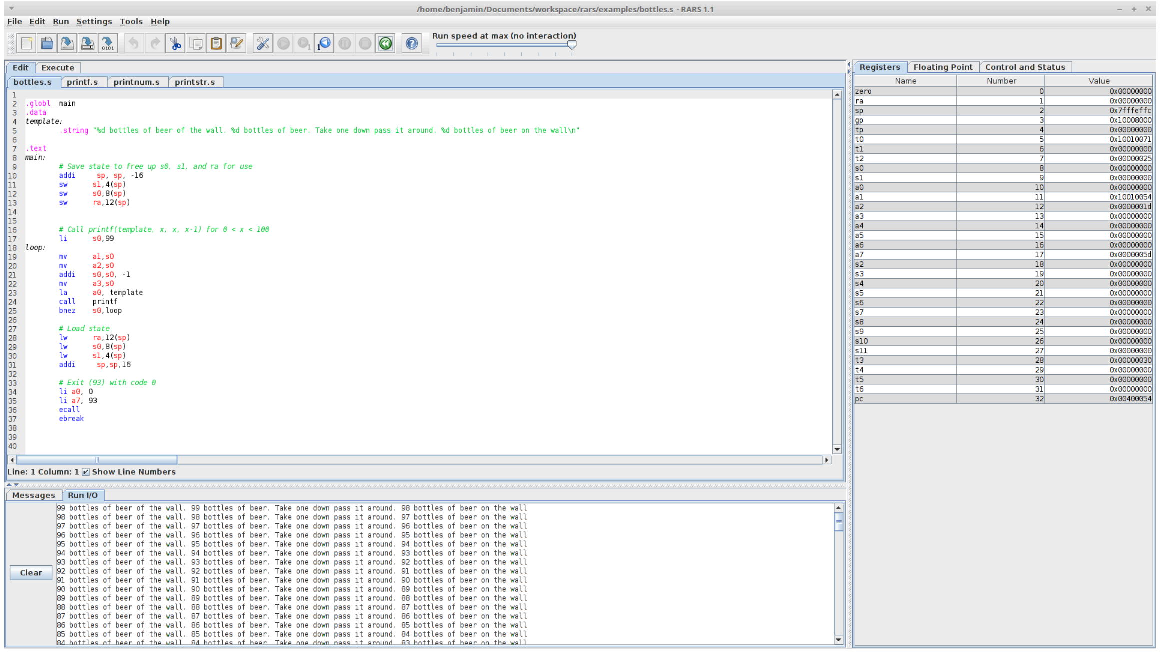
RISC-V Analyzer: verify assembly code compliance to register and procedure calling conventions.

Rajan Maghera, Nathan Ulmer, J. Nelson Amaral

University of Alberta

November 12, 2024

```
calculator:
li t4, -2
li t5, -3
l1:
lw t0, (a0)
beq t0, t5, term
addi a0, a0, 4
bge t0, zero, operand
lw t1, 8(a1)
lw t2, 4(a1)
addi a1, a1, 4
beq t0, t4, l2
add t3, t1, t2
j l3
l2:
```



RARS SIMULATOR FOR RISC-V

1.2k stars on GitHub

Basic assembler errors only

There is a wealth of dataflow analyses in the wild.

Can we use these to verify that our code conforms to conventions?

Let's see what we need to do.

LIVENESS ANALYSIS

```
LIVE_in[s] = GEN[s] ∪ (LIVE_out[s] – KILL[s])
LIVE_out[final] = {}

LIVE_out[s] = ∪ { LIVE_in[p] | p ∈ succ(s) }

GEN[s] = { x | x is used in s }
KILL[s] = { x | x is defined in s }
```

Determine the live range of variables.

A live variable contains a value that may be used in the future.

FIRST GOALS

- Ensure that caller-saved registers are not read from incorrectly
- Determine which registers may be argument/return registers

```
li a0, 12          # a0 <- 12
li a1, 23          # a1 <- 23
add a0, a0, a1     # a0 <- a0 + a1
```

PRE-WORK

- Lexing and parsing code
- Generating CFGs
- Data structures

PROBLEM 1

REGISTERS!

(are just like global variables?)

```
li a0, 12          # a0 <- 12
li a1, 23          # a1 <- 23
add a0, a0, a1     # a0 <- a0 + a1
```

```
a0 = 12          # a0 <- 12
a1 = 23          # a1 <- 23
a0 = a0 + a1     # a0 <- a0 + a1
```

```
# IN = {}
a0 = 12                 # a0 <- 12
# OUT = {a0}

# IN = {a0}
a1 = 23                 # a1 <- 23
# OUT = {a0, a1}

# IN = {a0, a1}
a0 = a0 + a1            # a0 <- a0 + a1
# OUT = {}
```

```
# IN = {a0, a1}
a0 = a0 + a1      # a0 <- a0 + a1
# Warning: a0 is not used
# OUT = {}
```

```
# IN = {a0, a1}
a0 = a0 + a1      # a0 <- a0 + a1
# OUT = {a0}

# IN = {a0}
print(a0)          # call print(a0)
# OUT = {}
```

```
# IN = {?}
li a0, 12          # a0 <- 12
# OUT = {a0, ?}

# IN = {a0, ?}
li a1, 23          # a1 <- 23
# OUT = {a0, a1, ?}

# IN = {a0, a1, ?}
add a0, a0, a1     # a0 <- a0 + a1
# OUT = {?}

# IN = {?}
jal ra, print       # call print(a0)
# OUT = {?}
```

PROBLEM 2

FUNCTIONS!

(oh boy, this is a big one)

SO, HOW CAN WE DEFINE A FUNCTION CALL?

In other words, what is generic about function calls?

RISC-V

REGISTER AND CALLING CONVENTIONS!

(our work is generic to other conventions and architectures)

```
# IN = {?}
li a0, 12           # a0 <- 12
# OUT = {a0, ?}

# IN = {a0, ?}
li a1, 23           # a1 <- 23
# OUT = {a0, a1, ?}

# IN = {a0, a1, ?}
add a0, a0, a1      # a0 <- a0 + a1
# OUT = {?}

# IN = {?}
jal ra, print        # call print(a0)
# OUT = {?}
```

A FUNCTION CALL

The `return_address` is set to `program_counter + 1 instruction`

Jump to instruction at `addr(function_name)`

```
# IN = {?}
li a0, 12          # a0 <- 12
# OUT = {a0, ?}

# IN = {a0, ?}
li a1, 23          # a1 <- 23
# OUT = {a0, a1, ?}

# IN = {a0, a1, ?}
add a0, a0, a1     # a0 <- a0 + a1
# OUT = {?}

# IN = {?}
jal ra, print       # ra <- pc + 4, jump to print
# OUT = {?}
```

A FUNCTION BODY

Arguments are read from `{s: s is an argument register}`

Return values are set to `{s: s is a return register}`

Jump to `return_addr` at the end

??? = print(???)

```
# IN = {?}
li a0, 12           # a0 <- 12
# OUT = {a0, ?}

# IN = {a0, ?}
li a1, 23           # a1 <- 23
# OUT = {a0, a1, ?}

# IN = {a0, a1, ?}
add a0, a0, a1      # a0 <- a0 + a1
# OUT = {?}

# IN = {?}
jal ra, print        # ra <- pc + 4 inst, jump to print
# OUT = {?}
```

??? = print(???)

```
# IN = {?}
li a0, 12           # a0 <- 12
# OUT = {a0, ?}

# IN = {a0, ?}
li a1, 23           # a1 <- 23
# OUT = {a0, a1, ?}

# IN = {a0, a1, ?}
add a0, a0, a1      # a0 <- a0 + a1
# OUT = {?}

# IN = {?}
jal ra, print        # ra <- pc + 4 inst, jump to print
# OUT = {?}
```

??? = print(???)

```
# IN = {s0, ?}
li a0, 12           # a0 <- 12
# OUT = {s0, a0, ?}

# IN = {s0, a0, ?}
li a1, 23           # a1 <- 23
# OUT = {s0, a0, a1, ?}

# IN = {s0, a0, a1, ?}
add a0, a0, a1      # a0 <- a0 + a1
# OUT = {s0, ?}

# IN = {s0, ?}
jal ra, print        # ra <- pc + 4 inst, jump to print
# OUT = {a0, a1, s0}
```

a0, a1 = print(???)

```
# IN = {s0, ?}
li a0, 12          # a0 <- 12
# OUT = {s0, a0, ?}

# IN = {s0, a0, ?}
li a1, 23          # a1 <- 23
# OUT = {s0, a0, a1, ?}

# IN = {s0, a0, a1, ?}
add a0, a0, a1     # a0 <- a0 + a1
# OUT = {s0, ?}

# IN = {s0, ?}
jal ra, print       # ra <- pc + 4 inst, jump to print
# OUT = {a0, a1, s0}
```

a0, a1 = print(a0)

```
print:

# IN = {a0}
# ... some code here
# OUT = {a0, a1}

# IN = {a0, a1}
ret                   # jump to ra
```

`a0, a1 = print(a0)`

```
# IN = {s0, ?}
li a0, 12          # a0 <- 12
# OUT = {s0, a0, ?}

# IN = {s0, a0, ?}
li a1, 23          # a1 <- 23
# OUT = {s0, a0, a1, ?}

# IN = {s0, a0, a1, ?}
add a0, a0, a1     # a0 <- a0 + a1
# OUT = {s0, ?}

# IN = {s0, ?}
jal ra, print       # ra <- pc + 4 inst, jump to print
# OUT = {a0, a1, s0}
```

a0, a1 = print(a0)

```
# IN = {s0}
li a0, 12          # a0 <- 12
# OUT = {s0, a0}

# IN = {s0, a0}
li a1, 23          # a1 <- 23
# OUT = {s0, a0, a1}

# IN = {s0, a0, a1}
add a0, a0, a1     # a0 <- a0 + a1
# OUT = {s0, a0}

# IN = {s0, a0}
jal ra, print       # ra <- pc + 4 inst, jump to print
# OUT = {a0, a1, s0}
```

ALGORITHM 1

LIVENESS ANALYSIS

with additional inter-procedural analysis

LIVENESS ANALYSIS

- Backward analysis
- Goal: determine the liveness ranges of register values
- Data-structure: bitset (32 registers)

OUR LIVENESS ANALYSIS

is an inter-procedural analysis.

Analysis is generic to register and calling conventions.

OUR LIVENESS ANALYSIS

We need to keep track of the unconditionally defined values in a function.

Use the IN set at function call sites as the return values

Use the IN set at function entry sites as the argument values

A FUNCTION BODY, REVISITED

- ~~Use argument registers for arguments~~
- ~~Use return registers for return values~~
- ~~Jump to return register at end~~
- Callee-saved registers must be restored before returning

NEW GOALS

- Ensure that callee-saved registers are restored to their original values

```
foo:
    jal ra, bar         # ra <- pc + 4, jump to bar
    add a0, a0, a1      # a0 <- a0 + a1
    ret                 # return
```

PROBLEM 3

RUNTIME VALUES!

(runtime values?)

STRATEGY

Keep track of the values in the registers as much as we can.

```
foo:
    # IN = {ra: ra_0}
    jal ra, bar          # ra <- pc + 4, jump to bar
    # OUT = {}

    # IN = {}
    add a0, a0, a1       # a0 <- a0 + a1
    # OUT = {}

    ret                  # return
```

```
foo:
    addi sp, sp, -4   # sp <- sp - 4
    sw ra, 0(sp)      # sp[0] <- ra

    # IN = {ra: ra_0}
    jal ra, bar       # ra <- pc + 4, jump to bar
    # OUT = {}

    # IN = {}
    add a0, a0, a1    # a0 <- a0 + a1
    # OUT = {}

    lw ra, 0(sp)      # ra <- sp[0]
    addi sp, sp, 4    # sp <- sp + 4
```

```
foo:
    # IN = {ra: ra_0, sp: sp_0}
    addi sp, sp, -4    # sp <- sp - 4
    # OUT = {ra: ra_0}

    # IN = {ra: ra_0}
    sw ra, 0(sp)       # sp[0] <- ra
    # OUT = {ra: ra_0}

    # IN = {ra: ra_0}
    jal ra, bar        # ra <- pc + 4, jump to bar
    # OUT = {}

    # IN = {}
    add a0, a0, a1     # a0 <- a0 + a1
```

```
foo:
    # IN = {ra: ra_0, sp: sp_0}
    addi sp, sp, -4   # sp <- sp - 4
    # OUT = {ra: ra_0, sp: sp_0 - 4}

    # IN = {ra: ra_0, sp: sp_0 - 4}
    sw ra, 0(sp)      # sp[0] <- ra
    # OUT = {ra: ra_0, sp: sp_0 - 4}

    # IN = {ra: ra_0, sp: sp_0 - 4}
    jal ra, bar       # ra <- pc + 4, jump to bar
    # OUT = {sp: sp_0 - 4}

    # IN = {sp: sp_0 - 4}
    add a0, a0, a1    # a0 <- a0 + a1
```

```
foo:
    # IN = {ra: ra_0, sp: sp_0}
    addi sp, sp, -4   # sp <- sp - 4
    # OUT = {ra: ra_0, sp: sp_0 - 4}

    # IN = {ra: ra_0, sp: sp_0 - 4}
    sw ra, 0(sp)      # sp[0] <- ra
    # OUT = {ra: ra_0, sp: sp_0 - 4, sp_0[-1]: ra_0}

    # IN = {ra: ra_0, sp: sp_0 - 4, sp_0[-1]: ra_0}
    jal ra, bar       # ra <- pc + 4, jump to bar
    # OUT = {sp: sp_0 - 4, sp_0[-1]: ra_0}

    # IN = {sp: sp_0 - 4, sp_0[-1]: ra_0}
    add a0, a0, a1    # a0 <- a0 + a1
```

ALGORITHM 2

AVAILABLE VALUE ANALYSIS

A limited form of abstract interpretation

AVAILABLE VALUE ANALYSIS

- Forward analysis
- Goal: determine the value in registers and certain memory locations at every point in the program
- Data-structure: hash-map, "location" to value

AVAILABLE VALUE ANALYSIS

REPRESENTED TYPES

- Constant
- Register value at entrypoint (+ scalar offset)
- Values in (some) memory

AVAILABLE VALUE ANALYSIS

We use the dataflow algorithm as a base and can compute/optimize values (constant folding).

Special care needed for the "universe" set of our hash-map.

We combine these two analyses to get...

RISC-V ANALYZER

APPLICATION 1

CODE DIAGNOSTICS

```
main:
    li a0, 5
    li a1, 3
    li t0, 2
    jal ra, bad_fib
    li a7, 10
    ecall
fib_inc:
    add t0, a0, a1
    add t0, a0, a1
    mv a1, a0
    mv a0, t0
    ret
bad_fib:
    addi sp, sp, -2
```

```
 1 main:
 2     li a0, 5
 3     li a1, 3
 4     li t0, 2
 5     jal ra, bad_fib
 6     li a7, 10
 7     ecall
 8 fib_inc:
 9     add t0, a0, a1
10     add t0, a0, a1
11     mv a1, a0
12     mv a0, t0
13     ret
14 bad_fib:
15     addi sp, sp, -2
```

```
line 9: unused value in t0
line 15: invalid stack pointer (-2)
line 16: t0 used before set
line 18: unreachable code
line 22: ra not restored
line 23: t1 overwritten by function call on line 22
line 25: t1 overwritten by function call on line 22
```

APPLICATION 2

CODE FIXES

```
sum:
  mv s0, a0           # s0 ← a0
  li t0, -1           # t1 ← -1
  addi a0, a0, -1     # a0 ← a0 - 1
  jal ra, sum         # sum(a0)
  add a0, s0, a0      # a0 ← s0 + a0
  ret                 # return a0
```

Dead-code elimination

```
sum:
  mv s0, a0            # s0 ← a0
  addi a0, a0, -1      # a0 ← a0 - 1
  jal ra, sum          # sum(a0)
  add a0, s0, a0       # a0 ← s0 + a0
  ret                  # return a0
```

Storing callee-saved registers to stack

```
sum:
  addi sp, sp, -8     # sp ← sp - 8
  sw ra, 0(sp)        # sp[0] ← ra
  sw s0, 4(sp)        # sp[1] ← s0
  mv s0, a0           # s0 ← a0
  addi a0, a0, -1     # a0 ← a0 - 1
  jal ra, sum         # sum(a0)
  add a0, s0, a0      # a0 ← s0 + a0
  lw ra, 0(sp)        # ra ← sp[0]
  lw s0, 4(sp)        # s0 ← sp[1]
  addi sp, sp, 8      # sp ← sp + 8
  ret                 # return a0
```

Storing callee-saved registers to stack

On a corpus of 995 code samples (~100 lines of code each) from novice programmers, we found

- 1032 uses before assignments
- 337 unused values
- 1335 callee-saved register overwrites
- 496 lost callee-saved register values

TAKEAWAYS

Solutions are quite elegent*

We have to choose a subset of programs to consider "valid".

TAKEAWAYS

These analyses are fully static, but we can still pull a lot of information from the code.

We require the whole program's source code. We don't have to worry about hidden values.

< 1 second per code sample to run

Implemented in Rust as a CLI tool.

Available as a Language Server Protocol (LSP) compatible tool.

Also available as a VSCode extension.

THANK YOU!

https://github.com/rajanmaghera/riscv-analysis

rmaghera@ualberta.ca