Ch 6: CPU - Datapath and Control

中央处理器：数据通路和控制器

第一讲 单周期数据通路的设计
第二讲 单周期控制器的设计
第三讲 多周期处理器的设计
第四讲 微程序控制器设计与异常处理

第一讲 单周期数据通路的设计

主 要 内 容

- CPU的功能及其与计算机性能的关系
- 数据通路的位置
- 单周期数据通路的设计
  - 数据通路的功能和实现
    - 操作元件（组合逻辑部件）
    - 状态 / 存储元件（时序逻辑部件）
  - 数据通路的定时
- 选择MIPS指令集的一个子集作为CPU的实现目标
  - 下条指令地址计算与取指令部件
  - R型指令的数据通路
  - 访存指令的数据通路
  - 立即数运算指令的数据通路
  - 分支和跳转指令的数据通路
- 综合所有指令的数据通路

CPU功能及其与计算机性能的关系

- CPU执行指令的过程：
  - 取指令
  - PC+1送PC

  （取指阶段）
  - 指令译码
  - 进行主存地址运算
  - 取操作数
  - 进行算术 / 逻辑运算
  - 存结果
  - 判断和检测“异常”事件
  - 若有异常，则自动切换到异常处理程序
  - 检测是否有“中断”请求，有则转中断处理

  （译码和执行阶段）

  （指令执行过程）

问题：
“取指令”一定在最开始做吗？
“PC+1”一定在译码之前做吗？
“译码”须在指令执行前做吗？
“异常”和“中断”的差别是什么？

CPU的基本功能
（1）控制指令执行顺序
（2）控制指令执行操作
（3）控制操作时序
（4）对数据进行运算
（5）对访存或I/O访问进行控制
（6）异常和中断处理

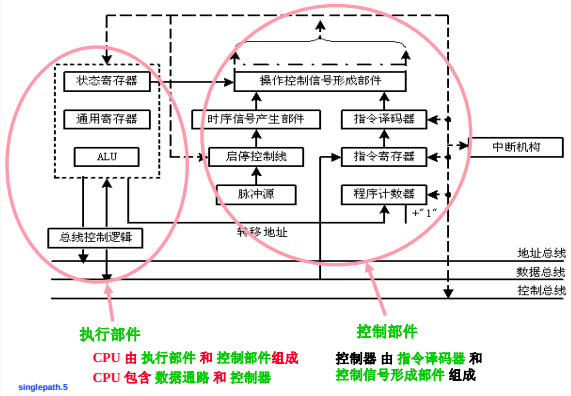
- CPU的实现与计算机性能的关系
  - 计算机性能(程序执行快慢)由三个关键因素决定：
    - 指令数目、时钟周期、CPI
      - 指令数目由编译器和指令集决定
      - 时钟周期和CPI由CPU的实现来决定

因此，CPU的设计与实现非常重要！它直接影响计算机的性能。

组成指令功能的四种基本操作

- 每条指令的功能总是由以下四种基本操作来实现：
  （1）读取某一主存单元的内容，并将其装入某个寄存器；
  （2）把一个数据从某个寄存器存入给定的主存单元中；
  （3）把一个数据从某个寄存器送到另一个寄存器或者ALU；
  （4 进行某种算术运算或逻辑运算，将结果送入某个寄存器。
- 操作功能可形式化描述
  - 描述语言称为寄存器传送语言RTL (Register Transfer Language)
  - 本章所用的RTL规定如下：
    （1）用R[r]表示寄存器r的内容；
    （2）用M[addr]表示读取主存单元addr的内容；
    （3）传送方向用“←”表示，传送源在右，传送目的在左；
    （4）程序计数器PC直接用PC表示其内容；
    （5）用OP[data]表示对数据data进行OP操作。

CPU基本组成原理图

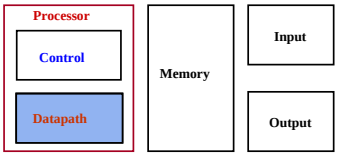


执行部件
CPU 由 执行部件 和 控制部件组成
CPU 包含 数据通路 和 控制器

控制部件
控制器 由 指令译码器 和 控制信号形成部件 组成

数据通路的位置

- 计算机的五大组成部分：



- 什么是数据通路（DataPath）？
  - 指令执行过程中，数据所经过的路径，包括路径中的部件。它是指令的执行部件。
- 控制器（Control）的功能是什么？
  - 对指令进行译码，生成指令对应的控制信号，控制数据通路的动作。能对指令的执行部件发出控制信号，是指令的控制部件。

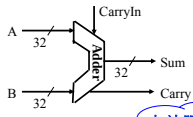
数据通路的基本结构

- 数据通路由两类部件组成
  - 组合逻辑元件（也称操作元件）
  - 存储元件（也称状态元件）
- 元件间的连接方式
  - 总线连接方式
  - 分散连接方式
- 数据通路如何构成?
  - 由“操作元件”和“存储元件”通过总线方式或分散方式连接而成
- 数据通路的功能是什么？
  - 进行数据存储、处理、传送

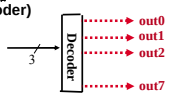
因此，数据通路是由操作元件和存储元件通过总线方式或分散方式连接而成的进行数据存储、处理、传送的路径。

操作元件：组合逻辑电路

- 加法器 (Adder)



- 译码器 (Decoder)

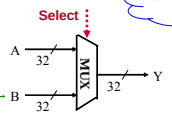


加法器需要什么控制信号?

控制信号

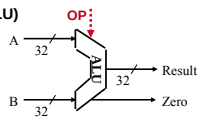
多路选择器 (MUX)

二选一 也可以多选一



何时要用到adder, ALU, MUX or Decoder?

算逻部件(ALU)



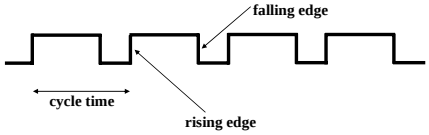
组合逻辑元件的特点：

其输出只取决于当前的输入。即：输入一样，其输出也一样

定时：所有输入到达后，经过一定的逻辑门延时，输出端改变，并保持到下次改变，不需要时钟信号来定时

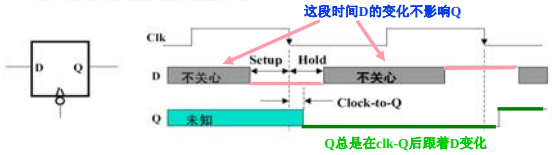
状态元件：时序逻辑电路

- 状态（存储）元件的特点：
  - 具有存储功能，在时钟控制下输入状态被写到电路中，直到下一个时钟到达
  - 输入端状态由时钟决定何时被写入，输出端状态随时可以读出
- 定时方式：规定信号何时写入状态元件或何时从状态元件读出
  - 边沿触发（edge-triggered）方式：
    - 状态单元中的值只在时钟边沿改变。每个时钟周期改变一次。
      - 上升沿（rising edge）触发：在时钟正跳变时进行读/写。
      - 下降沿（falling edge）触发：在时钟负跳变时进行读/写。



- 最简单的状态单元（回顾：数字逻辑电路课程内容）：
  - D触发器：一个时钟输入、一个状态输入、一个状态输出

存储元件中何时状态被改变?



Q总是在clk-Q后跟着D变化

- 建立时间（Setup Time）：在触发时钟边沿 之前 输入必须稳定
- 保持时间（Hold Time）：在触发时钟边沿 之后 输入必须保持
- Clock-to-Q time: (Latch Prop)
  - 在触发时钟边沿，输出并不能立即变化

切记：状态单元的输入信息总是在一个时钟边沿到达后的“Clk-to-Q”时才被写入到单元中，此时的输出才反映新的状态值

数据通路中的状态元件有两种：寄存器(组) + 存储器

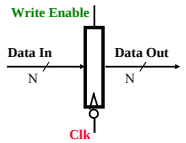
寄存器的种类

- 寄存器由N位触发器构成
- 有各种不同类型的寄存器
  - 由锁存器构成的暂存器：带“写使能”信号
  - 用于和总线相连的、输出端带三态门的寄存器：带“三态门控”信号
  - 带“复位”（清0）功能的寄存器：带“复位”信号
  - 带计数（自增）功能的寄存器：可带“自增”信号
  - 带移位功能的寄存器：带“移位”信号
  - 组合上述多个功能的寄存器：带多个控制信号
- 寄存器组有若干个寄存器组成
  - 通常是双口：两个读口 + 一个写口
- 可带时钟输入信号
  - 用于控制输入信号何时被写入到寄存器中

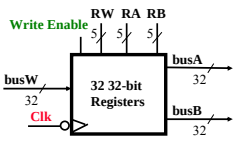
经过一个clk-to-Q，输入信号在寄存器的输出端有效！

存储元件: 寄存器和寄存器组

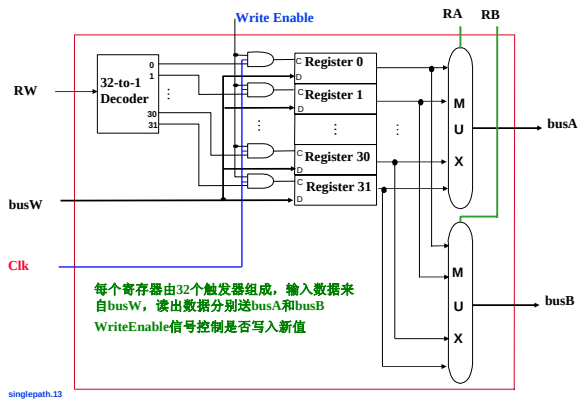
- 寄存器（Register）
  - 有一个写使能（Write Enable-WE）信号
    - 0: 时钟边沿到来时，输出不变
    - 1: 时钟边沿到来时，输出开始变为输入
  - 若在每个时钟边沿都写入，则不需要WE信号



- 寄存器组（Register File）
  - 两个读口（组合逻辑操作）：busA和busB分别由RA和RB给出地址。地址RA或RB有效后，经一个“取数时间(AccessTime)”，BusA和BusB有效。
  - 一个写口（时序逻辑操作）：写使能为1的情况下，时钟边沿到来时，busW传来的值开始被写入RW指定的寄存器中。



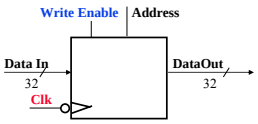
寄存器组的内部结构



每个寄存器由32个触发器组成，输入数据来自busW，读出数据分别送busA和busB

WriteEnable信号控制是否写入新值

存储元件: 理想存储器



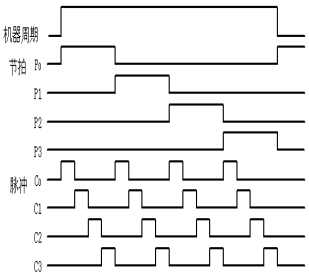
- 理想存储器（ idealized memory ）
  - Data Out：32位读出数据
  - Data In： 32位写入数据
  - Address：读写公用一个32位地址
  - 读操作（组合逻辑操作）：地址Address有效后，经一个“取数时间AccessTime”，Data Out上数据有效。
  - 写操作（时序逻辑操作）：写使能为1的情况下，时钟Clk边沿到来时，Data In传来的值开始被写入Address指定的存储单元中。

为简化数据通路操作的说明，在此把存储器简化为带时钟信号Clk的理想模型。

数据通路与时序控制

- 什么叫同步系统(Synchronous system)?
  - 所有动作有专门时序信号来定时
  - 由时序信号规定何时发出什么动作

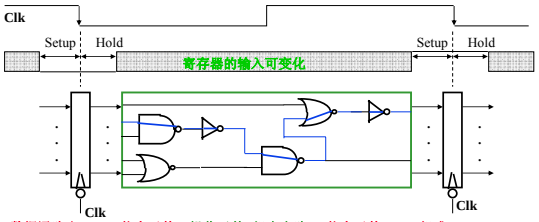
    例如，指令执行过程每一步都有控制信号控制，由定时信号确定控制信号何时发出、作用时间多长
- 什么是时序信号?
  - 同步系统中用于同步控制的定时信号
- 什么叫指令周期?
  - 取并执行一条指令的时间
  - 不同指令的指令周期可能不同
- 早期计算机的三级时序系统
  - 机器周期 - 节拍 - 脉冲
  - 指令周期可分为取指令、读操作数、执行并写结果等多个基本工作周期，称为机器周期。
  - 机器周期有取指令、存储器读、存储器写、中断响应等类型



现代计算机已不再采用三级时序系统，机器周期的概念已逐渐消失。整个数据通路中的定时信号就是时钟，一个时钟周期就是一个节拍。

数据通路与时序控制

现代计算机的时钟周期



数据通路由 “ … + 状态元件 + 操作元件(组合电路) + 状态元件 + …” 组成

只有状态元件能存储信息，所有操作元件都须从状态单元接收输入，并将输出写入状态单元中。其输入为前一时钟生成的数据，输出为当前时钟所用的数据

- 下降沿触发（负跳变）方式
  - 所有状态单元在下降沿写入信息，经过Latch Prop (clk-to-Q) 后输出有效
- Cycle Time = Latch Prop + Longest Delay Path + Setup + Clock Skew
- (Latch Prop + Shortest Delay Path - Clock Skew) > Hold Time

早期累加器型指令系统数据通路

- 最简单的数据通路结构
- 取指令数据路径为：

  PC→MAR,

  Read M,

  M→MBR→IBR→IR
- 取操作数、运算、送结果的数据路径为：

  操作数地址→MAR,

  Read M,

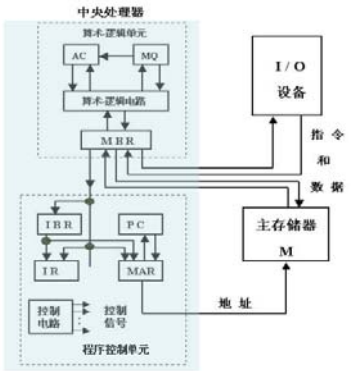
  M→MBR→ALU输入端,

  AC→ALU输入端,

  ALU操作,

  ALU结果→MBR,

  Write M



单总线数据通路

四种基本操作的时序

- 在通用寄存器之间传送数据 1Cycle?

  R0out，Yin
- 完成算术、逻辑运算 3Cycles?

  R1out，Yin

  R2out，Add，Zin

  Zout，R3in
- 从主存取字 R[R2]←M[R[R1]]

  R1out，MARin 3Cycles?

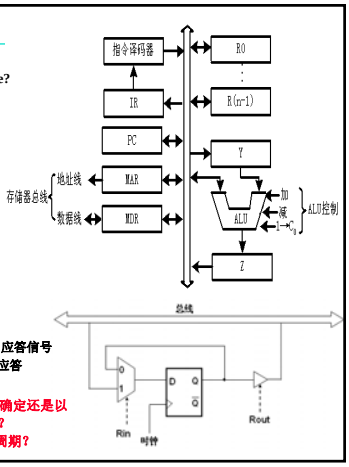
  Read, WMFC （等待MFC）

  MDRout，R2in
- 写字到主存 M[R[R1]] ← R[R2]

  R1out，MARin 3Cycles?

  R2out，MDRin，

  Write, WMFC



CPU访存有两种通信方式

早期：直接访问MM，“异步”方式，用MFC应答信号

现在：先Cache后MM，“同步”方式，无需应答

问题：时钟周期的宽度如何确定？

以“Riout,OP,Rjin”所花时间来确定还是以“Read/Write”所花时间来确定？

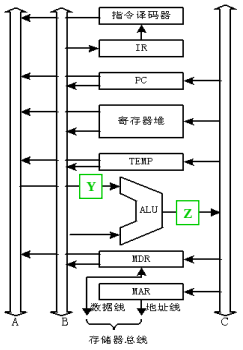
以上四种操作各需要几个时钟周期？

三总线数据通路

- 单总线中一个时钟内只允许传一个数据，因而指令执行效率很低
- 可采用多总线方式，同时在多个总线上传送不同数据，提高效率
- 例如：三总线数据通路
  - 总线A、B分别传送两个源操作数，总线C传送结果
  - 单总线中的暂存器Y和Z可取消，Why?
  - 采用双口寄存器
  - 如何实现： R[R3] ←R[R1] op R[R2]

    R1outA，R2outB，op，R3inC

    只要一个时钟周期即可



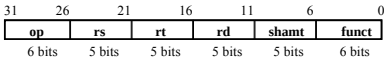
目前，大多数计算机都采用流水线方式执行指令，而上述单总线或三总线数据通路很难实现指令流水执行。

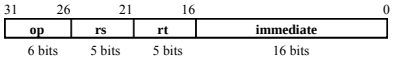
以下以MIPS指令系统为例介绍非总线式CPU的设计。

复习：MIPS的三种指令类型

大家记得是哪三种类型？ R-Type、I-Type、J-Type

- ADD and SUBSTRACT
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

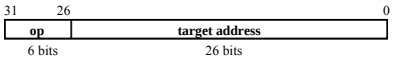




这些指令具有代表性！

有算术运算、逻辑运算；有RR型、RI型；
有访存指令；有条件转移、无条件转移。

本讲目标：实现以上7条指令的数据通路！



设计处理器的步骤

第一步：分析每条指令的功能，并用RTL(Register Transfer Language)来表示。

第二步：根据指令的功能给出所需的元件，并考虑如何将他们互连。

第三步：确定每个元件所需控制信号的取值。

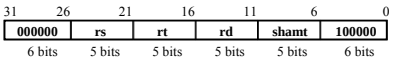
第四步：汇总所有指令所涉及到的控制信号，生成一张反映指令与控制信号之间关系的表。

第五步：根据表得到每个控制信号的逻辑表达式，据此设计控制器电路。

- 处理器设计涉及到数据通路的设计和控制器的设计
- 数据通路中有两种元件
  - 操作元件：由组合逻辑电路实现
  - 存储（状态）元件：由时序逻辑电路实现

RTL: The ADD Instruction（加法指令）

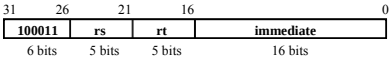
- add rd, rs, rt



- M[PC] 从PC所指的内存单元中取指令
- R[rd] ← R[rs] + R[rt] 从rs、rt 所指的寄存器中取数后相加，结果送rd 所指的寄存器中
- PC ← PC + 4 PC加4，使PC指向下一条指令

RTL: The Load Instruction（装入指令）

- lw rt, rs, imm16



- M[PC] (同加法指令)
- Addr ← R[rs] + SignExt(imm16)

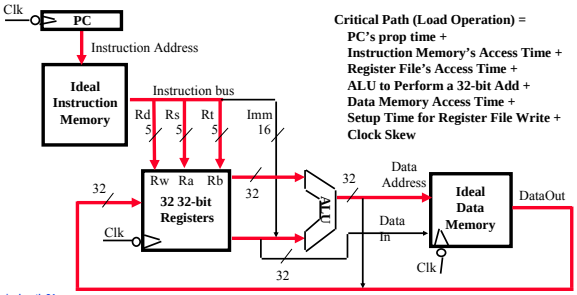
  计算数据地址 (立即数要进行符号扩展)
- R[rt] ← M[Addr] 从存储器中取出数据，装入到寄存器中
- PC ← PC + 4 (同加法指令)

BACK to design pro.

数据通路中的关键路径(Load操作)

- 记住：寄存器组和理想存储器的定时方式
  - 写操作时，作为时序逻辑电路。即：
    - 时钟到达前，输入需setup；到达后经“Clk to Q”，写入数据到达输出端
  - 读操作时，作为组合逻辑电路。即：
    - 地址有效后经过 “access time”，输出开始有效

Load操作：
R[Rt] ← M[(Rs)+Imm16]

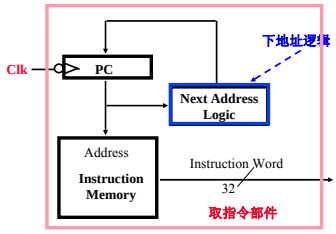


Critical Path (Load Operation) =
PC's prop time +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

取指令部件(Instruction Fetch Unit)

- 每条指令都有的公共操作：
  - 取指令： M[PC]
  - 更新PC：PC ← PC + 4

转移（Branch and Jump）时，PC内容再次被更新为 “转移目标地址”



顺序：先取指令，再改变PC的值（具体实现时，可以并行）

决不能先改变PC的值，再取指令

取指后，每条指令功能不同，在数据通路中信息流动过程也不同

下面分别对每条指令进行数据通路设计

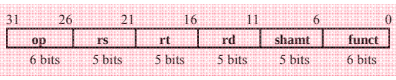
singlepath.25

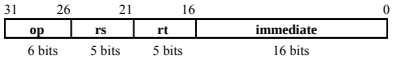
加法和减法指令(R-type类型）

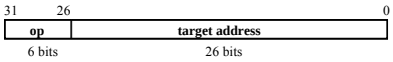
首先考虑add和sub指令（R-Type指令的代表）

实现目标（7条指令）：

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

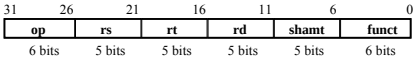






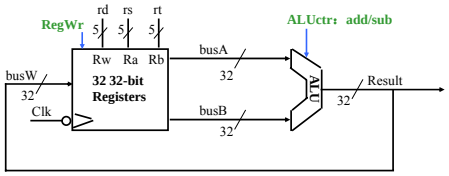
singlepath.26

RR（R-type）型指令的数据通路



- 功能：R[rd] ← R[rs] op R[rt]　　Example: add rd, rs, rt

不考虑公共操作，仅R-Type指令执行阶段的数据通路如下：



Ra, Rb, Rw 分别对应指令的rs, rt, rd

ALUctr 、RegWr：指令译码后产生的控制信号

指令“add rd, rs, rt”的控制信号应为？

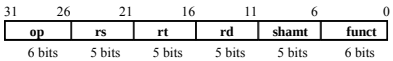
ALUctr=add，RegWr=1

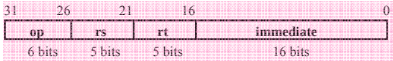
singlepath.27

带立即数的逻辑指令（ori指令）

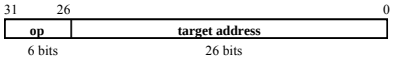
实现目标（7条指令）：

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



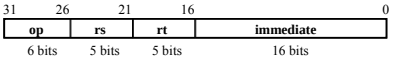


2. 考虑ori 指令（I-Type指令和逻辑运算指令的代表）



singlepath.28

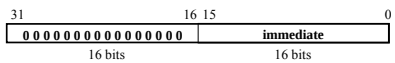
RTL: The OR Immediate Instruction



逻辑运算，立即数为逻辑数

- ori rt, rs, imm16
  - M[PC]　　取指令（公共操作，取指部件完成）
  - R[rt] ← R[rs] or ZeroExt(imm16)
    立即数零扩展，并与rs内容做“或”运算
  - PC ← PC + 4　　计算下地址（公共操作，取指部件完成）

零扩展 ZeroExt(imm16)：

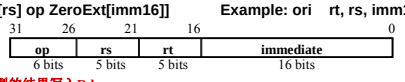


思考：应在前面数据通路上加哪些元件和连线？用什么控制信号来控制？

singlepath.29

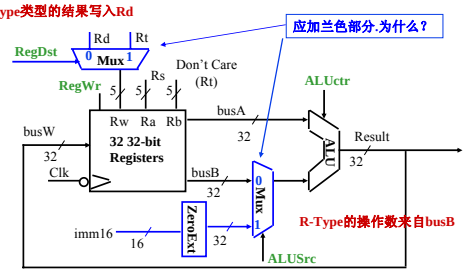
带立即数的逻辑指令的数据通路

- R[rt] ← R[rs] op ZeroExt[imm16]]　　Example: ori rt, rs, imm16



R-Type类型的结果写入Rd

应加兰色部分.为什么？



R-Type的操作数来自busB

Ori指令的控制信号：RegDst=？；RegWr=？；ALUctr=？；ALUSrc=？

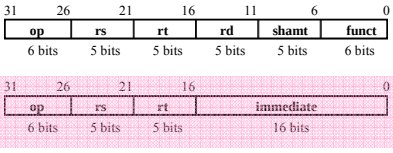
Ori指令的控制信号：RegDst=1；RegWr=1；ALUSrc=1；ALUctr=or

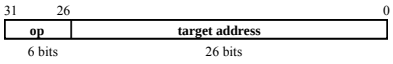
singlepath.30

访存指令中的数据装入指令 (lw)

实现目标（7条指令）：

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

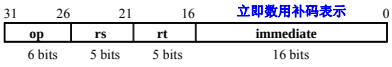




3. 考虑lw 指令（访存指令的代表）

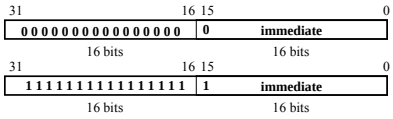
singlepath.31

RTL: The Load Instruction



- lw rt, rs, imm16
  - M[PC] 取指令（公共操作，取指部件完成）
  - Addr ← R[rs] + SignExt(imm16) 计算存储单元地址 (符号扩展！)
  - R[rt] ← M [Addr] 装入数据到寄存器rt中
  - PC ← PC + 4 计算下地址（公共操作，取指部件完成）

符号扩展(为什么不是零扩展？)：

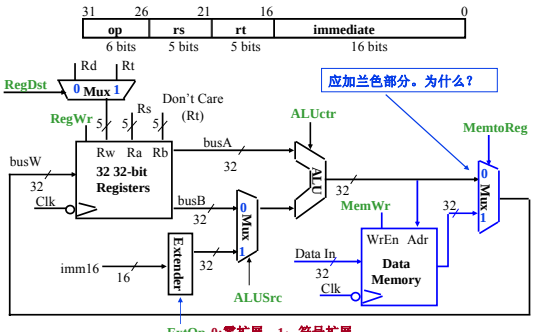


思考：应在原数据通路上加哪些元件和连线？用什么控制信号来控制？

singlepath.32

装入(lw)指令的数据通路

- R[rt] ← M[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16



ExtOp 0:零扩展，1：符号扩展

控制信号RegDst, RegWr, ALUctr, ExtOp, ALUSrc, MemWr, MemtoReg 各取何值？

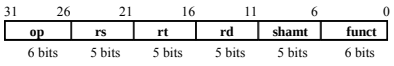
RegDst=1, RegWr=1, ALUctr=add, ExtOp=1, ALUSrc=1, MemWr=0, MemtoReg=1

singlepath.33

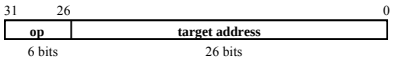
访存指令中的存数指令 (sw)

实现目标（7条指令）：

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

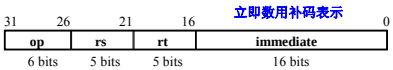


4. 考虑sw 指令（访存指令的代表）



singlepath.34

RTL: The Store Instruction



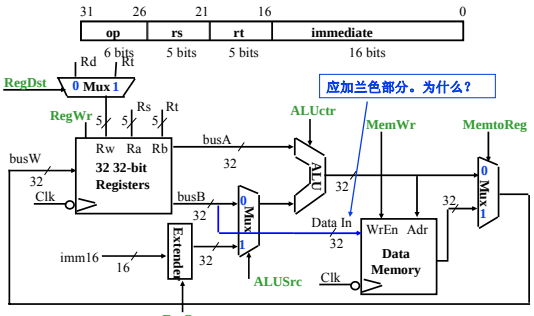
- sw rt, rs, imm16
  - M[PC] 取指令（公共操作，取指部件完成）
  - Addr ← R[rs] + SignExt(imm16) 计算存储单元地址（符号扩展！）
  - Mem[Addr] ← R[rt] 寄存器rt中的内容存到内存单元中
  - PC ← PC + 4 计算下地址（公共操作，取指部件完成）

思考：应在原数据通路上加哪些元件和连线？用什么控制信号来控制？

singlepath.35

存数(sw)指令的数据通路

- M[R[rs] + SignExt[imm16]] ← R[rt] Example: sw rt, rs, imm16



控制信号RegDst, RegWr, ALUctr, ExtOp, ALUSrc, MemWr, MemtoReg 各取何值？

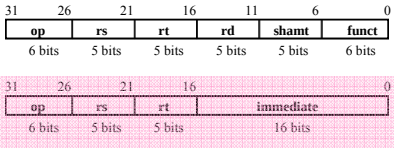
RegDst=x, RegWr=0, ALUctr=add, ExtOp=1, ALUSrc=1, MemWr=1, MemtoReg=x

singlepath.36

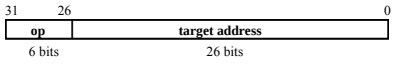
分支（条件转移）指令（相等转移：beq）

实现目标（7条指令）：

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

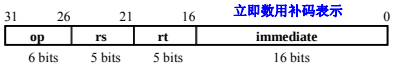


5. 考虑beq指令（条件转移指令的代表）



singlepath.37

RTL: The Branch Instruction



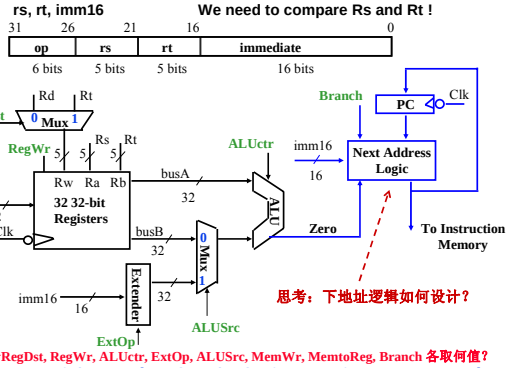
- beq rs, rt, imm16
  - M[PC] 取指令（公共操作，取指部件完成）
  - Cond ← R[rs] - R[rt] 做减法比较rs和rt中的内容
  - if (COND eq 0) 计算下地址（根据比较结果，修改PC）
    - PC ← PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC ← PC + 4

思考：立即数的含义是什么？是相对指令数还是相对单元数？
应在原数据通路上加哪些元件和连线？用什么控制信号来控制？

singlepath.38

条件转移指令的数据通路

- beq rs, rt, imm16 We need to compare Rs and Rt !



思考：下地址逻辑如何设计？

控制信号RegDst, RegWr, ALUctr, ExtOp, ALUSrc, MemWr, MemtoReg, Branch 各取何值？
RegDst=x, RegWr=0, ALUctr=sub, ExtOp=x, ALUSrc=0, MemWr=0, MemtoReg=x, Branch=1

singlepath.39

下地址计算逻辑的设计

PC是一个32位地址：

顺序执行时: PC<31:0> = PC<31:0> + 4

转移执行时: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4

采用32位PC时，可用左移2位实现“*4”操作，计算转移地址用2个加法器！

用更简便的方式实现如下：

MIPS按字节编址，每条指令为32位，占4个字节，故PC的值总是4的倍数，即后两位为00，因此，PC只需要30位即可。

PC采用30位后，其转移地址计算逻辑变得更加简单。

下地址计算逻辑简化为：

顺序执行时: PC<31:2> = PC<31:2> + 1

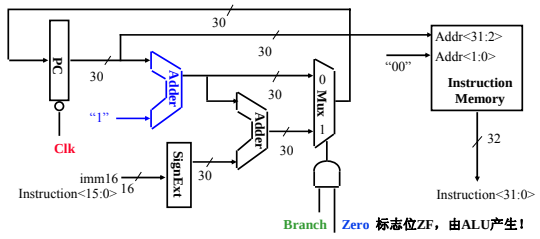
转移执行时: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]

取指令时: 指令地址 = PC<31:2> 串接 “00”

singlepath.40

下地址逻辑设计方案1: 快速但昂贵

- Using a 30-bit PC:
  - 顺序执行时: PC<31:2> = PC<31:2> + 1
  - 转移执行时: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - 取指令时: 指令地址 = PC<31:2> concat “00”



标志位ZF，由ALU产生！

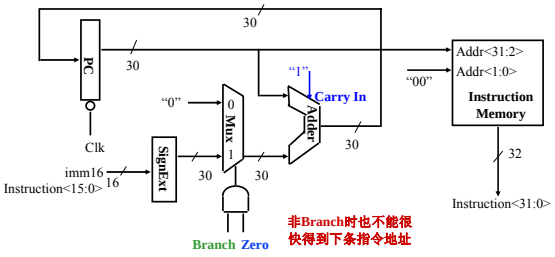
先根据当前PC取指令， 计算的下条指令地址在下一个时钟到来后才能写入PC！

为什么这里没有用“ALU”而是用“Adder”？ “ALU”和“Adder”有什么差别？

singlepath.41

下地址逻辑设计方案2: : 慢但便宜

- 为什么慢?
  - 只能等到“Zero”有值后才能进行地址计算
- 对性能有没有影响?
  - 没有，因为Load指令更慢。
- 为什么便宜?
  - “+1”操作用“进位”来实现，节省一个“Adder”



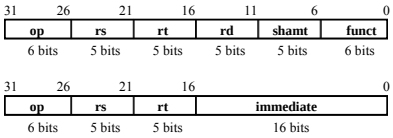
非Branch时也不能很快得到下条指令地址

singlepath.42

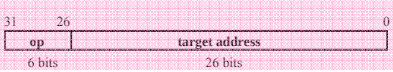
无条件转移指令

实现目标（7条指令）：

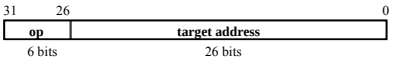
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



6. 考虑Jump指令（无条件转移指令的代表）



RTL: The Jump Instruction



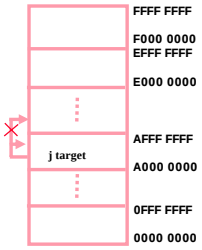
- j target
  - M[PC] 取指令（公共操作，取指部件完成）
  - PC<31:2> ← PC<31:28> 串接 target<25:0> 计算目标地址

想一想：跳转指令的转移范围有多大？

是当前指令后面的0x0000000~0xFFFFFFC 处？

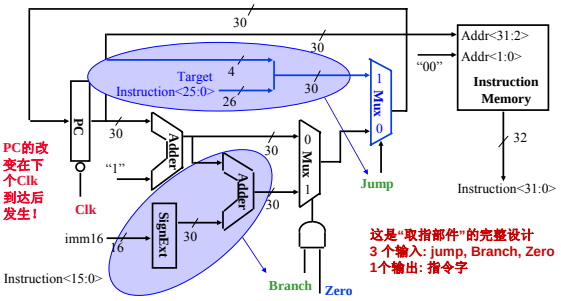
不对！它不是相对寻址，而是绝对寻址

思考：应在原数据通路上加哪些元件和连线？
用什么控制信号来控制？



Instruction Fetch Unit: 取指令部件

- j target
  - PC<31:2> ← PC<31:28> concat target<25:0>



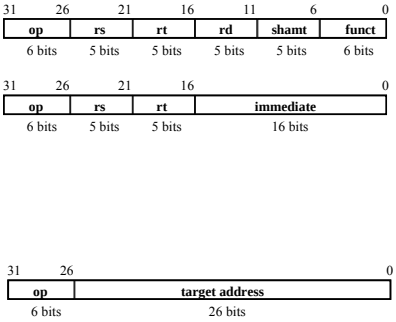
PC的改变在下个Clk到达后发生！

这是"取指部件"的完整设计
3 个输入: jump, Branch, Zero
1个输出: 指令字

RegDst, RegWr, ALUctr, ExtOp, ALUSrc, MemWr, MemtoReg, Branch, Jump 各取何值？
RegDst=ExtOp=ALUSrc=MemtoReg=ALUctr=x, RegWr=0, MemWr=0, Branch=0, Jump=1

The MIPS Subset(考察实现以下指令的数据通路)

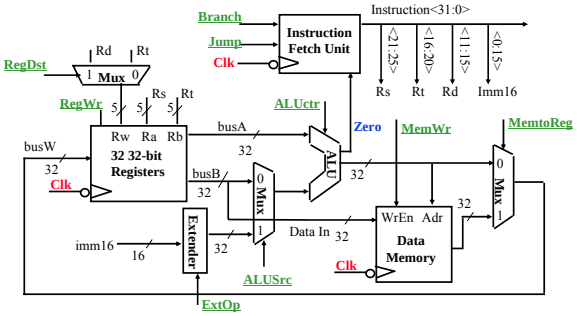
- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target



所有指令的数据通路都已经设计好，合起来的数据通路是什么样的？

Putting it All Together: A Single Cycle Datapath

已完成的每条指令所用数据通路（元件及其互连） 及其控制信号如下



指令执行结果总是在下个时钟到来时开始保存在 寄存器 或 存储器 或 PC 中！

下一讲考虑：如何产生控制信号！（这就是控制器的设计内容）

第一讲小结

- CPU设计直接决定了时钟周期宽度和CPI，所以对计算机性能非常重要！
- CPU主要由数据通路和控制器组成
  - 数据通路：实现指令集中所有指令的操作功能
  - 控制器：控制数据通路中各部件进行正确操作
- 数据通路中包含两种元件
  - 操作元件（组合电路）：ALU、MUX、Ext.、Adder、Reg/Mem Read等
  - 状态 / 存储元件（时序电路）：PC、Reg/Mem Write
- 数据通路的定时
  - 数据通路中的操作元件没有存储功能，其操作结果必须写到存储元件中
  - 在时钟到达后clk-to-Q时存储元件开始更新状态
- MIPS指令集的一个子集作为CPU的实现目标
  - 公共操作：取指令和PC+4
  - 下址计算：30位PC，三路选择：顺序、Branch（结合标志Zero）、Jump
  - R型：ALU两个操作数来自rs和rt，结果写到rd
  - 访存：符号扩展，数据在rt和主存单元中交换
  - 立即数：0扩展后的操作数送到ALU的一个输入端

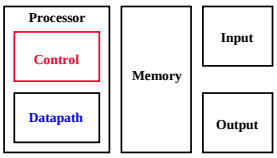
第二讲 单周期控制器的设计

主 要 内 容

- 考察每条指令在数据通路中的执行过程和设计到的控制信号的取值
  - 公共操作：取指令和计算下址PC
  - R-Type指令（add / sub）
  - 立即数指令 （ori）
  - 访存指令（lw / sw）
  - 分支指令 （beq）
  - 跳转指令 （j）
- 汇总各指令的控制信号取值
  - 分两类控制信号：直接送往数据通路 / 送往局部控制单元
- 分析ALU操作对应的控制信号与func字段之间的关系
- 设计ALU局部控制单元
- 设计主控制单元

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

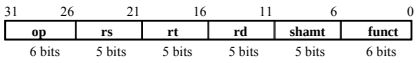


- 下一个目标：设计单周期数据通路的控制器。

设计方法：

1) 根据每条指令的功能，分析控制信号的取值，并在表中列出。
2) 根据列出的指令和控制信号的关系，写出每个控制信号的逻辑表达式。

ADD / SUB 指令



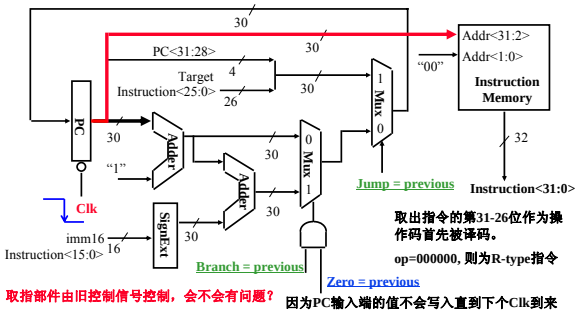
- add rd, rs, rt
  - M[PC] 取指（每条指令一样）
  - R[rd] ← R[rs] + R[rt] 实际操作（每条指令可能不同）
  - PC ← PC + 4 计算顺序执行时PC的值（每条指令一样）

Add / Sub操作开始时取指部件中的动作

- 取指令: Instruction ← M[PC]
  - 所有指令都相同

新指令还没有取出译码，所以控制信号的值还是原来指令的旧值。

新指令还没有执行，所以标志也为旧值。



取出指令的第31-26位作为操作码首先被译码。

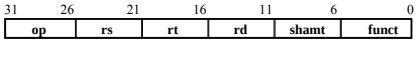
op=000000, 则为R-type指令

取指部件由旧控制信号控制，会不会有问题？

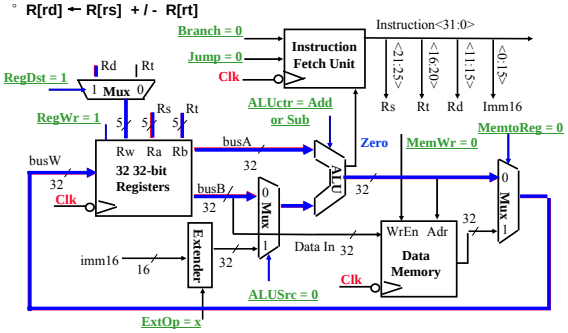
没有问题！ Why?

因为PC输入端的值不会写入直到下个Clk到来只要保证下个Clk来之前能产生正确的PC即可！

指令译码后R型指令（Add / Sub）操作过程

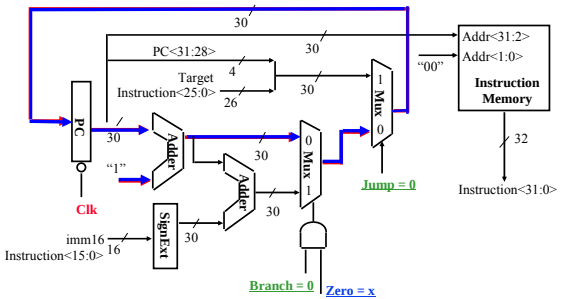


- R[rd] ← R[rs] + / - R[rt]



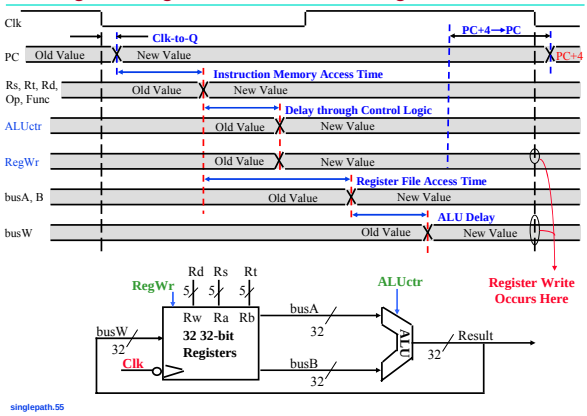
R型指令（Add /Sub）最后阶段取指部件中的动作

- PC ← PC + 4
  - 除 Branch and Jump以外的指令都相同



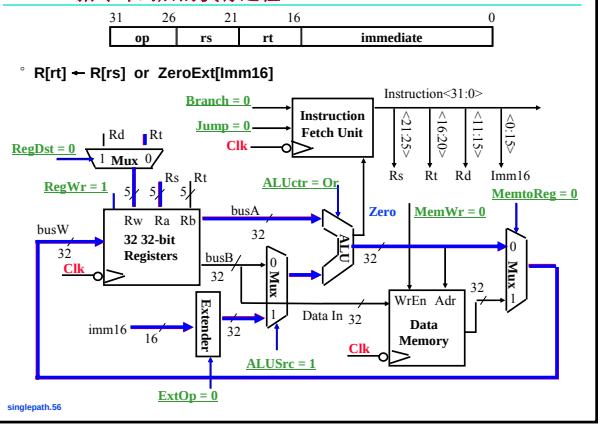
因为新的控制信号保证了正确的PC值的产生，在足够长的时间后，下个时钟Clk到来！

Register-Register（R型指令） Timing



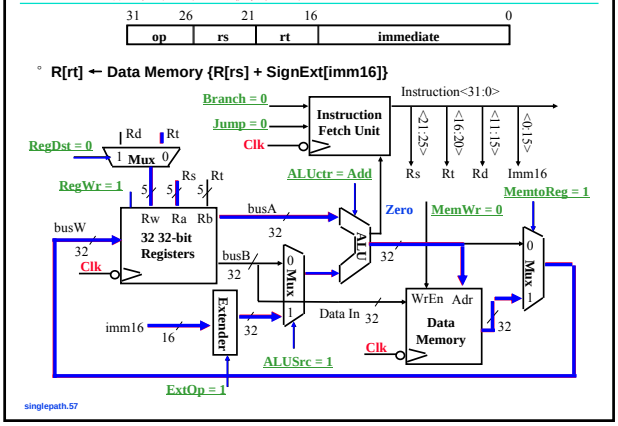
singlepath.55

ori 指令译码后的执行过程



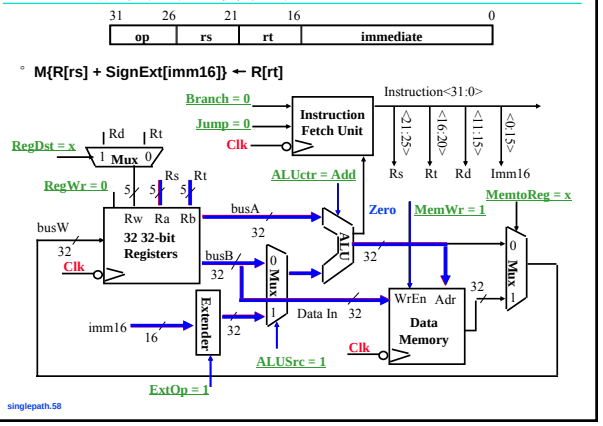
singlepath.56

Load指令译码后的执行过程



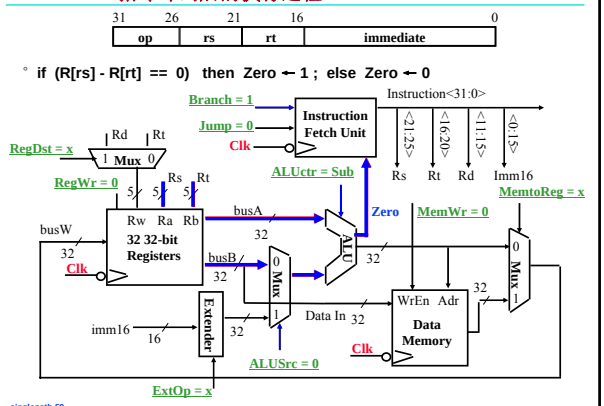
singlepath.57

Store指令译码后的执行过程



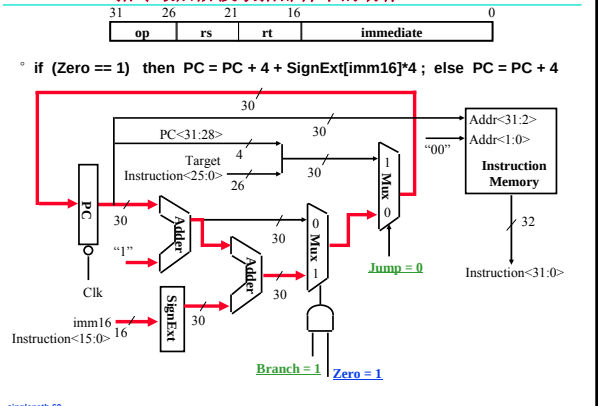
singlepath.58

Branch指令译码后的执行过程



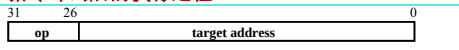
singlepath.59

Branch指令最后阶段取指部件中的动作



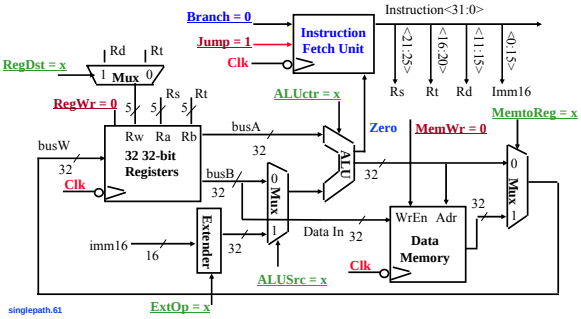
singlepath.60

Jump指令译码后的执行过程



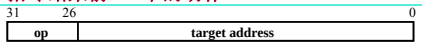
◦ IFU中目标地址送PC，其他什么都不做（只要保证存储部件不发生写的动作）

如何保证存储部件不发生写？

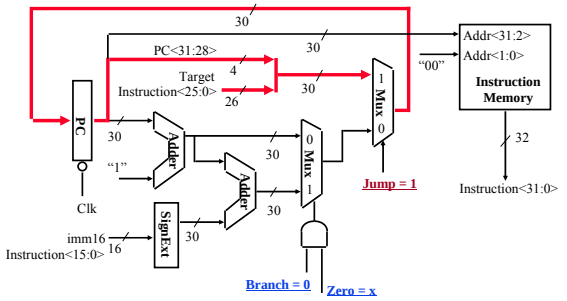


singlepath.61

Jump指令结束前IFU中的动作



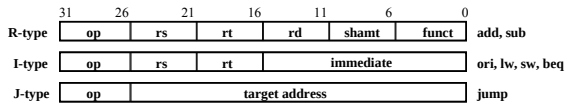
◦ PC ← PC<31:29> concat target<25:0> concat “00”



singlepath.62

综合分析结果，得到如下指令与控制信号的关系表

func	10 0000	10 0010	We Don't Care :-)				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

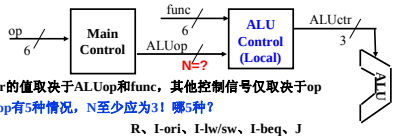


singlepath.63

主控制单元和ALU局部控制单元

MIPS指令格式中指示操作性质的字段有两个：op(主控) 和 func（ALU局控）。

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUctr	Add/Subtract	Or	Add	Add	Subtract	xxx



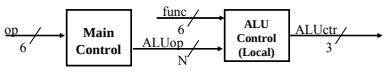
ALUctr的值取决于ALUop和func，其他控制信号仅取决于op

ALUop有5种情况，N至少应为3！哪5种？

R、I-ori、I-lw/sw、I-beq、J

singlepath.64

ALUop和“func”字段的译码

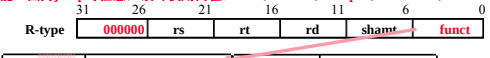


ALUop的编码定义如下： 书P182表6.3中ori和beq只能有一个x，否则编码冲突！

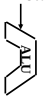
	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 xx	0 10	0 00	0 00	0x1	xxx

问题：ALUop能否仅用2位？ R-Type取1xx，不会发生编码冲突！

能！因为jump时任意，故可仅用两位：R:11, I-ori:10, I-beq:01, I-lw/sw:00, J-xx



funct<5:0>	Instruction Operation	ALUctr<2:0>	ALU Operation
10 0000	add	000	Add
10 0010	subtract	001	Subtract
10 0100	and	100	And
10 0101	or	101	Or
10 1010	set-on-less-than	010	Subtract



singlepath.65

ALUctr与func后4位有关，需建立ALUctr 与ALUop和func后四位之间对应关系

ALUctr控制信号的真值表

建立ALUop、func后4位和ALUctr之间的关系表

由关系表可得出ALUctr的逻辑表达式：

ALUctr[i] = f (ALUop[i], func[i])

ALUop (Symbolic)	R-type	ori	lw	sw	beq
	“R-type”	Or	Add	Add	Subtract
ALUop<2:0>	1 00	0 10	0 00	0 00	0 x1

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop bit2	ALUop bit1	ALUop bit0	func bit<3>	func bit<2>	func bit<1>	func bit<0>	ALU Operation	ALUctr bit<2>	ALUctr bit<1>	ALUctr bit<0>
0	0	0	x	x	x	x	Add	0	0	0
0	x	1	x	x	x	x	Subtract	0	0	1
0	1	0	x	x	x	x	Or	1	1	0
1	x	x	0	0	0	0	Add	0	0	0
1	x	x	0	0	1	0	Subtract	0	0	1
1	x	x	0	1	0	0	And	0	1	0
1	x	x	0	1	0	1	Or	1	1	0
1	x	x	1	0	1	0	Subtract	0	0	1

头三行是非R-Type，操作由ALUop决定，与func无关。R-Type时，操作完全由func决定。

singlepath.66 ALUctr可用更多位数，这样便于扩充，例如，可加入异或、移位等操作。

The Logic Equation for ALUctr<0>

从前面的真值表中，抽取出ALUctr[0]为1的行

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes func<3> a don't care

° ALUctr<0> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<2> & func<1> & !func<0>

The Logic Equation for ALUctr<1>

从前面的真值表中，抽取出ALUctr[1]为1的行

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	1	0	x	x	x	x	1
1	x	x	0	1	0	0	1
1	x	x	0	1	0	1	1

° ALUctr<1> = !ALUop<2> & ALUop<1> & ! ALUop<0> +
ALUop<2> & !func<3> & func<2> & !func<1>

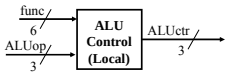
The Logic Equation for ALUctr<2>

从前面的真值表中，抽取出ALUctr[2]为1的行

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
0	1	0	x	x	x	x	1
1	x	x	0	1	0	1	1

° ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0>
+ ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

局部ALU控制单元逻辑



总结前面的结果，得到：

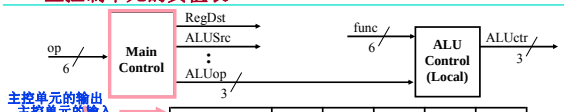
° ALUctr<0> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<2> & func<1> & !func<0>

° ALUctr<1> = !ALUop<2> & ALUop<1> & !ALUop<0> +
ALUop<2> & !func<3> & func<2> & !func<1>

° ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0> +
ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

根据以上逻辑方程，可实现局部ALU控制单元！

主控制单元的真值表

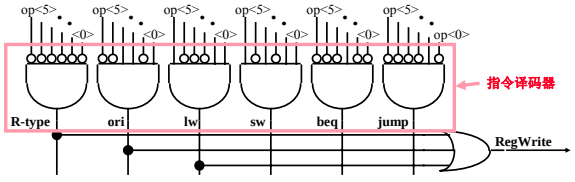


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	x	1	0	0	x	x
ALUop <0>	x	0	0	0	1	x

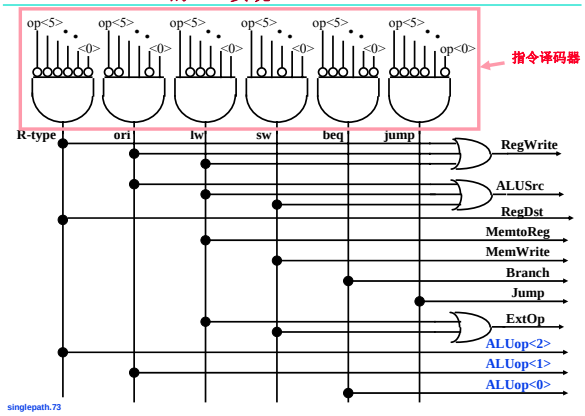
考察每个控制信号的逻辑方程（如：RegWrite）

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

° RegWrite = R-type + ori + lw
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)

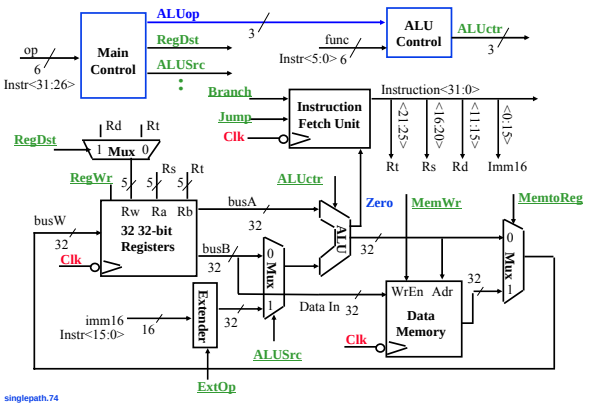


Main Control的PLA实现



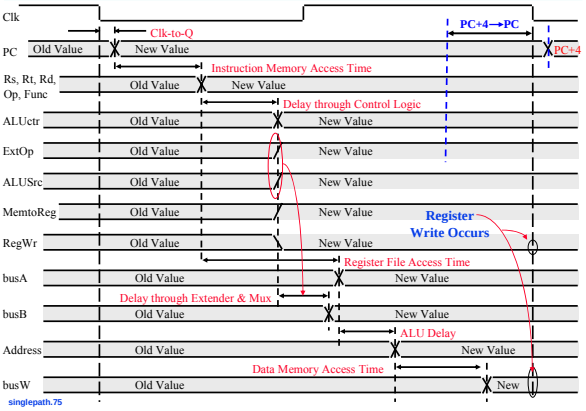
singlepath.73

执行前述7条指令的完整的单周期处理器



singlepath.74

lw指令的执行时间最长, 它所花时间作为时钟周期



singlepath.75

单周期计算机的性能

- 单周期处理器的CPI为多少？ CPI=1!
  - 其他条件一定的情况下，CPI越小，则性能越好！
  - CPI=1，不是很好吗？
- 单周期处理器的性能会不会很好？为什么？
  - 计算机的性能除CPI外，还取决于时钟周期的宽度
  - 单周期处理器的时钟宽度为最复杂指令的执行时间
  - 很多指令可以在更短的时间内完成

单周期计算机的性能

假设在单周期处理器中，各主要功能单元的操作时间为：

- 存储单元：200ps
- ALU和加法器：100ps
- 寄存器堆（读/写）：50ps

假设MUX、控制单元、PC、扩展器和传输线路都没有延迟，则下面实现方式中，哪个更快？快多少？

（1）每条指令在一个固定长度的时钟周期内完成

（2）每条指令在一个时钟周期内完成，但时钟周期仅为指令所需，是可变的（实际不可行，只是为了比较）

假设指令组成为：25%取数、10%存数、45%ALU、15%分支、5%跳转

singlepath.76

单周期计算机的性能

解：CPU执行时间=指令条数 x CPI x 时钟周期=指令条数 x 时钟周期

两种方案的指令条数都一样，CPI都为1，所以只要比较时钟周期宽度即可。

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

各指令类型要求的时间长度为：

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Singlepath.77

单周期计算机的性能

对于方式（1），时钟周期由最长指令来决定，应该是load指令，为600ps

对于方式（2），时钟周期取各条指令所需时间，时钟周期从600ps至200ps

根据各类指令的频度，计算出平均时钟周期长度为：

CPU时钟周期=600x25%+550x10%+400x45%+350x15%+200x5%=447.5ps

$$\text{CPU性能比}=\frac{\text{方式(1)的CPU执行时间}}{\text{方式(2)的CPU执行时间}}=\frac{\text{方式(1)的CPU 时钟周期}}{\text{方式(2)的CPU 时钟周期}}=\frac{600}{447.5}=1.34$$

由此可见，可变时钟周期的性能是定长周期的1.34倍！

但是，对每类指令采用可变长时钟周期实现非常困难，而且所带来的额外开销会很大，不合算！

早期的小指令集计算机用过单周期实现技术，但现代计算机都不采用。

下一讲介绍多周期数据通路和控制器，其特点是：

时钟周期固定、时钟周期数可变

singlepath.78

第二讲 小结

- 考察每条指令在单周期数据通路中的执行过程
  - 每条指令在一个时钟周期内完成
  - 每个时钟到来时，都开始进入取指令操作
    - 经过clk-to-Q，PC得到新值，经过access time后得到当前指令
    - 按三种方式分别计算下条指令地址，在branch / zero / jump的控制下，选择其中之一送到PC输入端，但不会马上写到PC中，一直到下个时钟到达时，才会更新PC。三种下址方式为：
      - branch=jump=0：PC+4
      - branch=zero=1； PC+4+signExt[imm16]*4
      - jump=1； PC<31:28> concat target<25:0> concat “00”
  - 指令取出后被译码，产生指令对应的控制信号
    - R-type指令：rd为目的寄存器，无访存操作，……
    - ori指令：rt为目的寄存器，0扩展，无访存操作，……
    - lw指令： rt为目的寄存器，计算地址、符号扩展，读内存，……
    - sw指令： rt为源寄存器，计算地址、符号扩展，写内存，……
- 汇总每条指令控制信号的取值，生成真值表，写出逻辑表达式，设计主控制逻辑和ALU局部控制逻辑

第三讲 多周期处理器的设计

主 要 内 容

- 多周期数据通路实现思想
- 单周期数据通路和多周期数据通路的差别
  - 通过简要分析LOAD指令分阶段执行过程，以加深理解单周期和多周期数据通路的差别
- 多周期通路中存储单元的“竞争”问题及其解决思路
- 详细分析7条指令在多周期通路中的执行过程
- 在分析执行过程基础上，分析每个周期内控制信号的取值，生成相应的状态
- 综合生成所有指令的状态转换图
- 根据状态转换图，生成控制器输出的逻辑表达式
- 根据逻辑表达式，用PLA(硬布线)实现控制逻辑

Drawback of Single Cycle Processor

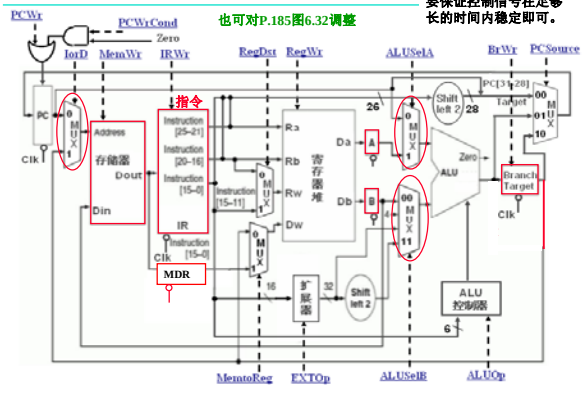
- 单周期处理器的CPI为1，所有指令的执行时间都以最长的load指令为准
- 最长指令时间为: Cycle time must be long enough for the load instruction
  - PC's Clock -to-Q +
  - Instruction Memory Access Time +
  - Register File Access Time +
  - ALU Delay (address calculation) +
  - Data Memory Access Time +
  - Register File Setup Time +
  - Clock Skew
- 时钟周期远远大于其他指令实际所需的执行时间，效率极低
  - R-type指令、立即数运算指令不需要读内存
  - Store指令不需要写寄存器
  - 分支指令不需要访问内存和写寄存器
  - Jump 不需要ALU运算，不需要读内存，也不需要读/写寄存器

多周期处理器的实现思想

- 单周期处理器的问题根源:
  - 时钟周期以最复杂指令所需时间为准，太长！
- 解决思路:
  - 把指令的执行分成多个阶段，每个阶段在一个时钟周期内完成
    - 时钟周期以最复杂阶段所花时间为准
    - 尽量分成大致相等的若干阶段
    - 规定每个阶段内最多只能完成：1次访存 或 1次寄存器堆读/写 或 1次ALU
  - 每步都设置相应的存储元件，每部执行结果都在下个时钟开始保存到相应单元
- 多周期处理器的好处:
  - 时钟周期短
  - 不同指令所用周期数可以不同，如：
    - Load： five cycles
    - Jump： three cycles
  - 允许功能部件在一条指令执行过程中被重复使用。如：
    - Adder + ALU（多周期时只用一个ALU，在不同周期可重复使用）
    - Inst. / Data mem（多周期时合用一个存储器，不同周期中重复使用）

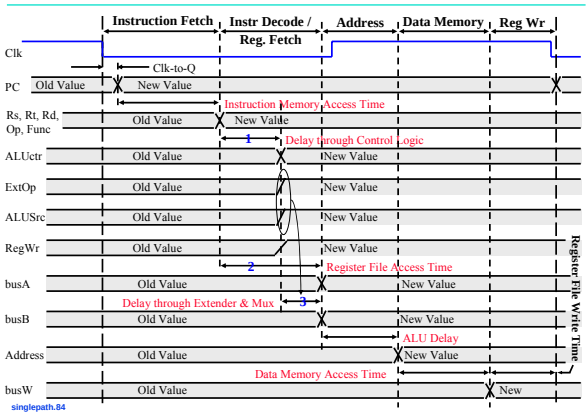
多周期数据通路

调整前的也没问题，只要保证控制信号在足够长的时间内稳定即可。



只有一个ALU、一个Memory、多处增加MUX和临时寄存器

Load指令分成5个阶段



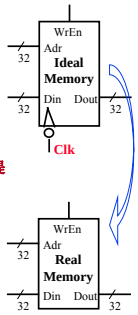
Load指令各阶段分析

- 取指令阶段
  - 执行一次存储器读操作
  - 读出的内容（指令）保存到寄存器IR（指令寄存器）中
  - IR的内容不是每个时钟都更新，所以IR必须加一个“写使能”控制
  - 在取指令阶段结束时，ALU的输出为PC+4，并送到PC的输入端，但不能在每个时钟到来时就更新PC，所以PC也要有“写使能”控制
- 译码/读寄存器堆阶段
  - 经过控制逻辑延迟后，控制信号更新为新值
  - 执行一次寄存器读操作
  - 读出的内容（操作数）保存到临时寄存器A和B中
  - 每个时钟到来时，A和B中的值都要更新，所以不需“写使能”控制
  - 对16位立即数进行符号扩展后，送到ALU的B口的多路选择器
- 地址生成阶段（ALU运算）
  - ALU的A口和B口的多路选择器在相应控制信号控制下选择操作数进行加法运算，输出结果在下个时钟到达时，保存到临时寄存器BranchTarget (ALUout)中
- 读存储器阶段
  - 由ALUout作为地址访问存储器，读出数据，保存在临时寄存器MDR中
- 写结果到寄存器
  - 把MDR中的内容写到寄存器堆中

寄存器堆和存储器的写定时（ Ideal vs. Reality）

- 单周期机器中，寄存器组和存储器被简化为理想的有时钟控制的：
  - 时钟边沿到来时，才进行写
  - 时钟边沿到来之前，地址、数据和写使能都已经稳定
- 实际机器中，寄存器组和存储器的情况为：
  - 都没有时钟输入
  - 写操作不是由时钟边沿触发，而是一个组合电路，其过程为：
    - 写使能(Write Enable)为 1，并且 Din信号已稳定的前提下，经过Write Access时间，Din信号被写入 Adr 处
  - 重要之处：地址和数据必须在写使能为1前稳定

因此，存在地址Adr、数据Din和写使能WrEn信号的“竞争”问题！



竞争（race）问题

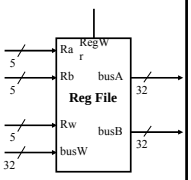
- Register File(寄存器组)：

实际寄存器组(不带Clk的)在单周期通路中不能可靠工作

这是因为：

- 不能保证 Rw 在RegWr = 1之前稳定

即：在 Rw 和RegWr (write enable)之间存在“race”



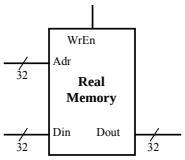
- Memory(存储器)：

实际存储器在单周期通路中也不能可靠工作

这是因为：

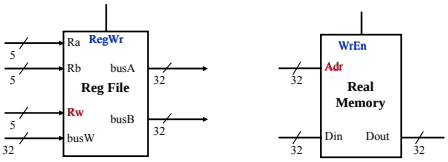
- 不能保证 Adr在WrEn = 1之前稳定

即：在 Adr 和 WrEn之间存在“race”



如何在多周期通路中避免“race”问题

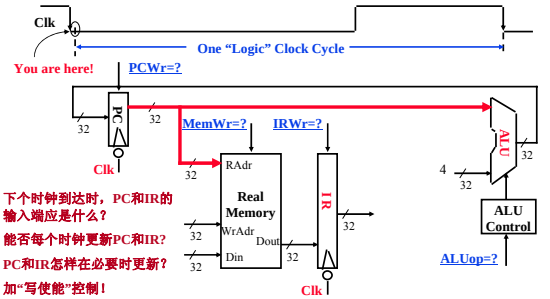
- 多时钟周期中解决“竞争”问题的方案
  - 确认地址和数据在第N周期结束时已稳定
  - 使写使能信号在一个周期后(即：第N+1周期)有效
  - 在写使能信号无效前地址和数据不改变



“Race”问题有时会导致机器神秘出错，甚至崩溃！

取指周期（取指令、计算下地址）开始时

- 在一个时钟到来的下降沿开始取指令周期的任务：
  - M[PC] ； PC ← PC + 4



下个时钟到达时，PC和IR的输入端应是什么？

能否每个时钟更新PC和IR？

PC和IR怎样在必要时更新？

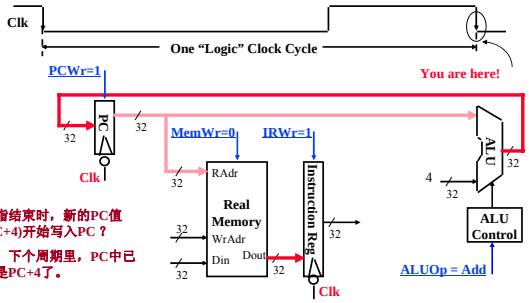
加“写使能”控制！

控制信号PCWr=?, MemWr=?, IRWr=?, ALUop=?

控制信号PCWr=1, MemWr=0, IRWr=1, ALUop=add

取指周期结束时

- 每一个周期都在下一个时钟到来时结束 (此时，存储元件被更新):
  - IR ← M[PC]　　PC ← PC + 4



取指结束时，新的PC值(PC+4)开始写入PC ？

即：下个周期里，PC中已经是PC+4了。

取指结束时，当前指令开始写入IR ！为保证本指令期间IR中指令不变，后面周期中IRWr应该为0

考察整个取指周期（第一个周期）

分析：取指周期中各控制信号的取值应为？

想想看，和单周期有哪些不同？

PC的更新时间

PC需要写使能信号（不是每个周期都写）

多了一个指令寄存器IR

每个周期产生各自的控制信号

• • • • • •

为什么多周期时需要PCWr？

Ifetch

ALUOp=Add
1: PCWr, IRWr
x: PCWrCond
RegDst, Mem2R
Others: 0s

PCWr=1 PCWrCond=x Zero

IorD=0 MemWr=0 IRWr=1

PCSrc=0 BrWr=0

ALUSelA=0

Target

PC

Real Memory

Instruction Reg

busA busB

ALU

Zero

ALU Control

ALUSelB=01

ALUOp=Add

分析的结果就是生成的一个“状态”

singlepath.91

寄存器取 / 指令译码周期（第二个周期）

- busA ← RegFile[rs] ; busB ← RegFile[rt] ;
- Decoder ← Op and Func;
- ALU is not being used: ALUctr = xx

指令未译码，故只执行公共操作

ALU空闲，可用ALU“投机计算”转移地址！

PCWr=0 PCWrCond=0 Zero

IorD=x MemWr=0 IRWr=0 RegDst=x RegWr=0 ALUSelA=x

PCSrc=x

PC

Real Memory

Instruction Reg

Reg File

ALU

Zero

ALU Control

Go to the Control

ALUSelB=xx

ALUOp=xx

问题：PC中已是下条顺序指令的地址，对本条指令的执行有没有影响？

没有影响，因为IRWr=0!

考虑转移地址的投机计算的数据通路如何？

singlepath.92

寄存器取 / 指令译码周期（第二个周期）

- busA ← Reg[rs] ; busB ← Reg[rt] ;
- Decoder ← Op and Func;
- 投机：Target ← PC + SignExt(Imm16)*4

（为什么不是PC +4+ SignExt(Imm16)*4？）

Rfetch/Decode

ALUOp=Add
1: BrWr, ExtOp
ALUSelB=10
x: RegDst, PCSrc
IorD, MemtoReg
Others: 0s

为什么不直接送PC？为什么加BrWr？

PCWr=0 PCWrCond=0 Zero

IorD=x MemWr=0 IRWr=0 RegDst=x RegWr=0 ALUSelA=0

PCSrc=x BrWr=1 Target

PC

Real Memory

Instruction Reg

Reg File

ALU

Zero

ALU Control

Beq Rtype Ori Memory

Control 指令译码器

Extend

<< 2

ALUSelB=10

ALUOp=Add

ExtOp=1

第二周期结束时，执行的结果是什么？

singlepath.93

寄存器取 / 指令译码周期（第二个周期）

PCWr=0 PCWrCond=0 Zero

IorD=x MemWr=0 IRWr=0 RegDst=x RegWr=0 ALUSelA=0

PCSrc=x BrWr=1 Target

PC

Real Memory

Instruction Reg

Reg File

ALU

Zero

ALU Control

Rtype Beq Ori Memory

Control

Extend

<< 2

ALUSelB=10

ALUOp=Add

ExtOp=1

如果指令译码输出为：Beq

下面第三个周期就是Beq指令的第一个执行周期！

singlepath.94

Branch指令执行并完成周期（第三个周期）

如果指令译码输出为：Branch

- if (busA == busB)
  - PC ← Target

若不“投机”，则在此周期前还要加一个周期，用来计算转移地址后保存到Target中！

控制信号的取值是什么？

BrFinish

ALUOp=Sub
ALUSelB=01
x: IorD, Mem2Reg
RegDst, ExtOp
1: PCWrCond
ALUSelA
PCSrc

PCWr=0 PCWrCond=1 Zero

IorD=x MemWr=0 IRWr=0 RegDst=x RegWr=0 ALUSelA=1

PCSrc=1 BrWr=0 Target

PC

Real Memory

Instruction Reg

Reg File

ALU

Zero

ALU Control

Extend

<< 2

ALUSelB=01

ALUOp=Sub

ExtOp=x

PC中是否在下个周期更新为Target，则由Zero决定！

“ALU”每步都不空闲，被重复使用

singlepath.95

寄存器取 / 指令译码周期（第二个周期）

PCWr=0 PCWrCond=0 Zero

IorD=x MemWr=0 IRWr=0 RegDst=x RegWr=0 ALUSelA=0

PCSrc=x BrWr=1 Target

PC

Real Memory

Instruction Reg

Reg File

ALU

Zero

ALU Control

Beq Rtype Ori Memory

Control

Extend

<< 2

ALUSelB=10

ALUOp=Add

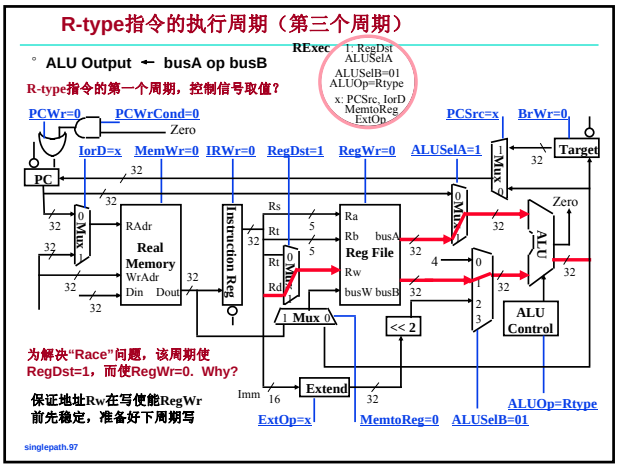
ExtOp=1

如果指令译码输出为：R-Type

下面第三个周期就是R-Type指令的第一个执行周期！

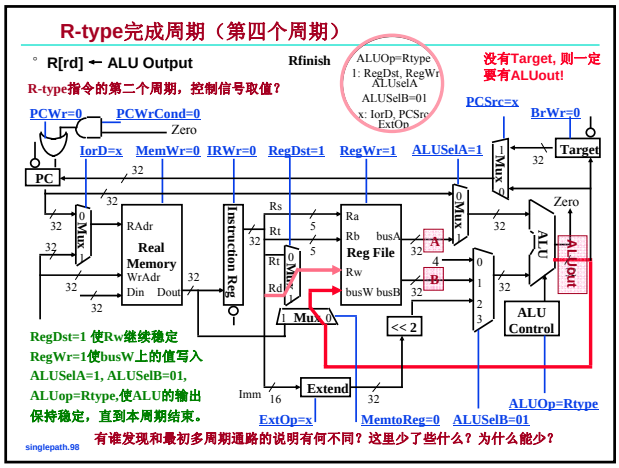
singlepath.96

R-type指令的执行周期（第三个周期）



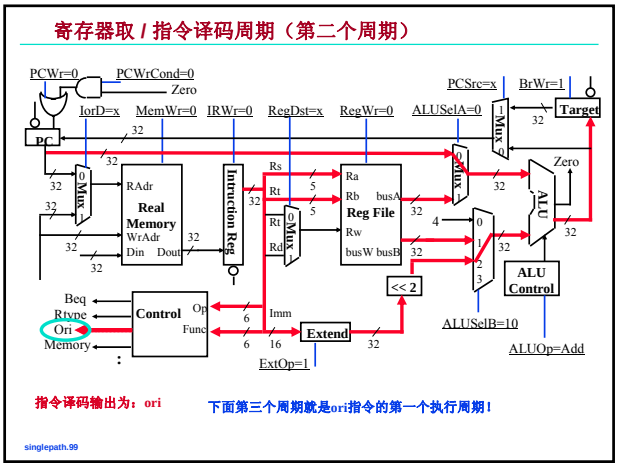
singlepath.97

R-type完成周期（第四个周期）



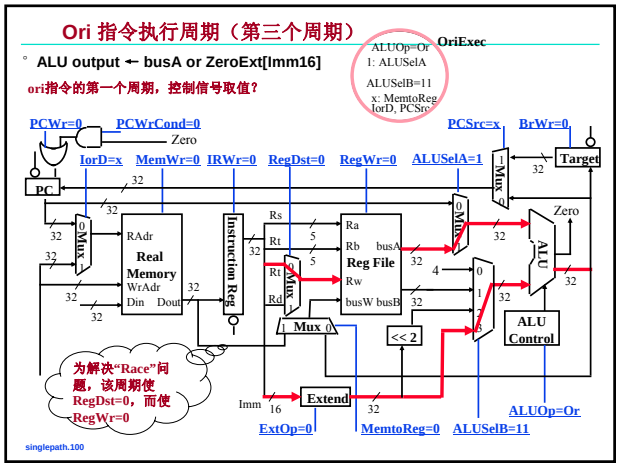
singlepath.98

寄存器取 / 指令译码周期（第二个周期）



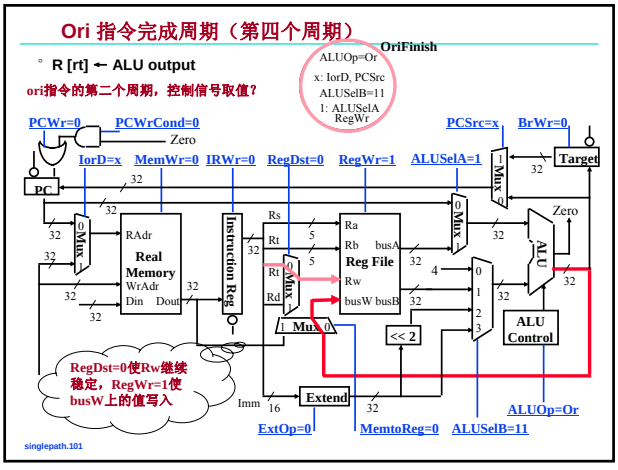
singlepath.99

Ori 指令执行周期（第三个周期）



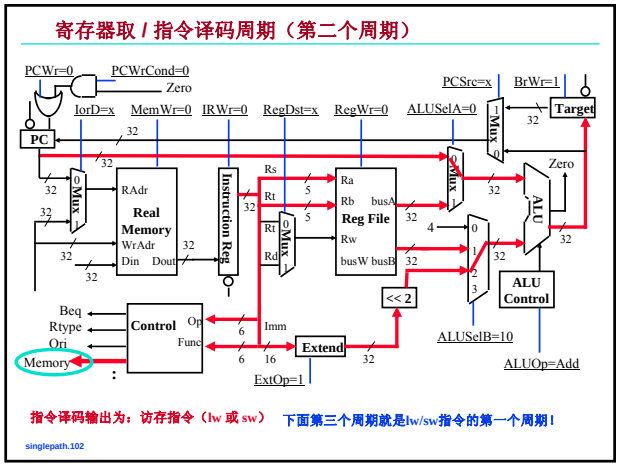
singlepath.100

Ori 指令完成周期（第四个周期）



singlepath.101

寄存器取 / 指令译码周期（第二个周期）

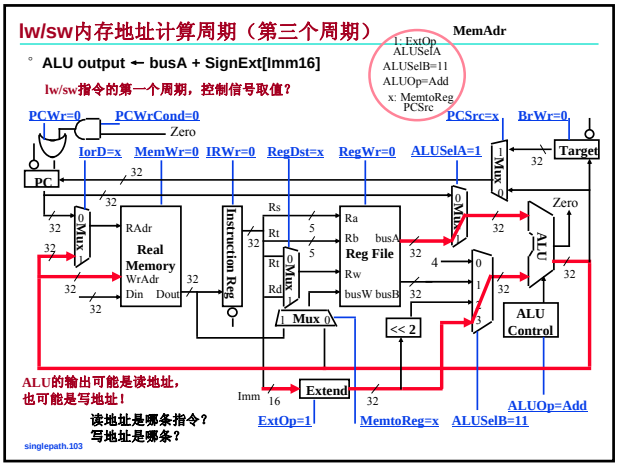


singlepath.102

lw/sw内存地址计算周期（第三个周期）

- ALU output ← busA + SignExt[Imm16]

lw/sw指令的第一个周期，控制信号取值？



ALU的输出可能是读地址，也可能是写地址！

读地址是哪条指令？

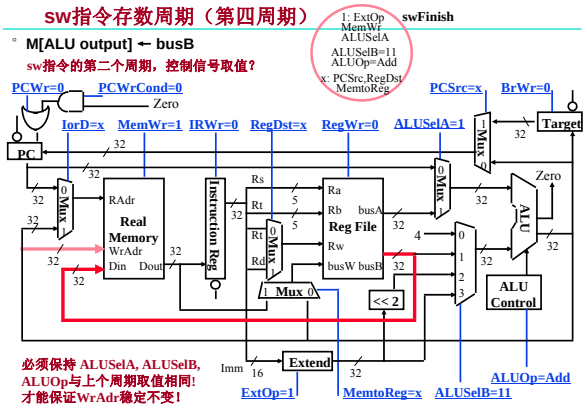
写地址是哪条？

singlepath.103

sw指令存数周期（第四周期）

- M[ALU output] ← busB

sw指令的第二个周期，控制信号取值？



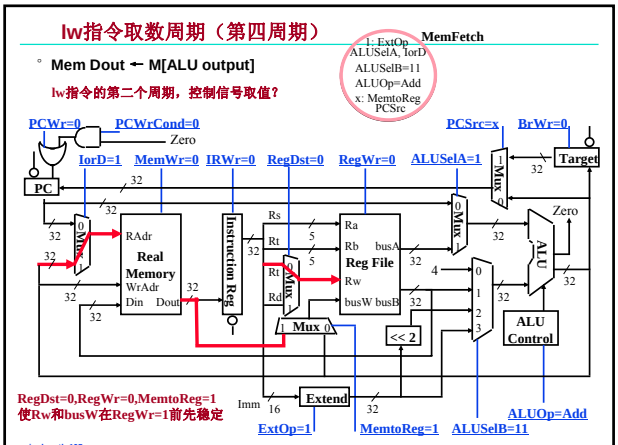
必须保持 ALUSelA, ALUSelB, ALUOp与上个周期取值相同！才能保证WrAdr稳定不变！

singlepath.104

lw指令取数周期（第四周期）

- Mem Dout ← M[ALU output]

lw指令的第二个周期，控制信号取值？



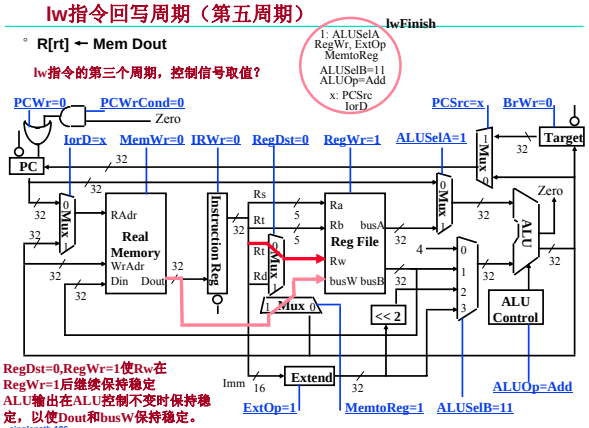
RegDst=0,RegWr=0,MemtoReg=1
使Rw和busW在RegWr=1前先稳定

singlepath.105

lw指令回写周期（第五周期）

- R[rt] ← Mem Dout

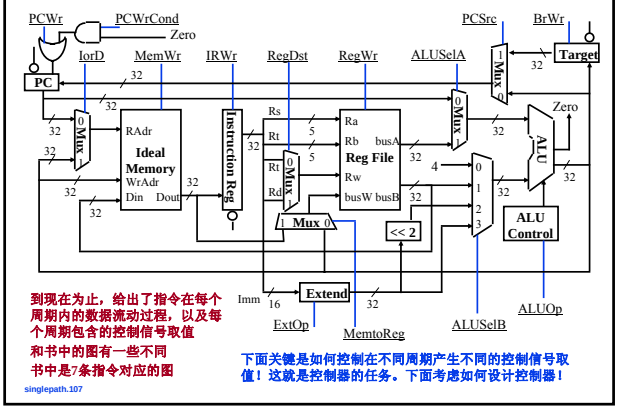
lw指令的第三个周期，控制信号取值？



RegDst=0,RegWr=1使Rw在RegWr=1后继续保持稳定
ALU输出在ALU控制不变时保持稳定，以使Dout和busW保持稳定。

singlepath.106

完成前述6条指令的完整多周期数据通路



到现在为止，给出了指令在每个周期内的数据流动过程，以及每个周期包含的控制信号取值

和书中的图有一些不同

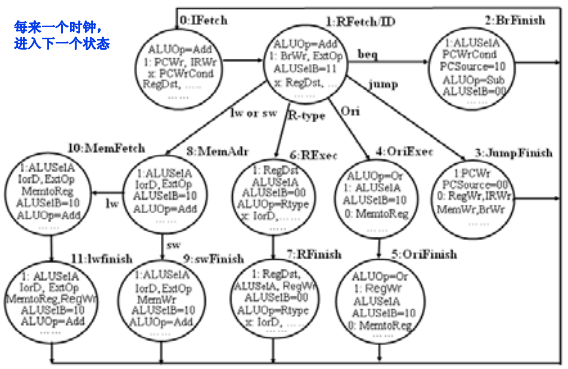
书中是7条指令对应的图

下面关键是如何控制在不同周期产生不同的控制信号取值！这就是控制器的任务。下面考虑如何设计控制器！

singlepath.107

状态转换图

每来一个时钟，进入下一个状态



问题：各指令的时钟数多少？ R-4, ori-4, beq-3, Jump-3, lw-5, sw-4

下一步目标：设计"状态转换电路"即：控制器

singlepath.108

多周期控制器的实现

回忆单周期控制器的实现：
控制信号在整个指令执行过程中不变，用真值表能反映指令和控制信号的关系。
根据真值表就能实现控制器！
多周期控制器能不能这样做？

多周期数据通路的控制更复杂，体现在：
每个指令有多个周期，每个周期控制信号取值不同！

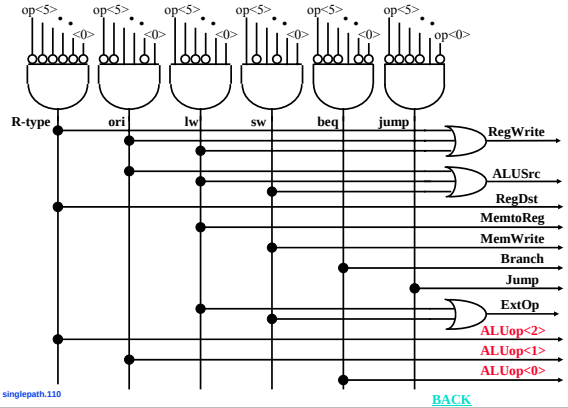
- 多周期控制器功能描述方式：
  - 有限状态机：
    采用组合逻辑设计
    用硬连线路(PLA)实现
  - 微程序：
    用ROM存放微程序实现

初始表示
顺序控制
逻辑表示
实现技术

Finite State Diagram	Microprogram
↓	↓
Explicit Next State Function	Microprogram counter + Dispatch ROMs
↓	↓
Logic Equations	Truth Tables
↓	↓
PLA	ROM
"hardwired control"	*"microprogrammed control"*
硬连线路控制器（硬布线控制器）	微程序控制器

SKIP

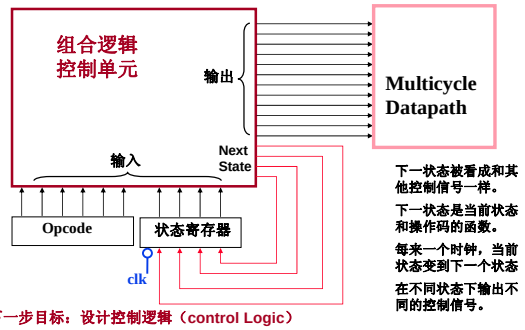
复习：单周期数据通路（The Main Control）



BACK

时序控制的描述

思路：由时钟、当前状态和操作码确定下一状态。不同状态输出不同控制信号值
控制逻辑采用"摩尔机"方式，即：输出函数仅依赖于当前状态



下一状态被看成和其他控制信号一样。
下一状态是当前状态和操作码的函数。
每来一个时钟，当前状态变到下一个状态
在不同状态下输出不同的控制信号。

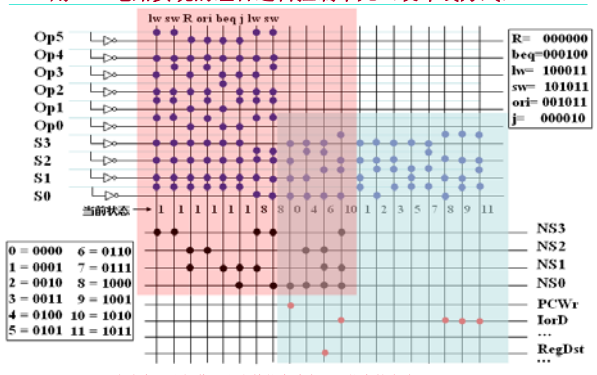
下一步目标：设计控制逻辑（control Logic）

多周期控制器状态转换表

当前状态 $S_3S_2S_1S_0$	指令操作码 $OP_5OP_4OP_3OP_2OP_1OP_0$	下一状态 $NS_3NS_2NS_1NS_0$
State2、3、5、7、9、11		0 0 0 0
State0 (IFetch)		0 0 0 1
State1 (ID/RFetch)	000100 (beq)	0 0 1 0
State1 (ID/RFetch)	000010 (jump)	0 0 1 1
State1 (ID/RFetch)	001101 (ori)	0 1 0 0
State4 (OriExec)		0 1 0 1
State1 (ID/RFetch)	000000 (R-type)	0 1 1 0
State6 (RExec)		0 1 1 1
State1 (ID/RFetch)	100011 (lw)	1 0 0 0
State1 (ID/RFetch)	101011 (sw)	1 0 0 0
State8 (MemAdr)	101011 (sw)	1 0 0 1
State8 (MemAdr)	100011 (lw)	1 0 1 0
State10 (MemFetch)		1 0 1 1

以上功能可以由PLA电路来实现！

用PLA电路实现的组合逻辑控制单元（硬布线方式）



左上角：由操作码和当前状态确定下一状态的电路
右下角：由当前状态确定控制信号的电路

第三讲小结

- 单周期CPU和多周期CPU的成本比较：
  - 单周期下功能部件不能重复使用；而多周期下可重复使用，比单周期省
  - 单周期指令执行结果直接保存在PC、Regfile和Memory；而多周期下需加一些临时寄存器保存中间结果，比单周期费
- 单周期CPU和多周期CPU的性能比较：
  - 单周期CPU的CPI为1，但时钟周期为最长的load指令执行时间
  - 多周期CPU的CPI是多少？时钟周期多长？

假定程序中22%为Load，11%为Store，49%为R-Type，16%为Branch，2%为Jump。每个状态需要一个时钟周期，CPI为多少？
分析如下：每种指令所需的时钟周期数为：
Load：5；Store：4；R-Type：4；Branch：3；Jump：3
CPI计算如下：
CPI=CPU时钟周期数 / 指令数 = Σ（指令数$_i$ x CPI$_i$）/ 指令数
= Σ（指令数$_i$ / 指令数）x CPI$_i$
CPI = 0.22x5+0.11x4+0.49x4+0.16x3+0.02x3 = 4.04
假设单周期时钟宽度为1，则多周期时钟周期约为单周期的1/5，所以，
多周期的总体时间约：4.04x1/5=0.8 ；而单周期总体时间为：1x1=1
由此看出：多周期比单周期效率高！

第四讲 微程序设计和异常处理

主 要 内 容

- 硬连线路控制器设计的优点和缺点
- 微程序设计控制器的基本思想
- 微程序、微指令、微操作和微命令的概念及其关系
- 微指令格式设计
  - 微操作码字段
    - 水平微程序：不译法、字段直接编译法、字段间接编译法
    - 垂直微程序：垂直编译法
  - 下条微指令地址确定方式
    - 增量法（计数器法）
    - 断定法（下址字段法）
- MIPS指令子集的微程序控制器设计
- 为什么处理器设计要考虑异常的处理
- “异常”和“中断”的概念
- 如何在数据通路中加入异常处理部件
- 如何控制数据通路中的异常处理部件

硬连线路设计和微程序设计

硬连线路设计的特点：

优点：速度快，适合于简单或规整的指令系统，例如，MIPS指令集。

缺点：它是一个多输入/多输出的巨大逻辑网络。对于复杂指令系统来说，结构庞杂，实现困难；修改、维护不易；灵活性差。甚至无法用有限状态机描述！

简化控制器设计的一个方法：微程序设计

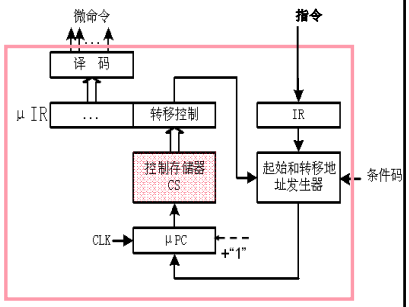
微程序控制器的基本思想：

- 仿照程序设计的方法，编制每个指令对应的微程序
- 每个微程序由若干条微指令构成，各微指令包含若干条微命令（一条微指令相当于一个状态，一个微命令就是状态中的控制信号）
- 所有指令对应的微程序放在只读存储器中，执行某条指令时，取出对应微程序中的各条微指令，对微指令译码产生对应的微命令，这个微命令就是控制信号。这个只读存储器称为控制存储器（Control Storage），简称控存CS 。

微程序设计的特点：具有规整性、可维性和灵活性，但速度慢。

微程序控制器的基本结构

- 输入：指令、条件码
- 输出：控制信号(微命令)
- 核心：控存CS
- μPC：指出将要执行的微指令在CS中的位置
- μIR： 正在执行的微指令
- 每个时钟执行一条微指令
- 微程序第一条微指令地址由起始地址发生器产生
- 顺序执行时， μPC+1
- 转移执行时，由控制转移字段指出对哪些条件码进行测试，转移地址发生器根据条件码修改μPC



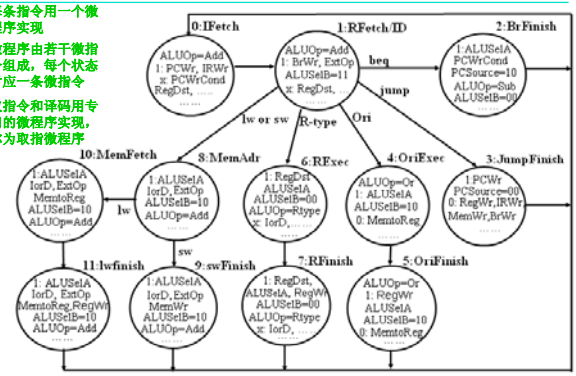
最初把固化在只读存储器的微程序称为固件（Firmware），表示用软件实现的硬部件，现在对固件通俗的理解是在ROM中“固化的软件”。

状态和微程序的对应关系

每条指令用一个微程序实现

微程序由若干微指令组成，每个状态对应一条微指令

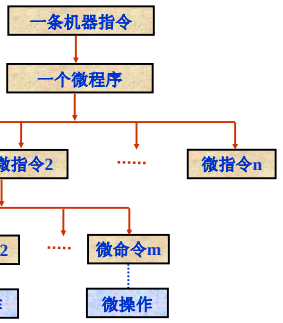
取指令和译码用专门的微程序实现，称为取指微程序



问题：上述取指微程序包含几条微指令？ 2条　　lw指令有几条微指令？ 3条

微程序\微指令\微命令\微操作的关系

将指令的执行转换为微程序的执行
微程序是一个微指令序列
每条微指令是一个0/1序列，其中包含若干个微命令（即：控制信号）
每个微命令控制数据通路的执行



控制程序执行要解决什么问题？
(1) 指令的编码和译码
(2) 下条指令到哪里去取
微程序执行也要解决两个问题：
(1)微指令中如何对微命令编码
(2)下条微指令在哪里

第一个问题：微指令格式的设计

微指令中包含了若干微命令、下条微指令地址（可选）、常数（可选）

微指令格式：

μOP	μADD	常数

μOP: 微操作码字段，产生微命令； μADD：微地址码字段，产生下条微指令地址

- 微指令格式设计风格取决于微操作码的编码方式（微命令：控制信号）
- 微操作码编码方式：
  - 不译法（直接控制法）
  - 字段直接编码（译）法
  - 字段间接编码（译）法

  } 水平型微指令风格

  指令采用的是哪种编码方式？

  最小（最短、垂直）编码（译）法 —— 垂直型微指令风格

- 水平型微指令
  - 基本思想：相容微命令尽量多地安排在一条微指令中。
  - 优点：微程序短，并行性高，适合于较高速度的场合。
  - 缺点：微指令长，编码空间利用率较低，并且编制困难。
- 垂直型微指令
  - 基本思想：一条微指令只控制一、二个微命令。
  - 优点：微指令短，编码效率高，格式与机器指令类似，故编制容易。
  - 缺点：微程序长，一条微指令只能控制一、二个，无并行，速度慢。
- 垂直型微指令面向算法描述，水平型微指令面向内部控制逻辑的描述

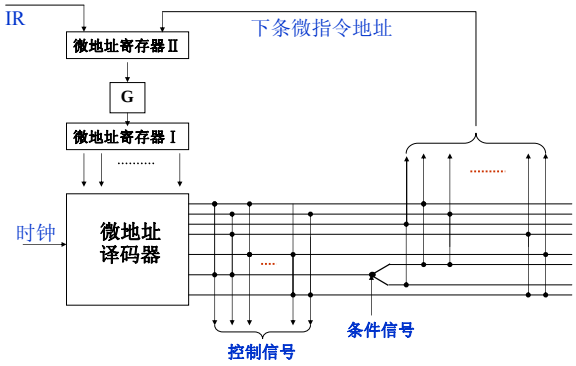
下面讨论第二个问题：
下条微指令的指定

不译法(直接控制法)

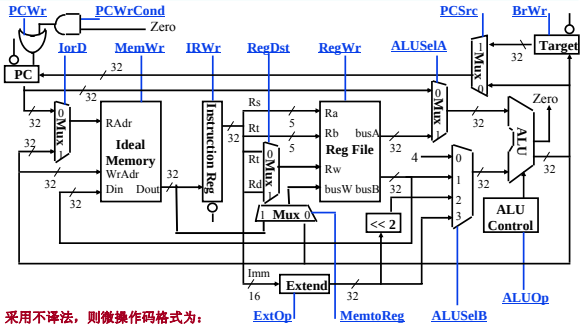
- **基本思想：**
  - 一位对应一个微命令（控制信号），不需译码。
  - 对于二值微命令(0/1)，本来就占一位，没有增加位数
  - 对于多值微命令，因为没有进行编码，因而相对来说，增加了位数。例如：
    - 4-1 MUX：编码则只需2位，不编码则要4位
    - ALUCtrl：编码则只需4位，不编码则要16位
- **优点：**
  - 并行控制能力强，不必译码，故执行速度快。
  - 编制的微程序短。
- **缺点：**
  - 微指令字很长，可能多达几百位。
  - 编码空间利用率低。(几百位中可能只有几位为1)

刚提出微程序设计时，采用的就是不译法。

Wilkes微程序控制器



多周期数据通路对应的微操作码



采用不译法，则微操作码格式为：

PCWr	IorD	MemWr	.....	PCSrc	BrWr	.....

BACK

控制字(即：微指令)的长度等于控制信号（微命令）的总位数

字段直接编码法

- **基本思想：**
  - 将微指令分成若干字段，每个字段对包含的若干微命令编码
  - 把互斥微命令组合在同一字段，相容微命令组合在不同字段
  - 一条微指令中最多可同时发出的微命令个数就是字段数

  相容微操作：能同时进行的微操作，称为相容的。

  互斥微操作：不能同时进行的微操作，称为互斥的。如：

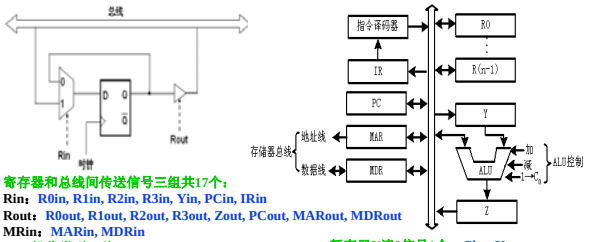
  ALU运算(add/sub/or/...)，存储器操作(读指令 / 读数据 / 写数据)

  你还能想出哪些互斥微操作？ 多路选择器的输入控制信号，等等
- **优点：**
  - 有较高的并行控制能力，速度较快。
  - 微指令短，能压缩到不译法的1/2到1/3，节省控存容量。
- **缺点:**
  - 增加译码线路，并开销一部分时间。但因分段后各字段位数少，所以译码对微指令的执行速度影响不大。

鉴于以上特点，它为大多数微程序控制的计算机所采用。

直接控制法和字段直接编译法举例

- **例1：假定图6.9和6.10所示单总线数据通路有4个通用寄存器R0，R1，R2和R3， 16种ALU操作，主存和CPU间采用“异步”方式通信，存取操作有Read和Write信号控制。每条指令结束时，都要执行一个公共操作，用来进行指令结束处理（如，查询是否有外部“中断”请求），由控制信号End控制。要求：分别写出采用直接控制法和字段直接编码法的微操作码格式。**



寄存器和总线间传送信号三组共17个：

Rin：R0in, R1in, R2in, R3in, Yin, PCin, IRin

Rout：R0out, R1out, R2out, R3out, Zout, PCout, MARout, MDRout

MRin：MARin, MDRin

ALU操作类型16种：add/sub/or/and/xor/…/mov

ALU进位信号1个：1→C0

暂存器Y清0信号1个：ClearY

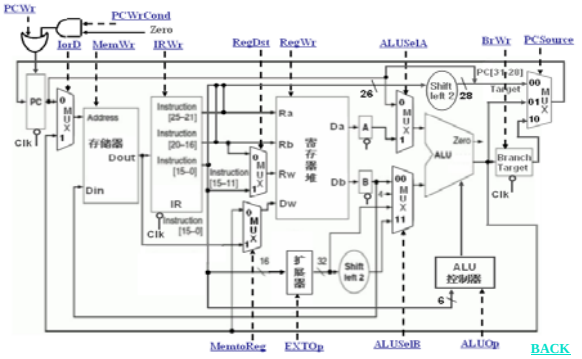
存储器信号3个：Read、Write、WMFC

结束信号1个：End

直接控制法和字段直接编译法举例

- **直接控制法**
  - μOP的长度 = 控制信号的总个数
  - ALU操作控制信号不是16个(是4个), 这是由ALU结构和功能决定的（为什么？）
  - 共有17+4+1+1+3+1=27个控制信号（微命令）
  - 微操作码字段共27位。某位为1，对应微命令有效，否则对应微命令无效
- **字段直接编码法（P.193 表6.8）**
  - 哪些微操作之间是互斥的？
    - Rout中信号之间：某时刻只能有一个寄存器输出到总线；
    - ALU操作控制信号间：某时刻ALU只能做一种操作
    - 主存读/写信号：不能同时读和写，有些节拍中没有读和写（No action）
  - 如何分组？
    - 按互斥关系分组：上述3个互斥组在3个不同字段中
    - 可同时做但不可能同时发生的：如Rin，MRin（这两组间可能同时发生）
    - 其余的需直接控制（无需编码）：如1→C_0, Clear Y, WMFC, END等
  - 共分9组， μOP仅有4+3+2+4+1+1+2+1+1=19位，比直接控制法少8位
  - 9组中有5组进行了编码，执行微指令时需译码

字段直接编码法举例（P.194 表6.9 / P.195 表6.10 ）



BACK

可分8组：ALUop(2位), ALUSelA(1位), ALUSelB(2位), RegOP(3位: RegWr/RegDst/MemtoReg), MemOP(2位: MemWr/IorD/IRWr), ExtOp, BrWr, PCWrOP(2位: PCSource/PCWr/PCWrCond)

字段间接编码法

- 基本思想：
  - 在字段直接编码法基础上，进一步压缩微指令长度。
  - 通过另一字段的编码或标志位来对某个字段的编码加以解释。

    即：一个微命令字段可以表示多个微命令组，到底代表哪一组微命令，则由另一个专门的字段来确定。
- 特点：
  - 可进一步缩短微指令字的长度，节省控存容量。（意义不大！）
  - 译码线路复杂，时间开销大。

鉴于以上特点，它只限于局部场合使用。

BACK

最小(最短、垂直)编码法

- 基本思想：
  - 采用指令编码思想（每条指令产生一个操作），每条微指令只包含一个微命令。即将所有微命令进行全编码。

    采用这种方式编码的微指令称为垂直型微指令

    由其组成的微程序称为垂直微程序。
- 特点：
  - 能得到最短的微指令字。
  - 微程序规整、直观，易于编制。
  - 但并行能力差，速度慢，并且微程序长。

主要用在具有两级微程序的控制器设计中，用垂直微程序解释指令，用水平微程序解释垂直微指令。此时，水平微程序称为毫微程序。

BACK

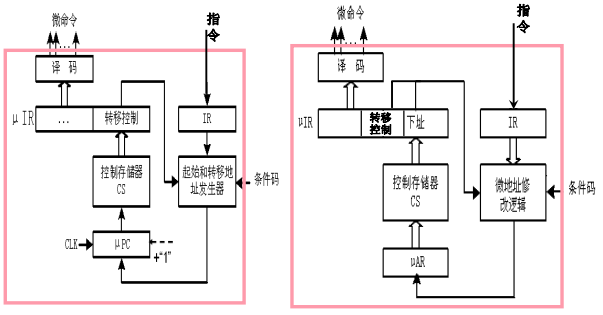
第二个问题：下条微地址的确定方式

- 什么是微程序执行顺序的控制？
  - 指在现行微指令执行完毕后，怎样控制产生下一条微指令的地址。
- 怎样控制微程序的执行顺序？
  - 通过在本条微指令中明显或隐含地指定下条微指令在控存中的地址来控制。
- 微指令地址的产生方法有两种：
  - 增量(计数器)法：下条微指令地址隐含在微程序计数器μPC中。
  - 断定（下址字段）法：在本条微指令中明显地指定下条微指令的地址。
- 选择下条要执行的微指令有三种情况：
  - 第一条微指令：每条指令执行完，就会取出下条指令执行，当指令取出后，需要转移到下条指令对应的第一条微指令执行。
  - 顺序执行时：在每条指令的微程序执行过程中顺序取出下条微指令执行。
  - 分支执行时：在遇到按条件转移到不同微指令执行时，需要根据控制单元的输入来选择下条微指令。

还有一种情况：

取指微程序首址：每条指令都要先执行“取指微程序”

不同微地址产生方法对应的控制器结构

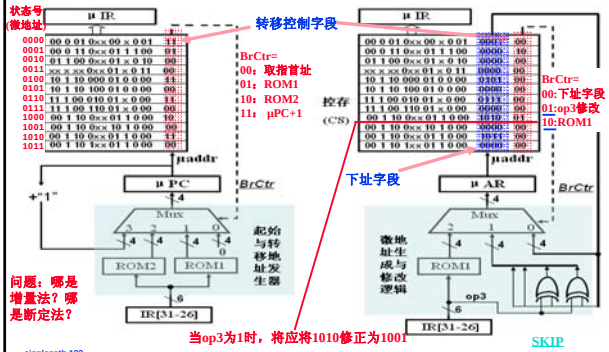


增量（计数器）法　　断定（下址字段）法

微程序控制器的设计

书 P.198 BrCtr 01和10换一下

- 举例：用“转移控制”字段实现分支，指令微程序首址在ROM中。分别采用计数器法和下址字段法实现表6.10给出的微程序，画出微程序控制器结构。



SKIP

多周期CPU的有限状态机

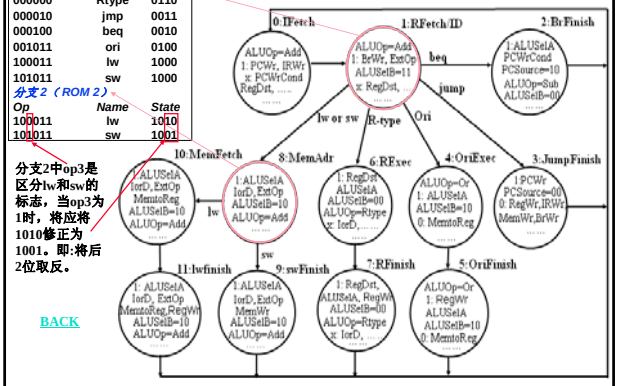
分支1（ROM 1）

Op	Name	State
000000	Rtype	0110
000010	jmp	0011
000100	beq	0010
001011	ori	0100
100011	lw	1000
101011	sw	1000

分支2（ROM 2）

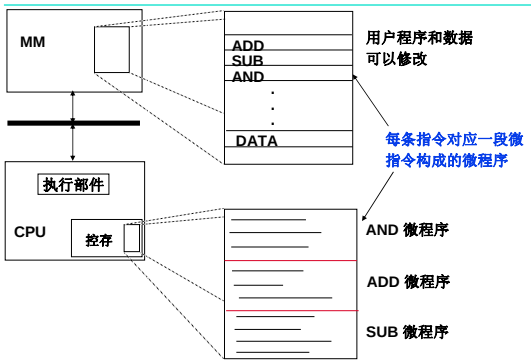
Op	Name	State
100011	lw	1010
101011	sw	1001

分支2中op3是区分lw和sw的标志，当op3为1时，将应将1010修正为1001。即:将后2位取反。



BACK

微指令字的解释执行



异常和中断的处理

- 程序执行过程中，CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - 此时，CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，执行后再返回到原被中止的程序继续执行
- 使程序执行被 “中断” 的事件有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件
    - 按发生原因分为硬故障中断和程序性中断两类
    - 硬故障中断：如电源掉电、硬件线路故障等
    - 程序性中断：执行某条指令时发生的“例外(Exception)”，如溢出、缺页、越界、越权、非法指令、除数为0、堆栈溢出、访问超时、断点、单步、系统调用等
    - 按处理方式分为故障(fault)、自陷(Trap)和终止(Abort)三类
    - 故障：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等
    - 自陷：预先安排的事件，如单步跟踪、系统调用(执行访管指令)等（自愿中断）
    - 终止：硬故障事件，机器将“终止”，调出中断服务程序来重启操作系统
    - 思考：哪些故障补救后可继续执行，哪些只好终止当前进程？
      - 缺页等：补救后可继续，回到发生故障的指令重新执行
      - 溢出、除数为0、非法操作、内存保护错等：终止当前进程
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理
    - 如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等
    - (中断是一种I/O方式，所以有关中断的概念在第9章介绍。)
- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！

思考：自陷处理完成后回到哪条指令执行
回到下条指令！

本章主要介绍如何在数据通路中增加“程序性异常”的检测和处理逻辑

举例-8086/8088中断系统

统称为“中断”：内中断（内部异常）和外中断（外部中断）

- 内中断：CPU自己产生而不通过中断请求线请求，皆为不可屏蔽中断。
  - 指令引起异常：CPU执行预置的指令后在特定的情况下发生的异常。
    - INTO 溢出：执行算术指令后，若发生溢出，则产生类型4中断。
    - INT n 用户定义：指令的第二字节给出一个类型号(n=0～255)。
      - 其中n=3 (INT 3)时为断点设置，该指令执行后，自动产生类型3中断。
  - 处理器检测异常：CPU执行指令时产生的异常，如：除法错、无效操作码、缺页、单步跟踪调试等。如：
    - 除法错：除数为0或商溢出，则产生类型0中断。
    - 单步跟踪：当自陷位TF=1且处在开中断状态(即IF=1)时，每条指令执行完就自动产生类型1中断。
- 外中断：通过中断请求线INTR和NMI来实现。
  - INTR：可屏蔽中断 (外设中断源引起的中断)。
  - NMI：不可屏蔽中断 (重要或紧急的硬件故障)，属于类型2中断。

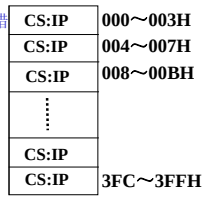
所有事件都被分配一个“中断类型号”
每个中断都有相应的“中断服务程序”
可根据中断类型号找到中断服务程序的入口地址

8086/8088的中断向量表

中断向量表也称中断入口地址表（或异常表），位于0000H～03FFH。共256组，每组占四个字节 CS:IP 。向量地址=中断类型号 x 4

例1：除法错的中断类型号为0，故其向量地址为：0x4=0

例2：NMI的中断类型号为2，故其向量地址为：2x4=8



- 中断向量表（异常表）中每一项是对应中断服务程序的入口地址。被称为中断向量(Interrupt Vector)
- 中断向量表的起始地址存放在一个异常表基址寄存器中。

处理器中的异常处理机制

检测到异常时，处理器必须进行以下基本处理：
① 关中断：使处理器处于“禁止中断”状态，以防止新异常(或中断)破坏断点和现场
②保护断点和程序状态：将断点和程序状态保存到堆栈或特殊寄存器中
- PC→堆栈 或 EPC（专门存放断点的寄存器）
- PSWR →堆栈 或 EPSWR （专门保存程序状态的寄存器）
- (PSW（Program Status Word）：程序状态字，包括条件码、中断码、状态位等
- PSWR（PSW寄存器）：用于存放程序状态字的寄存器。如，X86的FLAGS）

③识别异常事件：有两种不同的方式：软件识别和硬件识别（向量中断方式）

（1）软件识别（MIPS采用）

设置一个异常状态寄存器（MIPS中为Cause寄存器），用于记录异常原因。操作系统使用一个统一的异常处理程序，该程序按优先级顺序查询异常状态寄存器，识别出异常事件。

（例如：MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序，用于检测异常的具体原因，然后转到内核中相应的异常处理程序段中进行具体的处理）

（2）硬件识别（向量中断）（80x86采用）

用专门的硬件查询电路按优先级顺序识别异常，得到“中断类型号”，根据此号，到中断向量表中读取对应的中断服务程序的入口地址。

8086/8088的中断向量表

中断向量表也称中断入口地址表（或异常表），位于0000H～03FFH。共256组，每组占四个字节 CS:IP 。向量地址=中断类型号 x 4

例1：除法错的中断类型号为0，故其向量地址为：0x4=0

例2：NMI的中断类型号为2，故其向量地址为：2x4=8

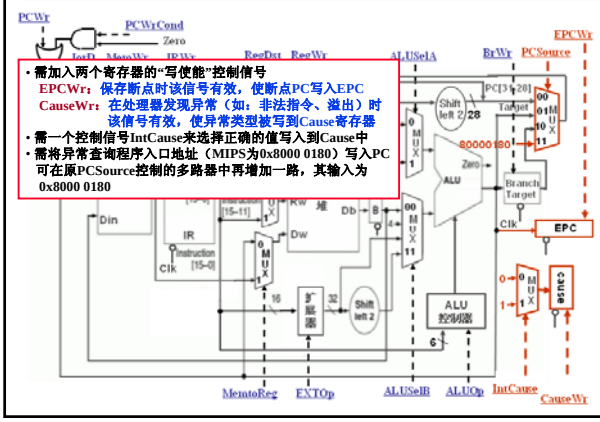
除法错	CS:IP	000～003H
单步	CS:IP	004～007H
NMI	CS:IP	008～00BH
	CS:IP	
	CS:IP	3FC～3FFH

- 中断向量表（异常表）中每一项是对应异常处理程序的入口地址。被称为中断向量(Interrupt Vector)
- 中断向量表的起始地址存放在一个异常表基址寄存器中。

MIPS带异常处理的数据通路设计

- MIPS采用软件（操作系统提供的一个特定的异常查询程序）识别中断源
- 数据通路中需增加以下两个寄存器：
  - EPC：32位，用于存放断点（异常处理后返回到的指令的地址）
    - 写入EPC的断点可能是正在执行的指令（故障时），也可能是下条指令（自陷和中断时）。前者需要把PC的值减4后送到EPC，后者则直接送PC到EPC
  - Cause：32位（有些位还没有用到），记录异常原因
    - 假定处理的异常类型有以下两种：
      - 未定义指令（Cause=0）
      - 数据溢出（Cause=1）
- 需要加入两个寄存器的“写使能”控制信号
  - EPCWr：在保存断点时该信号有效，使断点PC写入EPC
  - CauseWr：在处理器发现异常（如：非法指令、溢出）时，该信号有效，使异常类型被写到Cause寄存器
- 需要一个控制信号IntCause来选择正确的值写入到Cause中
- 需要将异常查询程序的入口地址（MIPS为0x8000 0180）写入PC，可以在原来PCSource控制的多路复用器中再增加一路，其输入为0x8000 0180

带异常处理的数据通路



- 需加入两个寄存器的“写使能”控制信号
  - EPCWr：保存断点时该信号有效，使断点PC写入EPC
  - CauseWr：在处理器发现异常（如：非法指令、溢出）时该信号有效，使异常类型被写到Cause寄存器
- 需一个控制信号IntCause来选择正确的值写入到Cause中
- 需将异常查询程序入口地址（MIPS为0x8000 0180）写入PC可在原PCSource控制的多路器中再增加一路，其输入为0x8000 0180

带异常处理的控制器设计

- 在有限状态机中增加异常处理的状态，每种异常占一个状态
- 每个异常处理状态中，需考虑以下基本控制
  - Cause寄存器的设置
  - 计算断点处的PC值（PC-4），并送EPC
  - 将异常查询程序的入口地址送PC
  - 将中断允许位清0（关中断）
- 假设要控制的数据通路中有以下两种异常处理
  - 未定义指令（Cause=0）：状态12
  - 数据溢出（Cause=1）：状态13

  注：7条指令共需12个状态：第0~11状态
- 在原来状态转换图基础上加入两个异常处理状态
  - 如何检测是否发生了这两种异常
    - 未定义指令：当指令译码器发现op字段是一个未定义的编码时
    - 数据溢出：当R-Type指令执行后在ALU输出端的Overflow为1时

加入异常处理后的有限状态转换图

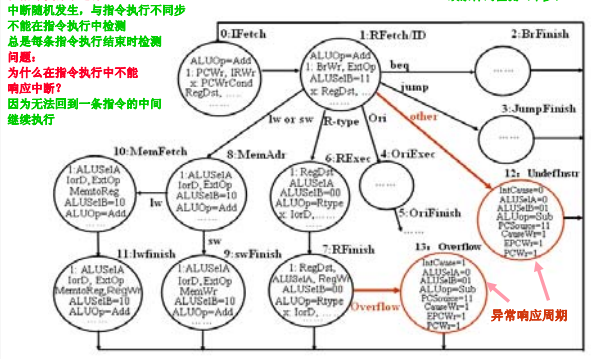
问题：何时检测“缺页”异常？MMU中地址转换时！

问题：中断检测能否和异常检测一样在指令执行中进行？中断随机发生，与指令执行不同步不能在指令执行中检测总是每条指令执行结束时检测

问题：为什么在指令执行中不能响应中断？因为无法回到一条指令的中间继续执行

“fault”异常的检测在指令执行中。“trap”异常怎样检测？

指令译码（系统调用）或条件码检测（单步）



异常响应周期

加入异常处理后的控制器设计可根据上述有限状态机实现！

TLB缺失处理和缺页处理

- TLB缺失处理（可以由硬件处理，也可发出“TLB缺失”异常由软件来处理）
  - TLB miss说明可能发生以下两种情况之一：
    - 页在内存中：只要把主存中的页表项装载到TLB中
    - 页不在内存中(缺页)：OS从磁盘调入一页，并更新主存页表和TLB
- 缺页（page fault）处理
  - 当主存页表的页表项中“valid”位为“0”时，发生page fault
  - Page fault是一种“故障”异常，按以下方式处理（MIPS异常处理）
    - 关中断（中断允许位清0）
    - 在Cause寄存器置相应位为“1”
    - 发生缺页的指令地址（PC减4）送EPC
    - 0x8000 0180(异常查询程序入口)送PC
      - 执行OS的异常查询程序，取出Cause寄存器中相应的位分析，得知发生了“缺页”，转到“缺页处理程序”执行
  - page fault一定要在发生缺失的存储器操作时钟周期内捕获到，并在下个时钟转到异常处理，否则，会发生错误。
    - 例：lw $1, 0($1)，若没有及时捕获“异常”而使$1改变，则再重新执行该指令时，所读的内存单元地址被改变，发生严重错误！

实例：IA-32处理器的实现

- 问题：IA-32处理器适合用单周期还是多周期方式来实现？
  - 单周期方式：
    - 每条指令都按最复杂指令时间执行（指令执行效率低！）
    - 功能部件不能重复使用，对于一条具有多个复杂寻址的指令来说，可能要用到相当多个ALU。（成本高！）
  - 多周期方式：
    - 每条指令执行时间可以不同，简单指令3-4个时钟，复杂指令几十个时钟（指令执行效率高！）
    - 功能部件可以在一条指令执行过程中重复使用，这对于一条指令中具有多个复杂寻址的指令，非常有好处（成本低！）
- 问题：IA-32处理器适合用硬连线路控制器还是微程序控制器来实现？
  - Hardwired Control：速度快，但无法实现复杂指令
  - Microprogrammed control：容易实现复杂指令，但速度慢
- 从80x486开始，采用了一种折中的方式：
  - 简单指令（在数据通路中可一遍执行完）用Hardwired Control
  - 复杂指令用microcoded control，不需为复杂指令构造复杂的数据通路

多周期数据通路和微程序控制器为IA-32指令集提供了一个实现框架

下一章详细介绍Pentium4处理器（是一种IA-32结构）的流水线实现

本讲小结

- 硬连线路控制器的优点是速度快，适合于简单规整指令集的数据通路；缺点是设计周期长、繁琐、不灵活、不易修改和增删指令
- 微程序控制器设计借用程序设计思想，将每个周期所涉及的状态用只读存储器保存起来，执行到某条指令时，把这条指令对应的状态按序取出，转换为控制信号。优点：简化设计、灵活、易修改、易维护；缺点：速度慢。
- 微指令格式设计
  - 微操作码字段大多采用字段直接编译法，将互斥微命令组合在同一个字段进行编码。这样，在缩短微指令字的同时，保证了并行性，并避免同一周期出现两个不能同时执行的微命令的问题。
  - 下条微指令地址可以采用计数器（增量）法和下址字段（断定）法；两种方法都要解决分支问题。可以增加一个“转移控制”信号来解决下条微地址的顺序控制问题。
- 异常会改变程序执行流程，所以处理器设计要考虑异常处理
- 在数据通路中加入异常处理必须考虑：
  - 保存断点和异常原因，并将控制转到异常处理程序的首地址处
- 带异常的有限状态机中，每个异常对应一个状态和进入状态的检测条件

本章总结1

- CPU的主要功能
  - 周而复始执行指令
  - 执行指令过程中，若发现异常情况，则转异常处理
  - 定时查询有没有DMA请求，有DMA请求的话，则让出总线
  - 每个指令结束，查询有没有中断请求，有则响应中断
- CPU的内部结构
  - 由数据通路(Datapath)和控制单元(Control unit)组成
    - 数据通路中包含组合逻辑单元和存储信息的状态单元
      - 组合逻辑单元用于对数据进行处理，如：加法器、运算器ALU、扩展器（0扩展或符号扩展）、多路选择器、以及状态单元的读操作线路等。
      - 状态单元包括触发器、寄存器、寄存器堆、数据/指令存储器等，用于对指令执行的中间状态或最终结果进行保存。
    - 控制单元对取出的指令进行译码，与指令执行得到的条件码或当前机器的状态、时序信号（时钟）等组合，生成对数据通路进行控制的控制信号

本章总结2

- CPU中的寄存器
  - 用户可见寄存器（用户可使用）
    - 通用寄存器：用来存放地址或数据，需在指令中明显给出
    - 专用寄存器：用来存放特定的地址或数据，无需在指令中明显给出
    - 数据寄存器：专用于保存数据，可以是通用或专用寄存器
    - 地址寄存器：专用于保存地址，可以是通用或专用寄存器。如：段指针、变址器、基址器、堆栈指针、栈帧指针等。
    - 标志(条件码)寄存器：部分可见。由CPU根据指令执行结果设定，只能以隐含方式读出其中若干位，用户程序（非内核程序）不能改变
  - 控制和状态寄存器（用户不可使用）
    - 程序计数器PC
    - 指令寄存器IR
    - 存储器地址寄存器MAR
    - 存储器缓冲(数据)寄存器 MBR / MDR
    - 程序状态字寄存器PSWR
    - 临时寄存器：用于存放指令执行过程中的临时信息
    - 其他寄存器：如，进程控制块指针、系统堆栈指针、页表指针等

本章总结3

- 指令执行过程
  - 取指、译码、取数、运算、存结果、查中断
  - 指令周期：取出并执行一条指令的时间，由若干个时钟周期组成
  - 时钟周期：CPU中用于信号同步的信号，是CPU最小的时间单位

（注：传统处理器中，一个指令周期由多个机器周期组成。一般把完成一次总线操作访问主存或I/O的时间称为机器周期，一个机器周期由多个时钟组成）

- 数据通路的定时方式
  - 现代计算机都采用时钟信号进行定时
  - 一旦时钟有效信号到来，数据通路中的状态单元可以开始写入信息
  - 如果状态单元每个周期都更新信息，则无需加“写使能”控制信号，否则，需加“写使能”控制信号，以使必要时控制信息写入
- 数据通路中信息的流动过程
  - 每条指令在取指令阶段和指令译码阶段都一样
  - 每条指令的功能不同，故在数据通路中所经过的部件和路径可能不同
  - 数据在数据通路中的流动过程由控制信号确定
  - 控制信号由控制器根据指令代码来生成

本章总结4

- 单周期处理器的设计
  - 每条指令都在一个时钟周期内完成
  - 时钟周期以最长的Load指令所花时间为准
  - 无需加临时寄存器存放指令执行的中间结果
  - 同一个功能部件不能重复使用
  - 控制信号在整个指令执行过程中不变，所以控制器设计简单，只要写出指令和控制信号之间的真值表，就可以设计出控制器
- 多周期处理器的设计
  - 每条指令分成多个阶段，每个阶段在一个时钟内完成
  - 不同指令包含的时钟个数不同
  - 阶段的划分要均衡，每个阶段只能完成一个独立、简单的功能，如：
    - 一次ALU操作
    - 一次存储器访问
    - 一次寄存器存取
  - 需加临时寄存器存放指令执行的中间结果
  - 同一个功能部件能在不同的时钟中被重复使用
  - 可用有限状态机来表示指令执行流程，并以此设计控制器

本章总结5

- 控制单元实现方式
  - 有限状态机描述方式
    - 每个时钟周期包含的控制信号的值的组合看成一个状态，每来一个时钟，控制信号会有一组新的取值，也就是一个新的状态
    - 所有指令的执行过程可用一个有限状态转换图来描述
    - 用一个组合逻辑电路（一般为PLA电路）来生成控制信号，用一个状态寄存器实现状态之间的转换
    - 也称为组合逻辑电路设计方式
    - 实现的控制器称为硬布线控制器
  - 微程序描述方式
    - 每个时钟周期所包含的控制信号的值的组合看成是一个0/1序列，每个控制信号对应一个微命令，控制信号取不同的值，就发出不同的微命令
    - 若干微命令组合成一个微指令，每条指令所包含的动作就由若干条微指令来完成，每来一个时钟，执行一条微指令
    - 每条指令执行时，先找到对应的第一条微指令，然后按照特定的顺序取出后续的微指令执行
    - 实现的控制器称为微程序控制器

第六章作业

- 2（4）（5）（6）
- 3，4，5，6，7，8，9，10，12，13，15，16

下星期五（5月22日）交作业！