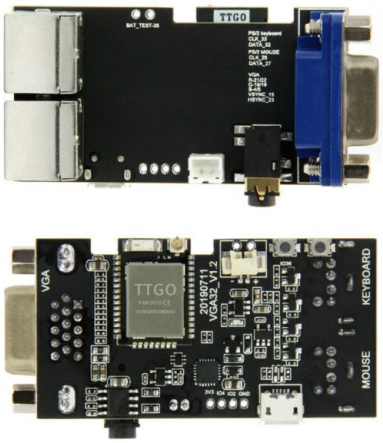
Eris Hardware

Written by Paul Robson March/April 2020

This is a design for a cheap to design, easy to implement, but relatively powerful 80s style home computer, known as “Eris”, for no other reason that it must have a name. I was going to call it “Pluto” but there’s already a computer called that …. so, Eris next.

It is currently implemented in two forms, both of which are emulation. However, it is technically feasible to port it to FPGA and it is designed with that in mind. The first form is a classic emulator, which should run on Linux, Windows and things like a Raspberry Pi. The emulator uses the SDL2 library and nothing else.

The second is the ESP32 code, which is designed to run on boards which support the FABGL hardware standard or compatible. The circuitry is relatively simple, and readymade boards are available on eBay.



The board on the right is a “TTGO VGA32 V1.2” board which as you can see has VGA out and 2 PS/2 inputs. The ESP32 code will run directly on this board. The V1.1 board is more commonly available but does not have audio. (the board is almost identical except for the jack socket). At the time of writing (early 2020) these were about £12-£13.

The system is designed for straightforward implementation and reasonable system efficiency in both FPGA and Emulation. The code does not take advantage of being an emulator per se, so for example while the instruction set has a 16x16 multiply it does not have a divide instruction (which is much more complicated in FPGA).

Likewise, the video display, is a bitmap updated by a blitter (which can be synchronous or asynchronous) is reasonably efficient but uncomplicated, thus providing useable graphics without being complex to emulate either in code or FPGA.

Specification

The specification is as follows: -

CPU	The CPU is designed to be reasonably powerful and easy to implement. It is a 16-bit CPU with a highly orthogonal instruction set. It borrows ideas from ARM, CP1600 and IMP16. The speed is theoretically unlimited. Current test runs on the base platform (24k RAM ESP32) show a speed (without blitting) of 0.94 MIPS. For comparison, a 4Mhz Z80 is about 0.58 MIPS.
Memory	Memory is ROM memory, from 0000-3FFF, and RAM memory, from 4000 to FEFF. Memory is word orientated. The upper 256 words are memory mapped input/output registers. The ESP32 implementation has 32k of RAM, so C000-FEFF are unused.
Display	The Display is 320 x 240 x 16 colours which have a definable palette in digital RGB (e.g. 8 colours). This memory is not directly accessible by the CPU, being accessed via the Blitter which can draw bitmap graphics in a single instruction. It does not affect the available program memory either as it is not memory mapped.
Blitter	The blitter is designed to draw on the display with reasonable efficiency, currently in a ‘bitmap’ form, e.g. it can draw a binary bit pattern in a specific colour. The blitter runs asynchronously by design but can be implemented either way.
Sound	Sound is one noise channel and two square wave channels.
Keyboard	A PS/2 connector is used for the keyboard, which is likely to involve a USB to PS2 adaptor as PS/2 keyboards are becoming rarer. The Arrow keys, Ctrl and Shift are synonyms for a game controller. The current implementation is for a UK keyboard layout, which is easy to change and will be generalised later. There are 6 programmable function keys.

Processor

Eris has a 16-bit processor, which access a 64k x 16-bit word accessed memory, divided into ROM (0000-3FFF) and RAM and I/O (4000-FFFF). All the I/O is mapped from FF00-FFFF.

The speed is approximately 1.0 MIPS (approximately twice the speed of a 4Mhz Z80 in terms of instructions). However, all instructions are 16 bits, as are most data operations, so this is somewhat faster than that. The ESP32 implementation should be considered the slowest available, running at about 1 MIPS (see note). Time synchronisation should not rely on CPU speed but be done using the running timer.

There are 16 x 16-bit registers, and a single flag, which holds the result of the carry out on addition or subtraction. These are known as R0-R15.

All registers are general purpose, except R15 which is the program counter. Their functions are interchangeable. However, by convention, R14 is zero (this allows the use of short constants), R13 is the “Link” register, and R12 the stack pointer which moves downwards through memory.

Instruction Format

The instructions all have the same format and are occupy either one or two words. The first word has the format below; the second word, when used, is a 16-bit integer constant or address

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
i	i	i	i	a	a	a	a	b	b	b	b	c	c	c	c

There are four four-bit parts to each instruction, I, a, b and c

iiii is a four-bit code representing the instruction

- aaaa is a four-bit code representing the target register, 0-15
- bbbb is a four-bit instruction representing the source register 0-14
- cccc is a four-bit constant representing the values 0-15

Instructions all have two parameters, the target register and the value. The target register (the value aaaa) represents a register from R0 to R15.

The value can be either the contents of the source register 0-14 plus the constant, or alternatively if it is 15 it is the value in the next memory location. In this case the constant value should be 0000.

So, to use an example, the AND instruction (like the others) has two forms.

and 0,2,4 the target register is R0, the value is R2 + 4

and 0,15,0 the target register is R0 and the value is 13207.

word 13207

To simplify this in assembler they are written as ***and r0,r2,#4*** and ***r0,#13207*** respectively.

This is the reason for the convention of R14 being zero. By having this as zero, one can write ***and r0,r14,#7*** and this is anding r0 with 7, because the value is r14+7 and r14 is zero. This is done automatically – if you do and r0,#255 you get and r0,r15,#0 ; word 255, but and r0,#15 gets you and r0,r14,#15 ; optimising short constants.

Note: The blitter on the ESP32 stops to the processor to do the blit. This is not a requirement of the specification but a function of how it is emulated; so, if a lot of “blitting” is performed the effective MIPS rate may drop significantly.

Even on a system where the blitter runs asynchronously the processor may have to wait for the blitter to finish its current activity if it is doing a sequence ; in this case the MIPS will be much the same, but time may be spent in a wait loop if there is no background housekeeping.

On emulators some operations are simply quicker than others anyway, so this figure varies.

Reset State

At reset all the registers have valid but unknown values, except for R15, the program counter, which is zero. The timer, keyboard latch, and blitter state is unknown.

Turbo Mode

On the emulation by writing to $FFFD a zero or -1, you can turn turbo mode off or on. This is not guaranteed, so cannot be used to make a game faster. However, for testing and development, it can be useful. If for example, your program takes 10 seconds to create data tables when it runs, this can be a pain when testing. Turning turbo on will make this much quicker (default is times 10). Note though that while it will not damage anything permanently timers and so on are not guaranteed to work in Turbo mode. Use with caution.

Instruction set

In this description target or Ra refers to the ‘aaaa’ register, and value, or X refers to either Rb+constant or the next value.

Instruction		Action
0000	mov	Load Ra with constant X
0001	ldm	Load Ra with the contents of memory location X
0010	stm	Save Ra to memory location X (in this instruction the “target” register is the source)
0011	add	Add X to Ra. The carry from the Rb+c calculation is ignored.
0100	adc	Add X and carry to Ra, as add but with carry in/carry out
0101	sub	Subtract X from Ra. This is done using two’s complement e.g. Ra := Ra + ((~X) + 1).
0110	and	And Ra with X
0111	xor	Exclusive or Ra with X
1000	mult	Multiply Ra by X (16 bits unsigned). The carry flag is set if the result does not fit in 16 bits, clears otherwise.
1001	ror	Rotate Ra right by X (0...15)
1010	brl	Branch Link performs the functions of JMP, JSR and RET. It’s action is to copy the current PC, R15 into the target register, and then set the PC to X. Different combinations allow the return address to be saved in the link register to call a subroutine (BRL R13,<address>), to be lost (BRL R15,<address>) and to restore the address from the link register (BRL R15,R13).
1011	skeq	Skip if Ra = value. Skips can be either one or two bytes depending on the bbbb value of the instruction being skipped.
1100	skne	Skip if Ra != value
1101	skse	Skip if the sign bits only (e.g. R15) of Ra and value are the same
1110	sksn	Skip if the sign bits only of Ra and value are different.
1111	skcm	Skip if the carry matches bit 0 of the value. Note that in this opcode Ra is not used and should be 1111.

Note, internally code $0000 is used for a breakpoint in the emulator, as

this instruction (mov r0,r0,#0) does absolutely nothing to the machine state.

Standard Synonyms

The normal format of an instruction is

<opcode>,<reg1>,<reg2>,#<constant>

A simplification and optimisation are permitted which can decide whether to use short or long constants. This is <opcode> <reg1>,#<constant> ; it is a synonym for <opcode> <reg1>,r14,#<constant> or alternatively <opcode> <reg1>,r15,#0 / word <constant> depending upon the value of constant (r14 is normally zero) If not defined yet it is assumed to be the latter as a forward label.

Some standard synonyms exist as follows

Macro	Equivalent to
jmp @1	brl r15,@1
jsr @1	brl r13,@1
ret	brl r15,r13,0
inc @1	add @1,#1
dec @1	sub @1,#1
clr @1	xor @1,@1,#0
sk* @1	Skip on *, which can be z, nz, p or m, tested on @1
skge sklt sknc skc	Tests the carry flag, used after an arithmetic operation. When used with ‘sub’ then ge/lt apply to the comparison
break	Compiles a break used in the emulator, ignored in the hardware

Branch-Link is the instruction that is the most unusual, as it has three functions. It copies its second operand to the PC, and the original PC to its first. So, it has (at least) three uses :

- The first operand is the PC itself. In this case the value is lost, so it just jumps.
- The first operand is the LINK register, in which case the old PC value can be saved and reused, which makes it a subroutine call
- The first operand is the PC and the second operand is the LINK register, so the LINK goes back to the PC, making it a return.

To have nested subroutine calls requires the LINK register to be saved, which is why many assembler subroutines push and pop LINK.

Assembler Commands

The assembler's basic syntax is the two basic syntaxes

<cmd> <reg>,<reg>,#<expr>

<cmd> <reg>,#<expr>

The first corresponds to the normal command format. The second either uses short or long constants depending on the expression. If the expression is undefined on pass 1, then it is assumed to be long (this should be a forward branch normally). If it isn't for some reason, then this will show up as a redefinition error (as the labels will move)

In the second format the expression cannot be zero, so you cannot write mov r0,#0 (you can write clr r0, xor r0,r0,#0 ; mov r0,r14,#0 and other things which do the same thing !)

Additionally:

.<label> Defines a label, there may be code after this or not.

; Defines a comment

: Use to separate multiple instructions on the same line

Values R0-R15, PC, DEF, LINK and SP are predefined (the latter 4 to 15,14,13 and 12). For the assembler to work correctly in anything other than the normal command syntax, R14 must be zero. There may be freak reasons to change it, but on your own head be it.

The following pseudo-operations exist.

Command	Action
byte a,b,c,d	Compiles bytes in word format, first byte low in word, must be an even number of bytes.
word a,b,c	Compiles words in line
text "text"	Compiles text in line, padded with zero if needed.
string "text"	Compiles text in line with a preceding length byte
org a	Set write origin to a (defaults to zero)
fill a	Allocate a words, but don't write anything to the code.

Keyboard

The keyboard is accessed via address $FF00. To select a row or rows, write a value to $FF00. This is latched. $FF00 can then be read to get the bit pattern. Keys are active high when pressed. The following keys are pressed , red being the shifted values (other than the alphabet) (this is a UK key layout).

	0	1	2	3	4	5	6	7	8	9	10	11	12
	!	"	#	$	%	^	&	*	(	)		+	
Bit 0	1	2	3	4	5	6	7	8	9	0	-	=	BS
Bit 1	TAB	Q	W	E	R	T	Y	U	I	O	P	[	]
										:	@		
Bit 2	A	S	D	F	G	H	J	K	L	;	‘	#	RET
										<	>	?	
Bit 3	SPC	\	Z	X	C	V	B	N	M	,	.	/	
Bit 4	F1	F2	F3	F4	F5	F6							
Bit 5	Left	Right	Up	Dwn	Ctrl	Shift							

The bit 4 row is Joystick and Cursor keys, which are duplicated on the keyboard and joystick - Ctrl is Fire A and Shift Fire B. Further joysticks can be added using bit 5,6 and 7 similarly.

Control keys normally mask out the lower 5 bits e.g. Ctrl-A => 1. However, the “function keys”, numbered F1-F6 can also be produced by Ctrl 1-6 (these generate key codes 241-246)

Blitter

The blitter is responsible for all communication with the display

The palette is accessed via writing to $FF10. The format is the palette entry is in the MSB, and the colour in 00000BGR format is in the MSB.

When the palette is updated the screen should be cleared ; it can be implemented either as a look up or a translate on write, so it cannot be used for palette effects e.g. changing it to flash the display as in for example the end of a level in Pacman.

The blitter is simple. It can draw a 16 x 255 pixel block, in a given colour, and can write in either ‘sprite’ mode, where it writes on top of what is there, or ‘background’ mode, where it clears the background as it writes.

The blitter chip is placed at $FF20-$FF2F and its registers are as follows, all are write only

Addr	Register	Description
$FF20	X Position	This is the x coordinate where the writing takes place. It is a 16-bit register, that wraps round, so sending out $FFF8 will draw just the last 8 pixels. It does not change.
$FF21	Y Position	This is the Y coordinate where writing takes place, this wraps around similarly. However, this increments with each written line, so at the end of a blitter command it will be set to the row below the last written line
$FF22	Data Source	Points to 16-bit data in memory, which is used for pixels (e.g. it represents a 16x1 pixel block). This register can be incremented for each row (when drawing say characters) or not (say when drawing solid blocks)
$FF23	Colour/ Mask	This is split into two. The lower byte indicates the colour to be written. The upper byte indicates the bits in the video ram data to be written to. This allows partial drawing and thus the display can be divided into multiple bit planes which can be written separately. On the base machine (with 4 bits per pixel), the mask is 00001111 and the colour depends on … the colour
$FF24	Command	Writing here starts a blit command and sets the status bit until it is finished. Implementations can be synchronous or asynchronous.

Additionally, the address can be read ($FF20 is used), and if bit 15 of this is logic ‘1’ the blitter is busy, so the CPU should wait for it to finish. Once it has finished, it will be available until the command register is written again.

Blitter Command Word

The blitter command word is set up as follows.

Bit(s)	Purpose
15	This controls whether the data source register is incremented or not. If this is clear, it is incremented – this is used for characters where successive memory locations form the graphic. For solid bars, this can be set, and the data source register will not change, repeating the same pixels.
14	If set this horizontally flips the pixels. Normally bit 15 is on the left and bit 0 is on the right; when set these are reversed.
13	If set the exclusive Ors the data source register address with $000F. The effect of this can be to vertically flip a graphic, if it is 16x16 pixels and the data source register is on a 16-word boundary.
12	If set, the sprite is double height and width.
11	If set, the video ram is written to whatever the pixels – the pixel determines whether the colour in the Colour/Mask register is written or zero, the background. If this bit is cleared, it writes ‘sprite style’ e.g. the video ram is only written when the pixel is one, superimposing the graphic on top of what is already there.
10-8	All 0, for future expansion
7-0	Vertical height, the number of vertical rows to be done, from 0-255. If this is zero nothing happens.

Timer

There is a 16-bit 100Hz timer at $FF30 – this is read only.

Audio

There are three sound channels, one is a noise channel at $FF40 and the others are square wave channels at $FF41 and $FF42

The pitch of the channels is set by writing a 16-bit value to the given address. This is used via a programmable divider to calculate the pitch, which is derived from a 5 MHz clock – so the frequency is 5,000,000 / n. If n is zero, the channel is silent.

This gives a range starting at about 75Hz which is a low hum, up to beyond the audible hearing range of a person.

Storage

The following functions are provided by the OSFileOperation call in the Kernel.

Command	Parameters	Notes
Initialise	R0:$00	Initialise the file system
Load	R0:$01 R1:Name	Load a file in at is given address
Load Elsewhere	R0:$02 R1:Name R2:Target Addr	Load a file to a given target address
Save	R0:03 R1:Name R2:Start Addr R3:Size	Write a section of memory to a given file.
Load Directory	R0:04 R1:Target	Load directory into memory at target. This is a list of filenames separated by spaces ended with a null string. This is uncompressed – 1 character per word.
Check Existence	R0:05 R1:Name	Checks the existence of a file (e.g. 0 if successful, file exists)
Get Load Address	R0:06 R1:Name	Returns the load address of a file (value unpredicatable if it doesn't exist)
Get File Size	R0:07 R1:Name	Returns the word size of the file *not including the load address which is the first word of the file.* This should be the 8 bit file size / 2 -1
Download	R0:08 R1:Download	The download is a three part string seperated by semicolons. The first is a URL to download, the second an SSID and the third a password. Stores the file in the filesystem using the URL stub.
Delete	R0:09 R1:Name	Delete the file if it exists.

The Files are given an alphanumeric name, which can include a '.', of up to 16 characters. Case is always ignored. R0 normally returns 0 if

successful and non-zero if an error has occurred.

Storage is on a single device (multiple selectable devices will be added later). On the emulator this is kept in a directory called “storage”, on the ESP32 in the SPIFFS filing system. Files are in the .prg format, e.g. there is a two byte load address prefix.

Text Display

The graphic display is 320 x 240 pixels using a 5 x 7 font. This makes a 53 x 30 character display. This is purely done in software.

A mirror of this display is held in memory for scrolling and screen editing. This contains the character in the cell in the BG/FG/Char format used in the OSDrawCharacter routine.

There are two types of spaces. Character code 32, ASCII space, is a code space. Character code 0 is a blank space, which is written on carriage returns, scrolling and clear screen and is used for unused space. So, when trying to identify code over more than one lines the screen editor uses the 0 character code or the end of screen memory to identify the end of the line. If the last character of (say) a program listing is on the last character of the line, the CR that follows will insert a blank line (because the cursor is already on the line below, having advanced).

Character codes used in the OSPrintCharacter () method are as follows: -

Codes	Purpose
0	Ignored, null
1	Moves cursor left, stops at top left
2	Move cursor right, stops at bottom right
3	Move cursor up
4	Move cursor down – this does not cause scrolling
5	Insert a space at the current cursor. Does not move cursor. Will not scroll text down to make room.
8	Backspace. If not the first character on the line, go back one space and erase the character there to a space.
9	Advance to next tab stop
11	Home Cursor to (0,0)
12	Clear screen to current background and home cursor
13	Carriage Return - Write spaces until at the start of the line (prints at least one)
14	No-overwrite carriage return – just moves the cursor.
15	Swap foreground/background colours
16-31	Set colour to 0-15
32-127	Display character at cursor position, advance by one-character space. If off the bottom scroll the screen up.

Other control codes are similarly ignored but this may change.

Tilemap

The tilemap has the following data structure

Offset	Value
+0	$ABCD – this identifies it as a tilemap.
+1	Size of tiles. Currently only supports 16x16, hence this is &0010
+2	Width of tile map
+3	Height of tile map
+4	Control values. Bits 0..2 are the colour drawn when you scroll off the tile map.
+5	Tile at (0,0)
+6	Tile at (1,0)

The total size of the tilemap is width * height + 5. Tiles are stored in row order, column by column, so the offset to tile x,y is x + y * width + 5

Each word describes a single tile.

Bits	Meaning
15	0
14,13	Horizontal and vertical flip bits as per the blitter
12..11	0
10..8	Colour the tile is drawn in
7..0	Tile image index.

Special addresses

$FFF0-$FFFF are reserved for ‘trick’ addresses.

Currently, writing to $FFFE triggers a file operation, writing to $FFFD turns the emulator turbo on/off (should only be used for things like setting up in games) and writing to $FFFC causes the character written to be sent to external hardware (screen, ESP32 debug port). This latter is used to get data out of storage such as SPIFFS which is not easily read.