VON unit Documentation

Christian Jenei

June 2021

Contents

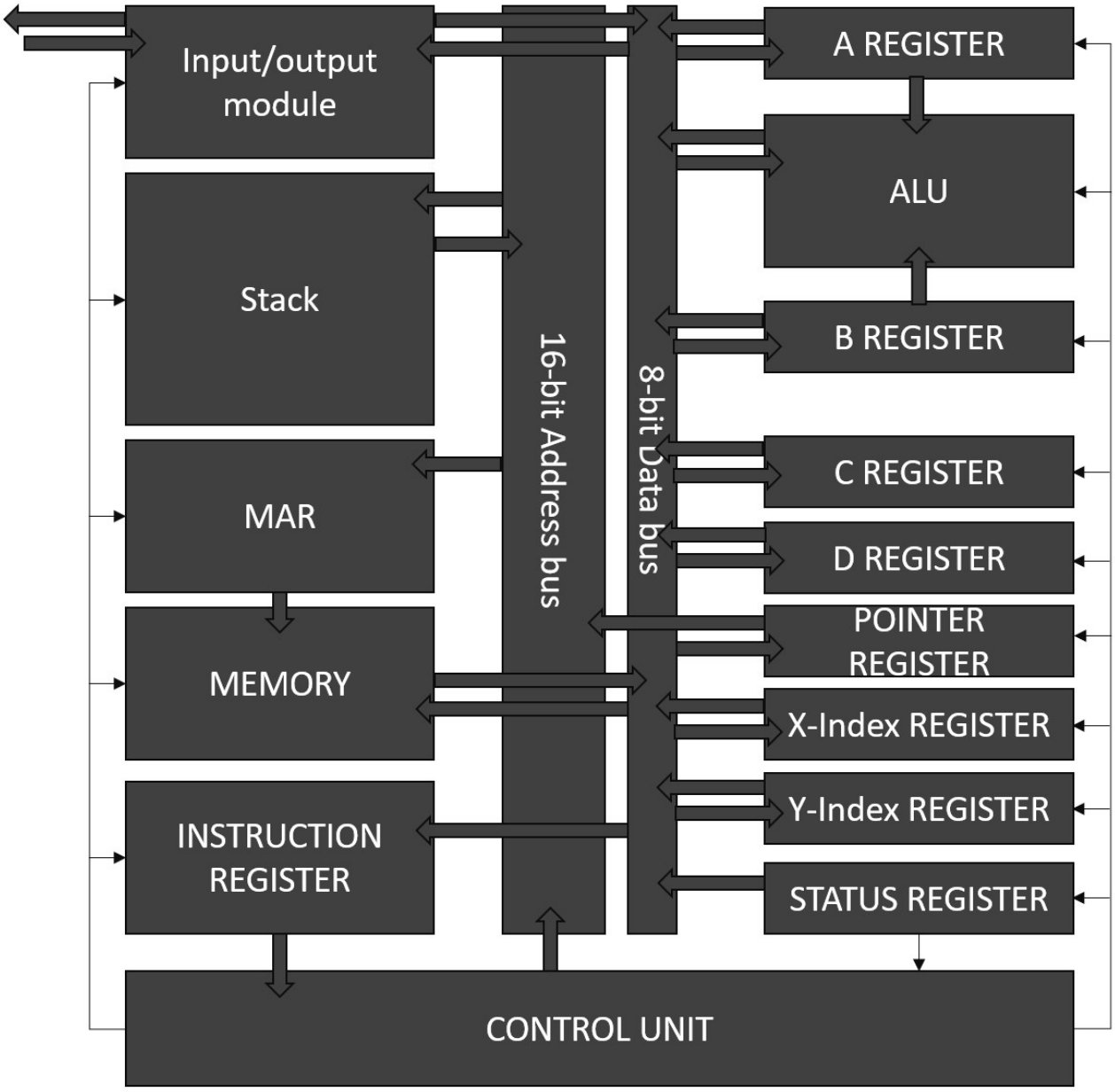
1 Introduction

The VON unit is a fully functional interactive CPU meant to display every interact process of a CPU. Its purpose is to educate students who are curious about the inner workings of a CPU. The VON unit includes a full blown schematic, Assembly language and a assembler. As the CPU is open and interactive, one can reprogram it's Control unit decoder

system to expand or recreate the instruction set. The VON unit includes features such as serial communication, a Stack, an ALU, index registers, IRQ handling and a fully re-programmable instruction set.

2 Overview

In this section we will give an overview of certain modules of the CPU and give a clearer understanding of different interactions throughout the CPU. Below is a block diagram of the whole VON unit CPU displaying the communication capabilities of every module. The VON unit is a 8-bit data, 16-bit addressable(memory is only 15 bit addressable) CPU. Different modules have the capabilities to send or receive data from the two existing buses. The address bus is used for memory addressing and storing pointers, while the data bus is used for data communication throughout the different modules. One can observe that the control unit has the capability to send out data to the address bus. This is only used for the IRQ handler bootstrap process(see 2.4 Serial communication (I/O module)) to load the Interrupt vector into the Stack. The control unit's essential job is read decode instructions from the instruction register with the integration of Status Flags and states to determine what Micro-Instruction needs to be executed. The control unit has the capability to perform many Micro-instructions that execute different logic in each module. One can get a better view and understandings of the interconnections of each module by viewing the block diagram below.



2.1 Tstates

Once the Instruction register has been loaded with an instruction, it's up to the control unit to decode the instruction. At this stage the control unit also takes into account what Tstate the machine is currently at to determine the right Micro-Instructions to execute. A Tstate is a way for the control unit to know what stage of the instruction has been executed and what Micro-Instructions currently need to be executed. The Tstates consist of a total of 6 states that go from T0-T5. These Tstates are generated within the control unit. For each Clock cycle, the Tstate gets increased by one. Usually the Tstates T0-T1 is the fetch cycle of the CPU and consists of loading the current instruction in memory into the Instruction register and making the machine ready for the next Instruction by increasing the address stored int he PC.The remaining Tstates are the "execution cycle". These Tstates are a way to organize what micro-instruction to execute in what order for each instruction.

2.2 ALU

The VON unit uses the SN74LS181N as its ALU, The SN74LS181N has 6 settings signals that determine what kind of operation currently should execute such as addition, subtraction, or XOR. these signals are labeled as S0, S1, S2, S3, S4, and S5 in the Micro-Instruction section. The following table should be used to decide what operation one would want to select.

SELECTION				ACTIVE-LOW DATA		
				S5 = H	S5 = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	S4 = L (no carry)	S4 = H (with carry)
L	L	L	L	$F = \overline{A}$	F = A MINUS 1	F = A
L	L	L	H	$F = \overline{AB}$	F = AB MINUS 1	F = AB
L	L	H	L	$F = \overline{A} + B$	F = $A\overline{B}$ MINUS 1	F = $A\overline{B}$
L	L	H	H	F = 1	F = MINUS 1 (2's COMP)	F = ZERO
L	H	L	L	$F = \overline{A + B}$	F = A PLUS (A + $\overline{B}$)	F = A PLUS (A + $\overline{B}$) PLUS 1
L	H	L	H	$F = \overline{B}$	F = AB PLUS (A + $\overline{B}$)	F = AB PLUS (A + $\overline{B}$) PLUS 1
L	H	H	L	$F = \overline{A \oplus B}$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	$F = A + \overline{B}$	F = A + $\overline{B}$	F = (A + $\overline{B}$) PLUS 1
H	L	L	L	$F = \overline{A}B$	F = A PLUS (A + B)	F = A PLUS (A + B) PLUS 1
H	L	L	H	$F = A \oplus B$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = $A\overline{B}$ PLUS (A + B)	F = $A\overline{B}$ PLUS (A + B) PLUS 1
H	L	H	H	F = A + B	F = (A + B)	F = (A + B) PLUS 1
H	H	L	L	F = 0	F = A PLUS A‡	F = A PLUS A PLUS 1
H	H	L	H	$F = A\overline{B}$	F = AB PLUS A	F = AB PLUS A PLUS 1
H	H	H	L	F = AB	F = $A\overline{B}$ PLUS A	F = $A\overline{B}$ PLUS A PLUS 1
H	H	H	H	F = A	F = A	F = A PLUS 1

The ALU also produces Status Flags on certain operations, these Status Flags can then be stored in the Status register via the FI Micro-Instruction. The ALU produces the flowing Status flags: A = B Flag(A=BF), A>B Flag(A>BF), A<B Flag(A<BF), Carry flag(CF) and the Zero Flag(ZF). For a more detail on the SN74LS181N see its data sheet.

2.3 Status Register

The Status register stores different flags and states used by the Control unit in order to determine instructions such as conditional jumps or interrupt request handling. In truth only some flags are stored in the register while others are just displayed at the physical location of the Status register on the PCB. The table below displays all the flags stored as well as the four remaining status flags that originates from the Index registers

Stored in Status register											
0	1	2	3	4	5	6	7	8	9	10	11
A<BF	A>BF	A=BF	ZF	CF	INT	HLT	IRQDIS	XinZF	XinCF	YinZF	YinCF

ALU Flags

bit 0 to 4 contains Flags that originate from the ALU. These have to be manually stored via the FI Micro-Instruction in order to be stored into the status register. This mechanism is to prevent interference once the Data contents of the A register has been replaced by the ALU's content.

A<BF indicates if the number stored in the B register is larger than the A register.
A>BF indicates if the number stored in the A register is larger than the B register.
A=BF indicates if the number stored in the B register is equal to the A register.
ZF indicates if the operation performed by the ALU between the A and B registers produced the number zero
CF indicates if an overflow has occurred under an operation performed by the ALU between the A and B registers

Interrupt Flags

The interrupt flags consist of bit 5 and 7. These flags dynamically change based upon IRQ signal given by the I/O module. The IRQDIS disables the INT flag but only comes into effect at T0 on words until the IRQDIS flag has been disabled. The INT flag is only produced if the following conditions are true:
1. IRQ signal from the I/O module is active.
2. IRQDIS flag is LOW
3. Current T state is T0

Once the INT flag is active, it stays on until the transition from T5 to T0 and will not be disturbed by IRQDIS activating nor IRQ deactivating. To further explain INT flags behavior, we observe the flowing two examples where T6 the transition state from T5 to T0:

INT example 1							
	T0	T1	T2	T3	T4	T5	T6
INT	L	L	L	L	L	L	L
IRQ	L	H	H	H	H	H	H
IRQDIS	H	H	L	L	L	L	L
INT example 1							
	T0	T1	T2	T3	T4	T5	T6
INT	H	H	H	H	H	H	L
IRQ	H	H	H	L	L	L	L
IRQDIS	L	H	H	H	L	L	L

HLT Flag

The HLT flag activates immediately as the HLT Micro-Instruction activates. Once the HLT flag has activated the Tstate halts to a stop leaving the machines to stop any execution at that current Tstate as well as not allowing the machine to proceed to the next Tstate.

Index Flags

The index flags indicate the state of the Index registers separately. These flags are not stored in the status register and therefore cannot output onto the data bus via the STRO Micro-Instruction. The zero flags of both index registers are connected directly to the control unit.

XinZF indicates if the contents of the X-index register equals zero
XinCF indicates if an overflow has occurred in the X-index register
YinZF indicates if the contents of the Y-index register equals zero
YinCF indicates if an overflow has occurred in the Y-index register

2.4 Serial communication (I/O module)

VON unit utilizes serial communication via an ACIA. The currently supported ACIA includes HD6350, HD6850 and the MC6850. Before serial communications can be initiated, one must set the right settings for the ACIA. For further information, read the desired ACIA's data-sheet. The supported ACIA:s can be initiated by the flowing code:

```
ICR 151
ICR 150
ICR 150
```

ICR loads an 8-bit number into the Control register of the ACIA.

The first number loaded by the ICR instruction is 10010111(151_{dec}), as the bit 0 and bit 1 are both high that tells the ACIA to do a master reset. The next line will then load in the number 10010110(150_{dec}), the first two bits 10 (bit0 and bit1) will set the divide ratio to 1/64, as the VON units I/O module uses a 7,3728Mhz clock the serial bit-rate will become 115200. The last line will load in the number 10010110(150_{dec}), the bit2 to bit4 is set to 101, this sets the ACIA encoding format to "8-bits + 1 stop bit", the remaining bits will set the right behavior for the IRQ signal such that the IRQ signal will be HIGH(Low from ACIA but gets inverted) when the ACIA has received data to be transmitted into the VON unit.

Once the ACIA has been configured via the ICR instruction, one can transmit data via the OUT instruction. This will transmit the current data stored in the A register.
If one wishes to add a new instruction that utilizes serial transmission, keep in mind that the EIO needs to be pulled HIGH then LOW in a pulse behavior while data flow is active at both states as the ACIA only transmits data at the HIGH to LOW state of EIO. This means that to transit data, one must have the following micro-instructions in both states:

Transmission pulse example			
T2	T3	T4	T5
XXXX	XXXX	TD, EIO	TD

The same EIO pulse behavior is necessary for any input or output from the ACIA. This includes the use of the Micro-Instructions: TD, RD, CR and SR.

2.4.1 Interrupt request handling

The Control unit only initiates the interrupt request handler´s bootstrap once the INT flag is HIGH. To understand the behavior of how the INT flag gets initiated refer to the Interrupt Flags subsection in the section about the Status register.
Once the INT flag has activated, the Control unit initiates the IRQ(Interrupt request) handler´s bootstrap which consists of the flowing set of Micro-instructions:

IRQ handler´s bootstrap					
T0	T1	T2	T3	T4	T5
PU	INTV, JMP	RD, EIO	RD		

This process bootstraps the actual IRQ handler by essentially doing a jump to subroutine to the predefined Interrupt Vector(changeable through two 8-bit dip switches) where the user will have their own desired IRQ handler programmed into the memory. The bootstrap process also disables the IRQ signal by relocating the contents of the ACIA's receive data register into the Input register. The content of the Input register can be loaded into the A register via the ITA Instruction. The data stored in the Input register is stored until the next INT flag gets initiated.

The VON unit has no reprogrammed IRQ handler and instead lets the user program it for its intended use. The IRQ handler should always be ending in a RFS(return from subroutine) instruction to return to the main program

3 Instruction set

Instruction set			
5-bit Instruction Word	Mnemonics	Description	Status Affected
00000	LDA	Load word into A register from specified address	
00001	LDB	Load word into B register from specified address	
00010	LDC	Load word into C register from specified address	
00011	LDD	Load word into D register from specified address	
00100	LDX	Load word into X-Index register from specified address	
00101	LDY	Load word into Y-Index register from specified address	
00110	LPR	Load word into Pointer register from specified address	
00111	ADD	Addition (load word into B from specified address then load (A+B) into A)	
01000	SUB	Subtraction (load word into B from specified address then load (A-B) into A)	
01001	XOR	XOR (load word into B from specified address then load (A $\oplus$ B) into A)	
01010	OUT	Transmit data from A via serial output	
01011	ITA	Move data from Input register into A register	
01100	STA	Store A register to memory location	
01101	JMP	jump to memory location	
01110	JAZ	Conditional jump to memory location	ZF
01111	JXZ	Conditional jump to memory location	XinZF
10000	JYZ	Conditional jump to memory location	YinZF
10001	JMS	Jump to subroutine	
10010	RFS	Return from subroutine	
10011	XIC	Increase X-Index register by 1	
10100	YIC	Increase Y-Index register by 1	
10101	XDC	Decrease X-Index register by 1	
10110	YDC	Decrease Y-Index register by 1	
10111	DIQ	Disable interrupt request (activates at T0)	
11000	HLT	Halt CPU	
11001	ICR	Load into Control Register(I/O module)	
11010	CMP	Compare A and B, if equal ZF will be activated	
11011	LDI	Load immediate into A register	
11100	N/A	Not assigned	
11101	N/A	Not assigned	
11110	N/A	Not assigned	
11111	N/A	Not assigned	

3.1 Instructions cycles

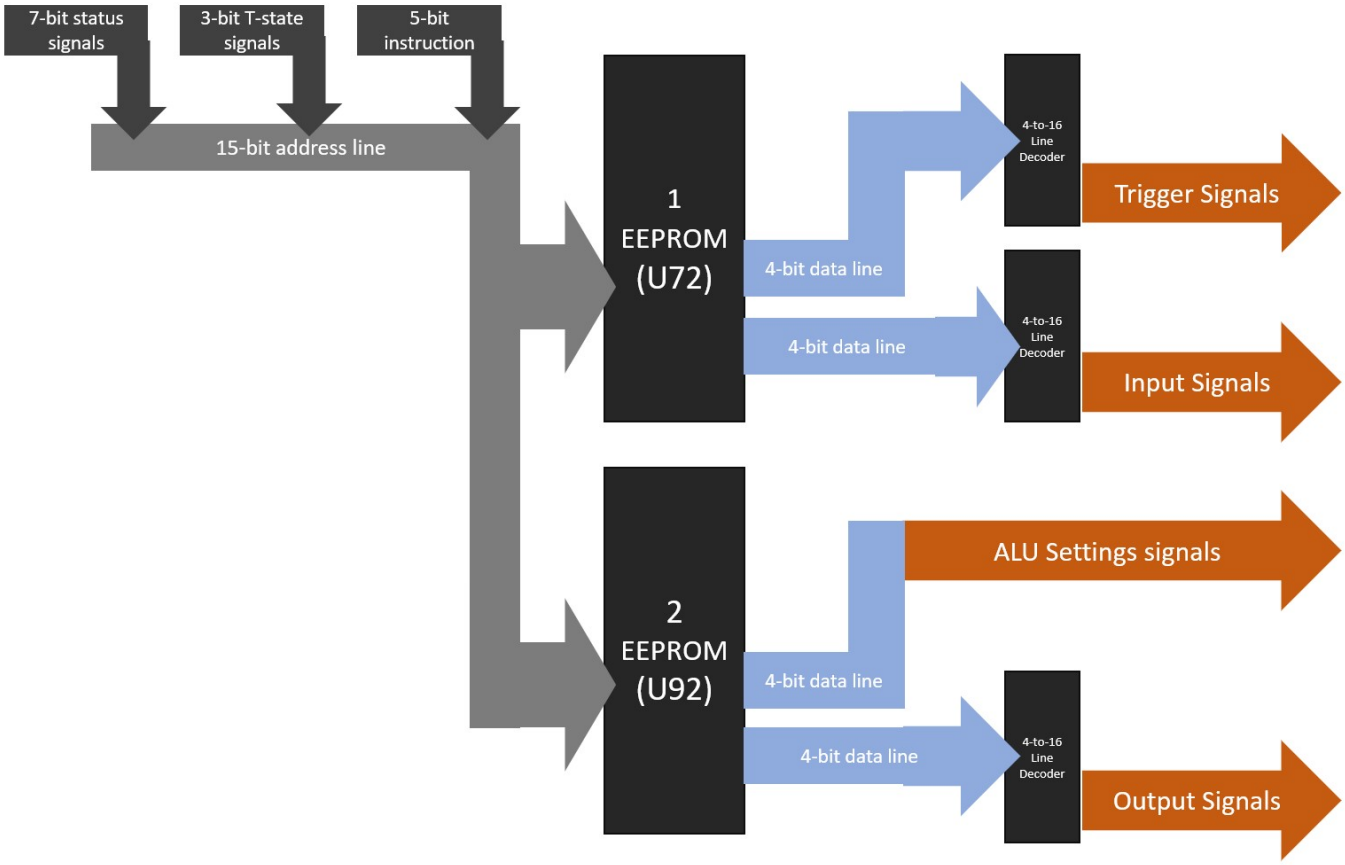
Every instruction consists of a set of Micro-Instructions that get executed at every "T" state. The table below presents the instruction executions of each Micro-Instruction and at what state it gets executed. One shall observe that no matter what instruction gets executed the first two states (T0 and T1) are always consistent. This is called "the fetch cycle". The fetch cycle's purpose is to load the next instruction in order to be executed. Execution of currently loaded instructions only occur in states T2 to T5.

Instruction set				
Mnemonics	T0	T1		
XXX	CO MI	RO II CE		
Mnemonics	T2	T3	T4	T5
LDA	PRO, MI	RO, AI		
LDB	PRO, MI	RO, BI		
LDC	PRO, MI	RO, CI		
LDD	PRO, MI	RO, DI		
LDX	PRO, MI	RO, XinI		
LDY	PRO, MI	RO, YinI		
LPR	CO, MI	RO, PR1I, CE	CO, MI	RO, PR2I, CE
ADD	ADO, MI	RO, BI	S3, S0, FI	S3, S0, AI, EO
SUB	ADO, MI	RO, BI	S2, S1, S5, FI	S2, S1, S5, AI, EO
XOR	ADO, MI	RO, BI	S2, S1, S4, FI	S2, S1, S4, AI, EO
OUT	EIO, AO, TD	AO, TD		
ITA	AI, SERIALO			
STA	PRO, MI	AO, RI		
JMP	PRO, JMP			
JAZ	PRO, JMP			
JXZ	PRO, JMP			
JYZ	PRO, JMP			
JMS	PU	PRO, JMP		
RFS	PD			
XIC	XinINC			
YIC	YinINC			
XDC	XinDEC			
YDC	YinDEC			
DIQ	IRQDIS			
HLT	HLT			
ICR	CO, MI	EIO, RO, CR	RO, CR, CE	
CMP	S2, S1, S4, FI			
LDI	CO, MI	RO, AI, CE		
N/A				
N/A				
N/A				
N/A				

3.2 Micro-Instructions

The decoding of each instruction with the combination of current state and status flags is handled by two EEPROM's that are located in the control unit. The instruction with the combination of current state and status flags make up an address from which the EEPROM's can decode the desired 8-bit data to be sent over to be decoded by three 4-to-16 Line Decodes to determine the right Micro-Instruction.

Each EEPROM's data output are split up into two 4-bit numbers for each 4-to-16 Line Decodes. This inherently means that one EEPROM can execute two Micro-Instructions at a time(with the exception of the second EEPROM that can execute a total of 5 Micro-instructions. (See block diagram below).



Each of these 4-bits are categorized by a certain type of Micro-Instruction. The first EEPROM (U72) contains all the Input Micro-Instructions as well as different Triggers such as Halt or Counter enable, while the second EEPROM (U92) contains the Output Micro-Instructions as well as the ALU's settings signals. The ALU's settings Section is special in this case as it is not connected to a 4-to-16 Line Decoder but rather directly to the EEPROM's output data lines. This means the second EEPROM (U92) is able to freely set each of the four ALU settings signals. As stated above, this means the second EEPROM (U92) can execute one Micro-Instruction form the Output section(see table below for EEPROM (U92)) and any of the ALU's settings lines(S0, S1, S2, S3) from the ALU Settings Section.

First decoder EEPROM (U72)		
Data	Micro-instruction name	Description
Input Section		
0000 0000	N/A	N/A
0000 0001	TD	Allow data flow to Transmission register in ACIA
0000 0010	CR	Allow data flow to Control register in ACIA
0000 0011	JMP	Load address from address bus into PC
0000 0100	MI	Load Data from data bus into MAR
0000 0101	FI	Load current ALU flags into Status register
0000 0110	RI	Load data from data bus into Memory
0000 0111	PR1I	Load data from data bus into first 8-bits into Pointer register
0000 1000	PR2I	Load data from data bus into Second 8-bits into Pointer register
0000 1001	AI	Load data from data bus into A register
0000 1010	BI	Load data from data bus into B register
0000 1011	CI	Load data from data bus into C register
0000 1100	DI	Load data from data bus into D register
0000 1101	II	Load data from data bus into Instruction register
0000 1110	XinI	Load data form data bus into X-index register
0000 1111	YinI	Load data form data bus into Y-index register
Triggers Section		
0000 0000	N/A	N/A
0001 0000	EIO	Enables ACIA. Used as confirmation pulse for TD,CR, RD and SR
0010 0000	IRQDIS	Disables/Enables Interrupt requests(initiates at state T0)
0011 0000	S4	ALU Mode, switches from arithmetic and logical operands (see ALU section for more)
0100 0000	N/A	N/A
0101 0000	HLT	Halt CPU
0110 0000	PU	Push stack up, select PC register above current one
0111 0000	PD	Push stack down, select PC register under current one
1000 0000	CE	Counter enable, count PC up by one
1001 0000	XinINC	Increase X-index register by 1
1010 0000	XinDEC	Decrease X-index register by 1
1011 0000	YinINC	Increase Y-index register by 1
1100 0000	YinDEC	Decrease Y-index register by 1
1101 0000	S5	ALU carry, Switches type of arithmetic operation table
1110 0000	N/A	N/A
1111 0000	N/A	N/A

Second decoder EEPROM (U92)		
Data	Micro-instruction name	Description
Output Section		
0000 0000	N/A	N/A
0000 0001	RO	Output current addressed memory data onto data bus
0000 0010	PRO	Output Pointer register onto data bus
0000 0011	AO	Output A register onto data bus
0000 0100	BO	Output B register onto data bus
0000 0101	CO-	Output C register onto data bus
0000 0110	DO	Output D register onto data bus
0000 0111	EO	Output ALU register onto data bus
0000 1000	XinO	Output X-index register onto data bus
0000 1001	YinO	Output Y-index register onto data bus
0000 1010	STRO	Output Status register onto data bus
0000 1011	SERIALO	Output Input register onto data bus
0000 1100	RD	Allows data flow from ACIA's receive register onto input register
0000 1101	SR	Allows data flow from ACIA's Status register onto input register
0000 1110	INTV	Output Interrupt vector onto Address bus
0000 1111	CO	Output current selected PC register onto data bus
ALU Settings Section		
0000 0000	N/A	N/A
0001 0000	S0	ALU select option (see ALU section for more information)
0010 0000	S1	ALU select option (see ALU section for more information)
0100 0000	S2	ALU select option (see ALU section for more information)
1000 0000	S3	ALU select option (see ALU section for more information)

4 VON Unit Assembly Language

The VON unit utilizes a custom assembly language designed specifically for its unique instruction set and architecture. To facilitate programming the VON unit, a Python-based assembler is provided, which translates assembly code into a binary `.bin` file suitable for loading into the VON unit's memory.

This document serves as a comprehensive guide to understanding the assembly language syntax, features, and how to use the Python assembler to create and upload programs to the VON unit.

4.1 Introduction

Programming the VON unit involves writing assembly code that conforms to its instruction set architecture (ISA). The provided Python assembler simplifies this process by handling many low-level details, allowing programmers to focus on the logic of their programs.

The VON unit features a modular memory system, enabling users to write programs externally and upload them via the provided assembler. For manual programming, the VON unit's PCB includes a programming mode accessible by setting the mode switch to `M`. Users can select the desired address using the Memory Address Register (MAR), input data via DIP switches, and program it into memory using the `PROG` button.

4.2 Addressing and Instruction Set Overview

The VON unit uses a 16-bit addressing scheme (15 bits in memory), with an 8-bit data bus. Due to this architecture, certain instructions require the use of the Pointer Register (`PR`) to reference memory addresses.

Many instructions, such as `LDA`, `STA`, `ADD`, `SUB`, `JMP`, and others, operate on the address stored in the `PR`. The `LPR` (Load Pointer Register) instruction is used to load addresses into the `PR`. The Python assembler simplifies this process by automatically inserting `LPR` instructions when necessary, based on the operands provided.

It's important to note that the `LPR` instruction occupies three memory locations: one for the opcode and two for the 16-bit address (stored in little-endian format).

4.3 Assembly Language Syntax

The assembly language syntax is designed to be straightforward and easy to read. Each line of assembly code can contain:

- An optional **label**
- An **instruction** (mnemonic) or assembler directive
- An optional **operand**
- An optional **comment** (following a semicolon `;`)

Here's the general structure:

```
[label:]    instruction    [operand]    ; comment
```

- **Labels**: Identifiers followed by a colon (`:`) used as targets for jump and branch instructions.
- **Instructions**: Mnemonics representing operations (e.g., `LDA`, `ADD`, `JMP`).
- **Operands**: Can be immediate values, memory addresses, or labels.
- **Comments**: Text following a semicolon (`;`) is ignored by the assembler.

4.3.1 Instructions and Operands

Immediate Instructions Instructions like `LDI` (Load Immediate) require an immediate value as an operand.

```
LDI 42          ; Load the value 42 into register A
```

Memory Access Instructions Instructions such as `LDA`, `STA`, `ADD`, `SUB`, etc., operate on the memory address pointed to by the `PR`. When an operand (e.g., a label) is provided, the assembler automatically inserts an `LPR` instruction to load the address into the `PR`.

Example:

```
LDA VALUE       ; Load the value at address 'VALUE' into register A
```

The assembler expands this into:

```
LPR VALUE       ; Load address of 'VALUE' into PR
LDA             ; Load from memory[PR] into A
```

Jump and Branch Instructions Instructions like `JMP`, `JAZ`, `JMS`, etc., require an address in the `PR`. The assembler handles the insertion of `LPR` as needed.

Example:

```
JMP START       ; Jump to label 'START'
```

The assembler expands this into:

```
LPR START       ; Load address of 'START' into PR
JMP             ; Jump to address in PR
```

4.3.2 Assembler Directives and Pseudo-Instructions

Data Definitions

- `DB`: Define a byte.

```
COUNT: DB 0     ; Define a byte at label 'COUNT' initialized to 0
```

Pseudo-Instructions

- `PRINT`: Outputs a string to the serial output.

```
PRINT "Hello World"
```

  The assembler expands this into a series of `LDI` and `OUT` instructions.

- `PRINTLN`: Outputs a string followed by a newline (Carriage Return and Line Feed).

```
PRINTLN "Hello World"
```

4.3.3 Comments

Comments are added using the semicolon `;`. Everything following the semicolon on that line is ignored by the assembler. Example:

```
LDI 10      ; Load value 10 into A
```

4.4 Using the Python Assembler

The Python assembler (`assembler.py`) translates assembly code into a binary `.bin` file compatible with the VON unit.

4.4.1 Steps to Assemble a Program

1. **Write the Assembly Code**: Create a text file with your assembly code, following the syntax outlined above. Save it with a `.asm` extension (e.g., `program.asm`).
2. **Run the Assembler**:

```
python assembler.py program.asm program.bin
```

   - `program.asm`: Your assembly code file.
   - `program.bin`: The output binary file to be loaded into the VON unit's memory.

3. **Load the Binary File into the VON Unit**:

   Use the provided tools or methods to load `program.bin` into the VON unit's memory (e.g., via EEPROM programming).

4.5 Example Program

Below is an example assembly program that demonstrates various features of the VON unit's assembly language and instruction set.

```
; Example Program for the VON Unit
; Demonstrates basic operations, loops, subroutines, and I/O

; Initialize serial communication
ICR 151          ; Initialize serial communication
ICR 150
ICR 150

START:
    ; Print "Hello World"
    PRINTLN "Hello World"

    ; Load immediate value into A
    LDI 42          ; A = 42
    ; Store A into memory location NUM1
    LPR NUM1
    STA             ; Memory[PR] = A

    ; Load immediate value into A and store into NUM2
    LDI 17          ; A = 17
    LPR NUM2
    STA             ; Memory[PR] = A

    ; Perform addition: SUM = NUM1 + NUM2
    LPR NUM1
    LDA             ; A = NUM1
    LPR NUM2
    LDB             ; B = NUM2
    ADD             ; A = A + B
    LPR SUM
    STA             ; SUM = A

    ; Output the sum
    LPR SUM
    LDA             ; A = SUM
    LPR ASCII_OFFSET
    LDB             ; B = ASCII offset ('0')
    ADD             ; Convert to ASCII character
    OUT             ; Output character

    ; Print newline
    LDI 13          ; Carriage Return (CR)
    OUT
    LDI 10          ; Line Feed (LF)
    OUT

    ; Loop demonstration using COUNTER
    LDI 5           ; A = 5
    LPR COUNTER
    STA             ; COUNTER = 5

LOOP_START:
    ; Check if COUNTER == 0
    LPR COUNTER
```

```
    LDA             ; A = COUNTER
    LPR ZERO
    LDB             ; B = 0
    CMP             ; Compare A and B
    LPR LOOP_END
    JAZ             ; Jump if A == B

    ; Output 'X'
    LDI 'X'
    OUT

    ; Decrement COUNTER
    LPR COUNTER
    LDA             ; A = COUNTER
    LPR ONE
    LDB             ; B = 1
    SUB             ; A = A - B
    LPR COUNTER
    STA             ; COUNTER = A

    ; Jump back to LOOP_START
    LPR LOOP_START
    JMP

LOOP_END:
    ; Call subroutine to print a message
    LPR SUBROUTINE
    JMS

    ; Halt the program
    HLT

; Subroutine to print a message
SUBROUTINE:
    PRINTLN "Subroutine Called!"
    RFS

; Data definitions
NUM1:
    DB 0            ; Variable to store NUM1
NUM2:
    DB 0            ; Variable to store NUM2
SUM:
    DB 0            ; Variable to store SUM
COUNTER:
    DB 0            ; Loop counter
ZERO:
    DB 0            ; Constant 0
ONE:
    DB 1            ; Constant 1
ASCII_OFFSET:
    DB 48           ; ASCII offset for '0'
```

4.5.1 Explanation

- **Initialization**: Serial communication is initialized using `ICR` instructions.
- **Printing**: The `PRINTLN` pseudo-instruction is used to output "Hello World".

- **Data Manipulation**: Values are loaded into registers and stored in memory.
- **Arithmetic Operations**: The program performs addition and stores the result.
- **Looping**: A loop is created using `JMP` and `JAZ` to output 'X' five times.
- **Subroutine**: A subroutine is called using `JMS` and returns with `RFS`.
- **Data Definitions**: Variables and constants are defined using `DB`.

5 Micro-Instruction Assembler

As the VON unit is quite modular, one can reprogram the control unit to establish their own Instruction set. There is an included Micro-Instruction assembler that can be used to generate the code for both chips (U72 and U92). To add instructions, one has to use the assembler and edit the Instruction matrix shown below

```
Instructions = ["DC", "CO|MI", "RO|II|CE","ADO|MI","RO|AI","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|BI","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|CI","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|DI","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|XinI","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|YinI","0","0";
                "DC", "CO|MI", "RO|II|CE","CO|MI","RO|AD1I|CE","CO|MI","RO|AD2I|CE";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|BI","S3|S0|FI","S3|S0|AI|EO";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|BI","S2|S1|S5|FI","S2|S1|S5|AI|EO";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","RO|BI","S2|S1|S4|FI","S2|S1|S4|AI|EO";
                "DC", "CO|MI", "RO|II|CE","EIO|AO|TD","AO|TD","0","0";
                "DC", "CO|MI", "RO|II|CE","AI|SERIALO","0","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|MI","AO|RI","0","0";
                "DC", "CO|MI", "RO|II|CE","ADO|JMP","0","0","0";
                "ZF", "CO|MI", "RO|II|CE","ADO|JMP","0","0","0";
                "XinZF", "CO|MI", "RO|II|CE","ADO|JMP","0","0","0";
                "YinZF", "CO|MI", "RO|II|CE","ADO|JMP","0","0","0";
                "DC", "CO|MI", "RO|II|CE","PU","ADO|JMP","0","0";
                "DC", "CO|MI", "RO|II|CE","PD","0","0","0";
                "DC", "CO|MI", "RO|II|CE","XinINC","0","0","0";
                "DC", "CO|MI", "RO|II|CE","YinINC","0","0","0";
                "DC", "CO|MI", "RO|II|CE","XinDEC","0","0","0";
                "DC", "CO|MI", "RO|II|CE","YinDEC","0","0","0";
                "DC", "CO|MI", "RO|II|CE", "IRQDIS","0","0","0";
                "DC", "CO|MI", "RO|II|CE", "HLT","0","0","0";
                "DC", "CO|MI", "RO|II|CE","CO|MI", "EIO|RO|CR","RO|CR|CE","0";
                "DC", "CO|MI", "RO|II|CE","S2|S1|S4|FI","0","0","0";
                "DC", "CO|MI", "RO|II|CE","CO|MI","RO|AI|CE","0","0";
                "DC", "CO|MI", "RO|II|CE","0","0","0","0";
                "DC", "CO|MI", "RO|II|CE","0","0","0","0";
                "DC", "CO|MI", "RO|II|CE","0","0","0","0";
                "DC", "CO|MI", "RO|II|CE","0","0","0","0";];
```

Each row represents an instruction and the row position corresponds to the numerical value of the instruction. for example row 0 is LDA for the current instruction set and its numerical value is "00000" in binary. The first column represents what Flag the Instruction should be active on. The remaining columns represents the Tstates T0-T5. Every Micro-Instruction that should be executed at each Tstate needs to be separated by a | character. Keep in mind that only one Micro-Instruction from the Input, Trigger and Output Section may be executed per Tstate. Meanwhile, any number of Micro-Instructions from the ALU section may be executed per Tstate.
Once the desired Instruction set has been established, run the Micro-instruction assembler and it should generate two .BIN files. The chip1.bin corresponds to the U72 EEPROM while the chip2.bin corresponds to the U92 EEPROM.