Lecture 11, 12 : RISC-V Instruction Format

History's Big Idea: Strored Program Computer

- Instructions are represented as bits patterns

Stored Program Computer: Implications

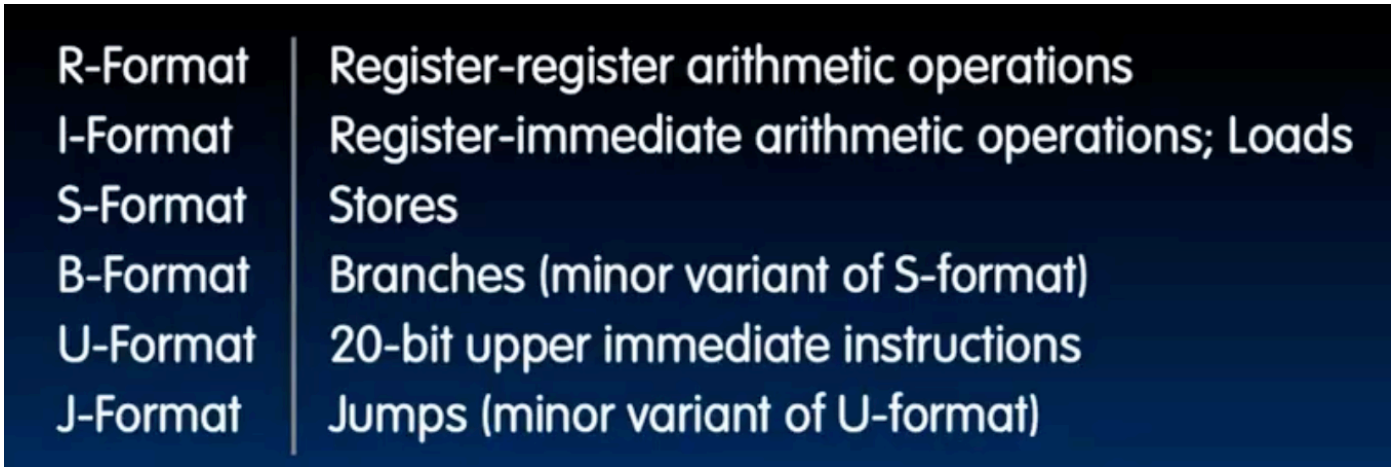
1. Everythiing (instructions, data words) has a memory address
    - The *Program Counter(PC)* register keeps address of *next* instruction to execute
2. Programs are distributed in *binary* form, as assembled machine code
    - 程序与特定的指令集绑定
    - 很多指令集都是向后兼容(*backwards-compatible*)并且随时间进化

RISC-V的指令都是32位字

指令格式

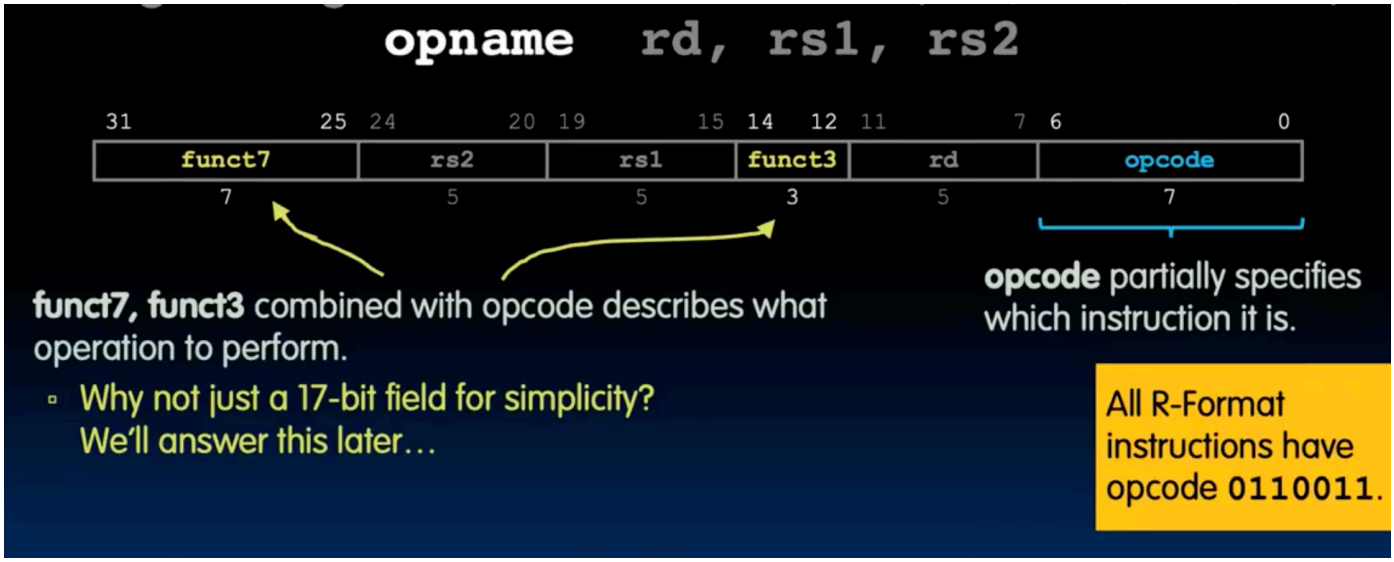
Fields -- 字段

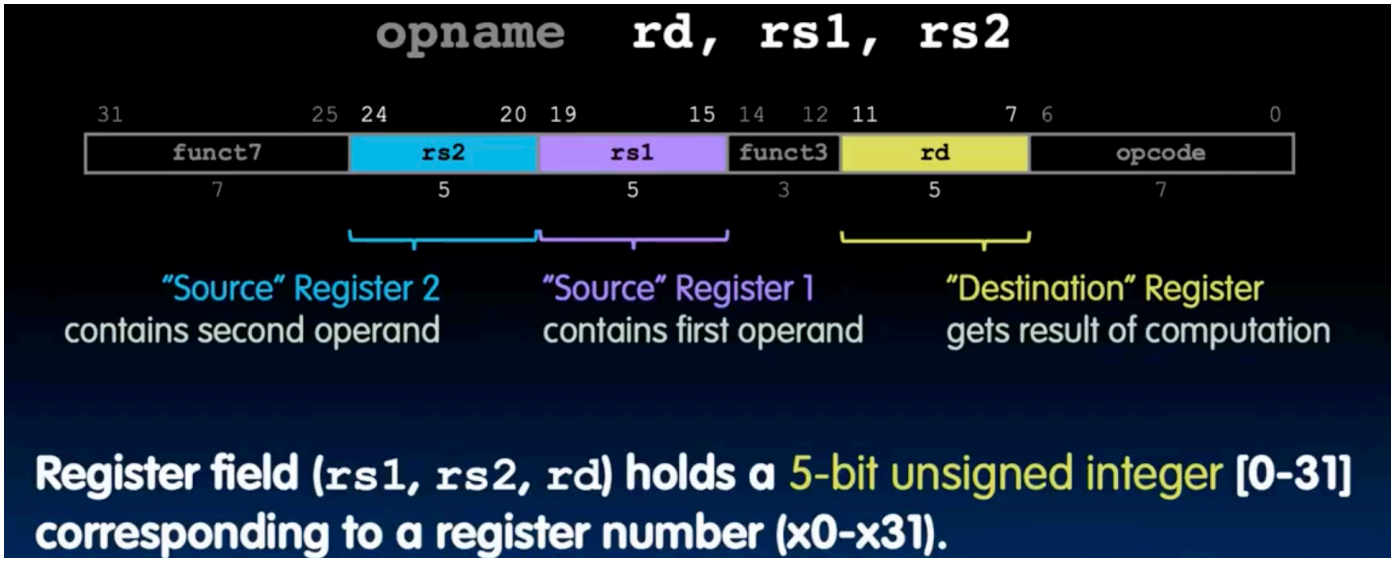
RISC-V有四种指令类型



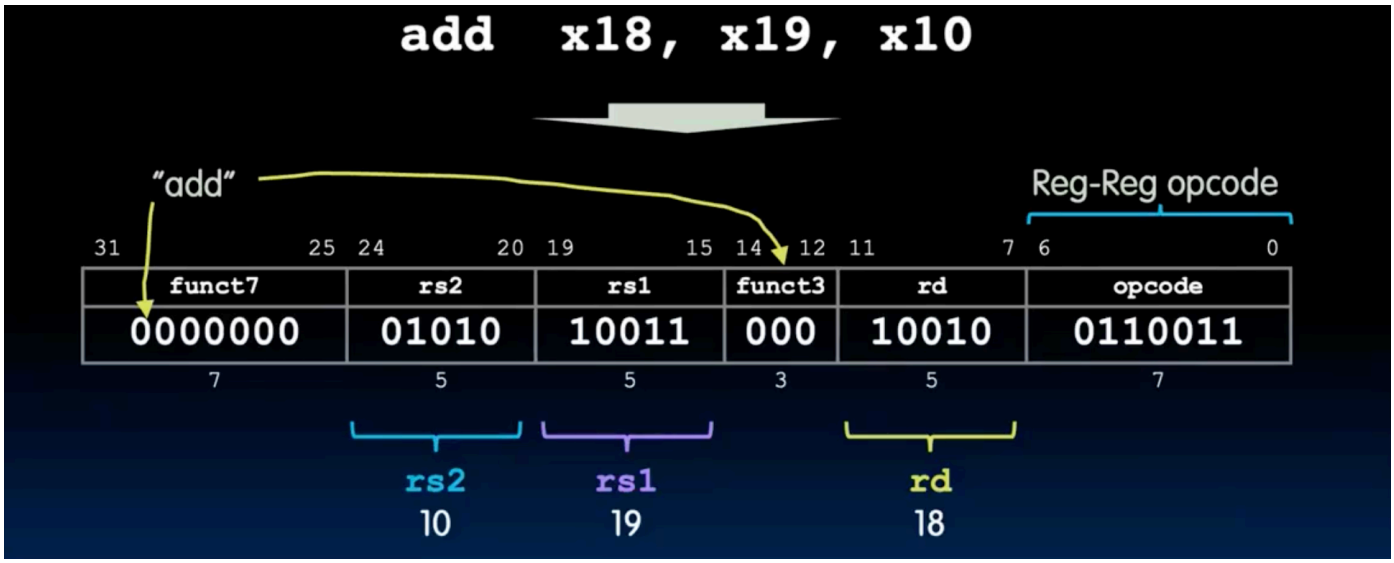
R-Format Layout

Register-Register Arthmetic Instructions(`add xor sll, etc.`)

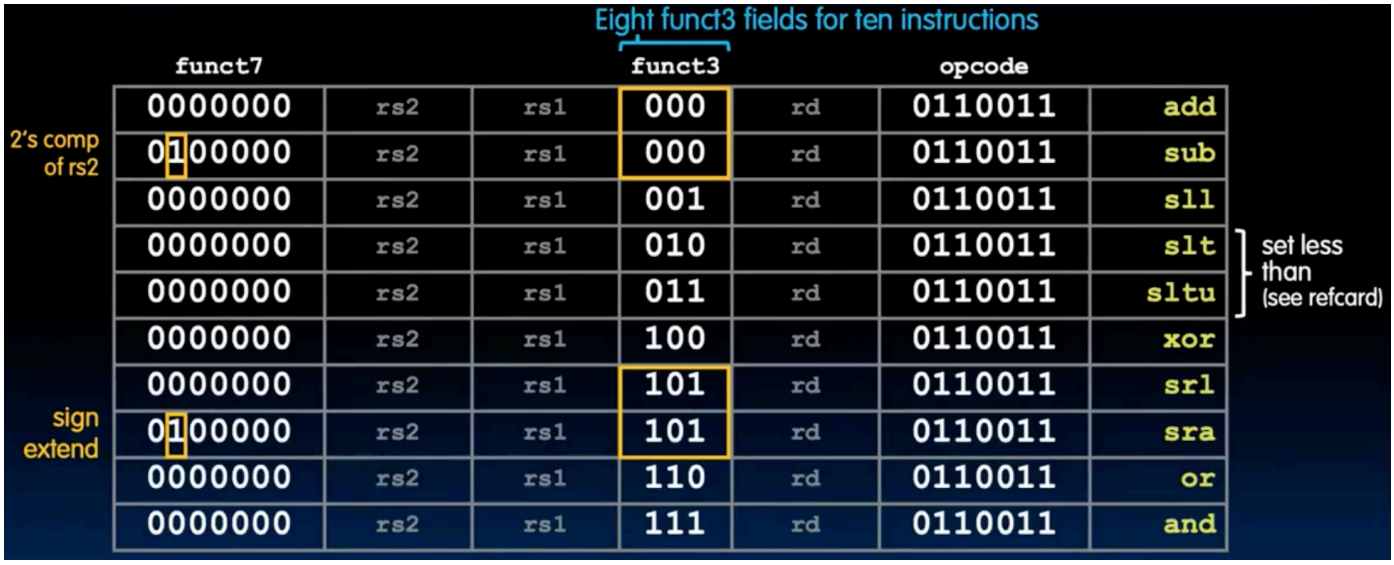




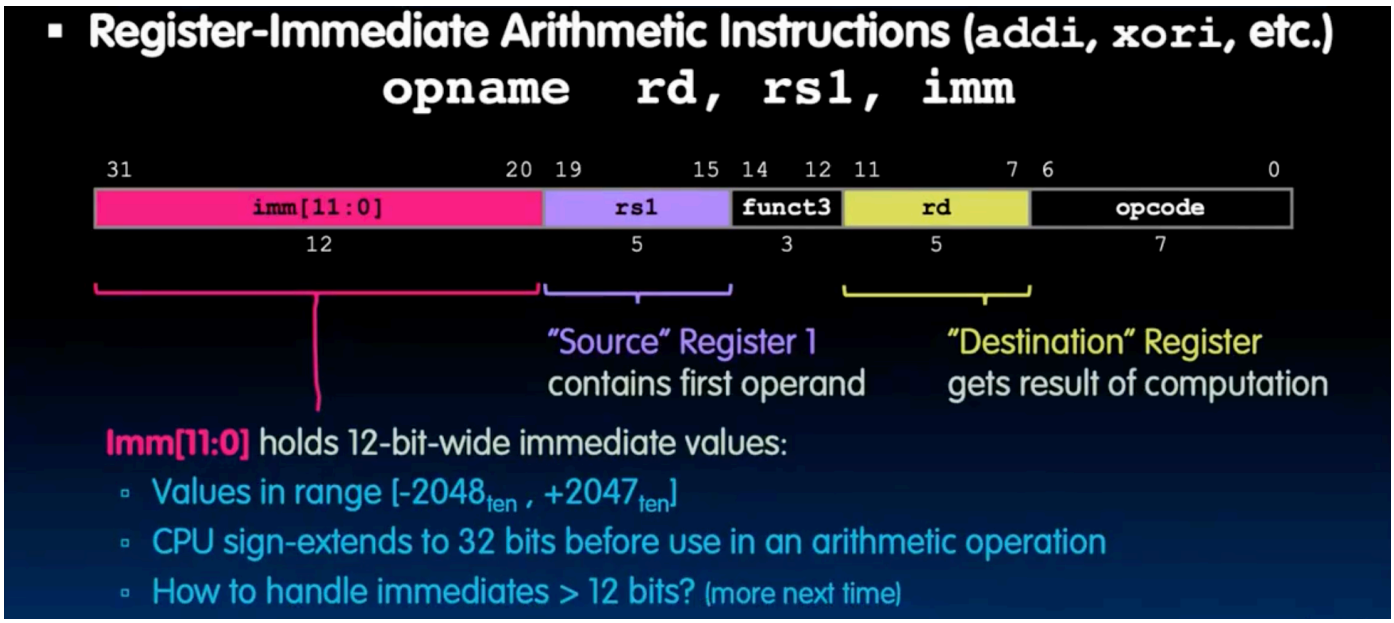
举个例子



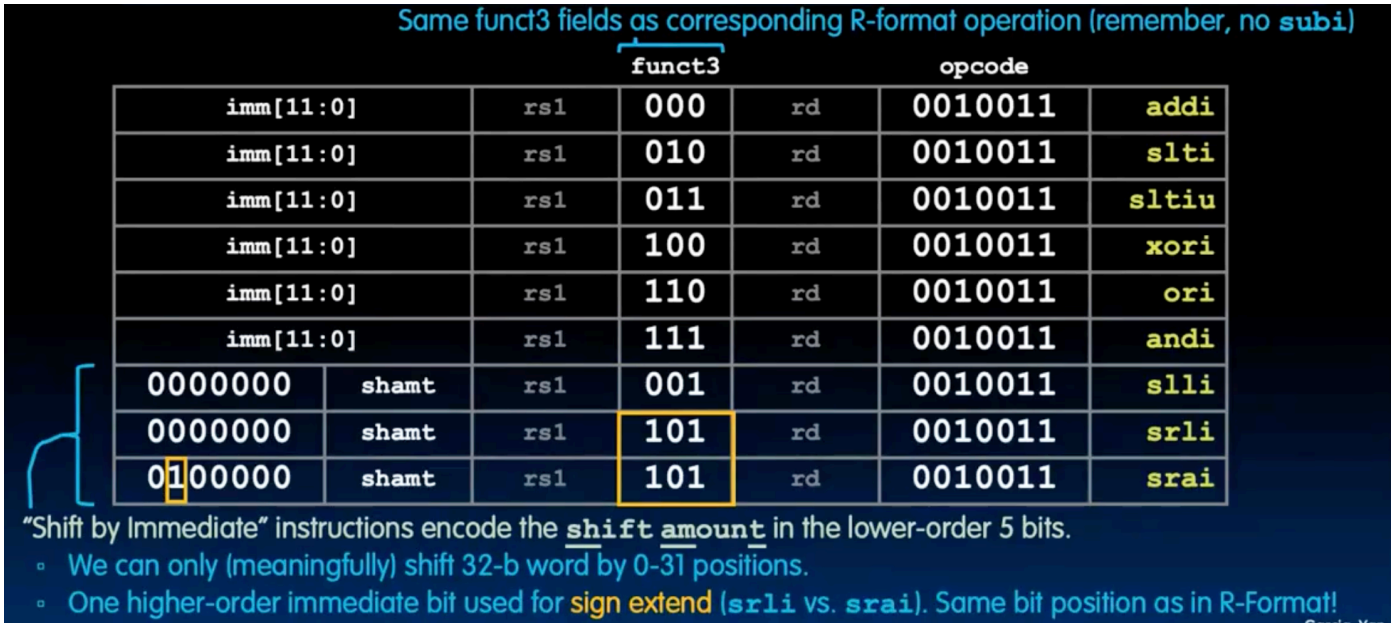
全部十个R型指令



I-Format

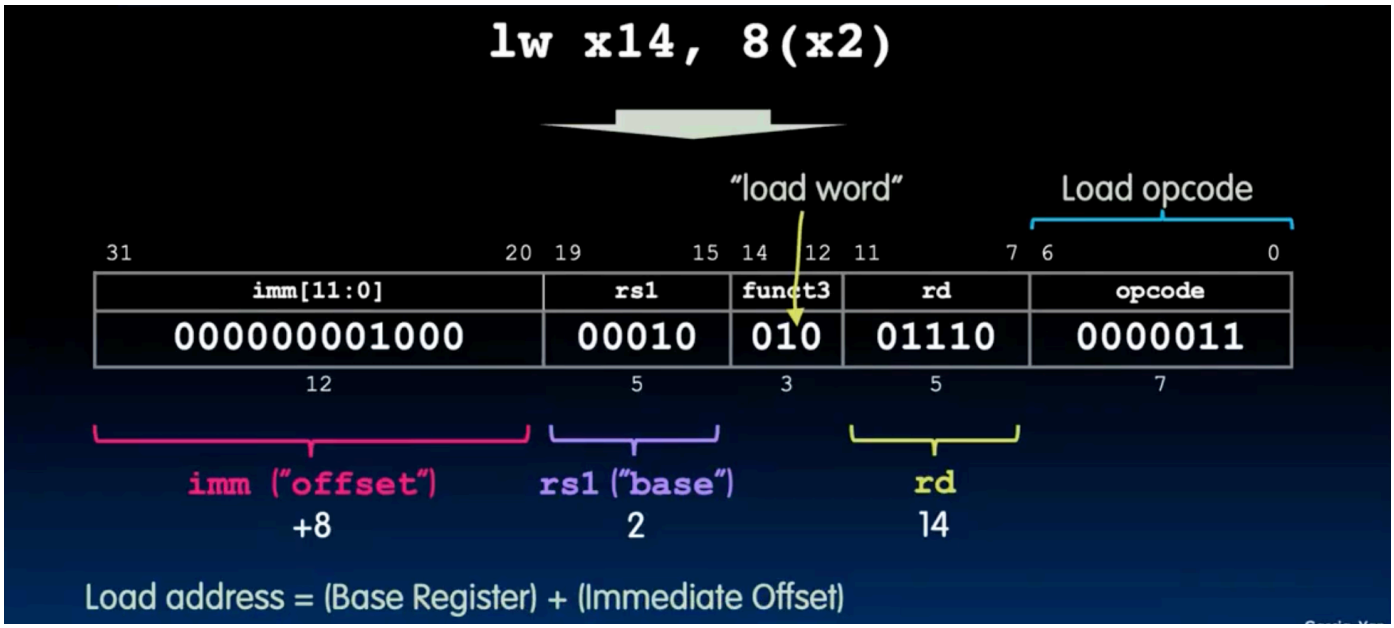


所有的9个I型运算指令

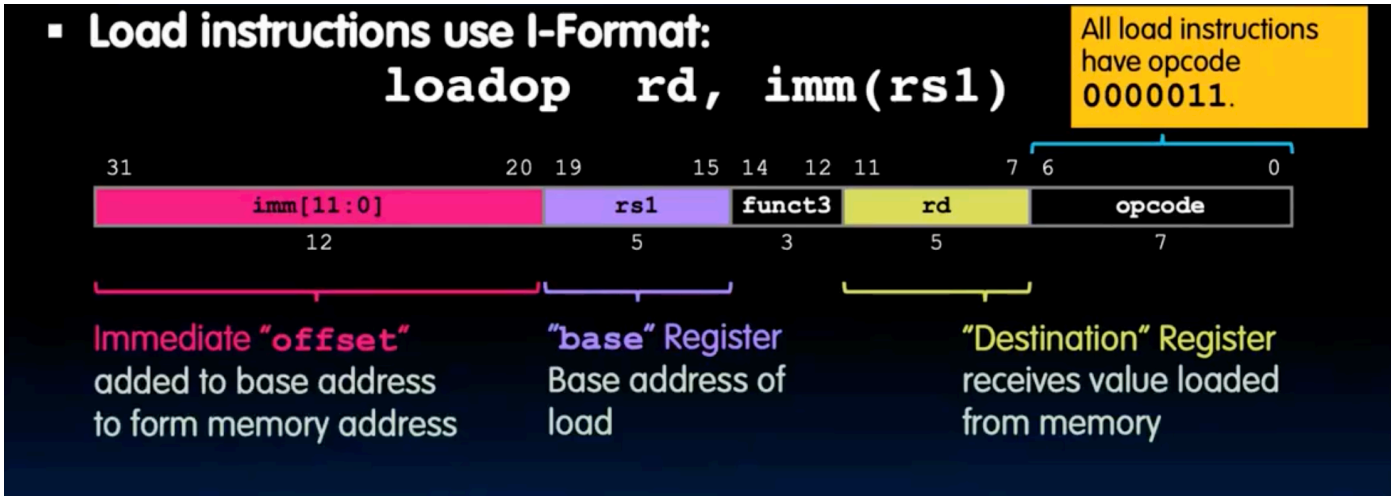


I型指令也可以用于加载指令

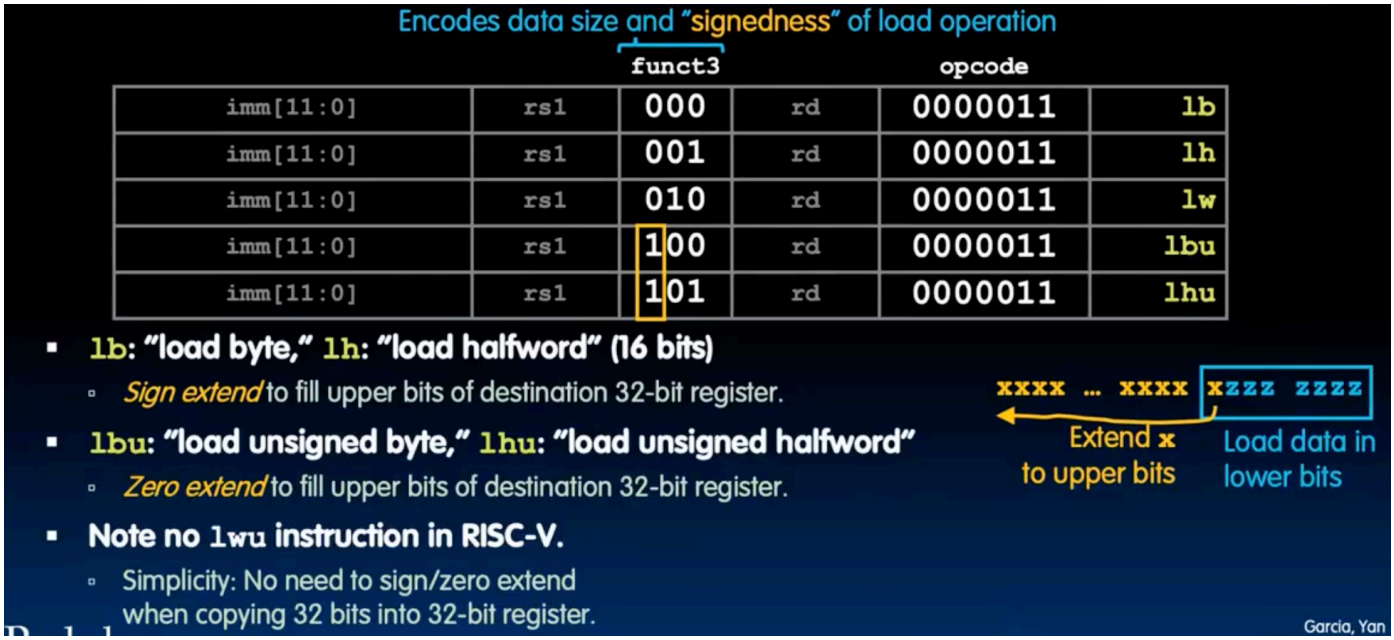
举个例子



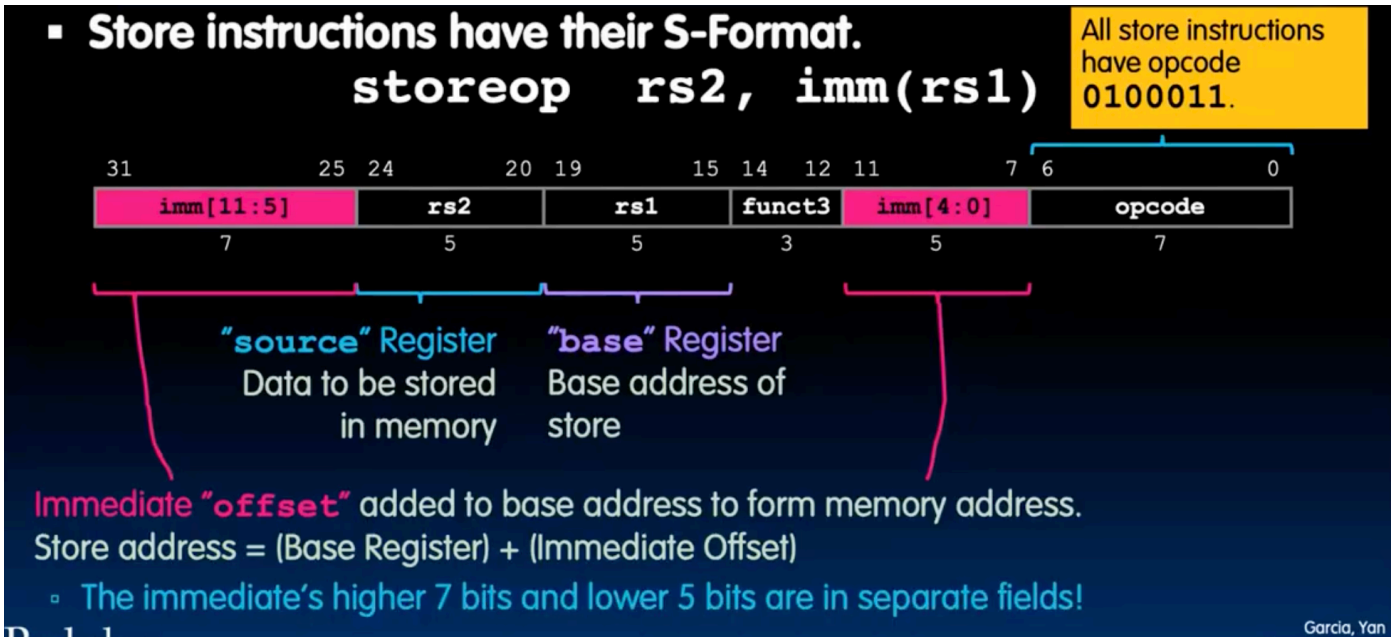
Load型格式



五个加载指令



S型指令

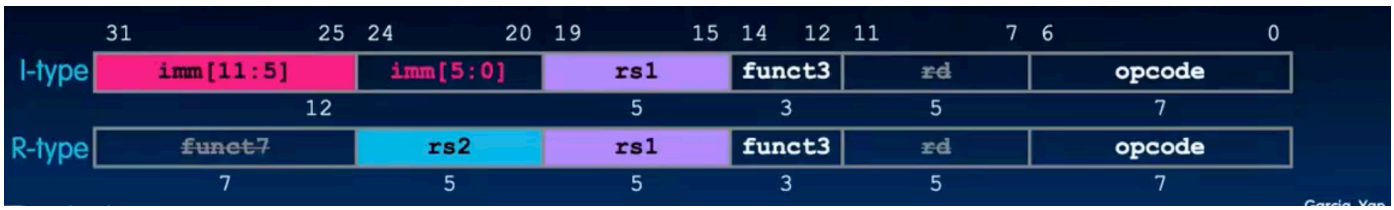


指令最重要的部分是读寄存器和写寄存器的位置保存一致

immediate对硬件来说相对不那么重要

所以我们将immediate分为两部分

以便我们有较为近似的指令格式



更新the Program Counter(PC)

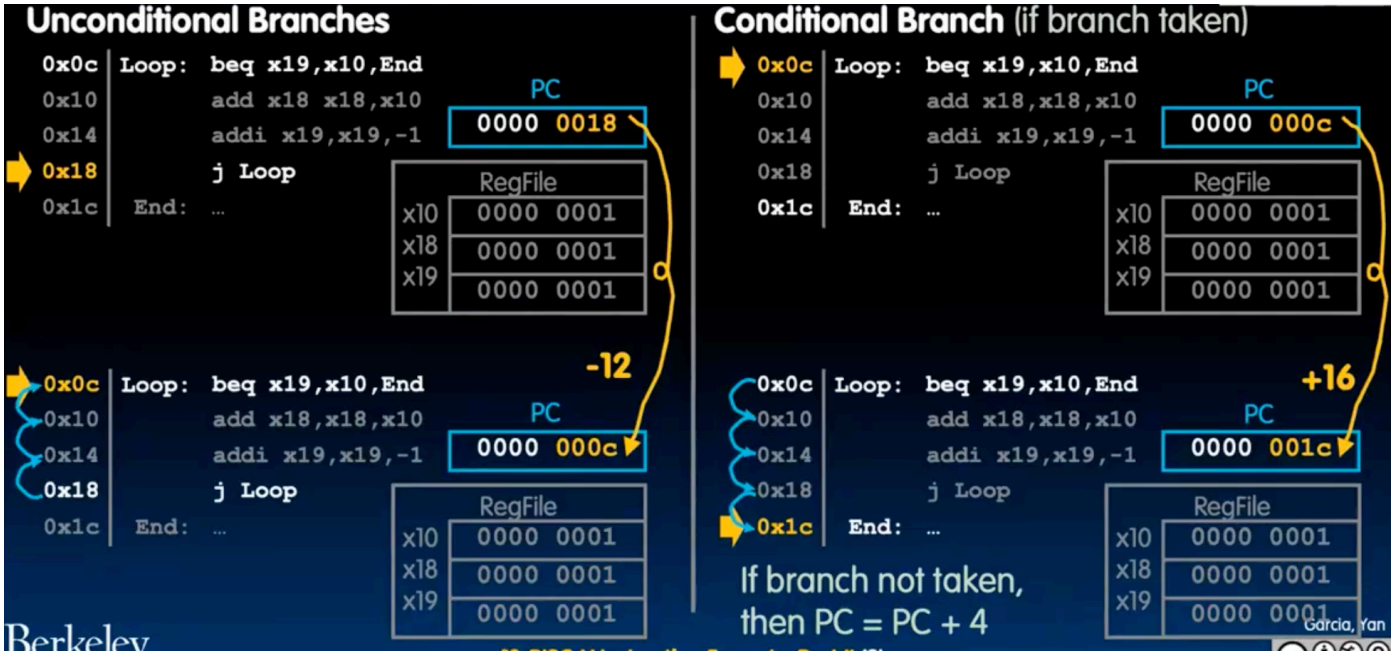
PC是一种在处理器内部的寄存器，并且与32个寄存器分开，它容纳了被执行指令的字节地址

PC（程序计数器） 存储的是 **当前正在执行的指令的地址**

一般来说每多一个指令，PC就会+4，因为32位就是4字节

对于分支指令来说

PC会直接更改到相应标签的地址



PC-Relative Addressing PC相对寻址

> **不是直接去找某个固定地址，而是“以当前 PC 为起点”加上一个偏移量，去找目标地址。**

B-Format

分支指令

使用PC相对寻址来编码labels标签

RISC-V尽管都拥有32位的指令，对于特定的架构，他也支持16位的指令

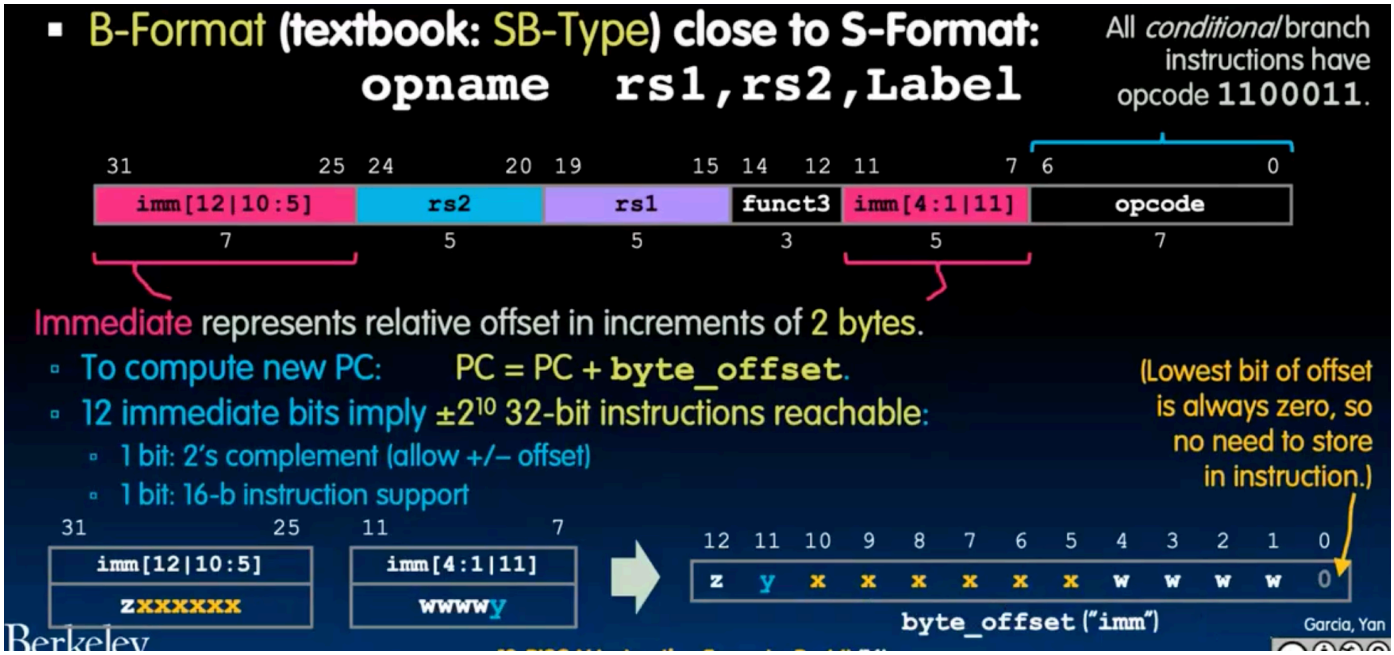
所以

> RISC-V 中，**分支指令的偏移量以 2 字节为单位**（为了兼容压缩指令），即便你用的全是 32-bit 指令，这也意味着：**偏移范围减半，只能跳到 $\pm 2^{10}$ 条指令**！
>
> > **分支偏移量 = 从当前指令（PC）起跳，要跳到目标位置的距离（单位是字节 or 指令）**

Branch immediate range [-2048, +2047]

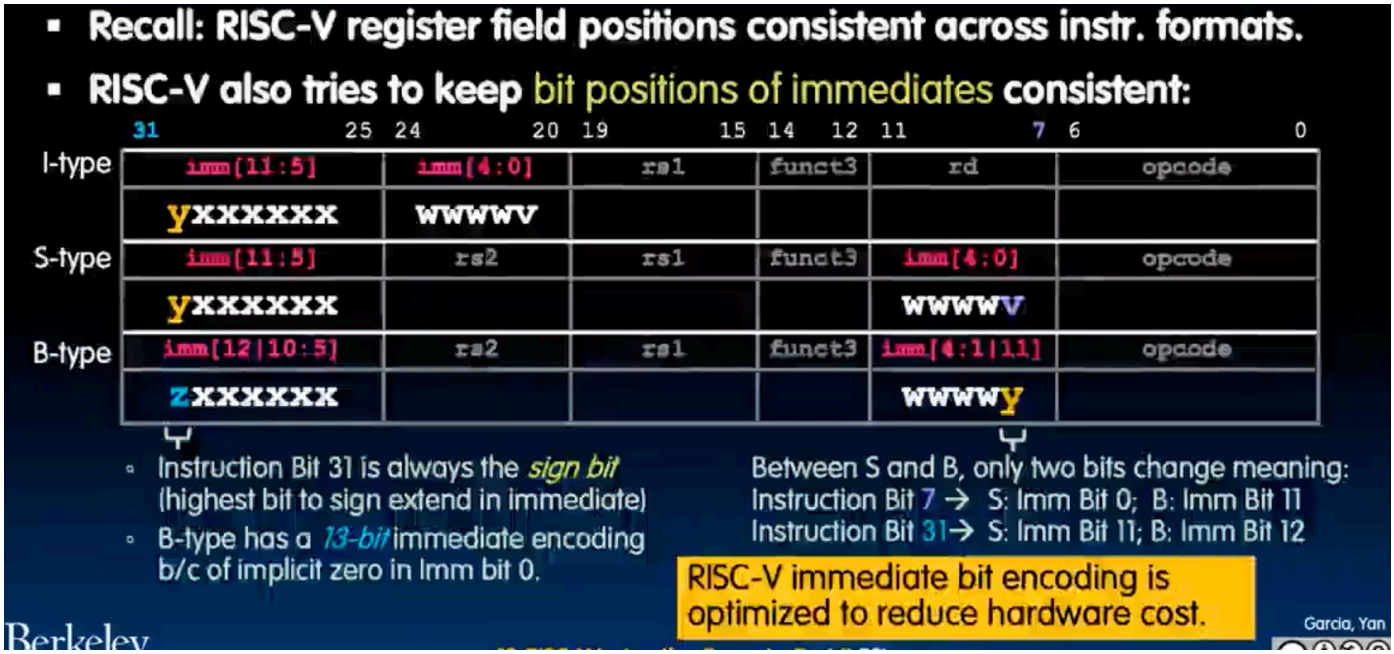
represents PC relative offsets [-4096, + 4094] in 2-byte increments

B型指令布局

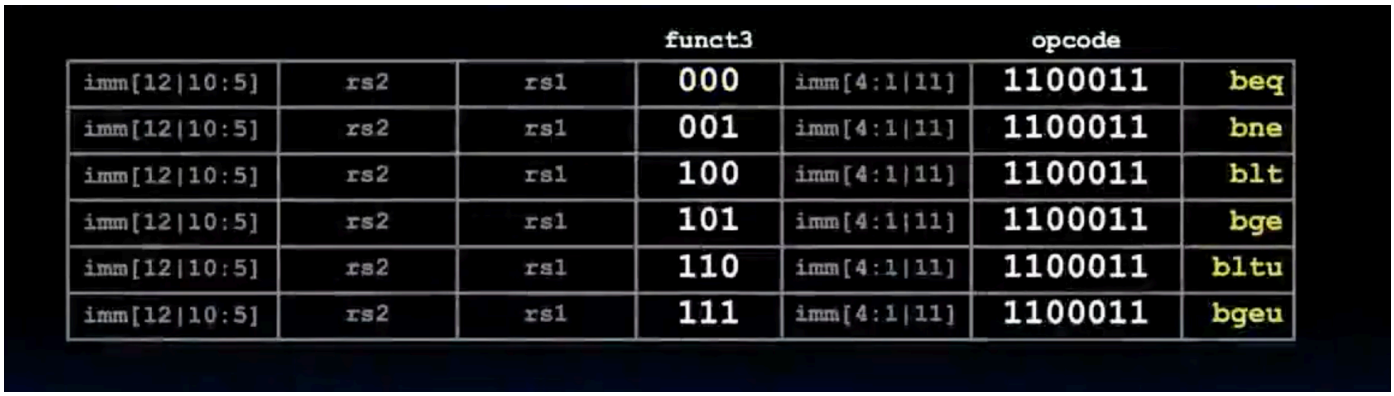


imme错位存储原因

> B-type 的立即数之所以“错位存储”，是为了和 S-type 共用字段，达到**最大限度复用硬件逻辑**的目的。这是一种**工程上的优化权衡**，让硬件设计更简单，但增加了一点指令解码时的复杂性。



所有的6个B型指令

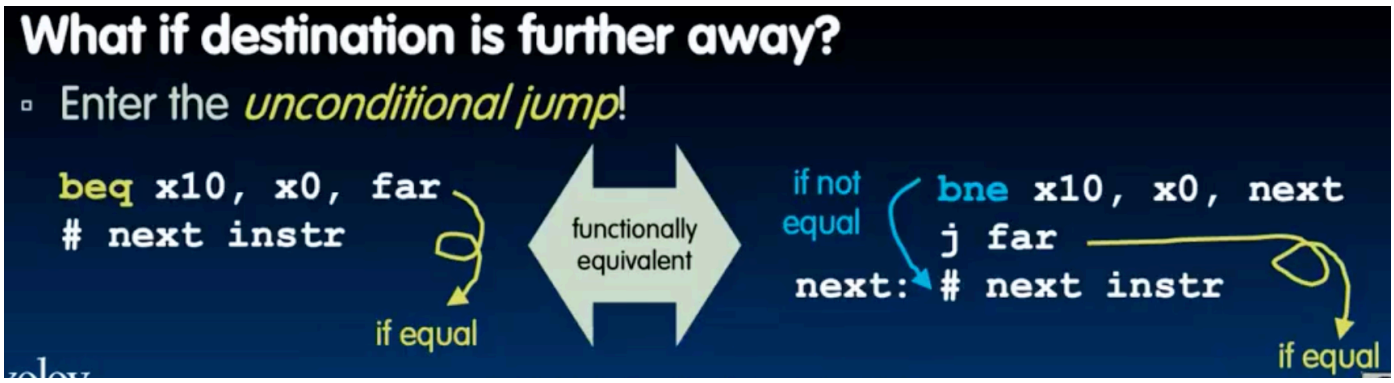


			funct3		opcode	
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	bgeu

条件分支和无条件分支

由于条件分支跳转的范围较小

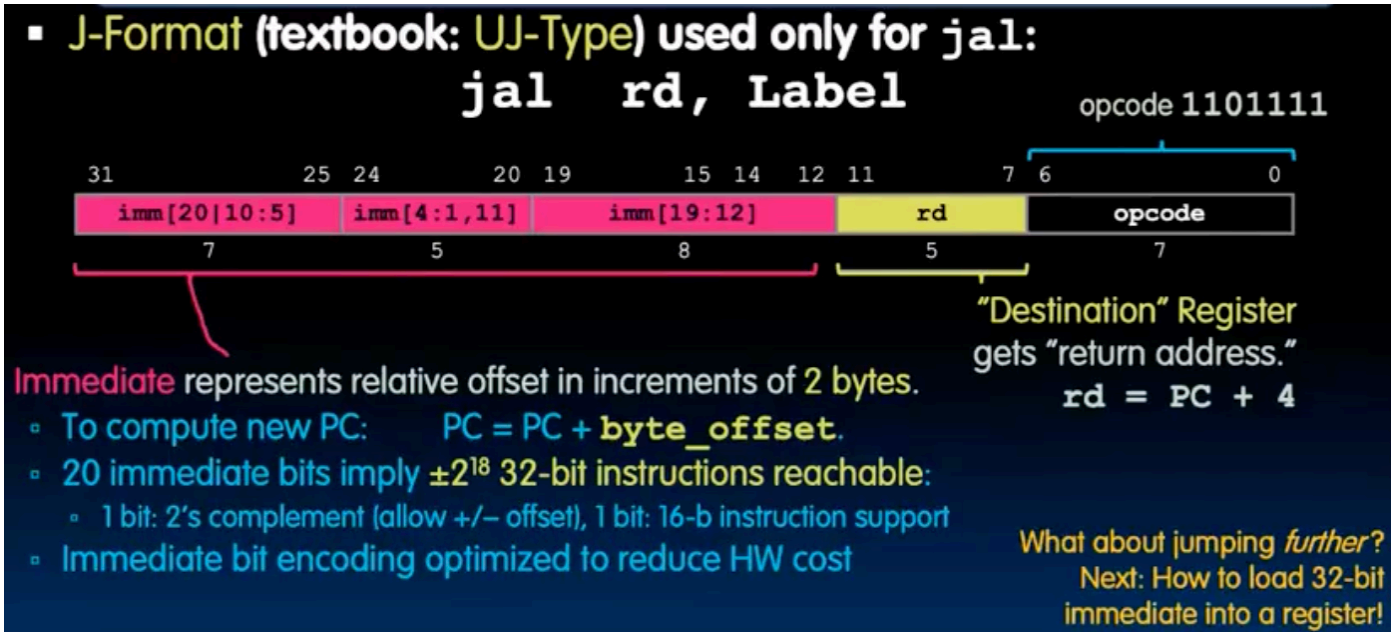
所以想要跳转到很远的地方要用无条件跳转



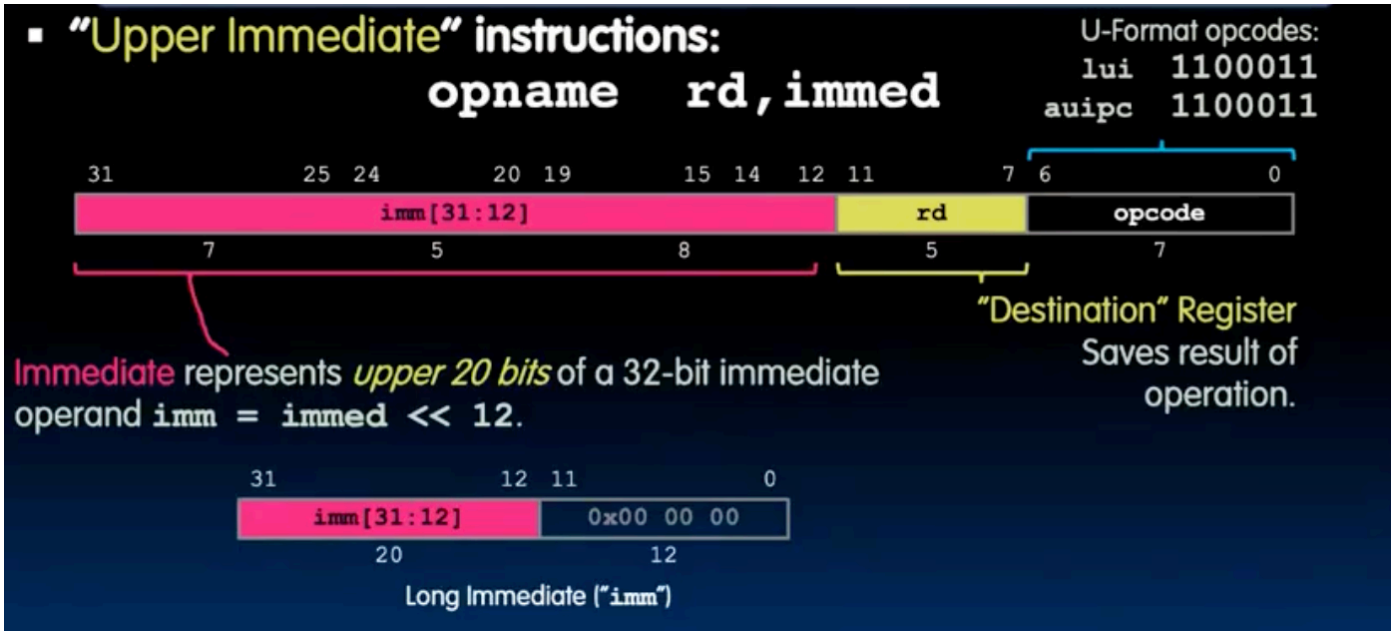
J-Format (UJ-Format)

只用于`jal`

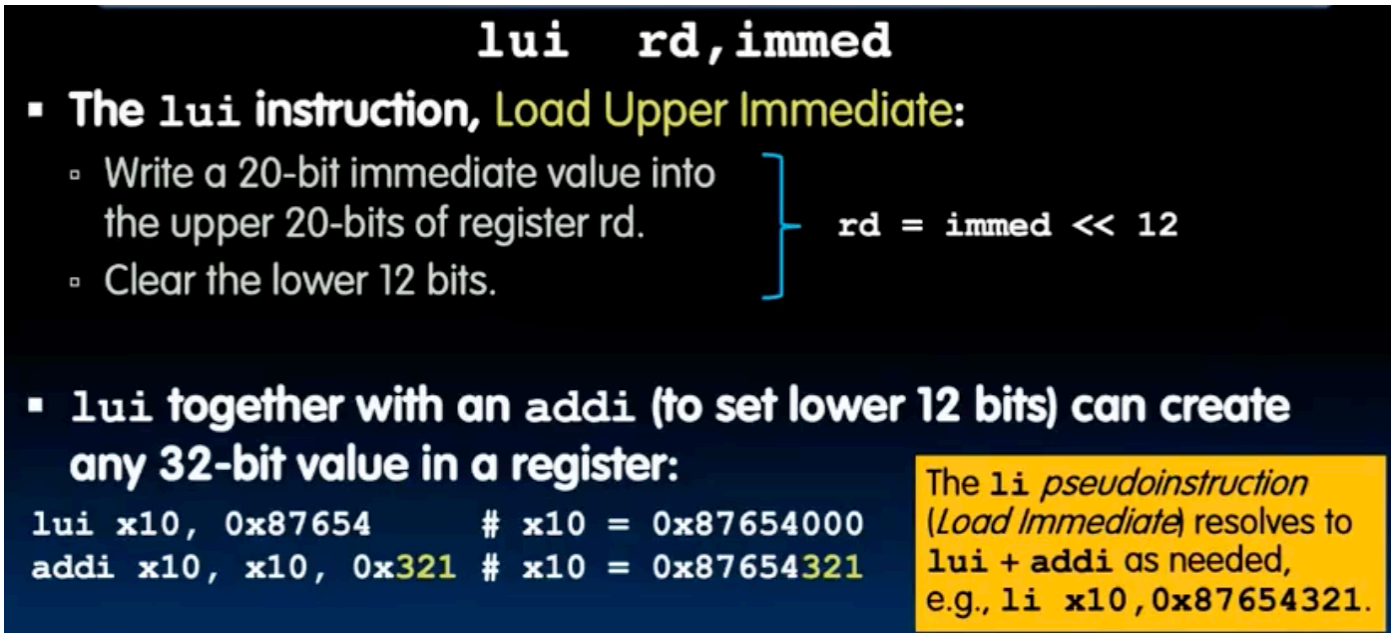
适用于处理无条件的跳转



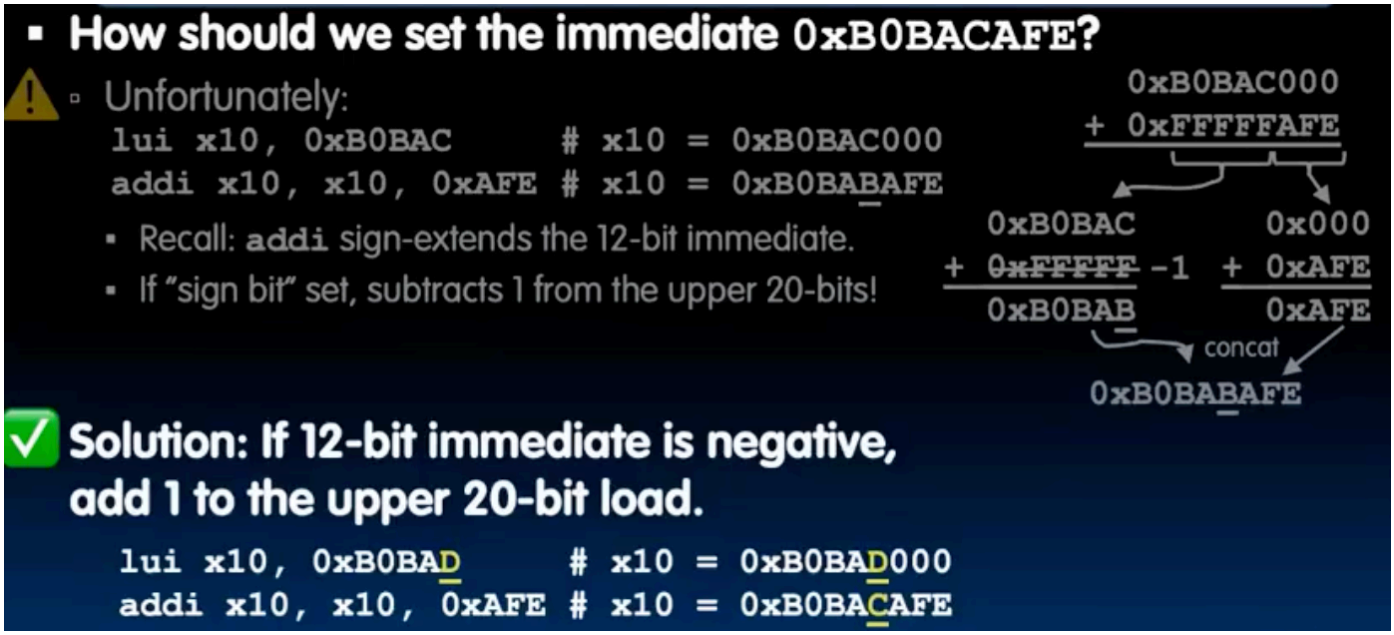
U-Format



`lui`与`addi`配合可以加载32位的imme



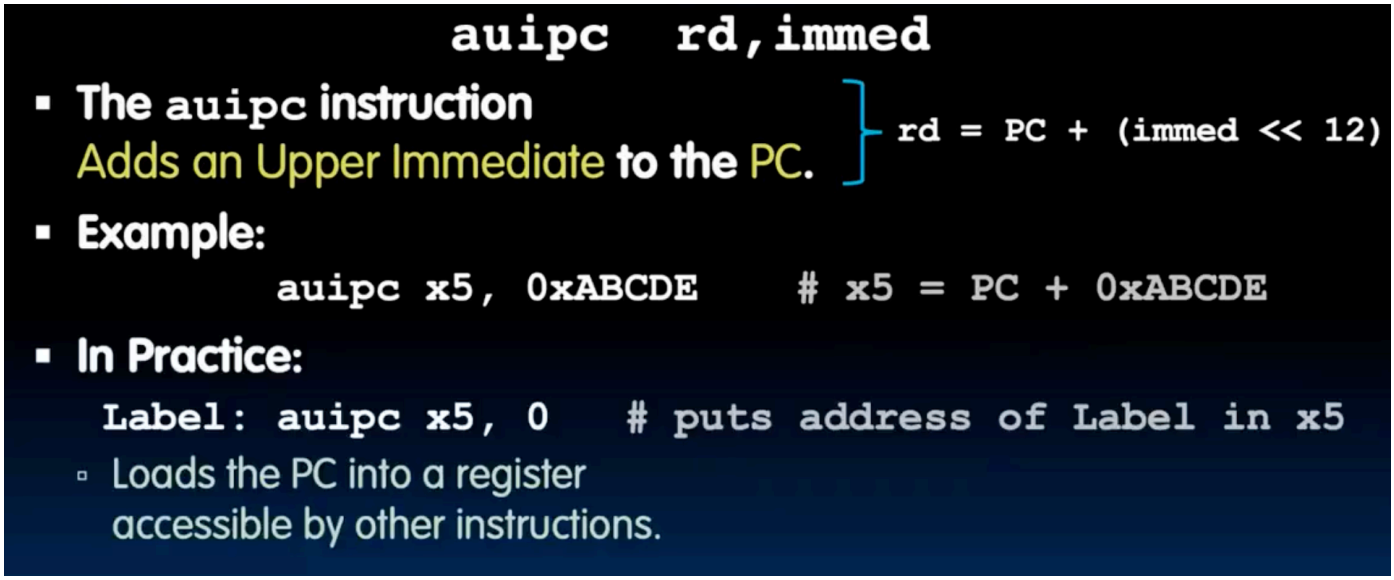
注意：`addi`会扩展符号位



也可以使用伪指令`li`自动地处理边缘情况

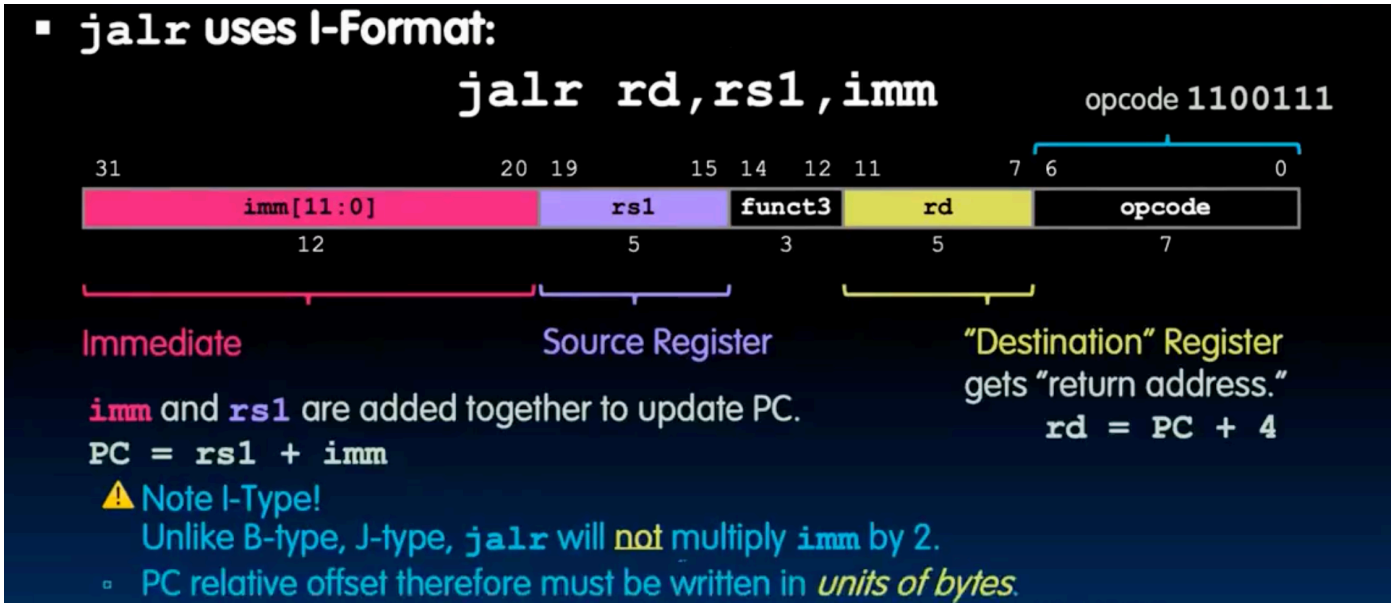
`auipc` loads the PC into the Register File

最常见的用法是直接复制当前的PC值



`auipc` is most often used together with `jalr` to do PC-relative addressing with super large offsets.

`jalr`: I-Format

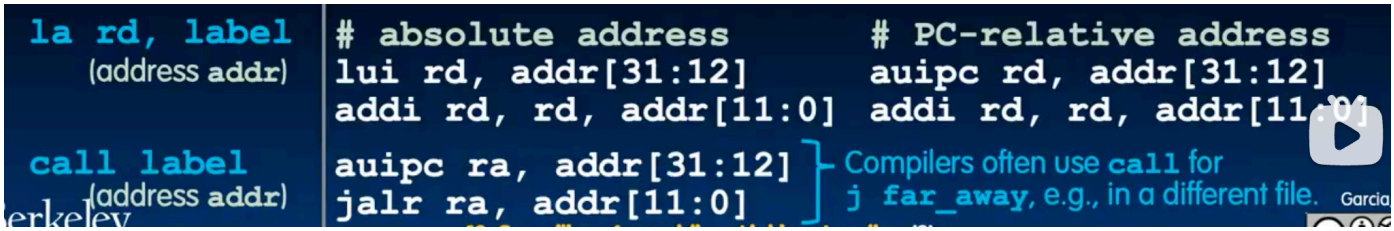


Unlike `jal`(relative to PC), `jalr` addresses are relative to `rs1`, which is modifiable by arithmetic instructions. We can do bigger jumps!

Lecture 13 : Running A Program :CALL (Compiler, Assembler, Linker, Loader)

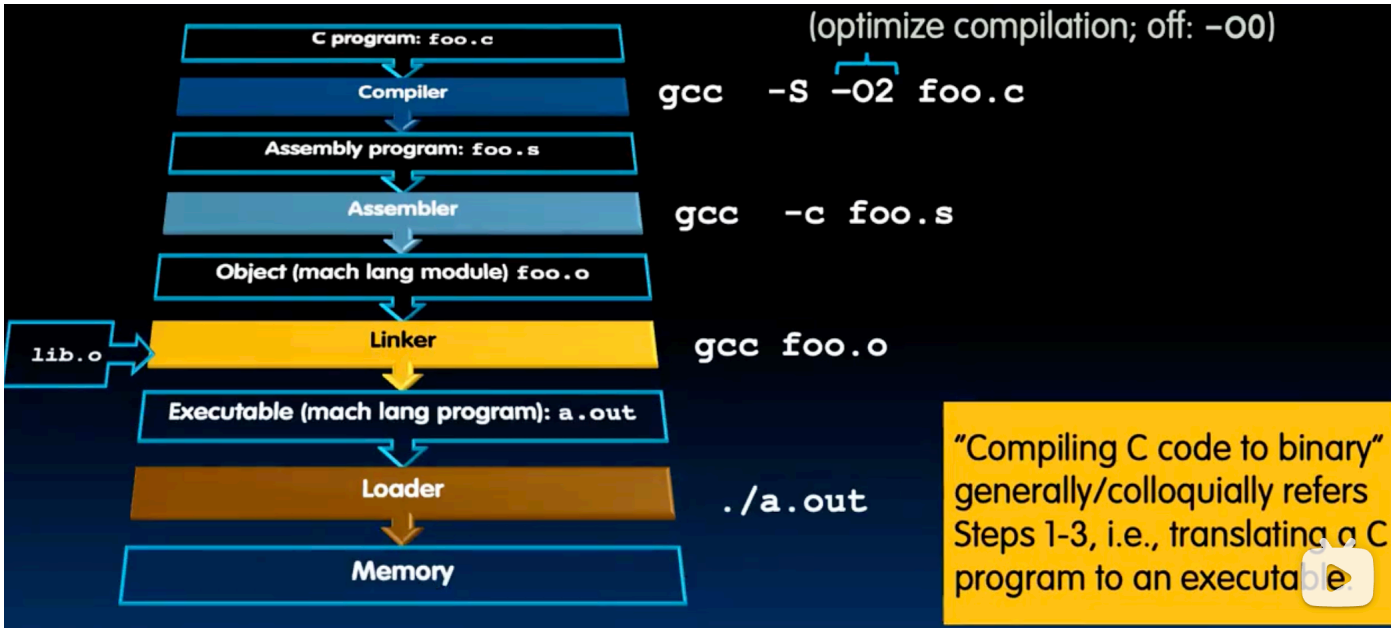
介绍两个新的伪指令：

`la (load address)` 与`call label`



Translating and Running A Program: Call

- Translator:将一种源语言转化为其他语言
  - 翻译/编译 成低级语言意味着更加高效
- Interpreter: 直接执行源语言



将C语言转化为二进制通常说的是前三步

Complier

Note : 将C语言转化为汇编，输出的结构可能会包含伪指令

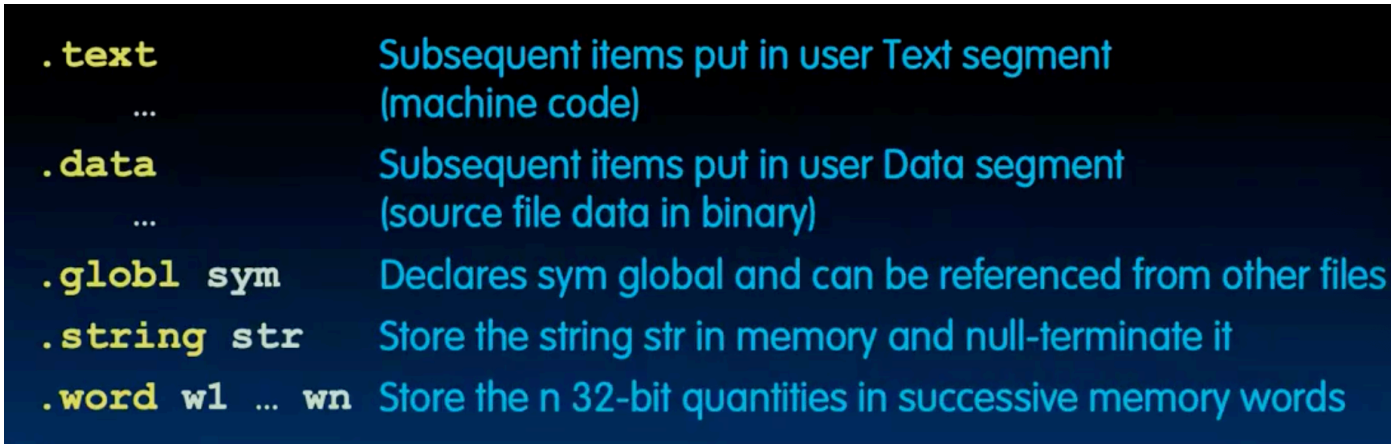
Assembler

将汇编语言转化为机器码

它会将伪指令替换为真正的汇编语言并转化为机器码

Directives

- Often generated by the complier
- do not produce machine instructions. Rather, they inform how to build different parts of the *object file*



Object File Format

1. Object File Header : size and position of other of the object file
2. Text Segment: machine code

3. Data Segment: binary representation of static data in the source file
4. Symbol Table: List of file's labels, static data that can be referenced by other programs
5. Relocation Information (重定位信息) :lines of code to fix later (by Linker)
6. Debugging Information

Producing Machine Code

伪指令将被替换为真实的指令，然后所有的PC相对地址可以被计算，我们称之为 **Position-Independent Code (PIC)**

Compute PC-Relative Addresses:两次浏览代码

为什么需要两次?

因为我们在进行分支跳跃的时候，标签可能在之后的地址，此时我们还没有访问到，无法进行替换

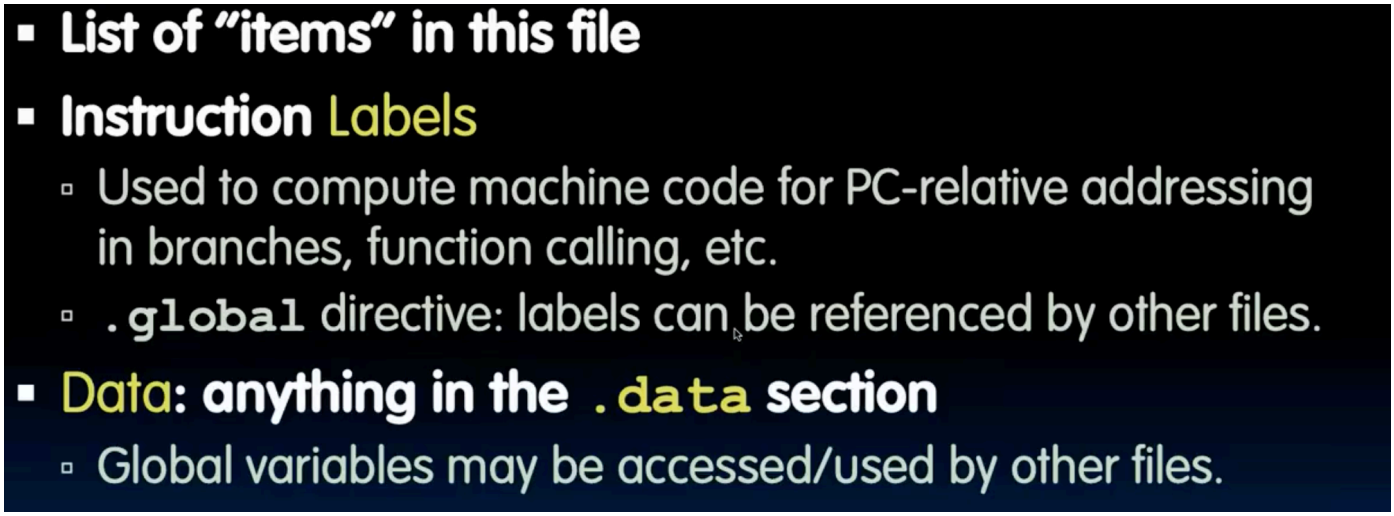
进行两次

- 第一次我们记下标签的位置(store in *symbol table*符号表)
- 第二次使用标签位置生成机器码

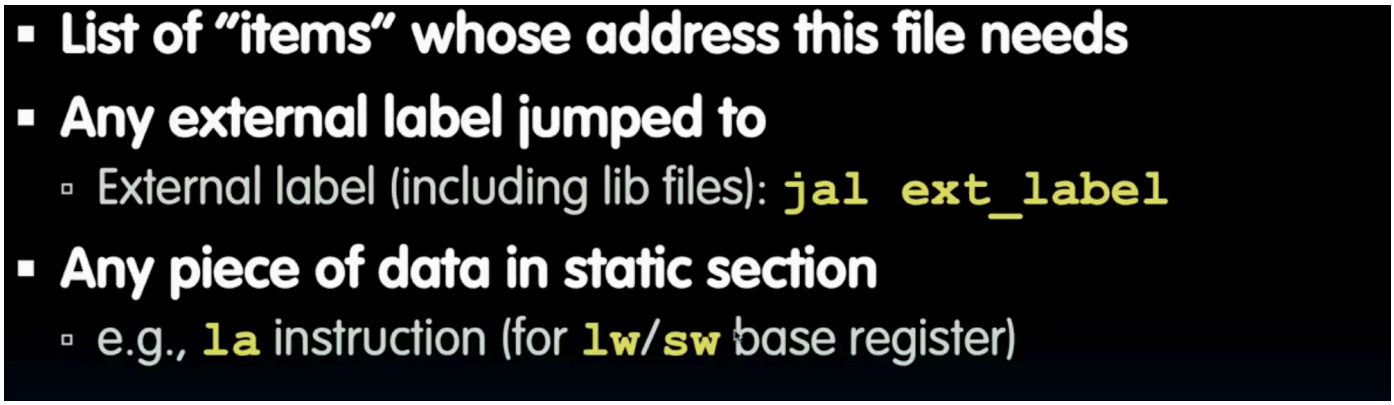
What about Other References?

- Reference to other files?
  - like `strlen` form C library
- Reference to static data
- These can't be determined yet, so the Assembler jots them down in two tables :
  - Relocation Information
  - Symbol Table

符号表



重定位信息

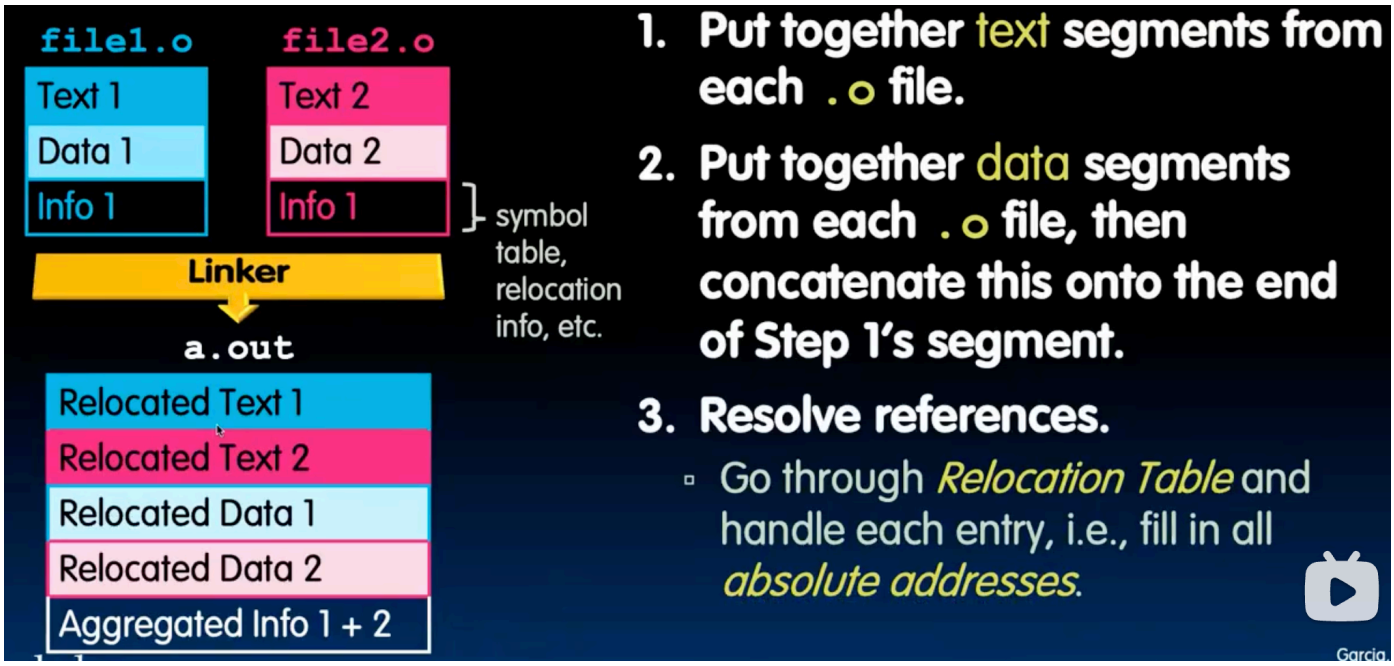


Linker

将目标文件转化为可执行的机器码

The linker enables separate compilation of files

- changes to one file does not require recompilation of the entire program



哪些地址需要重定向

❌PC-Relative addressing

✅External Function Reference

✅Static Data Reference

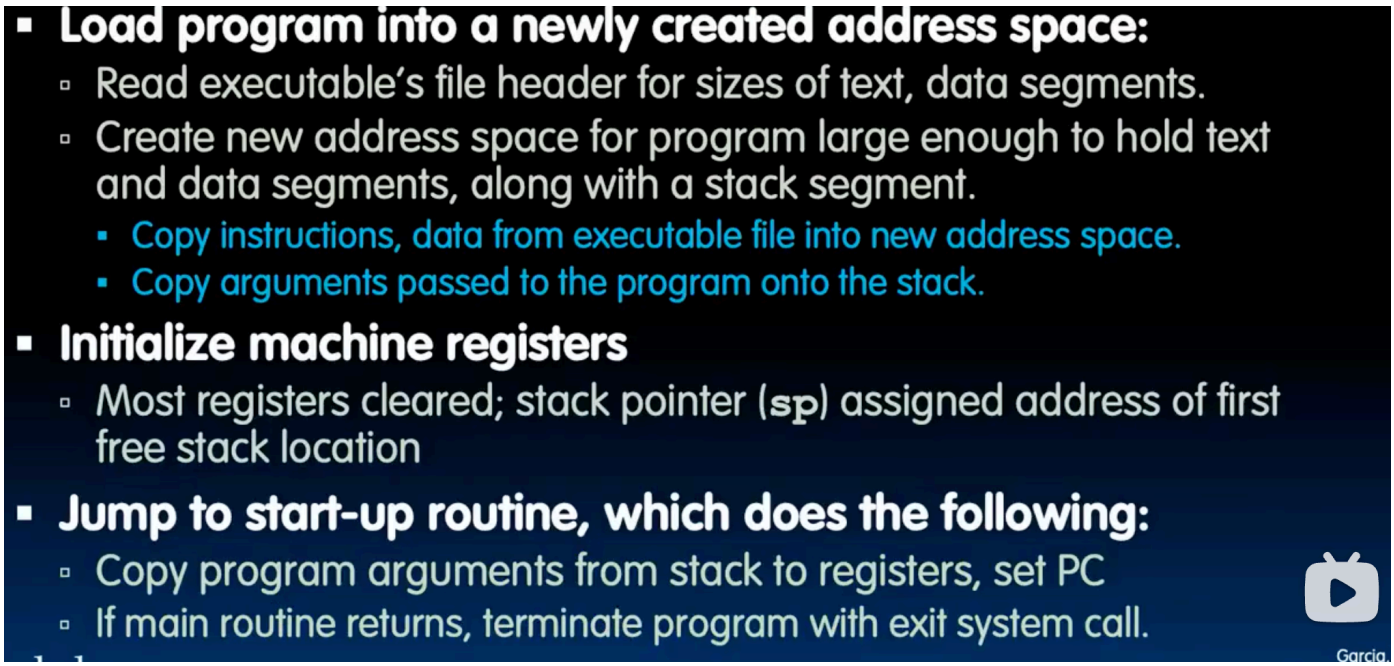
静态链接和动态链接

我们在上面讨论的都是静态链接

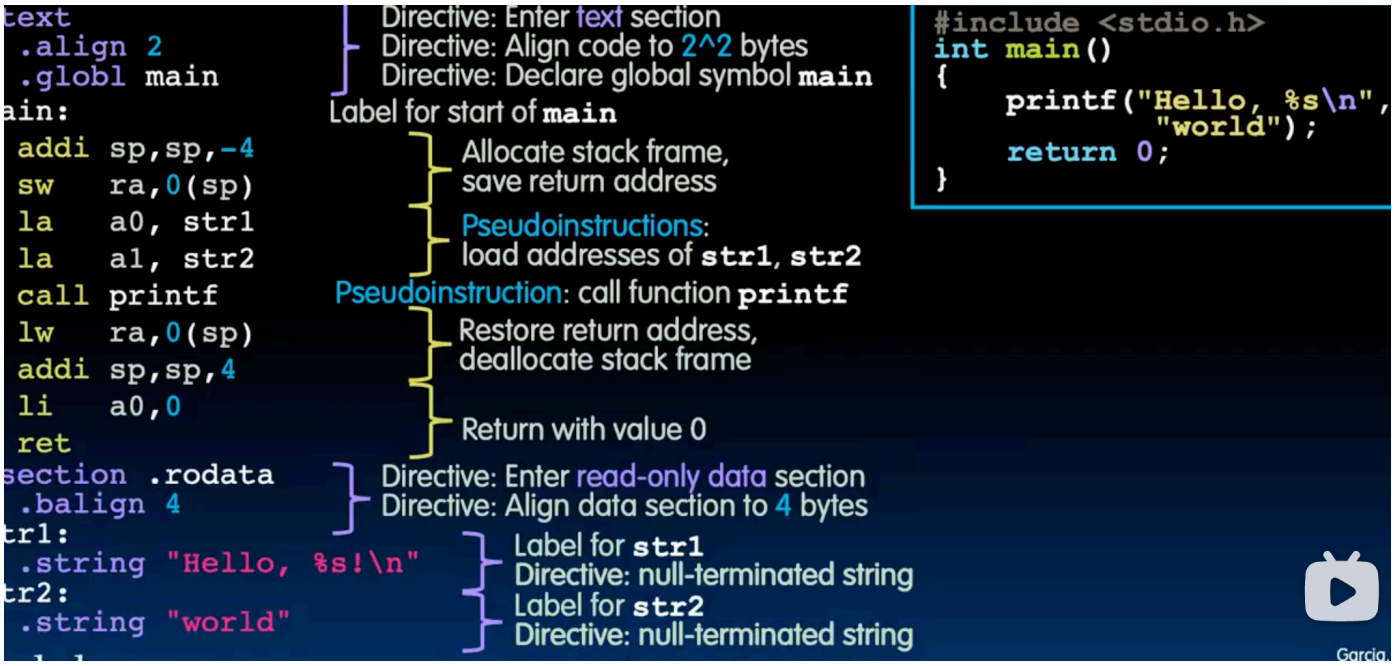
如今更常见的是动态链接DLL（dynamically-linked library)

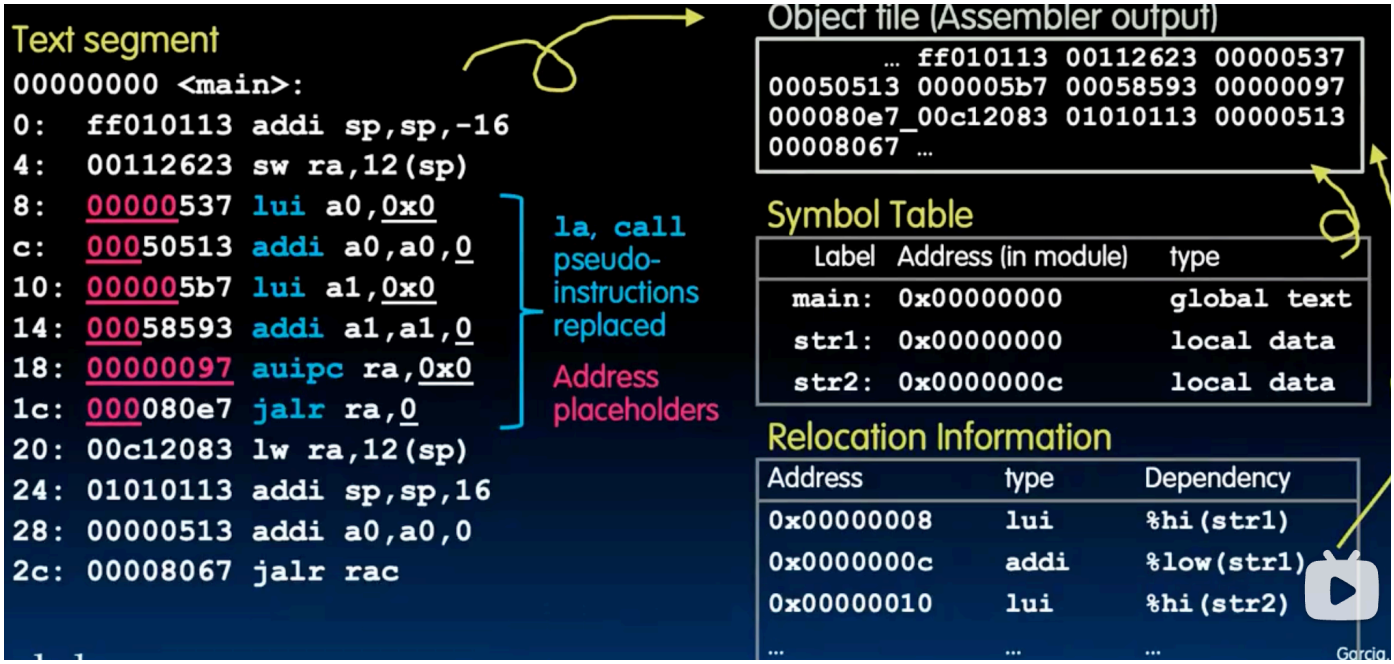
Loader

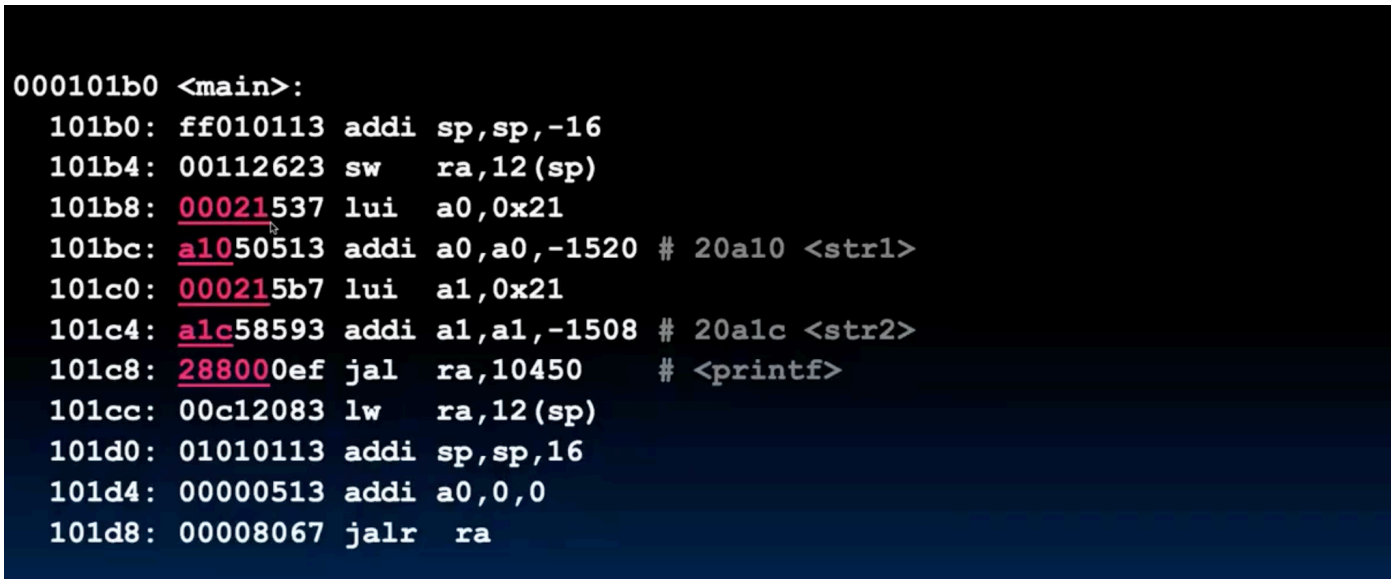
将机器码加载到内存上



Example





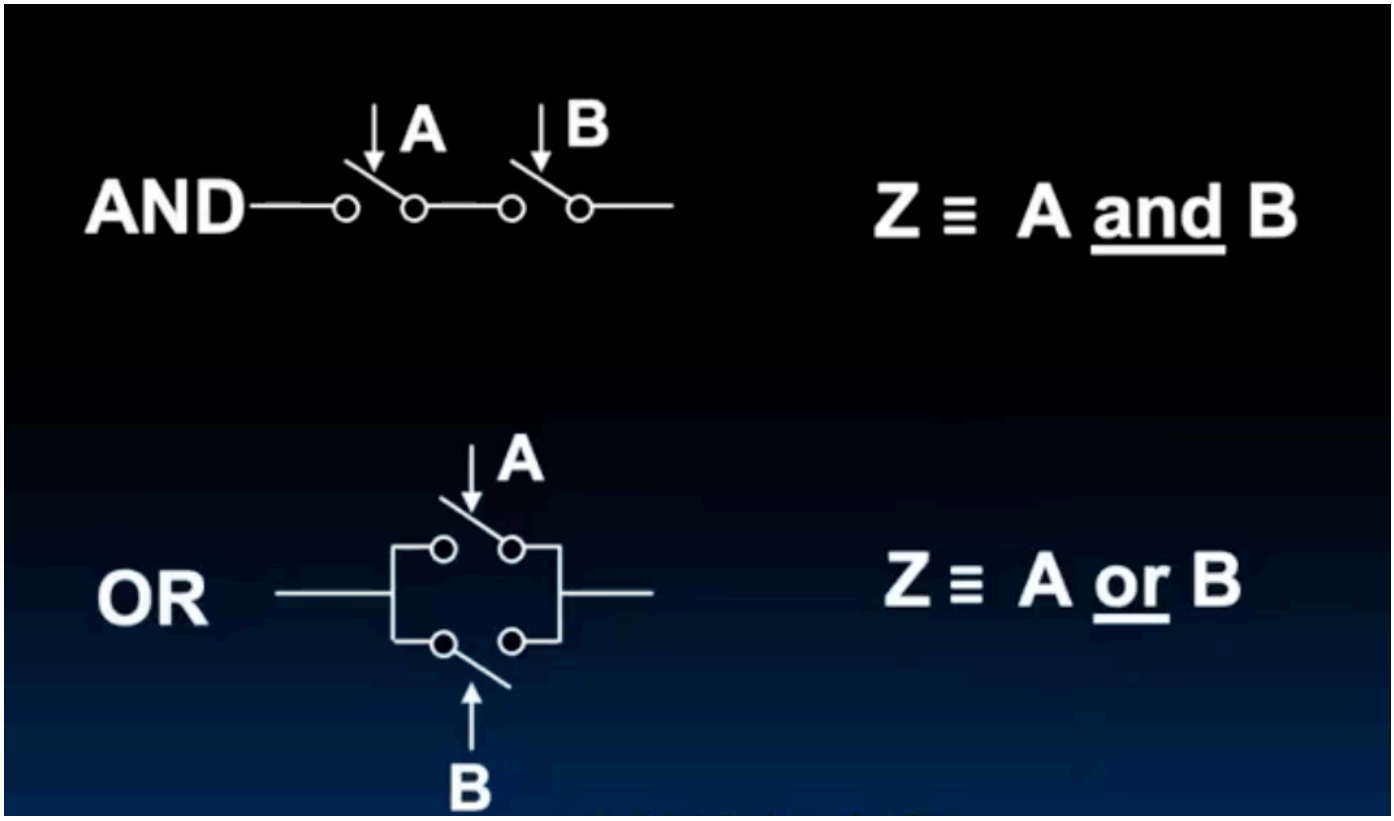


Lecture 14 : Introduction to Synchronous Digital Systems(SDS) : Switches, Transistors, Signals, & Waveforms (同步数字系统)

Switches 开关

Basic Element of Physical Circuit

构建更加复杂的逻辑

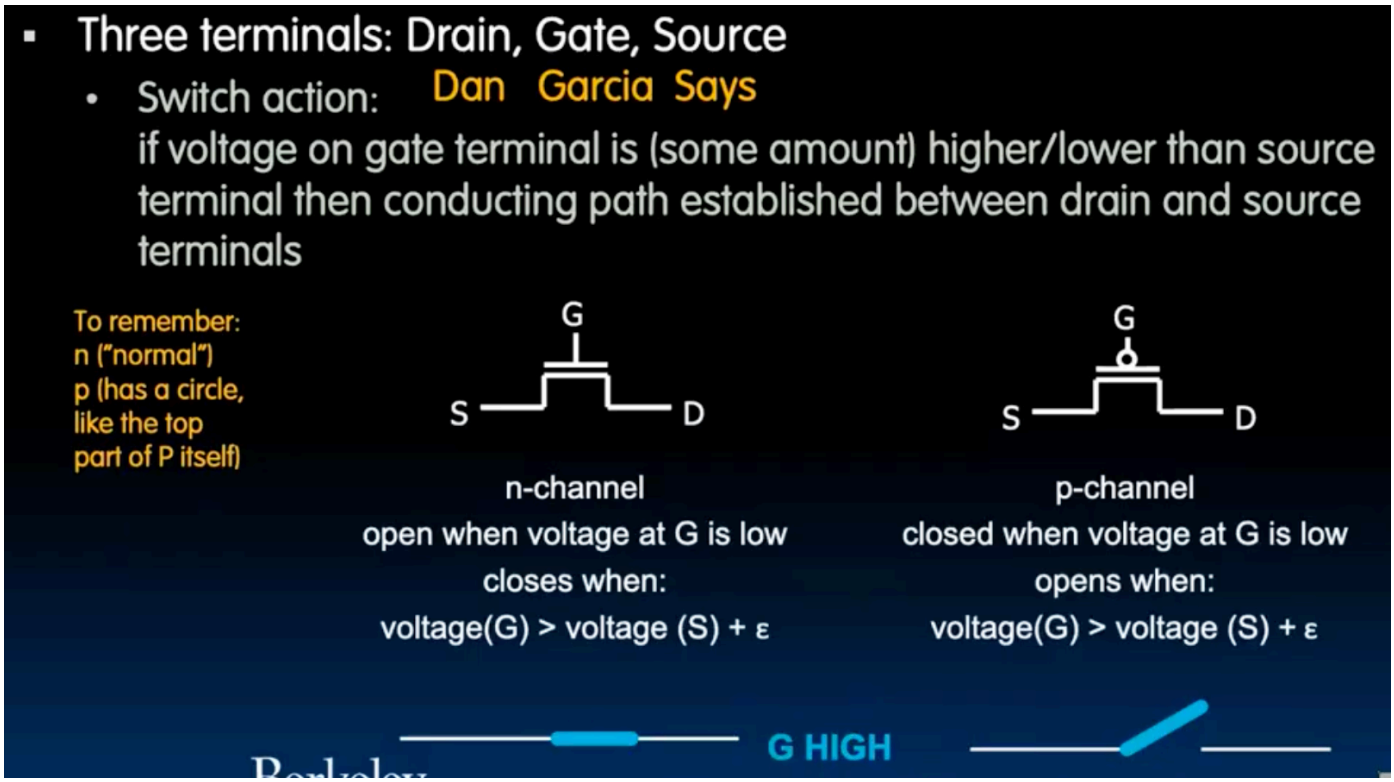


The Transistor("born" 1947-12-23) 晶体管

Modern digital systems designed in CMOS

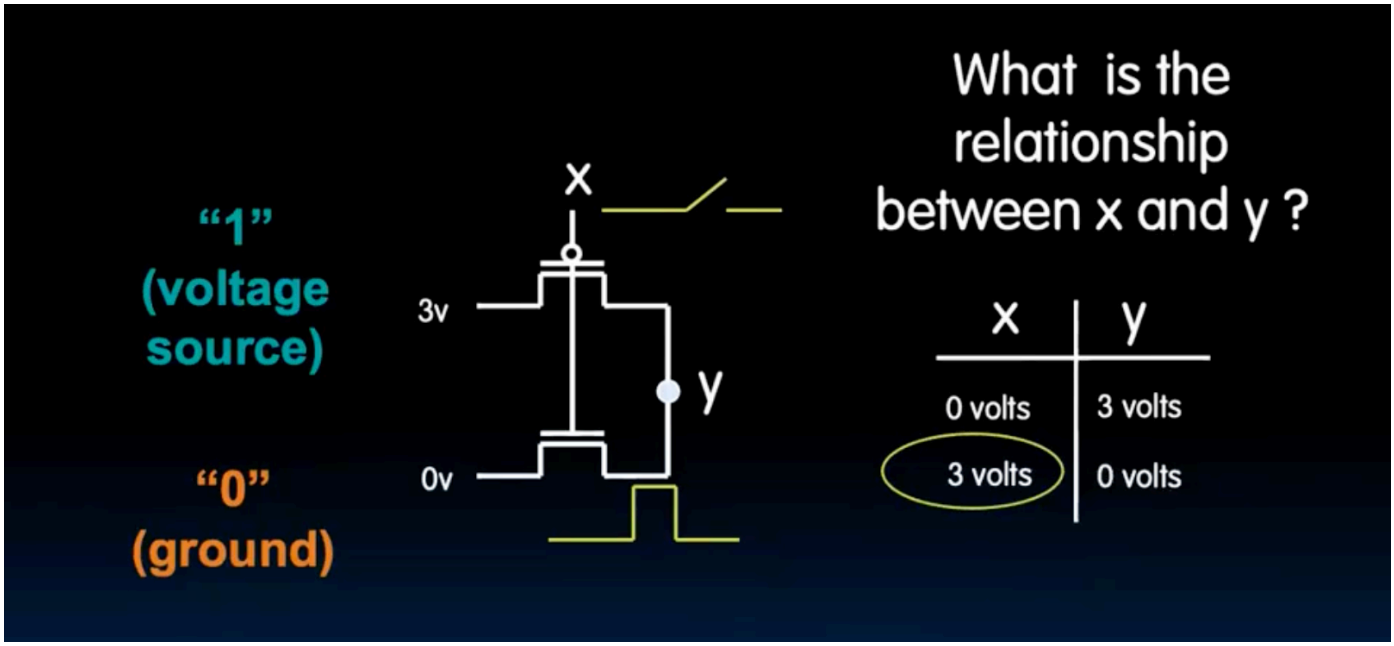
- MOS: Metal-Oxide on Semiconductor
- C for complementary : normally -open and normally-closed switches

MOS transistors act as voltage-controlled switched

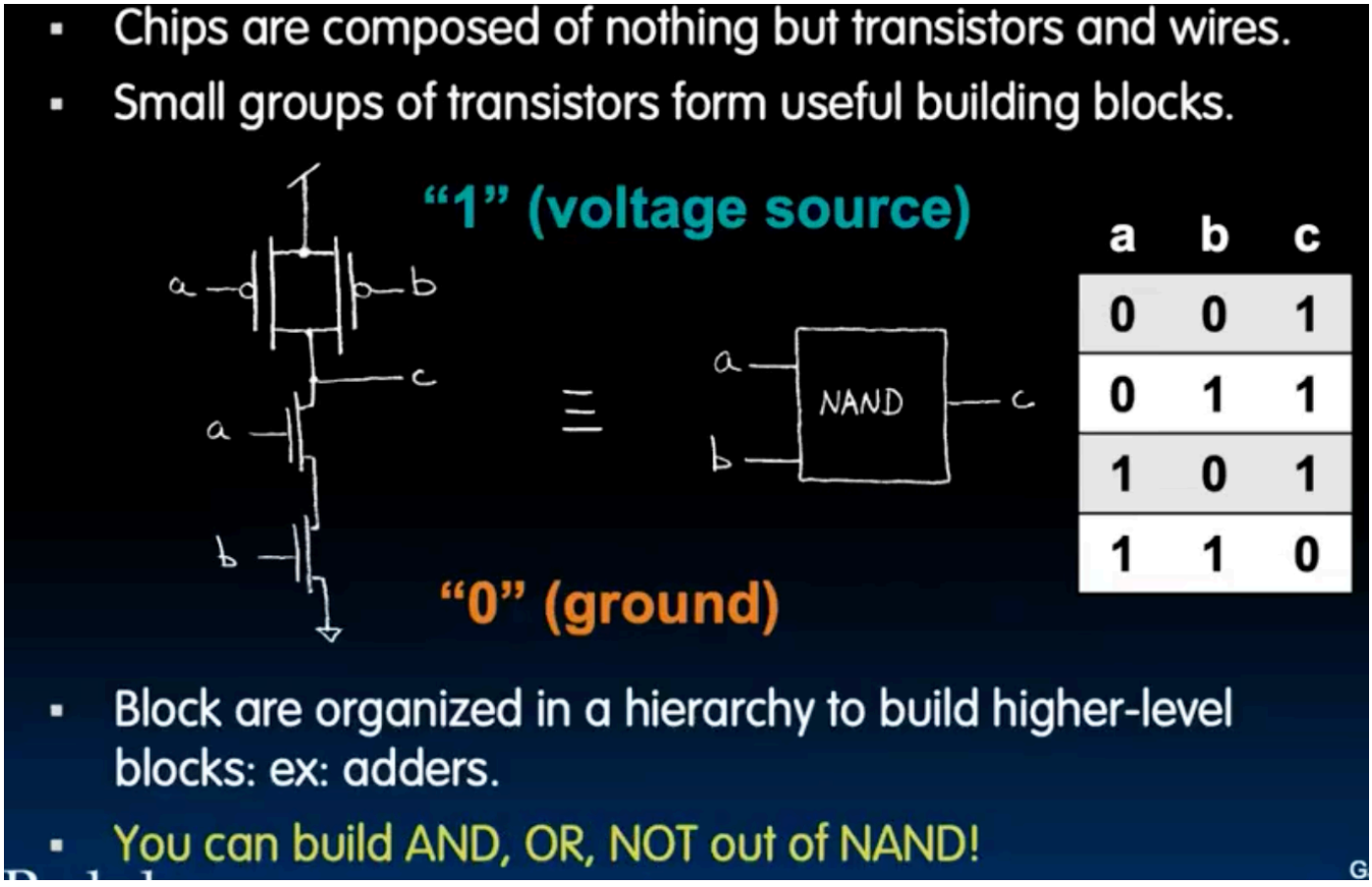


MOS Network

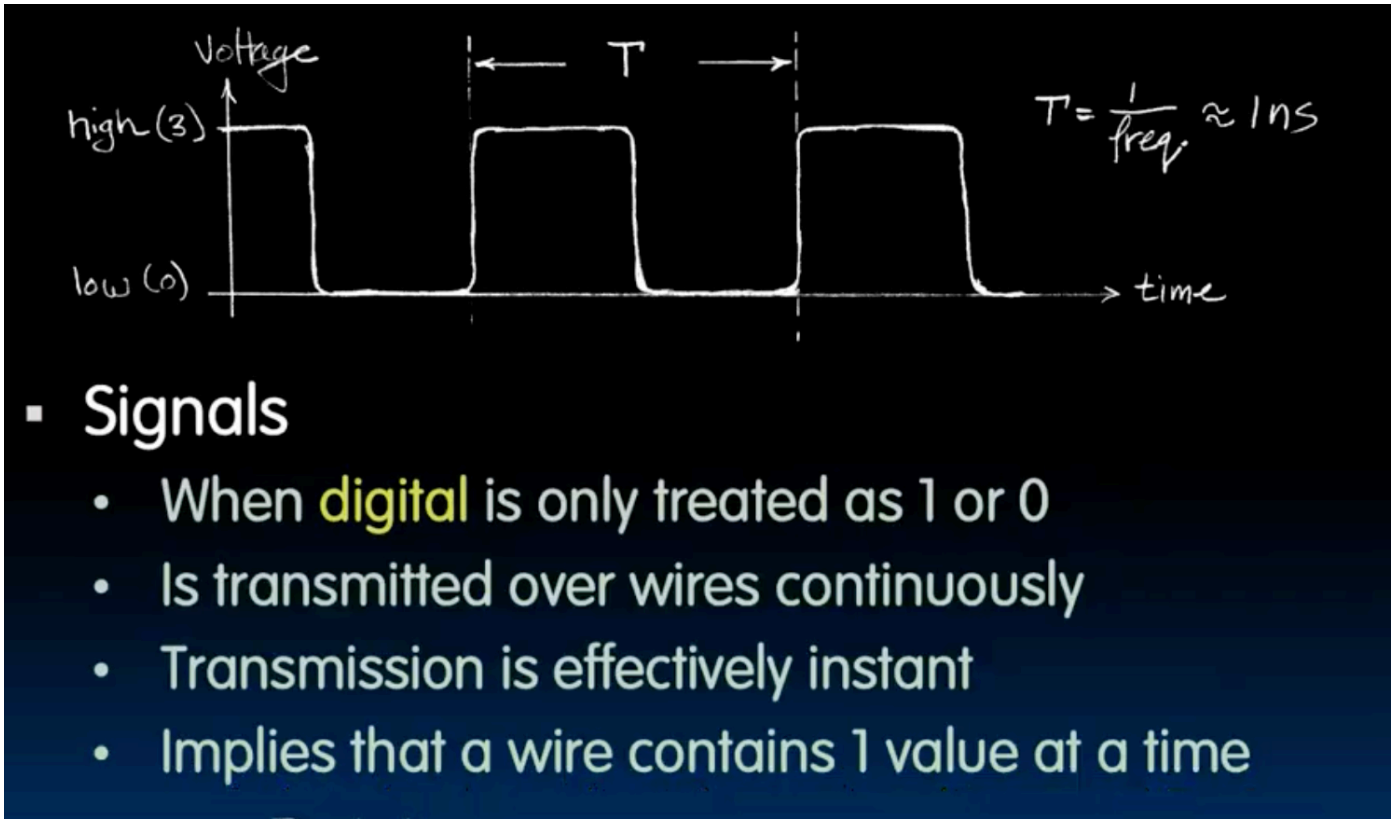
A NOT gate



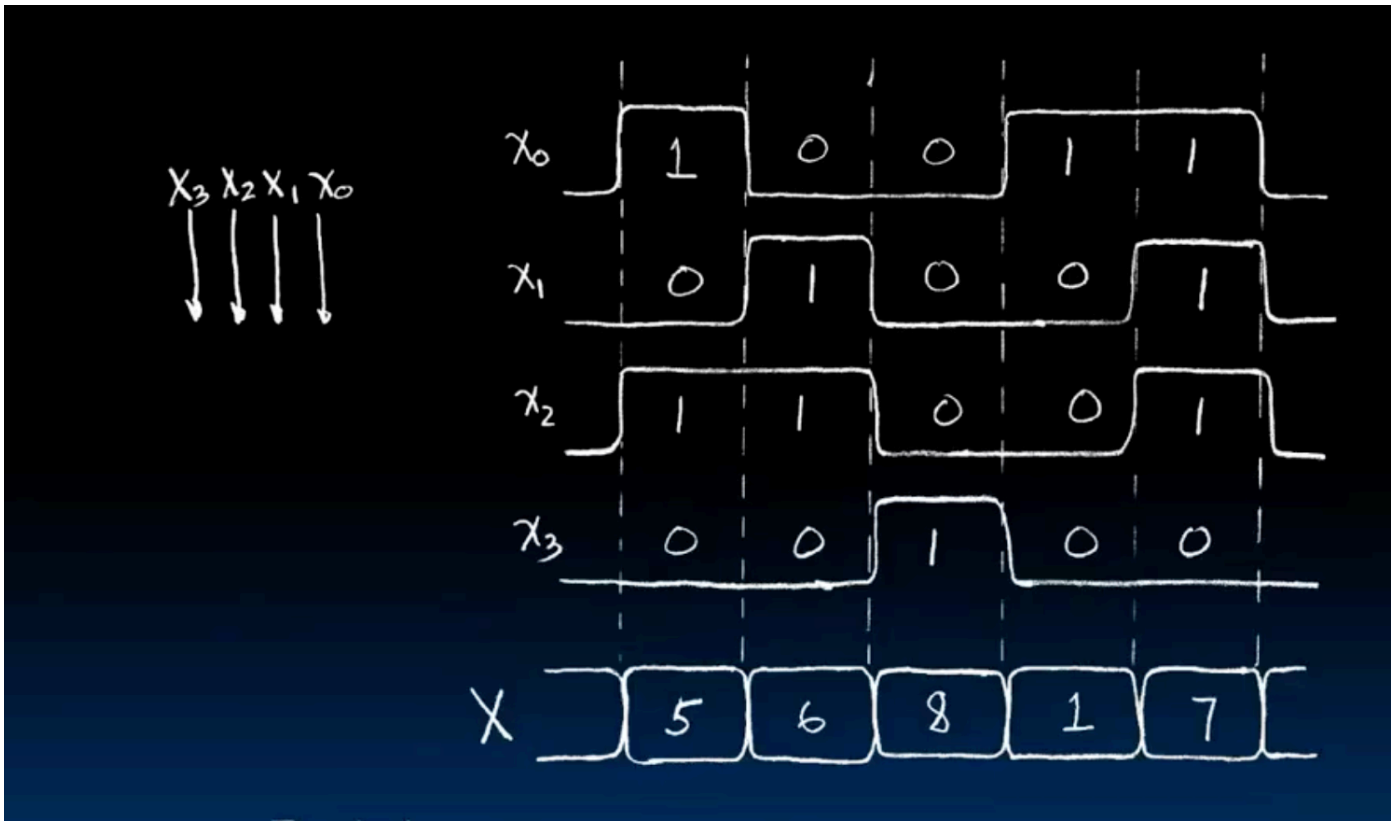
NAND gate (只有都是1时输出为0)



Signals and Waveforms: Clocks

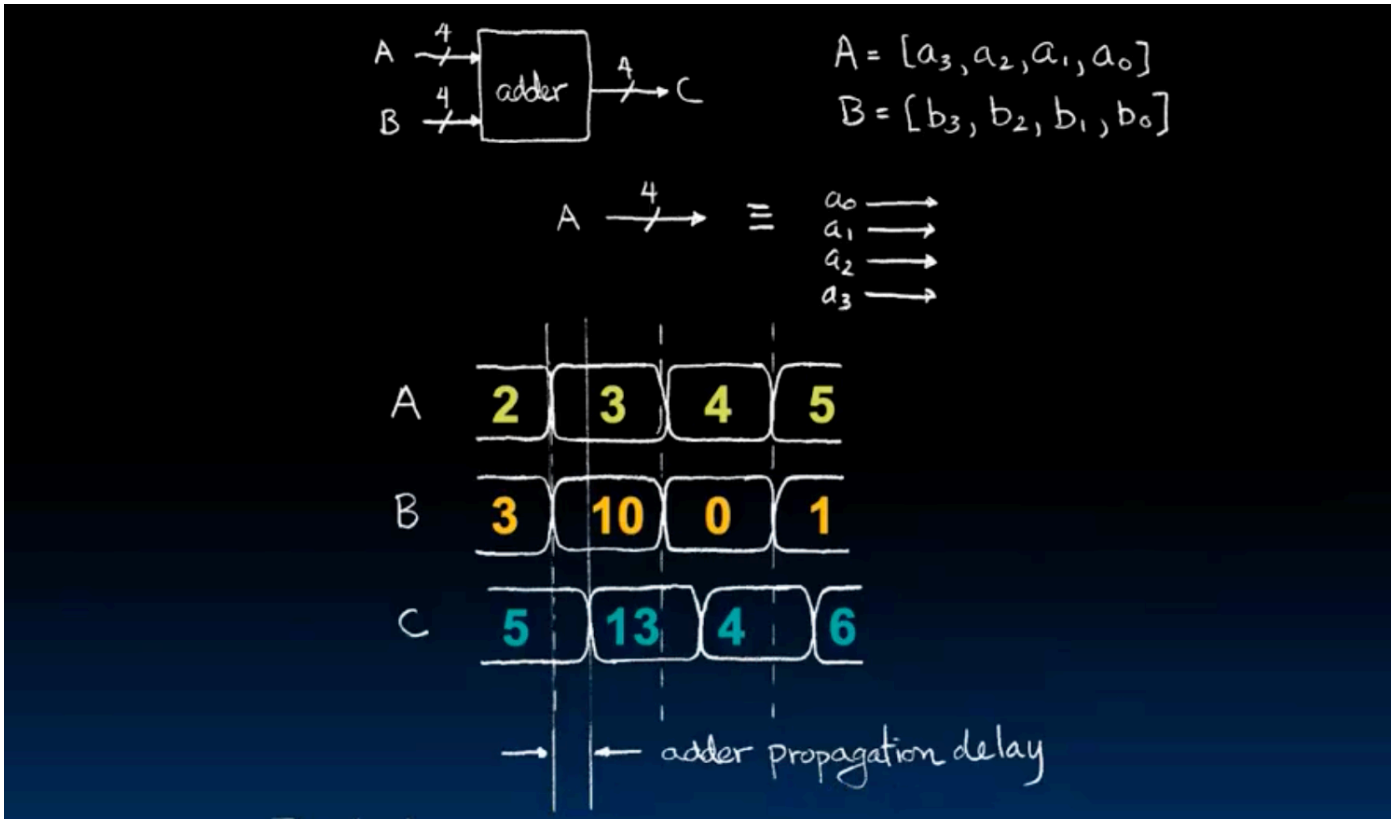


Grouping



`x0 – x3`都是相应的位

加法



Type of Circuits

Combinational Logic (CL) circuits

- 上面的加法电路就是一个例子
- 输出只是输入的一个函数结果
- 不存储信息

State Element

- 存储信息的电路

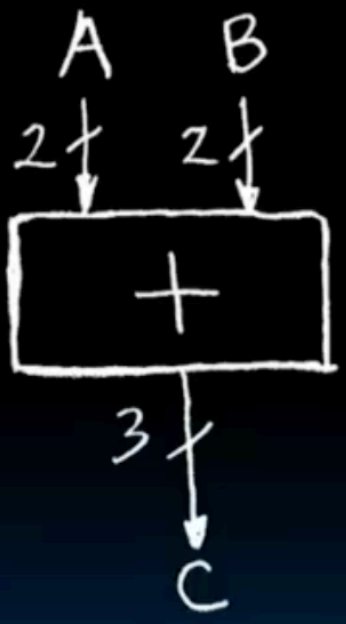
Lecture 15： Combinational Logic

条件相反器

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

a	y
0	b
1	$\bar{b}$

二位相加器 two-bit adder

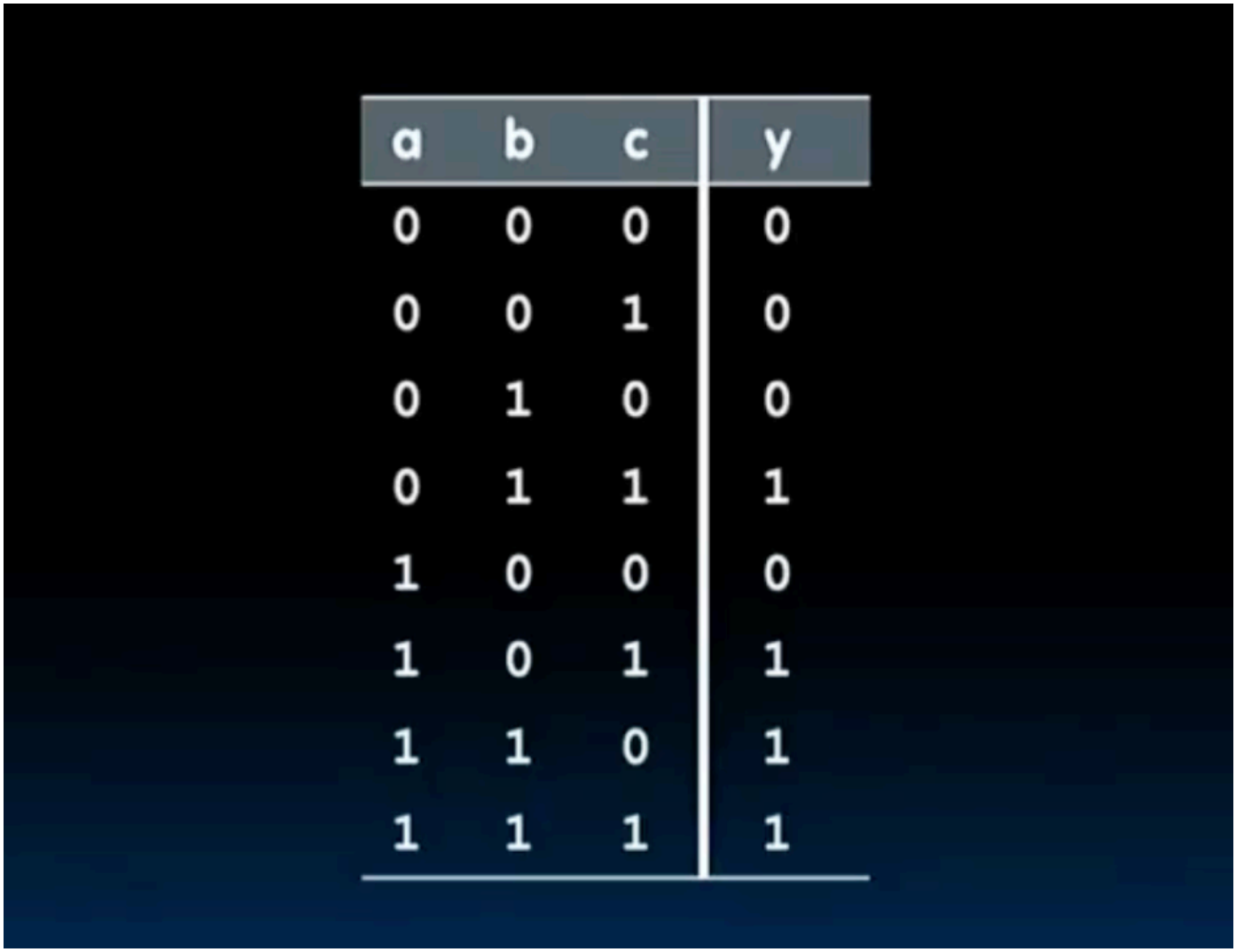


A a_1a_0	B b_1b_0	C $c_2c_1c_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

32位无符号相加器 32-bit unsigned adder

2^{64} 行

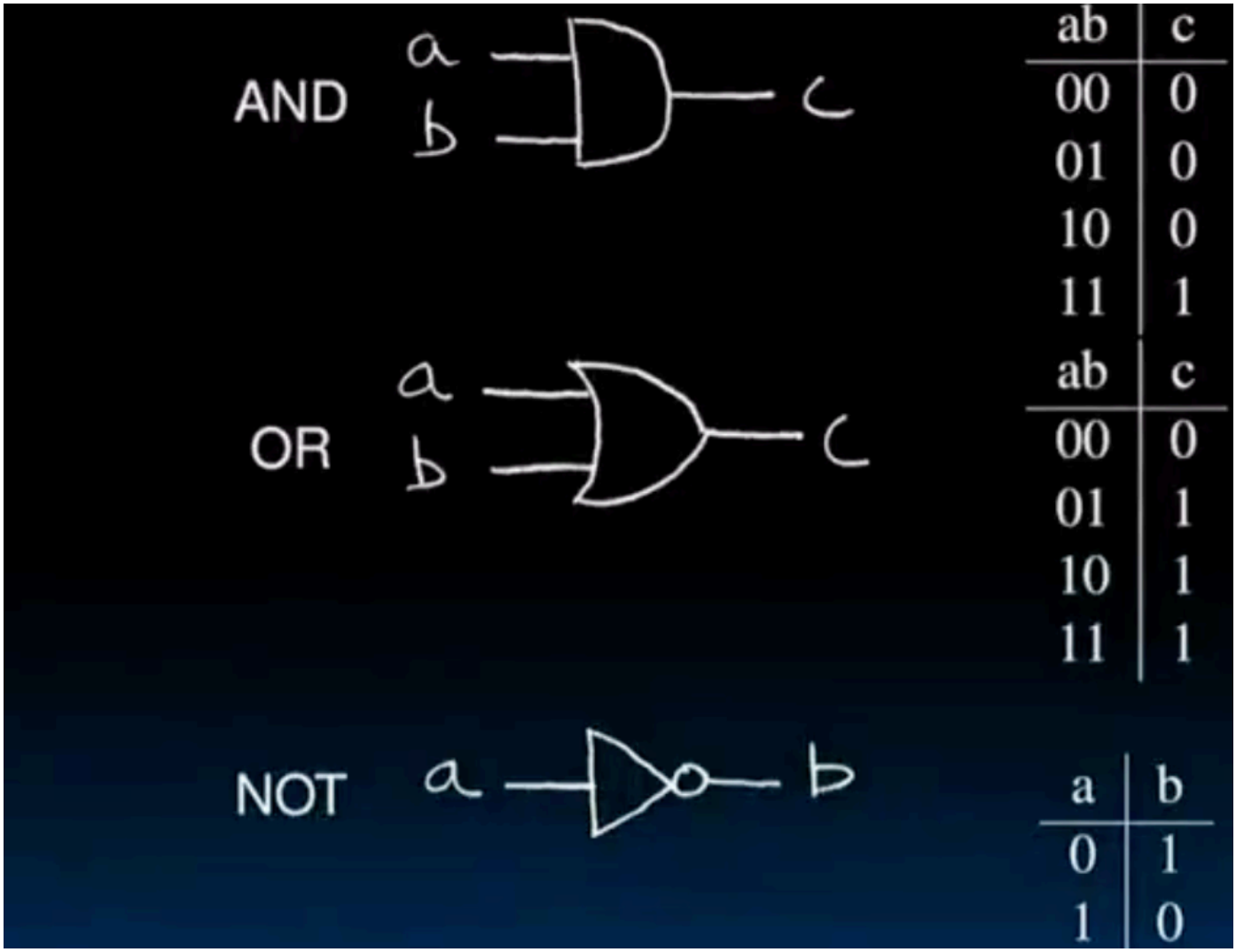
三输入的多数电路 3-input majority circuit



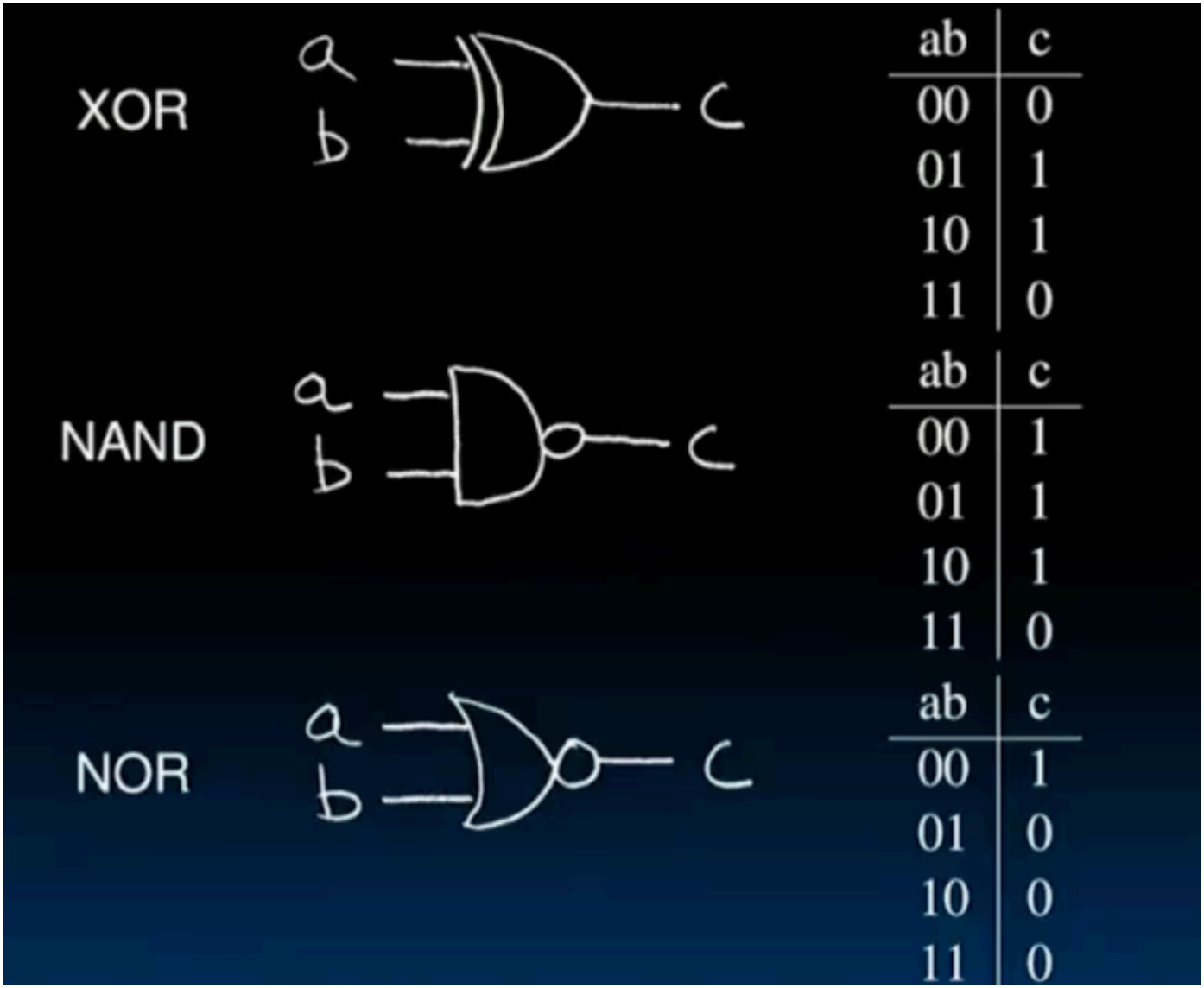
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Logic Gates

`AND OR NOT`


AND
a
b
c
ab | c
00 | 0
01 | 0
10 | 0
11 | 1
OR
a
b
c
ab | c
00 | 0
01 | 1
10 | 1
11 | 1
NOT
a
b
a | b
0 | 1
1 | 0

`XOR NAND NOR`

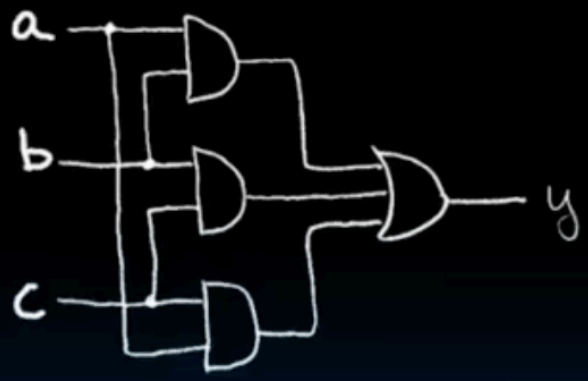


2-input gates extend to n-inputs

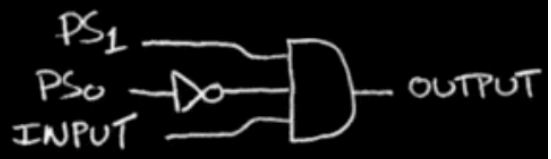
依次进行真值判断即可

Truth Table -> Gates

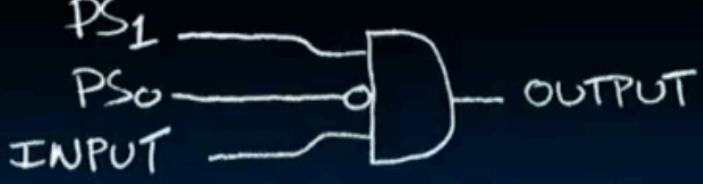
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1



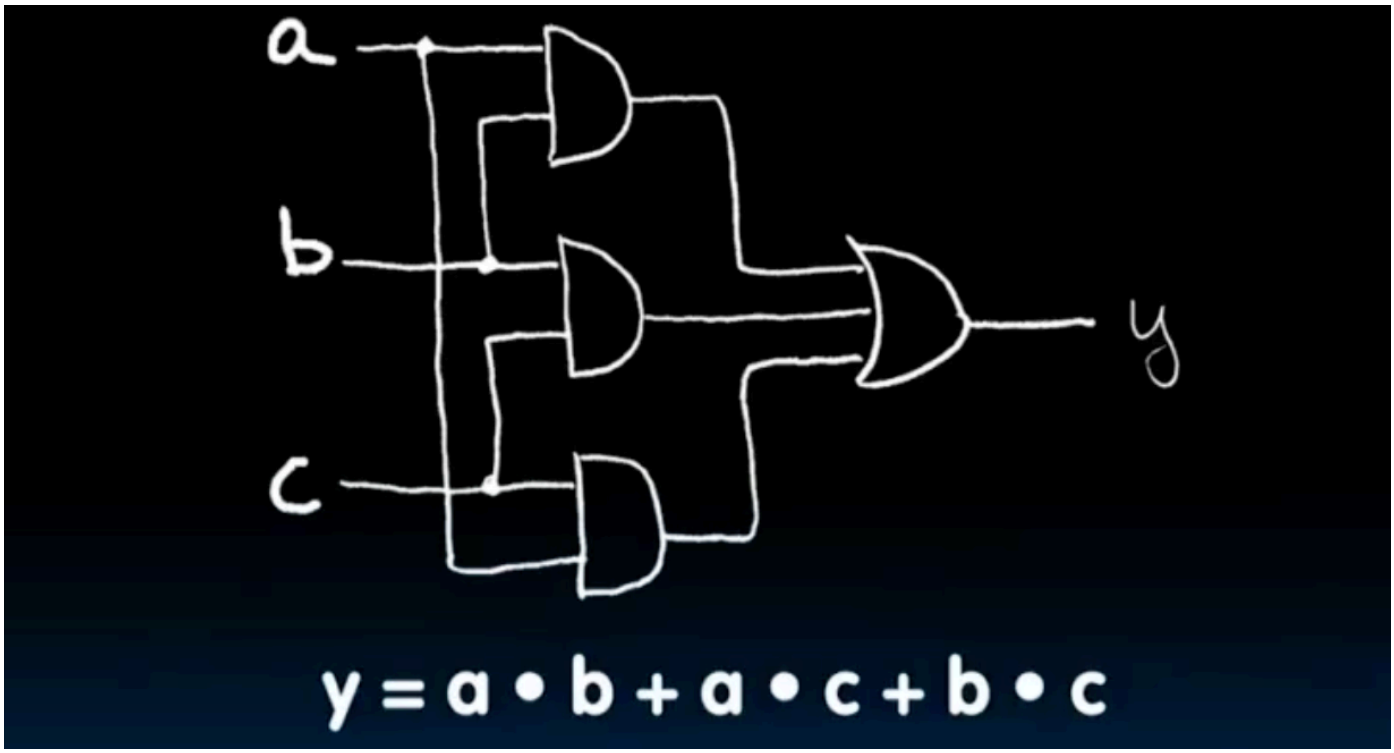
or equivalently...



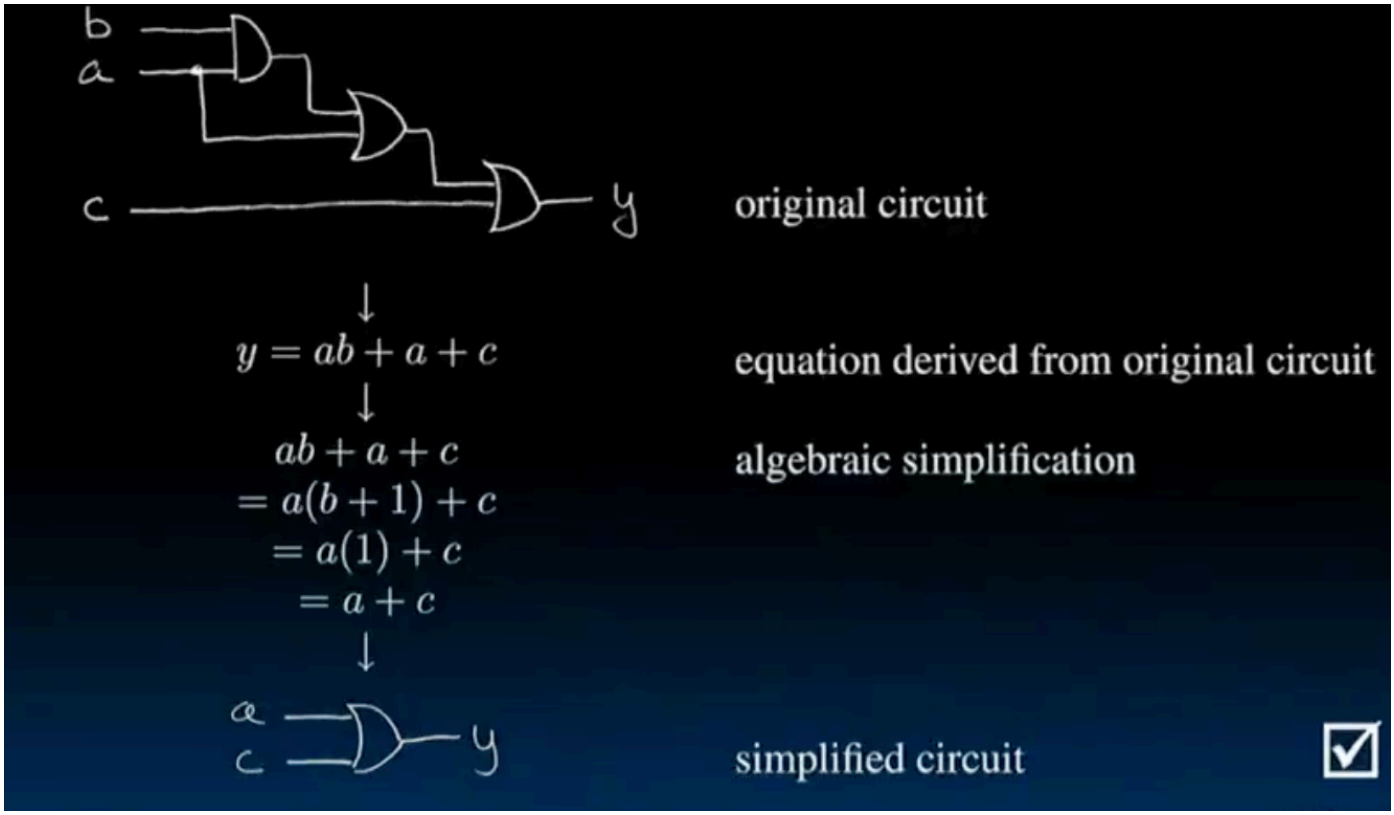
Boolean Algebra

George Boole, 19th Century mathematician

举例

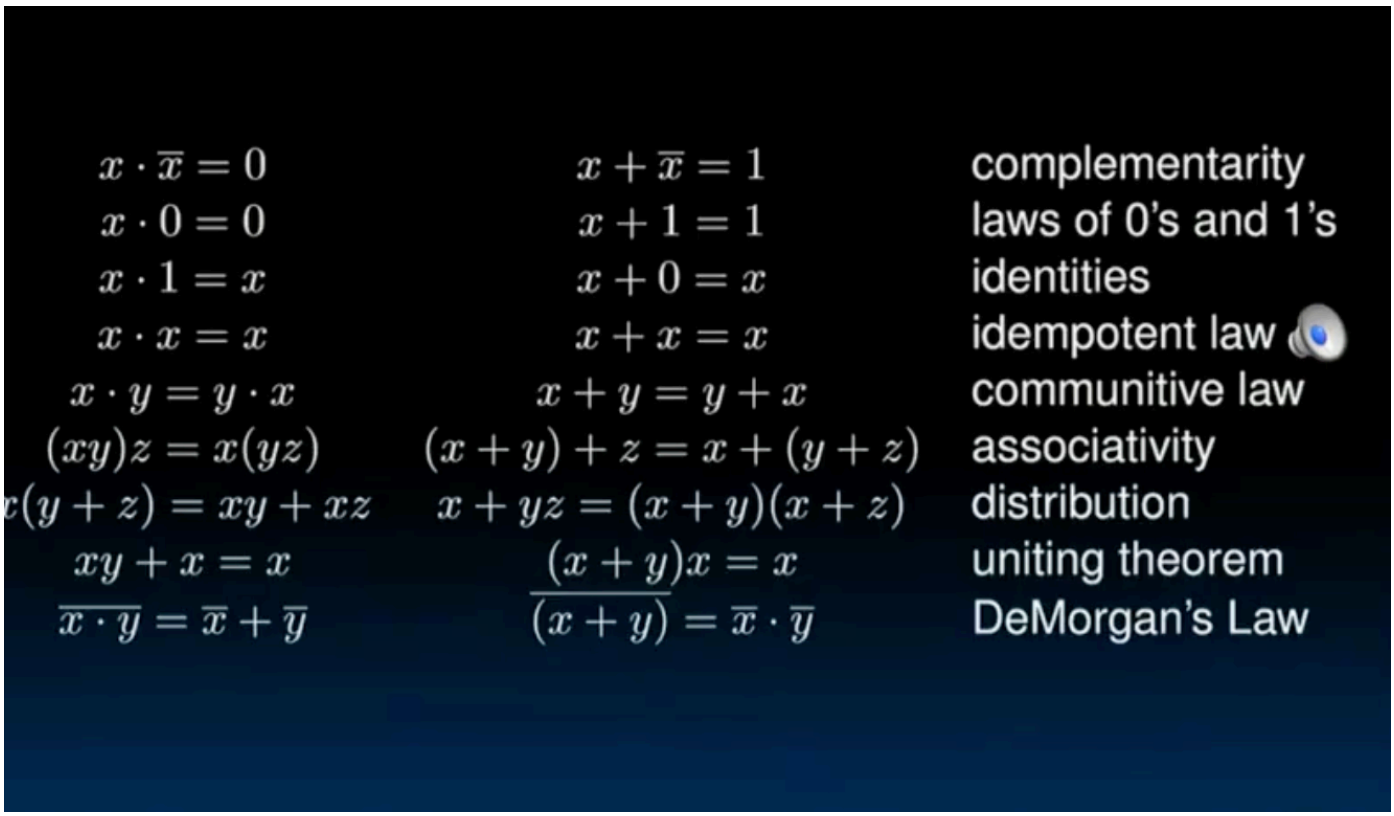


根据Boolean Algebra简化电路

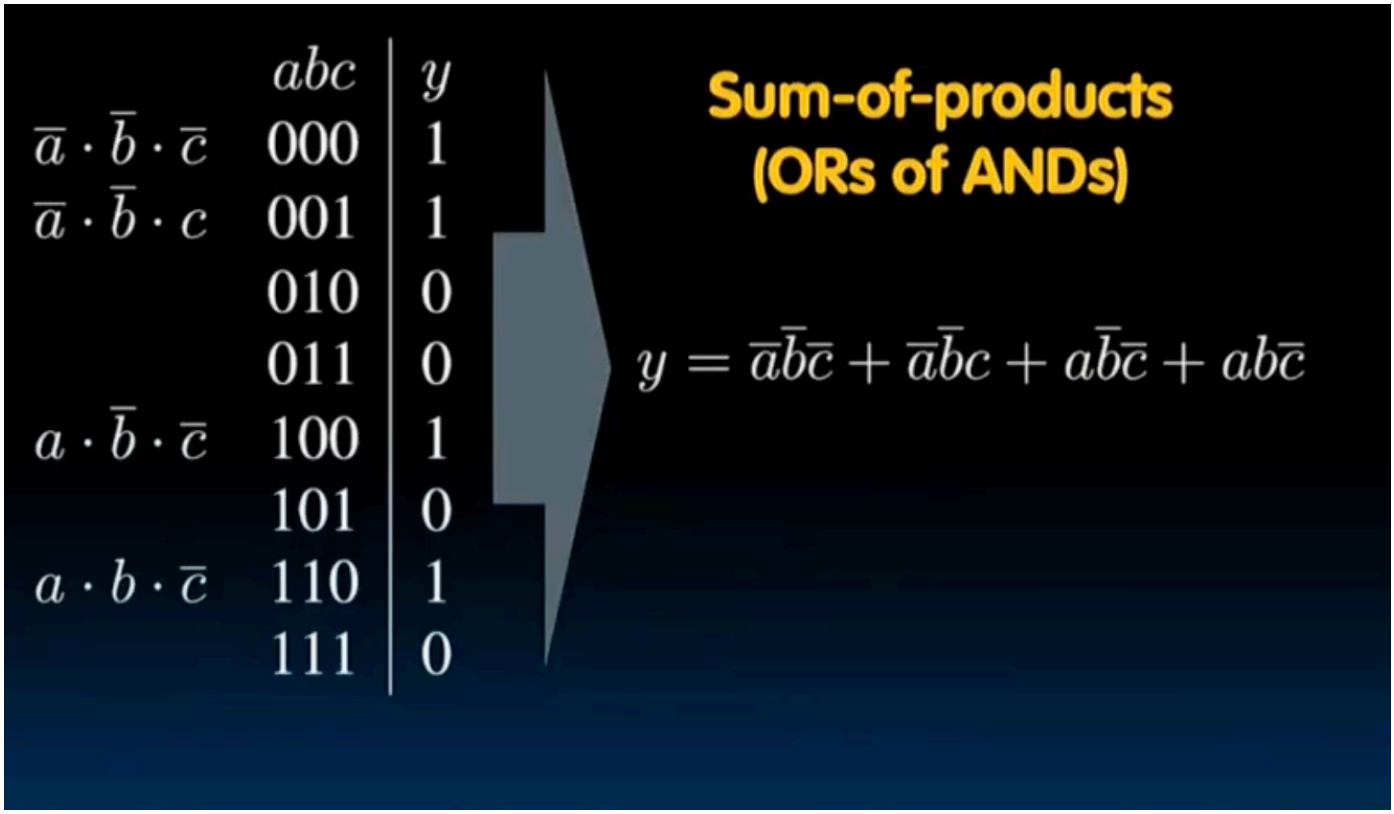


可以用布尔代数判断两个电路是否等价

Laws of Boolean Algebra



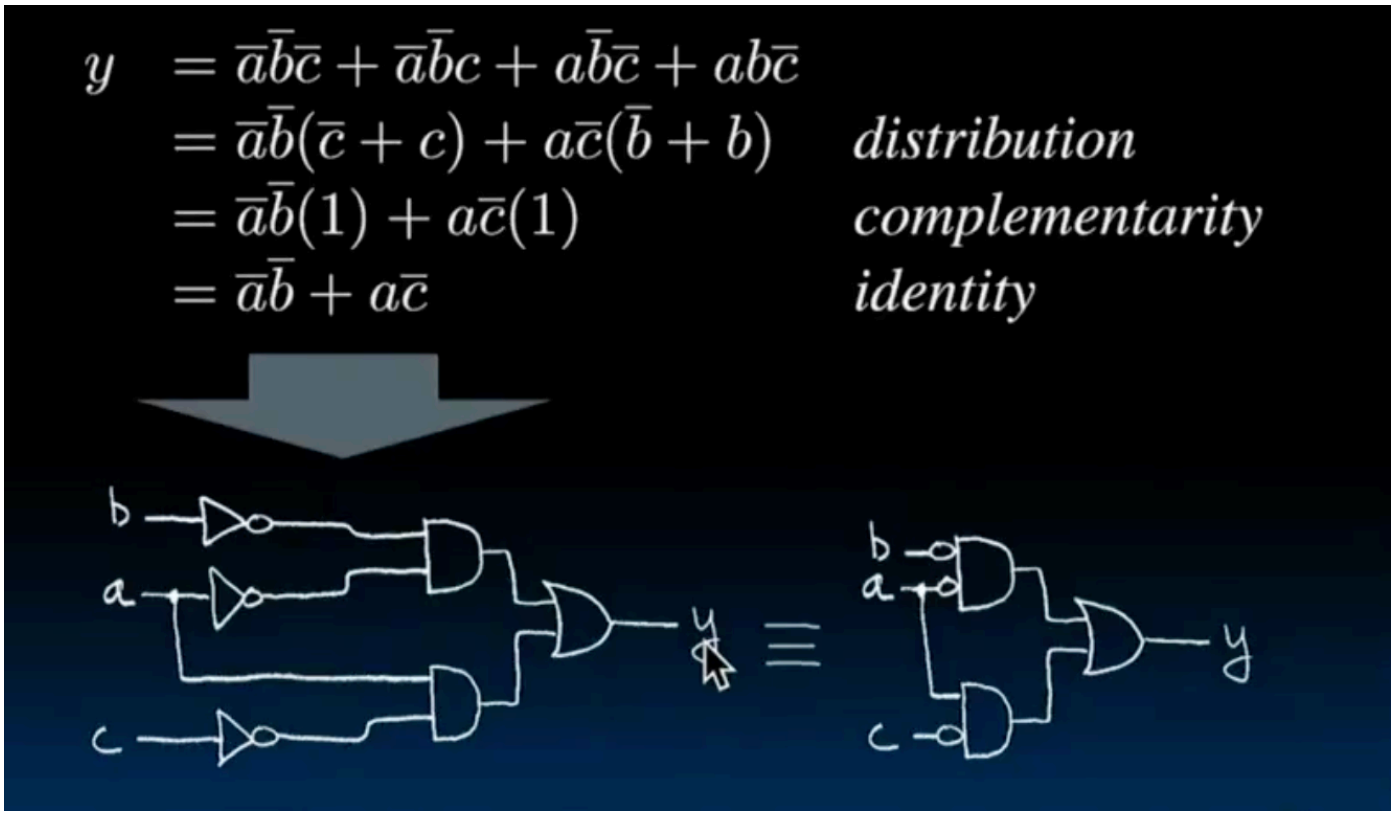
Canonical forms 标准格式



根据真值表写出布尔表达式的规则，其实就是把**输出为 1 的所有输入组合**提取出来，然后用**“析取范式”（和项的“或”）**来表示逻辑函数

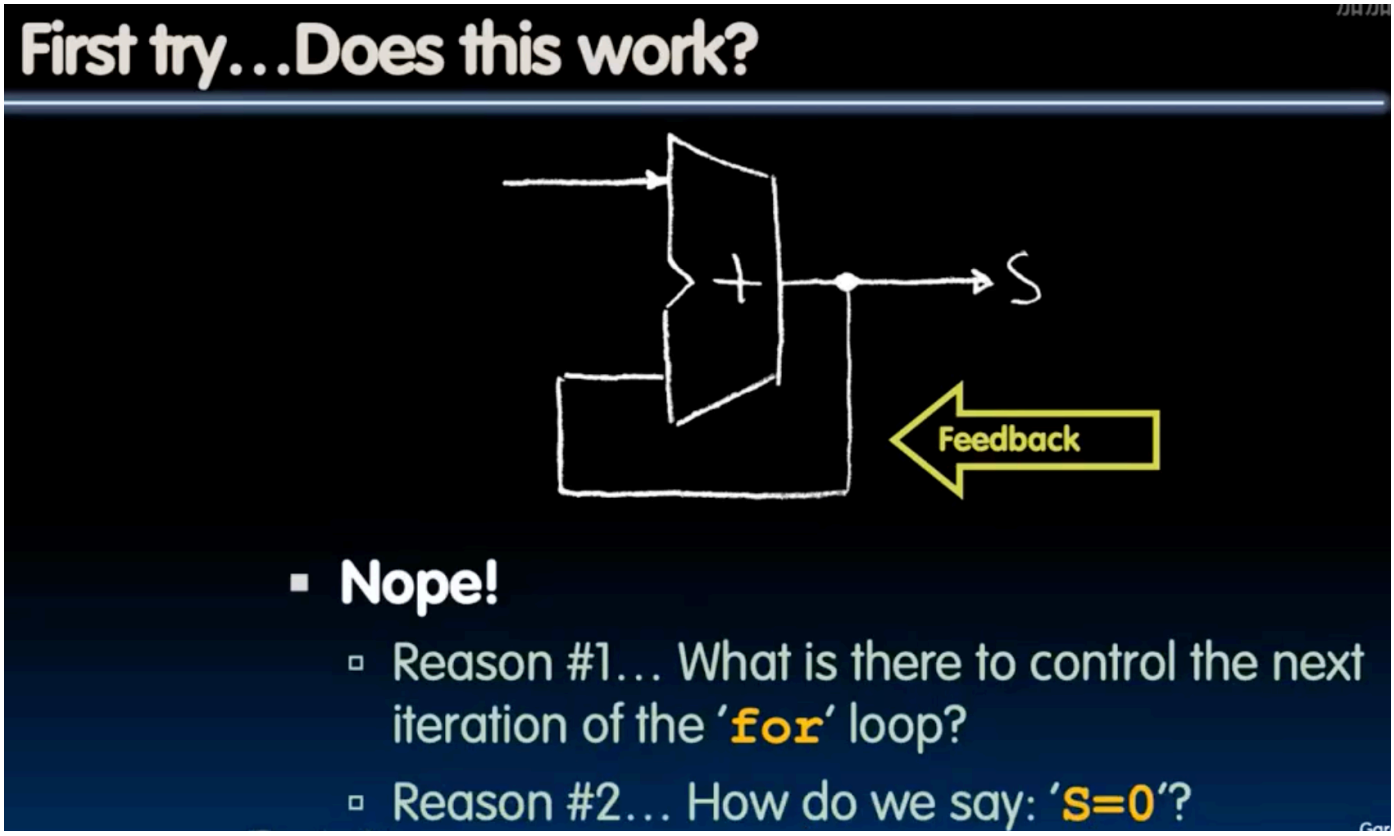
这叫做 **最小项和表达法**，又叫 **析取范式（Sum of Products, SOP）**

使用析取范式是因为，析取范式包含了可以使电路为真的所有可能，用或链接，只要满足一个我们的电路就可以连通

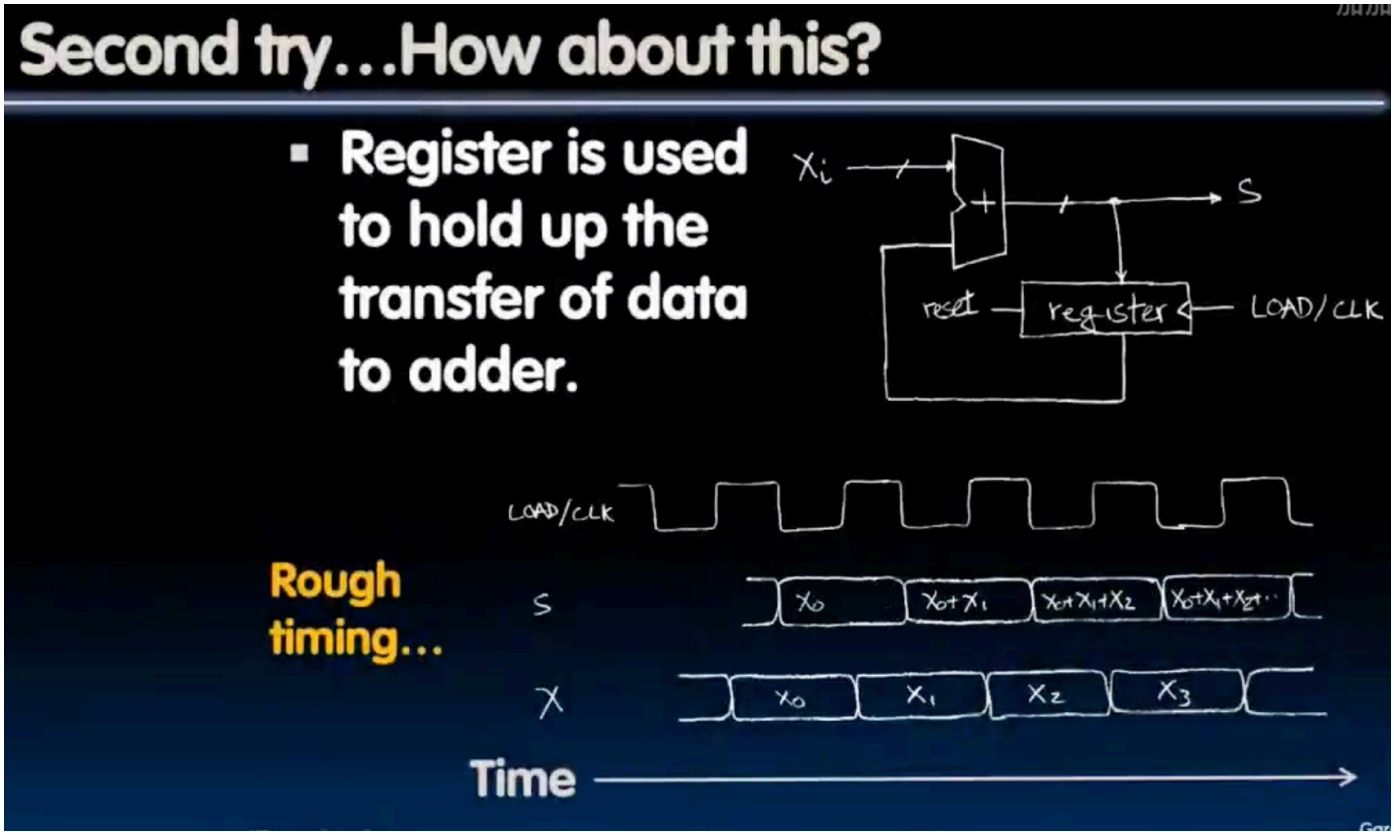


Lecture 16 : Synchronous Digital System (SDS) State

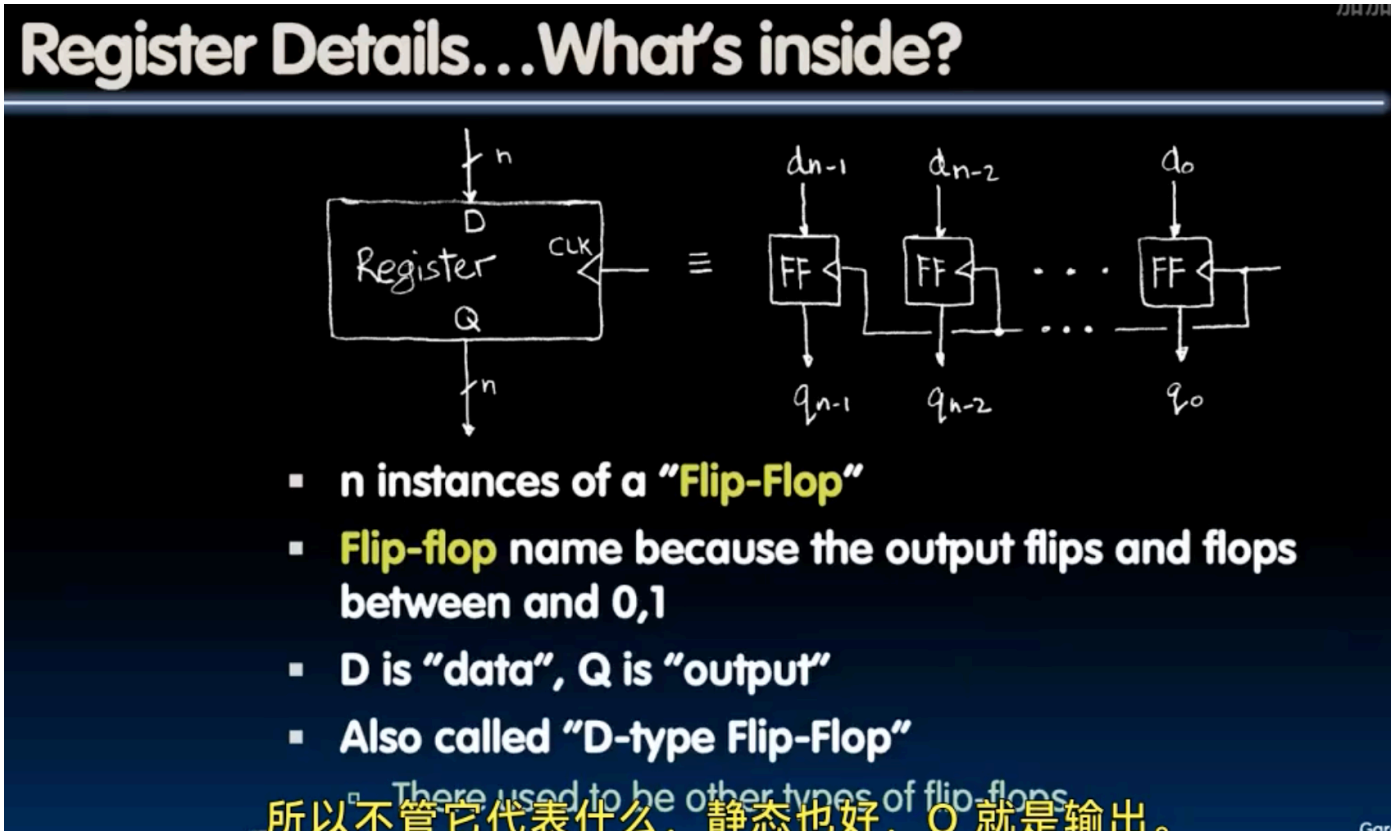
Accumulator



没法控制循环的结束，也没办法将S进行重置

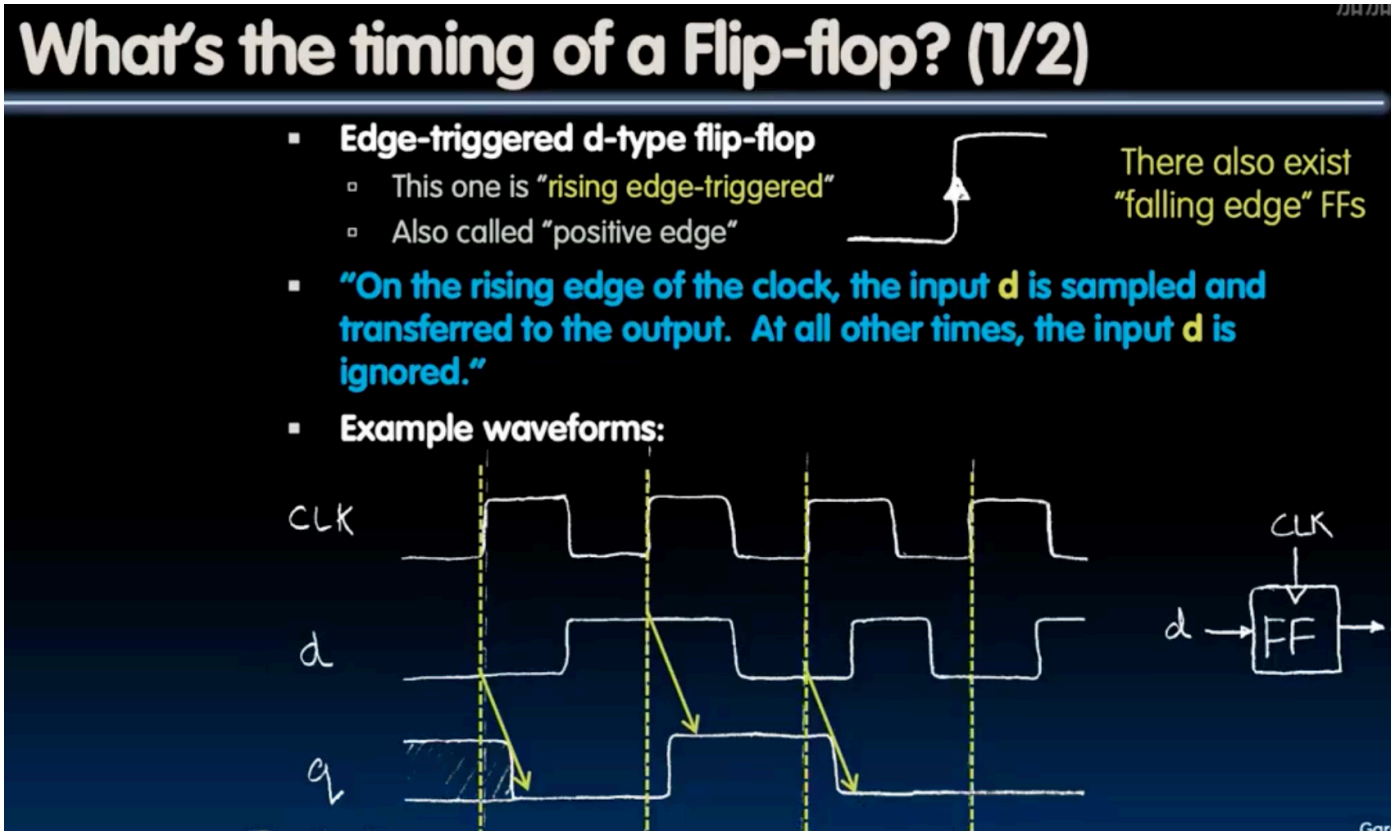


Register Details Flip-flops(触发器)

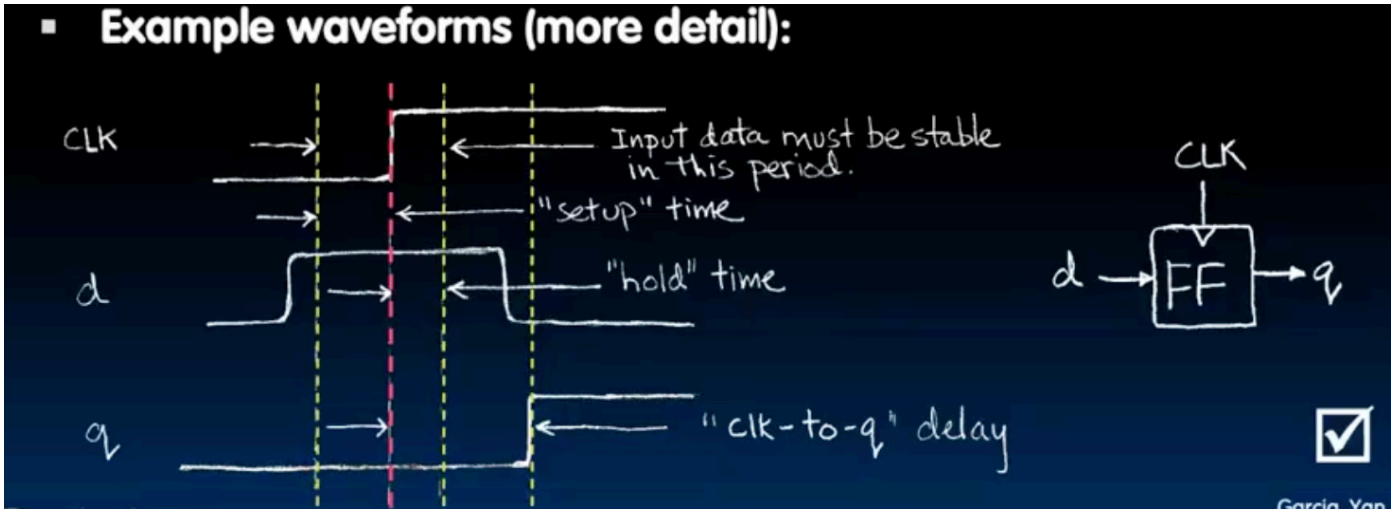


D (data) 是输入

Q是输出

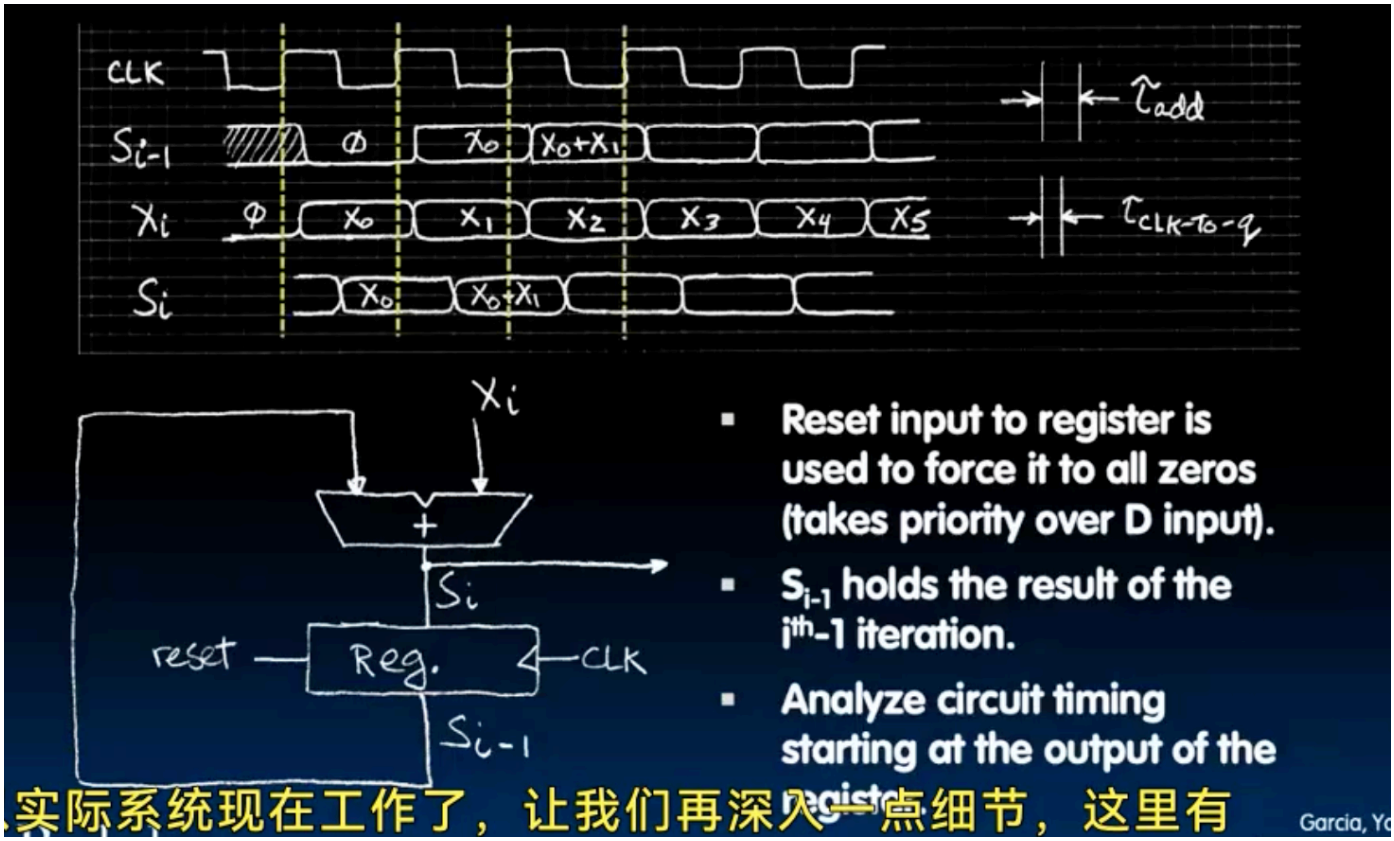


若d端输入在clock信号上沿之前的一段时间（建立时间）和之后的一段时间（维持时间）保持稳定不变，那么保证q端输出是稳定的，否则输入无效

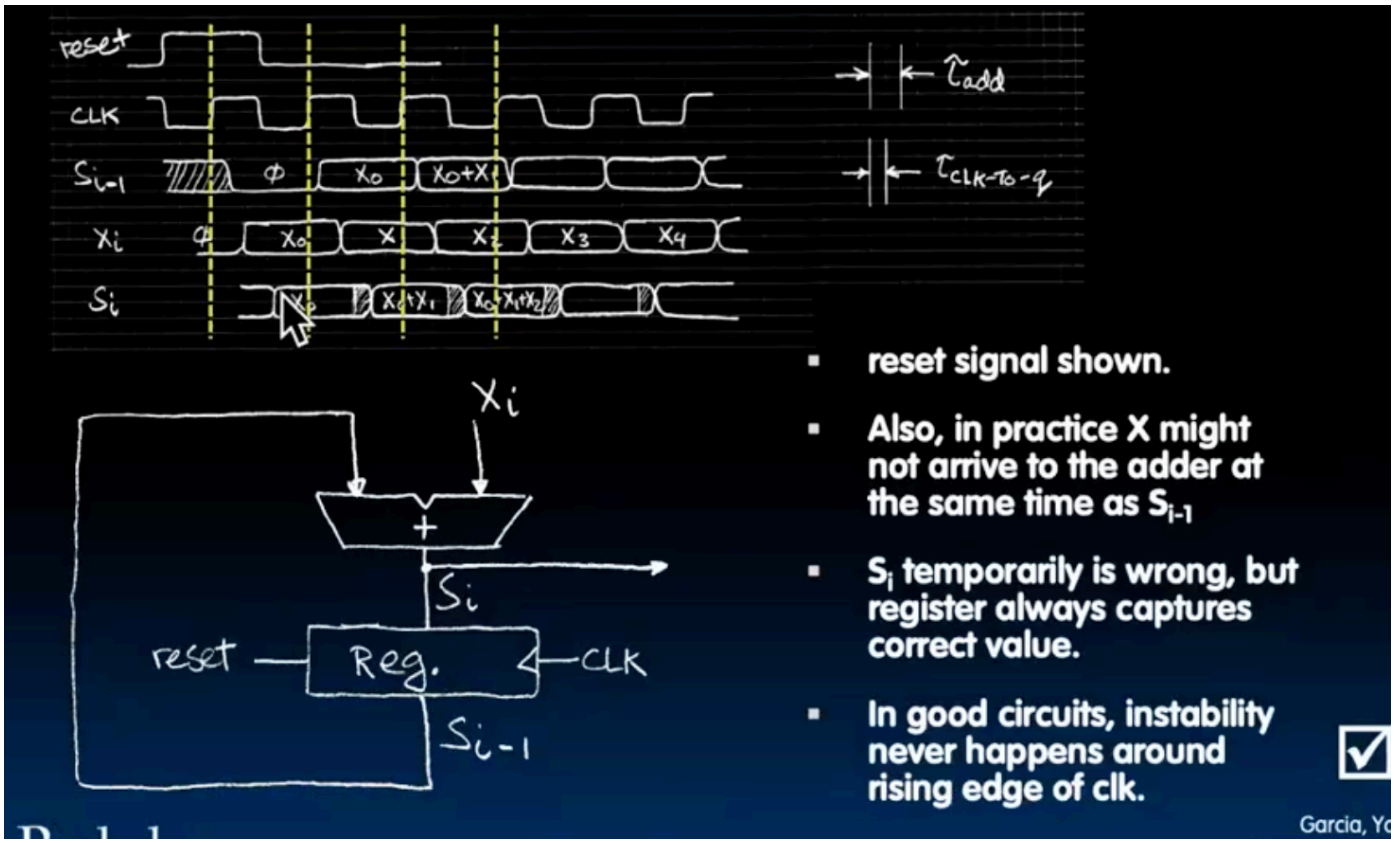


回到累加器

寄存器里锁定的是上一次的计算结果，所以S_{i-1}反而在S_i下面



reset也是一种输入，所以也要满足输入稳定的原则，当接收到reset信号，结果变为空，即清零



注意：在第二条黄线的右侧一点点，计算结果是x0 + x0显示在s_i 的阴影部分，所以s_i是错的，但是s_{i-1}是对的

所以图中的输出线不应该是s_i, 应该是s_{i-1}

Pipeling for Performance 流水线操作

最大化时间效率

period 周期

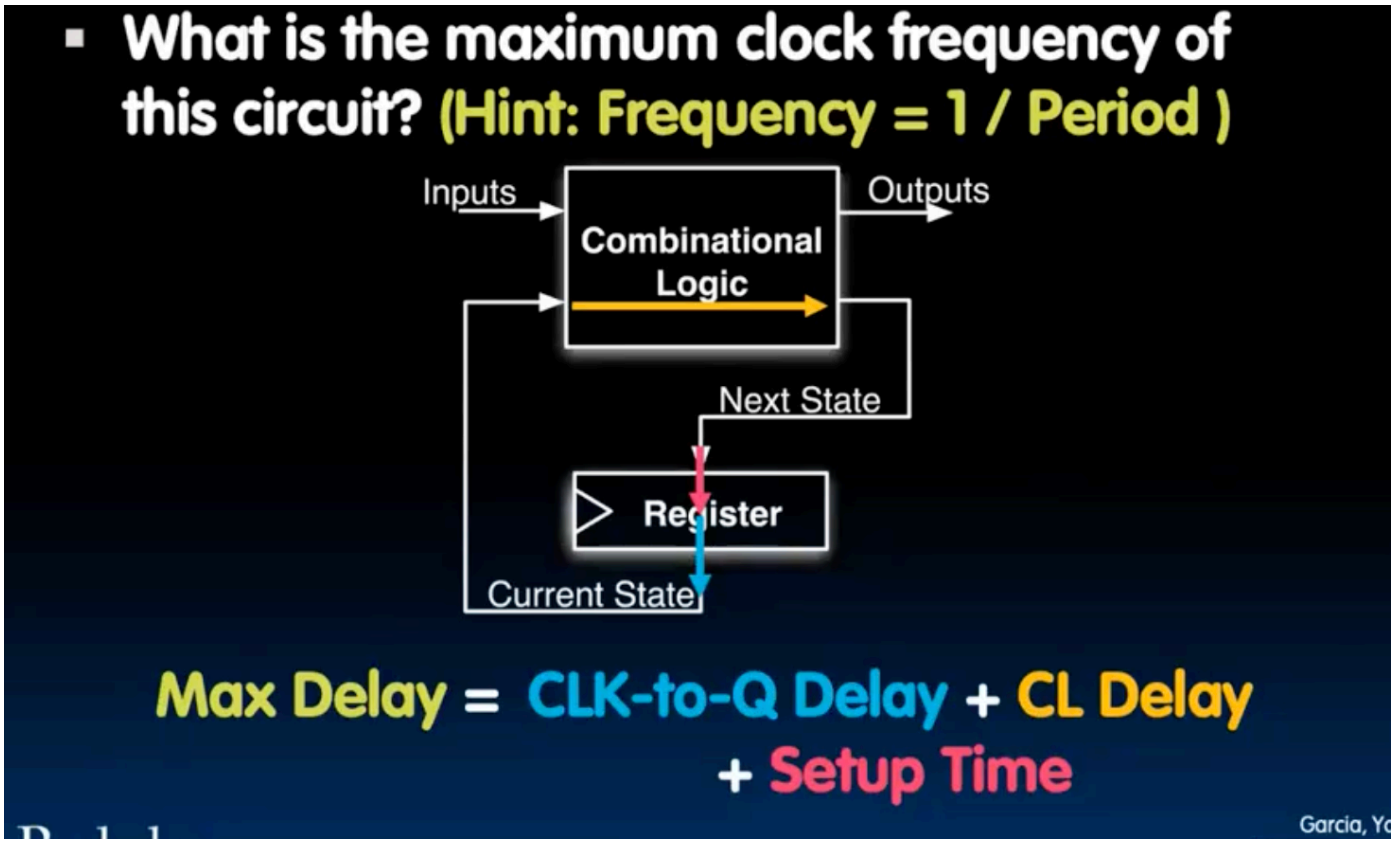
C-to-Q delay 是指：

- 时钟到来时，它会**采样输入 D**；
- 然后**经过一段硬件延迟**，把这个值传递到输出 Q；
- 这个延迟过程就是 **C-to-Q delay**。

Setup time 是指：

> 在时钟上升沿到来**之前**，输入数据 D 必须已经稳定并保持不变的最短时间。

> 否则，触发器可能采样到错误数据或进入不确定状态（metastability）。



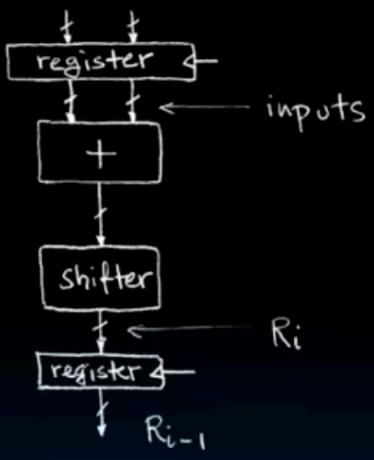
如果电路的周期小于最大延迟，会导致建立时间之前无法稳定，导致超频崩溃

使用流水线完成一件事情的延迟会增加

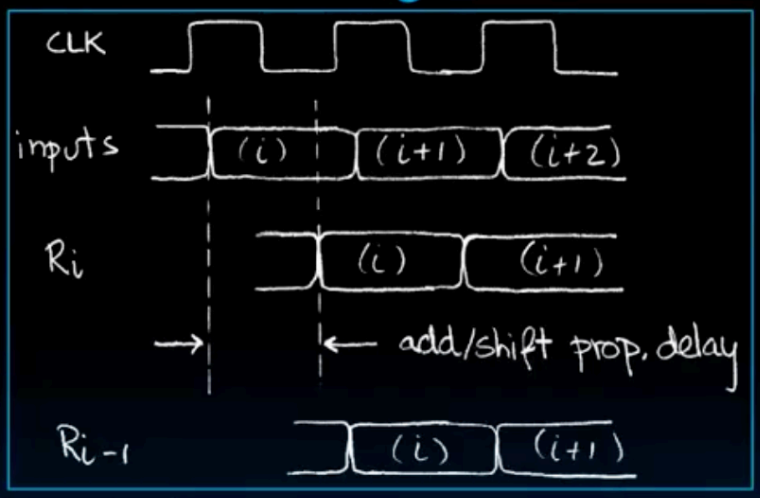
但是我们的效率更高，数据吞吐量会更大

> **流水线电路的关键路径长度 = 最长(某一级组合逻辑延迟 + setup 时间)**

- **Extra Registers** are often added to help speed up the clock rate.



Timing...

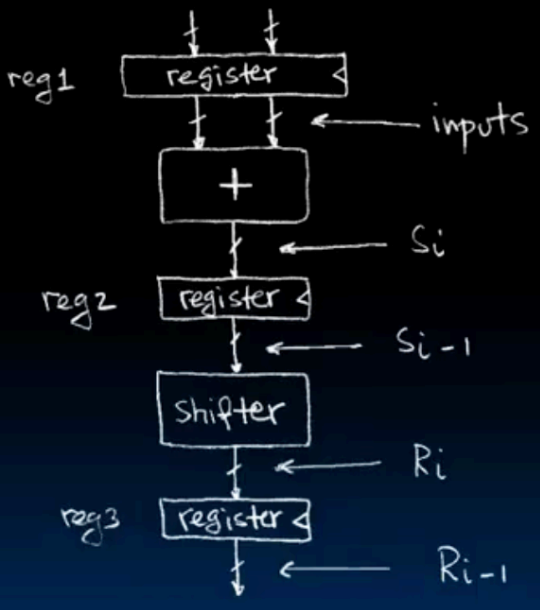


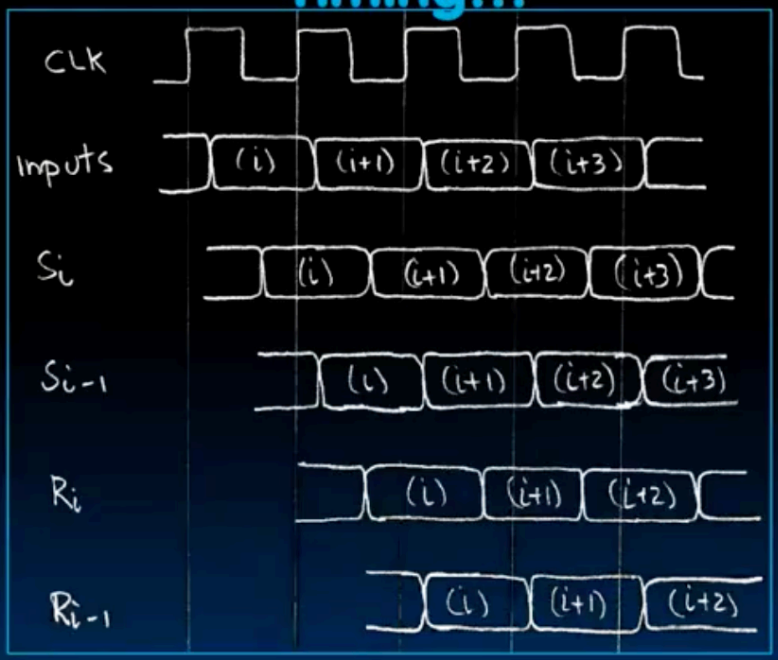
- Note: Delay of 1 clock cycle from input to output.
- Clock period limited by propagation delay of adder/shifter.

Garcia, Yo

- Insertion of register allows higher clock frequency.
- More outputs per second.

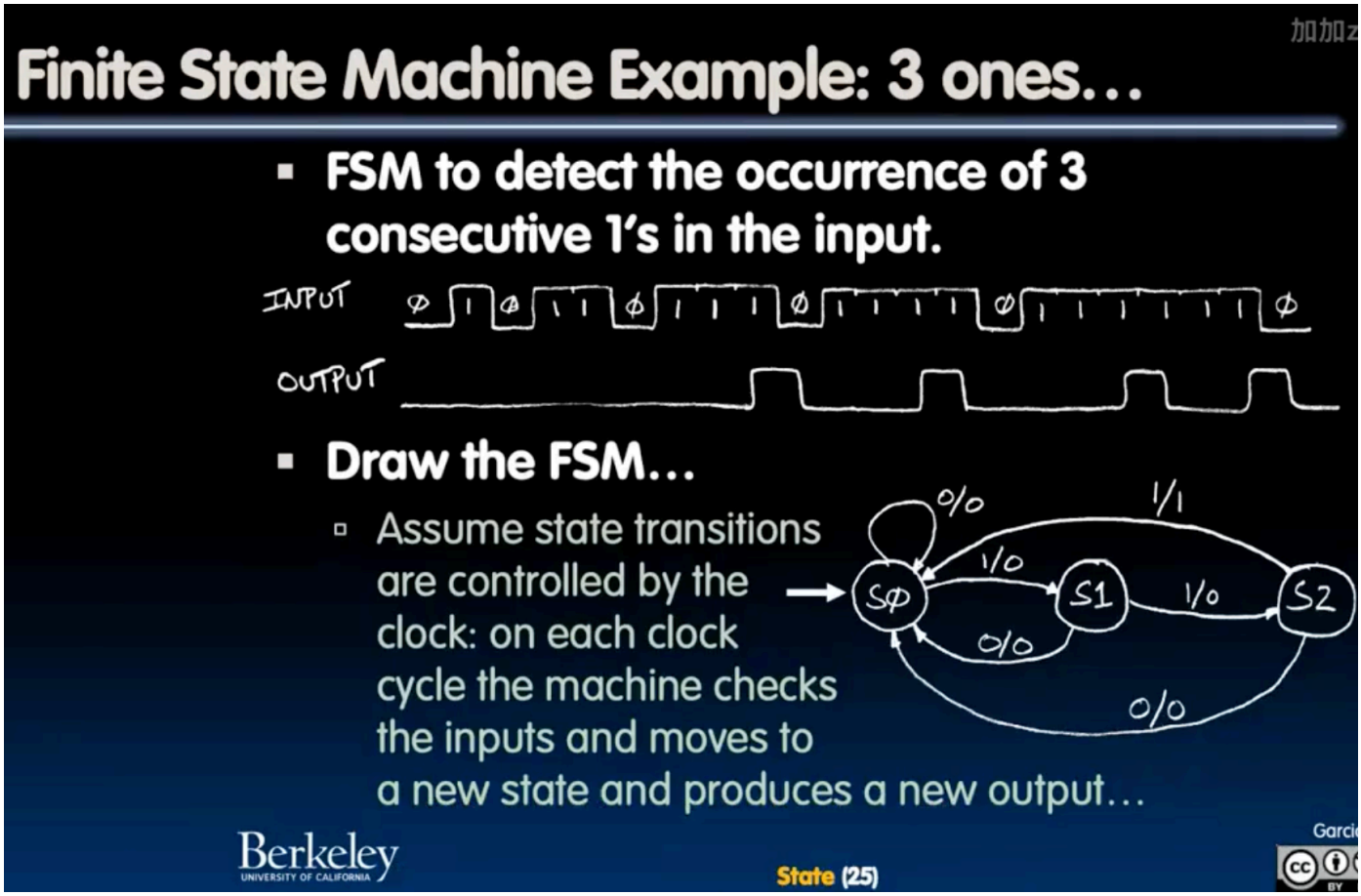
Timing...





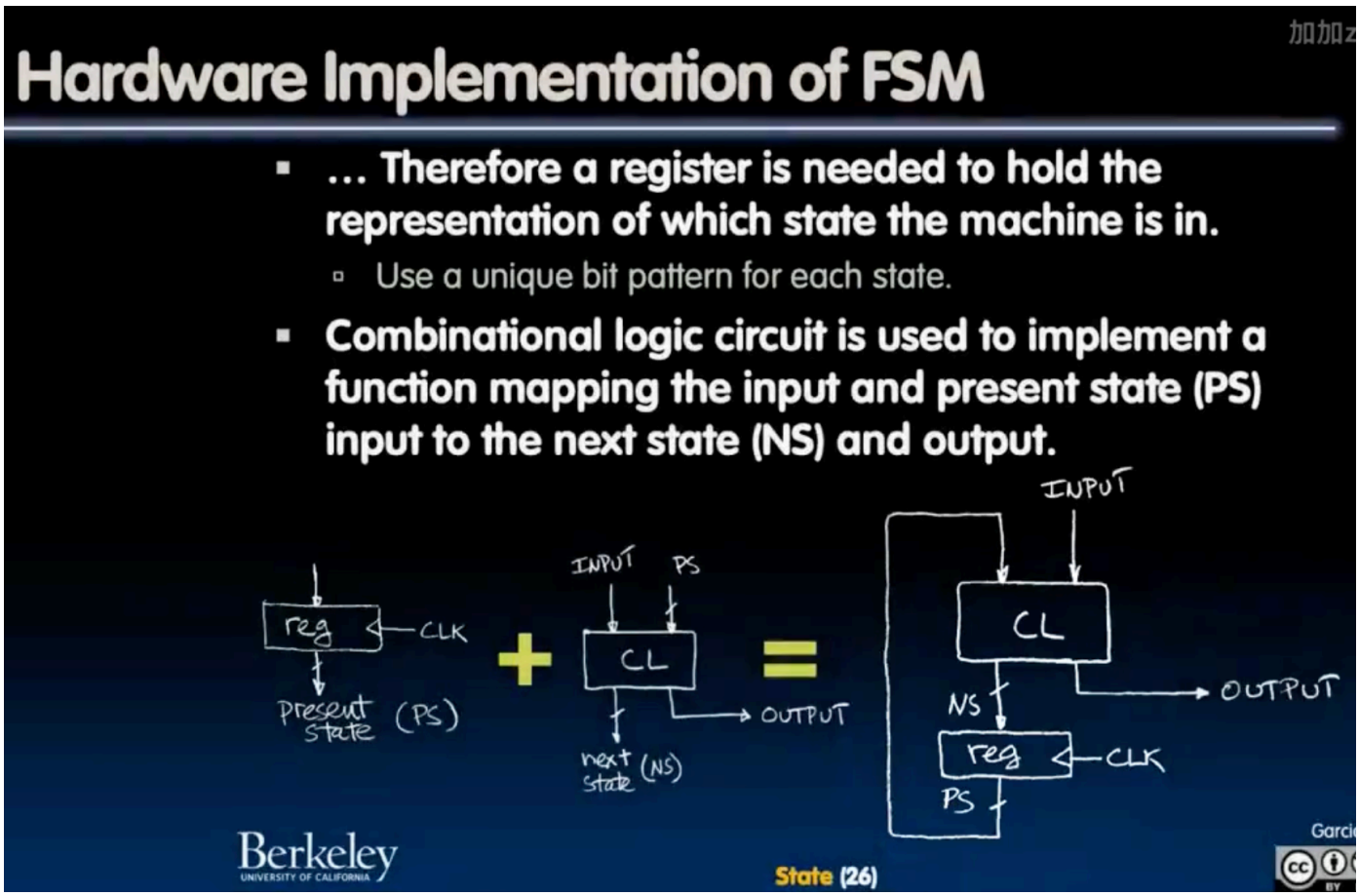
Garcia, Yo

Finite State Machines (FSM)



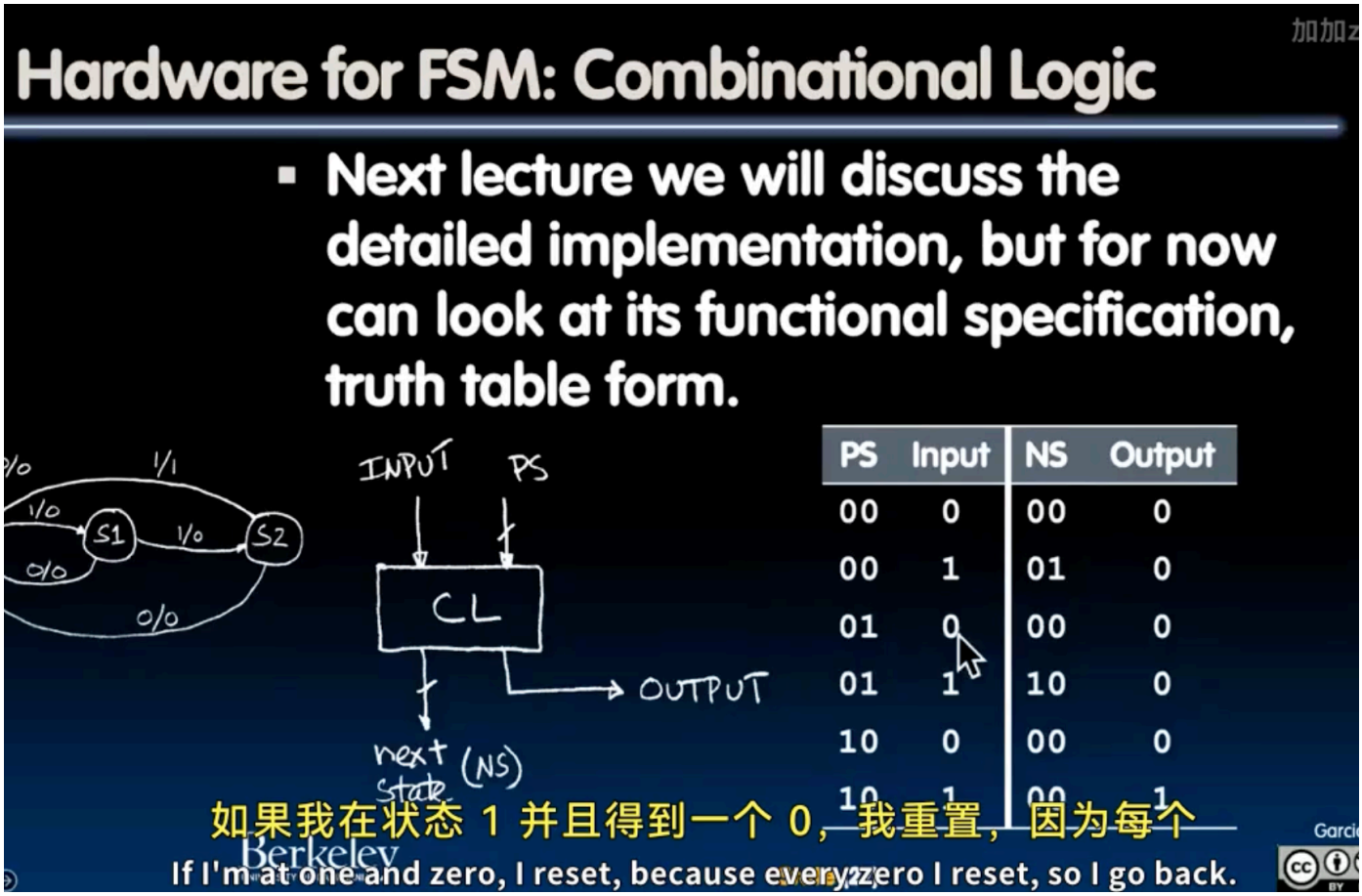
这台 FSM 就是一个能记住最近输入的 0/1情况的机器，当发现连续三个 1 时就发出一个 1 的信号

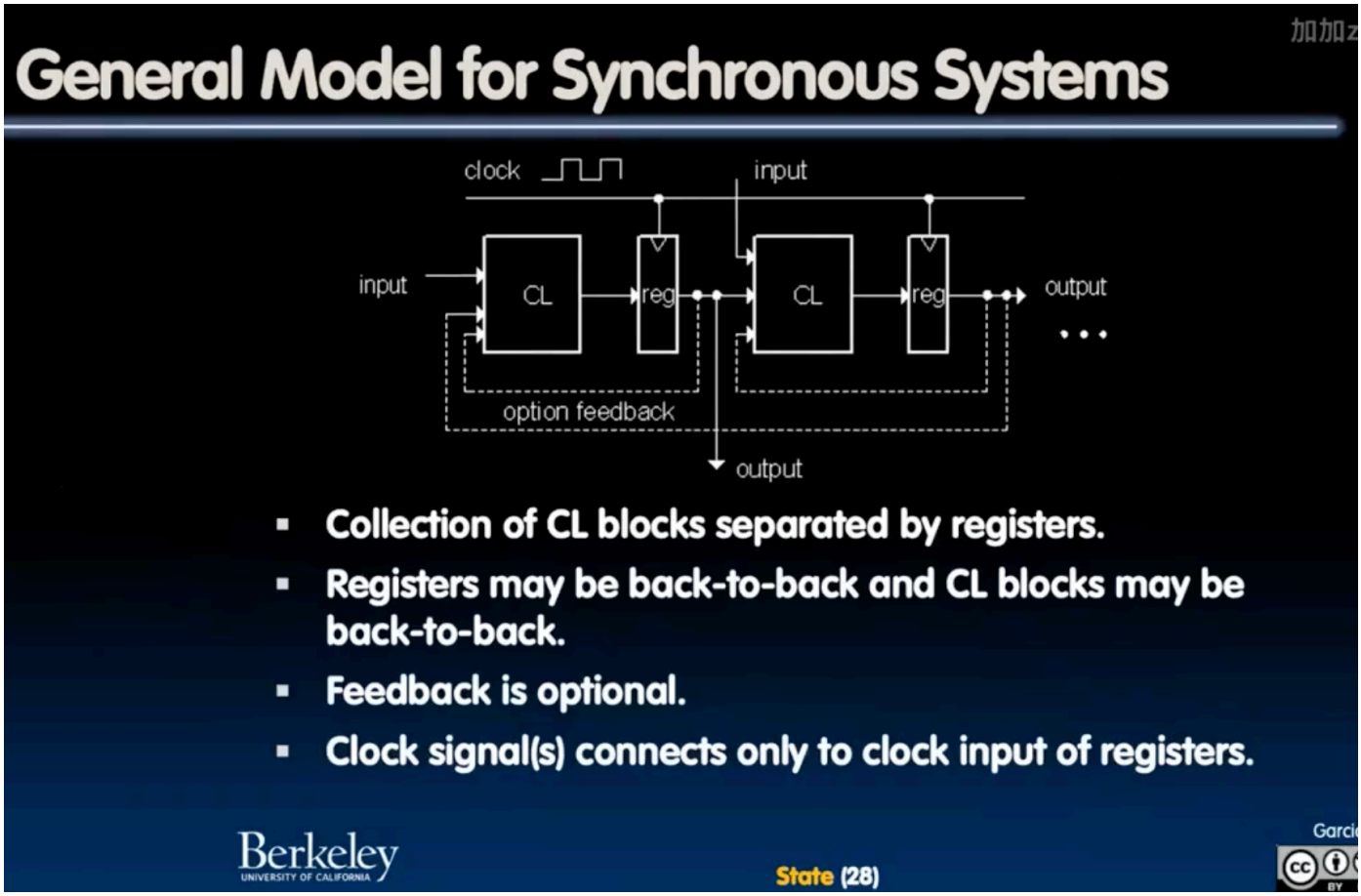
如何实现呢



根据**当前状态（PS）和输入（Input）**，决定**下一状态（NS）和输出（Output）**

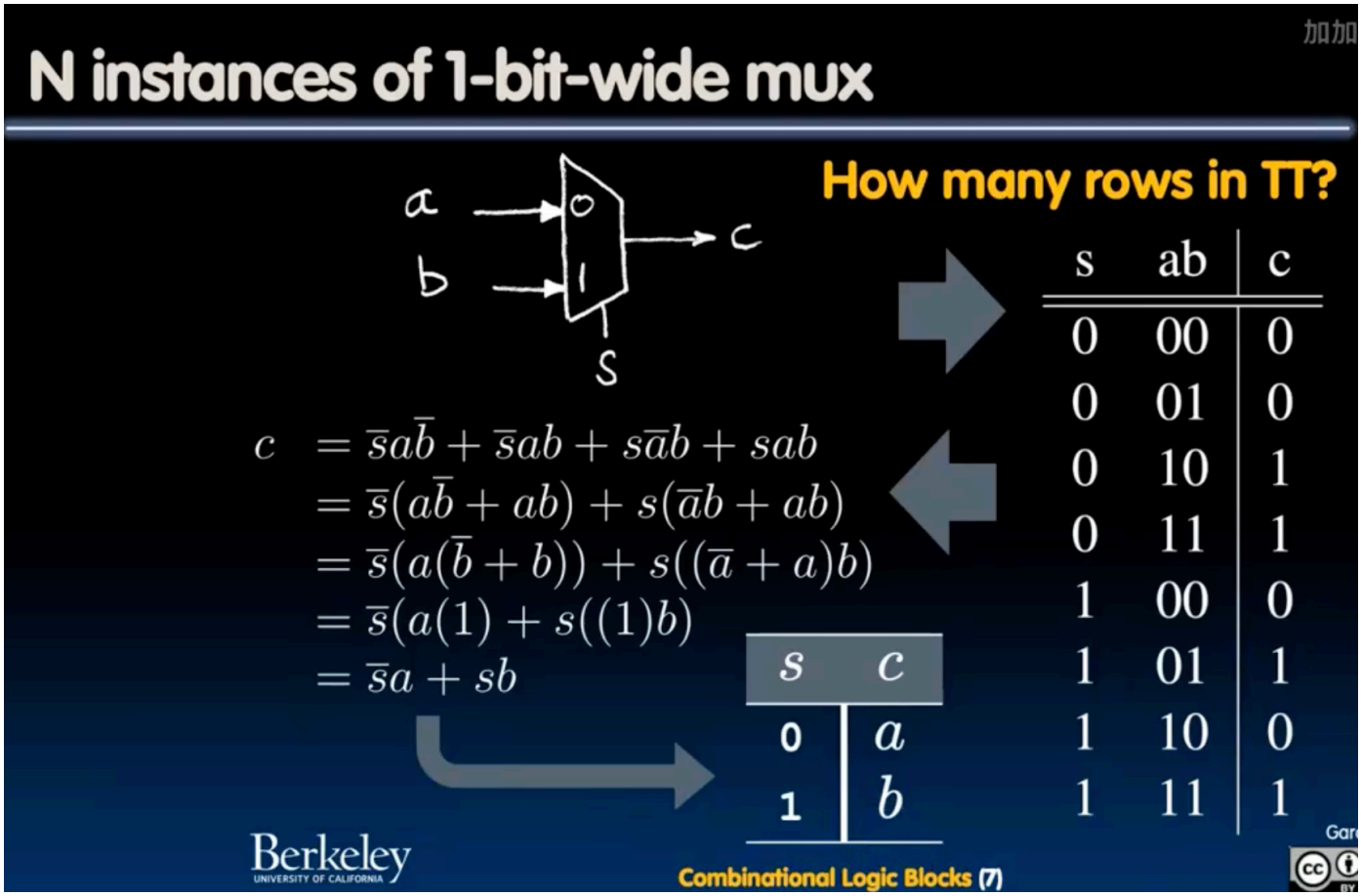
- **PS**（Present State）是用二进制表示的状态编号。
  - 00 代表 S0
  - 01 代表 S1
  - 10 代表 S2
- **Input** 是每一时钟周期采样到的输入（0或1）。
- **NS**（Next State）是根据PS和Input决定的跳转目标状态。
- **Output** 就是根据现在的情况输出0或1。

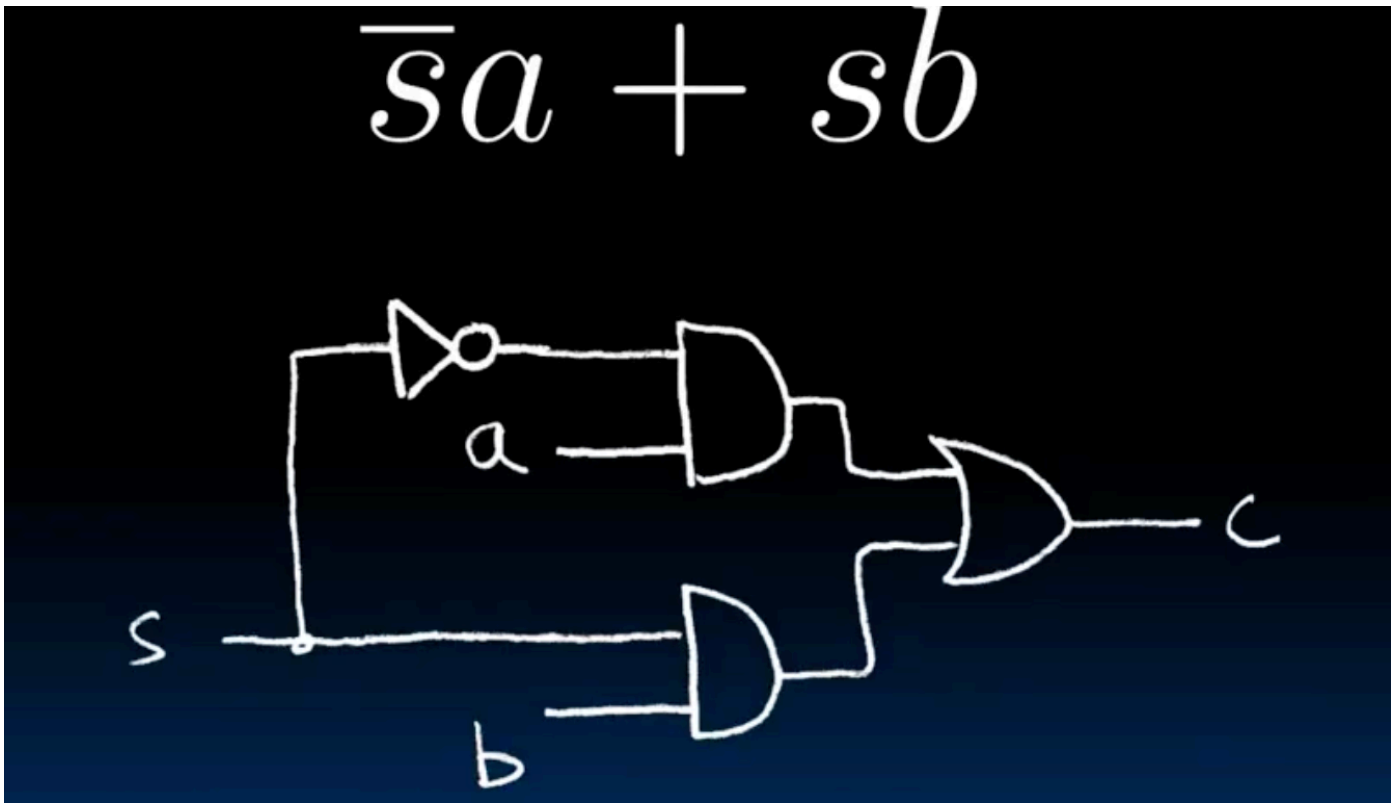




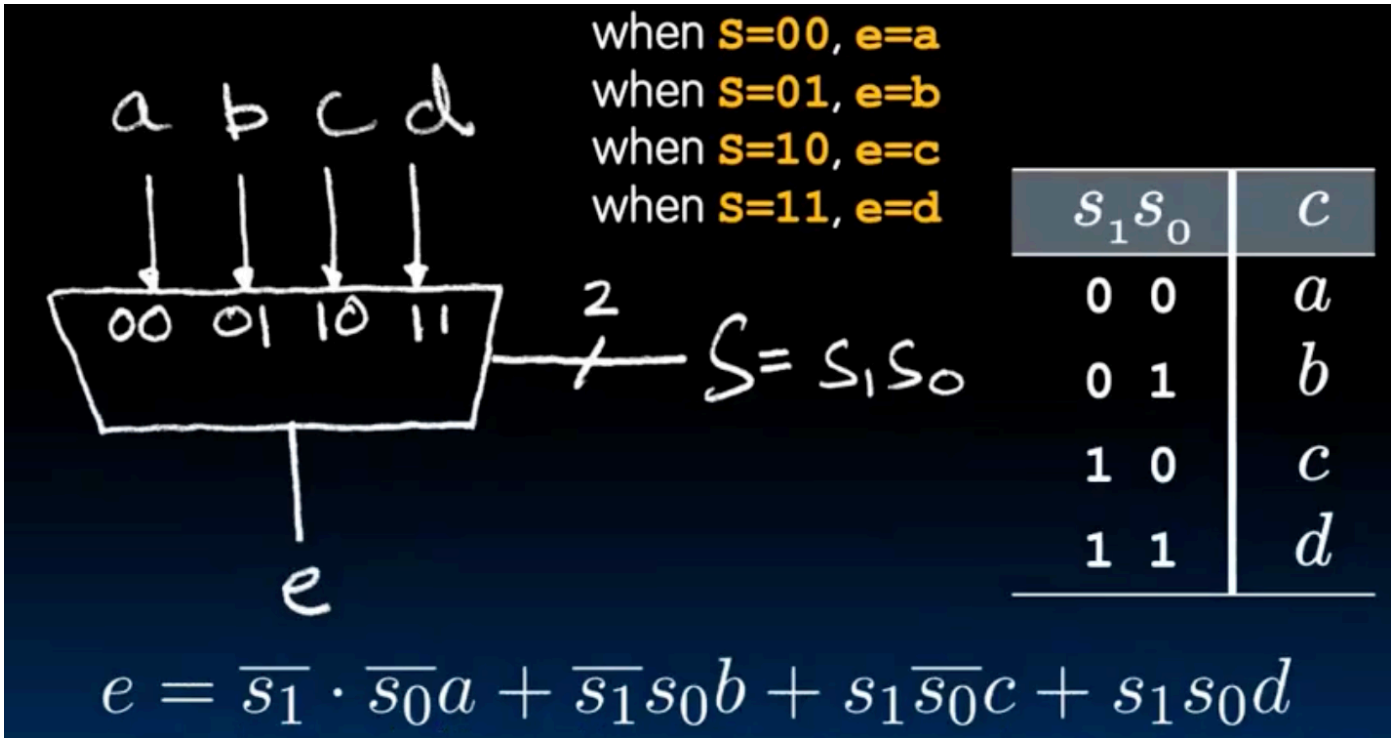
Lecture 17 : Combinational Logic Blocks

Data Multiplexors 多路复用器

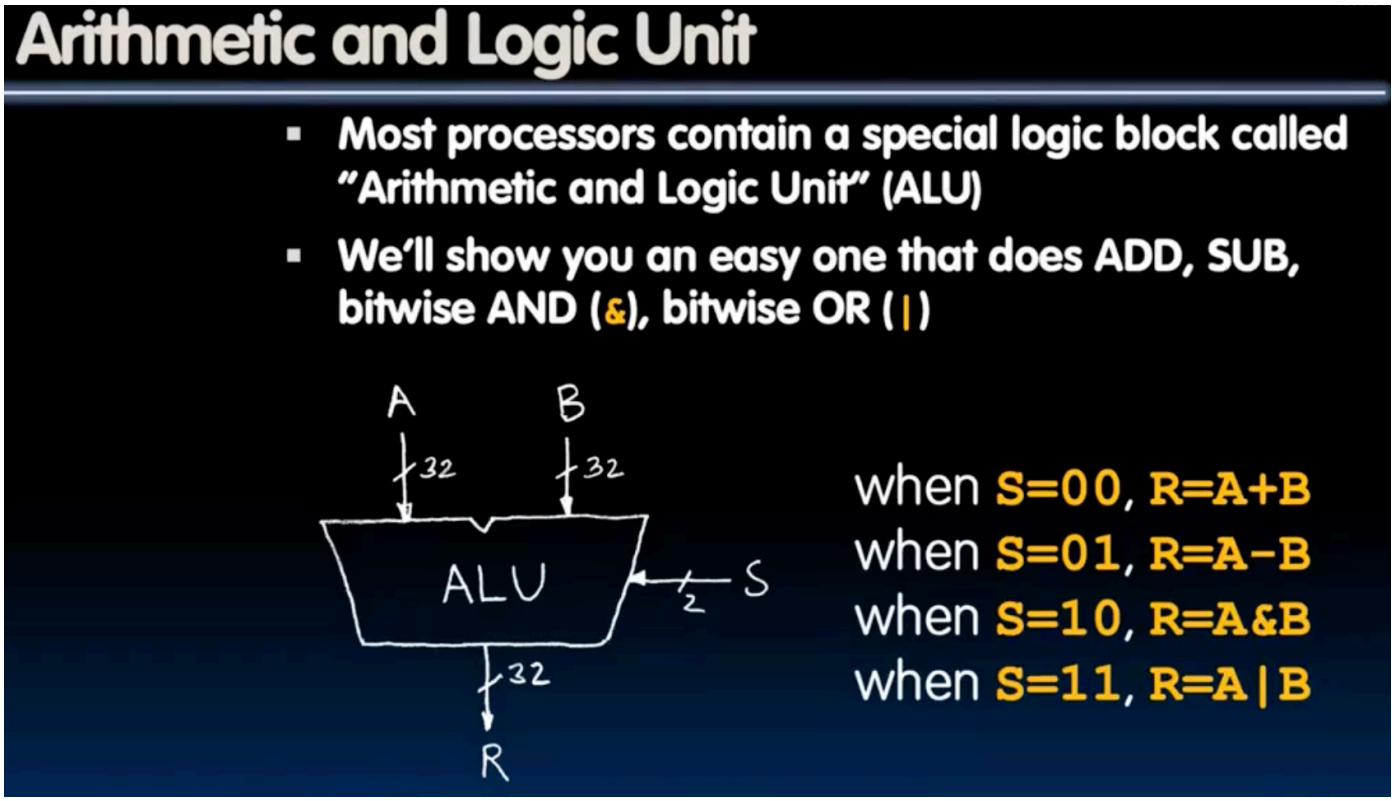

加加
N instances of 1-bit-wide mux
a
b
0
1
c
s
How many rows in TT?
s ab c
0 00 0
0 01 0
0 10 1
0 11 1
1 00 0
1 01 1
1 10 0
1 11 1
$c = \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab$
$= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab)$
$= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b)$
$= \bar{s}(a(1) + s((1)b)$
$= \bar{s}a + sb$
s c
0 a
1 b
Berkeley
UNIVERSITY OF CALIFORNIA
Combinational Logic Blocks (7)


$\bar{s}a + sb$
a
s
b
c

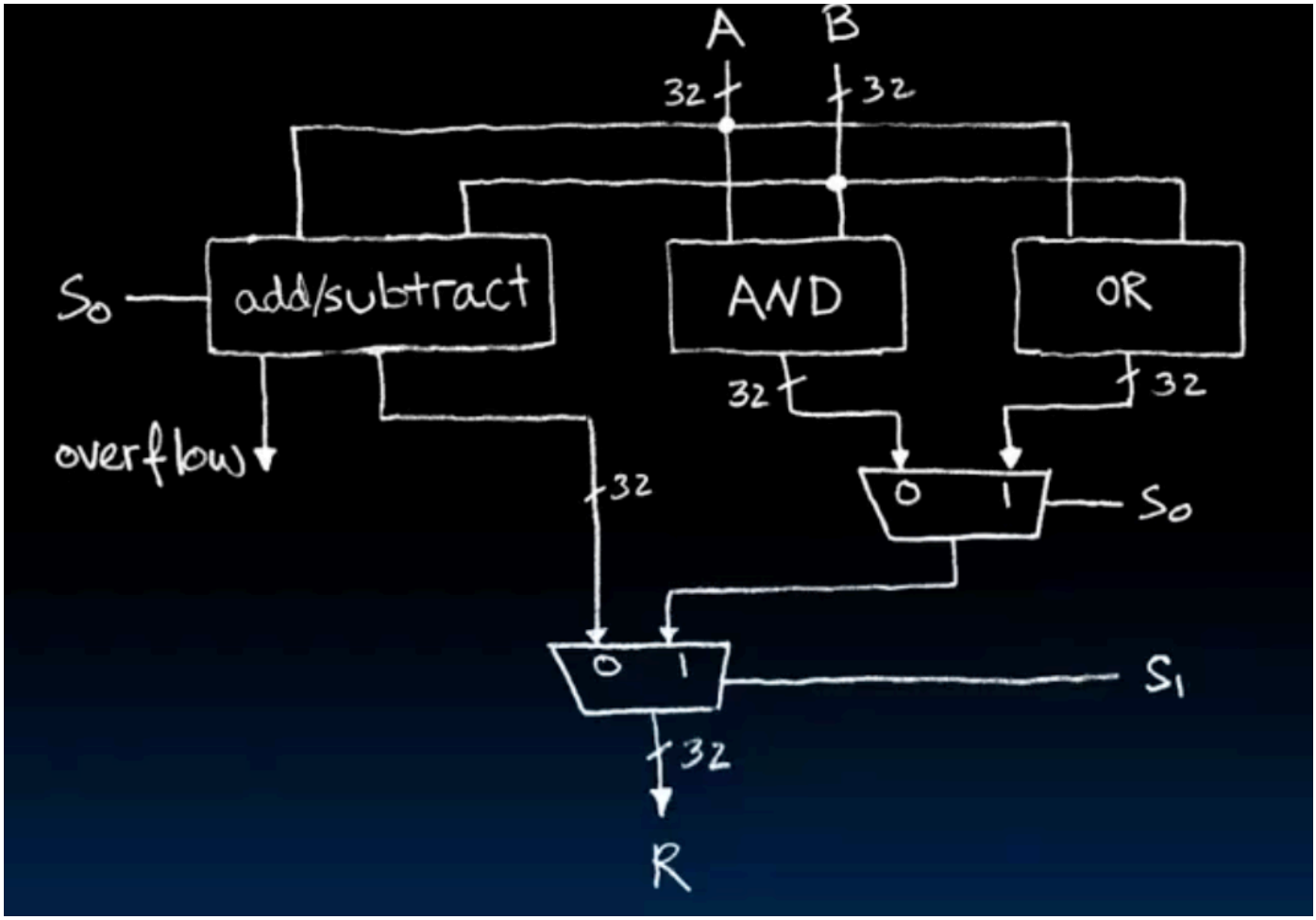
再次转化形式



Arithmetic Logic Unit (ALU)



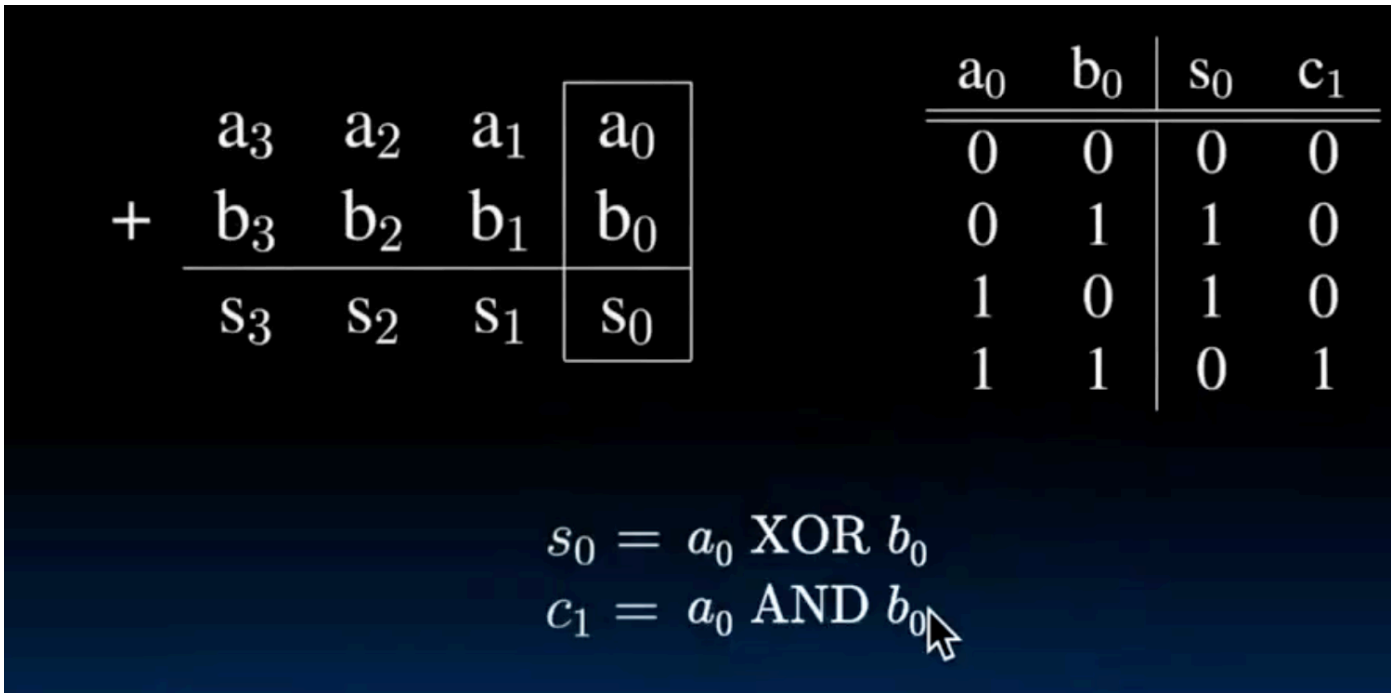
所有的组件都在工作，只不过我们只选取了某个结果：



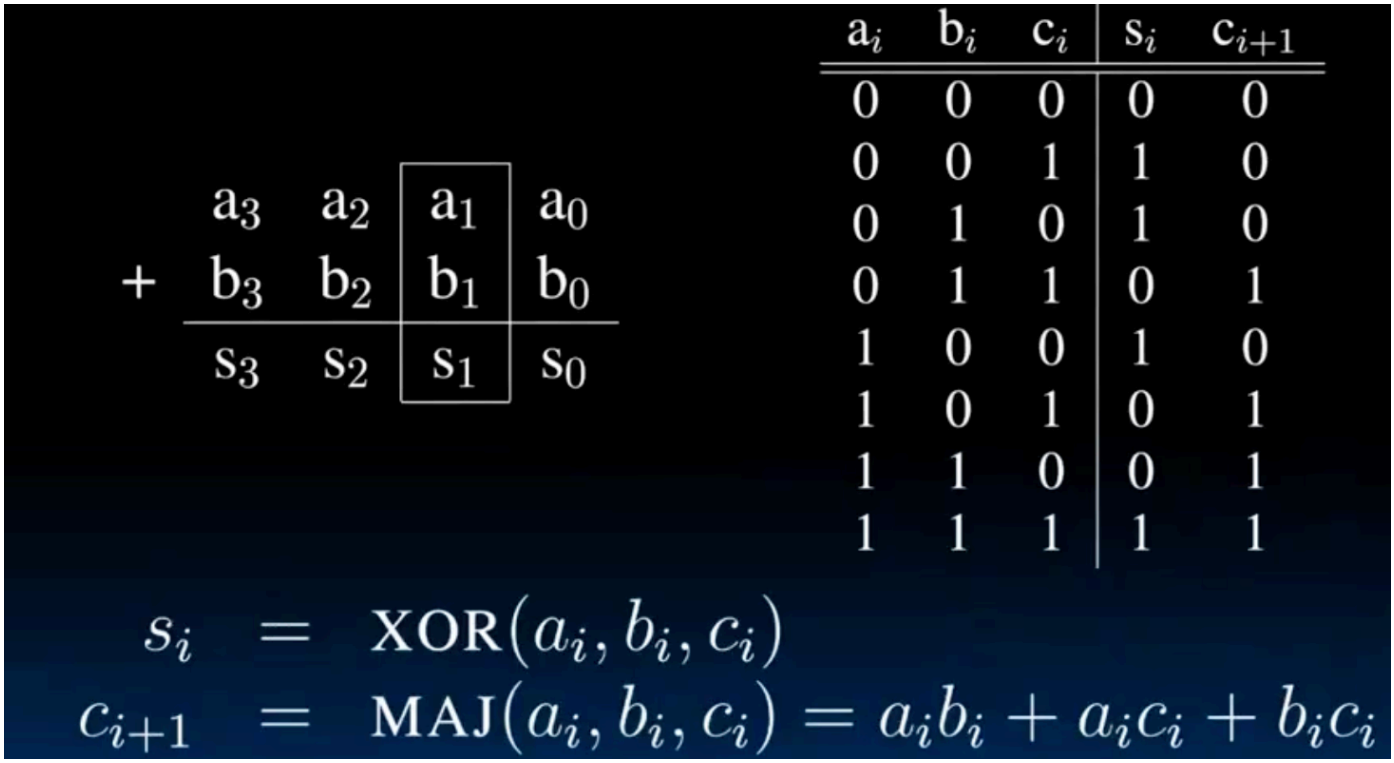
Adder / Subtractor

从1bit开始

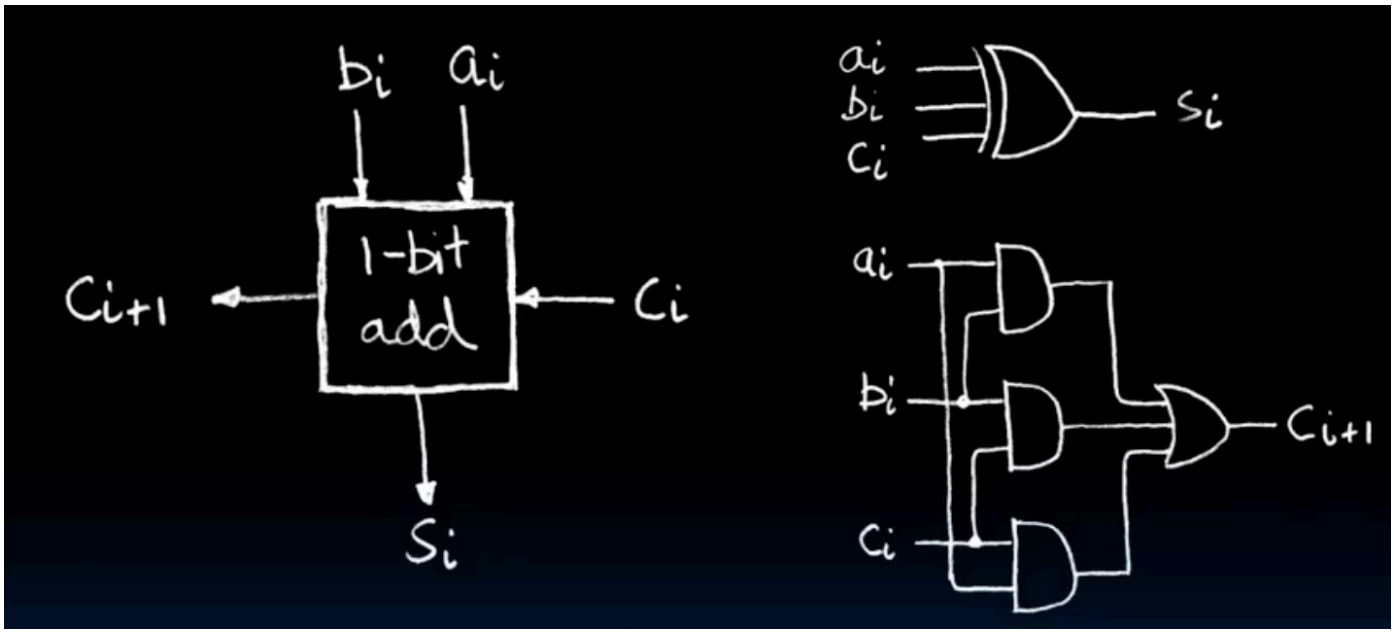
s_0是低位，c_1是进位



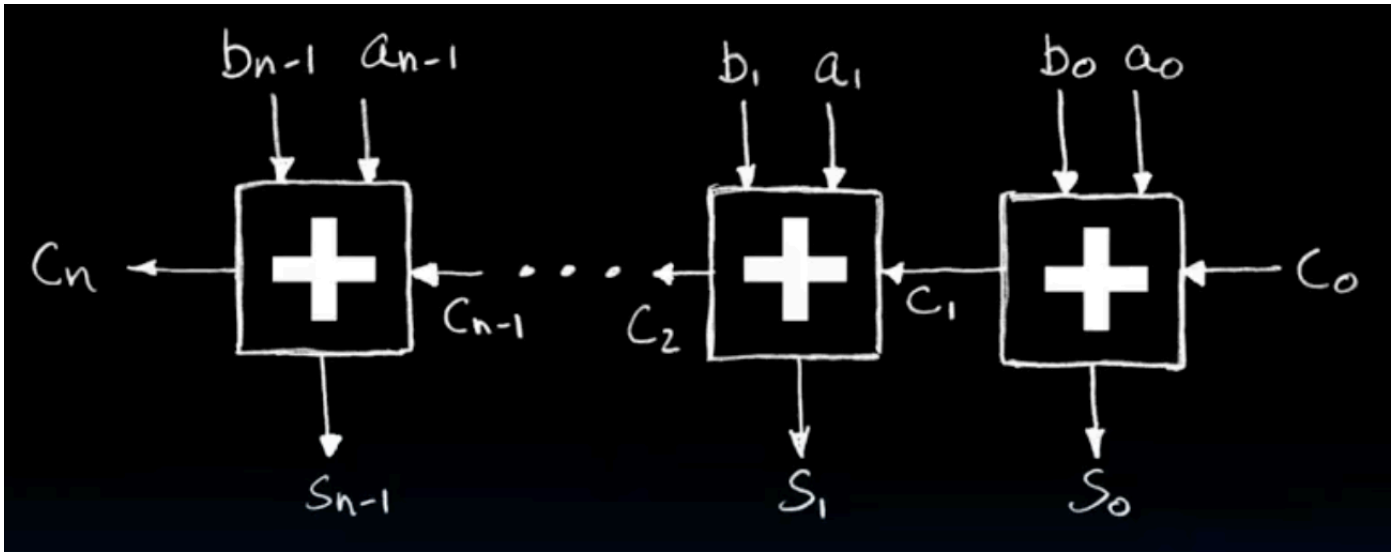
`XOR` 1的个数是奇数，结果是1



我们继续将它转化为逻辑门



接着我们将N个1bit相加器转化为1个N位相加器



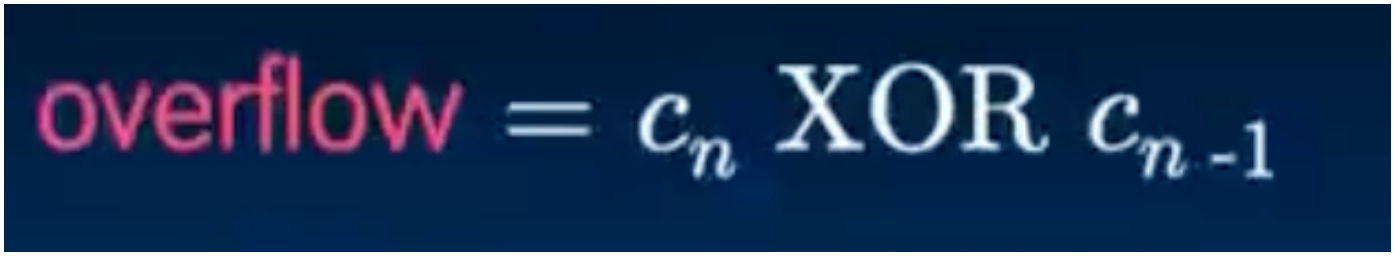
overflow?

在无符号数的加法中，我们需要考虑溢出问题

但是如果是在二进制补码中

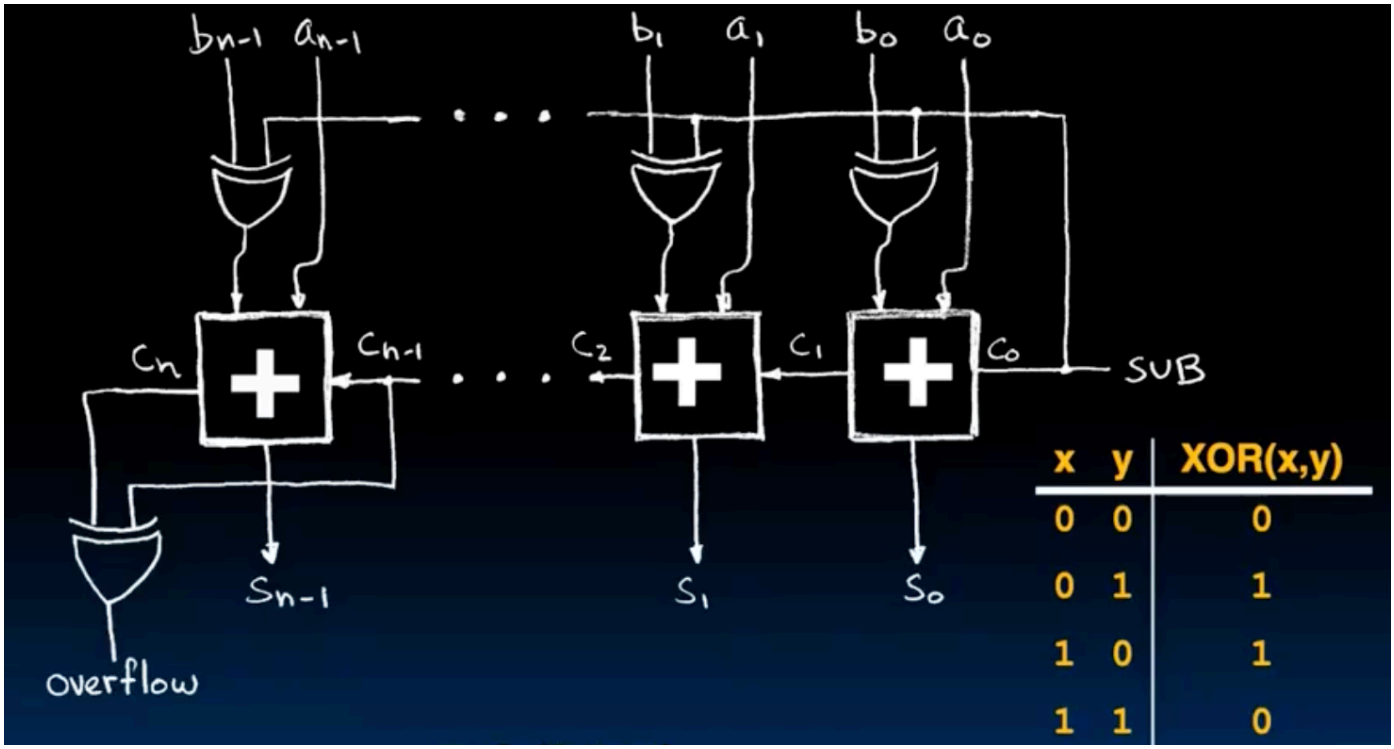
我们看最高位的adder

- 如果没有进位输出或者进位输入的话 不会溢出
- 如果既有进位输入又有进位输出的话 不会溢出
- 如果有进位输入或者输出中的一个 就会溢出



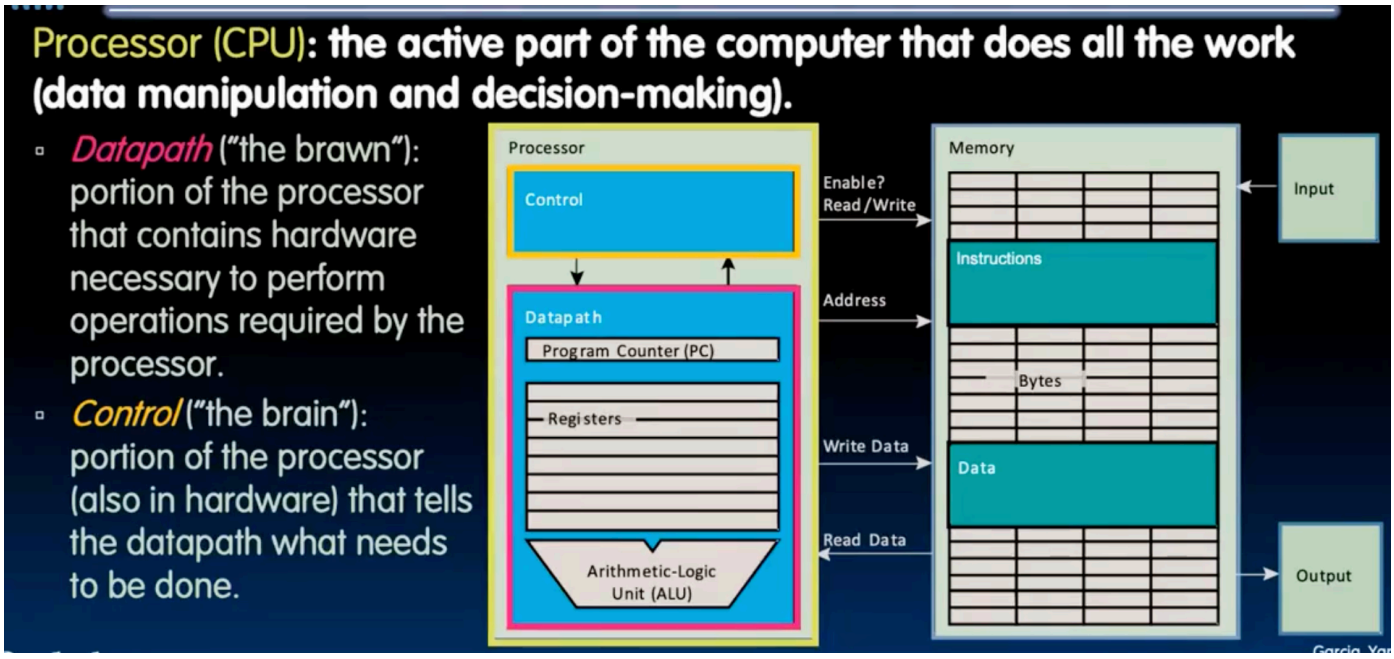
Subtractor

我们使用sub线，但它有高电平时，B的位就会反转，同时我们sub线还参与了计算，相当于+1，这样的话，我们可以得到减法操作

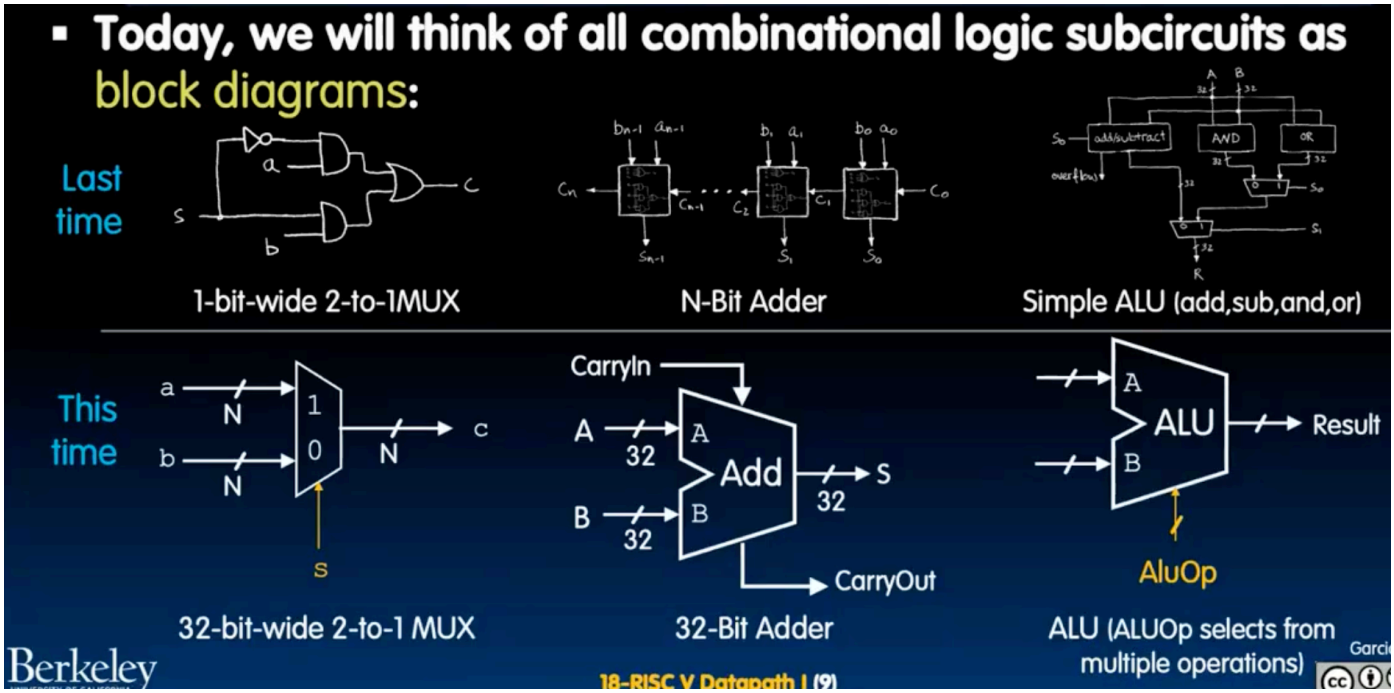


Lecture 18,19 : Datapath

Building a RISC-V Processor

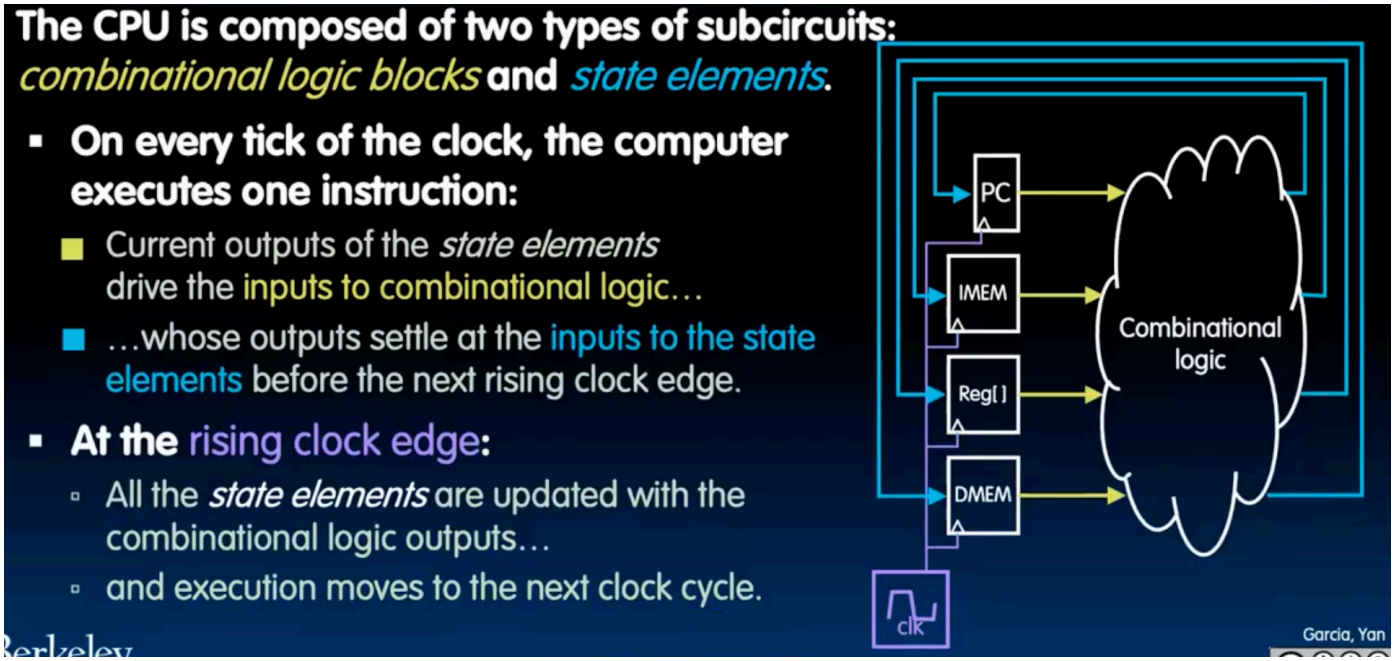


CPU Elements and Stages



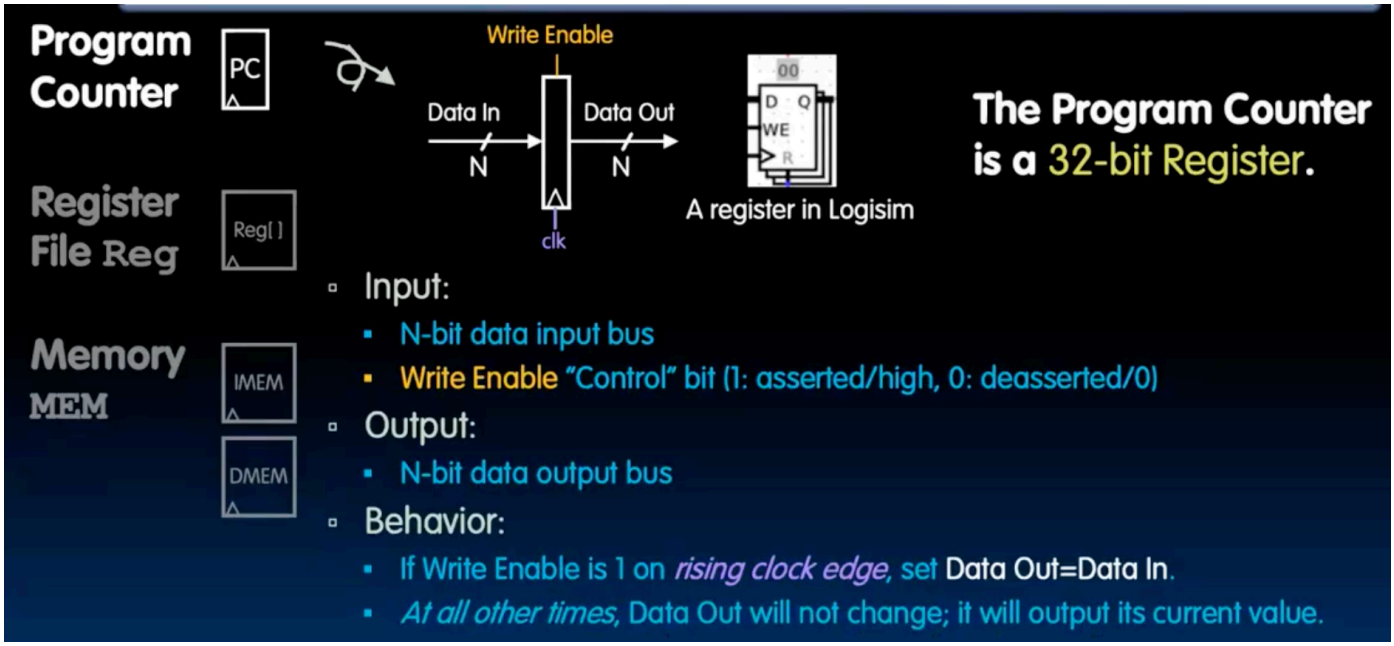
The CPU is composed of rwo types of subcircuits:

Comnination logic locks and *state elements*

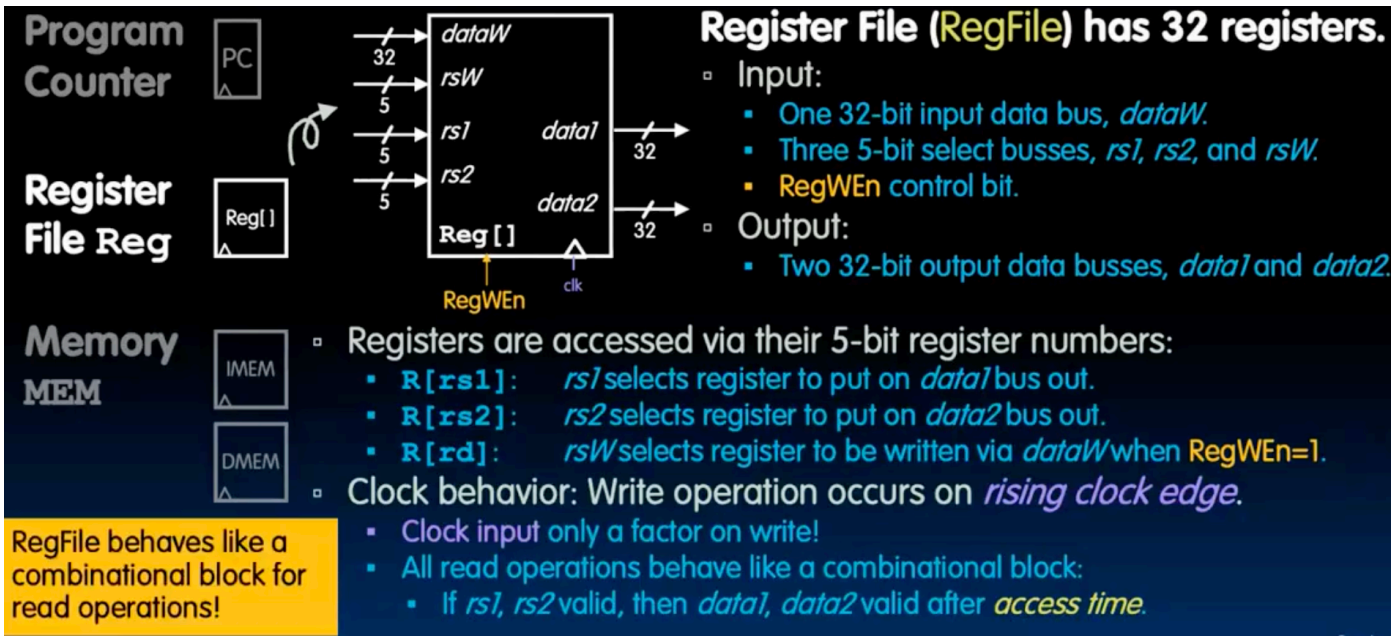


状态元素

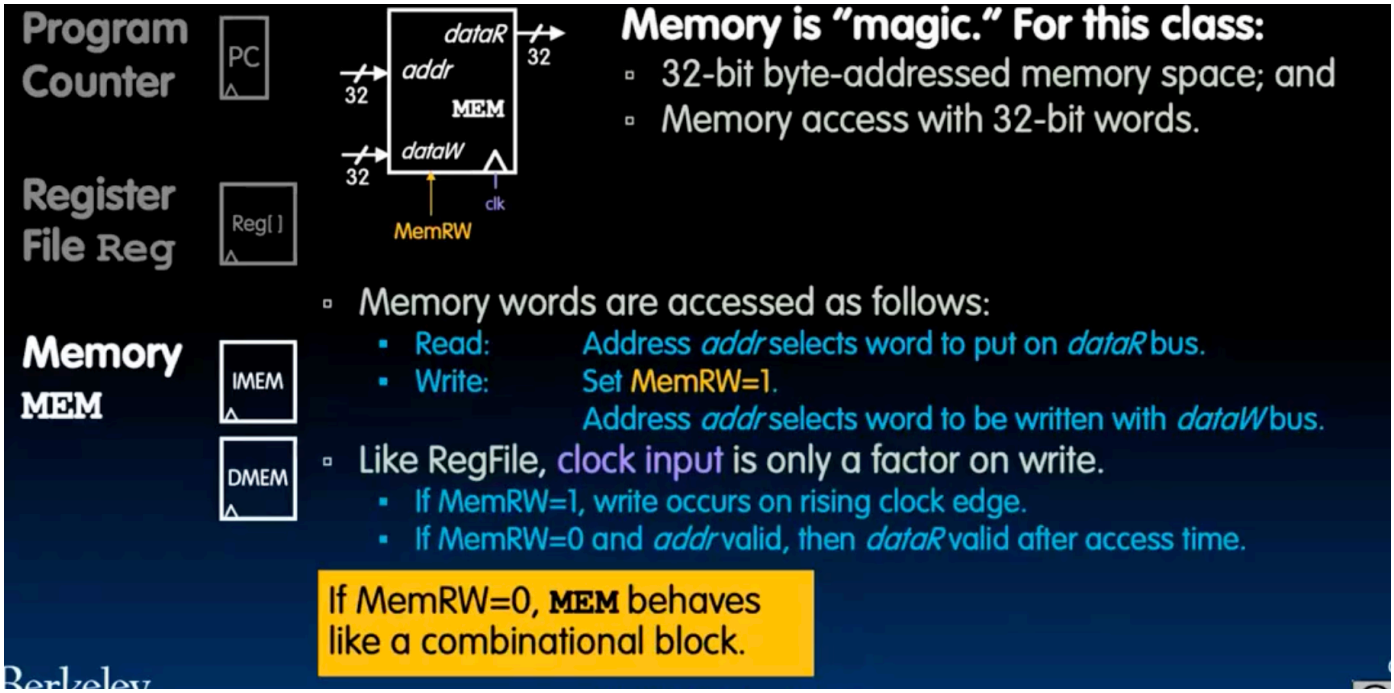
PC



Register File `Reg` 寄存器堆

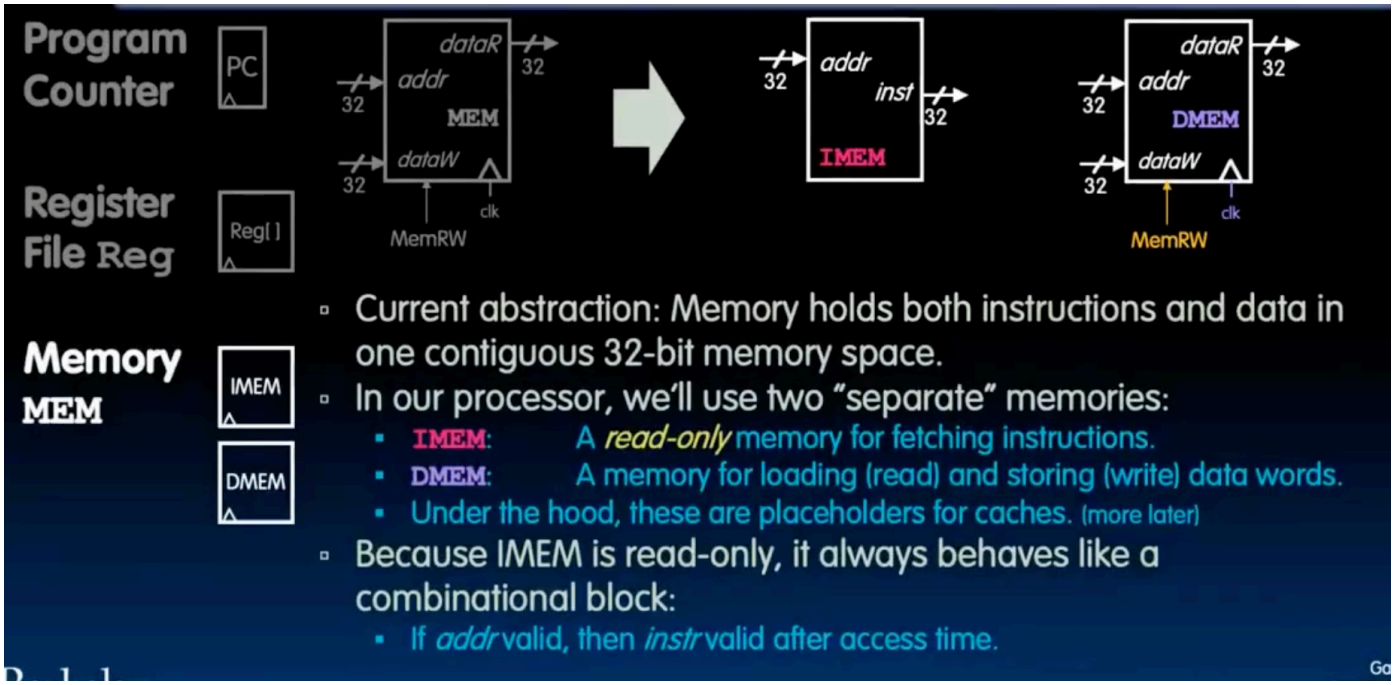


Memory `MEM`



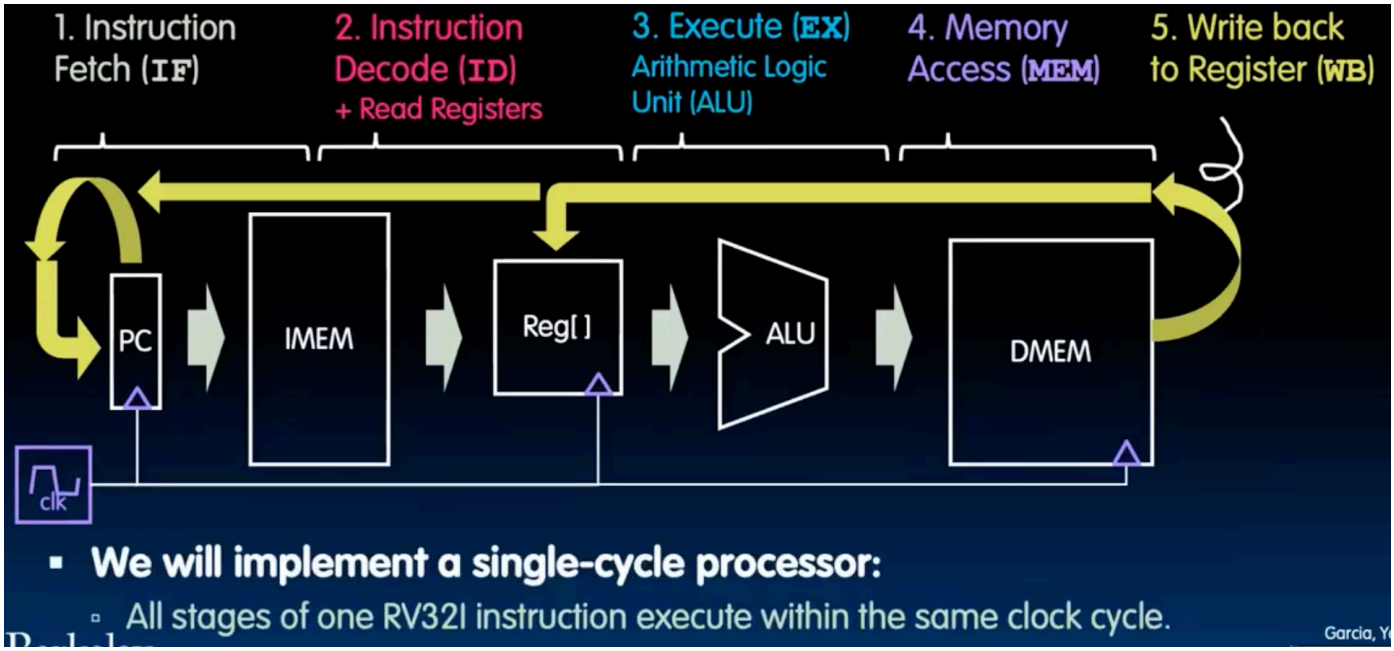
`IMEM` 代表 instruction memory 只读

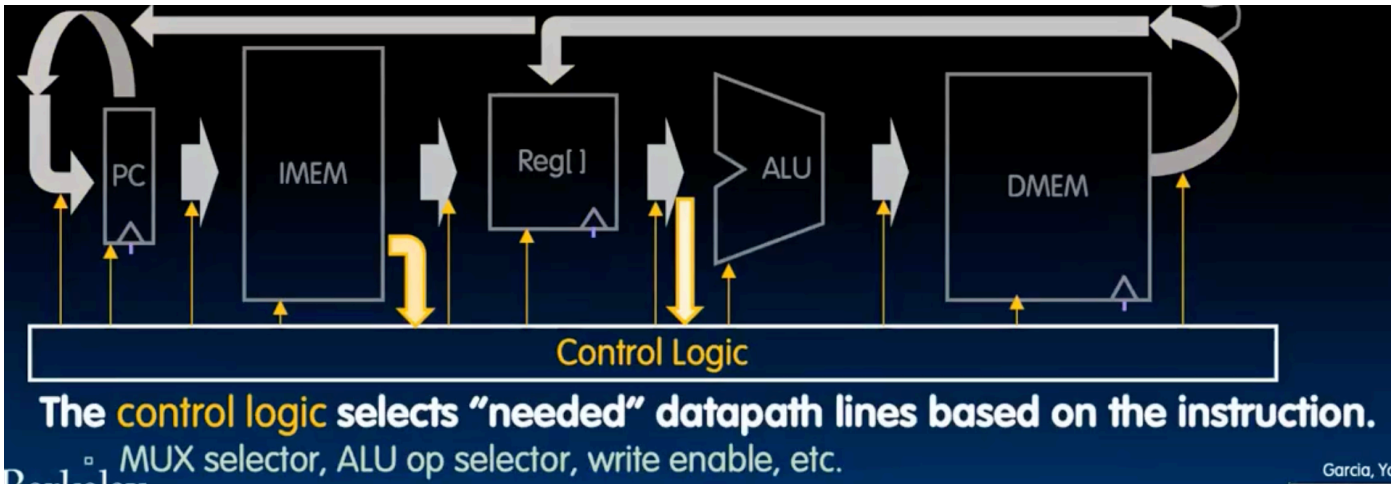
`DMEM` 代表data memory 支持读写



程序执行的5个基本阶段

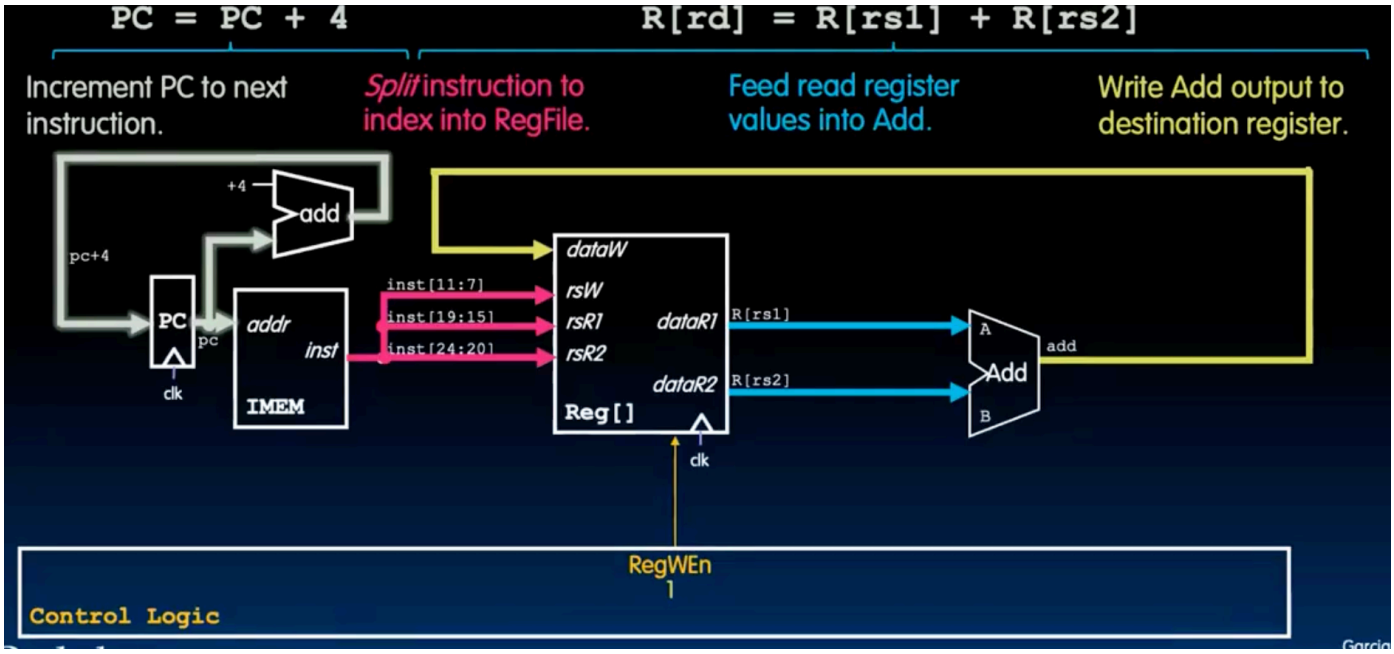
单周期处理器 -- 在一个时钟周期里完成指令





Datapath for `add`

这里我们先省略`DMEM`部分

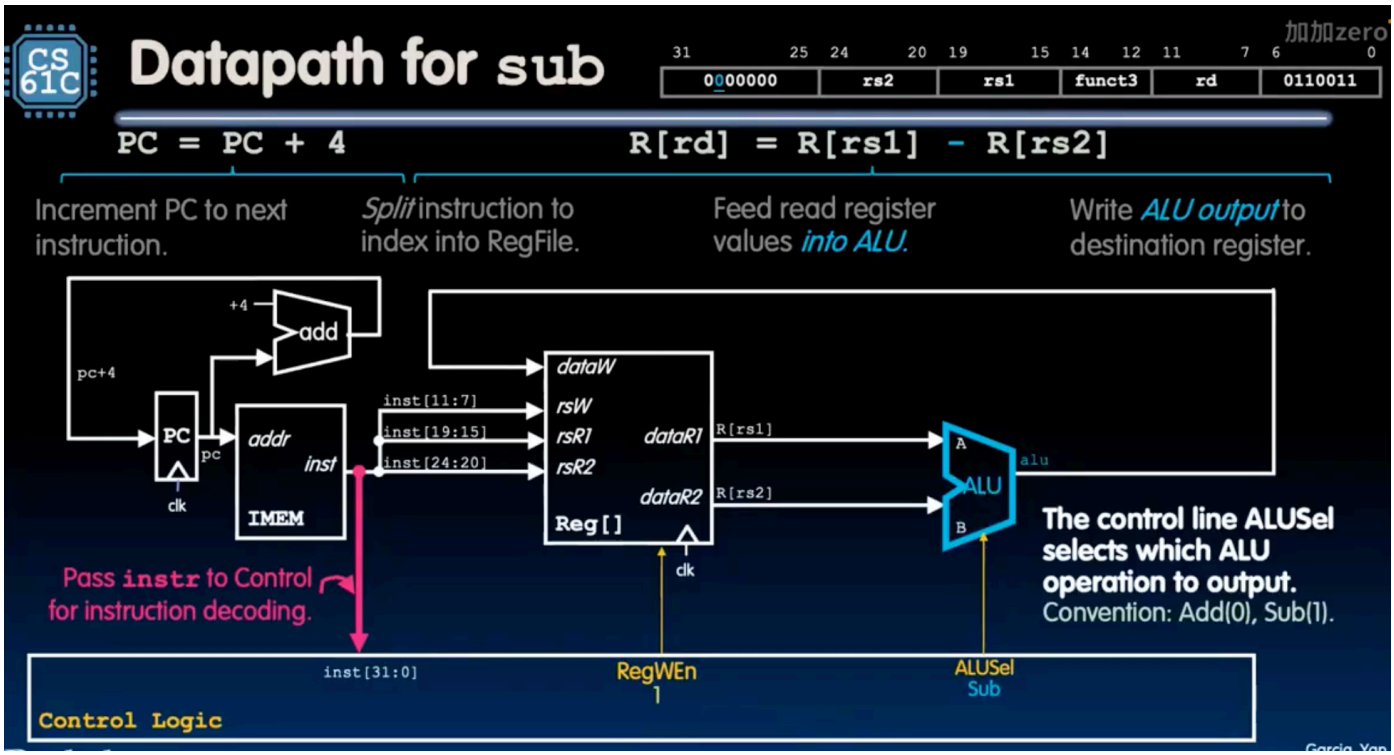


Datapath for `sub`

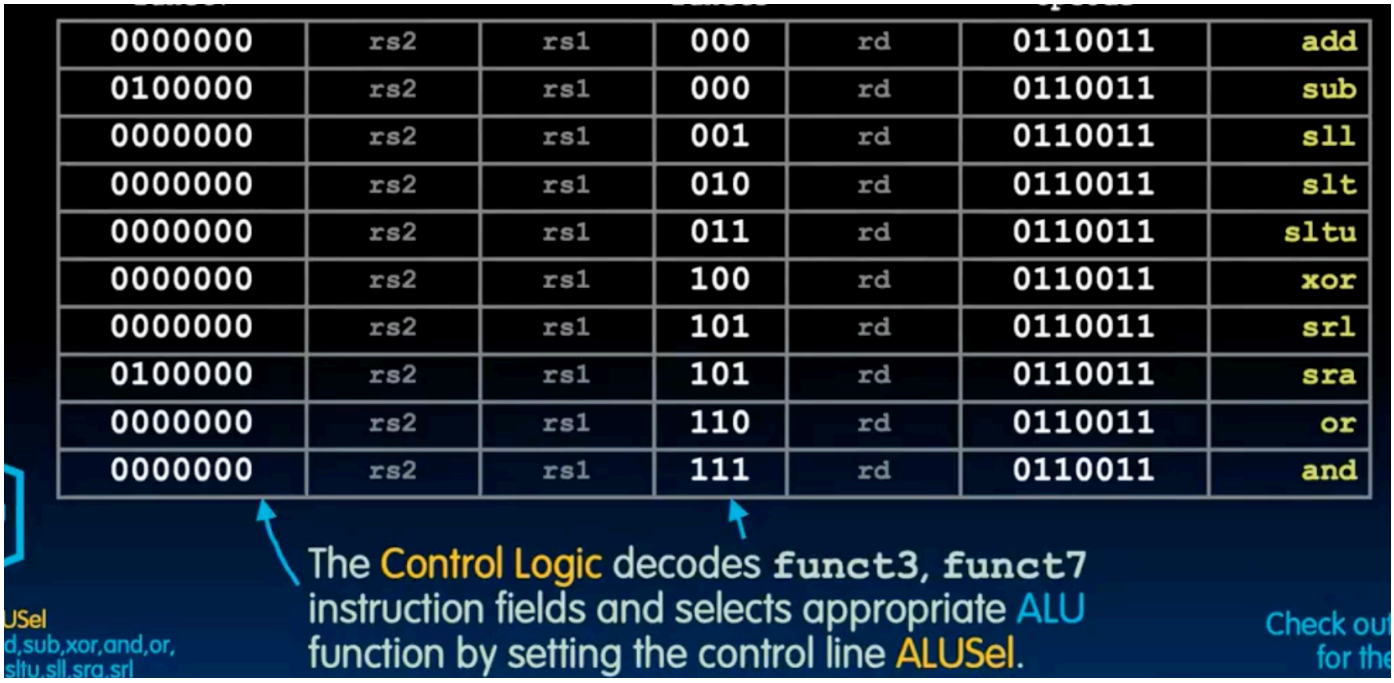
`sub`操作与`add`极为相似

指令的第30位决定了到底是加法操作还是减法操作

funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

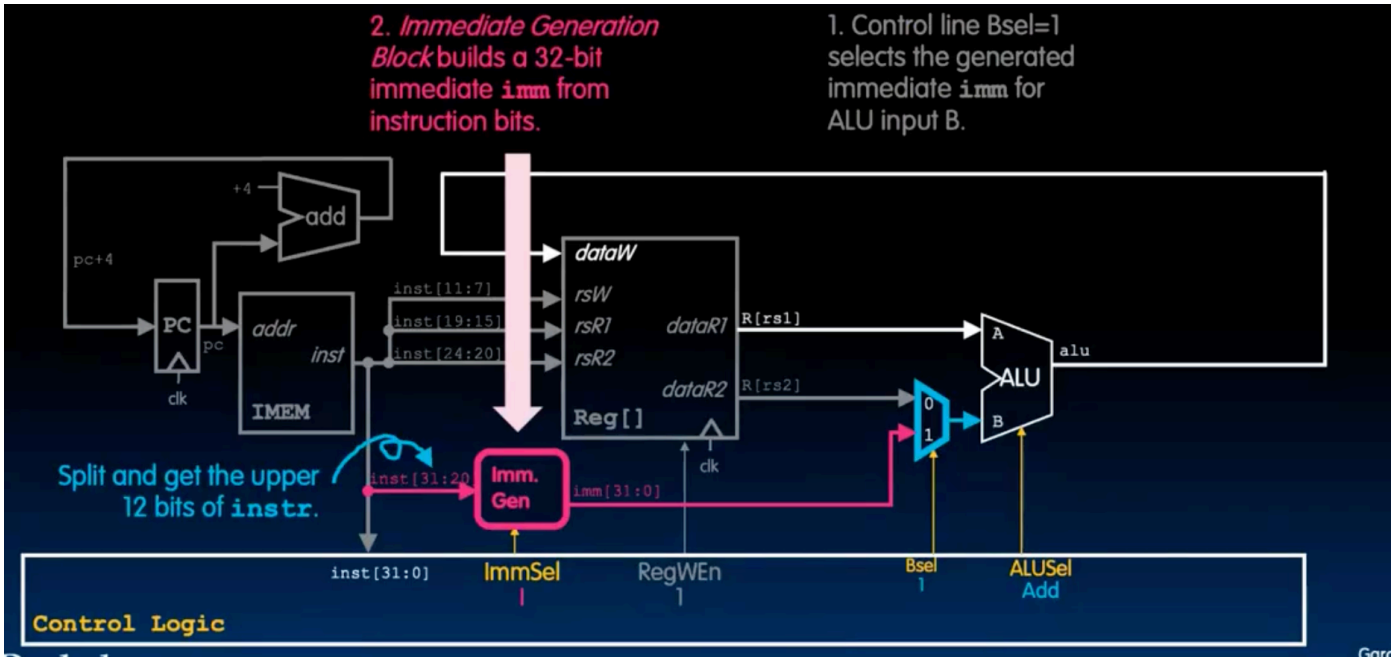


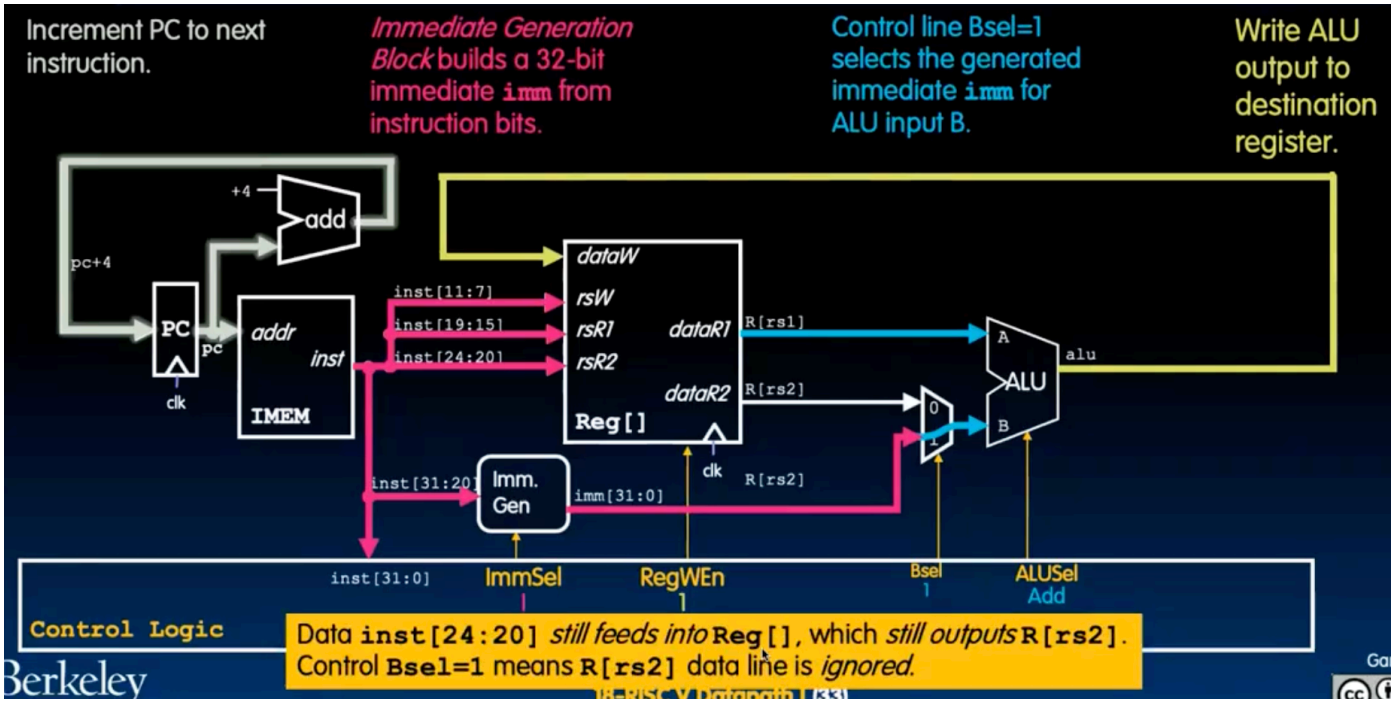
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and



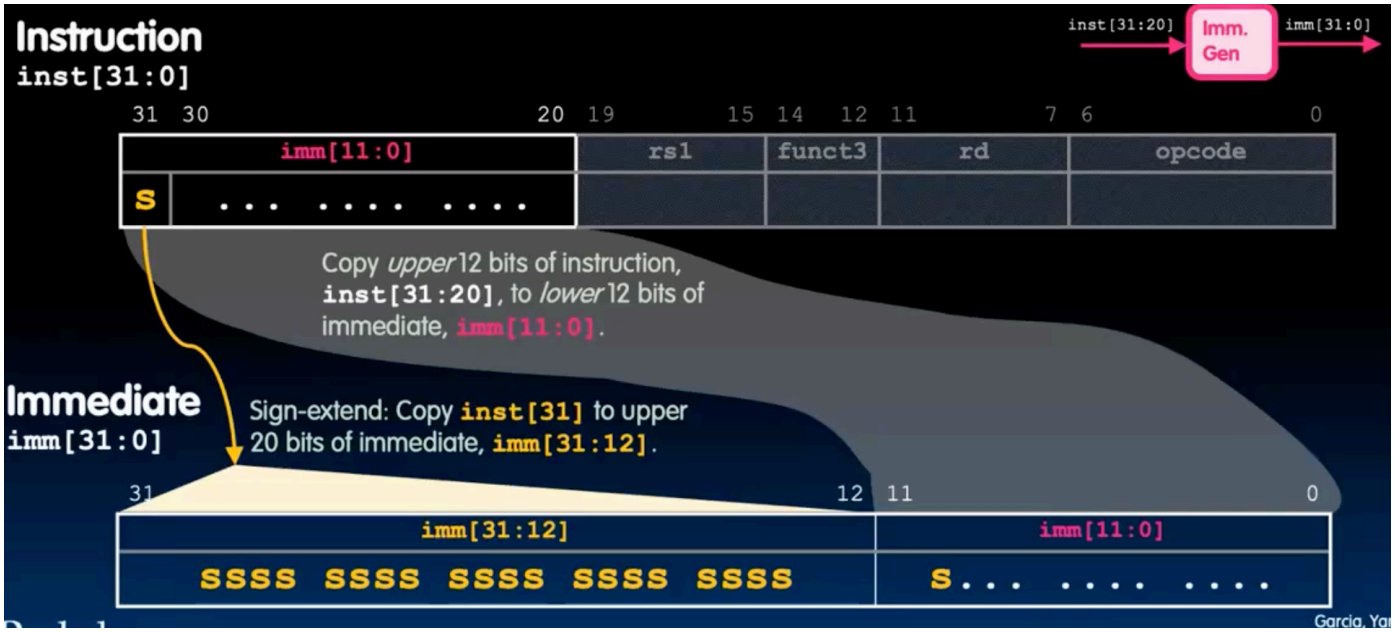
`addi` datapath

Bsel : ALU的B端输入选择


2. Immediate Generation Block builds a 32-bit immediate imm from instruction bits.
1. Control line Bsel=1 selects the generated immediate imm for ALU input B.
+4
add
pc+4
PC
clk
pc
addr
inst
IMEM
inst[11:7]
inst[19:15]
inst[24:20]
dataW
rsW
rsR1
rsR2
dataR1
dataR2
Reg[]
clk
R[rs1]
R[rs2]
A
ALU
B
alu
0
1
Split and get the upper 12 bits of instr.
inst[31:20]
Imm. Gen
imm[31:0]
inst[31:0]
ImmSel
RegWEn
1
Bsel
1
ALUSel
Add
Control Logic

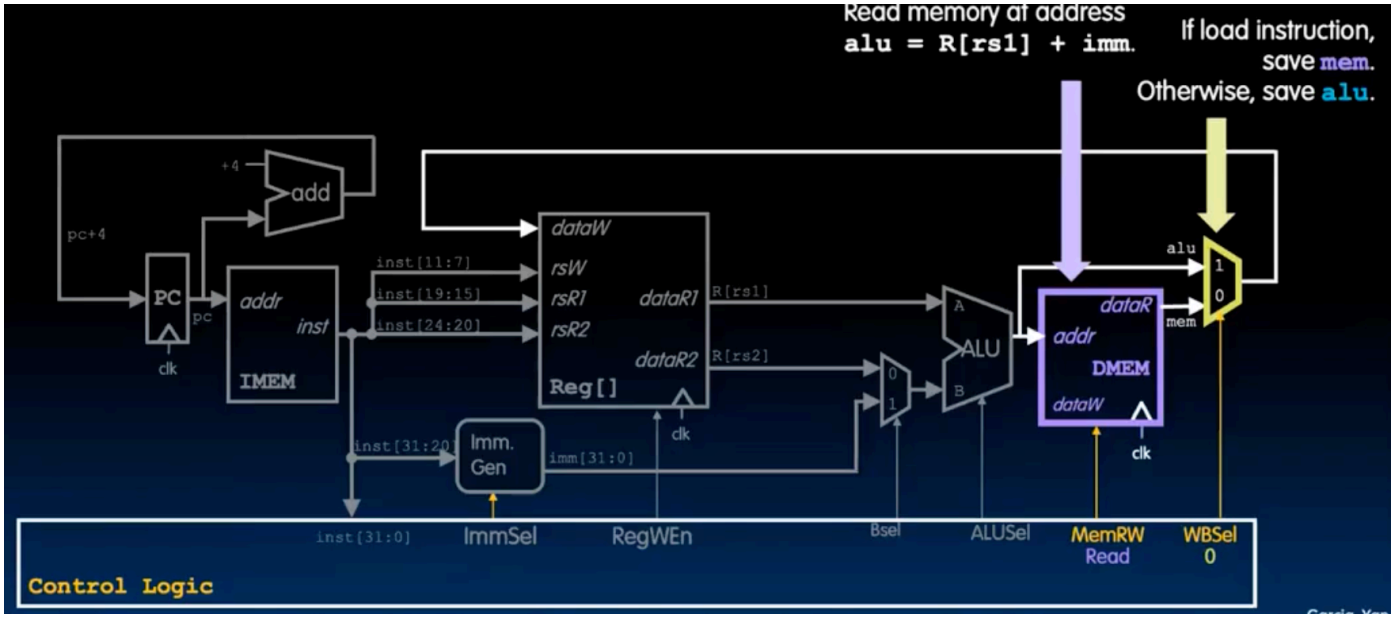

Increment PC to next instruction.
Immediate Generation Block builds a 32-bit immediate imm from instruction bits.
Control line Bsel=1 selects the generated immediate imm for ALU input B.
Write ALU output to destination register.
+4
add
pc+4
PC
clk
pc
addr
inst
IMEM
inst[11:7]
inst[19:15]
inst[24:20]
dataW
rsW
rsR1
rsR2
dataR1
dataR2
Reg[]
clk
R[rs1]
R[rs2]
A
ALU
B
alu
0
1
inst[31:20]
Imm. Gen
imm[31:0]
inst[31:0]
ImmSel
RegWEn
1
Bsel
1
ALUSel
Add
Control Logic
Data inst[24:20] still feeds into Reg[], which still outputs R[rs2].
Control Bsel=1 means R[rs2] data line is ignored.
Berkeley

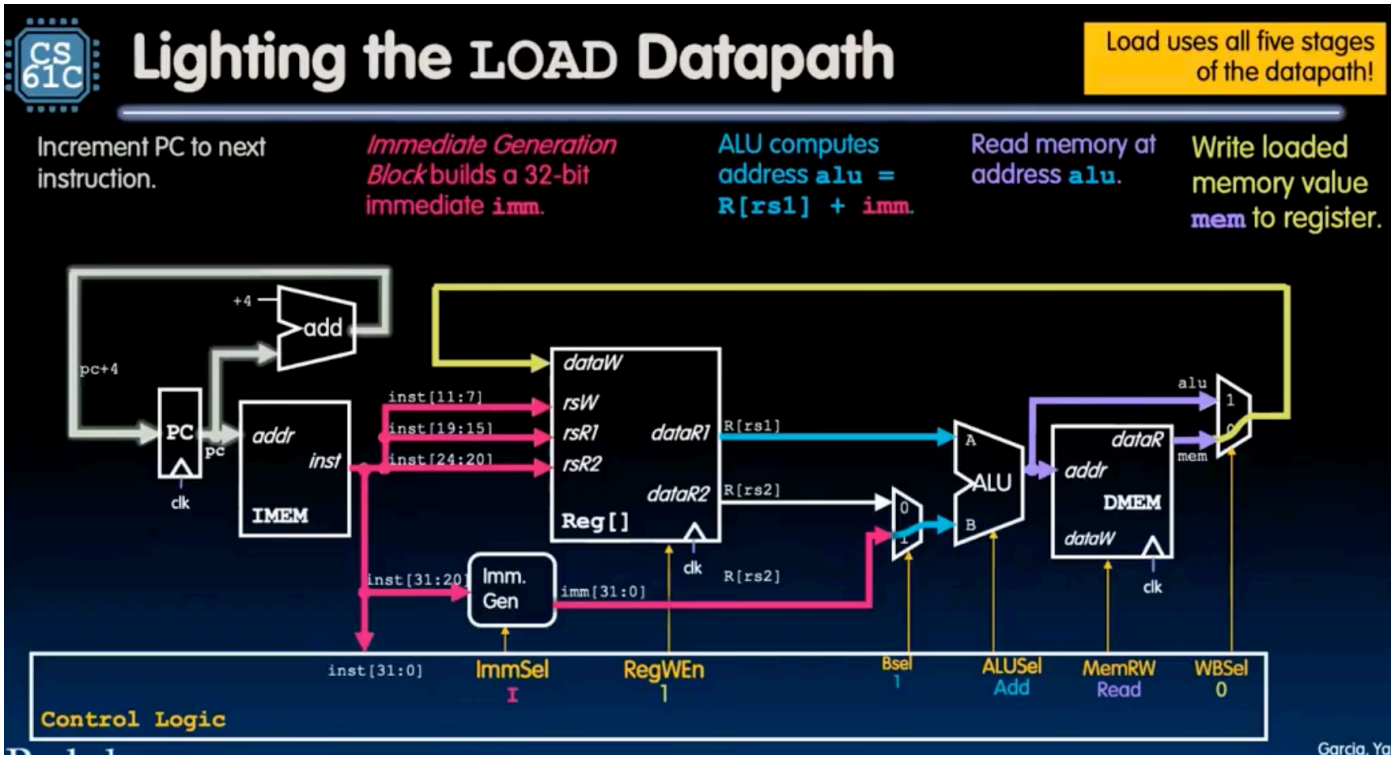
imme生成模块



Implementing Loads

WBSel : write back selection





- **To support narrower loads (`lb`, `lh`, `lbu`, `lhu`):**
  - Load 32-bit word from memory;
  - Add additional logic to extract correct byte or halfword; and
  - Sign- or zero-extend result to 32 bits to write into RegFile.
  - Can be implemented with MUX + and a few gates.

Implementing stores

- S-Format: `sw x14, 36(x2)`

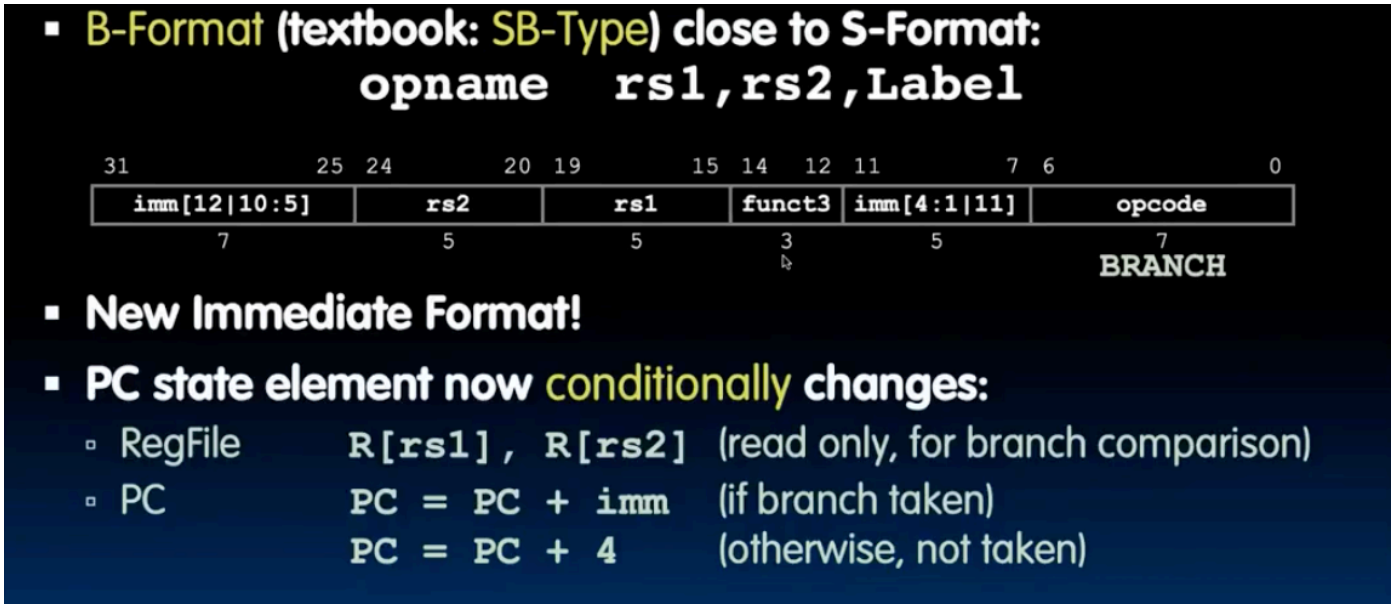
31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
0000001	01110	00010	010	00100	0100011
7	5	5	3	5	7

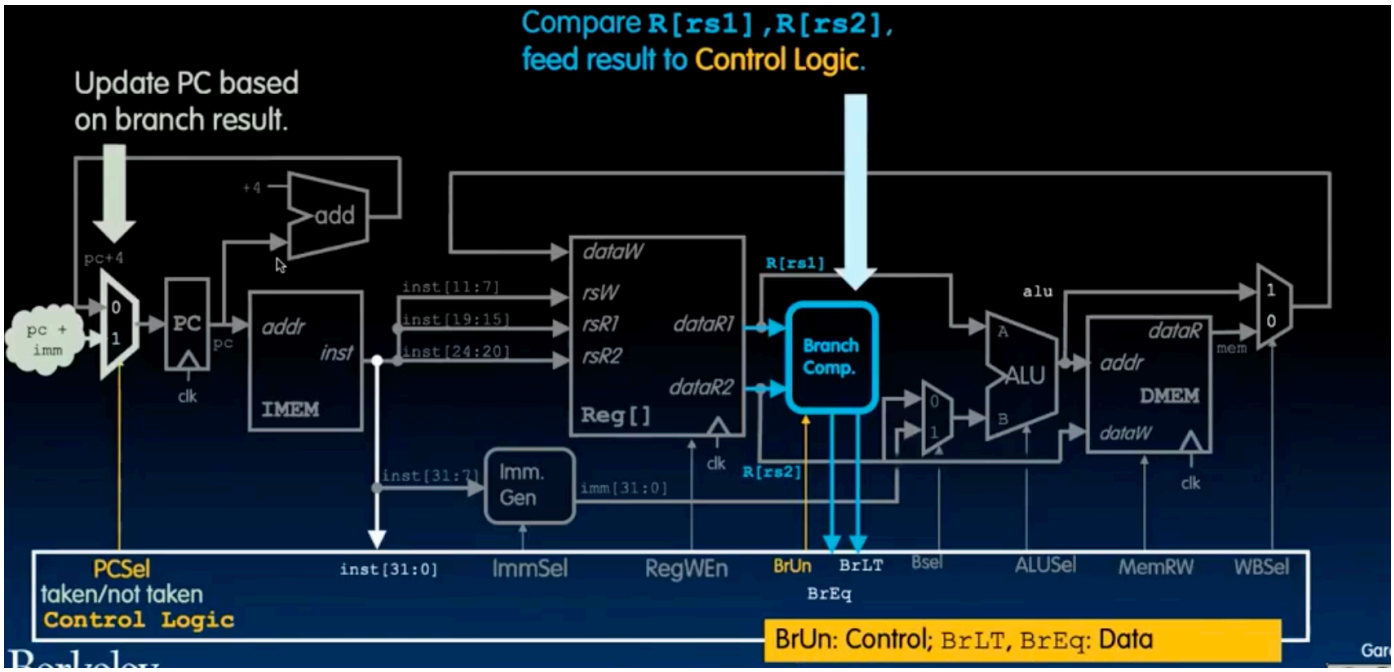
"store word" STORE

- **New Immediate Format:** `0000001` `00100`
  - `addr` = (Base register `rs1`) + (sign-extended `imm` offset)
- **State Elements Accessed:**
  - DMEM (write `R[rs2]` to word at address `addr`)
  - RegFile `R[rs1]` (base address), `R[rs2]` (value to store) — No RegFile write!
  - PC `PC = PC + 4`

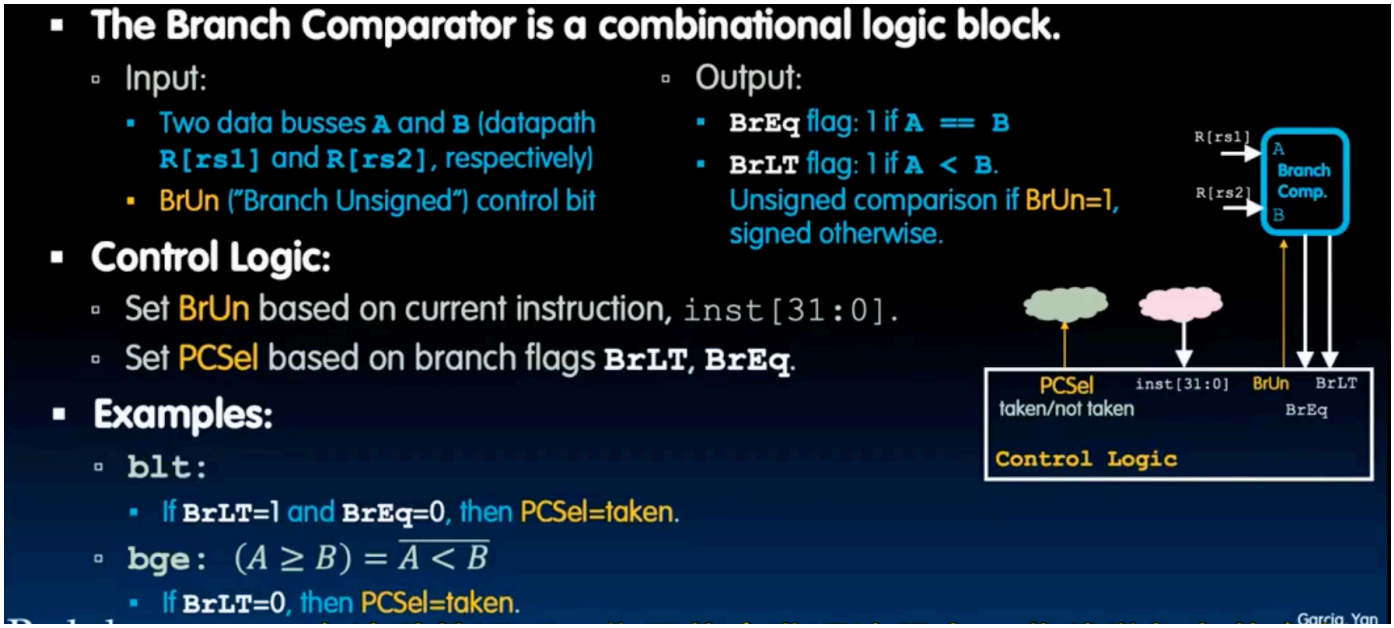
这里的WBSel是1或者0都无所谓，因为我们的RegWEn已经设置为0， 所以不会发生寄存器的更新

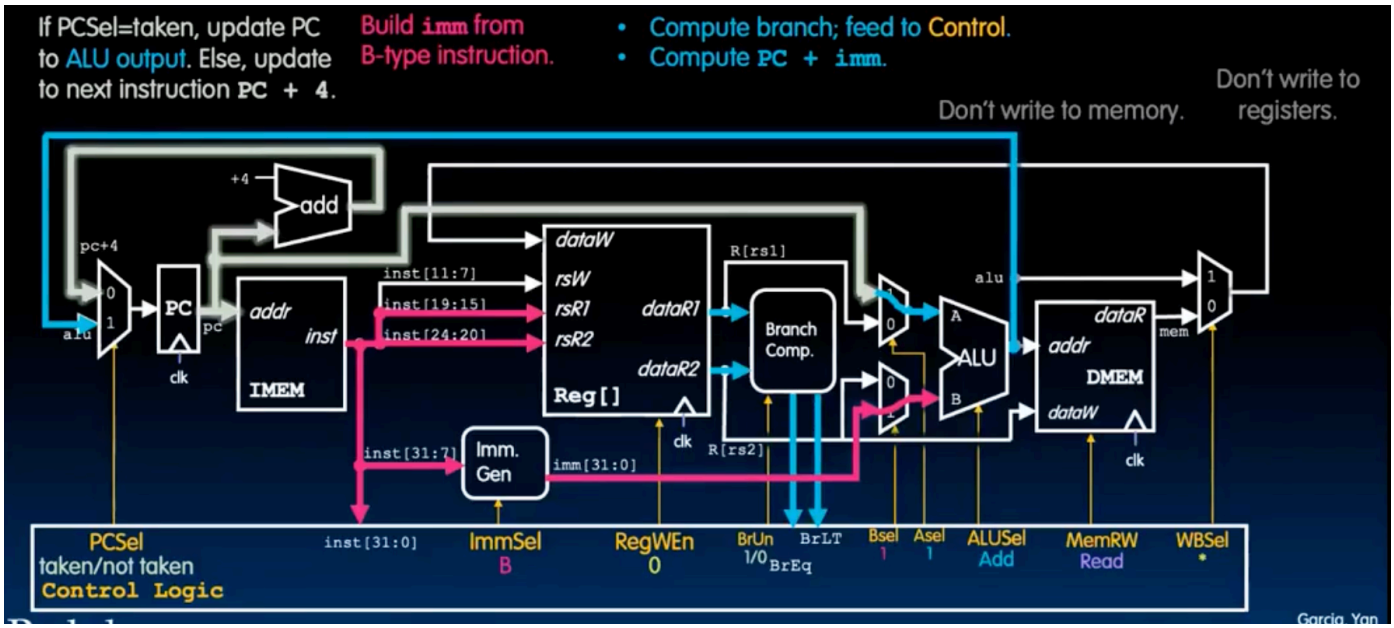
B-Format





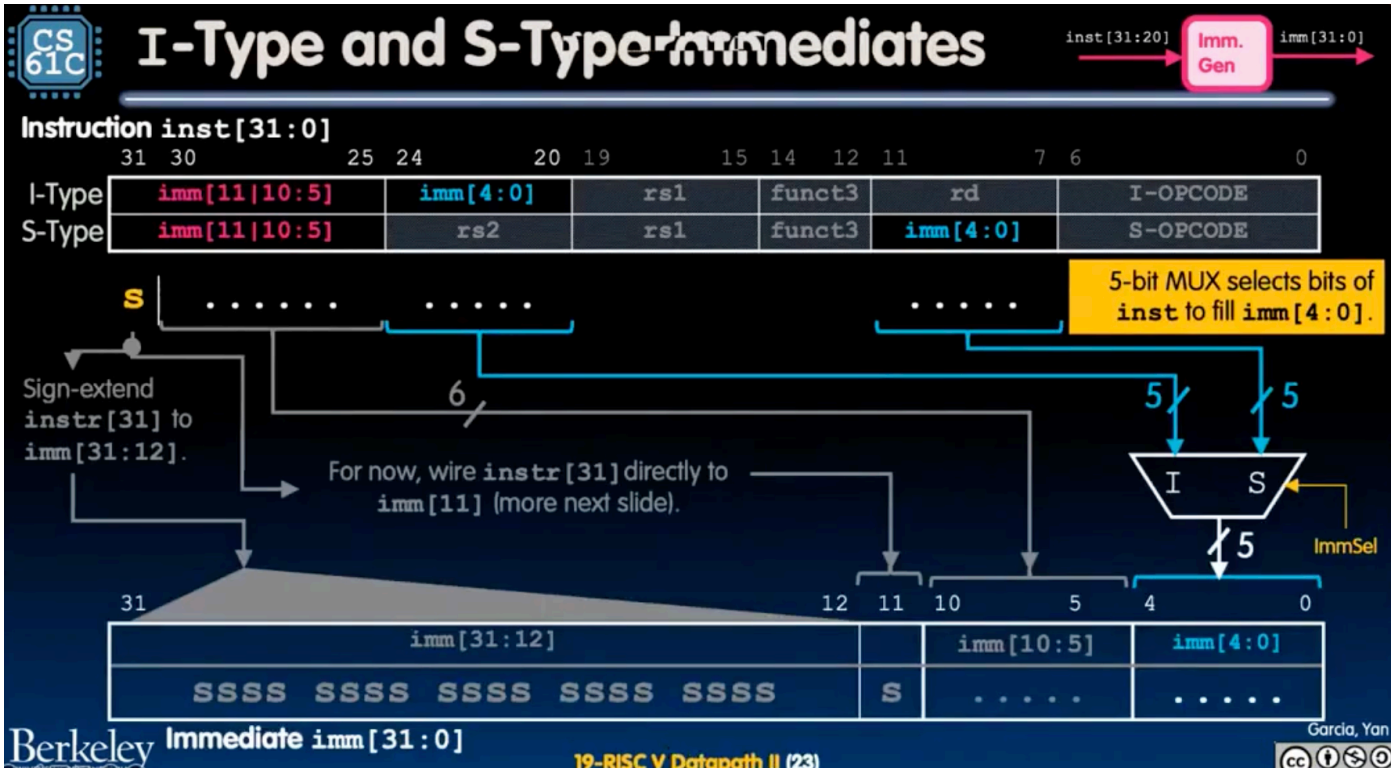
为了判断是否进行跳转，我们先在 *Branch Comp*中进行判断。使用BrUn（Branch Unsign), BrLT, BrEQ,判断指令类型返回比较结果进而控制PCSel是否启用





设计Imme生成模块 -- 第二部分

for I-Format and S-Format

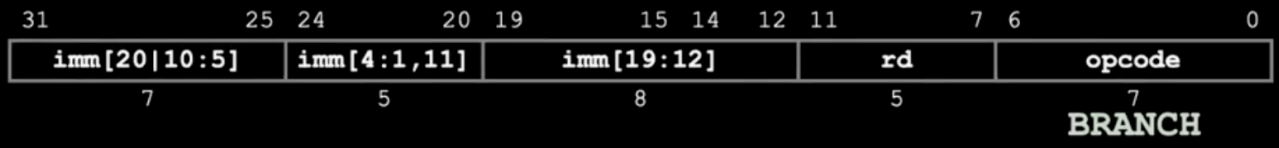


B-Format

Adding Jumps `jal jalr`

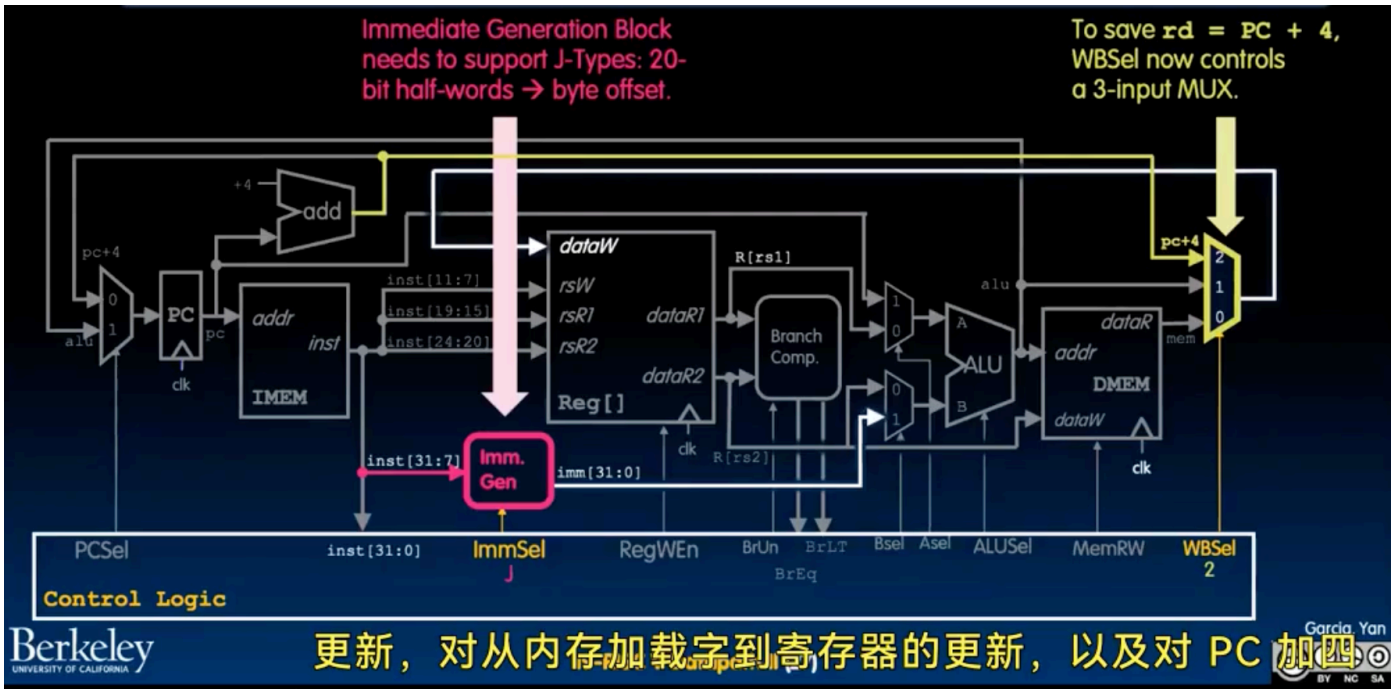
- J-Format:

jal rd, Label



- New immediate format! (see Project 3)
- State Elements updated:
  - PC `PC = PC + imm` (unconditional PC-relative jump)
  - RegFile `rd = PC + 4` } Save return address to RegFile destination register.

J型指令imme的第0位跟B型指令一样，也是隐式的0



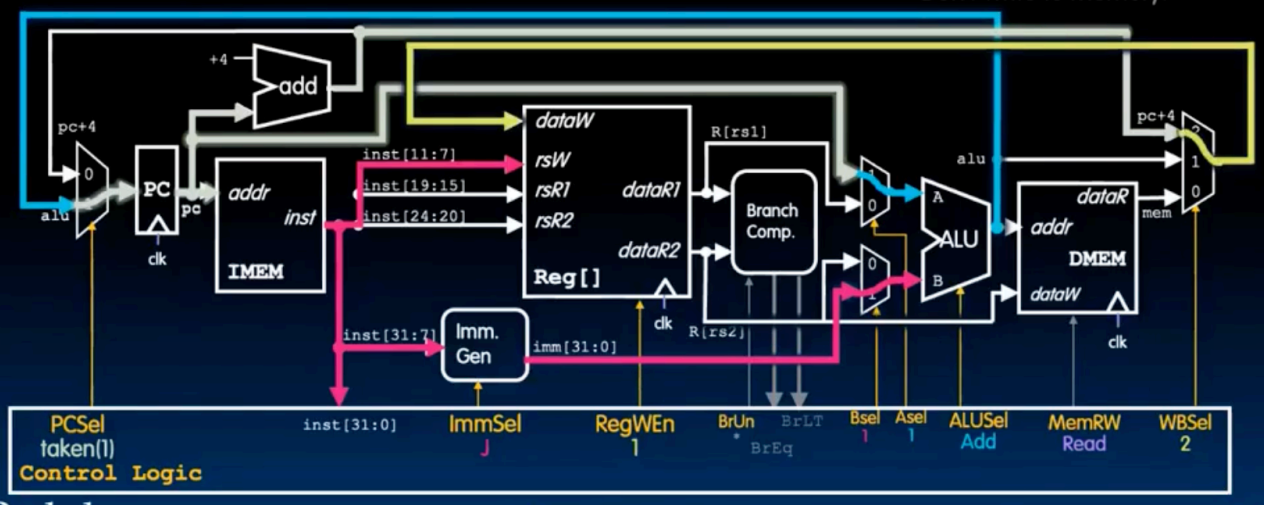
- Feed PC into blocks.
- Write ALU output to PC.

Generate byte offset `imm` for 20-bit PC-relative jump.

Compute `PC + imm`.

Write `PC + 4` to destination register.

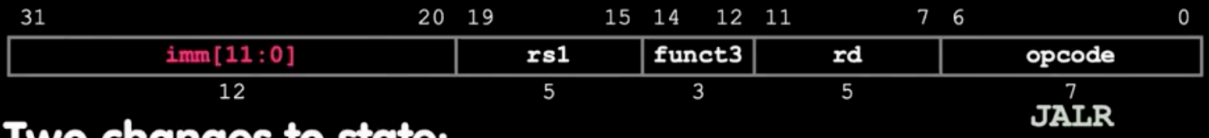
Don't write to memory.



Garcia, Yan

- `jalr` uses I-Format:

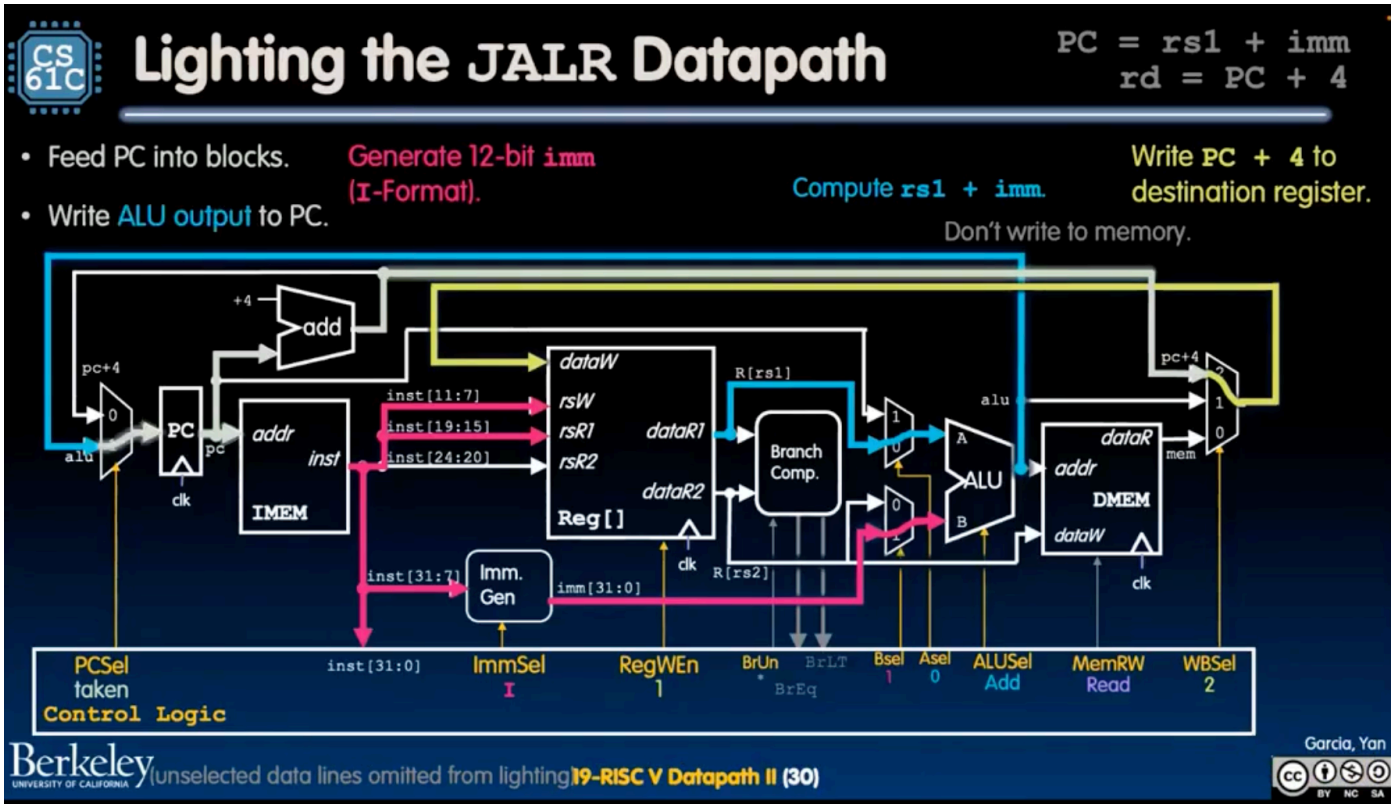
`jalr rd,rs1,imm`



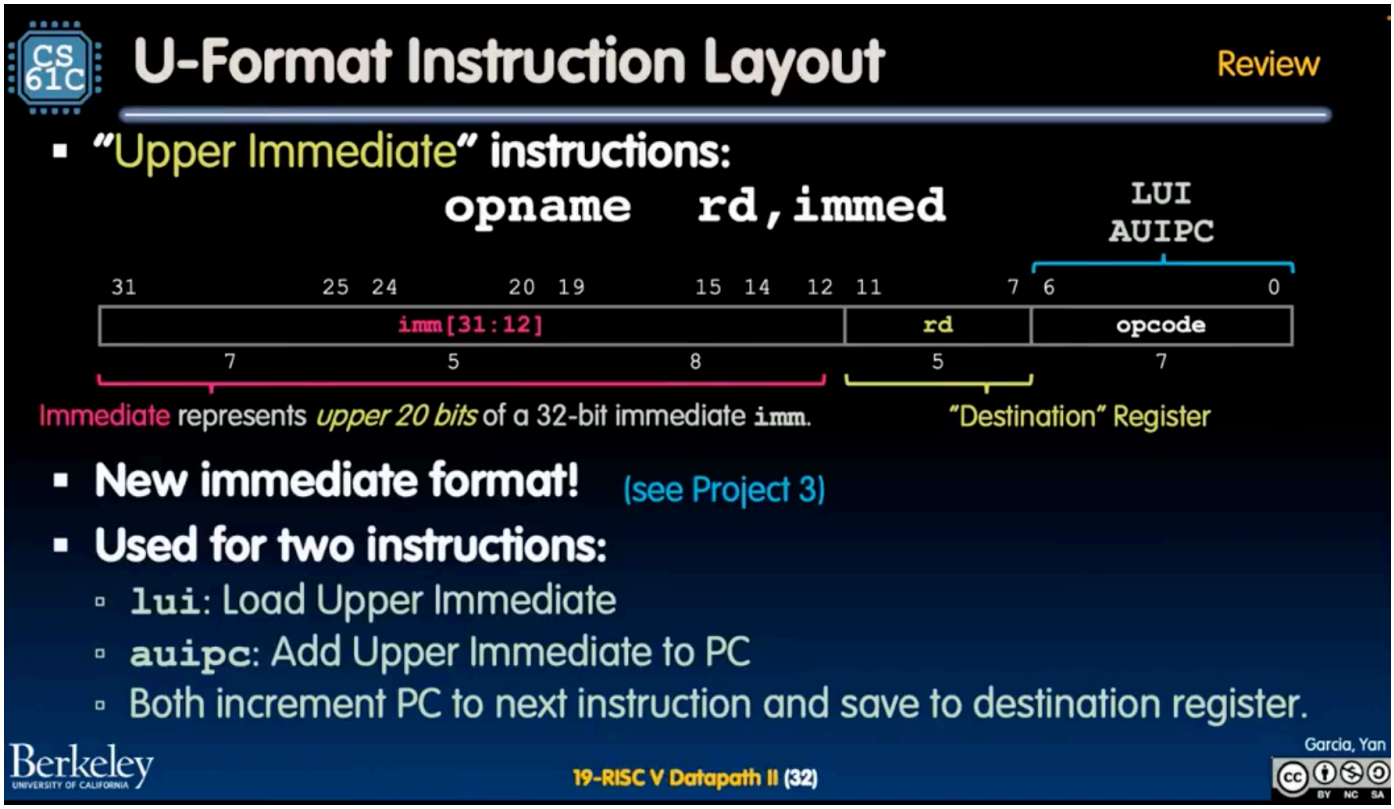
- **Two changes to state:**
  - PC `PC = rs1 + imm` (absolute addressing)
  - RegFile `rd = PC + 4`
- **I-Format means `jalr` uses the same immediates as arithmetic/loads!**
  - In other words, `imm` is already a byte offset.

Control ImmSel is based on instruction format, not instruction. So far: `I,S,B,J`

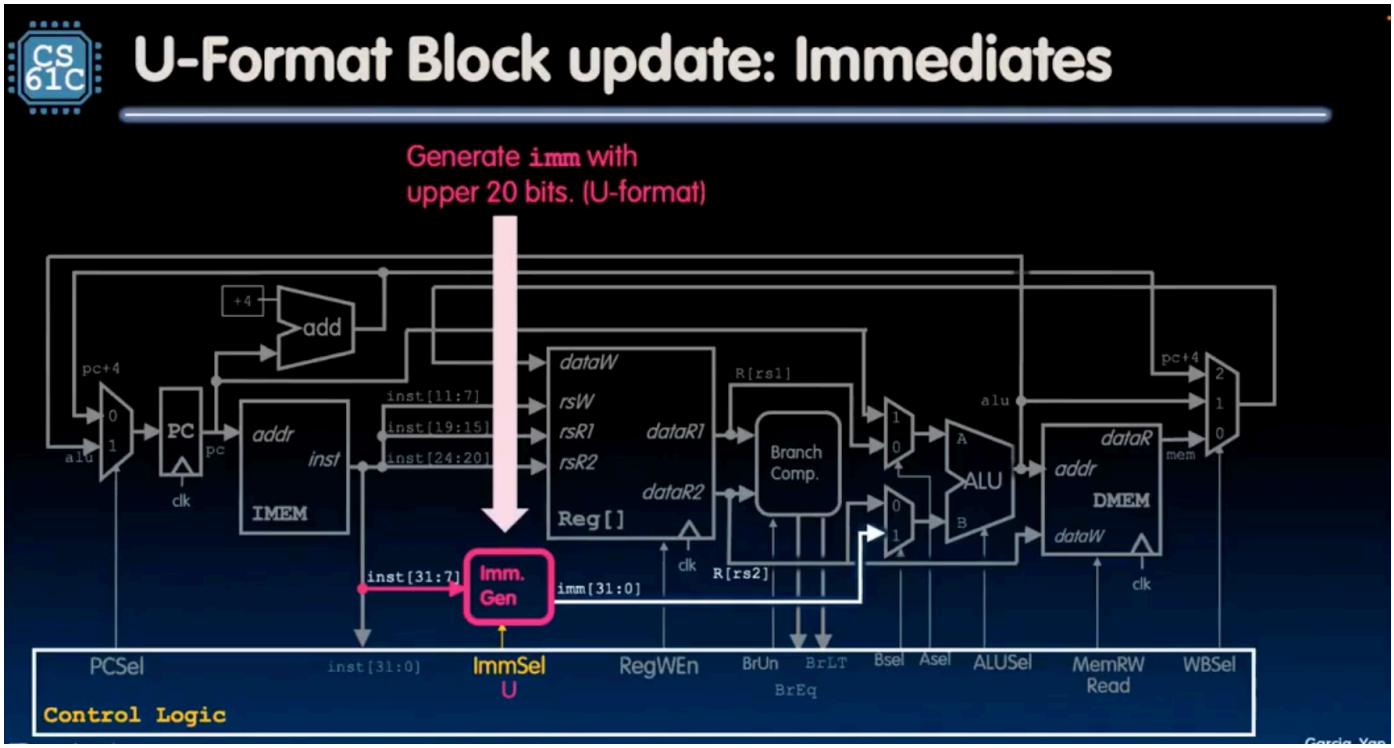
Garcia, Yan


Lighting the JALR Datapath
PC = rs1 + imm
rd = PC + 4
• Feed PC into blocks.
• Write ALU output to PC.
Generate 12-bit imm (I-Format).
Compute rs1 + imm.
Don't write to memory.
Write PC + 4 to destination register.
+4
add
pc+4
alu
PC
clk
pc
addr
inst
IMEM
inst[11:7]
inst[19:15]
inst[24:20]
dataW
rsW
rsR1
rsR2
dataR1
dataR2
Reg[]
clk
R[rs1]
R[rs2]
Branch Comp.
alu
A
B
ALU
dataR
addr
DMEM
dataW
mem
clk
pc+4
inst[31:7]
Imm. Gen
imm[31:0]
PCSel
taken
Control Logic
inst[31:0]
ImmSel
I
RegWEn
1
BrUn
BrEq
BrLT
Bsel
1
Asel
0
ALUSel
Add
MemRW
Read
WBSel
2
Berkeley
UNIVERSITY OF CALIFORNIA
(unselected data lines omitted from lighting)
19-RISC V Datapath II (30)
Garcia, Yan

U-Format


U-Format Instruction Layout
Review
• "Upper Immediate" instructions:
opname rd,immed
LUI
AUIPC
31
25 24
20 19
15 14
12 11
7 6
0
imm[31:12]
rd
opcode
7
5
8
5
7
Immediate represents upper 20 bits of a 32-bit immediate imm.
"Destination" Register
• New immediate format! (see Project 3)
• Used for two instructions:
◦ lui: Load Upper Immediate
◦ auipc: Add Upper Immediate to PC
◦ Both increment PC to next instruction and save to destination register.
Berkeley
UNIVERSITY OF CALIFORNIA
19-RISC V Datapath II (32)
Garcia, Yan

U-Format Block update: Immediates



Lighting the LUI Datapath

PC = PC + 4
rd = imm

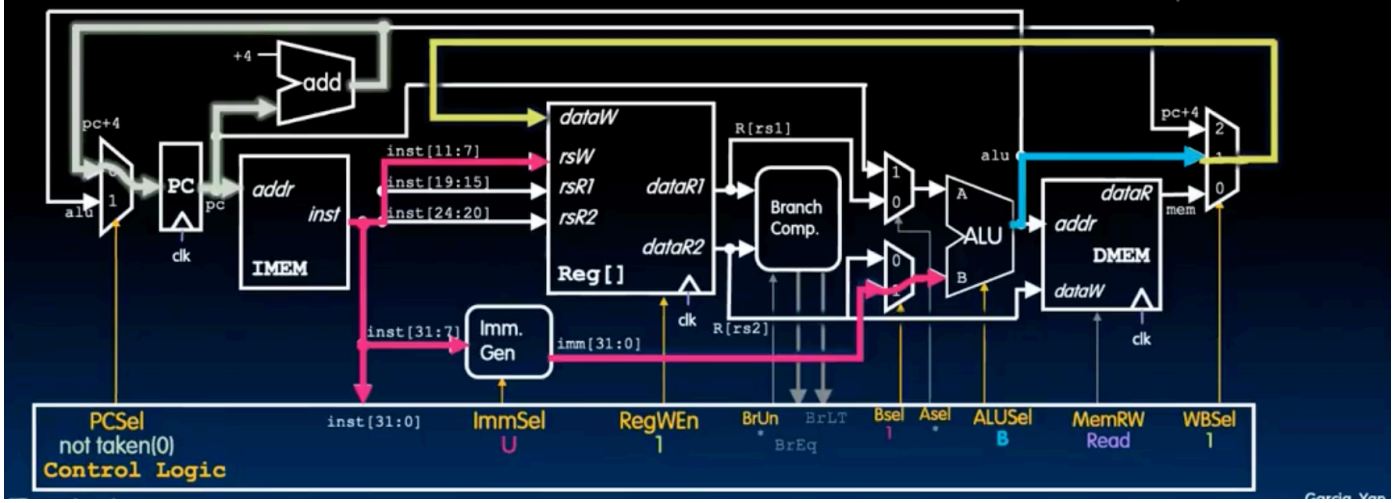
Increment PC to next instruction.

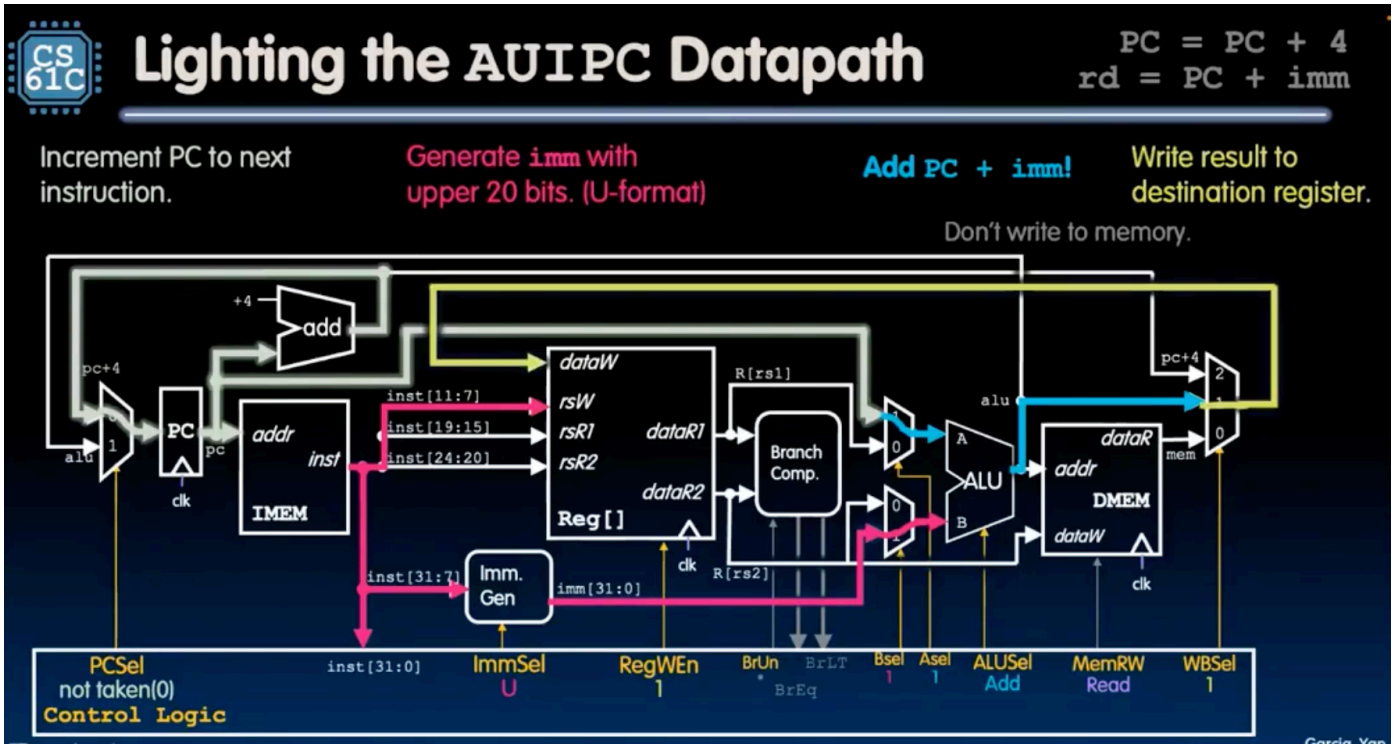
Generate imm with upper 20 bits. (U-format)

Grab only imm! (ALUSel=B)

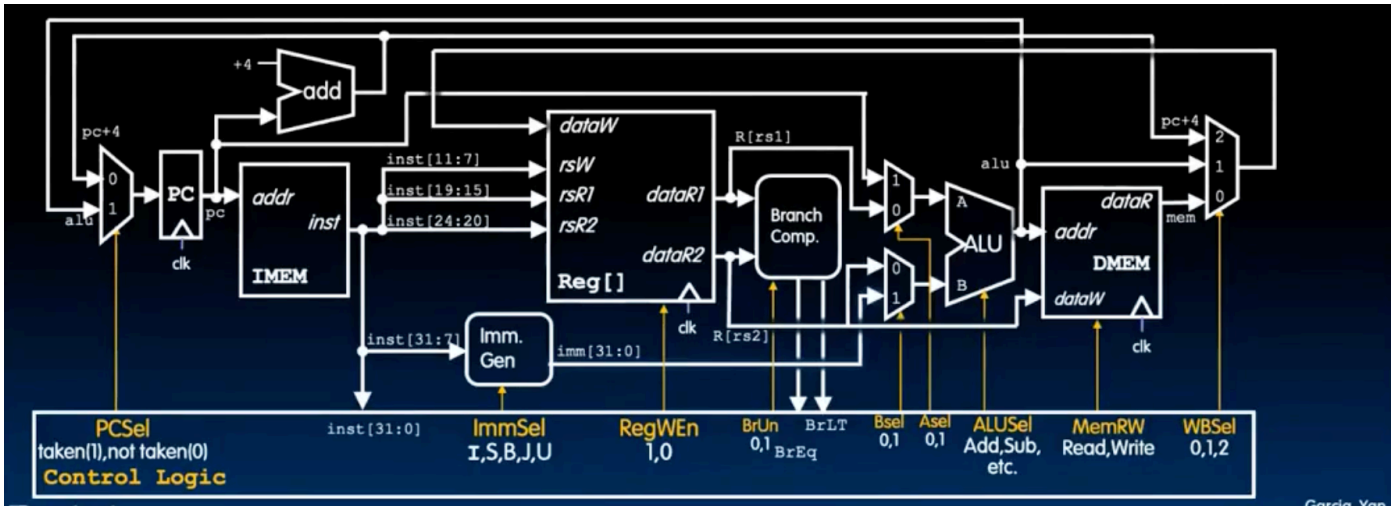
Write result to destination register.

Don't write to memory.





🥰

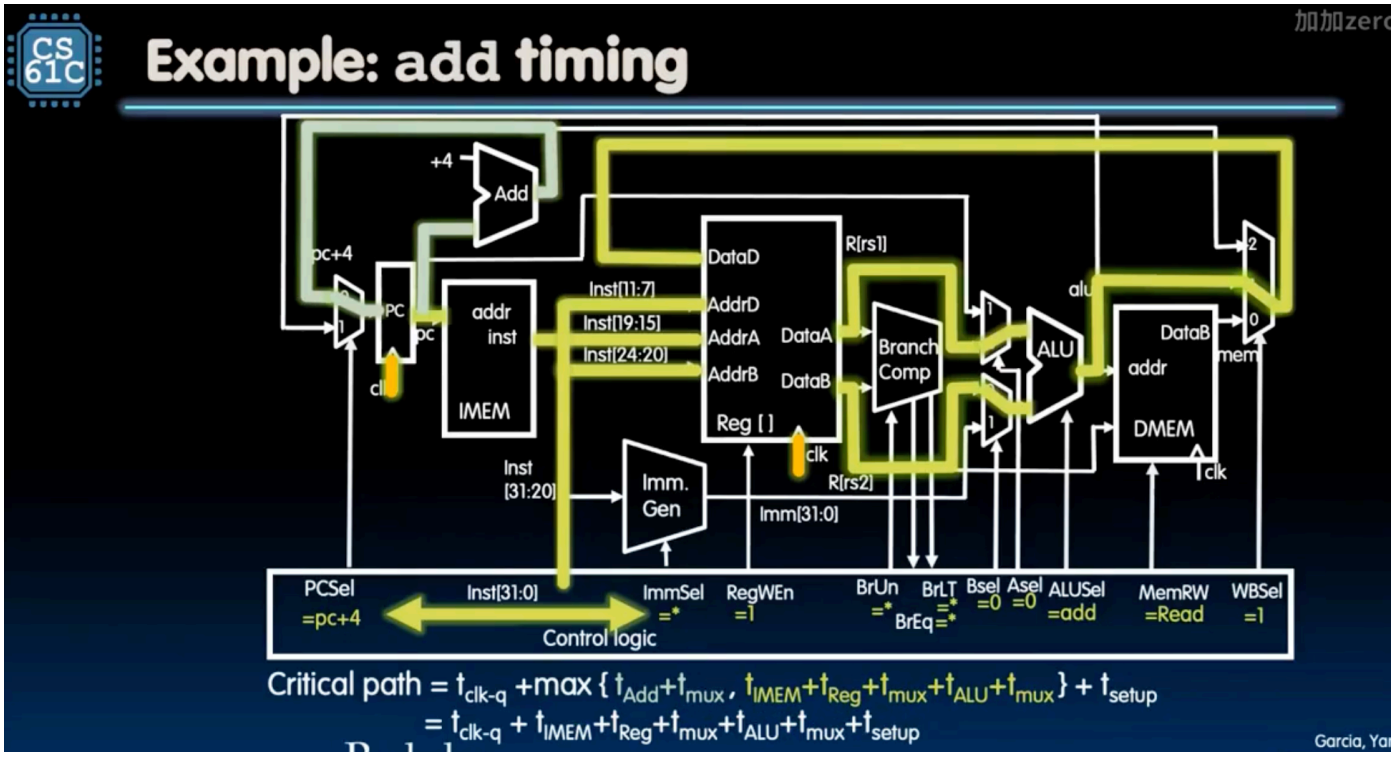


Lecture 20 : Control

Instruction Timing

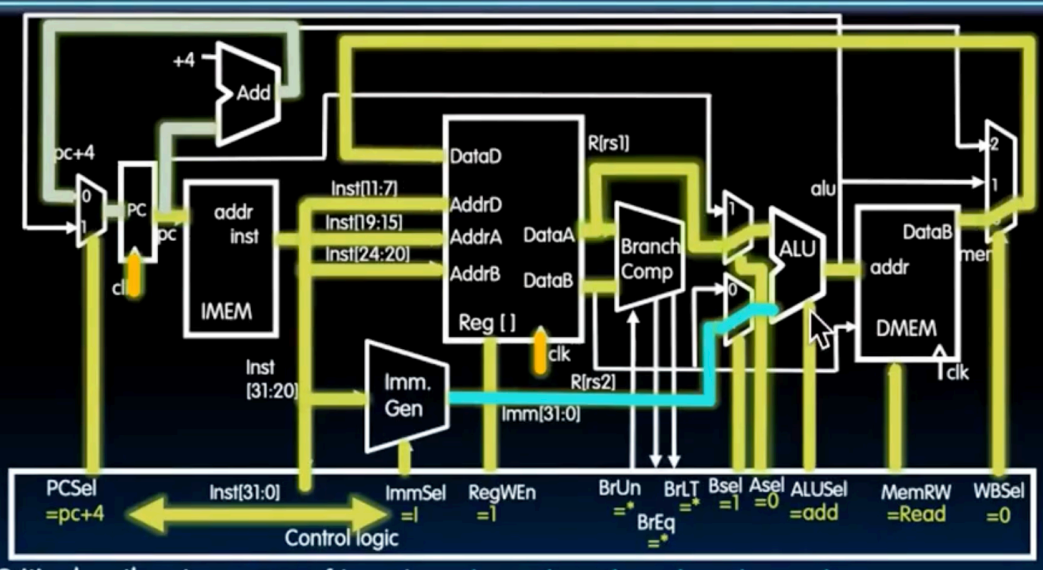
`add`举例


CS 61C
add Execution
e.g.,
add x1,x2,x3
Add
IMEM
Reg [ ]
Branch Comp
ALU
DMEM
Imm. Gen
Control logic
PCSel
ImmSel
RegWEn
BrUn
BrLT
Bsel
Asel
ALUSel
MemRW
WBSel
BrEq
Clock
PC
PC+4
inst[31:0]
Control logic
Reg[rs1]
Reg[rs2]
alu
wb
Reg[1]
1000
1004
1008
add x1,x2,x3
add x6,x7,x9
add control
Reg[2]
Reg[7]
Reg[3]
Reg[9]
Reg[2]+Reg[3]
Reg[7]+Reg[9]
???


CS 61C
Example: add timing
Add
IMEM
Reg [ ]
Branch Comp
ALU
DMEM
Imm. Gen
Control logic
PCSel
=pc+4
ImmSel
=*
RegWEn
=1
BrUn
=*
BrLT
=*
Bsel
=0
Asel
=0
ALUSel
=add
MemRW
=Read
WBSel
=1
Critical path = t_{clk-q} + max { $t_{Add}+t_{mux}$, $t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{mux}$ } + t_{setup}
= t_{clk-q} + $t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{mux}+t_{setup}$

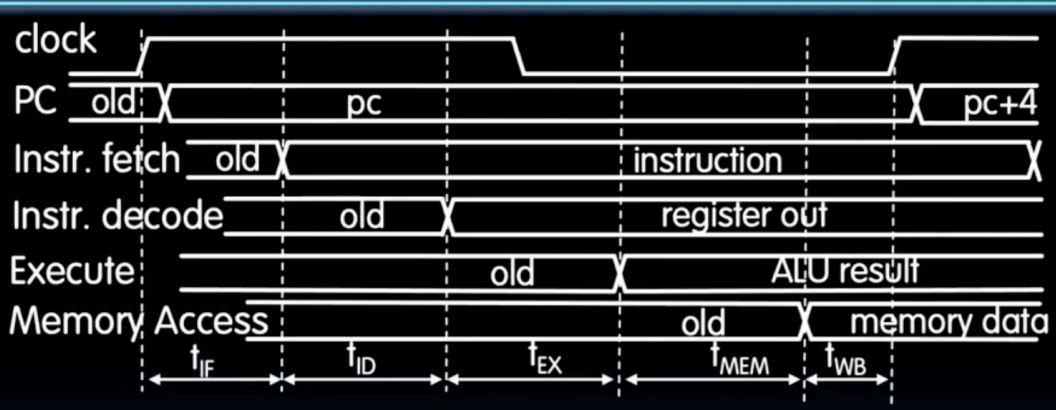
`lw`举例

Example: lw reg, offset(regbaseptr)

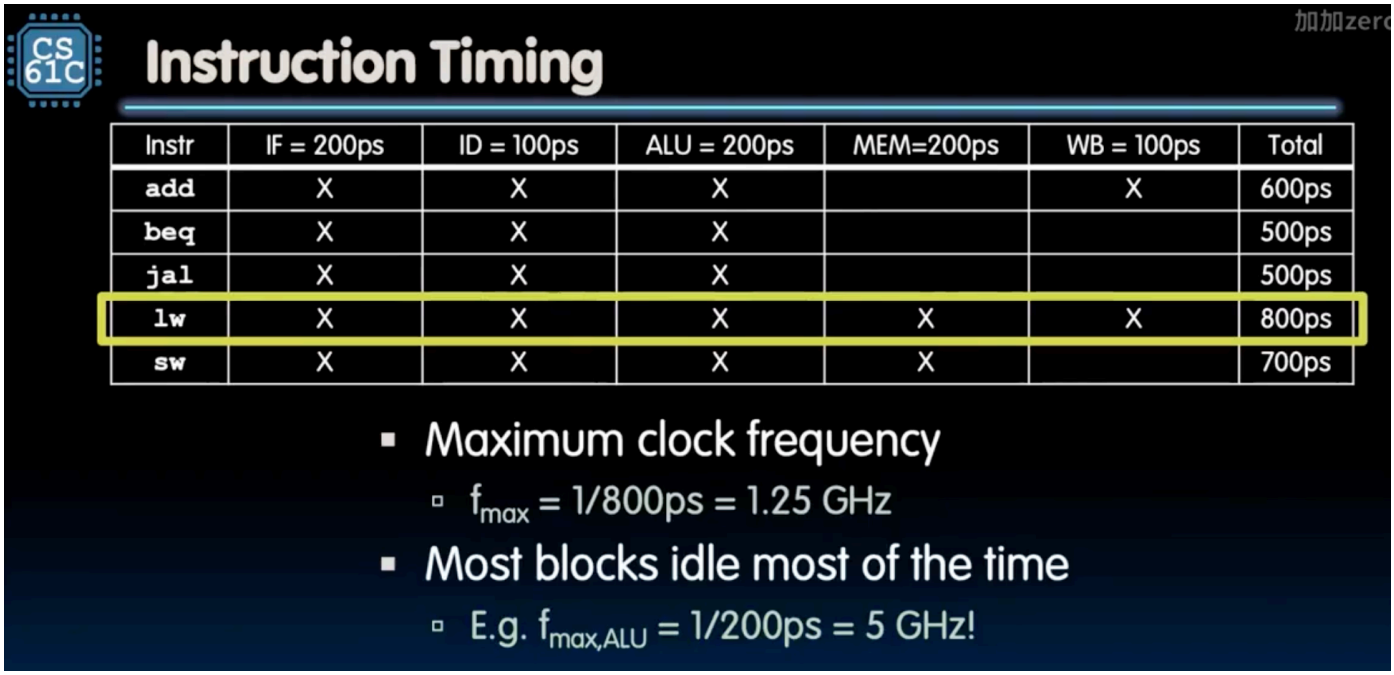


Critical path = $t_{clk\text{-}q} + \max\{ t_{Add}+t_{mux},\ t_{IMEM}+t_{Imm}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux},\ t_{IMEM}+t_{Reg}+t_{mux}+t_{ALU}+t_{DMEM}+t_{mux} \}+t_{setup}$

Instruction Timing

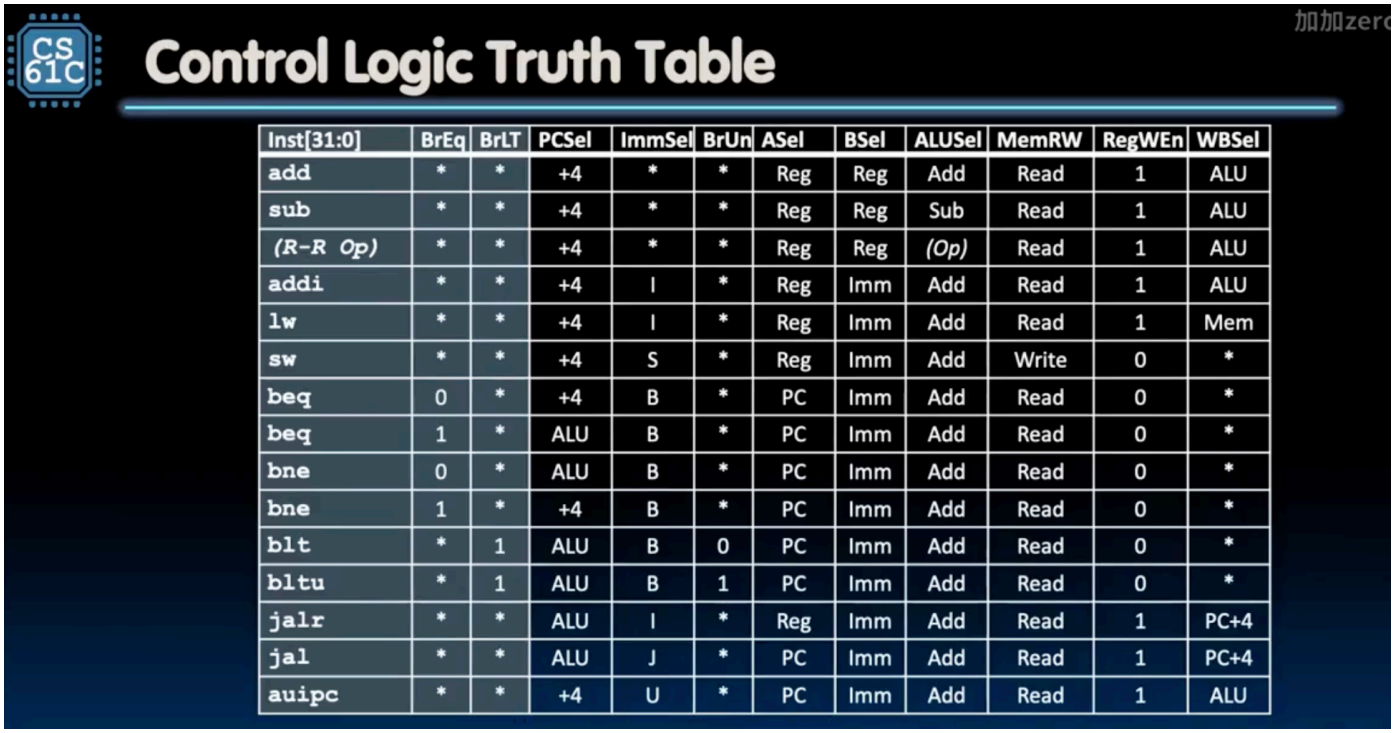


IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps



在延迟情况下，程序通过一个部分后这部分硬件就闲置了，效率太低 --- 考虑如何分配来提高效率

Control Logic Design



Realization Options

- ROM
  - Read-Only Memory
  - Regular structure
  - Can be easily reprogrammed
  - Popular when design control logic manually

利用指令的操作码进行解码，然后进行逻辑控制，例如：