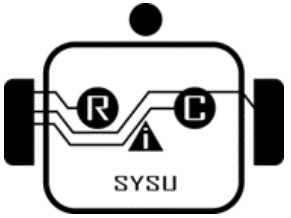
中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

第四章：MIPS CPU逻辑设计

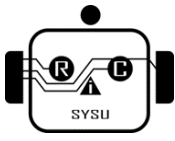
中山大学计算机学院
陈刚

2022年秋季

上节回顾

回顾内容

- 3.5 浮点数运算
  - 浮点数的表示(Floating-Point Representation)
  - 浮点数加/减法(Floating-Point Addition)
    - 步骤:求阶差、对阶、尾数加/减、规格化、舍入、判溢出
  - 浮点数乘/除法(Floating-Point Multiplication)
    - 步骤:阶码加/减、尾数乘/除、规格化、舍入、判溢出
  - MIPS浮点指令(MIPS Floating-Point Instructions)
    - lwcl和swcl、add.s、mult.s、l.d和s.d、add.d、mult.d等
  - 精确的算术运算(Accurate Arithmetic)
    - 浮点数的精确表示、浮点数的累积误差问题



本章概要

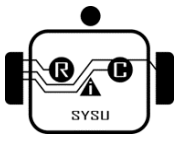
章节介绍

第四章 处理器设计

- 4.1 概述
- 4.2 逻辑设计规则
- 4.3 建立数据通路
- 4.4 单周期控制器的实现
- 4.5 多周期控制器的实现
- 4.6 流水线数据通路和控制
- 4.7 数据冒险：转发与阻塞
- 4.8 控制冒险
- 4.9 异常处理

主要内容

- 处理器概述
- 逻辑设计规则
  - 数据通路的功能及实现
    - 操作元件(组合逻辑)
    - 状态 / 存储元件(时序逻辑)
  - 数据通路的定时
- 单周期处理器
  - 数据通路
  - 控制器

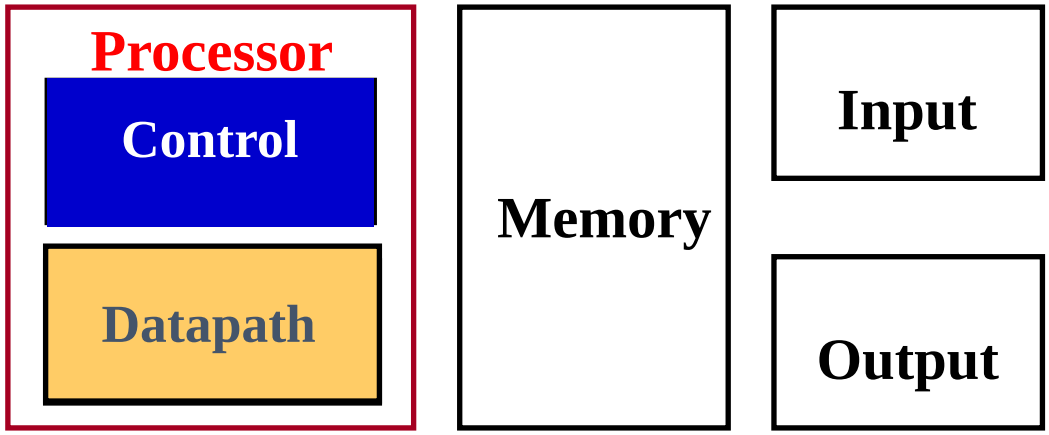


主要内容

- 处理器概述
- 逻辑设计规则
  - 数据通路的功能及实现
    - 操作元件(组合逻辑)
    - 状态 / 存储元件(时序逻辑)
  - 数据通路的定时
- 单周期处理器
  - 数据通路
  - 控制器

处理器概述

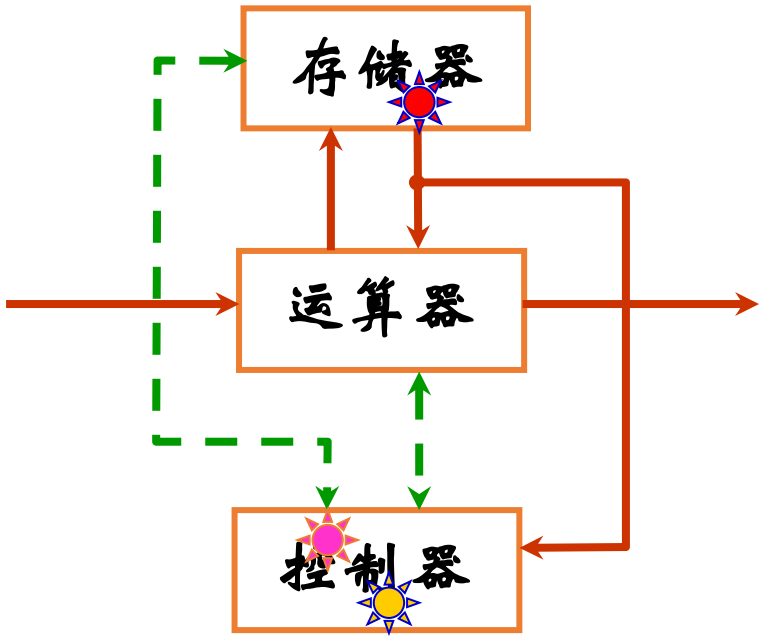
- 计算机的五大组成部分



- 什么是数据通路(DataPath)?(运算器)
  - 指令执行过程中，数据所经过的路径(包括路径中的部件)——指令的执行部件
- 控制器(Control)
  - 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——指令的控制部件

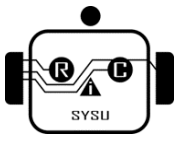
处理器概述

- 控制器的基本功能
  - 取指令(控制指令流出— PC)
  - 分析指令(控制指令分析— IR)
  - 执行指令，发出各种操作命令(控制指令执行— REGs和ALU)
  - 确定下一条指令的地址(控制指令流向—PC)
  - 执行环境的建立与保护(控制执行环境的维护—FLAGs/PSW)

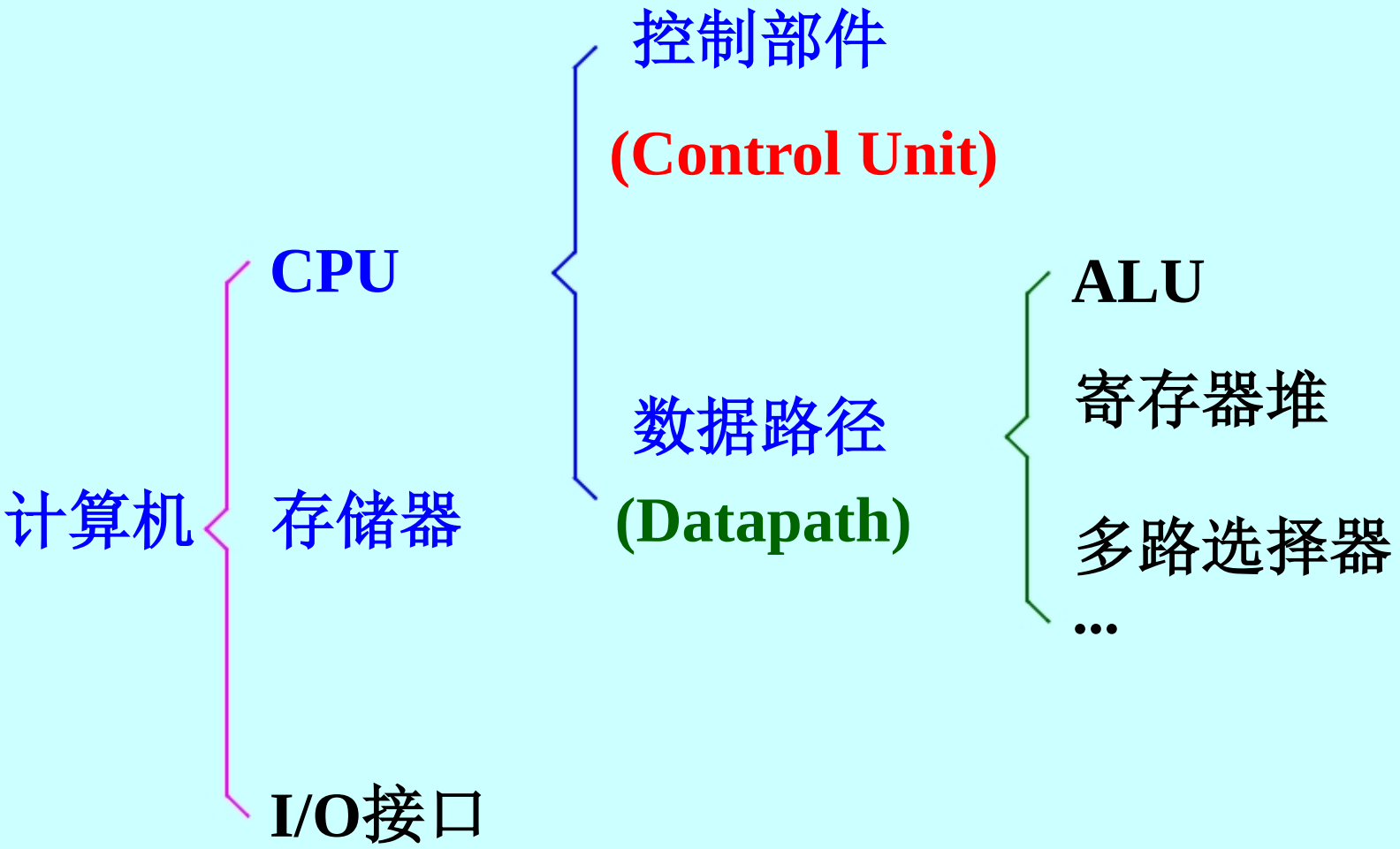


主要内容

- 处理器概述
- 逻辑设计规则
  - 数据通路的功能及实现
    - 操作元件(组合逻辑)
    - 状态 / 存储元件(时序逻辑)
  - 数据通路的定时
- 单周期处理器
  - 数据通路
  - 控制器

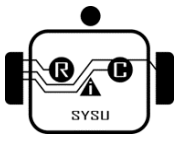


CPU的构成


计算机
CPU
存储器
I/O接口
控制部件
(Control Unit)
数据路径
(Datapath)
ALU
寄存器堆
多路选择器
...

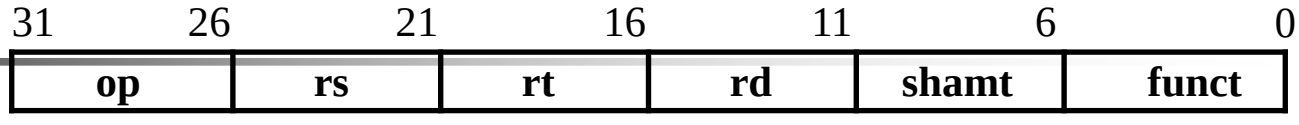
处理器设计: step-by-step

1. 分析指令集 => datapath 需求
   - 每个指令的含义都用寄存器传输级（RTL）给出
   - datapath 必须包含ISA所需要的各个寄存器
     - 为了实现功能还需要一些额外的寄存器
   - datapath 必须支持每一类寄存器转移操作
2. 选择datapath中包含的模块,确定这些模块的功能,时序
3. 组装datapath
4. 根据指令集决定datapath中需要的控制信号
5. 设计控制信号逻辑

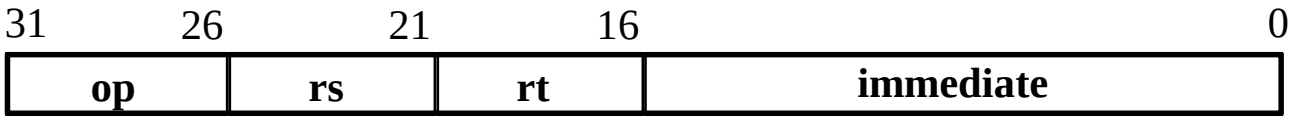


The MIPS Instruction Formats

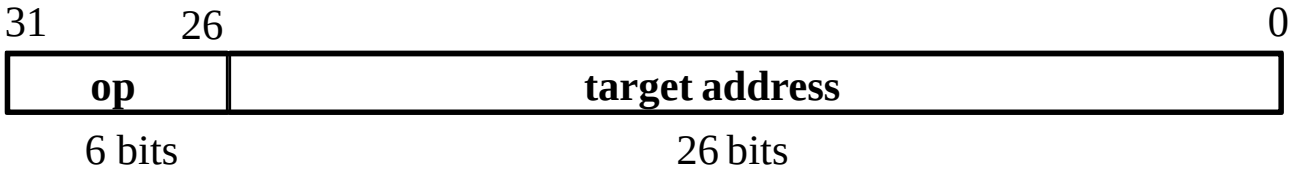
- 三类MIPS指令:
  - R-type



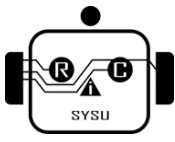
  - I-type



  - J-type



- 各个域分别是:
  - op: 指令操作
  - rs, rt, rd: 源,目的寄存器序号
  - shamt: 移位量
  - funct: 对op类操作，确定详细的操作类型
  - address / immediate: 立即数或者地址
  - target address: 目标地址

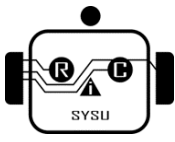


Step 1b: 从指令分析datapath组成

- 存储器
  - 一个指令存储器、一个数据存储器
- 寄存器(32 x 32bit)
  - 可以为 RS提供读取
  - 可以为 RT提供读取
  - 可以为RT或者RD提供写入端口
- PC(程序记数)
- 扩展电路: 为立即数提供
- 加法/减法等运算：两个寄存器之间的或者是一个寄存器一个经过扩展后的立即数之间
- PC更新电路：可以加4或者加上一个立即数扩展

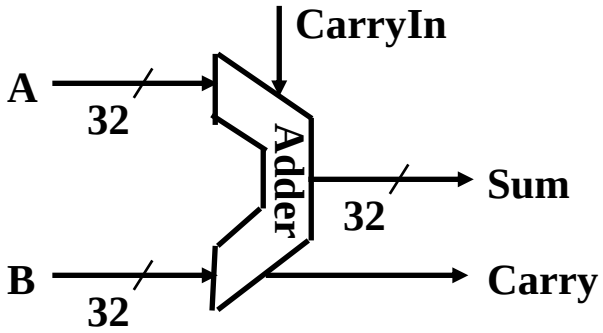
Step 2: datapath组成

- 组合逻辑电路部分
- 存储单元(“状态”)

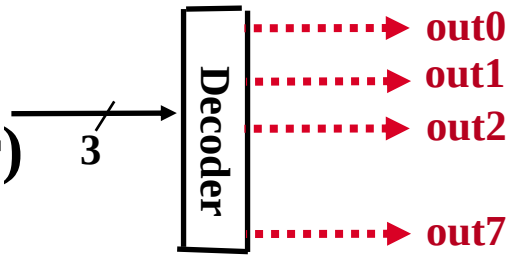


操作元件：组合逻辑电路

加法器
（Adder）



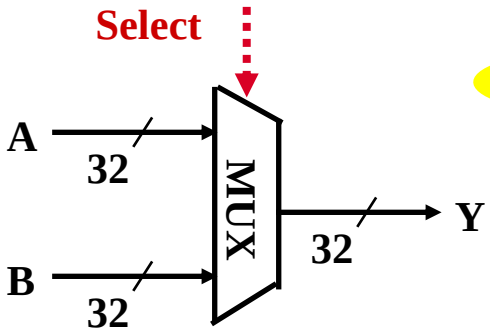
译码器
(Decoder)



控制信号

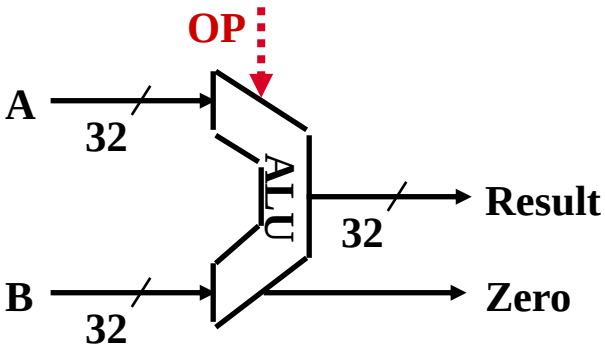
多路选择器
（MUX）

二选一
或多选一



何时要用到adder, ALU, MUX or Decoder?

算术逻辑部件（ALU）

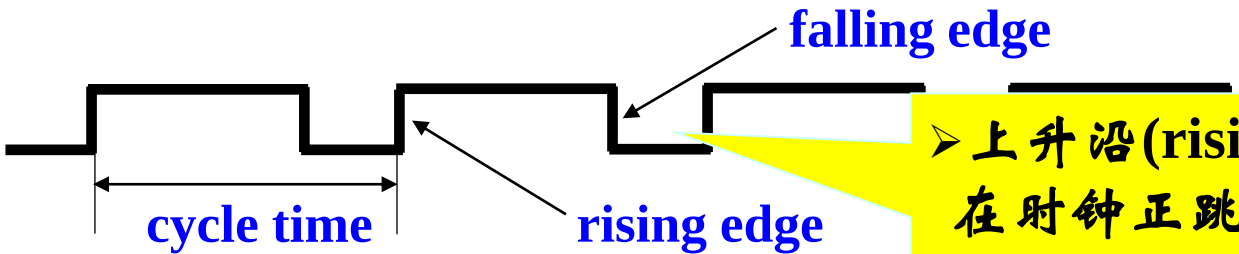


组合逻辑元件的特点：

- 其输出只取决于当前的输入
- 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

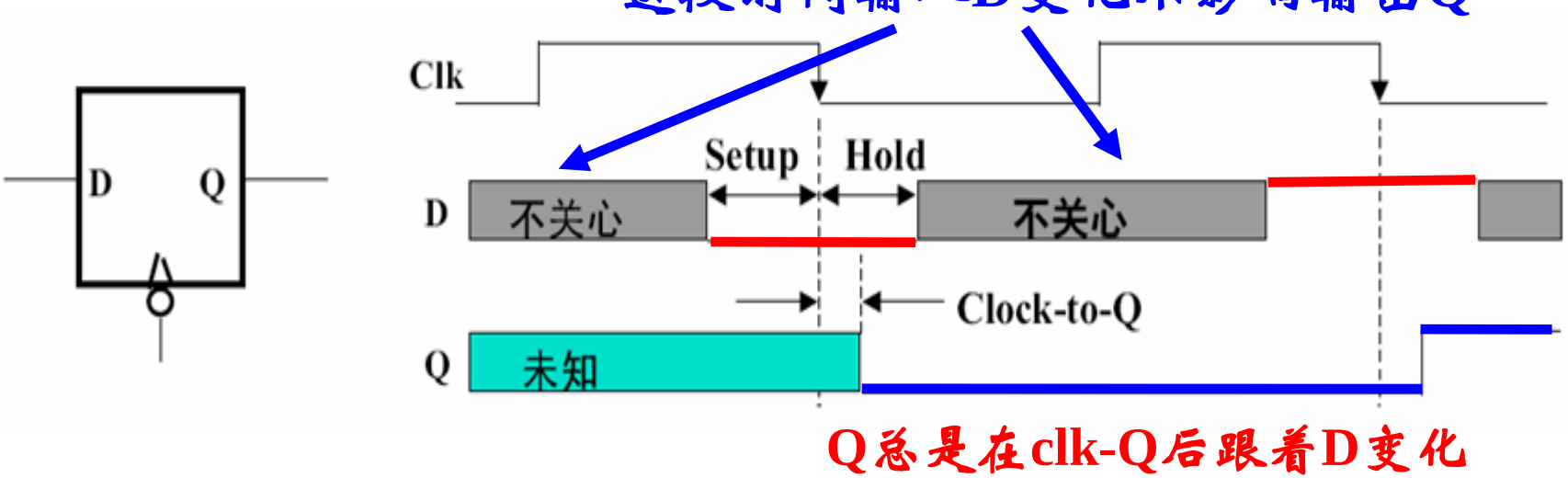
- 状态(存储)元件的特点
  - 具有存储功能，在时钟控制下输入状态被写到电路中，直到下一个时钟到达
  - 输入端状态由时钟决定何时写入，输出端状态随时可读出
- 定时方式：规定信号何时写入状态元件或何时从状态元件读出
  - 边沿触发(edge-triggered)方式
    - 状态单元中的值只在时钟边沿改变。每个时钟周期改变一次



- 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
  - D触发器：一个时钟输入、一个状态输入、一个状态输出

回顾——D触发器



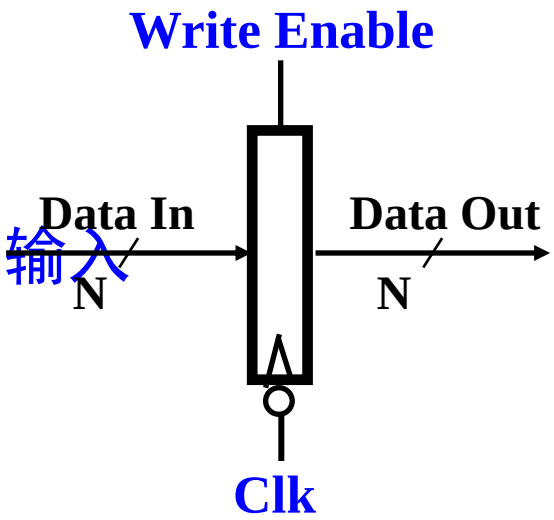
- 建立时间(Set Time)：在触发时钟边沿之前输入必须稳定
- 保持时间(Hold Time)：在触发时钟边沿之后输入必须保持
- Clock-to-Q-time：在触发时钟边沿，输出并不能立即变化

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)和存储器

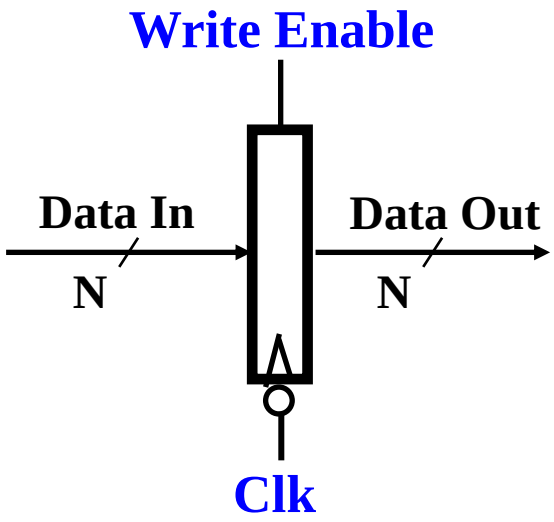
存储元件: 寄存器和寄存器组

- 寄存器(Register)
  - 写使能(Write Enable-WE)信号
    - 0：时钟边沿到时，输出不变
    - 1：时钟边沿到时，输出开始变为前面的输入
  - 若每个时钟边沿都写入，则不需WE信号

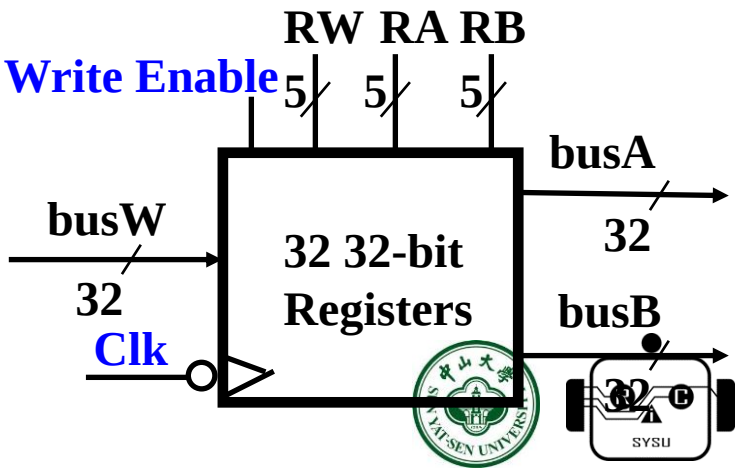


存储元件: 寄存器和寄存器组

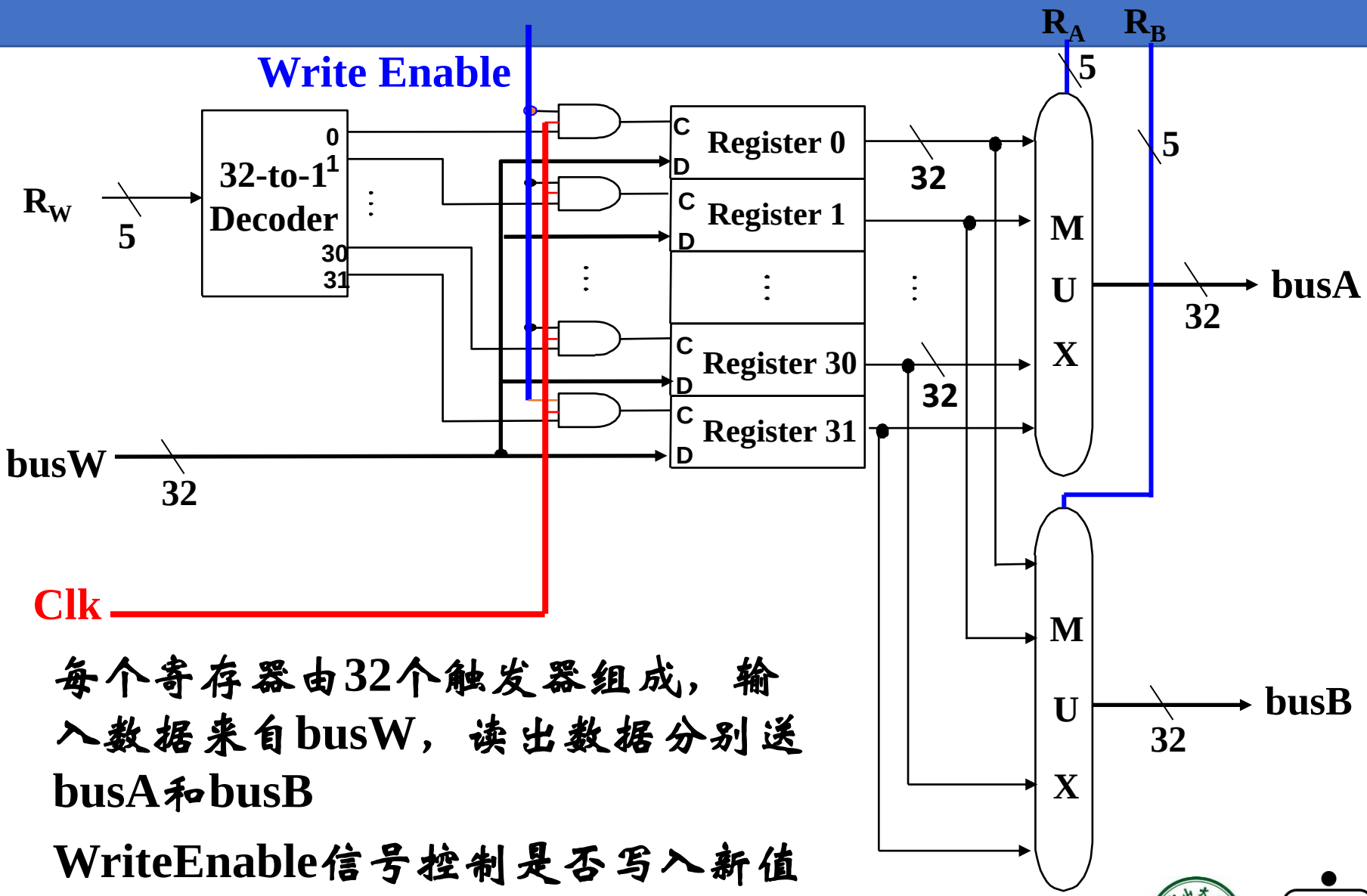
- 寄存器(Register)
  - 写使能(Write Enable-WE)信号
    - 0：时钟边沿到时，输出不变
    - 1：时钟边沿到时，输出开始变为输入
  - 若每个时钟边沿都写入，则不需WE信号



- 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且时钟边沿到时，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



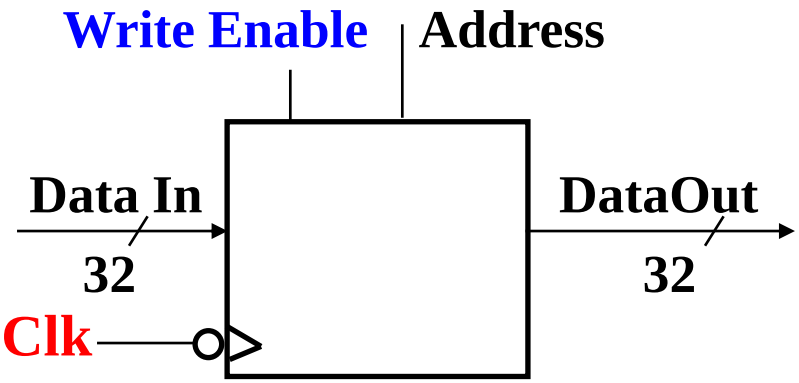
每个寄存器由32个触发器组成，输入数据来自busW，读出数据分别送busA和busB

WriteEnable信号控制是否写入新值

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: 理想存储器

- 理想存储器
  - Data Out：32位读出数据
  - Data In：　32位写入数据
  - Address：读写共用一个32位地址



- 读(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(时序逻辑操作)：写使能为1且时钟Clk边沿到来，Data In传来的值开始被写入Address指定的存储单元

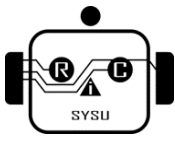
（为简化数据通路操作的说明，在此把存储器简化为带时钟信号Clk的理想模型。）

逻辑设计规则

- 数据通路由两类部件组成
  - 组合逻辑元件(亦称操作元件)（ALU）
  - 存储元件(亦称状态元件)（寄存器，指令存储器，数据存储器）
- 元件之间的连接方式
  - 总线式连接
  - 分散式连接

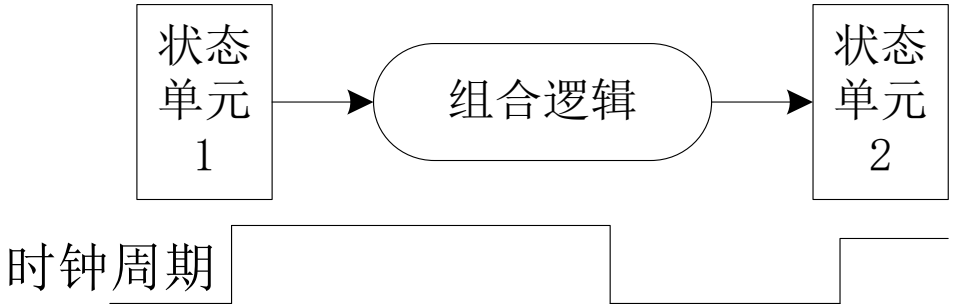
逻辑设计规则

- 数据通路由两类部件组成
  - 组合逻辑元件(亦称操作元件)（ALU）
  - 存储元件(亦称状态元件)（寄存器，指令存储器，数据存储器）
- 元件之间的连接方式
  - 总线式连接
  - 分散式连接
- 数据通路的构成
  - 由“操作元件”和“存储元件”通过总线或分散方式连接而成
- 数据通路的功能
  - 进行数据存储、处理、传送



数据通路与时序控制

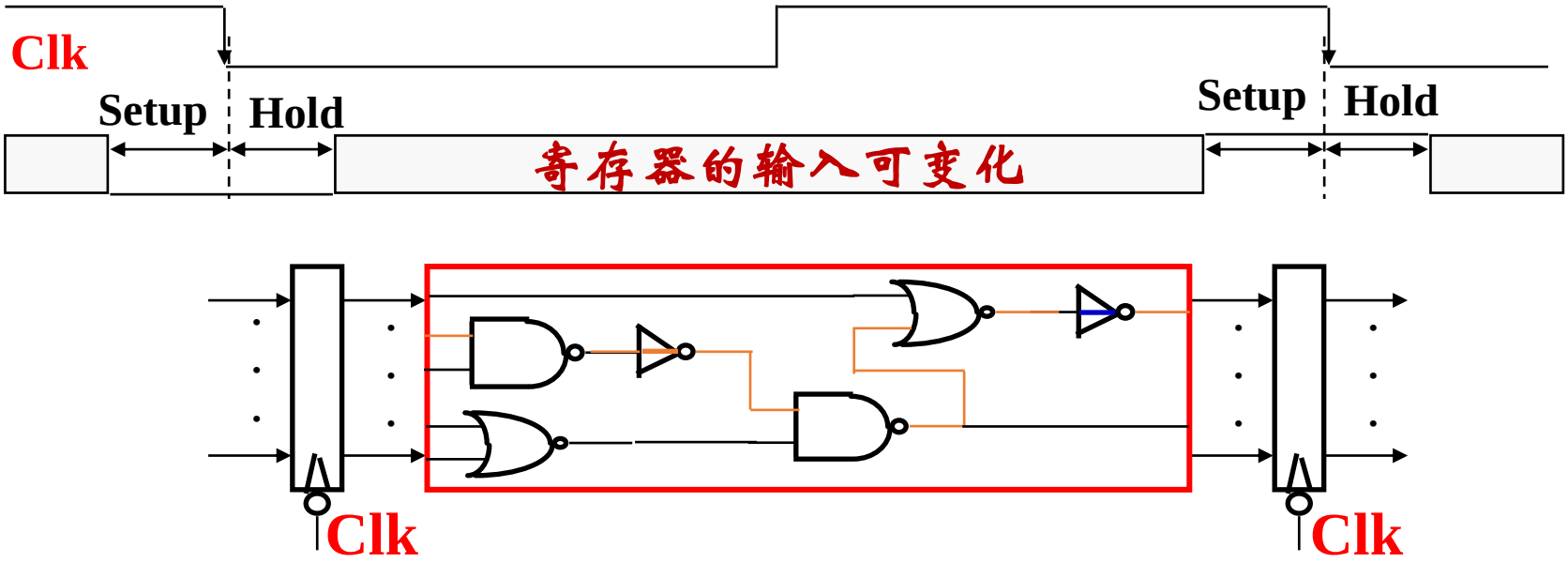
- 同步系统 (Synchronous system)
  - 用专门时序信号定时操作
  - 时序信号规定何时发何操作
- 时序信号
  - 用于同步控制的定时信号



- **指令周期**
  - **取出并执行一条指令的总时间**
  - **不同指令的指令周期会不同**

数据通路与时序控制

现代计算机的时钟周期

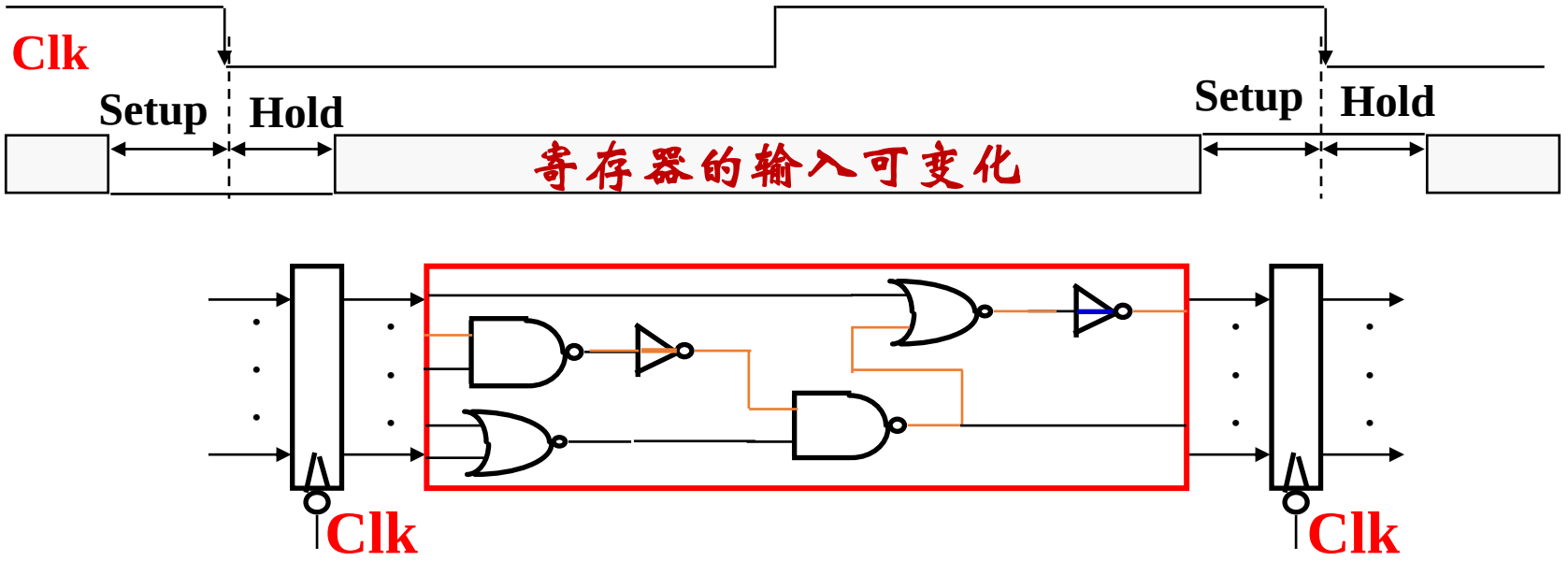


数据通路由“…+状态元件+操作元件(组合电路)+状态元件+…”组成

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

数据通路与时序控制

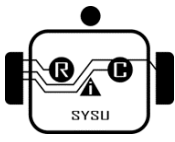
现代计算机的时钟周期



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

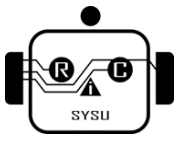
组成指令功能的四个基本操作

- 每条指令的功能可能由四个基本操作实现
  - (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ALU
  - (4) 进行某种算术或逻辑运算，将结果送入某个寄存器
- 操作功能可形式化描述
  - 寄存器传输语言RTL (Register Transfer Language)描述
  - 本章所用的RTL规定：
    - 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



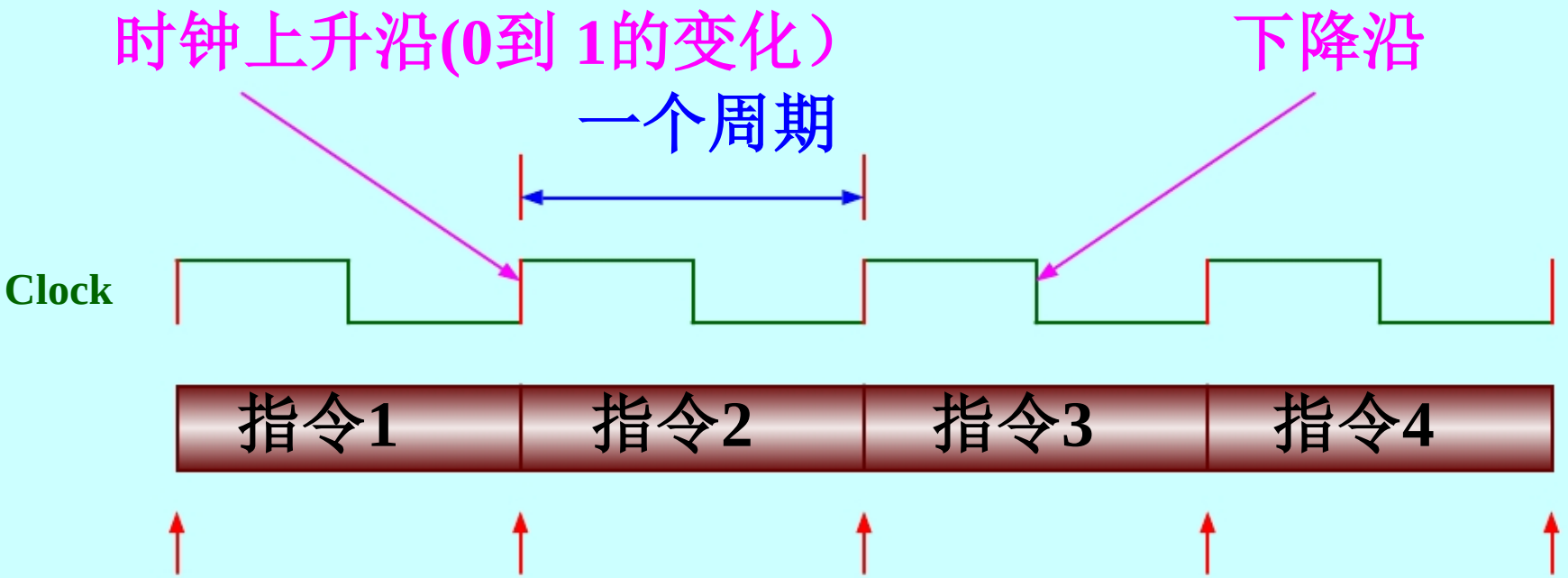
主要内容

- 处理器概述
- 逻辑设计规则
  - 数据通路的功能及实现
    - 操作元件(组合逻辑)
    - 状态 / 存储元件(时序逻辑)
  - 数据通路的定时
- 单周期处理器
  - 数据通路
  - 控制器



单周期CPU

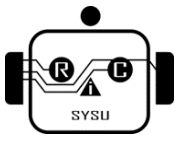
单周期(Single-Cycle) CPU执行一条指令用一个时钟周期
(最简单的执行方式)



在时钟上升沿保存指令的结果和下一条指令的地址

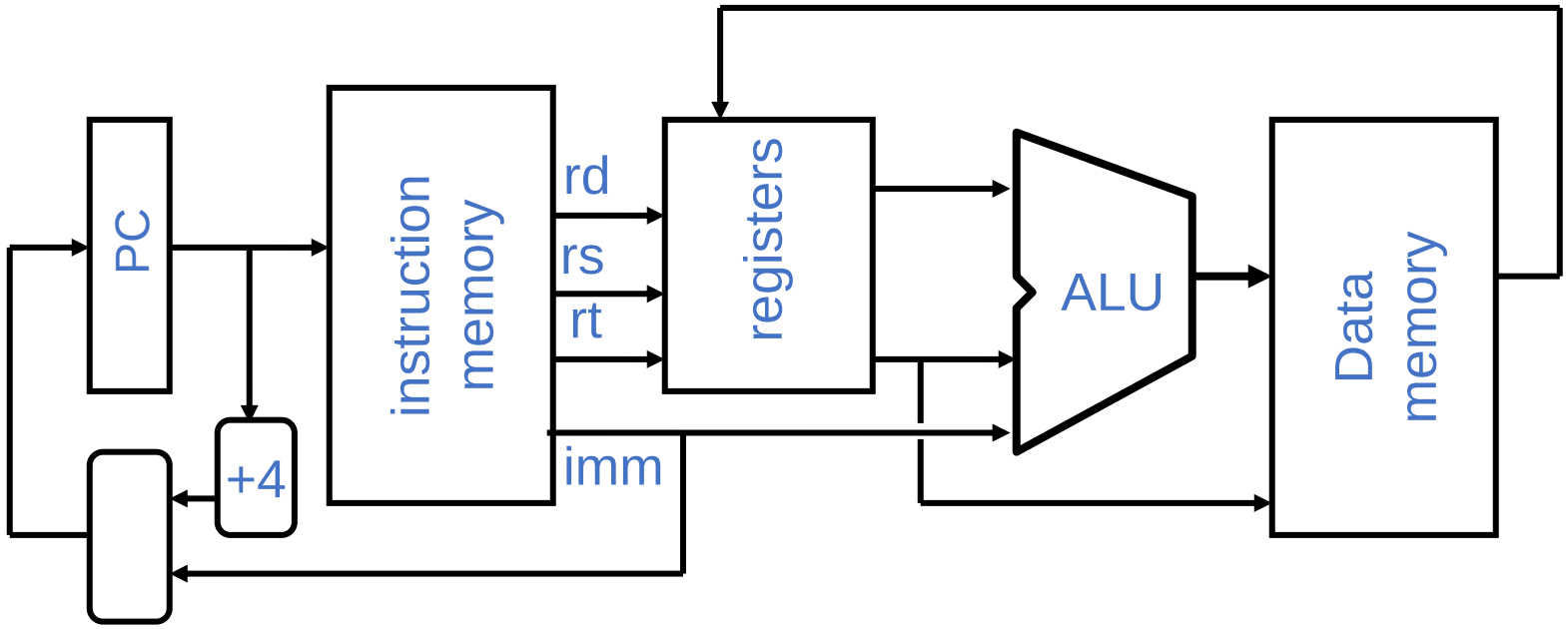
单周期数据通路设计

- 问题：将 “执行整个指令”的块做为一个整体
  - 太大(该块要执行从取指令开始的所有操作)
  - 效率不高
- 解决方案：将 “执行整个指令” 的操作分解为多个阶段（stage)，然后将所有阶段连接在一起产生整个datapath
  - 每一阶段更小，从而更容易设计
  - 方便优化其中一个阶段，而不必涉及其他阶段



单周期数据通路设计

Generic Steps of Datapath



1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Reg. Write

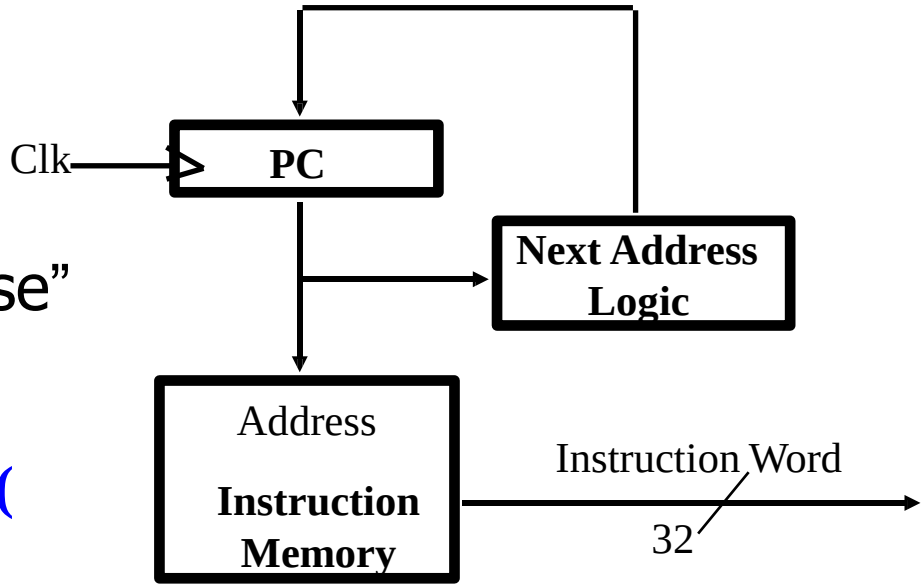
Stages of the Datapath (1/5)

取指令单元IF

- 所有指令都公用的单元
  - **Fetch the Instruction: mem[PC]**
  - 更新**PC:**
    - 顺序指令时:

      PC <= PC + 4
    - 分支跳转时:

      PC <= “something else”

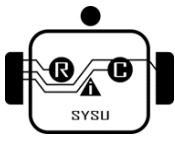


顺序：先取指令，再改变PC的值(具体实现可以并行)

决不能先改变PC的值，再取指令

Stages of the Datapath (2/5)

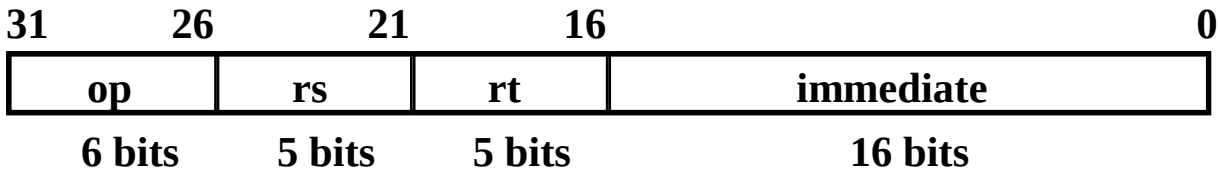
- Stage 2: **指令译码Instruction Decode**
  - 在取到指令后，下一步从各域(fields)中得到数据(对必要的指令数据进行*解码*)
  - 首先，读出Opcode，以决定指令类型及字段长度
  - 接下来，从相关部分读出数据
    - for add, read two registers
    - for addi, read one register
    - for jal, no reads necessary



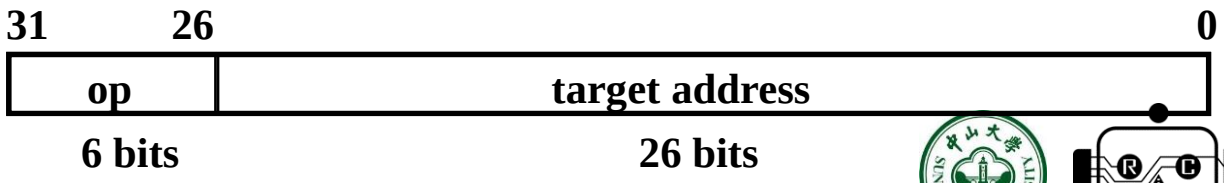
加减指令(R-type类型)

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

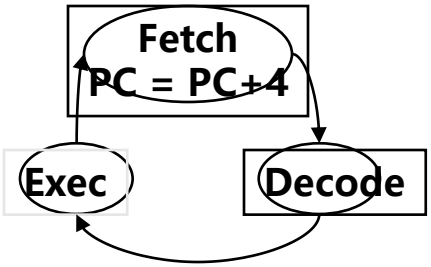


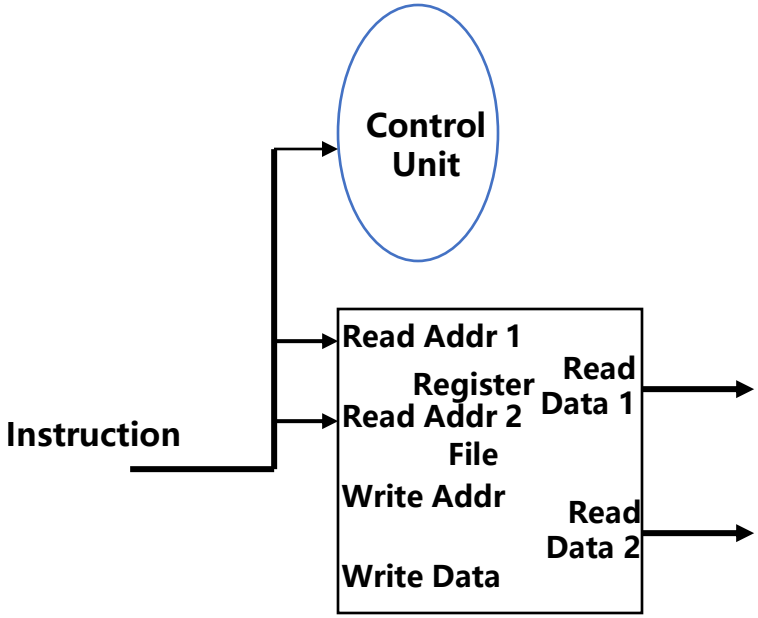
1. 考虑add和sub指令(R-Type指令的代表)



Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit





and

reading two values from the Register File

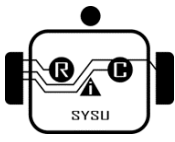
- **Register File addresses are contained in the instruction**

Stages of the Datapath (3/5)

- Stage 3: ALU (Arithmetic-Logic Unit)
  - 大多数指令的实际工作在此部完成：算术指令（+, -, *, /），shifting, logic (&, |), comparisons (slt)
  - what about loads and stores?
    - lw $t0, 40($t1)
    - 要访问的内存地址 = $t1的值 + 40
    - so we do this addition in this stage

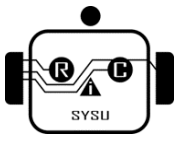
Stages of the Datapath (4/5)

- Stage 4: 内存访问Memory Access
  - 事实上只有load和store指令在此stage会做事；其它指令在此阶段空闲idle或者直接跳过本阶段
  - 由于load和store需要此步，因此需要一个专门的阶段 stage来处理他们
  - 由于cache系统的作用，该阶段有望加速
  - 如果没有caches，本阶段stage会很慢



Stages of the Datapath (5/5)

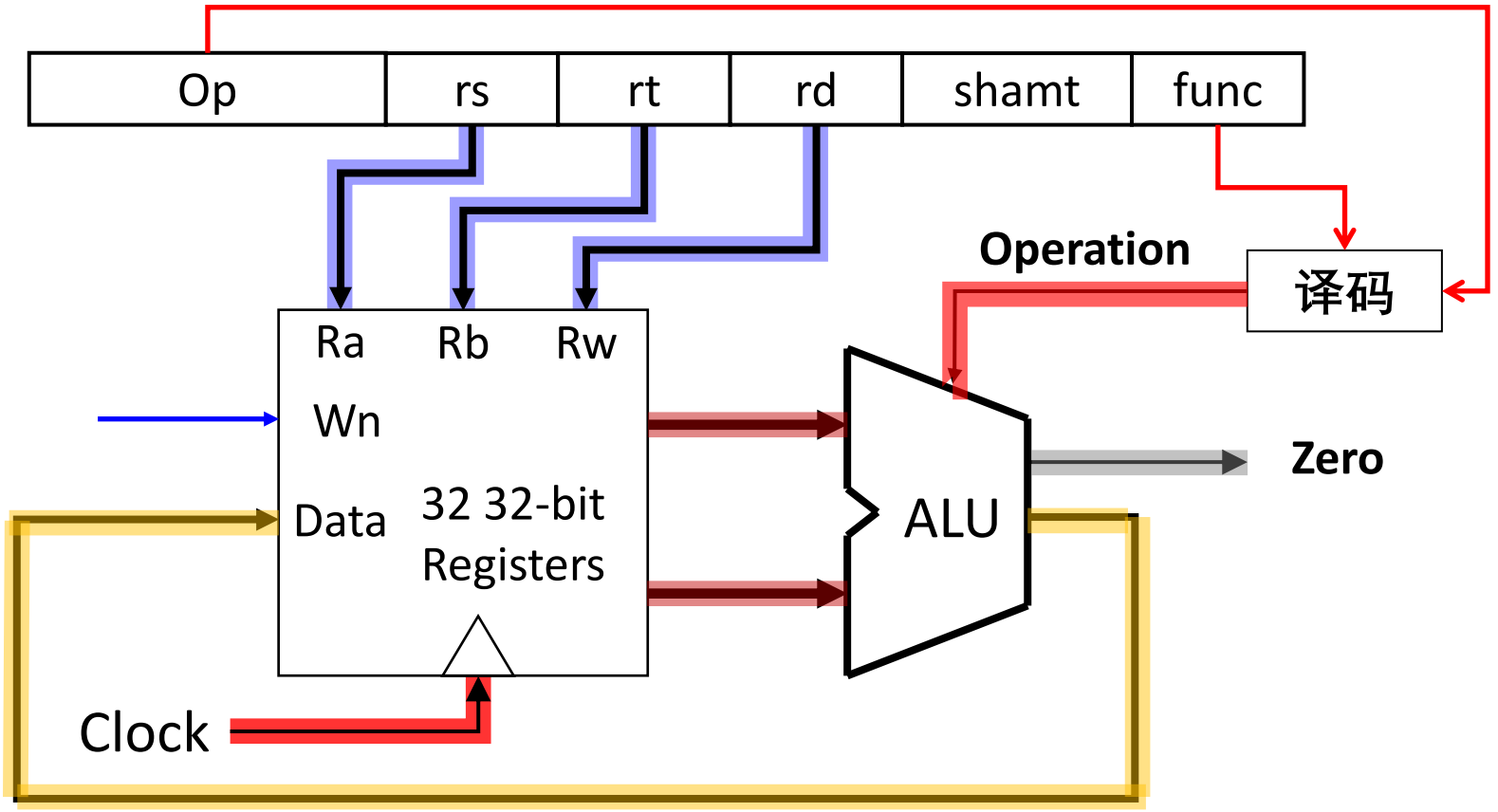
- Stage 5: 写寄存器Register Write
  - 大多数指令会将计算结果写到寄存器
  - 例如: arithmetic, logical, shifts, loads, slt
  - what about stores, branches, jumps?
    - don’t write anything into a register at the end
    - these remain idle during this fifth stage or skip it all together



数据路径—R型指令

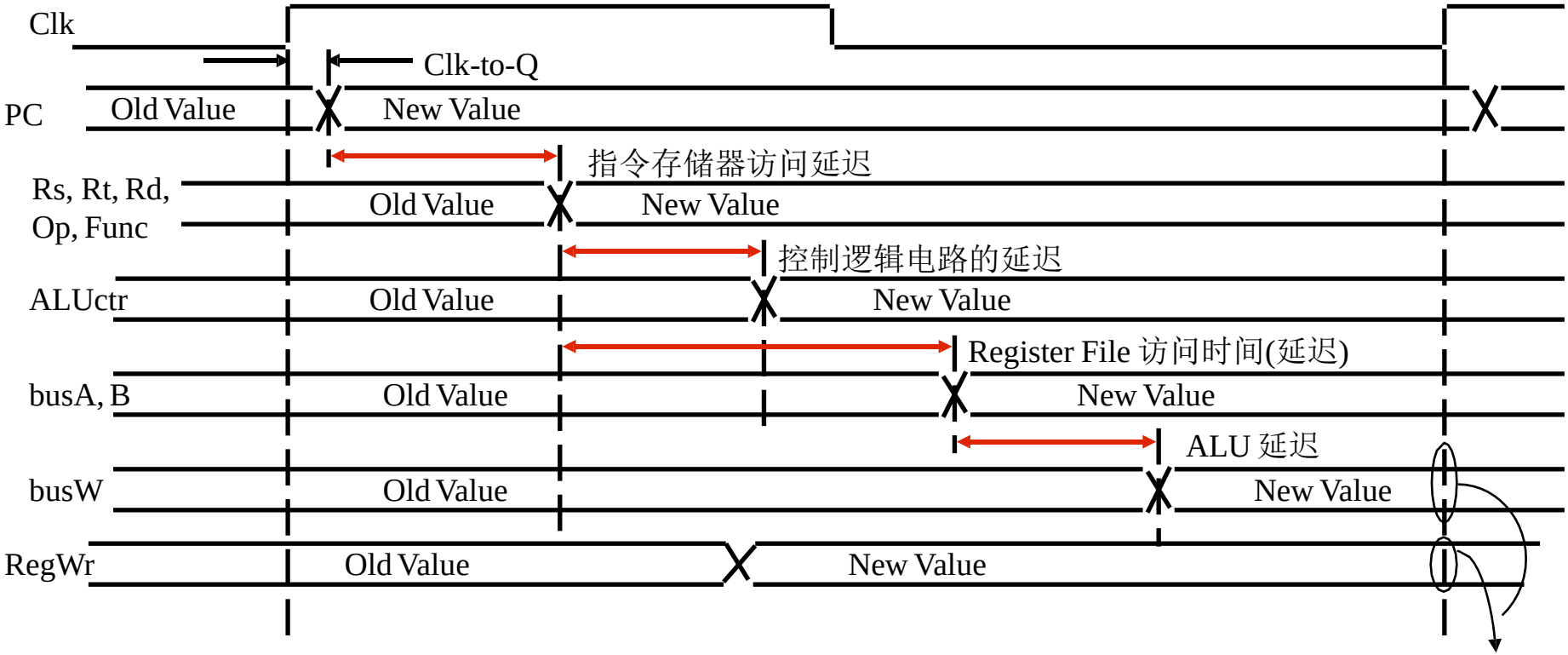
指令的意义:寄存器rs中的数据和寄存器rt中的数据相加，结果存放在寄存器rd中。把PC + 4写入PC

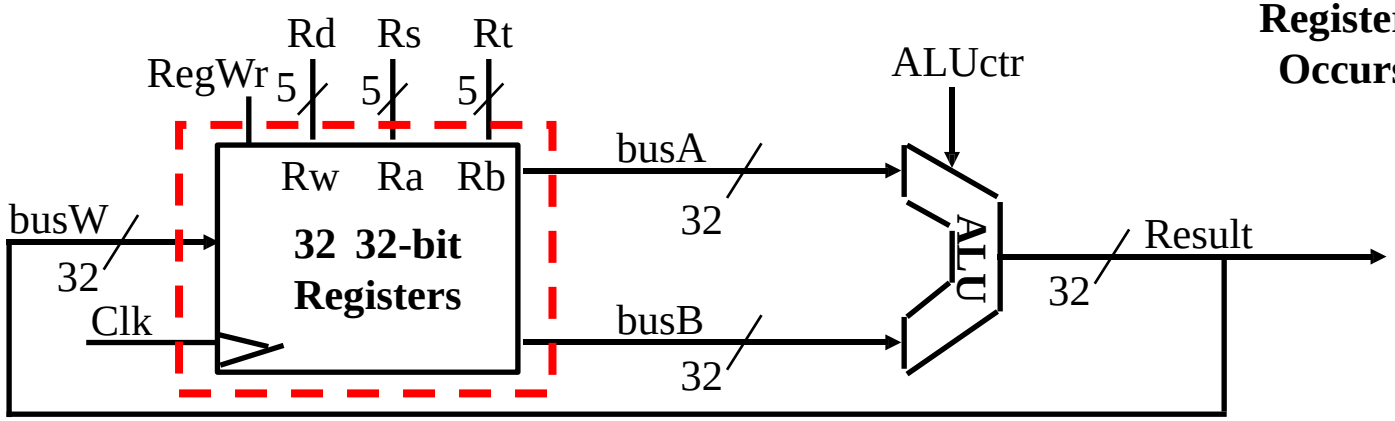
$$add\ rd\ ,\ rs\ ,\ rt \qquad R[rd] \leftarrow R[rs] + R[rt]$$



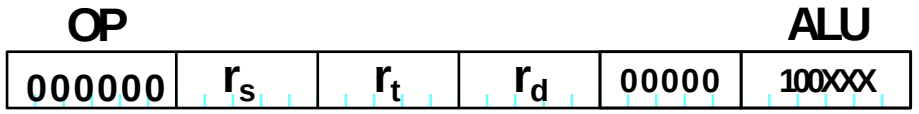
R format operations (`add, sub, slt, and, or`)

定时:完整周期


Clk
Clk-to-Q
PC
Old Value
New Value
指令存储器访问延迟
Rs, Rt, Rd,
Op, Func
Old Value
New Value
控制逻辑电路的延迟
ALUctr
Old Value
New Value
Register File 访问时间(延迟)
busA, B
Old Value
New Value
ALU 延迟
busW
Old Value
New Value
RegWr
Old Value
New Value
Register Write
Occurs Here

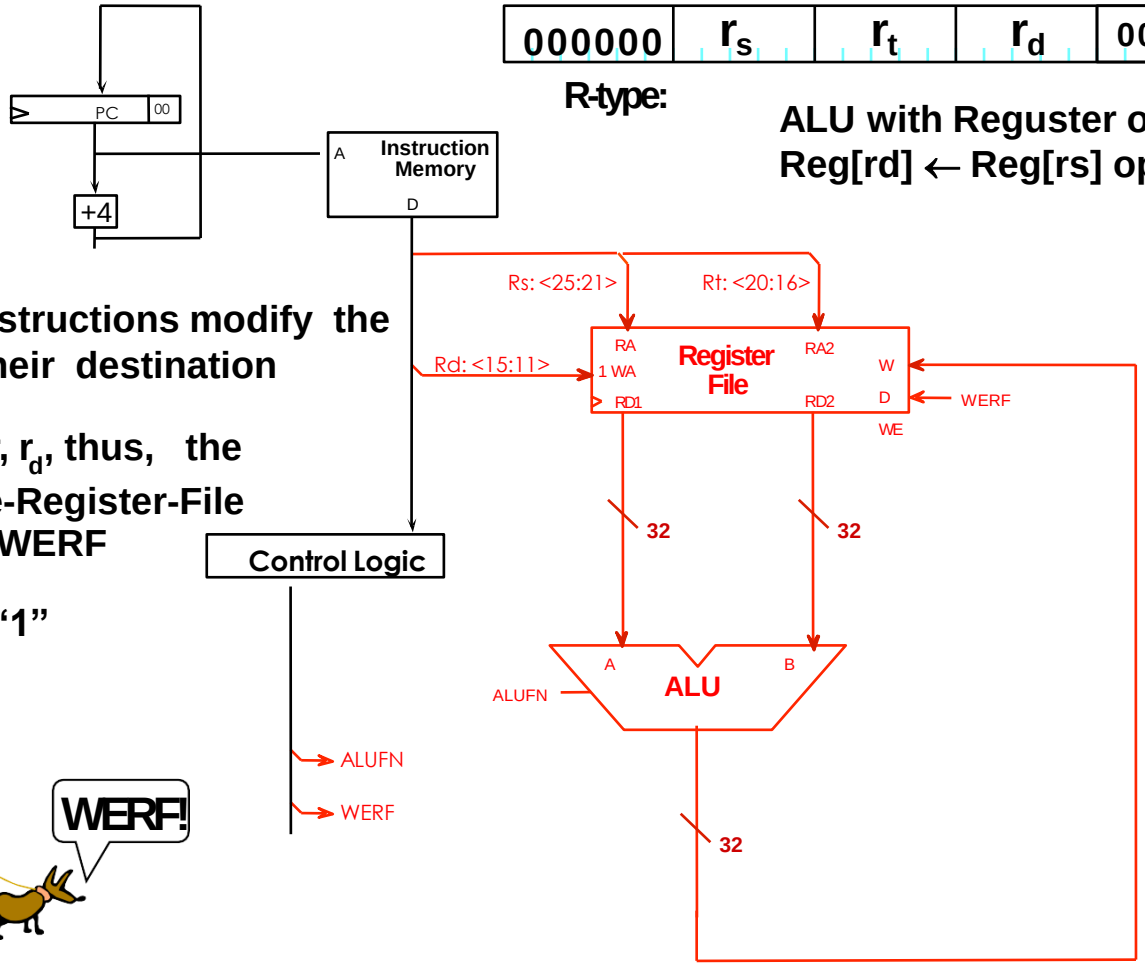

Rd
Rs
Rt
RegWr
5
5
5
ALUctr
Rw
Ra
Rb
busA
32
32-bit
Registers
busW
32
Clk
busB
32
ALU
Result
32

3-Operand ALU Data Path
需要的控制信号



R-type:

ALU with Reguster operands
Reg[rd] ← Reg[rs] op Reg[rt]

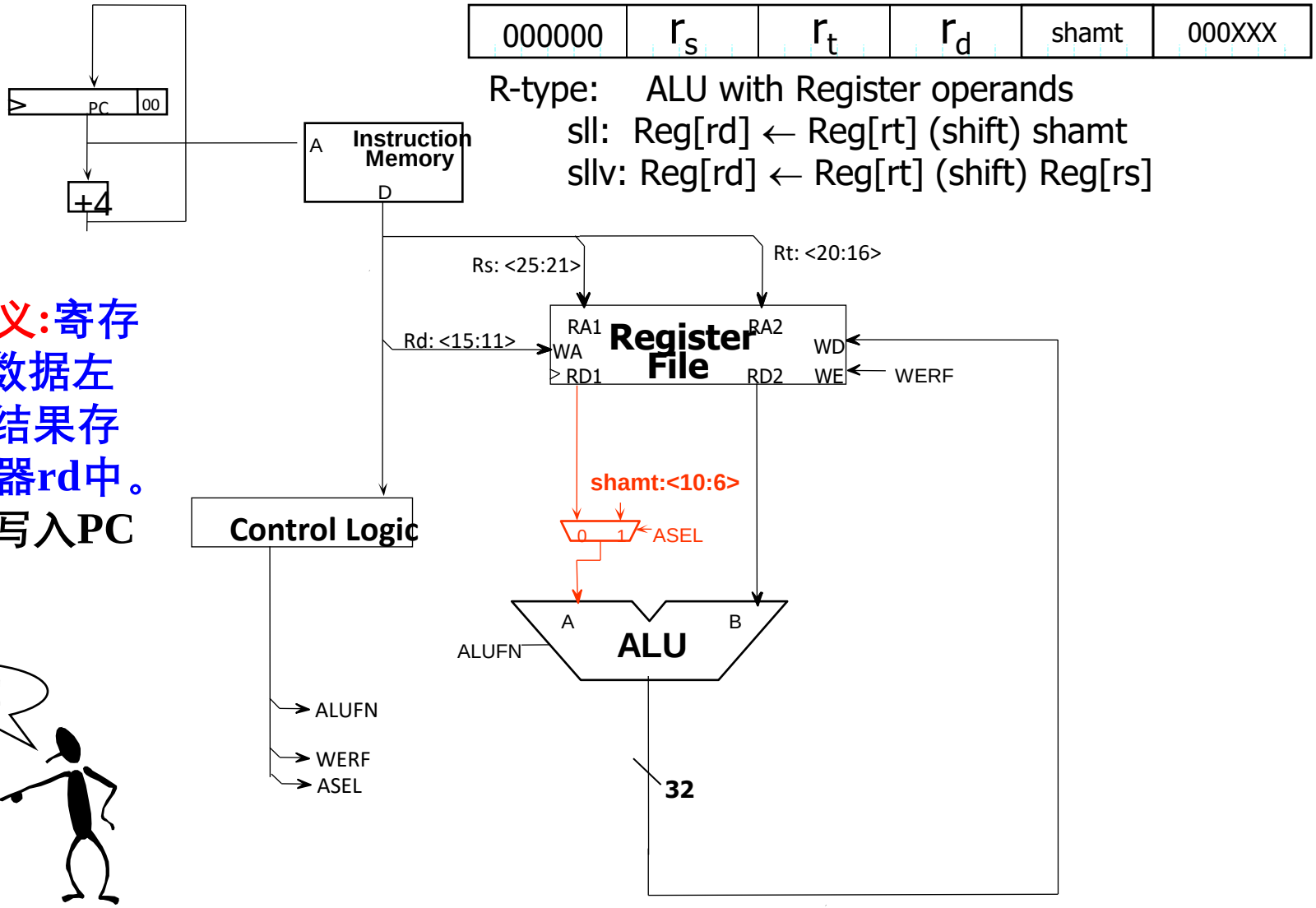


All of these instructions modify the contents of their destination

register, r_d, thus, the Write-Enable-Register-File control line, WERF

is always a "1"

Shift Instructions



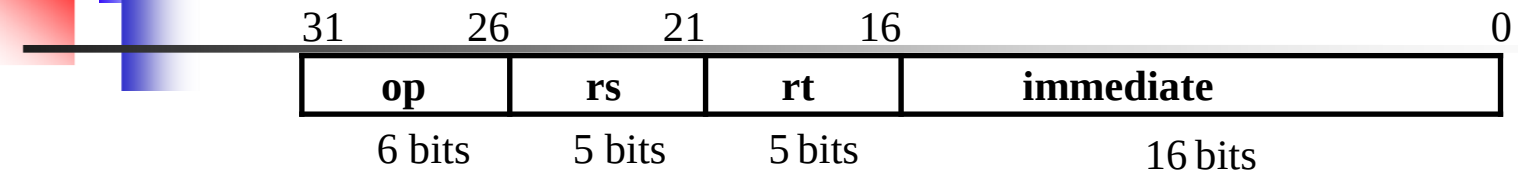
指令的意义:寄存器rt中的数据左移sa位，结果存放在寄存器rd中。

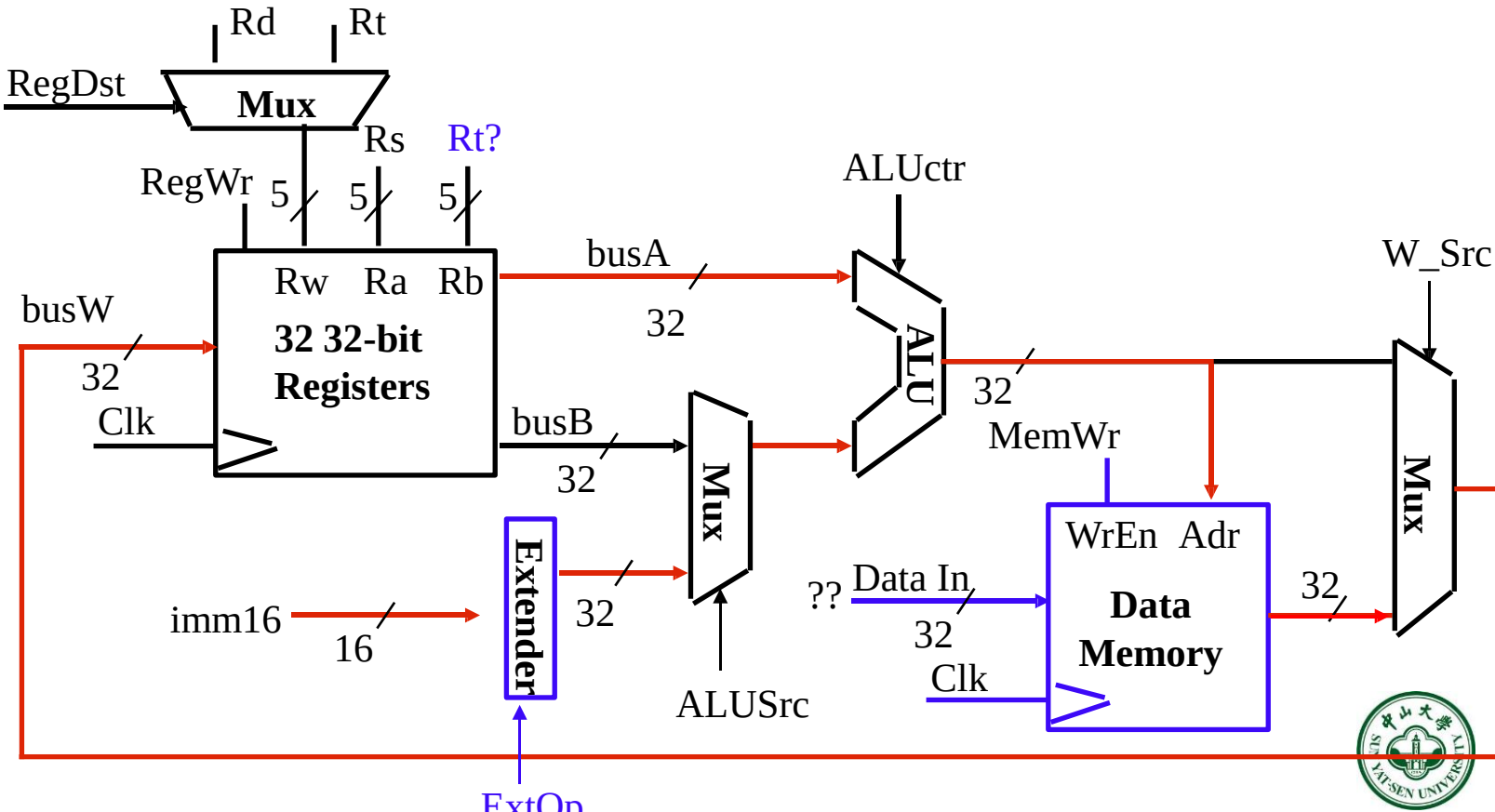
把PC + 4写入PC

与sll类似的指令：srl, sra

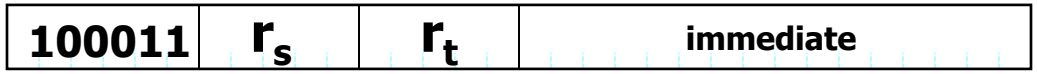
LW操作

- **R[rt] <= Mem[R[rs] + SignExt[imm16]]**
  - Example: lw rt, imm16(rs)





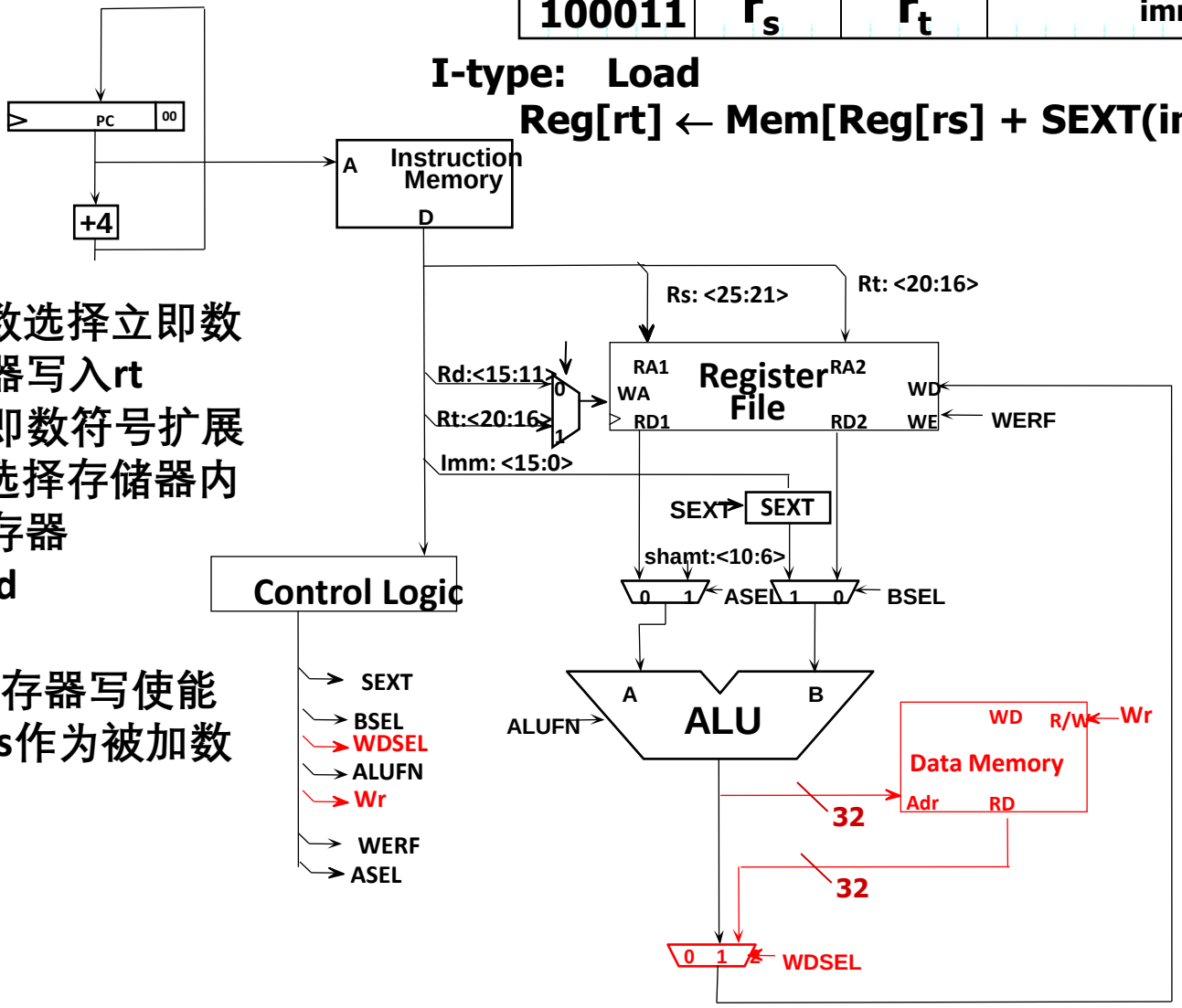
Load Instruction
需要的控制信号



I-type: Load

Reg[rt] ← Mem[Reg[rs] + SEXT(immediate)]

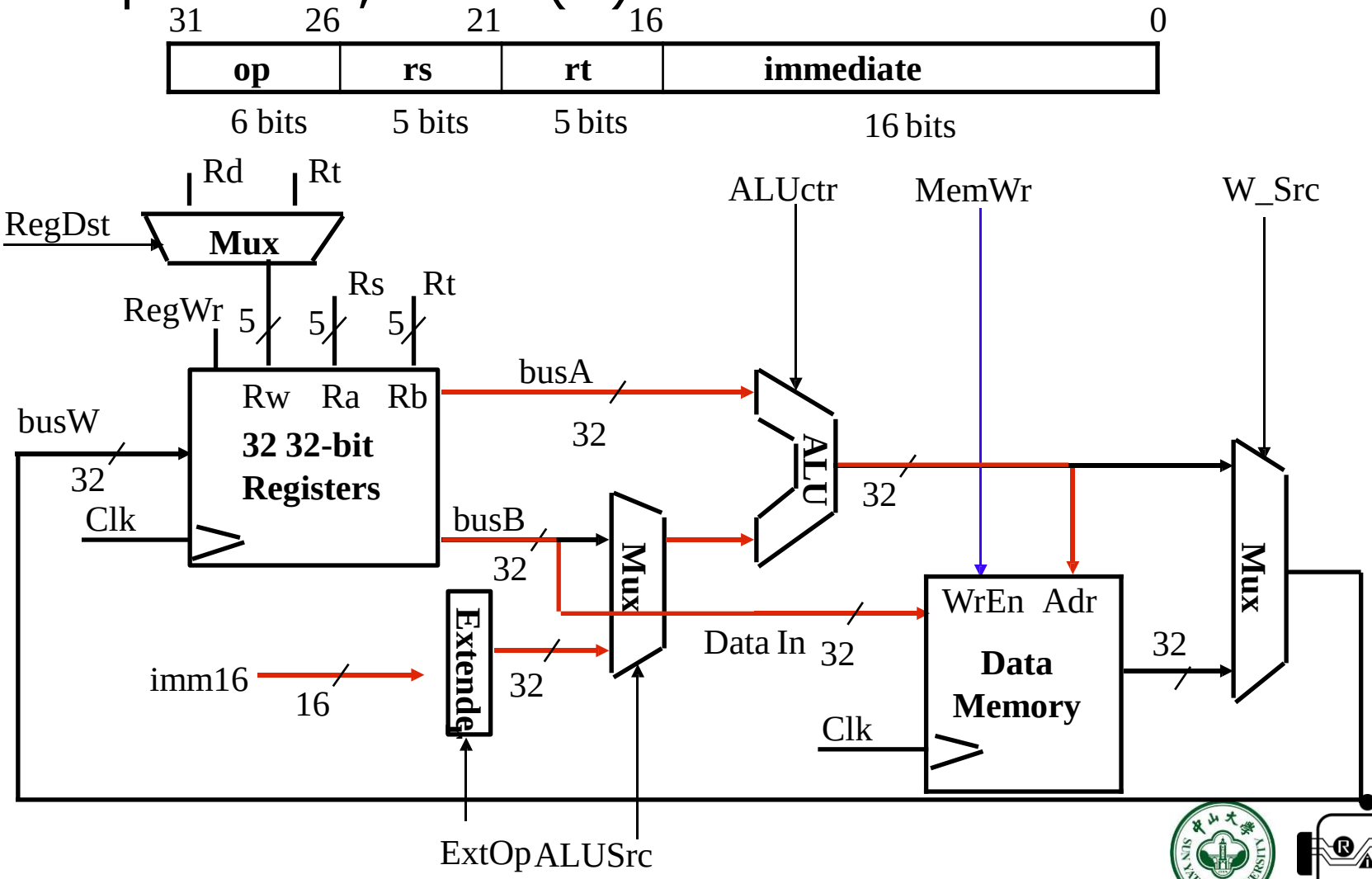
BSEL=1加数选择立即数和选寄存器写入rt
SEXT=1立即数符号扩展
WDSEL=2选择存储器内容写入寄存器
ALUFN=add
Wr=0
WERF=1寄存器写使能
ASEL=0选rs作为被加数



SW操作

- **Mem[R[rs] + SignExt[imm16]] <= R[rt]**

Example: sw rt, imm16(rs)



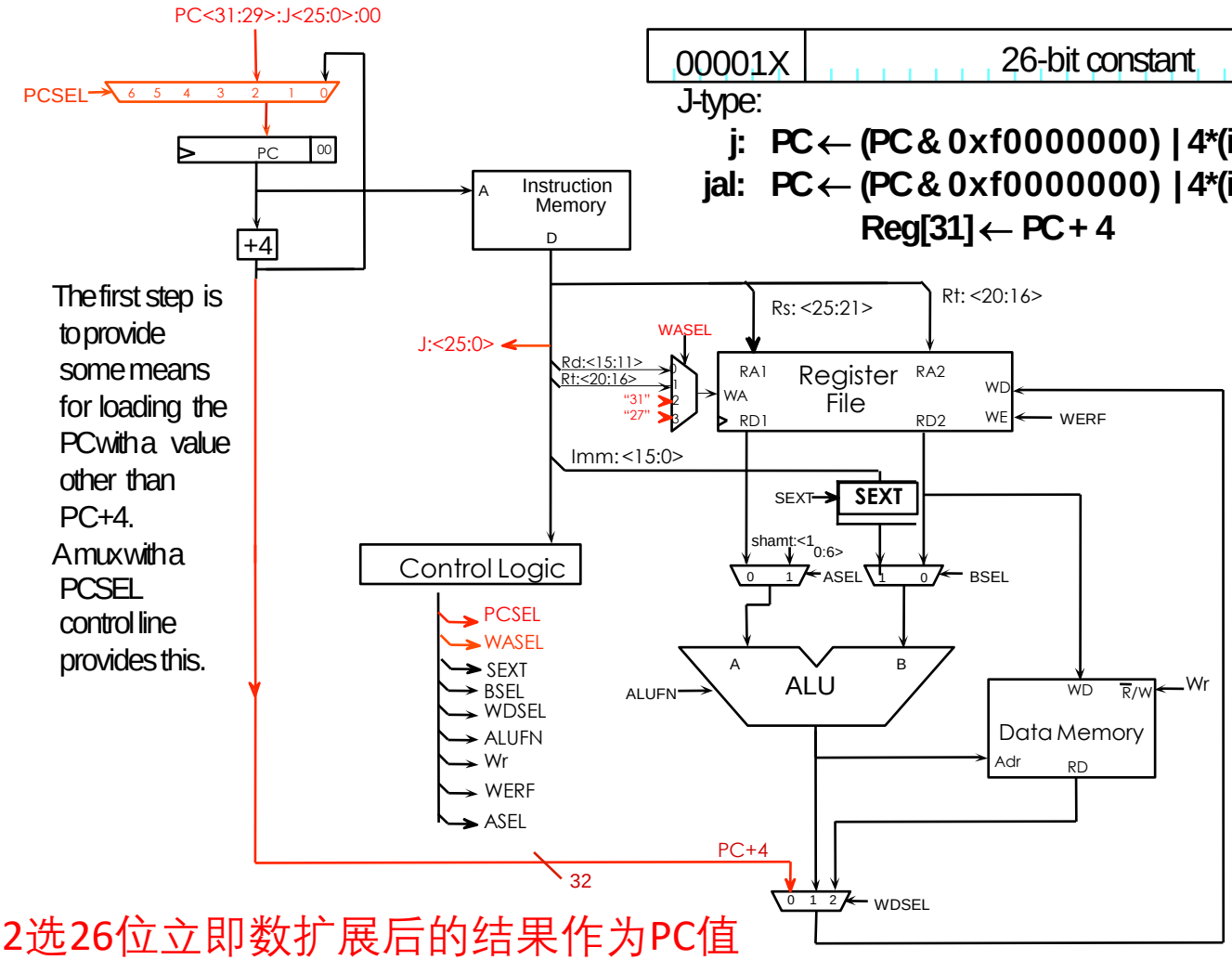
Store Instruction
需要的控制信号

I-type: Store

Mem[Reg[rs] + SEXT(immediate)] ← Reg[rt]

BSEL=1加数选择立即数
和选寄存器写入rt
SEXT=1立即数符号扩展
WDSEL=1选择存储器内
容写入寄存器
ALUFN=add
Wr=1
WERF=0寄存器写使能
ASEL=0选rs作为被加数

JMP Instructions



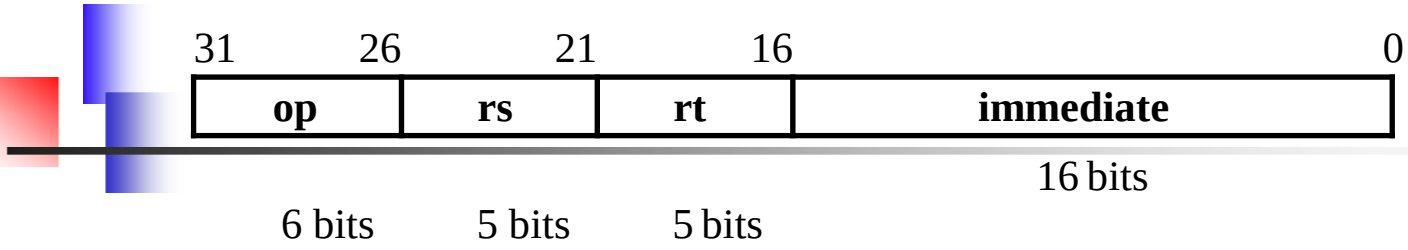
PCSEL=2选26位立即数扩展后的结果作为PC值

WDSEL=0选择PC+4写入$31

Jal指令，WDSEL=0选择PC+4写入$31

寄存器写入输入选择WASEL选$31

The Branch Instruction

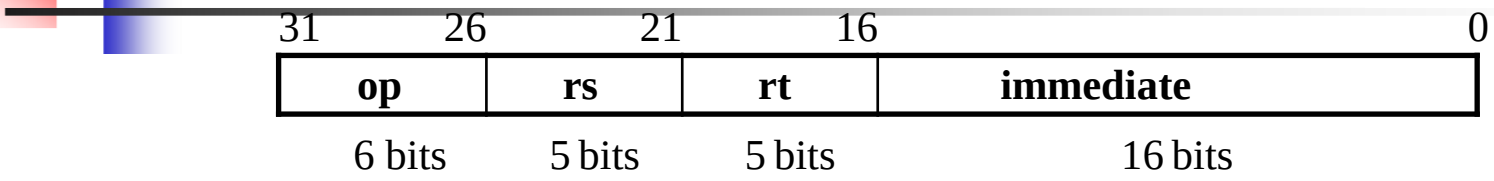


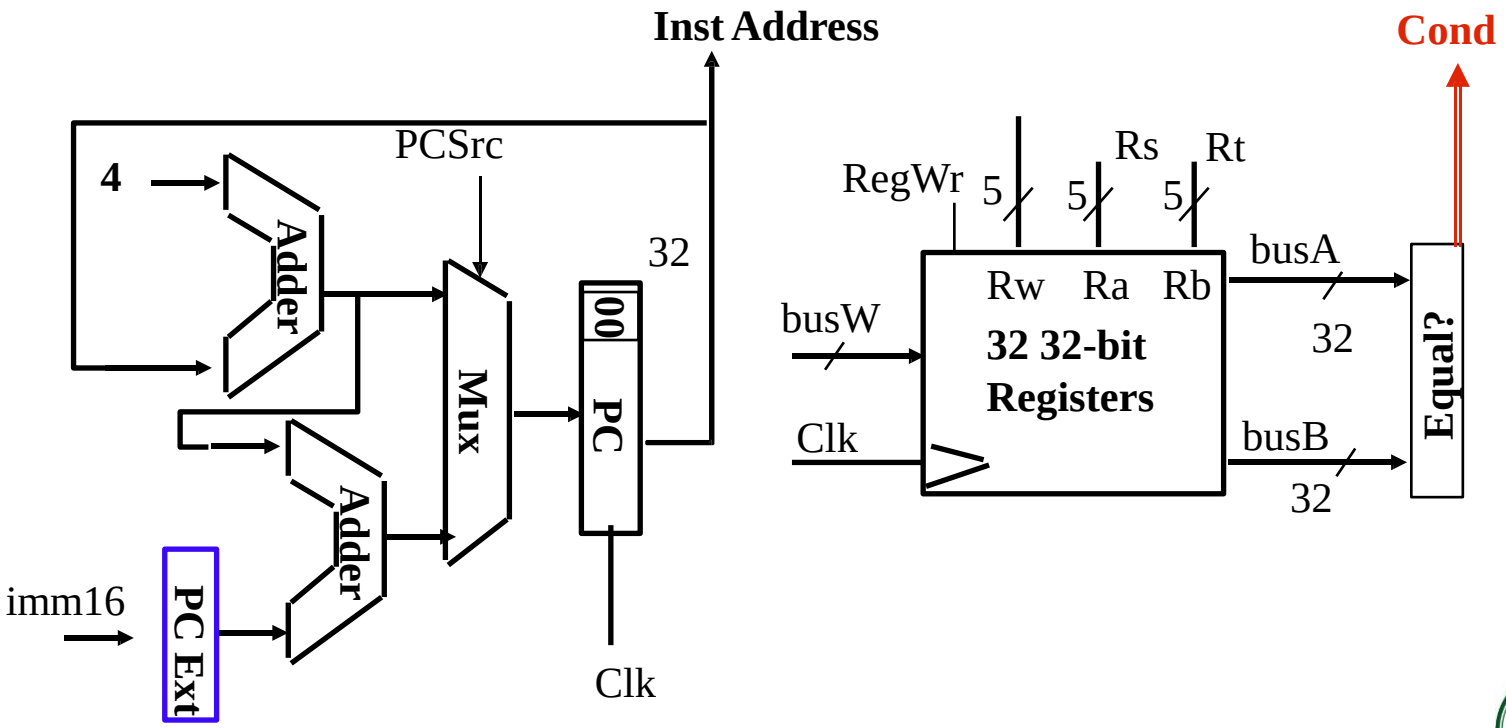
- **Beq rs, rt, imm16**
  - mem[PC] 从存储器取指令
  - Equal <= (R[rs] == R[rt]) 计算分支条件
  - if (Equal) 计算下一个指令的地址
    - PC <= PC + 4 + { SignExt(imm16) , 2b00 }
  - else
    - PC <= PC + 4

分支指令的datapath

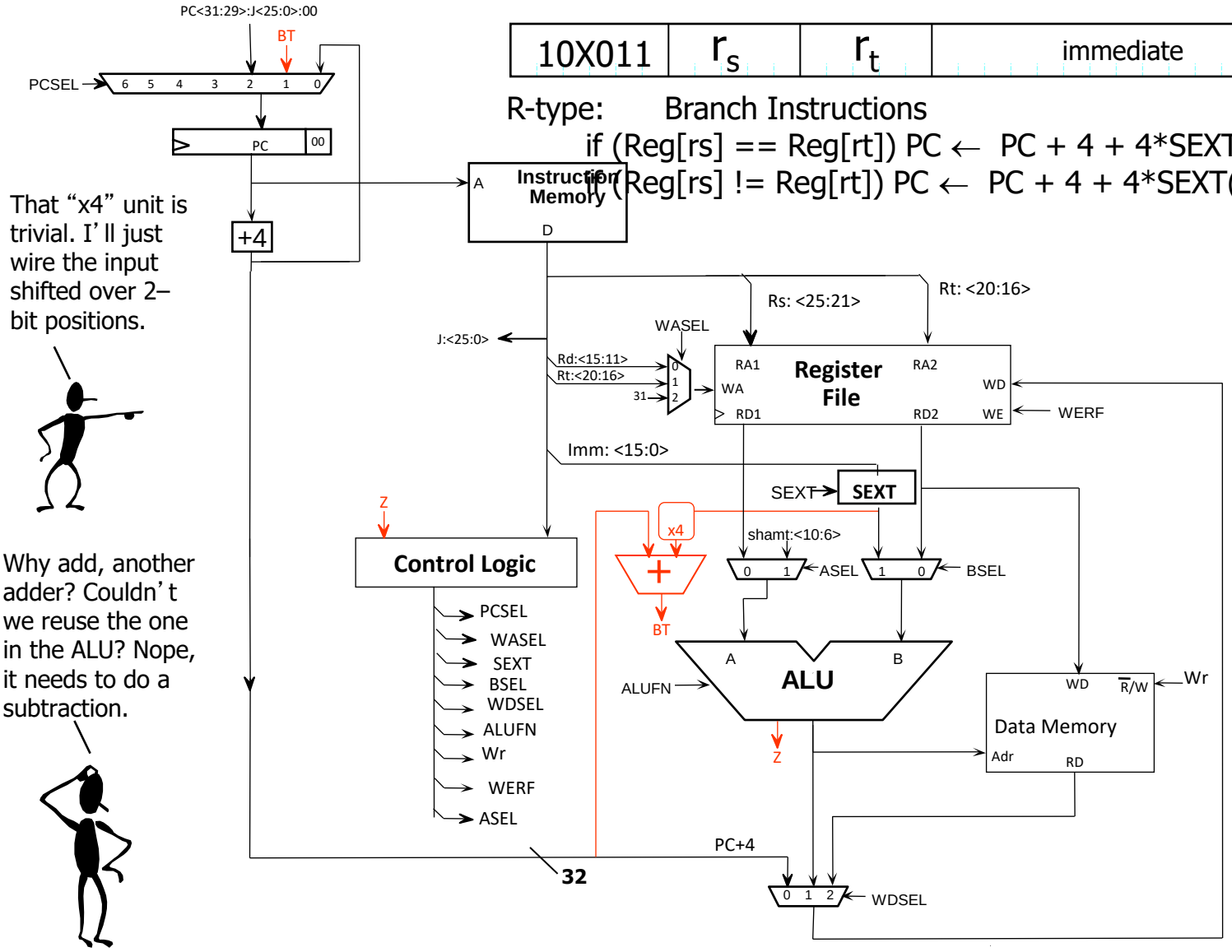
- beq rs, rt, imm16

  Datapath generates condition (equal)

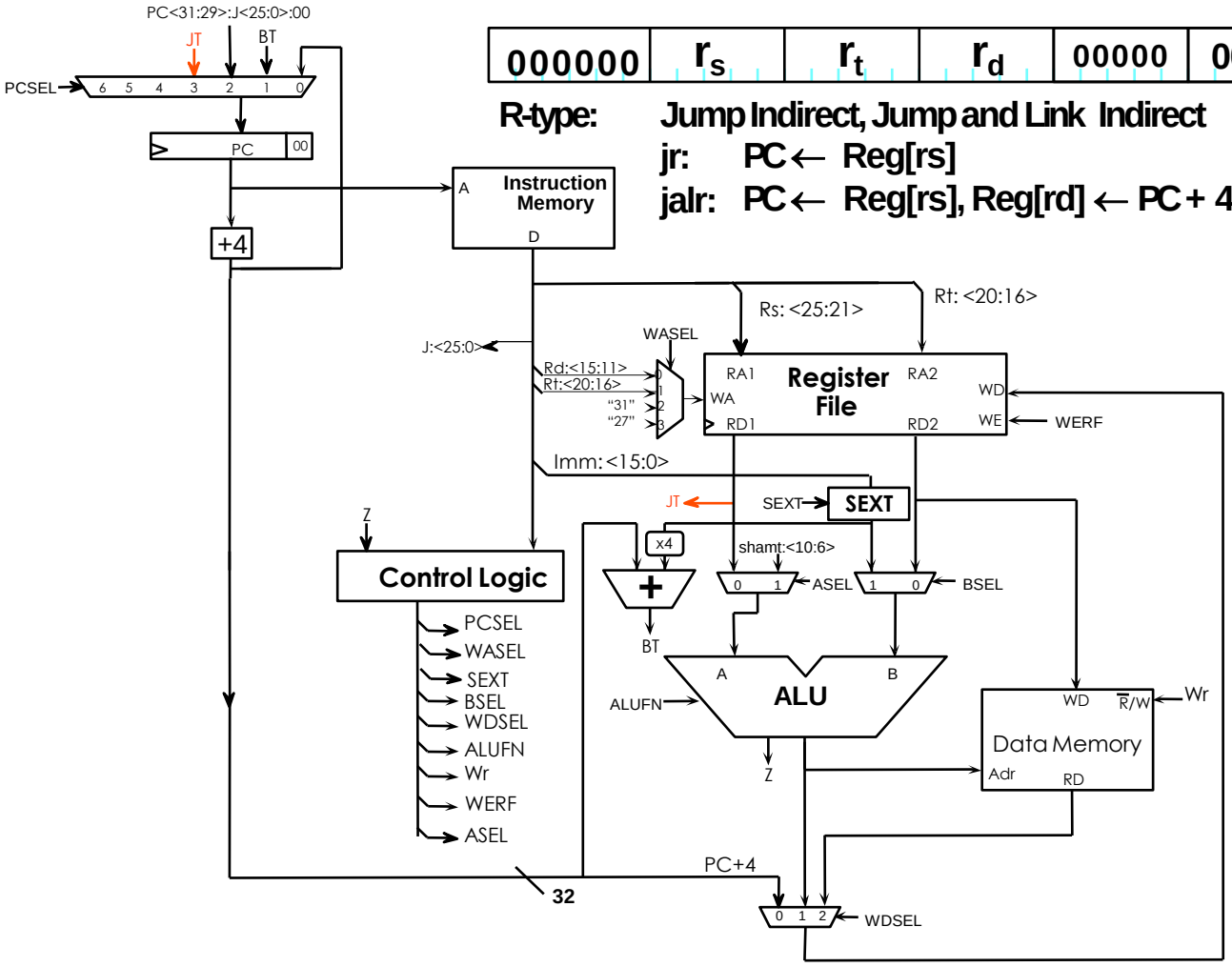




BEQ/BNE Instructions
需要的控制信号



Jump Indirect Instructions



000000	r_s	r_t	r_d	00000	00100X

R-type: Jump Indirect, Jump and Link Indirect

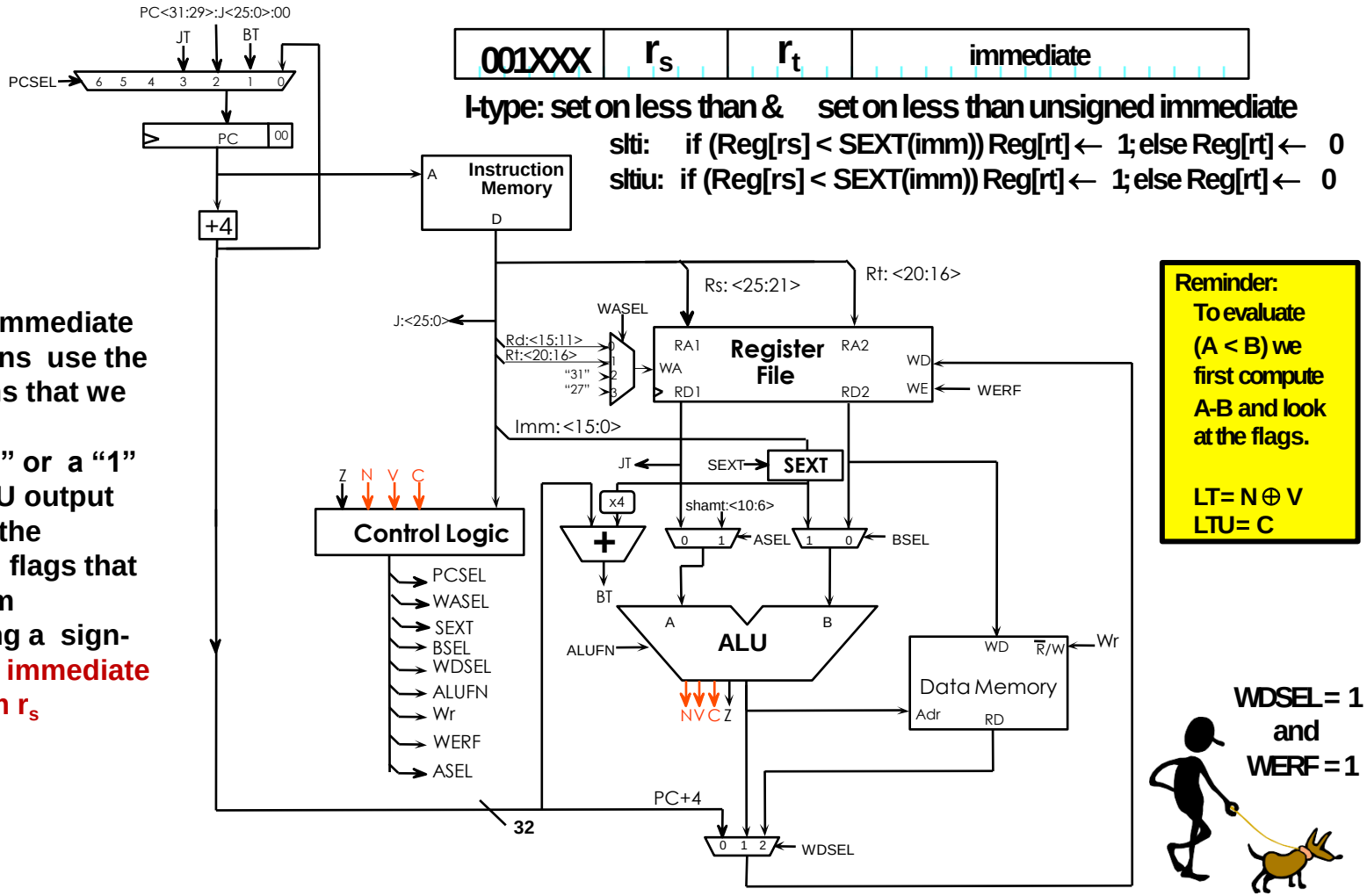
jr: PC ← Reg[rs]

jalr: PC ← Reg[rs], Reg[rd] ← PC + 4

The instructions, JR and JALR, set the next PC using the contents of the register specified by r_s. This requires the the output of the first register port be routed to the mux used to load the next PC.

The JALR instruction reuses the connections from JAL that allow PC+4 to be stored in "31".

Loose Ends 有立即数的操作



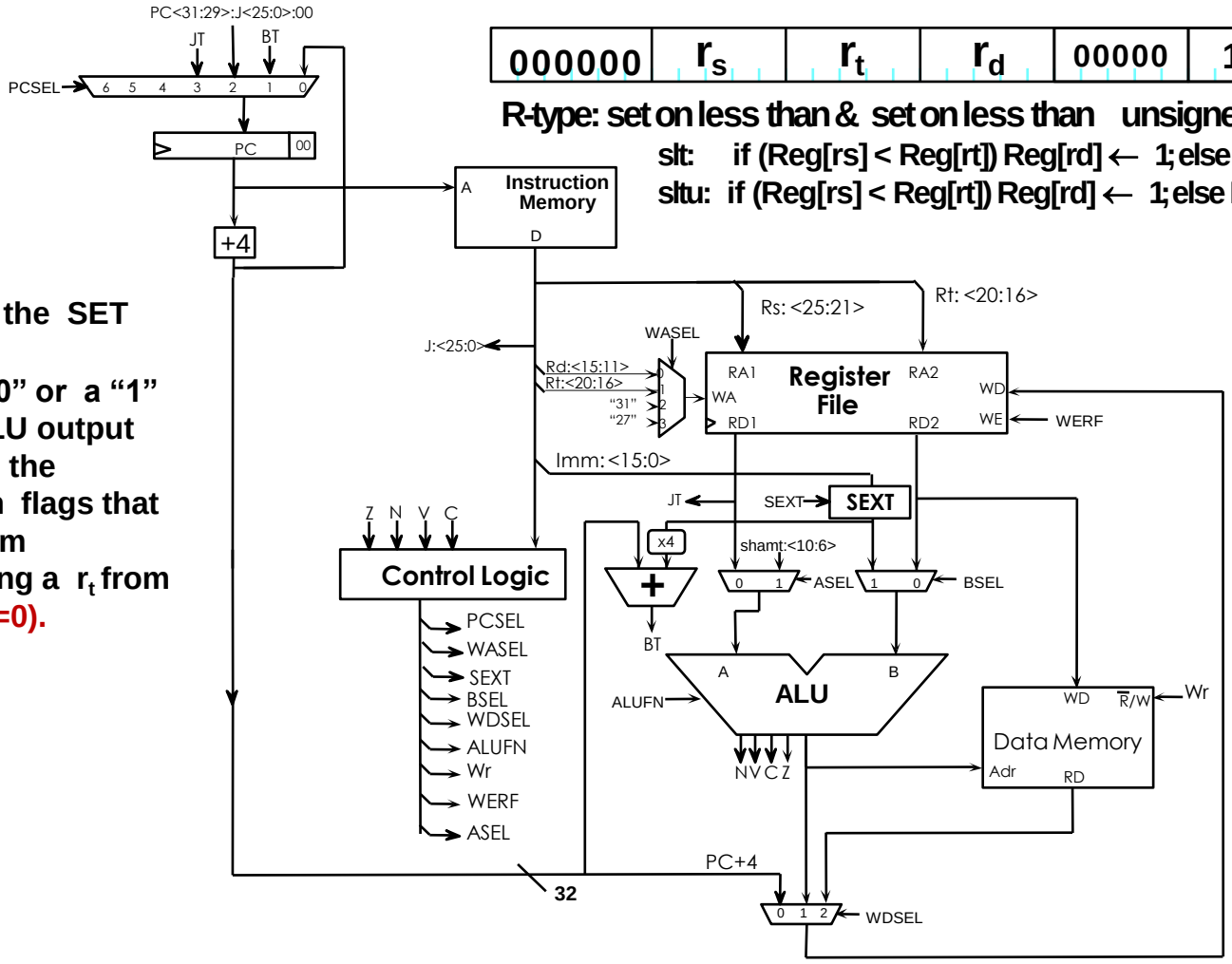
The SET immediate instructions use the ALU paths that we added to route a "0" or a "1" to the ALU output based on the condition flags that result from subtracting a sign-extended immediate value from r_s (BSEL=1).

rd?

- R[rt] <= R[rs] op ZeroExt[imm16]

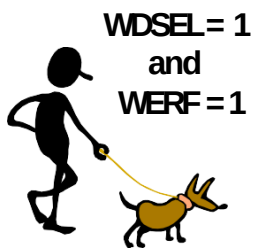
More Loose Ends

Similarly, the SET ALU route a "0" or a "1" to the ALU output based on the condition flags that result from subtracting a r_t from r_s (BSEL=0).

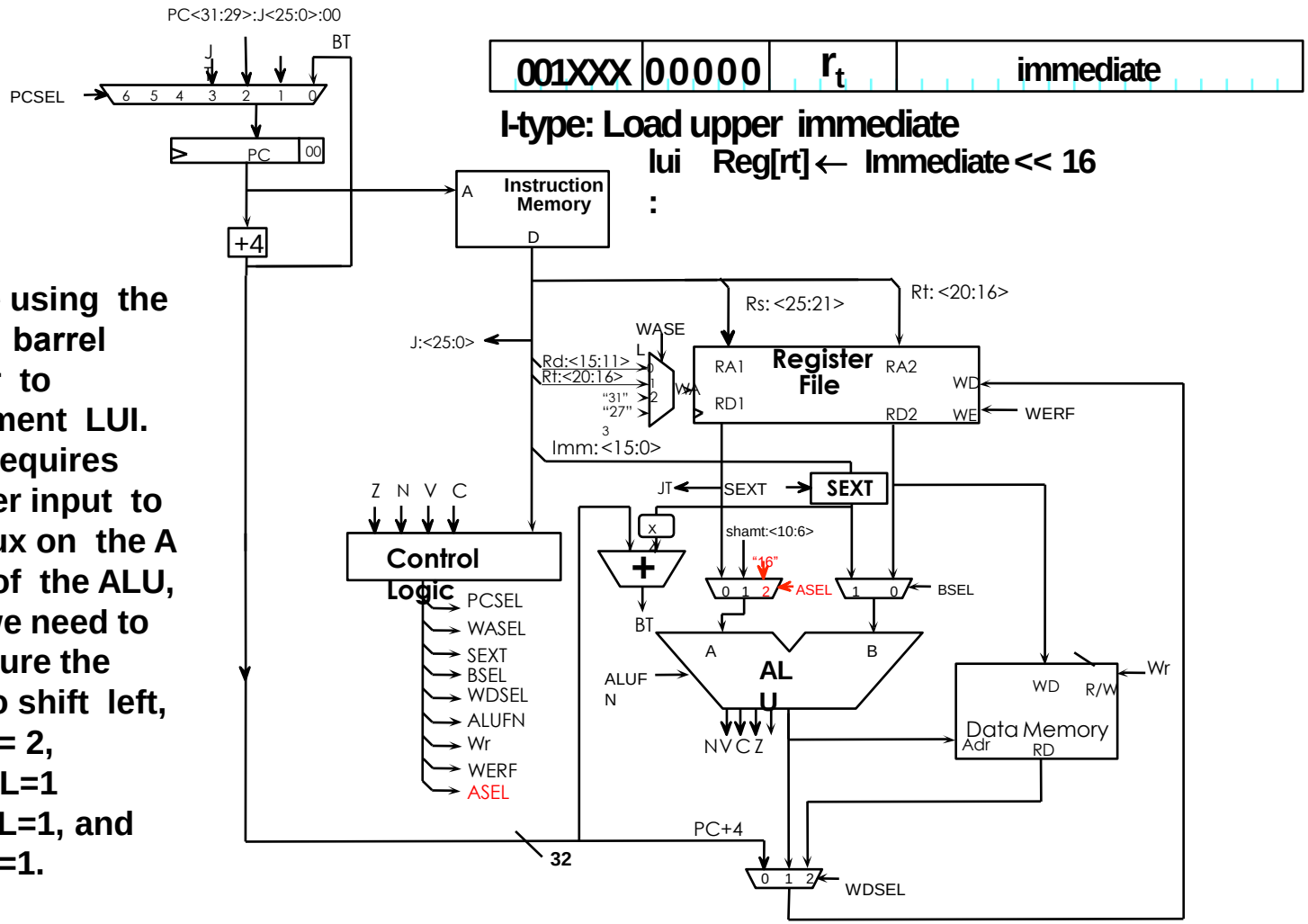


Reminder:
To evaluate (A < B) we first compute A-B and look at the flags.

LT= N ⊕ V
LTU= C



LUI Ends



We are using the ALU's barrel shifter to implement LUI. This requires another input to the mux on the A input of the ALU, and we need to configure the ALU to shift left, ASEL = 2, WDSEL=1 WASEL=1, and WERF=1.

Reset, Interrupts, and Exceptions

- FIRST, we need some way to get our machine into a known initial state. This doesn't mean that all registers will be initialized, just that we'll know where to fetch the first instruction. We'll call this control input, RESET
- We'd also like RECOVERABLE INTERRUPTS for
  - FAULTS (eg, Illegal or unimplemented Instruction)

- CPU or SYSTEM generated [*synchronous*]

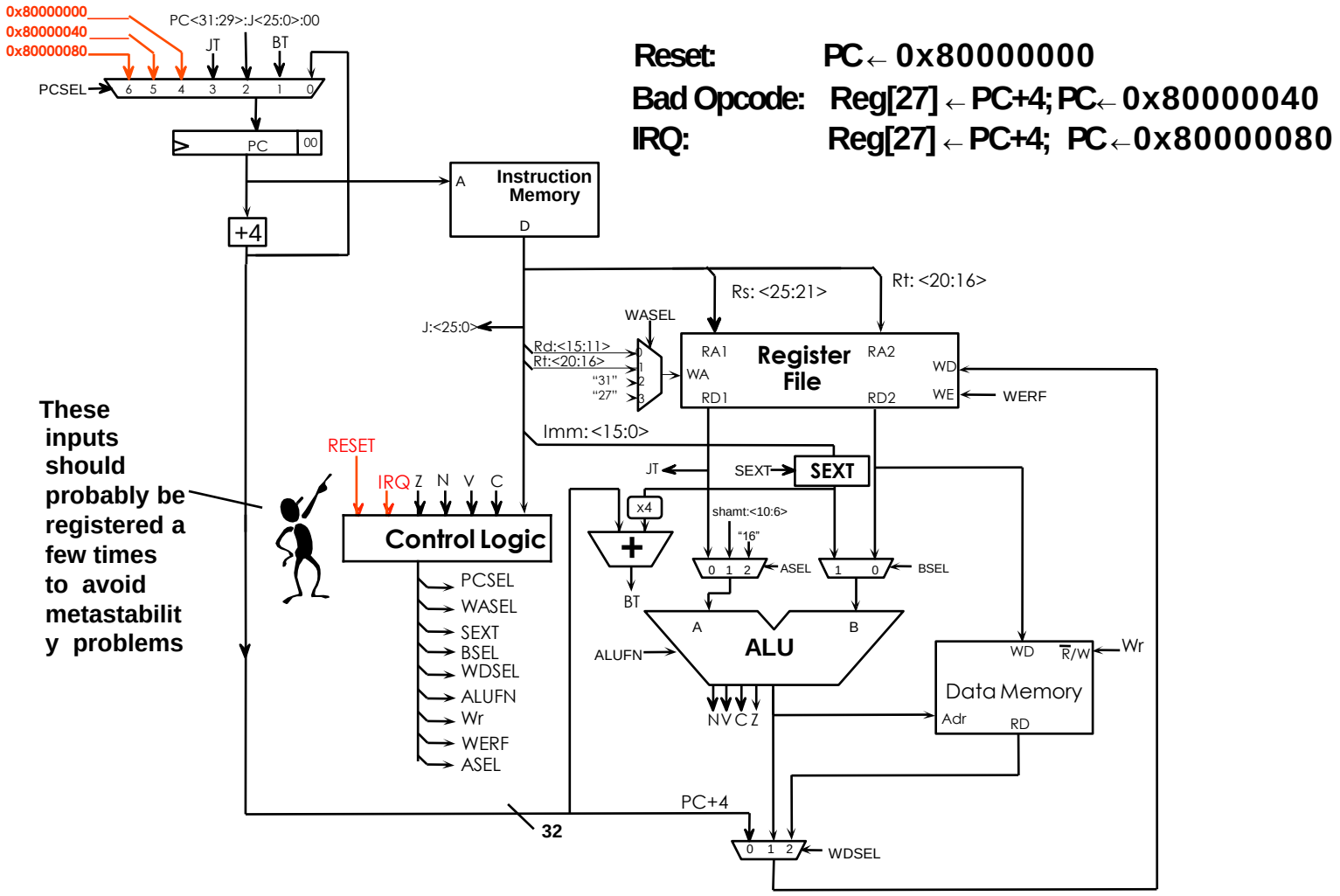
- **TRAPS & system calls (eg, read-a-character)**
  - **CPU generated [*synchronous, caused by an instruction*]**

  (Implemented as an "agreed" upon Illegal instruction)
- **I/O events (eg, key press)**
  - **externally generated [*asynchronous*]**

EXCEPTION GOAL: Interrupt running program, invoke exception handler, return to continue execution.

These are "Software" notions of synchrony and asynchrony.

Exceptions



These inputs should probably be registered a few times to avoid metastability problems

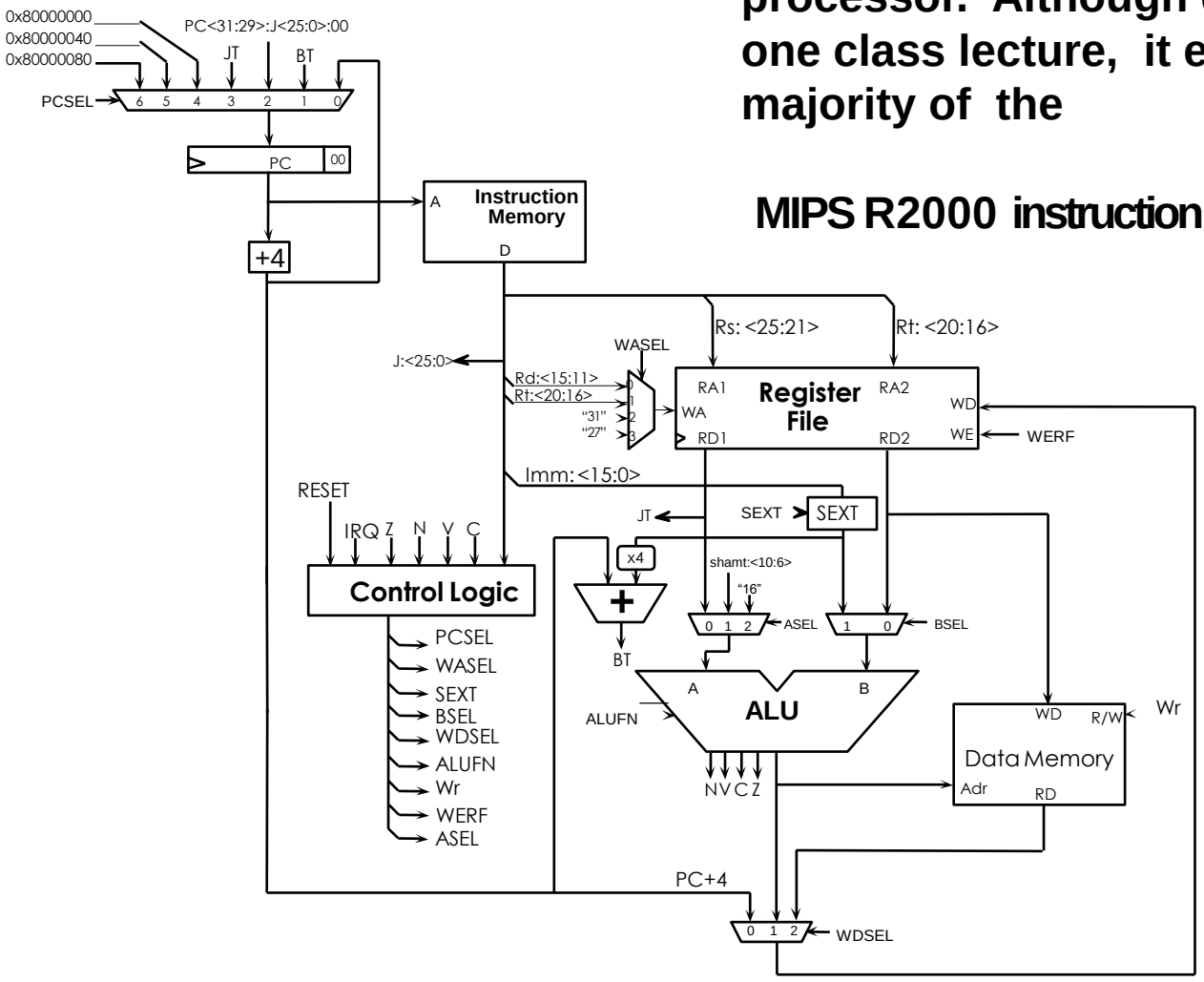
Reset: PC ← 0x80000000

Bad Opcode: Reg[27] ← PC+4; PC ← 0x80000040

IRQ: Reg[27] ← PC+4; PC ← 0x80000080

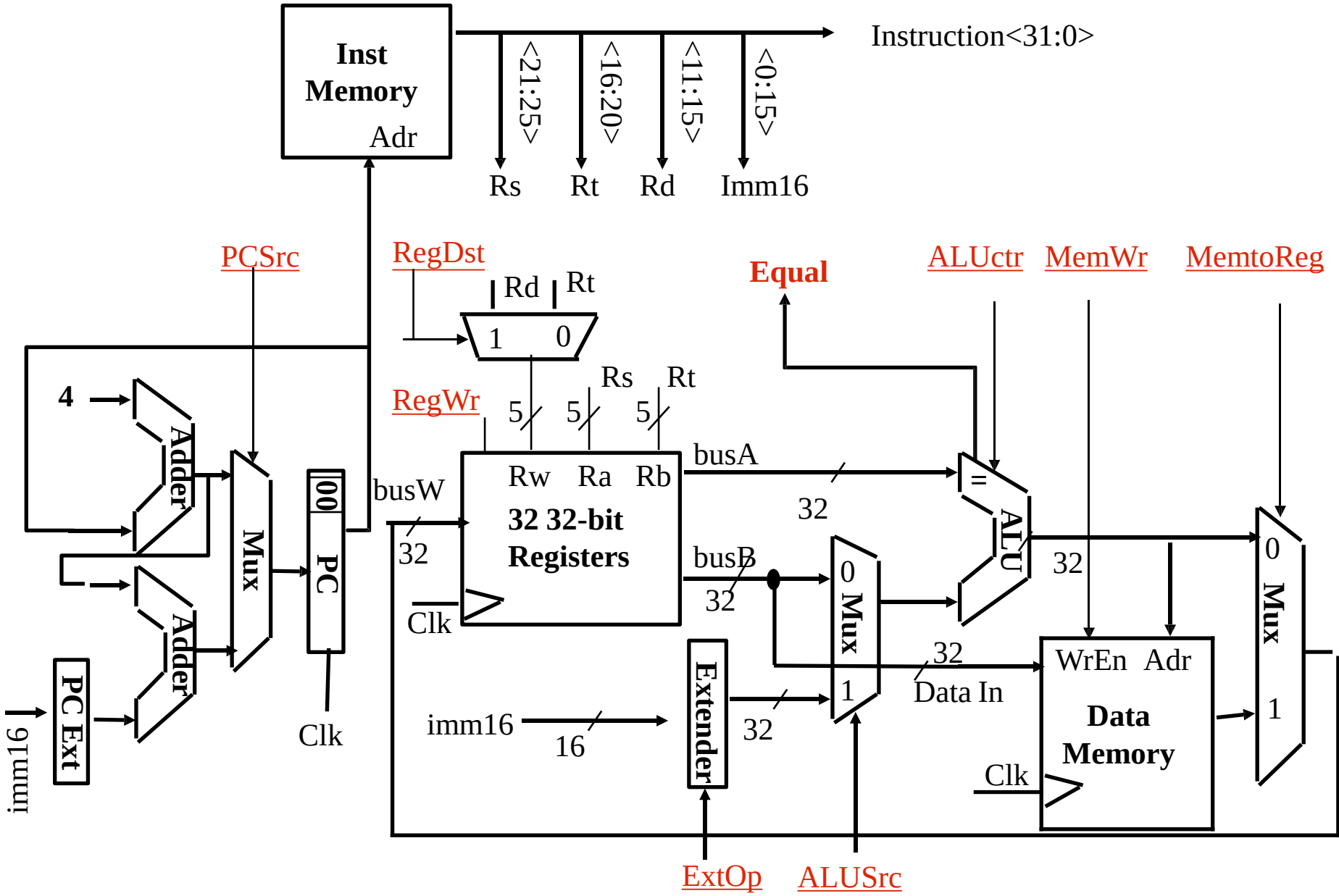
This is a complete 32-bit processor. Although designed in one class lecture, it executes the majority of the

MIPS R2000 instruction set.



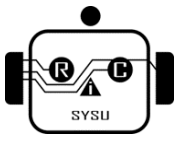
- **Executes one instruction per clock**
- **All that's left is the control logic design**

单周期完整设计–简化版


Inst
Memory
Adr
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rs
Rt
Rd
Imm16
PCSrc
RegDst
Rd
Rt
1
0
Equal
ALUctr
MemWr
MemtoReg
4
Adder
Mux
00
PC
RegWr
5
5
5
Rs
Rt
busW
Rw
Ra
Rb
32 32-bit
Registers
busA
32
busB
32
Clk
32
=
ALU
32
0
Mux
1
0
Mux
1
PC Ext
Adder
Clk
imm16
16
Extender
32
32
WrEn
Adr
Data In
Data
Memory
Clk
imm16
ExtOp
ALUSrc

小结

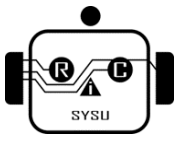
- CPU设计直接决定了时钟周期宽度和CPI，所以对计算机性能非常重要！
- CPU主要由数据通路和控制器组成
  - 数据通路：实现指令集中所有指令的操作功能
  - 控制器：控制数据通路中各部件进行正确操作
- 数据通路的两种元件
  - 操作元件(组合电路)：ALU、MUX、Ext.、Adder、Reg/Mem Read
  - 状态 / 存储元件(时序电路)：PC、Reg/Mem Write
- 数据通路的定时
  - 数据通路中操作元件没有存储功能，其操作结果须写到存储元件中
  - 在时钟到达后clk-to-Q时存储元件开始更新状态
- MIPS指令集的一个子集作为CPU的实现目标
  - 公共操作：取指令和PC+4
  - 下址计算：32位PC,四路选择：顺序、Branch、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



补充符号扩展问题Mips体系结构

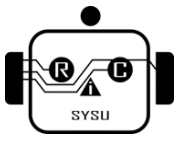
ADDI和ADDIU的实现有以下2个要点：

- 1）仅支持有符号运算（补码运算），16位立即数需要进行符号扩展；
- 2）ADDI带有溢出检测（或者说带“自陷”功能，虽然不常用）。
- ADDI和ADDIU的最大区别是有没有溢出标志（overflow）。
- 算术运算符号扩展
- 逻辑运算0扩展

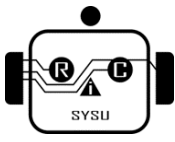
单周期控制器的设计

- 考察每条指令在数据通路中的执行过程和涉及到的控制信号取值
  - 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
  - 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字段间的关系
- 设计ALU局部控制单元
- 设计主控制单元



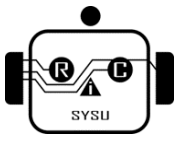
控制器的设计

- 功能
  - 给定一个指令字...
  - 产生执行该指令所需的控制信号
- 实现方式：组合逻辑电路
  - 输入信号
    - 指令字段的op和func。
    - ALU的zero、V、N状态



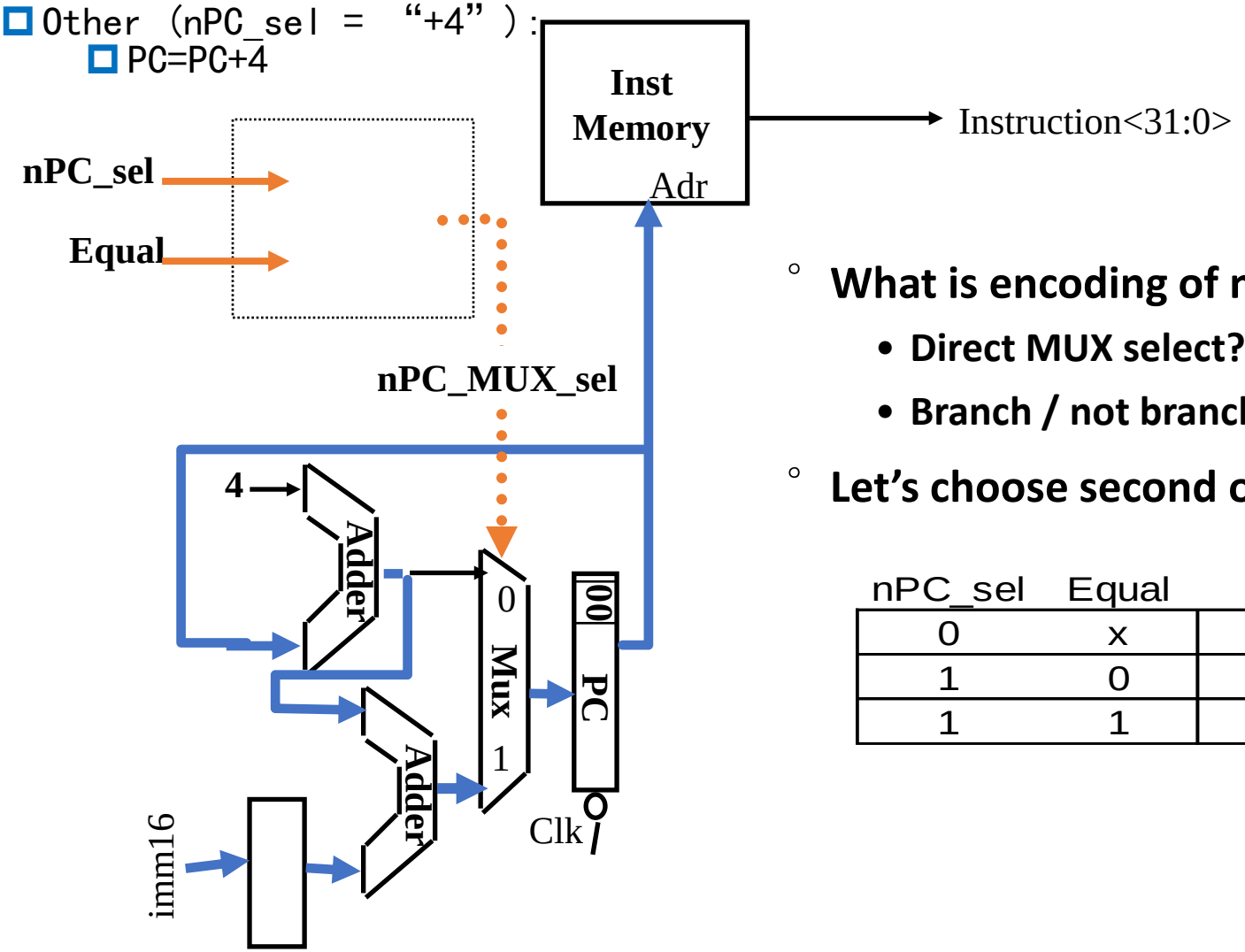
控制器的设计

- 确定每条指令的控制信号
  - 0
  - 1
  - X(与该指令无关)
- 构建控制信号的真值表



Recap: Flexible Instruction Fetch

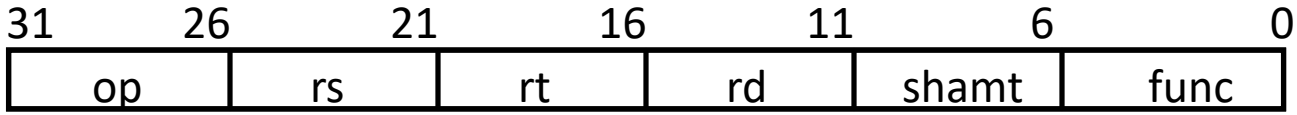
- Branch (nPC_sel = "Br"):
  - if (Equal == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4
- Other (nPC_sel = "+4"):
  - PC=PC+4



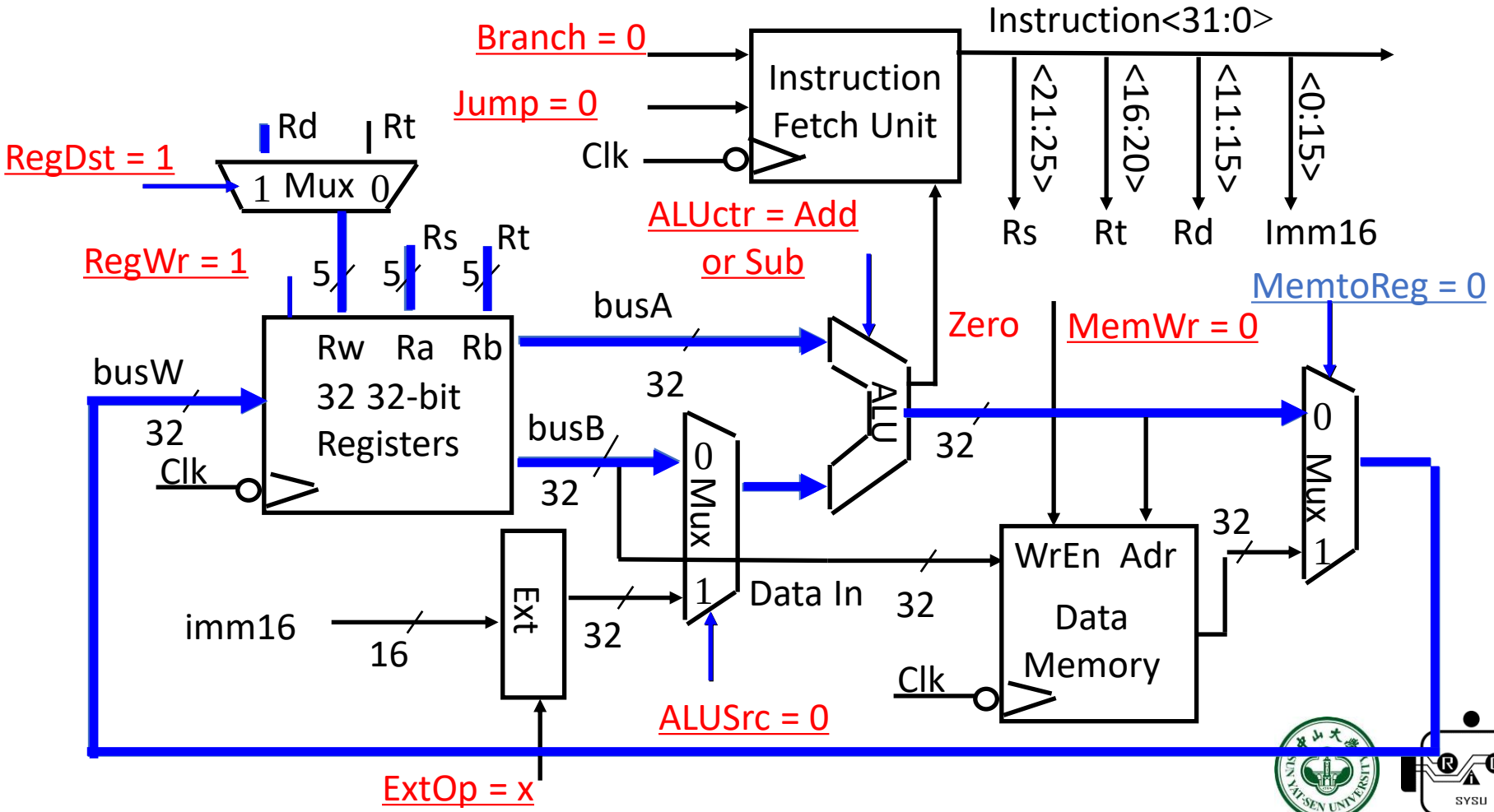
- What is encoding of nPC_sel?
  - Direct MUX select?
  - Branch / not branch
- Let's choose second option

nPC_sel	Equal	MUX
0	x	0
1	0	0
1	1	1

Recap: The Single Cycle Datapath during Add

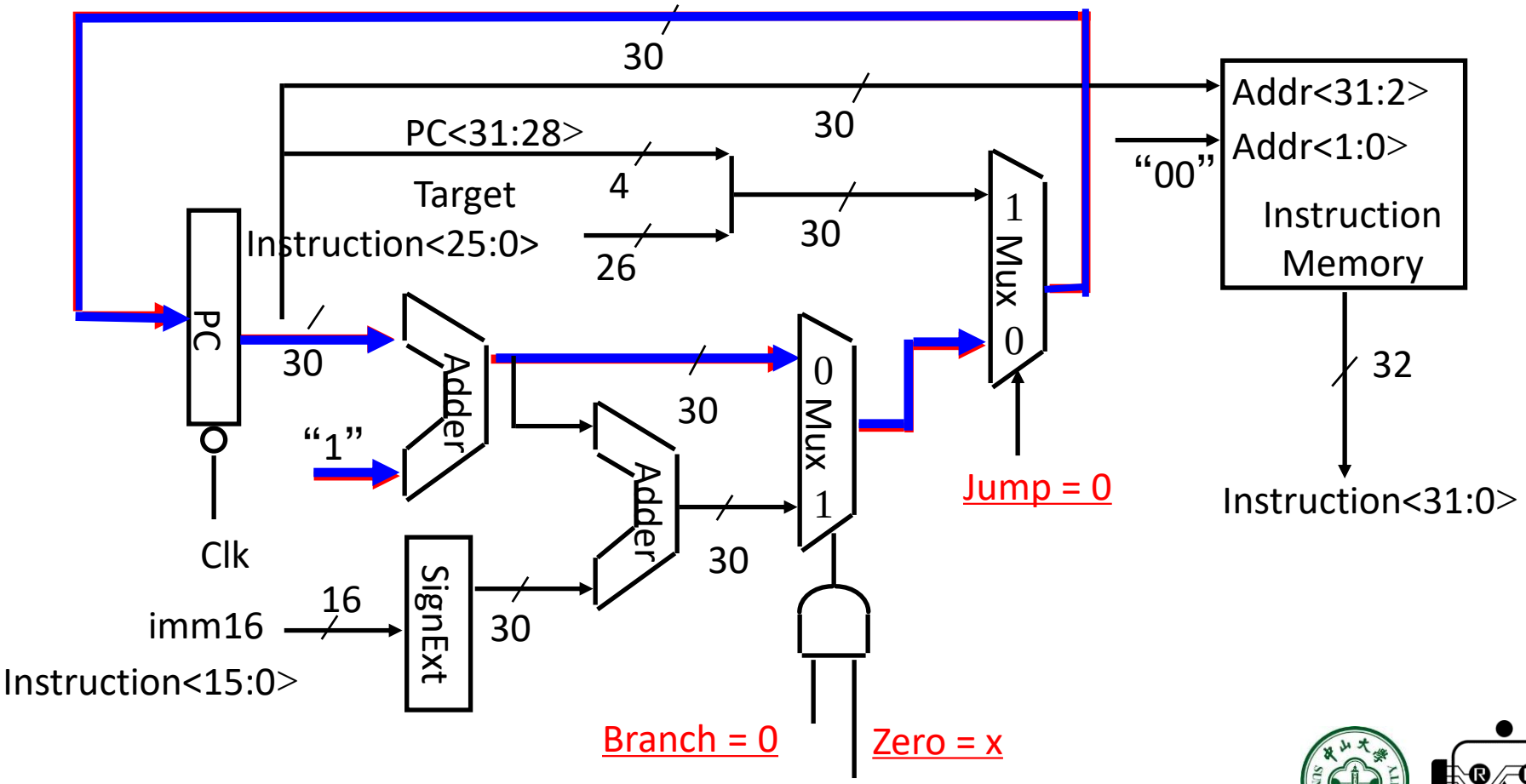


- R[rd] ← R[rs] + / − R[rt]

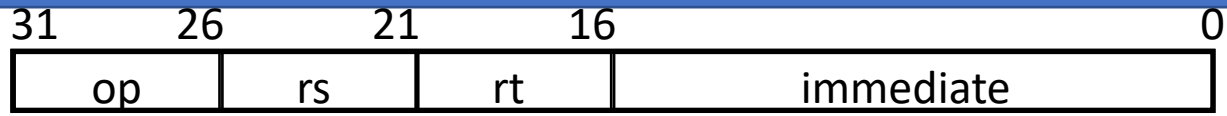


Instruction Fetch for R-type Instruction

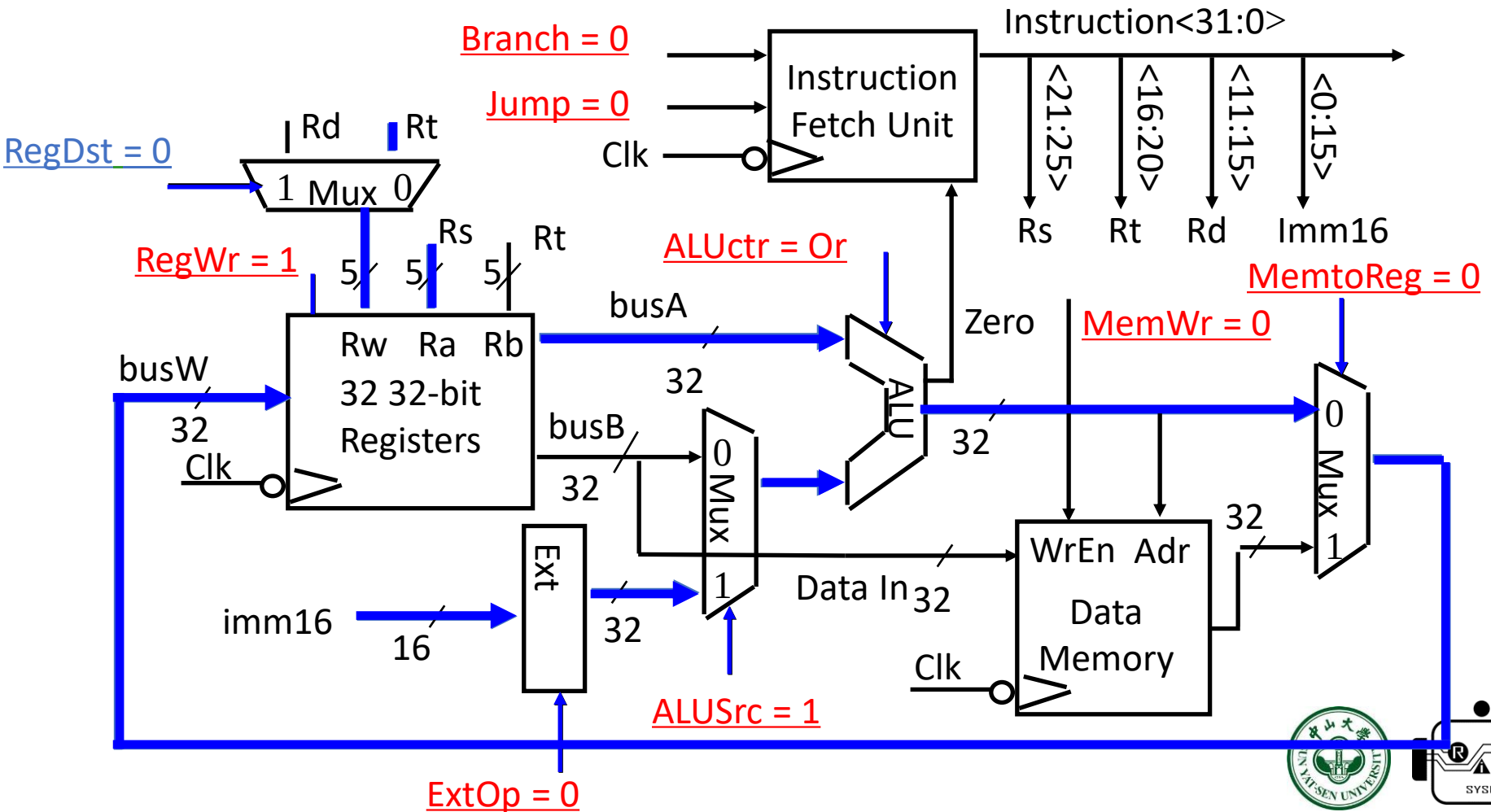
- PC ← PC + 4
  - All of the instructions except Branch and Jump are the same



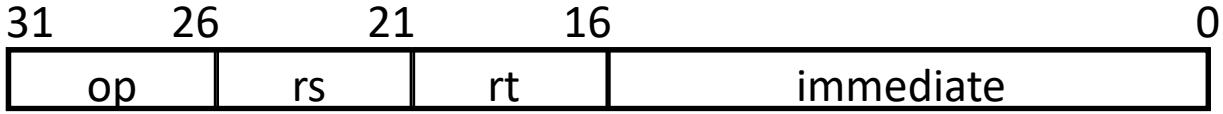
Recap: The Single Cycle Datapath during Or Immediate



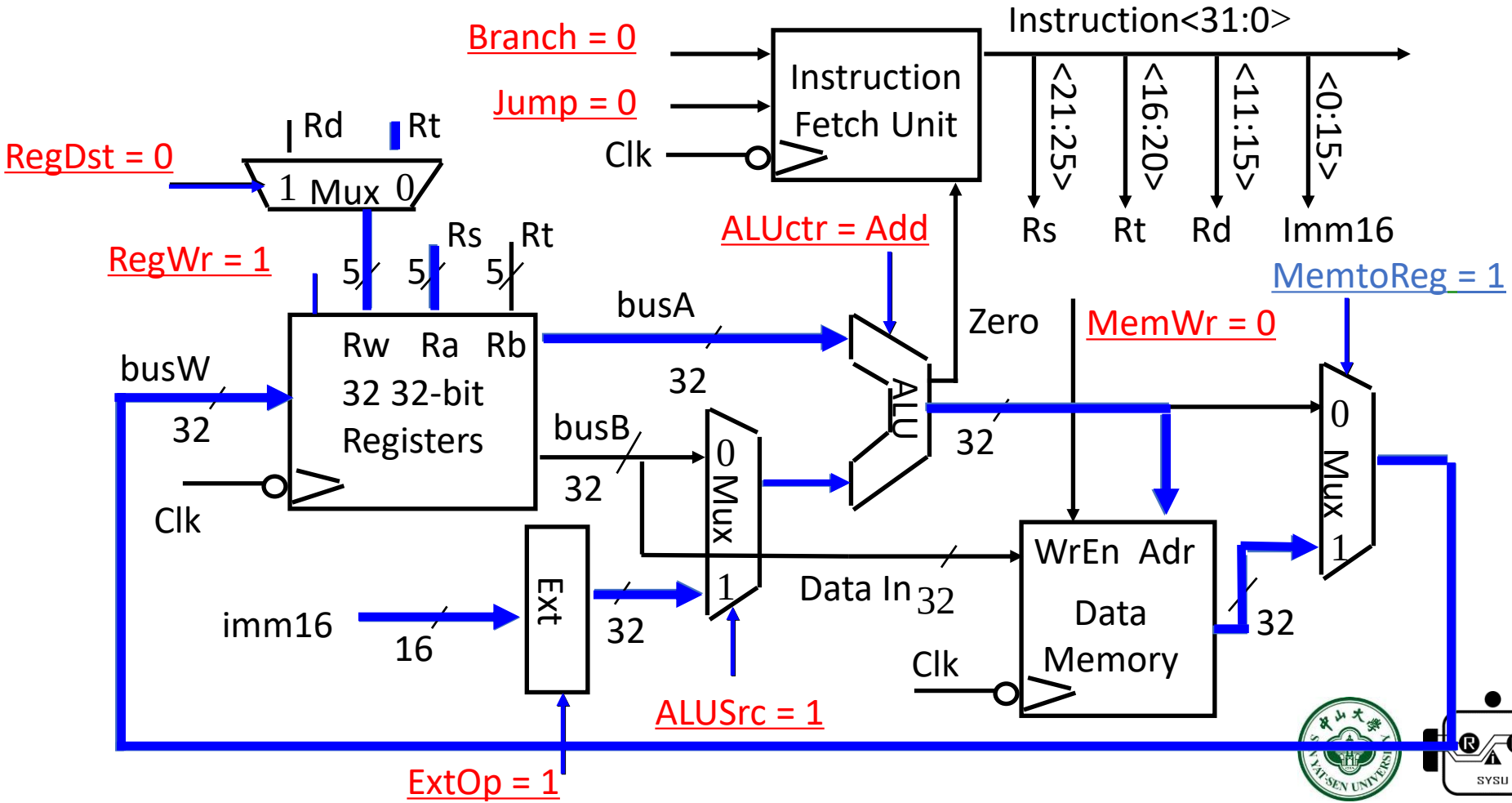
- R[rt] ← R[rs] or ZeroExt[Imm16]



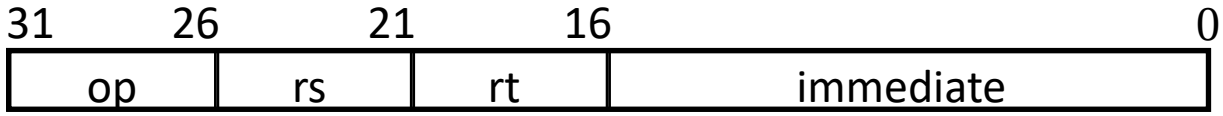
Recap: The Single Cycle Datapath during Load



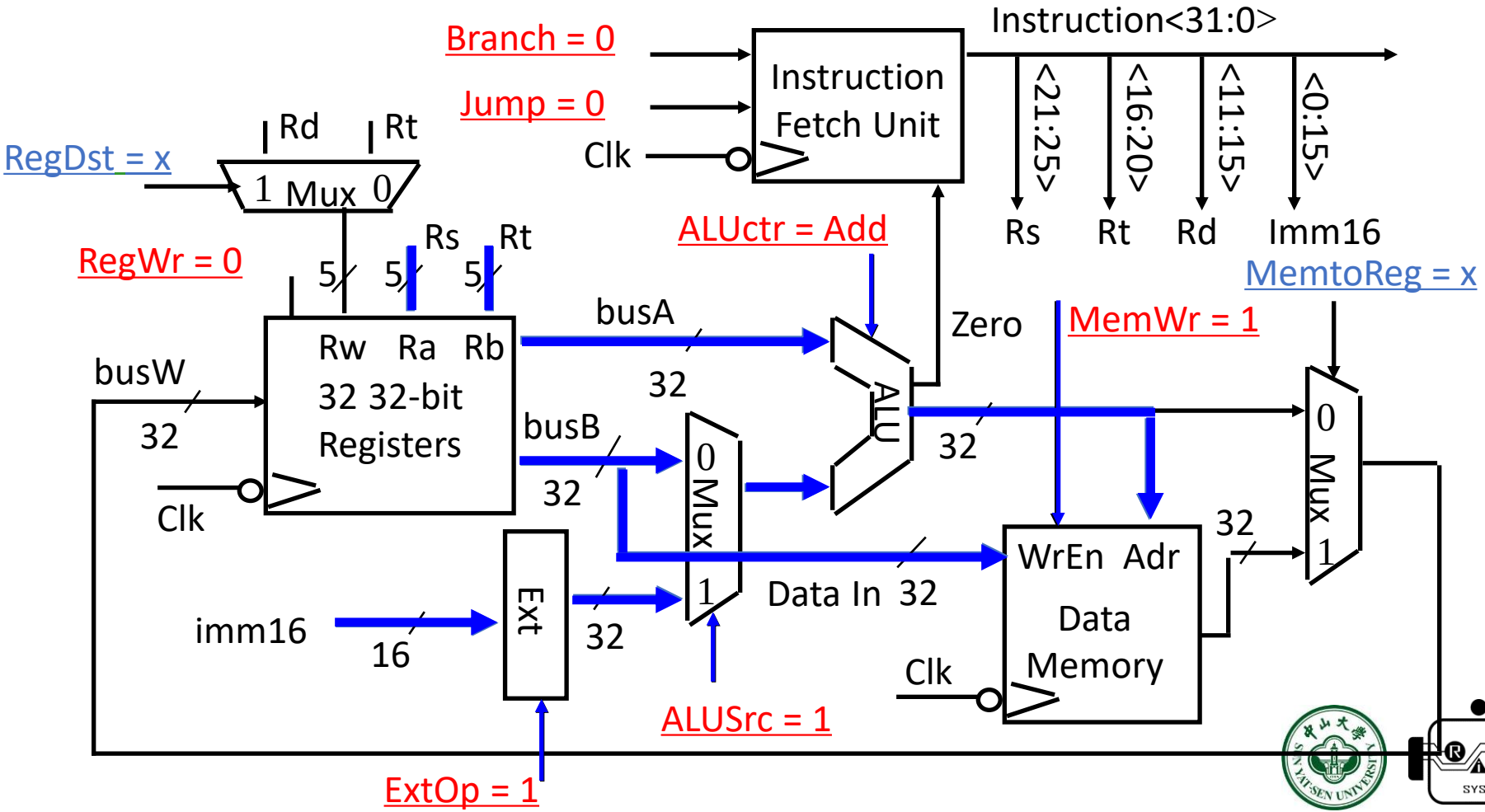
- R[rt] ← Data Memory {R[rs] + SignExt[imm16]}



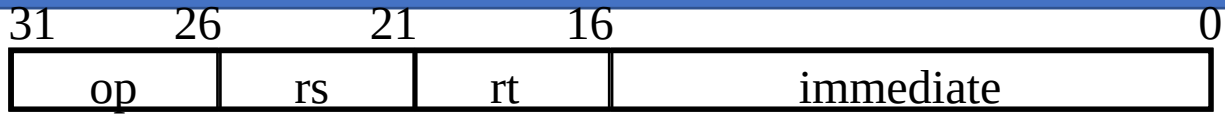
Recap: The Single Cycle Datapath during Store



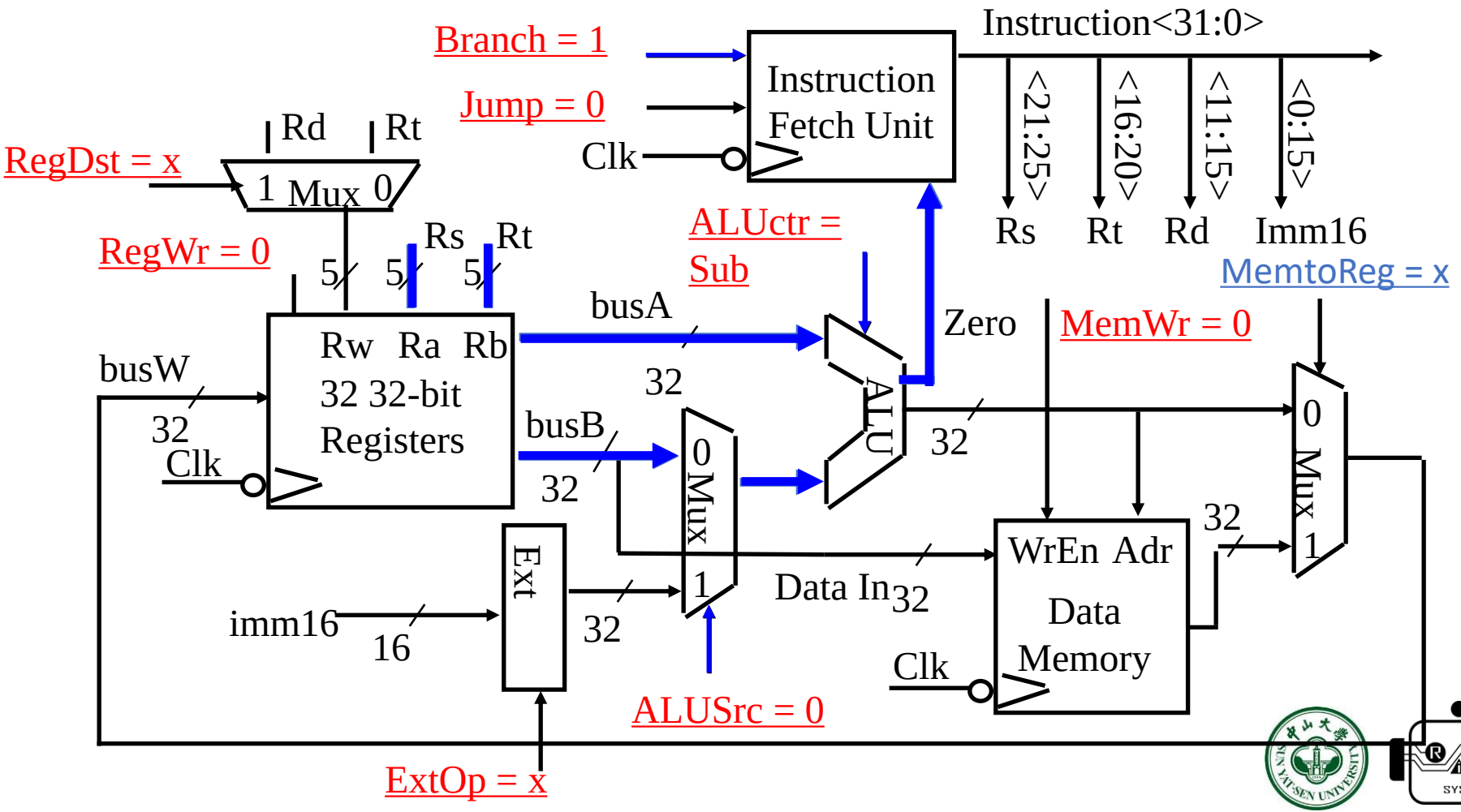
- M{R[rs] + SignExt[imm16]} ← R[rt]



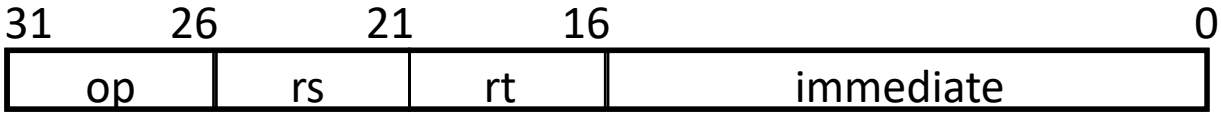
Recap: The Single Cycle Datapath during Branch



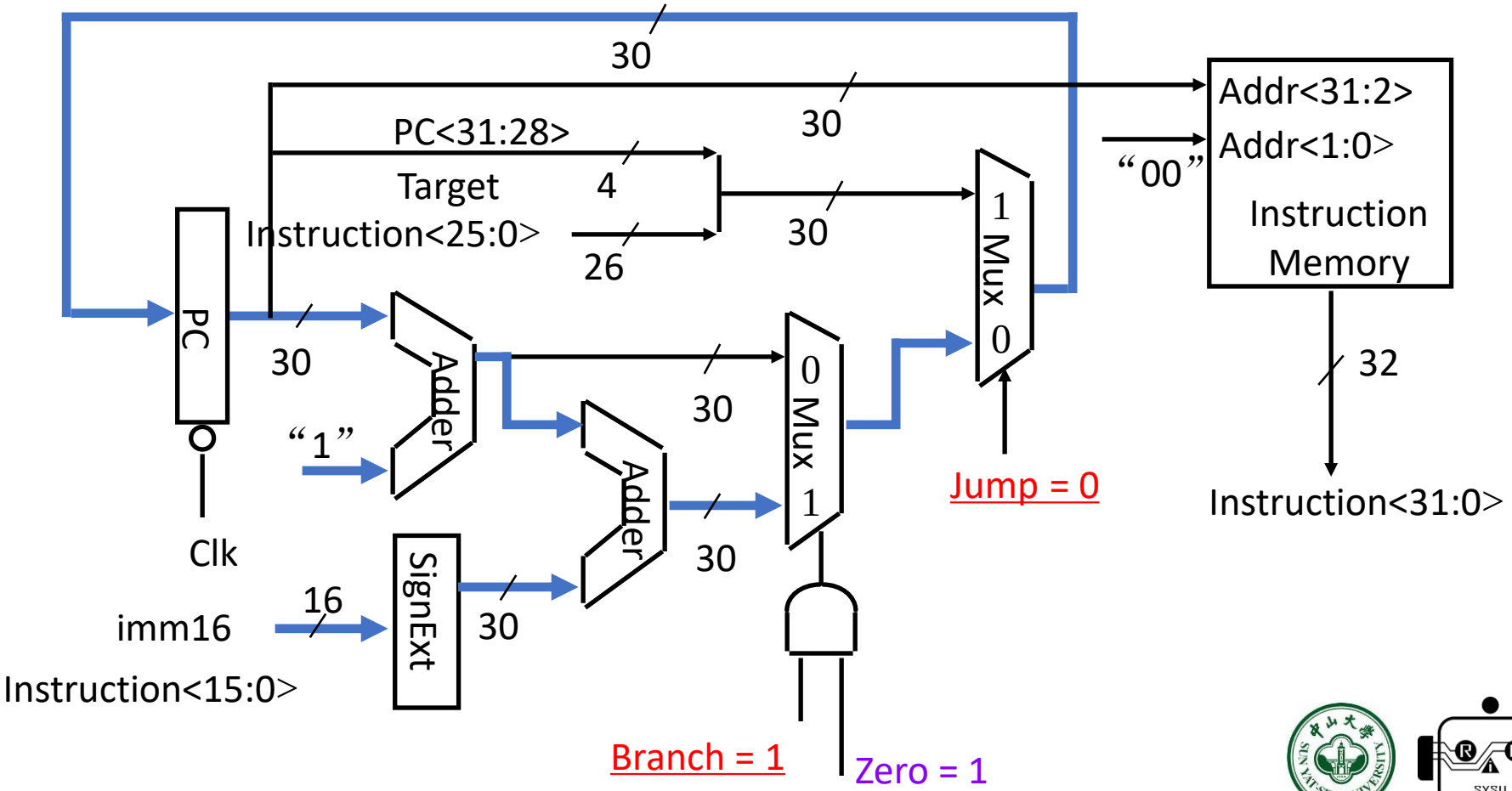
- if (R[rs] - R[rt] == 0) then Zero ← 1 ; else Zero ← 0



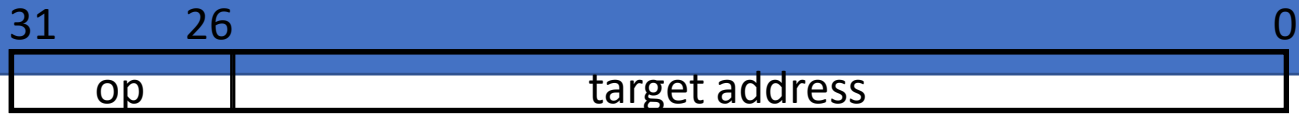
Instruction Fetch for Branch Instruction



- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



Recap: The Single Cycle Datapath during Jump

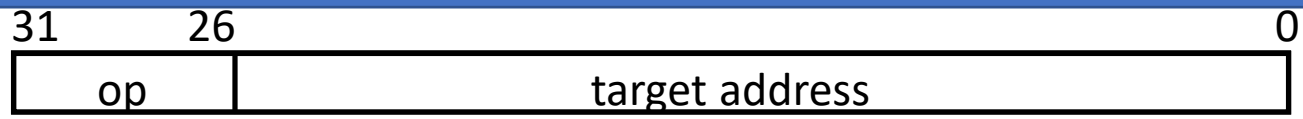


- Send the JUMP address to PC without any additional operation!

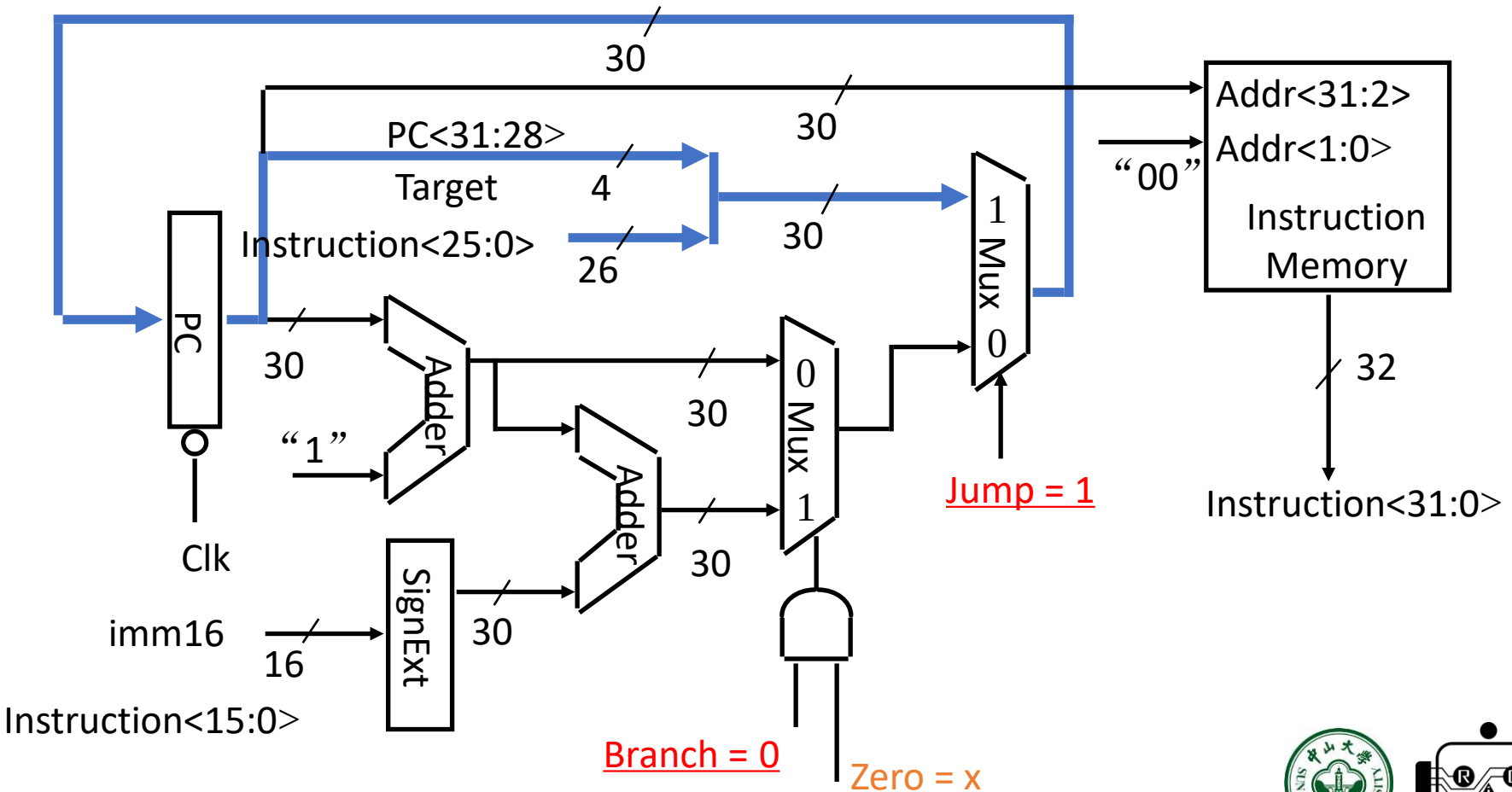
How to prevent the MEM and Reg writing ?

Branch = 0
Jump = 1
Clk
Instruction Fetch Unit
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rs
Rt
Rd
Imm16
RegDst = x
Rd
Rt
1 Mux 0
RegWr = 0
5
5
5
Rs
Rt
ALUctr = x
Zero
MemtoReg = x
MemWr = 0
busA
32
busW
32
Rw Ra Rb
32 32-bit
Registers
Clk
busB
32
0
Mux
1
ALU
32
0
Mux
1
32
WrEn Adr
Data
Memory
Data In
32
Clk
imm16
16
Ext
32
ALUSrc = x
ExtOp = x

Instruction Fetch for Jump Instruction

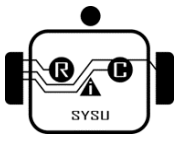


- PC ← PC<31:29> concat target<25:0> concat "00"

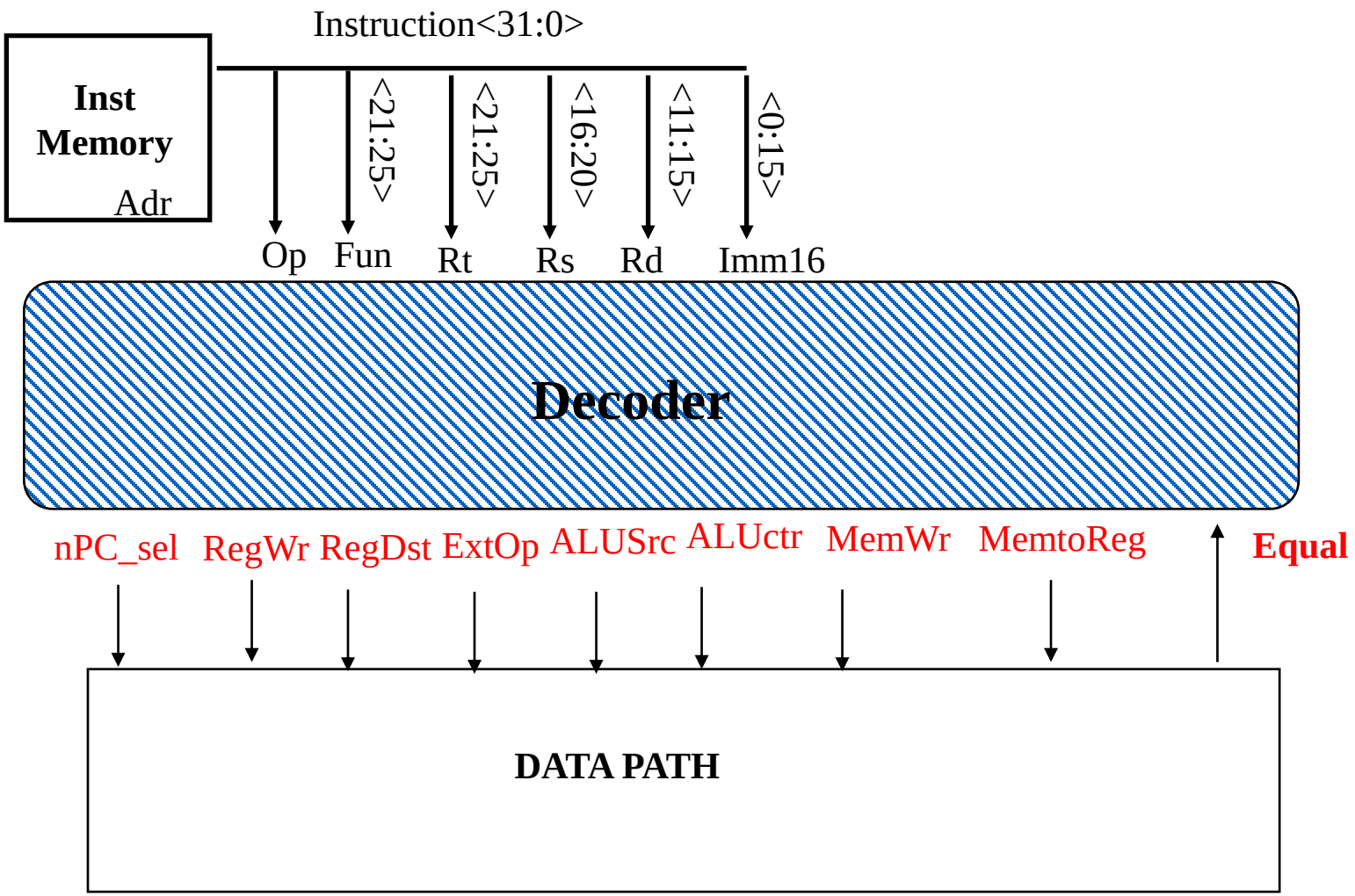


Recap: A Summary of Control Signals

inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr, nPC_sel = “+4”	
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = “Br”, ALUctr = “sub”	

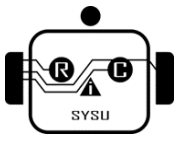


Assemble Control logic

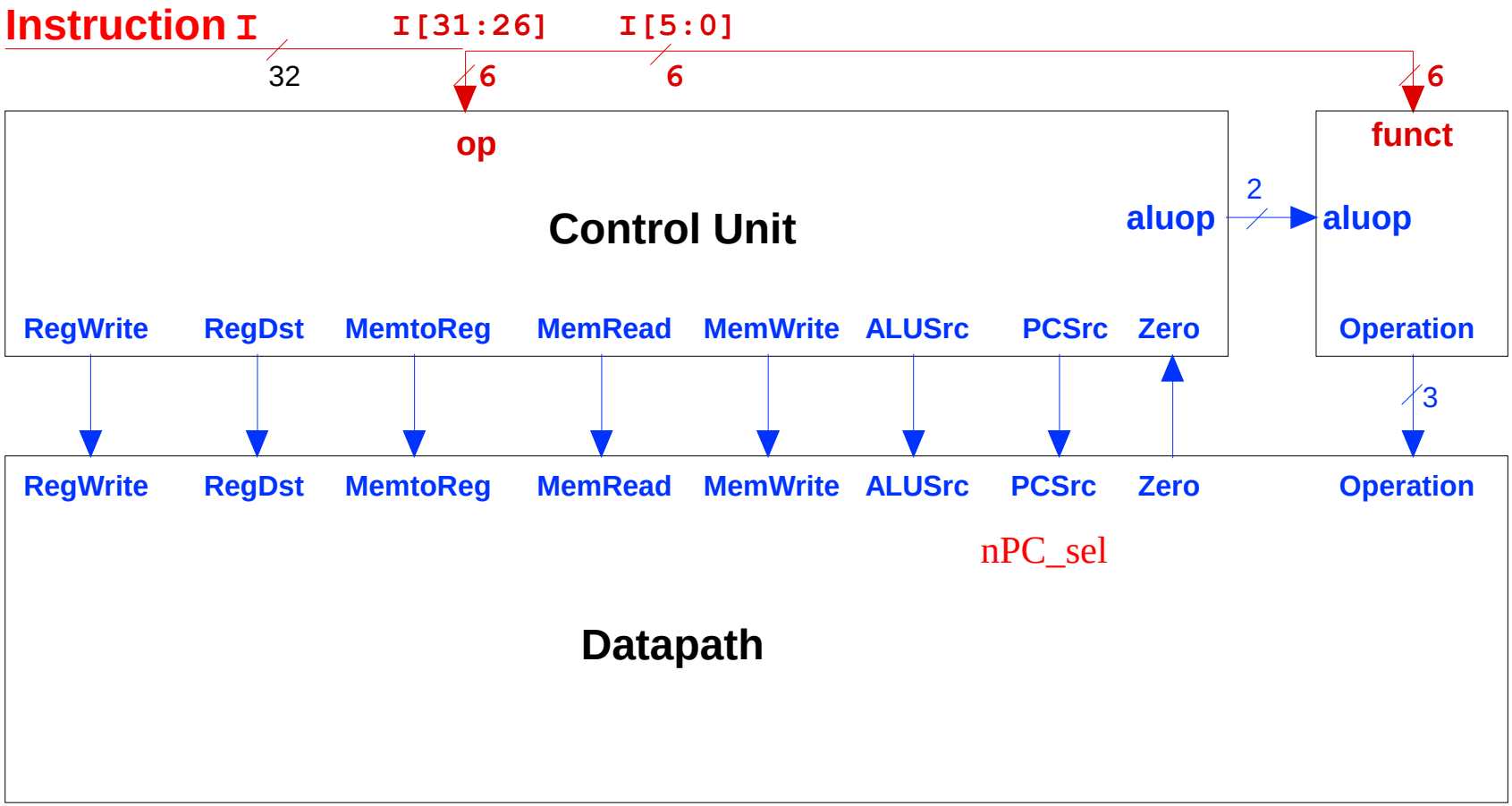


控制器的设计

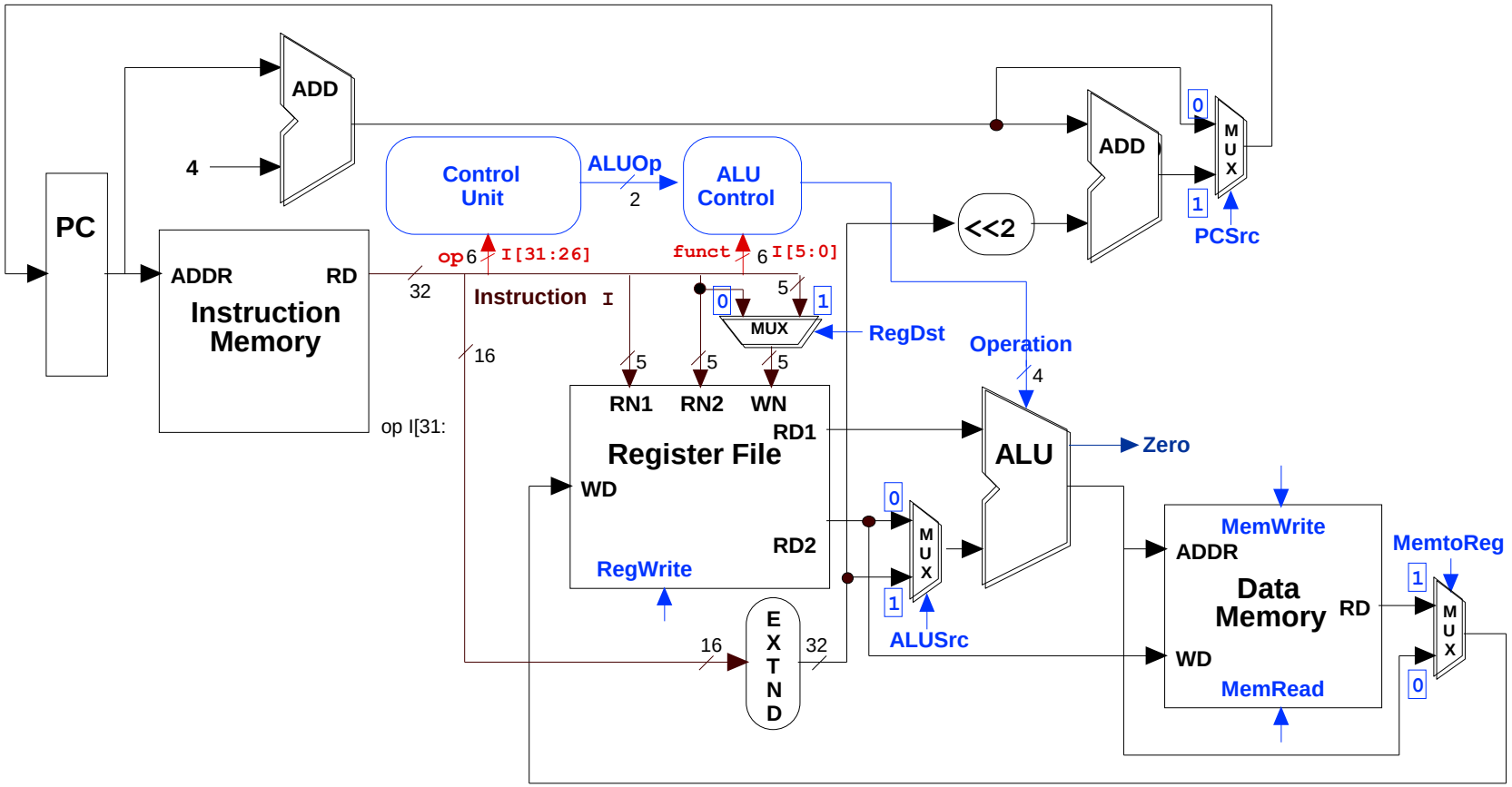
- 控制器：一个宠大的逻辑电路
- 好的方法：分成多个较小的逻辑电路
  - 较小规模的逻辑电路速度更快
  - 较小规模的逻辑电路更容易协同工作
- 显然
  - funct字段只与ALU的operation有关
  - 好的方法：建立一个单独的ALU控制电路



建立一个单独的ALU控制电路



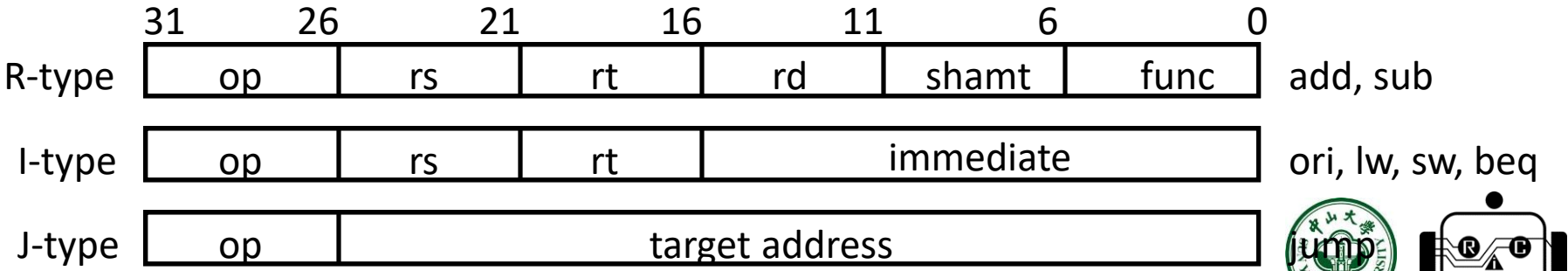
控制器修改后的数据路径



A Summary of the Control Signals

Ori rt,rs,imm
sw rt , offset(rs)
lw rt , offset(rs)

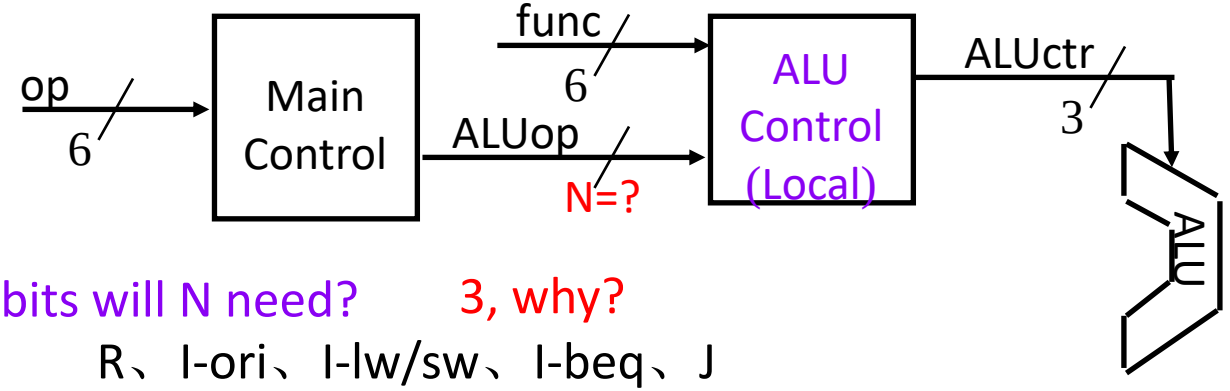
	func	10 0000	10 0010	We Don't Care :-)				
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
		add	sub	ori	lw	sw	beq	jump
RegDst		1	1	0	0	x	x	x
ALUSrc		0	0	1	1	1	0	x
MemtoReg		0	0	0	1	x	x	x
RegWrite		1	1	1	1	0	0	0
MemWrite		0	0	0	0	1	0	0
Branch		0	0	0	0	0	1	0
Jump		0	0	0	0	0	0	1
ExtOp		x	x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



The Concept of Local Decoding

Two levels of decoding: Main Control and ALU Control

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/Subtr	Or	Add	Add	Subtr	xxx

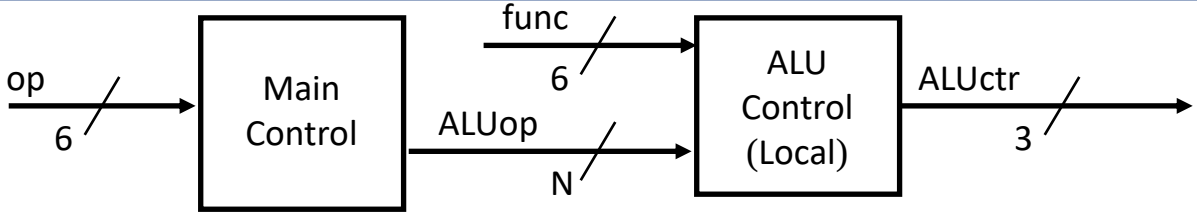


ALUctr is determined by ALUop and func, while other control signals are determined by op。

How many bits will N need? 3, why?

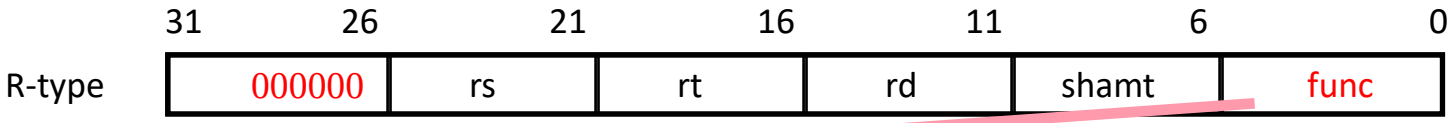
R、I-ori、I-lw/sw、I-beq、J

The Decoding of the "func" Field

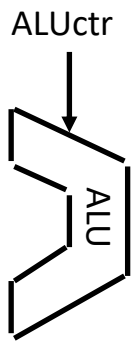


Encoding ALUop as follows

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



func<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



The Truth Table for ALUctr

R-type Instructions determined by funct

Non-R-type Instructions determined by ALUop

ALUop (Symbolic)	R-type	ori	lw	sw	beq
	"R-type"	Or	Add	Add	Subtr
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			func				ALU Operation	ALUctr		
bit2	bit1	bit0	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	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

The Logic Equation for ALUctr<0>

Choose the rows with ALUctr[0]=1

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>

Choose the rows with ALUctr[1]=1

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>

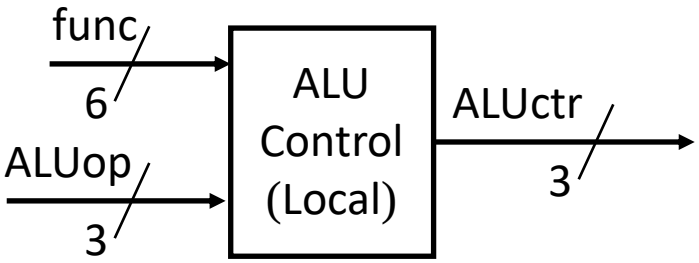
The Logic Equation for ALUctr<2>

Choose the rows with ALUctr[2]=1

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>

Summery of Control Logic of Local Control

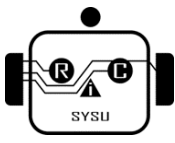


- ALUctr<0> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>
- ALUctr<1> = !ALUop<2> & ALUop<1> & !ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1>
- ALUctr<2> = !ALUop<2> & ALUop<1> & !ALUop<0> +
  ALUop<2> & !func<3> & func<2> & !func<1> & func<0>

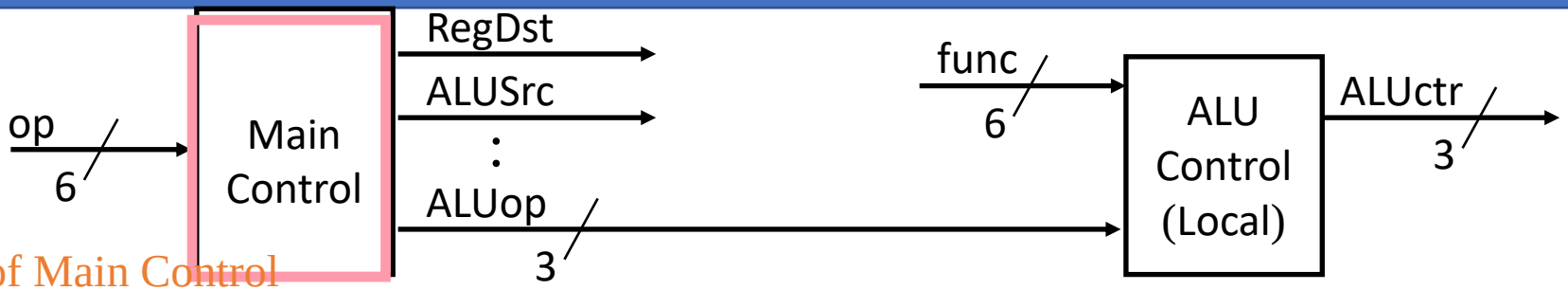
```
module ALUctrl(ALUop, func ,Ctrl);
   input [3:0] ALUop;
   input [3:0] func;
   output [2:0] ALUctrl;
```

```
assign ALUctr<0> = !ALUop[2] & ALUop[0] +
       ALUop[2] & !func[2] & func[1] & !func[0];
assign Ctrl[1] = !ALUop[2] & ALUop[1] & !ALUop[0] +
                 ALUop[2] & !func[3] & func[2] & !func[1] ;

 assign Ctrl[2] = !ALUop[2[ & ALUop[1] & !ALUop[0] +
       ALUop[2] & !func[3] & func[2] & !func[1] & func[0]

endmodule
```

```
module maindec(
    input wire[5:0] op,

    output wire memtoreg,memwrite,
    output wire branch,alusrc,
    output wire regdst,regwrite,
    output wire jump,
    output wire[1:0] aluop
    );
    reg[8:0] controls;
    assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump,aluop} = controls;
    always @(*) begin
        case (op)
            6'b000000:controls <= 9'b110000010;//R-TYRE
            6'b100011:controls <= 9'b101001000;//LW
            6'b101011:controls <= 9'b001010000;//SW
            6'b000100:controls <= 9'b000100001;//BEQ
            6'b001000:controls <= 9'b101000000;//ADDI
            6'b000010:controls <= 9'b000000100;//J
            default:  controls <= 9'b000000000;//illegal op
        endcase
    end
endmodule
```



The "Truth Table" for the Main Control



Output of Main Control

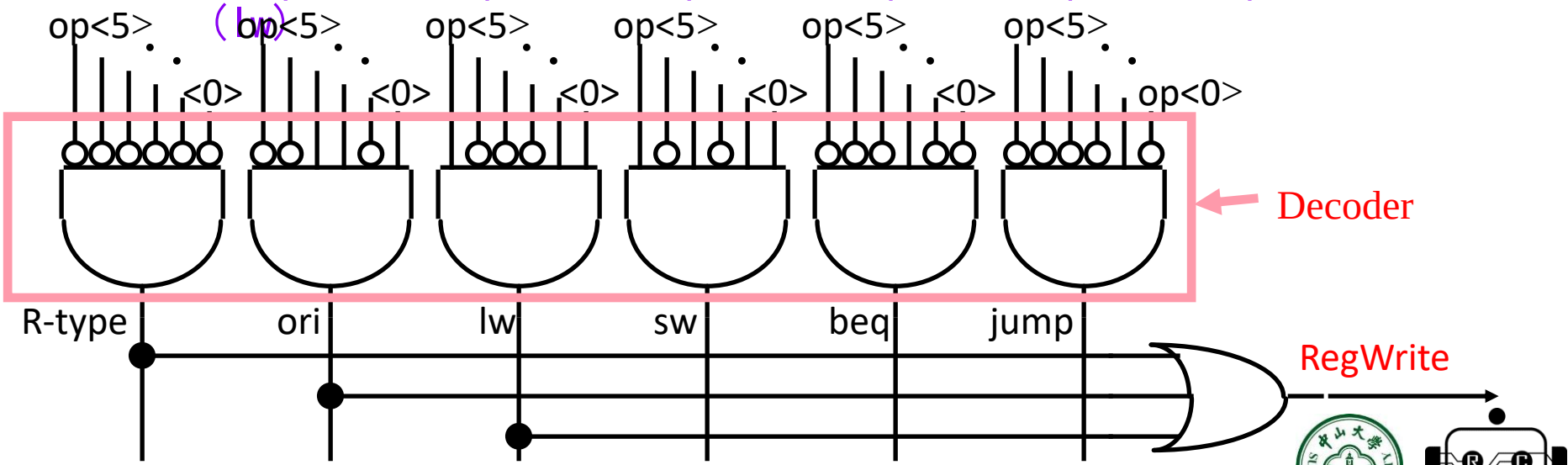
Input of main control

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	Subtr	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

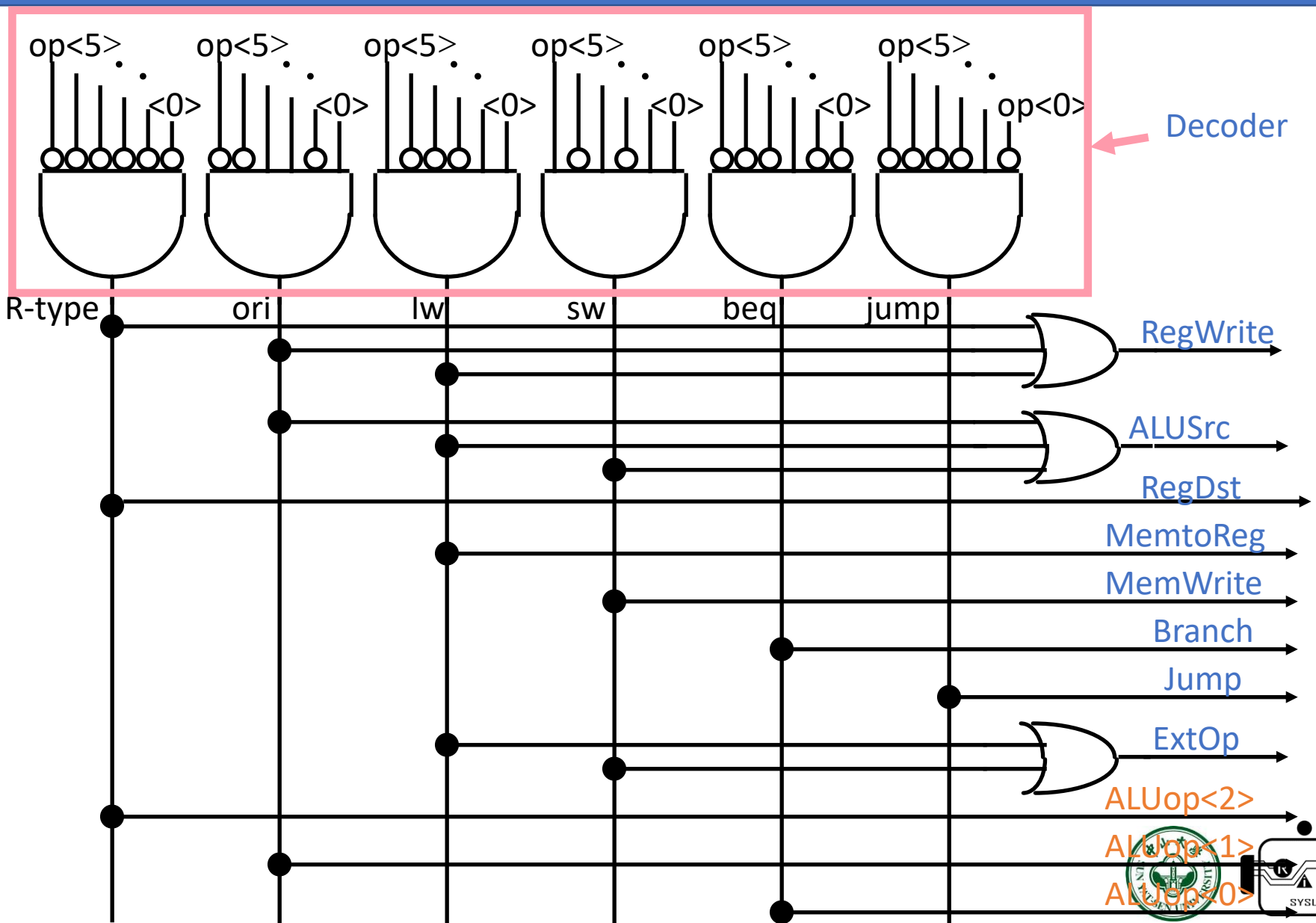
The "Truth Table" for RegWrite

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

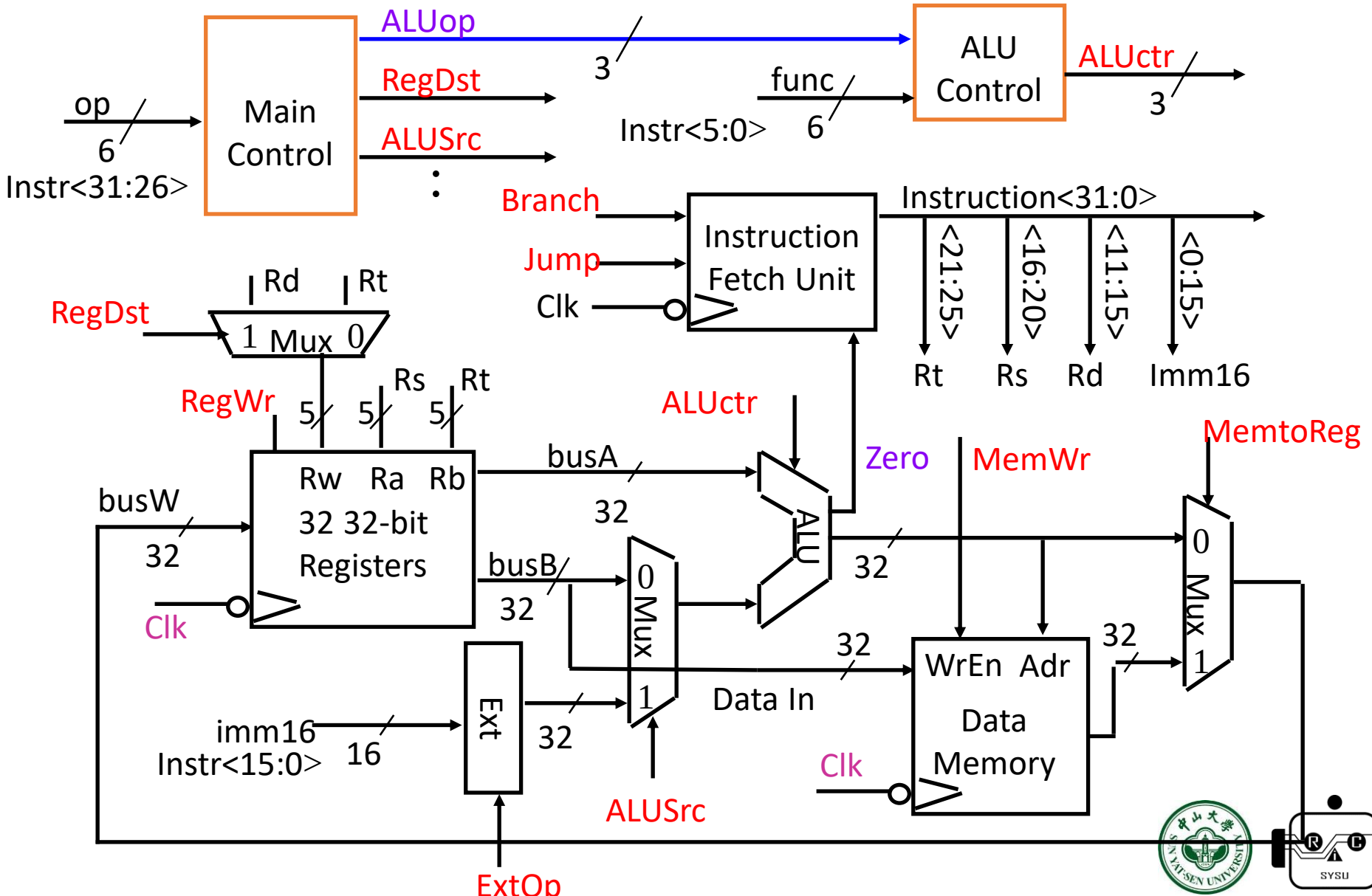
- RegWrite = R-type + ori + lw
  - = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  - + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  - + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



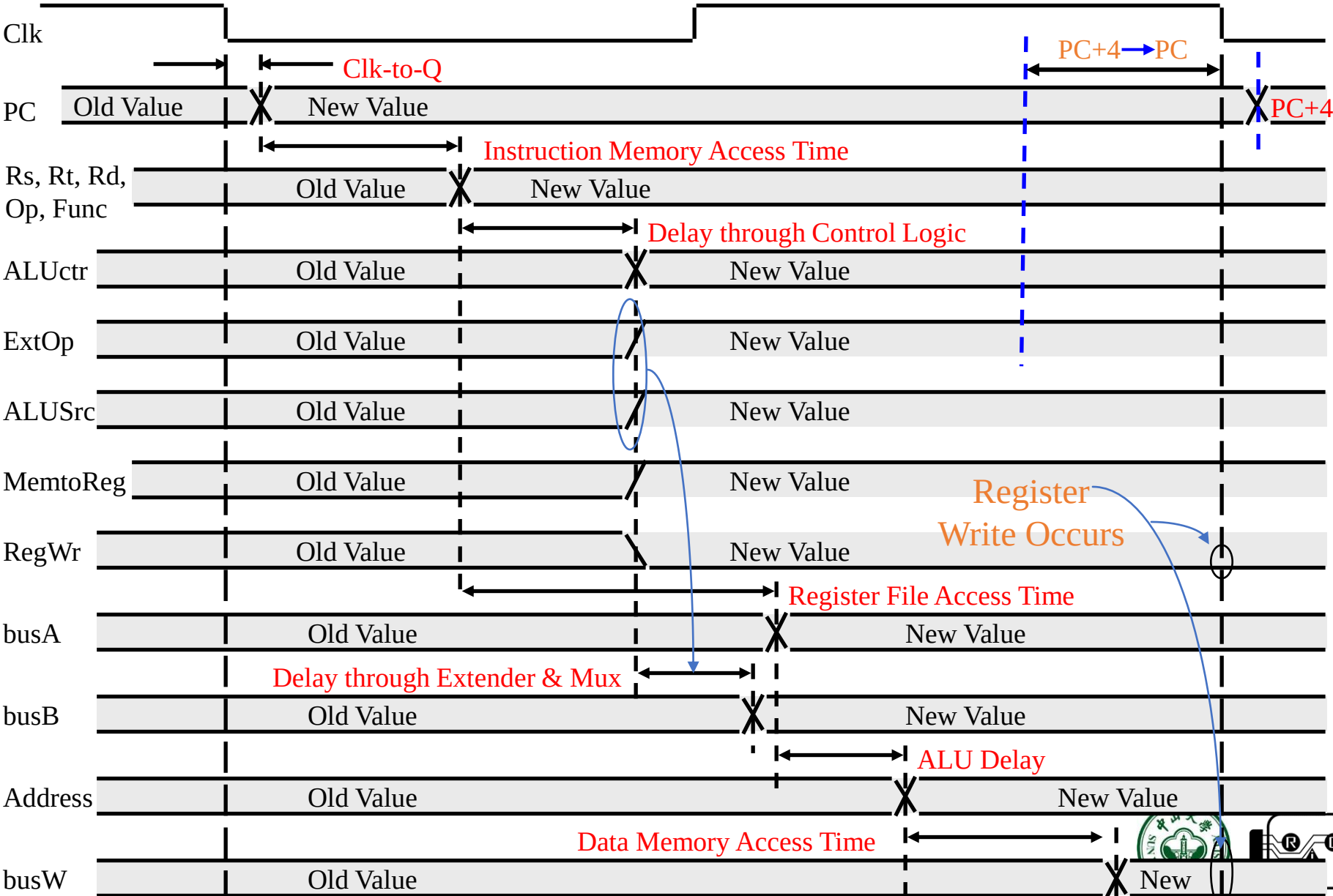
The “Truth Table” for RegWrite



The Complete Single Cycle Data Path with Control

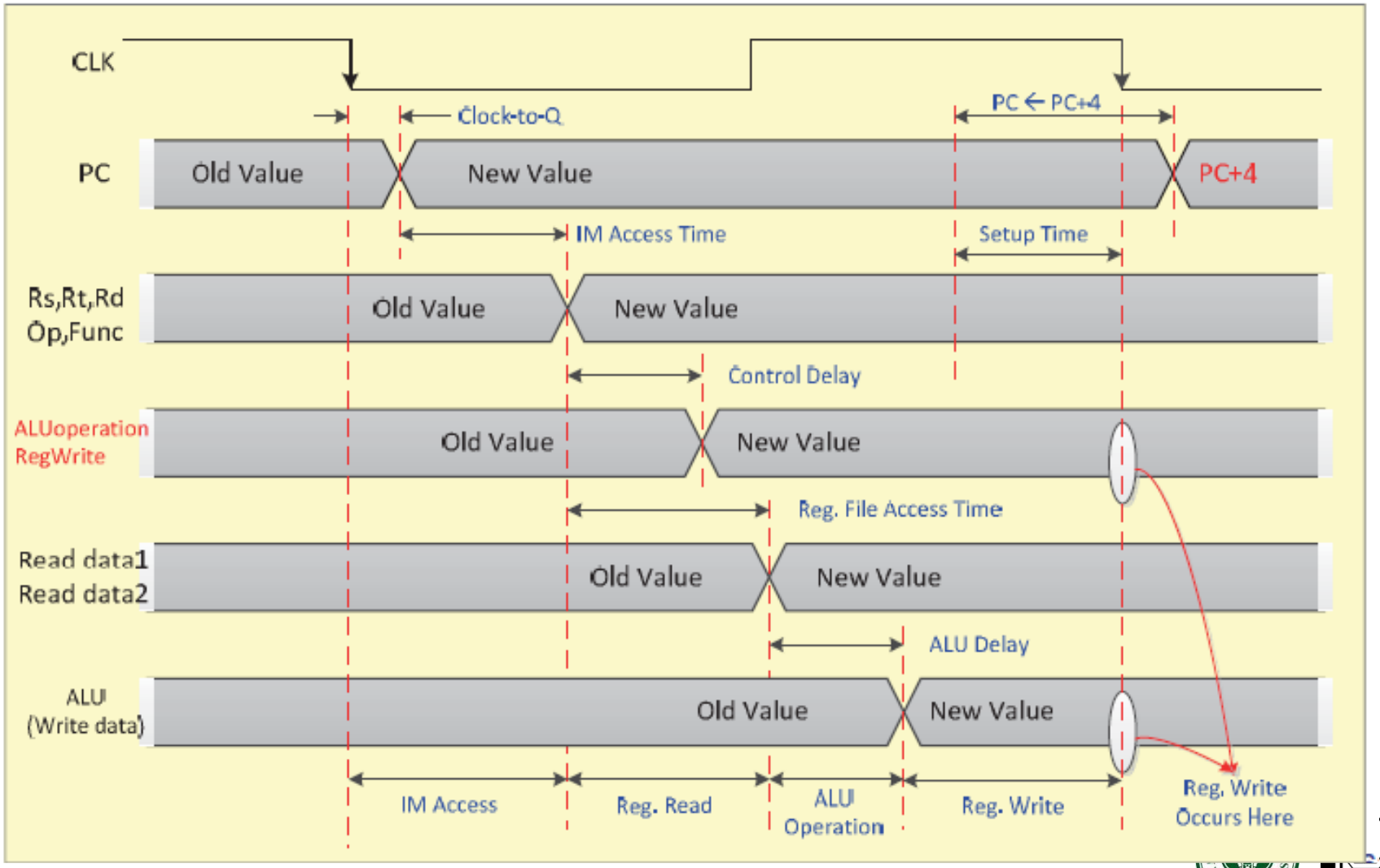


Time Delay for LW: Critical Path



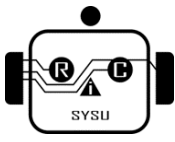
Time Delay for R型指令: Critical Path

- R型指令的指令周期

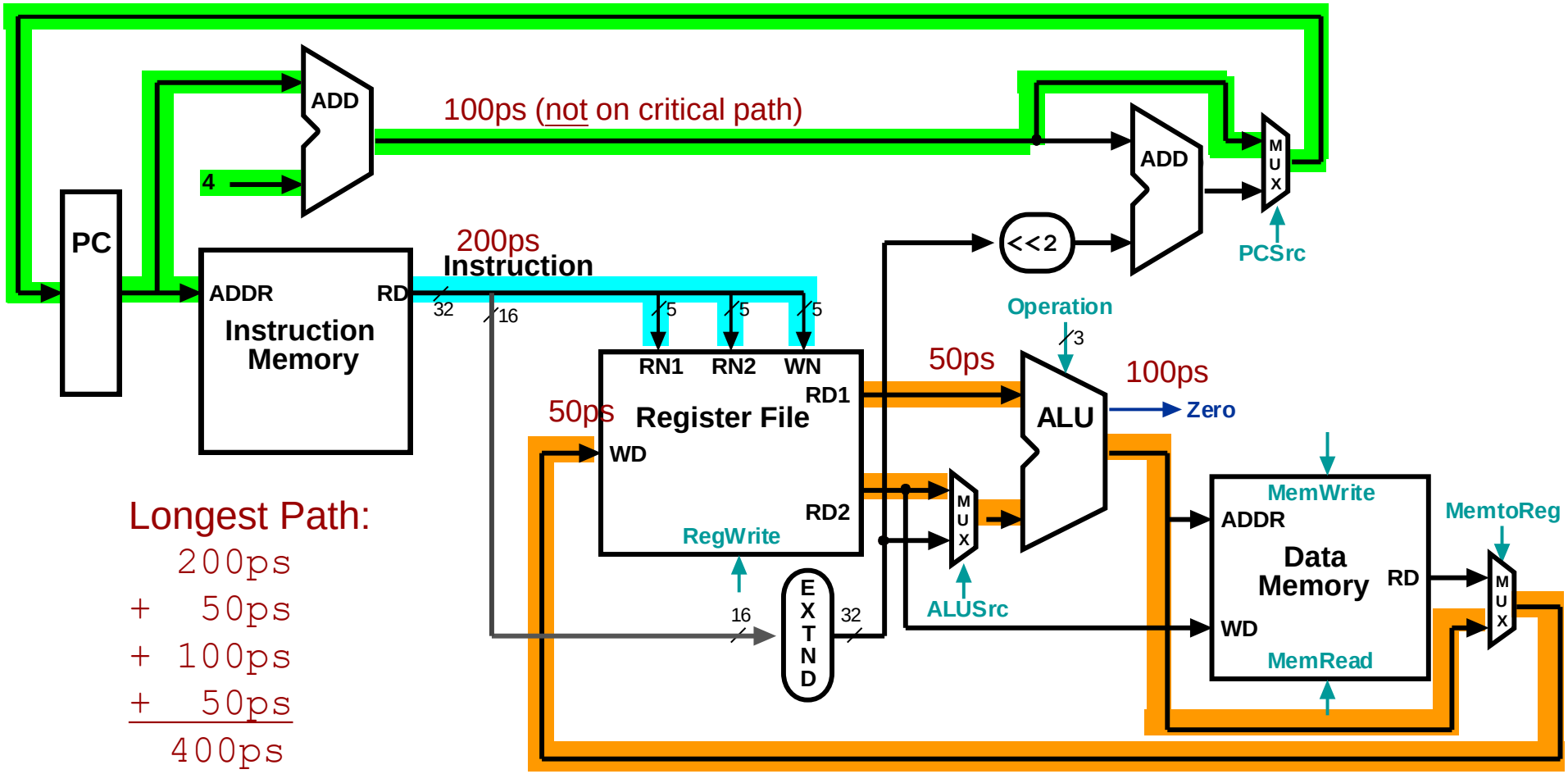


单周期CPU的性能

- 假设各部件延时如下：
  - 存储读/写：200ps
  - ALU、加法器：100ps
  - 寄存器堆读/写：50ps
  - 控制器 -忽略不计：0ps
  - 选择器-忽略不计：0ps
- 各种指令的执行所需的指令周期是多少?



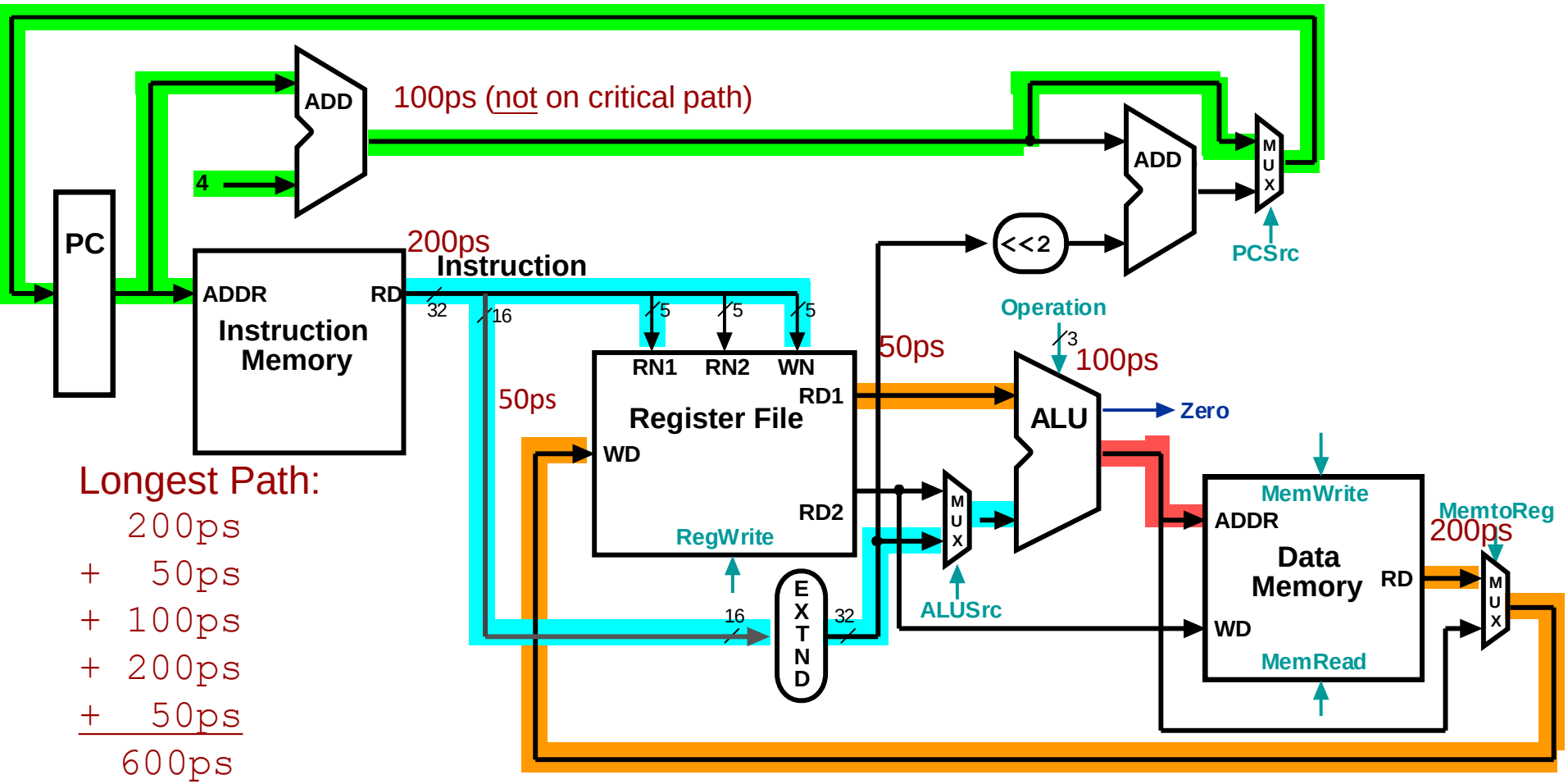
R型指令



Longest Path:

```
  200ps
+  50ps
+ 100ps
+  50ps
  400ps
```

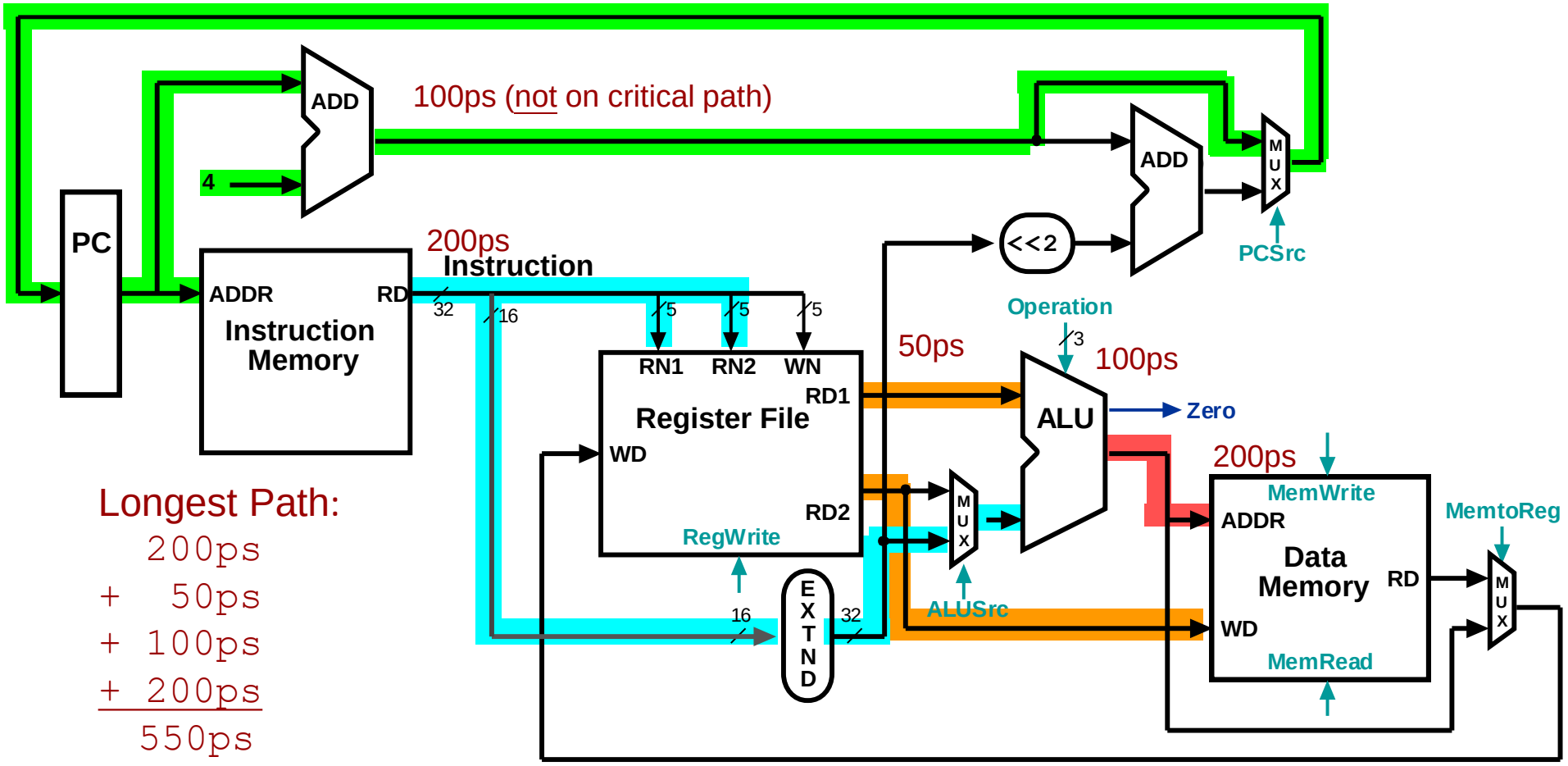
Lw指令



Longest Path:

```
  200ps
+  50ps
+ 100ps
+ 200ps
+  50ps
------
  600ps
```

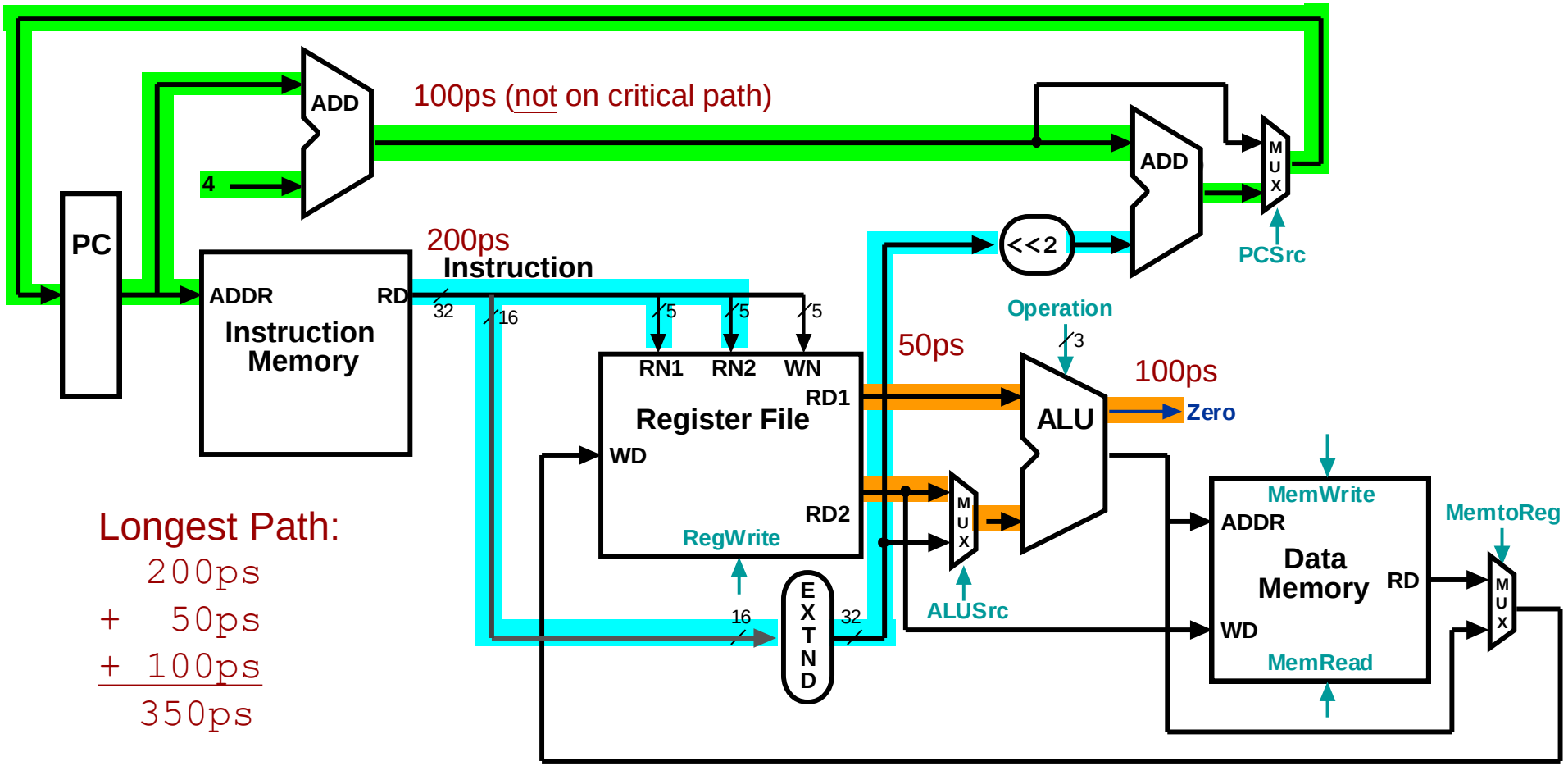
Sw指令



Longest Path:

200ps
+ 50ps
+ 100ps
+ 200ps
550ps

分支指令



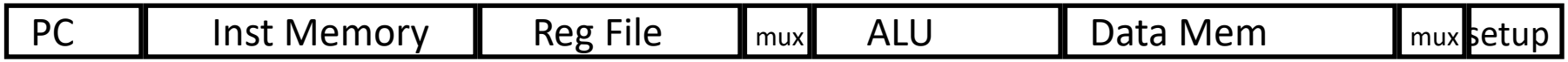
Longest Path:

200ps
+ 50ps
+ 100ps
350ps

What’s wrong with our CPI=1 processor?

Arithmetic & Logical

Load



Critical Path

Store

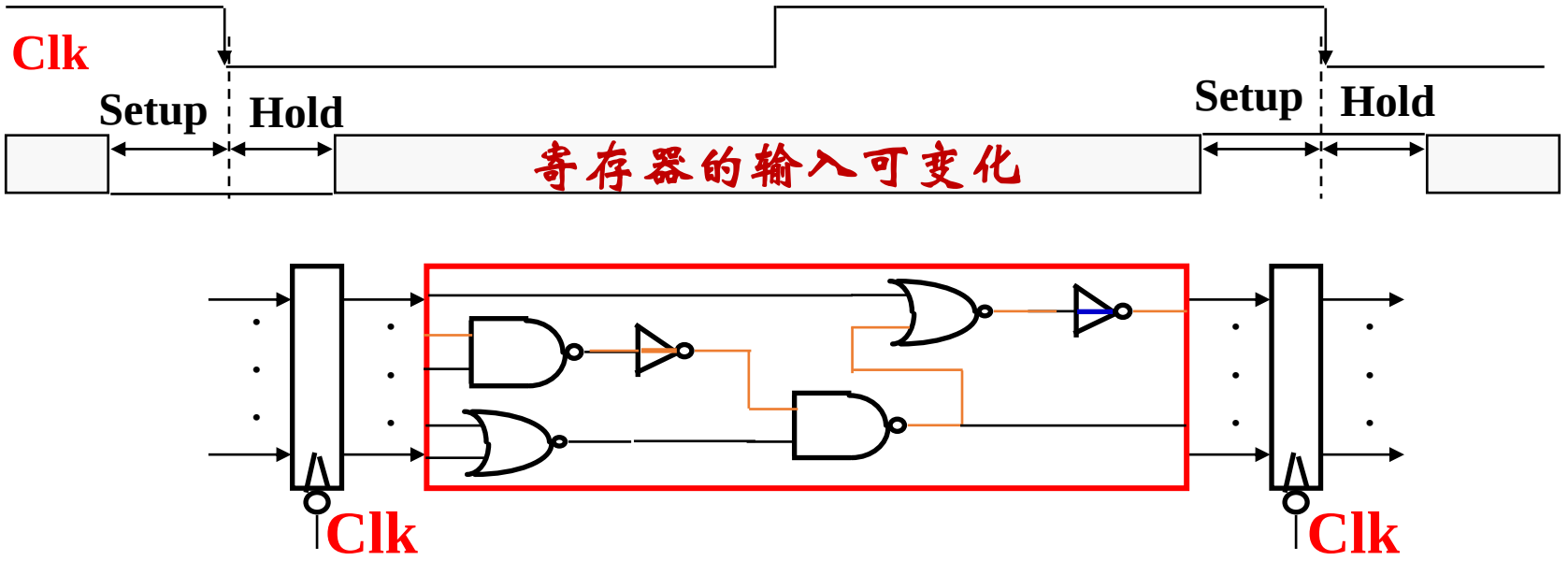
Branch

The critical path is the path to get the data for a load instruction, so we have I-Mem, Regs, Mux, ALU, D-Mem, and Mux , RegWrite on this path.

- Clock-to-Q-time:在触发时钟边沿，输出并不能立即变化，如果比较大时也要考虑进去
- 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

Reacp: 数据通路与时序控制

现代计算机的时钟周期



假定用下降沿触发(负跳变)方式

- 状态单元在下降沿写入信息，经Latch Prop (clk-to-Q)后输出有效
- Cycle Time = Hold + Longest Delay Path + Setup + Clock Skew

考虑Clk-to-Q和MUX

Clocking Methodology

- The input signal to each state element must stabilize before each rising edge.
- Critical path: Longest delay path between state elements in the circuit.
- $t_{clk} \geq t_{clk\text{-}to\text{-}q} + t_{CL} + t_{setup}$, where t_{CL} is the critical path in the combinational logic.
- If we place registers in the critical path, we can shorten the period by reducing the amount of logic between registers.

Single Cycle CPU Performance Analysis

The delays of circuit elements are given as follows:

Element	Register clk-to-q	Register Setup	MUX	ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup
Parameter	$t_{clk\text{-}to\text{-}q}$	t_{setup}	t_{mux}	t_{ALU}	$t_{MEMread}$	$t_{MEMwrite}$	t_{RFread}	$T_{RFsetup}$
Delay(ps)	30	20	25	200	250	200	150	20

- $t_{clk} \geq t_{clk\text{-}to\text{-}q} + t_{CL} + t_{setup}$, where t_{CL} is the critical path in the combinational logic.

1.Answer： tclk >= tPC, clk-to-q + tIMEMread + tRFread + tmux+ tALU + tDMEMread + tmux + tRFsetup= 30 + 250 + 150 + 200 + 250 + 50 + 20 =950 ps fclk = 1/tclk <= 1/ (950 ps) = 1.05 GHz

Single Cycle Implementation Cycle Time

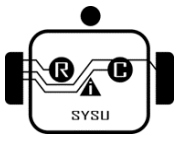
- Unfortunately, though simple, the single cycle approach is not used because it is very slow
- Clock cycle must have the same length for every instruction

- What is the longest (slowest) path (slowest instruction)?

- 单周期设计中时钟周期对所有的指令等长，因此要由计算机中可能的最长路径决定：

  ——取指令

违反了设计原则 —— 加速完成常用操作



Instruction Critical Paths

- **Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times) except:**

  Instruction and Data Memory (4 ns)

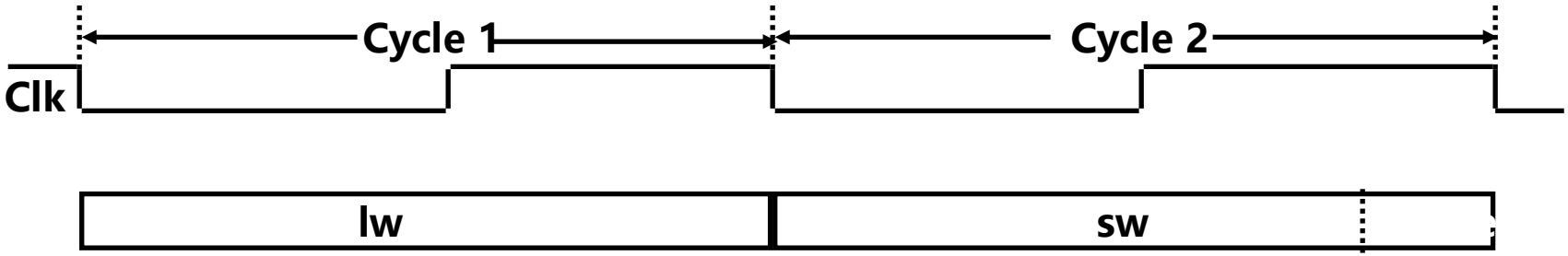
  ALU and adders (2 ns)

  Register File access (reads or writes) (1 ns)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	4	1	2		1	8
load	4	1	2	4	1	12
store	4	1	2	4		11
beq	4	1	2			7
jump	4					4

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instr
  - especially problematic for more complex instructions like floating point multiply



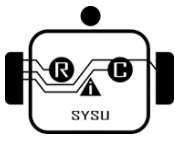
- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- It is simple and easy to understand

单周期CPU的性能

- 各类指令的数据路径长度
  - R型指令　200 + 50 + 100+0+50　400ps
  - **Load word　200 + 50 + 100 + 200 + 50　600ps**
  - Store word　200 + 50 + 100 + 200　550ps
  - 分支　200 + 50 + 100　350ps
  - 转移　200　200ps
- 性能受最慢指令的限制



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：

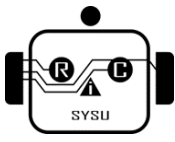
- 存储单元：200ps
- ALU和加法器：100ps
- 寄存器堆（读/写）：50ps

假设程序中各类指令占比：25%取数、10%存数、45%ALU、15%分支、5%跳转

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，也即为可变的（实际不可行，只是为了比较）



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				

各类指令要求的时间长度为：

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

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不等

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。

单周期计算机的性能

- 单周期计算机的性能练习：

  假设各主要功能单元的操作时间为：

  - 读存储器：10ns，写存储器：5ns
  - ALU和加法器：10ns
  - 寄存器堆（读/写）：5ns

  而MUX、控制单元、PC、扩展器和传输线路没有延迟，若各类指令的执行次数占总数的比例为：20%取数、10%存数、50%ALU、15%分支、5%跳转，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，但时钟周期是可以根据指令类型动态变化的。

单周期计算机的性能

解：方式(1)：时钟周期由最长指令来决定，应是load指令，为 40ns——Load指令的执行如下：

取指令10ns,读寄存器堆5ns，ALU计算地址10ns,读存储器10ns,写寄存器堆5ns,总的时间是40 ns。

方式(2)：时钟周期取各条指令所需时间，根据各类指令的频度，计算出平均时钟周期为：

30×50% + 40×20% + 35×10% +25×15% +10 ×5% = 30.75ns

加速比=可变周期处理机的性能/固定周期处理机的性能

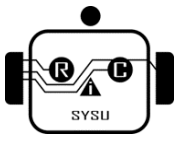
= (I ×1×40) / (I ×1 ×30.75) = 1.3

可变周期处理机的性能是固定周期处理机性能的1.3倍！

MIPS Control

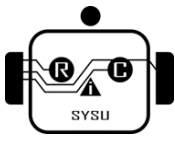
The control unitt can be implemented using a ROM

Insttructtion	RESET	IRQ	Z	N	V	C	PCSEL	SEXT	WASEL	WDSEL	ALUFN Sub	ALUFN Bool	ALUFN Shftt	ALUFN Matth	WR	WERF	ASEL	BSEL
---	1	X	X	X	X	X	4	0	0	0	0	00	0	0	0	0	0	0
---	0	1	X	X	X	X	6	0	3	0	0	00	0	0	0	0	0	0
add	0	0	X	X	0	X	0	0	0	1	0	00	0	1	0	1	0	0
sll																		
andi																		
lw	0	0	X	X	X	X	0	1	1	2	0	XX	0	1	0	1	0	1
sw	0	0	X	X	X	X	0	1	X	X	0	XX	0	1	1	0	0	1
beq	0	0	0	X	X	X	0	1	X	X	1	XX	0	1	0	0	0	0
beq	0	0	1	X	X	X	1	1	X	X	1	XX	0	1	0	0	0	0
lui																		



其它可实现的方式

- 多周期处理器
  - 缩短指令周期
  - 一条指令多个周期
  - 不同类型指令所需的周期数不同
  - 硬件代价小
- 流水线处理
  - 指令重叠执行
  - 尽可缩短时钟周期数和CPI
  - 硬件代价大，但性能更好



联系方式

- Acknowledgements:
- This slides contains materials from following lectures:
  - Computer Architecture (ETH，NUDT，USTC)

- Research Area：
  - 计算机视觉与机器人应用计算加速，
  - 人工智能和深度学习芯片及智能计算机

- Contact：
  - 中山大学计算机学院
  - 管理学院D101（图书馆右侧）
  - 机器人与智能计算实验室
  - cheng83@mail.sysu.edu.cn