Midterm Notes

1. Details on PPT

Classes of computers:
- personal computers: general purpose, cost/performance tradeoff
- server computers: network based, high capacity/performance
- supercomputers: highest capability
- embedded computers: stringent power/cost constraints

Common number system: binary, decimal, hexidecimal

Below programs:
- Application software: HLL
- System software: Compiler, Operating System
- ⇕ ISA
- Hardware

底层 → 硬件能理解，执行的指令的集合（让system后和hardware后都符合这个规定）

Level of program code:
- HLL
- Assembly
- Hardware representation: binary code

HLL —compiler→ assembly —assembler→ machine code

Components of the computer:
- Input (data transferred to memory)
- Output (data read from memory)
- Memory
- Processor:
  - datapath: 用于数据处理和传输的硬件 (register, I/O, …)
  - control: 负责协调管理数据的操作 (clock signal, instruction decoding, …)

ISA:
- native data types
- instructions
- registers
- addressing modes
- memory architecture
- interpret and exception handling
- external I/O

包括

→ 一种叫 RISC 的，MIPS 是 RISC（较简便，轻量级的）

当程序运行时，步骤：
1. fetch the instruction
2. decode the instruction
3. execute the instruction
4. write back

2. Digital Logic

- digital waveforms: discrete (e.g.: > certain threshold → 1)
- analog waveforms: continuous

logic:
- truth table
- logical expression
- Graphic form (K-map)

Universal Gate（单一 Gate 可搭建所有电路）：NAND 和 NOR

Gate: NOT, AND, OR, NAND, NOR, XOR

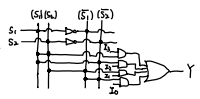
- combinational logic circuits: no memory ⇒ output can be expressed as logic functions of input
- sequential logic circuits: with memory! cannot

Combinational logic

① Multiplexer (selector)

2^n to 1 multiplexer:
- 2^n input
- n selection input (which to choose) → 这 n 位 input 对应的 binary number 就是选的 input（n selected lines）
- 1 output

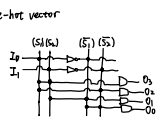
相当于先在前面的电路里用 not gate 把 n 位 selection input 转成 2^n 种组合，然后每种组合和一个 input 一起进 and gate，也就是必须满足特定的组合，input 才能过去



② Decoder (n to 2^n decoder)

n to 2^n decoder:
- n input
- 2^n output (每个 input 选择 I_n 还是 not I_n，每种组合对应一个 output)

one-hot vector



③ minterm and maxterm

canonical form:
- SOP (min): 所有结果为 1 的 input 乘起来之后的和
- POS (max): 所有结果为 0 的 input 加起来的积

④ two-level logic and PLA

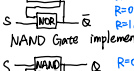
⑤ K-map

要注意的点：
- 表中 entry 都是 minterm
- 简化方法可能不唯一
- 最上面一行和最下面一行/最左边一行和最右边一行也是 adjacent 的
- 相邻的 2^n 个才能被圈

Sequential logic

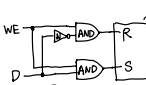
① S-R Latch

NOR Gate implementation:



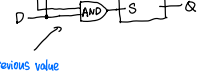
R=1, S=1 ⇒ forbidden state → 1 — 0 ⇒ 0 — 0 or 1 (X)
R=0, S=0 ⇒ Latch
R=0, S=1 ⇒ set
R=1, S=0 ⇒ Reset

NAND Gate implementation:



R=0, S=0 ⇒ forbidden state → 0 — 1 ⇒ 1 — 0 or 0 (X)
R=1, S=1 ⇒ Latch
R=0, S=1 ⇒ Reset
R=1, S=0 ⇒ Set

② D-Latch



WE=1: Q=D
WE=0: Q = previous value

③ Register

a collection of D-latches, controlled by a common WE

④ clock signal

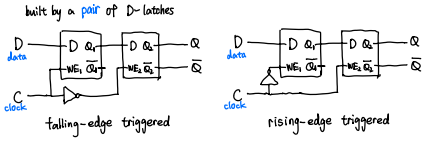
a shared signal that controls all the circuit element about time

clock speed/clock frequency × cycle time = 1

edge-triggering terminology (state change only occur in edge time)
↳ flip-flops (latch can do state change any time)

⑤ Master-Slave D Flip-flop

built by a pair of D-latches



3. Data representation

2's complement

① signed integer

$(6)_{10} = (0110)_2$ —flip→ 1001 —+1→ 1010

$(-6)_{10} = (1010)_2$ —flip→ 0101 —+1→ 0110

∴ 对 k-bit 来说，能表示的是 $[-2^{k-1}, 2^{k-1}-1]$

Overflow & Underflow

(unsigned 就 $0 \sim 2^k-1$，更 trivial，略。)

- signed extension: for signed ints
- zero extension: for unsigned ints

(more in MIPS)

注意：假如要把 -9 转成 8 位的 2's complement

在 $(9)_{10} = (1001)_2$ 之后记得先把 $(1001)_2$ extend 成 $(00001001)_2$ 再转

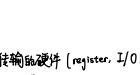
注意因为这个源码，由于是正数所以用 0-extension

$(00001001)_2 \xrightarrow{-} (11110110)_2 \xrightarrow{+1} (11110111)_2 = (-9)_{10}$

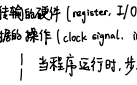
② floating point numbers

finite range, limited precision

(能表示的数不是 equally distributed 的，前窄后疏)



(32 bits, single precision) 7位精度



(64 bits, double precision) 16位精度

算法：(significand_bits+1) × $\log_{10}2$

IEEE 754 Standard

bias 目的：更容易做 comparison（直接 bit-by-bit 比就行）

1° normalized case

拿到一个数，先化为 $2^?+2^?+\cdots$ 的形式
- 整数：直接拆
- 小数：整数和小数分别拆，再合起来
  - 小数化二进制可用 "2 乘整数部分" 法

然后把二进制表示写出来，化为 normalized scientific representation

然后 significand 就写出来

exponent 是原本的 exp + bias 之后转为二进制源码的值

最后加上 sign bit，0 for 正，1 for 负

注意：exponent 在这个 case 里不能 = 000…0 or 111…1

假设是 32 位的，那么：

representable range:
- -max: $(-1)^1 \times 1.1\cdots1 \times 2^{(11\cdots10)_2-127} = 1.1\cdots1 \times 2^{127} \simeq -2^{128}$
- -min: $(-1)^1 \times 1.00\cdots0 \times 2^{(00\cdots01)_2-127} = -2^{-126}$
- +min: $(-1)^1 \times 1.00\cdots0 \times 2^{(00\cdots01)_2-127} = 2^{-126}$
- +max: $(-1)^0 \times 1.11\cdots1 \times 2^{(11\cdots10)_2-127} = 1.1\cdots1 \times 2^{127} \simeq 2^{128}$

underflow & overflow (都是 exponent 意义上的，significand 控制的是多足精度)

underflow: $-2^{-126} \sim 2^{-126}$

overflow: $-\infty \sim -2^{128}$ & $2^{128} \sim +\infty$

说得 significand 有点绕认知！！！

说明不能表示某个 range 里所有整数 (比如 half precision 才能表示 16-bit int)

$2^{14}+2^0 = 2^{14}(1+2^{-14})$ → significand 又有 10 位

2° de-normalized case

exponent = 00…0 = the exponent of $(00\cdots01)_2 = |-127| = -126$

significand 直接读，但没有 default 1

比如 0 后面是 0 000…0 01100…0

value $= (-1)^0 \times 2^{-126} \times (0.01100\cdots0)_2$

if exponent 和 significand 全是 0，那么就表示 0

3° Special case

exponent = 111…1
- significand 是全 0：infinite number，正负看 sign bit
- significand 不是全 0：NaN

一般习惯用 hexidecimal 的时候在前面加 "0x"

③ characters ⇒ unsigned bytes!

usually 1 byte (8-bit), follow ASCII standard

4. MIPS

Doing abstraction makes life easier!

Ideally for each ISA there is an assembly language (satisfying/following the ISA standard)

但其实对于一个 ISA 来说，有点小种 AL，在在细微差异，也挺常见

↳ Assembler 把它转为 machine code

Basic Knowledge

① Register: inside processor，处理速度 (transfer 速度) 比内存快很多

in MIPS: 32 registers (number: tradeoff)

data 必须从 memory 里拿进来，到 processor 里才能做操作

② data transfer
- lw (load word)
- sw (save word)

memory location: Offset (Base address)

↳ must be constant

③ Endianness

data 之间总体都是从小地址往大地址存的

寻址读整个永远先找 memory value 小的 byte，但可以是

0x00	0x01
12	34

or

34	12

但都是先读 0x00 再读 0x01，再根据 endianness 来决定怎么 "interpret" 数据

④ Immediate operands

(addi, no subi!)

constant zero: $zero (read-only)

↳ 但当成 register 用
- add $t2, $t1, $zero
- addi $t0, $zero, 5

Memory, register, constant

慢 ⟵ ⟶ 快

⑤ Logical operation (bit-by-bit，每个 bit 分别做 logical operation 再合起来)

and, or, nor / andi, ori (with an immediate value)

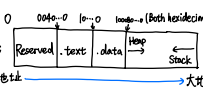
shift
- sll
- srl

整体往左/右移 k 格 e.g.: sll destination, source, shift_amount

得是 constant 而非 register

MIPS Program

① Areas in memory:



当我们在 .data 中 define variable 的时候，占用的内存都是连续的

```
.data
h: .word 1 2 3 4
s: .word 5
```

⇒ 4 byte 1 格 → 1 2 3 4 5 ⇒ 可通过 lw $s1, -4($s0)

declare 的 datatype:
- .byte (1 byte 一个)
- .word (4 byte 一个)
- .double (8 byte 一个)
- .ascii str (string)
- .asciiz str (string, 末尾 +'\0')

② Making decisions in MIPS

Conditional jump

beq : branch if equal
e.g.: beq reg1, reg2, L1
go to label L1 if reg1 == reg2

bne: branch if not equal
e.g.: bne reg1, reg2, L2
go to label L2 if reg1 != reg2

Unconditional jump

j : jump
e.g.: jump L1

jr: jump register → register用来jump之前要先lw把Label load给register
e.g.: jr reg (reg存着address)
通常在procedure call / case语句里用

这几个合起来可以implement if-else if-else语句

③ Loops

Basic blocks 就是一串sequentially执行的代码块 (不能从中间进来/出去)

One of the first early phases of compilation is breaking the program into basic blocks
A compiler identifies basic blocks for optimization
An advanced processor can accelerate execution of basic blocks

④ Comparison

slt : set on less than
e.g.: slt reg1, reg2, reg3
reg2 < reg3 : reg1 = 1
otherwise: reg1 = 0

slti : set on less than immediate
e.g.: slti $t0, $s2, 10
$t0 = 1 if $s2 < 10

如果要做 if (<) jump 操作，只能组合两条instruction

如果把comparison和jump统一到一起，那单条instruction就会变慢 (slower clock)
而且beq, bne才是common case

→ pseudo instructions

blt(z), bgt(z), ble(z), bge(z): branch if { l (less) / g (great) } + { t (than) / e (equal) } + (z) zero
(加了zero, format就变了)

⑤ Representing instructions

All 32 bits, 4 bytes, 3 format { R-format (register) / I-format (immediate) / J-format (Jump) }
前6位都是opcode，互相opcode不重合
decoding的时候看opcode就能知道是哪个format

R	opcode (31–26)	rs (25–21)	rt (20–16)	rd (15–11)	shamt (10–6)	funct (5–0)
I	opcode (31–26)	rs (25–21)	rt (20–16)	immediate (15–0)		
J	opcode (31–26)	address (25–0)				

R: 注意结果(也就是写code时候的第一个)是存在rd里的
rs, rt的顺序和code里一样
shamt: shift amount
funct: 当opcode为$(000000)_2$时，有很多种指令，这时候就靠function code区分

I: rs: 存着value的register
rt: 存着address的register
{ lw: rs ← rt / sw: rs → rt }
immediate: { constant / offset }

J:
to be continued (有个什么26bit的东西)

Instruction	Type	op	rs	rt	rd	shamt	funct	const/address
add	R	0	reg	reg	reg	0	32_{10}	-
sub	R	0	reg	reg	reg	0	34_{10}	-
and	R	0	reg	reg	reg	0	36_{10}	-
or	R	0	reg	reg	reg	0	37_{10}	-
sll	R	0	0	reg	reg	constant	0	-
srl	R	0	0	reg	reg	constant	2_{10}	-
addi	I	8_{10}	reg	reg	-	-		constant
lw	I	35_{10}	reg	reg	-	-		address
sw	I	43_{10}	reg	reg	-	-		address

- reg: a register number between 0 and 31
- constant/address: a constant or a 16-bit address (offset)

⑥ Other MIPS instructions

Load / Store a byte

lb $rt, offset($rs)   sign extension
lbu $rt, offset($rs)   zero extension
sb $rt, offset($rs)   把rt最右边的byte存到offset($rs)

Logical operation

nor $rd, $rs, $rt   把$rt换成$zero来implement logical not
sllv : sll 最后一个只能是register，这个可以是immediate了
sllr : same

⑦ Procedures

General purpose registers for procedure calling:
- $a0 – $a3: arguments (reg's 4 – 7)   For parameter passing: main传给procedure
- $v0, $v1: result values (reg's 2 and 3)   procedure返回回来的returned value存在这里，等着caller拿走
- $t0 – $t9: temporaries   暂时存储的(一般不需要一直保存)的data callee没有责任把它们的值keep在stack里再拿回去
  - Can be overwritten by callee   如果想caller想保存这些值的话 就在call之前存好
- $s0 – $s7: saved   callee有责任把它们存在stack里面的 然后回去之前再给它们存回register里 来保持s不变
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)   Stack pointer，指着stack的顶部(stack是从$sp的地方开始，内存最大值结束)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)   存着每次procedure结束之后回主程序的哪个地方

PC : 存着当前在执行的instruction的地址
(后面会学，在instruction fetch之后就会执行PC = PC+4)

jal ProcedureLabel1 (不是pseudo) { 当前"jal"指令的下一行的address被assign给$ra / jump到ProcedureLabel1 }
↳ 等这个procedure结束之后jr $ra回到原先位置

由于在nested procedure里，$ra会被override，引入$sp来辅助
callee开头先把$ra的值push到stack里去: { addi $sp, $sp, -4 / sw $ra, 0($sp) }
然后在procedure结束之前(jr $ra前一行)再把值从stack里pop出来 { lw $ra, 0($sp) / addi $sp, $sp, 4 }
(Leaf procedure 不用做push & pop 但是做了肯定没问题)
↳ 因为$ra不会被override

⑧ char & string

lb & sb 用来load/store char，做sign extension
psueodu instruction: la { lui : load upper immediate / ori }

la $s1, int { lui $s1, int (把int的前16位给$s1的前16位, then set the lower 16 bits of $s1 to be 0) / ori $s1, $s1, int (再把后16位和$s1的后16位or一起，$s1前16位不变) }

⑨ MIPS Arithmetic

1° instruction details about signed/unsigned adding & extension

add, addu, addi都是直接加，signed和unsigned的加法没差(运算规则没差)，差别只是signed的要检查overflow而unsigned的不用
addu和addiu 不说明instruction中的register/immediate中的值是unsigned的，只说明做完加法之后不检查是否overflow
↳ addiu是sign extension，因为要preserve value
(虽然unsigned的也会出现overflow的问题，但这是操作者的责任来保证它没有overflow，但如果做add(signed)的话，hardware会帮你查)

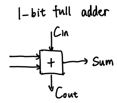
{ slt / sltu / slti / sltiu }
有u: 表示unsigned comparison (也就是按源码来treat每个数)
无u: 表示signed comparison (也就是按2's complement来treat每个数)
slti, sltiu → 要考虑extension: 都是sign extension (sltiu就是先做sign extension，再unsignedly地去比较)

lb $rt, imme($rs)   sign extension再跟$rs加，grab一个byte出来，再sign extension存进$rt里
lbu $rt, imme($rs)   sign extension再跟$rs加，grab一个byte出来，再zero extension存进$rt里
↳ load half同理，只有lw，没有lwu

andi $rt, $rs, imme   zero extension

2° ALU: arithmetic logic unit
↳ hardware in charge of { arithmetic operations / logical operations }

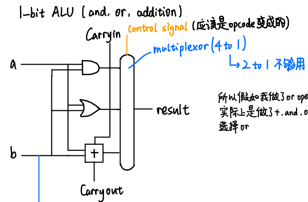
1-bit full adder



用truth table写出Cout, Sum
⇒ { $C_{out} = ab + bc + ca$ / $Sum = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$ }   (用k-map简化过了)

⇓

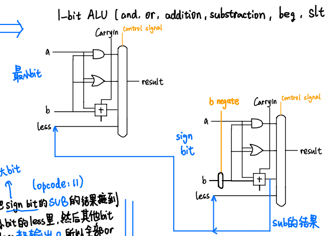
1-bit ALU (and, or, addition)



所以假如我做了or operation 实际上是做了+, and, or然后选择or

→ 假如想实现substract功能的话可以在这里放一个multiplexor (2 to 1) 然后把carryin设为1 → 下面有
(相当于先把b变成-b)

⟹ 1-bit ALU (and, or, addition, substraction, beq, slt)



SLT: (把sign bit的SUB的结果接到最小bit的less里，然后其他bit的less都输出0，所以全部or起来的输出就是sub完sign bit的值 (达到slt的效果))
最大bit (opcode: 11)

操作 - b negate - control signal对照表 (4 bit)

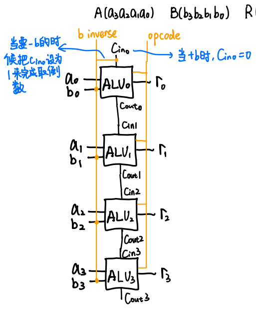
	op	$\bar{b}$	Cin
AND	00	0	0
OR	01	0	0
ADD	10	0	0
SUB	10	1	1
BEQ	10	1	1
SLT	11	1	1

SLT: 减完传给less gate看是否<0
SUB和BEQ的区别到底是啥?

→ beq: basic idea是做substraction，用两个register中的值相减看是否为0   咋实现?

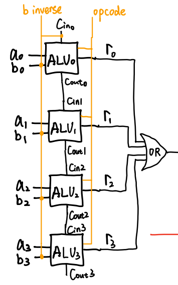
⇓

4-bit ALU (and, or, +, -)
$A(a_3a_2a_1a_0)$ $B(b_3b_2b_1b_0)$ $R(r_3r_2r_1r_0)$



当要-b的时候把Cin0设为1来完成取倒数
当+b时，Cin0 = 0

⟹ 4-bit ALU (and, or, +, -, beq, slt)
↳ equal comparison (branch在下一步做)



→ 这是SLT的图? 那不做SLT的时候怎么办? OR撤走?

Ripple Structure

优点: 32 bit的data只用了32个1 bit ALU存，省硬件

缺点: 下一个bit要等上一个bit算完，单线程，慢

Carry look ahead (Optional)

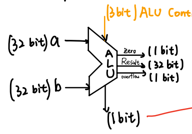
$$\begin{array}{cccc} & a_{i+1} & a_i & a_{i-1} \\ + & b_{i+1} & b_i & b_{i-1} \\ \hline & r_{i+1} & r_i & r_{i-1} \end{array}$$

实际的ALU: 每四位之间用传统的办法，但是四bit和四bit之间用Carry look ahead (8个并行单元)

define { $G_i = a_i \cdot b_i$ / $P_i = a_i + b_i$ }

$C_{i+1} = G_i + P_i \cdot C_i$
(要么该位全都为1，要么其中一个为1，而且Cin=1，Cout才能是1)
↳ 这个办法在用硬件换效率

Universal representation



↳ 2 bit 控制 multiplexor
1 bit 控制 b negate
zero是什么?
overflow是sign bit的Cout?
这玩意是啥?

Unsigned multiplication

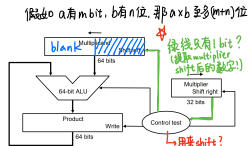
a × b → multiplier
↳ multiplicand

→ 列竖式

$$\begin{array}{r} 101 \\ \times\ 111 \\ \hline 101 \\ 101\ \ \\ 101\ \ \ \ \\ \hline 100011 \end{array}$$

思路: 把multiplicant加<multiplier的bit数>次
⇒ partial product = partial product + multiplicand (shifted)

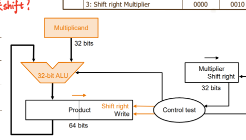
假如a有m bit，b有n位，那a×b至多(m+n)位



	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1→ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1→ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0→ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0→ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

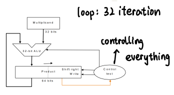
Multiplicand (4-bit)	1000	
Multiplier (4-bit)	1001	
Product (8-bit)	0000	0000
Add multiplicand	1000	
Product	1000	0000
Product (shift right)	0100	0000
Add 0	0000	
Product	0100	0000
Product (shift right)	0010	0000
Add 0	0000	
Product	0010	0000
Product (shift right)	0001	0000
Add multiplicand	1000	
Product	1001	0000
Product (shift right)	0100	1000

Addition on upper 4 bits



(changes made to previous version are highlighted in orange color)

考试只考这个



loop: 32 iteration
controlling everything

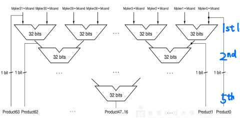
32-bit ALU
Multiplicand register: 32 bits, Product register: 64 bits (right half also used for storing multiplier)
Operations:
- The right half of the product register is initialized to the multiplier, and its left half is initialized to 0
- The two right-shifts at each step for version 2 are combined into only a single right-shift because the product and multiplier registers have been combined

Iteration	Multiplicand (M)	Product (P)	Remark
0	第0轮	0000 0011	Initial state
1	第1轮	0110 0011	Left(P) = Left(P) + M
		0011 0001	P = P >> 1
2	第2轮 0110	1001 0001	Left(P) = Left(P) + M
		0100 1000	P = P >> 1
3	第3轮	0100 1000	No operation
		0010 0100	P = P >> 1
4	第4轮	0010 0100	No operation
		0001 0010	P = P >> 1

Multiplier

initial state: 左半边留着全0，右半边放multiplier
第i轮: ① 看前一轮的partial product最后面那位是0 or 1 → 其实是multiplier右数第i位
是0: 不动，是1: partial product左半边 + multiplicand
② partial product整体往右shift一位
↳ i: multiplicand和multiplier的bit数

Hareware Speedup



	shift value
1st layer	2^0
2nd layer	2^1
⋮	⋮
5th layer	2^4

用硬件换时间的做法

比如 $(1100)_2 \times (1001)_2$ 本来是

```
 x  1100
    1001
   -----
    1100
   0000
  0000
 1100
```

这样加

在左图中就是把前俩 combine，先加好，再加后俩（第一层）

然后再把两堆加起来（第二层）

但要注意，每层加的时候 shift 的 value 依次 ×2

Multiplication in MIPS

Hi. Lo is used to store the product

- mult $rs, $rt
- multu $rs, $rt

乘的方式没区别，overflow 检测有

(R-format)

乘完可能超过 32 bit ⇒ 用两个特殊 register 存着

Fetch Results

- mflo (move from lo)　　mflo $s1
- mfhi (move from hi)　　mfhi $s1

Overflow testing（来看能不能把整个 result 存在一个 32 bit register 里）

① Unsigned overflow

```
multu $s0, $s1
mfhi $at
mflo $s2
beq $at, $0, no_overflow
j overflow
no_overflow:
```

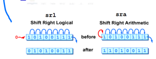
乘完之后 upper 32 bit 给 Hi，lower 32 bit 给 Lo

那假如 upper 32 位全为 0，那说明没有 overflow，32 bit 就能装下结果

② Signed overflow

```
mult $s0, $s1
mfhi $at
mflo $s2
sra $t0, $s2, 31
beq $at, $t0, no_overflow
j overflow
no_overflow:
```

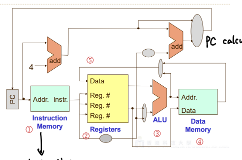
即检查 upper 32 bit 是不是都是 sign bit，是就没有 overflow



sra 就是在 srl 的同时在左边补上 sign bit

（无 sla）

MIPS Processor Overview



PC calculation

Instruction Memory → instruction cache (not real memory)

Example

- Division of a 4-bit unsigned number (0111) by another one (0011)

dividend

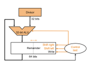
Iteration	Divisor (D)	Remainder (R)	Remark
0	0011	0000 0111	Initial state
		0000 1110	R = R << 1
1		1101 1110	Left(R) = Left(R) – D
		0000 1110	Undo
		0001 1100	R = R << 1, R_0 = 0
2		1110 1100	Left(R) = Left(R) – D
		0001 1100	Undo
		0011 1000	R = R << 1, R_0 = 0
3		0000 1000	Left(R) = Left(R) – D
		0001 0001	R = R << 1, R_0 = 1
4		1110 0001	Left(R) = Left(R) – D
		0001 0001	Undo
		0010 0010	R = R << 1, R_0 = 0
extra		0001 0010	Left(R) = Left(R) >> 1

Remainder　correction　Quotient

Hardware Optimized Version (cont.)

- 32-bit ALU
- Two registers:
  - Divisor register: 32 bits
  - Remainder register: 64 bits

  (right half also used for storing quotient)



- Operations:
  - 32-bit divisor is always subtracted from the left half of remainder register
    - The result is written back to the left half of the remainder register
  - The right half of the remainder register is initialized with the dividend
    - Left shift remainder register by one before starting
  - The new order of the operations in the loop is that the remainder register will be **shifted left one time too many**
    - Thus, **final correction step:** must **right shift back only the remainder** in the left half of the remainder register

initial state：左半边全为 0，右半边是 dividend

☆ 起手先 shift left 一格!!

第 i 轮：左半边减 divisor，观察相减后的 sign bit

- 1：undo 相减操作并左移一格，右边填 0
- 0：左移一格，右边填 1

i = divisor 的 bit 数

☆ 结束时 shift right 一格!!

?

Ideal pipeline analogy (M jobs, N stage, T time/stage)

sequential ⟶ pipeline : speedup = $\frac{T_{seq}}{T_{pip}}$

$= \lim_{M \to \infty} \frac{MNT}{(M+N-1)T}$

$= N$

unbalanced pipeline ⟶ not applicable

↓

can not align perfectly (should wait, add stall time)

Each step is independent of each other and takes different datapath