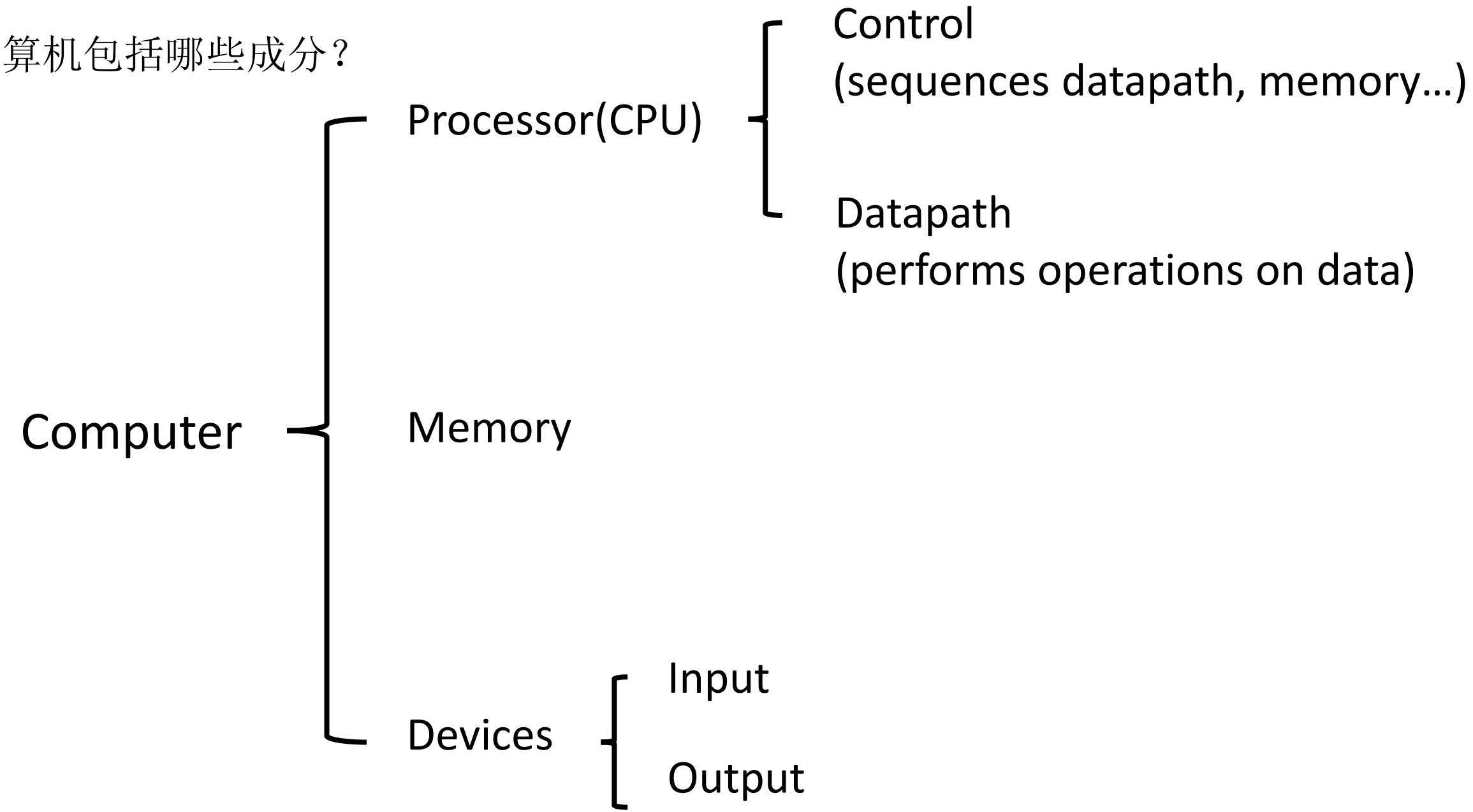
计算机组成原理

Chapter 1

复习

第一个知识点：

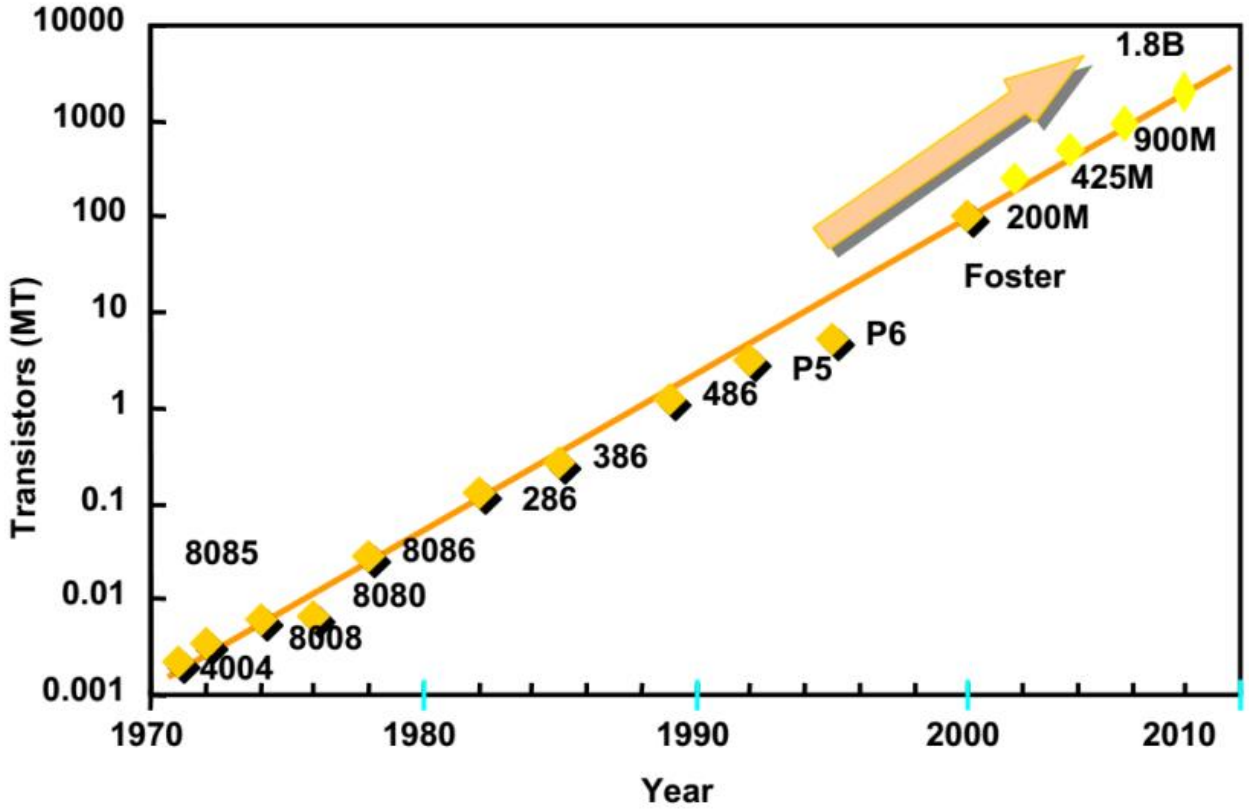
计算机包括哪些成分？



第二个知识点：

Moore’s Law

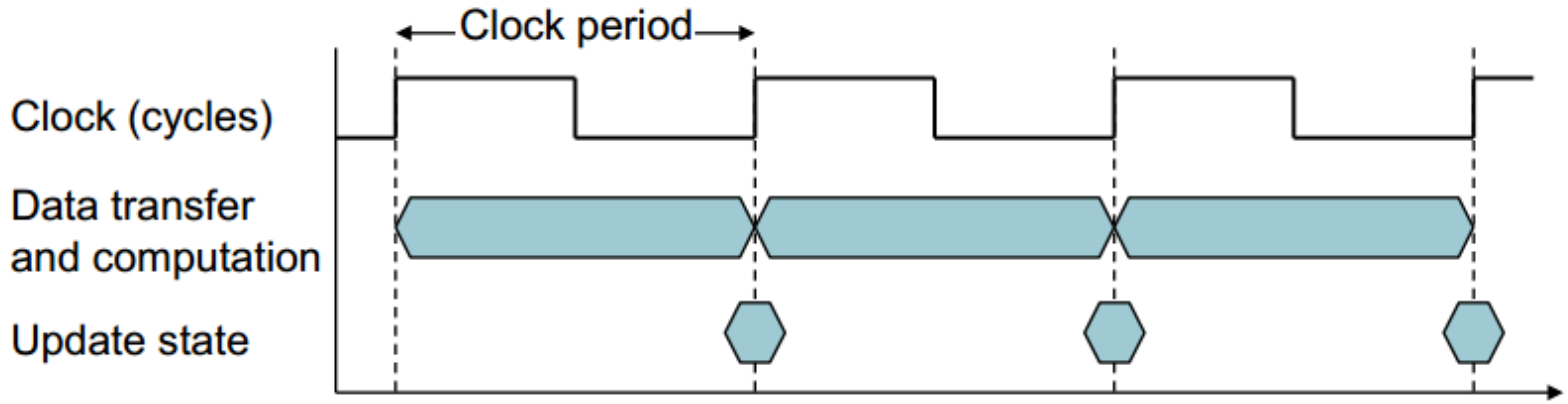
The number of transistors that can be integrated on a die would double every 18 to 24 months .



第三个知识点：

CPU clocking

Operation of digital hardware governed by a constant-rate clock



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

$$= \frac{\text{No. of Clock Cycles}}{\text{Clock Rate}}$$

第四个知识点：

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

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

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

第五个知识点：

Dynamic Power

- In CMOS IC technology

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

  - The power wall
    - We can’t reduce voltage further
    - We can’t remove more heat

第六个知识点：

Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

第七个知识点：

Amdahl’s Law

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl’s Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play

改进后的执行时间 = 受改进影响的执行时间/改进量 + 不受影响的执行时间

1.6 Consider two different implementations of the same instruction set architecture. The instructions can be divided into four classes according to their CPI (class A, B, C, and D). P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3, and P2 with a clock rate of 3 GHz and CPIs of 2, 2, 2, and 2.

Given a program with a dynamic instruction count of 1.0E6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which implementation is faster?

a. What is the global CPI for each implementation?

b. Find the clock cycles required in both cases.

a. Global CPI= clock cycles/count of instructions

$= \sum$ (Number of instructions $\times$ CPI)/ count of instructions

b. Clock cycles=$\sum$ (Number of instructions $\times$ CPI)

1.8 The Pentium 4 Prescott processor, released in 2004, had a clock rate of 3.6 GHz and voltage of 1.25 V. Assume that, on average, it consumed 10 W of static power and 90 W of dynamic power.

The Core i5 Ivy Bridge, released in 2012, had a clock rate of 3.4 GHz and voltage of 0.9 V. Assume that, on average, it consumed 30 W of static power and 40 W of dynamic power.

1.8.1 For each processor find the average capacitive loads.

1.8.2 Find the percentage of the total dissipated power comprised by static power and the ratio of static power to dynamic power for each technology.

1.8.1

Dy 1.8.2 *ncy switched*

So t Pentium 4:

the percentage of the total dissipated power comprised by static power

Pen =10/100=10%

Cor the ratio of static power to dynamic power

=10/90=11.11%

or: Core i5 Ivy Bridge:

the percentage of the total dissipated power comprised by static power *switched*

So t =30/70=42.86%

the ratio of static power to dynamic power

Pen =30/40=75%

Core i5 Ivy Bridge: $C=40/(0.9^2 \times 3.4 \times 10^9)= 1.5 \times 10^{-8}F$

1.8.3 If the total dissipated power is to be reduced by 10%, how much should the voltage be reduced to maintain the same leakage current? Note: power is defined as the product of voltage and current.

Analysis:

$$P_{total} \downarrow 10\% \longrightarrow \frac{P_{\text{new}}}{P_{old}} = 0.9 \qquad P_{total} = static\ power + dynamic\ power$$

$$static\ power = V \times I$$

$$dynamic\ power = \left(\frac{1}{2}\right) \text{C} \times V^2 \times f$$

Pentium 4: 1.18V

Without 1/2

Core i5 Ivy Bridge: 0.83V

1.15 When a program is adapted to run on multiple processors in a multiprocessor system, the execution time on each processor is comprised of computing time and the overhead time required for locked critical sections and/or to send data from one processor to another.

Assume a program requires t = 100 s of execution time on one processor. When run p processors, each processor requires t/p s, as well as an additional 4 s of overhead, irrespective of the number of processors. Compute the per-processor execution time for 2, 4, 8, 16, 32, 64, and 128 processors. For each case, list the corresponding speedup relative to a single processor and the ratio between actual speedup versus ideal speedup (speedup if there was no overhead).

Solution:

p=1, execution time per processor = 100 s.

For p>1, execution time per processor =100/p s

while the total time = execution time per processor + overhead = 100/p +4 s

Therefore, when p=2, 4, 8, 16, 32, 64, and 128

processors	Exec.time/ processor	Time w/overhead	Speedup	Actual speedup/ ideal speedup
1	100			
2	50	54	100/54=1.85	1.85/2=0.93
4	25	29	100/29=3.45	3.45/4=0.86
8	12.5	16.5	100/16.5=6.06	6.06/8=0.76
16	6.25	10.25	100/10.25=9.76	9.76/16=0.61
32	3.125	7.125	100/7.125=14.03	14.03/32=0.44
64	1.5625	5.5625	100/5.5625=17.98	17.98/64=0.41
128	0.78125	4.78125	100/4.78125=20.91	20.91/128=0.16

计算机组成原理

Chapter 2

复习

A

第一个知识点：

MIPS design principle

1. Simplicity favors regularity

2. Smaller is faster

3. Make the common case fast

4. Good design demands good compromises

第二个知识点：

Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word

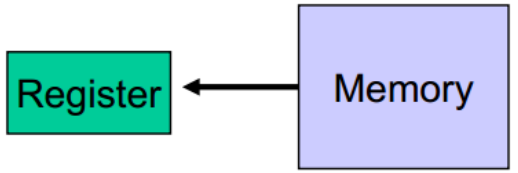
32个寄存器的作用…

第三个知识点：

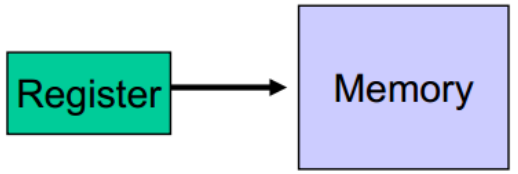
Memory Operands

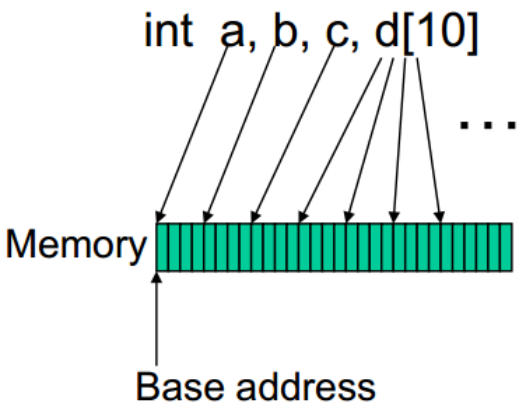
- Values must be fetched from memory before (add and sub) instructions can operate on them

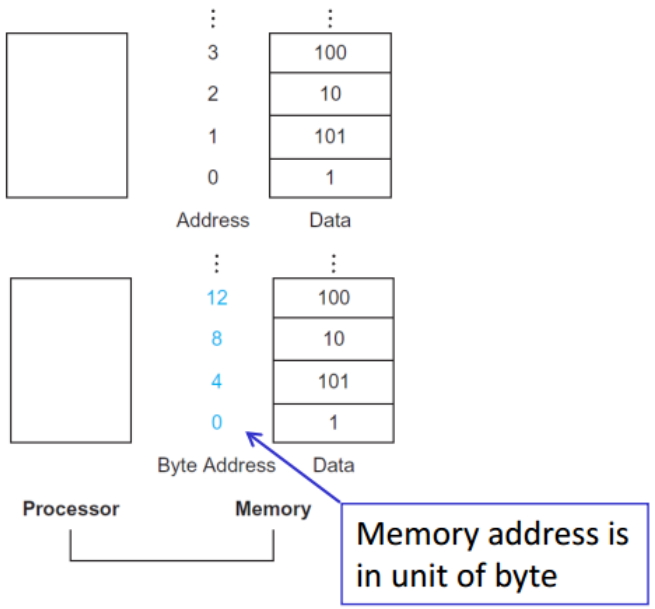
Load word
lw $t0, memory-address

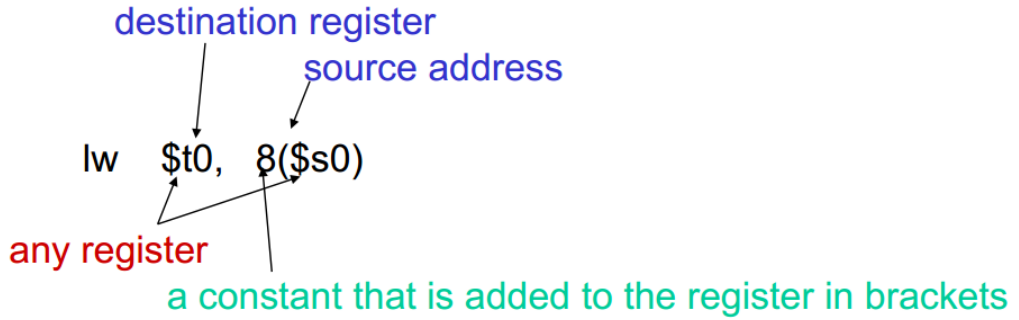


Store word
sw $t0, memory-address









第四个知识点：

Numeric Representations

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)

  0x 23 or 23_{hex}

  0-15 (decimal) → 0-9, a-f (hex)

Sign Extension:
正数前补0
负数前补1

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 $+2^{n-1} - 1$

Complement and add 1

- Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

第五个知识点：

Instruction Formats

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

第五个知识点：

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.

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{1b'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 (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)

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC	FOR-MAT	OPERATION	OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr (4)	11/8/1/--
Branch On FP False	bc1f	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*
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 (31–26) | fmt (25–21) | ft (20–16) | fs (15–11) | fd (10–6) | funct (5–0)
FI: opcode (31–26) | fmt (25–21) | ft (20–16) | immediate (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

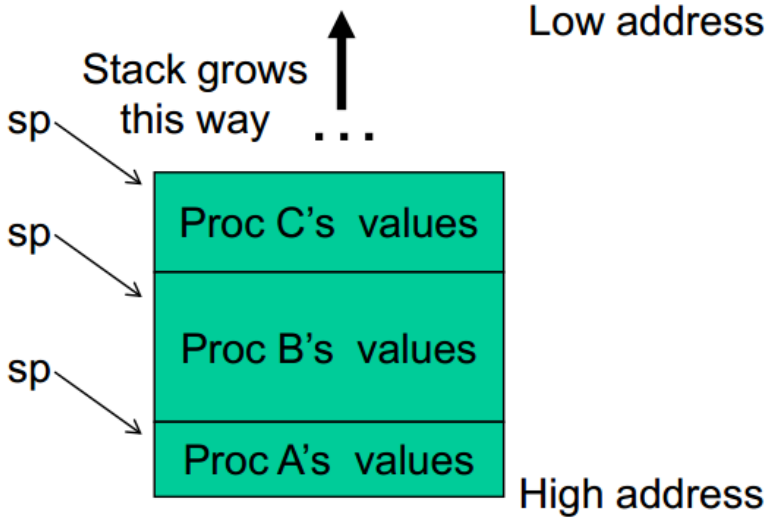
第六个知识点：

Control Instructions

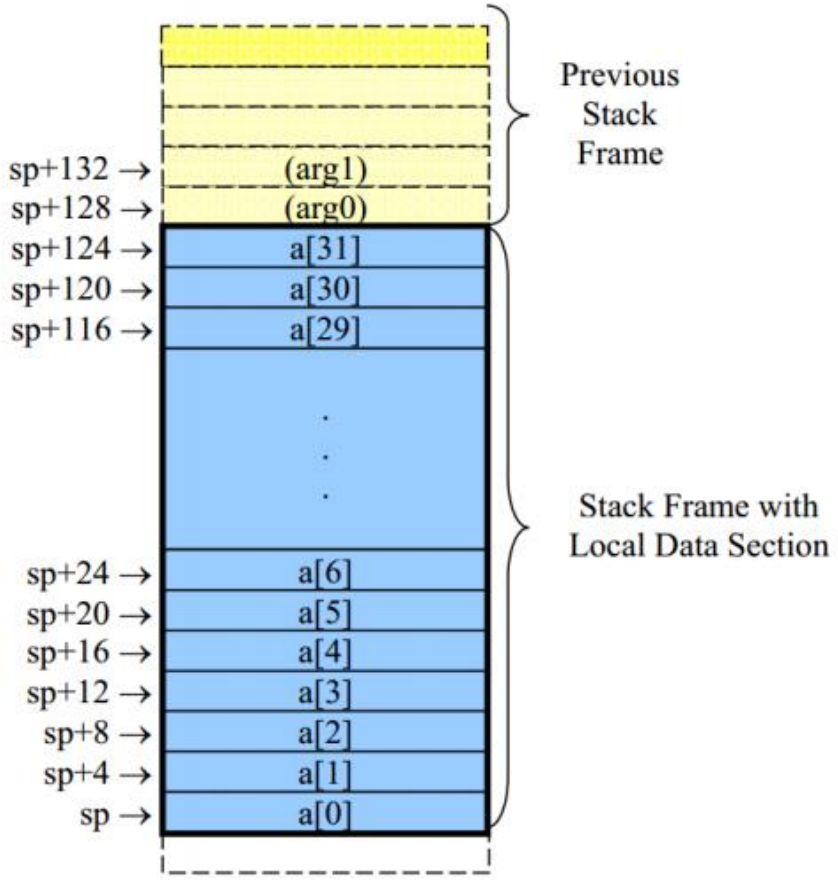
Instruction	Effect
beq Rs, Rt, label	if (Rs == Rt) PC ← label
bne Rs, Rt, label	if (Rs != Rt) PC ← label
bltz Rs, label	if (Rs < 0) PC ← label
blez Rs, label	if (Rs <= 0) PC ← label
bgtz Rs, label	if (Rs > 0) PC ← label
bgez Rs, label	if (Rs >= 0) PC ← label
j jlabel	PC ← jlabel
jr Rs	PC ← Rs
jal jlabel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

第七个知识点：

Stack



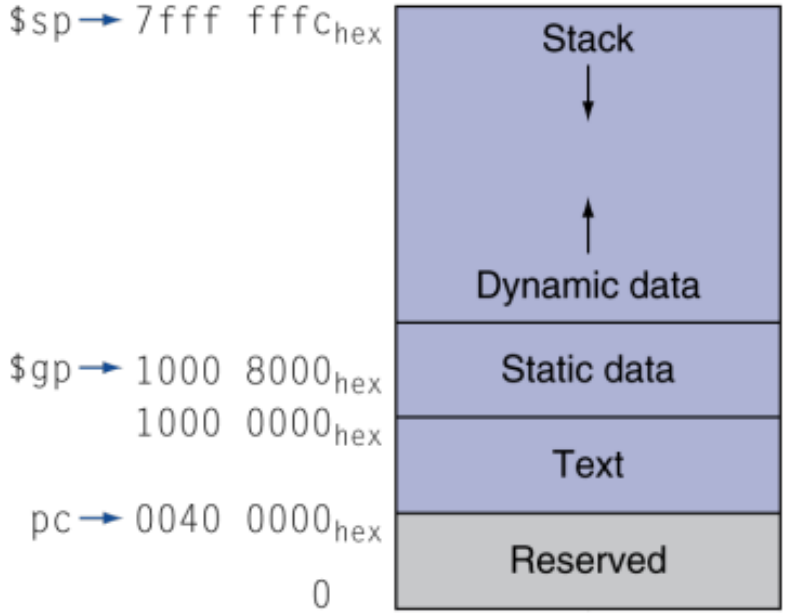

```
Proc  A

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



第八个知识点：

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



2.1 For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, and i are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.

$$f = g + (h - 5)$$

addi f, h, -5

add f, f, g

2.3 For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

$$B[8] = A[i - j]$$

```
sub $t0, $s3, $s4
sll $t0, $t0, 2
add $t0, $t0, $s6
lw $t1, 0($t0)
sw $t1, 32($s7)
```

2.6 The table below shows 32-bit values of an array stored in memory.

Address	Data
24	2
28	4
32	3
36	6
40	1

2.6.1 For the memory locations in the table above, write C code to sort the data from lowest to highest, placing the lowest value in the smallest memory location shown in the figure. Assume that the data shown represents the C variable called *Array*, which is an array of type *int*, and that the first number in the array shown is the first element in the array. Assume that this particular machine is a byte-addressable machine and a word consists of four bytes.

2.6.2 For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of *Array* is stored in register $s6.

Address	Data
24	2
28	4
32	3
36	6
40	1

```
temp = Array [0];
temp2 = Array [1];
Array [0] = Array [4];
Array [1] = temp;
Array [4] = Array [3];
Array [3] = temp2;
```

```
lw $t0, 0($s6)
lw $t1, 4($s6)
lw $t2, 16($s6)
sw $t2, 0($s6)
sw $t0, 4($s6)
lw $t0, 12($s6)
sw $t0, 16($s6)
sw $t1, 12($s6)
```

2.12 Assume that registers $s0 and $s1 hold the values 0x80000000 and 0xD0000000, respectively.

2.12.1 What is the value of $t0 for the following assembly code?

```
add $t0, $s0, $s1
```

2.12.2 Is the result in $t0 the desired result, or has there been overflow?

2.12.3 For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?

```
sub $t0, $s0, $s1
```

2.12.4 Is the result in $t0 the desired result, or has there been overflow?

2.12.5 For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

2.12.6 Is the result in $t0 the desired result, or has there been overflow?

0x 80000000 = 1000 0000 0000 0000 0000 0000 0000 0000 → Negative number

\+ 0x D0000000 = 1101 0000 0000 0000 0000 0000 0000 0000 → Negative number

---

0101 0000 0000 0000 0000 0000 0000 0000 = 0x 50000000 → Positive number

Overflow!

Overflow is usually when the sign of the result doesn't make sense compared to the operands. 0x80000000 and 0xD0000000 are both negative numbers. When you add two negative numbers you should get a negative number back. But instead you get 0x50000000, which is in fact a positive number. So yes, an overflow has occurred.

0x 80000000 = 1000 0000 0000 0000 0000 0000 0000 0000 → Negative number

− 0x D0000000 = 1101 0000 0000 0000 0000 0000 0000 0000 → Negative number

---

1011 0000 0000 0000 0000 0000 0000 0000 = 0x B0000000 → Negative number

No Overflow!

0x 80000000 = 1000 0000 0000 0000 0000 0000 0000 0000

\+ 0x D0000000 = 1101 0000 0000 0000 0000 0000 0000 0000

0101 0000 0000 0000 0000 0000 0000 0000

\+ 1000 0000 0000 0000 0000 0000 0000 0000

Overflow!

1101 0000 0000 0000 0000 0000 0000 0000=0x D0000000

2.16 Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

op=0, rs=3, rt=2, rd=3, shamt=0, funct=34

Type: r-type

funct = 34= 32+2=22_{hex}

Assembly language instruction: sub $v1, $v1, $v0

Binary representation:

op	rs	rt	rd	shamt	funct
000000	00011	00010	00011	00000	100010

0x00621822

计算机组成原理

Chapter 2
复习
B

第九个知识点：

Procedure call

Leaf procedure

Non-leaf procedure

- Caller saved: Temp registers $t0-$t9 (the callee won't bother saving these, so save them if you care), $ra (it's about to get over-written), $a0-$a3 (so you can put in new arguments)
- Callee saved: $s0-$s7 (these typically contain "valuable" data)

- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

第十个知识点：

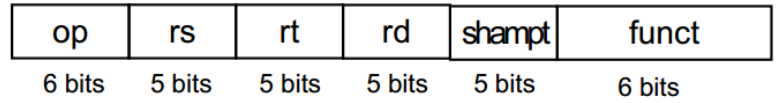
MIPS addressing mode

Immediate addressing

Example: addi $s0, $s1, 2

Register addressing

Example: add $s0, $s1, $s2

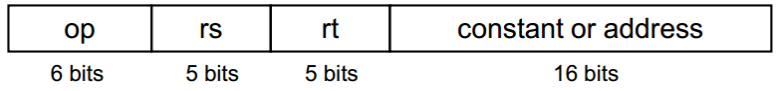


Base/Displacement addressing

Example: lw $s0, 0(s1)

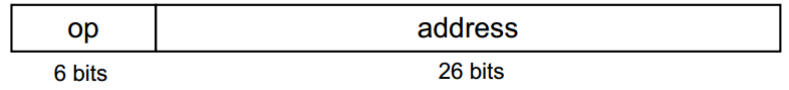
Branch addressing (PC-relative addressing)

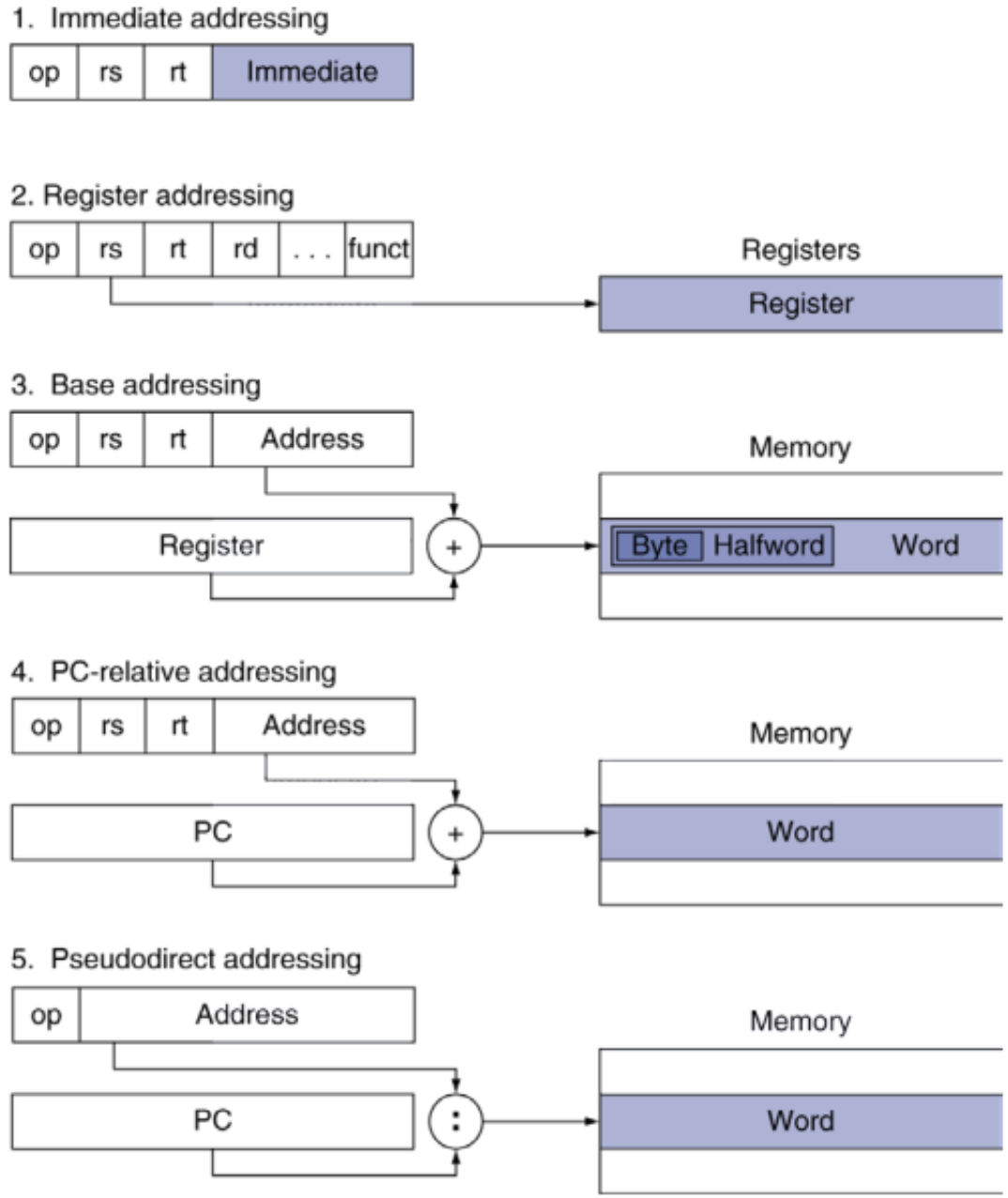
Example: bne $s0, $s1, EXIT



Jump addressing (Pseudo-direct addressing)

Example: jal FACT




1. Immediate addressing
op
rs
rt
Immediate
2. Register addressing
op
rs
rt
rd
. . .
funct
Registers
Register
3. Base addressing
op
rs
rt
Address
Memory
Register
+
Byte
Halfword
Word
4. PC-relative addressing
op
rs
rt
Address
Memory
PC
+
Word
5. Pseudodirect addressing
op
Address
Memory
PC
:
Word

第十一个知识点：

Assembler and Linker

Role of assembler:
1. Convert pseudo-instruction into actual hardware instructions
2. Convert assembly instructions into machine instructions

Role of linker:

- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory
- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

2.19 Assume the following register contents:

$$\$t0 = 0xAAAAAAAA, \qquad \$t1 = 0x12345678$$

2.19.1 For the register values shown above, what is the value of $t2 for the following sequence of instructions?

$$\text{sll } \$t2, \$t0, 4$$
$$\text{or } \$t2, \$t2, \$t1$$

2.19.2 For the register values shown above, what is the value of $t2 for the following sequence of instructions?

$$\text{sll } \$t2, \$t0, 4$$
$$\text{andi } \$t2, \$t2, -1$$

2.19.3 For the register values shown above, what is the value of $t2 for the following sequence of instructions?

$$\text{srl } \$t2, \$t0, 3$$
$$\text{andi } \$t2, \$t2, 0xFFEF$$

$$0\text{xAAAAAAAA} = 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010_2$$
$$0\text{x12345678} = \ \ 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000_2$$

2.19.1

sll \$t2, \$t0, 4 → $\$t2 = \ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 0000_2$

$0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000_2$

or \$t2, \$t2, \$t1 →

$\$t2 = \ 1011\ 1010\ 1011\ 1110\ 1111\ 1110\ 1111\ 1000_2$

$= 0xBABEFEF8$

$$0xAAAAAAAA = 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010_2$$
$$0x12345678 = \quad 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000_2$$

2.19.2

$$\text{andi } \$t2, \$t2, -1$$

$$\begin{aligned} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 \text{取反} + 1 \\ -1_{10} = & \ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110 \quad + 1 \\ & \ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 \end{aligned}$$

$$\text{sll } \$t2, \$t0, 4 \rightarrow \quad \$t2 = \ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 0000_2$$

$$\text{andi } \$t2, \$t2, -1 \rightarrow$$
$$\$t2 = 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 0000$$
$$= \ 0xAAAAAAA0$$

$$0xAAAAAAAA = 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010_2$$
$$0x12345678 = \quad 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000_2$$

2.19.3

$$0xFFEF = 0000\ 0000\ 0000\ 0000\ 1111\ 1111\ 1110\ 1111_2$$

srl \$t2, \$t0, 3 → \$t2 = $0001\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101_2$

andi \$t2, \$t2, 0xFFEF →

\$t2 = 0000 0000 0000 0000 0101 0101 0100 0101

= 0x00005545

2.26 Consider the following MIPS loop:

```
LOOP: slt $t2, $0, $t1
      beq $t2, $0, DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j LOOP
DONE:
```

2.26.1 Assume that the register $t1 is initialized to the value 10. What is the value in register $s2 assuming $s2 is initially zero?

2.26.2 For each of the loops above, write the equivalent C code routine. Assume that the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively.

2.26.3 For the loops written in MIPS assembly above, assume that the register $t1 is initialized to the value N. How many MIPS instructions are executed?

2.26.1 The loop will be executed 10 times, so the result is $2 \times 10 = 20$

2.26.2

```
i=10
do {
   B += 2;
   i = i -1;
} while (i > 0)
```

2.26.3

```
LOOP: slt $t2, $0, $t1
      beq $t2, $0, DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j LOOP
DONE:
```

$5 \times N + 2$

2.31 Implement the following C code in MIPS assembly. What is the total number of MIPS instructions needed to execute the function?

```
int fib(int n){
  if (n==0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n−1) + fib(n−2);
```

```
int fib(int n){
  if (n==0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n−1) + fib(n−2);
}
```

bgt $a0, $zero, ELSE1
add $v0, $zero, $zero

addi $t0, $t0, 1
bne $t0, $a0, ELSE2
add $v0, $zero, $t0

addi $a0, $a0, -1
jal fib
add $s0, $zero, $v0

addi $a0, $a0, -1
jal fib
add $v0, $v0, $s0

需要存起来的寄存器？ → $ra, $s0, $a0

```
fib:   addi $sp, $sp, -12
       sw $ra, 8($sp)
       sw $s0, 4($sp)
       sw $a0, 0($sp)
       bgt $a0, $zero, ELSE1
       add $v0, $zero, $zero
       j rtn
ELSE1: addi $t0, $t0, 1
       bne $t0, $a0, ELSE2
       add $v0, $zero, $t0
       j rtn
ELSE2: addi $a0, $a0, -1
       jal fib
       add $s0, $zero, $v0
       addi $a0, $a0, -1
       jal fib
       add $v0, $v0, $s0
rtn:   lw $a0, 0($sp)
       lw $s0, 4($sp)
       lw $ra, 8($sp)
       addi $sp, $sp, 12
       jr $ra
```

```
fib:    addi $sp, $sp, -12
        sw $ra, 8($sp)
        sw $s0, 4($sp)
        sw $a0, 0($sp)
        bgt $a0, $zero, ELSE1
        add $v0, $zero, $zero
        j rtn
ELSE1:  addi $t0, $t0, 1
        bne $t0, $a0, ELSE2
        add $v0, $zero, $t0
        j rtn
ELSE2:  addi $a0, $a0, -1
        jal fib
        add $s0, $zero, $v0
        addi $a0, $a0, -1
        jal fib
        add $v0, $v0, $s0
rtn:    lw $a0, 0($sp)
        lw $s0, 4($sp)
        lw $ra, 8($sp)
        addi $sp, $sp, 12
        jr $ra
```

When N=0, 12 instructions, when N=1, 14 instructions.

观察$a0的变化：

N=2

$a0： 2 1 0 2

N=3：

$a0： 3 2 1 0 2 1 3

fib(2) fib(1)

N=4：

$a0： 4 3 2 1 0 2 1 3 2 1 0 2 4

fib(3) fib(2)

N=5

$a0： 5 4 3 2 1 0 2 1 3 2 1 0 2 4 3 2 1 0 2 1 3 5

fib(4) fib(3)

N=6

$a0： 6 5 4 3 2 1 0 2 1 3 2 1 0 2 4 3 2 1 0 2 1 3 5 4 3 2 1 0 2 1 3 2 1 0 2 4 6

fib(5) fib(4)

```
fib:    addi $sp, $sp, -12
        sw $ra, 8($sp)
        sw $s0, 4($sp)
        sw $a0, 0($sp)
        bgt $a0, $zero, ELSE1
        add $v0, $zero, $zero
        j rtn
ELSE1:  addi $t0, $t0, 1
        bne $t0, $a0, ELSE2
        add $v0, $zero, $t0
        j rtn
ELSE2:  addi $a0, $a0, -1
        jal fib
        add $s0, $zero, $v0
        addi $a0, $a0, -1
        jal fib
        add $v0, $v0, $s0
rtn:    lw $a0, 0($sp)
        lw $s0, 4($sp)
        lw $ra, 8($sp)
        addi $sp, $sp, 12
        jr $ra
```

When N=0, 12 instructions, when N=1, 14 instructions.

When N≥2, $f(N) = f(N-1) + f(N-2) + 18$ instructions

N=2: 44 instructions
N=3: 76 instructions
N=4: 138 instructions
N=5: 232 instructions
N=6: 388 instructions

$$f(N) = f(N-1) + f(N-2) + 18$$

2　二阶非齐次线性递推数列的通项公式

定理2　若二阶非齐次线性递推数列的递推关系为$a_{n+1}=pa_n+qa_{n-1}+A$，其中$p\neq 0, q\neq 0, A\neq 0$，则有：

1)　若$p+q=1$，则当$q=-1$时，$a_n=a_1+(n-1)(a_2-a_1)+\frac{1}{2}(n-1)(n-2)A$；当$q\neq -1$时，$a_n=a_1+(a_2-a_1-\frac{A}{1+q})\cdot\frac{1-(-q)^{n-1}}{1+q}+(n-1)\cdot\frac{A}{1-q}$.

2) 若$p+q\neq 1$，则当$p^2+4q=0$时，$a_n=(a_1+\lambda)\beta^{n-1}+(n-1)[a_2+\lambda-\beta(a_1+\lambda)]\beta^{n-2}-\lambda$，其中$\beta=\frac{p}{2}$，$\lambda=\frac{A}{p+q-1}$.

当$p^2+4q>0$时，则有

$$a_n=[a_1+\lambda-\frac{a_2+\lambda-\alpha(a_1+\lambda)}{\beta-\alpha}]\cdot\alpha^{n-1}+\frac{a_2+\lambda-\alpha(a_1+\lambda)}{\beta-\alpha}\cdot\beta^{n-1}-\lambda,$$

其中$\alpha=\frac{p-\sqrt{p^2+4q}}{2}$，$\beta=\frac{p+\sqrt{p^2+4q}}{2}$，$\lambda=\frac{A}{p+q-1}$.

当$p^2+4q<0$时，则有

$$a_n=[a_1+\lambda-\frac{a_2+\lambda-\alpha(a_1+\lambda)}{\beta-\alpha}]\cdot\alpha^{n-1}+\frac{a_2+\lambda-\alpha(a_1+\lambda)}{\beta-\alpha}\cdot\beta^{n-1}-\lambda,$$

其中$\alpha=\frac{p-i\sqrt{-p^2-4q}}{2}$，$\beta=\frac{p+i\sqrt{-p^2-4q}}{2}$，$\lambda=\frac{A}{p+q-1}$.

$$f(N) = \left(30-\frac{17+15\sqrt{5}}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^N + \left(\frac{17+15\sqrt{5}}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^N - 18$$

```
fib:    addi $sp, $sp, -12
        sw $ra, 8($sp)
        sw $s0, 4($sp)
        sw $a0, 0($sp)
        bgt $a0, $zero, ELSE1
        add $v0, $zero, $zero
        j rtn
ELSE1:  addi $t0, $t0, 1
        bne $t0, $a0, ELSE2
        add $v0, $zero, $t0
        j rtn
ELSE2:  addi $a0, $a0, -1
        jal fib
        add $s0, $zero, $v0
        addi $a0, $a0, -1
        jal fib
        add $v0, $v0, $s0
rtn:    lw $a0, 0($sp)
        lw $s0, 4($sp)
        lw $ra, 8($sp)
        addi $sp, $sp, 12
        jr $ra
```

When N=0, 12 instructions, when N=1, 14 instructions.

When N≥2, $f(N) = f(N-1) + f(N-2) + 18$ instructions

$$f(N) = \left(30 - \frac{17+15\sqrt{5}}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^N + \left(\frac{17+15\sqrt{5}}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^N - 18$$

So when N≥2,

$$\left(30 - \frac{17+15\sqrt{5}}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^N + \left(\frac{17+15\sqrt{5}}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^N - 18$$

instructions

计算机组成原理

Chapter 3

复习

第一个知识点：

Addition and subtraction

Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands
  - Overflow if result sign is 1
- Adding two –ve operands
  - Overflow if result sign is 0

Overflow if result out of range

- Subtracting two +ve or two –ve operands, no overflow
- Subtracting +ve from –ve operand
  - Overflow if result sign is 0
- 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

Example: 7 – 6 = 7 + (–6)

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

第二个知识点：

Saturation arithmetic

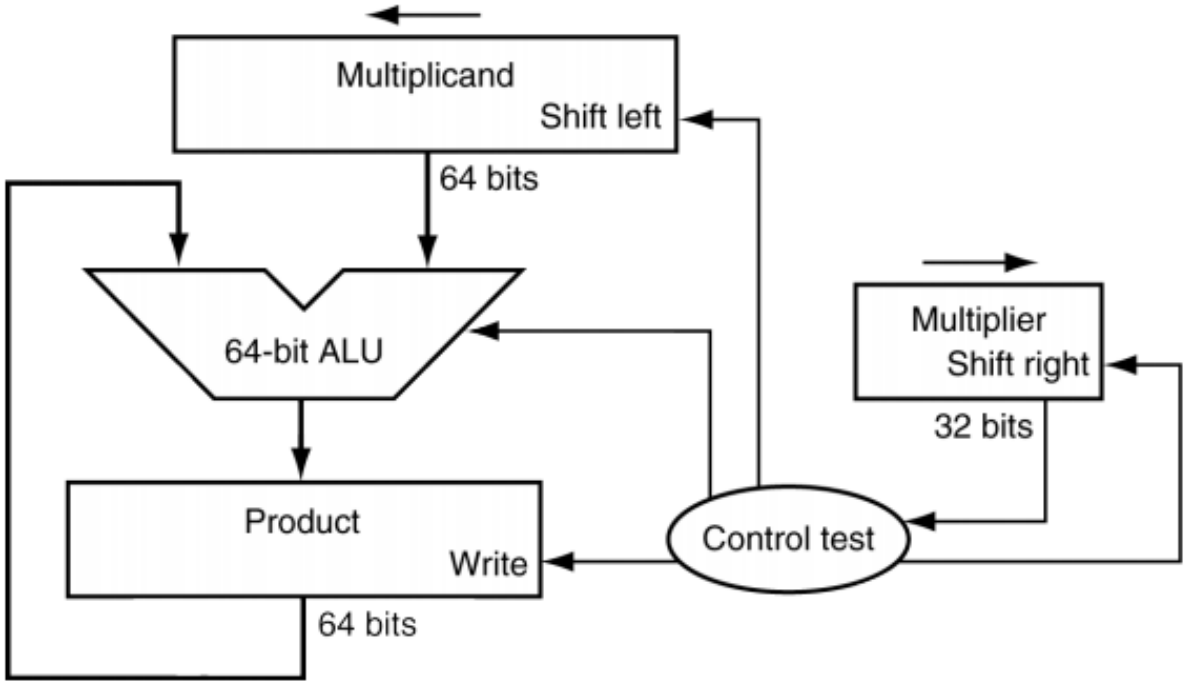
Saturation arithmetic is a version of arithmetic in which all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value.

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 × 5 − 30 × 1 = 70

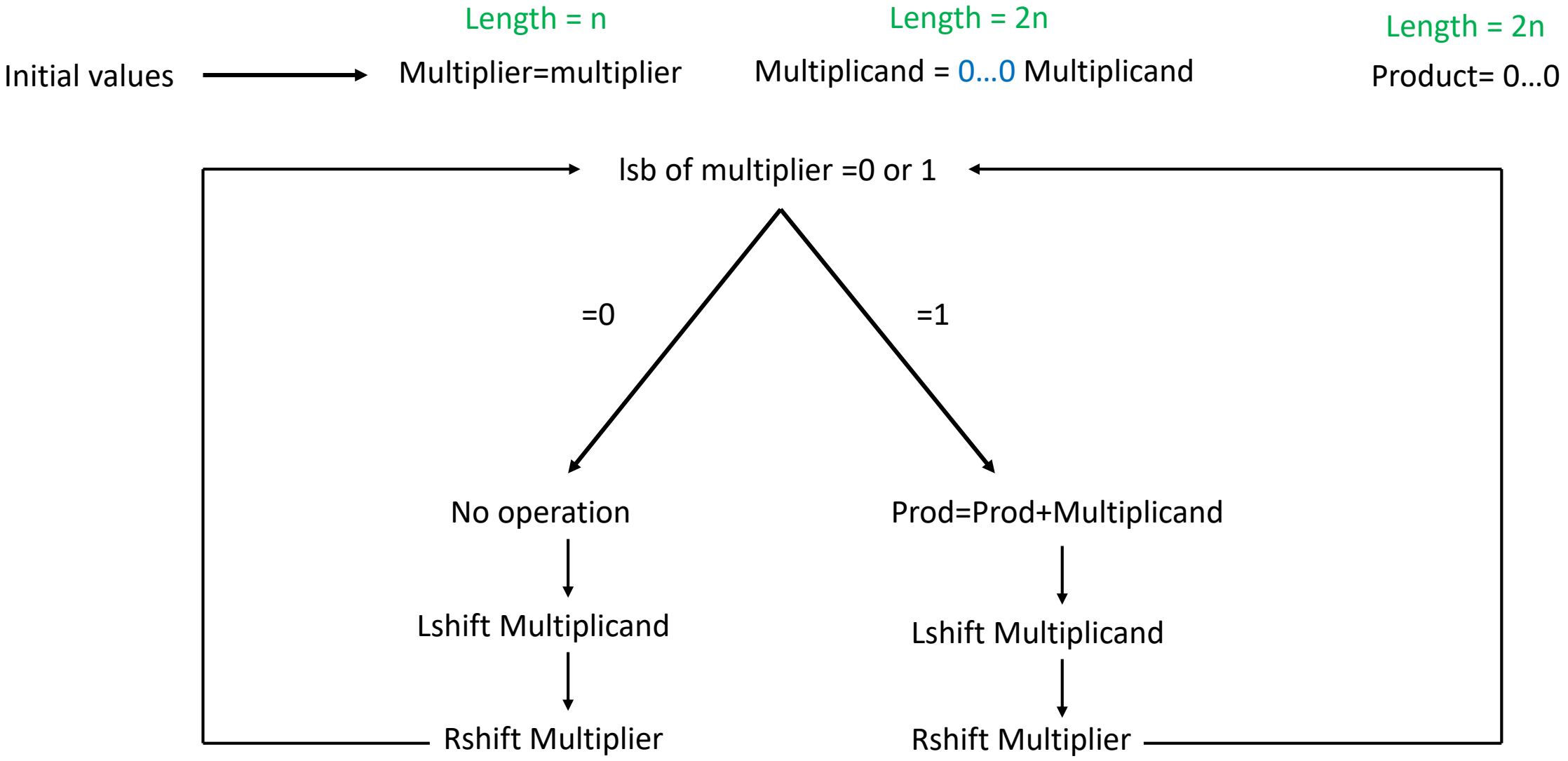
第三个知识点：

Multiplication



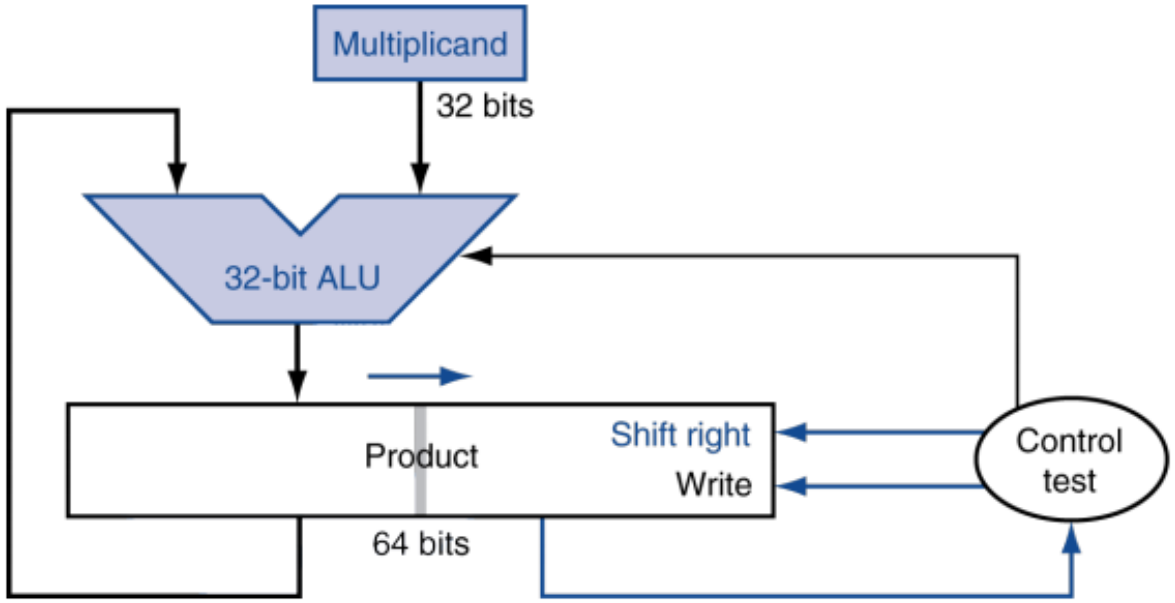
In every step:

- ✓ 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


Length = n
Length = 2n
Length = 2n
Initial values
Multiplier=multiplier
Multiplicand = 0…0 Multiplicand
Product= 0…0
lsb of multiplier =0 or 1
=0
=1
No operation
Prod=Prod+Multiplicand
Lshift Multiplicand
Lshift Multiplicand
Rshift Multiplier
Rshift Multiplier
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

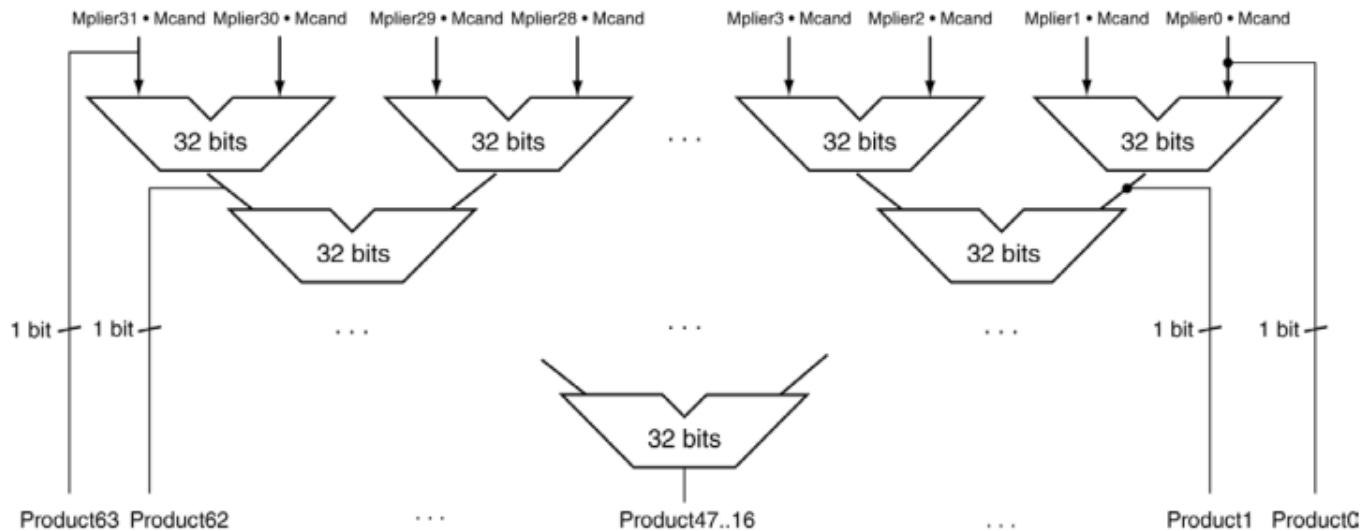
Optimized Multiplier



- ✓ 64-bit “Product” stores product and multiplier
- ✓ the sum keeps shifting right
- ✓ at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register
- ✓ 32-bit ALU and multiplicand are used instead of 64-bit ones.

Faster Multiplier

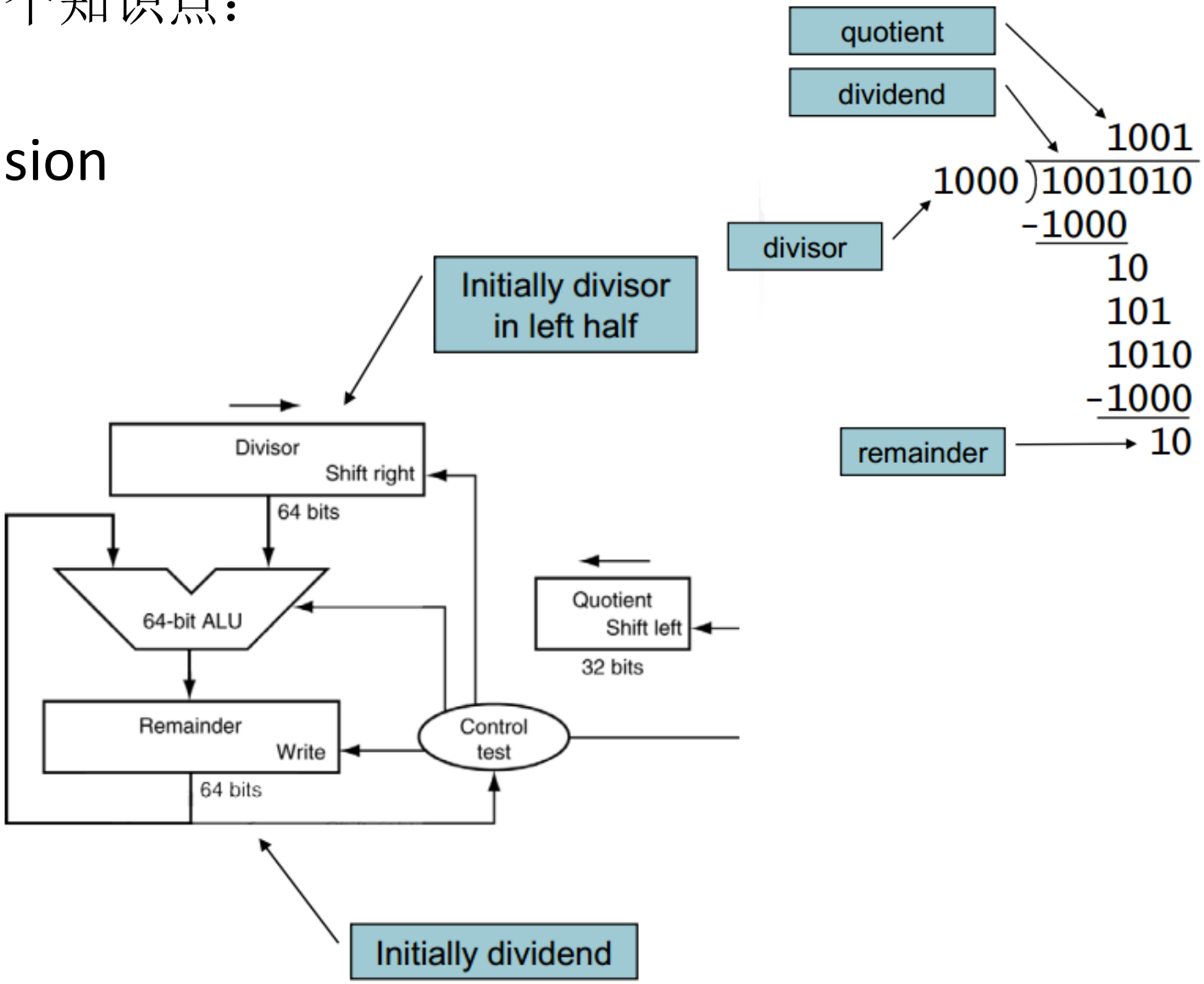
- Uses multiple pipelined adders
  - Cost/performance tradeoff



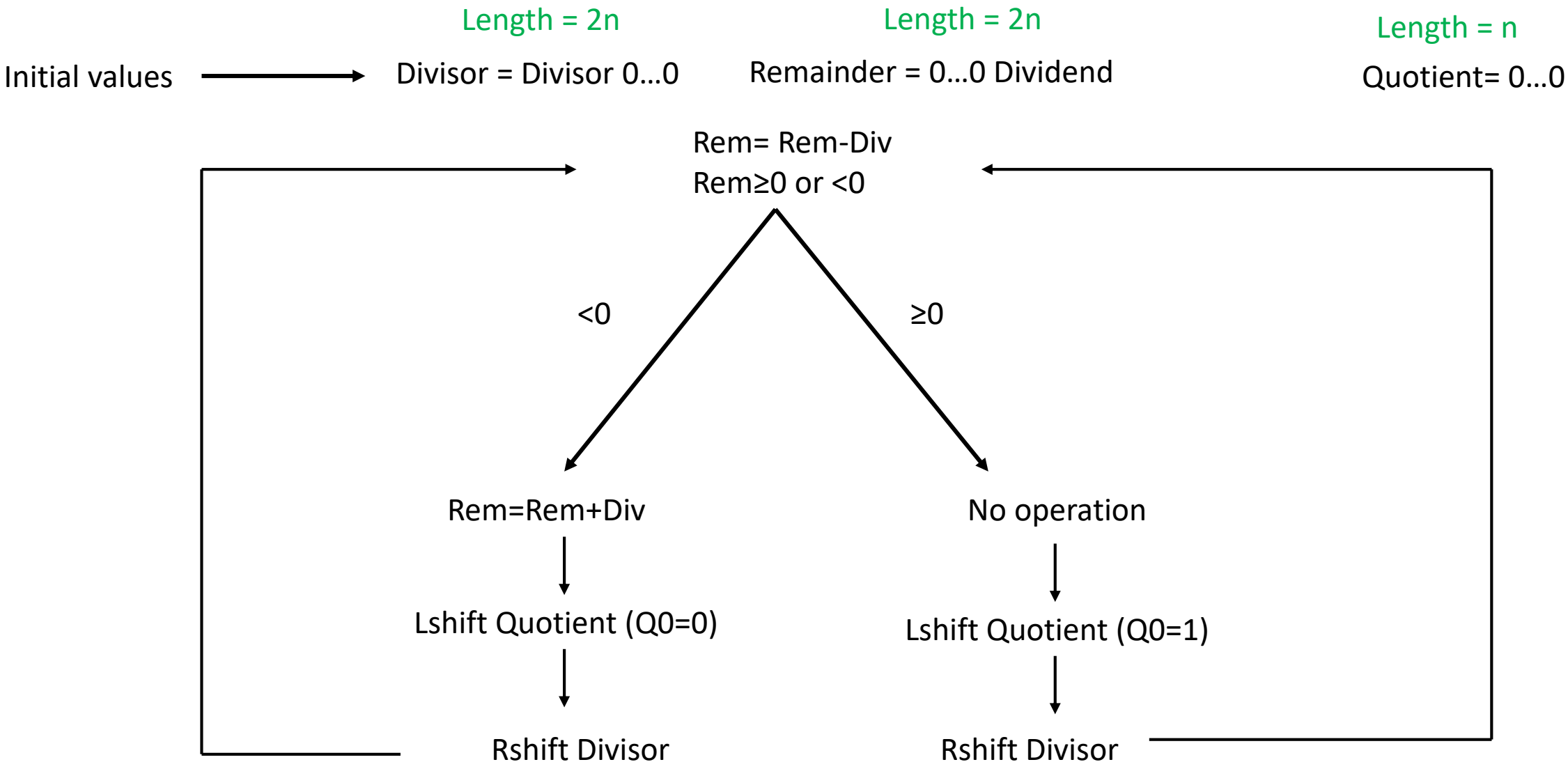
- Can be pipelined
  - Several multiplication performed in parallel

第四个知识点：

Division



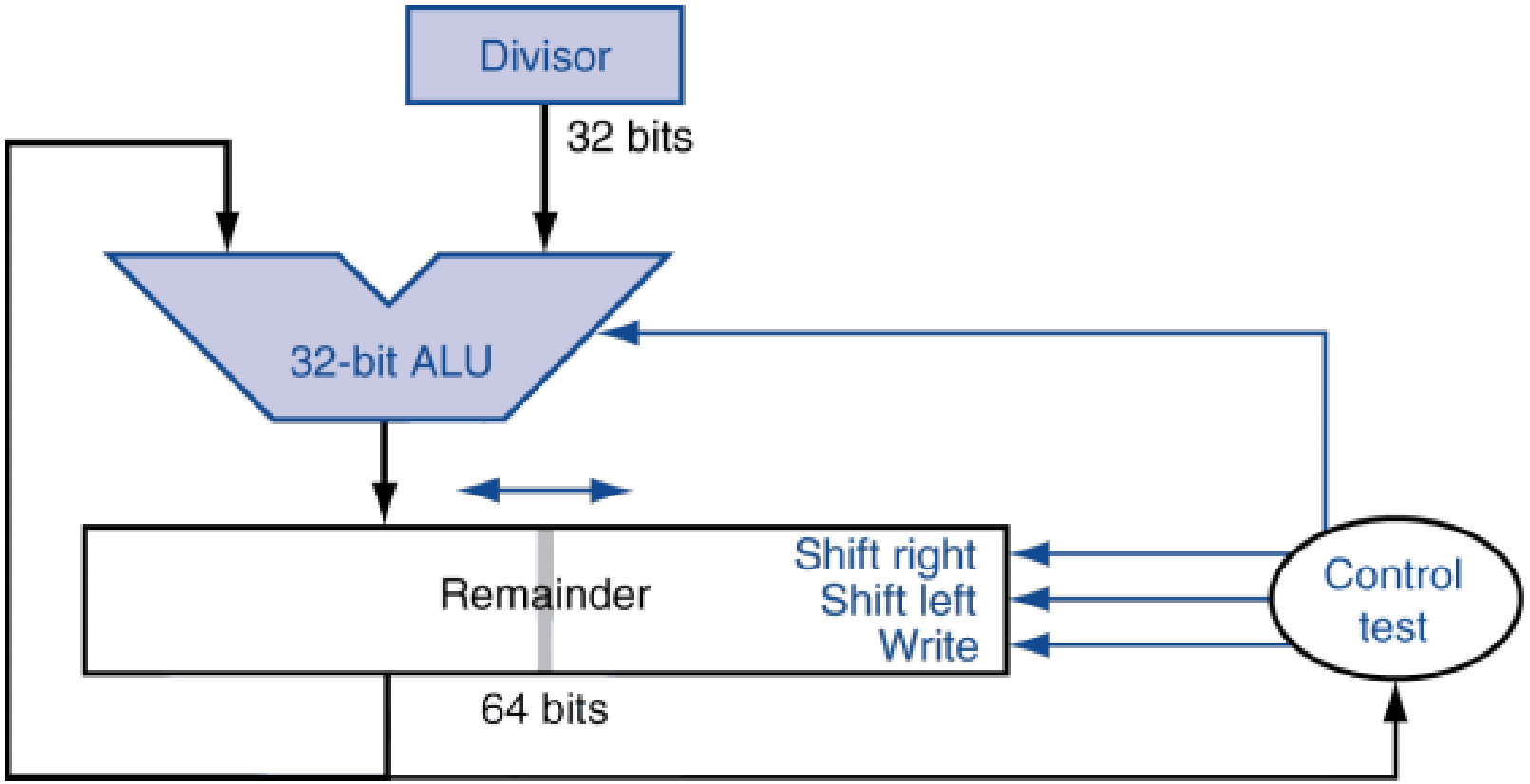
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using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



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

Optimized Divider

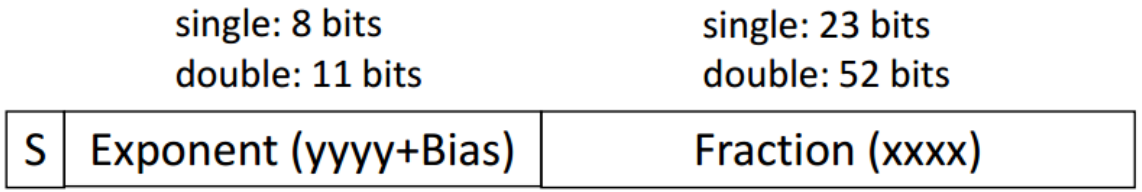


第五个知识点：

Represent Floating Point

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

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



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

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

Single: Bias = 127;

Double: Bias = 1023

$$-0.875$$

$$0.875 \times 2 = \ 1.75 \qquad \ldots\ldots \qquad 1$$

$$0.75 \times 2 = 1.5 \qquad \ldots\ldots \qquad 1$$

$$0.5 \times 2 = 1 \qquad \ldots\ldots \qquad 1$$

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

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

S=1

Fraction = $1100 \ldots 00_2$

Exponent = -1 + Bias

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

Single: 10111111011100…00

Double: 10111111111011100…00

第六个知识点：

Floating Point Range

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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

第七个知识点：

Floating-Point Precision

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

第八个知识点：

Floating-Point Instruction in MIPS

Separate FP registers

- 32 single-precision: $f0, $f1, … $f31
- Paired for double-precision: $f0/$f1, $f2/$f3, …
  - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's

FP instructions operate only on FP registers

- Programs generally don't do integer ops on FP data, or vice versa
- More registers with minimal code-size impact

FP load and store instructions

- `lwc1`, `ldc1`, `swc1`, `sdc1`
  - e.g., `ldc1 $f8, 32($sp)`

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, `div.s`
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, `lt`, `le`, ...)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

3.9 Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate 151 + 214 using saturating arithmetic. The result should be written in decimal. Show your work.

$$151 = 10010111_2 = -(01101001) = -105$$

$$214 = 11010110_2 = -(00101010) = -42$$

signed 8-bit decimal integers $\longrightarrow$ $00000000 \sim 01111111 \rightarrow 0 \sim 127$

$11111111 \sim 10000000 \rightarrow -1 \sim -128$

$$Range\ is: -128 \sim 127$$

$151 + 214 = -105 + (-42) = -147$

$-147\ is\ out\ of\ the\ range\ -128 \sim 127$

Using saturating arithmetic, the result is -128

3.10 Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate 151−214 using saturating arithmetic. The result should be written in decimal. Show your work.

$$151 = 10010111_2 = -(01101001) = -105$$

$$214 = 11010110_2 = -(00101010) = -42$$

signed 8-bit decimal integers ⟶ $00000000 \sim 01111111 \rightarrow 0 \sim 127$

$11111111 \sim 10000000 \rightarrow -1 \sim -128$

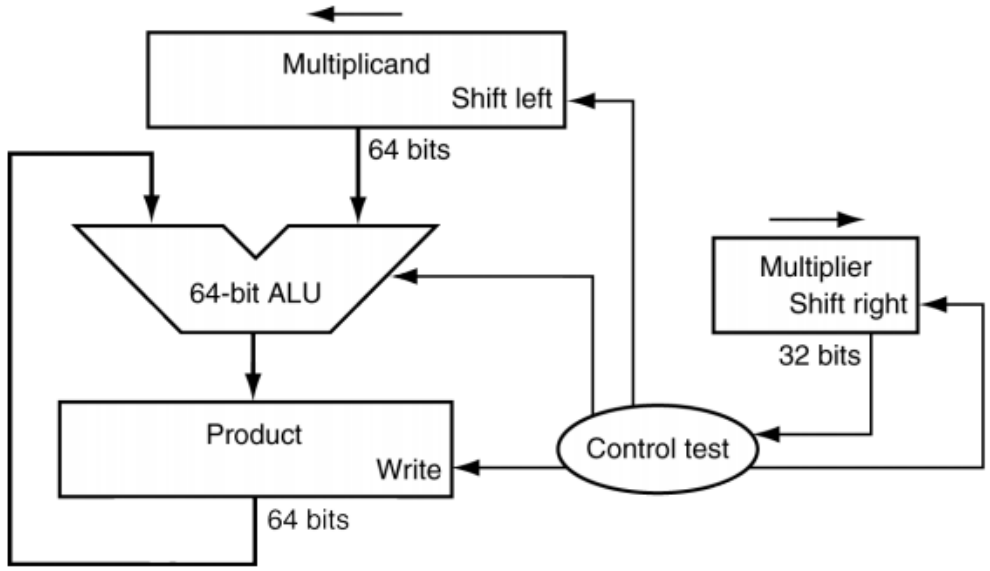
$Range\ is: -128 \sim 127$

$151 - 214 = -105 - (-42) = -63$

$-63\ is\ in\ the\ range\ -128 \sim 127$

Using saturating arithmetic, the result is -63

3.12 Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.

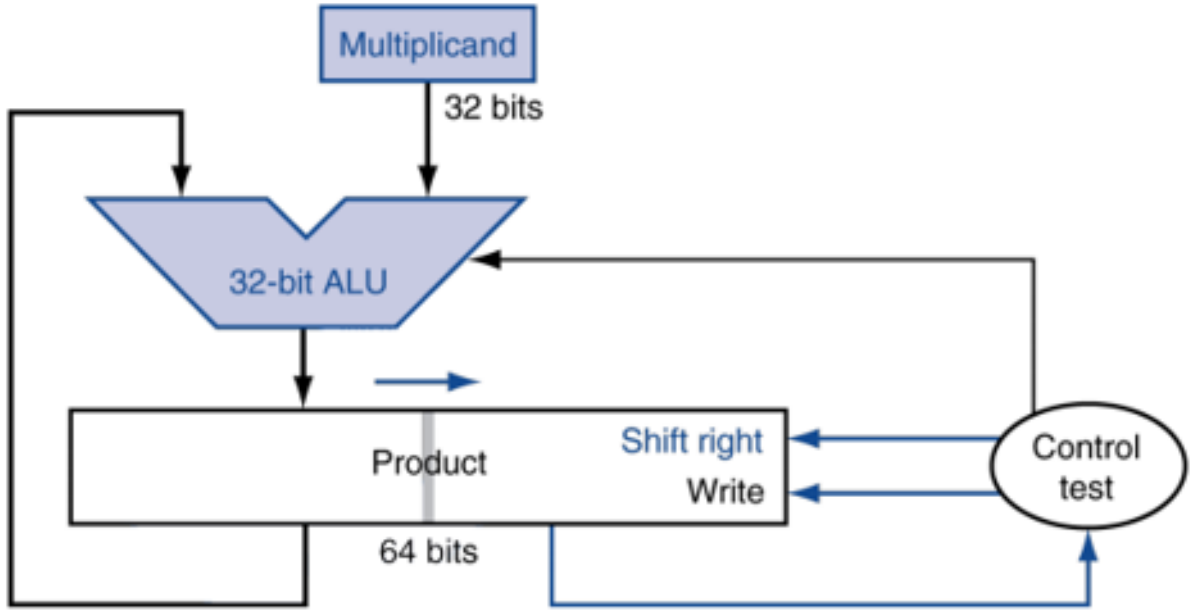


$62_{oct} = 110\ 010_2$

$12_{oct} = 001\ 010_2$

Step	Action	Multiplier	Multiplicand	Product
0	Initial Vals	001 010	000 000 110 010	000 000 000 000
1	lsb=0. no op	001 010	000 000 110 010	000 000 000 000
	Lshift Mcand	001 010	000 001 100 100	000 000 000 000
	Rshift Mplier	000 101	000 001 100 100	000 000 000 000
2	Prod = Prod + Mcand	000 101	000 001 100 100	000 001 100 100
	Lshift Mcand	000 101	000 011 001 000	000 001 100 100
	Rshift Mplier	000 010	000 011 001 000	000 001 100 100
3	lsb=0. no op	000 010	000 011 001 000	000 001 100 100
	Lshift Mcand	000 010	000 110 010 000	000 001 100 100
	Rshift Mplier	000 001	000 110 010 000	000 001 100 100
4	Prod = Prod + Mcand	000 001	000 110 010 000	000 111 110 100
	Lshift Mcand	000 001	001 100 100 000	000 111 110 100
	Rshift Mplier	000 000	001 100 100 000	000 111 110 100
5	lsb=0. no op	000 000	001 100 100 000	000 111 110 100
	Lshift Mcand	000 000	011 001 000 000	000 111 110 100
	Rshift Mplier	000 000	011 001 000 000	000 111 110 100
6	lsb=0. no op	000 000	011 001 000 000	000 111 110 100
	Lshift Mcand	000 000	110 010 000 000	000 111 110 100
	Rshift Mplier	000 000	110 010 000 000	000 111 110 100

3.13 Using a table similar to that shown in Figure 3.6, calculate the product of the hexadecimal unsigned 8-bit integers 62 and 12 using the hardware described in Figure 3.5. You should show the contents of each register on each step.



$62_{hex} = 0110\ 0010_2$

$12_{hex} = 0001\ 0010_2$

Step	Action	Multiplicand	Product/Multiplier
0	Initial Vals	0110 0010	0000 0000 0001 0010
1	lsb = 0, no op	0110 0010	0000 0000 0001 0010
	Rshift Product	0110 0010	`0000 0000 0000 1001
2	Prod = Prod +Mcand	0110 0010	0110 0010 0000 1001
	Rshift Product	0110 0010	0011 0001 0000 0100
3	lsb = 0, no op	0110 0010	0011 0001 0000 0100
	Rshift Product	0110 0010	0001 1000 1000 0010
4	lsb = 0, no op	0110 0010	0001 1000 1000 0010
	Rshift Product	0110 0010	0000 1100 0100 0001
5	Prod = Prod +Mcand	0110 0010	0110 1110 0100 0001
	Rshift Product	0110 0010	0011 0111 0010 0000
6	lsb = 0, no op	0110 0010	0011 0111 0010 0000
	Rshift Product	0110 0010	0001 1011 1001 0000
7	lsb = 0, no op	0110 0010	0001 1011 1001 0000
	Rshift Product	0110 0010	0000 1101 1100 1000
8	lsb = 0, no op	0110 0010	0000 1101 1100 1000
	Rshift Product	0110 0010	0000 0110 1110 0100

计算机组成原理

Chapter 4

复习

A

第一个知识点：

A basic MIPS Implementation

For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction.

第一个知识点：

A basic MIPS Implementation

- basic math (add, sub, and, or, slt)
- memory access (lw and sw)
- branch and jump instructions (beq and j)

  - We need memory
    - to store instructions
    - to store data
    - for now, let's make them separate units
  - We need registers, ALU, and a whole lot of control logic
  - CPU operations common to all instructions:
    - use the program counter (PC) to pull instruction out of instruction memory
    - read register values

第二个知识点：

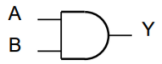
Clocking Methodology

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

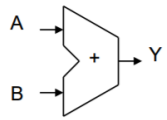
第二个知识点：

Combinational Elements

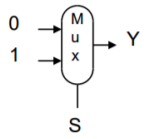
- And gate
  - Y = A & B



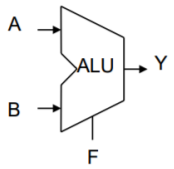
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? 1 : 0

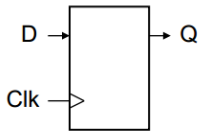


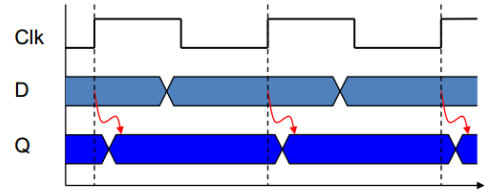
- Arithmetic/Logic Unit
  - Y = F(A, B)



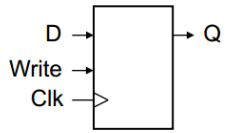
Sequential Elements

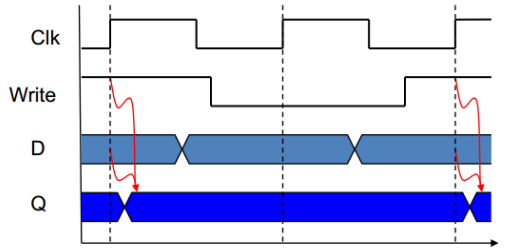
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





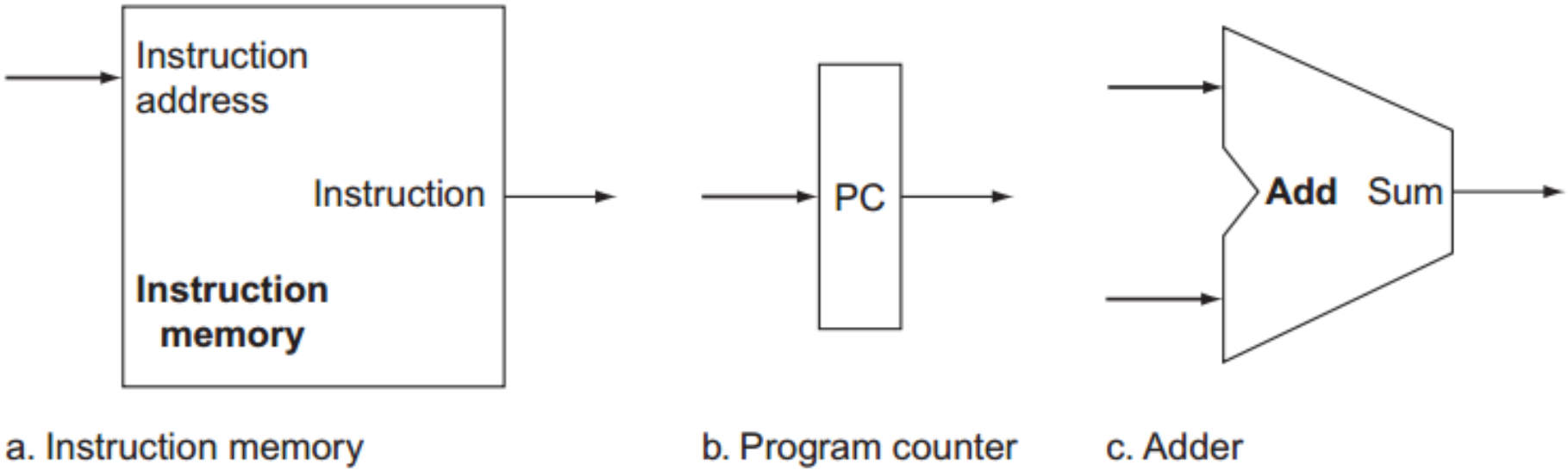
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later





第三个知识点：

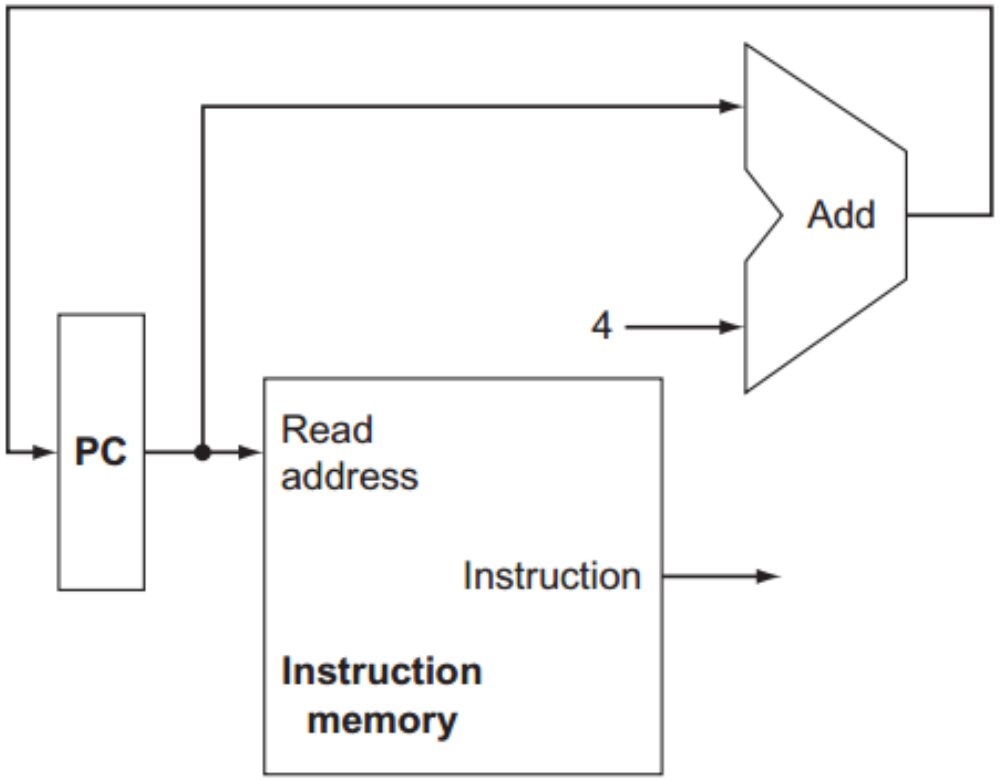
Datapath



Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

第三个知识点：

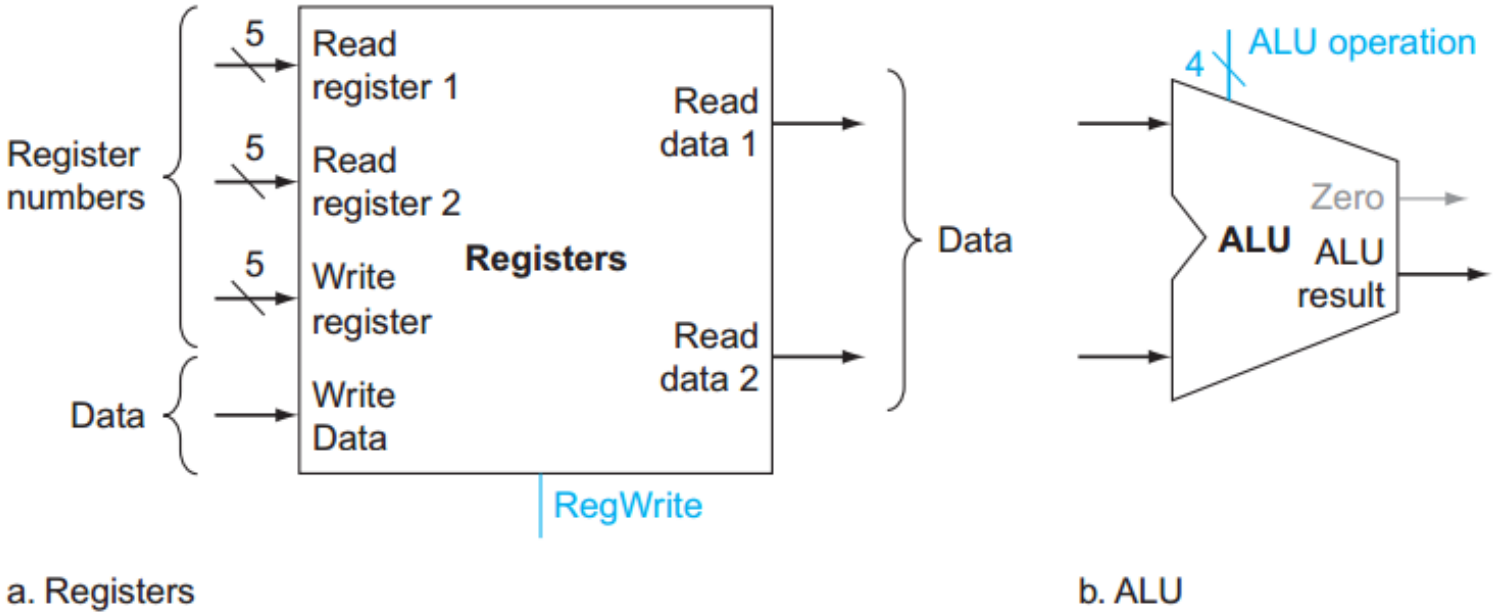
Datapath



A portion of the datapath used for fetching instructions and incrementing the program counter.

第三个知识点：

R Type

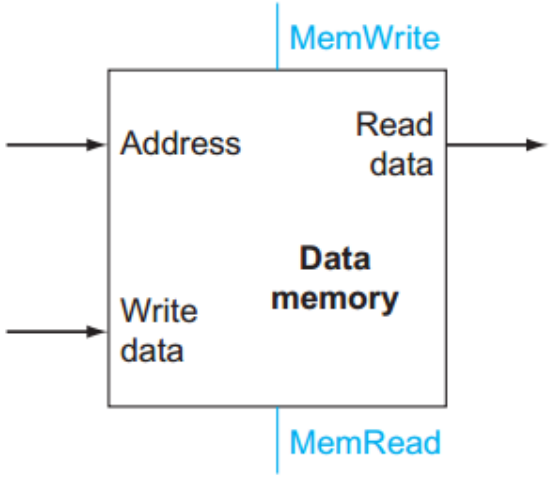


The two elements needed to implement R-format ALU operations are the register file and the ALU

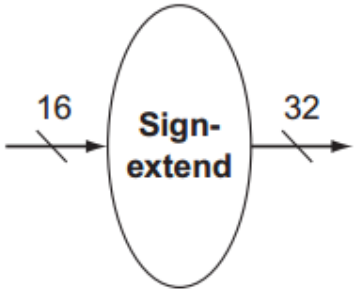
I Type

■ The instruction set architecture specifes that the base for the branch address calculation is the address of the instruction following the branch. Since we compute PC + 4 (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.

■ The architecture also states that the offset field is shifed left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of 4.

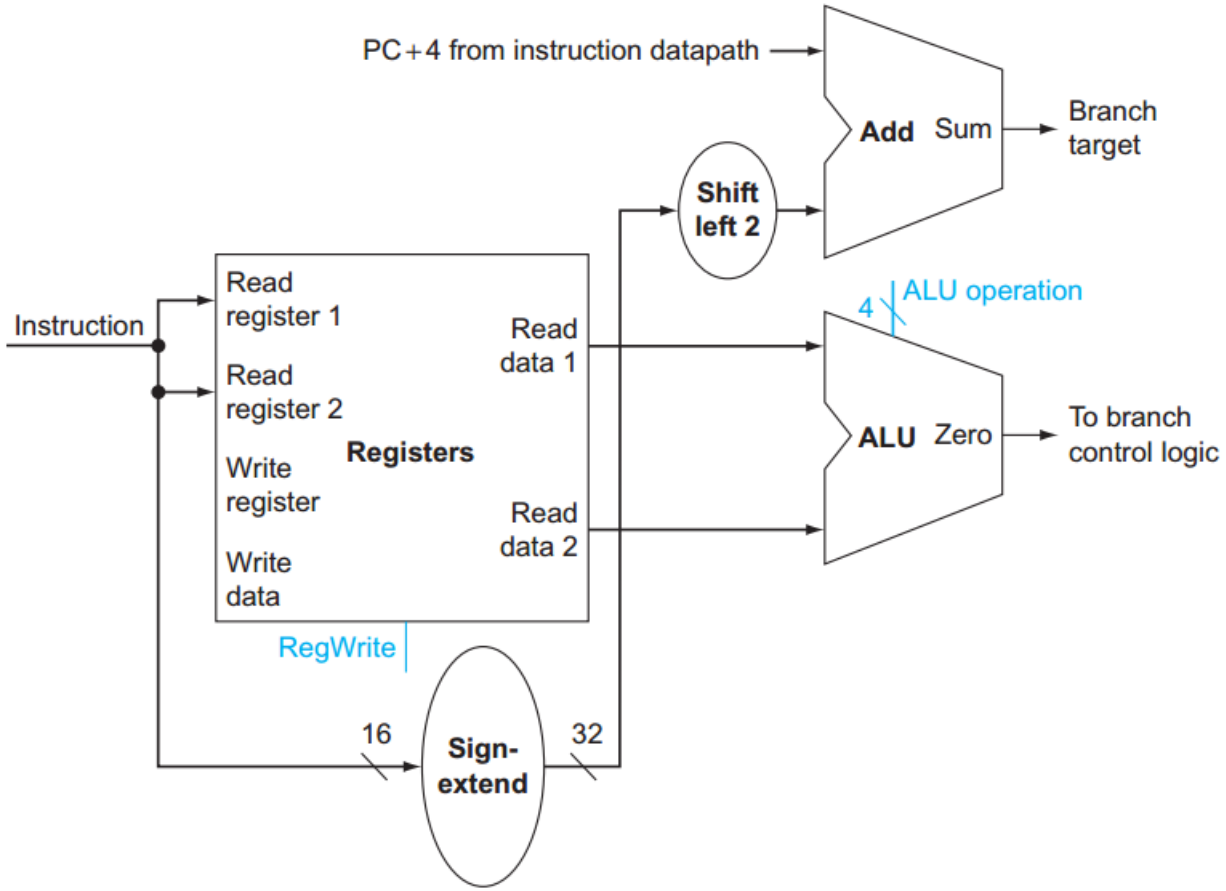


a. Data memory unit



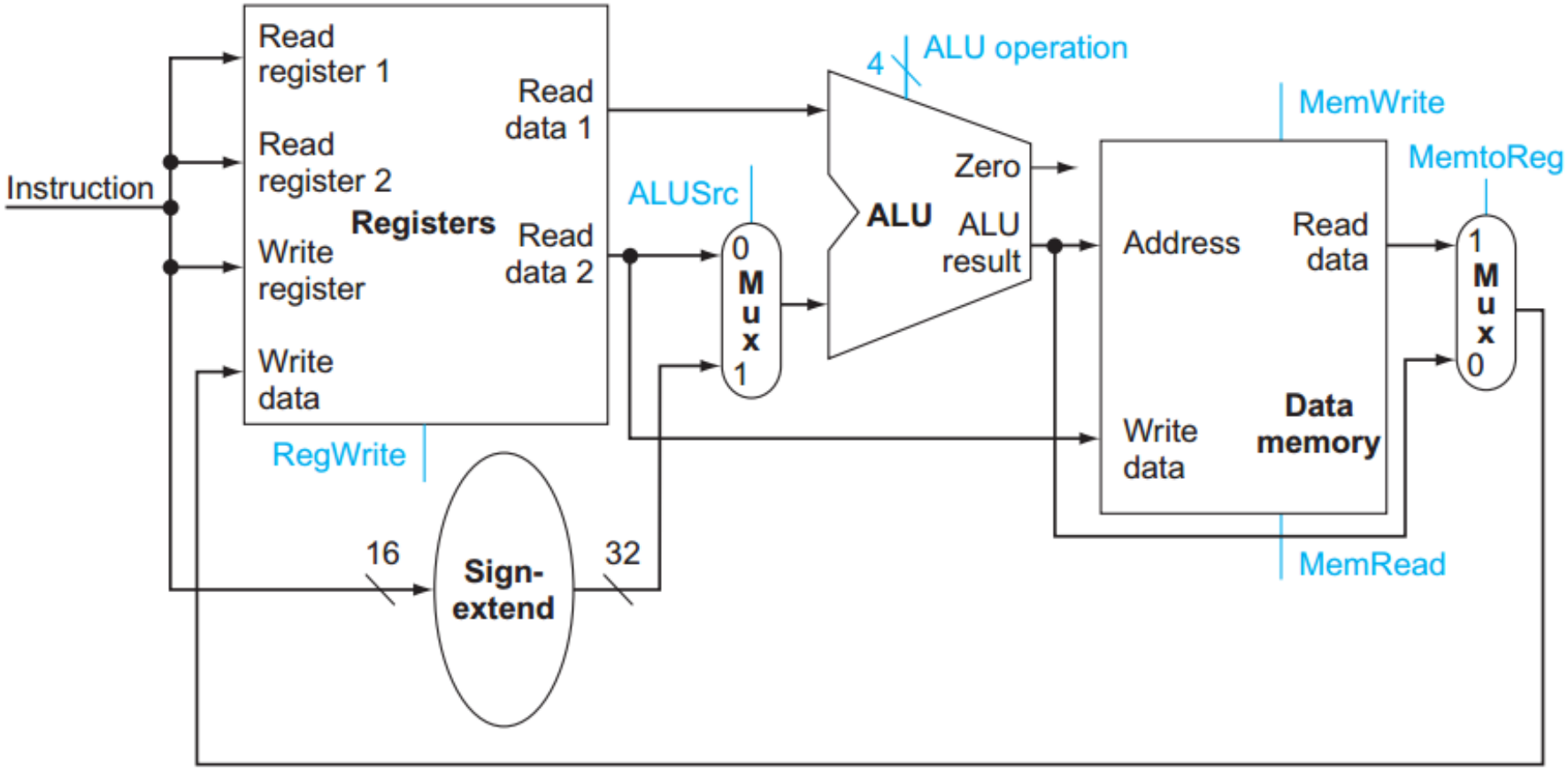
b. Sign extension unit

I Type

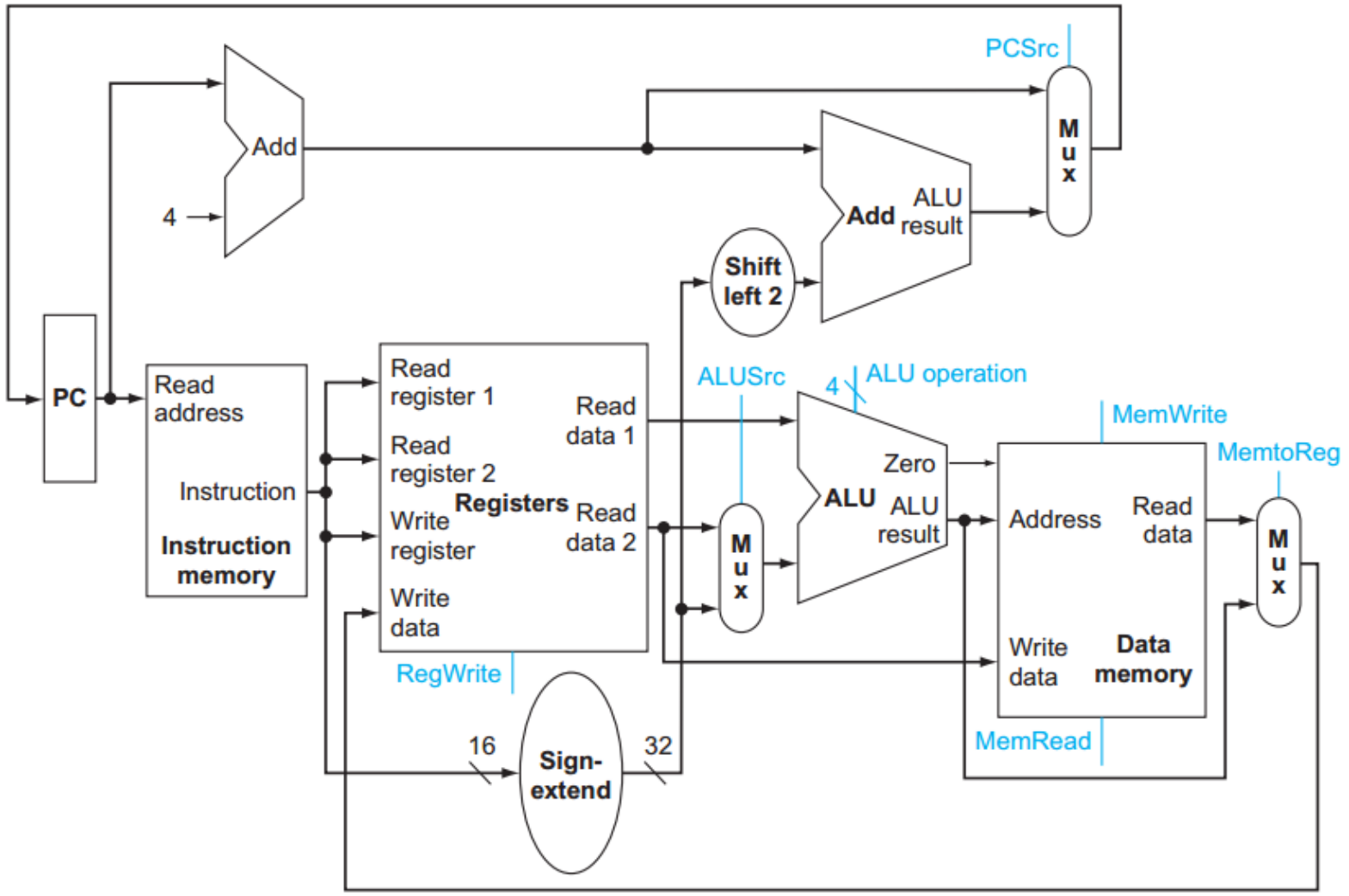


The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.

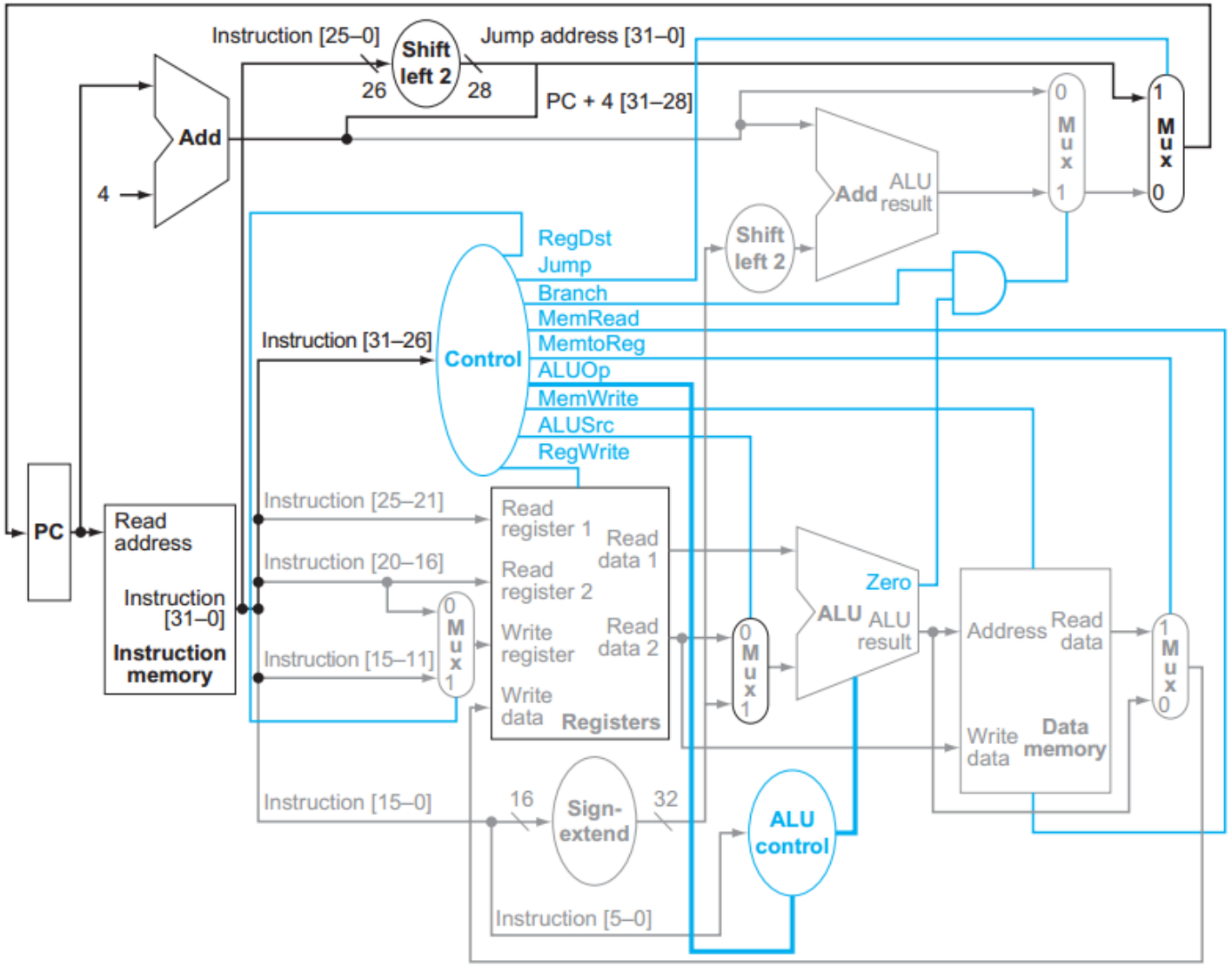
The datapath for the memory instructions and the R-type instructions.



The simple datapath for the core MIPS architecture combines the elements required by different instruction classes.



J Type and Control



Control

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	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

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111