N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

[Material partly adapted from lecture notes of Prof. KJ Lee]

Fall 2010

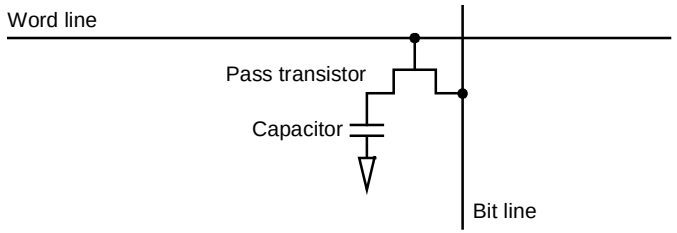
Design of Cache

Outline

- Memory Hierarchy
- Timing
- Building blocks
- Cache controller
- Test result
- A Complete Verilog Code

Memories: Review

- SRAM:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space than DRAM (4 to 6 transistors)
- DRAM:
  - value is stored as a charge on capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)

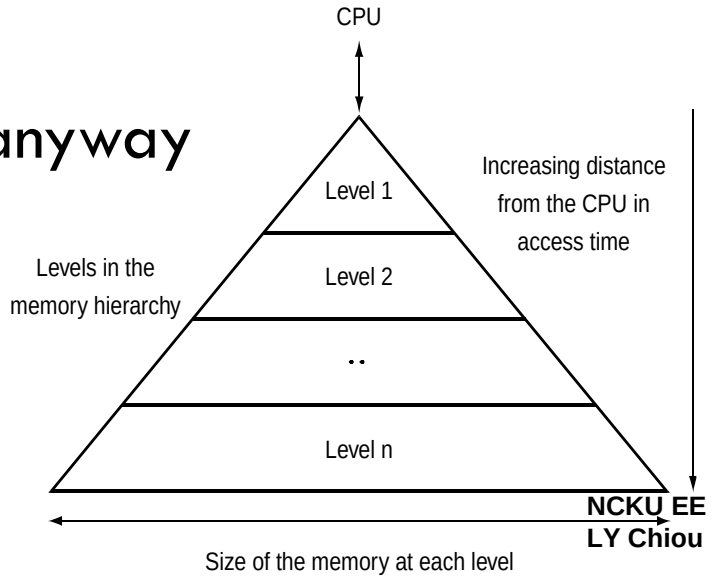


Exploiting Memory Hierarchy

- Users want large and fast memories!
- Cost of memory varying!!

	SRAM	DRAM	DISK
Access Time (ns)	0.5 ~ 5	50 ~ 70	(5 ~ 20) x 10^6
Cost (per GB)	$10,000	$100 to $200	$0.5 to $2

- Try and give it to them anyway
  - build a memory hierarchy



Locality

- A principle that makes a memory hierarchy a good idea
- **If an item is referenced,**

  temporal locality: it will tend to be referenced again soon

  spatial locality: nearby items will tend to be referenced soon.

- Definition
  - block: minimum unit of data
  - hit: data requested is in the upper level
  - miss: data requested is not in the upper level

Cache

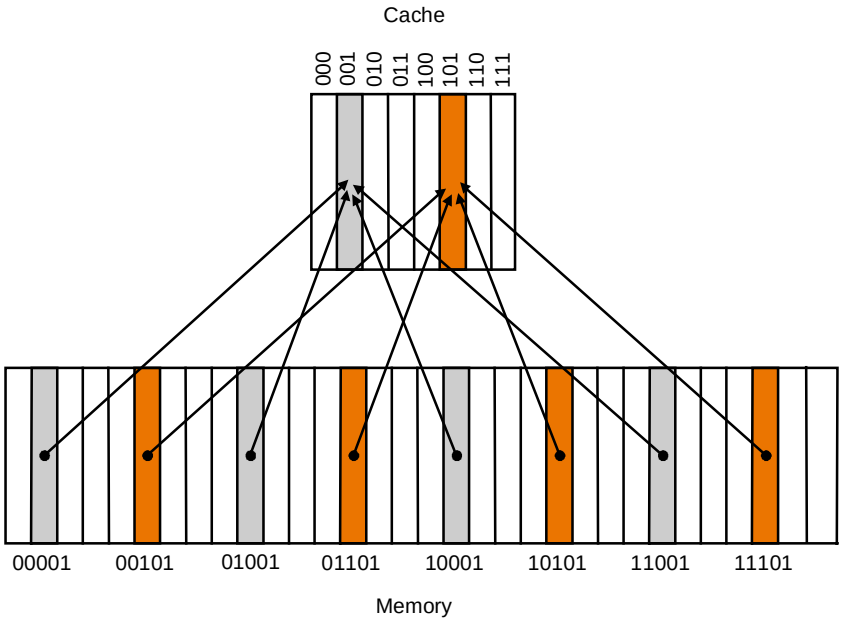
- Two issues:
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?
- Our first example:
  - block size is one word of data
  - "direct mapped"

For each item of data at the lower level,
there is exactly one location in the cache where it might be.

e.g., lots of items at the lower level share locations in the upper level

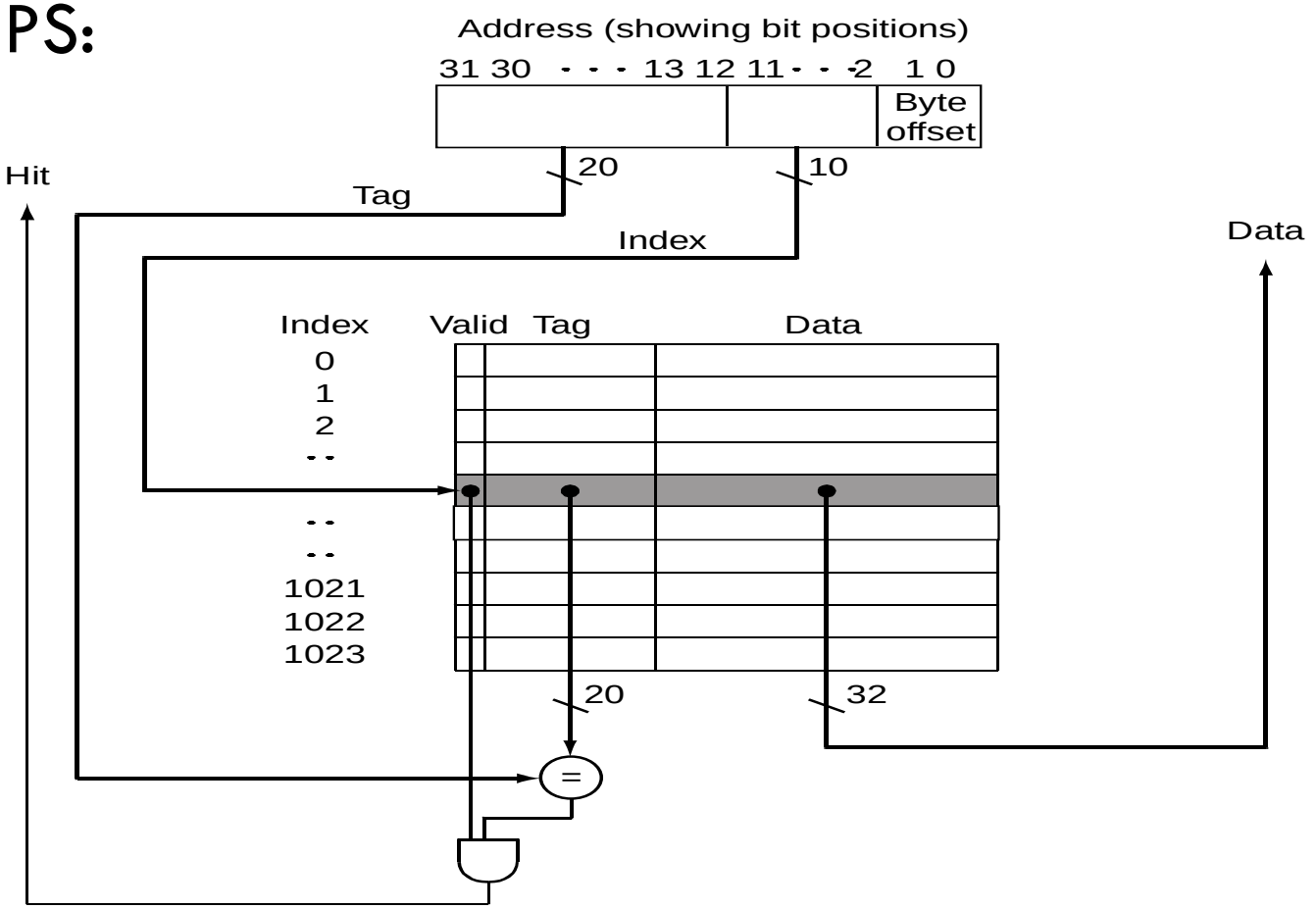
Direct Mapped Cache

- Mapping: address is modulo the number of blocks in the cache



Direct Mapped Cache

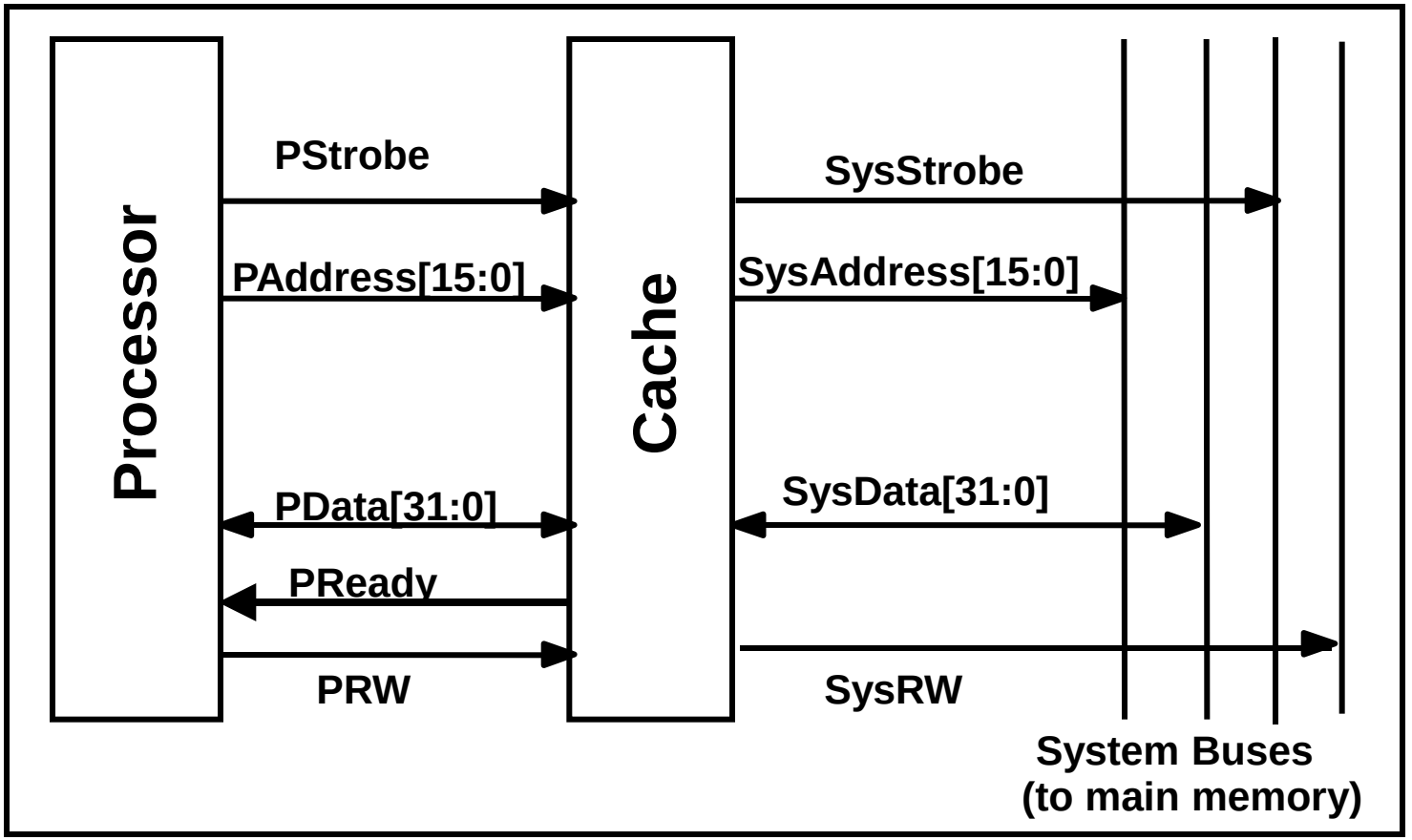
- For MIPS:



Cache Memory

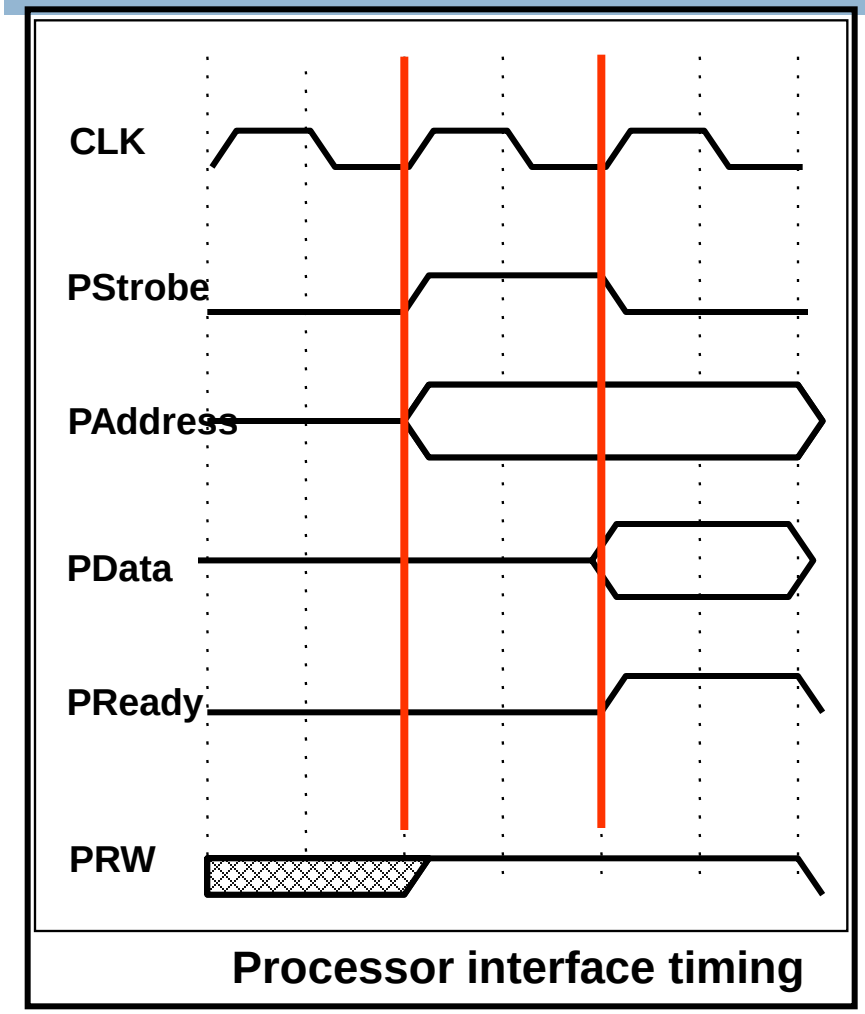
- **Small, fast, local, expensive**
- **Usually static RAM.**
- **Most advanced CPUs have on_chip cache.**

Block Diagram of Cache



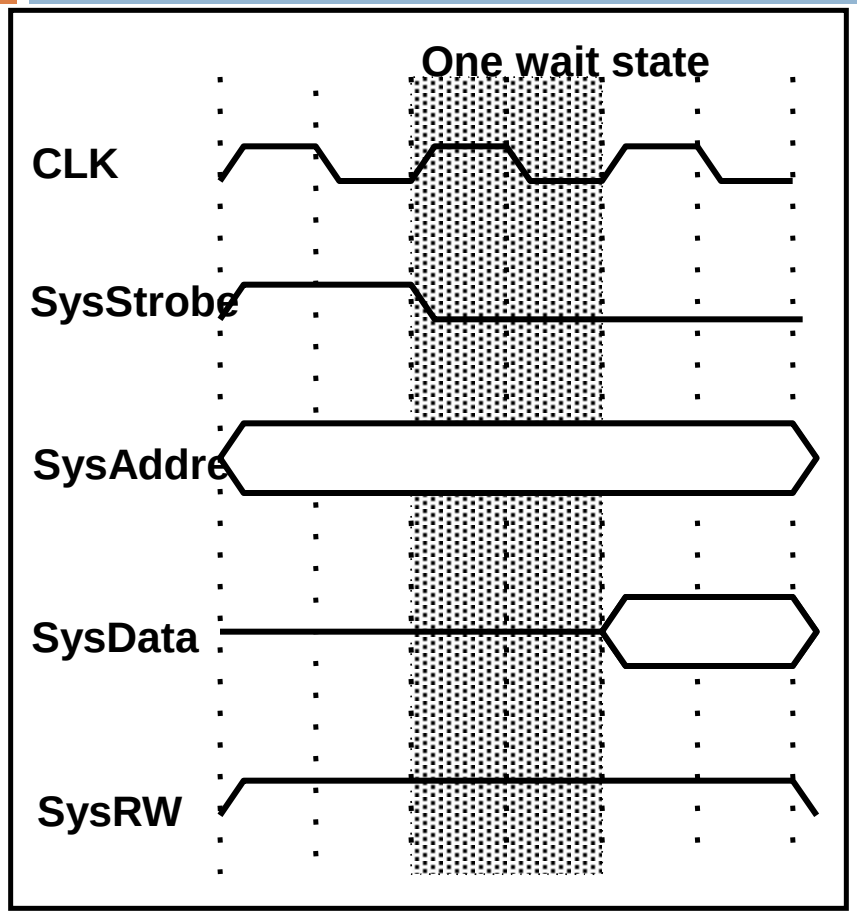
PRW=1 : read
PRW=0 : write

Timing of read operation from cache



1. PStrobe is asserted by the processor when PAddress is ready and PRW is determined. (starting a bus transaction).
2. PReady is signaled to the processor when the DATA is available. (a bus transaction is completed)

Reading from main memory



Timing for a read cycle with one wait state

Usually a fixed number of wait states is used. ---- no ready signal.

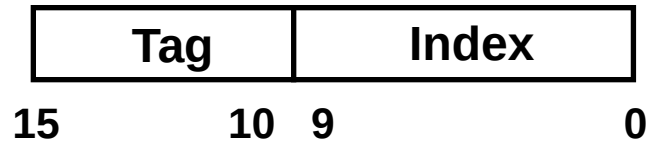
A write operation is similar, but the data is driven onto the PData bus immediately by the Cache and then is held a number of wait states before another write operation is issued.

SysRW = 1 : read
SysRW = 0 : write

A Cache Example

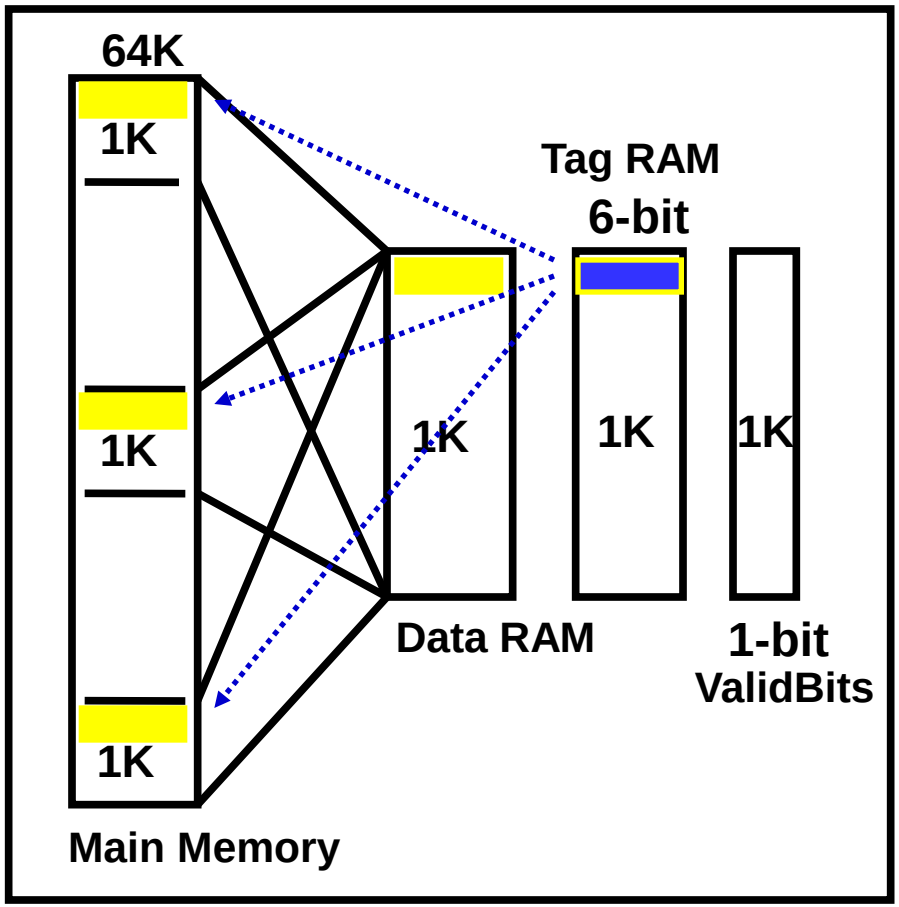
- 64K main memory, 1K cache
- Direct mapping
- Write-through policy

Address field:



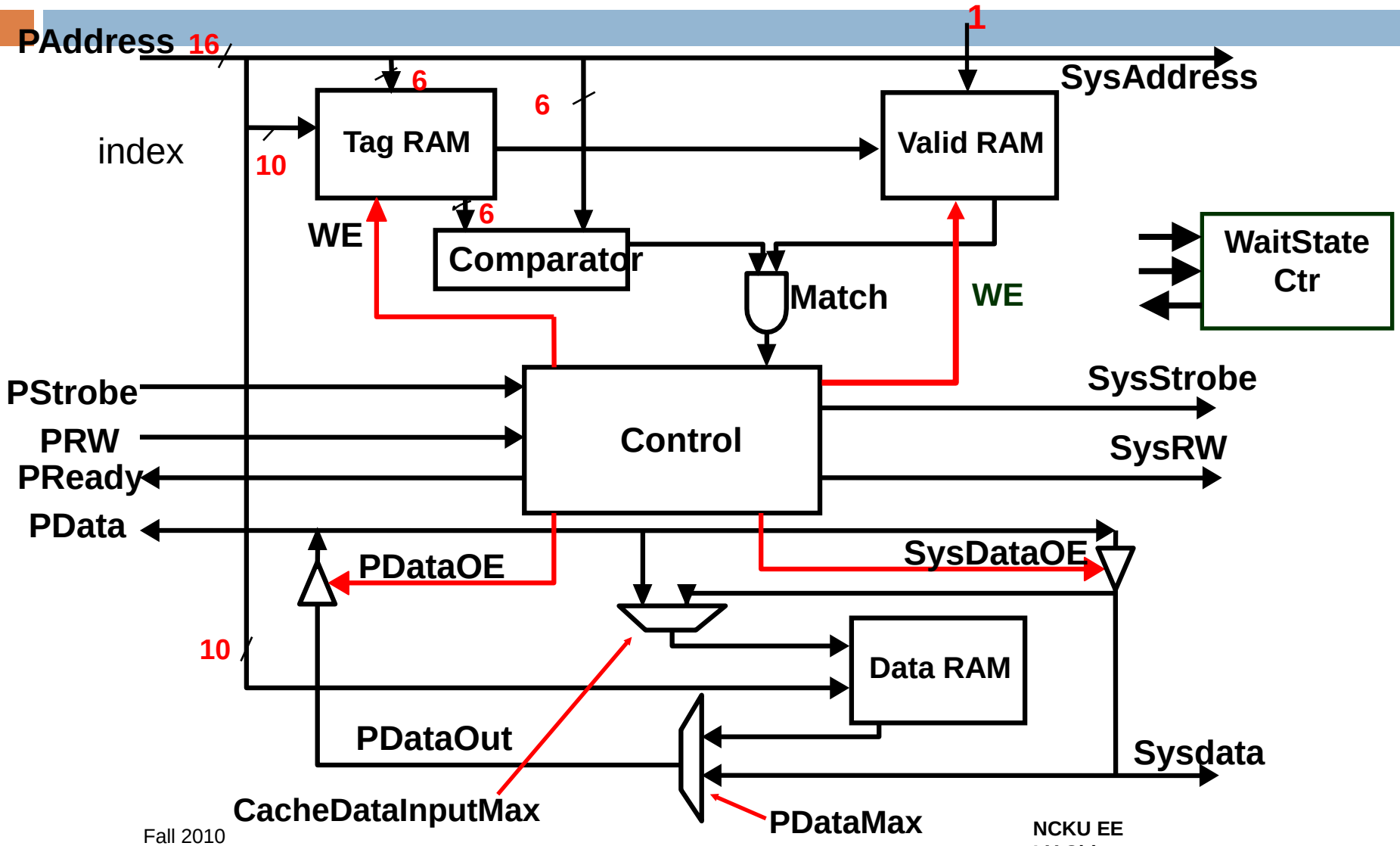
Only one word will be updated at a time in this design

- Each location in the cache is mapped to 64 different locations in the memory.

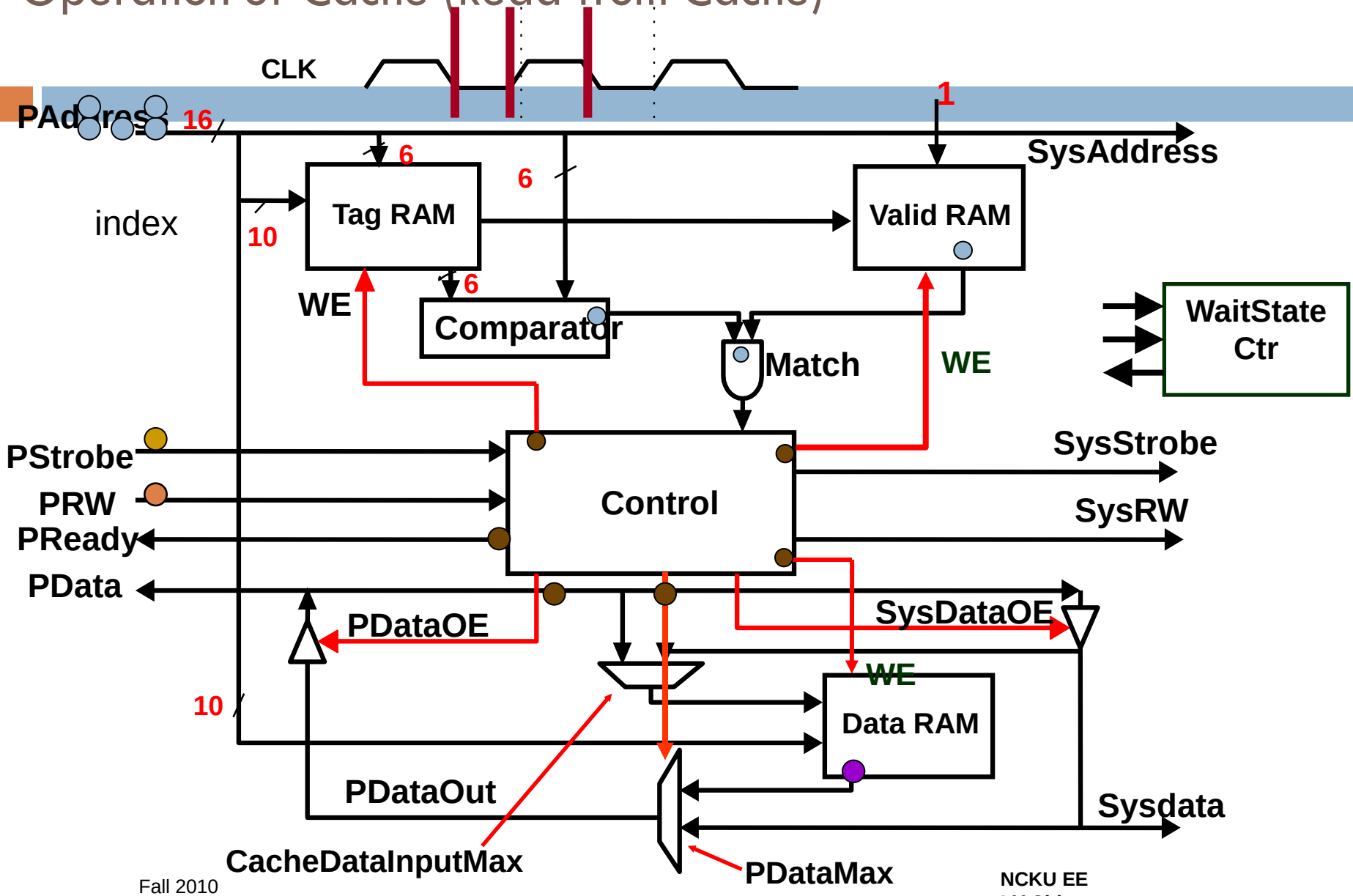


Mapping between main memory and cache

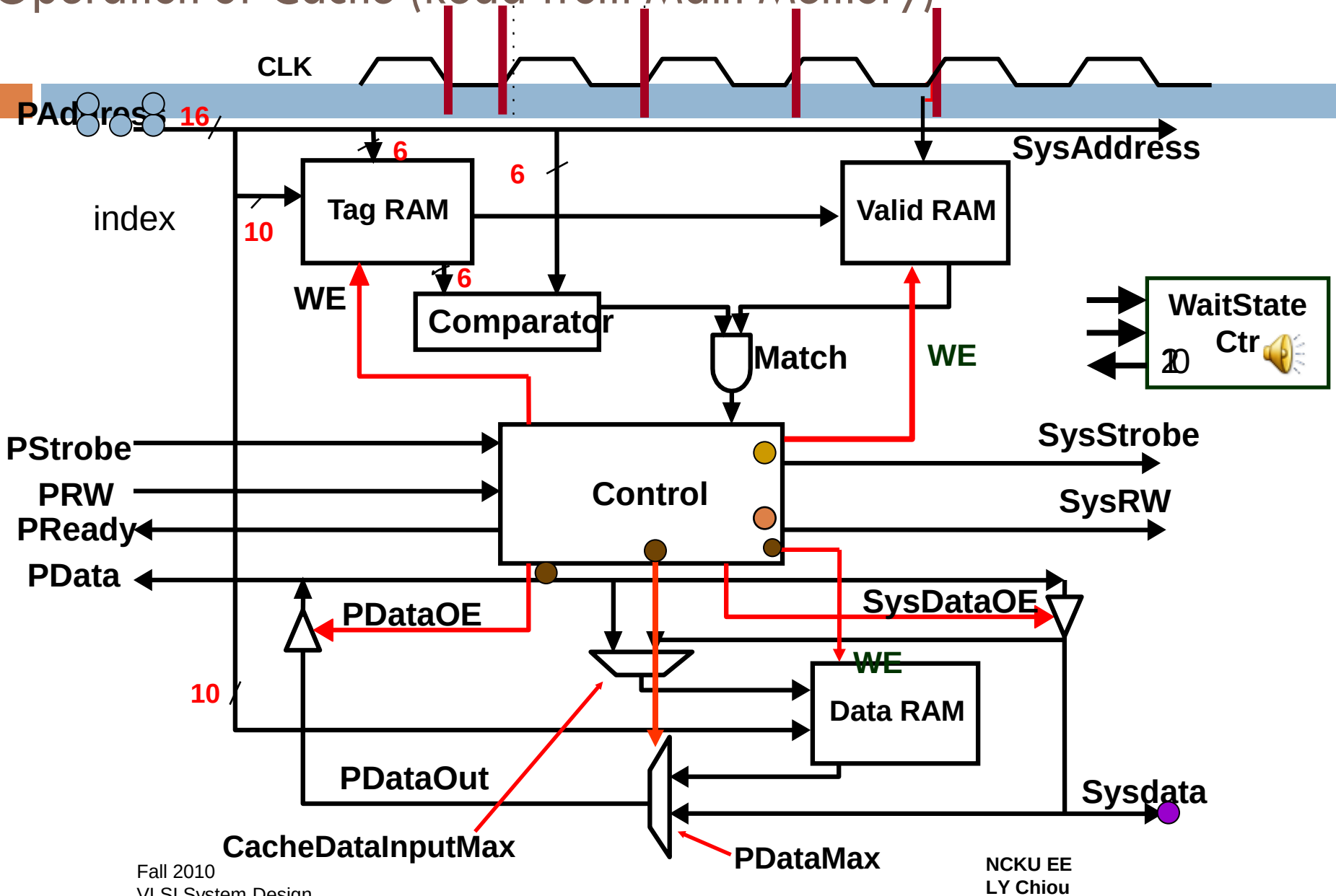
Structure of a Typical Cache



Operation of Cache (Read from Cache)



Operation of Cache (Read from Main Memory)



Main Cache Module (1/3)

```
`define READ 1
`define WRITE 0
`define CACHESIZE 1024
`define WAITSTATES 2
//number of wait states
//  required for system accesses
`define ADDR 15:0
`define INDEX 9:0
`define TAG 15:10
`define DATAWIDTH 32
`define DATA `DATAWIDTH-1:0
`define PRESENT 1
`define ABSENT !`PRESENT
module cache(
        PStrobe, PAddress,
        PData, PRW,
        PReady,
        SysStrobe, SysAddress,
        SysData, SysRW,
        Reset,
        Clk         );
```

```
Input  PStrobe;
input  [`ADDR]  PAddress;
inout  [`DATA]  PData;
input        PRW;
output    PReady;

output   SysStrobe;
output   [`ADDR] SysAddress;
input     [`DATA]  SysData;
output    SysRW;
input        Reset;
input        Clk;

//Bidirectional buses
wire  PDataOE;
wire  SysDataOE;
wire [`DATA]  PDataOut;
wire [`DATA]  PData=PDataOE ? PDataOut : 'bz;
wire [`DATA]  SysData=SysDataOE? PData : 'bz;
wire [`ADDR]  SysAddress = PAddress;
wire [`TAG]    TagRamTag;
```

Main Cache Module (2/3)

```
TagRam TagRam (
        .Address    (PAddress[`INDEX]),
        .TagIn      (PAddress[`TAG]),
        .TagOut     (TagRAmTag[`TAG]),
        .Write      (Write),
        .Clk        (Clk)
);
ValidRam ValidRam(
        .Address    (PAddress[`INDEX]),
        .ValidIn    (1`b1),
        .ValidOut   (Valid),
        .Write      (Write),
        .Reset      (Reset),
        .Clk        (Clk)
);
wire   [`DATA] DataRamDataOut;
wire   [`DATA] DataRamDataIn;
```

```
DataMux CacheDataInputMux(
     .S(CacheDataSelect),
     .A(SysData),
     .B(Pdata),
   .Z(DataRamDataIn)
);

DataMux PDataMax(
     .S(PDataSelect),
     .A(SysData),
     .B(DataRamDataOut),
   .Z(PDataOut)
);
```

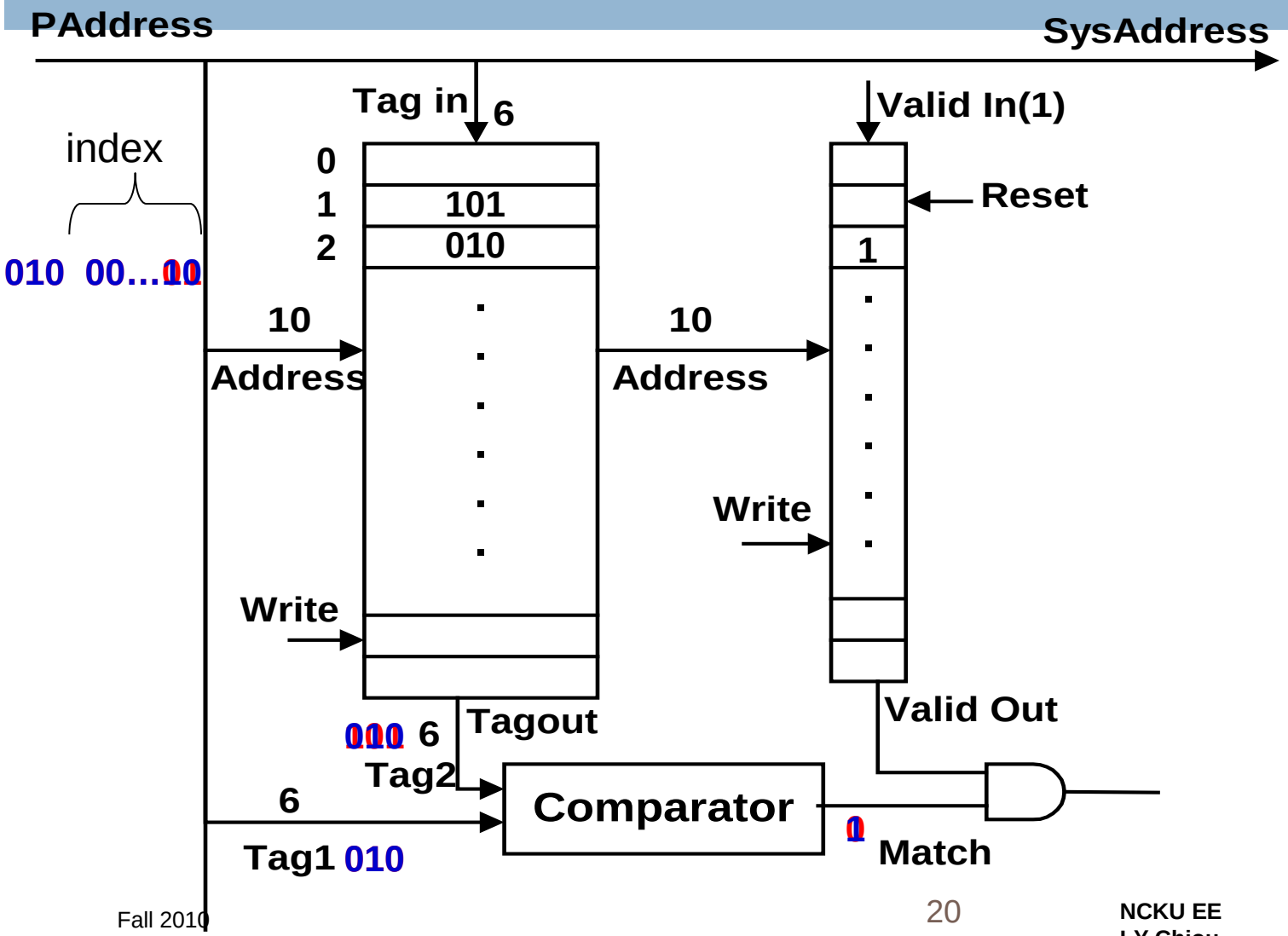
Main Cache Module (3/3)

```
DataRam DataRam (
        .Address(PAddress[`INDEX
        ]),
        .DataIn    (DataRamDataIn),
        .DataOut
        (DataRamDataOut),
        .Write     (Write),
        .Clk       (Clk)
        );
Comparator Comparator (
        .Tag1   (PAddress[`TAG]),
        .Tag2   (TagRamTag),
        .Match (Match)
        );
```

```
Control Control (
        .PStrobe(PStrobe),
        .PRW     (PRW),
        .Pready  (PReady),
        .Match   (Match),
        .Valid   (Valid),
        .CacheDataSelect(CacheDataSele
        ct),
        .PDataSelect  (PDataSelect),
        .SysDataOE   (SysDataOE),
        .Write              (Write),
        .PDataOE            (PDataOE),
        .SysStrobe          (SysStrobe),
        .SysRW              (SysRW),
        .Reset              (Reset),
        .Clk                (Clk)
        );

endmodule
```

Tag RAM and Valid Bit



Tag RAM Module

```
module TagRam(Address, TagIn, TagOut, Write, Clk);

input    [`INDEX]Address;
input    [`TAG]  TagIn;
output   [`TAG]  TagOut;
input            Write;
input            Clk;

reg      [`TAG]  TagOut;
reg      [`TAG]  TagRam     [0:`CACHESIZE-1];

always @  (negedge Clk)
        if (Write)
        TagRam[Address]=TagIn; // write

always @ (posedge Clk)
        TagOut = TagRam[Address]; // read
endmodule
```

Valid RAM module

```
module ValidRam (Address, ValidIn, ValidOut, Write,
        Reset, Clk );
input   [`INDEX]        Address;
input                   ValidIn;
output                  validOut;
input                   Write;
input                   Reset;
input                   Clk;
reg                 ValidOut;
reg     [`CACHESIZE-1:0]  ValidBits;
integer i;
always @ (negedge Clk) // Write
   if (Write && !Reset)
      ValidBits[Address]=ValidIn; // write
   else if (Reset)
      for (i=0;i<`CACHESIZE;i=i+1)
         ValidBits[i]=`ABSENT; //reset
always @ (posedge Clk)
         ValidOut = ValidBits[Address]; // read
```

- A single bit for each index location indicating whether that location contains valid data.
- The main difference between the ValidRAM model and the TagRAM model is that all bits in ValidBits need to be reset to zero when Reset is asserted.

Comparator module

```
module Comparator (Tag1,Tag2, Match);
    input   [`TAG]        Tag1;
    input   [`TAG]        Tag2;
    output                Match;
    wire    Match = (Tag1 == Tag2);
endmodule
```

- The "==" is used rather than "===" so that unknown values are propagated to the Match output correctly.

Cache Data RAM module

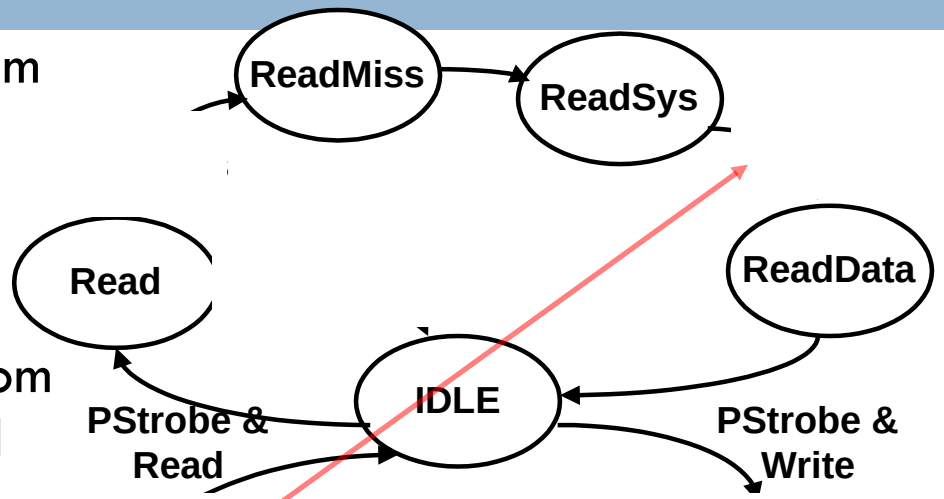
```
module DataRam(Address, DataIn, DataOut, Write, Clk);
input    [`INDEX]   Address;
input    [`DATA]    DataIn;
output   [`DATA]    DataOut;
input     Write;
input     Clk;
reg       [`DATA]   DataOut;
reg       [`DATA]   DataRam  [`CACHESIZE-1:0];
always @ (negedge Clk)
        if (Write)
              DataRam[Address]=DataIn; // write
always @ (posedge Clk)
              DataOut = DataRam [Address]; // read
endmodule
```

- The Data RAM is identical in function and timing to the Tag RAM except that it is 32 bits wide.

Cache controller

1.Read Hit: return data from cache.

2.Read Miss: fetch data from memory: return data and update cache.

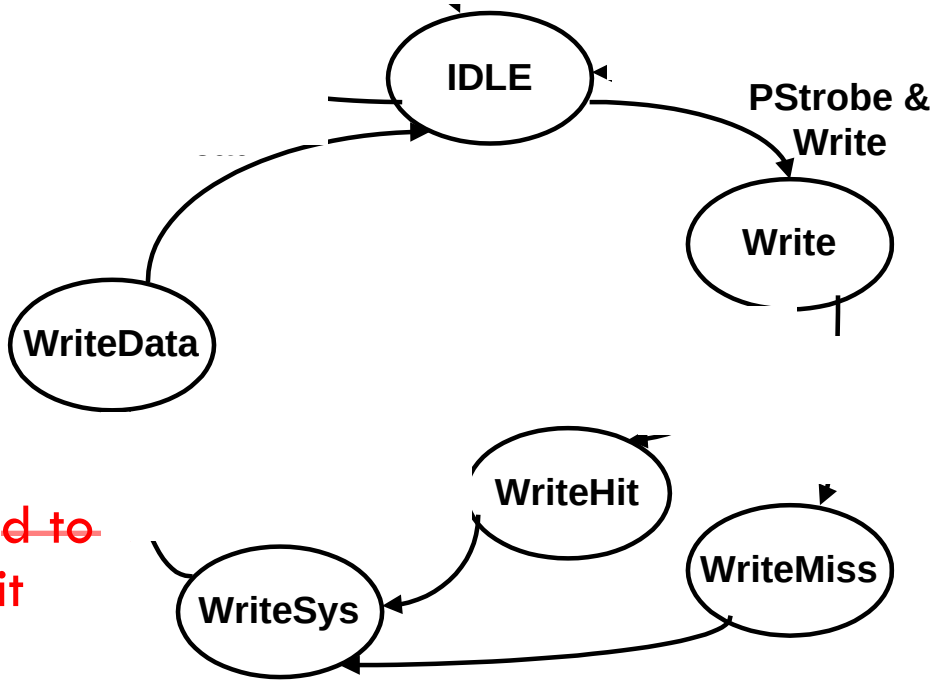


Note that a counter is used to count the number of wait states

Cache controller

3.Write Hit: write to cache and to memory.

4.Write Miss: write to memory only.



Note that a counter is used to count the number of wait states

Brief Description of Controller (I)

- Step 1: During IDLE, the state machine waits for the processor to start a cycle via Pstrobe.
  - **The signal is sampled at the rising edge of the clock.**
- Step 2: When Pstrobe is asserted, the address is looked up in the cache.
  - **If the tag TagRAM for this address matches the tag of the current processor address and the corres-ponding ValidBit is set, the data for this address is in the cache.**
  - **Based on whether the data is in cache and the state of the PRW line, the correct subsequent actions are determined.**
- All transactions on the system bus take a pre-defined number of cycles regardless of the type of transaction.
- During each bus operation, the wait state counter is loaded.
  - **The state machine does not advance state until the counter has overflowed.**
  - **The number of wait states is defined by the identifier WAITSTATES.**
- At the end of each bus transaction, the controller asserts PReady for one cycle to indicate to the processor that the transaction is finished.

Brief Description of Controller (II)

- Read Hit: Happens most often due to locality of reference and more reads than writes.
  - **In the cycle following the PStrobe, data is returned to the processor from the Data RAM along with the assertion of PReady.**
- Read Miss: A miss occurs if the location in the cache corresponding to current PAddress is either invalid or contains data for a different address.
  - **A memory cycle reads data from the main memory (after passing wait state).**
  - **The Tag RAM is updated to reflect the tag of the new data and ValidBit is set.**
- Write Hit: Since the cache protocol is write-through, both cache and main memory must be updated.
  - **A write cycle is performed on the system bus at the same time that data is written to the cache.**
  - **PReady is returned to the processor to indicate the completion of transaction.**
- Write Miss: Only the main memory is updated.
  - **This policy is chosen because the data read is most likely to be used again than the data is written.**
  - **It performs a write to memory only and returns PReady when the bus transaction is complete.**

Cache Controller Module


```
module Control (
        PStrobe, PRW, PReady,
        Match, Valid, Write,
        CacheDataSelect,
        PDataSelect,SysDataOE,
        PDataOE,
        SysStrobe, SysRW, Reset,
       ph1, ph2 );

input          PStrobe, PRW;
output         PReady;
input          Match, Valid;
output         Write;
output         CacheDataSelect;
output         PDataSelect;
output         SysDataOE, PDataOE;
output         SysStrobe, SysRW;
input          Reset, Clk;
```

```
wire  [1:0] WaitStateCtrInput =

            `WAITSTATES - 1;
reg   LoadWaitStateCtr;

WaitStateCtr WaitStateCtr(
       .load (LoadWaitStateCtr),
       .LoadValue (WaitStateCtrInput),
       .Carry (WaitStateCtrCarry),
       .Ph1 (Ph1),
       .Ph2 (Ph2)   );
reg  PReadyEnable;
reg  SysStrobe,SysRW;
reg  SysDataOE;
reg  Write, Ready;
reg  CacheDataSelect;
reg  PDataSelect, PDataOE;
reg  [3:0]  State, NextState;
```

Cont'd

Cache Controller Module (cont.)


```
initial State = 0;

paramerer
        STATE_IDLE = 0 ,
        STATE_READ = 1,
        STATE_READMISS = 2,
        STATE_READSYS = 3,
        STATE_READDATA = 4,
        STATE_WRITE = 5,
        STATE_WRITEHIT = 6,
        STATE_WRITEMISS = 7,
        STATE_WRITESYS = 8,
        STATE_WRITEDATA = 9;
```

Cont'd

.

```
always @ (posedge Clk)
                State = NextState;
always @ (State)
  if (Reset) NextState = `STATE_IDLE;
  else
  case (State)
     STATE_IDLE: begin
        if (PStrobe && PRW)
           NextState = `STATE_READ;
        else if (PStrobe && !PRW)
           NextState = `STATE_WRITE;
     end
     STATE_READ : begin
        if (Match && Valid)
           NextState = `STATE_IDLE;
                                  //read hit
        else
           NextState = STATE_READMISS;
       $display ("state = read");
     end
```

```
  `STATE_READMISS : begin
          NextState = `STATE_READSYS;
          $display("state = readmiss");
  end
  `STATE_READSYS : begin
       if (WaitStateCtrCarry)
            NextState = `STATE_READDATA;
       else NextState = `STATE_READSYS;
       $display ("state = readsys");
  end
  `STATE_READDATA : begin
          NextState = `STATE_IDLE;
          $display ("state = readdata");
  end
  `STATE_WRITE : begin
       if (Match && Valid)
         NextState = `STATE_WRITEHIT;
       else NextState = `STATE_WRITEMISS;
       $display ("state = WRITE");
  end
```

Cont'd

```
`STATE_WRITEHIT : begin
   NextState = `STATE_WRITESYS;
   $display ("state = WRITEHIT");
end
`STATE_WRITEMISS : begin
   NextState = `STATE_WRITESYS;
   $display ("state = WRITEmiss");
end
`STATE_WRITESYS :begin
   if (WaitStateCtrCarry)
     NextState = `STATE_WRITEDATA;
   else
     NextState = `STATE_WRITESYS;
   $display("state = WRITEsys");
end
`STATE_WRITEDATA : begin
   NextState = `STATE_IDLE;
   $display ("State = WRITEdata");
end
endcase
```

```
task OutputVec;
input [9:0] vector;
begin
   LoadWaitStateCtr=vector[9];
   PReadyEnable=vector[8];
   Ready=vector[7];
   Write=vector[6];
   SysStrobe=vector[5];
   SysRW=vector[4];
   CacheDataSelect=vector[3]
   PDataSelect=vector[2]
   PDataOE=vector[1]
   SysDataOE=vector[0];
end
endtask
```

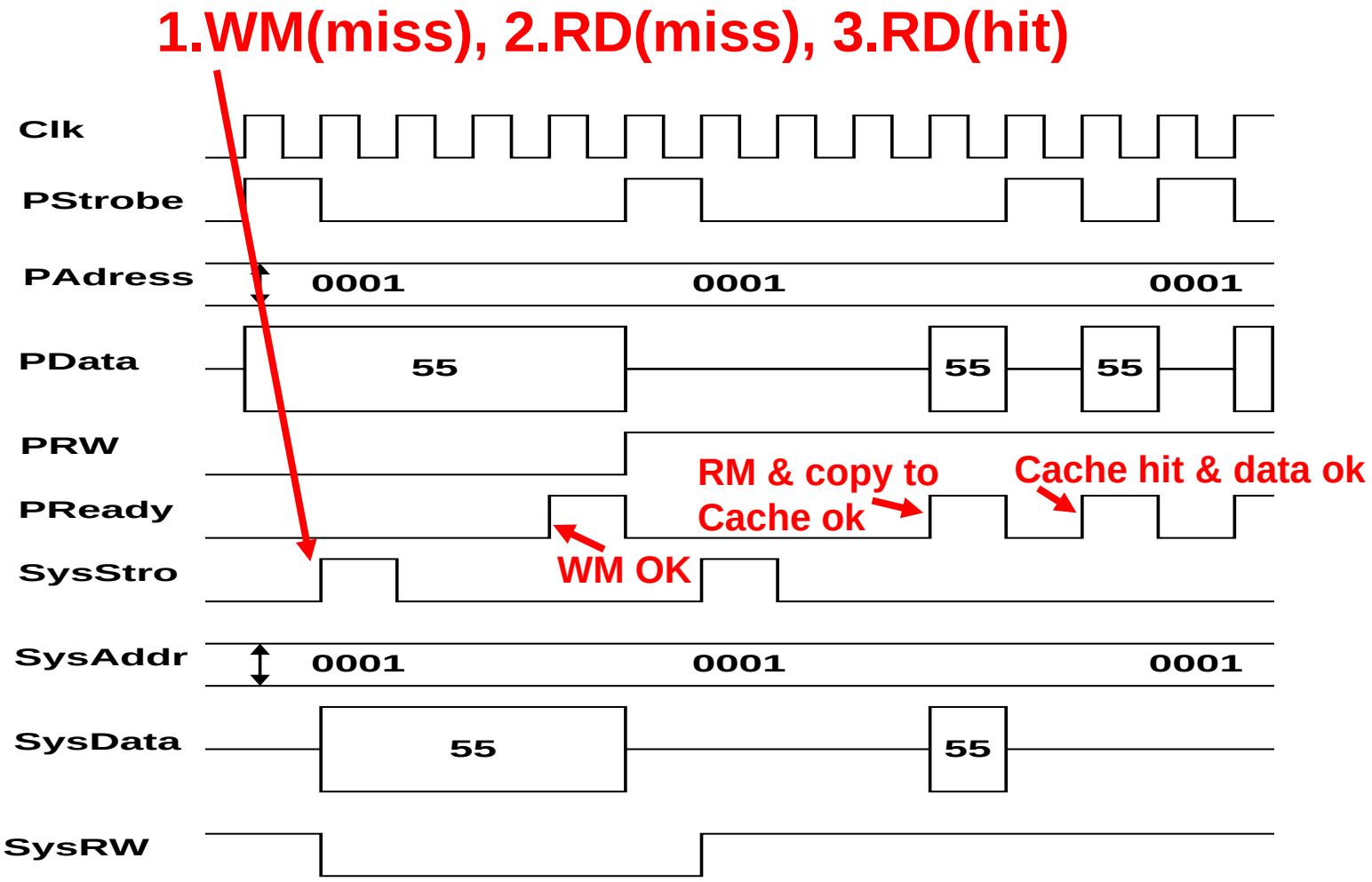
Cont'd

```
always @ (State)
case (State)                                  9876543210
   `STATE_IDLE:      OutputVec(10`b0000000000);
   `STATE_READ:      OutputVec(10`b0100000010);
   `STATE_READMISS:          OutputVec(10'b1000110010);
   `STATE_READSYS:           OutputVec(10'b0000010010);
   `STATE_READDATA:          OutputVec(10'b0011011110);
   `STATE_WRITEHIT:          OutputVec(10'b1001101100);
   `STATE_WRITE:             OutputVec(10'b0100000000);
   `STATE_WRITEMISS:         OutputVec(10'b1000100001);
   `STATE_WRITESYS:          OutputVec(10'b0000000001);
   `STATE_WRITEDATA:         OutputVec(10'b0011001101);
endcase                                                9876543210
wire PReady = (PReadyEnable && Match && Valid)
                  || Ready;
endmodule
```

System ready Cache ready

LoadWaitStateCtr=vector[9];
PReadyEnable=vector[8];
Ready=vector[7];
Write=vector[6];
SysStrobe=vector[5];
SysRW=vector[4];
CacheDataSelect=vector[3]
PDataSelect=vector[2]
PDataOE=vector[1]
SysDataOE=vector[0];

Result of a Short Test



Modeling Cache Memories

```
//Defines
`define READ 1'b1
`define WRITE 1'b0
`define CACHESIZE 1024
`define WAITSTATE 2'd2
`define ADDR 15:0
`define ADDRWIDTH 16
`define INDEX 9:0
`define TAG 15:10
`define DATA 31:0
`define DATAWIDTH 32
`define PRESENT 1'b1
`define ABSENT !`PRESENT

module cache(
        PStrobe, Paddress,
        PData, PRW, PReady,
        SysStrobe, SysAddress,
        SysData, SysRW,
        Reset, Clk
        );
    input   PStrobe;
    input   [`ADDR]   PAdress;
    input   [`DATA]   PData;
    input   PRW;
    output  PReady;

    output  SysStrobe;
    output  [`ADDR]   SysAddress;
    input   [`DATA]   SysData;
    output  SysRW;
    input   Reset;
    input   Clk;
```

continued

```
// Bidirectional Buses
wire PDataOE;
wire SysDataOE;
wire [`DATA] PDataOut;
wire [`DATA] PData =PDataOE ?
                PDataOut  :
    `DATAWIDTH`bz;
wire [`DATA] SysData= SysDataOE ?
                PData : `DATAWIDTH`bz;

wire [`ADDR] SysAddress =
    PAddress;
wire [`TAG]      TagRamTag;

wire       Write;
wire       Valid;
wire       CacheDataSelect;
wire       PDataSelect;
wire       Match;
```

```
TagRam TagRam(
  .Address (PAddress[`INDEX]),
  .TagIn     (PAddress[`TAG]),
  .TagOut   (TagRamTag[`TAG]),
  .Write     (Write),
  .Clk       (Clk)
);

ValidTam ValidRam(
  .Address
  (PAddress[`INDEX]),
  .ValidIn    (1`b1),
  .ValidOut (Valid),
  .Write     (Write),
  .Reset              (Reset),
  .Clk       (Clk)
);
```

continued

```
wire[`DATA] DataRamDataOut;
wire[`DATA] DataRamDataIn;
DataMux CacheDataInputMux(
   .S       (CacheDataSelect),
   .A       (SysData),
   .B       (PData),
   .Z       (DataRamDataIn)
);
DataMux PDatatMux(
   .S       (PDataSelect),
   .A       (SysData),
   .B       (DataRamDataOut),
   .Z       (PDataOut)
);
DataRam DataRam(
   .Address  (PAddress[`INDEX]),
   .DataIn          (DataRamDataIn),
   .DataOut (DataRamDataOut),
   .Write     (Write),
   .Clk       (Clk)
);
```

```
Comparator Comparator(
              .Tag1     (PAddress[`TAG]),
              .Tag2     (TagRamTag),
              .Match    (Match)
);
Control Control(
      .PStrobe          (PStrobe),
      .PRW  (PRW),
      .PReady           (PReady),
      .Match(Match),
      .Valid  (Valid),
      .CacheDataSelect (CacheDataSelect),
      .PDataSelect (PDataSelect),
      .SysDataOE        (SysDataOE),
      .Write  (Write),
      .PDataOE          (PDataOE),
      .SysStrobe        (SysStrobe),
      .SysRW            (SysRW),
      .Reset (Reset),
      Clk     (Clk)
);
endmodule
```

continued

```
module Control(
        PStrobe, PRW, Ready,
        Match, Valid, Write,
        CacheDataSelect,
        PDataSelect,
        SysDataOE, PDataOE,
        SysStrobe, SysRW, Reset,
        Clk,
);

input PStrobe, PRW;
output PReady;
input Match, Valid;
output Write;
output CacheDataSelect;
output PDataSelect;
output SysDataOE, PDataOE;
output SysStrobe, SysRW;
input Reset;
input Clk;
```

```
wire [1:0] WaitStateCtrInput =
                        ` WAITSTATES -2`d1;
wire WaitStateCtrCarry;
reg LoadWaitStateCtr;

WaitStateCtr WaitStateCtr(
      .Load  (LoadWaitStateCtr),
      .LoadValue       (WaitStateCtrCarry),
      .Carry (WaitStateCtrCarry),
      .Clk     (Clk)
   );
reg PReadyEnable;
reg SysStrobe, SysRW;
reg SysDataOE;
reg Write;
reg Ready;
reg CacheDataSelect, PDataSelect;
reg PDataOE;
reg [3:0] State; NextState
```

continued

```
reg[3:0]NextState;

`define STATE_IDLE          4`d0
`define STATE_READ          4'd1
`define STATE_READMISS      4'd2
`define STATE_READSYS       4'd3
`define STATE_READDATA      4'd4
`define STATE_WRITE         4'd5
`define STATE_WRITEHIT      4'd6
`define STATE_WRITEMISS     4'd7
`define STATE_WRITESYS      4'd8
`define STATE_WRITEDATA     4'd9

always @ (posedge Clk)
     State  = Reset ?
                `STATE_IDLE : NextState;

always @ (State or PStrobe or
            PRW or  Match or
            Valid or  WaitStateCtrCarry)
```

```
Case (State)
   `STATE_IDLE: begin
        if (PStrobe && PRW == `READ)
                NextState = `STATE_READ;
        else if (PStrobe && PRW ==` WRITE)
                NextState = `STATE_WRITE;
        else  NextState = `STATE_IDLE;
   end
   `STATE_READ : begin
        if (Match && Valid)
                NextState = `STATE_IDLE;
        else   NextState = `STATE_READMISS;
   end
   `STATE_READMISS : begin
                NextState = `STATE_READSYS;
   end
   `STATE_READSYS : begin
        if(WaitStateCtrCarry)
                NextState = `STATE_READDATA;
        else   NextState = `STATE_READSYS;
   end
```

continued

```
`STATE_READDATA : begin
        NextState = `STATE_IDEL;
end
`STATE_WRITE : begin
      if (Match && Valid)
        NextState = `STATE_WRITEHIT;
     else
        NextState = `STATE_READMISS;
end
`STATE_WRITEHIT : begin
        NextState = `STATE_WRITESYS;
end
`STATE_WRITEMISS : begin
        NextState = `STATE_WRITESYS;
end
`STATE_WRITESYS : begin
      if (WaitStateCtrCarry)
        NextState = `STATE_WRITEDATA;
      else NextState = `STATE_WRITESYS;
end
```

```
      `STATE_WRITEDATA : begin
            NextState = `STATE_IDLE;
      end
      default:NextState = `STATE_IDLE;
   endcase
task OutputVec;
input [9:0] vector;
begin
   {LoadWaitStateCtr, PReadyEnable,
   Ready, Write, SysStrobe, SysRW,
   CacheDataSelect,
   DataSelect,PDataOE, SysDataOE} =
   vector;
end
endtask
```

continued

```
always @ (State)
case (State)
      `STATE_IDLE:              OutputVec(10`b0000000000);
      `STATE_READ:    OutputVec(10'b0100000010);
      `STATE_READMISS:          OutputVec(10'b1000110010);
      `STATE_READSYS:           OutputVec(10'b0000010010);
      `STATE_READDATA:          OutputVec(10'b0011011110);
      `STATE_WRITEHIT:          OutputVec(10'b1001101100);
      `STATE_WRITE:   OutputVec(10'b0100000000);
      `STATE_WRITEMISS:         OutputVec(10'b1000100001);
      `STATE_WRITESYS:          OutputVec(10'b0000000001);
     `STATE_WRITEDATA:          OutputVec(10'b0011001101);
      default:              OutputVec(10'b0000000000);
endcase

wire PReady = (PReadyEnable && Match && Valid) || Ready;

endmodule

module DataMux(S, A, B, Z);
  input S; //Select Line
  input [`DATA] A, B, Z;
  wire [`DATA] Z = S ? A: B;
endmodule
```

continued

```
module WaitStateCtr(Load,
                    LoadValue, Carry, Clk);

input Load;
input [1:0] LoadValue;
output Carry;
input Clk;
reg [1:0] Count;

always @ (posedge Clk)
   if (Load)
      Count = LoadValue;
   else
      Count = Count - 2'b1;

 wire Carry = Count == 2'b0;
endmodule
```

```
module TagRam(Address, TagIn, TagOut,
                          Write, Clk);
input [`INDEX]Address;
input [`TAG] TagIn;
output [`TAG] TagOut;
input Write;
input Clk;

reg [`TAG] TagOut;
reg [`TAG] TagRam [`CACHESIZE-1:0];

always @ (negedge Clk)
   if (Write) TagRam[Address]=TagIn; //write

always @ (posedge Clk)
      TagOut = TagRam[Address]; //read
 endmodule
```

continued

```
module ValidRam(
        Address,
        ValidIn,
        ValidOut,
        Write,
        Reset,
        Clk
);
input       [`INDEX]Address;
input       ValidIn;
output      ValidOut;
input       Write;
input       Reset;
input       Clk;

reg ValidOut;
reg [`CACHESIZE-1:0] ValidBits;

integer i;
```

```
always @ (posedge Clk)
    if (Write && !Reset)
        ValidBits[Address]=ValidIn; // write
    else if (Reset)
        for (i=0; i<`CACHESIZE;i=i+1)
            ValidBits[I]=`ABSENT; //reset

always @ (posedge Clk)
    validOut = ValidBits[Address]; //read

endmodule
```

continued

```
module DataRam(Address, DataIn,
                        DataOut, Write, Clk);

input        [`INDEX] Address;
input        [`DATA] DataIn;
output      [`DATA] DataOut;
input        Write;
input  Clk;

reg [`DATA] Dataout;
reg [`Data] Ram [`CACHESIZE-1:0];

always @ (posedge Clk)
   if(Write)
      Ram[Address]=DataIn; //write

   always @ (posedge Clk)
      DataOut =  Ram[Address]; //read
endmodule
```

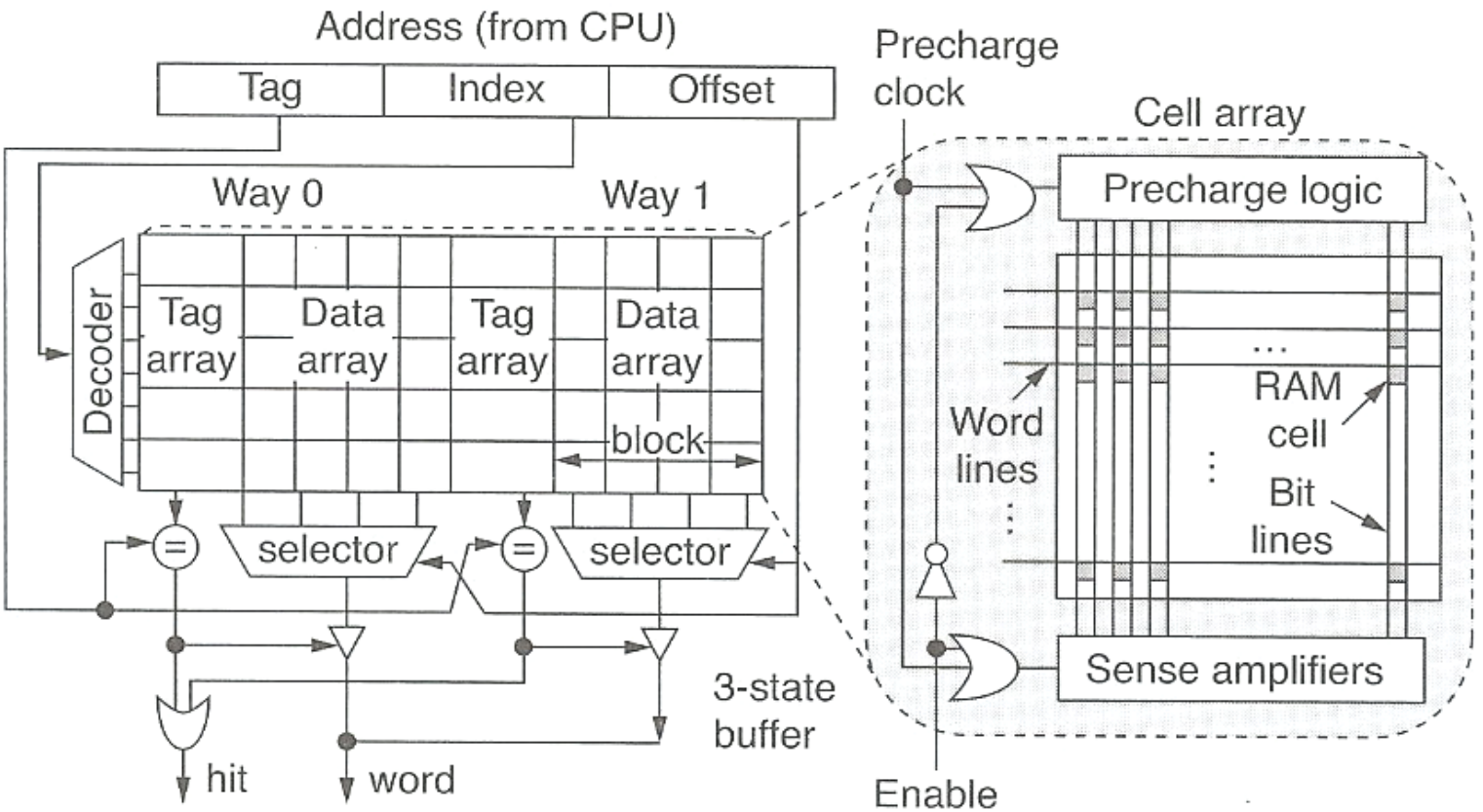
```
module Comparator(Tag1, Tag2,
   Match);

input [`TAG] Tag1;
input [`TAG] Tag2;
output Match;

wire Match = Tag1 == Tag2;

endmodule
```

2-Way Cache Organization



4-Way Cache Organization