Computer Organization and Design (2023)

The Processor

Jiang zhong

zhongjiang@cqu.edu.cn

QQ: 376917902

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Clock cycle time
    - Determined by CPU hardware
- We will examine MIPS CPU implementations
  - A simplified version
- Simple instruction subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`
- We will introduce additional materials about CPU

Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

- AND-gate
  - Y = A & B

- Multiplexer
  - Y = S ? I1 : I0

- Adder
  - Y = A + B

- Arithmetic/Logic Unit
  - Y = F(A, B)

Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1 (or changes from 1 to 0)

Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

寄存器堆的设计

32个32位寄存器堆的内部结构

封装后

两个读端口和一个写端口的寄存器堆

N1, N2--两个读端口的地址输入端，对应输出端为Q1和 Q2

ND--为写端口的地址输入

DI--为32位数据输入端

Verilog 寄存器堆的实现

```
`timescale 1ns / 1ps
module RegisterFile(
     input CLK,            // 时钟
     input RegWre,       // 写使能信号，为1时，在时钟上升沿写入
     input [4:0] rs,              // rs寄存器地址输入端口
     input [4:0] rt,              // rt寄存器地址输入端口
     input [4:0] WriteReg,      // 将数据写入的寄存器端口，其地址
来源rt或rd字段
     input [31:0]   WriteData,     // 写入寄存器的数据输入端口
     output [31:0] ReadData1,    // rs寄存器数据输出端口
     output [31:0]  ReadData2     // rt寄存器数据输出端口
    );
```

Verilog 寄存器堆的实现

```
reg [31:0] register[0:31];  // 新建32个寄存器，用于操作
                            // 初始时，将32个寄存器全部赋值为0
     integer i;
     initial
     begin
        for(i = 0; i < 32; i = i + 1)  register[i] <= 0;
     end
     // 读寄存器
    assign ReadData1 = register[rs];
    assign ReadData2 = register[rt];

    // 写寄存器
    always@(negedge CLK)    //在下降沿是改写寄存器
     begin
        // 如果寄存器不为0，并且RegWre为真，写入数据
        if (RegWre && WriteReg != 0)  register[WriteReg] = WriteData;
     end

endmodule
```

Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period
    - Must keep “critical path” as short as possible

Setup and hold time constraints – see Appendix C

主观题 10分

设置

1.请回顾计算机是如何执行程序的?

2.请讨论一条机器指令执行时完成了哪些具体的操作?

正常使用主观题需2.0以上版本雨课堂

作答

实现存储程序的计算机系统需要解决的问题

1. 在何处存放指令、如何读取指令?
2. 通过哪些部件来执行指令?
3. 如何进行指令的寻址?
4. 需要哪些部件? 如何来协调这些部件?

Instruction Execution: Main Steps

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Write result to register file (optional)
  - Access data memory for load/store (optional)
  - PC ← target address or PC + 4

CPU (the main frame)Overview

Multiplexers

- Can't just join wires together

Reference: Appendix C

单选题 1分

设置

如果某个部件存在两个以上的数据输入端，需要选择什么部件来解决输入冲突

A 寄存器

B 多路选择器

C 或非门

D 与非门电路

提交

Control

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

The Basic MIPS Instruction Types

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store	35 or 43	rs	rt	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch

Verilog PC 单元的实现

```
module PC(
    input CLK,                          // 时钟
    input Reset,                        // 重置信号
    input PCWre,                        // PC是否更改，如果为0，PC不更改
    input [31:0] newAddress,            // 新指令地址
    output reg[31:0] currentAddress     // 当前指令地址
    );
    initial begin
        currentAddress <= 0;  // 非阻塞赋值
    end
    always@(posedge CLK or posedge Reset)
     begin
        if (Reset == 1)  currentAddress <= 0;  // 如果重置，赋值为0
        else
         begin
            if (PCWre)  currentAddress <= newAddress;
            else  currentAddress <= currentAddress;
         end
     end
endmodule
```

Verilog 指令存储器的实现

```
module InstructionMemory(
    input InsMemRW,          // 读写控制信号，1为写，0位读
    input [31:0] IAddr,      // 指令地址输入入口
    output [5:0] op,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
    output [15:0] immediate   // 指令代码分时段输出
   );
   reg[7:0] mem[0:63];  // 新建一个字节数组老模拟指令储存
   initial
    begin
      $readmemb("test/test.txt", mem);  //读取测试文档中的指令
    end
   // 从地址取值，然后输出
   assign op = mem[IAddr][7:2];
   assign rs[4:3] = mem[IAddr][1:0];
   assign rs[2:0] = mem[IAddr + 1][7:5];
   assign rt = mem[IAddr + 1][4:0];
   assign rd = mem[IAddr + 2][7:3];
   assign immediate[15:8] = mem[IAddr + 2];
   assign immediate[7:0] = mem[IAddr + 3];
endmodule
```

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

a. Registers

b. ALU

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store	35 or 43	rs	rt	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

a. Data memory unit

b. Sign extension unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions

Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - No datapath resource can be used more than once per instruction
    - Any element needed more than once must be duplicated
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

Full Datapath

ALU Control

- ALU used for
  - Load/Store: F = ?
  - Branch: F = ?
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Generated output

The Main Control Unit

- Control signals derived from instruction

Add	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

Beq	4	rs	rt	address
	31:26	25:21	20:16	15:0

Datapath With Control

R-Type Instruction

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

Load Instruction

Load/ Store	35 or 43	rs	rt	address
	31:26	25:21	20:16	15:0

Branch-on-Equal Instruction

Branch	4	rs	rt	address
	31:26	25:21	20:16	15:0

Implementing Jumps

Jump	2	address
	31:26	25:0

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added

op	jump target

J format

(Verilog)单周期处理器的实现

(Verilog)单周期处理器的实现

顶层连接模块的部分代码

```
module SingleCPU(
    input CLK,
    input Reset,
    output [5:0] op,
     output [4:0] rs,
     output [4:0] rt,
     output [4:0] rd,
     output [15:0] immediate,
    output [31:0] ReadData1,
    output [31:0] ReadData2,
     output [31:0] WriteData,
     output [31:0] DataOut,
    output [31:0] currentAddress,
    output [31:0] result
    );
```

```
wire [2:0] ALUOp;
wire [31:0] B, newAddress;
wire [31:0] currentAddress_4,
extendImmediate, currentAddress_immediate;
wire [4:0] WriteReg;
wire zero, PCSrc, PCWre, ALUSrcB,
ALUM2Reg, RegWre, InsMemRW, DataMemRW,
ExtSel, RegOut;
```

参考链接:https://blog.csdn.net/MyCodecOdecoDecodE/article/details/72670113

(Verilog)单周期处理器的实现

```
ControlUnit cu(op, zero, PCSrc, PCWre, ALUSrcB, ALUM2Reg, RegWre,
InsMemRW, DataMemRW, ExtSel, RegOut, ALUOp);
PC pc(CLK, Reset, PCWre, newAddress, currentAddress);
InstructionMemory im(InsMemRW, currentAddress, op, rs, rt, rd, immediate);
RegisterFile rf(CLK, RegWre, rs, rt, WriteReg, WriteData, ReadData1,
ReadData2);
ALU alu(ALUOp, ReadData1, B, zero, result);
DataMemory dm(DataMemRW, result, ReadData2, DataOut);
   assign currentAddress_4 = currentAddress + 4;
   assign currentAddress_immediate = currentAddress_4 + (extendImmediate <<
2);
   Multiplexer5 m5(RegOut, rd, rt, WriteReg);

   Multiplexer32 m321(ALUSrcB, extendImmediate, ReadData2, B);
   Multiplexer32 m322(ALUM2Reg, DataOut, result, WriteData);
   Multiplexer32 m323(PCSrc, currentAddress_immediate, currentAddress_4,
newAddress);
endmodule
```

主观题 10分

设置

对于指令 AND Rd, Rs, Rt

1、数据通路将用到哪些功能单元?

2、控制单元将产生哪些控制信号？其取值是什么?

作答

简化的MIPS指令的多周期处理机设计方案

单周期处理机的缺陷---性能差

单周期：处理机执行每条指令都需要一个时钟周期。而确定时钟周期的长度时，要以最复杂的指令执行时需要多长时间为基准

- 对于装入（load）指令，周期时间必须足够长,指令的执行过程包括几个步骤：

1、从指令存储器取指令

2、从寄存器堆读出数据

3、用ALU计算地址

4、从数据存储器取出数据

5、最后把数据写入寄存器堆

主观题 10分

设置

关键路径分析：假设每个逻辑器件的延时（ns）如下：

I-MEM	ADD	MUX	ALU	Regs	D-MEM	SIG-Extend	Shift-Left2
200	70	20	90	90	250	15	10

请分析： ADD、LW 、SW 和 BNE 指令的执行的时间

作答

处理机执行指令步骤：

1、取指令

2、分析指令

3、取操作数

4、执行指令

5、保存结果

RR型/RI型

1、从指令存储器取指令

2、从寄存器堆读出数据

3、 ALU执行算术逻辑操作

~~4、从数据存储器取出数据~~

5、最后把数据写入寄存器堆

Store指令

1、从指令存储器取指令

2、从寄存器堆读出数据

3、 ALU计算地址

4、把数据写入数据存储器

~~5、最后把数据写入寄存器堆~~

根据统计各类指令的使用频度大约为：

ALU指令占	50%
load指令占	20%
Store指令占	10%
Branch指令占	20%

多周期处理机实现概述

单周期处理器的问题根源:

- 所有指令必须以最慢的指令使用的时间为准

解决方案:

- 提高时钟频率，即缩短时钟周期
- 使用不同数量的时钟周期完成不同类型的指令
- 将指令处理分为若干个周期
- 每个周期执行一步（而不是整个指令）
  - 周期时间： 执行最长步所需的时间
  - 使所有的步骤尽量具有相同的长度

多周期处理器的优点:

- 周期时间比较短
- 不同的指令可以使用不同的周期数来完成
  - 装入指令 LW 需要5个周期
  - 跳转指令仅仅需要3个周期
- 允许每条指令多次使用同一个功能部件

指令将完成的操作：

ALU 指令：RR型和RI型

RR型　rd←(rs1) op (rs2)　或　R(rd) ← R(rs1) op R(rs2)

RI型　rd←(rs1) op imme 或　R(rd) ← R(rs1) op imme

存储器访问指令：　RI型

load:　rd←((rs1) + imme)　或　R(rd) ← mem[(rs1) + imme]

store　rd→((rs1) + imme) 或　R(rd) → mem[(rs1) + imme]

转移/跳转：BR型

条件转移　beq：　if z=1 then pc=pc+disp，else pc=pc+1

bne:　if z=0 then pc=pc+disp,　else pc=pc+1

无条件转移：**Branch : pc=pc + disp**

假定多周期机器涉及的具体指令如下

ALU指令

- add rd, rs1, rs2　　addi rd, rs1, imme
- sub rd, rs1, rs2　　subi rd, rs1, imme
- and rd, rs1, rs2　　andi rd, rs1, imme
- or rd, rs1, rs2　　ori rd, rs1, imme

存储器访问

- load rd, rs1, imme
- stroe rd, rs1, imme

条件转移:　beq　disp　　bne　disp

无条件跳转: Branch　disp

所有的指令都是32位长　　具有如下三种格式：

RR型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1		rs2

RI型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1	Immediate	

BR型

31 26	25 21	20 16	15 5	4 0
opcode	Displacement			

opcode: 指令的操作码

rs1, rs2, rd: 源1和源2以及目的寄存器描述符

displacement: 偏移量

immediate: 立即数数值

指令系统和指令格式						
31 26	25 21	20 16	15 5	4 0	指令助记符	意 义
00 0000	rd	rs1		rs2	and rd, rs1, rs2	rd ←(rs1) and (rs2)
00 0001	rd	rs1	imme		andi rd, rs1, imme	rd←(rs1) and imme
00 0010	rd	rs1		rs2	or rd, rs1, rs2	rd ←(rs1) or (rs2)
00 0011	rd	rs1	imme		ori rd, rs1, imme	rd ←(rs1) or imme
00 0100	rd	rs1		rs2	add rd, rs1, rs2	rd ←(rs1) add (rs2)
00 0101	rd	rs1	imme		addi rd, rs1, imme	rd ←(rs1) add imme
00 0110	rd	rs1		rs2	sub rd, rs1, rs2	rd ←(rs1) sub (rs2)
00 0111	rd	rs1	imme		subi rd, rs1, imme	rd ←(rs1) sub imme
00 1000	rd	rs1	imme		load rd, rs1, imme	rd ←((rs1) + imme)
00 1001	rd	rs1	imme		store rd, rs1, imme	(rd)→((rs1) + imme)
00 1010	disp				bne disp	If z=0, pc←(pc)+disp
00 1011	disp				bnq disp	If z=1, pc←(pc)+disp
00 1100	disp				branch disp	pc←(pc) + disp

多周期处理机的数据路径总体图

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

数据路径有如下改动

1、存储器模块只用一个，指令和数据都使用该存储器

2、不使用专门的加法器，而是借用ALU

由此而产生如下影响：

1、取指令操作和访存指令执行时地址来源不同，增加多路选择器

2、访存指令执行时，正在执行的指令将丢失。增加指令寄存器IR

3、ALU前增加二选一和四选一多路选择器。解决PC+1和转移地址计算

多周期处理机执行指令的5个周期

1、取指令及PC加1周期

实现以下操作：IR=Memory[PC]
PC=PC+1

该周期用到的控制信号：

SelLdst=0 （选择PC的内容做存储器地址）
SelAluA=1 （选择PC的内容做ALU A端的输入）
SelAluB=10 （选择1做ALU B端的输入）
AluOP=10 （ALU 做加运算）
WriteIR=1 （周期结束时写指令寄存器IR）
WritePC=1 （周期结束时写程序计数器PC）

SelLdst=0
SelLoad
Registers
1
P
C
DO
IR
1
rd
A1
AD
A
Q1
Q
rs
CS
B
Z
Zero
WritePC=1
WriteZero
WriteMem
WriteIR=1
WriteReg
1
3
AluOP=
SelAluB=1
0
imme
Sign Extend
Imme
Sign Extend
opcod
e
Control
...
WritePC
...
WriteZero
Zero

2、指令译码，读寄存器及转移周期

实现以下操作：A=Register[rs1]

B=Register[rs2]

IM=SignExtend[imme]

If(BranchTaken)

PC=PC+SignExtend[disp]

该周期用到的控制信号：

SelSt=0(非store指令)

SelSt=1(store指令)

如果该指令是转移指令：

SelAluA=1 (选择PC的内容做ALU A端的输入)

SelAluB=11 (选择扩展的偏移量做ALU B端的输入)

AluOP=10 (ALU 做加运算)

WritePC=Branch+bne$\overline{\text{Zero}}$+beqZero(若转移条件满足时写PC)

在这个周期结束时，转移指令结束

指令译码，读寄存器(RR类型)

RR型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1		rs2

指令译码， store指令读寄存器

RI型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1	Immediate	

如果该指令是转移指令，并且条件满足时，把转移地址写入PC

RI型

31 26	25 21	20 16	15 5	4 0
opcode	rd	rs1	Immediate	

3、ALU执行或者存储器地址计算周期

实现以下操作：

RR型指令：AluOutput=A OP B

RI型指令：AluOutput=A OP IM

存储器访问指令load/stroe：AluOutpot=A + IM

该周期用到的控制信号：

SelAluA=0 （选择A做ALU A端的输入）

RR型指令 SelAluB=00 （选择B做ALU B端的输入）

RI型指令 SelAluB=01 （选择IM做ALU B端的输入）

AluOP=XX （ALU 完成指令指定的运算）

ALU指令（RR型RI型）执行

Load 指令计算存储器地址

Store指令计算存储器地址

4、ALU指令结束周期或者存储器访问周期

1）ALU指令结束周期实现以下操作：

Register[rd]=AluOutput
Zero=AluZero

用到的控制信号：
SelLoad=0
WriteZero=1
WriteReg=1

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

2）如果是load指令实现以下操作：

MemOutput=Memory[AluOutpot]

用到的控制信号：

SelLdst=1

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

3）如果是store指令数据写入存储器，指令结束

实现以下操作： Memory[AluOutpot]=B

用到的控制信号:SelLdst=1 WriteMem=1

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

5、写回周期—只有load指令才能达到这个周期

实现以下操作： Register[rd]=MemOutput

用到的控制信号：SelLoad=1 WriteReg=1

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

ALU指令RR型完整数据通路

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

ALU指令RI型完整数据通路

	31 26	25 21	20 16	15 5	4 0
RR型	opcode	rd	rs1		rs2
RI型	opcode	rd	rs1	Immediate	
BR型	opcode	Displacement			

Load指令完整数据通路

Store指令完整数据通路

转移指令完整数据通路

SelLdst
1
0
存储器
Add
DO
DI
WE
CS
P
C
WritePC
IR
WriteIR
SelLoad
0
1
寄存器堆
DI
A1
AD
A2
Q1
Q2
rs1
rd
rs2
1
0
SelSt
WriteReg
WriteMem
SelAluA
1
0
A
ALU
B
0
1
3
SelAluB
AluOP
Z
Zero
WriteZero
imme
立即数符号扩展
disp
偏移量符号扩展
opcode
控制部件
Zero
WritePC
...
WriteZero
RR型
31 26 | 25 21 | 20 16 | 15 5 | 4 0
opcode | rd | rs1 | | rs2
RI型
31 26 | 25 21 | 20 16 | 15 5 | 4 0
opcode | rd | rs1 | Immediate
BR型
31 26 | 25 21 | 20 16 | 15 5 | 4 0
opcode | Displacement

主观题 10分

设置

请简要叙述两个寄存器相加的ADD指令在以下数据通路上的执行过程，并说明每个多路选择器的控制信号

正常使用主观题需

作答

控制器设计

有限状态机设计基本方法

若时序电路有N个状态，则至少需要n=log2N个触发器

设计一个有限状态机的步骤一般是：

1、画出状态转移图。

2、写出状态转移表。

3、写出下一状态的布尔表达式，并尽可能化简。

4、写出输出信号的真值表。

5、写出输出信号的布尔表达式，并尽可能化简。

6、根据下一状态和输出信号的布尔表达式，画出逻辑图。

有限状态机电路模型

下一状态
CLK
当前状态寄存器
当前状态
下一状态产生函数
输入信号
（opcode和Zero）
输出信号产生函数
输出
（数据路径控制信号）

1、画出状态转移图。

IF
ID
EXE
MEM
WB
IF
S0
ID
S1
RR
RI
LOAD
STORE
BRANCH
ALURR
S2
ALURWB
S6
ALURI
S3
ALUIWB
S7
LOAD
S4
LDMEM
S8
LDWB
S10
STORE
S5
STMEM
S9

2、写出状态转移表。

当前状态		输入	下一状态	
状态	Q3Q2Q1Q0	条件	状态	D3D2D1D0
S0	0 0 0 0	X	S1	0 0 0 1
S1	0 0 0 1	BR	S0	0 0 0 0
	0 0 0 1	RR	S2	0 0 1 0
	0 0 0 1	RI	S3	0 0 1 1
	0 0 0 1	LOAD	S4	0 1 0 0
	0 0 0 1	STROE	S5	0 1 0 1
S2	0 0 1 0	X	S6	0 1 1 0
S3	0 0 1 1	X	S7	0 1 1 1
S4	0 1 0 0	X	S8	1 0 0 0
S5	0 1 0 1	X	S9	1 0 0 1
S6	0 1 1 0	X	S0	0 0 0 0
S7	0 1 1 1	X	S0	0 0 0 0
S8	1 0 0 0	X	S10	1 0 1 0
S9	1 0 0 1	X	S0	0 0 0 0
S10	1 0 1 0	X	S0	0 0 0 0

3、写出下一状态的布尔表达式，并尽可能化简。

D0=S0+S1 RI+S1 store + S3+S5

D1=S1 RI+S1 RR+ S2+ S5+S8

D2=S1 load+S1 store+ S2+ S3

D3=S4 + S5 + S8

当前状态		输入	下一状态	
状态	Q3Q2Q1Q0	条件	状态	D3D2D1D0
S0	0 0 0 0	X	S1	0 0 0 1
S1	0 0 0 1	BR	S0	0 0 0 0
	0 0 0 1	RR	S2	0 0 1 0
	0 0 0 1	RI	S3	0 0 1 1
	0 0 0 1	LOAD	S4	0 1 0 0
	0 0 0 1	STROE	S5	0 1 0 1
S2	0 0 1 0	X	S6	0 1 1 0
S3	0 0 1 1	X	S7	0 1 1 1
S4	0 1 0 0	X	S8	1 0 0 0
S5	0 1 0 1	X	S9	1 0 0 1
S6	0 1 1 0	X	S0	0 0 0 0
S7	0 1 1 1	X	S0	0 0 0 0
S8	1 0 0 0	X	S10	1 0 1 0
S9	1 0 0 1	X	S0	0 0 0 0
S10	1 0 1 0	X	S0	0 0 0 0

4、写出输出信号的真值表。

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
WritePC	1	BT	0	0	0	0	0	0	0	0	0
SelLdst	0	X	X	X	1	1	X	X	1	1	1
WriteMem	0	0	0	0	0	0	0	0	0	1	0
WriteIR	1	0	0	0	0	0	0	0	0	0	0
SelLoad	X	X	X	X	1	X	0	0	1	X	1
SelSt	X	0	0	X	X	1	0	X	X	1	X
WriteReg	0	0	0	0	0	0	1	1	0	0	1
SelAluA	1	1	0	0	0	0	0	0	0	0	0
SelAluB1	1	1	0	0	0	0	0	0	0	0	0
SelAluB0	0	1	0	1	1	1	0	1	1	1	1
WriteZero	0	0	0	0	0	0	1	1	0	0	0
AluOP1	1	1	OP1	OP1	1	1	OP1	OP1	1	1	1
AluOP0	0	0	OP0	OP0	0	0	OP0	OP0	0	0	0

$BT = branch + bne\ \overline{Zreo} + beq\ Zreo$

表中红色“1”信号与前面讨论提前了一个周期

指令	OP1	OP0
and/andi	0	0
or/ori	0	1
add/addi	1	0
sub/subi	1	1
OP0=or+ori+sub+subi	OP1=add+addi+sub+subi	

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
WritePC	1	BT	0	0	0	0	0	0	0	0	0
SelLdst	0	X	X	X	1	1	X	X	1	1	1
WriteMem	0	0	0	0	0	0	0	0	0	1	0
WriteIR	1	0	0	0	0	0	0	0	0	0	0
SelLoad	X	X	X	X	1	X	0	0	1	X	1
SelSt	X	0	0	X	X	1	0	X	X	1	X
WriteReg	0	0	0	0	0	0	1	1	0	0	1
SelAluA	1	1	0	0	0	0	0	0	0	0	0
SelAluB1	1	1	0	0	0	0	0	0	0	0	0
SelAluB0	0	1	0	1	1	1	0	1	1	1	1
WriteZero	0	0	0	0	0	0	1	1	0	0	0
AluOP1	1	1	OP1	OP1	1	1	OP1	OP1	1	1	1
AluOP0	0	0	OP0	OP0	0	0	OP0	OP0	0	0	0

SelLdst
1
0
存储器
Add
DO
DI
WE
CS
P
C
WritePC
IR
WriteIR
WriteMem
SelLoad
0
1
rs1
rd
1
rs2
0
SelSt
寄存器堆
DI
A1
AD
A2
Q1
Q2
WriteReg
SelAluA
1
0
A
ALU
B
0
1
3
SelAluB
AluOP
Z
Zero
WriteZero
imme
立即数符号扩展
disp
偏移量符号扩展
opcode
控制部件
Zero
...
WritePC
...
WriteZero

5、写出输出信号的布尔表达式，并尽可能化简。

WritePC = S0+S1 BT
SelLdst = S4+S5+S8+S9+S10
WriteMem = S9
WriteIR = S0
SelLoad = S4+S8+S9
SelSt = S5+S9
WriteReg = S6+S7+S10
SelAluA = S0+S1
SelAluB 1 = S0+S1
SelAluB0 = S1+S3+S4+S5+S7
WriteZero = S6+S7
AluOP1 = S0+S1+S2 OP1+S3 OP1+S4+S5+S6OP1 +S7 OP1+S8+S9+S10
AluOP0 = S2 OP1+S3 OP1+ S6 OP1 +S7 OP1

6、根据下一状态和输出信号的布尔表达式，画出逻辑图。

主观题 10分

设置

请绘制WritePC和SelLoad控制信号的逻辑电路图

WritePC = S0+S1 BT

SelLoad = S4+S8+S9

当前状态	
状态	Q3Q2Q1Q0
S0	0 0 0 0
S1	0 0 0 1
	0 0 0 1
	0 0 0 1
	0 0 0 1
	0 0 0 1
S2	0 0 1 0
S3	0 0 1 1
S4	0 1 0 0
S5	0 1 0 1
S6	0 1 1 0
S7	0 1 1 1
S8	1 0 0 0
S9	1 0 0 1
S10	1 0 1 0

正常使用主观题需2.0以上版本雨课堂

作答

多周期处理机的控制部件

数据路径总体图

从多周期到流水线处理器（ DataPath of Pipeline ）

专用总线系统和单总线系统

根据需要建立CPU中各个部件之间的数据通路和控制线路，因此其扩展性较差，添加新的功能或者部件时需要重新设计数据通路。

为了简化处理机的设计，提高系统的扩展性，提出了单总线结构，即所有的部件都通过相同的数据总线、地址总线和控制总线连接，添加的设备只要符合总线的接口标准就可以挂接到系统中。

由于采用单总线的结构，只能有一个部件向总线写数据，可以有多个部件接收数据。

The best way to predict the future is to invent it

Smalltalk发明人阿伦凯（Alan Kay 1940）简介

1961年，因为出面维护犹太移民，被迫从西佛吉尼亚**Bathany**音乐学院离开，以教授吉他为生，生活潦倒。直到参加的一项计算机编程潜在能力测试改变了命运。

1968年夏天，阿伦遇到麻省理工人工智能看到了最原始的手写识别系统。阿伦提出“把手写识别应用到计算机上，我能创造出一种超媒体——就像现在的报纸，但它是电子化的。”现代笔记本电脑原型的“**Dynabook**”。 **1969**年，阿伦获得犹他州立大学计算机科学博士学位，开始思考如何使庞大的计算机变得更小。这种**PC**对儿童来说会很方便，他们可以用它来代替纸。由于这种**PC**需要一种新的语言，阿伦便设计出了后来名震业界的**Smalltalk**语言。**Smalltalk**语言再现了阿伦的“分子**PC**思想”：程序好比一个个生物分子，通过信息相互连接。**Smalltalk**被业界公认为“面向对象编程系列语言”的代表作品。

指令将完成的操作：

ALU 指令：RR型和RI型

RR型　rd←(rs1) op (rs2)　或　R(rd) ← R(rs1) op R(rs2)

RI型　rd←(rs1) op imme 或　R(rd) ← R(rs1) op imme

存储器访问指令：　RI型

load:　rd←((rs1) + imme)　或　R(rd) ← mem[(rs1) + imme]

store　rd→((rs1) + imme) 或　R(rd) → mem[(rs1) + imme]

转移/跳转：BR型

条件转移　beq：　if z=1 then pc=pc+disp，else pc=pc+1

bne:　if z=0 then pc=pc+disp,　else pc=pc+1

无条件转移：Branch : pc=pc + disp

单总线CPU结构

基本构成：

控制器，运算器，
寄存器，数据通路

寄存器的类型：

指令寄存器(IR)
程序计数器(PC)
数据寄存器(MDR)
地址寄存器(MAR)
状态寄存器(SR)
通用寄存器（Ri）
用户不可见暂存器（Z Y）

数据通路：

单总线结构

单 总 线 C P U 结 构

R out
Q
C P
D
R in

控制信号
指令译码 / 控制器
IR
PC
地址总线
存储器
MAR
MDR
数据总线
R0
R_{n-1}
Y
A
B
ALU
SR
Z

单 总 线 C P U 结 构

单总线CPU结构指令的执行过程

一、ALU指令的执行过程

ADD R0, R1, R2

(1)PC→MAR

(2) PC+1→PC

a) PC由计数器构成

b) PC由寄存器构成，+1操作可由ALU完成。

单 总 线 C P U 结 构

一、 ALU指令的执行过程

ADD R0, R1, R2

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

单 总 线 C P U 结 构

ADD R0, R1, R2

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) R1→Y

单 总 线 C P U 结 构

ADD R0, R1, R2

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) R1→Y

(6) R2 + Y→Z

单 总 线 C P U 结 构

ADD R0, R1, R2

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) R1→Y

(6) R2 + Y→Z

(7) Z→R0

ADD指令执行结束

单 总 线 C P U 结 构

二、访存指令的执行过程

装人操作：LOAD R2,mem

(1) PC→MAR

(2) PC+1→PC

单 总 线 C P U 结 构

装人操作：LOAD R2,mem

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

所有指令的开始都相同。
即取指令阶段

单 总 线 C P U 结 构

装入操作：LOAD R2,mem

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) IR(地址段)→MAR，读存储器

单总线CPU结构

装入操作：LOAD R2,mem

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) IR(地址段)→MAR，读存储器

(6) DBUS→MDR

(7) MDR→R2

LOAD指令执行结束

单 总 线 C P U 结 构

二、访存指令的执行过程

写存储器操作：

STORE R1,mem

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) IR(地址段)→MAR

单总线CPU结构

写存储器操作：

STORE R1,mem

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) IR(地址段)→MAR

(6) R1→MDR，写存储器

STORE指令执行结束

单 总 线 C P U 结 构

三、转移控制指令的执行过程

无条件转移：

Branch disp

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) PC→Y

单 总 线 C P U 结 构

无条件转移：

Branch disp

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) PC→Y

(6) Y + IR(地址段)→Z

单 总 线 C P U 结 构

无条件转移：

Branch disp

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) PC→Y

(6) Y + IR(地址段)→Z

(7) Z→PC

BRANCH指令执行结束

单 总 线 C P U 结 构

条件转移：

BNE disp

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) if(!Z) PC→Y;

else goto END

单 总 线 C P U 结 构

条件转移：

BNE disp

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) if(!Z) PC→Y; else goto END

(6) Y + IR(地址段)→Z

单总线CPU结构

条件转移：

BNE disp

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) if(!Z) PC→Y; else goto END

(6) Y + IR(地址段)→Z

(7) Z→PC

BNE指令执行结束

单 总 线 C P U 结 构

注意、前面的执行是假设整条指令是一次取出。

如果不能一次取出，情况又会是怎样？

例如：mov r1,add_mem ; R1←(add_mem)

指令格式：

5位	3位	8位
opcode	Rd	mem

假定处理机字长8位，存储器的读写长度为8为，那么16位指令长度将分两次完成

单总线CPU结构

mov r1, add_mem

(1) PC→MAR

(2) PC+1→PC

(3) DBUS→MDR

(4) MDR→IR

(5) PC→MAR ;(PC)=n+1

(6) PC+1→PC ;(PC)=n+2,

指向下一条指令

(7) DBUS→MDR; (n+1)=add_mem

(8) MDR →MAR;以add_mem为地址，

访问存储器

(9) DBUS→MDR;取出add_mem单元的内容

(10) MDR →R1; (add_mem) →r1

指令流程图

一个操作步骤代表在一个机器周期中可完成的操作

执行步骤所需的控制信号

分支	操作	时钟周期	控制信号
取指令	PC→MAR, PC+1, read	T1	PCout, MARin, PC+1, Read
	MDR→IR	T2	MDRout, IRin
ADD 操作	R1→Y	T3	R1out, Yin
	R2+Y→Z	T4	R2out, Zin, ADD
	Z→R3	T5	Zout, R3in
LOAD 操作	IR→MAR, read	T3	IRout, MARin, Read
	MDR→R1	T4	MDRout, R1in
STORE 操作	IR→MAR	T3	IRout, MARin
	R1→MDR, write	T4	R1out, MDRin, Write
JMP 操作	PC→Y	T3	PCout, Yin
	IR+Y→Z	T4	IRout, ADD, Zin
	Z→PC	T5	Zout, PCin

微操作控制形成电路的逻辑表达形式

C = T1*(INS1 + INS2 + ...) + T2*(INS1 + INS2 + ...) + ...

ADD指令每个时钟周期内的控制信号为:

T1: PCout, MARin, PC+1, Read ;PC→MAR, PC+1, read

T2: MDRout, IRin ;MDR→IR

T3: R1out, Yin ;R1→Y

T4: R2out, Zin, Add ;R2+Y→Z

T5: Zout, R3in ;Z→R3

JMP指令中各时钟周期的控制信号为:

T1: PCout, MARin, PC+1, Read ;PC→MAR, PC+1, read

T2: MDRout, IRin ;MDR→IR

T3: PCout, Yin ;PC→Y

T4: IRout, Add, Zin ;IR+Y→Z

T5: Zout, PCin ;Z→PC

控制器的逻辑表达式

PC+1 = T1

PCin = T5*JMP

PCout = T1 + T3*JMP

Yin = T3*(ADD + JMP)

Add = T4*(ADD + JMP)

Zin = T4*(ADD + JMP)

Zout = T5*(ADD + JMP)

END = T5*(ADD + JMP)

...

组合逻辑（时序）控制器的特点

优点:速度快, 可用于速度要求较高的机器中.

缺点:(1) 缺乏规整性:将几百个微操作的执行逻辑组合在一起, 构成的微操作产生部件, 是计算机中最复杂、最不规整的逻辑部件. 不适合于指令复杂的机器.

(2) 缺乏灵活性:各微命令的实现是用硬连的逻辑电路完成, 改动不易, 设计困难.

PLA控制器

PLA控制器的设计步骤与组合逻辑控制器相同,只是实现方法不同,它采用PLA阵列(Programmed Logic Array). 从设计思想来看是组合逻辑控制器,从实现方法来看,是存储逻辑控制器.

特点:可使杂乱无章的组合逻辑规整化、微型化，而且可以利用PLA的可编程特性，用存储逻辑部分地取代组合逻辑，增加了一定的灵活性。

微程序控制器

微程序控制的基本思想，就是仿照通常的解题程序的方法，把操作控制信号编成所谓的“微指令”，存放到一个只读存储器里．当机器运行时，一条又一条地读出这些微指令，从而产生全机所需要的各种操作控制信号，使相应部件执行所规定的操作。

组合逻辑电路一经实现，不能变动其逻辑关系，必要时，必须改变其连线或重新设计。

微程序控制方法：把指令执行所需要的所有控制信号存放在控制存储器中，需要时从这个存储器中读取，存储逻辑可以修改ROM存放的数据,从而修改逻辑功能,速度略慢,有一个寻址和读数据的过程.

微程序控制的特点：灵活性好，速度慢

微程序控制方式的历史简介

由莫里斯·威尔克斯（**1967** 图灵奖）。1946年10月，以冯·诺伊曼的EDVAC为蓝本设计建造了EDSAC。它使用了水银延迟线作存储器，穿孔纸带为输入设备和电传打字机为输出设备。EDSAC是第一台诺依漫机器结构的电子计算机。 在设计与制造EDSAC和EDSAC2的过程中，威尔克斯创造和发明了许多新的技术概念。诸如“变址”、“宏指令”、微程序、子例程及子例程库、高速缓冲存储器（Cache）等等，这些都对现代计算机的体系结构和程序设计技术产生了深远的影响。

转到 4-2 介绍