TSEA83
¬DDR — Designspec.

Version b19c3a246d74aeed45de8421e6ea2309db0517ee

Alexander Basa (aleba538) Emil Draws (emidr065)
Hugo Hörnquist (hugho389)

15 juni 2018

Innehåll

Denna datorn är en två-stegs pipelinad dator. Detta ger möjligheten att utföra hämtning och exekvering på endast en klockcykel. Programminnet och dataminnet är implementerade separat vilket gör att programminnet endast kan läsas och inte skrivas till.

1 Allmänt

1.1 Avbrott

Avbrott sköts genom en MUX placerad mellan programminnet och instruktionsregistret. Avbrottet skickas som en signal till muxen som då istället för att utföra den utpekade instruktion utför en förprogrammerad subrutin som kommer hantera den signalerade händelsen. De olika avbrotten kan komma från tangentbordet eller inläsningen av data från någon port från kortet.

1.2 Tiles & Text

Om programmet ska innehålla text skapas den genom tiles. Ett komplett alfabete behöver inte finnas. T.ex. om enbart bokstäverna `MISGODPERFCT` finns kan orden “MISS“, “GOOD“, “PERFECT“, och “SCORE“ alla bildas. Med lite eftertanke kan antalet sprites där med minimeras.

Tilestorlek Varje tile har en storlek av 32×32 pixlar. Vilket på en skärm med upplösningen 640×480 ger det oss 20×15 tiles på skärmen.

Tile lista Åtminstone ska hela alfabetet (stora bokstäver), siffrorna 0-9, samt ett antal bilder för att kunna rita bakgrunder finnas.

1.3 Låtar

Låtarna laddas in via bussen och sparas i dataminnet. Den innehåller information om vilken not, på vilket instrument och dess speltid. Samt information om hur långt det är mellan noternas start tid.

1.4 Sprites

De fallande noterna i spelet kommer att implementeras som sprites. Antalet sprites som kan existera kommer att begränsas av hur stora vårt spriteminne kommer att vara och vi kommer i själva programmet att skriva det så att denna gräns aldrig överskrids. Dessa fallande noter kommer att representeras som någon geometrisk form i någon viss färg för varje not och detta utseende kommer att hårdkodas i spriteminnet i VGA motorn.

1.5 Ljud

Man har en ljudmotor man skriver till direkt via bussen. Målet är att datan ska skickas som "spela den här tonen på den här kanalen med det här instrumentet så här länge". Vi siktar på att ha 6 kanaler, varje kanal har 4-bits upplösning. Vi har det antalet kanaler med det antalet bitar för att vår DAC har som högs 24-bits upplösning. Det ljud som skickas till vår DAC är helt enkelt summan av alla kanalers värde.

Kanalernas uppläggning kan ses i blockschemat i figur 1.

1.6 Anropsstack

Datorn ska även ha en anropsstack för att tillåta implementation av subrutiner. Stacken kommer placeras någonstans där den inte riskerar att skriva över andra minnesplatser. Mest sannolikt längst ner i minnet. Stackpekaren kommer att hålla reda på stackens nuvarande position.

1.7 Instruktion

De instruktioner vilka sparas i programmminnet är alla 32 bitar breda. Bitarna har mest signifikant bit först. De första fem bitarna används för OP-koden, följande fem är registret vilken data sparas i. Resterande 22 är för instruktionsspecifik data.

2 Komponenter

Följande är en beskrivning av alla datorns komponenter vilka kan ses i figur 2. Matchande komponent i bilden hittas genom att ta rubrikens position, och jämföra med bilden som om den vore två kolumner.

2.1 M1

Hantering av huruvida programräknaren ska stegas, stå kvar eller förflyttas vid ett hopp.

2.2 PC

En simpel programräknaren.

2.3 PMEM

Programminnet, innehåller assembly-koden. Read only och flashas direkt in på chipet.

2.4 M2

MUX för att hantera NOP's vid vissa instruktioner (Eftersom datorn är pipelinad) samt avbrott.

2.5 IR

Instruktionsregistret, håller nuvarande instruktion som ska köras. Förutom decodern kan IR även skriva till bussen, det för att kunna placera hopp-adresser i samma cykel.

2.6 DECODER

Decodern är ett kombinatoriskt nät som hanterar instruktioner och är komponenten som bestämmer vad som ska göras för varje instruktion. Den har ansvar för att skriva och läsa data från bussen, bestämma ALU operation, med mera.

2.7 MILLI CLOCK

Extra klocka utöver systemklockan. Används framförallt för timing inom låtar. Håller internt ett 32-bitars tal. Programmet kan få ut de undre 16 bitarna på bussen, vilket är antalet millisekunder sedan klockan nollställdes, samt de övre 16 bitarna. Vilket är ca antalet minuter sedan klockan nollställdes.

Går även att nollställa den här klockan. En funktion som framförallt används för att kunna ha timing per låt.

2.8 SOUND MOTOR

Ljudmotorn ska fungera så att man skickar MIDI liknande info till den och sedan så sköter den uppspelningen av ljudet.

Formatet på det som skickas till ljud motorn:

- **kanal** 4 bitar
- **ton** 4 bitar
- **instrument** 4 bitar
- **längd** 16 bitar

} totalt 28 bitar

2.9 KEYBOARD

Läser av data från tangentbordet, skicka KBD-INT, samt skriver tangentbordsdatan till bussen.

Se labb 4 [1] ur labbserien för en ungefärlig idé om hur den fungerar.

2.10 DMEM

Dataminne. Början av minnet håller låt-information. Slutet av minnet innehåller anropsstacken. Minnet däremellan kan nyttjas till godtyckliga värden. Kommer troligen inte att finnas någonting som skyddar de tre minnesrymderna från varandra men vi kommer att anpassa storleken så att dessa inte kommer att skriva över varandra.

2.11 AREG

Allmänna register, dock är vissa register “mer allmänna“ än andra. Exakt antal register är fortfarande obestämd. Lika så exakt vilka som har vilken extrafunktion. Åtminstone ska dock följande extra funktioner finnas.

A Går till ALU’n

B Går till ALU’n

C Används för adressering av DMEM

2.12 ALU

Den aritmetiska enheten. Ska åtminstone ha stöd för de operationer som nämns i stycket om aritmetiska assemblerinstruktioner (3.3).

2.13 SR

Flaggor vilka sätts i ALU’n för att visa vad som hänt med numret. Används i första hand för vilkorliga hopp.

`N` = 0 Negative

`C` = 1 Carry

`Z` = 2 Zero

2.14 SP

Stackpekare, nyttjas för att kunna göra subrutinshopp.

2.15 PICT MEM

Bildminnet innehåller vilka typer av tiles som ska ritas ut var på skärmen. Utseendet för alla tiles hittas i tileminnet som finns i VGA-motorn. Bildminnet kan både läsa och skriva från VGA-motorn.

2.16 VGA MOTOR

Timing och utritning av grafik. Se labb 4 [1] i labbserien för en ungefärlig idé om dess konstruktion. Dock kommer vår version också ha sprites.

Hur de implementeras är vi ännu inte säkra på

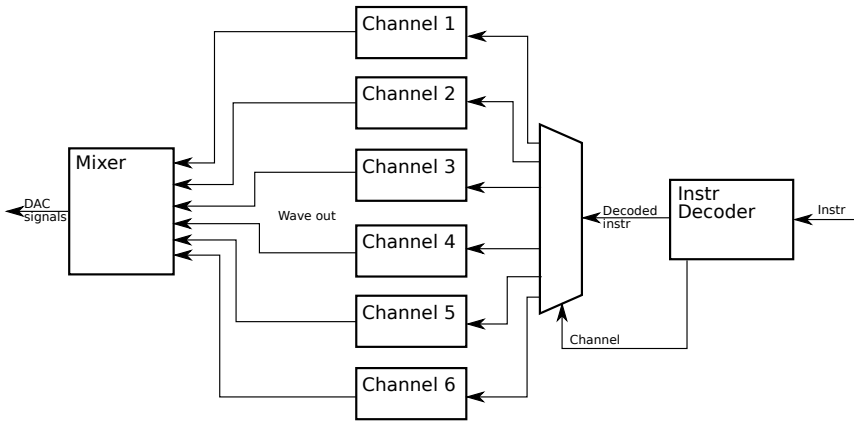
2.17 SPRITE REG X & Y

Spriteminnena kommer att innehålla x och y koordinater för alla sprites samt vilken typ av utseende de ska ha.

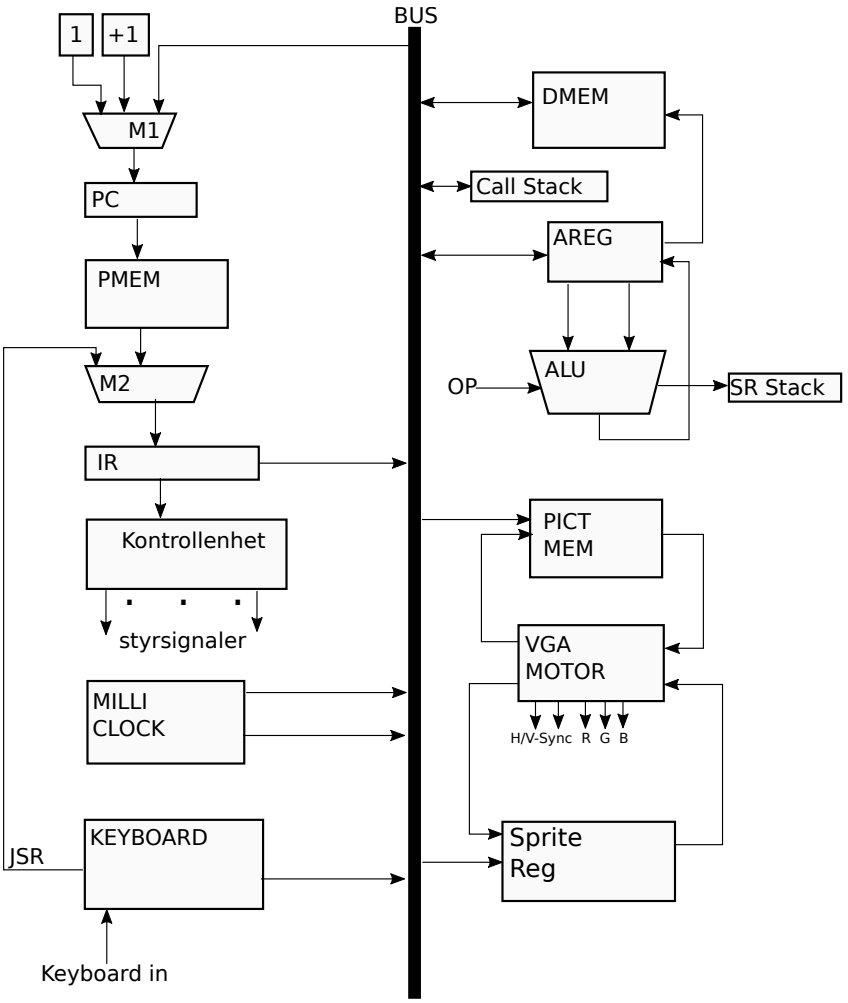
2.18 SERIAL DECODE

Avkodar seriell data, används för att läsa in låtar.

Avkodaren kommer att fungera genom att en ASM-instruktion säger att en låt ska läsas in, för att sedan låta "centraldatorn" sova. Komponenten läser in ord för ord, och sparar i DMEM (samt muterar AREG[C]) som den kommer att ha direkt tillgång till. Varje ord kommer ha någon form av start och slut, med 16 data-bitar emellan. När datan når ett specifikt slutvärde (t.ex. `0000000000000000`) så avbryts den seriella avkodningen och väcker "centraldatorn" igen. Schemat för komponenten kommer att vara väldigt lik avkodaren i labb 3.[2]



Figur 1: Blockschema för ljudmotorn



Figur 2: En två-stegs pipelinad dator

3 Assembly

Följande är en lista över de instruktioner som vår maskinkod implementerar.

OP-koder

Hex	Binärt	Instruktion
00_{16}	00000_2	`HALT`
10_{16}	00010_2	`LOAD`
18_{16}	00011_2	`LOADI`
20_{16}	00100_2	`STORE`
28_{16}	00101_2	`STORI`
30_{16}	00110_2	`LSLI`
38_{16}	00111_2	`LSRI`
40_{16}	01000_2	`JSR`
48_{16}	01001_2	`RTS`
50_{16}	01010_2	`ADD`
58_{16}	01011_2	`ADDI`
60_{16}	01100_2	`SUB`
68_{16}	01101_2	`SUBI`
70_{16}	01110_2	`MUL`
78_{16}	01111_2	`MULI`
$B8_{16}$	10111_2	`CMP`
$C8_{16}$	11001_2	`CMPI`
$D8_{16}$	11011_2	`CMPJ`
80_{16}	10000_2	`AND`
88_{16}	10001_2	`ANDI`
90_{16}	10010_2	`OR`
98_{16}	10011_2	`ORI`
$A0_{16}$	10100_2	`XOR`
$A8_{16}$	10101_2	`XORI`
$B0_{16}$	10110_2	`NOT`
$C0_{16}$	11000_2	`JMP`
$D0_{16}$	11010_2	`JEQ`
$E0_{16}$	11100_2	`JGE`
$F8_{16}$	11111_2	`NOOP`
$E8_{16}$	11101_2	*unused*
$F0_{16}$	11110_2	*unused*

3.1 Systemkontroll

`HALT` $\longmapsto$ Avbryt programexekeveringen.

`NOOP` $\longmapsto$ Gör ingenting.

3.2 Register- och minnestillgång

Läser och skriver till och från DMEM, flera av instruktionerna tar dessutom ett M argument, vilket är adresseringsmod. Vilka adresseringsmod som ska finnas är ännu inte bestämt. R_a & R_b är godtyckliga register.

`LOAD` R_c, M, ADR $\Big| \mathrm{R}_a \longmapsto$

$$\rightarrow \mathrm{R}_c := \mathrm{DMEM}\left(\begin{cases} \mathrm{ADR} & \text{om } M = \texttt{DIR} \\ \mathrm{R}_a & \text{om } M = \texttt{PTR} \end{cases}\right)$$

`STORE` R_c, M, ADR $\Big| \mathrm{R}_a \longmapsto$

$$\rightarrow \mathrm{DMEM}\left(\begin{cases} \mathrm{ADR} & \text{om } M = \texttt{DIR} \\ \mathrm{R}_a & \text{om } M = \texttt{PTR} \end{cases}\right) := \mathrm{R}_c$$

`LOADI` R_c, $n \longmapsto \mathrm{R}_c := n$

`STORI` ADR, $n \longmapsto \mathrm{DMEM}(\mathrm{ADR}) := n$

3.3 Aritmetik

Många av följande sätter flaggor i `SR`. Det är inte i alla fall noterat. Utgå från stycke 2.13 för vilka flaggor som finns, och härled därifrån.

`ADD` R_c, R_a, $\mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a + \mathrm{R}_b$

`SUB` R_c, R_a, $\mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a - \mathrm{R}_b$

`CMP` R_a, $\mathrm{R}_b \longmapsto \mathrm{R}_a - \mathrm{R}_b$
Inget värde returnernas, inget register sätts. Flaggor i `SR` sätts.

`CMPI` R_a, $n \longmapsto \mathrm{R}_a - n$

`CMPJ` n, $\mathrm{R}_a \longmapsto n - \mathrm{R}_a$

`ADDI` R_c, R_a, $n \longmapsto \mathrm{R}_c := \mathrm{R}_a + n$

`SUBI` R_c, R_a, $n \longmapsto \mathrm{R}_c := \mathrm{R}_a - n$

`MUL` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a \cdot \mathrm{R}_b$

`MULI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \cdot n$

`AND` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a \wedge \mathrm{R}_b$

`ANDI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \wedge n$

`OR` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto \mathrm{R}_c := \mathrm{R}_a \vee \mathrm{R}_b$

`ORI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \vee n$

`XOR` $\mathrm{R}_c, \mathrm{R}_a, \mathrm{R}_b \longmapsto Rc := \mathrm{R}_a \veebar \mathrm{R}_b$

`XORI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto Rc := \mathrm{R}_a \veebar n$

`NOT` $\mathrm{R}_c, \mathrm{R}_a \longmapsto \mathrm{R}_c := \neg \mathrm{R}_a$

`LSLI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \ll n$

`LSRI` $\mathrm{R}_c, \mathrm{R}_a, n \longmapsto \mathrm{R}_c := \mathrm{R}_a \gg n$

3.4 Hopp

Subrutiner är implementerade genom en call stack (se 1.6), Villkorliga hopp sker genom att rätt adress laddas in i programräknaren ifall rätt flagga är satt eller inte satt.

`JMP` $\text{ADR} \longmapsto PC := \text{ADR}$

Ovillkorligt hopp.

`JEQ` $\text{ADR} \longmapsto$

$$\to PC := \begin{cases} \text{ADR} & \text{om } \texttt{ZERO} \text{ flaggan satt i } \texttt{SR} \\ PC + 1 & \text{övriga fall} \end{cases}$$

`JGE` $\text{ADR} \longmapsto$

$$\to PC := \begin{cases} \text{ADR} & \text{om } \texttt{GE} \text{ flaggan satt i } \texttt{SR} \\ PC + 1 & \text{övriga fall} \end{cases}$$

I praktiken hoppar den om $\mathrm{R}_b \geq \mathrm{R}_a$ vid jämförelse.

`JSR` $\text{ADR} \longmapsto$

$$\to PC := \text{ADR}, \quad \text{DMEM}(SP) := \text{ADR}, \quad \text{INC}(SP)$$

Ovillkorligt hopp till subrutin. Lägger returaddress på anroppsstacken.

`RTS` $\varnothing^1 \longmapsto \quad PC := \text{DMEM}(SP), \quad \text{DEC}(SP)$

Retur från subrutin.

3.5 Ljud

`SOUND` $\text{ADR} \longmapsto$ [Spelar Ljud]

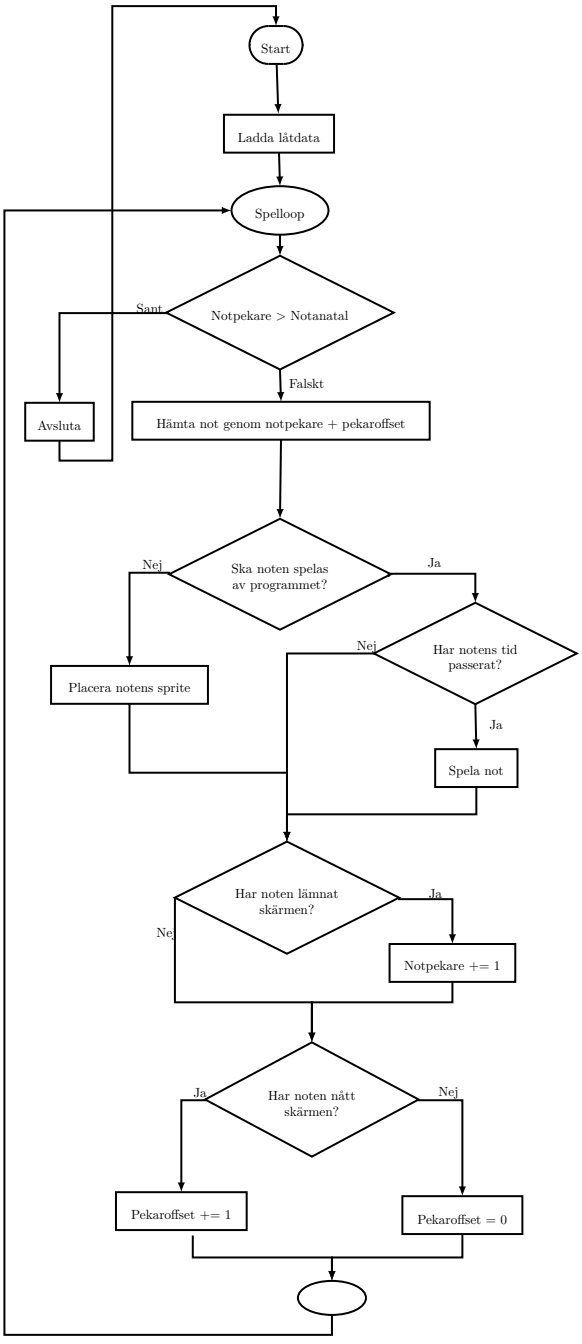
Ljudinstruktionen tar en pekare ADR vilket visar en plats i DMEM där musikdatan som ska spelas upp nu ligger. Se sektion 1.5 för detaljer över hur ljud specificeras.

4 Grovt flödesschema

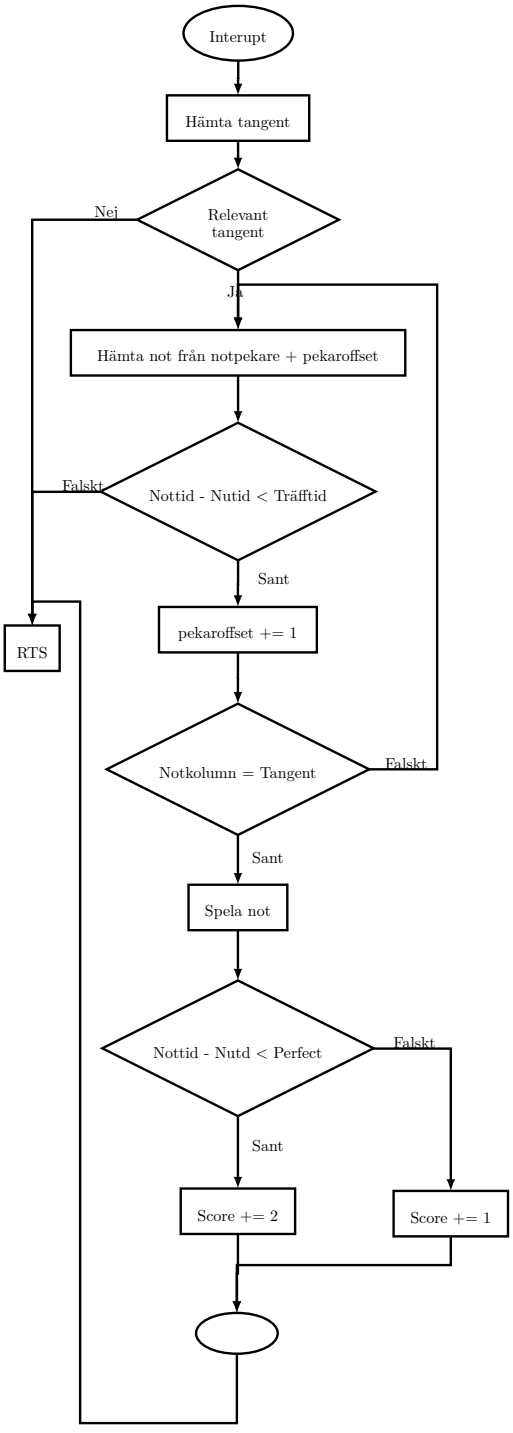
I figur 3 kan ett grovt flödesschema över programmet ses. Det kompleteras av flödesschemat i figur 4, vilket visar vad som ska hända när en tangentbordsinterrupt kommer.

Båda är ganska grova, men kommer troligen inte ändra sig mycket under proektets gång, om allting går som planerat.

[1] Ingen indata



Figur 3: Grovt flödesschema över programflödet



Figur 4: Flödesschema för hantering av tangentbordsinput

Referenser

[1] Anders Nilsson Li.U, ISY. VGA-lab — TSEA83 datorkonstruktion. `http://www.isy.liu.se/edu/kurs/TSEA83/pdf/lab_vga.pdf`, 2016. Version 1.0, [Online; accessed 26-February-2018].

[2] AN (2016) Li.U, ISY ON (2013). UART — TSEA83 datorkonstruktion. `http://www.isy.liu.se/edu/kurs/TSEA83/laboration/lab_uart.html`, 2016. Version 1.1, [Online; accessed 21-Mars-2018].