Live Variable Analysis

EECE 5183

Marshal Stewart

CE Undergrad

Goals

- Define Liveliness
- Utilize Liveliness to optimize our programs
- Discuss optimization methods that utilize liveliness

Static Single-Assignment Form (SSA)

- Property of an intermediate representation (IR) that requires each variable to be assigned exactly once and defined before it is used. (LLVM)
- **Each variable is independent, no side effects on other code sections**

SSA > C main.c

```c
int main() {
    int x = 5;
    int y = 7;
    int z = x + y;
    x = 10;
    y = 15;
    z = x + y;
    return 0;
}

```

SSA > $ commands.sh

```
clang -S -emit-llvm main.c
```

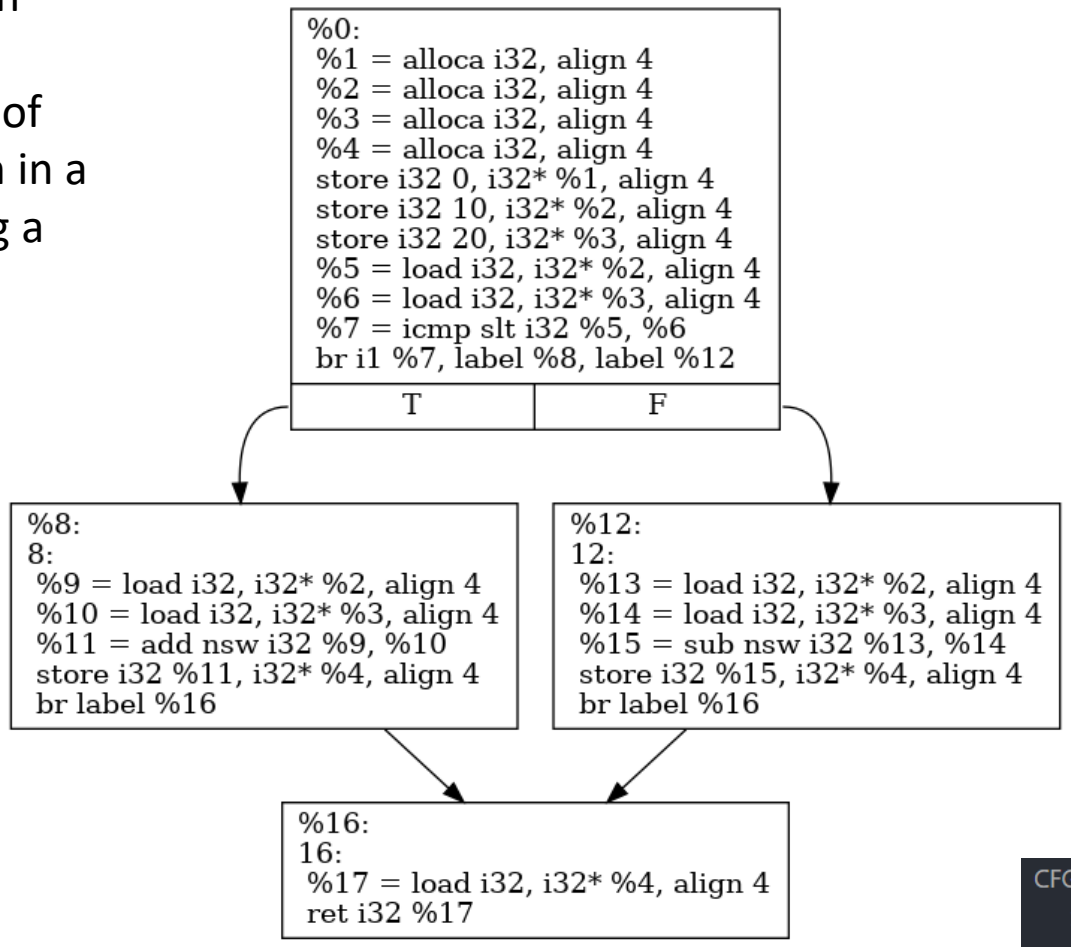
```
define dso_local i32 @main() #0 {
  ; Define variable memory once, all others are references
  ; Note: %1 is unused, it represents the function return type
  %1 = alloca i32, align 4          ; return value
  %2 = alloca i32, align 4          ; int x
  %3 = alloca i32, align 4          ; int y
  %4 = alloca i32, align 4          ; int z

  store i32 0, i32* %1, align 4     ; return value
  store i32 5, i32* %2, align 4     ; x = 5
  store i32 7, i32* %3, align 4     ; y = 7

  ; Create new variables for x and y reads (SSA)
  %5 = load i32, i32* %2, align 4   ; x_1 = x
  %6 = load i32, i32* %3, align 4   ; y_1 = y

  %7 = add nsw i32 %5, %6           ; (x_1 + y_1)
  store i32 %7, i32* %4, align 4    ; z = (x_1 + y_1)

  ; Assign original, not x_1 and y_1
  store i32 10, i32* %2, align 4    ; x = 10, line 5
  store i32 15, i32* %3, align 4    ; y = 15, line 6

  ; Create new variables again for x and y reads (SSA)
  %8 = load i32, i32* %2, align 4   ; x_2 = x
  %9 = load i32, i32* %3, align 4   ; y_2 = y

  %10 = add nsw i32 %8, %9          ; (x_2 + y_2)
  store i32 %10, i32* %4, align 4   ; z = (x_2 + y_2)

  ret i32 0                         ; return 0
}
```

Control Flow Graphs

Representation of sets of statements in a graph form, such that an edge is a branch of execution taken, a node is a set of instructions. The last instruction in a set before an edge usually being a condition check.

CFG > C example.c

```c
int main() {
    int x = 10;
    int y = 20;
    int z;

    if (x < y) {
        z = x + y;
    } else {
        z = x - y;
    }

    return z;
}
```



CFG for 'main' function

%1 => z
%2 => x
%3 => y
%4 => return value
%5 => x_1 in cond on line 6
%6 => y_1 in cond on line 6
%7 => (x_1 < y_1) output
%8 => stores label for (x<y)
    %9 => stores x_2 on line 7
    %10 => stores y_2 on line 7
    %11 => z_1, line 7
%12 => stores label for (else)
    %13 => stores x_3, line 9
    %14 => stores y_3, line 9
    %15 => stores z_2, line 9
%16 => stores label, after cond
%17 => stores return value

CFG > $ commands.sh

```
clang -S -emit-llvm example.c
opt -dot-cfg example.ll -o example.opt.ll
dot -Tpng .main.dot -o example.png
```

Def-Use and Use-Def Chains

- Represents the flow of information from the definition of the variable itself to its use.
- Variable Dependencies
- Def -> Write, Use -> Read

def_use_chains > C use_def.c

```c
// use-def chain for 'y' starts with use of 'x+3', ends with def of 'y'
int main() {
    int x = 5;
    // 'y' is defined, when 'x' is used in expression 'x+3'
    int y = x + 3;
    return 0;
}

```

def_use_chains > C def_use.c

```c
// def-use chain for 'x' starts with definition, ends with use at 'x + 3'
int main() {
    // 'x' is defined when it's initialized to 5
    int x = 5;
    int y;
    // 'x' is used in the expression 'x+3'
    y = x + 3;
    return 0;
}

```

Liveliness Algorithm Terms

- Liveliness Analysis
  - Group of techniques used for optimization, by determining life-times of variables
- Live Variable
  - A variable that at any given instant of time, is begin used to process a computation through evaluation, or hold a value that will be used in the future without re-definition
- Live Range
  - Defines continuous and/or discrete portions of code for which a variable is live. (Variables can move between the live and dead state
- Live
  - A variable ‘v’ is live on edge ‘e’ if there exists a directed path from the edge ‘e’ to use of ‘v’ that does not pass through any def(v)
- Live-In
  - A variable ‘v’ is live-in at node ‘n’ if the variable is live on any n’s in-edges
- Live-Out
  - A variable ‘v’ is live-out at a node ‘n’ if live on any of n’s out-edges

Liveliness Algorithm

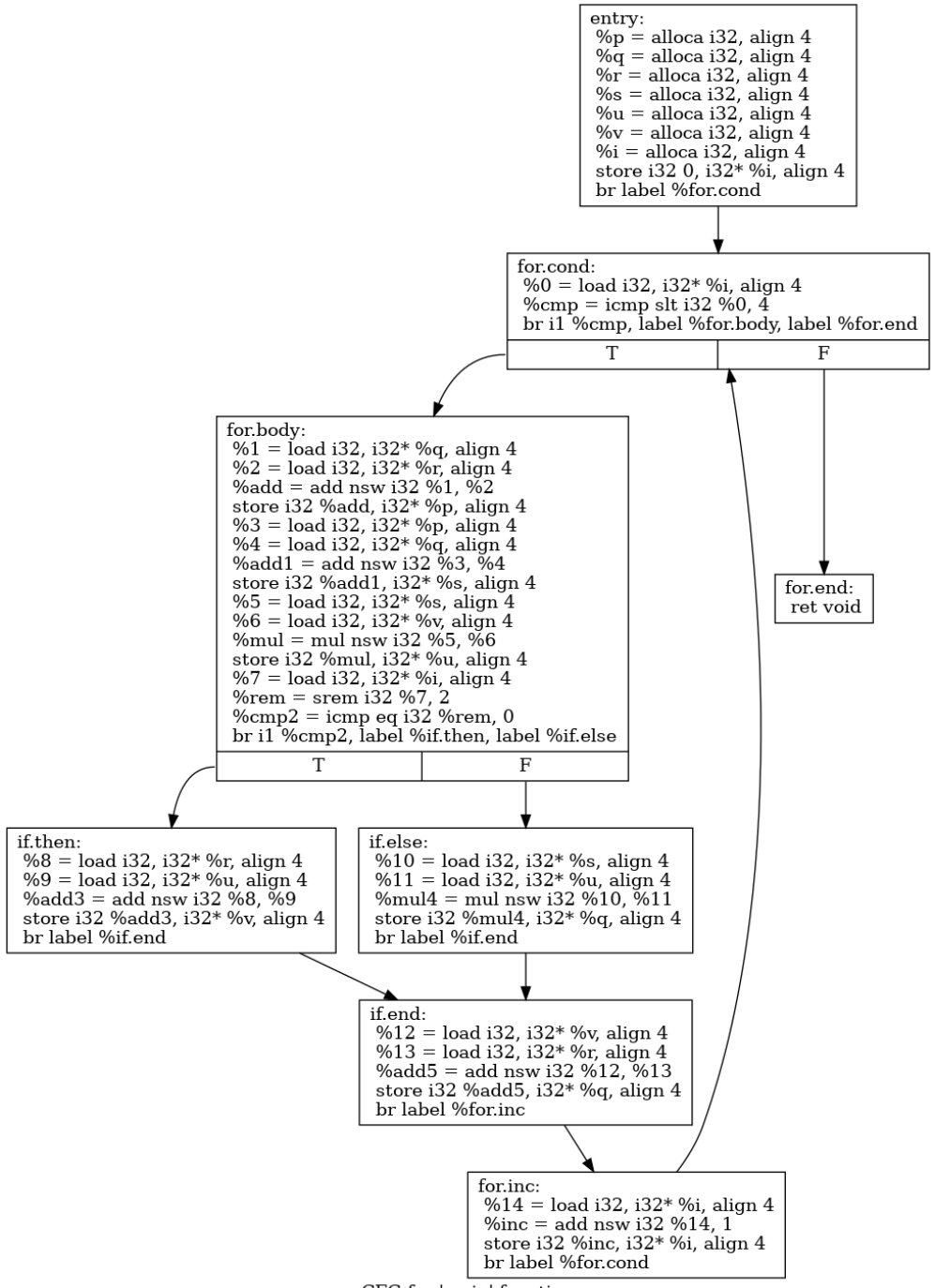
Evaluates the liveliness of each variable at each step. Analyzes the live ranges with the goal of sharing common registers that don’t overlap liveliness.

- Step 1 (Executed only once)
  - Identify defined variables, and which are used in **each basic block**. (def-use chains)
  - Initialize IN and OUT to null
- Step 2
  - Maintain global information records (transmission of live values). Compute IN and OUT sets from def and use sets by using the expression (utilize def-use chains)
- Step 3
  - Iterate step 2, until IN and OUT sets become constant for successive iterations. Def and use sets are constants, therefore path independent.

Step 1

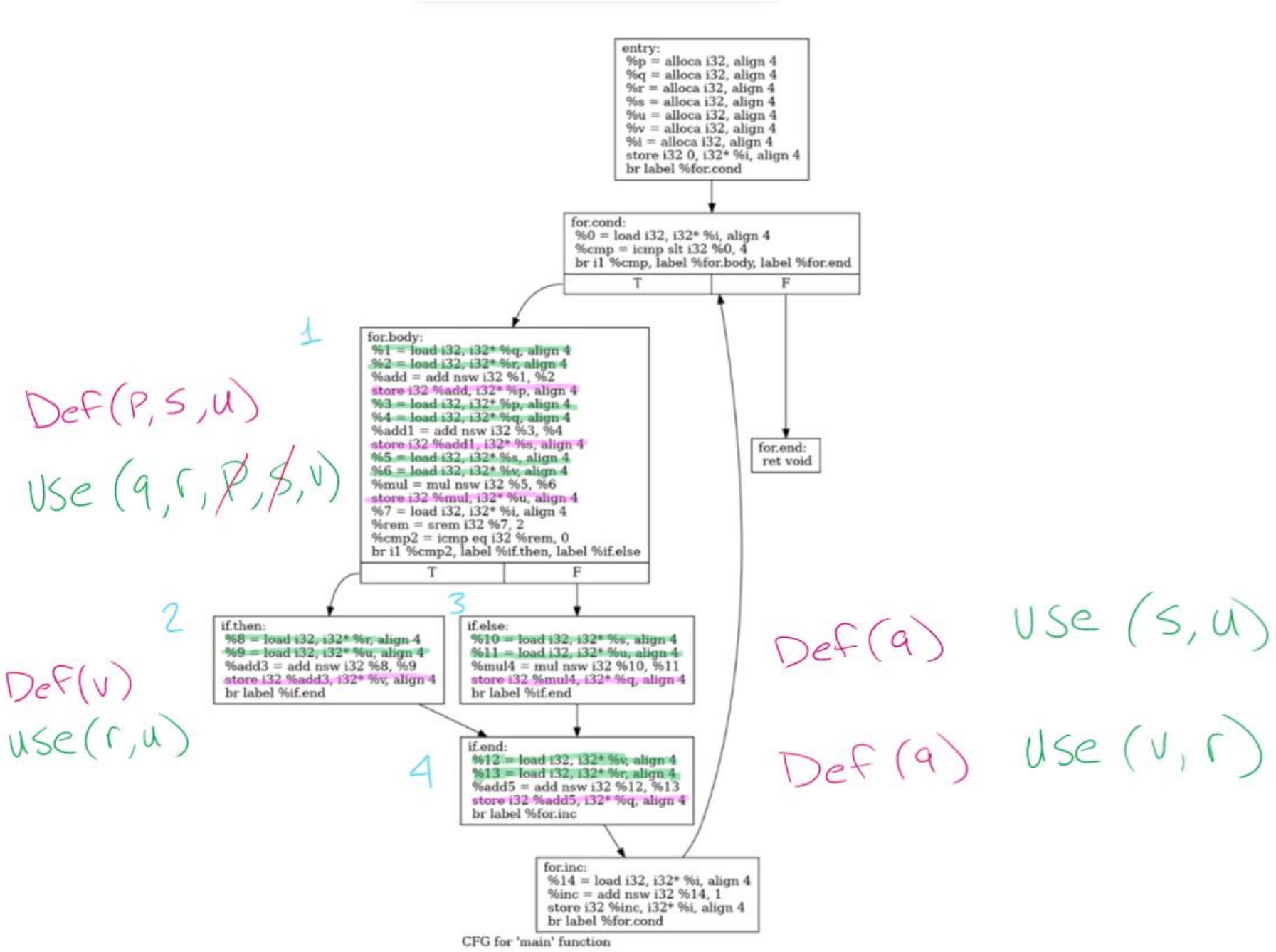
example > C example.c

```c
void main()
{
    int p, q, r, s, u, v;

    for (int i=0; i<4; i++)
    {
        // Node 1
        p = q + r;
        s = p + q;
        u = s * v;

        if ((i % 2) == 0) // Node 2
        {
            v = r + u;
        }
        else // Node 3
        {
            q = s * u;
        }

        // Node 4
        q = v + r;
    }

}
```



CFG for 'main' function

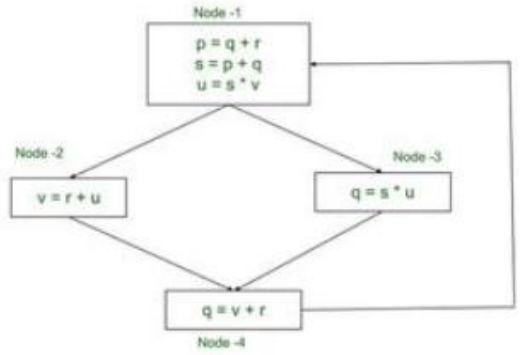
Step 1



CFG for 'main' function

Example + Proof

NODE(n)	use[n]	def[n]	Initial Value		1st Iteration		2nd Iteration		3rd Iteration	
			OUT_1	IN_1	OUT_2	IN_2	OUT_3	IN_3	OUT_4	IN_4
4	v,r	q	Ø	Ø	Ø	r,v	q,r,v	r,v	q,r,v	r,v
3	s,u	q	Ø	Ø	v,r	s,u,r,v	v,r	s,u,v,r	v,r	s,u,v,r
2	r,u	v	Ø	Ø	v,r	r,u	v,r	r,u	v,r	r,u
1	q,r,v	p,s,u	Ø	Ø	s,u,r,v	q,r,v	s,u,r,v	q,r,v	s,u,r,v	q,r,v



Control Flow Graph (Liveliness Analysis Example)

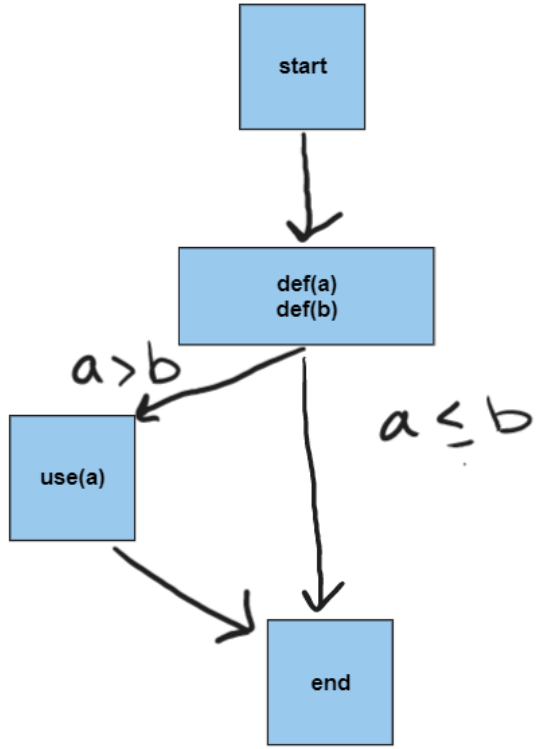
```
N : Set of nodes of CFG;
for each n ∈ N do
    in[n] ← ϕ;
    out[n] ← ϕ;
end
repeat
      for each n ∈ Nodes do
          // First save the current value for IN and OUT for comparison later.
        in'[n] ← in[n];
        out'[n] ← out[n];
        // For OUT, find out the union of previous variables
            in the IN set for each succeeding node of n.
        out[n] ← U_{s∈succ[n]} in[s] ; // Compute OUT for a node.
        in[n] ← use[n]U(out[n]-def [n]); // Compute IN for a node.
    end
// Iterate, until IN and OUT set are constants for last two consecutive iterations.
until ∀n,in'[n] = in[n] ∧ out' [n] = out[n];
```

Dead Code elimination

Removing code that has no effect on the program's output or behavior.

deadcode > C main.c

```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;

    // dead code
    if (a > b) {
        printf("The value of a is %d\n", a);
    }

    return 0;
}
```



```
8   ; Function Attrs: noinline nounwind optnone uwtable
9   define dso_local i32 @main() #0 {
10  entry:
11    %retval = alloca i32, align 4
12    %a = alloca i32, align 4
13    %b = alloca i32, align 4
14    store i32 0, i32* %retval, align 4
15    store i32 5, i32* %a, align 4
16    store i32 10, i32* %b, align 4
17    %0 = load i32, i32* %a, align 4
18    %1 = load i32, i32* %b, align 4
19    %cmp = icmp sgt i32 %0, %1
20    br i1 %cmp, label %if.then, label %if.end
21
22  if.then:                                          ; preds = %entry
23    %2 = load i32, i32* %a, align 4
24    %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([22 x i8], [22 x i8]* @.str, i64 0, i64 0), i32 %2)
25    br label %if.end
26
27  if.end:                                           ; preds = %if.then, %entry
28    ret i32 0
29  }
```

no opt

```
6   ; Function Attrs: norecurse nounwind readnone uwtable
7   define dso_local i32 @main() local_unnamed_addr #0 !dbg !6 {
8   entry:
9     ret i32 0, !dbg !8
10  }
```

opt

deadcode > $ commands.sh

```
1  #!/usr/bin/env bash
2  clang -S -O0 -emit-llvm -fno-discard-value-names -march=x86-64 -o output.ll main.c
3  clang -S -O3 -emit-llvm -fno-discard-value-names -march=x86-64 -Rpass=deadcode -S -o output_no_dead.ll main.c
```

deadcode > C main.c

```c
#include <stdio.h>

void main() {
    int a;
    scanf("%d", &a);
    int b = 10;

    // dead code
    if (a > b) {
        printf("The value of a is %d\n", a);
    }

}

```

NO OPT

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @main() #0 {
entry:
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %call = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]
  store i32 10, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %cmp = icmp sgt i32 %0, %1
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %2 = load i32, i32* %a, align 4
  %call1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([22 x i8], [22 x i8]* @.s
  br label %if.end

if.end:                                           ; preds = %if.then, %entry
  ret void
}
```

OPT

```
; Function Attrs: nounwind uwtable
define dso_local void @main() local_unnamed_addr #0 !dbg !6 {
entry:
  %a = alloca i32, align 4
  %0 = bitcast i32* %a to i8*, !dbg !8
  call void @llvm.lifetime.start.p0i8(i64 4, i8* nonnull %0) #3, !dbg !8
  %call = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* nonnull %a),
  %1 = load i32, i32* %a, align 4, !dbg !10, !tbaa !11
  %cmp = icmp sgt i32 %1, 10, !dbg !15
  br i1 %cmp, label %if.then, label %if.end, !dbg !10

if.then:                                          ; preds = %entry
  %call1 = call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([22 x i8], [22 x i8]* @.str.1, i64 0, i
  br label %if.end, !dbg !17

if.end:                                           ; preds = %if.then, %entry
  call void @llvm.lifetime.end.p0i8(i64 4, i8* nonnull %0) #3, !dbg !18
  ret void, !dbg !18
}
```

no load

Common Subexpression Elimination

Identifying expressions that are repeated and storing the result in a temporary variable

common_expr > C main.c

```c
#include <stdio.h>

int foo(int x, int y, int z) {
    int a = x * y;
    int b = a + z;
    int c = x * y;
    int d = c + z;
    return b + d;
}

void main()
{
    int x=0,y=0,z=0;
    scanf("%d\n%d\n%d\n", &x,&y,&z);

    printf("answer=%d\n", foo(x,y,z));
}
```

```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @foo(i32 %x, i32 %y, i32 %z) local_unnamed_addr #0 {
entry:
  %mul = mul nsw i32 %y, %x
  %add = add nsw i32 %mul, %z
  %add3 = shl nsw i32 %add, 1
  ret i32 %add3
}
```

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @foo(i32 %x, i32 %y, i32 %z) #0 {
entry:
  %x.addr = alloca i32, align 4
  %y.addr = alloca i32, align 4
  %z.addr = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 %x, i32* %x.addr, align 4
  store i32 %y, i32* %y.addr, align 4
  store i32 %z, i32* %z.addr, align 4
  %0 = load i32, i32* %x.addr, align 4
  %1 = load i32, i32* %y.addr, align 4
  %mul = mul nsw i32 %0, %1
  store i32 %mul, i32* %a, align 4
  %2 = load i32, i32* %a, align 4
  %3 = load i32, i32* %z.addr, align 4
  %add = add nsw i32 %2, %3
  store i32 %add, i32* %b, align 4
  %4 = load i32, i32* %x.addr, align 4
  %5 = load i32, i32* %y.addr, align 4
  %mul1 = mul nsw i32 %4, %5
  store i32 %mul1, i32* %c, align 4
  %6 = load i32, i32* %c, align 4
  %7 = load i32, i32* %z.addr, align 4
  %add2 = add nsw i32 %6, %7
  store i32 %add2, i32* %d, align 4
  %8 = load i32, i32* %b, align 4
  %9 = load i32, i32* %d, align 4
  %add3 = add nsw i32 %8, %9
  ret i32 %add3
}
```

Register Allocation Optimization

- Allocating registers to hold intermediate results,
  - Reduce the number of memory accesses
  - Minimizing the amount of data that needs to be loaded and stored in memory.
  - Unoptimized: 13 instructions, use of r9d prevents potential for pipelining
    - Wrote to registers, back to memory, back to register for printf
  - Optimized: 6 instructions

register_opt > C example.c

```c
#include <stdio.h>

void main()
{
    int a=1, b=2, c=3, d=4;
    int x=5, y=6, z=7;

    scanf("%d\n,%d\n,%d\n,%d\n",&a,&b,&c,&d);

    x = a + b;
    y = b + c;
    z = c + d;

    printf("x:%d,y:%d,z:%d\n", x,y,z);

}
```

x86_64 asm

```
====================================

    movl    -4(%rbp), %r9d      # load a into r9d
    addl    -8(%rbp), %r9d      # store a + b into r9d
    movl    %r9d, -20(%rbp)     # store a + b at x

    movl    -8(%rbp), %r9d      # load b into r9d
    addl    -12(%rbp), %r9d     # store b + c into r9d
    movl    %r9d, -24(%rbp)     # store b + c at y

    movl    -12(%rbp), %r9d     # load c into r9d
    addl    -16(%rbp), %r9d     # load d into r9d
    movl    %r9d, -28(%rbp)     # store c + d at z

    # printf register parameters
    movl    -20(%rbp), %esi
    movl    -24(%rbp), %edx
    movl    -28(%rbp), %ecx

====================================
```

```
====================================

    movl    16(%rsp), %edx      # load b into edx
    movl    20(%rsp), %esi      # load a into esi
    addl    %edx, %esi          # store b + a into esi

    movl    12(%rsp), %ecx      # load c into ecx
    addl    %ecx, %edx          # => store c + b into edx

    addl    8(%rsp), %ecx       # store c + d into ecx

====================================
```

Loop Optimization

- Loop Unrolling
  - Executing multiple loop iterations for a single iterator value
- Loop-Invariant Code Motion
  - Moves computations that are not dependent on the loop iteration outside the loop
- Loop Fusion
  - Combining two or more loops that have similar dependencies, access patterns, and bounds into a single loop

References

- Github: MarshalStewart, Repo: EECE5183_presentation
  - https://github.com/MarshalStewart/EECE5183_presentation
- GeeksForGeeks
- X86_instruction_listings
- https://clang.llvm.org/docs/
- https://llvm.org/docs/LangRef.html