Design of A Six-stage Pipelined MIPS Processor Based on FPGA

Qiao-Zhi Sun, De-Chun Kong, Cheng-Long Zhao, and Hui-Bin Shi

Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China

504914613@qq.com

hshi@nuaa.edu.cn

Abstract—**We design a 32-bit embedded six-stage pipelined processor which is compatible with MIPS instruction set. The six stages make the task of each stage balanced. We use forwarding and stalling to solve data hazards. Control hazards are solved by predicting which instruction should be fetched and when the pipeline will be flushed if the prediction is later determined to be wrong. The processor is implemented in DE2 development board, and its operating clock frequency can be up to 81.7MHz. In the end we present the comprehensive results of the design. Besides, we show the software simulation and hardware verification to prove the correctness of the design.**

***Keywords*—MIPS; embedded; pipelined processor; hazards; FPGA;**

I. Introduction

It proves that about 20% instructions in computer take the 80% of all the task by John Cocke, who first introduced the conception of RISC in 1947([1]), working in IBM research center in York of New York city. The first computer made by this theory is The IBM PC/XT in 1980. Later, IBM RISC System / 6000 also made use of this. The word RISC itself belongs to David Patterson a teacher at the university of California in Berkeley. The concept of RISC is also used by Sun's SPARC microprocessor, which promotes the technology of MIPS.

RISC technology has been an active field of computer development, especially in the embedded application. Today, almost all of the embedded microprocessor take RISC architecture in the market. These embedded microprocessors have been widely applied in real-time industrial control systems, multimedia, wireless network, and plays an important role as a core component of these systems. In this paper we design a 32-bit embedded six-stage pipelined processor with the high performance which is compatible with MIPS instruction set, based on analyzing the system structure of MIPS, combined with the flexibility characteristics of FPGA.

II. Design and Implementation

A. The design of the six- stage pipeline

The five-stage pipelined processor is classical pipelined processor, namely the data path is divided into five stages. In this paper, we design a six-stage pipeline processor. Specially, we call the six stages Fetch, Decode, Choose, Execute, Memory and Writeback.

In the Fetch stage, the processor reads the instruction from instruction memory. In the Decode stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In order to solve hazards, in the Choose stage, the processor selects the operands, which will be the two input ports of ALU in Execute stage. In the Execute stage, the processor performs a computation with ALU. In the Memory stage, the processor reads or writes the data memory. Finally, In the Writeback stage, the processor writes the result to the register file, when applicable. Figure 1 shows the six-stage pipeline diagram.

B. The design of control signal

As the key part of processor, the control unit computes the control signals based on the opcode and funct fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$.

Most of the control information comes from the opcode, but R-type instruction also use the funct field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 2. The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOP signal. The ALU decoder uses this ALUOP signal in conjunction with the funct field to compute ALUControl. The meaning of the ALUOp signal is given in table I.

Table Ⅱ is a truth table for the ALU decoder. Because ALUOp is never 11, the truth table can use do not care X1 and1X instead of 01 and 10 to simply the logic. When ALUOp is 00 or 01, the ALU should add or subtract, respectively. When ALUOp is 10, the decoder examines the funct field to determine the ALUControl. Note that, for the R-type instructions we implement, the first two bits of the funct field are always 10, so we may ignore them to simply the decoder.

TABLE II
ALUOP ENCODING

ALUOp	Meaning
00	add
01	subtract
10	Look at funct field
11	n/a

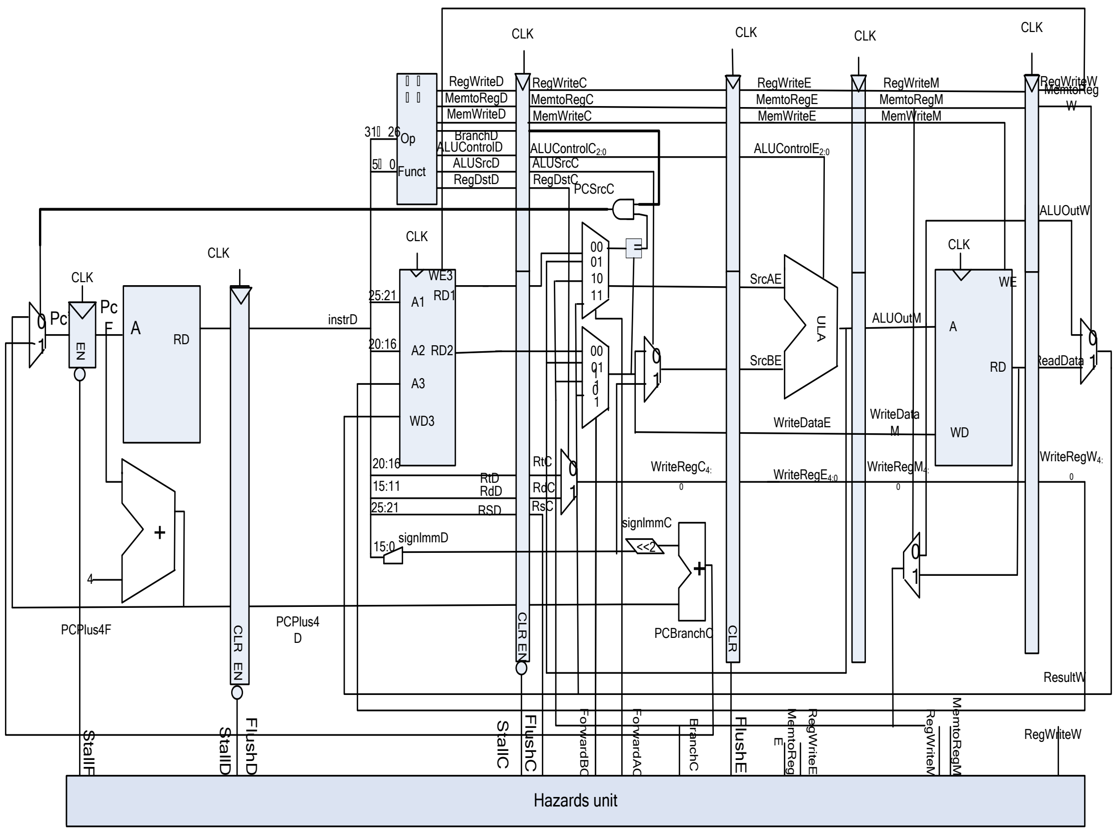


Fig. 1 The six stage pipeline diagram

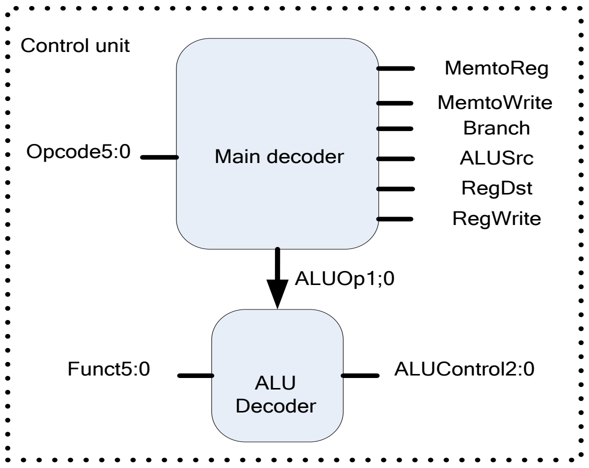


Fig. 2 Control unit internal structure

The control signals for each instruction were described as we built the datapath. Table III is the truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the ALU decoder output. For instructions that do not write to the register file(e.g., sw and beq), the RegDst and MemtoReg control signals are don't not cares(X); the address and data to the register write port do not matter because RegWrite is not asserted. The logic for the decoder can be designed using combinational logic design.

TABLE II
ALU DECODER TRUTH TABLE

ALUOp	funct	ALUControl
00	X	010(add)
X1	X	110(subtract)
1X	100000(add)	010(add)
1X	100010(sub)	110(subtract)
1X	100100(and)	000(and)
1X	100101(or)	001(or)
1X	101010(slt)	111(set less than)

TABLE III
MAIN DECODER TRUTH TABLE

instruction	opcode	RW	RD	ALUsrc	Branch	MW	MR	ALUOp
R-type	000000	1	1	0	0	0	0	10
Lw	100011	1	0	1	0	0	1	00
Sw	101011	0	X	1	0	1	X	00
Beq	000100	0	X	0	1	0	X	01

C. The solutions to the hazards

A central challenge in pipelined systems is handling hazards. In a pipelined system multiple instructions are handled concurrently. When one instruction is dependent on the results of another that not yet completed, a hazards occurs.

Hazards are classified as data hazards or control hazards

- data hazards: A data hazard occurs when an instruction tries to read a register that has not yet been written back by a previous instruction.
- control hazards: A control hazards occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place ([2]).

1) Data hazards:

The register file can be read and written in the same cycle. Let us assume that the write takes place during the first half of the cycle and the read takes place during the second half of the cycle, so that a register can be written and read in the same cycle without introducing a hazard.

Figure 3 illustrates hazards that occur when one instruction writes a register ($s0) and subsequent instruction read the register. This is called a read after write (RAW) hazard. The add instruction writes a results into $s0 in the first half of cycle 6. However ,the and instruction reads $s0 on cycle 3, obtaining the wrong value.

The or instruction reads $s0 on the cycle 4, again obtaining the wrong value. The sub instruction reads $s0 on the cycle 5, still obtaining the wrong value. The next instruction will reads the $s0 in the second half of the cycle 6, since data has been written in the first half of the cycle 6, subsequent instruction will read the correct value.

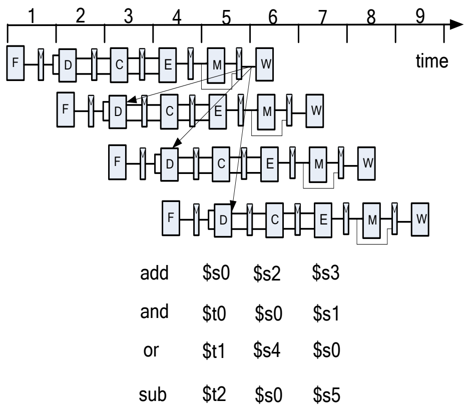


Fig. 3 Abstract pipelined diagram illustrating hazards

The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instruction read that register. Without specially treatment , the pipeline will compute the wrong result.

On a closer inspection, however, we observe that the sum from the add instruction is computed by the ALU in cycle 4 and is not strictly needed by the and instruction until the ALU uses it in cycle 5. In principle, we should be able to forward the result from one instruction to the next to solve the RAW hazard without slowing down the pipeline. Figure 4 shows that how to solve the data hazards with forwarding. In cycle 4, $s0 is forwarded from the Execute stage of the add instruction to the Choose stage of the and instruction. In cycle 5, $s0 is forwarded from the Memory stage of the add instruction to the Choose stage of the or instruction. In cycle 6, $s0 is forwarded from the Writeback stage of the add instruction to the Choose stage of the sub instruction.

There are two ways that the data obtained in the Memory stage. One way the value is the result that is computed by the ALU in the Execute stage, which will be written into the register. The other way the value is loaded from the memory only when execute lw instruction. In order to distinguish the value to be forwarded, we need add a two-input multiplexer in the Memory stage.

If the Execute stage, Memory stage and Writeback stage contain matching destination register together, the Execute stage should have highest priority, because it contains the

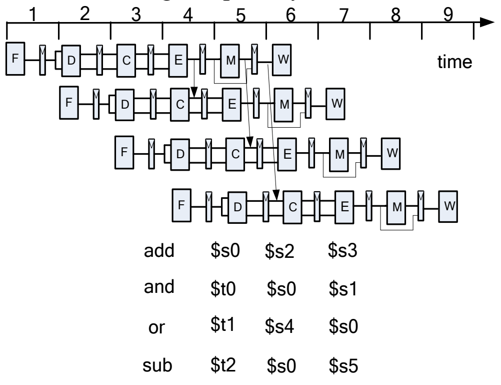


Fig. 4 Abstract pipelined diagram illustrating forwarding

more recently executed instruction, the Memory stage is second, the Writeback stage is the last.

In order to let the readers understand the signal used in the paper, we will give table IV below to explain the function of the signals used.

TABLE IV
HAZARD UNIT SIGNAL

Signal Name	**Functional Description**
forwardaC	A control signal to the four-input multiplexer
forwardbC	A control signal to the four-input multiplexer
lwstallC	Lw instruction stall signal
stallF	A control signal to stall Fetch stage register
stallD	A control signal to stall Decode stage register
stallC	A control signal to stall Choose stage register
flushE	A control signal to flush Execute stage register
flushD	A control signal to flush Decode stage register
flushC	A control signal to flush Choose stage register

In summary, the function of the forwarding logic for SrcA is given below. The function of the forwarding logic for SrcB is identical except that it checks rt rather than rs.

```
if(rsC!=0)
        if(rsC==writeregE&regwriteE)
                    forwardaC=2'b01;
        else if(rsC==writeregM&regwriteM)
                    forwardaC=2'b10
         else if(rsC==writeregW&regwriteW)
                    forwardaC=2'b11;
```

Forwarding is sufficient to solve RAW hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Choose stage of the next instruction. Unfortunately, the LW instruction does not finish reading data until the end of the Memory stage, so its results can not be forwarded to the Choose stage of the next instruction. We say that the lw instruction has two-cycle latency, because a dependent instruction cannot use its result until two cycles later.

Figure 5 shows this problem. The lw instruction receive data from memory at the end of cycle 5. But the and instruction needs that data as a source operand at the beginning of cycle 5. There is no way to solve this hazards with forwarding.

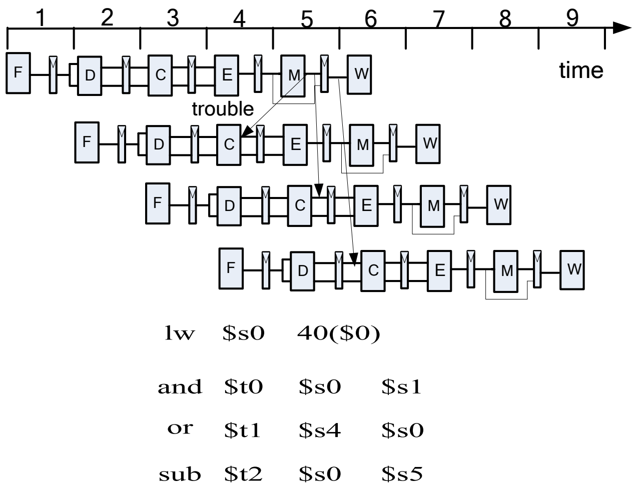


Fig. 5 Abstract pipelined diagram illustrating hazards from lw

The alternative solution is to stall the pipeline, holding up operation until the data is available. Figure 6 shows stalling the dependent instruction (and) in the Choose stage in cycle 4 and stalls there through cycle 4. The subsequent instruction (or) must remain in the Decode stage during both cycles as well, because the Choose stage is full. The next instruction (sub) must remain in Fetch stage.

In cycle 5, the result can be forwarded from Memory stage of the lw instruction to the Choose stage of the and instruction. In cycle 6, the result can be forwarded from Writeback stage of the lw instruction to the Choose stage of the and instruction.

In cycle 6, source $s0 of the sub instruction is read directly from the register file, with no need for forwarding.

When a lw stall occurs, stallC, stallD and stallF are asserted to force the Choose stage, Decode stage and Fetch stage to hold their old values. We also need flush the contents of the Execute stage pipeline register.

The MemtoReg signal is asserted for lw instruction. Hence, the logic to compute the stalls and flushes is

```
lwstallC=memtoregE&(writeregE==rsC|writeregE==rtC)
          stallC=lwstallC
           stallF=stallD=stallC
          flushE=stallC
```

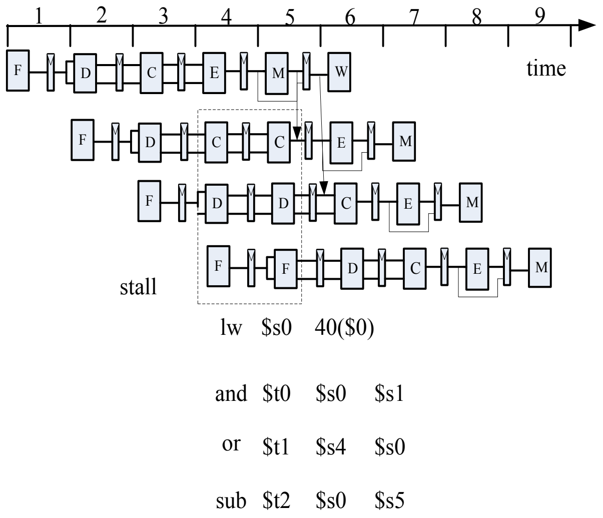


Fig. 6 Abstract pipelined diagram illustrating stall to solve hazards

2) Control hazards:

The beq instruction and jump instruction present a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch and jump decisions have not been made by the time the next instruction is fetched.

One mechanism for dealing with the control hazard is to stall the pipelined until the branch and jump decision is made. Because the decision is made in Memory stage, the pipelined would have to be stalled for four cycles at every branch or jump instruction. This would severely degrade system performance.

An alternative is to predict whether the branch will be taken and begin executing instruction based on the prediction. Due to jump instruction is always executed, we need not predict it.

Once the branch decision is available, the processor can throw out the instruction if the prediction was wrong. In particular, suppose that we predict that branches are not taken and simply continue executing the program in order. If the branch should have been taken, the four instructions following the branch must be flushed.

Figure 7 shows such a scheme, in which a branch from address 20 to address 64 is taken. The branch decision is not made until cycle 5, by which point the and, or, sub, and addi instructions at address 24, 28, 2C and 30 have already been fetched. These instructions must be flushed, and the slt instruction is fetched from address 64 in cycle 6. This is

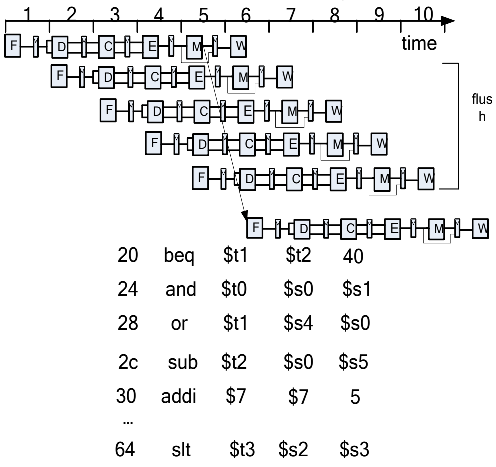


Fig. 7 Abstract pipelined diagram illustrating flushing

somewhat of an improvement, but flushing so many instructions when the branch is taken still degrades performance.

We could reduce the branch misprediction penalty if the branch decision could be made earlier. Making the decision simply requires comparing the values of two register. Using a dedicated equality comparator is much faster than performing a subtraction and zero detection.

Considering placing the comparator in the Decode stage will prolong the time of processor cycle. So we put the comparator in the Choose stage, so that the operands are passed from the register file and compared to determine the next PC by the end of the Choose stage.

Figure 8 shows the pipeline operation with the early branch decision being made in cycle 3. In cycle 4, the and and or instruction are flushed and the slt instruction is fetched. Now the branch misprediction penalty is reduced to only two instruction rather than four.

Besides we compute the transfer address of the jump instruction in the Choose stage. So we can make sure that the branch and jump instruction in the Choose stage. EqualC is the result of the comparator. The flush signals are given below.

pcsrcC= branchC&equalC
flushC=pcsrcC|jumpC
flushD=flushC

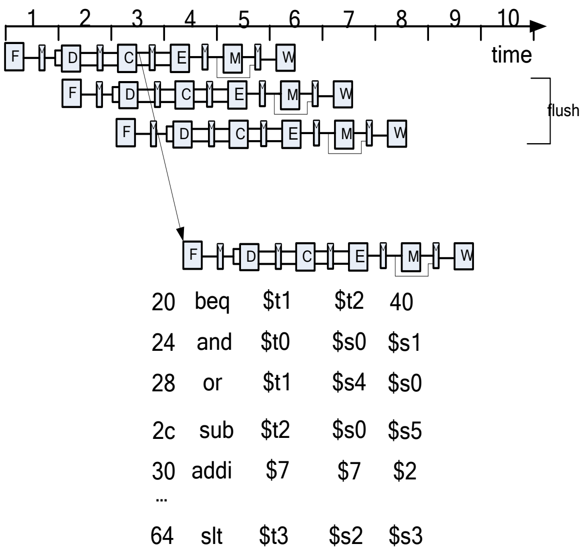


Fig. 8 Abstract pipelined diagram illustrating earlier branch decision

3) *Hazards summary:*

In summary, RAW data hazards occur when an instruction depends on the result of another instruction that has not yet been written into the register. The data hazards can be solved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction ([3], [4]).

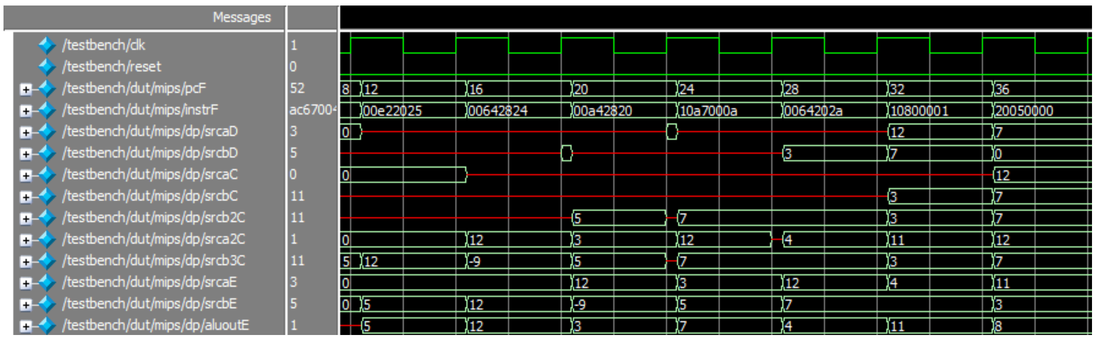

Messages
/testbench/clk 1
/testbench/reset 0
/testbench/dut/mips/pcF 52
/testbench/dut/mips/instrF ac6700
/testbench/dut/mips/dp/srcaD 3
/testbench/dut/mips/dp/srcbD 5
/testbench/dut/mips/dp/srcaC 0
/testbench/dut/mips/dp/srcbC 11
/testbench/dut/mips/dp/srcb2C 11
/testbench/dut/mips/dp/srca2C 1
/testbench/dut/mips/dp/srcb3C 11
/testbench/dut/mips/dp/srcaE 3
/testbench/dut/mips/dp/srcbE 5
/testbench/dut/mips/dp/aluoutE 1
8 12 16 20 24 28 32 36
00e22025 00642824 00a42820 10a7000a 0064202a 10800001 20050000
0 12 7
3 7 0
0 12
3 7
5 7 3 7
0 12 3 12 4 11 12
5 12 -9 5 7 3 7
0 12 3 12 4 11
0 5 12 -9 5 7 3
5 12 3 7 4 11 8

Fig. 9 the wave of simulation

III. SIMULATION AND SYNTHESIS

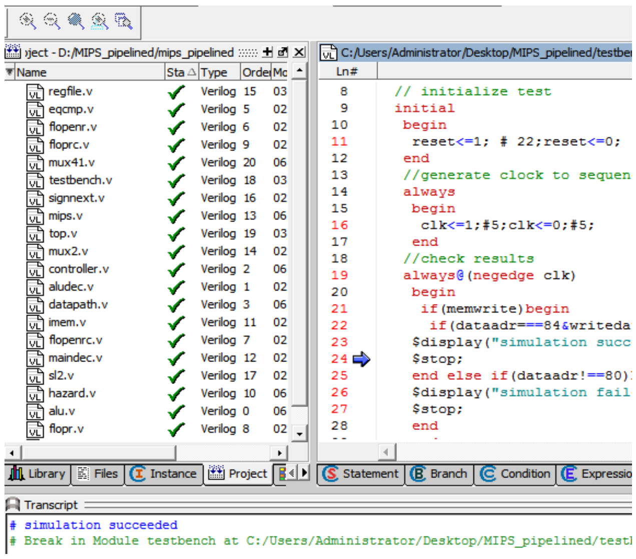
A Simulation In Modelsim

We use verilog language to describe the design of six-stage pipelined in this paper, with the ModelSim to simulate.

Figure 9 shows the waveform of the simulation. InstrF is the instruction which is fetched from instruction memory in the Fetch stage. The clk is the clock used in simulation.We can see from the Figure 9 one instruction is fetched in every cycle.

Because the simulation in ModelSim is functional simulation, it does not take the sequential design into consideration.

Figure 10 show the project of the six-stage pipelined. It contains many modules. The hazards unit is key module in the pipelined processor.


ɔject - D:/MIPS_pipelined/mips_pipelined

Name	Sta	Type	Order	Mo
regfile.v	✓	Verilog	15	03
eqcmp.v	✓	Verilog	5	02
flopenr.v	✓	Verilog	6	02
floprc.v	✓	Verilog	9	02
mux41.v	✓	Verilog	20	06
testbench.v	✓	Verilog	18	03
signnext.v	✓	Verilog	16	02
mips.v	✓	Verilog	13	06
top.v	✓	Verilog	19	03
mux2.v	✓	Verilog	14	02
controller.v	✓	Verilog	2	06
aludec.v	✓	Verilog	1	02
datapath.v	✓	Verilog	3	06
imem.v	✓	Verilog	11	02
flopenrc.v	✓	Verilog	7	02
maindec.v	✓	Verilog	12	02
sl2.v	✓	Verilog	17	02
hazard.v	✓	Verilog	10	06
alu.v	✓	Verilog	0	06
flopr.v	✓	Verilog	8	02

C:/Users/Administrator/Desktop/MIPS_pipelined/testbe

```
// initialize test
initial
 begin
  reset<=1; # 22;reset<=0;
 end
 //generate clock to sequen
 always
  begin
   clk<=1;#5;clk<=0;#5;
  end
 //check results
 always@(negedge clk)
  begin
   if(memwrite)begin
    if(dataadr===84&writeda
  $display("simulation succ
  $stop;
  end else if(dataadr!==80)
  $display("simulation fail
  $stop;
  end
```

Transcript

```
# simulation succeeded
# Break in Module testbench at C:/Users/Administrator/Desktop/MIPS_pipelined/testl
```

Fig. 10 the project structure diagram

The hazard module codes are given below. The suffix F, D, C, E, M and W imply the signal is used in Fetch stage, Decode stage, Choose stage, Execute stage, Memory stage and Writeback stage.

```
module hazard (input [4:0] rsC,rtC,
                input [4:0] writeregE,
               writeregM,writeregW,
               input regwriteE,regwriteM,regwriteW,
                input memtoregE,
               output reg [1:0] forwardaC,forwardbC,
               output stallF,stallD,stallC,flushE);
               wire lwstallC;
//forwarding sources to C stage (choose)
      always@(*)
      begin
      forwardaC=2'b00;forwardbC=2'b00;
               if(rsC!=0)
               if(rsC==writeregE&regwriteE)
                     forwardaC=2'b11;
      if(rtC!=0)
               if(rtC==writeregE&regwriteE)
                          forwardbC=2'b01;
               else if(rtC==writeregM&regwriteM)
                           forwardbC=2'b10;
               else if(rtC==writeregW&regwriteW)
                         forwardbC=2'b11;
   end
   // stalls
      Assign
lwstallC=memtoregE&(writeregE==rsC|writeregE==rtC);
      assign #1 stallC=lwstallC;
      assign #1 stallD=stallC;
      assign #1 stallF=stallD;
      //stalling C stalls all previous stages
      assign #1 flushE=stallC;
   endmodule
```

The control module also plays an important role in pipelined processor. The control unit examines the opcode and funct fields of the instruction in Decode stage to produce the control signals. We can see it clearly in figure 1. This control signals must be pipelined along with the data so that they remain synchronized with instruction.

In order to understand how the control signals are passed, we will give the code of control module below.

```
   module controller(input clk,reset,
                    input [5:0] opD,functD,
                    input flushE,stallC,flushC,equalC,
               output memtoregC, memtoregE,memtoregM,
                output memtoregW,memwriteM,
                output pcsrcC,branchC,alusrcC,
                output regdstC,regwriteC,regwriteE,
                output regwriteM,regwriteW,
                output jumpC,
                output [2:0] alucontrolE);
      wire [1:0] aluopD;
      wire memtoregD,memwriteD,alusrcD,
                regdstD,regwriteD;
      wire [2:0] alucontrolD,alucontrolC;
      wire memwriteC,memwriteE;

      maindec md(opD,memtoregD,memwriteD,branchD,
                          alusrcD,regdstD,regwriteD,jumpD,
                           aluopD);

      aludec ad(functD,aluopD,alucontrolD);
      assign pcsrcC= branchC&equalC;

      //pipeline registers
      flopenrc #(10) regC(clk,reset,~stallC,flushC,
                          {memtoregD,memwriteD,alusrcD,
      regdstD,regwriteD,alucontrolD,jumpD,branchD},
                          {memtoregC,memwriteC,alusrcC,
      regdstC,regwriteC,alucontrolC,jumpC,branchC});
```

```
        floprc #(6) regE(clk,reset,flushE,
                             {memtoregC,memwriteC,
                             regwriteC,alucontrolC},
                             {memtoregE,memwriteE,
                             regwriteE,alucontrolE});
        flopr #(3)  regM(clk,reset,
                     {memtoregE,memwriteE,regwriteE},
                     {memtoregM,memwriteM,regwriteM});

        flopr #(2)   regW(clk,reset,
                     {memtoregM,regwriteM},
                     {memtoregW,regwriteW});
endmodule
```

B Synthesis In FGA

If the project simulation succeeds in Modelsim, it implies that the design is correct in logic. The next step we can create a new project in QuartusⅡadding the contents of the project from Modelsim.

The complication report of the six-pipelined processor is given in Table V. We use the DE2 development board of Cyclone Ⅱ series. It can be seen from the table the total resources used in design are less than 3%. The lower processor resource utilization will lay the foundation for constructing a SoPC system using processor as the core.

TABLE V
COMPLICATION REPORT-FLOW SUMMARY

Resources	**Type or the use of resources**
Family	Cyclone Ⅱ
Device	EP2C35F672C6
Met timing requirements	Yes
Total logic elements	914/33,216(3%)
Total combinational functions	759/33,216(2%)
Dedicated logic registers	539/33,216(2%)
Total registers	539
Total pins	68/475(14%)
Total memory bits	4096/483,840(<1%)

Besides, the integrated module is given in Figure 11. It contains three sub module: imem module, dmem module and mips module. Imem is instruction memory used in Fetch stage. Dmem is data memory used in Memory stage. Mips is responsible for executing instructions.

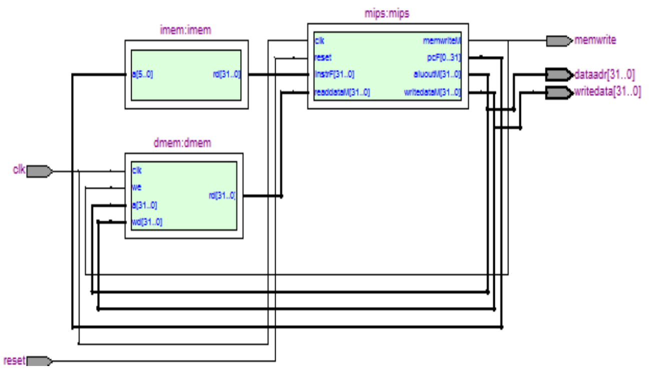


Fig. 11 the integrated module diagram

In this design, the instruction memory and data memory use logic elements in DE2 board to accomplish. The alternative is to use the memory device in DE2 board. There are three types of memory to select in DE2 board.

SRAM

- 512-Kbyte Static RAM memory chip
- Organized as 256K x 16 bits
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

SDRAM

- 8-Mbyte Single Data Rate Synchronous Dynamic RAM memory chip
- Organized as 1M x 16 bits x 4 banks
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

Flash memory

- 4-Mbyte NOR Flash memory (1 Mbyte on some boards)
- 8-bit data bus
- Accessible as memory for the Nios II processor and by the DE2 Control Panel

Table Ⅵ shows the pin allocation of MIPS processor in the DE2 development board. The frequency can reach 81.70MHZ.

TABLE Ⅵ
COMPLICATION REPORT-FLOW SUMMARY

function	**The port of design**	**PINS on DE2**
Digital oscillator （50 MHZ）	clock	PIN_N2
Reset signal	reset	PIN_N25
Verifying the led signal	led	PIN_AA13

We need to load a section of code to the instruction memory at first when verify the design. The instructions are composed of add instruction, sub instruction, and instruction, or instruction, slt instruction, addi instruction, lw instruction, sw instruction, beq instruction, jump instruction and so on at random([5], [6]).

Only when all instructions are executed correctly, can we get correct result. As for this paper, If the program runs correctly, we need to write the value 7 into address 84 in the memory. The check codes are given below.

```
always@(negedge clk)
begin
   if(memwrite)begin
   if(dataadr===84&writedata===7)
     led=1;
     end
    else if(dataadr!==80)begin
         led=0;
         end
end
```

After configuring, we download the design to the DE2 development board. We can see the led lights implying led=1 so the design of the pipeline is correct.

IV. Summary

Design of pipelined processors is the main method to improve the work efficiency of the processor. In this paper we design a six-stage pipelined processor originally. It divides the processor into six stages according to its function and makes each stage balanced. We also verify it in the FPGA development board to ensure the correctness of design.

After synthesis, the work frequency can reach 81.7MHZ. The realization of the design lays the foundation for studying more complex and efficient processor design.

The most paper we have seen about the six-stage pipeline processor is the students in Tsinghua University, as described in [7]. The work frequency of their can reach 53.6MHZ, compared with them the design in this paper improve the Performance.

Acknowledgment

Many thanks to Yang Ding-ding, Ying Juan for their contributions to the research work at this system.

Many thanks to Altera Corp. for supplying excellent FPGA chips and development board.

References

[1] (2013) The 360 website. [Online]RISC. http://baike.so.com/.

[2] David Money Harris and Sarah L.Harris, *Digital Design and Computer Arccchitecture*, America:Proprietor, 2007.

[3] ZhouNi, QiaoFei, TanSisi, LI Chang, and YangHuazhong, "Design for Testability of a32-Bit MIPS Processor and Its Implementation," *Microelectronics.*, vol. 40, pp. 782–791, Dec. 2010.

[4] Xue Bo, Zhou Yu-jie, "Design of CPU Simulator Compatible with MIPS32 Instruction Set," *Computer Engineering.*, vol. 35, pp. 263–265, Jan. 2009.

[5] Bai Zhongying，Dai Zhitao and Zhou Feng, *Principles of computer composition*, 4th ed., Bai Zhongying, Ed. Beijing, China: Science Press, 2008

[6] Yang Quansheng， Wang Xaowei and Zhang Zhizhen, *The comprehensive curriculum design of computer systems*, 1st ed., Yang Quansheng., Ed. Beijing, China: Tsinghua University press, 2008.

[7] Li Chang, "Design of MIPS EmbeddedMicroprocessor," M. Eng. thesis, Tsinghua University, Beijing, China, May. 2010.