

嵌入式微处理器系统

崔光佐

普适计算与应用实验室

北京大学现代教育技术中心

www.uclab.org



第一篇

第二讲 微处理器结构与实现技术简介

2004.2.16

主要内容

- ?微处理器的系统结构设计
- ?微处理器的执行机制设计
- ?微处理器流水线结构设计
- ?微处理器逻辑设计与实现
- ?处理器逻辑综合 (Synopsys)
- ?处理器物理综合 (Candence)
- ?处理器的验证与测试

处理器的按应用范围分类

通用处理器：面向一般应用，PC等

嵌入式处理器：不独立使用，而作为系统的一部分
ARM, MIPS等

专用处理器：面向特定领域/算法。如DSP, 加密处理器

通用处理器一般要求性能，价格，**功耗**：MIT, BERKELEY

嵌入式处理器注重功耗、开发时间、面积

专用处理器注重速度，开发时间，功耗

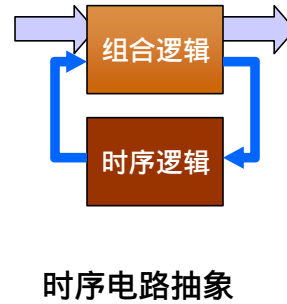
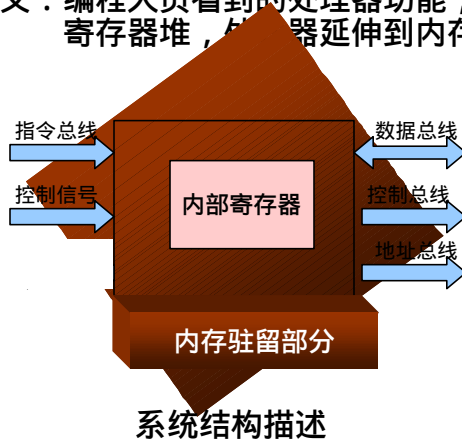
微处理器的系统结构设计

微处理器的设计与实现过程：

- 微处理器的系统结构设计
- 微处理器的执行机制设计
- 微处理器流水线结构设计
- 微处理器逻辑设计与实现
- 处理器逻辑综合(Synopsys)
- 处理器物理综合(Candence)
- 处理器的验证与测试

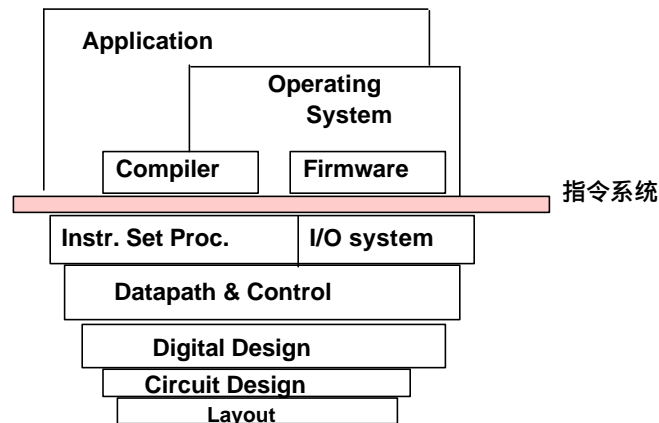
微处理器的系统结构设计

定义：编程人员看到的处理器功能，包括指令系统，寄存器堆，处理器延伸到内存的驻留部分。



处理器系统结构设计--指令系统设计

计算机系统结构抽象

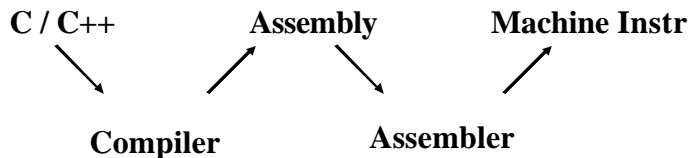


抽象的级别

- 固态物理
- 导体,绝缘体, 半导体.
- 掺杂二氧化硅形成二极管和晶体管.
- 建立简单的门, 布尔逻辑和真值表
- 组合逻辑: 多路选择器, 译码器, 加法器等
- 时钟
- 时序逻辑: 锁存器, 寄存器
- 状态机
- 处理器控制: 机器指令
- 计算机系统结构: 指令系统定义

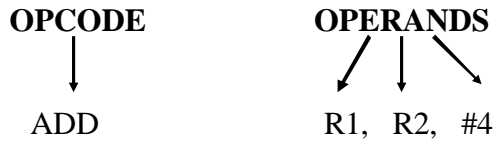
指令系统结构抽象

- 最底层的软件 (汇编语言) 与最高抽象层硬件表示的接口(指令集系统结构).



系统结构与实现

系统结构确定硬件和软件之间的接口



一种系统结构可由多种实现，如80x486, Pentium.

指令系统设计

- 指令系统应该包括什么样的指令？
 - add, branch, load/store
 - multiply, divide, sqrt
 - mmx_add
- 存储单元如何确定？
 - 多少个寄存器？
 - 多少存储器？
 - 多少专用寄存器，系统存储器？
- 指令的格式如何？
 - 多少个操作数？

指令系统结构评价的常规方法

- 性能
 - “my computer is **faster** than yours!”
- 价格
 - “my computer is **cheaper** than yours!”
- 大小
 - “my mp3 player is **smaller** than yours!”
 - “my machine has **more** memory than yours!”
- 功耗, 兼容性, 可靠性

结构设计者关心：以上问题与系统结构的关系。

指令系统与处理器结构的关系

指令系统设计：

决定于应用、性能、代码密度和方言的要求。
包括符号指令设计和编码设计。

指令的类型：

寻址方式：指令系统的重要特点。

与数据通路相关。

传输类指令：实现处理器内部存储之间以及
与外部存储之间的数据传送。

与数据通路相关

运算类指令：实现指令描述的功能。

与ALU和其它运算部件相关

系统类指令：完成对系统资源的访问。

与操作系统的支持有关。

指令系统与处理器结构的关系

指令的编码：相关因素：代码密度，功耗，译码器

垂直编码有利于译码器简化，但使用效率低

非垂直编码译码复杂，使用效率高。

常常采用二者折衷方案。ARM

指令编码与功耗：

连续执行的执行功耗取决于其引起的逻辑变化量

指令编码的海明距离，控制信号的海明距离，

执行情况等，编译技术，OS，嵌入式应用



指令系统考虑的问题：

兼容与非兼容问题：

兼容：x86, MIPS, ARM等

优点：设计难度小，软件丰富，开发时间短，市场

缺点：版权问题，竞争激烈，品牌效应

具有应用前景的处理器

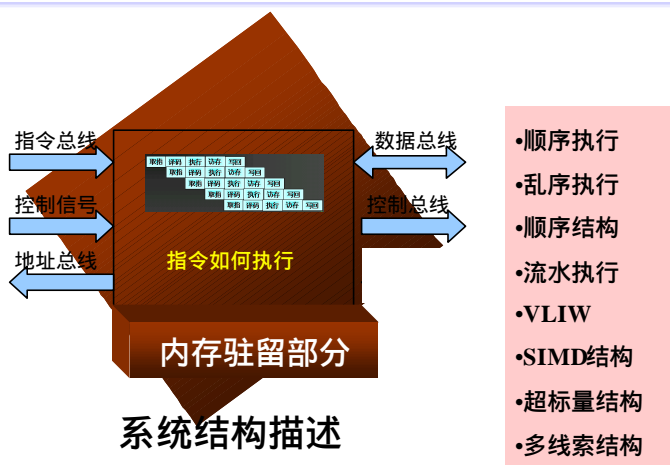
非兼容：

优点：版权所有,面向应用

缺点：设计难度大，软件自己开发，
开发周期长竞争激烈，用户难以接受

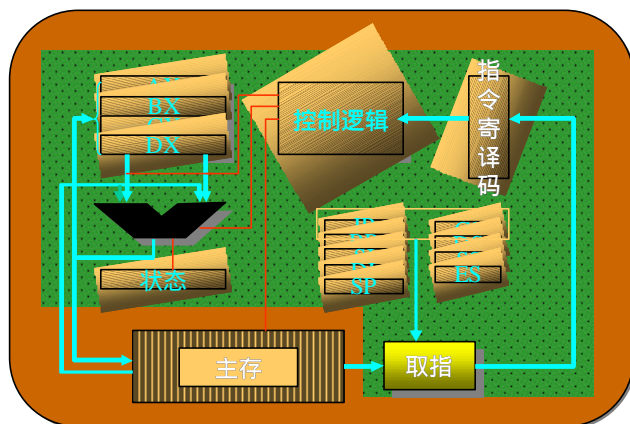
专用处理器，嵌入式处理器

微处理器的执行机制设计

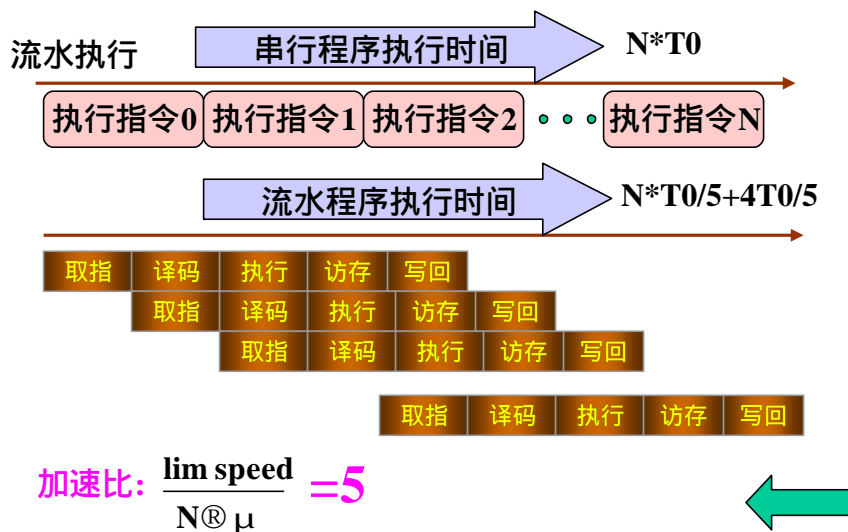


微处理器的执行机制设计

串行执行：每条指令不重叠。



微处理器的执行机制设计



微处理器的执行机制设计

超流水：Superpipeline, P4 Netburst

FSU

DEPARTMENT OF COMPUTER SCIENCE

Superpipelined Example

- Simple example with MIPS pipeline.

cycle	1	2	3	4	5	6	7	8
lw \$t0,0(\$t4)	IF	ID	EX	MEM	WB			
add \$3,\$1,\$3		IF	ID	EX	MEM	WB		
sub \$7,\$2,\$3			IF	stall	ID	EX	MEM	WB

- How would these instructions proceed through the following superpipeline?
 - IF (first half of inst fetch)
 - IS (second half of inst fetch)
 - ID (inst decode, reg fetch occurring during second half of clock cycle)
 - EE (first half of ALU oper and effective addr calc)
 - ES (second half of ALU oper and effective addr calc)
 - MF (first half of data cache access)
 - MS (second half of data cache access)
 - WB (write back occurring during first half of clock cycle)

微处理器的执行机制设计

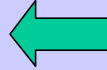
流水线的问题:

流水线相关:

数据相关: `mov r0,[r1]; add r1,r0`

控制相关: `sub r1,r2; bge label0`

结构相关: 资源分配不足.



相关解决:

Register rename, Reorder Buffer, Forwarding

ByPass, Branch Delay Slot , Speculative Execution

Branch Predict(static, dynamic)..

---静态解决方法,动态解决方法.

微处理器的执行机制设计

流水线的问题:

假设流水线有 s 个流水段, J 指令紧接 I 指令进入流水线, 而且 J 指令的第 q_j 流水段要用 I 指令在第 q_i 流水段的执行结果. 假设 J 指令置后 I 指令发射的时钟数为 x_{ij} , 且 I 、 J 指令分别于时钟 i 和 j 进入流水线, 则 $x_{ij} = j - i - 1$. 此时定义该两条指令的相关度为

$$d_{ij} = q_i - q_j - x_{ij}.$$

当 $d_{ij} \leq 0$ 时, 称为指令 I 和 J 没有流水线相关, d_{ij} 的值越小则流水线的性能越高.

流水线相关说明



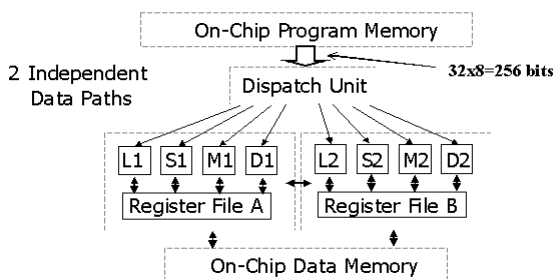
$$x_{ij} = 2 - 1 = 1$$

$$d_{ij} = q_i - q_j - x_{ij} \\ = 5 - 4 - 1 = 0$$

微处理器的执行机制设计

VLIW: Crosue, IA-64

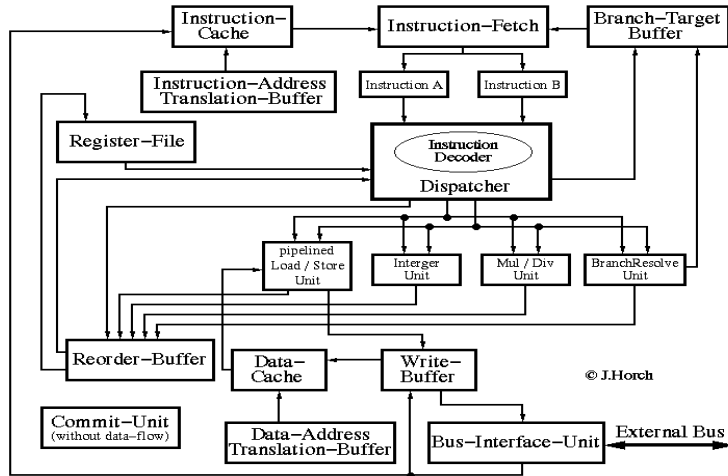
Example VLIW Data Path ('C6x)



编译要求高
代码密度低
存储带宽
部件利用率

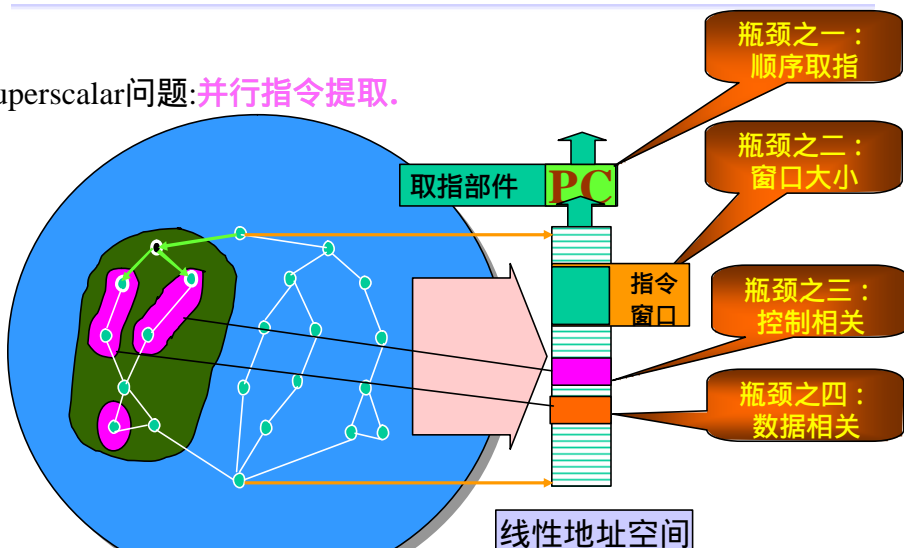
微处理器的执行机制设计

Superscalar: 顺序取指, 乱序执行, 顺序提交



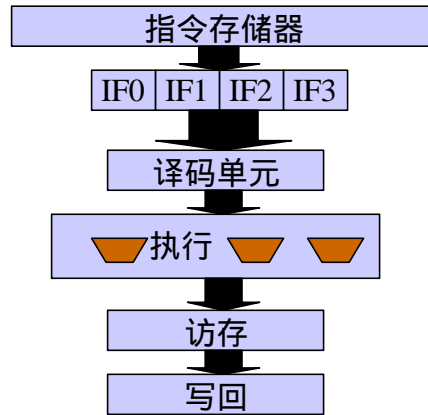
微处理器的执行机制设计

Superscalar问题: 并行指令提取.



微处理器的执行机制设计

多线索结构

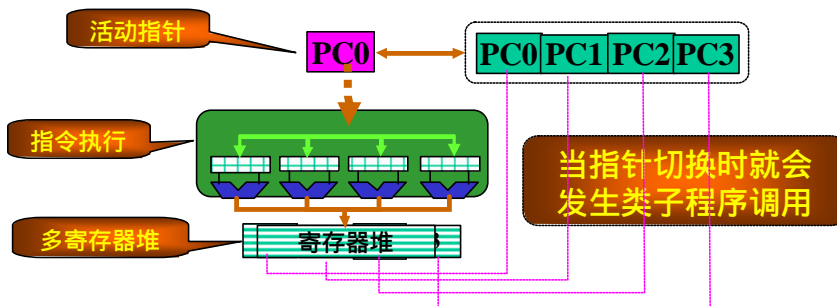


微处理器的执行机制设计

解决瓶颈之一：顺序取指

一般解决方案：

保留多个程序指针PC（多线索），交替执行多个pc的指令，达到无序取指。



微处理器的执行机制设计

解决瓶颈之二：指令窗口大小

从一个指令流中提取出并行指令的个数依赖于窗口中的指令数，即窗口大小。

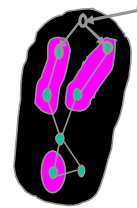
增大窗口的大小：硬件开销剧增。

对于大指令窗口，需要从cache取出多条指令

解决控制相关：

静态调度：编译时将分支后的指令尽量前提。VLIW

动态调度：执行阶段对转移指令进行推测。



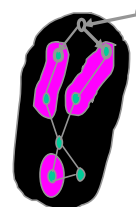
微处理器的执行机制设计

解决瓶颈之三：数据相关

通过指令窗口调度并行指令：解决数据相关。

解决：

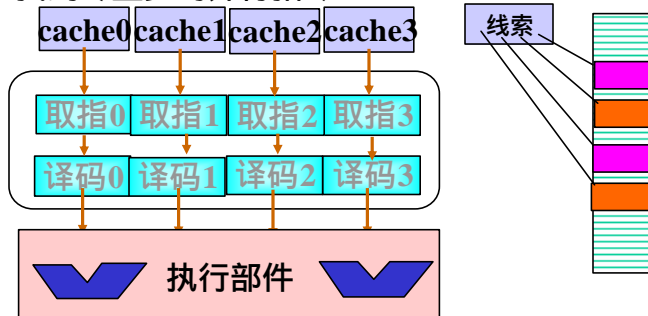
- (1) 对数据地址相关推测；Merced
- (2) 采用硬件算法解决数据相关；Tomasulo
- (3) 从其它线索中提取并行指令。多线索



微处理器的执行机制设计

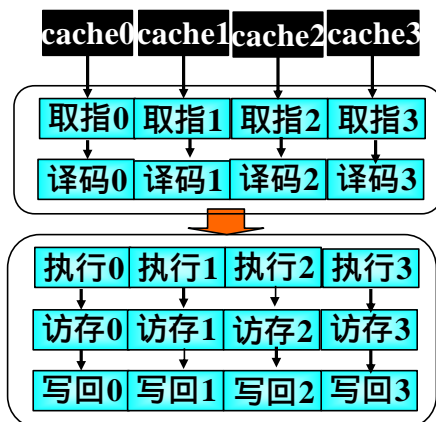
解决瓶颈之四：控制相关

多线索取指：找到尽量多的并行指令



多线索处理器举例

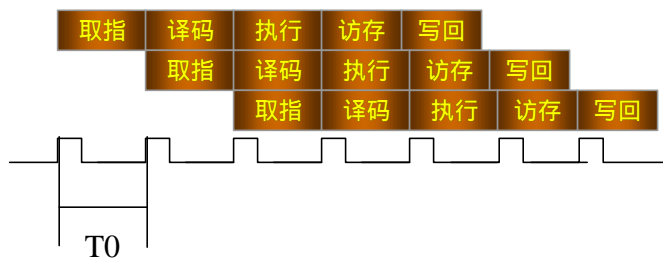
多线索处理器结构 MTPS结构(SMT结构)



微处理器流水线结构设计

- 流水深度
- 流水段的任务分配
- 流水线相关
- 时钟设计:周期,双时钟
- 寄存器与锁存器

流水深度



是否时钟周期随着流水深度的增加而永远减小呢？

流水段的任务分配

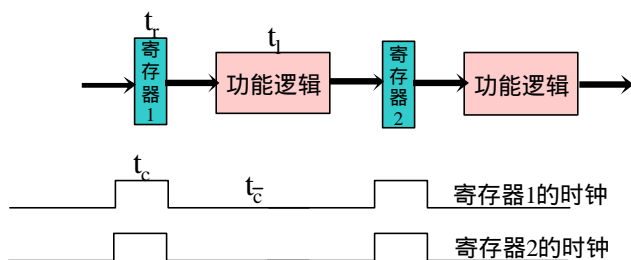


分配给每个流水段的部件时延小于时钟周期

流水线相关

流水线断流会导致其效率大大降低,流水段越多,其损失越大.P4的Netburst结构.

时钟设计



时间满足:

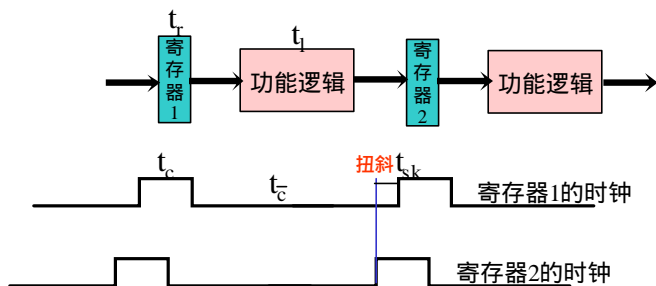
$$t_c \geq t_r$$

$$t_c \leq t_r + t_l \quad t_l \leq t_c - t_r$$

$$t_c + t_c \geq t_r + t_l$$

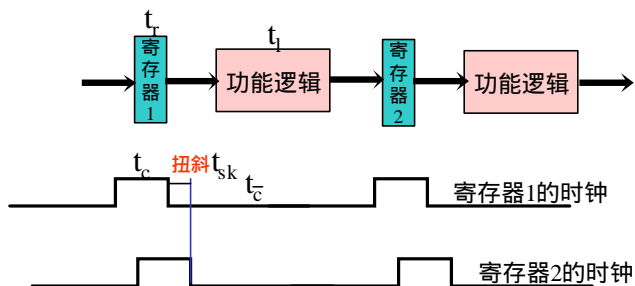
时钟设计:时钟扭斜

时钟负扭斜



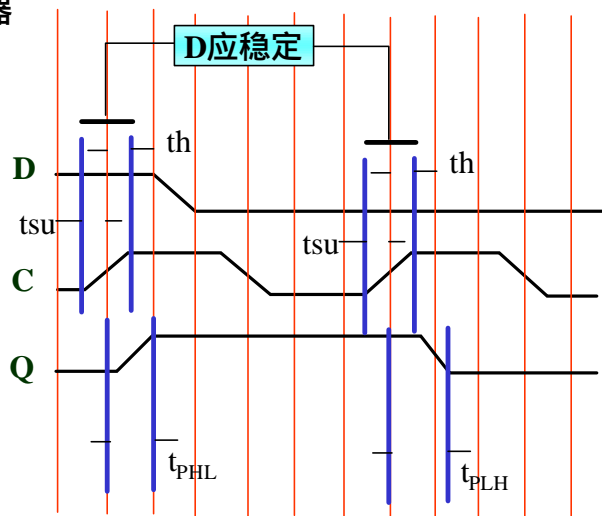
时钟设计:时钟扭斜

时钟正扭斜



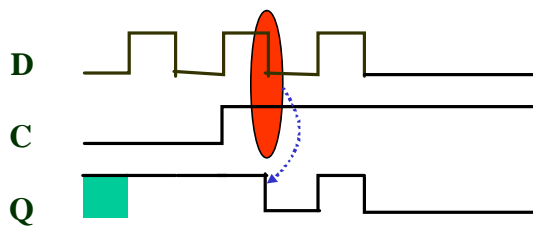
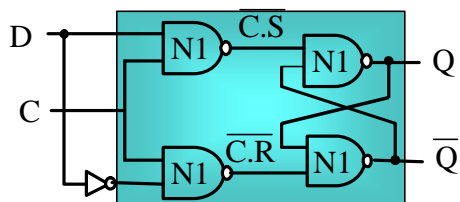
寄存器与锁存器

D触发器



锁存器

逻辑结构图

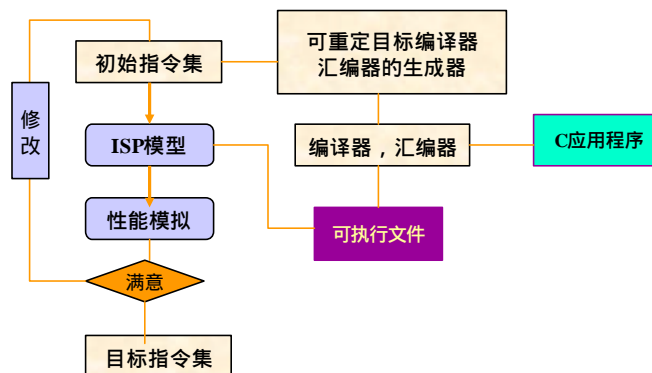


微处理器逻辑设计与实现

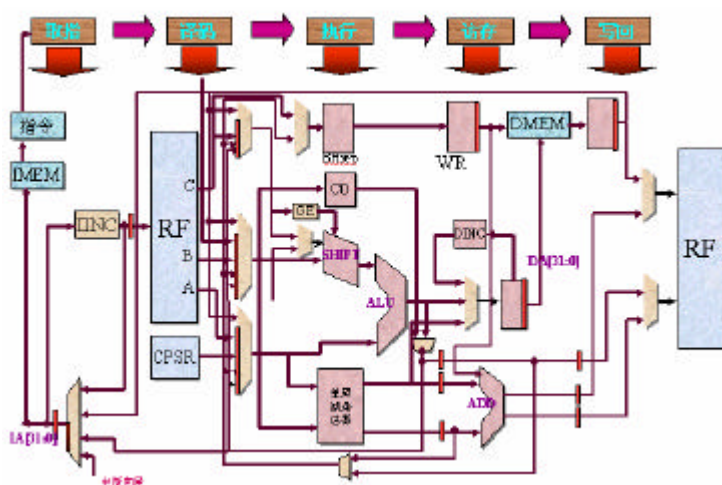
- 处理器结构级评价工具总结
- 寄存器级结构图
- 指令逻辑寄存器级描述
- 指令物理寄存器级描述
- 数据通路设计
- 控制器设计
- 中断控制设计
- 处理器集成

处理器结构级评价工具总结:指令级模拟

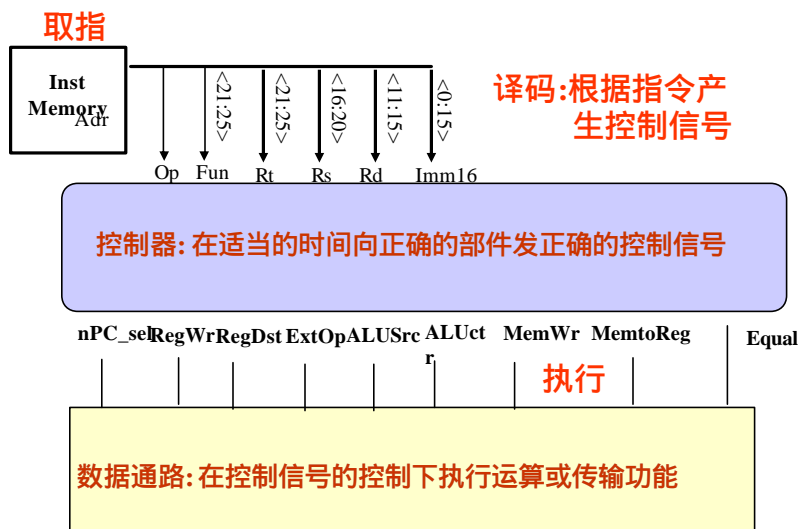
功能设计：确定指令系统



寄存器级结构图



微处理器设计



处理器逻辑设计:数据通路和控制器

1. 分析指令系统 => 数据通路要求

寄存器传输级指令描述:指令功能的逻辑描述

存储单元设计

数据通路应支持指令描述的所有功能和寄存器传输

2. 确定数据通路的部件集及其时序控制方法, 如寄存器堆控制

3. 确立满足以上要求的数据通路框图

4. 分析每条指令的实现确定其控制信号

指令物理寄存器传输级描述:

确定指令执行时每个流水段需要的控制信号

5. 完成数据通路的逻辑描述(VHDL.etc)

5. 分析中断机制

6. 设计并实现控制器

处理器逻辑设计

数据通路相关问题:

- 寄存器堆大小
- ALU的功能
- 立即数的产生
- 如何读/写内存
- 寄存器堆/ALU的带宽

控制器相关问题:

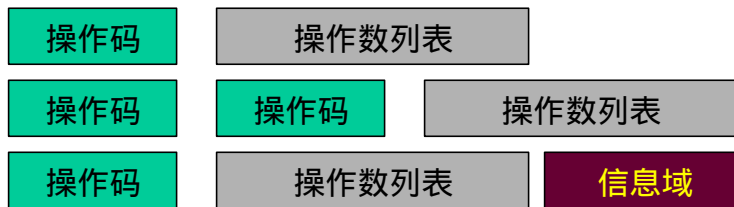
- 指令如何译码
- 控制信号的发生及其时序
- 指令编码压缩

处理器逻辑设计

指令的含义:

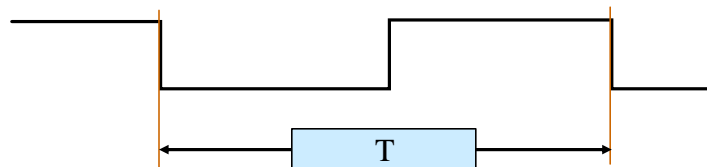
一般含义:指明要执行的操作以及操作的对象.

EPIC:显式并行指令计算.指明指令的可并行情况,包括可并行的指令,以及为指令级并行提供静态或动态信息(支持静态和动态调度).如VLIW,IA-64.



一个简单处理器设计举例：单周期MIPS

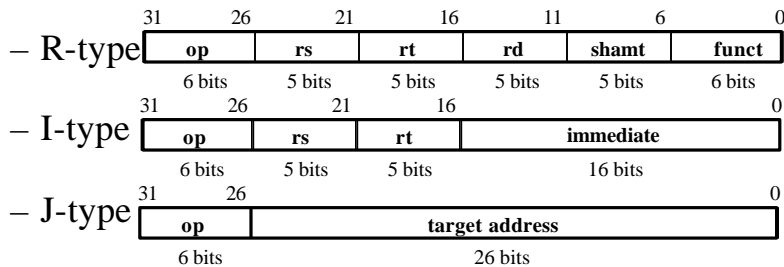
单周期的概念：指令的所有执行过程都在一个周期内完成。
包括：取指，译码，执行，访存等。



MIPS处理器：指令规整,硬件设计简化,流水线简单,很多优化依靠编译完成.

指令逻辑寄存器级描述

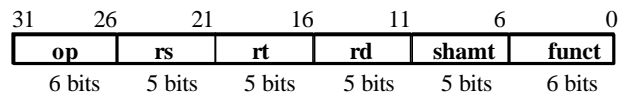
所有MISP指令32位长,三类指令格式:



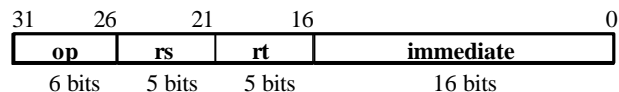
op: 指令操作
rs, rt, rd: 原操作数, 目标操作数, 目的操作数
shamt: 移位数量
funct: 选择操作码的种类
address / immediate: 地址位移或立即数
target address: 转移指令的目标地址.

指令逻辑寄存器级描述

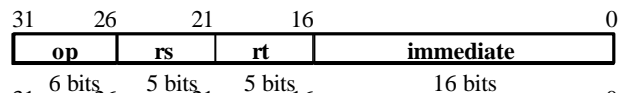
- ADD, SUB
 - addU rd, rs, rt
 - subU rd, rs, rt



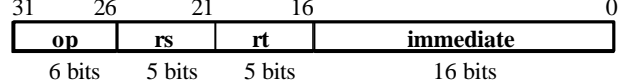
- OR Immediate:
 - ori rt, rs, imm16



- LOAD and STORE Word
 - lw rt, rs, imm16
 - sw rt, rs, imm16



- BRANCH:
 - beq rs, rt, imm16



指令逻辑寄存器传输级描述

- 逻辑寄存器传输级给出指令每流水段的功能
- 从取指开始

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst **Register Transfers**

```
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
```

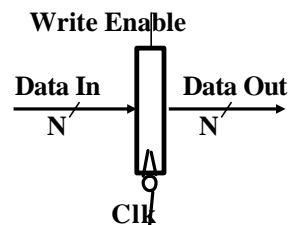
```
LOAD      R[rt] <- MEM[ R[rs] + sign_ext(Imm16)]; PC <- PC + 4
```

```
STORE    MEM[ R[rs ] + sign_ext(Imm16) ] <- R[rt]; PC <- PC + 4
```

```
BEQ      if ( R[rs] == R[rt] ) then PC <- PC + sign_ext(Imm16) || 00
          else PC <- PC + 4
```

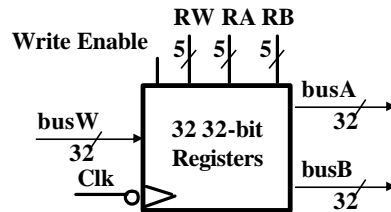
数据通路部件

- 组合单元
 - Adder (32 bit)
 - Multiplexer(32 bit)
 - ALU (32 bit)
- 存储单元
 - Register
 - D触发器
 - N-位输入/输出
 - 写使能
 - 写使能
 - 0: 输出不变
 - 1: 输入给输出
 - 时钟方法



存储单元设计

- 寄存器堆由32位寄存器组成：
 - 两路32-位输出总线(内部总线):
A 和 B
 - 一路32-位输入总线: W

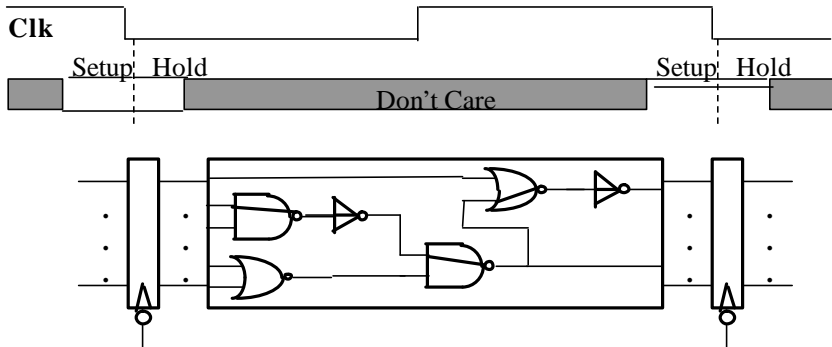


- 寄存器的选择：
 - RA (数) 选择发送给busA 的寄存器内容
 - RB (数) 选择发送给busB 的寄存器内容
 - RW (数) 选择被写的寄存器号
- 写时写使能为“1”

存储单元设计

- 时钟输入(CLK)
 - 只有写操作时时钟才起作用
 - 读操作时,寄存器堆类似组合模块:
 - busA or busB 通过一段访问时间后有效

时钟设计



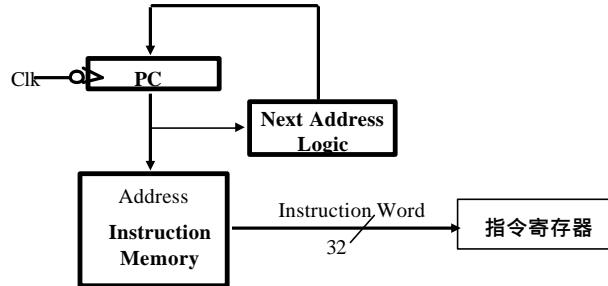
- 所有的记忆单元都使用时钟的相同边沿
- $\text{Cycle Time} = \text{CLK-to-Q} + \text{Longest Delay Path} + \text{Setup}$

物理寄存器传输级描述

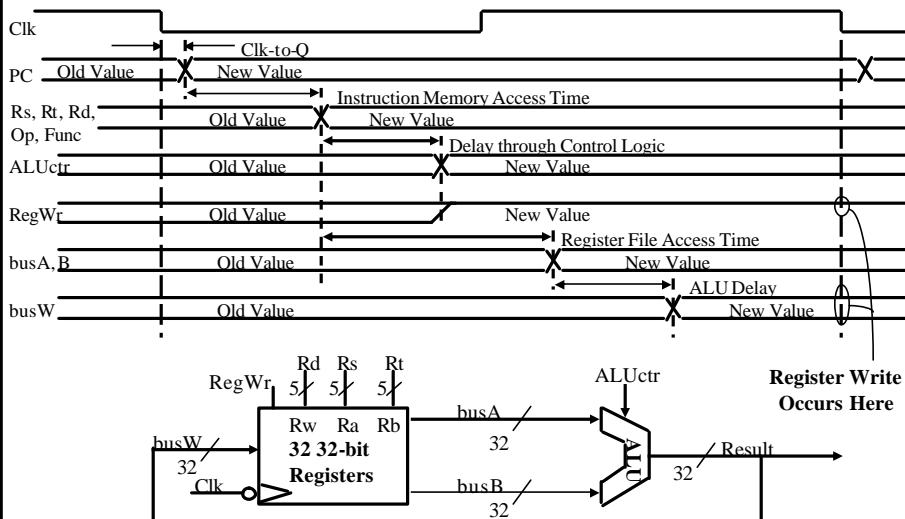
- 寄存器传输级→ 数据通路模块结构
- 取指
- 读操作数,执行操作

取指部件

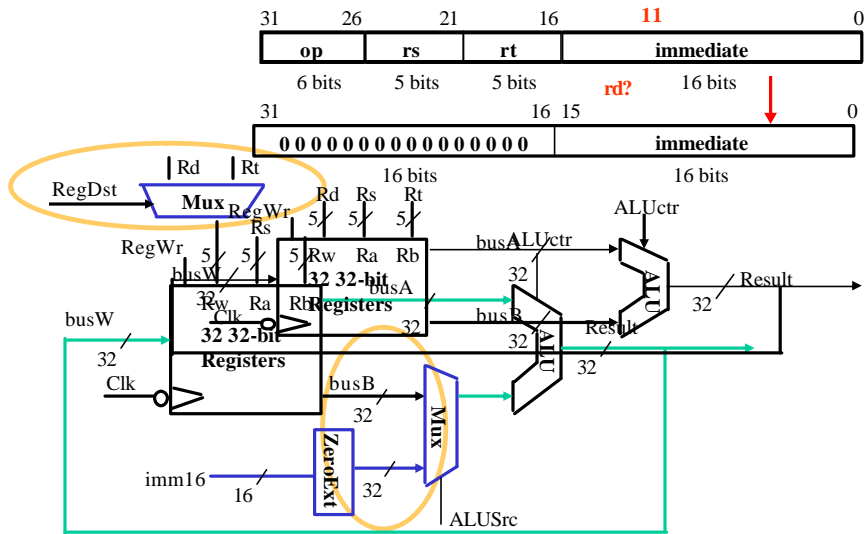
- 公共的RTL操作
 - Fetch the Instruction: $IR \leftarrow \text{mem}[\text{PC}]$
 - 更新程序计数器:
 - 顺序指令: $\text{PC} \leftarrow \text{PC} + 4$
 - 转移指令: $\text{PC} \leftarrow \text{转移地址/顺序地址}$



数据通路设计与实现：R-R指令时序

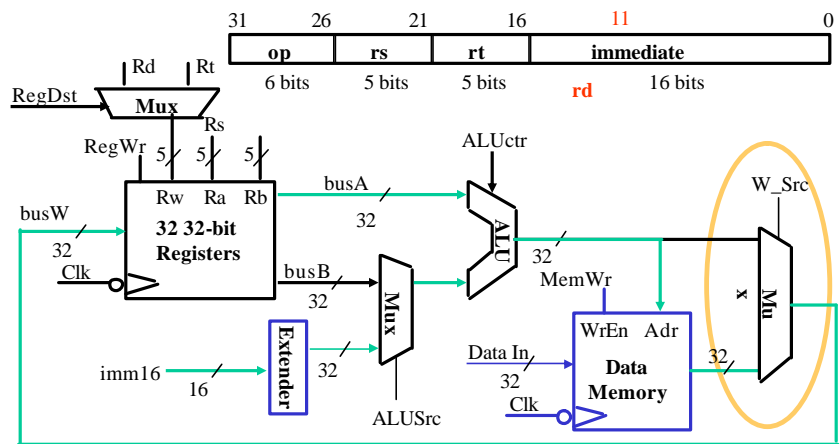


带立即数的逻辑操作



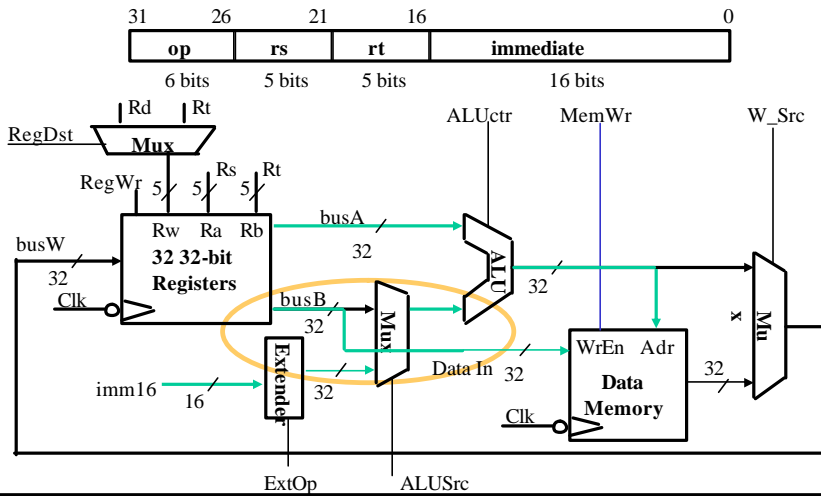
LOAD指令

- $R[\underline{rt}] \leftarrow \text{Mem}[R[\text{rs}] + \text{SignExt}[\text{imm16}]]$ Example: `lw rt, rs, imm16`



STORE指令

- Mem[R[rs] + SignExt[imm16] <- R[rt]] Example: sw rt, rs, imm16



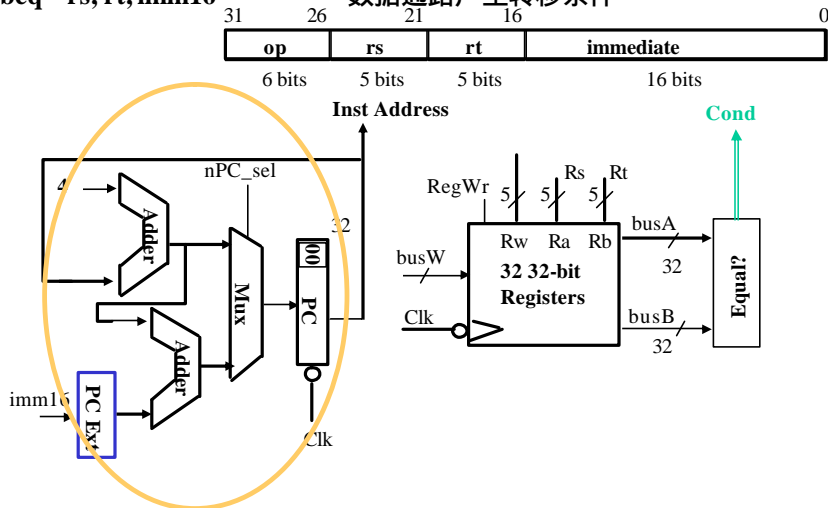
转移指令

- beq rs, rt, imm16
- 31 26 21 16 0
- op rs rt immediate
- 6 bits 5 bits 5 bits 16 bits
- mem[PC] 取指
 - Equal <- R[rs] == R[rt] 计算转移条件
 - if (COND eq 0) 计算下一条指令的地址
 - PC <- PC + 4 + (SignExt(imm16) x 4)
 - else
 - PC <- PC + 4

转移指令的数据通路

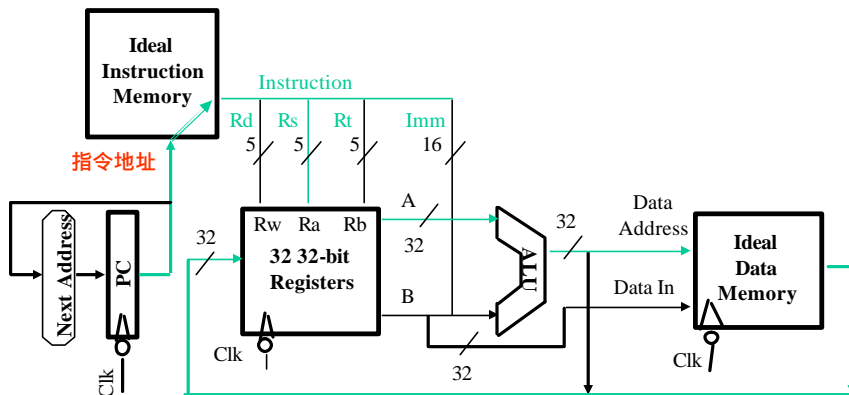
- `beq rs, rt, imm16`

数据通路产生转移条件



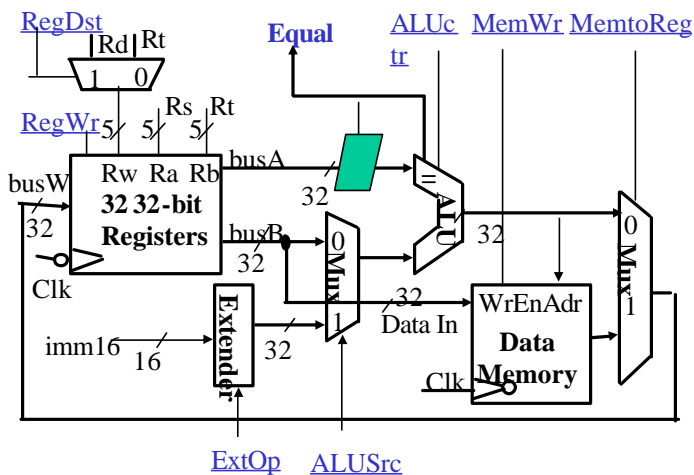
关键路径分析

- 关键路径(Load 操作) `lw rt, rs, imm16`
- =PC's Clk-to-Q + 指令访问时间 + 寄存器堆的访问时间 + ALU执行32位加的延时 + 数据存储访问时间 + 写寄存器堆的建立时间



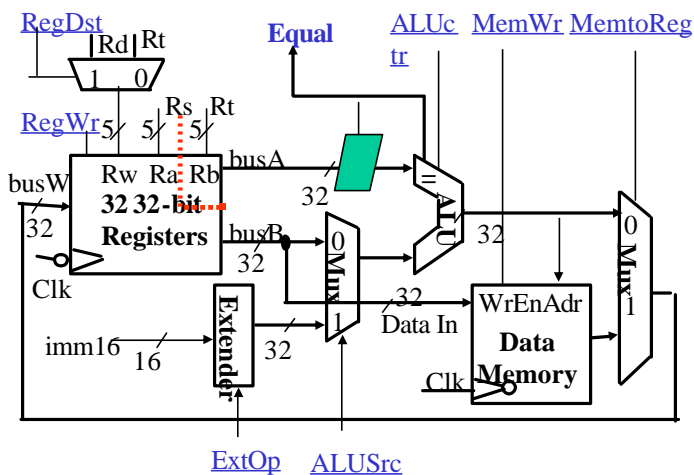
寻址方式与数据通路

例如：ADD r1,r2,[r3,ls1#4];功能：r1<-r2+r3*16;



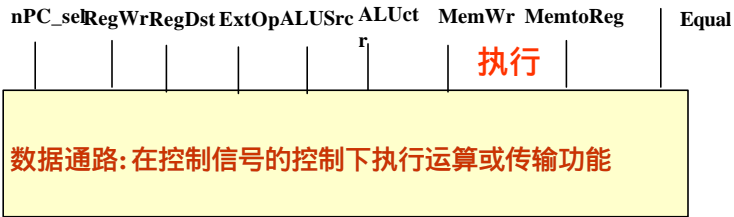
寻址方式与数据通路

例如：ADD r1,r2,[r3,ls1 r6];功能：r1<-r2+r3*r6;

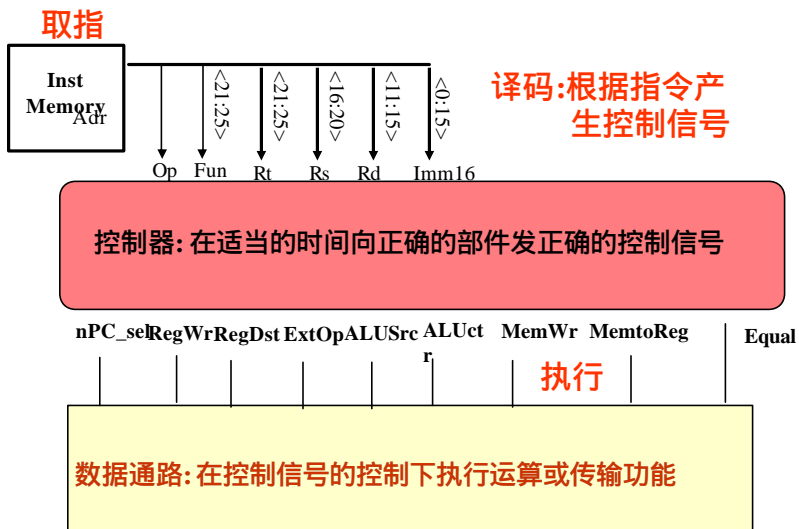


数据通路综合

通过以上每一条指令执行过程的分析，得到所有指令的逻辑寄存器传输级和物理寄存器传输级描述，将完成所有指令所需要的部件和信号综合在一起，形成以下的数据通路。



数据通路完成后：控制器实现



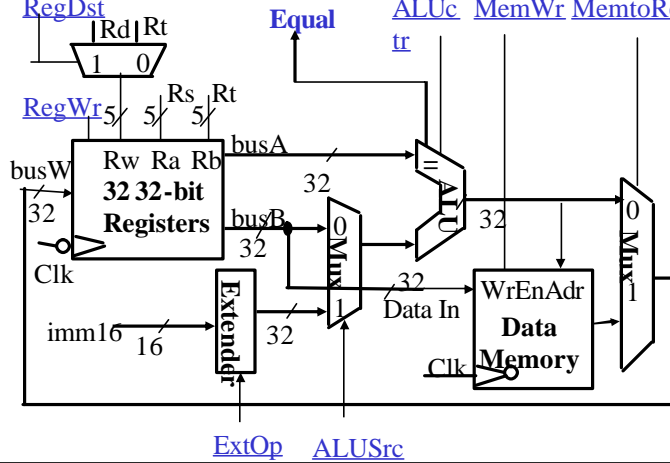


- [illegible]

控制信号的定义

° MemWr: 写内存 RegDst: 0 => "rt"; 1 => "rd"

° RegWr: 写目的寄存器 MemtoReg: 1 => Mem
RegDst ALUc MemWr MemtoReg

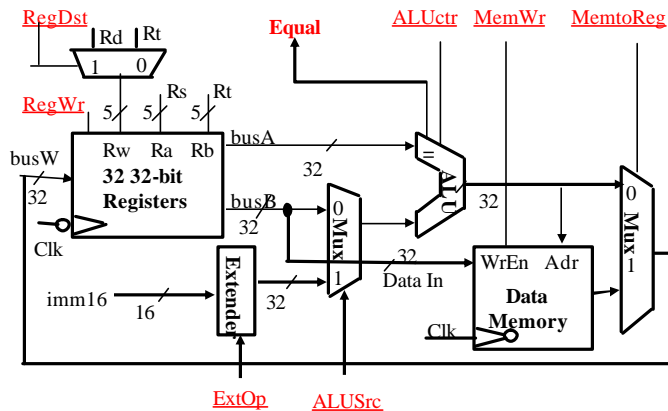


控制信号分析

指令 寄存器传输

ADD R[rd] <- R[rs] + R[rt]; PC <- PC + 4

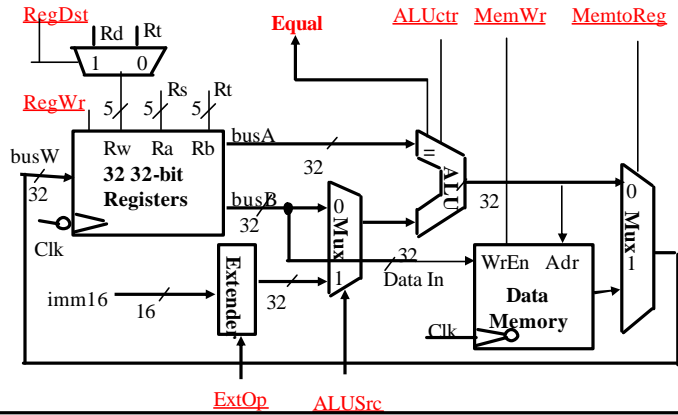
ALUSrc = RegB, ALUctr = "add", RegDst = rd, RegWr, nPC_sel = "+4"



控制信号分析

SUB $R[rd] \leftarrow R[rs] - R[rt]; \quad PC \leftarrow PC + 4$

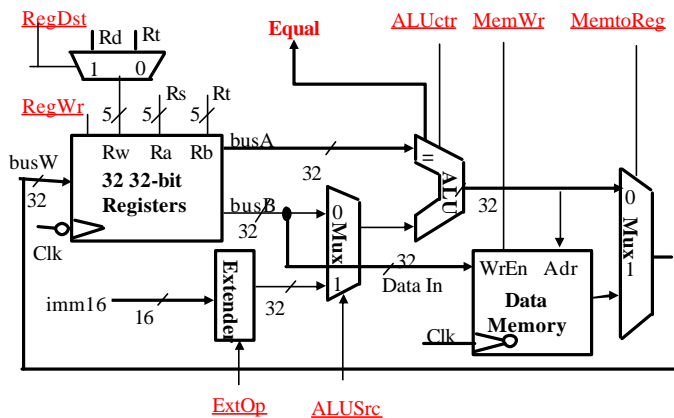
ALUSrc = RegB, ALUctr = "sub", RegDst = rd, RegWr, nPC_sel = "+4"



控制信号分析

ORI $R[rt] \leftarrow R[rs] \text{ or } \text{zero_ext}(\text{Imm16}); \quad PC \leftarrow PC + 4$

ALUSrc = Im, ExtOp = "Z", ALUctr = "or", RegDst = rt, RegWr, nPC_sel = "+4"

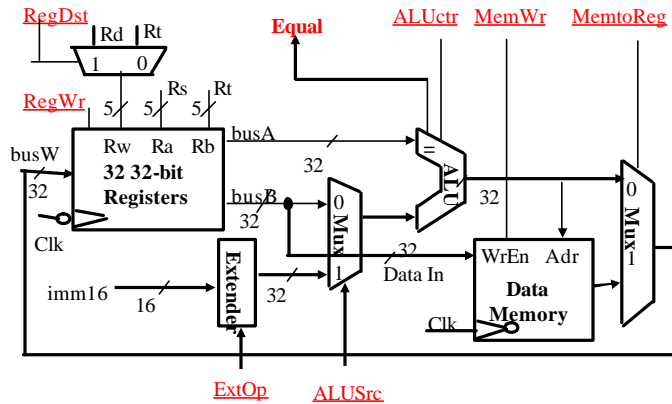


控制信号分析

LOAD $R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})];$

$PC \leftarrow PC + 4$

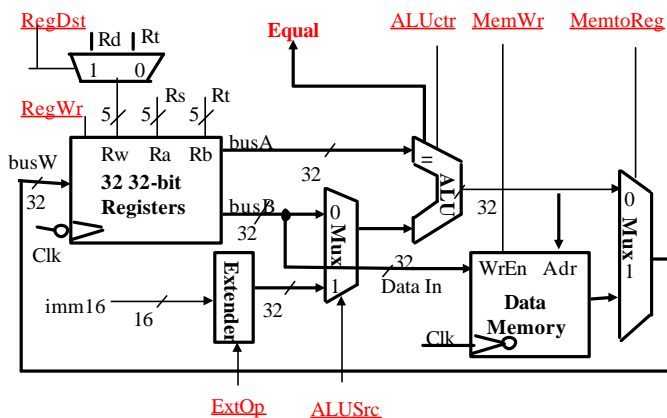
ALUSrc = Im, ExtOp = "Sn", ALUctr = "add", MemtoReg, RegDst = rt, RegWr, nPC_sel = "+4"



控制信号分析

STORE $\text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})] \leftarrow R[rt];$ $PC \leftarrow PC + 4$

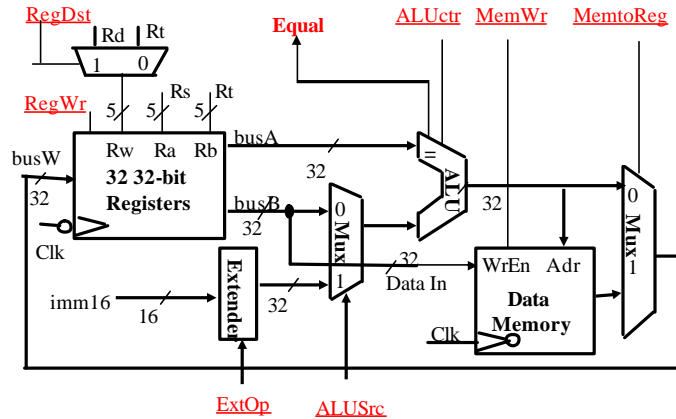
ALUSrc = Im, ExtOp = "Sn", ALUctr = "add", MemWr, nPC_sel = "+4"



控制信号分析

BEQ if (R[rs] == R[rt]) then PC <- PC + sign_ext(Imm16) || 00 else

PC <- PC + 4 **nPC_sel = EQUAL, ALUctr = "sub"**



控制信号总结

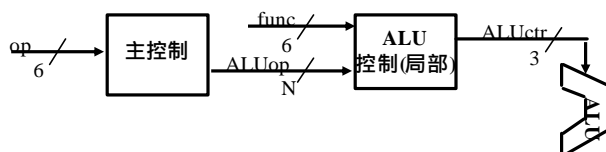
Appendix A

	func		We Don't Care :-)				
	op						
	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
nPCsel	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



译码器设计:局部译码概念

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<N:0>	"R-type"	Or	Add	Add	Subtract	xxx

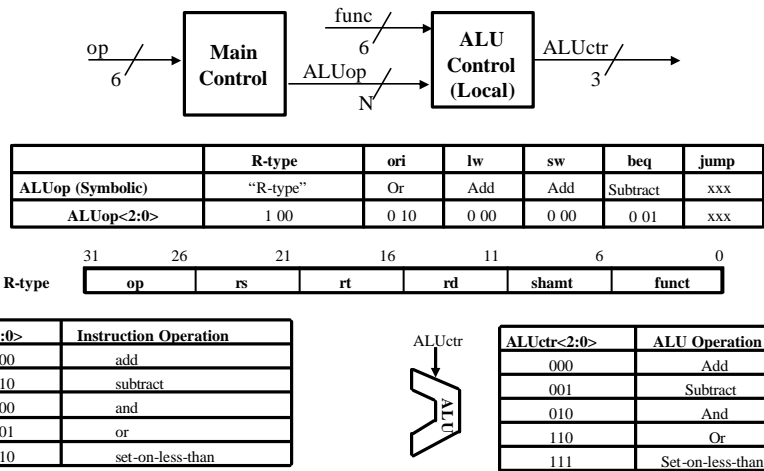


ALU操作码编码

- ALUop操作码用两位bit来表示:
 - (1) "R-type" instructions
 - 执行ALU操作的"I-type"指令:
 - (2) Or, (3) Add, (4) Subtract
- 若实现MIPS指令全集,ALU操作码需用3位表示:
 - (1) "R-type" 指令
 - 执行ALU操作的"I-type"指令:
 - (2) Or, (3) Add, (4) Subtract, (5) And

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

FUNC域的译码



ALU控制信号的真值表

						func<3:0>		Instruction Op.		
ALUop (Symbolic)		R-type	ori	lw	sw	beq	0000	add		
		"R-type"	Or	Add	Add	Subtract	0010	subtract		
ALUop<2:0>		1 00	0 10	0 00	0 00	0 01	0100	and		
							0101	or		
							1010	set-on-less-than		

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

ALU控制信号(2)的逻辑表达式

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

- $$\text{ALUctr}<2> = \text{!ALUop}<2> \ \& \ \text{ALUop}<0> +$$

$$\text{ALUop}<2> \ \& \ \text{!func}<2> \ \& \ \text{func}<1>$$

$$\& \ \text{!func}<0>$$

ALU控制信号(1)逻辑等式

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

- $$\text{ALUctr}<1> = \text{!ALUop}<2> \ \& \ \text{!ALUop}<0> +$$

$$\text{ALUop}<2> \ \& \ \text{!func}<2> \ \& \ \text{!func}<0>$$

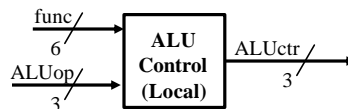
ALU控制信号(0)逻辑等式

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

- $$\begin{aligned} \text{ALUctr}<0> &= \text{!ALUop}<2> \& \text{ALUop}<1> \\ &+ \text{ALUop}<2> \& \text{!func}<3> \& \text{func}<2> \& \text{!func}<1> \\ &\& \text{func}<0> \\ &+ \text{ALUop}<2> \& \text{func}<3> \& \text{!func}<2> \& \text{func}<1> \\ &\& \text{!func}<0> \end{aligned}$$

ALU的控制模块

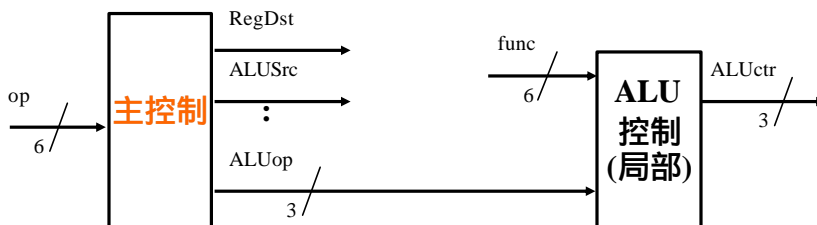
- $$\text{ALUctr}<2> = \text{!ALUop}<2> \& \text{ALUop}<0> + \text{ALUop}<2> \& \text{!func}<2> \& \text{func}<1> \& \text{!func}<0>$$
- $$\text{ALUctr}<1> = \text{!ALUop}<2> \& \text{!ALUop}<0> + \text{ALUop}<2> \& \text{!func}<2> \& \text{!func}<0>$$
- $$\text{ALUctr}<0> = \text{!ALUop}<2> \& \text{ALUop}<1> + \text{ALUop}<2> \& \text{!func}<3> \& \text{func}<2> \& \text{!func}<1> \& \text{func}<0> + \text{ALUop}<2> \& \text{func}<3> \& \text{!func}<2> \& \text{func}<1> \& \text{!func}<0>$$



每个控制信号的逻辑：行为级描述

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUSrc <= if (OP == "Rtype") then "regB" else "immed"
- ALUctr <= if (OP == "Rtype") then **func**
elseif (OP == ORi) then "OR"
elseif (OP == BEQ) then "sub"
else "add"
- ExtOp <= if (OP == ORi) then "zero" else "sign"
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

主控逻辑的真值表



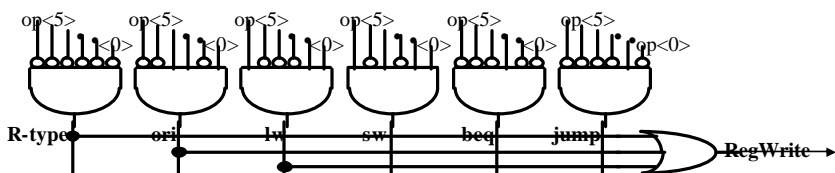
主控逻辑的真值表

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>	0	1	0	0	0	x
ALUOp <0>	0	0	0	0	1	x

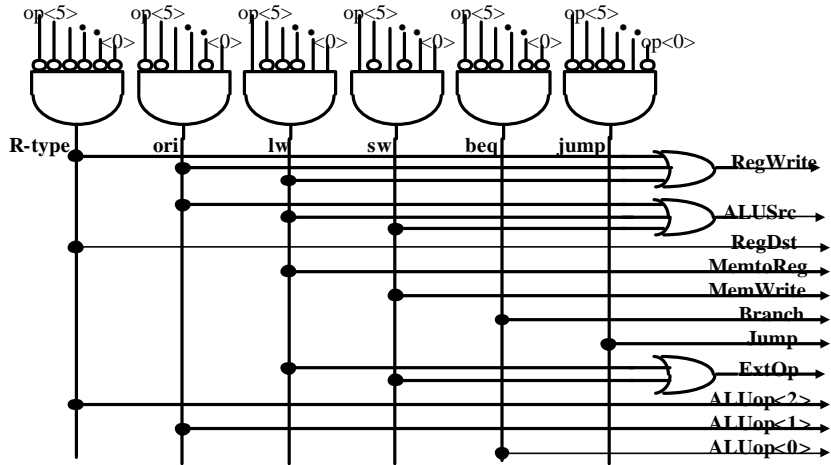
RegWrite的真值表：结构级描述

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

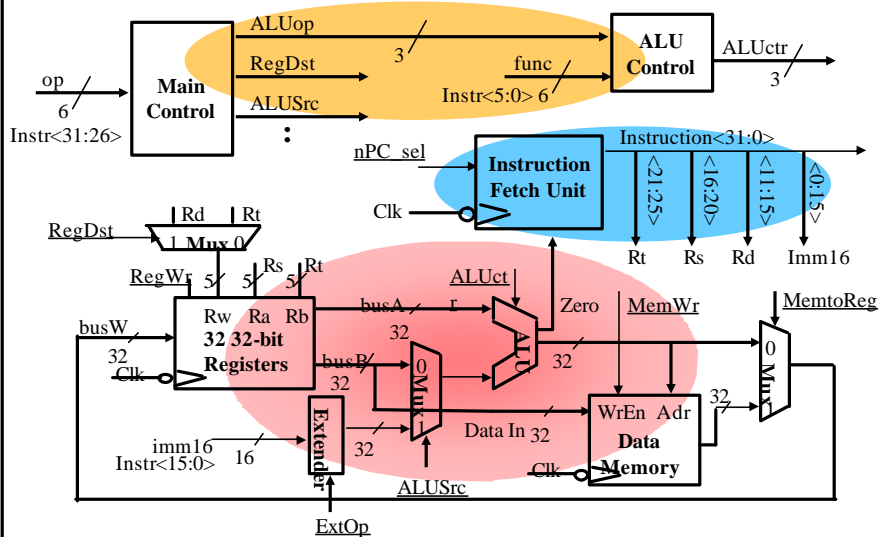
- RegWrite = R-type + ori + lw
 - = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
 - + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
 - + op<5> & !op<4> & !op<3> & !op<2> & !op<1> & op<0> (lw)



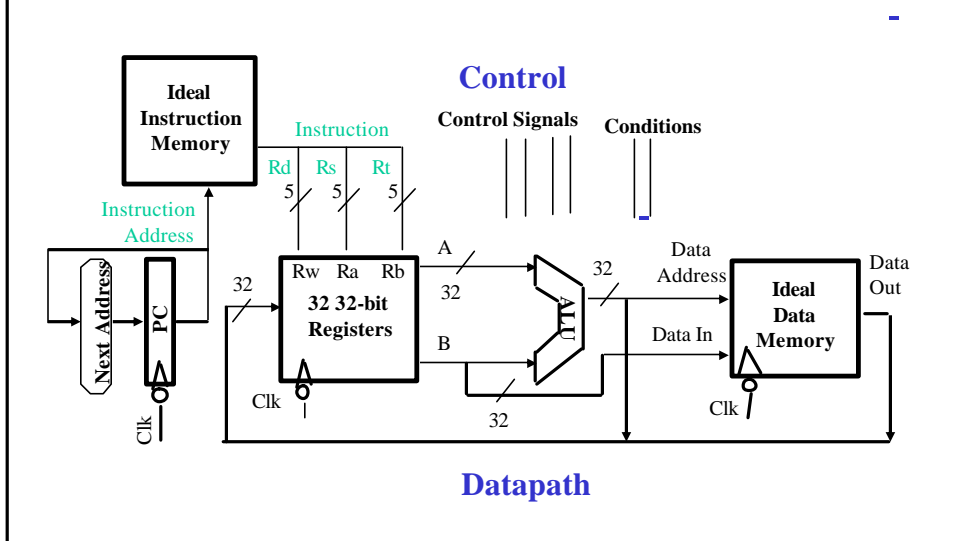
控制信号的实现



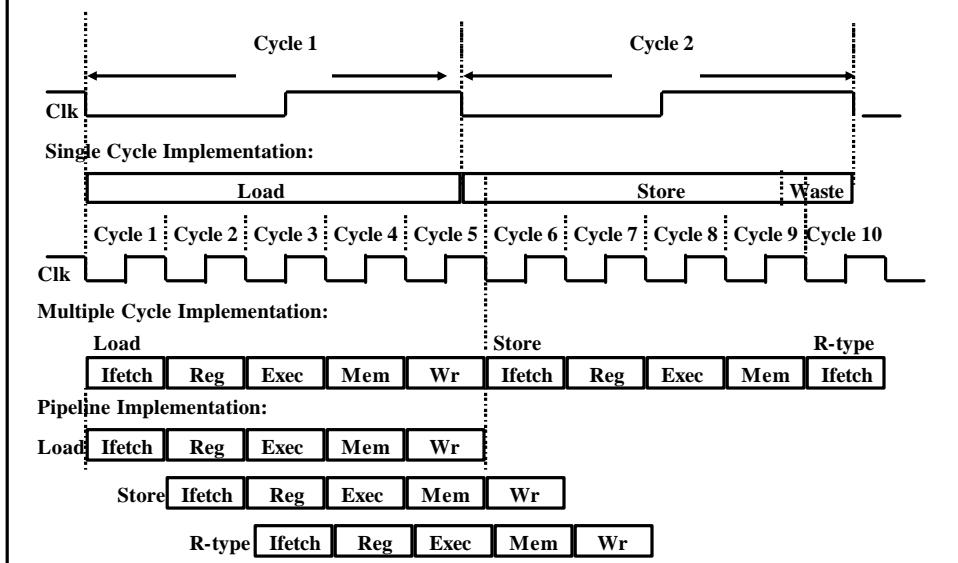
一个完整的单周期处理器



单周期MIPS处理器结构



单周期,多周期与流水线



行为级/逻辑级验证

功能级验证

 确保处理器与系统结构一致

等价验证

 确保设计结果与功能模型相匹配

电子验证

 确保电路的正确性，在运行电压温度等环境下的稳定性

流片前验证

流片后验证

验证的层次

系统验证

 芯片上系统验证

行为级验证

 验证所实现的与所描述的一致性

寄存器传输级

 结构验证

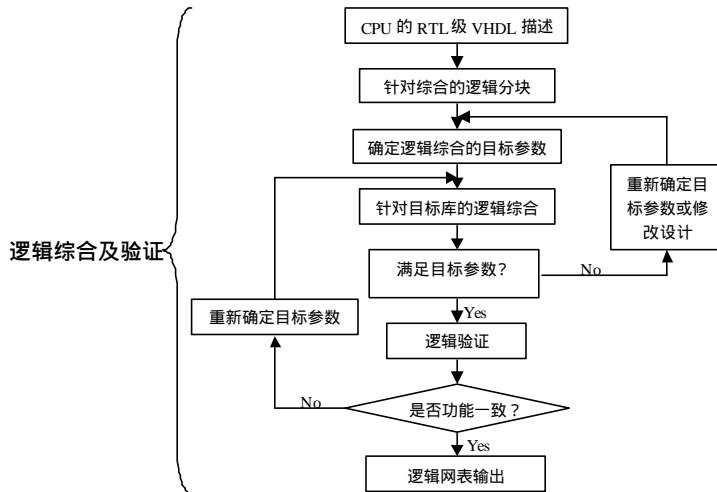
 模拟验证

 模型检验

门级/晶体管级

 逻辑等式检验

逻辑综合



逻辑综合

Translation

HDL源代码（行为级/结构级）
转换成布尔表示

Optimization

根据约束条件进行逻辑时序优化

Mapping

利用工艺库中的单元实现约束
条件下的设计

工艺库

```
Cell ( AND2_3) {  
    area: 8.000 ;  
    pin (Y) {  
direction : output;  
timing () {  
related_pin: "A";  
timing_sense: positive_unate;  
rise_propagation (drive_3_table_1){  
values =(0.2616,0.2608...)  
}  
function: "(A & B)";  
...  
pin (A) {  
direction : input;  
capacitance: 0.012000;  
}  
pin (B){  
direction : input;  
capacitance: 0.010000;  
}  
}
```

单元名

单元面积

pinA->pin Y的延时

单元的功能

输入的电特性