Computer Architecture 2019 Fall - Project 1 : A Pipeline CPU

Team : Cashy Cache & Pipeline Pie

1. Member & Team work

Member				Work
資工四	b05902133	范勝禹	40%	1. Overall debug 2. Most Pipelines Registers 3. debug CPU.v 4.ALU.v, ALU_Control.v, HazardDetection.v 5.The “Difficulties Encountered and Solutions Part” 6.The “CPU Implementation Report Part”
資工三	T08902109	賈成銷	30%	1. 寫MUX32, MUX3, Sign_Extend,Control , Forward, IF_ID幾部分的module。 2. 寫CPU部分，將各個module初步連接起來。 3. 協助隊友對CPU部分的code通過gtkware進行debug。 4. 寫report。
資工四	B03902002	高新造	30%	1.寫Adder, MUX_Control的module 2.Debug

				3.寫report

2. Implementation

(1) CPU.v Implementation

(A) 5 Stage Pipelining

The flow chart of stages, their modules, & pipeline registers :

(1) IF Stage :

PC MUX_PC Instruction_Memory Add_PC_4

| IF_ID.v Pipeline Register |

(2) ID Stage :

Sign_Extend Add_PC_Branch
Control MUX_Control
HazardDetection And Registers

| ID_EX.v Pipeline Register |

(3) EX Staege :

MUX_ALU_data1 MUX_ALU_data2 MUX_ALU_data2_src
ALU ALU_Control
Forward

| EX_MEM.v Pipeline Register |

(4) MEM Stage :

Data_Memory

| MEM_WB Pipeline Register |

(5) WB Stage :

MUX_MemtoReg : select data between ALU result or data memory to write the register file.

(B) EX & MEM Data Hazard : Known In EX Stage, By Forward.v

The conditions depend on the instructions in the EX, MEM, & WB Stages :

```
    // A
    if(EX_MEM_RegWrite_i && EX_MEM_RDaddr_i != 0 && EX_MEM_RDaddr_i ==
ID_EX_RS1addr_i)
        select_1_reg = 2'b10; // select data2 , from EX_MEM
    else if(MEM_WB_RegWrite_i && MEM_WB_RDaddr_i != 0 && MEM_WB_RDaddr_i ==
ID_EX_RS1addr_i)
        select_1_reg = 2'b01; // select data3, from MEM_WB
    else
        select_1_reg = 2'b00; // select data1, from ID_EX
```

There are 3 multiplexers for ALU's data :

1. Forward A selects → MUX_ALU_data1 → ALU data input 1
2. Forward B selects → MUX_ALU_data2 → MUX_ALU_data2_src : selects between register RS2 or immediate → ALU data input 2

(C) Branch Taken (Prediction Wrong) : Known In ID Stage, By :

If both (RS1_data == RS2_data) && (it's the beq instruction now) ,

```
And And(
    // directly from Control
    .data1_i (Branch),
    .data2_i ((RS_Data1 == RS_Data2)? 1'b1 : 1'b0),
    .data_o (Branch_taken)
);
```

Note that the "Branch" control signal directly comes from the "Control" module, not from the "MUX_Control" module.

Then we have the “Branch Taken (Prediction Wrong) of beq” , and :

(1) In IF_ID Pipeline Register, flush the instruction after the “beq” instruction :

“beq” now in ID Stage, flush the instruction in IF Stage

```
IF_ID IF_ID(
    //TODO : Magic : For IF_ID.v : cannot write inst_o at PC=0. At PC=0, inst_o
shall be 32'b0 , since 1st instruction is still in IF stage , and no one is in ID
stage !! So IF_IF.inst_o shall be 32'b0 at that time !!
    // 1. Method For Fixing : Utilize the signal `start_i` of CPU.v .
    .start_i (start_i),

    .clk_i (clk_i),
    .IF_ID_flush_i (Branch_taken_reg),
    .IF_ID_write_i (IF_ID_write_reg),
```

Note that “IF_ID_flush_i” is for “Branch Taken (Prediction Wrong)” : output result of the “And” module above
, while “IF_ID_write_i” is for “Load-Use Hazard” : output result of the “HazardDetection” module.

(2) “MUX_PC” selects the branch address as the next PC.

Shift Left 1 For Branch Addressing :

[New PC] = [PC] + 2 * [Immediate]

Immediate From Instruction → Sign Extend → Shift Left 1 → Add_PC_Branch

And in testbench.v :

```
    if(CPU.Branch_taken_reg == 1)flush = flush + 1; //TODO : indicating : (1) it's
beq and Branch Taken (Prediction Wrong) (2) known in ID stage, so flush (set IF_ID
instruction register to be 32'b0) instruction after "beq" , who is now in IF stage
(and "beq" is now in ID stage)
```

(D) Load-Use Hazard : Known In ID Stage, By HazardDetection.v :

The conditions depend on the instructions in the EX stage & the ID stage :

```
    if( ID_EX_MemRead_i && ( ID_EX_RDaddr_i == IF_ID_RS1addr_i || ID_EX_RDaddr_i ==
IF_ID_RS2addr_i ) )
        Stall_o_reg = 1'b1;
    else
        Stall_o_reg = 1'b0;
```

If it's Load-Use Hazard, then

```
assign MUX_Control_select_o = Stall_o;
assign PC_write_o = (Stall_o == 1'b1)? 1'b0:1'b1;
assign IF_ID_write_o = (Stall_o == 1'b1)? 1'b0:1'b1;
```

(1) MUX_Control sets control signals as zero in ID_EX Pipeline Register :

So the "Use Instruction" (that is currently in ID Stage) of "Load-Use" won't go to the next stage : EX Stage.

(2) "Freeze" Writing of IF_ID Pipeline Register :

So the "Use Instruction" (that is currently in ID Stage) of "Load-Use" remains in ID Stage.

```
IF_ID IF_ID(
    //TODO : Magic : For IF_ID.v : cannot write inst_o at PC=0. At PC=0, inst_o
shall be 32'b0 , since 1st instruction is still in IF stage , and no one is in ID
stage !! So IF_IF.inst_o shall be 32'b0 at that time !!
    // 1. Method For Fixing : Utilize the signal `start_i` of CPU.v .
    .start_i (start_i),

    .clk_i (clk_i),
    .IF_ID_flush_i (Branch_taken_reg),
    .IF_ID_write_i (IF_ID_write_reg),
```

Note that "IF_ID_flush_i" is for "Branch Taken (Prediction Wrong)" : output result of the "And" module above
, while "IF_ID_write_i" is for "Load-Use Hazard" : output result of the "HazardDetection" module.

As in IF_ID Pipeline :

```
if(IF_ID_write_i && start_i) begin
    PC_o <= PC_i;
    //TODO : why this won't have x initially ?
ranch_taken_reg in CPU.v , who is initialized
    inst_o <= (IF_ID_flush_i)?32'b0 : inst_i;
end
```

(3) “Freeze” writing of pc in PC :

For the instructions after the “Use Instruction” of “Load-Use”

(4) The “Load Instruction” of “Load-Use” keeps moving to MEM Stage. It won’t be influenced.

(5) Then finally the “Load Instruction” of “Load-Use” gets data from Data_Memory, and the “Use Instruction” of “Load-Use” can now use it.

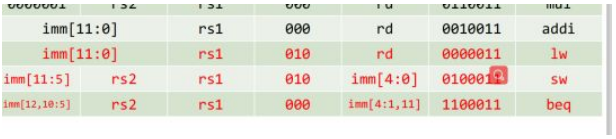
And in testbench.v :

```
    if(CPU.HazardDetection.Stall_o == 1 && CPU.Control.Branch_o == 0)stall = stall
+ 1; //TODO : " && Branch_o == 0 " to indicate that (1) it's not beq nor Branch
Taken (Prediction Wrong) (2) it's load-use hazard !! (3) known in ID stage by
HazardDection.v , so MUX_Control gives ID_EX all-zero control signals for the "use
instruction" (not the "load instruction" !!) of "load-use hazard"
```

(E) As In This Project’s Requirement , We Can Omit Data Hazard for beq.

(2) Modules Implementation

Module	Implementation
MUX32	input [31 : 0] data1_i; input [31 : 0] data2_i; input select_i; output [31 : 0] data_o; 選擇data1或data2作為data_o 1’b0 : data1 1’b1: data2
MUX3	input [31 : 0] data1_i; input [31 : 0] data2_i; input [31 : 0] data2_i; input [1 : 0] select_i; output [31 : 0] data_o; 選擇data1 or data2or data3作為data_o 2'b00: data1_i; 2'b01: data2_i;

<table>
<tr><td></td><td>2'b10: data3_i;
用於根據forward信號選擇不同的ALU_data</td></tr>
<tr><td>Sign_Extend</td><td>input [31 : 0] inst_i;
output [31 : 0] imm_o;
根據不同的指令類型輸出正確的immediate，參照下表
 
<table><tr><td>imm[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>0010011</td><td>addi</td></tr><tr><td>imm[11:0]</td><td>rs1</td><td>010</td><td>rd</td><td>0000011</td><td>lw</td></tr><tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>010</td><td>imm[4:0]</td><td>0100011</td><td>sw</td></tr><tr><td>imm[12,10:5]</td><td>rs2</td><td>rs1</td><td>000</td><td>imm[4:1,11]</td><td>1100011</td><td>beq</td></tr></table></td></tr>
<tr><td>Control</td><td>input [6 : 0] Op_i;
output [1:0] ALUOp_o;
output ALUSrc_o;
output Branch_o;
output MemRead_o;
output MemWrite_o;
output RegWrite_o;
output MemtoReg_o;
根據七位Op_i 輸出control signal,參見下圖：
<table><tr><th></th><th>ALUOp</th><th>ALUSrc</th><th>Branch</th><th>Mem-Read</th><th>Mem-Write</th><th>Reg-Write</th><th>Memto-Reg</th></tr><tr><td>R-format</td><td>10</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>ld</td><td>00</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>sd</td><td>00</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>X</td></tr><tr><td>beq</td><td>01</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>X</td></tr></table></td></tr>
<tr><td>IF_ID</td><td>input clk_i;
input IF_ID_flush_i;
input IF_ID_write_i;
input [31:0] PC_i;
input [31:0] inst_i;
output reg [31:0] PC_o;
output reg [31:0] inst_o;
根據IF_ID_write_i決定是否需要更新reg中的值，如果為0說明不需要更新。
如果為1，但IF_ID_flush_i為1，說明需要flush,此時將instruction置為32b'0，其他情況正常讀出inst_i的值。</td></tr>
<tr><td>Forward</td><td>input [4 : 0] ID_EX_RS1addr_i;
input [4 : 0] ID_EX_RS2addr_i;
input [4 : 0] EX_MEM_RDaddr_i;
input [4 : 0] MEM_WB_RDaddr_i;</td></tr>
</table>

	input EX_MEM_RegWrite_i; input MEM_WB_RegWrite_i; output [1 : 0] ForwardA; output [1 : 0] ForwardB; 如果EX_MEM中的RDaddr和ID_EX中的RS1addr或RS2addr相等，代表出現EXhazard,將select signal置為10。 如果MEM_WB中的RDaddr和ID_EX中的RS1addr或RS2addr相等，出現MEM hazard代表出現EXhazard,將select signal置為01。 其他情況無需forward,置為00。
Adder	input[31:0] data1_i input[31:0] data2_i; output[31:0] data_o; 將data1_i和data2_i相加並輸出
MUX_Control	input MUX_Control_select_i; input [1:0] ALUOp_i; input ALUSrc_i; input Branch_i; input MemRead_i; input MemWrite_i; input RegWrite_i; input MemtoReg_i; output [1:0] ALUOp_o; output ALUSrc_o; output Branch_o; output MemRead_o; output MemWrite_o; output RegWrite_o; output MemtoReg_o; 接收來自Control的control signal，若select_i的值為 1’b0，則讓control signal直接輸出；若 select_i的值為 1’b1，則表示CPU需要stall，要將control signal全部設為0再輸出 。
HazardDetection	input [4:0] IF_ID_RS1addr_i; input [4:0] IF_ID_RS2addr_i;

	input [4:0] ID_EX_RDaddr_i;
	input ID_EX_MemRead_i;
	output Stall_o;
	output MUX_Control_select_o;
	output PC_write_o; // 1'b1 to write, 1'b0 not to write
	output IF_ID_write_o; // 1'b1 to write, 1'b0 not to write

ID_EX	// write at posedge clock
	input clk_i;
	input [9:0] funct_i; // {funct7, funct3}
	output reg [9:0] funct_o; // {funct7, funct3}
	// EX stage
	input [1:0] ALUOp_i;
	output reg [1:0] ALUOp_o;
	input ALUSrc_i;
	output reg ALUSrc_o;
	// MEM stage
	input MemRead_i;
	output reg MemRead_o;
	input MemWrite_i;
	output reg MemWrite_o;
	// WB stage
	input RegWrite_i;
	output reg RegWrite_o;
	input MemtoReg_i;
	output reg MemtoReg_o;
	// register addr
	input [4:0] RS1addr_i;
	output reg [4:0] RS1addr_o;
	input [4:0] RS2addr_i;
	output reg [4:0] RS2addr_o;
	input [4:0] RDaddr_i;
	output reg [4:0] RDaddr_o;
	i

EX_MEM	// write at posedge clock
	input clk_i;
	// MEM stage

	input MemRead_i;
	output reg MemRead_o;
	input MemWrite_i;
	output reg MemWrite_o;
	// MEM stage additional
	input [31:0] ALU_Result_i;
	output reg [31:0] ALU_Result_o;
	// WB stage
	input RegWrite_i;
	output reg RegWrite_o;
	input MemtoReg_i;
	output reg MemtoReg_o;
	// register addr
	input [4:0] RDaddr_i;
	output reg [4:0] RDaddr_o;

MEM_WR	// write at posedge clock
	input clk_i;
	//
	input [31:0] ALU_Result_i;
	output reg [31:0] ALU_Result_o;
	// WB stage
	input RegWrite_i;
	output reg RegWrite_o;
	input MemtoReg_i;
	output reg MemtoReg_o;
	// WB additional
	input [31:0] memory_data_i;
	output reg [31:0] memory_data_o;
	// register addr
	input [4:0] RDaddr_i;
	output reg [4:0] RDaddr_o;

Data_Memory	// Interface input clk_i; input [31:0] addr_i; input MemWrite_i; input [31:0] data_i;

	output [31:0] data_o; // data memory reg [31:0] memory [0:1023]; 沿用助教給的Data_Memory.v

Instruction_Memory	// Interface
	input [31:0] addr_i;
	output [31:0] instr_o;
	// Instruction memory
	reg [31:0] memory [0:255];
	沿用助教給的Instruction_Memory.v

PC	// Ports
	input clk_i;
	input rst_i;
	input start_i;
	input PCWrite_i;
	input [31:0] pc_i;
	output [31:0] pc_o;
	
	// Wires & Registers
	//reg [31:0] pc_o;
	reg [31:0] pc_o = 32'b0;
	沿用助教給的PC.v

Registers	/ Ports input clk_i; input [4:0] RS1addr_i; input [4:0] RS2addr_i; input [4:0] RDaddr_i; input [31:0] RDdata_i; input RegWrite_i; output [31:0] RS1data_o; output [31:0] RS2data_o; // Register File reg [31:0] register [0:31]; 沿用助教給的Registers.v
ALU	input [31:0] data1_i; input [31:0] data2_i; input [3:0] ALUCtrl_i; output reg [31:0] data_o; output Zero_o; // not used

<table>
<tr><td></td><td>According to ALUCtrl_i, determine the computation for ALU.
<table><tr><td>ALUCtrl_i</td><td>Operation</td></tr><tr><td>4'b0010 / default</td><td>data1_i + data2_i</td></tr><tr><td>4'b0110</td><td>data1_i - data2_i</td></tr><tr><td>4'b0000</td><td>data1_i & data2_i</td></tr><tr><td>4'b0001</td><td>data1_i | data2_i</td></tr><tr><td>4'b0111</td><td>data1_i * data2_i</td></tr></table>Compute the result of data1_i and data2_i , and output data_o.</td></tr>
<tr><td>ALU_Control</td><td>input [9:0] funct_i; // instruction[31:25, 14:12] {funct7, funct3}
input [1:0] ALUOp_i;
output [3:0] ALUCtrl_o;

funct_i = concatenation of {funct7, funct3} fields of instruction.

ALUOp_i :
(1)2’b00 : ALU does addition for : lw, sw
(2)2’b01 : ALU does subtraction for : beq
(3)Otherwise, ALU depends on {funct7, funct3} fields : R-type

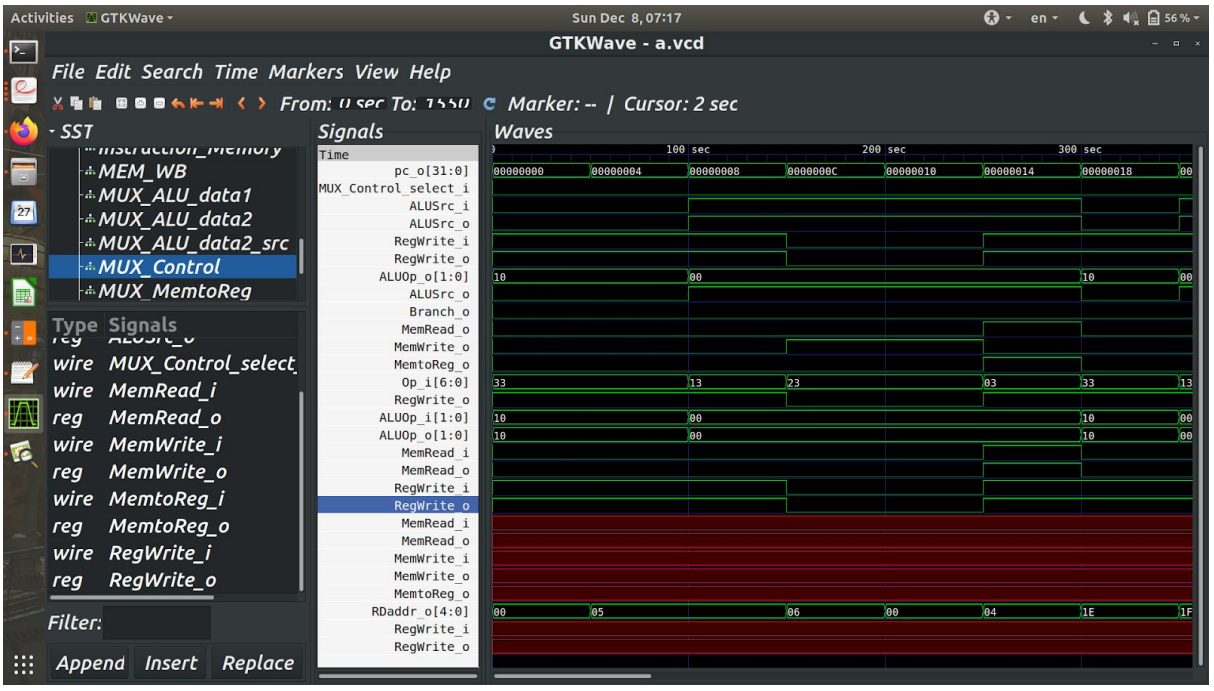
Then output the ALUCtrl_o for ALU.</td></tr>
<tr><td>And</td><td>If both (RS1_data == RS2_data) && (it’s the beq instruction now) ,</td></tr>
</table>

	then we have the “Branch Taken (Prediction Wrong)” , and we shall do the flushing.

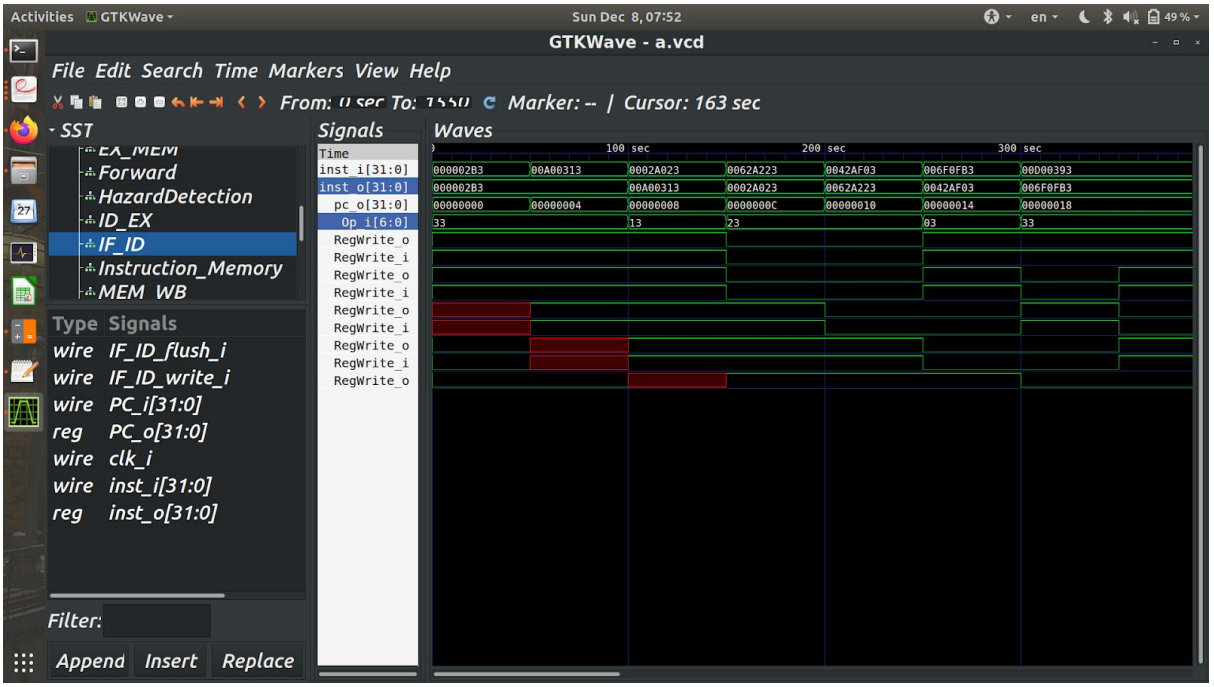
3. Difficulties Encountered & Solutions of This Project

(A) Initialize Pipeline Registers :

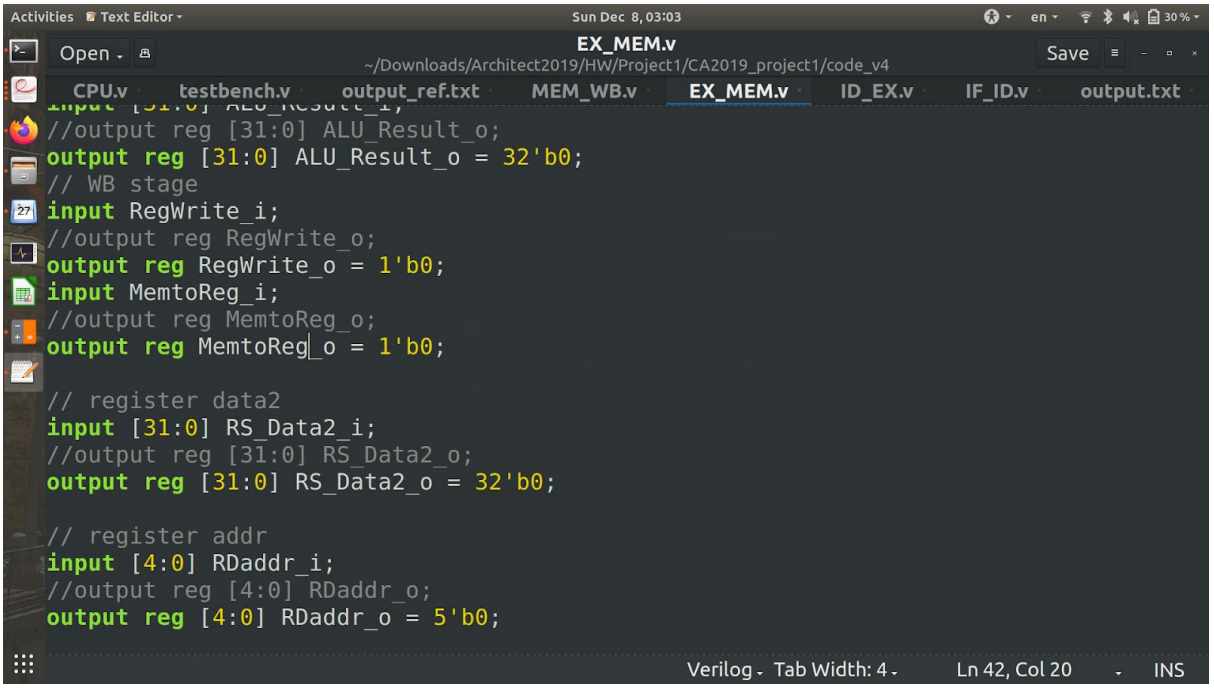
If not, then there will be signals x (red in the picture), and that will create wrong results.
And since the Pipeline CPU depends on signals in back stages, the CPU might not even work.



We must initialize possibly all signals / registers to zero, or the red x signals will propagate down through pipelines as :

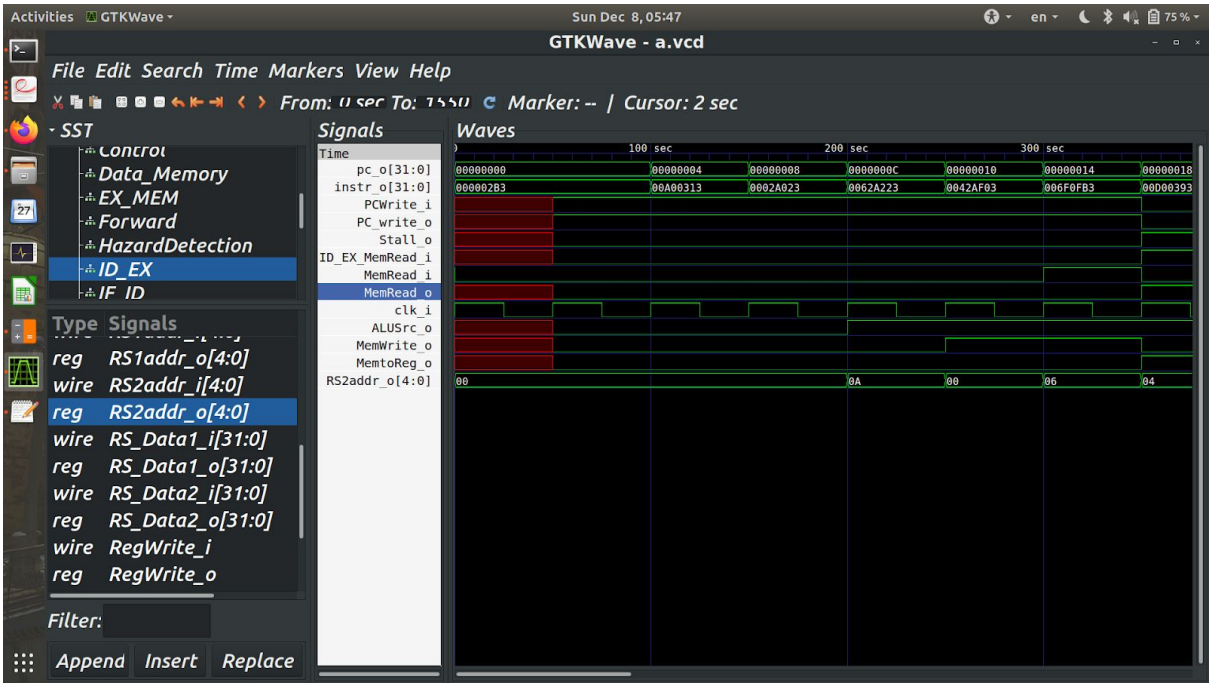


And we initialize them with the “initial” block.
We don’t know if the following works for Verilog (so we still used the “initial” block) :



(B) start_i For Each Pipeline :

Now we have initialized all signals / registers in pipelines, but we still got red x initially. Why ?



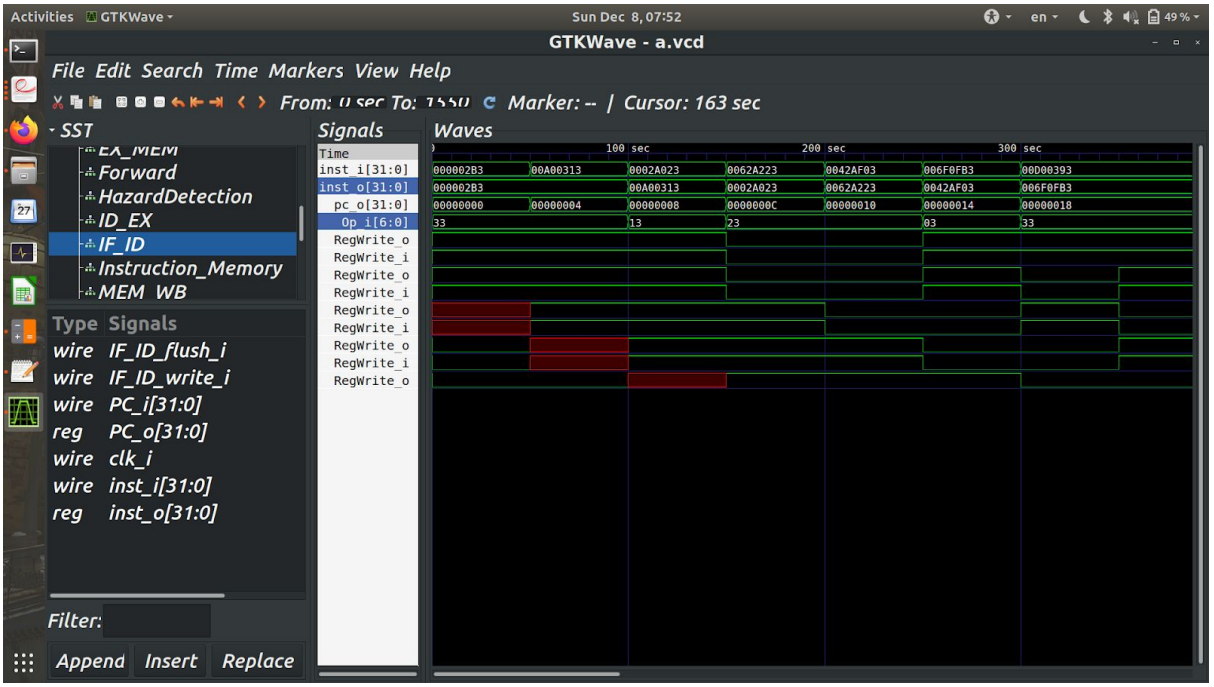
The reason is that : The input wires of pipeline registers are red x initially, then in the first PC, we write pipeline register on positive edge clock with the red x in the input wires !!!

Therefore, the solution is that : If start_i is 0 (start_i is 1 after ¼ cycle time), then don't write the pipeline register initially !!

(C) start_i For IF_ID Pipeline :

Now we initialized all registers to be zero in pipelines.

At first cycle, the first instruction shall still be in IF stage, so IF_ID pipeline output register "inst_o[31:0]" shall be 32'b0 , not the first instruction as the following gtkwave !!
Why ?!



The reason is : The input wire is from Instruction Memory, whose inst_o may be instruction[pc = 0] now.
So the solution is : only write the pipeline register “inst_o[31:0]” only after start_i = 1.

(D) Must Initialize PC Register :

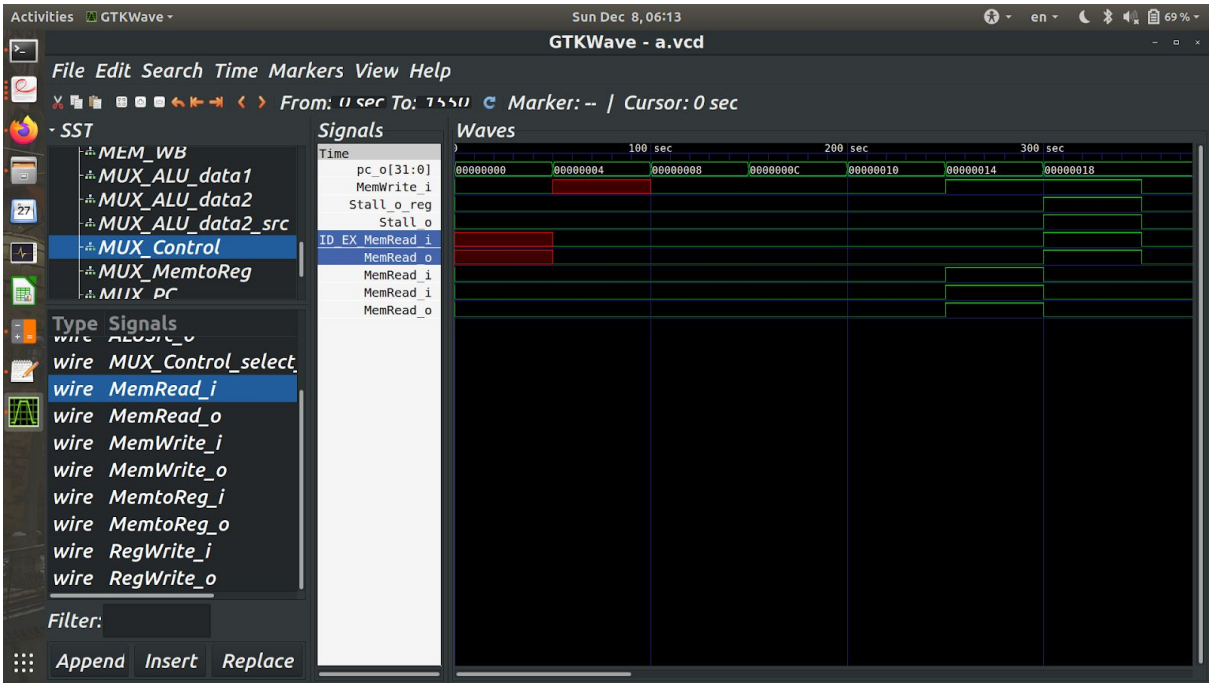
If not, then program counter will be x all the time.

(E) Initialize Control Signal Wires In CPU.v :

These wires in CPU.v are not registers, so we can’t initialize them as with registers in pipelines. Our solution is to use a register for these wires, so we can initialize them.

(F) The Conditional Operator `()? : `, Or The `if...else...` :

But...our control signals still contain the red x signals…
And the signals seem to come from the control modules like HazardDetection.v , not from the pipelines.



Then we changed the Conditional Operators

`output_signal = (condition)? choice 1 : choice 2 ` to the

`if(condition) choice 1; else choice 2;` block.

Our hypothesis is that : If the condition contains the red x signal, we expect the choice 2 be chosen.
However, the Conditional Operator seems to also assign the red x signal to its left hand side output_signal.
And the `if...else...` block seems to work as we expect : The choice 2 is chosen.

(G) The Red x Signals Are Crucial :

After eliminating all the red x signals, our output result is correct on the test data.

(H) Debug By Observing Signals in gtkwave :

Yes, painful !!! But we have to debug by observing signals at certain cycles.

(I) Tips With Verilog : “inout port” May Cause Wrong Signals ? :

In CPU.v , an output port of the output module :

.output_port(input_module.input_port);

(1)This may cause wrong signals :

.input_port(output_module.output_port);

(2)It works fine if we leave input port of the input module blank :

.input_port( );

(J) Many Hazards : EX Data Hazard, MEM Data Hazard, Load-Use Data Hazard, Branch Taken (Prediction Wrong) :

Also check the Forwarding signals in gtkwave.

(1)PC=8 : MEM hazard in A (ALU.data2 for immediate) for x5

(2)PC=12 : MEM hazard in B (ALU.data2 for immediate) for x6 (x6 uses rs2, while x5 uses rs1)

(3)PC=16 : no data hazard for x5 in !!

(4)PC=20 : Load-Use hazard for x30 in (and we have MEM hazard in A)

(5)PC=32 : Branch taken , and flush next instruction , which is already loaded !!

(6)(not used here, since it will select branch ; if not jumped : PC=40 EX hazard for x6)

(7)PC=44 : (not used here : if it's not jumped, then EX hazard in for x6 (not MEM hazard for x6 !!)) --> But !! Jumped from PC=32, so no data hazard for x6 !! And !! No data hazard for Branch taken (prediction wrong) , since its last 2 instructions are beq and nop.

(8)PC=48 : no data hazard for x6. But EX hazard in B for x7

(9)PC=52 : MEM hazard in A for x7. EX hazard in B for x28.

(K) Wrong Output Result With Github codes :

https://github.com/jnfem112/Computer-Architecture

https://github.com/jnfem112/Computer-Architecture/tree/master/Project%201

Their output result with the test data is not correct. So don’t copy codes from them. :)