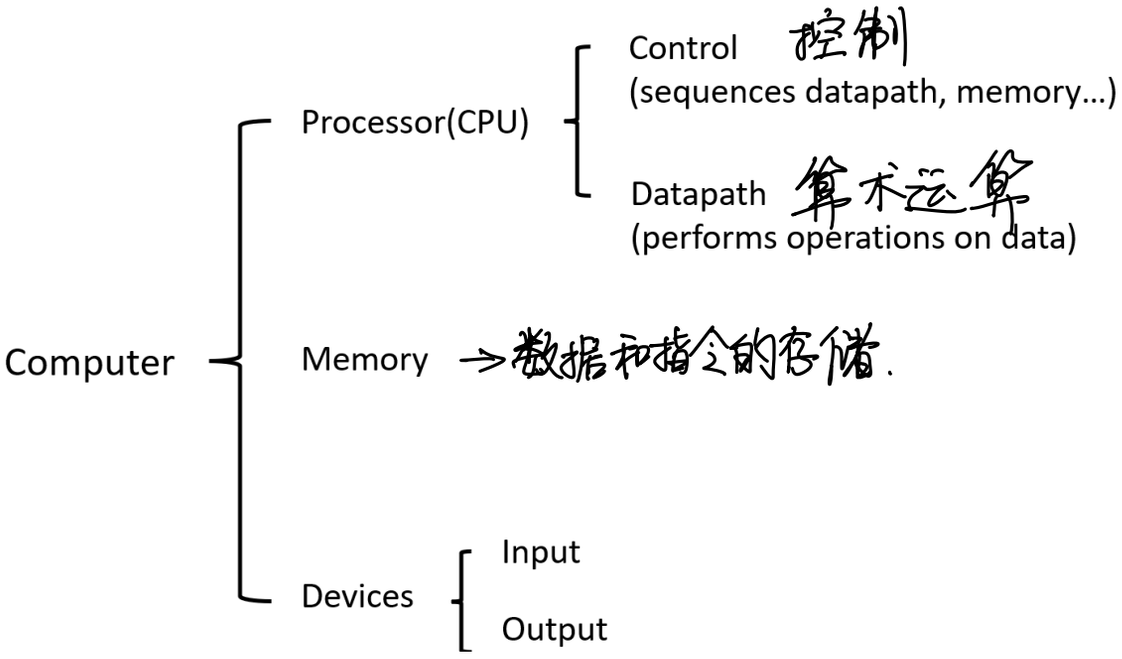
Computer Organization Review

Chapter 1

② 1. Components of a computer

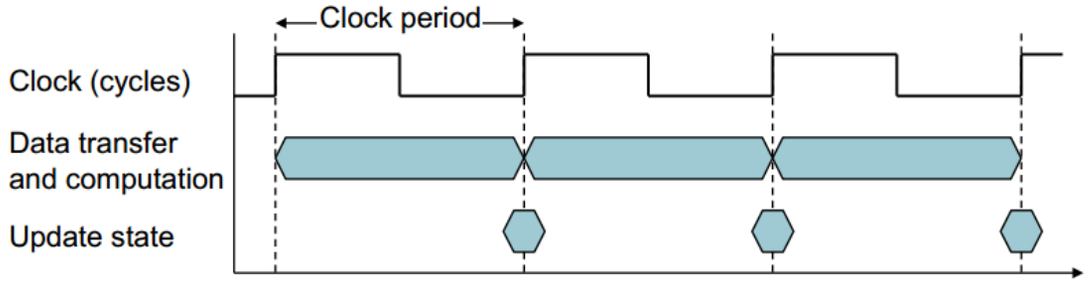


① 2. Moore's Law

The number of transistors that can be integrated would double every 18 to 24 months.

③ 3. CPU clocking

Operation of digital hardware governed by a constant rate clock.



$$CPU\ Time = No.of\ Clock\ Cycles \times Clock\ Period$$

$$CPU\ Time = No.of\ Clock\ Cycles\ /Clock\ Rate$$

③ 4. Instruction Count and CPI

Instruction count for a program is determined by program, ISA and compiler.

Average cycles per instruction is determined by CPU hardware.

$$Clock\ Cycles = Instruction\ Count \times Cycles\ per\ Instruction$$

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Cycle\ Time$$ →即 clock period

$$CPU\ Time = \frac{Instruction\ Count \times CPI}{Clock\ Rate}$$

If different instruction classes take different numbers of cycles:

$$Clock\ Cycles = \sum_{i=1}^{n}(CPI_i \times Instruction\ Count_i)$$

Weighted average CPI:

$$CPI = \frac{Clock\ Cycles}{Instruction\ Count} = \sum_{i=1}^{n}(CPI_i \times \frac{Instruction\ Count_i}{Instruction\ Count})$$

Performance Summary:

$$CPU\ time = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle}$$

Performance depends on: Algorithm, Programming language, Complier and Instruction set architecture.

Speedup of complier A versus Complier B:

$$Speed\ Up = \frac{Clock\ Rate_A}{Clock\ Rate_B}$$

(3) 5. Dynamic Power

static 和 dynamic 是独立的两种功率

$$Power = \frac{1}{2} \times Capacitive\ load \times Voltage^2 \times Frequency$$

(1) 6. Multiprocessors

More than one processor per chip;
Requires parallel programming;

(1) 7. Amdahl's Law

Performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play.

当提升系统的一部分性能时，对整个系统性能的影响取决于:1、这一部分有

多重要 2、这一部分性能提升了多少。

题型：
1. 概念题
2. 计算题

Chapter 2

1. MIPS design principle

   a. Simplicity favors regularity
   b. Smaller is faster
   c. Make the common case fast
   d. Good design demands good compromises

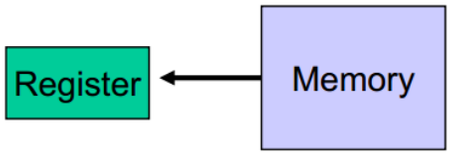
2. Register

   32 register;
   Each register is 32-bit wide (or 64-bit wide registers in modern 64-bit architecture);
   32-bit entity (4 bytes) → a word

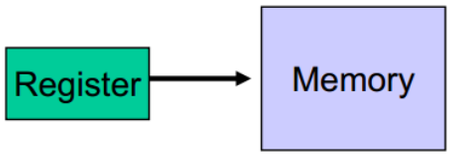
3. Memory operands

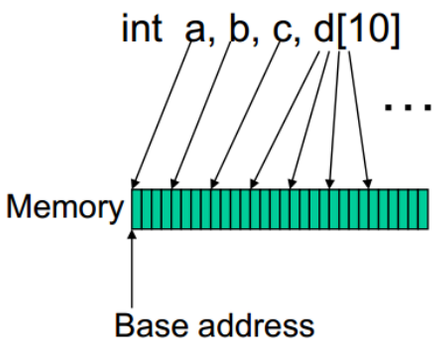
- Values must be fetched from memory before (add and sub) instructions can operate on them

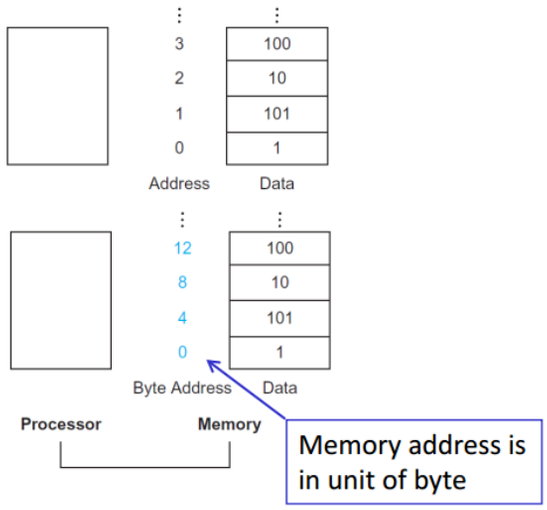
Load word
lw $t0, memory-address

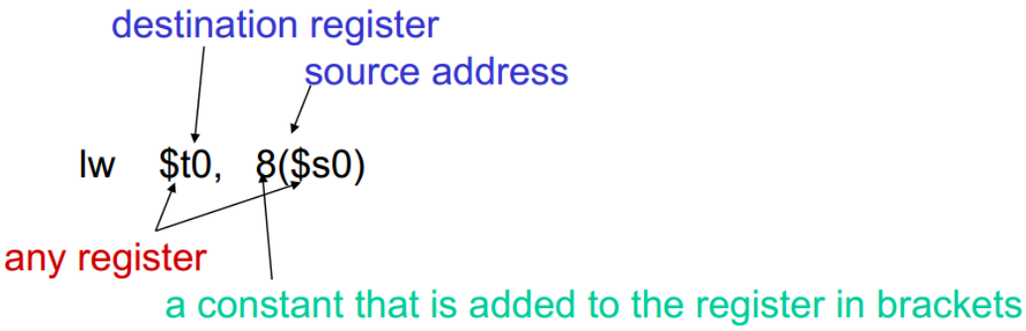


Store word
sw $t0, memory-address









4. Numeric Representations

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)
  0x 23 or 23_{hex}

0-15 (decimal) → 0-9, a-f (hex)

无符号数

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range: 0 to $+2^n-1$

有符号数 负数：取反+1

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range: -2^{n-1} to $+2^{n-1} - 1$

Sign Extension:
正数前补0
负数前补1

Complement and add 1

- Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$
$$\bar{x} + 1 = -x$$

Overflow is usually when the sign of the result doesn't make sense compared to the operands.

5. Instruction Formats

opcode

The field that denotes the operation and format of an instruction.

rs

The first register source operand.

rt

The second register source operand.

rd

The register destination operand. It gets the result of the operation.

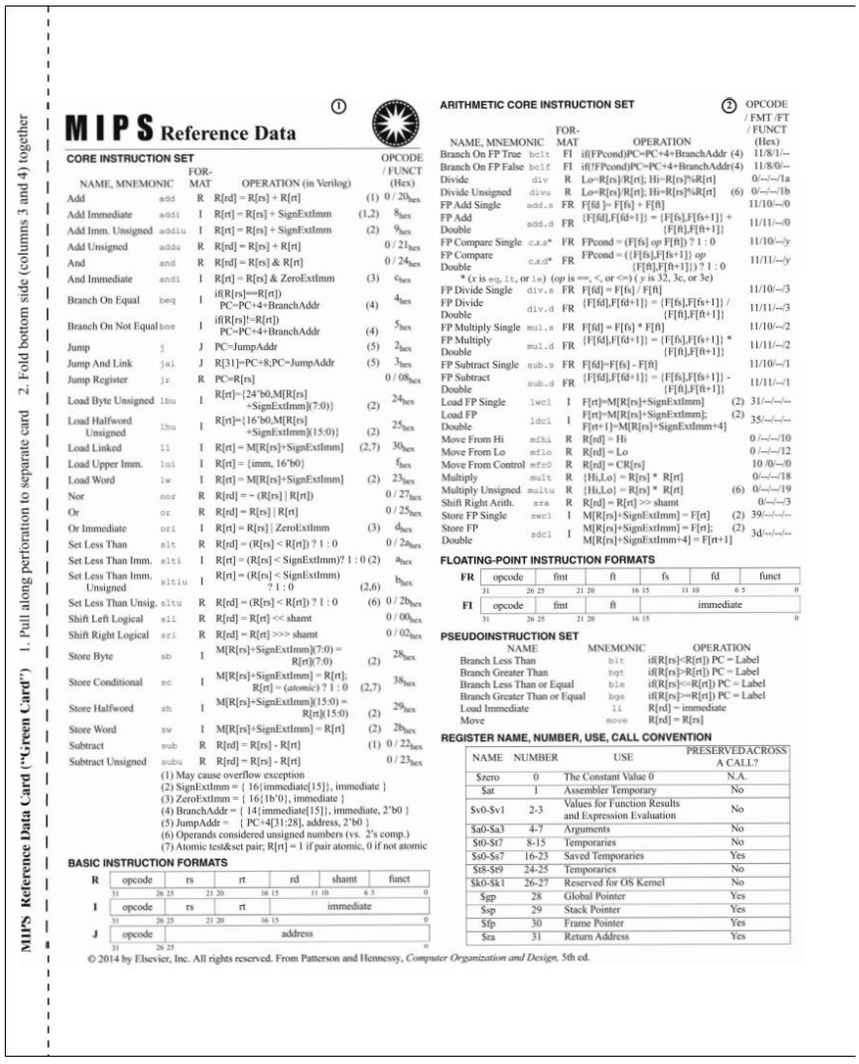
shamt

Shif amount.

funct

Function. This feld, ofen called the function code, selects the specifc variant of the operation in the op field.

Instruction Type: R\I\J



MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1'b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

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

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION	OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bclt	FI	if(FPcond)PC=PC+4+BranchAddr (4)	11/8/1/--
Branch On FP False	bclf	FI	if(!FPcond)PC=PC+4+BranchAddr(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt] (6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]	11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}	11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0	11/10/--/y
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0	11/11/--/y
* (x is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)				
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]	11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}	11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]	11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}	11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]	11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}	11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm] (2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; (2) F[rt+1]=M[R[rs]+SignExtImm+4]	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi	0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo	0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]	10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]	0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt] (6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt	0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt] (2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; (2) M[R[rs]+SignExtImm+4] = F[rt+1]	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct	
	31 26	25 21	20 16	15 11	10 6	5 0	
FI	opcode	fmt	ft	immediate			
	31 26	25 21	20 16	15 0			

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

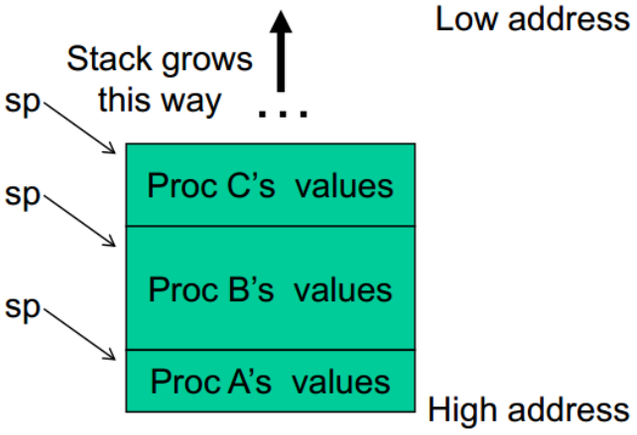
REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

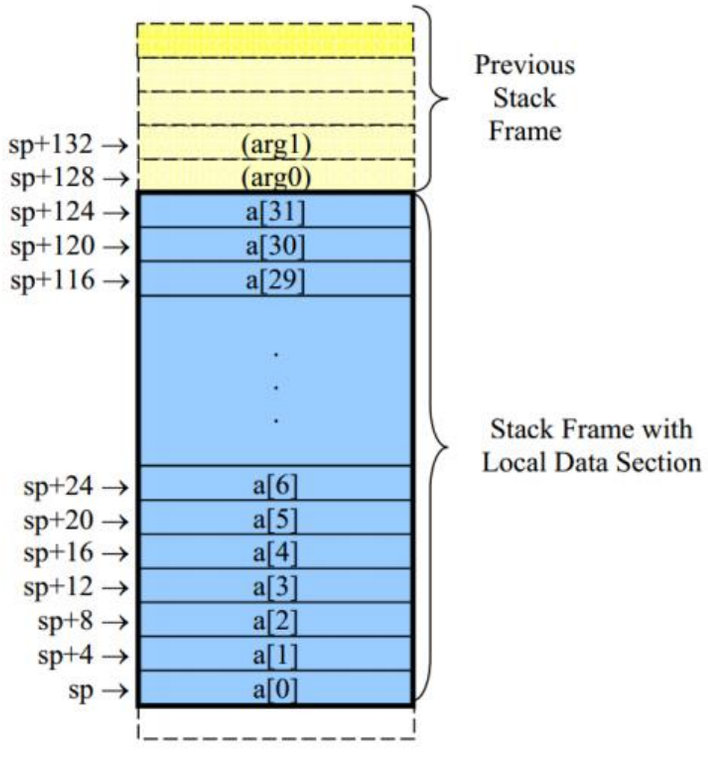
6. Control Instructions

Instruction	Effect
beq Rs, Rt, label	if (Rs == Rt) PC ← label
bne Rs, Rt, label	if (Rs != Rt) PC ← label
bltz Rs, label	if (Rs < 0) PC ← label
blez Rs, label	if (Rs <= 0) PC ← label
bgtz Rs, label	if (Rs > 0) PC ← label
bgez Rs, label	if (Rs >= 0) PC ← label
j jlabel	PC ← jlabel
jr Rs	PC ← Rs
jal jlabel	$ra ← PC+4, PC ← jlabel
jalr Rs	$ra ← PC+4, PC ← Rs

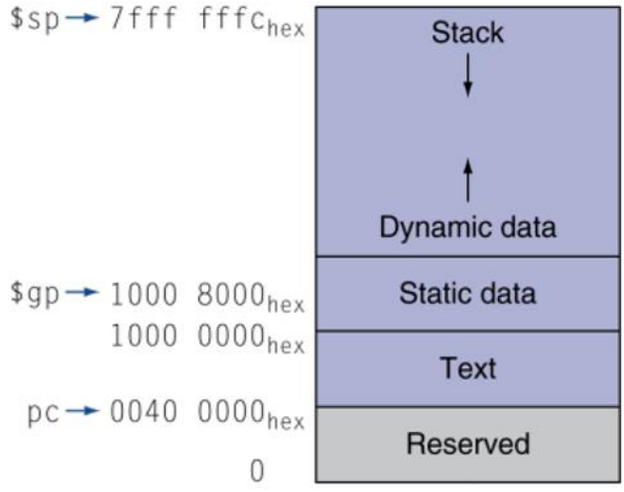
7. Stack



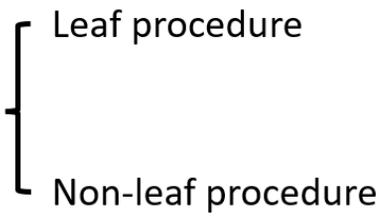

```
Proc  A

        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```



8. Memory Layout



9. Procedure call

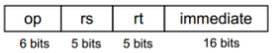
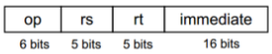
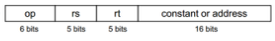
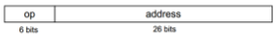


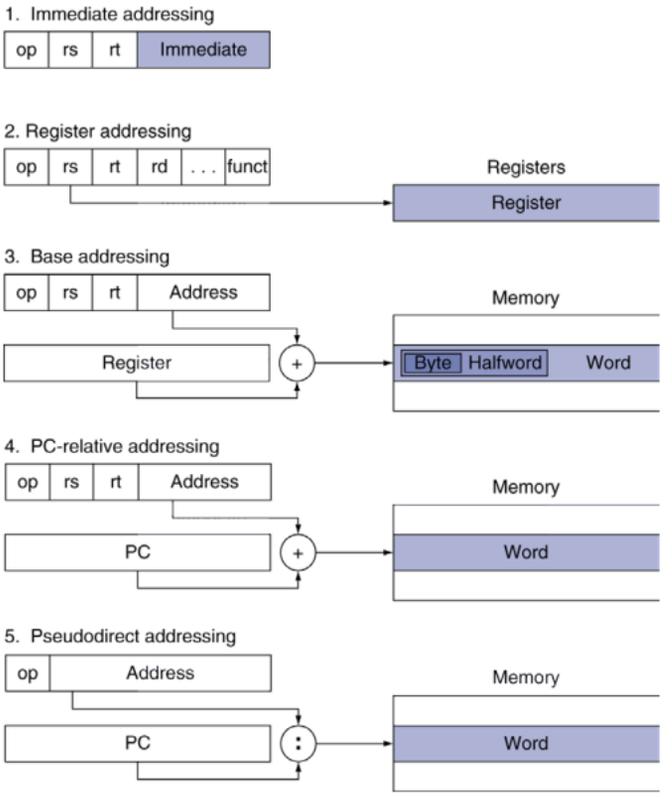
Caller saved: Temp registers $t0-$t9, $ra, $a0-$a3
Callee saved: $s0-$s7

For nested call, caller needs to save on the stack:
(1) Its return address;
(2) Any arguments and temporaries needed after the call.

Restore from the stack after the call.

10. MIPS addressing mode

Immediate addressing	Example: addi $s0, $s1, 2	
Register addressing	Example: add $s0, $s1, $s2	
Base/Displacement addressing	Example: lw $s0, 0(s1)	
Branch addressing (PC-relative addressing)	Example: bne $s0, $s1, EXIT	
Jump addressing (Pseudo-direct addressing)	Example: jal FACT	



11. Assembler and Linker

Role of assembler:

(1) Convert pseudo-instruction into actual hardware instructions

(2) Convert assembly instructions into machine instructions

Role of linker:

(1) Place code and data modules symbolically in memory.

(2) Determine the addresses of data and instruction labels.

(3) Patch both the internal and external references.

题型：

1. MIPS 与 C 的相互转化
2. MIPS 与机器码的相互转化
3. 执行一段 MIPS 以后寄存器的值是多少？（结合条件、循环、程序调用）
4. 一段 MIPS 代码被执行后运行了多少条指令？（通常结合循环与程序调用）
5. 数组的寻址（栈、lw\sw 的用法、地址与数值的区别）
6. 有符号数、无符号数的操作，是否 overflow 的判断

Chapter 3

7. Addition and Subtraction

Subtraction: Add negation of second operand

Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

Overflow if result out of range:

1. Addition

(1) Adding +ve and -ve operands, no overflow
(2) Adding two +ve operands: Overflow if result sign is 1
(3) Adding two -ve operands: Overflow if result sign is 0

2. Subtraction

(4) Subtracting two +ve or two -ve operands, no overflow
(5) Subtracting +ve from -ve operand: Overflow if result sign is 0
(6) Subtracting -ve from +ve operand: Overflow if result sign is 1

Dealing with overflow:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Note: addiu: "u" means it doesn't generate overflow exception, but the immediate can be a signed number

8. Saturation arithmetic

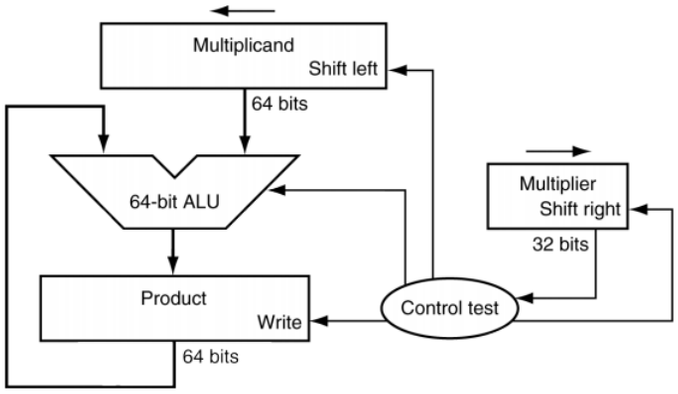
Saturation arithmetic is a version of arithmetic in which all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value.

For example, if the valid range of values is from 0 to 100, the following operations produce the following values:

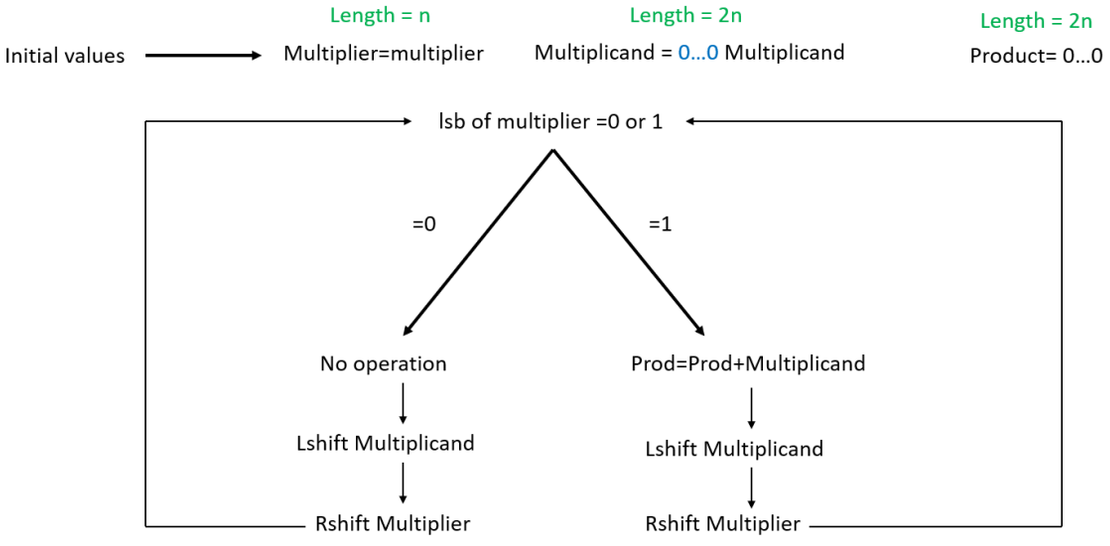
60 + 30 = 90
60 + 43 = 100
(60 + 43) − (75 + 75) = 0
10 × 11 = 100
99 × 99 = 100
30 × (5 − 1) = 100

$$30 \times 5 - 30 \times 1 = 70$$

9. Multiplication

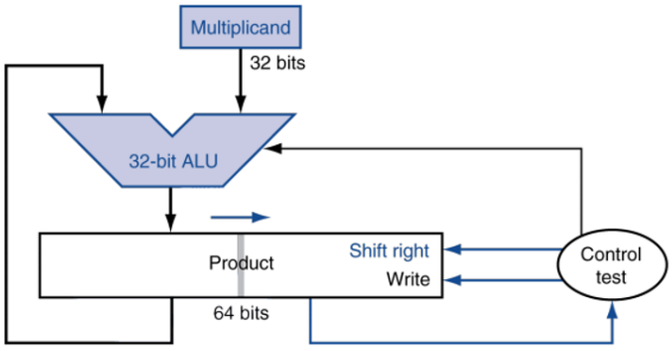


In every step:
- ✓ multiplicand is shifted
- ✓ next bit of multiplier is examined (also a shifting step)
- ✓ if this bit is 1, shifted multiplicand is added to the product

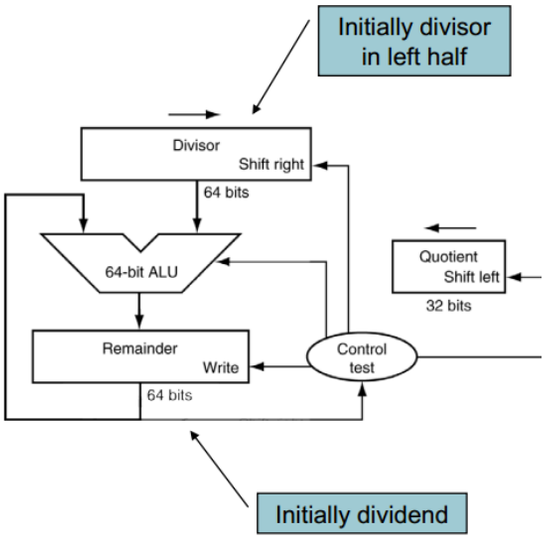


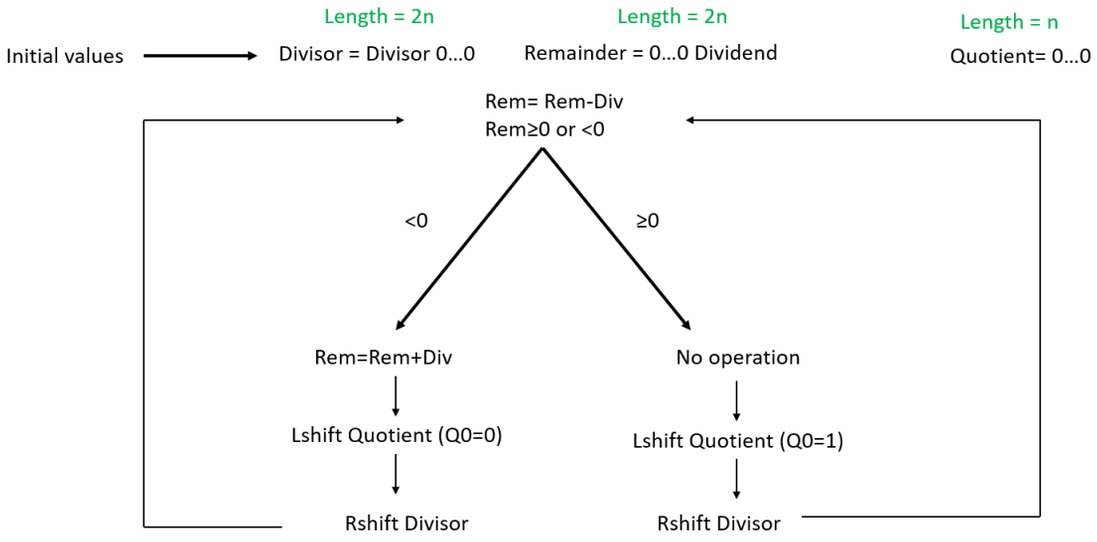
Iteration number = n

Iteration	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



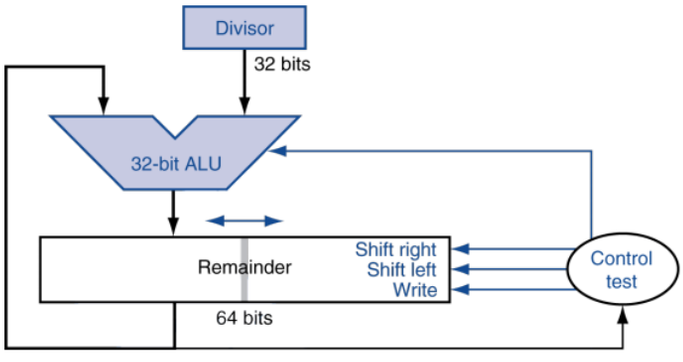
10. Division





Iteration number = n+1

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



11. Represent Floating Point

$\pm 1.xxxxxxx_2 \times 2^{yyyy}$

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent (yyyy+Bias)	Fraction (xxxx)

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Normalize significant: $1.0 \leq |\text{significant}| < 2.0$

- Single precision (32-bit)
- Double precision (64-bit)

Single: Bias = 127;
Double: Bias = 1023

Example:

$$-0.875$$

$0.875 \times 2 = 1.75$ …… 1

$0.75 \times 2 = 1.5$ …… 1

$0.5 \times 2 = 1$ …… 1

$$-0.111 \times 2^0 = -1.11 \times 2^{-1}$$

$$-0.875 = (-1)^1 \times 1.11_2 \times 2^{-1}$$

S=1

Fraction = $1100 \ldots 00_2$

Exponent = -1 + Bias

- Single: –1 + 127 = 126 = 01111110_2
- Double: –1 + 1023 = 1022 = 01111111110_2

Single: 10111111011000…00

Double: 10111111111011000…00

12. Floating Point Range

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 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
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

13. Floating Point Precision

- Relative precision
  - all fraction bits are significant
  - $\Delta A/|A| = 2^{-23} \times 2^{\text{exponent}} / |1 \times 2^{\text{exponent}}| = 2^{-23}$
  - 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

14. Rounding

Guard bit: 1st bit, Round bit: 2nd bit, Sticky bit: OR of remaining bits

Round condition

IEEE 754 has four rounding modes: always round up (toward +∞), always round down (toward -∞), truncate, and round to nearest even.

The final mode determines what to do if the number is exactly halfway in between.

e.g.

Directed roundings [edit]

- **Round toward 0** – directed rounding towards zero (also known as *truncation*).
- **Round toward +∞** – directed rounding towards positive infinity (also known as *rounding up* or *ceiling*).
- **Round toward −∞** – directed rounding towards negative infinity (also known as *rounding down* or *floor*).

Example of rounding to integers using the IEEE 754 rules

Mode	Example value			
	+11.5	**+12.5**	**−11.5**	**−12.5**
to nearest, ties to even	+12.0	+12.0	−12.0	−12.0
to nearest, ties away from zero	+12.0	+13.0	−12.0	−13.0
toward 0	+11.0	+12.0	−11.0	−12.0
toward +∞	+12.0	+13.0	−11.0	−12.0
toward −∞	+11.0	+12.0	−12.0	−13.0

题型：

1. 加减法 overflow 判断
2. 乘法器计算步骤（2 种）
3. 除法器计算步骤（2 种）
4. 浮点数表示法、范围、精度与加减运算