FRI-2G.303-1-CST-02

AN EXPERIMENTAL SOFT-CORE STACK PROCESSOR

Princ. Assist. Nikolay Kostadinov, PhD
Department of Computer Systems and Technologies
"Angel Kanchev" Univesity of Ruse
Tel.: +359 82 888 674
E-mail: nkostadinov@ecs.uni-ruse.bg

Assoc. Prof. Milen Loukantchevsky, PhD, IEEE Member, ACM Member
Department of Computer Systems and Technologies
"Angel Kanchev" Univesity of Ruse
Tel.: +359 82 888 674
E-mail: mil@ieee.org

Princ. Assist. Hovanes Avakyan, PhD
Department of Computer Systems and Technologies
"Angel Kanchev" Univesity of Ruse
Tel.: +359 82 888 674
E-mail: havakian@ecs.uni-ruse.bg

Abstract: *Soft-core processors have become a reasonable alternative for embedded system design due to their flexibility and the possibility to integrate with custom logic to build system-on-chip applications. While a wide variety of commercial soft-core processor implementations exists, a minority of them are suitable for teaching purposes. This paper presents the design and implementation of a simple soft-core stack processor as a part of authors' effort to contribute to the learning process in processor design and computer architecture fields. The design and implementation steps, as well as future extensions, are discussed.*

Keywords: *Soft-core processor, Processor design, Computer architecture, VHDL model, FPGA*

INTRODUCTION

Soft-core processors are processors described in Hardware Description Language (HDL), which can be synthesized and mapped to the fabric of Field Programmable Gate Arrays (FPGAs). Examples include Nios II (Altera, 2016), Xilinx MicroBlaze (Xilinx, 2017), LEON SPARC (Cobham Gaisler, 2017), etc. These processors have become a reasonable alternative for embedded system design due to their flexibility and the possibility to integrate with custom logic to build System-on-Chip (SoC) applications.

While a wide variety of sophisticated commercial soft-core processor implementations exists, a minority of them are suitable for teaching purposes. Therefore, to support the learning process, a number of educational processors have been developed. A partial list includes the traditional in computer architecture education processor MIPS (Patterson, D. A. & Hennessy, J. L., 2013) with its HDL presentation (Harris, D. & Harris, S., 2012), 16-bit RISC processor Sweet-16 (Angelov, V. & Lindenstruth, V., 2009), 16-bit accumulator based processor BIP (Pereira et al. 2012), 8-bit RISC soft-core processor (Zavala et al. 2015), etc.

To facilitate the teaching in processor design and computer architecture topics, we designed simplified soft-core processors for the three main architecture types – accumulator, stack and register. This paper presents one of them – a minimalistic experimental stack processor (MESP) with Harvard architecture. In the next sections, some details of the design and implementation steps are outlined, and future extensions are discussed.

EXPOSITION

The tools used for modelling, simulation and implementation of the processor include Xilinx ISE and Spartan-3E Starter Kit board.

The MESP uses separate instruction and data spaces. Both instruction and data addresses are 8-bit wide. The stack contains four registers (StackA, StackB, StackC and StackD). All registers are 8-bit wide except the top of the stack (StackA), which is complemented by a carry flag (CF). For input/output, two 8-bit ports (InPort, OutPort) are provided.

Instruction set design

The simplified instruction set consists of just seven instructions:

- Data movement:
  - PUSH – load the content of memory location/immediate operand into StackA;
  - POP – store the content of StackA in a memory location;
  - INP – read data from the input port and stores them in StackA;
  - OUTP – write the content of StackA to the output port;
- Arithmetic and logic operations:
  - ADD – add the contents of StackA and StackB;
  - NOR – bitwise negated OR of the contents of StackA and StackB;
- Conditional branch:
  - JCC – branch if carry flag cleared.

Although the stack machine has zero-operand architecture, absolute and immediate addressing modes were implemented for the PUSH instruction, as well as absolute addressing for the POP.

The instruction length is fixed to 12-bit, also single instruction format is used (Fig. 1).

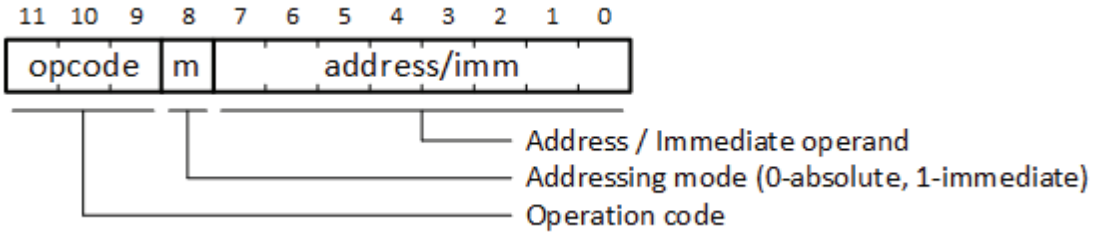

Fig. 1. MESP instruction format

While PUSH and INP append a new item onto the top of the stack and move the stack "down" (StackD ← StackC ← StackB ← StackA ← [new item]), POP and OUTP move the stack "up" (StackA ← StackB ← StackC ← StackD ← 0). Arithmetic and logic instructions operate on the top two elements of stack, leaving the result on StackA and moving the content of the stack "up" (StackB ← StackC ← StackD ← 0).

Table 1 summarizes the instructions with their corresponding syntax, encoding and function.

Table 1. MESP instruction set

Instruction	Syntax	Encoding	Function	Note
PUSH	PUSH address	0010[address]	StackA ← mem[address]	
	PUSH #imm	0011[imm]	StackA← imm	
POP	POP address	0100[address]	mem[address] ← StackA	
NOR	NOR	011000000000	StackA ← StackA nor StackB	
ADD	ADD	100000000000	StackA ← StackA + StackB	Affect CF

INP	INP	101000000000	StackA ← InPort	
OUTP	OUTP	110000000000	OutPort ← StackA	
JCC	JCC address	1110[address]	PC ← address if CF=0	Clear CF

The JCC instruction checks the CF to determine whether to branch. This instruction always clears the CF, so two consecutive JCC instructions form an unconditional jump. Similarly, new instructions can be synthesized from others (Table 2).

Table 2. Synthesized instructions

Instruction	Syntax	Description
CLR	PUSH #FFh, NOR	Clear StackA
NOT	PUSH #0, NOR	Invert the content of StackA
JMP	JCC dest, JCC dest	Unconditional jump to dest
JCS	JCC ($ + 2), JCC dest	Jump to dest if CF=1 ($ is the current PC)
SUB	PUSH #1, PUSH mem, PUSH #0, NOR, ADD, ADD	Subtract content of memory location mem from the content of StackA

Datapath

The datapath includes the black marked blocks shown in Fig 2.

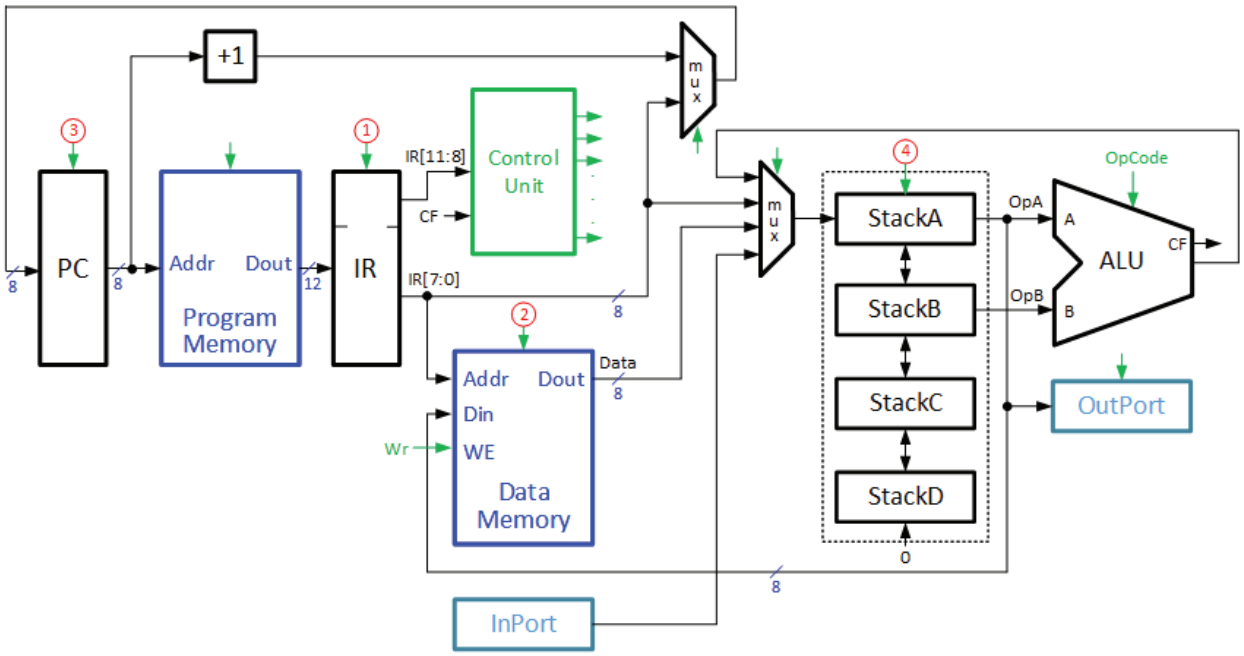

Fig. 2. MESP block diagram

The instruction fetch–execute cycle implies four steps, numbered in Fig. 2:

1) The instruction code retrieved from the program memory location, specified by the program counter (PC), is copied into the instruction register (IR). Then the instruction is decoded by the control unit using the four most significant bits of the IR (IR[11:8]). Depending on the addressing mode, the last eight bits (IR[7:0]) specify either an address of the data memory (for instructions PUSH or POP) or immediate value (only for PUSH).

2) Read/write from/to data memory. Read operation takes place only for PUSH instruction with absolute addressing - as a result, the output from the data memory (Data) is passed to the input of the StackA. With a POP instruction, the StackA content is stored to the data memory.

3) The PC is either incremented by 1 or loaded with the address from the JC instruction if CF=0.

4) The result of ADD/NOR operations is stored back into StackA. For the PUSH instruction with immediate addressing StackA is loaded with the 8-bit value from IR[7:0]. For input/output operations (IN and OUT instructions) data is exchanged between StackA and the ports. At this step, depending on the instruction type, the content of the stack is moved "up" or "down".

The program and data memory are implemented using the on-chip block RAM modules available in the Spartan-3E FPGA devices. The block RAM modules are synchronous with read/write operations performed at the rising clock edge. Taking into consideration both the datapath diagram and the fetch–execute steps, it is concluded that the instruction cycle can be accomplished within three edges of the clock signal.

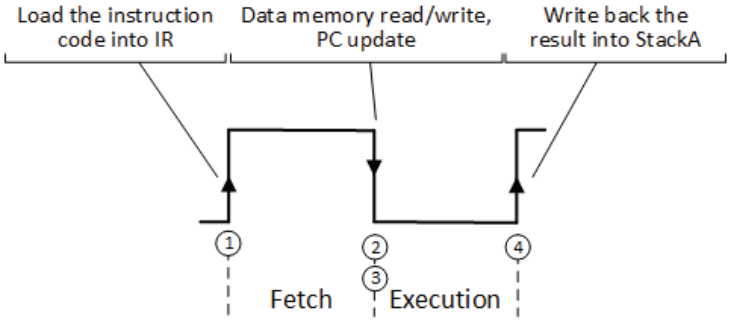

Fig. 3. Instruction cycle timing

To achieve this timing, first, the output register of the program memory plays a role of IR, and second, the clock signal of data memory is inverted. Thus, each instruction is executed in one clock cycle (Fig. 3), having the steps 2 and 3 of the current instruction as well as step 4 of the current and step 1 of the next instruction overlapped.

Control unit

The control unit was designed according to the states and operations defined in Table 3. Here, input signals of the state machine include *opcode* and addressing mode fields of the instruction IR[11:8], as well as flag CF.

Table 3. Control unit states and operations

State (code)	Register-transfer operation	Next state
FETCH_STATE (000)	IR ← mem[PC] Data ← mem[IR[7:0]]	Determined by IR[11:9]
PUSH_STATE (001)	StackD ← StackC ← StackB ← StackA StackA← Data if IR[8] = 0 StackA← IR[7:0] if IR[8] = 1 PC ← PC + 1	FETCH_STATE
POP_STATE (010)	mem[IR[7:0]] ← StackA StackA ← StackB ← StackC ← StackD PC ← PC + 1	FETCH_STATE
NOR_STATE (011)	StackA← StackAnorStackB StackB ← StackC ← StackD PC ← PC + 1	FETCH_STATE

ADD_STATE (100)	StackA ← StackA + StackB StackB ← StackC ← StackD PC ← PC + 1	FETCH_STATE
INP_STATE (101)	StackD ← StackC ← StackB ← StackA StackA← InPort; PC ← PC + 1	FETCH_STATE
OUTP_STATE (110)	OutPort ← StackA StackA ← StackB ← StackC ← StackD PC ← PC + 1	FETCH_STATE
JCC_STATE (111)	PC ← IR[7:0] if CF = 0, PC ← PC + 1 if CF = 1 CF ← 0	FETCH_STATE

To simplify the implementation, the state codes specified correspond to the value of *opcode* field. Thus, while the first state of the instruction cycle is FETCH_STATE, the next state is determined directly by bits 11:9 of the IR. Similarly, at the execution phase the current state code directly controls the arithmetic logic unit (ALU).

VHDL modelling and simulation

The datapath and control unit of MESP were described through behavioral VHDL approach. Then program and data memory were instantiated via parameterized FPGA block RAM modules. The final processor was composed using an enclosing VHDL entity that connects datapath, control unit and memories together at the structural level. Fig. 4 shows a fragment of VHDL process, describing the execution phase of the instruction.

```
if (Clk'event and Clk = '1') then
  case State is          -- Execution
    when PUSH_STATE =>
      if (IR(8) = '1') then
        StackA(7 downto 0) <= IR(7 downto 0); -- Immediate addressing
      else
        StackA(7 downto 0) <= Data;           -- Absolute addressing
      end if;
    when POP_STATE  => StackA(7 downto 0) <= StackB;
    when NOR_STATE  => StackA(7 downto 0) <= StackA(7 downto 0) nor StackB;
    when ADD_STATE  => StackA <= ('0' & StackA(7 downto 0)) + ('0' & StackB);
    when INP_STATE  => StackA(7 downto 0) <= InPort;
    when OUTP_STATE => OutPort <= StackA(7 downto 0); StackA(7 downto 0) <= StackB;
    when JCC_STATE  => StackA(8) <= '0';
    when others     => null; -- Fetch phase
  end case;
  case State is          -- Move the stack "up" and "down"
    when NOR_STATE | ADD_STATE | OUTP_STATE | POP_STATE =>
      StackB <= StackC; StackC <= StackD; StackD <= (others => '0');
    when INP_STATE | PUSH_STATE =>
      StackD <= StackC; StackC <= StackB; StackB <= StackA(7 downto 0);
    when others => null; -- Fetch phase
  end case;
end if;
```

Fig. 4. A fragment of VHDL code

To test the correctness of all instructions, various software routines were created. For, example, Fig. 5 shows a simulation trace of a program that calculates the greatest common divisor of two input numbers.

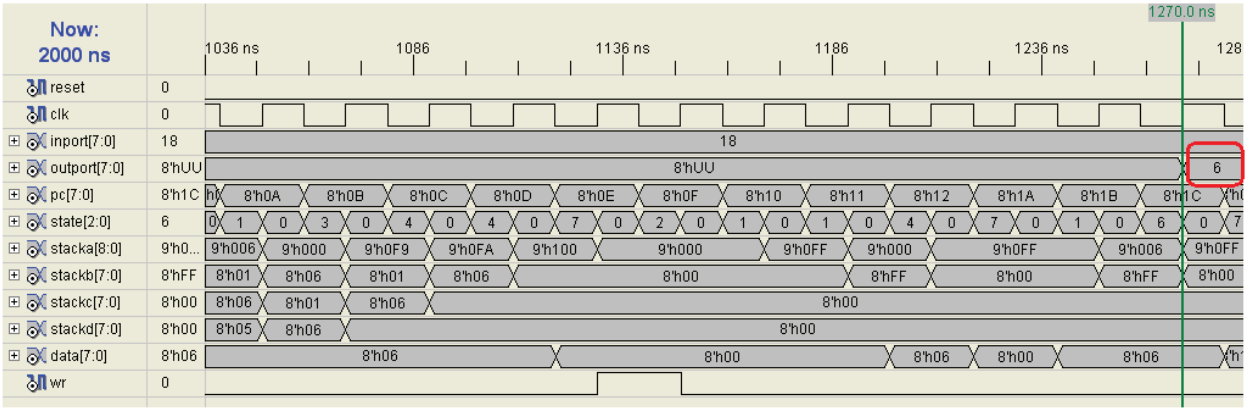

Fig. 5. Simulation trace of calculating the greatest common divisor of A and B (A=24, B=18)

The completed stack processor was implemented and tested on Spartan-3E Starter Kit. With provided clock from the on-board 50 MHz clock oscillator the instruction execution rate is 50 MIPS. The utilization of the target XC3S500E device reported by Xilinx ISE is less than 1%.

Assembler

The program memory of the processor must be initialized with the binary code of instructions prior to the synthesis phase.

To support the programming process, a simple assembler was developed. Fig. 6 shows the grammar of the MESP assembly language.

<StackAssemblyProg> → *<Line>* *<StackAssemblyProg>* | *<null>*
<Line> → *<Directive>* **EOL** | *<LabelledInstruction>* **EOL**
<Directive> → **IDENT EQU CONST**
<LabelledInstruction> → **LABEL :** *<LabelledInstruction>* | *<Instruction>*
<Instruction> → **PUSH** *<Oper>* | **POP** *<Expr>* | **NOR** | **ADD** | **INP** | **OUTP** | **JCC LABEL**
<Oper> → *<Expr>* | *<ImmExpr>*
<ImmExpr> → # *<Expr>*
<Expr> → **IDENT** | **CONST**

Fig. 6. MESP assembly language grammar

The output of the assembler is a VHDL description of the program memory with the binary code of instructions integrated in its initialization sections.

CONCLUSION

This paper presents the design and implementation of simple soft-core processor as a part of authors' effort to contribute to the learning process in processor design and computer architecture fields.

The presented processor is as simple as possible to give the students an insight into the design steps, as well as data flow within the instruction execution cycle. Despite the simplicity and limitations, it can be considered as a basis for various experiments and modifications: extension of the instruction set, the address spaces and the register width; support of interrupts and subroutines; implementation of multiprocessor systems within single FPGA, etc.

We expect the presented processor to be useful for students in their comprehension of the processor design and functioning.

REFERENCES

Altera (2016). *Nios II Classic Processor Reference Guide*. Available at:

https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf (Accessed: 18 Sep 2017).

Angelov, V. & Lindenstruth, V. (2009). The educational processor Sweet-16. *International Conference on Field Programmable Logic and Applications*. Prague, Czech Republic, 31 Aug.-2 Sept. 2009. IEEE, pp. 555–559.

Cobham Gaisler (2017). *LEON4 Processor*. Available at:
http://www.gaisler.com/index.php/products/processors/leon4 (Accessed: 18 Sep 2017).

Harris, D. & Harris, S. (2012). *Digital Design and Computer Architecture*. 2nd ed. Morgan Kaufmann.

Patterson, D. A. & Hennessy, J. L. (2013). *Computer Organization and Design: The Hardware/Software Interface***.** 5th ed. Morgan Kaufmann Publishers.

Pereira, M.C., Viera, P.V., Raabe, A.L., & Zeferino, C.A. (2012). A basic processor for teaching digital circuits and systems design with FPGA. *VIII Southern Conference on Programmable Logic*. 20-23 March 2012, Bento Goncalves, Spain. IEEE, pp. 1-6.

Xilinx (2017). *MicroBlaze Processor Reference Guide*. Available at:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug984-vivado-microblaze-ref.pdf (Accessed: 16 Sep 2017).

Zavala, H.A., Nieto, C.O., Ruelas, H.J.A., & Dominguez, C.A.R. (2015). Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning. *Computación y Sistemas*, 19(2), pp. 371-385.

