
- Arbitration - Priority scheme for allowing modules to be primary
- Time-sharing - When one module is using the bus other modules must be locked out

The bus is used due to its simplicity, flexibility and reliability in case a module failure.

The main drawback is performance.

Therefore, each processor is equipped with a cache memory to reduce bus use.

The operation system responsible for schedule execution and allocate resources.

Some issues arise with this responsibility

- Simultaneous concurrent processes - OS routines need to allow several processors to execute the same code and avoid deadlock or invalid operations
- Scheduling - assign ready processes to available processors
- Synchronization - Enforce mutual exclusion and event ordering
- Memory management - Exploit available hardware parallelism for performance with consistency
- Reliability and fault tolerance - Graceful degradation in the face of processor failure and restructure management tables according in processor failure

19.2 Cache Coherence and the MESI Protocol

Cache coherence problem occurs when multiple copies of same data exist in different caches simultaneously.

Approach is the MESI (modified,exclusive,shared,invalid) protocol.

Objective is get appropriate cache and stay there through numerous reads and writes while maintaining consistency.

19.2.1 Software Solutions

Achieve cache consistency through operation system and compiler

Issue arises with compilers being conservative and leading to inefficient cache utilization

Done by analyzing the code for unsafe and mark those points from be cached. A more efficient approach analyze code to determine safe periods for shared variables, enforced with instruction inserted in code.

19.2.2 Hardware Solutions

Also known as cache coherence protocols.

Transparent to the programmer and is dealt with when problems arise.

Comes in two categories directory protocols and snoopy protocols

Directory protocol

Collect and maintain information where copies are stored, by a centralized controller in a directory stored in main memory.

The directory contains global information and is checked for cache changes before data is requested.

The centralized controller maintains exclusive access for writing and alerts other cache with copies to invalidate.

Drawbacks is a lot of bus interactions and overhead communication.

Snoopy protocols

Each cache has responsibility to maintain coherence.

Updates is announced to caches by caches.

Two basic approaches: write-invalidate and write-updates.

Write-invalidate multiple readers and a single writer, mostly used

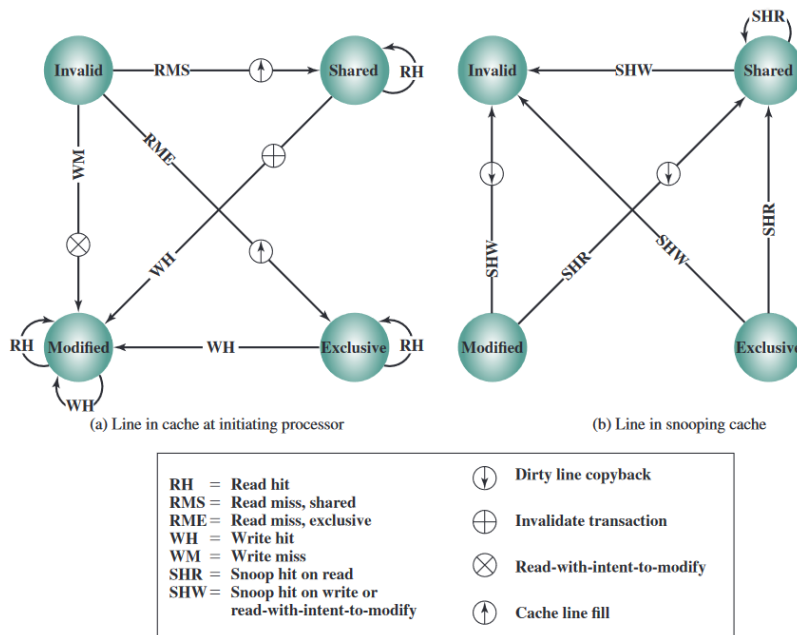
Write-update multiple writers and multiple readers, announce when updates are happening.

19.2.3 The MESI Protocol

Provides cache consistency on an SMP.

Data include two status bits such data can be in four states

- Modified - Cache has been modified and available only in this cache
- Exclusive - Not modified but only saved in main memory and current cache
- Shared - Not modified saved in main memory and multiple caches
- Invalid - The cache data is not valid anymore



When cache operation is done there is four action results.

- Read miss - A cache miss happens, and a read is initiated with the outcomes
 - Other caches have not modified the data, and is read from main memory
 - Other caches have a copy and is signaled to change to shared state
 - A cache contains modified copy, is signaled to save it to memory, and change to shared and data is read from shared bus
 - No other cache has a copy is read from main memory and set to exclusive
- Read hit - No state change
- Write miss - Gain data from memory, signal to bus read-with-intent-to-modify (WRITM), two scenarios: a cache contains modified data and gives bus to the cache for save and no cache contain data and a read and write is performed
- Write hit - if the data state is
 - Shared - Signal to bus to invalidate other cache with the shared data and set its own to modified

- Exclusive - Update its data to modified state
- Modified - Perform update

Having this in L1 cache with write through to L2 cache will work.

19.3 Multithreading and Chip Multiprocessors

The optimization of a single threading has reached a limit due to complexity and power consumption.

Multithreading divides instruction streams into smaller streams known as threads such as executed in parallel.

Process - instance of program with two characteristics - resource ownership and scheduling/execution ability

Process switch - switch from process to process and save current state

Thread - dispatchable unit of work within a process

Thread switch - Like process switch for threads

Two thread types - user threads and kernel threads not visible to the user.

19.3.1 Approaches to Explicit Multithreading

Explicit multithreading = execute instruction from different explicit threads.

Program counter for each thread of execution to be executed concurrently.

Instruction fetching takes place on a thread basis.

Four approaches to multithreading

- Interleaved multithreading - The processor deals with two or more thread contexts at a time switching from one thread to another at each clock cycle
- Blocked multithreading - The instructions of a thread are executed successively until an event occurs that may cause delay such as cache miss
- Simultaneous multithreading (SMT) - Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor
- Chip multiprocessing - multiple cores each handling separate threads

19.4 Clusters

A group of interconnected computers working together.

Benefits include

- Absolute scalability - surpass the power of even the largest standalone machines
- Incremental scalability - units can be added throughout use to scale
- High availability - A failure of a node does not affect the rest of the cluster
- Superior price/performance - common components and computers can be used with more competition

Can be configured based if the cluster share access to the same disks.

The simplest each computer has its own disks.

Other alternative is a shared disk space.

Passive standby - another computer to the primary stands by and listens to heart beats such in case of failure in can take over.

On a separate server the disk can be partitioned into volumes for each computer and in case of failure the volume is handed over.

19.5 Nonuniform memory access

Uniform memory access (UMA) all processors have access to all parts with same access time.

Nonuniform memory access (NUMA) access to all parts with different access time.

Cache-coherent NUMA (CC-NUMA) cache coherence in with NUMA.

SMP system have practical limit where bus performance becomes a bottleneck.

is organized in nodes containing multiple processors each with its own L1 and L2 cache and main memory.

Is connected by means such as network.

If data is needed from remote it is fetched and coherence is kept using some sort of directory.

This is more scalable than SMP and when data is placed out it is likely to not need to fetch around much.

But requires software changes from both application and operation system.

20 Hardware Performance Issues

Processor designs have focused on in chronological order:

- Pipelining
- Superscalar
- Simultaneous multithreading (SMT)

Power requirements have grown exponentially with chip density and clock frequency

Pollacks rule - performance increase is roughly proportional to the square root of increase in complexity

Threading granularity - minimal unit of work that can be beneficially parallelized, the finer granularity the more parallelizable is the program.

20.1 Multicore Organization

The main variables for multicore organization

- Number of core processor pr. chip
- Number of levels of cache memory
- How cache memory is shared
- Whether simultaneous multithreading (SMT) is employed
- Type of cores

The use of shared higher-level cache in the chip has several advantages over exclusive reliance on dedicated caches

- Reduce overall miss rates
- Data shared by multiple cores is not replicated at shared cache level
- With proper line replacement algorithms the amount of shared cache allocated to each core is dynamic so that threads have less locality can employ more cache
- Inter core communication is easy
- Confines the cache coherency problem to the lower cache levels

20.2 Heterogeneous Multicore Organization

Homogenous multicore organization - multiple cores is a chip with multiple identical cores

Heterogeneous multicore organization - Mix of different types of cores

Different types of cores may be cores which focus on some instruction sets such as GPU cores which focus on matrix and vector processing.

Heterogeneous System Architecture (HSA) key features is

- Entire virtual memory space is visible to both CPU and GPU
- Virtual memory system brings in pages to physical main memory as needed
- Coherence memory policy ensures that CPU and GPU caches both see correct data
- Unified programming interface that enables users to exploit the parallel capabilities of the GPUs withing program that rely on CPU execution as well

Digital signal processors (DSPs) provide fast processing of analogue data to digital data.

Arms big.Little architecture focus on a power core and an efficiency core.

Comes in two models Migration model where power and efficiency core are paired, and work is given to appropriate core while other goes idle.

Multiprocessing (MP) allow a mix of cores to be on and chosen on power demand from application, more complicated but more efficient.

21 Control Unit Operation and Microprogrammed Control

21.1 Micro operations

Micro operations are the atomic operations of a processor.

A micro-operation is performed in a single time unit.

Example the fetch cycle

The fetch cycle involves the four registers

- Memory address register (MAR) - specifies the address in memory for a read for write operation

- Memory buffer register (MBR) - value to be stored in memory or the last value read from memory
- Program counter (PC) - next instruction to be fetched
- Instruction register (IR) - last instruction fetched

The fetch cycle consists of

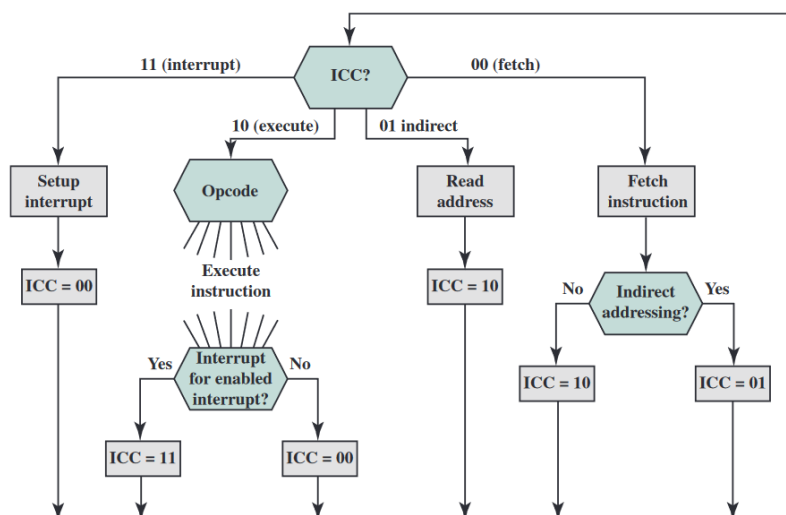
1. Move the PC address to the MAR
2. MAR is moved to bus
3. Control unit issues a read and result copied into MBR
4. Pc is incremented

The last two steps can be at the same time since they do not affect each other.

Grouping can be done if the proper sequence of events is followed, and conflicts is avoided.

When performing operations, a 2-bit register is kept up to date for the current state of the program execution where

- 00: fetch
- 01: Indirect
- 10: Execute
- 11: Interrupt



21.2 Control of the Processor

Control unit have the three characteristics

- Define the basic elements of the processor
- Describe the micro-operations that the processor performs
- Determine the functions that the control unit must perform to cause the micro-operations to be performed

The control unit perform two basic tasks

- Sequencing - Make the processor perform proper sequence of micro-operations
- Execution - Cause micro-operations to be performed

21.2.1 Control signals

For the control unit to perform its task it has some inputs

- Clock
- Instruction register - Opcode and addressing mode of the current instruction
- Flags - Status of the processor and outcome of previous ALU
- Control signal from control bus

From which it must create the outputs

- Control signal within the processor - Comes in two types: data to be moved from one register to another and activate specific ALU functions
- Control signal to control bus - Two types: control signal to memory and control signal to the I/O modules

Looking at the fetch cycle using by the signaling it will become

1. Signal open gate for content from MAR onto the address bus
2. Memory read signal on the control bus
3. Signal open gate from data bus in MBR

4. Control signal to ALU to add 1 to the PC

For optimization the processor can have both an internal bus and access to external to limit traffic on external which is only required internally but require more control signal to differentiate.

21.3 Control Unit Implementation

Hardwired approach tries to implement the control unit using logic gates with logic signal as input and output logic signals as output.

This the circuit can be construction by making a truth table and finding an optimized boolean expression.

Problem is the equations become very complicated when all functions is implemented. An alternative is microprogrammed control.

This implements firmware (middle between hardware and software) using microprogramming language.

Each line of the language describes a set of micro-operations occurring at one time and is known as a microinstruction.

This construct control words which describe the actions to be taken be the processor, like signal and conditions.

The control unit then implements a control memory, which have a concise description of the complete operation of the control unit with the sequence of microinstructions.

The next instruction or microprogram is then loaded from memory into the control buffer register, which when read will execute the content.

What to be loaded into the control buffer register is based on three decisions

- Get the next instruction - add 1 to the control address register
- Jump to new routine based on a jump microinstruction
- Jump to a machine instruction routing