EMU 1.0

Electro-Mechanical Unit for industrial applications

Engineer's Manual

Revision *A*

HTSP
Department of Computing
and Analytical Engines

EMU 1.0 Engineer's manual

Revision *A*

Tahlia

Table of Contents

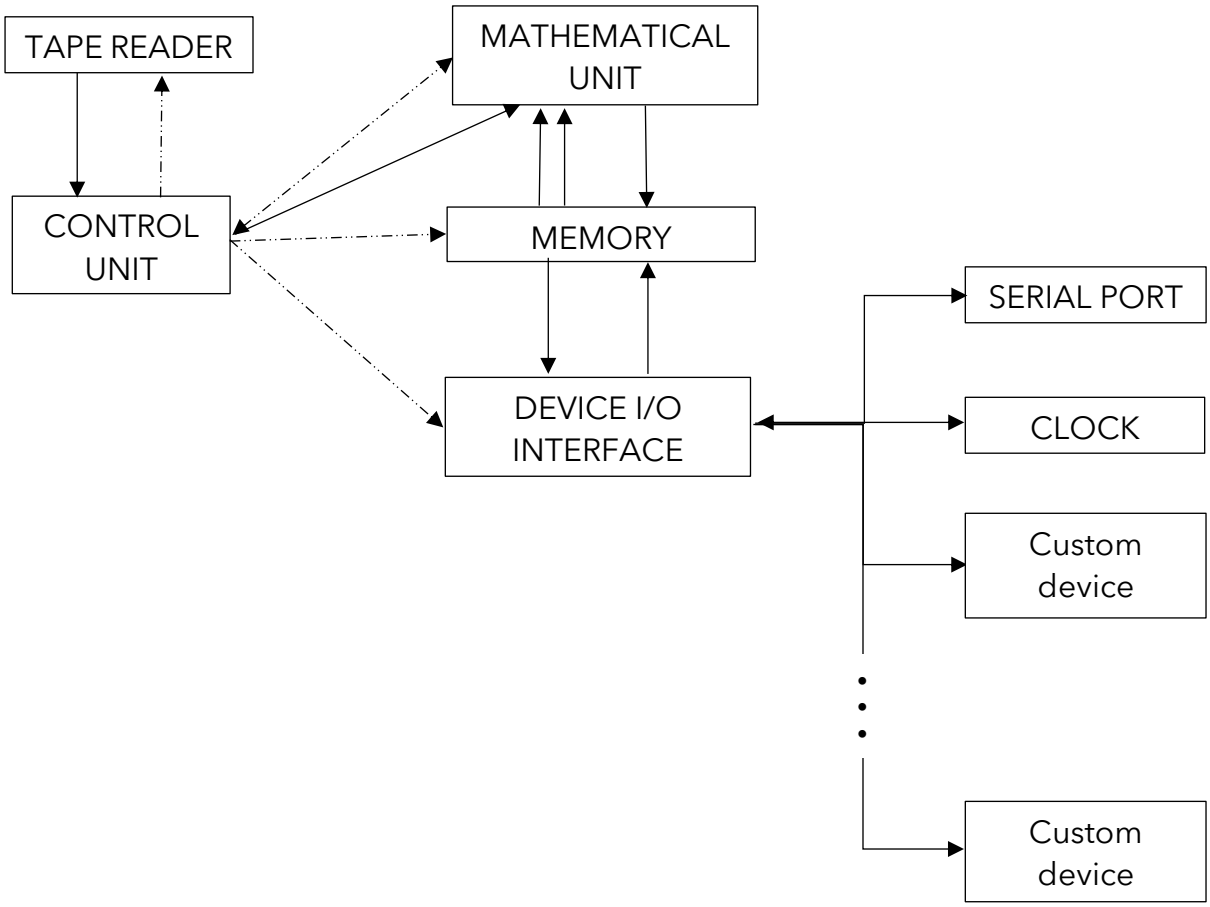
Overview of computer capabilities

The EMU 1.0 is a Harvard architecture 6-bit relay computer. It has 63 6-bit words of working memory, addressable either directly, as general-purpose registers, or indirectly, as Random Access Memory. Its program memory is stored on 24 row punched tape and is unlimited in size. EMU 1.0 can perform a variety of arithmetic and bitwise operations, including fixed point multiplication, and it has an extensible device I/O subsystem.

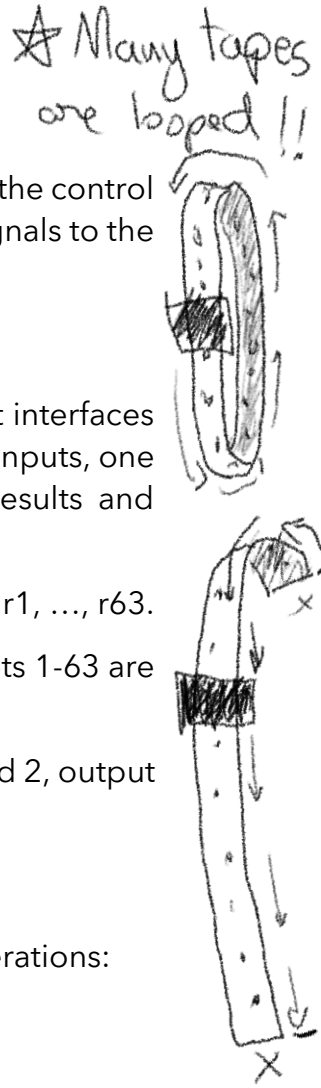
- Harvard architecture
- 6-bit arithmetic, logic, fixed point multiplication
- 63 words of working memory
- Unlimited program memory; punched tape, 24 rows
- Up to 60 custom I/O devices

Functional diagram

Dashed lines represent control signals. Solid lines represent data paths.



Component description, interface



Tape reader

The tape reader outputs the instruction currently under the read head to the control unit. The control unit can send “move forward” and “move backward” signals to the tape reader, which will actuate the tape accordingly by one step.

Memory

The memory bank has 64 addressable slots, selected via control lines. It interfaces with the other components via 3 data output latches: 2 for the math unit inputs, one for outputting data to devices, and 2 data inputs for storing math results and receiving data from devices.

In EMU 1.0 assembly, memory slots are referred to using the r prefix: r0, r1, …, r63.

Slot 0 will always equate to the value 0 and writing to it has no effect. Slots 1-63 are general purpose and can be used in any instruction.

Control signals: address select (6 bits), output operand 1, output operand 2, output io, input math, input io.

Mathematical unit

The mathematical unit encompasses multiple circuits for the various operations:

- Arithmetic and logic circuit (add, sub, cmp, xor, or, and)
- Shift, rotate circuit (shi, roi, shl, shr, mul)
- Multiplier circuit (mul)

The left operand is always received from the memory component, but the right operand may be either data from the memory or an immediate value received from the instruction decoder. The shifter may either operate on the value of the first operand, or on the output of the multiplier circuit. The ALU always outputs condition flags to the control unit, which are used to set the condition flag during the comparison instructions.

Control signals: immediate b, output select (0 = ALU, 1 = shifter/multiplier); (To ALU) subtract, operation select (0 = add/sub/cmp, 1 = xor, 2 = or, 3 = and); (To shifter) rotate, direction, multiply; (To multiplier) signed.

Output signals: Overflow, Zero, Carry, Sign.

Device I/O interface

On each I/O operation a word is sent to the selected device and a word is received and stored to memory.

Control signals: device select (6 bits), activate.

Serial port device

The serial port facilitates communication with other computers and exposes 3 device addresses:

Device 0: SERIAL_INCOMING – Reads the number of buffered incoming words, clamped to max 63.

Device 1: SERIAL_READ – Reads the next buffered word, -1 if there are none.

Device 2: SERIAL_WRITE – Writes send a word to the remote machine.

Serial port character set

Communications over the serial port conventionally use the following charset.

	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
00+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20+	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
40+	W	X	Y	Z	**SP**	+	-	*	/	<	=	>	(	)	[	]
60+	{	}	#	$	_	?	|	^	&	!	~	,	.	:	**LF**	

The last one is an invalid character, SP is the space character, LF is a line feed.

Clock device

Timing is a strict necessity in control applications, as well as user interfaces. The clock device exposes two device addresses with the following functionality:

Device 3: CLOCK_LO_CS – Reads least significant 6 bits of centisecond counter.

Device 4: CLOCK_HI_CS – Reads most significant 6 bits of centisecond counter.

Writing a nonzero value to either device resets the counter to 0. The 12-bit counter does not overflow and wrap around, instead it is clamped at a max value of 4095 = 40.95 seconds.

Upon computer startup, the value of the counter is automatically set to 0.

?A : seems like octal ?!

Digitisation project?
24 bit inst → 4 base 64 symbols?

Instruction encoding

The 24 instruction bits are split into 4 6-bit parts, from now on named **OPC, A, B, C**.

OPC = 0 is an invalid instruction and in turn halts the computer. The other 63 OPC values encode one of the 21 instruction codes (denoted **op**) and a ternary condition code: **OPC = 21 * condition + op + 1**. The condition code influences execution as follows:

condition	Assembly text representation	Description
0	(none)	Execute unconditionally
1	+	Execute if the condition flag is true
2	-	Execute if the condition flag is false

For instance, the following EMU 1.0 assembly and C pseudocode are equivalent:

```
  cmpeq r2, r3
+ add   r4, r5, r6
  add   r5, r5, 011
- sub   r4, r5, r6
```

```
flag = r2 == r3;
if(flag) r4 = r5 + r6;
r5 = r5 + 9;
if(!flag) r4 = r5 - r6;
```

op = 20 is reserved for future extensions. Executing it should halt the computer. op 0 through 19 are defined in the following pages.

Arithmetic and logic

op A B C	Assembly	Description
000 rd ra rb	add rd, ra, rb	Add **ra** and **rb** into **rd**.
001 rd ra ib	add rd, ra, ib	Add **ra** and immediate **ib** into **rd**.
002 rd ra rb	sub rd, ra, rb	Subtract **rb** from **ra** and store into **rd**.
004 rd ra rb	or rd, ra, rb	Bitwise or **ra** and **rb** into **rd**.
005 rd ra ib	or rd, ra, ib	Bitwise or **ra** and immediate **ib** into **rd**.
006 rd ra rb	xor rd, ra, rb	Bitwise xor **ra** and **rb** into **rd**.
007 rd ra rb	xor rd, ra, ib	Bitwise xor **ra** and immediate **ib** into **rd**.
010 rd ra rb	and rd, ra, rb	Bitwise and **ra** and **rb** into **rd**.
011 rd ra rb	and rd, ra, ib	Bitwise and **ra** and immediate **ib** into **rd**.
013 rd ra rb	shl rd, ra, rb	Logical* shift **ra** left **rb** bits into **rd**.
014 rd ra rb	shr rd, ra, rb	Logical* shift **ra** right **rb** bits into **rd**.

Octal again

* Logical shifts fill the undefined bits with zeros. There is no arithmetic right shift.

Arithmetic and logic instructions only change **rd**, they do not alter any other part of the computer state.

★ Undocumented ?!?
003 010+cc rb ra

Comparison

op A B C	Assembly	Description
003 000+cc ra rb	cmpCM ra, rb	Compare **ra** and **rb**.
003 020+cc ra ib	cmpCM ra, ib	Compare **ra** and immediate **ib**.
003 030+cc ia rb	cmpCM ia, rb	Compare immediate **ia** and **rb**.

Comparison instructions do not change the contents of the memory, but they set the Control Unit's conditional flag according to their result.

The comparison mnemonics **CM** and their numerical values **cc** are as follows:

cc	CM	Description	Logical expression
0	tr	True	1
1	fa	False	0
2	eq	Equal	ZF = 1
3	ne	Not equal	ZF = 0
4	sl	Signed less than	SF xor OF = 1
5	sg	Signed greater than	SF xor OF = 0 and ZF = 0
6	ul	Unsigned less than	CF = 1
7	ug	Unsigned greater than	CF = 0 and ZF = 0

} why?! crikey

"Less/greater than or equal" comparisons can be implemented by inverting the condition and swapping the comparison inputs. (e.g. "a <= b" equivalent to "b > a")

Shift/rotate by immediate

op A B C	Assembly	Description
`012 rd ra 000+ib`	`shl rd, ra, ib`	Logical shift **ra** left immediate **ib** bits into **rd**.
`012 rd ra 010+ib`	`shr rd, ra, ib`	Logical shift **ra** right immediate **ib** bits into **rd**.
`012 rd ra 020+ib`	`sar rd, ra, ib`	Arithmetic shift **ra** right immediate **ib** bits into **rd**.
`012 rd ra 030+ib`	`rol rd, ra, ib`	Rotate **ra** left immediate **ib** bits into **rd**.

Limits: 0 <= **ib** < 8.

Right rotations can be achieved by their complementary left rotation. Just like the other arithmetic instructions, these only change the destination memory slot.

Indirect memory access

C: V[i]
EMU: [V+i]

op A B C	Assembly	Description
`015 rd ra ib`	`ld rd, [ra+ib]`	Load value at address (slot **ra** + immediate **ib**) mod 64 into **rd**.
`016 rs ra ib`	`st [ra+ib], rs`	Store **rs** at address (slot **ra** + immediate **ib**) mod 64.

EMU1.0 assemblers should interpret `[ra]` as `[ra+0]` and `[ib]` as `[r0+ib]`.

Fixed-point multiply

op A B C	Assembly	Description
`023 rd ra 000+pr`	`fmu/pr rd, ra`	Multiply unsigned fixed-point **rd** and **ra** into **rd**, with **pr** fractional bits.
`023 rd ra 020+pr`	`fms/pr rd, ra`	Multiply signed fixed-point **rd** and **ra** into **rd**, with **pr** fractional bits.

Limits: 0 <= **pr** < 16.

Multiplies **rd** and **ra** to an internal 12-bit value, shifts right (logical for unsigned, arithmetic for signed) by **pr** bits, storing the final 6 least significant bits to **rd**.

Using **pr** = 6, one can also obtain the upper half of a multiplication result, disregarding its interpretation as a fixed-point value.

Fun: implement some 3D wireframe rendering

Control flow

op A B C	Assembly	Description
017 la lb lc	lbl lab, lc	Declare jump label lab, lc.
020 la lb rc	jup lab, rc	Jump upwards to label lab, rc.
021 la lb rc	jdn lab, rc	Jump downwards to label lab, rc.

Limits: 0 <= lab = 64*la + lb < 4096

EMU1.0 assemblers should allow the lc or rc values to be omitted, case in which they default to 0 and r0, respectively.

EMU 1.0 uses a system of designated jump targets to be able to branch without a program counter. Each jump target has an 18-bit identifier given by A, B, C, grouped into lab and lc. When jumping, the Control Unit scrolls the tape in the hinted direction until it hits a corresponding lbl instruction. That is, one which matches both the jump lab, as well as lc with the value of memory slot rc.

For normal jumps, lc and rc should be zero. Subroutine returns may be implemented by jumping to a return label and a designated "return pointer"-esque memory slot.

Label instructions may be conditional, case in which they will be ignored by the Control Unit when searching for a jump target if the condition dictates so.

Subroutine example

A subroutine at label 01000 is called from multiple places. It returns to the labels 01001. To differentiate them, each call site uses a different lc, which is passed to the subroutine via the r63 slot.

```
# First call site
add r63, r0, 1 # Set "return identifier" to 1
jdn 01000      # Jump to subroutine
lbl 01001, 1   # Return location with identifier 1


# Second call site
add r63, r0, 2 # Set "return identifier" to 2
jdn 01000      # Jump to subroutine
lbl 01001, 2   # Return location with identifier 2

# ...


lbl 01000      # Subroutine entry label
# ...          # Omitted subroutine body; does not trash r63
jup 01001, r63 # Jump to whichever return label matches r63
```

Device I/O

op A B C	Assembly	Description
022 rd ix rs	io rd, ix, rs	Send **rs** to device **ix** and receive **rd**.

EMU 1.0 assemblers should interpret allow either **rd** or **rs** to be omitted, interpreting partial **io** instructions as follows:

```
io rd, ix      =    io rd, ix, r0
io ix, rs      =    io r0, ix, rs
io ix          =    io r0, rx, r0
```

The Device I/O interface enables word-by-word communication with the 64 devices, selected by immediate **ix**. Each **io** invocation sends the value of **rs**, activates the device, gets a response value, and stores it to **rd**. The default **Serial Port** and **Clock** devices and their protocols are described in their respective sections. Other special-purpose devices may be installed on the 59 remaining device slots.

Accessing a device that does not exist/is not connected halts the computer.

Digitise TODO:

- ✓ Mandel Flag
- Robot arm controller
- ELIGMA decypher
- Business manager database
- Blotto Box impl.