

Определения целевых архитектур

Нотация, предназначенная для описания моделей памяти и наборов инструкций виртуальных машин, реализуемых в аппаратной форме.

Определение каждой архитектуры задаётся посредством описания поддерживаемых данной нотацией аспектов: конфигурации регистров, адресных пространств, стеков, набора команд и мнемоник к ним.

Общая форма

Определение каждой архитектуры заключается в блок `architecture`, содержащий перечисление присущих ей аспектов. Комментарии задаются аналогично языку Си:

```
architecture x86 {  
    registers:    /* аспекты конфигурации регистров */  
        ...  
    memory:      /* аспекты адресных пространств */  
        ...  
    stacks:      /* аспекты стеков */  
        ...  
    instructions: /* аспекты инструкций */  
        ...  
    mnemonics:   /* аспекты мнемоник */  
        ...  
}
```

Аспекты внутри определения архитектуры могут встречаться в любом порядке произвольное количество раз. Незадействованные аспекты могут быть опущены. Например:

```
architecture TestArch {  
    memory:      /* аспекты адресных пространств */  
        ...  
    registers:   /* аспекты конфигурации регистров */  
        ...  
    instructions: /* аспекты инструкций (фрагмент 1) */  
        ...  
    mnemonics:   /* аспекты мнемоник (фрагмент 1) */  
        ...  
    instructions: /* аспекты инструкций (фрагмент 2) */  
        ...  
    mnemonics:   /* аспекты мнемоник (фрагмент 2) */  
        ...  
}
```

Детали аспектов

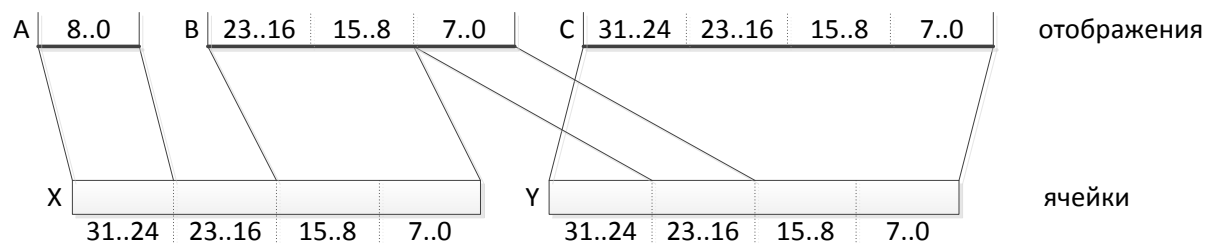
Конфигурация регистров

Описывается соотношение физических ячеек размещения данных (физических регистров) и адресуемых фрагментов этих ячеек (отображений).

Используются следующие конструкции:

```
registers:
    storage <name>[<bits count>];      /* ячейка размером с заданное число бит */
    view <name> = <storage name>;      /* отображение ячейки целиком */
    view <name> = <storage name>[<bin index>]; /* отображение одного бита */
    view <name> = <storage name>[<from>..<to>]; /* отображение фрагмента ячейки */
    view <name> = {<storage1 name>[<from>..<to>], <storage2 name>[<from>..<to>], ... };
                                     /* отображение объединения фрагментов нескольких ячеек */
```

Например:



```
registers:
    storage X[32];                      /* ячейка размером 32 бита */
    storage Y[32];                      /* другая ячейка размером 32 бита */

    view A = X[24..31];                 /* отображение старших восьми бит ячейки X */
    view B = { X[0..15], Y[16..24] }; /* объединение младших 16 бит ячейки X и
                                     младших 8 бит старшей половины ячейки Y */
    view C = Y;                         /* отображение ячейки Y целиком */
```

Адресные пространства

Диапазоны адресуемой памяти описываются с помощью блока следующего вида:

```
memory:
    range <name> [<min address>..<max address>] {
        cell = <memory cell size in bits>;
        endianness = <bytes order spec>;
        granularity = <r/w operations alignment>;
    }
```

Значение параметров:

- cell – размер ячейки в битах (обычно 8)
- endianness – порядок байт слова в отношении старшинства (little-endian или big-endian)
- granularity – требование к выровненности обращений к диапазону (например 2 – по четным адресам)

Пример:

```
memory:
  range ram [0x0000..0xffff] {
    cell = 8;
    endianness = little-endian;
    granularity = 0;
  }
```

Стеки

Такие структуры как стеки чаще всего реализуются двумя механизмами: через организацию стека в памяти с произвольным доступом, поверх регистрового файла. Оба эти случая в очевидных конфигурациях на данный момент могут быть описаны в рамках данной нотации посредством соответствующего адресуемого пространства и необходимых операций в составе инструкций.

Набор инструкций

Описание инструкций включает в себя кодирование соответствующего опкода со входящими в его состав полями, перечисленными через запятую, и функцию, задающую операции, предписываемые к выполнению в рамках модели вычислений. Кодировки всех полей должны быть явно описаны.

```
instructions:
  instruction <name> = { <comma-separated list of fields and fixed bit patterns> } {
    /* последовательность операций */
  };
```

Поля в составе фрагмента опкода могут быть следующих видов:

- непосредственные битовые паттерны, состоящие из символов 0 и 1, опционально разделяемых пробелами
 <bits pattern with spaces>
- аргумент в виде непосредственного значения, интерпретируемого как адрес или данные, аргумент в виде ссылки на регистр из известного набора вариантов, поле из predetermined битовых паттернов
 <field encoding name> as <field name>
- поле, представляющее собой одно значение из predetermined битовых паттернов
 <field encoding name>.<value name>
- predetermined подпоследовательность полей
 sequence <sequence encoding name>
- predetermined подпоследовательность полей с реинтерпретацией аргументов
 <sequence encoding name>(<subfield name> as <field name>, ...)

Кодирование полей в составе опкода

В общем случае, за исключением непосредственных битовых паттернов, кодировки полей инструкций описываются с помощью конструкции

```
encode <encoding name> <encoding kind> = <encoding content>;
```

, где слева направо: имя кодировки, вид содержимого (field – самостоятельное поле, sequence – последовательность полей или группа возможных последовательностей), сама кодировка.

Битовые паттерны

Пример:

```
instructions:
    instruction nop = { 10010000 }; /* непрерывная последовательность бит */
    instruction invd = { 0000 1111, 0000 1000 }; /* биты могут быть сгруппированы */
```

Непосредственные значения, ссылки на регистры и предопределённые паттерны

Описание поля, содержащего непосредственное значение может быть дополнено указанием способа интерпретации данного значения в составе инструкции (data, offset, displacement).

```
instructions:
    encode <name> field = immediate[<bits count>];
    encode <name> field = immediate[<bits count>] <interpretation>;
```

Описание поля, ссылающегося на регистр, содержит список возможных значений с сопоставлением их отображениям конфигурации регистров. Битовые последовательности, задающие возможные значения, должны быть одинаковой длины.

```
instructions:
    encode <name> field = register {
        <reg view 1 name> = <bits>,
        <reg view 2 name> = <bits>,
        ...
    };

```

Описание поля из предопределённых паттернов аналогично ссылке на регистр содержит список поименованных возможных значений, интерпретация которых даётся явно при описании операций конкретной инструкции.

```
instructions:
    encode <name> field = cases {
        <case 1 name> = <bits>,
        <case 2 name> = <bits>,
        ...
    };

```

Пример:

```
instructions:
    encode imm8 field = immediate[8] data;
    encode off16 field = immediate[16] offset;

    encode reg32W0 field = register {
        a1 = 000,
        c1 = 001,
        d1 = 010,
        b1 = 011,
        ah = 100,
        ch = 101,
        dh = 110,
        bh = 111
    };

    instruction adc-imm-to-reg-1 = { 1000 0000, 11 010, reg32W0 as reg, imm8 as value };
```

Последовательности полей

Описание последовательности может включать как одну последовательность полей, входящую в состав опкодов различных инструкций, так и группу взаимно альтернативных наборов полей. Во втором случае, каждый из представленных вариантов последовательности создаёт свою альтернативную форму опкода, в котором задействована данная последовательность, поэтому

длины вариантов не обязаны быть одинаковыми, в отличие от predetermined битовых паттернов.

instructions:

```
    encode <name> sequence = { <comma-separated list of fields and fixed bit patterns> };

    encode <name> sequence = alternatives {
        <case 1 name> = { <comma-separated list of fields and fixed bit patterns> },
        <case 2 name> = { <comma-separated list of fields and fixed bit patterns> },
        ...
    };
```

Пример:

instructions:

```
    encode rmModByte sequence = alternatives {
        rm00reg = { 00, rmModByteReg as reg1, rmMod00AsReg as reg2 },
        rm00sib = { 00, rmModByteReg as reg1, 100                      , sequence sibByte },
        rm00off = { 00, rmModByteReg as reg1, 101                      , off32 as off },
        rm01reg = { 01, rmModByteReg as reg1, rmModXXAsReg as reg2, off8  as off },
        rm01sib = { 01, rmModByteReg as reg1, 100                      , sequence sibByte,
                                                             off8 as off },
        rm10reg = { 10, rmModByteReg as reg1, rmModXXAsReg as reg2, off32 as off },
        rm10sib = { 10, rmModByteReg as reg1, 100                      , sequence sibByte,
                                                             off32 as off },
        rm11reg = { 11, rmModByteReg as reg1, rmMod11AsReg as reg2 }
    };

    instruction adc3w1 = { 0001 0011, sequence rmModByte };
```

Функции инструкций

По форме функции инструкций похожи на функции без возвращаемого значения и со списком именованных полей опкода (as <field name>) в качестве аргументов. Задают предписываемые данной инструкцией к выполнению операции над состоянием VM.

В теле функции инструкции допустимы следующие конструкции:

Вычисление некоторого выражения и запись его результата в регистр, память или переменную	<target> = <expression>;
Объявление переменной с обязательной инициализацией	let <var name> = <expression>;
Блок	{ ... }
Условная конструкция с ветвью для ненулевого выражения	if <expression> then <non-zero case branch>
Условная конструкция с ветвями для обоих случаев	if <expression> then <non-zero case branch> else <zero case branch>
Цикл с пред-условием	while <expression> do <loop body>
Цикл с пост-условием	do <loop body> while <expression>;
Разрыв цикла	break;

Выражения в составе перечисленных конструкций могут включать в себя следующие операторы:

Бинарные		
Приоритет	Группа	Символы
Сильнее	Умножение, деление, остаток	*, /, %
	Сумма и разность	+, -
	Битовые	~, &, ^, <<, >>
	Сравнение	<, >, <=, >=, ==, !=
	Логические	, &&
Слабее	Присваивание	=

Унарные	
Группа	Символы
Инкремент, Декремент	--, ++
Обращение	~, !
Разыменование	*

Кроме перечисленных операторов, в состав выражений могут входить численные литералы трёх видов:

- Двоичные: 0b00011010
- Десятичные: 12345
- Шестнадцатеричные: 0x78ab25FD

Мнемоники

Описания мнемоник ставят в соответствие каждому варианту каждого опкода ту или иную комбинацию мнемоники и форматной строки аргументов. В зависимости от наличия аргументов у инструкции, количества и структуры частей опкода, варианты мнемоник могут быть описаны различными способами. Они могут также быть разбиты на несколько вхождений с различными условиями так, чтобы отдельные форматы не пересекались по соответствующим вариантам инструкций.

Форматные строки аргументов могут быть описаны отдельно и совместно использованы несколькими мнемониками. Если группа форматных строк, соответствующих присутствию некоторых полей в опкоде, встречается в мнемониках различных инструкций, такая группа также может быть описана отдельно вместе с условиями для различения вариантов опкода в похожей части кодировок инструкций.

```
mnemonics:
    format <name> is "<format string>";           /* предопределенная форматная строка */

    format <name> of <comma-separated predefined format spec>; /* группа форматных строк */

    mnemonic <name> <comma-separated format per insn specs>; /* определение мнемоники */
```

Каждый формат из группы форматных строк предваряется списком аргументов и завершается опциональным дополнительным условием, с помощью которого могут быть устранены неоднозначности при сопоставлении нескольких мнемоник и опкодов с похожими списками аргументов. Вместо собственно форматной строки на месте её шаблона может стоять либо имя предопределённой форматной строки, либо имя группы форматных строк. В последнем случае в списке аргументов должно стоять троеточие ('...'), так как списки аргументов для форматных строк группы определены в ней.

```
mnemonics:
    format <name> of (<arg names list 1>) "<format string 1>",      /* ф. строка в группе */
    (...) <format strings group name>,      /* ссылка на группу ф. Строк */
    (<arg names list 2>) "<format string 2>" when <condition>,
    (<arg names list 3>) <format string name> when <condition>,
    ...;
```

Описание самой мнемоники в общем случае отличается от группы форматных строк тем, что перед списком аргументов опкода может стоять указание на имя инструкции, если оно отличается от имени мнемоники.

```
mnemonics:
    mnemonic <name> for <insn name> (<arg names list 1>) "<format string 1>",
    (...) <format strings group name>,
    for <insn name> (<arg names list 2>) "<format string 2>" when <cond>,
    (<arg names list 2>) <format string name> when <condition>,
    ...;
```

Простейшая форма описания мнемоники действует для случая единственного опкода без аргументов:

```
mnemonics:
    mnemonic nop();
```

Если в составе инструкции нет поля, задающего регистр, с которым она работает, но в мнемонике он должен быть указан, то в качестве аргумента могут выступать имена регистров из соответствующей части определения.

```
mnemonics:
    format plain2 is "{1}, {2}";

    mnemonic in for in_imm_to_al (registers.al, port) plain2;      /* in al, <port> */
    mnemonic in for in_imm_to_eax (registers.eax, port) plain2;    /* in eax, <port> */

    mnemonic in for in_dx_to_al (registers.al, registers.dx) plain2, /* in al, dx */
    for in_dx_to_eax (registers.eax, registers.dx) plain2; /* in al, dx */
```

Более сложный пример мнемоники для инструкции `adc3w1`, приведённой ранее:

```
mnemonics:
    format sib-to-plain of (reg1, reg2)      "{1}, [{2}]"          when rm00reg,
    (reg1, r32)      "{1}, [{2} * 1]"        when rm00sib and x1,
    (reg1, r32)      "{1}, [{2} * 2]"        when rm00sib and x2,
    (reg1, r32)      "{1}, [{2} * 4]"        when rm00sib and x4,
    (reg1, r32)      "{1}, [{2} * 8]"        when rm00sib and x8,
    (reg1, off)      "{1}, [{2}]"            when rm00off,
    (reg1, reg2, off) "{1}, [{2} + {3}]"      when rmModXXAsReg,
    (reg1, r32 , off) "{1}, [{2} * 1 + {3}]"  when x1,
    (reg1, r32 , off) "{1}, [{2} * 2 + {3}]"  when x2,
    (reg1, r32 , off) "{1}, [{2} * 4 + {3}]"  when x4,
    (reg1, r32 , off) "{1}, [{2} * 8 + {3}]"  when x8,
    (reg1, reg2)      "{1}, {2}"            when rm11reg;

    mnemonic adc for adc3w1(...) sib-to-plain;
```

Каждый из описанных форматов аргументов мнемоники будет сопоставлен варианту опкода, отвечающему наличию заданных частей в его составе. Например вариант с двумя аргументами `reg1` и `off` будет соответствовать опкоду с вариантом `rm00off` в последовательности `rmModByte`, которая выглядит так: `00, rmModByteReg as reg1, 101, off32 as off`. Результирующий опкод будет следующим: `0001 0011, 00, <encoded register>, 101, <32-bit number>`.