

Примеры архитектур: Intel x86 и RISC-V

Луцив Дмитрий Вадимович

Кафедра системного программирования СПбГУ



- 1 Архитектура и система команд x86
 - Регистры и адресация
 - Система команд
- 2 Архитектура и система команд RISC-V
 - Расширения и профили
 - Регистровый файл
 - Соглашение о вызовах
 - Типы команд и формат машинного кода
 - Конвейер
 - Кросс-компиляция

Архитектура и система команд x86

- Регистры и адресация
- Система команд

Регистры (для 32-битной ЭВМ)

- EAX (общий, аккумулятор), EDX (умножение и деление вместе с EAX), EBX (указатели), ECX (счетчик)
- EDI (dest index), ESI (source index)
- EBP, ESP, EIP
- CS, SS, DS, ES, FS, GS — сегментные
- EFLAGS

Регистры (для 32-битной ЭВМ)

- EAX (общий, аккумулятор), EDX (умножение и деление вместе с EAX), EBX (указатели), ECX (счетчик)
- EDI (dest index), ESI (source index)
- EBP, ESP, EIP
- CS, SS, DS, ES, FS, GS — сегментные
- EFLAGS

Фрагменты регистров

- _H, _L — 8-разрядные
- _X, _S — 16-разрядные
- E_ — 32-разрядные
- R_ — 64-разрядные

Например, для аккумулятора

(AH (8), AL (8)) → AX (16) → EAX (32) → RAX (64)

Регистр флагов EFLAGS. Весь регистр 32-битный (начиная с 80386).

Основные флаги (с 8086):

- OF — флаг переполнения
- DF — флаг направления
- IF — флаг прерывания
- TF — флаг трассировки
- SF — флаг знака
- ZF — флаг нуля
- AF — флаг дополнительного переноса (для упакованных двоично-десятичных операций)
- PF — флаг четности
- CF — флаг переноса

- Непосредственная (аргументы в коде)
- Регистровая (номер регистра в коде)
- Память[E_X + смещение], Память[EBP + смещение], + возможно префиксы сегментов

- MOV память обменивается только с арифметическими регистрами, ESI, EDI
- XCHG reg, mem/reg
- LAHF, SAHF — флаги \leftrightarrow AH

Логические

AND, OR, XOR, NOT

Арифметические

- ADD, SUB, ADC, SBB, INC, DEC, NEG
- MUL (reg/mem), DIV (reg/mem), IMUL, IDIV,
- CWD (EAX → EDX:EAX)

Сдвига

- ROR, ROL
- RCL, RCR — с переносом
- SHL, SHR — без переноса
- SAL, SAR — со знаковыми битами

ASCII и BCD — для быстрого преобразования двоично-десятичных чисел

- PUSH, POP
- PUSH, POPA
- Косвенно — CALL, RET, INT, IRET

Переходы Безусловные

- JMP FAR, NEAR, JMP M[xx], JMP REG

Команды Сравнения

- CMP — как SUB
- TEST — как AND
- CMPS — CMPSB, CMPSW, CMPSD

Переходы Безусловные

- JMP FAR, NEAR, JMP M[xx], JMP REG

Команды Сравнения

- CMP — как SUB
- TEST — как AND
- CMPS — CMPSB, CMPSW, CMPSD

Compare-exchange

- CMPXCHG dest, src — Сравнивает аккумулятор (8-64 bits) с dest. Если равны, то в dest грузят src, иначе в аккумулятор загружают dest

Команды сравнения и передачи управления

Переходы Безусловные

- JMP FAR, NEAR, JMP M[xx], JMP REG

Команды Сравнения

- CMP — как SUB
- TEST — как AND
- CMPS — CMPSB, CMPSW, CMPSD

Compare-exchange

- CMPXCHG dest, src — Сравнивает аккумулятор (8-64 bits) с dest. Если равны, то в dest грузят src, иначе в аккумулятор загружают dest

Ужас. Кошмар. Для чего она?.. [↗](#)

По результату R или итогам сравнения $A ? B$, в зависимости от получившихся значений флагов.

Беззнаковые

- JA/JNBE — если $A > B$;
- JAE/JNB/JNC — если $A \geq B$;
- JB/JNAE/JC — если $A < B$;
- JBE/JNA — если $A \leq B$.

Знаковые

- JG/JNLE — если $A > B$;
- JGE/JNL — если $A \geq B$;
- JL/JNGE — если $A < B$;
- JLE/JNG — если $A \leq B$;
- JNS — если $R \geq 0$;
- JS — если $R < 0$.

По результату R или итогам сравнения $A ? B$, в зависимости от получившихся значений флагов.

- JE/JZ – если $A = B \vee R = 0$;
- JNE/JNZ – если $A \neq B \vee R \neq 0$;
- JNO – $\neg OF$;
- JO – OF ;
- JCXZ – $CX = 0$ – для организации циклов
do ... while(--CX);;
- JNP/JPO – $\neg PF$;
- JP/JPE – PF .

- Вызовы
 - CALL адрес
- Прерывания
 - Управление STI, CLI
 - Ожидание (HALT)

IN (mem/DX), OUT (mem/DX) — с AL

- REP, REPE, REPZ, REPNE, REPNZ
- LODS (загружает в аккумулятор),
STOS (пишет из аккумулятора),
MOVS (B-W-D — пересылка память-память),
CMPS(сравнение память-память),
SCAS (вычитает из аккумулятора)

- REP, REPE, REPZ, REPNE, REPNZ
- LODS (загружает в регистр),
STOS (пишет из регистра),
MOVS (B-W-D — пересылка регистр-регистр),
CMPS(сравнение регистр-регистр),
SCAS (вычитает из регистра)

Команды учитывают DF — флаг направления. Выставив его «неправильно» можно быстро размножить участок памяти

Стековые

- FLD, FSTP — загрузить из памяти / выгрузить в память, формат
- FILD, FLD, ... — загрузить из регистра целое / выгрузить в регистр целое
- FMUL, ... — операции и функции
- FWAIT — синхронизация для старых процессоров

Регистровые

MULSD, MULSF — работают с векторными регистрами (появились в Pentium MMX, позволяют производить по несколько операций с числами разной длины)

Загрузка таблиц страниц и дескрипторов, изменение режимов работы процессора

Архитектура и система команд RISC-V

- Расширения и профили
- Регистровый файл
- Соглашение о вызовах
- Типы команд и формат машинного кода
- Конвейер
- Кросс-компиляция

Расширения и профили

Базовые наборы и расширения

- Есть минимальные базовые наборы инструкций, регистров, прочих свойств процессора
- Есть стандартные расширения, пополняющие базовые наборы

Википедия [↗](#)

Профили и реализации

- Наборы расширений образуют профили, профили объединяются в семейства профилей
- Реализации могут включать разные расширения и реализовывать разные профили

Например, [профили для микроконтроллеров](#) [↗](#) или [профиль для запуска полновесных ОС](#) [↗](#)

Да давайте сразу [в английскую Википедию](#)  !

Обращаем внимание:

- Регистров много (характерно для RISC), поскольку для работы с ОЗУ команды отдельные
- У них есть разные названия (просто номер и смысловое) — для более дружественного кода на языке ассемблера
- Есть *интересные* регистры, точнее их предназначения:
 - `x0 zero` — тождественный ноль, наподобие `/dev/zero`
 - `x1 ra` — адрес возврата в регистре!
 - `x10-x17 a0-a7` — специальные регистры для аргументов и возвращаемых функциями значений
 - Оберегаемые/сохраняемые (`x18-27 s2-11`, вызываемая функция должна их восстанавливать) и не оберегаемые/временные (`x28-31 t3-6`, вызываемая функция не должна их восстанавливать) регистры

- Листовые функции — которые сами никого не вызывают, адрес возврата в вызывающую функцию при вызове сохраняется в регистре `ra`
- Не листовые функции — которые вызывают другие функции, и перед вызовом должны сохранять значение `ra` (на стеке, во временном регистре — это их дело), а потом восстановить его

Это позволяет быстро и часто вызывать «мелкие» функции. Подробнее:

- В английской Википедии => [↗](#)
- *Сара Л. Харрис, Дэвид Харрис. Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021. – 810 с.: ил.*

Формат машинного кода и примеры команд (1, 2, 3)

32-bit RISC-V instruction formats																																			
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										
<ul style="list-style-type: none">• <i>opcode</i> (7 bits): Partially specifies which of the 6 types of <i>Instruction formats</i>.• <i>funct7</i>, and <i>funct3</i> (10 bits): These two fields, further than the <i>opcode</i> field, specify the operation to be performed.• <i>rs1</i>, <i>rs2</i>, or <i>rd</i> (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.																																			

- **Reg/reg** операции над несколькими регистрами
 $s0 = s1 + s2; \rightarrow \text{add } s0, s1, s2;$ или
 $s0 = s1; \rightarrow \text{add } s0, s1, \text{zero}$
- **Immediate** операция над регистром и константой
 $s0 = s1 - 12; \rightarrow \text{addi } s0, s1, -12$
- **Upper immediate** загрузка константы в старшие биты регистра
 $s2 = 0xABCDE123; \rightarrow$
 $\rightarrow \text{lui } s2, 0xABCDE$
 $\text{addi } s2, s2, 0x123$

Формат машинного кода и примеры команд (4, 5)

32-bit RISC-V instruction formats																																							
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												
<ul style="list-style-type: none">• <i>opcode</i> (7 bits): Partially specifies which of the 6 types of <i>Instruction formats</i>.• <i>funct7</i>, and <i>funct3</i> (10 bits): These two fields, further than the <i>opcode</i> field, specify the operation to be performed.• <i>rs1</i>, <i>rs2</i>, or <i>rd</i> (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.																																							

- [Load]/Store операция с короткими индексами массивов
`char a[10]; s2 = a[5];` →
→ загр. адр. а в t1
`lb t5, 5(t1)`
- Branch ветвление или безусловный переход
`if(s1 >= t2) goto недалеко;` → bge s1, t2, **недалеко**
`goto недалеко;` → beq zero, zero, **недалеко** (можно и проще, ниже)
(недалеко – адрес относительный)

Формат машинного кода и примеры команд (6)

32-bit RISC-V instruction formats																																				
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						

- *opcode* (7 bits): Partially specifies which of the 6 types of *Instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

- **Jump** – вызов, т.е. переход с сохранением адреса возврата

```
void leaf(){}
int main(){leaf(); return 0;}
```

→ jal ra, leaf

Но так сделать и безусловный переход: j **недалеко** ↔ jal zero, **недалеко**

- **Снова Immediate** – возврат из функции

возврат ret на самом деле jalr zero, ra, 0 – перейти по адресу в ra + 0, исходный адрес записать в zero (выкинуть)

- Язык всё ещё не для человека:
 - загрузка длинных констант в два захода
 - «не очевидная» (но удобная для электроники!) мешанина с командами вызова и перехода
- Машинный код действительно простой!
- Благодаря регистру zero из «странных» команд делаются более очевидные «псевдокоманды»:
 - jalr zero, ra, 0 \rightarrow ret
 - jal zero, недалеко \rightarrow j недалеко
 - sub s1, zero, s0 \rightarrow neg s1, s0 \leftrightarrow s1 = -s0
 - addi x0, x0, 0 \rightarrow nop
 - и т.д.

Общее впечатление от системы команд

- Язык всё ещё не для человека:
 - загрузка длинных констант в два захода
 - «не очевидная» (но удобная для электроники!) мешанина с командами вызова и перехода
- Машинный код действительно простой!
- Благодаря регистру zero из «странных» команд делаются более очевидные «псевдокоманды»:
 - jalr zero, ra, 0 → ret
 - jal zero, недалеко → j недалеко
 - sub s1, zero, s0 → neg s1, s0 ↔ s1 = -s0
 - addi x0, x0, 0 → nop
 - и т.д.

За пределами лекции:

- расширение C(ompact) — формат машинного кода, в котором некоторые команды по 2 байта
- RISC-V 32 vs 64 — у RISC-V 64 регистры 64-битные, но машинный код 32-битный

Наличие и характер конвейера зависит от реализации

Наличие и характер конвейера зависит от реализации

Конвейер распознаёт конфликты

Потому что реализации стремятся быть (и являются) достаточно умными, и поддерживать:

- Суперскалярность
- Предсказание переходов
- В пределе — внеочередное исполнение

Т.е. у конвейера нет шансов остаться таким же простым, как и у MIPS

Наличие и характер конвейера зависит от реализации

Конвейер распознаёт конфликты

Потому что реализации стремятся быть (и являются) достаточно умными, и поддерживать:

- Суперскалярность
- Предсказание переходов
- В пределе — внеочередное исполнение

Т.е. у конвейера нет шансов остаться таким же простым, как и у MIPS

Обсуждение на [Stack Overflow](#) ↗

- При помощи **BuildRoot** [↗](#)
- С отладкой — при помощи **Ripes** [↗](#)

Вопросы

- 1 Расскажите о составе регистрового файла x86
- 2 Расскажите о составе регистрового файла RISC-V
- 3 Приведите примеры арифметико-логических команд x86
- 4 Приведите примеры арифметико-логических команд RISC-V
- 5 Что такое микроциклы x86?
- 6 Приведите примеры и опишите работу нескольких команд условного перехода x86
- 7 Приведите примеры и опишите работу нескольких команд условного перехода RISC-V

Упражнения

- 1 Скомпилируйте программу из примера для любой незнакомой архитектуры; пользуйтесь справочниками, объясните действия всех машинных команд

Вопросы



[EDU.DLUCIV.NAME](#) ↗