Final 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 (包括):
- native data types
- instructions
- registers
- addressing modes
- memory architecture
- interpret and exception handling
- external I/O

→ 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. memory operation
5. write back

2. Digital Logic

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

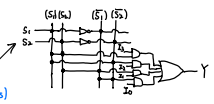
logic:
- truth table
- logical expression
- Graphic form (K-map)

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
- 1 output

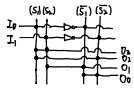
(n selected lines)

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 Sum (min)
- POS (max): 所有结果为 0 的 input 加起来的积 Pro (max)

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

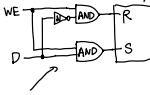
Sequential logic

① S-R Latch

NOR Gate implementation:

R=1, S=1 ⇒ forbidden state → $\overline{1}=0 \Rightarrow 0=\overline{0}$ or $\overline{1}$ (X)
R=0, S=0 ⇒ Latch → "Quiescent" State
R=0, S=1 ⇒ set
R=1, S=0 ⇒ Reset

NAND Gate implementation:



R=0, S=0 ⇒ forbidden state → $0=\overline{1} \Rightarrow 1=\overline{0}$ or $\overline{0}$ (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

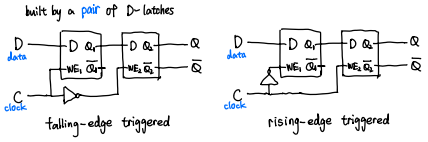
a shared signal that controls all the circuit element about time

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

$(-6)_{10} = (1010)_2$ ← 1010 ←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 一点。)

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

(more in MIPS)

注意：(段如要把 -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

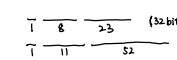
注意因为这个源码，由9是正数所以用 0-extension

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

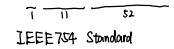
② floating point numbers

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 比较行)

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，小数部分分别拆，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 expo + 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: $(-1)^1 \times 1.1\cdots1 \times 2^{(11\cdots10)_2-127} = 1.1\cdots1 \times 2^{127} \approx -2^{128}$
- -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: $(-1)^0 \times 1.11\cdots1 \times 2^{(11\cdots10)_2-127} = 1.1\cdots1 \times 2^{127} \approx 2^{128}$

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 只有 10 位

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！！！

2° de-normalized case

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

significand 直接读，但没有 default 1

比如后者是 0.000…0 01100…0

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

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)

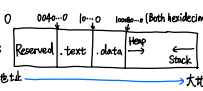
shift
- sll
- srl

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

得是 constant 而非 register

MIPS Program

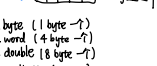
① Areas in memory:



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)



⇒ 可通过
- la $s0, s
- lw $s1, -4($s0)

来访问 h[3]=4

declare 别的 type:
- .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 L2

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(不能从中间进入/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 $0, $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 (2), bgt (3), ble (3), bge (3): branch if { l (less), g (great) } + { t (than), e (equal) } + (3) zero
(加) zero, format就变3)

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

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

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:

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那个地方

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 结束后 jr $ra 回到原来位置

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
pseudo 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 Addressing mode

(1) Immediate addressing

addi $s0, $s0, 4
无需访问 register/RAM

(2) register addressing

add $s0, $s0, $s1
要访问 register, 不用访问 RAM

jr $t0 (R format, 20~6位全是0, 没用上) 可以jump去RAM的任何地方
要访问 register & RAM

(3) base addressing

lw $t1, 0($t0)
同时访问 register & RAM

(4) PC-relative addressing

beq $s0, $s1, Label
Immediate
真正存进 immediate 的是 (Target-address − PC−4) ÷ 4
因此 Label 可以是 PC 前后 2^{15} 条 instruction

(5) pseudo-direct addressing (embed 32-bit immediate in 26-bit storage)

jal Label / j Label
Can jump to any instruction
↳ ∵ 最前面4位一定是0111 (所有instruction的地址都以0111开头)
最后面2位一定是00 (instruction 一定是4的倍数)
∴ 可以只存26位

∴ 当要 jump 去的 address 太远时, 把 beq 换成 bne + j

⑩ 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)

Overflow handling: ⌐ 感觉类似一个 handling function
PC → jump to a predefined code → jr 回原来地方
Error address saved to EPC (special register)
↳ 通过 $epc 重新处理该 instruction

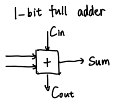
{ slt, sltu, slti, sltiu }
有u: 表示 unsigned comparison (也就是按源码来treat每个数)
无u: 表示 signed comparison (也就是按 2's complement 来treat每个数)
slti, sltiu → 要考虑extension: 都是 sign extension (sltiu 就是先做sign extension, 再unsignedly地去比较)

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

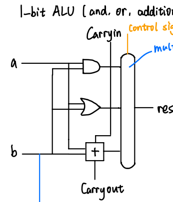


用 truth table 写出 Cout, Sum

$$\Rightarrow \begin{cases} C_{out} = ab + bc + ca \\ Sum = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc \end{cases}$$
(用 k-map 简化过了)

⇓

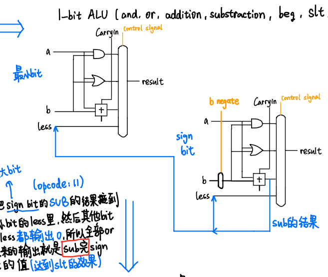
1-bit ALU (and, or, addition)



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)



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

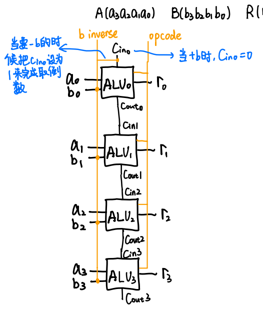
SLT: 减完传给 less gate 看是否 < 0
发现 $\bar{b}$ 和 Cin 值总一样, 于是合成同一个 bnegate

SLT: (opcode: 11) (把 sign bit 的 SUB 的结果连到 最大bit → 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)

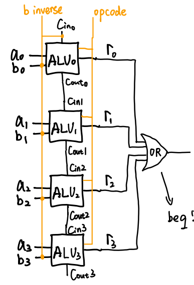
beq: basic idea 是做 substraction, 用两个 register 中的值相减看是否为0

⇓

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



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



beq 实现 (∵ 每位sub完, 只要有一个不是0, 就不==)

Ripple Structure

优点: 32 bit 的 data 只用了 32个 1 bit ALU 而, 省硬件
缺点: 下一个bit要等上一个bit算完, 串行程, 慢

Carry look ahead (Optional)

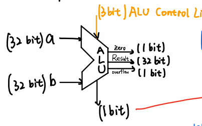
$$\begin{array}{r} C_{i+1}\ C_i \\ a_{i+1}\ a_i\ a_{i-1} \\ +\ b_{i+1}\ b_i\ b_{i-1} \\ \hline r_{i+1}\ r_i\ r_{i-1} \end{array}$$

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

define $\begin{cases} G_i = a_i \cdot b_i \\ P_i = a_i + b_i \end{cases}$

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

Universal representation



(3bit) ALU Control Lines → 2 bit 控制 multiplexor; 1 bit 控制 b negate

* zero是什么? overflow是sign bit的Cout?

(1 bit) → 这玩意是啥?

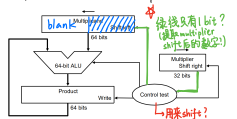
Unsigned multiplication

a × b → multiplier → multiplicand

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

```
    101
  x 111
  -----
    101
   101
  101
  -----
 100011
```
→列竖式

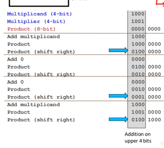
假如 a有m bit, b有n位, 那 a×b 至多(m+n)位

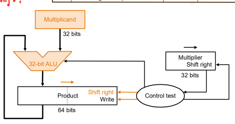


* 为什么只有1 bit? (最低位 multiplier shift后的数字!)

→ 用来shift?

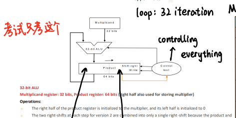
	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1→ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1→ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift 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





考试只考这个

loop: 32 iteration



controlling ↑ everything

product　multiplier

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

Multiplicand * Multiplier

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

(第0轮, 第1轮, 第2轮, 第3轮, 第4轮; 最后一位 → Multiplier)

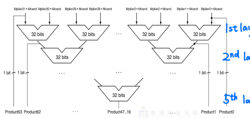
其实是 multiplier右数第i位

initial state: 左半边留着全0, 右半边放multiplier

第i轮: 看前一轮的partial product 最后那位是0 or 1
- 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

```
    1100
  x 1001
  ------
    1100
   0000
  0000
 1100
```
这样加

multiplier[i]·Mcand

```
  1100      0000
+ 0000    + 1100
  01100     11000
```
01100 + 11000 = 1101100

在左图中就是把前两个combine, 先加起, 再加后两(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32bit ⇒ 用两个特殊register存着

Fetch Results
- mflo (move from lo)　mflo \$s1
- mfhi (move from hi)　mfhi \$s2

Overflow testing (本着不能把整个result 存在一个32 bit register里)

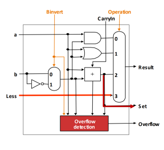
① Unsigned overflow

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

乘完后 upper 32bit 給Hi, lower 32bit 給Lo

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

Signed Overflow detection



1-bit ALU for the MSb (bit 31)

拿ADD作例子

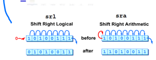
假如两个数均为正, 结果负 ⇒ overflow

一正一负加不可能overflow

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)

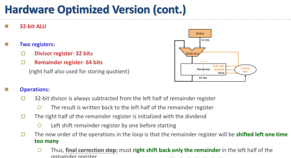
Example

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

Iteration	Divisor (D)	Remainder (R)	Remark
0		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	0011	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.)

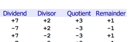


initial state: 左半边全为0, 右半边是 dividend

* 起手先 shift left 一格!!

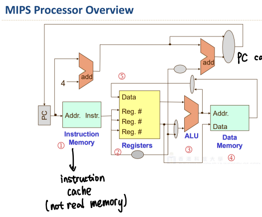
第i轮:
- 左半边 减 divisor, 观察相减后的 sign bit
  - 1: undo 相减操作 并左移一格, 右边填0
  - 0: 左移一格, 右边填1
- i = divisor 的 bit数

* 结束时 shift right 一格!! → 只 shift remainder, 不 shift quotient

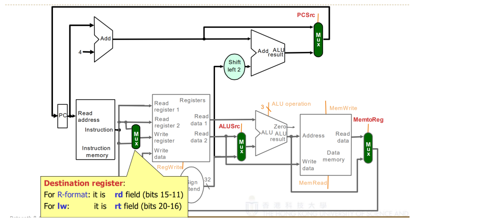


⑪ Single-cycle implementation

MIPS Processor Overview

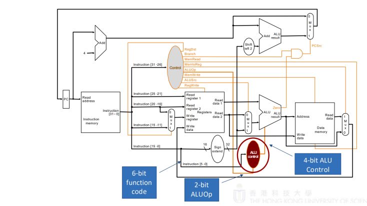


Full Datapath: Muxing Two Possible Destination Registers



⑫ Control signal

Generation of ALU Control Input Bits



Setting of control lines (output of control unit):

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

sw & beq will not modify any register, it is ensured by making RegWrite to 0

So, we don't care what write register & write data are

⑬ MIPS pipeline

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

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

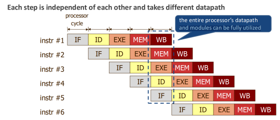
$= \lim_{M\to\infty} \frac{MNT}{(M+N-1)T}$

$= N$

unbalanced pipeline → not applicable

cannot align perfectly (should wait, add stall time)

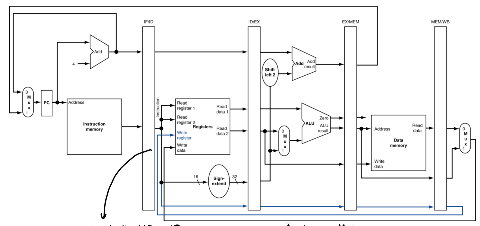
Each step is independent of each other and takes different datapath



single cycle

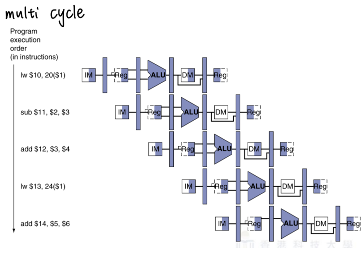
mips design for pipeline
- instructions of same length
- few instruction formats
- memory operands only in lw, sw
- alignment in word boundaries

Stages:
- IF　instruction fetch
- ID　instruction decoding
- EX　execution
- MEM　memory operation
- WB　write back



write back 的时候, 得从 MEM/WB register拿, 因为之前的 write register 已经被后面的进程 override了

multi cycle

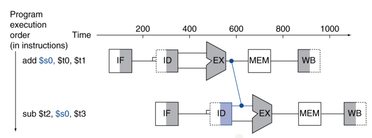


Hazards

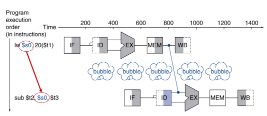
-- Structural Hazards: (两个操作用到同一个硬件)
- 一边 MemRead 一边 Instruction Fetch ⇒ Solution: separate instruction/data memory
- 一边 regWrite 一边 regRead ⇒ Solution: Register file 支持前半clock cycle Write, 后一半 Rea

-- Data hazards: (pipeline stall/bubble: delay of execution)

(后面的操作用到前面的data) ⇒ Solution: Forwarding



但有时候还是得bubble (前一步是lw尤其注意)



-- Control hazards　上一步是 beq 情况下, 下一步要等比较结果才知道是否 branch

⇒ Solution: 不管上一步beq 结果, 继续做 IF

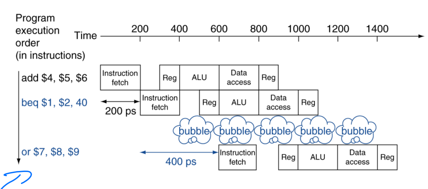
假如① 真的不用branch: 正常继续跑

② 要 branch:

beq: IF | ID | EX

下一步: IF | IF

加了额外 hardware 让beq 的结果在 ID阶段能出来, 假如要branch, 那就计算新PC, 下一步继续IF



(14) Memory technology

Memory { RAM { Static RAM (cache); Dynamic RAM (main memory) }; Hard Disk (secondary storage) }

SRAM Technology

- SRAM is a type of semiconductor memory
  - Each cell is constructed using transistors only (i.e. not capacitors)
  - Density not as high as DRAM, it is more expensive
  - Does not need to be periodically refreshed, the memory retains its contents as long as power remains applied
  - Read is not destructive in SRAM design; that's why called static

Major problem of using capacitors:

- Real-world capacitors are not ideal, hence leak electrons
- Information stored in capacitors eventually fades
- Need to recharge the capacitors periodically to restore the "values"
- Read operation discharges the electrons; read is destructive

Because of this refresh requirement, it is called dynamic memory

Hard Disk Calculation

Access to a sector involves

- Queuing delay if other accesses are pending
- Seek: move the heads
- Rotational latency
- Data transfer
- Controller overhead

average = seek time + 转半圈的时间 + data transfer time + controller delay

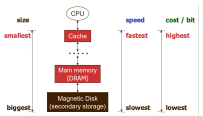
Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
  - + ½ / (15,000/60) = 2ms rotational latency
  - + 512 / 100MB/s = 0.005ms transfer time
  - + 0.2ms controller delay
  - = 6.205ms
- If actual average seek time is 1ms
  - Average read time = 3.205ms

Locality

{ temporal locality: 最近用过的很快还会再用
spacial locality: 最近用过的边上的很快会用上

Memory hierachy



CPU request for data的时候：
- 先层层找！先找 cache，没找再找 main memory
- 最后再找 hard disk

对任意一层：找到 hit
无 miss ⟶ suffer from miss penalty

{ Hit rate: hit/all
miss rate: miss/all

Cache hit time:
= time to determine miss or hit + time to access the cache

Cache miss penalty:
= time to bring a block from lower level to upper level

Hit time << miss penalty

Cache Performance

Average memory access latency

= hit_time + miss_rate * miss penalty (期望)

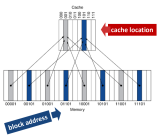
(两层) = hit_time + miss_rate * (hit_time' + miss_rate' * miss penalty') / miss penalty

Cache Operations

Cache Organization
↳ array of cache blocks 相当于最小单位 (一般32/64 bits)

{ direct-map
set-associative
fully-associative

direct-map 一般取模 (缺点：frequent move in & kick out)



e.g.：要address为1200的东西

⇒ 第 ⌊1200/32⌋ = 37个 block 被送到 cache 中第 37%8 = 5个位置

标签：记录该block在哪8个block里

Byte address 1200: 0000 0000 0000 0000 0000 0100 1011 0000
Tag | Index | Block address | Byte offset

block 在 cache 中的位置

该bit 在block中的位置

Valid bit: mark着该block是否存着有效数字
(initialize的时候都是垃圾值，valid bit就都是0)

Bits in Cache

- A direct-mapped cache with 16KB of data and 8-word (2^5 bytes) blocks
- Assume 32-bit memory address
- What is the actual cache size?

16KB = 4K words = 2^{12} words
Block size of 8 words -> 2^3 blocks
Each block has, 8 x 32 = 256 bits of data
A tag has 32 - 9 - 5 = 18 bits
And, 1 valid bit
Total bits per entry = (256 + 18 + 1)

实际上里存的东西除了data还有valid bit和tag

Total cache size
2^9 x (256 + 18 + 1) = 2^9 x 275 = 140800 = 137.5 Kbits

Cache miss

<TBC>

Associate Cache

Fully-associative

每个block都能放在cache里任意位置
(缺点：找空位麻烦) 像 open addressing

N-way set associative

每个block只能放在cache里指定的n个位置

- Example: associativity in a cache with 8 blocks

像 separate chaining

increase N (associativity)
⇒ { 好处：miss rate↓
坏处：hit time↑
∴一般用2way / 4 way 即可

Block replacement

When cache misses:
{ direct map：没有这个 concern
n associative：LRU (Least recently used)

① replace LRU
② set it to MRU

① 0001 换 0011
② 把 0001 设为MRU
③ 把 其他往下移
(0010 自动变为LRU)

Data modification (dirty data processing)

(快，麻烦)
{ write back：数据修改后写回cache，cache 回收 返回memory时再对 main memory修改 (被kicked out/replace)
write through：只要数据发生改动就存回 cache 和 main memory
(很简单，慢，需要 write buffer)

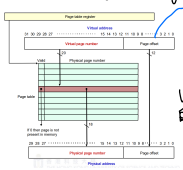
Virtual memory

hard disk 把 main memory 当 cache 用

physical address = f(virtual address)
↳ page table
↳ 一个关于 block的单位

Page table (在memory里，起始 address stored in page table register)

不用 tag 因为是 1-1 mapping (查时间)，不会有2个 virtual page map到同一个physical page) ⇒不用验证
(cache 是 many-1)



block在page里的位置

由 virtual page number 直接唯一的 physical page number

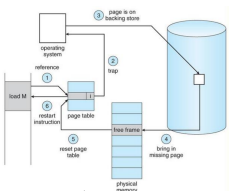
Valid bit：判断 memory 的该page是否为有效值

Page fault

Valid bit == 0 或 page table中没有该mapping ⇒ 不在在 valid 的 memory

∴ OS 把 data 从 hard disk 里调出来 慢

流程：



程序会停下来等 page fault resolve

∴ page size { 太大 浪费空间，导致 page fragmentation
太小 总有 page fault

Fast address Translation

在cache里存一个 page table 的 copy ⇒ TLB
要不然每次 access 两次 memory (一次 page table 一次拿 data)

Draw K-map for output G_1 and then use it to derive the simplest sum-of-product logic expression of G_1 (please circle the groups of 1's in the K-map). (5 marks)

G_1 = $B_0\overline{B_3} + \overline{B_0}B_3$

$B_1\overline{B_2} + \overline{B_1}B_2$

2

搞反了……
前两列是B_3B_2，后两列是B_1B_0……

Student ID:

```
# load word[i],word[j]
sll $t2, $t0, 2
add $t2, $t2, $a0
lw $t4, 0($t2)
sll $t3, $t1, 2
add $t3, $t3, $a0
lw $t5, 0($t3)
```

不用×4 因为是char array

lbu ×

lbu ×

用lbu不用lb：char是unsigned地储存在memory里的

```
continue:
  addi $t0, $t0, 1
  addi $t1, $t1, -1
  j while_loop
return:
```

Excluding NANs and ±∞, how many **distinct numbers** can be represented standard? Note: +0 and -0 are considered to be a single number, +1.5 an hand are two numbers. Justify your answer or show your calculation briefly

$2 \times (2^5-1) \times 2^{10} = 2^{16} - 2^{11}$

sign, exponent, significand

$2^{16}-2^{11}-1$

全0数了两遍 (sign bit 不一样但其实一样)

$2 \times$ (sign)
- exponent = 0, significand = 00⋯0
- exponent = 1, significand = 0/1 0⋯0
- exponent = 2, significand = 0/1 0/1 0⋯0
- ⋮
- exponent = 10, significand = 0/1 ⋯ 0/1 (10)
- ⋮
- exponent = $2^3+2^2+\cdots+2^0 = 15$, significand = 0/1 ⋯ 0/1

$2 \times (1 + 2^1 + 2^2 + \cdots + 2^{10} + 2^{10} \times 5)$

$= 2 \times (2 \times 2^{10} - 1 + 5 \times 2^{10})$

$= 2 \times (7 \times 2^{10} - 1)$

$= 7 \times 2^{11} - 2$

少了个0

$7 \times 2^{11} - 1$

00001 00100	adjust remainder

注意最后adjust remainder的时候只动remainder（左边）不动quotient（右边）

0x8FFFFFF8
0x08C02001
undetermined
~~undetermined~~ 0x00000000
undetermined
undetermined

正常读register 2是$zero

	1	2	3	4	5	6	7	8	9	10	11	12
$8, $1, $4	IF	ID	E	M	WB							
$7, $8, $4		IF	ID	E	M	WB						
$8, 4($7)			IF	ID	E	M	WB					
$1, $2, $8				BUB	IF	ID	E	M	WB			
$1, 4($7)						IF	ID	E	M	WB		

还是用M→M

理由：假如forwarding path是通过把算出来的值放在stage中间的register里，那add时放在E/M里的会被sw时放的overwrite掉

Consider the following MIPS procedure. The numbers at the left are the line numbers of the code.

```
1)  procedure:
2)  label1:     addi $t0,$zero,1
3)              add  $v0,$zero,$zero
4)  label2:     slt  $t7,$t0,$a1
5)              beq  $t7,$zero,label3
6)              sll  $t1,$t0,2
7)              add  $t1,$t1,$a0
8)              lw   $t2, 0($t1)
9)              lw   $t3, -4($t1)
10)             slt  $t7,$t3,$t2
11)             beq  $t7,$zero,label4
12)             addi $t0,$t0, 1
13)             j    label2
14)  label3:    addi $v0,$zero,1
15)  label4:    jr   $ra
```

$t_0=1$ #counter; $v_0=0$; $t_0 \geq a_1$ 去label3; $t_0<a_1$ 进泣; t_1是第t_0个内容; t_3是(t_0-1)个内容; $t_3 \geq t_2$ 就滚

a) Assume that register $a0 holds the base address of an *integer array* A, and register $a1 holds an integer indicating the number of elements in array A (i.e. if array A has 6 elements, $a1 will be 6). What will be the return value of the procedure in $v0, if A={1,5,3}? You just need to write a single value. (3 marks)

~~2~~ 0

b) By referring to part (a) or otherwise, explain what the return value in $v0 indicates with respect to the elements in array A using one sentence only. (4 marks)

the max length of ascending sequence in array A

ascending order → return 1; otherwise 0

c) The procedure does not preserve the $ra register value. Explain in one sentence whether this would create potential problem. (3 marks)

no, this would not create potential problem since no "jal" is used so $ra won't be modified in the whole process

Iteration	Divisor (D)	Remainder (R)	Action
0		00000 11100	Initial State
		00001 11000	Shift left
1		11100 11000	substraction
		00001 11000	undo
		00011 10000	shift left, set LSB 0
2		11110 10000	substraction
		00011 10000	undo
		00111 00000	shift left, set LSB 0
3	00101	00010 00000	substraction
		00100 00001	shift left, let LSB 1
		11111 00001	substraction
4		00100 00001	undo
		01000 00010	shift left, let LSB 0
		00011 00010	substraction
5		00110 00101	shift left, let LSB 1
		00011 00101	adjust remainder
6			

标准写法是在不用undo的时候空一行

我记得是 and 00, or 01, + 10, less 11

b) Write down the values of control signals required for the lw instruction. (4 marks)

Note:
1. A "don't care" control signal is represented by an "x". If the signal is a "don't care", onl "x" is considered correct (i.e. "0", "1" are both incorrect).
2. Below table asks for 2-bit ALUOp but not 4-bit ALU control

RegDst	ALUsrc	MemtoReg	RegWrite	MemRead	MemWrite	ALUOp1	ALUOp0
0	1	1	1	1	0	0	~~1~~ 0

```
        lw   $s1, 0($sp)
        addi $sp, $sp, 4
        andi $t3, $s1, 1
        beq  $t3, $zero, exit
        add  $v0, $v0, $s1
exit:
        lw $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra
end:
```

$S_1 = arr[0]$; $S_1=0$, exit; $S_1 \neq 0$, $V_0=V_0+S_1$

筛选奇数

stored in $v0: ~~23~~ 5

be in one sentence what the procedure does:

~~calculate the sum of arr~~

calculate the sum of odd elements in arr

6

0001 0000 0100 0011 0000 0000 0000 1000

ruction below (for example if you think it is an add instruction, then you can swer)

beq $v0, $v1, <Label>

I: opcode rs rt imme

or example if you think the register encoded in the $rs field is $s0, you can swer):

$rs: ~~$v1~~ $v0

or example if you think the register encoded in the $rt field is $s0, you can swer):

$rt: ~~$v0~~ $v1