MIPS's Pipelined Processor Design

Course Project for Computer Organization and Principle

Class B of Computer Science and Technology
Author: Zhang Huazhi 08380131
Instructor: Li Guozhen
2010-1-6

MIPS's Pipelined Processor Design

Computer Science and Technology Class B 08380131 Zhang Huazhi

1. Background

In the course Computer Organization and Principle, I have learned a lot about the five classic components of a computer. But I want to know more about how they work together, how the processor implements instructions and how the control signals affect the process. Through the study of the book *Computer Organization and Design* by David A. Patterson and John L. Hennessy, I get to know the design of the datapath and control signals from the single-cycle CPU, to multi-cycle, and then to pipelined CPU. I'm amazed to discover how complicated and precise the processor is. With the knowledge acquired from the book, I found it possible to design a CPU of my own.

I choose the MIPS architecture and pipelining technology as the features of my CPU because they are in consistent with my wish to build a processor with reduced structure and enhanced performance. What's more, they are seldom referred to in the textbook even though they are important concept or technology. I think it's necessary to enrich my knowledge about MIPS and Pipelining by designing a CPU featuring them.

Pipelining is a technique that increases the number of simultaneously executing instructions and increase the throughput rather than individual instruction execution time. To design a pipelining processor, we have to take instruction sets into consideration because inappropriate use of certain instruction sets can make the design harder. For example, some Instruction Sets may sometimes need a series of interlocks that are the major performance barrier since they had to communicate to all the modules in the CPU and limit the clock speed. A major aspect of the MIPS architecture was to fit every sub-phase, including cache-access, of all instructions into one cycle, thereby removing any needs for interlocking, and permitting a single cycle throughput. So, I choose the MIPS architecture for my pipelined CPU in order to reduce the hazards of pipelining.

2. Technical Method

The MIPS's pipeline is a five-stage pipeline that initiates an instruction at every other stage. Since it's difficult and error-prone to design a pipelined CPU directly, I decided to finish the design of a single-cycle CPU first and then add pipeline registers, forwarding path, data hazard detection and flushing instructions on branch hazards to build the pipelined processor.

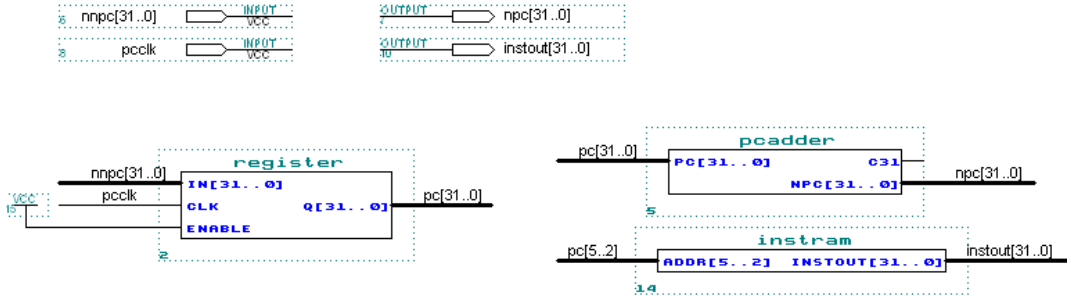
I choose MAX+PLUS II as the software for designing, compiling and simulating. As for the design of individual element, I use both VHDL and graphic editing. I use VHDL to create the control unit and the forward logic while design other elements in the graphic way. My design is mostly based on the book *Computer Organization and Design* by David A. Patterson and John L. Hennessy. I also got help from the book by Zhu Ziyu and Li Yamin and the paper by Li Weichao.

3. The Designing Process

Step1: Building single-cycle datapath and control unit. I separate the single-cycle

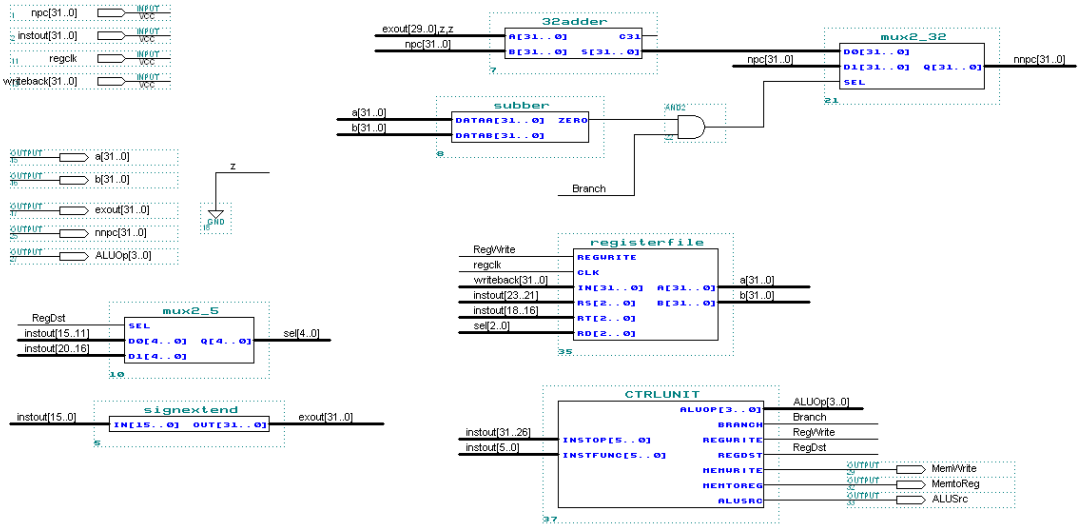
datapath into five stages in order to make it easier to transform it into a pipelined datapath.

(1) The IF(Instruction Fetch) stage



The PC (program counter) is a 32-bit register that will be written at the end of every clock cycle. The PCADDER is a 32-bit adder that performs an addition of the value of PC and 4. The INSRAM is read-only and do not need any clock or control signal. I use the RAM in the LPM library for ISTRAM.

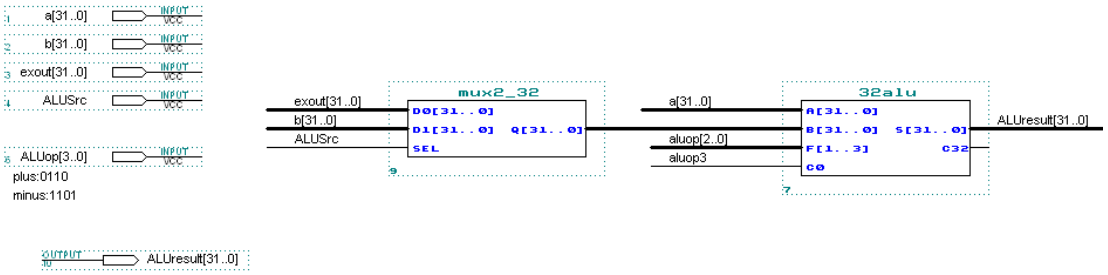
(2) The ID (Instruction Decode) stage



The REGISTERFILE should have contained 32 registers, but if so, the pin number will be too large for the software to compile. Therefore, I reduce the number of registers to 8 and use the lower 3 bits of the 5-bit register number input. In order to deal with the immediate number, a sign-extend unit is needed. The output of it has two ways to go, the first is to act as the relative address for SW or LW instruction, the other is to serve as the branch instruction for the BEQ instruction after shifted 2 bits to the left. The SUBBER on the top right can judge whether the two read output of the registerfile are equal. The signal from the SUBBER, combined with the branch signal, will determine whether to implement branch or not.

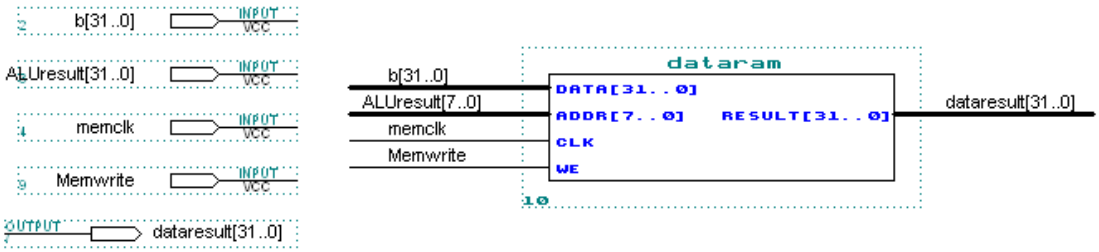
The control unit is difficult to design by graphic editing. So I use VHDL to create it. Since I put more emphasis on the architecture of my CPU rather than the number of functions, I only made it implement 5 instructions: ADD, SUB, LW, SW, and BEQ. And the control unit is designed only to deal with these five instructions, much simpler than the real control unit. The VHDL code can be found in Appendix A.

(3) The EX (Execute) stage



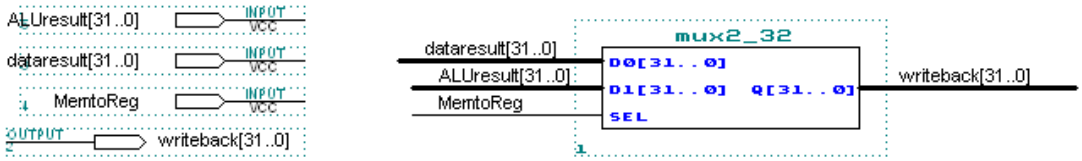
The ALU is 32 bits wide. The operation to be performed by the ALU is controlled by the ALU operation signal, which will be 4 bits wide. The input of the B port is chosen by a MUX 2_32. The ALUSRC signal determined whether the sign-extended immediate number or the value of the RT register is to be computed.

(4) The MEM (Memory Access) stage



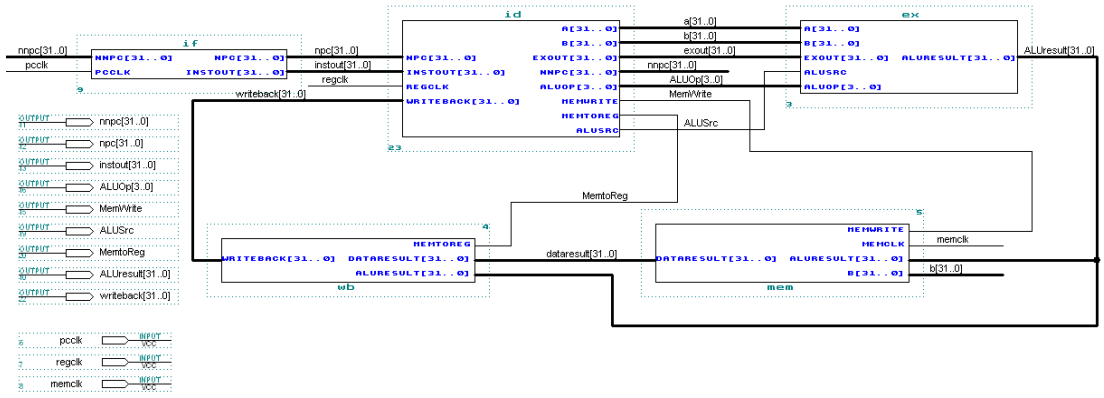
The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. The write control signal and clock signal are essential for data writing.

(5) The WB (Write Back) stage



The MUX 2_32 determines which input data will be written back to the registerfile. The MEMTOREG signal is used as the select signal.

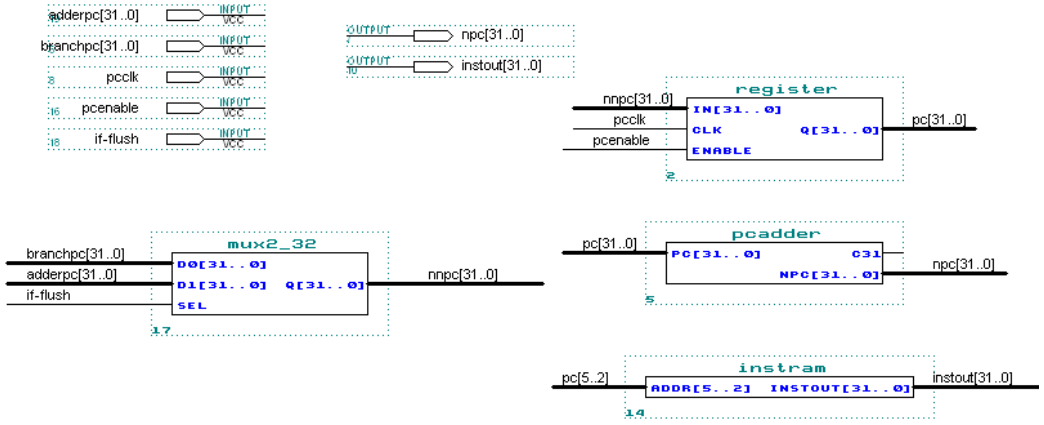
(6) The whole datapath and control for a single-cycle CPU



Step2: Transform the single-cycle CPU into a pipelined CPU

I will illustrate the process in the same five stages, only describing the addition to the previous design and the features of pipelining.

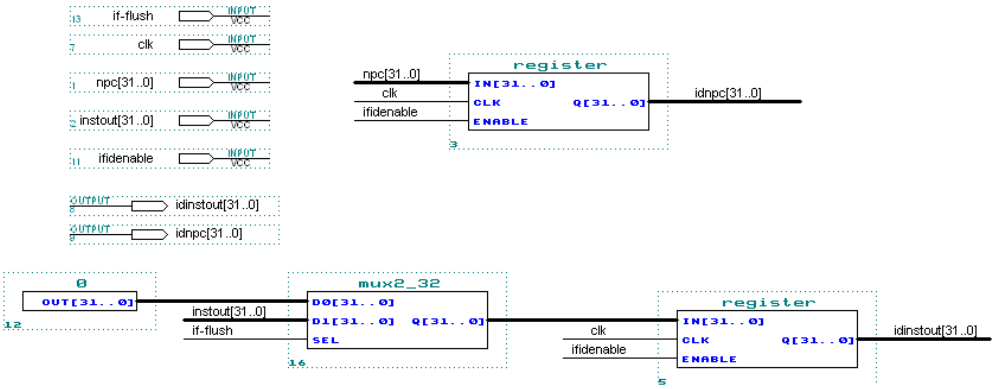
(1) The IF(Instruction Fetch) stage



A MUX 2_32 is added to select whether the next PC is a branch address or the address of the next instruction. This is controlled by the IF-FLUSH signal. Although this is a little change, it's essential because this is a necessary part of resolving branch hazards.

The PCENABLE signal can prevent the PC register from changing in order to resolve data hazards by stalling.

(2) The IF/ID pipeline register

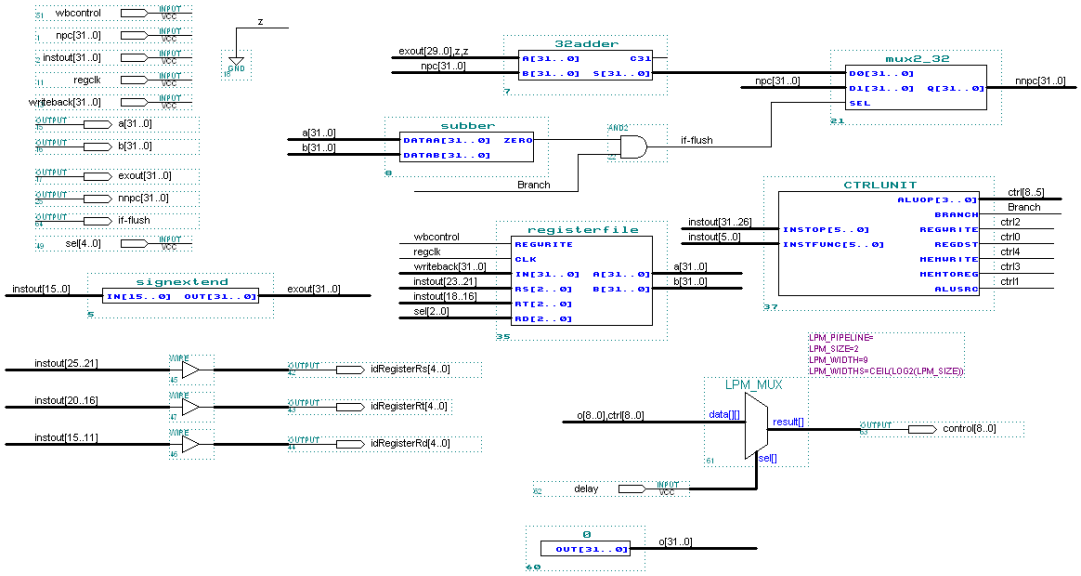


The pipeline register is used to hold the data of one instruction temporarily. We need four pipeline registers for the whole datapath: IF/ID, ID/EX, EX/MEM and MEM/WB. The pipeline registers separate each pipeline stage. All of them, except IF/ID, consist of registers only. So I will not show the graphic design of those three. The IF/ID register is special because it plays a role in resolving the data hazards by stalling. When a stall is needed, the IFIDENABLE signal will be deasserted in order to prevent the contents of the IF/ID register from changing.

Not only can the IF/ID register deal with data hazards, it can also play a main role in resolving branch hazards, I use the Assume Branch Not Taken scheme to resolve branch hazards. The IF-FLUSH signal can reset the instruction fetched out

and create a “bubble” when a branch happens.

(3) The ID (Instruction Decode) stage

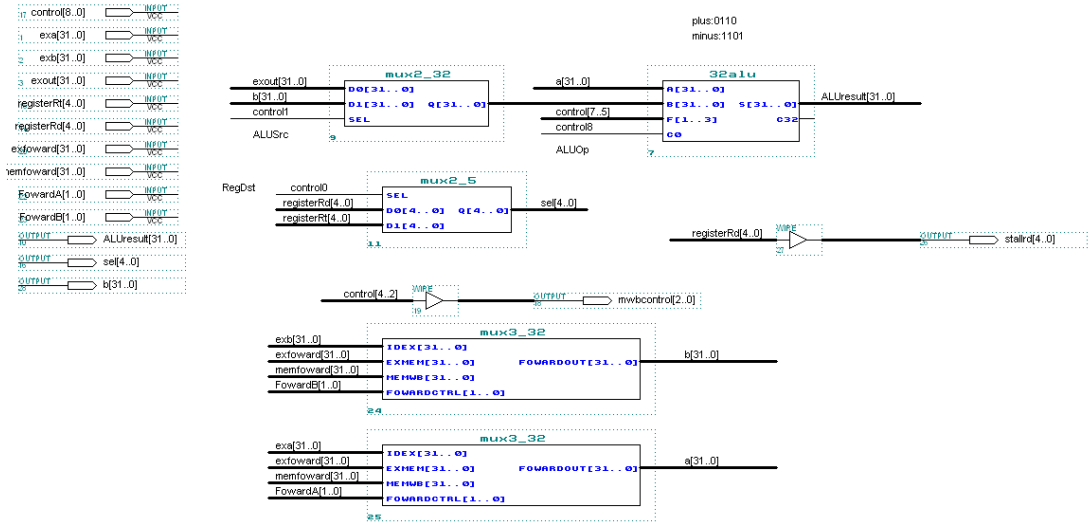


Drastic change has been made in this stage.

Firstly, the register number of RS, RT and RD and the control signals generated by the control unit should be passed to the next stage since they will be used for the Forward Logic and the writeback process. From this point, we can find out one of the main differences between the single-cycle datapath and the pipelined one, i.e. we have to pass all the data and control signals of a instruction to the next stage if they are need in later stages because, unlike single-cycle datapath, we cannot retrieve the data at their previous place when they are needed, they will be replaced by the data or signals of the next instruction.

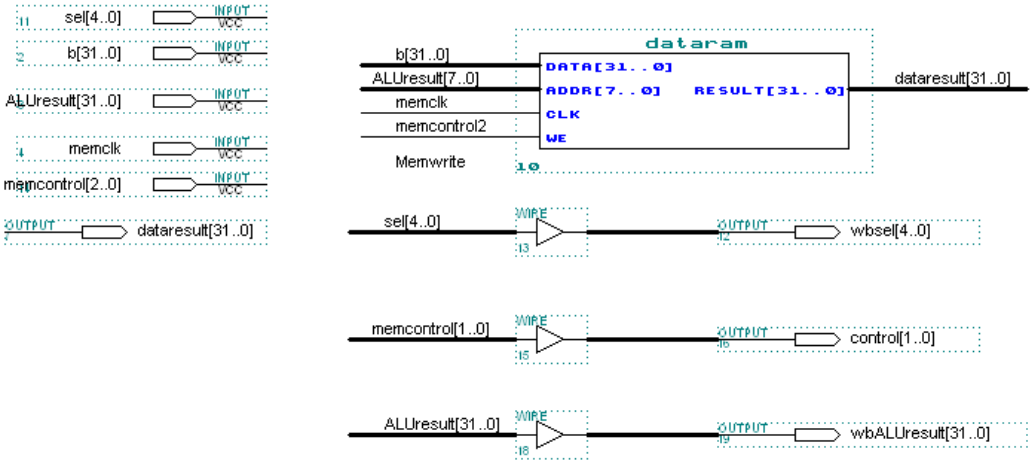
Secondly, I added a MUX 2_9 on the bottom right. This is necessary for resolving data hazards by stalling, the delay signal will be asserted when a stall is needed and all the 9 control signals will be changed to 0. As I described above, three actions (deasserting the enable signal of the PC and IF/ID, change all control signals to 0) lead to the generation of a “bubble” in the datapath.

(4) The EX (Execute) stage



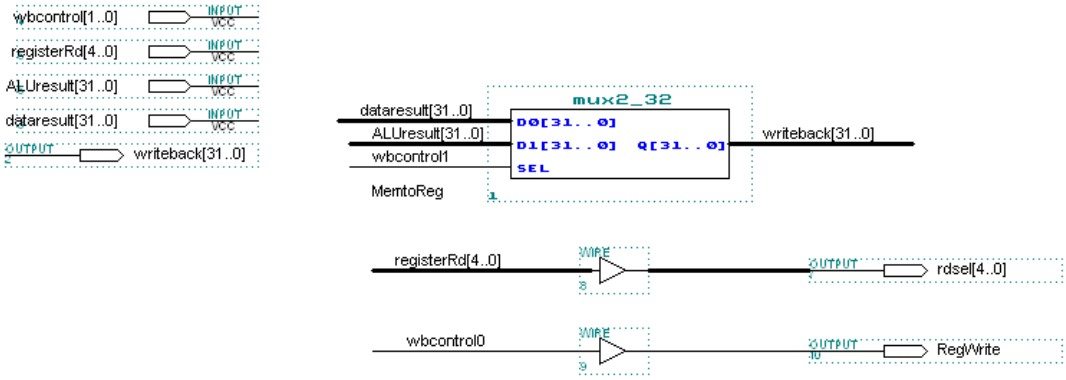
The EX stage is where data hazards frequently happen. We can solve these hazards by “forwarding”. The design of the Forward Logic will be shown later. Two MUX 2_32 is used to select either the register file value or one of the forwarded values. Just like I did in the ID stage, the control signals and data needed in later stages are passed to the next stage.

(5) The MEM (Memory Access) stage



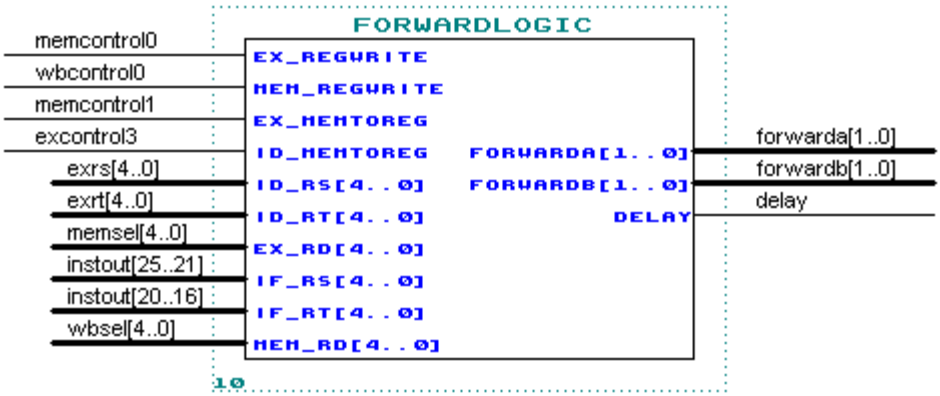
This stage has little change compared to the single-cycle datapath except that essential data and signals are passed to the WB stage.

(6) The WB (Write Back) stage



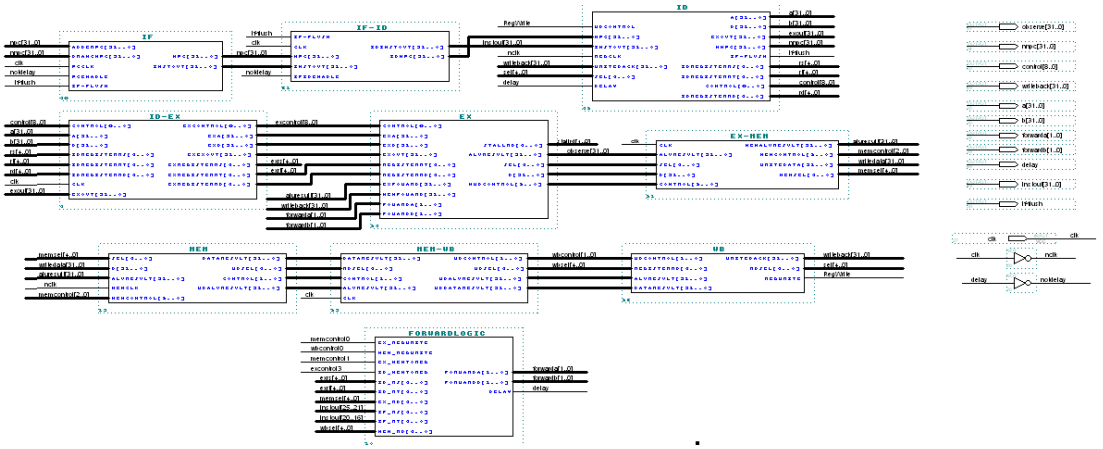
This stage is almost the same as the singly-cycle one.

(7) The design of the Forward Logic



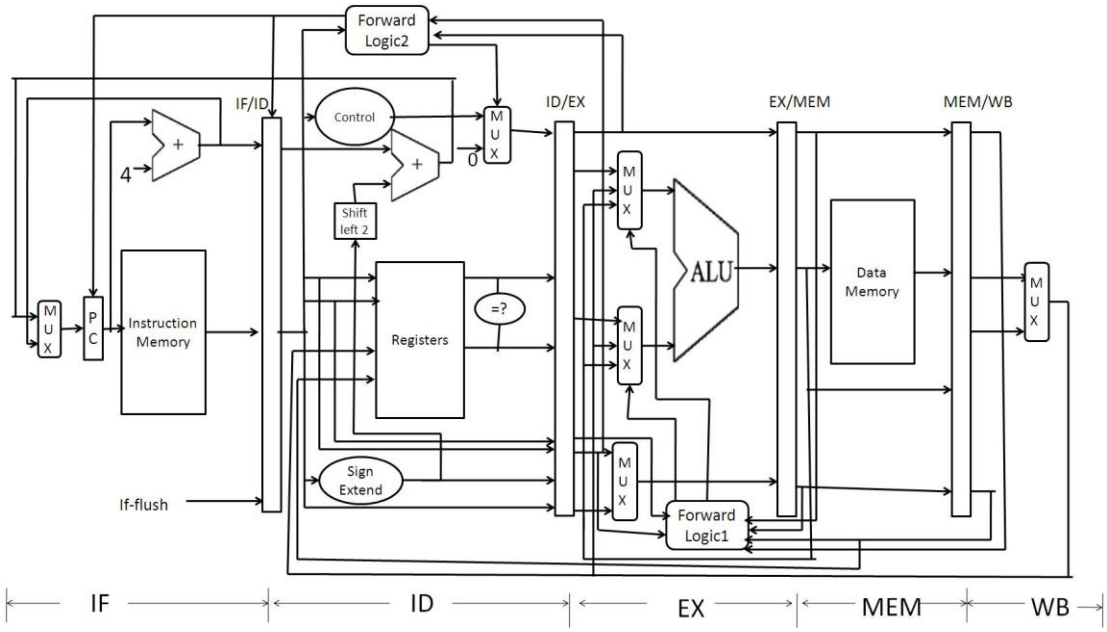
The Forward Logic has two functions; the first one is to generate forward signals to resolve data hazards by forwarding, while the other is to generate delay signals to resolve data hazards by stalling. It needs a lot of input ports which contain the essential control signals and the number of certain registers. The VHDL code for the design of this unit is shown in Appendix B.

(8) The whole datapath and control for a pipelined CPU



I cannot show it clearly due to limited space. The graphic editor file is “pipeline.gdf”.

The structure of the whole pipelined processor is shown below (some control signals omitted)



Step3: Performance Simulation

(1) The instructions

My instruction sequence consists of 5 kinds of instructions: LW, SW, ADD, SUB and BEQ. I carefully arrange their order so that all kinds of operation and all kinds of data

hazards and branch hazards can be tested when simulating.

MIPS Instruction Sequence	Registers				
	$1	$2	$3	$4	$5
00 Lw $1, 20($0) 35 16 17 20	6	0	0	0	0
04 Lw $2, 21($0) 35 16 18 21	6	5	0	0	0
08 Lw $3, 22($0) 35 16 19 22	6	5	6	0	0
0C Lw $4, 23($0) 35 16 20 23	6	5	6	2	0
10 Lw $5, 24($0) 35 16 21 24	6	5	6	2	1
14 Sub $2, $1, $4 0 17 20 18 0 34	6	4	6	2	1
18 Add $4, $2, $5 0 18 21 20 0 32	6	4	6	5	1
1C Lw $2, 20($1) 35 17 18 20	6	2	6	5	1
20 Add $4, $2, $5 0 18 21 20 0 32	6	2	6	3	1
24 Beq $1, $3, 12 4 17 19 3	6	2	6	3	1
28 Add $0,$0,$0 0 0 0 0 0 32	6	2	6	3	1
2C Add $0,$0,$0 0 0 0 0 0 32	6	2	6	3	1
30 Add $0,$0,$0 0 0 0 0 0 32	6	2	6	3	1
34 Sw $4,50($3) 43 19 20 50	6	2	6	3	1
38 lw $1,50($3) 35 19 17 50	3	2	6	3	1
3C Add $0,$0,$0 0 0 0 0 0 32	3	2	6	3	1

(2) Create the "MIF" file to initialize the memory

Since there is a data memory and an instruction memory in the datapath, two MIF files are needed. I name them to be "dataram.mif", "instram.mif".

Dataram.mif:

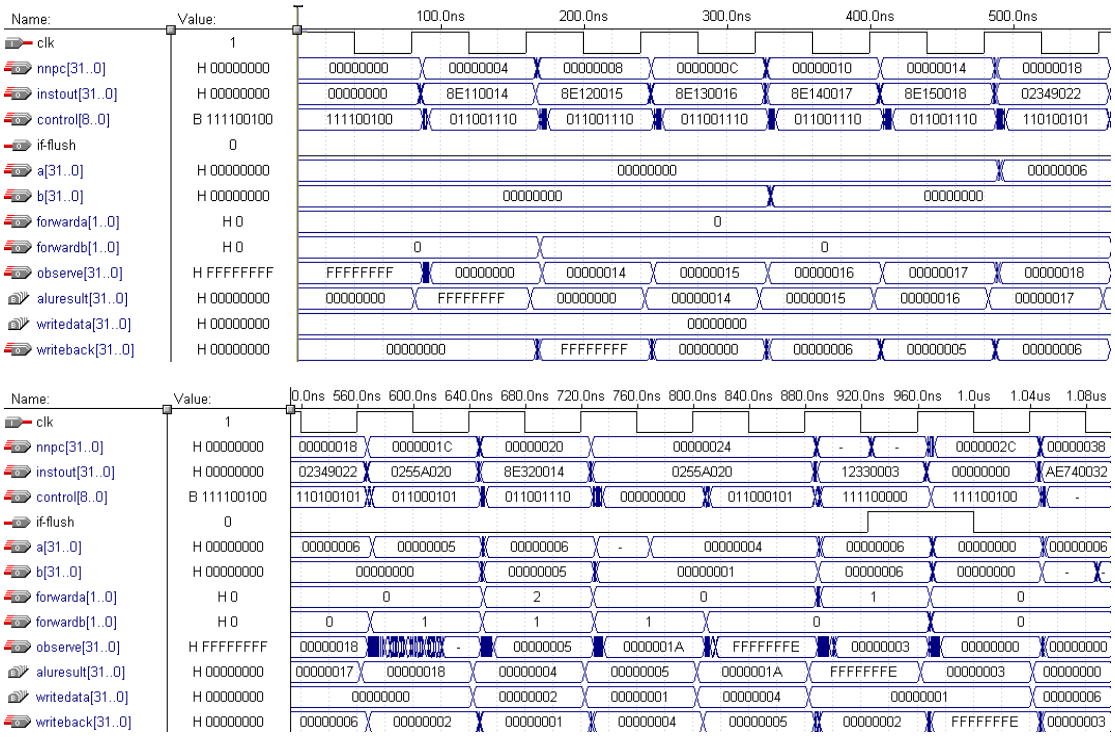
```
WIDTH = 32;
DEPTH = 256;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
     0    :    00000000;

    ......

--   :    --------;
14   :    00000006;
15   :    00000005;
16   :    00000006;
17   :    00000002;
18   :    00000001;
19   :    00000000;
1a   :    00000002;
1b   :    00000000;

    ......
```

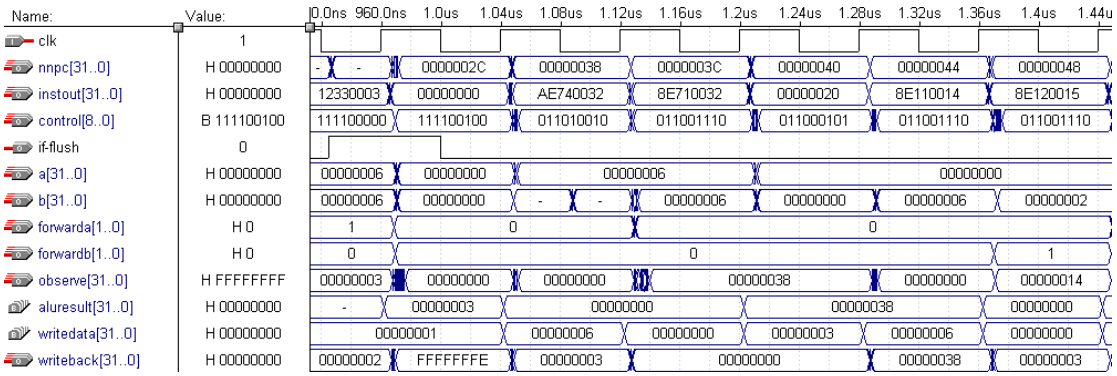
Instram.mif:

```
WIDTH = 32;
DEPTH = 16;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
     0    :    8e110014;
     1    :    8e120015;
     2    :    8e130016;
     3    :    8e140017;
     4    :    8e150018;
     5    :    02349022;
     6    :    0255a020;
     7    :    8e320014;
     8    :    0255a020;
     9    :    12330003;
     a    :    00000020;
     b    :    00000020;
     c    :    00000020;
     d    :    ae740032;
     e    :    8e710032;
     f    :    00000020;
END;
```

(3) Simulation

The clock cycle time is 80ns, so it takes 1.45μs to finish implementing these 16 instructions.





From the waveform editor file, we can find out that the pipelined CPU works well. It got all the results right and solved all data hazards and branch hazards successfully.

4. Effectiveness Evaluation

The processor I have designed has all the basic components of a CPU. It can implement an MIPS instruction from instruction fetching to writing back. As a result of the lack of time, I can only design a CPU that executes five kinds of MIPS instructions and the ALU cannot execute multiplication or division. However, based on the structure I've already built up, we can extend the function easily.

What's more, my CPU can avoid errors caused by data hazards and branch hazards which are the main troubles of pipelined CPU. The Forward Logic is simple but effective. Given enough time, the CPU can be adjusted to deal with exceptions as well. Pipelining improves the average execution time per instruction. The clock cycle time of my single-cycle CPU is 160ns while that of the pipelined CPU is 80ns. The performance of a pipelined processor can be enhanced with more advanced pipelining technology.

It's a complicated and time-consuming process to build a processor on my own. It requires careful thinking and meticulous design. In spite of all the difficulties and obstacles popping up at any time, I managed to finish my course project in two weeks' time. The process of making the CPU reinforces my knowledge on computer organization and will remain an exciting memory of mine.

I carried out my design mostly according to the guidance of the book Computer Organization and Design by Patterson and Hennessey. So, the processor reflects my understanding of the book's contents. I try to use the same port name in my design so that my CPU can be a reference for students who will use this book as textbook.

Appendix A: VHDL code for control unit

```
--Control Unit
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY CtrlUnit IS
PORT
(
 instOP              : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
 instFUNC       : IN STD_LOGIC_VECTOR (5 DOWNTO 0);

 ALUOP               : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
 BRANCH              : OUT STD_LOGIC;
 REGWRITE            : OUT STD_LOGIC;
 REGDST              : OUT STD_LOGIC;
 MEMWRITE            : OUT STD_LOGIC;
 MEMTOREG            : OUT STD_LOGIC;
 ALUSRC              : OUT STD_LOGIC
);
END ENTITY CtrlUnit;

ARCHITECTURE SelectResult OF CtrlUnit IS
BEGIN
 ALUOP <=      "1101" WHEN (
                        instOP = "000000" and instFUNC = "100010"--A 减 B，有符号
              ) else

 "0110" WHEN (
                        (instOP = "000000" and instFUNC = "100000")
 or     (instOP = "101011" ) -- SW
                    or     (instOP = "100011" ) -- LW

              ) else
 "1111"; --Not defined yet


 BRANCH<= '1' WHEN (
                    (instOP = "000100") -- B,BEQ
       ) else
 '0';

 REGWRITE <= '1' WHEN (
                    (instOP = "000000"  ) -- Special
```

```
            or    (instOP = "100011" ) -- LW
        ) else
    '0';

    REGDST <= '1' WHEN (
                  (instOP = "000000" and instFUNC = "100000") -- ADD
            or    (instOP = "000000" and instFUNC = "100010") -- SUB

        ) else
    '0';

    MEMWRITE <= '1' WHEN (
                  (instOP = "101011" ) -- SW
        ) else
    '0';

    MEMTOREG <= '1' WHEN (
                  (instOP = "100011" ) -- LW
        ) else
    '0';

    ALUSRC <= '1' WHEN (
                  (instOP = "100011" ) -- LW
            or    (instOP = "101011" ) -- SW
        ) else
    '0';


    END ARCHITECTURE SelectResult;
```

Appendix B: VHDL code for Forward Logic

```
--Forward Logic
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ForwardLogic IS
 PORT
 (
      EX_REGWRITE          : IN STD_LOGIC;
      MEM_REGWRITE : IN STD_LOGIC;
      EX_MEMTOREG          : IN STD_LOGIC;
      ID_MEMTOREG          : IN STD_LOGIC;
```

```
        ID_RS               : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        ID_RT               : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        EX_RD               : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        IF_RS               : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        IF_RT               : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        MEM_RD                  : IN STD_LOGIC_VECTOR (4 DOWNTO 0);

        FORWARDA        : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
        FORWARDB        : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
        DELAY               : OUT STD_LOGIC
 );
END ENTITY ForwardLogic;

ARCHITECTURE ForwardResult OF ForwardLogic IS
BEGIN
DELAY <= '1' WHEN (
            (ID_MEMTOREG = '1' and ((ID_RT = IF_RS) or (ID_RT = IF_RT))
        ))
        ELSE '0';

FORWARDA <= "10" WHEN (
            ( EX_REGWRITE = '1' and EX_MEMTOREG = '0'
and ID_RS = EX_RD)
        )
        ELSE      "01" WHEN (
            (MEM_REGWRITE = '1'
and ID_RS = MEM_RD)
        )
        ELSE      "00";

 FORWARDB <= "10" WHEN (
            ( EX_REGWRITE = '1' and EX_MEMTOREG = '0'
and ID_RT = EX_RD)
        )
        ELSE      "01" WHEN (
            (MEM_REGWRITE = '1'
  and ID_RT = MEM_RD)
        )
        ELSE      "00";

END ARCHITECTURE ForwardResult;
```