所有：① Abstraction ② Moore's Law ③ Make common case fast
④ Memory Hierarchy ⑤ Parallelism ⑥ Pipeline

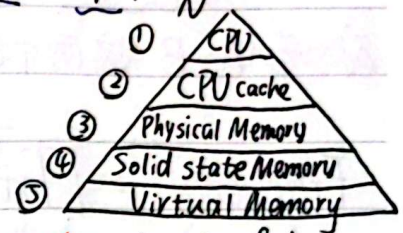
CS110 Review：⑦ Dependability with redundancy ⑧ Performance Evaluation

Great ideas in CA:

Moore's Law: The number of transistors on microchips doubles every two years. It's a prediction, not Law!

Amdahl's Law: 加速比 $= \frac{1}{(1-P)+P/N}$ ← 处理器数量；P：可并行化部分占总任务比例；1-P：不可并行部分比例

优化后：$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$




Principle of Locality / Memory Hierarchy

右图从上至下：存储速度变慢，但容量逐渐增大

每一层更详细地：① Register ② On-chip cache：L1、L2 & L3

③ Main memory；RAM；内存条 ④ 固态内存；SSD；闪存；NVM

Parallelism / Pipeline：

→ Preamble 预处理 → Fork() 分叉 { worker thread 工作线程 ..... } → Join() 合并 → post-processing 后处理

Dependability via Redundancy:

Increasing transistor density reduces the cost of redundancy & Redundancy so that a failing piece doesn't make the system fails

Info_Representation:

| 63 | 62 | ... | 2 | 1 | 0 |
|---|---|---|---|---|---|
| 0 | 0 | | 1 | 0 | 1 |

(LSB) least significant bit：最右边一位（0号） RISC-V double：64 bits long

(MSB) most：最左边一位（63号）

若是 unsigned number，range 则为 $[0, 2^{64}-1]$

如表示负数？法一：sign & magnitude，abandoned

法二：One's complement：正数不变，负数所有位翻转，LSB为sign bit

但 range：$0 \sim 2^{n-1}-1$；$-0 \sim -(2^{n-1}-1)$；0两种表达！不友好！

KOKUYO

No.

Date . .

法三：Two's Complement：正数不变，负数全位反转并加1

在2补码下的表示数 $(a_n a_{n-1} \cdots a_1 a_0)_2$，它表示：

$$-a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Arithmetically Friendly! But remind to check overflow!

+5 0101

-3 A:1 B:1 0 1

1 0010 = 2, ✓

-7 1001

-2 A:1 B:1 1 0

1 0111 = 7, ?

☆：若 A.B 这两个进位不同，则 overflow!

Fractional：Floating Point!

A number in scientific notation that has no leading 0 is called: normalized.

Eg：$3.14 \times 10^{-9}$ ✓，$0.314 \times 10^{-8}$ or $31.4 \times 10^{-10}$ ✗

用二进制角度：则能不能也：$1.xxx\cdots x_{two} \times 2^{yyy}$ 呢？

其中：$1.xxx\cdots$ ？= k，一个"1"代表 $2^k$，$k = -1, -2, \cdots$

2？：0 -1 -2 -3 ---

Floating-Point Representation：In RISC-V：

bit：1 8(含sign bit)* 23

meaning：sign exponents fraction (mentissa)

number = $(-1)^S \times$ Fraction $\times 2^E$

☆ Floating-Point Representation：In IEEE 754

使 leading 1 bit of normalized binary numbers implicit (implied)

因此，实际数是 24 bits long in single precision (1+23)

正因有隐性1，"0"如何表示？则令 exponent 为0，硬件识别到则不加1

因此 $00\cdots00_{two}$ 代表0，其余的数表示法为：

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

详细地：若 fraction 部分 左至右为 $s_1, s_2 \cdots$，则值为：

$$(-1)^S \times (1 + s_1 \times 2^{-1} + s_2 \times 2^{-2} + \cdots) \times 2^E$$

即：E=0 & Fraction=0 ⇒ 0

Campus

No.

Date

它还可表示 $\pm\infty$，当 fraction 为 0，exponent 全 1 时，代表 $\infty$ (sign bit 决定是 $+\infty$ 还是 $-\infty$)

进一步还可表示 NaN：exponent 全 1，fraction 不为 0

至于 Exponent，若采用 2's complement form，则比大小还要多看一步 E 的 sign bit (如负E & 负E 的比大小)，但计算机算正数二进制大小很友好。故 IEEE 754 采用 a bias of 127 for single precision. Eg. -1 指数，则 E 应为 $-1+127 = 126_{ten} = 01111110_{two}$

则 "E" 8-bit range: $[0, 255] \Rightarrow$ E range $[-127, 128]$

但之前提过："E" 8位 全为 0/1 (0/255) 有含义，故实则：

Exponent $\in [-126, 127]$，则 IEEE normalized *FP32 范围：

$$\pm 1.\overbrace{000\cdots0}^{23个}_{two} \times 2^{-126} \rightarrow \pm 1.\overbrace{111\cdots1}^{23个}_{two} \times 2^{127}$$

Example: $-0.75_{ten}$ : $-1$ & $0.5+0.25 \Rightarrow 0.11_{two} \times 2^0$

$\Rightarrow 1.1 \times 2^{-1}$ → fraction. "E" $= -1+127 = 126_{ten} = 01111110_{two}$

则：

| 31 | 30 29 28 27 26 25 24 23 | 22 ------ 0 |
|---|---|---|
| 1 | 0 1 1 1 1 1 1 0 | 1 0 0 --- 0 |

FP32: 1 | 1000000 1 | 0100 -- —

$S=-1$, $E = 129-127 = 2$, 则：$(-1) \times 1.01_{two} \times 2^2$

$= (-1) \times (1+0.25) \times 4 = -0.5$

之前说 E=0 时且 Fraction=0 代表 0，那么 Fraction≠0 呢？

代表 subnormal / denormalized 数！$\Rightarrow$ zero E but nonzero fraction

当 E≠0 时，表达最小数为：

$$1.00\cdots0_{two} \times 2^{-126}$$

KOKUYO

No.

Date

Subnormal

但通过"Exponent为0去触发implicit为0", 则可表达的最小数为: $0.000\cdots1_{two}\times2^{-?}$, 这一类数最大为: $0.111\cdots1_{two}\times2^{-?}$

-? 应是多少? 是-127么? (因为 0 - bias = -127, i.e., bias = 127)

但! $0.11\cdots1_{two}\times2^{-?}$ 应能与 $1.00\cdots0_{two}\times2^{-126}$ 接上!

则: ? 固定为 126 !! ☆

Subnormal 范围:

$\pm\ 0.00\cdots1_{two}\times2^{-126} \sim 0.11\cdots1_{two}\times2^{-126}$

(即 $1.0\times2^{-149}$)

总结: normal: $\pm1.00\cdots0_{two}\times2^{-126}$ ② ~ ④ $\sim\pm1.11\cdots1_{two}\times2^{127}$

subnormal $\pm0.00\cdots1_{two}\times2^{-126}$ ③ ~ ① $\sim\pm0.11\cdots1_{two}\times2^{-126}$

① : 非常接近! ② $1.0_{two}\times2^{-126}$ ③ $1.0_{two}\times10^{-149}$ ④ $<\approx 1.0_{two}\times2^{128}$

Special Cases:

| Exponent | Mantissa (Fraction) | Value. |
|---|---|---|
| 全1 | 0 | ±∞ |
| 全1 | 非0 | NaN |
| 全0 | 0 | Zero (0) |
| 全0 | 非0 | Subnormal. |

Campus

No.

Date

# CS110 Review: C

## C Compilation:

(Abstract Syntax Tree)

main.c → pre-process: macro*, #include → check errors, 生成AST → AST2LLVMIR, then生成汇编(.s file) → 汇编 => 机器码 → machine code object file (.o file) (main.o) → Linker* → main.out

lib.o ↓

*: main.o → Linker → main.out

*: macro convention: 在哪儿都加括号!!

Eg. # define MAG0(x,y) sqrt(x*x+y*y)

# define MAG1(x,y) (sqrt((x)*(x)+(y)*(y)))

则 MAG0(i+1, j+1) output: sqrt(i+1*i+1 + j+1*j+1).

MAG1(i+1, j+1) output: sqrt((i+1)*(i+1)+(j+1)*(j+1))

因此, 尽可能少用宏, 几乎只有很小的 speed up

## Size of type:

| | char | short | short int | int | long int | unsigned int |
|---|---|---|---|---|---|---|
| 32 bit蓝 | 1 | 2 | 2 | 4 | 4 | 4 |
| 64 bit红 | 1 | 2 | 2 | 4 | 8 | 4 |

| void* | size_t | float | double |
|---|---|---|---|
| 8 | 8 | 4 | 8 |

Diff: long int = 电脑多少位 = void* = size_t

一个地址在几位PC便几位

size_t 设计目的便是对齐32/64 bit

## "True or False" Boolean in C:

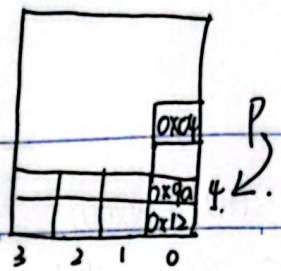
FALSE ⇔ 0 (int). NULL (pointer); TRUE ⇔ ¬FALSE

## Address vs. Value

Memory 其实是就是 byte array!

KOKUYO

△: malloc ▲: dangling pointer: function
calloc (0初始化) 内开局部 var, return其地址;
realloc . 但它在heap上, 故会回收




一个cell, i.e., 一个byte; int: 4 bytes; char: 1 byte. stack

因此存放不同数据的结构体有alignment现象!

一个地址多少byte: up to PC。CS110默认32位, 4 byte.

一个地址 ⇒ particular memory location; Pointer就是存放地址的变量.

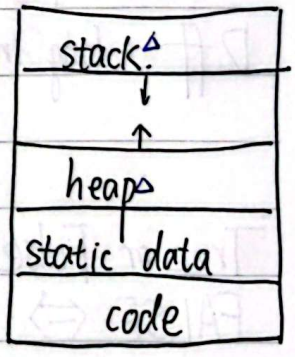
Syntax of Pointer:

&: 取address. *p: 解引用, 返回p地址对应value

注意函数传参: 是传地址! (引用传递; 值传递⇔copy init)
避免指针这个variable声明时不告诉它地址 (Undefined Behavior)

Array: *
String: 最后有'\0' (NULL) Byte ☆ (NULL Terminator).
Eg: char s[] = "abc" ⇒ {a, b, c, '\0'}.
char s[3] = "abc" ⇒ {a, b, c}

☆: Array variable is a "pointer" to the first element
ar[2] ~ *(ar+2). *: type* p: p+i指向地址: p+i*sizeof(type)
Key Concept: 传参时欲对传入变量就地操作, 传其指针! △
我有int array的指针 int *q, 欲指向下一个地址(含q).
则函数应传入 int ** p




C Memory Management:
stack: function中局部变量
heap: malloc() 开辟动态内存
static data: function外声明的变量, 可修改.
code: (a.k.a. text) 不可修改

*: 只能Array隐式转至pointer! 反之不行!
△: 传参只能传pointer! 传array也行, 但array转化为了pointer!

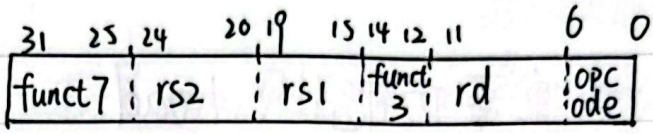
No.

Date . .

# CA Review: RISC-V

Assembly Instructions: 4个核心 type (RISU) + B/J
不同type有不同format，但 rs1, rs2, rd 在相同位置

R-type: Register-register operation
接受 rs1, rs2，输出至 rd (S: source; d: destination)
不可访问 main memory
$a = b \pm c$




$a \Leftrightarrow rd \quad b \Leftrightarrow rs1 \quad c \Leftrightarrow rs2$
还可以是位运算：AND/OR/XOR；还可比较：SLT/SLTU：~ rd, rs1, rs2
rd=1 if rs1 < rs2 else 0；且把 number 当作 signed/unsigned with slt/sltu
(SLT(U): store if less than, U: unsigned)

slt(u) 可各自检测 加减法在 signed/unsigned 数域内是否 overflow:
unsigned: 如 add $x_2, x_1, x_1$ 若 $x_3=0$, $x_2<x_1$, 则溢;
sltu $x_3, x_2, x_1$ 否则 $x_2=x_1+x_1>x_1$

signed:
add $t_0, t_1, t_2$ $t_3=1$ if $t_2<0$ $t_0=t_1+t_2$.
slti $t_3, t_2, 0$ $t_4=1$ if $t_0<t_1$
slt $t_4, t_0, t_1$ 若 $t_2<0$, $t_0>t_1$ 矛盾? 溢出!
bne $t_3, t_4$, overflow 若 $t_2>0$, $t_0<t_1$. 矛盾? 溢出!

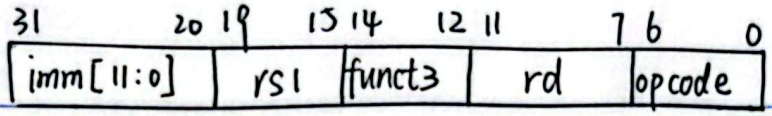
还可是 shift 操作: shift left/right (arithmetic) sll/srl/sra rd, rs1, rs2
rs1 左/右移，多少位呢? rs2 中低5位所代表的十进制数 ($2^3=32$)
而 arithmetic: sign-extended ※指的是右移后前面空出的位
用1/0填充，rs1原为负(符号位为1)则填1; 反之填0
而: srl 总用0填充; sll, 空出的低5位必用0填

※: RISC-V中，所有Imm均是sign-extended

KOKUYO

No.

Date . .




I-type：有两个operands：一个从reg中拿(rs1)，一个是立即数(sign-extended)

常用命令：addi x1, x0, 10 / -10 (有符号)

slli/srli/srai 按imm低5位十进制移；如何从机器码分辨是srai？

用imm高7位中1位 (imm[11:0]). (or funct7 field).

andi/ori/xori/slti/sltui 都是与signed-imm操作：

slti: rs1中数→有符号 ↘ sltui: rs1中数→unsigned.

还有一类重要功能：Load

lw: lw rd, imm(rs1) : load word at addr. to rd.

而 addr.: rs1中数 + imm 注意 word是4byte 32位

附：Little Endian: ADDR: 3 2 1 0 (Big Endian反之)

Byte value: 3 2 1 0

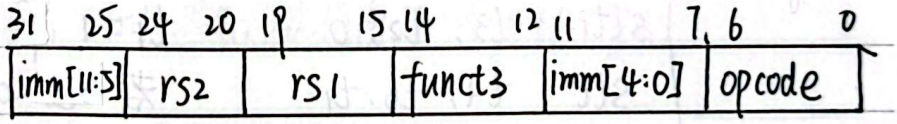
lb/lbu : load signed/unsigned byte at addr. to rd.

☆：拿一个byte来，用1/0填至32位(填高24位)；8位中sign-bit为1，则1而lbu总用0填充

类似地：lh/lhu : halfword: 2 bytes

而能load，就能store：也就是 S-type：

S-type:




sw rs2, imm(rs1) : store word (32bit) at rs2 to addr.

addr. = (number in rs1) + imm.

由于rs2中数本身32bit，故也就无"填充之说"；而sh, sb呢？

无shu, sbu !! 直接取rs2中数低16/8位，就sh/sb了。

Recap addi x11, x0, 0x4F6 ⇒ 0x85F6.

Exercise. sw x11, 0(x5) (假设已超过分配指令)

lb x12, 1(x5)

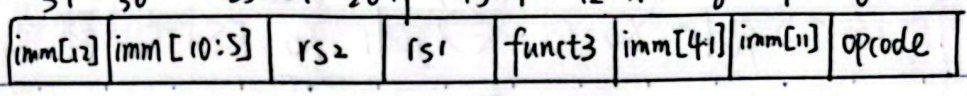
Campus x12中：0x4. ↙0x FFFFFF85.

(lb: signed ⇒ padding)

No.

Date

B-type(又称SB)




conditional branch (unconditional: J-type).

bne/beq rs1, rs2, L (imm/Label). going to L if rs1 ≠/= rs2

blt/bltu/bge/bgeu rs1, rs2, L

< ≥ → 比较时：视为unsigned (无符号).

△ Otherwise: go to next statement

当前命令地址

L若是Label，则跳至label；若是imm，则跳至 (PC+imm).

介绍了type后，介绍RISC-V中如何实现类似"call function"

正常情况，PC一条条运行指令，但跳至函数后，不像Label，要能返回跳时的下一指令，i.e., PC+4

可见应有reg保管PC信息 ⇒ ra (x1).

传参统一用 $a_0$-$a_7$ 表示与保存 ($a_0$-$a_1$ 用于返回)；

$s_1$, $s_2$-$s_{11}$ 用于作saving reg；$t_3$-$t_6$ 用作临时变量

sp: stack pointer, sp↓开辟空间，↑则恢复原状

☆ Calling convention: callee reg 通通存起来！

sp. $s_1$, $s_2$-$s_{11}$, $s_0$ (fp) (不要求了解).

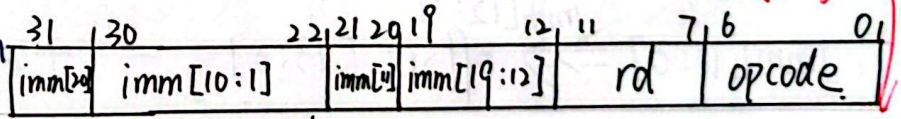
良好使用 calling convention，可以支持 nested call，因为在进入函数后，原来callee内容存至memory，那么function就能使用它们了；只需ret前lw拿回来即可

⚠⚠它是J-type！放在此介绍仅因其与function call有关

最后的J-type，就与函数调用有关：

J-type:

jalr rd rs1 label




rs1+imm 最低位清零 (2 byte对齐)

jump to label and return PC+4 to rd; if imm, jump to (rs1+imm)&~1

而：jal offset ⇔ jal x1, offset (x1=ra) | jal rd, offset, 跳至PC+offset并PC+4给rd.

j offset ⇔ jal x0, offset (相当于不存PC+4) (常用于Loop)

区别.

KOKUYO

No. Δ: 指令 machine code 中都是拿 imm 的；若传 label，则会计算 imm(offset)！

Date

jr rs ⇔ jalr x0, rs ⇔ jalr x0, rs, 0，跳至 rs，并不保存 PC+4（x0 永远为 0）

Δ: nested call 中，sp 移动且 sw callee reg 且 setup argument reg 如 add a0, s0, x0），这一堆操作称为 Prologue
而最后 lw 恢复 callee 且 sp 移回，这一堆操作称为 Epilogue

最后简单介绍 instruction → machine code 的 format:
rd, rs1, rs2 用 5 位 unsigned bits 表示 reg. 的编号 ($2^5=32$)
opcode: 7位 & funct3 (3位) & funct7 (7位) 共同决定是什么操作

但不是任何指令都有 funct3 & 7；也在 I type 中，funct3 & 7 together 足以判定指令

Encoding in: I-type: imm 12 bit, signed, ∈[-2048, 2047] ⇐ rule: 2 的补码
① imm 在指令用它操作前，要先 sign-extended !! ②但若用于位移，则直接取[4:0]，不必 sign-extend
③而 load 中，funct3 中第一位 indicates signed(0)/unsigned(1)
而后两位 indicates word(10)/h(01)/b(00)

(I type 3类任务):
① ⇒ arithmetic ② ⇒ shift ③ ⇒ load

S-type: funct3 前一位总0，后两位：word(10)/h(01)/b(00)

B-type: 如何表示 label*? 相对于该命令地址的相对地址!
use imm field as a two's complement offset to PC
可见，address 相对范围为 $\pm 2^n$。但是一个指令四 byte，怎么说也要 2-byte alignment，则 offset 最后两位无用，则：

offset[11:0] ⇒ offset[13:2]，一下子 offset 范围扩至 $\pm 2^{13}$

但若至少 2-byte，则是 1 位无用，范围：$\pm 2^{12}$ (from PC)
相当于：$\pm 2^{10}$ 32 bit instructions [signed]

Campus * 机器码中，imm(即 offset)的 [12:1] 被 encode

R-type: rd. rs1. rs2. 5 bit unsigned ⇒ No. reg

funct3 & 7 共同定 operation type

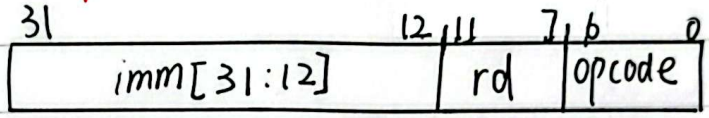
J-type: jal rd, label , store PC+4 to rd, jump to label

label = PC + offset (imm). , 故 label 转为 offset , ± $2^{18}$ 32-bit instructions (因为支持的 imm: [20:1])

I-type 补 jalr: store PC+4 to rd and jump to label = rs+imm

imm ∈ [-2048, 2047], 不再相对于 PC. 而相对于 rs reg 中的地址。注意: imm [11:0] 前10位全部保留, 而非最后一位清零, 因为 rs 中地址可能不是对齐的

U-type: Lui & auipc




lui rd, imm , rd = imm <<12

先前 I-type 中 imm 范围为 [-2048, 2047], 但和 lui 结合, 给 reg 使的数的范围就能 32-bit 了: [11:0]

i.e., li x5 imm ⇒ lui x5 imm[31:12] + addi x5, x5, imm

(附: 若 li rd imm 的 imm 小于等于12位, 则 addi 即可)

△: Given imm, signed, 若 imm 第12位为1呢? 则 低12位照取, 而高20位最后一位要加1

auipc: rd, imm: rd = PC + (imm . <<12).

这是因为 S type imm 12位, 但理应支持32位的 offset

则 Store/Load with PC相对32位 offset / 32位绝对地址:

auipc x5, <high20> / lui x5, <high20>

sw/lw rd, (<low12>) x5

↳ store/load ↳ 相对offset / 绝对address.

最后: 拿过来 32bit instruction, 如何看出指令?

step1: 查 opcode / funct3 / funct7 ⇒ type / operation.

step2: 找 rs1 / rs2 / rd / imm values (若存在)

KOKUYO

No.

Date

CA review: CALL : Compiling + Assembler + Linker + Loading

Main Topic: C program from storage (disk or flash) into a program running on a computer

C program → Compiler → Assembly → Assembler → machine code

library routine (machine code)

Memory ← Loader ← 可执行机器码程序 ← Linker

Compiler: C → Assembly

Assembler: 汇编中可能有伪指令, Assembler 基本任务是: Assembly → machine code

具体而言: assembly → object file, a combination of machine 指令, 数据 以及指令放在内存中位置的信息

指令中的 label 地址将会十分重要, 故 assemble 会将追踪 labels 并记录它们与 data transfer instructions* 在 symbol table 中

对于 UNIX, object file 有以下六个信息:

① header: the size and the position of the other pieces of obj file

② text segment: contains machine code

③ static data segment: data allocated for the life of program

④ relocation information: 当 program load 进内存时, identifies instructions and data word that 依赖于绝对路径.

⑤ symbol table ⑥ debugging info

Linker: 有时要调库 (如头文件, 其它.o), 要把所有独立的 machine language programs 都"拼"在一起

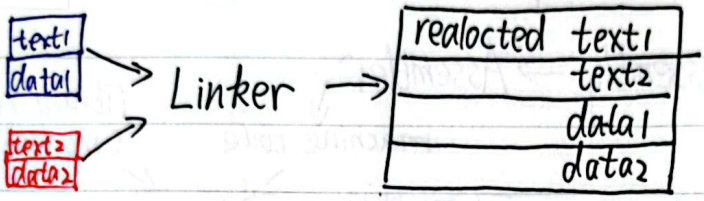
它一共有三步:

① * Place code and data modules symbolically in memory

KOKUYO

② Determine the address of data & labels

③ Patch both the internal and externel references
↳ fill in absolute addresses; must relocate to reflect location

*: 


Loader: 可执行文件 : disk → memory , and start it.
Read header → create an address for text & data
→ copy data & 指令 至内存 → 初始化 reg , set stack pointer
→ Jump to start-up routine , calls main routine of program

Ex: When are the machine codes generated ?
1) add x5, x6, x7 ⇒ After assembly
2) jal x1, fprintf ⇒ After assembly
Determine ? 1) After assembly 2) After linking

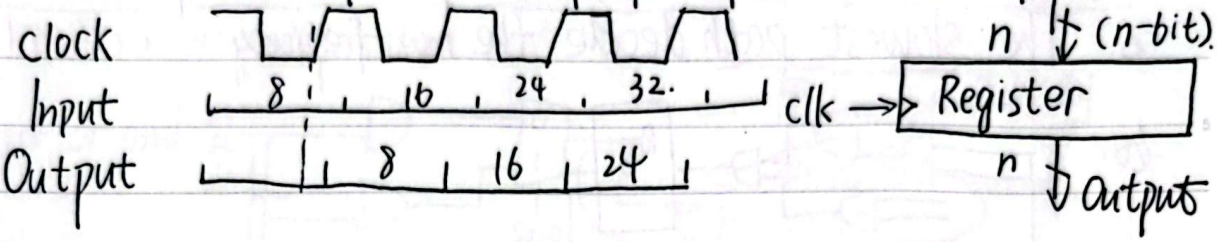
No.

Date

# CA Review: Digital Circuit - State elements

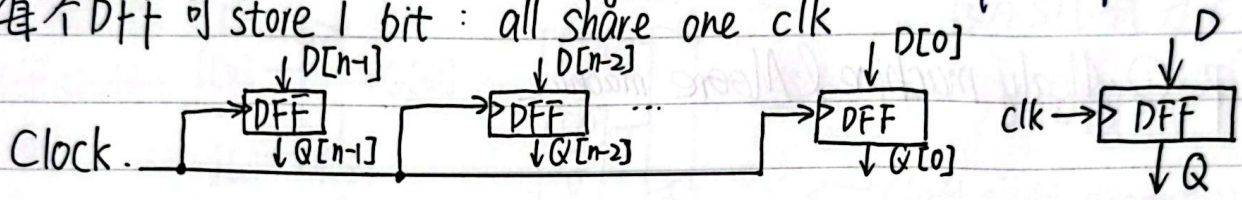
(Digital System ; Combinational logics 略)

Registers: 在clock上升沿时，采样input并输出

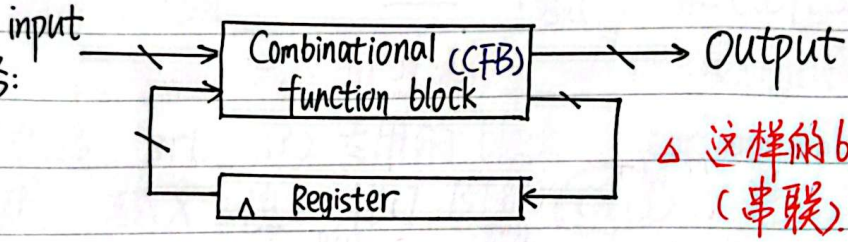



In side a reg: Implemented by multiple D-flip-flops (DFF)

每个DFF可store 1 bit : all share one clk



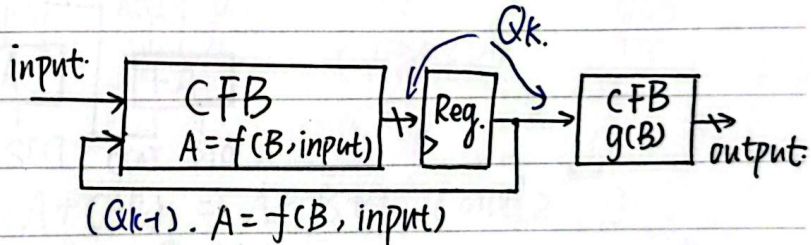

同步电路:




△ 这样的block可重复叠多块(串联).

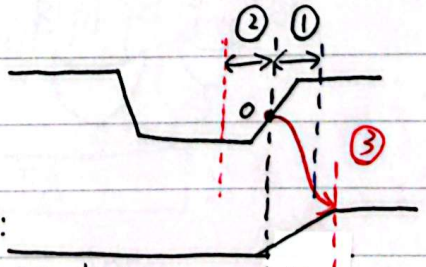
同步电路表示FSM

input & $Q_{k-1}$ ⇒ $Q_k$ & output




某种程度上: ISA (指令集架构) 也是一种FSM

☆ Timing in synchronous circuits: clk:




① Hold Time ② Setup Time

D信息在①+②期间应保持不变 Q:

同时, Q在上升沿中期(o点)(上升沿可视为非常短暂)可变化,

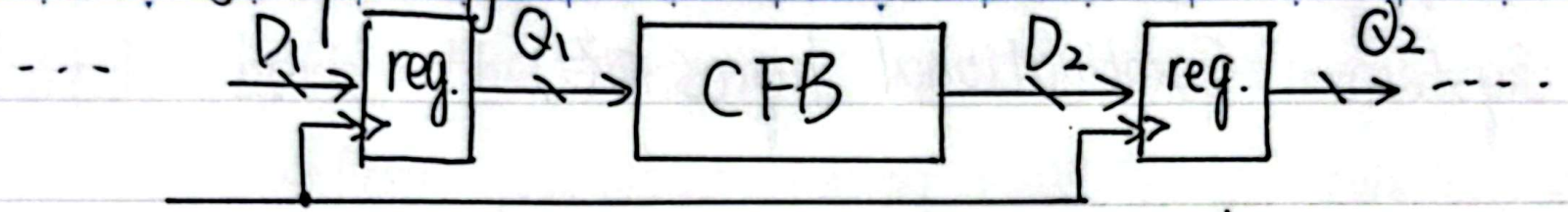
但需在③: $t_{clk\text{-}to\text{-}Q}$ 后才能到达稳定信号

KOKUYO

No.

Date

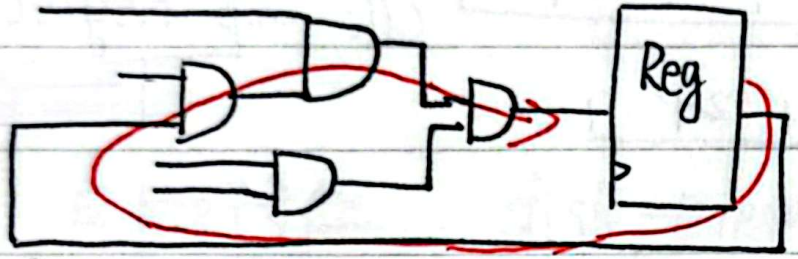
可见，clk frequency 不能∞，有限制！如下电路：




则：$t_{ck\text{-}to\text{-}Q} + t_{CFB} \leq$ min clock period - setup-time.

△. The slowest path decide the max frequency (critical path).

如：




ns及注意：$\frac{1}{1ns} = 1000MHz$, $1ns = 10^{-9}s$

No.

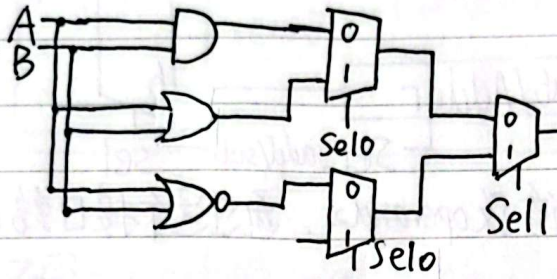
Date

# CA Review: Datapath

## Useful blocks:

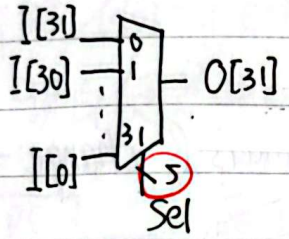
① ALU: arithmetic instructions 有: add sub slt sltu sll srl sra. 异或 或 与门 xor or and.
加减 store if less than shift.

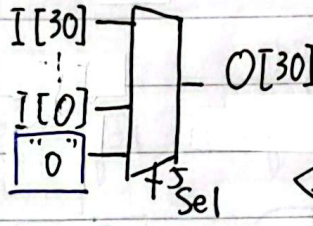
对与 xor or and:




用 SeloSel1 来控制输出哪个位运算.

对于 shift:



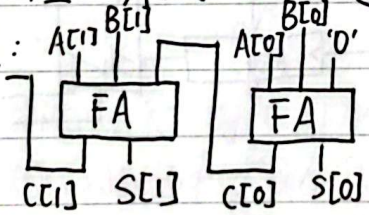

因为 SLL 就是 0 填... 用 0 填充 ↑

⇐ (left shift example).

5位 input, 因为 shift 命令就是移动 imm 低 5 位代表的十进制数.

类似可构造 srl sra 专用的 block (or: notation: C[i-1]).

对于加减: 输入 3 位: A[i] B[i] C[i] (进位) 输出 result[i] & 下一位进位。故:

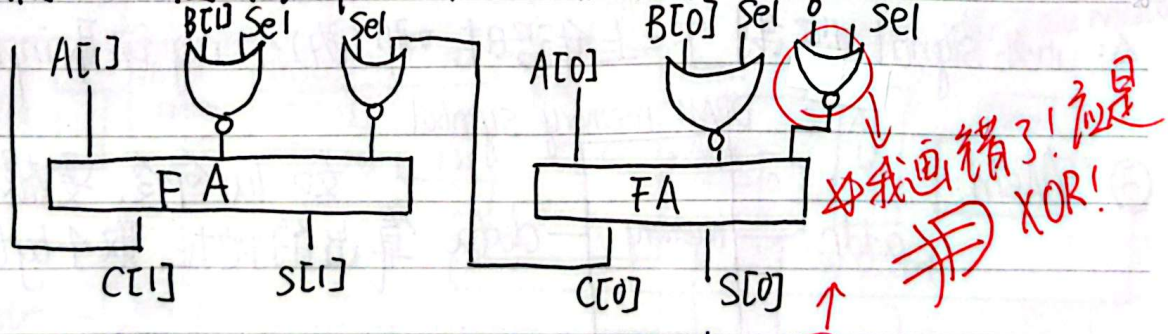



(ADDER).

而减法呢? $A - B = A + (-B) = A + \bar{B} + 1 \pmod{2^{N-1}}$

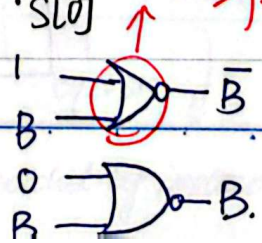
则 B[i] 输入时, 应有元件控制是否翻转.... XOR Gate! *

故:...




由 Sel 信号控制: 1. 则是减, B[i] 反转 (包括 0 位的进位: 0 => 1).
0. 则是加法

*:



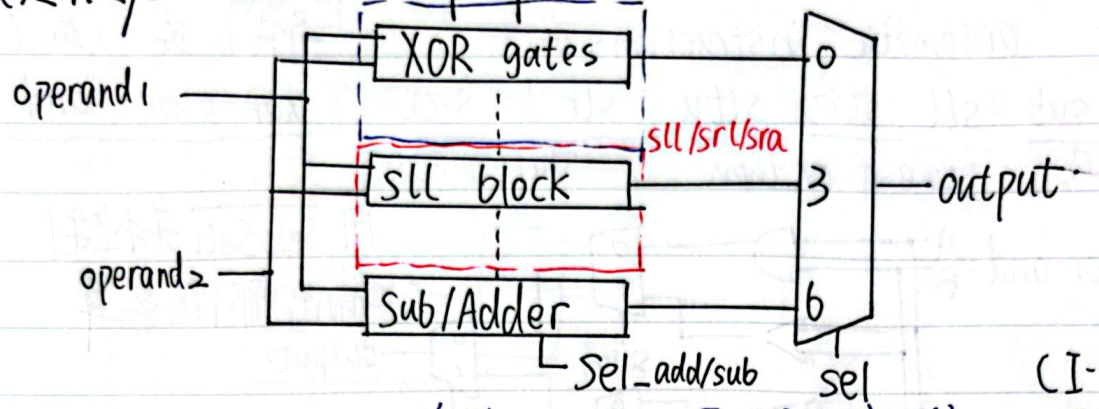

KOKUYO

No.

Date

至此，三种操作：add/sub，shift，and bit operation，都准备了block

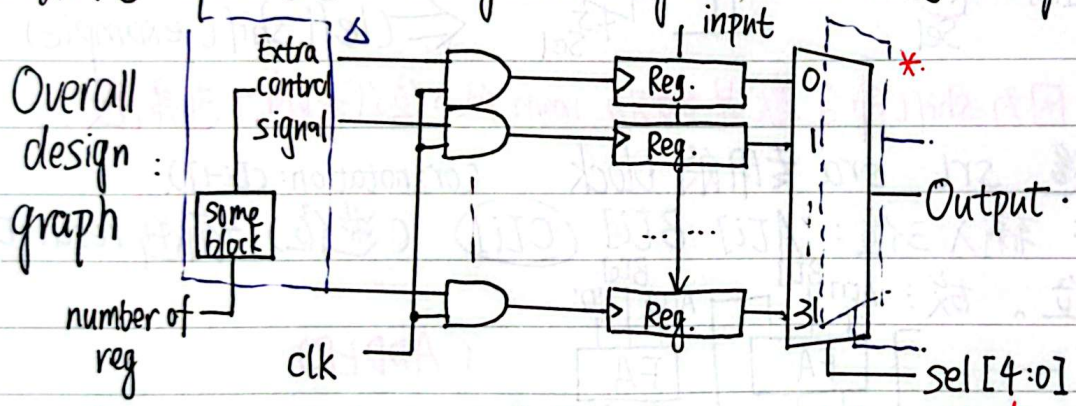
最终依据 sel 信号选择即可：xor/and/or




△：对于 immediate，只需修改 operand2，通过这个接口给 imm，而非 reg 中数即可

② Reg. ⇒ Register file.

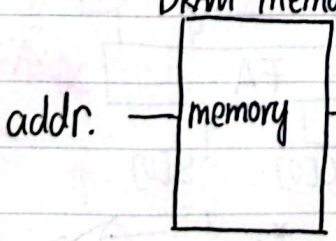
功能：provide data given register numbers & change the stored value




*：这个 32-multiplexer 可能用 5层 2-multiplexer 搭建；且一个32-plexer 输出一个 reg value，故此处应用两个 32-multiplexer！

△：此处 signal 控制下一上升沿时，哪(两)个 reg 读取 input 值

③ Mem:




如 lw 命令，要根据 offset 算出的地址，取4 byte 过来。

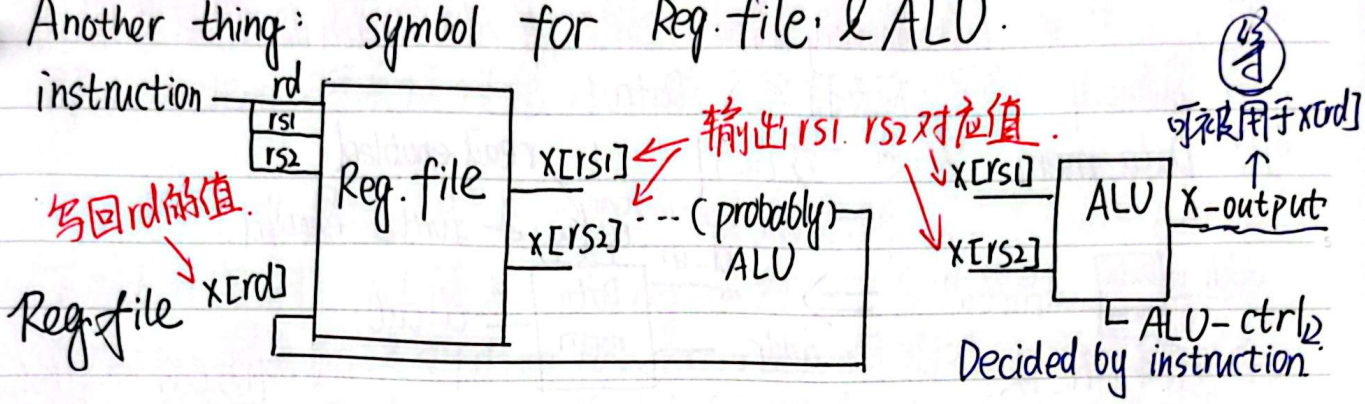
No.

Date

介绍了 useful blocks，接下来关键便是如何运用它们于指令！

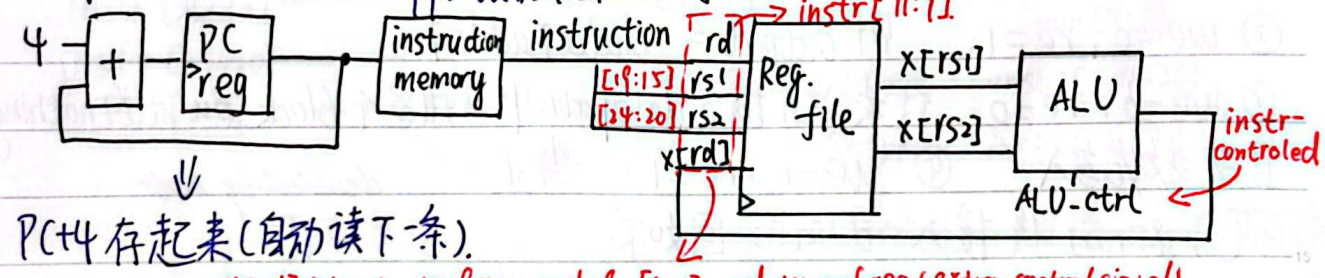
如：明显地：ALU 将会被 R-/I- type 指令使用

Another thing: symbol for Reg. file & ALU.




下面，将介绍不同 type 中的 datapath 搭建：

R-type 说白便是用 rs2 中值对 rs1 作操作最后给 rd 值



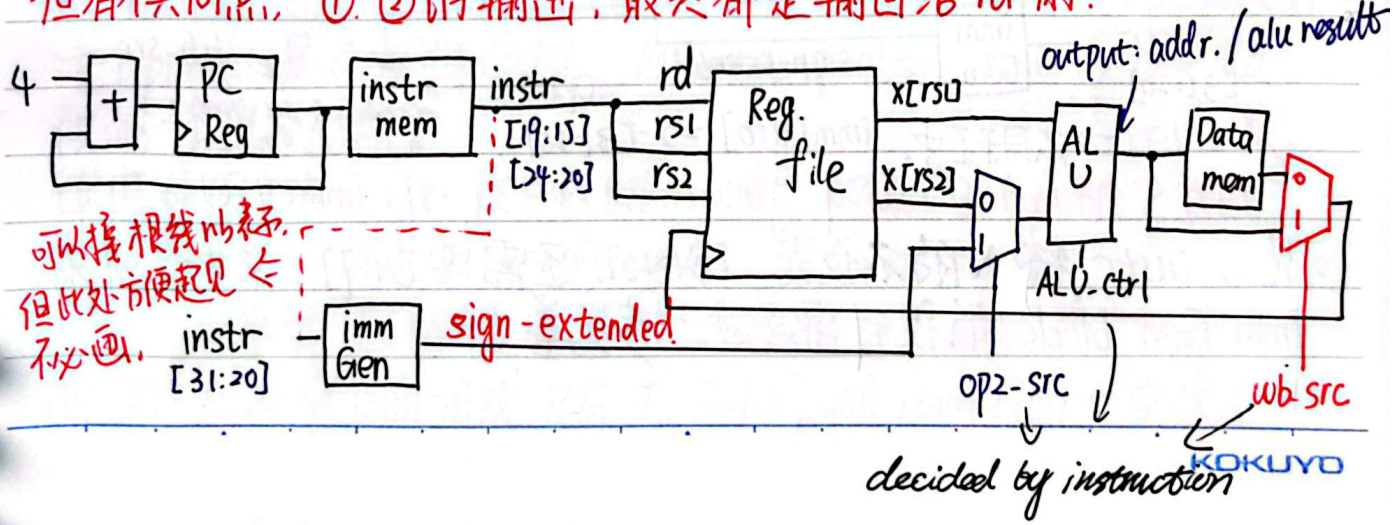

I-type: 回顾一下 I-type 与 R-type 的区别：

① I-type 中有许多操作与 R 中类似 (addi. slti, ori)，只过一 reg 一个 imm

② 除①之外，还有 l w/b/h，要从 Data memory 中拿数据

sign-extended

但有个共同点，①②的输出，最终都是输回给 rd 的！




KOKUYO

No.

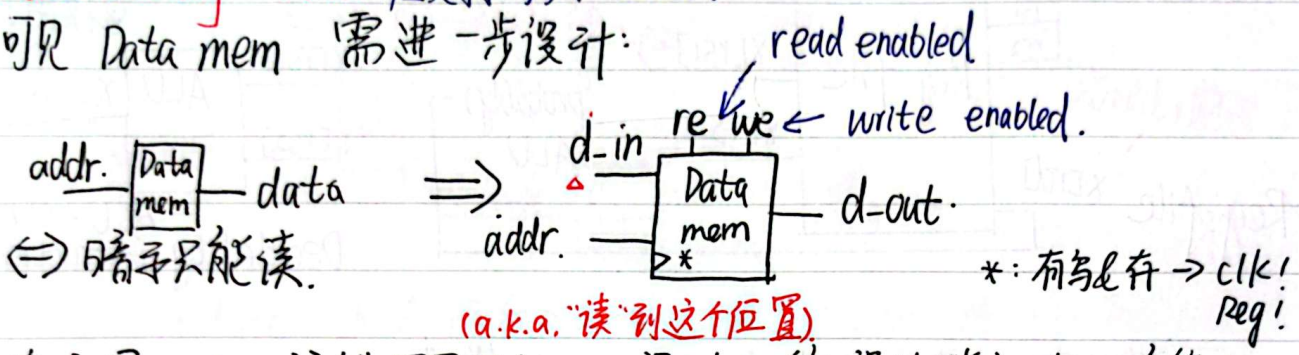
Date

## S-type：在开始前，要厘清 S-type 在干什么：如 sw: rs2, imm(rs1):

store word at rs2 to memory addr. = ( X[rs1] + imm).

可见，这里进 Data memory 的 addr. 依然是 ALU 的输出，但这里的 Data memory block 应支持写入 Data！

可见 Data mem 需进一步设计：



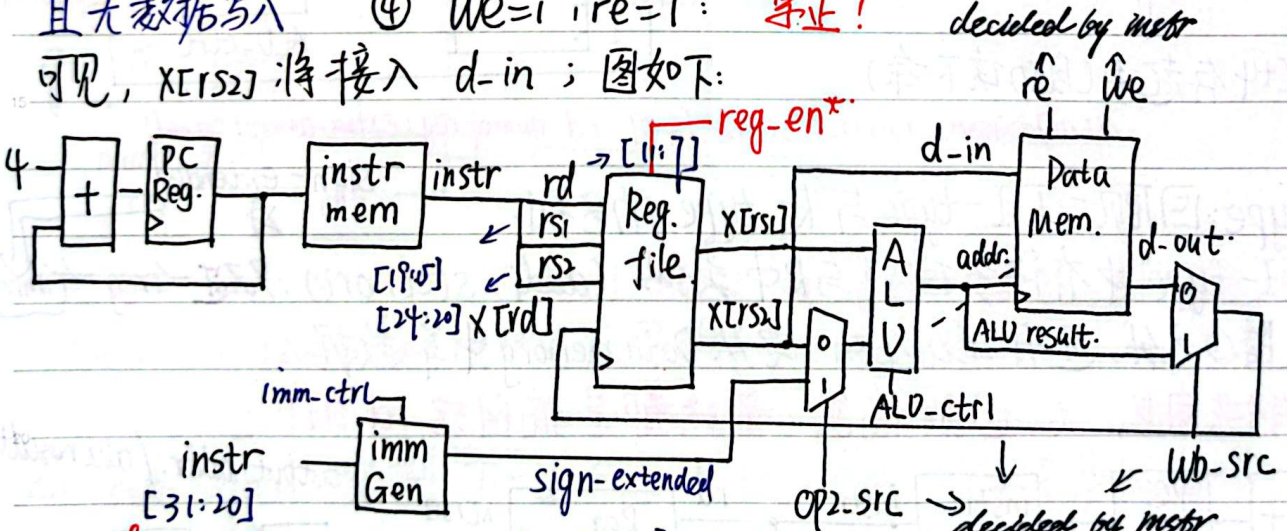

在这里 addr. 依然可用于定位，但 d-in 的设计指定了写入啥值；而 re. we 将控制此时 block 是读 or 写，i.e., 控制 block 行为：

① we=1, re=0, 则 addr. 写入 d-in 值., d-out 为 nothing

② we=0, re=1, 则 d-out = data[addr.]

③ we=0, re=0, 代表这个指令 datapath 中不用这个 block, d-out 为 nothing 且无数据写入 ④ we=1, re=1: 禁止！

可见，X[rs2] 将接入 d-in；图如下：




& [11:7] ⇒ 不同于 I 中, imm[11:0] ⇒ [31:20].

因为 S 的 machine code 中, imm[11:5] => [31:25]; imm[4:0] ⇒ [11:7]

可见，instr 输入的不仅是 [31:20], 还需要 [11:7]；这也意味着，imm Gen block 的设计需要进一步完善！△

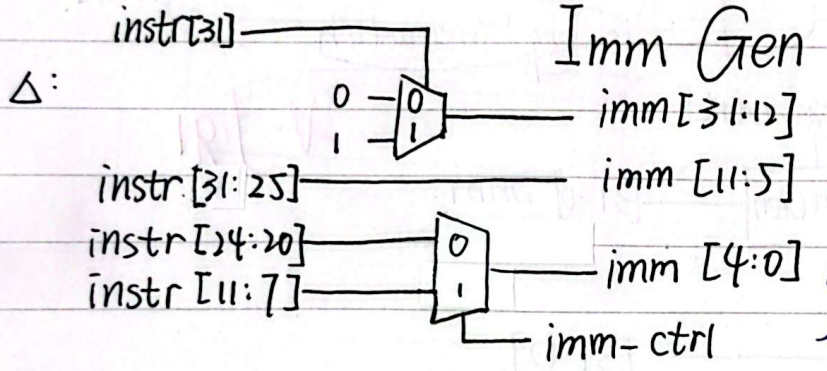
No.

Date

*：同时，reg file block 加入了额外的 control signal：reg-en

why？假设 sw 之后，d-out 为 nothing；虽然 S-type 无 rd，但 I[11:7] 处依然是5位，且传给 reg. file（但其实这5位是 imm[4:0]！）那么，nothing 将会误传给 rd！故要 reg-en 控制 wb 决策！

那么可见，若是 lw 或 R-type，rd 的更新将在下-clk 上升沿；下-上升沿时，rd 读入（if reg-en 为1），与此同时，PC reg 给出 instr，将会陆续拆出新 rd，rs1，rs2；但前者进程更快，故新 reg 已来时，旧 rd 已读入，已准备好被读了（如上一个 rd 是现在 rs1）




* 还有 bne. blt. bge；可让 ALU 伸出 portal，并 instr 控制发挥

B-type：eg. beq rs1, rs2, L (imm/label)，若 X[rs1] == X[rs2]，则前往 L（PC+imm/label；但机器码用 imm/label-PC）

如何简便地在上一个版本的图（S-type）中修改，以支持 B-type？

回想 ALU 中计算过 X[rs1] - X[rs2] 的！可以用一个大或门判断32-bit结果是否为0！因此，ALU 可以额外出一个 zero portal.

这个 portal 是否发挥作用，应由 instruction machine code 提供的信息决定。

发挥作用时：
$$\begin{cases} PC = PC+4, & \text{if zero} \neq 0 \\ PC = PC+imm, & \text{if zero} = 0 \end{cases}$$

可见：原来的 PC+4 简单逻辑也需要进一步完善

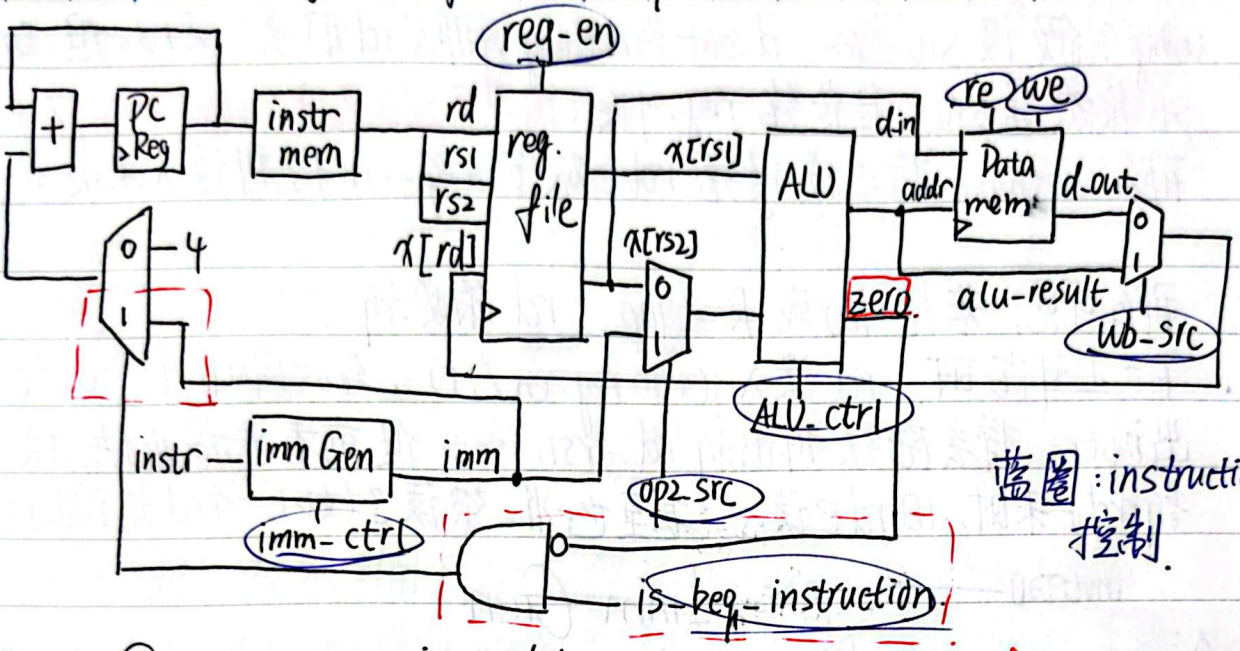
而 B-type 中 imm 组成又与 I. S 不同，故 imm Gen 也要改

KOKUYO

No.

Date

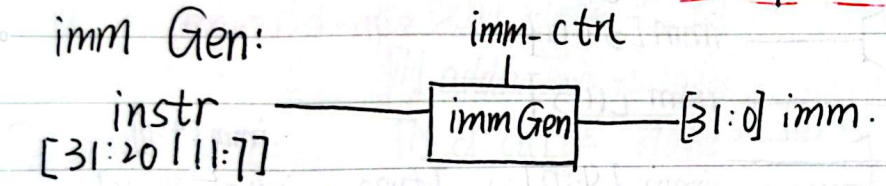
以下是支持 R.I-type & bge 的 datapath 图：

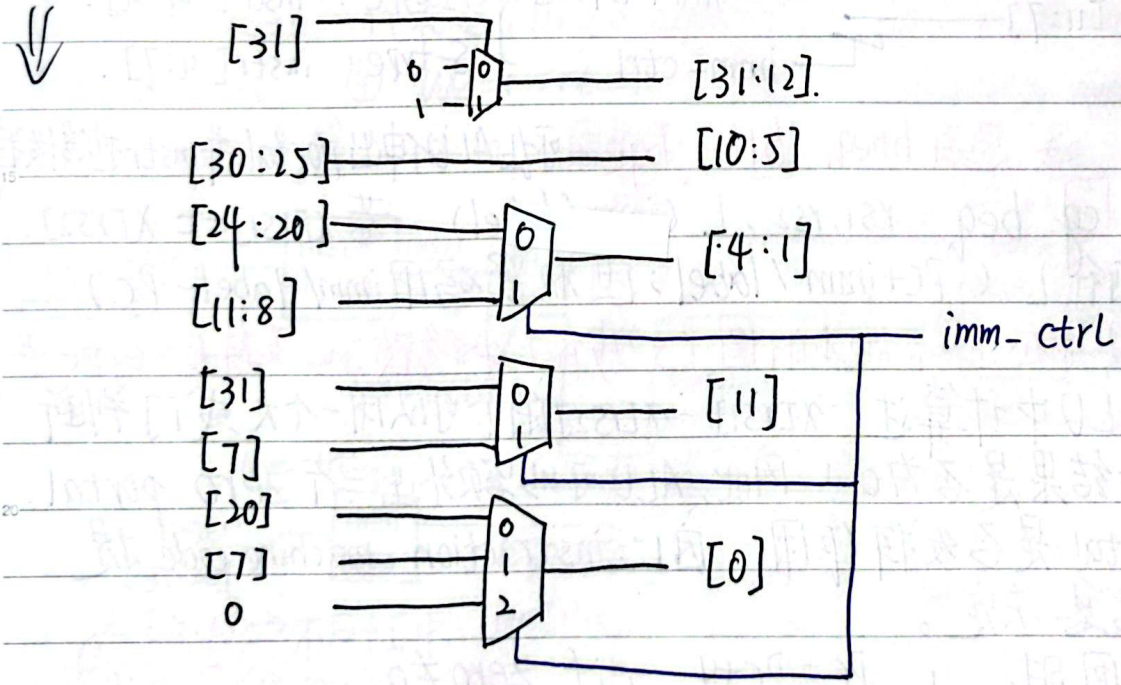



蓝圈：instruction 控制

★fig1

imm Gen:




Campus

No.

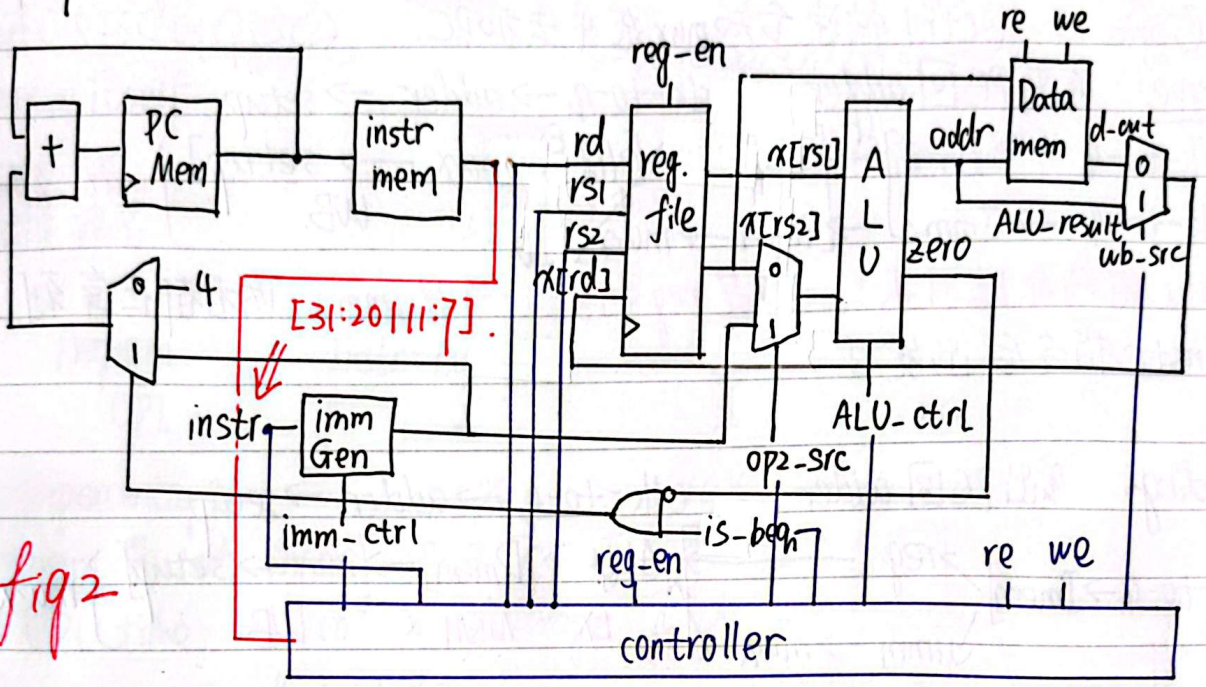
Date . .

## Controler:

从之前能看出：许多控制器件的信号由instr而来，有：

reg-en　re　we　alu-ctrl　imm-ctrl　wb-src.

op2-src　is-beq.




~~fig2~~ fig2

controller 接input (instr). and output a series of ctrl signals

为了保证 信号到器件时，ctrl signal 已到达，故 instr 先过 controller，由 controller 分发 ctrl signal 以及 rd. rs1.rs2，instr[31:20|11:7]

但画图采用 fig1 (题中) 也可以，只不过需要明白，ctrl-signal. instr[31:20|11:7] regs 是谁给！(甚至部分题目算timing可能不考虑 controller 的延时).

Timing：在Datapath图中，不难发现 path 由5部分组成：

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|
| instr fetch | instr decode | execution | memory access | write back |

下面讨论各个 Type 延时：

不论是 ~~[rd]~~ → reg.file

还是 PC±imm → PC.reg

No.

Date

IF 无mem环节! ID EX

R-type: [clk-to-q (PC) → Imem] → [Reg file → mux] → [alu]
→ mux → [setup] WB → 关键setup是PCreg的!

同时，q出的PC回到adder也有时间：clk-to-q → adder → setup
但一般比上一条path快 (assume；且假设control signal is fast)
AdLogic ↓ctrl快速告知mux选4去加PC.

Itype: q出PC回adder : clk-to-q → adder → setup

[clk-to-q → Imem] IF → [reg] ① → [alu] EX → [mux → setup] WB
[clk-to-q → Imem] → [imm → mux] ② ID
} max，无mem

alu前两条路过来！一个是reg中值，一个是imm；两者都在拿到instr指令后出发！

Loading: q出PC回adder : clk-to-q → adder → setup

[clk-to-q → Imem] IF → [reg → [alu] EX → [Dmem] MEM → [mux → setup] WB
[imm → mux] → [alu] ID
} max. 结!

因此，load操作是耗时最长的!

S-type: clk-to-q → adder → setup

[clk-to-q → Imem] IF → [reg] ID. → rs[rs2] → [Dmem] Mem
[imm → mux] → [alu] EX → addr.
} max. 无WB

B-type: 现在，clk-to-q → adder → setup，不是这么简单了!

[clk-to-q → Imem] IF → [reg → mux] ID → [alu] EX → [and → mux → adder → setup] WB.
→ imm
} max
无Mem

Campus

No.

Date

# CA Review: Pipeline

Performance: $\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instructions}} \cdot \frac{\text{Time}}{\text{Cycle}}$

简称：CPI

factors:

- ISA (RISC vs CISC)
- Program itself
- Compiler
- 编程语言

- Compiler
- Program

Microarchitecture implementation or circuit design / ISA

Eg:

| Program: | Instr-type: A | B | C |
|---|---|---|---|
| CPI | 2 | 2 | 4 |
| percentage | 20% | 40% | 40% |

Program 有 $10^6$ instr, 以及 CPU 频率：2.5GHz

则 CPU time $= 10^6 \times$ Average CPI $\times 1/2.5\text{GHz}$.

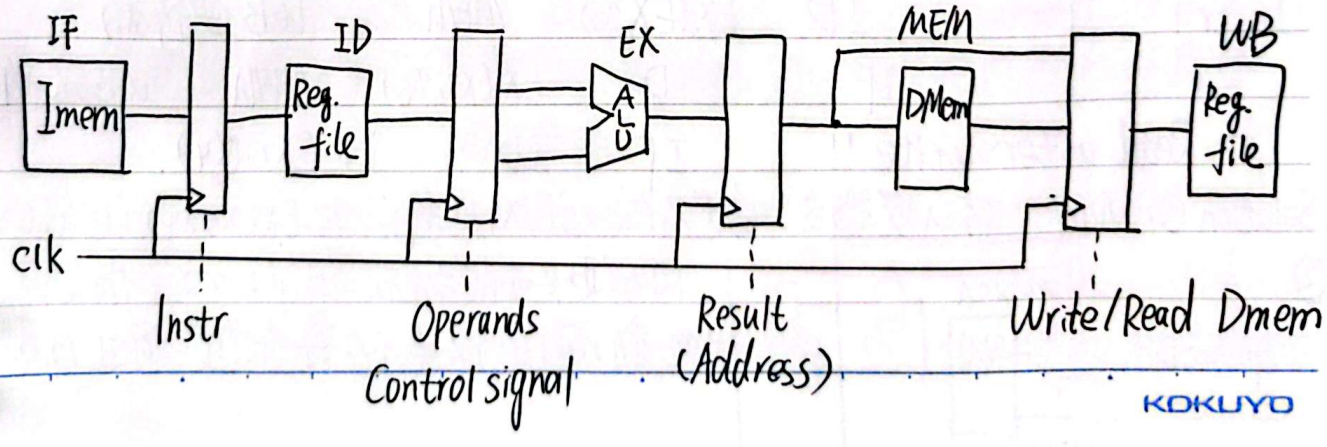
$= 10^6 \times 2.8 \times \frac{1}{2.5 \times 10^9} = 1.12 \times 10^{-3}\text{s} = 1.12\text{ms}$

如何提高 Time/Cycle，若仅是 $t_{critical\_path} \le t_{clk}$，一个path里面至多可经过5大组件 (IF. ID. EX. MEM. WB).

但若五个阶段视为5个收费站呢？而不是车必过5站，下辆车才可进？

这样，Pipelined CPU: $\max\{t_{IF}, t_{ID}, t_{EX}, t_{MEM}, t_{WB}\}$ (顶尖是 $\Sigma$).

那则需要设计"站"以让车有序过站了，⇒ reg!

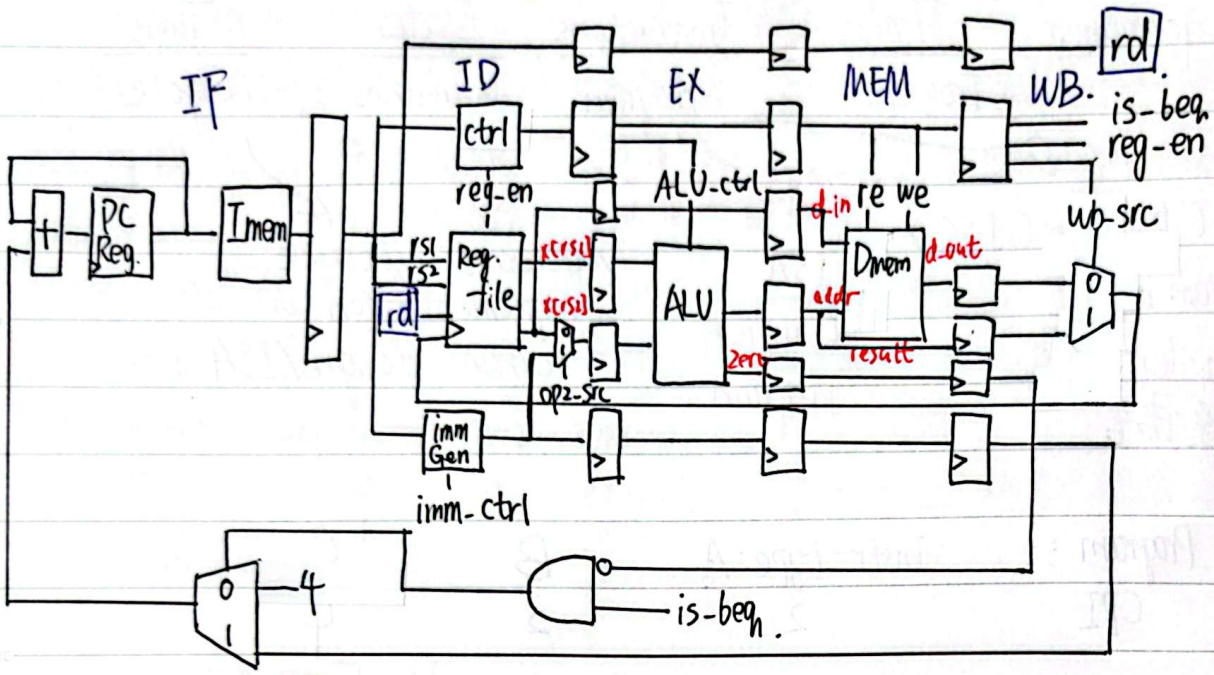



KOKUYO

No.

Date

同时注意到：rd不能赖在reg.file中不走，直到x[rd]来！故rd也要流动！




Structure

Hazards: ① CC1 CC2 CC3 CC4 CC5

Instr: 1 IF ID EX MEM X WB X

2 - - - - - -

3 — - -

4 IF X ID X

X: Dmem与Imem 说到底，在一块内存上！

X: Reg.file 能又存又写么？有先后顺序么？ to use physical resources

可在硬件上解决问题（从解决 Data&Instr Mem issue）；从及指令 take turns

② Data.1) CC1 CC2 CC3 CC4 CC5 CC6

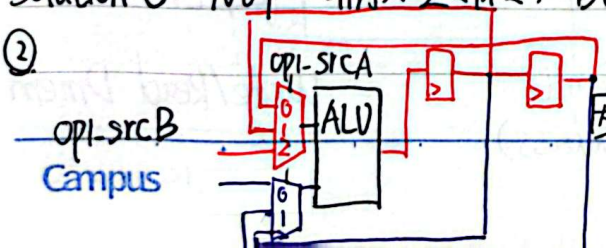
Instr.1 IF ID EX($x_2$, $x_3$) MEM WB (更新$x_1$)

2 IF ID EX($x_5$, $x_6$) MEM WB(更新$x_4$)

3 Read after write!! IF ID EX($x_1$, $x_4$).

Solution ①. NOP: 输入空指令，啥也不做；让$x_4$更新后，再 EX($x_1$, $x_4$). (Wait!)

②




因为冲突可能发生在相邻1or2条指令，因此新reg值但未不及存的值，通过mux

Campus

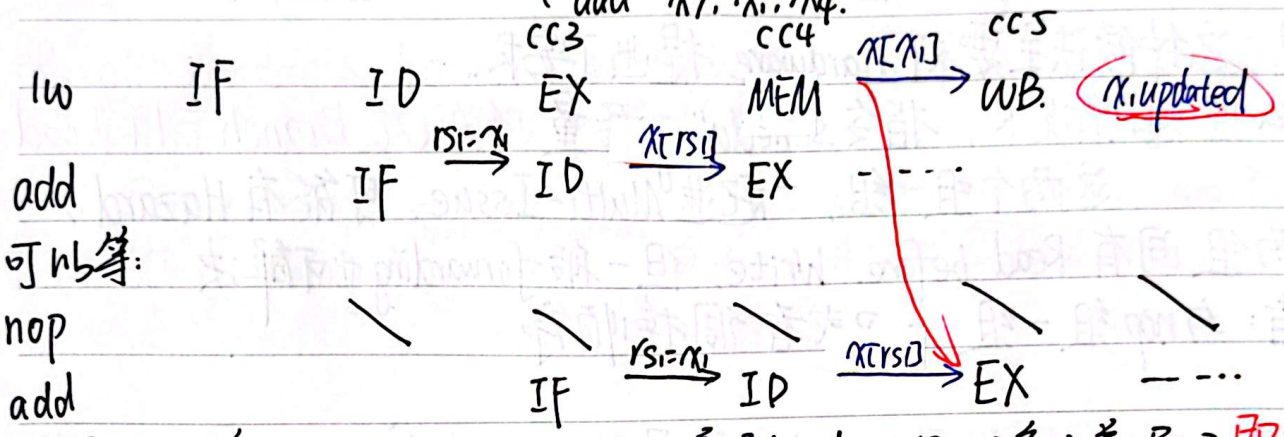
No.

Date

一言以蔽之：ALU结果值得以在1or2轮后返回Mux，在被存入Reg前就被可以利用，解决RAW干扰

而关于op1-src，大致上先可用reg_en判断：为1时，它便会注意之后是否要提前于WB而读它；而op1-src具体多少，取决于之后几条(1/2)命令后是rs1还是rs2要利用它

2). 'load delay slot'：{ lw x1, 0(x3)
add x7, x1, x4.




在隔1.2条instr时 Data Hazard 意料之中；但3条之差呢？即：能又写又读么？一般认为，这也有Hazard

或者 Code Rescheduling，将不冲突、不影响的指令先放lw后(compiler实现)

③ Control：考虑一个beq指令，若beq在3阶段后令PC跳至一个地方，那么前面pipeline中在跑的beq后几条指令岂不是全错？

一种解决方案是用nop等！待3阶段后PC给出正确address，再继续

另一种是照常：若不jump，则OK；若jump，则 额外花时间flush前四条指令所对reg. mem. 做的操作！

或者：Dynamic branch prediction: Predict if the branch will be taken based on the current record

最后，也可考虑类似于rd解决方案：forwarding to reduce delay of branches (将is_beq信号一出ALU就送过去).

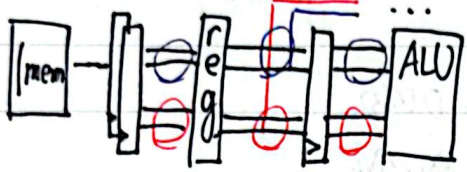
KOKUYO

No.

Date

## Multi-Issue:

在之前讨论情景中，1个 clock cycle 中最多1个 instruction 有问题
考虑如何提高并行效率：①是 加深流程，如一个环节拆成2个
②是'多几套这样的流水线'，这称为 multiple issue
在②做法下，原来 pipeline 中的 CPI ≥1，可能变为 CPI <1

fig: 

两套！比方：洗衣房中一湿一烘
⇒三湿三烘

可见，这种做法主要对 Hardware 提出了要求。
那么理想情况下，指令 scheduling 尽量 1个ALU or Branch，1个 Load or Store，这两个组一组，一起进 Multi-Issue. 可能有 Hazard，如两组间有 Read before Write，但一般 forwarding 也可解决
或者：与 nop 组一组；又或者 调换顺序

可见 scheduling 是重要的！它可能是 compile time 或 execution time 来 'schedule' 的 (又名: Packaging)
Static ↑ Dynamic (by hardware)*.

In most static issue processors: partially handled by Compiler
In dynamic issue designs: be normally dealt with at runtime by processor, although compiler often have already tried to help improve by placing instructions in a beneficial order.

而对于 dealing with data/control hazard:
hardware technique
Static: compiler handles some or all data/control hazards.
Dynamic: processors attempt to alleviate some classes of hazards

*: Also called superscalar:
示意图：IF & ID → Reservation Station → Integer ALU
→ R S → Float Unit
→ R S → Load/Store Unit
→ Commit Unit with reorder buffer

Multi-issue is not Multi-core, nor SIMD; it can be combined with pipelining, SIMD, etc

No.

Date

# Paralellism

Motivation: 2000年后，processor 遇到了功率墙 (power wall)
问题：虽可加晶体管数量，但因功率与热量限制，提升性能变得困难

Tow ways to improve performance:
① 多道程序：multiprogramming，并行运行多个程序
② 并行：让一个程序更快 topic

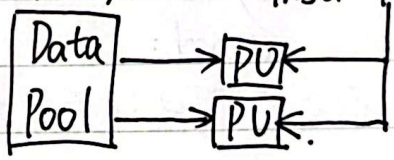
Recall Amdahl's Law: Speed up = $\frac{1}{(1-F)+F/S}$ 提升量 受影响比例

关于并行提升效率，有两种描述法：
① Strong scaling：Size of problem 不变，运行时间 $\times\frac{1}{n}$
② Weak scaling：变大，并行效率提高 (×n)，运行时间不变
↓更易实现 ↘更难，但更理想

Flynn's Taxonomy：将指令集与数据流并行性分为四种：
SISD SIMD MISD MIMD
↓
Single instruction, single data

但SIMD不同：利用单一指令流来同时处理多个数据流：

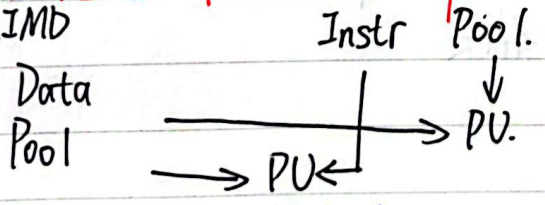
SIMD.




PU: processing unit
应用举例：Intel SIMD Instr Extensions：允许多个数据元素并行处理

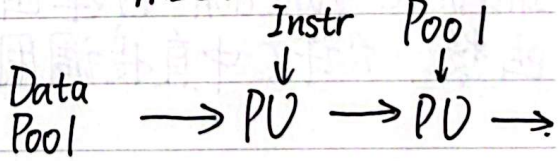
SIMD 是这一节的 main topic！

MIMD




Multicore (之后会讲)

MISD




Rare!

KOKUYO

```
for (int i---
    for (int j-----).
        double  Cij=0
        for (int k=0---)
            Cij+= A [i+ k*N]*B[j*N+k]
        C[i+j*N] = Cij.
```

## What is SIMD ?

考虑矩阵乘的代码：红色部分可视为一种"操作"，其涉及大量数据作mul

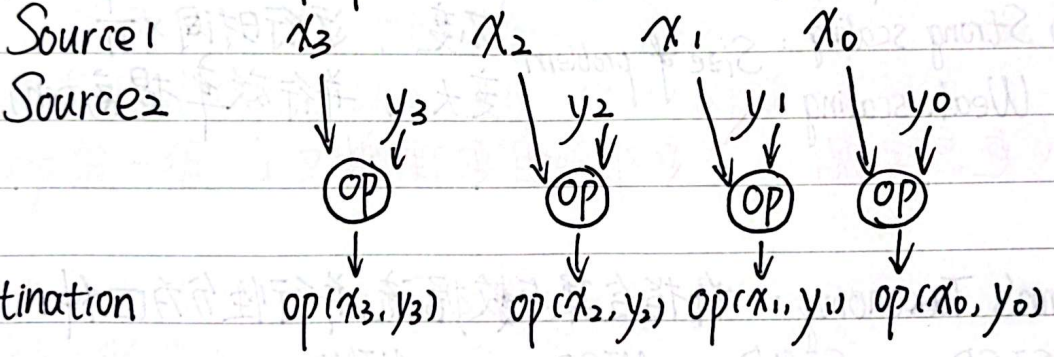
SIMD恰好就可提升此种 Data Level Processing

* One instr is fetched & decoded for entire operation
* SIMD 支持内存的并行访问. 这很重要！

那么现实中如何支持SIMD？ Inter X86 intrinsics 的 SIMD指令集有 SSE. AVX 等命令可允许扩展指令集

一般 SIMD指令会从两个源寄存器组中拿数据，送入op(operation)组件，将寄存器组会将计算结果存在一起

figure:




为了实现上述思想，甚至一些特殊数据类型被设计出来：

Packed byte：64 byte，存8个8bytes (MMX)

word： 4个16bytes

并且有寄存器专门存这种玩意儿，ns支持多个数据打包至单一reg.

Intrinsic Function：内建函数是高级语言中与汇编指令一一对应的函数，允许C中直接调用SIMD指令