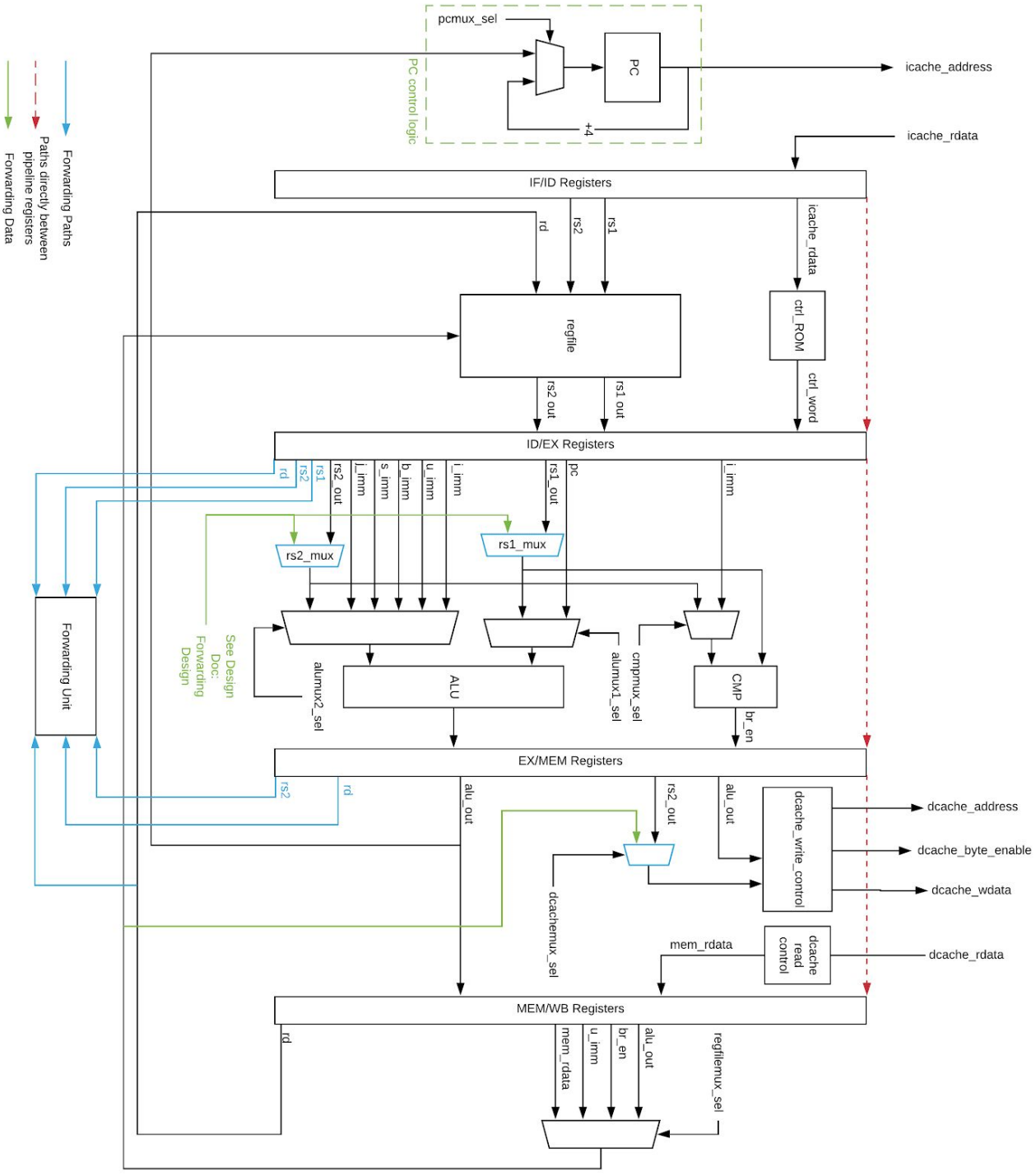
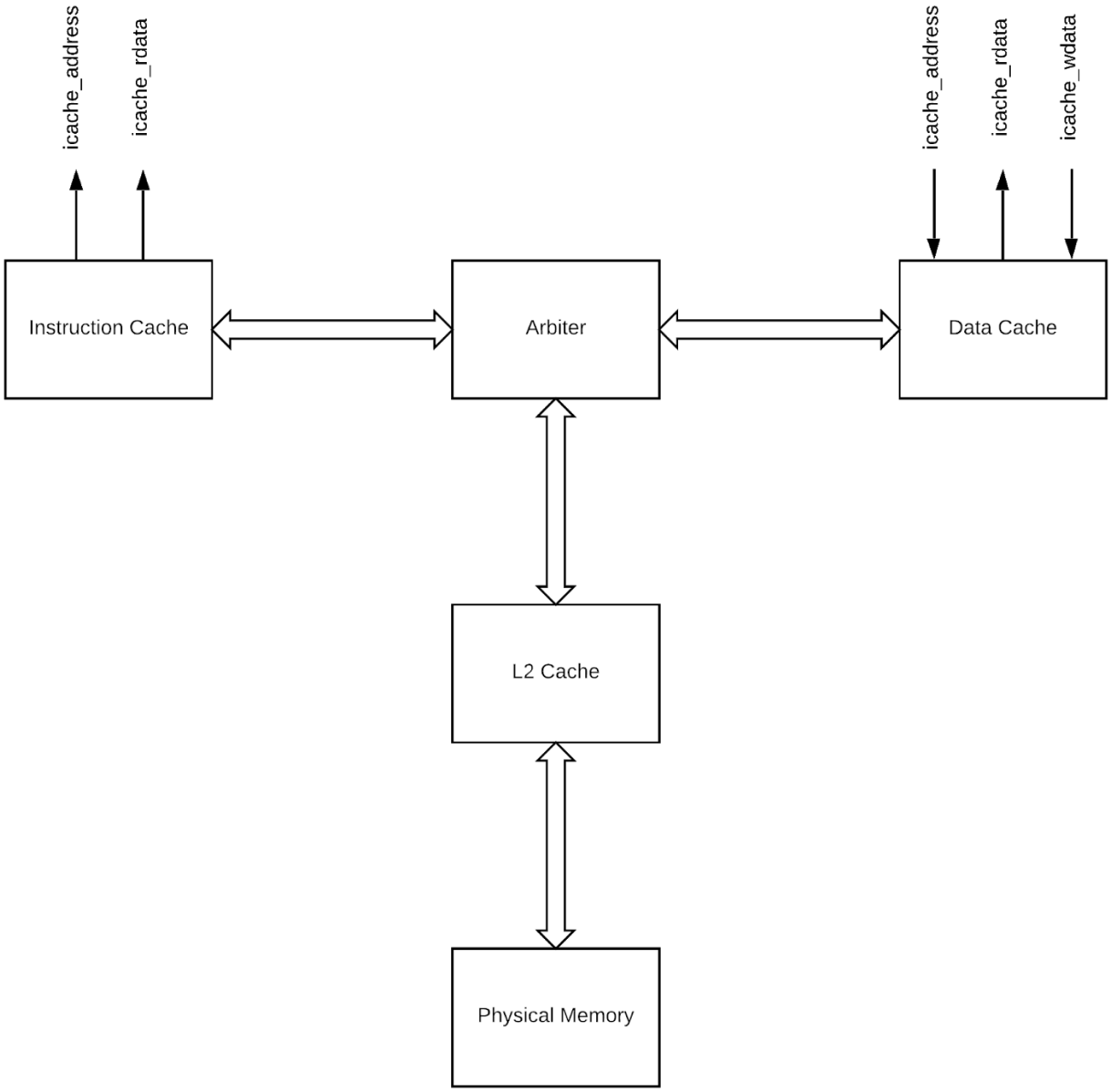
# ECE 411 MP3 Design Specification: Group ZBA

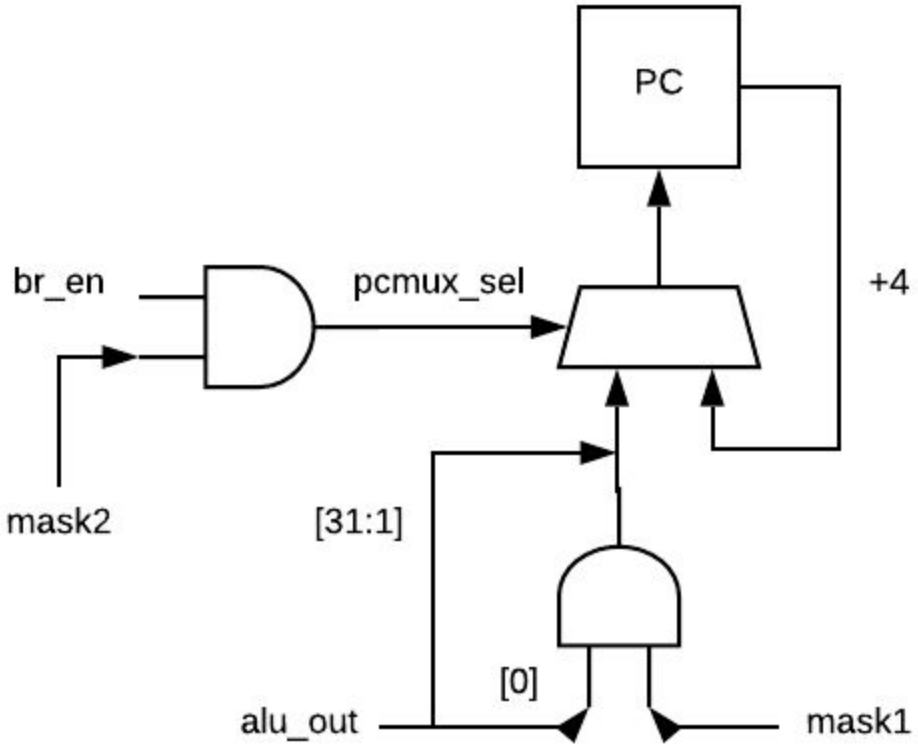
Zerui “Jerry” An, Austin Smithe, Brock Boehler


Forwarding Paths
Paths directly between pipeline registers
Forwarding Data
pcmux_sel
PC
+4
PC control logic
icache_address
icache_rdata
IF/ID Registers
rd
rs2
rs1
icache_rdata
regfile
ctrl_ROM
ctrl_word
rs2 out
rs1 out
ID/EX Registers
rd
rs2
rs1
rs2_out
i_imm
s_imm
b_imm
u_imm
i_imm
rs1_out
pc
i_imm
rs2_mux
rs1_mux
Forwarding Unit
See Design Doc: Forwarding Design
alumux2_sel
ALU
alumux1_sel
cmpmux_sel
CMP
br_en
EX/MEM Registers
rs2
rd
alu_out
rs2_out
alu_out
dcache_address
dcache_byte_enable
dcache_wdata
dcache_write_control
dcachemux_sel
mem_rdata
dcache read control
dcache_rdata
MEM/WB Registers
rd
mem_rdata
u_imm
br_en
alu_out
regfilemux_sel

# Memory Hierarchy



PC Control Logic




mask1 = ~(opcode == jalr)
mask2 = (opcode == br/jal/jalr)

# Pipeline Registers

IF/ID

| Register name | Description |
|---|---|
| icache_rdata | 32-bit instruction |
| PC | Program counter |

ID/EX

| Register name | Description |
|---|---|
| icache_rdata | 32-bit instruction |
| ctrl_word | A struct containing all control signals. Output of control ROM. |
| PC | Program counter |

EX/MEM

| Register Name | Description |
|---|---|
| alu_out | ALU output |
| rs2_out | Contents of register 2 |
| rd | Destination register |
| br_en | Branch enable |
| u_imm | Unsigned immediate from instruction data |
| ctrl_word | A struct containing all control signals. |

MEM/WB

| Register Name | Description |
|---|---|
| alu_out | ALU output |
| rd | Destination register |

| | |
|---|---|
| br_en | Branch enable |
| u_imm | Unsigned immediate from instruction data |
| MDR_out | Data received from data cache |
| ctrl_word | A struct containing all control signals.i |

## Control Signals

IF

| **Signal Name** | **Description** |
|---|---|
| load_pc | Load program counter |
| pcmux_sel | Selects between PC+4 or WB stage output |

ID

| **Signal Name** | **Description** |
|---|---|
| *None* | |

EX

| **Signal Name** | **Description** |
|---|---|
| rs1mux_sel | Output of forwarding unit. Used to select the signal to be forwarded. |
| rs2mux_sel | |
| alumux1_sel | ALU input 1 select |
| alumux2_sel | ALU input 2 select |
| cmpmux_sel | Chooses input to CMP module |
| aluop | ALU operation |
| cmpop | Compare operation |

MEM

| Signal Name | Description |
|---|---|
| dcachemux_sel | Selects input to write to data cache |

WB

| Signal Name | Description |
|---|---|
| regfilemux_sel | Chooses data to be written to regfile |
| load_regfile | Loads regfile |

## Information Shared Between Pipeline Registers

| IF/ID → ID/EX | ID/EX → EX/MEM | EX/MEM → MEM/WB |
|---|---|---|
| icache_rdata<br>PC<br>ctrl_word | rd<br>u_imm<br>ctrl_word | alu_out<br>rd<br>br_en<br>u_imm<br>ctrl_word |

## ctrl_word struct:

| Signal Name | Length | Stage |
|---|---|---|
| alumux1_sel | 1 | EX |
| alumux2_sel | 3 | |
| cmpmux_sel | 1 | |
| aluop | 3 | |
| cmpop | 3 | |
| dcache_read | 1 | MEM |

| | | |
|---|---|---|
| dcache_write | 1 | |
| mask1 | 1 | |
| mask2 | 1 | |
| regfilemux_sel | 3+ | WB |
| load_regfile | 1 | |

## Control Word Logic:

*Note: STB = See Table Below*

| **Signal Name** | **lui** | **auipc** | **jal** | **jalr** | **br** | **load** | **store** | **imm/reg** |
|---|---|---|---|---|---|---|---|---|
| alumux1_sel | x | pc | pc | rs1_out | pc | rs1_out | rs1_out | rs1_out |
| alumux2_sel | x | u_imm | j_imm | i_imm | b_imm | i_imm | s_imm | *STB* |
| cmpmux_sel | x | x | x | x | rs2_out | x | x | *STB* |
| aluop | x | alu_add | alu_add | alu_add | alu_add | alu_add | alu_add | *STB* |
| cmpop | x | x | x | x | funct3 | x | x | *STB* |
| dcache_read | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| dcache_write | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| regfilemux_sel | u_imm | alu_out | pc_plus4 | pc_plus4 | x | MDR_out | x | *STB* |
| load_regfile | 1 | 1 | 1 | 1 | 0 | 1 | 0 | *STB* |

| **Signal Name** | **imm(default)** | **imm(srai)** | **imm(slti)** | **imm(sltiu)** |
|---|---|---|---|---|
| alumux1_sel | rs1_out | | | |
| alumux2_sel | i_imm | i_imm | x | x |
| cmpmux_sel | x | x | i_imm | i_imm |
| aluop | funct3 | alu_sra | x | x |

| | | | | |
|---|---|---|---|---|
| cmpop | x | x | blt | bltu |
| dcache_read | 0 | | | |
| dcache_write | 0 | | | |
| regfilemux_sel | alu_out | alu_out | br_en | br_en |
| load_regfile | 1 | | | |

| **Signal Name** | **reg(default)** | **reg(sra)** | **reg(slt)** | **reg(sltu)** | **reg(sub)** |
|---|---|---|---|---|---|
| alumux1_sel | rs1_out | | | | |
| alumux2_sel | rs2_out | rs2_out | x | x | rs2_out |
| cmpmux_sel | x | x | rs2_out | rs2_out | x |
| aluop | funct3 | alu_sra | x | x | alu_sub |
| cmpop | x | x | blt | bltu | x |
| dcache_read | 0 | | | | |
| dcache_write | 0 | | | | |
| regfilemux_sel | alu_out | alu_out | br_en | br_en | alu_out |
| load_regfile | 1 | | | | |

## Read / Write Logic

| **Instruction** | **dcache_address & 0x03** | **dcache_mem_byte_enable** |
|---|---|---|
| SW | x | 4’b1111 |
| SH | 0 | 4’b0011 |
| | 1 | 4‘b0110 |
| | 2 | 4’b1100 |
| | 3 | x |

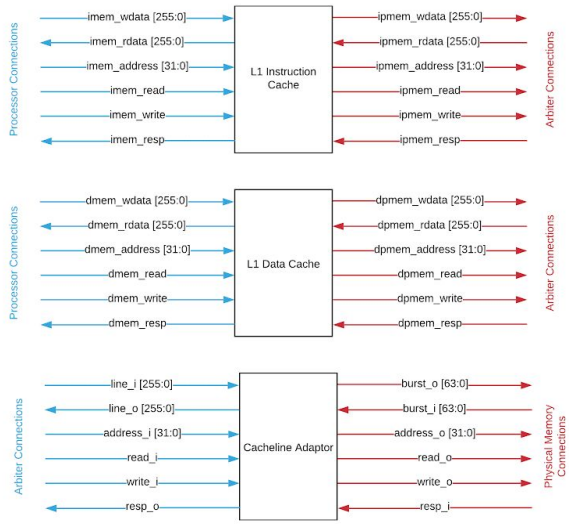
| | | |
|---|---|---|
| SB | 0 | 4’b0001 |
| | 1 | 4’b0010 |
| | 2 | 4’b0100 |
| | 3 | 4’b1000 |

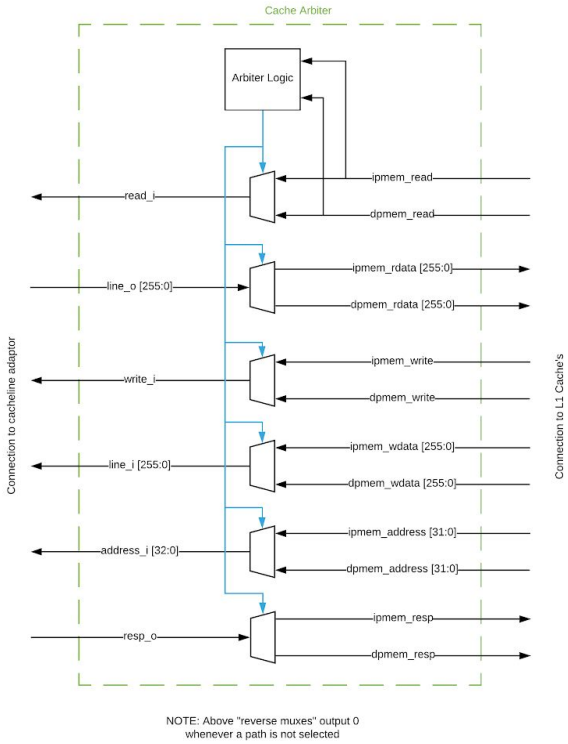
| **Instruction** | **dcache_wdata** |
|---|---|
| SW | `dcache_wdata <= rs2_out` |
| SH | `dcache_wdata <= (rs2_out << {alu_out[1:0]}, 3’d0})` |
| SB | `dcache_wdata <= (rs2_out << {alu_out[1:0]}, 3’d0})` |

| **instruction** | **dcache_ address & 0x03** | **MDRreg_out** |
|---|---|---|
| lw | x | `dcache_rdata` |
| lh | 0 | `{{16{dcache_rdata[15]}}, dcache_rdata[15:0]}` |
| | 1 | `{{16{dcache_rdata[23]}}, dcache_rdata[23:8]}` |
| | 2 | `{{16{dcache_rdata[31]}}, dcache_rdata[31:16]}` |
| | 3 | `x` |
| lhu | 0 | `{16’d0, dcache_rdata[15:0]}` |
| | 1 | `{16’d0, dcache_rdata[23:8]}` |
| | 2 | `{16’d0, dcache_rdata[31:16]}` |
| | 3 | `x` |
| lb | 0 | `{{24{dcache_rdata[7]}}, dcache_rdata[7:0]}` |
| | 1 | `{{24{dcache_rdata[15]}}, dcache_rdata[15:8]}` |
| | 2 | `{{24{dcache_rdata[23]}}, dcache_rdata[23:16]}` |
| | 3 | `{{24{dcache_rdata[31]}}, dcache_rdata[31:24]}` |

| | | |
|---|---|---|
| lbu | 0 | `{24'd0, dcache_rdata[7:0]}` |
| | 1 | `{24'd0, dcache_rdata[15:8]}` |
| | 2 | `{24'd0, dcache_rdata[23:16]}` |
| | 3 | `{24'd0, dcache_rdata[31:24]}` |

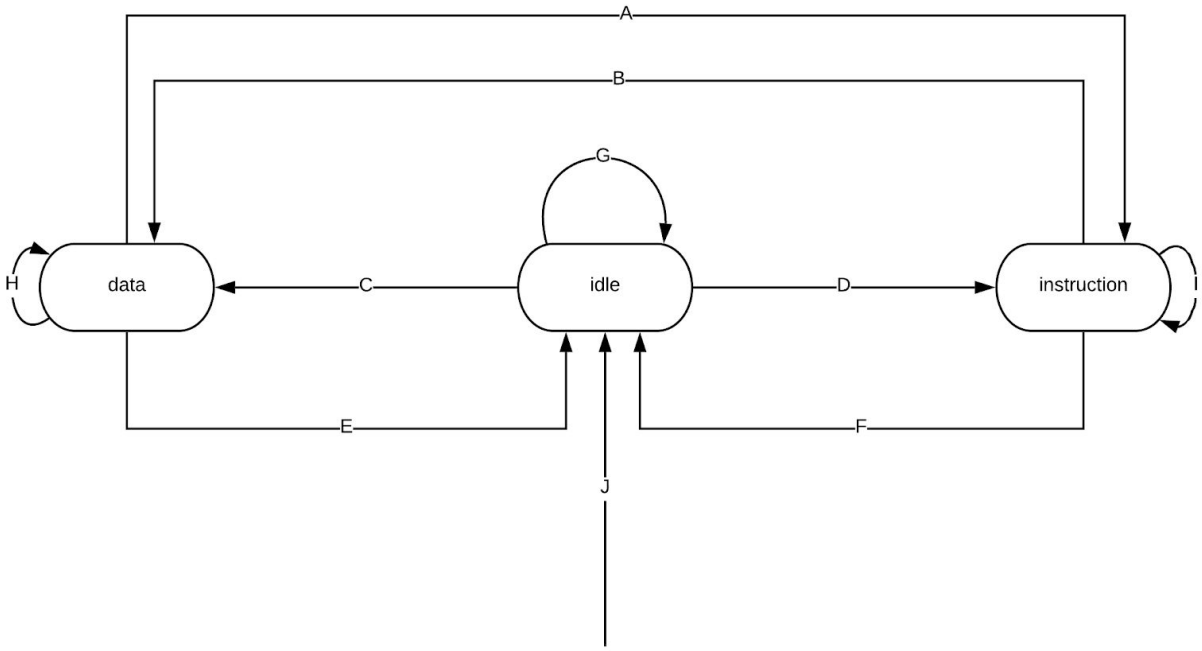
# Cache Arbiter




NOTE: Above "reverse muxes" output 0 whenever a path is not selected

# Cache Arbiter Control / State Machine




Note: The cache arbiter prioritizes the instruction cache over the data cache. Whenever both want to read at the same time it is the instruction cache that gets to read / write.

| Transition | Condition |
|---|---|
| A | `(dpmem_read == 1'b0 & dpmem_write == 1'b0) & (ipmem_read == 1'b1 \| ipmem_write = 1'b1) & (resp_o == 1'b1)` |
| B | `(ipmem_read == 1'b0 & ipmem_write == 1'b0) & (dpmem_read == 1'b1 \| dpmem_write == 1'b1) & (resp_o == 1'b1)` |
| C | `(dpmem_read == 1'b1 \| dpmem_write == 1'b1) & (ipmem_read == 1'b0 & ipmem_write == 1'b0)` |
| D | `(ipmem_read == 1'b1 \| ipmem_write == 1'b1)` |
| E | `(dpmem_read == 1'b0 & dpmem_write == 1'b0) & (ipmem_read == 1'b0 & ipmem_write == 1'b0)` |
| F | `(dpmem_read == 1'b0 & dpmem_write == 1'b0)` |

| | |
|---|---|
| | `(ipmem_read == 1'b0 & ipmem_write == 1'b0) &` |
| G | `(dpmem_read == 1'b0 & dpmem_write == 1'b0) &`<br>`(ipmem_read == 1'b0 & ipmem_write == 1'b0)` |
| H | `(dpmem_read == 1'b1 \| dpmem_write == 1'b1) &`<br>`(ipmem_read == 1'b0 & ipmem_write == 1'b0) &`<br>`(resp_o == 1'b1)` |
| I | `(ipmem_read == 1'b1 \| ipmem_write == 1'b1) &`<br>`(resp_o == 1'b1)` |
| J | `rst == 1'b1` |

| **State** | **Output** |
|---|---|
| idle | Don't Care |
| data | Muxes output data cache signals |
| instruction | Muxes output instruction cache signals. |

# Forwarding Design

There are hazard between the following stages:

**EX/MEM ⇒ EX**

```
if (
      exmem_load_regfile == 1
&&    exmem_opcode != load
&&    exmem_rd == idex_rs
)
```

Example:

```
ADD x1, x2, x2;
ADD x3, x1, x1;
```

**MEM/WB ⇒ EX**

```
if (
      memwb_load_regfile == 1
&&    memwb_rd == idex_rs
)
```

Example:

```
ADD x1, x2, x2;
NOP
ADD x3, x1, x1;

if (
      exmem_load_regfile == 1
&&    exmem_opcode == load
&&    !(idex_opcode == store && exmem_rd != idex_rs1)
&&    exmem_rd == idex_rs
)
```

Example(2):

```
LW x1, ADDR;
ADD x3, x1, x1; (1-cycle stall)
```

**MEM/WB ⇒ MEM**

Example:

```
LW x1, ADDR1;
SW x1, ADDR2;
```

The forwarding muxes, pictured in the datapath, have the following connections:

| **Forwarding Mux** | **Inputs** |
|---|---|
| rs1_mux | rs1_out |
| | EXMEM_ALU_out |
| | EXMEM_BR_EN |
| | regfilemux_out |
| | mem_rdata |
| | EXMEM_u_imm |
| rs2_mux | rs2_out |
| | EXMEM_ALU_out |
| | EXMEM_BR_EN |
| | regfilemux_out |
| | mem_rdata |

| | |
|---|---|
| | EXMEM_u_imm |
| dcache_mux | rs2_out |
| | regfilemux_out |

The hazards and their forwarding solutions are as follows:

**rs1_idex == rd_exmem**
Between Stages: EX/MEM ⇒ EX

| **EX/MEM Instruction** | **rs1_mux** |
|---|---|
| LUI | EXMEM_u_imm |
| AUIPC | EXMEM_ALU_out |
| branch | x |
| load | mem_rdata |
| store | x |
| SLTI | EXMEM_BR_EN |
| SLTUI | EXMEM_BR_EN |
| IMM (other) | EXMEM_ALU_out |
| SLT | EXMEM_BR_EN |
| SLTU | EXMEM_BR_EN |
| REG (other) | EXMEM_ALU_out |

**rs2_idex == rd_exmem**
Between Stages: EX/MEM ⇒ EX

| **EX/MEM Instruction** | **rs2_mux** |
|---|---|
| LUI | EXMEM_u_imm |
| AUIPC | EXMEM_ALU_out |
| branch | x |
| load | mem_rdata |

| | |
|---|---|
| store | x |
| SLTI | EXMEM_BR_EN |
| SLTUI | EXMEM_BR_EN |
| IMM (other) | EXMEM_ALU_out |
| SLT | EXMEM_BR_EN |
| SLTU | EXMEM_BR_EN |
| REG (other) | EXMEM_ALU_out |

**rs1_idex == rd_memwb**
Between Stages: MEM/WB ⇒ EX

| **MEM/WB Instruction** | **rs1_mux** |
|---|---|
| *Any* | regfilemux_out |

**rs2_idex == rd_memwb**
Between Stages: MEM/WB ⇒ EX

| **MEM/WB Instruction** | **rs2_mux** |
|---|---|
| *Any* | regfilemux_out |

**rs2_exmem == rd_memwb**
Between Stages: MEM/WB ⇒ MEM

| **MEM/WB Instruction** | **dcache_mux** |
|---|---|
| *Any* | regfilemux_out |

**Important Notes:**

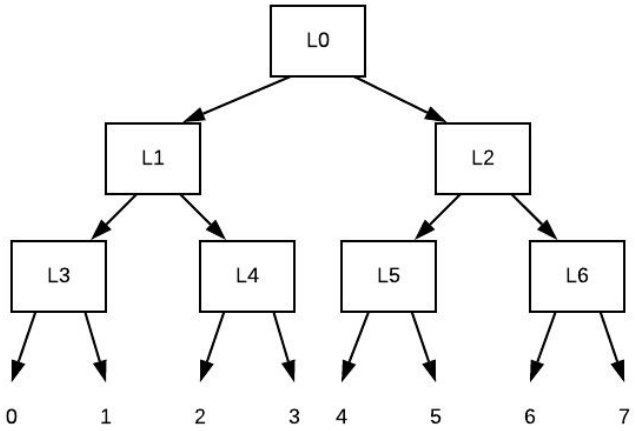
1. Whenever there is a hazard with both the memory stage and writeback stage needing to write to the same register in the execute stage, the instruction in the memory stage takes precedence.
2. The mem_rdata mentioned above is not the data directly from dcache, which would create a very long critical path if used. Instead, we load the data from dcache in a register first, and use the register output in forwarding after a 1-cycle stall.

# Advanced Design Options

## 8-way pLRU L2 cache:

Interface of pLRU_tree:

| **Signal Name** | **Direction** | **Notes** |
|---|---|---|
| clk | input | |
| rst | input | |
| load | input | Signals the module to update its internal states (L6:L0) |
| index | input | Index of the set to update |
| last_access | input | The most-recent accessed way |
| plru | output | The pseudo-LRU |




Our pLRU design would be a tree-like structure, where each bit of L[6:0] is kept in a tree node. A tree node would be similar to a register, but in addition to the standard I/O of a register array (clk, rst, load, index, data_in, data_out), it also has 2 1-bit output: load_left and load_right, which are the load signals that goes to its children.
For simplicity, we'll ignore the "index" input in the following discussion, as the pLRU-tree of each set functions identically and independently.

Update Rule:

(1) L0 (tree root) is always updated to last_access[2](MSB)
(2) If a node is updated (load = 1), one of its children also needs to be updated.
(3) If a node is not updated (load = 0), none of its children will be updated.

Thus, load_left and load_right is calculated as follows:

(1) load_left = load & (~data_in)  // Go left if data_in is 0

(2) load_right = load & data_in  // Go right if data_in is 1

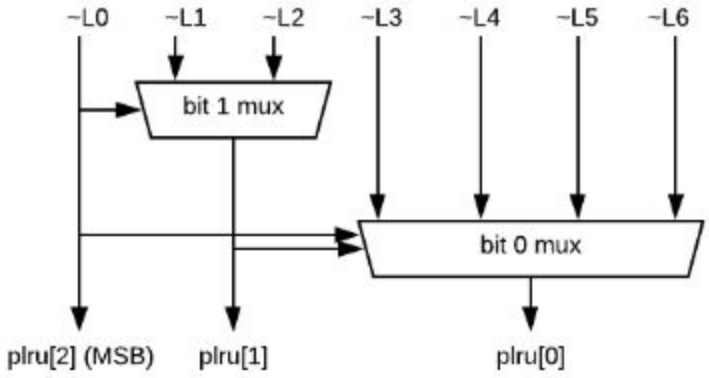
Data_in of each node is assigned as follows:

(1) last_access[2] (MSB) goes to L0 (root level)

(2) last_access[1] goes to L1 and L2

(3) last_access[0] (LSB) goes to L3, L4, L5, and L6 (leaf level)

Pseudo-LRU is computed by the following circuit:



## Eviction Write Buffer

Signals:

Ports to L2 cache: *read_i, write_i, addr_i, rdata_o, wdata_i, resp_o*
Ports to pmem (cacheline adapter): *read_o, write_o, addr_o, rdata_i, wdata_i, resp_i*

Internal data: *wdata, addr, valid*.

State Actions and Transitions:

**Idle**:

If (read_i == 1):

If requested data matches the cacheline stored in EWB, set *rdata_o* = *wdata*, and assert *resp_o*. Otherwise, directly connect L2 cache to pmem ports.

If (write_i == 1):

Load addr_i and wdata_i into internal registers (addr and wdata), then assert resp_o.
Go to **Wait** state.

**Wait**:

(Idea: because we are using write-back cache, and a memory write is performed just now, there's very likely a memory read immediately after. Write-back will start after the next read completes)
Directly connect L2 cache (read interface) to pmem ports.
Wait for one of the following:

- L2 finishes reading from memory
- Several cycles of inactivity
- L2 asserts *write_i*

Then, go to **Write** state.

**Write**:

Write buffered data back to memory.
Can serve EWB-hit reads at this state.
Ignores all other r/w requests from L2 cache.
Go to **Idle** state once write is completed.

# RISC-V M Extension

## Integration into Datapath

Integration into the datapath is trivial - divider and multiplier circuits will be added into the execution stage of the pipeline and used much like the ALU, and the control signals will be modified to accommodate the new hardware.

Due to the slow, complex nature of multiplication, and especially division, may force the datapath to be halted whenever waiting for a mathematical result from those units.

## Multiplier

The multiplier is a modified “Wallace Tree” multiplier, which is based off of lattice multiplication:

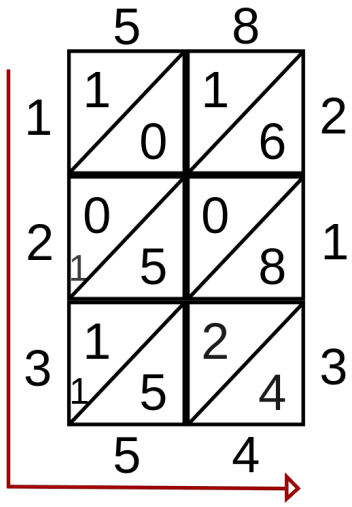



*Image Source:* Javier Rosa, *Step 3 of the lattice (shabakh) multiplication algorithm*, 7 March 2014, SVG Image, Wikipedia, accessed 20 April 2020, https://en.wikipedia.org/wiki/Lattice_multiplication#/media/File:Example_of_step_3_of_lattice_(shabakh)_multiplication_algorithm.svg

In this method of multiplication, the two large numbers are multiplied digit-wise in a table, the overflow and result are organized in the way shown above, and then summed along the diagonals. This summation is done in layers known as reduction layers, and for two numbers of length N, there are log(N) reduction layers, which is good for speeding up the computation.

Direct implementation of lattice multiplication / wallace tree algorithm in a bitwise fashion is highly straightforward:

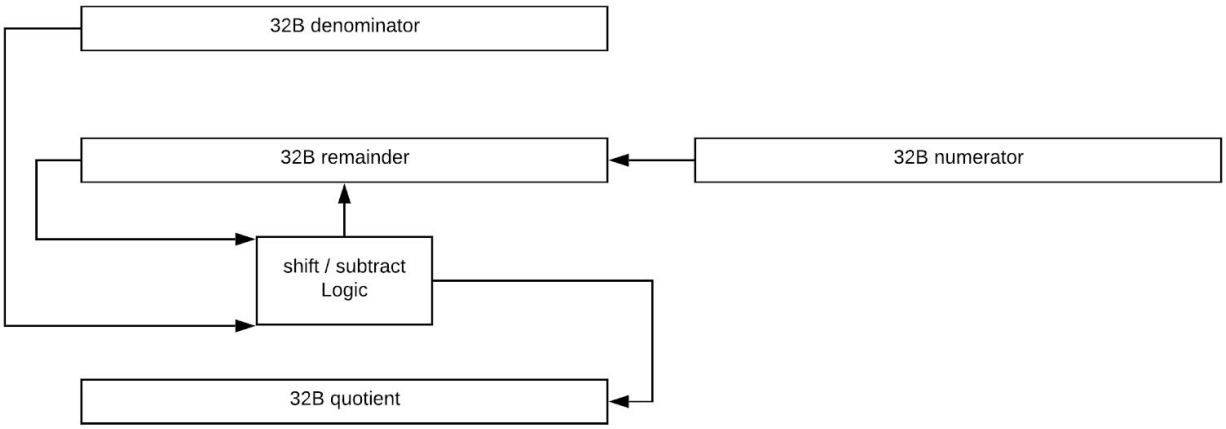
1. Multiply (AND operation) the bits and get a grid or N^2 results.
2. Sum these results diagonally with log(N) reduction layers
3. Sum the previous summation results together to get the final multiplication product

However, for 2 32-bit numbers the hardware can grow extremely complex and large, and this leads to the modification of the Wallace tree algorithm which uses BRAM to speed up the computations, using pre-computed results. If instead of performing the lattice operation bit-wise we can perform the operations on 4-bits at a time and use BRAM to store the pre-computed results of such 4-bit operations. This would change the original 5 reduction layers for 32B numbers to 3 reduction layers, leading to shorter critical paths.

The multiplication between each digit could be handled by a multiplication table (in addition to a multiplication “overflow” table) stored in memory of size 16 x 16 to quickly generate the lattice, and summing the results from these digit-wise multiplications over 3 layers should result in paths of lengths that should be able to be computed in a single 100MHz cycle. Assuming the critical paths match this prediction, multiplication should be able to be performed in a single clock cycle.

### Divider

The divider is analogous to a shift-add multiplier but instead of shift-add it is now shift-subtract:




It’s algorithm is analogous to the famous long division everyone is taught in grade school. The operation of the divider is best described in pseudo-code:

- If denominator > numerator:
    - Quotient = 0
    - Remainder = numerator
- If denominator == numerator:
    - Quotient = 1
    - Remainder = 0
- Otherwise, repeat 32 times (for 32B number):
    - Quotient[31:1] = quotient[30:0] (shift left)
    - If denominator >= remainder:
        - Remainder[31:1] = remainder - denominator

        - Quotient[0] = 1
    - Else:
        - Remainder[31:1] = remainder[30:0] (shift left)
        - Quotient[0] = 0
    - Remainder[0] = numerator[31]
- What remains in the quotient register is the quotient, and what remains in the remainder register is the remainder

The control for the divider is straightforward if none of the initial two “easy” conditions are met, then it’s a simple loop-shift-subtract 32 times. With this setup, the guaranteed worst case performance of the division algorithm is 32 cycles of the clock.