

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机体系结构 |
| 姓 名： | 蔡武威 |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术系 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3130101796 |
| 指导教师： | 姜晓红 |

2016 年 6月 11日

**浙江大学实验报告**

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： Lab5: Implement a CPU accessing a two level memory hierarchy

学生姓名： 蔡武威 专业： 计算机科学与技术 学号： 3130101796

同组学生姓名： None 指导老师： 姜晓红

实验地点： 曹二期301 实验日期： 2016 年 6 月 10 日

1. 实验目的和要求
2. Experiment Purpose

* Understanding the principle of Memory Hierarchy
* Implement CPU with cache and memory by steps
* Implement a CPU accessing memory in multiple cycle cpu
* Implement a CPU with cache- memory two level hierarchy
* Part i: CPU access Mem in multiple cycles
* Understand the principle of ACPU accessing Mem. in multiple cycles
* Master the design methods of pipelined CPU accessing Mem. In multiple cycle
* Master methods of program verification of Pipelined CPU accessing Mem. In multiple cycle
* Part ii: Cache design
* Understand Cache Line
* Understand the principle of Cache Management Unit(CMU) and State Machine of CMU
* Master the design methods of CMU
* Master the design methods of Cache Line
* Master verification methods of Cache Line

1. Experiment Task and Requirements

* Part i CPU access Mem in multiple cycles
* Design of Pipelined CPU accessing Mem. In multiple cycle
* Redesign Inst. ROM & Data RAM
* Modify CPU Controller
* Modify datapath
* Verify the Pipeline CPU with program and observe the execution of program
* Part ii: Cache design
* Design of Cache Line and CMU
* Verify the Cache Line and CMU
* Observe the Waveform of Simulation

1. 实验内容和原理

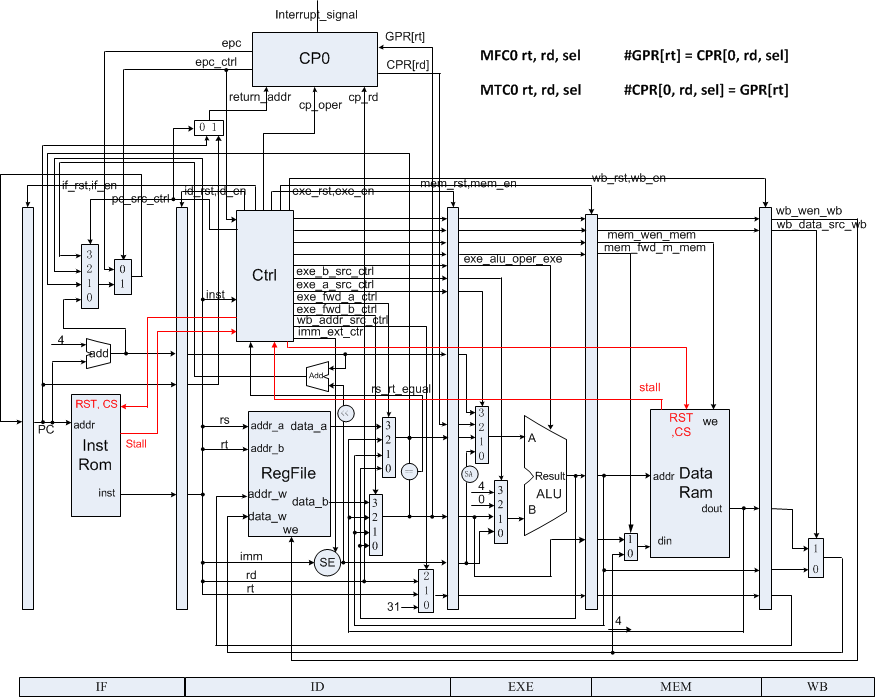
* The Verilog code can be divided into the following modules:
* Part i CPU access Mem in multiple cycles

1. mips\_top: The top module of the whole project.
2. clk\_gen: To generate different clocks.
3. anti\_jitter: To avoid the jitter of the button.
4. display: To display some information, including instruction address of which is IF\ID\EXE\MEM\WB stage, register values and some other important information.
5. mips: The main module to implement a MIPS CPU, which has also some sub-module, such as mips\_core, inst\_rom, data\_ram. Also, the mips\_core module has sub-module named controller and datapath. The sub-module “controller” is a sub-module to generate control signals to make pipeline work and the sub-module “datapath” is the datapath of pipeline which also has sub-modules named “regfile” and “alu”.
6. coregfile: new module added to implement co-register file

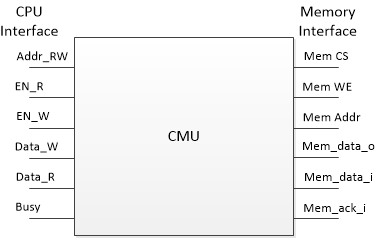
* Part ii: Cache design

1. Sim\_mips: the top module of Simulation
2. top: the module to implement the memory hierachy, which has also sub-module, such as cmu, inst and ram. And the module cache is included in cmu module.

* The main circuit graph is shown as following:
* Part i CPU access Mem in multiple cycles



* Part ii: Cache design



* Control signals

1. Part i CPU access Mem in multiple cycles

Here we define the following control signals(signal for new instructions mfc\mtc\eret is highlighted with red):

1. For PC source(which is the next PC):

PC\_NEXT: no jump and branch, the next PC is just PC+4

PC\_JUMP: used when the instruction is j or jal

PC\_JR: used when the instruction is jr

PC\_BEQ: used when the instruction is beq

PC\_BNE: used when the instruction is bne

PC\_ERET: used when the instruction is eret

1. For ALU A source(the input a of ALU):

A\_RS: come from rs of the instruction

A\_SA: come from sa of the instruction

A\_LINK: used when the instruction is jal

A\_BRANCH: used when the instruction is beq or bne

1. For ALU B source(the input b of ALU):

B\_RT: come from rt of the instruction

B\_IMM: come from imm of the instruction

B\_LINK: used when the instruction is jal

B\_BRANCH: used when the instruction is beq or bne

B\_CP: used when the instruction is mfc

1. For ALU operation: ALU\_ADD, ALU\_SUB, ALU\_SLT, ALU\_LUI, ALU\_AND, ALU\_OR, ALU\_SR
2. For WB(write back) address source

WB\_ADDR\_RD: to rd of the instruction

WB\_ADDR\_RT: to rt of the instruction

WB\_ADDR\_LINK: used when the instruction is jal

1. For WB data source

WB\_DATA\_ALU: from ALU result

WB\_ADDR\_MEM: from memory

1. Imm\_ext: Imm\_ext=1 when the immediate of the instruction should be sign-extension
2. Mem\_ren: memory read enable
3. Mem\_wen: memory write enable
4. WB\_wen:write back enable
5. Rs\_used: whether rs is used in this instruction
6. Rt\_used: whether rt is used in this instruction
7. Cp\_used: whether cp0 is used in this instruction as a source register
8. Wb\_wen\_en: whether enable to write back to cp0 regfile

The following is the table which describes the control signal of each instruction:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | add | sub | and | or | slt | srl |
| PC source | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT |
| ALU A source | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS |
| ALU B source | B\_RT | B\_RT | B\_RT | B\_RT | B\_RT | B\_RT |
| ALU operation | ALU\_ADD | ALU\_SUB | ALU\_AND | ALU\_OR | ALU\_SLT | ALU\_SR |
| WB address source | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD |
| WB data source | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 0 | 0 | 0 | 0 |
| Mem\_ren | 0 | 0 | 0 | 0 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 | 0 | 0 | 0 |
| WB\_wen | 1 | 1 | 1 | 1 | 1 | 1 |
| Rs\_used | 1 | 1 | 1 | 1 | 1 | 0 |
| Rt\_used | 1 | 1 | 1 | 1 | 1 | 1 |
| Cp\_used | 0 | 0 | 0 | 0 | 0 | 0 |
| Wb\_wen\_cp | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | jr | j | jal | beq | bne | addi |
| PC source | PC\_JR | PC\_J | PC\_J | PC\_BEQ | PC\_BNE | PC\_NEXT |
| ALU A source | / | / | A\_LINK | A\_BRANCH | A\_ BRANCH | A\_RS |
| ALU B source | / | / | A\_LINK | B\_ BRANCH | B\_ BRANCH | B\_IMM |
| ALU operation | / | / | ALU\_ADD | ALU\_SUB | ALU\_SUB | ALU\_ADD |
| WB address source | / | / | WB\_ADDR\_LINK | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RT |
| WB data source | / | / | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 0 | 1 | 1 | 1 |
| Mem\_ren | 0 | 0 | 0 | 0 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 | 0 | 0 | 0 |
| WB\_wen | 0 | 0 | 1 | 0 | 0 | 1 |
| Rs\_used | 1 | 0 | 0 | 1 | 1 | 1 |
| Rt\_used | 0 | 0 | 0 | 1 | 1 | 0 |
| Cp\_used | 0 | 0 | 0 | 0 | 0 | 0 |
| Wb\_wen\_cp | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | andi | ori | slti | Lw | sw | lui |
| PC source | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT |
| ALU A source | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS |
| ALU B source | B\_IMM | B\_IMM | B\_IMM | B\_IMM | B\_IMM | B\_IMM |
| ALU operation | ALU\_AND | ALU\_OR | ALU\_SLT | ALU\_ADD | ALU\_ADD | ALU\_LUI |
| WB address source | WB\_ADDR\_RT | WB\_ADDR\_RT | WB\_ADDR\_RT | WB\_ADDR\_RT | / | WB\_ADDR\_RT |
| WB data source | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_MEM | / | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 1 | 1 | 1 | 1 |
| Mem\_ren | 0 | 0 | 0 | 1 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 | 0 | 1 | 0 |
| WB\_wen | 1 | 1 | 1 | 1 | 0 | 1 |
| Rs\_used | 1 | 1 | 1 | 1 | 1 | 0 |
| Rt\_used | 0 | 0 | 0 | 0 | 1 | 1 |
| Cp\_used | 0 | 0 | 0 | 0 | 0 | 0 |
| Wb\_wen\_cp | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | mtc | mfc | eret |
| PC source | PC\_NEXT | PC\_NEXT | PC\_ERET |
| ALU A source | A\_LINK | A\_LINK | A\_RS |
| ALU B source | B\_RT | B\_CP | B\_RT |
| ALU operation | ALU\_ADD | ALU\_ADD | ALU\_ADD |
| WB address source | WB\_ADDR\_RD | WB\_ADDR\_RT | WB\_ADDR\_RD |
| WB data source | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 0 |
| Mem\_ren | 0 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 |
| WB\_wen | 0 | 1 | 0 |
| Rs\_used | 0 | 0 | 0 |
| Rt\_used | 1 | 0 | 0 |
| Cp\_used | 0 | 1 | 0 |
| Wb\_wen\_cp | 1 | 0 | 1 |

* Here are signals to control forward:

1. Forwards: If forwards = 0, forwards doesn’t work; if forwards = 1, the output of alu in EXE stage is forwarded to the data of rs; if forwards = 2, the output of alu in MEM stage is forwarded to the data of rs; if forwards = 3, the data got from memory is forwarded to the data of rs.
2. Forwardt: If forwardt = 0, forwardt doesn’t work; if forwardt = 1, the output of alu in EXE stage is forwarded to the data of rt; if forwardt = 2, the output of alu in MEM stage is forwarded to the data of rt; if forwardt = 3, the data got from memory is forwarded to the data of rt.
3. Reg\_stall: if reg\_stall = 1, the forward is not taken and a stall is implemented to deal with data hazard.

* Here is the signal to control interrupt:

1. Valid\_interrupter: if valid\_interrupter = 1, there is a valid interrupt and the instructions before WB can be cancelled. Only if interrupt is open and a interrupt signal is sent, the valid\_interrupter = 1

* Here are signals to control stall when accessing memory

1. Inst\_rom\_stall: stall when accessing Inst. ROM
2. Data\_ram\_stall: stall when accessing Data. RAM
3. Part ii: Cache design

* Key code segment

1. Part i CPU access Mem in multiple cycles
2. Inst\_rom\_module

|  |
| --- |
| module inst\_rom (  input wire clk,  input wire rclk, // real clk not reset clk  input wire rst,  input wire cs,  input wire [31:0] addr,  output reg [31:0] dout,  output wire stall  );    parameter  ADDR\_WIDTH = 6;    reg [31:0] data [0:(1<<ADDR\_WIDTH)-1];  reg [31:0] out;  reg [2:0] count;  reg old\_cs, old\_rst, old\_ack;  reg ack;  reg [31:0] old\_addr;  initial begin  $readmemh("inst\_mem.hex", data);  ack <= 0;  out <= 32'b0;  count <= 3'b0;  old\_cs <= 0;  old\_rst <= 0;  old\_addr <= 32'hFFFFFFFF;  end  assign stall = cs & ~ack;    always @(posedge rclk) begin  old\_addr <= addr;  if (addr != old\_addr) begin  ack <= 0;  count <= 0;  end  else if (cs) begin  if (count == 6) begin  ack <= 1;  end  else begin  ack <= 0;  count <= count + 1;  end  end  else begin  count <= 0;  ack <= 0;  end  end    always @(negedge clk) begin  out <= data[addr[ADDR\_WIDTH-1:0]];  end    always @(\*) begin  if (addr[31:ADDR\_WIDTH] != 0)  dout = 32'h0;  else  dout = out;  end  endmodule |

1. Data\_ram\_modul

|  |
| --- |
| module data\_ram (  input wire clk,  input wire rclk, //  input wire rst,  input wire cs, //  input wire we,  input wire [31:0] addr,  input wire [31:0] din,  output reg [31:0] dout,  output wire stall //  );    parameter  ADDR\_WIDTH = 5;    reg [31:0] data [0:(1<<ADDR\_WIDTH)-1];  reg [2:0] count;  reg [31:0] out;  reg ack;  initial begin  $readmemh("data\_mem.hex", data);  count <= 0;  out <= 0;  end  assign stall = (cs | we) & ~ack;  always @(posedge rclk) begin  if (cs | we) begin  if (count == 7) begin  ack <= 1;  count <= 0;  end  else begin  ack <= 0;  count <= count + 1;  end  end  else begin  count <= 0;  ack <= 0;  end  end    always @(negedge clk) begin  if (we && addr[31:ADDR\_WIDTH]==0)  data[addr[ADDR\_WIDTH-1:0]] <= din;  end    always @(negedge clk) begin  out <= data[addr[ADDR\_WIDTH-1:0]];  end    always @(\*) begin  if (addr[31:ADDR\_WIDTH] != 0)  dout = 32'h0;  else  dout = out;  end  endmodule |

1. Code for Control signal in controller module

|  |
| --- |
| always @(\*) begin  if\_rst = 0;  if\_en = 1;  id\_rst = 0;  id\_en = 1;  exe\_rst = 0;  exe\_en = 1;  mem\_rst = 0;  mem\_en = 1;  wb\_rst = 0;  wb\_en = 1;  status\_set = 0;  epc\_wen = 0;  pc\_ehb = 0;  if (valid\_interrupter) begin  if\_en = 0;  id\_en = 0;  exe\_en = 0;  mem\_en = 0;  wb\_en = 1;  if\_rst = 1;  id\_rst = 1;  exe\_rst = 1;  mem\_rst = 1;  epc\_wen = 1;  status\_set = 1;  pc\_ehb = 1;  end  else begin  if (rst) begin  if\_rst = 1;  id\_rst = 1;  exe\_rst = 1;  mem\_rst = 1;  wb\_rst = 1;  end  `ifdef DEBUG  // suspend and step execution  else if ((debug\_en) && ~(~debug\_step\_prev && debug\_step)) begin  if\_en = 0;  id\_en = 0;  exe\_en = 0;  mem\_en = 0;  wb\_en = 0;  end  `endif  // this stall indicate that ID is waiting for previous instruction, should insert NOPs between ID and EXE  else if (reg\_stall) begin  if\_en = 0;  id\_en = 0;  exe\_rst = 1;  end  // this stall indicate that a jump/branch instruction is running, so that 3 NOP should be inserted between IF and ID  else if (branch\_stall == 1) begin  if\_en = 0;  if\_rst = 1;  id\_rst = 1;  end  else if (branch\_stall == 2) begin  id\_rst = 1;  end  else if (predict\_wrong) begin  id\_rst = 1;  exe\_rst = 1;  mem\_rst = 1;  end  else if (data\_ram\_stall) begin  if\_en = 0;  id\_en = 0;  exe\_en = 0;  mem\_en = 0;  wb\_rst = 1;  end  else if (inst\_rom\_stall) begin  if\_en = 0;  id\_rst = 1;  end  end  end |

1. Part ii: Cache design
2. 实验过程和数据记录及结果分析
3. Input data sets and output results (each instruction has a inst\_rom\_stall)

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction addr | Assembly code | Machine code | Result |
| 0 | Lui r1,0x0 | 3c010000 | R1=0 |
| 4 | Addi r1,r1,0x4c | 2021004c | R1=0x4c |
| 8 | Mtc r1,r3 | 40811800 | CP0[r3] = 0x3c |
| C | Add r2,r0,r0 | 00001020 | R2=0 |
| 10 | Add r3,r0,r0 | 00001820 | R3=0 |
| 14 | Lw r6,0(r0) | 8c060000 | R6=mem[0]=0; data\_ram\_stall |
| 18 | Lw r6,4(r0) | 8c060004 | R6=mem[4]=4; data\_ram\_stall |
| 1c | Sw r6,0(r0) | Ac060000 | Mem[0]=4; data\_ram\_stall |
| 20 | Lw r7,0(r0) | 8c070000 | R7=4; data\_ram\_stall |
| 24 | Mfc0 r4,r9 | 40044800 | R4 = CPR9 |
| 28 | Add r5,r0,r0 | 00002820 | R5 = 0 |
| 2c | Addi r2,r2,1 | 20420001 | Loop |
| 30 | Slti r1,r2,10000 | 28412710 |  |
| 34 | Bne r1,r2,-3 | 1420fffd |  |
| 38 | Mfc0 r5,r9 | 40054800 | R5 = CPR9 |
| 3c | Sub r5,r5,r4 | 00a42822 |  |
| 40 | Addi r2,r2,1 | 20420001 | Loop |
| 44 | J 0x40 | 08000010 |  |
| 48 | Nop | 00000000 | Never executed |
| 4c | Addi r3,r3,1 | 20630001 | R3= the number of interrupter |
| 50 | eret | 4200001c | PC =EPC |

1. Screen shots and pictures of the running result on Spartan3E

* The first line of the screen display some important information, including register values and instruction data or address of instruction in IF\ID\EX\MEM\WB stages; the second line of the screen display the instruction address in IF\ID\EX\MEM\WB stages. The following image shows the stage before interrupter where the second instruction “addi r1,r1,32” is in WB stage and r1 = 0x20,which is right



* The following images show the stage before and when and “mtc r1,r3”(addr= 8) is in WB stage. We can see before it’s in WB stage, CP0[R3] = 0; and when it’s in WB stage, CP0[R3] = 0x20, which is right and indicates the “mtc” works correctly.





* The following images show the stage before and after a interrupter arises. We can see after interrupter, the instruction before mem stage(addr = 0x18) is cancelled and the pc in if stage become 0x20, which is the entry of interrupt handle. That indicated the interrupter is correct handled.





* The following images show the stage before and after a interrupter handle exits. We can see when “eret”(addr = 0x28) is in wb stage, the pc in IF stage is set to epc(0x14), which is right and indicates “eret” works correctly.





1. Steps to do this experiment:

* Review and extend the key code in lab3
* Add three new instructions “mtc”\””mfc”\”eret”
* Implement forwarding on three new instructions
* Design cp0 regfile
* Extend and modify signals to implement interrupter
* Use new display module to display data to satisfy the new requirement.
* Design the testing code of MIPS.
* Compile and download the bit file to the board.
* Execute the testing code and debug.

1. 讨论与心得

Lab4 is a truly challenge to me because it requires me to master the knowledge of interrupter and coprocessor of MIPS. Fortunately, I have implemented interrupter in previous courses (but without pipeline). To implement interrupter with pipelined CPU, I spend a lot time reviewing the previous code and course ware about interrupter and even more time debugging. Thanks to lab4, although it time-consuming, I have a better master on interrupter.