Decompiler internals: microcode

Hex-Rays
Ilfak Guilfanov
Presentation Outline

- Decompiler architecture
- Overview of the microcode
- Opcodes and operands
- Stack and registers
- Data flow analysis, aliasability
- Microcode availability
- Your feedback

Online copy of this presentation is available at
Hex-Rays Decompiler

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

- It uses very straightforward sequence of steps:

  1. Generate microcode
  2. Transform microcode (optimize, resolve memrefs, analyze calls, etc)
  3. Allocate local vars
  4. Generate ctree
  5. Beautify ctree
  6. Print ctree
Decompiler architecture

- We will focus on the first two steps:
  - Generate microcode
  - Transform microcode (optimize, resolve memrefs, analyze calls, etc)
  - Allocate local vars
  - Generate ctree
  - Beautify ctree
  - Print ctree
Why microcode?

- It helps to get rid of the complexity of processor instructions
- Also we get rid of processor idiosyncrasies. Examples:
  - 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
- Writing a decompiler without an intermediate language looks like waste of time
Is implementing an IR difficult?

- Your call :)
- How many IR languages do you know?
Why not use an existing IR?

- There are tons of other intermediate languages: LLVM, REIL, Binary Ninja's ILs, RetDec's IL, etc.
- Yes, we could use something
- But I started to work on the microcode when none of the above languages existed
- This is the main reason why we use our own IR

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

(this is how it looked like in 1999)
A long evolution

- I started to work on the microcode in 1998 or earlier
- The name is nothing fancy but reflects the nature of it
- Some design decisions turned out to be bad (and some of them are already very difficult to fix)
- For example, the notion of virtual stack registers
- We will fix it, though. Just takes time
- Even today we modify our microcode when necessary
- For example, I reshuffled the instruction opcodes for this talk...
Design highlights

- **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

- **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. Example:

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. 0</td>
<td>mov ebx.4, eoff.4</td>
<td>; 4014FB u=ebx.4 d=eoff.4</td>
</tr>
<tr>
<td>2. 1</td>
<td>mov ds.2, seg.2</td>
<td>; 4014FB u=ds.2 d=seg.2</td>
</tr>
<tr>
<td>2. 2</td>
<td>add eoff.4, #4.4, eoff.4</td>
<td>; 4014FB u=eoff.4 d=eoff.4</td>
</tr>
<tr>
<td>2. 3</td>
<td>ldx seg.2, eoff.4, et1.4</td>
<td>; 4014FB u=eoff.4,seg.2, (STACK,GLBMEM) d=et1.4</td>
</tr>
<tr>
<td>2. 4</td>
<td>mov et1.4, eax.4</td>
<td>; 4014FB u=et1.4 d=eax.4</td>
</tr>
<tr>
<td>2. 5</td>
<td>mov eax.4, eoff.4</td>
<td>; 4014FE u=eax.4 d=eoff.4</td>
</tr>
<tr>
<td>2. 6</td>
<td>mov ds.2, seg.2</td>
<td>; 4014FE u=ds.2 d=seg.2</td>
</tr>
<tr>
<td>2. 7</td>
<td>add eoff.4, #1.4, eoff.4</td>
<td>; 4014FE u=eoff.4 d=eoff.4</td>
</tr>
<tr>
<td>2. 8</td>
<td>ldx seg.2, eoff.4, t1.1</td>
<td>; 4014FE u=eoff.4,seg.2, (STACK,GLBMEM) d=t1.1</td>
</tr>
<tr>
<td>2. 9</td>
<td>mov t1.1, dl.1</td>
<td>; 4014FE u=t1.1 d=dl.1</td>
</tr>
<tr>
<td>2.10</td>
<td>mov #0x61.1, t1.1</td>
<td></td>
</tr>
<tr>
<td>2.11</td>
<td>setb dl.1, t1.1, cf.1</td>
<td>; 401501 u=dl.1,t1.1 d=cf.1</td>
</tr>
<tr>
<td>2.12</td>
<td>seto dl.1, t1.1, of.1</td>
<td>; 401501 u=dl.1,t1.1 d=of.1</td>
</tr>
<tr>
<td>2.13</td>
<td>sub dl.1, t1.1, dl.1</td>
<td>; 401501 u=dl.1,t1.1 d=dl.1</td>
</tr>
<tr>
<td>2.14</td>
<td>setz dl.1, #0.1, zf.1</td>
<td>; 401501 u=dl.1 d=zf.1</td>
</tr>
<tr>
<td>2.15</td>
<td>setp dl.1, #0.1, pf.1</td>
<td>; 401501 u=dl.1 d=pf.1</td>
</tr>
<tr>
<td>2.16</td>
<td>sets dl.1, sf.1</td>
<td>; 401501 u=dl.1 d=sf.1</td>
</tr>
<tr>
<td>2.17</td>
<td>mov cs.2, seg.2</td>
<td>; 401504 u=cs.2 d=seg.2</td>
</tr>
<tr>
<td>2.18</td>
<td>mov #0x401517.4, eoff.4</td>
<td>; 401504 u= d=eoff.4</td>
</tr>
<tr>
<td>2.19</td>
<td>jcnd zf.1, $loc_401517</td>
<td>; 401504 u=zf.1</td>
</tr>
</tbody>
</table>
The first optimization pass

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 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 |

- Only 8 microinstructions
- Some intermediate registers disappeared
- Sub-instructions appeared
- Still too noisy and verbose
Further microcode transformations

And the final code is:

```
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
```

This code is ready to be translated to ctree. (numbers in curly braces are value numbers)

The output will look like this:

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

- Reading microcode is not easy (but hey, it was not designed for that! :) )
- All operand sizes are spelled out explicitly
- The initial microcode is very simple (RISC like)
- As we transform microcode, nested subinstructions may appear
- 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
Opcodes: constants and move

- Copy from (l) to (d)estination
- Operand sizes must match

```
ldc  l,    d   // load constant
mov  l,    d   // move
```
Opcodes: changing operand size

- Copy from (l) to (d)estination
- Operand sizes must differ
- Since real world programs work with partial registers (like al, ah), we absolutely need low/high

```
xds l, d // extend (signed)
xdu l, d // extend (unsigned)
low l, d  // take low part
high l, d // take high part
```
Opcodes: load and store

- \{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

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

Example:

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

- Compare (l)left against (r)right
- The result is stored into (d)estination, a bit register like CF,ZF,SF,...

```assembly
sets  l,    d  // sign
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
setge l, r, d  // greater or equal
setg  l, r, d  // greater
setle l, r, d  // less or equal
setl  l, r, d  // less
seto  l, r, d  // overflow of (l-r)
```
Opcodes: arithmetic and bitwise operations

- Operand sizes must be the same
- The result is stored into (d)estination

```
  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
```
Opcodes: shifts (and rotations?)

- 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 cannot be nicely represented in C

```
shl l, r, d  // shift logical left
shr l, r, d  // shift logical right
sar l, r, d  // shift arithmetic right
```
Opnodes: condition codes

- Perform the operation on (l)eft and (r)ight
- Generate carry or overflow bits
- Store CF or OF into (d)estination
- We need these instructions to precisely track carry and overflow bits
- Normally these instructions get eliminated during microcode transformations

```plaintext
cfadd l, r, d   // carry of (l+r)
ofadd l, r, d   // overflow of (l+r)
cfshr l, r, d   // carry of (l>>r)
cfshr l, r, d   // carry of (l>>r)
```
Opcodes: unconditional flow control

- 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

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

call $___org_fprintf <...:
  “FILE *” &($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

- Compare (l)eft against (r)ight and jump to (d)estination if the condition holds
- Jtbl is used to represent 'switch' idioms

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>jcnd</td>
<td>1, d</td>
<td>Not Equal</td>
</tr>
<tr>
<td>jnz</td>
<td>1, r, d</td>
<td>ZF=0</td>
</tr>
<tr>
<td>jz</td>
<td>1, r, d</td>
<td>ZF=1</td>
</tr>
<tr>
<td>jae</td>
<td>1, r, d</td>
<td>CF=0</td>
</tr>
<tr>
<td>jb</td>
<td>1, r, d</td>
<td>CF=1</td>
</tr>
<tr>
<td>ja</td>
<td>1, r, d</td>
<td>CF=0 &amp; ZF=0</td>
</tr>
<tr>
<td>jbe</td>
<td>1, r, d</td>
<td>CF=1</td>
</tr>
<tr>
<td>jg</td>
<td>1, r, d</td>
<td>SF=OF &amp; ZF=0</td>
</tr>
<tr>
<td>jge</td>
<td>1, r, d</td>
<td>SF=OF</td>
</tr>
<tr>
<td>jl</td>
<td>1, r, d</td>
<td>SF!=OF</td>
</tr>
<tr>
<td>jle</td>
<td>1, r, d</td>
<td>SF!=OF</td>
</tr>
<tr>
<td>jtbl</td>
<td>1, cases</td>
<td>Table jump</td>
</tr>
</tbody>
</table>

(c) 2018 Ilfak Guilfanov
Opcodes: floating point operations

- 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 use helper functions (e.g. sqrt)

```
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
```
Opcodes: miscellaneous

- 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. “aam”)
- “und” is used when a register is spoiled in a way that we can not predict or describe (e.g. ZF after mul)

```
 nop           // no operation
 und          d // undefined
 ext l, r, d   // external insn
 push l
 pop          d
```
More opcodes?

- We quickly reviewed all 72 instructions
- Probably we should extend microcode
- Ternary operator?
- Post-increment and post-decrement?
- All this requires more research
Operands!

- As everyone else, initially we had only:
  - constant integer numbers
  - registers
- Life was simple and easy in the good old days!
- Alas, the reality is more diverse. 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
Register operands

- The microcode engine provides unlimited (in theory) number of microregisters
- Process registers are mapped to microregisters:
  - eax => microregisters (mreg) 8, 9, 10, 11
  - al => mreg 8
  - ah => mreg 9
- Usually there are more microregisters than the processor registers. We allocate them as needed when generating microcode
- Examples:

  ```
  eax.4
  rsi.8
  ST00_04.4
  ```
Stack as microregisters

- I was reluctant to introduce a new operand type for stack variables and decided to map the stack frame to microregisters.
- Like, the stack frame is mapped to the microregister #100 and higher.
- A bright idea? Nope!
- Very soon I realized that we have to handle indirect references to the stack frame.
- Not really possible with microregisters.
- But there was so much code relying on this concept that we still have it.
- Laziness pays off now and in the future (negatively).
Stack as viewed by the decompiler

- Yellow part is mapped to microregisters
- Red is aliasable

Stack diagram:

- **Input stkargs**
- **Shadow stkargs**
- **Return address**
- **Saved registers**
- **Local variables**
- **Output stkargs**
  (not visible in IDA)

**Key Areas:***
- **inargtop**
- **minargref**
- **inargoff**
- **typical ebp**
- **minstkref**
- **minimal esp**

Yellow part is mapped to microregisters.
Red is aliasable.
More operand types!

- 64-bit values are represented as pairs of registers
- Usually it is a standard pair like edx:eax
- Compilers get better and nowadays use any registers as a pair; or even pair a stack location with a register: sp+4:esi
- We ended up with a new operand type:
  - operand pair
- It consists of low and high halves
- They can be located anywhere (stack, registers, glbmem)
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
- 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
A simple scattered return value

- A function that returns a struct in rax:

```c
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]
```
A simple scattered return value

...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 constantly improve it
More detailed look at microcode transformations

- The initial “preoptimization” step uses very simple constant and register propagation algorithm
- It is very fast
- It gets rid of most temporary registers and reduces the microcode size by two
- Normally we use a more sophisticated propagation algorithm
- It also works on the basic block level
- It is much slower but can:
  - handle partial registers (propagate eax into an expression that uses ah)
  - move entire instruction inside another
  - work with operands other that registers (stack and global memory, pair and scattered operands)
Global optimization

- We build the control flow graph
- 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”)
Synthetic assertion instructions

- If jump is not taken, then we know that eax is zero

  
  jnz eax.4, #0, @5

- Assertions can be propagated and lead to more simplifications

  
  blk5: ...
  mov #0.4, eax.4 ; assert ...

  true

  false
Simple algebraic transformations

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

  \[
  \begin{align*}
  (x-y)+y & \Rightarrow x \\
  x-\sim y & \Rightarrow x+y+1 \\
  x\ast m-x\ast n & \Rightarrow x\ast (m-n) \\
  (x\ll n)-x & \Rightarrow (2\ll n-1)\ast x \\
  -(x-y) & \Rightarrow y-x \\
  (\sim x) < 0 & \Rightarrow x \geq 0 \\
  (-x)\ast n & \Rightarrow x\ast-n
  \end{align*}
  \]

- They are simple and sound
- They apply to all cases without exceptions
- Overall the decompiler uses sound rules
- They do not depend on the compiler
More complex rules

- 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), rhhigh.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   x1, y1, r1
  neg   (xdu(lnot(x1 >=u y1))+yh), rh

- We have a swarm of rules like this. They work like little ants :)

(c) 2018 Ilfak Guilfanov
Data dependency dependent rules

- Naturally, all these rules are compiler-independent, they use common algebraic number properties.
- Unfortunately we do not have a language to describe these rules, so we manually added these rules in C++.
- However, the 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 rule CMB43 looks for the 'low' instruction by searching forward for an instruction that accesses the 'x' operand:

```
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
```
Interblock rules

- Some rules work across multiple blocks:

  \[ jl \ xh, \ yh, \ SUCCESS \]

  \[ jg \ xh, \ yh, \ @4 \]

  \[ jb \ xl, \ yl, \ SUCCESS \]

  FAILED: ...

  SUCCESS: ...

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

\[ jl \ x, \ y, \ SUCCESS \]

SUCCESS: ...

FAILED: ...

(xh means high half of x
xl means low half of x
yh means high half of y
yl means low half of y)
Signed division is sometimes replaced by a shift:

```
jcnd !SF(x), b3
```

```
add x, (1<<N)-1, x
```

```
sar x, N, r
```

A simple rule transforms it back:

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

- It is possible to hook to the optimization engine and add your own transformation rules
- The Decompiler SDK has some examples how to do it
- 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
ARM hooks

- For example, the ARM decompiler has the following rule:

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

  so that a construct like this: BX LR will be converted into: RET

  only if we can prove that the value of LR at the "BX LR" instruction is equal to the initial value of LR at the entry point.

- However, how do we find if we jump to the initial_lr? Data analysis is to help us
Data flow analysis

- In fact 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=
```
Use-def lists

- 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
```

- 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:

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

- 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
Global propagation in action

- 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)
Global propagation in action

- The use-def chains 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
  that does not modify esi.4
  use: ...
  def: ...

  blk3
  call func(esi.4)
  use: esi.4{1}
  def: ...

Therefore it can be propagated:

  call func(#5.4)
Data flow analysis

- The devil is in details
- Our analysis engine can handle partial registers (they are a pain)
- Big endian and little endian can be handled as well (however, we sometimes end up with the situations when a part of the operand is little endian and another part – big endian)
- The stack frame and registers are handled
- Registers can be addressed only directly
- Stack location can be addressed indirectly and our analysis takes this into account
- Well, we have to make some assumptions...
Aliasability

- Take this example:

```c
mov #1.4, %stkvar ; store 1 into stkvar
stx #0.4, ds.2, eax.4 ; store 0 into [eax]
call func(%stkvar)
```

can we claim that %stkvar == 1 after stx?

- Naturally, in general case we can not
- But it turns out that in some case we can claim it
- Namely:
  - If we haven't taken the address of any stack variable
  - Or, if we did, the address we took is higher (*)
  - Or, if the address is lower, it was not moved into eax
- Overall it is a tough question

(*)note: yes, this is one of the assumptions our decompiler makes
Stack as viewed by the decompiler

- Yellow part is mapped to microregisters
- Red is aliasable

- **Input stkargs**
- **Shadow stkargs**
- **Return address**
- **Saved registers**
- **Local variables**
- **Output stkargs** (not visible in IDA)

- stkvar base 0

- inargtop
- minargref
- inargoff
- typical ebp
- minstkref
- minimal esp

(c) 2018 Ilfak Guilfanov
Minimal stack reference

- Aliasability is unsolvable problem in general
- We should optimize things only if we can prove the correctness of the transformation
- We keep track of expressions like &stkvar and calculate the minimal reference (minstkref)
- We assume that everything below minstkref can be accessed only directly, i.e. is not aliasable
- We propagate this information over the control graph
- One value is maintained per block (we could probably improve things by calculating minstkref for each instruction)
- A similar value is maintained for the incoming stack arguments (minargref)
Minstkref propagation

- We use the control flow graph:

```plaintext
lea  ecx, [esp+10] ; take offset 10
call func          ; probably uses ecx
mov  rax, [esp+14] ; stkvar sp+14
...

minstkref=10
```

```plaintext
lea  ecx, [esp+20] ; take offset 20
call func          ; probably uses ecx
mov  rax, [esp+14] ; microregister ST14
...

minstkref=20
```

```
mov  rax, [esp+14] ; stkvar sp+14
...

minstkref=10
```
Testing the microcode

- Microcode if verified for consistency after every transformation
- BTW, third party plugins should do the same
- Very few microcode related bug reports
- We have quite extensive test suites that constantly grow
- A hundred or so of processors cores running tests
- However, after publishing microcode there will be a new wave of bug reports
- Found a bug? Send us the database with the description how to reproduce it
- Most problems are solved within one day or faster
Publishing microcode

- The microcode API for C++ will be available in the next version of IDA
- Python API won't be available yet
- We will start beta testing the next week
- Decompiler users with active support: feel free to send an email to support@hex-rays.com if you want to participate
- Check out the sample plugins that show how to use the new API
Was it interesting?

Thank you for your attention!
Questions?