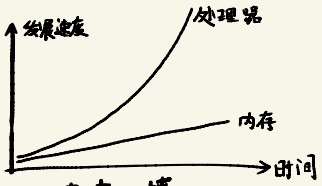
Instruction set architechture（指令集）：连接软硬件的接口



处理器衡量 { response time（响应时间）：完成一次任务的时间 (execution time)
throughput（吞吐量）：单位时间内的任务完成总量

性能：performance = 1 / execution time.

x is n faster than y：$n = \frac{\text{performance}(x)}{\text{performance}(y)} = \frac{\text{execution time}(y)}{\text{execution time}(x)}$

↳ 后面用 CPU Time 之比表示.

Total response time：{ processing
I/O（输入输出）
OS overhead 操统管理任务开销
idle time 空闲时间

CPU Time：忽略 I/O 等其它时间，仅考虑 processing 所消耗的时间.

CPU-Time = Cycles × Time per cycle

CPU Time = CPU Clock Cycles × Clock Cycle Time

降低算法时间复杂度　　提高工艺

Time per Cycle = Instructions × CPI

CPU Clock cycles = Instruction n Count × Cycle per instruction

即 CPI

由 program、ISA、compiler 决定

由 CPU 硬件决定

$$CPI = \sum_{i=1}^{n} CPI_i \times \frac{\text{Instruction count } i}{\text{Instruction count}}$$

relative frequency

{ Algorithm：IC、possibly CPI
programming language：IC、CPI
Compiler：IC、CPI
ISA：IC、CPI、T_c

CPU Time = IC × CPI × T

Power = Capacitive load x Voltage^2 x Frequency

多核：并行编程！

Benchmark：衡量CPU的单位

SPEC：standard performance evaluation corp：develop benchmarks for CPU, I/O, web ...

SPEC-CPU：① 找到 excution time

② $\text{spec ratio}_i = \frac{\text{reference time}_i}{\text{excution time}_i}$

③ $\sqrt[n]{\prod_{i=1}^{n} \text{spec ratio}}$

Amdahl's Law:

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected}$$ （尽可能去优化占比大的部分）

MIPS：millions of instructions per second.（一种单位）

8 Ideas：

① Design for Moore's Law.（设计紧跟摩尔定律）
每两年单位面积上的晶体管数目每2年翻一倍.

② Use abstraction to simplify design.（使用"抽象"）

③ Make the common case fast.（优化最常用的）

④ Performance via parallelism.（并行）

⑤ Performance via pipelining.（流水线）

⑥ Performance via prediction.（预测）

⑦ Hierarchy of memories.（存储器层次）

⑧ Dependability via redundancy.（通过冗余提高可靠性）

Chapter 3. Arithmetic for Computers

3.1 Introduction

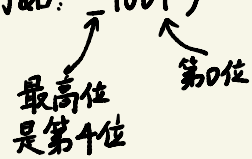
一个 word 默认是 32位 bits. (也可能是 64位)

基本操作：取地址、取操作、读寄存器、操作

3.2 Arithmetic

Overflow：同号相加，异号相减，可能溢出。

判断方法：记最高位是第 n 位。(比已有数字多一位，例如：_1001)



若第 n-2 位向第 n-1 位的进位是 i，
第 n-1 位向第 n 位进位是 j。
若 i≠j 则发生 overflow.

Overflow 发生时：①检测异常发生 ②存下 instruction address (not PC) ③jump

Construct an ALU

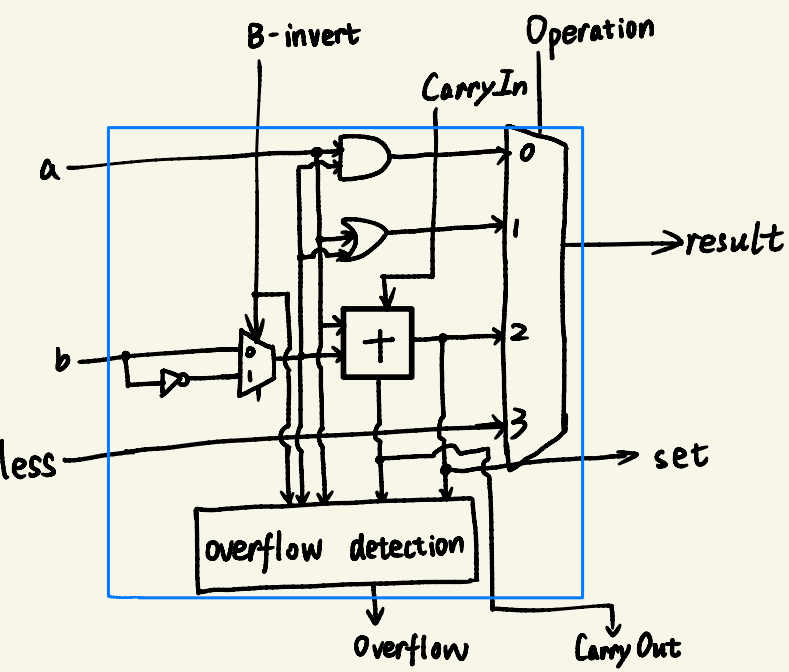
1 bit ALU：AND, OR, ADD

Substract：Inverting b, 1st CarryIn = 1.

再增一个操作：comparison. slt rd, rs, rt.

If rs<rt, rd=1; else rd=0.

* Substract (rs-rt), Use the sign bit as indicator.

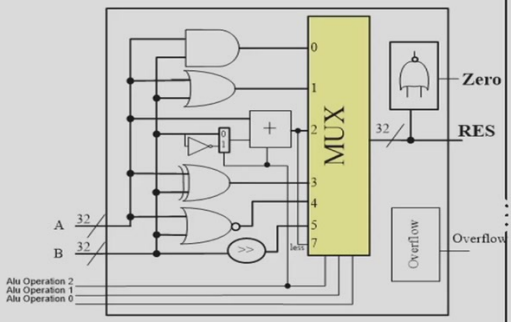


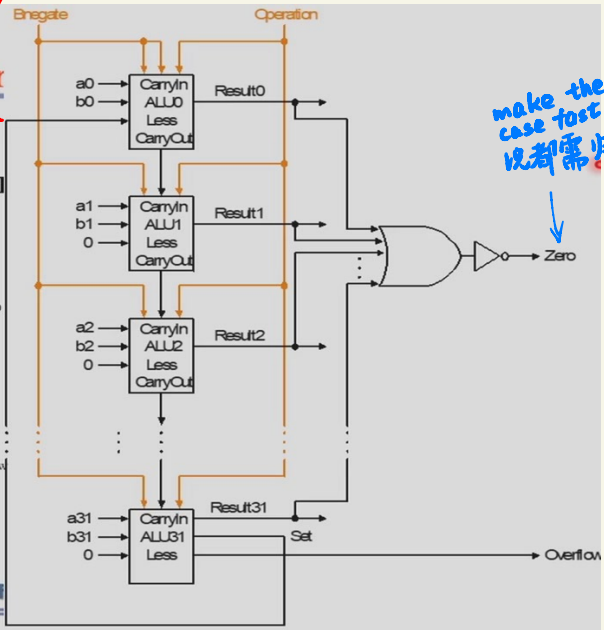
ALU Cell

这根线返回成法的结果。
若小于成立，则 result 0 =1，其余为 0；否则全 0.

Complete ALU —with Zero detector

- Add a Zero detector

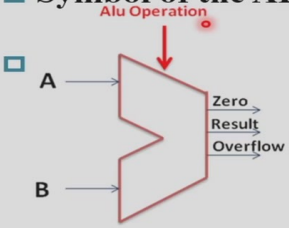




make the common case fast. 很多情况都需归零检测

ALU symbol & Control

Symbol of the ALU



Control: Function table

ALU Control Lines	Function
000	And
001	Or
010	Add
110	Sub
111	Set on less than
100	nor
101	srl
011	xor

→ 右移.

ALU Hardware Code

```
module alu(A, B, ALU_operation, res, zero, overflow );
   input [31:0] A, B;
   input [2:0] ALU_operation;
   output [31:0] res;
   output zero, overflow ;
   wire [31:0] res_and,res_or,res_add,res_sub,res_nor,res_slt;
   reg [31:0] res;
   parameter one = 32'h00000001, zero_0 = 32'h00000000;
     assign res_and = A&B;
     assign res_or = A|B;
     assign res_add = A+B;
     assign res_sub = A-B;
     assign res_slt =(A < B) ? one : zero_0;
     always @ (A or B or ALU_operation)
          case (ALU_operation)
          3'b000: res=res_and;
          3'b001: res=res_or;
          3'b010: res=res_add;
          3'b110: res=res_sub;
          3'b100: res=~(A | B);
          3'b111: res=res_slt;
          default: res=32'hx;
          endcase
     assign zero = (res==0)? 1: 0;
endmodule
```

敏感条件需包括所有赋值等号右端的变量，再加上所有case语句括号内的变量。

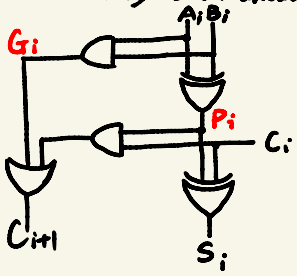
How do you write with overflow code ?

What is the difference The codes in the Synthesize?

```
always @ (A or B or ALU_operation)
          case (ALU_operation)
                    3'b000: res=A&B;
                    3'b001: res=A|B;
                    3'b010: res=A+B;
                    3'b110: res=A-B;
                    3'b100: res=~(A | B);
3'b111: res=(A < B) ? one : zero_0;
                    default: res=32'hx;
          endcase
```

ALU 操作中，仅有加减法延时会叠加，
而与、或等操作是并行操作的，不会有上述情况。

CLA：Carry Lookahead Adder：超前进位加法器



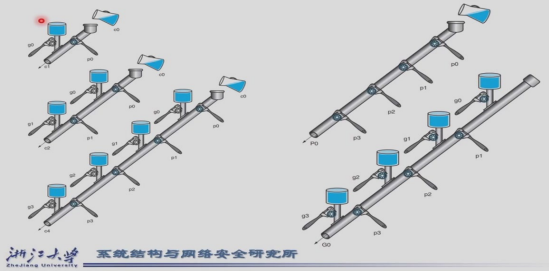
G：generate：进位来源于新生成：A&B 生成。

P：propagate：进位来源于传播：上一级的 C_i。

$P_i = A_i \oplus B_i$，$G_i = A_i B_i$

$S_i = P_i \oplus C_i$，$C_{i+1} = G_i + P_i C_i$

A plumbing analogy



浙江大学 系统结构与网络安全研究所

Extended Example: 16 carry lookahead adder

$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_{0\sim3} + P_{0\sim3}C_0$

$C_8 = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4 + P_7P_6P_5P_4C_4 = G_{4\sim7} + P_{4\sim7}C_4$

$C_{12} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8 + P_{11}P_{10}P_9P_8C_8 = G_{8\sim11} + P_{8\sim11}C_8$

$C_{16} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12} + P_{15}P_{14}P_{13}P_{12}C_{12} = G_{12\sim15} + P_{12\sim15}C_{12}$

$= G_{12\sim15} + P_{12\sim15}(G_{8\sim11} + P_{8\sim11}(G_{4\sim7} + P_{4\sim7}(G_{0\sim3} + P_{0\sim3}C_0)))$

$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

$G_{0\sim3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

$G_{4\sim7} = G_7 + P_7G_6 + P_7P_6G_5 + P_7P_6P_5G_4$

$G_{8\sim11} = G_{11} + P_{11}G_{10} + P_{11}P_{10}G_9 + P_{11}P_{10}P_9G_8$

$G_{12\sim15} = G_{15} + P_{15}G_{14} + P_{15}P_{14}G_{13} + P_{15}P_{14}P_{13}G_{12}$

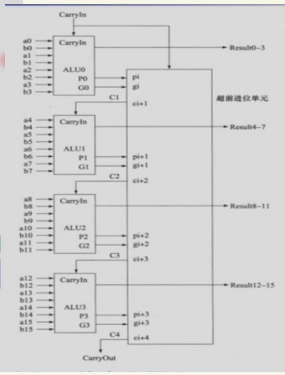
$P_{0\sim3} = P_3\ P_2\ P_1\ P_0$

$P_{4\sim7} = P_7\ P_6\ P_5\ P_4$

$P_{8\sim11} = P_{11}\ P_{10}\ P_9\ P_8$

$P_{12\sim15} = P_{15}\ P_{14}\ P_{13}\ P_{12}$

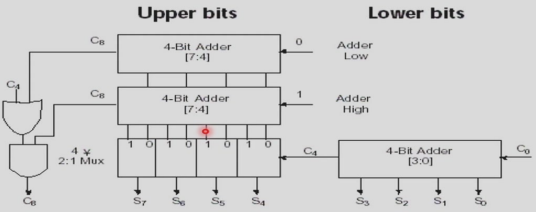
浙江大学 系统结构与网络安全研究所



$C_{16} = G_{12-15} + P_{12-15}G_{8-11} + P_{12-15}P_{8-11}G_{4-7} + P_{12-15}P_{8-11}P_{4-7}G_{0-3} + P_{12-15}P_{8-11}P_{4-7}P_{0-3}C_0$

可看成一个新的 CLA

Carry select adder (CSA)

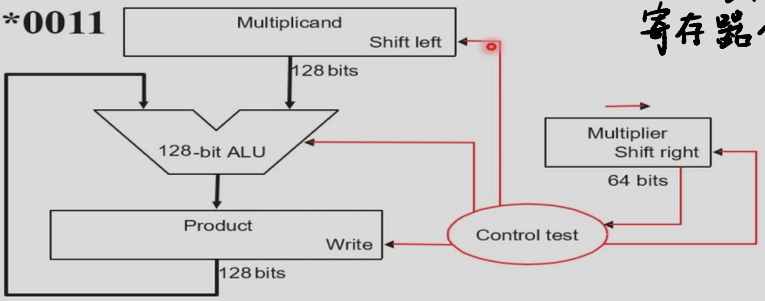


好处：8位计算时间与4位计算相同。
（高4位与低4位并行）
但牺牲了空间复杂度

Multiplicand（被乘数） × Multiplier（乘数）

Multiplier V1– Logic Diagram

- 64 bits: multiplier
- 128 bits: multiplicand, product, ALU
- 0010*0011



2个64位的数相乘，需要一个128位的加法器来做乘法，但每次加法实际是64位操作。且左移、右移后，寄存器会大量空间浪费。

always @ (posedge clk)

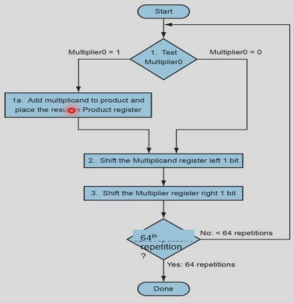
A <= A<<1 ;

被乘数左移，乘数右移。

进行补零　　弹出某位

Multiplier V1--Algorithmic rule

- Requires *64 iterations*
  - Addition
  - Shift
  - Comparison
- Almost 200 cycles
- Very big, Too slow!



Multiplier V2

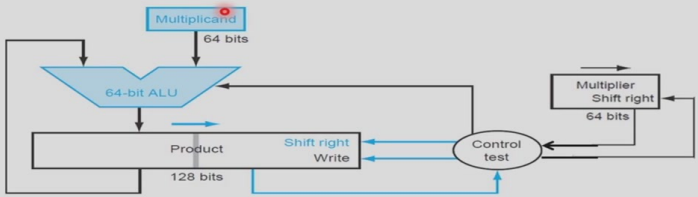
- Real addition is performed only with 64 bits
- Least significant bits of the product don't change
- New idea:
  - Don't shift the multiplicand
  - Instead, **shift the product**
  - Shift the multiplier
- ALU reduced to 64 bits!

```
          1 0 0 0
        × 1 0 0 1
          1 0 0 0
        0 0 0 0
      0 0 0 0
  + 1 0 0 0
    1 0 0 1 0 0 0
```


通过移动乘积
因为实际只有前一半
高位参与了加法运算。

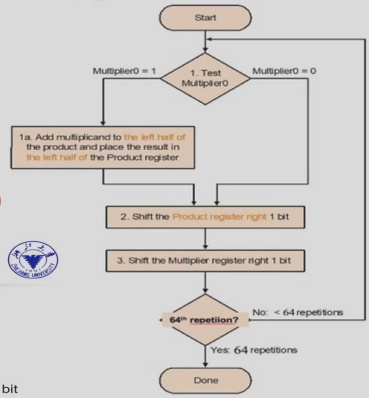
Multiplier V2-- Logic Diagram

- Diagram of the V2 multiplier
- Only left half of product register is changed



Multiplier V2----Algorithmic rule

- Addition performed only on left half of product register
- Shift of product register



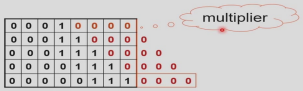
Revised 4-bit example with V2

- Multiplicand x multiplier: 0001 x 0111

```
Multiplicand:     0001
Multiplier:×      0111
                  00000000          #Initial value for the product
1                 00010000          #After adding 0001, Multiplier=1
                  00001000    0     #After shifting right the product one bit
                  0001
2                 00011000          #After adding 0001, Multiplier=1
                  00001100    0     #After shifting right the product one bit
                  0001              #After adding 0001, Multiplier=1
3                 00011100
                  00001110    0     #After shifting right the product one bit
                  0000
4                 00001110          #After adding 0001, Multiplier=0
                  00000111    0     #After shifting right the product one bit
```

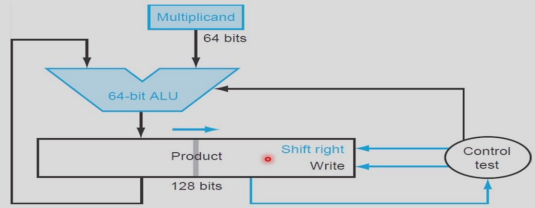
Multiplier V 3

- Further optimization
- At the initial state the product register contains only '0'
- The lower 64 bits are simply shifted out
- Idea: use these lower 64 bits for the multiplier



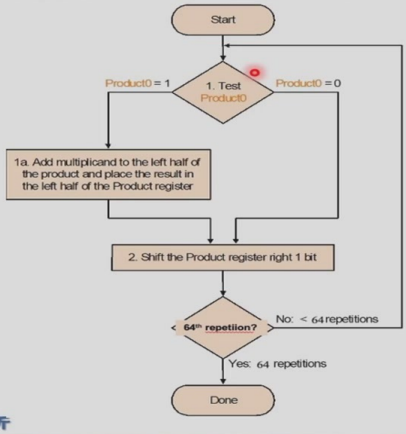
结果寄存器中，长半部分恰好可以被用来储存乘数。这样每进行一次加法，乘数与结果都恰好右移1位。因此无需额外的寄存器来存储乘数。

Multiplier V3 Logic Diagram



Multiplier V3--Algorithmic rule

- Set product register to '0'
- Load lower bits of product register with multiplier
- Test least significant bit of product register



Example with V3

- Multiplicand x multiplier: 0001 x 0111

		Shift out	
Multiplicand:	0001		
Multiplier:×	0111		
	00000111		#Initial value for the product
1	00010111		#After adding 0001, Multiplier=1
	00001011	1	#After shifting right the product one bit
	0001		
2	00011011		#After adding 0001, Multiplier=1
	00001101	1	#After shifting right the product one bit
	0001		#After adding 0001, Multiplier=1
3	00011101		
	00001110	1	#After shifting right the product one bit
	0000		
4	00001110		#After adding 0001, Multiplier=0
	00000111	0	#After shifting right the product one bit

带符号乘法：

① 先转成无符号，计算后再加正负。

② Booth 算法（仅使用加减法）：适用于较多1的乘法情况

shift 代替add.
若 shift 比 add 快很多，则Booth会有较大提速。

Principle -- Decomposable multiplication

- Assumes: Z=y×10111100

Z=y(10000000+111100+100-*100*)

=y(1×2^7+1000000-*100*)

=y(1×2^7+1×2^6-2^2)

=y(1×2^7+1×2^6+0×2^5+0×2^4+0×2^3+0×2^2+0×2^1+0×2^0-*1×2^2*)

= y(1×2^7+1×2^6+0×2^5+0×2^4+0×2^3+0×2^2-*1×2^2*+0×2^1+0×2^0)

= y×2^7+y×1×2^6+0×2^5+0×2^4+0×2^3+0×2^2-y×2^2+0×2^1+0×2^0)

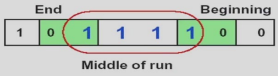
add | Only shift | sub | Only shift

1 | 01 | 11 | 1 | 00

浙江大学 Zhejiang University 系统结构与网络安全研究所

Booth's Algorithm

- Idea: If you have a sequence of '1's
  - subtract at first '1' in multiplier
  - shift for the sequence of '1's
  - add where prior step had last '1'



- Result:
  - Possibly less additions and more shifts
  - Faster, if shifts are faster than additions

浙江大学 Zhejiang University 系统结构与网络安全研究所

Booth's Algorithm rule

- Analysis of two consecutive bits

Current	last	Explanation	Example
1	0	Beginning	00001111**10**000
1	1	middle of '1'	0000**11**110000
0	1	End	0000**11**110000
0	0	Middle of '0'	000011110000

- Action

1 0 subtract multiplicand from left

1 1 no arithmetic operation-shift

0 1 add multiplicand to left half

0 0 no arithmetic operation-shift

- Bit$_{-1}$ = '0'
- Arithmetic shift right:
  - keeps the **leftmost bit constant**
  - no change of sign bit !

浙江大学 Zhejiang University 系统结构与网络安全研究所

Example with negative numbers

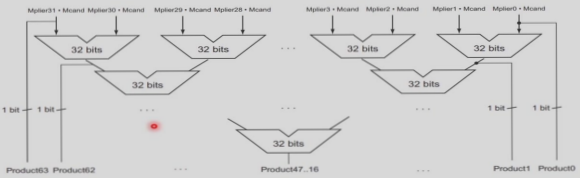
- 2 * (-3) = - 6
- 0010 * 1101 = 1111 1010

iteration	step	Multiplicand	product
0	Initial Values	0010	0000 1101 0
1	1.c:10→Prod=Prod-Mcand	0010	1110 11**01** 0
	2: shift right Product	0010	1111 011**0 1**
2	1.b:01→Prod=Prod+Mcand	0010	0001 01**10** 1
	2: shift right Product	0010	0000 101**1 0**
3	1.c:10→Prod=Prod-Mcand	0010	1110 10**11** 0
	2: shift right Product	0010	1111 010**1 1**
4	1.d: 11 → ***no operation***	0010	1111 0101 1
	2: shift right Product	0010	1111 1010 1

浙江大学 Zhejiang University 系统结构与网络安全研究所

Faster Multiplication

- Unrolls the loop



浙江大学 Zhejiang University 系统结构与网络安全研究所

2个32位数相乘，可视为32次加法。
加法器间并行操作，
因此只需 $\log_2 32 = 5$ 轮即可。

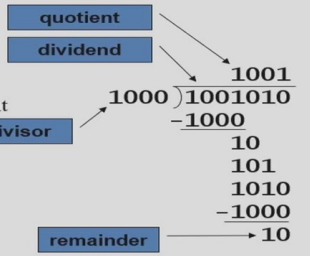
RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 64-bit overflow

浙江大学 Zhejiang University 系统结构与网络安全研究所

3.5 Division

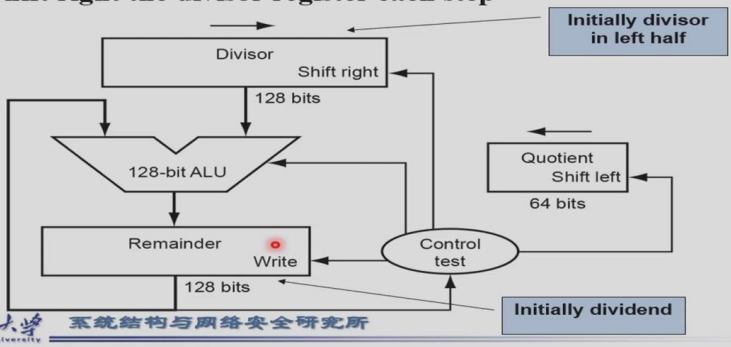
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required



n-bit operands yield *n*-bit quotient and remainder

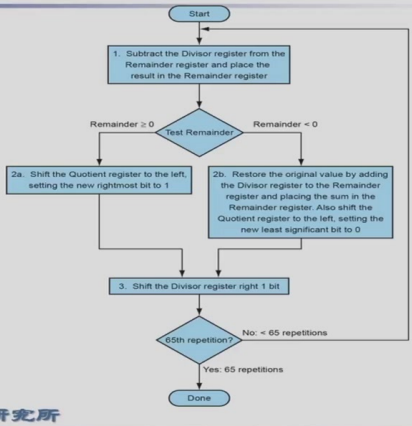
Division V1 --Logic Diagram

- At first,the divisor is in the left half of the divisor register, the dividend is in the right half of the remainder register.
- Shift right the divisor register each step



Algorithm V 1

- Each step:
  - Subtract divisor
  - Depending on Result
    - Leave or
    - Restore
  - Depending on Result
    - Write '1' or
    - Write '0'



Example 7/2 for Division V1

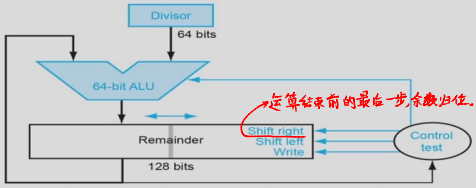
Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem<0 => +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 => +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem 0 => sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem 0 => sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: shift Div right	0011	0000 0001	0000 0001

浙江大学 Zhejiang University 系统结构与网络安全研究所

为何 divisor 放在前64位而 remainder 放在后64位？

Modified Division

- Reduction of Divisor and ALU width by half
- Shifting of the remainder
- Saving 1 iteration
- Remainder register keeps quotient No quotient register required



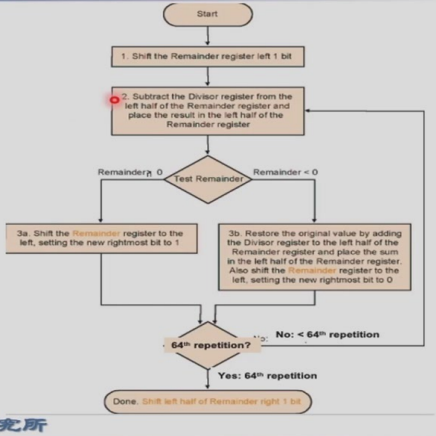
- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

浙江大学 Zhejiang University 系统结构与网络安全研究所

省去商的寄存器，将商存在 remainder 的寄存器内。

Algorithm V 3

- Much the same than the last one
- Except change of register usage



浙江大学 Zhejiang University 系统结构与网络安全研究所

Example 7/2 for Division V3

- Well known numbers: 0000 0111/0010

iteration	step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	**0000 1110**
1	1.Rem=Rem-Div	0010	1110 1110
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0001 1100**
2	1.Rem=Rem-Div	0010	1111 1100
	2b: Rem<0 →+Div,sll R,R_0=0	0010	**0011 1000**
3	1.Rem=Rem-Div	0010	0001 1000
	2a: Rem>0 →sll R,R_0=1	0010	**0011 0001**
4	1.Rem=Rem-Div	0010	0001 0001
	2a: Rem>0 →sll R,R_0=1	0010	**0010 0011**
	Shift left half of Rem right 1		**0001 0011**

→ 余数被多左移3一位，因此需复原

srl 右移
sll 左移

R_0指的是本次减法后的结果，正为1，负为0.

被除数放在 remainder 低位，而 divisor 是和 remainder 高位做运算。
除法完成后，remainder 的高位是余数，低位是商。

*高、低位分别指 left、right half.

* remainder（商）与 devidend（被除数）符号一致。

RISC-V Division

- **Four instructions:**
  - div, rem: signed divide, remainder
  - divu, remu: unsigned divide, remainder
- **Overflow and division-by-zero don't produce errors**
  - Just return defined results
  - Faster for the common case of no error

RISC-V 不会检查除数是否为0，需要软件解决。
→ 不会产生 error，而会返回一个 defined result.

浮点数：① sign ② significand ③ exponent

Floating point numbers

- **Form**
  - Arbitrary $363.4 \cdot 10^{34}$
  - Normalised $3.634 \cdot 10^{36}$
- **Binary notation**
  - Normalised $1.xxxxxx \cdot 2^{yyyyy}$
- **Standardised format IEEE 754**
  - Single precision 8 bit exp, 23 bit significand
  - Double precision 11 bit exp, 52 bit significand
- **Both formats are supported by RISC-V**

Single precision

31	30 23	22 0
S	exponent	fraction
1 bit	8 bits	23 bits

Double precision

31	30 20	19 0
S	exponent	fraction
1bit	11 bits	20 bits

31	fraction (continued)	0

浙江大学 Zhejiang University 系统结构与网络安全研究所

IEEE 754 standard

- **Leading '1' bit of significand is implicit**
  →saves one bit
- **Exponent is biased:**
  00...000 smallest exponent
  11...111 biggest exponent
  } exponent 一般不能是全0或全1.
  - Bias 127 for single precision
  - Bias 1023 for double precision
- Summary:

$$(-1)^{sign} \cdot (1 + significand) \cdot 2^{exponent - bias}$$

Example

- Show the binary representation of -0.75 in IEEE single precision format
- Decimal representation: $-0.75 = -3/4 = -3/2^2$
- Binary representation: $-0.11 = -1.1 \cdot 2^{-1}$
- Floating point
  - $(-1)^{sign} \cdot (1 + fraction) \cdot 2^{exponent - bias}$
  - $(-1)^{sign} = -1$, so Sign = 1
  - 1+ fraction = 1.1, so Significand=.1
  - exponent -127 =-1, so Exponent=(-1 +127) = 126

Single precision 32 位

31	30 23	22 0
1	0111 1110	100 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits

Double precision 64 位

31	30 20	19 0
1	011 1111 1110	1000 0000 0000 0000 0000
1bit	11 bits	20 bits

0000 0000 0000 0000 0000 0000 0000 0000

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx$ 6 decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx$ 16 decimal digits of precision

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0

Infinities and NaNs

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

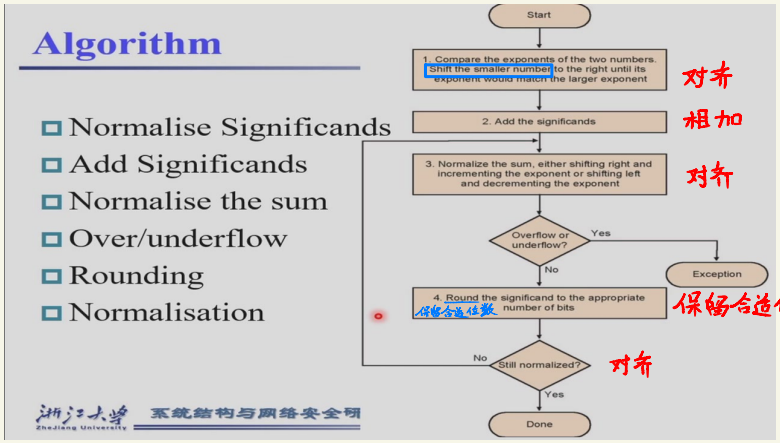
浮点数运算： $9.999 \cdot 10^1 + 1.610 \cdot 10^{-1}$

① Alignment（对齐）：小的往大的靠近 ：$9.999 \cdot 10^1 - 0.016 \cdot 10^1$

② Addition ：
$$\begin{array}{r} 9.999 \cdot 10^1 \\ 0.016 \cdot 10^1 \\ \hline 10.015 \cdot 10^1 \end{array}$$

③ Normalization：$1.0015 \cdot 10^2$

④ Rounding（四舍五入）$1.002 \cdot 10^2$



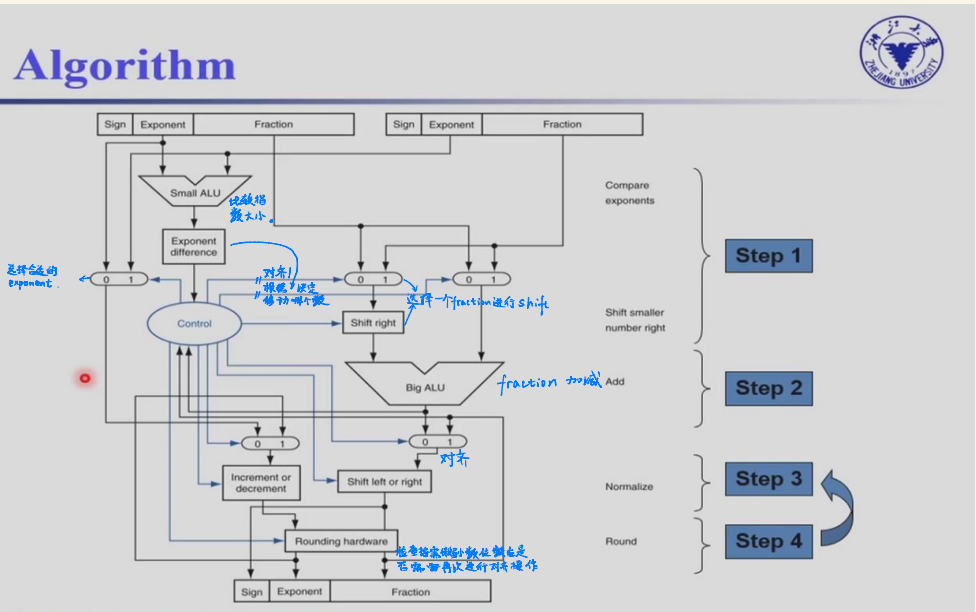
加法

Eg： $y = 0.5 + (-0.4375)$ 即 $1.000_2 \times 2^{-1}$ 与 $-1.110_2 \times 2^{-2}$

Step 1： $-1.110 \times 2^{-2} \rightarrow -0.111_2 \times 2^{-1}$

Step 2：
$$\begin{array}{r} 1.000 \times 2^{-1} \\ -0.111 \times 2^{-1} \\ \hline 0.001 \times 2^{-1} \end{array}$$

Step 3： 1.000×2^{-4}



乘法：$(S_1 \cdot 2^{e_1}) \cdot (S_2 \cdot 2^{e_2}) = (S_1 \cdot S_2) \cdot 2^{e_1+e_2}$

$e_1+e_2 > 127$（单精度）/ 1023（双精度），需相应减127/1023

↳因为实际指数分别是 (e_1-127)，(e_2-127)

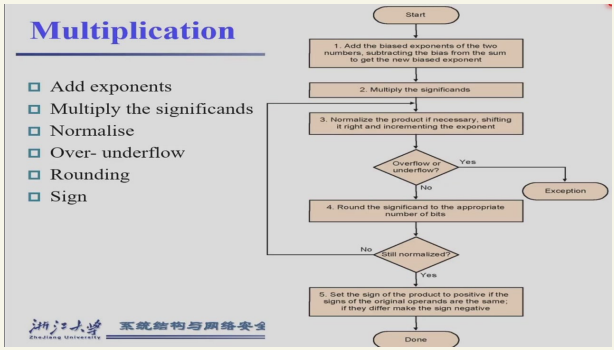
Eg：1 1000_0010(3) 00……0 $= -1 \times 2^3$

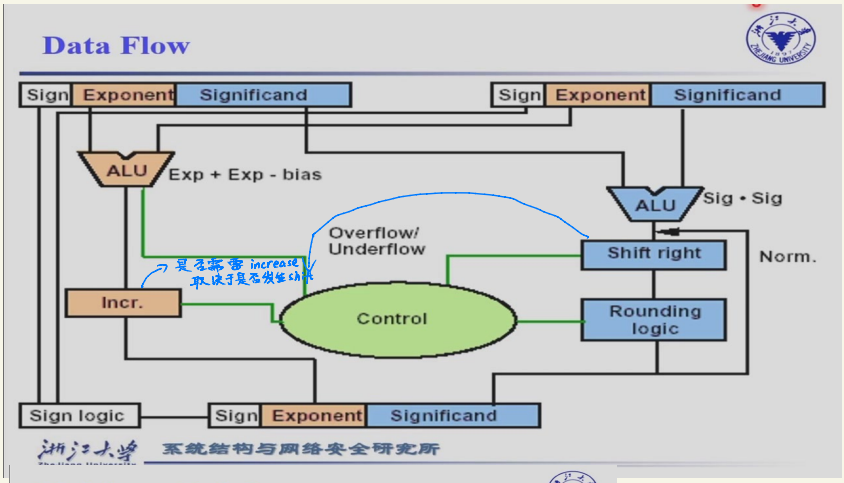
0 1000_0011(4) 00……0 $= 1 \times 2^4$

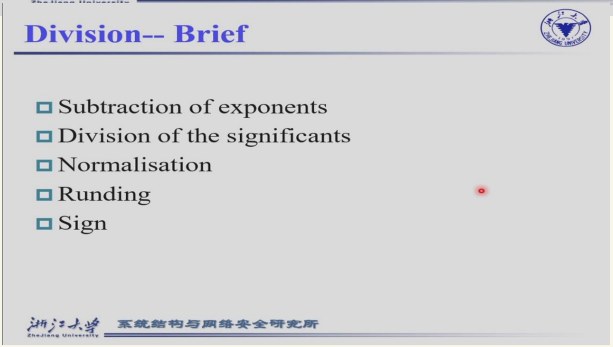
相乘：1 ~~1000_0101~~ 00000000

-0111_1111

= 1000 0110 (7)







IEEE Std 754 specifies

① 为了保留更高精度，允许在规定保留位数后额外再多保存不超过3位.
分别称为 guard、round、sticky 位.
(保留位)、(近似位)、(粘滞位)

Eg: $0.b_1b_2b_3\ 10\ 1$ （保留3位；10 分别为 g、r，1 为 s）

② 有 rounding models 可选择

ulp: units in last place: 最后一位的精度单位.

Eg: 2进制数: 1.11_{Δ} ulp = 0.25

10进制数: 2.36_{Δ} ulp = 0.01

使用 guard 和 round 的原因：保证最后结果误差不超过 $\frac{1}{2}$ ulp.

Eg:
$$\begin{array}{r} 2.34\,00 \\ (g\ r) \\ 0.0256 \\ \hline 2.3656 \end{array} \qquad \begin{array}{r} 2.34 \\ +0.02 \\ \hline 2.36 \end{array}$$

→ 误差为 0.1

↳ 与真实值 2.37 误差为 0.0044 < 0.005

只要 sticky bit 及其以后的位上出现 1，sticky bit 就取 1.

Chapter 2 Language of the Machine

4.1 Introduction.

Instruction : Operators + Operands

Two Key Simples :

① Instruction are represented as numbers.

② Programs can be stored in memory to be read/written just like numbers.

add a, b, c (a ← b+c)

substract a, b, c (a ← b−c)

Arithmetic instructions use register operands.

RISC-V has 32 64-bit registers.

They are marked as x0 to x31.

32-bit data is called a "word", 64-bit is a "doubleword".

RISC-V Registers

Name	Register name	Usage	Preserved On call?
x0	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7	5-7	Temporaries	no
x8-x9	8-9	Saved	yes
x10-x17	10-17	Arguments/results	no
x18-x27	18-27	Saved	yes
x28-x31	28-31	Temporaries	no

Memory is byte addressed. (Each address identifies 8 bits)

RISC-V is Little Endian.
Least-significant byte at least address of word.

第n+1条指令的address是第n条指令的address+4.

Eg: 一个十六进制数: 0x 12345678

显然存在内存里需要 4 bytes, 记为 0, 1, 2, 3.

显然, 应该"78"存在"0"地址的内存中, 将"12"存在"3"地址的内存中.

Word ALigned. (一个word 32位, 需要 4 address)

第1个 word 访问"0"地址内存, 第2个 word 访问"4"地址内存, …

RISC-V doesn't require word aligned. 但不好! 最好对齐!
比如强行取"1"地址的数据, 就会得到地址 1234 内的数据, 而地址123内的数据是第1个word的高位, 地址4内的数据是第2个word的低位。

Memory Alignment

Eg: 保证访问每一个变量时都只需访问1个 word.

```
Struct{
    int a;
    char b;
    char c[2];
    char d[3];
    int e;
}
```


Byte 1: 0x 01, Byte 0: 0x 02.

对大端来说: 数据是 0x 0201, 即 513
对小端来说: 数据是 0x 0102, 即 258

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add x5,x6,x7	x5=x6 + x7	Add two source register operands
	subtract	sub x5,x6,x7	x5=x6 - x7	First source register subtracts second one
	add immediate	addi x5,x6,20	x5=x6+20	Used to add constants
Data transfer	load doubleword	ld x5, 40(x6)	x5=Memory[x6+40]	doubleword from memory to register
	store doubleword	sd x5, 40(x6)	Memory[x6+40]=x5	doubleword from register to memory
	load word	lw x5, 40(x6)	x5=Memory[x6+40]	word from memory to register
	load word, unsigned	lwu x5, 40(x6)	x5=Memory[x6+40]	Unsigned word from memory to register
	store word	sw x5, 40(x6)	Memory[x6+40]=x5	word from register to memory
	load halfword	lh x5, 40(x6)	x5=Memory[x6+40]	Halfword from memory to register
Data transfer	load halfword, unsigned	lhu x5, 40(x6)	x5=Memory[x6+40]	Unsigned halfword from memory to register
	store halfword	sh x5, 40(x6)	Memory[x6+40]=x5	halfword from register to memory
	load byte	lb x5, 40(x6)	x5=Memory[x6+40]	byte from memory to register
	load word, unsigned	lbu x5, 40(x6)	x5=Memory[x6+40]	Unsigned byte from memory to register
	store byte	sb x5, 40(x6)	Memory[x6+40]=x5	byte from register to memory
	load reserved	lr.d x5,(x6)	x5=Memory[x6]	Load;1st half of atomic swap
	store conditional	sc.d x7,x5,(x6)	Memory[x6]=x5; x7 = 0/1	Store;2nd half of atomic swap
	Load upper immediate	lui x5,0x12345	x5=0x12345000	Loads 20-bits constant shifted left 12 bits

Memory Operand Example

- C code:
  - `A[12] = h + A[8];`
  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword
  - Offset: the constant in a data transfer instruction
  - Base register: the register added to form the address

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```


ld x9 64(x22)
sd x9 96(x22)

一个重要的 idea 是在 register 里放常用的数据.

Immediate Operands : addi x22, x22, 4 // x22 += 4

(Make the common case fast)

RISC-V operands

Name	Example	Comments
32 register	x0~x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory double words	Memory[0], Memory[8], ..., Memory[18446744073709551608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

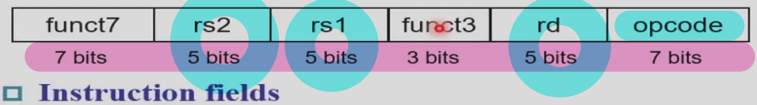
RISC-V fields (format)

Name	Fields						Comments
Field size	31 7bits 25	24 5bits 20	19 5bits 15	14 3bits 12	11 5bits 7	6 7bits 0	All RISC-V instruction 32 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	*imm[12,10:5]*	rs2	rs1	funct3	*imm[4:1,11]*	opcode	Conditional branch format
UJ-type	*immediate[20,10:1,11,19:12]*				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- **op:** *basic operation of the instruction, traditionally called the opcode.*
- **rd:** *destination register number.*
- **funct3:** *3-bit function code (additional opcode).*
- **rs1:** *the first register source operand.*
- **rs2:** *the second register source operand.*
- **funct7** *7-bit function code (additional opcode).*

RISC-V R-Format Instructions

所有操作数都是 register 类型



- **Instruction fields**
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)
- **Design Principle 3**
  - *Good design demands good compromises*
- **All instructions in RISC-V have the same length**
  - Conflict: same length ←----→ single instruction

为了方便，常用十六进制来表示机器码.

对 add. substract 两种类型。

(opcode 都是 51，减是让 funct7=32

add rd, rs1, rs2

add x9, x20, x21

⇒ machine code :

	0	21	20	0	9	51
	7 bits	5bits	5bits	3bits	5 bits	7 bits
⇒	0000000	10101	10100	000	01001	0110011

RISC-V I-Format Instructions

一个立即数、一个 register

immediate 是一个二进制补码，$(-2^{11} \sim 2^{11}-1)$

- **Immediate arithmetic and load instructions**
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended

对 addi 和 ld 两种类型

- ***Design Principle 3:* Good design demands good compromises**
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
- **Example: ld x9, 64(x22)**
  - 22 (x22) is placed rs1;
  - 64 is placed immediate
  - 9 (x9) is placed rd

ld rd, offset (rs1) :

rd = Memory[rs1 + offset]

RISC-V S-Format Instructions

immediate 被拆成两部分.

给 store 类型。

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

- Example: sd x9, 64(x22)
  - 22 (x22) is placed rs1;
  - 64 is placed immediate
  - 9 (x9) is placed rs2

sd rs2, offset(rs1):

Memory[rs1 + offset] = rs2

Shift Operations

用0补位，imm无符号

slli
srli

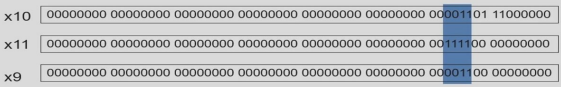
=最多移64位，因此imm需6位

- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by i bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and  x9,x10,x11
```

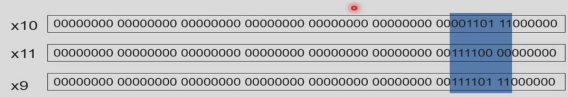


用于把某几位清零

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or  x9,x10,x11
```

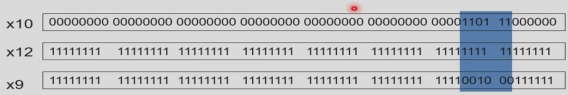


用于把某几位置 1

XOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

```
xor  x9,x10,x12   // NOT operation
```



用于 NOT 操作.
与全1进行 XOR.

R类操作

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory double words	Memory[0], Memory[8], …, Memory[18,446,744,073,709,551,608]]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential double word accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and x5, x6, 3	x5=x6 & 3	Arithmetic shift right by register
	inclusive or	or x5,x6,x7	x5=x6 | x7	Bit-by-bit OR
	exclusive or	xor x5,x6,x7	x5=x6 ^ x7	Bit-by-bit XOR
	and immediate	andi x5,x6,20	x5=x6 & 20	Bit-by-bit AND reg. with constant
	inclusive or immediate	ori x5,x6,20	x5=x6 | 20	Bit-by-bit OR reg. with constant
	exclusive or immediate	xori x5,x6,20	X5=x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	shift left logical	sll x5, x6, x7	x5=x6 << x7	Shift left by register
	shift right logical	srl x5, x6, x7	x5=x6 >> x7	Shift right by register
	shift right arithmetic	sra x5, x6, x7	x5=x6 >> x7	Arithmetic shift right by register
	shift left logical immediate	slli x5, x6, 3	x5=x6 << 3	Shift left by immediate

补符号位

Branch Instruction { beq r1, r2, L1 (若 $r_1 = r_2$, 跳转到L1)
bne r1, r2, L1 (若 r1 != r2, 跳转到L1)

分支结束需要一个绝对跳转: beq x0, x0, EXIT.
(EXIT 标识3跳出分支时程序的位置.

Supports while

Example 2.12 Compiling a *while* loop

(Assume: i ~ k---- x22 and x24 base of save ---- x25)

- C code:

```
while ( save[i] == k )
    i = + i ;
```

- RISCV assembly code:

```
Loop: slli  x10, x22, 3      # temp reg $t1 = 8 * i
      add   x10, x10, x25    # x10 = address of save[i]
      ld    x9, 0(x10)       # x9 gets save[i]
      bne   x9, x24, Exit    # go to Exit if save[i] != k
      addi  x22, x22, 1      # i += 1
      beq   x0, x0, Loop     # go to Loop
Exit:
```

需A[i]地址!

slt rd, rs1, rs2
若 rs1 < rs2, 则令 rd=1, 否则令 rd=0.

```
slt   x5, x8, x9
bne   x5, 0, Less
```

if x8 < x9, go to "Less".

Less:

More Conditional Operations

- **blt rs1, rs2, L1**
  - if (rs1 < rs2) branch to instruction labeled L1
- **bge rs1, rs2, L1**
  - if (rs1 >= rs2) branch to instruction labeled L1
- **Example**
  - if (a > b) a += 1;
  - a in x22, b in x23

```
bge  x23, x22, Exit      # branch if b >= a
addi x22, x22, 1
Exit:
```


Signed vs. Unsigned

- **Signed comparison: blt, bge**
- **Unsigned comparison: bltu, bgeu**
- **Example**
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23  # signed`
    - –1 < +1
  - `x22 > x23  # unsigned`
    - +4,294,967,295 > +1

Jump register & jump address table

- **Jump with register content**

jalr x1, 100(x6)

将下一条指令的地址存在x1内

- **jump address table**

x7←x6+4/8*K

这根斜杠不是除法，而是"或".

RISC-V assembly code:

```
Boundary  blt   x25, x0, Exit      # test if k < 0
          bge   x25, x5, Exit      # if k >= 4, go to Exit
          slli  x7, x25, 3         # temp reg x7 = 8 * k (0<=k<=3)
          add   x7, x7, x6         # x7 = address of JumpTable[k]
          ld    x7, 0(x7)          # temp reg x7 gets JumpTable[k]
          jalr  x1, 0(x7)          # jump based on register x7(entrance)
```

Exit:

jump address table

x7= x6 +8 * k:

L0:address

L1:address

L2: address

L3:address

Memory1

```
L0:  add   $s0,$s3, $s4   # k = 0 so f gets i + j
     jalr  x0, 0(x1)      # end of this case so go to Exit
L1:  add   $s0, $s1, $s2  # k = 1 so f gets g + h
     jalr  x0, 0(x1)      # end of this case so go to Exit
L2:  sub   $s0, $s1, $s2  # k = 2 so f gets g - h
     jalr  x0, 0(x1)      # end of this case so go to Exit
L3:  sub   $s0, $s3, $s4  # k = 3 so f gets i - j
     jalr  x0, 0(x1)      # end of switch statement
```

Memory2

跳转到程序原来的位置.

Basic Block：没有 Branch，不存在任何跳转的最小代码段。

调用子函数的 6 步：

① Place Parameters ② Transfer control

③ Storage Resources（得到 register 资源）

④ Perform ⑤ Place the result ⑥ Return Control

Caller

`jal x1, Procedure Address`

- 跳到新地址
- 相对地址
- jump and link
- 把 PC+4（下一条 PC 的地址）存在 x1

Callee

`jalr x0, 0(x1)`

- special register
- jump and link register
- x0 不会变，恒为 0

Using More Register

a0 ~ a7（x10 ~ x17）：eight argument registers

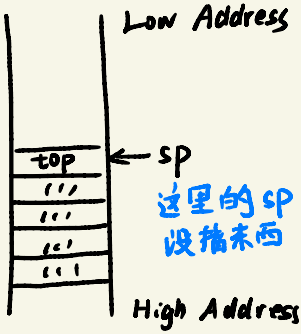
ra（x1）：return address

用于寄存函数所需参数。

对于 STACK 而言，top 是 lower address.

Push : addi sp, sp, -8
sd ..., 8(sp)

Pop : ld ..., 8(sp)
addi sp, sp, 8



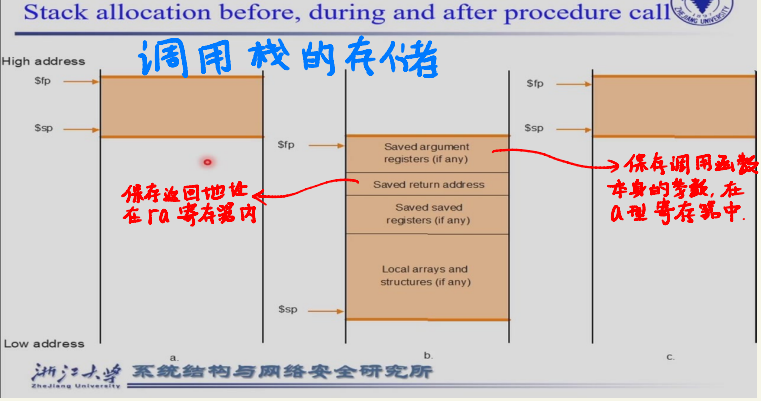
- Two classes of registers
  - t0 ~ t6: 7 temporary registers, by the callee not preserved
  - s0 ~ s11: 12 saved registers, must be preserved If used → 什指在调用者的栈

Name	Register no.	Usage	Preserved on call
x0(zero)	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7(t0-t2)	5-7	Temporaries	no
x8(s0/fp)	8	Saved/frame point	Yes
x9(s1)	9	Saved	Yes
x10-x17(a0-a7)	10-17	Arguments/results	no
x18-x27(s2-s11)	18-27	Saved	yes
x28-x31(t3-t6)	28-31	Temporaries	No
PC	-	Program counter	Yes

求 n!

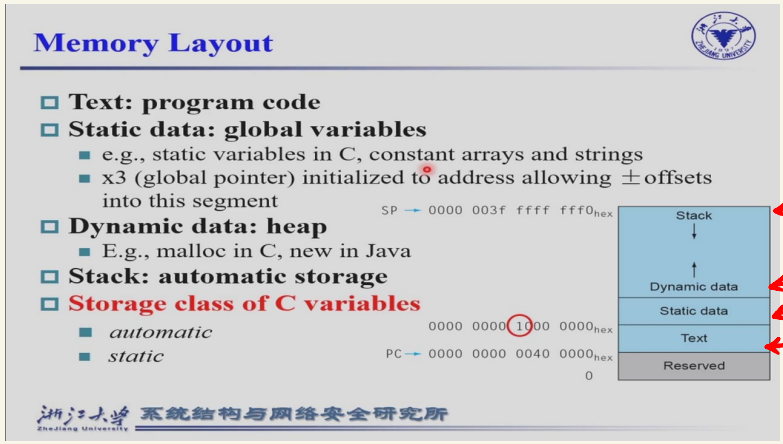
这里的 sp 指向存内容的最 top 地址

```
fact: addi  sp, sp, -16      # adjust stack for 2 items
      sd    ra, 8(sp)        # save the return address: x1
      sd    a0, 0(sp)        # save the argument n: x10
      addi  t0, a0, -1       # x5 = n - 1
      bge   t0, zero, L1     # if n >= 1, go to L1(else)
      addi  a0, zero, 1      # return 1 if n <1
      addi  sp, sp, 16       # Recover sp (Why not recover x1and x10 ?)
      jalr  zero, 0(ra)      # return to caller
L1:   addi  a0, a0, -1       # n >= 1: argument gets ( n - 1 )
      jal   ra, fact         # call fact with ( n - 1 )
      add   t1, a0, zero     #move result of fact(n - 1) to x6(t1)
      ld    a0, 0(sp)        # return from jal: restore argument n
      ld    ra, 8(sp)        # restore the return address
      add   sp, sp, 16       # adjust stack pointer to pop 2 items
      mul   a0, a0, t1       # return n*fact ( n - 1 )
      jalr  zero, 0(ra)      # return to the caller
```



sp 指向 top

fp (frame pointer) 不动，指向本 procedure 栈底 (第一个有效数据) → 起始地址. (high address 方便指针的计算)



静态栈，CPU分配

动态堆，malloc 分配

全局变量

codes

RISC-V assembly language

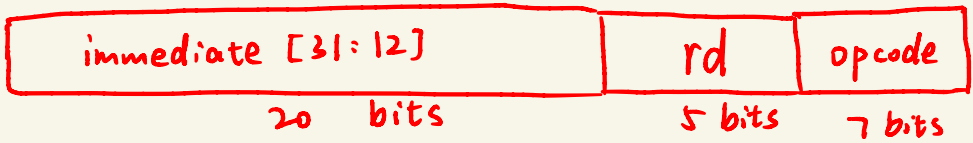
Category	Instruction	Example	Meaning	Comments
Logical	and	and x5, x6, 3	x5=x6 & 3	Arithmetic shift right by register
	inclusive or	or x5,x6,x7	x5=x6 | x7	Bit-by-bit OR
	exclusive or	xor x5,x6,x7	x5=x6 ^ x7	Bit-by-bit XOR
	and immediate	andi x5,x6,20	x5=x6 & 20	Bit-by-bit AND reg. with constant
	inclusive or immediate	ori x5,x6,20	x5=x6 | 20	Bit-by-bit OR reg. with constant
	exclusive or immediate	xori x5,x6,20	X5=x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	shift left logical	sll x5, x6, x7	x5=x6 << x7	Shift left by register
	shift right logical	srl x5, x6, x7	x5=x6 >> x7	Shift right by register
	shift right arithmetic	sra x5, x6, x7	x5=x6 >> x7	Arithmetic shift right by register
	shift left logical immediate	slli x5, x6, 3	x5=x6 << 3	Shift left by immediate
Shift	shift right logical immediate	srli x5,x6,3	x5=x6 >> 3	Shift right by immediate
	shift right arithmetic immediate	srai x5,x6,3	x5=x6 >> 3	Arithmetic shift right by immediate
Conditional branch	branch if equal	beq x5, x6, 100	if(x5 == x6) go to PC+100	PC-relative branch if registers equal
	branch if not equal	bne x5, x6, 100	if(x5 != x6) go to PC+100	PC-relative branch if registers not equal
	branch if less than	blt x5, x6, 100	if(x5 < x6) go to PC+100	PC-relative branch if registers less
	branch if greater or equal	bge x5, x6, 100	if(x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	branch if less, unsigned	bltu x5, x6, 100	if(x5 >= x6) go to PC+100	PC-relative branch if registers less, unsigned
	branch if greater or equal, unsigned	bgeu x5, x6, 100	if(x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	jump and link	jal x1, 100	x1 = PC + 4; go to PC+100	PC-relative procedure call
	jump and link register	jalr x1, 100(x5)	x1 = PC + 4; go to x5+100	procedure return; indirect call

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add x5,x6,x7	x5=x6 + x7	Add two source register operands
	subtract	sub x5,x6,x7	x5=x6 - x7	First source register subtracts second one
	add immediate	addi x5,x6,20	x5=x6+20	Used to add constants
Data transfer	load doubleword	ld x5, 40(x6)	x5=Memory[x6+40]	doubleword from memory to register
	store doubleword	sd x5, 40(x6)	Memory[x6+40]=x5	doubleword from register to memory
	load word	lw x5, 40(x6)	x5=Memory[x6+40]	word from memory to register
	load word, unsigned	lwu x5, 40(x6)	x5=Memory[x6+40]	Unsigned word from memory to register
	store word	sw x5, 40(x6)	Memory[x6+40]=x5	word from register to memory
	load halfword	lh x5, 40(x6)	x5=Memory[x6+40]	Halfword from memory to register
Data transfer	load halfword, unsigned	lhu x5, 40(x6)	x5=Memory[x6+40]	Unsigned halfword from memory to register
	store halfword	sh x5, 40(x6)	Memory[x6+40]=x5	halfword from register to memory
	load byte	lb x5, 40(x6)	x5=Memory[x6+40]	byte from memory to register
	load word, unsigned	lbu x5, 40(x6)	x5=Memory[x6+40]	Unsigned byte from memory to register
	store byte	sb x5, 40(x6)	Memory[x6+40]=x5	byte from register to memory
	load reserved	lr.d x5,(x6)	x5=Memory[x6]	Load;1st half of atomic swap
	store conditional	sc.d x7,x5,(x6)	Memory[x6]=x5; x7 = 0/1	Store;2nd half of atomic swap
	Load upper immediate	lui x5,0x12345	x5=0x12345000	Loads 20-bits constant shifted left 12 bits

ZheJiang University

lui : U-TYPE



lui rd, immediate

把这个立即数填到 rd 的高 20 位，低位取 0.

因此把一个寄存器的值赋成一个 32 位的数，

需要先用 lui 把高 20 位放入寄存器，再用 addi 加上后 12 位.

Branch Addressing

- **Addressing in branches**
  - Branch instructions specify
    - Opcode, two registers, target address
  - Most branch targets are near branch
    - Forward or backward
- **SB-type: bne x10, x11, 2000, //2000 = 0111 1101 0000**

inst[31:0]

0	111110	01011	01010	001	1000	0	1100011
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

imm[31:0]

Inst[31] signextension	Inst[7]	Inst[30:25]	Inst[11:8]	1'B0

  - PC-relative addressing

Target address = PC + Branch offset
= PC + immediate × 2

不存 immediate 的第0位：默认 half-word 对齐，offset是偶数

Jump Addressing

- **Jump and link (jal)** → 依然受范围限制
  - target uses 20-bit immediate for larger range
- **UJ format:**

31 1bit	10bits	1bit	8bits 12	11 7	6 0
inn[20]	*imm[10:1]*	*imm[11]*	*imm[19:12]*	rd	Opcode
	20bit			5bit	7bit

- **jal x0, 2000** # 2000_{10} =(0 *00000000* 0 *111 1101 000*0)$_2$

31 1bit	10bits	1bit	8bits 12	11 7	6 0
0	*1111101000*	0	*00000000*	00000	1101111
	20bit			5bit	7bit

- **For more long jumps:** eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

可以跳得更远

beq a0, zero, L1 超范围？

⇒ bne a0, zero, L2

jal zero, L1

L2：

Summary of RISC-V architecture in Ch. 2

RISC-V Instruction Format and Their Operands

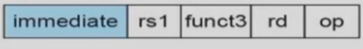
Name	Fields						Comments
Field size	7bits	5bits	5bits	3bits	5bits	7bits	All RISC-V instruction 32 b
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	*imm[12,10:5]*	rs2	rs1	funct3	*imm[4:1,11]*	opcode	Conditional branch format
UJ-type	*immediate[20,10:1,11,19:12]*				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Name	Operands
32 registers	$zero, ra, sp, gp, tp, t0-t6, s0~s11, a0~a7
Mem words	Memory[0], Memory[8], Memory[10], . . . , Memory[18,446,744,073,709,551,608]

x0(zero)	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7(t0-t2)	5-7	Temporaries	no
x8(s0/fp)	8	Saved/frame point	Yes
x9(s1)	9	Saved	Yes
x10-x17(a0-a7)	10-17	Arguments/results	no
x18-x27(s2-s11)	18-27	Saved	yes
x28-x31(t3-t6)	28-31	Temporaries	No
PC	-	Auipc(Add Upper Immediate to PC)	

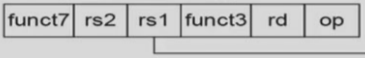
RISC-V Addressing Summary

1. Immediate addressing

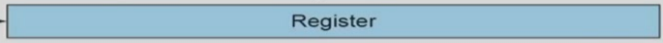


addi x5,x6,4

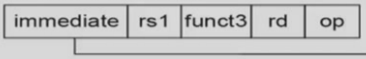
2. Register addressing



add x5,x6,x7

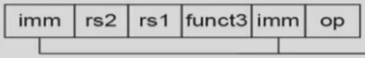


3. Base addressing



ld x5,100(x6)

4. PC-relative addressing



beq x5,x6,L1

read / write 做原子操作，无法被影响

atomatic operation 原子操作：

lr.d rd,(rs1)

sc.d rd,(rs1),rs2

Synchronization in RISC-V

- **Load reserved:** `lr.d rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- **Store conditional:** `sc.d rd,(rs1),rs2`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

Synchronization in RISC-V

- **Example 1: atomic swap (to test/set lock variable)**

```
again:  lr.d x10,(x20)
        sc.d x11,(x20),x23 // X11 = status
        bne  x11,x0,again  // branch if store failed
        addi x23,x10,0     // X23 = loaded value
```

- **Example 2: lock**

```
        addi x12,x0,1          // copy locked value
again:  lr.d x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.d x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

- **Unlock:**

```
        sd   x0,0(x20)         // free lock
```

把C代码转成汇编：
①给变量赋寄存器 ②翻译程序 ③还原寄存器

寄存器压栈	addi ld	sp, sp, -8 ~, 8(sp)
函数实现	------	
寄存器出栈	sd addi	~, 8(sp) sp, sp, 8

mv r2, r1 将r1中的值移动到r2中

精简指令集（RISC）：MIPS、RISC-V 指令长度都是32位
复杂指令集（CISC）：x86 指令的长度不同，因此操作起来很麻烦，如取址

Reduced / Complex Instruction Set Computer.

对于RISC：一条指令翻译为一条微指令。
对于CISC：一条指令翻译为多条微指令。
这些微指令类似，因此现在的CISC通过compiler也可以避免复杂指令，使CISC的性能与RISC类似

RV64I：向下兼容32位
RV32I：全是32位（寄存器、操作码）
↑
Integer，扩展的一种，是最基础的一种
其它扩展还有M（支持

Instruction Set Extensions

- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

Chapter 4 The Processor

4.1 Introduction

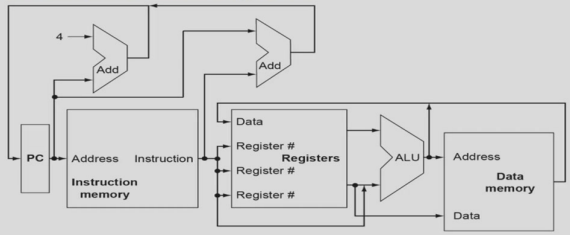
First two steps of instructions:

① 从 memory 中获得 instruction

② Decode and Read registers.

An overview of Implementation

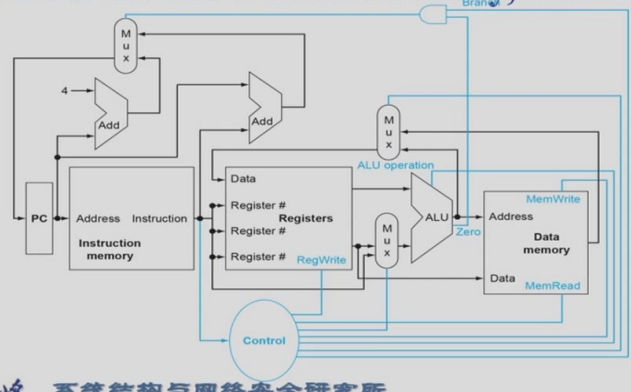
- Send PC to the code memory.
- Read one or two register.



浙江大学 Zhejiang University 系统结构与网络安全研究所

Control

- Different Sources for unit.
- Control the units: Read Memory, Write Memory.



浙江大学 Zhejiang University 系统结构与网络安全研究所

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Name	Register no.	Usage	Preserved on call
x0(zero)	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7(t0-t2)	5-7	Temporaries	no
x8(s0/fp)	8	Saved/frame point	Yes
x9(s1)	9	Saved	Yes
x10-x17(a0-a7)	10-17	Arguments/results	no
x18-x27(s2-s11)	18-27	Saved	yes
x28-x31(t3-t6)	28-31	Temporaries	No
PC	-	Auipc(Add Upper Immediate to PC)	

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

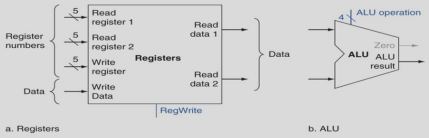
Instruction execution in RISC-V

- **Fetch :**
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- **Instruction decoding & Read Operand:**
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use
  - Reading Register Operands, whether or not to use
- **Executive Control:**
  - Control the implementation of the corresponding ALU operation
- **Memory access:**
  - Write or Read data from memory
  - Only ld/sd
- **Write results to register:**
  - If it is R-type instructions, ALU results are written to rd
  - If it is I-type instructions, memory data are written to rd
- **Modify PC** for branch instructions

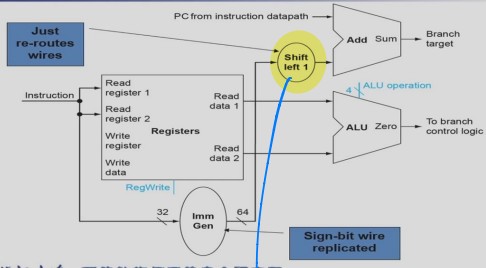
1. Fetch

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

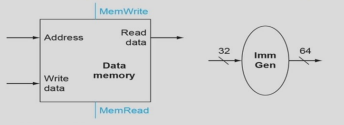


Branch Instructions



Load/Store Instructions

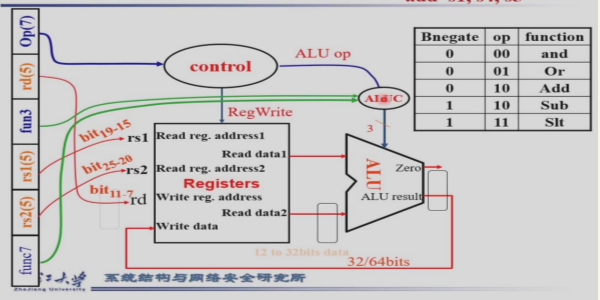
- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



指的是BR操作类，
指令中的12位首先补成
31位，然后末位置0，成32位。
但最终要变成64位，
所以可以先变成64位后左移1位，
相当于末位置0.

R type Instruction & Data stream

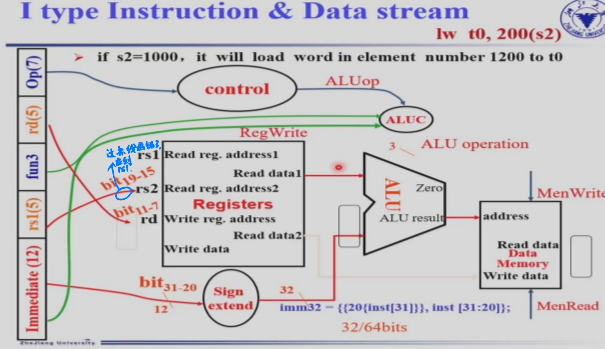
add s1, s4, s5



Bnegate	op	function
0	00	and
0	01	Or
0	10	Add
1	10	Sub
1	11	Slt

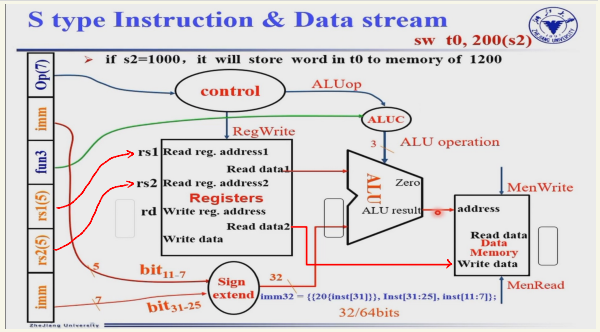
I type Instruction & Data stream

lw t0, 200(s2)



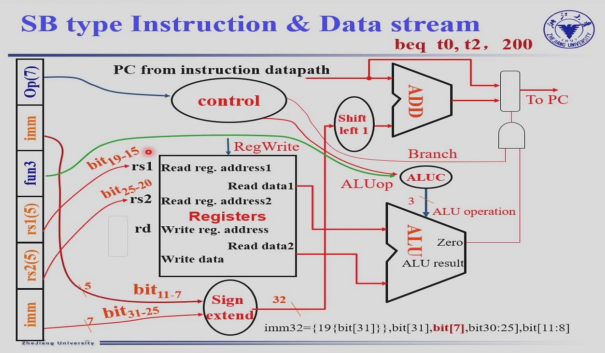
S type Instruction & Data stream

sw t0, 200(s2)



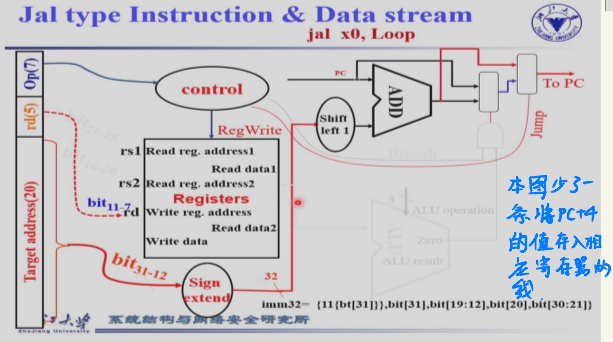
SB type Instruction & Data stream

beq t0, t2， 200



Jal type Instruction & Data stream

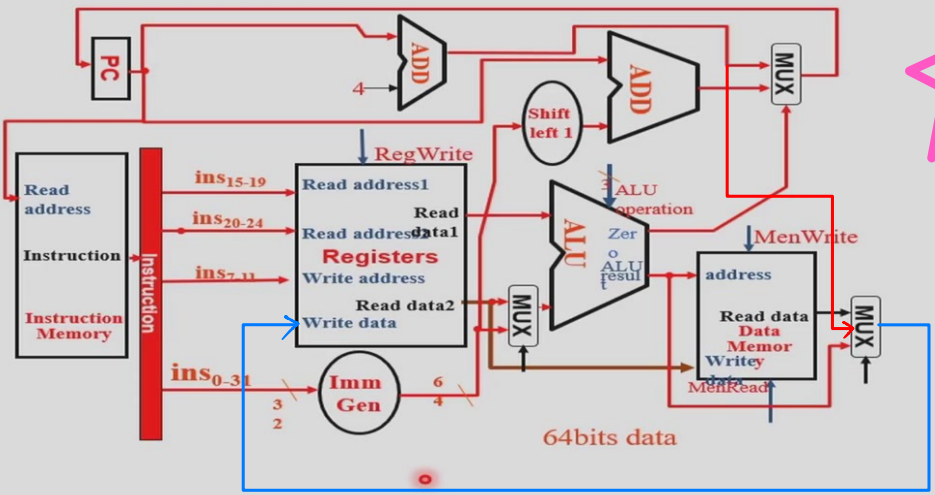
jal x0, Loop

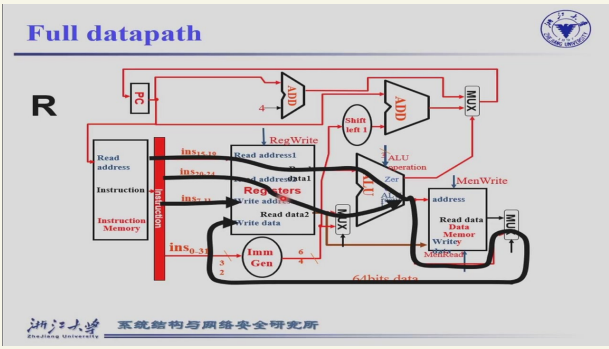


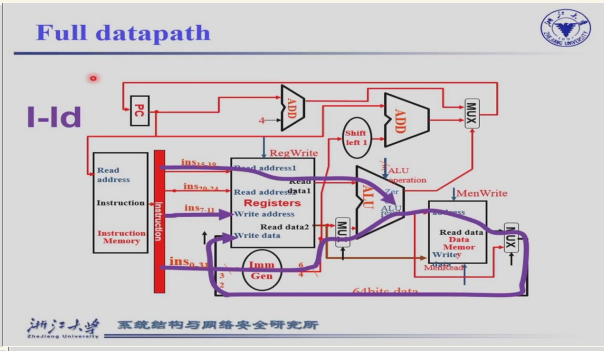
本图少了一条将PC+4的值存入相应寄存器的线

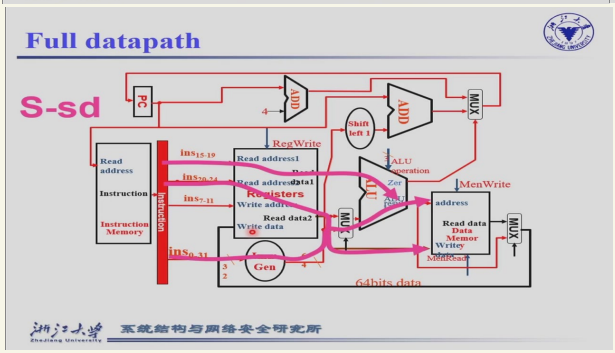
上述五合一：

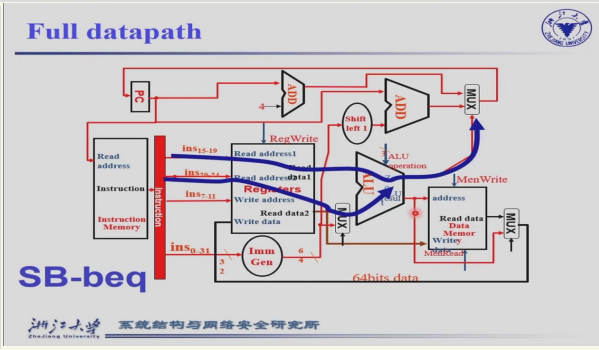
Full datapath

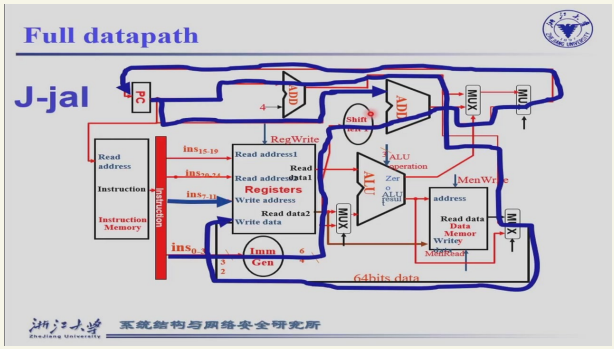




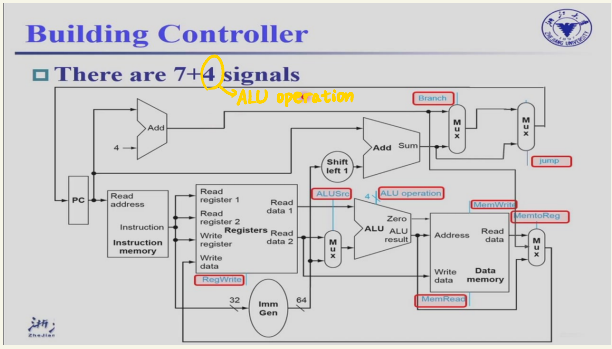








接下来设计 Controller



Signals for datapath　　Defined 7 control signals

Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
RegWrite	None	Register destination input is written with the value on the Write data input
ALUScr	The second ALU operand come from the second register file output (Read data 2)	The second ALU operand is the sign-extended lower 16 bits of the instruction..
Branch (PCSrc)	The PC is replaced by the output of the adder that computers the value PC+4	The PC is replaced by the output of the adder that computers the branch target.
Jump	The PC is replaced by PC+4 or branch target	The PC is updated by jump address computed by adder
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by value on the Write data input.
MemtoReg (2位)	00：The value fed to register Write data input comes from the Alu	01：The value fed to the register Write data input comes from the data memory.
		10：The value fed to the register Write data input comes from PC+4

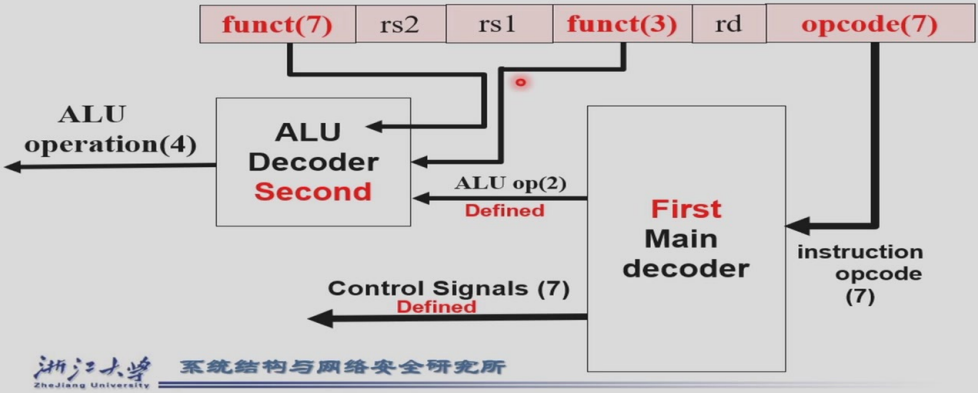
这些 control 信号(7个)在 opcode 决定时就已决定

(opcode 仅与指令类型有关)

而 ALU 的控制信号需要进一步确定！

Scheme of Controller

- 2-level decoder

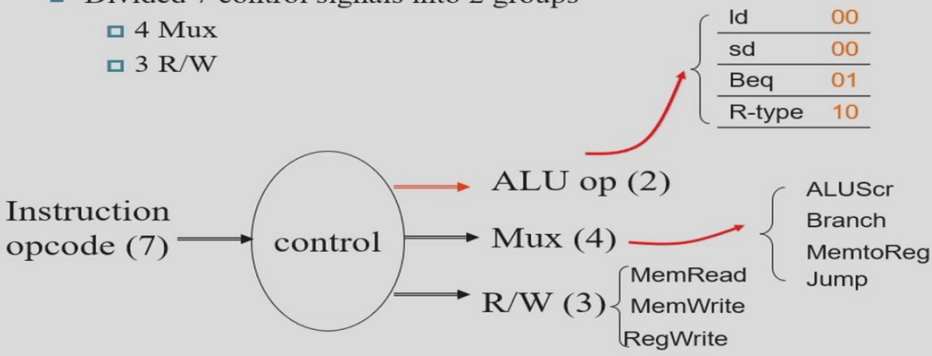


输入		输出								
Instruction	OPCode	ALUSrcB	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
R-format	0110011	0	00	1	0	0	0	0	1	0
Ld(I-Type)	0000011	1	01	1	1	0	0	0	0	0
sd(S-Type)	0100011	1	X	0	0	1	0	0	0	0
beq(SB-Type)	1100111	0	X	0	0	0	1	0	0	1
Jal(UJ-Type)	1101111	X	10	1	0	0	0	1	X	X

Designing the Main Control Unit

First level

- **Main Control Unit function**
  - ALU op (2)
  - Divided 7 control signals into 2 groups
    - 4 Mux
    - 3 R/W



指令译码器参考描述

```
`define CPU_ctrl_signals {ALUSrc_B,MemtoReg,RegWR,MemWrite,Branch,Jump,ALUop}
        always @* begin
                case(OPcode)
                5'b01100: begin CPU_ctrl_signals = ?; end          //ALU
                5'b00000: begin CPU_ctrl_signals = ?;  end         //load
                5'b01000: begin CPU_ctrl_signals = ?;  end         //store
                5'b11000: begin CPU_ctrl_signals = ?;  end         //beq
                5'b11011: begin CPU_ctrl_signals = ?;  end         //jump
                5'b00100: begin CPU_ctrl_signals = ?;  end         //ALU(addi;;;;)
                                        ......
                default:        begin  CPU_ctrl_signals = ?;  end
                endcase
        end
```

Design the ALU Decoder second level

- **ALU operation is decided by 2-bit ALUOp derived from opcode, and funct7 & funct3 fields of the instruction**
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	xxx	add	0010
sd	00	store register	XXXXXXX	xxx	add	0010
beq	01	branch on equal	XXXXXXX	xxx	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001
		SLT	0000000	010	Slt	0111

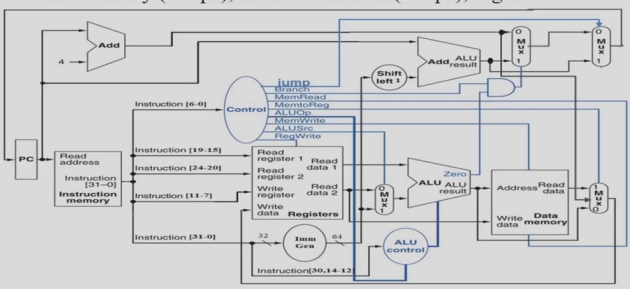
ALU Controller Code

- **ALU Control HDL Description**

```
assign Fun = {Fun3,Fun7[5]};
always @* begin
    case(ALUop)
    2'b00: ALU_Control = ? ;                          //add计算地址
    2'b01: ALU_Control = ? ;                          //sub比较条件
    2'b10:
        case(Fun)
            4'b0000: ALU_Control = 3'b010 ;    //add
            4'b0001: ALU_Control = ? ;         //sub
            4'b1110: ALU_Control = ? ;         //and
            4'b1100: ALU_Control = ? ;         //or
            4'b0100: ALU_Control = ? ;         //slt
            4'b1010: ALU_Control = ? ;         //srl
            4'b1000: ALU_Control = ? ;         //xor
                    ......
            default:  ALU_Control=3'bx;
            endcase
            ..................................................
    endcase
```

4.5 An overview of pipelining

- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (200ps), register file access (100ps)



add 指令 600ps : ①②③④

ld 指令 800ps : ①②③④⑤

sd 指令 700ps : ①②③④

beq 指令 500ps : ①②③

因此 clock 周期至少 800ps.

clock 周期会受最长延时的限制 ⇒ 流水线!

RISC-V Pipeline

- **Five stages, one step per stage**
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipelining RISC-V instruction set

- **Since there are five separate stages, we can have a pipeline in which one instruction is in each stage.**
- **CPI is decreased to 1, since one instruction will be issued (or finished) each cycle.**
- **During any cycle, one instruction is present in each stage.**

	Clock Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

- **Ideally, performance is increased five fold !**

时钟周期被压缩为 200ps !

Pipeline Speedup

- **If all stages are balanced**
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = Time between instructions$_{nonpipelined}$ /Number of stages
- **If not balanced, speedup is less**
- **Speedup due to increased throughput**
  - Latency (time for each instruction) does not decrease

RISC-V 的指令集很适合流水线

但流水线的任意一环出问题，其余都会受影响

Hazards 竞争！

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards 结构竞争
  - A required resource is busy
- Data hazard 数据竞争
  - Need to wait for previous instruction to complete its data read/write
- Control hazard 控制竞争
  - Deciding on control action depends on previous instruction

Structure Hazards

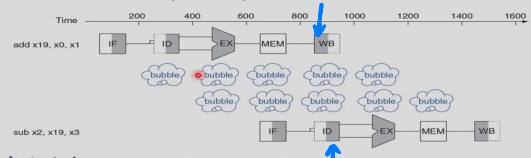
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

- An instruction depends on completion of data access by a previous instruction

```
add  x19, x0, x1
sub  x2, x19, x3
```

上升沿 write



下降沿 read

double bump

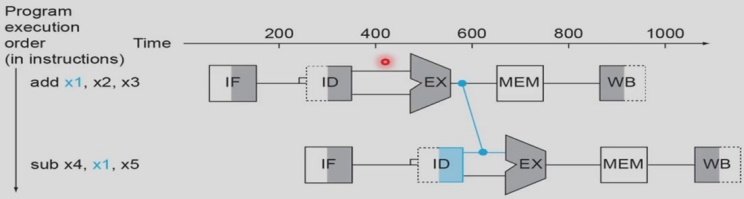
添加 bubble，避免 数据竞争

前半时钟：上升沿

后半时钟：下降沿

Forwarding (aka Bypassing)

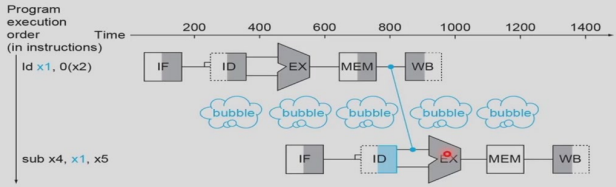
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



提前把计算结果给下一次，避免3 Bubble.

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



load 指令必须插一个 bubble，但也不需要等第一条指令完全结束.

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for a = b + e; c = b + f;

```
ld   x1, 0(x0)          ld   x1, 0(x0)
ld   x2, 8(x0)          ld   x2, 8(x0)
add  x3, x1, x2         ld   x4, 16(x0)
sd   x3, 24(x0)         add  x3, x1, x2
ld   x4, 16(x0)         sd   x3, 24(x0)
add  x5, x1, x4         add  x5, x1, x4
sd   x5, 32(x0)         sd   x5, 32(x0)
```

stall, stall

13 cycles | 11 cycles

编译器通过调整顺序来减少时钟周期

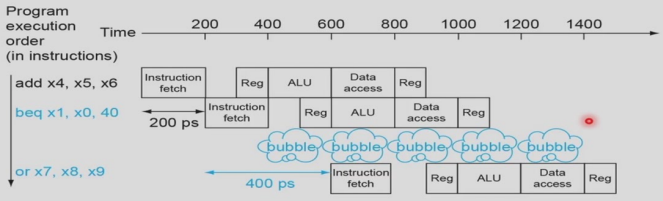
Control Hazards

- **Branch determines flow of control**
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- **In RISC-V pipeline**
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

尽管放到ID

Stall on Branch

- **Wait until branch outcome determined before fetching next instruction**



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

浙江大学 Zhejiang University 系统结构与网络安全研究所

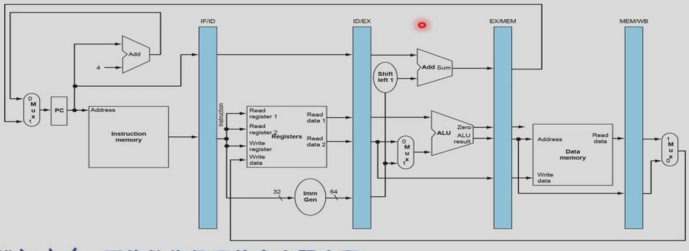
预测：Branch 是否跳转，
若预测错误则另作打算。

流水线 Pipeline：五个阶段：IF、ID、EX、MEM、WB
增加吞吐量 throughput，不改变延迟 latency.
但是面临 structure / data / control hazard 竞争问题。

不同阶段间插入寄存器，储存上条指令的结果

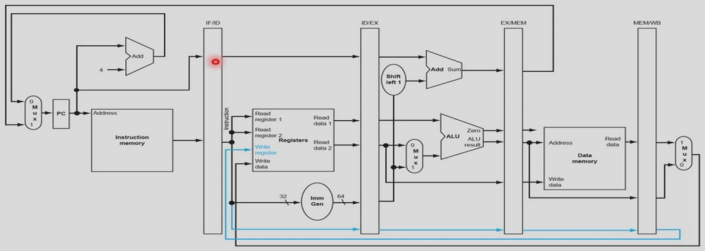
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle



浙江大学 Zhejiang University 系统结构与网络安全研究所

Corrected Datapath for Load



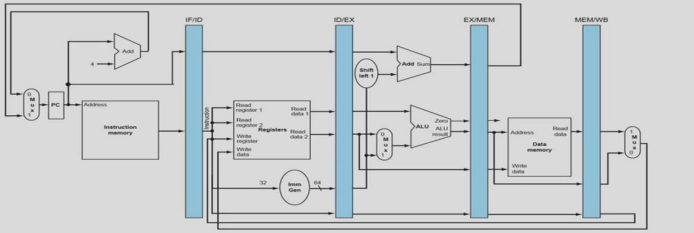
注意这条蓝色的线！

Load的 Write_Address 不是直接从 IF/ID 寄存器中来！此时的 IF/ID中存的是其它指令。

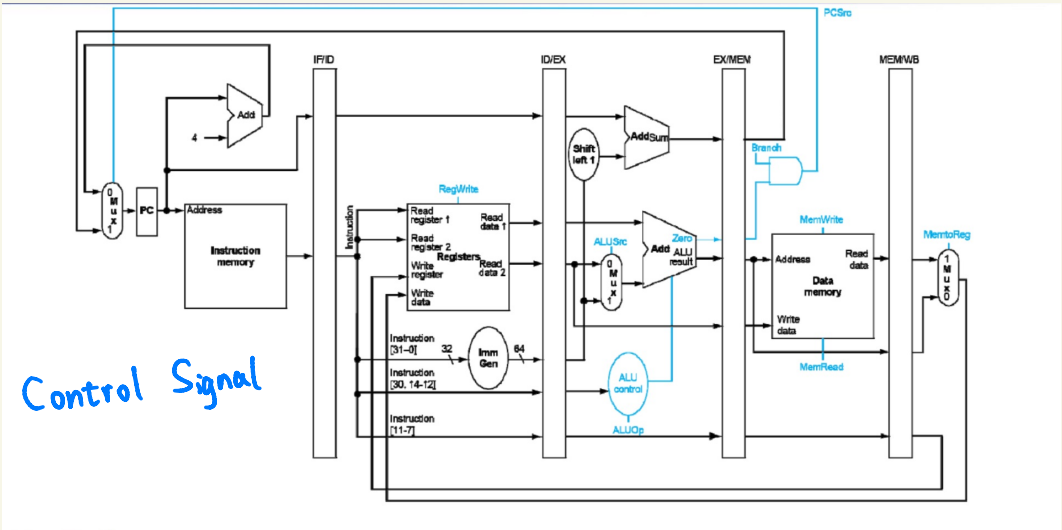
Single-Cycle Pipeline Diagram

State of pipeline in a given cycle

add x14, x5, x6	ld x13, 48(x1)	add x12, x3, x4	sub x11, x2, x3	ld x10, 40(x1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



实际图.



Control Signal