COMP 520 - Compilers

Lecture 16 – Code/Data Path Analysis

Reminders

- Midterm 2 on next Thursday, 4/11
- WA3 due tonight

Announcements

- PA4- Make(`Reg ridx, int mult, int disp, Reg r`)
  - This is probably the most difficult Make method
  - Two ways around this:
    - Don’t use [ridx*mult+disp] in your CodeGenerator
    - Solve the mystery in the ModRM+SIB table

- WA3
  - `rep stosq`- “REP” is a prefix that repeats the subsequent instruction “STOSQ”. The documentation for REP will tell you what the end condition is. Assume DF=0 or 1, either is fine.

Announcements (2)

PA4- Clarifications

- Callee should clean the stack
  - See ret imm16, where imm16 bytes are removed after returning.

- If you want your own username on the test server, make a private Piazza post, and Eric or I will adduser you.
  - Don't need to do this, use the generic comp520 login otherwise.
  - You do not get sudo permissions though, for the sanity of everyone involved.

Announcements (3)

PA4- Clarifications

- You are given mmap, which allocates a 4kb chunk.
- Remember, your PA4 goal is to make the code work before you optimize, so just make everything a 4kb allocation even if the size isn't 4kb.

- Can change this later in PA5's optional extra credit.

Announcements (4)

PA4- Clarifications

- Do not allocate objects on the stack.
- Some tests check for this where too many objects on the stack will crash the program.

- Lastly, you need to find out how to do sys_write. Use the given sys_mmap example.
  - Enables System.out.println(int n);

Compiler Optimization

Dataflow Analysis

Code Analysis

Data Liveness

Expr Liveness

Register Minimalization

Multiple CodePath Generation

TODAY

Available Expressions / Expression Lifetime Analysis

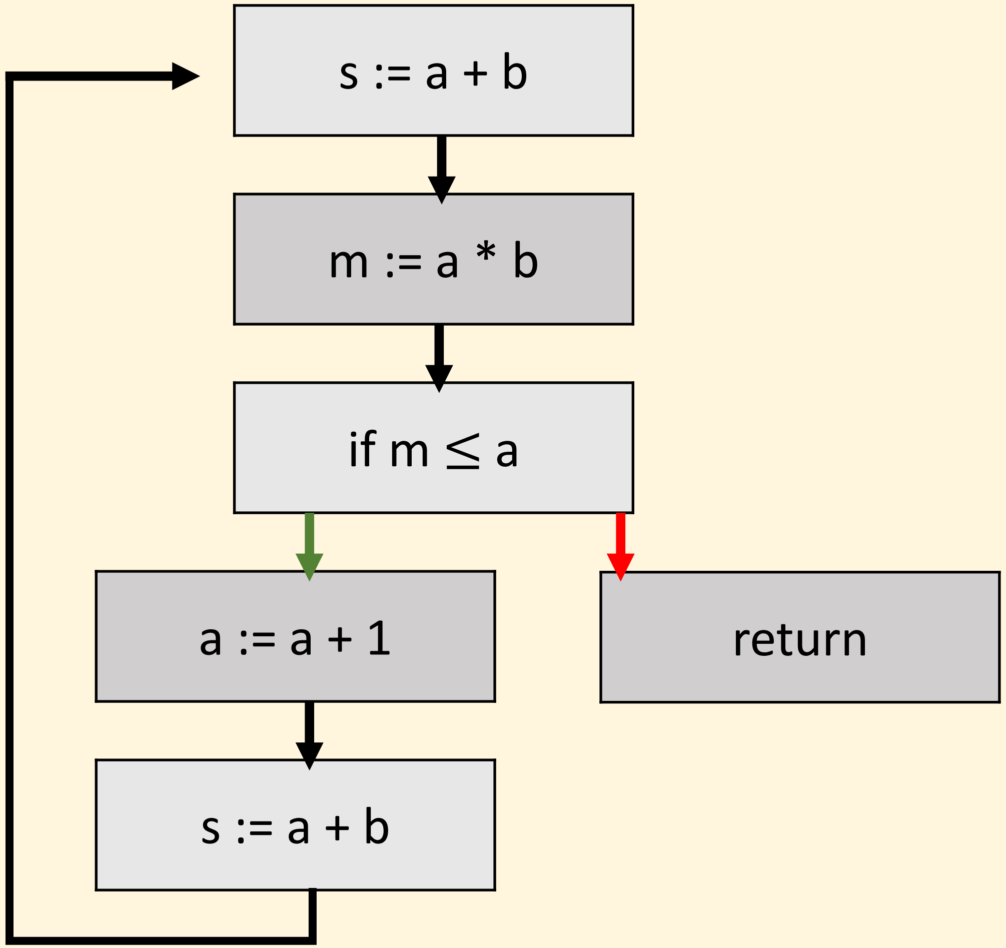
Can also apply lifetime analysis to expressions, not just variables.

Consider the following code:

```
s := a + b;
m := a * b;
while( m > a ) {
    a := a + 1;
    s := a + b;
}
```

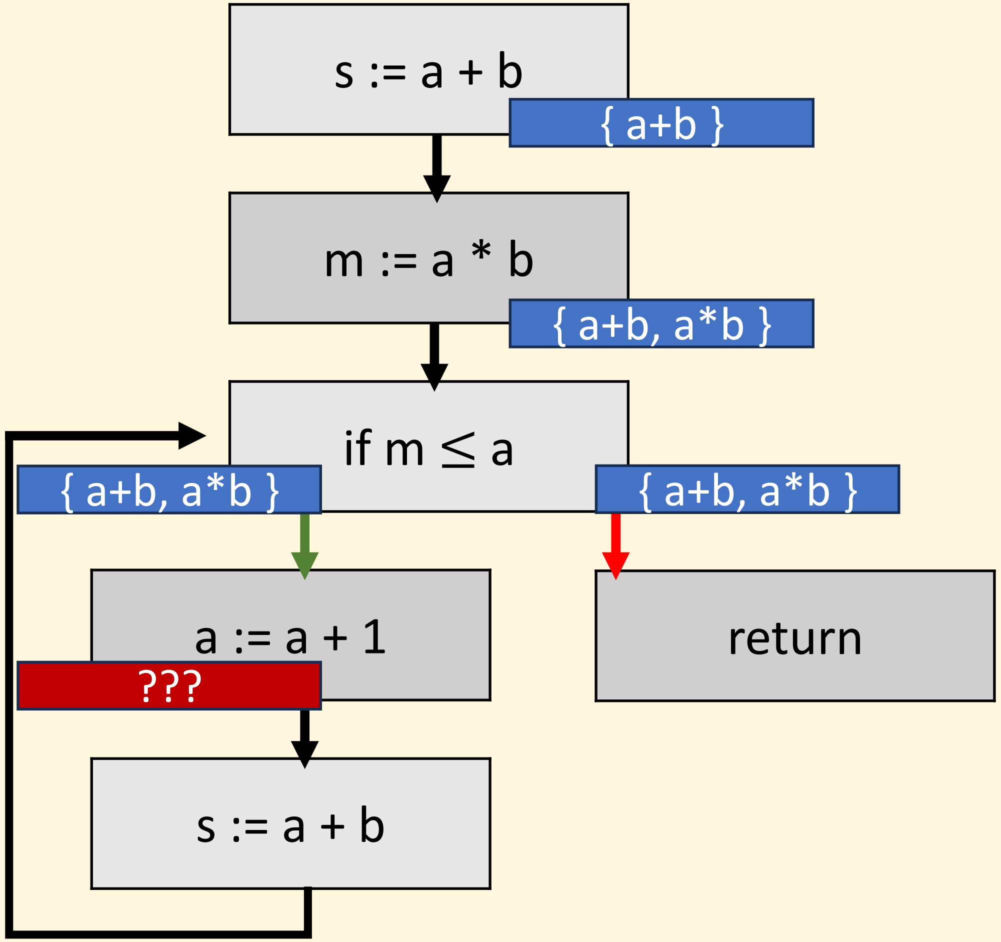
Construct the CFG

```
s := a + b;
m := a * b;
while( m > a ) {
    a := a + 1;
    s := a + b;
}
```



When data is *invalidated*, so are all expressions utilizing that data.

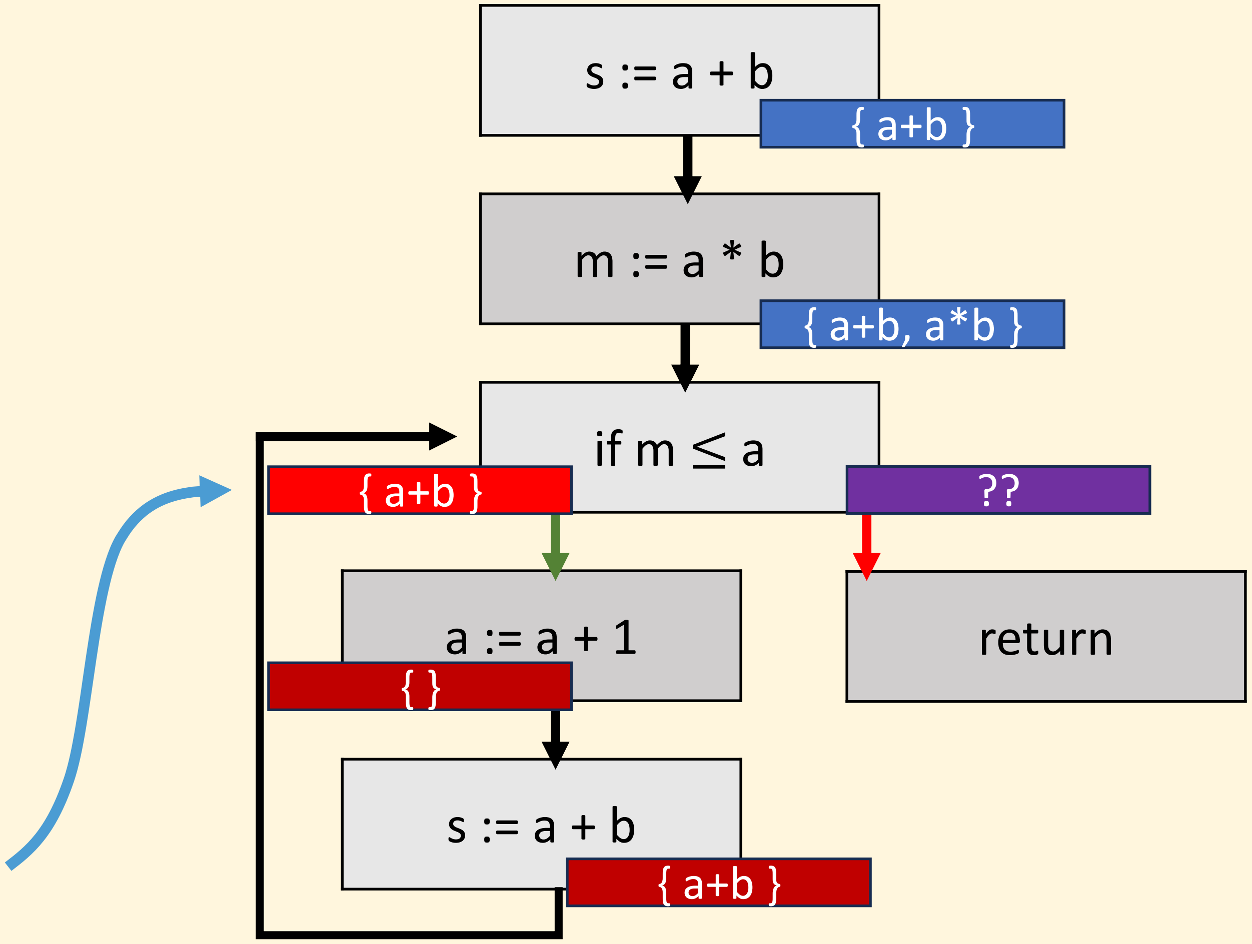
```
s := a + b;
m := a * b;
while( m > a ) {
    a := a + 1;
    s := a + b;
}
```



When data is *invalidated*, so are all expressions utilizing that data.

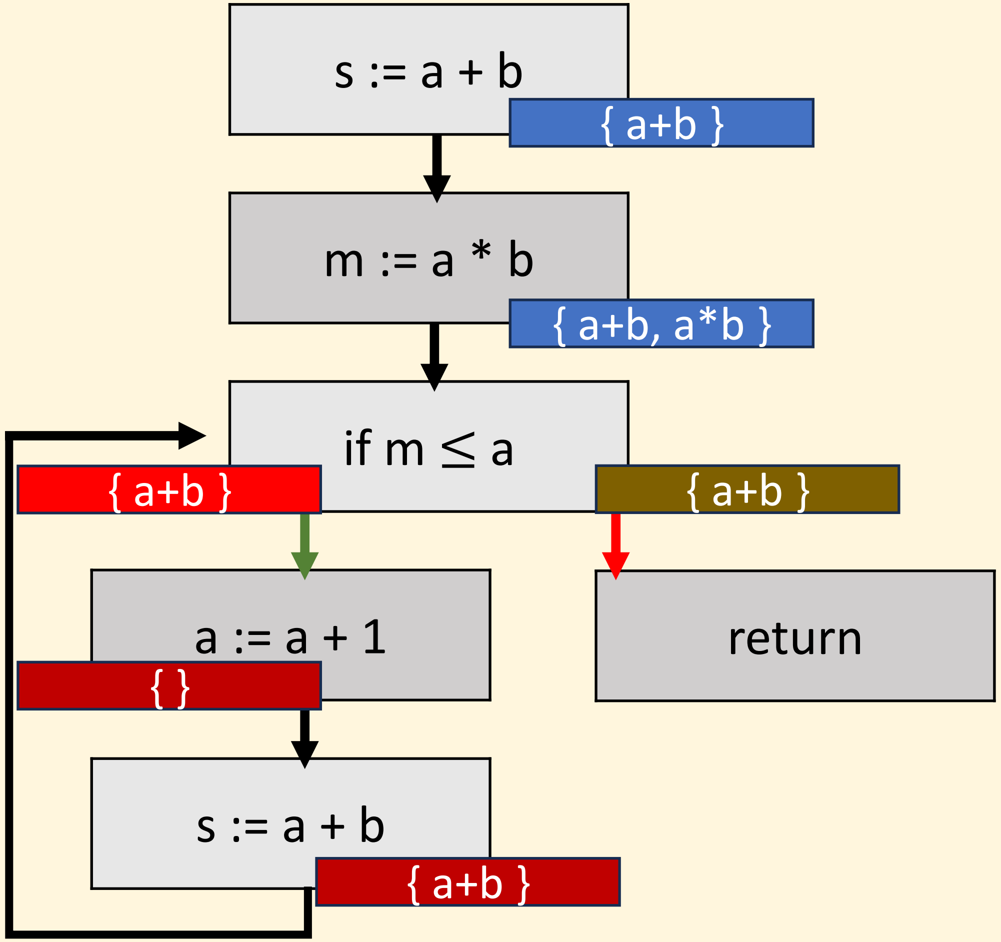
```
s := a + b;
m := a * b;
while( m > a ) {
    a := a + 1;
    s := a + b;
}
```

Note: we lost a*b here:



When data is *invalidated*, so are all expressions utilizing that data.

```
s := a + b;
m := a * b;
while( m > a ) {
    a := a + 1;
    s := a + b;
}
```

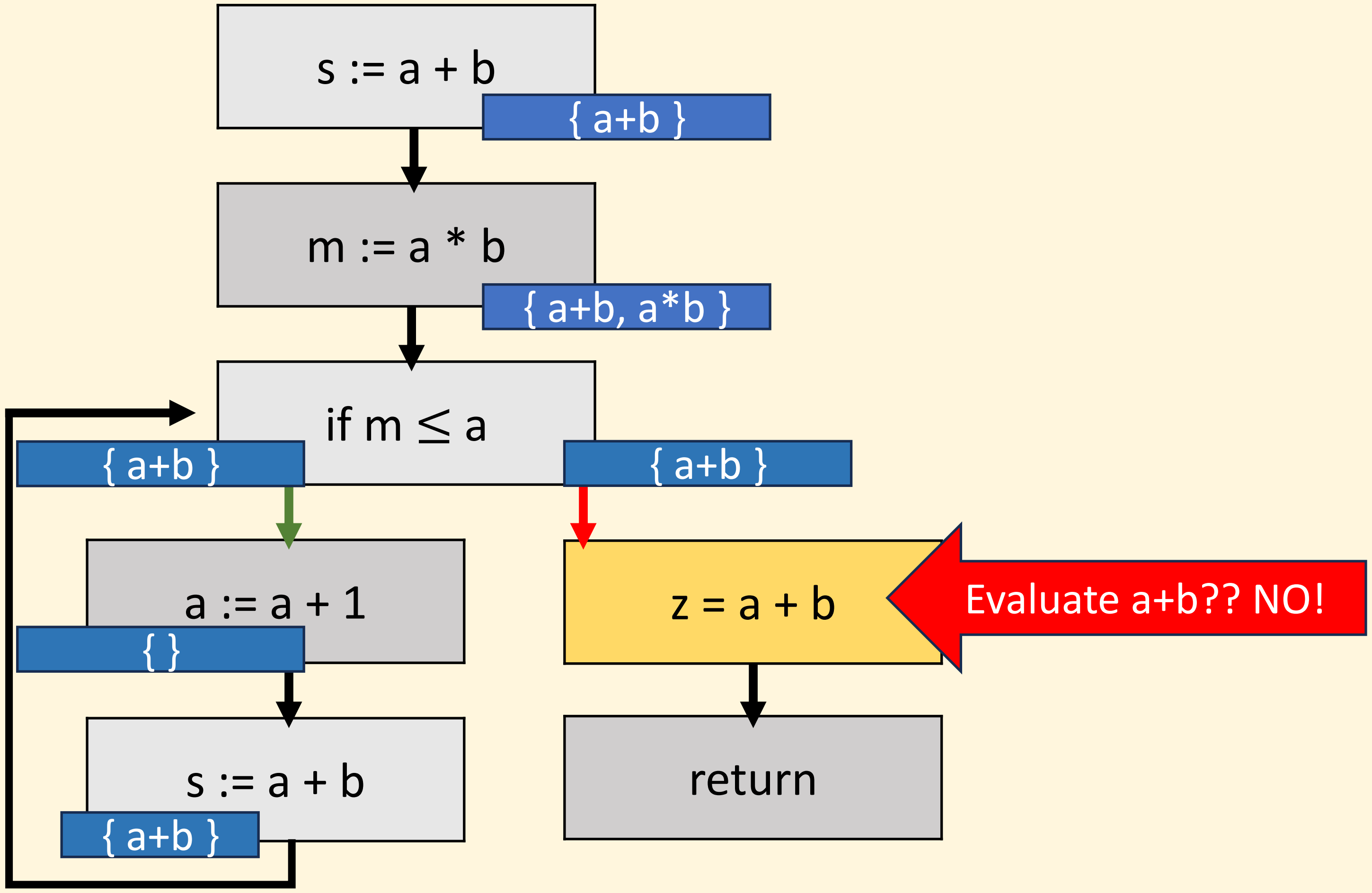


Expression Liveness

- Very useful so that an expression does not have to be re-evaluated.
- Let's look at that example earlier with one minor modification.

No need to re-evaluate a+b, because s is an alias.

```
s := a + b;
m := a * b;
while( m > a ) {
    a := a + 1;
    s := a + b;
}
z := a + b;
```



Formal Description: Expression Liveness

- Each vertex generates some "facts"
- Each vertex invalidates some "facts"
- Expression Liveness:
  - $\mathrm{gen}_{\mathbf{e}}(v) =$ expressions evaluated
  - $\mathrm{kill}_{\mathbf{e}}(v) =$ all expressions that contain $\mathrm{def}(v)$
  - $\mathrm{in}_{\mathbf{e}}(v) = \bigcap_{p \in \mathrm{predecessor}(v)} \mathrm{out}_{\mathbf{e}}(p)$
  - $\mathrm{out}_{\mathbf{e}}(v) = \mathrm{gen}_{\mathbf{e}}(v) \cup \left(\mathrm{in}_{\mathbf{e}}(v) \setminus \mathrm{kill}_{\mathbf{e}}(v)\right)$

Another Description: Data Liveness

- Each vertex generates some "facts"
- Each vertex invalidates some "facts"
- Data Liveness:
  - $\mathrm{gen_d}(v) = \mathrm{use}(v)$
  - $\mathrm{kill_d}(v) = \mathrm{def}(v)$
  - $\mathrm{out_d}(v) = \bigcup_{s \in (\dots)} \mathrm{in_d}(s)$
  - $\mathrm{in_d}(v) = \mathrm{gen_d}(\mathrm{v}) \cup \left(\mathrm{out_d}(v) \setminus \mathrm{kill}(v)\right)$

Termination in “Expression Liveness”

- Only re-evaluate vertices when a predecessor has a change in the out_e set.

- Will eventually reach a fixed-point.

Not so simple…

- Problem: what about more complex expressions:

$$(x + y) == (z + w)$$

- We can keep many expressions alive:
  - *Parts:* $x + y,\ z + w$
  - *The entire:* $(x + y) == (z + w)$
  - What about: not x, y alive, but instead $\alpha = x + y$ alive
    - $\alpha == (z + w)$
  - Etc.

Idea: Break up vertices

- Break every expression into small constituent components. **Generate extra code**!

"(x + y) == (z + w)" ⇒ { x+y, z+w }

Original	Generate Code	
c := (x+y) == (z+w)	a := x+y	
	b := z+w	
d := z+w	c := (x+y)==(z+w)	
	d := z+w	

Some part of the expression is used later → (pointing at d := z+w)

return c+d

Apply Expression Liveness Analysis

- Replace expressions with aliased expressions

Original	Generate Code	Apply Aliases
c := (x+y) == (z+w)	a := x+y	a := x+y
	b := z+w	b := z+w
d := z+w	c := (x+y)==(z+w)	c := a==b
	d := z+w	d := b

return c+d

Apply Data Liveness Analysis

- Reuse variable names

Original	Generate Code	Apply Aliases	x	y	z	w	a	b	c	d	New Data Aliases
c := (x+y) == (z+w)	a := x+y	a := x+y									
	b := z+w	b := z+w									
d := z+w	c := (x+y)==(z+w)	c := a==b									
	d := z+w	d := b									
return c+d											

Apply Data Liveness Analysis

- Can eliminate redundant operations

Original	Generate Code	Apply Aliases	x	y	z	w	a	b	c	d	New Data Aliases
c := (x+y) == (z+w)	a := x+y	a := x+y	x	y	z	w					x := x+y
	b := z+w	b := z+w					x				y := z+w
d := z+w	c := (x+y)==(z+w)	c := a==b						y			x := x==y
	d := z+w	d := b							x		~~y := y~~
return c+d									x	y	x := x+y ret x

Review

- **Data Liveness Analysis:**
  - Reduces the amount of data you need in memory at any given time
  - Somewhat related to minimizing register usage (minimizing registers can be done after data+expression liveness)

- **Expression Liveness Analysis:**
  - Can eliminate the need to re-process expressions

- **Combined**:
  - They can eliminate instructions and reduce memory consumption.
  - Without the other, significantly less effective.

More Optimization?

Statements	# live
x := x+y	4 (x,y,z,w)
y := z+w	4 (x,y,z,w)
x := x==y	2 (x,y)
x := x+y	2 (x,y)
ret x	1 (x)

Does that mean we need 4 registers?

More Optimization?

Statements	# live
x := x+y	4 (x,y,z,w)
y := z+w	4 (x,y,z,w)
x := x==y	2 (x,y)
x := x+y	2 (x,y)
ret x	1 (x)

Does that mean we need 4 registers?

Nope! More optimization possible that will be related to the target architecture.

Register Minimalization is not Dataflow/Expression Analysis

Statements	# live	X64	# live
x := x+y	4 (x,y,z,w)	mov rax,[x]	1 (rax)
		add rax,[y]	1 (rax)
y := z+w	4 (x,y,z,w)	mov rcx,[z]	2 (rax,rcx)
		add rcx,[w]	2 (rax,rcx)
x := x==y	2 (x,y)	cmp rax,rcx	2 (rax,rcx)
		xor rax,rax	2 (rax,rcx)
		sete al	2 (rax,rcx)
x := x+y	2 (x,y)	add rax,rcx	2 (rax,rcx)
ret x	1 (x)	ret	1 (rax)

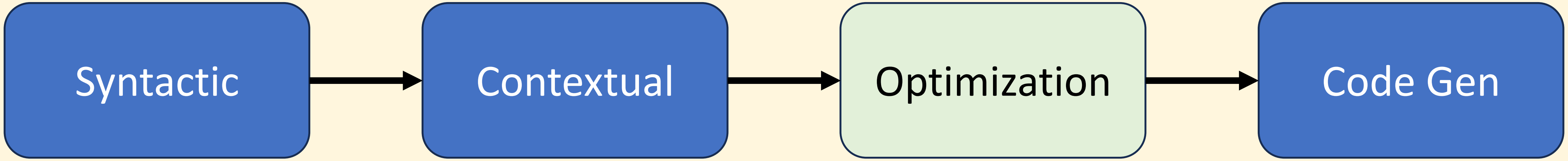
Only needed two registers.

Why? Because x64 can do "load memory" operations inside of instructions!

Optimized Code Generation – AST Reprocessing

You can find optimizations before you reach code generation.

- AST-level optimizations can become an “Optimization” AST traversal phase before CodeGeneration



Expression Optimization

- Developer writes the code:

```
// This page needs to start 3 full pages
//   after base address
int somePg = ( 3 * 4096 ) + 0x80000000;
```

..So we generate the code

Input Code

```
int somePg = ( 3 * 4096 )
      + 0x80000000;
```

Generated Code

```
mov rax,4096
imul 3                # pseudocode
add rax,0x80000000
mov dword [somePg],rax
```

Anyone have any problems with this?

Fix the AST

Input Code

```
int somePg = ( 3 * 4096 )
      + 0x80000000;
```

Input AST

BinExpr (+)
- **BinExpr (*)**
  - **3**
  - **4096**
- **LiteralExpr (0x80000000)**

Create visitor

"Expression Pre-evaluator Visitor"

Visit BinExpr / UnaryExpr

Visit sub-expressions.

If all expressions are LiteralExpr, Return a new LiteralExpr

Returned LiteralExpr pre-evaluates constant operations.

Fix the AST (Example)

Input Code

```
int somePg = ( 3 * 4096 )
      + 0x80000000;
```

Input AST

BinExpr (+)
- **BinExpr (*)**
  - **3**
  - **4096**
- **LiteralExpr (0x80000000)**

Output AST

First Converge

BinExpr (+)
- **LiteralExpr (12288)**
- **LiteralExpr (0x80000000)**

Fix the AST (Example 2)

Input Code

```
int somePg = ( 3 * 4096 )
      + 0x80000000;
```

Input AST

BinExpr (+)
    BinExpr (*)
        3
        4096
    LiteralExpr (0x80000000)

Output AST

First Converge

BinExpr (+)
    LiteralExpr (12288)
    LiteralExpr (0x80000000)

Second Converge

LiteralExpr(0x80003000)

Optimization Visitor Model (Finish)

- Each visit returns an AST
- Most of them return themselves.
- Otherwise, return a new optimized AST.

- E.g., BinExpr.LHS = BinExpr.LHS.visit(…);
  BinExpr.RHS = BinExpr.RHS.visit(…);
  … Now are LHS and RHS instanceof LiteralExpr?
  … If so, return new LiteralExpr(… **"LHS op RHS"** …);

Intel C Compiler- Case Study

An example as an intro to multiple code-path generation.

“Genuine Intel”

- This study is an interesting intersection of topics.
- Things to keep in mind:
  - x86 belongs to Intel
  - x64 (the 64-bit extension) was developed by AMD, was so popular that Intel was forced to adopt it (in lieu of IA64)
  - AMD has a license to make x86/x64 processors
- Exactly how fair does Intel have to be towards its competition?

ICC Generates this code

- cpuid
- cmp ebx, 0x756E6547
- jne OtherLoc
- cmp edx, 0x49656E69
- jne OtherLoc
- cmp ecx, 0x6C65746E
- jne OtherLoc

Problem Statement:

This code looks nothing like the input program's source code.

Consider the code:

```
int a[100], b[100], c[100];
… // Initialize Data

for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

Generate Simple Code

Original

```
int a[100], b[100], c[100];
… // Initialize Data

for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

x64

Assume data initialized already

int = 4 bytes long

From this line, generate code

Option 1 (Initialization)

Original

```
int a[100], b[100], c[100];

… // Initialize Data

for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

x64

```
# From this line, generate code
  mov rax,0                    # Initialize i=0
loopStart: cmp rax,100         # compare, i to 100
  jge loopEnd                  # if i >= 100, end loop
  mov rdx, [a+rax*4]           # rdx= a[i]
  add rdx, [b+rax*4]           # rdx += b[i]
  mov [c+rax*4],rdx            # c[i] = rdx
  inc rax                      # ++i
  jmp loopStart                # loop
loopEnd:
```

Option 1 (Condition)

Original

```
int a[100], b[100], c[100];

… // Initialize Data

for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

x64

```
# From this line, generate code
  mov rax,0                  # Initialize i=0
loopStart: cmp rax,100       # compare, i to 100
  jge loopEnd                # if i >= 100, end loop
  mov rdx, [a+rax*4]         # rdx= a[i]
  add rdx, [b+rax*4]         # rdx += b[i]
  mov [c+rax*4],rdx          # c[i] = rdx
  inc rax                    # ++i
  jmp loopStart              # loop
loopEnd:
```

Option 1 (Body)

Original

```
int a[100], b[100], c[100];

… // Initialize Data

for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

x64

```
# From this line, generate code
  mov rax,0                  # Initialize i=0
loopStart: cmp rax,100       # compare, i to 100
  jge loopEnd                # if i >= 100, end loop
  mov rdx, [a+rax*4]         # rdx= a[i]
  add rdx, [b+rax*4]         # rdx += b[i]
  mov [c+rax*4],rdx          # c[i] = rdx
  inc rax                    # ++i
  jmp loopStart              # loop
loopEnd:
```

Option 1 (Incremental)

Original

```
int a[100], b[100], c[100];

… // Initialize Data

for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

x64

```
# From this line, generate code
  mov rax,0                      # Initialize i=0
loopStart: cmp rax,100           # compare, i to 100
  jge loopEnd                    # if i >= 100, end loop
  mov rdx, [a+rax*4]             # rdx= a[i]
  add rdx, [b+rax*4]             # rdx += b[i]
  mov [c+rax*4],rdx              # c[i] = rdx
  inc rax                        # ++i
  jmp loopStart                  # loop
loopEnd:
```

Observation 1

- Each loop has no dependency on the previous loop

```
for( int i = 0; i < 100; ++i ) {
    c[i] = a[i] + b[i];
}
```

Option 2

Original

```
int a[100], b[100], c[100];

… // Initialize Data

for( int i = 0; i < 100; ++i )
{
      c[i] = a[i] + b[i];
}
```

```
cld
mov rcx,100
lea rdi,[c]
lea rsi,[a]
rep movsd
mov rax,0
```

```
startLoop: cmp rax,100
  jge endLoop
  mov rdx,[b+rax*4]
  add [c+rax*4],rdx
  inc rax
  jmp startLoop
endLoop:
```

Option 2 – Loop Comparison

Option 1		Option 2
loopStart: cmp rax,100	# compare, i to 100	startLoop: cmp rax,100
jge loopEnd	# if i >= 100, end loop	jge endLoop
mov rdx, [a+rax*4]	# rdx= a[i]	← **One less instruction**
add rdx, [b+rax*4]	# rdx += b[i]	mov rdx,[b+rax*4]
mov [c+rax*4],rdx	# c[i] = rdx	add [c+rax*4],rdx
inc rax	# ++i	inc rax
jmp loopStart	# loop	jmp startLoop
loopEnd:		endLoop:

Option 2 – Initialization Comparison

Option 1		Option 2	
mov rax,0	# Initialize i=0	cld	# Set DF=0
		mov rcx,100	
		lea rdi,[c]	
		lea rsi,[a]	
		rep movsd	# store all of [a] into [c]
		mov rax,0	# Initialize i=0

Is 1 or 2 always better than the other?

What is better?

- Depends on how fast rep movsd works.
- Instead, consider an even more optimized vectorized instructions, like AVX.

- VEX.128.66.0F.WIG FE VPADDD
  - Does 3 additions in one, if the loop was only 3 integers, could do the entire loop in one instruction.

Intel C Compiler

- Optimization can rewrite code, but what if I take it a step further?

- How about this:

```
for( int i = 0; i < 100; ++i )
        d[i] = s[i];
```

Intel C Compiler

No AMD Allowed

- How about this:

```
for( int i = 0; i < 100; ++i )
        d[i] = s[i];
```

COMPILER REWRITES

```
mov rcx,100
if( "IntelProcessor" ) {
    lea rsi,[s]
    lea rdi,[d]
    rep movsd
} else {
    Do loop like option 1
}
```

Can make it worse!

```
If( "IntelProcessor" ) {
        // do optimized code
} else {
        // don't even copy integers (4 bytes)
        // copy ONE BYTE AT A TIME
        for(...) { ... mov [rdi+0], al
                mov [rdi+1], ah
                mov [rdi+2], cl
                mov [rdi+3], ch ... }
}
```

People even make patchers…

- As a part of the development process, when code is compiled using ICC…
- Use a tool as a part of the build process to always patch out "if(Intel)" checks

Largely patched now in ICC

- …or is it?
- But this gives us some excellent ideas on our own compiler project!

- What if we can optimize for certain scenarios during runtime?
  (Even if those scenarios don't always happen!)

ICC Generates this code

- cpuid # Get CPU info
- cmp ebx, 0x756E6547 # “Genu”
- jne OtherLoc
- cmp edx, 0x49656E69 # “inel”
- jne OtherLoc
- cmp ecx, 0x6C65746E # “ntel”
- jne OtherLoc

Multiple Code Path Generation

Round up to the nearest multiple of 8

- Take a moment, and think about the code needed to round an integer, x, to the nearest multiple of 8

Round up to the nearest multiple of 8

- Take a moment, and think about the code needed to round an integer, x, to the nearest multiple of 8

```
while( ( x % 8 ) != 0 )
    ++x;
```

Now try nearest multiple of “y”

- Not a big change

```
while( ( x % y ) != 0 )
    ++x;
```

Rounding up to a power of 2

- Earlier example (to the next multiple of 8)
- Analyze the following:

```
add [x],7          # x += 7
and [x],~7         # x &= 0xFFFFFFF8 (32-bit sign extended)
```

What does this code accomplish?

So let's rewrite the second example

```
if( __popcnt(y) == 1 && y > 0 ) {
        c = y-1;                         // Optimized Code
        x = ( x + c ) & ~c;              // Not always faster
} else {
        while( (x % y) != 0 )            // General Code
                ++x;
}
```

Loop Unrolling

Let’s try to handle some cases of “small iterations are still faster”

Compiler cleans up the mess

Bad Code

```
printf("\t");
printf("\t");
printf("\t");
printf("\t");
```

Clean Code

```
for( int i = 0; i < 4; ++i )
    printf("\t");
```

Compiler cleans up the mess

~~Bad~~ Faster Code

```
printf("\t");
printf("\t");
printf("\t");
printf("\t");
```

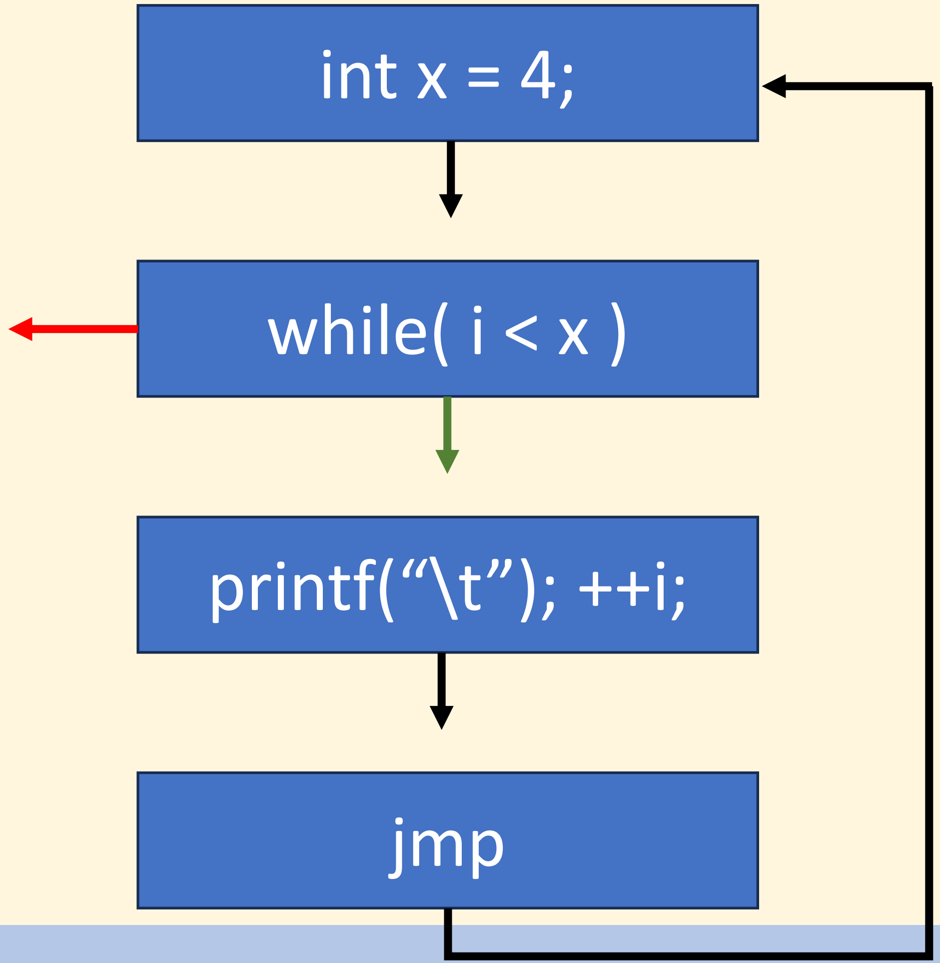
~~Clean~~ Inefficient Code

```
for( int i = 0; i < 4; ++i )
    printf("\t");
```

From Dataflow Analysis

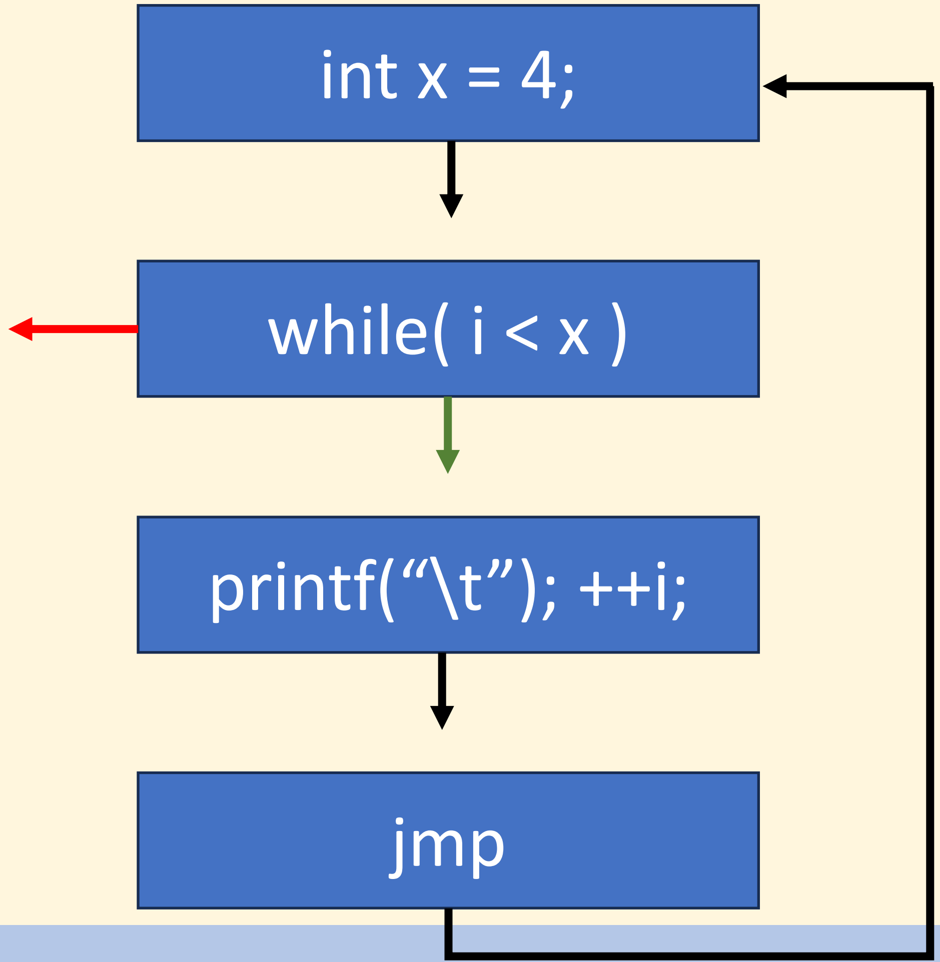
- Can track the lifetime of variables.
- This also means we can simulate the range of variables.
- This type of analysis is expensive
  - Idea: simulate only variables that are used in conditions where the variable’s lifetime is not easily invalidated.

Input CFG



- Observation: lifetime of condition "x" is easily analyzable.
- Loop is bounded, unroll the loop

Output Code




```
push "\t\0"
call [printf]
call [printf]
call [printf]
call [printf]
add rsp,8
```

Way better than having a bunch of condition jumps, comparison statements, etc.

Formally, this is the problem

- $f(n) \in O\big(g(n)\big) \equiv \forall n\!: \boldsymbol{n \geq N} :: f(n) \leq g(n)$

- We can optimize around the scenario $n < N$ in the compiler's generated code.

Formally, this is the problem

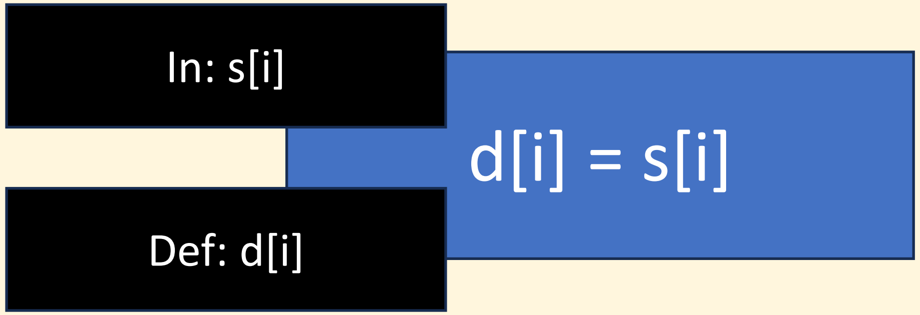
- $f(n) \in O(g(n)) \equiv \forall n\colon \boldsymbol{n \geq N} :: f(n) \leq g(n)$
- Idea: Add code, if $n < N$, then take a different code path (use optimized algorithm instead of normal code)

How to Apply Multiple Code Paths

Check Dataflow Analysis

- Is there a memory dependency on the previous loop?

```
for( int i = 0; i < 100; ++i )
        d[i] = s[i];
```

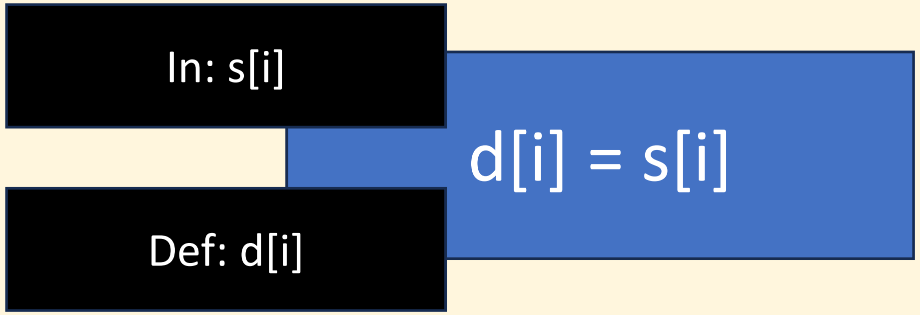


No i-1 or i+1, can apply code generation optimizations

Check Dataflow Analysis (2)

- Is there a memory dependency on the previous loop?

```
for( int i = 0; i < 100; ++i )
        d[i] = s[i];
```



Use: **rep movsd**

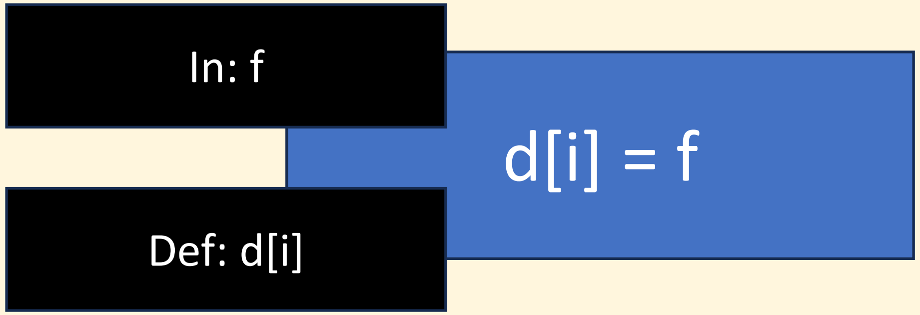
No i-1 or i+1, can apply code generation optimizations

Check Dataflow Analysis (3)

- Is there a non-array dependency in every loop?

```
f = 0;
for( int i = 0; i < 100; ++i )
        d[i] = f;
```

Use: rep stosd



Dependency is on single memory location

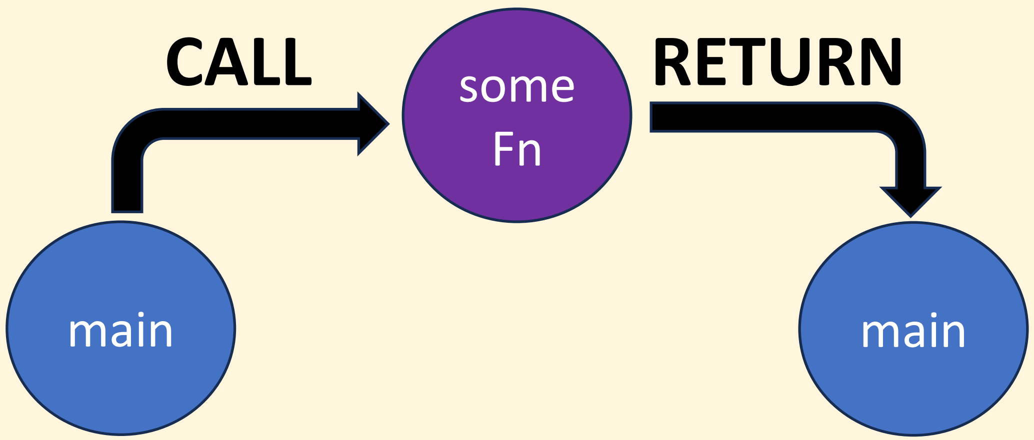
Optimization – Your imagination is the limit

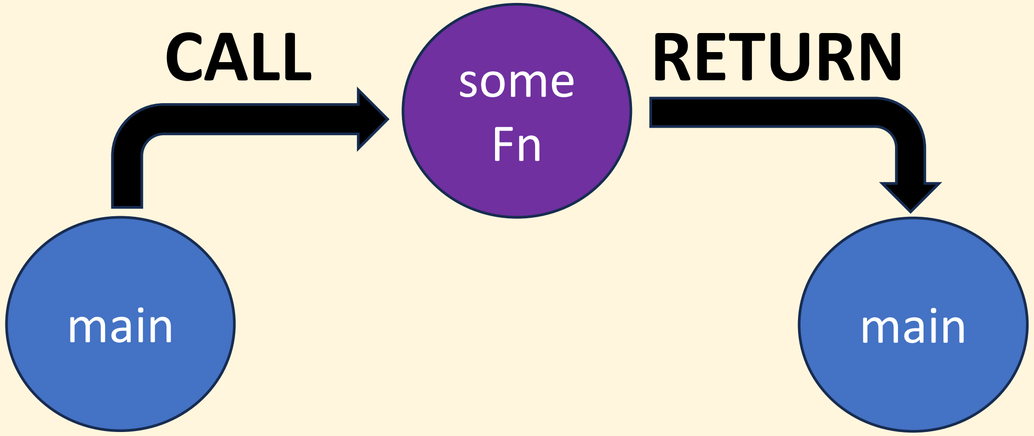
Consider the following code:

```
int someFn() {
        …
        return code;
}

void mainFn() {
        int x = someFn();
        printf(“%d\n”,x);
}
```

Consider the following code:

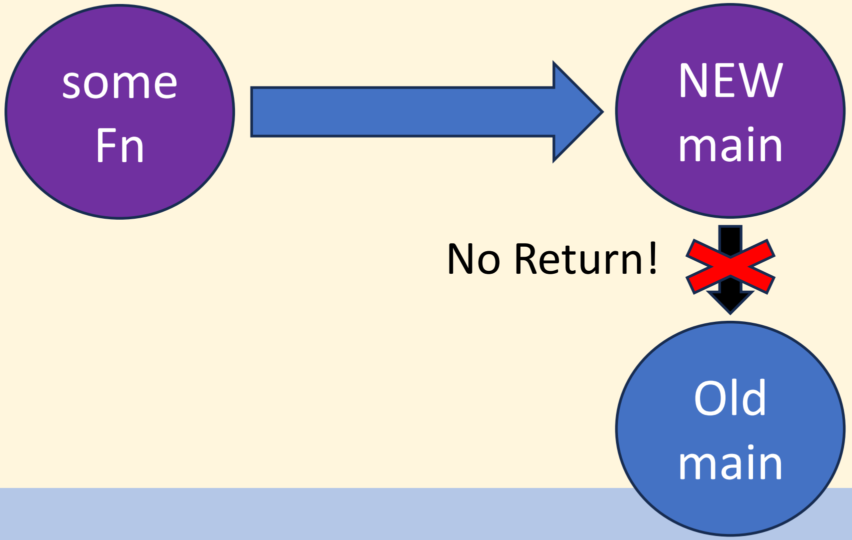
```
int someFn() {
        …
        return code;
}

void mainFn() {
        int x = someFn();
        printf("%d\n",x);
}
```



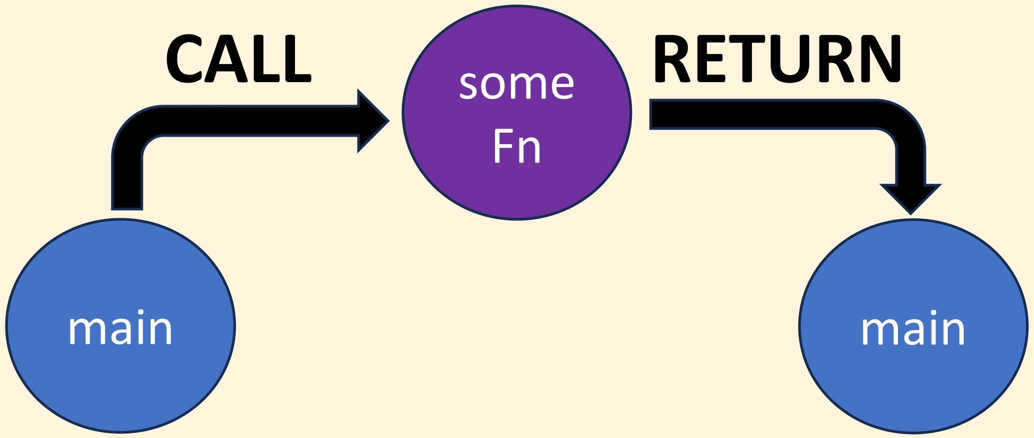



```
int someFn() {
        …
        return code;
}

void mainFn() {
        int x = someFn();
        printf("%d\n",x);
}
```

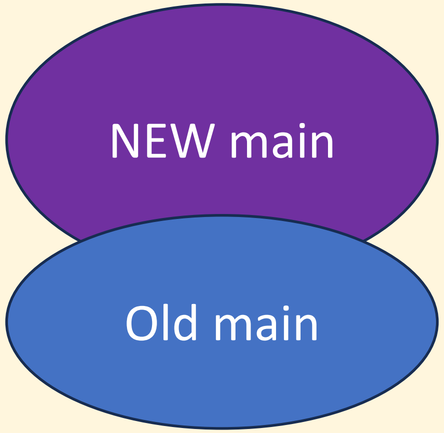
ICC did the craziest optimization:



Old Code

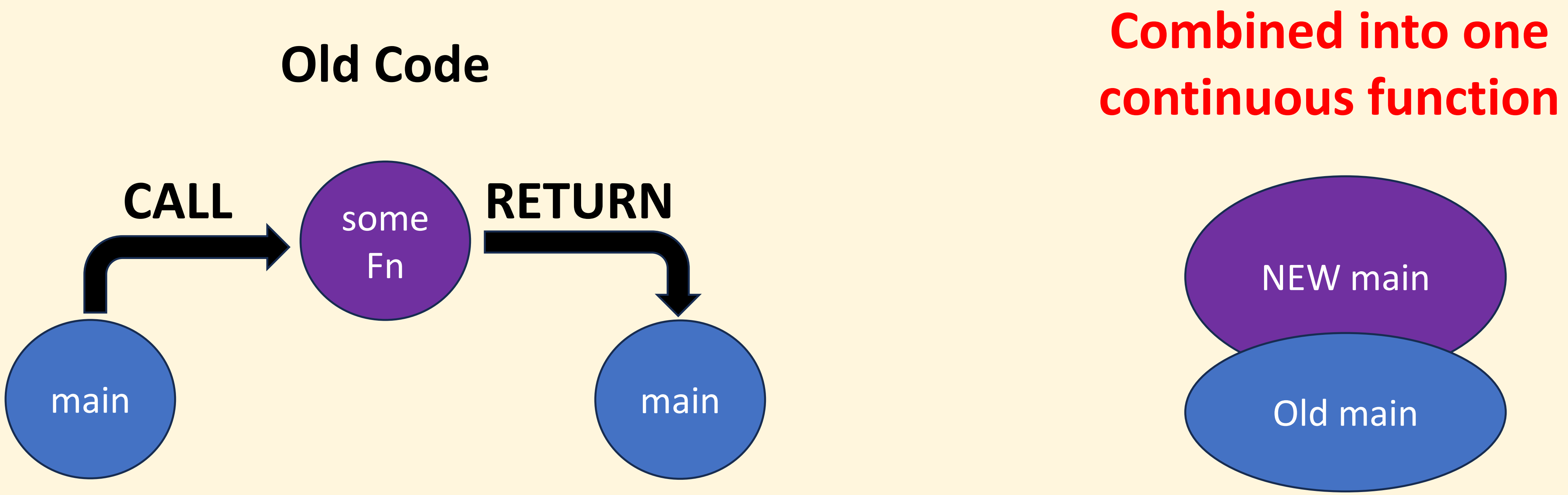


Combined into one continuous function



Ideas?

- How can we create such optimizations?



Inline operations

- Compiler can detect “method was only used once”, instead of generating “push, call, return, pop”, just take the method’s code and place it where it is used.

- Apply a translation to ParameterDecl to map to local variables.

Have a great weekend!

- Work on PA4, get some experience for the Midterm
- Midterm next week.

- WA3 due tonight.

End