# 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
Goals .......... 3
HOW TO USE? .......... 3
Feature .......... 4
1. Bypassing .......... 4
2. L1 Cache .......... 5
- **Direct Mapping** .......... 6
- **Write Back** .......... 7
3. Dynamic Branch Prediction .......... 7
- **Reducing the delay of branches** .......... 7
- **Dynamic Branch Prediction method** .......... 8
Design .......... 9
- **Project Outlines** .......... 9
- **Pipeline** .......... 9
- **Stage ID** .......... 10
- **Stage EX** .......... 11
- **Stage ME** .......... 12
- **Stage WB** .......... 12
Timetable .......... 12
Acknowledgement .......... 12
Reference .......... 12

*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 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
```

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
```

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.
    a. The register is in : p(pipeline) -> d(stage id) -> rf -> data -> [10 - 13]
    b. 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

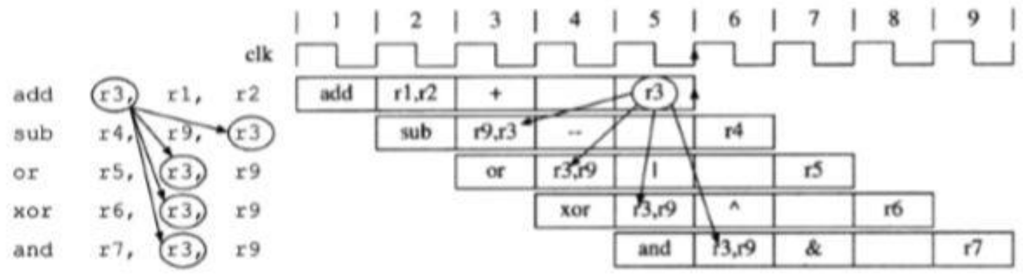
### 1. Bypassing

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)

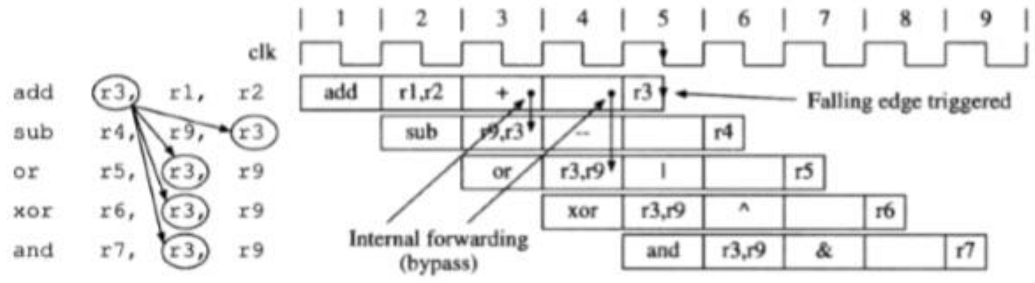
| Instr. No. | Pipeline Stage | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | IF | ID | EX | MEM |
| 5 | | | | | IF | ID | EX |
| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

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)));
```

## 2. 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.

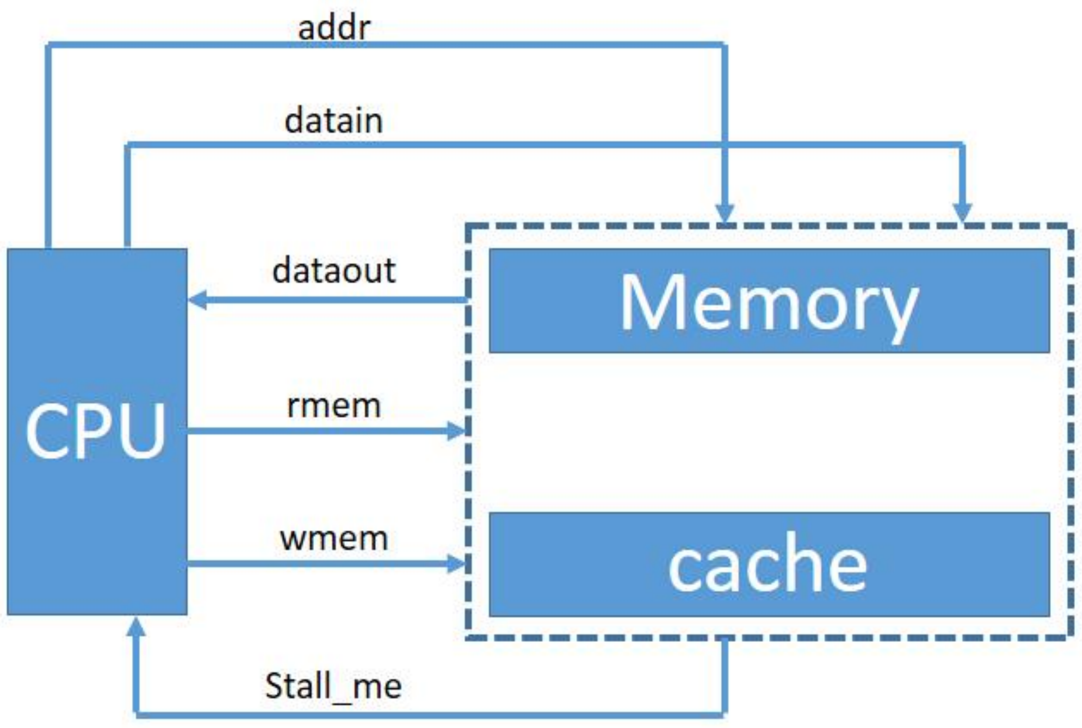
To make our design more realistic, according to i5-3317U(Intel Ivy bridge):

- 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:

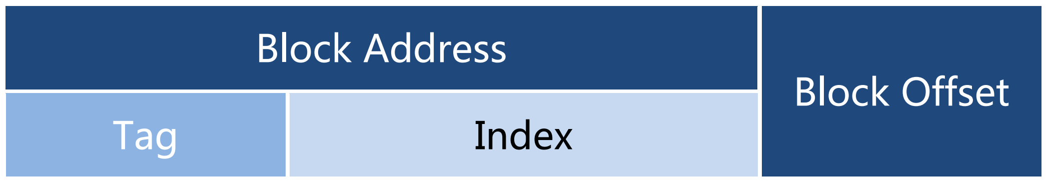
```
stall_me = (tags !== cache_tag[mods]) & (rmem | wmem);
```



- **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:



**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.

$\log_2 512 = 9$ and there are 512 blocks in each cache, index needs 10 bits.

$\log_2 16 = 4$ the bit of block offset is 4.

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

$$(block\ address)\ module\ (number\ of\ blocks\ in\ the\ cache)$$

and

$$(word\ address)\ module\ (number\ of\ words\ in\ the\ cache)$$

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
```

## 3. 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.

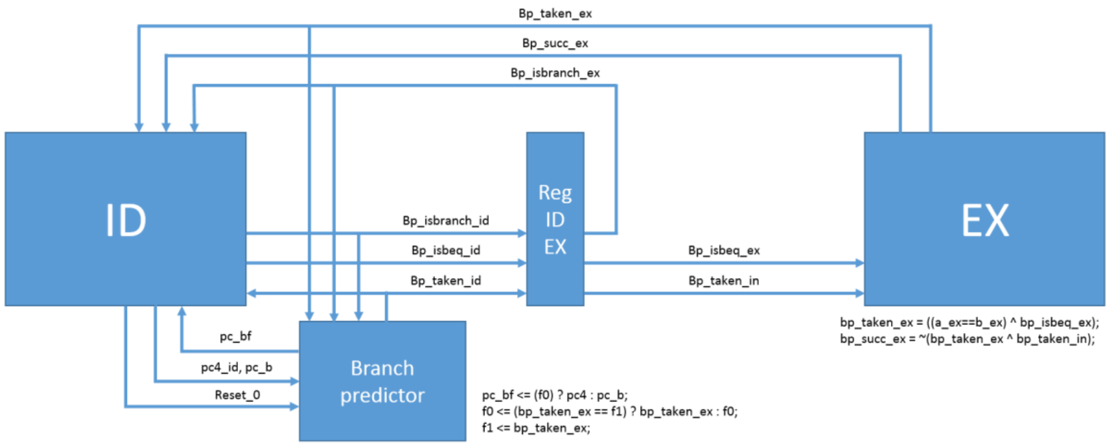
```
        instr_true = instr_id & {32{bp_reset}};
```

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 = ((a_ex==b_ex) ^ bp_isbeq_ex);
bp_succ_ex = ~(bp_taken_ex ^ bp_taken_in);
```

```
pc_bf <= (f0) ? pc4 : pc_b;
f0 <= (bp_taken_ex == f1) ? bp_taken_ex : f0;
f1 <= bp_taken_ex;
```

| | | |
|---|---|---|
| '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 |

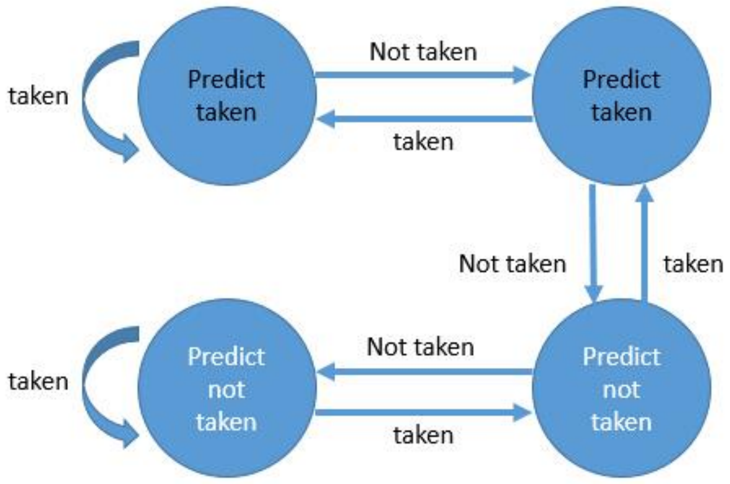
- **Dynamic Branch Prediction method**

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**

| Top Level | Stage Level | Tool Level |
|---|---|---|
| ☐ pipeline.v | ☐ stage_if.v | ☐ reg_array.v |
| ☐ final_test.v | ☐ stage_id.v | ☐ reg_cell.v |
| | ☐ stage_ex.v | ☐ mem_advanced.v |
| | ☐ reg_ifid.v | ☐ select_4.v |
| | ☐ reg_idex.v | |
| | ☐ reg_exme.v | |
| | ☐ reg_mewb.v | |

- **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:<br>x = if/id/ex/me/wb<br>which represent using in each stage |
|---|---|
| **clock** | CPU clock |
| **Reset_0** | To reset CPU with all reg |
| **pc** | Instruction address<br>➢ pc4 : Next pc if nothing happens(pc + 4)<br>➢ pc_b : Next pc if Branch happens<br>➢ pc_j : Next pc if Jump happens<br>➢ pc_r : Next pc if Jr happens<br>➢ pc_next : Real next pc<br>➢ pc_select : Select the real next pc |

- **Stage ID**

| **Ans** | Answer of ALU |
|---|---|
| **mo** | Memory output |
| **wreg** | Wreg = 1 write register<br>Wreg = 0 don't write register |
| **rmem** | Rmem =1 write register with the data in memory<br>Rmem = 0 write register with ALU |
| **rw_select** | Rw_select = 1 select rt<br>Rw_select = 0 select rd |
| **wmem** | Wmem = 1 write memory<br>Wmem = 0 don't write memory |
| **shift** | Shift = 1 ALU a used for shift<br>Shift = 0 use the data in register |
| **aluimm** | Aluimm = 1 ALU b used as immediate number<br>Aluimm = 0 use data in register |
| **sext** | Sext = 1 expand the width of immediate number<br>Sext = 0 no expansion of immediate number |
| **jal** | Jal = 1 The op is jal<br>Jal = 0 the op is not |
| **op_id[3:0]** | table of ALU operation<br>`ADD     0000;`<br>`SUB     0100;`<br>`MUL     1000;`<br>`DIV     1100;`<br>`AND     0001;`<br>`OR      0101;`<br>`XOR     1001;`<br>`LUI     1101;` |

| | |
|---|---|
| | SLL 0010;<br>SRL 1110;<br>SRA 1010; |
| **pc_select[1:0]** | The selection of address for the next instruction<br>`00: choose pc+4`<br>`01: choose goto address`<br>`10: choose the address in the register`<br>`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》