Realization Of An 8-bit Pipelined Microprocessor in Verilog HDL

Jayant Chowdhary[*](Corresponding Author)   Vivek Garg   Tushar Negi   Shreya Jain

Delhi Technological University,Shahbad Daulatpur,Main Bawana Road,Delhi-42

*Email of the corresponding author: jayantc11@gmail.com

Abstract

Pipelining is a technique of decomposing a sequential process into sub-operations, with each sub process being divided segment that operates concurrently with all other segments. A pipeline may be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing segments dictated by the way the task is partitioned. The result obtained in one segment is transferred to subsequent segments in each step. The final result is obtained after the data has passed through all segments.This paper develops a code for the implementation of an 8-Bit microprocessor which implements instruction pipelining. After synthesis, an FPGA realization may be obtained . Simulation using Xilinx and ModelSim also produces favourable results which showcase the speedup (in terms of time) to carry out a program as compared to a non-pipelined version of this microprocessor.

Keywords: Pipelining, Segments,sysnthesis,realization,FPGA,microprocessor

1.Introduction

Instruction pipelining

An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction 'fetch' and 'execute' phases to overlap and perform simultaneous operations.

An instruction can generally be decomposed into the following steps:

1. **FI- Fetch instruction**: In this segment an instruction is fetched from memory.
2. **DA- Decode the instruction and calculate the effective address**: This instruction gets its input from the FI segment and the instruction is decoded. This step does not require a clock cycle. The operation which requires a clock cycle is the calculation of effective address.
3. **FO- Fetch operand**: In this step the operand is fetched from the memory.
4. **EX- Execute and store**: In this step the instruction is executed and the result is also stored in an appropriate memory location.

A Space time diagram for an instruction pipeline can be made as follows:

Instruction	Step1	Step2	Step3	Step4	Step5	Step6	Step7
1	FI	DA	FO	EX			
2		FI	DA	FO	EX		
3			FI	DA	FO	EX	
4				FI	DA	FO	EX

As we can see each segment is simultaneously busy processing some part of an instruction.

2. Processor Realization

CPU ORGANIZATION AND LAYOUT :

1) 8 bit cpu
2) Architecture: VON NEUMAN
3) Behavioural modelling only
4) Different Program Memory and Data Memory
5) 16 words of Program memory
6) 4096 words of data memory
   Register Organization:

1)Program Counter: PC – 6 bits

2)Instruction register: IR - 16 bits

3)Address register: AR - 12 bits

4)Temporary address storage: ad - 6 bits (6 LSBs of AR)

5)Memory check register: memocheck -1 bit (for register reference and memory reference istructions)

6)Current state register: current_state – 2 bits (to check the present state- fetch/decode/execute/store)

7)Next state register: next_state- 1 bit (to hold the value of next operation to be carried out- fetch/decode/execute/return)

8)OPcode register: opcode – 3 bits

9)Instruct register: instruct -4 bits (in case instruction type is arithmetic/logical then this specifies type of arithmetic/logical op to be carried out)

10)Registers RA,RB,rstore - 4 bits (to hold the address of source and destination registers respectively)

11)Temporary register: W- 8 bits (to hold the value of the end result of any arithmetic/ logical operation and incase of data MOV operation it holds the immediate value of the data to be transferred)

3.Instruction Format:

1 bit	3 bits	4 bits	4 bits	4 bits

I   Opcode   Instruct   RA(index)   RB(index)

I : 1- Memory reference

0-Register reference

Opcode: operational code which tells us what type of operation is to be carried out on the data present at source registers ,given by the index in RA RB. Opcode list:

000 - HLT: The cpu goes into an infinite loop terminating the program

001 - MVI: Move immediate value to destination register register(indicated by rstore), {RA,RB} is the 8-bit concatenated word used as immediate data, rstore: address of destination register

000 (I=1) - STA: store contents of R0 (virtual accumulator) in the memory address given by ad;

001 (I=1) - LDA: load the contents of memory specified by ad into the accumulator (R0)

010 (I=1) - JMP: jump to the location specified by 6 bits in ad

010 – operation: opcode for all arithmetic /logical instructions further classified by 4 bits instruct field

MOV=0000; Move Contents Of Register[RB] To Register[RA]

ADD=0001; Add Contents Of Register[RB] with Contents Of Register[RA] & save result in R[A]

ADC=0010; ADD With Carry, Add contents of register[RB] with contents Of Register[RA] & save result In R[A]

SBB=0011; Subtract With Borrow

SUB=0100; Subtract contents of Register[RB] with contents of Register[RA] & save result in R[A]

INC=0110;Increment contents of Register[RA] & save result in R[A]

DEC=0111; Decrement contents of Register[RA] & save result in R[A]

AND=1001;Logically AND contents of Register[RB] with contents of Register[RA] & save result in R[A]

OR=1010; Logically OR contents of Register[RB] with contents of Register[RA] & save result in R[A]

XOR=1011; Logically XOR contents of Register[RB] with contents of Register[RA] & save result in R[A]

CMP=1000; Complement contents of Register[RA]

SHR=1100; Shift right contents of Register[RA] by 1 bit

SHL=1101; Shift left contents of Register[RA] by 1 bit

4.Program Code

```
module micropipeline(clk,rst);                    reg [0:7] mem [0:255];
input clk;                                        reg memocheck;
input rst;                                        reg [0:1] current_state;   //stages
reg [0:3]PC;                                       reg [0:1] next_state;       //for keeping track
reg [0:15] imem [0:15];        //instruction memory     reg [0:11] AR;              //Address register
reg [0:15] IR;                 // instruction register  reg [0:2] opcode;
reg [0:7] datareg;             //data register          reg [0:3] instruct;
reg [0:15] IRW;                                   reg [0:3] RA,RB,rstore;
```

```
reg [0:7] regfile [0:15];      //16 register(R0-R7)
reg [0:7] W;                              //temporary register
reg [0:2] count;
reg fetch,decode,execute,store;
reg [0:100]TR;
reg moveimm,movein,operation,sta,halt,halt1;
reg loadmemory,memorystore;
reg load;
wire carry;
wire [0:7] temp;
reg carry1;
reg [0:7] tink;
reg car;
initial
begin
PC=0;
TR=0;
fetch=1;
decode=0;
execute=0;
store=0;
moveimm=0;
movein=0;
operation=0;
halt=0;
sta=0;
loadmemory=1;
memorystore=1;
load=0;
next_state=4'b1000;
halt1=0;
IRW=0;
datareg[0]=0;
        imem[0]={16'b0001000111001111};
imem[1]={16'b00010010011011000};
imem[2]={16'b0011000000010010};
imem[4]={16'b0000000000000000};
imem[3]={16'b0000000000000000};
imem[5]={16'b0000000000000000};
imem[6]={16'b0000000000000000};
imem[7]={16'b0000000000000000};
end
always@(clk or rst)
begin
TR<=TR+1;
if(rst)
begin
PC<=0;
TR<=0;
fetch<=1'b1;
decode<=1'b0;
```

```
execute<=1'b0;
store<=1'b0;
sta<=1'b0;
halt<=1'b0;
end
else
    begin
if(store)
        begin
if(loadmemory&memorystore)
                begin
regfile[rstore]<=W;
end
if(halt1)
                        begin
fetch<=#1 1'b0;
                        decode<=#1 1'b0;
                        execute<=#1 1'b0;
store<=#1 1'b0;
end
end
if(execute)
        begin
if(halt)
                begin
halt1<=1'b1;
end

if(moveimm)
                begin
W<=IRW[8:15];
                        rstore<=IRW[4:7];
                        moveimm<=1'b0;
end
if(movein)
begin
W<=datareg[IRW[12:15]];
rstore<=IRW[8:11];
movein<=1'b0;
end
if(operation)
begin
case(IRW[4:7])
4'b0000: begin
W<=regfile[RA]+regfile[RB];
end

4'b0001: begin

W<=regfile[RA]+regfile[RB]+1;
end
```

```
4'b0010: begin
W<=regfile[RA]-regfile[RB]-1;
end

4'b0011:begin
W<=regfile[RA]-regfile[RB];
end

4'b0100: begin
W<=regfile[RA]+1;
end

4'b0101: begin
W<=regfile[RA]-1;
end
4'b0110: begin
W<=regfile[RB];
end

4'b1000: begin


W<=regfile[RA]&regfile[RB];
end

4'b1001: begin
W<=regfile[RA]|regfile[RB];end
4'b1010: begin
W<=regfile[RA]^regfile[RB];
end

4'b1011: begin
W<=regfile[RA];        // has to include ~
End
                        endcase
// alu operation
rstore<=4'b0000;
if(count==2'b00)
operation<=1'b0;
end
if(sta)
begin
memorystore<=1'b0;
mem[IRW[8:15]]<=regfile[4'b0000];
sta<=1'b0;
end
if(load)
begin
loadmemory<=1'b0;
regfile[4'b0000]<=mem[IRW[8:15]];
```

```
load<=1'b0;
end
store<=1'b1;
end
if(decode)
begin
        IRW<=IR;
case(IR[0])
0:begin
case(IR[1:3])
3'b000: begin
halt<=1'b1;
end
3'b001:
begin
moveimm<=1'b1;
end            //the instruction is mvi
3'b010: begin
movein<=1'b1;
end
3'b011:begin
operation<=1;
RA<=IR[8:11];
RB<=IR[12:15];
end
3'b100:begin
sta<=1'b1;
end
default:begin
end
endcase
end
1:begin
load<=1'b1;
end
endcase
execute<=1'b1;
end
if(fetch)
begin
IR<=imem[PC];
PC<=PC+1;
decode<=1'b1;
end
end
if((IRW[4:7]==IR[8:11]|IRW[4:7]==IR[12:15])&IR[1:
3]==3'b011)
begin
count<=2'b01;
fetch<=#2 1'b0;
decode<=#2 1'b0;
```

```
end                                      begin
if(count==2'b01)                         count<=2'b00;
begin                                    fetch<=1'b1;
count<=2'b10;                            decode<=1'b1;
execute<=1'b0;                           end
end                                      end
if(count==2'b10)                         endmodule
```

5.Results

The following was the result of synthesis of the code on Xilinx and on comparing the timing diagram with that of a non-pipelined microprocessor running the same program we find that there is a considerable speedup of 125ps.The synthesis and timing diagrams of pipelined and non pipelined microprocessors are shown below:
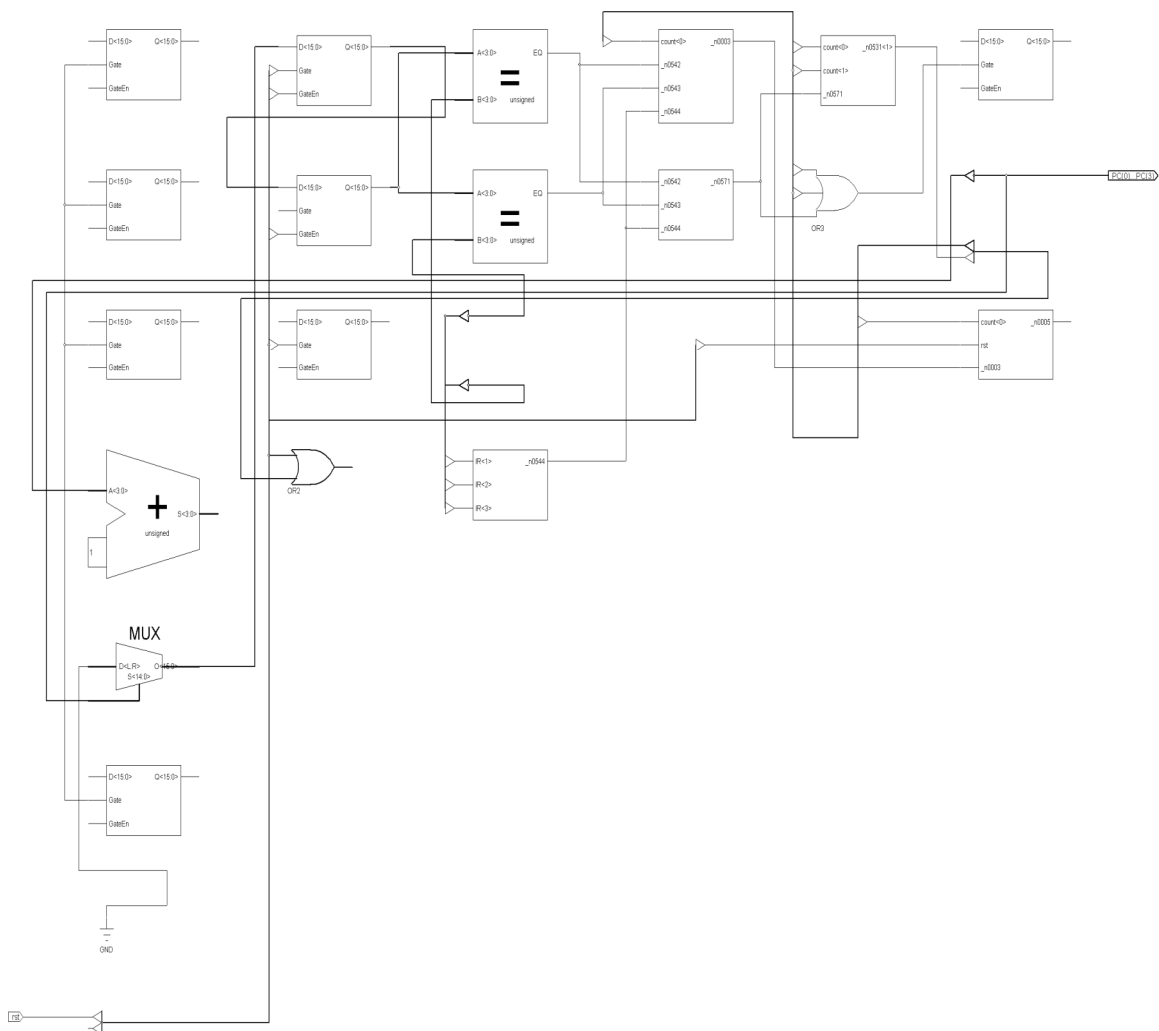
Given above is the timing diagram of a non pipelined microprocessor built on specs to the pipelined one. A 4 instruction program takes 201 ps as shown to get executed. Given below is the timing diagram of the same program
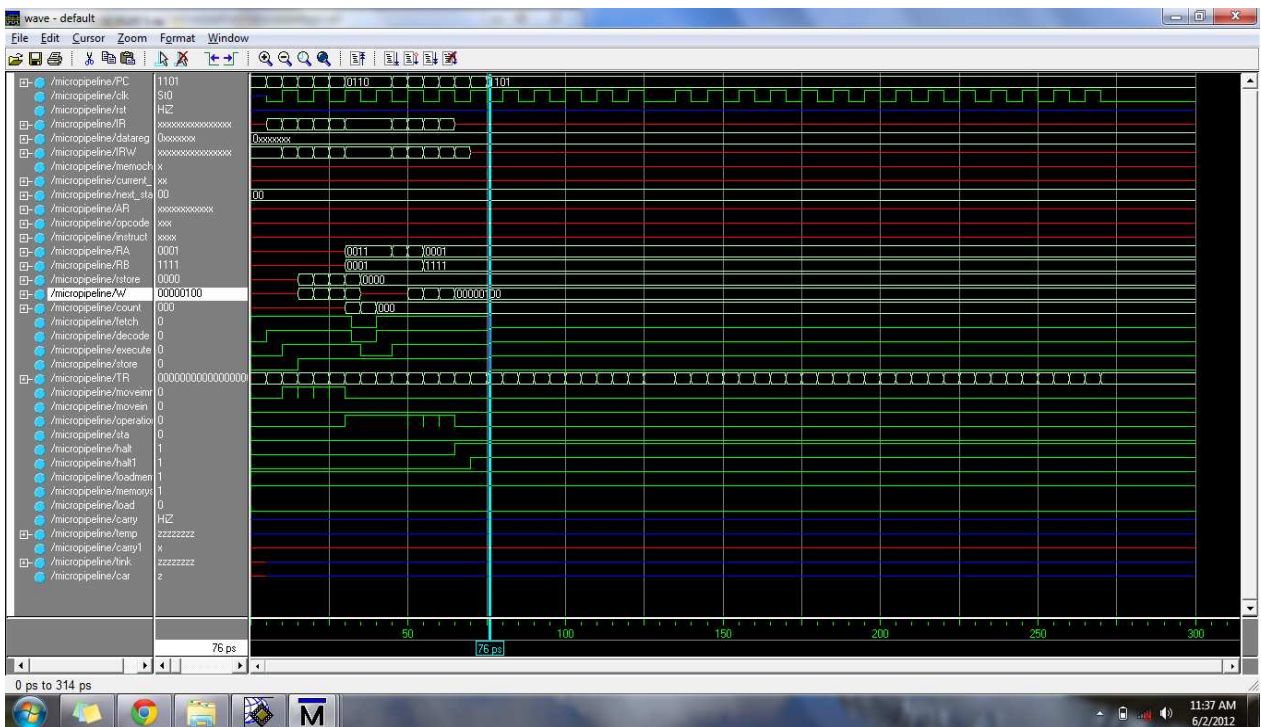
using the pipelined microprocessor presented in this paper. It can be seen the program execution time has reduced considerably to 76ps.

5.Conclusion

This paper presents a basic pipelined microprocessor which has been written in Verilog HDL. This is simulation is unconventional in a way such that it showcases the beauty which goes into implementing pipelining of instructions in a microprocessor without going into too many complexities of the same.Moreover,this paper may be very effectively used as a tool in   the field of education to introduce students to computer simulations of microprocessors. Students will definitely benefit from building upon this. The purpose of implementing an efficient microprocessor who's working and intricacies are easy to understand has been successfully achieved.

6.Acknowledgments

References

1. *Verilog HDL   (2nd Edition),Samir Palnitkar*
2. *Computer System Architecture (3rd Edition)   M Morris Mano,Prentice Hall (1993)*
3. *Computer Systems Organization and Architecture ,John D. Carpinelli*
4. Ronald, J. Tocci, Widmer, N. Moss, G. (1998), "Digital Systems Principles and Application", Prentice-Hall
5. International Inc., New Jersey, 182-341.

6. *Digital Design and Verilog HDL Fundamentals ,Joseph Canavagh,CRC Press(1st Edition 2008)*
7. *Verilog for Digital Design   Frank Vahid , Roman Lysecky ,Wiley*

Digital Design (4th Edition)M.Morris Mano, Prentice Hall; 4 edition (December 25, 2006)