Decompiler internals: microcode

(c) Ilfak Guilfanov

Introduction

This presentation is about the Hex-Rays Decompiler. It is a de-facto standard tool used by the security professionals. Its main features include:

- Interactive, fast, robust, and programmable decompiler
- Can handle x86, x64, ARM, ARM64, PowerPC
- Runs on top of the IDA Pro disassembler
- Has been evolving for more than 10 years
- Internals have not been published yet
- Namely, the intermediate language

The intermediate language is called **microcode**. I like the name because it reflects its nature very well: each processor instruction is represented by several microinstructions in the microcode.

The decompiler performs a very straightforward sequence of steps:

1. Generate microcode
2. Transform microcode (optimize, resolve memrefs, analyze calls, etc)
3. Allocate local variables
4. Generate ctree (it is very similar to AST that is used in compilers)
5. Beautify ctree, make it more readable
6. Print ctree

We will focus on the first two steps, generating and transforming microcode.

Before we delve into details, let us justify the use of microcode. Why do we need an intermediate language when building a decompiler? Below are the reasons:

- It helps to get rid of the complexity of processor instructions
- Also we get rid of processor idiosyncrasies. Below are examples, they all require special handling one way or another:
    - x86: segment registers, fpu stack
    - ARM: thumb mode addresses
    - PowerPC: multiple copies of CF register (and other condition registers)
    - MIPS: delay slots
    - Sparc: stack windows
- It makes the decompiler portable. We "just" need to replace the microcode generator. I'm using quotes here because writing a microcode generator is still a complex task.

Overall writing a decompiler without an intermediate language looks like waste of time to me.

Is implementing an intermediate language difficult? Your call :) The devil is in details, as usual. While designing simple arithmetic operations like **add,sub,mul**,etc is very simple, there are many other design

solutions that will affect the simplicity, usefulness, and expressiveness of the language. There are many potential pitfalls, many of them not obvious at the start.

Examples of design choices:

- how should we model the memory? flat model? segments? i/o ports?
- will it be RISC or CISC or something else?
- will there be a stack?
- how many registers? any special registers?

Overall it can be compared to designing a new processor. The simpler we make it, the easier to implement in software. However, it must be powerful enough to express real world code.

One thing is certain: it is a lot of fun, indeed!

There are many other intermediate languages: **LLVM, REIL, Binary Ninja's ILs, RetDec's IL**, etc. However, we use our own language just because I started working on the microcode very long ago (in 1998 or earlier). In 1999 the microcode looked like this:

```
mov.d EAX,, T0
ldc.d #5,, T1
mkcadd.d T0, T1, CF
mkoadd.d T0, T1, CF
add.d T0, T1, TT
setz.d TT,, ZF
sets.d TT,, ZF
mov.d TT,, EAX
```

At that time the intermediate languages listed above did not exist.

Naturally, some design decisions turned out to be bad (and some of them are extremely difficult to fix). For example, the notion of virtual stack registers. As many other researchers, I first implemented various optimization rules that work on the processor registers. Processor registers are never aliasable and we can reason about them without taking into account possible indirect accesses. The same logic applies to parts of the memory that can not be addresses indirectly. The idea was “brilliant”: let us map these memory regions to so called virtual registers. This way we do not need to modify the existing optimization rules, and the accesses to non-aliasable memory will get optimized without any additional effort. While the idea worked for a while, the shortcomings of this approach became more and more visible with time. For example, it was unclear what to do with the objects that are only partially aliasable.

It required a lot of efforts but I'm happy to tell you that the upcoming version will not have virtual registers anymore. While the decompiler engine became more complex, a big artificial concept disappeared, and this is a good thing.

Design goals

The main design goal for the microcode was **simplicity**:

- No processor specific stuff
- One microinstruction does one thing
- Small number of instructions (only 45 in 1999, now 72)
- Simple instruction operands (register, number, memory)
- Consider only compiler generated code

And we discard things we do not care about:

- Instruction timing (anyway it is a lost battle)
- Instruction order (exceptions are a problem!)
- Order of memory accesses (later we added logic to preserve indirect memory accesses)
- Handcrafted code

Initially the microcode looks like RISC code:

- Memory loads and stores are done using dedicated microinstructions
- The desired operation is performed on registers
- Microinstructions have no side effects
- Each output register is initialized by a separate microinstruction

It is very verbose. Let us take this code for the x86 processor:

```
004014FB mov eax, [ebx+4]
004014FE mov dl, [eax+1]
00401501 sub dl, 61h ; 'a'
00401504 jz  short loc_401517
```

This is the initial microcode, it looks like RISC code:

```
2. 0 mov    ebx.4, eoff.4           ; 4014FB u=ebx.4      d=eoff.4
2. 1 mov    ds.2, seg.2             ; 4014FB u=ds.2       d=seg.2
2. 2 add    eoff.4, #4.4, eoff.4    ; 4014FB u=eoff.4     d=eoff.4
2. 3 ldx    seg.2, eoff.4, et1.4    ; 4014FB u=eoff.4,seg.2,(STACK,GLBMEM) d=et1.4
2. 4 mov    et1.4, eax.4            ; 4014FB u=et1.4      d=eax.4
2. 5 mov    eax.4, eoff.4           ; 4014FE u=eax.4      d=eoff.4
2. 6 mov    ds.2, seg.2             ; 4014FE u=ds.2       d=seg.2
2. 7 add    eoff.4, #1.4, eoff.4    ; 4014FE u=eoff.4     d=eoff.4
2. 8 ldx    seg.2, eoff.4, t1.1     ; 4014FE u=eoff.4,seg.2,(STACK,GLBMEM) d=t1.1
2. 9 mov    t1.1, dl.1              ; 4014FE u=t1.1       d=dl.1
2.10 mov    #0x61.1, t1.1           ; 401501 u=           d=t1.1
2.11 setb   dl.1, t1.1, cf.1        ; 401501 u=dl.1,t1.1  d=cf.1
2.12 seto   dl.1, t1.1, of.1        ; 401501 u=dl.1,t1.1  d=of.1
2.13 sub    dl.1, t1.1, dl.1        ; 401501 u=dl.1,t1.1  d=dl.1
2.14 setz   dl.1, #0.1, zf.1        ; 401501 u=dl.1       d=zf.1
2.15 setp   dl.1, #0.1, pf.1        ; 401501 u=dl.1       d=pf.1
2.16 sets   dl.1, sf.1              ; 401501 u=dl.1       d=sf.1
2.17 mov    cs.2, seg.2             ; 401504 u=cs.2       d=seg.2
2.18 mov    #0x401517.4, eoff.4     ; 401504 u=           d=eoff.4
2.19 jcnd   zf.1, $loc_401517       ; 401504 u=zf.1
```

The 4 processor instructions got translated into 20 microinstructions. Each microinstruction does just one thing. This approach simplifies analyzing and optimizing microcode. However, microcode can represent more complex expressions. Let us see how it looks after the pre-optimization pass:

```
2. 0 ldx    ds.2, (ebx.4+#4.4), eax.4 ; 4014FB u=ebx.4,ds.2,
                                      ;(STACK,GLBMEM) d=eax.4
2. 1 ldx    ds.2, (eax.4+#1.4), dl.1 ; 4014FE u=eax.4,ds.2,
                                      ;(STACK,GLBMEM) d=dl.1
2. 2 setb   dl.1, #0x61.1, cf.1     ; 401501 u=dl.1       d=cf.1
2. 3 seto   dl.1, #0x61.1, of.1     ; 401501 u=dl.1       d=of.1
2. 4 sub    dl.1, #0x61.1, dl.1     ; 401501 u=dl.1       d=dl.1
2. 5 setz   dl.1, #0.1, zf.1        ; 401501 u=dl.1       d=zf.1
2. 6 setp   dl.1, #0.1, pf.1        ; 401501 u=dl.1       d=pf.1
2. 7 sets   dl.1, sf.1              ; 401501 u=dl.1       d=sf.1
2. 8 jcnd   zf.1, $loc_401517       ; 401504 u=zf.1
```

As we see, only 9 microinstructions remain; some intermediate registers disappeared. Sub-instructions (like **eax.4+#1.4**) appeared. Overall the code is still too noisy and verbose.

After further microcode transformations we have:

```
2. 1 ldx    ds.2{3}, ([ds.2{3}:(ebx.4+#4.4)].4+#1.4), dl.1{5} ; 4014FE
                        ; u=ebx.4,ds.2,(GLBLOW,sp+20..,GLBHIGH) d=dl.1
2. 2 sub    dl.1{5}, #0x61.1, dl.1{6} ; 401501 u=dl.1       d=dl.1
2. 3 jz     dl.1{6}, #0.1, @7       ; 401504 u=dl.1
```

(numbers in curly braces are value numbers)

The final microcode is:

```
        jz [ds.2:([ds.2:(ebx.4+#4.4)].4+#1.4)].1, #0x61.1, @7
```

I would not call this code "very simple" but it is ready to be translated to ctree. It maps to C in a natural way. The output will look like this:

```
        if ( argv[1][1] == 'a' )
          ...
```

I have to admit that reading microcode is not easy (but hey, it was not designed for that! :) Seriously, we will think of improving the readability.

We implemented the translation from processor instructions to microinstructions in plain C++. We do not use automatic code generators or machine descriptions to generate them. Anyway there are too many processor specific details to make them feasible.

Microcode details

Each microinstruction has 3 operands:

```
        opcode left, right, destination
```

where **left/right/destination** are the operands. Some of them may be missing for some instructions. Let us quickly enumerate the microinstruction opcodes. The following groups exist:

- constants and moves
- changing operand size
- loading from memory and storing to memory
- comparisons
- arithmetic and bitwise operations
- shifts
- generating condition codes
- unconditional flow control
- conditional jumps
- floating point operations
- miscellaneous

Opcodes: constants and move

There are just 2 instructions in this group:

```
ldc  l,    d   // load constant
mov  l,    d   // move
```

They copy a value from (l)eft to (d)estination. The operand sizes of 'l' and 'd' must be the same.

Opcodes: changing operand size

Copy from (l) to (d)estination

```
xds  l,    d   // extend (signed)
xdu  l,    d   // extend (unsigned)
low  l,    d   // take low part
high l,    d   // take high part
```

They too copy a value from (l)eft to (d)estination. However, the operand sizes of 'l' and 'd' must differ. Since real world programs work with partial registers (like **al, ah**), we absolutely need **low/high**.

Opcodes: load and store

We have only 2 instructions that explicitly work with memory:

```
stx  l, sel, off // store value to memory
ldx  sel, off, d // load value from memory
```

where **{sel, off}** is a **segment:offset** pair. Usually **seg** is **ds** or **cs**; for processors with flat memory it is ignored. **off** is the most interesting part, it is a memory address. Example:

```
ldx ds.2, (ebx.4+#4.4), eax.4
stx #0x2E.1, ds.2, eax.4
```

Opcodes: comparisons

The standard set of comparisons, nothing unexpected:

```
sets  l,    d  // sign bit
setp  l, r, d  // unordered/parity
setnz l, r, d  // not equal
setz  l, r, d  // equal
setae l, r, d  // above or equal
setb  l, r, d  // below
seta  l, r, d  // above
setbe l, r, d  // below or equal
setg  l, r, d  // greater
setge l, r, d  // greater or equal
setl  l, r, d  // less
setle l, r, d  // less or equal
seto  l, r, d  // overflow of (l-r)
```

They compare (l)left against (r)right. The result is stored in (d)estination, a bit register like CF,ZF,SF,...

Opcodes: arithmetic and bitwise operations

This group of instructions won't cause any difficulties in understanding :)

```
neg  l,    d   // -l    -> d
lnot l,    d   // !l    -> d
bnot l,    d   // ~l    -> d
add  l, r, d   // l + r -> d
sub  l, r, d   // l - r -> d
mul  l, r, d   // l * r -> d
udiv l, r, d   // l / r -> d
sdiv l, r, d   // l / r -> d
umod l, r, d   // l % r -> d
smod l, r, d   // l % r -> d
or   l, r, d   // bitwise or
and  l, r, d   // bitwise and
xor  l, r, d   // bitwise xor
```

Operand sizes must be the same. The result is stored in (d)estination.

Opcodes: shifts (and rotations?)

```
shl  l, r, d   // shift logical left
shr  l, r, d   // shift logical right
sar  l, r, d   // shift arithmetic right
```

Shift (l)eft by the amount specified in (r)ight. The result is stored into (d)estination. Initially our microcode had rotation operations but they turned out to be useless because they can not be nicely represented in C.

Opcodes: condition codes

We need these instructions to precisely track carry and overflow bits. Normally these instructions get eliminated during microcode transformations.

```
cfadd l, r, d   // carry    of (l+r)
ofadd l, r, d   // overflow of (l+r)
cfshl l, r, d   // carry    of (l<>r)
```

Perform the operation on (l)left and (r)ight. Generate carry or overflow bits. Store CF or OF into (d)estination.

Opcodes: unconditional flow control

```
ijmp {sel, off}    // indirect jmp
goto  l            // unconditional jmp
call  l      d     // direct call
icall {sel, off} d // indirect call
ret                // return
```

Initially calls have only the callee address. The decompiler retrieves the callee prototype from the database or tries to guess it. After that the 'd' operand contains all information about the call, including the function prototype and actual arguments. Example (the 'd' operand is enclosed in the angle brackets):

```
call $___org_fprintf <...:
        â€œFILE *â€ &($stdout).4,
        "const char *" &($aArIllegalSwitc).4,
        _DWORD xds.4([ds.2:([ds.2:(ebx.4+#4.4)].4+#1.4)].1)>.0
```

Opcodes: conditional jumps

Again, a very familiar set of instructions, many processors have them exactly like this:

```
jcnd  l,    d   // any arbitrary condition
jnz   l, r, d   // ZF=0          Not Equal
jz    l, r, d   // ZF=1          Equal
jae   l, r, d   // CF=0          Above or Equal
jb    l, r, d   // CF=1          Below
ja    l, r, d   // CF=0 & ZF=0   Above
jbe   l, r, d   // CF=1 | ZF=1   Below or Equal
jg    l, r, d   // SF=OF & ZF=0  Greater
jge   l, r, d   // SF=OF         Greater or Equal
jl    l, r, d   // SF!=OF        Less
jle   l, r, d   // SF!=OF | ZF=1 Less or Equal
jtbl  l, cases  // Table jump
```

Compare (l)eft against (r)right and jump to (d)estination if the condition holds. **jtbl** is used to represent 'switch' idioms.

Opcodes: floating point operations

```
f2i   l,    d   // int(l) => d; convert fp -> int, any size
f2u   l,    d   // uint(l)=> d; convert fp -> uint,any size
i2f   l,    d   // fp(l)  => d; convert int -> fp, any size
i2f   l,    d   // fp(l)  => d; convert uint-> fp, any size
f2f   l,    d   // l      => d; change fp precision
fneg  l,    d   // -l     => d; change sign
fadd  l, r, d   // l + r  => d; add
fsub  l, r, d   // l - r  => d; subtract
fmul  l, r, d   // l * r  => d; multiply
fdiv  l, r, d   // l / r  => d; divide
```

Basically we have conversions and a few arithmetic operations. There is little we can do with these operations, they are not really optimizable. Other fp operations (like **sqrt**) use helper functions.

Opcodes: miscellaneous

```
nop             // no operation
und        d    // undefine
ext  l, r, d    // external insn
push l
pop        d
```

Some operations can not be expressed in microcode. If possible, we use intrinsic calls for them (e.g. **sqrtpd**). If no intrinsic call exists, we use **ext** for them and only try to keep track of data dependencies (e.g. we know that **aam** uses **ah** and modifies **ax**). **und** is used when a register is spoiled in a way that we can not predict or describe (e.g. **ZF** after **mul**).

This completes enumerating of 72 instructions that are currently defined in the microcode. Probably we should extend microcode, for example, to include the ternary operator and pre/post increment/decrement operators. However, we are not in hurry to do so because it is not obvious if benefits overweight the added complexity.

Operand types

As everyone else, initially we had only constant integer numbers and registers. All textbooks use these 2 operand types, this is why :) Second, registers and plain numbers are the easiest things to handle.

Life was simple and easy in the good old days! Alas, the reality is more diverse. In order to be able to represent real software, we quickly added:

- stack variables
- global variables
- address of an operand
- list of cases (for switches)
- result of another instruction
- helper functions
- call arguments
- string and floating point constants

Let us inspect them.

Register operands

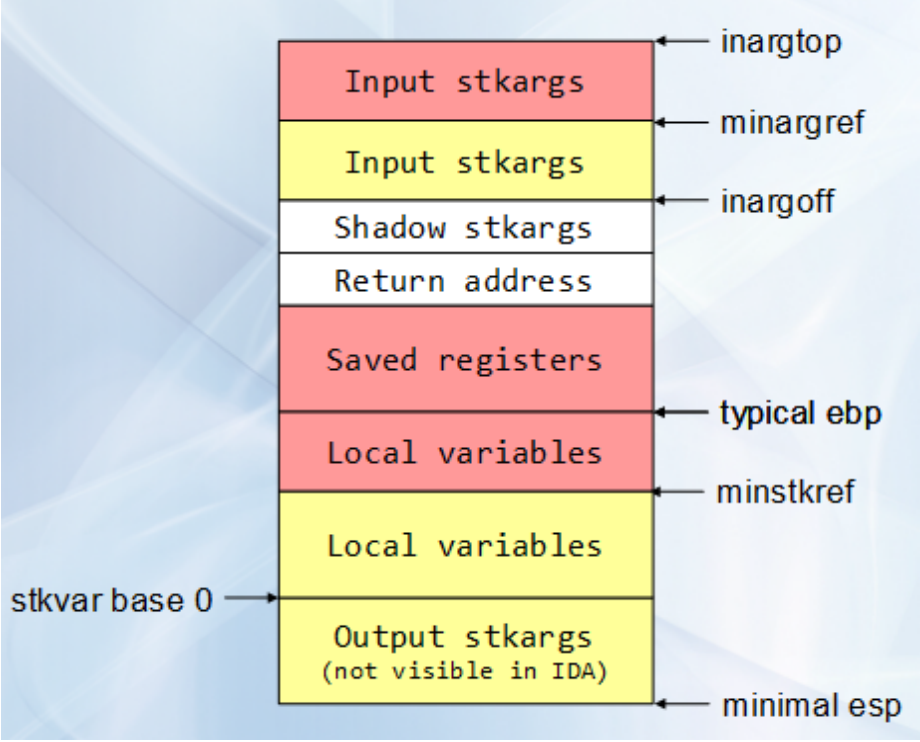
Our microcode engine provides unlimited number of microregisters (unlimited at least in theory). Processor registers are mapped to microregisters. Usually there are more microregisters than the processor registers. We allocate them as needed when generating microcode. The microregister names follow the processor register names (for simplicity), with some exceptions. For example:

```
AL is mapped into al.1 (mreg number 8)
AH is mapped into ah.1 (mreg number 9)
EAX is mapped into eax.4 (mreg numbers 8-11)
RSI is mapped into rsi.8
```

Microregisters are considered to be little endian even for big engian platforms. This is logical because the endianness is an attribute of the memory, not the processor itself.

Stack as viewed by the decompiler

The function stack frame is modeled the following way:



The yellow part is mapped to microregisters. The red part is aliasable.

The "output stkargs" region is a notion that does not exist in IDA. The lowest part of the frame for IDA are the "local variables" region, offsets below would have negative values.

However, we need more precise modelling of the frame in the decompiler because we want to track and reason about the outgoing stack arguments. This is why the stack offsets in IDA and the decompiler are different. In the decompiler the stack offsets are never negative.

More operand types!

When compilers need to manipulate 64-bit values in a 32-bit platform, they usually use a standard pair of registers like **edx:eax**. My initial idea was to map **EAX** and **EDX** registers into contiguous registers:

```
EAX maps to 8-11
EDX maps to 12-15
EDX:EAX naturally maps to 8-15
```

Unfortunately (or fortunately?) compilers get better and nowadays may use any registers as a pair; or even pair a stack location with a register: **sp+4:esi**. Therefore we ended up with a new operand type: an **operand pair**. It consists of low and high halves. They can be located anywhere (stack, registers, memory). Examples:

```
        :(edx.4:eax.4).8 holds a 64-bit value; edx is the high part, eax is the low
part
        :(rsi.8:rdi.8).16 holds a 128-bit value
```

Scattered operands

The nightmare has just begun, in fact. Modern compilers use very intricate rules to pass structs and unions by value to and from the called functions. This is dictated by ABI (example: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf)

A register like RDI may contain multiple structure fields. Some structure fields may be passed on the stack, some in the floating registers, some in general registers (unaligned wrt register start). We had no other choice but to add **scattered operands** that can represent all the above.

For example, a function that returns a struct in rax:

```
struct div_t { int quot; int rem; };
div_t div(int numer, int denom);
```

Assembler code:

```
mov     edi, esi
mov     esi, 1000
call    _div
movsxd  rdx, eax
sar     rax, 20h
add     [rbx], rdx
imul    eax, 1000
cdqe
add     rax, [rbx+8]
```

and the output is:

```
v2 = div(a2, 1000);
*a1 += v2.quot;
result = a1[1] + 1000 * v2.rem;
```

Our decompiler managed to represent things nicely! Similar or more complex situations exist for all 64-bit processors. Support for scattered operands is not complete yet but we continue to improve it.

Microcode transformations

The initial **preoptimization** step uses a very simple constant and register propagation algorithm. It is very fast and gets rid of most temporary registers and reduces the microcode size by two. After this step we use a more sophisticated propagation algorithm. It also works on the basic block level. It is much slower than the preoptimization step, but it can

- handle partial registers (propagate **eax** into an expression that uses **ah**)
- move an entire instruction inside another
- work with operands other that registers (stack and global memory, pairs, and scattered operands)

At the next step we build the control flow graph and perform data flow analysis to find where each operand is used or defined. The use/def information is used to:

- delete dead code (if the instruction result is not used, then we delete the instruction)
- propagate operands and instructions across block boundaries

- generate assertions for future optimizations (we know that **eax** is zero at the target of **jz eax** if there are no other predecessors; so we generate **mov 0, eax**). These assertions can be propagated and lead to more simplifications.

We implemented (in plain C++) hundreds of very small optimization rules. For example:

```
(x-y)+y    => x
x- ~y      => x+y+1
x*m-x*n    => x*(m-n)
(x<<n)-x   => (2**n-1)*x
-(x-y)     => y-x
(~x) < 0   => x >= 0
(-x)*n     => x*-n
```

These rules are simple and sound. They apply to all cases without exceptions. And they do not depend on the compiler. This is one of the reasons why our decompiler does not require the exact version of a compiler that was used to build the input file: almost all our rules are generic. Naturally, compiler specific stuff exists but we try to handle it with configuration parameters. For more details please see the "Compiler" dialog and IDA and the **hexrays.cfg** configuration file.

There are more complex rules than the above. For example, this rule recognizes 64-bit subtractions:

```
CMB18 (combination rule #18):
  sub xlow.4, ylow.4, rlow.4
  sub xhigh.4, (xdu.4((xlow.4 <u ylow.4))+yhigh.4), rhigh.4
=>
  sub x.8, y.8, r.8

if yhigh is zero, then it can be optimized away

a special case when xh is zero:

  sub    xl, yl, rl
  neg    (xdu(lnot(xl >=u yl))+yh), rh
```

We have a swarm of rules like this. They work together like little ants and the outcome is bigger than a simple sum of parts.

Unfortunately we do not have a language to describe these rules, so we manually implemented these rules in C++. However, our pattern recognition does not naively check if the previous or next instruction is the expected one. We use data dependencies to find the instructions that form the pattern. For example, the CMB43 rule looks for the **low** instruction by searching forward for an instruction that accesses **x**:

```
CMB43:
  mul #(1<<N).4, xl.4, yl.4
  low (x.8 >>a #M.1), yh.4, M == 32-N

  =>

  mul x.8, #(1<<N).8, y.8
```

It will successfully find **low** in the following sequence:

```
mul #8.4, eax.4, ecx.4
  add #124.4, esi.4, esi.4
low (rax.8 >>a #29.1), ebx.4, M == 32-N
```

The **add** instruction does not modify **rax** and will be simply skipped.

There are also rules that work across multiple blocks:

```
BLK1 jl xh, yh, SUCCESS
BLK2 jg xh, yh, @FAILED
BLK3 jb xl, yl, SUCCESS
BLK4 ... (FAILED)

BLK5 ... (SUCCESS)
```

where:
xh means high half of x
xl means low half of x
yh means high half of y
yl means low half of y

The **64bit 3-way check** rule transforms this structure into simple:

```
jl x, y, SUCCESS
FAILED: ...

SUCCESS: ...
```

Another example: signed division by power2. Signed division is sometimes replaced by a shift:

```
BLK1 jcnd !SF(x), BLK3
BLK2 add x, (1<<N)-1, x
BLK3 sar x, N, r
```

There is a simple rule that transforms it back:

```
sdiv x, (1<<N), r
```

Microcode hooks

I'm happy to tell you that it is possible to write plugins for the decompiler. Plugins can invoke the decompiler engine and use the results, or improve the decompiler output. It is also possible to use the microcode to find the possible register values at any given point, compare blocks of code, etc.

It is possible to hook to the optimization engine and add your own transformation rules. Please check the Decompiler SDK: it has many examples.

While it is possible to add new rules, currently it is not possible to disable an existing rule However, since almost all of them are sound and do not use heuristics, it is not a problem. In fact the processor specific parts of the decompiler internally use these hooks as well.

For example, the ARM decompiler has the following rule:

```
        ijmp cs, initial_lr => ret
```

so that a construct like **BX LR** will be converted into **RET**. However, we have to prove that the value of LR at the "BX LR" instruction is equal to the initial value of LR at the entry point. How do we find if we jump to the initial_lr? We need to use data flow analysis.

Data flow analysis

Virtually all transformation rules are based on data flow analysis. Very rarely we check the previous or the next instruction for pattern matching. Instead, we calculate the use/def lists for the instruction and search for the instructions that access them. We keep track of what is used and what is defined by every microinstruction (in red). These lists are calculated when necessary:

```
mov    %argv.4, ebx.4     ; 4014E9 u=arg+4.4     d=ebx.4
mov    %argc.4, edi.4     ; 4014EC u=arg+0.4     d=edi.4
mov    &($dword_41D128).4, ST18_4.4 ; 4014EF u=       d=ST18_4.4
goto   @12                ; 4014F6 u= d=
```

where u stands for **use** and d stands for **define**.

Similar blocks are maintained for each block. Instead of calculating them on request we keep them precalculated:

```
; 1WAY-BLOCK 6 INBOUNDS: 5 OUTBOUNDS: 58 [START=401515 END=401517]
; USE: ebx.4,ds.2,(GLBLOW,GLBHIGH)
; DEF: eax.4,(cf.1,zf.1,sf.1,of.1,pf.1,edx.4,ecx.4,fps.2,fl.1,
;             c0.1,c2.1,c3.1,df.1,if.1,ST00_12.12,GLBLOW,GLBHIGH)
; DNU: eax.4
```

For each block we keep both **must** and **may** access lists. The values in parenthesis are part of the **may** list. For example, an indirect memory access may read any memory. This is why **GLBLOW** and **GLBHIGH** are present in the **may** list:

```
add [ds.2:(ebx.4+#4.4)].4, #2.4, ST18_4.4 ; u=ebx.4,ds.2,(GLBLOW,GLBHIGH) d=ST18_4.4
```

Based on use-def lists of each block the decompiler can build global use-def chains and answer questions like:

- Is a defined value used anywhere? If yes, where exactly? Just one location? If yes, what about moving the definition there? If the value is used nowhere, what about deleting it?
- Where does a value come from? If only from one location, can we propagate (or even move) it?
- What are the values are the used but never defined? These are the candidates for input arguments
- What are the values that are defined but never used but reach the last block? These are the candidates for the return values.

Here is an example how we use the data flow analysis. Image we have code like this:

```
BLK1 mov #5.4, esi.4

BLK2 Do some stuff
     that does not modify esi.4

BLK3 call func(esi.4)
```

We calculate the use-def chains for all blocks. They clearly show that esi is defined only in block #1:

```
BLK1 mov #5.4, esi.4                    use:
                                        def: esi.4{3}

BLK2 Do some stuff                      use: ...
     that does not modify esi.4         def: ...
```

```
BLK3 call func(esi.4)                  use: esi.4{1}
                                       def: ...
```

Therefore it can be propagated:

```
BLK3 call func(#5.4)
```

Publishing microcode

The microcode API for C++ is available now, it is shipped together with the decompiler (see the plugins/hexrays subdirectory). The Python API is not available yet, we will add it later. Please check out the sample plugins that show how to use the new API.

Our decompiler verifies the microcode for consistency after every transformation. Thanks to this very strict approach we receive very few microcode related bug reports. Second, we have quite extensive test suites that constantly grow. A hundred or so of processors cores run tests endlessly, this also helps us to discover and fix bugs before they cause inconveniences to anyone.

However, there is still a chance of a bug leaking to the release. So, found a bug? Please send us the database with the description how to reproduce it. We solve most problems within one day or even faster.

Any technical feedback or challenges? We love that, bring them on! :)