A very simple 8-bit RISC processor for FPGA

S. de Pablo, J.A. Cebrián, L.C. Herrero
University of Valladolid
E.T.S.I.I., Paseo del Cauce, s/n
47011 Valladolid (Spain)
sanpab@eis.uva.es

A.B. Rey
Polytechnic University of Cartagena
Ant. Cuartel de Antiguones (C. de la Muralla)
30202 Cartagena, Murcia (Spain)
alexis.rey@upct.es

Abstract

This article presents the "RISCuva1" processor, a very simple 8-bit RISC processor for FPGA. Its most important feature is that this processor is very simple. Its Verilog code has about 120 sentences, and most of them are easy to understand. It would be a good starting point for students who need to know how processors work and for those engineers who wish to design their own processor. The proposed processor has been physically tested on Xilinx SpartanIIe FPGAs with a performance of 40 MIPS in -6C grade devices.

1. Introduction

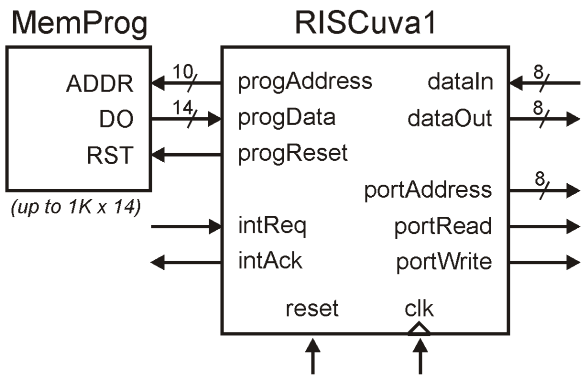
Over the past 40 years, it seems that Moore's Law [9] has been true: from 50 transistors per chip we have reached to more than 50 million transistors, doubling every two years. We may think, why not, that in the near future the number of *processors* per chip would also duplicate every two years. Most personal computers have nowadays two processor cores on a chip. Several student works at the University of Valladolid (Spain) are also in that trend: first we synthesize on a Xilinx FPGA a third party processor [2], then we run three DSP processors to control a photovoltaic system [3][11] and now we have implemented six small general-purpose processors for an audio-RF application [1].

This paper presents a very simple 8-bit general purpose processor for FPGA called "RISCuva1". Its small Verilog code demonstrates that it is very easy to design a simple core with processing capabilities. The processor size, performance and external interface are similar to Xilinx *picoBlaze*, created by Ken Chapman [4], but this one can execute bigger programs. Its internal architecture has been inspired in Jan Gray's GR0000 [5]. More FPGA processors can be found at [6][8][10].

2. Main features of the RISCuva1

The RISCuva1 is an 8-bit general-purpose RISC processor with Harvard architecture: it gets instructions on a regular basis using dedicated buses to its program memory, executes all its native instructions in just one clock cycle and exchanges data with several devices using other buses. It allows one source of interrupts.

Figure 1. Processor external connections.



This processor can execute programs with up to 1024 instructions (the equivalent *picoBlaze* is limited to a quarter of it) and it exchanges data through 256 ports, all of them with indirect access and 32 of them –from 224 to 255– also with direct access. The later are more intended for specific devices, such as an external stack for data o several seven-segment displays. In addition, it has special instructions to access external ports in sequential mode, a great feature for compilers who continuously access to multi-byte data.

Internally it has sixteen 8-bit general purpose registers that can be used in all operations. It includes twelve native operations (arithmetic, logic and bit rotation), but it also execute others as *"macros"* (see tables 2 and 3).

To make decisions it uses only two flags, 'Zero' and 'Carry', that are enough for most situations. With them it can evaluate up to four conditions. It also has unconditional jumps ('goto'), jumps to subroutines ('call') and return from them ('ret' and 'reti'). Its internal stack for returning addresses allows up to 16 nested subroutines[1]. This stack also keeps the value of flags during interrupt execution.

This processor has been synthesized and tested at a system frequency of 40 MHz (40 MIPS) on SpartanIIe-6C devices. Its frequency can be extended up to 85 MHz when it is implemented standalone[2].

3. The instruction set

The instruction set of the RISCuva1 has been designed following several rules:

- All instructions are executed in just one clock cycle. Doing so, processor is simpler, smaller, faster and easier to understand.
- The instruction code is received at the beginning of each cycle, all operations are executed during the clock period, and results are stored at the end of it.
- ALU operations take two operands from registers and store the result in one of them.
- No hardware nor binary codes have been used for several instructions ('inc', 'nop', …) that can be executed using other instructions (an 'add' may replace an 'inc', an so on).
- External read and write operations are synchronous.

The result, as shown in table 1, is very simple. It consists of 29 different instructions, but it can execute a lot of macros with them.

Table 1. RISCuva1 instruction set.

Opcode	Mnemonic	Operation
`00 00dd dddd dddd`	call <dest>	Call to subroutine
`00 01dd dddd dddd`	goto <dest>	Unconditional jump
`00 1000 •••• ••••`	ret	Subroutine return
`00 1001 •••• ••••`	reti	Interrupt return
`00 1010 •••• ••••`	di	Disable interrupts
`00 1011 •••• ••••`	ei	Enable interrupts
`00 1100 mmmm nnnn`	mov (rM),rN	Write to indirect port
`00 1101 •••• nnnn`	mov (++),rN	Write to next port
`00 111k kkkk nnnn`	mov (<k>),rN	Write to direct port
`01 00dd dddd dddd`	jpZ <dest>	Jump if Z == 1
`01 01dd dddd dddd`	jpNZ <dest>	Jump if Z == 0
`01 10dd dddd dddd`	jpC <dest>	Jump if C == 1
`01 11dd dddd dddd`	jpNC <dest>	Jump if C == 0
`10 kkkk kkkk nnnn`	mov rN,<k>	rN = <k>
`11 0000 mmmm nnnn`	mov rN,rM	rN = rM
`11 0001 mmmm nnnn`	xnor rN,rM	rN = rN xnor rM
`11 0010 mmmm nnnn`	or rN,rM	rN = rN or rM
`11 0011 mmmm nnnn`	and rN,rM	rN = rN and rM
`11 0100 mmmm nnnn`	add rN,rM	rN = rN + rM
`11 0101 mmmm nnnn`	adc rN,rM	rN = rN + rM + C
`11 0110 mmmm nnnn`	sub rN,rM	rN = rN – rM
`11 0111 mmmm nnnn`	sbc rN,rM	rN = rN – rM – C
`11 1000 •••• nnnn`	asr rN	Arithmetic shift
`11 1001 •••• nnnn`	rrc rN	Rotate through carry
`11 1010 •••• nnnn`	ror rN	Rotate right
`11 1011 •••• nnnn`	rol rN	Rotate left
`11 1100 mmmm nnnn`	mov rN,(rM)	Read on indirect port
`11 1101 •••• nnnn`	mov rN,(++)	Read on next port
`11 111k kkkk nnnn`	mov rN,(<k>)	Read on direct port

Flag 'Z' is affected by all '1x xxxx' operations (those who change a register) and flag 'C' changes only when '10 00xx', '10 01xx' or '10 10xx' instructions (ALU operations) are used.

All jumps use only 10 bits to address their destination, so programs are limited to 1024 instructions. This length would be more than enough for most applications, such as USB controllers [12] and others. Anyway, it is four times more than classic *picoBlaze* [4], and it can be extended in the same way.

External data access is quite good: its indirect addressing allows the processor to manage variables and lists on data memory and its direct addressing provides easy access to peripherals. Additionally, a sequential access improves multi-byte data manipulation, a nice feature for compilers[3].

Anyway, this instruction set seems to be very poor: it has no direct increment nor decrement instructions, a 'nop' instruction cannot be found, and it would be nice to find others like 'push', 'pop', etc. To solve all these deficiencies, a basic macro set has been included in the assembler program, as shown in table 2: it replaces several instruction-like constructions by one or more processor native instructions who work in the same way, including the effect on flags. But if programmers want to use them, they must follow several conventions: *registers 'r0' and 'r1' must be locked at '0x00' and '0x01' from the beginning of all programs, and register 'r2' must not be used* (mainly for advanced macro set, see table 3).

[1] It is very easy to increment the stack to 32 levels or even more.

[2] The processor core can be compiled at 85 MHz on -6 devices, but its performance decreases to 40 MHz when it is connected to a BlockRAM program memory and other peripherals.

[3] When compilers manage multi-byte variables ('int', for example) they need one indexed access –included here as a macro– to the first byte, and then other accesses to sequential addresses.

Table 2. RISCuva1 basic macro set.

Macro	Equivalent	Operation
nop	mov r0,r0	Do nothing (keep Carry)
inc rN	add rN,r1	rN = rN + 1 (r1 = 1!)
dec rN	sub rN,r1	rN = rN – 1 (r1 = 1!)
not rN	xnor rN,r0	Bit inversion (r0 = 0!)
neg rN	not rN; inc rN	rN = –rN
xor rN,rM	xnor rN,rM; not rN	rN = rN xor rM
cmp rN,rM	mov r2,rN; sub r2,rM	Compare two registers
setC	and r0,r0	Set the Carry flag
clrC	or r0,r0	Clear the Carry flag
rlc rN	adc rN,rN	Rotate left through carry
sl0 rN	add rN,rN	Shift left adding ‘zero’
sl1 rN	setC; rlc rN	Shift left adding ‘one’
sr0 rN	clrC; rrc rN	Shift right adding ‘zero’
sr1 rN	setC; rrc rN	Shift right adding ‘one’
push rN	mov (255),rN	Send value to data stack
pop rN	mov rN,(255)	Get from data stack
break	*Not documented*	Simulator break-point
halt	goto <PC>	Stop the processor

The main advantage of *not* including these instructions in the processor hardware is that the result is simpler, so smaller and faster. The instruction decoder will implement logic functions with less inputs, an important feature when an FPGA is the main target of this design.

Table 3. RISCuva1 advanced macro set.

Macro	Equivalent	Operation
and rN,<k>	mov r2,<k>; and rN,r2	AND with constant
or rN,<k>	mov r2,<k>; or rN,r2	OR with constant
xor rN,<k>	mov r2,<k>; xor rN,r2	XOR with constant
add rN,<k>	mov r2,<k>; add rN,r2	ADD with constant
adc rN,<k>	mov r2,<k>; adc rN,r2	ADC with constant
sub rN,<k>	mov r2,<k>; sub rN,r2	SUB with constant
sbc rN,<k>	mov r2,<k>; sbc rN,r2	SBC with constant
clr rN	mov rN,0	Clear a register
clr rN":rN':rN	clr rN; clr rN'; clr rN"	Clear multi-byte value
not rN":rN':rN	not rN; not rN'; not rN"	Invert multi-byte value
inc rN":rN':rN	add rN,r1; adc rN',r0; adc rN",r0	Increment multi-byte value
dec rN":rN':rN	sub rN,r1; sbc rN',r0; sbc rN",r0	Decrement multi-byte value
mov (rM),(rN)	mov r2,(rN); mov (rM),r2	Move between ports
mov rN,(<k> + rM)	mov r2,<k>; add r2,rM; mov rN,(r2)	Indexed read access
mov rN,(<k> – rM)	mov r2,<k>; sub r2,rM; mov rN,(r2)	Indexed read access
mov (<k> + rM),rN	mov r2,<k>; add r2,rM; mov (r2),rN	Indexed write access
mov (<k> – rM),rN	mov r2,<k>; sub r2,rM; mov (r2),rN	Indexed write access

These advanced macros, and others not shown in these tables, are included to ease programmers work.

4. Verilog source code

The internal architecture of this processor is RISC like. It executes all of its native instructions regularly, in just one clock cycle. Soon after the rising edge of the clock signal, a 14-bit instruction code is received from the program memory; these bits are decoded in order to execute all duties involved by the instruction; when the following rising edge arrives, all processor parts are prepared to update to newer values, and the address of the following instruction *has been sent* to the program memory to receive a new instruction code.

The Verilog code of this processor begins, as usual, with a declaration of its ports. It uses a ‘clk’ signal for synchronization and an active high ‘reset’. This processor is connected to a private program memory through three signals: it sends the address of the *following* instruction using ‘progAddress’; on the rising edge of the next clock cycle it receives the 14-bit code of the current instruction on ‘progData’. This processor also sends a ‘progReset’ signal to clear the ‘progData’ value received from the program memory in order to reset or interrupt the processor[4].

```
module RISCuva1 ( clk, reset,
                  progAddress, progData, progReset,
                  dataIn, dataOut,
                  portAddress, portRead, portWrite,
                  intReq, intAck );

  // Inputs and outputs:
  input          clk, reset;       // Clock and Reset

  output  [9:0]  progAddress;      // Up to 1K instructions (10 bits)
  input   [13:0] progData;         // Current instruction code
  output         progReset;        // Reset of Program Memory

  input   [7:0]  dataIn;           // Data input (from an I/O port)
  output  [7:0]  dataOut;          // Data output (through a port)

  output  [7:0]  portAddress;      // Addressed I/O Port (0..255)
  output         portRead;         // Read signal
  output         portWrite;        // Write signal
```

4 This technique was first used by Jan Gray [5]: a RST signal clears the data output of program memory when Xilinx BlockRAM are used, and then processor executes a ‘call 0’ (0x0000) instruction.

```
input       intReq;      // Interrupt request
output      intAck;      // Interrupt Acknowledge
```

Now we must decode the instruction code we receive from program memory: all instructions are executed in one clock cycle and each bit or group of bits has a meaning for it.

```
// Instruction decoding from the instruction code:
wire [13:0]  opCode = progData;     // Instruction code

wire   [1:0]  opA  = opCode[13:12];  // 1st operation code
wire   [1:0]  opB  = opCode[11:10];  // 2nd operation code
wire   [1:0]  opC  = opCode[  9:  8];  // 3rd operation code
wire   [3:0]  rM    = opCode[  7:  4];  // Source register
wire   [3:0]  rN    = opCode[  3:  0];  // Destination register

wire   [9:0]  immAddr = opCode[  9:0];  // Address for jumps
wire   [7:0]  immData = opCode[11:4];  // Immediate data
wire   [4:0]  immPort  = opCode[  8:4];  // For direct access

wire     MISC        =  (opA == 2'b00);
wire     JP          =  (opA == 2'b01);
wire     LOAD        =  (opA == 2'b10);
wire     ALU         =  (opA == 2'b11);

wire     CALL        =  (opB == 2'b00);
wire     GOTO        =  (opB == 2'b01);
wire     RETS        =  (opB == 2'b10);
wire     MOVOUT      =  (opB == 2'b11);

wire     RET         =  (opC == 2'b00);
wire     RETI        =  (opC == 2'b01);
wire     DI          =  (opC == 2'b10);
wire     EI          =  (opC == 2'b11);

wire     FLAG_Z      =  (opB == 2'b00);
wire     FLAG_NZ     =  (opB == 2'b01);
wire     FLAG_C      =  (opB == 2'b10);
wire     FLAG_NC     =  (opB == 2'b11);

wire     LOGIC       =  (opB == 2'b00);
wire     ARITH       =  (opB == 2'b01);
wire     SHIFT       =  (opB == 2'b10);
wire     MOVIN       =  (opB == 2'b11);

wire     MOV         =  (opC == 2'b00);
wire     XNOR        =  (opC == 2'b01);
wire     OR          =  (opC == 2'b10);
wire     AND         =  (opC == 2'b11);

wire     ADD         =  (opC == 2'b00);
wire     ADC         =  (opC == 2'b01);
wire     SUB         =  (opC == 2'b10);
wire     SBC         =  (opC == 2'b11);

wire     ASR         =  (opC == 2'b00);
wire     RRC         =  (opC == 2'b01);
wire     ROR         =  (opC == 2'b10);
wire     ROL         =  (opC == 2'b11);

wire     IND         =  (opC == 2'b00);
wire     SEQ         =  (opC == 2'b01);
wire     DIR         =  (opC >= 2'b10);
```

After these definitions, at least several general resources must be introduced: two DFF used by flags, an 8-bit bus used to collect the results of all operations, and the 12-bit output of the internal stack used to store returning addresses of subroutines and flags during interrupts. They will be referred before their implementation.

```
// General Resources:
reg               zeroFlag, carryFlag;      // DFFs used by flags
wire      [7:0]  dataBus;            // Data bus for all operations
wire  [2+9:0]  stackValue;         // Internal stack output
```

Now we can begin with the design of several units that compose the processor. The first one is the register file, a dual-port memory [7] used to store the 8-bit values of ‘r0’ to ‘r15’ registers. It allows two asynchronous reads at the beginning of each clock cycle and one synchronous write at the end of it.

```
// Register file (r0-r15) and operand buses:
reg  [7:0]  registerFile[0:15];       // 16x8 dual-port memory
always@(posedge clk)
begin
     if (LOAD | ALU)
          registerFile[rN] <= dataBus;    // Synchronous write
end
wire [7:0]  busN = registerFile[rN];       // Async. read of rN
wire [7:0]  busM = registerFile[rM];       // Async. read of rM
```

The data interface is very easy because all accesses are synchronous and they are executed in just one clock cycle. The address signal ‘portAddress’ chooses between the direct port given by the instruction code, the indirect value given by a register, or the last address used incremented by one. Read and write signals are simply decoded from the instruction code and the output data always comes from a register.

```
// Port signals for direct, indirect and sequential accesses:
reg [7:0]      nextPort;
always@(posedge clk)
begin
     if (portRead | portWrite)
          nextPort <= portAddress + 1;   // For sequential use
end
assign dataOut       = busN;                   // Output from rN
assign portRead      = ALU  &  MOVIN;      // Read signal
assign portWrite     = MISC & MOVOUT;   // Write signal
assign portAddress = IND  ? busM :                // Indirect
                              SEQ ? nextPort :              // Sequent.
                                    {3'b111,immPort};  // Direct
```

The ALU for logic operations computes all its functions and then selects the needed result. The carry of this unit has a special meaning: it will be ‘1’ for any ‘and’ operation and ‘0’ for any ‘or’ one, so ‘setC’ and ‘clrC’ functions (that set and clear the carry flag) are implemented as macros with no additional cost. Carry flag will be kept constant on all register movements to allow several macros. The whole unit can be synthesized using only eight LUT4 and one LUT3.

```
// Logic ALU: AND, OR, XNOR and MOV.
wire        logicCarry = AND ? 1'b1 : OR ? 1'b0 : carryFlag;
wire  [7:0] logicALU   =  AND   ? busN  & busM :
                          OR    ? busN  |  busM :
                          XNOR ? busN ~^ busM :
                                         busM ;
```

The full adder/subtracter for arithmetic operations implements its four operations using a single chain of LUTs.

```
// Arithmetic ALU: ADD, ADC, SUB and SBC.
wire  [7:0]  arithALU, altM;
wire         arithCarry, x, y, z;
assign  x = ADD ? 1'b0 : ADC ? carryFlag :
             SUB ? 1'b1 : ~carryFlag;
assign  altM                = (SUB | SBC) ? ~busM : busM;
assign  {z, arithALU, y}  = {busN, 1'b1} + {altM, x};
assign  arithCarry         = (SUB | SBC) ? ~z : z;
```

The shifter ALU is very similar to the logic one. Only ‘asr’ and ‘rrc’ operations are required, because other shifts and rotations can be replaced through *macros*, but ‘ror’ and ‘rol’ bit rotations are also included.

```
// Shifter: ASR, RRC, ROR and ROL.
wire [7:0]  shiftALU;
wire        shiftCarry;
assign  {shiftALU, shiftCarry} =
                     ASR ? {busN[7],    busN} :
                     RRC ? {carryFlag, busN} :
                     ROR ? {busN[0],    busN} :
                           {busN[6:0], busN[7], busN[7]};
```

Finally all possible results are collected in a tristate bus. It consumes no additional resources and its delay is meaningless in comparison with a full multiplexer.

```
// This data bus collects results from all sources:
assign dataBus = (LOAD | MISC)          ? immData   : 8'bz;
assign dataBus = (ALU | JP) & LOGIC  ? logicALU  : 8'bz;
assign dataBus = (ALU | JP) & ARITH   ? arithALU  : 8'bz;
assign dataBus = (ALU | JP) & SHIFT   ? shiftALU  : 8'bz;
assign dataBus = (ALU | JP) & MOVIN  ? dataIn      : 8'bz;
```

The control part of this processor may begin with the interrupt controller, who has three DFFs: ‘userEI’ allows user to enable or disable interrupts; ‘intAck’ is an output than acknowledges the interrupt; and ‘callingIRQ’ is used, with ‘validIRQ’, to coordinate the processor response when an IRQ is attended. Several instructions, those excluded by ‘mayIRQ’, are preserved from being interrupted: ‘di’, ‘ei’ and ‘reti’ for a clean work and all external accesses to allow sequential mode and, if wanted, wait states (not included in this design).

```
// Interrupt Controller:
reg        userEI, callingIRQ, intAck;
wire       mayIRQ = ! (MISC & RETS
                     | MISC & MOVOUT
                     | ALU  &  MOVIN);
wire       validIRQ = intReq & ~intAck & userEI & mayIRQ;
wire [9:0]  destIRQ = callingIRQ ? 10'h001 : 10'h000;
always@(posedge clk or posedge reset)
begin
    if (reset)                        userEI <= 0;
    else if (MISC & RETS & DI)       userEI <= 0;
    else if (MISC & RETS & EI)       userEI <= 1;

    if (reset)                        intAck <= 0;
    else if (validIRQ)                intAck <= 1;
    else if (MISC & RETS & RETI)  intAck <= 0;

    if (reset)                        callingIRQ <= 0;
    else                              callingIRQ <= validIRQ;
end
```

Following we describe two DFFs to store flags (‘Z’ and ‘C’), that are updated only when needed. This is an important feature that allows lots of macros and extends flag use possibilities.

```
// Flag DFFs:
always@(posedge clk)
begin
    if (MISC & RETS & RETI)   // Flags recovery when ‘reti’
        {carryFlag,zeroFlag} <= stackValue[11:10];
    else begin
        if (LOAD | ALU)          // 'Z' changes with registers
            zeroFlag  <= (dataBus == 8'h00);
        if (ALU & ~MOVIN)       // but 'C' only with ALU ops
            carryFlag <= LOGIC ? logicCarry :
                           SHIFT ? shiftCarry  :
                                   arithCarry ;
    end
end
// 'validFlag' evaluates one of four conditions for jumps.
wire validFlag = FLAG_Z   ?   zeroFlag  :
                 FLAG_NZ ? ~zeroFlag  :
                 FLAG_C   ?   carryFlag :
                              ~carryFlag ;
```

The “Program Counter” of this processor, in order to make it simpler, has only three functions: it can load a new immediate address on jumps, load a returning address when subroutines end, or increment itself otherwise. When this processor is connected to a synchronous reading program memory, like Xilinx BlockRAM, the ‘progAddress’ signal must be connected to ‘nextPC’ rather than ‘PC’, because of the registered nature of the program memory data output. Additionally, the synchronous ‘RST’ signal must be controlled to get a ‘call 0x0000’ instruction (codified as 0x0000) at reset time and a ‘call 0x0001’ when interrupts are acknowledged. Thanks to Jan Gray [5] for this idea.

```
// Program Counter (PC): the address of current instruction.
reg  [9:0]  PC;
wire [9:0] nextPC, incrPC;
wire     onRet    = MISC & RETS & (RETN | RETI);
wire     onJump = MISC & (GOTO | CALL) | JP & validFlag;
assign  incrPC = PC + (callingIRQ ? 0 : 1);
assign  nextPC = onRet    ? stackValue[9:0]      : 10'bz;
assign  nextPC = onJump ? immAddr | destIRQ : 10'bz;
assign  nextPC = !(onRet | onJump) ? incrPC     : 10'bz;
always@(posedge clk)
begin
    PC <= nextPC;
end
// When using Xilinx BlockRAM as program memory:
assign  progAddress = nextPC;
assign  progReset    = reset | validIRQ;
```

To implement the last feature of this processor, an internal stack for returning addresses and flags, we use a single-port distributed 16x12 memory and a pointer.

```
// Internal stack for returning addresses (16 levels):
reg  [3:0]  SP;                        // Stack Pointer register
always@(posedge clk or posedge reset)
begin
    if (reset)                                 SP <= 0;
    else if (MISC & CALL)                      SP <= SP + 1;
    else if (MISC & RETS & (RETN|RETI))  SP <= SP – 1;
end
wire [3:0]  mySP = (CALL | GOTO) ? SP : SP – 1;
reg  [2+9:0]  stackMem[0:15];         // Stack 16x12 memory
always@(posedge clk)
begin
    if (MISC & CALL)    // Keep returning address and flags
        stackMem [mySP] <= {carryFlag, zeroFlag, incrPC};
end
assign stackValue = stackMem[mySP];
```

At last we reach the end of this small module, with about 120 Verilog sentences. It uses 148 LUT4[5] and 70 TBUF when compiled for speed.

```
endmodule     /// RISCuva1 (all in one file!)
```

[5] The implementation result on a XC2S300E was 84 slices, less than 3% of the device. *PicoBlaze* uses 154 logic cells as can be seen on Xilinx Press Release #0270.

5. Programming example

An integrated development environment (IDE, see figure 2) with a ‘C’-like assembler[6], a simulator and emulator has been developed for this processor. The RISCuva1 capabilities can be observed in the following programming example.

```
/*
   Demonstration program of RISCuva1 for FPGAworld'2006
   2006/05/27    Santiago de Pablo (sanpab@eis.uva.es)
*/
#include     “uva1std.h”  // Several common definitions
#device  LIFO16    255  // An external data stack at port 255
#define  DISPLAY  224  // Four 7-seg displays at 224 and 225

PROGRAM:                    // Defined to begin at 0x0000
        goto Main           // Jump to main program
IRQ:                        // Defined to begin at 0x0001
        push  r2            // Keep r2 at IRQ when using macros
        inc  r15:r14        // Increments the 16-bit counter
        mov (DISPLAY),r14; mov (++),r15   // Displays its value
        pop  r2             // Recovery of r2
        reti                // Exit the interrupt recovering ‘flags’
Main:
        mov r0,0; mov r1,1  // User must lock them to 0 and 1
        clr    r15:r14        // Resets a counter used on IRQ
        ei                    // Enable interrupts
        mov  r3, Source       // Address the original text
        mov  r4, Destiny      // Address the space for the copy
StrCpy:                       // Copy a string from (r3) to (r4)
        mov  (r4),(r3)        // Copy one char of the string
        jpZ    Continue       // Ends the string copy with ‘\0’
        inc r3;  inc r4       // Update both pointers
        goto  StrCpy          // Repeat until end of string
Continue:
        nop                   // Yes, there are ‘nop’ instructions
        break                 // ... and also breakpoints.
        /*  Anything more to do?  */
        halt                  // Stops the processor at the end.
        // This state implies a “low power mode” where ...
        // ... processor do nothing except interrupt attention.
PORTS:                        // Defined to allow port initialization
Source:     text  “RISCuva1 rules!\0”// 32x8 RAM from port 0
Destiny:    space 16                    // Reserved for a copy
PORT224:  null  2             // Ports 224 and 255 are connected
                              // ... to four 7-seg displays.
```

This program begins with three declarations after a multi-line comment: it includes several common definitions from a file, instantiates a 16-level LIFO stack for the simulator at port 255 and defines the ‘DISPLAY’ string to address four seven-segment displays that are

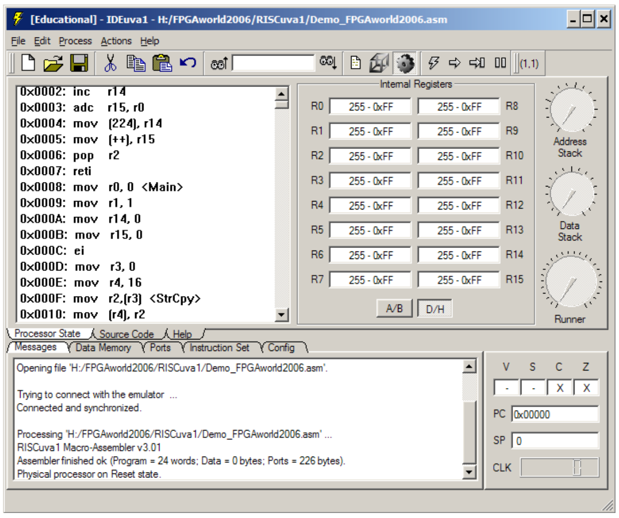
[6] A ‘C’ compiler for fixed point values is under work.

attached to ports 224 and 225. The external data stack allows 'push' and 'pop' macroinstructions.

User programs begin at 0x0000 address. The 'PROGRAM' string, defined at "uva1std.h" file as '0x0000', is used as an *absolute label* to prepare the assembler for programs. Other similar strings are 'IRQ' (assigned to 0x0001), 'PORTS' (assigned to 0x4000, the assembler knows what to do), 'PORT224' (assigned to 0x40E0), etc.

This small demonstration program has a main program (beginning at the *relative label* 'Main:') who copies a string located on data memory using ports to access it. A small interrupt subroutine has also been included (at 'IRQ:') which increments and displays a 16-bit counter. Multi-byte macros and one sequential access are used in this program to show the processor-plus-assembler low level possibilities.

Figure 2. An image of the IDE of this processor.



```
0x0002:  inc   r14
0x0003:  adc   r15, r0
0x0004:  mov   [224], r14
0x0005:  mov   [++], r15
0x0006:  pop   r2
0x0007:  reti
0x0008:  mov   r0, 0  <Main>
0x0009:  mov   r1, 1
0x000A:  mov   r14, 0
0x000B:  mov   r15, 0
0x000C:  ei
0x000D:  mov   r3, 0
0x000E:  mov   r4, 16
0x000F:  mov   r2,[r3]  <StrCpy>
0x0010:  mov   [r4], r2
```

R0	255 - 0xFF	255 - 0xFF	R8
R1	255 - 0xFF	255 - 0xFF	R9
R2	255 - 0xFF	255 - 0xFF	R10
R3	255 - 0xFF	255 - 0xFF	R11
R4	255 - 0xFF	255 - 0xFF	R12
R5	255 - 0xFF	255 - 0xFF	R13
R6	255 - 0xFF	255 - 0xFF	R14
R7	255 - 0xFF	255 - 0xFF	R15

```
Opening file 'H:/FPGAworld2006/RISCuva1/Demo_FPGAworld2006.asm'.

Trying to connect with the emulator  ...
Connected and synchronized.

Processing 'H:/FPGAworld2006/RISCuva1/Demo_FPGAworld2006.asm' ...
RISCuva1 Macro-Assembler v3.01
Assembler finished ok (Program = 24 words; Data = 0 bytes; Ports = 226 bytes).
Physical processor on Reset state.
```

V	S	C	Z
-	-	X	X

PC 0x00000
SP 0
CLK

6. Conclusions

This article has presented a small and easy to understand processor developed using Verilog for FPGA. It is very similar in size and features to *picoBlaze*, but it improves its possibilities. It would be a good starting point for students who need to know how processors work and for those engineers who wish to design their own processor.

It executes all the instructions in one clock cycle, including jumps, returns from subroutines and external accesses. A sequential access to ports has also been included for compilers who make an intensive use of multi-byte data. The assembler of this processor is full of macros that extend the native instruction set to facilitate low level programming.

7. Acknowledgments

The authors would like to acknowledge the financial support of the *Junta de Castilla y Leon* under grants VA004B06 and VA021B06. Our thanks also to those students of our University who work hard and contribute to develop this processor and others, their Integrated Development Environments and their peripherals.

References

[1] R. Aceves, *Desarrollo de un enlace inalámbrico para telefonía fija empleando una FPGA*. Final Project at the ETSII, University of Valladolid, Spain, 2006.

[2] M. Alonso, *Diseño de un Entorno de Desarrollo de Alto y Bajo Nivel para un Procesador de Propósito General integrado en FPGA*, Final Project at the ETSII, University of Valladolid, Spain, 2003.

[3] J. del Barrio, *Desarrollo sobre FPGA de un Emulador de una Planta de Microgeneración Eléctrica*, Final Project at the ETSII, University of Valladolid, Spain, 2004.

[4] K. Chapman, "PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices", Xilinx XAPP213 (v2.0), online at *http://www.xilinx.com/xapp/xapp213 .pdf*, December, 2002.

[5] J. Gray, "Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip", *DesignCon'2001*, online at *http://www.fpgacpu.org/gr/index.html*, 2001.

[6] J. Gray, "FPGA CPU Links", on line at *http://www. fpgacpu.org/links.html*, September, 2002.

[7] S. K. Knapp, "XC4000 Series Edge-Triggered and Dual-Port RAM Capability", Xilinx XAPP065, 1996.

[8] J. Kent, "John's FPGA Page", online at *http://members. optushome.com.au/jekent/FPGA.htm*, January, 2002.

[9] G. Moore, "Cramming more components onto integrated circuits", *Electronics Magazine*, 19 April, 1965.

[10] Opencores: *http://www.opencores.org/*.

[11] S. de Pablo et al., "A soft fixed-point Digital Signal Processor applied in Power Electronics", *FPGAworld Conference 2005*, Stockholm, Sweden, 2005.

[12] I. Rodríguez, *Desarrollo en FPGA de un interfaz USB con un ordenador personal*, Final Project at the ETSII, University of Valladolid, Spain, 2005.