An Approach to Reducing the Number of Instructions for Conditional Branches in FPGA Based 8 bit RISC SMCore Microcontroller

Ivan Kanev

Abstract. *This article presents an approach to reducing the number of instructions that organize conditional branches in designing a FPGA-based 8-bit RISC SMCore microcontroller. The logical conditions for setting the flags of the status register are defined. A method for organizing all conditional branches with only two instructions is offered, the architectural features of the microcontroller being considered. The components realizing the "Skip" mechanism in VHDL are shown.*

Keywords*: Conditional, Branch, FPGA, VHDL, Reduced, Instruction, Set, 8 bit, Microcontroller, SMCore.*

1. INTRODUCTION

Organizing conditional branches plays an important role in the designing of 8-bit FPGA-based microcontrollers. Many of the existing general purpose microcontrollers either partially solve this problem [1] or use multiple instructions [2], [3]. This is due to the desire of general purpose single chip microcontroller manufacturers to ensure programming and architectural inheritance in developing their sophisticated families.

The design of mechanisms and instructions for conditional branches in FPGA-based systems with limited resources and reduced instruction sets may happen to require a number of instructions comparable to the number of all other instructions in the set.

The research conducted in this article refers to an 8-bit RISC SMCore microcontroller [4] and aims at:

- To examine the logical conditions for setting the flags of the status register and the possibilities for organizing branches with operations on signed or unsigned operands.
- To offer a method allowing the reduction of the number of instructions for conditional branches, having made an analysis of the conditions for branches.
- To offer suitable mechanisms and instructions for organizing conditional branches in VHDL.

2. ARCHITECTURAL FEATURE OF THE 8-BIT RISC SMCore MICROCONTROLLER

The main blocks and components that form the architectural model of the SMCore microcontroller are shown in fig. 1.

The microcontroller is based on the Harvard architecture and has the following most important features:

1.All system, input/output registers and general purpose registers are eight bits long and are unified in a single 256-byte File Register (FR).

For the purpose of this research, we assume that the accumulator "A" and the status register (SR) are integrated into the system registers, on certain addresses in the FR. The output of the accumulator form the Acc bus. *The bits of the SR, also called flags, are set in accordance to the results of a certain operation:*

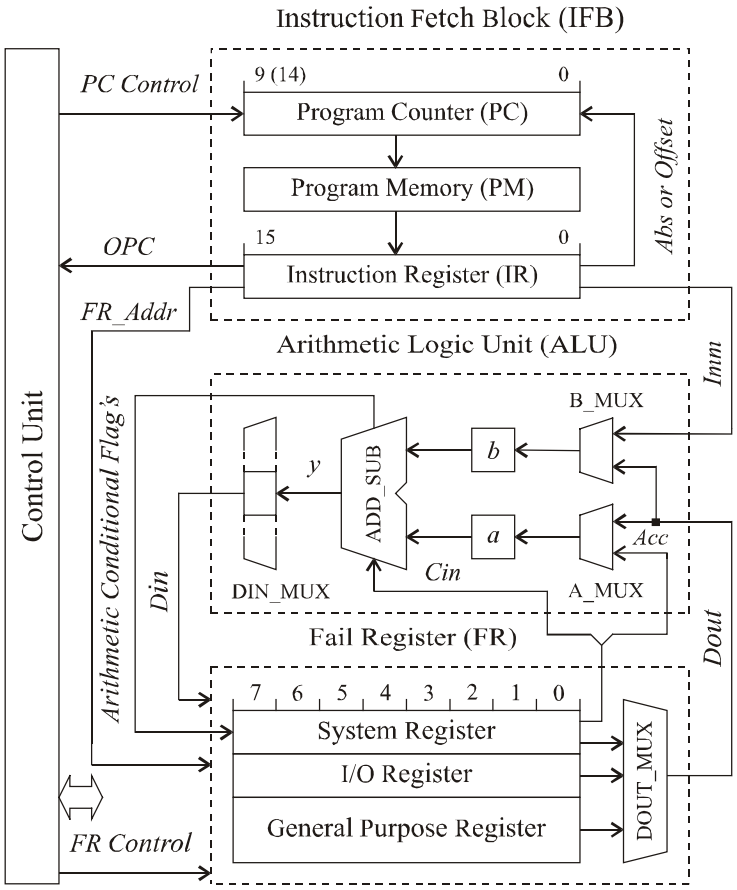

Fig. 1. SMCore – architectural model

Carry-Borrow; Zero; Overflow; Negative i.e. The SR flags are used in organizing a large number of conditional branches in the microcontroller. The output of the Carry-Borrow flag forms the Cin signal, which is used in addition or subtraction, taking into account the results of previous arithmetic operations.

The outputs of the FR registers are multiplexed by DOUT_MUX and form the DOUT bus.

2. Every instruction is coded in the program memory (PM) of the microcontroller in a single 16-bit program word. The instructions are fetched from the PM by a 9-bit program counter (PC). The PC can be extended up to 14 bits. The current instruction is stored in the instruction register (IR). The operation code (OPC) of the instructions is decoded in the Control Unit. This Unit generates all signals controlling the microcontroller.

Based on the addressing modes used, the operands of the current instruction can be transported to the other units by three buses: IMM, when immediate addressing is used; AbsorOffset, when absolute or relative addressing is used; FR_Addr, when register - register is used.

3. All arithmetic, logical, bit and transitive operations are realized in the ALU. The outputs of the ALU components are multiplexed by DIN_MUX and form the *Din* bus. Fig.1 only shows the part of the ALU that is related to the problems of forming the logical conditions for setting the SR flags.

3. LOGICAL CONDITIONS FOR SETTING STATUS REGISTER FLAGS

We use the following notation:

m	–	order of the operands of the microcontroller;
a, b	–	input ALU registers;
y	–	output of the addition/subtraction component;
Din	–	ALU output.
$\Rightarrow$	–	branch;

Then the Status Register flags are set with the following logical functions [4]:

- Zero results of operations with signed or unsigned operands:

$$Z = \bigwedge_{i=0}^{m-1} \overline{Din_i}\,. \tag{1}$$

- Negative result of operations with signed operands

$$N = Din_{m-1} \tag{2}$$

- Carry or Borrow resulting from arithmetic operations with unsigned operands:
  - addition,

$$C = a_{m-1}.b_{m-1} \vee a_{m-1}.\overline{y}_{m-1} \vee b_{m-1}.\overline{y}_{m-1}; \tag{3}$$

  - subtraction,

$$C = \overline{a}_{m-1}.b_{m-1} \vee \overline{a}_{m-1}.y_{m-1} \vee b_{m-1}.y_{m-1}. \tag{4}$$

If the current operand is contained within the "b" register:

  - left shift/left rotation,

$$C = b_{m-1} \tag{5}$$

  - right shift/right rotation

$$C = b_0 \tag{6}$$

- Arithmetic overflow resulting from arithmetic operations with signed operands:
  - addition,

$$V = a_{m-1}.b_{m-1}.\overline{y}_{m-1} \vee \overline{a}_{m-1}.\overline{b}_{m-1}.y_{m-1}; \tag{7}$$

  - subtraction,

$$V = \overline{a}_{m-1}.b_{m-1}.y_{m-1} \vee a_{m-1}.\overline{b}_{m-1}.\overline{y}_{m-1}. \tag{8}$$

The *C, Z, V, N* flags can realize eight conditional branches.

$$C = \begin{Bmatrix} 0 \Rightarrow Not\ Carry \\ 1 \Rightarrow Carry \end{Bmatrix}; \tag{9}$$

$$Z = \begin{Bmatrix} 0 \Rightarrow Not\ Zero \\ 1 \Rightarrow Zero \end{Bmatrix}; \tag{10}$$

$$V = \begin{Bmatrix} 0 \Rightarrow Not\ Overflow \\ 1 \Rightarrow Overflow \end{Bmatrix}; \tag{11}$$

$$N = \begin{Bmatrix} 0 \Rightarrow Positive \\ 1 \Rightarrow Negative \end{Bmatrix}. \tag{12}$$

Let's assume *a, b* are unsigned operands and the following operation is performed $SUB_{USG} \rightarrow$ a-b. Then, referring to (1), (4) we can define the conditions for the branches and the concrete values of the flags *C* and *Z* after the SUB_{USG} operation is performed:

$$
\begin{aligned}
&if\ a = b\ then\ Z = 1, C = 0;\\
&if\ a > b\ then\ Z = 0, C = 0;\\
&if\ a \geq b\ then\ Z = 1, C = 0;\\
&if\ a < b\ then\ Z = 0, C = 1;\\
&if\ a \leq b\ then\ Z = 1, C = 1.
\end{aligned}
$$

The *C* and *Z* flags can realize six conditional branches:

$$C = \begin{Bmatrix} 0 \Rightarrow Greater\ or\ Equal \\ 1 \Rightarrow LessThan \end{Bmatrix}; \tag{13}$$

$$Z = \begin{Bmatrix} 0 \Rightarrow Not\ Equal\ to \\ 1 \Rightarrow Equal\ to \end{Bmatrix}; \tag{14}$$

$$C \vee Z = \begin{Bmatrix} 0 \Rightarrow Greater\ Than \\ 1 \Rightarrow Less\ or\ Equal \end{Bmatrix}. \tag{15}$$

Let's now assume that *a and b* are signed operands and the following operation is performed $SUB_{SG} \rightarrow$a-b. Then, referring to (1), (2), (8) we can define the conditions for the branches and the values of the *Z*, *N*, *V* flags after the SUB_{SG} operation is performed. We can substitute the concrete values of *N* and *V* with the $N \oplus V$ function, which is zero if $a \geq b$:

While defining the conditions for the branches after the SUB_{SG} operation, we can omit the ones, where $a = b$ and $a \neq b$, because the *Z* flag does not depend on the sign of the operand (1).

$$
\begin{aligned}
&if\ a \geq b\ then\ Z = 1, N \oplus V = 0\\
&if\ a > b\ then\ Z = 0, N \oplus V = 0\\
&if\ a < b\ then\ Z = 0, N \oplus V = 1\\
&if\ a \leq b\ then\ Z = 1, N \oplus V = 1
\end{aligned}
$$

We can realize four conditional branches with the *Z*, *N and V* flags.
If:

$$N \oplus V = \begin{Bmatrix} 0 \Rightarrow Greater\ or\ Equal \\ 1 \Rightarrow Less\ Than \end{Bmatrix}; \tag{16}$$

$$Z \vee (N \oplus V) = \begin{Bmatrix} 0 \Rightarrow Greater\ Than \\ 1 \Rightarrow Less\ or\ Equal \end{Bmatrix}. \tag{17}$$

2. Prerequisites for reduction of the number of instructions for conditional branches

Based on the analysis of the conditions for organizing conditional branches, the following conclusions can be drawn:

1. We can organize 18 branches with the *C, Z, V* and *N* flags using signed or unsigned operands.

2. Branches (9) and (10) can be combined with branches (14) and (15) respectively as they use the same flags.

3. Twelve branches can be organized by checking only one flag. In order to organize the other six branches, conditions formed as logical functions (15), (16), (17) containing the *C, Z, V, N* flags will have to be checked.

Table 1. Status Register - configuration, flags, branches.

Flags			Branches
0	*C*	0	Carry Clear; (Greater or Equal)$_{usg}$
		1	Carry Set; (Less Than)$_{usg}$
1	*Z*	0	Not Zero;(Not Equal to)$_{usg\ or\ sg}$
		1	Zero, (Equal to) $_{usg\ or\ sg}$
2	*V*	0	Not Overflow
		1	Overflow
3	*N*	0	Positive
		1	Negative
4	*CF0*	0	(Greater Than) $_{usg}$
		1	(Less or Equal) $_{usg}$
5	*CF1*	0	(Greater or Equal) $_{sg}$
		1	(Less Than) $_{sg}$
6	*CF2*	0	(Greater Than) $_{sg}$
		1	(Less or Equal) $_{sg}$
7	---		

The idea of reducing the number of instructions for conditional branches is based on the assumption that all branches can be organized by checking only one of the Status Register flags for every branch. In this case, the number of instructions for organizing conditional branches can be reduced to two instructions checking the SR flags as bit operands.

Let's assume that the SR flags are divided into two groups – base flags and complex flags, where:

- *Base flags are the ones which can be used to organize conditional branches checking only one of the C, Z, V and N flags.*
- *Complex flags are those set by logical functions including two or more base flags:*

$$CF0 = C \vee Z; \quad (18)$$

$$CF1 = N \oplus V; \quad (19)$$

$$CF2 = Z \vee (N \oplus V). \quad (20)$$

Then, if the base and complex flags are integrated in a single status register, the efforts to organize conditional branches can be reduced to checking bit operands.

An example configuration of a status register containing base and complex flags is shown in Table 1. The branches that can be organized after the SUB_{USG} and SUB_{SG} operations are denoted in brackets.

5. SKIP MECHANISMS AND INSTRUCTIONS FOR ORGANIZING CONDITIONAL BRANCHES

For the architectural model chosen, we assume that every instruction is coded in the program memory in a single 16-bit program word. Then, the form of instruction coding proves to be significant for the choice of mechanisms for organizing conditional branches. Let's assume the following notations:

OPC_{cb}	–	code of the operations for conditional branches;
$< Rn >$	–	operand: address of register *n* from the FR, *n* = 0..255;
$<\#Bit>$	–	immediate operand: number of the bit of *Rn* (#Bit = 0..7);
$< Branch\ Addr >$	–	operand: address of the branch
$(\mathrm{Re}\, g)$	–	register content
$\leftarrow$	–	data transfer

In order to organize conditional branches on bit operands on all *Rn* registers, the instructions for conditional branches have to be encoded in the following way:

$$OPC_{cb} < Rn >,\ <\#Bit >,\ < Branch\ Addr > \qquad (21)$$

Taking into account the number of combinations needed to encode the operations of the instruction set and the fact that <*#Bit*> has to be indicated when checking bit operands, it is obvious that the form (21) cannot be realized in a single 16-bit program word.

Therefore, mechanisms allowing indirect coding of one of the operands $< Branch\ Addr >$ *and* $< Rn >$ *have to be chosen, using the concrete architectural model.* A proper solution for indirect coding of $< Branch\ Addr >$ is the *Skip* mechanism.

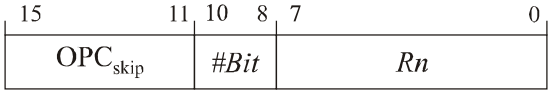

Fig.2. Format of the *Skip* instructions

Let's assume the instruction for conditional branch has been fetched from program memory and the content of the Program Counter is (PC). Then if the condition checked is true, the instruction located immediately after the current instruction (PC+1) is missed and the next instruction located on address (PC+2) is executed.

Let Rn[#Bit] denote a random bit of Rn. The *Skip* mechanism can be described by the following operator for organizing conditional branches:

$$\begin{aligned} &\texttt{if Rn[\#Bit]= " test bit value" then(Pc)}\leftarrow\texttt{(Pc+2)-- \{0 or 1\}} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\texttt{else(Pc)}\leftarrow\texttt{(Pc+1)} \end{aligned} \qquad (22)$$

The *Skip* mechanism can be implemented with two instructions:

Test Rn Bit and Skip if Set $< \mathrm{Rn} >, <\#\mathrm{bit} >$

$$\begin{aligned} &\texttt{-- Description:} \\ &\texttt{if Rn[\#bit]=1 then (PC)}\leftarrow\texttt{(PC+2)} \\ &\qquad\qquad\texttt{else (PC)}\leftarrow\texttt{(PC+1)} \end{aligned} \qquad (23)$$

Test Rn Bit and Skip if Clear $< \mathrm{Rn} >, <\#\mathrm{bit} >$

$$\begin{aligned} &\texttt{-- Descrition:} \\ &\texttt{if Rn[\#bit]=0 then (PC)}\leftarrow\texttt{(PC+2)} \\ &\qquad\qquad\texttt{else (PC)}\leftarrow\texttt{(PC+1)} \end{aligned} \qquad (24)$$

Instructions (23) and (24) can be used to organize conditional branches with all bits (#Bit) of all Rn. In case that the base and complex flags are integrated into the SR (Table) and $\mathrm{Rn}[\#\mathrm{bit}] = \mathrm{SR}[\#\mathrm{bit}]$ *, these instructions can be used to organize 18 conditional branches on signed or unsigned operands.*

The microcontroller components needed for the *Skip* mechanism implementation are shown on Fig.3.

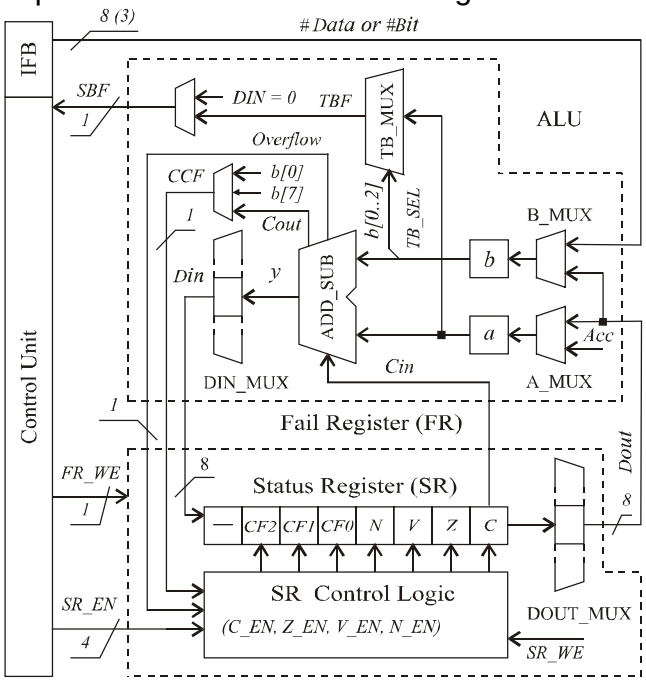

Fig. 3. Status Register. Multiplexer for checking bit operands

The SR description in VHDL is shown in program **<<1>>**. It consists of three fragments:

<< *1.1* >> If SR receives the result of the currently executed instruction, FR generates the sr_we signal, which allows sending Din to SR. If $Din = 0$ then the *Z* flag must be set.

<<*1.2*>> Otherwise, the *C, Z, V, N* triggers of a buffer of the base SR flags are set. A change in the state of these triggers is only possible if the currently executed instruction can affect the SR flags. CU generates the signals $C\_EN, Z\_EN, \_V\_EN, N\_EN$ while decoding every instruction. Those signals are used as a condition allowing the change of *C, Z, V, and N. A change in the state of one or more of those triggers sets the trigger* sr_cf .

```
--<<1>> VHDL Program "Status Register"
process     (clk, reset,din,sr, Cout,
      Vout,sr_we,c_en,z_en,v_en,n_en)
begin
if reset = '1' then sr <= "00000000";
elsif (clk'event and clk = '0') then
-- << 1.1 >> write din in sr
  if (sr_we = '1')then --if sr is dest
      sr <= din;
      if (din = ("00000000") and
                (z_en = '1')
      -- set actual Z flag
       then Z <= '1';    sr_cf <= '1';
       else Z <= '0';    sr_cf <= '1';
       end if;
  else
-- << 1.2 >> set status register buffer
    if c_en = '1'
      then  C <= CCF; sr_cf <= '1';
    end if;
    if (din = ("00000000") and
              (z_en = '1')
      then  Z <= '1';  sr_cf <= '1';
      else  Z <= '0';    sr_cf <= '1';
    end if;
    if v_en = '1'
     then  V <= Overflow;sr_cf <= '1';
    end if;
    if n_en = '1'
      then  N <= din(7); sr_cf <= '1';
    end if;
  end if;
-- << 1.3 >> set status register if
-- change one or more base flag's
  -- executing in next clk

  if sr_cf = '1' then
  -- set base flag's {C,Z,V,N}
      sr(0) <= C;
      sr(1) <= Z;
      sr(2) <= V;
      sr(3) <= N;
  -- set complex flag's {CF0,..,CF2}
      sr(4) <= Z or C; -- CF0
      sr(5) <= N xor V; -- CF1
      sr(6) <= Z or ( N xor V );-- CF2
  -- reset sr change flag
     sr_cf <= '0';
  end if;
end if;
end process;
```

<< *1.3* >> If there is a change in the buffer of the base flags $(sr\_cf = '1')$, the SR flags are set during the next cycle of clk . The content of the C, Z, V, N triggers is stored in the base flags of SR.

The complex flags $CF0,..,CF2$ are set in conjunction with the logical functions defined in (20),.. (22). Then, sr_cf is cleared.

A multiplexer (TB_MUX) integrated in the ALU (fig. 3) can be used to implement the checking of bit operands. Let's assume that the input ALU registers are set with:

$(b) \leftarrow (\#Bit)$

$(a) \leftarrow (Rn) --$ *all Rn including SR*.

Then (TB_MUX) can be realized with the following function:

$$tbf = \overline{b}_2.\overline{b}_1.\overline{b}_0.a_0 \vee \overline{b}_2.\overline{b}_1.b_0.a_1 \vee,..,\vee b_2.b_1.b_0.a_7$$

Obviously, the first conjunction of the tbf function tests the zero bits of all Rn. If the *a* register is set with SR, then the first conjunction of the tbf function corresponds to the current state of the *C* flag. The other conjunctions of the tbf function can be used to test the other bits of Rn and the other SR flags respectively.

The VHDL program realizing the tbf multiplexer is shown in <<2>>.

```
-- <<2>> VHDL Program "test bit mux"
process (a,b)
begin
-- convert #bit in integer
 sbf_sel<=conv_integer(b(2 downto 0));
end process;
   with sbf_sel select
   -- tbf multiplexor
      sbf <= a(0) when 0,       -- C
             a(1) when 1,       -- Z
             a(2) when 2,       -- V
             a(3) when 3,       -- N
             a(4) when 4,       -- CF0
             a(5) when 5,       -- CF1
             a(6) when 6,       -- CF2
             a(7) when 7;       --
SR[7]
```

6. CONCLUSIONS

A method for configuring the SMCore microcontroller with base and complex flags is offered, based on the analysis of the conditions for setting the status register flags.

An approach to reducing the number of instructions for organizing conditional branches is shown, based on the architectural features of the microcontroller.

A method for organizing all branches with two instructions based on the *Skip* mechanisms is chosen.

The components realizing the *Skip* mechanisms in VHDL are shown.

The results of the research conducted can be used in designing FPGA-based 8-bit RISC microcontrollers.

REFERENCES

[1] www. microchip.com/pic 16 family

[2] www. intel.com/msc51 family

[3] www. atmel.com/atmega family

[4] Kanev I., *FPGA Based Micro controller for Voice Message Synthesis*, Proc. of CompSysTech'05, Int. Conf. on Computer Systems and Technologies, 2005, pp 1.3.1-1.3.7

ABOUT THE AUTOR

Ivan Kanev, Department of Computer Systems, Technical University Sofia – Branch Plovdiv Phone +359 32 659 704, E-mail: ikanev@it-academy.bg.