16 Implementation of the RISC processor

16.1. Introduction

The design of the processor to be described here in detail was guided by two intentions. The first was to present an architecture that is distinct in its regularity, minimal in the number of features, yet complete and realistic. It should be ideal to present and explain the main principles of processors. In particular, it should connect the subjects of architectural and compiler design, of hardware and software, which are so closely interconnected.

Clearly “real”, commercial processors are far more complex than the one presented here. We concentrate on the fundamental concepts rather than on their elaboration. We strive for a fair degree of completeness of facilities, but refrain from their “optimization”. In fact, the dominant part of the vast size and complexity of modern processors and software is due to speed-up called optimization. It is the main culprit in obfuscating the basic principles, making them hard, if not impossible to study. In this light, the choice of a RISC (Reduced Instruction Set Computer) is obvious.

The use of an FPGA provides a substantial amount of freedom for design. Yet, the hardware designer must be much more aware of availability of resources and of limitations than the software developer. Also, timing is a concern that usually does not occur in software, but pops up unavoidably in circuit design. Nowadays circuits are no longer described in terms of elaborate diagrams, but rather as a formal text. This lets circuit and program design appear quite similar. The circuit description language – we here use *Verilog* – appears almost the same as a programming language. But one must be aware that differences still exist, the main one being that in software we create mostly sequential processes, whereas in hardware everything “runs” concurrently. However, the presence of a language – a textual definition – is an enormous advantage over graphical schemata. Even more so are systems (tools) that compile such texts into circuits, taking over the arduous task of placing components and connecting them (routing). This holds in particular for FPGAs, where components and wires connecting them are limited, and routing is a very difficult and time-consuming matter.

The development of this RISC progressed through several stages. The first was the design of the architecture itself, (more or less) independent of subsequent implementation considerations. Then followed a first implementation called RISC-0. For this a *Harvard Architecture* was chosen, implying that two distinct memories are used for program and for data. For both chip-internal *block RAMs* were used. The Harvard architecture allows for a neat separation of the arithmetic from the control unit.

But these blocks of RAM are relatively small on the used Spartan-3 development board (1 - 4K words). This board, however, provides also an FPGA-external static RAM with a capacity of 1 MByte. In a second effort, the BRAM for data was replaced by this SRAM. Both instructions and data are placed into the SRAM, resulting in a *von Neumann* architecture.

The RISC hardware is characterized by three interfaces. The first is the programmer's interface, the architecture, that is, those aspects that are relevant to the programmer, in particular, the instruction set. It is described in Appendix A2. The second is the hardware interface between the processor core and its environment, described here. The third is that which connects the environment with physical devices such as memory, keyboard and display. This is described in Chapter 17.

```
module RISC5(
   input clk, rst, stallX,
   input [31:0] inbus, codebus,
   output [19:0] adr,                  // memory and device addresses
   output rd, wr, ben,                 // read, write,  byte enable control signals for memory
   output [31:0] outbus);
```

The main parts of the hardware interface are three busses, the data input and output busses, the code bus, and the address bus. Signals *rd* and *wr* indicate, whether a read or a write operation is to be performed. *ben* indicates a byte (rather than word) access. The entire processor operates synchronously on the clock *clk* (25 MHz on Spartan-3), *rst* is the reset signal (from a push button on the development board), and *stall* is the input to stall the processor.

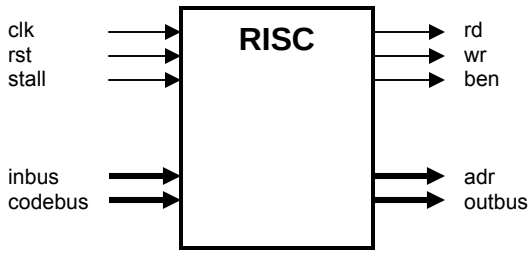

Figure 16.1 The processor's interface

First we concentrate on the implementation of the processor core, its realization in the form of circuits. They are divided into two parts, the arithmetic/logic unit processing data, and the control unit determining the flow of instructions.

16.2. The arithmetic and logic unit

The ALU features a bank of 16 registers with 32 bit *words*. Arithmetic and logical operations, represented by instructions, always operate on these registers. Data can be transferred between memory and registers by separate load and store instructions. This is an important characteristic of RISC architectures, developed between 1975 and 1985. It contrasts with the earlier CISC architectures (Complex instruction set): Memory is largely decoupled from the processor. A second important characteristic is that *most instructions take a single clock cycle* (25 MHz) for their execution. The exceptions are access to memory, multiplication and division.. More about this will be presented later. This single-cycle rule makes such processors *predictable* in performance. The number of cycles and the time required for executing any instruction sequence is precisely defined. Predictability is essential in all real-time applications.

The data processing unit consisting of ALU and registers is shown in Figure 16.2. Evidently, data cycle from registers through the ALU, where an operation is performed, and the result is deposited back into a register. The ALU embodies the circuits for arithmetic operations, logical operations, and shifts. The operations available are listed below. They are described in more detail in Appendix A2. The operand *n* is either a register or a part of the instruction itself.

0	MOV a, n	R.a := n	
1	LSL a, b, n	R.a := R.b ← n	(shift left by n bits)
2	ASR a, b, n	R.a := R.b → n	(shift right by n bits with sign extension)
3	ROR a, b, n	R.a := R.b rot n	(rotate right by n bits)
4	AND a, b, n	R.a := R.b & n	logical operations
5	ANN a, b, n	R.a := R.b & ~n	
6	IOR a, b, n	R.a := R.b or n	inclusive or
7	XOR a, b, n	R.a := R.b xor n	exclusive or
8	ADD a, b, n	R.a := R.b + n	integer arithmetic
9	SUB a, b, n	R.a := R.b – n	
10	MUL a, b, n	R.a := R.a x n	
11	DIV a, b, n	R.a := R.b div n	

```
12    FAD  a, b, c    R.a := R.b + R.c          floating-point arithmetic
13    FSB  a, b, c    R.a := R.b – R.c
14    FML  a, b, c    R.a := R.a x R.c
15    FDV  a, b, c    R.a := R.b / R.c
```

The following excerpt describes the essence of the ALU circuits. It is written in the HDL Verilog and refers to the following wires and registers.

```
wire [31:0] IR;
wire p, q, u, v, w;          // instruction fields IR[31], IR[30], IR[29], IR[28], IR[16]
wire [3:0] op, ira, irb, irc;  // instruction fields IR[19:16], IR[27:24], IR[23:20], IR[3:0]
wire [15:0] imm;             // instruction field IR[15:0]

wire [31:0] A, B, C0, C1, regmux;
wire [31:0]  s3, t3, quotient, fsum, fprod, fquot;
wire [32:0] aluRes;
wire [63:0] product;

reg [31:0] R [0:15];         // array of 16 registers
reg  N, Z, C, OV;            // condition flags
```

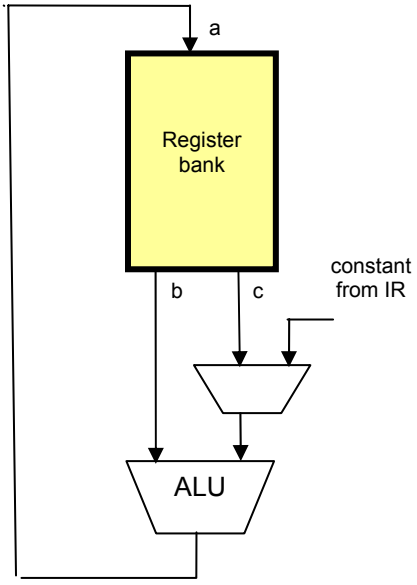

Fig. 16.2. Processor core with ALU and registers

B and C0 are the outputs from the register bank, and A is its input. The register numbers *ira* for port A, *irb* for port B, and *irc* for port C0 are taken from 4-bit fields of the instruction register IR. C1 is the multiplexer selecting among the register output C0 and the immediate field *imm*. *s3* and *t3* are outputs of the shift units (Sect. 16.2.1). *product* is the output of the multiplier (16.2.2), *quotient* and *remainder* those of the divider (16.2.3), *fsum* that of the floating-point adder (16.2.4), *fprod* that of the floating-point multiplier (16.2.5), and *fquot* the output of the floating-point divider (16.2.6).

```
assign A = R[ira];
assign B = R[irb];
assign C0 = R[irc];
assign C1 = q ? {{16{v}}, imm} : C0;
```

The following represents the main instruction decoding and selection of results. The opcodes refer to specific values of fields *p* and *op* of IR. Note that *if x then y else z* is denoted in Verilog by *x ? y : z.*

```
assign aluRes =
```

```
MOV ?     (q ? (~u ? {{16{v}}, imm} : {imm, 16'b0}) :
                (~u ? C0 : (~irc[0] ? H : {N, Z, C, OV, 20'b0, 8'b01010000}))) :
LSL ?           t3 :                    // output of left shift unit
(ASR|ROR) ?  s3 :                       // output of right shift unit
AND ?           B & C1 :
ANN ?           B & ~C1 :
IOR  ?          B | C1 :
XOR ?           B ^ C1 :
ADD ?           B + C1 + (u & C) :
SUB ?           B - C1 - (u & C) :
MUL ?           product [31:0] :        // output of multiplier
DIV ?           quotient :
(FAD|FSB) ?  fsum :
FML ?           fprod :
FDV ?           fquot : 0;
```

The input to the register bank, *regmux,* is selected from either *alures, inbus* (for LDR instructions), or the program address *nxpc* (for branch and link instructions). The signal *regwr* determines, whether data are to be stored (written) into the register bank. Details must be gathered from the respective program listing RISC.v.

```
always @ (posedge clk) begin
   R[ira] <= regwr ? regmux : A;
   N <= regwr ? regmux[31] :  N;
   Z <= regwr ? (regmux == 0) : Z;
    C <= (ADD|SUB) ? aluRes[32] : C;
   OV <= (ADD|SUB) ? aluRes[32] ^ aluRed[31] : OV ;
end
```

Whenever a register is written, the condition flags are also affected. They are N (*aluRes* negative), *Z* (*aluRes* zero), C (carry), and OV (overflow). The latter apply only to addition and subtraction.

16.2.1 Shifters

Shifters are multi-way multiplexers. For a 32-bit word, the simplest solution would be 32 32-way multiplexers. But this is hardly economical. On the FPGA used here, 4-way muxes are basic cells. It is therefore beneficial, to compose a shifter out of 4-way muxes. Now the obvious solution is to use 3 levels of muxes through which data flow. The first level shifts by amounts of 0, 1, 2, or 3, the second by amounts of 0, 4, 8, 12, and the third by 0 or 16. This scheme is programmed as follows for left shifts (instruction LSL) with B as input, *sc0* = C1[1:0] and *sc1* = C1[3:2] as shift counts, and *t3* as output:

```
assign t1 =   (sc0 == 3) ? {B[28:0], 3'b0} :
              (sc0 == 2) ? {B[29:0], 2'b0} :
              (sc0 == 1) ? {B[30:0], 1'b0} : B;
assign t2 =   (sc1 == 3) ? {t1[19:0], 12'b0} :
              (sc1 == 2) ? {t1[23:0], 8'b0} :
              (sc1 == 1) ? {t1[27:0], 4'b0} : t1;
assign t3 =   C1[4] ? {t2[15:0], 16'b0} : t2;
```

The solution for right shifts is analogous. An additional level of multiplexing is required, shifting in either the sign bit (ASR with sign propagation) or bits from the low end of the word (ROR), making a barrel shifter. This selection is controlled by the instruction bit w = IR[16].

16.2.2. Multiplication

Multiplication is an inherently more complex operation than addition and subtraction. After all, multiplication can be composed (of a sequence) of additions. There are many methods to implement multiplication, all – of course – based on the same concept of a series of additions. They show the fundamental problem of trade-off between time and space (circuitry). Some solutions operate with a minimum of circuitry, namely a single adder used for all 32 additions executed sequentially (in time). They obviously sacrifice speed. The other extreme is multiplication

in a single cycle, using 32 adders in series (in space). This solution is fast, but the amount of required circuitry is high..

Before we present the sequential solution, let us briefly recapitulate the basics of a multiplication *p := x × y*. Here *p* is the product, *x* the multiplier, and *y* the multiplicand. Let *x* and *y* be *unsigned* integers. Consider *x* in binary form.

$$x = x_{31}\times 2^{31} + x_{30}\times 2^{30} + \ldots + x_1\times 2^1 + x_0\times 2^0$$

Evidently, the product is the sum of 32 terms of the form $x_k\times 2^k\times y$, i.e. of *y* left shifted by *k* positions multiplied by x_k. Since x_k is either 0 or 1, the product is either 0 or *y* (shifted). Multiplication is thus performed by an adder and a selector. The selector is controlled by x_k, a bit of the multiplier. Instead of selecting this bit among $x_0 \ldots x_{31}$, we right shift x by one bit in each step. Then the selection is always according to x_0. The *add-shift step* then is

```
IF ODD(x) THEN p := p + y END ;
y := 2*y; x := x DIV 2
```

whereby multiplication by 2 is done by a left shift, and division by 2 by a right shift: As an example, consider the multiplication of two 4-bit integers x = 5 and y = 3, requiring 4 steps:

	p	x	y	
	0000'0000	0101	0000'0011	
add y to p	0000'0011	010**1**	0000'0011	
shift	0000'0011	0010	0000'0110	
add 0 to p	0000'0011	001**0**	0000'0110	
shift	0000'0011	0001	0000'1100	
add y to p	0000'1111	000**1**	0000'1100	
shift	0000'1111	0000	0001'1000	
add 0 to p	0000'1111	000**0**	0001'1000	
shift	0000'1111	0000	0011'0000	p = 15

The shifting of *x* to the right also suggests that instead of shifting *y* to the left in each step, we keep *y* in the same position and shift the partial sum *p* to the right. We notice that the size of *x* decreases by 1 in each step, whereas the size of *p* increases by 1. This allows to pack *p* and *x* into a single double register <B, A> with a shifting border line. At the end, it contains the product p = x × y.

	p	x	
	0000	0101	
add y to p	00**11**	010**1**	
shift	00011	010	
add 0 to p	00011	01**0**	
shift	000011	01	
add y to p	00**11**11	0**1**	
shift	0001111	0	
add 0 to p	0001111	**0**	
shift	00001111		p = 15

p = {B[31:0], A{31:[32-k]}, x = A[31-k:0] k = 0 … 31

The multiplier is controlled by a rudimentary state machine S, actually a simple 5-bit counter running from 0 to 31. The multiplier is shown schematically in Figure 16.3.

The multiplier interprets its operands as signed (*u* = 0) or unsigned (*u* = 1) integers. The difference between unsigned and signed representation is that in the former case the first term has a negative weight ($-x_{31}\times 2^{31}$). Therefore, implementation of signed multiplication requires very little change: Term 31 is subtracted instead of added (see complete program listing below).

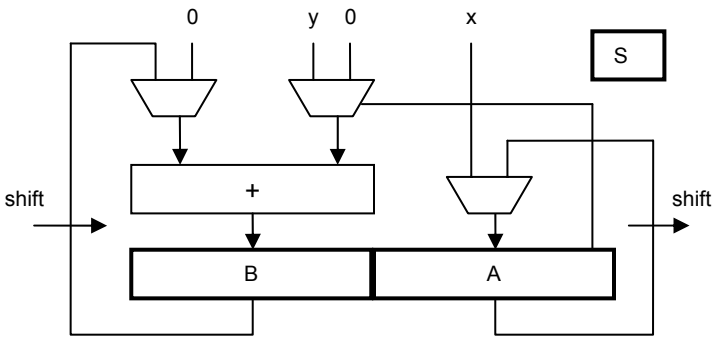

Figure 16.3. Schematic of multiplier

During execution of the 32 add-shift steps the processor must be stalled. The process proceeds and the counter S advances as long as the input MUL is active (high). MUL indicates that the current operation is a multiplication, and the signal is stable until the processor advances to the next instruction. This happens when step 31 is reached (Figure 16.4).

```
stall  =  MUL & ~(S == 31);
S  <=  MUL ? S+1 : 0;
```

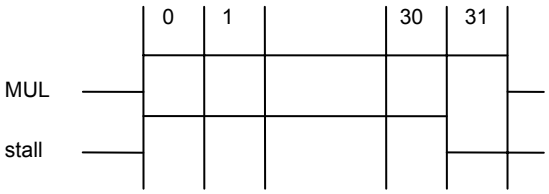

Figure 16.4. Generating *stall*

The details of the simple multiplier are listed below:

```
module Multiplier(
  input CLK, MUL, u,
  output stall,
  input [31:0] x, y,
  output [63:0] z);

reg [4:0] S;    // state
reg [31:0] B2, A2;  // high and low parts of partial product
wire [32:0] B0, B00, B01;
wire [31:0] B1, A0, A1;

assign stall = MUL & ~(S == 31);
assign B00 = (S == 0) ? 0 : {B2[31] & u, B2};
assign B01 = A0[0] ? {y[31] & u, y} : 0;
assign B0 = ((S == 31) & u) ? B00 - B01 : B00 + B01;
assign B1 = B0[32:1];
assign A0 = (S == 0) ? x : A2;
assign A1 = {B0[0], A0[31:1]};
assign z = {B1, A1};

always @ (posedge(CLK)) begin
  B2 <= B1; A2 <= A1;
  S <= MUL ? S+1 : 0;
end
endmodule
```

Implementing multiplication in hardware made the operation about 30 times faster than its solution by software. A significant factor! As multiplication is a relatively rare operation – at least in comparison with addition and subtraction – early RISC designs (MIPS, SPARC, ARM) refrained from its full implementation in hardware. Instead, an instruction called *multiply step* was provided, performing a single add-shift step in one clock cycle. A multiplication was then programmed by a sequence of 32 step instructions, typically provided as a subroutine. This measure of economy was abandoned, when hardware became faster and cheaper.

The FPGA used on the Spartan-3 board features a welcome facility for speeding up multiplication, namely fast 18 x 18 bit multiplier units. These are made available as basic cells of the FPGA, and they multiply in a single clock cycle. Considering an operand $x = x_1 \times 2^{16} + x_0$, the product is obtained as the sum of only 4 terms:

$$p = x \times y = x_1 \times y_1 \times 2^{32} + (x_0 \times y_1 + x_1 \times y_0) \times 2^{16} + x_0 \times y_0$$

Thereby multiplication of two 32-bit integers can be performed in 2 cycles only, one for multiplications, one for addition. Four multipliers are needed. For details, the reader is referred to the program listing (module *Multiplier1*).

16.2.3. Division

Division is similar to multiplication in structure, but slightly more complicated. We present its implementation by a sequence of 32 *shift-subtract steps*, the complement of add-shift. We here discuss division of *unsigned* integers only.

$$q = x \text{ DIV } y \qquad r = x \text{ MOD } y$$

q is the *quotient*, *r* the *remainder*. These are defined by the invariants

$$x = q \times y + r \qquad \text{with} \qquad 0 \le r < y$$

Both *q* and *r* are held in registers. Initially we set r to x, the dividend, and then subtract multiples of y (the divisor) from it, each time checking that the result is not negative. This *shift-subtract step* is

```
r := 2*r; q := 2*q;
IF r – y ≥ 0 THEN r := r – y END
```

As an example, consider the division of the 8-bit integer *x* = 14 by the 4-bit integer *y* = 4, where multiplication and division by 2 are done by shifts:

	r	q	y	
	0000'1110	0000	0001'1000	
shift	0000'1110	0000	0001'1000	r < y
sub 0 from r	0000'1110	0000	0000'1100	
shift	0000'1110	0000	0000'1100	r >= y
sub y from r	0000'0010	000**1**	0000'1100	
shift	0000'0010	0010	0000'0110	r < y
sub 0 from r	0000'0010	0010	0000'0110	
shift	0000'0010	0100	0000'0011	r < y
sub y from r	0000'0010	0100	0000'0011	q = 4, r = 2

As with multiplication this arrangement may be simplified by putting *r* and *q* into a double-length shift register, and by shifting *r* to the left instead of *y* to the right. This results in

	r	q	
	0000'1110		
shift	0001'110	0	r < Y
sub 0 from r	0001'110	0	
shift	0011'10	00	r >= Y
sub y from r	0000'10	0**1**	
shift	0001'0	010	r < Y
sub 0 from r	0001'0	010	

shift	0010	0100	r < Y
sub 0 from r	0010	0100	q = 4, r = 2

This scheme is represented by the circuit shown in Figure 16.5.

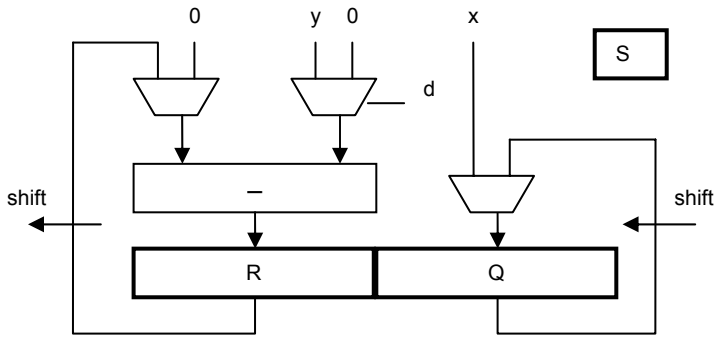

Figure 16.5. Schematic of divider

Stall generation is the same as for the multiplier. A division takes 32 clock cycles. Further details are shown in the subsequent program listing.

```
module Divider(
  input clk, DIV,
  output stall,
  input [31:0] x, y,
  output [31:0] quot, rem);

reg [4:0] S;  // state
reg [31:0] r3, q2;
wire [31:0] r0, r1, r2, q0, q1, d;

assign stall = DIV & ~(S == 31);
assign r0 = (S == 0) ? 0 : r3;
assign d = r1 - y;
assign r1 = {r0[30:0], q0[31]};
assign r2 = d[31] ? r1 : d;
assign q0 = (S == 0) ? x : q2;
assign q1 = {q0[30:0], ~d[31]};
assign rem = r2;
assign quot = q1;

always @ (posedge(clk)) begin
  r3 <= r2; q2 <= q1;
  S <= DIV ? S+1 : 0;
end
endmodule
```

16.3. Floating-point arithmetic

The RISC uses the IEEE Standard for representing REAL (floating-point) numbers with 32 bits. The word is divided into 3 fields: *s* for the sign, *e* for the exponent, and *m* for the mantissa. The value is

$$x = (-1)^s \times 2^{e-127} \times 1.m \quad \text{with} \quad 1.0 \le m < 2.0 \quad \text{(normalized form)}$$

Numbers are represented in sign-magnitude form. This implies that for sign inversion only the sign bit must be inverted, and exponent and mantissa remain unchanged.

Zero is a special case represented by 32 0-bits, and therefore has to be treated separately. Furthermore, e = 255 denotes "not a number". It is generated in the case of arithmetic overflow.

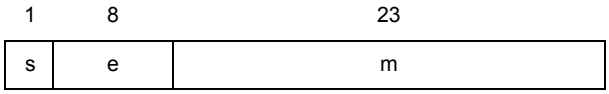

Figure 16.6 IEEE standard floating-point representation of REAL numbers

16.3.1. Floating-point addition

If two numbers are to be added, they must have the same exponent. This implies that the summand with the smaller exponent must be denormalized. *m* is shifted to the right and *e* is incremented accordingly. That is, if *d* is the difference of the two exponents, *m* is multiplied by 2^d, and *e* is incremented by d. After the addition, the sum must be rounded and post-normalized. *m* is shifted to the left and *e* is decremented accordingly. The shift amount is determined by the position of the leftmost one-bit. This results in the scheme shown in Figure 16.7, and the module's interface is

```
module FPAdder(
  input clk, run, u, v,
  input [31:0] x, y,
  output stall,
  output [31:0] z);
```

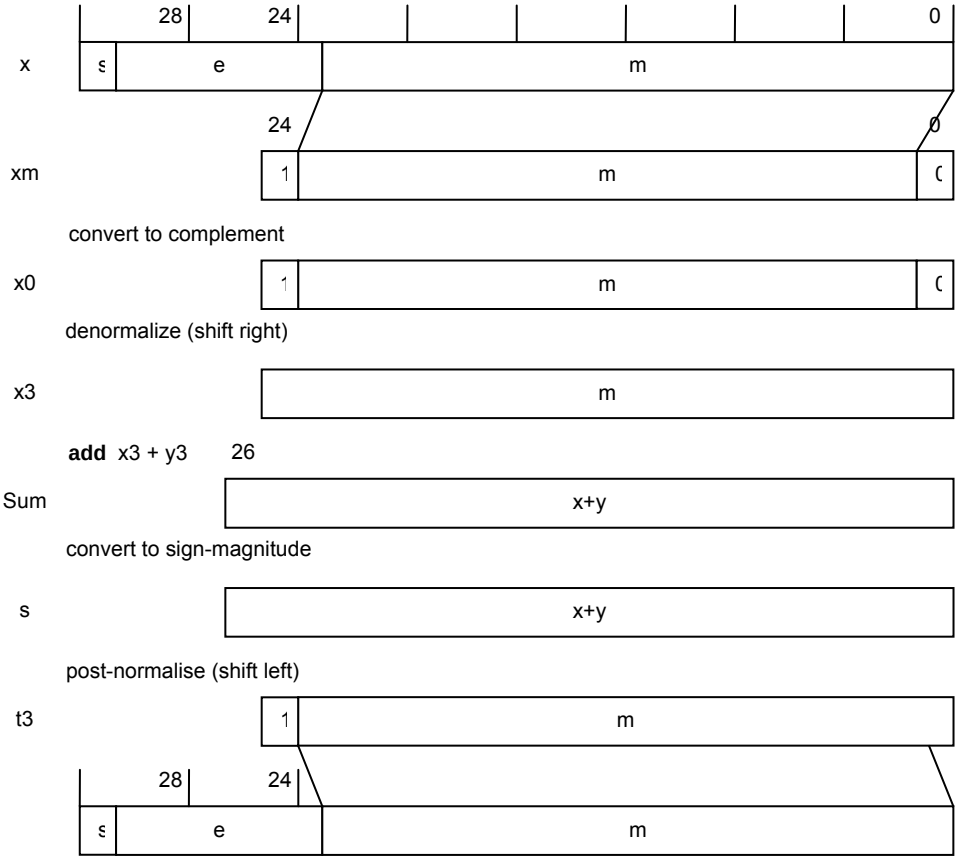

Figure 16.7 Steps of floating-point addition

It is important to achieve proper rounding. This is done by extending the mantissa of both operands by a guard bit, initialized to 0. A one is added (effectively 0.5) and at the end the guard bit is discarded.

The two predefined conversion functions FLT and FLOOR are conveniently implemented as additions. A denormalized 0 is added to the argument, effecting the proper shift. In the case of FLT (modifier bit u = 1), denormalization is omitted (no 1-bit inserted), and in the case of FLOOR (modifier bit v = 1), post-normalization is suppressed.

16.3.2. Floating-point multiplication

A product is given by the equation

$$p = x \times y = (2^{xe} \times xm) \times (2^{ye} \times ym) = 2^{xe+ye} \times (xm * ym)$$

$$p = (xs, xe, xm) \times (ys, ye, ym) = (xs \text{ xor } ys, xe + ye, xm \times ym)$$

That is, exponents are added, mantissas multiplied. Denormalization is not needed. Post-normalization is a right shift of at most one bit, because if $1.0 \leq xm, ym < 2.0$, the result satisfies $1.0 \leq xm*ym < 4.0$. The sign of the product is the exclusive or of the signs of the arguments. The multiplier module's interface is

```
module FPMultiplier(
  input clk, run,
  input [31:0] x, y,
  output stall,
  output [31:0] z);
```

16.3.3. Floating-point division

A quotient is given by the equation

$$q = x / y = (2^{xe} \times xm) / (2^{ye} \times ym) = 2^{xe-ye} \times (xm / ym)$$

$$q = (sx, ex, mx) / (sy, ey, my) = (sx \text{ xor } sy, ex - ey, mx / my)$$

That is, exponents are subtracted, mantissas divided. Denormalization is not needed. Post-normalization requires a left shift by at most a single bit, because if $1.0 \leq xm, ym < 2.0$, the result satisfies $0.5 \leq xm/ym < 2.0$. The sign of the product is the exclusive or of the signs of the arguments. The divider module's interfaces is

```
module FPDivider(
   input clk, run,
   input [31:0] x, y,
   output stall,
   output [31:0] z);
```

16.4. The Control Unit

The control unit determines the sequence of executed instructions. It contains two registers, the program counter PC holding the address of the current instruction, and the current instruction register IR holding the instruction currently being interpreted. Instructions are obtained from memory through the *codebus* (see interface), from where the decoding signals emanate. Mostly, the arithmetic unit and the control unit operate concurrently (in parallel). While the arithmetic unit performs the operation held in register IR and data signals flow through the ALU, the control unit fetches in the same clock cycle the next instruction from memory in the location with the address held in PC. Next address and next instruction are latched in the registers at the end of a cycle. This scheme constitutes a one-element pipeline of instructions.

The principal task of the control unit is to generate the address of the next instruction. There are essentially only four cases:

0. Zero on reset.

1. The next instructions address is PC+1 (all instructions except branches)
2. The branch target PC+1 + offset. (Branch instructions).
3. It is taken from a data register. (This is used for returning from procedures).

This is reflected by the following program text, and shown in Figure 16.8.

```
reg [17:0] PC;
reg [31:0] IRBuf;
wire [31:0] IR;
wire [31:0] pmout;
wire [17:0] pcmux, nxpc;
wire cond;

IR = codebus;
nxpc = PC + 1;
pcmux = (~rst) ? 0 :
        (stall) ? PC :  // stall
        (BR & cond & u) ? off + nxpc :
        (BR & cond & ~u) ? C0[19:2]  :
        nxpc;

always @ (posedge clk)  PC <= pcmux; end
```

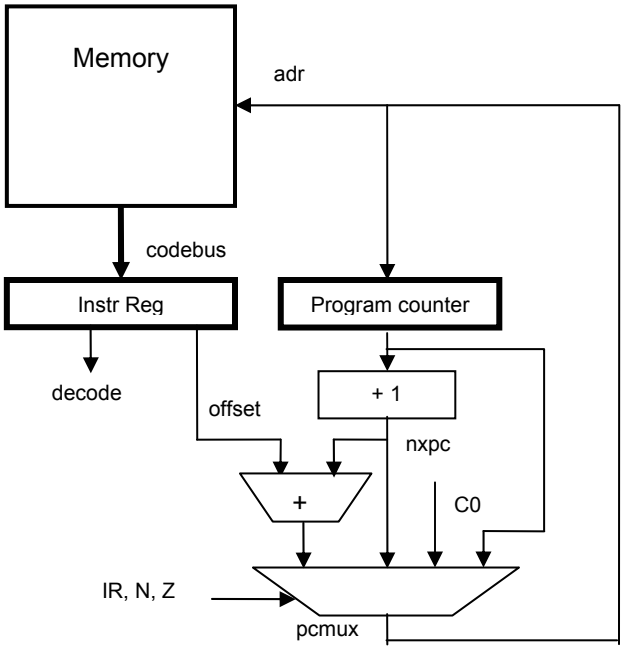

Fig. 16.8. The control unit

Branches are the only conditional instructions. Whether a branch is taken or not, is determined by the combination of the condition flags selected by the condition code field of the branch instruction. IR[27] is the condition sense inversion bit.

```
reg N, Z, C, OV;  // condition flags
wire S;
assign S = N ^ OV;
assign cond = IR[27] ^
    ((cc == 0) & N |          // MI, PL
    (cc == 1) & Z |           // EQ, NE
    (cc == 2) & C |           // CS, CC
    (cc == 3) & OV |          // VS, VC
    (cc == 4) & (C|Z) |       // LS, HI
    (cc == 5) & S |           // LT, GE
```

```
    (cc == 6) & (S|Z) |         // LE, GT
    (cc == 7));                 // T, F
```

There is, unfortunately, a complication obfuscating the simple scheme presented so far. It stems from the necessity to initialize the processor. Only registers and memory blocks (BRAM) can be initialized and loaded by the available FPGA-tools. How, then, is a program (in our case the boot loader) moved into memory, the chip-external SRAM? The following scheme has been chosen:

The initial program is loaded into a BRAM (1K x 32). This block is memory-mapped into high-end addresses in the range of the data stack. On startup, the flag *PMsel* is set and IR is loaded from *pmout* (from the BRAM) at *StartAdr*. At the end of the program (boot loader), a branch instruction with destination 0 jumps to the beginning of the program that had just been loaded into SRAM by the boot loader. This is, presumably, but not necessarily, the operating system. The following changes and additions are required:

```
localparam StartAdr = 18'b111111100000000000; // 0FE000H

reg PMsel;  // memory select for instruction fetch
reg [31:0] IRBuf;

dbram32 PM (  // BRAM
   .clka (clk),
   .rdb (pmout),  // output port
   .ab (pcmux[10:0]));  // address

assign IR = PMsel ? pmout : IRBuf;

always @ (posedge clk) begin
    PMsel <= ~rst | (pcmux[17:11] == 7'b1111111);
    IRBuf <= stall ? IRBuf : codebus;
    ...
end ;
```

17 The processor's environment

The RISC processor is embedded in an environment (module RISCTop.v) connecting it with elements that are FPGA-chip external, but whose are provided on the Spartan development board (Figure 17.1). The environment consists of an address decoder, a data multiplexer, and interfaces to the memory and peripheral devices..

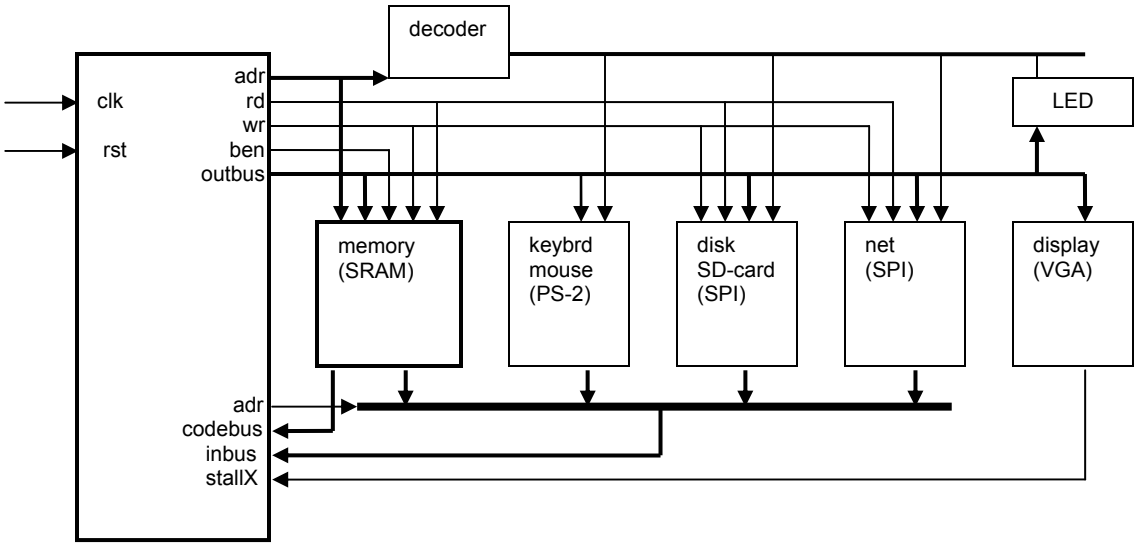

Figure 17.1 The RISC configuration

The decoder for output and the multiplexer for input determine the various addresses of devices:

adr		input	output
0	0FFFFFFC0H	millisecond counter	reserved
4	0FFFFFFC4H	switches	LEDs
8	0FFFFFFC8H	RS-232 data	RS-232 data
12	0FFFFFFCCH	RS-232 status	RS-232 control
16	0FFFFFFD0H	SPI data (SD-card, net)	SPI data (SD-card, nat)
20	0FFFFFFD4H	SPI status	SPI control
24	0FFFFFFD8H	PS/2 keyboard	
28	0FFFFFFDCH	mouse	

The circuitry connecting with the SRAM is part of this module, whereas the drivers for the other devices are described in separate modules. Note: The signals to and from devices must be listed in the heading of the top module, which is not imported by any other module. Their pin numbers are specified in a configuration file (.ucf). For details, the reader is referred to the program listing, as several items are rather dependent on the given Spartan-3 board.

17.1. The SRAM memory

The design of the circuitry around a static RAM is quite straight forward. The only controls are a read (SRoe) and a write enable signal (SRwe). Since the SRAM multiplexes data lines for input and output, a tri-state driver (SRbuf) must be used on the FPGA. This is shown schematically in Figure 17.2.

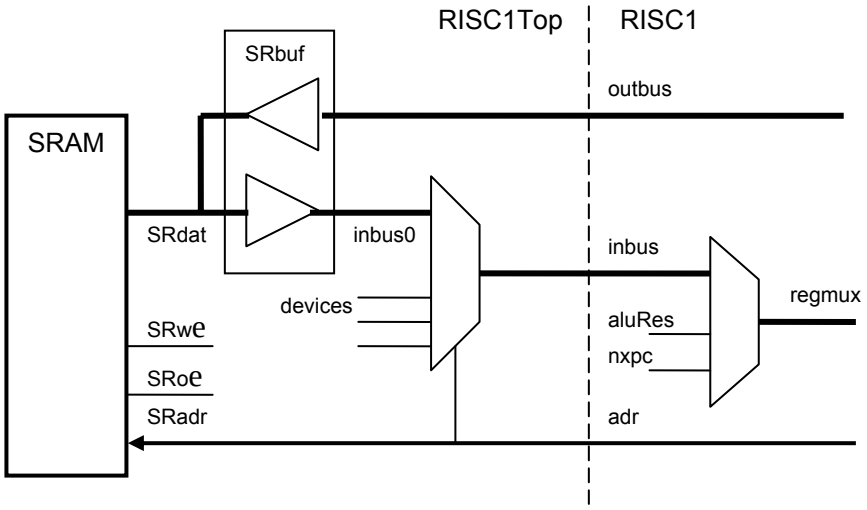

Figure 17.2. Connections between processor and SRAM

However, there is a complication: the feature of byte-wise access. After all, the present RAM is 32 bits wide (actually there are two 512K x 16-bit chips in parallel). Evidently, some multiplexing is unavoidable. The task is significantly eased by the chip's feature of four separate write enables, one for each byte of a word. The selection of the byte affected is determined by address bits 0 an 1 (which are ignored in the case of word-access). This scheme is shown in Figure 17.3. The codebus bypasses the multiplexers.

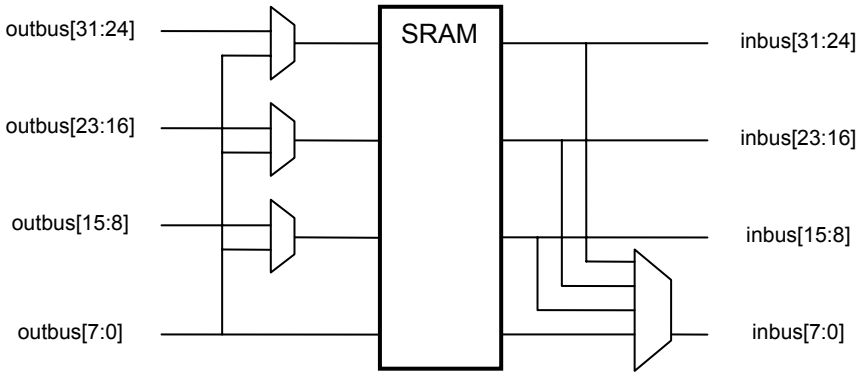

Figure 17.3 Multiplexers for SRAM byte access

17.2. Peripheral interfaces

Each of the interfaces to external media is implemented as a separate module and can therefore easily be exchanged. Modules are connected with the processor by the input and the output bus, and by enable signals *wr* and *rd*.

17.2.1. The PS/2 interface for the keyboard

PS/2 is mostly used for input devices. It uses 2 wires (apart from ground)., one for data, one for the clock. It uses a *synchronous* transmission, and the clock is driven by the device. Here it is used for the keyboard and the mouse (see Sect. 17.2.5). Transmission occurs in packets of 8 bits. An optional third wire serves for output. It is not used in this application. The interface is very simple and consists of an 8-bit buffer register. The following describes the interface for the keyboard.

A bit is shifted into the data register whenever the clock shows a falling edge, i.e. the clock signal *Q0* is low and the clock delayed by one cycle *Q1* is high.

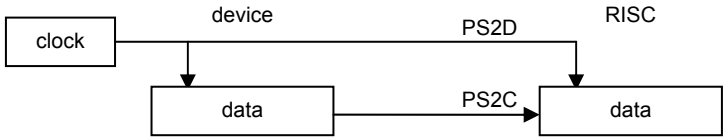

Figure 17.4 The PS/2 configuration

In the driver for the keyboard a 16-byte fifo buffer is inserted, forming a queue. This is necessary in order to avoid loss of characters when the processor is tied up in computation.

```
module PS2(
   input clk, rst,
   input done,   // "byte has been read"
   output rdy,   // "byte is available"
   output shift, // shift in, transmitter
   output [7:0] data,
   input PS2C,   // serial input
   input PS2D);   // clock

reg Q0, Q1;  // synchronizer and falling edge detector
reg [10:0] shreg;
reg [3:0] inptr, outptr;
reg [7:0] fifo [15:0];  // 16 byte buffer
wire endbit;

assign endbit = ~shreg[0];  //start bit reached correct pos
assign shift = Q1 & ~Q0;
assign data = fifo[outptr];
assign rdy = ~(inptr == outptr);

always @ (posedge clk) begin
  Q0 <= PS2C; Q1 <= Q0;
  shreg <= (~rst | endbit) ? 11'h7FF :
    shift ? {PS2D, shreg[10:1]} : shreg;
  outptr <= ~rst ? 0 : rdy & done ? outptr+1 : outptr;
  inptr <= ~rst ? 0 : endbit ? inptr+1 : inptr;
  if (endbit) fifo[inptr] <= shreg[8:1];
end
endmodule
```

17.2.2 The Mouse

Subsequently we present two Mouse interfaces. The first (MouseP) is based on the PS/2 Standard and caters for most commercially available mice. The second (MouseX) is included here for historical reasons. It was used by the computer Lilith in 1979, and used the same Mouse as its ancestor Alto (at PARC, 1975). It is distinguished by a very simple hardware without its own microprocessor, which is currently contained in most mice. This goes at a cost of a 9-wire cable. But today, microprocessors are cheaper than cables. We include this interface here, because it allows for a simple explanation of the principle of pointing devices.

The first interface uses the PS/2 Standard, that is, a 2-wire cable (not counting ground and power). It complies with the commercial standard of pointing devices. Details are shown on module MouseP.v.

```
module MouseP (input rst, clk,
   inout PS2C, PS2D,
   output [27:0] out);
endmodule
```

The second interface described here is not based on any standard, but it features the same interface to the software environment. Its principles are very simple and easily explained, and it refrains from the use of a mouse-internal processor. The price for this simplicity is a cable with 7 wires (plus 2 for power and ground), namely 3 for 3 buttons, and 2 for each direction, x (left/right) and y (up/down).

Let us first explain how signals indicating movements are derived. The key reason for the solution's simplicity is that these signals are directly mirrored by the position of a cursor on the display. The human user simply moves the Mouse until the cursor has reached the desired position (for example, at a displayed object). Thereby, the human eye and hand are included in the feedback loop providing the desired precision. This represents a very clever symbiosis between man and computer.

An actual movement is recognized by a simple light sensor (we will restrict our observation to a single coordinate x). The movement is transmitted to a wheel consisting of a transparent disc with intransparent spokes. A light beam shines through the disk and is received by the light sensor. Each time a spoke passes, the light is blocked. Any change in the sensor output signals a movement (see Figure 17.5). Unfortunately, this scheme does not allow to recognize the direction of the movement (left or right). A second light and sensor solve this problem. The distance between the two lights is half the distance of adjacent spokes.

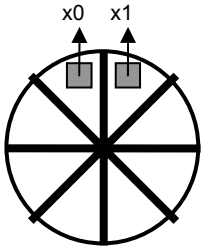

Figure 17.5 Wheel with spokes and sensors

The signal pair x0, x1 originating from a movement (with constant speed) to the left or to the right is shown in Figure 17.6.

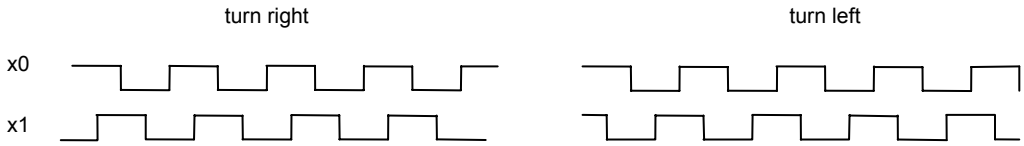

Figure 17.6 Signals resulting from movements

The logic equations for movements to the left and right (or up and down) are derived from this signal pair. For each signal a register records the state. Therefore it can be determined whether a move to the left, or to the right, or no move had occurred. The sampling frequency is irrelevant, as long as it is high enough. Let *x01* be *x00* delayed by one clock cycle, and *x11* be *x10* delayed by one cycle.

x00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
right	0	1	0	0	0	0	0	1	1	0	0	0	0	0	1	0
left	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0

Every active *right* signal causes the 10-bit *x* counter to be incremented, and every *left* to be decremented.

An identical circuit is used for the up/down direction, with its wheel set perpendicular to the first wheel. Finally, the output is packed into a single word. 3 bits are taken by the keys, and 10 by each of the two counters.

```
module MouseX(
  input clk,
  input [6:0] in,
  output [27:0] out);

  reg x00, x01, x10, x11, y00, y01, y10, y11;
  reg ML, MM, MR;  // keys
  reg [9:0] x, y;  // counters

  wire xup, xdn, yup, ydn;

  assign xup = ~x00&~x01&~x10&x11 | ~x00&x01&x10&x11 | x00&~x01&~x10&~x11 | x00&x01&x10&~x11;
  assign yup = ~y00&~y01&~y10&y11 | ~y00&y01&y10&y11 | y00&~y01&~y10&~y11 | y00&y01&y10&~y11;
  assign xdn = ~x00&~x01&x10&~x11 | ~x00&x01&~x10&~x11 | x00&~x01&x10&x11 | x00&x01&~x10&x11;
  assign ydn = ~y00&~y01&y10&~y11 | ~y00&y01&~y10&~y11 | y00&~y01&y10&y11 | y00&y01&~y10&y11;
  assign out = {1'b0, ML, MM, MR, 2'b0, y, 2'b0, x};

  always @ (posedge clk) begin
    x00 <= in[3]; x01 <= x00; x10 <= in[2]; x11 <= x10;
    y00 <= in[1]; y01 <= y00; y10 <= in[0]; y11 <= y10;
    MR <= ~in[4]; MM <= ~in[5]; ML <= ~in[6];
    x <= xup ? x+1 : xdn ? x-1 : x;
    y <= yup ? y+1 : ydn ? y-1 : y;
  end
endmodule
```

17.2.3. The SPI interface for the SD-card (disk) and the Net

SPI (Standard Peripheral Interface) is similar to PS/2, and also synchronous. However, there may be many participants. They are configured in a loop as shown in Figure 17.7, and the clock is provided by a master, namely the RISC. SPI requires 3 wires (apart from ground).

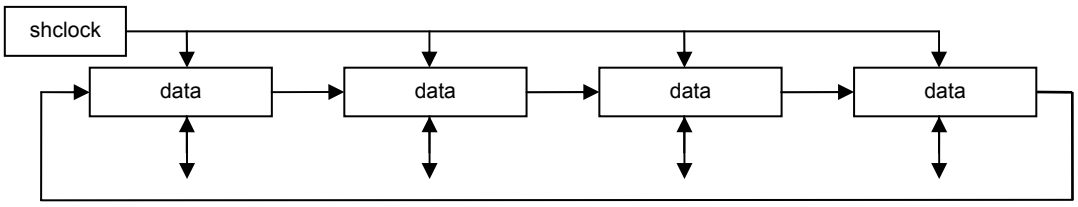

Figure 17.7 SPI-configuration as a ring

Here, however, no use is made of SPI's ring topology. Instead, One master interface is serving both the disk and the net. The connection is determined in module RISC5Top. The packet (and thus the shift register) is 32 bits long

Transmission frequency is 0.4 MHz at startup (as required by the SD-card), and then is raised to 8.33 MHz.. Details are shown in the respective program listing.

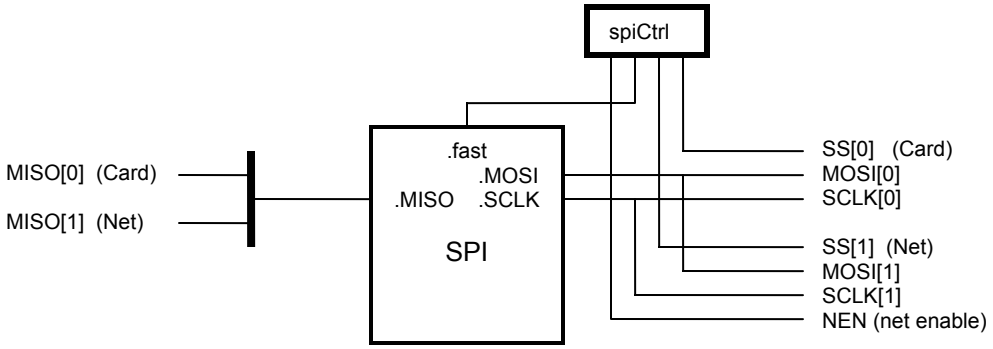

Figure 17.8 Connections between SPI, SD-card, and Net (see RISCTop5.v)

```
// Motorola Serial Peripheral Interface (SPI) PDR 23.3.12 / 16.10.13
// transmitter / receiver of words (fast, clk/3) or bytes (slow, clk/64)
// e.g 8.33MHz or ~400KHz respectively at 25MHz (slow needed for SD-card init)
// note: bytes are always MSbit first; but if fast, words are LSByte first

module SPI(
  input clk, rst,
  input start, fast,
  input [31:0] dataTx,
  output [31:0] dataRx,
  output reg rdy,
  input MISO,
  output MOSI, SCLK);
endmodule
```

The SPI specifications postulate that bytes are sent with the most significant bit first. This results in a somewhat twisted scheme for shifting bits (see Fig. 17.9).

```
shreg <= {shreg[30:24], MISO, shreg[22:16], shreg[31], shreg[14:8],
      shreg[23], shreg[6:0], shreg[15]}
```

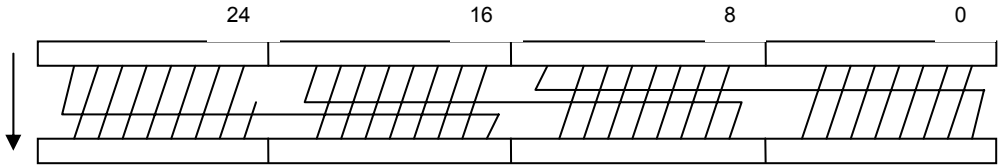

Figure 17.9 Shifting with MSB first

17.2.4. The display controller

A controller for a raster scan display feeds data from memory to the display. The data area in memory is called *frame buffer*. It contains a fixed number of bits for each pixel on the screen. In this case, there is exactly one bit per pixel, signalling black or white. For a 1024 x 768 pixel display area, 96 Kbyte are required.

The pixel position on the display is not determined by an address. Instead, data are received by rhe display purely sequentially, and the position is indirectly determined by two synchronization signals, *hsync* (for horizontal sync) at the end of each line, and *vsync* (for vertical sync) at the end of every frame. This scheme originates from cathode ray tube (CRT) monitors, where an electron beam is sweeping the screen. It is deflected by magnetic fields, which require some time to sweep back. The timing with retrace periods was retained for LCD displays as a legacy.

The heart of the controller consists of a data *buffer* (32 bits) fed from memory and shifted out bit by bit to the display, and of two counters *hcnt* and *vcnt*, representing the horizontal and vertical coordinates.The memory word address is derived from *hcnt* and *vcnt*:

vidadr = (hcnt DIV 32) + (vcnt * 32) + org

Every line consists of 1024 pixels (32 words). The challenge is to find a design with as few registers and comparators as possible. There are two signals for suppressing video data: *hblank, vblan*k. They are needed for turning the light off durich retrace.

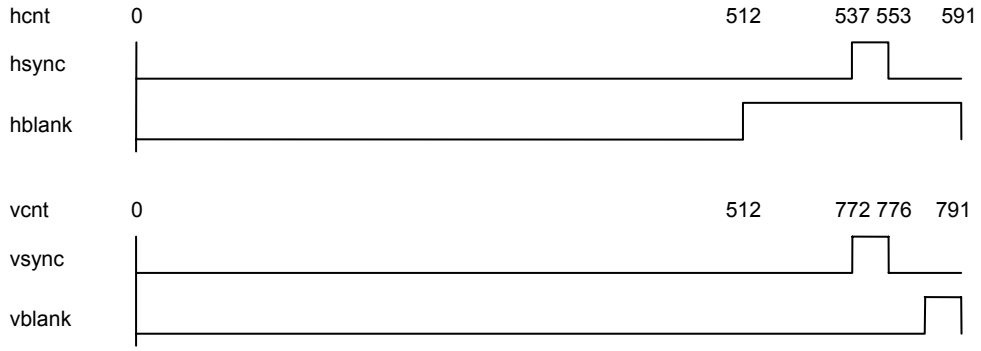

Figure 17.10 Synchronization and blanking signals

Let us generalize this scheme to displays of *w* pixels per line and *h* lines per frame. Also, let *w'* be the number of pixels per line including those of the retrace time, and *h'* be the number of lines including the vertical retrace. Also, let the number of displayed frames per second be *n*. Then the pixel frequency is

$f = w' \times h' \times n$.

This will in all probability be different from the system clock's frequency. Therefore the need arises for a diffenernt pixel clock. It is generated by the FPGA's built-in *digital clock manager* (dcm). It multiplies and divides the system clock by selectable factors. Note that the refresh rate may vary within certain bounds for all brands of monitors. Therefore, a simple factor may be chosen for division and multiplication. Examples:

(1024 x 768)	1182 x 791 x 60 = 56'097'720	rounded up to	60 MHz
(1280 x 1024)	1536 x 1280 x 60 = 117'964'800	rounded up to	125 MHz

The pixel buffer is fed from the video buffer driven by the system clock, and it is shifted and read by the pixel clock. This makes a double-buffering necessary, as shown in Figure 17.11. Also the counters are driven by this *pixel clock*. The numbers for *hcnt* and *vcnt* shown are, of course, device-specific (see Figure 17.10).

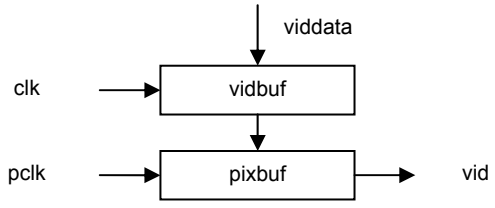

Figure 17.11 Buffering the video output

```
module VID(
   input clk, clk25, inv,
   input [31:0] viddata,
   output reg req,  // read request
   output hsync, vsync,  // to display
```

```
    output [17:0] vidadr,
    output [2:0] RGB);
  localparam Org = 18'b1101_1111_1111_0000_00;  // DFF00
  reg [9:0] hcnt, vcnt;
  reg [4:0] hword;  // from hcnt, but latched in the clk domain
  reg [31:0] vidbuf, pixbuf;
  reg hblank;
  endmodule
```

Both the display controller and the processor access memory directly. It therefore becomes necessary to arbitrate in the case where both require access simultaneously, that is, to decide which has priority. The decision is simple, because the display controller is time-critical and must not be delayed. The processor, on the other hand, can easily be delayed by the already present stalling scheme. The signal (wire) *dspreq* stalls the processor (stallX) and decides whether the memory address (SRadr) should be taken from the processor (adr) or the display controller (vidadr). The following multiplexer is placed in module RISCTop:

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

17.2.5. The RS-232 interface

RS-232 is an old standard for serial data transmission (see also Sect. 9.4). We chose to describe it here in detail because of its frequent use and inherent simplicity. RS-232 uses 2 wires (apart from ground), one for input (RxD) and one for output (TxD) as shown in Figure 17.12. Data are transmitted in packets of a fixed length, here of length 8, i.e. byte-wise. Since there is no clock wire, bytes are transmitted *asynchronously*. Their beginning is marked by a start-bit, and at the end a stop-bit is appended. Hence, a packet is 10 bits long (see Figure 17.13). Within a packet, transmission is synchronous, i.e. with a fixed clock rate, on which transmitter and receiver agree. The packet length is short enough to admit slight deviations. The standard defines several packet lengths and many clock rates. Here we use a rate of 19200 or 115200 bit/s.

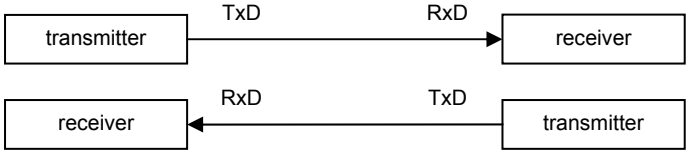

Figure 17.12 RS-232 configuration

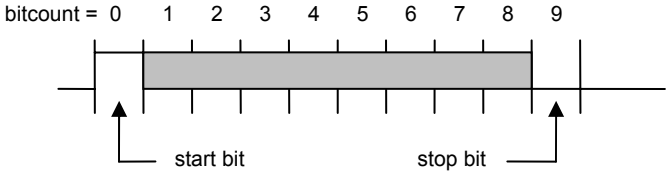

Figure 17.13 RS-232 packet format

The input signal *start* triggers the state machine by setting register *run*. The transmitter has 2 counters and a shift register. Counter *tick* runs from 0 to 1302, yielding a frequency of 25'000 / 1302 = 19.2 KHz, the transmission rate for bits. The signal *endtick* advances counter *bitcnt*, running from 0 to 9 (the number of bits in a packet). Signal *endbit* resets *ru*n and the counter to 0. Signal *rdy* indicates whether or not a next byte can be loaded and sent.

```
module RS232T(
    input clk, rst,  // system clock, 25 MHz
    input start, // request to accept and send a byte
    input [7:0] data,
```

```
    output rdy,   // status
    output TxD);  // serial data

wire endtick, endbit;
reg run;
reg [11:0] tick;
reg [3:0] bitcnt;
reg [8:0] shreg;

assign endtick = tick == 1302;
assign endbit = bitcnt == 9;
assign rdy = ~run;
assign TxD = shreg[0];

always @ (posedge clk) begin
  run <= (~rst | endtick & endbit) ? 0 : start ? 1 : run;
  tick <= (run & ~endtick) ? tick + 1 : 0;
  bitcnt <= (endtick & ~endbit) ? bitcnt + 1 :
    (endtick & endbit) ? 0 : bitcnt;
  shreg <= (~rst) ? 1 : start ? {data, 1'b0} :
    endtick ? {1'b1, shreg[8:1]} : shreg;
end
endmodule
```

The receiver is structured very similarly with 2 counters and a shift register. The state machine is triggered by an incoming start bit at RxD. The state *rdy* is set when the last data bit has been received, and it is reset by the *done* signal, generated when reading a byte. The line RxD is sampled in the middle of the bit period rather than at the end, namely when *midtick* = *endtick/2*.

```
module RS232R(
    input clk, rst,
    input done,   // "byte has been read"
    input RxD,
    output rdy,
    output [7:0] data);

wire endtick, midtick;
reg run, stat;
reg [11:0] tick;
reg [3:0] bitcnt;
reg [7:0] shreg;

assign endtick = tick == 1302;
assign midtick = tick == 651;
assign endbit = bitcnt == 8;
assign data = shreg;
assign rdy = stat;

always @ (posedge clk) begin
  run <= (~RxD) | (~rst | endtick & endbit) & run;
  tick <= (run & ~endtick) ? tick + 1 : 0;
  bitcnt <= (endtick & ~endbit) ? bitcnt + 1 :
    (endtick & endbit) ? 0 : bitcnt;
  shreg <= midtick ? {RxD, shreg[7:1]} : shreg;
  stat <= (endtick & endbit) ? 1 : (~rst | done) ? 0 : stat;
end
endmodule
```