Arch Lab3

Cache Design

Chenlu Miao 11/2022

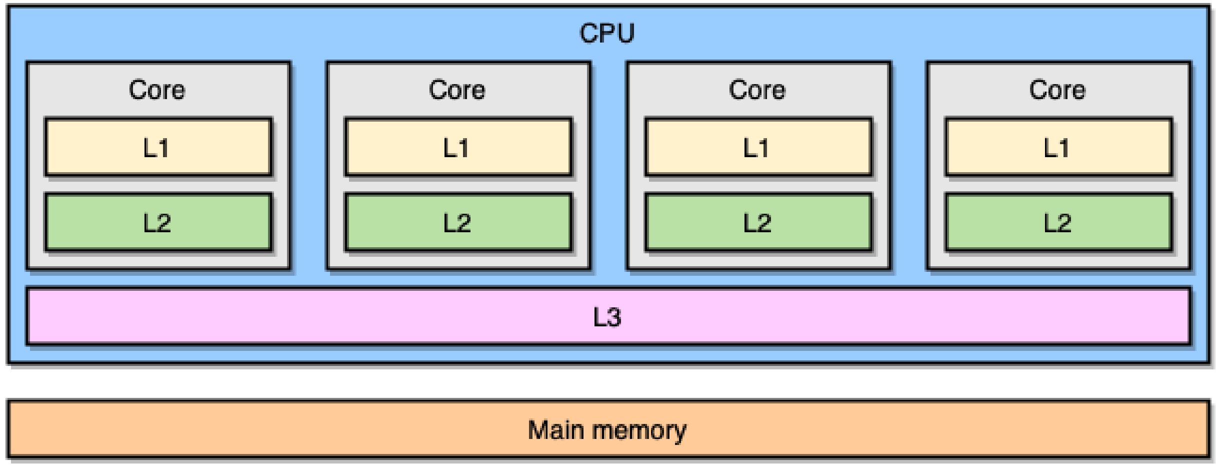
Tasks

- Design of Cache Line
- Verify the Cache Line
- Observe the Waveform of Simulation

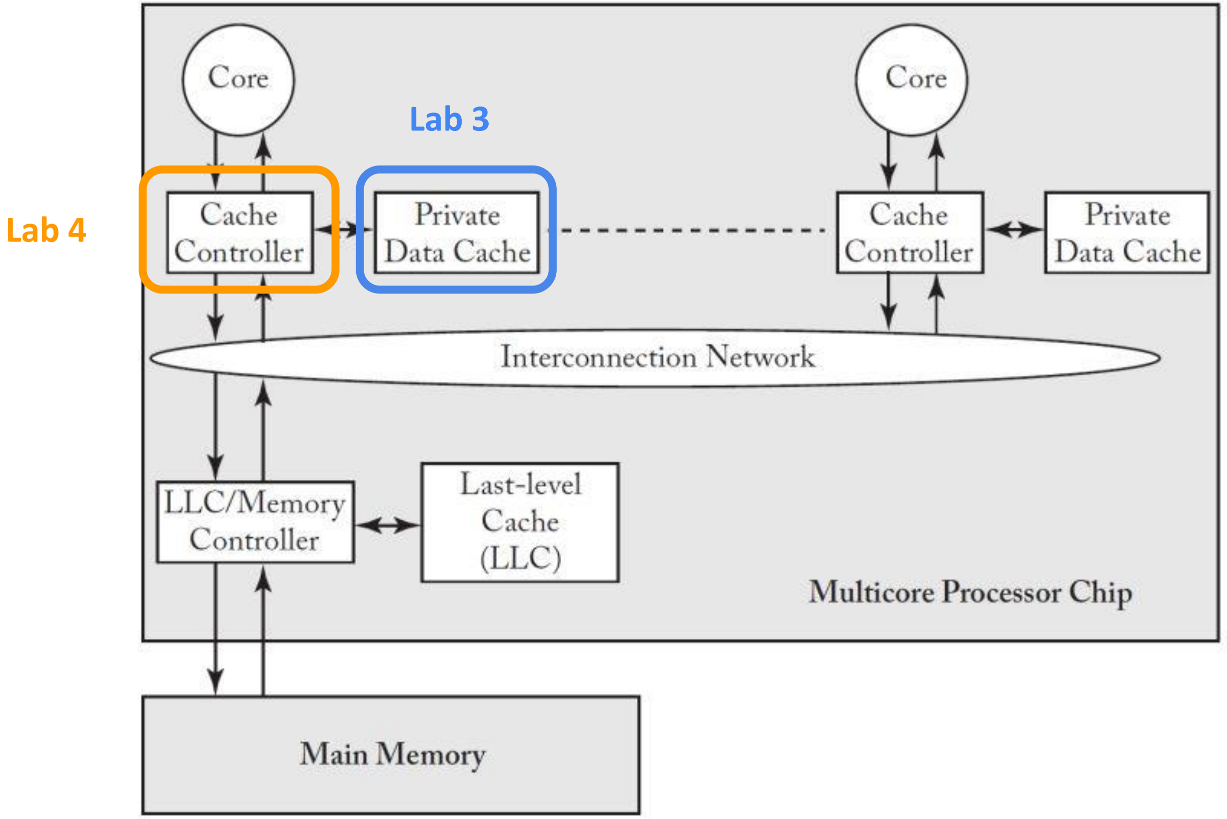
Overview

- Cache Overview
- Configuration
- Address Decoding
- Cache Storage
- Access Cache
- Simulation

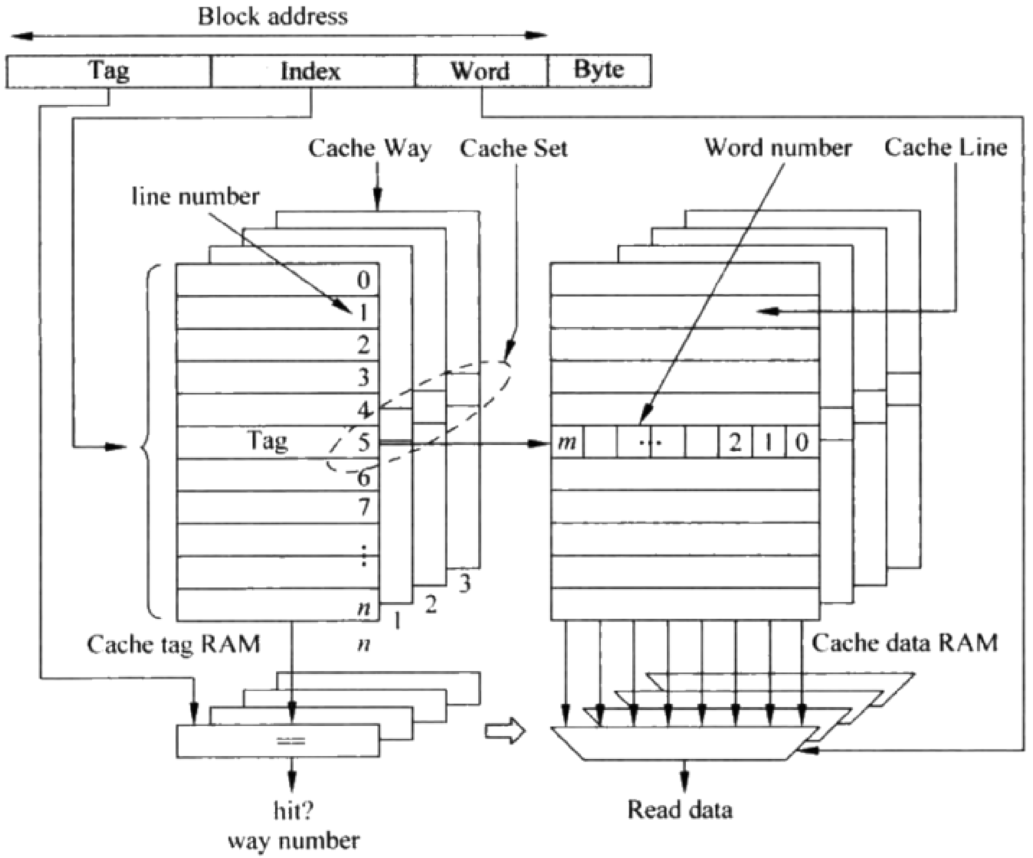
Cache Overview



Cache Overview



Cache Overview



Configuration

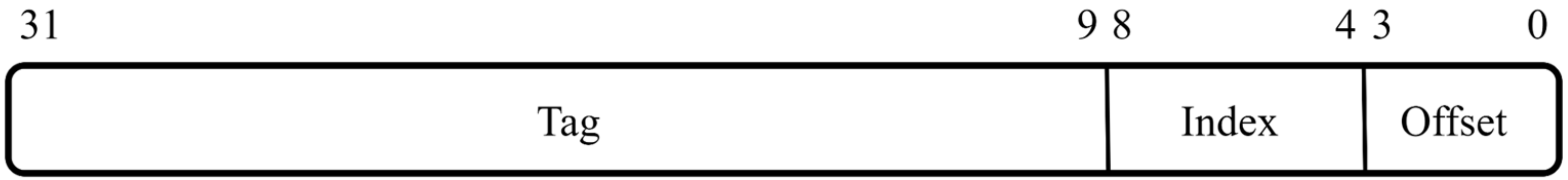
- Block Size
- Block Organization
  - Direct-mapped
  - Fully associative
  - **Set-associative**
- Block replacement policy
  - FIFO
  - **LRU**
  - Random
- Write policy
  - **Writeback**
  - Write-through

Configuration

- 64 cache lines
    - Cache Line Size = 16 Bytes = 4 words = 4 * 32 bits
- 2-way set associative
- Replacement policy: LRU
- Write policy
    - Write Back
    - Write Allocate

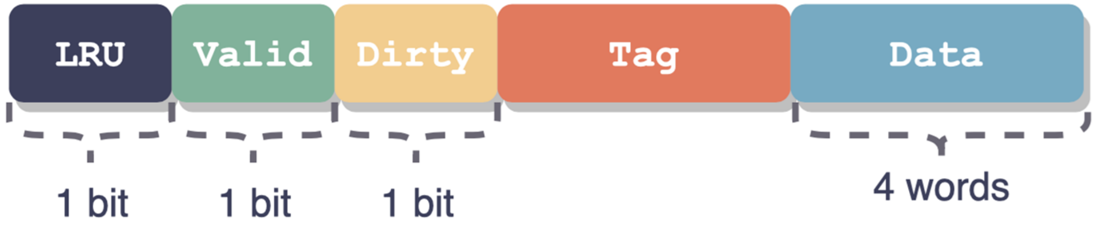
Address Decoding

- Offset
  - log2(cache line size) = 4 bits
- Index
  - log2(# cache line) - log2(cache assoc) = 5 bits
- Tag
  - 32 - Offset - Index = 23 bits



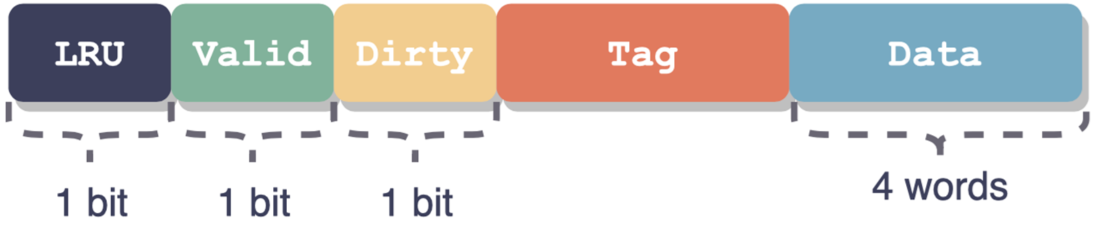
Cache Storage

- Tag
- Valid
- Dirty
- LRU
- Data



Cache Storage

- Tag
- Valid
- Dirty
- LRU
- Data



```
reg [ELEMENT_NUM-1:0] inner_recent = 0;
reg [ELEMENT_NUM-1:0] inner_valid = 0;
reg [ELEMENT_NUM-1:0] inner_dirty = 0;
reg [TAG_BITS-1:0]    inner_tag [0:ELEMENT_NUM-1];
reg [31:0]            inner_data [0:ELEMENT_NUM*ELEMENT_WORDS-1];
```

32 sets; 64 cache lines

set i way 0: $i\times 2+0 \rightarrow$ inner_recent[{i, 1'b0}]

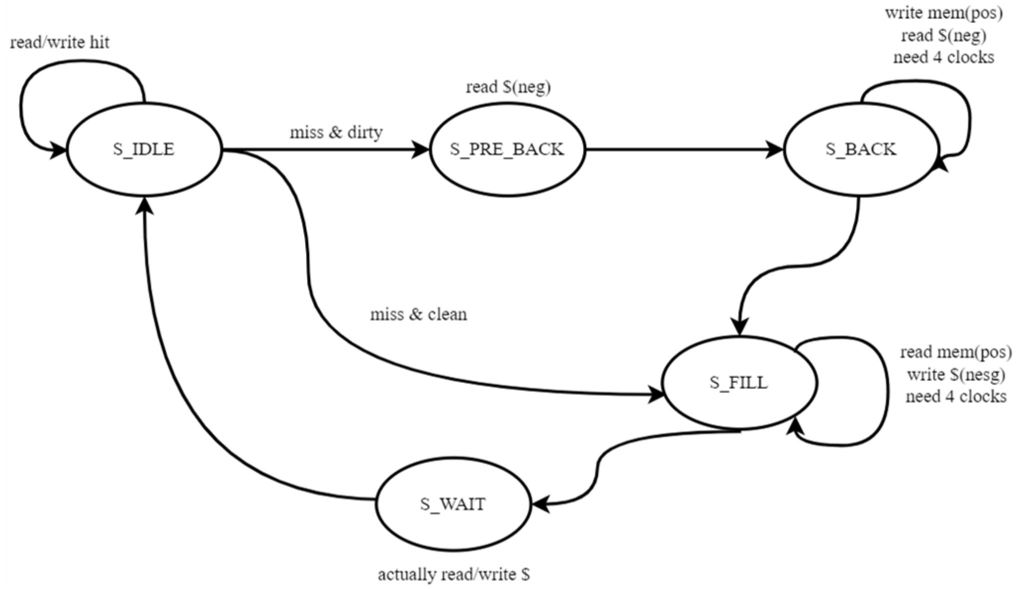
set i way 1: $i\times 2+1 \rightarrow$ inner_recent[{i, 1'b1}]

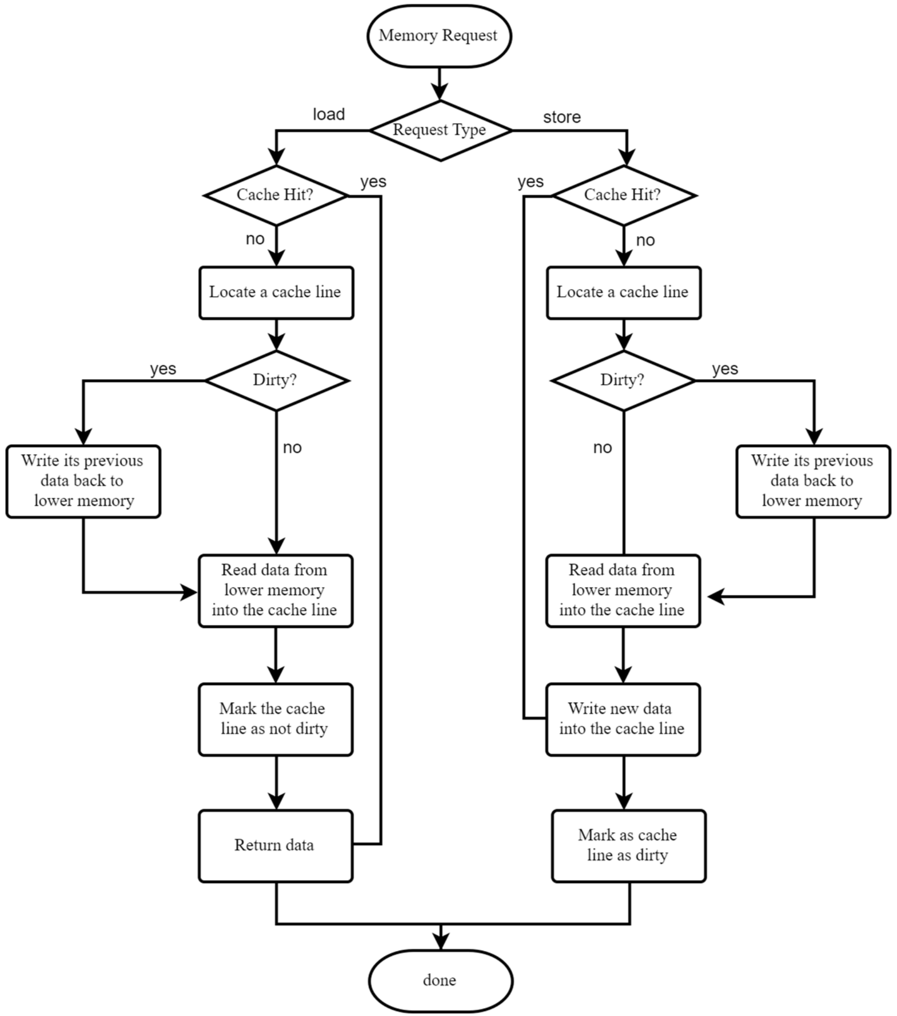
set i way j word k: $i\times 8+j\times 4+k \rightarrow$ inner_data[{i, j, k}]

Access Cache

```verilog
module cache (
    input wire clk,  // clock
    input wire rst,  // reset
    input wire [ADDR_BITS-1:0] addr,  // address
    input wire load,    //  read refreshes recent bit
    input wire store,  // set valid to 1 and reset dirty to 0
    input wire edit,  // set dirty to 1
    input wire invalid,  // reset valid to 0
    input wire [2:0] u_b_h_w, // select signed or not & data width  LB, LH, LW, LBU, LHU
    input wire [31:0] din,  // data write in
    output reg hit = 0,  // hit or not
    output reg [31:0] dout = 0,  // data read out
    output reg valid = 0,  // valid bit
    output reg dirty = 0,  // dirty bit
    output reg [TAG_BITS-1:0] tag = 0  // tag bits
    );
```

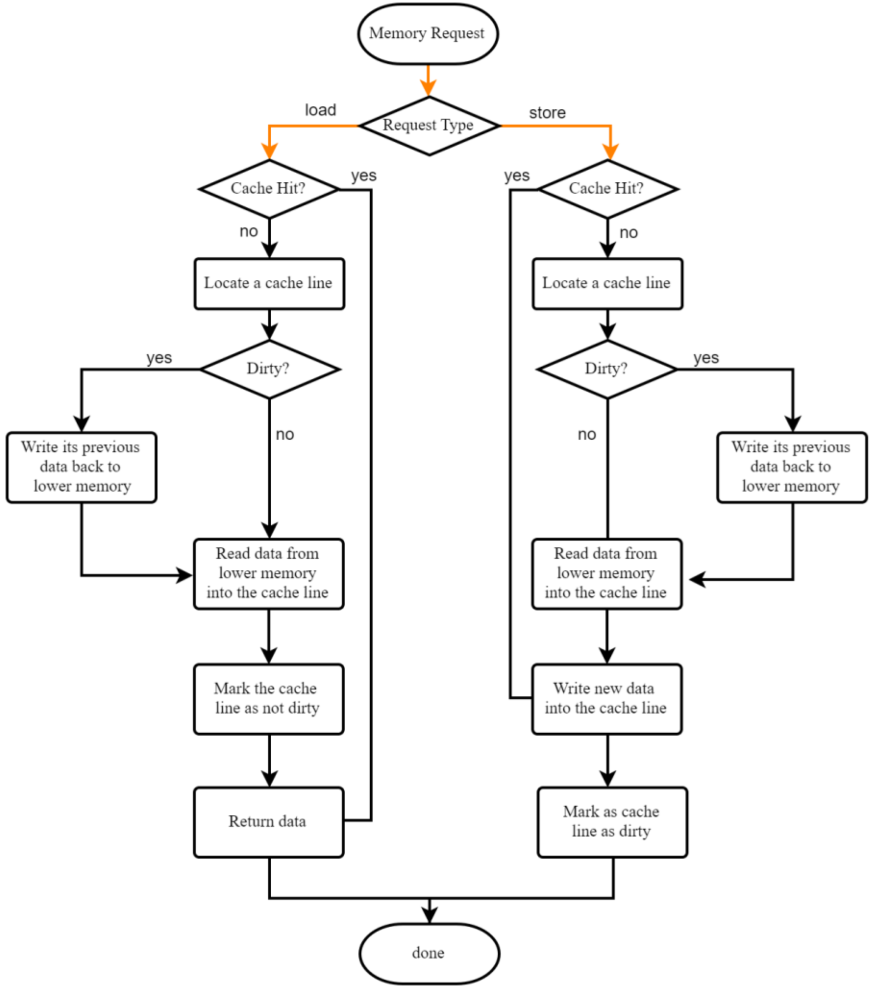
Access Cache




Memory Request
Request Type
load
store
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Mark the cache line as not dirty
Return data
Cache Hit?
yes
no
Locate a cache line
Dirty?
no
yes
Write its previous data back to lower memory
Read data from lower memory into the cache line
Write new data into the cache line
Mark as cache line as dirty
done

Access Caches

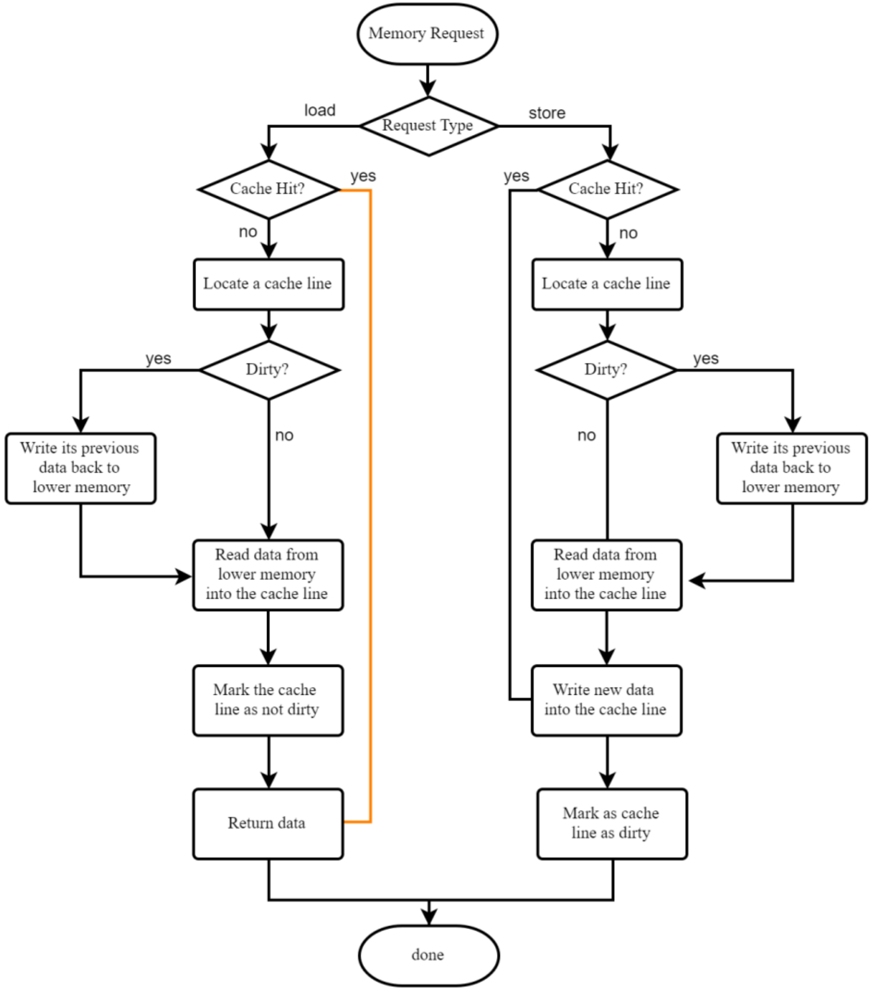
```
assign addr_tag = ...
assign addr_index = ...
...
assign valid1 = inner_valid[addr_element1];
assign valid2 = ...

assign tag1 = inner_tag[addr_element1];
assign tag2 = ...

assign hit1 = valid1 & (tag1 == addr_tag);
assign hit2 = …

always @ (posedge clk) begin
        ...
        hit <= ...
end
```



Memory Request
Request Type
load
store
Cache Hit?
yes
no
Cache Hit?
yes
no
Locate a cache line
Locate a cache line
Dirty?
yes
no
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Write its previous data back to lower memory
Read data from lower memory into the cache line
Mark the cache line as not dirty
Write new data into the cache line
Return data
Mark as cache line as dirty
done

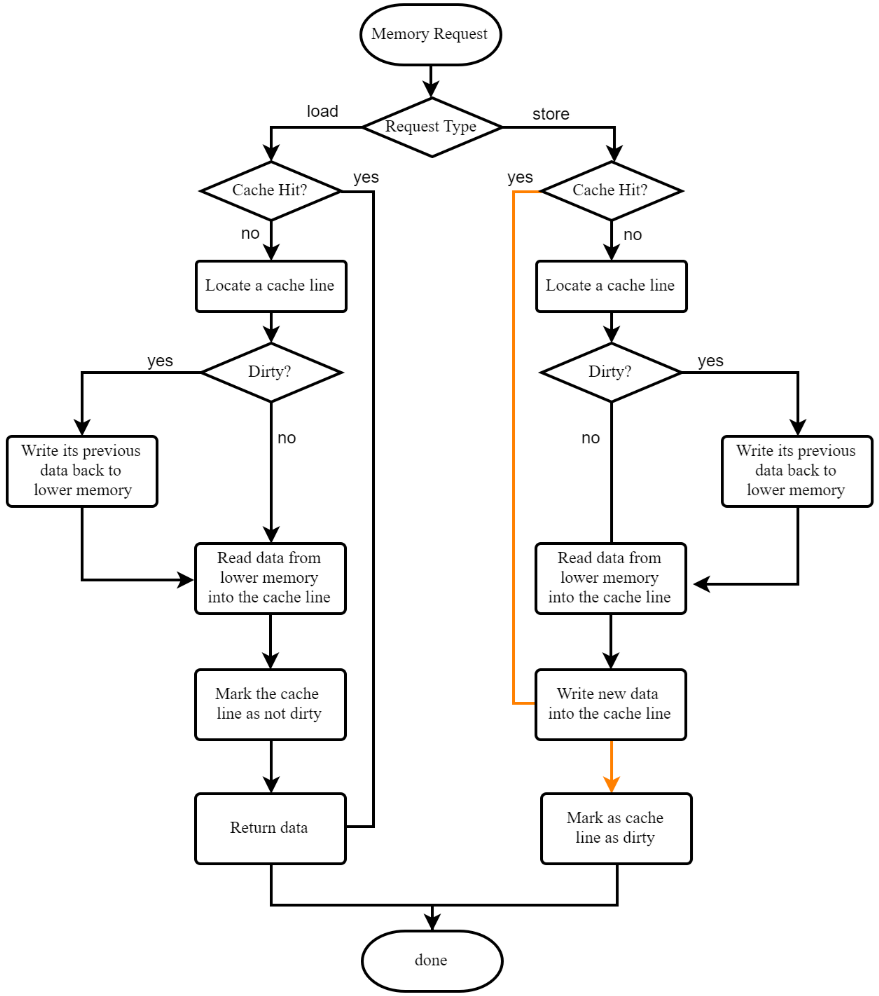
Access Caches

```
always @ (posedge clk) begin
    ...
    if (load) begin
        if (hit1) begin
            dout <=...

            inner_recent[addr_element1] <= 1'b1;
            inner_recent[addr_element2] <= 1'b0;
        end
          else if (hit2) ...
        end
        else ...
    ...
end
```


Memory Request
Request Type
load
store
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Mark the cache line as not dirty
Return data
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Write new data into the cache line
Mark as cache line as dirty
done

Access Caches

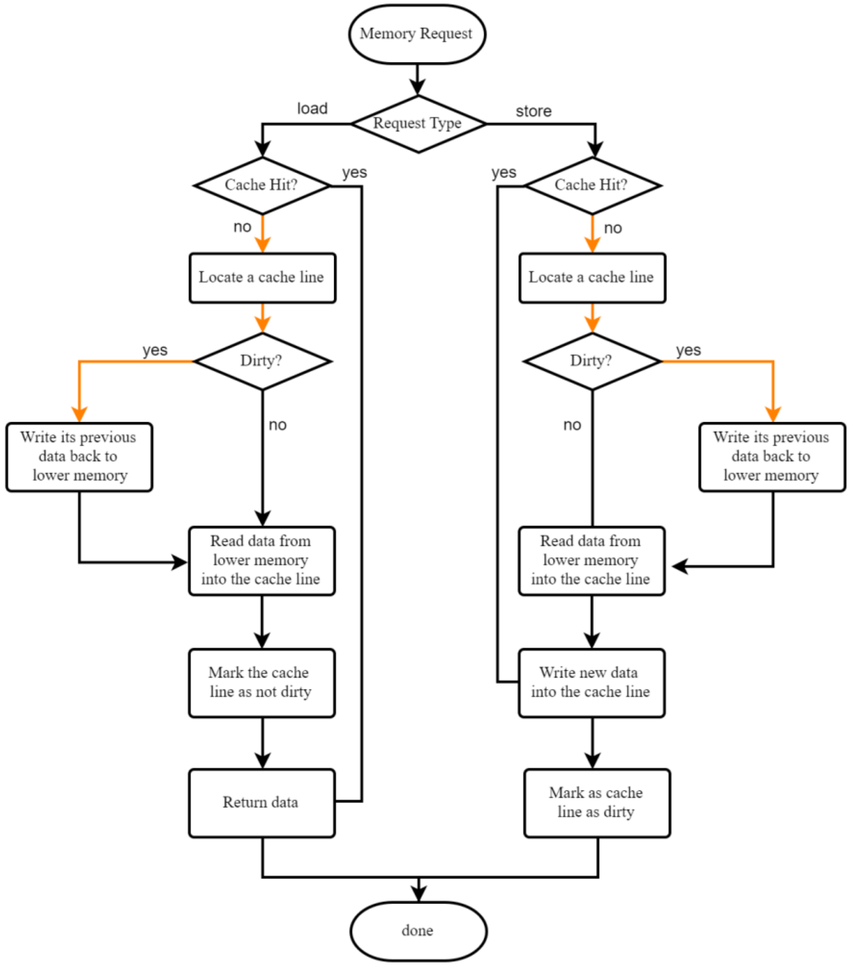
```
always @ (posedge clk) begin
    ...
    if (edit) begin
        if (hit1) begin
            inner_data[addr_word1] <= ...
            inner_dirty[addr_element1] <= 1'b1;
            inner_recent[addr_element1] <= 1'b1;
            inner_recent[addr_element2] <= 1'b0;
        end
        else if (hit2) ...
    end
    ...
end
```


Memory Request
Request Type
load
store
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Mark the cache line as not dirty
Return data
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Write new data into the cache line
Mark as cache line as dirty
done

Access Caches

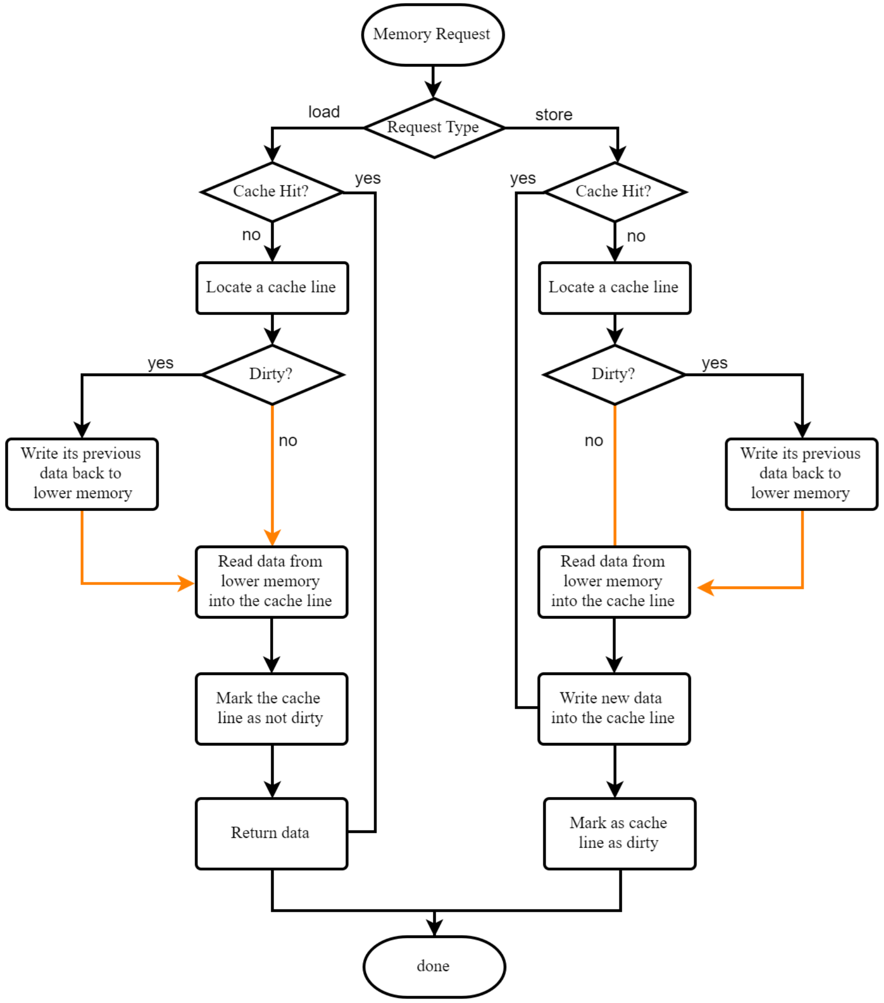
```
assign recent1 = inner_recent[addr_element1];
assign recent2 = ...
assign dirty1 = inner_dirty[addr_element1];
assign dirty2 = ...
always @ (posedge clk) begin
     valid <= recent1 ? valid2 : valid1;
     dirty <= ...
     tag <= ...
     hit <= ...
     if (load) begin
          ...
     end
     else dout <= inner_data[ recent1 ? addr_word2
     addr_word1 ];
end
```

for writeback


Memory Request
Request Type
load
store
Cache Hit?
yes
no
Cache Hit?
yes
no
Locate a cache line
Locate a cache line
Dirty?
yes
no
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Read data from lower memory into the cache line
Write its previous data back to lower memory
Mark the cache line as not dirty
Write new data into the cache line
Return data
Mark as cache line as dirty
done

Access Caches

```
always @ (posedge clk) begin
    ...
    if (store) begin
        if (recent1) begin  // replace 2
            inner_data[addr_word2] <= din;
            inner_valid[addr_element2] <= 1'b1;
            inner_dirty[addr_element2] <= 1'b0;
            inner_tag[addr_element2] <= addr_tag;
        end else begin
            ...
        end
    end
    ...
end
```


Memory Request
Request Type
load
store
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Mark the cache line as not dirty
Return data
Cache Hit?
yes
no
Locate a cache line
Dirty?
yes
no
Write its previous data back to lower memory
Read data from lower memory into the cache line
Write new data into the cache line
Mark as cache line as dirty
done

Simulation

```
// init
32'd10: begin
    load <= 0;
    store <= 1;
    edit <= 0;

    din <= 32'h11111111;
    addr <= 32'h00000004;
end

32'd11: begin
    addr <= 32'h0000000C;
end

32'd12: begin
    addr <= 32'h00000010;
end

32'd13: begin
    addr <= 32'h00000014;
end
```

```
// read miss
32'd14: begin
    load <= 1;
    store <= 0;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 0;
    addr <= 32'h00000020;
end

// read hit
32'd15: begin
    u_b_h_w <= 3'b010;
    addr <= 32'h00000010;
end
```

Simulation

```
// write miss
32'd16: begin
    load <= 0;
    store <= 0;
    edit <= 1;

    u_b_h_w <= 3'b010;
    din <= 32'h22222222;
    addr <= 32'h00000024;
end

// write hit
32'd17: begin
    u_b_h_w <= 3'b010;
    addr <= 32'h00000014;
end
```

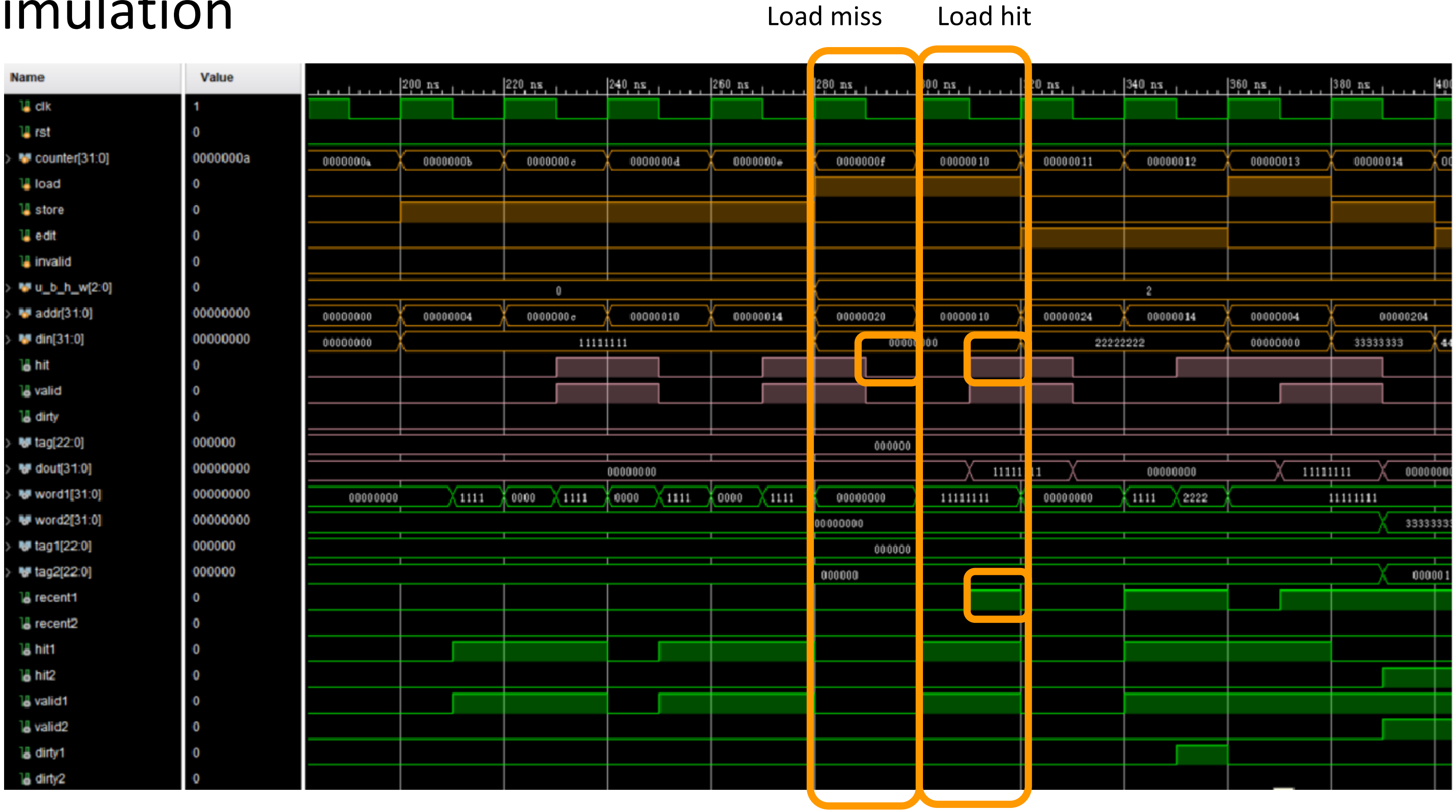
```
// read line 0 of set 0, set
recent bit
32'd18: begin
    load <= 1;
    store <= 0;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 0;
    addr <= 32'h00000004;
end

// store to line 1 of set 0
due to line 0 recent
32'd19: begin
    load <= 0;
    store <= 1;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 32'h33333333;
    addr <= 32'h00000204;
end
```

Simulation

```
// edit line 1 of set 0, set
dirty & recent
32'd20: begin
    load <= 0;
    store <= 0;
    edit <= 1;

    u_b_h_w <= 3'b010;
    din <= 32'h44444444;
    addr <= 32'h00000204;
end

// read line 0 of set 0, set
recent bit
32'd21: begin
    load <= 1;
    store <= 0;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 0;
    addr <= 32'h00000004;
end
```

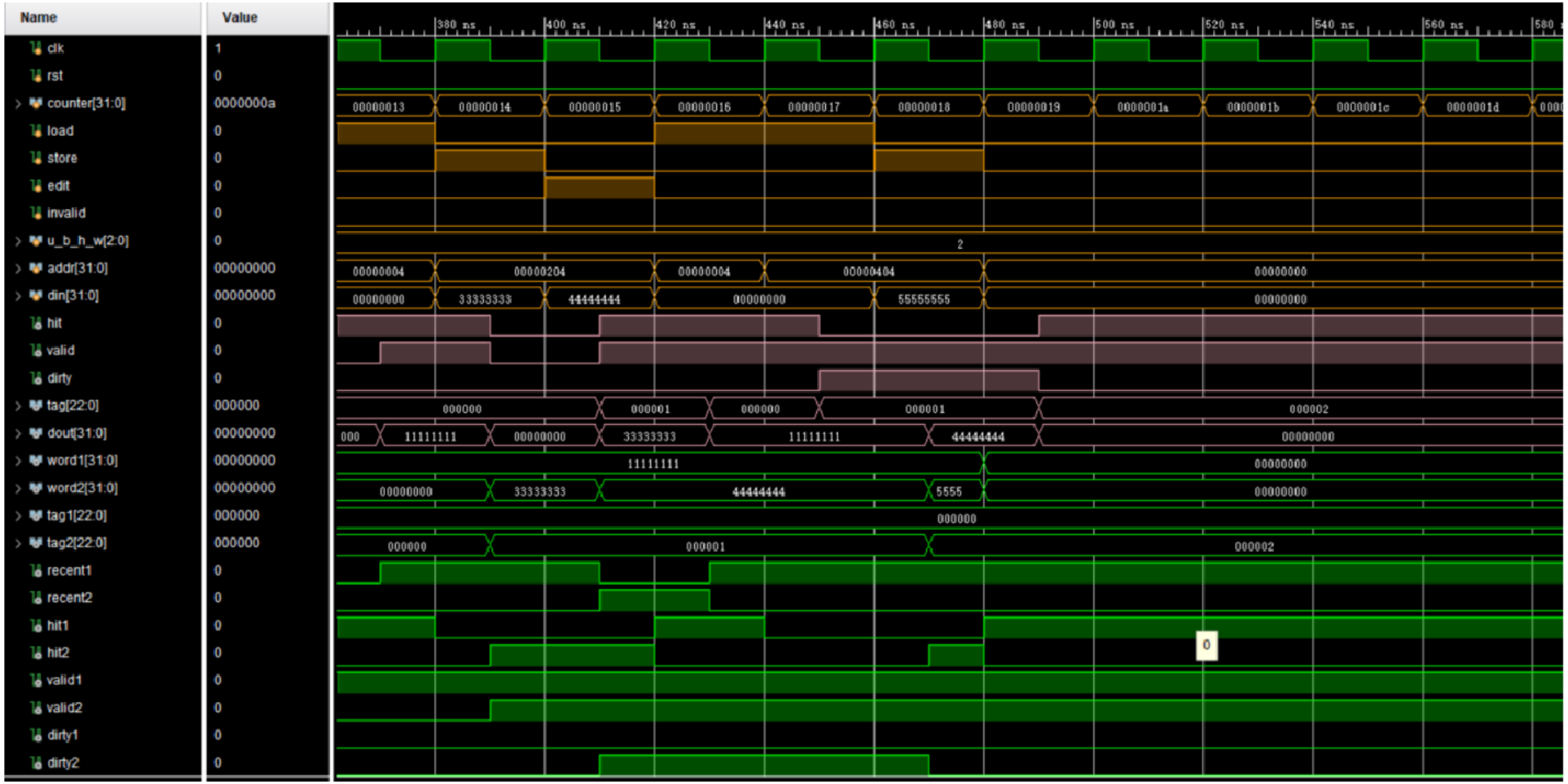
```
// read miss, tag mismatch.
output tag (of line 1), valid
and dirty == 1
32'd22: begin
    load <= 1;
    store <= 0;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 32'h0;
    addr <= 32'h00000404;
end

// auto replace line 1 of set 0
32'd23: begin
    load <= 0;
    store <= 1;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 32'h55555555;
    addr <= 32'h00000404;
end
```

Simulation



Name	Value
clk	1
rst	0
counter[31:0]	0000000a
load	0
store	0
edit	0
invalid	0
u_b_h_w[2:0]	0
addr[31:0]	00000000
din[31:0]	00000000
hit	0
valid	0
dirty	0
tag[22:0]	000000
dout[31:0]	00000000
word1[31:0]	00000000
word2[31:0]	00000000
tag1[22:0]	000000
tag2[22:0]	000000
recent1	0
recent2	0
hit1	0
hit2	0
valid1	0
valid2	0
dirty1	0
dirty2	0

Simulation


Name
Value
clk 1
rst 0
counter[31:0] 0000000a
load 0
store 0
edit 0
invalid 0
u_b_h_w[2:0] 0
addr[31:0] 00000000
din[31:0] 00000000
hit 0
valid 0
dirty 0
tag[22:0] 000000
dout[31:0] 00000000
word1[31:0] 00000000
word2[31:0] 00000000
tag1[22:0] 000000
tag2[22:0] 000000
recent1 0
recent2 0
hit1 0
hit2 0
valid1 0
valid2 0
dirty1 0
dirty2 0
380 ns
400 ns
420 ns
440 ns
460 ns
480 ns
500 ns
520 ns
540 ns
560 ns
580
00000013
00000014
00000015
00000016
00000017
00000018
00000019
0000001a
0000001b
0000001c
0000001d
2
00000004
00000204
00000004
00000404
00000000
33333333
44444444
55555555
000000
000001
000002
000
11111111
5555
0

References

- 超标量处理器设计
- Modern Processor Design