Binary Analysis

Reverse Compilation

Cliff Bakalian, Drake Petersen,
Ivan Quiles, Riley Wilburn

Overview:

Using the Binary Analysis Platform (BAP) developed by Carnegie Mellon University, we set out to construct a decompiler for our CMSC838E class compiler using classical compiler operations and BAP as our foundational component. BAP provides an OCaml API for their framework, which enables creating individual plugins or passes that can be chained together. We developed our passes such that it was clear what each pass offered to improve the decompilation. This is not a complete decompiler, but we came close by producing very understandable, interpretable BIR output. All code and test binaries can be found at https://gitlab.com/drakemp/cmsc838e-decompiler.

Background:

A critical component of a decompiler is the Intermediate Representation (IR) that abstracts from the architecture. Binary Intermediate Representation (BIR) is built from Binary Intermediate Language (BIL). Initially BAP will lift into BIL, translating each assembly instruction into comparable BIL instructions. BIL then can be lifted into BIR which is a semi-graphical representation for semantic analysis, whereas BIL is an AST for syntax analysis.

Once the program is lifted to BIR we are nearly finished using BAP as a front-end component of the decompiler. We then take advantage of the builtin Static Single Assignment (SSA) pass to turn the program into *three address code* to simplify our code optimizations. Our passes are based on standard compiler theory optimizations, such as constant code propagation and dead code elimination.

The target backend would be the last piece of the puzzle where we would attempt to produce C-like target code from our optimized BIR output. Unfortunately time did not permit these last steps, but could have been as simple as translating BIR syntax and wrapping lines and code with C syntax.

The class compiler is not a fully featured compiler and is written to input Racket-like source code, however we take advantage of certain knowns about this compiler such as certain register usage. This compiler also has no looping functionality, so we are able to avoid loop detection and code reorganization passes, though these would still be preferable in the general cases.

Decompilation Step by Step:

We started with making simple programs to compile into binaries. Since the class compiler wraps the assembly output in a C runtime, we ignore code not produced by the compiler itself. We primarily focused on the following programs:

`hello_nostd.rkt`:

```
#lang racket
"hello"
```

`prop_nostd.rkt`:

```
#lang racket
(let ((a 1)
      (b 5)
      (c (+ 1 2))
      (d 3))
     (+ (+ a b) (+ c d)))
```

```
3e50 <entry>:
3e50:  mov  eax,0x1a10
3e55:  mov  QWORD PTR [rdi+0x8],rax
3e59:  mov  eax,0x1950
3e5e:  mov  QWORD PTR [rdi+0x10],rax
3e62:  mov  eax,0x1b10
3e67:  mov  QWORD PTR [rdi+0x18],rax
3e6b:  mov  eax,0x1b10
3e70:  mov  QWORD PTR [rdi+0x20],rax
3e74:  mov  eax,0x1bd0
3e79:  mov  QWORD PTR [rdi+0x28],rax
3e7d:  mov  eax,0x140
3e82:  mov  QWORD PTR [rdi],rax
3e85:  mov  rax,rdi
3e88:  add  rax,0x3
3e8c:  add  rdi,0x30
3e90:  ret
```

```
3e50 <entry>:
3e50:  mov  eax,0x40
3e55:  mov  QWORD PTR [rsp-0x8],rax
3e5a:  mov  eax,0x140
3e5f:  mov  QWORD PTR [rsp-0x10],rax
3e64:  mov  eax,0x80
3e69:  mov  rbx,rax
3e6c:  and  rbx,0x3f
3e70:  cmp  rbx,0x0
3e74:  jne  3f2a <err>
3e7a:  mov  QWORD PTR [rsp-0x18],rax
3e7f:  mov  eax,0x40
3e84:  mov  rbx,rax
3e87:  and  rbx,0x3f
3e8b:  cmp  rbx,0x0
3e8f:  jne  3f2a <err>
…[see Appendix for full]...
```

These two programs when compiled, create a variety of situations for us to tackle with our decompilation. First we see that `hello_nostd` uses the heap (rdi) to store unicode characters for each letter of “hello”, then in `prop_nostd` there is more stack usage for local variables. Another compiler specific detail is the runtime type checking of variables, this poses issues regarding code reorganization, since C-like targets would not need these type checks, but could be kept (using if-statement error checking) to preserve the compiler functionality in the target backend. Either can be handled in a script translator from optimized BIR to C source, but not something we worry about for the time being. You may have noticed the ‘nostd’ in the title, we compiled these binaries

without the standard library option since that previously produced monstrous amounts of code for binaries like `hello_nostd`.

First let's look at the BIL output for `hello_nostd`. BAP presents the BIL with the original assembly instructions so we can see right away how BIL is produced. We can immediately see some instructions are a simple memory or register operation, but then we also see arithmetic operations produce many BIL instructions. This is because BIL needs to verbosely capture all the interactions an instruction performs which includes all side effects.

hello_nostd.bil:

```
$ bap -dbil \
--print-matching=subroutine:entry \
../testbin/hello_nostd.run
```

```
3e50: <entry>
3e50:
3e50: movl $0x1a10, %eax
{
  RAX := 0x1A10
}
3e55: movq %rax, 0x8(%rdi)
{
  mem := mem with [RDI + 8, el]:u64 <- RAX
}
3e59: movl $0x1950, %eax
{
  RAX := 0x1950
}
3e5e: movq %rax, 0x10(%rdi)
{
  mem := mem with [RDI + 0x10, el]:u64 <- RAX
}
3e62: movl $0x1b10, %eax
{
  RAX := 0x1B10
}
3e67: movq %rax, 0x18(%rdi)
{
  mem := mem with [RDI + 0x18, el]:u64 <- RAX
}
3e6b: movl $0x1b10, %eax
{
  RAX := 0x1B10
}
3e70: movq %rax, 0x20(%rdi)
{
  mem := mem with [RDI + 0x20, el]:u64 <- RAX
}
3e74: movl $0x1bd0, %eax
{
  RAX := 0x1BD0
}
3e79: movq %rax, 0x28(%rdi)
{
  mem := mem with [RDI + 0x28, el]:u64 <- RAX
}
3e7d: movl $0x140, %eax
{
  RAX := 0x140
}
3e82: movq %rax, (%rdi)
{
  mem := mem with [RDI, el]:u64 <- RAX
}
3e85: movq %rdi, %rax
{
  RAX := RDI
}
3e88: addq $0x3, %rax
{
  #33 := RAX
  RAX := RAX + 3
  CF := RAX < #33
  OF := ~high:1[#33] & (high:1[#33] | high:1[RAX]) & ~(high:1[#33] & high:1[RAX])
  AF := 0x10 = (0x10 & (RAX ^ #33 ^ 3))
  PF := ~low:1[let $1 = RAX >> 4 ^ RAX in
    let $2 = $1 >> 2 ^ $1 in
    $2 >> 1 ^ $2]
  SF := high:1[RAX]
  ZF := 0 = RAX
}
3e8c: addq $0x30, %rdi
{
  #36 := RDI
  RDI := RDI + 0x30
  CF := RDI < #36
  OF := ~high:1[#36] & (high:1[#36] | high:1[RDI]) & ~(high:1[#36] & high:1[RDI])
  AF := 0x10 = (0x10 & (RDI ^ #36 ^ 0x30))
  PF := ~low:1[let $1 = RDI >> 4 ^ RDI in
    let $2 = $1 >> 2 ^ $1 in
    $2 >> 1 ^ $2]
  SF := high:1[RDI]
  ZF := 0 = RDI
}
3e90: retq
{
  #39 := mem[RSP, el]:u64
  RSP := RSP + 8
  jmp #39
}
```

Taking a look at the BIR we can see it is very similar to the BIL, we've removed the original instructions and each instruction is now an independent line of code that can be manipulated. Right now there are a number of things that need to be optimized to

start looking like high level code such as removing register usages, referenced memory should become variables, and remove dead code

hello_nostd.bir

```
$ bap -dbir --print-matching=subroutine:entry ../testbin/hello_nostd.run
```

```
0000af46: sub entry()
00000303:
00000313: RAX := 0x1A10
0000031a: mem := mem with [RDI + 8, el]:u64 <- RAX
00000321: RAX := 0x1950
00000328: mem := mem with [RDI + 0x10, el]:u64 <- RAX
0000032f: RAX := 0x1B10
00000336: mem := mem with [RDI + 0x18, el]:u64 <- RAX
0000033d: RAX := 0x1B10
00000344: mem := mem with [RDI + 0x20, el]:u64 <- RAX
0000034b: RAX := 0x1BD0
00000352: mem := mem with [RDI + 0x28, el]:u64 <- RAX
00000359: RAX := 0x140
00000360: mem := mem with [RDI, el]:u64 <- RAX
00000367: RAX := RDI
00000375: #33 := RAX
00000378: RAX := RAX + 3
0000037b: CF := RAX < #33
0000037e: OF := ~high:1[#33] & (high:1[#33] | high:1[RAX]) & ~(high:1[#33] & high:1[RAX])
00000381: AF := 0x10 = (0x10 & (RAX ^ #33 ^ 3))
00000384: PF := ~low:1[let $1 = RAX >> 4 ^ RAX in let $2 = $1 >> 2 ^ $1 in
$2 >> 1 ^ $2]
00000387: SF := high:1[RAX]
0000038a: ZF := 0 = RAX
00000398: #36 := RDI
0000039b: RDI := RDI + 0x30
0000039e: CF := RDI < #36
000003a1: OF := ~high:1[#36] & (high:1[#36] | high:1[RDI]) & ~(high:1[#36] & high:1[RDI])
000003a4: AF := 0x10 = (0x10 & (RDI ^ #36 ^ 0x30))
000003a7: PF := ~low:1[let $1 = RDI >> 4 ^ RDI in let $2 = $1 >> 2 ^ $1 in
$2 >> 1 ^ $2]
000003aa: SF := high:1[RDI]
000003ad: ZF := 0 = RDI
000003b6: #39 := mem[RSP, el]:u64
000003b9: RSP := RSP + 8
000003bd: call #39 with noreturn
```

The last heap of code we need to look at is the SSA optimized code before we get into optimizations. SSA forces any use of a value to be defined once and only once, we can use this to our advantage later, but for now let's look at a snip of `hello_nostd` in SSA. As you can see `RAX` may be used multiple times, but every new definition of `RAX` produces a new label so we can guarantee a label does not reassign.

```
0000af46: sub entry()
00000303:
00000313: RAX.1 := 0x1A10
0000031a: mem.1 := mem with [RDI + 8, el]:u64 <- RAX.1
00000321: RAX.2 := 0x1950
00000328: mem.2 := mem.1 with [RDI + 0x10, el]:u64 <- RAX.2
0000032f: RAX.3 := 0x1B10
00000336: mem.3 := mem.2 with [RDI + 0x18, el]:u64 <- RAX.3
[snipped output]
```

Code Optimizations

The first optimization we'll talk about is the dead code elimination (DCE) since we adopted the concept from bap for the other passes. The plugin implementation we took from bap-plugins. BAP has two DCE implementations, the snipped comes from their optimization plugin, where they use a visitor object over each instruction once collecting defined labels and used labels in respective sets.

```
let def_use_collector = object
  inherit [Var.Set.t * Var.Set.t] Term.visitor

  method! enter_def t (defs,uses) =
    Set.add defs (Def.lhs t), Set.union uses (Def.free_vars t)

  method! enter_phi t (defs,uses) =
    Set.add defs (Phi.lhs t), Set.union uses (Phi.free_vars t)

  method! enter_jmp t (defs,uses) =
    defs, Set.union uses (Jmp.free_vars t)
end

let computed_def_use sub =
  def_use_collector#visit_sub sub (Var.Set.empty,Var.Set.empty)
```

Traditionally a Define-Use Chain (du-chain) would aggregate every use instance of a defined label and keep a running list, but instead they use SSA form to speed up this process. We know any time `RAX.1` is used, that instance is still live, whereas if `RAX.20` is defined but never used, then it won't be found in the `uses` set and therefore dead. We created a `duchain` plugin that prints the def-use chain for a program. The first set is the `defs` and the second is the `uses`. With this we can take the difference of sets and determine the dead code to eliminate.

duchain output:

```
$ bap -p ssa,duchain \
--print-matching=subroutine:entry \
../testbin/hello_nostd.run
```

```
{AF.1, AF.2, CF.1, CF.2, OF.1, OF.2, PF.1, PF.2, RAX.1, RAX.2, RAX.3, RAX.4, RAX.5, RAX.6,
RAX.7, RAX.8, RDI.1, RSP.1, SF.1, SF.2, ZF.1, ZF.2, mem.1, mem.2, mem.3, mem.4, mem.5,
mem.6, #33.1, #36.1, #39.1}
{RAX.1, RAX.2, RAX.3, RAX.4, RAX.5, RAX.6, RAX.7, RAX.8, RDI, RDI.1, RSP, mem, mem.1, mem.2,
mem.3, mem.4, mem.5, mem.6, #33.1, #36.1, #39.1}
```

In reality DCE works best after Constant Code propagation (CCP) since that will help remove register uses throughout the program and then later be eliminated by DCE. The way we approached CCP was similar to DCE implementation referenced, we wanted to propagate expressions throughout and we were working with SSA form. First we needed a Use-Def chain (ud-chain), a chain of labels and their expressions, but we could optimize our implementation by keeping a map of labels to their expressions, eliminating the bulk of ud-chain compilation. Then to propagate, we use the map and replace any label with it's mapped expression in every BIR instruction, but also updating the map so the propagated definitions get forward propagated in a single pass (otherwise this would take any number of passes to fully propagate). Let's take a look at hello_nostd with our optimizations so far:

CCP&DCE:

```
$ bap -dbir -p ssa,express-prop,deadcode --print-matching=subroutine:entry
../testbin/hello_nostd.run
```

```
0000af46: sub entry()
00000303:
00000313: RAX.1 := 0x1A10
0000031a: mem.1 := mem with [RDI + 8, el]:u64 <-
0x1A10
00000321: RAX.2 := 0x1950
00000328: mem.2 := mem with [RDI + 8, el]:u64 <-
0x1A10
                    with [RDI + 0x10, el]:u64 <-
0x1950
0000032f: RAX.3 := 0x1B10
00000336: mem.3 := mem with [RDI + 8, el]:u64 <-
0x1A10
                    with [RDI + 0x10, el]:u64 <-
0x1950
                    with [RDI + 0x18, el]:u64 <-
0x1B10
0000033d: RAX.4 := 0x1B10
00000344: mem.4 := mem with [RDI + 8, el]:u64 <-
0x1A10
                    with [RDI + 0x10, el]:u64 <-
0x1950
                    with [RDI + 0x18, el]:u64 <-
0x1B10
                    with [RDI + 0x20, el]:u64 <-
0x1B10
0000034b: RAX.5 := 0x1BD0
00000352: mem.5 := mem with [RDI + 8, el]:u64 <-
0x1A10
                    with [RDI + 0x10, el]:u64 <-
0x1950
                    with [RDI + 0x18, el]:u64 <-
0x1B10
                    with [RDI + 0x20, el]:u64 <-
0x1B10
                    with [RDI + 0x28, el]:u64 <-
0x1BD0
00000359: RAX.6 := 0x140
00000360: mem.6 := mem with [RDI + 8, el]:u64 <-
0x1A10
                    with [RDI + 0x10, el]:u64 <-
0x1950
                    with [RDI + 0x18, el]:u64 <-
0x1B10
                    with [RDI + 0x20, el]:u64 <-
0x1B10
                    with [RDI + 0x28, el]:u64 <-
0x1BD0
                    with [RDI, el]:u64 <- 0x140
00000367: RAX.7 := RDI
00000378: RAX.8 := RDI + 3
0000039b: RDI.1 := RDI + 0x30
000003b6: #39.1 := mem with [RDI + 8, el]:u64 <-
0x1A10
                    with [RDI + 0x10, el]:u64 <-
0x1950
                    with [RDI + 0x18, el]:u64 <-
0x1B10
                    with [RDI + 0x20, el]:u64 <-
```

```
0x1B10
                        with [RDI + 0x28, el]:u64 <-
0x1BD0
                        with [RDI, el]:u64 <-
0x140[RSP, el]:u64
```

```
000003b9: RSP.1 := RSP + 8
000003bd: call #39.1 with noreturn
```

Okay so we can see propagation worked pretty well, but mem states are building up weird, and weirdly `RAX`'s are not getting removed. We attribute the `RAX` business with how BAP interprets the `RAX` as a return register or to the deadcode implementation since `--optimization-level=3` does a better job removing the `RAX`'s but not enough… We aren't using the optimization option BAP offers since we can't control when that pass runs in relation to our other passes.

The last plugin we have to optimize the code is the *variable label creator*. This is required since we know stack items are local variables, and heap references are just part of a large array, which from a high level can be handled more easily with variable names. Looking at the optimized code below we can see it's cleaned up a lot but still has `RAX` registers, and it doesnt understand that the `RDI` register is the same as `heap_0`. From here there are small changes that are needed to improve them to be properly decompiled.

Variable Naming:

```
bap -dbir -p ssa,stack-var,express-prop,deadcode
--print-matching=subroutine:entry ../testbin/hello_nostd.run
```

```
0000af46: sub entry()
00000303:
00000313: RAX.1 := 0x1A10
0000b4f3: heap_0[8] := 0x1A10
00000321: RAX.2 := 0x1950
0000b4f4: heap_0[16] := 0x1950
0000032f: RAX.3 := 0x1B10
0000b4f5: heap_0[24] := 0x1B10
0000033d: RAX.4 := 0x1B10
0000b4f6: heap_0[32] := 0x1B10
0000034b: RAX.5 := 0x1BD0
```

```
0000b4f7: heap_0[40] := 0x1BD0
00000359: RAX.6 := 0x140
0000b4f8: heap_0[0] := 0x140
00000367: RAX.7 := RDI
00000378: RAX.8 := RDI + 3
0000039b: RDI.1 := RDI + 0x30
000003b6: #39.1 := mem.6[RSP,
el]:u64
000003b9: RSP.1 := RSP + 8
000003bd: call #39.1 with noreturn
```

prop_nostd

So we have seen how these code optimizations affect the classic "hello world" example, but next is seeing how these perform on something more complicated, `prop_nostd`. This was named 'prop' since it was our propagation test binary, to see how well constants propagated with several variables. (see here original racket and

here for assembly).

Looking below there's some good and there's some ugly. The last `RAX` is perfectly propagated, `0000b192: RAX.13 := 0x40 + 0x140 + 0x40 + 0x80 + 0xC0`. The compiler's type checking introduces branching which causes separated blocks of code. Depending on how one plans to optimize it may or may not be necessary to keep the type checking.

```
00000343: RBX.2 := 0x80 & 0x3F
00000374: ZF.2 := 0 = (0x80 & 0x3F)
0000037e: when ~ZF.2 call @err with noreturn
0000b4d7: goto %0000ae78
```

We also see that there is some propagation that could be done to remove the uses of `ZF`, and more dead code that could be eliminated. The latter can be fixed by cleaning up the return instructions at the end and making the dead code more aggressive.

```
...
0000b192: RAX.13 := 0x40 + 0x140 + 0x40 + 0x80 + 0xC0
0000b1ad: #1883.1 := mem.8[RSP, el]:u64
0000b1b0: RSP.1 := RSP + 8
0000b1b4: call #1883.1 with noreturn
```

Should become:

```
...
0000b192: RAX.13 := 0x40 + 0x140 + 0x40 + 0x80 + 0xC0
0000b1b4: ret RAX.13
```

Once those passes are fixed, a new pass for removing unnecessary basic blocks (only has type checking jumps) then the final program could be as simple as the snipped directly above.

```
$ bap -dbir -p ssa,stack-var,express-prop,deadcode --print-matching=subroutine:entry
../testbin/prop_nostd.run
```

```
0000b24b: sub entry()
00000303:
00000313: RAX.1 := 0x40
0000b808: local_8 := 0x40
00000321: RAX.2 := 0x140
0000b809: local_16 := 0x140
0000032f: RAX.3 := 0x80
00000336: RBX.1 := 0x80
00000343: RBX.2 := 0x80 & 0x3F
00000374: ZF.2 := 0 = (0x80 & 0x3F)
0000037e: when ~ZF.2 call @err with noreturn
0000b4d7: goto %0000ae78
```

```
0000ae78:
0000b80a: local_24 := 0x80
0000ae84: RAX.4 := 0x40
0000ae8b: RBX.3 := 0x40
0000ae98: RBX.4 := 0x40 & 0x3F
0000aec9: ZF.4 := 0 = (0x40 & 0x3F)
0000aed2: when ~ZF.4 call @err with noreturn
0000b4d8: goto %0000aed8

0000aed8:
0000aeeb: RAX.5 := 0x40 + 0x80
0000b80b: local_24.1 := 0x40 + 0x80
0000af0b: RAX.6 := 0xC0
```

```
0000b80c: local_32 := 0xC0
0000af19: RAX.7 := 0xC0
0000af20: RBX.5 := 0xC0
0000af2d: RBX.6 := 0xC0 & 0x3F
0000af5e: ZF.7 := 0 = (0xC0 & 0x3F)
0000af67: when ~ZF.7 call @err with noreturn
0000b4d9: goto %0000af6d

0000af6d:
0000b80d: local_40 := 0xC0
0000af79: RAX.8 := 0x40 + 0x80
0000af80: RBX.7 := 0x40 + 0x80
0000af8d: RBX.8 := 0x40 + 0x80 & 0x3F
0000afbe: ZF.9 := 0 = (0x40 + 0x80 & 0x3F)
0000afc7: when ~ZF.9 call @err with noreturn
0000b4da: goto %0000afcd

0000afcd:
0000afe0: RAX.9 := 0x40 + 0x80 + 0xC0
0000aff9: RBX.9 := 0x40 + 0x80 + 0xC0
0000b006: RBX.10 := 0x40 + 0x80 + 0xC0 & 0x3F
0000b037: ZF.12 := 0 = (0x40 + 0x80 + 0xC0 &
0x3F)
0000b040: when ~ZF.12 call @err with noreturn
0000b4db: goto %0000b046

0000b046:
0000b80e: local_40.1 := 0x40 + 0x80 + 0xC0
0000b052: RAX.10 := 0x140
0000b059: RBX.11 := 0x140
```

```
0000b066: RBX.12 := 0x140 & 0x3F
0000b097: ZF.14 := 0 = (0x140 & 0x3F)
0000b0a0: when ~ZF.14 call @err with noreturn
0000b4dc: goto %0000b0a6

0000b0a6:
0000b80f: local_48 := 0x140
0000b0b2: RAX.11 := 0x40
0000b0b9: RBX.13 := 0x40
0000b0c6: RBX.14 := 0x40 & 0x3F
0000b0f7: ZF.16 := 0 = (0x40 & 0x3F)
0000b100: when ~ZF.16 call @err with noreturn
0000b4dd: goto %0000b106

0000b106:
0000b119: RAX.12 := 0x40 + 0x140
0000b132: RBX.15 := 0x40 + 0x140
0000b13f: RBX.16 := 0x40 + 0x140 & 0x3F
0000b170: ZF.19 := 0 = (0x40 + 0x140 & 0x3F)
0000b179: when ~ZF.19 call @err with noreturn
0000b4de: goto %0000b17f

0000b17f:
0000b192: RAX.13 := 0x40 + 0x140 + 0x40 + 0x80
+ 0xC0
0000b1ad: #1883.1 := mem.8[RSP, el]:u64
0000b1b0: RSP.1 := RSP + 8
0000b1b4: call #1883.1 with noreturn
```

References:

Cifuentes, C. (1994, July). Reverse Compilation Techniques. Retrieved March 31, 2020, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.6990

Brumley, D., Jager, I., Avgerinos, T., & Schwartz, E. J. (2011, July). BAP: A binary analysis platform. In International Conference on Computer Aided Verification (pp. 463-469). Springer, Berlin, Heidelberg.
https://github.com/BinaryAnalysisPlatform/bap
https://github.com/BinaryAnalysisPlatform/bap-plugins

Goldstein , S. C. (2020, May).
http://www.cs.cmu.edu/afs/cs/academic/class/15745-s05/www/lectures/lect2.pdf

Appendix A

---

prop_nostd.rkt (objdump):

```
3e50 <entry>:
3e50:  mov  eax,0x40
3e55:  mov  QWORD PTR [rsp-0x8],rax
3e5a:  mov  eax,0x140
3e5f:  mov  QWORD PTR [rsp-0x10],rax
3e64:  mov  eax,0x80
3e69:  mov  rbx,rax
3e6c:  and  rbx,0x3f
3e70:  cmp  rbx,0x0
3e74:  jne  3f2a <err>
3e7a:  mov  QWORD PTR [rsp-0x18],rax
3e7f:  mov  eax,0x40
3e84:  mov  rbx,rax
3e87:  and  rbx,0x3f
3e8b:  cmp  rbx,0x0
3e8f:  jne  3f2a <err>
3e95:  add  rax,QWORD PTR [rsp-0x18]
3e9a:  mov  QWORD PTR [rsp-0x18],rax
3e9f:  mov  eax,0xc0
3ea4:  mov  QWORD PTR [rsp-0x20],rax
3ea9:  mov  rax,QWORD PTR [rsp-0x20]
3eae:  mov  rbx,rax
3eb1:  and  rbx,0x3f
3eb5:  cmp  rbx,0x0
3eb9:  jne  3f2a <err>
3ebb:  mov  QWORD PTR [rsp-0x28],rax
3ec0:  mov  rax,QWORD PTR [rsp-0x18]
3ec5:  mov  rbx,rax
3ec8:  and  rbx,0x3f
3ecc:  cmp  rbx,0x0
3ed0:  jne  3f2a <err>
3ed2:  add  rax,QWORD PTR [rsp-0x28]
3ed7:  mov  rbx,rax
3eda:  and  rbx,0x3f
3ede:  cmp  rbx,0x0
3ee2:  jne  3f2a <err>
3ee4:  mov  QWORD PTR [rsp-0x28],rax
3ee9:  mov  rax,QWORD PTR [rsp-0x10]
3eee:  mov  rbx,rax
```

```
3ef1:  and  rbx,0x3f
3ef5:  cmp  rbx,0x0
3ef9:  jne  3f2a <err>
3efb:  mov  QWORD PTR [rsp-0x30],rax
3f00:  mov  rax,QWORD PTR [rsp-0x8]
3f05:  mov  rbx,rax
3f08:  and  rbx,0x3f
3f0c:  cmp  rbx,0x0
3f10:  jne  3f2a <err>
3f12:  add  rax,QWORD PTR [rsp-0x30]
3f17:  mov  rbx,rax
3f1a:  and  rbx,0x3f
3f1e:  cmp  rbx,0x0
3f22:  jne  3f2a <err>
3f24:  add  rax,QWORD PTR [rsp-0x28]
3f29:  ret
```