

# Lecture 11, 12 : RISC-V Instruction Format

## History's Big Idea: Strored Program Computer

- Instructions are represented as bits patterns

## Stored Program Computer: Implications

- Everything (instructions, data words) has a memory address
  - The *Program Counter*(PC) register keeps address of *next* instruction to execute
- 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	Register-register arithmetic operations
I-Format	Register-immediate arithmetic operations; Loads
S-Format	Stores
B-Format	Branches (minor variant of S-format)
U-Format	20-bit upper immediate instructions
J-Format	Jumps (minor variant of U-format)

### R-Format Layout

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

## opname rd, rs1, rs2



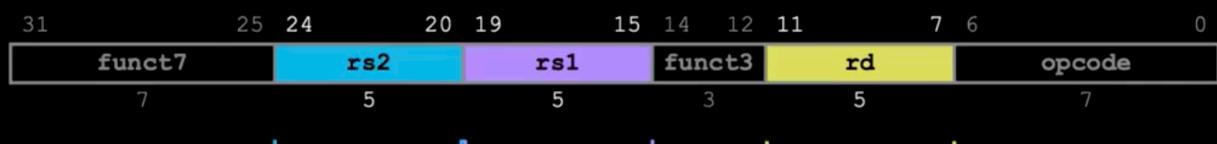
funct7, funct3 combined with opcode describes what operation to perform.

- Why not just a 17-bit field for simplicity?  
We'll answer this later...

opcode partially specifies which instruction it is.

All R-Format instructions have opcode 0110011.

## opname rd, rs1, rs2



"Source" Register 2  
contains second operand

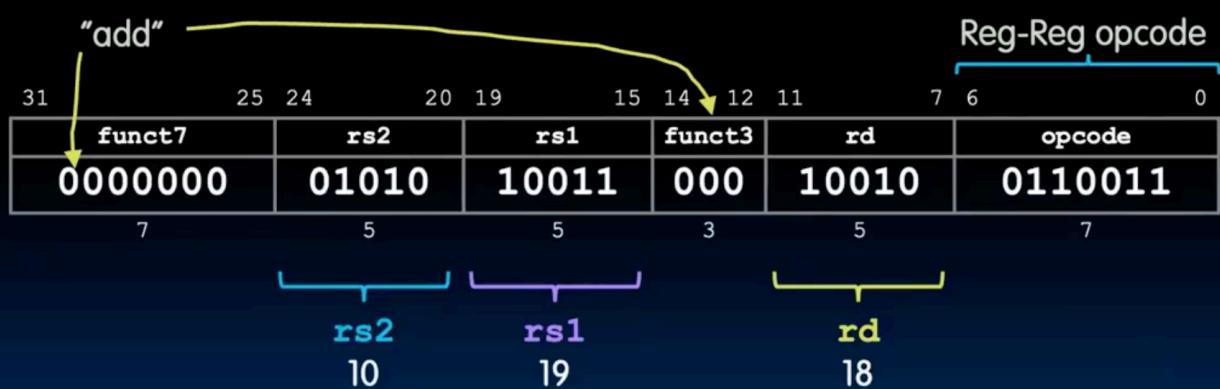
"Source" Register 1  
contains first operand

"Destination" Register  
gets result of computation

Register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31).

举个例子

## add x18, x19, x10

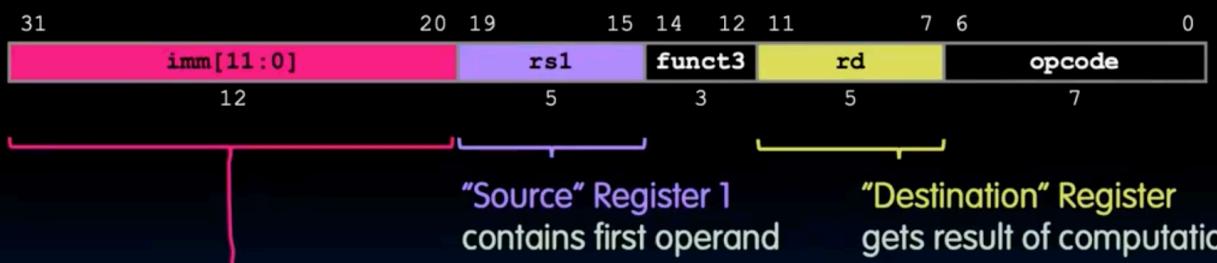


全部十个R型指令

			Eight funct3 fields for ten instructions				
			funct3	opcode			
funct7	rs2	rs1	000	rd	0110011	add	
2's comp of rs2	0100000	rs2	000	rd	0110011	sub	
	0000000	rs2	001	rd	0110011	sll	
	0000000	rs2	010	rd	0110011	slt	
	0000000	rs2	011	rd	0110011	sltu	
	0000000	rs2	100	rd	0110011	xor	
sign extend	0000000	rs2	101	rd	0110011	srl	
	0100000	rs2	101	rd	0110011	sra	
	0000000	rs2	110	rd	0110011	or	
	0000000	rs2	111	rd	0110011	and	

## I-Format

- **Register-Immediate Arithmetic Instructions (addi, xori, etc.)**
- opname   rd, rs1, imm**



Imm[11:0] holds 12-bit-wide immediate values:

- Values in range  $[-2048_{\text{ten}}, +2047_{\text{ten}}]$
- CPU sign-extends to 32 bits before use in an arithmetic operation
- How to handle immediates > 12 bits? (more next time)

所有的9个I型运算指令

Same funct3 fields as corresponding R-format operation (remember, no subi)

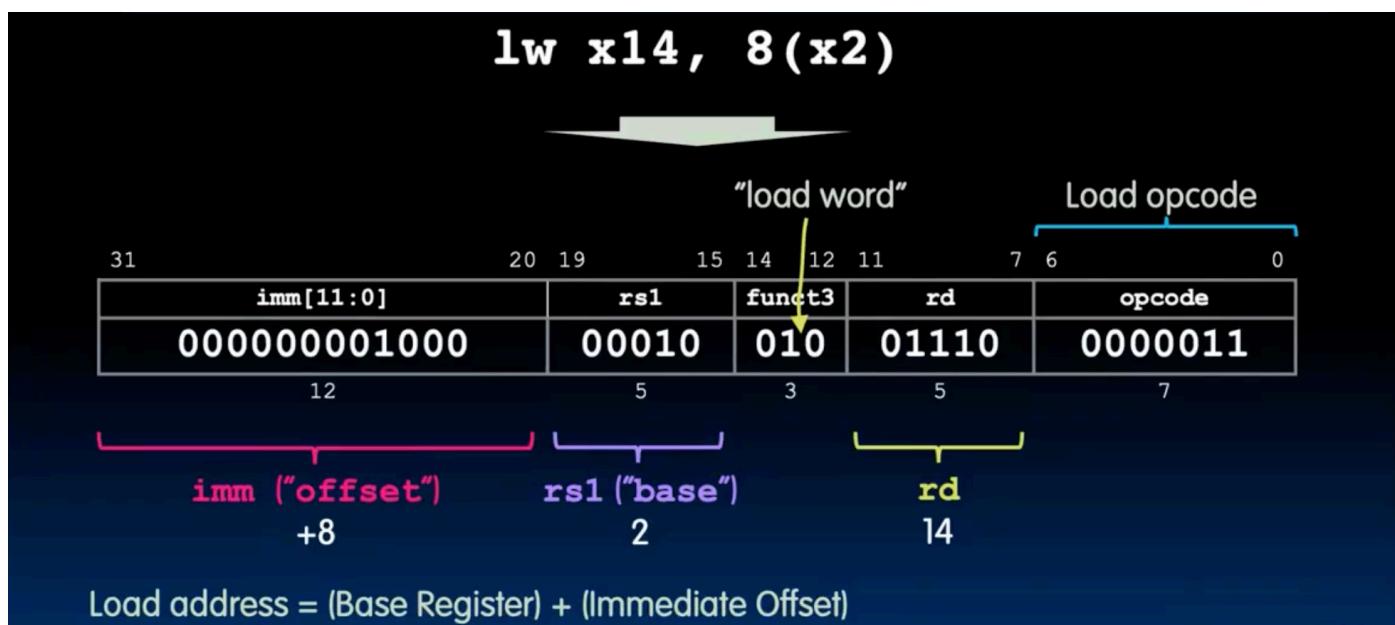
			<u>funct3</u>	opcode		
imm[11:0]	rs1	000	rd	0010011	addi	
imm[11:0]	rs1	010	rd	0010011	slti	
imm[11:0]	rs1	011	rd	0010011	sltiu	
imm[11:0]	rs1	100	rd	0010011	xori	
imm[11:0]	rs1	110	rd	0010011	ori	
imm[11:0]	rs1	111	rd	0010011	andi	
00000000	shamt	rs1	001	rd	0010011	slli
00000000	shamt	rs1	101	rd	0010011	srlti
01000000	shamt	rs1	101	rd	0010011	srai

"Shift by Immediate" instructions encode the shift amount in the lower-order 5 bits.

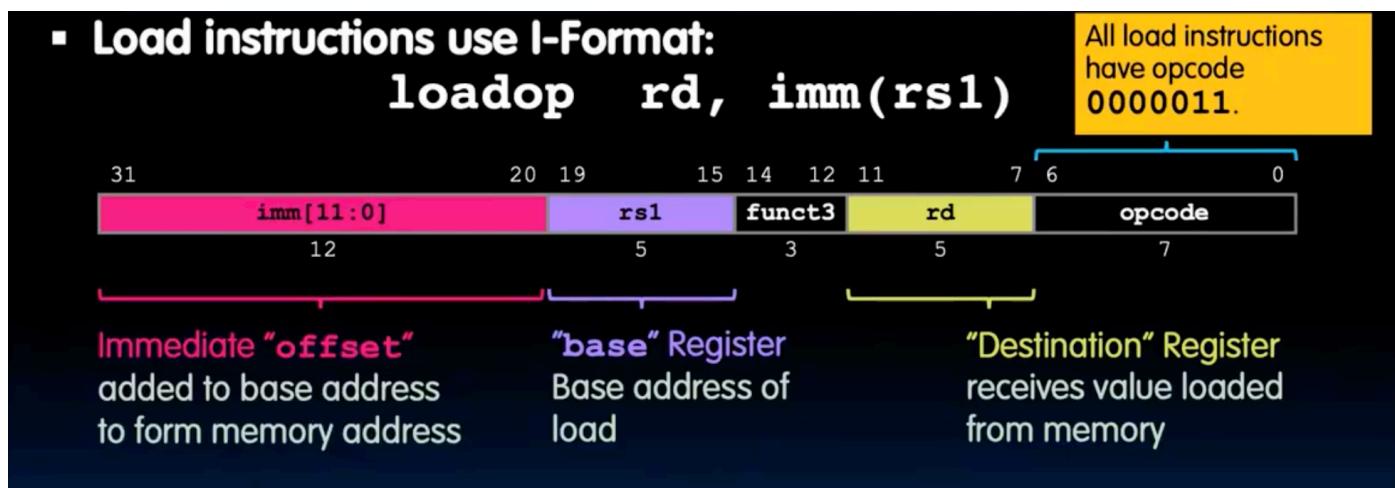
- We can only (meaningfully) shift 32-b word by 0-31 positions.
- One higher-order immediate bit used for sign extend (**srlti** vs. **srai**). Same bit position as in R-Format!

I型指令也可以用于加载指令

举个例子



Load型格式

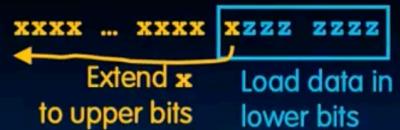


## 五个加载指令

Encodes data size and "signedness" of load operation					
		funct3	opcode		
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

- lb: "load byte," lh: "load halfword" (16 bits)**
  - Sign extend* to fill upper bits of destination 32-bit register.
- lbu: "load unsigned byte," lhu: "load unsigned halfword"**
  - Zero extend* to fill upper bits of destination 32-bit register.
- Note no lwu instruction in RISC-V.**
  - Simplicity: No need to sign/zero extend when copying 32 bits into 32-bit register.

Garcia, Yan



## S型指令

▪ Store instructions have their S-Format. <b>storeop rs2, imm(rs1)</b>							
All store instructions have opcode 0100011.							
Immediate "offset" added to base address to form memory address. Store address = (Base Register) + (Immediate Offset)							
<ul style="list-style-type: none"> <li>The immediate's higher 7 bits and lower 5 bits are in separate fields!</li> </ul>							

Garcia, Yan

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

I-type							
31	25 24	20 19	15 14 12 11	7 6	0		
imm[11:5]	imm[5:0]	rs1	funct3	rd	opcode		
12		5	3	5	7		
R-type							
funct7	rs2	rs1	funct3	rd	opcode		
7	5	5	3	5	7		

Garcia, Yan

# 更新the Program Counter(PC)

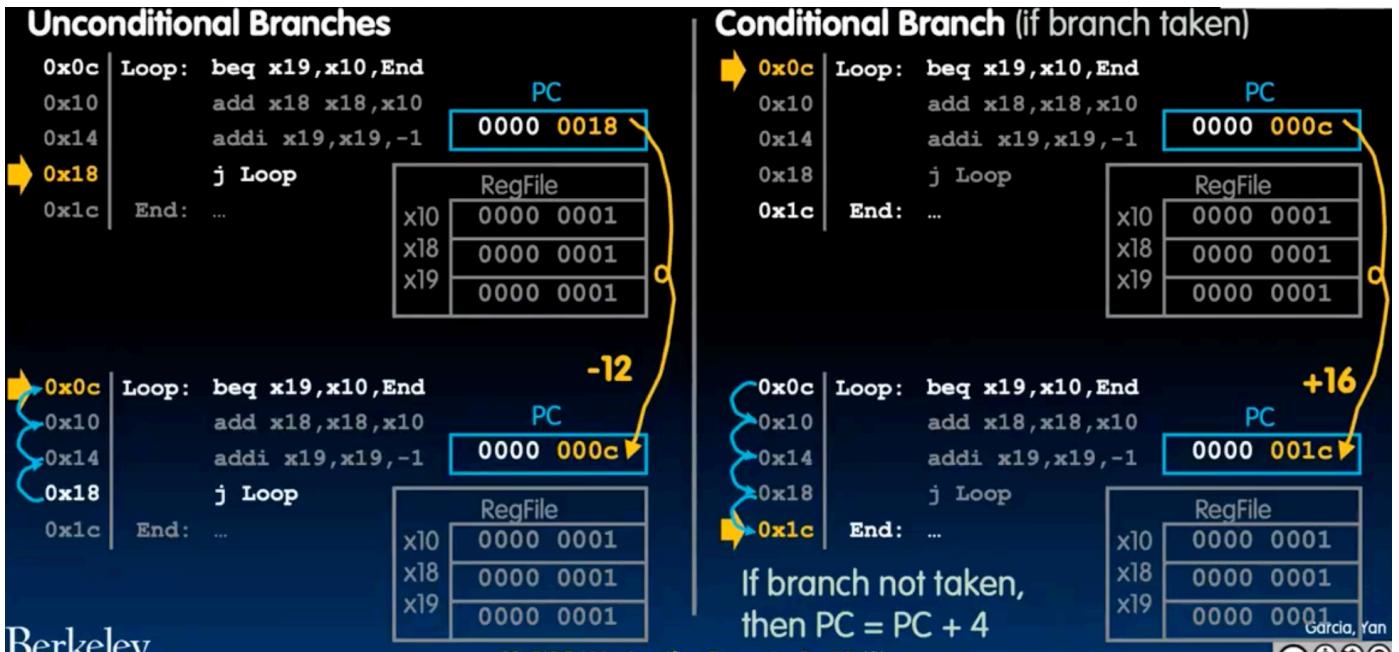
PC是一种在处理器内部的寄存器，并且与32个寄存器分开，它容纳了被执行指令的字节地址

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)

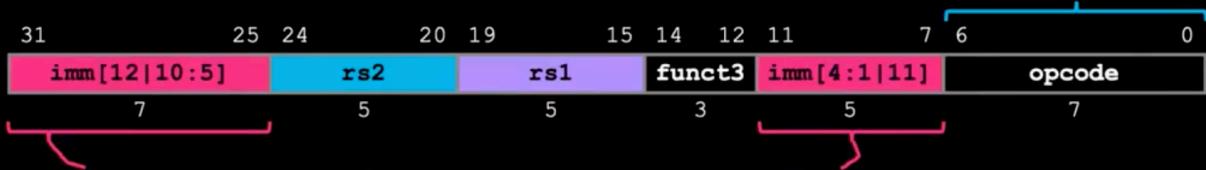
Branch immediate range [-2048, +2047]

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

## B型指令布局

- B-Format (textbook: SB-Type) close to S-Format:  
opname rs1,rs2,Label

All conditional/branch instructions have opcode 1100011.



Immediate represents relative offset in increments of 2 bytes.

- To compute new PC:  $PC = PC + \text{byte\_offset}$ .
- 12 immediate bits imply  $\pm 2^{10}$  32-bit instructions reachable:
  - 1 bit: 2's complement (allow +/- offset)
  - 1 bit: 16-b instruction support

(Lowest bit of offset is always zero, so no need to store in instruction.)



Berkeley

Garcia, Yan

## 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。

- Recall: RISC-V register field positions consistent across instr. formats.
- RISC-V also tries to keep bit positions of immediates consistent:

	31	25 24	20 19	15 14	12 11	7	6	0
I-type	imm[11:5]	imm[4:0]	rs1	funct3	rd			opcode
S-type	yxxxxxx	wwwwv						
B-type	imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]			opcode

- Instruction Bit 31 is always the *sign bit* (highest bit to sign extend in immediate)
- B-type has a 13-bit immediate encoding b/c of implicit zero in Imm bit 0.

Between S and B, only two bits change meaning:  
Instruction Bit 7 → S: Imm Bit 0; B: Imm Bit 11  
Instruction Bit 31 → S: Imm Bit 11; B: Imm Bit 12

RISC-V immediate bit encoding is optimized to reduce hardware cost.

Berkeley

Garcia, Yan

## 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

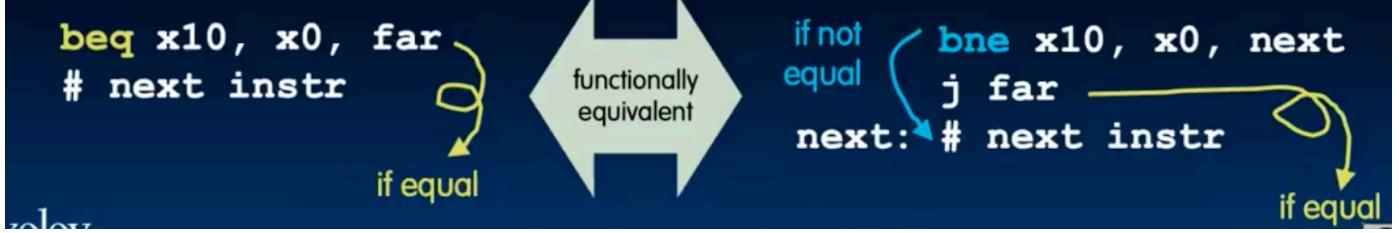
## 条件分支和无条件分支

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

## What if destination is further away?

- Enter the *unconditional jump!*



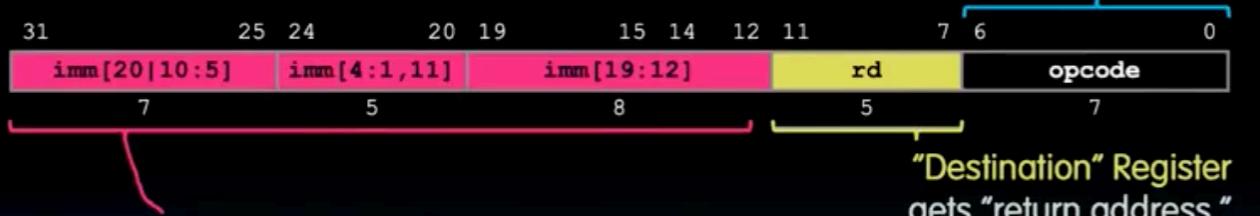
## J-Format (UJ-Format)

只用于 `jal`

适用于处理无条件的跳转

- J-Format (**textbook: UJ-Type**) used only for `jal`:

**jal rd, Label**



Immediate represents relative offset in increments of 2 bytes.

- To compute new PC:  $PC = PC + \text{byte\_offset}$ .
- 20 immediate bits imply  $\pm 2^{18}$  32-bit instructions reachable:
  - 1 bit: 2's complement (allow  $+$ / $-$  offset), 1 bit: 16-b instruction support
- Immediate bit encoding optimized to reduce HW cost

"Destination" Register gets "return address."

$$rd = PC + 4$$

What about jumping *further*?  
Next: How to load 32-bit immediate into a register!

## U-Format

- “Upper Immediate” instructions:

**opname rd, immed**

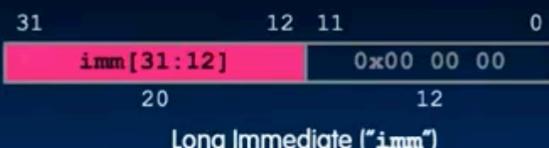
U-Format opcodes:  
 lui 1100011  
 auipc 1100011



“Destination” Register

Saves result of operation.

Immediate represents *upper 20 bits* of a 32-bit immediate operand  $\text{imm} = \text{immed} \ll 12$ .



`lui` 与 `addi` 配合可以加载32位的imme

**lui rd, immed**

- The **lui instruction**, Load Upper Immediate:

- Write a 20-bit immediate value into the upper 20-bits of register rd.
- Clear the lower 12 bits.

}       $rd = \text{immed} \ll 12$

- lui together with an addi (to set lower 12 bits) can create any 32-bit value in a register:**

```
lui x10, 0x87654      # x10 = 0x87654000
addi x10, x10, 0x321 # x10 = 0x87654321
```

The `li` pseudoinstruction (`Load Immediate`) resolves to `lui + addi` as needed, e.g., `li x10, 0x87654321`.

注意： `addi` 会扩展符号位

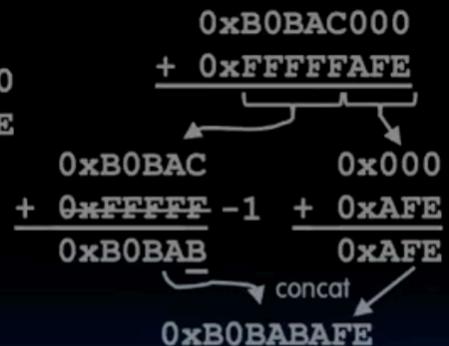
## ▪ How should we set the immediate 0xB0BACAFE?



- Unfortunately:

```
lui x10, 0xB0BAC      # x10 = 0xB0BAC000
addi x10, x10, 0xAF   # x10 = 0xB0BABAFE
```

- Recall: **addi** sign-extends the 12-bit immediate.
- If "sign bit" set, subtracts 1 from the upper 20-bits!



## ✓ Solution: If 12-bit immediate is negative, add 1 to the upper 20-bit load.

```
lui x10, 0xB0BAD      # x10 = 0xB0BAD000
addi x10, x10, 0xAF   # x10 = 0xB0BACAFE
```

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 rd, imm**

- The **auipc** instruction      }       $rd = PC + (imm << 12)$
- Adds an Upper Immediate to the PC.      }
- Example:**

```
auipc x5, 0xABCD     # x5 = PC + 0xABCD
```

- In Practice:**

Label: `auipc x5, 0` # puts address of Label in x5

- Loads the PC into a register accessible by other instructions.

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

## `jalr`: I-Format

- jalr does a Jump And Link Register:**

**jalr rd, rs1, imm**

- Jump to `rs1 + imm`.      }       $PC = rs1 + imm$
- Write address of the *following* instruction to `rd`.      }       $rd = PC + 4$

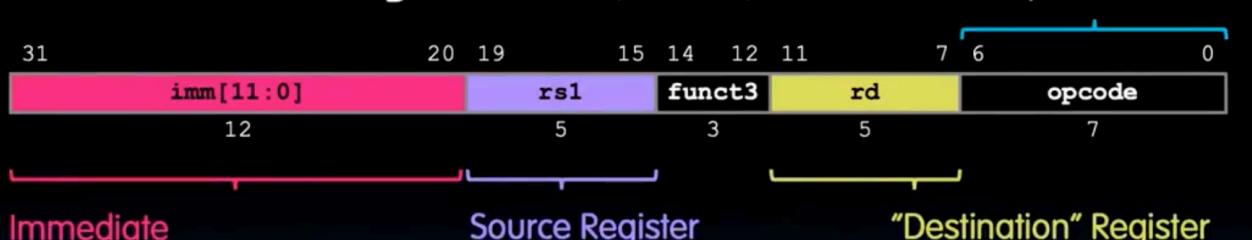
- Use cases (so far):**

- Return to callee

`jr ra`       `jalr x0, ra, 0`

- jalr uses I-Format:

**jalr rd,rs1,imm**



Immediate

Source Register

"Destination" Register  
gets "return address."

$\text{imm}$  and  $\text{rs1}$  are added together to update PC.

$$\text{PC} = \text{rs1} + \text{imm}$$

⚠ Note I-Type!

Unlike B-type, J-type, jalr will not multiply  $\text{imm}$  by 2.

◦ PC relative offset therefore must be written in *units of bytes*.

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)

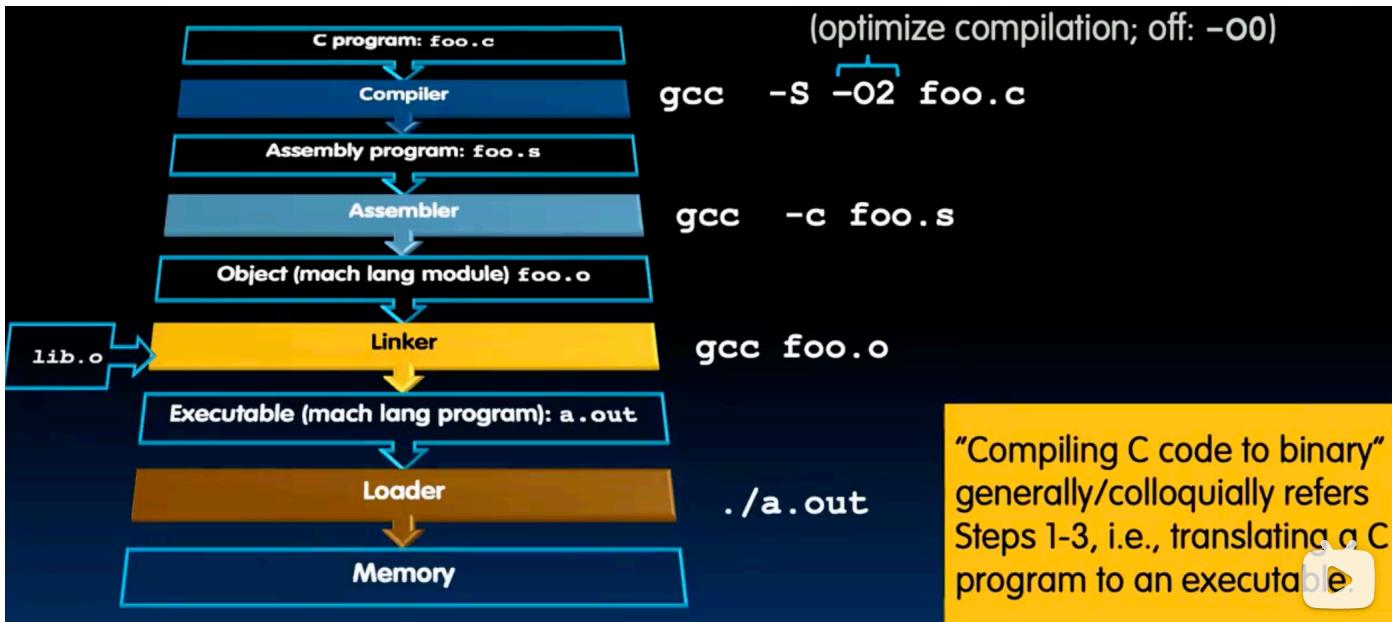
介绍两个新的伪指令：

`la (load address)` 与 `call label`

<code>la rd, label</code> (address <code>addr</code> )	# absolute address      # PC-relative address
	<code>lui rd, addr[31:12]</code> <code>auipc rd, addr[31:12]</code>
<code>call label</code> (address <code>addr</code> )	<code>addi rd, rd, addr[11:0]</code> <code>addi rd, rd, addr[11:0]</code>
	<code>auipc ra, addr[31:12]</code> <code>j far_away</code> , e.g., in a different file.
	<code>jalr ra, addr[11:0]</code>

## 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 compiler
- do not produce machine instructions. Rather, they inform how to build different parts of the *object file*

<b>.text</b>	Subsequent items put in user Text segment (machine code)
...	
<b>.data</b>	Subsequent items put in user Data segment (source file data in binary)
...	
<b>.globl sym</b>	Declares sym global and can be referenced from other files
<b>.string str</b>	Store the string str in memory and null-terminate it
<b>.word w1 ... wn</b>	Store the n 32-bit quantities in successive memory words

## 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` from 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

- **List of “items” in this file**
- **Instruction Labels**
  - Used to compute machine code for PC-relative addressing in branches, function calling, etc.
  - `.global` directive: labels can be referenced by other files.
- **Data: anything in the `.data` section**
  - Global variables may be accessed/used by other files.

## 重定位信息

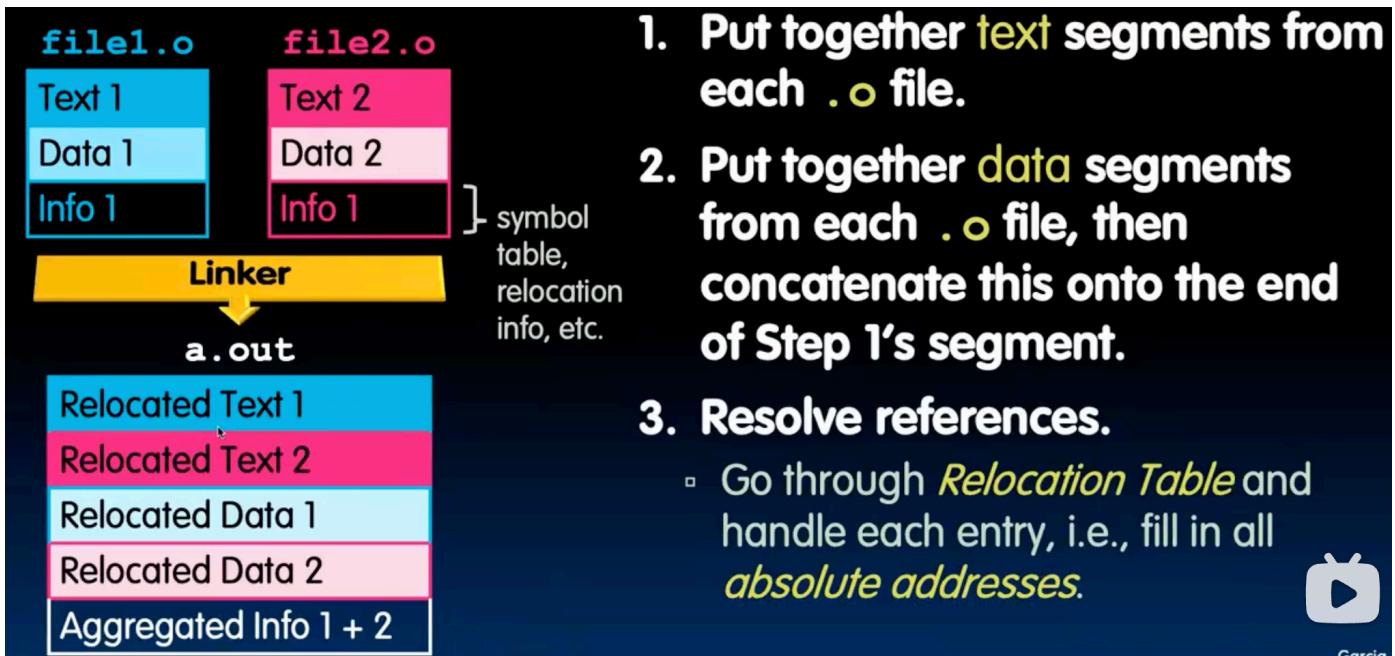
- List of “items” whose address this file needs
- Any external label jumped to
  - External label (including lib files): `jal ext_label`
- Any piece of data in static section
  - e.g., `la` instruction (for `lw/sw` base register)

## 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

## ▪ Load program into a newly created address space:

- Read executable's file header for sizes of text, data segments.
- Create new address space for program large enough to hold text and data segments, along with a stack segment.
  - Copy instructions, data from executable file into new address space.
  - Copy arguments passed to the program onto the stack.

## ▪ Initialize machine registers

- Most registers cleared; stack pointer (**sp**) assigned address of first free stack location

## ▪ Jump to start-up routine, which does the following:

- Copy program arguments from stack to registers, set PC
- If main routine returns, terminate program with exit system call.



Garcia

## Example

<pre> text .align 2 .globl main main:     addi sp,sp,-4     sw ra,0(sp)     la a0, str1     la a1, str2     call printf     lw ra,0(sp)     addi sp,sp,4     li a0,0     ret section .rodata .balign 4 str1:     .string "Hello, %s!\n" str2:     .string "world"     . </pre>	<p>         Directive: Enter <b>text</b> section          Directive: Align code to <b>2^2</b> bytes          Directive: Declare global symbol <b>main</b>          Label for start of <b>main</b>          Allocate stack frame,          save return address  <b>Pseudoinstructions:</b>          load addresses of <b>str1</b>, <b>str2</b>  <b>Pseudoinstruction:</b> call function <b>printf</b>          Restore return address,          deallocate stack frame          Return with value 0       </p>	<pre> #include &lt;stdio.h&gt; int main() {     printf("Hello, %s\n",            "world");     return 0; } </pre>
--	---	---



Garcia

Text segment

```

00000000 <main>:
0: ff010113 addi sp,sp,-16
4: 00112623 sw ra,12(sp)
8: 00000537 lui a0,0x0
c: 00050513 addi a0,a0,0
10: 000005b7 lui a1,0x0
14: 00058593 addi a1,a1,0
18: 00000097 auipc ra,0x0
1c: 000080e7 jalr ra,0
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 00008067 jalr rac

```

Object file (Assembler output)

...	ff010113	00112623	00000537
00050513	000005b7	00058593	00000097
000080e7	00c12083	01010113	00000513
00008067	...		

Symbol Table

Label	Address (in module)	Type
main:	0x00000000	global text
str1:	0x00000000	local data
str2:	0x0000000c	local data

Relocation Information

Address	Type	Dependency
0x00000008	lui	%hi(str1)
0x0000000c	addi	%low(str1)
0x00000010	lui	%hi(str2)
...	...	...

```

000101b0 <main>:
101b0: ff010113 addi sp,sp,-16
101b4: 00112623 sw ra,12(sp)
101b8: 00021537 lui a0,0x21
101bc: a1050513 addi a0,a0,-1520 # 20a10 <str1>
101c0: 000215b7 lui a1,0x21
101c4: a1c58593 addi a1,a1,-1508 # 20a1c <str2>
101c8: 288000ef jal ra,10450 # <printf>
101cc: 00c12083 lw ra,12(sp)
101d0: 01010113 addi sp,sp,16
101d4: 00000513 addi a0,0,0
101d8: 00008067 jalr ra

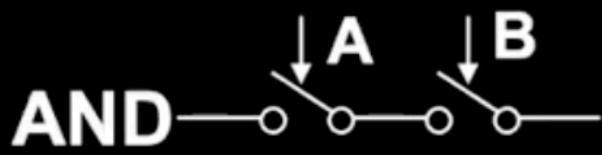
```

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

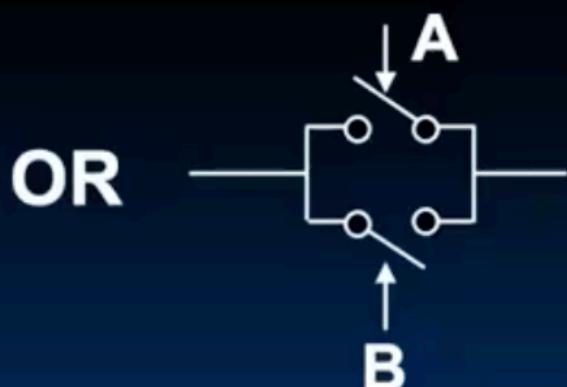
### Switches 开关

#### Basic Element of Physical Circuit

构建更加复杂的逻辑



$Z \equiv A \text{ and } B$



$Z \equiv A \text{ or } B$

## The Transistor("born" 1947-12-23) 晶体管

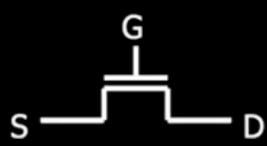
Modern digital systems designed in CMOS

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

MOS transistors act as voltage-controlled switched

- Three terminals: Drain, Gate, Source
  - Switch action: **Dan Garcia Says**  
if voltage on gate terminal is (some amount) higher/lower than source terminal then conducting path established between drain and source terminals

To remember:  
n ("normal")  
p (has a circle,  
like the top  
part of P itself)



n-channel  
open when voltage at G is low  
closes when:  
 $voltage(G) > voltage(S) + \epsilon$

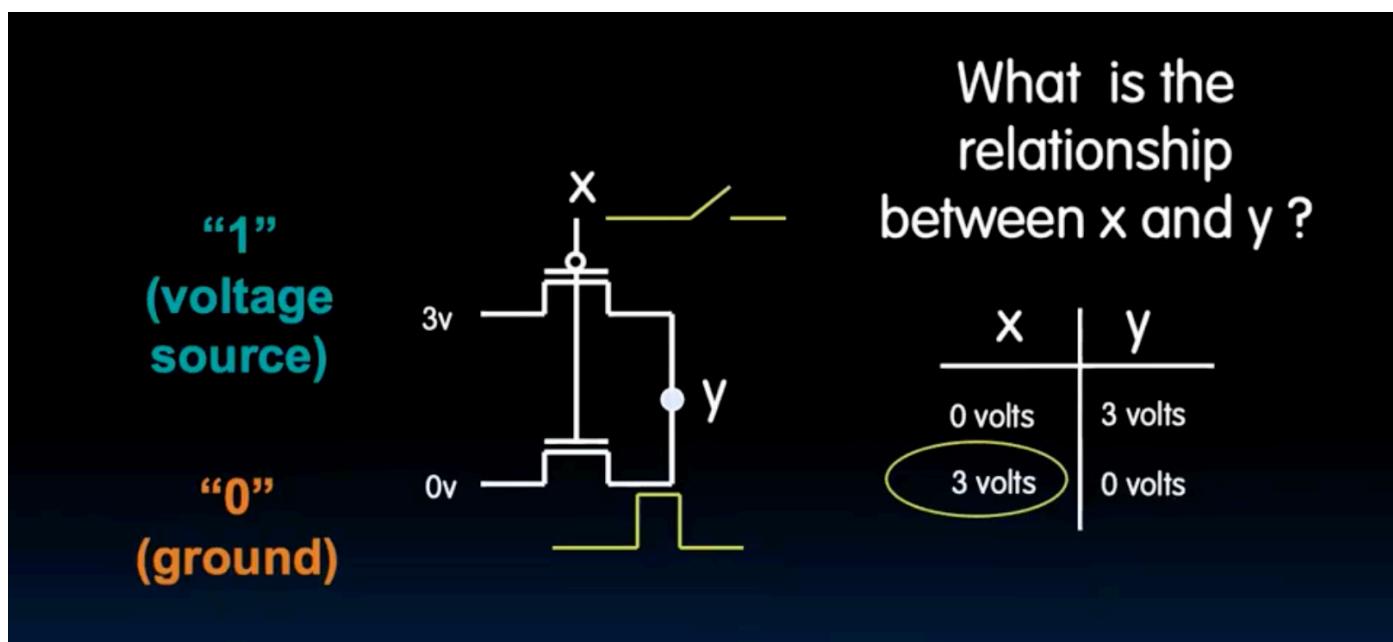


p-channel  
closed when voltage at G is low  
opens when:  
 $voltage(G) > voltage(S) + \epsilon$



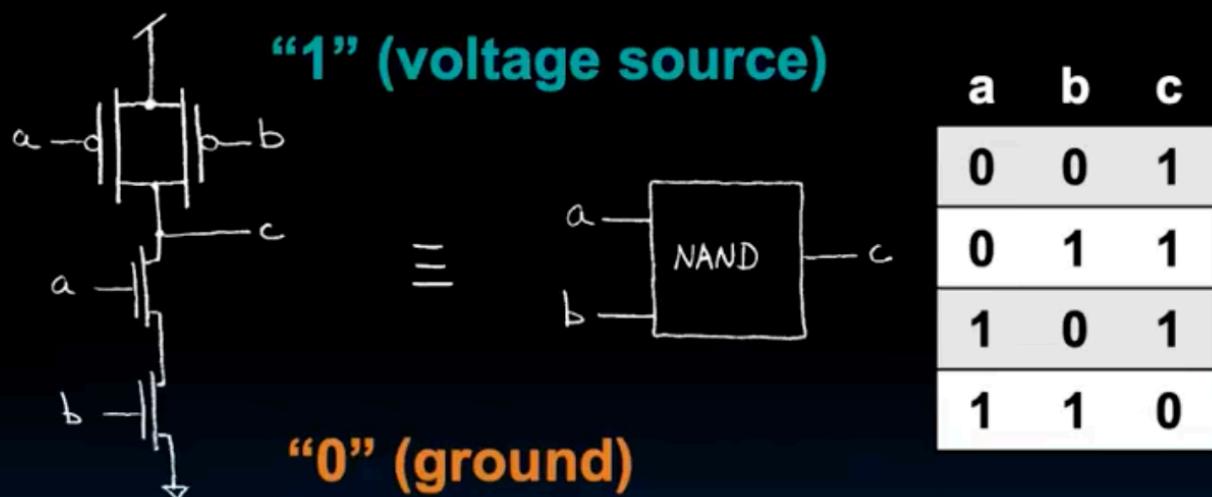
## MOS Network

A NOT gate



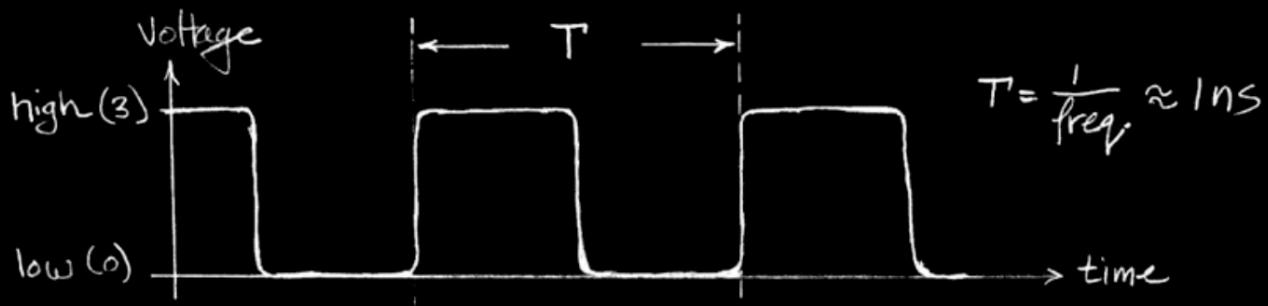
NAND gate (只有都是1时输出为0)

- Chips are composed of nothing but transistors and wires.
- Small groups of transistors form useful building blocks.



- Block are organized in a hierarchy to build higher-level blocks: ex: adders.
- You can build AND, OR, NOT out of NAND!

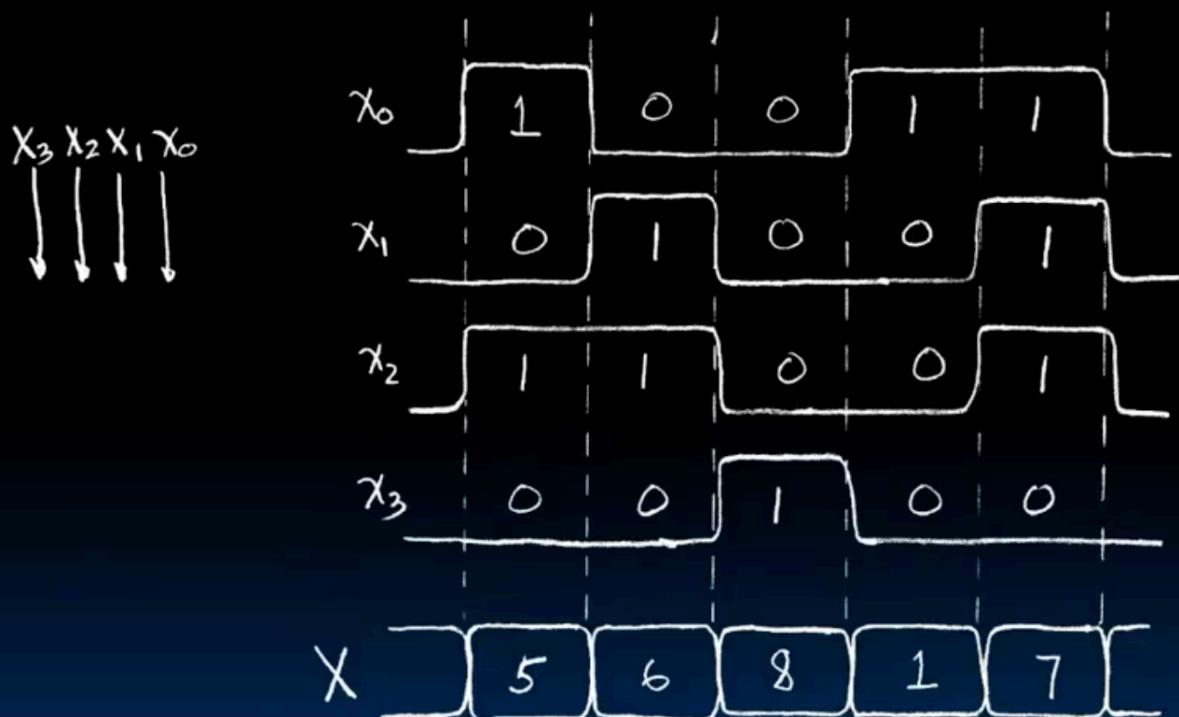
## Signals and Waveforms: Clocks



## ▪ Signals

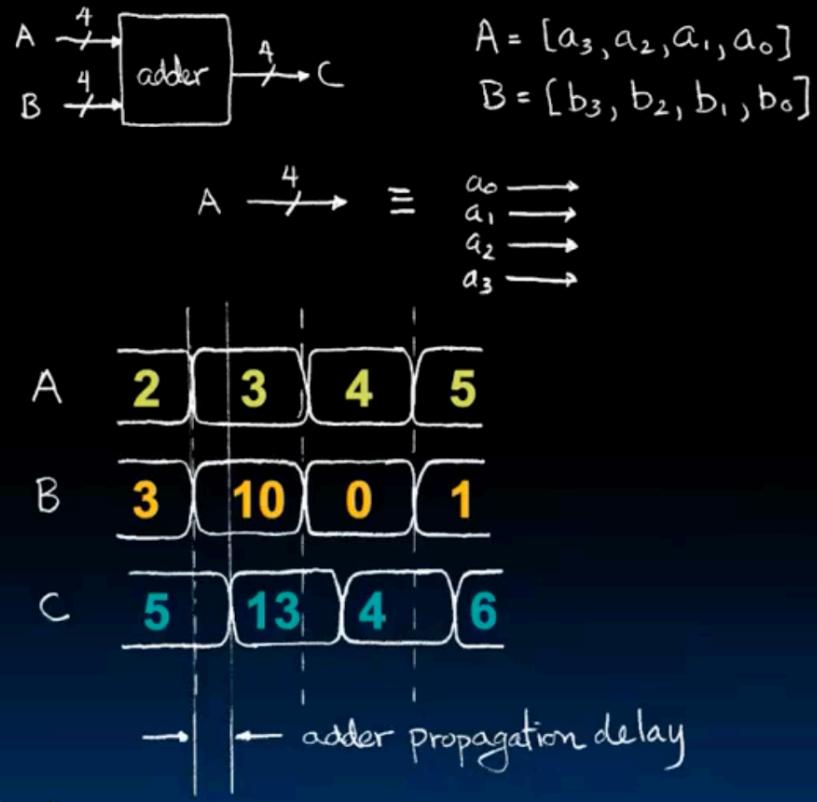
- When **digital** is only treated as 1 or 0
- Is transmitted over wires continuously
- Transmission is effectively instant
- Implies that a wire contains 1 value at a time

Grouping



$x_0 - x_3$  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

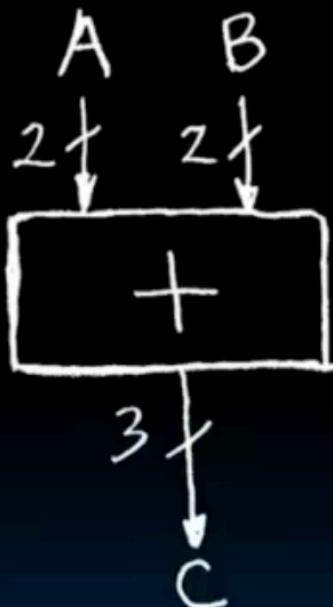
### 条件相反器

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	B	C
$a_1a_0$	$b_1b_0$	$c_2c_1c_0$
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



ab	c
00	0
01	0
10	0
11	1

ab	c
00	0
01	1
10	1
11	1

a	b
0	1
1	0

XOR NAND NOR

XOR



ab	c
00	0
01	1
10	1
11	0

NAND



ab	c
00	1
01	1
10	1
11	0

NOR



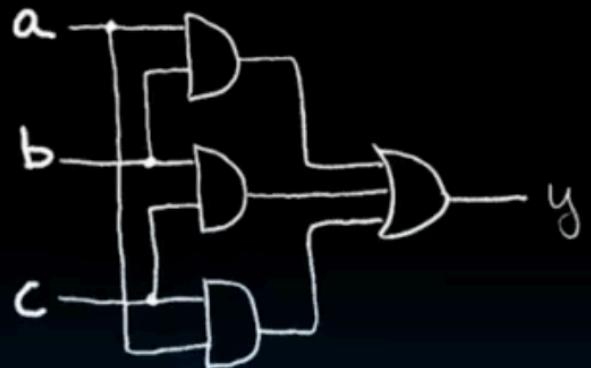
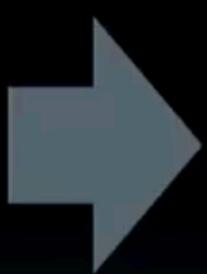
ab	c
00	1
01	0
10	0
11	0

2-input gates extend to n-inputs

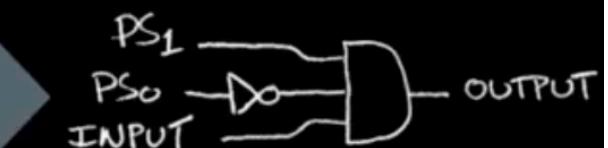
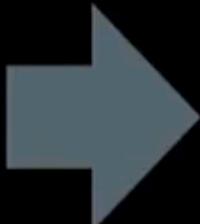
依次进行真值判断即可

Truth Table -> Gates

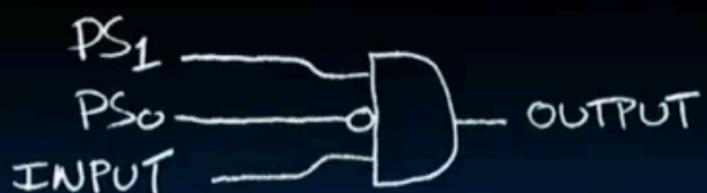
a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



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...

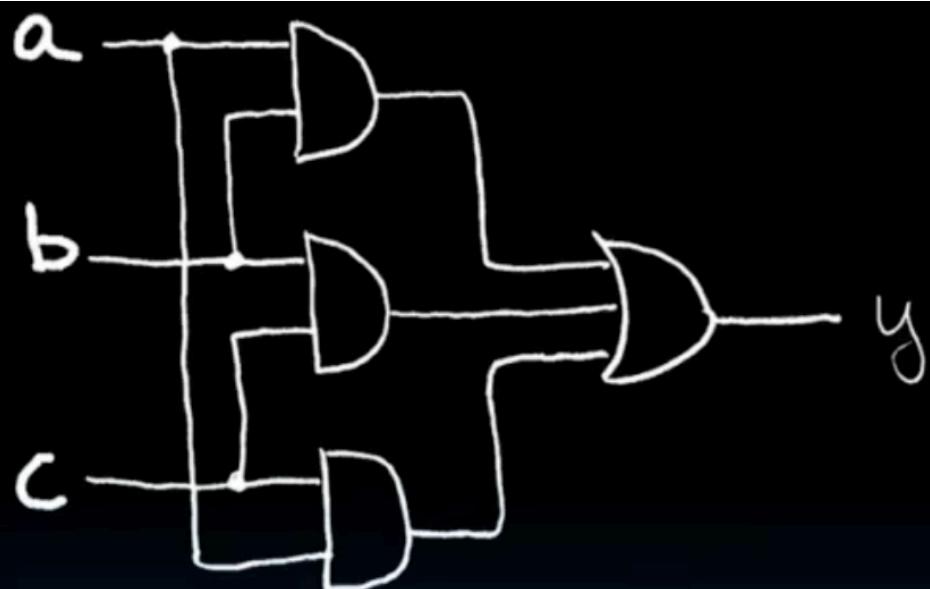


## Boolean Algebra

George Boole, 19th Century mathematician

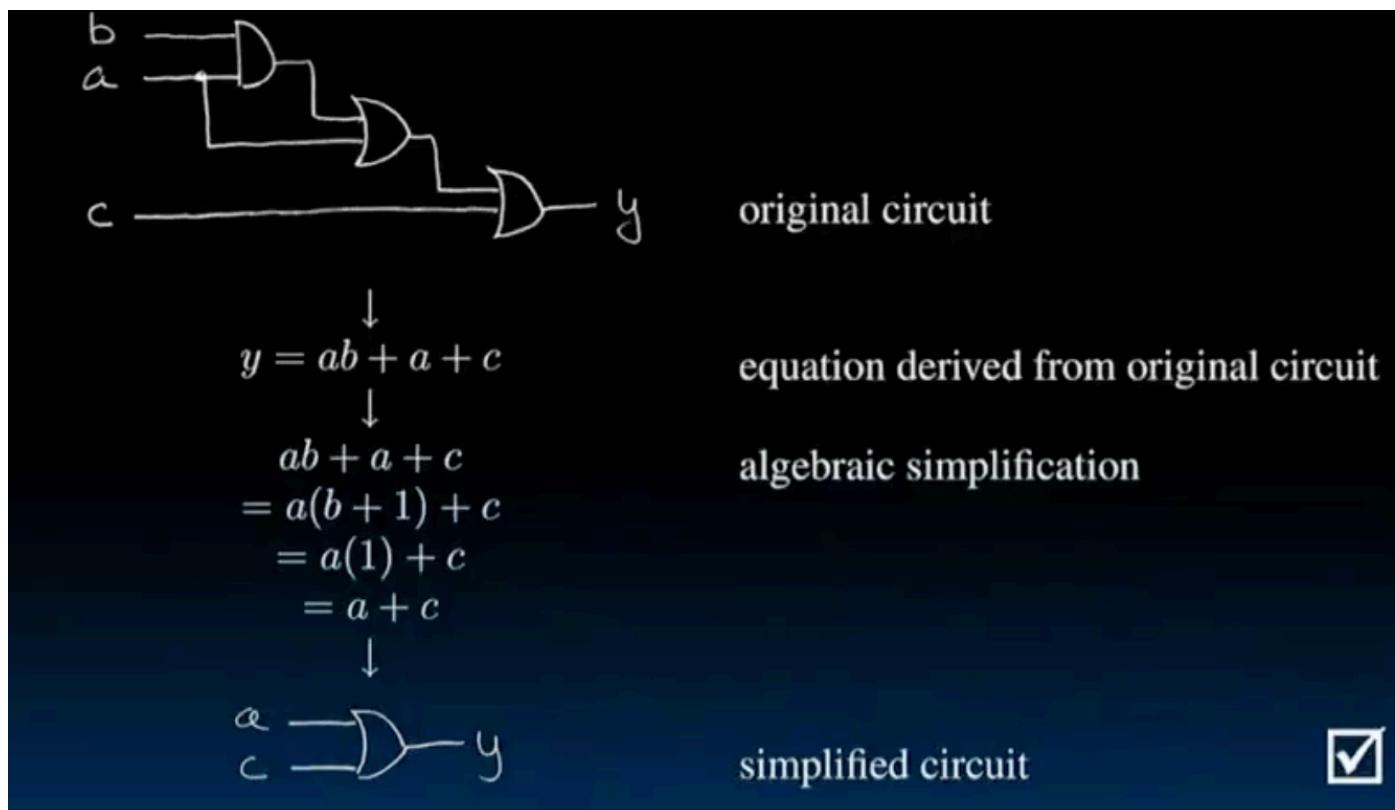
- $+$  means OR,  $\bullet$  means AND,  $\bar{x}$  means NOT

举例



$$y = a \bullet b + a \bullet c + b \bullet c$$

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

$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$	complementarity
$x \cdot 0 = 0$	$x + 1 = 1$	laws of 0's and 1's
$x \cdot 1 = x$	$x + 0 = x$	identities
$x \cdot x = x$	$x + x = x$	idempotent law 
$x \cdot y = y \cdot x$	$x + y = y + x$	commutative law
$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$	associativity
$x(y+z) = xy + xz$	$x+yz = (x+y)(x+z)$	distribution
$xy + x = x$	$\frac{(x+y)x}{(x+y)} = \bar{x} \cdot \bar{y}$	uniting theorem
$\bar{x} \cdot \bar{y} = \bar{x} + \bar{y}$		DeMorgan's Law

## Canonical forms 标准格式

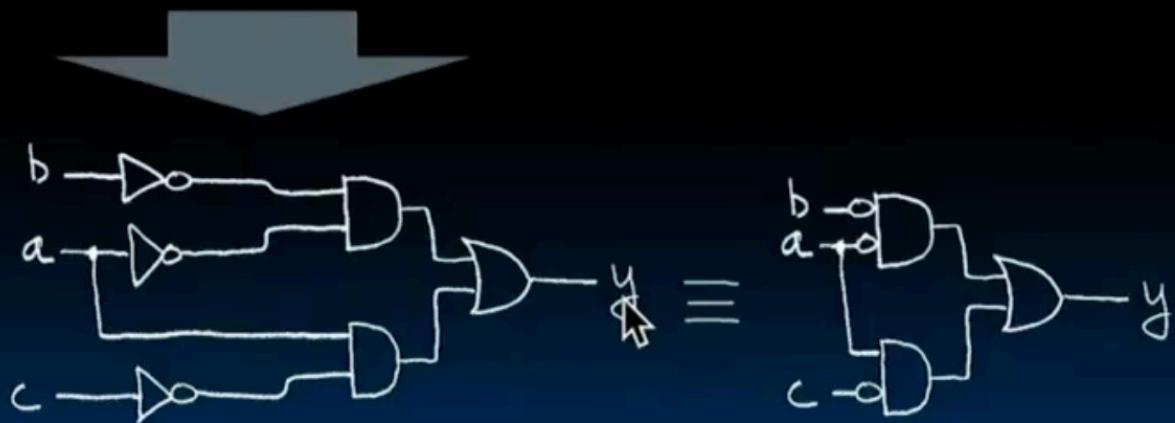
	abc	y	Sum-of-products (ORs of ANDs)
$\bar{a} \cdot \bar{b} \cdot \bar{c}$	000	1	
$\bar{a} \cdot \bar{b} \cdot c$	001	1	
	010	0	
	011	0	
$a \cdot \bar{b} \cdot \bar{c}$	100	1	$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$
	101	0	
$a \cdot b \cdot \bar{c}$	110	1	
	111	0	

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)

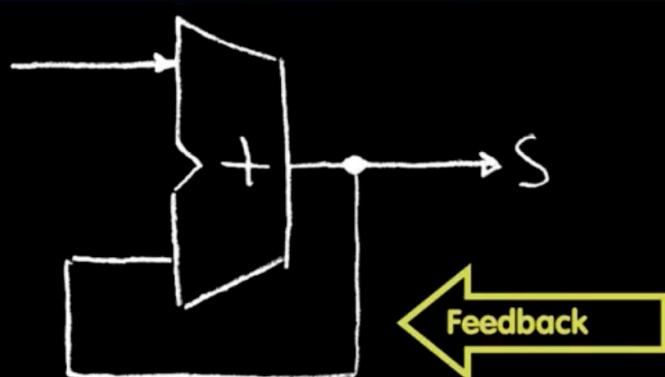
使用析取范式是因为，析取范式包含了可以使电路为真的所有可能，用或链接，只要满足一个我们的电路就可以连通

$$\begin{aligned}y &= \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c} \\&= \bar{a}\bar{b}(\bar{c} + c) + a\bar{c}(\bar{b} + b) \quad \text{distribution} \\&= \bar{a}\bar{b}(1) + a\bar{c}(1) \quad \text{complementarity} \\&= \bar{a}\bar{b} + a\bar{c} \quad \text{identity}\end{aligned}$$



## Lecture 16 : Synchronous Digital System (SDS) State Accumulator

First try... Does this work?



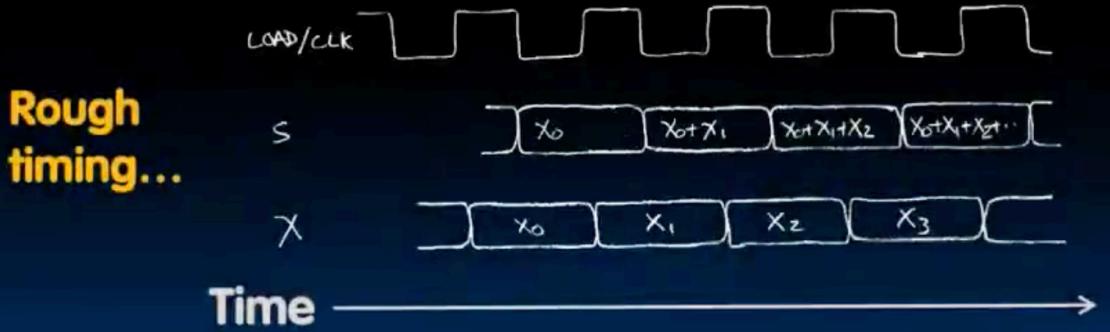
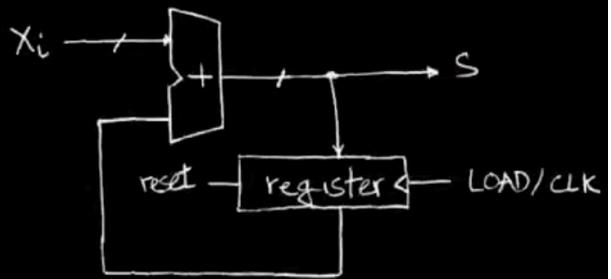
- **Nope!**

- Reason #1... What is there to control the next iteration of the '**for**' loop?
- Reason #2... How do we say: '**S=0**'?

没法控制循环的结束，也没办法将S进行重置

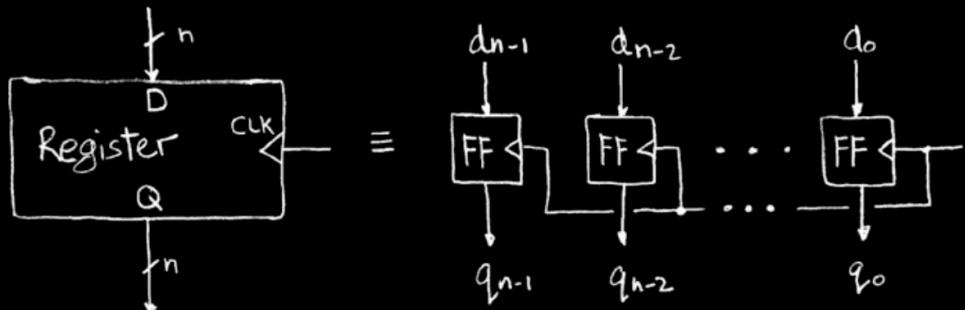
## Second try...How about this?

- Register is used to hold up the transfer of data to adder.



## Register Details Flip-flops(触发器)

## Register Details...What's inside?



- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between 0,1
- D is "data", Q is "output"
- Also called "D-type Flip-Flop"

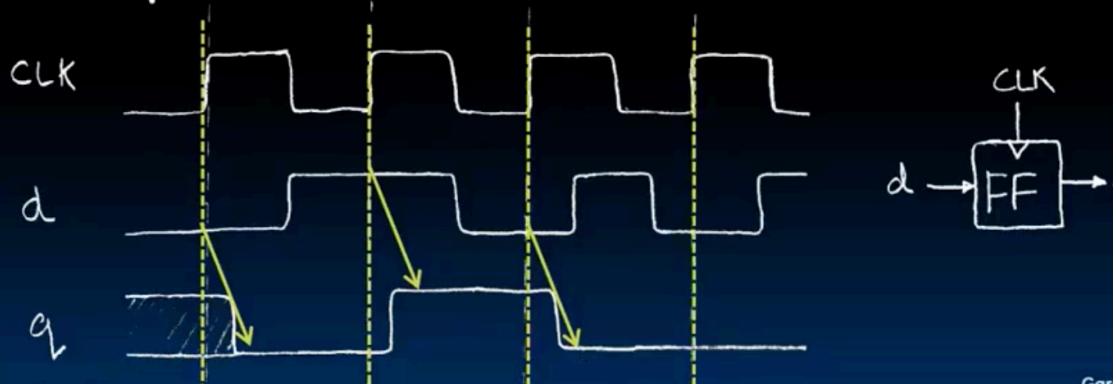
所以不管它代表什么，静态也好，Q就是输出。

D (data) 是输入

Q是输出

# What's the timing of a Flip-flop? (1/2)

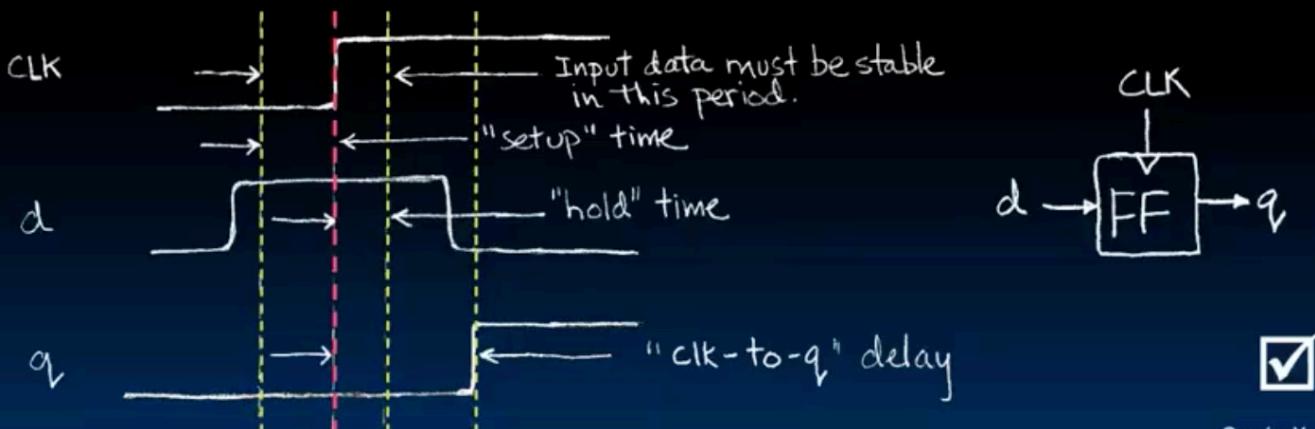
- Edge-triggered d-type flip-flop
  - This one is "rising edge-triggered"
  - Also called "positive edge"
- "On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored."
- Example waveforms:



There also exist  
"falling edge" FFs

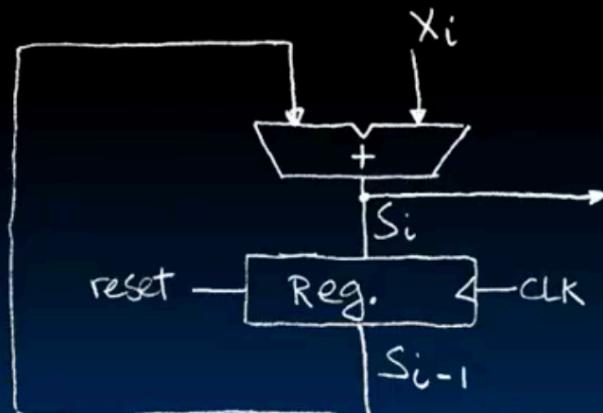
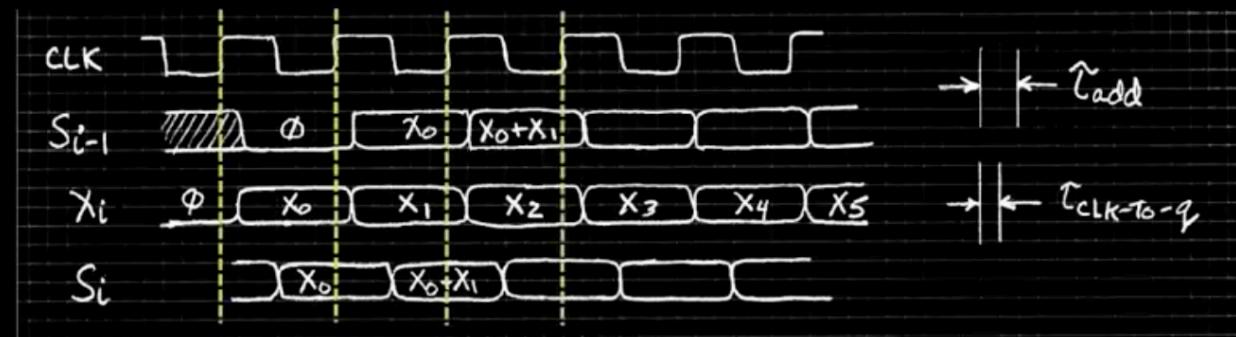
若d端输入在clock信号上沿之前的一段时间(建立时间)和之后的一段时间(维持时间)保持稳定不变，那么保证q端输出是稳定的，否则输入无效

- Example waveforms (more detail):



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

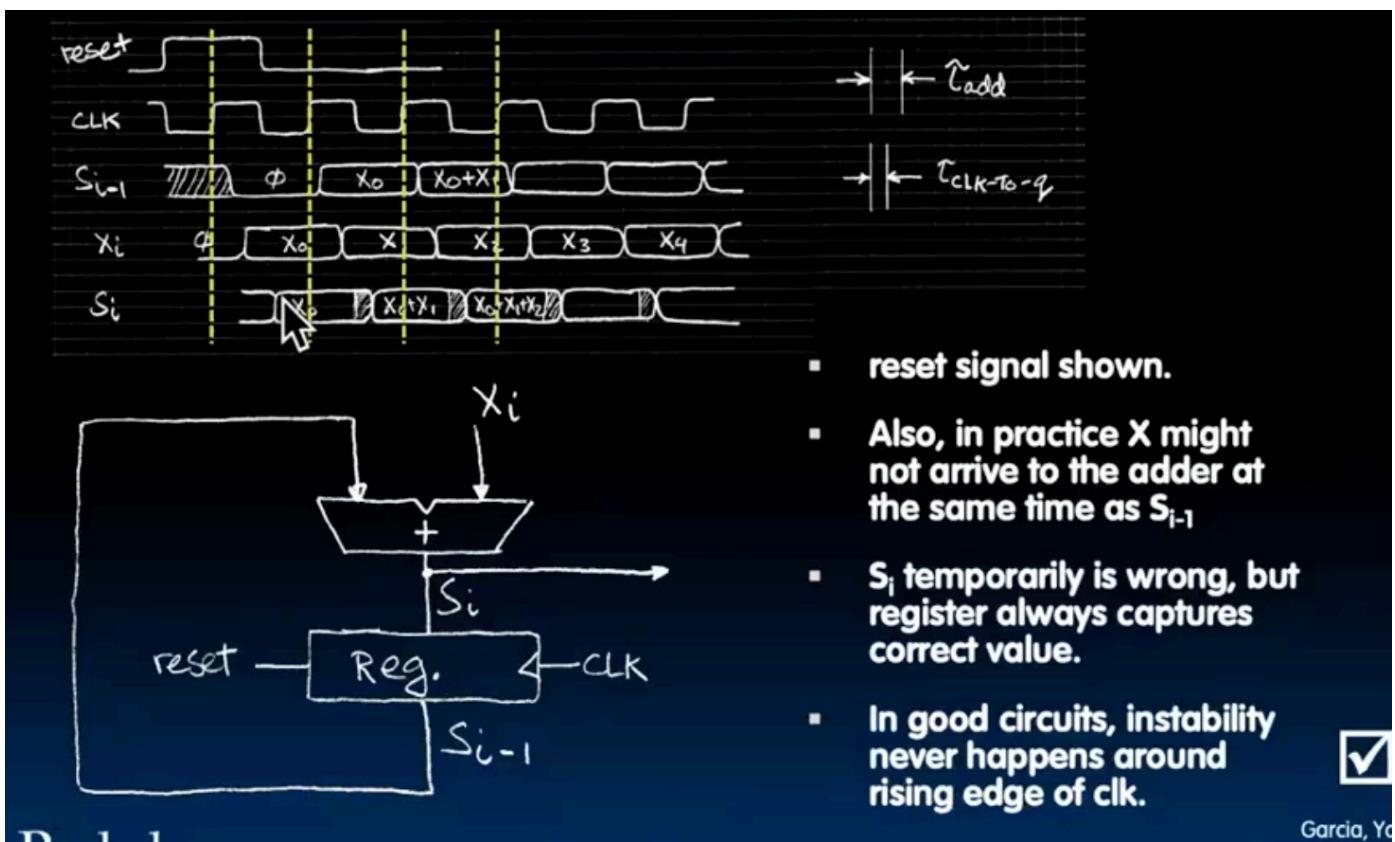


- Reset input to register is used to force it to all zeros (takes priority over D input).
- $S_{i-1}$  holds the result of the  $i^{th}-1$  iteration.
- Analyze circuit timing starting at the output of the register.

实际系统现在工作了，让我们再深入一点细节，这里有

Garcia, Yo

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



- reset signal shown.
- Also, in practice X might not arrive to the adder at the same time as  $S_{i-1}$ .
- $S_i$  temporarily is wrong, but register always captures correct value.
- In good circuits, instability never happens around rising edge of clk.

Rule 1

Garcia, Yo

注意：在第二条黄线的右侧一点点，计算结果是 $x_0 + x_0$ 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 $s_{i-1}$

## Pipelining for Performance 流水线操作

## 最大化时间效率

period 周期

C-to-Q delay 是指：

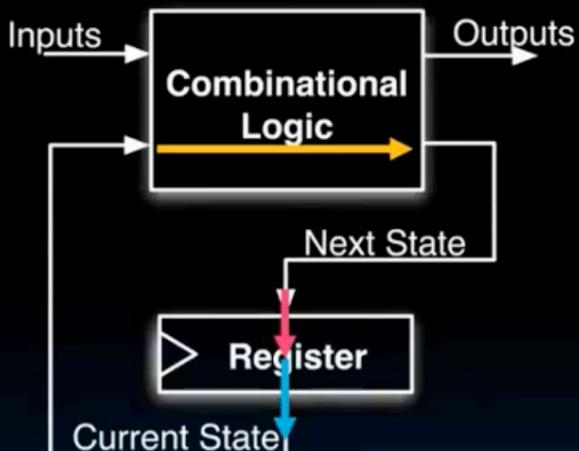
- 时钟到来时，它会采样输入 D；
- 然后经过一段硬件延迟，把这个值传递到输出 Q；
- 这个延迟过程就是 C-to-Q delay。

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)。

▪ What is the maximum clock frequency of this circuit? (Hint: Frequency = 1 / Period )



$$\text{Max Delay} = \text{CLK-to-Q Delay} + \text{CL Delay} + \text{Setup Time}$$

D. 1.1

Garcia, Yo

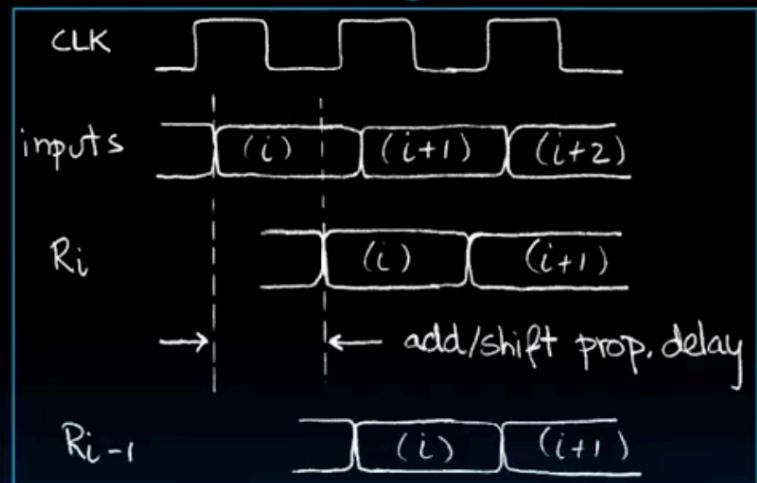
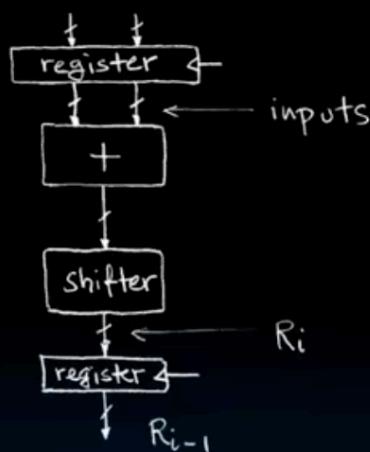
如果电路的周期小于最大延迟，会导致建立时间之前无法稳定，导致超频崩溃

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.

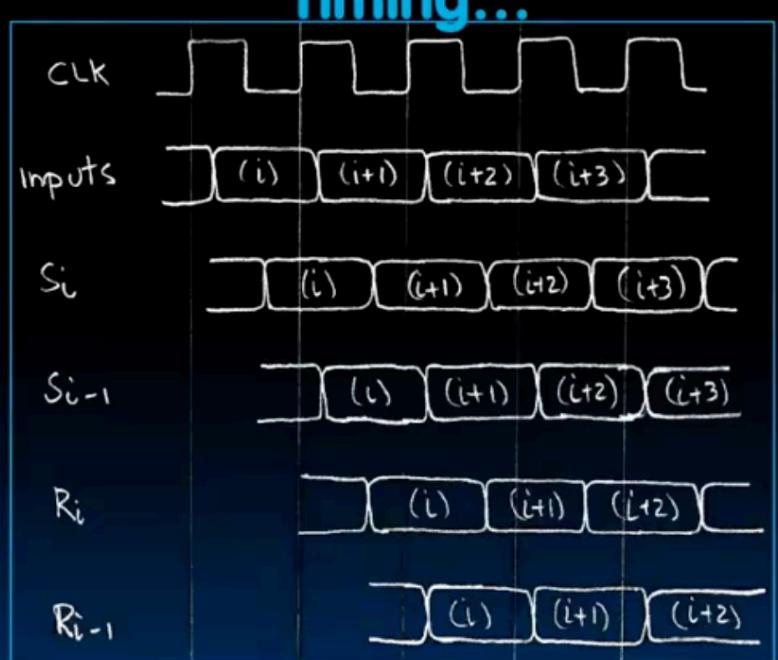
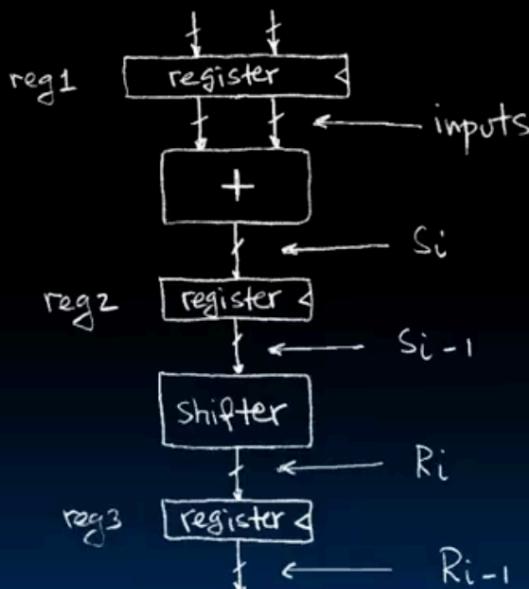


- Note: Delay of 1 clock cycle from input to output.
- Clock period limited by propagation delay of adder/shifter.

D. 1.1

Garcia, Yo

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



D. 1.1

Garcia, Yo

Finite State Machines (FSM)

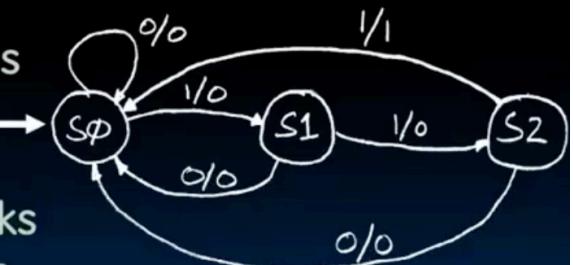
# Finite State Machine Example: 3 ones...

- **FSM to detect the occurrence of 3 consecutive 1's in the input.**



- **Draw the FSM...**

- Assume state transitions are controlled by the clock: on each clock cycle the machine checks the inputs and moves to a new state and produces a new output...



Berkeley  
UNIVERSITY OF CALIFORNIA

State (25)

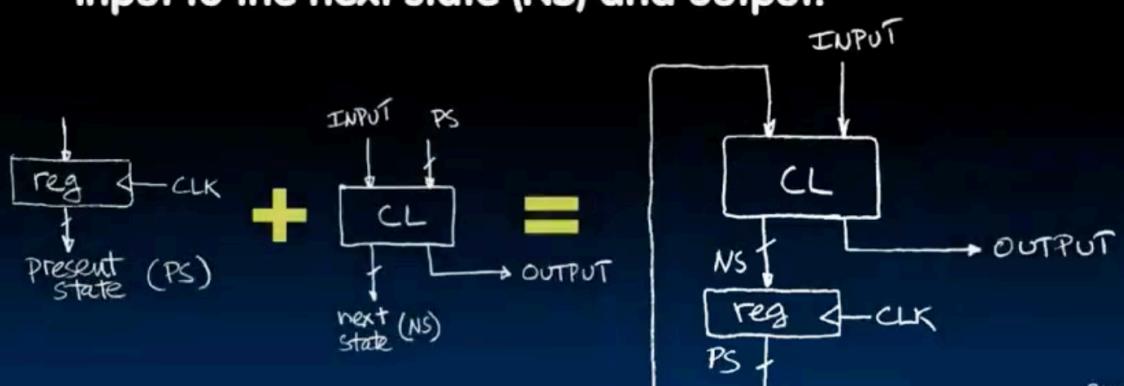


这台 FSM 就是一个能记住最近输入的 0/1 情况的机器，当发现连续三个 1 时就发出一个 1 的信号

如何实现呢

## Hardware Implementation of FSM

- ... Therefore a register is needed to hold the representation of which state the machine is in.
  - Use a unique bit pattern for each state.
- Combinational logic circuit is used to implement a function mapping the input and present state (PS) input to the next state (NS) and output.



Berkeley  
UNIVERSITY OF CALIFORNIA

State (26)



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。

## Hardware for FSM: Combinational Logic

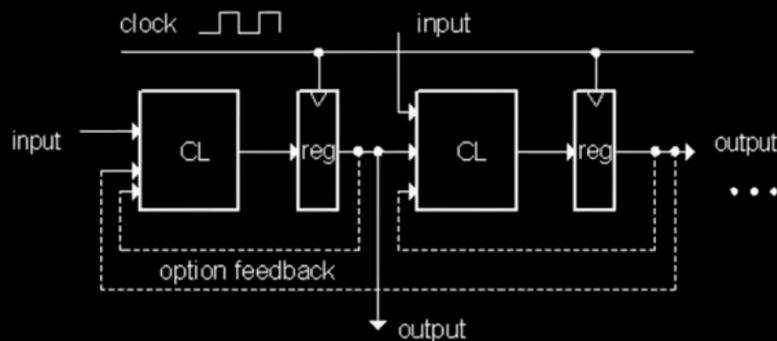
▪ Next lecture we will discuss the detailed implementation, but for now can look at its functional specification, truth table form.

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

如果我在状态 1 并且得到一个 0, 我重置, 因为每个  
Berkeley If I'm at one and zero, I reset, because every zero I reset, so I go back.

CC BY Garcia

# General Model for Synchronous Systems

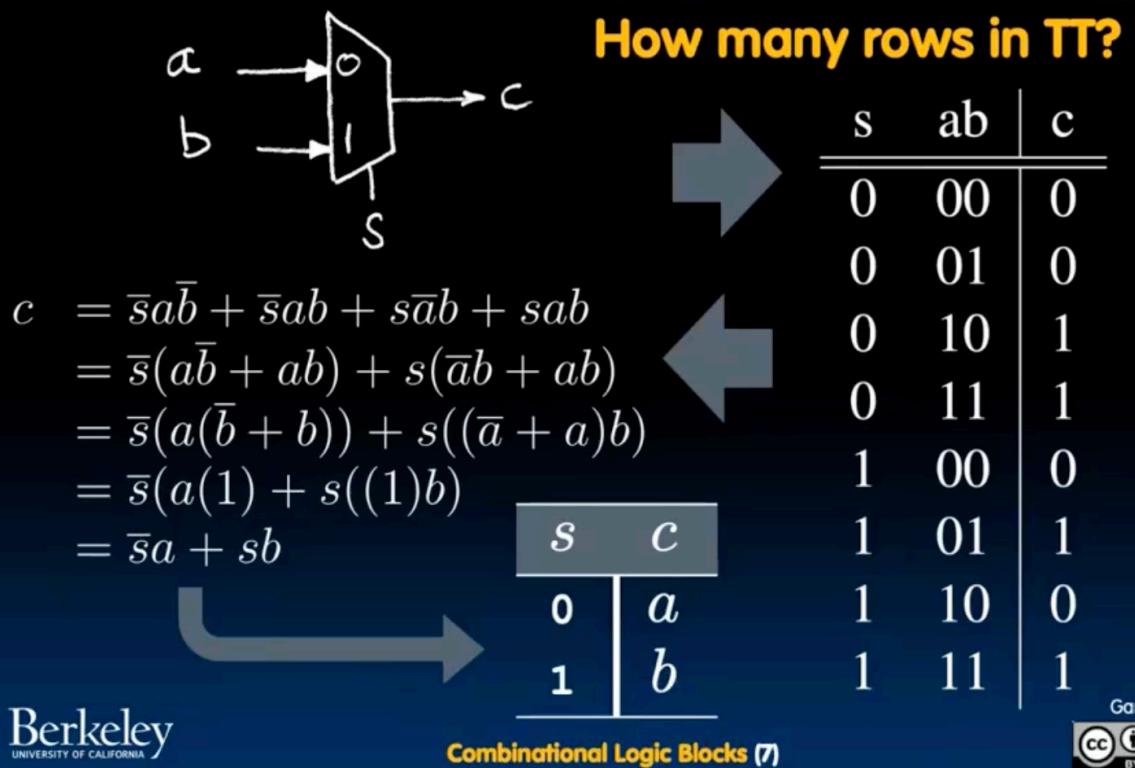


- **Collection of CL blocks separated by registers.**
- **Registers may be back-to-back and CL blocks may be back-to-back.**
- **Feedback is optional.**
- **Clock signal(s) connects only to clock input of registers.**

## Lecture 17 : Combinational Logic Blocks

Data Multiplexors 多路复用器

# N instances of 1-bit-wide mux

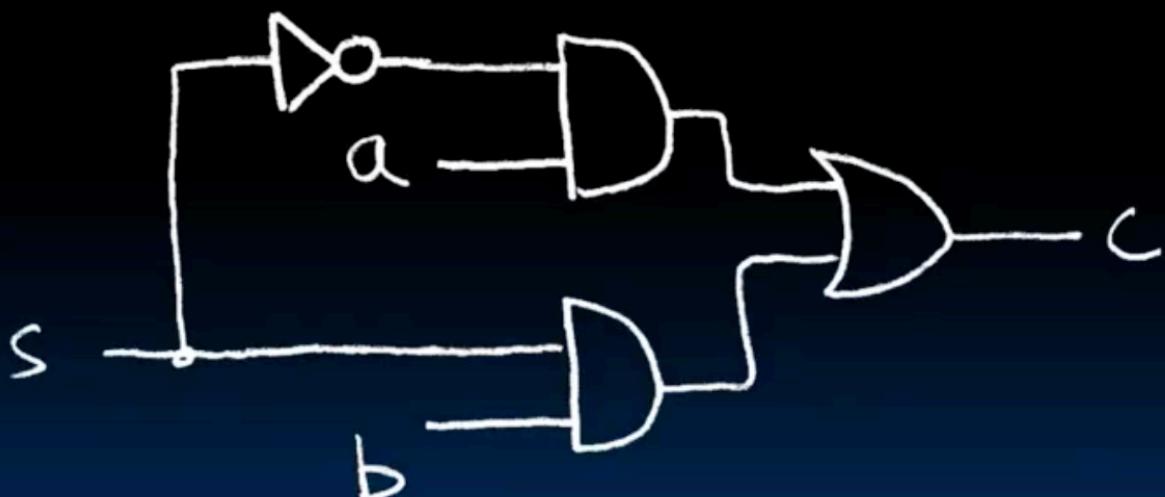


Berkeley

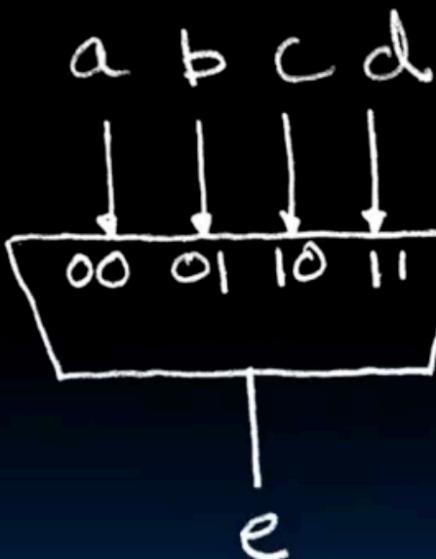
Combinational Logic Blocks (7)



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



再次转化形式



when  $S=00, e=a$   
when  $S=01, e=b$   
when  $S=10, e=c$   
when  $S=11, e=d$

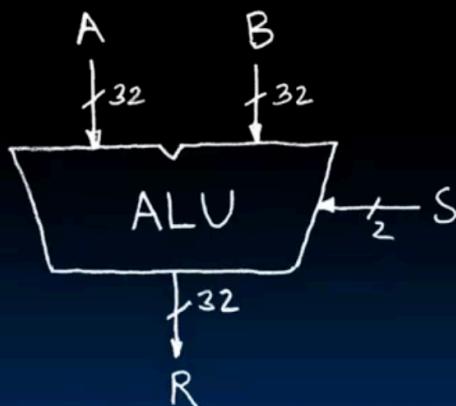
$s_1 s_0$	$c$
0 0	$a$
0 1	$b$
1 0	$c$
1 1	$d$

$$e = \overline{s_1} \cdot \overline{s_0}a + \overline{s_1}s_0b + s_1\overline{s_0}c + s_1s_0d$$

## Arithmetic Logic Unit (ALU)

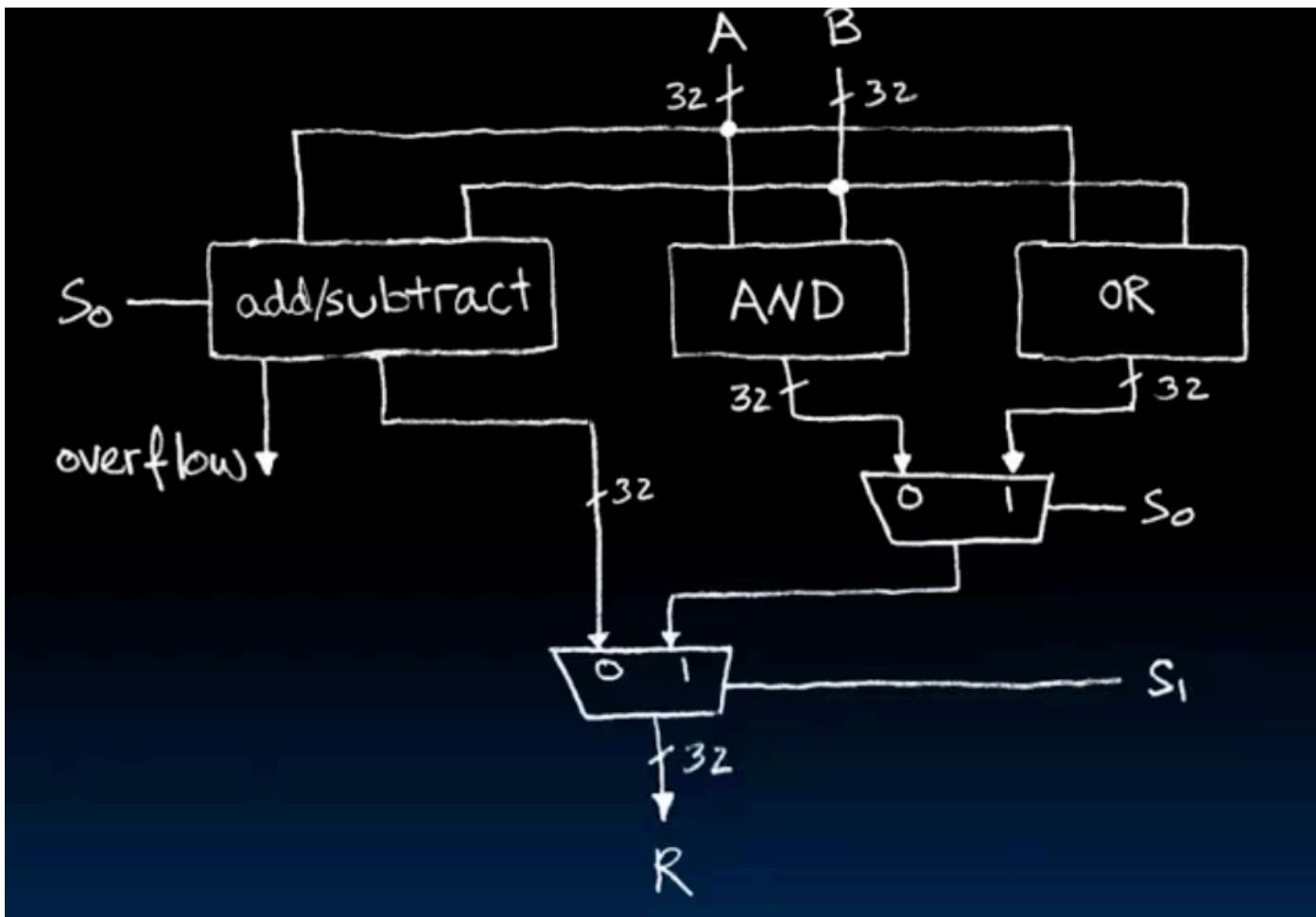
# Arithmetic and Logic Unit

- Most processors contain a special logic block called "Arithmetic and Logic Unit" (ALU)
- We'll show you an easy one that does ADD, SUB, bitwise AND ( $\&$ ), bitwise OR ( $|$ )



when  $S=00, R=A+B$   
when  $S=01, R=A-B$   
when  $S=10, R=A\&B$   
when  $S=11, R=A|B$

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

$$\begin{array}{r}
 & a_3 & a_2 & a_1 & \left| \begin{matrix} a_0 \\ b_0 \end{matrix} \right. \\
 + & b_3 & b_2 & b_1 & \hline
 & s_3 & s_2 & s_1 & \left| \begin{matrix} s_0 \end{matrix} \right.
 \end{array}$$

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

$$\begin{aligned}
 s_0 &= a_0 \text{ XOR } b_0 \\
 c_1 &= a_0 \text{ AND } b_0
 \end{aligned}$$

XOR 1的个数是奇数, 结果是1

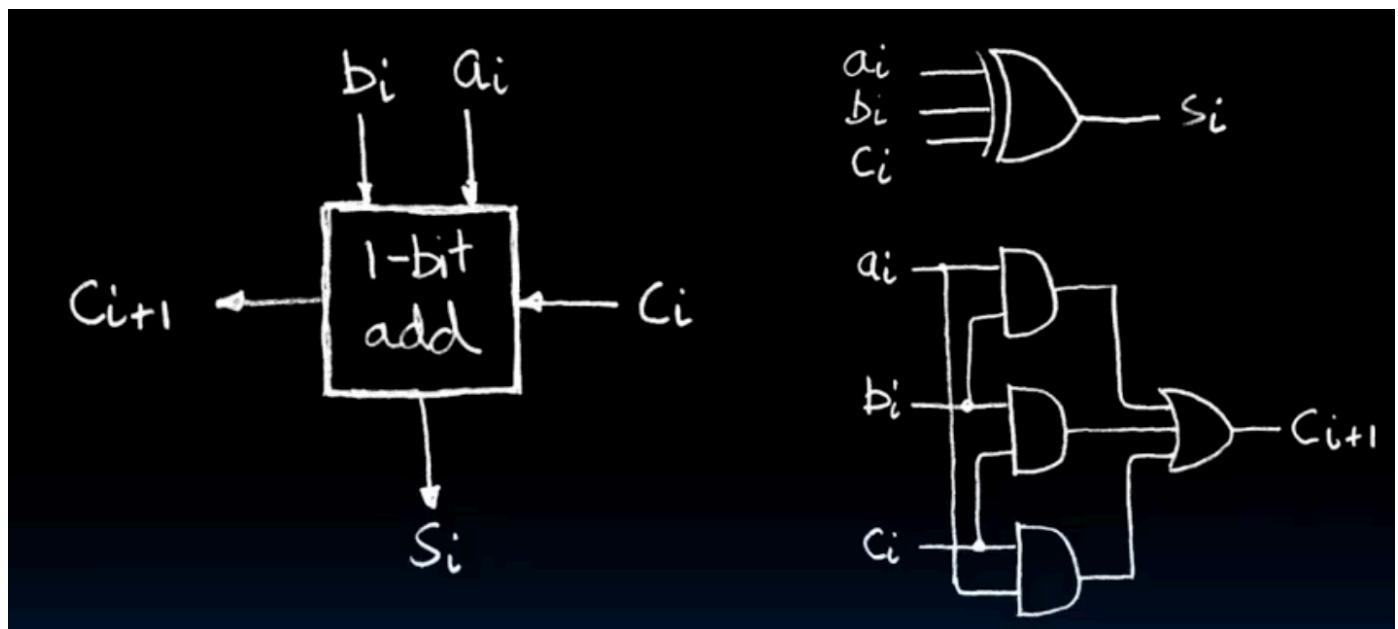
$$+ \begin{array}{cc|cc|c} a_3 & a_2 & a_1 & a_0 \\ b_3 & b_2 & b_1 & b_0 \\ \hline s_3 & s_2 & s_1 & s_0 \end{array}$$

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

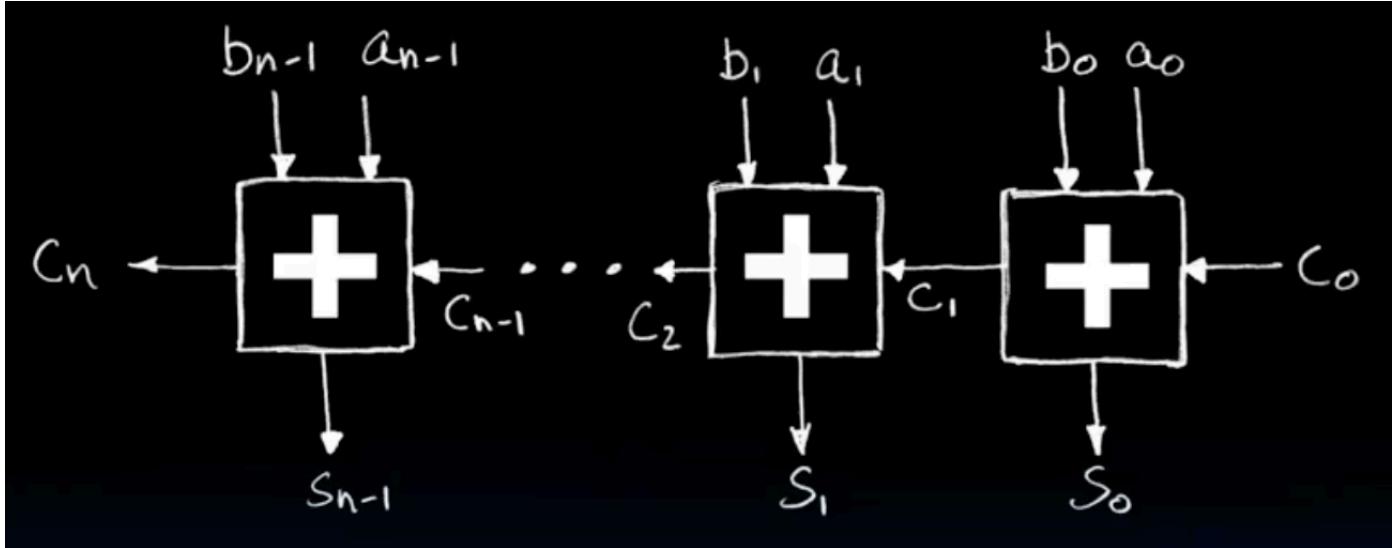
$$s_i = \text{XOR}(a_i, b_i, c_i)$$

$$c_{i+1} = \text{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

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

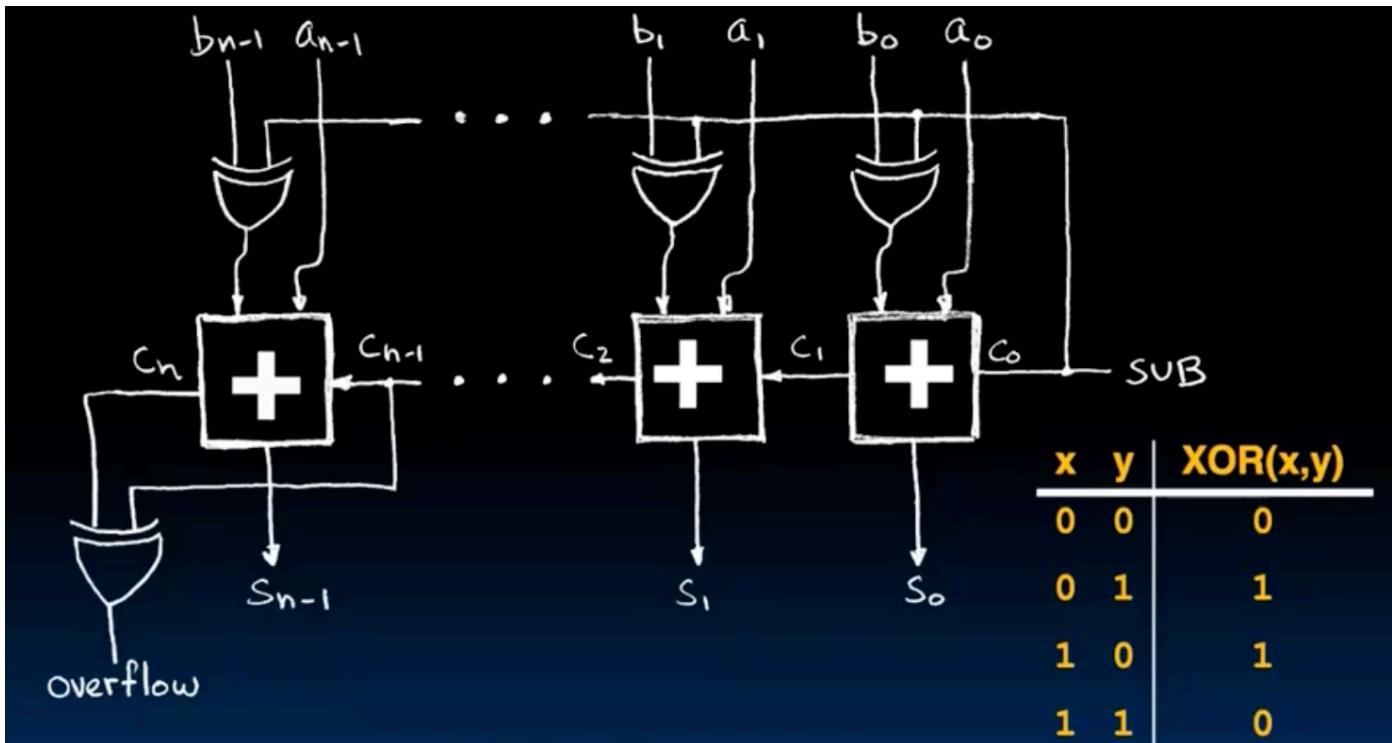
我们看最高位的adder

- 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

$$\text{overflow} = c_n \text{ XOR } c_{n-1}$$

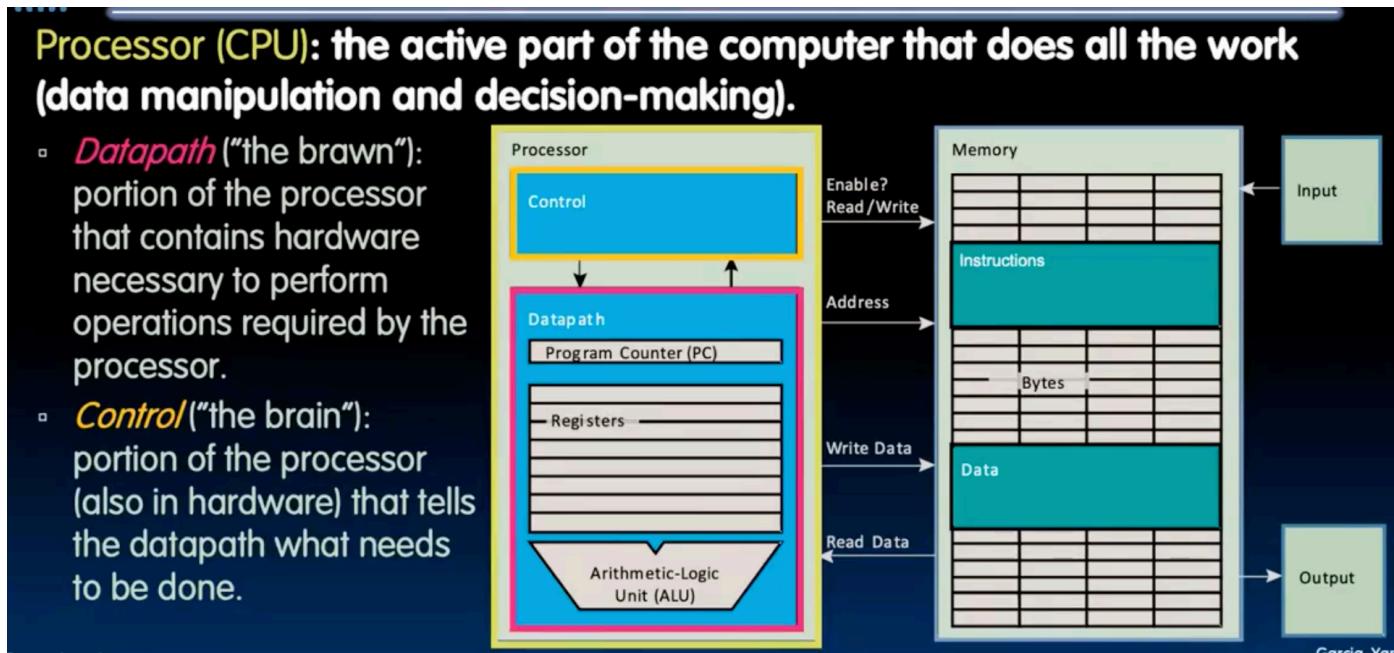
## Subtractor

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



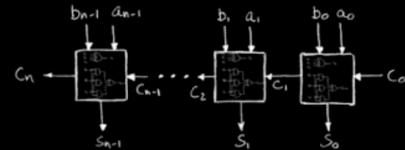
### CPU Elements and Stages

- Today, we will think of all combinational logic subcircuits as block diagrams:

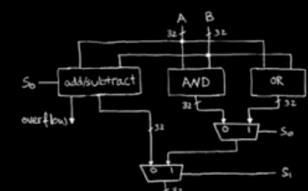
Last time



1-bit-wide 2-to-1MUX

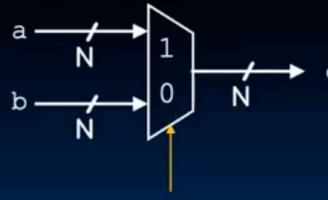


N-Bit Adder

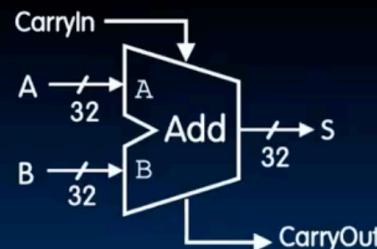


Simple ALU (add,sub, and,or)

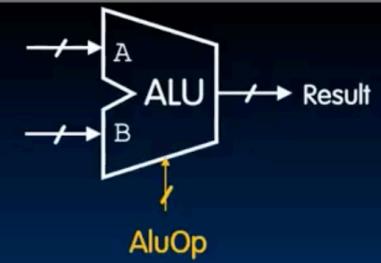
This time



32-bit-wide 2-to-1 MUX



32-Bit Adder



ALU (ALUOp selects from multiple operations) Garcia (cc) CC BY

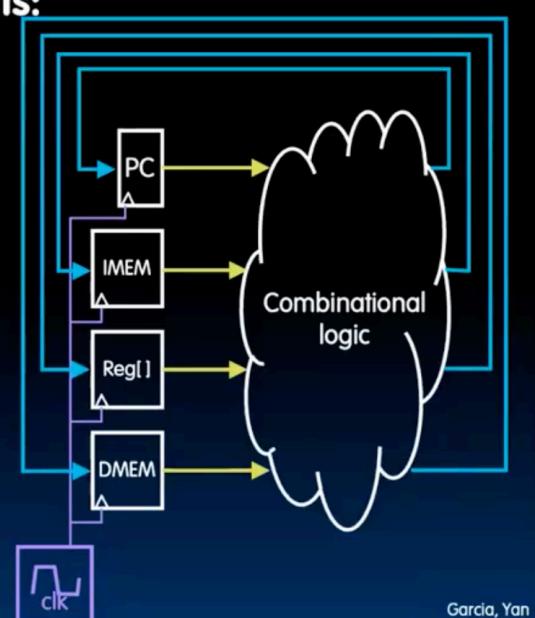
Berkeley

The CPU is composed of two types of subcircuits:

*Combination logic locks and state elements*

**The CPU is composed of two types of subcircuits: *combinational logic blocks* and *state elements*.**

- On every tick of the clock, the computer executes one instruction:
  - Current outputs of the *state elements* drive the inputs to *combinational logic*...
  - ...whose outputs settle at the *inputs to the state elements* before the next rising clock edge.
- At the rising clock edge:
  - All the *state elements* are updated with the *combinational logic* outputs...
  - and execution moves to the next clock cycle.



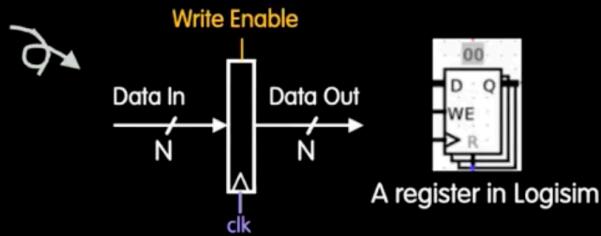
Berkeley

Garcia, Yan

状态元素

PC

## Program Counter



## Register File Reg



## Memory MEM



- Input:
  - N-bit data input bus
  - Write Enable "Control" bit (1: asserted/high, 0: deasserted/0)
- Output:
  - N-bit data output bus
- Behavior:
  - If Write Enable is 1 on *rising clock edge*, set Data Out=Data In.
  - At all other times, Data Out will not change; it will output its current value.

The Program Counter is a 32-bit Register.

Register File Reg 寄存器堆

## Program Counter



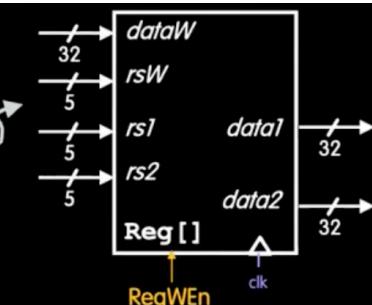
## Register File Reg



## Memory MEM



RegFile behaves like a combinational block for read operations!

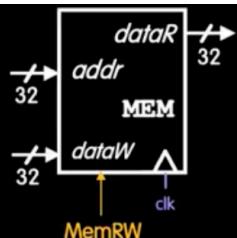


## Register File (RegFile) has 32 registers.

- Registers are accessed via their 5-bit register numbers:
  - R[rs1]: rs1 selects register to put on data1 bus out.
  - R[rs2]: rs2 selects register to put on data2 bus out.
  - R[rd]: rsW selects register to be written via dataW when RegWEn=1.
- Clock behavior: Write operation occurs on *rising clock edge*.
  - Clock input only a factor on write!
  - All read operations behave like a combinational block:
    - If rs1, rs2 valid, then data1, data2 valid after *access time*.

Memory MEM

## Program Counter



## Register File Reg



## Memory MEM

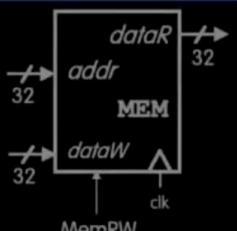


Berkeley

**IMEM** 代表 instruction memory 只读

**DMEM** 代表data memory 支持读写

## Program Counter



## Memory is "magic." For this class:

- 32-bit byte-addressed memory space; and
- Memory access with 32-bit words.

- Memory words are accessed as follows:

- Read: Address *addr* selects word to put on *dataR* bus.
- Write: Set *MemRW*=1.

Address *addr* selects word to be written with *dataW* bus.

- Like RegFile, clock input is only a factor on write.

- If *MemRW*=1, write occurs on rising clock edge.
- If *MemRW*=0 and *addr* valid, then *dataR* valid after access time.

If *MemRW*=0, **MEM** behaves like a combinational block.

## Register File Reg



## Memory MEM



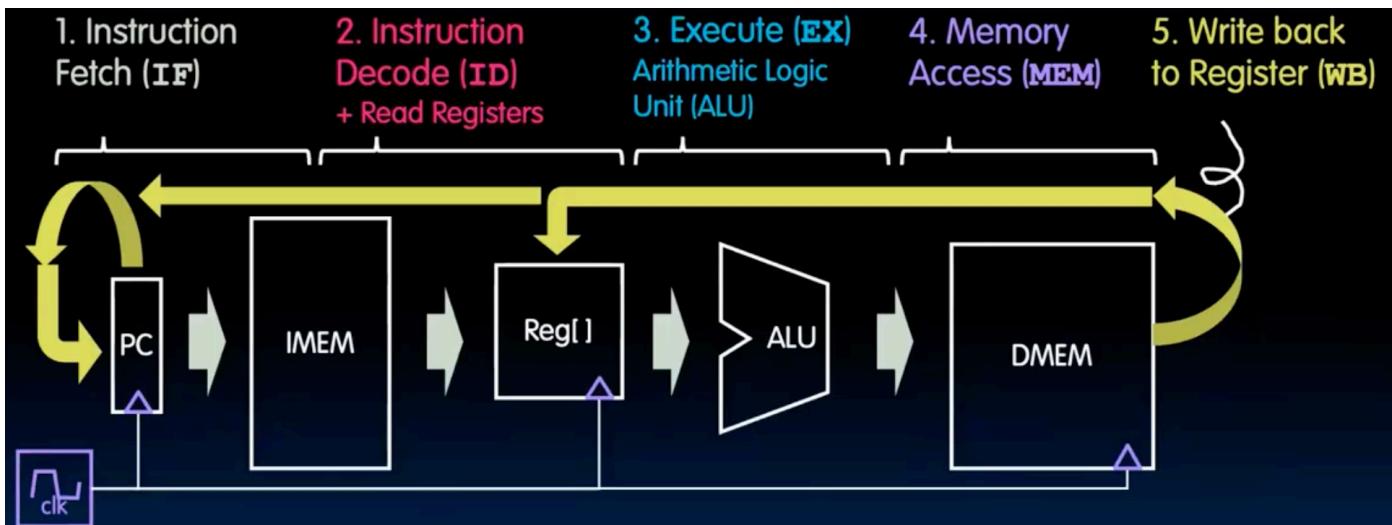
- Current abstraction: Memory holds both instructions and data in one contiguous 32-bit memory space.
- In our processor, we'll use two "separate" memories:
  - IMEM**: A *read-only* memory for fetching instructions.
  - DMEM**: A memory for loading (read) and storing (write) data words.
  - Under the hood, these are placeholders for caches. (more later)
- Because IMEM is read-only, it always behaves like a combinational block:
  - If *addr* valid, then *inst* valid after access time.

D 1.1

Go

## 程序执行的5个基本阶段

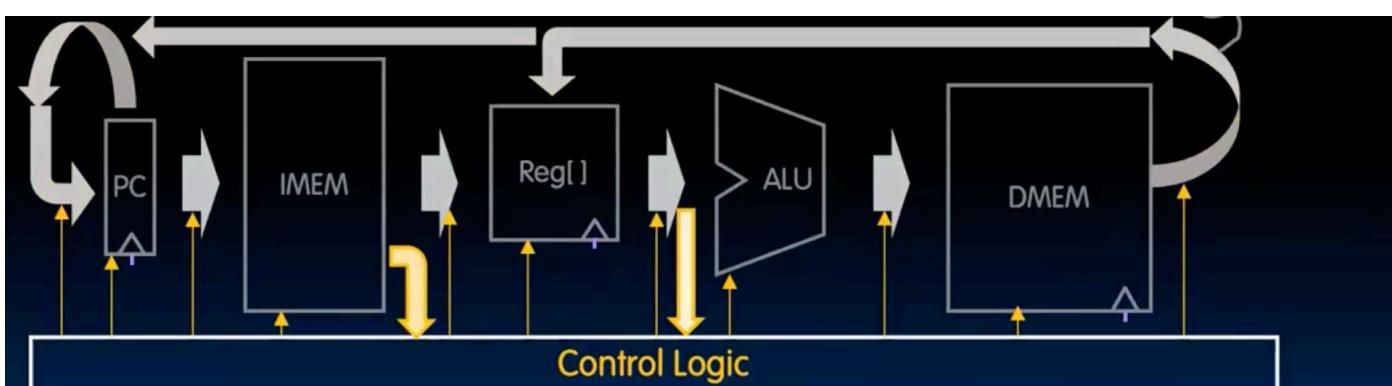
单周期处理器 -- 在一个时钟周期里完成指令



- We will implement a single-cycle processor:

- All stages of one RV32I instruction execute within the same clock cycle.

Garcia, Y



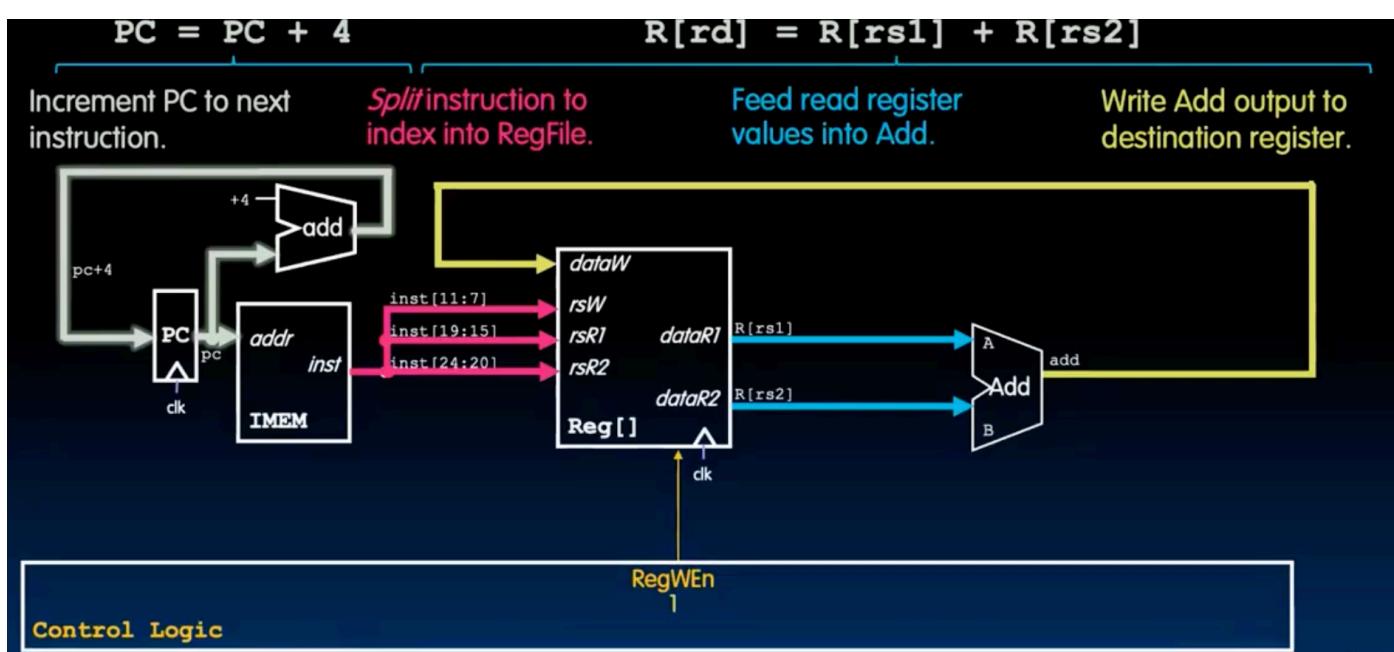
The control logic selects "needed" datapath lines based on the instruction.

- MUX selector, ALU op selector, write enable, etc.

Garcia, Y

### Datapath for add

这里我们先省略 DMEM 部分

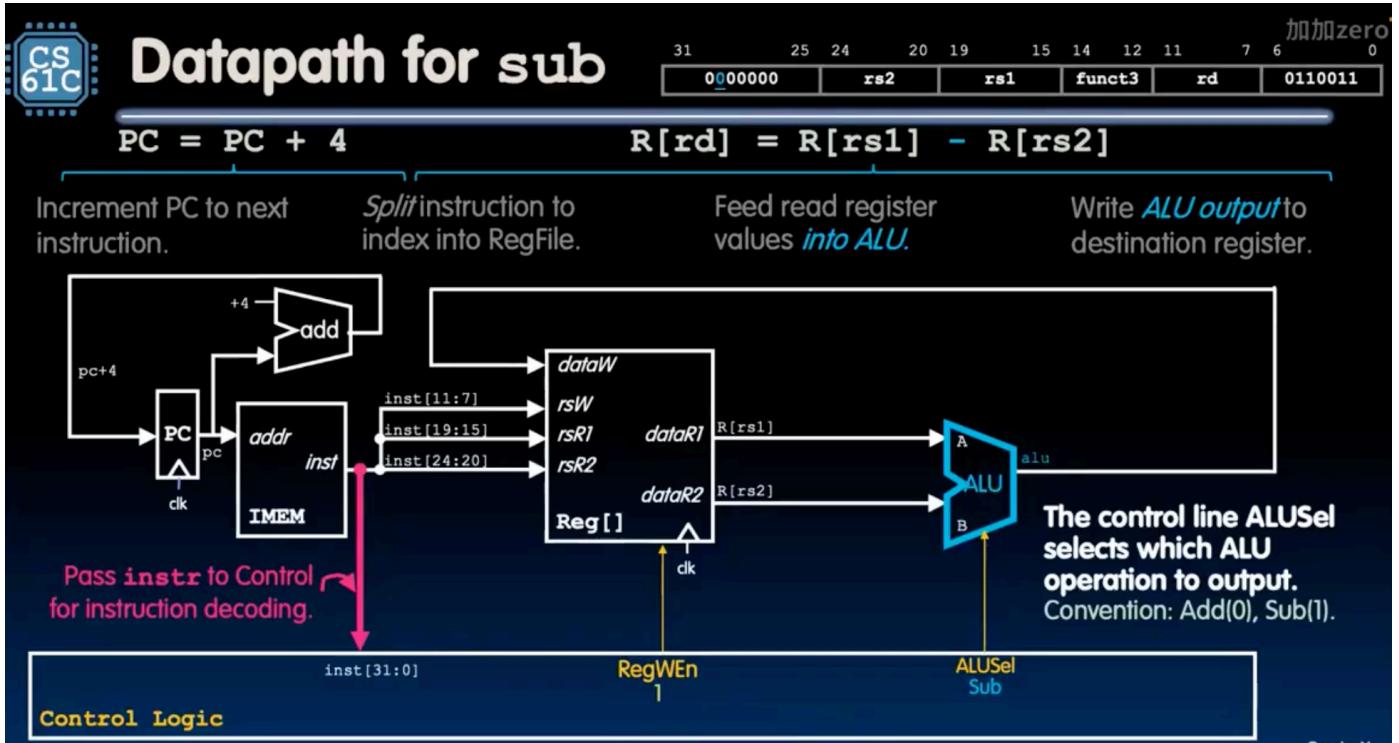


## Datapath for sub

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

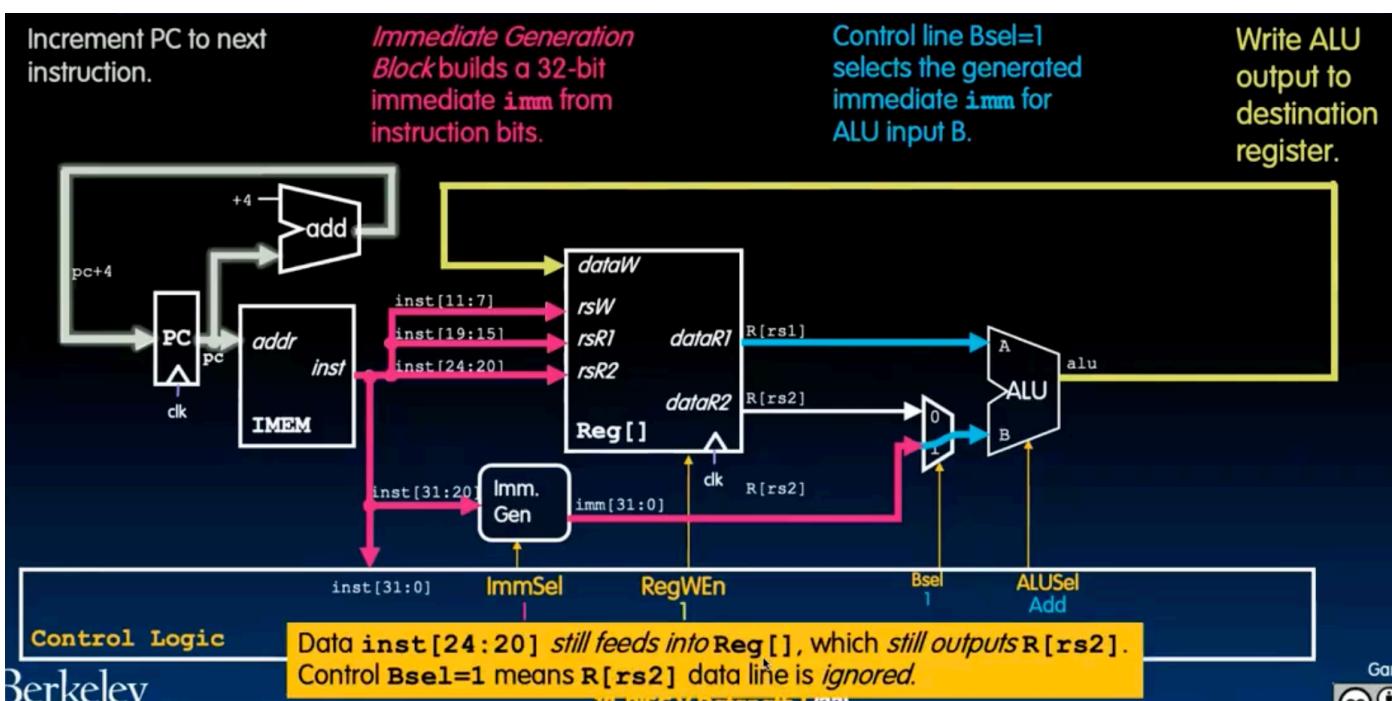
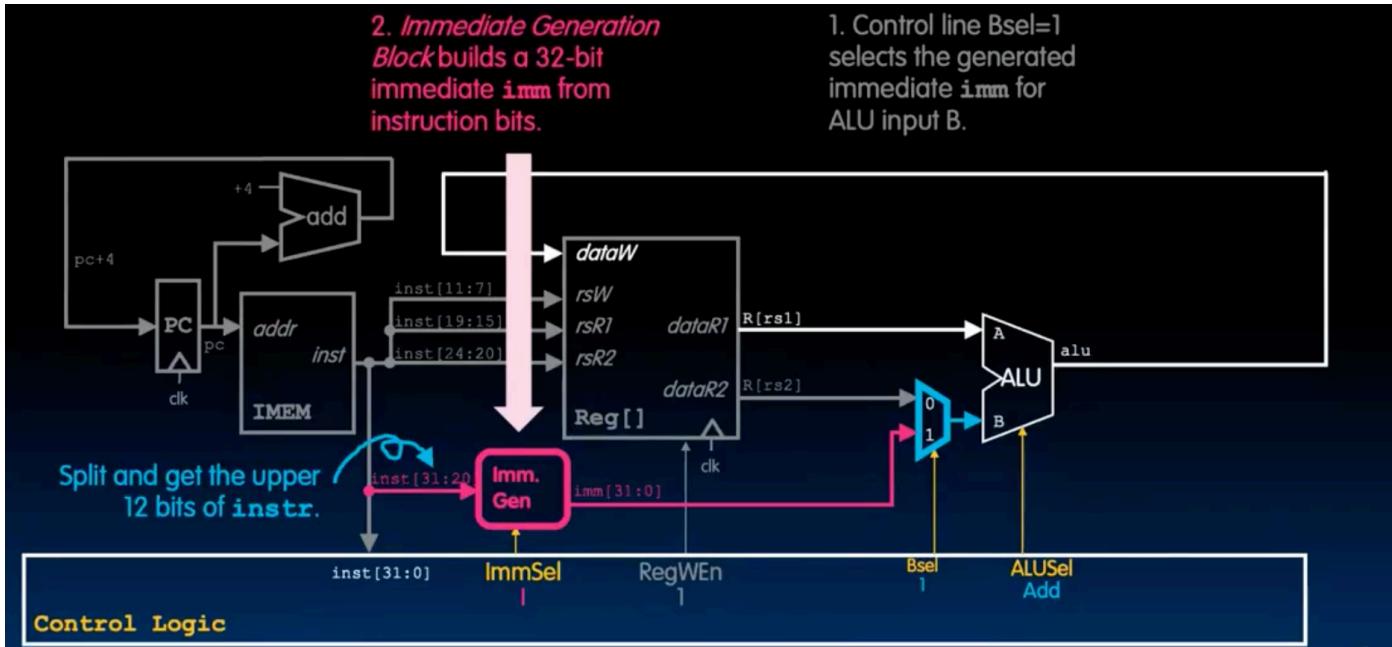
USel  
d, sub, xor, and, or,  
sltu, sll, sra, srl

The Control Logic decodes funct3, funct7 instruction fields and selects appropriate ALU function by setting the control line ALUSel.

Check out for the

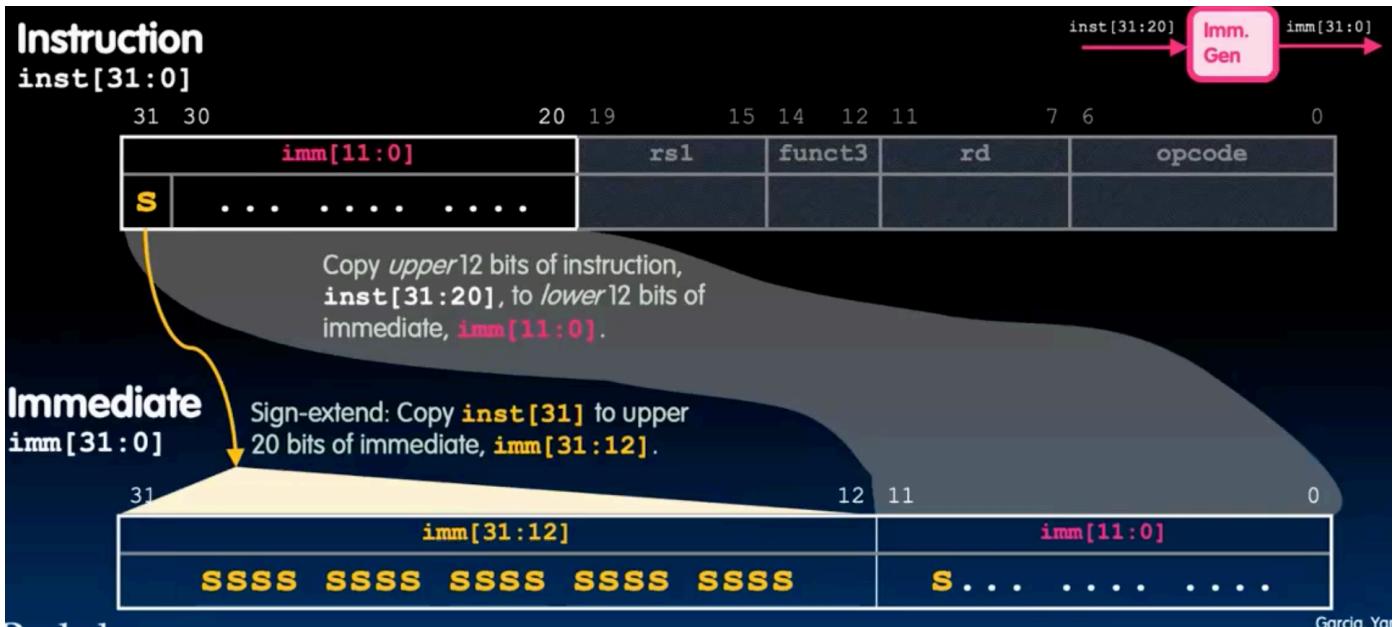
## addi datapath

addi : ALU的B端输入选择



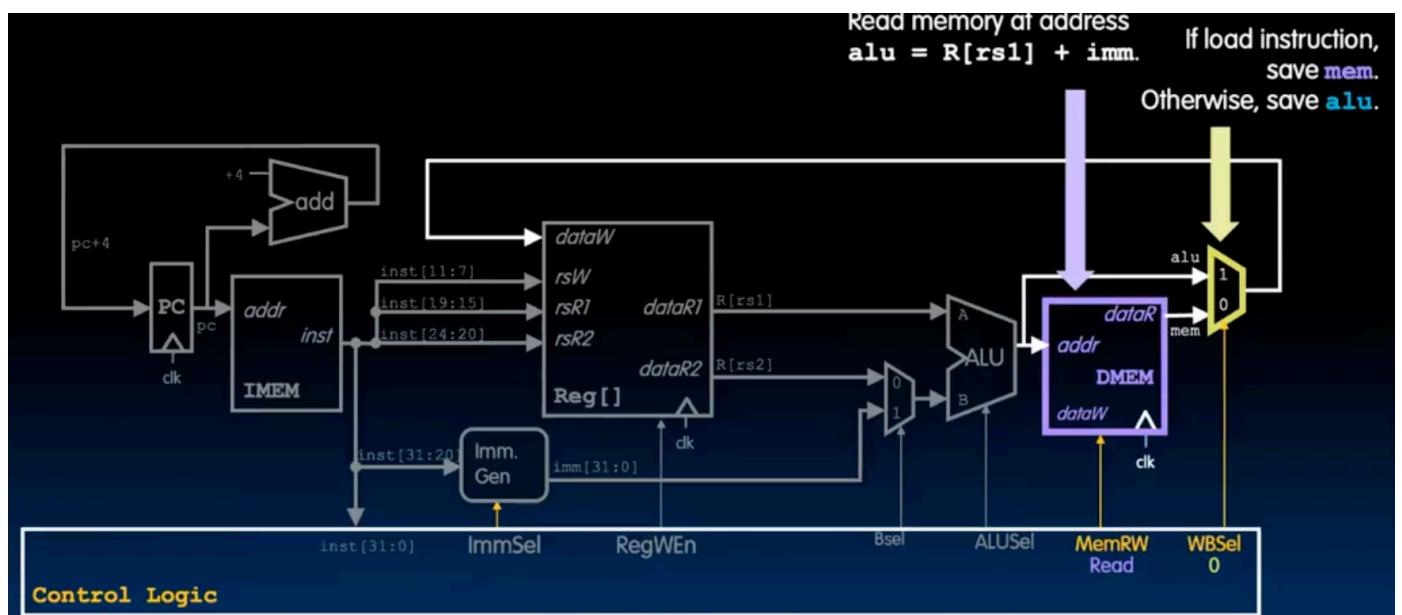
Berkeley

imme生成模块



## Implementing Loads

WBSel : write back selection



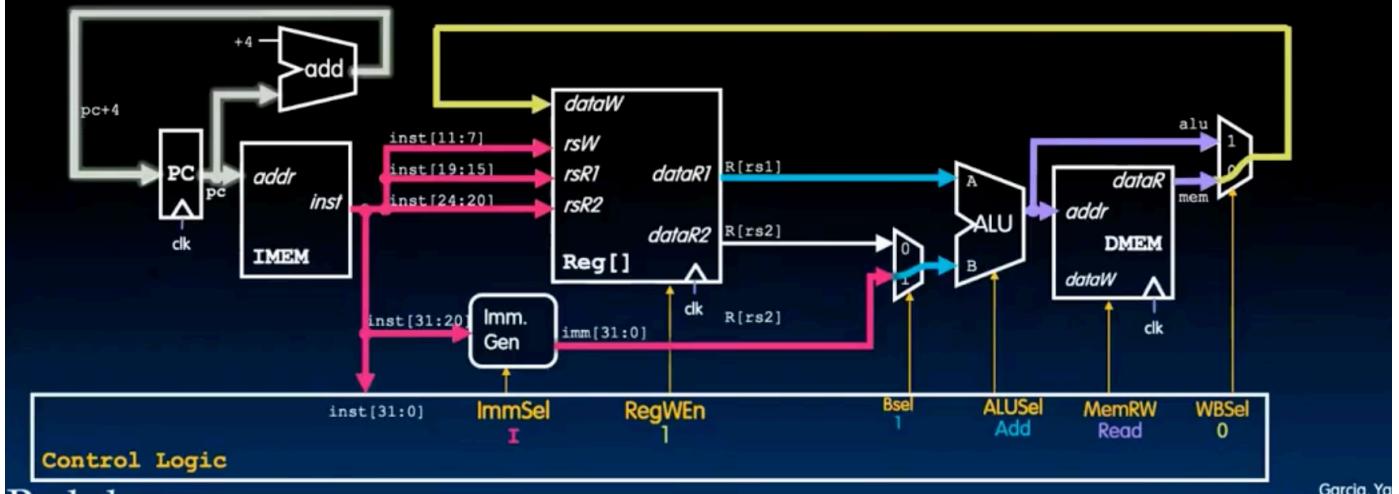
Increment PC to next instruction.

*Immediate Generation Block* builds a 32-bit immediate **imm**.

ALU computes address **alu** =  $R[rs1] + imm$ .

Read memory at address **alu**.

Write loaded memory value **mem** to register.



- 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"

- 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))

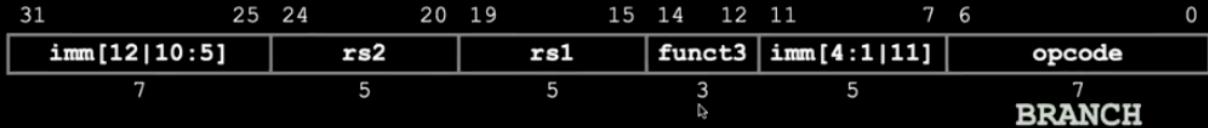
- PC** = **PC** + 4

No RegFile write!

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

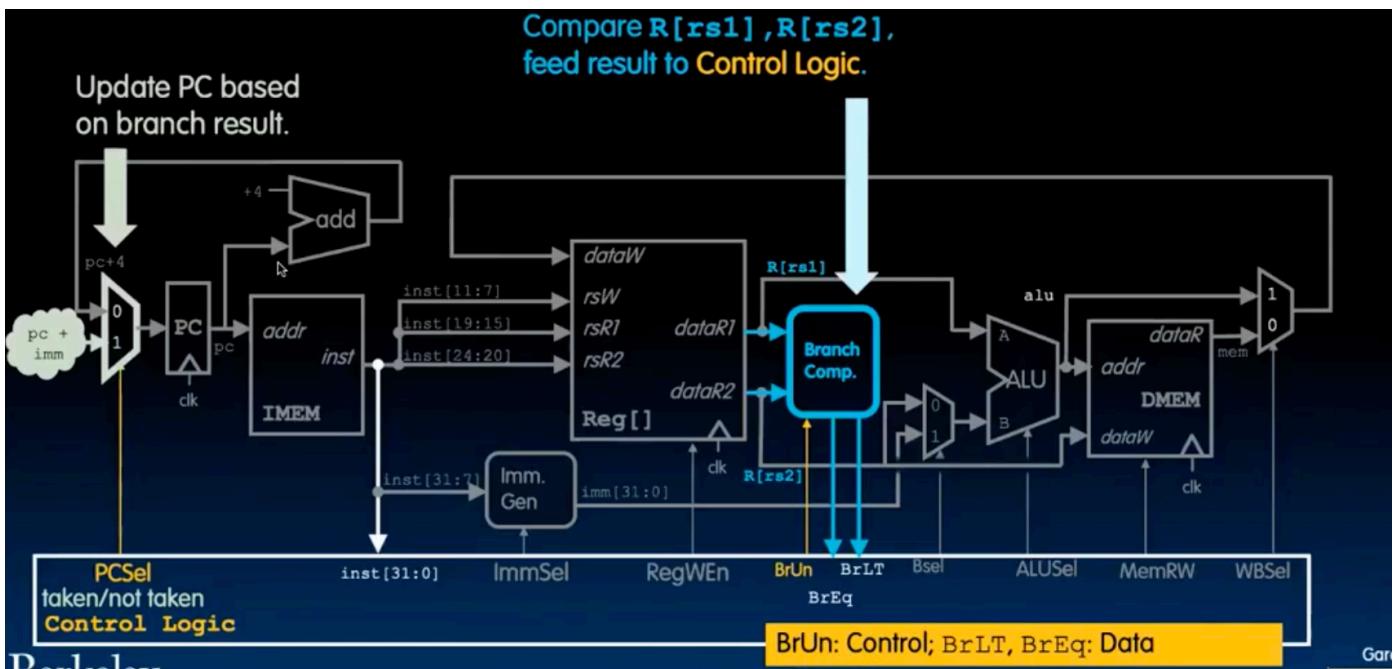
- B-Format (textbook: SB-Type) close to S-Format:**  
**opname    rs1,rs2,Label**



- New Immediate Format!**

- PC state element now conditionally changes:**

- RegFile     $R[rs1], R[rs2]$  (read only, for branch comparison)
- PC             $PC = PC + imm$  (if branch taken)  
 $PC = PC + 4$       (otherwise, not taken)



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 Unsigned), *BrLT*, *BrEQ*, 判断指令类型返回比较结果进而控制 *PCSel* 是否启用

## ▪ The Branch Comparator is a combinational logic block.

- Input:
  - Two data busses **A** and **B** (datapath **R[rs1]** and **R[rs2]**, respectively)
  - **BrUn** ("Branch Unsigned") control bit

### ▫ Output:

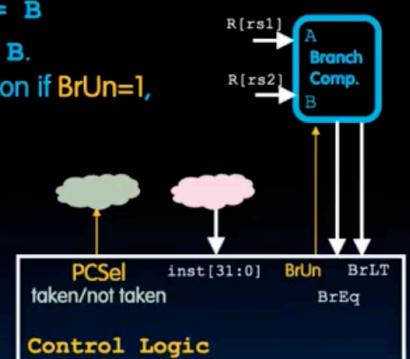
- **BrEq** flag: 1 if **A == B**
- **BrLT** flag: 1 if **A < B**.  
Unsigned comparison if **BrUn=1**, signed otherwise.

## ▪ Control Logic:

- Set **BrUn** based on current instruction, **inst[31:0]**.
- Set **PCSel** based on branch flags **BrLT**, **BrEq**.

## ▪ Examples:

- **blt**:
  - If **BrLT=1** and **BrEq=0**, then **PCSel=taken**.
- **bge**:  $(A \geq B) = \overline{A < B}$ 
  - If **BrLT=0**, then **PCSel=taken**.



Garcia-Yon

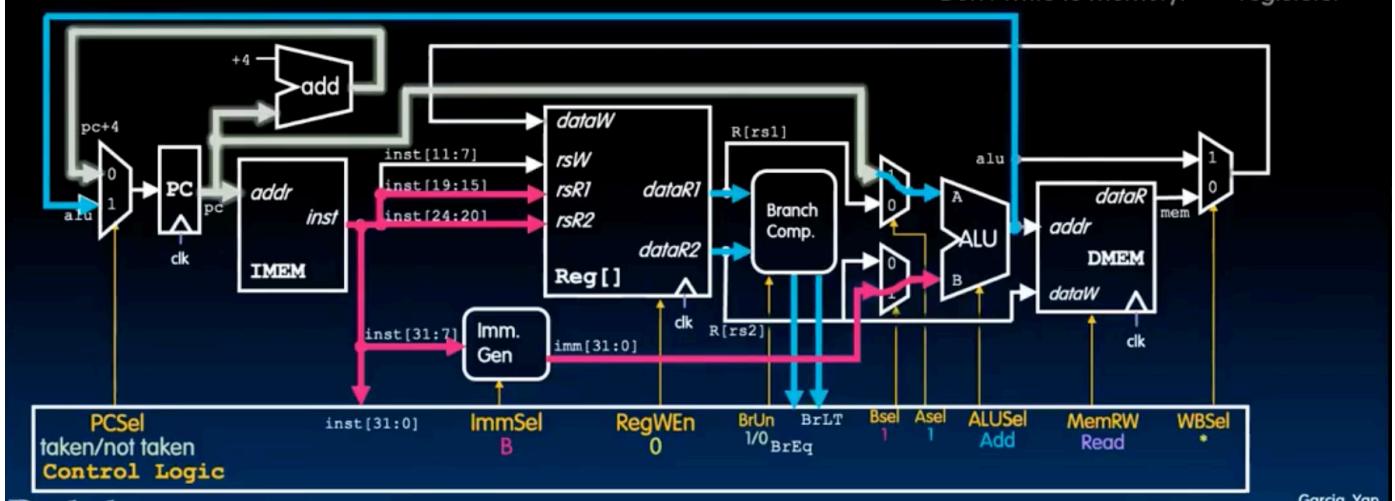
If **PCSel=taken**, update **PC** to **ALU output**. Else, update to next instruction **PC + 4**.

Build **imm** from B-type instruction.

- Compute branch; feed to Control.
- Compute **PC + imm**.

Don't write to memory.

Don't write to registers.



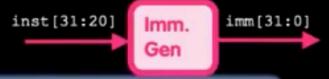
Garcia-Yon

## 设计Imme生成模块 -- 第二部分

for I-Format and S-Format

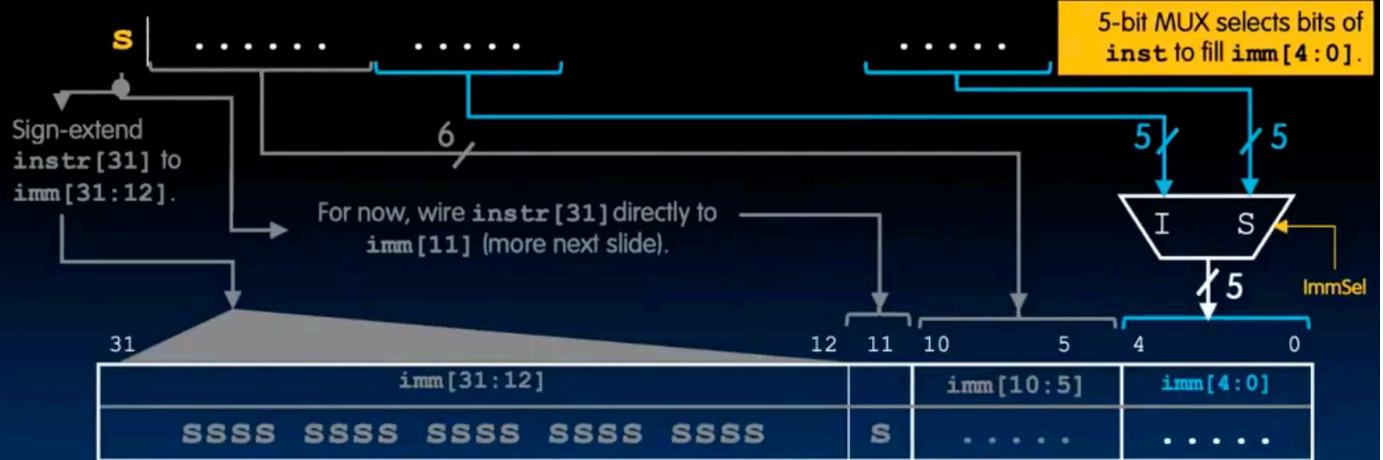


# I-Type and S-Type immediates



Instruction  $\text{inst}[31:0]$

	31	30	25	24	20	19	15	14	12	11	7	6	0
I-Type	imm[11 10:5]		imm[4:0]		rs1	funct3	rd	I-OPCODE					
S-Type	imm[11 10:5]		rs2		rs1	funct3	imm[4:0]	S-OPCODE					



Berkeley Immediate  $\text{imm}[31:0]$

19-RISC V Datapath II (23)

Garcia, Yan  
CC BY NC SA

## B-Format

Instruction  $\text{inst}[31:0]$

	31	30	25	24	20	19	15	14	12	11	7	6	0
I-Type	imm[11:5]		imm[4:0]		rs1	funct3	rd	I-OPCODE					
S-Type	imm[11 10:5]		rs2		rs1	funct3	imm[4:0]	S-OPCODE					
B-Type	imm[12 10:5]						imm[4:1 11]	B-OPCODE					

$\text{inst}[31]$  is always the sign bit.

↳

MUX for  $\text{imm}[11]$ :

- S:  $\text{inst}[31]$
- B: 0 (implicit 0; half-words  $\rightarrow$  bytes)

MUX for  $\text{imm}[0]$ :

- S:  $\text{inst}[7]$
- B: 0 (implicit 0; half-words  $\rightarrow$  bytes)

	31	30	25	24	20	19	15	14	12	11	7	6	0
	imm[31:12]						imm[10:5]	imm[4:0]					
	ssss ssss ssss ssss ssss						.	....					

Berkeley Immediate  $\text{imm}[31:0]$

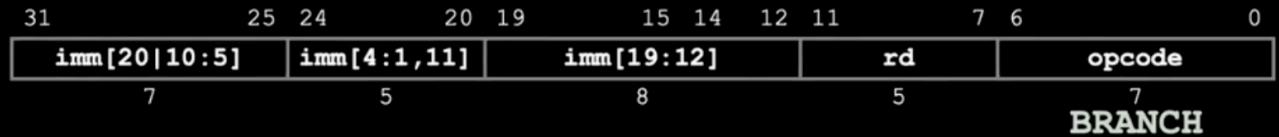
我们根据指令的类型选择我们想要的数据片段。

Garcia, Yan  
CC BY NC SA

## Adding Jumps jal jalr

- J-Format:

**jal rd, Label**

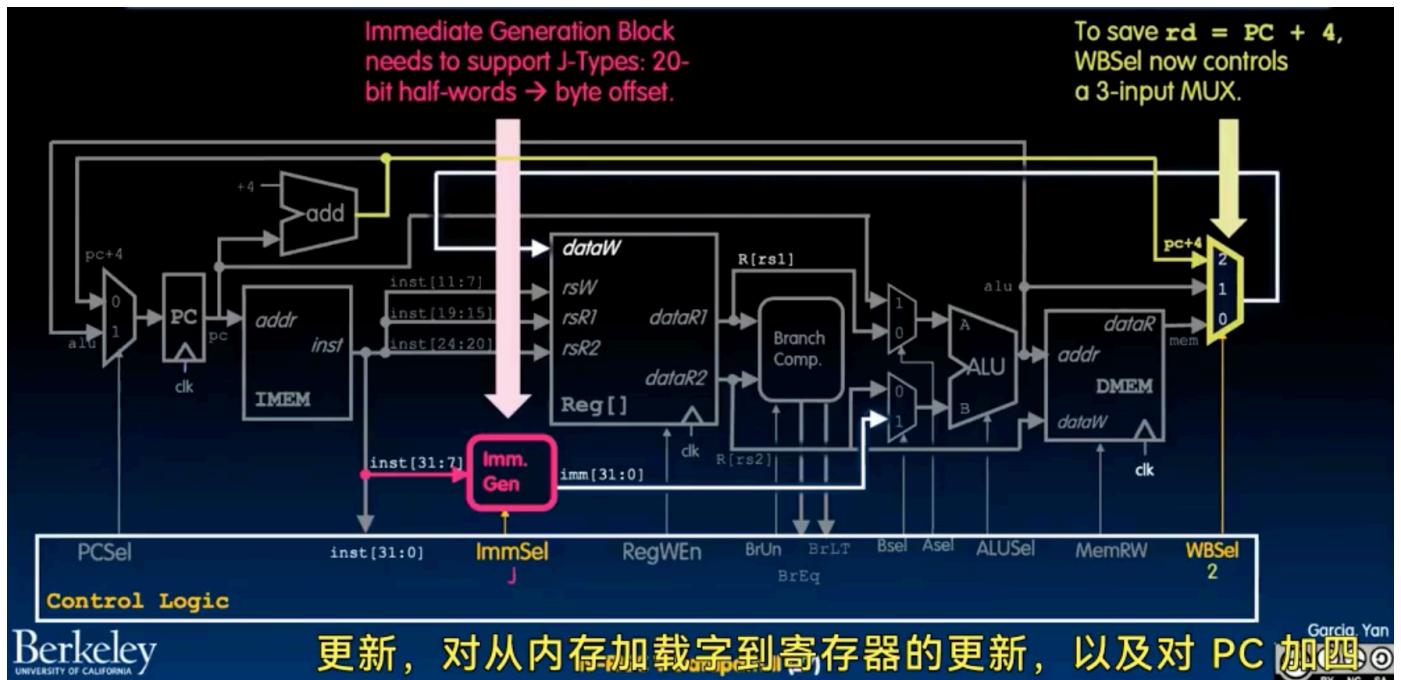


- New immediate format! (see Project 3)

- 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



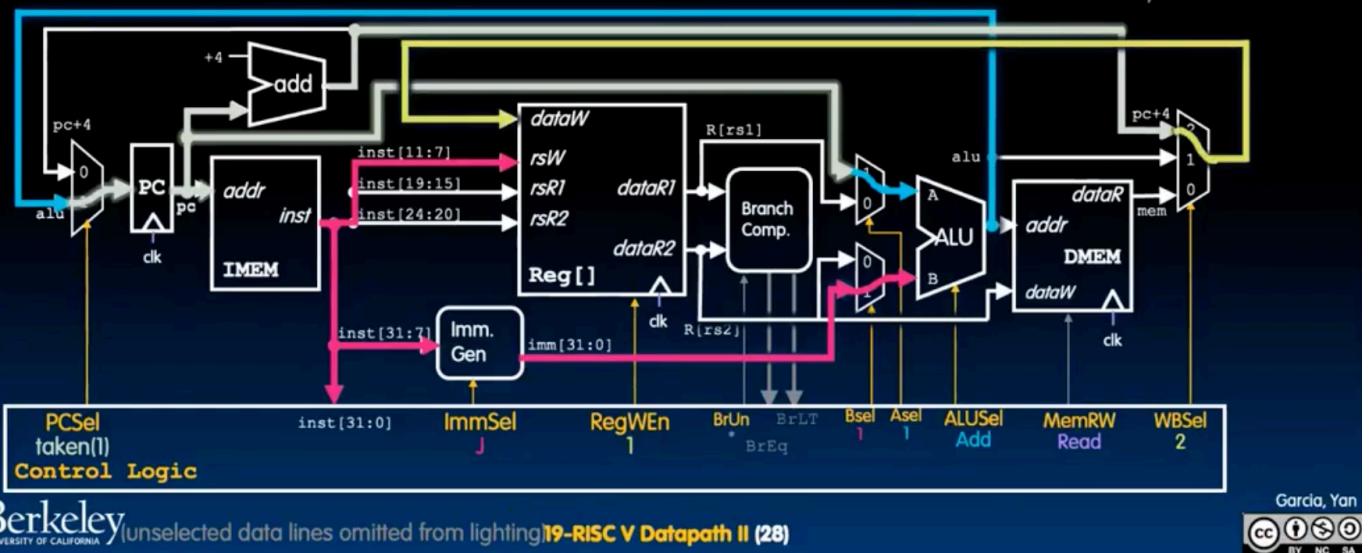
- Feed PC into blocks.
- Write ALU output to PC.

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

Compute **PC + imm**.

Write **PC + 4** to destination register.

Don't write to memory.



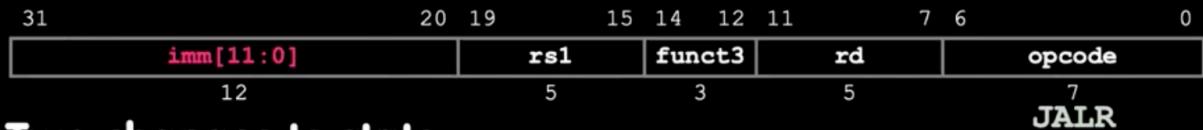
Berkeley  
UNIVERSITY OF CALIFORNIA

(unselected data lines omitted from lighting) 19-RISC V Datapath II (28)

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

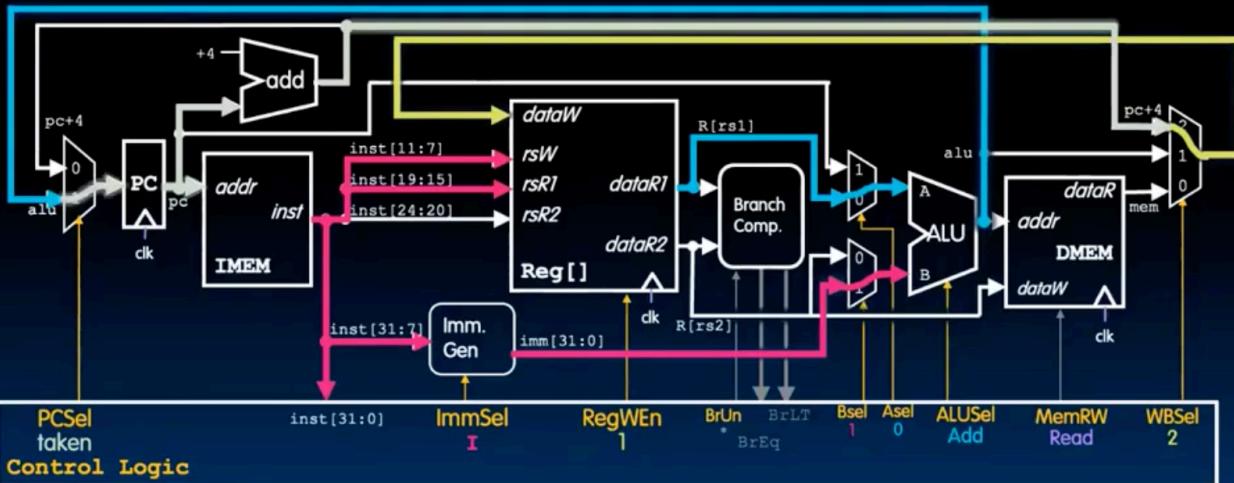
Berkeley  
UNIVERSITY OF CALIFORNIA

19-RISC V Datapath II (29)

Garcia, Yan

- Feed PC into blocks.
- Write ALU output to PC.
- Generate 12-bit **imm** (I-Format).
- Compute **rs1 + imm**.
- Write **PC + 4** to destination register.

Don't write to memory.

Berkeley  
UNIVERSITY OF CALIFORNIA

19-RISC V Datapath II (30)



## U-Format

## U-Format Instruction Layout

Review

- “Upper Immediate” instructions:

**opname      rd, imm<sub>ed</sub>**



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)

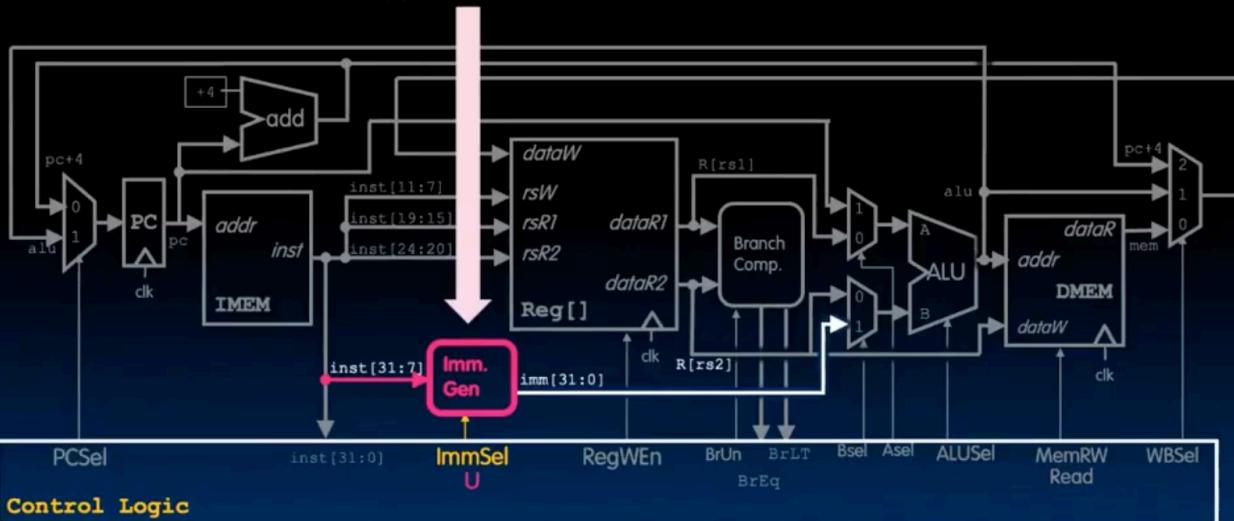


Garcia, Yan



# U-Format Block update: Immediates

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



Garcia, Yen



## Lighting the LUI Datapath

$$\begin{aligned} \text{PC} &= \text{PC} + 4 \\ \text{rd} &= \text{imm} \end{aligned}$$

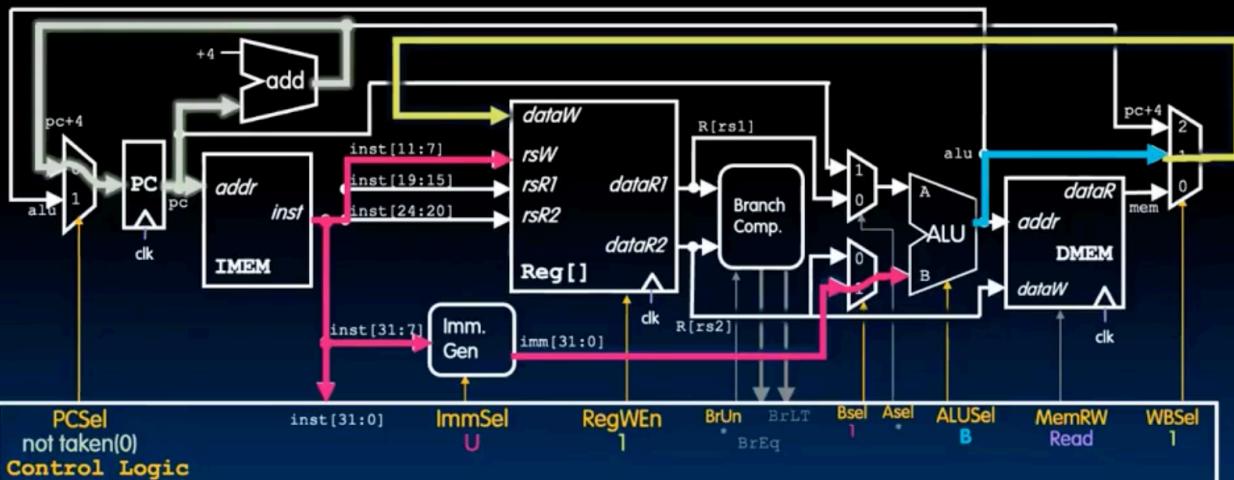
Increment PC to next instruction.

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

**Grab only imm!**  
(ALUSel=B)

Write result to destination register.

Don't write to memory.



Garcia, Yen

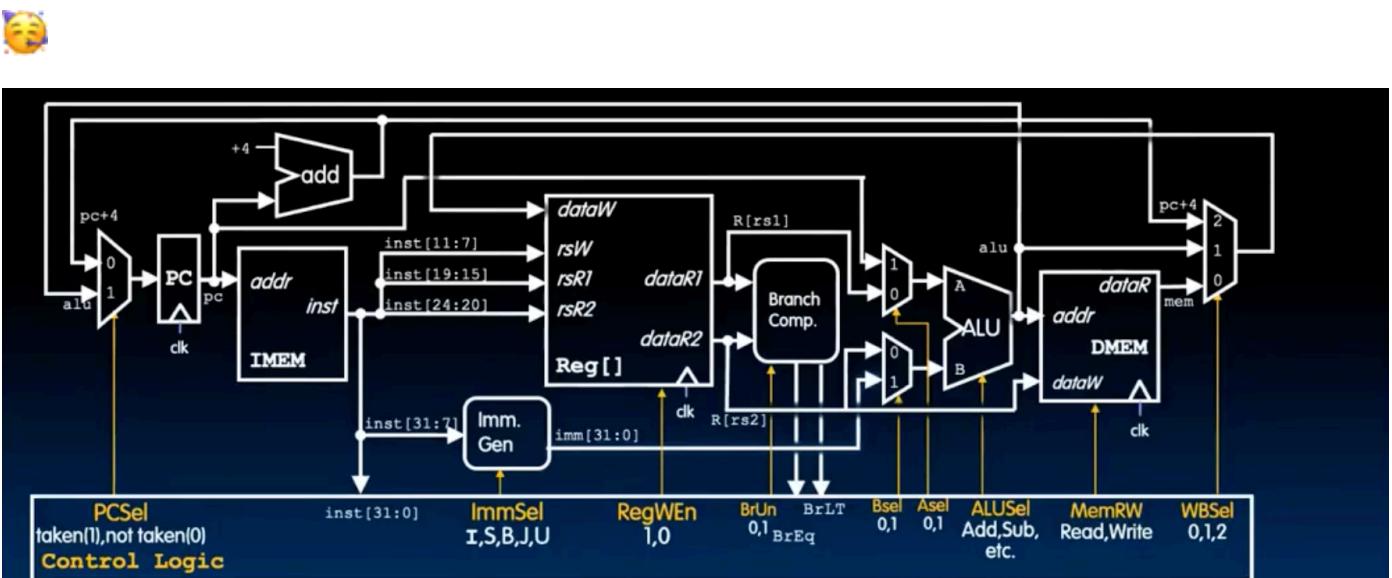
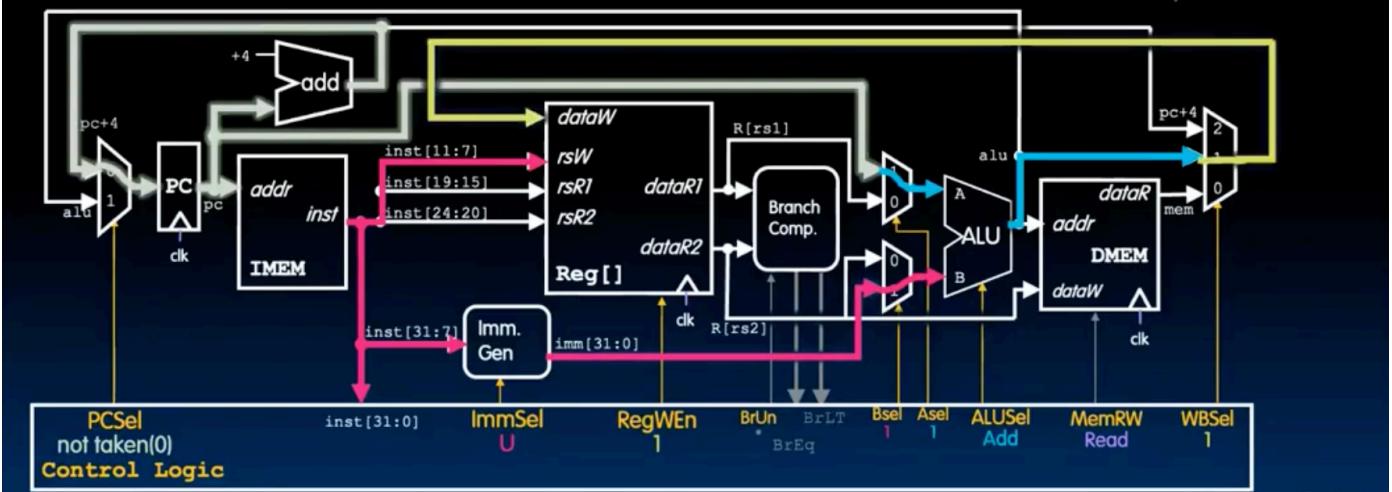
Increment PC to next instruction.

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

Add  $PC + imm!$

Write result to destination register.

Don't write to memory.



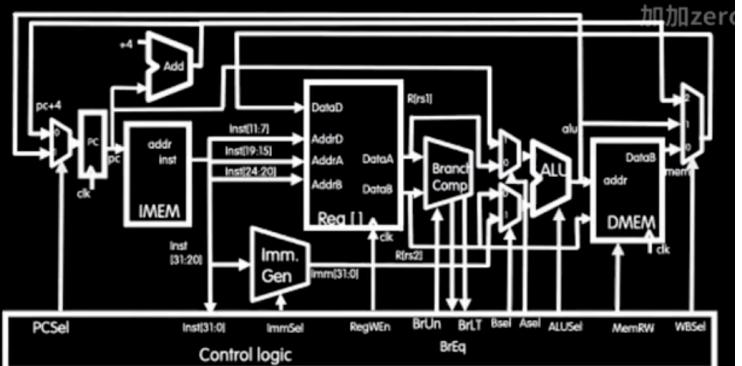
## Lecture 20 : Control

### Instruction Timing

add 举例

# add Execution

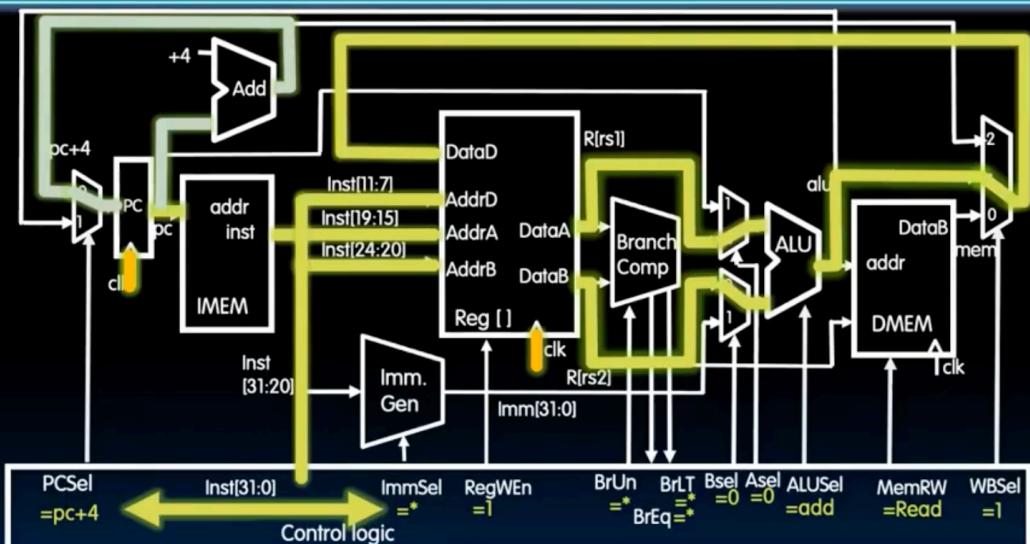
e.g.,  
**add x1, x2, x3**



	Control logic		
Clock			
PC	X	1000	X
PC+4	X	1004	X
inst[31:0]	X	add x1,x2,x3	X
Control logic	X	add control	X
Reg[rs1]	X	Req[2]	X
Reg[rs2]	X	Req[3]	X
alu		X Req[2]+Req[3]	X Req[7]+Req[9]
wb		X Req[2]+Req[3]	X Req[7]+Req[9]
Reg[l]	???	X Req[2]+Req[3]	

Garcia, Yau

## Example: add timing



$$\begin{aligned} \text{Critical path} &= t_{\text{clk-q}} + \max \{ t_{\text{Add}} + t_{\text{mux}}, t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} \} + t_{\text{setup}} \\ &= t_{\text{clk-q}} + t_{\text{IMEM}} + t_{\text{Reg}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}} \end{aligned}$$

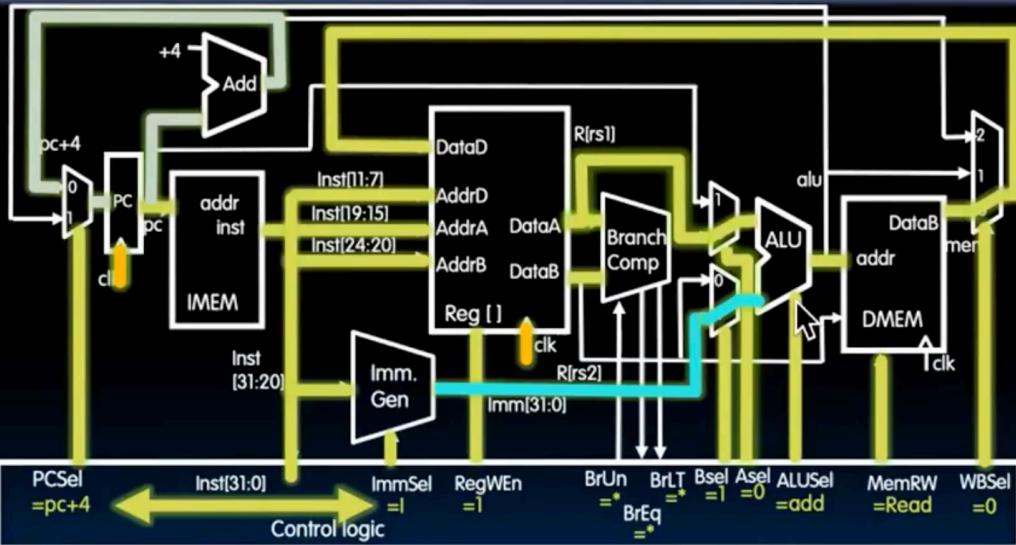
Garcia, Yau

lw 举例



## Example: lw reg, offset(regbaseptr)

加加zero



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

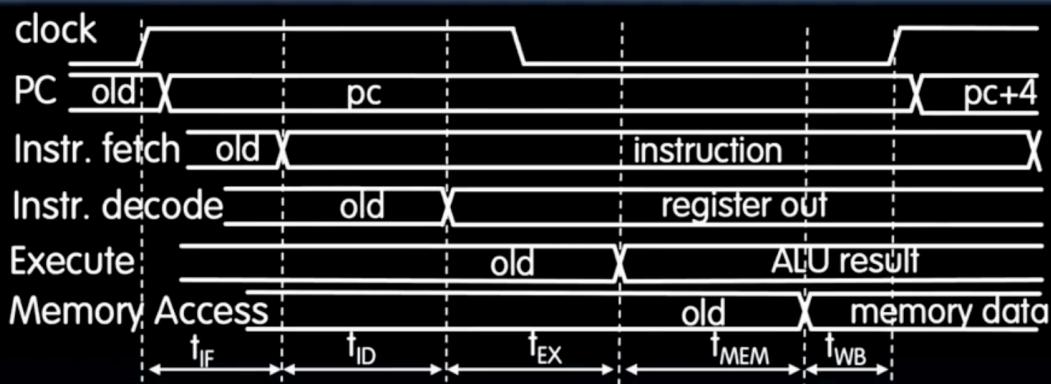
P. 1.1

Garcia, Yau



## Instruction Timing

加加zero



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

P. 1.1

Garcia, Yau



# Instruction Timing

加加zero

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
<b>add</b>	X	X	X		X	600ps
<b>beq</b>	X	X	X			500ps
<b>jal</b>	X	X	X			500ps
<b>lw</b>	X	X	X	X	X	800ps
<b>sw</b>	X	X	X	X		700ps

- Maximum clock frequency
  - $f_{max} = 1/800\text{ps} = 1.25 \text{ GHz}$
- Most blocks idle most of the time
  - E.g.  $f_{max,ALU} = 1/200\text{ps} = 5 \text{ GHz!}$

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



### Control Logic Truth Table

加加zero

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
<b>add</b>	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
<b>sub</b>	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
<b>(R-R Op)</b>	*	*	+4	*	*	Reg	Reg	(Op)	Read	1	ALU
<b>addi</b>	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
<b>lw</b>	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
<b>sw</b>	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
<b>beq</b>	0	*	+4	B	*	PC	Imm	Add	Read	0	*
<b>beq</b>	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
<b>bne</b>	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
<b>bne</b>	1	*	+4	B	*	PC	Imm	Add	Read	0	*
<b>blt</b>	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
<b>bltu</b>	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
<b>jalr</b>	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
<b>jal</b>	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
<b>auipc</b>	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

## 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：



## Control Logic to Decode add

加加zero

inst[30]	inst[14:12]	inst[6:2]				
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
000000	rs2	rs1	000	rd	0110011	ADD
010000	rs2	rs1	000	rd	0110011	SUB
000000	rs2	rs1	001	rd	0110011	SLL
000000	rs2	rs1	010	rd	0110011	SLT
000000	rs2	rs1	011	rd	0110011	SLTU
000000	rs2	rs1	100	rd	0110011	XOR
000000	rs2	rs1	101	rd	0110011	SRL
010000	rs2	rs1	101	rd	0110011	SRA
000000	rs2	rs1	110	rd	0110011	OR
000000	rs2	rs1	111	rd	0110011	AND

$$\text{add} = \overline{i[30]} \cdot \overline{i[14]} \cdot \overline{i[13]} \cdot \overline{i[12]} \cdot \text{R-type}$$

$$\text{R-type} = \overline{i[6]} \cdot i[5] \cdot i[4] \cdot i[3] \cdot \overline{i[2]} \cdot \text{RV32I}$$

$$\text{RV32I} = i[1] \cdot i[0]$$