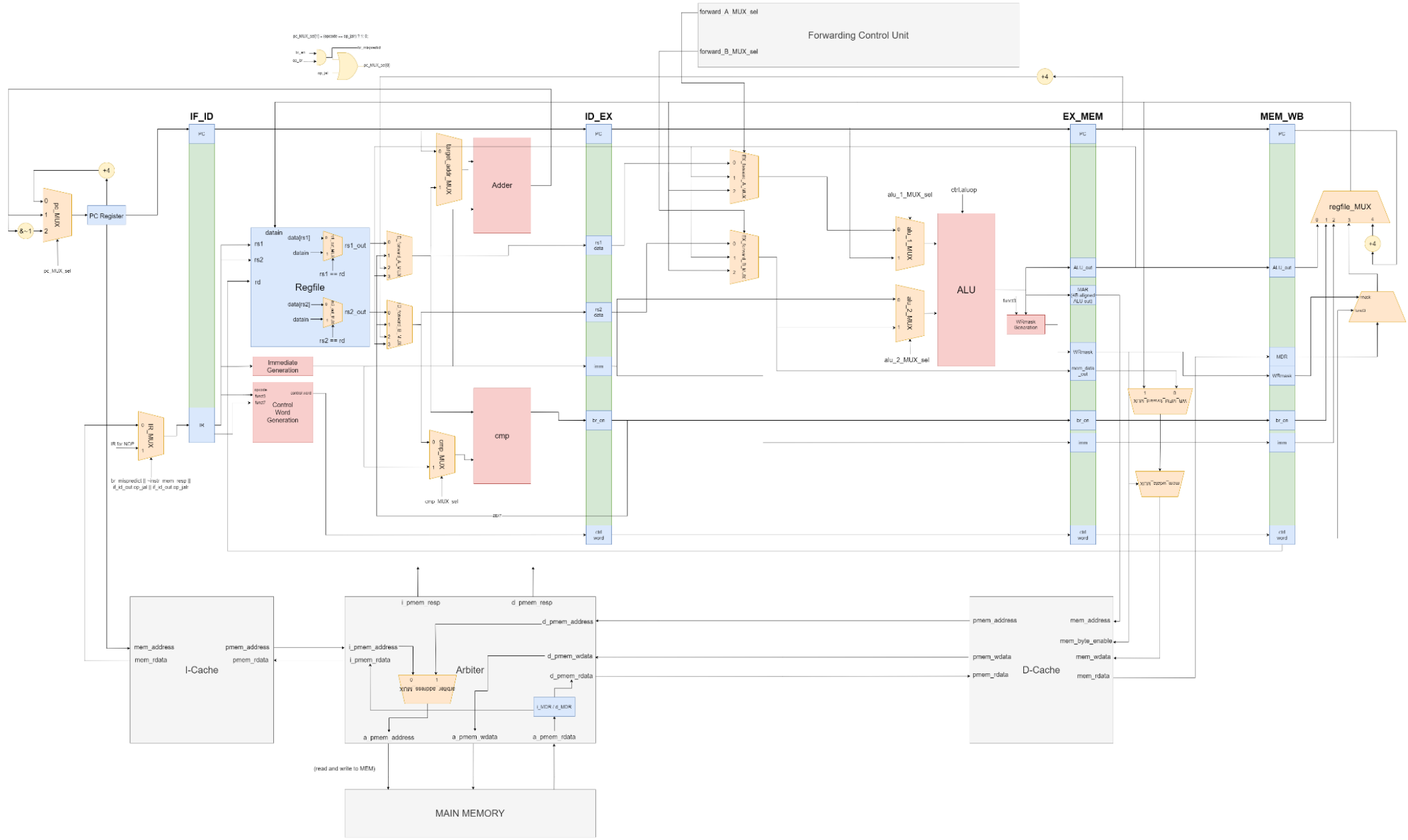
MP4 CP2 Design Document

HighRISCHighReward

Datapath Diagram



Forwarding Control Unit Logic

```
// MEM-EX Forward Path
ex_forward_A_MUX = 2'b00;
ex_forward_B_MUX = 2'b00;
mem_ex_v_A = 1'b0;
mem_ex_v_B = 1'b0;
mem_wb_v_A = 1'b0;
mem_wb_v_B = 1'b0;
if ex_mem_out.ctrl.load_regfile && (ex_mem_out.ctrl.rd_id == id_ex_out.ctrl.rs1_id) && ex_mem_out.ctrl.rd_id != 0
begin
      if (ex_mem_out.ctrl.opcode != op_load):
            mem_ex_v_A = 1'b1;
end
if ex_mem_out.ctrl.load_regfile && (ex_mem_out.ctrl.rd_id == id_ex_out.ctrl.rs2_id) && ex_mem_out.ctrl.rd_id != 0
begin
      if (ex_mem_out.ctrl.opcode != op_load)
            mem_ex_v_B = 1'b1;
end
// WB-EX Forward Path
if mem_wb_out.ctrl.load_regfile && (mem_wb_out.ctrl.rd_id == id_ex_out.ctrl.rs1_id) && mem_wb_out.ctrl.rd_id != 0
begin
      mem_wb_v_A  = 1'b1;
end
if mem_wb_out.ctrl.load_regfile && (mem_wb_out.ctrl.rd_id == id_ex_out.ctrl.rs2_id) && mem_wb_out.ctrl.rd_id != 0:
begin
      mem_wb_v_B = 1'b1;
end
```

```
if (mem_ex_v_A && mem_wb_v_A)
	ex_forward_A_MUX = 01
else if (mem_ex_v_A)
	ex_forward_A_MUX = 01
else if(mem_wb_v_A)
	ex_forward_A_MUX = 10

if (mem_ex_v_B && mem_wb_v_B)
	ex_forward_B_MUX = 01
else if (mem_ex_v_B)
	ex_forward_B_MUX = 01
else if(mem_wb_v_B)
	ex_forward_B_MUX = 10
```

```
// MEM-ID Forward Path
id_forward_A_MUX = 2'b00;
id_forward_B_MUX = 2'b00;
if ex_mem_out.ctrl.load_regfile && (ex_mem_out.ctrl.rd_id == id_ex_in.ctrl.rs1_id) && ex_mem_out.ctrl.rd_id != 0
begin
	if (if_id_out.ir[6:0] == op_br || op_jal || op_jalr):
		id_forward_A_MUX = 1'b1
	if (if_id_out.ir[6:0] == op_imm && if_id_out.ir[14:12] == slt || sltu):
		id_forward_A_MUX = 1'b1
	if (if_id_out.ir[6:0] == op_reg && if_id_out.ir[14:12] == slt || sltu):
		id_forward_A_MUX = 1'b1
end
if ex_mem_out.ctrl.load_regfile && (ex_mem_out.ctrl.rd_id == id_ex_in.ctrl.rs2_id) && ex_mem_out.ctrl.rd_id != 0
begin
```

```
       if (if_id_out.ir[6:0] == op_br || op_jal || op_jalr):
              id_forward_B_MUX = 1'b1
       if (if_id_out.ir[6:0] == op_reg && if_id_out.ir[14:12] == slt || sltu):
              id_forward_B_MUX = 1'b1
end
```

```
// WB-MEM Forwarding Path
wb_mem_forward_MUX_sel = 1'b0;
if mem_wb_out.ctrl.load_regfile && (mem_wb_out.ctrl.rd_id == ex_mem_out.ctrl.rs2_id) && mem_wb_out.ctrl.rd_id != 0
begin
       if(mem_wb_out.ctrl.opcode == op_load && ex_mem_out.ctrl.opcode == op_store)
              wb_mem_forward_MUX_sel = 1'b1;
end
```

Stalling Condition

Stalling after load:

1. Store after load (rs1 is written by load) (1 cycle stall b/c we have wb_ex forwarding)
    a. For a ST instruction, RS1 needs to be ready by EX stage but RS2 needs to be ready by MEM stage.
2. Arithmetic instruction after load (1 cycle stall b/c we have wb_ex forwarding)
3. Branch after load (2 cycles stall b/c we need the updated register value in decode stage for CMP) (no forwarding required)

Branch after arithmetic: 1 cycle stall b/c MEM-ID stage forwarding

Instruction Cache Miss

The architectural PC register will not be loaded until I-cache raises instr_mem_resp signal. The IR_MUX will be set to 1 so that for each cycle the instr_mem_resp is low, a NOP will be inserted and propagated through the pipeline.

Data Cache Miss

The architectural PC register and all 4 pipeline registers will not be loaded until D-cache raises data_mem_resp.

Arithmetic Instruction After Load

Check if the instruction in MEM is a load and if the instruction in EX is op_reg or op_imm and if there is a match of RD with RS. If so, do not load the architectural PC register, if_id_pipeline_register, and id_ex_pipeline_register. Insert one NOP into ex_mem_pipeline_register (flush). In the cycle after, the arithmetic instruction is still in EX, a NOP is in MEM, and the load is in WB. Then we can forward WB to EX.

Branch, JALR, (or any operation using comparator) after Load

Check if the instruction in EX is a load and if the instruction in ID is op_branch or op_jalr or SLT or SLTI and if there is a match of RD with RS1 (for jalr) or RS2. If so, do not load the architectural PC register, if_id_pipeline_register. First, insert one NOPs into

id_ex_pipeline_register (flush). Then, we will check if there is a NOP in EX, a load in MEM and branch in ID, we put the second NOP in ID. We also check if there is a NOP in EX, a NOP in MEM and Branch in ID, we continue loading the if_id_pipeline_register and PC register. Then we can forward WB to ID.

Implicit no-op after JAL or JALR

After a JAL or JALR operation

Branch Mispredict Handling

The result of the branch instruction is resolved in the ID decode. Since the branch predictor always predicts NT, the branch misprediction occurs precisely when the branch is resolved to be Taken. When a branch is resolved in the ID stage and it is a mispredict, the next instruction is in the IF stage. Instead of loading the mem_rdata of the I-cache into the IR, we load the NOP instruction into the IR by hardwiring one input of the IR_MUX. This will create a bubble that shifts through each pipestage. The correct PC value will be set by PC_MUX.

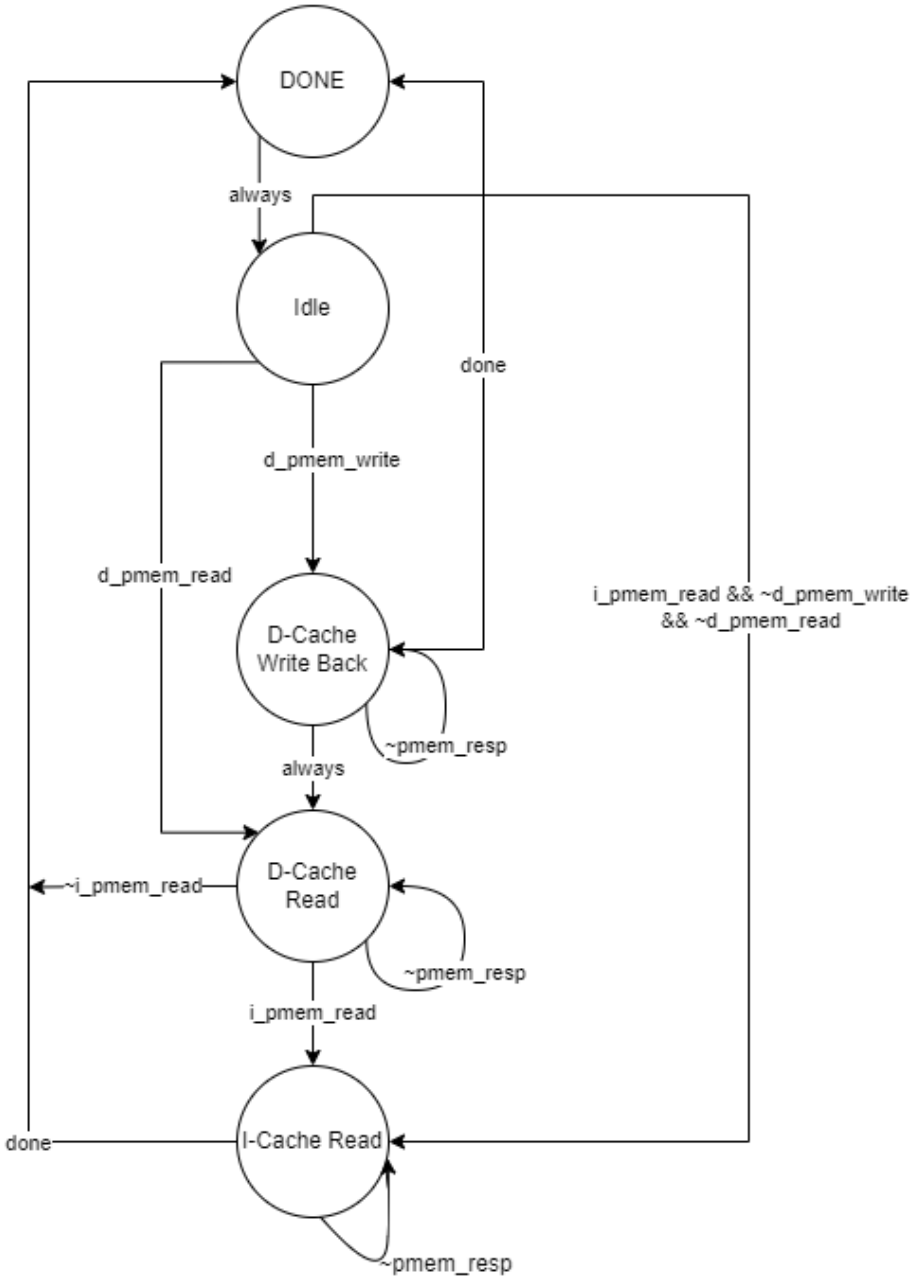
Important Notes:

```
LD $4, $2(100)
NOP
ST $4, $31(100)
```

It is necessary to use the WB-EX path. Usually for ST, you only need to forward RS2 in the MEM stage, but in this case, you need to use WB-EX path otherwise the data produced by the producer instruction will be gone from the pipeline. (This is a weird case where the timing constraint of forwarding is imposed by the Producer instruction.)

For a ST instruction, RS1 needs to be ready by EX stage but RS2 needs to be ready by MEM stage.

Arbiter State Description



State Name	State Description	Control Signals Set	State Transitions

Idle	Waiting for a memory request from the data or instruction caches		`if d_pmem_write:`
			`  Idle -> D-Cache Write Back`
			`else if d_pmem_read:`
			`  Idle -> D-Cache Read`
			`else if i_pmem_read:`
			`  Idle -> Instruction`

D-Cache Write Back	Handle requests from data cache to write back (when data cache miss and dirty)	arbiter_address_MUX_sel = 1	`if ~p_mem_resp:`
			`  D-Cache Write Back -> D-Cache Write Back`
			`else:`
			`  D-Cache Write Back -> Instruction`

D-Cache Read	Handle requests from data cache to read cache line from memory (when data cache miss and not dirty)	arbiter_address_MUX_sel = 1
		d_MDR_load = 1
			`if ~p_mem_resp:`
			`  D-Cache Read -> D-Cache Read`
			`else if i_pmem_read:`
			`  D-Cache Read -> I-Cache Read`
			`else:`
			`  D-Cache Read -> Idle`

I-Cache Read	Handle request from instruction cache to read cache line from memory	arbiter_address_MUX_sel = 0
		i_MDR_load = 1
			`if ~p_mem_resp:`
			`  I-Cache Read -> I-Ca`
			`else:`
			`  Instruction -> Idle`

DONE	The data and instruction access for the current instruction is completed. Raise response signal.	i_pmem_resp = 1 d_pmem_resp = 1	

- Control Signals are default to 0.