Exercise Session 2

Systems Programming and Computer Architecture

Fall Semester 2024

Disclaimer

Systems@ETH Zürich

- **Website**: n.ethz.ch/~falkbe/
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**: they are there to complement and go into more depth where I found appropriate
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Remark: Labs, Homework

Systems@ETH Zürich

- Labs and Homework are both ungraded
- I would still **highly** recommend doing them
- **Labs**: show you C programming: many **exam tasks** in this style (like bitlab, general coding skills from malloc lab)
- **Pen&Paper**: give you thorough understanding of the concepts (i.e. praxis part)

Remark: SPCA Setup

Systems@ETH Zürich

- Any setup issues? Come to me in the **break** or **after** the exercise session

Agenda

- More on C-programming…
- **`.c`** and **`.h`** files
- **`Make`** and **`makefiles`**
- **`gcc`** flags

Bit Lab

Systems@ETH Zürich

Deadline for Assignment 1 is next week.

Questions?

Systems@ETH Zürich

C Programming Whirlwind Tour

Touching on this week's lectures

Example Structure of a C file

```c
#include <stdio.h>

int i = 79;

static void print_name(void)
{
  const char s[] = "Mothy";
  printf("My name is %s and I work in CAB F %d\n", s, i);
}

int main(int argc, char *argv[])
{
  print_name();
  return 0;
}
```

- You have function definitions and declarations and calls.
- You have variable declarations

How about calling `print_name()` from another source file?

Or

How does the other_program.c know about the location / signature of `print_name()` ?

Solution: Header Files and Modules

- There is a difference between **declaration** and **definition**
  - Declaration gives the signature of the function / variable
  - Definitions gives the code / storage space for variables

- put declarations in header files

```
void print_name(void);
```

```
void print_name(void)
{
  const char s[] = "Mothy";
  printf("My name is %s and I work
          in CAB F %d\n", s, i);
}
```

http://en.wikipedia.org/wiki/Header_file

Outsourced print_name()

```c
/* print_name.h */
void print_name(void);
```

```c
/* print_name.c */

#include <stdio.h>

int i = 79;

void print_name(void)
{
  const char s[] = "Mothy";
  printf("My name is %s and I work in CAB F %d\n", s, i);
}
```

New Structure of Main

```c
#include “print_name.h”

int main(int argc, char *argv[])
{
  print_name();
  return 0;
}
```

Note: You do not need to include stdio.h anymore, since you do not make use of printf here. print_name makes use of printf and stdio.h is included in print_name.h

`#include “print_name.h”` → Your header files (same directory)

`#include “../print_name.h”` (in the parent directory)

`#include “folder/print_name.h”` (in the subdirectory)

`#include <stdio.h>` → Header file of the system (libc)

Some C standard library headers: `<stdlib.h>, <math.h>` …

Different file types

Header Files (*.h)

- Forward declarations (function prototypes, …)
- Globally usable definitions, typedefs, structs, …
- [Macro definitions]

Source Files (*.c)

- Function definitions (source code)
- Variable storage
- Local (static) function declarations & definitions

Note: Everything that is declared in a header file which can be included is considered to be globally accessible. Only put there what's necessary i.e. the public interface

Header Files

- Header files are included by text injection (copy-paste) by macro pre-processor:

```
#include "header1.h"
#include <system-file>
```

- Include Header Guards to make sure that a header file is only included once in a compilation unit (roughly a C file):

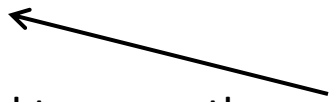
```
#ifndef HEADER_FILE
#define HEADER_FILE

// the entire header file

#endif // HEADER_FILE
```

c-demo

Compiling The Program

- Just executing gcc with your program.c does not work anymore
- You have to specify every source file you used:

```
gcc –o program program.c print_name.c
```



-o is used to name the **output**, if -o is not specified the output will be named **a.out** for historic reasons.

- You do not have to list the header files
  - gcc looks for header files in the current directory
  - gcc also looks for header files in the system include directories

Example: .c and .h files (makefile-demo)

Systems@ETH Zürich

Main.c

```c
#include <stdio.h>
#include "functions.h"

int main(int argc, char** argv){
  printf("hello, world\n");
  printf("square of 3: %d\n", square(3));
  return 0;
}
```

Functions.c

```c
#include <stdio.h>

int square(int x){
  return x*x;
}
```

Functions.h

```c
int square(int);
```

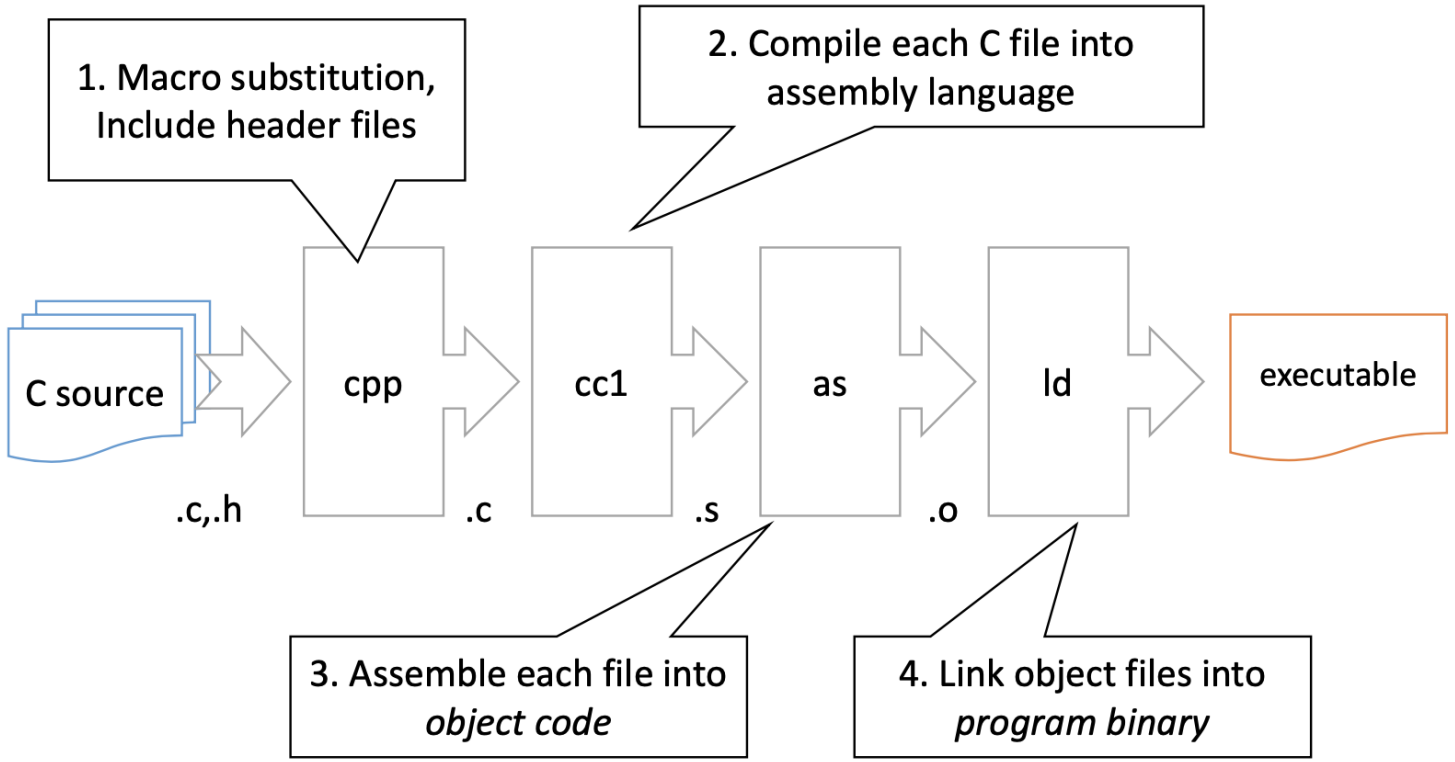
Systems@ETH Zürich

Makefile-demo (splitting .c, .h)

Absolute Basics: Compiling: source code, assembly files, object files, executables

Systems@ETH Zürich

GNU gcc Toolchain



Absolute Basics: Compiling: source code, assembly files, object files, executables

Systems@ETH Zürich

```c
#include <stdio.h>
#include "functions.h"

int main(int argc, char** argv){
  printf("hello, world\n");
  printf("square of 3: %d\n", square(3));
  return 0;
}
```

```
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 15, 0     sdk_version 15, 0
        .globl  _main                           ## -- Begin function main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movl    %edi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    L_.str(%rip), %rdi
        movb    $0, %al
        callq   _printf
        movl    $3, %edi
        callq   _square
        movl    %eax, %esi
        leaq    L_.str.1(%rip), %rdi
        movb    $0, %al
        callq   _printf
        xorl    %eax, %eax
        addq    $16, %rsp
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "hello, world\n"

L_.str.1:                               ## @.str.1
        .asciz  "square of 3: %d\n"

.subsections_via_symbols
```

```
student-net-hg-1584:make-demo bene
00000000: 11001111 11111010 111011
00000006: 00000000 00000001 000000
0000000c: 00000001 00000000 000000
00000012: 00000000 00000000 000010
00000018: 00000000 00100000 000000
0000001e: 00000000 00000000 000110
00000024: 10001000 00000001 000000
0000002a: 00000000 00000000 000000
00000030: 00000000 00000000 000000
00000036: 00000000 00000000 000000
0000003c: 00000000 00000000 000000
00000042: 00000000 00000000 000000
00000048: 00101000 00000010 000000
0000004e: 00000000 00000000 110010
00000054: 00000000 00000000 000000
0000005a: 00000000 00000000 000001
00000060: 00000100 00000000 000000
00000066: 00000000 00000000 010111
0000006c: 01111000 01110100 000000
00000072: 00000000 00000000 000000
00000078: 01011111 01011111 010101
0000007e: 00000000 00000000 000000
00000084: 00000000 00000000 000000
0000008a: 00000000 00000000 000000
00000090: 01000110 00000000 000000
00000096: 00000000 00000000 001010
0000009c: 00000100 00000000 000000
000000a2: 00000000 00000000 000001
000000a8: 00000000 00000100 000000
000000ae: 00000000 00000000 000000
000000b4: 00000000 00000000 000000
```

Makefiles

Systems@ETH Zürich

- **Purpose**: automate compiling process, s.t. only parts of the program that have changed get recompiled
- ”make” reads its instruction from Makefile (called **descriptor file**) by default: specifies set of rules to determine which part of the program needs to be recompiled

Command Line Approch to Compile

Systems@ETH Zürich

- gcc –c main.c functions.c
- ls *.o
  main.o functions.o
- gcc –o myprogram main.o functions.o
- ./myprogram
- **If we want to modify (add functions**): need to recompile only certain files (or everything to be sure: that's slow)
- Gcc –c functions.c
- Gcc –o myprogram main.o functions.o

General Makefile structure

Systems@ETH Zürich

- **Target**: Outputfile we want to create (executable or object file)
- **Dependencies**: Fields that are required to create the target (source files)
- **Command:** Shell commands to run (e.g. gcc to compile)

```
31    target: dependencies
32        command
33
34
```

Example of a Makefile

- **Rule (executable)**: Determines how to create TARGET executable from object files (main.o, functions.o)
- Rule **.c->.o:** Rule tells make how to compile .c to .o files: **$<** special variable that represents the dependencies (specified after target: **dependencies**)
- In Makefiles: % wild card not * like in Bash (ls *.c)

```
# Makefile to compile a simple C program

# Compiler to use
CC = gcc

# Compiler flags
CFLAGS = -Wall -g

# Target executable
TARGET = myprogram

# Source files
SRCS = main.c functions.c

# Object files (compiled versions of .c files)
OBJS = main.o functions.o

# Rule to create the final executable (TARGET) from object files (OBJS)
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Rule to create .o files from .c files
%.o: %.c
    $(CC) $(CFLAGS) -c $<

# Clean up compiled files
clean:
    rm -f $(OBJS) $(TARGET)

```

Example of a Makefile

- **Phony targets**: phony target is one that is not really the name of a file: will only have a list of commands, **no dependencies**
- **Cleaning up** (phony target): tells make to delete object files and the executable

```
# Makefile to compile a simple C program

# Compiler to use
CC = gcc

# Compiler flags
CFLAGS = -Wall -g

# Target executable
TARGET = myprogram

# Source files
SRCS = main.c functions.c

# Object files (compiled versions of .c files)
OBJS = main.o functions.o

# Rule to create the final executable (TARGET) from object files (OBJS)
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Rule to create .o files from .c files
%.o: %.c
    $(CC) $(CFLAGS) -c $<

# Clean up compiled files
clean:
    rm -f $(OBJS) $(TARGET)

```

Makefiles

Systems@ETH Zürich

- When running “make” without specifying a target (unlike **make clean**) it will specify **the first rule** in the Makefile => that’s why we should have that the first rule builds the main target (executable)
- **Dependency resolution**: make figures out what files need to be updated: for the executable myprogram, we need main.o, functions.o => if they are not up to date because we updated functions.c for instance, it will **recursively** look for rules to build those dependencies (%.o: %.c here)

Systems@ETH Zürich

Makefile-demo (makefile)

make ?

- GNU make:
  - *“In software development, **Make** is a utility that automatically builds executable programs and libraries from source code by reading files called **makefiles** which specify how to derive the target program.” - https://en.wikipedia.org/wiki/Make_(software)*
  - *Only builds the parts if they are modified and necessary w.r.t. the makefiles.*
  - *https://makefiletutorial.com/*

Example Makefile (from assignment 1)

```
CC = gcc
CFLAGS = -O -Wall

btest: btest.c bits.c decl.c tests.c btest.h bits.h
	$(CC) $(CFLAGS) -o btest bits.c btest.c decl.c tests.c

clean:
	rm -f *.o btest
```

Usage:

`make` or `make btest`:	runs the compilation but only if the files are modified
`make clean`:	removes your generated binary file

Some hints

- Function Pointers
  http://www.cprogramming.com/tutorial/function-pointers.html

- Pointer Tutorial
  http://www.cplusplus.com/doc/tutorial/pointers/

- More on modules and header files
  - http://www.tutorialspoint.com/cprogramming/c_header_files.htm
- Make files (important for later…)
  - http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- More on this in the lecture next week… ☺

Demo

The compiler is your friend!

GCC Flags for better coding style

- -Werror
  - Make all warnings into errors.
- -Wpedantic
  - Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions
- -Wall
  - Enables a number of warnings about questionable code
- -Wextra
  - This enables some extra warning flags that are not enabled by -Wall (such as -Wuninitialized)

https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

GCC Flags for catching errors at runtime

- -fsanitize=address
  - Instrument code to detect memory errors
- -fsanitize=undefined
  - Instrument code to detect undefined behavior at runtime
- -fstack-protector-all
  - Instruments code to detect buffer overflows on the stack

https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Gcc-demo (flags)

Systems@ETH Zürich

Exercise

Let’s match some C expressions.

Why should you care? Old **exam** question

Systems@ETH Zürich

Question 5

[12 points] ☐

In the following question assume:

- `a` and `b` are declared as `int` in C.
- The machine uses 32-bit two's complement format for signed `int`s.
- `MAX_INT` and `MIN_INT` are the maximum and minimum representable signed integer values respectively
- `W` is one less than the number of bits needed to represent an `int` (i.e. `W == 31`).

For each of the descriptions in the left column, write in the letter of the corresponding expression in the right column.

(12 points)

1. `a` ☐
2. `(a < 0) ? 1 : -3` ☐
3. One's complement of `a` ☐
4. `a & b` ☐

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a << 3) + ~a`

d. `(a << 4) + (a << 2) + (a << 1)`

e. `((a < 0) ? (a + 3) : a) >> 2`

Recall

Shifting:

Signed mult: $x \ll k = x \cdot 2^k$

Signed Div: $s \gg k = \frac{s}{2^k}$, $s \geq 0$

$$(s + (2^k - 1)) \gg k = \frac{s}{2^k}, \; s < 0$$

$$= (s + ((1 \ll k) - 1)) \gg k$$

Numbers (signed): 4 bit

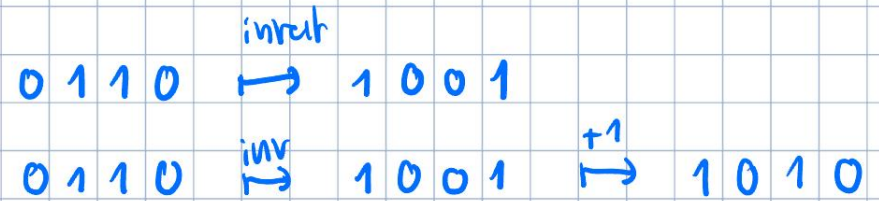
TMin: 1 0 0 0

TMax: 0 1 1 1

Complements

1s: Invert all bits

2s: 1s + 1



Exercise: Matching expressions

a

Assumptions

- a and b are declared as int in C.
- The machine uses 32-bit two's complement format for signed ints.
- `MAX_INT` and `MIN_INT` are the maximum and minimum representable signed integer values, respectively.
- `W` is one less than the number of bits needed to represent an int (i.e., `W == 31`).

Answers

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a « 3) + ~a`

d. `(a « 4) + (a « 2) + (a « 1)`

e. `((a < 0) ? (a + 3) : a) » 2`

f. `a ^ (MIN_INT + MAX_INT)`

g. `~((a | (~a + 1)) » W) & 1`

h. `~((a » W) « 1)`

i. `a » 2`

Systems Programming and Computer Architecture Exam, Autumn Term 2022, Question 15

Justification

Justification

Systems@ETH Zürich

Ex1. $a \wedge b \equiv (a \wedge \neg b) \vee (\neg a \wedge b)$

Total: $((a \wedge \neg b) \vee (\neg a \wedge b)) \wedge \neg b) \vee (\neg((a \wedge \neg b) \vee (\neg a \wedge b)) \wedge b)$

$\equiv (a \wedge \neg b \wedge \neg b) \vee (\neg a \wedge \underbrace{b \wedge \neg b}_{\bot}) \vee (((\neg a \vee b) \wedge (a \vee \neg b)) \wedge b)$

$\equiv (a \wedge \neg b) \vee ((\neg a \vee b) \wedge (a \vee \neg b) \wedge b)$

$\equiv (a \wedge \neg b) \vee (\underbrace{(\neg a \wedge b) \vee (b \wedge b)}_{\equiv b} \wedge \underbrace{((a \wedge b) \vee (\neg b \wedge b))}_{\equiv \top})$

$\equiv (a \wedge \neg b) \vee (b \wedge (a \wedge b))$

$\equiv (a \wedge \neg b) \vee (a \wedge b)$

$\equiv a$

Exercise: Matching expressions

```
a * 7
```

Assumptions

- a and b are declared as int in C.
- The machine uses 32-bit two's complement format for signed ints.
- MAX_INT and MIN_INT are the maximum and minimum representable signed integer values, respectively.
- W is one less than the number of bits needed to represent an int (i.e., W == 31).

Answers

```
a. ~(~a | (b ^ (MIN_INT + MAX_INT)))
b. ((a ^ b) & ~b) | (~(a ^ b) & b)
c. 1 + (a « 3) + ~a
d. (a « 4) + (a « 2) + (a « 1)
e. ((a < 0) ? (a + 3) : a) » 2
f. a ^ (MIN_INT + MAX_INT)
g. ~((a | (~a + 1)) » W) & 1
h. ~((a » W) « 1)
i. a » 2
```

Systems Programming and Computer Architecture Exam, Autumn Term 2022, Question 15

Justification

Justification

Systems@ETH Zürich

Ex 2: $a \cdot 7$

$$1 + (a << 3) + \sim a$$

$$= (\sim a + 1) + 2^3 \cdot a$$

$$= -a + 8a = 7a$$

Exercise: Matching expressions

Systems@ETH Zürich

One’s complement of a

Assumptions

- a and b are declared as int in C.
- The machine uses 32-bit two’s complement format for signed ints.
- MAX_INT and MIN_INT are the maximum and minimum representable signed integer values, respectively.
- W is one less than the number of bits needed to represent an int (i.e., W == 31).

Answers

```
a. ~(~a | (b ^ (MIN_INT + MAX_INT)))
b. ((a ^ b) & ~b) | (~(a ^ b) & b)
c. 1 + (a « 3) + ~a
d. (a « 4) + (a « 2) + (a « 1)
e. ((a < 0) ? (a + 3) : a) » 2
f. a ^ (MIN_INT + MAX_INT)
g. ~((a | (~a + 1)) » W) & 1
h. ~((a » W) « 1)
i. a » 2
```

Systems Programming and Computer Architecture Exam, Autumn Term 2022, Question 15

Justification

Justification

Systems@ETH Zürich

Ex3: 1s compl

MinInt: 1 0 0 0

MaxInt: 0 1 1 1

\+ 1 1 1 1

⇒ xoring w/ 1 invals all bits (1s to 0s, 0s to 1s)

e.g. a = 1 0 1 1

```
  1 0 1 1
^ 1 1 1 1
-------
  0 1 0 0
```

Exercise: Matching expressions

```
a / 4
```

Assumptions

- a and b are declared as int in C.
- The machine uses 32-bit two's complement format for signed ints.
- MAX_INT and MIN_INT are the maximum and minimum representable signed integer values, respectively.
- W is one less than the number of bits needed to represent an int (i.e., W == 31).

Answers

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a « 3) + ~a`

d. `(a « 4) + (a « 2) + (a « 1)`

e. `((a < 0) ? (a + 3) : a) » 2`

f. `a ^ (MIN_INT + MAX_INT)`

g. `~((a | (~a + 1)) » W) & 1`

h. `~((a » W) « 1)`

i. `a » 2`

Systems Programming and Computer Architecture Exam, Autumn Term 2022, Question 15

Justification

Justification

Ex 4 : a/4 SIGNED INTs!!

Pos shift: a >> 2 — the same

Neg shift (round → 0): $(a + (2^k - 1)) >> k$

$\overset{k=2}{\Rightarrow}$ $(a + (4-1)) >> 2$

if true if false

⇒ written in one line: (bool) ? (statement 1) : (statement 2)

⇒ $((a<0)\ ?\ (a+3) : (a)) >> 2$

Exercise: Matching expressions

```
a & b
```

Assumptions

- a and b are declared as int in C.
- The machine uses 32-bit two's complement format for signed ints.
- `MAX_INT` and `MIN_INT` are the maximum and minimum representable signed integer values, respectively.
- `W` is one less than the number of bits needed to represent an int (i.e., `W == 31`).

Answers

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a « 3) + ~a`

d. `(a « 4) + (a « 2) + (a « 1)`

e. `((a < 0) ? (a + 3) : a) » 2`

f. `a ^ (MIN_INT + MAX_INT)`

g. `~((a | (~a + 1)) » W) & 1`

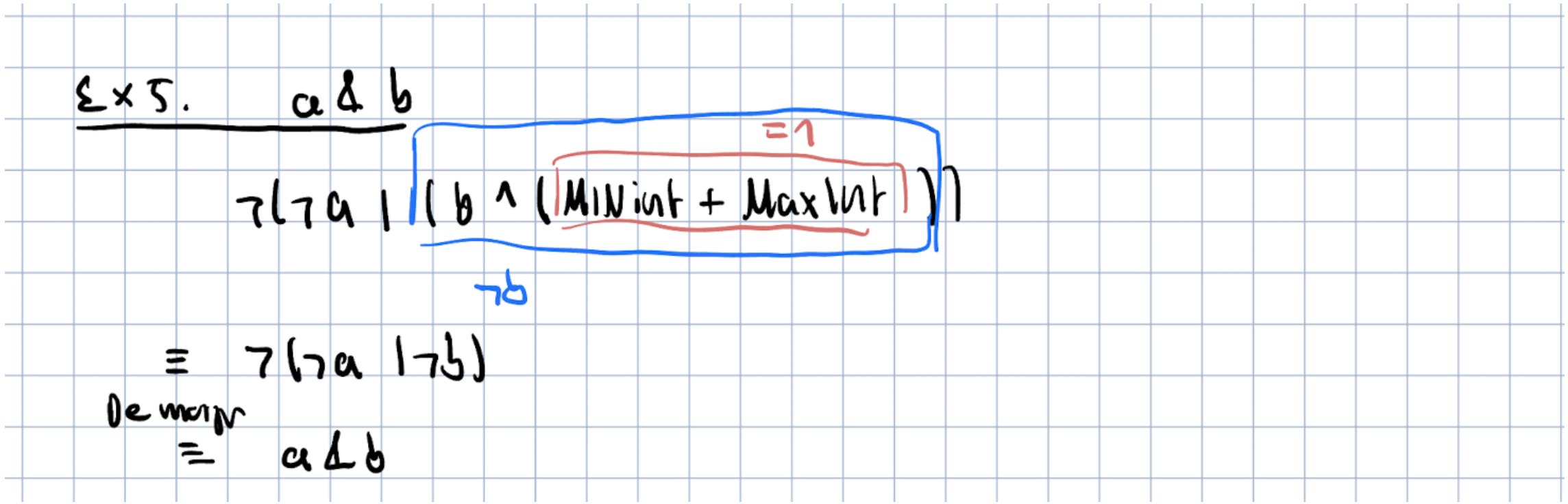
h. `~((a » W) « 1)`

i. `a » 2`

Systems Programming and Computer Architecture Exam, Autumn Term 2022, Question 15

Justification

Justification



Exercise: Matching expressions

```
(a < 0) ? 1 : -1
```

Assumptions

- a and b are declared as int in C.
- The machine uses 32-bit two's complement format for signed ints.
- MAX_INT and MIN_INT are the maximum and minimum representable signed integer values, respectively.
- W is one less than the number of bits needed to represent an int (i.e., W == 31).

Answers

a. `~(~a | (b ^ (MIN_INT + MAX_INT)))`

b. `((a ^ b) & ~b) | (~(a ^ b) & b)`

c. `1 + (a « 3) + ~a`

d. `(a « 4) + (a « 2) + (a « 1)`

e. `((a < 0) ? (a + 3) : a) » 2`

f. `a ^ (MIN_INT + MAX_INT)`

g. `~((a | (~a + 1)) » W) & 1`

h. `~((a » W) « 1)`

i. `a » 2`

Systems Programming and Computer Architecture Exam, Autumn Term 2022, Question 15

Justification

Justification

Systems@ETH Zürich

Ex. 6 $(a<0) ? 1 : -1$

Recall: $1 = 0\,0\,0\,1$

$-1 = 1\,1\,1\,1$

Signed shift: extends msb

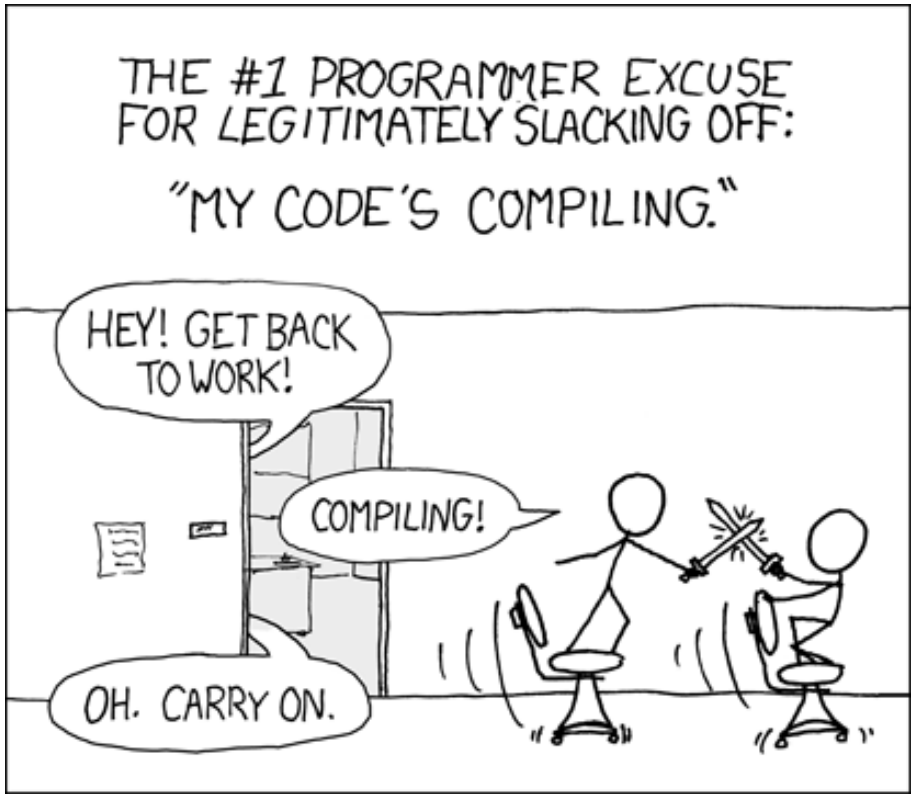
$1\,0\,1\,0 >> 2 \equiv 1\,1\,1\,0$

$0\,1\,1\,0 >> 2 \equiv 0\,0\,0\,1$

$a >> W$:

neg: $1\,1\,1\,1$ $<<1$ $\equiv 1\,1\,1\,0$ $\Rightarrow$ (1s compl) $0\,0\,0\,1$

pos: $0\,0\,0\,0$ $<<1$ $\equiv 0\,0\,0\,0$ $\Rightarrow$ (1s compl) $1\,1\,1\,1$



Good luck and have fun!