计算机组成与实现

单周期CPU

高小鹏
北京航空航天大学计算机学院

目录

系统的观点

- 系统是相互作用的诸元素的综合体。系统的特点：
  - ①一个系统包含至少2个以上的要素
  - ②要素之间是存在连接关系的
  - ③要素的功能以及要素间的连接关系决定了系统的整体功能
- 如果把CPU视为系统，则寄存器堆、ALU、控制器等就是要素
- 宏观上，一个CPU的功能正确性取决于：
  - 各个功能部件的正确性
  - 功能部件间建立了正确的连接关系

问题
CPU被分解为寄存器堆、ALU等功能部件，遵循的是什么方法或原则？

※要素也可以被称为子系统
※要素、子系统、功能部件是含义相同的表述方式

机制与策略

- 机制：按照功能最大无关性及组合最大灵活性为基本原则构造的功能部件的集合
- 策略：根据高层需求选取并控制功能部件子集执行相应的功能

	CPU	人	路口交通控制
高层需求	执行指令	走、跑、跳	各种通行需求
机制	程序计数器 寄存器堆 算数逻辑单元 指令存储器 数据存储器	四肢 躯干	各路口的红绿灯 各灯的定时器
策略	控制器	大脑	交通控制箱

※要素也可以被称为子系统

机制与策略

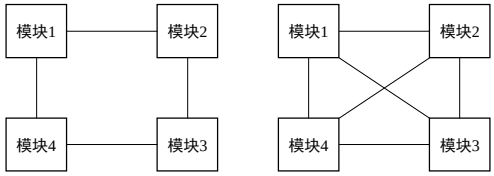
- 机制与策略分离的设计方法学
  - 降低系统复杂度。
    - 多数系统包含大量高度相似的功能需求，将其由相对独立的功能部件实现
    - 通过设计和修改策略部分就能很容易的实现一个个的具体需求了
  - 提高系统灵活性
    - 假设功能部件是较为充分，则可以仅通过增加新的策略来实现某个新需求
  - 提高系统扩展性
    - 当新需求无法通过现有机制与策略无法满足时，可以将其转换为相应的机制与策略，然后添加到已有的机制与策略集合中
    - 该设计模式对已有设计影响(如修改、重构、甚至推倒重建)可能是最小的
  - 提高系统稳定性
    - 在机制与策略分离设计模式下，通常每个功能部件的规模都非常有限
    - 规模有限通常意味着复杂度低、出错概率低
    - 如此，最终集成起来的系统就会具有足够的可靠性

高内聚与低耦合

- 内聚：是指某个功能部件的内部各功能的相关性
- 耦合：是指功能部件间连接关系的强度
- 高内聚：
  - 强调单一性，即功能部件或者其功能单一或者其多个功能高度相似
  - 遵循的是单一职责原则(Single Responsibility Principle)
- 低耦合：
  - 强调功能部件的接口应该尽可能少，彼此间的连接关系尽可能的简单
  - 遵循的是最少知识原则(Least Knowledge Principle)
    - 最少知识原则也称作迪米特法则(Law of Demeter)
- 高内聚低耦合是更具可操作性的系统设计原则
  - 高内聚的设计结构几乎必然导致低耦合
  - 低耦合必然意味着高内聚

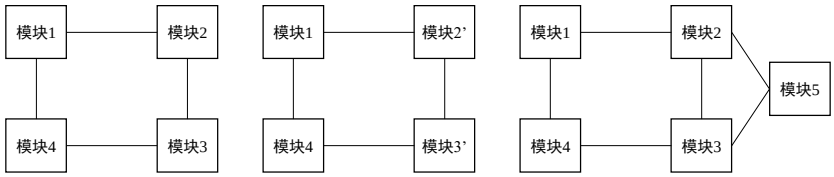
高内聚与低耦合

- 示例1：方案A比方案B具有更低的耦合度，因此优于方案B



A. 内聚及耦合特性好　　B. 内聚及耦合特性差

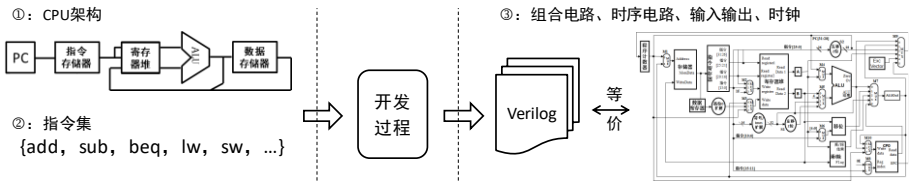
- 示例2：对于新需求，方案B仅仅修改了已有模块，而方案C则引入了新模块，因此方案B优于方案C



A. 已有设计　　B. 好设计(新增需求)　　C. 坏设计(新增需求)

形式建模综合方法概述

- CPU开发是什么？
  - 以某种CPU架构（如单周期、多周期或流水线）为模型，面向给定的指令集，设计并构造出以Verilog形式表达的CPU具体实现



设计输入　　具体实现

- CPU架构模型是什么？
  - CPU架构模型：以寄存器堆、ALU等功能部件为基础，描述了各功能部件的基本接口特性以及相互间的基本连接关系
  - CPU架构模型关注两点
    - 功能部件的外特性，即功能部件的某个具体功能与控制的对应关系
    - 功能部件间传递信息的基本依赖关系

形式建模综合方法概述

- CPU开发基本过程
  - 1）构造功能部件。构造一组必要的数据通路功能部件。对于每个功能部件，都需要定义其功能及接口信号，并用HDL（Hardware Description Language，硬件描述语言）描述其内部具体的电路行为或结构。
  - 2）构造指令级别的数据通路与控制信号取值。根据指令操作语义构造指令级别的数据通路与相关功能部件的控制信号取值。
  - 3）综合数据通路。当所有指令级别的数据通路构造完毕后，将其综合为完整数据通路。
  - 4）综合控制器。当全部指令对应的控制信号取值均建立后，合并所有的控制信号取值形成控制信号矩阵，并生成相应的控制信号表达式。
  - 5）生成工程项目。将功能第2步和第3步的结果翻译为HDL语言，并在顶层工程文件中组装数据通路与控制器。

形式建模综合方法概述

- 形式建模综合方法的要点是什么？
  - 1）开发过程是基于模型的
    - 这里的模型就是所谓的CPU架构，如单周期、多周期、流水线
  - 2）开发过程被显式的分为设计和实现两个环节
    - 设计：是指建模每条指令的数据通路和控制信号
    - 实现：是指将设计结果用Verilog等语言表达
  - 3）基于“系统-子系统”视角的建模层次
    - CPU被视为系统，各功能部件被视为子系统
  - 4）指令级别独立建模
    - 独立分析每条指令操作语义，推演其执行过程中的数据流信息和控制流信息
    - 独立构造每条指令对应的数据通路以及相关功能部件的控制信号取值
  - 5）一次性的系统级综合
    - 将所有分离的数据通路及控制信号取值高效合成为完整的数据通路和控制器

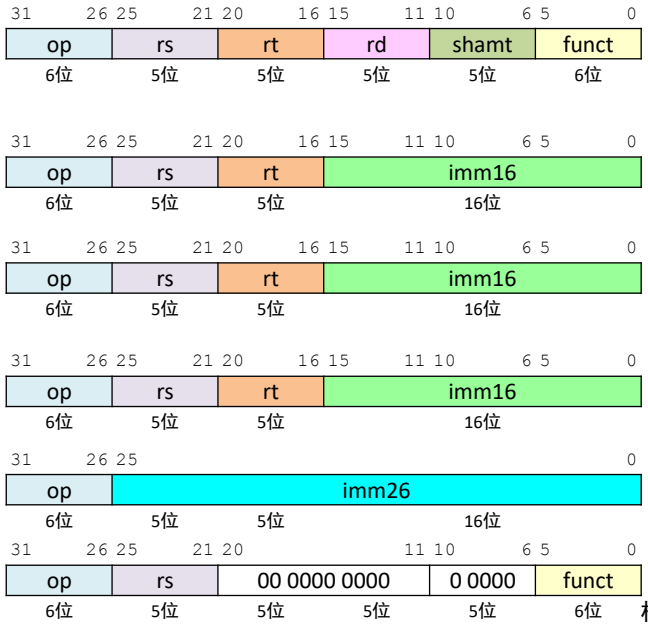
形式建模综合方法概述

- 指令级别独立建模是什么？
  - 1）根据指令的RTL，梳理和总结出数据通路的设计需求
  - 2）选择恰当的数据通路功能部件
  - 3）建立功能部件间的正确连接关系
  - 4）根据指令的RTL，选择功能部件应执行的功能，反推控制信号取值

目录

MIPS-C0指令集

- 加减法指令
  - `addu rd,rs,rt`
  - `subu rd,rs,rt`
- 立即数指令
  - `ori rt,rs,imm16`
- 存储访问指令
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`
- 分支指令
  - `beq rs,rt,imm16`
- 函数调用指令
  - `jal target`
- 函数返回指令
  - `jr rs`



MIPS-C0指令集

- 从功能角度，MIPS-C0可以构造程序设计的绝大多数功能
  - lw、sw：存储指令的典型代表
  - addu、subu：运算类指令的典型代表
  - beq：分支类指令的典型代表
  - jal、jr：支持函数
- 从结构角度，MIPS-C0包含了所有3种指令格式
  - R型：addu、subu、jr
  - I型：lw、sw、beq
  - J型：jal

注意
Jr是R型指令，而不是J型指令

MIPS-C0的RTL描述

- 任何指令执行的第一步都是取指令

```
R-format:  {op, rs, rt, rd, shamt, funct} ← MEM[ PC ]
I-format:  {op, rs, rt, imm16} ← MEM[ PC ]
```

- MIPS-C0的RTL描述

指令	RTL描述
addu	R[rd]←R[rs]+R[rt]; PC←PC+4
subu	R[rd]←R[rs]-R[rt]; PC←PC+4
ori	R[rt]←R[rs]|zero_ext(imm16); PC←PC+4
lw	R[rt]←MEM[R[rs]+sign_ext(imm16)]; PC←PC+4
sw	MEM[R[rs]+sign_ext(imm16)]←R[rt]; PC←PC+4
beq	if (R[rs] == R[rt]) then PC←PC+4 + (sign_ext(imm16) || 00) else PC←PC+4
jal	PC←PC31..28||instr_index||02; R[31]←PC + 4
jr	PC←R[rs]

指令集功能需求

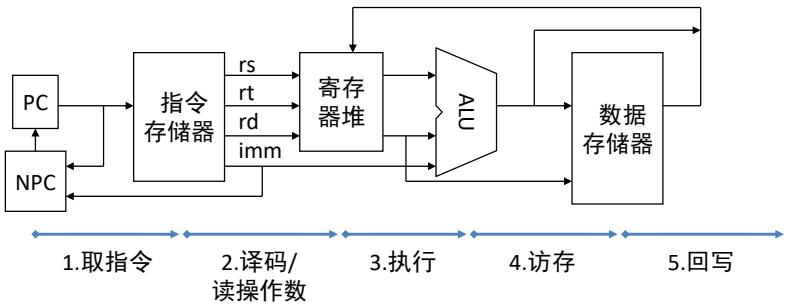
- 存储器(MEM)
  - 指令存储器 & 数据存储器(分离的存储器模拟了cache结构)
  - 指令存储器只有读取数据（取指令）
  - 数据存储器既有读取数据又有写入数据
- 寄存器堆(32个32位寄存器)
  - 能同时读出rs和rt两个寄存器
  - 可以写数据至rt或rd寄存器
- PC
- 下一条指令地址的计算单元（NPC，NextPC）
- 扩展立即数：符号/零扩展
- 执行Add/Sub/OR等运算的计算单元
  - 如何执行beq的比较操作？

数据通路的主要功能部件

- 指令存储器（Instruction Memory，IM）
  - 读数据：输入地址，输出数据（即指令）
- 数据存储器（Data Memory，DM）
  - 读数据：输入地址，输出数据
  - 写数据：输入地址&数据
- 寄存器堆（Register File，RF）
  - 读寄存器：输入2个读寄存器编号，输出两个32位值
  - 写寄存器：输入写寄存器编号&32位值
- PC
- NPC（NextPC，NPC）
  - 计算下一个PC值：输入PC，输入Imm，输出计算结果
- ALU（Arithmetic and Logic Unit， ALU）
  - 加/减/或：输入2个操作数，输出计算结果

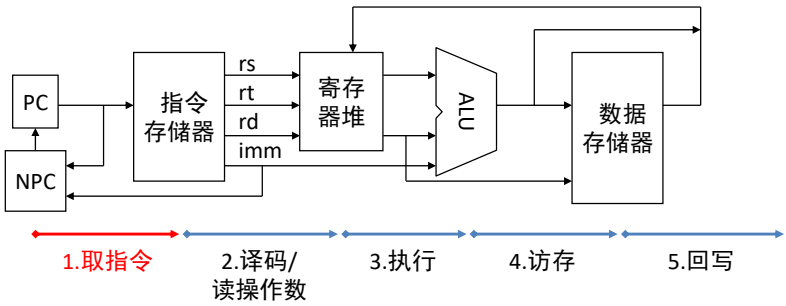
数据通路的抽象模型

- 指令执行的主要步骤
  - 取指令、译码/读操作数、执行、访存、回写



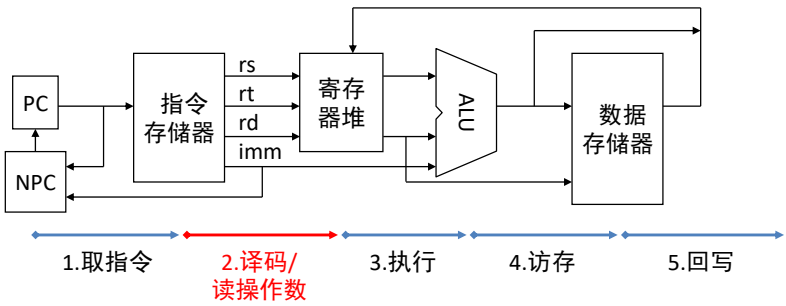
注意
模型刻画了指令执行过程中的主要信息流的基本流动路径。
模型不是完整的设计。
对于分析更细微、更精确的设计细节的依赖与连接关系，模型具有重要指导意义。

数据通路分解1/5



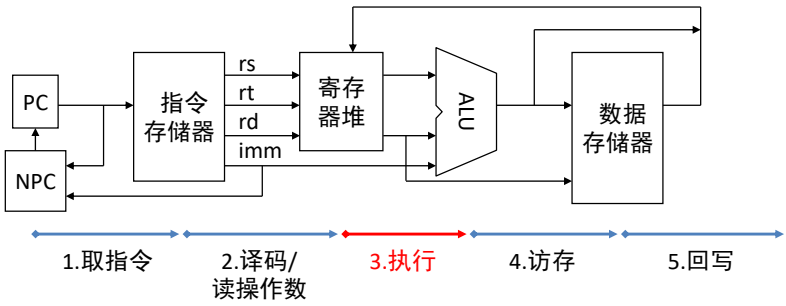
- 第1步：取指令
  - 操作1：PC驱动IM输出指令
  - 操作2：PC驱动NPC计算下一个PC值

数据通路分解2/5



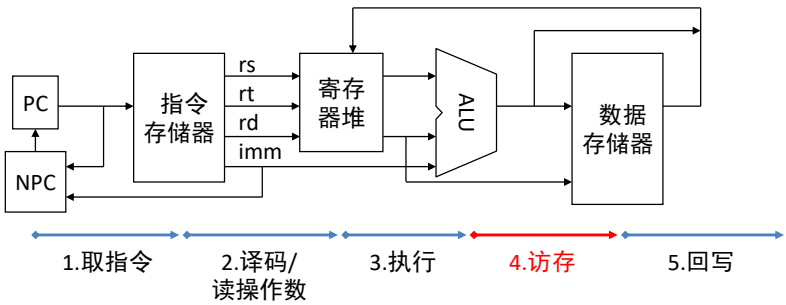
- 第2步：译码/读操作数
  - 操作1：IM驱动控制器（图中未画）分析指令的opcode和funct域
  - 操作2：IM驱动RF读出2个寄存器值

数据通路分解3/5



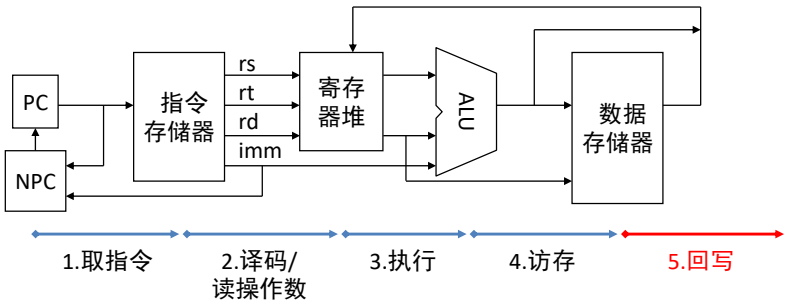
- 第3步：执行
  - 操作：RF输出的寄存器值驱动ALU完成相应的计算
    - 算数运算(+，-，*，/)，移位，逻辑(&,|)，比较(slt,==)
    - 同时还承担计算lw和sw的地址

数据通路分解4/5



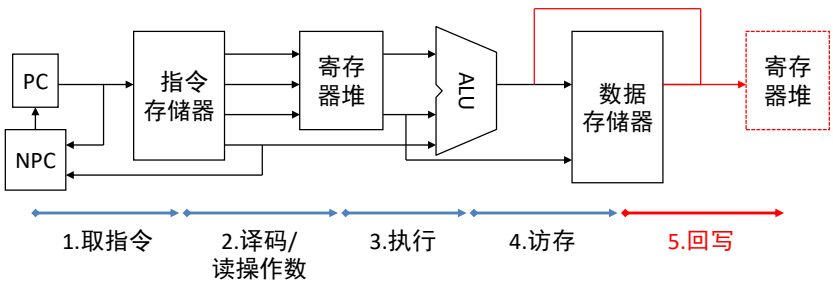
- 第4步：访存
  - 操作1：对于lw，输入地址后则输出数据
  - 操作2：对于sw，需要同时输入地址&要写入的数据
  - 只有lw和sw指令在该环节有实际操作，其他指令不涉及该环节

数据通路分解5/5



- 第5步：回写
  - 操作：写ALU计算结果或数据存储器读出的数据至寄存器堆
  - beq、jal、jr、sw不涉及该环节

寄存器堆在数据通路中的独特地位



- 寄存器堆是一个特殊部件，对于很多指令来说，它具有双重功效
  - 它在第3阶段服务于读取操作数
  - 它在第5阶段服务于回写结果
- 为逻辑清晰起见，可以“再部署”一个寄存器堆
  - “消除”了回馈线路

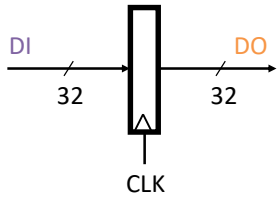
为什么是5个阶段？

- 是否可以有不同的阶段数？
  - 可以有不同的阶段数
    - 早期CPU的阶段数甚至只有2-3个
    - 现代CPU的阶段数甚至可能达到20-30个
- 为什么MIPS采用5阶段？
  - 虽然有些指令用不到5阶段，但lw却必须用到5阶段
    - 事实上，有些早期MIPS也不是5阶段

目录

PC

- PC非常简单，但也非常重要
- PC本质上就是一个32位的寄存器
  - 因为每条指令占用4B，所以PC的b1与b0恒为0
  - 32位寄存器可以优化为30位寄存器
- PC需要在系统复位后有一个确定的初值，即第1条指令的地址
  - 这里先假设为0000_0000h
- 根据上述分析，可以总结出PC的功能及输入输出信号



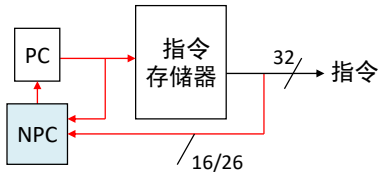
功能描述	Reset有效，寄存器置初值0x0000_0000。	
信号名	方向	描述
Clk	I	MIPS-C处理器时钟
Reset	I	复位信号
DI[31:0]	I	32位输入
DO[31:0]	O	32位输出

目录

NPC

- 任何指令的第一步除了取指令外，还要更新PC
- 更新PC，首先是NPC要计算出下一条指令的地址

指令	NPC执行的计算	
顺序执行指令 addu/subu/ori/lw/sw	PC+4	
分支或跳转指令 beq/jal/jr	其他某个计算结果	
	beq	计算与PC和imm16相关
	jal	计算与PC和imm26相关
	jr	计算与rs寄存器相关



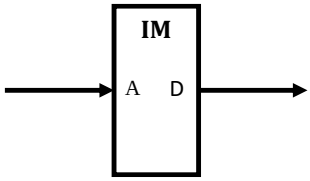
功能描述	计算下一条指令的地址	
信号名	**方向**	**描述**
PC[31:0]	I	32位输入
NPC[31:0]	O	32位输出

目前先只考虑顺序执行指令，因此NPC只需要计算PC+4，故NPC只需要输入PC值。

目录

指令存储器

- 存储器可以理解为一个数组
  - 存储器的地址就是数组的下标
  - 给出存储器地址，存储器就输出对应单元的数据
- 执行读出操作时，存储器行为可视为组合逻辑
  - 地址A有效一段时间后，数据RD就输出正确的值
    - 这个所谓的“一段时间”被称为访问时间



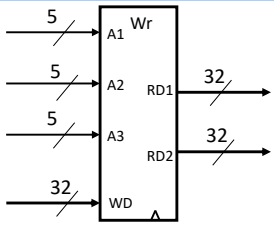
有关不同存储器类型的详细内容，在存储层次中介绍

access time~访问时间

目录

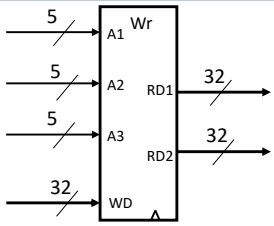
寄存器堆

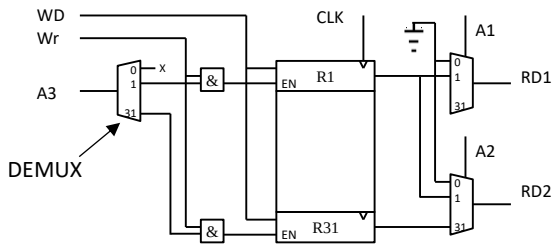
- 寄存器堆包含32个寄存器
  - RD1和RD2：读出的2个寄存器值
  - WD：写回的值
- 寄存器编号
  - A1和A2：读取的第1个和第2个寄存器的编号
  - A3：写入的寄存器编号
- 写使能
  - 并非所有的指令都要写寄存器，因此寄存器堆需要有写使能信号Wr
  - 在时钟上升沿时，如果Wr=1，则WD3才能被写入A3寄存器中
- 与指令存储类似，寄存器堆执行读出操作时可视为组合逻辑
  - A1/A2有效一段时间后，RD1/RD2就输出正确的值



寄存器堆的设计考虑

- 内部需要多少个32位寄存器？
  - 31个。0号寄存器采用接地的特殊设计
- 读出数据功能：32位31选1 MUX
  - 2个读出端口是独立工作
  - 无需彼此等待
- 写入数据功能：关键是写使能
  - 每个寄存器需要一个写使能
- DEMUX：分离器/解码器
  - N位编码产生2^N个输出
  - 有且仅有1个输出有效





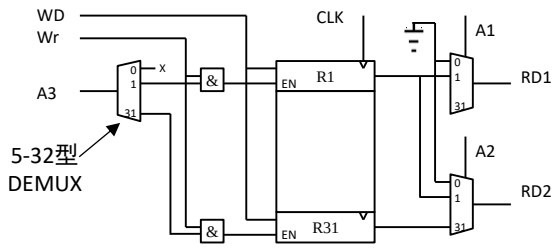
demultiplexer~DEMUX

寄存器堆的设计考虑

- 示例：2-4型DEMUX
  - A为输入2位， Y3至Y0均为1位输出

A	Y3	Y2	Y1	Y0
00	0	0	0	1
01	0	0	1	0
10	0	1	0	0
11	1	0	0	0

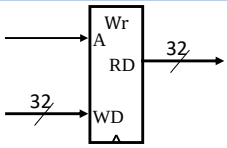
$$Y3 = \overline{A1} \cdot \overline{A0}$$
$$Y2 = \overline{A1} \cdot A0$$
$$Y1 = A1 \cdot \overline{A0}$$
$$Y0 = A1 \cdot A0$$



目录

数据存储器

- 与指令存储器不同，数据存储器要支持写入
  - WD：写入的数据
- 写使能
  - 与寄存器堆类似，数据存储器需要有写使能信号Wr
- 存储器访问
  - 读：Wr=0，A单元数据从RD输出
    - 读出操作时可视为组合逻辑，即A有效一段时间后，RD就输出正确的值
  - 写：在时钟上升沿时，如果Wr=1，则WD被写入A单元中



目录

ALU需求分析

- 计算需求：加、减、或、相等
- 相等
  - 方法1：设计独立的比较电路，例如利用XOR运算
  - 方法2：执行减法运算，然后再判断结果是否全0

采用方法2，减法电路被重用

指令	RTL描述
addu	R[rd]←R[rs]+R[rt]; PC←PC+4
subu	R[rd]←R[rs]−R[rt]; PC←PC+4
ori	R[rt]←R[rs] | zero_ext(imm16); PC←PC+4
lw	R[rt]←MEM[R[rs]+sign_ext(imm16)]; PC←PC+4
sw	MEM[R[rs]+sign_ext(imm16)]←R[rt]; PC←PC+4
beq	if (R[rs] == R[rt]) then PC←PC+4 + (sign_ext(imm16) || 00) else PC←PC+4

1位LSB加法器

- 加法是计算机中最基础、最高频的运算
- 示例：4位加法运算

$$\begin{array}{cccc|c} a_3 & a_2 & a_1 & a_0 & \\ + \; b_3 & b_2 & b_1 & b_0 & \\ \hline s_3 & s_2 & s_1 & s_0 & \end{array}$$

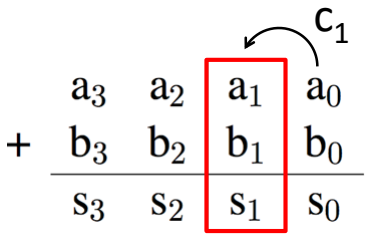
进位输出

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$s_0 = a_0 \text{ XOR } b_0$$
$$c_1 = a_0 \text{ AND } b_0$$

LSB(least signiant bit)~最低有效位

加法器：1位加法器

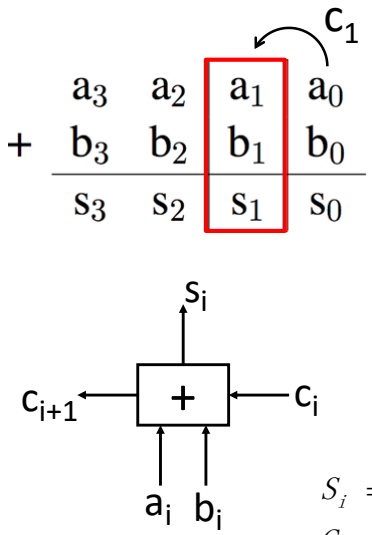


a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S_i = \bar{A}_i\bar{B}_iC_i + \bar{A}_iB_i\bar{C}_i + A_i\bar{B}_i\bar{C}_i + A_iB_iC_i$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

加法器：1位加法器



a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

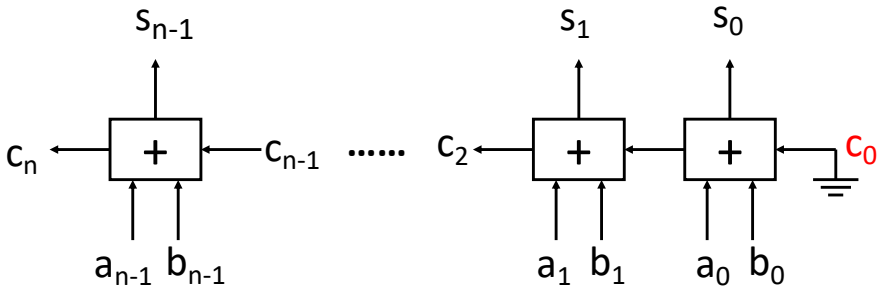
$$S_i = \bar{A}_i\bar{B}_iC_i + \bar{A}_iB_i\bar{C}_i + A_i\bar{B}_i\bar{C}_i + A_iB_iC_i$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

用N个1位加法器构造N位加法器

- 把C_i与C_{i-1}连接，形成进位链
  - 串行加法器；行波进位加法器

Q
如何处理C_0？



减法运算

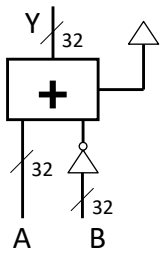
- 计算Y=A-B，可以将其等价转换为Y=A+(-B)
- 利用二进制补码的负数转换规则：

$$-\mathrm{B} = \bar{B} + 1$$

- 因此

$$\mathrm{Y} = \mathrm{A} - \mathrm{B} = \mathrm{A} + \bar{B} + 1$$

- 可以利用加法器构造减法运算
  - B输入取反
  - Cin为1

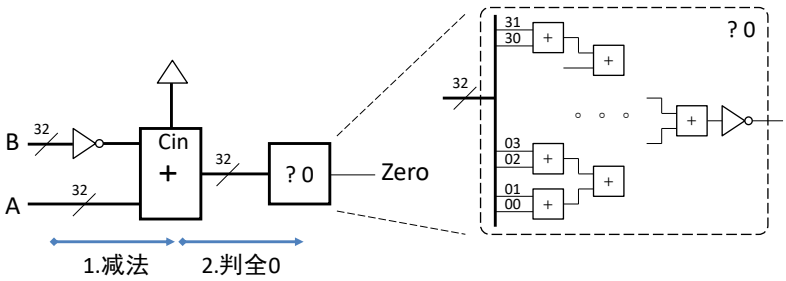


相等

- 采用重用减法的思路
  - 先执行减法，然后再判断结果是否全0
- Zero
  - 0：A≠B
  - 1：A=B

问题1
判0电路需要多少OR门？

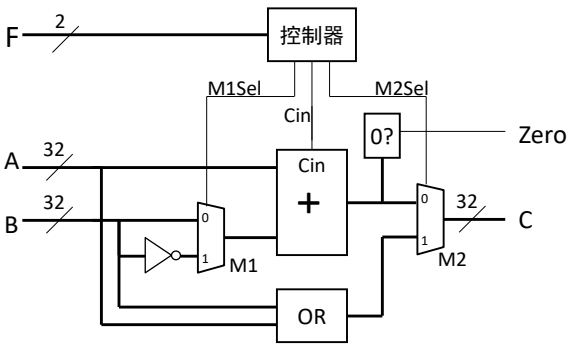
问题2
如果采用XOR实现相等，需要多少AND门、OR门、NOT门？



集成

- M1：0通道对应加法，1通道对应减法
  - 加法：A+B
  - 减法：$A - B = A + \bar{B} + 1$
- M2：0通道对应加减法的结果，1通道对应OR的结果
- Cin：如果执行加法则为0，如果执行减法则为1

F	功能
00	加
01	减
10	或

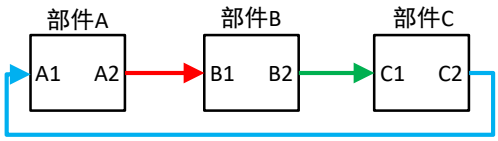


根据控制器的功能表，结合ALU内部结构设计，可以建立内部各控制信号的真正表，进而推导出表达式，从而构造出控制器的门电路结构

目录

建模的基本表示方式

- 数据通路本质上是一个连接关系的集合
  - 集合的元素：相关部件的输入输出信号之间的连接关系
- 示例：3个部件的连接关系



部件	输入	输出
A	A1	A2
B	B1	B2
C	C1	C2

  - 连接关系集合：{<C.C2,A.A1>，<A.A2,B.B1>，<B.B2,C.C1>}
    - 连接关系<X.m,Y.n>：部件X的输出信号m，连接至部件Y的输入信号n
  - 连接关系表格：该方式表示连接关系集合，在布局上更易于对应图形方式

A	B	C	------ 部件
A1	**B1**	**C1**	------ 部件的输入信号
C.C2	**A.A2**	**B.B2**	------ 部件及其输出信号

建模的基本推理过程

- 基本事实1：当CPU基本模型确定后，意味着指令的执行路径总体是确定的
  - 执行路径也可以认为是信息流路径
  - 执行路径大体包括：取指、译码/读操作数、计算、访存、回写
- 基本事实2：构成指令执行路径的各分段之间存在着明确的依赖关系
  - 示例1：指令执行的前提是必须先读取指令；而读取指令就必然是用PC驱动IM的地址线，然后IM输出的就是当前要执行的指令
  - 示例2：读取DM的前提是必须先计算出地址；而地址是通过加法运算完成的，因此ALU执行加法就是非常合理的

建模的推理方法：倒推法

- 由于指令执行存在强逻辑依赖关系，因此建模指令基本的数据通路时既可采用正向推理方法，也可采用反向推理法，即倒推法
  - 反向推理结束后，将推理顺序倒置即得到正向的推理顺序

环节	步骤	语义	连接关系	功能部件	控制信号取值
-1	A	每步的具体操作	<X.m,Y.n>	F	信号名:取值
-2	B				
-3	C				

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	C				
2	B				
3	A	每步的具体操作	<X.m,Y.n>	F	信号名:取值

数据通路表

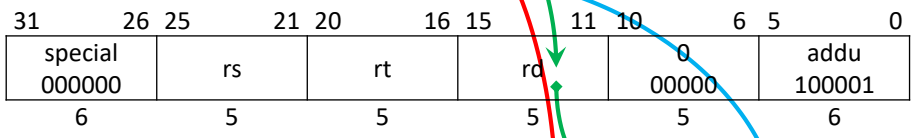
- 为了直观展示及易于综合，每条指令推理结束后，用数据通路表记录该指令的所有连接关系
- 示例：包含PC、NPC、IM、RF、ALU、DM的数据通路表
  - 第1行：为功能部件名
  - 第2行：该功能部件的各个输入信号
  - 第3行：与该输入信号连接的某个功能部件的输出信号

指令	NPC	PC	IM	RF				ALU		DM	
	PC	DI	A	A1	A2	A3	WD	A	B	A	WD

Addu建模

R[rd]←R[rs]+R[rt]

- 最后环节：必定是将加法运算的计算结果写入寄存器堆
- 写入寄存器堆涉及3个要素，即写入编号、写入数据、写使能
  - 基本事实：ALU具有加法功能
  - 推理1：写入编号来自指令的rd域
  - 推理2：写入数据来自ALU
  - 推理3：写使能需要有效，即为1



环节	步骤	语义	连接关系	功能部件	控制信号取值
-1	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1

Addu建模

R[rd]←R[rs]+R[rt]

- 将连接关系用表格表示
  - 控制信号取值暂时不表示，后续会在控制器设计中使用

环节	步骤	语义	连接关系	功能部件	控制信号取值
-1	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1

指令	NPC	PC	IM	RF				ALU		DM	
				A1	A2	A3	WD	A	B	A	WD
addu						IM.D[15:11]	ALU.C				

Addu建模

R[rd]←R[rs]+R[rt]

- -2环节：既然写入数据来自ALU，因此需要由ALU完成加法计算
- ALU运算涉及3个要素，即第1个操作数、第2个操作数、ALU操作
  - 推理1：第1个操作数是从RF的RD1输出
  - 推理2：第2个操作数是从RF的RD2输出
  - 推理3：ALU控制信号取值为加法编码

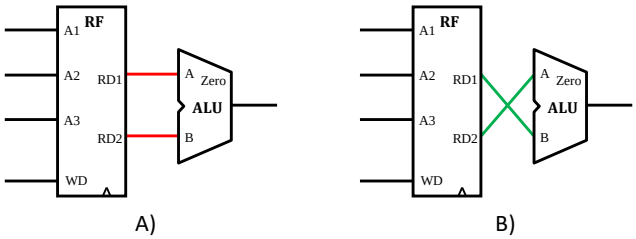
这是具有可读性的宏定义

环节	步骤	语义	连接关系	功能部件	控制信号取值
-1	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1
-2	执行	ALU执行加法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	ALUOp:ADD

指令	NPC	PC	IM	RF				ALU		DM	
	PC	NPC	A	A1	A2	A3	WD	A	B	A	WD
addu				IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C	RF.RD1	RF.RD2		

RF与ALU的连接关系

- RF有2个32位输出，RD1和RD2；ALU有2个输入，A和B
- 问题：所有指令的RTL都不定义两者的连接关系，那么ALU的A/B连接RF的RD1/RD2还是RD2/RD1？



- 解答
  - 1）对于任一指令而言，两种连接均可以
  - 2）推荐采用固定连接方案（习惯上采用方案A）

※※※
如果有些指令采用方案A，有些指令采用方案B，这不影响功能正确性。但是，今后综合完整数据通路时需要更多MUX。

Addu建模

R[rd]←R[rs]+R[rt]

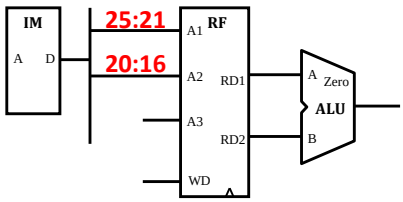
- -3环节：为了支持ALU计算，RF需要读取2个操作数
- RF读取操作数涉及2个要素，即第1操作数编号、第2操作数编号
  - 推理1：第1操作数编号是指令的rs域（先假设RF的RD1输出对应rs）
  - 推理2：第2操作数编号是指令的rt域（先假设RF的RD2输出对应rt）

环节	步骤	语义	连接关系	功能部件	控制信号取值
		31 26 25 21 20 16 15 11 10 6 5 0			
		special 000000	rs	rt	rd	0 00000	addu 100001
		6	5	5	5	5	6
-3	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2>	RF	

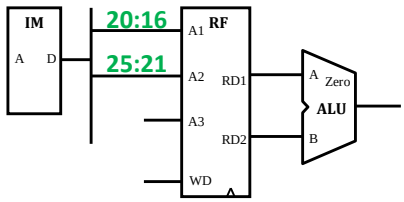
指令	NPC	PC	IM	RF				ALU		DM	
	PC	NPC	A	A1	A2	A3	WD	A	B	A	WD
addu				IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C	RF.RD1	RF.RD2		

IM与RF的连接关系

- RF有2个读寄存器编号，A1和A2
- 问题：RF的A1/A2对应**rs/rt**还是**rt/rs**？

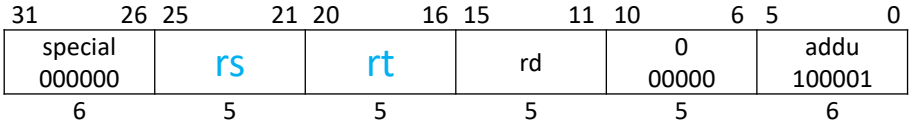


A)



B)

- 解答
  - 1）对于任一指令而言，两种连接均可以
  - 2）推荐采用固定连接方案（为保持一致性，习惯上采用方案A）



计算机组成与实现

Addu建模

R[rd]←R[rs]+R[rt]

- -4环节：必然是取指令
- 取指令涉及2个要素，即读取IM、计算PC+4、更新PC
  - 推理1：为了读取IM，需要PC驱动IM的地址线
  - 推理2：为了计算PC+4，需要PC输入至NPC
  - 推理3：为了更新PC，需要将NPC计算结果再写入PC

环节	步骤	语义	连接关系	功能部件	控制信号取值
-1	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1
-2	执行	ALU执行加法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	ALUOp:ADD
-3	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2>	RF	
-4	读指令	读取指令 计算下条指令地址 更新PC	<**PC.DO,IM.A**> <**PC.DO,NPC.PC**> <**NPC.NPC,PC.DI**>	IM PC NPC	

指令	NPC	PC	IM	RF				ALU		DM	
	PC	DI	A	A1	A2	A3	WD	A	B	A	WD
addu	PC.DO	NPC.NPC	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C	RF.RD1	RF.RD2		

计算机组成与实现

Addu建模

R[rd]←R[rs]+R[rt]

- 将推理过程倒置后，得到正向的推理过程

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2>	RF	
3	执行	ALU执行加法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	ALUOp:ADD
4	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1

- 将推理得到的连接关系转换为表格表示
  - 示例：红绿蓝三色对应环节1的3个连接关系

指令	NPC	PC	IM	RF				ALU		DM	
	PC	DI	A	A1	A2	A3	WD	A	B	A	WD
addu	PC.DO	NPC.NPC	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C	RF.RD1	RF.RD2		

目录

Subu建模

R[rd]←R[rs]-R[rt]

- Subu与addu高度相似，区别仅仅在于ALU执行减法运算

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2>	RF	
3	执行	ALU执行加法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	ALUOp:**SUB**
4	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1

指令	NPC	PC	IM	RF				ALU		DM	
	PC	DI	A	A1	A2	A3	WD	A	B	A	WD
subu	PC.DO	NPC.NPC	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C	RF.RD1	RF.RD2		

目录

Ori建模

R[rt]←R[rs] OR zero_ext(imm16)

- zero_ext()：这是一个新的计算需求
  - 原有的功能部件无法满足该需求
- EXT：新增功能部件，用于将16位数进行0扩展为32位数

信号名称	方向	描述
Imm16[15:0]	输入	16位输入。
Ext[31:0]	输出	32位0扩展结果。

HDL建模：Extender.v

```
module EXT( Imm16, Ext ) ;
    。。。
    assign Ext = {16{0}, Imm16} ;
end module
```


Ori建模

R[rt]←R[rs] OR zero_ext(imm16)

- Ori与addu/sub非常相似，区别在于
  - 1）ALU的第2个操作数为EXT的扩展结果
  - 2）ALU执行OR运算
  - 3）写入rt寄存器
  - 3）

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[15:00],EXT.IMM16>	RF EXT	
3	执行	ALU执行加法	<RF.RD1,ALU.A> <EXT.Ext,ALU.B>	ALU	ALUOp:**OR**
4	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[20:16],RF.A3>	RF	RFWr:1

指令	NPC	PC	IM	RF				EXT	ALU		DM	
	PC	NPC	A	A1	A2	A3	WD	Imm16	A	B	A	WD
addiu	PC.DO	NPC.NPC	PC.DO	IM.D[25:21]		IM.D[20:16]	ALU.C	IM.D[15:0]	RF.RD1	EXT.Ext		

目录

Lw建模

R[rt]←MEM[R[rs]+sign_ext(imm16)]

- 新增DM功能部件
  - 由于不是写存储器操作，因此DM.WD无需连接
- 新增符号扩展功能
  - 0扩展与符号扩展，其输入位数、输出位数及基本目的相同
  - 根据“高内聚、低耦合”原则，由EXT同时实现两种扩展较为合理
    - 由于EXT同时支持两种扩展，因此必须增加控制信号EXTOp

信号名称	方向	描述
Imm[15:0]	输入	16位输入。
EXTOp	输入	扩展功能选择 0：符号扩展 1：无符号扩展
Ext[31:0]	输出	32位0扩展结果。

Lw建模

R[rt]←MEM[R[rs]+sign_ext(imm16)]

- 新增DM功能部件
  - 由于不是写存储器操作，因此DM.WD无需连接
- 新增符号扩展功能
  - 0扩展与符号扩展，其输入位数、输出位数及基本目的相同
  - 根据“高内聚、低耦合”原则，由EXT同时实现两种扩展较为合理
    - 由于EXT同时支持两种扩展，因此必须增加控制信号EXTOp

HDL建模：Extender.v

```
module EXT( Imm, F, Ext ) ;
    。。。
    assign Ext = EXTOp==`ZEXT ? {16{0}, Imm} :
                                {16{Imm[15]}, Imm} ;
end module
```

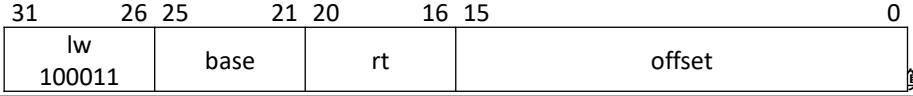
※※※
书写表达式时，尽量多使用宏，增加可读性。

Lw建模

R[rt]←MEM[R[rs]+sign_ext(imm16)]

- ALU输入寄存器与扩展数，执行加法，结果输出至DM地址
  - 1）地址计算与ori的计算过程类似（寄存器与扩展数运算）
  - 2）ALU具有执行加法的功能
- 为完成回写，DM的数据输出连接至RF的WD，且DM写使能为1

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[15:00],EXT.IMM16>	RF EXT	EXTOp:SEXT
3	执行	ALU执行加法	<RF.RD1,ALU.A> <EXT.Ext,ALU.B>	ALU	ALUOp:ADD
4	访存	读取DM	<ALU.C,DM.A>	DM	
4	回写	计算结果回写至rd寄存器	<DM.RD,RF.WD> <IM.D[20:16],RF.A3>	RF	RFWr:1



Lw建模

R[rt]←MEM[R[rs]+sign_ext(imm16)]

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[15:00],EXT.IMM16>	RF EXT	EXTOp:SEXT
3	执行	ALU执行加法	<RF.RD1,ALU.A> <EXT.Ext,ALU.B>	ALU	ALUOp:ADD
4	访存	读取DM	<ALU.C,DM.A>	DM	
5	回写	计算结果回写至rd寄存器	<DM.RD,RF.WD> <IM.D[20:16],RF.A3>	RF	RFWr:1

指令	NPC	PC	IM	RF				EXT	ALU		DM	
	PC	NPC	A	A1	A2	A3	WD	Imm16	A	B	A	WD
lw	PC.PC	NPC.NPC	PC.PC	IM.D[25:21]		IM.D[20:16]	DM.RD	IM.D[15:0]	RF.RD1	EXT.Ext	ALU.C	

- 注意：由于EXT部件被修改了，因此凡是与该部件相关的指令均需被重新建模
  - 主要是建模需要考虑EXT的控制信号取值

目录

Sw建模

MEM[R[rs]+sign_ext(imm16)]←R[rt]

- Sw与lw在地址计算方面完全一致
- Sw与lw不同点在于其不需要读DM，而是将rt寄存器写入DM
  - 即RF的RD2要写入DM

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[15:00],EXT.IMM16>	RF EXT	EXTOp:SEXT
3	执行	ALU执行加法	<RF.RD1,ALU.A> <EXT.Ext,ALU.B>	ALU	ALUOp:ADD
4	访存	读取DM	<ALU.C,DM.A> <RF.RD2,DM.WD>	DM	DMWR:1

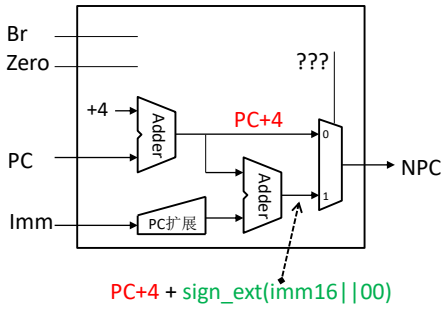
指令	NPC	PC	IM	RF				EXT	ALU		DM	
	PC	NPC	A	A1	A2	A3	WD	Imm16	A	B	A	WD
lw	PC.PC	NPC.NPC	PC.PC	IM.D[25:21]				IM.D[15:0]	RF.RD1	EXT.Ext	ALU.C	RF.RD2

目录

Beq建模

```
if ( R[rs] == R[rt] )
    PC←PC+4 + sign_ext(imm16||00)
else
    PC←PC+4
```

- 本质上，beq涉及2大功能
  - 功能1：根据比较的结果，计算PC。这属于NPC的功能范畴
  - 功能2：寄存器比较。让ALU执行减法，然后把比较结果zero传递给NPC
- 对于NPC，现在需要知道当前指令是否是beq及zero的结果

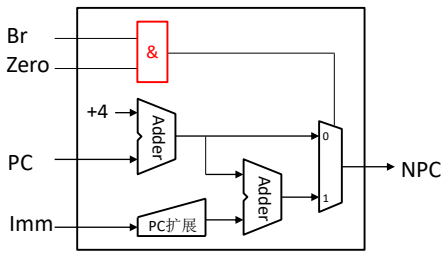


信号名	方向	描述
PC[31:0]	I	32位输入
Imm[15:0]	I	16位立即数
Br	I	beq指令标志 1：当前指令是beq 0：当前指令不是beq
Zero	1	rs和rt相等标志 1：相等 0：不等
NPC[31:0]	O	32位输出

Beq建模

```
if ( R[rs] == R[rt] )
    PC←PC+4 + sign_ext(imm16||00)
else
    PC←PC+4
```

- 本质上，beq涉及2大功能
  - 功能1：根据比较的结果，计算PC。这属于NPC的功能范畴
  - 功能2：寄存器比较。让ALU执行减法，然后把比较结果zero传递给NPC
- 对于NPC，现在需要知道当前指令是否是beq及zero的结果



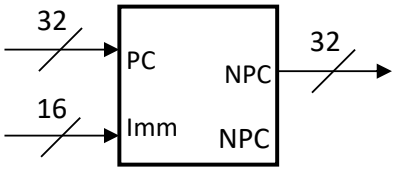
Br	Zero	MUX
0	0	0
0	1	0
1	0	0
1	1	1

信号名	方向	描述
PC[31:0]	I	32位输入
Imm[15:0]	I	16位立即数
Br	I	beq指令标志 1：当前指令是beq 0：当前指令不是beq
Zero	1	rs和rt相等标志 1：相等 0：不等
NPC[31:0]	O	32位输出

Beq建模

```
if ( R[rs] == R[rt] )
    PC←PC+4 + sign_ext(imm16||00)
else
    PC←PC+4
```

- 本质上，beq涉及2大功能
  - 功能1：根据比较的结果，计算PC。这属于NPC的功能范畴
  - 功能2：寄存器比较。让ALU执行减法，然后把比较结果zero传递给NPC
- 对于NPC，现在需要知道当前指令是否是beq及zero的结果



※图中未包含Br和Zero

信号名	方向	描述
PC[31:0]	I	32位输入
Imm[15:0]	I	16位立即数
Br	I	beq指令标志 1：当前指令是beq 0：当前指令不是beq
Zero	1	rs和rt相等标志 1：相等 0：不等
NPC[31:0]	O	32位输出

Beq建模

```
if ( R[rs] == R[rt] )
    PC←PC+4 + sign_ext(imm16||00)
else
    PC←PC+4
```

- NPC计算PC
- ALU执行减法

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <IM.D[15:0],NPC.Imm> <NPC.NPC,PC.DI>	IM PC NPC	Br:1 Zero:ALU的zero
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2 >	RF	
3	执行	ALU执行减法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	ALUOp:SUB

指令	NPC		PC	IM	RF				ALU		DM	
	PC	Imm	NPC	A	A1	A2	A3	WD	A	B	A	WD
beq	PC.PC	IM.D[15:0]	NPC.NPC	PC.PC	IM.D[25:21]	IM.D[20:16]			RF.RD1	RF.RD2		

及时调整已有设计

- 一旦增加了新的功能部件或修改了已有的功能部件，就需要及时调整已经完成的设计
- 示例：由于修改了NPC的设计，因此需要调整addu的设计
  - 在beq之前的设计中，由于各指令均顺序执行，因此NPC是单功能部件
  - 现为满足beq的计算需求调整了NPC的设计，所以需修改addu指令的设计
    - subu、lw、sw等指令类似

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	**Br:0**
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2>	RF	
3	执行	ALU执行加法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	ALUOp:ADD
4	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	RFWr:1

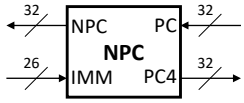
※后续还有类似的问题

目录

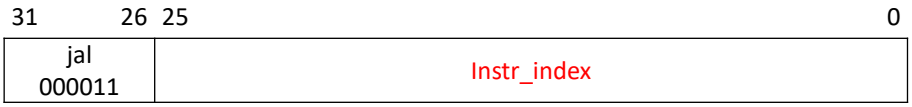
Jal建模

$PC \leftarrow PC_{31..28} \| instr\_index \| 0^2$
$R[31] \leftarrow PC+4$

- 包含2个操作：①计算PC值；②将PC+4写入R31
- NPC改造1：支持新的PC计算需求
  - NPC的功能定位是计算PC值
  - NPC已输入指令低16位，只需要扩大至指令低26位，就能满足全部需求
  - NPC的计算需求包含3种情况，分别是PC+4、PC+4+偏移、jal地址；必须将br调整为NPCOp[1:0]
    - br信号只有1位，不能表达3种含义
- NPC改造2：支持回写
  - NPC已经计算出了PC+4，因此只需要将PC+4输出即可



※图中未包含控制类信号



Jal建模

$PC \leftarrow PC_{31..28} \| instr\_index \| 0^2$
$R[31] \leftarrow PC+4$

- 调整后的NPC的I/O信号

信号名	方向	描述	调整说明
PC[31:0]	I	32位输入	
Imm[25:0]	I	26位立即数	从Imm[15:0]调整为Imm[25:0]
NPCOp[1:0]	I	NPC功能选择 00：计算顺序地址（PC+4） 01：计算beq地址 10：计算jal地址 11：保留	取消1位的Br，改为2位的NPCOp
Zero	I	rs和rt相等标志 1：相等 0：不等	
NPC[31:0]	O	32位输出	
PC4[31:0]	O	32位输出	输出PC+4值

注意：由于NPC的控制信号发生了变化，因此前面各条指令的NPC控制信号取值也需要随之调整。

Jal建模

$PC \leftarrow PC_{31..28} \| instr\_index \| 0^2$
$R[31] \leftarrow PC+4$

- 对于回写数据来说，数据来自NPC输出的PC4
- 对于回写寄存器编号来说，需要直接表示为0x1F
  - 其并未出现在指令格式中，而是固定为31

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <IM.D[25:0],NPC.Imm> <NPC.NPC,PC.DI>	IM PC NPC	NPCOp:JAL
2	回写		<0x1F,RF.A3> <NPC.PC4,RF.WD >	RF	RFWr:1

指令	NPC		PC	IM	RF				EXT	ALU		DM	
	PC	Imm	DI	A	A1	A2	A3	WD	Imm	A	B	A	WD
jal	PC.DO	IM.D[25:0]	NPC.NPC	PC.DO			0x1F	NPC.PC4					

目录

Jr建模

PC←R[rs]

- 设计分歧：虽然Jr的功能很简单，但却有2种可能思路
  - 方案1：将RF输出的rs值直接输出至PC
  - 方案2：将RF输出的rs值先输出至NPC，然后再从NPC输出至PC
- 基本思路：遵循某些基本原则或方法
- 基本原则：“高内聚低耦合”中的最少知识原则
  - 方案1：PC与NPC都是“知道”如何计算PC值的功能部件
  - 方案2：NPC是唯一“知道” 如何计算PC值的功能部件

31 26	25 21	20 11	10 6	5 0
special 000000	rs	0 00 0000 0000	0 00000	jr 001000

Jr建模

PC←R[rs]

- NPC接口：①增加函数返回地址RA[31:0]；②增加新的计算模式

信号名	方向	描述	调整说明
PC[31:0]	I	32位输入	
Imm[25:0]	I	26位立即数	从Imm[15:0]调整为Imm[25:0]
RA[31:0]	I	32位返回地址	rs寄存器保存的返回地址
NPCOp[1:0]	I	NPC功能选择 00：计算顺序地址（PC+4） 01：计算beq地址 10：计算jal地址 11：计算jr地址	NPCOp的0b11项用于产生jr相关的PC目的地址
Zero	I	rs和rt相等标志 1：相等 0：不等	
NPC[31:0]	O	32位输出	
PC4[31:0]	O	32位输出	输出PC+4值

注意：由于NPC的控制信号发生了变化，因此前面各条指令的NPC控制信号取值也需要随之调整。

Jr建模

PC←R[rs]

- 推理发现存在冲突：同一个部件执行多个操作。如何选择？
  - 在环节1和3，NPC执行不同的计算功能

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	NPCOp:PC+4 ①
2	读操作数		<IM.D[26:21],RF.A1>	RF	
3	计算返回地址		<RF.RD1,NPC.RA>	NPC	NPCOp:JR ②
4	更新PC	将rs值写入PC	<NPC.NPC,PC.DI>	PC	

- 选择后执行的，即环节3。原因：
  - 单周期特点是所有操作都必须在一个时钟周期内完成
  - 逻辑上，后发生的事件是会覆盖先发生的事件的

```
// a的值为6
a = 5 ;
a = 6 ;
```


Jr建模

PC←R[rs]

- 单周期的特点是所有操作都必须在一个时钟周期内完成。这意味

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	~~NPCOp:PC+4 ①~~
2	读操作数		<IM.D[26:21],RF.A1>	RF	
3	计算返回地址		<RF.RD1,NPC.RA>	NPC	NPCOp:JR ②
4	更新PC	将rs值写入PC	<NPC.NPC,PC.DI>	PC	

指令	NPC			PC	IM	RF				EXT	ALU		DM	
	PC	Imm	RA	DI	A	A1	A2	A3	WD	Imm	A	B	A	WD
jal	PC.DO		RF.RD1	NPC.NPC	PC.DO	IM.D[25:21]								

目录

- 设计方法学概述
- 单周期CPU设计模型
- 数据通路基础部件建模
- 指令级别数据通路与控制器建模
- 数据通路综合方法
- 控制器综合方法
- 单周期CPU性能分析

汇聚

- 将独立建模的指令级别数据通路汇聚在一起

部件	PC	NPC			IM	RF				EXT	ALU		DM
输入信号	DI	PC	Imm	RA	A	A1	A2	A3	WD		A	B	A
addu	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C		RF.RD1	RF.RD2	
subu	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C		RF.RD1	RF.RD2	
ori	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]		IM.D[20:16]	ALU.C	IM.D[15:0]	RF.RD1	EXT.Ext	
lw	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]		IM.D[20:16]	DM.RD	IM.D[15:0]	RF.RD1	EXT.Ext	ALU.C
sw	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]		IM.D[20:16]		IM.D[15:0]	RF.RD1	EXT.Ext	ALU.C
beq	NPC.NPC	PC.DO	IM.D[15:0]		PC.DO	IM.D[25:21]	IM.D[20:16]				RF.RD1	RF.RD2	
jal	NPC.NPC	PC.DO	IM.D[25:0]	RF.RD1	PC.DO			0x1F	NPC.PC4				
jr	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]							

综合

- 归并每个输入信号的信号来源：保留不同来源

部件	PC	NPC			IM	RF				EXT	ALU		DM
输入信号	DI	PC	Imm	RA	A	A1	A2	A3	WD		A	B	A
addu	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C		RF.RD1	RF.RD2	
subu	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11]	ALU.C		RF.RD1	RF.RD2	
ori	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]		IM.D[20:16]	ALU.C	IM.D[15:0]	RF.RD1	EXT.Ext	
lw	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]		IM.D[20:16]	DM.RD	IM.D[15:0]	RF.RD1	EXT.Ext	ALU.C
sw	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]		IM.D[20:16]		IM.D[15:0]	RF.RD1	EXT.Ext	ALU.C
beq	NPC.NPC	PC.DO	IM.D[15:0]		PC.DO	IM.D[25:21]	IM.D[20:16]				RF.RD1	RF.RD2	
jal	NPC.NPC	PC.DO	IM.D[25:0]	RF.RD1	PC.DO			0x1F	NPC.PC4				
jr	NPC.NPC	PC.DO			PC.DO	IM.D[25:21]							
完整	NPC.NPC	PC.DO	IM.D[25:0] ※	RF.RD1	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11] IM.D[20:16] 0x1F	ALU.C DM.RD NPC.PC4	IM.D[15:0]	RF.RD1	RF.RD2 EXT.Ext	ALU.C

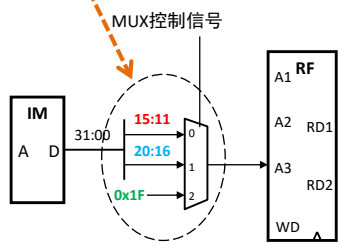
※由于IM.D[25:0]覆盖了IM.D[15:0]，因此只保留前者即可

构造MUX

- 以RF.A3为例，综合后的结果表明该信号有3个输入源：
  - IM.D[15:11]、IM.D[20:16]、0x1F

部件	PC	NPC			IM	RF				EXT	ALU		DM
输入信号	DI	PC	Imm	RA	A	A1	A2	A3	WD		A	B	A
完整	NPC.NPC	PC.DO	IM.D[25:0]	RF.RD1	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11] IM.D[20:16] 0x1F	ALU.C DM.RD NPC.PC4	IM.D[15:0]	RF.RD1	RF.RD2 EXT.Ext	ALU.C

- 根据数字电路知识可知，必须部署一个MUX
- 由此可见，MUX是“自动”综合出来



※在控制部分再详述MUX的工程化规范设计

构造MUX

- MUX各路输入信号分派并无特定的原则
  - 示例：RF的A3信号

指令与RF.A3

指令	RF.A3
addu	IM.D[15:11]
subu	IM.D[15:11]
ori	IM.D[20:16]
lw	IM.D[20:16]
sw	IM.D[20:16]
beq	
jal	0x1F
jr	

RF.A3的MUX

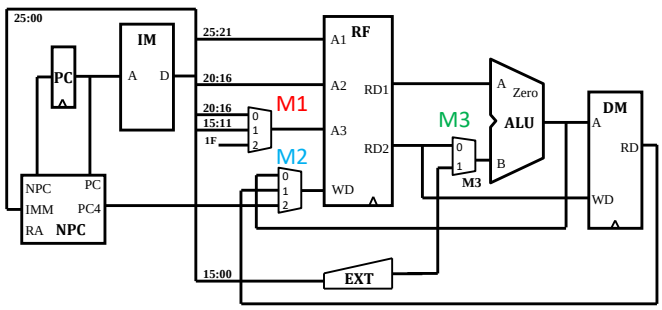
端口编号	输入源
0	IM.D[15:11]
1	IM.D[20:16]
2	0x1F

注意
一旦确定输入源与MUX端口的对应关系后，务必在数据通路、控制信号等设计上确保一致性

数据通路的两种表示方式

- 从形式建模的数据通路可以很容易的构造出图形化的数据通路

部件	PC	NPC			IM	RF				EXT	ALU		DM
输入信号	DI	PC	Imm	RA	A	A1	A2	A3	WD		A	B	A
完整	NPC.NPC	PC.DO	IM.D[25:0]	RF.RD1	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11] IM.D[20:16] 0x1F	ALU.C DM.RD NPC.PC4	IM.D[15:0]	RF.RD1	RF.RD2 EXT.Ext	ALU.C
								M1	M2			M3	



从设计模型到VerilogHDL

- 示例：ALU的B输入

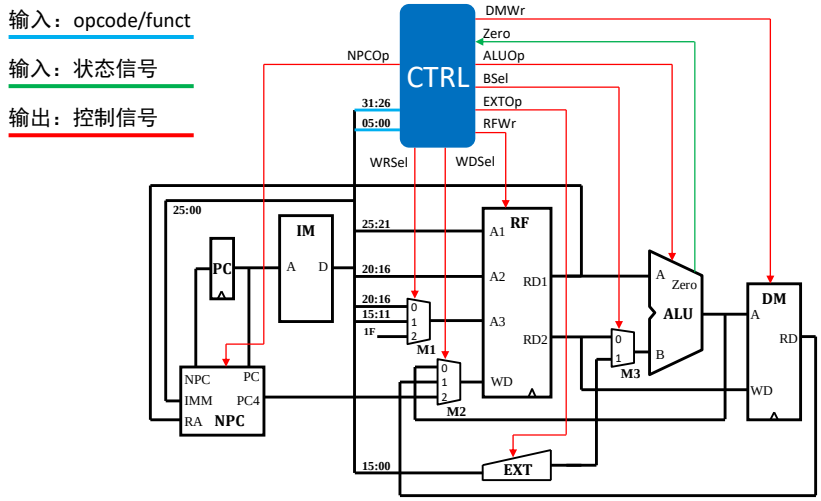
部件	PC	NPC			IM	RF				EXT	ALU		DM
输入信号	DI	PC	Imm	RA	A	A1	A2	A3	WD	Imm	A	B	A
完整	NPC.NPC	PC.DO	IM.D[25:0]	RF.RD1	PC.DO	IM.D[25:21]	IM.D[20:16]	IM.D[15:11] IM.D[20:16] 0x1F	ALU.C DM.RD NPC.PC4	IM.D[15:0]	RF.RD1	RF.RD2 EXT.Ext	ALU.C

```
wire    [31:0]      RD2 ;
wire    [31:0]      Ext ;
wire    [31:0]      ALU_B ;

RF    U_RF( ..., RD2, ... ) ;          // 实例化寄存器堆
EXT   U_EXT( ..., Ext, ... ) ;         // 实例化扩展单元
ALU   U_ALU( ..., ALU_B, ... ) ;       // 实例化ALU

// 实例化
MUX32_2_1 U_MUX_ALUB( RD2, Ext, ALU_B, ALUBSrc ) ;
```


目录

数据通路与控制器

- 控制器的职责：根据各指令的opcode（指令的31:26位）和funct（指令的05:00位），产生各功能部件的控制信号表达式



计算机组成与实现

合成各指令的控制信号取值矩阵

- 示例：addu指令

环节	步骤	语义	连接关系	功能部件	控制信号取值
1	读指令	读取指令 计算下条指令地址 更新PC	<PC.DO,IM.A> <PC.DO,NPC.PC> <NPC.NPC,PC.DI>	IM PC NPC	**NPCOp:PC+4**
2	读操作数		<IM.D[25:21],RF.A1> <IM.D[20:16],RF.A2>	RF	
3	执行	ALU执行加法	<RF.RD1,ALU.A> <RF.RD2,ALU.B>	ALU	**ALUOp:ADD**
4	回写	计算结果回写至rd寄存器	<ALU.C,RF.WD> <IM.D[15:11],RF.A3>	RF	**RFWr:1**

- 对于不涉及的组合逻辑功能部件，如EXT，其控制信号可以设置为**X**
  - X有助于化简。一般来说，令其为0会得到更简单的控制信号表达式
- 对于不涉及的存储功能部件，如DM，其写使能必须设置为**0**
  - 如果DM的写使能不设置为0，则意味着addu指令也会导致DM被写入

指令	NPCOp	RFWr	EXTOp	ALUOp	DMWr	M1Sel	M2Sel	M3Sel
addu	PC+4	1	X	ADD	0	?	?	?

计算机组成与实现

合成各指令的控制信号取值矩阵

- 完整的控制信号取值矩阵（不包含MUX的控制信号）

指令	NPCOp	RFWr	EXTOp	ALUOp	DMWr	M1Sel	M2Sel	M3Sel
addu	PC+4	1	X	ADD	0	?	?	?
subu	PC+4	1	X	SUB	0	?	?	?
ori	PC+4	1	0	OR	0	?	?	?
lw	PC+4	1	1	ADD	0	?	?	?
sw	PC+4	0	1	ADD	1	?	?	?
beq	0b01	0	X	SUB	0	?	?	?
jal	0b10	1	X	X	0	?	?	?
jr	0b11	0	X	X	0	?	?	?

合成各指令的控制信号取值矩阵

- 构造MUX控制信号真值表
  - 根据指令与输入信号关系及端口与输入源关系，就可以“自动”确定控制信号表达式。以sw指令为例
    - ①指令的20:16位指定寄存器堆的回写编号，由此可以查出MUX的端口编号
    - ②MUX的端口编号与控制信号取值是一一对应的

指令与RF.A3

指令	RF.A3
addu	IM.D[15:11]
subu	IM.D[15:11]
ori	IM.D[20:16]
lw	IM.D[20:16]
sw	IM.D[20:16]
beq	
jal	0x1F
jr	

RF.A3的MUX

端口编号	输入源
0	IM.D[15:11]
1	IM.D[20:16]
2	0x1F

①　②

MUX控制信号真值表

指令	端口编号	控制信号取值
add	0端口	0b00
sub	0端口	0b00
ori	1端口	0b01
lw	1端口	0b01
sw	1端口	0b01
slt	0端口	0b00
beq		
jal	2端口	0b10
jr		

合成各指令的控制信号取值矩阵

- 完整的控制信号取值矩阵（包含MUX的控制信号）

指令	NPCOp	RFWr	EXTOp	ALUOp	DMWr	M1Sel	M2Sel	M3Sel
addu	PC+4	1	X	ADD	0	00	00	0
subu	PC+4	1	X	SUB	0	00	00	0
ori	PC+4	1	0	OR	0	XX	00	1
lw	PC+4	1	1	ADD	0	XX	01	1
sw	PC+4	0	1	ADD	1	XX	XX	1
beq	0b01	0	X	SUB	0	XX	XX	0
jal	0b10	1	X	X	0	XX	10	X
jr	0b11	0	X	X	0	XX	XX	X

- 对比NPCOp/ALUOp与M1Sel/M2Sel/M3Sel，显然宏定义的表达方式更好：不仅易于理解，而且有利于工程维护

合成各指令的控制信号取值矩阵

- 完整的控制信号取值矩阵（包含MUX的控制信号）

指令	NPCOp	RFWr	EXTOp	ALUOp	DMWr	M1Sel	M2Sel	M3Sel
addu	PC+4	1	X	ADD	0	00	00	0
subu	PC+4	1	X	SUB	0	00	00	0
ori	PC+4	1	0	OR	0	XX	00	1
lw	PC+4	1	1	ADD	0	XX	01	1
sw	PC+4	0	1	ADD	1	XX	XX	1
beq	0b01	0	X	SUB	0	XX	XX	0
jal	0b10	1	X	X	0	XX	10	X
jr	0b11	0	X	X	0	XX	XX	X

- Q：该如何将这个表格转换为组合逻辑表达式？

计算机组成与实现

构造控制信号的布尔表达式

- 将每条指令用一个变量与之对应
  - 使用指令的opcode与funct产生变量
  - 为了提高可读性，变量名就是指令名
  - 由于R型指令的opcode都为0，因此需产生一个单独变量Rtype
- 示例

```
beq = op[5]'·op[4]'·op[3]'·op[2]·op[1]'·op[0]'

Rtype = op[5]'·op[4]'·op[3]'·op[2]'·op[1]'·op[0]'

add = Rtype·funct[5]·funct[4]'·funct[3]'
    ·funct[2]'·funct[1]'·funct[0]'
```

指令	opcde	funct
…	…	…
addu	000000	100000
beq	000100	
…	…	…

构造控制信号的布尔表达式

- 将每条指令用一个变量与之对应
  - 使用指令的opcode与funct产生变量
  - 为了提高可读性，变量名就是指令名
  - 由于R型指令的opcode都为0，因此需产生一个单独变量Rtype
- 示例：Verilog表达式

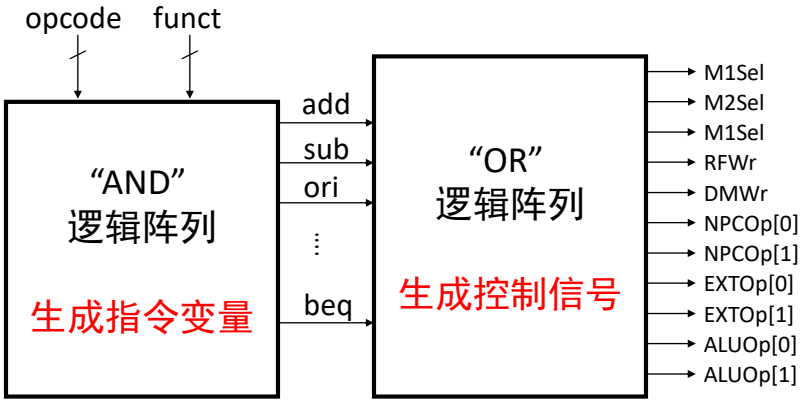
```
assign beq = (op==`BEQ) ;

assign Rtype = (op==6'b000000) ;

assign add = Rtype&(funct==`ADDU) ;
```

`BEQ和`ADDU是宏定义。宏有助于提高可读性，更易于工程维护

指令	opcde	funct
…	…	…
addu	000000	100000
beq	000100	
…	…	…

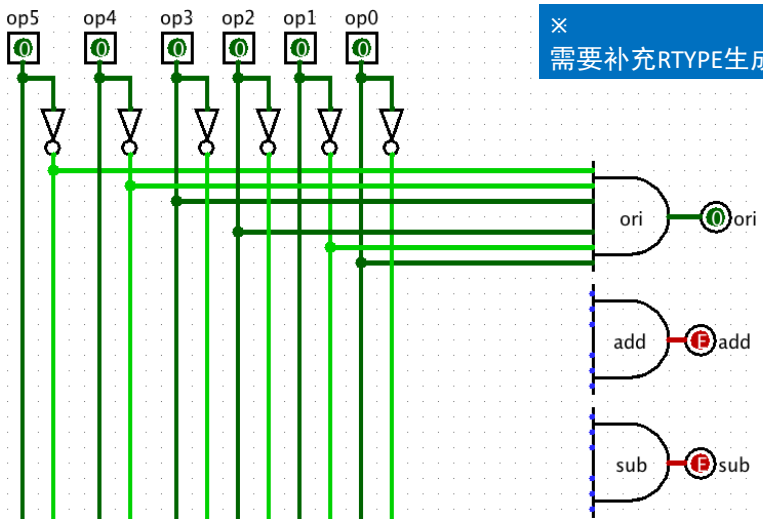
实现控制器

- 用AND和OR两个阵列来实现控制器



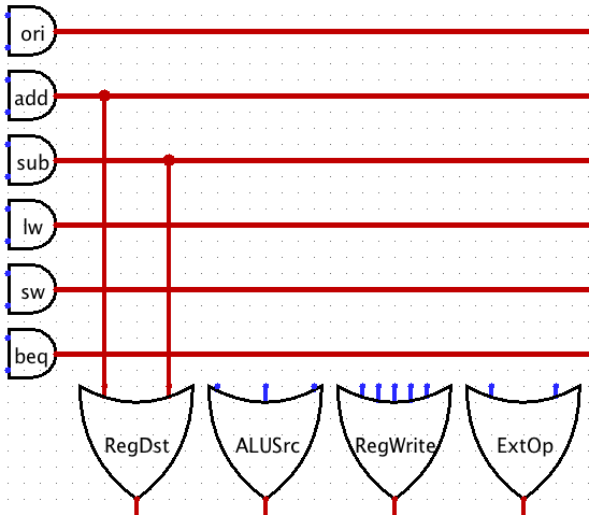
实现控制器

- 用AND和OR两个阵列来实现控制器
  - AND阵列：用Logisim实现



实现控制器

- 用AND和OR两个阵列来实现控制器
  - OR阵列：用Logisim实现



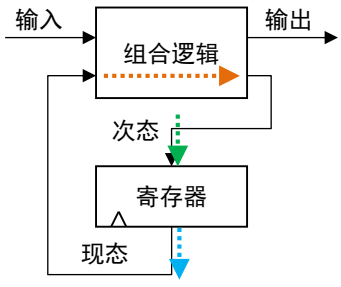
目录

寄存器时序术语

- 建立时间（Setup Time）：输入信号在时钟上升沿之前就必须有效的时间
- 保持时间（Hold Time）：输入信号在时钟上升沿之后仍然必须保持有效的时间
- 输出延迟（ CLK-to-Q）：输出信号在时钟上升沿之后输出有效值的时间

最大时钟频率

- 如何计算电路的最大频率？
  - 最大频率取决于最大延迟
  - 最大延迟取决于为了确保寄存器的正确输入所需要的时间

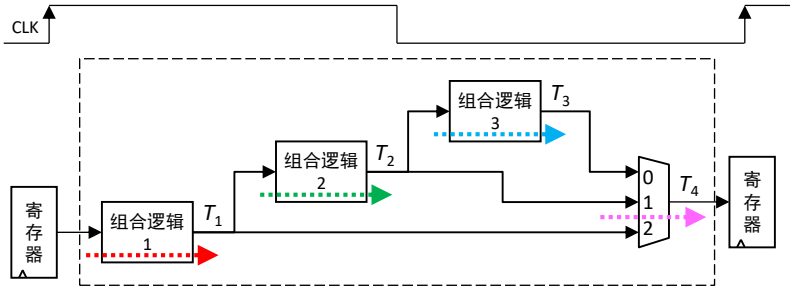


T = 建立时间 +
输出延迟 +
组合逻辑延迟

$f_{max} = 1/T$

关键路径

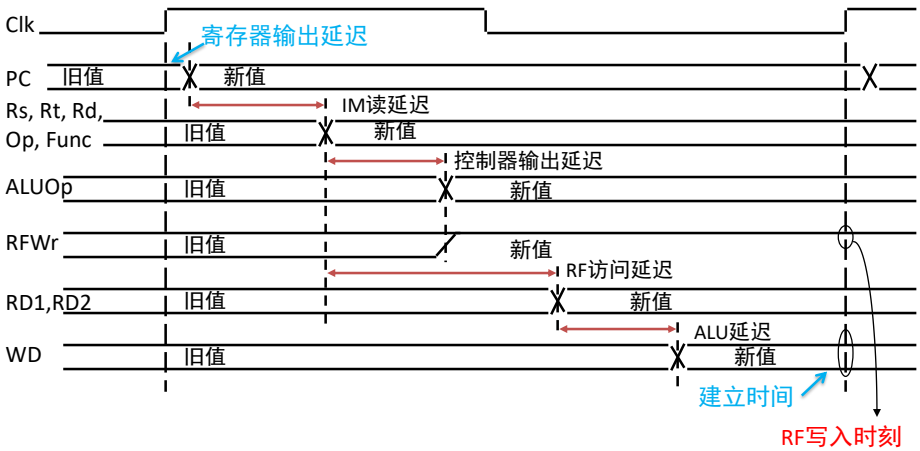
- 关键路径：电路中的任意2个寄存器之间的最大延迟
- 电路的时钟周期必须大于关键路径，否则信号将不能正确的传递到下一个寄存器



关键路径 = 组合逻辑延迟1 +
组合逻辑延迟2 +
组合逻辑延迟3 +
MUX延迟

Add指令执行延迟分析

- 执行延迟包括：PC输出延迟、IM读延迟、控制器输出延迟、ALU运算延迟、寄存器建立延迟



单周期性能

- 假设：RF的读写延迟均为100ps，其他部件延迟为200ps
- 最大时钟频率是多少？
  - lw延迟最大，即关键路径为800ns。因此，最大时钟频率为1.25GHz

指令	读取指令	读寄存器	ALU	数据存取	写寄存器	理想执行时间	实际执行时间
addu	200	100	200		100	600	800
subu	200	100	200		100	600	800
ori	200	100	200		100	600	800
lw	200	100	200	200	100	800	800
sw	200	100	200	200		700	800
beq	200	100	200			500	800
jal	200				100	300	800
jr	200	100				300	800

- 如何提高时钟频率？