An Update of the RISC5 Implementation

Niklaus Wirth, 15.6.2018

1. Introduction

This note describes an update of the implementation of the RISC5 processor, described in Verilog and implemented on an FPGA.: https://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/RISC.pdf

The core of the new concept is that memory is treated, as is customary today, as an external device. This implies that circuitry concerned with memory access is moved from the processor module RISC5 to the environment module RISC5Top, where also all other connections to peripheral devices reside. Thereby the ROM containing the startup code (boot loader) also resides in the Top module.

A facility for interrupts has been added. This required the addition of the parameter *irq* to the interface of RISC5. The top module supplies this signal with a tick, resulting in an interrupt every millisecond. Apart from this, RISC5 uses the same auxiliary modules as the old RISC5. The instruction set remains unchanged, and therefore *no changes in the software* are necessary.

Several further details have been cleaned up or simplified. An example is byte selection for byte-wise access to memory.

2. Memory access

Memory access for code and data are clearly separated in the interface of RISC5. Input of program code is handled by *codebus*, input and output of data by *inbus,* and *outbus*. The signals *rd* and *wr* indicate reading and writing of data, and *ben* determines byte selection.

```
module RISC6(
  input clk, rst, irq, stallX,
  input [31:0] inbus, codebus,
  output [21:0] adr,
  output rd, wr,
  output [3:0] ben,
  output [31:0] outbus);
```

In a von Neumann architecture, program code and data are stored in the same memory. Therefore, a memory access requires two clock cycles, and the processor must be stalled for one cycle. A stall effectively keeps the state of the processor unchanged (PC, IR, registers). In the first cycle the address is computed and the data are fetched or stored. In the second cycle, the next instruction is fetched.

```
PC <= stall ? PC : pcmux;
IR <= stall ? IR : codebus;
```

Apart from the parameter irq there are no changes in RISC5Top. RISC5 is instantiated by the statement

```
RISC5 riscx(.clk(clk), .rst(rst), .irq(limit),
  .rd(rd), .wr(wr), .ben(ben), .stallX(stallX),
  .adr(adr), .codebus(codebus), .inbus(inbus), .outbus(outbus));
```

The ROM for holding the initial program (typically the boot loader) has been moved from module *RISC5* to *RISC5Top*. Writing to the SRAM occurs in the second cycle of a memory instruction.

```
PROM PM( .adr(adr[10:2]), .data(romout), .clk(~clk));
assign codebus = (adr[21:12] == 10'h3FF) ? romout : inbus;
```

```
assign SRadr = vidreq ? vidadr : adr[19:2];
```

3. Registers

In order to separate the handling of registers from other concerns, like the ALU,, a new module *Registers* has been introduced. It features 1 input and 3 output ports selected by respective register numbers

```
Module Registers(
    input clk,wr,
    input [3:0] rno0, rno1, rno2,
    input [31:0] din,
```

```
  output [31:0] dout0, dout1, dout2);

reg [31:0] R[15:0];
assign dout0 = R[rno0];
assign dout1 = R[rno1];
assign dout2 = R[rno2];
always @ (posedge clk) R[rno0] <= wr ? din : R[rno0];
endmodule
```

Module *Regsters* is instantiated by the statement

```
Registers regs (.clk(clk), .wr(regwr), .rno0(ira0), .rno1(irb), .rno2(irc),
    .din(regmux), .dout0(A), .dout1(B), .dout2(C0));
```

4. Interrupts

Additions due to the interrupt facility are all within the processor module RISC5. Two new instructions have been added, one for returning from an interruptr procedure, and one for enabling or disabling interrupts. Both are encoded as BR instructions.

New variables are

```
wire intAck;
reg irq1, intEnb, intPnd, intMd;
reg [25:0] SPC;  // saved PC on interrupt
```

When a rising edge of the interrupt signal *irq* is present and the processor is not yet handling an earlier interrupt, then *intPnd* is asserted. The interrupt is then pending until acknowledged by signal intAck Then the processor enters interrupt mode (intMd) and the processor jumps to location 4 (interrupt vector). The current PC and the condition bits are saved in register *SPC*. No register values are saved, because every interrupt handler is assumed to save (at least) registers R0 and R1. During interrupt mode, no further interrupt is accepted.

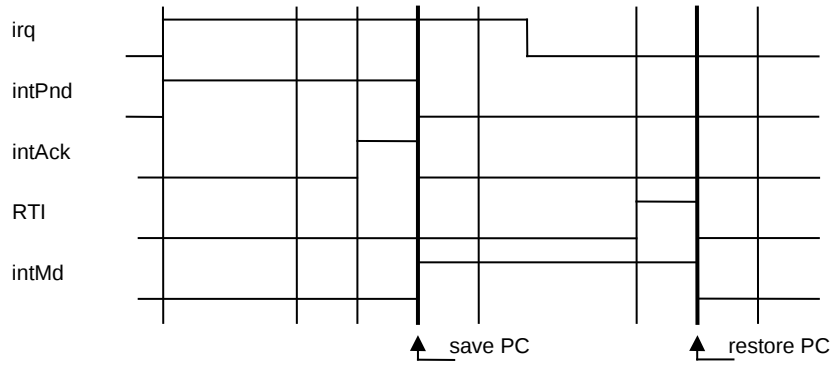
```
intPnd <= rst & ~intAck & ((~irq1 & irq) | intPnd);
assign intAck = intPnd & intEnb & ~intMd & ~stallr;
intMd <= rst & ~RTI & (intAck | intMd);
SPC <= intAck ? {nn, zz, cx, vv, pcmux0} : SPC;

assign pcmux0 = stall ? PC :
  RTI ? SPC[21:0] :
  (BR & cond) ? (u? nxpc + disp : C0[23:2]) : nxpc;
assign pcmux = ~rst ? StartAdr : intAck ? 1 : pcmux0;;
```

Interrupt handlers are supposed to end with an RTI instruction (return from interrupt) instead of a regular return instruction. An **RTI** instruction (BR with bit 4 set) causes the next instruction address to be taken back from SPC[21:0] and the condition bits from SPC[25:22]. Also, the processor leaves the interrupt mode.

```
RTI = p & q & ~u & IR[4];
```

The Oberon compiler recognizes an interrupt handler through an asterisk after the symbol PROCEDURE. Variable *intEnb* is set and reset by a register branch instruction with bit 5 set. It copies bit 0 to *intEnb*. The Oberon compiler accomplishes this through the **LDPSR** procedure, a branch register instruction with bit 5 set

```
intEnb <= ~rst ? 0 : (p & q & ~u & ~v & IR[5]) ? IR[0] : intEnb;.
```



The following is an example of an interrupt test module. Its handler causes the LEDs to toggle at intervals of 500 ms.

```
MODULE TestInt;
  IMPORT SYSTEM;
  VAR led, cnt: INTEGER;

  PROCEDURE* Int; (*interrupt handler called every millisecond*)
  BEGIN INC(cnt);
    IF cnt = 500 THEN led := 3 - led; LED(led); cnt := 0 END
  END Int;

PROCEDURE On*;
BEGIN SYSTEM.LDPSR(1)
END On;

PROCEDURE Off*;
BEGIN SYSTEM.LDPSR(0)
END Off;

BEGIN led := 1; cnt := 0; (*install Int at address 4*)
  SYSTEM.PUT(4, 0E7000000H + SYSTEM.ADR(Int) DIV 4 - 2)
END TestInt.
```