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

Niklaus Wirth

11.11.11

Released

Sept. 1st, 2015

Revised

Aug. 9th, 2018

The idea for this project has two roots. The 1st 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 2nd root is a book written by me 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 (Reduced Instruction Set Computer) 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 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 1st is the design of the architecture itself, (more or less) independent of subsequent implementation considerations. Then follows a 1st implementation called RISC-0. For this we chose a Harvard Architecture, implying that 2 distinct memories are used for program and for data. For both we use chip-internal memory, so-called Block RAMs

(BRAM). The Harvard architecture allows for a neat separation of the arithmetic from the control unit.

But these BRAMs are relatively small (1 - 4K words). The development board used in this project, however, provides an FPGA-external Static RAM (SRAM) with a capacity of 1 MByte. In the 2nd stage (RISC-1) the BRAM for data is replaced by this SRAM.

In the 3rd 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 4th stage (RISC-3) 2 features are added to round up the development, byte accessing and interrupts. (They might be omitted in a course without consequences).

RISC-4 adds non-volatile storage (disk) in the form of an SD-card (flash memory). This device is accessed through a serial line according to the SPI (Serial Peripheral Interface) standard.

RISC-5 finally adds further peripheral devices. They are a keyboard and a mouse (both accessed via standard PS-2 interface), a memory-mapped display (accessed via VGA standard), and a network (accessed via SPI standard).

It follows that RISC-0 and RISC-5 are the corner stones of this project. But we keep in mind that from a programmers' standpoint all implementations appear identical (with the exception of the features added in RISC-3). All 4 have the same architecture and the same instruction set.

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* (CU). 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. AL 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 (Complex ISC) architectures: *Memory is largely decoupled from the processor*. A 2nd 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 Figure 1. Evidently, data cycle from

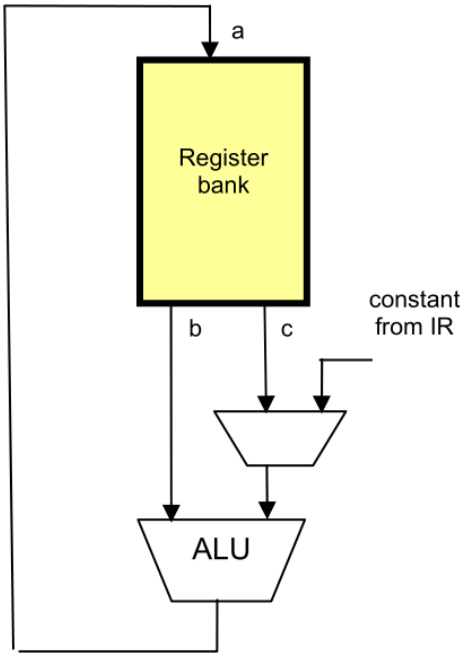

Figure 1: Processor core with ALU and registers

registers into the ALU, where an operation is performed, and the result is deposited back into a register.

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

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.

The term *reduced* suggests that keeping the Instruction Set (IS) 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 IS should make it possible to compose any more complex operation out of the basic IS. The 2nd 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 3 classes:

1. AL instructions operating on registers,
2. data transfer instructions between registers and memory, and
3. control (branch) instructions (BI).

2.1 *RI: Register Instruction*

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 4 basic operations of addition, subtraction, multiplication and division. The inclusion of the latter 2 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 Logical Shift Left (LSL) and Arithmetic Shift Right (ASR). The former feeds zeroes in at the low end, while the latter replicates the top bit at the high end. The shift count can be any number between 0 and 31.

All 16 RIs use the same 2 formats:

1. F0: both operands are registers;

2. F1: one operand is a register, the other is a constant held in the instruction itself.

The complete set of RIs is shown in the following codes in an assembler-like form. *R.a* is the destination register, and *R.b* is the 1st operand. The 2nd 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 2 forms of instructions are encoded as shown in Figure 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 // Logical Shift Left
 2 ASR a,b,n R.a:=R.b  -> n // Arithmetic Shift
    // Right (fill with sign bit instead of zero)
 3 ROR a,b,n R.a:=R.b rot n // ROtate Right

 4 AND a,b,n R.a:=R.b  &  n // logical AND
 5 ANN a,b,n R.a:=R.b  & ~n // ANd Not
 6 IOR a,b,n R.a:=R.b  or n // Inclusive OR
 7 XOR a,b,n R.a:=R.b xor n // eXclusive OR

 8 ADD a,b,n R.a:=R.b  +  n // integeral ADD
 9 SUB a,b,n R.a:=R.b  -  n // SUBstract
10 MUL a,b,n R.a:=R.b  *  n // MULtiply
11 DIV a,b,n R.a:=R.b div n // DIVide

12 FAD a,b,c R.a:=R.b + R.c // Float ADd
13 FSB a,b,c R.a:=R.b - R.c // Float SuB
14 FML a,b,c R.a:=R.b * R.c // Float MuL
15 FDV a,b,c R.a:=R.b / R.c // Float DiV

v=0 extension of im with 16 zero bits
v=1 extension of im with 16 one bits
```

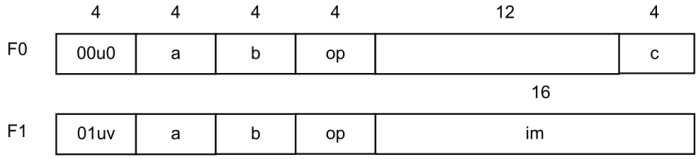

Figure 2: Formats Fo and F1 of RIs

All RIs (with the exception of multiplication and division) have side effects on the 4 single-bit condition registers:

1. N := bit 31 (the highest bit) of result. Hence, N indicates whether the result is negative.

2. Z := all bits of result are zero. Z indicates whether the result is zero.

3. C := carry out bit. (For addition and subtraction only).

4. V := overflow. (For addition and subtraction only)

These 4 condition registers are tested by BIs. The DIV instruction deposits the remainder in an auxiliary register H.

RIs 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.

2.2 *MI: Memory instruction*

There are only 2 MIs, 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.

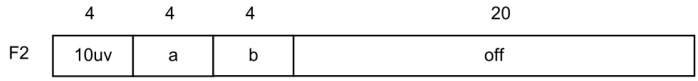

Figure 3: Format F2 of MIs

```
LD a,b,off // R.a := Mem[R.b+off] u=0 LoaD
ST a,b,off // Mem[R.b+off] := R.a u=1 STore
```

The 2nd modifier bit v have the following significance:

```
v=0: word,
v=1: byte. // implemented on RISC-3 only
```

2.3 *BI: Branch instruction*

BIs 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 a word long. 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 Figure 4:

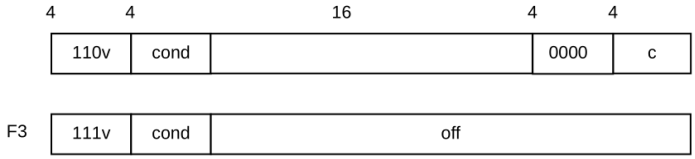

Figure 4: Format F3 of BIs

```
PC := (u = 1) ? PC + 1 + off : R.c; // u = 0
IF v = 1 THEN R15 := PC + 1 END;

B<cond> <dest>; // jump <dest> if <cond> satisfied
```

The following conditions are available:

code cond	mean -ing		code cond	mean -ing	
0000 MI	−Neg.	N	1000 PL	+PLus	~N
0001 EQ	=Zero	Z	1001 NE	$\neq 0$	~Z
0010 CS	CarrySet	C	1010 CC	CClear	~C
0011 VS	oVerflow	V	1011 VC	VClear	~V
0100 LS	L.∣Same	~C∣Z	1100 HI	HIgh	~(•)
0101 LT	$<$	N≠V	1101 GE	$\geq$	~(•)
0110 LE	$\leq$	(•)∣Z	1110 GT	$>$	~(•)
0111	True	T	1111	False	F

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 hardwire 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 **RISCo**, implements the processor.

- 2 submodules implementing a **multiplier** and a **divider**.

- **RISCoTop**, representing the processor's "environment". It is a Verilog rule that only a top module can import off-chip signals. It contains the connections to external devices, including:

  – an **RS-232 transmitter** and an **RS-232 receiver**, representing a serial line.

  – connections to a set of 8 LEDs and to 8 switches.

This configuration is shown in Figure 5:

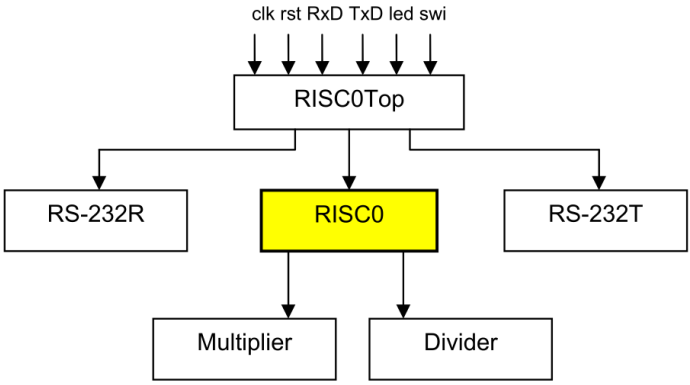

Figure 5: Block diagram of the RISC-0 configuration

The header of module RISCo 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
```

As mentioned earlier, the processor is divided into two parts:

- the ALU, and
- the Control Unit (CU).

In the block diagram of Figure 6 the ALU is to the left, the CU to the right, and it is evident that they are connected by a few wires only.

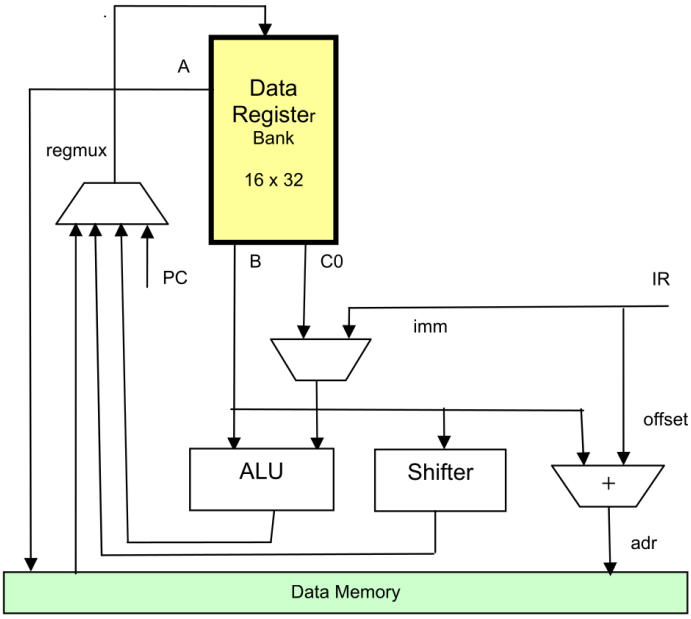

Figure 6: Block diagram of computation unit

The heart of the CPU is the bank of registers, each of the 16 registers with 32 bits. They are represented as an array of 16 words, connected by the following signals (wires).

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

```
wire[31:0] A   // data input  (to registers)
wire[31:0] B   // data output (from registers)
wire[31:0] C0  // data output

wire[3:0] ira0 // address of port A
wire[3:0] irb  // address of port B
wire[3:0] irc  // address of port C0
wire clk       // clock
wire regwr     // write enable
```

Before proceeding, we postulate the following auxiliary signals derived from the Instruction Register (IR), representing instruction fields and their decoding (see instruction formats in Chapter 2).

```
IR = pmout; // instruction from Program Memory (PM)

p = IR[31]; // instruction fields
q = IR[30];
u = IR[29];
v = IR[28];
w = IR[16];

cc = IR[26:24];
op = IR[19:16];

ira = IR[27:24]:
irb = IR[23:20];
irc = IR[ 3: 0];
im  = 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 Figure 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}}, im};

aluRes = MOV
  ? C1
  :  LSL  ? t3       // L-shifter output
  : (ASR
    |ROR) ? s3       // R-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)

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

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

Assignment to the register bank is specified as

```
  R[ira0] <= regwr ? regmux : A;
```

In addition to the data registers, 4 single-bit registers are provided to hold some predicates of the computed result. They are traditionally called Condition Codes (CC), and they are used (tested) by BIs. These registers are N, Z, C, V.

```
N <= regwr ? regmux[31]  : N; // sign
Z <= regwr ? (regmux==0) : Z; // zero
```

N and Z are set by all register operations, C and V by additions and subtractions only. The values for C and V are shown in the detailed program listings. Incidentally, C depends on its interpretation as a carry or a borrow bit.

Parallel to the path from register output to register input through the ALU lies the path through 2 shifters, one for left and one for right. 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, 3 stages are required. The 1st shifts by 0, 1, 2, or 3 bits, the 2nd by 0, 4, 8, or 12 bits, and the 3rd by 0 or 16 bits. The following signals are involved (the shift count is split):

```
wire[31:0] t1,t2,t3, s1,s2,s3; wire[1:0] sc1, sc0;
                                    // shift counts
assign sc0 = C1[1:0]:
assign sc1 = C1[3:2];
```

The shifter input is B, output for LSL is t3. The intermediates between multiplexers are t1 and t2:

```
assign t1 = (sc0==3) ? { B[28:0],  3'b0} :
            (sc0==2) ? { B[29:0],  2'b0} :
```

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

The right shifter is quite similar (output is named s3 instead of t3). It implements both right shift (ASR) and right rotation (ROR) (determined by *u*). Bits shifted into the left end, for ASR, are replicas of the leftmost (sign) bit B[31], whereas for ROR, are repeatedly taken from the rightmost:

```
assign s1 =
 (sc0==3) ? {(u ? B[2:0] : {3{B[31]}}), B[31:3]} :
 (sc0==2) ? {(u ? B[1:0] : {2{B[31]}}), B[31:2]} :
 (sc0==1) ? {(u ? B[0] : B[31]), B[31:1]} : B;
assign s2 =
 (sc1==3) ? {(u?s1[11:0]:{12{s1[31]}}),s1[31:12]}:
 (sci==2) ? {(u?s1[7:0]:{8{s1[31]}}),s1[31:8]}:
 (sci==1) ? {(u?s1[3:0]:{4{s1[31]}}),s1[31:4]}:s1;
assign s3 =
 C1[4] ? {(u?s2[15:0]:{16{s2[31]}}),s2[31:16]} : s2;
```

And this concludes the ALU explanation.

3.2 *DM: Data Memory*

The DM is represented in our case (RISC0) by a so-called Block RAM (BRAM), a facility provided by the FPGA. Fortunately (newer versions of) Verilog compilers allow a memory to be specified as a simple array, mapping the array onto a BRAM:

```
reg[31:0] mem[size-1 : 0];

mem[adr] <= dmin;  // outbus A
dmout <= mem[adr];
```

The memory uses the following signals:

```
wire[31:0] dmin, dmout; // input and output ports
wire wr;                // write enable
wire[13:0] adr;         // 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).

3.3 *CU: Control Unit*

The CU 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 BRAM 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 4 cases:

1. Zero on reset;
2. The next address is the current plus 1 (i.e. $PC + 1$);
3. Given by the instruction explicitly (BIs);
4. Taken from a register, (which is used for returning from procedures).

In a Harvard architecture, ALU and CU operate simultaneously. In each clock cycle, the CU fetches the next instruction from the PM into the IR, and the ALU operates on registers or the DM. 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 CU is quite straight-forward, as show in Figure 7, and defined by the following expressions:

```
IR = pmout;
```

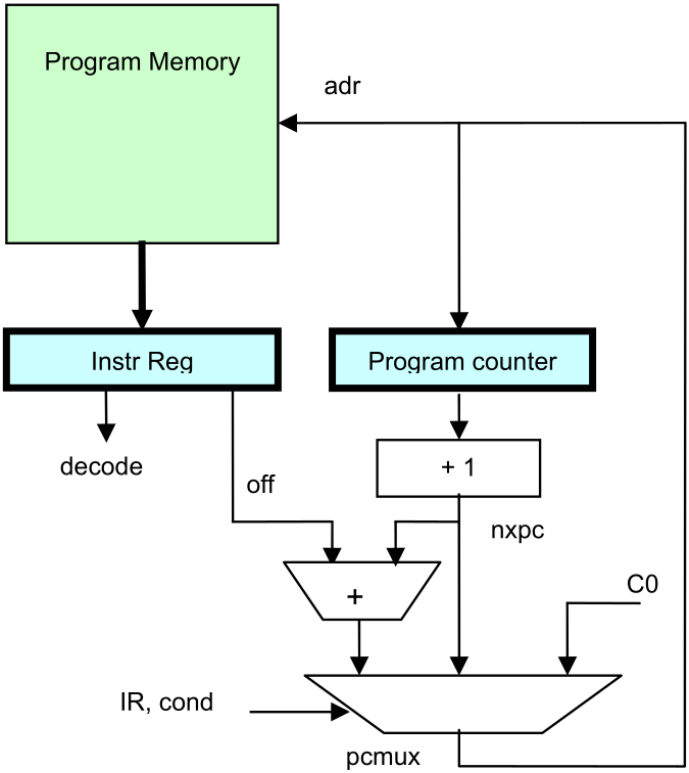

Figure 7: The control unit

```
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 BIs 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) &     V | // VS, VC
   (cc == 4) & (C|Z) | // LS, HI
   (cc == 5) &     S | // LT, GE
   (cc == 6) & (S|Z) | // LE, GT
   (cc == 7));         //  T, F
```

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 multiplexer. 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;
```

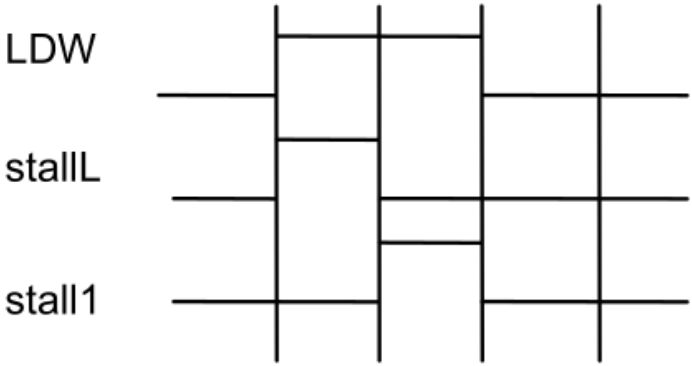

Figure 8: Stall generation for memory access

This circuit actually represents a very simple state machine.

3.4 *multiplier*

Multiplication is an inherently more complex operation than either addition or subtraction. After alll, multiplication can be composed (of a sequence) of additions. There are many methods to implement multiplication, all 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 concur-

rently. They are fast, but the amount of required circuitry is high.

Here we present 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 $p := xy$. 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:

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

We obtain the product by a sequence of 32 additions, each term of the form $x_k 2^k y$, i.e. of y left shifted by k positions multiplied by x_k. Since x_k is either 0 or 1, the term is either 0 or y. Multiplication is thus performed by an adder and a selector. Instead of using x, as a selector value, A (initially equal to x) is shifted right in each so-called *add-shift* step, so that A_0 is always the selector. It is represented by $ODD(A)$. The algorithm is shown by the following piece of program. B must be a variable of double length:

```
B := 0; A := x; i := 0;

REPEAT
  IF ODD(A) THEN B := B+y; A := A-1 END;
  B := 2 * B; A := A DIV 2; i := i+1
UNTIL i = 32
```

The invariant $Ay + B = xy$ is maintained over the iterations, and at the end $A = 0$, yielding the desired result $B = xy$. $A := A-1$ can be omitted due to the subsequent division.

Let us now transpose this algorithm into hardware. Instead of shifting B to the left, we shift it to the right, adding y to B's upper half. This has the advantage of using the concatenated B and A, denoted as P (product), as a double register. In step i, $P[63 : 32 - i]$ is the partial product, and $P[31 - i : 0]$ is the multiplier. The size of the multiplier decreases by 1 in each step, whereas the size of the product increases by 1.

The multiplier is controlled by a rudimentary state machine S, actually a simple counter running from 0 to 32. Step 0 is for initialization. The multiplier is shown schematically in Figure 9.

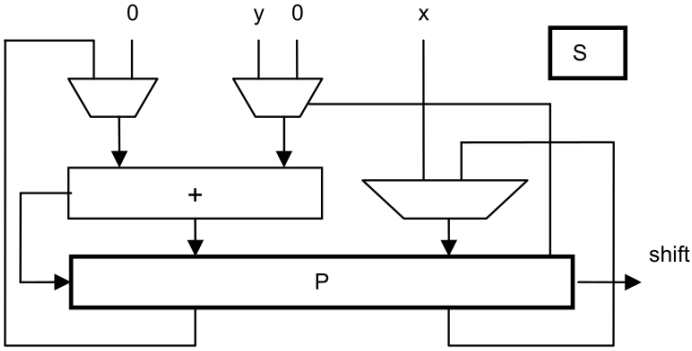

Figure 9: Schematic of multiplier

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 most significant term has a negative weight ($-x_{31}2^{31}$). Therefore, implementation of signed multiplication requires

very little change: Term 31 is subtracted instead of added (see complete program listing below).

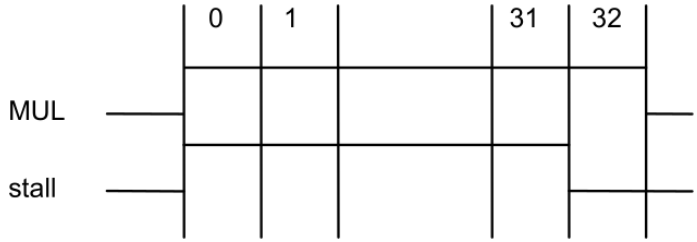

Figure 10: Generating stall

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 == 33);
S <= MUL ? S+1: 0;
```

The complete multiplier is listed below, $u = 0$ specifies singned, $u = 1$ unsigned operands:

```
module Multiplier(
  input clk, run, u,
  output stall,
  input [31:0] x, y,
  output[63:0] z
);
reg[ 5:0] S;    // state
reg[63:0] P;    // product
wire[31:0] w0;
```

```
wire[32:0] w1;

assign stall = run & ~(S == 33);
assign w0 = P[0] ? y : 0;
assign w1 = (S==32)&u ? {P[63],P[63:32]}-{w0[31],w0}
                      : {P[63],P[63:32]}+{w0[31],w0};
assign z = P;

always @ (posedge(clk)) begin
  P <= (S == 0) ? {32'b0, x}: {w1[32:0], P[31:1]};
  S <= run ? S+1 : 0;
end
endmodule
```

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

3.5 *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. Here we discuss division of unsigned integers

only with the following definitions. x is the dividend, y the divisor.

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

where q is the quotient, r the remainder, satisfying

$$x = qy + r \text{ with } 0 \leq r < y$$

The division algorithm is shown by the following piece of program with variables R and Q:

```
R := x; Q := 0; i := 0;
REPEAT R := R DIV 2; Q := 2*Q; i := i+1;
  IF R>=y THEN R := R-y; Q := Q+1 END
UNTIL i = 32
```

The invariants $Qy + R = x$ and $0 \leq R < y2^{32-i}$ are maintained over the iterations, with $i = 32$ yielding the desired result.

When transposing the algorithm into hardware, we use the same trick of using a double register RQ as in multiplication. Initially we set RQ to x, the dividend, and then subtract multiples of y from it, each time checking that the result is not negative. Initially, R may be a 64-bit value initially; its size decreases by 1 in each step. Q is initially 0, and its size increases by 1 in each step. $(R, Q[31 : i])$ holds the remainder, $Q[i - 1 : 0]$ holds the quotient.

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

```
module Divider(
```

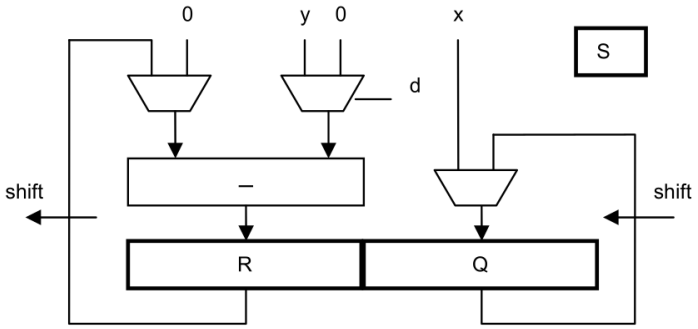

Figure 11: Schematic of divider

```
  input clk, run, u,
  output stall,
  input [31:0] x, y,        // y > 0
  output[31:0] quot, rem
);
reg [ 5:0] S;               // state
reg [63:0] RQ;
wire       sign;
wire[31:0] x0, w0, w1;

assign stall = run & ~(S == 33);
assign sign = x.31 & u;
assign x0 = sign ? -x : x;
assign w0 = RQ[62:31];
assign w1 = w0 - y;

always @(posedge(clk)) begin
  RQ <= (S == 0) ? {32'b0, x0} :
        {(w1[31] ? w0 : w1), RQ[30:0], ~w1[31]}:
  S <= run ? S+1 : 0;
end

assign quot = sign ? -RQ[31:0] - 1 : RQ[31:0];
```

```
assign rem  = sign ? y - RQ[63:32] : RQ[63:32];
endmodule
```

3.6 *environment*

Module RISCo is imported by RISCoTop, which thereby represents the processors 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:

1. A counter incremented every millisecond ($adr : 0$)
2. A set of 8 dip switches ($adr : 4$)
3. A set of 8 Light Emitting Diodes (LED) ($adr : 4$)
4. An RS-232 receiver (serial data line) ($adr : 8, 12$)
5. An RS-232 transmitter ($adr : 8, 12$)

The RS-232 receiver and transmitter are described in subsequent sections, 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 Figure 5).

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

3.7 *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 Figure 12). 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.

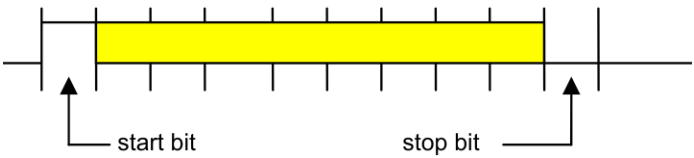

Figure 12: RS-232 packet format

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 & 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
```

3.8 *receiver*

The RS-232 receiver is structured very similarly with 2 counters and a shift register. The state machine is triggered by an incoming start bit at RxD. For the purpose of robustness, a synchronizer and edge detector are provided for the incoming signal. 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 input is sampled in the middle of the bit period rather than at the end, namely when $midtick = endtick/2$.

```
module RS232R(
  input clk, rst, RxD,
  input       done, // "byte has been read"
  output      rdy,
  output[7:0] data
);
wire endtick, midtick;
reg       run, stat;
reg       Q0, Q1; // synchronizer & edge detector
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
  Q0 <= RxD; Q1 <= Q0;
  run <= (Q1 & ~Q0) ? 1 :
         (~rst | endtick & endbit) ? 0 : run;
  tick <= (run & ~endtick) ? tick + 1 : 0;
  bitcnt <= (endtick & ~endbit) ? bitcnt+1 :
            (endtick &  endbit) ? 0 : bitcnt;
  shreg <= midtick ? {Q1, shreg[7:1]} : shreg;
  stat <= (endtick & endbit) ? 1 :
                (~rst | done) ? 0 : stat;
end
endmodule
```

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.

4.1 *RISC1: Using large SRAM*

Before proceeding to a conversion of our RISC to the von Neumann scheme, we show the replacement of the small

2KB BRAM for data storage by the large 1 MB 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 doubled from 12 to 24 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 ford 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[23:0] 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, inbusO, 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[23:6] == 18'h3FFFF):
assign inbus = ~ioenb ? inbuso :
       ((iowadr == 0) ? cnt :
        (iowadr == 1) ? swi :
        (iowadr == 2) ? {24'bO, dataRx} :
        (iowadr == 3) ? {30'bO, 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 SAbuf. It is required, because the SRAM is connected with the FPGA by a single, bi-directional bus SAdat. The circuits are shown schematically in Figure 13.

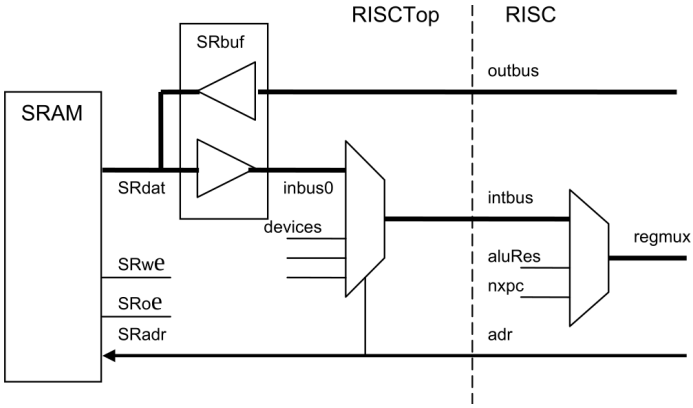

Figure 13: Connections between processor and SRAM

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 2nd 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 2nd half of

the cycle. The various enable signals *ce*0, *ce*1, are all active (low). (Note that like in RISCo 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.

4.2 *RISC2: SRAM for both program and data*

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 inbuso 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 24 bits. Because instructions are always 4 bytes long, their address is always a multiple of 4. This concerns the declarations

```
output[23:0] adr;
reg   [21:0] PC;
wire  [21:0] pcmux, nxpc;
```

In addition, the following changes are necessary:

```
regmux =... (BR & v) ? {8'bO,nxpc,2'b0} : aluRes;
pcomux =... (BR & cond & ~u) ? CO[23:2] : nxpc;
```

When using the on-chip BRAM for PM, the IR 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 *IR*.

```
reg [31:0] IR;
wire[31:0] ins;
ins = IR;
IR <= 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 IR. 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 RIs in a von Neumann architecture, but not for MIs, because a data access would interfere with the fetching of the next instruction. Therefore, a stall cycle must be introduced for MIs, 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*, while in the latter it is $B + off$ (see Figure 14). 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[23:0] + {4'b00, off}
                   : {pcmux, 2'b00};
assign wr = STW & ~stall1;
assign rd = ~wr;
```

```
regwr = ccwr | (LDW & ~stall1);
```

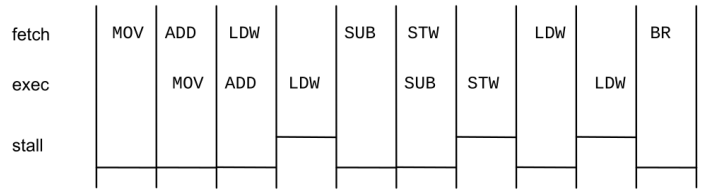

Figure 14: Stalling memory instructions

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.

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 = 22'h3FF800;
IRBuf <= stall ? IRBuf : codebus;
PMsel <= ~rst | (pcmux[18:11] == 8'hFF;
ins = PMsel ? pmout: IR;
```

The FPGA's BRAMs 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.

The only addition to the processor interface is the signal *ben* (byte enable) derived from the modifier bit v in MIs.

For byte-wise reading, multiplexers are inserted into the input path. We rename the IOBuf's output from *inbus* to *inbus*0, and redefine *inbus* as

```
inbusL = (~ben | a0) ? inbus[ 7: 0] :
                 a1  ? inbus[15: 8] :
                 a2  ? inbus[23:16] :
                       inbus[31:24];
inbusH = ~ben ? inbus[31:8] : 24'b0;
inbus = {inbusH, inbusL}:

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.

```
assign obB0 = A|7:0];
assign obB1 = ben & a1 ? A[7:0] : A[15: 8];
assign obB2 = ben & a2 ? A[7:0] : A[23:16];
assign obB3 = ben & a3 ? A[7:0] : A[31:24];
assign outbus = {obB3, obB2, obB1, obB0};
```

The various chip- and byte-enable signals are defined as (see also 4.1):

```
SRced = be &  adr[1];
SRce1 = be & ~adr[1];
SRbe0 = be &  adr[0]:
SRbe1 = be & ~adr[0];
SRbe[2] = {SRce1, SRce0, SRbe1, SRbeO}:
SRadr = adr[19:2];
```

4.4 *Interrupts*

The last facility to be added is 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 processes 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 3 new instructions, one for returning from an interrupt routine, and 2 for setting the enable state. They are all encoded in the form of BIs with $u = 0$:

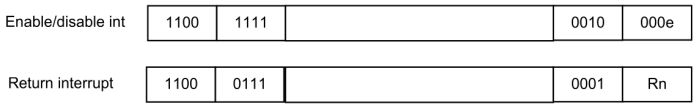

Figure 15: Special instructions for interrupt handling

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       irq;    // INT request
reg         irq1;   // edge detector
reg         intMd, intEnb, intPnd;
                    // INT mode,enable,pending
reg [25:0]  SPC;    // saved PC & CC on INT
wire[21:0]  pcmux0;
wire        intAck; // INT request enabled
wire        nn, zz, cx, vv;     // CC
wire        RTI;    // return from II
```

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.

```
intAck = intPnd & intEnb & ~intMd & ~stall;
intPnd <= rst & ~intAck & ((~irq1 & irq) | intPnd);
pcmux = ~rst ? 0: intAck ? 1 : pcmux0;

N <= nn; Z <= zz; C <= cx; OV <= vv;
SPC <= intAck ? {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[21:0] and the condition bits from SPC[25:22]. Furthermore, the processor leaves the interrupt mode.

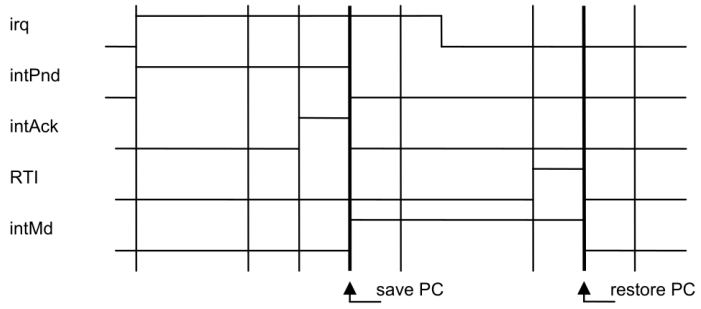

Figure 16: Interrupt signals

```
RTI = p & q & ~u & IR[4];
pcmux0 = stall ? PC : RTI ? SPC[21:0] : ...
pcmux  = ~rsc ? StartAdr : intAck ? 1 : pcmux0;

nn = RTI ? SPC[25] : ccwr ?  aluRes[31]       : N;
zz = RTI ? SPC[24] : ccwr ? (aluRes[31:0]==0) : Z;
```

The values for *cx* and *vv* are shown in the detailed program listings. The *interrupt mode instruction* copies *ins*[0] into the *intEnb* register, which is cleared on reset.

```
intEnb <=                  ~rst ?     0  :
       (BR & ~u & ~v & ins[5]) ? ins[0] : intEnb;
```

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 RISCTop is extended as follows. When *cnt*0 reaches a limit, *cnt*1 is incremented and *irq* is set to 1. This causes an interrupt every millisecond, given a clock frequency of 25 MHz.

```
assign limit = (cnt0 == 24999);

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

V5.12.10

A.1 *Resources and registers*

From the viewpoints of the programmer and the compiler designer the computer consists of an Arithmetic Unit (AU), a Control Unit (CU) and a store (memory).

- The AU contains 16 registers R0-R15, with 32 bits each.
- The CU consists of the Instruction Register (IR), holding the instruction currently being executed, and the Program Counter (PC), holding the word-address of the instruction to be fetched nextly. All Branch Instructions (BI) are conditional.
- The memory consists of 32-bit words, and it is byte-addressed. Furthermore, there are 4 flag registers N, Z, C and V, called the Condition Codes (CC).

There are 3 types of instructions and 4 instruction formats (F0-F3):

- Register Instructions (RI) operate on registers only and feed data through a shifter or the Arithmetic Logic Unit (ALU) (formats F0 and F1).
- Memory instructions (MI) fetch and store data in memory (format F2).
- BIs affect the PC (format F3).

A.2 *RI*

RIs assign the result of an operation to the destination register $R.a$. The 1st operand is the register $R.b$ and the 2nd operand n is either register $R.c$ (format F0) or a constant im (format F1).

Immediate values are extended to 32 bits with 16 v-bits to the left. Apart from $R.a$, these instructions also affect the flag registers N(egative) and Z(ero). The ADD and SUB instructions also set the flags C(arry, borrow) and V (oVerflow).

A.3 *MI*

If $v = 0$, access is for a word (4 bytes). If $v = 1$, a single byte is accessed.

A.4 *BI*

If $u = 0$, the destination address is taken from register $R.c$. If $u = 1$, it is $PC + 1 + offset$. If $v = 1$, the link address $PC + 1$ is deposited in the register $R15$.

A.5 *Special features*

Modifier bit $u = 1$ changes the effect of certain instructions as follows:

```
ADD', SUB' add, subtract also carry C
MUL' unsigned multiplication
MOV' form 0, v = 0: R.a := H
MOV' form 0, v = 1: R.a := [N, Z, C, V]
```

```
MOV' form 1 R.a := [im 16'b0] (im left shifted 16 bits)
```

The MUL instruction deposits the high 32 bits of the product in the auxiliary register H. The DIV instruction deposits the remainder in H.

A.6 *Interrupts*

The addition of an interrupt facility required the addition of 2 new instructions, as well as the status register *intenb* (interrupt enable). The instructions are

```
RTI     1100 0111 xxxx xxxx xxxx xxxx 0001  Rn
                        // return from interrupt
STI/CLI 1100 1111 xxxx xxxx xxxx xxxx 0010 000e
            // set/clear interrupt, intenb := e
```

These instructions are encoded as BIs with bits 4 or 5 set.

A.7 *RISCo*

The RISC architecture has been implemented on a Xilinx FPGA contained on the development board Spartan. RISCo stands at the origin of an evolving series of extensions. It represents a Harvard architecture, and it uses FPGA-internal RAM for its memory, which is restricted to 8K words of program and 8K words for data. It does not feature byte access, and the floating-point instructions are not available.

RISCo's external devices are the following:

1 bit 0: receiver ready, bit 1: transmitter ready.

adr	hex	input	output
-64	0FFFC0H	millisecond counter	
-60	0FFFC4H	switches	LEDs
-56	0FFFC8H	RS-232 data	RS-232 data
-52	0FFFCCH	RS-232 status[1]	RS-232 control

A.8 *RISC5*

RISC5 is an extension of RISC0 based on a von Neumann architecture and uses the same instruction set. The memory consists of the board-internal SRAM with a capacity of 1 MB. Byte access is available, and so are floating-point instructions.

RISC5's external devices are the following:

adr	hex	input	output
-64	0FFFC0H	millisecond counter	
-60	0FFFC4H	switches	LEDs
-56	0FFFC8H	RS-232 data	RS-232 data
-52	0FFFCCH	RS-232 status	RS-232 control
-48	0FFFD0H	disk, net SPI data	SPI data
-44	0FFFD4H	disk, net SPI status	SPI control
-40	0FFFD8H	keyboard data (PS2)	
-36	0FFFDCH	mouse (and kbd status)	

A further added device is the video controller. It maps memory at 0E7F00H - 0FFEFFH onto the display (1024 x 768 pixels).

1. Xilinx, Spartan-3 Starter kit board user guide

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