



计算机系统基础05



Machine-Level Programming I: Basics

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
 - Now 3 volumes, about 5,000 pages of documentation
- Complex instruction set computer (CISC)
 - Many different instructions with many different formats
 - But, only small subset encountered with Linux programs
 - Hard to match performance of Reduced Instruction Set Computers (RISC)
 - But, Intel has done just that!
 - In terms of speed. Less so for low power.

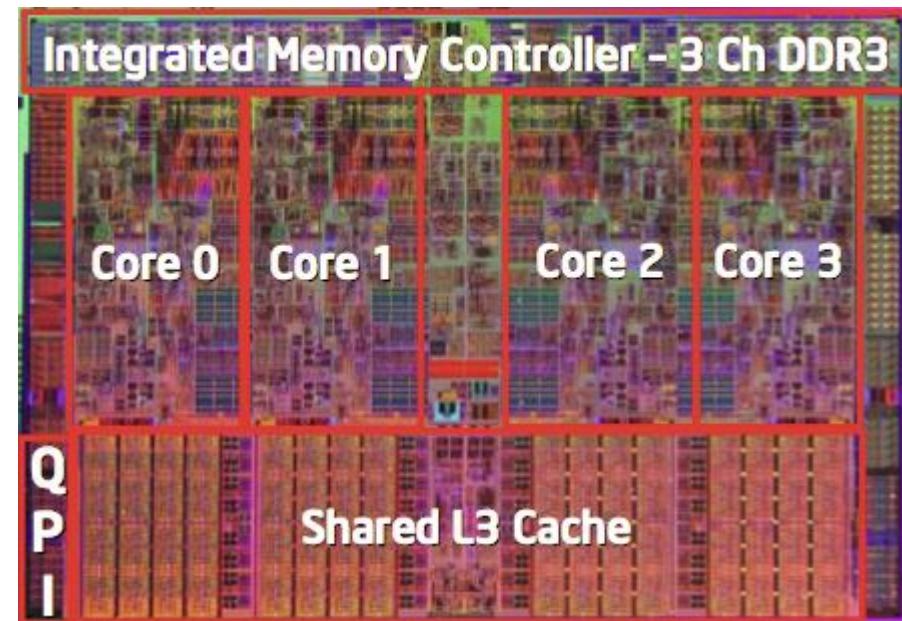
Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
			<ul style="list-style-type: none">▪ First 16-bit Intel processor. Basis for IBM PC & DOS▪ 1MB address space
■ 386	1985	275K	16-33
			<ul style="list-style-type: none">▪ First 32 bit Intel processor , referred to as IA32▪ Added “flat addressing”, capable of running Unix
■ Pentium 4E	2004	125M	2800-3800
			<ul style="list-style-type: none">▪ First 64-bit Intel x86 processor, referred to as x86-64
■ Core 2	2006	291M	1060-3333
			<ul style="list-style-type: none">▪ First multi-core Intel processor
■ Core i7	2008	731M	1600-4400
			<ul style="list-style-type: none">▪ Four cores (our shark machines)

Intel x86 Processors, cont.

■ Machine Evolution

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2000	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M
■ Core i7 Skylake	2015	1.9B



■ Added Features

- Instructions to support **multimedia** operations
- Instructions to enable **more efficient conditional** operations
- Transition from 32 bits to 64 bits
- More cores

Intel x86 Processors, cont.

■ Past Generations

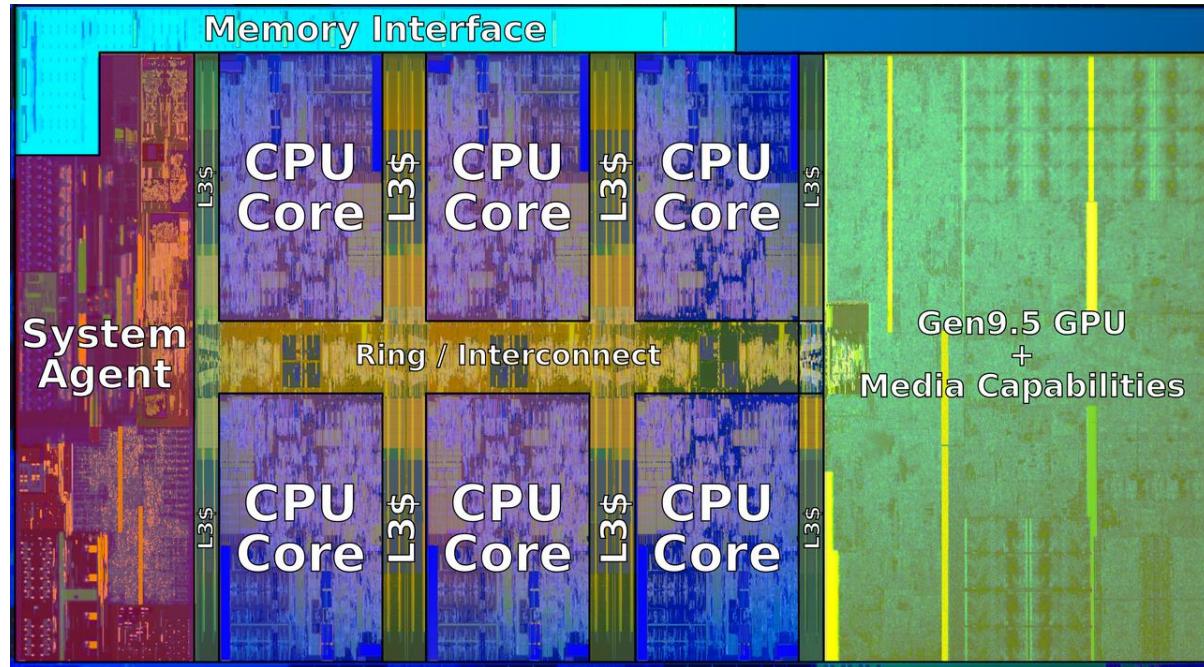
		Process technology
■	1 st Pentium Pro	1995
■	1 st Pentium III	1999
■	1 st Pentium 4	2000
■	1 st Core 2 Duo	2006

■ Recent & Upcoming Generations

1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
9.	Cannonlake	2018	10 nm
10.	Ice Lake	2019	10 nm
11.	Tiger Lake	2020	10 nm
12.	Alder Lake	2022	“intel 7” (10nm+++)

Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

2018 State of the Art: Coffee Lake



■ Mobile Model: Core i7

- 2.2-3.2 GHz
- 45 W

■ Desktop Model: Core i7

- Integrated graphics
- 2.4-4.0 GHz
- 35-95 W

■ Server Model: Xeon E

- Integrated graphics
- Multi-socket enabled
- 3.3-3.8 GHz
- 80-95 W

x86 Clones: Advanced Micro Devices (AMD)

■ Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

■ Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

■ Recent Years

- Intel got its act together
 - 1995-2011: Lead semiconductor “fab” in world
 - 2018: #2 largest by \$\$ (#1 is Samsung)
 - 2019: reclaimed #1
- AMD fell behind: Spun off GlobalFoundries
- 2019-20: Pulled ahead! Used TSMC for part of fab
- 2022: Intel re-took the lead

Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
 - Totally different architecture (Itanium)
 - Executes IA32 code only as legacy
 - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
 - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
 - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
 - Extended Memory 64-bit Technology
 - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
 - But, lots of code still runs in 32-bit mode

Our Coverage

■ IA32

- The traditional x86

■ x86-64

- The standard
- bupt1> gcc hello.c

■ Presentation

- Book covers x86-64
- Web aside on IA32
- We will only cover x86-64

Today: Machine Programming I: Basics

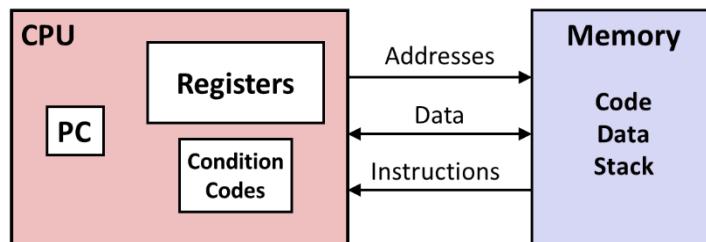
- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
寄存器 操作数
- Arithmetic & logical operations
- C, assembly, machine code

Levels of Abstraction

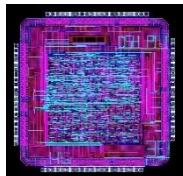
C programmer

```
#include <stdio.h>
int main(){
    int i, n = 10, t1 = 0, t2 = 1, nxt;
    for (i = 1; i <= n; ++i){
        printf("%d, ", t1);
        nxt = t1 + t2;
        t1 = t2;
        t2 = nxt; }
    return 0; }
```

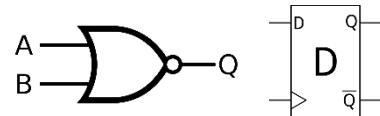
Assembly programmer



Computer Designer



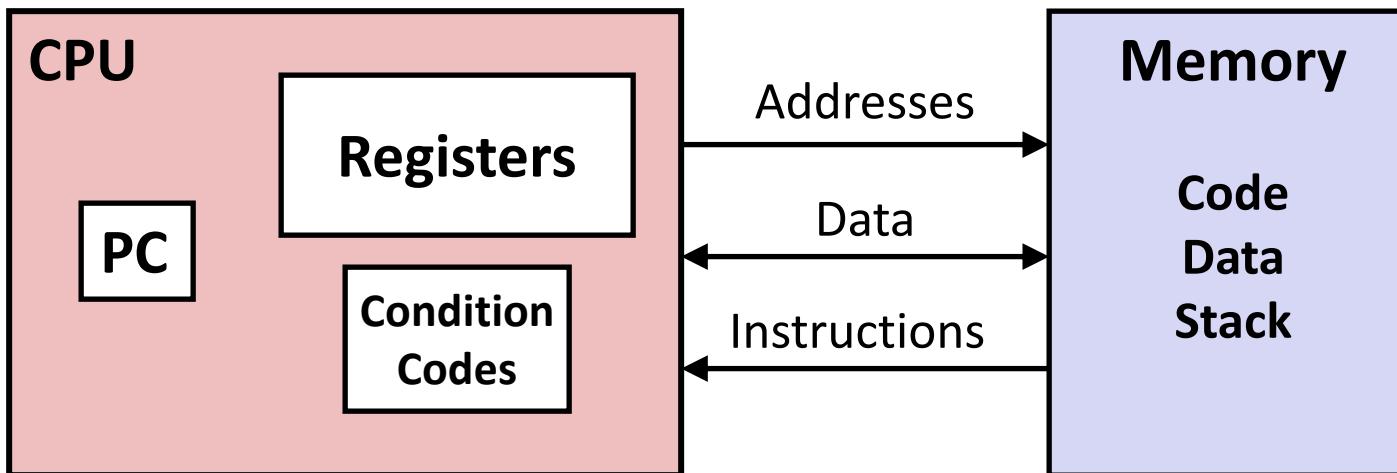
Gates, clocks, circuit layout, ...



Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing assembly/machine code.
 - Examples: instruction set specification, registers
- **Microarchitecture:** Implementation of the architecture
 - Examples: cache sizes and core frequency
- **Code Forms:**
 - **Machine Code:** The byte-level programs that a processor executes
 - **Assembly Code:** A text representation of machine code
- **Example ISAs:**
 - Intel: x86, IA32, Itanium, x86-64
 - ARM: Used in almost all mobile phones
 - RISC V: New open-source ISA

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called “RIP” (x86-64)
- **Register file**
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
 - Data values
 - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- (SIMD vector data types of 8, 16, 32 or 64 bytes)
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
 - Just contiguously allocated bytes in memory

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers

general purpose



Origin
(mostly obsolete)

accumulate

counter

data

base

source index

destination index

stack pointer

base pointer

16-bit virtual registers
(backwards compatibility)

Assembly Characteristics: Operations

- Transfer data between memory and register
 - Load data from memory into register
 - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
 - Unconditional jumps to/from procedures
 - Conditional branches
 - Indirect branches

Moving Data

■ Moving Data

`movq Source, Dest`

■ Operand Types

操作数

Immediate: Constant integer data

- Example: `$0x400`, `$-533`
- Like C constant, but prefixed with '\$'
- Encoded with 1, 2, or 4 bytes

Register: One of 16 integer registers

- Example: `%rax`, `%r13`
- But `%rsp` reserved for special use
- Others have special uses for particular instructions

Memory 8 consecutive bytes of memory at address given by register

- Simplest example: (`%rax`)
- Various other “addressing modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rn (n=8~15)`

**Warning: Intel docs use
mov Dest, Source**

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	<i>Imm</i>	<i>Reg</i>	movq \$0x4,%rax	temp = 0x4;
		<i>Mem</i>	movq \$-147,(%rax)	*p = -147;
	<i>Reg</i>	<i>Reg</i>	movq %rax,%rdx	temp2 = temp1;
	<i>Reg</i>	<i>Mem</i>	movq %rax,(%rdx)	*p = temp;
	<i>Mem</i>	<i>Reg</i>	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

寻址

■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C
间接引用

movq (%rcx), %rax

■ Displacement D(R) Mem[Reg[R]+D]

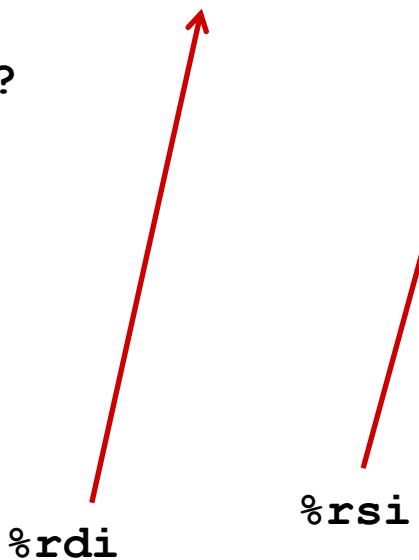
偏移量

- Register R specifies start of memory region
- Constant displacement D specifies offset

movq 8(%rbp), %rdx

Example of Simple Addressing Modes

```
void  
whatAmI (<type> a, <type> b)  
{  
    ????  
}
```



whatAmI:

```
    movq    (%rdi), %rax  
    movq    (%rsi), %rdx  
    movq    %rdx, (%rdi)  
    movq    %rax, (%rsi)  
    ret
```

Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

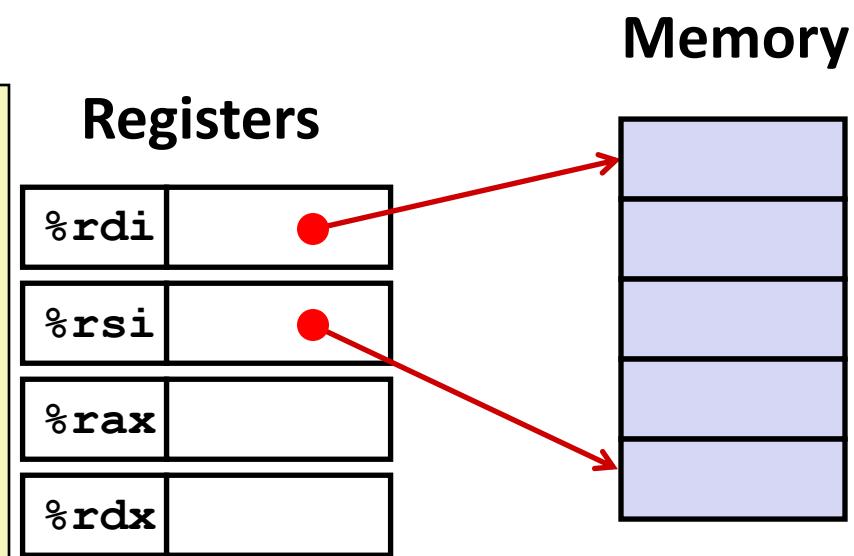
swap:

movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)
ret	

Understanding Swap()

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1



swap:

movq	(%rdi), %rax	# t0 = *xp
movq	(%rsi), %rdx	# t1 = *yp
movq	%rdx, (%rdi)	# *xp = t1
movq	%rax, (%rsi)	# *yp = t0
ret		

Understanding Swap()

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

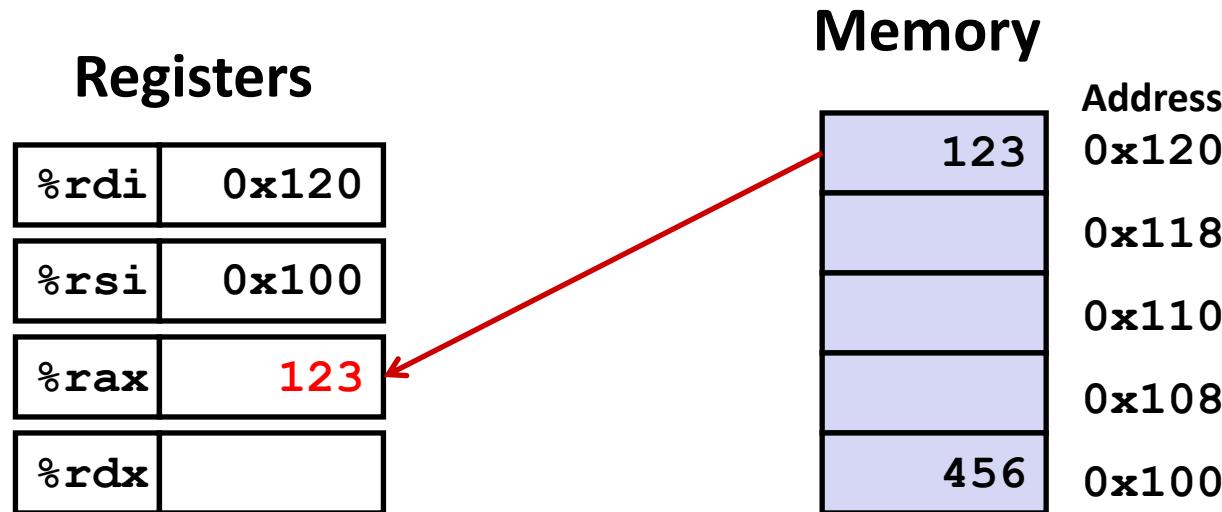
Memory

123	Address 0x120
	0x118
	0x110
	0x108
456	0x100

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

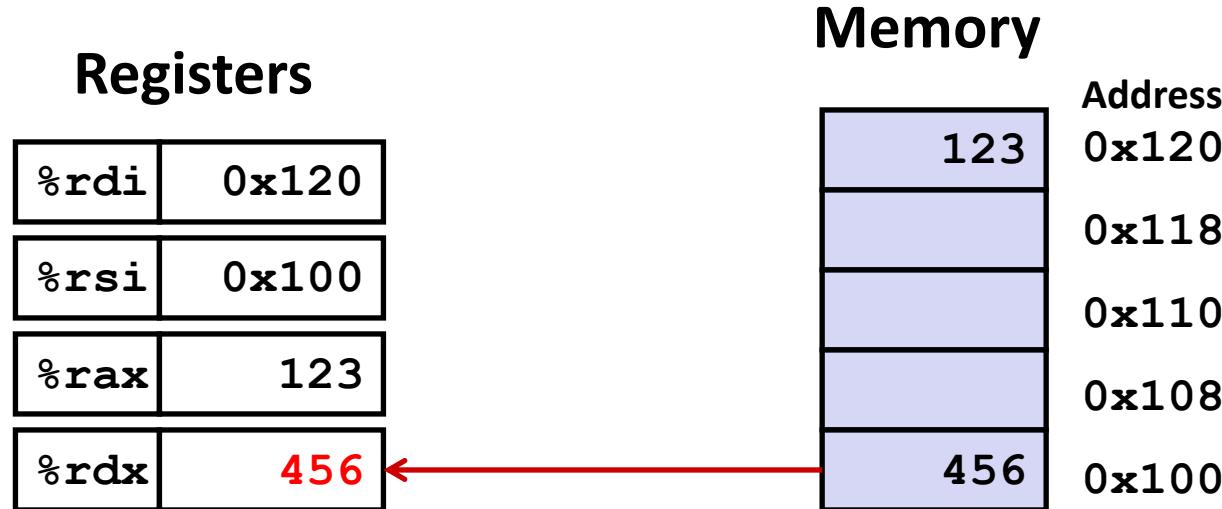
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

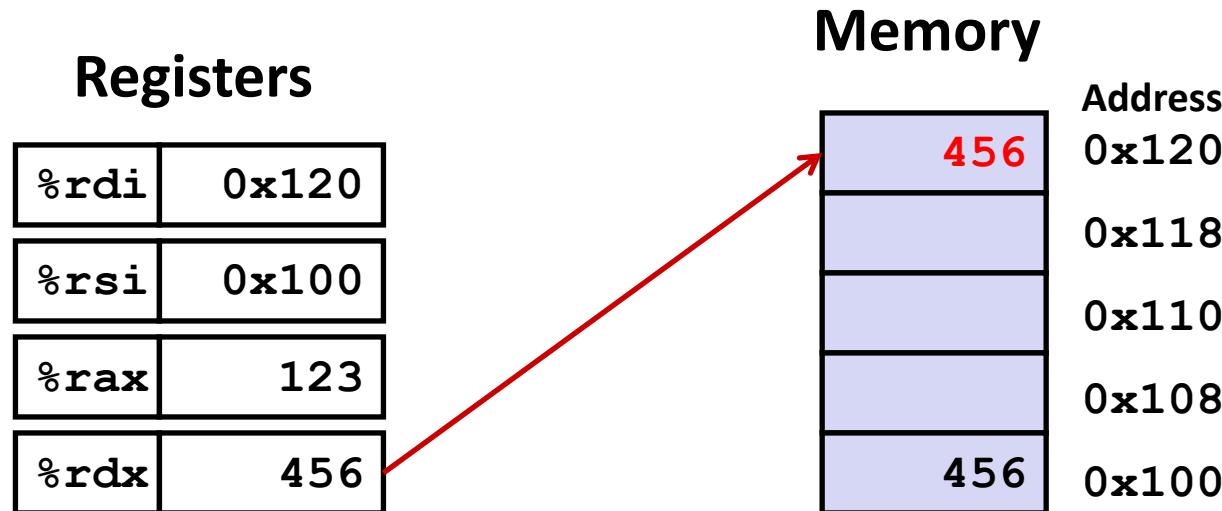
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

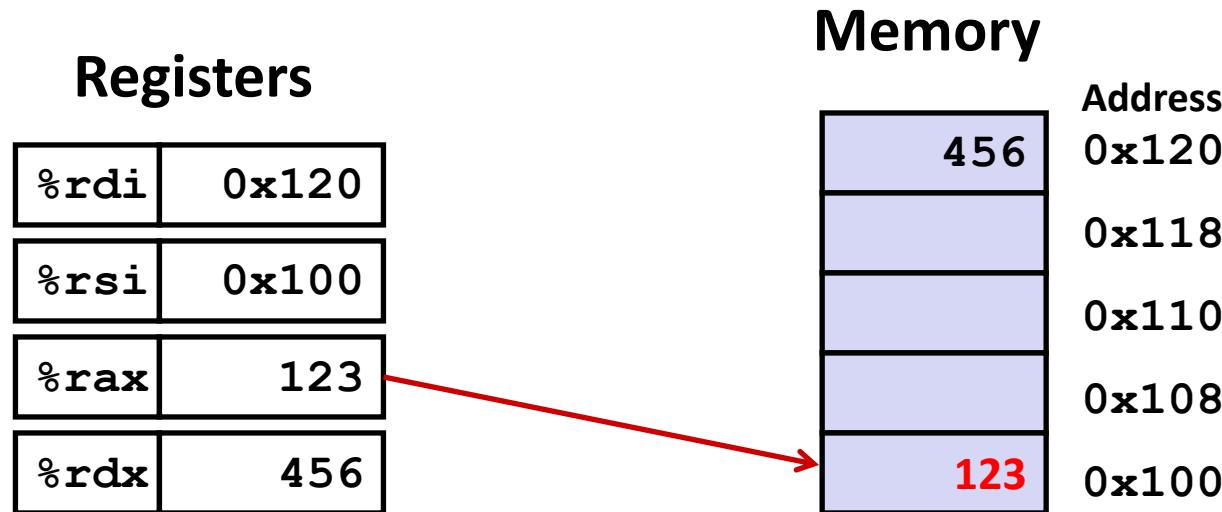
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Moving Data (Register load/store)

A64

■ LDR instruction

- LDR Load data from an address into a register
`LDR X0, <addr>` ; Load from <addr> into X0

■ STR instruction

- STR Store data from a register to an address
`STR X0, <addr>` ; Store content of X0 to <addr>

■ By default, the size of the load/store is determined by the source/destination register name

- LDR `X0, [x1]` ; Load from address held in x1
`movq (%rbx), %rax` ; x86 analog
- STR `W0, [X1]` ; Store 32-bit data to address held in x1
`movl (%rbx), %eax` ; x86 analog
- LDR `X0, [X1, #8]` ; Load from address [X1+8 bytes]
`movq 8(%rbx), %rax` ; x86 analog

Example of Simple Addressing Modes A64

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)
ret	

对比：华为鲲鹏处理器代码

swap:

A64代码

ldr	x2, [x0] @x0参数1
ldr	x3, [x1] @x1参数2
str	x3, [x0]
str	x2, [x1]
ret	

Simple Memory Addressing Modes

■ Normal (R) $\text{Mem}[R]$

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx), %rax
```

■ Displacement $D(R)$ $\text{Mem}[R+D]$

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp), %rdx
```

Complete Memory Addressing Modes

■ Most General Form

$$D(Rb, Ri, S) \quad \text{Mem}[Reg[Rb]+S*Reg[Ri]+ D]$$

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, **except for %rsp**
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

■ Special Cases

$$(Rb, Ri) \quad \text{Mem}[Reg[Rb]+Reg[Ri]]$$

$$D(Rb, Ri) \quad \text{Mem}[Reg[Rb]+Reg[Ri]+D]$$

$$(Rb, Ri, S) \quad \text{Mem}[Reg[Rb]+S*Reg[Ri]]$$

Address Computation Examples

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

D(Rb,Ri,S)**Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Expression	Address Computation	Address
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

Address Computation Instruction

■ **leaq *Src*, *Dst***

- *Src* is address mode expression
- Set *Dst* to address denoted by expression

■ **Uses**

- Computing addresses without a memory reference
 - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form $x + k^*y$
 - $k = 1, 2, 4, \text{ or } 8$

■ **Example**

```
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t = x+2*x
salq $2, %rax           # return t<<2
```

Why use LEA?

■ CPU designers' intended use: calculate a pointer to an object

- An array element, perhaps
- For instance, to pass just one array element to another function

Assembly	C equivalent
lea (%rbx,%rdi,8), %rax	rax = &rbx[rdi]

■ Compiler authors like to use it for ordinary arithmetic

- It can do complex calculations in one instruction
- It's one of the only three-operand instructions the x86 has
- It doesn't touch the condition codes (we'll come back to this)

Assembly	C equivalent
lea (%rbx,%rbx,2), %rax	rax = rbx * 3

Some Arithmetic Operations

■ Two Operand Instructions:

	<i>Format</i>	<i>Computation</i>	
addq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} + \text{Src}$	
subq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} - \text{Src}$	
imulq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} * \text{Src}$	
salq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} \ll \text{Src}$	<i>Also called shlq</i>
sarq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} \gg \text{Src}$	<i>Arithmetic</i>
shrq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} \gg \text{Src}$	<i>Logical</i>
xorq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} \wedge \text{Src}$	
andq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} \& \text{Src}$	
orq	<i>Src,Dest</i>	$\text{Dest} = \text{Dest} \mid \text{Src}$	

- Watch out for argument order! *Src,Dest*
(Warning: Intel docs use “op *Dest,Src*”)
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

■ One Operand Instructions

incq	<i>Dest</i>	$Dest = Dest + 1$
decq	<i>Dest</i>	$Dest = Dest - 1$
negq	<i>Dest</i>	$Dest = -Dest$
notq	<i>Dest</i>	$Dest = \sim Dest$

■ See book for more instructions

- Figure3.4-3.6, Figure3.10

Some Arithmetic Operations

A64指令

■ Two Operand Instructions:

	<i>Format</i>	<i>Computation</i>	
add{ s }	<i>rd, rn, rm</i>	$rd = rn + rm$	setting flags if appended with 'S'
sub{ s }	<i>rd, rn, rm</i>	$rd = rn - rm$	带s的指令影响标志位
mul	<i>rd, rn, rm</i>	$rd = rn \times rm$	
lsl	<i>rd, rn, rm</i>	$rd = rn \ll rm$	
asr	<i>rd, rn, rm</i>	$rd = rn >> rm$	<i>Arithmetic</i>
lsr	<i>rd, rn, rm</i>	$rd = rn >> rm$	<i>Logical</i>
eor	<i>rd, rn, rm</i>	$rd = rn ^ rm$	
and{ s }	<i>rd, rn, rm</i>	$rd = rn \& rm$	
orr	<i>rd, rn, rm</i>	$rd = rn rm$	

Arithmetic Expression Example

```
long arith  
(long x, long y, long z)  
{  
    long t1 = x+y;  
    long t2 = z+t1;  
    long t3 = x+4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

arith:

```
    leaq    (%rdi,%rsi), %rax  
    addq    %rdx, %rax  
    leaq    (%rsi,%rsi,2), %rdx  
    salq    $4, %rdx  
    leaq    4(%rdi,%rdx), %rcx  
    imulq   %rcx, %rax  
    ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
 - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:

leaq	(%rdi,%rsi), %rax	# t1
addq	%rdx, %rax	# t2
leaq	(%rsi,%rsi,2), %rdx	
salq	\$4, %rdx	# t4
leaq	4(%rdi,%rdx), %rcx	# t5
imulq	%rcx, %rax	# rval
ret		

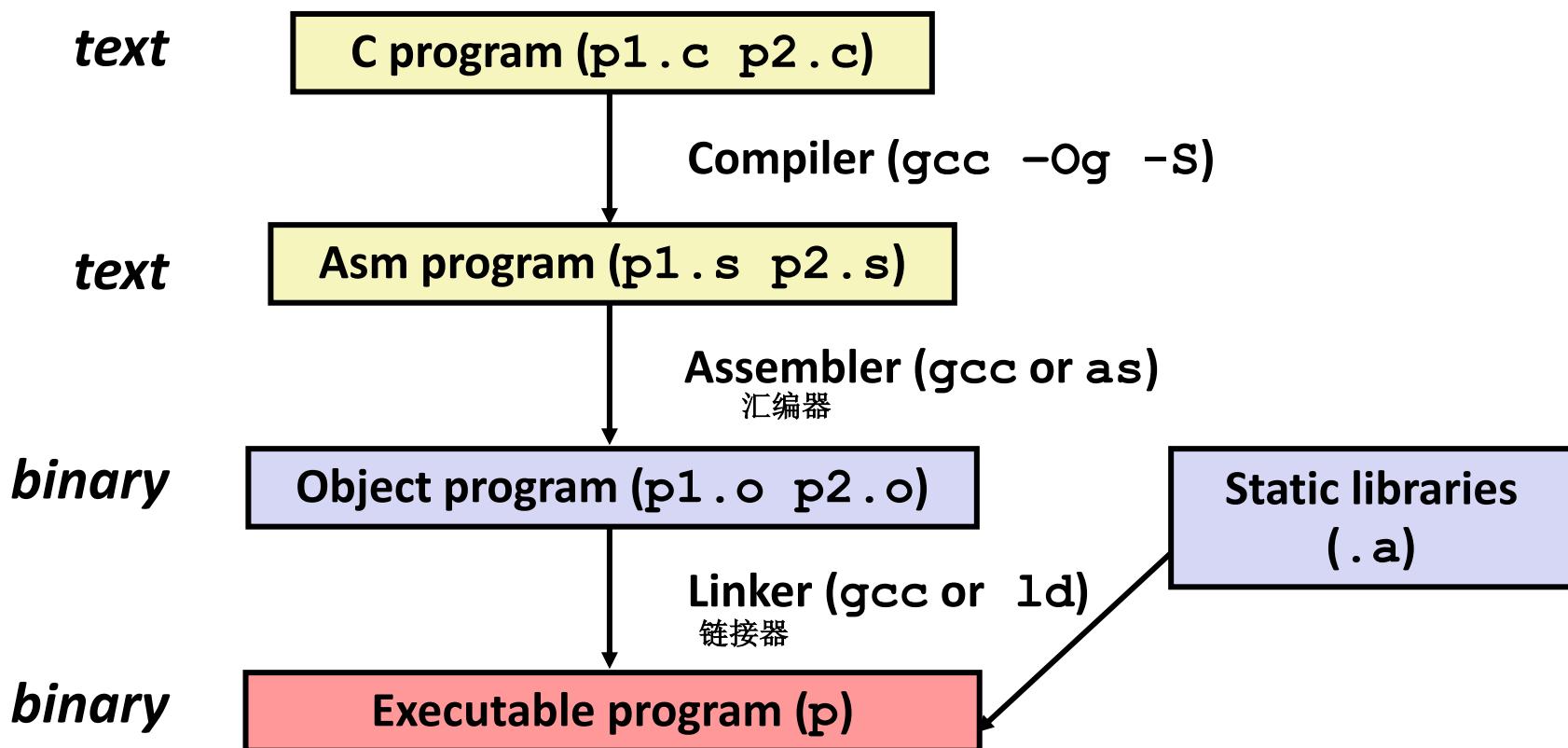
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z , t4
%rax	t1, t2, rval
%rcx	t5

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use basic optimizations (`-Og`) [New to recent versions of GCC]
 - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Generated x86-64 Assembly

```
sumstore:  
    pushq   %rbx  
    movq   %rdx, %rbx  
    call    plus  
    movq   %rax, (%rbx)  
    popq   %rbx  
    ret
```

Obtain (on shark machine) with command

```
gcc -Og -S sum.c
```

Produces file sum.s

Warning: Will get very different results on non-Shark machines (Andrew Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

What it really looks like

```
.globl sumstore
.type sumstore, @function

sumstore:
.LFB35:
    .cfi_startproc
    pushq %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc

.LFE35:
    .size sumstore, .-sumstore
```

What it really looks like

```

.globl sumstore
.type sumstore, @function

sumstore:
.LFB35:
    .cfi_startproc
    pushq %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc

.LFE35:
    .size sumstore, .-sumstore

```

**Things that look weird
and are preceded by a ':
are generally directives.**

伪指令

```

sumstore:
    pushq %rbx
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    ret

```

Object Code

Code for sumstore

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

■ Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

■ Linker

- Resolves references between files
- Combines with static run-time libraries
 - E.g., code for **malloc**, **printf**
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

■ C Code

- Store value **t** where designated by **dest**

```
movq %rax, (%rbx)
```

■ Assembly

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:
 - t:** Register **%rax**
 - dest:** Register **%rbx**
 - *dest:** Memory **M[%rbx]**

```
0x40059e: 48 89 03
```

■ Object Code

- 3-byte instruction
- Stored at address **0x40059e**

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:  
 400595: 53          push    %rbx  
 400596: 48 89 d3    mov      %rdx,%rbx  
 400599: e8 f2 ff ff ff  callq   400590 <plus>  
 40059e: 48 89 03    mov      %rax,(%rbx)  
 4005a1: 5b          pop     %rbx  
 4005a2: c3          retq
```

■ Disassembler

`objdump -d sum`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Disassembled

```
Dump of assembler code for function sumstore:  
0x0000000000400595 <+0>: push    %rbx  
0x0000000000400596 <+1>: mov     %rdx,%rbx  
0x0000000000400599 <+4>: callq   0x400590 <plus>  
0x000000000040059e <+9>: mov     %rax,(%rbx)  
0x00000000004005a1 <+12>: pop    %rbx  
0x00000000004005a2 <+13>: retq
```

- Within gdb Debugger
 - Disassemble procedure

gdb sum

disassemble sumstore

Alternate Disassembly

Object Code

```
0x0400595:  
 0x53  
 0x48  
 0x89  
 0xd3  
 0xe8  
 0xf2  
 0xff  
 0xff  
 0xff  
 0x48  
 0x89  
 0x03  
 0x5b  
 0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:  
 0x0000000000400595 <+0>: push    %rbx  
 0x0000000000400596 <+1>: mov     %rdx,%rbx  
 0x0000000000400599 <+4>: callq   0x400590 <plus>  
 0x000000000040059e <+9>: mov     %rax,(%rbx)  
 0x00000000004005a1 <+12>:pop    %rbx  
 0x00000000004005a2 <+13>:retq
```

■ Within gdb Debugger

- Disassemble procedure

```
gdb sum
```

```
disassemble sumstore
```

- Examine the 14 bytes starting at sumstore

```
x/14xb sumstore
```

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:      file format pei-i386

Disassembly of section .text:

0000000140001000 <.text>:
140001000: 40 53          rex push    %rbx
140001002: 55             push    %rbp
140001003: 56             push    %rsi
140001004: 57             push    %rdi
140001005: 41 56          push    %r14
140001007: 48 81 ec 70 02 00 00 sub    $0x270,%rsp
14000100e: 48 8b 05 13 40 00 00 mov    0x4013(%rip),%rax
...
...
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

What Can be Disassembled?

```
% objdump -d WINWORD.EXE  
  
WINWORD.EXE:      file format pei-i386
```

```
Disassembly of section .text:
```

```
0000000140001000 <.text>:  
140001000:  
140001002:  
140001003:  
140001004:  
140001005:  
140001007:  
14000100e:  
...  ...
```

Reverse engineering forbidden by
逆向工程
Microsoft End User License Agreement

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Machine Programming I: Summary

- **History of Intel processors and architectures**
 - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
 - New forms of visible state: program counter, registers, ...
 - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **Assembly Basics: Registers, operands, move**
 - The x86-64 move instructions cover wide range of data movement forms
- **Arithmetic**
 - C compiler will figure out different instruction combinations to carry out computation

教材阅读

■ 第3章 3.1-3.4.3、3.5

鲲鹏处理器简介



前言

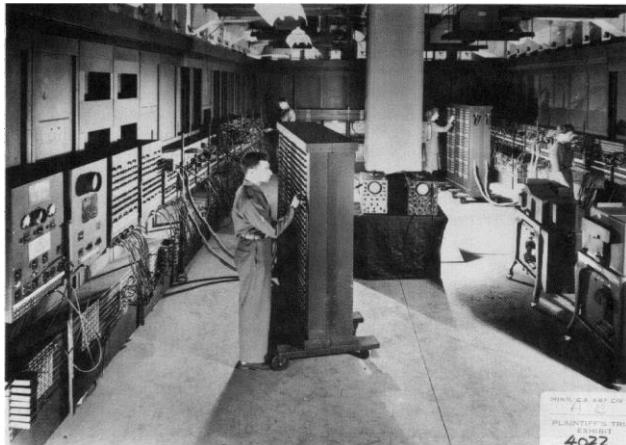
- 鲲鹏处理器是面向ICT领域兼容ARM 64bit指令集的多核处理器芯片，基于华为自研的具有完全知识产权的ARMv8架构，采用业界领先的7nm制程，多Die合封的Chiplet封装工艺，在提供强大计算能力的同时还集成了丰富且强大的I/O能力，为行业用户实现业务加速提供支撑。

目录

1. ARM处理器简介
2. 鲲鹏处理器
3. ARM寻址方式
4. ARM指令集

芯片是信息社会的基石

世界第一台通用电子计算机—ENIAC



1946年诞生，由18,800多个电子管组成，重量30多吨，占地面积170多平方米。

得益于芯片的发展



小型化，高性能



内存



手机

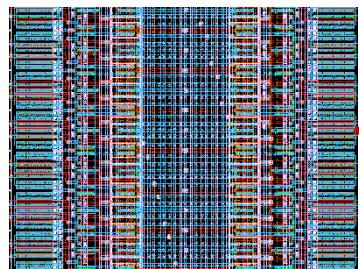


计算机



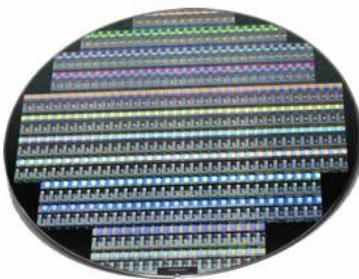
数码相机

芯片行业产业链



芯片设计

集成电路设计涉及对电子器件和器件间互连线模型的创建。



晶圆加工

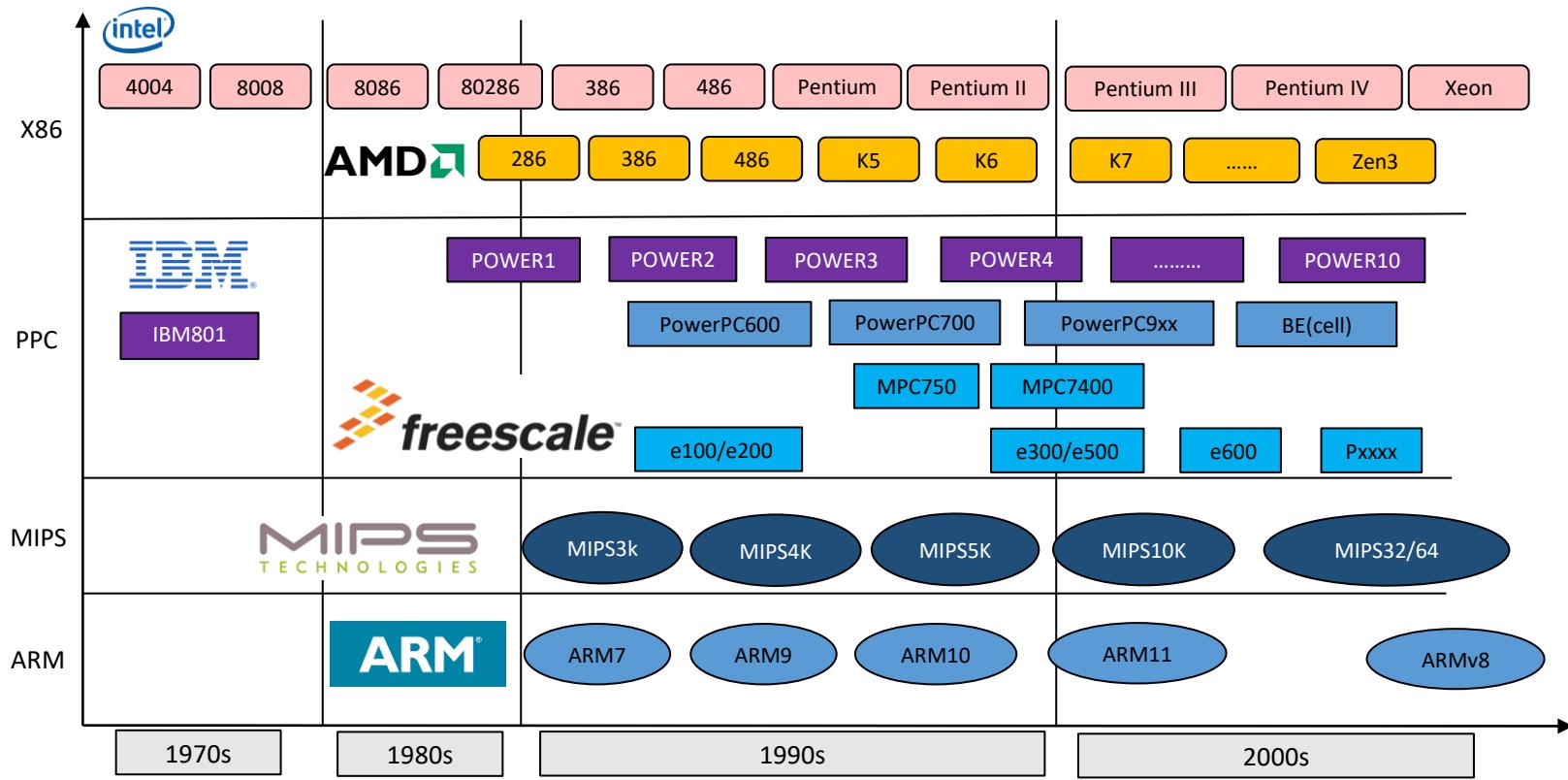
晶圆加工是一系列化学处理步骤，使得电子电路逐渐形成在使用纯半导体材料制作的芯片上。



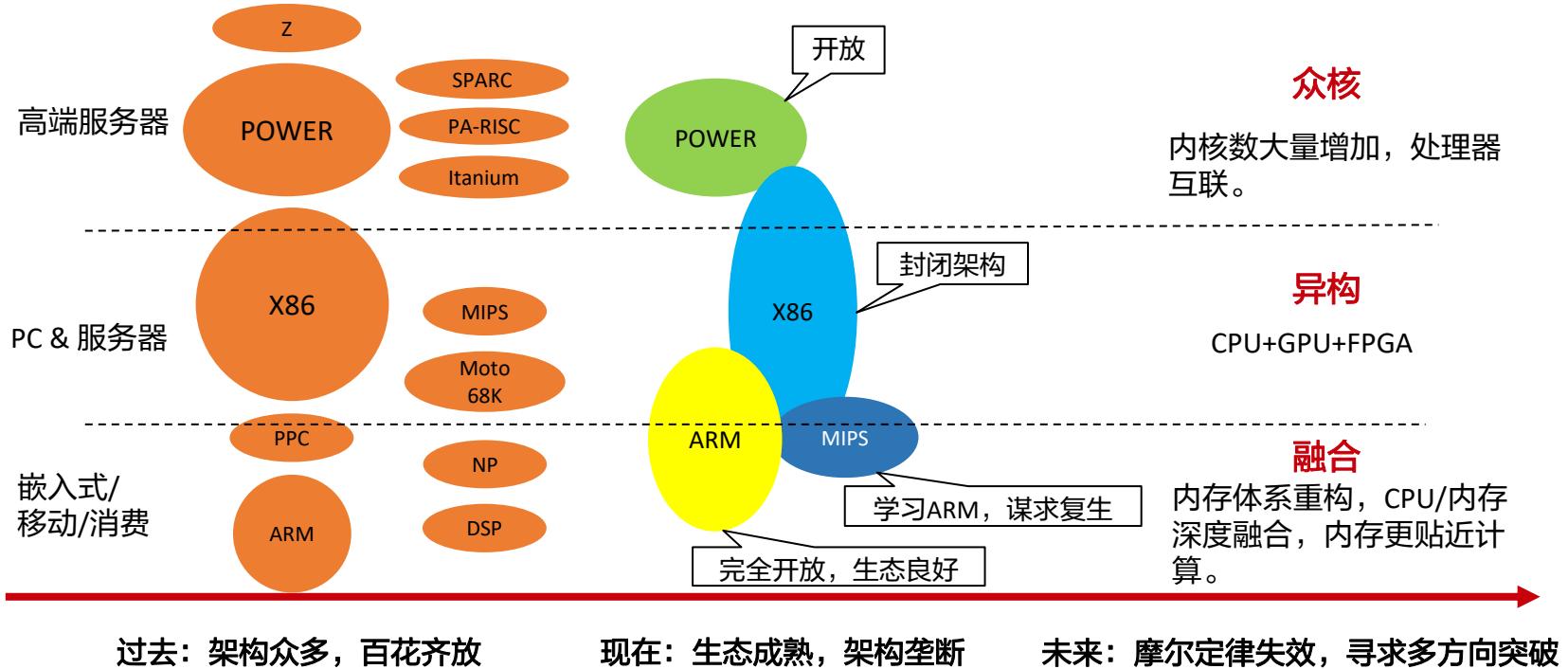
芯片封装测试

封装是将器件的核心晶粒封装在一个支撑物之内的过程，这个封装可以防止物理损坏以及化学腐蚀，并提供对外连接的引脚，之后将进行集成电路性能测试。

主流CPU发展路径



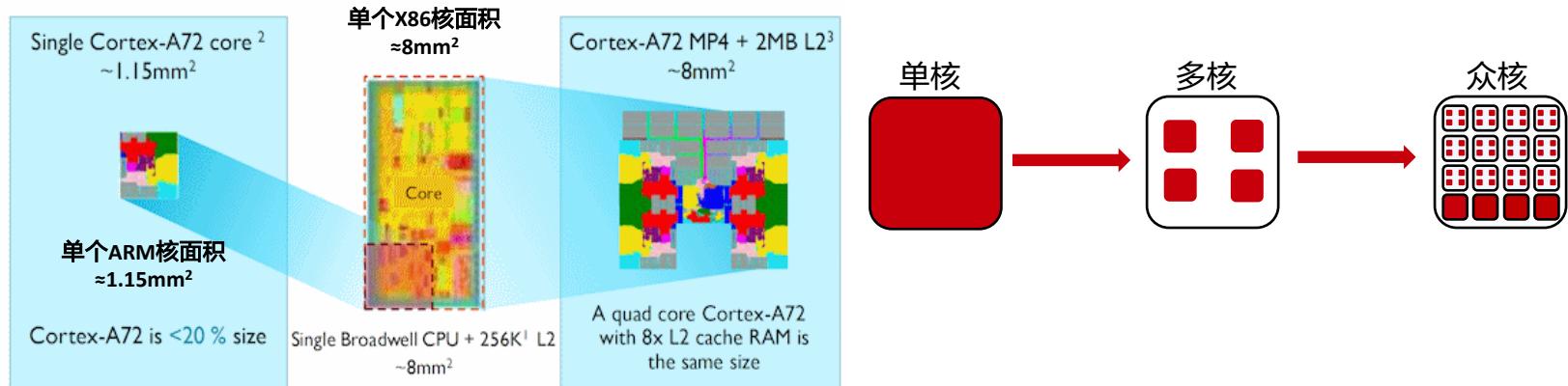
处理器发展趋势



ARM提供更多计算核心

- 工艺、主频遇到瓶颈后，开始通过增加核数的方式来提升性能；
- 芯片的物理尺寸有限制，不能无限制的增加；
- ARM的众核横向扩展空间优势明显。

Cortex-A72: Ideal for dense compute environments



ARM服务器级别处理器一览



Hi1612

32C, 2.1GHz
16nm

Hi1616

32C,2.4GHz
16nm

Hi1620

64C,3.0GHz
7nm

CAVIUM

Thunder-X

48C,2.5GHz
28nm

Thunder-X2

32-54C,3.0GHz
14nm

高通

2400

48Cores,2.2-2.6GHz
14nm

AMPERE

X-Gene3

32C, 3.3GHz
16nm



飞腾

FT1500

16 Cores,1.6GHz
28nm

FT2000+

64Cores,2.3GHz
16nm

横轴代表性能

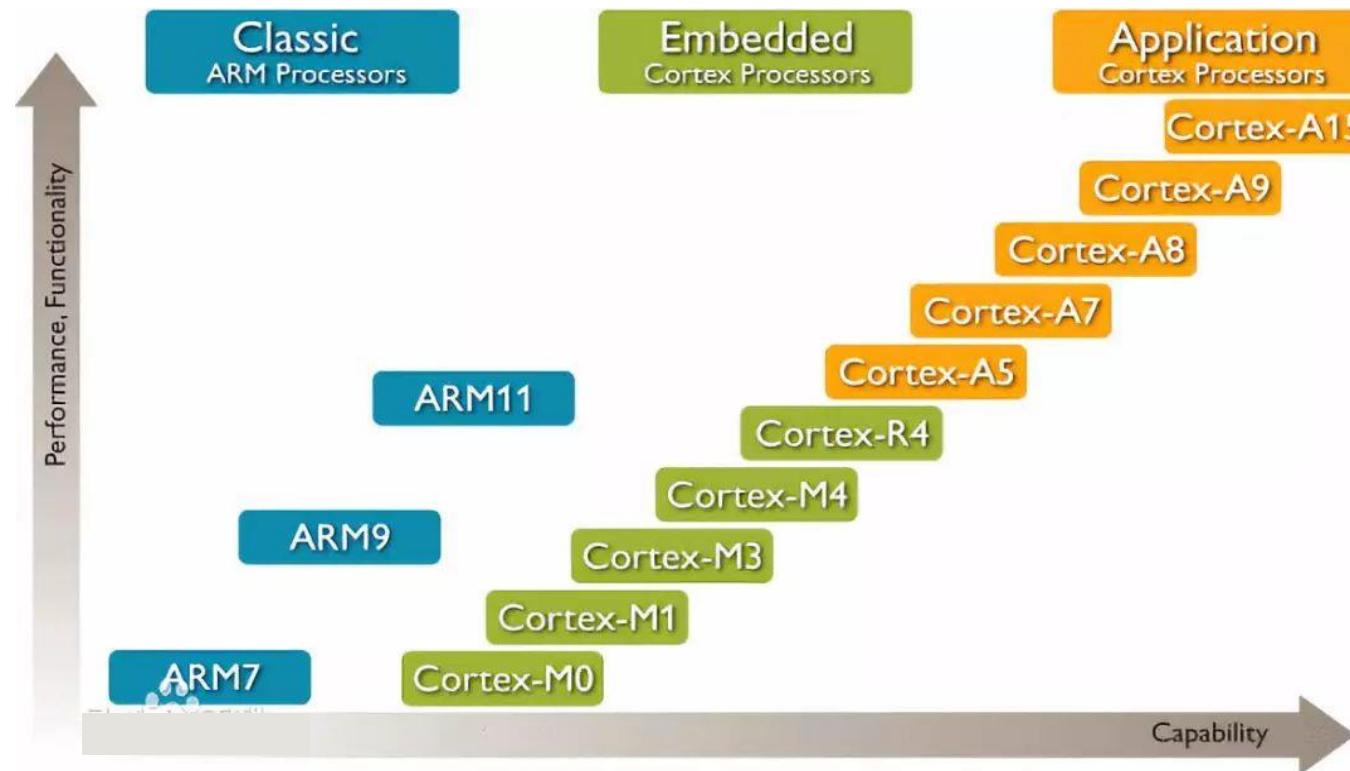
ARM处理器类别

- ARM处理器的分类：
 - ARM经典处理器（ Classic Processors ）；
 - ARM Cortex系列处理器；
 - ARM Cortex嵌入式处理器；
 - ARM专业处理器。

ARM架构发展史 (1)

架构	处理器家族
ARMv1	ARM1
ARMv2	ARM2、ARM3
ARMv3	ARM6、ARM7
ARMv4	Strong ARM、ARM7TDMI、ARM9TDMI
ARMv5	ARM7EJ、ARM9E、ARM10E、Xscale
ARMv6	ARM11、ARM Cortex-M
ARMv7	ARM Cortex-A、ARM Cortex-M、ARM Cortex-R
ARMv8	Cortex-A50,Cortex-A57,Cortex-A72

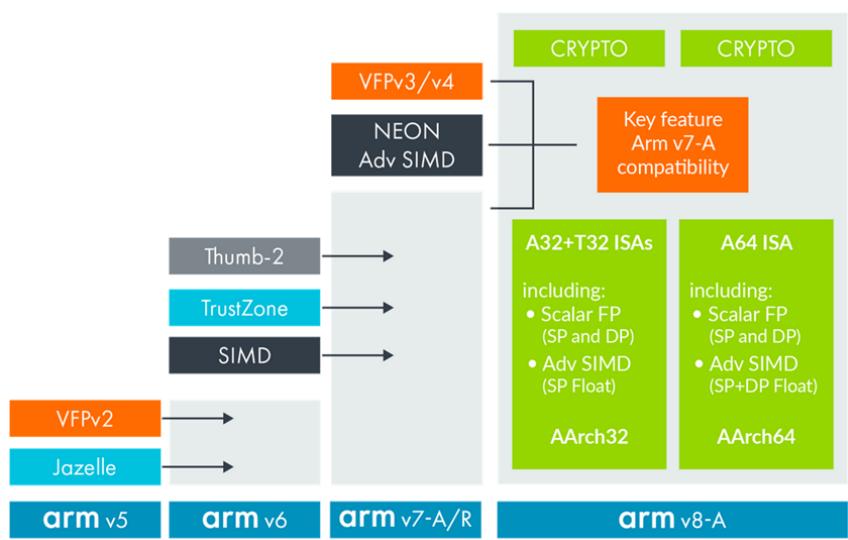
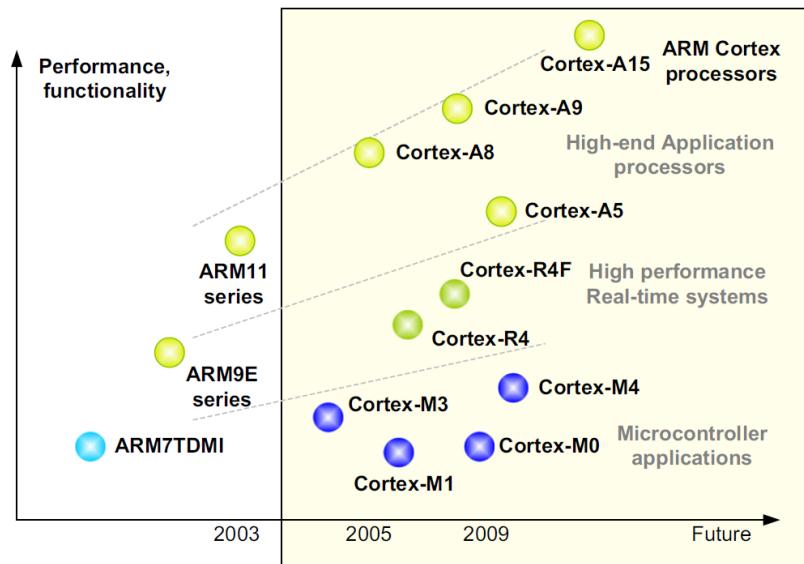
ARM架构发展史 (2)



ARM架构发展史 (3)

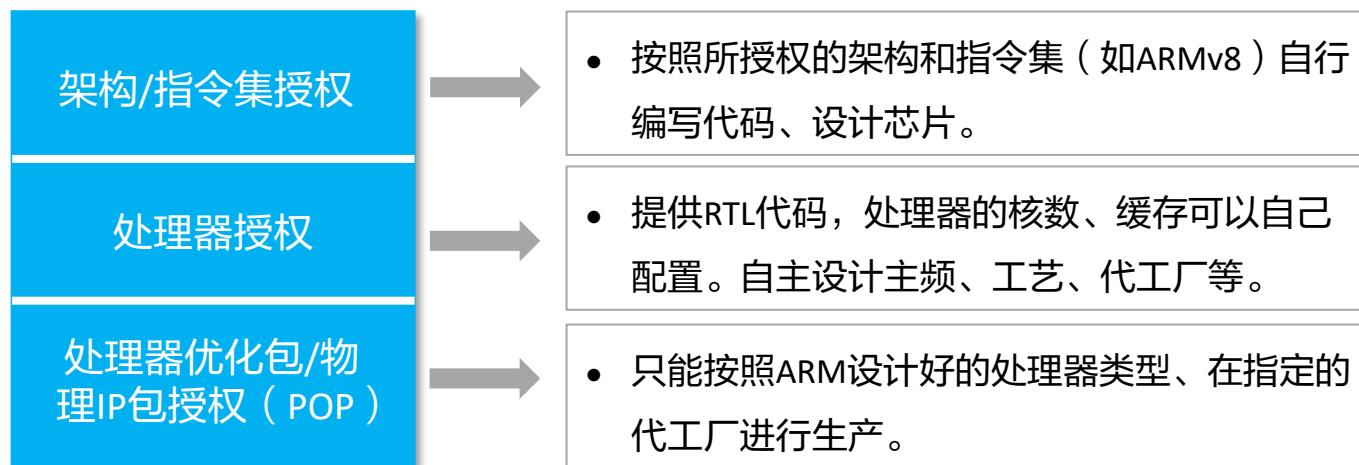
- 从ARMv7开始，CPU命名为Cortex，并划分为A、R、M三大系列，分别为不同的市场提供服务
 - A (Application)系列：应用型处理器，面向具有复杂软件操作系统的面向用户的应用，为手机、平板、AP等终端设备提供全方位的解决方案；
 - R (Real-Time)系列：实时高性能处理器，为要求可靠性、高可用性、容错功能、可维护性和实时响应的嵌入式系统提供高性能计算解决方案；
 - M (Microcontroller)系列：高能效、易于使用的处理器，主要用于通用低端，工业，消费电子领域微控制器。

ARM架构发展史 (4)



ARM公司授权体系

- ARM目前在全球拥有大约1000个授权合作商、320家伙伴，但是购买架构授权的厂家不超过20家，中国有华为、飞腾获得了架构授权。

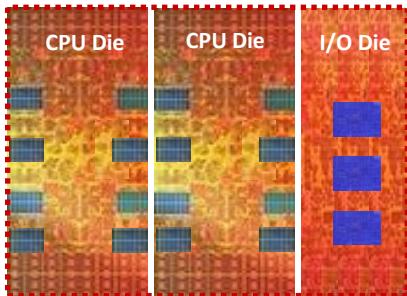


目录

1. ARM处理器简介
- 2. 鲲鹏处理器**
3. ARM寻址方式
4. ARM指令集

鲲鹏简介

鲲鹏是SoC



制程工艺领先：业界领先7nm制程，
多Die合封的Chiplet架构

自研多核内核：自研CPU内核算力提升50%，
自研片间互联，支持多路互联

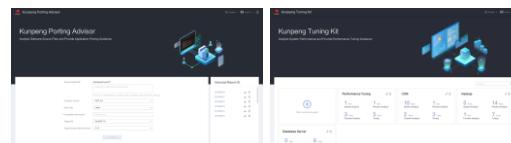
率先支持下一代网络和接口：支持8通道
内存控制器和100GE端口

鲲鹏是计算平台

处理器->单机->集群，鲲鹏开放硬件平台



完备的软件工具链，发挥鲲鹏最佳性能



分析扫描工具
代码迁移工具

性能优化工具
加速库

鲲鹏是生态应用

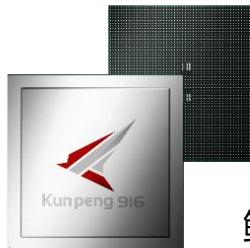
使能合作伙伴

- 应用
- 中间件
- 数据库
- 操作系统
- 服务器/PC

使能行业应用

- 大数据
- 分布式存储
- 高性能计算
- 原生应用
- 云服务

鲲鹏处理器



鲲鹏916

支持多路互联的ARM处理器

- 32核， 2.4 GHz主频
- SPECint性能匹配业界中端，功耗低至75 W
- 支持4通道DDR4控制器
- 支持PCI-e 3.0和SAS/SATA 3.0
- 集成板载GE/10 GE网络
- 支持2路互联



鲲鹏920

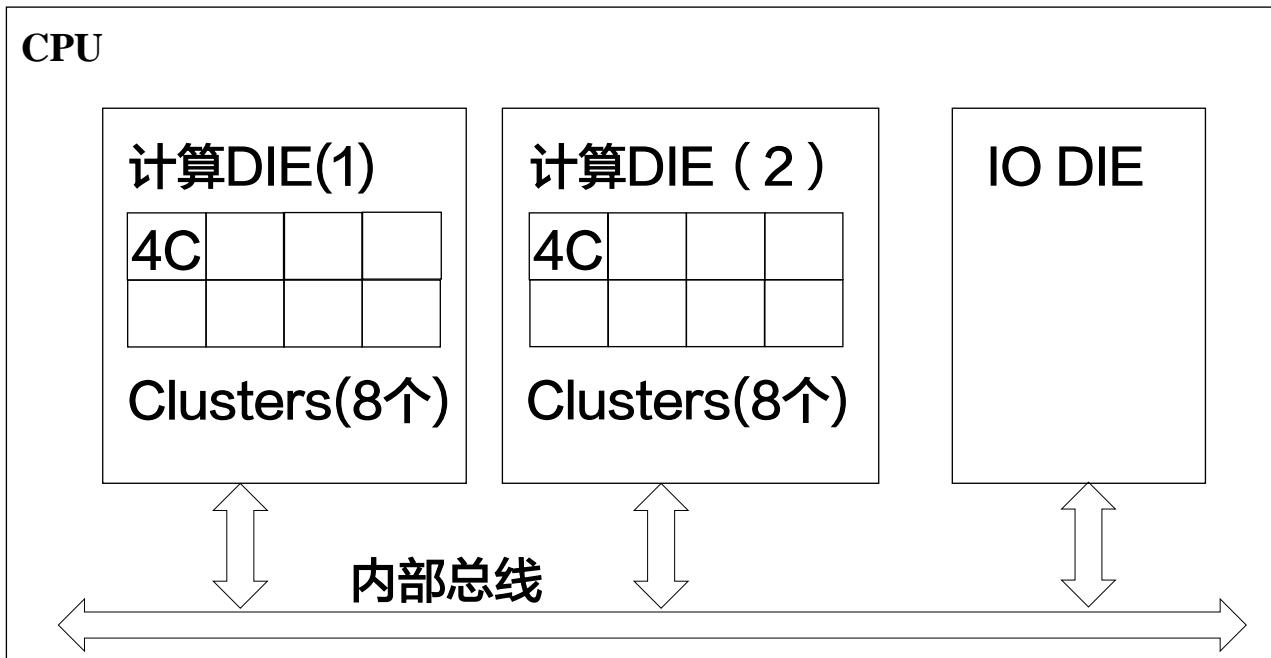
7nm制程， 数据中心ARM处理

- 计算核数提升1倍，最多64核
- SPECint性能提升超过2倍
- 内存通道数提升1倍，支持8通道DDR4控制器
- 支持PCI-e 4.0和CCIX
- 集成板载100 GE网络和加密、压缩加速引擎
- 支持2路或4路互联

鲲鹏920系列芯片概览

- 鲲鹏920提供强大的计算能力，基于华为自研的具有完全知识产权的ARMv8架构，**最多支持64 Core。**
- 通过片间Cache一致性接口Hydra可扩展系统核数，**最多支持到256 Core**，形成性能超强的板级计算节点。
- 支持CPU Core虚拟化、内存虚拟化、中断虚拟化、I/O虚拟化等多项虚拟化技术，使得系统的资源共享更加灵活、系统的迁移过程变得相对简单。
- 鲲鹏920具有丰富且强大的I/O能力，集成以太网控制器、SAS控制器、以及PCI-e控制器。
- 芯片集成安全算法、压缩/解压缩算法、存储算法等加速引擎进行业务加速。

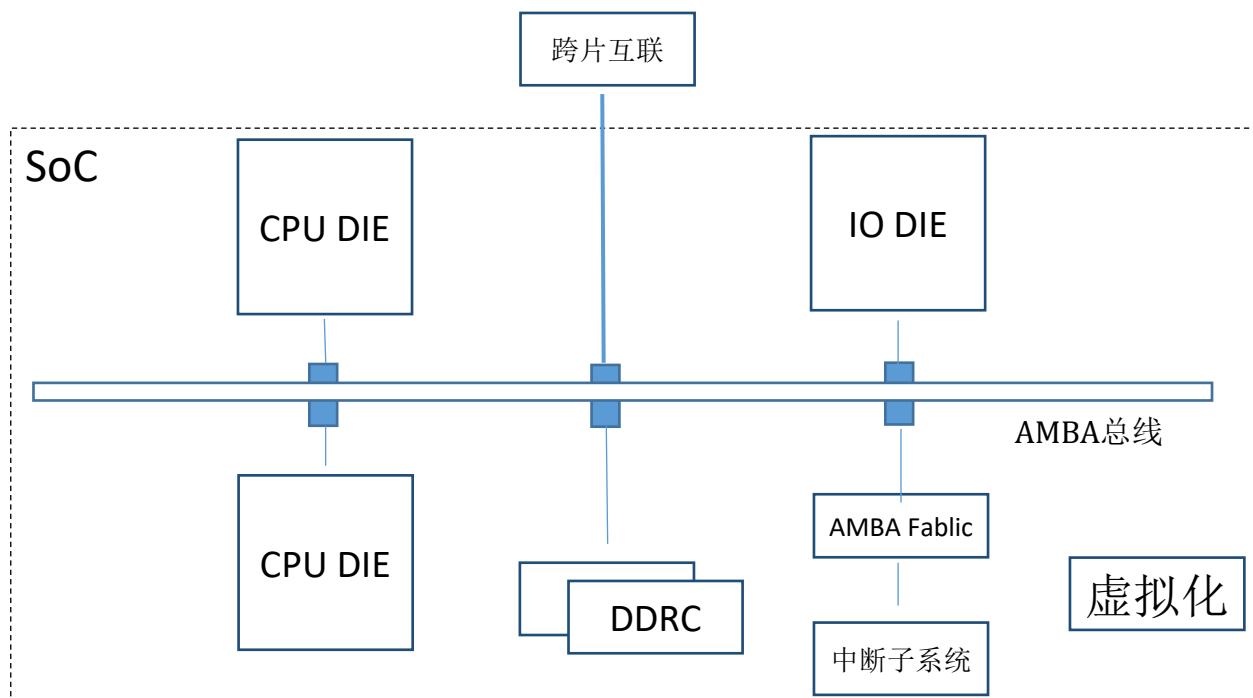
鲲鹏920芯片的架构



如图所示：

- 1片SoC上包含3个DIE，2个计算DIE，1个IO DIE
- 1个计算DIE中8个Cluster
- 1个Cluster中4个Core
- 因此一个鲲鹏920芯片中包含 $4*8*2=64$ 个核
- 计算DIE上的每一个core具有自己的L1和L2级cache，所有的core共享L3级cache
- IO DIE上集成有网络模块、PCIe模块
- 这些DIE在芯片内部通过高速内部总线进行连接

鲲鹏920的其他子系统



- 鲲鹏处理器包含计算、存储、设备IO、中断以及虚拟化等子系统
- 鲲鹏920含有两个CPU DIE、一个IO DIE、以及共8组DDR4 channel，它们通过AMBA (Advanced Microcontroller Bus Architecture)总线互联

基于鲲鹏主板的多样化计算产品

鲲鹏主板开放
共享接口与设备规范



整机参考设计
共享工程能力



- 主板(含鲲鹏处理器)
- 主板接口规范
- BIOS软件 & 规范
- BMC芯片/软件 & 设备管理规范

打造多样化计算产品



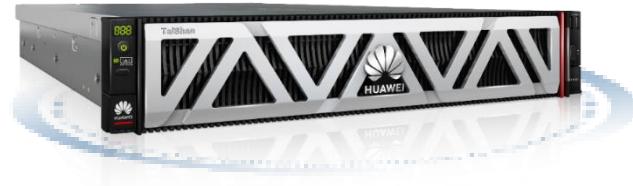
伙伴价值

持续供应

快速开发
(3个月)

安全可靠
(低于业界平均故障率15%)

基于鲲鹏920的华为TaiShan 200服务器



17年的工程工艺积累

40度以上异常高温运行

液冷散热

56G/112G板级高速互联

高速互联

无源背板&三重硬盘抗震

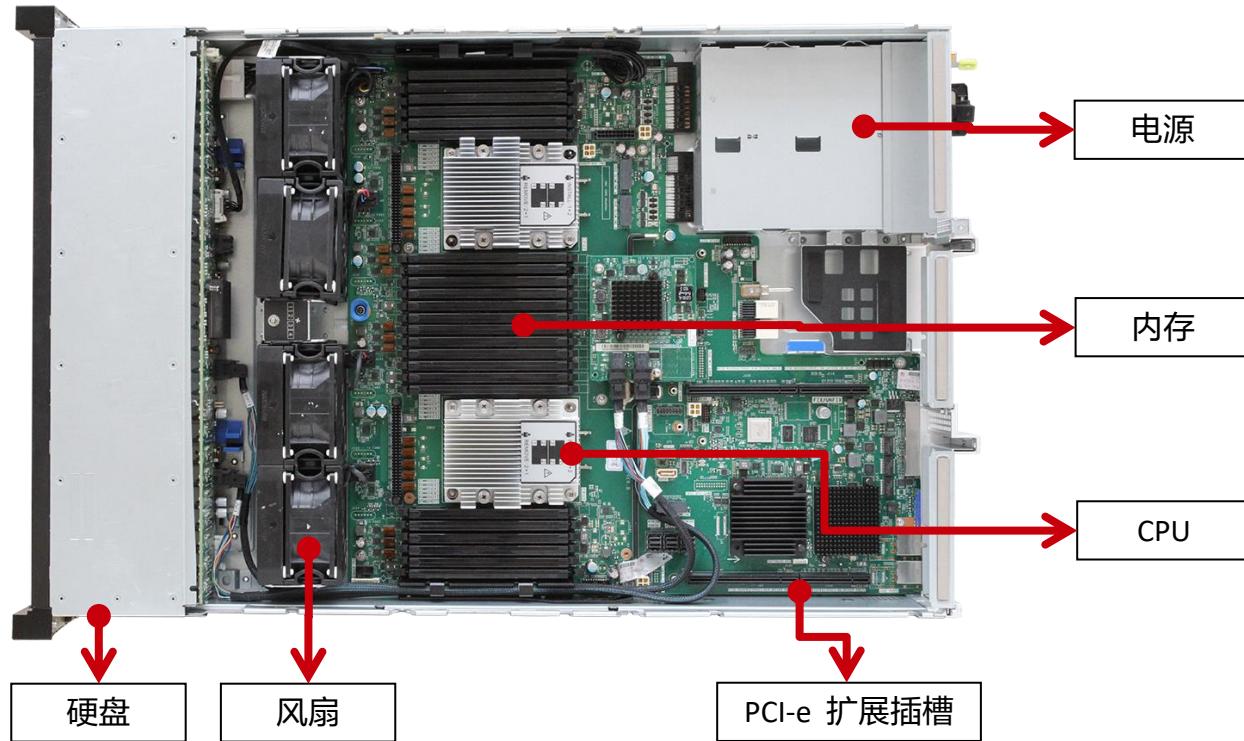
可靠设计

质量品控

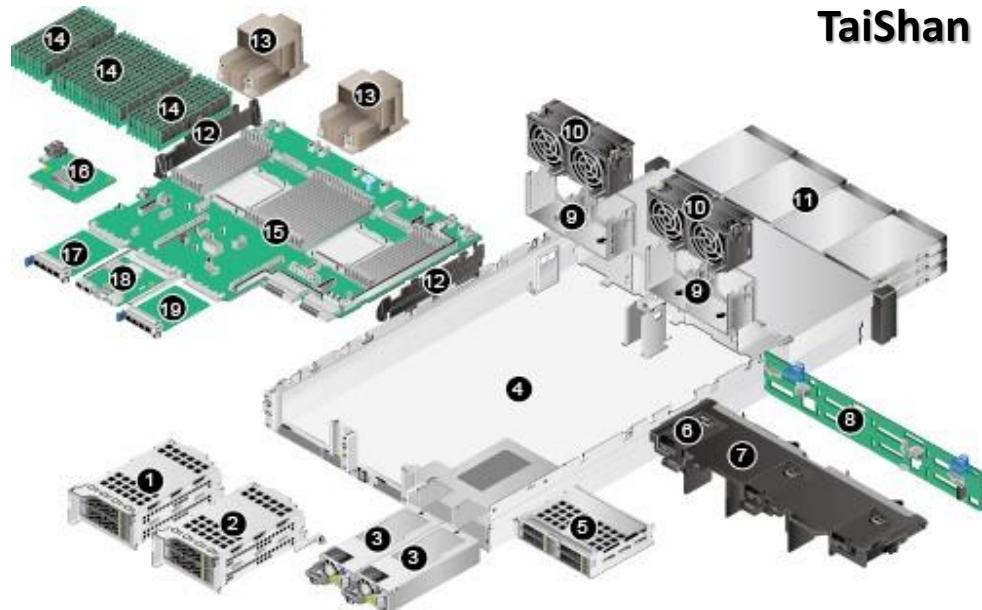
故障率低于业界15%



服务器基础硬件



TaiShan服务器物理结构

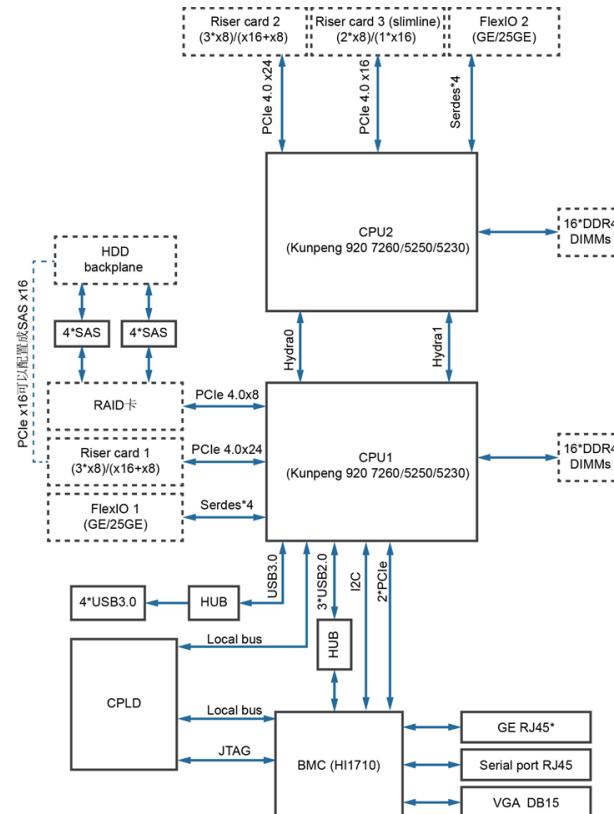


TaiShan 200服务器（型号2280）

1	I/O模组1	2	I/O模组2
3	电源模块	4	机箱
5	I/O模组3	6	超级电容支架
7	导风罩	8	前置硬盘背板
9	风扇支架	10	风扇模块
11	前置硬盘	12	理线架
13	散热器	14	DIMM
15	主板	16	RAID扣卡
17	灵活I/O卡1 (CPU 1)	18	iBMC插卡
19	灵活I/O卡2 (CPU 2)	-	-

TaiShan服务器逻辑结构

- 支持两路自研鲲鹏920 7260、5250、5240或5230处理器，每个处理器支持16个DDR4 DIMM；
- 以太网灵活插卡可支持2种插卡，包括4*GE和4*25GE，通过CPU本身自带高速Serdes接口连接通信介质；
- RAID卡通过PCI-e总线跟CPU1连接，RAID卡引出的SAS信号线缆跟硬盘背板连接，通过不同的硬盘背板可支持多种本地存储规格；
- BMC使用自研管理芯片Hi1710，可外出VGA、管理网口、调试串口等管理接口。



目录

1. ARM处理器简介
2. 鲲鹏处理器
- 3. ARM寻址方式**
4. ARM指令集

基于ARMv8架构的处理器体系结构

- AArch64重要寄存器：

寄存器类型	Bit	描述
X0-X30	64bit	通用寄存器, 如果有需要可以当做32bit使用: W0-W30
LR (X30)	64bit	通常称X30为程序链接寄存器, 保存跳转返回信息地址
SP_ELx	64bit	若PSTATE.M[0] ==1, 则每个ELx选择SP_ELx, 否则选择同一个SP_EL0
ELR_ELx	64bit	异常链接寄存器, 保存异常进入ELx的异常地址 (x={0,1,2,3})
PC	64bit	程序计数器, 俗称PC指针, 总是指向即将要执行的下一条指令
SPSR_ELx	32bit	寄存器, 保存进入ELx的PSTATE状态信息
NZCV	32bit	允许访问的符号标志位
DAIF	32bit	中断使能位: D-Debug, I-IRQ, A-SError, F-FIQ , 逻辑0允许
CurrentEL	32bit	记录当前处于哪个Exception level
SPSel	32bit	记录当前使用SP_EL0还是SP_ELx, x= {1,2,3}
HCR_El2	32bit	HCR_El2.{TEG,AMO,IMO,FMO,RW}控制EL0/EL1的异常路由 逻辑1允许
SCR_El3	32bit	SCR_El3.{EA,IRQ,FIQ,RW}控制EL0/EL1/EL2的异常路由 逻辑1允许
ESR_ELx	32bit	保存异常进入ELx时的异常综合信息, 包含异常类型EC等.
VBAR_ELx	64bit	保存任意异常进入ELx的跳转向量地址 x={0,1,2,3}
PSTATE		不是一个寄存器, 是保存当前PE状态的一组寄存器统称, 其中可访问寄存器有: PSTATE.{NZCV,DAIF,CurrentEL,SPSel},属于ARMv8新增内容,64bit下代替CPSR

AArch64与x86部分寄存器对比(1)

Low 32-bits	ARM Register	Conventional use	X86 Register	Low 32- bits	Low 16- bits	Low 8- bits
W0	X0	Return value, callee-owned	%rax	%eax	%ax	%al
W0	X0	1st argument, callee-owned	%rdi	%edi	%di	%dl
W1	X1	2nd argument, callee-owned	%rsi	%esi	%si	%sl
W2	X2	3rd argument, callee-owned	%rdx	%edx	%dx	%dl
W3	X3	4th argument, callee-owned	%rcx	%ecx	%cx	%cl
W4	X4	5th argument, callee-owned	%r8	%r8d	%r8w	%r8b
W5	X5	6th argument, callee-owned	%r9	%r9d	%r9w	%r9b
W6	X6	7th argument, callee-owned				
W7	X7	8th argument, callee-owned				
W16	X16	Scratch/temporary, callee-owned	%r10	%r10d	%r10w	%r10b
W17	X17	Scratch/temporary, callee-owned	%r11	%r11d	%r11w	%r11b
WSP	SP	Stack pointer, caller-owned	%rsp	%esp	%sp	%spl
W19	X19	Local variable, caller-owned	%rbx	%ebx	%bx	%bl
W20	X20	Local variable, caller-owned	%rbp	%ebp	%bp	%bpl
W21	X21	Local variable, caller-owned	%r12	%r12d	%r12w	%r12b
W22	X22	Local variable, caller-owned	%r13	%r13d	%r13w	%r13b
W23	X23	Local variable, caller-owned	%r14	%r14d	%r14w	%r14b
W24	X24	Local variable, caller-owned	%r15	%r15d	%r15w	%r15b

AArch64与x86部分寄存器对比(2)

Low 32-bits	ARM Register	Conventional use	X86 Register
W25	X25	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W26	X26	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W27	X27	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
W28	X28	Local variable, caller-owned(用于保存跨函数调用长生命周期数据，子函数避免使用)	
	PC(不可直接访问)	Instruction pointer	%rip (可用于地址访问)
	%CPSR	Status/condition code bits	%eflags
W9...W18	X9...X18	Temporary registers (临时寄存器，子函数内可使用)	
	LR(X30)	The Link Register.	
	FP(X29)	The Frame Pointer	
WZR	XZR	Zero-register(read only)	
		Code Segment	CS
		Default Data Segment	DS
		Stack Segment	SS
		Extra Data Segment	ES
		Additional Data Segment	FS
		Additional Data Segment	GS

状态寄存器

AArch64	Flag Name	含义	X86
CF	Carry Flag	进位/借位Carry。如果算术指令的结果的最高有效位产生了进位/借位，则置1；否则置0。这个标志位指示了无符号整数计算的结果是否溢出	CF
	Parity Flag	奇偶校验位Parity。如果指令结果的最低字节含有偶数个比特1，则置1；否则置0。	PF
	Auxiliary Flag	辅助进位/借位Auxiliary Carry。如果算术运算在第3比特位（最低比特位是0）产生了进位/借位，则置1；否则置0。这个标志主要用于二进制编码的十进制数BCD算术运算。	AF
ZF	Zero Flag	零标志位zero。如果指令结果为0，则置1；否则置0	ZF
NF(negative)	Sign Flag	符号位Sign。设置为指令结果的最高有效比特位，即有符号整数的符号比特位。0表示结果是正数或者是0；1表示负数。	SF
VF	Overflow Flag	溢出位Overflow。如果整数运算结果超出了目标操作数可容纳的值域（正数过大，负数过小，不包括符号位），则置1；否则置0（即结果可信）。这个标志指示了有符号整数算术运算的结果是否溢出。	OF
	DF Direction Flag		

ARM寻址方式(1)

- 寻址就是找到存储数据或指令的地址，寻址方式的方便与快捷是衡量CPU性能的一个重要方面，ARM处理器共有八种寻址方式：
 - 立即数寻址
 - 寄存器寻址
 - 寄存器间接寻址
 - 基址寻址
 - 多寄存器寻址
 - 堆栈寻址
 - 相对寻址
 - 寄存器移位寻址

ARM寻址方式(2)

- 立即数寻址：
 - 立即数寻址指令中的地址码就是操作数本身，可以立即使用的操作数。其中，#0xFF000和#64都是立即数。如操作数是常量，用#表示常量；0x或&表示16进制数，否则表示十进制数。

例如：

MOV R0,#0xFF000

@指令省略了第1个操作数寄存器。将立即数0xFF000(第2操作数)装入R0寄存器

SUB R0,R0,#64 @R0减64，结果放入R0

ARM寻址方式(3)

- 寄存器寻址：
 - 操作数的值在寄存器中，指令执行时直接取出寄存器值来操作，寄存器寻址是根据寄存器编码获取寄存器内存储的操作数

例如：

MOV R1,R2

@将R2的值存入R1 在第1个操作数寄存器的位置存放R2编码

SUB R0,R1,R2

@将R1的值减去R2的值，结果保存到R0在第2操作数位置，存放的是寄存器R2的编码

即： $R0 = R1 - R2$

ARM寻址方式(4)

- 寄存器间接寻址：
 - 操作数从寄存器所指向的内存中取出，寄存地存储的是内存地址

例如：

LDR R1,[R2]

@将R2指向的存储单元的数据读出，保存在R1中。R2相当于指针变量

STR R1,[R2]

@将R1的值写入到R2所指向的内存

SWP R1,R1,[R2]

@将寄存器R1的值和R2指定的存储单元的内容交换

[R2]表示寄存器所指向的内存

LDR 指令用于读取内存数据

STR 指令用于写入内存数据

ARM寻址方式(5)

- **基址变址寻址：**

- 基址寄存器的内容与指令中的偏移量相加，得到有效操作数的地址，然后访问该地址空间，基址变址寻址分为三种：

- 前索引，例如：

LDR R0, [R1,#4] @R1存的地址+4，访问新地址里面的值，放到R0；

- 自动索引，例如：

LDR R0, [R1,#4]! @在前索引的基础上，将新地址回写到R1；(注：!表示回写地址)

- 后索引，例如：

LDR R0, [R1],#4 @将R1指向的存储单元内容装载到R0，R1存的地址+4再写入R1；

ARM寻址方式(6)

- 多寄存器寻址：

- 一条指令完成多个寄存器的传送，最多16个寄存器，也称为**块拷贝寻址**

例如：

LDMIA R1!, {R2-R7,R12}

@将R1指向的存储单元中的数据读写到R2 ~ R7、R12中，然后R1自加1

STMIA R1!, {R2-R7,R12}

@将寄存器R2 ~ R7、R12的值保存到R1指向的存储单元中，然后R1自加1

注：基址寄存器不允许为R15，寄存器列表可以为R0 ~ R15 的任意组合。这里R1没有写成[R1]!，是因为这个位不是操作数位，而是寄存器位

LDMIA 和 STMIA 是块拷贝指令，LDMIA是从R1所指向的内存中读数据，STMIA是向R1所指向的内存写入数据

- 执行这类指令要考虑如下几个问题：

- 基址寄存器指向原始地址有没有放一个有效值
 - 寄存器列表哪个寄存器被最先传送
 - 存储器地址增长方向
 - 指令执行完成后，基址寄存器有没有指向一个有效值

ARM寻址方式(7)

- 寄存器堆栈寻址：

- 是按特定顺序存取存储区，按后进先出原则，使用专门的寄存器SP（堆栈指针）指向一块存储区

例如：

LDMIA SP!,{R2-R7,R12}

@将栈内的数据，读写到R2～R7、R12中，然后下一个地址成为栈顶

STMIA SP!,{R2-R7,R12}

@将寄存器R2～R7、R12的值保存到SP指向的栈中，SP指向的是栈顶

ARM寻址方式(8)

- 相对寻址:
 - 即读取指令本身在内存中的地址。是相对于PC内指令地址偏移后的地址。
由**程序计数器PC提供基准地址**，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址

例如：

BL ROUTE1

@调用到 ROUTE1 子程序

BEQ LOOP

@条件跳转到 LOOP 标号处

...

LOOP MOV R2,#2

...

ROUTE1

...

目录

1. ARM处理器简介
2. 鲲鹏处理器
3. ARM寻址方式
- 4. ARM指令集**

ARM指令集

- GNU ARM汇编语言语法格式：

[<label>]:[<instruction or directive or pseudo-instruction>} @comment

- instruction 指令
- directive 伪操作
- pseudo-instruction 伪指令
- <label>: 为标号, GNU ARM汇编中, 任何以冒号结尾的标识符都被认为是一个标号, 而不一定非要在一行的开始
- comment 为语句的注释

注意：

- ARM指令, 伪指令, 伪操作, 寄存器名可以全部为大写字母, 也可全部为小写字母, 但不可大小写混用。
- 如果语句太长, 可以将一条语句分几行来书写, 在行末用 “\” 表示换行 (即下一行与本行为同一语句), “\” 后不能有任何字符, 包含空格和制表符 (Tab)。

ARM指令集

- GNU ARM汇编语言语法格式：
 - 局部变量定义的语法格式： **N{routname}**
 - N：为0~99之间的数字。
 - routname：当前局部范围的名称（为符号），通常为该变量作用范围的名称(用ROUT伪操作定义的)
 - 局部变量引用的语法格式： **%{F|B}{A|T}N{routname}**
 - %：表示引用操作
 - N：为局部变量的数字号
 - routname：为当前作用范围的名称（用ROUT伪操作定义的）
 - F：指示编译器只向前搜索
 - B：指示编译器只向后搜索
 - A：指示编译器搜索宏的所有嵌套层次
 - T：指示编译器搜索宏的当前层次

ARM指令集

- GNU ARM汇编语言语法格式：
 - GNU ARM汇编特殊字符和语法
 - 代码行中的注释符号: '@'
 - 整行注释符号: '#'
 - 语句分离符号: ;'
 - 立即数前缀: '#' 或 '\$'

ARM指令集

- 指令分类：重点掌握红色指令类型

指令类型	说明
跳转指令	条件跳转、无条件跳转 (#imm、register) 指令
异常产生指令	系统调用类指令 (SVC、HVC、SMC)
系统寄存器指令	读写系统寄存器，如：MRS、MSR指令 可操作PSTATE的位段寄存器
数据处理指令	包括各种算数运算、逻辑运算、位操作、移位 (shift) 指令
load/store内存访问指令	load/store {批量寄存器、单个寄存器、一对寄存器、非-暂存、非特权、独占} 以及load-Acquire、store-Release指令 (A64没有LDM/STM指令)
协处理器指令	A64没有协处理器指令

ARM指令集

- A64指令特点：

A64指令编码宽度**固定32bit**

31个（X0-X30）个64bit通用用途寄存器（用作32bit时是W0-W30），寄存器名使用5bit编码

PC指针不能作为数据处理指或load指令的目的寄存器，X30通常用作LR

移除了批量加载寄存器指令 LDM/STM, PUSH/POP, 使用STP/LDP 一对加载寄存器指令代替

增加支持未对齐的load/store指令立即数偏移寻址，提供非-暂存LDNP/STNP指令，不需要hold数据到cache中

没有提供访问CPSR的单一寄存器，但是提供访问PSTATE的状态域寄存器

相比A32少了很多条件执行指令，只有条件跳转和少数数据处理这类指令才有条件执行.

支持48bit虚拟寻址空间

大部分A64指令都有32/64位两种形式

A64没有协处理器的概念

ARM指令集

- 跳转指令：
 - 条件跳转

指令	说明
B.Cond	cond为真跳转
CBNZ	CBNZ X1, label //如果X1!=0则跳转到label
CBZ	CBZ X1, label //如果X1==0则跳转到label
TBNZ	TBNZ X1, #3 label //若X1[3]!=0,则跳转到label
TBZ	TBZ X1, #3 label //若X1[3]==0,则跳转到label

ARM指令集

- 跳转指令：
 - 绝对跳转

指令	说明
B	绝对跳转
BL	绝对跳转 #imm，返回地址保存到LR (x30)
BLR	绝对跳转reg，返回地址保存到LR (x30)
BR	跳转到reg内容地址，
RET	子程序返回指令，返回地址默认保存在LR (x30)

例如：

BL func

@调用子程序func

...

func

...

MOV R15,R14

@子程序返回

ARM指令集

- 异常产生和返回指令：

指令	说明
SVC	SVC系统调用，目标异常等级为EL1
HVC	HVC系统调用，目标异常等级为EL2
SMC	SMC系统调用，目标异常等级为EL3
ERET	异常返回，使用当前的SPSR_ELx和ELR_ELx

ARM指令集

- 系统寄存器指令：

指令	说明
MRS	R <- S: 通用寄存器 <= 系统寄存器
MSR	S <- R: 系统寄存器 <= 通用寄存器

例如：

MRS R0,CPSR
存器R0中

@状态寄存器CPSR的值存入寄

MSR CPSR_f,R0
域

@用R0的值修改CPSR的条件标志

MSR CPSR_fsxc,#5

@CPSR的值修改为5

ARM指令集

- 数据处理指令：

数据处理指令类型					
算数运算	逻辑运算	数据传输	地址生成	位段移动	移位运算
ADDS	ANDS	MOV	ADRP	BFM	ASR
SUBS	EOR	MOVZ	ADR	SBFM	LSL
CMP	ORR	MOVK		UBFM	LSR
SBC	MOVI			BFI	ROR
RSB	TST			BFXIL	
RSC				SBFIZ	
CMN				SBFX	
MADD				UBFIZ	
MSUB					
MUL					
SMADDL					
SDIV					
UDIV					

ARM指令集

- 算术运算指令：

指令	说明
ADDS	加法指令，若S存在，则更新条件位flag
ADCS	带进位的加法，若S存在，则更新条件位flag
SUBS	减法指令，若S存在，则更新条件位flag
SBC	将操作数1 减去操作数2，再减去标志位C的取反值，结果送到目的寄存器Xt/Wt
RSB	逆向减法，操作数2 - 操作数1，结果 Rd
RSC	带借位的逆向减法指令，将操作数2 减去操作数1，再减去标志位C的取反值，结果送目标寄存器Xt/Wt
CMP	比较相等指令
CMN	比较不等指令
NEG	取负数运算，NEG X1, X2 // X1 = X2按位取反+1 (负数=正数补码+1)
MADD	乘加运算
MSUB	乘减运算
MUL	乘法运算
SMADDL	有符号乘加运算
SDIV	有符号除法运算
UDIV	无符号除法运算

ARM指令集

- 算术运算指令：

例如：

ADD 加法指令

ADD R0,R1,#5

@R0=R1+5

ADD R0,R1,R2

@R0=R1+R2

ADD R0,R1,R2,LSL #5

@R0=R1+R2左移5位

SUB 减法指令

SUB R0,R1,R2

@R0=R1-R2

SUB R0,R1,R2,LSL #5

@R0=R1-R2左移5位

ARM指令集

- 逻辑运算指令：

指令	说明
ANDS	按位与运算, 如果S存在, 则更新条件位标记
EOR	按位异或运算
ORR	按位或运算
TST	例如: TST W0, #0X40 //指令用来测试W0[3]是否为1, 相当于: ANDS WZR,W0, #0X40

ARM指令集

- 逻辑运算指令：

例如：

AND逻辑与指令

AND R0,R0,#5

@保持R0的第0位和第2位，其余位清0

ORR逻辑或指令

ORR R0,R0,#5

@R0的第0位和第2位设置为1，其余位不变

EOR逻辑异或指令

EOR R0,R0,#5

@R0的第0位和第2位取反，其余位不变

ARM指令集

- 数据传输指令：

指令	说明
MOV	赋值运算指令
MOVZ	赋值#uimm16到目标寄存器xd
MOVN	赋值#uimm16到目标寄存器xd，再取反
MOVK	赋值#uimm16到目标寄存器xd，保存其它bit不变

ARM指令集

- 数据传输指令：

例如：

MOV 数据传送指令

MOV R0,#0xFF000

@立即寻址，将立即数0xFF000(第2操作数)装入R0寄存器

MOV R1,R2

@寄存器寻址，将R2的值存入R1

MOV R0,R2,LSL #3

@移位寻址，R2的值左移3位，结果放入R0

MVN 数据取反传送指令

MVN R0,#0;R0=-1

ARM指令集

- 地址生成指令：

指令	说明
ADRP	base = PC[11:0]=ZERO(12); Xd = base + label;
ADR	Xd = PC + label

ARM指令集

- 位段移动指令：

指令	说明
BFM	
SBFM	BFM Wd, Wn, #r, #s if $s \geq r$ then $Wd[s-r:0] = Wn[s:r]$, else $Wd[32+s-r:32-r] = Wn[s:0]$.
UBFM	
BFI	
BFXIL	
SBFIZ	
SBFX	
UBFX	
UBFZ	

ARM指令集

- 移位运算指令：

指令	说明
ASR	算术右移 >> (结果带符号)
LSL	逻辑左移 <<
LSR	逻辑右移 >>
ROR	循环右移：头尾相连
SXTB	
SXTH	
SXTW	字节、半字、字符号/0扩展移位运算 关于SXTB #imm和UXTB #imm 的用法可以使用以下图解描述：
UXTB	
UXTH	

ARM指令集

- Load/Store指令：

Load/Store指令							
对齐偏移	非对齐偏移	PC-相对寻址	访问一对	非暂存	非特权	独占	Acquire Release
LDR	LDUR	LDR	LDP	LDNP	LDTR	LDXR	LDAR
LDRB	LDURB	LDRSW	LDRSW	STNP	LDTRB	LDXRB	LDARB
LDRSB	LDURSB		STP		LDTRSB	LDXRH	LDARH
LDRH	LDURH				LDTRH	LDXP	STLR
LDRSH	LDURSH				LDTRSH	STXR	STLRB
LDRSW	LDURSW				LDTRSW	STXRB	STLRH
STR	STUR				STTR	STXRH	LDAXR
STRB	STURB				STTRB	STXP	LDAXB
STRH	STURH				STTRH		LDAXRH
							LDAXP
							STLXR
							STLXR
							STLXRH
							STLXP

AArch64指令集 vs. x86-64指令集（1）

- ARM指令集是固定大小，固定格式的指令编码。指令长度32bit位宽，4字节对齐；
- X86体系结构是可变长度指令集体系结构。指令地址没有对齐要求。

主要差异列举如下：

- 1. ARM是一种load-store架构

在RISC体系结构中很常见，这意味着数据处理指令不能直接对内存的内容进行操作，它们仅对寄存器进行操作。反过来，加载和存储指令只能在寄存器和内存之间传输数据。相比之下，x86-64指令集的数据处理指令可以直接在内存以及寄存器上处理数据。

x86-64	AArch64
addq %rax, 16(%rdi) # (%rdi+16)+%rax →(%rdi+16)	ldr x2, [x3, #16] ; (x3+16) → x2 add x0, x1, x2 ; x1+x2 → x0 str x0, [x3, #16] ; x0 → (x3+16)

- 2. x86-64支持访问I/O地址空间的单独I/O指令

X86的指令集包括一部分指令可以直接对IO地址空间进行操作。例如IN或者OUT，直接对I/O端口进行数据读写。ARM没有等效功能，而是假定所有外围设备都在标准4GB地址空间内映射到内存里的。

AArch64指令集 vs. x86-64指令集（2）

- **3. 无法在ARM指令中嵌入任意32位地址**

因为ARM指令是固定32bit宽度的，所以不可能直接在arm指令中编码进32bit的地址。所有的内存访问，都是基于存放在一个寄存器中的地址进行索引的。换句话说，所有ARM的内存访问都是通过通用寄存器间接进行的。由于指令集的可变长度性质，在x86-64指令中可以在指令中嵌入32位地址。x86可以进行直接地址访问。

- **4. ARM指令不能包含任意的32位常量（ARM不能直接生成32bit宽度的立即数）**

同上的原因。ARM操作长度较长的立即数时，需要通过mov/movk多次进行生成。

- **5. IA-32使用分段寻址模型**

所有IA-32内存/存储器访问都相对于段寄存器之一，因此必须首先设置这些访问。较大的偏移量需要较大的指令才能对较大的常数进行编码。ARM没有分段寻址的概念，并且没有等效的分段寄存器。

DS: Offset(base,index,scale) → offset(base)

AArch64指令集 vs. x86-64指令集（3）

- 6. ARM指令通常都有3个操作数；x86 指令存在隐含操作数（cpuid mul）

大多数ARM数据处理指令采用三个操作数，所有这些操作数可以是寄存器，其中之一可以是常量。相反，x86-64指令通常具有只有两个操作数，从而使它们本质上具有破坏性（即，其中一个操作数被结果覆盖）。这使ARM在指令级别上更加灵活。

例如，可以将两个寄存器加在一起并将结果放在第三个寄存器中使用一条指令。相同的操作在x86-64上需要两条指令。

x86-64	AArch64
movq %rcx, %rax ; %rcx → %rax addq %rbx, %rax ; %rax+%rbx → %rax	add x0, x1, x2 ; x0 = x1 + x2

- 7. 许多复杂的IA-32指令都没有与ARM直接对应的指令

因为x86的芯片电路相对设计复杂，可以支持较多复杂操作的指令。例如BCD转换和操作，点积，三角函数和对数函数等。

AArch64指令集 vs. x86-64指令集（4）

- **8. ARM子例程通常不使用stack**

ARM BL/B/B.cond 指令（用于子例程调用），是将返回地址放置在链接寄存器LR，而不是把地址放到栈里。同样，子例程返回之后的返回地址，是从LR里返回，而不是存储在栈上的。

x86 call 和Ret 指令，将返回地址放在堆栈上并分别取出。此外，提供了ENTER和LEAVE指令，以在进入和离开过程时创建和释放固定格式的堆栈帧。ARM没有与此等效的方法，因为从堆栈指针中添加固定值或从中减去固定值非常简单。

- **9. x86在被零除错误时产生异常**

ARM UDIV和SDIV指令不会检测被零除的情况，并且始终返回零结果。(编译器高级别优化如-O2可能会检测到除零的情况，才生成brk指令，运行时报错)

- **10. ARM不提供数组绑定检查指令**

在x86中，BOUND指令用于在执行潜在的无效访问之前针对数组边界检查内存地址。ARM指令集无等效指令。