Eris 32 CPU

Written by Paul Robson 15th May 2020

This is a draft specification for the 32 bit version of the Eris retrocomputer.

Hardware

The processor has 32 x 32 bit registers, known as R0-R31. Registers are interchangeable, except for R31 which is the program counter.

As with Eris, there are at least three reserved registers, one for the return address from routines, one to hold constant zero, and one as the stack pointer. A register is also reserved for the brr to hold unwanted link data.

Memory has shifted from being word addressed (64k x 16) to being byte addressed (theoretical maximum of 4G x 8). Hence rather than instruction words being at 0,1,2,3 .... they are now at 0, 4, 8, 12 ....

Instructions and long reads must be word aligned. Data is stored in memory in little endian order.

Instruction word

The instruction word has the same basic consistent format, however it has been extended, to have an additional specified register and conditional execution bits

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Condition			Opcode					Rtarget					Rsource					Rvalue					Constant									

As before, all instructions operate identically.

Phase 1 : Calculating the value

The effective value is calculated. This is something like the EAC on an 8 bit processor, but it is often used as a value.

This can have one of two values. Either the contents of Register rValue added to the 9 bit constant (this constant is now signed), or alternatively if Rvalue is 11111 (e.g. the program counter), the effective value is the next long word in memory. This allows single instruction constants from -512 .. 511 and long constants in the full 32 bit range.

Phase 2 : Checking the condition

The upper 3 bits are checked to see if the command is to be executed. As with the ARM design, all opcodes are conditional. These bits have the following values.

Bit 31	Bit 30	Bit 29	Mnemonics	Effect
0	0	0	None	Always execute
0	0	1	Z	Execute if zero flag set
0	1	0	MI	Execute if sign bit set
0	1	1	C / GE	Execute if carry flag set
1	0	0	-	Reserved *(overflow possibly)*
1	0	1	NZ	Execute if zero flag clear
1	1	0	PL	Execute if sign bit clear
1	1	1	NC / LT	Execute if carry flag clear

The flags are set as a consequence of the instructions. The zero and sign flags are set to the result of the final value of the calculation in all instructions ; this is the value in Rtarget at the end of the instruction.

These flags are not updated if the instruction is not executed.

The carry flag is set/cleared following add and subtract operations and not affected by any other operation.

Phase 3 : Executing the instruction

If the condition is successful, the instruction is executed. Most instructions are of the form

Rtarget ← Rsource <operation> effective value

or in its two extended forms.

Rtarget ← Rsource <operation> (Rvalue + 9 bit constant)
Rtarget ← Rsource <operation> 32 bit constant

When the instruction is completed the value in Rtarget is used to set the Zero and Sign flags.

Reset State

The processor reset state is R31 (the program counter) is zero. All other registers do not have a specific value.

Instruction Set

This is the base instruction set.

Mnemonic	Function	Notes
MOV	Rtarget = value	Register → Register movement
LDM	Rtarget = Mem32[value]	32 bit read
STM	Mem32[value] = Rtarget	32 bit write
LDB	Rtarget = Mem8[value]	8 bit read , clears upper 24 bits of Rtarget
STB	Mem8[value] = Rtarget	8 bit write
INP	Rtarget = Input[value]	32 bit hardware input
OUT	Output[value] = Rtarget	32 bit hardware output
ADD	Cy.Rtarget = Rsource + value	Add
ADC	Cy,Rtarget = Rsource + value + cy	Add with carry in.
SUB	Cy.Rtarget = Rsource + ~value + 1	Subtract (inverse borrow out as 6502, no in)
MUL	Rtarget = Rsource * value	32 bit multiply
AND	Rtarget = Rsource & value	Bitwise And
ORR	Rtarget = Rsource | value	Bitwise Logical Or
XOR	Rtarget = Rsource ^ value	Bitwise Exclusive Or
ROR	Rtarget = Rsource >> value	Rotate source right value times, this value is “and 15” as rotating right 16 times takes you back where you started
BRR	Rtarget = PC, PC = PC + value	Used for all and branch ; saves current PC and does a *relative* branch.

Notes

- Move is technically superfluous as it could be done with add rtarget,rsource,#0 but because of the simplicity and the usefulness of the carry remaining unchanged it has remained.
- Multiplying two 32 bit numbers gives a 64 bit product, at present this is lost.
- There is no overflow detection, which probably isn’t an issue, but the reserved condition 100 could be execute on overflow.
- BRR can use PC as Rtarget. In this case, the value written is indeterminate, and as such the SZ flags are indeterminate.
- The PC is a post increment fetch, hence PC values used in the instructions are four or eight (depending on the mode) more than the main instruction. Rsource calculations using the PC should be done carefully.

Programming Notes

The BRR instruction works like the link register in ARM, and the subroutine mechanism in a CP1600 processor. However, all branches save the address branched from in the link register (which can be any register other than R31) or the branch junk register (which can be any other register). As before pseudo operations jmp jsr and rts exist to make code more readable.

As with the ARM/CP1600 if you want to nest subroutines you have to push the link register on the stack, and pop it off.

Syntax is similar to before. There are three forms (discounting pseudo operations), with one, two or three specified registers. The constant is optional and defaults to zero.

<opcode>[<condition>] Rtarget,Rsource,Rvalue[,#constant]
<opcode>[<condition>] Rtarget,Rvalue[,#constant]
<opcode>[<condition>] Rtarget[,#constant]

In the second form, Rsource and Rtarget are the same, so ADD R4,R5,#7 is actually R4 ← R4 + (R5+7)

In the third form Rsource and Rtarget are the same and Rvalue is the fixed zero register so ADD R4,#7 is actually R4 ← R4 + (Rzero+7). This is the reason for the zero register.

Note that with long constants e.g. ADD R4,#1024 only the third format is permissible (as Rvalue in this case will be R31, the program counter) ; this will do R4 ← R4 + 1024

By convention:

- R30 is the junk register
- R29 is always zero (this has to be cleared at start up !)
- R28 is the link register (known as LINK, a predefined constant)
- R27 is the stack pointer (known as SP, a predefined constant)