A Fully Pipelined 16-bit RISC Processor Featuring Hazard Resolution, Forwarding Logic, and ASIC Physical Implementation

Introduction

What we are going to do is a MIPS like 16-bits general purpose CPU core. As MIPS architecture is famous for its simplicity and easy to pipeline, we will spend some time to study the original MIPS instruction set and its architecture, and later on came up with our own 16-bit instruction set, design and implementation.

Our 16-bit CPU will be able to basic arithmetic including add, sub and multiplier and logic functions including shift, and, or, nor (not). We will test it use a binary coded program.

The basic goal of our design is to study in-depth computer architecture and learn to the design tools, experience how a modern integrated circuit is designed, implemented and tested.

Specifications

16-bit CPU
8 general register
ROM(instruction memory) &RAM (data memory) up to 32K
Frequency 200MHz
Voltage 1.1v
Standard cell area 8502 um^2
Allocation area 12996 um^2
Density 61.4%
Total internal power 4.463 mW
Total switching power 1.753 mW
Total leakage power 0.05 mW
Total power 6.267 mW

- **Instruction set**

MIPS Instruction has 3 types; the format of each type of instruction is shown below.

Type	-31- format (bits) -0-					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Fig 1 MIPS Instruction format

R, I, J type instructions refers to register operation, immediate number operation and jump (branch) accordingly. Each type can be easily identified due to the unique pattern of opcode. As follow this idea, we came up with our 16-bit instruction set as shown below.

		Opcode(15-11)	Rs(10-8)	Rt(7-5)	Rd(4-2)	Shamt(1-0)
R	ADD	00100				00
	SUB	00000				00
	AND	00101				00
	OR	00110				00
	XOR	00001				00
	SLL	00010				Amt(1-0)
	SRL	00011				Amt(1-0)
	MUL	00111				00
I	ADDi	01100			Immediate(4-0)	
	ANDi	01101			Immediate(4-0)	
	ORi	01110			Immediate(4-0)	
	LW	10100			Immediate(4-0)	
	SW	11100			Immediate(4-0)	
	BEG	11000			Immediate(4-0)	
J	J	100XX	Address (10-0)			
BU	BU	010XX				

Fig 2 16-bit instruction set

Look at this instruction set, the most significant bit is always 0 in R-type while it's always 1 in I-type. When the first two bits are both 1, it refers to J-type. In this way, we make it easy to identify and has some redundancy in I and J type.

We have 2 bits for shift amount that means we can shift 1, 2, or 3 bits at a time. The width for immediate is 5.

Single cycle design

- **Control unit design**

Control unit is very important to our MIPS circuit. It is related to the functionality of our MIPS and also one fatal factor to determine whether our circuit work right or not. We use the first three bits of operation code and control units' truth table to design our control unit structure.

		Opcode(15-11)
R	ADD	00100
	SUB	00000
	AND	00101
	OR	00110
	XOR	00001
	SLL	00010
	SRL	00011
	MUL	00111
I	ADDi	01100
	ANDi	01101
	ORi	01110
	LW	10100
	SW	11100
	BEG	11000
J	J	100XX
	halt	010XX

	R 00x	iA 011	Lw 101	Sw 111	BEG 110	J 100	Halt 010
RegDst	1	0	0	0	0	0	0
Branch	0	0	0	0	1	0	0
Jump	0	0	0	0	0	1	0
MemRead	0	0	1	0	0	0	0
MemtoReg	1	1	0	1	1	X	0
MemWrite	0	0	0	1	0	0	0
RegWrite	1	1	1	0	0	0	0
ALUSrc	0	1	1	1	0	x	0
Regread	1	1	1	1	1	0	0

5

Fig 3 control unit truth table

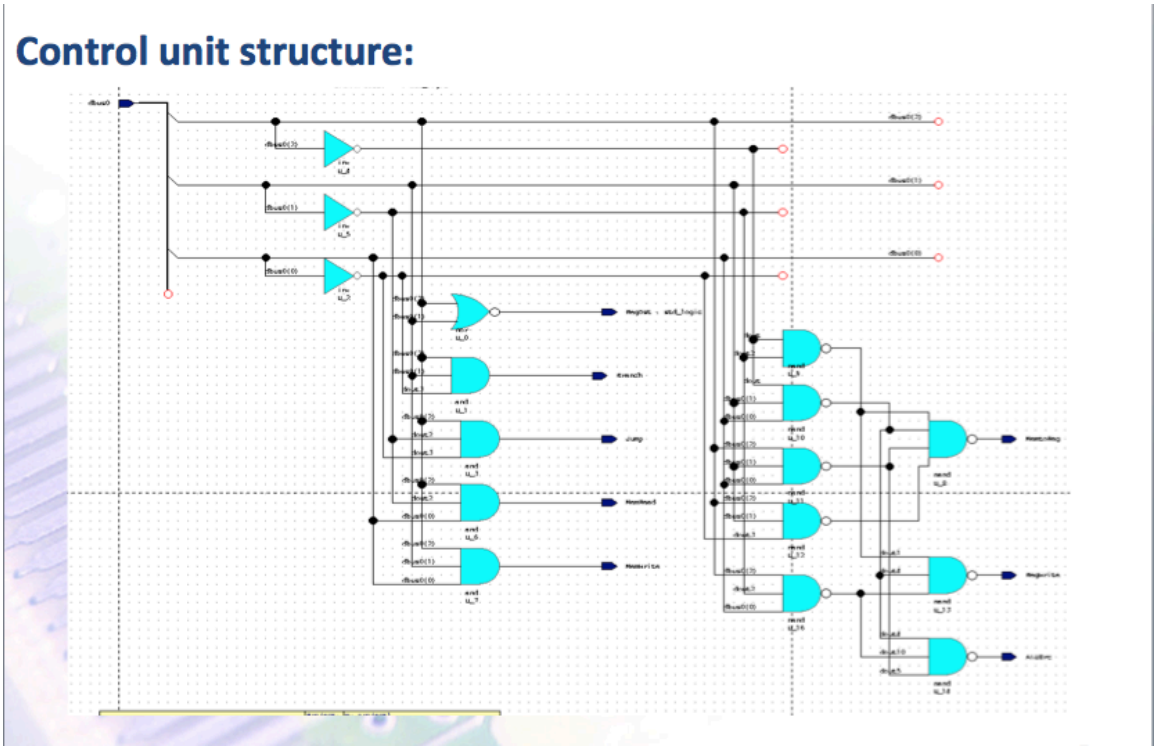


Fig 4 control unit structure

- **ALU design**

ALU is the key part to do the operation, in our ALU, we have to work the functionality such as addition, subtraction, Multiplication, sll, srl, and, or, xor logic. We also use the subtractor in the ALU to determine whether the branch instruction should be taken or not. We use the opcode's last three bits to be the function selective input signal.

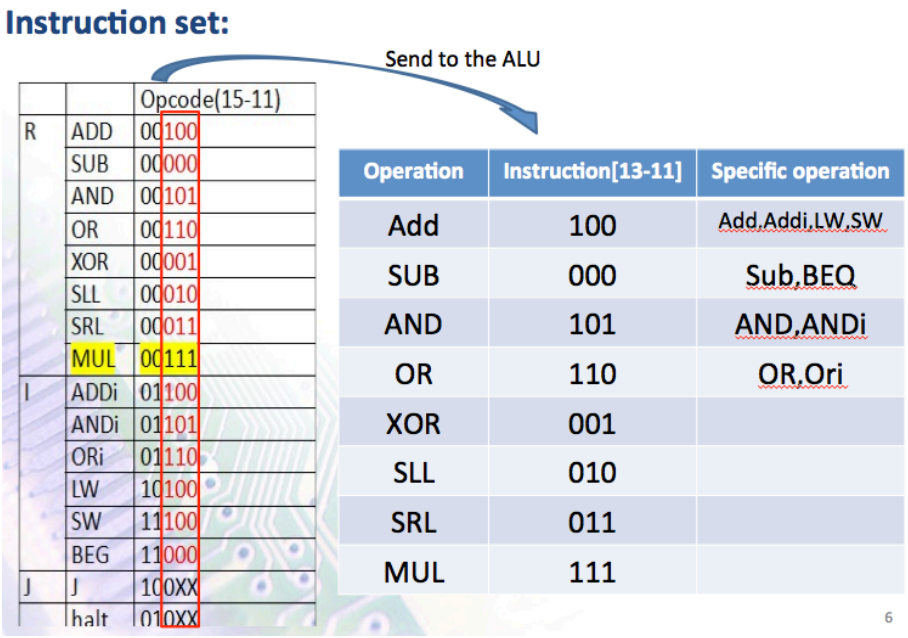


Fig 5

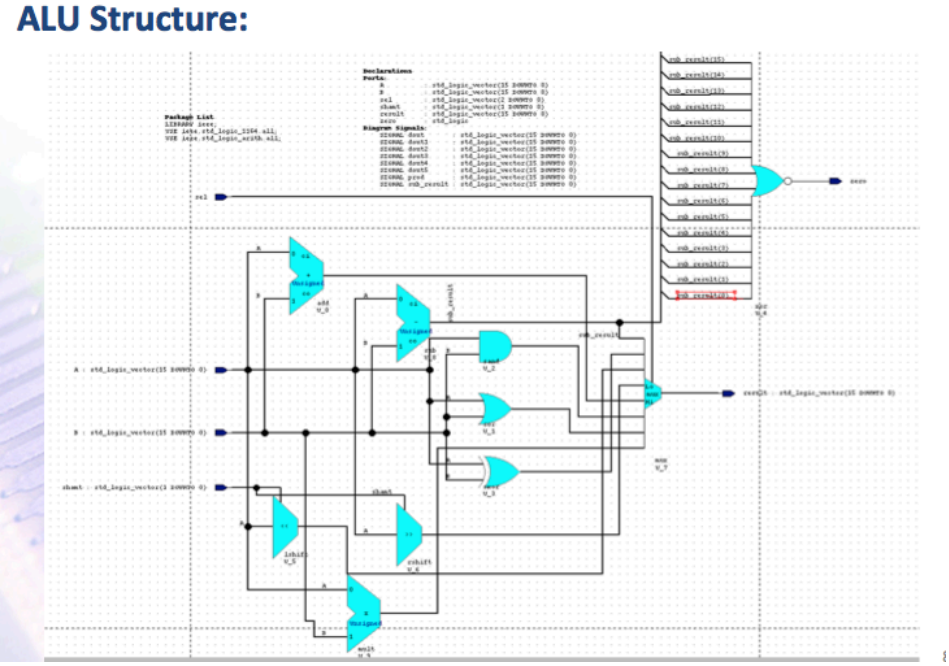


Fig 6 ALU structure

- **Data path of single cycle**

After finish designing the ALU and control unit, we still should add many other components. Here is the single cycle data path diagram for our design. We will build this first, test it and make it pipelined later.

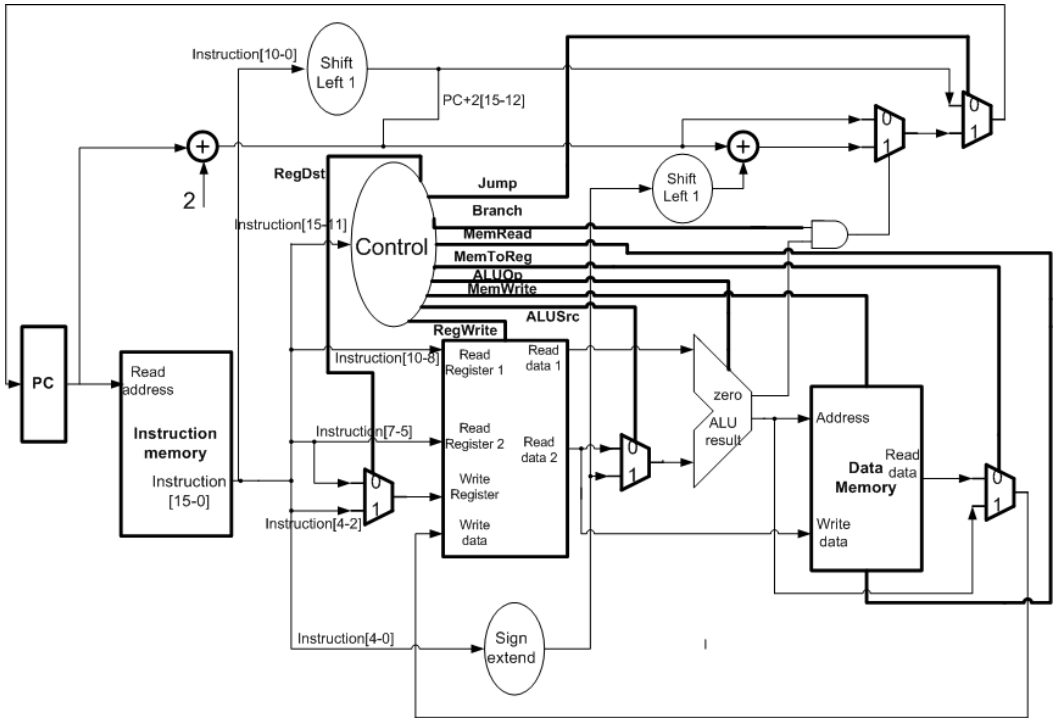
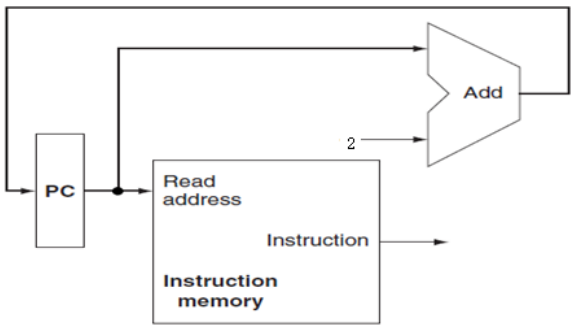


Fig 7 the single cycle data path diagram

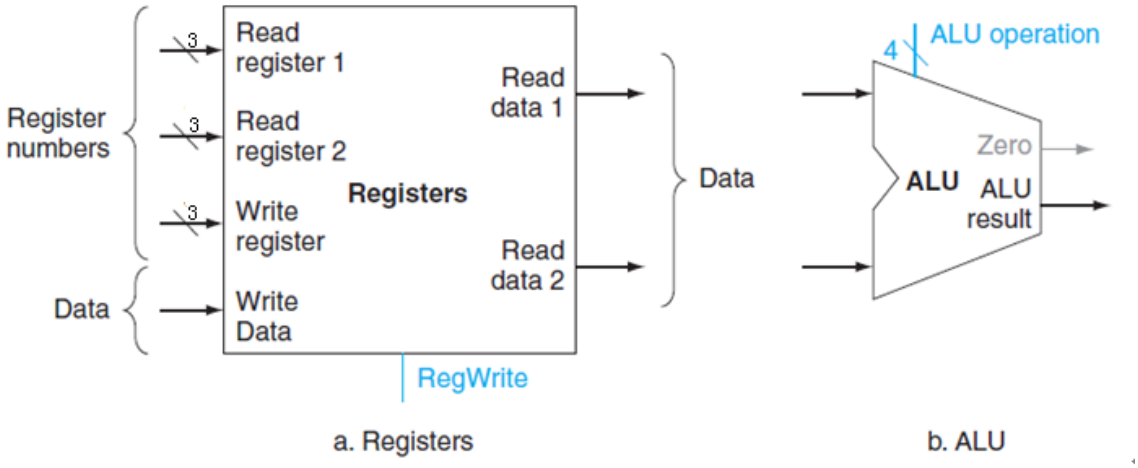
The opcode field directly connected to the control unit to determine the operation. We cut the separate ALU control in original MIPS design because we don't have the function field in our instruction set and all the control logic is determined by one control unit.

- A portion of the datapath used for fetching instructions and incrementingthe program counter



The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that will be written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always perform an add of its two 32-bit inputs and place the result on its output.

- The two elements needed to implement R-format ALU operations are the register file and the ALU.



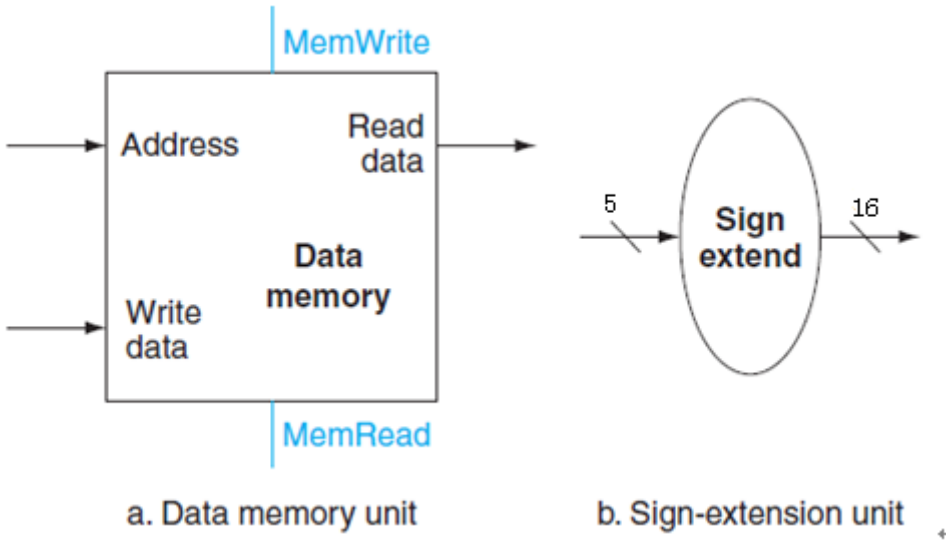
The register file contains all the registers and has two read ports and one write port.

The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal.

Remember that writes are edge-triggered, so that all the write inputs \must be valid at the clock edge. Since writes to the register file are edge triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide.

The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in Appendix B. We will use the Zero detection output of the ALU shortly to implement branches.

- the data memory unit and the sign extension unit are needed to implement loads and stores



a. Data memory unit b. Sign-extension unit

The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. Assuming the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes.

Based on the design and information above we build our own MIPS single cycle.

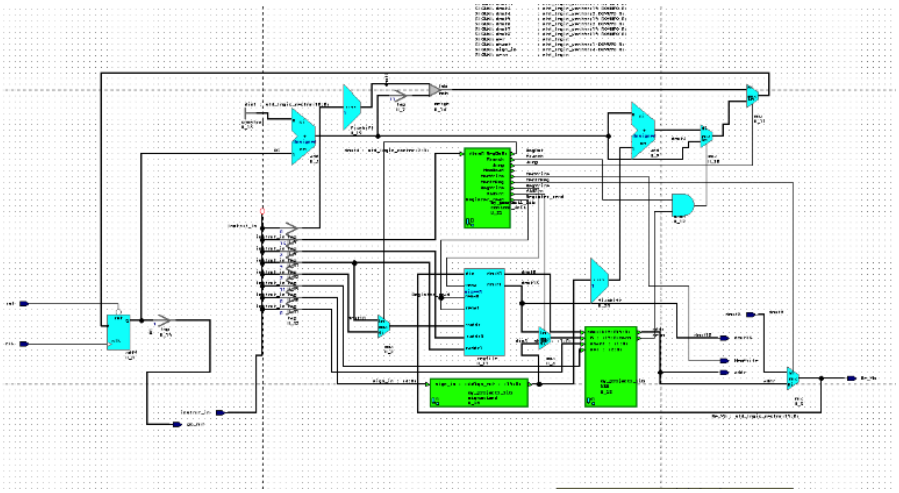


Fig 8 single cycle we really make

Pipeline Design

- **Data path and stage of pipeline design**

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. It is a way to enhance the performance. Therefore, here we develop the data path and operation of MIPS into 5 steps based on the single cycle we have designed including 1.instruction fetch, 2.instruction decode and register file read, 3.execution or address calculation, 4.data memory access, 5.write back as figure2

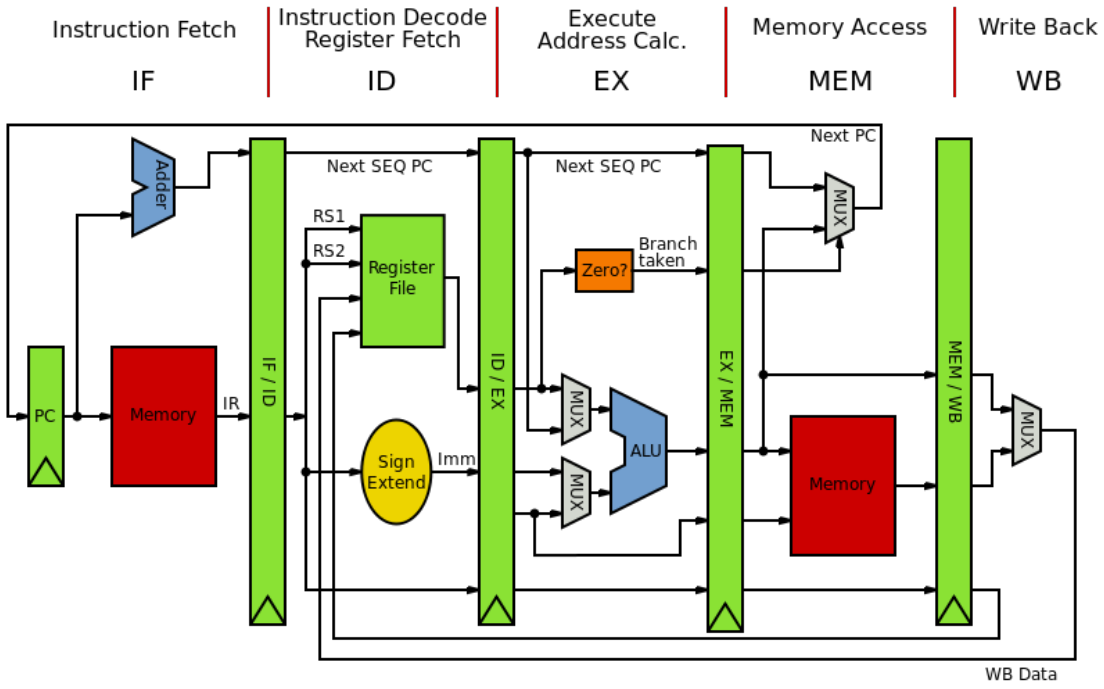


Fig 9 stages of pipeline

Take the instruction lw as example, the executive process in each pipe would be

1. Instruction fetch: the instruction is read from memory using the address in the PC and then placed in the IF/ID register. The PC address is increment by 2 and then write back into the PC to be ready for the next cycle. This increment address is also stored in the IF/ID register in case it is need latter.
2. Instructions decode and register file read: the IF/ID register supplying the immediate field, which is extended to 16-bit. The register numbers to read the two registers. All three values are store in ID/EX pipeline register with the increment address.
3. Execute or address calculation: the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register

4. Memory access: the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

5. Write back: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure

- **Design a basic pipeline structure**

➢ Based on the data path and stages in Fig 9, we can design our own basic pipeline structure based on our single cycle. And something we have to notice:

1. We have to understand the functionality of different stages and put the corresponding components in the right stages.
2. As the instruction has been pipelined, the corresponding control signal should also be pipelined at the right timing.
3. We have to fully understand what each instruction will do in each stage. Details have to be focused on. As we will encounter many problems during the simulation, understanding details is the fastest way to solve them.

➢ What major components in each stage

1. IF stage: PC register, ALU (used to calculate the next pc address) , ROM(which is used to store the instructions) and a mux which is used to select the address from the next pc, branch or jump address.

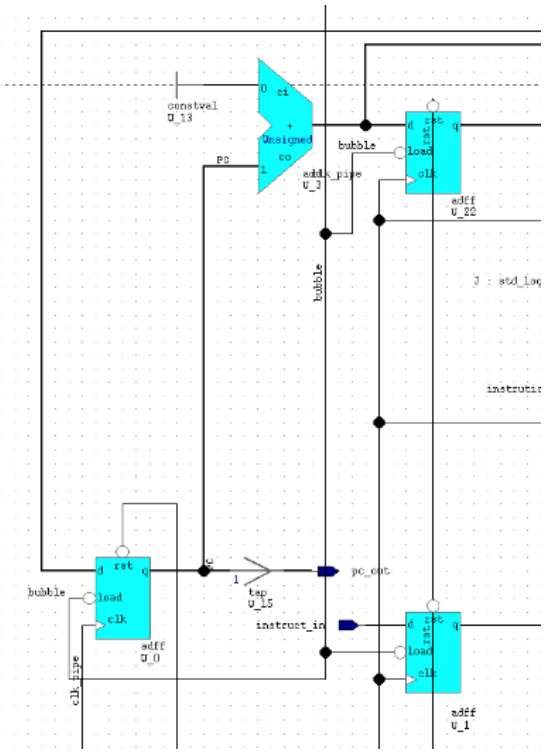


Fig 10 IF stage

2. ID stage: control unit, register file, sign extended In the execution

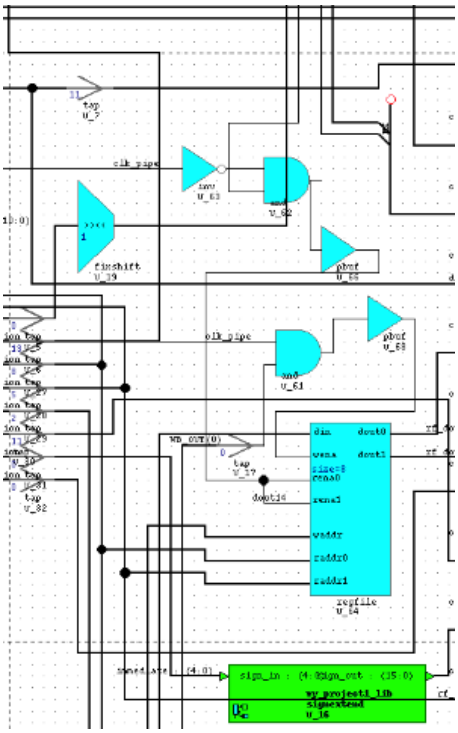


Fig 11 ID stage

3. EX stage: ALU

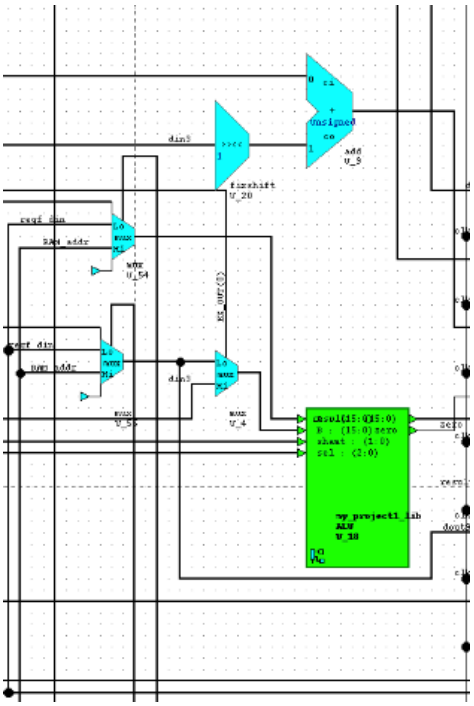


Fig 12 EX stage

4. MEM stage: ram (which is used to store the data)

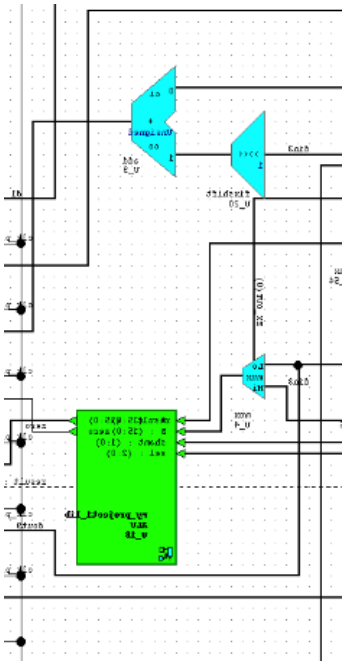


Fig 13 MEM stage

5. WB stage: mux, which is used to select the write back data between the result from ALU and the result from the memory

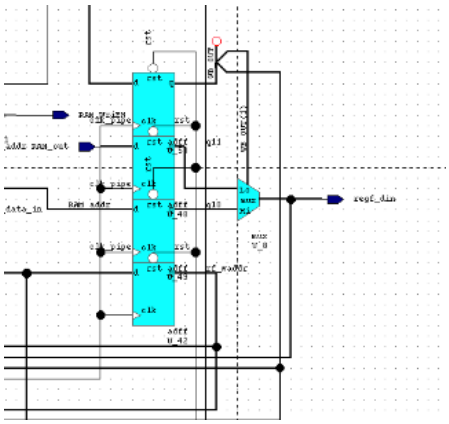


Fig 14 WB stage

- Things we have to concern about

1. The address calculation of the branch and jump instruction should be done in the IF stage. However, we cannot use them until the zero signal of branch instruction occur. Thus, the branch address and jump address should also be pipeline to the MEM stage.
2. This pipeline structure cannot solve any data dependence, thus, to design the test program, we have to concern about the data hazards and add bubble when we need.

3. The write in address of the register file should also be pipelined to the WB stage, as destination register of R-type and I-type instruction is different then we need a mux to select between them.

- **Design a optimal pipeline structure**

As I have mentioned just now, the basic pipeline structure can not solve any data dependence, thus, to make a better pipeline design, we wish make it work like a single cycle. To achieve this goal some optimal methods have to be made.

➢ Data dependence

Data dependence is a very important problem that we have to solve.
For example:
Add $4,$1,$2;
Add $5,$4,$4;
In these two instructions, we can see that the second instruction needs the first instruction's result to do its operation. However, we know that the result of the first instruction cannot be written into the register file until the WB stage. Thus, the second instruction can never get the right result without two bubbles between them. We call this situation as data dependence. Totally, there are two kinds of data dependence. The first one is the data dependence between the instructions use the ALU to do calculation and then store their result back to the register file. To solve this data dependence, we just need to write the first instruction's result back to the EX stage when it is in the MEM stage and WB stage. The forwarding circuit and cope with such situation.

➢ Design forwarding circuit

Before we design the forwarding circuit, we have to figure out two questions.

1. How to determine a data dependence situation?

As we all know, every instruction has its own control signal, thus, we need to use the control signal from different stages and compare their source register and destination register to determine whether the data dependence situation exist or not.

2. How many situation we have to consider?

In this forwarding circuit, we have to consider the data dependence between two instructions and three instructions.
For example:

ADD $1, $2, $2;	ADD $1, $2, $2;
Bubble ;	ADD $2, $1, $1;
ADD $2, $1, $1;	

In the first example we can see that, the data dependence exit between the first and the third instruction. Related to the structure, it happens at EX and WB stage.
In the second example we can see that, the data dependence exit between the first and the second instruction. Related to the structure, it happens at EX and MEM stage.

3. What components that we need?
   If we want to make a forwarding circuit, we need to build a circuit which can use the input control signal to determine whether there is data dependence and which kind of data dependence happens. In addition, we need to add two mux in front of the input of the ALU, and use the output of the forwarding circuit to determine which input we are going to use.

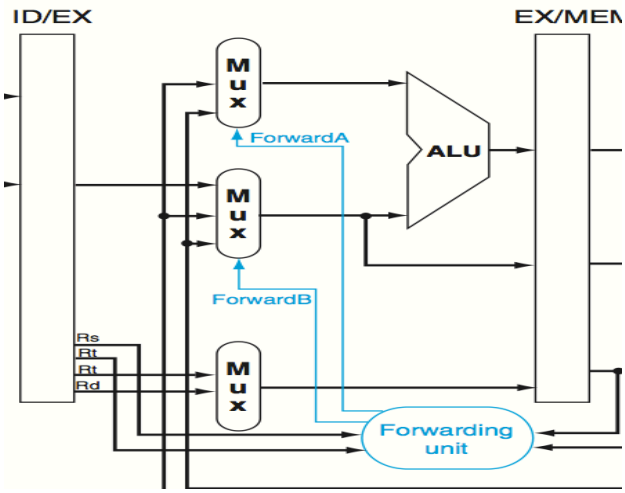


Fig 15 forwarding circuit structure

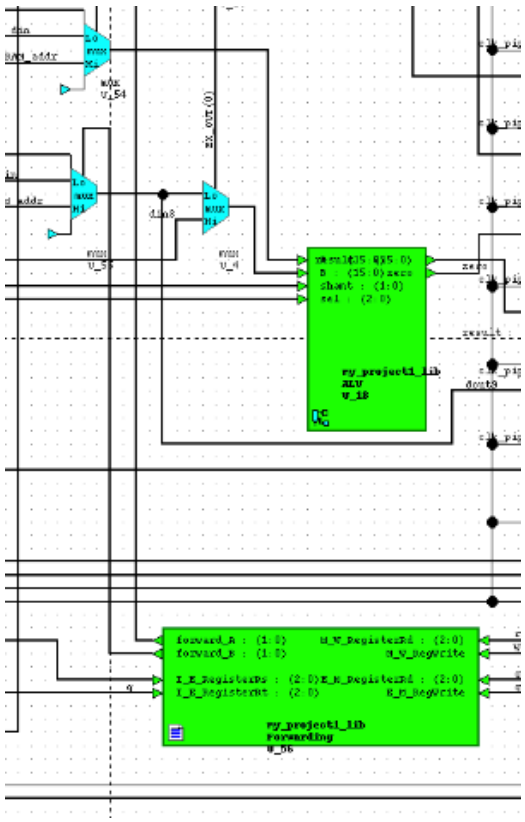


Fig 16 forwarding circuit we actually make

- Another data dependence

Although forwarding circuit have been designed, the second kind of data dependence still can not be solve.As we all know that the result of the load instruction will appear in the WB stage. Thus, if data dependence happen concerning the load instruction, it cannot be solved by the forwarding circuit we just mentioned. For example:

Load $1,$2,$3;

ADD $4,$1,$1;

In this example, we can see that, data dependence exist between the load instruction and the add instruction. The result of the load instruction will only appear at the WB stage. However, when the load instruction at the WB stage the add instruction will be pipelined to the MEM stage. In this case, the forwarding circuit we just mentioned cannot solve this problem.

- Design hazard detection

Instead of using a bubble between these two instructions, we try to make another circuit called "hazard detection". The functionality of this component is to detect this kind of data dependence, and then control the registers in ID and IF stages to stop them getting new data for one clock cycle. In this case, the load instruction in the MEM stage will be copied, thus, the MEM and WB stages will contain the same load instruction. However what we need is just the load instruction in the WB stage, thus, we have to wipe up the load instruction's influence in the MEM stage. The best way to achieve this goal is to change its corresponding control. To make it easy, we just try to set them to be zero. Before design the hazard detection we have to answer the following questions.

1. How to determine such kind of data dependence?

Like the forwarding circuit, we also use the control signal and compare these two instructions' destination register and source register to determine this kind of data dependence. As the "memread" signal will be true only when the load instruction happen, thus, we can use it to identify this instruction.

2. How to stop the registers in the corresponding stage?

As the register has its enable signal, thus, we can use it to stop it when we need.

3. How to make the control signal zero?

To make the control signal to be zero, we need a mux to select the input of the control unit. If this data dependence happens, then the mux should choose the data which make the output of the control unit to be zero. If not, then the mux should choose the current instruction's data to be the input of the control unit.

4. What components we need?

To accomplish this goal, we should design a component used to determine whether such situation happens or not. And we need it to output a signal to control the mux and registers in the stages.

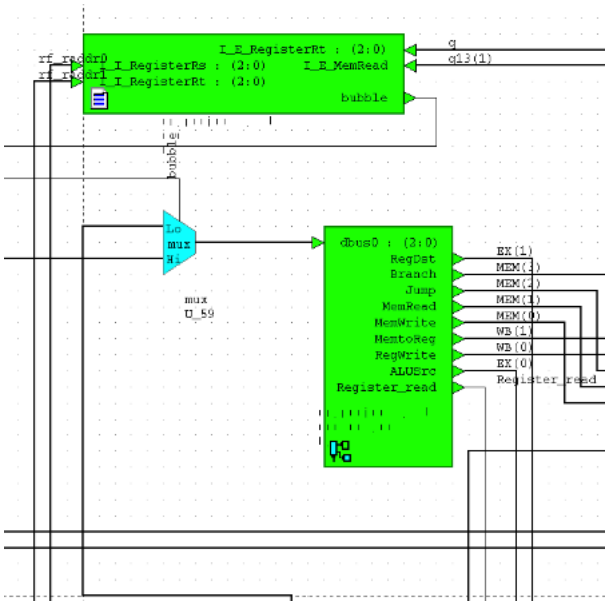


Fig 17 hazard detection we make

Based on the information above, eventually we design our pipeline circuit.

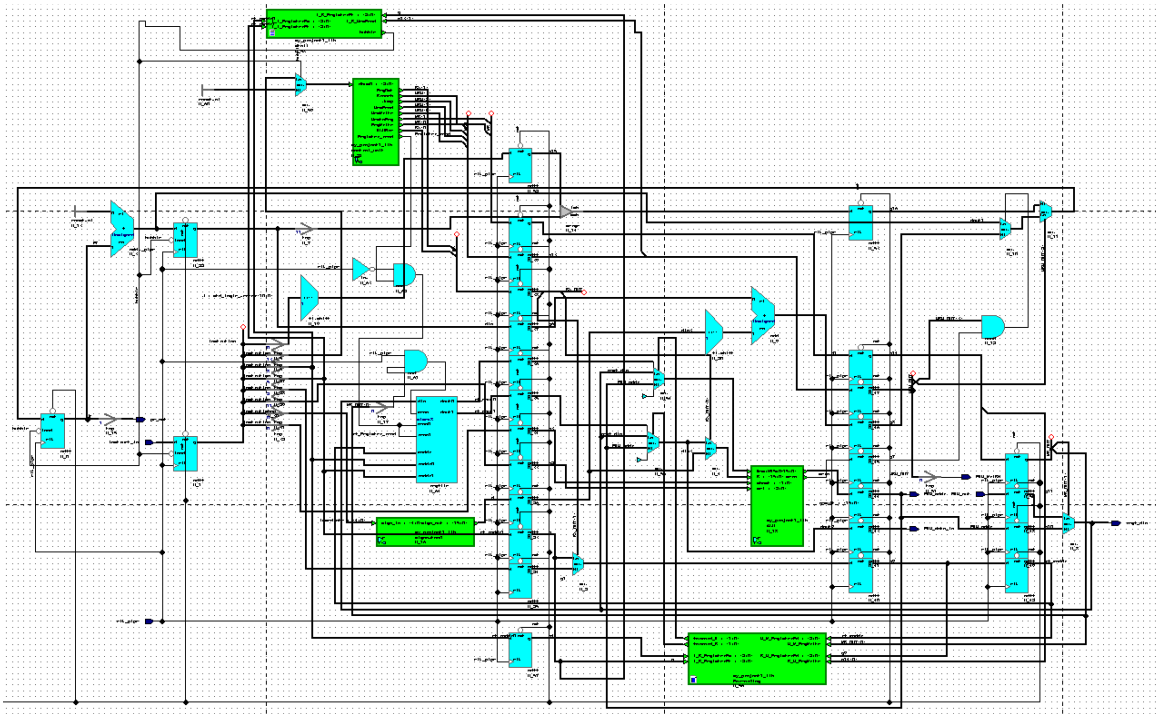


Fig 18 our pipeline structure

- **Other optimal method**

➢ Move to IF stage

As we have mentioned, we finish the branch and jump instruction calculation in the EX stag, however whether we are going to take the branch instruction or not will be determined at the EX stage. Thus, we need to put at least two bubble after the branch and jump instruction. To avoid using too many bubbles and make it works more like single

cycle. We can move these steps to the second stage. In this case, we just need to add one bubble after the branch and jump instruction. Before moving , we have to answer the question following:

1. How to finish all the calculation in the second stage?

Just move all relevant components to the second stages and connect them in the right logic.

2. How to decide whether we need to take the branch instruction or not in the second stage?

Instead of using the ALU in the EX stage, we can design a circuit just use 16 2-input xor gates and one 16-input nor gates to determine whether the input value is equal or not.

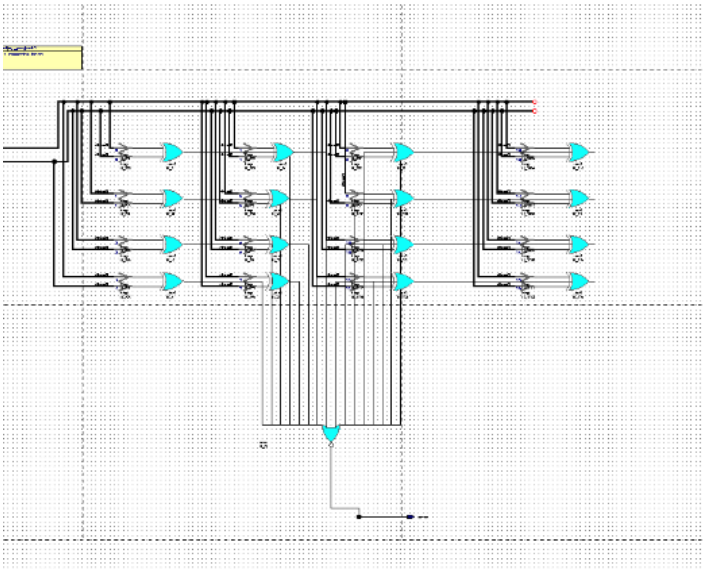


Fig 19 zero determine circuit

- Extra forwarding circuit and hazard detection circuit

The first forwarding circuit we build in the EX stage can solve the branch instruction's data dependence. However, if we move the branch and jump logic to the ID stage, then the first forwarding circuit cannot solve the branch instruction data dependence any more. In this case, to make our circuit work like a single cycle we need to design another forwarding and hazard detection to solve the branch instruction data dependence. Before making it we have to answer the following question.

1. What situation we have to be considered?

To solve this problem, we have to consider the following situation:The data dependence between the branch instruction and the instruction will go through the ALU component. For example:

Add $1, $2, $2

Branch $1, $1,xx

In this situation, the branch instruction has to wait for the first instruction's result comes out for one clock cycle.
The data dependence between the branch instruction and the load instruction:
For example:
Load $1, $2, xx
Branch $1, $1, xx
In this situation, the branch instruction has to wait for the result of the load instruction comes out for two clock cycle.

2. How to design this forwarding circuit?

The method to design this forwarding circuit is the same as the first forwarding circuit. We just use the control signal from different stages to identify which instruction is going through that particular stage. And then we compare the source register of the branch instruction in the ID stage and the destination register in the MEM and WB stage to identify whether this kind of data dependence exist or not. The difference of the forwarding circuit and the first forwarding circuit is that this forwarding circuit is more dedicated. It just solves the branch instruction's data dependence.

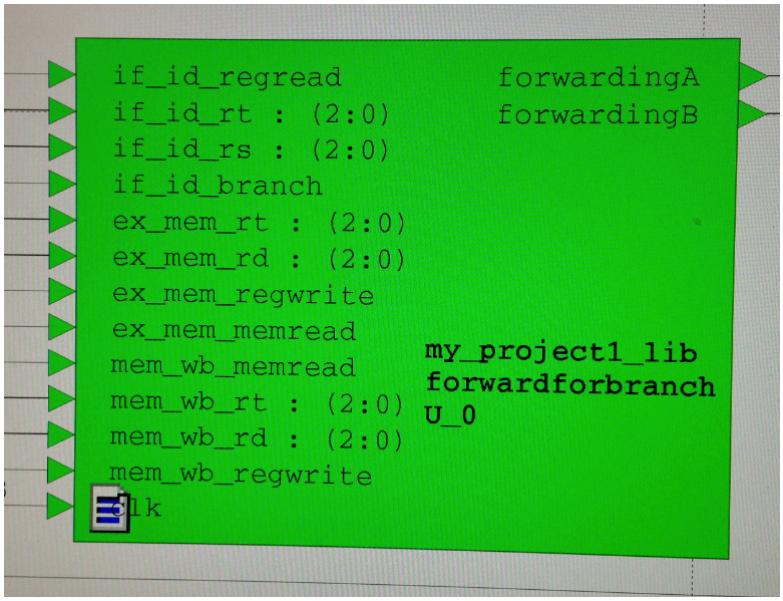


Fig 20 Extra forwarding circuit

In this picture, we can see that, the left side is the input ports and the right side is the output ports. The output is used to control the select signal of the input mux of the zero signal compare component.

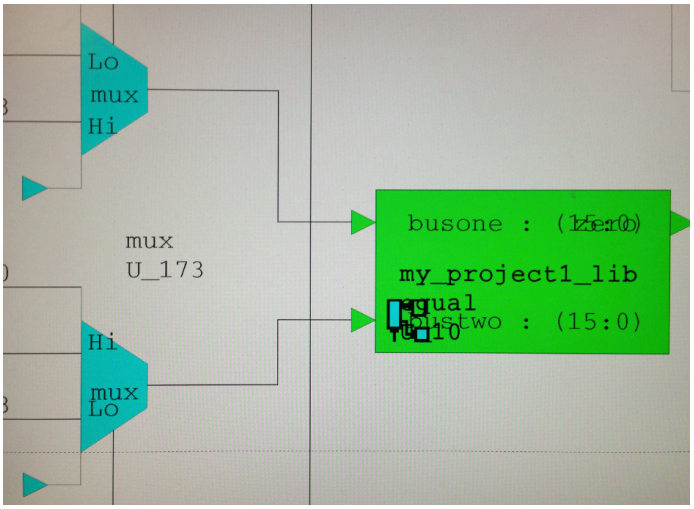


Fig 21 mux of the zero determine circuit

In this picture, the select signal comes from the output Fig 19.

- Extra hazard detection circuit

As I have mentioned, we have to solve two kinds of situation here, the first one we have to wait for one clock cycle and the second one we have to wait for two clock cycles. Thus, the hazard detection circuit should also have the ability to solve this kind of situation. The best way to do that is using the finite state machine. In this case, I just use VHDL code to write Moore state machine without reset signal because I just test their behavior and ignore the synthesize part.

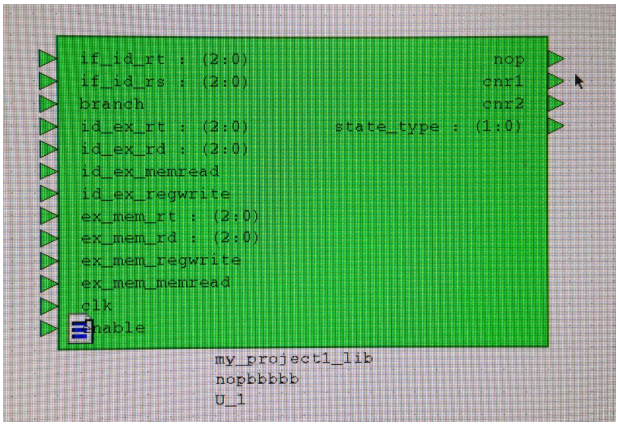


Fig 22 extra hazard detection

The ports on the left side are used to connect the input signal used for operation. The ports on the right side are the output ports used to control the other components. And I will explain the functionality of the output ports here.

Nop : connected to the enable of the stage registers, which is used to control whether they should get a new data or not.

Cnr1: used to control the selectivity of mux between the zero and the original control signal at the EX stage.
Cnr2: used to control the selectivity of mux between the zero and the original control signal at the MEM stage.
State_type: used for the communication of different component when we get the branch-prediction into the circuit.
And then I will explain the state situation of the state-machine and the output situation:

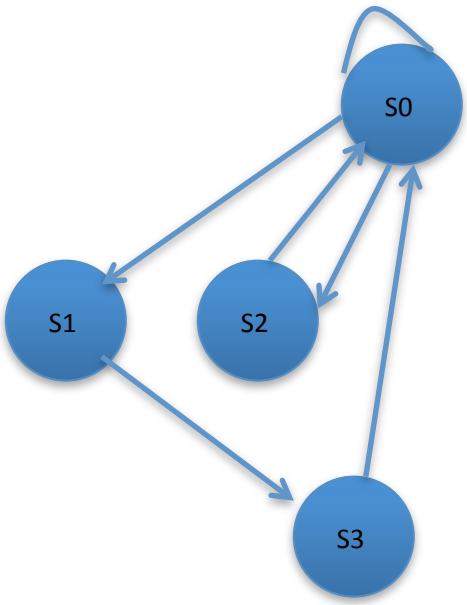


Fig 23 state changing

This picture shows how the state changes in the hazard detection circuit. At the beginning, the component change their state base on the condition, after that, statement machine will output data base on their states. All these work done in just one clock cycle. And I will explain how the state machine works in each state.

S0:
This state is the beginning state, in this state it will change the state based on the condition. If there is not data dependence, then it will change it state to S0.
If the data dependence happens and we should wait for two clock cycle then we should change the state to s1.
If the data dependence happens and we should wait for one clock cycle then we should change the state to s2.
S1:
Just change the state to s3 without at the rising edge of the clock cycle.
S2:
Just back to state S0 at the rising edge of the clock cycle.

S3:
Just back to state s0 at the rising edge of the clock cycle.
What is going to output in each state?
S0:
Nop (control signal of the state register) : true
Cnr1: zero;
Cnr2: zero;
S1:
Nop :false;
Cnr1:zero;
Cnr2:one;
S2:
Nop: false:
Cnr1:zero;
Cnr2:one;
S3:
Nop: flase;
Cnr1:zero;
Cnr2:one;

- How to deal with the conflicts between two hazard detection components?

As we know that, now we have two hazard detection components in the same circuit. The first hazard detection circuit just used for the load instruction and the second one used for the branch instruction. The first hazard detection component just control the first and second stages' registers and the second hazard detection component will control the first, second and third stages' registers. Thus, when multiple data dependence situation happens, some conflict situation will happen. In this case, we have to come up with a method to solve the problem. In order to solve the problem, I have to make another circuit.

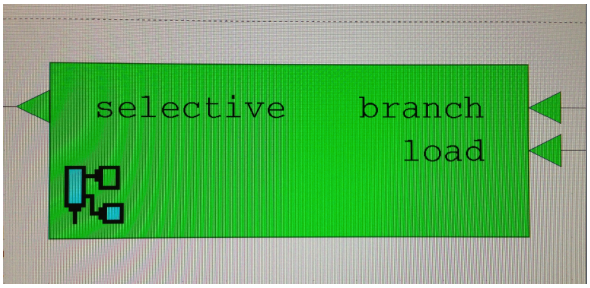


Fig 24 the selective components

The right side is the input side and the left side is the output side. We can just use inverter and AND gate to solve this problem.

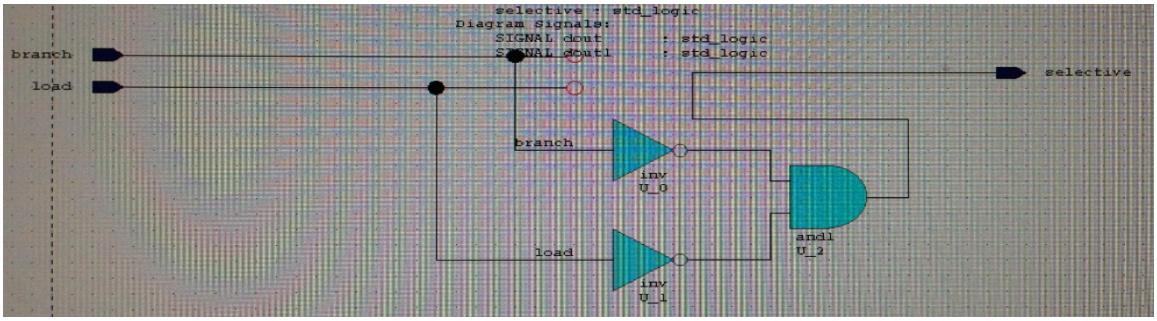


Fig 25 logic of the selective components

This is the inside logic of this component. This component should be used in the enable signal conflict of the stage register and the conflict of the mux of control signal.

- Branch prediction

Before we make a branch prediction, we should understand the following question:

1. Why we need a branch prediction?

As we have hazard detection and forwarding circuit, most of the data hazard can be solved. However, we do not know whether the branch should be taken or not before the zero signals occur. Thus, we still have to put at least one bubble after the branch instruction. In this case, the MIPS cannot be operated at full speed. So we propose the branch prediction component to solve this problem. Branch instruction is related to the loop in our programming and we always use loop in our program. So it is important for us to solve this problem and improve the performance of the MIPS chip.

2. What branch prediction does?

The branch prediction is a component that works for the branch instruction. If have two states, the first one is predict the branch instruction will be taken and the second one is predict the branch will not be taken. If it is in the first state, then this component will output control signal to the stage register to make it halt. If it is in the second state, then this component will do nothing until the zero signal comes. The component changes its state based on the zero signal, as it is the only signal can tell itself whether the prediction is right or not.

3. What is the logic of branch prediction in details?

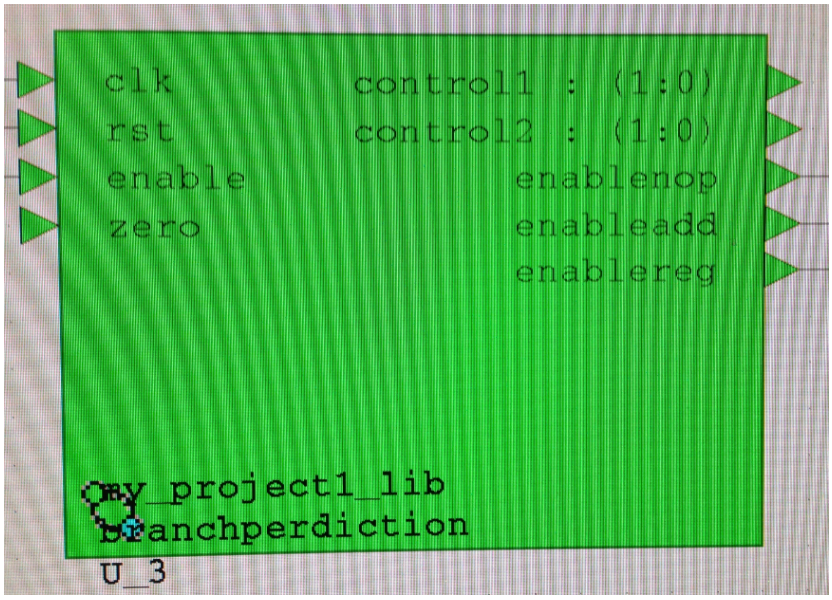


Fig 26 branch prediction

In this picture, we can see that the left side is input ports and the right side is output ports. The enable signal is connected to the branch signal of control unit. When branch instruction occur then the input of enable is true thus the branch prediction will work at the rising edge of the clock cycle.
Branch prediction is a finite state machine. To make is simple, I just use two bits finite state-machine. It means that we have four states totally and all these states can be identified by two-bits data. And the following picture will illustrate how the state changes

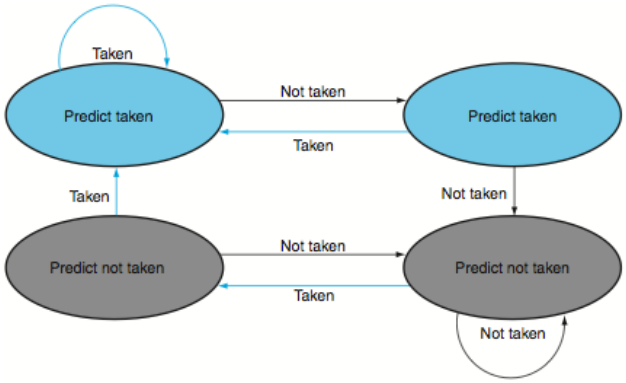


Fig 27 state changing

After that we have to understand what the state-machine going to do in each state. Before explaining the complicated logic, we have to understand what it is going to do and what components it is going to control.
If we predict the branch instruction is untaken, firstly, the branch prediction will do nothing; secondly, we have to use one register to store the branch adderss when we need

it. Thirdly, if the prediction is wrong, then the BP component should release the unexcept instruction. And use the branch address which stored in the register, jump back to what it should be

If we predict the branch instruction is taken, firstly, we need a register to store the next pc address of this branch instruction, when the prediction is wrong we can jump back to what it should be. Secondly, we can have to halt the circuit for one clock cycle wait for the branch address comes out. Thirdly, when the branch address comes out the BP circuit should affect the mux of the PC to make it select the branch address. Fourthly, the BP should wait for the zero signal to determine whether the prediction is right or not. Fifthly, if the prediction is wrong then the BP circuit should release the unexpected instruction and use the address stored in the register to jump back to the address it should be.

Something has to be concerned:

1. Another zero component:

As the BP component change their state based on the zero signal, however, if data dependence happens, we have to wait for one clock cycle or two clock cycles to get the zero result. This circuit is a very powerful one, it combines many different functionality together including forwarding circuit, hazard and zero compare circuit. This circuit should also understand when, where to the get the data in and understand when and where to output their value. So the timing issue is very import here.

2. One clock cycle or two clock cycle:

As the branch instruction have several data dependence situation, thus, the circuit I just mentioned above should have the ability to deal with this problem. And the branch prediction circuit also need to understand the timing, thus, because if it just have to wait one clock cycle then it just need to nop one unexpected instruction, if it have to wait two clock cycle , then it have to nop two unexpected instruction.

4. Conclusion of the branch prediction:

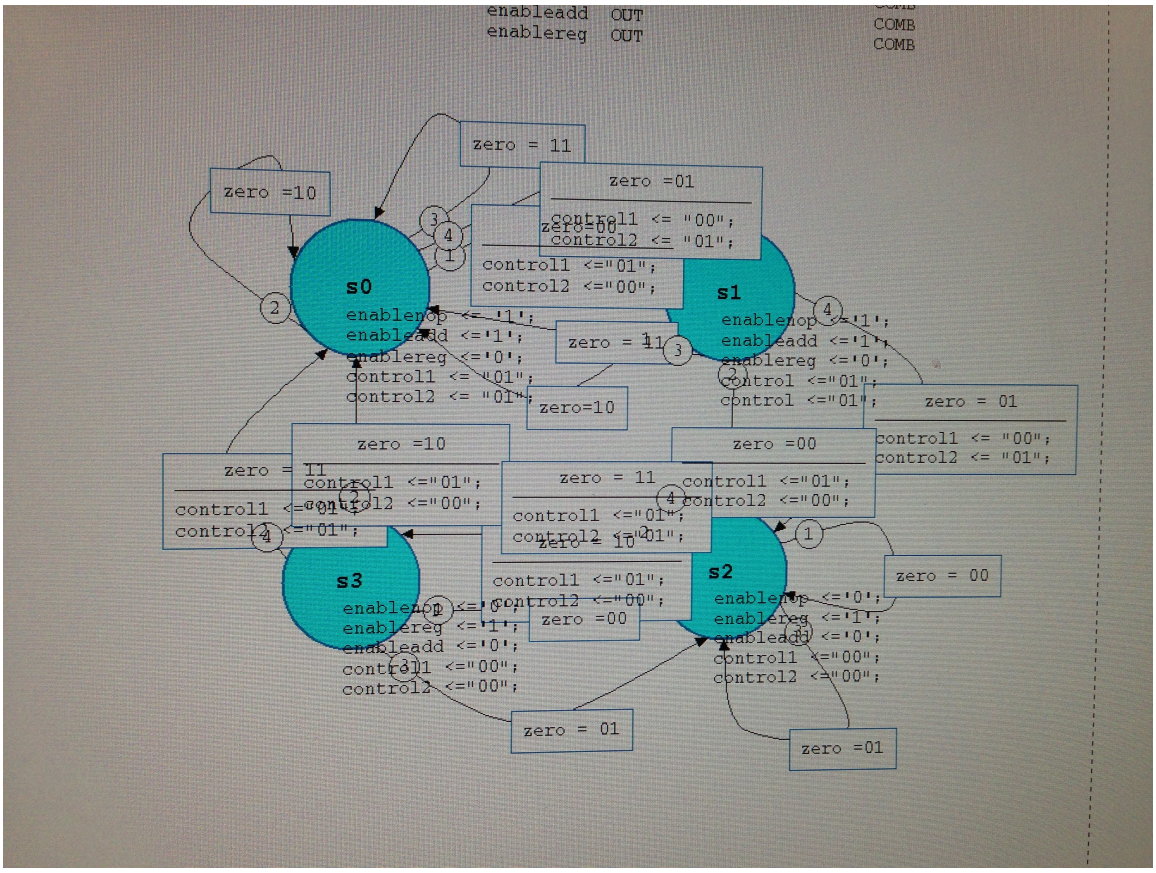


Fig 28 inside logic of branch prediction

The picture above is the branch prediction logic I try to make. It is complicate and not finished. In my design, to satisfy the need of the logic, the finite state machine is both Moore and Mealy type.

I cannot make the BP component eventually, as it is too complicated. All the information I offered in this report is what I was thinking and what I have got while I am trying to design this component. I would like to share with people who want to know the details of this part.

- **Synthesize**
- **Synthesize and Place&Route**

To do synthesize and place& route the whole design, we chose to do it separately, beginning with small blocks and the single cycle design. After that, synthesize and test the pipelined design. Finally, we do place and route to the pipeline design.

- **Single cycle**

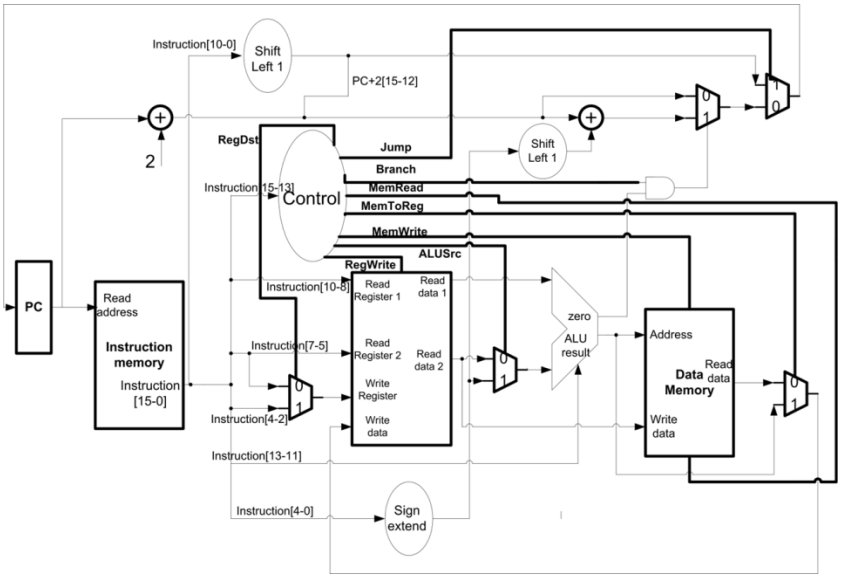


Figure 29 single cycle design

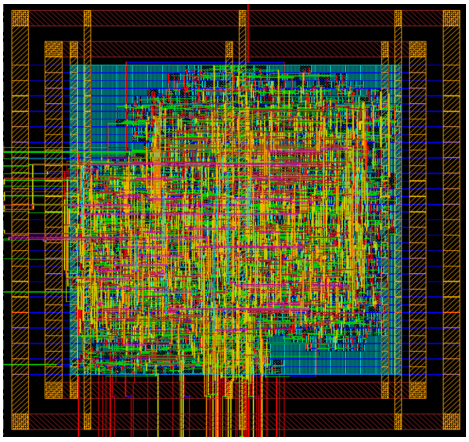


Figure 10 single cycle layout

After completing synthesize and place & route, we were facing a problem at that time. How to test the circuit in different level? Our solution is to use the test bench. Things that don't need to be synthesized in the design make up the test bench. Insert the synthesized

block together with the sdf file (the time delay file) to the HDL designer as a module. Finally we got the test result in Modalism.

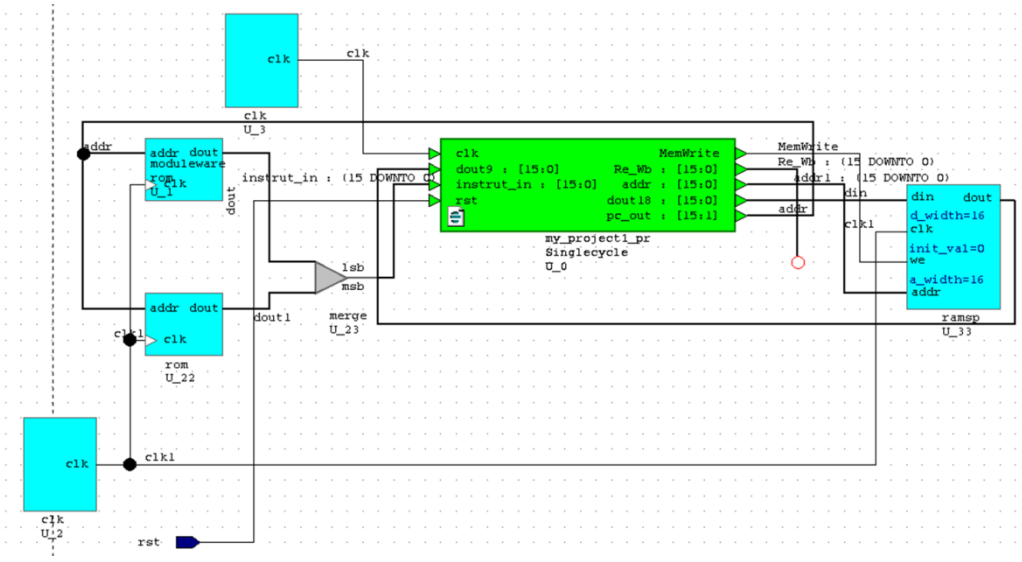


Figure 31 single cycle test bench

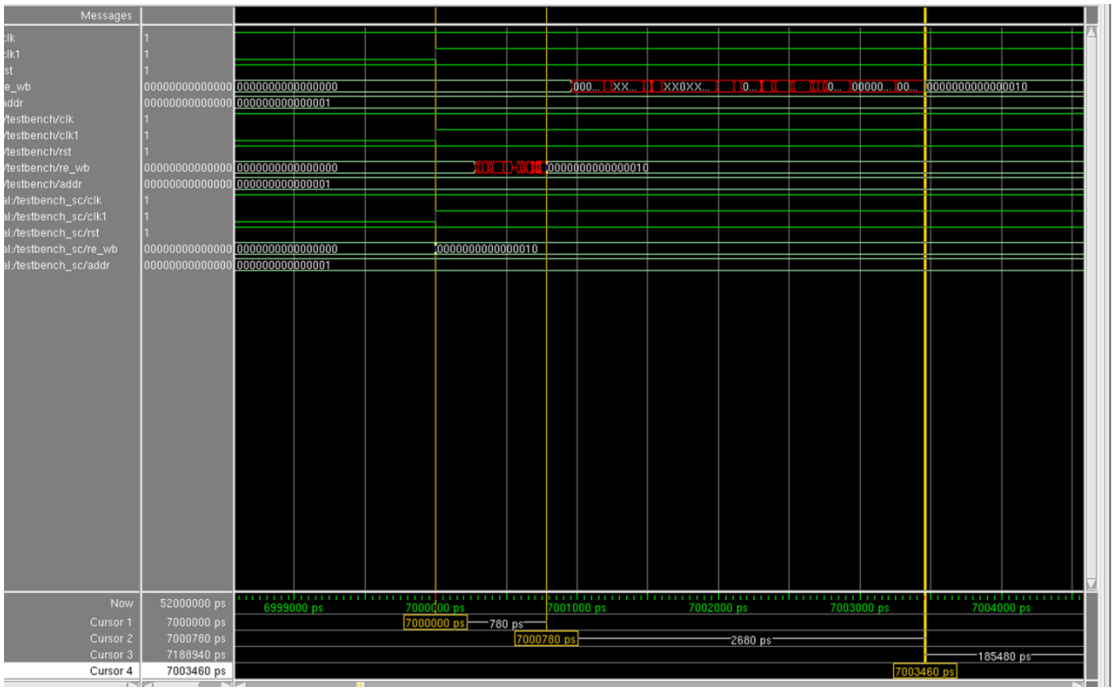


Figure 32 test result of the single cycle design

As we can see from the waveform, the waveform after synthesize and place & route has time delay compared to the behavior simulation result. The red ones are the transaction procedure.

1. Pipelined design

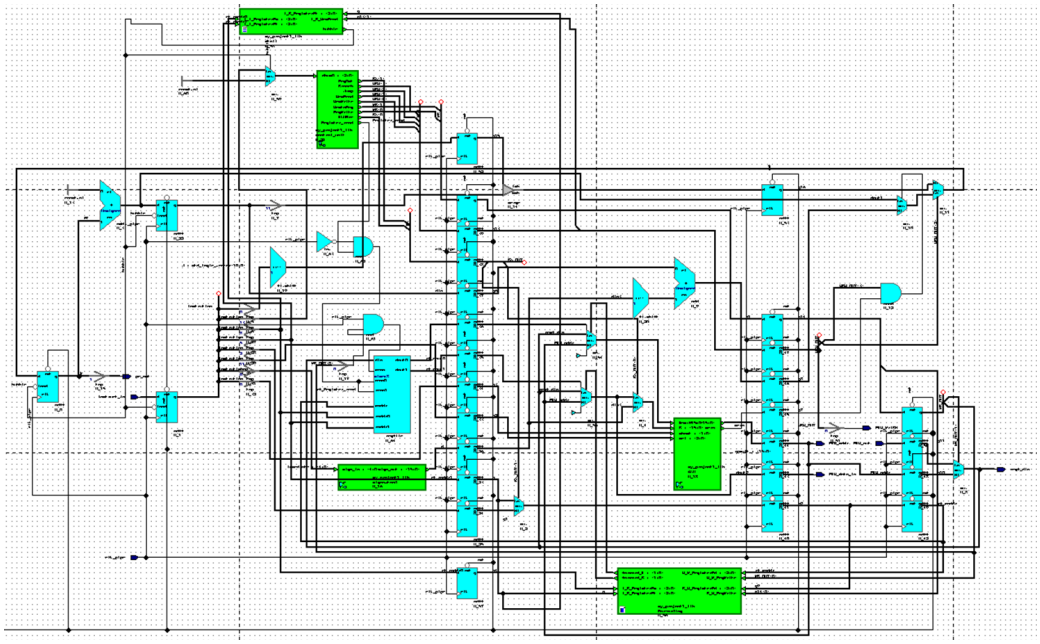


Figure 33 pipeline design

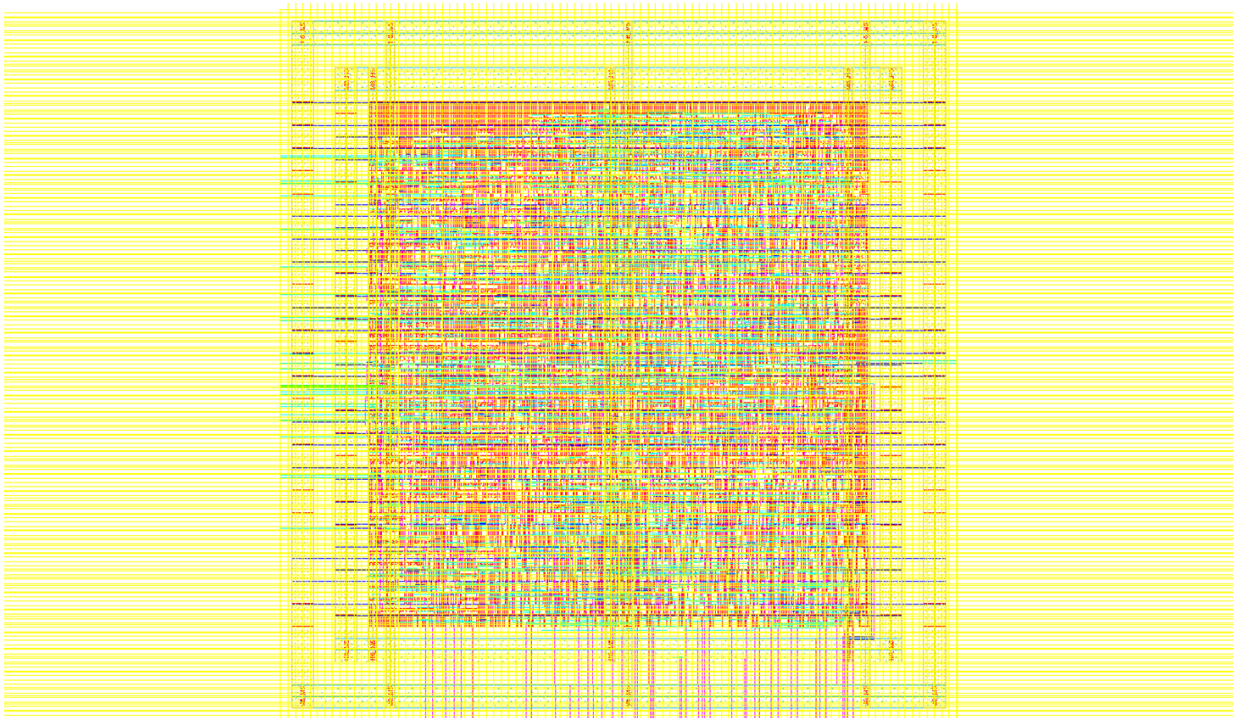


Figure 34 pipeline design layout

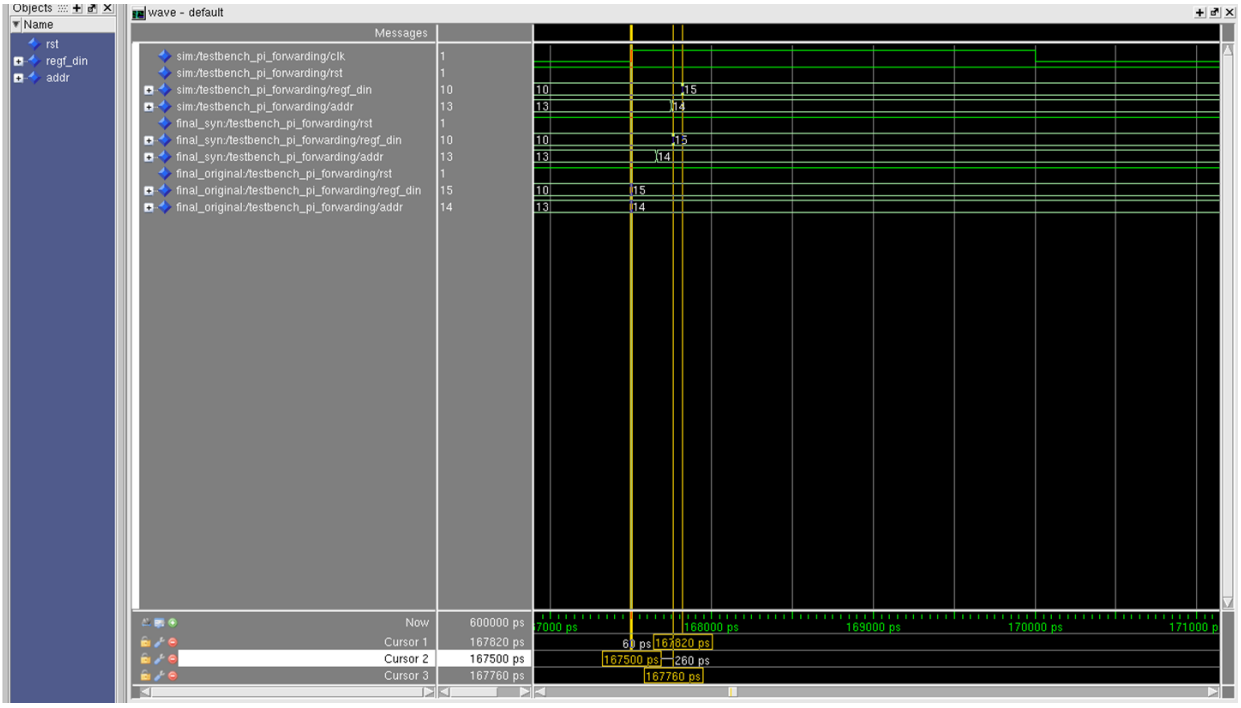


Figure 35 pipeline design simulation result

As shown in the simulation result figure, we can see the delay among the behavior simulation, synthesized simulation and simulation after place and route.

2. simulation result comparison

Single cycle design:

	Delay
BEH---> Synthesize	780ps
Synthesize---> Layout	3460ps

Pipelined design:

	Delay
BEH---> Synthesize	260ps
Synthesize---> Layout	3200ps

Comparing the time delay of the pipelined design to the single cycle design, we can see that it is a little shorter than the single cycle. It is because that we here use 200MHz to synthesize the pipelined design rather than 100MHz when synthesizing single cycle design. Another possible reason is that the mark point is not the same.

3. Problems and Solutions

To do synthesize and place & route is not difficult, but how to make sure the simulation result is correct is. Because of the different default situation in different level or the problems delay may create.

When simulating the synthesized and place & route design, we met several questions that lead to wrong waveforms.After going into the detailed waveform we found these problems are caught by the register file. First, the Moduleware's register file is triggered by electric level, instead of edge. Second, write and read simultaneously lead to errors. Third, write enable and read enable signal overlap.

To solve these problems, we modified the design.

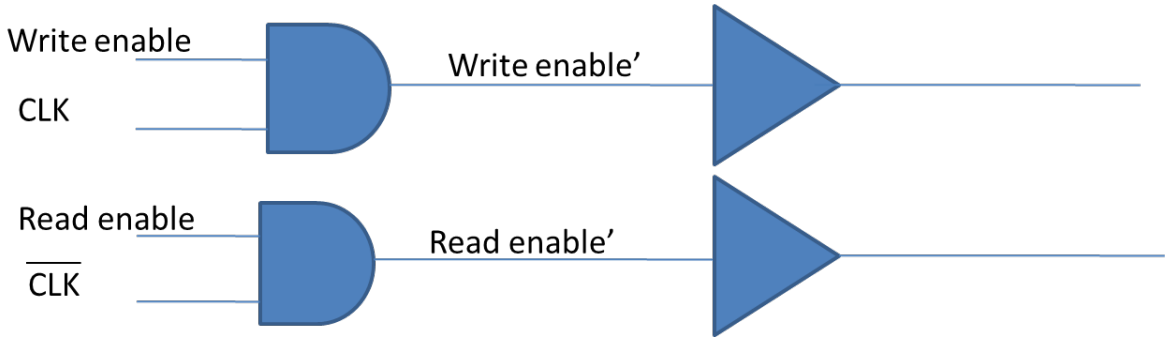


Figure 36 enable signal of the register file

After using the AND and buffer before the enable signals, we got the signals look like the following picture.

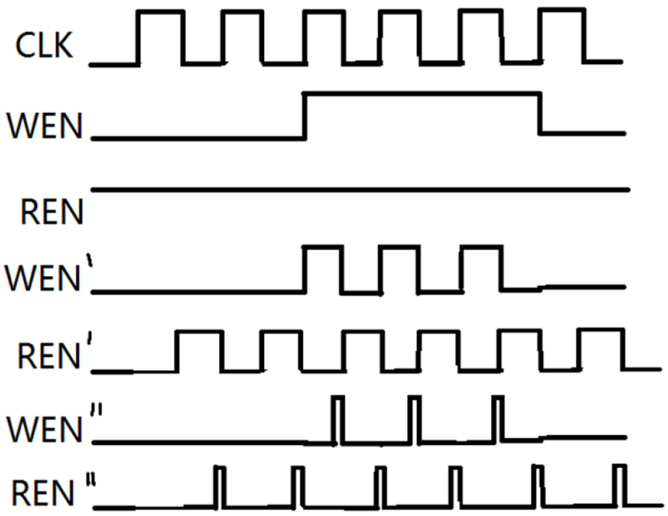


Figure 37

The register works correctly after modifying the enable signals

- Assembler

➢ **Instruction Memory**

As we adopt Harvard architecture to construct our MIPS like circuit, it has separate instruction memory and data memory. For the instruction memory, we choose to use the ROM from moduleware.

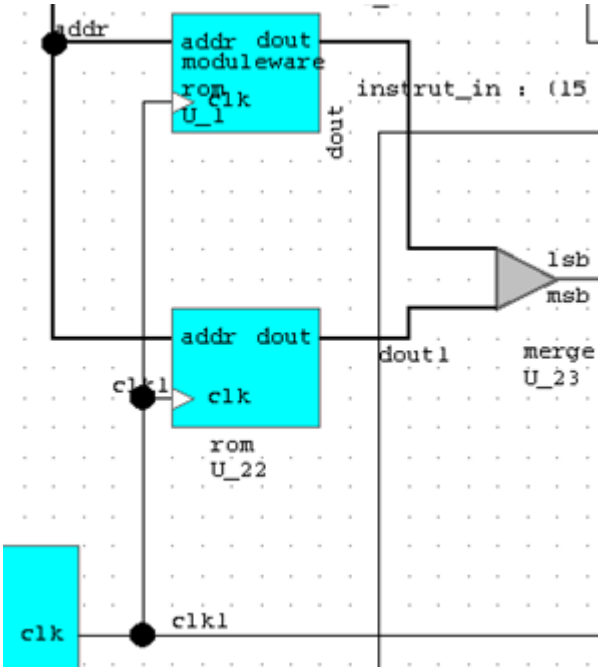


Figure 38 the design of ROM

This part implements a synthesizable Read Only Memory (ROM) generator which reads an initialization file during HDL generation. The generator supports Intel HEX file format. The Intel HEX-file-format only supports byte-addressing, so the width of the dout port must be exactly 8 bits. To build a 16-bit data-path, we need to use two ROMs in parallel with same address.

In addition, we created a *Perl* program which generate HEX file from binary code for us. It read a text file generated by assembler that each line of it corresponds to an instruction. And then output the two HEX file to fit the high byte and low byte ROM. The source code of this program is in Appendix.

➢ **Assembler**

An assembler creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. In our case, we created a Perl program as an assembler to translate assembly test code into

binary code. The key to achieving this in Perl is using regular expression to recognize the instruction from our instruction set, replace with binary code and reorganize it to follow the designated format.

Another important function of our assembler is to add bubble automatically. In our pipeline design, it takes 3 circles to determine if a jump or branch will be taken after the instruction been fetched. We need fill these 3 circles with bubble (opcode: bubble) to make it execute smoothly and correctly. While adding bubble manually while we write a program is feasible, it makes previous program written for single circle design incompatible. Plus different operation may need different amount of bubble to execute under some architecture optimization which makes it more complex and undesirable for programmer to remember. Consequently we need a smarter assembler that adds bubbles for us, which also make previous code compatible with current design. This transformation not simply equals adding bubbles, changing the offset of related instructions is a more important part. To implement this function, we need to realize that the key point of the offset is that the destination remains unchanged, that means if we can figure out a way record the destination and transfer that old PC to new PC then we can convert the offset to an adjusted offset. To achieve this goal, an array is created for transferring PC which new PC can be sourced by old PC. The Perl program source code of our assembler can be found in appendix.

APPENDIX:

VHDL code of the forwarding circuit at EX stage:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Forwarding IS
  port
  (
    M_W_RegisterRd,E_M_RegisterRd, I_E_RegisterRs, I_E_RegisterRt: IN
std_logic_vector(2 downto 0);
    M_W_RegWrite, E_M_RegWrite : IN std_logic;
    forward_A, forward_B : out std_logic_vector( 1 downto 0));

END ENTITY Forwarding;

--
ARCHITECTURE behave OF Forwarding IS
BEGIN

    forward_A <="10" WHEN ((E_M_RegWrite='1')  and (E_M_RegisterRd
=I_E_RegisterRs))ELSE
              "01" WHEN ((E_M_RegisterRd /= I_E_RegisterRs) and ((M_W_RegWrite='1')
and (M_W_RegisterRd = I_E_RegisterRs)))ELSE
              "00";


    forward_B <="10" WHEN ((E_M_RegWrite='1')  and (E_M_RegisterRd
=I_E_RegisterRt))ELSE
              "01" WHEN ((E_M_RegisterRd /= I_E_RegisterRt) and(( M_W_RegWrite='1')
and (M_W_RegisterRd = I_E_RegisterRt)))ELSE
              "00";


END ARCHITECTURE behave;
```

VHDL code of the hazard detection circuit for load instruction:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY stall IS
  port(
       I_E_RegisterRt, I_I_RegisterRs, I_I_RegisterRt : IN std_logic_vector ( 2 downto 0);
       I_E_MemRead: in std_logic;
       bubble: out std_logic );

END ENTITY stall;

--
ARCHITECTURE behavior OF stall IS
BEGIN

  bubble<= '1'when ((I_E_MemRead ='1') and ((I_E_RegisterRt = I_I_RegisterRs) or
(I_E_RegisterRt = I_I_RegisterRt)))else
           '0';


END ARCHITECTURE behavior;
```

VHDL code of the hazard detection for branch :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY nopbbbbb IS
  port
  (
   if_id_rt: in std_logic_vector(2 downto 0);
   if_id_rs: in std_logic_vector(2 downto 0);
   branch: in std_logic;
   id_ex_rt: in std_logic_vector(2 downto 0);
   id_ex_rd: in std_logic_vector(2 downto 0);
   id_ex_memread: in std_logic;
   id_ex_regwrite: in std_logic;
   ex_mem_rt: in std_logic_vector(2 downto 0);
   ex_mem_rd: in std_logic_vector(2 downto 0);
   ex_mem_regwrite: in std_logic;
   ex_mem_memread: in std_logic;
   clk: in std_logic;
   nop: out std_logic;
   enable: in std_logic;
--cnr1 is used for no dependence--
--cnr2 is used for dependece--
   cnr1: out  std_logic;
   cnr2: out  std_logic;
   state_type: out std_logic_vector(1 downto 0));
END ENTITY nopbbbbb;
ARCHITECTURE branch OF nopbbbbb IS
type state_type is (s0,s1,s2,s3);
signal state:state_type;
BEGIN
state_define: process(clk)
variable v0,v1,v2,v3,v4,v5 : std_logic_vector(2 downto 0);
variable v6,v7,v8,v9,v10: std_logic;
begin
v0:= if_id_rt;
v1:= if_id_rs;
v2:= id_ex_rt;
v3:= id_ex_rd;
v4:= ex_mem_rt;
v5:= ex_mem_rd;
v6:= branch;
v7:= id_ex_memread;
v8:= id_ex_regwrite;
```

```
v9:= ex_mem_regwrite;
v10:= ex_mem_memread;
if(clk'event and  clk) then
  if(enable) then
case  state is
  when s0 =>
    if((v9 = 1) and (v6 = 1) and ((v0 = v2) or (v1 = v2)))then
       state <= s1;
    elsif(((v9 /= 1) and (v8 = 1) and  (((v2 = v0) or (v2 = v1)) or ((v3 = v0) or (v3 = v1)))
and (v6 = 1)))then
       state <= s2;
    elsif(v10 = 1) and (v6 = 1) and ((v4 = v0) or (v4 = v1)) then
       state <= s2;
    end if;
  when s1 =>
    state <= s3;
  when s2 =>
    state <= s0;
  when s3 =>
    state <= s0;
  when others =>
    state <= s0;
end case;
case state  is
  when s0 =>
    nop <='1';
    cnr2 <='0';
    cnr1 <='0';
  when s1 =>
    nop <='0';
    cnr2 <='1';
    cnr1 <='0';
  when s2 =>
    nop <='0';
    cnr2 <='1';
    cnr1<= '0';
    state_type <= "01";
--01 stand for just one clock cycle data dependence--
  when s3 =>
    nop <='0';
    cnr2 <='1';
    cnr1 <='0';
--00 stand for no data dependence--
   when others =>
    nop <='1';
    cnr1 <='0';
```

```
cnr2 <= '0';
end case;
end if;
end if;
end process;
END ARCHITECTURE branch;
```

VHDL code of the forwarding circuit for branch

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY forwardforbranch IS
  port
  (
--for the id stage branch instructon--
  if_id_regread: in std_logic;
  if_id_rt: in std_logic_vector(2 downto 0);
  if_id_rs: in std_logic_vector(2 downto 0);
  if_id_branch: in std_logic;
  ex_mem_rt: in std_logic_vector(2 downto 0);
  ex_mem_rd: in std_logic_vector(2 downto 0);
  ex_mem_regwrite: in std_logic;
  ex_mem_memread: in std_logic;
  mem_wb_memread: in std_logic;
  mem_wb_rt: in std_logic_vector(2 downto 0);
  mem_wb_rd: in std_logic_vector(2 downto 0);
  mem_wb_regwrite: in std_logic;
  clk:in std_logic;
  forwardingA: out std_logic;
  forwardingB: out std_logic);
END ENTITY forwardforbranch;
ARCHITECTURE two OF forwardforbranch IS
BEGIN
forwardingA: process (clk)
variable v0,v1,v2,v3,v4,v5: std_logic_vector(2 downto 0);
variable v6,v7,v8,v9,v10,v11: std_logic;
begin
v0:= if_id_rt;
v1:= if_id_rs;
v2:= ex_mem_rt;
v3:= ex_mem_rd;
v4:= mem_wb_rt;
v5:= mem_wb_rd;
v6:= if_id_regread;
v7:= if_id_branch;
v8:= ex_mem_regwrite;
v9:= ex_mem_memread;
v10:= mem_wb_memread;
v11:= mem_wb_regwrite;
if(clk'event and clk) then
  if(((v4 = v0) or (v0 = v5)) and (v11 = 1) and (v6 = 1)) then
```

```
    forwardingA <="10";
  elsif((v9 /= 1) and (v8 = 1) and (v6 = 1) and ((v0 = v2) or (v0 = v3))) then
    forwardingA <= "01";
  else
    forwardingA <= "00";
end if;
end if;
end process;
forwardingB:process(clk)
variable v0,v1,v2,v3,v4,v5: std_logic_vector(2 downto 0);
variable v6,v7,v8,v9,v10,v11: std_logic;
begin
v0:= if_id_rt;
v1:= if_id_rs;
v2:= ex_mem_rt;
v3:= ex_mem_rd;
v4:= mem_wb_rt;
v5:= mem_wb_rd;
v6:= if_id_regread;
v7:= if_id_branch;
v8:= ex_mem_regwrite;
v9:= ex_mem_memread;
v10:= mem_wb_memread;
v11:= mem_wb_regwrite;
if(clk'event and clk) then
  if(((v1 = v4) or (v1 = v5)) and (v11 = 1) and (v6 = 1)) then
    forwardingB <= "10";
  elsif((v9/= 1) and (v8 = 1) and (v6 = 1) and ((v1 = v2) or (v1 = v3))) then
    forwardingB <= "01";
  else
    forwardingB <= "00";
end if;
end if;
end process;
END ARCHITECTURE two;
```

assembler_ble.pl

```
#!/usr/local/bin/perl
########################################
#assembler_ble.pl
#
# Function:*convert assembly to binary code;
#      *automatically add bubble, change offset;
#
# Author: Junxiang Wu
# Date:2013-4-20
########################################
#print "Input source name = $ARGV[0]\n";
if ($ARGV[1] eq "-code") {
   $CODEGEN = 1;
}else {
   $CODEGEN = 0;
}

%add_ble_num =(
   "j" => "3",
   "beq" => "3",
);

%with_offset =(
   "j" => undef,
   "beq" => undef,
);
$filename = $ARGV[0];

$source_pc = 0;
$out_pc =0;
$change_offset = undef;

open(FP,"$filename");
while(<FP>){

   chomp($_);
   $_ =~ s/,/ /g;
   $_ =~ s/\s+/ /g;
   ($opcode, $destC, $sourceA, $sourceB) = split(/ /,$_);
   $opcode = lc($opcode);# lc stands for lowercase, convert a string to lowercase.
   $destC =~ s/R//gi; #i means ignore capitcal,fits r or R
   $sourceA =~ s/R//gi;
   $sourceB =~ s/R//gi;
```

```
        ($opcode[$out_pc], $destC[$out_pc], $sourceA[$out_pc], $sourceB[$out_pc]) =
($opcode, $destC, $sourceA, $sourceB);

    $transfer[$source_pc] = $out_pc;

    if(exists $with_offset{$opcode}) {
        $change_offset = true;
        #printf $change_offset."\n";
        push @change_offset_pc, $source_pc;
        #push @old_pc,$source_pc;
    }

    if(exists $add_ble_num{$opcode}) {
        for(my $loop=0;$loop <$add_ble_num{$opcode};$loop++){
            $out_pc ++;
            ($opcode[$out_pc], $destC[$out_pc], $sourceA[$out_pc], $sourceB[$out_pc]) =
("ble", "", "", "");
        }
    }

        $out_pc ++;
    $source_pc ++;


}

if($change_offset)
{
    #printf $change_offset."\n";
    foreach $change_offset_pc(@change_offset_pc){
    #   printf $change_offset_pc."\n";
        $new_pc = $transfer[$change_offset_pc];
        $opcode = $opcode[$new_pc];
        if ($opcode =~ /\bbeq\b/) {
            #printf $opcode."\n";
            $destPC=$sourceB[$new_pc]+$change_offset_pc+1;
            $sourceB[$new_pc]=$transfer[$destPC]-$new_pc-1;
        }
        if ($opcode =~ /\bj\b/) {
            #printf $opcode."\n";
            #$destPC=$sourceB[$new_pc];
            $destC[$new_pc]=$transfer[$destC[$new_pc]];
        }

    }
}
```

```
for ($i=0;$i<$out_pc-1;$i++)
{
 ($opcode, $destC, $sourceA, $sourceB) =  ($opcode[$i], $destC[$i], $sourceA[$i],
$sourceB[$i]);

 #printf $opcode." ".$destC." ".$sourceA." ".$sourceB;
 #printf ("\n");
   if ($opcode =~ /\bble\b/) {
      printf ("01000");
      printf ("00000111100");
   }
   if ($opcode =~ /\badd\b/) {
      printf ("00100");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bsub\b/) {
      printf ("00000");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\band\b/) {
      printf ("00101");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bor\b/) {
      printf ("00110");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bxor\b/) {
      printf ("00001");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bsll\b/) {
      printf ("00010");
      printf ("%03b000%03b%02b",$sourceA, $destC, $sourceB);
   }
   if ($opcode =~ /\bsrl\b/) {
      printf ("00011");
      printf ("%03b000%03b%02b",$sourceA, $destC, $sourceB);
   }
   if ($opcode =~ /\bmul\b/) {
      printf ("00111");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\baddi\b/) {
      printf ("01100");
```

```
        $sourceB = 0b0000000000011111 & $sourceB;
        #support/in case of negtive number
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bandi\b/) {
        printf ("01101");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bori\b/) {
        printf ("01110");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\blw\b/) {
        printf ("10100");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bsw\b/) {
        printf ("11100");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bbeq\b/) {
        printf ("11000");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bj\b/) {
        printf ("10000");
        $destC = 0b0000011111111111 & $destC;
        printf ("%011b",$destC);
    }

    if ($CODEGEN) {
        printf ("\n");
    } else {
        printf("\t/*pc = %s * \n",$i);
    }

}
```

assembler_ble.pl

```
#!/usr/local/bin/perl
########################################
#assembler_ble.pl
#
# Function:*convert assembly to binary code;
#      *automatically add bubble, change offset;
#
# Author: Junxiang Wu
# Date:2013-4-20
########################################
#print "Input source name = $ARGV[0]\n";
if ($ARGV[1] eq "-code") {
   $CODEGEN = 1;
}else {
   $CODEGEN = 0;
}

%add_ble_num =(
   "j" => "3",
   "beq" => "3",
);

%with_offset =(
   "j" => undef,
   "beq" => undef,
);
$filename = $ARGV[0];

$source_pc = 0;
$out_pc =0;
$change_offset = undef;

open(FP,"$filename");
while(<FP>){

   chomp($_);
   $_ =~ s/,/ /g;
   $_ =~ s/\s+/ /g;
   ($opcode, $destC, $sourceA, $sourceB) = split(/ /,$_);
   $opcode = lc($opcode);# lc stands for lowercase, convert a string to lowercase.
   $destC =~ s/R//gi; #i means ignore capitcal,fits r or R
   $sourceA =~ s/R//gi;
   $sourceB =~ s/R//gi;
```

```
        ($opcode[$out_pc], $destC[$out_pc], $sourceA[$out_pc], $sourceB[$out_pc]) =
($opcode, $destC, $sourceA, $sourceB);

    $transfer[$source_pc] = $out_pc;

    if(exists $with_offset{$opcode}) {
        $change_offset = true;
        #printf $change_offset."\n";
        push @change_offset_pc, $source_pc;
        #push @old_pc,$source_pc;
    }

    if(exists $add_ble_num{$opcode}) {
        for(my $loop=0;$loop <$add_ble_num{$opcode};$loop++){
            $out_pc ++;
            ($opcode[$out_pc], $destC[$out_pc], $sourceA[$out_pc], $sourceB[$out_pc]) =
("ble", "", "", "");
        }
    }

        $out_pc ++;
    $source_pc ++;


}

if($change_offset)
{
    #printf $change_offset."\n";
    foreach $change_offset_pc(@change_offset_pc){
    #   printf $change_offset_pc."\n";
        $new_pc = $transfer[$change_offset_pc];
        $opcode = $opcode[$new_pc];
        if ($opcode =~ /\bbeq\b/) {
            #printf $opcode."\n";
            $destPC=$sourceB[$new_pc]+$change_offset_pc+1;
            $sourceB[$new_pc]=$transfer[$destPC]-$new_pc-1;
        }
        if ($opcode =~ /\bj\b/) {
            #printf $opcode."\n";
            #$destPC=$sourceB[$new_pc];
            $destC[$new_pc]=$transfer[$destC[$new_pc]];
        }

    }
}
```

```
for ($i=0;$i<$out_pc-1;$i++)
{
 ($opcode, $destC, $sourceA, $sourceB) =  ($opcode[$i], $destC[$i], $sourceA[$i],
$sourceB[$i]);

 #printf $opcode." ".$destC." ".$sourceA." ".$sourceB;
 #printf ("\n");
   if ($opcode =~ /\bble\b/) {
      printf ("01000");
      printf ("00000111100");
   }
   if ($opcode =~ /\badd\b/) {
      printf ("00100");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bsub\b/) {
      printf ("00000");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\band\b/) {
      printf ("00101");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bor\b/) {
      printf ("00110");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bxor\b/) {
      printf ("00001");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\bsll\b/) {
      printf ("00010");
      printf ("%03b000%03b%02b",$sourceA, $destC, $sourceB);
   }
   if ($opcode =~ /\bsrl\b/) {
      printf ("00011");
      printf ("%03b000%03b%02b",$sourceA, $destC, $sourceB);
   }
   if ($opcode =~ /\bmul\b/) {
      printf ("00111");
      printf ("%03b%03b%03b00",$sourceA, $sourceB, $destC);
   }
   if ($opcode =~ /\baddi\b/) {
      printf ("01100");
```

```
        $sourceB = 0b0000000000011111 & $sourceB;
        #support/in case of negtive number
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bandi\b/) {
        printf ("01101");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bori\b/) {
        printf ("01110");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\blw\b/) {
        printf ("10100");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bsw\b/) {
        printf ("11100");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bbeq\b/) {
        printf ("11000");
        $sourceB = 0b0000000000011111 & $sourceB;
        printf ("%03b%03b%05b",$sourceA, $destC, $sourceB);
    }
    if ($opcode =~ /\bj\b/) {
        printf ("10000");
        $destC = 0b0000011111111111 & $destC;
        printf ("%011b",$destC);
    }

    if ($CODEGEN) {
        printf ("\n");
    } else {
        printf("\t/*pc = %s * \n",$i);
    }

}
```

hex.pl

```
#!/usr/local/bin/perl
$filename = $ARGV[0];
open(FP,"$filename");
$data[2];
$data_sum[2];
$line=0;

while(<FP>){
   chomp($_);
   my $temp= '0b'.substr($_,0,8);
   my $value =oct($temp);
   $data[0] .= sprintf("%02X",$value);
   $data_sum[0] += $value;
   #printf ("%02X",$value);
   #print "    ";

   my $temp= '0b'.substr($_,8,8);
   my $value =oct($temp);
   $data[1] .= sprintf("%02X",$value);
   $data_sum[1] += $value;
   #printf ("%02X",$value);

   #print "    ";
   $line++;
   #print $line;
   #print "\n";
}

$filename =~ s/\.(\w+)$//;

$checksum = -($line + $data_sum[0])& 0xff;
open(FH,">".$filename."_high.hex");
printf FH (":%02X000000".$data[0]."%02X\n",$line,$checksum);
print FH (":00000001FF");
close(FH);

$checksum = -($line + $data_sum[1])& 0xff;
open(FH,">".$filename."_low.hex");
printf FH (":%02X000000".$data[1]."%02X\n",$line,$checksum);
print FH (":00000001FF");
close(FH);
```

generate.pl

```
#!/usr/local/bin/perl
$filename = $ARGV[0];
$name=$filename;
$name =~ s/\.(\w+)$//;
$name .="_b.txt";

`perl assembler_ble.pl $filename -code > $name`;
`perl hex.pl $name`;
#`del /q $name`;
print "Thank you! \n";
```