Современные RISC-архитектуры: состояние и технологии

Андрей Белеванцев
ИСП РАН им. В.П. Иванникова
8 сентября 2020 г.

ispras.ru
facebook.com/IvannikovISPRAS
t.me/ispras

План доклада

- Коротко про RISC
- Более длинно про ARM
- Чуть короче про RISC-V
- Как с этим жить

RISC – Reduced Instruction Set Computer

- Stanford MIPS → MIPS (Hennessy)
- Berkeley RISC → SPARC (Patterson)
- IBM Power
- ARM
- RISC-V
- …

John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach. 6th ed., Morgan Kaufmann.

2017 ACM Turing Award.

RISC – Reduced Instruction Set Computer

- Простые операции
  - Ограниченный набор простых команд (например, нет деления)
  - Команда выполняется за один такт
  - Фиксированная длина команды (простота декодирования)
- Конвейер
  - Каждая операция разбивается на однотипные простые этапы, которые выполняются параллельно
  - Каждый этап занимает 1 такт, в т.ч. декодирование
- Регистры
  - Много однотипных взаимозаменяемых регистров (могут использоваться и для данных, и для адресации)

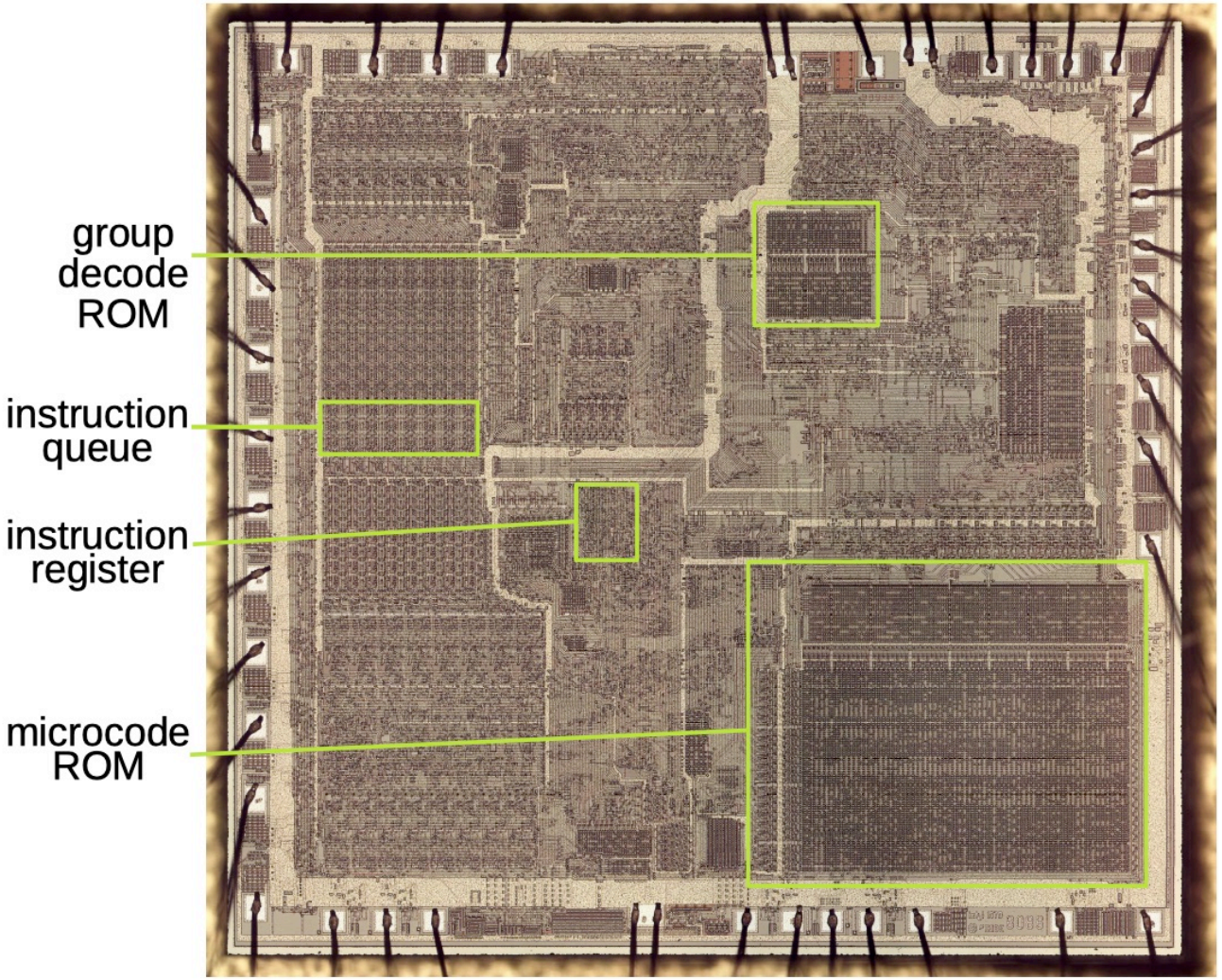
Пример: формат команды RISC-V

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

RISC – Reduced Instruction Set Computer

- Модель работы с памятью
  - Отдельные команды для загрузки/сохранения в память
  - Команды обработки данных работают только с регистрами
- Сложность оптимизаций перенесена из процессора в компилятор
  - Производительность сильно зависит от компилятора
- Итого: более простое ядро, выше частота процессора

RISC внутри CISC



- Фотография процессора 8086 под микроскопом (~29,000 транзисторов)
  - http://www.righto.com/2020/08/latches-inside-reverse-engineering.html
- Микрокод 8086
  - https://patents.google.com/patent/US4363091
  - https://www.reenigne.org/blog/8086-microcode-disassembled/
  - 1-8 микроинструкций на инструкцию
  - Все, кроме самых простых, реализованы в микрокоде
- Pentium и далее
  - Out-of-order, register renaming
  - Возможности обновления

ARM – Advanced (Acorn) RISC Machine

- Разработка ведется с начала 1980-х (Acorn → ARM)
  - Sophie Wilson, Steve Furber
- RISC-ядро + расширяемость
- Arm Holdings: лицензирование архитектуры (IP Core) для последующего создания процессоров, SoC, микроконтроллеров
  - Apple, Qualcomm, Samsung …
- Не только смартфоны
  - Нетбуки – первые модели с 2009 года
  - Десктопы и ноутбуки – Apple переходит на ARM (2020)
  - Суперкомпьютеры – первое место в Top500 (машина Fugaku, процессор Fujitsu A64FX, 2020)

Поколения ARM

Архитек-тура	Разряд ность	Семейство процессоров	Год	Примеры устройств
ARMv1	32 бит	ARM1	1985	
ARMv2	32 бит	ARM2, ARM3		
ARMv3	32 бит	ARM6, ARM7	1992	
ARMv4	32 бит	StrongARM, ARM7TDMI, ARM9TDMI	2003	iPaq 4150
ARMv5	32 бит	ARM7EJ, ARM9E, ARM10E, XScale		
ARMv6	32 бит	ARM11	2007	iPhone (original, 3G)
ARMv7	**32 бит**	**Cortex A8, A9, A15(A7)** **самая распространенная**	**2008**	**N900, Galaxy до S4, iPhone (3GS, 4, 5)**
ARMv8	64 бит	Cortex A53,A57	2011	iPhone (5S, 6), Galaxy S6

- Уже сейчас самая распространенная – ARMv8
- 2010 – 6 млрд, 2012 – 30 млрд, 2019 – 130 млрд ARM-процессоров

32-bit ARM

- Режимы процессора (modes)
  - Пользовательский, привилегированные (гипервизор, системные…)
- Инструкции
  - RISC: 32-битные инструкции (+ Thumb), одинаковые регистры, load/store, в основном 1 такт
  - Дополнительно: условное выполнение, флаги, константы (сдвиги)
- 16 регистров общего назначения (r0-r15)
  - PC (r15) – Program Counter
  - LR (r14) – Link Register
  - SP (r13) – Stack Pointer

32-bit ARM: предикаты и условные флаги

Флаг	Описание	Когда устанавливается
Q	Satura-tion	Насыщение/переполнение в DSP-операциях
V	oVerflow	Переполнение (знаковое)
C	Carry	Перенос бита (беззнаковый)
Z	Zero	Нулевой результат (равенство)
N	Negative	Отрицательное значение (бит 31 установлен)

Предикат	Отрицание	Флаги	Семантика
EQ	**NE**	Z	==
CS / HS	**CC / LO**	C	>= (беззнаковое)
MI	**PL**	N	< 0
VS	**VC**	V	переполнение
HI	**LS**	zC	> (беззнаковое)
GE	**LT**	NV || nv	>= (знаковое)
GT	**LE**	NzV || nzv	> (знаковое)
AL	**-**	-	Безусловная команда

Условное выполнение

Почти все команды ARM могут быть записаны в *условной форме*. В этом случае условие приписывается после команды, и она будет выполнена, только если условие истинно.

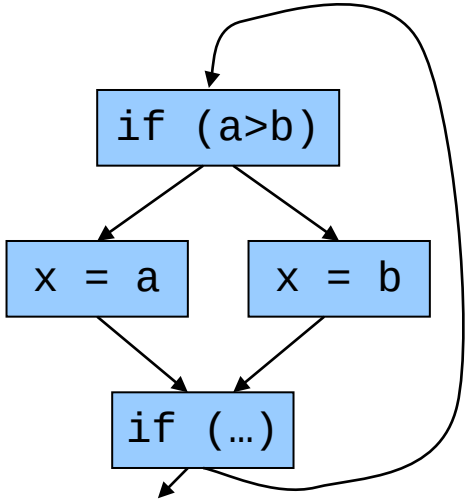
```
addge r1, r1, r1
```

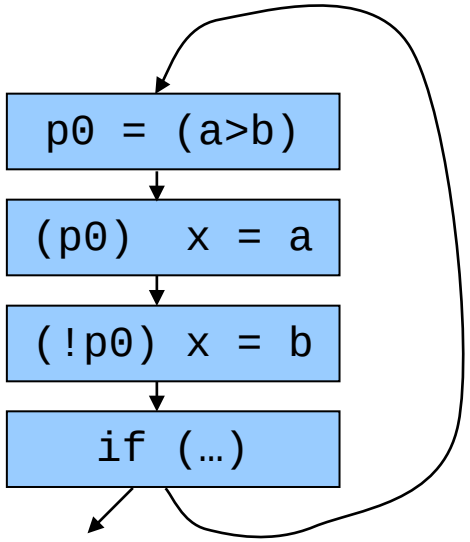
Вычисление модуля |r1 – r2|:

```
subs r1, r1, r2
rsblt r1, r1, #0
```

Преобразование в условную форму

- Команды условного перехода заменяются командами условного выполнения
- Количество ветвлений сокращается, это позволяет:
  - Избежать сбоя конвейера при неправильном предсказании перехода
  - Выполнять конвейеризацию циклов





Модуль сдвига (barrel shifter)

Второй аргумент ALU-команд может быть представлен в виде

ARG2 = R shift_op B, где

R – регистр
B – величина сдвига (0-31)
shift_op – один из LSL, ASL, RSL, ROR или RRX

`add r1, r1, r1 lsl #2`// *r1 = r1 + r1 *4 = r1 * 5*

Примеры «хитрых» арифметических команд

1. Вычислить X = X * 81 в две команды без использования MUL

Решение:

81 = 9^2 = (8 + 1)^2 [или 81 = 5 * 16 + 1]

```
add r1, r0, r0 asl #3    // r1 = r0 * 9 = r0 + r0 * 8 = r0 + r0 << 3
add r2, r1, r1 asl #3    // r2 = r0 * 81 = r1 * 9
```

2. Записать в одну команду выражение:

X = (X >= 0) ? X : X - 1

Решение:

```
add r0, r0, r0 asr #31
```

Работа с памятью

Команды загрузки/сохранения LDR / STR:

```
ldr r1, [r2, #+/-imm12]
ldr r1, [r2, +/-r3, shift imm5]
```

Примеры:

```
    ldr r1, [pc, #256]
    ldr r1, [sp, r2, asl #2]
```

```
switch (x) {
  case 1: …
  case 2: …
  …
  case N: …
}
```

```
L1:
    ldr r1, L1 + 248          // по L1 + 248 находится таблица адресов
  ldr pc, [r1, r2, asl #2]    // table-jump для switch
```

Оптимизации ARM/RISC

- Ограничения на записи констант

  Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

  $$CONST\_32 = CONST\_8 << (2 * N),\ \ 0 <= N < 16$$

  - Машинно-независимая оптимизация (удаление общих подвыражений) должна «знать» эту архитектурную особенность

Исходный пример

```
if (x)
  a = b + c << 2;
else
  d = e + c << 2;
```

Было в GCC (неэффективно)

```
  mov r1, r2, asl #2
  cmp r3, #0
  bne L1
  add r4, r5, r1
  b .L2
.L1:
  add r6, r7, r1
```

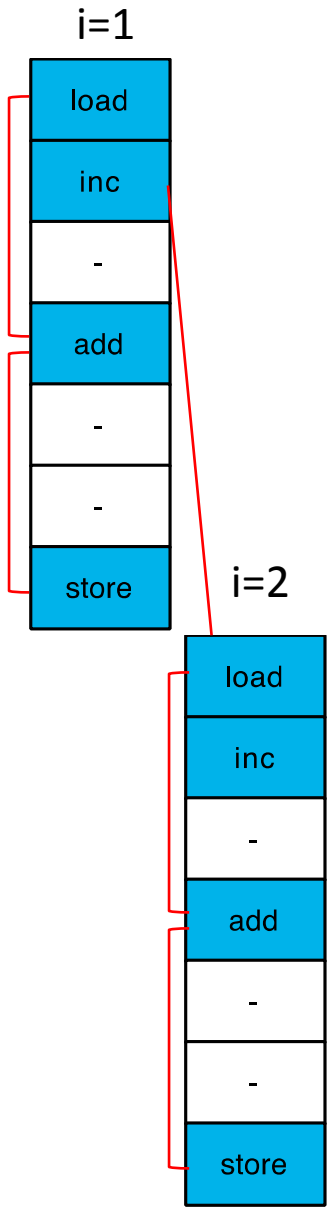
После исправления

```
  cmp r3, #0
  bne L1
  add r4, r5, r2, asl #2
  b .L2
.L1:
  add r6, r7, r2, asl #2
```

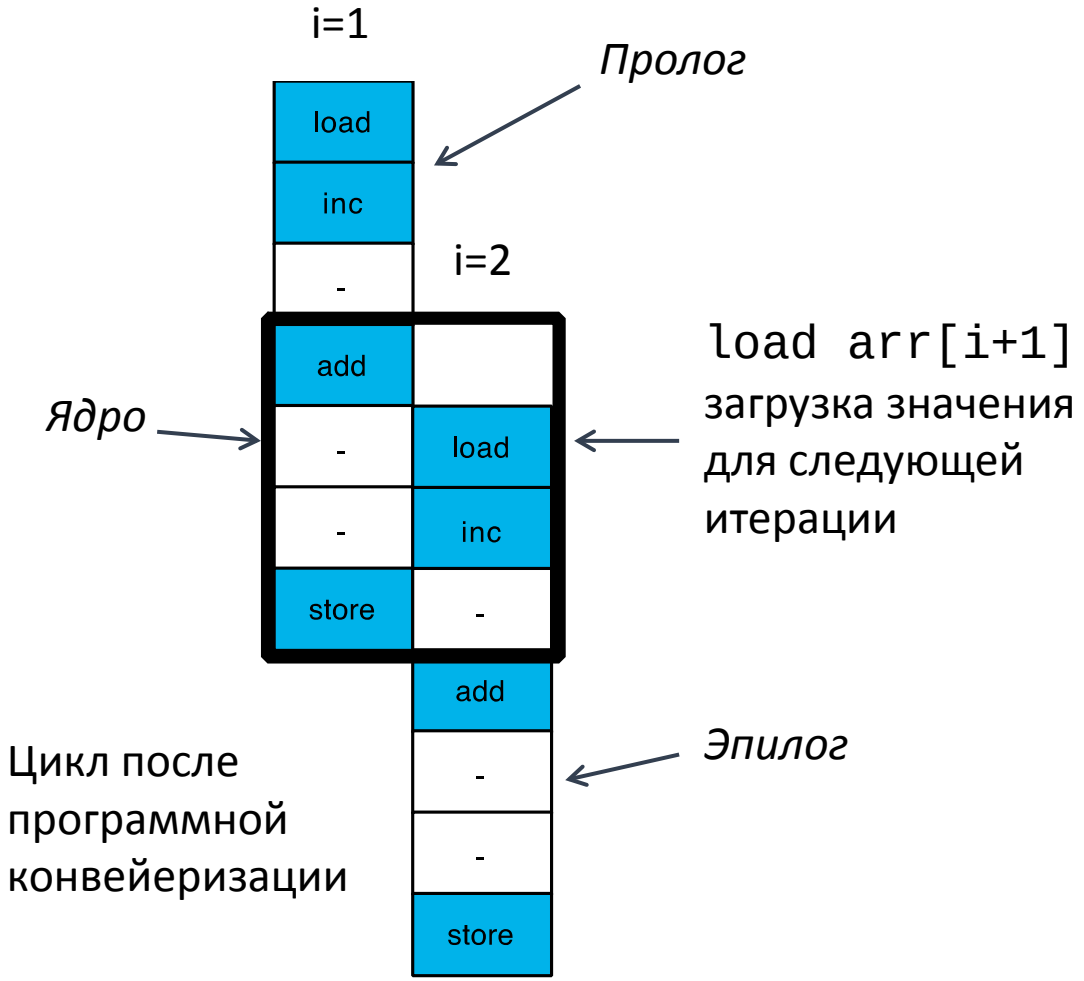
Оптимизации ARM/RISC

- Декомпозиция сложных операций в простые
  - Сложные операции могут быть не реализованы «в железе»
- Планирование команд и конвейеризация циклов
  - Моделирование конвейера
  - Скрытие задержек в загрузке из памяти
- Поддержка новых свойств архитектуры
  - Безопасность (тегированные указатели, TrustZone…)
  - Векторные команды
  - Компиляторы и JIT-компиляторы часто отстают (кроме основных: GCC и LLVM)

Программная конвейеризация



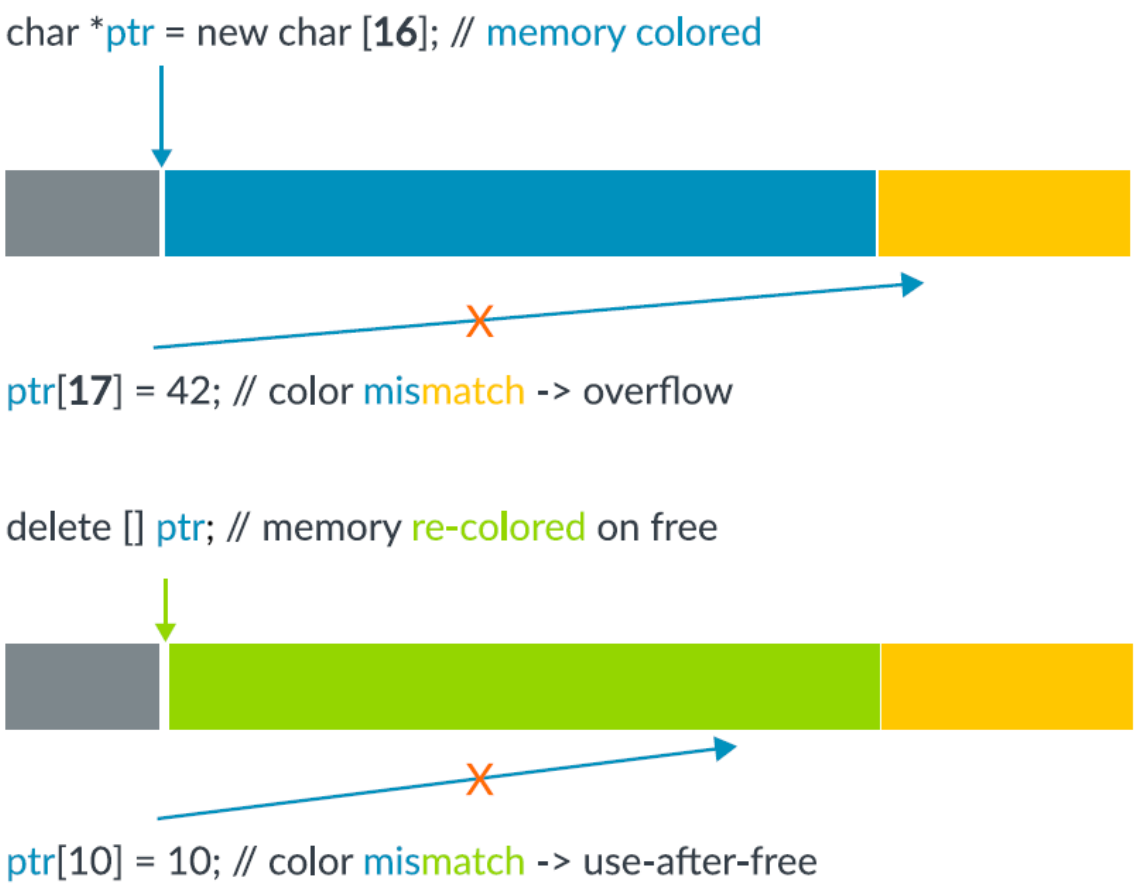
Цикл после планирования команд



`load arr[i+1]` загрузка значения для следующей итерации

Цикл после программной конвейеризации

ARM MTE (Memory Tagging Extension)



- Lock/Key доступ к регионам памяти
  - Тегирование указателей (4 бита)
  - При несовпадении тегов можно выбросить синхронное или асинхронное исключение (определить ошибочную инструкцию или только поток выполнения)
  - Меняется ядро и библиотека C, компилятор выдает загрузки/записи как и раньше

mmap(..., PROT_READ | ... | **PROT_MTE**)

Векторное выполнение на ARM

- Scalable Vector Extension (2016)
  - Переменный размер векторов от 128 до 2048 бит
  - Обычные арифметические операции (целые и вещественные), редукция
  - Тригонометрические функции и др.
  - Gather/Scatter, условное выполнение
  - Спекулятивная векторизация
  - SVE реализован в процессоре A64FX суперкомпьютера Fugaku
- SVE 2 (2019)
  - Больше типов инструкции, в т.ч. для сложной обработки данных (DSP)
  - Криптография (опционально)

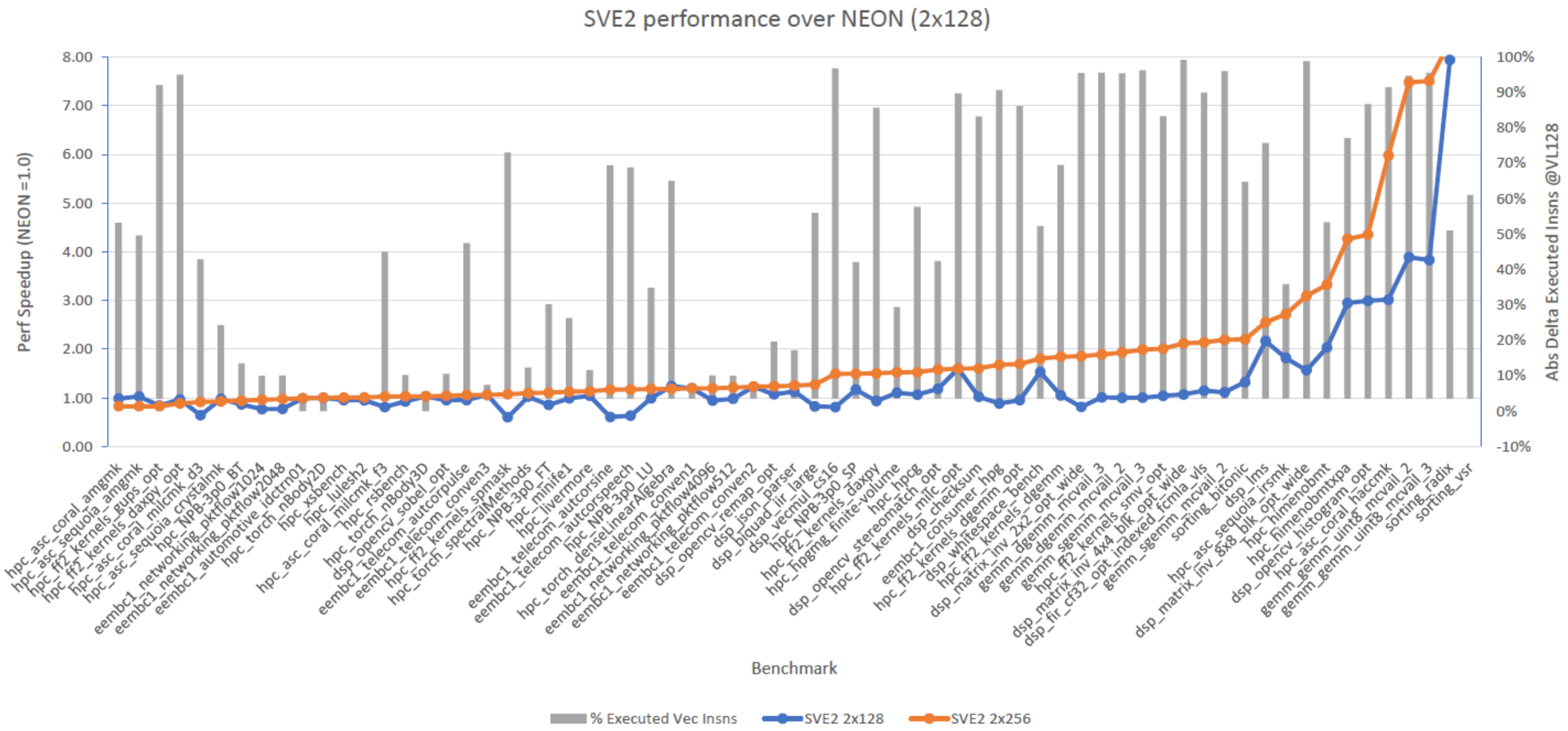
Scalar SAXPY

```
// ----------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ----------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                  // x4=i=0
    b       .latch

.loop:
    ldr     s1, [x0,x4,lsl 2]       // s1=x[i]
    ldr     s2, [x1,x4,lsl 2]       // s2=y[i]
    fmadd   s2, s1, s0, s2          // s2+=x[i]*a
    str     s2, [x1,x4,lsl 2]       // y[i]=s2
    add     x4, x4, #1              // i+=1
.latch:
    cmp     x4, x2                  // i < n
    b.lt    .loop                   // more to do?
    ret
```

SVE SAXPY

```
// ----------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ----------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                      // x4=i=0
    whilelt p0.s, xzr, x2               // p0=while(i++<n)
    dup     z0.s, s0                    // z0=dup(A)
.loop:
    ld1w    z1.s, p0/z, [x0,x4,lsl 2]   // p0:z1=x[i]
    ld1w    z2.s, p0/z, [x1,x4,lsl 2]   // p0:z2=y[i]
    fmla    z2.s, p0/m, z1.s, z0.s      // p0?z2+=x[i]*a
    st1w    z2.s, p0, [x1,x4,lsl 2]     // p0?y[i]=z2
    sqincw  x4                          // i+=(VL/32)

    whilelt p0.s, x4, x2                // p0=while(i++<n)
    b.first .loop                       // more to do?
    ret
```

Слайд из презентации ARM



Слайд из презентации ARM

Сложности работы с векторами

- Добавление векторных расширений требует обновления ОС
  - Хочется иметь call-saved векторные регистры → нужно планировать расширение набора команд заранее
- Динамический загрузчик также должен работать с векторными регистрами
  - В SVE предлагается запретить ленивое связывание для функций, работающих с векторами
- Варианты использования: ассемблер, интринсики, автовекторизация, языки типа OpenCL
  - SVE – ставка на автовекторизацию
  - Маски (ограничение пролога/эпилога), условное выполнение, gather/scatter позволяют векторизовать больше циклов

Транзакционная память на ARM

- Transactional Memory Extension (TME, 2019)
- Инструкции tstart/tcommit
- Гарантии аппаратуры
  - Изоляция
  - Атомарность
  - ПО должно обеспечить путь выполнения без транзакций
- Можно запускать транзакционно и параллельно критические секции (меняется lock/unlock на tstart/tcommit)

Ассоциация Linaro (linaro.org)

- Занимается оптимизацией свободного программного обеспечения для развития экосистемы ARM
- Один из пяти крупнейших контрибьюторов ядра Linux; работает над десятками проектов на основе СПО
- Создана в 2010 году и в 2016 уже насчитывала 36 компаний-участников
- В продвижение платформы ARM вкладываются такие крупные компании, как IBM, Google, Samsung и др.
- Команда насчитывает 300 разработчиков
- Дважды в год проводится конференция Linaro Connect, которая собирает более 400 участников

Архитектура RISC-V (riscv.org)

- Зародилась в университете Беркли как исследовательский проект (май 2010 г., Асанович, Паттерсон)
  - Существующие RISC-процессоры были или сложны, или дороги для использования в учебе
- Основная идея: спецификация набора команд + расширения, **открытая лицензия**
- RISC-V не является конкретным процессором: можно реализовывать спецификацию, добавлять расширения
- RISC-V не является коммерческой компанией (разработка управляется некоммерческой организацией RISC-V Foundation наподобие Linux Foundation)
  - Образована в 2015 году, сейчас более 600 компаний-участников
  - Ядра и платы: SiFive, Syntacore, Andes…

Архитектура RISC-V

- RV[###][abc.....xyz]
  - [###] - {32, 64, 128} → битность регистров и адресного пространства
  - [abc...xyz] → набор расширений архитектуры
  - Единственное обязательное расширение: I (Integer)
- Примеры: RV32I, RV32GC, RV64GCXext

Extension	Description
I	Integer
M	Integer Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
G	General Purpose = IMAFD
C	16-bit Compressed Instructions
Non-Standard User-Level Extensions	
Xext	Non-standard extension “ext”

Common RISC-V Standard Extensions
*Not a complete list

Регистры RISC-V

- 32 целочисленных (опционально еще 32 вещественных)
- Ширина определяется архитектурой
- Бинарные соглашения (ABI) определяют функции некоторых регистров

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function Arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Режимы архитектуры, память

- Три уровня привилегий (режимы): пользовательский, супервизор, машинный (привилегированный – единственный требуемый)
  - Пользовательский режим: версия 2.2
  - Привилегированный: версия 1.11
- Защита физической памяти (защита области памяти, требует привилегий для любого доступа)
  - До 16 областей размером от 4 байт
- Виртуальная память (требует реализации режима супервизора)
  - Можно реализовывать управление памятью в т.ч. для Linux

Инструкции RISC-V

- 32-битные, константы расширяются знаково
- Аргументы в одних местах

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

Инструкции RISC-V

ISP RAS

RISC-V ① ② ③ **RISC-V Reference Card** ④

Base Integer Instructions (32|64|128)

Category	Name	Fmt	RV{32|64|128}I Base	
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Word	I	L{W|D|Q}	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	L{H|W|D}U	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	S{W|D|Q}	rs1,rs2,imm
Shifts	Shift Left	R	SLL{|W|D}	rd,rs1,rs2
	Shift Left Immediate	I	SLLI{|W|D}	rd,rs1,shamt
	Shift Right	R	SRL{|W|D}	rd,rs1,rs2
	Shift Right Immediate	I	SRLI{|W|D}	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA{|W|D}	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI{|W|D}	rd,rs1,shamt
Arithmetic	ADD	R	ADD{|W|D}	rd,rs1,rs2
	ADD Immediate	I	ADDI{|W|D}	rd,rs1,imm
	SUBtract	R	SUB{|W|D}	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm
Jump & Link	J&L	UJ	JAL	rd,imm
	Jump & Link Register	I	JALR	rd,rs1,imm
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE	rd
	ReaD CYCLE upper Half	I	RDCYCLEH	rd
	ReaD TIME	I	RDTIME	rd
	ReaD TIME upper Half	I	RDTIMEH	rd
	ReaD INSTR RETired	I	RDINSTRET	rd
	ReaD INSTR upper Half	I	RDINSTRETH	rd

RV Privileged Instructions (32|64|128)

Category	Name	Fmt	RV mnemonic	
CSR Access	Atomic R/W	R	CSRRW	rd,csr,rs1
	Atomic Read & Set Bit	R	CSRRS	rd,csr,rs1
	Atomic Read & Clear Bit	R	CSRRC	rd,csr,rs1
	Atomic R/W Imm	R	CSRRWI	rd,csr,imm
	Atomic Read & Set Bit Imm	R	CSRRSI	rd,csr,imm
	Atomic Read & Clear Bit Imm	R	CSRRCI	rd,csr,imm
Change Level	Env. Call	R	ECALL	
	Environment Breakpoint	R	EBREAK	
	Environment Return	R	ERET	
Trap Redirect	to Superviso	R	MRTS	
	Redirect Trap to Hypervisor	R	MRTH	
	Hypervisor Trap to Supervisor	R	HRTS	
Interrupt	Wait for Interrup	R	WFI	
MMU	Supervisor FENCE	R	SFENCE.VM	rs1

Optional Multiply-Divide Extension: RV32M

Category	Name	Fmt	RV32M (Mult-Div)	
Multiply	MULtiply	R	MUL{|W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2
Divide	DIVide	R	DIV{|W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2
Remainder	REMainder	R	REM{|W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU{|W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV{32|64|128}A (Atomic)	
Load	Load Reserved	R	LR.{W|D|Q}	rd,rs1
Store	Store Conditiona	R	SC.{W|D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.{W|D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.{W|D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.{W|D|Q}	rd,rs1,rs2
	AND	R	AMOAND.{W|D|Q}	rd,rs1,rs2
	OR	R	AMOOR.{W|D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.{W|D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.{W|D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.{W|D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.{W|D|Q}	rd,rs1,rs2

3 Optional FP Extensions: RV32{F|D|Q}

Category	Name	Fmt	RV{F|D|Q} (HP/SP,DP,QP)	
Load	Load	I	FL{W,D,Q}	rd,rs1,imm
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2
Categorize	Classify Typ	R	FCLASS.{S|D|Q}	rd,rs1
Move	Move from Integer	R	FMV.S.X	rd,rs1
	Move to Integer	R	FMV.X.S	rd,rs1
Convert	Convert from Int	R	FCVT.{S|D|Q}.W	rd,rs1
	Convert from Int Unsigned	R	FCVT.{S|D|Q}.WU	rd,rs1
	Convert to Int	R	FCVT.W.{S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{S|D|Q}	rd,rs1
Configuration	Read Stat	R	FRCSR	rd
	Read Rounding Mode	R	FRRM	rd
	Read Flags	R	FRFLAGS	rd
	Swap Status Reg	R	FSCSR	rd,rs1
	Swap Rounding Mode	R	FSRM	rd,rs1
	Swap Flags	R	FSFLAGS	rd,rs1
	Swap Rounding Mode Imm	I	FSRMI	rd,imm
	Swap Flags Imm	I	FSFLAGSI	rd,imm

3 Optional FP Extensions: RV{64|128}{F|D|Q}

Category	Name	Fmt	RV{F|D|Q} (HP/SP,DP,QP)	
Move	Move from Integer	R	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.{L|T}.{S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.{L|T}U.{S|D|Q}	rd,rs1

Optional Compressed Instructions: RVC

Category	Name	Fmt	RVC	
Loads	Load Word	CL	C.LW	rd',rs1',imm
	Load Word SP	CI	C.LWSP	rd,imm
	Load Double	CL	C.LD	rd',rs1',imm
	Load Double SP	CI	C.LWSP	rd,imm
	Load Quad	CL	C.LQ	rd',rs1',imm
	Load Quad SP	CI	C.LQSP	rd,imm
	Load Byte Unsigned	CL	C.LBU	rd',rs1',imm
	Float Load Word	CL	C.FLW	rd',rs1',imm
	Float Load Double	CL	C.FLD	rd',rs1',imm
	Float Load Word SP	CI	C.FLWSP	rd,imm
	Float Load Double SP	CI	C.FLDSP	rd,imm
Stores	Store Word	CS	C.SW	rs1',rs2',imm
	Store Word SP	CSS	C.SWSP	rs2,imm
	Store Double	CS	C.SD	rs1',rs2',imm
	Store Double SP	CSS	C.SDSP	rs2,imm
	Store Quad	CS	C.SQ	rs1',rs2',imm
	Store Quad SP	CSS	C.SQSP	rs2,imm
	Float Store Word	CSS	C.FSW	rd',rs1',imm
	Float Store Double	CSS	C.FSD	rd',rs1',imm
	Float Store Word SP	CSS	C.FSWSP	rd,imm
	Float Store Double SP	CSS	C.FSDSP	rd,imm
Arithmetic	ADD	CR	C.ADD	rd,rs1
	ADD Word	CR	C.ADDW	rd',rs2'
	ADD Immediate	CI	C.ADDI	rd,imm
	ADD Word Imm	CI	C.ADDIW	rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm
	Load Immediate	CI	C.LI	rd,imm
	Load Upper Imm	CI	C.LUI	rd,imm
	MoVe	CR	C.MV	rd,rs1
	SUB	CR	C.SUB	rd',rs2'
	SUB Word	CR	C.SUBW	rd',rs2'
Logical	XOR	CS	C.XOR	rd',rs2'
	OR	CS	C.OR	rd',rs2'
	AND	CS	C.AND	rd',rs2'
	AND Immediate	CB	C.ANDI	rd',rs2'
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm
	Shift Right Immediate	CB	C.SRLI	rd',imm
	Shift Right Arith Imm	CB	C.SRAI	rd',imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm
	Branch≠0	CB	C.BNEZ	rs1',imm
Jump	Jump	CJ	C.J	imm
	Jump Register	CR	C.JR	rd,rs1
Jump & Link	J&L	CJ	C.JAL	imm
	Jump & Link Register	CR	C.JALR	rs1
System	Env. BREAK	CI	C.EBREAK	

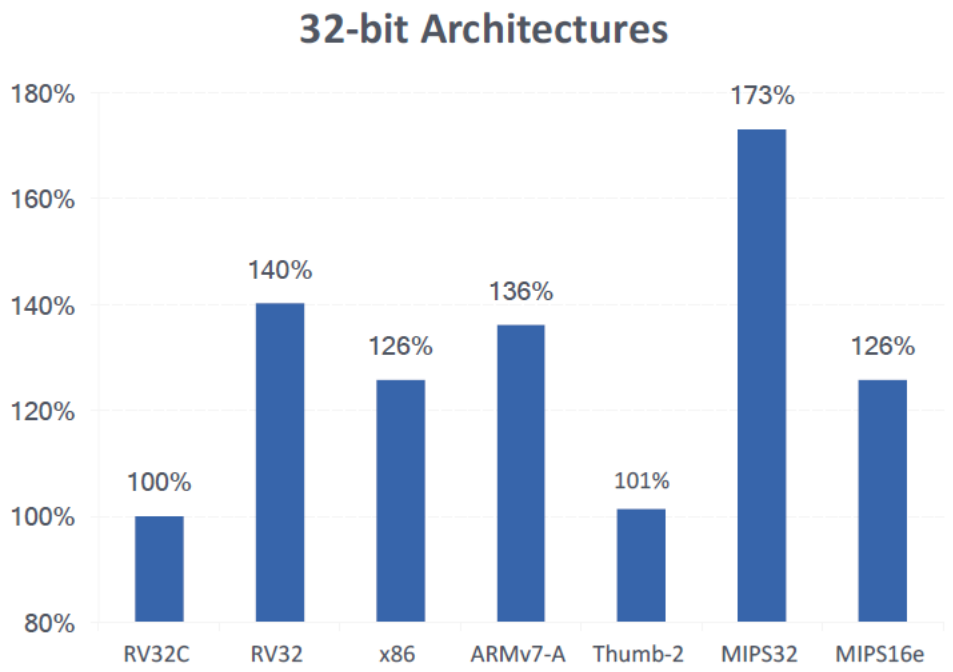
16-bit (RVC) and 32-bit Instruction Formats

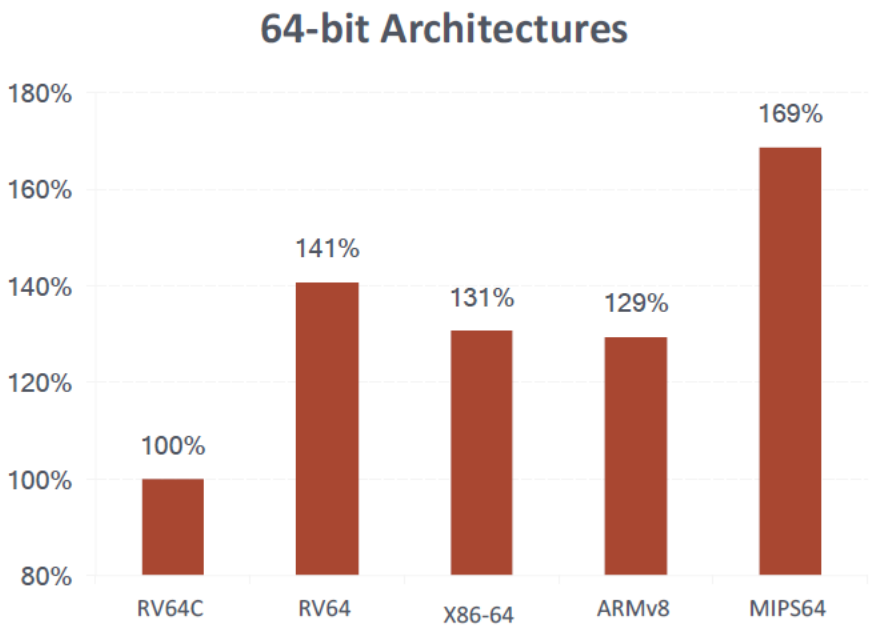
	15 14 13	12	11 10 9	8 7 6	5 4 3	2 1 0
CI	funct4		rd/rs1		rs2	op
CSS	funct3	imm	rd/rs1		imm	op
CIW	funct3	imm			rs2	op
CL	funct3	imm			rd'	op
CS	funct3	imm	rs1'	imm	rd'	op
CB	funct3	imm	rs1'	imm	rs2'	op
CJ	funct3	offset	rs1'	offset		op
	funct3	jump target				op

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

«Компактные» команды (C-extension)

- Большинство I-команд можно записать в 16 бит
  - Схожий режим поддерживается и 64-битной архитектурой
  - «Маленькие» команды позволяют делать чипы меньшего размера и могут помочь улучшить производительность
- Размер кода SPECint2006 на различных архитектурах





Другие расширения

- A-extension: атомарные операции с памятью (read-modify-write)
- Упорядочивание доступа к памяти/IO: инструкция FENCE
- Инструкции доступа к CSR-регистрам
  - ID ядра (hart), версия архитектуры, производитель
  - mstatus – чтение и управление состоянием
  - mtime/mtimecmp – таймеры
- Прерывания: внешние, таймер, программные
  - mcause – определение причины прерывания
- Глобальные прерывания: доставляются любому ядру

Итак…

- RISC-процессоры повсюду: от датчиков до суперкомпьютеров
- Модель развития: «ядро» архитектуры + расширения
  - Безопасность и производительность
- Ключевой фактор успеха: открытость архитектуры
  - RISC-V 10 лет: взрывной рост сообщества
  - Управляется некоммерческой организацией
  - Открытое «железо» следует за открытым ПО: уменьшение стоимости разработки, расширяемость, безопасность
  - Использование вендорами: Microsemi, Nvidia, Western Digital
  - Использование государствами: Индия, Америка, Китай, Израиль, ЕС
- Нужно системное ПО: компиляторы, ОС, библиотеки

Спасибо!

- Материалы презентаций: ARM, SiFive, Embecosm
- Отдел компиляторных технологий ИСП РАН: Дмитрий Мельник, Александр Монаков