Technical Report August. 2010

Experiments in Computer System Design

Niklaus Wirth

Table of Contents

PART 5

Technical Report 2. 8. 2010

Experiments in Computer System Design

Niklaus Wirth

PART 1

Introduction and Perspective

Modern Field Programmable Gate Arrays (FPGA) provide an ideal ground for experiments in the design of computer systems and of computer architecture. Here we present the designs of a processor, of channels, of a communicating ring, of a memory interface, and of a floating-point unit. These are described as Verilog texts, representing circuits to be automatically generated by synthesizer, placer and router tools. On a higher level, systems can then be described as consisting of such components. Thus the systems are flexibly configurable.

In particular, it is possible to configure multicore systems, and to experiment with various configuration and models of cooperation. The designs are implemented on a single FPGA. We use a commercial development board connected to a host computer.

Not only experience in designing software involving many processors is becoming more important (mostly due to the availability of multi-core chips), but also experience in designing entire systems including the hardware. A configuration of 12 processors (described below) has been realized on a single chip. Hardware design, processor architecture, communication links, system configuration are all covered by the presented tool kit, including, of course, their programming. It is therefore an ideal experimenting ground for modern computer practice and experience. Directing this project towards requirements in education, we try to present it in the style of a tutorial.

In Part 1 we describe a simple processor. It is called TRM (for Tiny Register Machine). It is a sound principle in teaching a new subject, to concentrate on its essential ingredients. This principle had originally been followed closely by the designers of RISC architectures, and therefore our design does so as well. In addition, we must consider the limited resources available on an FPGA, even a large one, and in particular, if we wish to place many of these processors on a single chip. A straight-forward design is therefore mandatory.

In Part 2 we describe communication facilities. These are a uni-directional channel (point-to-point connection), a ring (connecting many processors), and, for the sake of utility, an RS-232 transmitter and receiver. The latter is used to connect the FPGA development board with the host computer.

In Part 3 we describe an interface between a processor and a large memory of the DDR type (dynamic RAM). In a first step, the memory is considered as an external device accessed through the processor's I/O bus. A much faster solution

is presented in a second step. It includes a direct memory access channel (DMA). Of course a final solution will be the use of a cache memory.

This small, initial set of hardware components can be augmented freely. Other processors may be added, more sophisticated links, and drivers for other devices, several of which are available on the development board currently used (ML-505). On the host computer reside a compiler (in our case for the programming language Oberon), and a system for (down-) loading the bitstream file (configuration file) onto the target FPGA..

This configurable system actually emerged from a project, whose ultimate goal was an application suitable to demonstrate the power of multi-processor systems. Its somewhat grandiose title was *Supercomputer in a Pocket*. The target application was a surveillance system for heart diseases. Signal analysis required reasonably large computing power, its being carried by patients required reasonably low power consumption and a small size to fit into a pocket. The system elements described below were used in this application.

We first describe the architecture, the programmer's interface, of the processor, and a brief account of its history of development. Thereafter we present its circuit interface and its implementation.

The Tiny Stack Machine (TSM)

The initial impulse for our design of a processor came from Ch. Thacker's Simple-32 architecture, designed at Microsoft Research in Mountain View. Thacker called it an "FPGA-optimized computer architecture". The Simple-32 strongly mirrors a modern RISC architecture with a bank of 16 registers. Most instructions feature a 4-bit opcode and three 4-bit register fields. Thus, the Simple-32 is a typical 3-address machine.

The Simple-32 architecture, however, appeared as not optimally suited for use with a high-level programming language, at least not without a compiler optimizing register usage. It was felt that just as the architecture ought to be perspicuous, so should be the compiler, that is, the instruction sequence compiled from a given piece of source program should be quite predictable. Certainly, due to small memory size, an overriding consideration was code density. Our experience of the past decades let a stack architecture appear as most desirable. Instructions for a stack machine do not contain register numbers. They are implicit and derived at run-time from the stack principle, which naturally governs the evaluation of expressions.

It is important to note that this expression stack stands in place of the register bank of a RISC architecture. This stack is typically implemented by a set of registers plus a register pointer, a 4-bit register connected with an up-down counter. It is to be distinguished from the stack of procedure activation records, which is entirely a concept of software architecture, and which is stored in the local data memory and uses a register as stack pointer.

Many instructions of a stack-oriented architecture fit into a single byte. Even instructions containing a short, immediate operand fit into a byte. Instructions containing a larger literal, such as an address, are placed in two (or 3) bytes. Therefore, a variable-length instruction scheme is required.

As is to be expected, the price for an expression stack and a variable-length instruction fetch machinery is an increase in complexity of the decoding circuitry. The longer signal path lengths and propagation times result in a longer clock cycle and thus a decrease in speed.. This is particularly noticeable, if pipelining is desired to speed up instruction interpretation. This complexity proved to be such that it was decided to abandon the idea of a stack architecture and to return to a register array in place of the register stack. The price is less code density, longer codes, and that program length hits the available limits sooner.

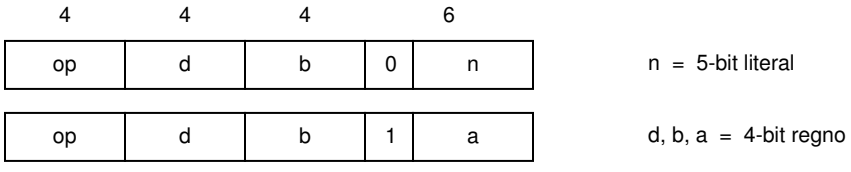
The Tiny Register Machine TRM-1

At this point, we need to know more about the available resources on the FPGA chip. Apart from the regular cells with gates and registers, and apart from routing resources, the FPGA contains 60 static RAM blocks (BRAMs), of 1K words of 36 bits. The foremost question is how to make use of these BRAMs. Considering that we wish to be able to place many processors on the chip, we decided to allocate 4 BRAMs to a processor. The maximum number of processors in any configuration is therefore 15.

Each processor contains 4 BRAMs. We allocate 2 for data and 2 for program. By using half words for instructions we obtain a maximum program size of 4K (4096) instructions. Evidently, programs will be quite small, and this justifies the word *tiny* in the machine's name. Evidently, the TRM uses a Harvard architecture, where data and program. memories are separate and are accessed through separate ports.

The essential property of the RISC architecture is that all operations on data are executed in registers. Instructions accessing memories treat memories like being external. There are only 2 of them: *Load* and *Store*. Instructions performing operations typically specify 3 registers, 2 for the arguments, 1 for the result. Such a scheme is called a *3-address architecture*. An instruction thus has 4 fields: the operation code and 3 register numbers.

It is most desirable, that (at least) one of the arguments may be a literal instead of a register. This leads us to the following format for register instruction, with one bit as a tag distinguishing whether the second argument is a register or a literal.



R.d := R.b **op** n or R.d := R.b **op** R.a

An unpleasant property of the short instruction (18 bits) is that the field for immediate operands and addresses is very short. The decision to use not only 16, but all 18 bits of the FPGA's block memories alleviated this problem to some degree. However, the necessity to use a separate instruction each time a constant or an address longer than 5 bits is present, was felt to be too detrimental to code density and efficiency, and a better solution was sought, resulting in the TRM-2.

The Tiny Register Machine TRM-2

The question was how to obtain more bits for the literal field. Three measures were used:

1. Replacing the 3-address architecture by a 2-address architecture, where the destination register is the same as one of the argument registers. This causes relatively little loss in flexibility and code density, unless a sophisticated scheme of register allocation is used for complex expressions.
2. Reducing of the number of registers from 16 to 8. This saves 1 bit in each register field.
3. Using Huffman encoding, with short opcode fields (and longer literal fields) for the most frequently used instruction, and longer opcodes for less frequently used instructions.

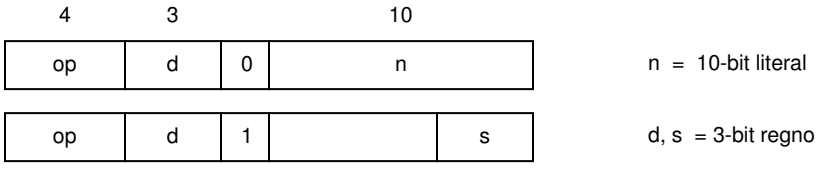
All this resulted in address offsets of 10 bits for data and 12 bits for branch instructions, the latter covering the entire address space of the instruction memory. Quite obviously, this leads to a very considerable increase in code density.

So far, so good. But every gain causes a loss somewhere else. The Huffman coding required a more complex decoding circuitry. Instruction decoding lies in the critical signal path of every instruction, and here it proved to be a bottleneck. For some instructions, the desired clock frequency could not be achieved. Further deliberations led to the TRM-3.

The Tiny Register Machine TRM-3

In this design the measures 1 and 2 leading to TRM-2 were retained, but Huffman coding was dropped. Thereby the same speed as with TRM-1 is achieved, and the literal field in register instructions is still 10 bits long. The instruction formats are for

Register operations



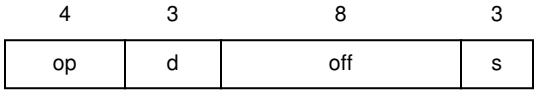
R.d := R.d **op** n or R.d := R.d **op** R.s

We can now turn to the selection of instructions to be represented. This selection is essentially determined by the programming language envisaged, i.e. the operators used in expressions. However, in this respect most general-purpose languages feature the same requirements: Arithmetic and logical instructions. In detail, they are the following:

op	operation	
0	MOV	R.d := R.s (in place of R.s may stand the literal imm)
1	NOT	R.d := ~R.s
2	ADD	R.d := R.d + R.s
3	SUB	R.d := R.d - R.s
4	AND	R.d := R.d & R.s
5	BIC	R.d := R.d & ~R.s
6	OR	R.d := R.d | R.s
7	XOR	R.d := R.d xor R.s
8	MUL	R.d := R.d * R.s
9	DIV	R.d := R.d div R.s
10	ROR	R.d := R.d ror R.s (rotate right)
11	BR	PC := R.s (see below)

In addition there are the Load and Store instructions providing access to memory. Their format is slightly different. The actual address is computed as the sum of base address R.s and an offset:

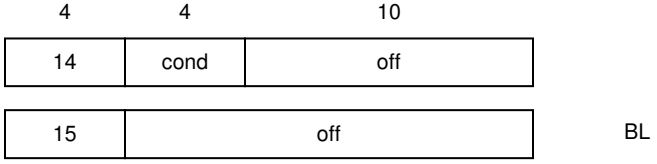
Load and Store instructions



op	operation	
12	LD	R.d := Mem[R.s + adr]
13	ST	Mem[R.s + adr] := R.d

The only remaining instructions are branch instructions used for implementing conditional and repetitive statements, i.e. if, while, repeat and for statements. They are executed conditionally, i.e. when a condition is satisfied. These conditions are the result of preceding register instructions, and they are held in 4 *condition registers N, Z, C, V*, defined as shown below. The branch and link instruction is unconditional. It is used to implement procedure calls. It stores the current value of the *program counter PC* in R7.

Branch instructions

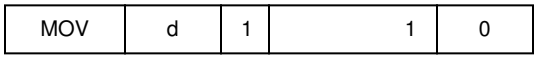


if condition then PC := PC + 1 + offset

code	mnemonic	condition	
0000	EQ	equal (zero)	Z
0001	NE	not equal	~Z
0010	CS	carry set	C
0011	CC	carry clear	~C
0100	MI	negative (minus)	N
0101	PL	positive (plus)	~N
0110	VS	overflow set	V
0111	VC	overflow clear	~V
1000	HI	high	~(~C|Z)
1001	LS	less or same	~C|Z
1010	GE	greater or equal	~(N≠V)
1011	LT	less than	N≠V
1100	GT	greater than	~((N≠V)|Z)
1101	LE	less or equal	(N≠V)|Z
1110		true	T
1111		false	F

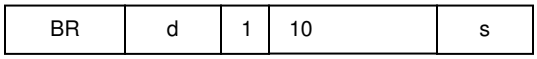
Special instructions

The TRM furthermore features some special instructions. The first to be mentioned is an instruction to obtain the high part of a product. Multiplications generate a result of 64 bits. The high-order part is usually ignored, but it is stored in a special register H. The LDH instruction fetches this value.



LDH

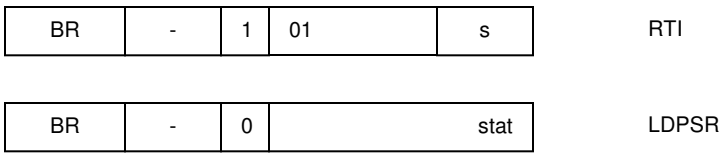
The instruction to return from a procedure is BR, branching with the address taken from register R.s. This instruction also allows the current PC+1 to be stored in R.d. This instruction is used for calling procedures which are a formal parameter or are represented by a variable (methods).



BLR

The TRM also features an *interrupt facility*. There are 2 external signals that can cause an interrupt. It functions like a procedure call. As the place in a program where an interrupt may be triggered is unknown, the state of the machine must be preserved in order to be recovered after the interrupt was handled. Thus the TRM switches to *interrupt mode*, in which it uses a second bank of registers and stores the PC and the condition bits in R7 of this bank. Further interrupts are immediately disabled.

The return from interrupt to normal mode is caused by an RTI instruction, a slight variant of the BR. It restores PC and the condition bits, and re-enables interrupts. The interrupt facility requires a *processor status register* (PSR) indicating the processor mode and whether or not interrupts are enabled.



The LDPSR instruction loads the Program Status Register from its literal field with the following bit assignments:

0 interrupt 0 enable
1 interrupt 1 enable
2 processor mode (0 = normal; 1 = interrupted)
3 cache enable (if available)

The Implementation of the TRM-3

The circuit representing the TRM processor is described in Verilog. Any system description in Verilog is composed as a hierarchy of modules. Only the top module can specify signals leaving or entering the FPGA. All such signals used in a subordinate module must flow through the top module. The TRM system consists of 4 modules, the TRM itself, the RS232R for receiving signals from the RS-232 serial line, the RS232T for transmitting signals to the serial line, and the top module. We will first describe the TRM module, the heart of the system.

The hardware interface of the TRM module follows the example of typical micro-processors. The inputs are:

clk	the processor clock (116 MHz)
rst	reset, active low
stall	if high, causes the processor to stall
irq0, irq1	interrupt signals, active high
inbus	32-bit bus

The output signals are

iord, iowr	read and write enable
ioadr	6-bit I/O address
outbus	32-bit bus

The *iord* signal is included, because some read commands may not only read data, but also change the state of the device, such as moving a buffer pointer ahead for sequential access.

The processor essentially consists of two sections, the *data processing unit,* computing the results of single instructions, and the *control unit*, controlling the sequence of instructions.

The data processing unit

The choice of functions to be computed by the *Arithmetic/Logic Unit* (ALU) is, as said before, much determined by the programming language to be implemented.

But not only. The second factor are the resources available. The early RISC designs held to principle that every instruction should be executed in a single clock tick. This is readily possible for addition and the logical operations. But already a shifter may cause difficulties, let alone multiplication and division. They are inherently more complex than the former. There are three solutions to the dilemma: The first is to provide more and faster circuitry – possible only within limits -, and the second is to give up the principle, i.e. to allow some operations to take more than a single clock cycle. The third solution is to omit the operation altogether. Indeed, early RISC designs left out multiplication and division instructions, as these are relatively rare operations – in particular division.

In our context, we let multiplication and division take 32 cycles. This requires that the control unit can be stopped from progressing to the next instruction. The signal indicating such delay is called *stall*. Both the multiplication and the division units have a stall signal as output. Fortunately, it proved to be possible to implement a full barrel shifter operating within a single cycle.

The processing unit consists of the *Arithmetic/Logic Unit* (ALU) and a set of 8 registers. The ALU is – apart from multiplier and divider - a purely combinational circuit yielding results of arithmetic or logical operations. The main data path of the processor forms a loop from selected source registers (A, B) through the ALU to a multiplexer (aluRes) back to a destination register. The multiplexer in the A-path determines, whether the A-operand is a register or a literal, i.e. a constant in the instruction IR. The additional registers H and CC store the high-order part of a product, and the conditions N, Z, C, V respectively.

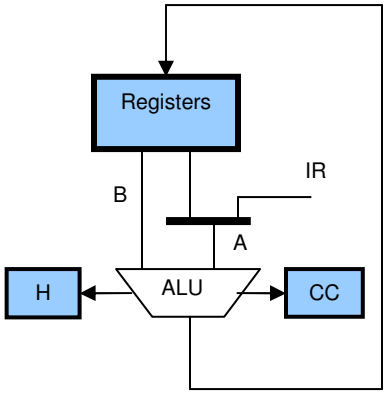


Fig. 1.1. ALU

The register bank is generated from 32 dual-port LUT slices (RAM16X1D). Its addresses are the register numbers denoted by *dst* and *irs*, where *dst* denotes both the destination and the first source. The register numbers stem from the instruction register in the control until. The data input comes through *regmux* from various sources (defined below), including *aluRes*. The data path is 32 bits wide. Declarations and definitions of signals (wires) are shown below. The signals *op, imm, ird, irs,* and *off* are fields of the instruction register IR, which belongs to the control unit.

```
wire [3:0] op;
wire [9:0] imm;
wire [2:0] ird, irs, dst;
wire [7:0] off;

wire [31:0] AA, A, B, s1, s2, s3, divRes, remRes;
wire [32:0] aluRes;
wire [63:0] mulRes;
wire MOV, NOT, ADD, SUB, MUL, DIV, AND, BIC, OR, XOR, ROR;
wire BR, LDR, ST, Bc, BL, ADSB;

assign op  = IR[17:14];
assign ird = IR[13:11];
assign irs = IR[2:0];
assign imm = {22'b0, IR[9:0]};
assign off = {4'b0, IR[10:3]};

assign MOV = (op == 0);
assign NOT = (op == 1);
assign ADD = (op == 2);
assign SUB = (op == 3);
assign AND = (op == 4);
assign BIC = (op == 5);
assign OR  = (op == 6);
assign XOR = (op == 7);
assign MUL = (op == 8);
assign DIV = (op == 9);
assign ROR = (op == 10);
assign BR  = (op == 11);
assign LDR = (op == 12);
assign ST  = (op == 13);
assign Bc  = (op == 14);
assign BL  = (op == 15);
assign ADSB = (IR[17:15] == 1);  // ADD | SUB

assign A = (IR[10]) ? AA: {22'b0, imm};
assign regwr = (~ST & ~ … );
assign aluRes =
   (MOV) ? A :
   (NOT) ? ~A :
   (ADD) ? {B[31], B} + {A[31], A} :
   (SUB) ? {B[31], B} - {A[31], A} :
   (AND) ? B & A :
   (BIC) ? B & ~A :
   (OR)  ? B | A :
   (MUL) ? mulRes[31:0] :
   (DIV) ? divRes : B ^ A;   // XOR
```

The register bank is implemented by 32 1-bit LUT RAM-slices, expressed in Verilog by a generate statement. There are 2 addresses (register numbers). The first is *dst* (stemming from instruction field *ird*), controlling RAM input *D* (regmux) and RAM output *B*, and the second is *irs*, controlling RAM output *AA*.

```
genvar i;

generate    //dual port register file
  for (i = 0; i < 32; i = i+1)
  begin: rf32
```

```
  RAM16X1D_1 # (.INIT(16'h0000))
  rfa(
  .DPO(AA[i]), // data out
  .SPO(B[i]),
  .A0(dst[0]),   // R/W address, controls D and SPO
  .A1(dst[1]),
  .A2(dst[2]),
  .A3(intMd),
  .D(regmux[i]),  // data in
  .DPRA0(irs[0]), // read-only adr, controls DPO
  .DPRA1(irs[1]),
  .DPRA2(irs[2]),
  .DPRA3(intMd),
  .WCLK(~clk),
  .WE(regwr));
  end
endgenerate
```

Apart from the 8 32-bit registers the data processing unit contains four 1-bit registers: N, Z, C and V. Together they form the *condition code*. It is set by the general instructions and tested by conditional branch instructions. N indicates whether a result is negative, and Z whether it is zero. C and V hold the carry and overflow bits of additions and subtractions. There is also the 32-bit register H holding the high order part of products or the remainder of divisions.

```
always @ (posedge clk)
  if (regwr) begin
    N <= aluRes[31];
    Z <= (aluRes == 0);
    C <= (ADSB) ? aluRes[32] : (ROR) ? s3[0] : C;
    V <= (ADSB) ? (aluRes[32] ^ aluRes[31]) : V;
    H <= (MUL) ? mulRes[63:32] : (DIV) ? remRes : H;
end
```

The Shifter

The TRM has only a single shift instruction. It rotates to the right. The rotate mode was chosen, because it does not lose any information; all bits are still present unchanged, albeit at another position. Hence, all other shift modes can be derived from rotation with the help of masking. The shifter is a barrel shifter. This implies that any amount of shift is possible with one instruction, i.e. the shift count ranges from 0 to 31.

Typically, shifters are built from a series of multiplexers, the first shifting by 0 or 1, the second by 0 or 2, etc. the fifth by 0 or 16. Here, we use 4-input multiplexers (a number favored by Xilinx FPGA cells), and thus can reduce the series from 5 to 3, denoted by *s1, s2*, and *s3*. Now the first multiplexer shifts by 0, 1, 2, or 3, the second by 0, 4, 8, or 12, and the third by 0 or 16. A generate statement is used to build the 32 multiplexers for each stage. The shift count is A[4:0]. The output *s3* goes to *regmux* instead of *aluRes*. (Note: “%” denotes modulo in Verilog).

```
wire [1:0] sc1, sc0;
wire [31:0] s1, s2, s3;
```

```
assign sc0 = A[1:0];
assign sc1 = A[3:2];

generate
  for (i = 0; i < 32; i = i+1)
  begin: rotblock
    assign s1[i] = (sc0 == 3) ? B[(i+3)%32] : (sc0 == 2) ? B[(i+2)%32] :
          (sc0 == 1) ? B[(i+1)%32] : B[i];
    assign s2[i] = (sc1 == 3) ? s1[(i+12)%32] : (sc1 == 2) ? s1[(i+8)%32] :
          (sc1 == 1) ? s1[(i+4)%32] : s1[i];
    assign s3[i] = A[4] ? s2[(i+16)%32] : s2[i];
  end
endgenerate
```

The Multiplier

The multiplier is declared as a separate module, instantiated by the following statement:

```
Multiplier mulUnit (CLK(clk), .mul(MUL),
   .A(A), .B(B),
   .stall(stallM), .mulRes(mulRes));
```

The multiplier described below follows the traditional algorithm of *n* add-shift steps, where *n* is the word length, here 32.

```
s := 0;  (*x is the multiplier, y the multiplicand*)
REPEAT
   IF ODD(x) THEN z := z+y END ;
   x := x DIV 2; z := z DIV 2; INC(s)  (*right shift*)
UNTIL s = 32
```

This implies that the multiplier is a state machine. Its state is a counter *S* running from 0 to 32. We use a double-length register, here called *Hi* (initialized to 0) and *Lo* (initialized with the multiplier *B*. In each step, the multiplicand *A* is added to the high part, if the least bit of the multiplier is 1. Then the register is shifted one bit to the right.

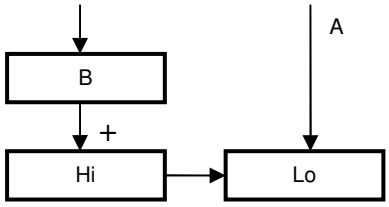


Fig. 1.2. Multiplier

It is important to consider also negative numbers. Whereas negative multiplicands do not pose a problem, this is not obvious for negative multipliers. However, there is an elegant solution. Considering the value *x* (represented in 2's complement form) as the sum

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

it is obvious that the solution lies in subtracting rather than adding the multiplicand in the last step, because term 31 has a minus sign. Note that we

introduce *Hix* and *Bx* as extended versions of *Hi* and *B*. This is necessary, because the carry bit of addition must not be lost. It enters register *Hi* with the right shift.

```
module Multiplier(
  input CLK, mul,
  output stall,
  input [31:0] A, B,
  output [63:0] mulRes);

reg [5:0] S;    // state
reg [31:0] Hi, Lo;  // high and low parts of partial product
wire [32:0] p, Hix, Bx;

assign stall = mul & ~S[5];
assign Hix = {Hi[31], Hi};
assign Bx = {B[31], B};
assign p = (S == 0) ? (A[0] ? Bx : 0) :
   Lo[0] ? ((S == 31) ? (Hix - Bx) : (Hix + Bx)) : Hix;
assign mulRes = {Hi, Lo};

always @ (posedge(CLK)) begin
  if (mul & stall) begin
    Hi <= p[32:1];
    Lo <= (S == 0) ? {p[0], A[31:1]} : {p[0], Lo[31:1]};
    S <= S + 1; end
  else if (mul) S <= 0;
end
endmodule
```

The parameter *mul* indicates a multiplication in progress. The *stall* signal is asserted, when *mul* is 1 and *S* has not yet reached the value 32.

The FPGA used in this project features (a large number of) DSPs (digital signal processors). A DSP can be used to speed up multiplication, because it can multiply two 18-bit numbers in a single clock tick. Thus, we need only 4 (instead of 32) cycles for a multiplication of two 32-bit arguments. We refrain from presenting this solution here, because it is rather complicated and highly dependent on the particular DSP design.

The Divider

The divider is declared as a separate module, instantiated by the following statement:

```
Divider divUnit(.clk (clk), .div(DIV),
   .x(B), y(A),
   .stall(stallD),
   .quot(divRes), .rem(remRes));
```

The divider described below follows the traditional algorithm with *n* shift-subtract steps, where *n* is the wordlength.

```
s := 0; r := x; q := 0; (*x is the dividend, y the divisor*)
REPEAT (*q*y + r = x*)
   r := 2*r; q := 2*q; INC(s);  (*left shift*)
   IF r >= Y THEN r := r - Y END ;  (*Y = 2^32*y*)
```

```
UNTIL s = 32
(*q is the quotient, r the remainder*)
```

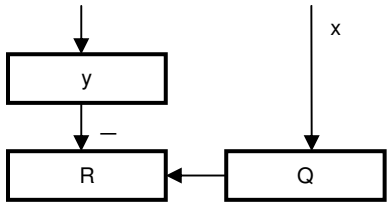


Fig. 1.3. Divider

This implies that also the divider is a state machine. Its state is represented by the counter *S* running from 0 to 32

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

reg [5:0] S;  // state
reg [31:0] R, Q;  // remainder, quotient
wire [31:0] xa, rsh, qsh, d;

assign stall = div & ~S[5];
assign xa = (x[31]) ? -x : x;
assign rsh = (S == 0) ? 0 : {R[30:0], Q[31]};
assign qsh = (S == 0) ? {xa[30:0], ~d[31]} : {Q[30:0], ~d[31]};
assign d = rsh - y;
assign quot = (~x[31]) ? Q : (R == 0) ? -Q : -Q-1;
assign rem =  (~x[31]) ? R : (R == 0) ? 0 : y - R;

always @ (posedge(clk)) begin
  if (div & stall) begin
    R <= (~d[31]) ? d : rsh; Q <= qsh; S <= S + 1; end
  else if (div) S <= 0;
end
endmodule
```

The dividend is taken as the absolute value of *x*. In case of a negative *x*, a correction is made after the computation of quotient and remainder:

```
IF x < 0 THEN
   IF r = 0 THEN q := -q ELSE q := -q-1; r := y-r END
END
```

The Local Data Memory

The (local) data memory is composed of two 1Kx32 block RAMs. This is expressed by the macro *dbram32*, with *dmin* and *dmout* as input and output ports, *dmadr* as its 11-bit address, and *dmwr* as write enable.:

```
assign dmadr = ((irs == 7) ? 0 : AA[11:0]) + {4'b0, offset};
assign dmwr = ST;  // write for store instructions only
```

```
assign dmin = B;

dbram32 DM (.wda(dmin),  //write port
   .aa (dmadr[10:0]),
   .wea (dmwr),
   .clka (clk),
   .rdb (dmout),  //read port
   .ab (dmadr[10:0]),
   .enb (1'b1),
   .clkb (clk));

assign regmux =
   (LDR) ? dmout :  // read for LDR instructions only
   (ROR) : s3 :
   aluRes;
```

(Other terms will be added to *regmux*, the registers' input, later).

The control unit

The control unit fetches instructions from the program memory into the *instruction register* IR and computes the address of the next instruction. This is the old address, held in the *program counter* (PC), plus 1, except for branch instructions. In their case, the address of the next instruction is the sum of the current location and the (signed) offset in the current branch instruction. The signal and register declarations are shown below together with the macro for the program memory.

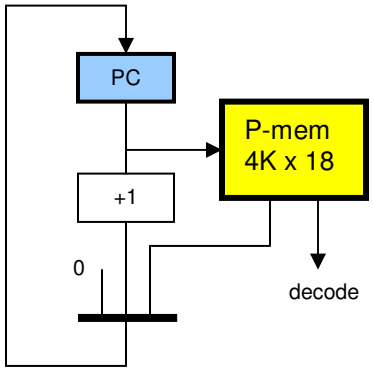


Fig. 1.4 Control Unit

```
reg [11:0] PC;

wire [17:0] IR;  // 36-bit register IR is contained in module pbram
wire [35:0] pmout;
wire [11:0] pcmux, nxpc;
wire cond;

pbram36 PM (.wda(36'b0),  // write port, not used
   .aa (11'b0),
   .wea (1'b0),
   .clka (clk),
   .rdb (pmout),  //read port
   .ab (pcmux[11:1]),
   .enb (1'b1),
```

```
    .clkb (clk));

assign IR = (PC[0]) ? pmout[35:18] : pmout[17:0];
assign nxpc = PC + 1;

assign pcmux =
  (~rst) ? 0 :
  (stall0) ? PC :
  (BL) ? IR[13:0] + nxpc :
  (Bc & cond) ? {IR[9], IR[9], IR[9:0]} + nxpc :
  (BR & IR[10]) ? A[11:0] : nxpc;
```

The sequencing of instructions is finally achieved by the statement

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

This rather straight-forward scheme was used for the TRM-1.

Unfortunately, reading data from local memory is slow compared to functions implemented by the normal logic cells (LUT). It required the use of a clock rate not greater than 58.3 MHz. A simple measure called (single stage) *pipelining* allows to double the clock rate to 116.6 MHz. It requires two incarnations of IR and PC. An instructions is first fetched with address PCf into IRf, and thereafter moved from IRf to IR. While it is interpreted from IR, the next instruction is fetched into IRf. This sequential flow is broken by branch instructions. In their case, a NOP instruction must be inserted, causing a hiccup, i.e. a delay of one tick. The pipelining machinery is described as follows:

```
localparam NOP = 18'b111011110000000000; // never jump

reg [11:0] PCf, PC;
reg [17:0] IR;
reg stall1;

wire [17:0] IRf;  //36-bit register IRf is contained in module pbram
wire [11:0] pcmux, nxpcF, nxpc;

always @ (posedge clk) begin
  PCf <= pcmux;
  if (~rst) begin PC <= 0; IR <= NOP; end
  else if (stall0) begin PC <= PC; IR <= IR; end
  else if ((Bc & cond) | BL | BR & IR[10])
    begin PC <= pcmux; IR <= NOP; end
  else begin PC <= PCf; IR <= IRf; end
end
```

The signal *cond* determines, whether a branch is taken or not. It is derived from the various condition code registers. Bit IR[10] inverts the sense of the condition.

```
assign cond = IR[10] ^       // xor
  ((ird == 0) & Z |          // EQ, NE
   (ird == 1) & C |          // CS, CC
   (ird == 2) & N |          // MI, PL
   (ird == 3) & V |          // VS, VC
   (ird == 4) & ~(~C|Z) |    // HI, LS
   (ird == 5) & ~S |         // GE, LT
   (ird == 6) & ~(S|Z) |     // GT, LE
   (ird == 7));              // T, F
```

Input and Output

Input and output is handled in the conventional way by including the memory data buses in the processor's interface, and by reserving a (small) portion of the address space for external devices (address-mapped I/O). Addresses 0FC0H – 0FFFH are designated for devices. This is a range of 64 addresses. If such an address is generated, the signal *ioenb* becomes active.

```
assign ioenb = (dmadr[11:6] == 6'b111111);
assign iord = LDR & ioenb;
assign iowr = ST & ioenb;
assign ioadr = dmadr[5:0];
assign outbus = B;
```

Regmux now includes an entry for input data:

```
assign regmux =
  (LDR & ~ioenb) ? dmout :
  (LDR & ioenb) ? inbus :
  (ROR) ? s3 :
  (BL | BR) ? {20'b0, nxpc} : aluRes;
```

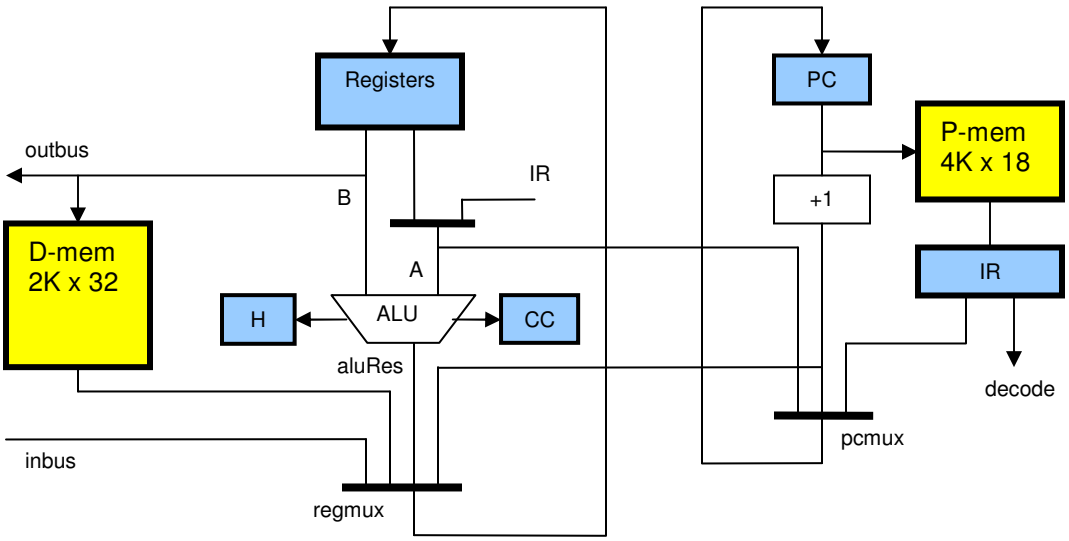


Fig. 1.5. ALU, CU, memory, and I/O

Stalling the processor

Originally, the idea behind the RISC movement was to simplify the instruction set in such a way that every instruction could be interpreted in a single clock cycle. This condition simplifies pipelining very significantly, which is a backbone of the RISC idea. It makes it desirable that all instructions, that is, all data paths cause the roughly same delays. Unfortunately, this is only possible, if exceptions are allowed. In the case of this TRM implementation, there are two (only 3) such exceptions. The first is the LD instruction, reading data from the local block RAM. It requires 2 cycles. The others are, not surprisingly, multiplication and division. They require 32 cycles.

The problem is solved by introducing a facility to stall the instruction fetch when the mentioned cases occur. The necessary additions to the TRM circuit are listed below: *stall* is an input to the TRM.

```
reg stall1;
wire stall0, stallM, stallD;
assign stall0 = (LDR & ~stall1) | stallM | stallD | stall;

assign pcmux =
  (~rst) ? 0 :
  (stall) ? PC : ... : nxpc;

always @ (posedge clk) begin // stall generation
  if (~rst) stall1 <= 0;
  else stall1 <= (LDR & ~stall1);
end
```

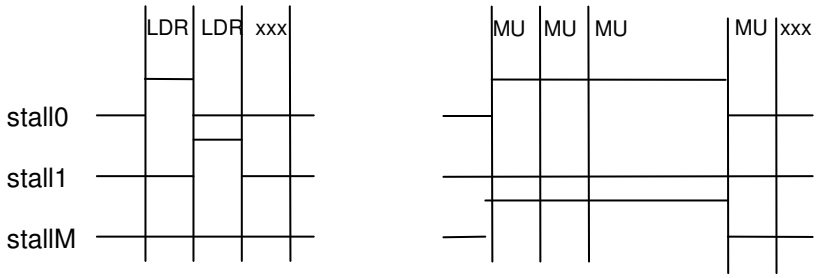


Fig. 1.6. Stalling for 1 and 32 cycles

Interrupts

An interrupt facility is necessary, if the processor needs to be able to respond quickly to signals from external devices, i.e. where (occasional) polling of such signals is inadequate. Interrupts are based on letting (external) signals determine the choice of the next instruction at any time, i.e. by directly letting them control *pcmux*. Our TRM features two distinct interrupt signals, *irq0* and *irq1*:

```
assign pcmux =
  (~rst) ? 0 :
  (irq0 & intAck)? 2:
  (irq1 & intAck)? 3: : ... : nxpc;
```

Of course, it is mandatory to preserve the processor state upon interrupt, because the interrupt may occur at any arbitrary point in the program. The interrupt resembles somewhat a procedure call, and the response to an interrupt that of executing a procedure (called *interrupt handler*). In the first place, the processor stores its current PC value in a link register, from where it can be recovered after the interrupt had been serviced. Then a fixed value according to the interrupt source is forced to the PC (2 or 3 in our case). Typically an interrupt handler would save the values of all other registers, or at least those which the handler makes itself use of. This saving and later restoring of all registers is time-consuming and not acceptable, if hard real-time constraints have to be met.

The TRM therefore features a second bank of 8 registers. Upon interrupt, the processor switches to interrupt mode by setting *intMd*, and to the use of the alternate bank (address bit 3). It thereby *disables further interrupts*, and then deposits the PC and the flag registers N, Z, C, V in the link register of the alternate bank. For all this, an extra cycle must be inserted. It is marked by the signal *intAck*.

As a consequence, a special return instruction must be provided which, in addition to restoring the PC also switches back to the normal register bank and restores N, Z, C, V. This is done by a BR instruction with IR[8] being set.

It is of course necessary to disable interrupt signals. Thus we introduce state registers *intEnb0* and *intEnb1*. Evidently, a special instruction is required to set these registers and abuse a form of the BR instruction for this purpose (with bit 10 being zero). We call this instruction *Set Processor Status* (PSR).

The additions necessary for the interrupt system are listed below, and there are remarkably few of them.

```
reg intEnb, intAck, intMd, intAck;
wire irq0e, irq1e;

assign irq0e = irq0 & intEnb0;
assign irq1e = irq1 & intEnb1;

always @ (posedge clk) begin  // interrupt and mode handling
  if (~rst) begin intEnb0 <= 0; intEnb1 <= 0; intMd <= 0; intAck <= 0; end
  else if ((irq0e | irq1e) & ~intMd & ~stall0 & ~(IR == NOP)) begin
    intAck <= 1; intMd <= 1; end
  else if (BR & IR[10] & IR[8]) intMd <= 0; // return from interrupt
  else if (BR & ~IR[10]) begin // SetPSR
    intEnb0 <= IR[0]; intEnb1 <= IR[1]; intMd <= IR[2]; end
  if (intAck & ~stall0) intAck <= 0;
end
```

Furthermore, we must provide an additional case in the code governing the PC:

```
  else if ((irq0e | irq1e) & intAck) begin PC <= PCf; IR <= NOP; end
```

For *regmux* the additional case *intAck* must be included, bringing it to its final form:

```
assign pcmux =
  (~rst) ? 0 :
  (stall0) ? PCf :
  (irq0e & intAck) ? 2 :
  (irq1e & intAck) ? 3 :
  (BL) ? IR[11:0] + nxpc :
  (Bc & cond) ? {IR[9], IR[9], IR[9:0]} + nxpc :
  (BR & IR[10]) ? A[11:0] : nxpcF;
```

This concludes the description of the TRM processor implementation.

Technical Report 8. 8. 2010

Experiments in Computer System Design

Niklaus Wirth

PART 2

The Environment and the Top Module

In order to be useful, a processor must be made available to users through its environment. It must connect to devices, such as keyboard and display, or to another computer. The development board used in this project features a number of such devices and connections. Their signals are available by standard specifications is a so-called *configuration file* (.ucf), and they include clock and reset signals. They are available only in the *top module* of the constructed module hierarchy. We therefore present the TRM's top module first, and then discuss the implementations of other (service) modules. Starting with a transmitter and a receiver for a standard serial line, we obtain the simple module hierarchy shown below:

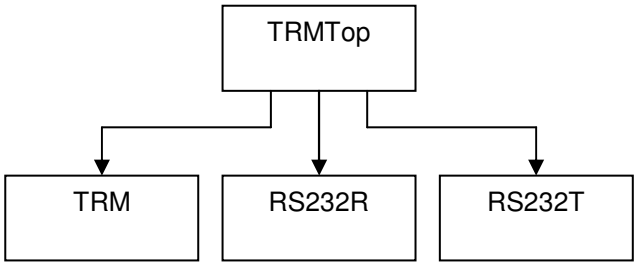


Fig. 2.1. Hierarchy of Verilog modules

The principal purpose of the top module is to connect signals of one module with signals of another module (or with external signals). This connecting occurs under control of the TRM, i.e. according to the TRM's interface signals *ioadr, iowr*, and *iord*. Hence, the main components to be found in the top module are multiplexers and decoders driven by *ioadr*. This is shown by the diagram, in which the boxes in the middle represent individual devices, which can be either implemented by other modules or (exceptionally) in the top module itself, as in the cases of dip switches and LEDs.

The I/O addresses driving the decoders and multiplexers in this top module are: .

adr	input	output	
4	data Rx	data Tx	RS-232
5	status	--	bit 0: RxRdy, bit 1: TxRdy
6	millisec timer	reset timer interrupt (tick)	
7	8 dip switches	10 LEDs	

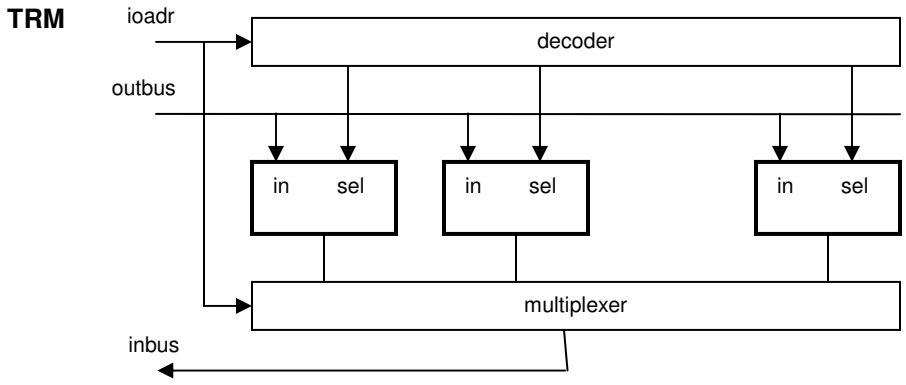


Fig. 2.2. Environment with I/O devices

In this sample top module one instance of each of TRM, FPU, RS232R, and RS232T are created (imported). Furthermore 8 dip switches are made available as inputs and 10 LEDs as outputs. They are represented as signals *swi* and *leds* in the top module's interface (heading). And so are the serial input *RxD* and output *TxD*. Signals *CLKBN* and *CLKBP* stem from an oscillator, and *rstIn* from a push button. We refrain from presenting the clock generation circuitry in detail, but emphasize that the entire *design is synchronous*, i.e. driven by the single clock *clk*.

Another feature of this top module is a timer (*cnt1*) counting elapsed milliseconds. It is driven by another counter (*cnt0*) which counts, according to the clock rate of 116.6 MHz, up to 116600 and then advances *cnt1* and sets the *tick* register to 1. The tick signal is fed to the TRM's *irq0* input, and thus may cause an interrupt every millisecond, if enabled.

Hint: The FPU can be deleted by simply dropping its instantiation.

```
module TRM3Top(
  input CLKBN,
  input CLKBP,
  input rstIn,
  input RxD,
  input [7:0] swi,
  output TxD,
  output [9:0] leds);

wire ClockIn;
wire PLLBfb;
wire pllLock;
wire clk, CLKx;
reg rst, tick;

wire[5:0] ioadr;
wire iord, iowr, stall, io4, io5, io6, io7, io16;
wire[31:0] inbus, outbus, fpubus;

wire [7:0] dataTx, dataRx;
wire rdyRx, doneRx, startTx, rdyTx;

reg [9:0] Lreg;  // for LEDs
```

```
reg [17:0] cnt0;  //driver of the millisecond counter
reg [31:0] cnt1;  // millisecond counter

TRM trmx(.clk(clk), .rst(rst), .stall(stall), .irq0(tick), .irq1(1'b0),
   .inbus(inbus), .ioadr(ioadr), .iord(iord), .iowr(iowr), .outbus(outbus));
FPU fpu(.clk(clk), .rst(rst), .stall(stall), .iowr(iowr & io16),
   .ioadr(ioadr[1:0]), .inbus(outbus), .outbus(fpubus));
RS232R receiver(.clk(clk), .rst(rst), .RxD(RxD), .done(doneRx), .data(dataRx), .rdy(rdyRx));
RS232T transmitter(.clk(clk), .rst(rst), .start(startTx), .data(dataTx), .TxD(TxD), .rdy(rdyTx));

assign io4 = (ioadr == 4);
assign io5 = (ioadr == 5);
assign io6 = (ioadr == 6);
assign io7 = (ioadr == 7);
assign io16 = (ioadr[5:2] == 4'b0100);

assign inbus = io4 ? {24'b0, dataRx} :
   io5 ? {30'b0, rdyTx, rdyRx} :
   io6 ? cnt1 :
   io7 ? swi : fpubus;
assign dataTx = outbus[7:0];
assign startTx = iowr & io4;
assign doneRx = iord & io4;
assign leds = Lreg;

always @(posedge clk)
  if (~rst) begin tick <= 0;cnt0 <= 0; Lreg <= 0; end
  else begin
    if (iowr & io6) tick <= 0;
    if (iowr & io7) Lreg <= outbus[9:0];
    else if (cnt0 == 116600) begin
      cnt1 <= cnt1 + 1; cnt0 <= 0; tick <= 1;end
    else cnt0 <= cnt0 + 1;
  end
 always @(posedge clk) rst <= rstIn & pllLock;
endmodule
```

A Transmitter for the RS-232 serial line

RS-232 is one of the oldest standards for data transmission between computers and devices. It is based on a single line, packets (bytes) as elements, asynchronous transmission of packets, and synchronous transmission within packets. There exist a number of standard bit rates. Here we use 115.2 Kb/s. A unit of transmission consists of 10 bits, a start bit, the 8 data bits, and a stop bit. The latter serves to keep a minimal delay between packets. RS-232 is primarily used for low-speed transmission. The standard is particularly useful for simple implementations. Our version uses a 116.6 MHz clock, which is divided by 1012 to obtain a bit rate of 115.2 Kb/s.

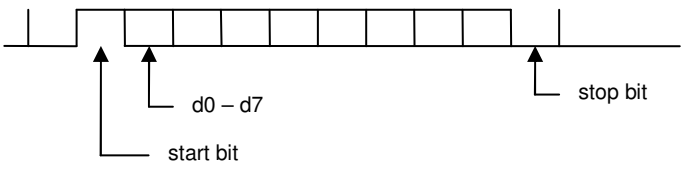


Fig. 2.3. The RS-232 data format

```
// RS232 transmitter for 115200 bps, 8 bit data, 1 stop bit
// clock is 116.6 MHz; 116600 / 1012 = 115.2 KHz

        module RS232T(
           input clk, rst,
           input start, // request to accept and send a byte
           input [7:0] data,
           output rdy,
           output TxD);

        wire endtick, endbit;
        reg run;
        reg [9:0] tick;
        reg [3:0] bitcnt;
        reg [9:0] shreg;

        assign endtick = tick == 1012;
        assign endbit = bitcnt == 9;
        assign rdy = ~start & ~run;
        assign TxD = ~shreg[0];

        always @ (posedge clk) begin
           if (run & ~endtick) tick <= tick + 1;
             else tick <= 0;
           if (~run) bitcnt <= 0;
             else if (endtick & (bitcnt < 10)) bitcnt <= bitcnt + 1;
           if (~rst) run <= 0;
             else if (start & ~run & (bitcnt < 10)) run <= 1;
             else if (endtick & endbit) run <= 0;
           if (~rst) shreg <= 0;
           else if (start & ~run)
             begin shreg[0] <= 1'b1; shreg[8:1] <= ~data; end
           else if (run & endtick)
             begin shreg[8:0] <= shreg[9:1]; end
        end
        endmodule
```

A Receiver for the RS-232 serial line

The neutral state of the transmission line is “high”. The receiver’s state machine is triggered whenever the input *RxD* becomes low. The line is sampled in the middle of a “bit-cell” (midtick). in order to minimize the chance of reading an incorrect value.

```
        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 [8:0] tick;
reg [3:0] bitcnt;
reg [7:0] shreg;

assign endtick = tick == 1012;
assign midtick = tick == 506;
assign endbit = bitcnt == 8;
assign data = ~shreg;
assign rdy = stat;

always @ (posedge clk) begin
  if (~rst) begin stat <= 0; run <= 0; end
  else begin
    if (run & ~endtick) tick <= tick + 1;
    else tick <= 0;
    if (~run) bitcnt <= 0;
    else if (endtick & ~endbit) bitcnt <= bitcnt + 1;
    else if (endtick & endbit) bitcnt <= 0;
    if (~RxD) run <= 1;
    else if (endbit & endtick) run <= 0;
    if (run & midtick) begin
      shreg[6:0] <= shreg[7:1]; shreg[7] <= ~RxD; end
    if (endbit & endtick) stat <= 1;
    else if (done) stat <= 0;
  end
end
endmodule
```

A Floating-point Unit

Scientific computation is almost without exception based on floating-point arithmetic. Fractional numbers (type REAL) are represented by a pair mantissa-exponent, i.e.

$$x = m \times B^{e} \qquad 1.0 \le m < B$$

where B is a fixed base. The universally adopted, *single-precision IEEE Standard* defines B = 2 and

$$x = \langle s, m', e' \rangle \qquad m = 1.m',\ e = e' - 127, \text{ and } 1.0 \le m < 2.0$$

with a sign bit *s*, an exponent *e'* of 8 bits, and a mantissa *m'* of 23 bits. The leading 1 of *m* is suppressed. A few examples of real numbers and their representation in hexadecimal form are:

x	e	m	
0.5	-1	1.0	3F000000
1.0	0	1.0	3F800000
1.5	0	1.5	3FC00000
1.75	0	1.75	3FE00000
2.0	1	1.0	40000000
10.0	3	1.25	41200000
100.0	6	1.5625	42C80000

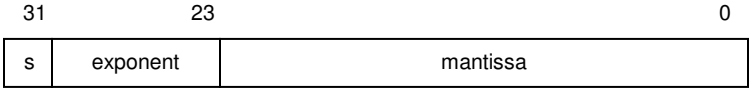


Fig. 2.3. IEEE Floating-point number format

In this logarithmic representation the value 0 is inherently not representable and must be treated as a special case. The Standard defines it to be represented by all zero bits. Note that the representation is zero-symmetric, i.e. x and –x are distinguished only by the sign bit. Also values with e' = 255 are considered as special. They represent non-representable values (overflow). The largest absolute value is 2^{128} (about 10^{38}) and the least value is 2^{-128} (about 10^{-38}).

Floating-point **multiplication** is easily explained. It consists of the multiplication of the mantissas and the addition of exponents:

$$x0 * x1 = 2^{e0+e1-127} * (m0 * m1)$$

The product of the mantissas will be in the range $1.0 \leq m0*m1 < 4.0$. Therefore, a *postnormalization* is mandatory to reach the normal form $1.0 \leq n < 2.0$. It is achieved by a right shift of the product mantissa and an increment of the exponent by 1, if $x \geq 2.0$.

The cases of any of the operands being zero must be detected as special case resulting in a zero product.

It is inherent in computing with rael numbers that the results may be slightly inaccurate. It is essential that the reults are rounded properly. This is done by adding 0.5 at the low order position, or a 1 at the position that will be truncated. This complexifies the circuitry, but it is necessary to achieve a usable arithmetic.

Floating-point **addition** (and subtraction) are more complicated. Before performing an additionof the mantissas, the exponents must be made equal. This is possible only by dividing (the mantissa of) the operand with the smaller exponent by 2^n, where *n* is the difference of the two exponents. This operation is called denormalization. Division by 2^n is of course achieved by shifting the mantissa to the right by *n* bits:

m := SHR(n, n); e := e+n denormalization

After the addition, the sum must be brought to its absolue value, and thereafter to its normalized form. This means that leading zeroes must be eliminated and the exponent decreased accordingly. This can be done sequentially by testing the leading bit ans left shifting the mantissa by one bit, until the leading bit is 1. A faster solution is to determine the number *n* of leading zeroes, and then left shifting the mantissa by *n* bits. The cost of this solution is increased complexity of the circuit.

We point out that all this is achieved by a purely sequential circuit with no registers involved. However, a path length may result that causes a signal propagation delay larger that a single clock cycle. In this case, registers must be introduced.

Floating-point **division** is not discussed here. It is a rather rare operation, too rare to warrant a complex circuit. It is better implemented by an iterative method in software. The following algorithm computes $x = 1/a$.

```
x := 1.0; z := 1.0 - a;
REPEAT x := x*(1.0+z); z := z*z UNTIL z = 0.0
```

Implementation of floating-point operations

It is generally desirable to provide floating-point operations by processor instructions. We instead chose to consider the FPU as an external divice, i.e. not as part of the basic processor. The advantage is a modular decoupling of the basic processor from the floating-point unit. The FPU's interface therefore contains registers for the arguments *x* and *y*, and output ports for sum and product. Storing y also causes the respective operation to be triggered.

adr	input	operation	output
16	x	store x	sum
17	y	store y, addition	product
18	y	store -y, addition	
19	y	store y, multiplication	

The following instruction sequences are used and generated by the compiler for the statements *z := x+y; z := x-y; z := x*y.* Assume that R0 contains operand x, R1 operand y, and R2 the device address of the floating-point unit.

```
ST   R0   R2   0      x
ST   R1   R2   1      y
LD   R0   R2   0      R0 := x + y
ST   R0   R7   2      z := R0

ST   R0   R2   0      x
ST   R1   R2   2      -y
LD   R0   R2   0      R0 := x + (-y)
ST   R0   R7   2      z := R0

ST   R0   R2   0      x
ST   R1   R2   3      y
LD   R0   R2   1      R0 := x*y
ST   R0   R7   2      z := R0
```

```
module FPU(
  input clk, rst, iowr,
  input [1:0] ioadr,
  input [31:0] inbus,
  output stall,
  output [31:0] outbus);

reg [31:0] X, Y;   // arguments
reg [26:0] s;  // pipe reg
reg mulR;

wire io0, io1, io2, io3;
wire [27:0] x0, y0;
```

```
wire [36:0] x1, y1;
wire [40:0] x2, y2;
wire [26:0] x3, y3;
wire [7:0] xe, ye;
wire [8:0] dx, dy, e0, e1;
wire [7:0] sx, sy;  // shift counts
wire [1:0] sx0, sx1, sy0, sy1;
wire sxh, syh;
wire [26:0] ss;
wire [31:0] Sum;

reg [31:0] prodReg; // product buffer
wire mul, sign, startM, stallM;
wire [8:0] e2, e3;
wire [69:0] mulRes;
wire [24:0] p0, p1;
wire [31:0] Prod;

wire z24, z22, z20, z18, z16, z14, z12, z10, z8, z6, z4, z2;
wire [4:0] u;  // shift counts
wire [1:0] u0, u1;
wire [41:0] t0, t1, t2, t3;
wire [24:0] t4;

assign io0 = (ioadr == 0);  // address assignments
assign io1 = (ioadr == 1);
assign io2 = (ioadr == 2);
assign io3 = (ioadr == 3);

always @ (posedge(clk))
  if (~rst) begin X <= 0; Y <= 0; end
  else begin
    if (iowr & io0) X <= inbus;
    if (iowr & io1) Y <= inbus;
    if (iowr & io2) Y <= {~inbus[31], inbus[30:0]};
    if (iowr & io3) Y <= inbus;
  end

assign xe = X[30:23];  // addition  denormalization
assign ye = Y[30:23];
assign dx = xe - ye;
assign dy = ye - xe;
assign e0 = (dx[8]) ? ye : xe;

assign sx = dy[8] ? 0 : dy;
assign sy = dx[8] ? 0 : dx;
assign sx0 = sx[1:0];
assign sx1 = sx[3:2];
assign sy0 = sy[1:0];
assign sy1 = sy[3:2];
assign sxh = sx[7] | sx[6] | sx[5];
assign syh = sy[7] | sy[6] | sy[5];

assign x0 = {4'b0001, X[22:0], 1'b0};  // guard digit
assign y0 = {4'b0001, Y[22:0], 1'b0};

genvar i;
generate  // denormalize, shift right
  for (i = 0; i < 25; i = i+1)
```

```
  begin: shiftblk0
    assign x1[i] = (sx0 == 3) ? x0[i+3] : (sx0 == 2) ? x0[i+2] : (sx0 == 1) ? x0[i+1] : x0[i];
    assign y1[i] = (sy0 == 3) ? y0[i+3] : (sy0 == 2) ? y0[i+2] : (sy0 == 1) ? y0[i+1] : y0[i];
  end
  for (i = 0; i < 25; i = i+1)
  begin: shiftblk1
    assign x2[i] = (sx1 == 3) ? x1[i+12] : (sx1 == 2) ? x1[i+8] : (sx1 == 1) ? x1[i+4] : x1[i];
    assign y2[i] = (sy1 == 3) ? y1[i+12] : (sy1 == 2) ? y1[i+8] : (sy1 == 1) ? y1[i+4] : y1[i];
  end
  for (i = 0; i < 25; i = i+1)
  begin: shiftblk2
    assign x3[i] = sxh ? 0 : (sx[4]) ? x2[i+16] : x2[i];
    assign y3[i] = syh ? 0 : (sy[4]) ? y2[i+16] : y2[i];
  end
endgenerate

assign ss = (X[31] ? -x3 : x3) + (Y[31] ? -y3 : y3);  // add or subtract
always @ (posedge(clk)) s <= ss[26] ? -ss : ss;

assign z24 = ~s[25] & ~ s[24];
assign z22 = z24 & ~s[23] & ~s[22];
assign z20 = z22 & ~s[21] & ~s[20];
assign z18 = z20 & ~s[19] & ~s[18];
assign z16 = z18 & ~s[17] & ~s[16];
assign z14 = z16 & ~s[15] & ~s[14];
assign z12 = z14 & ~s[13] & ~s[12];
assign z10 = z12 & ~s[11] & ~s[10];
assign z8 = z10 & ~s[9] & ~s[8];
assign z6 = z8 & ~s[7] & ~s[6];
assign z4 = z6 & ~s[5] & ~s[4];
assign z2 = z4 & ~s[3] & ~s[2];

assign u[4] = z10;  // u = shift count of post normalization
assign u[3] = z18 & (s[17] | s[16] | s[15] | s[14] | s[13] | s[12] | s[11] | s[10])
     | z2;
assign u[2] = z22 & (s[21] | s[20] | s[19] | s[18])
     | z14 & (s[13] | s[12] | s[11] | s[10])
     | z6 & (s[5] | s[4] | s[3] | s[2]);
assign u[1] = z24 & (s[23] | s[22])
     | z20 & (s[19] | s[18])
     | z16 & (s[15] | s[14])
     | z12 & (s[11] | s[10])
     | z8 & (s[7] | s[6])
     | z4 & (s[3] | s[2]);
assign u[0] = ~s[25] & s[24]
     | z24 & ~s[23] & s[22]
     | z22 & ~s[21] & s[20]
     | z20 & ~s[19] & s[18]
     | z18 & ~s[17] & s[16]
     | z16 & ~s[15] & s[14]
     | z14 & ~s[13] & s[12]
     | z12 & ~s[11] & s[10]
     | z10 & ~s[9] & s[8]
     | z8 & ~s[7] & s[6]
     | z6 & ~s[5] & s[4]
     | z4 & ~s[3] & s[2];

assign e1 = e0 - u + 1;
```

```
assign u0 = u[1:0];  // u = shift count
assign u1 = u[3:2];
assign t0 = {s[25:0], 16'b0};

generate // normalize, shift left
  for (i = 16; i < 42; i = i+1)
  begin: shiftblk4
    assign t1[i] = (u0 == 3) ? t0[i-3] : (u0 == 2) ? t0[i-2] : (u0 == 1) ? t0[i-1] : t0[i];
  end
  for (i = 16; i < 42; i = i+1)
  begin: shiftblk5
    assign t2[i] = (u1 == 3) ? t1[i-12] : (u1 == 2) ? t1[i-8] : (u1 == 1) ? t1[i-4] : t1[i];
  end
  for (i = 16; i < 42; i = i+1)
  begin: shiftblk6
    assign t3[i] = u[4] ? t2[i-16] : t2[i];
  end
endgenerate

assign t4 = t3[41:17] + 1;  // rounding
assign Sum = (xe == 0) ? Y : (ye == 0) ? X :
    ((t3[41:17] == 0) | e1[8]) ? 0 : {ss[26], e1[7:0], t4[23:1]};


MulDSP mulUnit(.CLK (clk),   // multiplication
   .mul (mul & ~startM),
   .A ({11'b0, 1'b1, X[22:0]}),
   .B ({11'b0, 1'b1, Y[22:0]}),
   .stall (stallM),
   .mulRes (mulRes));

assign mul = iowr & io3;
assign startM = mul & ~mulR;
assign stall = startM | stallM;
assign sign = X[31] ^ Y[31];
assign e2 = X[30:23] + Y[30:23];
assign e3 = mulRes[47] ? e2 - 126 : e2 - 127;
assign p0 = {1'b0, mulRes[47] ? mulRes[46:23] : mulRes[45:22]};
assign p1 = p0 + 1;  // rounding

assign Prod = (xe == 0) | (ye == 0) ? 0 :
    (~e3[8]) ? {sign, e3[7:0] + {7'b0, p1[24]}, p1[23:1]} :
    (~e3[7]) ? {sign, 8'b11111111, 23'b0} : 0;  // overflow

always @ (posedge(clk)) begin mulR <= mul;
  if (mul) prodReg <= Prod;
end

assign outbus = (io0) ? Sum : prodReg;
endmodule
```

It is remarkable that the program of the FPU is almost as long as that for the entire processor TRM. It is therefore of interest to compare its performance with that of a solution implementing real arithmetic by software. The result of a comparison indicates that the hardware solution performs between 10 and 30 times faster than the software implementation. The extreme case is that of subtraction with almost identical operands. This leads to a long post-normalization shift, which is done in a loop in software. This case is a weak point

of floating-point arithmetc in general. It implies a loss of precision and is called *cancellation*.

Experiments in Computer System Design

Niklaus Wirth

PART 3

About memories

In the early years of computers, memories had been considered as an integral part of the central computing unit. This remained so through the eras of magnetic drum memories, magnetic core memories, and static semiconductor memories (SRAM). A change came with the RISCs (Reduced Instruction Set Computer), which more strongly decouple memory and processing unit. Whereas the speed of processors increased dramatically, the speed of memories also increased, but at a lesser pace. But their capacity increased substantially, mainly due to *dynamic random access memories* (DRAM). Cells in static RAMs consist of two transistors and have 2 stable states. Thus they hold a bit (until given a new value), and therefore they are called *static*. The *dynamic* RAM holds a bit in a small capacitor coupled with a single transistor. This cell requires less space on a die and therefore is dominant for large capacity devices.

The DRAM has, however, a few drawbacks. The most prominent is that capacitors leak and discharge through the transistor. Therefore the charge must be refreshed. This is achieved by reading the cell and restoring the old value (through recharge). Refreshing requires additional circuitry, which must not interfere with normal data access. DRAMs are typically refreshed at least every millisecond.

Memory chips of the latest provenience have capacities in the order of a gigabyte and therefore require large multiplexers for reading and decoders for writing. As a consequence, access is slower than for smaller devices. In the last decades, the speeds of processors and of memories have increasingly diverged. Two remedies are in use: 1. Data in memory are accessed in larger portions than single words or bytes. 2. Buffers are placed in the data path between memory and processor. These buffers are fast memories, called *caches*. Modern processors feature cache memories on-chip. Naturally, caches further complicate memory access, leading to more complex circuit. It is common that such cache mechanisms are to be invisible (transparent) to the computer user and to the software. We will here first show how a large DDR memory is interfaced with the TRM.

A DDR memory as an external device

Let us connect a DDR memory to the TRM's input/output bus. The memory in question here is a 256 MB chip MT4HTF3264HY-53E of Micron, present on the ML-505 evaluation board. In fact, we will not connect the memory directly to the TRM, but place an intermediary agent in between. It is called a *DDR Controller,*

and it was designed by Ch. Thacker of Microsoft Research in Mountain View. Thereby we obtain some freedom to ignore details of this particular type of DDR.

In addition to being periodically refreshed, the DDR memory must be initialized at startup. This involves the loading of certain constants. Also, once the RAMs have been configured, the individual delay lines associated with the FPGA data pins must be adjusted to center the strobe in the "data valid" window.

These complicated task appear to require a substantial amount of circuitry. This can be avoided by employing a dedicated, simple, programmed processor for these tasks. The design of such a processor is described below. It is called TRM-0. Once the system is running, the TRM-0 controls the periodic refresh of the RAMs. Note that calibration can fail. The signal *CalFailed* is available to programs as a status bit of the DDR interface. The TRM-0 will be presented at the end of this Part.

Let us now describe the top module that connects to the DDR-Controller as an external device of TRM. We start by showing the heading (interface) of this Top module. In addition to the signals of the top module described in Part-1 of this Report, it contains all signals leading to the memory chip on the ML-505 board. They are directly passed on to the DDR-Controller module.

```
module TRM3DTop(
  input CLKBN,
  input CLKBP,
  input rstIn,
  input RxD,
  input [7:0] swi,
  output TxD,
  output [9:0] leds,

  inout [63:0] DQ, //the 64 DQ pins, signals to the memory chip
  inout [7:0] DQS, //the 8  DQS pins
  inout [7:0] DQS_L,
  output [1:0] DIMMCK,  //differential clock to the DIMM
  output [1:0] DIMMCKL,
  output [12:0] A, //addresses to DIMMs
  output [7:0] DM,
  output [1:0] BA, //bank address to DIMMs
  output RAS, CAS, WE, ODT, ClkEn, S0);
```

The connections between the various modules are best sketched by the following block diagram: The registers and signals, in addition to those present in the basic version of TRM3Top are:

```
reg Read, Write;  // DDR commands
reg RBempty1, Write1;  // delayed DDR signals
reg RDrdy, shiftRD;
reg [22:0] Address;
reg [255:0] RD;  // read data buffer from DDR
reg [255:0] WD;  // write data buffer to DDR

wire AFfull, WBfull, RBempty, WriteAF, ReadRB, WriteWB;
wire [127:0] ReadData, WriteData;
```

Data are read and written in blocks of 256 bits (8 words). The memory can be considered as consisting of 256-bit elements. When writing, first 8 words are deposited into the WD buffer by 8 consecutive IO commands (with address 10).

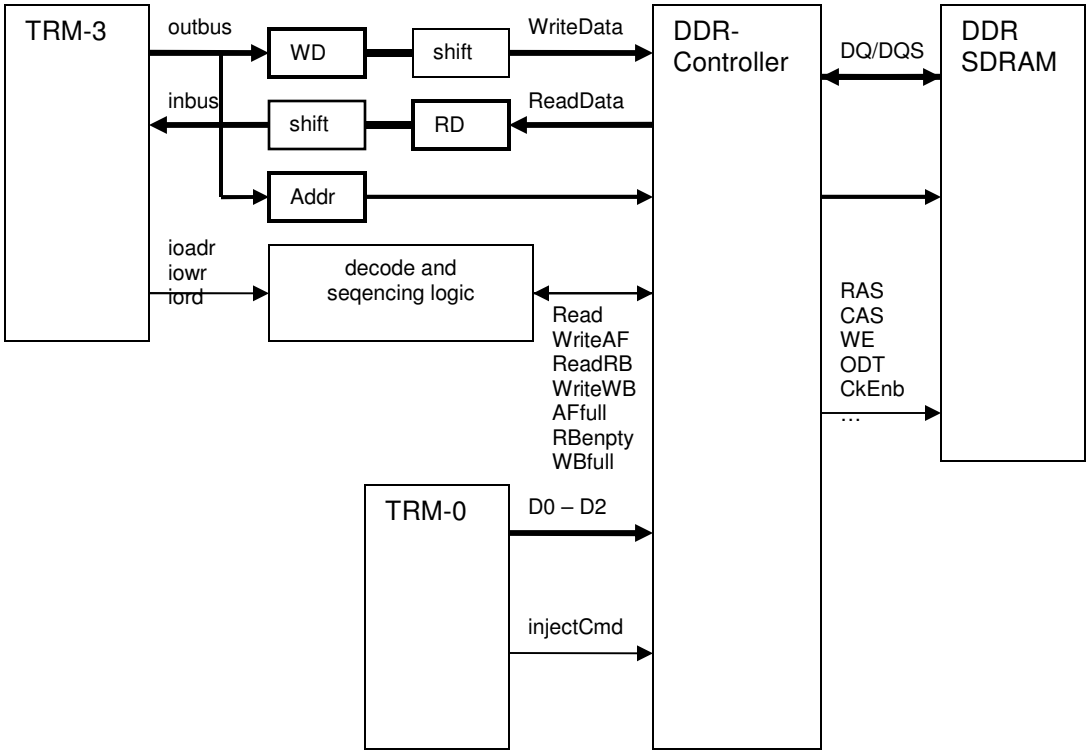


Fig. 3.1. TRM-DDR interface in TRM3DTop

The DDR2 memory used here has a capacity of 256 MByte, Each “word” consists of 32 bytes, which results in a word address of 23 bits. Data are loaded into a 256-bit shift register. Each command shifts down the data by 32 bits and places the *iobus* data into the high end of the shift register. A subsequent command with address 11 initiates the DDR reading. It supplies the DDR address. This occurs in a single cycle with 128 bits transferred on the rising, and 128 bits of the falling edge of the clock.

Reading starts with a DDR-read command (address 11), supplying the DDR-address of the 256-bit block. Reading also is done in two bursts at the rising and falling edges of the clock. The data are deposited in the RD buffer. Then follow 8 consecutive read commands (address 10), each moving a word from the low end of the RD buffer to the TRM’s inbus and shifting the data in the buffer down.

The operations of writing and reading a block are best described by the following procedures in Oberon. Bit 0 of the status register means “read buffer not empty”, bit 1 mena “write buffer full”, and bit 2 means “command buffer full”.:

```
CONST A0 = 0FFFFFFCAH; A1 = 0FFFFFFCBH;
TYPE Block = ARRAY 8 OF INTEGER;

PROCEDURE Write(dst: INTEGER; VAR B: Block);
    VAR i: INTEGER;
BEGIN i := 0;
    REPEAT UNTIL ~BIT(A1, 1);   (*write buffer not full*)
    REPEAT PUT(A0, B[i]); INC(i) UNTIL i = 8;
    REPEAT UNTIL ~BIT(A1, 2);   (*command buffer not full?*)
    PUT(A1, 2000000H + dst);  (*write DDR*)
END Write;

PROCEDURE Read(src: INTEGER; VAR B: Block);
    VAR i: INTEGER;
BEGIN i := 0;
    REPEAT UNTIL ~BIT(A1, 2);
    PUT(A1, 1000000H+ src);   (*read DDR*)
    REPEAT UNTIL ~BIT(A1, 0);
    REPEAT UNTIL BIT(A1, 0);
    REPEAT GET(A0, B[i]); INC(i) UNTIL i = 8;
END Read;
```

The details of the implementation in TRM3DTop are given by the following statements in Verilog (see also declarations above):

```
assign Reset = ~rstIn  | ~pllLock | ~ctrlLock;
assign WriteAF = Read | Write1;
assign WriteWB = Write | Write1;  // commands to DDR controller
assign ReadRB = ~RBempty;
assign WriteData = (Write1) ? WD[127:0] : WD[255:128];

always @(posedge clk) rst <= rstIn & pllLock & DDRCalSuccess;

// writing DDR:  outbus(32) --> WD(256) --> WriteData(128)
// reading DDR:  ReadData(128) --> RD(256) --> inbus(32)

always @(posedge clk)
  begin
    if (io11 & iowr) begin  // DDR command
      Address <= outbus[22:0];
      Read <= outbus[24]; RDrdy <= 0;
      Write <= outbus[25];
    end
    else begin Read <= 0; Write <= 0;
      if (~RBempty1 & RBempty) RDrdy <= 1;
    end

    if (io10 & iowr) begin  // write a word to TRM
      WD[223:0] <= WD[255:32]; WD[255:224] <= outbus;
    end

    if (io10 & iord) shiftRD <= ~shiftRD; else shiftRD <= 0;
    Write1 <= Write; RBempty1 <= RBempty;
    if (shiftRD) RD[223:0] <= RD[255:32]; else
    if (~RBempty) begin RD[255:128] <= RD[127:0]; RD[127:0] <= ReadData; end
  end
```

Introducing a Direct Memory Access Channel (DMA)

The solution presented above is the simplest, as far as hardware is concerned. Its drawback is, however, low speed. This can be remedied by avoiding the use of one instruction for each word transferred, and instead to transfer an entire block through a single instruction. This solution introduces an important concept of computer architecture: The *direct memory access channel*. It postulates that not only the processor, but also other agents may obtain direct memory access.

The two driver procedures are then simplified to:

```
PROCEDURE Write(dst: INTEGER; VAR B: Block);
BEGIN
    REPEAT UNTIL ~BIT(A1, 1);     (*write buffer not full*);
    PUT(A0, ADR(B) + 2000000H); (*DMA transfer of 8 words from B*)
    REPEAT UNTIL ~BIT(A1, 2);     (*command buffer not full?*)
    PUT(A1, 2000000H + dst);      (*write DDR*)
END Write;

PROCEDURE Read(src: INTEGER; VAR B: Block);
BEGIN
    REPEAT UNTIL ~BIT(A1, 2);     (*command buffer not full*)
    PUT(A1, 1000000H+ src);       (*read DDR*)
    REPEAT UNTIL ~BIT(A1, 0);
    REPEAT UNTIL BIT(A1, 0);
    PUT(A0, ADR(B) + 1000000H); (*DMA transfer of 8 words to B*)
END Read;
```

The result is remarkable: The speed of transferring blocks has tripled. What are the consequences for the hardware interface? Evidently, the TRM itself must be modified by adding external access signals to its data memory. This implies that its *interface must change* Furthermore, a data transfer lasts over several clock cycles, and therefore requires a *state machine* to control it. Furthermore, it must be possible to *stall* the processor, i.e. to prevent it from proceeding to the next instruction.

We will let the state machine to be part of the device interface rather than of the TRM processor. This minimizes the changes to the interface, which we will describe first. It is extended with 5 signals:

```
module TRM3X(
input clk, rst, stall,
input irq0, irq1,
input[31:0] inbus,
output [5:0] ioadr,
output iord, iowr,
output [31:0] outbus,

input dmaenb, dmawr,  // dma connections
input [11:0] dmaAdr,
input[31:0] dmain,
output [31:0] dmaout);
```

In the TRM itself, all inputs signals to the local memory obtain a multiplexer with the existing input plus the one from the interface:

```
assign dmadr = (dmaenb) ? dmaAdr : ((irs == 7) ? 0 : AA[11:0]) + off);
assign dmwr = (dmaenb) ? dmawr : ST & ~ioenb;
assign dmin = (dmaenb) ? dmain : B;
assign dmaout = dmout;
```

The only further change to the TRM logic is the *dmaenb* signal stalling the processor:

```
assign stall0 = (LDR & ~stall1) | stallM | stallD | stall | dmaenb;
```

The major addition to the top module is the state machine. It controls the block transfer between the TRM output to the write buffer WD, and the block transfer between the read buffer RD and the TRM input. The state machine is triggered by a PUT statement with I/O address 10. The command word contains the address of local memory in bits 0 – 10, and either a read in bit 24 or a write in bit 25.

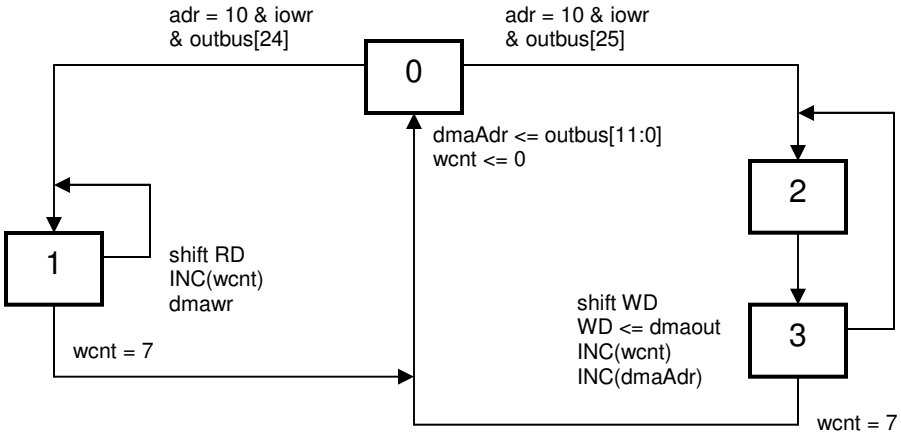


Fig. 3.2. DMA state machine

When reading the DDR memory, the local memory is written, which takes a single clock cycle. When writing DDR memory, the local memory is read, which takes 2 clock cycles. This is reflected by the state machine having 2 states in the write branch. The word counter, running from 0 to 7, is kept separate from the state. The signal and register declarations are as follows:

```
reg Read, Write;  // DDR commands
reg RBempty1, Write1;  // delayed DDR signals
reg RDrdy;   // data read from DDR ready

reg [1:0] state;  // dma state (0 = idle)
reg [2:0] wcnt;  // word count for DMA
reg [11:0] dmaAdr;
reg [22:0] ddradr;
reg [255:0] RB;  // input buffer from DDR
reg [255:0] WB;  // output buffer to DDR

wire WriteAF, ReadRB, WriteWB;
wire AFfull, WBfull, RBempty;
wire dmaenb, dmawr;
wire [31:0] dmain, dmaout;
```

```
wire [127:0] ReadData, WriteData;  // data from/to DDR

assign WriteAF = Read | Write1;
assign WriteWB = Write | Write1;  // commands to DDR controller
assign ReadRB = ~RBempty;
assign WriteData = (Write1) ? WB[127:0] : WB[255:128];
assign dmaenb = ~(state == 0);
assign dmawr = (state == 1);
assign dmain = RB[31:0];
```

The state machine is expressed in Verilog by the following statements:

```
always @(posedge clk)
  begin
    if ((ioadr == 11) & iowr) begin  // DDR command
      ddradr <= outbus[22:0];
      Read <= outbus[24]; RDrdy <= 0;
      Write <= outbus[25];
    end
    else begin Read <= 0; Write <= 0;
      if (~RBempty1 & RBempty) RDrdy <= 1;
    end

    if (~rst) state <= 0;
    else if ((ioadr == 10) & iowr) begin  // DMA command
      dmaAdr <= outbus[11:0]; wcnt <= 0;
      state <= outbus[25:24]; end // 01: read RB, 10: write WB
    else begin
      if (state == 2) state <= 3;
      if (state == 3) begin
        WB[223:0] <= WB[255:32]; WB[255:224] <= dmaout;
        wcnt <= wcnt + 1; dmaAdr <= dmaAdr + 1;
        state <= (wcnt == 7) ? 0 : 2;
      end
      if (state == 1) begin
        RB[223:0] <= RB[255:32];
        wcnt <= wcnt + 1; dmaAdr <= dmaAdr + 1;
        state <= (wcnt == 7) ? 0 : 1;
      end
    end
    Write1 <= Write; RBempty1 <= RBempty;
    if (~RBempty) begin RB[255:128] <= RB[127:0]; RB[127:0] <= ReadData; end
  end
```

This concludes the addition of a DMA facility to the TRM top module. It causes only a moderate increase of complexity and results in a very substantial gain in performance. Note that this addition was (almost entirely) an addition to the environment: The DMA facility is considered part of the device.

Initializing and refreshing the SDRAM memory

DRAMs must be initialized and periodically refreshed. An economical way of doing this is to dedicate a small processor to this task. Here we first describe the program used, and then the processor, TRM-0. The program is described in pseudo-language, suppressing details. It was actually implemented by a very simple assembler code.

```
        PROCEDURE refresh;
        BEGIN prechargeall; wait(1); (*unit of delay = 32ns*) refreshall; wait(3);
        END refresh;

Start: inhibit DDR; wait(6000); toggleDDR; setDIMMclk;
InitMem: wait(12); prechargeall; wait(1);
        babk2; bank3; babk1; MRS1;
        wait(49);  (*wait for DLL to lock*)  refresh; refresh;
        MRS2; MRS3; MRS4; wait(11); prechargeall; wait(1);
Calibrate: inhibitDDR; set Force;
        wait(1); … ; wait(3); WriteCmd; toggle(StartDQcal); n := 0;
        REPEAT ReadCmd; DEC(n) UNTIL n = 0;
        wait(16); refresh; enableDDR; clear Force;
        REPEAT wait(768); disbleDDR; refresh; enableDDR END
```

4 of the 8 registers are directly connected to the DDRcontroller, providing commands (D0, D1) and status (D2); When assigning a value to register D1, a strobe is issued (injectCmd), causing the DDRcontroller to insert the command in D0, D1 in the command sequence supplied by the interface of the TRM-3. The status bits of D2 are:

0	StartDQcal
1	inhibit DDR
2	DDR clock enable
3	reset DDR
7	Force calibration

The instantiation of TRM-0 in the Top module is:

```
TRM0 trm0x (.clk(clk), .rst(rst), .trig(injectCmd),
  .Din(12'b0),
  .D0(D0), .D1(D1), .D2(D2), .D3(D3));
```

TRM-0: Architecture and instruction set

The TRM-0 processor consists of a compute unit consisting of an ALU, implementing addition and subtraction, and the basic logic operations used for setting and clearing individual bits, and of a set of 8 12-bit registers. It has a 2K program memory, but no data memory.

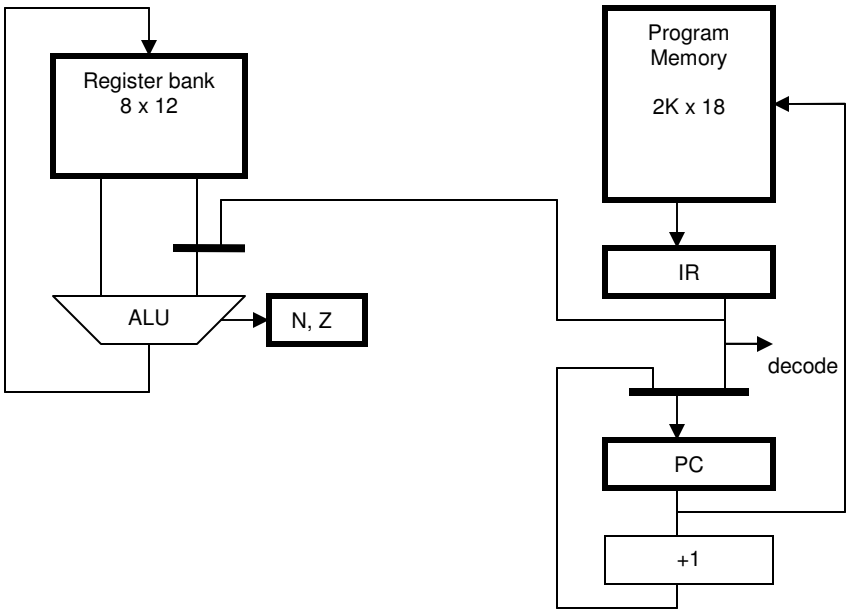


Fig. 3.3. TRM-0 Block Diagram

Register operations

op	operation	
0	BIS	R.d := R.d | R.s (in place of R.s may stand the literal n)
1	BIC	R.d := R.d & ~R.s (bit clear)
2	ADD	R.d := R.d + R.s
3	SUB	R.d := R.d - R.s
4	MOV	R.d := R.s
5	not used	

If s = 7, Din instead of R.7 is used as source

Branch instructions

op	cond	operation
6	CL	R.d := PC+1; PC := n

7	0	BEQ	PC := n, if Z
7	1	BNE	PC := n, if ~Z
7	2	BLT	PC := n, if N
7	3	BGE	PC := n, if ~N
7	4	BLE	PC := n, if N|Z
7	5	BGT	PC := n, if ~(N|Z)
7	6	B	PC := n
7	7	NOP	

The implementation of TRM-0

The implementation of TRM-0 is described by the following Verilog program, which somewhat resembles that of TRM-3, but is very substantially simpler.

```
module TRM0(
 input clk, rst,
 output trig,
 input [11:0] Din,
 output [11:0] D0, D1, D2, D3);

reg N, Z, T;
reg [10:0] PC;
reg [11:0] R0, R1, R2, R3, R4, R5, R6, R7;

wire [17:0] IR;  // register contained in module pbram
wire [35:0] pmout;
wire [2:0] op, dst, src, cond;
wire [1:0] cc;
wire [10:0] pcmux, nxpc;

wire [11:0] A, B, AluRes;

dpbram36 im(  // program memory, 1K x 36
 .wda(36'b0), // port A is the write port.
 .aa(10'b0),
 .wea(1'b0),
 .ena(1'b0),
 .clka(1'b0),
 .rdb(pmout), // port B is the read port.
 .wdb(36'b0),
 .ab(pcmux[10:1]),
 .web(1'b0),
 .enb(1'b1),
 .clkb(clk));

assign D0 = R4;
assign D1 = R5;
assign D2 = R6;
assign D3 = R7;
assign trig = T;

assign IR = PC[0] ? pmout[35:18] : pmout[17:0];
assign op = IR[17:15];
assign cc = IR[14:13];
assign dst = IR[14:12];
assign src = IR[2:0];
```

```
assign A = (~IR[11]) ? {IR[10], IR[10:0]} :
        (src == 0) ? R0 :
        (src == 1) ? R1 :
        (src == 2) ? R2 :
        (src == 3) ? R3 :
        (src == 4) ? R4 :
        (src == 5) ?          R5 :
        (src == 6) ? R6 : Din;

assign B = (dst == 0) ? R0 :
        (dst == 1) ? R1 :
        (dst == 2) ? R2 :
        (dst == 3) ? R3 :
        (dst == 4) ? R4 :
        (dst == 5) ?          R5 :
        (dst == 6) ? R6 : R7;

assign AluRes = (op == 0) ? B | A :
        (op == 1) ? B & ~A :
        (op == 2) ? B + A :
        (op == 3) ? B - A :
        (op == 4) ? A :
        (op == 5) ? A : nxpc;

assign cond = IR[12] ^ (
        (cc == 0) ? Z :
        (cc == 1) ? N :
        (cc == 2) ? Z|N : 1);

assign nxpc = PC + 1;
assign pcmux = ((op == 6) | (op == 7) & cond) ? A : nxpc;

always @ (posedge clk) begin
  if (~rst) begin PC <= 0; R6 <= 0; end
  else begin PC <= pcmux;
    if (op != 7) begin
      if (dst == 0) R0 <= AluRes;
      if (dst == 1) R1 <= AluRes;
      if (dst == 2) R2 <= AluRes;
      if (dst == 3) R3 <= AluRes;
      if (dst == 4) R4 <= AluRes;
      if (dst == 5) R5 <= AluRes;
      if (dst == 6) R6 <= AluRes;
      if (dst == 7) R7 <= AluRes;
      N <= AluRes[11];
      Z <= (AluRes == 0);
      T <= (dst == 5);
    end
  end
end
endmodule
```

Experiments in Computer System Design

Niklaus Wirth

PART 4

Multiprocessor-systems and interconnects

The FPGA is an ideal ground for experimenting with multi-processor configurations. On the chip of the ML-505 board, a configuration with 12 TRMs was successfully installed and tested. The limiting factor is the number of block RAMs (and DSPs) available. The interesting questions is how to interconnect the individual processors.

A frequently encountered model for systems is the *data flow* model. It is based on the premise that data flow on fixed channels from processor to processor. The key property of the data flow scheme as postulatd by Jack Dennis is that a processor takes action as soon as all necessary inputs are ready on the respective input channels. With this property, processor synchronization is implicit. Typical DF-Systems have been built whose nodes are essentially ALUs with buffers at the inputs. This model has not been successful in spite of several revivals over the past decades. A more promising model is the one where the nodes are autonomous, programmable processors. Evidently, the structure and the channels need to be custom-tailored to the application on hand. Here we merely present one kind of connection, actually the simplest connection possible, the *buffered channel*, also called *point-to-point connection*.

Point-to-point connection: The buffered channel

We assume that the elements of the sequence to be transmitted over a channel are 32-bit words. For this case, the interface can be particularly straight forward. Apart from the data, it contains the commands *wreq* and *rdreq* (for writing, sending, and reading, receiving respectively). Furthermore, there must be the status signals *empty* and *full*, indicating whether the write buffer is full, or the read buffer is empty, conditions where processing cannot proceed. In our first version, the buffer contains a single entry and is implemented as a register.

```
module Channel(
  input clk, rst,
  input wreq, rdreq,
  output empty, full,
  input [31:0] indata,
  output [31:0] outdata);

reg loaded;
reg [31:0] Buf;

assign outdata = Buf;
assign empty = ~loaded;
assign full = loaded;
```

```
always @ (posedge clk)
  if (~rst) loaded <= 0;
  else if (wreq) begin Buf <= indata; loaded <= 1; end
  else if (rdreq) loaded <= 0;
endmodule
```

If a higher degree of decoupling between the sending and the receiving nodes is required, a buffer with several slots must be provided. In the following example, 16 entries are provided. The buffer is implemented by 32 LUT slices with the macro RAM16X1D_1. The buffer is organized circularly; the counters are modulo 16 due to the fact that they consist of 4 bits.

```
module Channel6(
  input clk, rst,
  input wreq, rdreq,
  output empty, full,
  input [31:0] indata,
  output [31:0] outdata);

reg [3:0] in, out;  // buffer pointers

assign empty = (in == out);
assign full = (out == (in+1));
assign outdata = D;

genvar i;
generate    //dual port register file
  for (i = 0; i < 32; i = i+1)
  begin: rf32
  RAM16X1D_1 # (.INIT(16'h0000))
  rfa(
  .DPO(outdata),
  .SPO(),
  .A0(out[0]),   // R/W address, controls D and SPO
  .A1(out[1]),
  .A3(out[3]),
  .D(indata),  // data in
  .DPRA0(in[0]), // read-only adr, controls DPO
  .DPRA1(in[1]),
  .DPRA2(in[2]),
  .DPRA3(in[3]),
  .WCLK(~clk),
  .WE(wreq));
  end
endgenerate

always @(posedge clk)
  if (~rst) begin in <= 0; out <= 0; end
  else if (wreq) in <= in + 1;
  else if (rdreq) out <= out + 1;
endmodule
```

It is noteworthy and important that the interface of the two versions of channels are identical, and therefore easily interchangeable. The interfacing of such channels and TRMs occurs in the same way as that between RS-232 lines and TRM, as presented in Part 2 of this Report.

The ring structure

Point-to-point connections are less suitable in a multi-processor system, where no pairs of processors are a-priory known to communicate particularly frequently. In this case, a system is required that potentially connects every node with every other node. The traditional solution in this case is a bus. It inherently carries the problems of delays, of access priorities, and of bottlenecks. Also, since buses are usually implemented with tri-state gates, it is not easily practicable on FPGAs, as they do not contain tri-state gates.

The most general soultion is a crossbar switch, a martix of gates. Each row represents an input, each column an output. Crossbar switches are fast, but require many resources. FPGAs are not particularly suitable for their implementation, mostly because of the relative scarcity of long wires.

A likely alternative is the ring structure, where every processor is included as a ring node. The ring has technically the advantage that it consists of unidirectional, point-to-point connections only. It is therefore simple to implement and simple to operate. However, depending on the number of nodes lying between source and destination, there may be delays involved, Also, long messages may monopolize the ring, thus inducing longer waits for nodes also requesting access.

Nevertheless, we present here a basic implementation of a ring node as an example of how processors may be connected in a simple way on an FPGA chip.

Each node contains a register between the ring input and ring output. This register holds one data element and introduces a latency (delay of the data traveling through the ring) of a single clock cycle. The node also contains two buffers, one for the received data, and one for the data to be sent. Their purpose is to decouple the nodes in time and thereby to increase the efficiency of the connections. We postulate that the data are always sequences of bytes, and they are called *messages*.

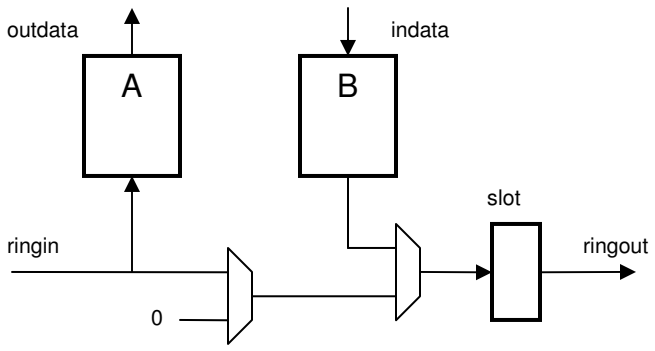


Fig. 4.1. Ring Node

The figure shows that if a node is sending a message over the ring, buffer B is fed to the ring output. If a message is received, the ring input is fed to buffer A. Otherwise the input is transmitted to the output, with a single cycle's delay.

We have chosen the elements of messages to be bytes. The ring and its registers, called *slots*, are 10 bits wide, 8 for data and 2 for a tag to distinguish between data and control bytes.

We postulate the following conventions: Messages are sequences of (tagged) bytes. The first element of a message is a *header*. It indicates the destination and the source number of the nodes engaged in the transmission. We assume a maximum of 16 nodes, resulting in 4-bit node numbers. The last element is the *trailer*. In between lie an arbitrary number of data bytes.

tag	data	
10	source, destination	header
00	xxxxxxxx	data byte
01	00000000	trailer
11	00000000	token

When no messages are to be transferred, the ring is said to be *idle*. When a node is ready to send a message, it must be granted permission in order to avoid collisions with messages sent by other nodes. One may imagine a central agency to rotate a pointer among the nodes, and the node so designated having the permission to send its message. As we wish to avoid a central agency, we instead insert a special element into the ring which takes over the role of the pointer. It is called the *token*. In the idle state, only the token is in the ring. Such a scheme is called a *token-ring*

Only a single message can be in the ring. When a node is ready to send a message, it waits until the token arrives, and then replaces the token by the message. The token is reinserted after the message. The header contains the number of the destination node which triggers the receiver to become active. When the message header arrives at the destination, that node feeds the message into its receiver buffer. It removes the header from the ring by replacing the slot with a zero data item.

The implementation of a token ring

A node of the ring is described as a Verilog module. The module interface consists of the ring input and ring output, and of the connections to the associated TRM processor.

```
module RingNode(clk, rst, wreq, rreq, ringin, ringout, indata, status, outdata);
input clk, rst wreq, rreq;
input [9:0] ringin;
input [9:0] indata;  // from processor
output [9:0] ringout;
output [7:0] status;
output [9:0] outdata;  // to processor

reg sending, receiving, rdyS;   // states
reg [5:0] inA, outA, inB, outB;   // buffer indices
reg [9:0] slot;    // element in ring
```

The buffers are implemented as LUT memories with 64 elements, 10 bits wide. Registers *inA* and *outA* are the 6-bit indices of the input buffer A, *inB*, *outB* those of the output buffer B.

There are 4 separate activities proceeding concurrently:

1. A byte is fed from *indata* to the output buffer B, and the pointer *inB* is advanced (incremented modulo buffer size). This is triggered by the input strobe *wreq* with *ioadr* = 2 (0FC2H).

2. A byte is transferred from buffer *B* to the ring, and pointer *outB* is advanced. This happens in the sending state, which is entered when the buffer contains a message and the token appears in the slot. The sending state is left when a trailer is transmitted.

3. A byte is transferred from the slot (ring input) to the input buffer *A* and the pointer *inA* is advanced. This is triggered by the slot containing a message header with the receiver's number. A zero is fed to the ring output.

4. A byte is transferred from buffer A to *outdata*. This happens when the TRM reads input. Thereafter the TRM must advance pointer *outA* by applying a *wreq* signal and *ioadr* = 1

The four concurrent activities are expressed in Verilog as shown below in one block clocked by the input signal *clk*. The input of buffer *A* is *ringin*, that of buffer *B* is *indata* (input from processor).

```
wire startsnd, startrec, stopfwd;
wire [9:0] A, B;   // buffer outputs

assign startsnd = ringin[9] & ringin[8] & rdyS;  //token here and ready to send
assign startrec = ringin[9] & ~ringin[8] & ((ringin[3:0] == mynum) | (ringin[3:0] == 15));
assign stopfwd = ringin[9] & ~ringin[8] & ((ringin[3:0] == mynum) | (ringin[7:4] == mynum));

assign outdata = A;
assign status = {mynum, sending, receiving, (inB == outB), (inA == outA)};
assign ringout = slot;

always @(posedge clk)
  if (~rst) begin  // reset and initialization
    sending <= 0; receiving <= 0; inA <= 0; outA <= 0; inB <= 0; outB <= 0;
    rdyS <= 0;
    if (mynum == 0) slot <= 10'b1100000000; else slot <= 0; end
  else begin
    slot <= (startsnd | sending) ? B : (stopfwd) ? 10'b0 : ringin;
    if (sending) begin // send data
      outB <= outB + 1;
      if (B[9] & B[8]) sending <= 0; end  // send token
    else if (startsnd) begin // token here, send header
      outB <= outB + 1; sending <= 1; rdyS <= 0; end

    else if (wreq) begin
      inB <= inB + 1; // msg element into sender buffer
      if (indata[9] & indata[8]) rdyS <= 1; end

    if (receiving) begin
      inA <= inA + 1;
      if (ringin[8]) receiving <= 0; end   // trailer: end of msg
```

```
  else if (startrec) begin // receive msg header
    inA <= inA + 1; receiving <= 1; end

  if (rreq) outA <= outA + 1;   // advancing the read pointer
end
```

A software driver

The pertinent driver software is described in Oberon. It is responsible for the maintenance of the prescribed protocol and message format, and it is therefore presented as a module. This module alone contains references to the hardware through procedures PUT, GET, and BIT. Clients are supposed not to access the hardware interface directly.

The module encapsulates and exports procedures *Send* and *Rec*, a predicate *Avail* indicating whether any input had been received, and a function *MyNum* yielding the node number.

```
MODULE Ring;

  CONST data = 0FC2H; stat = 0FC3H;  (*device register addresses*)

  PROCEDURE Avail*(): BOOLEAN;
    VAR status: SET;
  BEGIN GET(stat, status); RETURN ~(0 IN status)
  END Avail;

  PROCEDURE Send*(dst, typ, len: INTEGER; VAR data: ARRAY OF INTEGER);
    VAR i, k, w, header: INTEGER;
  BEGIN REPEAT UNTIL BIT(stat, 1);  (*buffer empty*)
    GET(stat, header); header := MSK(header, 0F0H) + MSK(dst, 0FH);
    PUT(data, header + 200H); PUT(data, MSK(typ, 0FFH)); i := 0;
    WHILE i < len DO
      w := data[i]; INC(i); k := 4;
      REPEAT PUT(data, MSK(w, 0FFH)); w := ROR(w, 8); DEC(k) UNTIL k = 0
    END ;
    PUT(data, 100H); PUT(0F02H, 300H)  (*trailer, token*)
  END Send;

  PROCEDURE Rec*(VAR src, typ, len: INTEGER; VAR data: ARRAY OF INTEGER);
    VAR i, k, d, w, header: INTEGER;
  BEGIN
    REPEAT UNTIL ~BIT(stat, 0);  (*buffer not empty*)
    GET(data, header); src := MSK(ROR(header, 4), 0FH);
    GET(data, typ); GET(data, d);
    i := 0; k := 4; w := 0;
    WHILE MSK(d, 300H) = 0 DO
      w := ROR(MSK(d, 0FFH) + w, 8); DEC(k);
      IF k = 0 THEN data[i] := w; INC(i); k := 4; w := 0 END ;
      GET(data, d)
    END ;
    len := i
  END Rec;

  PROCEDURE MyNum*(): INTEGER;
    VAR x: INTEGER;
  BEGIN GET(stat, x); RETURN MSK(ROR(x, 4), 0FH)
  END MyNum;
```

```
END Ring.
```

Because the TRM is a word-addressed machine, the data to be transmitted are arrays of integers, whereas the ring interface transmits bytes as elements of a packet. Each array element must therefore by sent over the ring as four bytes. Procedure *Send*, after composing and sending the header byte, unpacks each integer into 4 bytes with the aid of a rotate instruction (ROR). The *Rec* procedure packs 4 consecutive bytes into an integer (word) by rotating and masking. (The second byte of each message is the parameter *typ*, which is not used in this context).

In PUT and GET operations, the first parameter indicates the address of the interface to be accessed. Here 0FC2H is the address of the data port, and 0FC3H that of the status. The status consists of 8 bits. It contains the following fields:

bit 0	input buffer empty (in = out)
bit 1	output buffer full (in = out)
bits 2, 3	0
bits 4-7	ring node number

A test setup

For testing and demonstrating the Ring with 12 nodes we use a simple test setup. It involves the program *TestTRMRing* for node 11, and the identical program *Mirror* for all nodes 0 – 10. The former is connected via the RS-232 link to a host computer running a general test program *TestTRM* for sending and receiving numbers. The main program *TestTRMRing* (running on TRM) accepts commands (via RS-232) for sending and receiving messages to any of the 12 nodes. Program *Mirror* then receives the sent message and returns it to the sender (node 11), which buffers it until requested by a read message command.

Communication over the link is performed by module RS, featuring procedures for sending and receiving integers and other items. The following are examples of commands:

```
TestTRM.SR 1 3 10 20 30 40 50 0 0~
TestTRM.SR 1 8 0 0~
TestTRM.SR 1 3 10 0 4 11 12 0 5 13 14 0 7 15 16 17 0 0~
TestTRM.SR 2~   receive message
```

The first command sends to node 3 the sequence of numbers 10, 20, 30, 40, 50. The second sends the empty message to node 8, and the third sends to node 3 the number 10, to node 4 the items 11 12, to node 5 the numbers 13, 14, and to node 7 the numbers 15, 16, 17.

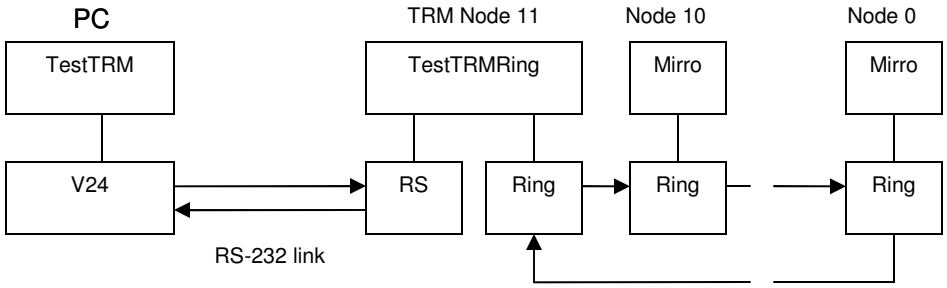


Fig. 4.2. The test setup

```
MODULE TestTRMRing;
  IMPORT RS, Ring;
  VAR cmd, dst, src, x, len, typ, s, i: INTEGER;
    buf: ARRAY 16 OF INTEGER;
BEGIN
  REPEAT RS.RecInt(cmd);
    IF cmd = 0 THEN RS.SendInt(Ring.MyNum())
    ELSIF cmd = 1 THEN  (*send msg*)
      RS.RecInt(dst);
      REPEAT len := 0; RS.RecInt(x);
        WHILE x # 0 DO buf[len] := x; INC(len); RS.RecInt(x) END ;
        Ring.Send(dst, 0, len, buf); RS.RecInt(dst)
      UNTIL dst = 0;
      RS.SendInt(len)
    ELSIF cmd = 2 THEN  (*receive msg*)
      IF Ring.Avail() THEN
        Ring.Rec(src, typ, len, buf);
        RS.SendInt(src); RS.SendInt(len); i := 0;
        WHILE i < len DO RS.SendInt(buf[i]); INC(i) END
      END
    ELSIF cmd = 3 THEN RS.SendInt(ORD(Ring.Avail()))
    END ;
    RS.End
  UNTIL FALSE
END TestTRMRing.

MODULE Mirror;
  IMPORT Ring;
  VAR src, len, typ: INTEGER;
    buf: ARRAY 16 OF INTEGER;
BEGIN
  REPEAT Ring.Rec(src, typ, len, buf); Ring.Send(src, 0, len, buf) UNTIL FALSE
END Mirror.
```

Broadcast

The design presented here was, as already mentioned, intentionally kept simple and concentrated on the essential, the transmission of data from a source to a destination node. A single extension was made, first because it is useful in many applications, and second in order to show that it was easy to implement thanks to a sound basis. This is the facility of broadcasting a message, that is, to send it to all nodes. The ring is ideal for this purpose. If the message passes once around

the ring, simply all nodes must be activated as receivers. We postulate that address 15 signals a broadcast. There are only two small additions to the circuit are necessary, namely the addition of the term *ringin[3:0] = 15* in the expression for *startrec*, and of the term *ringin[7:4] = mynum* in that of *stopfwd*.

Discussion

The presented solution is remarkably simple and the Verilog code therefore brief and the circuit small. This is most essential for tutorial purposes, where the essence must not be encumbered by and hidden in a myriad of secondary concerns, although in practice they may be important too.

Attractive properties of the implementation presented here are that there is no central agency, that all nodes are perfectly identical, that no arbitration of any kind is necessary, and that the message length is not a priori bounded. No length counters are used; instead, explicit trailers are used to designate the message end. All this results in a simple and tight hardware.

The data path of the ring is widened by 2 bits, a tag for distinguishing data from control bytes, which are token, message header, and message trailer. Actually, a single bit would suffice for this purpose. Two are used here in order to retain an 8-bit data field also for headers containing 4-bit source and destination addresses.

The simplicity has also been achieved by concentrating on the basic essentials, that is, by omitting features of lesser importance, or features whose function can be performed by software, by protocols between partners. The circuit does not, for example, check for the adherence to the prescribed message format with header and trailer. We rely on the total “cooperation” of the software, which simply belongs to the design. In this case, the postulated invariants can be established and safeguarded by packing the relevant drivers into a module, granting access to the ring by exported procedures only.

A much more subtle point is that this hardware does not check for buffer overflow. Although such overflow would not cause memory beyond the buffers to be affected, it would overwrite messages, because the buffers are circular. We assume that overflow of the sending buffer would be avoided by consistent checking against pending overflow before storing each data element, for example, by waiting for the buffer not being full before executing any PUT operation:

```
REPEAT UNTIL ~BIT(adr, 1)
```

In order to avoid blocking the ring when a message has partially been stored in the sending buffer, message sending is not initiated before the message end has been put into the buffer (signal *rdyS*). This effectively limits the length of messages to the buffer size (64), although several (short) messages might be put into the buffer, and messages being picked from the buffer one after the other.

A much more serious matter is overflow of the receiving buffer. In this case, the overflowing receiver would have to refuse accepting any further data from the ring. This can only be done by notifying the sender, which is not done by the presented hardware. For such matters, communication protocols on a higher level (of software) would be the appropriate solution rather than complicated hardware.

We consider it essential that complicated tasks, such as avoiding overflow, or of guaranteeing proper message formats, can be left to the software. Only in this way can the hardware be kept reasonably simple. A proper module structure encapsulating a driver for the ring is obviously necessary.

Experiments in Computer System Design

Niklaus Wirth

PART 5

The principle of Cache Memories

The introduction of a DMA for the DDR memory tripled its access speed. Nevertheless, it remains an unsatisfactory solution. The programmer does not wish to consider the memory as an array of blocks that can be written and read back. He wishes the large memory to be *the* memory, and any problems arising from using a DDR with wide access path to be problems of the hardware implementation. Considering this wish led to the concept of a *cache*.

A cache is a memory lying in the path between the processor and *the* memory. It is faster than the large memory, connects to it with a wide bus, and to the processor with a narrower bus. A *cache mechanism* handles all data transfers.

There exist several types of cache arrangements and mechanisms. The general idea is the following:

Let there be a main memory with 2^n words and a fast cache memory with 2^m words, where m << n. This cache is regarded as consisting of 2^{m-k} groups of 2^k words. Each group is called a *cache line*. Its length 2^k is equal to the access port width of the main memory. Data to and from the main memory are always transferred in groups of 2^k consecutive words forming a cache lines. The relatively large access time of large memories is compenstaed by a wider access path.

In addition to the cache memory, we introduce a table (array) T of 2^{m-k} entries, one for each cache line, called *tag*. A tag is the address of the associated cache line in main memory. When a word is to be read, the line in which it lies is read into any (free) line of the cache, and the group's address is stored in the corresponding tag. Actually, before reading a line, the table of tags is searched for the given address. If found, the line is aleady present in the cache and reading from main memory can be avoided. As the cache is much faster than main memory, this results in a significant gain in speed. Finding the given address is called a *hit*, not finding it a *miss*. The gain in speed depends significantly on the probability of hits. It is surprisingly large due to the fact that sequential access to consecutive words is predominant. This is particularly the case for reading instructions.

This scheme implies that the entire table of tags T must be searched to find the address. Evidently, a sequential search is out of the question. A sufficiently efficient solution requires an *associative memory*, where not a content is delivered given an address, but rather an address given a content, namely the address of the tag containing the memory address. This scheme is called a *(fully)*

associative cache. Such memories, however, are unpopular, because they require a lot of circuitry, essentially a comparator for every element of the table of tags. Simpler, but still effective solutions exist.

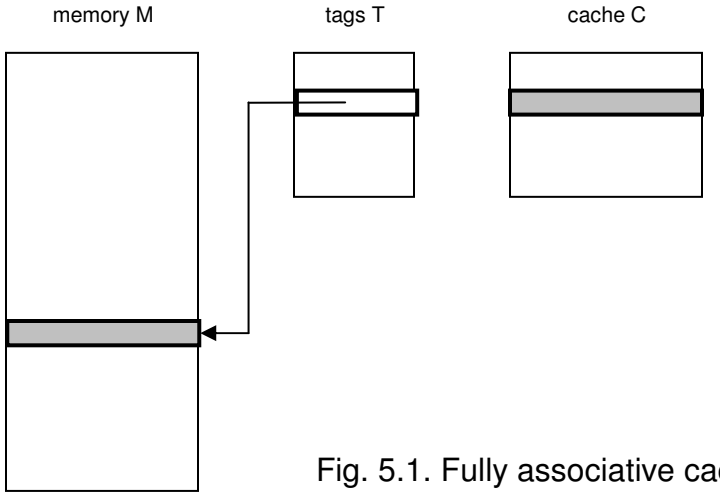


Fig. 5.1. Fully associative cache

The direct mapped cache

Here we will present only the simplest solution, the *direct-mapped* cache. It also uses a cache organized as a matrix, i.e. an array of cache lines, and a table of tags, one entry per cache line. The time-consuming search is avoided by mapping all blocks of the cache size 2^m directly onto the cache. Then all words with address $a\ MOD\ 2^m = b$, i.e. with the last m bits equal to b, correspond to the cache word with address b. The tag table *entry T[b]* then contains the address of the block in main memory containing the cache line *C[b]*.

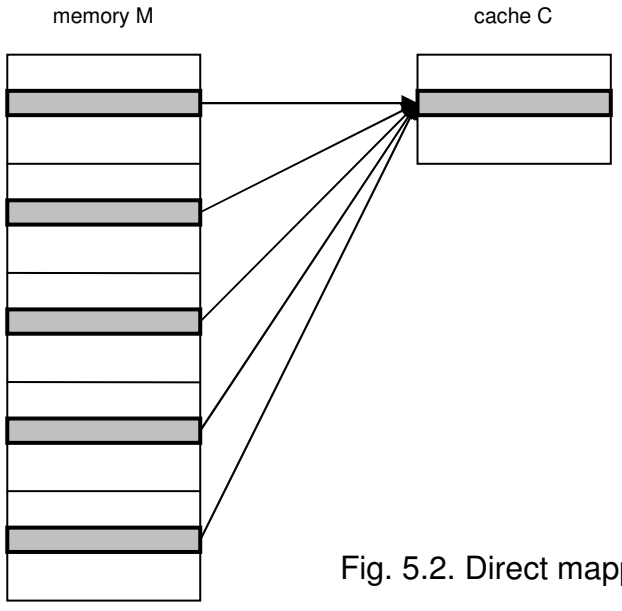


Fig. 5.2. Direct mapping

The constants *k, m, n* are determined by the available hardware components. In the present case (ML-505 board), the memory size is 256 MB = 2^{28} bytes. As we

deal with words rather than bytes (4 per word) a word address consists of $n = 26$ bits. The DDRAM's access path is 256 bits, i.e. 8 words wide. Hence $k = 3$.The cache memory is implemented as a single block RAM with $1K = 2^{10}$ words. Hence $m = 10\text{-}k = 7$. The main memory then consists of 2^{16} blocks of 1K words, and an address *a* consists of 3 fields:

Madr	16 bits	a[25:10]	block in memory
Tadr	7 bits	a[9:3]	line in block
Wadr	3 bits	a[2:0]	word in line

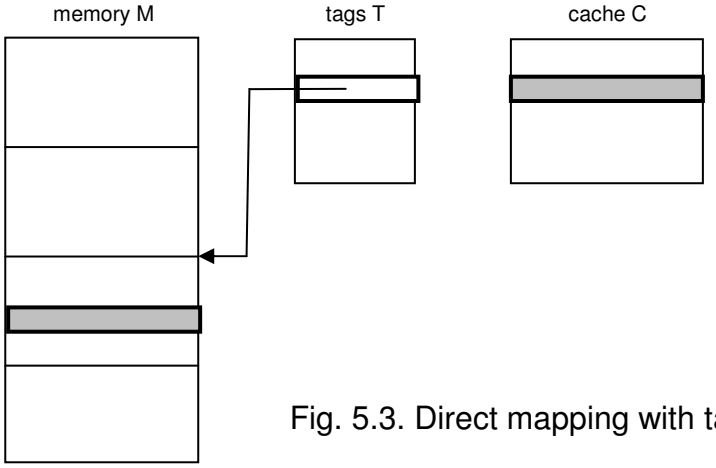


Fig. 5.3. Direct mapping with tags

We summarize the operations involved in accessing a word in memory:

1. Compare *Madr* with *T[Tadr]*.
2a. If equal, the desired word is in the cache at address [*Tadr, Wadr*].
2b. If not equal, the word is not in the cache. The cache line is fetched from address [*Madr, Tadr, 0*] in memory and stored at address *Tadr* in the cache. Then the word is selected at *Wadr* in the line.

In case 2b, the line is overwritten in the cache. The old line is lost. This implies that the line must be stored in main memory beforehand. This can be omitted, if no word in the line had been overwritten (by any Store instruction) since the line had been loaded. For this purpose, an additional bit in the tag memory is introduced for each entry, showing whether or not any word in the line had been modified. This bit is called *modif*. Unless it had been set by a Store instruction, the cache line need not be written back. The bit is cleared whenever a cache line is loaded. Accordingly, we expand step 2b to

2b. If not equal: Store line at *Tadr* into memory at address [*T[Tadr], Tadr*, 0], if modif[Tadr] = 1. Then fetch cache line at [*Madr, Tadr*, 0].

Implementing the cache

We now present our implementation of the direct cache for the TRM in detail. We recall that the SDRAM has a capacity of 256 MB, and the cache contains 1K words. This is only half the TRM's local memory. We remove the upper half from the cache mechanism and map addresses 0FFFFFC0H – 0FFFFFFFFH to the upperhalf with addresses 400H – 0FFFH. This range includes I/O addresses and

the stack. The software stack is frequently accessed , which justifies its exclusion from the cache.

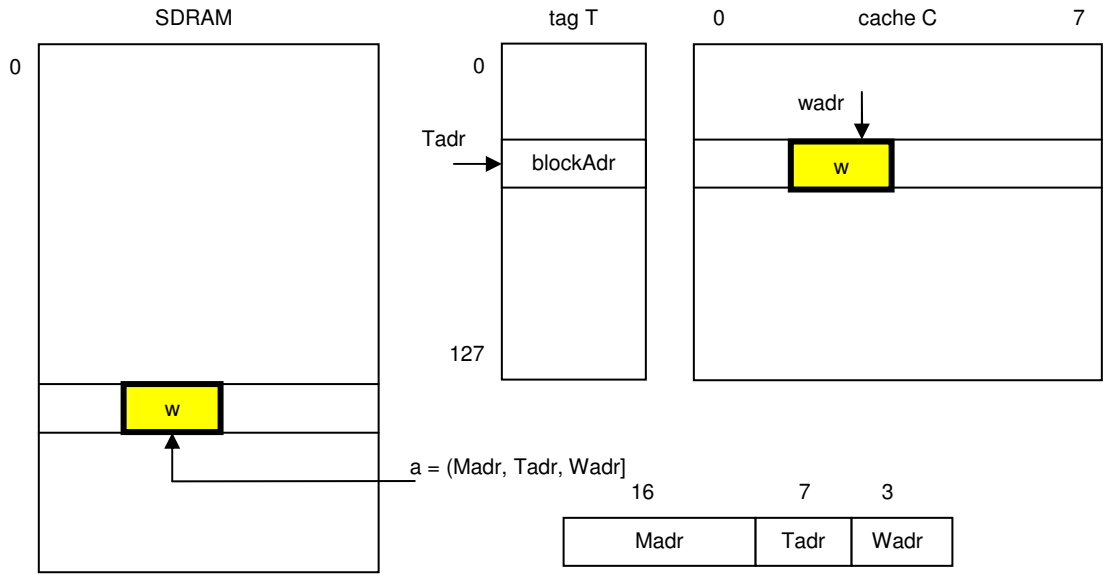


Fig. 5.4. Cache memory C with tag table T

Table *T* must provide fast access and is therefore implemented as an LUT RAM similar to the TRMs register bank. It consists of 16 +1 slices of RAM128X1D with 2^7 entries.

The access mechanism for SDRAM is largely taken over from the previous version with DMA. However, the complexity of the setup and of the algorithm to access the memory suggest that the state machine be moved from the device into the processor, i.e. from module TRM3CTop into TRM3C. The additions to the TRM turn out to be nontrivial and substantial. The DDR signals in the TRM interface now are *dmain* and *dmaout* as with DMA. Additionally there are the address *ddradr* and the control outputs *dmard, dmawr, ddwr, ddrd,* and the state input *DDstat*.:

```
module TRM3C(
input clk, rst, stall,
input irq0, irq1,
input[31:0] inbus,
output [5:0] ioadr,
output iord, iowr,
output [31:0] outbus,
input [2:0] DDstat,   // dma/ddr connections
input [31:0] dmain,
output dmard, dmawr, ddrd, ddwr,
output [22:0] ddradr,
output [31:0] dmaout);
```

The new wires and registers are:

```
reg caEnb;   // cache enable and states
reg Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q11, Q12;
```

```
reg [2:0] wcnt;  // DMA word count
reg [11:0] dmaAdr;
reg [22:0] DDadr;
reg [26:0] adrR;

wire [26:0] adr;
wire Twr, Twr1, miss, missE, modif, dbit, adrHi, modif; dmEnb;
wire [6:0] Tadr;
wire [15:0] Madr;
wire [16:0] Tin, Tout;
```

Twr (and *Twr1*) are the write enables for the tags. *Miss* signals *cache misses*, and it is active when the address part *Madr* does not match the corresponding tag entry (*Tout*), and if not the uppermost 1K block of memory is addressed (~*adrHi*). The table of tags is defined by

```
genvar i;
generate   // tags for cache 128 x (16+1)
  for (i = 0; i < 17; i = i+1)
  begin: tags
    RAM128X1D #(.INIT(128'h00000000000000000000000000000000))
   TAG(
   .A(Tadr),  // r/w adr, controls D, SPO
   .D(Tin[i]),
   .SPO(),
   .DPRA(Tadr),  // read only adr, controls DPO
   .DPO(Tout[i]),
   .WCLK(clk),
   .WE((i == 16) ? Twr1 : Twr));
  end
endgenerate
```

The signal *adr* is now extended from 12 bits to 26 bits. The dma-Signals are taken over from the DMA implementation.

```
assign adr = ((irs == 7) ? 0 : AA[26:0]) + {19'b0, off};
assign dmadr = (dmEnb) ? dmaAdr : {1'b0, adrHi, adr[9:0]};
assign dmwr = (dmEnb) ? dmawr : ST &  ~miss;
assign dmin = (dmEnb) ? dmain : B;

assign ddradr = DDadr;
assign dmaout = dmout;
assign adrHi = (Madr == 16'hffff);
assign miss = ~(Madr == Tout[15:0]) & ~adrHi;
assign missE = miss & caEnb;
assign Tadr = adr[9:3];
assign Madr = adr[25:10];
assign Tin = {dbit, Madr};
assign modif = Tout[16];
```

The heart of the cache system is the state machine controlling data transfers between SDRAM (DDR2) and cache. It is triggered out of the idle state whenever a cache miss occurs. We chose the one-hot form of state machine with states Q0 – Q12. The – after many considerations – obvious solution is to extend the already present rudimentary state machine, which stalls the LDR instruction for one cycle, from 2 to 13 states with the following assocuated actions:

Q0	idle
Q1	extend memory access
Q2	initialize DMA
Q3, Q4	transfer 8 words from cache to buffer
Q5	wait until SDRAM ready
Q6	write buffer to SDRAM
Q7	initialize DMA
Q8	wait until SDRAM ready
Q9	read buffer from SDRAM
Q10, Q11	wait until data ready
Q12	transfer 8 words from buffer to cache

The state machine is described by the following diagram (MEM = LDR | ST).

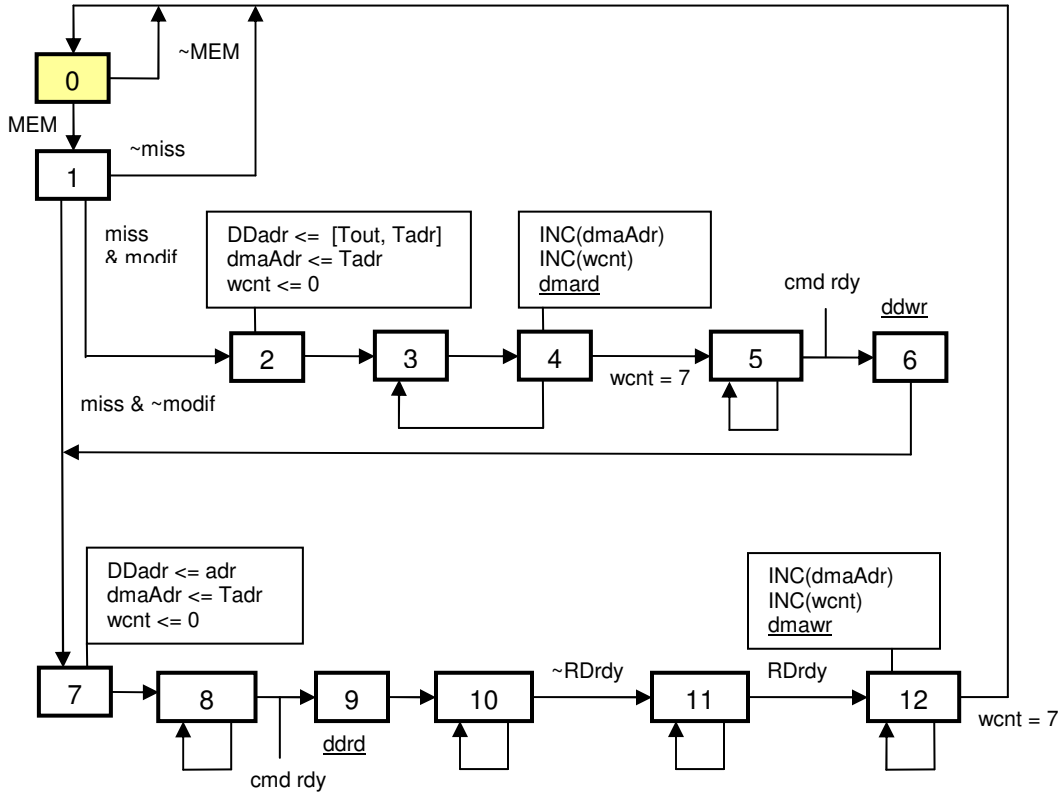


Fig. 5.5. Cache control state machine

In Verilog, the state machine is expressed by the following clocked statements.

```
always @ (posedge clk) begin  // cache state machine
  Q0 <= ~rst | Q0 & ~MEM | Q1 & ~missE | Q12 & wc7;
  Q1 <= Q0 & MEM;  // Twr1
  Q2 <= Q1 & missE & modif;
  Q3 <= Q2 | Q4 & ~wc7;
  Q4 <= Q3;  // dmard
  Q5 <= Q4 & wc7 | Q5 & DDstat[2];
  Q6 <= Q5 & ~DDstat[2];  // ddwr
  Q7 <= Q6 | Q1 & missE & ~modif; // Twr
  Q8 <= Q7 | Q8 & DDstat[2];
  Q9 <= Q8 & ~DDstat[2];  // ddrd
```

```
  Q10 <= Q9 | Q10 & DDstat[0];
  Q11 <= Q10 & ~DDstat[0] | Q11 & ~DDstat[0];
  Q12 <= Q11 & DDstat[0] | Q12 & ~wc7;  // dmawr
end
```

Unfortunately it turned out that the condition *miss* cannot be established in a single clock cycle. *adr* is computed in one cycle, but the comparison *Madr = Tout* takes a second cycle. We therefore must resort to the trick of inserting a register (*adrR*) in the signal path, not the least because in this way the load (fanout) condition can be met. This is of no negative consequence for the LDR instruction. However, the ST instruction now also takes a second cycle, whereas this had not been necessary before. The *stall* condition is asserted in the second cycle unconditionally by both the LDR and ST instructions. Thereafter it is asserted by the *miss* condition, and by the state machine in all states except Q0 and Q1.

```
assign dmEnb = ~Q0 & ~Q1;
assign stallC = (Q0 & MEM) | (Q1 & missE) | dmEnb;
assign stall0 = stallM | stallD | stallC | stall;
```

The state machine controls the data transfer by the signals *dmard, dmawr, ddrd,* and *ddwr* (control signals to DDR and DMA in the interface to DDRController). *wcnt* is the counter that controls the dma-transfer by counting 8 words. DDstat[2] means : “DDR controller busy”, and DDstat[0] means “DDR output ready”.

```
always @ (posedge clk) begin
  adrR <= adr;
  DDadr <= Q2 ? {Tout[15:0], Tadr} : Q7 ? adr[25:3] : DDadr;
  dmaAdr <= (Q2|Q7) ? {2'b0, Tadr, 3'b0} :  (Q4|Q12) ? dmaAdr + 1 : dmaAdr;
  wcnt <= (Q2|Q7) ? 0 : (Q4|Q12) ? wcnt + 1 : wcnt;
end
```

Noting that states Q3, Q4 and Q12 are actually repeated 8 times, we conclude that an access with cache miss costs either 19 or 36 cycles (depending on whether or not the cache line had been modified), whereas an access with a hit takes only 2 cycles. A remarkable difference!

And this concludes the introduction of a direct cache store. It is not obvious that the direct cache method would prove efficient. After all, it seems likely that cache misses are frequent with 2^{16} lines mapping from SDRAM to the same line in the cache. But in fact the direct-mapped cache proved quite satisfactory, considering its *relative* simplicity. An intermediary method between fully associative and direct mapped cache is the *n-way associative cache*. Here *n* tag tables and *n* cache memories coexist, and if any one of the tags in corresponding lines matches the desired address, the associated cache yields the word to be accessed. Only *n* comparators are needed. In present commercial processors up to 8-way associative caches are provided. A much simpler and hardly less effective solution is to double or quadruple the size of the cache.

Typically, separate caches are provided for data and program access. Here we have shown only a data cache. A program cache is simpler, because instructions are read only. No *modif* condition and no write-back are needed.

Acknowledgement

My sincere thanks go to Ling Liu for her help, encouragement and drive in this project. Without her advice and support the author would never have mustered the patience to overcome the difficulties and aggravations caused by the necessary tools, in particular the Verilog compiler and the Xilinx placer and router. They were a great disappointment, as they proved to be rather unhelpful in locating mistakes, and instead provide innumerable pitfalls through their misguided efforts to “correct” programmers’ mistakes. In addition, huge lists of “warnings” are utterly unattractive to find those warnings that actually may point out mistakes.

References

[1] Xilinx, ML505/ML506/ML507 evaluation platform user guide, http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

[2] Xilinx, Spartan-3 Starter kit board user guide, http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD_RM.pdf