Finals Review

CS406: Compilers

Spring 2022

Practice Exercise: CFGs and low-level loop optimizations

1. Draw CFG for the code shown. Identify loops if any. For each loop identified, mark entry node and all basic blocks.

(refer to slides 10, 36, and 48 of week12)

2. Identify loop invariant statements. Can they be moved outside their enclosing loop?

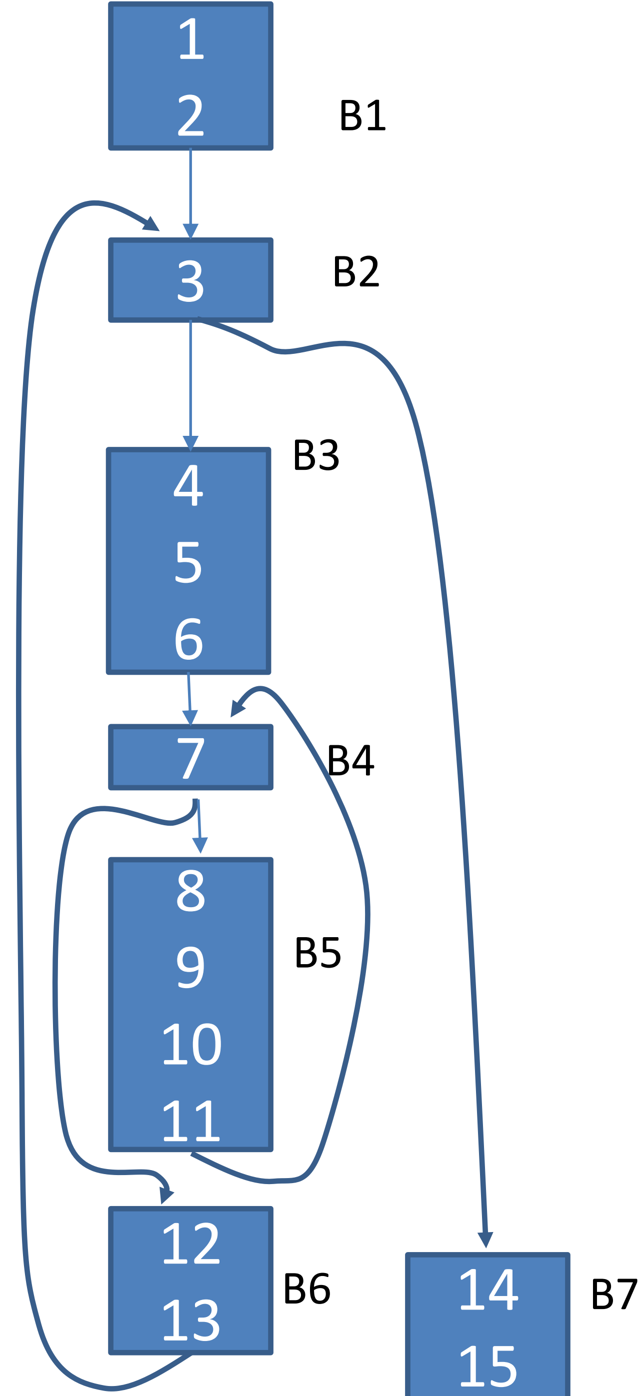
(refer to slides 59, 65-67, week12)

3. Identify induction variables. Show the code that results after performing possible strength reduction

(refer to slides 59, 65-69, week12)

```
1. X = 2;
2. Y = 10;
3. if (X < Y) goto 14
4. A = Y * X;
5. B = X * 2 + Y;
6. Z = 10;
7. if (B < Z) goto 12
8. D = Y + Z * -3;
9. Q = Y - 8;
10. Z = Z - Q;
11. goto 7;
12. X = X + 2;
13. goto 3;
14. Y = D;
15. halt;
```

```
1. X = 2;
2. Y = 10;
3. if (X < Y) goto 14
4. A = Y * X;
5. B = X * 2 + Y;
6. Z = 10;
7. if (B < Z) goto 12
8. D = Y + Z * -3;
9. Q = Y - 8;
10. Z = Z - Q;
11. goto 7;
12. X = X + 2;
13. goto 3;
14. Y = D;
15. halt;
```

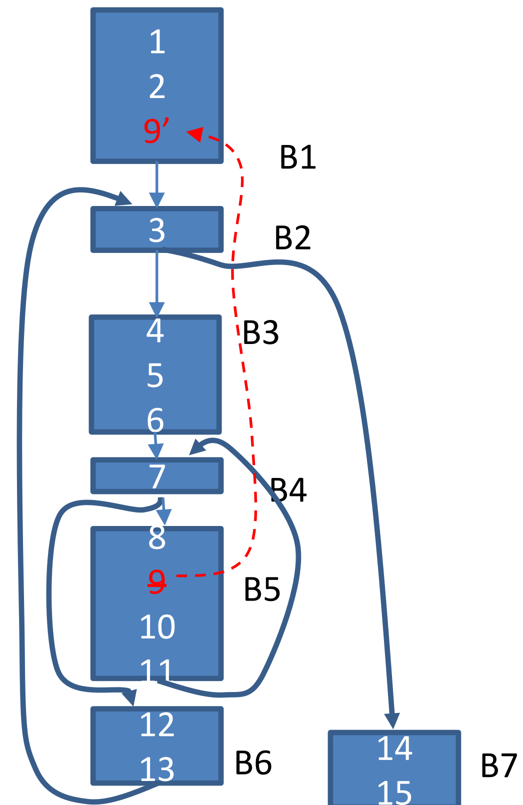


Loop headers/entry nodes: B2 and B4. The BBs in the loop for B2 are: B2, B3, B4, B5, B6. The BBs in the loop for B4 are: B4, B5.

Loop invariant statements: 6, 9.

- 6 cannot be moved, because Z is defined twice inside the loop (note that if we try to move it outside the loop, Z will not get reinitialized before the inner loop).
- 9 can be moved, though. Note that first it gets moved outside the inner loop, but then is still loop invariant (because Y is only defined outside the loop), so can be moved outside the outer loop, as well. Code after moving invariant statement:

```
1.   X = 2;
2.   Y = 10;
9'. Q = Y - 8;
3.   if (X < Y) goto 14
4.     A = Y * X;
5.     B = X * 2 + Y;
6.     Z = 10;
7.     if (B < Z) goto 12
8.       D = Y + Z * -3;
10.      Z = Z - Q;
11.      goto 7;
12.    X = X + 2;
13.    goto 3;
14. Y = D;
15. halt;
```



In the inner loop, Z is an induction variable (because Q is loop invariant), and D is a mutual induction variable. After performing strength reduction on the inner loop, we get:

```
1.   X = 2;
2.   Y = 10;
9’. Q = Y - 8;
3.   if (X < Y) goto 14
4.     A = Y * X;
5.     B = X * 2 + Y;
6.     Z = 10;
8’.    D’ = Y + Z * -3;
7.     if (B < Z) goto 12
8.       D = D’
10.      Z = Z - Q;
10’.     D’ = D’ + -3 * -Q;
11.      goto 7;
12.   X = X + 2;
13.   goto 3;
14. Y = D;
15. halt;
```

In the outer loop, X is an induction variable, and both A and B are mutual induction variables. After strength reduction, we get:

```
1.   X = 2;
2.   Y = 10;
9’. Q = Y - 8;
4’. A’ = Y * X;
5’. B’ = X * 2 + Y;
3.  if (X < Y) goto 14
4.    A = A’
5.    B = B’
6.    Z = 10;
6’.   D’ = Y + Z * -3;
7.    if (B < Z) goto 12
8.       D = D’
10.      Z = Z - Q;
10’.     D’ = D’ + -3 * -Q;
11.      goto 7;
12.    X = X + 2;
12’.   A’ = A’ + 2 * Y;
12’’   B’ = B’ + 4;
13.    goto 3;
14.  Y = D; 15. halt
```

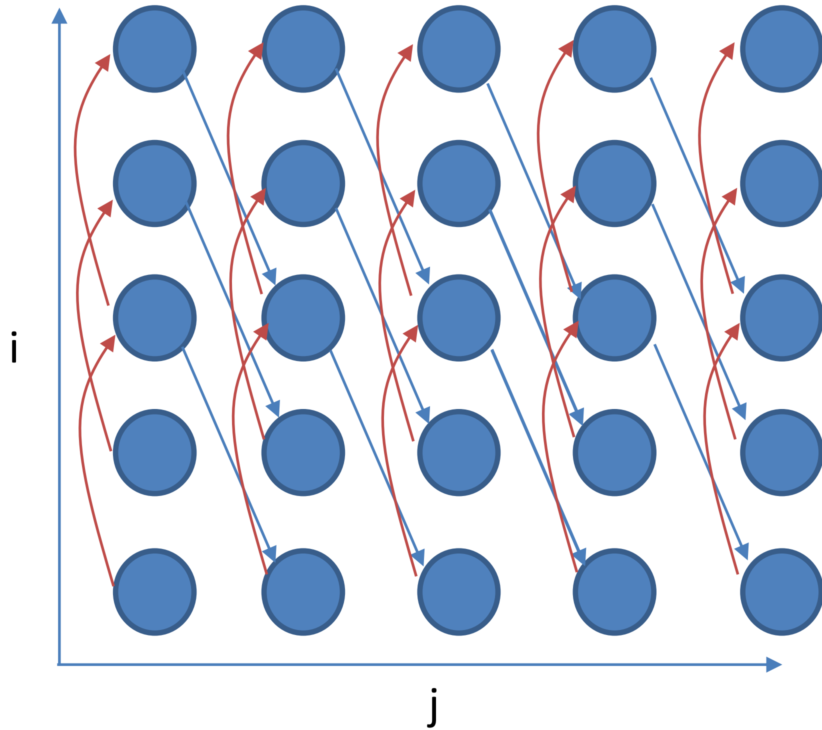
Exercise

1. Draw iteration graph for:

```
for(j=0;j<5;j++)
   for(i=0;i<5;i++)
            a[j][i]=a[j-1][i+2]+a[j][i-2]
```

2. What are the distance and direction vectors?
3. Can the loops be interchanged?
4. Repeat 1,2,and 3 for the following loop.

```
for(i=0;i<5;i++)
    for(j=0;j<5;j++)
        a[i][j]=a[i-1][j-2]
```



The blue arrows are the dependences between the write and the A[j-1][i+2]read, and the red arrows are the dependences between the write and the A[j][i-2]read.

The distance vectors are (1,−2) and (0,2)

The direction vectors are (+,−) and (0,+) (or, alternately, (<,>) and (0,<))

The loops cannot be interchanged because the (1,−2) flow dependence would trans-form into a (−2,1) dependence, which is not possible (more specifically, we would eliminate the flow dependence and introduce a (2,−1) anti-dependence).

Short Quiz

- https://forms.gle/zmUph7ixTN9CaUMF8

Practice Exercise: Peephole Optimizations - CSE

1. Show the results of performing CSE on the code shown. Write 3AC after performing CSE.

2. Suppose A and F are aliased. How would that change the results of CSE

```
1. READ(A)
2. READ(B)
3. C = A + B
4. A = A + B
5. B = C * D
6. T1 = C * D
7. T2 = T1 + C
8. F = A + B
9. C = F + B
10. G = A + B
11. T3 = F + B
12. WRITE(T3)
```

Worksheet

	Expression available before executing the stmt	After performing CSE
1. READ(A)	1.	
2. READ(B)	2.	
3. C = A + B	3.	
4. A = A + B	4.	
5. B = C * D	5.	
6. T1 = C * D	6.	
7. T2 = T1 + C	7.	
8. F = A + B	8.	
9. C = F + B	9.	
10. G = A + B	10.	
11. T3 = F + B	11.	
12. WRITE(T3)	12.	

Worksheet

Suspose A and F are aliased	Expression available before executing the stmt	After performing CSE
1. READ(A)	1.	
2. READ(B)	2.	
3. C = A + B	3.	
4. A = A + B	4.	
5. B = C * D	5.	
6. T1 = C * D	6.	
7. T2 = T1 + C	7.	
8. F = A + B	8.	
9. C = F + B	9.	
10. G = A + B	10.	
11. T3 = F + B	11.	
12. WRITE(T3)	12.	

Each row shows in parentheses what expressions are available before the expression is evaluated.

```
1. READ(A)
2. READ(B)
3. C = A + B
4. A = A + B     (A + B)
5. B = C * D     (nothing -- writing to A kills A + B)
6. T1 = C * D       (C * D)
7. T2 = T1 + C      (C * D)
8. F = A + B     (T1 + C, C * D)
9. C = F + B     (A + B, T1 + C, C * D)
10. G = A + B       (A + B, F + B)
11. T3 = F + B      (A + B, F + B)
12. WRITE(T3)       (A + B, F + B)
```

After performing CSE:

```
1. READ(A)
2. READ(B)
3. C = A + B
4. A = C           (A + B)
5. B = C * D       (nothing -- writing to A kills A + B)
6. T1 = B          (C * D)
7. T2 = T1 + C     (C * D)
8. F = A + B       (T1 + C, C * D)
9. C = F + B       (A + B, T1 + C, C * D)
10. G = F          (A + B, F + B)
11. T3 = C         (A + B, F + B)
12. WRITE(T3)      (A + B, F + B)
```

If A and F are aliased, writing to F will kill any expression that uses A (and vice versa), and also, computing F + B is the same as computing A + B.

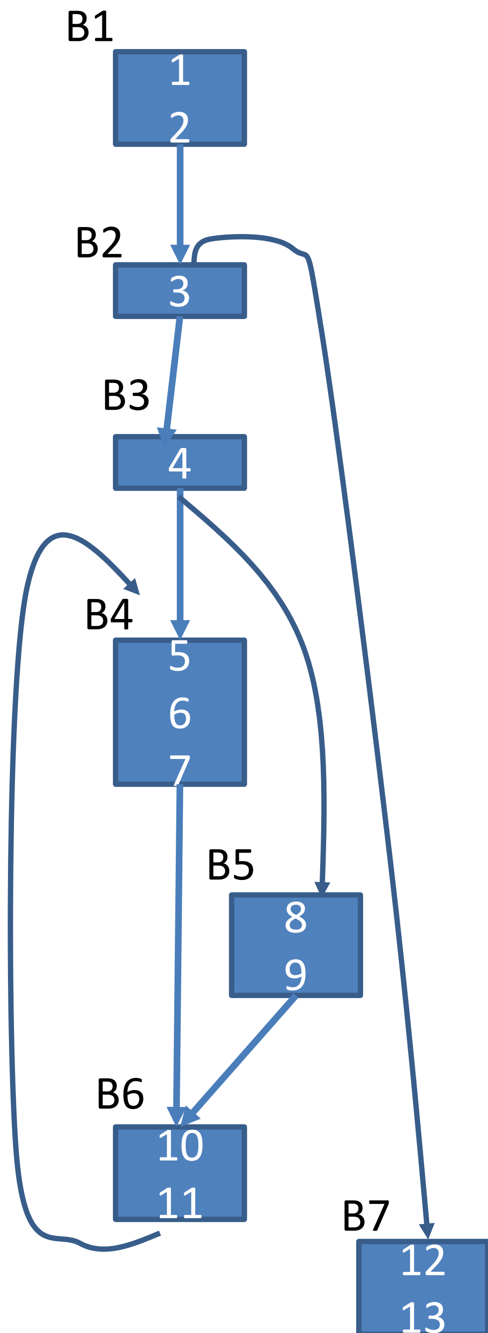
```
1. READ(A)
2. READ(B)
3. C = A + B
4. A = C          (A + B, F + B)
5. B = C * D      (nothing --writing to A kills A+B and F+B)
6. T1 = B         (C * D)
7. T2 = T1 + C    (C * D)
8. F = A + B      (T1 + C, C * D)
9. C = F + B      (T1 + C, C * D -- writing to F kills A+B)
10. G = C         (A + B, F + B – we are using C, not F)
11. T3 = C        (A + B, F + B)
12. WRITE(T3)     (A + B, F + B)
```

Practice Exercise: Dataflow analysis (available expressions)

1. Show the results of running an "available expressions" analysis on the code shown. For each line of code, show which expressions are available in that line of code.

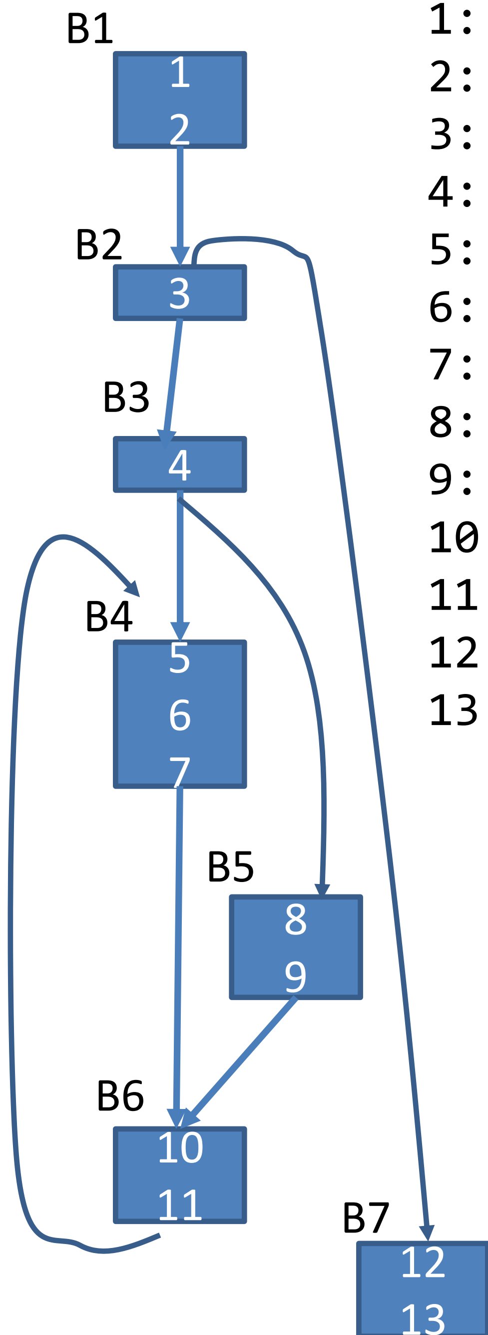
(refer to slide 8, week 14. Section 9.2.6 in Dragon book)

```
1: x = 4;
2: y = 7;
3: if (x > c) goto 12
4: if (y > 3) goto 8
5: c = x + 1;
6: b = a + x;
7: goto 10
8: a = a + x;
9: b = x + 1;
10: y = a + x;
11: goto 3;
12: c = a + x
13: halt
```




```
1: x = 4;
2: y = 7;
3: if (x > c) goto 12
4: if (y > 3) goto 8
5: c = x + 1;
6: b = a + x;
7: goto 10
8: a = a + x;
9: b = x + 1;
10: y = a + x;
11: goto 3;
12: c = a + x
13: halt
```

worksheet




```
1: x = 4;
2: y = 7;
3: if (x > c) goto 12
4: if (y > 3) goto 8
5: c = x + 1;
6: b = a + x;
7: goto 10
8: a = a + x;
9: b = x + 1;
10: y = a + x;
11: goto 3;
12: c = a + x
13: halt
```

Basic Block	Pred.	Succ.	Gen	Kill
B1	Entry	B2		
B2	B6,B1	B3,B7		
B3	B2	B4,B5		
B4	B3	B6		
B5	B3	B6		
B6	B4,B5	B2		
B7	B2	Exit		

Basic Block	Pred.	Succ.	Gen	Kill
B1	Entry	B2	4,7	x+1, a+x, x>c, y>3
B2	B6,B1	B3,B7	x>c	-
B3	B2	B4,B5	y>3	-
B4	B3	B6	x+1,a+x	x>c
B5	B3	B6	x+1	a+x
B6	B4,B5	B2	a+x	y>3
B7	B2	Exit	a+x	x>c

Basic Block	IN	OUT
B1	-	4, 7
B2	4, 7	4, 7, x>c
B3	4, 7, x>c	4, 7, x>c, y>3
B4	4, 7, x > c, y > 3	4, 7, y > 3, x + 1, a + x
B5	4, 7, x > c, y>3	4, 7, x > c, x+1, y>3
B6	4, 7, y>3, x+1, a + x	4, 7, x + 1, a + x, y>3
B7	4, 7, x > c	4, 7, a+x

GDB

– GNU Debugger – A tool for inspecting your C/C++ programs
  - How to begin inspecting a program using gdb?
  - How to control the execution?
  - How to display, interpret, and alter memory contents of a program using gdb?
  - Misc – displaying stack frames, visualizing assembler code.

GDB

– Compile your programs with –g option

```
hegden$gcc gdbdemo.c -o gdbdemo -g
hegden$
```

```
 1 #include<stdio.h>
 2 int foo(int a, int b)
 3 {
 4     int x = a + 1;
 5     int y = b + 2;
 6     int sum = x + y;
 7
 8     return x * y + sum;
 9 }
10
11 int main()
12 {
13    int ret = foo(10, 20);
14    printf("value returned from foo: %d\n",ret);
15    return 0;
16 }
```

GDB – Start Debug

- Start debug mode (`gdb gdbdemo`)
  - Note the executable on first line (not .c files)
  - Note the last line before `(gdb)` prompt:
    - if –g option is not used while compiling, you will see “`(no debugging symbols found)`”

```
[ecegrid-thin4:~/ECE264] hegden$gdb gdbdemo
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/min/a/hegden/ECE264/gdbdemo...done.
(gdb)
```

GDB

- GNU Debugger – A tool for inspecting your C/C++ programs
  - How to begin inspecting a program using gdb?
  - How to control the execution?
  - How to display, interpret, and alter memory contents of a program using gdb?
  - Misc – displaying stack frames, visualizing assembler code.

GDB – Set breakpoints

- Set breakpoints (b)
  - At line 14
  - Beginning of `foo`

```
1 #include<stdio.h>
2 int foo(int a, int b)
3 {
4     int x = a + 1;
5     int y = b + 2;
6     int sum = x + y;
7
8     return x * y + sum;
9 }
10
11 int main()
12 {
13    int ret = foo(10, 20);
14    printf("value returned from foo: %d\n",ret);
15    return 0;
16 }
```

```
(gdb) b gdbdemo.c:14
Breakpoint 1 at 0x400512: file gdbdemo.c, line 14.
(gdb) b foo
Breakpoint 2 at 0x4004ce: file gdbdemo.c, line 4.
(gdb)
```

GDB – Start execution

- Start execution (`r <command-line arguments>`)
  - Execution stops at the first breakpoint encountered

```
(gdb) r
Starting program: /home/min/a/hegden/ECE264/gdbdemo

Breakpoint 3, main () at gdbdemo.c:13
13          int ret = foo(10, 20);
```

  - Continue execution (`c`)

```
(gdb) c
Continuing.

Program exited normally.
```

GDB – Printing

– Printing variable values (`p <variable_name>`)

```
Breakpoint 2, foo (a=10, b=20) at gdbdemo.c:4
4           int x = a + 1;
(gdb) n
5           int y = b + 2;
(gdb) p x
$3 = 11
```

– Printing addresses (`p &<variable_name>`)

```
(gdb) p &x
$5 = (int *) 0x7fffffffc4f4
```

GDB – Manage breakpoints

- Display all breakpoints set (`info b`)

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400512 in main at gdbdemo.c:14
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Delete a breakpoint (`d <breakpoint num>`)

```
(gdb) d 1
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Disable a breakpoint (`disable <breakpoint num>`)

```
(gdb) disable 2
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep n   0x00000000004004ce in foo at gdbdemo.c:4
(gdb)
```

- Enable breakpoint (`enable <breakpoint num>`)

```
(gdb) enable 2
(gdb) info b
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x00000000004004ce in foo at gdbdemo.c:4
```

GDB – Step in

– Steps inside a function call (`s`)

```
Breakpoint 3, main () at gdbdemo.c:13
13          int ret = foo(10, 20);
(gdb) s
foo (a=10, b=20) at gdbdemo.c:4
4     _      int x = a + 1;
```

GDB – Step out

– Jump to return address (`finish`)

```
(gdb) finish
Run till exit from #0  foo (a=10, b=20) at gdbdemo.c:4
0x000000000040050f in main () at gdbdemo.c:13
13          int ret = foo(10, 20);
Value returned is $2 = 275
```

GDB

- GNU Debugger – A tool for inspecting your C/C++ programs
  - How to begin inspecting a program using gdb?
  - How to control the execution?
  - How to display, interpret, and alter memory contents of a program using gdb?
  - Misc – displaying stack frames, visualizing assembler code.

GDB – Memory dump

– Printing memory content (`x/nfu <address>`)

- n = repetition (number of bytes to display)
- f = format ('x' – hexadecimal, 'd'-decimal, etc.)
- u = unit ('b' – byte, 'h' – halfword/2 bytes, 'w' – word/4 bytes, 'g' – giga word/8 bytes)
- E.g. `x/16xb 0x7fffffffc500` (display the values of 16 bytes stored from starting address

```
(gdb) x/16xb 0x7fffffffc500
0x7fffffffc500: 0x20    0xc5    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffc508: 0x0f    0x05    0x40    0x00    0x00    0x00    0x00    0x00
```

GDB – Printing addresses

– Registers ($rsp, $rbp)

- Note that we use the ‘x’ command and not the ‘p’ command.

```
(gdb) x $rsp
0x7fffffffc500: 0x20
(gdb) x $rbp
0x7fffffffc500: 0x20
```

GDB – Altering memory content

- Set command (`set variable <name> = value`)

```
(gdb) n
6           int sum = x + y;
(gdb) p x
$7 = 11
(gdb) p y
$8 = 22
(gdb) set variable y = 0
(gdb) n
8           return x * y + sum;
(gdb) p sum
$9 = 11
```

- Set command (`set *(<type *>addr) = value`)

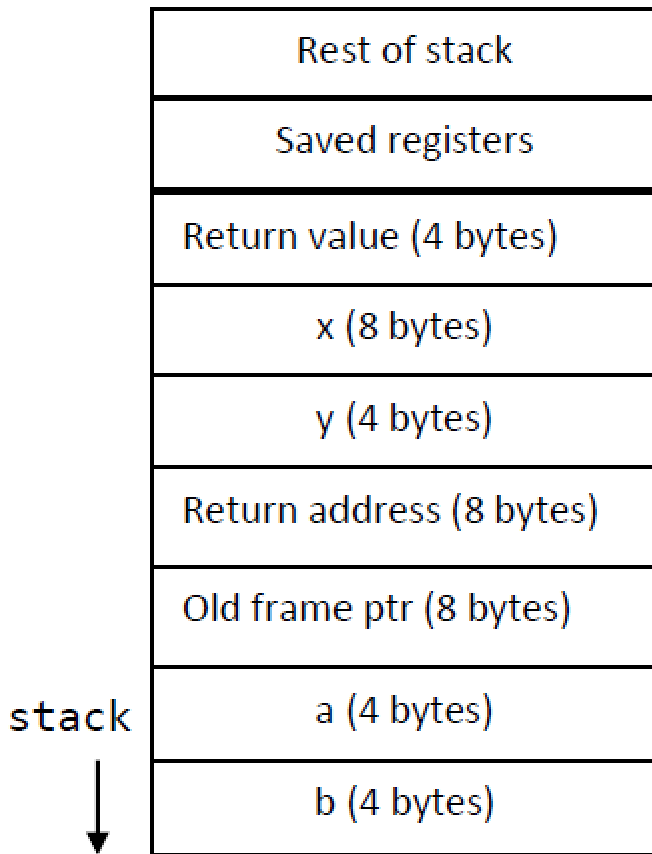
GDB

– GNU Debugger – A tool for inspecting your C/C++ programs

- How to begin inspecting a program using gdb?
- How to control the execution?
- How to display, interpret, and alter memory contents of a program using gdb?
- Misc – displaying stack frames, visualizing assembler code.

Concluding Remarks: CS406 in Spring 2022

- What we did not study
  - Algorithms for fast buffering (lexical analysis)
  - LR(1), LALR, etc. (Syntactic analysis)
  - Syntax directed translation (attributes and attribute grammar)
  - Type systems (code generation)
  - Runtime environment and Garbage Collection
  - Operational and Denotational Semantics
  - Points-to analysis, shape-analysis (optimization) etc.


Rest of stack
Saved registers
Return value (4 bytes)
x (8 bytes)
y (4 bytes)
Return address (8 bytes)
Old frame ptr (8 bytes)
a (4 bytes)
b (4 bytes)
stack