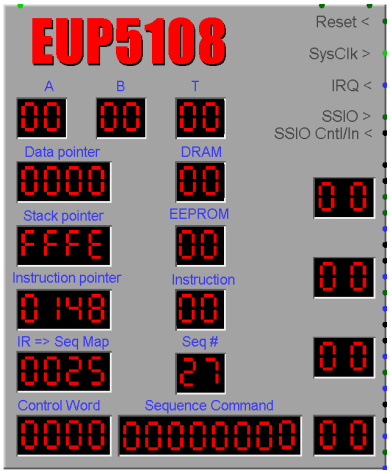
EUP5108 User's Guide

A Retro 8-bit, Scratch-built, Minimalist, Micro-controller

Designed from the ground up and tested in simulation using

https://github.com/logisim-evolution/logisim-evolution

Features:



- ✓1 bit serial ALU – 8 & 16 bit calculations w/ same hardware
- ✓Complete 256 CISC TLA Instruction Set
- ✓Bit level Test, Set, and Clear instructions
- ✓Direct loading of the most common constants (x00, x01, & xFF)
- ✓Conditional long and short jumps on Carry, Not Zero, & Borrow flags
- ✓Dual software, single maskable hardware, and single NMI interrupts
- ✓8 bit data and 16 bit little endian address bus
- ✓Fully addressable 64KB of EPROM & 64KB DRAM
- ✓4 Bi-directional 8 bit data ports, one of which supports Synchronous Serial IO

Boot Sequence:

Upon boot or reset the EUP5108 clears the A, B, T, DR, and IP registers. The stack pointer is initialized to xFFFF. Hardware IRQ is masked. The IP is loaded with the reset vector, and instruction processing proceeds from that point.

IRQ Sequence:

When an IRQ is raised the current instruction is completed. The current values of A,B,T, DR, and IP are pushed onto the stack, the IP is loaded with the IRQ vector located at the base of EPROM according to the table below and instruction processing proceeds from that point.

Vector	EPROM Location	Typical
Reset	x00 x01	x08 x00
HW IRQ	x02 x03	x00 xD0
SW IRQ	x04 x05	x00 xE0
SW IRQ2	x06 x07	x00 xF0

Note: For proper operation the EPROM reset vector should be set to 0x0008 or higher (little endian) and the IRQ vectors set to a location containing at least a HLT or RTI instruction.

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com

Method of Operation

The EUP5108 Operates using a series of lookup tables held in three ROMs within the Instruction Decode and Sequencer Logic (IDSL) module.

The Sequencer ROM (256x32bit) is addressed using the Sequence Counter (8bit) and each value from the Sequencer ROM is used to drive control lines as well as the expression of each nibble of the Control Word (16bit). At reset the Sequence Counter is 0x00 and the Boot Sequence is followed until the first instruction is loaded.

The IRSeqMap ROM (256x16bit) is addressed using the loaded instruction and its output value is used to provide the ALU command (4bits) & ALU Inputs (4bits) for the instruction along with setting the Sequence Counter (8bits) to the location in the Sequencer ROM used to process that class of instructions. While there are 256 instructions only 30 instruction sequences are used to process all of them.

The IRQSeqMap ROM (8x16bit) is only enabled when an IRQ is pending and is addressed by the active IRQ bits. When the current instruction reaches completion, and loads the next instruction, the output from the IRQSeqMap is used instead of the IRSeqMap ROM output for that instruction thus initiating the IRQ call sequence.

The above could, and would normally, be simplified into a single larger ROM were it to be developed as a single chip. I'm using separate standard size ROMs in this design so it can be more easily built using ICs

Design Considerations

The EUP5108 was designed from the ground up based only my experience and vague memories of the features and operation of the Intel 8051 Micro-controller. As my first exposure to embedded systems some 30+ years ago the 8051 represents a significant turning point in my software career

The feature list above was the extent of the system requirements that I started with. Learning Logisim-Evolution along the way I first developed the ALU, then the Serial Registers, and a Microcode execution unit that was too small for a CISC system and was closer to the simulated RISC processor used in my advanced CS course(s) (circa 1993). It was after reading the biography of the Intel 8051's designer that I switched from defining the instruction set on the hardware and instead designed the hardware to match an instruction set.

Instruction Set Design & Layout

Fundamentally a CPU has three jobs:

Copy data from one location to another

Combine two values via some function and place the result in one location or the other

Manage system operations such as NOP, SysHalt, IRQ control

There are fewer locations used for calculation than there are in the system over all. Only the A (1byte),B (1byte),T (1byte), and DR (2bytes) registers can be accessed by the ALU whereas the remaining locations, ROM (1byte), RAM (1byte), Stack Pointer (2bytes), Instruction Pointer (2bytes), along with the 4 I/O ports (1byte ea) brings the total number of possible locations to 15.

To copy a byte from one location to another first the Output Enable (OE) of the source location must be asserted so as to place the location's value onto the shared data bus. Then the Load/Write Enable (WE) line of the destination location is asserted to complete the copy.

With 15 possible locations it was only natural to break down the Instruction into high and low nibbles. The low nibble contains the location whose output is to be enabled and the high nibble defines which location to Load/WE. This gives us an instruction numbering pattern where the first hex digit is the destination of the operation and the second hex digit is the source. This accounts for the majority of instructions except along the diagonal where the source and the destination are the same. These diagonal instructions are used for system operations and some ALU functions. Doing this for all 15 locations however leaves no space in the instruction table for the rest of the ALU and Bit functions.

To make space for these other instructions it made sense to remove some uncommon/illogical memory transfer operations such as directly from one I/O Port to the next and transfers between the instruction pointer and stack pointer, as well as directly between RAM and ROM.

At this point the instruction set is pretty much stable but alternate instructions could be developed and used in place of the L*Z, L*O, & L*H instructions (replaced with L*E instructions). Other less useful instructions include LRP, LDP, LRI, LDI, LRK, LDK, LRS, LDS, LKR, LSR, LKD, & LSD

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com

ROM Layouts

IRSeqMap (256 x16bit) & IRQSeq ROM (8 x 16bit)

	ALU Function	ALU Dst/Src	Sequencer Start Address	
msb	4 Bits	2 Bits ea	8 Bits	lsb
	16 Bits			

Example entry for the A=A+B (EAB) instruction

0x5	b0001	0x72
0x5172		

Sequencer ROM (256 x 32 bit)

					IRH out x1000	ALU Func out x0100	Dec SP x0010	Load IR x0001	
					Use IRH Const x2000	Use ALU Func Const x0200	Inc SP x0020	Inc IP x0002	
					IRL out x4000	Dst/Src out x0400	ALU Reset x0040	Toggle DR/SP x0004	
	IRL Const	IRH Const	ALU Dst/Src Const	ALU Func Const	Use IRL Const x8000	Use Dst/Src Const x0800	Sys Ops x0080	Toggle Bank x0008	
msb	4 Bits	4 Bits	4 Bits	4 Bits	4 Bits	4 Bits	4 Bits	4 Bits	lsb
	32 Bits								

Example sequence used for all ALU logic and Math Functions

Set up the ALU to perform a new calculation by presenting the Dst/Src and ALU Reset

Step 0	0x0	0x0	0x0	0x0	0x0	0x4	0x4	0x0
						Dst/Src Output	Hold ALU Reset	

Release ALU reset, output ALU Function & Dst/Src Triggering ALU busy.
In parallel, configure the data bus with ROM OE and Increment IP

Step 1	0x1	0x0	0x0	0x0	0xC	0x5	0x0	0x2
	OE ROM				Output IRL using IRL Const	Dst/Src Output, ALU Func Output	Release ALU Reset	Inc IP

While keeping the bus configured for ROM OE, load the next instruction into the IR

Step 2	0x1	0x0	0x0	0x0	0xC	0x0	0x0	0x1
	OE ROM				Output IRL using IRL Const			Load IR

System Operations

The selection of which system operation to execute is controlled through the ALU Dst/Src nibble

NOP 0x0	HLT 0x1	IQR 0x2	IQC 0x3	IQU 0x4	IQM 0x5	DES 0x6	INS 0x7
IQ1 0x8	IQ2 0x9	RST 0xA	TDS 0xB			ALU Ret Flags	ALU Sto Flags

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com

Version 1.0 Release Notes

Not all instructions have been implemented as indicated by a 00 in their sequence number. This was intentional so that I can produce future videos showing the addition and debug of these instructions. I have also written a C++ based software emulation that reads all the same .rom files but operates at much greater clock rate. For example, using my primary development system the Logisim-Evolution circuit model only runs at about 500hz whereas the C++ command line program can go as high as 20+Mhz (Processor AMD A10-7700K Radeon R7, 10 Compute Cores 4C+6G, 3400 Mhz, 2 Core(s), 4 Logical Processor(s)). While this is great for writing complex ASM programs, such as an assembler, it does not allow for connection to the Logisim I/O devices. Next on the list is the porting of the C++ version to JAVA so as to make it a loadable device library that is instruction set compatible but will run much faster.

The .rom and .asm files were written in Notepad++ (https://notepad-plus-plus.org/) in which I have defined a language for the syntax coloring (EUPASMSyntaxFile)

The .Rom file format is built upon the workings of the “Hex Word Addressed” file format of Logisim: Lines starting with '#' are ignored, as is everything beyond two sequential white space characters. Lines starting with '##' are treated as single line comments – this is just for highlighting purposes.

Currently the programs are written using intermixed machine code with ASM code and C equivalent code as comments.

BuildIt.bat first concatenates all the .ASM files into one .ROM file. Then the .ROM file is filtered for “: *” to create the .MAP file that is used to look up function and symbol information.

```
#############################################################
## Print a string of characters to Port1 from EEPROM
## Takes advantage of the hardware increment in the SP
## rather than adding 1 to DR each time
# 0F00: *PrintStr1E(DR)
        IQM Mask IRQs since we are using SP for other stuff
0F00: 55
        XDS swap(DR,SP)
0F01: 21
        TDS Toggle(DR/SP)
0F02: 12
  |pb LBF B=EROM(SP)
0F03: 42
        W1B Port1=B
0F04: D4
        JSN if(B) goto |kp
0F05: 82 0B
        TDS Toggle(DR/SP)
0F07: 12
        XDS swap(DR,SP)
0F08: 21
        IQU unmask IRQs
0F09: 44
        RTL Return
0F0A: 97
  |kp INS SP++
0F0B: 77
        JPS goto |pb
0F0C: 80 03
```

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com

EUP5108 ASM Code Grid

Mnemonic Pattern

L	A	E
Action	Dest	Source

Load into the A register the contents of Eprom (byte following instruction) a.k.a. Load Immediate

		OE														
		DRAM (DR)	EROM (IP)	EROM (DR|SP)	A	B	T	R	D	P	I	K	S	P0	P1	P2	P3	
Load/WE		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
DRAM (DR)	0	NOP	POE *	PSD *	SIA	SIB	SIT	SIR	SID	SIP	SII	SIK	SIS	RI0	RI1	RI2	RI3	0
DRAM (SP)	1	PDO *	HLT	TDS	SCA	SCB	SCT	SCR	SCD	SCP	SCI	SCK	SCS	RC0	RC1	RC2	RC3	1
EROM (DR|SP)	2	PDS	XDS *	IQR	SEA	SEB	SET	SER	SED	Sep	SEI	SEK	SES	RE0	RE1	RE2	RE3	2
A	3	LAM	LAE	LAF	IQC	LAB	LAT	LAR	LAD	LAP	LAI	LAK	LAS	LA0	LA1	LA2	LA3	3
B	4	LBM	LBE	LBF	LBA	IQU	LBT	LBR	LBD	LBP	LBI	LBK	LBS	LB0	LB1	LB2	LB3	4
T	5	LTM	LTE	LTF	LTA	LTB	IQM	LTR	LTD	LTP	LTI	LTK	LTS	LT0	LT1	LT2	LT3	5
R	6	LRM	LRE	LRF	LRA	LRB	LRT	DES	DXR *	LRP ?	LRI ?	LRK ?	LRS ?	BT0	BT1	BT2	BT3	6
D	7	LDM	LDE	LDF	LDA	LDB	LDT	LDR *	INS	LDP ?	LDI ?	LDK ?	LDS ?	BS0	BS1	BS2	BS3	7
P	8	JPS	JSC	JSN	JSR	JSB	JST	CAS	RTS	IQ1	MBA	MTA	MDA	TLA	EBA	ETA	EDA	8
I	9	JPL	JLC	JLN	JLR	JHB	RTI	CAL	RTL	MAB	IQ2	MTB	MDB	EAB	TLB	ETB	EDB	9
K	A	LKM	LKE	LKF	LKA	LKB	LKT	LKR ?	LKD ?	MAT	MBT	RST	MDT	EAT	EBT	TLT	EDT	A
S	B	LSM	LSE	LSF	LSA	LSB	LST	LSR ?	LSD ?	DEA	DEB	DED	BSA	INA	INB	IND	TLD	B
P0	C	W0M	W0E	W0F	W0A	W0B	W0T	BTA	RRA	AAT	ABA	OAT	OBA	XAT	XBA	NOA	NBA	C
P1	D	W1M	W1E	W1F	W1A	W1B	W1T	BTB	RRB	AAB	ABT	OAB	OBT	XAB	XBT	NAB	NOB	D
P2	E	W2M	W2E	W2F	W2A	W2B	W2T	BSB	RRT	LAO	LBO	LTO	LAH	LBH	LTH	BCA	NOT	E
P3	F	W3M	W3E	W3F	W3A	W3B	W3T	BCB	RRD	LAZ	LBZ	LTZ	LDZ	BC0	BC1	BC2	BC3	F
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

SYS OPS	DATA MOVES	BIT FUNCS	LOAD CONST	JMPS CALLS	DIFF	SUM	SHIFT LEFT	AND	OR	XOR	NOR	NOT	INC	DEC	ROT RIGHT
PTR COPY															

EUP5108 ASM Code Grid Notes

SYS OPS	System operations: No OPeration, HaLT clock, IrQ Raise hw, IrQ Clear, IrQ Unmask, IrQ Mask, DEcrement/INcrement the Stack pointer using hardware, IQ1 raise sw irq 1, IQ2 raise sw irq 2, RST reset, TDS TogDR/SP
DATA MOVES	Data move actions are: Load registers, Read/Write ports, Store to ram/rom. Loading a value into the B register resets the ALU Not Zero flag based on the value loaded. LDR loads the next two bytes (little endian) into the DR register in a single instruction. DXR swaps the DR hi & lo (endian swap). Single register stores into RAM/ROM based on SP (SC*) automatically decrement SP
BIT FUNCS	Bits are Set/Cleared/Tested using the next byte as a bit mask where the 1s are the bit(s) of interest. If any bit is set upon test then the Not Zero flag is set. Port bit functions overwrite the B register.
LOAD CONST	Load one of three built in constants: *Z*ero (x00), *O*ne (x01) & *H*igh (xFF)
JMPS	JumP Long, Jump Long if Carry, Jump Long if Not zero, and Jump Long if boRrow take the next two bytes as the destination address (little endian). JumP Short, Jump Short if Carry, Jump Short if Not zero, and Jump Short if boRrow take the next byte as the low byte of the destination address within the current high byte
CAL CAS	Long/Short function calls: The current IP (hi/lo for Long & just lo for Short) are pushed onto the stack and the next two bytes (for Long & 1 byte for Short) are used as the destination address (little endian).
RTL RTS	Long/Short function call Return: The return IP (hi/lo for Long & lo for Short) are popped off the stack and incremented to the instruction after CAL/CAS
RTI	Return from IRQ: Simular to RTS with the addition of setting A,B,T & DR to the values on the stack placed there when the IRQ was triggered.
DIFF	Minus (subtraction), also used to compare two values as the result will be Zero if identical.
SUM	∑ (addition)
SHIFT LEFT	shifT Left, a.k.a. Multiply by 2 (add value to itself)
AND / OR / XOR / NOR / NOT	Bitwise logical functions
INC / DEC	Increment and Decrement by adding/subtracting 1 using the ALU
ROT RIGHT	Circular rotate right
PTR COPY	PSD, PDS Copies the values of the DR(hi/lo) to the SP(hi/lo) or the SP(hi/lo) to the DR(hi/lo). PDO copies the SP to DR and adds A. POE copies the SP and adds the next byte in ROM XDS swaps the value of DR and SP.

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com

EUP5108 Instruction Information Grid (Alphabetical)

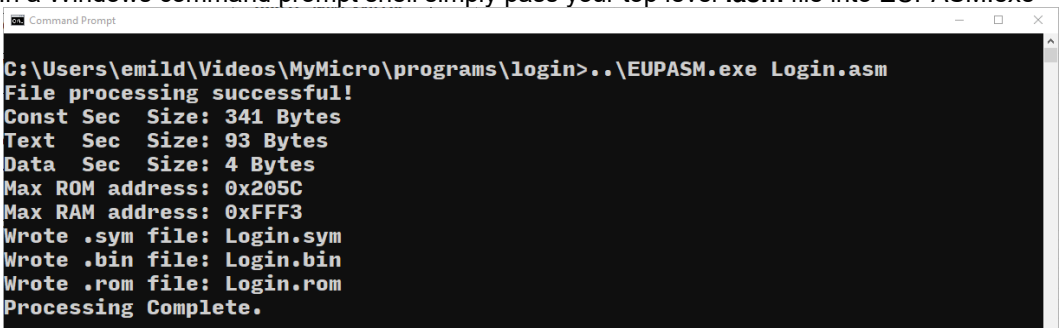
INS	OP	SIZE	CYCS	INS	OP	SIZE	CYCS	INS	OP	SIZE	CYCS	INS	OP	SIZE	CYCS
AAB	D8	1	0	LAB	34	1	0	LSD	B7	1	0	RTI	95	1	0
AAT	C8	1	0	LAD	37	1	0	LSE	B1	2	0	RTL	97	1	0
ABA	C9	1	0	LAE	31	1	0	LSF	B2	1	0	RTS	87	1	0
ABT	D9	1	0	LAF	32	1	0	LSM	B0	1	0	SCA	13	1	0
BC0	FC	2	0	LAH	EB	1	0	LSR	B6	1	0	SCB	14	1	0
BC1	FD	2	0	LAI	39	1	0	LST	B5	1	0	SCD	17	1	0
BC2	FE	2	0	LAK	3A	1	0	LT0	5C	1	0	SCI	19	1	0
BC3	FF	2	0	LAM	30	1	0	LT1	5D	1	0	SCK	1A	1	0
BCA	EE	2	0	LAO	E8	1	0	LT2	5E	1	0	SCP	1B	1	0
BCB	F6	2	0	LAP	38	1	0	LT3	5F	1	0	SCR	16	1	0
BS0	7C	2	0	LAR	36	1	0	LTA	53	1	0	SCS	1B	1	0
BS1	7D	2	0	LAS	3B	1	0	LTB	54	1	0	SCT	15	1	0
BS2	7E	2	0	LAT	35	1	0	LTD	57	1	0	SEA	23	1	0
BS3	7F	2	0	LAZ	F8	1	0	LTE	51	2	0	SEB	24	1	0
BSA	BB	2	0	LB0	4C	1	0	LTF	52	1	0	SED	27	1	0
BSB	E6	2	0	LB1	4D	1	0	LTH	DE	1	0	SEI	29	1	0
BT0	6C	2	0	LB2	4E	1	0	LTI	59	1	0	SEK	2A	1	0
BT1	6D	2	0	LB3	4F	1	0	LTK	5A	1	0	Sep	28	1	0
BT2	6E	2	0	LBA	43	1	0	LTM	50	1	0	SER	26	1	0
BT3	6F	2	0	LBD	46	1	0	LTO	EA	1	0	SES	2B	1	0
BTA	C6	2	0	LBE	41	2	0	LTP	58	1	0	SET	25	1	0
BTB	D6	2	0	LBF	42	1	0	LTR	56	1	0	SIA	03	1	0
CAL	96	3	0	LBH	EC	1	0	LTS	5B	1	0	SIB	04	1	0
CAS	86	2	0	LBI	49	1	0	LTZ	FA	1	0	SID	07	1	0
DEA	B8	1	0	LBK	4A	1	0	MAB	98	1	0	SII	09	1	0
DEB	B9	1	0	LBM	40	1	0	MAT	A8	1	0	SIK	0A	1	0
DED	BA	1	0	LBO	E8	1	0	MBA	89	1	0	SIP	08	1	0
DES	66	1	0	LBP	48	1	0	MBT	AB	1	0	SIR	06	1	0
DXR	67	1	0	LBR	46	1	0	MDA	8B	1	0	SIS	0B	1	0
EAB	9C	1	0	LBS	4B	1	0	MDB	9B	1	0	SIT	05	1	0
EAT	AC	1	0	LBT	45	1	0	MDT	AB	1	0	TDS	12	1	0
EBA	8D	1	0	LBZ	F9	1	0	MTA	8A	1	0	TLA	8C	1	0
EBT	AD	1	0	LDA	73	1	0	MTB	9A	1	0	TLB	9D	1	0
EDA	8F	1	0	LDB	74	1	0	NAB	DE	1	0	TLD	BF	1	0
EDB	9F	1	0	LDE	71	2	0	NBA	CF	1	0	TLT	AE	1	0
EDT	AF	1	0	LDF	72	1	0	NOA	CE	1	0	W0A	C3	1	0
ETA	8E	1	0	LDI	78	1	0	NOB	DF	1	0	W0B	C4	1	0
ETB	9E	1	0	LDK	7A	1	0	NOP	00	1	0	W0E	C1	2	0
HLT	11	1	0	LDM	70	1	0	NOT	EF	1	0	W0F	C2	1	0
INA	BC	1	0	LDP	78	1	0	OAB	DA	1	0	W0M	C0	1	0
INB	BD	1	0	LDR	76	3	0	OAT	CA	1	0	W0T	C5	1	0
IND	BE	1	0	LDS	7B	1	0	OBA	CB	1	0	W1A	D3	1	0
INS	77	1	0	LDT	75	1	0	OBT	DB	1	0	W1B	D4	1	0
IQ1	88	1	0	LDZ	FB	1	0	PDO	10	1	0	W1E	D1	2	0
IQ2	99	1	0	LKA	A3	1	0	PDS	20	1	0	W1F	D2	1	0
IQC	33	1	0	LKB	A4	1	0	POE	01	1	0	W1M	D0	1	0
IQM	55	1	0	LKD	A7	1	0	PSD	02	1	0	W1T	D5	1	0
IQR	22	1	0	LKE	A1	2	0	RC0	1C	1	0	W2A	E3	1	0
IQU	44	1	0	LKF	A2	1	0	RC1	1D	1	0	W2B	E4	1	0
JHB	94	2	0	LKM	A0	1	0	RC2	1E	1	0	W2E	E1	2	0
JLC	91	3	0	LKR	A6	1	0	RC3	1F	1	0	W2F	E2	1	0
JLN	92	3	0	LKT	A5	1	0	RE0	2C	1	0	W2M	E0	1	0
JLR	93	3	0	LRA	63	1	0	RE1	2D	1	0	W2T	E5	1	0
JPL	90	3	0	LRB	64	1	0	RE2	2E	1	0	W3A	F3	1	0
JPS	80	2	0	LRE	61	2	0	RE3	2F	1	0	W3B	F4	1	0
JSB	84	2	0	LRF	62	1	0	RI0	0C	1	0	W3E	F1	2	0
JSC	81	2	0	LRI	69	1	0	RI1	0D	1	0	W3F	F2	1	0
JSN	82	2	0	LRK	6A	1	0	RI2	0E	1	0	W3I	F0	1	0
JSR	83	2	0	LRM	60	1	0	RI3	0F	1	0	W3T	F5	1	0
JST	85	2	0	LRP	68	1	0	RRA	C7	1	0	XAB	DC	1	0
LA0	3C	1	0	LRS	6B	1	0	RRB	D7	1	0	XAT	CC	1	0
LA1	3D	1	0	LRT	65	1	0	RRD	F7	1	0	XBA	CD	1	0
LA2	3E	1	0	LSA	B3	1	0	RRT	E7	1	0	XBT	DD	1	0
LA3	3F	1	0	LSB	B4	1	0	RST	AA	1	0	XDS	21	1	0

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com

EUP 5108 SOAP Commands & Syntax

CMD	Parameter	Description	APA = ADA = 0
#	*anything*	Ignores everything after '#' up to the EOL	n/a
I	Filename	Includes the contents of Filename at this point then continues processing at the next line	n/a
O	0xHHHH	Set the value of the APA (Assembler's Program Address)	APA=0xHHHH
D	Const_Name	Defines value of 'Const_Name' to the current APA value	Const_Name.Value = APA
L	XX XX...	Writes hex values 'XX XX...' to ROM at current APA	ROM[APA++] = {XX, XX...}
S	Char string	Writes Char string to ROM w/o trailing space(s), not including terminator. Accepts '\' escape codes: b=8,f=0xFF,n=10,r=13,t=9,0=0,e=27	ROM[APA++] = "Char String"
W	Const_Name	Writes the 16bit value (little endian) of Const_Name value to ROM at current APA	ROM[APA++] = Const_Name.Value
C	Const_Name	Defines Const_Name to the hex value parameter	Const_Name.Value = Hex Parameter
[spc]	Instruction TLA	Writes the Instruction's Opcode & Parameter or Const_Name value at the current APA value	ROM[APA++] = INS.OpCode ROM[APA++] = Parameter || Const_Name.Value
A	0xHHHH	Set the ADA (Assembler's Data Address) to 0xHHHH	ADA=0xHHHH
V	Var_Name	Sets 'Var_Name'.Value to the current ADA value	Var_Name.Value=ADA
a	0xHH	Increase the ADA by 0xHH to reserve space in RAM for the variable e.g. char(1 byte), short(2 bytes), char[10] (10 bytes)	ADA+=0xHH

Using the EUP5108 Assembler (EUPASM.exe):
In a Windows command prompt shell simply pass your top level **.asm** file into EUPASM.exe



The primary output files are named after the top level .asm file and include:

A **.sym** file that contains the symbol table for all *Global* symbols (by convention a symbol that contains a period ('.') is considered local and meant to be only referenced within the function e.g. loop lables).

A **.rom** file that contains the program ROM in the "v3.0 hex words addressed" format that is easily loadable into Logisim.
A **.bin** file containing the binary version of the program ROM for use in the upcoming processor emulator program.

For every **.asm** file processed a **.map** file containing the mapping of each line of source to locations in ROM and/or RAM.

The assembler is not very robust and has a number of built in limitations:
File paths, symbol names, and line lengths within the files are limited to 255 characters.
The symbol table is limited to 1024 symbols (both global and local).
There is no protection from including a file multiple times although redefined symbols will cause an assembly failure.

© 2021 Emil DeVries, All Rights Reserved. For licensing support our youtube channel via Pateron – www.EunumPluribus.com