Digital Computer Architecture

July 18, 2010

Some random notes for the course exam. Based on slides by the course staff.
May contain errors, use at your own responsibility.

DOES NOT CONTAIN ALL OF THE COURSE MATERIAL!!! Only parts of it.
Version 0x5B

Contents

I As seen on the lectures... 3
- 0.0.1 Architectures . . . 3
- 0.0.2 CPI and other measurements . . . 3
- 0.0.3 Transistors: . . . 4
- 0.0.4 Boolean algebra: . . . 4
- 0.0.5 Circuits . . . 5
- 0.0.6 State machines: . . . 5
- 0.0.7 Timing: . . . 5
- 0.0.8 RTL and VHDL . . . 5
- 0.0.9 CPU Data-path . . . 5

II Stuff from Tirgulim 6
- 0.1 Number Bases . . . 6
- 0.2 Representation methods . . . 7
  - 0.2.1 Sign and magnitude . . . 7
  - 0.2.2 r's complement (general definition) . . . 7
  - 0.2.3 $(r-1)$'s complement (general definition) . . . 7
  - 0.2.4 Calculating the $(r-1)$ complement . . . 7
  - 0.2.5 Calculating the r's complement using the $r-1$ complement . . . 8
  - 0.2.6 1's complement (representation method) . . . 8
  - 0.2.7 2's complement (representation method) . . . 9
- 0.3 Representing real numbers . . . 10
  - 0.3.1 The fixed point representation system . . . 10
  - 0.3.2 The floating point representation system . . . 10
  - 0.3.3 IEEE 754 ("Aye triple-E") . . . 11
- 0.4 Circuit timing . . . 13
  - 0.4.1 Combinatinoal circuit timings . . . 13
  - 0.4.2 Sequencial circuit timings . . . 14
- 0.5 Cache . . . 14
- 0.6 Boolean algebra . . . 15
  - 0.6.1 Axioms . . . 15
  - 0.6.2 Claims . . . 16
  - 0.6.3 Useful equivalences . . . 16
  - 0.6.4 Basic truth tables . . . 17

III Other things **17**

0.6.5 Binary subtraction 17

Part I

As seen on the lectures...

0.0.1 Architectures

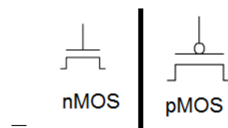
- **Von-Neumann** vs. **Harvard** architectures:
  - Von-Neumann: Single memory with code and data.
  - Harvard: Separate memory for code and data.
  - Most computers use Von-Neumann architecture, but are implemented using Harvard uArch.
- **RISC** vs. **CISC**
  - RISC (Reduced Instruction Set Computer): The instructions are primitive operations (mostly).
  - CISC (Complex Instruction Set Computer): The instructions are complext and in many cases include several primitive operations.
  - More in-depth comparison on lecture 6, slide 9.
- **ISA**: Instruction Set Architecture. This is the so-called "interface" of commands supplied by the hardware. It is the hardware's internal responsibility to implement these commands, and the user only sees the instruction set, from the outside.
  - The architecture is what's visible to the user.
  - The micro-architecture (a.k.a uArch) is the way in which the processor implements the architecture.
    - This includes caches size and structure, number of execution units, etc.. Also note that timing is considered uArch even though it is visible to the user.
- Different processor families have a different instruction set (and syntax). Compatibility between two different processor types can be one of:
  - Assembly compatible: Use the same instruction set.
  - Binary compatible: Use the same machine codes.

0.0.2 CPI and other measurements

- **CPI:** Cycles per instruction.
- **IC** is the instruction count; the total number of instructions executed in the program.
- $CPI = \frac{\text{\# of } \textbf{cycles} \text{ required to execute the program}}{IC}$
- **IPC:** Instructions per cycle. $IPC = \frac{1}{CPI}$.
- CPU Time $= IC \times CPI \times$clock cycle $=$ #cycles required to execute program $\times$clock cycle.
- More details on calculating CPI:
  - IC_i : Number of times instruction of type i is executed in the program.
  - IC : Number of instructions executed in the program. $IC = \sum_{i=1}^{n} IC_i$.
  - F_i : The relative frequency of instruction of type i: $F_i = \frac{IC_i}{IC}$.
  - CPI_i : Number of cycles needed to execute instruction of type i.
    - For example, we can have $CPI_{add} = 1$, $CPI_{mul} = 3$, etc.
  - # of cycles required to execute the program: $\#cyc = \sum_{i=1}^{n} CPI_i \cdot IC_i$
  - $CPI = \frac{\#cyc}{IC} = \frac{\sum CPI_i \cdot IC_i}{IC} = \sum CPI_i \cdot \frac{IC_i}{IC} = \sum CPI_i \cdot F_i$
  - Note that $\#cyc = CPI \cdot IC$.

- **Amdahl's law**
  - Suppose an enhancement accelerates a fraction of a task by a certain factor, and the remainder of the task is unaffected.
  - $ExTime_{new} = ExTime_{old} \times \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}\right)$
  - $Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1-Fraction_{enhanced})+\frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$
  - Example on lecture 1, slide ~34:
    * Floating point instructions improved to run at 2×, but only 10% of executed instructions are FP.
    * $ExTime_{new} = ExTime_{old} \times (0.9 + \frac{0.1}{2}) = 0.95 \times ExTime_{old}$
    * $Speedup_{overall} = \frac{1}{0.95} = 1.053$
- Measuring performance: Using benchmarks, etc. on real hardware, or running simulations. Note that when measuring CPI/IPC for comparison, the data only has meaning when using the same ISA and the same compiler.

0.0.3 Transistors:

- MOSFET transistors (Metal Oxide Semiconductor Field Effect Transistor - MOS) are the most common types of transistors in use today.
  - **N-MOS:** When the gate is at low voltage, transistor is OFF, and no current flows through it. When the gate is at high voltage, transistor is ON, and current can flow through it.
  - **P-MOS:** Behaves the opposite of nMOS. Low voltage →ON, high voltage →OFF.
  - 

  - Example of NOT circuit using nMOS,pMOS: Lecture 2, slide 20. Also on lecture 3, slide 5.
  - Example of NAND circuit: Lecture 3, slide 10.
  - Example of NOR: Lecture 3, slide 13.
  - TODO more gates as circuits??????????????????

0.0.4 Boolean algebra:

- Sum of products, product of sums: Lecture 3, slides 18+.
  -

x	y	z	F1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

מכפלות סטנדרטיות:

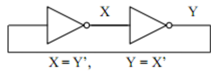
F1(x, y, z) = x'y'z + x'yz +
xy'z' + xyz'

סכומים סטנדרטיים:

F1(x, y, z) = (x + y + z)·(x + y' + z)·
(x' + y + z')·(x' + y' + z')

  - Ones in the F column represent the multiplications, and zeroes represent the sums.
  - We can also use shorthand notation, using the binary number value of the assignment to x, y, z: Here we have $F1(x, y, z) = m_1 + m_3 + m_4 + m_6 = \sum(1, 3, 4, 6)$, and also $F1(x, y, z) = M_0 + M_2 + M_5 + M_7 = \prod(0, 2, 5, 7)$.
  - Min-max representation: Something is very unclear, to me at least, in the lecture slides (lec. 3, slide 20). The Wikipedia article "Canonical form (Boolean algebra)" which covers this topic seems to be in contradiction to the lecture? TODO???????????

0.0.5 Circuits

- **Combinational circuits:** Have inputs, outputs. At any given moment, the value of the outputs is determined by the momentary value of the inputs.
- **Sequencial (?) circuits (מעגלים סדרתיים):** Logical circuit that contains feedback relations. The value of the outputs is determined by the value of the inputs as well as the memory units. The value of the memory units is a function of the past inputs (with significance to the order of the inputs in the past).
  - Asynchronous serial circuit: A circuit whose state doesn't change in coordination with the clock. The input and feedback values can affect the circuit state at any given moment.
  - Synchronous serial circuit: A circuit whose behaviour depends on input values and the internal components only in discrete times (clock-dependent). It is coordinated with the clock.
- This thing: 

is only stable for $X = 1, Y = 0$ or $X = 0, Y = 1$.
- SR-Latch: Lecture 3, slides 30+
- Flip Flop: A basic memory unit which is synchronized with the clock. Lecture 3, slides 44+ show how to build a D-Flip Flop using two Gated D-Latches.

0.0.6 State machines:

- Mealy machine - outputs depend on current inputs.
  - Therefore, in the machine diagram, the outputs are shown on the arrows together with their triggering inputs.
- Moore machine - outputs depend on state only (w/o current inputs).
  - Therefore, in the machine diagram, the outputs are written inside the states themselves.

0.0.7 Timing:

- Use Tirgul section in this document, lecture slides are not clear...

0.0.8 RTL and VHDL

- RTL = Register Transfer Language.
  - A general name for hardware description languages.
  - Unlike "regular" programming languages, RTL defines operations that can be done in parallel, like the actual hardware behaves.
  - VHDL is a type of RTL language (and so are HDL, Verilog, System-C and others).
- VHDL = VHSIC Hardware Description Language (VHSIC = Very High Speed Integrated Circuit).
  - Examples: Lecture 4, slides 25++++

0.0.9 CPU Data-path

- In MIPS, the the offset in the *beq* (branch on equal) command is shifted left 2 bits, so that it's a word offset. This means we have an effective range that is 4x larger, and that we always jump in whole word sizes. (P&H page 295).
- In MIPS, branches are delayed: The instruction immediately after a branch is always performed. This is done because of pipeline considerations. Note that in P&H chapter 5, this is ignored and the branch is not delayed. (P&H page 297).

Part II
Stuff from Tirgulim

0.1 Number Bases

- Algorithm to switch from base r to base 10:
  - Input: A number $(a_n a_{n-1}...a_1 a_0.a_{-1}a_{-2}....a_{-m})_r$
  - Output: The number's value in base 10 representation is

$$\sum_{i=-m}^{n} a_i r^i$$

  - For example: $(253.1)_6 = 2 \cdot 6^2 + 5 \cdot 6^1 + 3 \cdot 6^0 + 1 \cdot 6^{-1} = (105.167)_{10}$
- Algorithm to switch a whole number from base 10 to base r:
  - Input: A (whole) number N in base 10, and a base r to convert to.
  - Output: $(a_{i-1}a_{i-2}...a_1 a_0)_r$
    - $i \leftarrow 0$
$while(N > 0)\{$
$a_i \leftarrow N \, mod \, r$
$N \leftarrow \lfloor \frac{N}{r} \rfloor$
$i \leftarrow i + 1$
$\}$
- Algorithm to switch a number $0 < N < 1$ from base 10 to base r:
  - Input: A number $0 < N < 1$ in base 10, and a base r to convert to.
  - Output: $(0.a_{-1}a_{-2}...a_{-m})_r$
    - Let $N = a_{-1}r^{-1} + a_{-2}r^{-2} + ... + a_{-m}r^{-m}$, and the algorithm is:
$i \leftarrow 1$
$while(\lfloor N \rfloor \neq N)\{$
$N \leftarrow N \cdot r$
$a_{-i} \leftarrow \lfloor N \rfloor$
$N \leftarrow N - \lfloor N \rfloor$
$i \leftarrow i + 1$
$\}$
- A simple algorithm for converting base 2 to base 2^k:
  - Let $N = a_0 2^0 + a_1 2^1 + ... + a_m 2^m$. We will convert it to $N = b_0(2^k)^0 + b_1(2^k)^1 + ... + b_{\frac{m}{k}}(2^k)^{m/k}$.
  - $for(i = 1 \, to \, \frac{m}{k})\{$
$a_i \leftarrow (a_0, ..., a_k)_{2^k}$
$N \leftarrow \lfloor \frac{N}{2^k} \rfloor$
$i \leftarrow i + 1$
$\}$
  - TODO find a normal algorithm, this doesn't make sense.

0.2 Representation methods

0.2.1 Sign and magnitude

The number is represented as: $a_{n-1}a_{n-2}...a_0$.

a_{n-1} is the sign bit: $Sign = \begin{cases} Positive & a_{n-1} = 0 \\ Negative & a_{n-1} = 1 \end{cases}$.

Therefore,

$$N = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} a_i 2^i$$

- Range of representation: $-(2^{n-1}-1), ..., -1, 0, 1, ..., +(2^{n-1}-1)$.
- Attributes:
  - Readable.
  - Easy to implement.
  - Complicated arithmetic operations.
  - Zero has two representations! $-0 = 100...0$, $+0 = 000...0$.

0.2.2 r's complement (general definition)

Given a positive number N in base r with a whole part that consists of n digits, N's r's complement is defined as $r^n - N$ for $N \neq 0$, and 0 for $N = 0$.

- The 10's complement of $(52520)_{10}$ is $10^5 - 52520 = 47480$. Note that $n = 5$ here.
- The 10's complement of $(25.639)_{10}$ is $10^2 - 25.639 = 74.361$.
- The 2's complement of $(101100)_2$ is $(2^6)_{10} - (101100)_2 = (1000000 - 101100)_2 = 010100$
- The 2's complement of $(0.0110)_2$ is $(1 - 0.0110)_2 = (0.1010)_2$, and notice that $n = 0$ here.

0.2.3 $(r-1)$'s complement (general definition)

Given a positive number N in base r with a whole part that consists of n digits and a fractional part which consists of m digits ($N = a_{n-1}a_{n-2}...a_1a_0.a_{-1}a_{-2}...a_{-m}$), N's $(r-1)$'s complement is defined as $r^n - r^{-m} - N$.

- The 9's complement of $(52520)_{10}$ is $(10^5 - 1 - 52520) = 99999 - 52520 = 47479$
- The 9's complement of $(25.639)_{10}$ is $10^2 - 10^{-3} - 25.639 = 99.999 - 25.639 = 74.360$
- The 1's complement of $(101100)_2$ is $(2^6 - 1)_{10} - (101100)_2 = (111111 - 101100)_2 = (010011)_2$
- The 1's complement of $(0.0110)_2$ is $(1 - 2^{-4} - 0.0110)_2 = (0.1111 - 0.0110)_2 = (0.1001)_2$

0.2.4 Calculating the $(r-1)$ complement

- The 9's complement of a decimal number is done by reducing each digit from 9.
  - Example: 9's complement of $(52520)_{10}$ is $(47479)_{10}$
- The 1's complement of a binary number is obtained by changing all 0's to 1's and vice versa.
  - Example: 1's complement of $(101100)_2$ is $(010011)_2$
    - In fractions it's possibly more tricky. See example above.

0.2.5 Calculating the r's complement using the $r-1$ complement

From the dry definitions, we get that in order to get the r's complement, we can compute the $r-1$ complement and just add r^{-m} to it. (Reminder: m is the number of digits in the fractional part of the number).

- When dealing with whole numbers (non-fractional), $m=0$ so we can just add 1 to the $r-1$ complement.

0.2.6 1's complement (<u>representation method</u>)

The number is represented as: $a_{n-1}a_{n-2}...a_0$.

$$N = -a_{n-1}(2^{n-1}-1) + \sum_{i=0}^{n-2} a_i 2^i$$

Practically, this means:
To convert from 1's complement representation to regular base-10 numbers:

- If the leftmost bit is 1, then when we calculate the 1's complement (by bitwise-not-ing the number), we'll get the positive value of the number. But since the leftmost bit was 1, we should make it negative.
  - Example: $11011 \rightarrow$complement: $00100 = 2^2 = 4 \Rightarrow$We get: $11011 = -4$.
    Can also be calculated directly: $11011 = -1 \cdot (2^4 - 1) + (2^3 + 2^1 + 2^0) = -15 + 11 = -4$.
- If the leftmost bit is 0, then we just read the number as a regular binary number (it's positive).

To convert from regular base-10 numbers to 1's complement representation:

- If the number is positive, just stick a 0 as the sign bit.
- If the number is negative, just take its positive value, and calculate the complement of it (bitwise-not).
  - Example: The number -5 represented in 1's complement using 5 bits:
    $00101 \rightarrow$complement: 11010, and thus 11010 is the representation.

Addition (and subtraction):

1. Add up the two numbers.
2. If there's a carry ("circular carry", ???), delete it, and add 1 to the result.

- For example:
- $-5-6=(-5)+(-6)=-11$

$$\begin{array}{rr} & 11010 \\ + & 11001 \\ \hline & 110011 \\ + & \longrightarrow 1 \\ \hline & 10100 \end{array}$$

Detecting an overflow:
If the result of adding/subtracting two numbers is not representable by the determined number of digits we use, then the sign of the result will be **inverse** to that of the two operands. (Note that this situation can only occur when both operands are of the same sign).

- Example: $11+13=24$

$$\begin{array}{rr} & 01011 \\ + & 01101 \\ \hline & 11000 \end{array}$$

Note the sign bit is inverse to that of the operands!

Properties of this representation method:

- Range: $-(2^{n-1}-1), ..., -1, 0, 1, ..., +(2^{n-1}-1)$
- Arithmetic operations are relatively simple, but we have to add the carry to the calculation.
- Two representations for zero: $-0 = 111...1$, $+0 = 000...0$.
- The complement of a complement of a number equals itself.

0.2.7 2's complement (representation method)

The number is represented as: $a_{n-1}a_{n-2}...a_0$.

$$N = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Practically, this means:
To convert from 2's complement representation to regular base-10 numbers:

- If the sign bit is 0, just use regular binary conversion.
- If the sign bit is 1, then use the method of finding the 1's complement and then adding 1. This will bring us the positive version of our number, and we must make it negative. (For a clear example see below ↓).
  - Example: $-4 = 11100$. To convert: $11100 \rightarrow$The 1's complement: 00011, and add 1: $00100 = 2^2 = 4$. But since we're dealing with a negative number, make it negative: -4.

To convert from regular base-10 numbers to 2's complement representation:

- **Note**: I'm not 100% sure about this bit, there may be an error here.
- First of all, if the number is positive, just convert it to binary and set the sign bit to 0. This means, $0[number,\, in\, binary]$.
- Otherwise, the number is negative. In this case, calculate the regular binary representation of the number in its absolute value. For example if $N = -5$, take the binary representation of 5: 00101.
  - Now, use one of the following:
  - Method 1:
    1. Calculate the number's 1's complement.
    2. Add 1 to the result.
    * Example: $01101100 \rightarrow$One's complement: 10010011, add 1 and get 10010100.
  - Method 2:
    1. Advance from the right, leaving 0's unchanged until the first digit which is a 1.
    2. Leave the first 1 unchanged.
    3. For every digit to the left of this first 1, replace the digit with its "NOT", i.e. inverse the rest of the number.
       * Example: The one from before. Try and see that it gives the same result.

Addition (and subtraction):

1. Add up the two numbers.
2. If there's a carry, delete it.

- Example: $9 - 4 = (+9) + (-4) = 5$

$$\begin{array}{r} 01001 \\ +\quad 11100 \\ \hline 100101 \\ \downarrow \\ 00101 \end{array}$$

Detecting an overflow:
Note: This was a bit unclear from the slides, so it might be incorrect.

- Just like in 1's complement, if we add two positives and get a negative, or add two negatives and get a positive, then we have an overflow. It's sufficient to look at the sign bit to determine this.

Properties of this representation method:

- Range: $-(2^{n-1}), ..., -1, 0, 1, ..., +(2^{n-1}-1)$
- Simple arithmetic operations.
- Single representation for zero.
- If we take a number (works for negatives, too, apparently) N in binary representation and calculate its two's complement (take 1's complement and add 1, or use method-2 from before), we'll get $-N$ in 2's complement representation. (I think?).
- A number's complement's complement is equal to the number itself.

0.3 Representing real numbers

0.3.1 The fixed point representation system

We define a constant number of digits to the left and to the right of the "dot".

Our number is of the type: $b_i b_{i-1} ... b_2 b_1 b_0 . b_{-1} b_{-2} ... b_{-j}$, for predefined i, j.

Its value:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

- Example: We have 8 bits to represent a number. The fraction point is between the 4th and 5th bits. The combination 10101100 represents the number: $(1010.1100)_2 = 10.75$
- Addition is performed normally, and the fixed dot is added afterwards (it doesn't matter).
- We use fixed point when:
  - There's no Floating Point Unit (FPU) in the CPU.
  - The desired accuracy is bounded.
  - In cases where it's more efficient to use fixed point.
- Disadvantage: The method is inefficient when we need to represent a wide range of numbers.

0.3.2 The floating point representation system

Representing a number in the floating point system (with base r) is done as follows:

Divide the bit sequence into two segments, called a **mantissa** and **exponent.**

Exponent (E)	Mantissa (M)

The number's value is

$$N = M \times r^E$$

(for a binary base: $r = 2$)

- We say that a number's floating point representation is **normalized** if $1 \leq Mantissa < base$.
Unnormalized	Normalized
2997.9×10^5	2.9979×10^8
$B1.39FC \times 16^{11}$	$B.139FC \times 16^{12}$
0.01011×2^{-1}	1.011×2^{-3}
  - In a binary base: The number is normalized if $1 \leq |M| < 2$, that is, $M = 1.....$.
- A number's representation in floating point is **biased** with bias B if the value of the represented number is given by $N = M \times r^{(E-B)}$.
  - Usually, we take $B = 2^{|E|-1}$ or $B = 2^{|E|-1} - 1$.
- Note that when the mantissa is normalized, we can compare exponents in order to compare two numbers to see which is bigger. (Not sure about that? TODO)

0.3.3 IEEE 754 ("Aye triple-E")

- The mantissa is represented with a sign bit in a normalized form. The exponent is biased with a bias of $B = 2^{|E|-1} - 1$.
S	E	F
  - S is the sign bit.
  - E is the exponent.
  - F is the fractional part.
- The number's value is $N = (-1)^S \times 2^{E-B} \times 1.F$
  - **Unless we're on the special case in which $E = 00...0$ or $E = 11...1$. More details in the part about Denormalized IEEE.**
- The standard dictates "special values": For instance, the number zero is represented by $E = F = 0$.

Single precision (float) - 32 bit:

- 23 bits for the mantissa (F).
- 1 bit for the mantissa's sign (S).
- 8 bits for the exponent (E).
  - $B = 2^{|E|-1} - 1 = 127$
  - The actual exponent value ranges from -127 for $E = 00000000$ to 128 for $E = 11111111$. However, the IEEE standard limits us to $-126 \leq exponent \leq +127$, since exponent $-127, +128$ have special meanings.
- Since the mantissa is assumed to be 1.*something*, the 23 bits we keep is only the *something* and therefore we actually have the power of 24 bits at our disposal. The accuracy here is $\frac{24}{log_2 10} \approx 7.2$ so the precision is that of 7 decimal digits.

Double precision (double) - 64 bit:

- 52 bits for the mantissa (F).
- 1 bit for the mantissa's sign (S).
- 11 bits for the exponent (E).

8-bit IEEE-like format:

- 4 bits for the mantissa (F).
- 1 bit for the mantissa's sign (S).
- 3 bits for the exponent (E).
- The value of the number is $N = (-1)^S \times 2^{E-B} \times 1.F$
- Bias: $B = 2^{|E|-1} - 1 = 2^{3-1} - 1 = 3$
  - Therefore, $N = (-1)^S \times 2^{E-3} \times 1.F$
- Example: 01101010. $S = 0$, $E = 110$, $F = 1010$. $N = (-1)^0 \times 2^{110-011} \times 1.1010 = (13)_{10}$
  - Note that since we had $(2^3)_{10} \times (1.1010)_2$, we can just push the dot 3 slots to the right: $(1101.0)_2$ and get to $(13)_{10}$ easily.

Converting a decimal number to floating point:

Example (using 8-bit IEEE-like):
Convert the number -2.5 to floating point.
$(-2.5)_{10} = (-10.1)_2$
Normalize: $-10.1 = -1.01 \times 2^1$
$S = 1$
$F = (0100)_2$
$(E - 011)_2 = (1)_{10} \Rightarrow (E)_2 = (4)_{10} \Rightarrow E = (100)_2$
So we got:

S	E	F
1	100	0100

Addition of floating point numbers:

1. Find the number with the lower exponent.
2. Convert this number's representation, so that its exponent equals the larger number's exponent (by dividing the mantissa by the appropriate value).
3. Add up the mantissas.
4. Normalize the result (if necessary, delete a bit from the right).

   - Examples: Tirgul 2, slides 27-32.

Normalized numbers:

- Condition: $E \neq 00...0$ and $E \neq 11...1$
- Exponent coded as biased value:
  - $Exp = E - Bias$
  - $B = 2^{|E|-1} - 1$
- Mantissa: $1.F$

Denormalized:

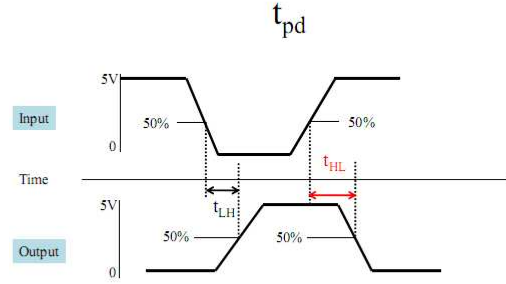
- Condition: $E = 00...0$
- Value:
  - $Exp = -Bias + 1$
  - $M = 0.F$
- Cases:
  - $E = 00...0$, $F = 00...0$ represents value 0.
  - $E = 00...0$, $F \neq 00...0$ numbers very close to 0.

Special values:

- Condition: $E = 11...1$
- Cases:
  - $E = 11...1$, $F = 00...0$ represents value ∞. An operation that has overflown.
  - $E = 11...1$, $F \neq 00...0$ represents "not a number" (NaN). For example, $\sqrt{-1}$.

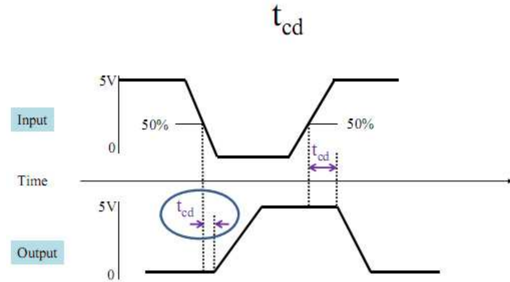
0.4 Circuit timing

0.4.1 Combinatinoal circuit timings

- t_{LH} - זמן השהייה מכניסה ליציאה כאשר היציאה "עולה" ($0 \leftarrow 1$).
  - נמדד בד"כ מהרגע בו הכניסה הגיעה ל-50% בין 0 ל-1 עד הרגע בו היציאה הגיעה לאותו ערך של 50%.
- t_{HL} - זמן השהייה מכניסה ליציאה כאשר היציאה "יורדת" ($1 \leftarrow 0$).
- 

- t_{pd} - Propagation delay :

$$t_{pd} = \max(t_{HL}, t_{LH})$$

- t_{cd} - זמן "זיהום" - משך הזמן מהרגע בו הכניסה החלה "להשתנות" (ערך 50%) אשר בו מובטח שהיציאה לא תשתנה.
  - **הזמן המקסימלי** בו מובטח לנו כי היציאה **טרם** השתנתה.
  - 

- זמני עליה, ירידה (Rise and fall):
  - t_r - זמן עליה - מהרגע בו המתח עבר 10% מכלל השינוי שאותו יעבור, עד שהגיע ל-90%.
  - t_f - זמן ירידה. כנ"ל, רק הפוך.
- t_{pd} **של מעגל**:
  - **הזמן המינימלי** בו מובטח לנו כי היציאה **כבר** השתנתה.
  - יש פה כל מיני דקויות, מומלץ לעיין בתרגול 7, שקפים 15 - 23.
  - אם רוצים לעבוד ברמה הכי מדויקת, צריך להסתכל על כל האופציות השונות של שינויים בערכי הכניסה, ולסכום את הערכים של t_{HL} או t_{LH} בהתאם, לכל רכיב במעגל. בנוסף צריך את הt_r (או t_f) של הרכיב הראשון ואת ה t_r (או t_f) של הרכיב האחרון במסלול, לסכום את שניהם ולחלק ב-2. כל זה מפורט יותר טוב בתרגול 7, שקפים 15-18. זה הזמן ה"אמיתי" המינימלי.
    אם רוצים הערכה יותר גסה, לוקחים רק את המקסימום בין ה-t_r (או t_f) -ים, במקום לקחת את שניהם ולחלק ב-2.
  - דרך אחרת לחשב - לוקחים את **המסלול האיטי ביותר**: בכל מסלול במעגל, לכל רכיב לוקחים את ה-t_{pd} שלו (שמוגדר כ-$\max(t_{HL}, t_{LH})$ של הרכיב), ובנוסף מוסיפים את המקסימום של ה-t_r, t_f של כל הכניסות והיציאות במעגל (במסלול?). סוכמים את זה ומקבלים את הזמן של המסלול. בסוף, הt_{pd} של המעגל הוא הזמן **המקסימלי בין כל המסלולים**.
    - * דוגמה: תרגול 7, שקפים 21-23.

* לפעמים גם לא מוסיפים את ה־t_r או t_f ומסתפקים בסכימת ה־t_{pd} של הרכיבים.

- t_{cd} **של מעגל**:
  - **הזמן המקסימלי** בו מובטח לנו כי יציאת המעגל **טרם** השתנתה.
  - כפי שניתן לראות בשקף 25 בתרגול 7, ישנם פקטורים ("R") שאנחנו לא יכולים לחשב במדויק, ולכן אנו מתעלמים מהם בחישוב: בשקף 26 אנו מתעלמים מערך זה, שכן הוא רק מוסיף זמן לt_{cd} ה"אמיתי", ובכך שאנו מסתכלים על ערך יותר נמוך אנחנו רק מגדילים את טווח הבטחון שלנו, ולא להיפך.
  - כדי למצוא את הt_{cd} של **השמה** במעגל: נחבר את כל ה־t_{cd} של הרכיבים במסלול הרלוונטי.
  - לבסוף, כדי למצוא את t_{cd} של **המעגל כולו**, נחפש מסלול **מינימלי (ה"מהיר ביותר") בין כל המסלולים**. ערך ה־t_{cd} של המעגל יהיה סכימת כל ה־t_{cd} של הרכיבים במסלול זה, כפי שאמרנו (מה שכתוב בסעיף הקודם).

0.4.2 Sequencial circuit timings

- מכיוון שאנחנו כעת מתעסקים במעגלים סדרתיים, נניח שלפליפ־פלופ שלנו יש כניסות: C לשעון, D לקלט, ויציאה Q. כמו כן, נסמן את הלוגיקה של המעגל ב־$Logic$.
- $t_{pd\,C\to Q}$ ־ זמן ההשהיה של הזיכרון מהמעבר האקטיבי של השעון עד להתייצבות יציאותיו.
- $t_{cd\,C\to Q}$ ־ משך הזמן מהמעבר האקטיבי של השעון אשר בו מובטח שהיציאה של ה־FF לא תשתנה.
- t_{setup} ־ משך הזמן שיש להחזיק את הכניסה של ה־FF קבועה **לפני** המעבר האקטיבי של השעון על מנת שהיציאות יהיו נכונות.
- t_{hold} ־ משך הזמן שיש להחזיק את הכניסה של ה־FF קבועה **אחרי** המעבר האקטיבי של השעון על מנת שהיציאות יהיו נכונות.
- $T_{min}, T_{hold}, F_{max}$:
- T_{min} הוא הזמן המינימלי שבו המעגל ירוץ. F_{max} הוא התדר המקסימלי שבו המעגל ירוץ. T_{hold} הוא למעשה t_{hold} של הפליפפלופ (כנראה).
  - $T_{min} = t_{pd\,C\to Q} + t_{pd}(Logic) + t_{setup}$
    * בדרך כלל יחידות הזמן של הנתונים הן ב־$nSec$, ובמקרה זה $F_{max} = \frac{1}{T_{min}} \cdot 1000$ הוא ב־MHz (כנראה?).
  - $T_{hold} < t_{cd\,C\to Q} + t_{cd}(Logic)$ (זה משהו שדורשים כדי שהמעגל יהיה תקין, זאת לא הגדרה).
- עד מתי ניתן לשנות את הקלט? ("לכמה זמן לפחות צריך להחזיק את הקלט קבוע לפני ירידת השעון?")
  - לפני ירידת השעון, הכניסה D חייבת להיות יציבה ומוכנה לדגימה. הקלט חייב להתייצב עד לזמן $t_{pd}(Logic) + t_{setup}$ לפני ירידת השעון.
- מתי הפלט של המעגל יציב? ("כמה זמן מירידת השעון צריכים לחכות כדי לקרוא את הערך החדש של היציאה?")
  - אחרי ירידת השעון, היציאה של רכיב הזכרון תשתנה. כמה זמן צריך לחכות עד שהיא תתייצב? $t_{pd\,C\to Q} + t_{pd}(Logic)$.

0.5 Cache

- Average memory access time (AMAT):

$$AMAT = Hit\,time + Miss\,penalty \times Miss\,rate$$

  - Hit time: The time to find and retrieve data from the current level cache.
  - Hit rate: % of requests that are found in current level cache.
  - Miss rate: $1 - Hit\,rate$.
  - Miss penalty: The average time to retrieve data on a current level miss (includes the possiblity of misses on successive leves of memory hierarchy).

- Analyzing multi-level cahce hierarchy:

$$AMAT = L1\,Hit\,time + L1\,Miss\,rate \times L1\,Miss\,penalty$$

and $L1\,Miss\,penalty$ is defined as:

$$L1\,Miss\,Penalty = L2\,Hit\,time + L2\,Miss\,rate \times L2\,Miss\,penalty = AMAT_{L2}$$

and so on, we continue with the cache levels.

- From the 2009B exam:
  - A system has main memory of size A MB, cache that is n-way of size B KB, a block size of C words and a word size of D bytes.
    - Given an address X in the main memory, it will be mapped to the $X\,mod\,\left(\frac{2^{10}\cdot B}{nCD}\right)$ set of the cache.
    - The number of bits needed for the "tag" is $log_2\left(n\cdot\frac{2^{10}\cdot A}{B}\right)$.
  - Explanation for the above:
    - The address X will be mapped to the $X\,mod\,\#sets$ set. The number of sets is the size of the cache ($2^{10}\cdot B$ bytes), divided by the size of the block, the size of a word, and the associativity factor n.
    - The number of bits needed for the tag is equal to log of the maximal number of different elements in main memory that may be mapped to the same cache set. To calculate this number, take $\frac{A\cdot 2^{20}}{B\cdot 2^{10}}$ which is the size of the main memory in bytes, divided by the size of the cache in bytes. This gives us $\frac{2^{10}\cdot A}{B}$. But since we must also take into account the associativity, we multiply by n.

0.6 Boolean algebra

0.6.1 Axioms

- סגירות:
  - $x + y \in B$
  - $x \cdot y \in B$
- איבר יחידה:
  - $x + 0 = x$
  - $x \cdot 1 = x$
- חילוף:
  - $x + y = y + x$
  - $x \cdot y = y \cdot x$
- פילוג:
  - $x \cdot (y + z) = x \cdot y + x \cdot z$
  - $x + (y \cdot z) = (x + y) \cdot (x + z)$
- משלים:
  - $x + x' = 1$
  - $x \cdot x' = 0$
- גודל:
  - $|B| \geq 2$

0.6.2 Claims

- אידמפוטנטיות
  - $x + x = x$
  - $x \cdot x = x$
- אסוציאטיביות:
  - $(x + y) + z = x + (y + z)$
  - $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- צמצום:
  - $x + xy = x$
  - $x(x + y) = x$
- דה־מורגן:
  - $(x + y)' = x' \cdot y'$
  - $(xy)' = x' + y'$
- כללים שונים:
  - $x + 1 = 1$
  - $x \cdot 0 = 0$
  - $(x')' = x$

0.6.3 Useful equivalences

- "**A xor B**" is equivalent to **ab' + a'b**
- "**not p**" is equivalent to "**p NAND p**"
- "**p and q**" is equivalent to "**(p NAND q) NAND (p NAND q)**"
- "**p or q**" is equivalent to "**(p NAND p) NAND (q NAND q)**"
- "**not p**" is equivalent to "**p NOR p**"
- "**p and q**" is equivalent to "**(p NOR p) NOR (q NOR q)**"
- "**p or q**" is equivalent to "**(p NOR q) NOR (p NOR q)**"

0.6.4 Basic truth tables

AND

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

NOT

A	Not A
0	1
1	0

NAND

A	B	A nand B
0	0	1
0	1	1
1	0	1
1	1	0

NOR

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

XOR

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Part III
Other things

0.6.5 Binary subtraction

Works like regular subtraction. Put the numbers one on top of the other and subtract. In case we have to do "$0-1$", we write 1 as the result, and "borrow" a 1 from the next digit to the left. This means that we should subtract and extra 1 from the digit to our left. For example:

$$\begin{array}{rr} & 1000000 \\ - & 0101100 \\ \hline & 0010100 \end{array}$$