# Project

## CMPEN 331 – Computer Organization and Design

**Late submission will not be accepted and will result in not getting any credits for the project**

**Item 17 is for Honor Option students only**

In this project, the students will implement a Single-Cycle CPU using the Xilinx design package for FPGAs for the R-type, I-type and J-type instructions. You can use any information available in previous labs if needed.

1. **Instruction Fetch**
   The same as in previous labs.

2. **Instruction Execution**
   The same as in previous labs.

3. **R-Format Instructions**
   The same as in previous labs.

4. **I-Format Instructions**
   The same as in previous labs.

5. **Conditional Branch Instructions**
   The same as in previous labs.

6. **Jump Instructions**
   The same as in previous labs.

7. **Single-Cycle CPU**
   By summarizing the previous labs, Figure 1 shows the schematic of the single-cycle CPU plus the instruction memory and the data memory. Putting the memory modules outside, Figure 2 shows the single-cycle computer that consists of a single-cycle CPU and two memory modules. The reason why we use separate instruction memory and data memory is that the single-cycle CPU completes the execution of an instruction, including instruction fetch and data memory access, in one clock cycle. The CPU consists of a data path and a control unit. The control unit is a component that directs the operations of the data path.

   i. **Selection for Next PC**

   The PC will be updated on the rising edge of the clock in the single-cycle CPU. If the current instruction is neither a branch nor a jump, PC+4 will be written to the PC. A 4-to-1 multiplexer can be used to select an address for the next PC as shown in Figure 4. The input 0 of the multiplexer is PC+4; input 1 is the branch target address; input 2 is the jump target address coming from the register rs of the register file; and input 3 is the jump target address coming from addr and PC+4. The 2-bit pcsrc is the selection signal of the multiplexer.

   ii. **Selection for ALU Input A (Selection Signal: shift)**

   Shift instruction use sa as the shift amount that will be sent to the input a of the ALU for the operation. Other instructions may use the value in the register rs of the register file. If shift is a 1, sa is selected. Otherwise, qa of the register file is selected.

**iii. Selection for ALU Input B (Selection Signals: aluimn and regrt)**

The multiplexer in the front of the ALU input b has 32 bits and its selection signal is aluim. If aluimm is a 1, the extended immediate is selected. Otherwise, qb of the register file is selected. The multiplexer in front of the register file's input wn has 5 bits, and its selection signal is regrt. If regrt is a 1, rt is selected. Otherwise, rd is selected.

**iv. Selection for Register File Inputs (Selection Signals: m2reg and jal)**

The data that will be written to the register file may be output of the ALU, the data in the data memory, or the return address (for jal instruction). The destination register number may be rd, rt, or a constant 31 (for jal instruction). The subroutine call jal instruction saves the return address to register $31. Two 32-bit 2-to-1 are used for selecting the data that will be written to the register file as shown in Figure 1. The selection signal of the multiplexeron the right side of the figure is m2reg. If m2reg is a 1, the data read from memory is selected. The selection signal of the multiplexer on the left side is jal. If jal is a1, the return address p4 is selected. For the jal instruction, because the destination register number 31 does not appear in the instruction, we must generate it by hardware (the component f in the figure). The following Verilog HDL code is a hardware implementation for generating the 5-bit destination register number:

```
assign wn = reg_dest | {5{jal}};
```

it performs a bitwise logical OR operation. If jal is a 1, a 5-bit patern 11111 (decimal 31) will be assigned to the destination register number wn. Otherwise, reg_dest, which is rd or rt, will be assigned to wn.

## 8. Logic design of the Control Unit

The control unit generates control signals based on the instruction that is currently being executed. The first step of designing the control unit is to decode the instruction based on the instruction's opcode (op), as listed in Table 5. The function code (func) needs to be checked also if the instruction has an R-format. In order to avoid conflict with the keywords of Verilog HDL, an i_ is prefixed to the name of each instruction. From the table, we can get each instruction decode expression as some examples below. A temporary wire, rtype, is used for decoding all the R-format instructions.

$$\text{rtype} = \overline{\text{op}[5]}\ \overline{\text{op}[4]}\ \overline{\text{op}[3]}\ \overline{\text{op}[2]}\ \overline{\text{op}[1]}\ \overline{\text{op}[0]}$$

$$\text{i\_add} = \text{rtype}\ \text{func}[5]\ \overline{\text{func}[4]}\ \overline{\text{func}[3]}\ \overline{\text{func}[2]}\ \overline{\text{func}[1]}\ \overline{\text{func}[0]}$$

$$\text{i\_sll} = \text{rtype}\ \overline{\text{func}[5]}\ \overline{\text{func}[4]}\ \overline{\text{func}[3]}\ \overline{\text{func}[2]}\ \overline{\text{func}[1]}\ \overline{\text{func}[0]}$$

$$\text{i\_jr} = \text{rtype}\ \overline{\text{func}[5]}\ \overline{\text{func}[4]}\ \text{func}[3]\ \overline{\text{func}[2]}\ \overline{\text{func}[1]}\ \overline{\text{func}[0]}$$

$$\text{i\_andi} = \overline{\text{op}[5]}\ \overline{\text{op}[4]}\ \text{op}[3]\ \text{op}[2]\ \overline{\text{op}[1]}\ \overline{\text{op}[0]}$$

$$\text{i\_jal} = \overline{\text{op}[5]}\ \overline{\text{op}[4]}\ \overline{\text{op}[3]}\ \overline{\text{op}[2]}\ \text{op}[1]\ \text{op}[0]$$

The control signals can be classified into the following four categories: (i) the selection signals of the multiplexers; (ii) the ALU operation control signal; (iii) the register file and memory write enables; and (iv) others such as sign or zero extension.

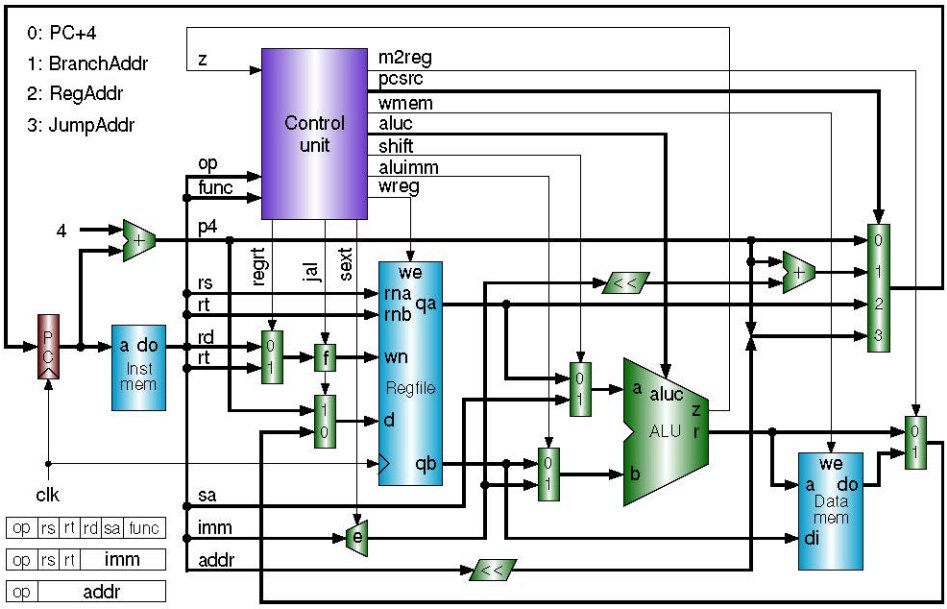


Figure 1 Block diagram of a single-cycle CPU

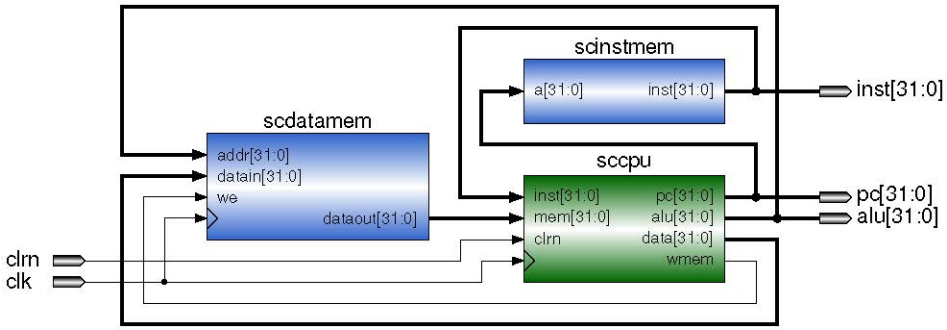


Figure 2 Block diagram of single-cycle computer

9. **Table 1** lists the names and usages of the 32 registers in the register file.

Table 1 MIPS general purpose register

| Register Name | Register Number | Usage |
|---|---|---|
| $zero | 0 | Constant 0 |
| $at | 1 | Reserved for assembler |
| $v0, $v1 | 2, 3 | Function return values |
| $a0 - $a3 | 4 – 7 | Function argument values |
| $t0 - $t7 | 8 – 15 | Temporary (caller saved) |
| $s0 - $s7 | 16 – 23 | Temporary (callee saved) |
| $t8, $t9 | 24, 25 | Temporary (caller saved) |
| $k0, $k1 | 26, 27 | Reserved for OS Kernel |
| $gp | 28 | Pointer to Global Area |
| $sp | 29 | Stack Pointer |
| $fp | 30 | Frame Pointer |
| $ra | 31 | Return Address |

10. **Table 2** lists some MIPS instructions that will be implemented in our CPU

Table 2 MIPS integration instruction

| Inst. | [31:26] | [25:21] | [20:16] | [15:11] | [10:6] | [5:0] | Meaning |
|---|---|---|---|---|---|---|---|
| add | 000000 | rs | rt | rd | 00000 | 100000 | Register add |
| sub | 000000 | rs | rt | rd | 00000 | 100010 | Register subtract |
| and | 000000 | rs | rt | rd | 00000 | 100100 | Register AND |
| or | 000000 | rs | rt | rd | 00000 | 100101 | Register OR |
| xor | 000000 | rs | rt | rd | 00000 | 100110 | Register XOR |
| sll | 000000 | 00000 | rt | rd | sa | 000000 | Shift left |
| srl | 000000 | 00000 | rt | rd | sa | 000010 | Logical shift right |
| sra | 000000 | 00000 | rt | rd | sa | 000011 | Arithmetic shift right |
| jr | 000000 | rs | 00000 | 00000 | 00000 | 001000 | Register jump |
| addi | 001000 | rs | rt | | Immediate | | Immediate add |
| andi | 001100 | rs | rt | | Immediate | | Immediate AND |
| ori | 001101 | rs | rt | | Immediate | | Immediate OR |
| xori | 001110 | rs | rt | | Immediate | | Immediate XOR |
| lw | 100011 | rs | rt | | offset | | Load memory word |
| sw | 101011 | rs | rt | | offset | | Store memory word |
| beq | 000100 | rs | rt | | offset | | Branch on equal |
| bne | 000101 | rs | rt | | offset | | Branch on not equal |
| lui | 001111 | 00000 | rt | | immediate | | Load upper immediate |
| j | 000010 | | | address | | | Jump |
| jal | 000011 | | | address | | | Call |

11. **Table 3** lists all the control signals of the single cycle CPU

Table 3 Control signals of single cycle CPU

| Signal | Meaning | Action |
|---|---|---|
| `wreg` | Write register | 1: write; 0: do not write |
| `regrt` | Destination register is `rt` | 1: select `rt`; 0: select `rd` |
| `jal` | Subroutine call | 1: is `jal`; 0: is not `jal` |
| `m2reg` | Save memory data | 1: select memory data; 0: select ALU result |
| `shift` | ALU A uses `sa` | 1: select `sa`; 0: select register data |
| `aluimm` | ALU B uses immediate | 1: select immediate; 0: select register data |
| `sext` | Immediate sign extend | 1: sign-extend; 0: zero extend |
| `aluc[3:0]` | ALU operation control | x000: ADD; x100: SUB; x001: AND<br>x101: OR; x010: XOR; x110: LUI<br>0011: SLL; 0111: SRL; 1111: SRA |
| `wmem` | Write memory | 1: write memory; 0: do not write |
| `pcsrc[1:0]` | Next instruction address | 00: select PC+4; 01: branch address<br>10: register data; 11: jump address |

12. **Table 4** shows the truth table of the control signals

Table 4 Truth table of the control signals

| Inst. | z | wreg | regrt | jal | m2reg | shift | aluimm | sext | aluc[3:0] | wmem | pcsrc[1:0] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| i_add | x | 1 | 0 | 0 | 0 | 0 | 0 | x | x000 | 0 | 00 |
| i_sub | x | 1 | 0 | 0 | 0 | 0 | 0 | x | x100 | 0 | 00 |
| i_and | x | 1 | 0 | 0 | 0 | 0 | 0 | x | x001 | 0 | 00 |
| i_or | x | 1 | 0 | 0 | 0 | 0 | 0 | x | x101 | 0 | 00 |
| i_xor | x | 1 | 0 | 0 | 0 | 0 | 0 | x | x010 | 0 | 00 |
| i_sll | x | 1 | 0 | 0 | 0 | 1 | 0 | x | 0011 | 0 | 00 |
| i_srl | x | 1 | 0 | 0 | 0 | 1 | 0 | x | 0111 | 0 | 00 |
| i_sra | x | 1 | 0 | 0 | 0 | 1 | 0 | x | 1111 | 0 | 00 |
| i_jr | x | 0 | x | x | x | x | x | x | xxxx | 0 | 10 |
| i_addi | x | 1 | 1 | 0 | 0 | 0 | 1 | 1 | x000 | 0 | 00 |
| i_andi | x | 1 | 1 | 0 | 0 | 0 | 1 | 0 | x001 | 0 | 00 |
| i_ori | x | 1 | 1 | 0 | 0 | 0 | 1 | 0 | x101 | 0 | 00 |
| i_xori | x | 1 | 1 | 0 | 0 | 0 | 1 | 0 | x010 | 0 | 00 |
| i_lw | x | 1 | 1 | 0 | 1 | 0 | 1 | 1 | x000 | 0 | 00 |
| i_sw | x | 0 | x | x | x | 0 | 1 | 1 | x000 | 1 | 00 |
| i_beq | 0 | 0 | x | x | x | 0 | 0 | 1 | x010 | 0 | 00 |
| i_beq | 1 | 0 | x | x | x | 0 | 0 | 1 | x010 | 0 | 01 |
| i_bne | 0 | 0 | x | x | x | 0 | 0 | 1 | x010 | 0 | 01 |
| i_bne | 1 | 0 | x | x | x | 0 | 0 | 1 | x010 | 0 | 00 |
| i_lui | x | 1 | 1 | 0 | 0 | x | 1 | x | x110 | 0 | 00 |
| i_j | x | 0 | x | x | x | x | x | x | xxxx | 0 | 11 |
| i_jal | x | 1 | x | 1 | x | x | x | x | xxxx | 0 | 11 |

We take the lw instruction as an example to explain the truth table of Table 5. The ALU performs addition to calculate the memory address (aluc[3:0] = x000); one operand of the addition comes from register rs of the register file (shift = 0); the other operand is the immediate (aluimm = 1); the immediate is sign extended (sext = 1); the result will be written to register file (wreg = 1); it is the memory data (m2reg = 1); the destination register number is rt (regrt = 1); it is not a jal instruction (jal = 0); it does not write memory (wmem = 0); and the address of the next instruction is PC+4 (pcsrc[1:0] = 00).

Table 5 Instruction Decode

| R-format | | | I- and J-format | |
|---|---|---|---|---|
| Inst. | op[5:0] | func[5:0] | Inst. | op[5:0] |
| i_add | 000000 | 100000 | i_addi | 001000 |
| i_sub | 000000 | 100010 | i_andi | 001100 |
| i_and | 000000 | 100100 | i_ori | 001101 |
| i_or | 000000 | 100101 | i_xori | 001110 |
| i_xor | 000000 | 100110 | i_lw | 100011 |
| i_sll | 000000 | 000000 | i_sw | 101011 |
| i_srl | 000000 | 000010 | i_beq | 000100 |
| i_sra | 000000 | 000011 | i_bne | 000101 |
| i_jr | 000000 | 001000 | i_lui | 001111 |
| | | | i_j | 000010 |
| | | | i_jal | 000011 |

## 13. Test Program and Simulation Waveform

Write a Verilog code that implement the following instructions to verify the correctness of your design. The code should be used to initialize scinstmem block shown in Figure 2. The register file should be all initialized to zeros.

```
// instruction    // (pc) label    instruction
32'h3c010000;     // (00) main:    lui  $1, 0
32'h34240050;     // (04)          ori  $4, $1, 80
32'h20050004;     // (08)          addi $5, $0,  4
32'h0c000018;     // (0c) call:    jal  sum
32'hac820000;     // (10)          sw   $2, 0($4)
32'h8c890000;     // (14)          lw   $9, 0($4)
32'h01244022;     // (18)          sub  $8, $9, $4
32'h20050003;     // (1c)          addi $5, $0,  3
32'h20a5ffff;     // (20) loop2:   addi $5, $5, -1
32'h34a8ffff;     // (24)          ori  $8, $5, 0xffff
32'h39085555;     // (28)          xori $8, $8, 0x5555
32'h2009ffff;     // (2c)          addi $9, $0, -1
32'h312affff;     // (30)          andi $10,$9, 0xffff
32'h01493025;     // (34)          or   $6, $10, $9
32'h01494026;     // (38)          xor  $8, $10, $9
32'h01463824;     // (3c)          and  $7, $10, $6
32'h10a00001;     // (40)          beq  $5, $0, shift
32'h08000008;     // (44)          j    loop2
32'h2005ffff;     // (48) shift:   addi $5, $0, -1
32'h000543c0;     // (4c)          sll  $8, $5, 15
32'h00084400;     // (50)          sll  $8, $8, 16
32'h00084403;     // (54)          sra  $8, $8, 16
32'h000843c2;     // (58)          srl  $8, $8, 15
32'h08000017;     // (5c) finish:  j    finish
32'h00004020;     // (60) sum:     add  $8, $0, $0
32'h8c890000;     // (64) loop:    lw   $9, 0($4)
32'h20840004;     // (68)          addi $4, $4,  4
32'h01094020;     // (6c)          add  $8, $8, $9
32'h20a5ffff;     // (70)          addi $5, $5, -1
32'h14a0fffb;     // (74)          bne  $5, $0, loop
32'h00081000;     // (78)          sll  $2, $8, 0
32'h03e00008;     // (7c)          jr   $31
```

The test data should be stored in the data memory: see the following code

```
module scdatamem (clk,dataout,datain,addr,we);        // data memory, ram
    input         clk;                    // clock
    input         we;                     // write enable
    input  [31:0] datain;                 // data in (to memory)
    input  [31:0] addr;                   // ram address
    output [31:0] dataout;                // data out (from memory)
    reg    [31:0] ram [0:31];             // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]];      // use word address to read ram
    always @ (posedge clk)
        if (we) ram[addr[6:2]] = datain; // use word address to write ram
    integer i;
    initial begin                         // initialize memory
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data          // (byte_addr)
        ram[5'h14] = 32'h000000a3;        // (50hex)
        ram[5'h15] = 32'h00000027;        // (54hex)
        ram[5'h16] = 32'h00000079;        // (58hex)
        ram[5'h17] = 32'h00000115;        // (5chex)
        // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
endmodule
```

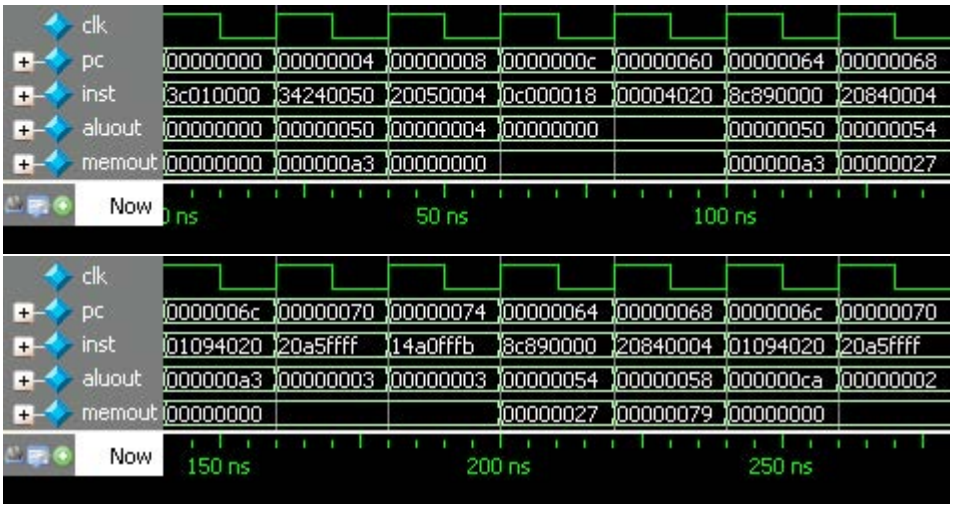


Figure 3 Part of the Waveform of a single-cycle CPU

14. Write a report that contains the following:
    a. Your Verilog® design code. Use:
        i. Device Family: Zynq-7000
        ii. Device: XC7Z010- -1CLG400C or ZyboBoard
    b. Your Verilog Test Bench code. Add "`timescale 1ns/1ps" as the first line of your test bench file.
    c. The waveforms resulting from the verification of your design. Figure 3 shows the simulation till ~ 275 ns, you need to show the simulation until 1350 ns. The clock period is 20 ns as shown in Figure 3. The memout signal in Figure 3 is the dataout signal in Figure 2.
    d. The design schematics from the Xilinx synthesis of your design. Do not use any area constraints.
    e. Snapshot of the I/O Planning
    f. Snapshot of the floor planning
    g. Generate the bitstream.

h. The design should be free from errors when synthesized, implemented and generated of bit stream.

15. **REPORT FORMAT:** Free form, but it must be:
    a. One report per student.
    b. Have a cover sheet with identification: Title, Class, Your Name, etc.
    c. You have to write an abstract at the beginning of the project report to describe what you are doing in the project.
    d. You should include an introduction for the project explaining with diagrams the connection between all the stages and what would be the benefit of using that architecture in the computer organization field.
    e. Use Microsoft word and it should be uploaded in word format not PDF. If you used LaTex, you should upload the Tex file in addition to the PDF file.
    f. Single spaced

16. You have to upload the whole project design file zipped with the word file.

17. **For students who took this class as an (honor option). In addition to all of the above requirements, you need to design the following:**
    **i.** Data Cache Design
    Find below a simple data cache design. The data cache is located in between the CPU and the memory as shown in Figure 4. We prefix the names of the signals that connect the CPU and cache with character "p" (processor) and the names of the signals connecting cache and the memory with "m" (memory). Ignoring the prefixes "p" and "m", a is a 32-bit memory address; dout is 32-bit data out; din is a 32-bit data in; strobe is a 1-bit strobe indicating a memory access; rw indicates that the memory access is a read (rw= 0) or a write (rw = 1); and ready indicates whether ready or not. Although we use the name p_ready, its meaning is that the data for the CPU is ready. uncached = 1 if I/O is accessed. It prohibits updating the cache. The direct mapping method is used to design the cache. The cache size is 256 bytes, and the block size is one word or 4 bytes. The memory address has 32 bits. Therefore, there are 64 blocks, and the block index has 6 bits and the tag has 24 bits. The write through policy and the write allocate strategy are used. The detailed block diagram of the data cache is shown in Figure 5. There are three cache RAMs: valid RAM, tag RAM and data RAM. Valid RAM needs to be cleared, whereas the others need not. Two multiplexers are used in the cache data input port and CPU data input port. The data written to cache (c_din) comes from either the CPU (for store instruction) or memory (for cache miss). The data sent to the CPU (p_din) comes from either cache (for cache hit) or memory (for cache miss).

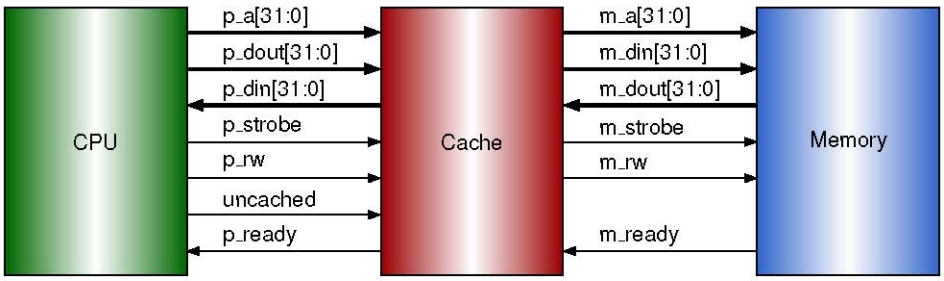


Figure 4 Cache in between the CPU and the main memory

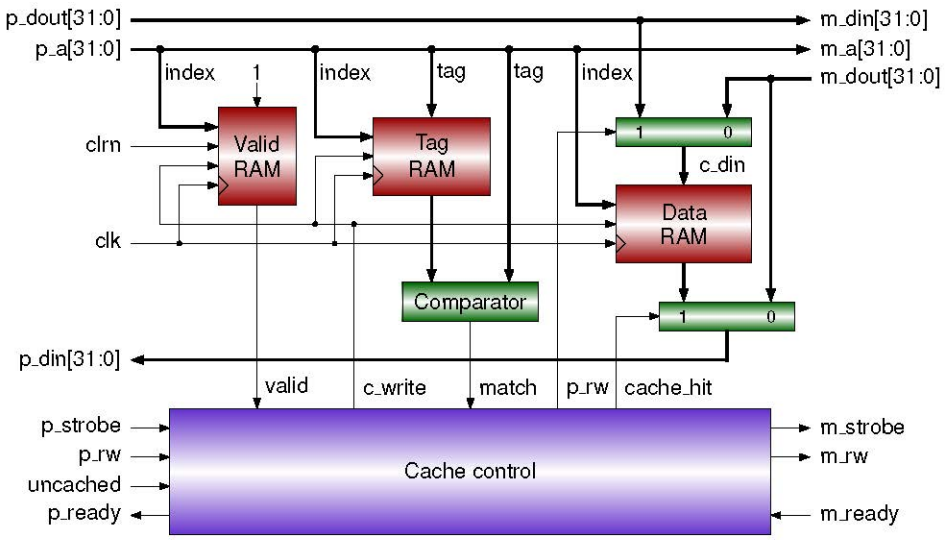


Figure 5 Block diagram of the write through cache

Figure 6 shows the simulation waveforms of the cache read. The CPU loads a data word from the memory location 0x00000100. This results in a cache miss. The missed data word is loaded from the memory and written to cache. When the CPU loads the data word from the same memory location again, a cache hit occurs. Given Figure 4, 5 and 6, writ a Verilog code of the data cache read design and its test Verilog code. You can assume any missing information.

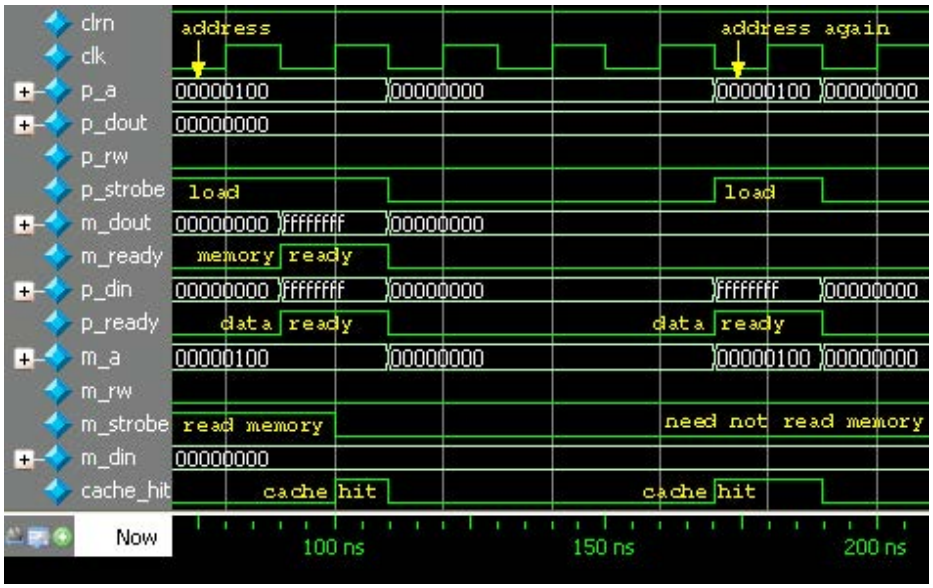


Figure 6 Waveform of data cache read