

**Pipeline CPU**

Computer System (1) ------ Final Project

**Report**

MIPS 5-Level Pipeline CPU (Advanced)

Author : Xu Yifei & Zhang Yiyi

Stu. ID : 5130309056 & 5132409031

Class : F1324004 (ACM2013)

College : Zhiyuan College

University : Shanghai Jiao Tong University

Contents

[Introduction 2](#_Toc422423110)

[Goals 3](#_Toc422423111)

[HOW TO USE? 3](#_Toc422423112)

[Feature 4](#_Toc422423113)

[1. Bypassing 4](#_Toc422423114)

[2. L1 Cache 5](#_Toc422423115)

[ **Direct Mapping** 6](#_Toc422423116)

[ **Write Back** 7](#_Toc422423117)

[3. Dynamic Branch Prediction 7](#_Toc422423118)

[ **Reducing the delay of branches** 7](#_Toc422423119)

[ **Dynamic Branch Prediction method** 8](#_Toc422423120)

[Design 9](#_Toc422423121)

[ **Project Outlines** 9](#_Toc422423122)

[ **Pipeline** 9](#_Toc422423123)

[ **Stage ID** 10](#_Toc422423124)

[ **Stage EX** 11](#_Toc422423125)

[ **Stage ME** 12](#_Toc422423126)

[ **Stage WB** 12](#_Toc422423127)

[Timetable 12](#_Toc422423128)

[Acknowledgement 12](#_Toc422423129)

[Reference 12](#_Toc422423130)

# 

**Pipeline CPU**

Computer System (1) ------ Final Project

# Introduction

This is a MIPS 5-Level Pipelined CPU with a few advanced features.

There are following features:

- Full bypassing supported;

- 2-Level branch predictor;

- 1-level cache provided;

This CPU is written in Verilog HDL and will be simulated with ModelSim 10.4a Student Edition.

This project is a final project of Computer System (1) by instructor Xiaoyao Liang in Spring 2015.

- Data through: 2015.06.01 - 2015.06.19

- Presentation: 2015.06.19 11:00 - 11:15

# Goals

The requiry of the project is : <http://www.cs.sjtu.edu.cn/~liang-xy/ms108/project.pdf>

The C++ code :

int main() {

int e = 200;

int ans = 0;

do {

int x = memory[i];

i++;

ans += x;

} while (i != 200)

memory[e] = ans;

}

The binary code runs is :

001101 00000 01010 00000 01100 100000 //04

001101 00000 01011 00000 00000 000000 //08

001101 00000 01100 00000 00110 010000 //0C

100011 01100 01101 00000 00000 000000 //10

001000 01100 01100 00000 00000 000100 //14

000000 01011 01101 01011 00000 100000 //18

000101 01100 01010 11111 11111 111100 //1C

101011 01010 01011 00000 00000 000000 //20

111111 11111 11111 11111 11111 111111 //24

The mips code runs is :

ori $10, $0, 800

ori $11, $0, 0

ori $12, $0, 400

lw $13, 0($12)

addi$12, $12, 4

add $11, $11, $13

bne $12, $10, -4

sw $11, 0($10)

stop

Our full code repository is available on: <https://github.com/fei960922/Pipeline_CPU>

We hope our work will benefit other guys.

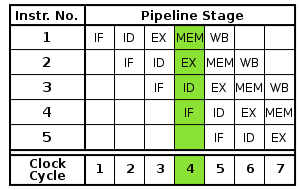
Cheers!

# HOW TO USE?

1. Change the path of file and file\_mips in pipeline.v to follow your directory.
2. Open the modelsim to start a new library.
3. Add all the file to the library through the compile buttion and compile them.
4. Start simulation final\_test.v by double click.
5. Move the variable generated by ‘final\_test.v’ to Objects and open the wave option in the ‘View’ menu.
6. Change the run length to more than 6000ns (we have a cycle of 2ns) and run.
7. It will be automatically stopped when faced 111111 11111 11111 11111 11111 111111.
8. You can add all variable into the wave windows to see its value.
   1. The register is in : p(pipeline) -> d(stage id) -> rf -> data -> [10 - 13]
   2. The data memory is in : p(pipeline) -> h(mem\_ad) -> memory -> [100 - 200]
9. The answer will be find in : p(pipeline) -> h(mem\_ad) -> memory -> [200]

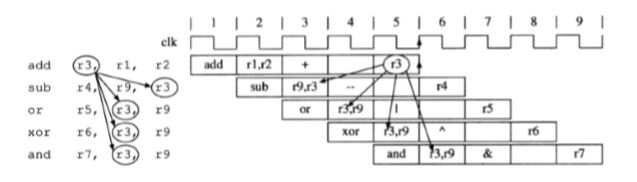
# Feature

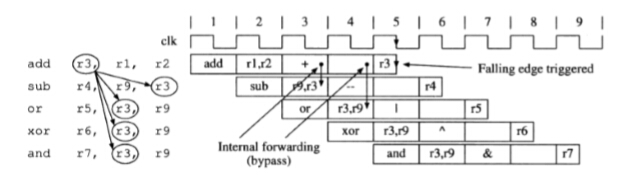
1. Bypassing

Pipeline is a technique used in the design of computers to increase their instruction throughput. Rather than processing each instruction sequentially (one at a time, finishing one instruction before starting the next), each instruction in our pipeline CPU project is split up into 5 steps so different steps can be executed concurrently (at the same time) and in parallel (by different circuitry)

However, pipeline may also has some problem. (1) Structural Hazard/ Dependence (2) Data Hazard/ Dependence (3) Branching Hazard/ Dependence

(2) Data Hazard

During pipeline, we may face the situation below:

The first add instruction write the result of r1+r2 back to the register r3 in the fifth cycle, then the sub, or, xor instructions below can not get the correct answer from register r3 in their ID step. In our CPU, this problem can be solved by moving the bypassing to the ID step with 2 select\_4 mux.

We save the result during the WB step when the falling edge triggered (it is the same to say that WB step cost only half of the cycle) and a\_select and b\_select are chosen by the 2 select\_4 mux to select the right data, and write it back to the register when the ID step ends.

// bypassing

a\_select = 2'b00;

if (wreg\_ex & (rw\_ex != 0) & (rw\_ex == rs) & ~rmem\_ex)

a\_select = 2'b01;

else if (wreg\_me & (rw\_me != 0) & (rw\_me == rs) & ~rmem\_me)

a\_select = 2'b10;

else if (wreg\_me & (rw\_me != 0) & (rw\_me == rs) & rmem\_me)

a\_select = 2'b11;

When forwarding a, we follow the step by assuming there is no hazard -> select exe\_alu -> select mem\_alu -> select men\_lw. This is because, the result of ALU can be forwarded both from EX & ME, however, instruction lw can be only forwarded from ME. Thus, the instruction just after the lw may be installed if its data is used in lw.

The stall may be written as:

stall = wreg\_ex & rmem\_ex & (rw\_ex != 0) & ((use\_rs & (rw\_ex == rs)) | (use\_rt & (rw\_ex == rt)));

1. L1 Cache

Cache refers to any storage managed to take advantage of locality of access. There are 3 ways to assign a location in the cache for each word in memory: (1) Direct Mapping (2) Fully Associative Mapping (3) Set Associative Mapping, and 2 ways to keep the main memory and the cache consistent: write-through and write-back.

In our project, we use direct mapping and write-back.

To make our design more realistic, according to i5-3317U(Intel Ivy bridge):

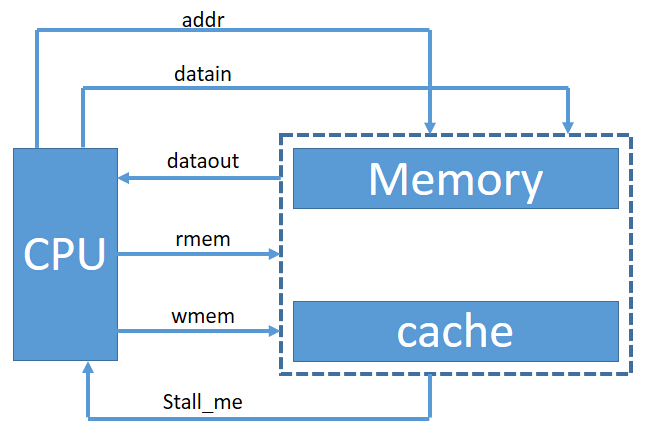
* The L1 cache is 32KB data and 32KB instruction.
* The block size of it is 64Bytes.

That is equals to 16 words per block and total 512 blocks in each cache.

Our cache is designed to link with CPU and Memory as follows:

stall\_me = (tags !== cache\_tag[mods]) & (rmem | wmem);

* **Direct Mapping**



In direct mapping, we use index as the address of the cache register to visit cache, however, tag which only contain the upper portion of the address are required to identify whether the block corresponds to the request word.

The address from the CPU to the cache is divided into:

|  |  |  |
| --- | --- | --- |
| Block Address | | Block Offset |
| Tag | Index |

**A block address contains tag and index:**

* An index field to choose a block set in the cache.
* A tag field to search and match addresses in the selected set.

**A block offset to select the data from the block**.

assign tags = addr[31:15];

assign mods = addr[14:6];

assign offs = addr[5:2];

Our cache has the data of 32KB = 32 \* 1024 = 2^15 B

Then, the bits of tag is 32 – 15 = 17.

and there are 512 blocks in each cache, index needs 10 bits.

the bit of block offset is 4.

In direct-mapped cache we use this mapping to find the block:

and

to find out the data. So we define:

assign data\_out = cache[mods][offs];

* **Write Back**

With write back strategy, we update values only to the block in the cache, and write to the memory when it is replaced. We use cache\_dirty to realize if the data has been updated.

In write back cache with no-write allocate, we update as follows:

for (i=0;i<16;i=i+1) begin

if (cache\_dirty[mods])

memory[{cache\_tag[mods], mods, 4'h0} + i] = cache[mods][i];

cache[mods][i] = memory[{addr[31:6], 4'h0} + i];

end

1. Dynamic Branch Prediction

* **Reducing the delay of branches**

Traditionally, branch instruction should be done in EX stage because we won't know whether we take the branch or not before we compare the conditions in EX stage. Stall will happened until the branch is complete, which makes our pipeline too slow.

A 2-bit global branch prediction is added into our design.

When we face a branch instruction(That is 'bp\_isbranch\_id' == 1). We will do following:

(1) Fetch result from branch prediction (The result is 'bp\_taken\_id').

If the prediction result is TAKEN: Let pc\_select = 2'b01 which mean pc\_b will be selected;

Otherwise, pc\_select = 2'b00 which mean pc4 will be selected;

(2) Storage another PC into the branch prediction.

if prediction result is TAKEN, storage pc4. Otherwise, storage pc\_b instead.

(3) Transmit 'bp\_taken\_id', 'bp\_isbeq\_id', 'bp\_isbranch\_id' to EX stage.

\*\*\* Next cycle \*\*\*

(4) In EX stage, we calculate the condition and output these:

bp\_taken\_ex = ((a\_ex == b\_ex) ^ bp\_isbeq\_ex); // Whether this branch should be TAKEN.

bp\_succ\_ex = ~(bp\_taken\_ex ^ bp\_taken\_in); // Whether this branch is predicted correctlly.

// \* bp\_taken\_in is the predict result.

These wire will be linked to ID stage.

(5) Decide whether the new instruction in ID stage is correct or not.

bp\_reset = (~bp\_isbranch\_ex) | bp\_succ\_ex;

// If the previous instruction is a branch and its prediction failed.

If the prediction is failed (bp\_reset==0):

a) Clean the current instruction because it is a wrong one.

instr\_true = instr\_id & {32{bp\_reset}};

b) Modify next pc to a correct one. Notice another PC we storage in the branch prediction is a correct one.

pc\_b = (bp\_reset) ? (pc4\_id + {imm\_id[29:0], 2'b00}) : pc\_bf;

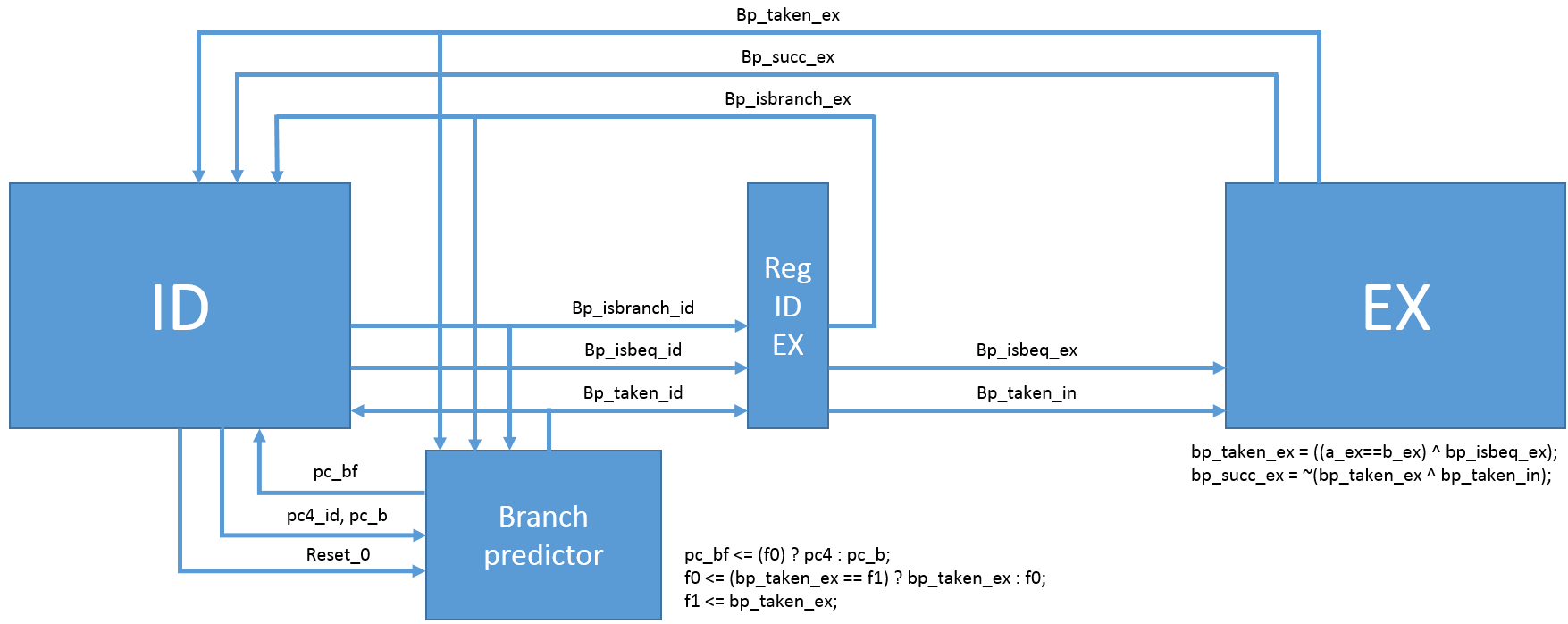
// pc\_bf is the storaged PC.

if (bp\_reset) pc\_select = 2'b01;

// We transmit this PC by pc\_b.

Whatever the prediction is success or not, we should give the branch prediction a feedback.

Here follows the graph lines:



'bp\_taken\_ex' : transmitted to the branch prediction.

bp\_isbranch\_id : whether the current instruction is a branch or not

bp\_isbeq\_id : Whether current instruction is jump if equal or not

pc\_now : The current pc

pc\_b : Next pc if Branch happens

pc4\_id : Next pc if Branch not happens

pc\_bf : pc if predict failed

bp\_succ\_ex : True if predict success

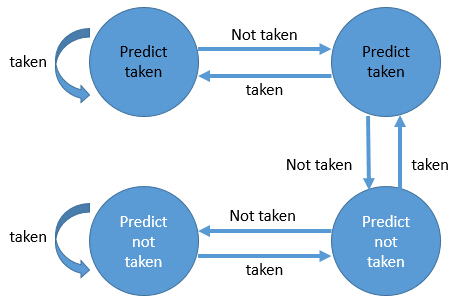
* **Dynamic Branch Prediction method**

In our design, we use dynamic branch prediction, which is an approach to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time.

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, we choose to use 2-bit prediction schemes.

We use two register f0 and f1 to memorize whether the branch was recently taken or not, in such 2-bit scheme, a prediction must be wrong twice before it is changed.

Below shows the finite-state machine for our 2-bit prediction scheme:



always @(negedge reset\_0 or posedge clock) begin

if (reset\_0 == 0) begin

f0 <= 1;

f1 <= 1;

end else begin

if (bp\_isbranch\_id)

pc\_bf <= (f0) ? pc4 : pc\_b;

if (bp\_isbranch\_ex) begin

f0 <= (bp\_taken\_ex == f1) ? bp\_taken\_ex : f0;

f1 <= bp\_taken\_ex;

end

end

end

# Design

* **Project Outlines**
* **Pipeline**

Pipeline.v is the top of this pipeline CPU project.

The pipeline includes 5 stages: IF, ID, EX, ME, WB.

|  |  |
| --- | --- |
| Reg\_name | Name\_x:  x = if/id/ex/me/wb  which represent using in each stage |
| clock | CPU clock |
| Reset\_0 | To reset CPU with all reg |
| pc | Instruction address   * pc4 : Next pc if nothing happens(pc + 4) * pc\_b : Next pc if Branch happens * pc\_j : Next pc if Jump happens * pc\_r : Next pc if Jr happens * pc\_next : Real next pc * pc\_select : Select the real next pc |

* **Stage ID**

|  |  |
| --- | --- |
|  |  |
| Ans | Answer of ALU |
| mo | Memory output |
| wreg | Wreg = 1 write register  Wreg = 0 don’t write register |
| rmem | Rmem =1 write register with the data in memory  Rmem = 0 write register with ALU |
| rw\_select | Rw\_select = 1 select rt  Rw\_select = 0 select rd |
| wmem | Wmem = 1 write memory  Wmem = 0 don’t write memory |
| shift | Shift = 1 ALU a used for shift  Shift = 0 use the data in register |
| aluimm | Aluimm = 1 ALU b used as immediate number  Aluimm = 0 use data in register |
| sext | Sext = 1 expand the width of immediate number  Sext = 0 no expansion of immediate number |
| jal | Jal = 1 The op is jal  Jal = 0 the op is not |
| op\_id[3:0] | table of ALU operation  ADD 0000;  SUB 0100;  MUL 1000;  DIV 1100;  AND 0001;  OR 0101;  XOR 1001;  LUI 1101;  SLL 0010;  SRL 1110;  SRA 1010; |
| pc\_select[1:0] | The selection of address for the next instruction  00: choose pc+4  01: choose goto address  10: choose the address in the register  11: choose jump or jump link address |

Below is te table of the 20 MIPS instruction:

* Note:
* Sa : shift amount
* rs, rt : input register number
* rd : destination register number

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| instruction | Instruction name | [31:26] | [25:21] | [20:16] | [15:11] | [10:6] | [5:0] |
| add | Add to register | 000000 | rs | rt | rd | 00000 | 100000 |
| sub | Sub to register | 000000 | rs | rt | rd | 00000 | 100010 |
| and | And to register | 000000 | rs | rt | rd | 00000 | 100100 |
| or | Or to register | 000000 | rs | rt | rd | 00000 | 100101 |
| xor | Xor to register | 000000 | rs | rt | rd | 00000 | 100110 |
| sll | Shift left | 000000 | 00000 | rt | rd | sa | 000000 |
| srl | Right Logic | 000000 | 00000 | rt | rd | sa | 000010 |
| sra | Right Arithmetic | 000000 | 00000 | rt | rd | sa | 000011 |
| jr | Jump register | 000000 | rs | 000000 | 00000 | 00000 | 001000 |
|  |  |  |  |  |  |  |  |
| addi | Add immediate | 001000 | rs | rt | Immediate | | |
| andi | And immediate | 001100 | rs | rt | Immediate | | |
| ori | Or Immediate | 001101 | rs | rt | Immediate | | |
| xori | XOR Immediate | 001110 | rs | rt | Immediate | | |
| lw | Load word | 100011 | rs | rt | Offset | | |
| sw | Store word | 101011 | rs | rt | Offset | | |
| beq | Branch on equal | 000100 | rs | rt | Offset | | |
| bne | Branch on not equal | 000101 | rs | rt | Offset | | |
| lui | Load upper immediate | 001111 | 00000 | rt | Immediate | | |
| j | jump | 000010 | | | address | | |
| jal | Jump and link | 000011 | | | address | | |

* **Stage EX**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| instr | wreg | rw\_select | jal | rmem | shift | aluimm | sext | op\_id | wmem | pc\_select |
| ADD | 1 | 0 | 0 | 0 | 0 | 0 | X | 0000 | 0 | 00 |
| SUB | 1 | 0 | 0 | 0 | 0 | 0 | X | 0100 | 0 | 00 |
| MUL | 1 | 0 | 0 | 0 | 0 | 0 | X | 1000 | 0 | 00 |
| DIV | 1 | 0 | 0 | 0 | 0 | 0 | X | 1100 | 0 | 00 |
| AND | 1 | 0 | 0 | 0 | 0 | 0 | X | 0001 | 0 | 00 |
| OR | 1 | 0 | 0 | 0 | 0 | 0 | X | 0101 | 0 | 00 |
| XOR | 1 | 0 | 0 | 0 | 0 | 0 | X | 1001 | 0 | 00 |
| SLL | 1 | 0 | 0 | 0 | 1 | 0 | X | 0010 | 0 | 00 |
| SRL | 1 | 0 | 0 | 0 | 1 | 0 | X | 1110 | 0 | 00 |
| SRA | 1 | 0 | 0 | 0 | 1 | 0 | X | 1010 | 0 | 00 |
| LUI | 1 | 1 | 0 | 0 | x | 1 | x | 1101 | 0 | 00 |

* **Stage ME**

The ME stage in our project is a L1 cache mentioned above.

* **Stage WB**

data\_w = rmem\_wb ? mo\_wb : ans\_wb;

Get the answer and get back to the register in ID stage.

# Timetable

- 2015.06.01 Project Started.

- 2015.06.03 Reference reading.

- 2015.06.05 Repository established.

- 2015.06.13 Basic Version release.

- 2015.06.15 Version Alpha release.

- 2015.06.17 Final Version release.

- 2015.06.19 Final DDL. Presentation.

# Acknowledgement

Thanks Prof Liang Xiaoyao’s hard work for us to get the knowledge of computer architecture.

Thanks TA Ye Ran’s dedicated work to receive our homework on time.

# Reference

《Computer Principles and Design in Verilog HDL》

《Computer Architecture Experiment Instruction (LAB1-6)》

《Verilog for Ditigal Circuits》