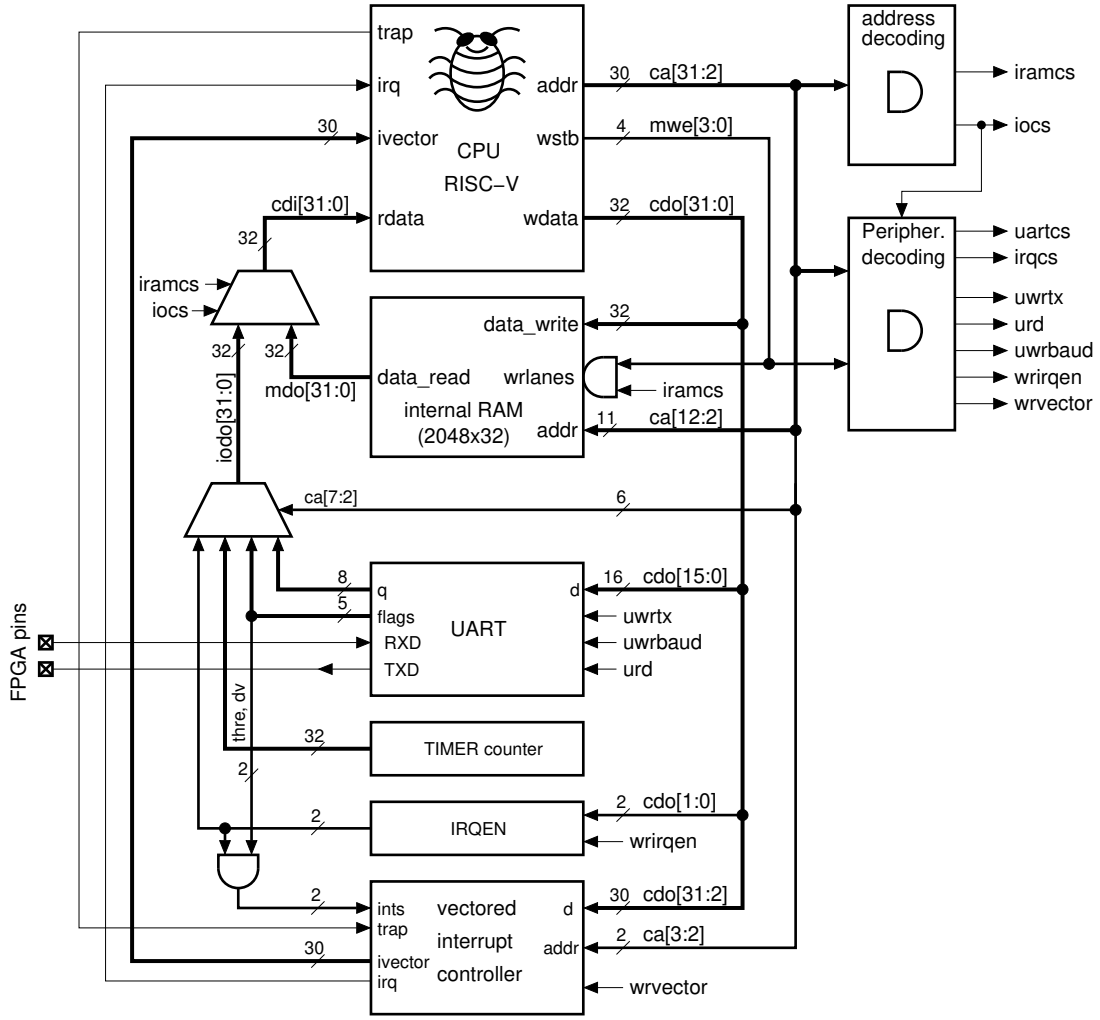
LaRVa's Small System

Jesús Arias

1 System description

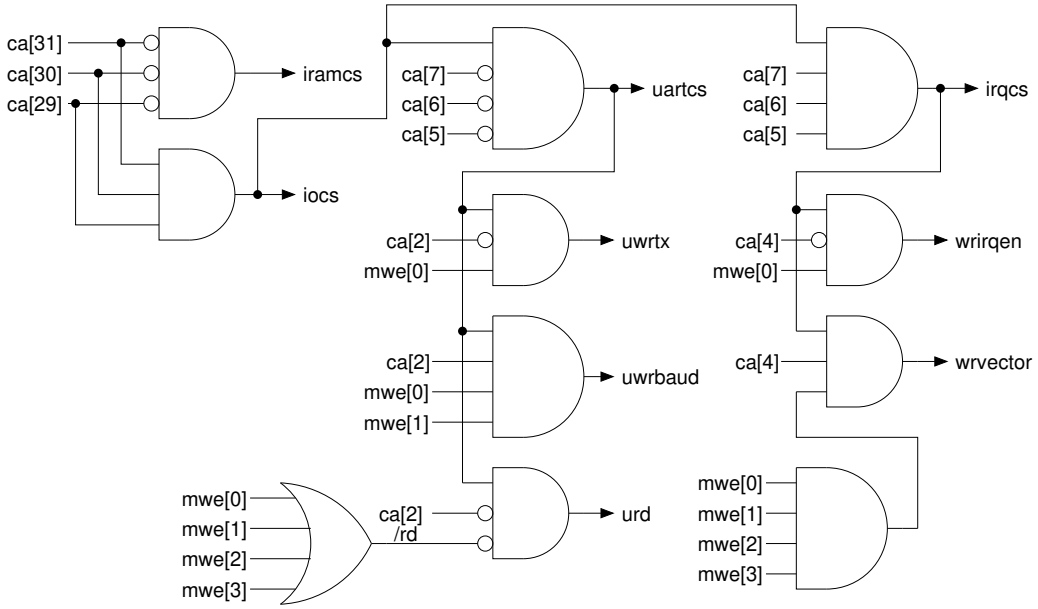


The above figure shows the block diagram of the system, that basically includes a laRVa CPU, 8KB of internal RAM with a known initial content, and an small number of peripherals:

- An Universal Asynchronous Receiver/Transmitter, UART, that provides a bidirectional communication channel with the outside World.
- A free running timer counter for time measurement.
- An interrupt enable register with one enable bit for each interrupt source: Bit 0: UART RX, Bit 1: UART TX.
- A Vectored Interrupt Controller with up to 4 different programmable interrupt vectors. These vectors are presented to the CPU when some interrupt is requested (including the execution of ECALL and EBREAK opcodes)

The RAM and the peripheral registers are mapped to the address space by means of the address decoding blocks (mainly for writings) and multiplexers (for reads).

1.1 Address decoding and memory map



A more detailed schematic of the address decoding logic is shown in the above figure, where "ca[]" are the address lines coming from the CPU and "mwe[]" are the memory write enable lanes for bytes. As data buses are 32 bits wide but the address is for bytes, the two lowest address bits are always 00 and they aren't included in the address bus. The four write lane enable signals are provided instead because we have to select which bytes to write during Store-Byte and Store-Halfword instructions. With this logic the memory map is:

strobe	address bits	R/W	Base address (x=A=0)
iramcs	`000x.xxxx.xxxx.xxxx.xxxA.AAAA.AAAA.AA00`	x	0x00000000
iocs	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xx00`	x	0xE0000000
uartcs	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.xx00`	x	0xE0000000
uwrtx	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x000`	mwe[0]	0xE0000000
urd	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x000`	rd	0xE0000000
uwrbaud	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x100`	mwe[1:0]	0xE0000004
irqcs	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.111x.xx00`	x	0xE00000E0
wrirqen	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.1110.xx00`	mwe[0]	0xE00000E0
wrvector	`111x.xxxx.xxxx.xxxx.xxxx.xxxx.1111.AA00`	mwe[3:0]	0xE00000F0

The above table is mainly for writes, where a write strobe signal is generated for each register to be written. The exception is "urd", a read strobe for the UART that is used to clear some flags (received data valid, for instance). Also notice that the UART baud rate register has to be written using Store-Word or Store-Halfword instructions because it holds more than 8 bits, and the interrupt vectors only allows the Store-Word instruction (32 bit write).

On the other hand, the mapping for reads is performed in the multiplexers for the CPU data input bus, resulting in the following table:

address bits	data read	Base address	Data width
000x.xxxx.xxxx.xxxx.xxxA.AAAA.AAAA.AA00	internal RAM	0x00000000	D[31:0]
111x.xxxx.xxxx.xxxx.xxxx.xxxx.xxxx.xx00	peripherals	0xE0000000	D[31:0]
111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x000	UART RX (Q)	0xE0000000	D[7:0]
111x.xxxx.xxxx.xxxx.xxxx.xxxx.000x.x100	UART flags	0xE0000004	D[4:0]
111x.xxxx.xxxx.xxxx.xxxx.xxxx.011x.xx00	Timer counter	0xE0000060	D[31:0]
111x.xxxx.xxxx.xxxx.xxxx.xxxx.111x.xx00	IRQEN	0xE00000E0	D[1:0]

Notice that, apart from the timer which requires a Load-Word, all the peripherals can also be read with Load-Byte or Load-Halfword instructions.

2 Peripherals

The peripheral address space is located at the upper 512MB area (0xE0000000 to 0xFFFFFFFF) . The included peripherals are:

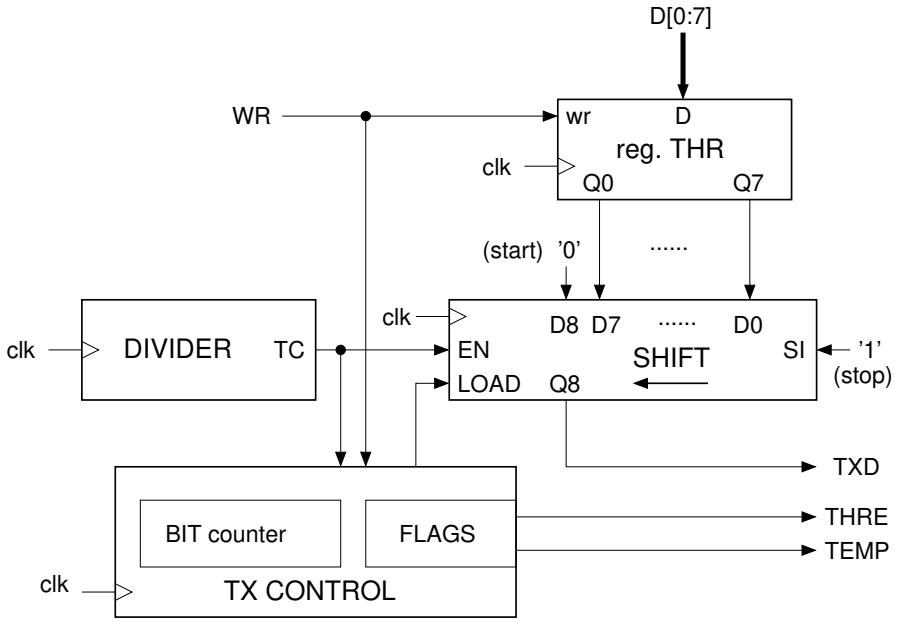
- A simple UART with fixed data format (8-bit, no parity) but now with a programmable baud rate. The UART can request two different interrupts: one for data received and other when ready to transmit.
- A free-running timer.
- A vectored interrupt controller.

2.1 UART

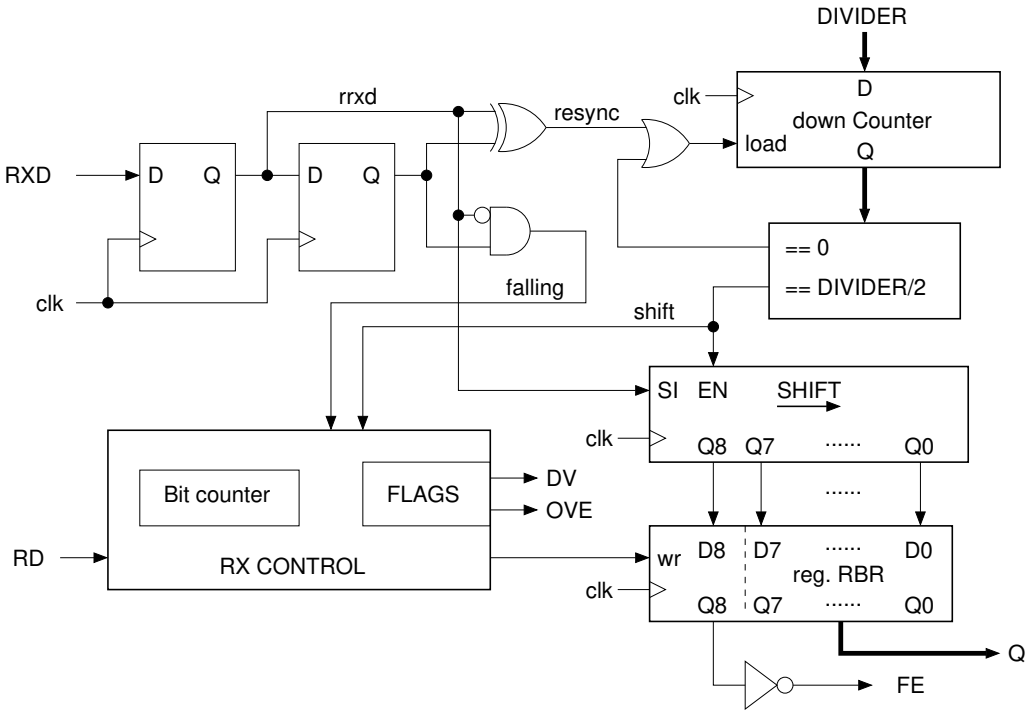
The UART used here is a custom design with the following features / limitations:

- Fixed data format: 8-bits, No Parity, and 1 Stop bit.
- Programmable baud rate with no prescaler. Baud rates as fast as $f_{CLK}/6$ are possible.
- No marketing shit like “sample three times and majority decision” is included. But on the other hand the receiver is able to resynchronize its clock on every data transition.
- Single character buffer registers for TX and RX.

A simplified diagram of the transmitter is shown next. It includes a clock divider, a 9-bit shift register, a buffer register (THR), and a finite-state machine controller mainly built around a bit counter. It provides two flags: THRE is set to one when THR can be written, and TEMP that is set to one when both THR and the shift registers are empty. THRE can request an interrupt when set.



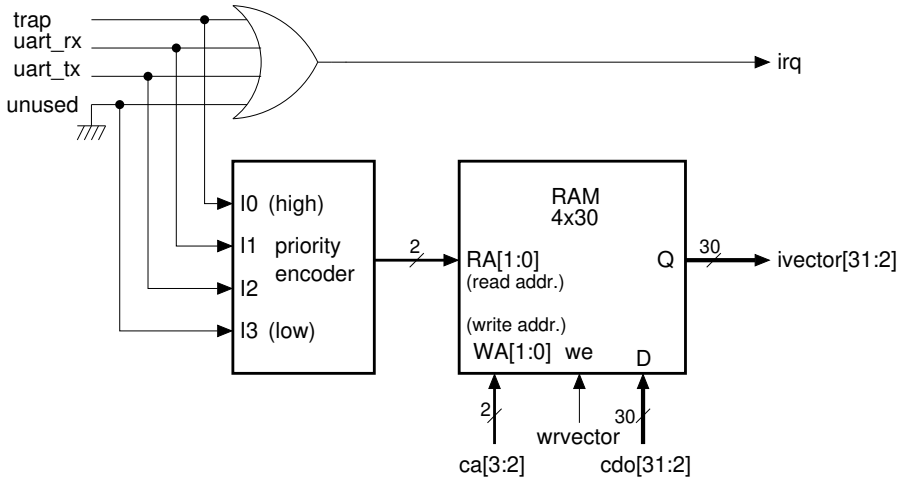
The diagram of the receiver is also shown, more explicitly detailing the detection of the start bit (falling pulse during idle state in the FSM) and clock recovery: the "resync" pulse on data transitions resets the clock divider, thus moving the "shift" pulse to the center of the bit time. The receiver also includes a 9-bit shift register and buffer (RBR). The nineth bit is used for framing error detection, FE. The FSM controller provides two outputs flags: a Data Valid that is set to one after a character reception and cleared when reading RBR, and an Overrun error that is set when a character is received with Data Valid already in one. OVE is also cleared reading RBR. DV can request an interrupt when set.



2.2 Timer

The timer is simply a 32 bit counter that gets incremented each clock cycle. Its value can be read at address 0xE0000060. This is a read-only peripheral.

3 Vectored Interrupts



The diagram of the simple Vectored Interrupt Controller is shown in the above figure. The system has 3 possible interrupt sources, each of them with an specific interrupt vector (write-only) where the memory address of the corresponding interrupt service routine is stored. There is also an interrupt enable register that allows the selective masking of interrupts:

Interrupt Cause	INTEN bit mask (0=masked)	Vector #	Priority	Vector address
ECALL, EBREAK	non-maskable	0	Highest	0xE00000F0
UART RX	bit #0	1		0xE00000F4
UART TX	bit #1	2	Lowest	0xE00000F8

The software interrupts (instructions ECALL and EBREAK) can't be masked. These are always serviced.