The Design of a RISC Architecture
and its Implementation with an FPGA

Niklaus Wirth, 11.11.11, rev. 5.10.13

Abstract

1. Introduction

The idea for this project has two roots. The first was a project to design and implement a small processor for use in embedded systems with several interconnected cores. It was called the *Tiny Register Machine* (TRM). The second root is a book written by this author some 30 years ago, and revised several times since. Its subject is *Compiler Construction*. The target for the developed compiler is a hypothetical computer. In the early years this computer was a stack architecture, later replaced by a RISC architecture. Now the intent is to replace the hypothetical, emulated computer by a real one. This idea was made realistic by the advent of programmable hardware components called *field programmable gate arrays* (FPGA).

The development of this architecture is described in detail in this report. It is intentionally held in a tutorial style and should provide the reader with an insight into the basic concepts of processor design. In particular, it should connect the subjects of architectural and compiler design, which are so closely interconnected.

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

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

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

But these blocks of RAM are relatively small (1 - 4K words). The development board used in this project, however, provides an FPGA-external Static RAM with a capacity of 1 MByte. In the second stage (RISC-1) the BRAM for data is replaced by this SRAM.

1

In the third stage (RISC-2) we switch to a *von Neumann Architecture*. It uses a single memory for both instructions and data. Of course we use the SRAM of stage 1.

In the last stage (RISC-3) two features are added to round up the development, byte accessing and interrupts. (They might be omitted in a course without consequences).

It follows that RISC-0 and RISC-2 are the corner stones of this project. But we keep in mind that from a programmers' standpoint all four implementations appear identical (with the exception of the features added in RISC-3). All four have the *same architecture and the same instruction set.*

2. The RISC Architecture

An architecture describes a computer as seen by the programmer and the compiler designer. It specifies the resources, i.e. the registers and memory and defines the instruction set. (possibly implying data types).

Processors consist of two parts, the *arithmetic/logic unit* (ALU) and the *control unit*. The former performs arithmetic and logical operations, the latter controls the flow of operations. In addition to the processor there is memory. Our RISC consists of a memory whose individually addressable elements are *bytes* (8 bits).

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

The core of the data processing unit consisting of ALU and registers is shown in Figure1. Evidently, data cycle from registers into the ALU, where an operation is performed, and the result is deposited back into a register.
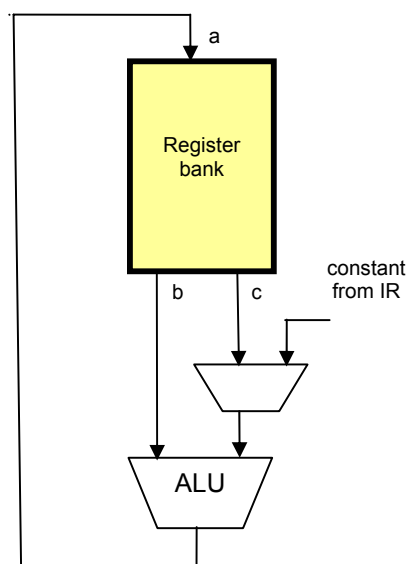

Fig. 1. Processor core with ALU and registers

The control unit contains a register holding the instruction currently being under execution. This is called the *instruction register* (IR). It also features a register holding the address of the instruction being fetched from program memory. This is traditionally called the *program counter* (PC). The control unit computes the address of the next instruction. As this is mostly the one in sequence, i.e. with address PC+1, the control unit also holds an incrementer. Branch instructions take the next address from the instruction register.

External devices are mapped into the address space of memory. This means that a certain range of addresses is reserved for access to input and output devices.

3. The Instruction Set

The term "reduced" suggests that keeping the instruction set small is a good concept. But how much can one reduce it? Is there a lower limit? Which are its criteria? There is, of course, no clear cut measure. An important goal is a certain *completeness*. An instruction set should make it possible to compose any more complex operation out of the basic instructions, The second goal must be *regularity*, straight rules without exceptions. It facilitates description, understanding, and implementation enormously.

The principal enemy of regularity and simplicity is the drive for speed, for efficiency. We place priority on a simple, regular design. Improvements to increase speed can be added at a later stage. A sound rule to begin is that instructions simple to implement should have priority. Among them are certainly all logical operations, and because of their frequency the basic arithmetic operations such as addition and subtraction.

The RISC architecture divides instructions into three classes, namely (1) arithmetic and logic instructions operating on registers, (2) operations to transfer data between registers and memory, and (3) control (branch) instructions.

3.1. Register instructions

We follow the established convention to provide instructions with 3 register numbers, two specifying the operands (sources), one the result (destination). Thus we obtain a 3-address computer. It gives compilers the largest degree of freedom to allocate registers for optimal efficiency. Logical operations are the conventional AND, OR, XOR. The arithmetic operations are the four basic operations of addition, subtraction, multiplication and division. The inclusion of the latter two is to some degree questionable. They are considered basic in mathematics, although they can be constructed out of addition or subtraction. Their complexity is indeed of a higher order. This must be paid by higher "cost", either in time or circuitry.

Furthermore, we include a set of shift instructions, moving bits horizontally. In fact, a single one, namely rotation would suffice. Rotation is the best choice because it loses no information, and thus all other can be constructed out of it. However, we also provide left shift (LSL) and (arithmetic) right shift (ASR). The former feeds zeroes in at the low end, the latter replicates the top bit at the right end. The shift count can be any number between 0 and 31.

All 16 register instructions use the same two forms. In form *F0*, both operands are registers. In format *F1* one operand is a register, the other is a constant held in the instruction itself. In format *F1*,

The complete set of register instructions is shown the following Table 1 in an assembler-like form. *R.a* is the destination register, and *R.b* is the first operand. The second operand is either register *R.c*, or the literal ("immediate") *im*. In this case, the *modifier bit v* determines how the 16-bit constant *im* is extended into a 32-bit value. The two forms of instructions are encoded as shown in Fig. 2 *n* stands for either *R.c* or *im*.

0	MOV a, n	R.a := n	
1	LSL a, b, n	R.a := R.b ← n	(shift left by n bits)
2	ASR a, b, n	R.a := R.b → n	(shift right by n bits with sign extension)
3	ROR a, b, n	R.a := R.b rot n	(rotate right by n bits)

4	AND	a, b, n	R.a := R.b & n	logical operations
5	ANN	a, b, n	R.a := R.b & ~n	
6	IOR	a, b, n	R.a := R.b or n	
7	XOR	a, b, n	R.a := R.b xor n	
8	ADD	a, b, n	R.a := R.b + n	integer arithmetic
9	SUB	a, b, n	R.a := R.b – n	
10	MUL	a, b, n	R.a := R.a x n	
11	DIV	a, b, n	R.a := R.b div n	
12	FAD	a, b, c	R.a := R.b + R.c	floating-point arithmetic
13	FSB	a, b, c	R.a := R.b – R.c	
14	FML	a, b, c	R.a := R.a x R.c	
15	FDV	a, b, c	R.a := R.b / R.c	

v = 0   extension of *im* with 16 zero bits
v = 1   extension of *im* with 16 one bits
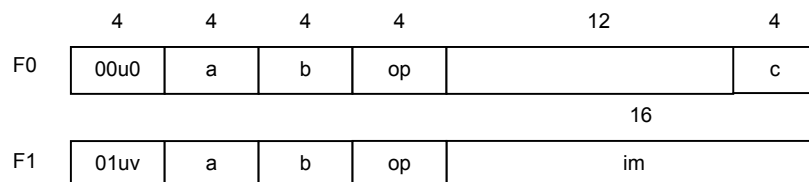
Table 1. Register instructions

Fig.2. Formats F0 and F1. for register instructions

All register instructions (with the exception of multiplication and division) have side effects on the four single-bit condition registers, namely

N := bit 31 of result. Hence, N indicates whether the result is negative.
Z := all bits of result zero. Z indicates whether the result is zero.
C := carry out bit. (For addition and subtraction only).
V := overflow. (For addition and subtraction only)

These four condition registers are tested by branch instructions. The DIV instruction deposits the remainder in an auxiliary register H.

Register instructions contain two modifier bits *u* and *v*. The instruction MOV with *u* set to 1 shifts the immediate value *im* by 16 bits to the left. Instructions ADD and SUB with the modifier bit *u* set to 1 add (subtract) the carry bit C, and the MUL instruction with *u* set to 1 considers the operands as unsigned numbers, yielding a 64-bit unsigned product.

3.2. Memory instructions

There are only two memory instructions, namely load and store. They specify a destination register *R.a* for load, or a source register for store. The address in memory is the sum of register *R.b* and a 20-bit offset. The format is shown in Figure 3.

LD   a, b, off   R.a := Mem[R.b + off]
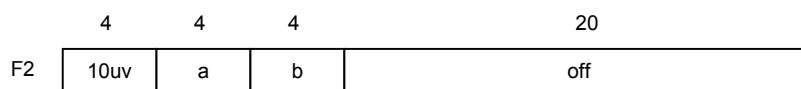ST   a, b, off   Mem[R.b + off] := R.a

Fig. 3. Format F2 for memory instructions

The modifier bits have the following significance:

> u = 0 load, u = 1 store
> v = 0: word, v = 1: byte   (implemented on RISC-3 only)

3.3. Branch instructions

Branch instructions are used to break the sequence of instructions. The next instruction is designated either by a 24-bit signed offset, or by the value of a register, depending on the modifier bit *u*. It indicates the length of the jump forward or backward (PC-relative addressing). This offset is in *words*, not bytes, as instructions are always one word long.

> Bcond   off
>
> u = 0   PC := R.c   u = 1   PC := PC+1+off
> v = 0   no link   v = 1   R15 := PC+1

The modifier v determines, whether the current value of PC be stored in register R15 (the *link* register). This facility is used for calls to procedures. The value stored is then the return address. The format is shown in Fig. 4.
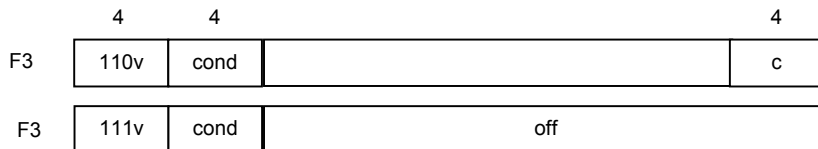
	4	4		4
F3	110v	cond		c

F3	111v	cond	off	

Fig. 4. Format F3 of branch instructions

The following conditions are available:

code	cond	condition		code	cond	condition	
0000	MI	negative (minus)	N	1000	PL	positive (plus)	~N
0001	EQ	equal (zero)	Z	1001	NE	positive (plus)	~Z
0010	CS	carry set	C	1010	CC	carry clear	~C
0011	VS	overflow set	V	1011	VC	overflow clear	~V
0100	LS	less or same	~C|Z	1100	HI	high	~(~C}Z)
0101	LT	less than	N≠V	1101	GE	greater or equal	~(N≠V)
0110	LE	less or equal	(N≠V)|Z	1110	GT	greater than	~((N≠V)|Z)
0111		true	T	1111		false	F

4. Implementing a Harvard Architecture (RISC-0)

The building of circuits involves physical components, such as gates and registers, and wires connecting them. The use of a modern FPGA on a chip facilitates this task tremendously. There is no need to select component chips with registers, mutiplexers, or decoders, and no need to hard-wire them with soldering iron or wire-wrap gun. Instead, the circuit description is fed to a tool consisting of a synthesizer, a placer (selecting appropriate elements from the ones available on-chip), and a router (using the available wires to connect them).

Although this scheme has simplified hardware design drastically, there still exist the limitations of available components and of routing resources. If a design becomes too complex, or too large, the tool may not be able to perform its task Therefore, we have a strong incentive to keep our design reasonably simple and well structured.

In addition and in contrast to software design, there exists the consideration of timing. Signals propagate with finite speed. Each gate introduces a certain delay. We must keep path lengths, the number of gates a signal passes between its origin register and its destination register, small. The timing considerations are greatly simplified, if we adhere to *synchronous* design. This implies that all

registers are run by *the same clock.* The FPGA provides special wiring for clock signals in order to keep clock skew limited and ignorable.

We describe the circuit by a program text in the HDL Verilog, and we stick to the following scheme, where a module consists of 4 parts:

1. The header with the list of input and output signals (parameters).
2. The declaration part, introducing the names of signals (wires) and registers.
3. The assignments of (Boolean) values to signals (wires)..They have the form

      assign s = expression;

4. The assignments to registers. They have the form

      R <= expression;

and occur under the heading

      always @ (posedge clk) **begin ….. end**

where *clk* is the global clock signal.

There are two input signals occurring in the header of every module, namely *clk* and *rst*. The latter is a negative reset signal (in our case activated by a push button).

The RISC system consists of several modules. The principal module is called *RISC0*, and it implements the processor. There are two submodules implementing a multiplier and a divider. There is also a module called *RISC0Top*, representing the processor's "environment". It is a Verilog rule that only a top module can import off-chip signals. This module contains the connections to external devices. They include an RS-232 transmitter and an RS-232 receiver, representing a serial line. The module also contains the connections to a set of 8 LEDs and to 8 switches. This configuration is shown in Figure 5.
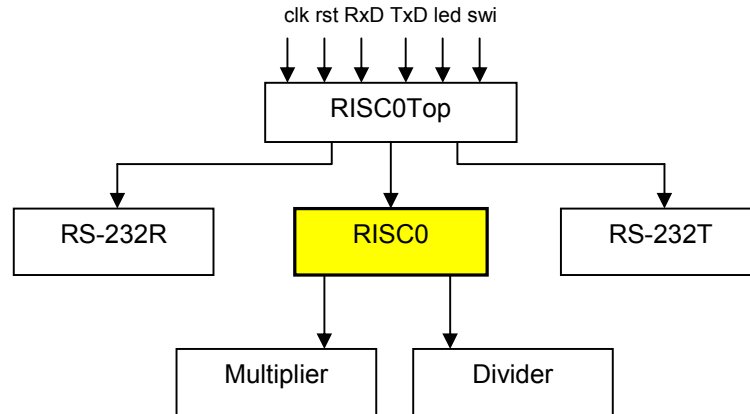

Fig. 5. Block diagram of the RISC-0 configuration

The header of module RISC0 lists its parameters, signals to and from the "environment":

      input clk, rst,
      input [31:0] inbus, codebus,
      output [11:0] adr,
      output rd, wr,
      output [31:0] outbus

4.1. The computation unit (ALU)

As mentioned earlier, the processor is divided into two parts, the computation unit and the control unit. In the block diagram of Figure 6 the computation unit is to the left, the control unit to the right, and it is evident that they are connected by a few wires only.

The heart of the computation unit is the *bank of registers*, each of the 16 registers with 32 bits. From the description of the architecture it is evident that there must be 3 output ports corresponding to the register numbers *a, b,* and *c* (fields in the register instructions)., plus one output port addressed by *a*. Here we consider the register bank as a "black box" with the following input and output signals. We will return to further details of implementation later in this text.

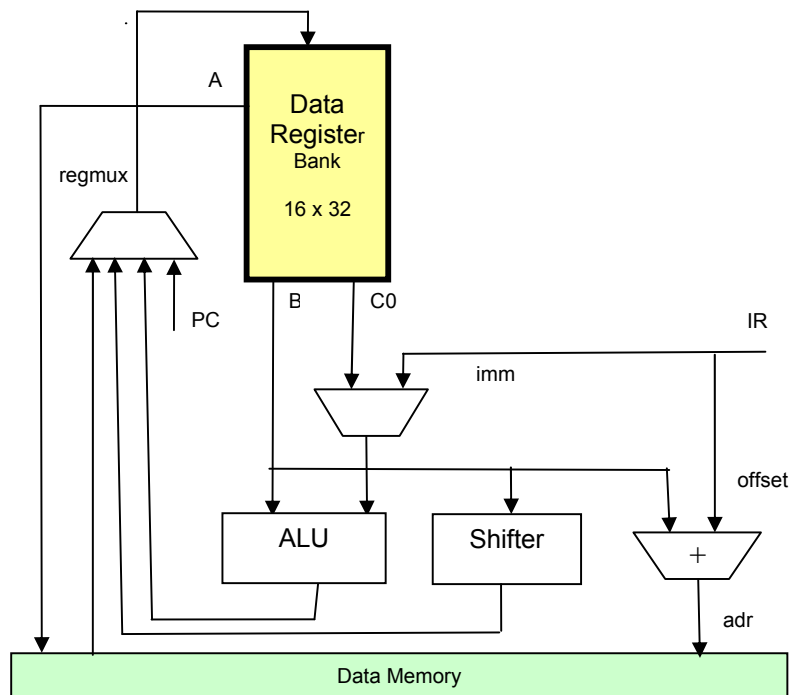|	
A	data output port (32 bit)
B	data output port (32 bit)
C0	data output port (32 bit)
D	data input port (32 bit)
ira0	address of ports A and D (4 bit)
irb	address of port B (4 bit)
irc	address of port C0 (4 bit)
clk	clock
regwr	write enable

Fig.6. Block diagram of computation unit

Before proceeding, we postulate the following auxiliary signals derived from the instruction register IR, representing instruction fields and their decoding (see instruction formats in Ch. 3).

IR = pmout;   instruction from program memory
p = IR[31];   instruction fields
q = IR[30];
u = IR[29];
v = IR[28];
w = IR[16];
cc = IR[26:24];

```
ira = IR[27:24];
irb = IR[23:20];
op  = IR[19:16];
irc = IR[3:0];
imm = IR[15:0];
off = IR[19:0];

MOV = ~p & (op == 0);   decode signals
LSL = ~p & (op == 1);
ASR = ~p & (op == 2);
ROR = ~p & (op == 3);
AND = ~p & (op == 4);
ANN = ~p & (op == 5);
IOR  = ~p & (op == 6);
XOR = ~p & (op == 7);
ADD = ~p & (op == 8);
SUB = ~p & (op == 9);
MUL = ~p & (op == 10);
DIV = ~p & (op == 11);

LDW = p & ~q & ~u;
STW = p & ~q & u;
BR  = p & q;
```

From the block diagram of Fig. 6 we now derive the following expressions for further signals:

```
ira0 = (BR) ? 15 : ira;                         return address of BL to R15
C1 = ~q ? C0 : {{16{v}}, imm};

aluRes =
 MOV ? C1 :
 LSL ? t3 :                                      left shifter output
 ASR ? s3 :                                      right shifter output
 ROR ? s3 :                                      right shifter output
 AND ? B & C1 :
 ANN ? B & ~C1 :
 IOR  ? B | C1 :
 XOR ? B ^ C1 :
 ADD ? B + C1 + (u & C) :
 SUB ? B - C1 – (u & C) :
 MUL ? product[31:0] :                           multiplier output
 DIV ? quotient : 0;                             divider output

regmux =                                         register input D
 (LDW & ~ioenb) ? dmout                          from memory :
 (LDW & ioenb) ? inbus                           from input bus        :
 (BR & v) ? {18'b0, nxpc, 2'b0} : aluRes         return address from control unit;
```

In most instructions, the result *regmux* is stored in a register. The exceptions are:

> Branch instructions, unless the return address is to be saved (BR & ~u)
> Branch instructions whose condition is not met (BR & ~cond)
> Store instructions
> Stalls (discussed later)

The register set is represented by an array of 16 registers:

```
reg [31:0] R [0:15]
```

Writing to registers is controlled by the signal *regwr,* defined as follows:

```
regwr = (~p & ~stall) | (LDW & ~stall & ~stall1) (BR & cond & v);
```

In addition to the data registers, four single-bit registers are provided to hold some predicates of the computed result. They are traditionally called *condition codes,* and they are used (tested) by branch instructions. These registers are

```
N <= aluRes[31]              result is negative
Z <= (aluRes[31:0] == 0)     result is zero
C <= aluRes[32]              carry out
OV <= aluRes[32] ^ aluRes[31]  overflow
```

N and Z are set by all register operations, C and V by additions and subtractions only. Sums and differences are 33 bit wide, Bit 32 is the carry out (overflow for unsigned arithmetic).

Parallel to the path from register output to register input through the ALU lies the path through two *shifters*, one for left, one for right shifts. Shifters are, like circuits for logical and arithmetic operations, entirely combinational circuits. They consist of multiplexers only. The *left shift* (LSL) feeds zero bits to the right.

A basic element on our FPGA are "lookup tables" with 4 inputs and a single output resulting from any function of the inputs. Hence, a 4-input multiplexer is the preferred element. In order to be able to shift by any amount between 0 and 31, three stages are required. The first shifts by 0, 1, 2, or 3 bits, the second by 0, 4, 8, or 12 bits, and the third by 0 or 16 bits. The following signals are involved:

```
wire [31:0]  t1, t2,  t3;
wire [1:0] sc1, sc0;  // shift counts
```

The shifter input is *B*, its output is *t3*. The intermediate outputs between multiplexers are t1 and t2. The shift count is split into

```
assign sc0 = C1[1:0];
assign sc1 = C1[3:2];

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

The right shifter is quite similar. It implements both right shift and right rotation (determined by *u*). In the former case, bits shifted in at the left end are replicas of the leftmost (sign) bit B[31], whereas for rotation they are taken from the right end.

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

assign s1 = (sc0 == 3) ? {(w ? B[2:0] : {3{B[31]}}), B[31:3]} :  // shifter for ASR and ROR
  (sc0 == 2) ? {(w ? B[1:0] : {2{B[31]}}), B[31:2]} :
  (sc0 == 1) ? {(w ? B[0] : B[31]), B[31:1]} : B;
assign s2 = (sc1 == 3) ? {(w ? s1[11:0] : {12{s1[31]}}), s1[31:12]} :
  (sc1 == 2) ? {(w ? s1[7:0] : {8{s1[31]}}), s1[31:8]} :
  (sc1 == 1) ? {(w ? s1[3:0] : {4{s1[31]}}), s1[31:4]} : s1;
assign s3 = C1[4] ? {(w ? s2[15:0] : {16{s2[31]}}), s2[31:16]} : s2;
```

And this concludes the explanation of the computation unit of the processor.

4.2. The data memory

The data memory DM is represented in our case by a so-called Block RAM, a facility provided by the FPGA used. (Its name is *dbram32*), and it has the following signals:

```
wire [31:0] dmin, dmout;        input and output ports
wire dmwr;                      write enable
```

```
wire [13:0] dmadr;                    memory address

dmin = A;
dmwr = STW & ~stall;
dmadr = B[13:0] + off[13:0];
```

The memory is configurable as a 2K (or 4K) block. Its elements are 32-bit words. As the RISC uses byte addresses, the address fed to the RAM is actually *dmadr[12:2]*, and only words, not bytes can be accessed.. Address bits 0 and 1 are ignored. (The *stall* signal will be explained later).

4.3. The control unit

The control unit determines the sequence of executed instructions. It contains two registers, the program counter PC holding the address of the current instruction, and the current instruction register IR holding the instruction currently being interpreted. Instructions are obtained from the program memory PM, also a Block RAM with 2K words and the signals

```
wire [31:0] pmout;                    output port
wire [11:0] pcmux;                    memory address
```

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

    0. Zero on reset.
    1. The next instructions address is the current one's (PC) plus 1
    2. It is given by the instruction explicitly. (Branch instructions).
    3. It is taken from a data register. (This is used for returning from procedures).

In a Harvard architecture, computational and control units operate simultaneously. In each clock cycle, the control unit fetches the next instruction for the program memory into the instruction register IR, and the computation unit operates on registers (or on the data memory). As an example the short instruction sequence

```
0       ADD R0 R0 2
1       SUB R0 R0 1
2       B  0
```

 is traced as follows over 5 clock cycles:

pcmux	IR	PC	R0
0			
1	ADD	0	0
2	SUB	1	2
0	B	2	1
1	ADD	0	1
2	SUB	1	3

The control unit is quite straight-forward, as show in Figure 7, and defined by the following expressions:

```
IR = pmout;
nxpc = PC + 1;
pcmux = (~rst) ? 0 :
  (stall) ? PC :  // do not advance
  (BR & cond & u) ? off[11:0] + nxpc :
  (BR & cond & ~u) ? C0[13:2]  :  nxpc;

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

Here we notice that all branch instructions are subject to a condition. The condition is specified by the condition field of the instruction and is a logical function of the 4 condition registers N, Z, C, V. The signal *cond* is defined as

```
cond = IR[27] ^            IR[27] inverts sense
  ((cc == 0) & N |         MI, PL
   (cc == 1) & Z |         EQ, NE
   (cc == 2) & C |         CS, CC
   (cc == 3) & OV |        VS, VC
   (cc == 4) & (C|Z) |     LS, HI
   (cc == 5) & S |         LT, GE
   (cc == 6) & (S|Z) |     LE, GT
   (cc == 7));             T, F
```
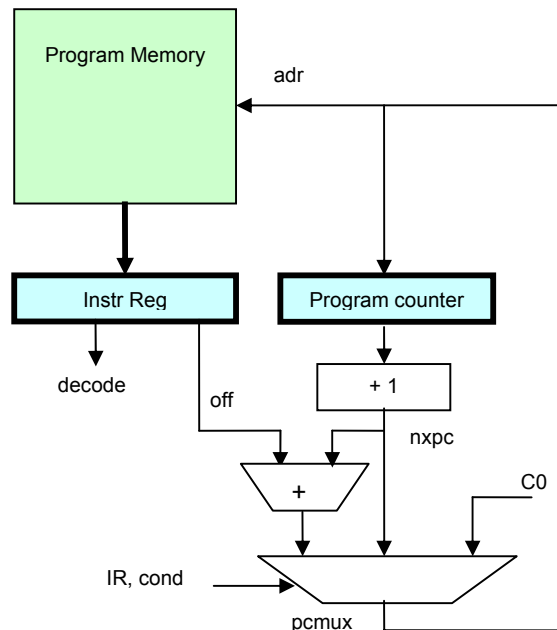

Fig. 7. The control unit

The memory block of the FPGA already contains an output register. For this reason IR is actually declared as a wire, but not as a register (which is represented by *pmout*). Note that the memory has *clk* as an input signal. This seemingly trivial and incidental detail has yet another consequence.

As the address signal travels through the decoder and the selected value proceeds to this output register, the read value is actually available only in the next cycle after the address was applied to the memory. This implies that actually the reading of a value takes 2 cycles. This contradicts the rule that in RISCs every instruction be executed in a single cycle. However, memory access is considered as a special case, and provisions must be made to allow it to take longer.

The device to solve this problem is *stalling* the processor, i.e. to retain it in the same state over 2 or more cycles. This is achieved by simply generating a *stall* signal which, if active, feeds PC itself to the multiplxer. In our case, this signal must arise when a LD instruction is present, and it must remain active for exactly one clock period. This is achieved by the following circuit with signals shown in Figure 8::

```
    reg stall1;
    wire stall, stallL;

    stall = stallL;
    stallL = LDW & ~stall1;
    stall1 <= stallL;
```

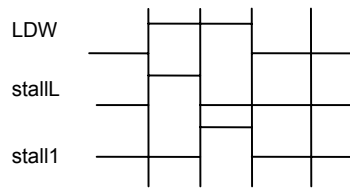This circuit actually represents a very simple state machine.

Fig. 8. Stall generation for memory access

4.4. The multiplier

Multiplication is an inherently more complex operation than addition and subtraction. After all, multiplication can be composed (of a sequence) of additions. By not vice-versa. There are many methods to implement multiplication, all – of course – based on the same concept of a series of additions. They show the fundamental problem of trade-off between time and space (circuitry). Some solutions operate with a minimum of additional circuitry – actually without – but sacrifice speed. These are the implementations in software. Naturally, they perform the necessary additions in sequence. The partial sums are stored in variables. At the other end of the spectrum lie solutions which use a multitude of adders operating concurrently (in parallel). They are fast, but the amount of required circuitry is high.

And, of course, there are solutions that lie in between these extremes. We here present such a middle solution. It needs a single adder only and performs additions in sequence, with partial sums stored in a special register. The sequence of additions is triggered by a single multiply instruction, which makes this solution still faster than a pure software solution, which would typically be presented as a procedure.

Let us briefly recapitulate the basics of a multiplication $p := x \times y$. Here p is called the product, x the multiplier, and y the multiplicand. Let x and y be *unsigned* integers. Consider x in binary form.

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

We obtain the product by a sequence of 32 additions, each term of the form $x_k \times 2^k \times y$, i.e. of y left shifted by k positions multiplied by x_k. Since x_k is either 0 or 1, the product is either 0 or y (shifted). Multiplication is thus performed by an adder and a selector.

The selector is controlled by x_k, a bit of the multiplier. Instead of selecting this bit among $x_0 \ldots x_{31}$, we right shift x by one bit in each step. Then the selection is always according to x_0. This suggests that instead of shifting y to the left in each step, we keep y in the same position and shift the partial sum p to the right. We notice that the size of x decreases by 1 in each step, whereas the size of p increases by 1. This suggests to pack p and x into a single register pair <B,A> with a shifting border line. At the end, it contains the 64-bit product p = x × y.

$$p = \{B[31:0], A\{31:[32-k]\}, \qquad x = A[31-k:0] \qquad k = 0 \ldots 31$$

The multiplier is controlled by a rudimentary state machine S, actually a simple 5-bit counter running from 0 to 31. The multiplier is shown schematically in Figure 9.

The multiplier interprets its operands as signed (u = 0) or unsigned (u = 1) integers. The difference between unsigned and signed representation is that in the former case the first term has a negative weight ($-x_{31} \times 2^{31}$). Therefore, implementation of signed multiplication requires very little change: Term 31 is subtracted instead of added (see complete program listing below).
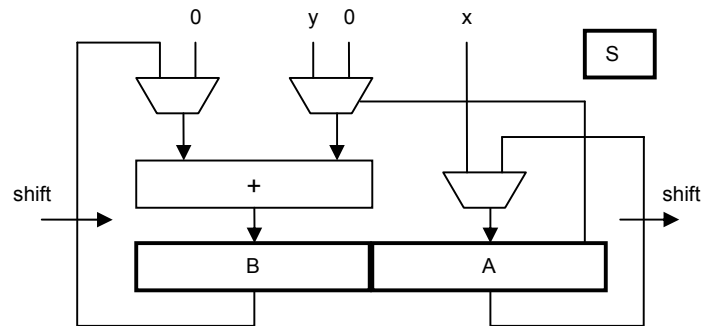
Fig. 9. Schematic of multiplier

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

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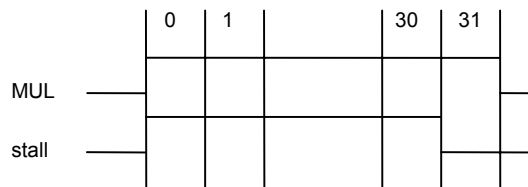

Fig. 10. Generating *stall*

The details of the multiplier are listed below:

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

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

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

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

13

```
endmodule
```

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

4.5. The divider

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

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

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

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

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

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

Initially, r may be a 64-bit value; its size decreases by 1 in each step. q is initially 0, and its size increases by 1 in each step, suggesting to use a single double register with shifting border. This is the same as in multiplication, except that the shift is to the left.
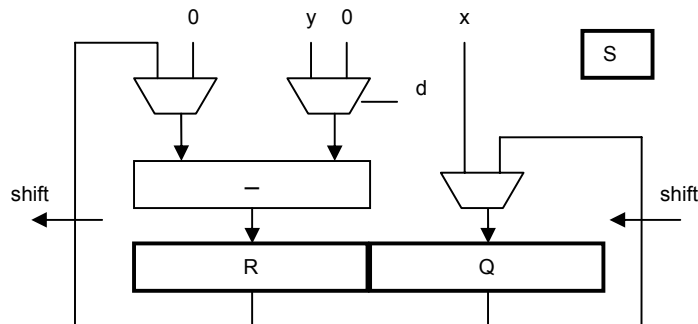

Fig. 11. Schematic of divider

Stall generation is the same as for the multiplier. Further details are shown in the subsequent program listing.

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

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

assign stall = DIV & ~(S == 31);
assign r0 = (S == 0) ? 0 : r3;
```

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

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

4.6. The environment

The RISC-0 processor module is imported by module RISC0Top, which thereby represents the processor's environment. Apart from providing clock and reset signals, it connects various devices with the processor's input and output busses, and selects them according to the i/o address. These devices are

   0. A counter incremented every millisecond (adr = 0)
   1. A set of 8 dip switches (adr = 4)
   2. A set of 8 light emitting diodes (LED) (adr = 4)
   3. An RS-232 receiver (serial data line) (adr = 8, 12)
   4. An RS-232 transmitter (adr = 8, 12)

The RS-232 receiver and transmitter are described in subsequent chapters, the top module itself is listed here with the omission of clock generation). The clock frequency is 35 MHz. The signals in its heading refer to off-chip components (see also Fig. 5)..

```
module RISC0Top(
  input CLK50M,
  input rstIn,
  input RxD,
  input [7:0] swi,
  output TxD,
  output [7:0] leds);

wire clk, clkLocked;
reg rst;

wire[5:0] ioadr;
wire [3:0] iowadr;
wire iowr;
wire[31:0] inbus, outbus;

wire [7:0] dataTx, dataRx;
wire rdyRx, doneRx, startTx, rdyTx;
wire limit;  // of cnt0

reg [7:0] Lreg;
reg [15:0] cnt0;
reg [31:0] cnt1; // milliseconds

RISC0 riscx(.clk(clk), .rst(rst),
```

```
      .iord(iord), .iowr(iowr), .ioadr(ioadr), .inbus(inbus), .outbus(outbus));

   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 iowadr = ioadr[5:2];
   assign inbus = (iowadr == 0) ? cnt1 :
      (iowadr == 1) ? swi :
      (iowadr == 2) ? {24'b0, dataRx} :
      (iowadr == 3) ? {30'b0, rdyTx, rdyRx} : 0;

   assign dataTx = outbus[7:0];
   assign startTx = iowr & (iowadr == 2);
   assign doneRx = iord & (iowadr == 2);
   assign limit = (cnt0 == 35000);
   assign leds = Lreg;

   always @(posedge clk)
   begin
     rst <= ~rstIn & clkLocked;
     Lreg <= ~rst ? 0 : (iowr & (iowadr == 1)) ? outbus[7:0] : Lreg;
     cnt0 <= limit ? 0 : cnt0 + 1;
     cnt1 <= limit ? cnt1 + 1 : cnt1;
   end
   endmodule
```

4.7. The RS-232 transmitter

RS-232 is an old standard for serial data transmission. We chose to describe it here because of its inherent simplicity. The data are transmitted in packets of a fixed length, here of length 8, i.e. byte-wise. Bytes are transmitted asynchronously. Their beginning is marked by a start-bit, and at the end a stop-bit is appended. Hence, a packet is 10 bits long (see Fig. 10). Within a packet, transmission is synchronous, i.e. with a fixed clock rate, on which transmitter and receiver agree. The standard defines several packet lengths and many clock rates. Here we use 19200 bit/s.
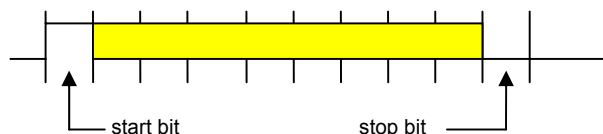

Fig. 10. RS-232 packet format

The input signal *start* triggers the state machine by setting register *run*. The transmitter has 2 counters and a shift register. Counter *tick* runs from 0 to 1823, yielding a frequency of .35'000 / 1823 = 19.2 KHz, the transmission rate for bits. The signal *endtick* advances counter *bitcnt*, running from 0 to 9 (the number of bits in a packet). Signal *endbit* resets *run* and the counter to 0. Signal *rdy* indicates whether or not a next byte can be loaded and sent.

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

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

    assign endtick = tick == 1823;
    assign endbit = bitcnt == 9;
    assign rdy = ~run;
    assign TxD = shreg[0];

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

4.8. The RS-232 receiver

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

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

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

    assign endtick = tick == 1823;
    assign midtick = tick == 911;
    assign endbit = bitcnt == 8;
    assign data = shreg;
    assign rdy = stat;

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

5. Implementing a von Neumann Architecture

The presented Harvard Architecture uses separate memories for program and data. The advantage is that instruction fetch and data execution can proceed concurrently. It was the revolutionary idea of John von Neumann to merge the two. This made it possible to treat program code as data and to perform operations on it. This was, for example, used for adding an array of numbers stored in consecutive memory cells, typically performed by a tight program loop. Before each iteration, the address of the instruction fetching the value to be added is incremented by 1.

This trick was made redundant by the later addition of *index registers*. Now computed values would automatically be added to the constant in the instruction to form the effective address. In general, modification of program code during execution proved to be a technique loaded with pitfalls, and it is now virtually extinct. However, on a larger scale, loading of a (new) program is also based on von Neumann's concept of shared program and data memory, a feature without which computers are unthinkable. Yet, the Harvard architecture is still used in applications, where the computer always executes the same program. This is mostly the case in embedded applications for sensing data or controlling machinery. Its advantage is a gain in speed, because program and data memories are accessed concurrently.

5.1. Using the large memory (RISC-1)

Before proceeding to a conversion of our RISC to the von Neumenn scheme, we show the replacement of the small 2K Block-RAM for data storage by the large 1 MByte SRAM. This large memory is not part of the FPGA chip, but resides outside the FPGA on the Spartan-3 board. This necessitates that also the environment of the RISC be adapted. We start by showing the *changes* necessary in the processor circuit, rather than re-listing the entire circuit, and we start with the processor itself.

First of all, the address signals are widened from 12 to 20 bits, the output *adr* is widened from 5 to 20 bits.

```
output [19:0] adr;
adr = B[19:0] + off;
```

The data BRAM *dm* is removed, and with it the following signals:

```
dmadr, dmwr, dmin, dmout, ioenb
```

Connection to the SRAM is through the module's interface (header), and thus the separation of memory access and external device access is made in the environment (RISC1Top) rather than the processor module. The signals *iowr* and *iord* are now simply *wr, rd*.

So far, the changes are all removals resulting in simplifications. Memory access will now occur via the environment like access to external devices, and the selector between memory and devices is therefore moved to the environment. Hence, it is the environment that receives the extensions. This is so, because the SRAM is external to the FPGA, and access via the FPGA's pins can only be specified in a top module. They appear in its parameter list:

```
module RISC1Top( …
  output SRce0, SRce1, SRwe, SRoe,
  output [3:0] SRbe,
  output [19:0] SRadr,
  inout [31:0] SRdat);
```
The new declarations are

```
wire[19:0] adr;
wire [17:0] iowadr; // word address
wire rd, wr, ioenb;
wire[31:0] inbus, inbus0, outbus;

RISC1 riscx(.clk(clk), .rst(rst), .rd(rd), .wr(wr),.adr(adr), .inbus(inbus), .outbus(outbus));
```

The new signals (wires) obtain the following values:

```
assign iowadr = adr[5:2];
assign ioenb = (adr[19:6] == 14'b11111111111111);
assign inbus = ~ioenb ? inbus0 :
    ((iowadr == 0) ? cnt1 :
     (iowadr == 1) ? swi :
     (iowadr == 2) ? {24'b0, dataRx} :
     (iowadr == 3) ? {30'b0, rdyTx, rdyRx} : 0);

assign SRce0 = 1'b0;
assign SRce1 = 1'b0;
assign SRwe = ~wr | clk;
assign SRoe = ~rd;
assign SRbe = 4'b0;
assign SRadr = adr[19:2];

genvar i;
generate // tri-state buffer for SRAM
  for (i = 0; i < 32; i = i+1)
  begin: bufblock
    IOBUF SRbuf (.I(outbus[i]), .O(inbus0[i]), .IO(SRdat[i]), .T(~wr));
  end
endgenerate
```

The last paragraph specifies the tri-state buffer *SRbuf*. It is required, because the SRAM is connected with the FPGA by a single, bi-directional bus *SRdat*. The circuits are shown schematically in Fig. 11.
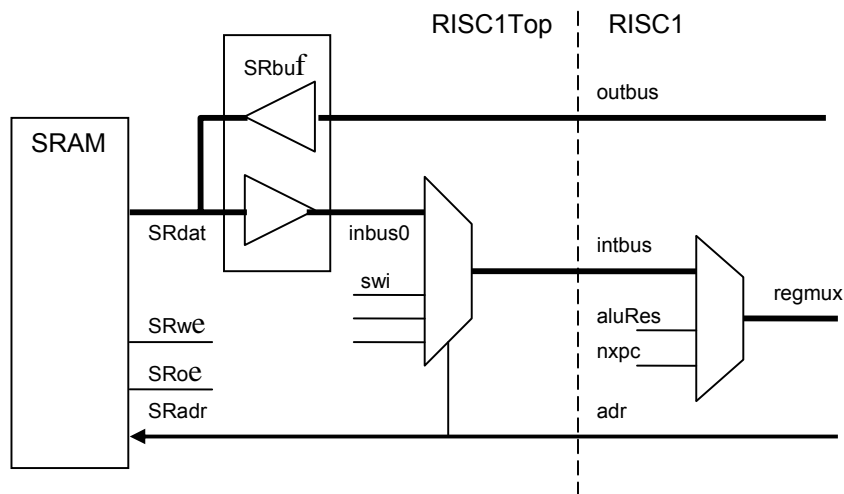

Fig. 11. Connections between processor and SRAM

The write strobe *wr* determines the direction of the tri-state buffer. An absolutely essential detail is that the write strobe be ored with the clock signal (~wr | clk). This is to activate writing only in the second half of the clock period. The SRAM is a combinational circuit, and writing must wait until the address lines are stable, which is the case in the second half of the cycle. The various enable signals *ce0, ce1, be* are all active (low). (Note that like in RISC0 we deal with word transfers only, without byte selections. The least 2 address bits are ignored)

As an aside, we note that the SRAM is fast enough and is without a register at data output. This makes a stall cycle unnecessary, again simplifying the processor. The clock rate is 35 MHz.

5.2. Using SRAM for program *and* data (RISC-2)

We are now in a position to unite program and data memories following the concept of *von Neumann*. There is practically no change in the top module, except that *inbus0* is also brought to the interface. It lets instructions bypass the multiplexer for input from external devices. Let us consider the evolution of the processor.

First we note the widening of *PC* (and *pcmux* and *nxpc*) from 12 to 18 bits. Because instructions are always 4 bytes long, their address is always a multiple of 4. Hence, the address path is not 20 (as would be needed for 1 MB), but only 18 bits wide. This concerns the declarations

```
output [19:0] adr;
reg [17:0] PC;
wire [17:0] pcmux, nxpc;
```

In addition, the following changes are necessary:

```
regmux = ….. (BR & v) ? {12'b0, nxpc, 2'b0} : aluRes;
pcmux = ….. (BR & cond & ~u) ? C0[19:2] : nxpc;
```

When using the on-chip block RAM for program memory, the instruction register is contained within the RAM. Its output *pmout* is a register. This is not so for the SRAM, and therefore a register has to be declared explicitly. We call it *IRBuf*:

```
reg [31:0] IRBuf;
wire [31:0] IR;
  IR = IRBuf;
  IRBuf <= codebus;
```

But this is only part of the story. We need to review the entire fetch/execute mechanism. Every instruction is first fetched from memory into the instruction register. It is interpreted (executed) in the *next* cycle. In order to double speed, the instruction flow is pipelined: While an instruction is executed, the next instruction is fetched. This works fine in the case of the Harvard architecture. It also works for register instructions in a von Neumann architecture, but not for memory instructions, because a data access would interfere with the fetching of the next instruction. Therefore, a stall cycle must be introduced for memory instructions, loads as well as stores. We let the *stall* signal indicate, whether an instruction fetch or a data access cycle is in progress. In the former case, the address is *pcmux*, in the latter it is *B + off*: (see Fig. 12). The clock frequency had to be reduced from 35 to 25 MHz.

```
assign stall = stallL | stallM | stallD;
assign stallL = (LDW | STW) & ~stall1;

assign adr = stall ? (B[19:0] + off) : {pcmux, 2'b00};
assign wr = STW & ~stall1;
assign rd = ~wr;
regwr = ccwr | (LDW & ~stall1);
```

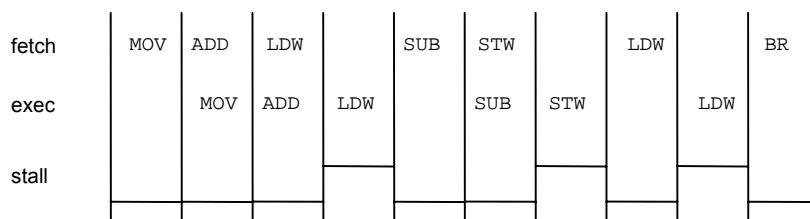| fetch	MOV	ADD	LDW		SUB	STW		LDW		BR
exec		MOV	ADD	LDW		SUB	STW		LDW	
stall										

Fig. 12. Stalling memory instructions

There remains the (nasty) problem of how to load a program into the SRAM. The Xilinx loader always loads a bit-stream into BRAM. In our case this is a boot loader for moving compiled code into the SRAM. This implies that transfer of control from BRAM to SRAM must occur after the boot loading. Of course we wish to use the RISC itself to execute the boot loader. For this purpose, we map the BRAM

(2K) onto SRAM, actually to its high end. *PMsel* determines the memory from which instructions are read into *IR*.

The starting address on reset is now changed from 0 to *StartAdr*. The transfer of control after boot loading is achieved by a branch to 0.

```
reg PMsel;

StartAdr = 18'b111111100000000000;
IRBuf <= stall ? IRBuf : codebus;
PMsel <= ~rst | (pcmux[17:11] == 7'b1111111;
IR = PMsel ? pmout : IRBuf;
```

5.3. Implementing byte-access (RISC-3)

The FPGA's block RAMs are arrays of words, not bytes. We have therefore refrained from implementing direct access to bytes. (of course bytes can be extracted and inserted by mask and rotate instructions). Here we show how to provide byte-access in an efficient way which is possible, because the SRAM allows to store individual bytes, although basically it is also word-oriented. (In fact the SRAM of the Spartan board consists of 2 16-bit memory chips, and hence it might be called *half-word oriented*). The SRAM provides 4 write-enable signals, one for each byte in a word. By generating these signals from the low 2 bits of the address, individual bytes can be stored without affecting the other 3 bytes. Reading of a byte, however, always involves the reading of the word containing it, and then shifting and masking the byte desired.

Because this scheme is determined by the SRAM used, we place the necessary circuits exclusively into the environment module as shown below. The only addition to the processor interface is the signal *be* (byte enable) derived from the modifier bit *v* in memory instructions.

For byte-wise *reading*, multiplexers are inserted into the input path. We rename the IOBuf's output from *inbus0* to *inbus1*, and redefine *inbus0* as

```
inbus0[7:0] = (~be | a0) ? inbus1[7:0] : a1 ? inbus1[15:8] : a2 ? inbus1[23:16] : inbus1[31:24];
inbus0[31:8] = ~be ? inbus1[31:8] : 24'b0;
```

with

```
a0 = ~adr[1] & ~adr[0];
a1 = ~adr[1] & adr[0];
a2 = adr[1] & ~adr[0];
a3 = adr[1] & adr[0];
```

For byte-access *writing,* multiplexers need be inserted in the output path. The IOBuf's input *outbus* is replaced by *outbus1*, which is defined as:

```
outbus1[7:0] = outbus[7:0];
outbus1[15:8] = be & a1 ? outbus[7:0] : outbus[15:8];
outbus1[23:16] = be & a2 ? outbus[7:0] : outbus[23:16];
outbus1[31:24] = be & a3 ? outbus[7:0] : outbus[31:24];
```

The various chip- and byte-enable signals are defined as (see also 5.1.):

```
SRce0 = be & adr[1];
SRce1 = be & ~adr[1];
SRbe[0] = be & adr[0];
SRbe[1] = be & ~adr[0];
SRbe[2] = SRbe[0];
SRbe[3] = SRbe[1];
```

5.4. Interrupts

The last facility to be added to the RISC is that for interrupts. It was first introduced in computers around 1960. The principal motivation was avoiding the need for distinct, small processors to handle small, peripheral task, such as the acceptance of input data and of buffering them, before the actual computing process was ready to accept them. Instead, the main process was to be interrupted, i.e. the processor was to be borrowed (for a short time) to handle the request, and thereafter to be returned to its interrupted task. Hence, the interrupt facility had a purely economical motivation.

One might assume that in the era of unbounded computing resources those small processess would no longer have to share a processor, but would be represented by programs on separate, distinct processors such as microcontrollers. This is partly happening. However, the interrupt is such a convenient feature for economizing hardware that it continues to persist. It is indeed, as will be seen shortly, very cheap to implement – at least in its basic form.

One should consider the effects of an interrupt evoked by an external signal as if a procedure call had been inserted at random in the current instruction sequence. This implies that not only the call instruction is executed, but also that the entire state of the current computation be preserved and recovered after ending the interrupt procedure. Different strategies for implementation differ primarily by the techniques for saving the state. The simplest implementation saves, like a procedure call, only the current PC (in an extra register), and leaves the rest to the interrupting program, such as the saving of used registers to software. This is the cheapest solution for hardware, but also the most time-consuming for software. At the other end of the spectrum lies hardware saving all registers including stack pointers, or even providing multiple sets of registers for interrupts, letting an interrupt handler look as a regular procedure. A further sophistication lies in providing several, distinct interrupt signals, perhaps even with distinct priorities, or even programmable priorities. There is no limit to complexification.

One addition, however, is mandatory, namely a state register indicating whether or not interrupts are admitted (*intEnb*). For the programmer of interest is that we obtain three new instructions, one for returning from an interrupt routine, and two for setting the enable state. They are all encoded in the form of branch instructions with u = 0:

Enable/disable int 0	1100	1111		1100	00ee
Enable/disable int 1	1100	1111		0100	0000
Return interrupt	1100	0111		0001	Rn
Trap	1101	cond		0010	dst

Fig. 13. Special instructions for interrupt and trap handling

Following our credo for simplicity we here present a solution requiring the minimal effort on the side of the hardware. This also lies in accord with the needs for teaching the concept. These are the declarations of new variables::

input irq0, irq1;	interrupt requests
reg intMd, intEnb0, intEnb1;	interrupt mode, interrupt enable
reg [21:0] SPC;	saved PC on interrupt
wire [17:0] pcmux0;	
wire intAck;	enabled interrupt request
wire nn, zz, cx, vv;	condition codes
wire RTI;	return from interrupt instruction

When one of the *irq* signals becomes high, an interrupt is pending. It causes *intAck* to become high, if interrupts are enabled (*intEnb*), the processor is not currently executing another interrupt handler (~*intMd*), and the processor is not stalled. When *intAck* is high, the current PC is saved in the new

SPC register, and *pcmux* becomes 1, i.e. the next instruction is read from address 4. At the same time, the condition bits are also saved (in *SPC[21:18]*), and the processor enters the interrupt mode.

```
intAck0 = irq0 & intEnb0 & ~intMd & ~stall;
intAck1 = irq1 & intEnb1 & ~intMd & ~stall;
pcmux = ~rst ? 0 : intAck0 ? 1 : intAck1 ? 2 : pcmux0;

N <= nn; Z <= zz; C <= cx; OV <= vv;
SPC <= (intAck0 | intAck1) ? {nn, zz, cx, vv, pcmux0} : SPC;
intMd <= rst & ~RTI & (intAck0 | intAck1 | intMd);
```

A return from interrupt instruction (RTI) causes the next instruction address to be taken back from SPC[17:0] and the condition bits from SPC[21:18]. Furthermore, the processor leaves the interrupt mode.
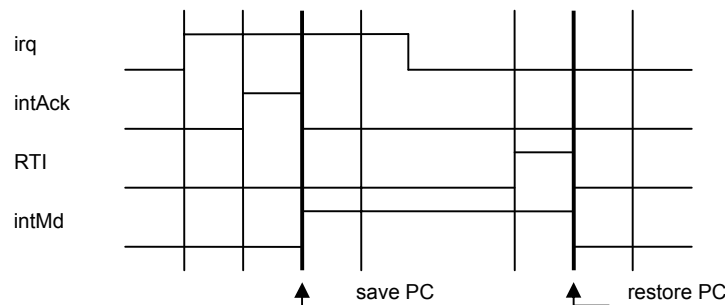

Fig. 14. Interrupt signals

```
RTI = p & q & ~u & IR[4];
pcmux0 = stall ? PC : RTI ? SPC[17:0] : ……

nn = RTI ? SPC[21] : ccwr ? aluRes[31] : N;
zz = RTI ? SPC[20] : ccwr ? (aluRes[31:0] == 0) : Z;
cx = RTI ? SPC[19] : (ADD|SUB) ? aluRes[32] : C;
vv = RTI ? SPC[18] : (ADD) ? (sa & ~sb & ~sc | ~sa & sb & sc) :
        (SUB) ? (sa & ~sb & sc | ~sa & sb & ~sc) : OV;
```

The interrupt mode instruction copies IR[7] into the *intEnb* register, which is cleared upon reset.

```
intEnb0 <= ~rst ? 0 : (BR & ~u & IR[6]) ? IR[0] : intEnb0;
intEnb1 <= ~rst ? 0 : (BR & ~u & IR[7]) ? IR[1] : intEnb1;
```

There is evidently very little additional circuitry due to the interrupt facility. However, this solution requires that software saves and restored all registers used in an interrupt routine. Note also that the interrupt routine must reset the external interrupt request. This typically is done by a command to the respective device, effectively an acknowledge signal.

In order to generate an interrupt signal, the environment is extended as follows. When *cnt0* reaches a limit, *cnt1* is incremented and *irq* is set to 1. This causes an interrupt every millisecond, given a clock frequency of 25 MHz.

```
reg irq;
assign limit = (cnt0 == 25000);

always @(posedge clk) begin
  irq <= ~rst | (wr & ioenb& (iowadr == 0)) ? 0 : limit ? 1 : irq;
end
```

References

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

2. N. Wirth. *Compiler Construction.* Addison-Wesley, 1995.