International Conference on Smart Electronic Systems - 2016
KLE Society's Dr M S Sheshgiri College of Engineering & Technology, Belagavi

International Journal of Technology and Science, ISSN (Online) 2350-1111, (Print) 2350-1103 Volume 3, Issue 3, 2016 pp. 36-39

Design of a Modular Processor Framework

Shilpa Mayannavar[1], Uday V. Wali[2]
[1] VTU Research Resource Centre/EC, Belagavi, India
[2] KLE Dr. MSS CET/ECE, Belagavi, India
Email: [1]mayannavar.shilpa@gmail.com, [2]udaywali@gmail.com

Abstract — **The paper discusses a framework for design of special purpose processors using Harvard Architecture. The framework supports various modules like the priority encoded bus, ALU, instruction memory, data memory, IO ports, etc. It allows users to design and test various instruction sets and compare their performance. Both RISC and CISC type of instructions can be implemented. Various addressing modes can also be incorporated. The framework essentially supports instruction level simulation of the processor being designed. The design is primarily intended for students to understand the basics of processor design with emphasis on application specific integrated circuits (ASIC). Modularity of the design allows upgrades to individual modules without affecting the design of other modules. Support for advanced modules like instruction pipelines, semaphores, mutex is available within the framework.**

Index Terms— **Instruction Set Architecture (ISA), Processor design, Processor framework.**

I. Introduction

Few steps ahead of the Microcontroller is the Custom System on Chip (cSoC). The idea behind a cSoC is derived from the fact that most of the processors can be built with a set of reusable building blocks and a customized controller, whose design depends on the Instruction Set Architecture (ISA) and the Bus design. Internet of Things (IoT) will expand the need to build customized processors in years to come. Instruction set architecture for IoT devices is largely driven by the end application and the bus design depends on how fast we need to move data between individual blocks. IoT market is sensitive to time-to-market as well as cost. Therefore it is necessary to build a capability to assemble a cSoC in the shortest possible time using the least area on chip. This is a technological challenge and presents a trade-off between design time and chip area. One way to achieve optimal chip design is to make use of pre-defined, optimized set of processor building blocks, made available in a processor design framework. Such a framework should provide common interfaces between individual modules, reducing time delays on various data paths. The framework should also provide processor blocks individually optimized for delays and interconnectivity between blocks in the same processor framework.

Some such processor frameworks are already in use. For example, processors built using ARM core, MIPS, Arduino, etc. Many chips have been designed using 8051 architecture. Microchip has developed many PIC processors around its core design and instruction set. Development of a processor in traditional sense involved development of a complete eco system, which holds in IoT era as well. However, availability of software tools makes it easy to develop a processor that fits existing an ecosystem and meets application specific requirements.

We also wanted to develop a platform for simulating experimental processor designs and instruction set architectures, useful for students of digital electronics. Availability of processor design library should enable students to learn internals of a processor design.

In this paper, we present design of a processor framework that meets the requirements presented above. The framework is being used to test new types of instruction sets and bus arbitration schemes. An illustrative processor design is also included in the paper.

II. Design Methodology

The basic building blocks of a Harvard Processor Architecture include Clock generator, Program Counter (PC), Instruction Memory, Data Memory, Controller (Instruction fetch, decode and execute), Arithmetic and Logic Unit (ALU), bus arbitrator, etc. We have developed a library of hardware modules in Verilog that can be used to build application specific processors and experiment with various instruction sets. We intend to use this library to build cSoCs that for use in IoT research. The modules are kept simple so that we can concentrate on the experiments with design rather than implementation. As an illustration, we have shown how the library can be used to implement an instruction set using load-store architecture.

We have presented some of the essential modules here.

A. Clock Generator

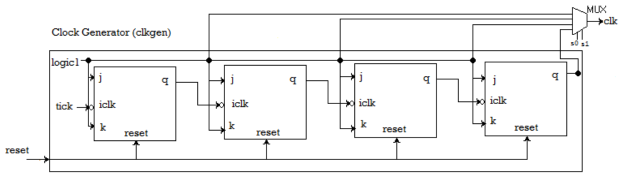


Figure 1: Block diagram of Clock Generator

Clock Generator module can accept basic clock from an external port or can generate the signal internally. A clock divider is used to improve the wave shape and provide some frequency scaling. A Digital PLL based internal clock is currently under development. The clock should be reset on power-up to ensure proper initialization of the module. The clock can also be reset at other times, such as system reboot. It uses four JK flip-flops connected in a cascade. It is important to note that JK flip-flops are negative edge triggered.

Output of the clock generator can be derived from any one of the flip-flops, giving various speeds and clocks. We will denote it as *clk* for convenience.

B. *Program Counter*

Program Counter (PC) is n-bit register which holds the address of next instruction to be executed. Program counter does three operations, viz., *increment, set* and *reset*. Corresponding inputs to the PC are *incrPC, setFlag and reset* flags, in increasing order of priority.

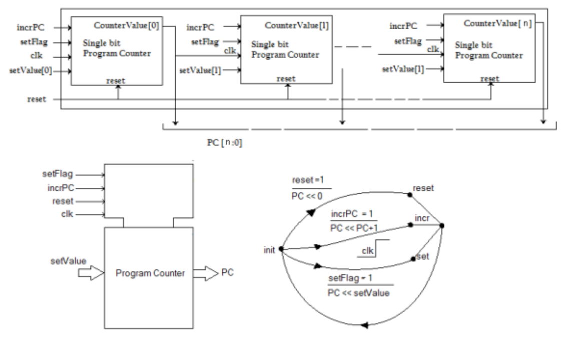


Figure 2: Block diagram, Interface diagram & FSM for Program Counter

These signals are mutually exclusive, so in case there is multiple request, action corresponding to the higher priority is initiated. For example, reset will over-ride other flags. However, the cSoC controller should ensure that only one of flags is active at any rising edge of *clk*.

High value on any of the *flag* pins should be available before 1 *sclk* (setup time) and 1 *sclk* after clk (hold time). Controller should set *incrPC* to indicate that the PC can now be incremented. In case of branching instructions, PC has to be set to *setValue* often decided by *address* field of the branch instruction being executed. Controller has to set *setFlag* high before rising edge of clk as well as present the branch address on *setValue*.

. On *reset*, the PC will be set to 0. Input flags to PC should be cleared after reset so that the instruction at location 0 can be fetched. On rising edge of *clk*, if *incrPC* is set, PC will increment, before falling edge of *clk*. On rising edge of *clk*, if *setFlag* is set, *setValue* is transferred to PC before falling edge of *clk*. Output from PC is generally connected to address of instruction memory. It is also connected to stack for stack operations.

C. *Instruction Memory*

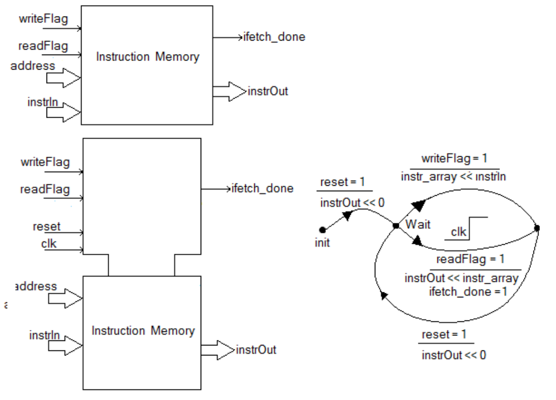


Figure 3: Block diagram and FSM of Instruction Memory

Instruction Memory stores the instructions in a memory called *instr_array*. On rising edge of *clk*, if *WriteEn* flag is high then *instr_In* is written into the *instr_array* and if *ReadEn* flag is high then content of *instr_array* at the address specified in *address* is read to *instr_Out* register. In this paper Instruction memory is fixed and hence we do not have to write the instruction to the memory, instead we are just reading the available instructions from the instruction memory.

Once the instruction is available in the *instr_Out* register, *ifetch_done* flag is set.

D. *Control unit*

Control unit (or controller) generates required gating signals depending on the instruction to be executed. It also connects immediate data to ALU, data memory, program counter, etc depending on the instruction.

Control unit has two modules in it; instruction decode and execute. The *iReady* flag of control unit indicates availability of instruction for decoding.

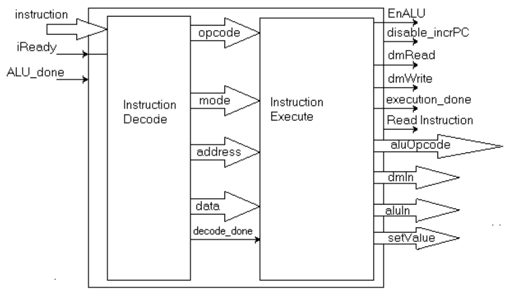


Figure 4: Block diagram of Control unit

The *ifetch_done* flag of instruction memory is connected to *iReady* to indicate that the instruction is ready to be decoded. Instruction is decoded only if *execution_done* flag is set. On reset, this is set so that decoding of the first instruction starts. This event should also clear *execution_done* flag so that decode operation does not restart till current instruction is executed. Once the instruction fetch is completed, PC should be allowed to increment. This is done by clearing disable_incrPC.

IncrPC flag of Program Counter is set as

~reset & ~disable_incrPC & iReady & ALU_done

On the next rising edge of *clk,* if *decode_done* is high then *disable_incrPC* should be set to 1, which stops further increment of PC register till execution is complete (or otherwise if pipelined).The control unit will set the disable_*incrPC* and *execution_done* flag on *reset.* On the rising edge of *clk*, instruction memory unit fetches the instruction at location pointed to by PC and stores it in *instr_out* buffer. It also sets *iReady* flag. As the iReady flag is set, on the negative edge of clk, instruction decode module will split the instruction in the *instr_out* buffer into *opcode*, *mode*, *data* and *address* fields. Simultaneously, *decode_start* flag is set and disable_*incrPC* is cleared. This enables the program counter to increment the PC register.

The flag *decode_done* is connected to *execution_start* flag of the execute module. Execution starts after decoding of instruction is completed and only after execution is completed, next fetch cycle should start. On the rising edge of *clk*, if the *execution_start* is set, then depending on the *opcode* and *mode,* control unit will send the corresponding flags such as dmRead, dmWrite, setFlag, etc to different modules. Some of these are described below.

The flag *EnALU* is sent to ALU module to indicate the start of the intended operation. If the instruction is Load immediate data then content of *data* field is transferred to *aluIn* field. If the instruction is Store immediate data, then *dmWrite* flag is set and also immediate data is transferred to *dmIn.* Similarly if the instruction is Add immediate data with the content of data memory then *dmRead* flag is set and *dmOut* is transferred to *aluIn* through the data bus. The *aluOpcode* field represents the operational code of ALU.

On the rising edge of *clk,* if *ALU_done* flag is high then *ReadInstruction* flag is set to 1 so that the next instruction can be fetched and the cycle repeats.

E. Data Memory

Data memory stores the data in the memory called *mem.* On the rising edge of *clk,* if *readFlag* is set then data from *mem* is read to the *dmOut* register and similarly if the *writeFlag* is set then incoming data is written to *mem* at the address specified in the *address* field..

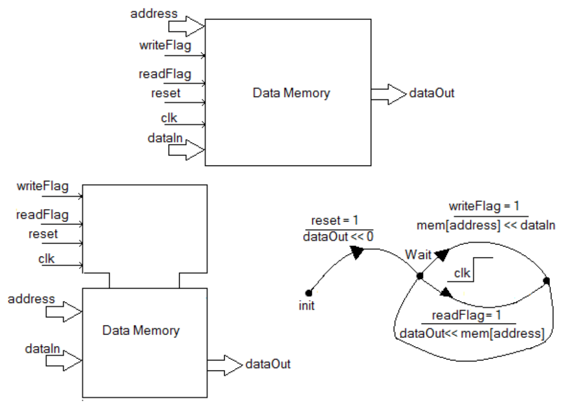


Figure 5: Block diagram and FSM of Data Memory

F. Arithmetic and Logic Unit (ALU)

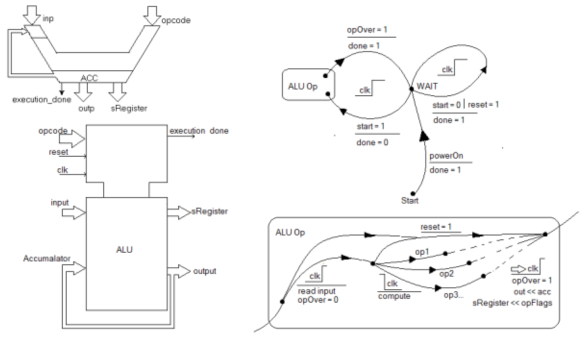


Figure 6: Block diagram and FSM of ALU

ALU performs arithmetic and logical operations on two inputs depending on the *opcode.* The result is stored in the accumulator (ACC). Once the intended operation is done, ALU sets the *execution_done* flag. The *sRegister* field is a 16bit register, out of which only 4 bits are currently defined to hold the status of carry flag, zero flag, negative flag and overflow flag, respectively from LSB to MSB.

G. Input Bus port

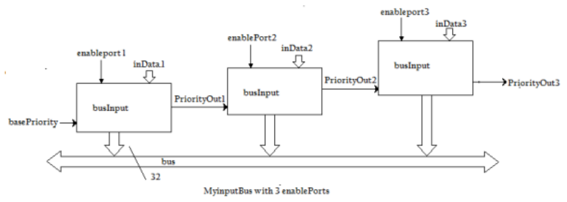


Figure 7a: Block diagram of Bus Input port

The figure 7a shows Input bus port with 3 enable ports. If any of the port wants to write the data to bus then corresponding *enableport* must be set. In a processor many ports may want to write the data to bus at a time, in such cases priority chain selects the port in increasing order of priority. The block diagram of priority chain is shown in figure 7b.

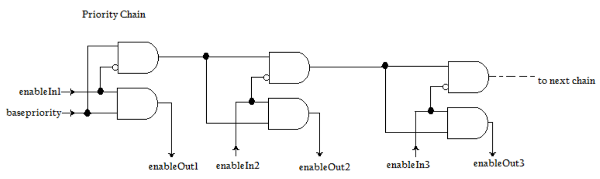


Figure 7b: Block diagram of priority chain

H. Output Bus port

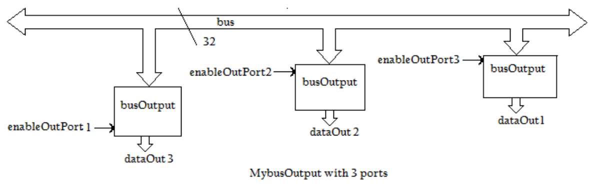


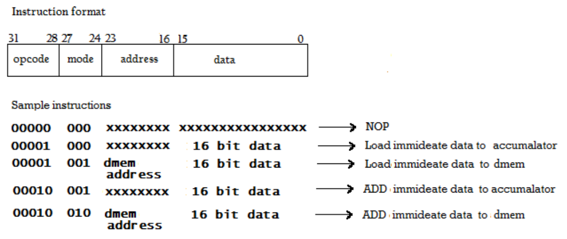
Figure 8: Block diagram of Bus Output port

Once the data is available on the data bus, any of the output ports can receive the output from bus at a time. In that case *enablePort* signal is used to enable the particular port. It is important to note that there is no need of priority chain in the bus output port, as one or more ports can receive the data available from the bus at a time.

III. Sample Instruction Format

Instruction is divided into different fields, namely; *opcode, mode, data* and *address* as shown in figure 9. *Opcode* represents the instruction class, *mode* represents addressing mode *and address* represents address of the memory and *data* field represents 16bit immediate data.

Sample instructions are shown in the figure 9. If the *opcode* and *mode* are 0 then it's No operation (NOP). If the *opcode* is 00001, it means load operation and immediate data is loaded into either accumulator or data memory depending on *mode.* In the load instruction, if the *mode* is 000 then data is loaded into accumulator and if the *mode* is 001 then the data is stored in data memory and so on.


Instruction format

31 28	27 24	23 16	15 0
opcode	mode	address	data

Sample instructions

```
00000  000  xxxxxxxx      xxxxxxxxxxxxxxxx  ⟶ NOP
00001  000  xxxxxxxx      16 bit data       ⟶ Load immideate data to accumalator
00001  001  dmem address  16 bit data       ⟶ Load immideate data to dmem
00010  001  xxxxxxxx      16 bit data       ⟶ ADD immideate data to accumalator
00010  010  dmem address  16 bit data       ⟶ ADD immideate data to dmem
```

Figure 9: Instruction format

IV. Sample Instruction for Adding two numbers

Sample instruction for adding two immediate numbers is explained using the above discussed modules.The *clk* is generated from the Clock Generator module, the same *clk* is used for all the modules. Initially the PC is pointing at 0 location of instruction memory. On the rising edge of *clk*, if the *iReady* flag is set then control unit will enable the instruction decode to decode the instruction. Once the decoding is done, control unit will enable the instruction execute module on the rising edge of *clk*, to execute the instruction. Depending on the *opcode* and *mode*, control unit will enable the ALU module and then ALU will perform the intended operations and store the result in outp register.

The sample instruction is tested and simulated as shown in figure 15.

V. Simulation Results

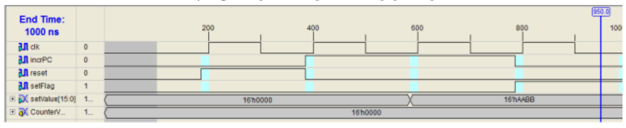


Figure 10a: Test Bench of Program Counter

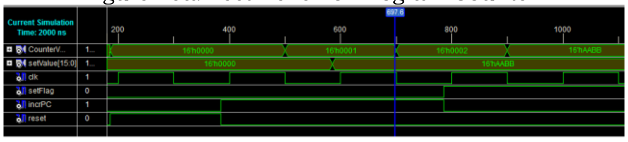


Figure 10b: Simulation result of Program Counter

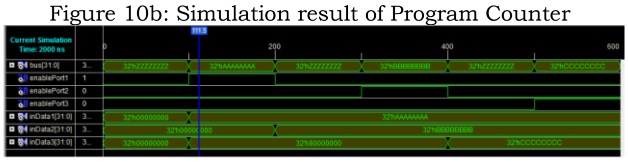


Figure 11: Simulation result of My Bus Input port

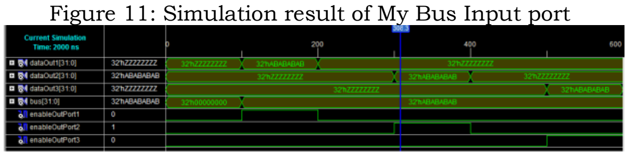


Figure 12: Simulation result of My Bus Output port

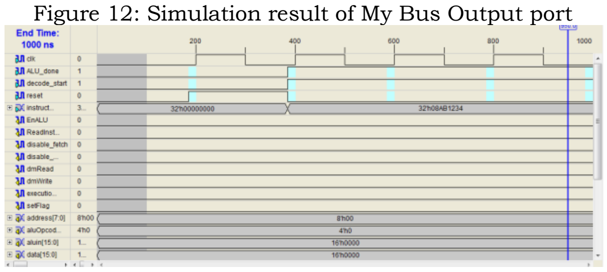


Figure 13a: Test Bench of Control unit

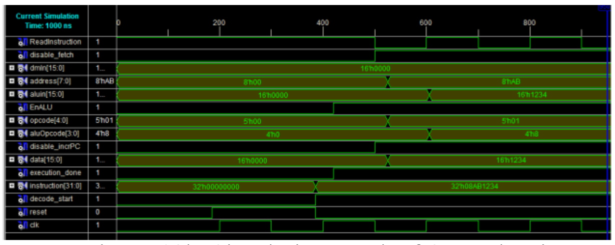


Figure 13b: Simulation result of Control unit

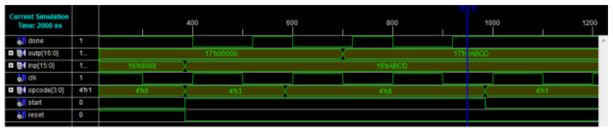


Figure 14: Simulation result of ALU

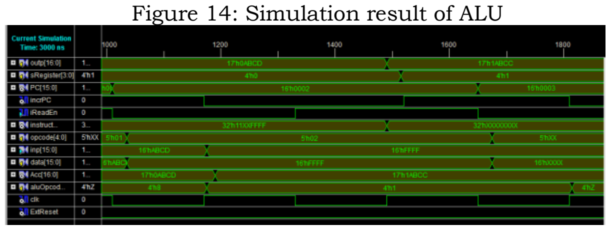


Figure 15: Simulation result of Sample instruction

Conclusions

We have implemented a small set of Verilog modules necessary to simulate instruction sets. Test benches for each of the modules has been written and tested. Modules have been integrated to support an illustrative instruction set. The code is stable enough to synthesize on FPGA. We have not currently described any timing constraints. We have assumed Harvard architecture supporting both RISC and CISC instruction sets. The work is still in progress and will remain so for some time. Modules like IO devices and their memory maps, index registers, hierarchical buses, etc need to be implemented and tested. We intend to use the library to test new high level instruction sets that may find use in IoT devices and other processors.

Acknowledgment

The authors would sincerely like to thank CQuad Computers for all the help and support.

References

[1] Sivarama P Dandamudi , "Guide to RISC Processors for Programmers and Engineers", Springer.

[2] Shashank Kaithwas, "Design of 16-bit Data Processor Using Finite State Machine in Verilog", IJERGS, Volume 2, Issue 5, August – September 2014

[3] V.Prasanth , "FPGA Based 64-Bit Low Power RISC Processor Using Verilog HDL", International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, (An ISO 3297: 2007 Certified Organization) Vol. 3, Issue 11, November 2014

[4] Anjana R & Krunal Gandhi, "VHDL Implementation of a MIPS RISC Processor", August 2012, International Journal of Advanced Research in Computer Science and Software Engineering, pp 83-88

[5] Jianming Liu "Design of Simple CPU Based on Hardware Description Language", Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)