




举例说明CPU指令集设计的 依据


邱吉 <qiuji.odyssey@gmail.com>



2023年10月28日 南盘江计划

为什么会有 这期课程

 2023年9月28日 09:40
有没有分享指令集设计依据的，感觉只要加减乘除指令就够了，其他指令为什么要加进来

 qjivy 2023年9月28日 09:54
回复  因为随着半导体制程规格越来越小，晶体管更充足了，可以把很多复杂运算都合并起来，做一条指令，这样写软件的人和写编译器的人就得到了解放。（硬件强，软件就可以相对省事）

 2023年9月28日 11:15
回复 qjivy: 希望可以分享一节指令集设计的依据啊，好像还没看到有人分享，虽然不知道也不影响编程，因为是硬件固定死的，类似函数库

 qjivy 2023年9月28日 15:31
回复 : 好呀，已经加入wishlist

我们从哪里开始：以前的一期课程



提纲

- 背景概念
- 简明体系结构入门
 - 指令集架构
 - 微体系结构
- 指令集架构对编译器后端的影响
- 微体系结构对编译器后端的影响
- 其他的Tips
- 总结

指令集体系结构 (比如: RISC-V, X86, ARM64, PPC)	CISC/RISC 地址宽度 (32/64bit) 寄存器宽度 (32/64/128/256/512bit) 指令集 (语义和编码) 寄存器组 (GPR/FPR/VR, 控制/状态寄存器, FLAG Bits) 数据类型 (Int/FP) 寻址模式 ABI (传参, 返回值, 内存对齐, 数据类型约定)
微体系结构 (比如: 每一个具体的CPU IP 或者芯片, 都有它的微体系结构参数)	发射宽度 Function Unit类型和数量 Pipeline延迟 存储层次中的Cache组织
系统架构 (比如: 每一个开发板、每一台计算机, 每一个集群系统都会有这些参数)	内存组织 多核, 大小核 多芯片互联 集群 网络互联

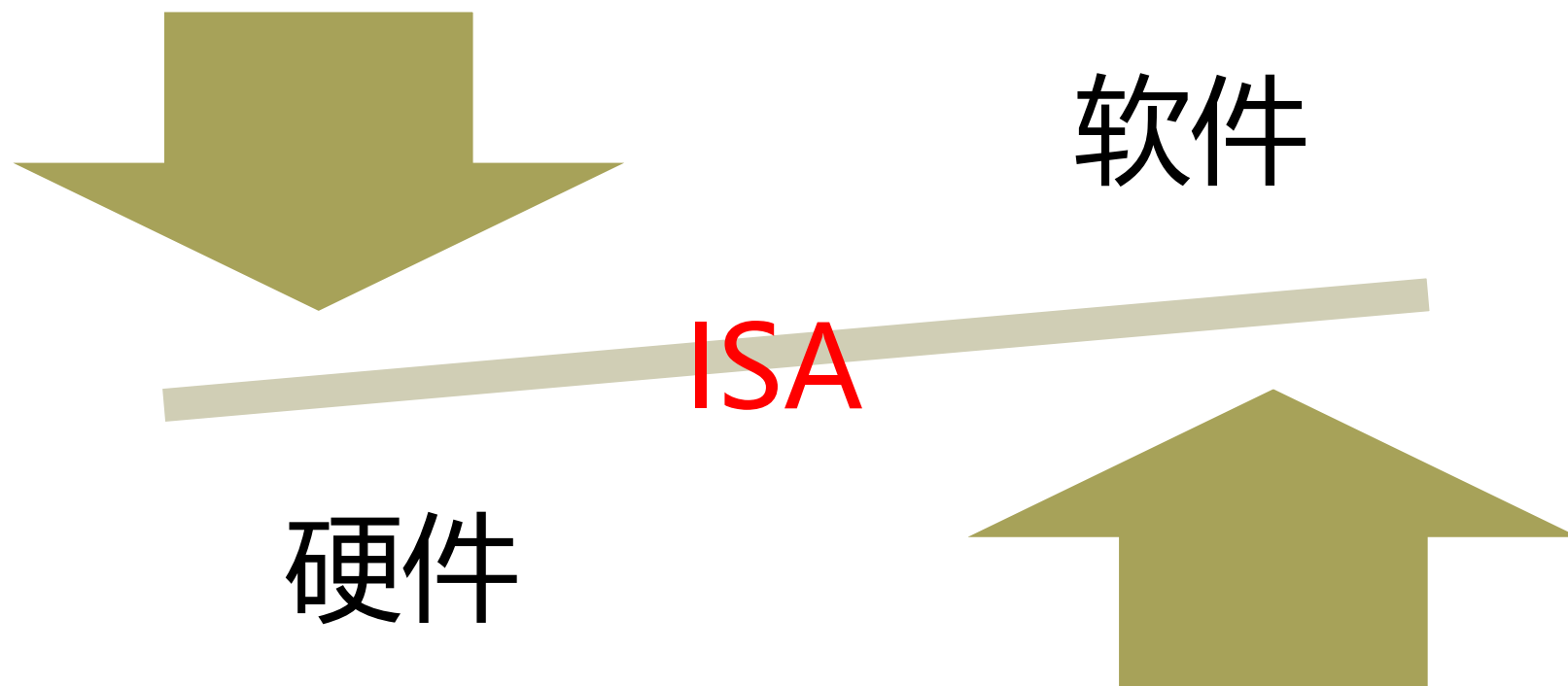
今天的内容

- CPU指令集是什么
- 来自软件的需求
 - 通用指令集
 - 专用扩展
- 来自硬件和编译器实现的需求
 - 二进制编码
 - 高效的硬件实现
 - 模块化的可裁剪性

今天的内容

- CPU指令集是什么
- 来自软件的需求
 - 通用指令集
 - 专用扩展
- 来自硬件和编译器实现的需求
 - 二进制编码
 - 高效的硬件实现
 - 模块化的可裁剪性

CPU指令集是什么？ 软件和硬件之间的接口



软件看到的是什​​么？

```
#include <stdio.h>
#include <conio.h>
int main()
{
    printf("Hello,World"); // Displaying as Output
    getch();
    return 0;
}
```

```
int a, b, c, d;
__asm__ (
    "add %0, %1, %2"
    //    c = a + b
    : "r"(c)
    : "+r"(a), "+r"(b)
    )
```

```
int a, b, c, d;
__asm__ (
    "add %[c2], %[a2], %[b2]"
    //    c = a + b
    : [c2]"r"(c)
    : [a2]"r"(a), [b2]"r"(b)
    )
```

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::atomic_int acnt;
int cnt;

void f()
{
    for (int n = 0; n < 10000; ++n)
    {
        ++acnt;
        ++cnt;
        // Note: for this example, relaxed memory order
        // is sufficient, e.g. acnt.fetch_add(1, std::memory_order_relaxed);
    }
}

int main()
{
    {
        std::vector<std::jthread> pool;
        for (int n = 0; n < 10; ++n)
            pool.emplace_back(f);
    }

    std::cout << "The atomic counter is " << acnt << '\n'
              << "The non-atomic counter is " << cnt << '\n';
}
```

硬件看到的是什么？

```
#include <stdio.h>
#include <conio.h>
int main()
{
    printf("Hello,World"); // Displaying as Output
    getch();
    return 0;
}
```

out/store到外设端口或显存

```
int a, b, c, d;
__asm__ (
    "add %0, %1, %2"
    //    c = a + b
    : "r"(c)
    : "+r"(a), "+r"(b)
    )
```

```
int a, b, c, d;
__asm__ (
    "add %[c2], %[a2], %[b2]"
    //    c = a + b
    : [c2]"r"(c)
    : [a2]"r"(a), [b2]"r"(b)
    )
```

add指令

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>
```

内存原子操作指令

```
std::atomic_int acnt;
int cnt;

void f()
{
    for (int n = 0; n < 10000; ++n)
    {
        ++acnt;
        ++cnt;
        // Note: for this example, relaxed memory order
        // is sufficient, e.g. acnt.fetch_add(1, std::memory_order_relaxed);
    }
}

int main()
{
    {
        std::vector<std::jthread> pool;
        for (int n = 0; n < 10; ++n)
            pool.emplace_back(f);
    }

    std::cout << "The atomic counter is " << acnt << '\n'
              << "The non-atomic counter is " << cnt << '\n';
}
```

最终硬件看到的是什么？

- 01010101.... 二进制码流
- 根据规则，以1byte, 2byte, 4byte等长度为单元去解析
- 然后触发相应的电路
- 执行算术、逻辑、位运算，分支跳转，访存读写，陷井例外。。。

今天的内容

- CPU指令集是什么
- 来自软件的需求
 - 通用指令集
 - 专用扩展
- 来自硬件和编译器实现的需求
 - 二进制编码
 - 高效的硬件实现
 - 模块化的可裁剪性

来自软件的需求-干活

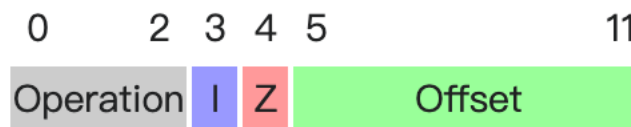
- 跑纸带程序
- 跑操作系统
- 外设输入输出
- 多核并行
- 虚拟化
- 图形图像加解密加速 (SIMD/向量/密码)
- . . .

每一种命令的下达，都需要一个二进制编码来区分

编码数量= $\log_2(\text{命令种类})$

最小指令集的例子—Dec的PDP8

指令集只包含8条操作



Basic instructions [\[edit \]](#)

- 000 — AND — [AND](#) the memory operand with AC.
- 001 — TAD — [Two's complement ADd](#) the memory operand to <L,AC> (a *12 bit* signed value (AC) w. carry in L).
- 010 — ISZ — Increment the memory operand and Skip next instruction if result is Zero.
- 011 — DCA — Deposit AC into the memory operand and Clear AC.
- 100 — JMS — JuMp to Subroutine (storing [return address](#) in first word of subroutine!).
- 101 — JMP — JuMP.
- 110 — IOT — Input/Output Transfer (see below).
- 111 — OPR — microcoded OPeRations (see below).

程序员/编译器的高超“合成”技艺

Optimized logical 'XOR' using “carryless addition”

```
CLA          / clear accumulator (AC) since all we have is add!
TAD  ArgOne  / add (TAD) ArgOne to the just-zeroed AC
AND  ArgTwo  / AND ArgTwo to determine where the carries will be
CLL RAL      / clear the LINK (CLL) and rotate the accumulator left (RAL)
CMA IAC      / compliment (CMA) & increment (IAC) the accumulator (negate)
TAD  ArgOne  / add the first argument to the negated accumulator
TAD  ArgTwo  / and add the second argument as well
              / the accumulator now contains the XOR of ArgOne & ArgTwo
```

The “carryless addition” code shown above efficiently implements the following algebraic expression:

$$\text{ArgOne} + \text{ArgTwo} - 2 * (\text{ArgOne AND ArgTwo})$$

程序员/编译器的高超“合成”技艺

晶体管很贵很少的时代，
程序员必须身怀绝技

Optimized logical 'XOR' using “carryless addition”

```
...tor (AC) since all we have is add!  
...the just-zeroed AC  
...the carries will be  
...the accumulator left (RAL)  
...accumulator (negate)  
...tor
```

The “carryless addition” code shown above efficiently

ArgOne + ArgTwo

<https://www.grc.com/dp-8/isp-musings.htm>

一个能完备支持 应用程序的例子： RISC-V 32基础指 令集只有40条指令

- *RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments*
- *RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity)*

2023/10/30

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20 10:1 11 19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12 10:5]		rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]		rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]		rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]		rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[12 10:5]		rs2	rs1	110	imm[4:1 11]	1100011	BLTU
imm[12 10:5]		rs2	rs1	111	imm[4:1 11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

再加上跑Linux操作系统呢？

Trap-Return Instructions

0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET

Interrupt-Management Instructions

0001000	00101	00000	000	00000	1110011	WFI
---------	-------	-------	-----	-------	---------	-----

Supervisor Memory-Management Instructions

0001001	rs2	rs1	000	00000	1110011	SFENCE.VMA
0001011	rs2	rs1	000	00000	1110011	SINVAL.VMA
0001100	00000	00000	000	00000	1110011	SFENCE.W.INVALID
0001100	00001	00000	000	00000	1110011	SFENCE.INVALID.IR

但为什么还要有标准扩展



RISC-V Base Plus Standard Extensions

- Four base integer ISAs
 - RV32E, RV32I, RV64I, RV128I
 - RV32E is 16-register subset of RV32I
 - Only <50 hardware instructions needed for base
- Standard extensions
 - M: Integer multiply/divide
 - A: Atomic memory operations (AMOs + LR/SC)
 - F: Single-precision floating-point
 - D: Double-precision floating-point
 - G = IMAFD, "General-purpose" ISA
 - Q: Quad-precision floating-point
- Above use standard RISC encoding in fixed 32-bit instruction word
- Frozen in 2014, ratified 2019, supported forever after

程序员/编译器的高超“合成”技艺

- 因为硅片上的晶体管密度、芯片主频以摩尔定律发展
- 硬件可以更高效地实现更多不同的功能的电路
- 而软件模拟的方式太低效

The “carryless addition” code

<https://www.grc.com/dp-8/isp-musings.htm>

反例：软件合成的浮点操作

Libgcc简介

Libgcc是GCC的低级运行时库，当一些运算在特定平台上不支持时，GCC 会自动生成对libgcc库函数的调用，使用库函数中的定义来模拟运算实现。

Libgcc定义了一组标准的运算函数，不同目标机器可以对其进行特定的指令实现，来支持常见的运算处理功能。

官方介绍：<https://gcc.gnu.org/onlinedocs/gccint/Libgcc.html#Libgcc>

代码地址：<https://gcc.gnu.org/git/?p=gcc.git;a=tree;f=libgcc>

RISC-V浮点操作调用Libgcc

在RISC-V中，如果没有启用F/D扩展时使用了浮点运算，就会调用Libgcc中定义的库函数，通过库函数完成浮点运算支持

```
func:
.file "fadd.c"
.option nopic
.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_c2p0_zicsr2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.align 1
.globl func
.type func, @function

addi    sp,sp,-32
sd      s0,24(sp)
addi    s0,sp,32
sw      a0,-20(s0)
sw      a1,-24(s0)
flw     fa4,-20(s0)
flw     fa5,-24(s0)
fadd.s  fa5,fa4,fa5
fmv.x.s a0,fa5
ld      s0,24(sp)
addi    sp,sp,32
jr      ra
.size   func,.-func
.ident  "GCC: (GNU) 13.0.0 20220829 (experimental)"
```

```
func:
.file "fadd.c"
.option nopic
.attribute arch, "rv64i2p1_m2p0_a2p1_c2p0_zicsr2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.globl __addsf3
.align 1
.globl func
.type func, @function

addi    sp,sp,-32
sd      ra,24(sp)
sd      s0,16(sp)
addi    s0,sp,32
sw      a0,-20(s0)
sw      a1,-24(s0)
lw      a1,-24(s0)
lw      a0,-20(s0)
call    __addsf3
mv      a5,a0
mv      a0,a5
ld      ra,24(sp)
ld      s0,16(sp)
addi    sp,sp,32
jr      ra
.size   func,.-func
.ident  "GCC: (GNU) 13.0.0 20220829 (experimental)"
```


RISC-V浮点操作调用Libgcc

库函数由一系列汇编指令组成，通过宏定义来生成汇编代码

```
#include "soft-fp.h"
#include "single.h"

SFtype
__addsf3 (SFtype a, SFtype b)
{
    FP_DECL_EX;
    FP_DECL_S (A);
    FP_DECL_S (B);
    FP_DECL_S (R);
    SFtype r;

    FP_INIT_ROUNDMODE;
    FP_UNPACK_SEMIRAW_S (A, a);
    FP_UNPACK_SEMIRAW_S (B, b);
    FP_ADD_S (R, A, B);
    FP_PACK_SEMIRAW_S (r, R);
    FP_HANDLE_EXCEPTIONS;

    return r;
}
```

```
00000000001055a <__addsf3>:
1055a: 00800737      lui      a4,0x800
1055e: 1101         addi     sp,sp,-32
10560: 177d         addi     a4,a4,-1 # 7ffffff
10562: 00a777b3     and      a5,a4,a0
10566: 0175d61b     srliw    a2,a1,0x17
1056a: 8f6d         and      a4,a4,a1
1056c: e822         sd       s0,16(sp)
1056e: 0175541b     srliw    s0,a0,0x17
10572: 0ff47413     zext.b   s0,s0
10576: 0ff67613     zext.b   a2,a2
1057a: 1782         slli     a5,a5,0x20
1057c: 1702         slli     a4,a4,0x20
1057e: e426         sd       s1,8(sp)
10580: 9381         srli     a5,a5,0x20
10582: 01f5549b     srliw    s1,a0,0x1f
10586: 9301         srli     a4,a4,0x20
10588: 40c4083b     subw     a6,s0,a2
1058c: ec06         sd       ra,24(sp)
1058e: e04a         sd       s2,0(sp)
10590: 01f5d59b     srliw    a1,a1,0x1f
10594: 00379693     slli     a3,a5,0x3
```

所以需要定义General-purpose ISA



RISC-V Base Plus Standard Extensions

- Four base integer ISAs
 - RV32E, **RV32I**, RV64I, RV128I
 - RV32E is 16-register subset of RV32I
 - Only <50 hardware instructions needed for base
- Standard extensions
 - **M**: Integer multiply/divide
 - **A**: Atomic memory operations (AMOs + LR/SC)
 - **F**: Single-precision floating-point
 - **D**: Double-precision floating-point
 - **G** = **IMAFD+Zicsr+Zifencei**
 - ~~Q: Quad-precision floating-point~~
- Above use standard RISC encoding in fixed 32-bit instruction word
- Frozen in 2014, ratified 2019, supported forever after

专用扩展： 如果定义的操作太多/太复杂会怎么样

- RISC 就逐渐变成了 CISC，同一条指令操作太多（精简就变成了复杂），例子，ARMv7的带桶形移位寄存器的指令
- 指令越来越多,超级复杂：例子，ARMv8
- “Overall, the ISA is complex and unwieldy: there are 1070 instructions, comprising 53 formats and eight data addressing modes [18], all of which takes 5,778 pages to document [9].” --**Design of the RISC-V Instruction Set Architecture, Andrew Waterman**

<https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf>

今天的内容

- CPU指令集是什么
- 来自软件的需求
 - 通用指令集
 - 专用扩展
- 来自硬件和编译器实现的需求
 - 二进制编码
 - 高效的硬件实现
 - 模块化的可裁剪性

指令集体系结构的属性	来自软件的需求和定义	指令操作 CISC/RISC ABI (传参, 返回值, 内存对齐, 数据类型约定)
	来自硬件的需求和定义	地址宽度 (32/64bit) 寄存器宽度 (32/64/128/256/512bit) 指令集编码 寄存器组 (GPR/FPR/VR, 控制/状态寄存器, FLAG Bits) 数据类型 (Int/FP) 寻址模式 尾端

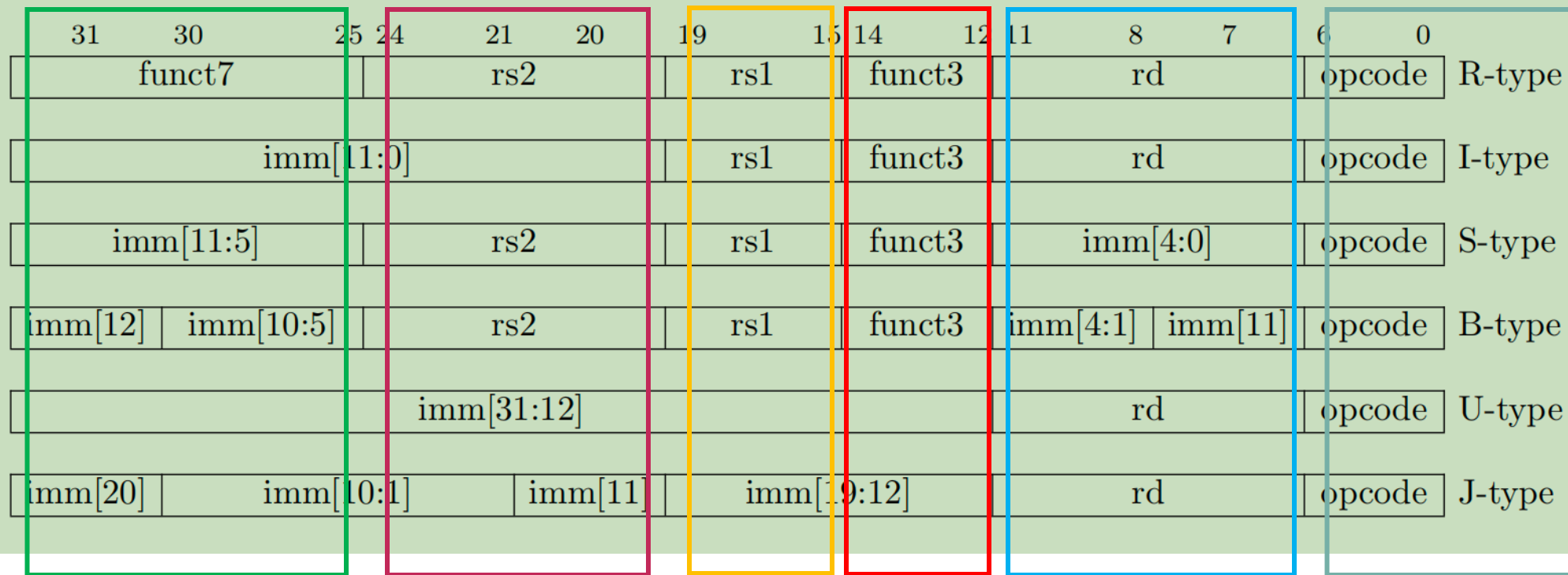
RV32I的例子： 兼顾取指、译码效率的编码方式

1. major opcode位于bit0~7，在小尾端低字节先传下，CPU可以最快获得并译码，指导接下来的解析

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2			rs1		funct3		rd			opcode		R-type	
imm[11:0]						rs1		funct3		rd			opcode		I-type		
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type	
imm[12]		imm[10:5]		rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]										rd			opcode		U-type		
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type		

RV32I的例子： 兼顾取指、译码效率的编码方式

2. rd/rs1/rs2/funct3/funct7以及imm域的划分在不同格式类别的指令之间也是基本对齐的，降低了判断逻辑的开销（延时）



RV32I的例子：

兼顾取指、译码效率的编码方式

3. 寄存器编码域在所有类别指令中一致，读寄存器可以被提前到与译码并行

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0		
funct7			rs2			rs1		funct3		rd			opcode		R-type	
imm[11:0]						rs1		funct3		rd			opcode		I-type	
imm[11:5]			rs2			rs1		funct3		imm[4:0]			opcode		S-type	
imm[12]	imm[10:5]		rs2			rs1		funct3		imm[4:1]	imm[11]		opcode		B-type	
			imm[31:12]							rd			opcode		U-type	
imm[20]		imm[10:1]			imm[11]		imm[19:12]				rd			opcode		J-type

总结：今天的内容

- CPU指令集是什么
- 来自软件的需求
 - 通用指令集
 - 专用扩展
- 来自硬件和编译器实现的需求
 - 二进制编码
 - 高效的硬件实现
 - 模块化的可裁剪性

总结：例子

- 最小指令集的CPU： DEC PDP8 8条指令
- 能支持现代应用程序和编译器的指令集RV32I只有40条指令，再加上7条特权态指令，可以支持Linux系统
- 为什么需要标准扩展M/A/F/D：一个去掉FD的例子，libgcc的用途
- 硬件设计要考虑的方面： RV32I的例子说明如何在兼顾微架构特性进行指令集编码

参考文献

- Design of the RISC-V Instruction Set Architecture, Andrew Waterman, <https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf>
- Instruction Set Architecture Design, <https://minnie.tuhs.org/CompArch/Lectures/week02.html>
- Computer Architecture: A Quantitative Approach, John L. Hennessy , David A. Patterson
- 一个RISC-V的压缩指令集是否应该保留，是否应该加入新的压缩指令集的讨论 <https://lists.riscv.org/g/tech-profiles/topic/101784675>
- 一个RISC-V的新扩展Memory tagging的讨论：
<https://lists.riscv.org/g/tech-j-ext/message/523>

Q&A
Thanks ~