Computer architecture project

Tomasulo algorithm application

Final report

Group member: Yuanjun Dai, Junxiang Wu, Zhe

Section one: requirement

Requirement
1.Run environment: windows 7 or above
2.Compile: Visual Studio 2008 or above
Complie
You can use exsit project in tomasulo floder: double click tomasulo.sln
Or you may want to create a new project in Visual Studio:
1.create a C++ project in visual studio,project windows console application, select empty project on creation
2.import all source file (*.cpp) under "\tomasulo\tomasulo" as your source file
3.import all header file (*.h) under "\tomasulo\tomasulo" as your header file , then press build or debug
Input and Output
The program will read input from its current path and wrtie to the same floder. Which means you need to put input and program in the same floder.When you complie in Visual Studio,you need to put the put file under the "projectname\Debug\"
1. Input.txt: Instructions.
2. Config.txt: Hardware configuration, Register & Memory Initialization.
3. Output.txt: Result file.
Execute Instruction

1.Executable file name: tomasulo.exe
2.You can run demo by runing tomasulo.exe under "demo" flode
Support Operation Code

1. Add/Sub: integral and immediate.
2. Add.d/Sub.d: float and immediate.
3. Mul.d: float and immediate.
4. Sd/Ld: Example: Ld R1, 3(R1),
5. Beq/Bne: Example: Bne R1,R2,-3

Section two: test case

Introduction:

Hardware description:

Project description: In this project, we are going to implement the Tomasulo algorithm with register renaming, ROB, speculative execution (branch prediction + misprediction handling)*, and memory disambiguation technique. We will use java to perform our simulation, in order to make debugging easier, we will also design UI interfaces for every stage.

Hardware description:The architecture of this CPU is different from the MIPS architecture we know. In the traditional design, we divide the whole chips into IF, ID, EX, MEM and WB stages. However, this kind of machine cannot perform multi-issue instruction and out of order performing. With tomasulo algorithm, the machine can achieve function and the function of each stage is different from normal pipeline. In the new machine, we divide the machine into Issue, Ex , Mem, Wb and Commit stages. And I am going to tell what function or work each stage will finish.

Issue stage:

1.Decide which address is going to use(pc+4/branch address/misprediction)
2.fetch instruction from instruction queue
3.put the instruction into reservation station, decoding and doing renaming related to ROB &RAT

Ex stage:

1.get data from the issue stage
2.do the calculation work if we need(integer/floating point calculation, laod/store address calculation, branch comparision)

Mem stage:

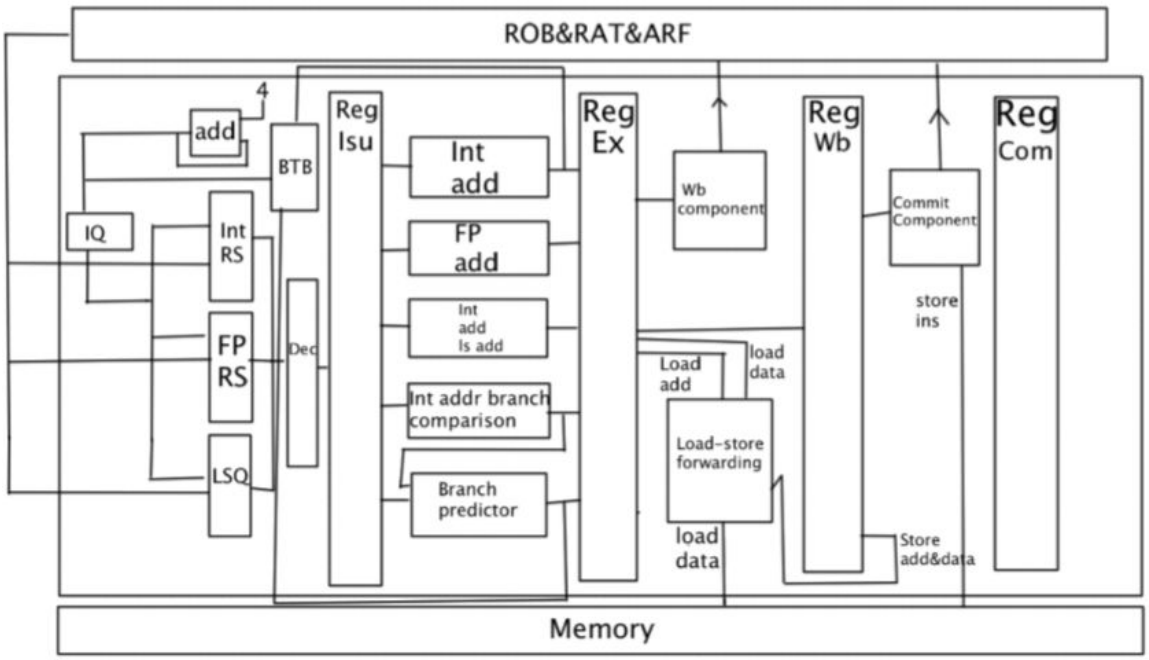
1.This stage is just used for load and store instruction to get or store data into the memory component.

Wb stage:

1.For the instruction which has destination register will write the result into the ROB and broadcast the data to the instruction which need the current data through CDB.
2.The memory operating instruction has memory dependence will operate the load-store forwarding operation at this stage.

Commit stage:

1.In the commit stage, the instruction will write the data from ROB to ARF.
2.Renew the RAT
3.Delete the instruction from reservation station.
Illustration of this diagram:



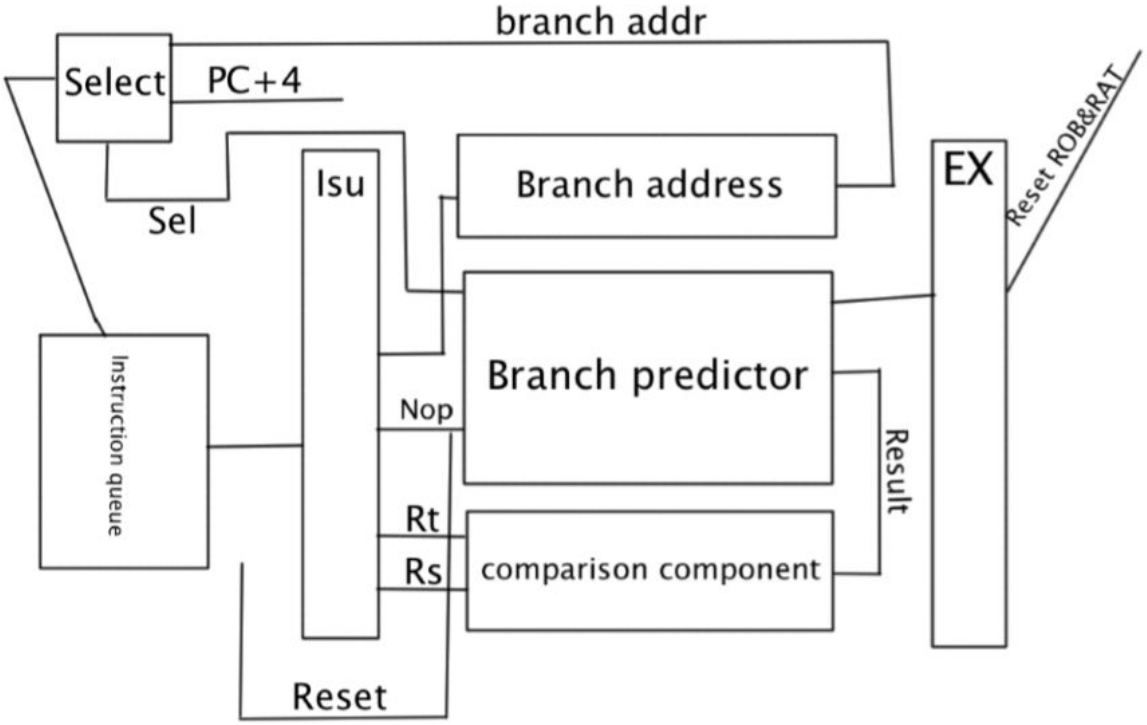
1.ROB&RAT&ARF does not belongs to any unique stage, it is a global components which will be used in ISSUE, WB and COMMIT stage. 2.MEM stage should be viewed as a component instead of specific stages. It is just used by load and store instructions. And it will be used by load instruction at MEM stage after EX stage and it will be used by store instruction after COMMIT stage. 3.In addition to some basic function units which is used to do the calculation job, we still

need other adders to perform other works, such as do the comparison for branch instruction, calculating the address for branch instruction and load/store instruction. The following picture will be more precise about EX stage.

Major component:

Branch predictor:

1.Branch prediction without BTB



Demonstration of brancThis component shows the branch prediction system without BTB. The branch instruction work in this way: Cycle one: Issue stage detects the branch instruction using opcode. Cycle two: Issue stage has been stalled and Branch instruction goes into EX stage, at the same time, it will forward the target address and the prediction from branch predictor to issue stage to select the next address. At the same time the comparison component will finish the result and know whether the prediction is right or wrong. Cycle Three: Next address will be fetched, if the branch prediction is right, then we do not need to do anything. If the prediction is wrong there are several things we have to be concern. 1. The branch predictor should change its predict result. 2. The fetched instruction should be clear by resetting the stage register and major components such as reservation, ROB and RAT. We should also send the right address to the next pc selection component. Cycle four: In this cycle we should get the new address in the issue stage.

2. Branch prediction with BTB

When we use branch prediction with BTB, the branch instruction will be operated at issue stage based on the predict result of Branch predictor which will save one cycle than the branch predictor without BTB. The branch instruction should operated follow the sequence below.

When the cpu read one instruction from the instruction queue in issue stage. If the fetched instruction is not branch instruction, then the program will run as usual.If the fetched instruction is branch instruction.Then after ID stage, the cpu will search the BTB and try to find a “hit”(hit means the branch instruction already exists in the BTB. If hit happens, the cpu will fetch the target address from the BTB entry and make the branch decision based on the branch prediction

history which stored in the BTB entry. If miss happens, the BTB will record this branch instruction in one new BTB entry and get the target address and branch prediction history when the branch instruction go out of EX stage(we use ALU in EX stage to calculate the target address and branch decision result)

If the branch instruction hit at BTB, we will execute the program as usual. If the branch instruction is predicted to be taken then the next instruction after the branch instruction should be the instruction which is stored in the target address stored in BTB. If the branch instruction is predicted to be not taken, next instruction should be the normal next PC.

If the prediction is wrong, then the whole system should wipe out the wrong instruction after the branch instruction.As the branch result will come out at EX stage, then we only need to wipe out one instruction in ISSUE stage and jump back to the right target address(we should store the address before we branch).

3.ROB&RAT

These two components are extremely important to Tomasulo algorithm, the ROB is responsible for the renaming of the target registers and RAT will do the mapping between ARF and ROB.

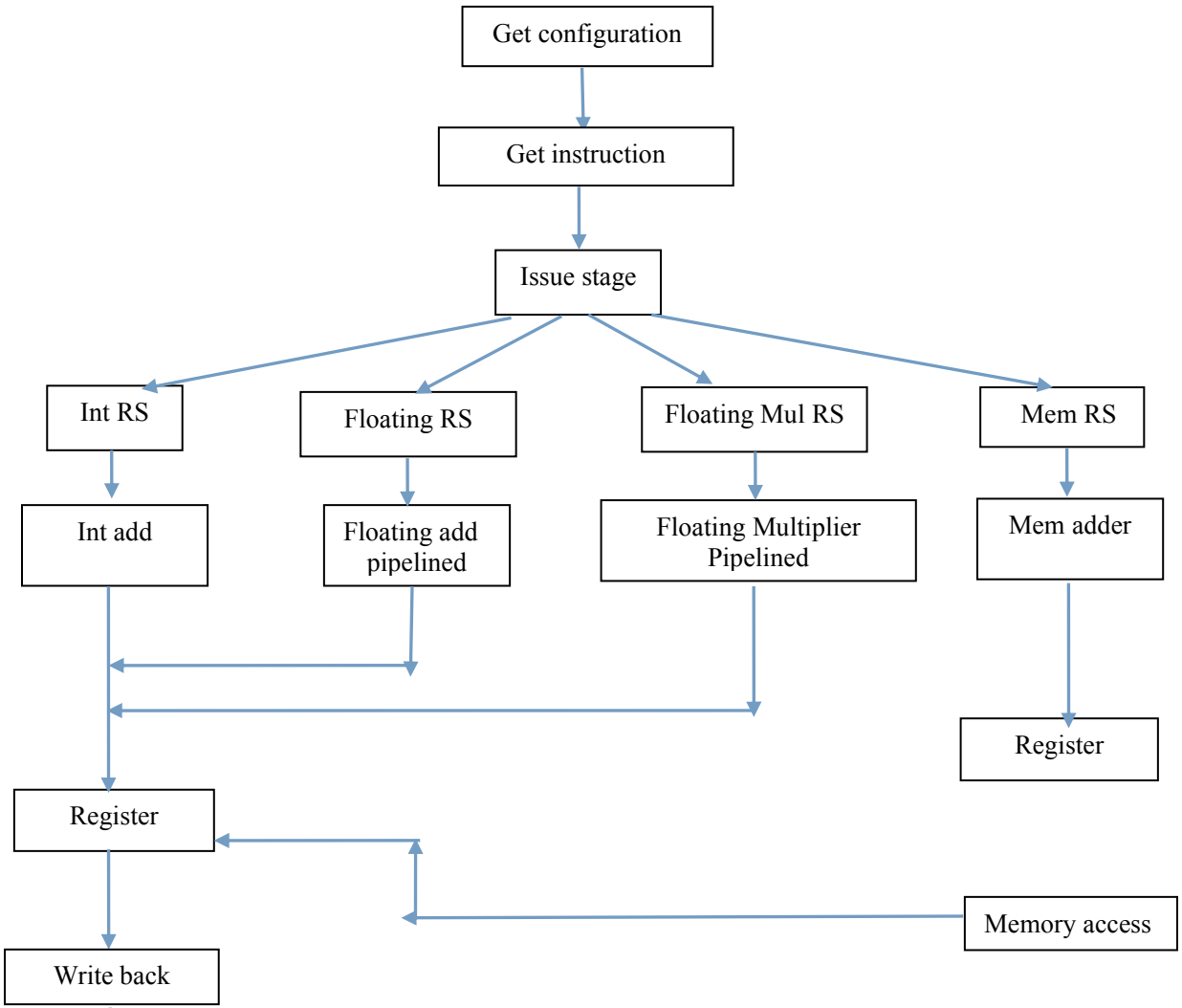
When one instruction comes in, the CPU will detect whether the instruction has target registers. If it does, then the ROB will do the renaming for the instruction and store the result in the corresponding registers in ROB. And the RAT will map the ARF and ROB at the same time.

The ROB should be written at WB stage and will write the its data to ARF at commit stage based on the mapping stored in RAT. And the RAT will also renew when it is in commit stage.

4.Pipeline function unit

The pipelined function units allow several instructions execute in them at the same time. Which can run the program much more efficience.

Code structure:



Test case: The test case I created is to test the function of our code step by step, every test cases has its own target component to test. And they will be illustrated in the illustration part.

Part one: No data dependence Cycles in Me

I will divide the test cases into two parts as different instruction has different data-flow and go through different component in physical design and different parts of program in the code. 1. No dependence, no branch prediction, no load&store instruction: The input data, instructions and specification:

1. No dependence, no branch prediction, no load&store instruction: The input data, instructions and specification:2

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

	Entry	Initialization number
ROB	128	
R1		1
R2		2
R0		0
F2		2.2
F1		1.1
Mem[4]		2.2

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

	Entry	Initialization number
ROB	128	
R1		1
R2		2
R0		0
F2		2.2
F1		1.1
Mem[4]		2.2

Input program:

Add R3, R1,R0

Addi R4, R1,#1

Add.d F3, F2,F1

Sub.d F4, F2,F1

Mult.d F5, F2, F1

Sub R5, R1,R0

Opcode	Rd /RT/Fd	Rs/Fs	Rt/Imme/Ft	Issue	Ex	Mem	Wb	Commmit
Add	R3	R1	R0	1	2		3	4
Addi	R4	R1	#1	2	3		4	5
Add.d	F3	F2	F1	3	4-6		7	8
Sub.d	F4	F2	F1	4	5-7		8	9
Mult.d	F5	F2	F1	5	6-25		27	28
Sub	R5	R1	R0	6	7		9	10

R0	0	F1	1.1
R1	1	F2	2.2
R2	2	F3	3.3
R3	1	F4	1.1
R4	2	F5	2.42
R5	1		

In this test case, we should check the output files and ARF to verify the result.

2. load&store instruction without data dependence:
   The input data, instructions and specification:

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

	Entry	Initialization number
ROB	128	
R1		1
R2		2
R0		0
F2		2.2
Mem[0]		0
Mem[4]		1.1

Input instruction:
Load F3, #1[R0]
Store F2, #2[R0]

Opcode	Rd /RT/Fd	Rs/Fs	Issue	Ex	Mem	Wb	Commit	Mem
Load	F3	#4[R0]	1	2	3-6	7	8	
Store	F2	#8[R0]	2	3		4	9	6-9

F2	2.2	MEM[0]	0
F3	1.1	MEM[4]	1.1
		MEM[8]	2.2

In this test case, we should check the output files, data in memory and ARF.

Note:
Add.d F4, F2 , F1;// F4 RAW, WAW
Sub.d F5, F4 , F1;// F5 RAW, F4 WAR
Mul.d F4, F5, F1;/

F1	1.1
F2	2.2
F3	3.3
F4	2.42
F5	2.2

We can use output files and ARF to check the result.

Part 3:Load/Store forwarding

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

	Entry	Initialization number
ROB	128	
R1		1
R2		2
R0		0
F1		1.1
F2		2.2
Mem[0]		0

Input instruction:
Add.d F3, F2,F1
Store F3, #4[RO];
Load F4, #4[RO];

Opcode	Rd /RT/Fd	Rs/Fs	Rt/Imme/Ft	Issue	Ex	Mem	Wb	Commit	Mem
Add.d	F3	F2	F1	1	2-4		5	6	
Store	F3	#4[R0]		2	3		6	7	8-11
Load	F4	#4[R0]		3	4	7	8	9	

F1	1.1	MEM[0]	0
F2	2.2	MEM[4]	3.3
F3	3.3		
F4	3.3		

We have to explain how the load/store instruction and how the load/store forwarding circuit works here. 1. The sequence that the load instruction go through the CPU should be Issue, Ex, Mem, Wb, commit. We assume the load instruction write back its target address to load/store queue in Mem stage 2. The sequence that the store instruction go through the CPU should be Issue, Ex, Wb, commit, Mem. We assume the store instruction write back its target address to load/store queue in Wb stage. 3. How load-store forwarding circuit works? In our design, we have

the following assumption: 1. We can do NOP to every stage register. 2. There are store instructions appear before load instruction.
In this case, the load instruction can just be operated after all the address of store instructions have ben calculated, in order to check the memory dependences situation. After the store instruction get the data of register which is used to calculated the target address, it can go through Issue stage and Ex stages. If the store instruction has some data dependence with the previous instruction, then the store instruction can just go into the Wb stage when it gets the data it needs from ROB. In order to design our chip correctly, we need to make some assumption. We assume that the store instruction will write its target address into load-store queue in the EX stage after the finish of calculation. We also assume the load instruction did it in this way. However, the CDB can just be occupied by one instruction in one cycle. Thus, the load instruction should execute the EX stage one clock cycle at least after the previous store instruction go into the EX stage. With the same reason, the load instruction should go into the MEM stage one clock cycle after the store instruction goes into the WB stage.

Part 4:Structure hazards:

I divide this test case into two parts, the first one is for reservation station structure hazards, the second one is for the function unit structure hazards.

1. Structure hazards in reservation station:

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	1	1		1
FP adder	1	3		1
FP multiplier	1	20		1
Load/store unit	1	1	4	1

In order to make the test case more clear and easier, I try to design the RS as lease as possible.

	Entry	Initialization number
ROB	128	
R1		1
R2		2
R0		0
F2		2.2
F1		1.1
Mem[4]		2.2

Input instruction:
Add R3, R1, R0
Addi R4, R1,#1
Add.d F3,F2,F1
Sub.d F4,F2,F1
Mult.d F5,F2,F1
Mult.d F6,F5,F4
Store F6, #4[R0]

Load F7, #4[R0]

Opcode	Rd /RT/Fd	Rs/Fs	Rt/Imme/Ft	Issue	Ex	Mem	Wb	Commmit
Add	R3	R1	R0	1	2		3	4
Addi	R4	R1	#1	5	6		7	8
Add.d	F3	F2	F1	6	7-9		10	11
Sub.d	F4	F2	F1	12	13-16		17	18
Mult.d	F5	F2	F1	13	14-33		34	35
Mult.d	F6	F5	F4	36	37-56		57	58
Store	F6	#4[R0]		37	38		58	59
Load	F7	#4[R0]		60	61	62-64	65	66

R0	0	F1	1.1	MEM[4]	2.662
R1	1	F2	2.2		
R2	2	F3	3.3		
R3	1	F4	1.1		
R4	2	F5	2.42		
		F6	2.662		
		F7	2.662		

2. Structure hazard in function unit In order to test the structure hazard in function unit, I try to make avoid any structure hazard in reservation station. In this case, I try to make the reservation station larger than usual.

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	10	1		1
FP adder	10	3		1
FP multiplier	10	20		1
Load/store unit	10	1	4	1

	Entry	Initialization number
ROB	128	
R1		1
R2		2
R0		0
F2		2.2
F1		1.1
Mem[4]		2.2

Input instruction:
Add R3, R1, R0
Addi R3, R1, R0
Add.d F3,F2, F1
Sub.d F4,F2, F1
Add.d F8, F2,F1
Sub.d F9, F2,F1
Mult.d F5,F2,F1
Mult.d F6,F5, F4
Store F6, #4[R0]
Load F7, #4[R1]

Opcode	Rd /RT/Fd	Rs/Fs	Rt/Imme/Ft	Issue	Ex	Mem	Wb	Commmit	Mem
Add	R3	R1	R0	1	2		3	4	
Addi	R4	R1	#1	2	3		4	5	
Add.d	F3	F2	F1	3	4-6		7	8	
Sub.d	F4	F2	F1	4	5-7		8	9	
Add.d	F8	F2	F1	5	6-8		9	10	
Sub.d	F9	F2	F1	6	7-9		10	11	
Mult.d	F5	F2	F1	7	8-27		28	29	
Mult.d	F6	F5	F4	8	29-48		49	50	
Store	F6	#4[R0]	7	9	10		50	51	52-53
Load	F7	#4[R1]	8	10	11	12-14	15	52	

R0	0	F1	1.1	MEM[4]	2.662
R1	1	F2	2.2	MEM[8]	2.662
R2	2	F3	3.3	F8	3.3
R3	1	F4	1.1	F9	1.1
R4	2	F5	2.42		
		F6	2.662		
		F7	2.662		

We can check the ARF, memory to do the verification.

Part five: Simple Loop:

1. Without any branch prediction and branch target block:

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	10	1		1
FP adder	1	3		1
FP multiplier	10	20		1
Load/store unit	10	1	4	1

	Entry	Initialization number
ROB	128	
R1		0
R2		2
R0		0
F2		2.2
F1		1.1
Mem[4]		2.2

Input instruction:

Addi R3, R1,#1

Add.d F3, F1, F2

Sub.d F4, F2, F1

Mult.d F5, F4, F3

Bnq R3, R2, -6

Opcode	Rd /RT/Fd	Rs/Fs	Rt/Imme/Ft	Issue	Ex	Mem	Wb	Commmit	Mem
Addi	R3	R1	#1	1	2		3	4	
Add.d	F3	F1	F2	2	3-5		6	7	
Sub.d	F4	F2	F1	8	9-11		12	13	
Mult.d	F5	F4	F3	9	13-32		33	34	
Bnq	R3	R2	-6	10	11		12	13	
Addi	R3	R1	#1	11	12		13	14	
Add.d	F3	F1	F2	12	13-15		16	17	
Sub.d	F4	F2	F1	18	19-21		22	23	
Mult.d	F5	F4	F3	19	23-42		43	44	
Bnq	R3	R2	-6	20	21		22	23	

R0	0	F0	0
R1	0	F1	1.1
R2	2	F2	2.2
R3	2	F3	3.3
R4		F4	1.1
R5		F5	3.63
R6		F6	

We can check ARF to verify our result

Part Six: Branch prediction:

1.Branch prediction without BTB

We use 1 bit branch predictor in our design, and we set our branch predictor to 0 at first.

	# of RS	Cycles in EX	Cycles in Mem	# of FUS
Integer adder	10	1		1
FP adder	10	3		1
FP multiplier	10	20		1
Load/store unit	10	1	4	1

	Entry	Initialization number
ROB	128	
R1		0
R2		2
R0		0
R3		2
R4		0
R5		2

Input instruction

Add R6,R2,R1

Beq R0,R1,-2

Sub R8,R2,R1

Opcode	Rd /RT/Fd	Rs/Fs	Rt/Imme/Ft	Issue	Ex	Mem	Wb	Commmit	Mem
Add	R6	R2	R1	1	2		3	4	
Beq	R0	R1	-2	2	3		4	5	
Sub	R8	R2	R1	4					
Add	R6	R2	R1	5	6		7	8	
Beq	R0	R1	-2	6	7		8	9	

R0	0	R6	2
R1	0	R7	0
R2	2	R8	2
R3	2		
R4	0		
R5	2		

New test case two:

This test case is used to test whether the branch instruction can branch up or not

	# of rs	Cycle in EX	Cycle in Mem	# of FUs
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

ROB entries =10
R1=21,R3=2, R2=1, R0=0, R5=1, R6=6, F1=4.1,F2=3.2,F6=3.0
Mem[4]=2.2

```
Add R1,R7,R3
Add R2,R2,R5
Add.d F7,F6,F1
Add R4,R1,R3
Add R8,R1,R3
Beq R2 R3 -5
Add.d F3,F2,F1
```

	ISSUE	EX	MEM	WB	COMMIT
Add R1,R7,R3	1	2		3	4
Add R2,R2,R5	2	3		4	5
Add.d F7,F6,F1	3	4		7	8
Add R4,R1,R3	4	5		6	9
Add R8,R1,R3	5	6		8	10
Beq R2 R3 -5	6	8		9	11
Add R1,R7,R3	8	9		10	12
Add R2,R2,R5	9	10		11	13
Add.d F7,F6,F1	10	11		14	15
Add R4,R1,R3	11	12		13	16
Add R8,R1,R3	12	13		15	17
Beq R2 R3 -5	13	15		16	18
Add.d F3,F2,F1	15	16		19	20

Register Values:

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
0	2	3	2	4	1	6	0	4	0	0	0	0	0	0	0	0	0
R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31	R32	R33	R34	R35
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
0	4.1	3.2	0	0	0	3	7.1	0	0	0	0	0	0	0	0	0	0

New test three:(used to solve the specific bug in the program)
This test case is used to test the load and store instruction.

	# of rs	Cycle in EX	Cycle in Mem	# of FUs
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

ROB entries =10
R1=21,R3=3, R2=0, R0=0, R5=1, R6=6, F1=4.1,F2=3.2,F6=3.0
Mem[4]=2.2

```
Add.d F3 F2 F1
Add  R3,R4,R5
Add  R7,R5,R1
Sub  R8 R6 R2
Sd   F3,4(R0)
Add  R3,R4,R5
Add  R7,R5,R2
Ld   F4,4(R0)
```

		ISSUE	EX	MEM	WB	COMMIT
Add.d F3 F2 F1	1	2		7	8	
Add R3,R4,R5	2	3		4	9	
Add R7,R5,R1	3	4		5	9	
Sub R8 R6 R2	4	5		6	9	
Sd F3,4(R0)	5	7	8	12	13	
Add R3,R4,R5	6	7		8	14	
Add R7,R5,R2	7	8		9	14	
Ld F4,4(R0)	8	9	12	16	17	

Register Values:

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
0	21	0	1	0	1	6	1	6	0	0	0	0	0	0	0	0	
R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31	R32	R33	R34	R35
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
0	4.1	3.2	7.3	7.3	0	3	0	0	0	0	0	0	0	0	0	0	
F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31	F32	F33	F34	F35
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

New test four:
This test case is used to test the data dependence

	# of rs	Cycle in EX	Cycle in Mem	# of FUs
Integer adder	2	1		1
FP adder	3	3		1
FP multiplier	2	20		1
Load/store unit	3	1	4	1

ROB entries =10
R1=1, R2=2, R0=0, F1=1.1,F2=2.2
Mem[4]=2.2

Add R3,R1,R0
Add R4,R3,1
Add R4,R2,R1
Sub R2,R4,R1

	ISSUE	EX	MEM	WB	COMMIT
Add R3,R1,R0	1	2		3	4
Add R4,R3,1	2	3		4	5
Add R4,R2,R1	3	4		5	6
Sub R2,R4,R1	4	5		6	7

Register Values:

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
0	1	2	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0
R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31	R32	R33	R34	R35
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
0	1.1	2.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31	F32	F33	F34	F35
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Memory values:
Mem[4]=2.2

New test case five
This test case is the demo test case

	# of rs	Cycle in EX	Cycle in Mem	# of FUs
Integer adder	4	1		1
FP adder	3	4		1
FP multiplier	2	15		1
Load/store unit	5	1	5	1

ROB entries =64
R1=12, R2=32, F20 = 3.0
Mem[4]=3.0, Mem[8]=2.0, Mem[12]=1.0, Mem[24]=6.0, Mem[28]=5.0, Mem[32]=4.0

```
Ld F2, 0(R1)
Mul.d F4, F2, F20
Ld F6, 0(R2)
Add.d F6, F4, F6
Sd F6, 0(R2)
Add R1, R1, -4
Add R2, R2, -4
Bne R1, R0, -7
Add.d F20, F2, F2
```

	ISSUE	EX	MEM	WB	COMMIT
Ld F2, 0(R1)	1	2	3	8	9
Mul.d F4, F2, F20	2	8		23	24
Ld F6, 0(R2)	3	4	8	13	25
Add.d F6, F4, F6	4	23		27	28
Sd F6, 0(R2)	5	27	28	33	34
Add R1, R1, -4	6	7		9	35
Add R2, R2, -4	7	9		10	36
Bne R1, R0, -7	8	10		11	37
Ld F2, 0(R1)	10	28	33	38	39
Mul.d F4, F2, F20	11	38		53	54
Ld F6, 0(R2)	12	29	38	43	55
Add.d F6, F4, F6	13	53		57	58
Sd F6, 0(R2)	14	57	58	63	64
Add R1, R1, -4	15	16		17	65
Add R2, R2, -4	16	17		18	66
Bne R1, R0, -7	17	18		19	67
Ld F2, 0(R1)	18	58	63	68	69
Mul.d F4, F2, F20	19	68		83	84
Ld F6, 0(R2)	27	59	68	73	85
Add.d F6, F4, F6	28	83		87	88
Sd F6, 0(R2)	29	87	88	93	94
Add R1, R1, -4	30	31		32	95
Add R2, R2, -4	31	32		34	96
Bne R1, R0, -7	32	34		35	97
Add.d F20, F2, F2	34	84		88	98

Register Values:

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
0	0	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31	R32	R33	R34	R35
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17
0	0	3	0	9	0	15	0	0	0	0	0	0	0	0	0	0	

F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31	F32	F33	F34	F35
0	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Memory values:

Mem[4]=3	Mem[8]=2	Mem[12]=1	Mem[24]=15	Mem[28]=11	Mem[32]=7

Section 3: responsibility

Architecture design and function description:
Major: Yuanjun Dai Associate: Junxiang Wu, Zheng Liu
Programming :
Major: Zheng Liu Associate: Zheng Liu
Debugging:
Major: Junxiang wu Associate: Zheng Liu, Yuanjun Dai
Test case design
Major Yuanju Dai Associate: Junxiang Wu, Zheng Liu
Report writing:
Major: Yuanjun Dai , Zheng Liu

The code main developer is zheng Liu, he develops the whole program and junxiang wu also helps him to discuss how to implement it in c++ and how to solve the problem of the coding.

All the members contributes their effort to debugging, we always try to find the bug together. Junxiang Wu is the guy who solve most of the bugs in the program.

Mostly of the test cases are made by Yuanjun Dai, including the old test cases and new test cases. With the deep understanding of the hardware mechanism, always try to find and explain the bugs in mechanism and explain how the chips should work. Also responsible for the phase one and phase two report writing.