32-bit 5-stage RISC Pipeline Processor with 2-Bit Dynamic Branch Prediction Functionality

By

CHANG BOON CHIAO

1200485

SUPERVISED BY

MR. MOK KAI MING

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfilment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONS)

COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Perak Campus)

Jan 2015

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: _____

_____

_____

Academic Session: _____

I _____

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

_____ _____

(Author's signature) (Supervisor's signature)

Address:

_____

_____ _____

_____ Supervisor's name

Date: _____ Date: _____

DECLARATION OF ORIGINALITY

I declare that this report entitled "**32-bit 5-stage RISC Pipeline Processor with 2-Bit Dynamic Branch Prediction Functionality**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : _____

Name : _____

ACKNOWLEDGEMENTS

I would like to say thanks lot to my final year project supervisor, Mr. Mok Kai Ming. Thanks for his guidance throughout the project with the right direction, good patience and also well explanation in helping me to complete my project. He had helped me a lot to understand my project as well as making me to get really interested into it. The project might not be completed without his guidance. Really thanks for spending so much of precious time with me throughout the whole project.

Next, I wish to thanks my friends, who support me when I am facing problems and help to comfort me and release my stress when I am feeling pressure and tension.

Last but not the least, a big thank you to my family for their support and encouragement to continue to work on and complete the course.

Chang Boon Chiao

ABSTRACT

This project is an integration of the Branch Predictor for the development of the RISC32 processor based on RISC Architecture that previously developed in Universiti Tunku Abdul Rahman under Faculty of Information and Communication Technology. After reviewing the previous projects, there is a part where the developed branch target buffer is not yet integrate into the processor and the coverage of the control hazards is not well defined and implemented. The purpose of this project is to integrate the branch target buffer into the processor with a branch predictor and design the control logic to handle control hazards caused by different instructions. All the modeling will be using Verilog which is a type of Hardware Description Language and verification will be done to test the functionality and compatibility.

CONTENTS

List of Figures

List of Table

List of Equation

List of Abbreviation

BPB	Branch Prediction Buffer
BTB	Branch Target Buffer
CPU	Central Processing Unit
EX	Execute
FSM	Finite State Machine
ID	Instruction Decode
IF	Instruction Fetch
MIPS	Microprocessor without Interlocked Pipelined Stages
PC	Program Counter
RISC	Reduced Instructions Set Computer
RISC32	RISC 32-bit
RTL	Register Transfer Level
VHDL	VHSIC Hardware Description Language

Project Title

32-bit 5-stage RISC Pipeline Processor with 2-Bit Dynamic Branch Prediction Functionality

Chapter 1: Introduction

1.1 RISC

RISC(*Reduced Instruction Set Computer*) is a philosophy introduced by Cocke IBM, Patterson, Hennessy at 1980s [2]. The main idea of this philosophy is to keep the instruction set small with highly-optimized instructions and simple through fixed instruction length(*eg:32-bit*), limiting the number of addressing modes and operations to make the hardware simpler hence easy to build and test as compared to the other philosophy that insists of a more specialized set of instructions like **CISC**(*Complex Instruction Set Computer*) philosophy. In RISC, the software will do complicated operations by separating the operations into several simple instructions instead of adding in more complex hardware to perform the operation in one instruction.

The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s.

Example of processors implemented based on RISC philosophy are SUN's, UltraSparc, ARM's ARM11, Motorola's PowerPC, MIPS etc. We will focused on MIPS(*Microprocessor without interlocked Pipelined Stages*) throughout this project.

1.2 MIPS

MIPS, which stand for Microprocessor without interlocked Pipelined Stages is computer architecture developed by David A. Patterson based on the RISC philosophy [3]. In the early 80s, Professor John L. Hennessy started the development of MIPS processor with his graduate students in Standford University [3]. The concept is to create a faster processor by using simple instruction with great compiler for instruction scheduling and also pipelining the hardware so that each stages of the hardware can run independently on different instructions at the same time to fully utilize the processor time. The project is done in the year of 1984 [3].

1.3 Pipeline

Pipeline here refer to a set of registers that insert between the hardware processor to divide them into different stages. These stages are Instruction Fetch, Instruction Decode & Operand Fetch, Execute, Memory access and Write Back.

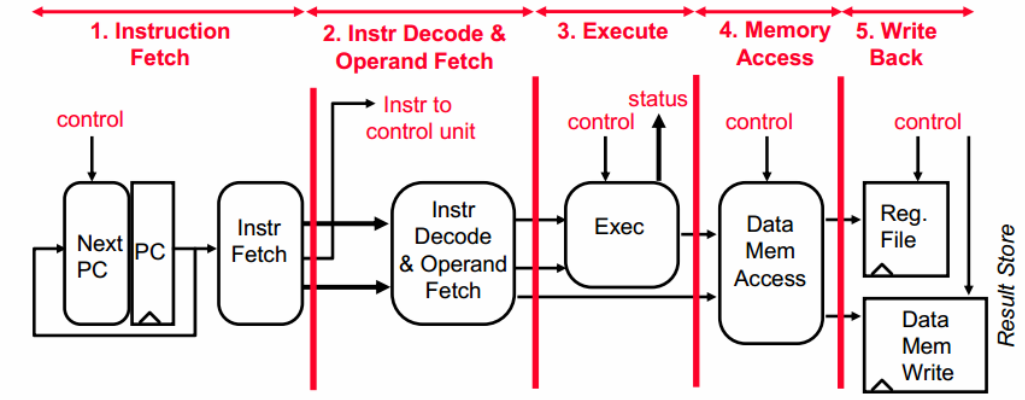


Figure 1 Different stages of an instruction execution

The input of the pipeline register in the current stage is the output to the next stage and all the pipeline registers are clocked synchronously. This help in preventing loss of information of an instruction as it go from one stage to next.

Pipelining a processor enable the processor to begin executing next instruction before the current one is complete [5]. This enhanced the speed performance of the processor as each stages of the processor are able to operate different parts of different instructions independently at a single clock cycle.

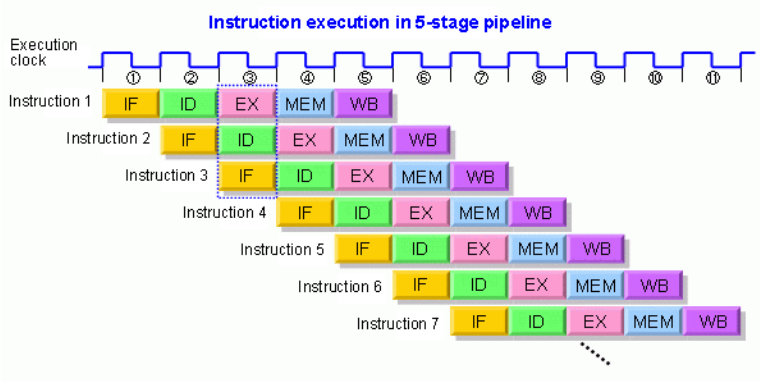


Figure 2 Overlapping instruction execution

1.4 Branch Prediction

Branch prediction is used to enhance the performance of pipelining or superscalar processors especially when the number of stages is large [6]. The branch predictor pre-fetches a limited form of data and attempt to predict the result of branch instructions. The processor will speculatively execute instruction base on the predicted result from the branch predictor. A processor can have a better overall performance especially when the prediction rates are high enough to offset miss-prediction penalties as up to 20 percent of instructions are branches [10]. Without branch prediction, a processor must stall whenever there are unresolved branch instructions [10].

1.5 Motivation

A 32-bit 5-stage pipeline RISC soft-core can be advantageous in creating a core–based environment to assist research and development work in the area of developing Intellectual Properties (IP) cores. However, there are limitations in obtaining such workable core-based design environment.

Microchip design companies develop microprocessors cores as IP for commercial purposes. The microprocessor IP includes information on the entire design process for the front – end (modelling and verification) and back – end (layout and physical design) IC design. These are trade secrets of a company and certainly not made available in the market at an affordable price for research purposes.

Several freely available microprocessor cores are freely available from source such as the miniMIPS (www.opencores.org), the PH processor (Leicester University), uCore, Yellow Star (Manchester University), etc. Unfortunately, these processors do not implement the entire MIPS Instruction Set Architecture (ISA) and lack of comprehensive documentation. This makes them unsuitable for reuse and customization.

Verification is vital for proving the functionality of any digital design. The microprocessor cores mentioned above are handicapped by incomplete and poorly developed verification specifications. This hampers the verification process, slowing down the overall design process.

The lack of well – developed verification specifications for these microprocessor cores will inevitably affect the physical design phase. A design needs to be functionally proven before the physical design can proceed smoothly. Otherwise, if the front – end design needs to be changed, the physical process also needs to be redone.

1.6 Problem Background

Pipelining can improve the performance of the processor by increase the throughput, the average instructions completed per clock cycle. However, it also introduced hazards to the processor.

The hazard can be categorizes three different types: structural hazard, data hazard and control hazard. This project will only focus on control hazard.

Control hazard, which is also known as branching hazard often take place when there is a branch occur. The problem arises when the branch is taken, the program flow will be incorrect due to the instructions that should not be executed had been fetched into the ID stage of the pipelined processor before the branch condition is evaluated at EX stage.

Control hazard can be caused by the following event:

- Conditional branch: beq, bne, blez, bgtz
- Unconditional branch: j, jr, jal, jalr
- Exception

There are several ways can be used to solve this problem, such as stall the next instruction until the branch is completed, flush all the inappropriate stages in the pipeline or through hardware implementation.

In this project, a branch predictor is aimed to integrate into the pipeline processor to predict both the conditional and unconditional branch dynamically based on the information stored in the BTB. The BTB is a small cache memory inside the branch predictor that used to record and update previous information of different branch instruction.

1.7 Problem Statement

Based on the ongoing project that has been developing in the Faculty of Information and Communication Technology of Universiti Tunku Abdul Rahman, it is consists of RISC32 processor and branch predictor.

The current problem found:

1. The developed branch predictor is not integrated into the processor.
2. So far, only the beq instruction is tested on the non-integrated branch predictor. Other program control instructions such as bne, blez, bgtz, j, jal, jr and jalr are not supported.
3. The evaluation of the predictor's accuracy has not been carried out yet.
4. The accuracy profile of the predictor has not been created and studied.

Chapter 2: Literature Review

2.1 Branch Target Buffer (BTB)

A BTB is a small piece of memory use to gather and store the information related to branch instruction. The BTB behave like a look-up table for the branch predictor to look for information of previous branch instruction that having same tag with the current address to perform branch prediction for the current instruction. If the branch instruction was not found in the BTB, a new entry will be created inside the BTB to store the information of the branch instruction. The information stored included valid bit, tag bit, branch target address, prediction state and LRU bit.

In the previous design of the BTB, cache memory is used as the memory element for the BTB due to the cache has shorter accessing time compare to main memory, which can provide faster read and write operation. The cache memory also consume less power compared to other memory design.

However, this BTB is implemented outside the branch predictor, which introduced latency during branch prediction. A solution to this problem arise is to integrate the BTB into the branch predictor. This will not only minimizes the latency for branch prediction, it also increases the scalability of the processor.

2.1.1 State transition of the BTB

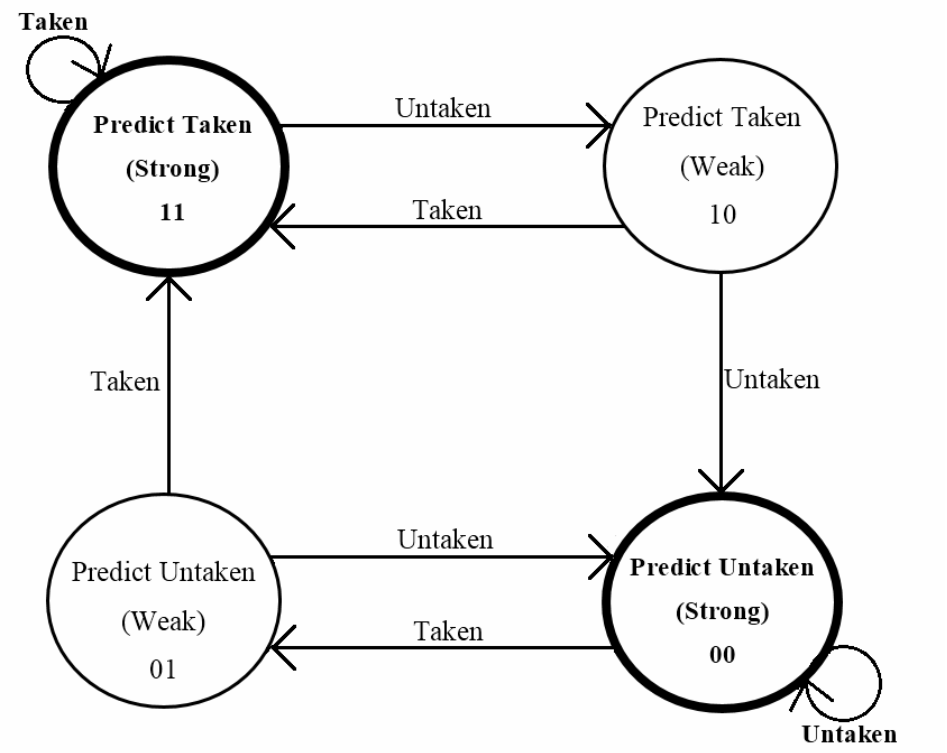


Figure 3 BTB FSM

Figure 3 shown the FSM of the BTB used in this project. A strong state will either remain at the same state when the evaluated condition(taken or untaken) is same at its state, or go to weak state of the same prediction when the evaluated condition is different.

A weak state will not remain at its own state and there is no transition between two weak states. A weak state will go to strong state with same prediction when evaluated condition is same or directly go to a strong state of opposite prediction when the evaluate condition is different with the current state, instead of go to a weak state of opposite prediction.

For example, if the current state is weakly predict untaken, the next state will be strongly predict taken instead of weakly predict taken(opposite prediction of current state). This applied if the current state is weakly predict taken as well.

When a new entry is create in the BTB, it start with state weakly predict taken or weakly predict not taken depend on scenario(*subchapter 2.2.2*).

2.1.2 BTB Cache Associativity

The BTB is implemented using 4-way set-associative mapping technique with 4K or 4096 entries. Each way will have 1k or 1024 entries. This can help to increase the prediction rate as four locations are reserved for the branch instruction with the same index, each data stored can have a chance to stay longer in the buffer with the cost of extra 2 bits LRU array for each entry. The LRU bits are used to indicate which entry within a set to be replaced when required. The LRU bits within the same set(sharing the same index) are updated together during a read or write operation to preserve the relational information. Beside LRU bits, a block of BTB entry also contain 1 valid bit, 20 tag bit, 32-bit branch target address and 2-bit prediction array.

56	55 36	35 4	3 2	1 0
valid	tag[19:0]	br_taddr[31:0]	pred[1:0]	lru[1:0]
1-bit	20-bit	32-bit	2-bit	2-bit

Figure 4 data allocation in one entry of BTB

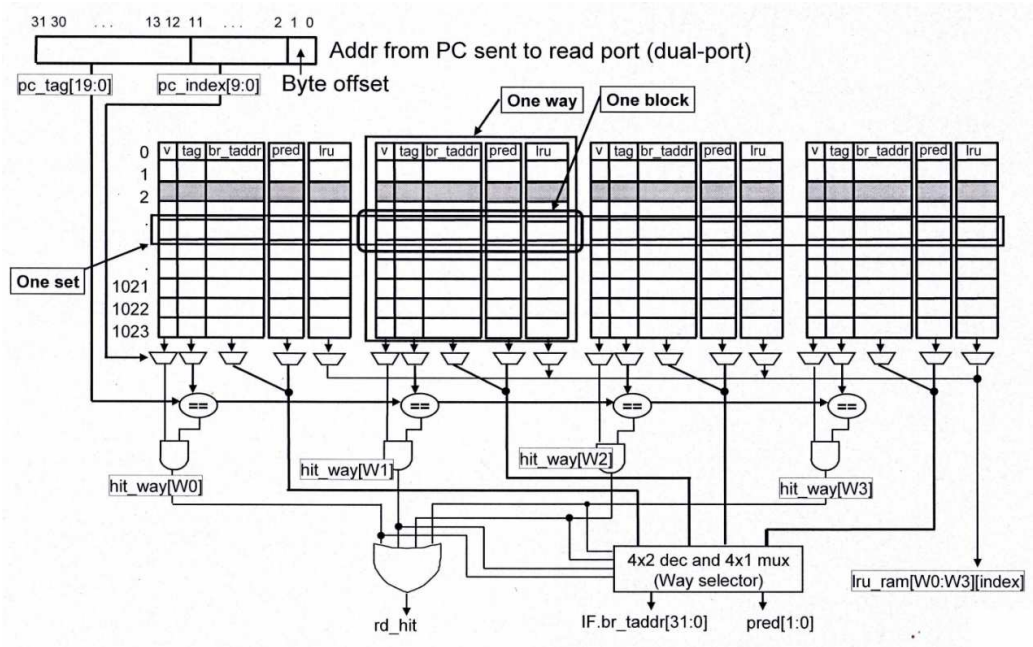


Figure 5 4-way set-associative mapping implementation

2.2 Branch Predictor Block (BPB)

A branch predictor perform dynamic branch prediction with the information store inside the BTB. A branch predictor can help in speed up the processor when prediction predicted correctly especially for pipelined processor with many stages. However, one clock cycle will be wasted to flush away the instruction that has been wrongly fetched into ID stage when prediction predicted wrongly. Hence, it is important to implement a branch predictor with high prediction accuracy to tolerant miss-prediction.

In this project, the branch prediction block is implemented between IF and ID stage of the processor and the BTB will be integrated into the branch predictor to minimize latency and increase overall performance. The BTB is only access for read in IF stage to retrieve information needed for prediction and only access for write in ID stage to update the information.

2.2.2 Flowchart of the BPB

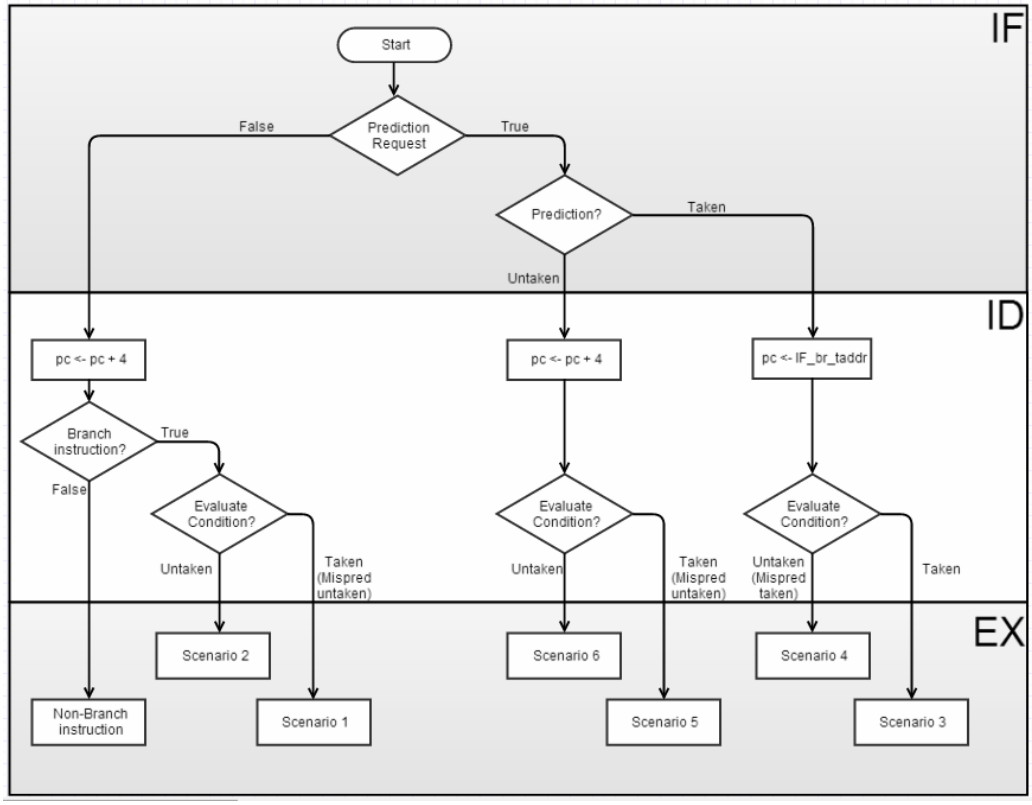


Figure 6 Flowchart of BPB

The flowchart shown in Figure 6 illustrated how the BPB work in IF and ID stage and then evaluated into non-branch instruction or six different scenarios at EX stage. At the beginning of the instruction, the BPB check if there is any data store in the BTB has the same tag with the program counter. If a matched tag is found in the BTB then there is a read hit case else there is a read miss.

Scenario 1 or 2 occurred after a read miss in IF stage. Normal program counter address will be used to fetch an instruction from the instruction memory. The instruction is evaluated to see if it is a branch instruction. If that is not a branch instruction, then that is a normal non-branch instruction. If that is a branch instruction, it will be further evaluate to scenario 1 if branch is taken else scenario 2.

Scenario 3 or 4 occurred after a read hit and predict taken in IF stage. Branch address stored in the BTB is used to fetch instruction due to prediction taken. The prediction

is then evaluated in ID stage and the evaluation will lead to scenario 3 when that is a correct prediction else scenario 4.

Chapter 3: Project Scope and Objectives

3.1 Project Scope

This project is aimed to repartition the internal module of the processor and integrated a 2-bit dynamic branch predictor into the RISC32 processor. Using a branch predictor can increase the throughput and improve the performance of the processor during branch instructions.

The outcome of this project is focus on:

1. Integrate branch predictor into the processor of the RISC32.
2. Provide a well develop test program to test the initial working of the branch predictor.
3. To characterize how well the branch predictor work.

3.2 Project Objective

The objectives of the project include:

1. Analyze the existing branch predictor with 2-bit BTB.
2. Integrate the branch predictor into the pipeline processor.
3. Develop the control for the branch predictor using RTL Modeling.
4. To develop the test plan and test program to verify the integration of the branch predictor.
5. To develop test program to characterize the accuracy of the branch predictor.
6. Analyze the performance of the CPU, and the branch predictor after integration.

Chapter 4: Methodology

4.1 Design Methodology

Basically, design methodology refer to the method used in development of a system. It provides a set of guideline that leading to complete and success of a design work.

Design methodologies ensure the following:

- Correct functionality.
- Satisfaction of performance and power goals.
- Catching bugs early
- Good documentation

This project will be implemented using Top-down design methodology.

This methodology revises the overall system design from the end of solution backwards to the smallest part of the design and makes changes along the way. This methodology provides advantages in functionality, performance, power consumption and area of silicon.

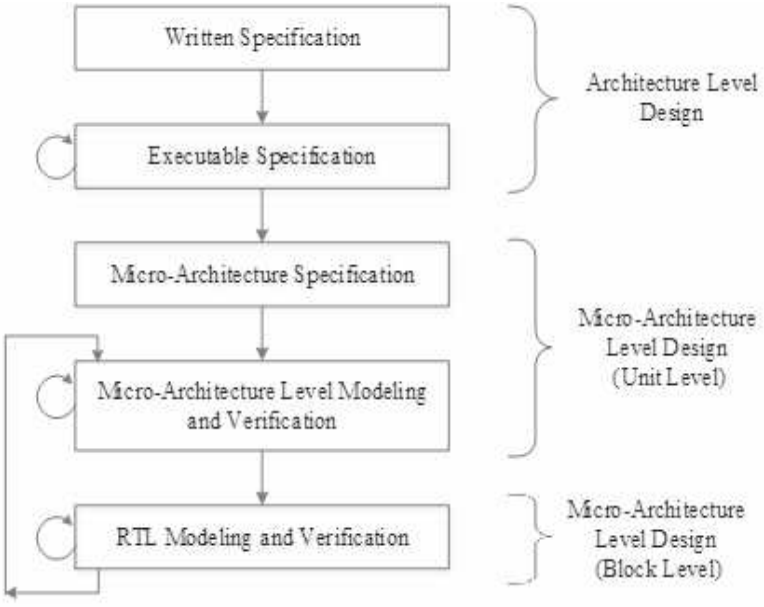


Figure 7 General Design Flow without Logic Synthesis and Physical Design

4.1.1 Architecture Level Design

The specification is developed in written specification and executable specification.

- Written Specification: Function, performance, time and cost of design is written in English. Function specification, verification specification, development plan and packaging specification are included as well.

- Executable Specification: High level programming language such as Verilog or VHDL is used to program according to the design features and functionalities.

4.1.2 Micro-Architecture Level Design (unit level)

There are two phases included in this level of design: Micro-architecture Specification and Micro-architecture Level Modeling & Verification. The designed system is divided into several units to carry different functions with each of the unit describe the algorithm and data flow of the system.

4.1.3 Micro-Architecture Level Design (block level)

RTL Modeling & Verification RTL Modeling & Verification is implemented. In this level of design, each unit designed in previous level is further divided into smaller partition named block to describe the interior work of the units and reduce the complexity of design.

The RTL modeling is done with the following information:

- Overview of functional description
- I/O pin description
- Function table
- Finite-state machine (FSM) & Algorithm-state machine (ASM)
- Test plan

After the development of Micro-architecture specification is completed, the RTL modeling with programming language can be started. Model can be simulate and synthesize with software while verification also can be done.

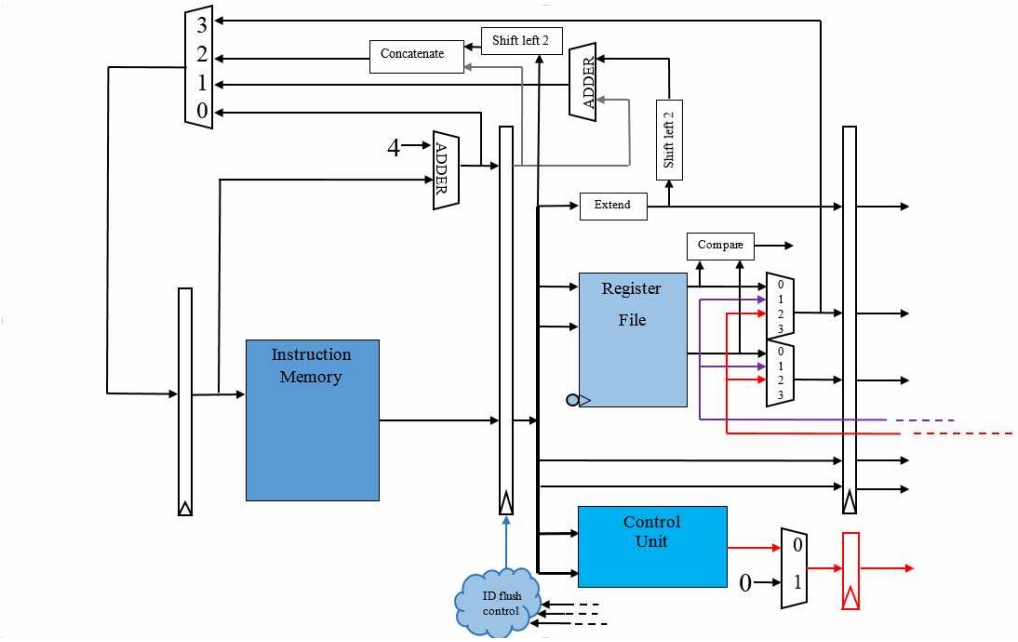
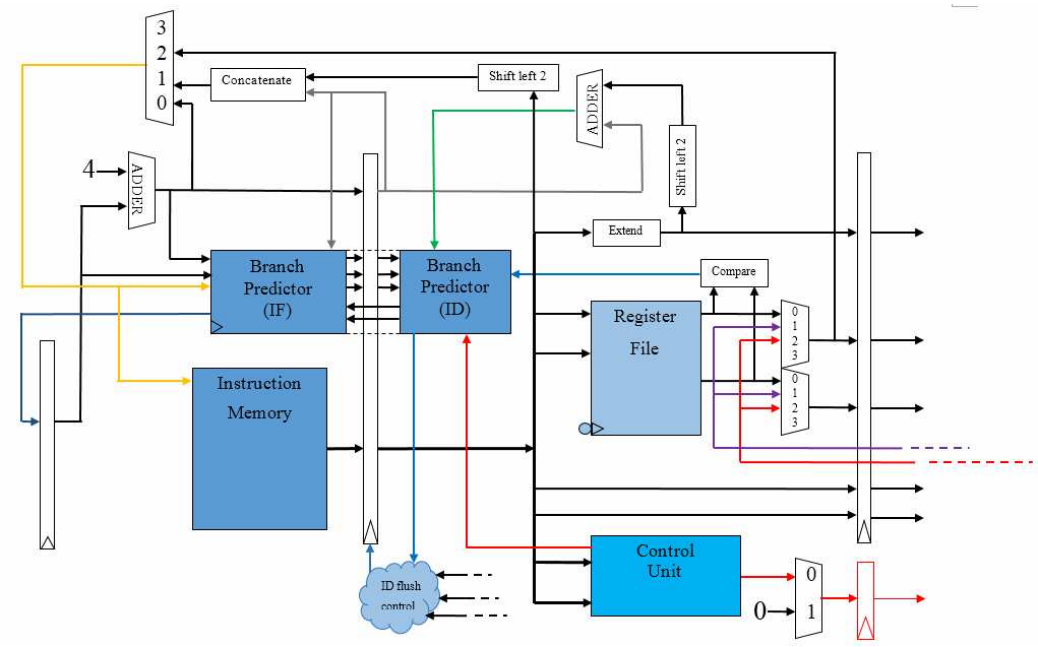


Figure 8 IF-ID stage of a pipeline processor without branch predictor



The integration of the branch predictor involves:

1. The development of the control logic.
2. The modification of the instruction address path.

4.2 Design Language

In this project, Verilog is used as the design language. Verilog is a type of HDL(Hardware Description Language) that is standardized as IEEE 1364, used to model electronic systems. It is commonly used to model and verify hardware at the RTL.
There is another famous type of HDL named VHDL standardized as IEEE 1076 but VHDL is not chosen for this project as it is more complex and Verilog is more readable for user.

4.3 Design Tools

Design tools is required in order to simulate and verify the RTL model.

The verification includes:

- development of test plan
- timing requirement
- system functionality

Since Verilog is used in this project, Software tools that support Verilog HDL and provide simulation environment to verify the functional behaviors and timing design is required. There are a lot of HDL simulators available on the market which all of them have their own advantages and disadvantages.

Some researches had been done and the top three HDL simulators with the highest rating and qualified for application-specific integrated circuit (ASIC) (validation) sign-off at nearly all semiconductor fabrications which are also well-known as the 'big 3' simulator:

- Incisive Enterprise Simulator by Cadence Design Systems
- ModelSim by Mentor Graphic
- Verilog Compiled code Simulator (VCS) by Synopsys

The *Modelsim SE 10.3a* by Mentor Graphic is chosen to be the simulation tools in this project. This is because it met the requirement of the project and it also provided student edition for free compared to other simulation tools that required license. The price for the license can cost up to $25,000 which is not affordable by student.

4.4 Project Timeline

4.4.1 Gantt Chart for Project I

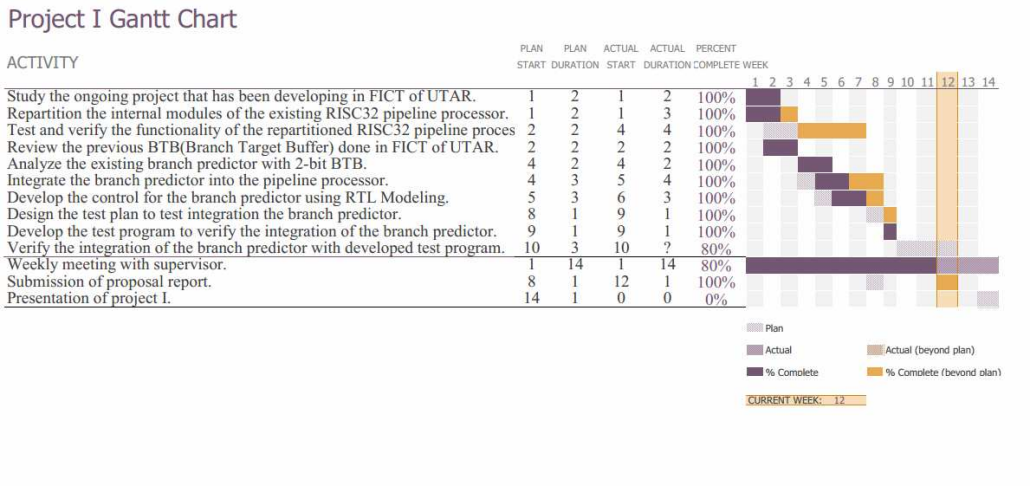


Figure 10 Gantt Chart for Project I

4.4.2 Gantt Chart for Project II

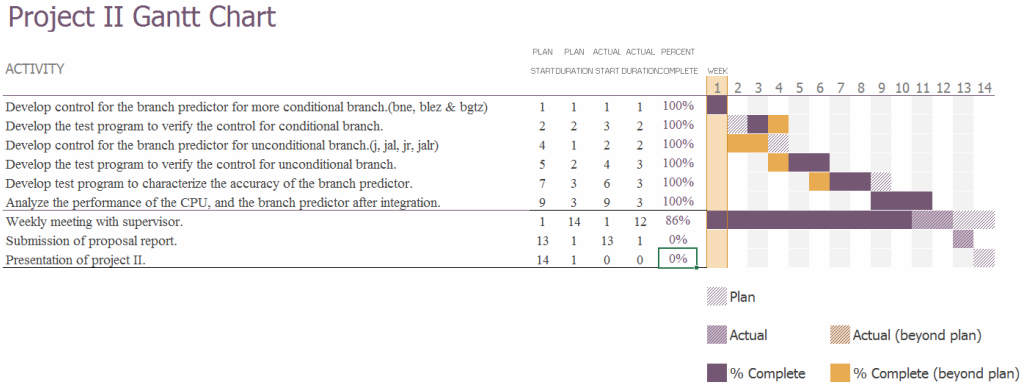


Figure 11 Gantt Chart for Project II

Chapter 5: System Specification

5.1 System Feature

Description	RISC32
Dummy Instruction Cache (KB)	8
Dummy Data Cache (KB)	8
Data width (bits)	32
Instruction width (bits)	32
General Purpose Register	32
Special Purpose Register	HILO, PC
Pipelined Stage	5
Data Hazard Handling	Yes
Structural Hazard Handling	Yes
Control Hazard Handling	Yes
Interlock Handling	Yes
Data Dependency Forwarding	Yes
Branch Prediction	Dynamic – 2bits scheme
Multiplication (size of multiplier and multiplicand)	yes – 32 bits
Branch Delay Slot	Not supported
Instruction supported	37

Table 1 RISC32 features

5.1.1 System Functionality

- Divide execution of instruction into following 5 stages, allowed up to 5 i

  IF(Instruction Fetch) Fetch instruction from instruction cache into the datapath.
  ID(Instruction Decode) Decode instruction and fetch $rs & $rt registers.
  EX(Execute) Execute instruction in the ALB.
  MEM(Memory) Access data cache, load or store.
  WB(Write Back) Write back the result to the register file.

  uctions to run concurrently:

- Resolve data hazards by data forwarding.

- Resolve load-use instructions problem by stalling.

- Resolve structural hazards by separating data and instruction cache.
- Resolve control hazards by branch prediction.
- Perform instructions as listed in 5.3.

5.2 Operating Procedure

- Start the system.
- Porting sequence of instruction into instruction cache.
- Reset the system for at least 2 clocks.
- After the reset, the system will automatically fetch and run the program inside instruction cache.
- Observe the waveform from the development tools (Modelsim).

5.3 Naming Convention

Module – [lvl]_[mod. name]

Instantiation – [lvl]_[abbr. mod. name]

Pin – [lvl][type]_[abbr. mod. name]_[pin name]

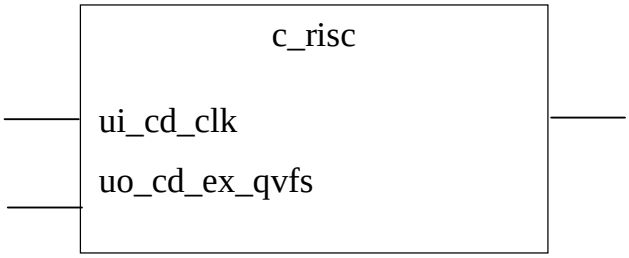
Signal – [type]_[abbr. mod. name]_<stage>_[pin name]

Abbreviation	Description	Case	Available	Remark
lvl	level	lower	c : Chip u : Unit b : Block	
mod. name	Module Name	lower all	any	
abbr. mod. name	Abbreviated module name	lower all	any	maximum 3 characters
Type	Pin type	lower	o : output i : input r : register w: wire	
stage	Stage name	lower all	if, id, ex, mem, wb	Optional
pin name	Pin name	lower all	any	Several word separate by “_”

Table 2 Naming Convention

5.4 RISC32 Pipeline Processor and I/O pin description

5.4.1 Processor interface



5.4.2 I/O pin description

c_risc
Input:
Pin name : ui_cd_clk **Pin Class** : Global **Registered**: Yes **Source->Destination**: External → c_risc **Pin Function**: Provide clock signal for the pipeline processor.
Pin name : ui_cd_rst **Pin Class** : Global **Registered**: Yes **Source->Destination**: External → c_risc **Pin Function**: Provide reset signal for the pipeline processor.
Output:
Pin name : uo_cd_ex_qvfs **Pin Class** : Global **Registered**: Yes **Source->Destination**: c_risc → External **Pin Function**: Overflow signal of the processor.

Table 3 RISC32 I/O pin descriptions

5.5 Memory Map

Purpose	start address	Direction	Segment
Kernel module	0xC000 0000	Up	Kseg2
Boot Rom		Up	Kseg1
i/o register(if below 512MB)	0xA000 0000	Up	
Direct view of memory to 512MB linux kernel code and data		Up	Kseg0
Exception Entry point	0x8000 0000	Up	
Stack	0x7fff ffff	Down	Kuseg
Program heap	0x1000 8000	Up	
Dynamic library code and data	0x1000 0000	Up	
Main program	0x0040 0000	Up	
Reserved	0x0000 0000	Up	

Table 4 Memory Map

Memory map description

Kernel module

- Accessible by kernel*

Boot Rom

- Start up ROM which keep the system configuration*

I/O registers (if below 512MB)

- External IO device register*

Direct view of memory to 512MB linux kernel code and data

- *

Exception Entry point

- Software exception handling *

Stack

- Use for argument passing

Program heap

- Dynamic memory allocation such as malloc()

Dynamic library code and data

- Data segment which is access by

Main program

- Text segment which contain the main program

Reserved

Note *: required CP0

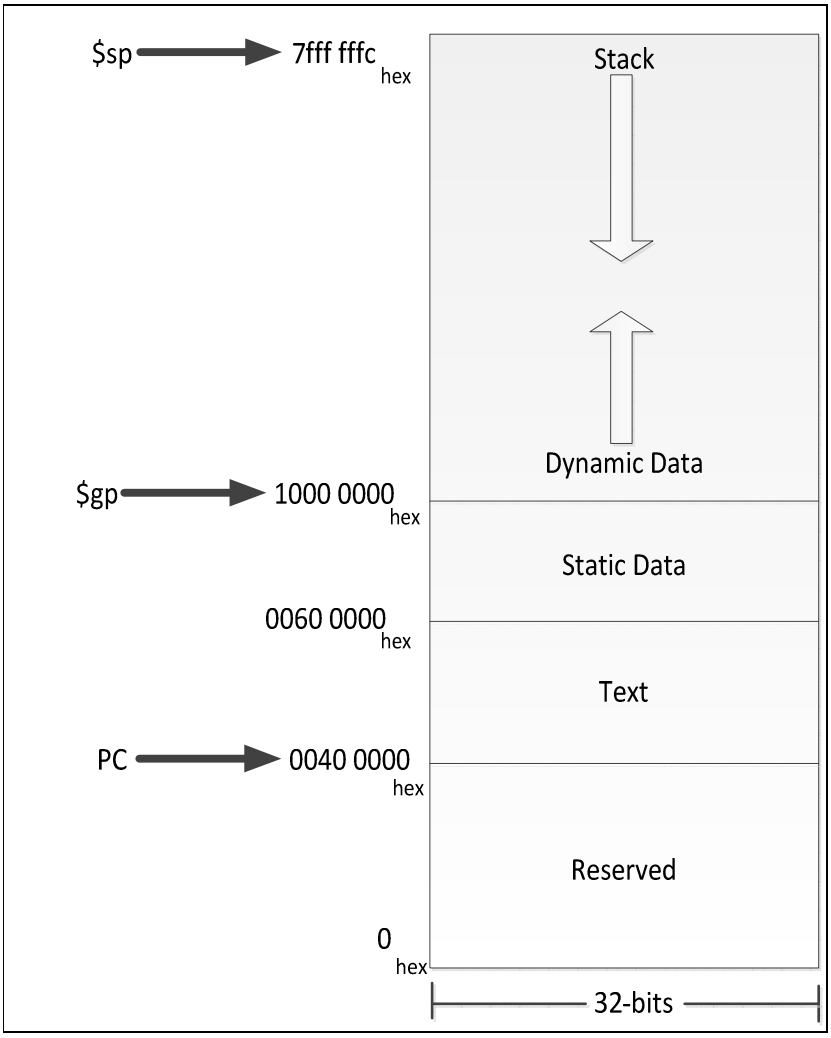


Figure 12 Memory map for Kuseg section, accessible without CP0

5.6 System Register

5.6.1 General Purpose Register

Width : 32-bits

Size : 32 units

Retrieving method : 5-bits address as index

Name	Address	Use	Preserved On Call
$zero	0	Constant Value 0(hardwired)	N.A.
$at	1	Assembler Temporary	No
$v0 - $v1	2 - 3	Value for Function Results and Expression Evaluation	No
$a0 - $a3	4 - 7	Arguments	No
$t0 - $t7	8 – 15	Temporaries	No
$s0 - $s7	16 - 23	Saved temporaries	Yes
$t8 - $t9	24 – 25	Temporaries	No
$k0 - $k1	26 -27	Reserved for OS kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Table 5 General Purpose Register

5.6.2 Special Purpose Register

Width : 32-bits

Size : 2 units

Retrieving method : via instruction: MFHI, MTHI, MFLO, MTLO, MULT or MULTU

Name	Definition	Location in double [64:0]
HI	Most Significant Word	Double [63:32]
LO	Least Significant Word	Double [31:0]

Table 6 Special Purpose Register

5.6.3 Program Counter Register

Width : 32-bits

Size : 1 unit

Retrieving method : Control by instruction address generator control

5.7 Instruction Formats and Addressing modes

5.7.1 Instruction Formats

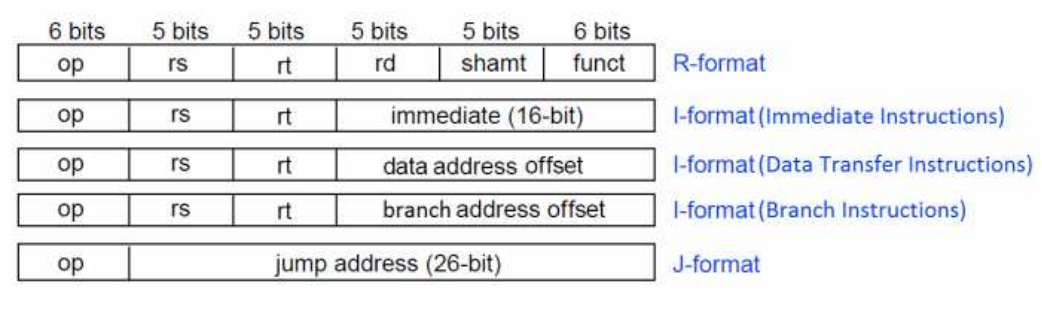


Figure 13 Instruction Format

5.7.1.1 R-format

Register addressing: Perform operation on source and target register and store the result into destination register.

5.7.1.2 I-format

Immediate addressing: Perform operation on source register and immediate and store the result into target register.

Based displacement addressing: Perform operation on source register and immediate, the result is then uses as address to access the data memory to load/store data to/from target register.

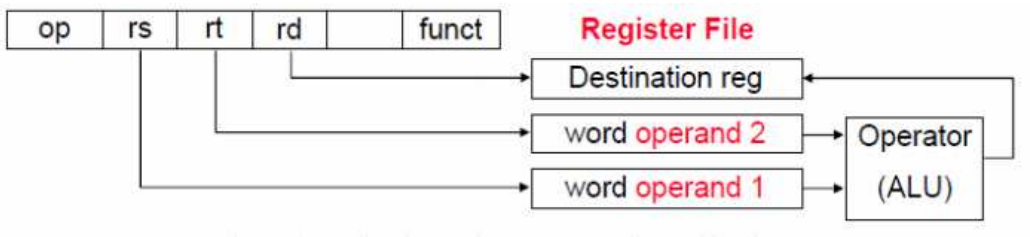
PC-relative addressing: Perform operation on source and target register to determine next PC condition, the immediate is uses as address offset for next PC.

5.7.1.3 J-format

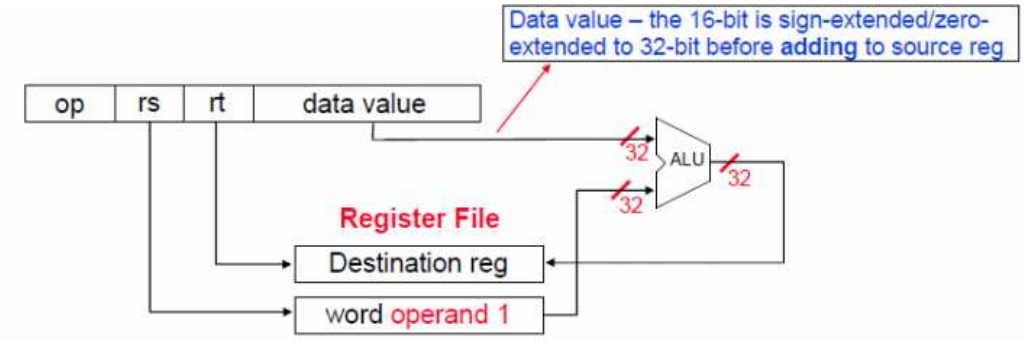
Pseudo-direct addressing: Perform operation by concatenating the upper bits of PC with the jump address.

5.7.2 Addressing modes

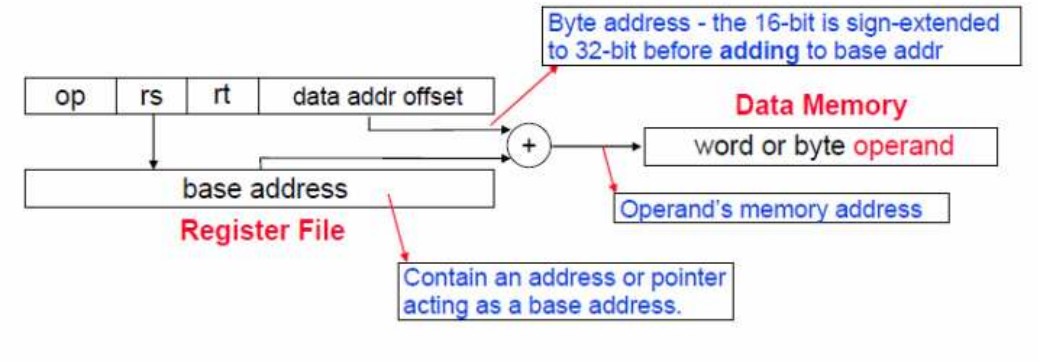
a) Register addressing



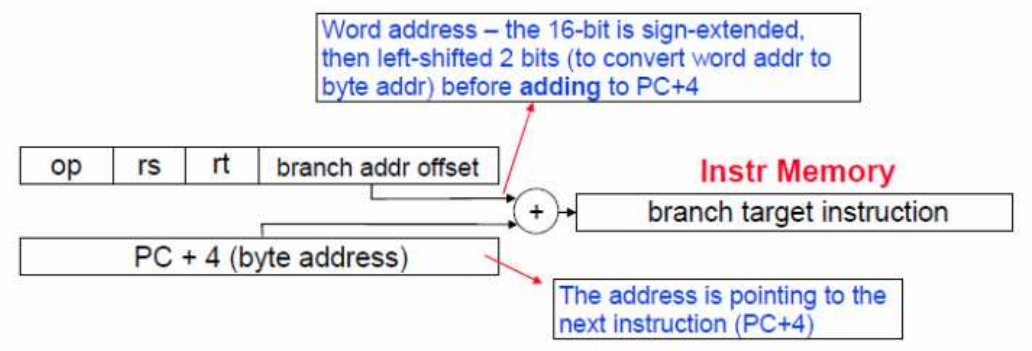
b) Immediate addressing:



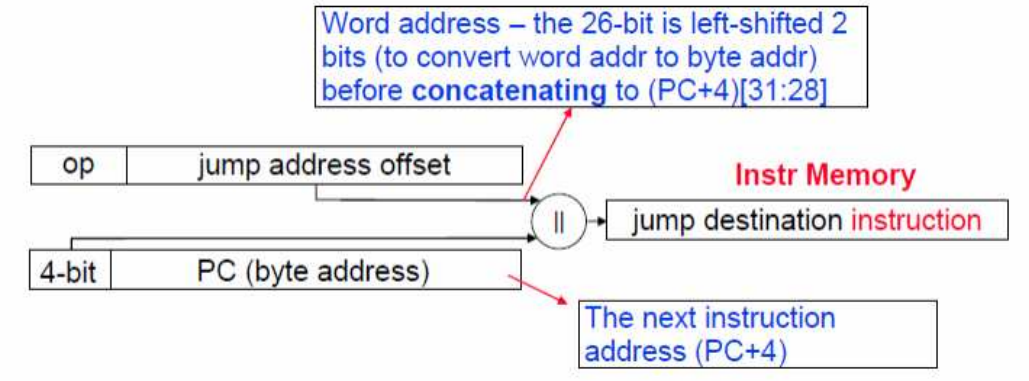
c) Based displacement addressing



d) PC-relative addressing



e)Pseudo-direct addressing



5.8 Supported Instruction set, machine language and RTN

Instruction	Format	Addr. Mode	Machine Language						Register Transfer Notation	Assembly Format	Over flow
			OpCode	Rs	Rt	Rd	Shamt	Func			
sll	R	Register	0x00	0	$rt	$rd	n	0x01	R[rd] =R[rs] << n	sll $rd, $rt, n	no
srl	R	Register	0x00	0	$rt	$rd	n	0x03	R[rd] =R[rs] >> n	srl $rd, $rt, n	no
sra	R	Register	0x00	0	$rt	$rd	n	0x04	R[rd] =R[rs] >>> n	sra $rd, $rt, n	no
jr	R	Register	0x00	$rs	0	0	0	0x0A	PC = R[rs]	jr $rs	no
jalr	R	Register	0x00	$rs	0	0	0	0x0B	PC = R[rs], R[31] = PC + 4	jalr $rs	no
mfhi	R	Register	0x00	0	0	$rd	0	0x10	R[rd] = HI	mfhi $rd	no
mthi	R	Register	0x00	$rs	0	0	0	0x11	HI = R[rs]	mthi $rs	no
mflo	R	Register	0x00	0	0	$rd	0	0x12	R[rd] = LO	mflo $rd	no
mtlo	R	Register	0x00	$rs	0	0	0	0x13	LO = R[rs]	mtlo $rs	no
mult	R	Register	0x00	$rs	$rt	0	0	0x24	HILO = R[rs] * R[rt]	mult $rs, $rt	no
multu	R	Register	0x00	$rs	$rt	0	0	0x24	HILO = U(R[rs]) * U(R[rt])	multu $rs, $rt	no
add	R	Register	0x00	$rs	$rt	$rd	0	0x20	R[rd] = R[rs] + R[rt]	add $rd, $rs, $rt	yes
addu	R	Register	0x00	$rs	$rt	$rd	0	0x21	R[rd] = U(R[rs]) + U(R[rt])	addu $rd, $rs, $rt	no
sub	R	Register	0x00	$rs	$rt	$rd	0	0x22	R[rd] = R[rs] - R[rt]	sub $rd, $rs, $rt	yes
subu	R	Register	0x00	$rs	$rt	$rd	0	0x23	R[rd] = U(R[rs]) - U(R[rt])	subu $rd, $rs, $rt	no
and	R	Register	0x00	$rs	$rt	$rd	0	0x24	R[rd] = R[rs] & R[rt]	and $rd, $rs, $rt	no
or	R	Register	0x00	$rs	$rt	$rd	0	0x25	R[rd] = R[rs] | R[rt]	or $rd, $rs, $rt	no
xor	R	Register	0x00	$rs	$rt	$rd	0	0x26	R[rd] = R[rs] ^ R[rt]	xor $rd, $rs, $rt	no
nor	R	Register	0x00	$rs	$rt	$rd	0	0x27	R[rd] = ~(R[rs] | R[rt])	nor $rd, $rs, $rt	no
slt	R	Register	0x00	$rs	$rt	$rd	0	0x2A	R[rd] = (R[rs] < R[rt]) ? 1 : 0	slt $rd, $rs, $rt	no
sltu	R	Register	0x00	$rs	$rt	$rd	0	0x2	R[rd] = (U(R[rs]) < U(R[rt])) ?	sltu $rd, $rs, $rt	no

								B	1 : 0		
j	J	Pseudo-Direct	0x02	JumpAddr (Label)					PC = {(PC+4) [31:28], JumpAddr, 2'b00}	j label	no
jal	J	Pseudo-Direct	0x03	JumpAddr (Label)					PC = {(PC+4) [31:28], JumpAddr, 2'b00} R[31] = PC + 4	jal label	no
beq	I	PC-Relative	0x04	$rs	$rt	BranchAddr (Label)			PC = (R[rs] == R[rt]) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	beq $rs, $rt, label	no
bne	I	PC-Relative	0x05	$rs	$rt	BranchAddr (Label)			PC = (R[rs] != R[rt]) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	bne $rs, $rt, label	no
blez	I	PC-Relative	0x06	$rs	0	BranchAddr (Label)			PC = (R[rs] <=0) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	blez $rs, $rt, label	no
bgtz	I	PC-Relative	0x07	$rs	0	BranchAddr (Label)			PC = (R[rs] > 0) ? (PC + 4 + (SE(BranchAddr)<<2)) : (PC + 4)	bgtz $rs, $rt, label	no
addi	I	Immediate	0x08	$rs	$rt	Imm			R[rt] = R[rs] + SE(Imm)	addi $rt, $rs, imm	yes
addiu	I	Immediate	0x09	$rs	$rt	Imm			R[rt] = U(R[rs]) + U(ZE(Imm))	addiu $rt, $rs, imm	no
slti	I	Immediate	0x0A	$rs	$rt	Imm			R[rt] = (R[rs] < SE(Imm)) ? 1 : 0	slti $rt, $rs, imm	no
sltiu	I	Immediate	0x0B	$rs	$rt	Imm			R[rt] = (U(R[rs]) < U(SE(Imm))) ? 1 : 0	sltiu $rt, $rs, imm	no

andi	I	Immediate	0x0C	$rs	$rt	Imm	R[rt] = R[rs] & ZE(Imm)	andi $rt, $rs, imm	no
ori	I	Immediate	0x0D	$rs	$rt	Imm	R[rt] = R[rs] | ZE(Imm)	ori $rt, $rs, imm	no
xori	I	Immediate	0x0E	$rs	$rt	Imm	R[rt] = R[rs] ^ ZE(Imm)	xori $rt, $rs, imm	no
lui	I	Immediate	0x0F	$rs	$rt	Imm	R[rt] = Imm << 16	lui $rt, imm	no
lw	I	Based-Displacement	0x23	$rs	$rt	Imm	R[rt] = MEM[R[rs] + SE(Imm)]	lw $rt, imm($rs)	no
sw	I	Based-Displacement	0x2B	$rs	$rt	Imm	MEM[R[rs] + SE(Imm)] = R[rt]	sw $rt, imm($rs)	no

Table 7 Supported Instruction set, machine language and description

Chapter 6: Micro Architecture Specification

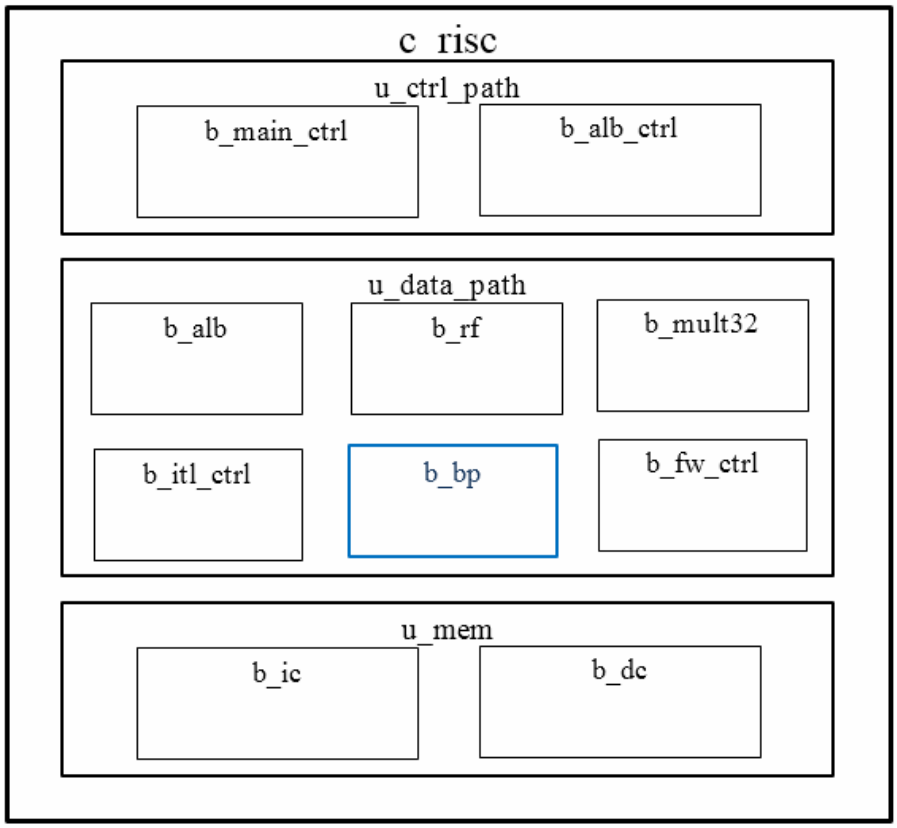
6.1 Design Hierarchy and Partitioning

Design Hierarchy:

Chip Partitioning (Top Level) at Architecture Level	Unit Partitioning at Micro-Architecture Level	Block and Functional Block Partitioning at RTL (Micro-Architecture Level)
RISC32 Pipeline Processor (c_risc)	Datapath (u_dp)	Branch Predictor (b_bp_4way)
		Register File (b_rf)
		Interlock Control (b_itl_ctrl)
		Forward Control (b_fw_ctrl)
		32-bit Multiplier (b_mult32)
		ALB (b_alb)
	Controlpath (u_cp)	Main Control (b_main_ctrl)
		ALB Control (b_alb_ctrl)
	Memory (u_mem)	Instruction Cache (b_ic)
		Data Cache (b_dc)

Table 8 Design Hierarchy

Design Hierarchy block partitioning:



6.2 Micro-Architecture (block level)

6.2.1 Micro-Architecture without branch predictor

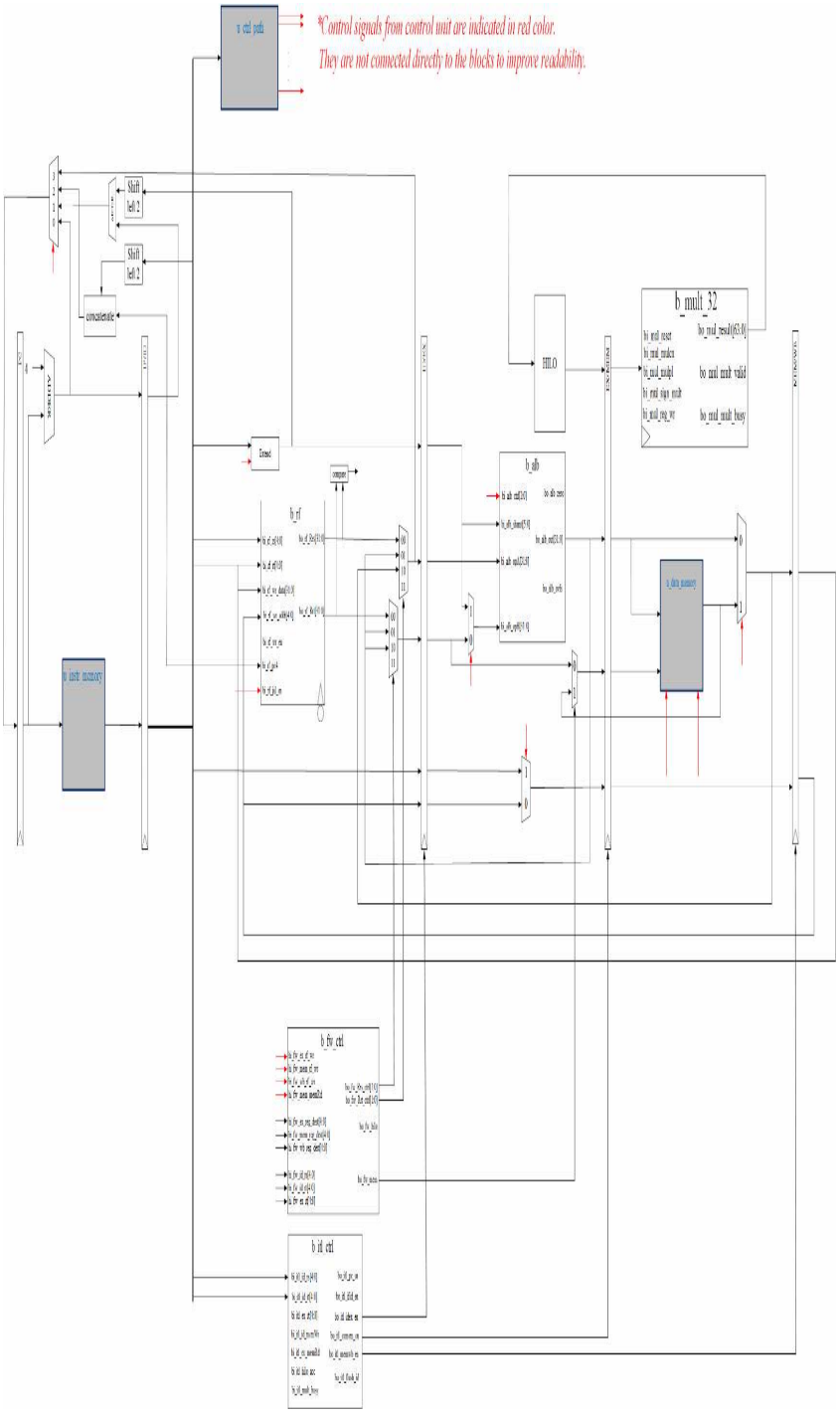


Figure 14 System Micro-Architecture without branch predictor

6.2.2 Micro-Architecture with branch predictor

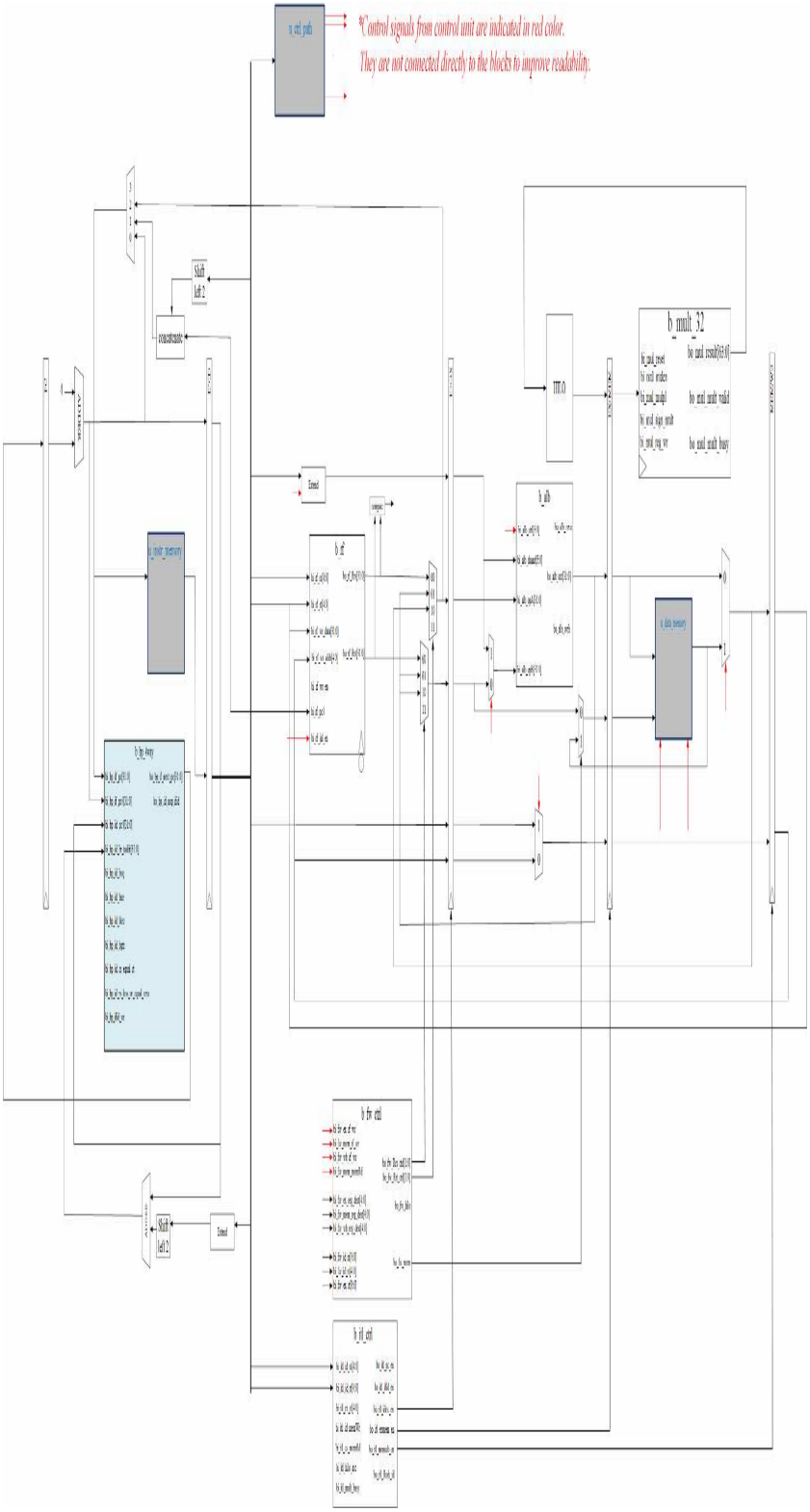


Figure 15 System Micro-Architecture with branch predictor

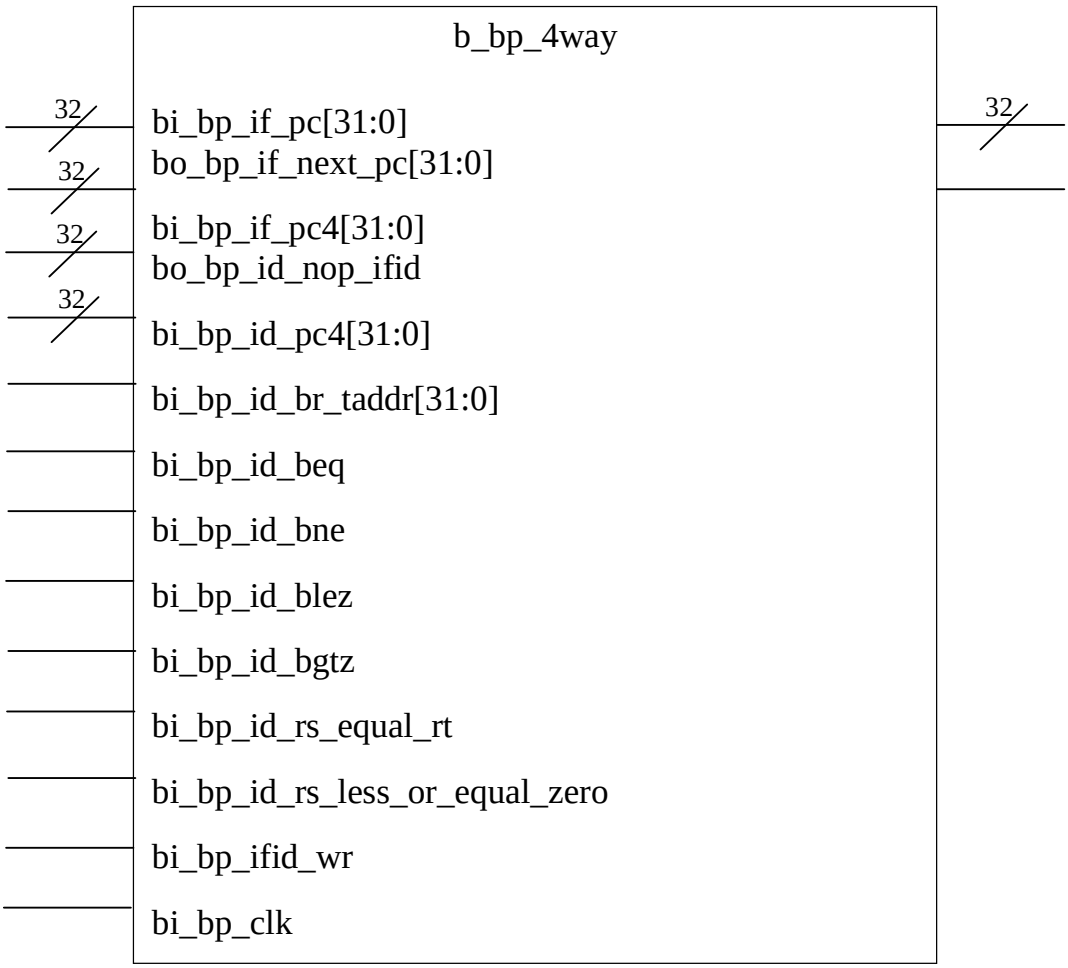
6.3 Branch Predictor Functionality

- Predict next instruction address for the processor with information provided in the branch target buffer.
- Verify predicted address and update the branch target buffer.
- Provide correction address when miss prediction occurred.
- 4 way SRAM with 4k X 57-bits entry as shown in Figure 18.

56	55 36	35 4	3 2	1 0
valid	tag[19:0]	br_taddr[31:0]	pred[1:0]	lru[1:0]
1-bit	20-bit	32-bit	2-bit	2-bit

Figure 16 57-bits entry of SRAM.

6.4 Branch Predictor Interface



6.5 I/O pin description

b_bp_4way
Input:
Pin name : bi_bp_if_pc[31:0] **Pin Class** : Address **Registered**: No **Source->Destination**: Datapath(IF) → b_bp_4way **Pin Function**: Provide index for reading ram and tag for next instruction.
Pin name : bi_bp_if_pc4[31:0] **Pin Class** : Address **Registered**: No **Source->Destination**: Datapath(IF) → b_bp_4way **Pin Function**: Prediction address for case predict untaken.
Pin name : bi_bp_id_pc4[31:0] **Pin Class** : Address **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Correction address for case miss predict taken.
Pin name : bi_bp_id_br_taddr[31:0] **Pin Class** : Address **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Correction address for case miss predict untaken.
Pin name : bi_bp_id_beq **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Indicate current instruction is beq when asserted high.
Pin name : bi_bp_id_bne **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Indicate current instruction is bne when asserted high.
Pin name : bi_bp_id_blez **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Indicate current instruction is blez when asserted high.
Pin name : bi_bp_id_bgtz **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Indicate current instruction is bgtz when asserted high.

Pin name : bi_bp_id_rs_equal_rt **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Use to evaluate the correctness of previous predition for beq, bne.
Pin name : bi_bp_id_rs_less_or_equal_zero **Pin Class** : Data **Registered**: No **Source->Destination**: Datapath(ID) → b_bp_4way **Pin Function**: Use to evaluate the correctness of previous predition for blez, bgtz.
Pin name : bi_bp_ifid_wr **Pin Class** : Control **Registered**: No **Source->Destination**: b_itl_ctrl → b_bp_4way **Pin Function**: Enable signal for the internal ifid pipeline of the branch predictor.
Pin name : bi_bp_clk **Pin Class** : Global **Registered**: No **Source->Destination**: System → b_bp_4way **Pin Function**: Clock signal for the branch predictor.
Pin name : bi_bp_rst **Pin Class** : Global **Registered**: No **Source->Destination**:System → b_bp_4way **Pin Function**: Reset signal for the branch predictor/
Output:
Pin name : bo_bp_if_next_pc[31:0] **Pin Class** : Address **Registered**: Yes **Source->Destination**: b_bp_4way → u_mem → i_ic **Pin Function**: Provide next PC address to fetch next instruction from instruction cache.
Pin name : bo_id_nop_ifid **Pin Class** : Control **Registered**: No **Source->Destination**: b_bp_4way → Datapath(ifid pipeline) **Pin Function**: Insert a nop to the ifid pipeline when miss prediction happened.

Table 9 b_bp_4way I/O pin description

6.6 Internal Block Diagram

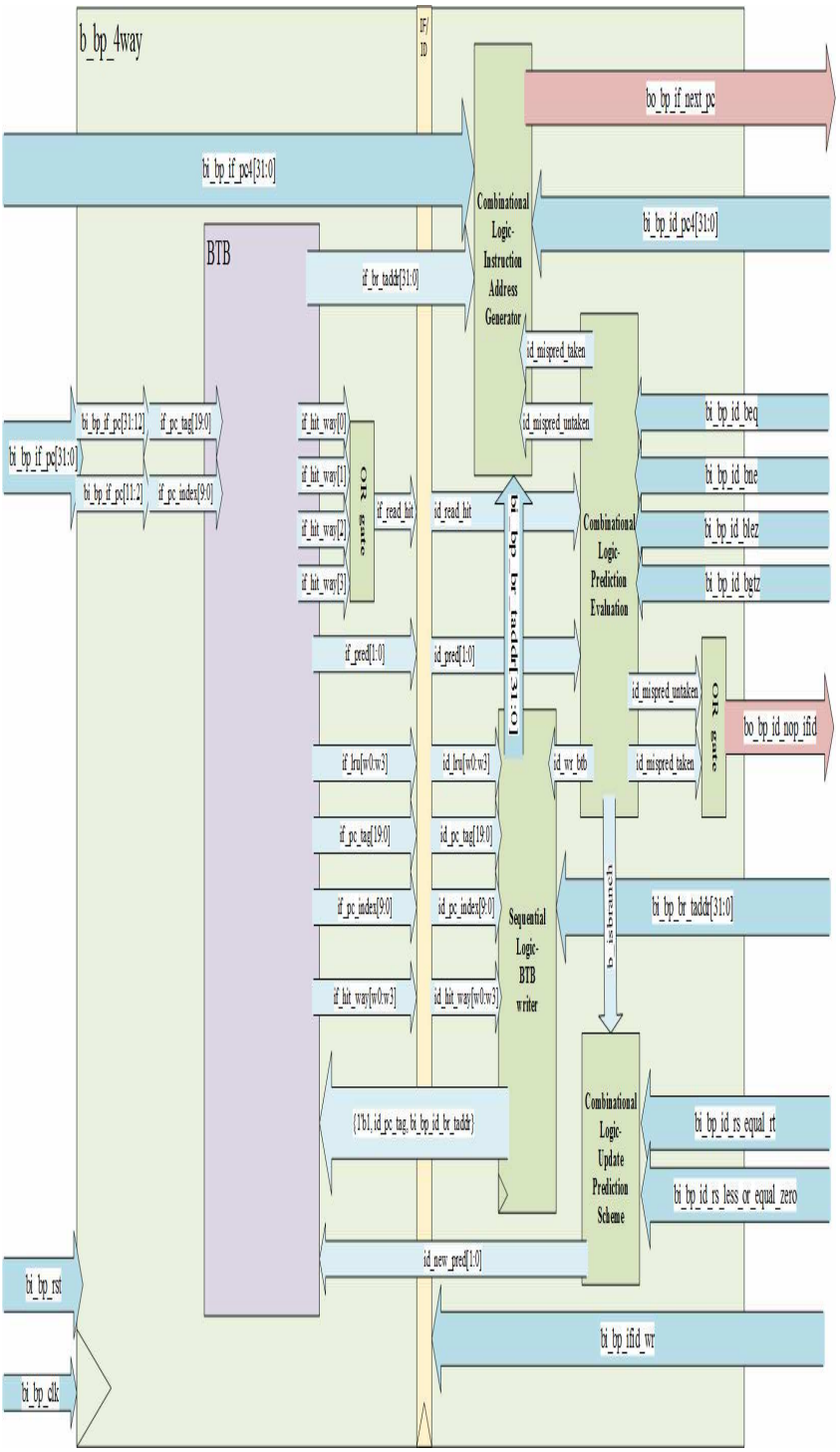


Figure 17 Branch predictor internal block diagram

6.7 Internal Operation

IF stage: BTB read

In IF stage, the branch predictor read its internal branch target buffer to perform next instruction address prediction according to following steps:

1) Look for an entry of the BTB with the same tag and index as the PC.
2) If the specific entry is found, the branch target address store in the entry is assigned as the next instruction address.
3) If the specific entry is not found, the branch predictor will assign the normal PC with an increment of 4 as the instruction address.

ID stage: BTB write

In ID stage, the branch predictor will update the information of the current instruction when the current instruction is a branch instruction in following steps:

1) Evaluate the correctness of the prediction with information get from datapath ID stage.
2) If the prediction is incorrect, the ID stage is flushed.
3) A new entry is created to store the latest information of the instruction when none matching entry is found previously in IF stage or the matching entry in BTB is updated with the latest information.

Chapter 7: Verification

7.1 Instruction Verification Test Program

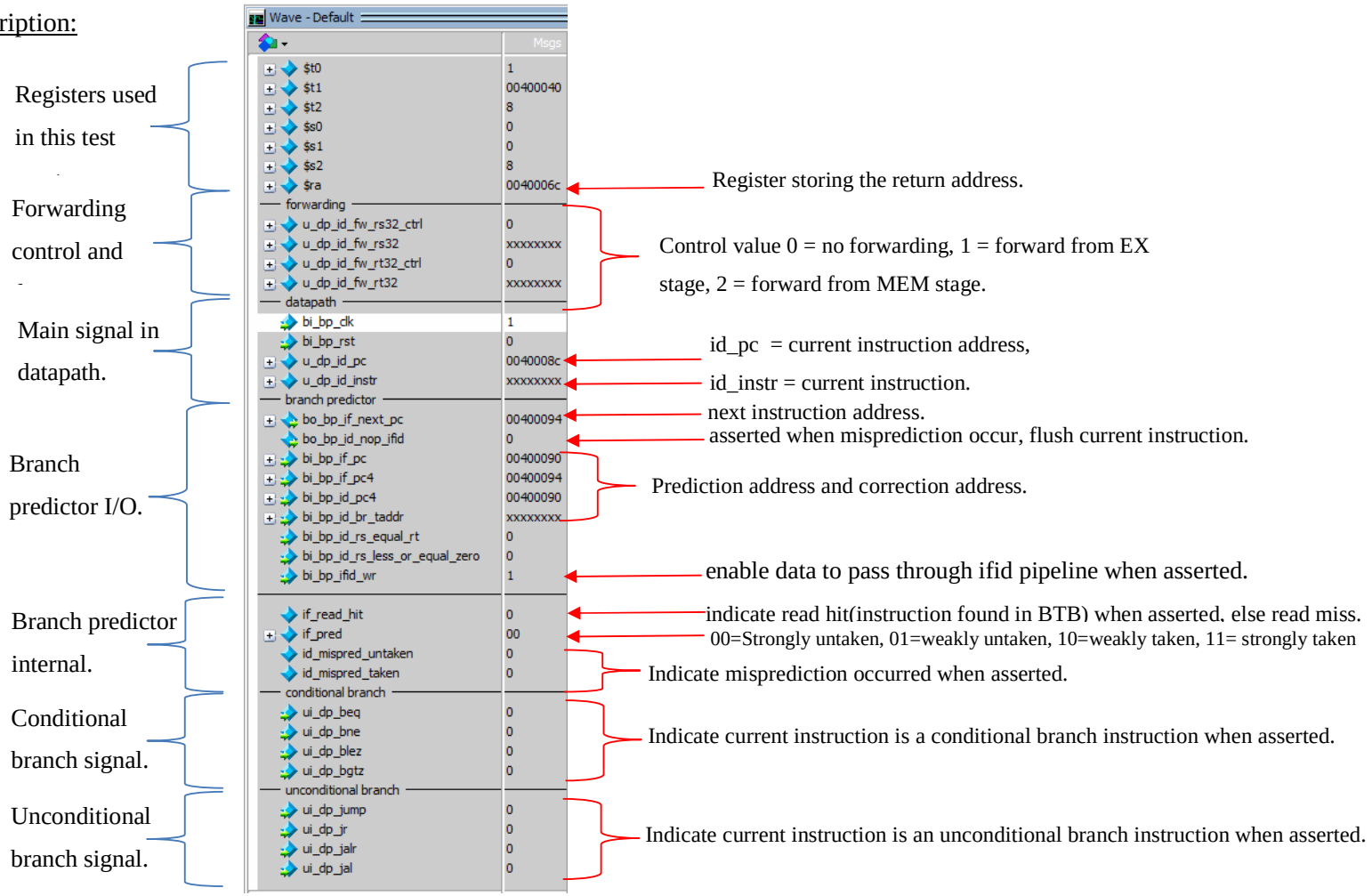
The following program is designed to verify the correctness of both the conditional branch and unconditional branch instructions implemented in this project.

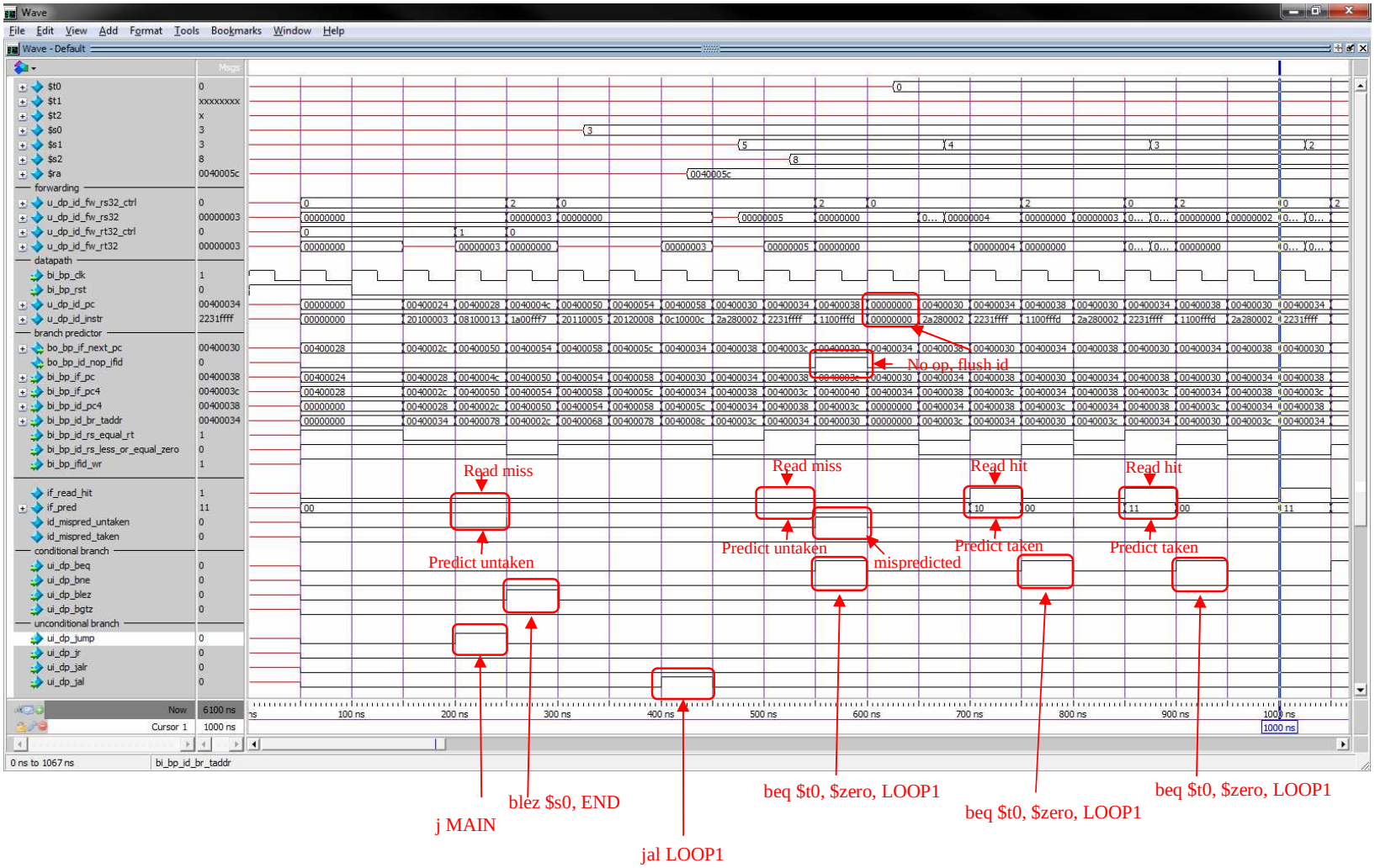
Instruction address	Instruction code	Label	Mnemonic	Operand 1	Operand 2	Operand 3
0x00400024	20100003	INIT:	addi	$s0,	$zero,	3
0x00400028	08100013		j	MAIN		
0x0040002C	0810001D	END:	j	EXIT		
0x00400030	2A280002	LOOP1:	slti	$t0,	$s1,	2
0x00400034	2231FFFF		addi	$s1,	$s1,	-1
0x00400038	1100FFFD		beq	$t0,	$zero,	LOOP1
0x0040003C	03E00008		jr	$ra		
0x00400040	014A5020	LOOP2:	add	$t2,	$t2,	$t2
0x00400044	164AFFFE		bne	$s2,	$t2,	LOOP2
0x00400048	03E00008		jr	$ra		
0x0040004C	1A00FFF7	MAIN:	blez	$s0,	END	
0x00400050	20110005		addi	$s1,	$zero,	5
0x00400054	20120008		addi	$s2,	$zero,	8
0x00400058	0C10000C		jal	LOOP1		
0x0040005C	3C090040		lui	$t1,	0x0040	
0x00400060	35290040		ori	$t1,	$t1,	0x0040
0x00400064	200A0001		addi	$t2,	$zero,	1
0x00400068	01200009		jalr	$t1		
0x0040006C	2210FFFF		addi	$s0,	$s0,	-1
0x00400070	1E00FFF6		bgtz	$s0,	MAIN	
0x00400074	00000000	EXIT:				

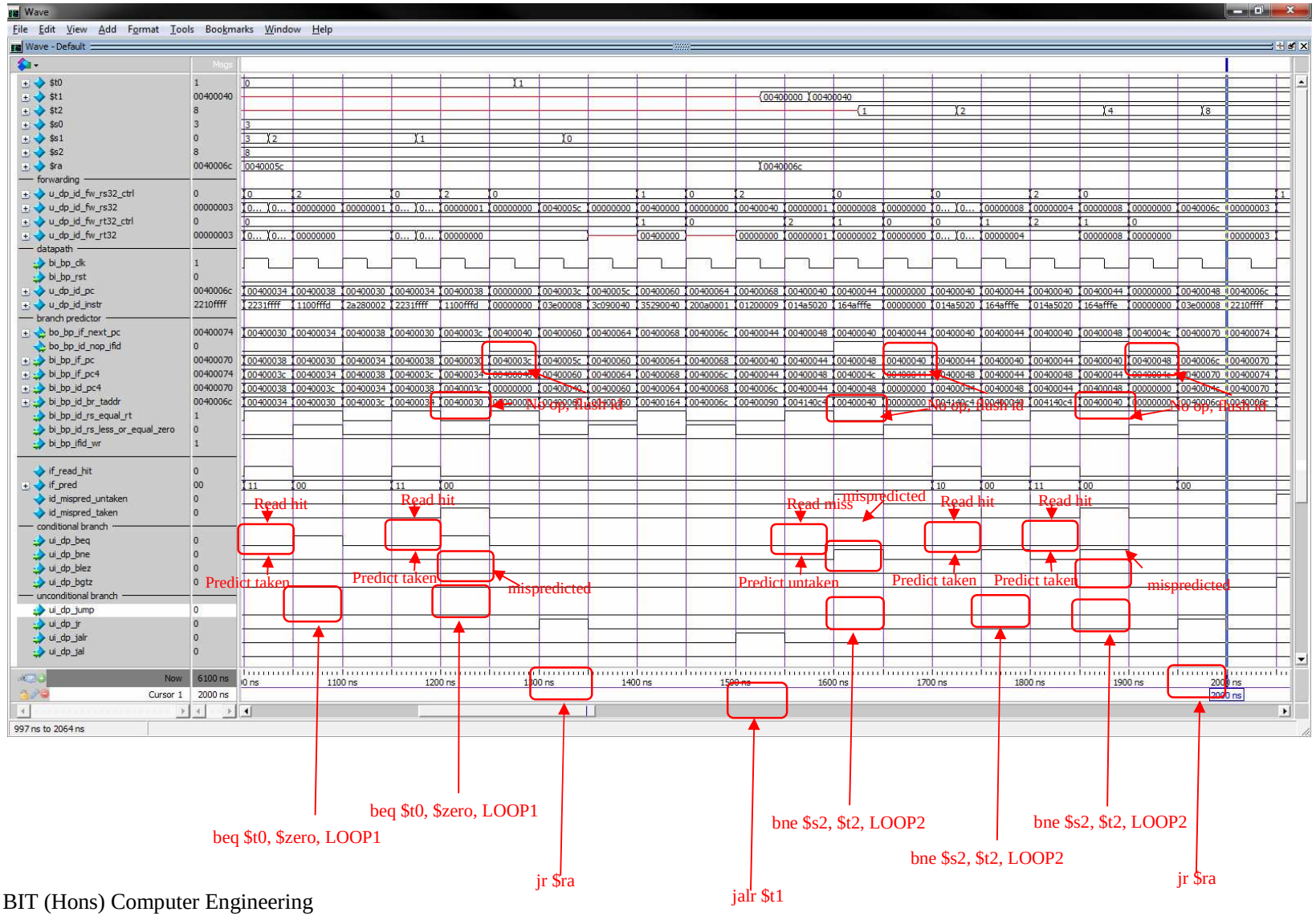
	unconditional branch
	conditional branch

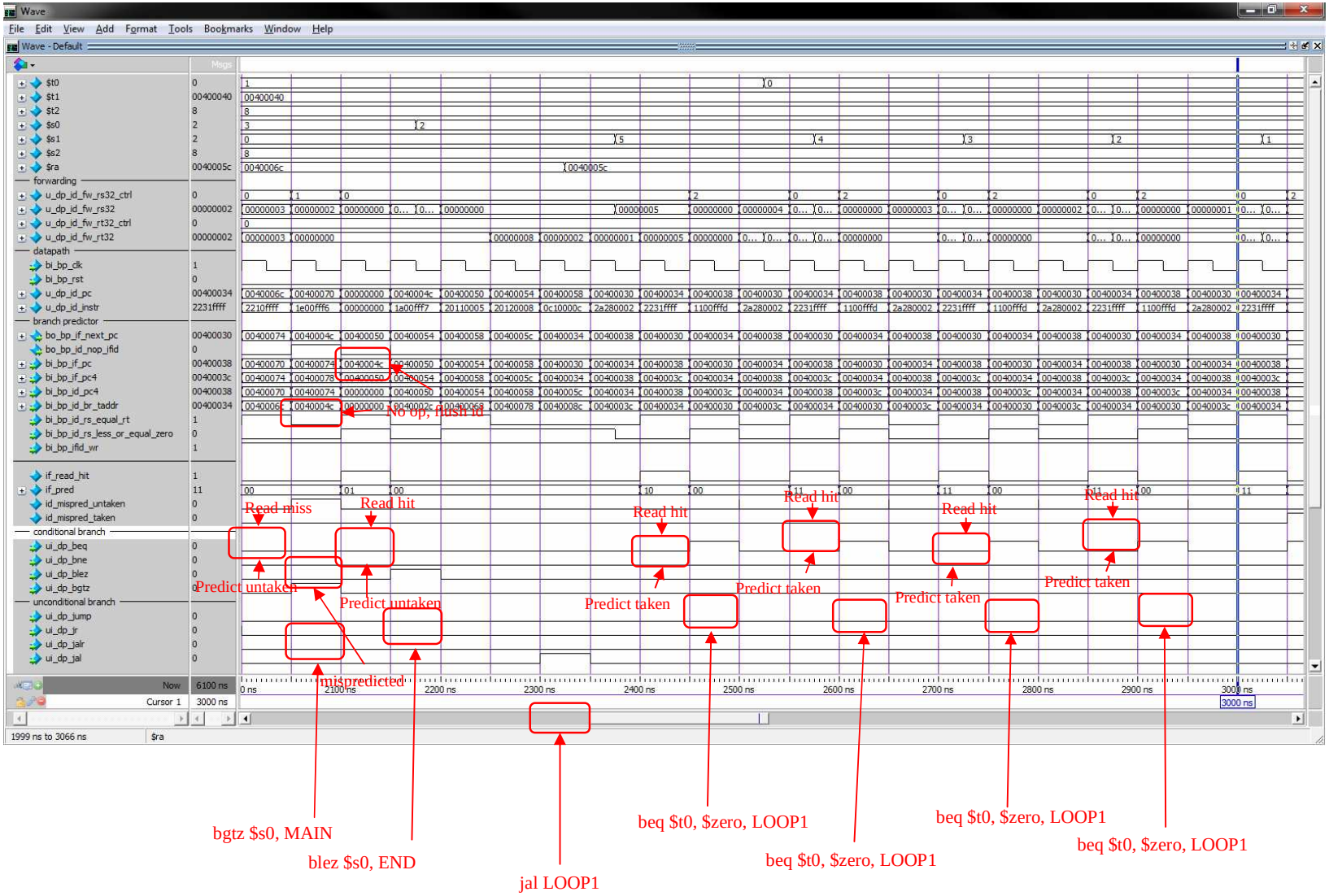
7.1.1 Simulation Result

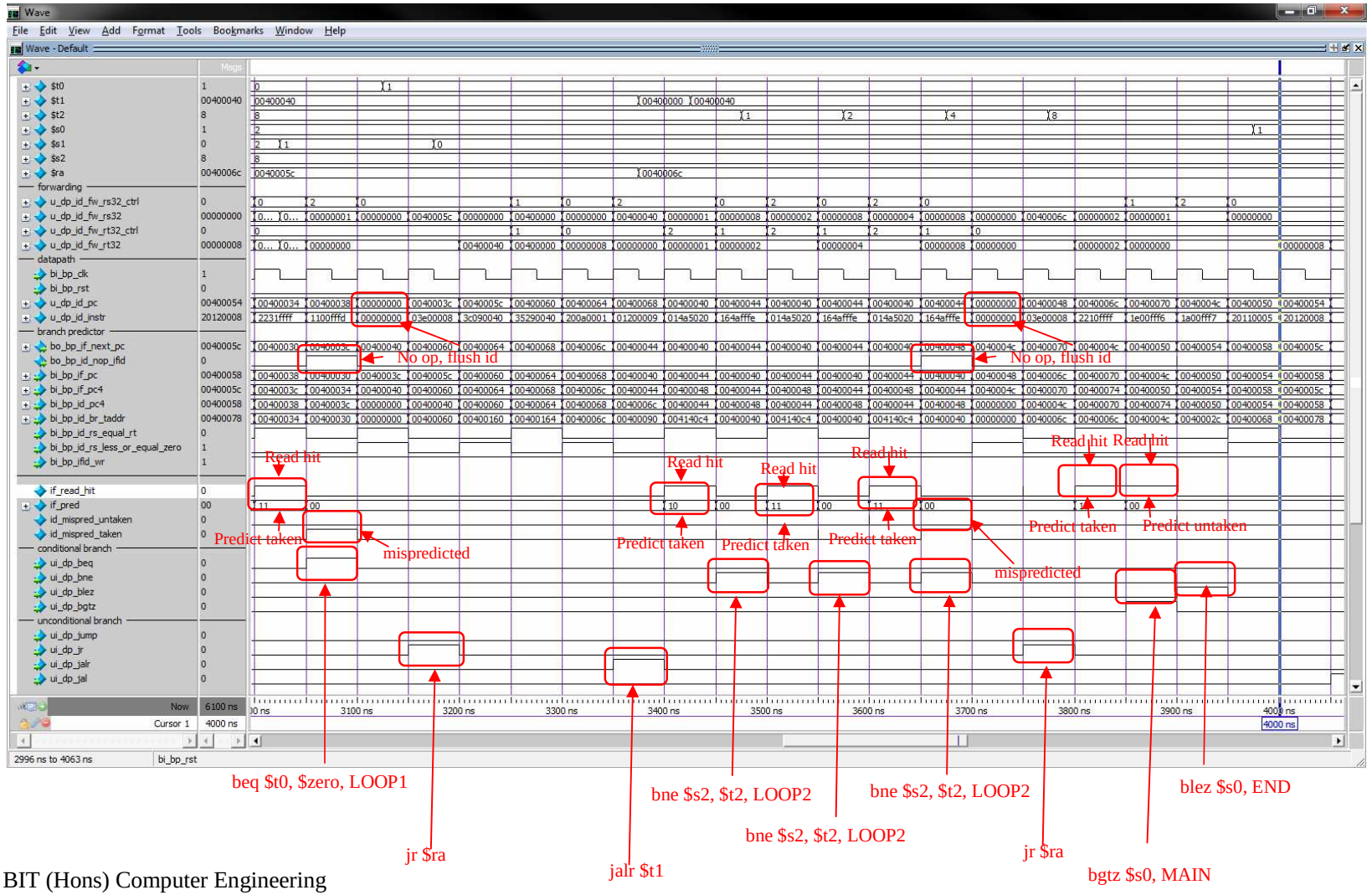
Simulation Description:

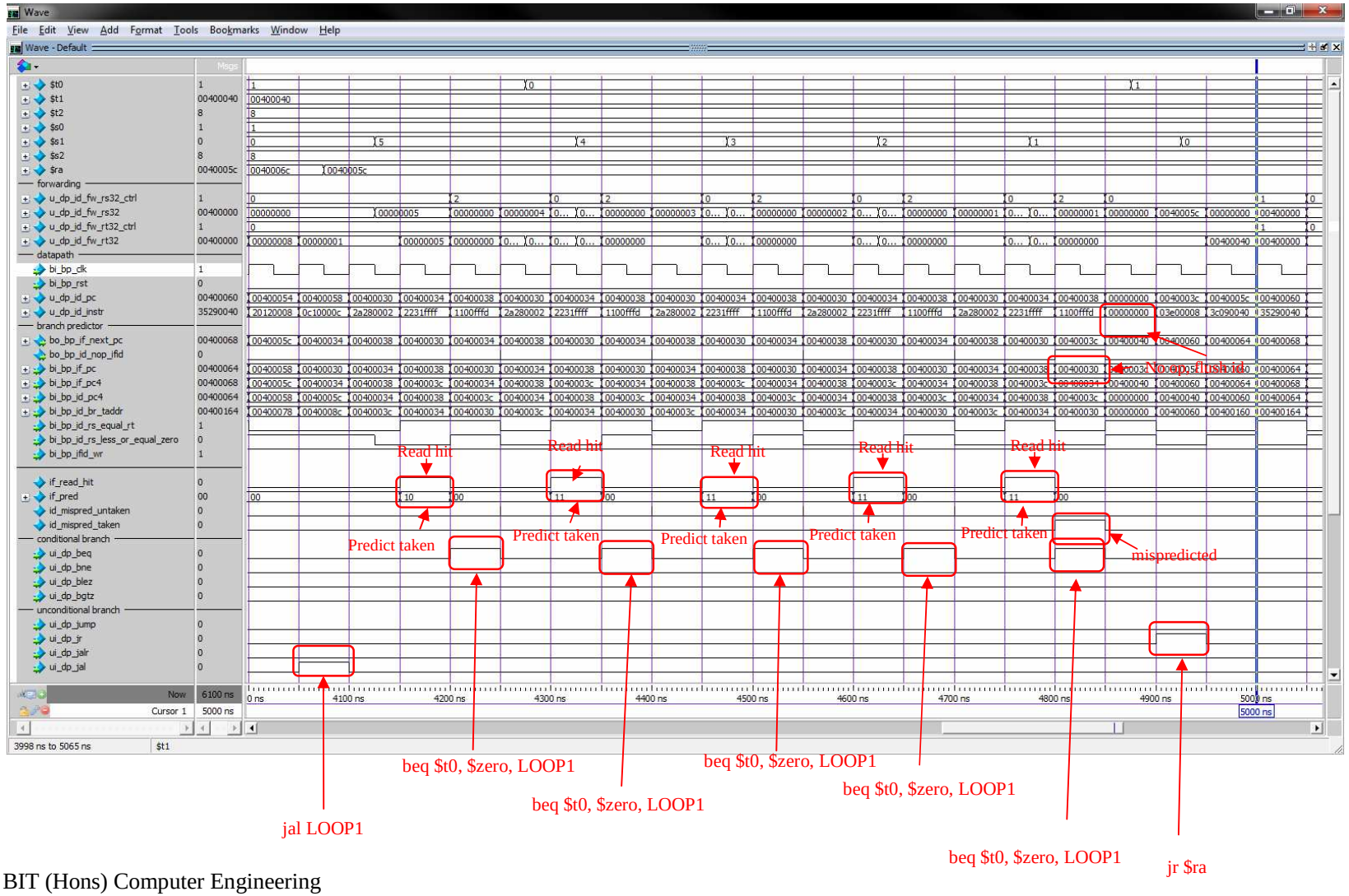


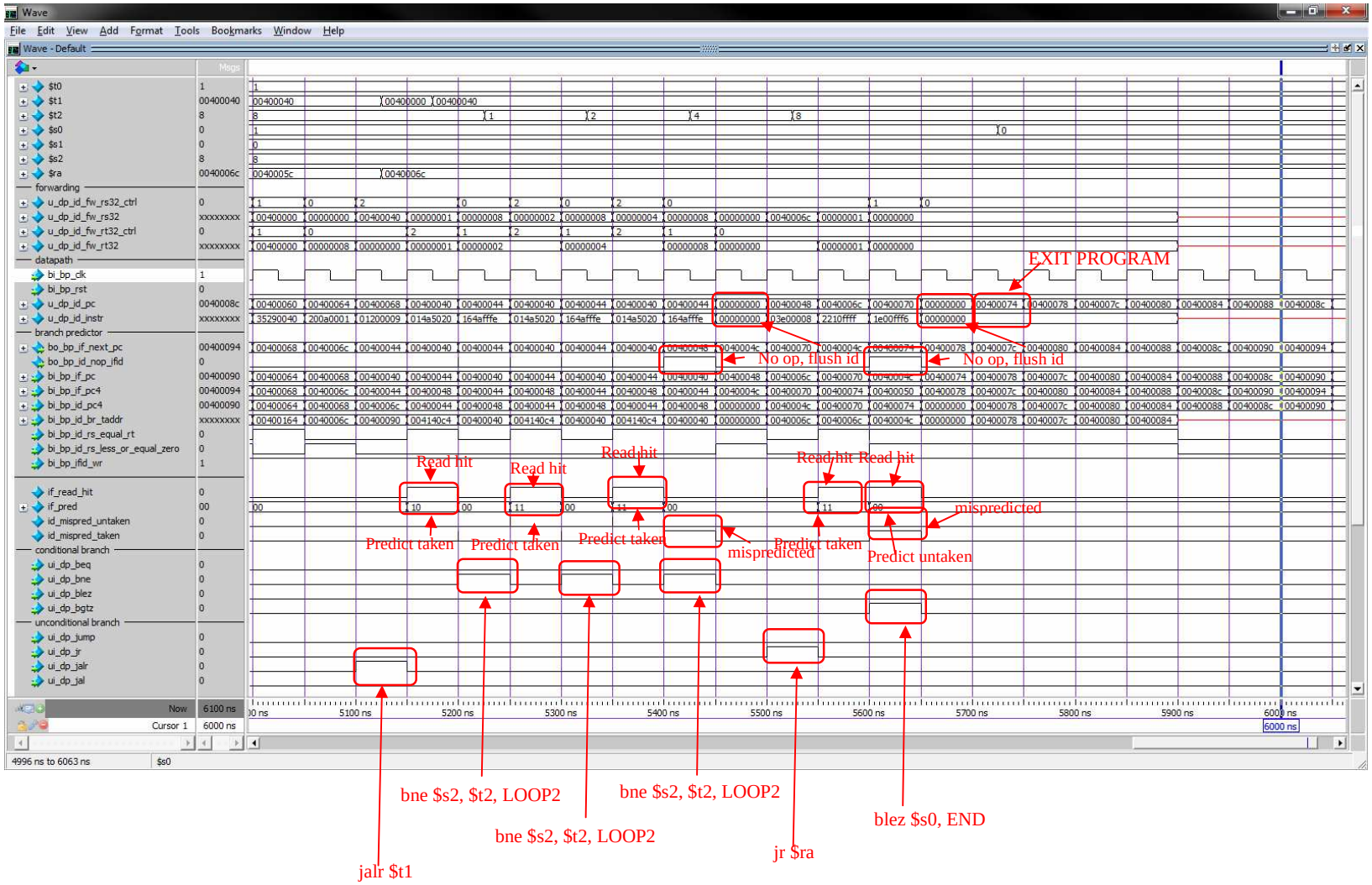

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
forwarding
datapath
branch predictor
conditional branch
unconditional branch
No op, flush id
Read miss
Predict untaken
Read miss
Predict untaken
mispredicted
Read hit
Predict taken
Read hit
Predict taken
j MAIN
blez $s0, END
jal LOOP1
beq $t0, $zero, LOOP1
beq $t0, $zero, LOOP1
beq $t0, $zero, LOOP1


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
$t0
$t1
$t2
$s0
$s1
$s2
$ra
forwarding
u_dp_id_fw_rs32_ctrl
u_dp_id_fw_rs32
u_dp_id_fw_rt32_ctrl
u_dp_id_fw_rt32
datapath
bi_bp_clk
bi_bp_rst
u_dp_id_pc
u_dp_id_instr
branch predictor
bo_bp_if_next_pc
bo_bp_id_nop_ifid
bi_bp_if_pc
bi_bp_if_pc4
bi_bp_id_pc4
bi_bp_id_br_taddr
bi_bp_id_rs_equal_rt
bi_bp_id_rs_less_or_equal_zero
bi_bp_ifid_wr
if_read_hit
if_pred
id_mispred_untaken
id_mispred_taken
conditional branch
ui_dp_beq
ui_dp_bne
ui_dp_blez
ui_dp_bgtz
unconditional branch
ui_dp_jump
ui_dp_jr
ui_dp_jalr
ui_dp_jal
Now
Cursor 1
997 ns to 2064 ns
Read hit
Read hit
Read miss
mispredicted
Read hit
Read hit
No op, flush id
No op, flush id
No op, flush id
Predict taken
Predict taken
mispredicted
Predict untaken
Predict taken
Predict taken
mispredicted
beq $t0, $zero, LOOP1
beq $t0, $zero, LOOP1
jr $ra
jalr $t1
bne $s2, $t2, LOOP2
bne $s2, $t2, LOOP2
bne $s2, $t2, LOOP2
jr $ra


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
$t0
$t1
$t2
$s0
$s1
$s2
$ra
forwarding
u_dp_id_fw_rs32_ctrl
u_dp_id_fw_rs32
u_dp_id_fw_rt32_ctrl
u_dp_id_fw_rt32
datapath
bi_bp_clk
bi_bp_rst
u_dp_id_pc
u_dp_id_instr
branch predictor
bo_bp_if_next_pc
bo_bp_id_nop_ifid
bi_bp_if_pc
bi_bp_if_pc4
bi_bp_id_pc4
bi_bp_id_br_taddr
bi_bp_id_rs_equal_rt
bi_bp_id_rs_less_or_equal_zero
bi_bp_ifid_wr
if_read_hit
if_pred
id_mispred_untaken
id_mispred_taken
conditional branch
ui_dp_beq
ui_dp_bne
ui_dp_blez
ui_dp_bgtz
unconditional branch
ui_dp_jump
ui_dp_jr
ui_dp_jalr
ui_dp_jal
Now
Cursor 1
No op, flush id
Read miss
Read hit
Read hit
Read hit
Read hit
Read hit
Predict untaken
Predict untaken
Predict taken
Predict taken
Predict taken
Predict taken
mispredicted
bgtz $s0, MAIN
blez $s0, END
jal LOOP1
beq $t0, $zero, LOOP1
beq $t0, $zero, LOOP1
beq $t0, $zero, LOOP1
beq $t0, $zero, LOOP1


No op, flush id
No op, flush id
Read hit
Predict taken
mispredicted
Read hit
Predict taken
Read hit
Predict taken
Read hit
Predict taken
mispredicted
Read hit Read hit
Predict taken
Predict untaken
beq $t0, $zero, LOOP1
jr $ra
jalr $t1
bne $s2, $t2, LOOP2
bne $s2, $t2, LOOP2
bne $s2, $t2, LOOP2
jr $ra
blez $s0, END
bgtz $s0, MAIN


Read hit
Predict taken
mispredicted
beq $t0, $zero, LOOP1
jal LOOP1
jr $ra


EXIT PROGRAM
No op, flush id
No op, flush id
Read hit
Read hit
Read hit
Read hit
Read hit
Predict taken
Predict taken
Predict taken
mispredicted
Predict taken
mispredicted
Predict untaken
jalr $t1
bne $s2, $t2, LOOP2
bne $s2, $t2, LOOP2
bne $s2, $t2, LOOP2
jr $ra
blez $s0, END

7.2 Branch Prediction Accuracy Test Program

The branch instructions often take place in if else statement, loop or function call in a program. The program uses to generate Fibonacci series and the program uses to check if a number is prime number involved a number of these conditions. Hence, a program named FiboPrime.c is used in this project to determine the accuracy of the branch predictor. This program combine both the Fibonacci and is Prime program by counting number of prime number out of the nth Fibonacci series.

The maximum Fibonacci number can be generated in this project is limited by the data cache of the RISC32 processor. Since each memory in the data cache is 32-bit, the maximum data value can be stored is equal to 2^{32} = 4,294,967,296.

- The 47^{th} Fibonacci number, Fibo(47) = 2,971,215,073.
- The 48^{th} Fibonacci number, Fibo(48) = 4,807,526,976.

Since Fibo(48) is larger than 2^{32}, the maximum unsigned Fibonacci number can be generated by this processor is the 47^{th} Fibonacci number.

This project runs the program to compute up to the 35^{th} Fibonacci number to determine the prediction accuracy of the branch predictor as it is sufficient to prove the accuracy. Each of the subsequence number will took hours to simulate in a way that the time required is equivalent to the sum of time taken for previous two numbers. $(T_n = T_{n-1} + T_{n-2})$

7.2.1 FiboPrime.c

```
//Fibonacci function, generate nth number of Fibonacci series
int fibo(int n){
	int s1 = 0, s2 = 1, s3 = 0;
	if ( n == 0 || n == 1){
		return n;
	}else {
		for ( int i = 1 ; i < n ; i++){
			s3 = s2 + s1;
			s1 = s2;
			s2 = s3;
		}
		return s3;
	}
}

//isPrime function, calculate the number of prime numbers within the Fibonnacci
series
int isPrime(int n){

	if ( n <= 1) return 0;
	if (n == 2) return 1;
	if ( n % 2 == 0 ) return 0;

	double sq = sqrt(n);

	for (int i = 3; i < sq; i = i + 2){
		if (n%i==0){//this is not a prime
			return 0;
		}
	}
	return 1;
}

//Main Function
void main(void){
	int count = 0;
	for ( int i = 0 ; i <= 35; i++){
		int fb = fibo(i);
		if (isPrime(fb)){
		count = count++;
		}
	}
}
```

7.2.2 MIPS like FiboPrime.c

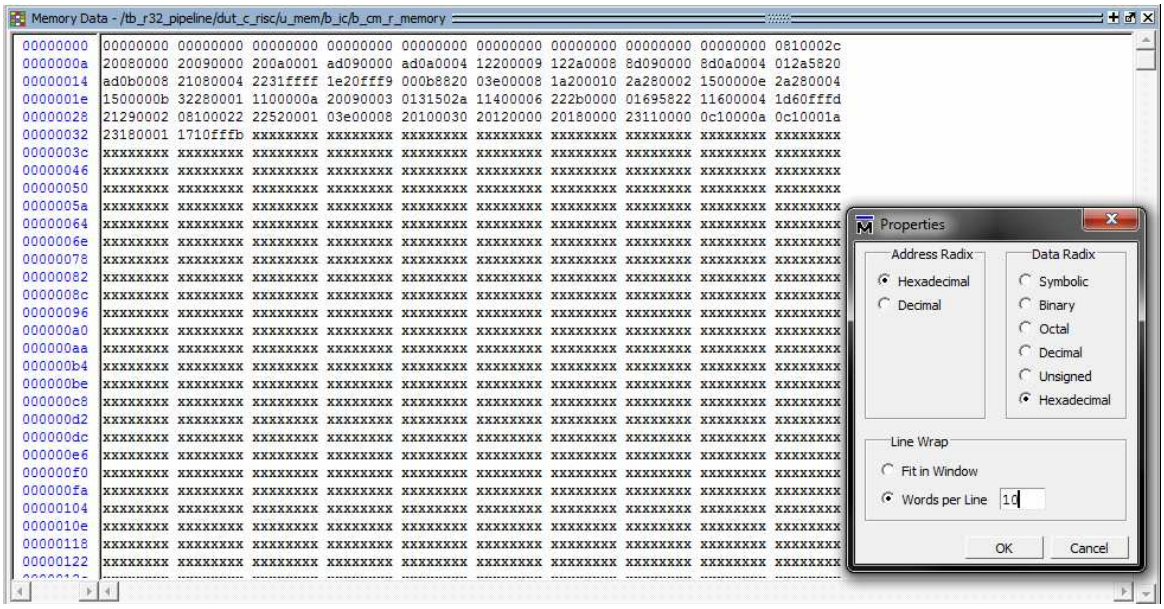
```
void fibo()
{
	vector<int> arr;

	$t0 = 0, $t1 = 0, $t2 = 1;

	arr.pushback($t1);
	arr.pushback($t2);

	if($s1 == 0) return;
	if($s1 == 1) return;

	while($s1 > 0){
		$t3 = arr($t0) + arr($t0 + 1);
		arr.pushback($t3);
		$t0 = $t0 + 1;
		$s1 = $s1 - 1;
	}
	$s1 = $t3;
}

void prime()
{
	if($s1 <= 0)return;
	if($s1 < 2) return;
	if($s1 < 4) {
		$s2 = $s2 + 1;
		return;
	}

	if($s1 % 2 == 0) return;

	$t1 = 3;
	while($t1 < $s1){
		$t3 = $s1;
		do{
			$t3 = $t3 - $t1;
			if($t3 == 0) return;
		}while($t3 > 0);
		$t1 = $t1 + 2;
	}
	$s2 = $s2 + 1;
}

void main()
{
	$s0 = 48, $s2 = 1, $t8 = 0;

	do{
		$s1 = $t8;
		fibo();
		prime();
		$t8 = $t8 + 1;
	}while($t8 < n);
}
```

7.2.3 FiboPrime.c in MIPS

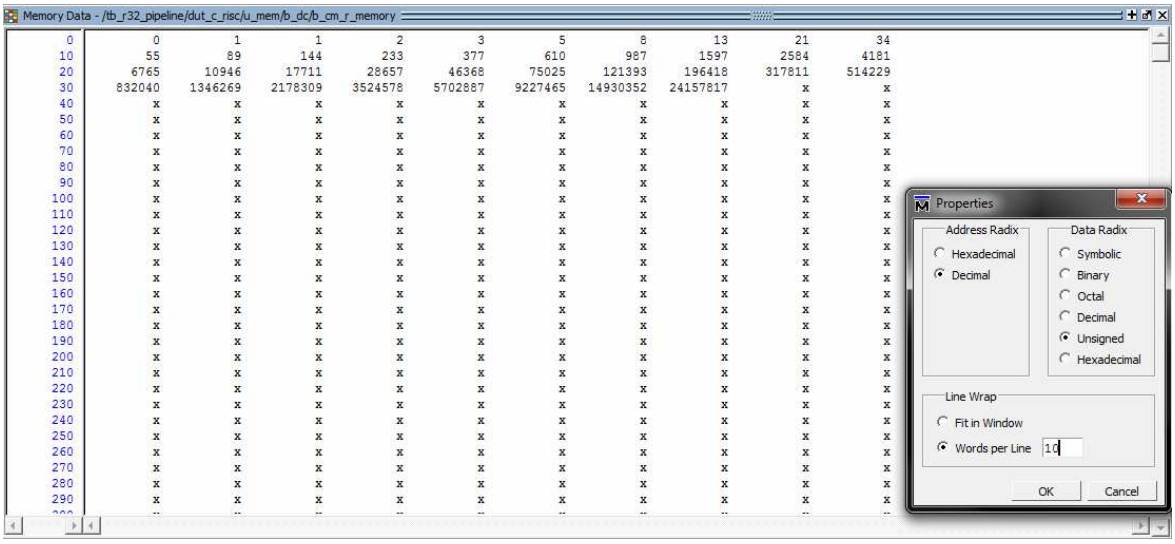
Instruction adress	Instruction code	Label	Mnemonic	Operand 1	Operand 2	Operand 3
0x00400024	0810002C		j	MAIN		
0x00400028	20080000	FIBO:	addi	$t0,	$zero,	0
0x0040002C	20090000		addi	$t1,	$zero,	0
0x00400030	200A0001		addi	$t2,	$zero,	1
0x00400034	AD090000		sw	$t1,	0($t0)	
0x00400038	AD0A0004		sw	$t2,	4($t0)	
0x0040003C	12200009		beq	$s1,	$zero,	BACK
0x00400040	122A0008		beq	$s1,	$t2,	BACK
0x00400044	8D090000	LOOP:	lw	$t1,	0($t0)	
0x00400048	8D0A0004		lw	$t2,	4($t0)	
0x0040004C	012A5820		add	$t3,	$t1,	$t2
0x00400050	AD0B0008		sw	$t3,	8($t0)	
0x00400054	21080004		addi	$t0,	$t0,	4
0x00400058	2231FFFF		addi	$s1,	$s1,	-1
0x0040005C	1E20FFF9		bgtz	$s1,	LOOP	
0x00400060	000B8820		add	$s1,	$zero,	$t3
0x00400064	03E00008	BACK:	jr	$ra		
0x00400068	1A200010	PRIME:	blez	$s1,	FALSE	
0x0040006C	2A280002		slti	$t0,	$s1,	2
0x00400070	1500000E		bne	$t0,	$zero,	FALSE
0x00400074	2a280004		slti	$t0,	$s1,	4
0x00400078	1500000B		bne	$t0,	$zero,	TRUE
0x0040007C	32280001		andi	$t0,	$s1,	0x0001
0x00400080	1100000A		beq	$t0,	$zero,	FALSE
0x00400084	20090003		addi	$t1,	$zero,	3
0x00400088	0131502A	LOOP2:	slt	$t2,	$t1,	$s1
0x0040008C	11400006		beq	$t2,	$zero,	TRUE
0x00400090	222b0000		addi	$t3,	$s1,	0
0x00400094	01695822	LOOP3:	sub	$t3,	$t3,	$t1
0x00400098	11600004		beq	$t3,	$zero,	FALSE
0x0040009C	1D60FFFD		bgtz	$t3,	LOOP3	
0x004000A0	21290002		addi	$t1,	$t1,	2
0x004000A4	08100022		j	LOOP2		
0x004000A8	22520001	TRUE:	addi	$s2,	$s2,	1
0x004000AC	03E00008	FALSE:	jr	$ra		
0x004000B0	20100030	MAIN:	addi	$s0,	$zero,	48
0x004000B4	20120001		addi	$s2,	$zero,	0
0x004000B8	20180000		addi	$t8,	$zero,	0
0x004000BC	23110000	START:	addi	$s1,	$t8,	0
0x004000C0	0C10000A		jal	FIBO		
0x004000C4	0C10001A		jal	PRIME		
0x004000C8	23180001		addi	$t8,	$t8,	1
0x004000CC	1710FFFB		bne	$t8,	$s0,	START
0x004000D0	00000000	EXIT:				

7.2.4 Simulation results

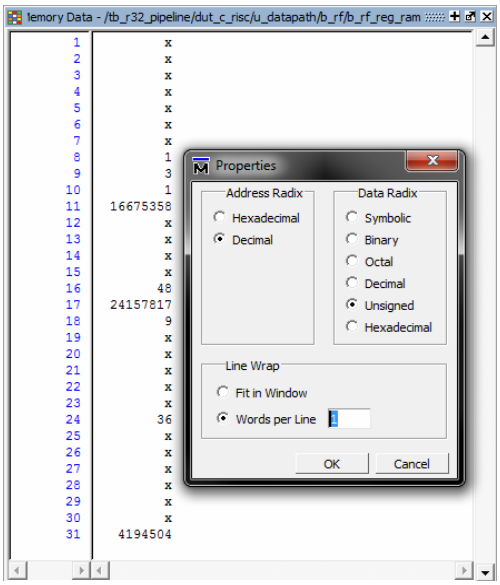
Instruction memory:



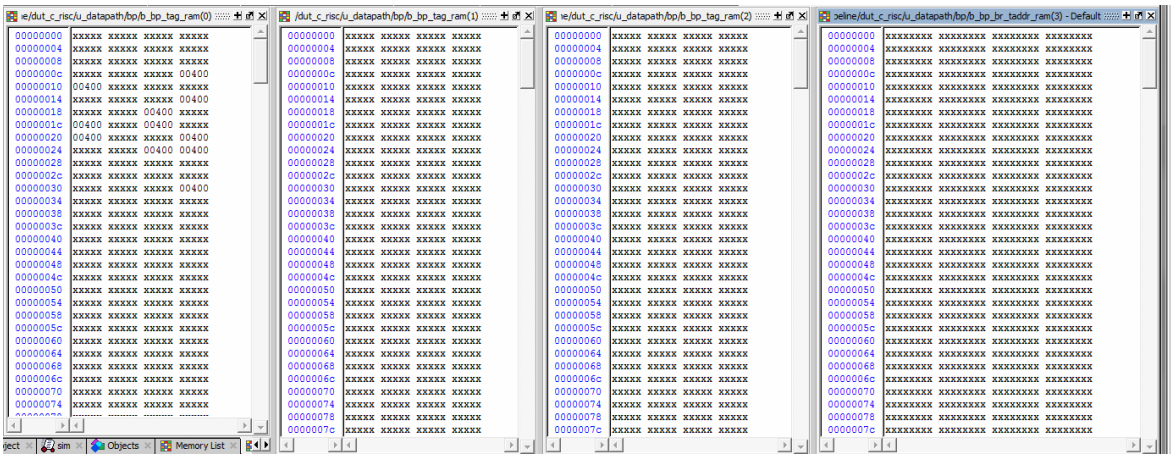
Data memory:



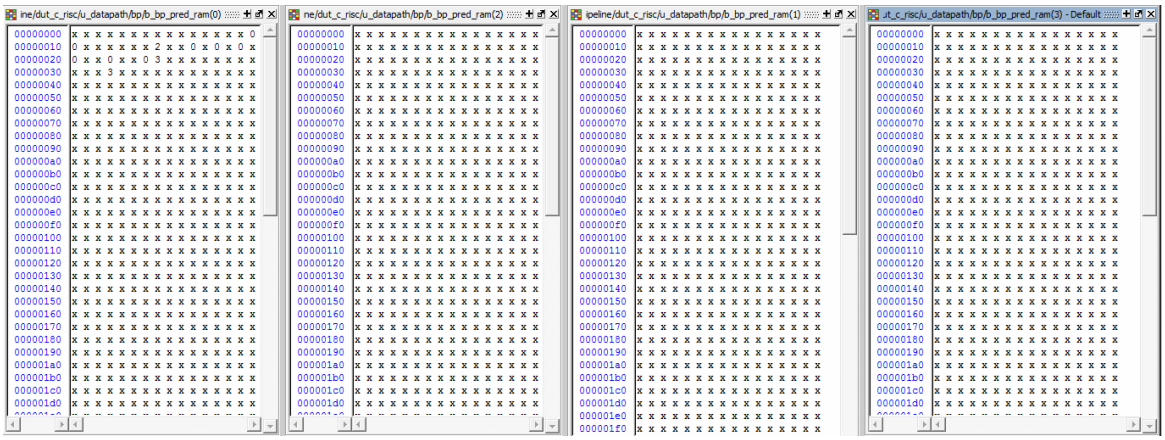
Register File:



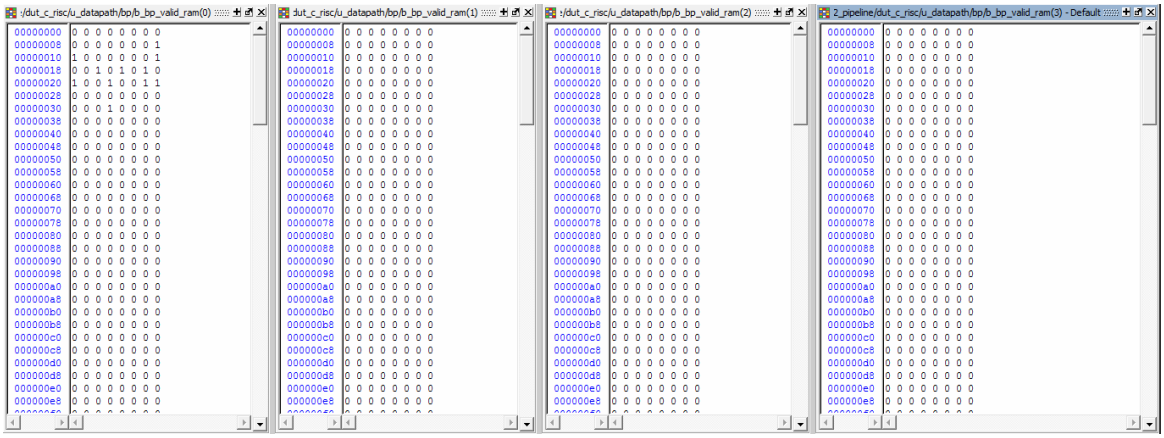
Tag ram:



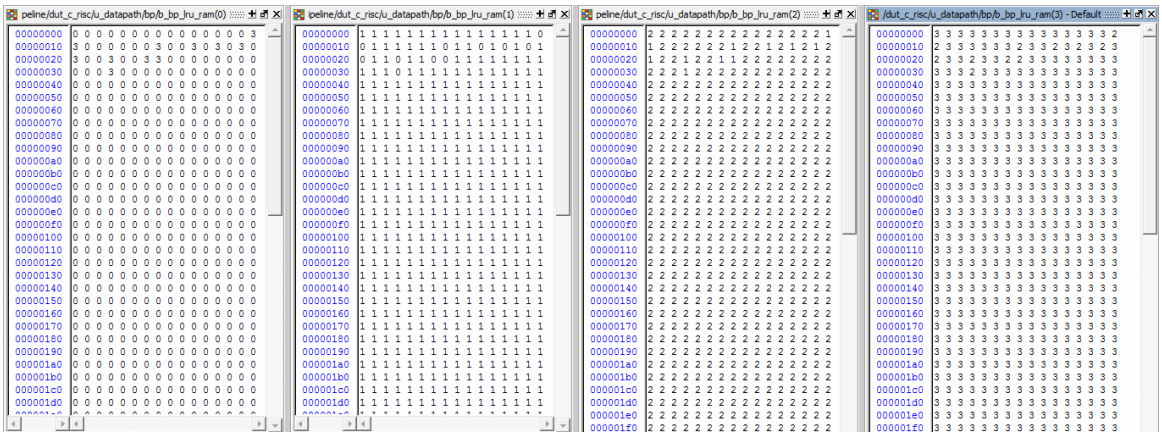
Prediction ram:



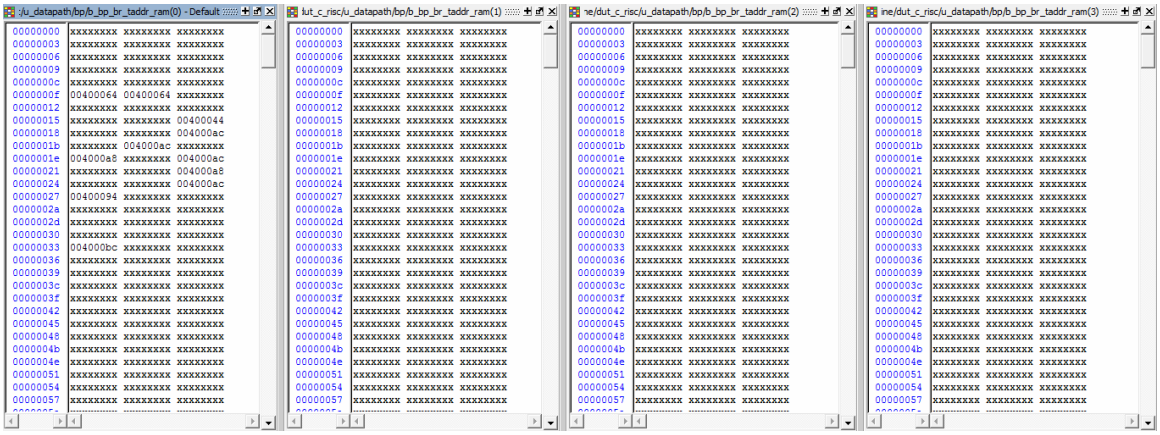
Valid ram:



LRU ram:



Branch target address ram:



N^{th} Fibonacci number, N	Total conditional branch instructions (beq, bne, blez, bgtz)	Number of miss prediction			Total prime number counted by the program, $s2	Branch prediction accuracy, %
		Taken	Untaken	Total		
5	42	9	11	20	3	52.38
10	221	25	18	43	4	80.54
15	2,817	193	23	216	6	92.33
20	32,636	1,012	28	1,040	7	96.81
25	483,729	15,388	33	15,421	8	96.81
30	8,210,758	272,621	38	272,659	9	96.68
35	64,946,330	273,701	43	273,744	9	99.58

Table 10 Branch prediction accuracy

7.2.5 Branch Prediction Accuracy

Total correct prediction = Total conditional branch instructions - Total misprediction

$$\text{Branch prediction accuracy} = \frac{\textit{Total correct prediction}}{\textit{Total conditional branch instructions}} X\ 100\%$$

Equation 1 Branch prediction accuracy

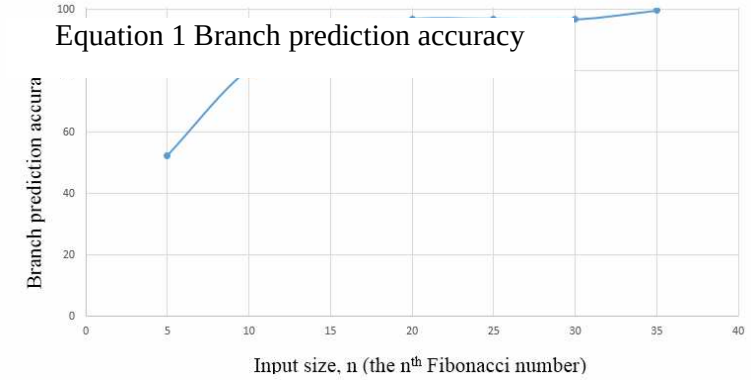


Figure 18 Branch prediction accuracy

$$\text{Average prediction accuracy} = \frac{52.38 + 80.54 + 92.33 + 96.81 + 96.81 + 96.68 + 99}{700}$$

7.2.6 Discussion on branch prediction accuracy

The simulation result shows that the accuracy of branch prediction increases logarithmically approaching 100% as the input size increases. The reason is branch miss-prediction often take place in two cases:

- During the first prediction of a branch instruction, where there is a read miss(instruction is not found in the buffer) and then predictor is forced to predict untaken by default without any data reference of the instruction called previously.
- The prediction bits of a branch instruction is trained to be strongly predict taken or untaken within a loop but the exit of the loop made the prediction predicted wrongly by predicted the oppose condition.

As the input sizes increased, the occurrence of the above cases can be reduced thus improve the accuracy.

7.3 CPU performance after branch predictor integration

The FiboPrime.c is run on both the RISC32 pipeline processor with and without branch predictor to analyse the performance of the RISC32 pipeline processor before and after branch predictor integration.

N^{th} Fibonacci number, n	Number of clock cycle used by RISC32 Pipeline Processor		Performance improvement after branch predictor integration, %
	without branch predictor	with branch predictor	
5	262	211	19.47
10	1,060	817	22.92
15	8,769	5,910	32.60
20	87,631	54,933	37.31
25	1,280,844	797,033	37.77
30	21,756,916	13,546,056	37.73

Table 11 CPU performance analysis

Performance improvement

$$= \frac{(clock\ cycle\ used\ without\ branch\ predictor - @clock\ cycle\ used\ with\ l}{clock\ cycle\ used\ without\ branch\ predictor}$$

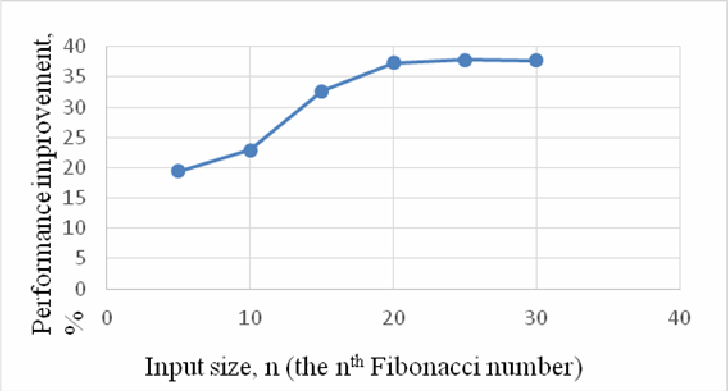


Figure 19 Performance improvement

Average performance improvement

$$= \frac{19.47 + 22.92 + 32.60 + 37.31 + 37.77 + 37.73}{600} X$$

7.3.1 Discussion on CPU performance analysis

The simulation result shows that the performance of the branch predictor increases logarithmically and started to saturate at about 38% as the input size increases.

The RISC32 pipeline processor without branch predictor has to stall for one clock cycle whenever a branch instruction is detected to avoid control hazard no matter the branch condition is taken or untaken.

The RISC32 pipeline processor with branch predictor integrated is able to perform prediction (taken or not taken) when a branch instruction is detected. Thus, the stalling become unnecessary. When the prediction is done correctly, one clock cycle is saved from stalling. When a miss prediction is occurred, one clock cycle is used to flush the incorrect instruction and fetch in the correct instruction.

Chapter 8: Conclusion

8.1 Conclusion

A RISC32 pipeline processor without branch predictor required one clock cycle for stalling whenever a conditional or unconditional branch instruction is detected. This is expensive as research shows that about 20 percent of a program involves branch instructions.

For unconditional branch, the condition will always be taken. Thus, the branch predictor will branch directly to the correct instruction address based on the instruction information when an unconditional branch instruction is detected. In this case, the clock cycle used for stalling is not required.

For conditional branch, the branch predictor is able to save this clock cycle by performing prediction for conditional branch instruction. Even when a miss prediction occurred, the time used for correction is same as the time required for stalling, one clock cycle. A branch predictor with high accuracy can minimize the occurrence of miss prediction.

In short, the integration of branch predictor into the RISC32 pipeline processor does not improve the processor processing speed but is able to improve the performance of the processor in term of the number of clock cycle spent to run a program.

8.2 Future works

The tag ram of the BTB is same as the upper-bits of the branch target address store inside its entry. Thus, the size of the BTB can be reduced at the cost of extra circuitry to read the BTB branch target address upper-bits as tag.

The branch predictor will create new entry for both condition evaluated as taken or untaken when read miss occur. There is a suggestion that to create new entry only for condition evaluated as taken to save the buffer availability for more branch instructions as the branch predictor will predict untaken for read miss case by default. However, the side-effect has to be studied.

REFERENCE

[1] K.M Mok, *Computer Organization and Architecture Notes*, University of Tunku Abdul Rahman, Faculty of Information and Communication Technology, 2014.

[2] http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html [Accessed: 9 APRIL 2014]

[3] http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html [Accessed: 9 APRIL 2014]

[4] Stephen Bailey, *Comparison of VHDL, Verilog and SystemVerilog*, Techincal Marketing Engineer [online] Available at http://www.fpga.com.cn/advance/vhdl_14919.pdf [Accessed: 11 APRIL 2014]

[5] http://www.webopedia.com/TERM/P/pipelining.html [Accessed: 10 APRIL 2014]

[6] http://www.ece.unm.edu/~jimp/611/slides/chap4_5.html [Accessed: 10 APRIL 2014]

[7] http://mail.humber.ca/~paul.michaud/Pipeline.htm [Accessed: 10 APRIL 2014]

[8] http://www.cs.iit.edu/~cs561/cs350/CPU/5stage.html [Accessed: 10 APRIL 2014]

[9] http://en.wikipedia.org/wiki/Branch_predictor [Accessed: 19 JULY 2014]

[10] Professor Ben Lee, *Dynamic Branch Prediction*, Oregon State University. [online] Available at http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/ [Accessed: 8 APRIL 2014]

[11] Patterson, David. *Computer Architecture A Quantitative Approach*. 2ed . s.l. : Morgan Kaufmann.

[12] Hennessy, John L. and Patterson, David A. *Computer Organization and Design : The Hardware/Software Interface*. 4th. San Francisco : Morgan Kaufmann

[13] Min Cheng, Ho, The Design And Development Of A Branch Target Buffer Based On A 2-bit Prediction Scheme For A 32-bit RISC32 Pipeline Processor, Faculty of Information and Communication Technology, University Tunku Abdul Rahman.

[14]http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/ [Accessed: 19 JULY 2014]