Flag reduction

SPC | 04.15

Why we need flag reduction

```
IN:
0x000400e5f:  test      rdx, rdx

OUT:
0x1608a49a4:  and       a4,at,at
0x1608a49a8:  ori       a5,zero,0x1
0x1608a49ac:  dsll      a5,a5,16
0x1608a49b0:  ori       a5,a5,0x6081
0x1608a49b4:  dsll      a5,a5,16
0x1608a49b8:  ori       a5,a5,0x8f00
0x1608a49bc:  andi      a0,a4,0xff
0x1608a49c0:  daddu     a0,a5,a0
0x1608a49c4:  lbu       a5,0(a0)
0x1608a49c8:  andi      s8,s8,0xfffb
0x1608a49cc:  or        s8,s8,a5      // pf
0x1608a49d0:  ori       a1,s8,0x40
0x1608a49d4:  andi      s8,s8,0xffbf
0x1608a49d8:  movz      s8,a1,a4      // zf
0x1608a49dc:  dsrl32    a2,a4,24
0x1608a49e0:  andi      a2,a2,0x80
0x1608a49e4:  andi      s8,s8,0xff7f
0x1608a49e8:  or        s8,s8,a2      // sf
0x1608a49ec:  andi      s8,s8,0xfffe // cf
0x1608a49f0:  andi      s8,s8,0xf7ff // of
```

从左侧可以看出，flag的翻译会导致大量指令生成。

但其实往往flag中要被用到的标志很少，大量标志位未被使用就被后续的指令覆盖，这些标志位就是无效位，在翻译过程中可以通过省略，起到优化的作用。

How can we make the flag reduction

首先需要记录单条指令的 eflags 信息。

```
typedef struct {
    uint8 use;     // 使用的位
    uint8 def;     // 可能修改的位
    uint8 undef;   // 未定义的位
} IR1_EFLAG_USEDEF;
```

针对每条指令，定义一个 eflag_use 和 eflag_def 域，记录全局信息。

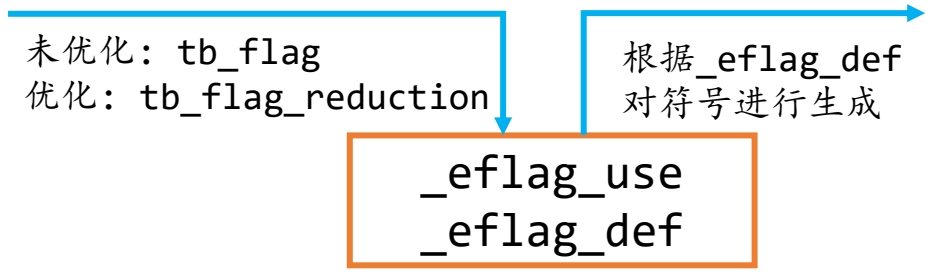
```
typedef struct IR1_INST {
    cs_insn *info;
    uint8_t _eflag_use;      // eflag在未来被使用的
    uint8_t _eflag_def;      // eflag中定义要被翻译
    uint8_t _native_inst_num;
    uint8_t flags;
} IR1_INST;
```

其中 eflag_def 域在无优化情况下定义为X64手册明确的类型，优化后根据后续指令的情况，对需要翻译的指令进行调整。

在无优化的情况下，每条指令记录的都是本身情况，没有做任何的信息删除。

```
tb_flag(void *tb) {
    _eflag_use ← use
    _eflag_def ← def & (~undef)
}
```

优化的情况下，调用 tb_flag_reduction。



tb_flag_reduction

由于翻译是以TB为基本单位，所以在TB最后一条指令的时候需要给出这个TB的正确flag，即生成正确的eflag。

```
0x000415b2e:  sub       eax, 1
0x000415b31:  test      eax, eax
0x000415b33:  mov       dword ptr [rip + 0x2a61db], eax
0x000415b39:  jne       0x415b7c
```

从下边的表格可以知道，sub指令的符号生成可以全部被省略，因为test指令将sub指令的符号全部都覆盖了。

	OF	SF	ZF	AF	PF	CF
sub	M	M	M	M	M	M
test	0	M	M	U	M	0
mov	-	-	-	-	-	-
jne	-	-	- +	-	-	-
->	0	M	M	U	M	0

M: Modify, U:Undefined, -:No change, +:used

一个更极端的例子：

	OF	SF	ZF	AF	PF	CF
sub	M	M	M	M	M	M
adc	M	M	M	M	M	M +
mov	-	-	-	-	-	-
imul	M	U	U	U	U	M
add	M	M	M	M	M	M
rcl	M	-	-	-	-	M +
jne	-	-	- +	-	-	-
->	0	M	M	U	M	0

M: Modify, U:Undefined, -:No change, +:used

算法1.1 - flag_reduction

```
pendind_use ← all_flags;
For i = n..1 insts in TB
DO
  insts[i].def ← pendind_use & insts[i].def;
  pendind_use  ← insts[i].use |
                 (pendind_use & ~insts[i].def);
END
return pendind_use;
```

前序指令需要翻译以使用

没有定义的向前传导

tb_flag_reduction

但在一个TB中寻找flag，其实优化作用并不明显，因为往往一个块中对flag操作的指令并不多，此时可以考虑跨TB块进行。

算法1.1

计算后续TB的flag状况

使用后续TB的flag状况作为初始状态

算法1.2 – tb_flag_reduction

```
Func flag_reduction(init_flag, modify)
  pendind_use ← init_flag;
  For i = n..1 insts in TB
  DO
    current.def ← pendind_use & insts[i].def;
    pendind_use ← insts[i].use |
                  (pendind_use & ~insts[i].def);
    If(modify)
      insts[i].def ← current.def;
    Endif
  END
  return pendind_use;
Endfunc

Func panding_use(tb, i)
  If(i = 0) return all_flags;
  succ_pending_use0 ← panding_use(tb->succ[0], i-1);
  succ_pending_use1 ← panding_use(tb->succ[1], i-1);
  succ_pending_use  ← succ_pending_use0 | succ_pending_use1;
  succ_pending_use  ← flag_reduction(succ_pending_use, FALSE);
  return succ_pending_use
Endfunc

flag_reduction(panding_use(this_tb, MAX_DEPTH), TRUE);
```