JAMES BATES 8BIT COMPUTER SIMULATION

Document Version V1_00 13-Sept-2023 Author 6V6GT

1 INTRODUCTION

This is a software simulation of the 8bit computer created by James Bates [Ref 1] for which he used basic logic integrated circuits and discrete components with an impressive build covering multiple component boards. The design of this computer is described in later chapters but basically it has a Harvard architecture with two banks of 256 bytes of memory, 4 general purpose

Figure 1-1 Original James Bates 8bit computer

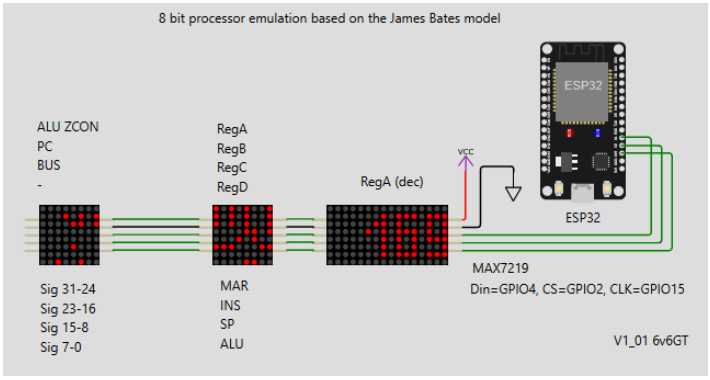


Figure 1-2 ESP32 based C++ simulation

registers, a stack and a shared 8bit bus and is capable of running small programs.

The components of each module, that is the clock, controller, ALU etc. (full list in Ch 3) are grouped together on dedicated breadboard sets. The simulation, which is the subject of this document, is a near 1 to 1 model of the original with each of the modules and shared bus being represented by its own C++ object hosted on an ESP32 microcontroller and built using the Arduino development system.

The simulation provides a similar learning experience to the original in that the user can load small programs and trace their execution watching how each assembly level instruction is executed as a number of discrete micro instructions, transferring instructions and data between registers and the bus and thus help to develop an understanding of the basic architecture of simple computers. Of course the simulation lacks the striking visual appearance of the original but otherwise, at the software level, offers something functionally equivalent without the need to build a comprehensive hardware device.

No changes have been made to the architecture, the instruction set or the microcode so programs which were developed for the original 8bit computer should run on this simulation without modification and all documentation should still be relevant. The simulation has been designed in such a way that it could be the basis of a hybrid computer; that is a part physical build in the style of the original with the remaining parts being simulated.

2 CONTENTS

2.1 ABBREVIATIONS / GLOSSARY

1 to 1	A close representation of the original
ALU	Arithmetic and Logic Unit
Breadboard	A multi-connection component board for developing small circuits
EEPROM	a non-volatile storage IC
ESP32	A 32 bit microcontroller with 4MB flash and 512KB RAM (model dependent)
Host	A computer providing resources for a simulation or virtual computer
IC	Integrated Circuit
IDE	Interactive Development Environment
JB	James Bates, the author of the original architecture
LED	Light Emitting Diode
MCU	Microcontroller Unit
Microcode	The individual instructions which implement the instruction set
Opcode	The code which represents a single assembly level instruction
PCB	Printed Circuit Board
RAM	Random access memory (volatile)
SMD	Surface Mount Device

3 JB 8BIT COMPUTER

This is a teaching machine but has a powerful enough specification to run some quite complex demonstration programs including some hand compiled from C programs. It is supported by full documentation [Ref 2] and some support tools. There is an assembler for generating machine code and a tool for generating and loading the microcode onto EEPROMs. There are also a number of programs written in assembly language to try out. Each module of the James Bates computer is described in his online Video series and a schematic diagram included in his Github repository [Ref 2]. The specification of the machine together with the quality of the documentation, explanatory videos, and the complete support tools set was the main reason for selecting the JB 8bit computer for the basis of this simulation project.

3.1 HISTORICAL DEVELOPMENT

James Bates based his 8bit computer is based on the work of Ben Eater [Ref 3] who produced a simpler specification computer but again backed up with a comprehensive video series describing the project and he even sells kits for people who may wish to duplicate his design. Ben Eater in turn based his design on the work of Albert Paul Malvino who described a series of basic computer architectures is his book Digital Computer Electronics [Ref 4] including one he named "SAP-1" for Simple as Possible but also including a design which is upwards compatible with the Intel 8085 microprocessor.

3.2 ARCHITECTURE

This computer has a Harvard architecture with two banks of 256 bytes RAM one bank for instructions and the other for data, 4 general purpose registers, a stack and a shared 8bit bus for instructions and data.

The following is a brief description of each module and further details can be found in [Ref 2]. Refer to the diagram below to see the relationship between the modules.

3.2.1 BUS

This is an 8bit shared data and program bus. Only one module may write to the bus at any one time and this is ensured by the controller and generally only one module will read the bus contents at any one time.

3.2.2 CLOCK

The clock synchronizes the transfer of data between components. Most activities/transfers take place on the rising clock edge. There are exceptions such at the clearing of the micro timer in the controller which happens on the falling clock edge. This has an adjustable speed to make it easy to observe the execution of code.

3.2.3 RAM

The memory is divided into two 256 byte banks one of which is for the program and the other for the data which is similar to the Harvard architecture which is often used for microprocessors where these run programs from read only memory. To access the RAM, an address is put into the memory address register which is connected directly to the RAM. A controller signal (PGM) is asserted if the program bank is to be addressed. The program RAM can be loaded from the console through switches.

3.2.4 CONTROLLER

This drives all the modules. It fetches the next instruction to executed from RAM. This is pointed at by the program counter and loaded into the instruction register. It executes the instruction by first examining its opcode (and maybe ALU flags) then fetching the microcode from the EEPROMs and incrementing the micro timer. All instructions are implemented by multiple sets of microcode and the next set to be executed is pointed at by the micro timer. The micro code here is simply a set of controller signals. At the end of the execution of an instruction the micro timer is reset to zero.

3.2.5 PROGRAM COUNTER

This is simply a register used to hold the address of the next instruction to be executed. It has an single increment function.

3.2.6 STACK POINTER

This is a register which points at the top of the stack. The stack grows downwards from the top of the data memory bank.

3.2.7 ALU – ARITHMETIC AND LOGIC UNIT

This is based on the functionality of the SN74LS382 ALU chip. In principle it is a full 4bit adder (two chips are used) and performs also subtraction and some logical operations. It can take optionally its second operand from the B register or use a zero instead. The SN74LS382 is obsolete but some stocks exist for those wishing to build this unit [Ref 5]

3.2.8 REGISTERS A, B, C AND D

These are general purpose 8 bit registers. Register A is displayed in decimal on the console. Register B is directly connected to the ALU.

3.2.9 CONSOLE

This can be viewed as the user interface to the computer. This allows the entry of programs, stepping through them using the clock control and displaying intermediate values and registers, signals etc.

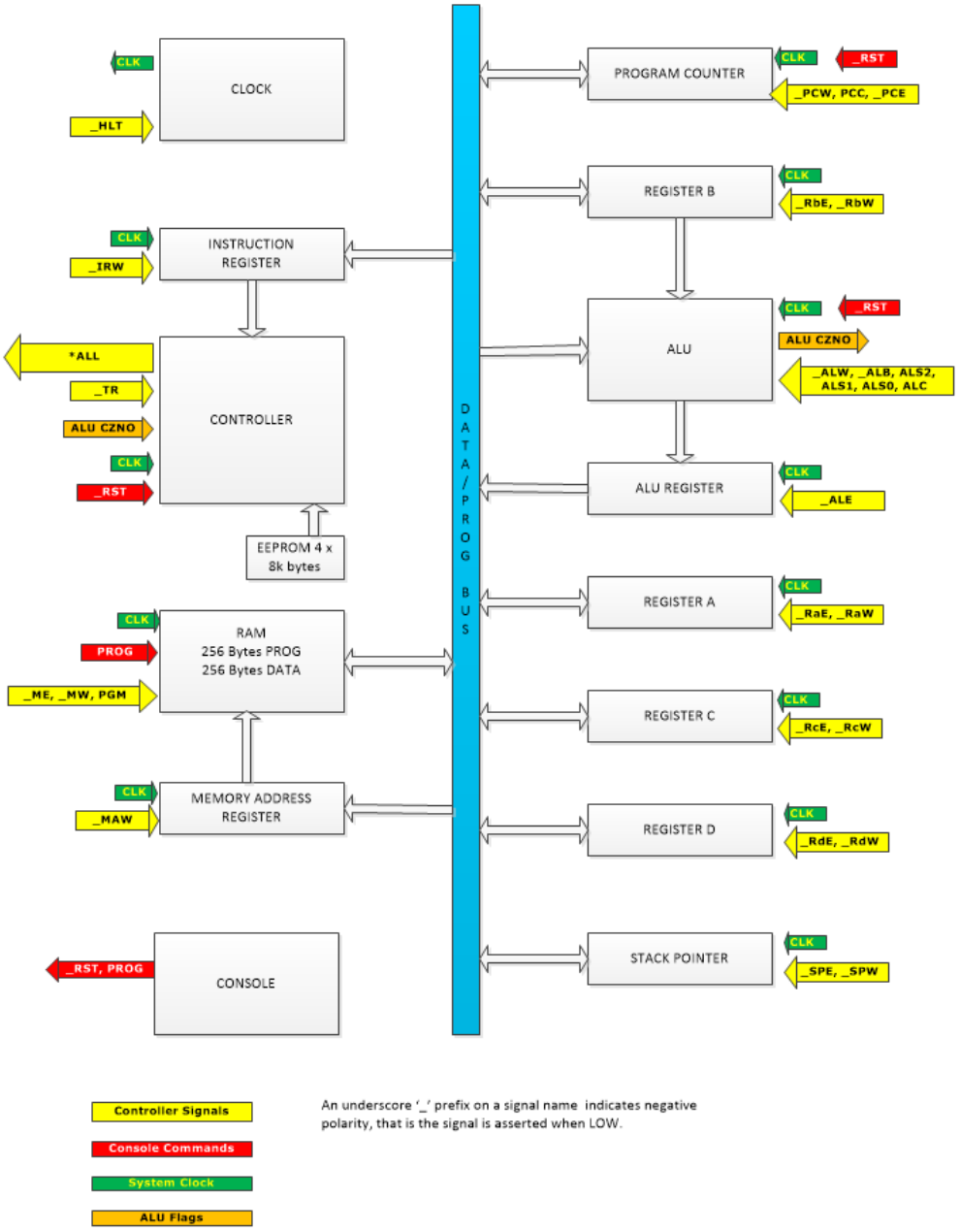


Figure 3-1 Architecture JB 8bit computer

4 SIMULATION

The simulation uses a C++ class model to represent each module in the physical 8bit computer design. Within a module the basic functionality is preserved but no attempt has been made to have a direct representation of the individual module sub-components say at the chip level. The purpose of this project is to create a training/educational experience without the necessity to immediately assemble a large hardware project. For those who wish to experience the authentic effect of blinking

lights to show the status of registers, signals etc., which is one of the impressive visual features of the original hardware design, an optional model has been added to support this. However, important is also that there are multiple trace levels available so the flow through a running program can be monitored, for example, at the assembly instruction level or the microcode level with explanatory text status messages being written to the serial console. The target hardware on which the simulator runs, an ESP32, can itself be simulated using an online simulator so it is not even necessary to possess any hardware to play with it.

4.1 BUILD

The hardware for the simulation is extremely simple. It consists of an ESP32 device and optionally four 8x8 LED MAX7219 matrixes (see schematic in Ch. 1). Because of the simplicity the physical hardware hosting the simulation is easily simulated using an online simulator and one has already been prepared [Ref 6]. The C++ code for the simulation has been prepared for the Arduino development system. The actual development was done using Sloeber 4.4.1 which is more suitable for large projects but the results are completely compatible with the Arduino IDE.

There is a close match between the architecture of the original computer and the simulation described here. In principle, most modules have a dedicated class with object names corresponding to that of the original modules. The registers objects, most of which are identical are, however, derived from a single class. The distribution of the microcode between the four 8KB EEPROMS of the original computer has been kept and modeled as four arrays in the simulation. The emphasis of the design has been to provide a simple translation from the physical to the simulated object and not necessarily to produce a showcase C++ solution. The ESP32 has been chosen because of its large RAM/Flash store. In this application its wireless features, however, are not used. A Uno class MCU would not even be able to hold the simulated EEPROMS.

There are, however, also a number of differences to the original hardware model. For example, the clock is explicitly 4 phase to model the hardware clock which has high and low states, but also transient states rising edge and falling edge which are critical for the transfer of data between modules via the bus. Further, the console of the original computer is rudimentary for example loading a program is via a set of switches to set the address and the data/operations. In the simulation, the console is managed over the Arduino serial console using text command and output providing a significantly more convenient user interface.

4.2 USER INTERFACE FEATURES

The user interface features available in the original computer are duplicated and supplemented. The optional led display gives an immediate snapshot of the state of all registers, signals, ALU flags and the bus including a decimal digits indication of the contents of the display register (Reg_A). Admittedly, it does not have the same appealing visual effect as the original but is nevertheless clear and serviceable. The clock has a single step through mode and a currently 3 predefined automatic modes to step through the running program at different speeds. Multiple trace levels are

available. Suppressing tracing can be useful when, for example viewing the contents of one of the memory banks without the console scrolling. Just showing the value of Reg_A (display register) is the minimum to show the system is working, that is in the absence of the optional LED matrix displays. Tracing the running program at the assembler instruction level is probably the easiest for debugging. There are two more levels, one for showing all the microcode or micro instructions generated by each assembly level instruction and a similar one which omits the instruction fetch cycle with is always identical for each instruction. All console commands are listed in the chapter "Usage Instructions".

4.3 Basic Usage Instructions

The location of the software source packet is in the appendix. Instructions for modifying the code, mainly selecting the display options or default startup behaviour are in the code itself (.ino file). Basic instructions, in case necessary, for installing the Arduino IDE, compiling a program and loading such a program onto the target ESP32 can be found in [Ref 7] and [Ref 8]. The serial console must be configured for (a) 115200 baud and (b) with a CRLF termination sequence (not always standard!).

The simulation is configured by default to initiate a program and step through it automatically but with tracing suppressed. This start up behaviour can be changed in the Console class.

The console commands available are listed below. These are case sensitive.

Table 4-1 Clock Commands

Command	Description
clocks	clock slow. Note that the effective clock speed is dependent on the trace level
clockm	clock medium
clockf	clock fast
cmm	clock mode manual. Hit enter to advance to next clock phase.
cma	clock mode automatic. The clock advances based on the chosen speed
<enter>	valid only in manual mode to advance the clock

Table 4-2 Trace Level Commands

Command	Description
tl0	trace level 0. No trace output. Useful to prevent the output generated by other console commands scrolling away
tl1	trace level 1. Only the contents of the display register (Reg_A) are displayed
tl2	trace level 2. Shows the status at the assembly code level
tl3	trace level 3. Shows each microcode instruction but omits the instruction fetch sequence because this is identical for every instruction.
tl4	trace level 4. Everything. This is recommended for manual clock mode

Table 4-3 Programming Commands

Command	Description
progs	Enter programming mode.
progt	Terminate programming mode.
<dec> <dec>	Two space separated numbers in programming mode indicate the address and the instruction or data to be entered at that address
progv	View the contents of the program memory. All 256 bytes are displayed

Table 4-4 Preload sample program

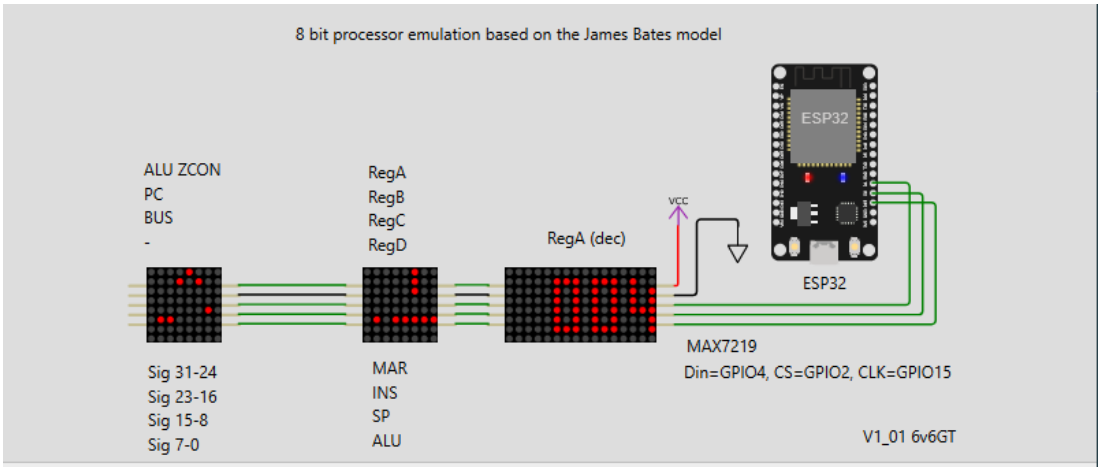
Command	Description
progFib	Fibonacci series program 1, 2, 3, 5, 8, 13 etc.
progPower	Power series, 2^1, 2^2, 2^3 . . 2^7, 3^1, 3^2 . . 3^5, 4^1 . . where the result is less than 256
progPower	Runs the above two programs plus a simple multiplication table in sequence

Table 4-5 Miscellaneous Commands

Command	Description
datav	View entire data memory. All 256 bytes.
rst	Reset. Restarts the running program
?	Displays a list of all commands. It is usually best to issue the command tl0 beforehand to suppress any program tracing.

4.3.1 SAMPLE SESSION

This is the simulation itself running in a simulator (Wokwi) [Ref 6]



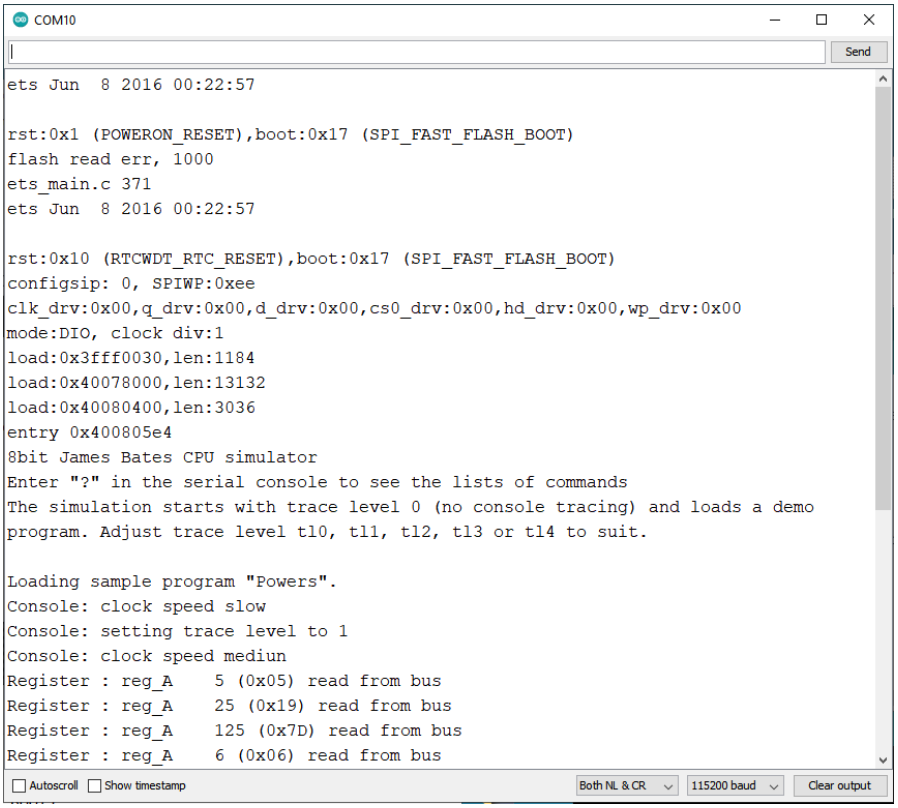

```
ets Jul 29 2019 12:21:46

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1156
load:0x40078000,len:11456
ho 0 tail 12 room 4
load:0x40080400,len:2972
entry 0x400805dc

8bit James Bates CPU simulator
Enter "?" in the serial console to see the lists of commands
The simulation starts with trace level 0 (no console tracing) and loads a demo
program. Adjust trace level tl0, tl1, tl2, tl3 or tl4 to suit.

Loading sample program "Powers".
```

Or with a real (non-emulated) ESP32



4.4 Sample Program Entry and Tracing

Here is a trivial program which simply loops incrementing a counter from 0 to 255. For simplicity, it is done in the display register Reg_A. We simply set the contents of Reg_A to zero, increment the contents and loop. It is hand assembled as follows using the instruction list in the appendix. The blue shaded items Address and Opcode are entered in decimal into the program RAM.

Address	Data/Opcode	Instruction	Description
0	0x07 (dec 7)	DATA Ra, #IMM	Put the following data item in Reg_A
1	0	-	zero
2	0xC0 (dec 192)	Inc Ra	Increment the contents of Reg_A
3	0x2F (dec 47)	JMP #imm	Jump to the address following
4	2	-	Address 2

To load the program into program RAM enter the following in the serial console:

entry	description
tl0	Trace level 0 to suppress tracing
cmm	Stop the clock by forcing it into manual mode
progs	Start programming mode
0 7	Address Data/Opcode pair
1 0	Address Data/Opcode pair
2 192	Address Data/Opcode pair
3 47	Address Data/Opcode pair
4 2	Address Data/Opcode pair
progt	Terminate programming mode
rst	reset
tl1	Trace level 1 (just show changes to Reg_A display register)
clocks	Slow clock speed
cma	Automatic clock advance

This gives the following output on the serial console. After the clock mode is set to cma (automatic) the program runs through its counting sequence.

```
Console: setting trace level to 0
Console: clock mode set to manual
Console: starting programming mode
Console: added address= 0 ; opcode = 7 (0x07) to program RAM
Console: added address= 1 ; opcode = 0 (0x00) to program RAM
Console: added address= 2 ; opcode = 192 (0xC0) to program RAM
Console: added address= 3 ; opcode = 47 (0x2F) to program RAM
Console: added address= 4 ; opcode = 2 (0x02) to program RAM
Console: terminating programming mode
Console: resetting CPU
Console: setting trace level to 1
Console: clock speed slow
Console: clock mode set to automatic
Register : reg_A    0 (0x00) read from bus
Register : reg_A    1 (0x01) read from bus
Register : reg_A    2 (0x02) read from bus
Register : reg_A    3 (0x03) read from bus
...
```

Increasing the trace level to tl2 gives this sample output. It is not a complete program cycle and is intended for illustration only.

```
ProgramCounter :  0 (0x00)
ProgramCounter : Writing to bus 0 (0x00)
Register : reg_MA   0 (0x00) read from bus
Ram : prog[ 0 ]  (0x07) written to bus
ProgramCounter :  0 (0x00)
Register : reg_IR   7 (0x07) read from bus
ProgramCounter : Incremented to 1 (0x01)
Controller: opcode= [DATA Ra, #IMM]
ProgramCounter :  1 (0x01)
ProgramCounter : Writing to bus 1 (0x01)
Register : reg_MA   1 (0x01) read from bus
ProgramCounter : Incremented to 2 (0x02)
Controller: opcode= [DATA Ra, #IMM]
```

```
Ram : prog[ 1 ]  (0x00) written to bus
ProgramCounter :  2 (0x02)
Register : reg_A    0 (0x00) read from bus
ProgramCounter :  2 (0x02)
ProgramCounter : Writing to bus 2 (0x02)
Register : reg_MA   2 (0x02) read from bus
Ram : prog[ 2 ]  (0xC0) written to bus
ProgramCounter :  2 (0x02)
Register : reg_IR   192 (0xC0) read from bus
ProgramCounter : Incremented to 3 (0x03)
Controller: opcode= [INC Ra]
ProgramCounter :  3 (0x03)
Register : reg_A    0 (0x00) written to bus
Alu : value written to registerAL=1 ; C=0 ; O=0 ; Z=0 ; N=0
Controller: opcode= [INC Ra]
ProgramCounter :  3 (0x03)
Register : reg_AL   1 (0x01) written to bus
Register : reg_A    1 (0x01) read from bus
ProgramCounter :  3 (0x03)
ProgramCounter : Writing to bus 3 (0x03)
Register : reg_MA   3 (0x03) read from bus
Ram : prog[ 3 ]  (0x2F) written to bus
ProgramCounter :  3 (0x03)
Register : reg_IR   47 (0x2F) read from bus
ProgramCounter : Incremented to 4 (0x04)
Controller: opcode= [JMP #imm]
ProgramCounter :  4 (0x04)
ProgramCounter : Writing to bus 4 (0x04)
Register : reg_MA   4 (0x04) read from bus
ProgramCounter : Incremented to 5 (0x05)
. . .
```

And increasing the tracing level further to *tl4* gives the following sample output where the micro instruction level can also be seen. Again, this is only a fragment of a complete program cycle:

```
Console: setting trace level to 4
Console: clock mode set to automatic
CpuClock: falling edge
Controller: microtimer incremented to 2
ProgramCounter : Incremented to 4 (0x04)
CpuClock: low
CpuClock: rising edge
Controller: eeprom address = 0x022F
Controller: 0x839FF7F0  0000'0000'0010'0000'0000'1000'1000'0000' 0x2F 2 0 0 [JMP #imm]
Controller: asserted signals- PCC, _PCE, _MAW,
ProgramCounter :  4 (0x04)
ProgramCounter : Writing to bus 4 (0x04)
CpuClock: high
Register : reg_MA   4 (0x04) read from bus
CpuClock: falling edge
Controller: microtimer incremented to 3
ProgramCounter : Incremented to 5 (0x05)
CpuClock: low
```

```
CpuClock: rising edge
Controller: eeprom address = 0x032F
Controller: 0x01FFFB70  1000'0010'0100'0000'0000'0100'0000'0000' 0x2F 3 0 0 [JMP #imm]
Controller: asserted signals- _PCW, PGM, _ME, _TR,
Ram : prog[ 4 ]  (0x02) written to bus
ProgramCounter :  5 (0x05)
CpuClock: high
ProgramCounter : Fetching from bus 2 (0x02)
CpuClock: falling edge
Controller: microtimer set to 0
CpuClock: low
CpuClock: rising edge
. . .
```

The contents of the program memory can be shown with the *progv* console command as follows:

```
Console: dumping program RAM . . .

  0 : 0x07 0x00 0xC0 0x2F 0x02 0x00 0x00 0x00 0x00 0x00
  1 : 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
  ...
```

4.5 More Advanced Programming

To progress beyond simply using the ready prepared programs or hand assembling simple programs, there is the possibility of building an assembler from the software kit and creating new assembly level programs or experimenting with the available examples. This is described in [Ref 2] together with a link to a dedicated video.

As an example, for this project, the assembler was built using a virtual Linux instance (ubuntu-20.04.3-desktop) running under VMware Workstation 17 player on a Windows 10 host computer. Some familiarity with Linux would be useful for this stage.

A sample assembly program “Powers” appears in the appendix.

4.6 Trouble Shooting

- Compile with the exact tool chain versions used for creating this system (see source code)
- Attempt to duplicate the problem in an online simulator for example Wokwi and see [Ref 6] for a ready made configuration.

- If the optional led matrices are not functioning, check the configuration options within the code and adapt if necessary. Only two display types, both variants of MAX7219 based matrix displays, are directly supported.
- If Data entered at the serial console appears not to be accepted ensure that the configuration of the serial console according to the usage instructions.

4.7 Next Stage

- Possible use of the simulation as a test bed to replace the original but obsolete SN74LS382 ALU chip for a more easily obtainable 74HCT181
- Design interface and mods to integrate hardware modules, possibly starting with the ALU module together with the ALU register and Register B

5 Appendix

5.1 Other JB 8bit Computer inspired developments

This is a short list with comments of some of the similar projects derived at least in part from the James Bates 8bit computer found by a quick search and assessment

- Build only but with explanations https://projects.descan.com/stuff/Project_8_bit_compi.pdf
- Very well documented. Enhancements such as 16bit address bus plus videos https://github.com/DerULF1/8bit-computer
- An interesting PCB version but with some modules replaced with MCUs. https://www.reddit.com/r/beneater/comments/ctcptl/a_ben_eater_james_bates_inspired_design/?rdt=58668
- Another variant build with prototype boards on a back plane and a nice console. Including schematics together with a simulation and FPGA implementation and videos. https://www.dijkens.com/mdComputer8/
- PCB version with interesting ALU design but unfinished. https://www.dvatp.com/tech/eight_bit_cpu
- An excellent project including tools, a web frontend and emulator with the emphasis on visual fidelity with the breadboard build https://forums.overclockers.com.au/threads/web-based-emulator-of-my-ben-eater-inspired-8-bit-computer.1271445/ https://github.com/visrealm/vrcpu

Maybe there are many more which are not so easily searchable.

5.2 References

[Ref 1] James Bates Youtube video series describing his 8bit computer https://youtu.be/RNVJYGRNvwU

[Ref 2] James Bates Github repository which includes documentation, schematics etc. for his 8bit computer https://github.com/jamesbates/jcpu

[Ref 3] Ben Eater 8bit Computer Ben Eater https://eater.net/8bit/

[Ref 4] Digital Computer Electronics by Albert Paul Malvino ISBN 978-0074622353 (also available as a PDF download from various sources)

[Ref 5] Possible supplier of the rare SN74LS382 ALU chip used in the hardware design: Grieder Elektronik Bauteile AG, Switzerland. https://shop.griederbauteile.ch/product_info.php?cPath=25_27_119&products_id=6610

[Ref 6] Wokwi online simulation of the ESP32 simulator described in this document. https://wokwi.com/projects/374430064307724289

[Ref 7] ESP32 tutorial: https://lastminuteengineers.com/getting-started-with-esp32/

[Ref 8] ESP32 tutorial: https://randomnerdtutorials.com/getting-started-with-esp32/

5.3 Software Source Packet

Arduino Forum Exhibition Gallery (under author 6v6gt)

https://forum.arduino.cc/t/emulator-of-a-8-bit-breadboard-logic-chip-teaching-computer/1168240

5.4 CONTROLLER SIGNAL LIST

Bit		Name	Description
0		ALC	ALU 74LS382 Carry in
1		ALS0	ALU 74LS382 Function Input S0
2		ALS1	ALU 74LS382 Function Input S1
3		ALS2	ALU 74LS382 Function Input S2
4		_ALB	ALU 74LS382 B operand source = Reg_B when asserted or zero otherwise
5		_ALW	Bus-> Reg_ALU (Write Bus to Reg_ALU)
6		_ALE	Reg_ALU -> Bus (Enable Reg_ALU to Write to Bus)
7		PCC	Increment Program Counter
8		_SPW	Bus -> Reg_SP (stack pointer)
9		_SPE	Reg_SP -> Bus (stack pointer)
10		_PCW	Bus -> Reg_PC (program counter)
11		_PCE	Reg_PC -> Bus (program counter)
12		_RaW	Register -> Bus
13		_RaE	Bus -> Register
14		_RbW	Register -> Bus
15		_RbE	Bus -> Register
16		_RcW	Register -> Bus
17		_RcE	Bus -> Register
18		_RdW	Register -> Bus
19		_RdE	Bus -> Register
20		_IRW	Bus -> Reg_IR (instruction register)
21		_MAW	Bus -> Reg_MAW (memory address register)
22		PGM	Select program RAM bank
23		_HLT	Halt
24		_MW	Bus -> memory cell pointed at by Reg_MAR (also PGM)
25		_ME	memory cell pointed at by Reg_MAR (also PGM) -> Bus
26		unused	
27		unused	
28		unused	
29		unused	
30		unused	
31		_TR	Controller Micro Timer reset to 0

The underscore prefix '_' indicates that the signal has negative polarity, that is zero when asserted and 1 otherwise.

5.5 INSTRUCTION LIST

Opcode	Instruction
0x00	"NOP" ,
0x01	"MOV Ra, Rb" ,
0x02	"MOV Ra, Rc" ,
0x03	"MOV Ra, Rd" ,
0x04	"MOV Ra, SP" ,
0x05	"MOV Ra, PC" ,
0x06	"undefined" ,
0x07	"DATA Ra, #IMM"
0x08	"MOV Rb, Ra" ,
0x09	"MOV Rb, Rb" ,
0x0A	"MOV Rb, Rc" ,
0x0B	"MOV Rb, Rd" ,
0x0C	"MOV Rb, SP" ,
0x0D	"MOV Rb, PC" ,
0x0E	"undefined" ,
0x0F	"DATA Rb, #IMM"
0x10	"MOV Rc, Ra" ,
0x11	"MOV Rc, Rb" ,
0x12	"MOV Rc, Rc" ,
0x13	"MOV Rc, Rd" ,
0x14	"MOV Rc, SP" ,
0x15	"MOV Rc, PC" ,
0x16	"undefined" ,
0x17	"DATA Rc, #IMM"
0x18	"MOV Rd, Ra" ,
0x19	"MOV Rd, Rb" ,
0x1A	"MOV Rd, Rc" ,
0x1B	"MOV Rd, Rd" ,
0x1C	"MOV Rd, SP" ,
0x1D	"MOV Rd, PC" ,
0x1E	"undefined" ,
0x1F	"DATA Rd, #IMM"
0x20	"MOV SP, Ra" ,
0x21	"MOV SP, Rb" ,
0x22	"MOV SP, Rc" ,
0x23	"MOV SP, Rd" ,
0x24	"MOV SP, SP" ,
0x25	"MOV SP, PC" ,
0x26	"undefined" ,
0x27	"DATA SP, #IMM"
0x28	"JMP Ra" ,
0x29	"JMP Rb" ,
0x2A	"JMP Rc" ,
0x2B	"JMP Rd" ,
0x2C	"JMP SP" ,
0x2D	"HLT" ,
0x2E	"undefined" ,

```
0x2F        "JMP #imm"  ,
0x30        "undefined"  ,
0x31        "undefined"  ,
0x32        "undefined"  ,
0x33        "undefined"  ,
0x34        "undefined"  ,
0x35        "undefined"  ,
0x36        "undefined"  ,
0x37        "undefined"  ,
0x38        "JC #imm"  ,
0x39        "JZ #imm"  ,
0x3A        "JN #imm"  ,
0x3B        "undefined"  ,
0x3C        "JO #imm"  ,
0x3D        "undefined"  ,
0x3E        "undefined"  ,
0x3F        "undefined"  ,
0x40        "LOD Ra, [Ra]"
0x41        "LOD Ra, [Rb]"
0x42        "LOD Ra, [Rc]"
0x43        "LOD Ra, [Rd]"
0x44        "LOD Ra, [SP]"
0x45        "LOD Ra, [PC]"
0x46        "POP Ra"  ,
0x47        "LOD Ra, [#imm]
0x48        "LOD Rb, [Ra]"
0x49        "LOD Rb, [Rb]"
0x4A        "LOD Rb, [Rc]"
0x4B        "LOD Rb, [Rd]"
0x4C        "LOD Rb, [SP]"
0x4D        "LOD Rb, [PC]"
0x4E        "POP Rb"  ,
0x4F        "LOD Rb, [#imm]
0x50        "LOD Rc, [Ra]"
0x51        "LOD Rc, [Rb]"
0x52        "LOD Rc, [Rc]"
0x53        "LOD Rc, [Rd]"
0x54        "LOD Rc, [SP]"
0x55        "LOD Rc, [PC]"
0x56        "POP Rc"  ,
0x57        "LOD Rc, [#imm]
0x58        "LOD Rd, [Ra]"
0x59        "LOD Rd, [Rb]"
0x5A        "LOD Rd, [Rc]"
0x5B        "LOD Rd, [Rd]"
0x5C        "LOD Rd, [SP]"
0x5D        "LOD Rd, [PC]"
0x5E        "POP Rd"  ,
0x5F        "LOD Rd, [#imm]
0x60        "LOD SP, [Ra]"
0x61        "LOD SP, [Rb]"
0x62        "LOD SP, [Rc]"
0x63        "LOD SP, [Rd]"
0x64        "LOD SP, [SP]"
0x65        "LOD SP, [PC]"
0x66        "POP SP"  ,
```

0x67	"LOD SP, [#imm]"
0x68	"LOD PC, [Ra]"
0x69	"LOD PC, [Rb]"
0x6A	"LOD PC, [Rc]"
0x6B	"LOD PC, [Rd]"
0x6C	"LOD PC, [SP]"
0x6D	"LOD PC, [PC]"
0x6E	"RET" ,
0x6F	"LOD PC, [#imm]
0x70	"undefined" ,
0x71	"undefined" ,
0x72	"undefined" ,
0x73	"undefined" ,
0x74	"undefined" ,
0x75	"undefined" ,
0x76	"undefined" ,
0x77	"undefined" ,
0x78	"undefined" ,
0x79	"undefined" ,
0x7A	"undefined" ,
0x7B	"undefined" ,
0x7C	"undefined" ,
0x7D	"undefined" ,
0x7E	"undefined" ,
0x7F	"undefined" ,
0x80	"STO [Ra], Ra"
0x81	"STO [Ra], Rb"
0x82	"STO [Ra], Rc"
0x83	"STO [Ra], Rd"
0x84	"STO [Ra], SP"
0x85	"STO [Ra], PC"
0x86	"undefined" ,
0x87	"undefined" ,
0x88	"STO [Rb], Ra"
0x89	"STO [Rb], Rb"
0x8A	"STO [Rb], Rc"
0x8B	"STO [Rb], Rd"
0x8C	"STO [Rb], SP"
0x8D	"STO [Rb], PC"
0x8E	"undefined" ,
0x8F	"undefined" ,
0x90	"STO [Rc], Ra"
0x91	"STO [Rc], Rb"
0x92	"STO [Rc], Rc"
0x93	"STO [Rc], Rd"
0x94	"STO [Rc], SP"
0x95	"STO [Rc], PC"
0x96	"undefined" ,
0x97	"undefined" ,
0x98	"STO [Rd], Ra"
0x99	"STO [Rd], Rb"
0x9A	"STO [Rd], Rc"
0x9B	"STO [Rd], Rd"
0x9C	"STO [Rd], SP"
0x9D	"STO [Rd], PC"
0x9E	"undefined" ,

```
0x9F        "undefined"  ,
0xA0        "STO [SP], Ra"
0xA1        "STO [SP], Rb"
0xA2        "STO [SP], Rc"
0xA3        "STO [SP], Rd"
0xA4        "STO [SP], SP"
0xA5        "STO [SP], PC"
0xA6        "undefined"  ,
0xA7        "undefined"  ,
0xA8        "STO [PC], Ra"
0xA9        "STO [PC], Rb"
0xAA        "STO [PC], Rc"
0xAB        "STO [PC], Rd"
0xAC        "STO [PC], SP"
0xAD        "STO [PC], PC"
0xAE        "undefined"  ,
0xAF        "undefined"  ,
0xB0        "PUSH Ra"  ,
0xB1        "PUSH Rb"  ,
0xB2        "PUSH Rc"  ,
0xB3        "PUSH Rd"  ,
0xB4        "PUSH SP"  ,
0xB5        "CALL Rc"  ,
0xB6        "undefined"  ,
0xB7        "PUSH #imm"  ,
0xB8        "STO [#imm], Ra
0xB9        "STO [#imm], Rb
0xBA        "STO [#imm], Rc
0xBB        "STO [#imm], Rd
0xBC        "STO [#imm], SP
0xBD        "STO [#imm], PC
0xBE        "undefined"  ,
0xBF        "undefined"  ,
0xC0        "INC Ra"  ,
0xC1        "INC Rb"  ,
0xC2        "INC Rc"  ,
0xC3        "INC Rd"  ,
0xC4        "undefined"  ,
0xC5        "SUB Rb, Rb"  ,
0xC6        "undefined"  ,
0xC7        "undefined"  ,
0xC8        "SUB Ra, Rb"  ,
0xC9        "undefined"  ,
0xCA        "SUB Rc, Rb"  ,
0xCB        "SUB Rd, Rb"  ,
0xCC        "ADD Ra, Rb"  ,
0xCD        "ADD Rb, Rb"  ,
0xCE        "ADD Rc, Rb"  ,
0xCF        "ADD Rd, Rb"  ,
0xD0        "XOR Ra, Rb"  ,
0xD1        "XOR Rb, Rb"  ,
0xD2        "XOR Rc, Rb"  ,
0xD3        "XOR Rd, Rb"  ,
0xD4        "OR Ra, Rb"  ,
0xD5        "OR Rb, Rb"  ,
0xD6        "OR Rc, Rb"  ,
```

```
0xD7        "OR Rd, Rb"  ,
0xD8        "AND Ra, Rb"  ,
0xD9        "AND Rb, Rb"  ,
0xDA        "AND Rc, Rb"  ,
0xDB        "AND Rd, Rb"  ,
0xDC        "NOT Ra"  ,
0xDD        "NOT Rb"  ,
0xDE        "NOT Rc"  ,
0xDF        "NOT Rd"  ,
0xE0        "DEC Ra"  ,
0xE1        "DEC Rb"  ,
0xE2        "DEC Rc"  ,
0xE3        "DEC Rd"  ,
0xE4        "undefined"  ,
0xE5        "SBC Rb, Rb"  ,
0xE6        "undefined"  ,
0xE7        "undefined"  ,
0xE8        "SBC Ra, Rb"  ,
0xE9        "undefined"  ,
0xEA        "SBC Rc, Rb"  ,
0xEB        "SBC Rd, Rb"  ,
0xEC        "ADC Ra, Rb"  ,
0xED        "ADC Rb, Rb"  ,
0xEE        "ADC Rc, Rb"  ,
0xEF        "ADC Rd, Rb"  ,
0xF0        "undefined"  ,
0xF1        "undefined"  ,
0xF2        "undefined"  ,
0xF3        "undefined"  ,
0xF4        "CMP Rb, Ra"  ,
0xF5        "CMP Rb, Rb"  ,
0xF6        "CMP Rb, Rc"  ,
0xF7        "CMP Rb, Rd"  ,
0xF8        "CMP Ra, Rb"  ,
0xF9        "undefined"  ,
0xFA        "CMP Rc, Rb"  ,
0xFB        "CMP Rd, Rb"  ,
0xFC        "TST Ra"  ,
0xFD        "TST Rb"  ,
0xFE        "TST Rc"  ,
0xFF        "TST Rd"  ,
```

5.6 SAMPLE PROGRAM POWERS

This is one of the pre-built sample programs the object code of which can be selected and run from the console.

```
; Ver 12.02.2023 power3_v03.s
; Show all power series (n^1, n^2, n^3, ...) up to base 15. Limited to 255
;
;
; core logic:
; n^E = n^(E-1) * n = n^(E-1) + n^(E-1) .... + n^(E-1) [repeated n times]
; where n is the base and E is the exponent.
; That is, previous displayed power added to itself base-1 times
; gives new displayed power
; Increment exponent until result too big then start with next base.
; Increment base and repeat until > 15.
;
                          .org 0
start:                    data Rd, #1
                          sto [#0], Rd       ; ram[#0] base = 1
;
newBase:                  lod Rd, [#0]       ; base
                          inc Rd             ; base++
                          sto [#0], Rd       ; base
                          data Rb, #15
                          and Rd,Rb          ; test if base outside range 1 to 15 (0x0F)
                          jz #start          ; base > 15 goto start
                          lod Ra, [#0]       ; RegA always holds last calculated power for
                                             ; current base. initial value = base
;
newExp:                   mov Rc, Ra         ; last calculated power for current base to accum
                          mov Rb, Ra         ; RegB is addend
                          lod Rd, [#0]       ; multiplier (initial value as base - 1 )
                          dec Rd             ; multiplier--
;
innerAdd:                 add Rc, Rb         ; add old power to self then base-1 more times
                          jc #newBase        ; too big so start with next base
                          dec Rd             ; multiplier--
                          jz #showPower      ; multiplier == zero. We have a new power to show
                          jmp #innerAdd      ; keep adding base to accum
;
showPower:                mov Ra, Rc         ; loading to RegA shows it on display
                          jmp #newExp        ; next exponent with current base.
```