CS 110
Computer Architecture
Intro to C I

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

Course Info

- HW1 due Feb. 16th!
- Team (Lab & project) partners are required to be within the same lab session! Decide before Feb. 11th!
- Labs & Projects must be done in a group of two students. Please let the TA know immediately if you cannot find a partner. It is not allowed to change your lab-mate after this week.
- Lab 1 is available and in this week's Lab session
- Lab & Discussion starts this week.
- https://piazza.com/shanghaitech.edu.cn/spring2023/cs110 (access code: **uutib6ruvql**)

Review

- Moore’s Law; Amdahl’s Law; Dennard scaling
- Binary system: Integer (Unsigned & Signed integer)
    - We use “0x” as the prefix of hexadecimal number
- We use 2’s complement to represent signed integer in modern computer: easy arithmetic (addition & subtraction)
- C: portable (GPU/CUDA, DSP, MCU, 寒武纪MLU/Bang C, etc.) and efficient (used in building UNIX/MATLAB/python, etc.); utilized to understand how computer works in this course.
- Compiler first step: C pre-processing (text editing for further compiling steps)

Function-Like Macro

- #define MAG(x, y) (sqrt((x)*(x) + (y)*(y)))

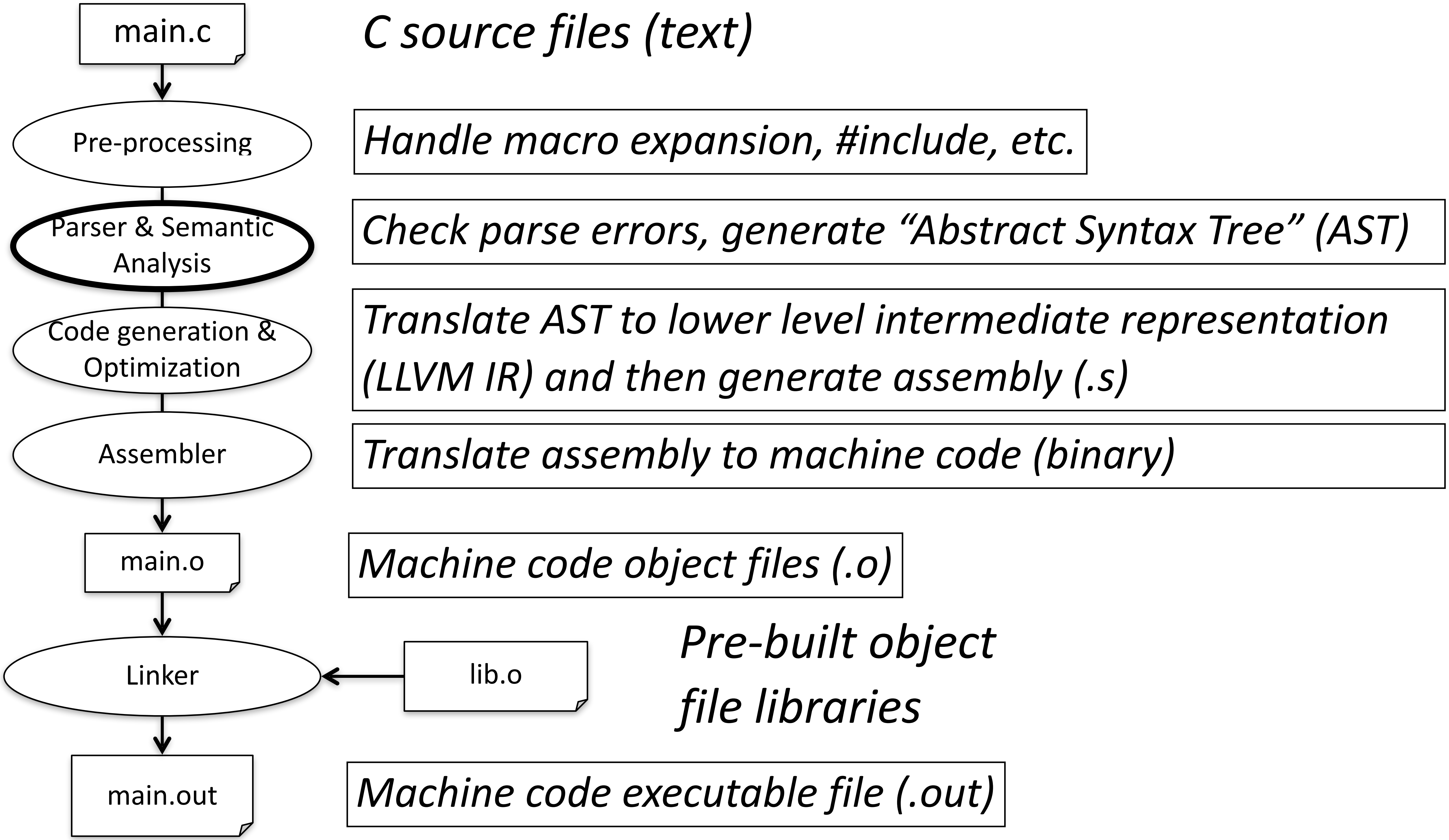
```
%clang/gcc -E introC_1_0.c
```

```
#include <stdio.h>
#include <math.h>
#define MAG0(x, y) sqrt(x*x + y*y)
#define MAG(x, y) (sqrt((x)*(x) + (y)*(y)))
#define MAG2(x,y) ({double a=x; double b=y; sqrt(a*a + b*b);})
#define MSG "Hello \
World!\n"
int main() {
#ifdef MSG
  printf(MSG /* "hi!\n" */);
#endif
    printf("%f\n",MAG(3.0,4.0));
    double i=2, j=3, k0, k1, k2, k3;
    double c=2, d=3;
    k0=MAG0(i+1,j+1);
    k1=MAG(i+1,j+1);
    k2=MAG(++i,++j);
    k3=MAG2(++c,++d);
    printf("%f\n",k0);
    printf("%f\n",k1);
    printf("%f\n",k2);
    printf("%f\n",k3);
  return 0;
}
```

```
kO=sqrt(i+1*i+1 + j+1*j+1);
k1=(sqrt((i+1)*(i+1) + (j+1)*(j+1)));
k2=(sqrt((++i)*(++i) + (++j)*(++j)));
k3=({double a=++c; double b=++d; sqrt(a*a + b*b);});
```

=> Convention: put parenthesis EVERYWHERE!

Outline



C source files (text)

Handle macro expansion, #include, etc.

Check parse errors, generate "Abstract Syntax Tree" (AST)

Translate AST to lower level intermediate representation (LLVM IR) and then generate assembly (.s)

Translate assembly to machine code (binary)

Machine code object files (.o)

Pre-built object file libraries

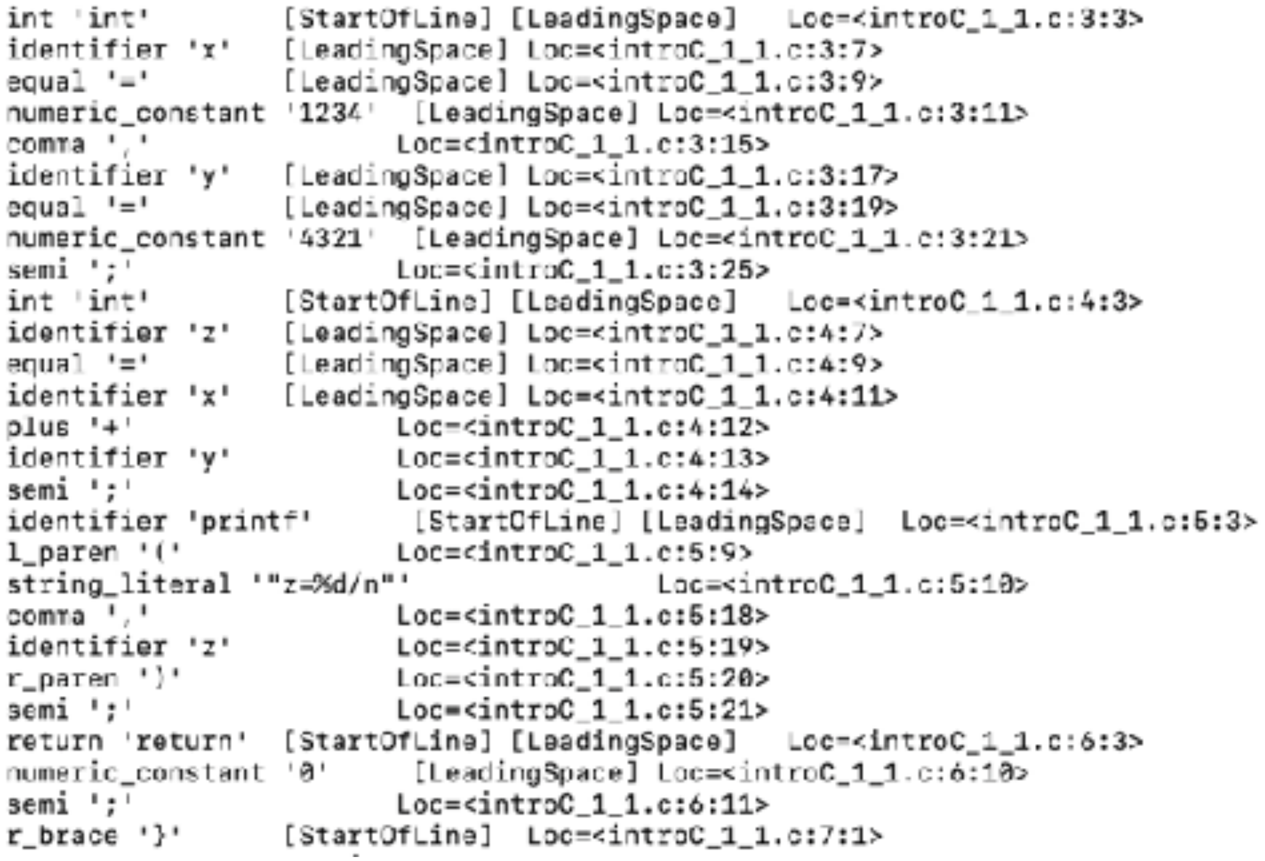
Machine code executable file (.out)

Parser & Semantic Analysis

- Recognize each code word as a “token” (identifiers/symbols, C keywords, constant, comma, semicolon, etc.)
- Record the location of each token

```
%clang -fsyntax-only -Xclang -dump-tokens introC_1_1.c
```

```c
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```



```
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<introC_1_1.c:3:3>
identifier 'x'   [LeadingSpace] Loc=<introC_1_1.c:3:7>
equal '='        [LeadingSpace] Loc=<introC_1_1.c:3:9>
numeric_constant '1234'  [LeadingSpace] Loc=<introC_1_1.c:3:11>
comma ','                Loc=<introC_1_1.c:3:15>
identifier 'y'   [LeadingSpace] Loc=<introC_1_1.c:3:17>
equal '='        [LeadingSpace] Loc=<introC_1_1.c:3:19>
numeric_constant '4321'  [LeadingSpace] Loc=<introC_1_1.c:3:21>
semi ';'                 Loc=<introC_1_1.c:3:25>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<introC_1_1.c:4:3>
identifier 'z'   [LeadingSpace] Loc=<introC_1_1.c:4:7>
equal '='        [LeadingSpace] Loc=<introC_1_1.c:4:9>
identifier 'x'   [LeadingSpace] Loc=<introC_1_1.c:4:11>
plus '+'                 Loc=<introC_1_1.c:4:12>
identifier 'y'           Loc=<introC_1_1.c:4:13>
semi ';'                 Loc=<introC_1_1.c:4:14>
identifier 'printf'      [StartOfLine] [LeadingSpace]  Loc=<introC_1_1.c:5:3>
l_paren '('              Loc=<introC_1_1.c:5:9>
string_literal '"z=%d/n"'               Loc=<introC_1_1.c:5:10>
comma ','                Loc=<introC_1_1.c:5:18>
identifier 'z'           Loc=<introC_1_1.c:5:19>
r_paren ')'              Loc=<introC_1_1.c:5:20>
semi ';'                 Loc=<introC_1_1.c:5:21>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<introC_1_1.c:6:3>
numeric_constant '0'     [LeadingSpace] Loc=<introC_1_1.c:6:10>
semi ';'                 Loc=<introC_1_1.c:6:11>
r_brace '}'      [StartOfLine]  Loc=<introC_1_1.c:7:1>
```

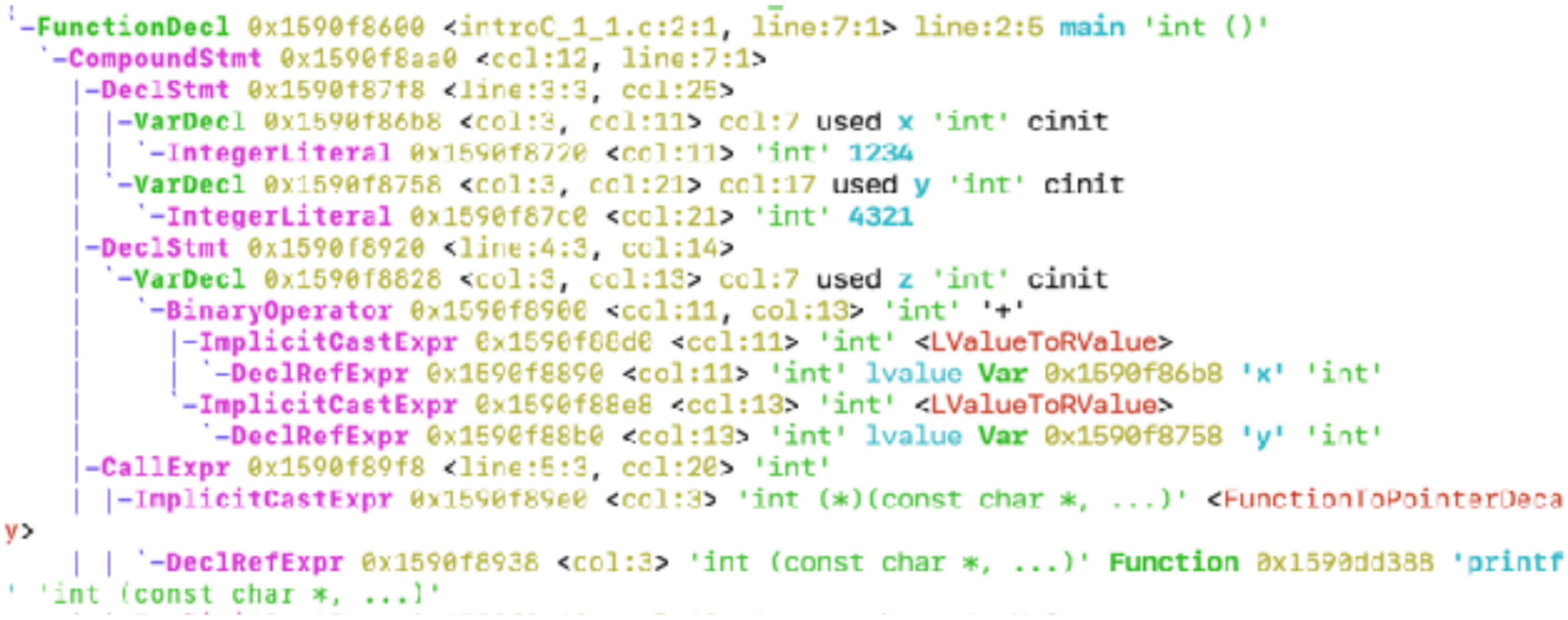
Lexer

Parser & Semantic Analysis

- Organize tokens as “AST” tree
- Report errors

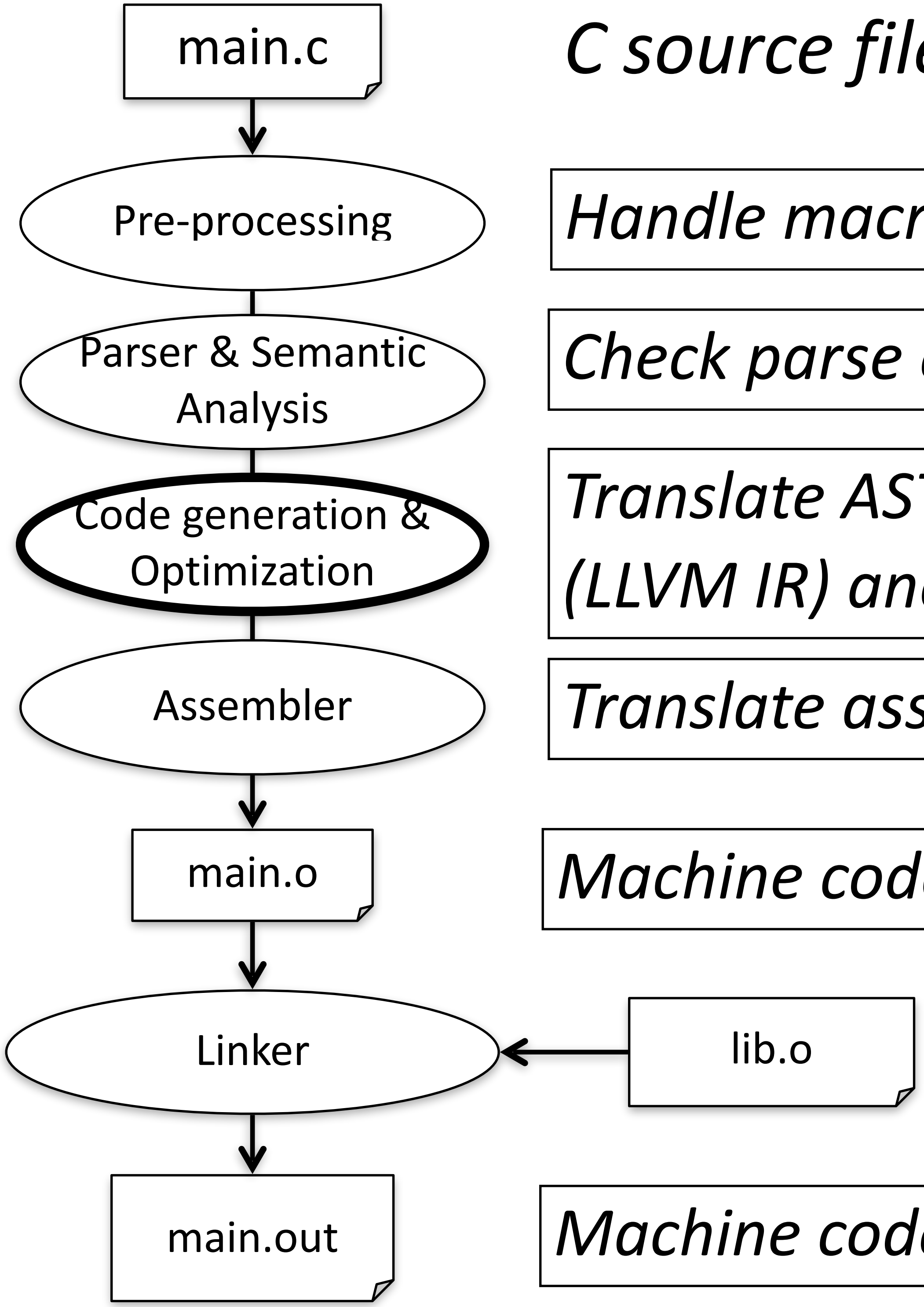
```
% clang -fsyntax-only -Xclang -ast-dump introC_1_1.c
```

```c
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```



```
`-FunctionDecl 0x1590f8600 <introC_1_1.c:2:1, line:7:1> line:2:5 main 'int ()'
  `-CompoundStmt 0x1590f8aa0 <col:12, line:7:1>
    |-DeclStmt 0x1590f87f8 <line:3:3, col:25>
    | |-VarDecl 0x1590f86b8 <col:3, col:11> col:7 used x 'int' cinit
    | | `-IntegerLiteral 0x1590f8720 <col:11> 'int' 1234
    | `-VarDecl 0x1590f8758 <col:3, col:21> col:17 used y 'int' cinit
    |   `-IntegerLiteral 0x1590f87c0 <col:21> 'int' 4321
    |-DeclStmt 0x1590f8920 <line:4:3, col:14>
    | `-VarDecl 0x1590f8828 <col:3, col:13> col:7 used z 'int' cinit
    |   `-BinaryOperator 0x1590f8900 <col:11, col:13> 'int' '+'
    |     |-ImplicitCastExpr 0x1590f88d0 <col:11> 'int' <LValueToRValue>
    |     | `-DeclRefExpr 0x1590f8890 <col:11> 'int' lvalue Var 0x1590f86b8 'x' 'int'
    |     `-ImplicitCastExpr 0x1590f88e8 <col:13> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x1590f88b0 <col:13> 'int' lvalue Var 0x1590f8758 'y' 'int'
    |-CallExpr 0x1590f89f8 <line:5:3, col:20> 'int'
    | |-ImplicitCastExpr 0x1590f89e0 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDeca
y>
    | | `-DeclRefExpr 0x1590f8938 <col:3> 'int (const char *, ...)' Function 0x1590dd388 'printf
' 'int (const char *, ...)'
```

C Compilation Simplified Overview (more later in course)



main.c

Pre-processing

Parser & Semantic Analysis

Code generation & Optimization

Assembler

main.o

Linker

lib.o

main.out

C source files (text)

Handle macro expansion, #include, etc.

Check parse errors, generate “Abstract Syntax Tree” (AST)

Translate AST to lower level intermediate representation (LLVM IR) and then generate assembly (.s)

Translate assembly to machine code (binary)

Machine code object files (.o)

Pre-built object file libraries

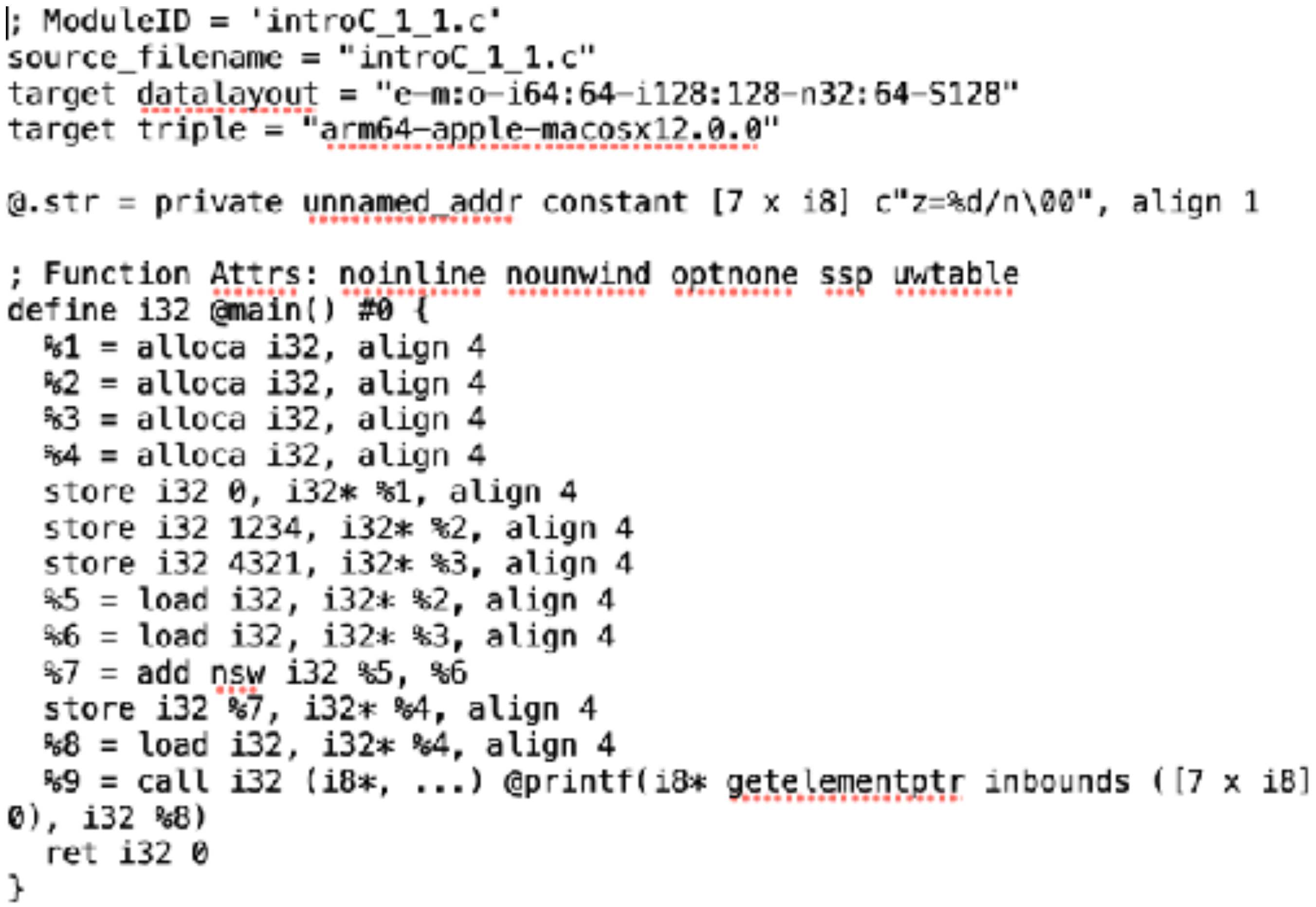
Machine code executable file (.out)

Code Generation & Optimization

- Generate intermediate representation (IR)
  - LLVM IR for clang/LLVM
  - GIMPLE for gcc

```
%clang -S -emit-llvm introC_1_1.c -o introC_1_1.ll
```

```
#include <stdio.h>
int main() {//compute 1234
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```



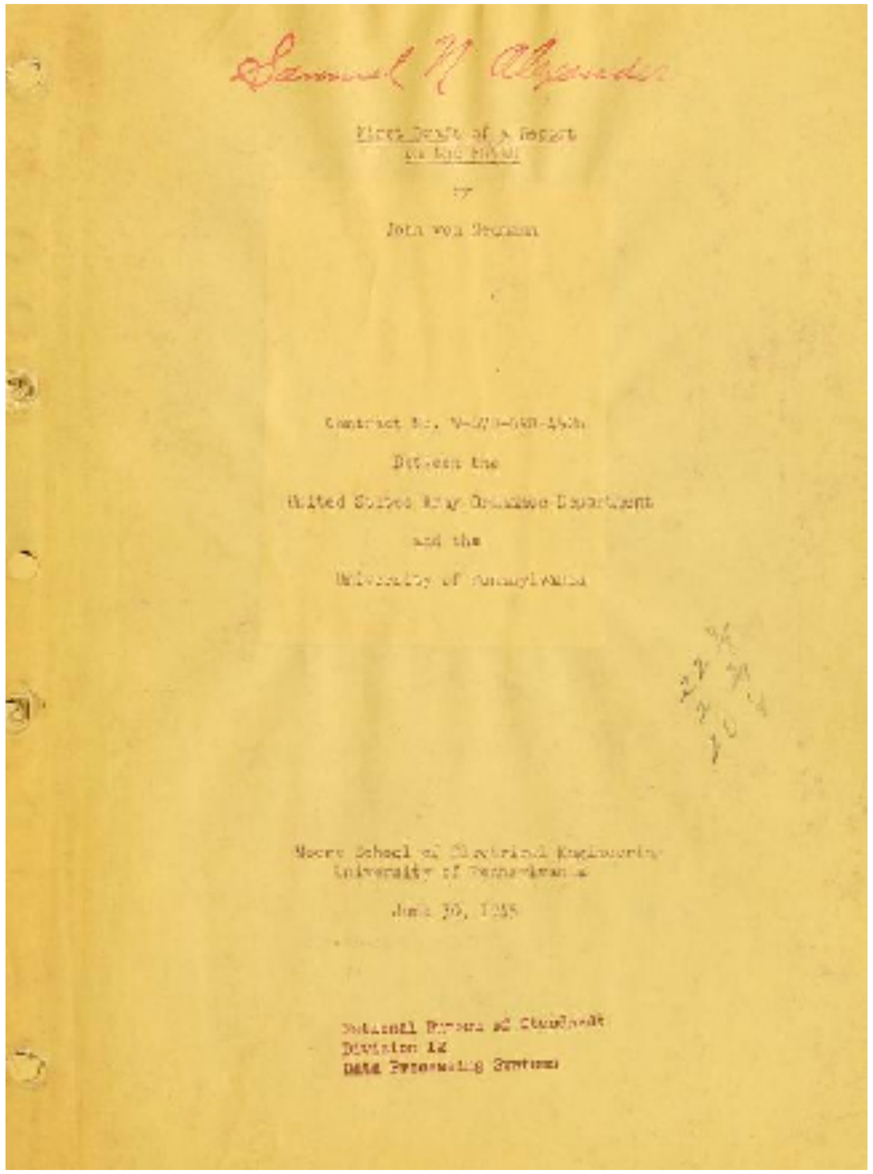
```
; ModuleID = 'introC_1_1.c'
source_filename = "introC_1_1.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx12.0.0"

@.str = private unnamed_addr constant [7 x i8] c"z=%d/n\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1234, i32* %2, align 4
  store i32 4321, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i8]
0), i32 %8)
  ret i32 0
}
```

Components of Computers

- Von Neumann Architecture
  - *First Draft of a Report on the EDVAC*



Samuel N Alexander

First Draft of a Report
on the EDVAC

by

John von Neumann

Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

and the

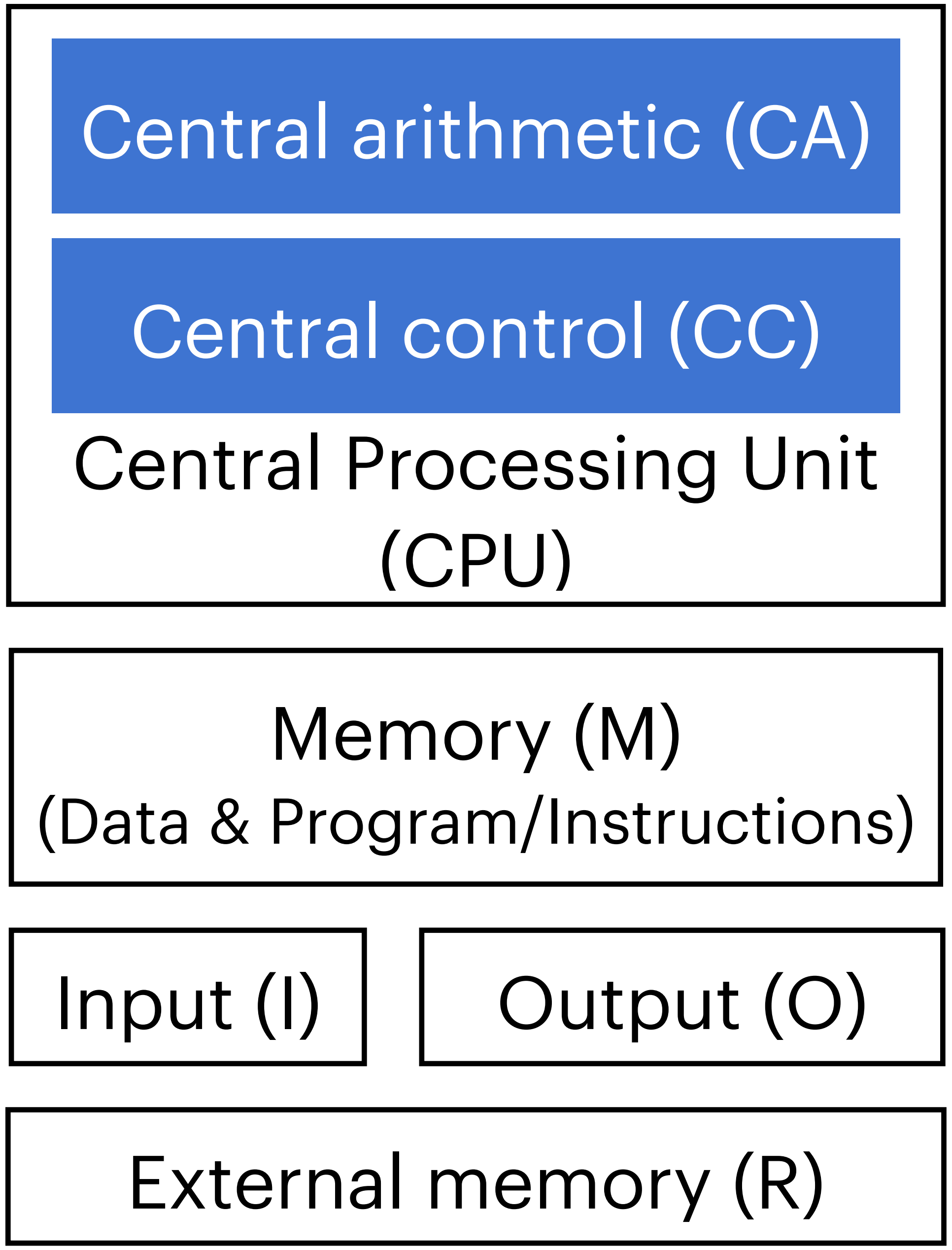
University of Pennsylvania

Moore School of Electrical Engineering
University of Pennsylvania

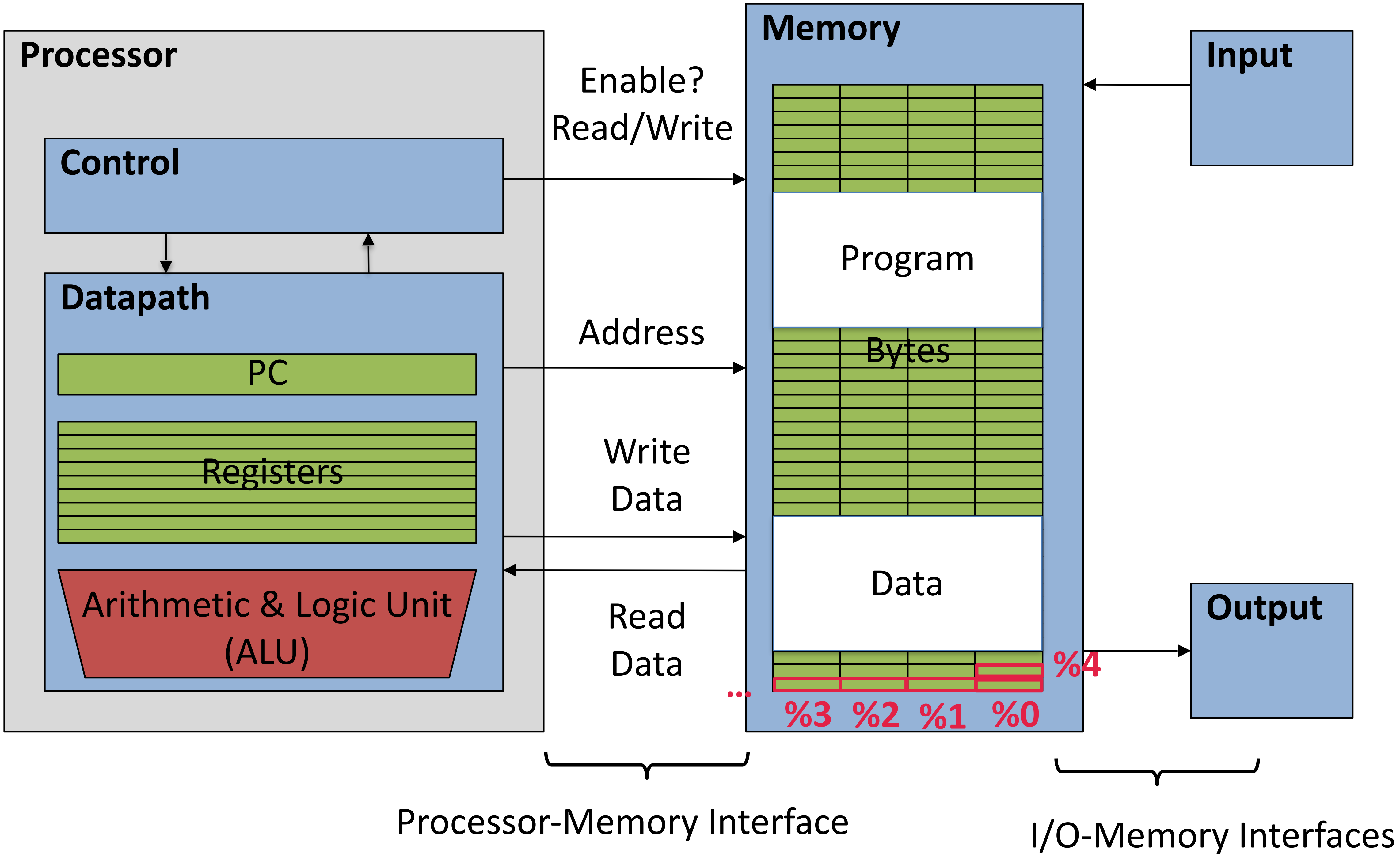
June 30, 1945

National Bureau of Standards
Division 12
Data Processing Systems

By John von Neumann - https://archive.org/stream/firstdraftofrepooovonn#page/n1/mode/2up, Public Domain, https://commons.wikimedia.org/w/index.php?curid=26685284



Components of Computers



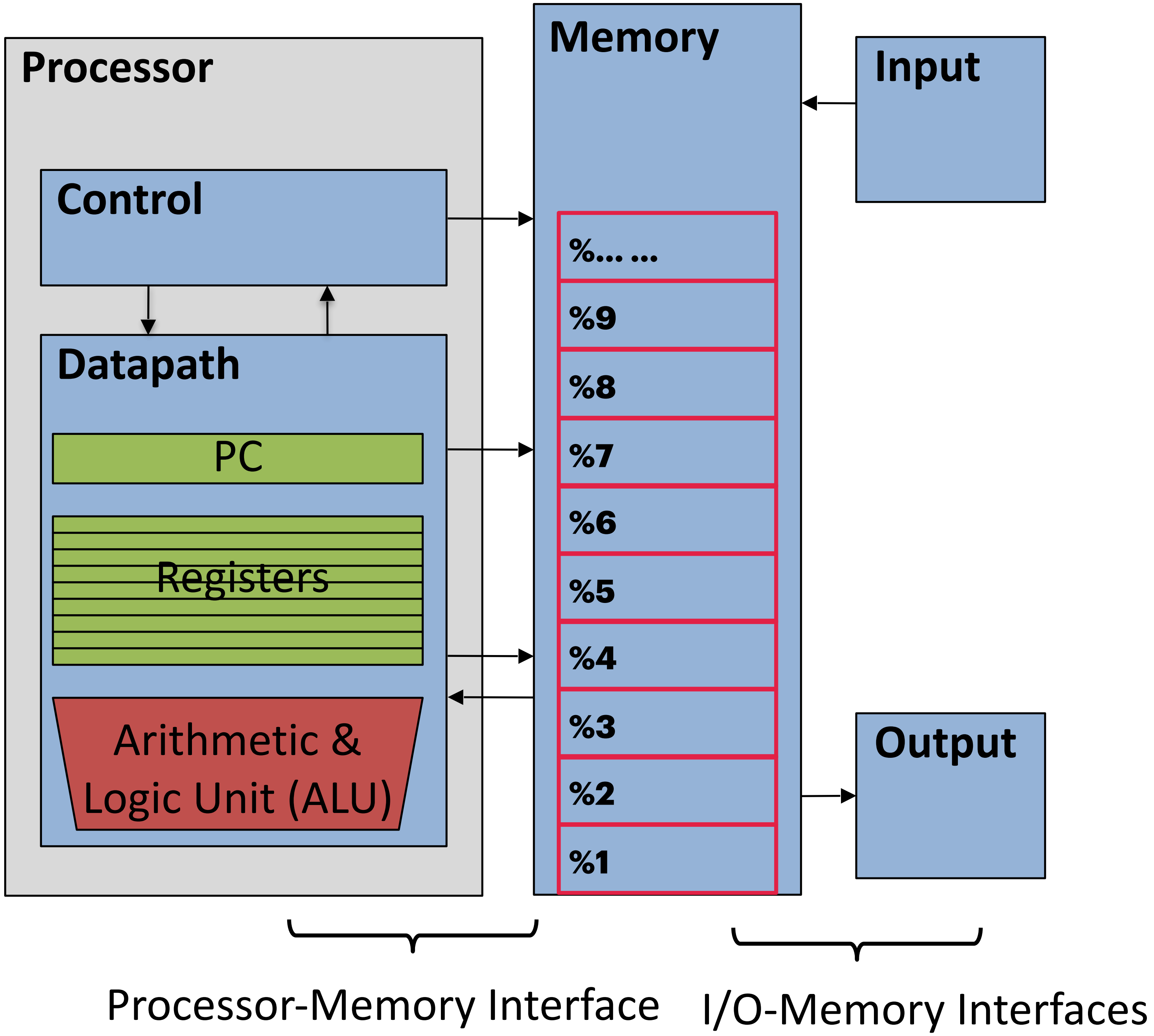
IR Implication on Hardware

```c
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

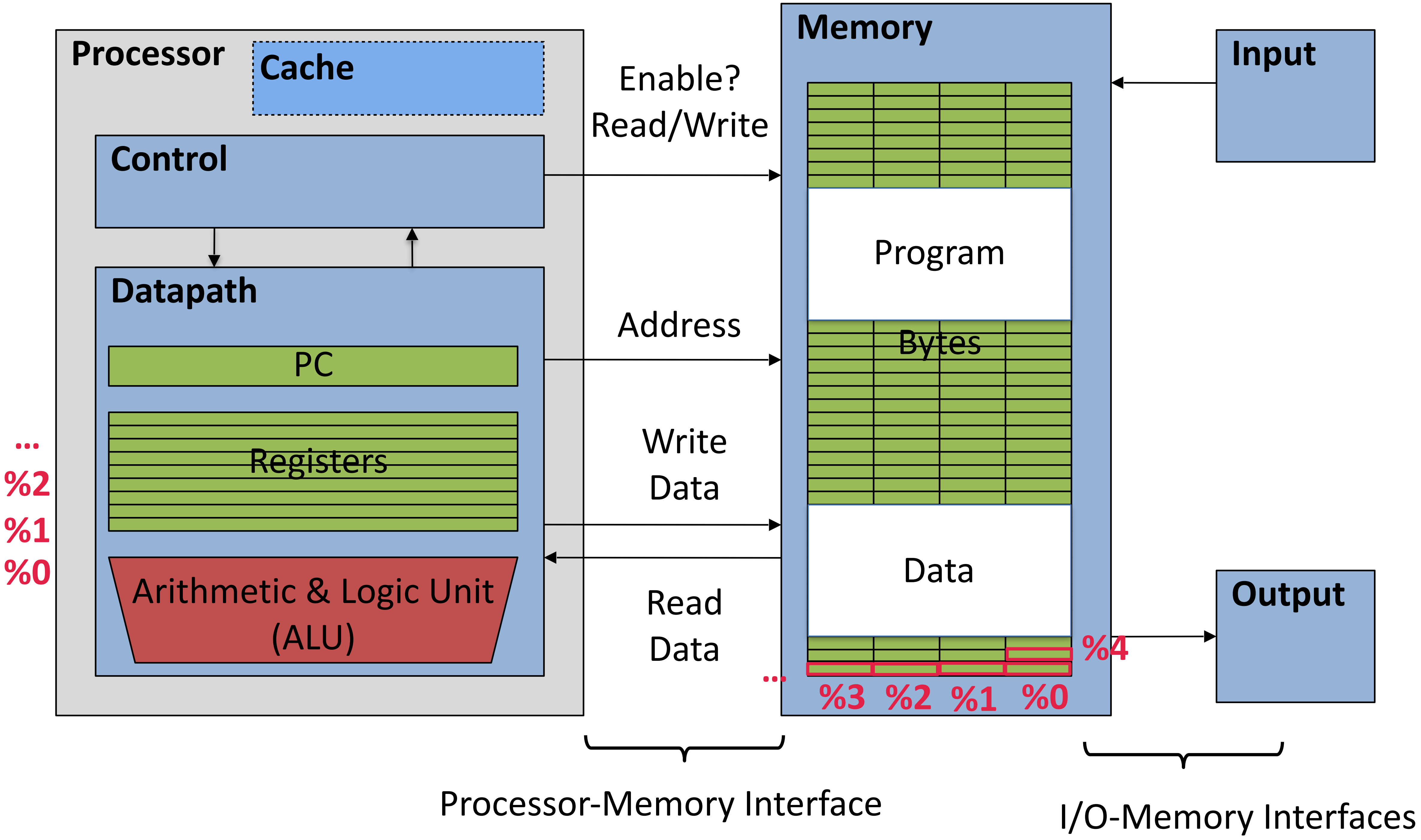
Original code

```
define i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 1234, i32* %2, align 4
 store i32 4321, i32* %3, align 4
 %5 = load i32, i32* %2, align 4
 %6 = load i32, i32* %3, align 4
 %7 = add nsw i32 %5, %6
 store i32 %7, i32* %4, align 4
 %8 = load i32, i32* %4, align 4
 %9 = call i32 (i8*, ...) @printf(i8*
getelementptr inbounds ([7 x i8], [7
x i8]* @.str, i64 0, i64 0), i32 %8)
 ret i32 0
}
```

LLVM IR



Optimization



IR to Assembly

```
% clang -S introC_1_1.c -o introC_1_1.s
```

```
#include <stdio.h>
int main() {//compute 1234 + 4321
  int x = 1234, y = 4321;
  int z = x+y;
  printf("z=%d/n",z);
  return 0;
}
```

Original code

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1234, i32* %2, align 4
  store i32 4321, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %5, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  %9 = call i32 (i8*, ...) @printf(i8*
getelementptr inbounds ([7 x i8], [7 x i8]*
@.str, i64 0, i64 0), i32 %8)
  ret i32 0
}
```

LLVM IR

```
    .section __TEXT,__text,regular,pure_instructions
    .build_version macos, 12, 0   sdk_version 13, 1
    .globl   _main                     ; -- Begin function
main
    .p2align 2
_main:                                 ; @main
    .cfi_startproc
; %bb.0:
    sub  sp, sp, #48
    stp  x29, x30, [sp, #32]          ; 16-byte Folded Spill
    add  x29, sp, #32
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    mov  w8, #0
    str  w8, [sp, #12]                ; 4-byte Folded Spill
    stur wzr, [x29, #-4]
    mov  w8, #1234
    stur w8, [x29, #-8]
    mov  w8, #4321
    stur w8, [x29, #-12]
    ldur w8, [x29, #-8]
    ldur w9, [x29, #-12]
    add  w8, w8, w9
    str  w8, [sp, #16]
    ldr  w9, [sp, #16]
     … …
     add sp, sp, #48
     ret
```

ARM Assembly

(Hardware abstraction)

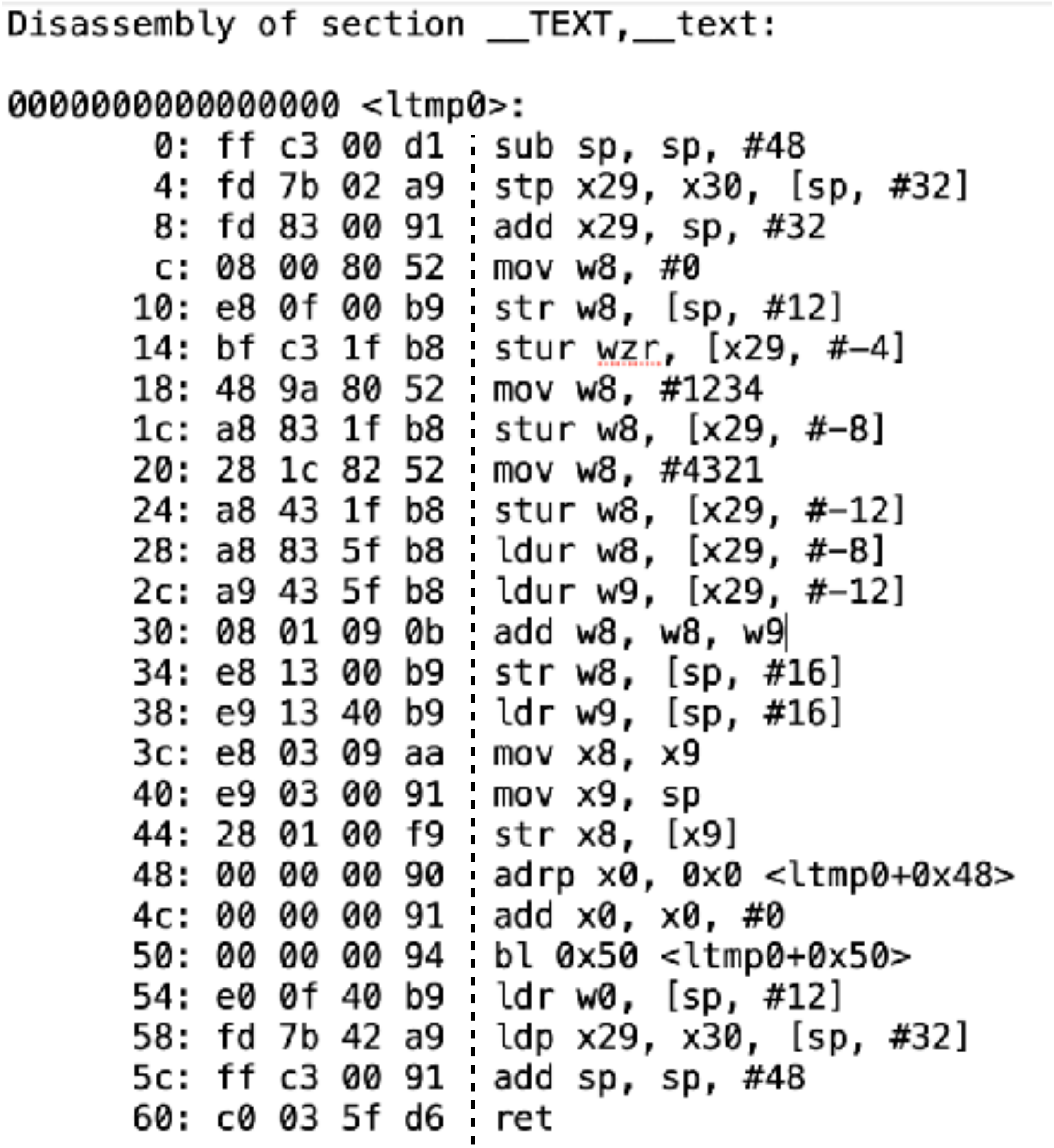
Translated to machine code defined by ISA

IR to Assembly to Machine Code

```
% clang -c introC_1_1.c -o introC_1_1.o
% objdump -d introC_1_1.o
```

Machine Code

(Stored program/ instructions)



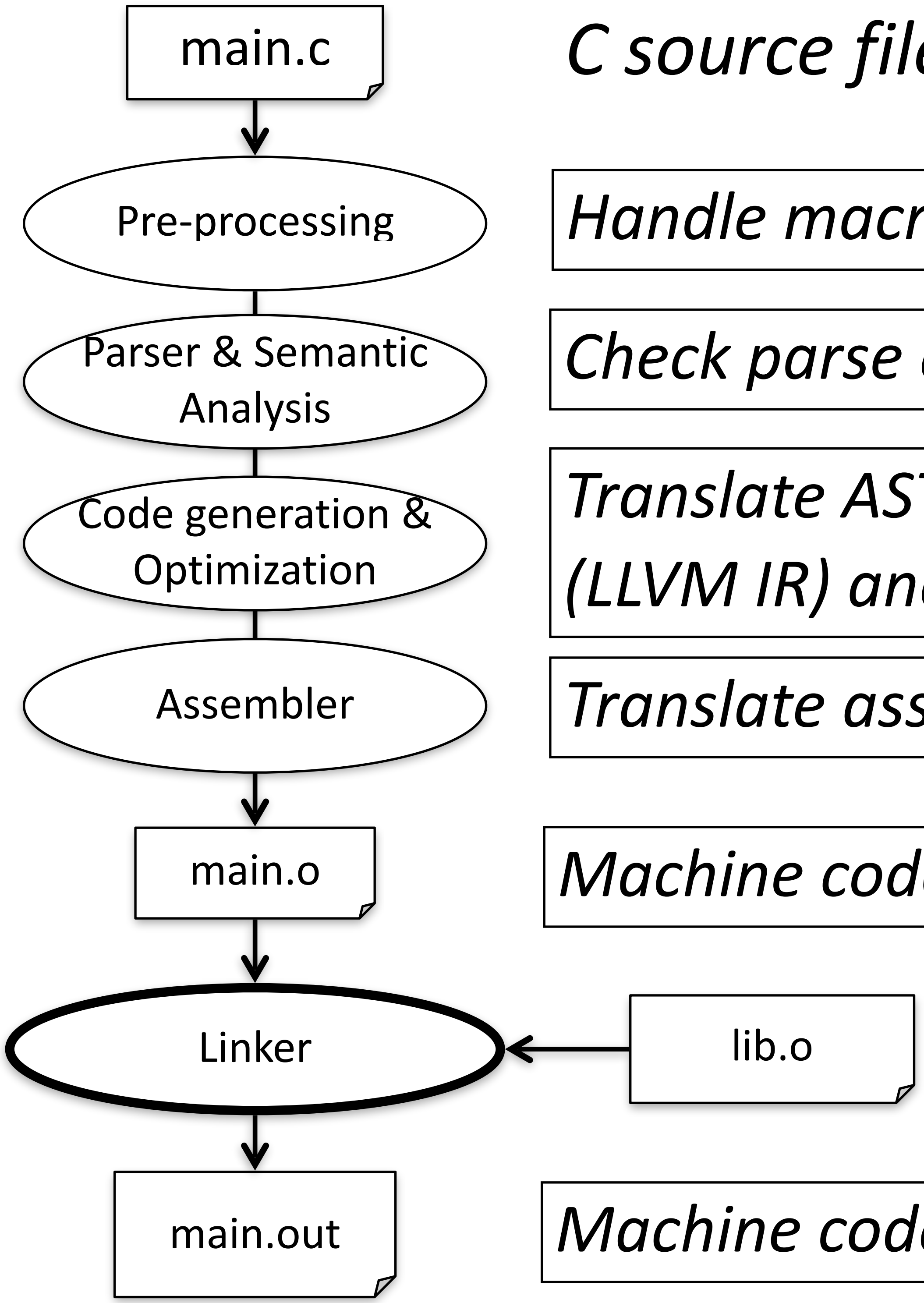
```
Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:
       0: ff c3 00 d1   sub sp, sp, #48
       4: fd 7b 02 a9   stp x29, x30, [sp, #32]
       8: fd 83 00 91   add x29, sp, #32
       c: 08 00 80 52   mov w8, #0
      10: e8 0f 00 b9   str w8, [sp, #12]
      14: bf c3 1f b8   stur wzr, [x29, #-4]
      18: 48 9a 80 52   mov w8, #1234
      1c: a8 83 1f b8   stur w8, [x29, #-8]
      20: 28 1c 82 52   mov w8, #4321
      24: a8 43 1f b8   stur w8, [x29, #-12]
      28: a8 83 5f b8   ldur w8, [x29, #-8]
      2c: a9 43 5f b8   ldur w9, [x29, #-12]
      30: 08 01 09 0b   add w8, w8, w9
      34: e8 13 00 b9   str w8, [sp, #16]
      38: e9 13 40 b9   ldr w9, [sp, #16]
      3c: e8 03 09 aa   mov x8, x9
      40: e9 03 00 91   mov x9, sp
      44: 28 01 00 f9   str x8, [x9]
      48: 00 00 00 90   adrp x0, 0x0 <ltmp0+0x48>
      4c: 00 00 00 91   add x0, x0, #0
      50: 00 00 00 94   bl 0x50 <ltmp0+0x50>
      54: e0 0f 40 b9   ldr w0, [sp, #12]
      58: fd 7b 42 a9   ldp x29, x30, [sp, #32]
      5c: ff c3 00 91   add sp, sp, #48
      60: c0 03 5f d6   ret
```

ARM Assembly

(ARM ISA)

C Compilation Simplified Overview



main.c

Pre-processing

Parser & Semantic Analysis

Code generation & Optimization

Assembler

main.o

Linker

lib.o

main.out

C source files (text)

Handle macro expansion, #include, etc.

Check parse errors, generate “Abstract Syntax Tree” (AST)

Translate AST to lower level intermediate representation (LLVM IR) and then generate assembly (.s)

Translate assembly to machine code (binary, .o)

Machine code object files (.o)

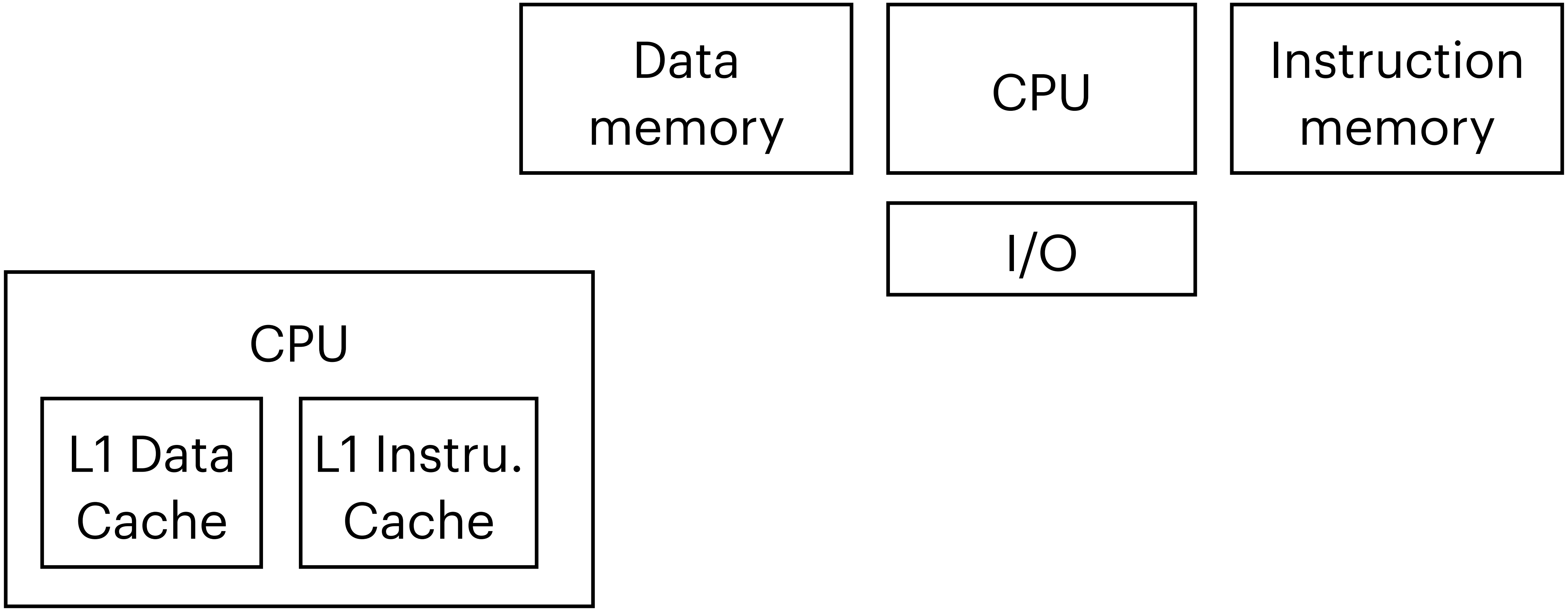
Pre-built object file libraries

Machine code executable file (.out)

Organization of Computers

- Von Neumann Architecture
  - a.k.a. Princeton architecture
  - Uniform memory for data & program/instruction

- Harvard Architecture
  - Separated memory for data & program
  - E.g. MCU, DSP, L1 Cache

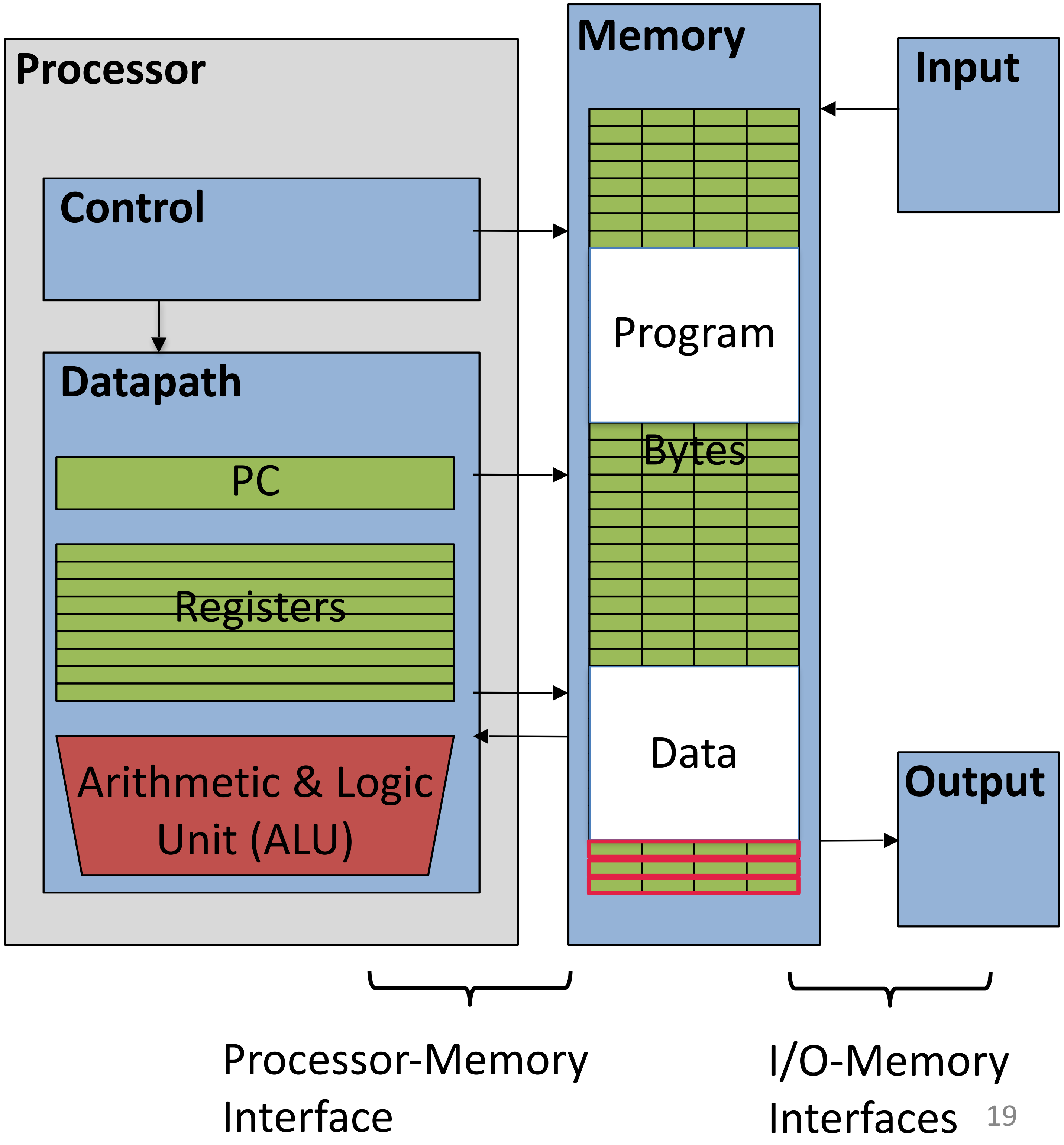


Wrap-it-up

- From C to machine code (clang *.c → *.out & ./*.out)
  - Pre-processing (macro, function-like macro, text editing, #include)
    - Use "()" whenever necessary, or use "function" directly
  - Parser & Semantic Analysis (tokenization & generate AST, basic operations)
  - Translate to IR & optimize (computer components)
  - Translate to assembly and then machine code, executed by hardware (**Covered in future lectures**)
  - Clang manual: https://releases.llvm.org/14.0.0/tools/clang/docs/UsersManual.html
  - GCC: https://gcc.gnu.org/

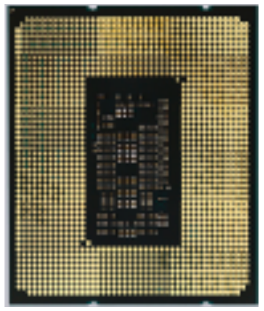
Wrap-it-up

- Von Neumann Architecture
- Harvard Architecture
- Stored-program computer



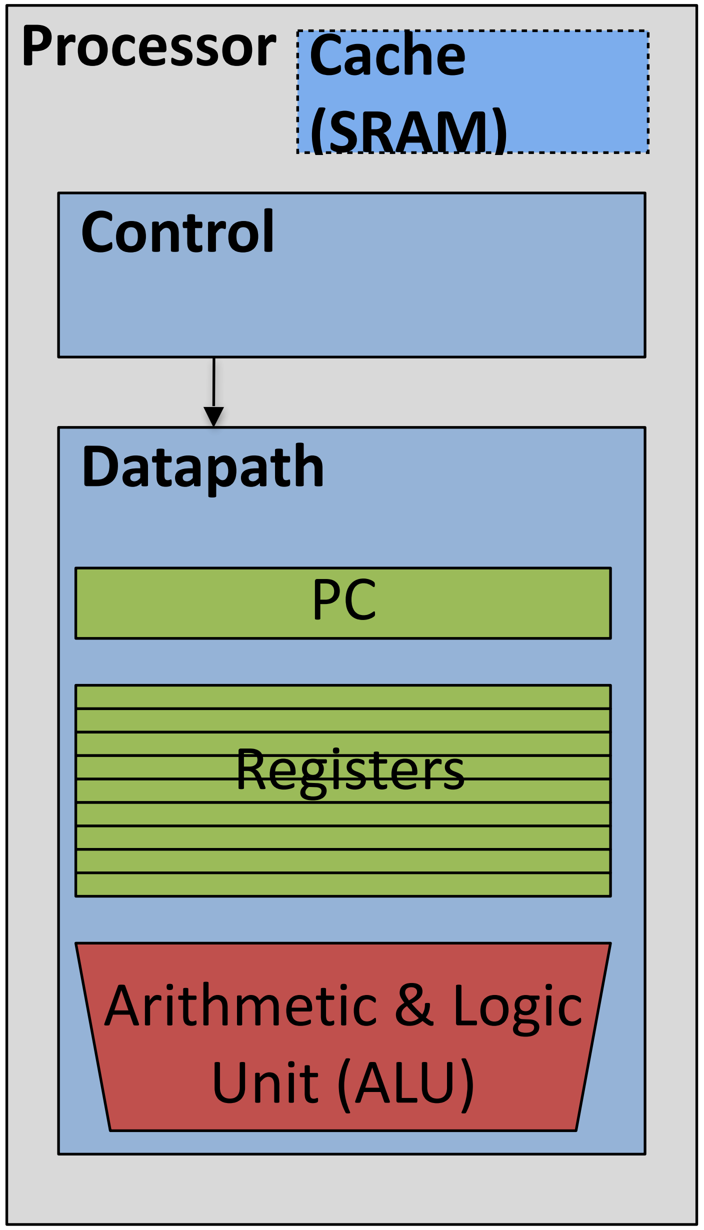
Real Stuff

Intel i7 12700   4.90 GHz

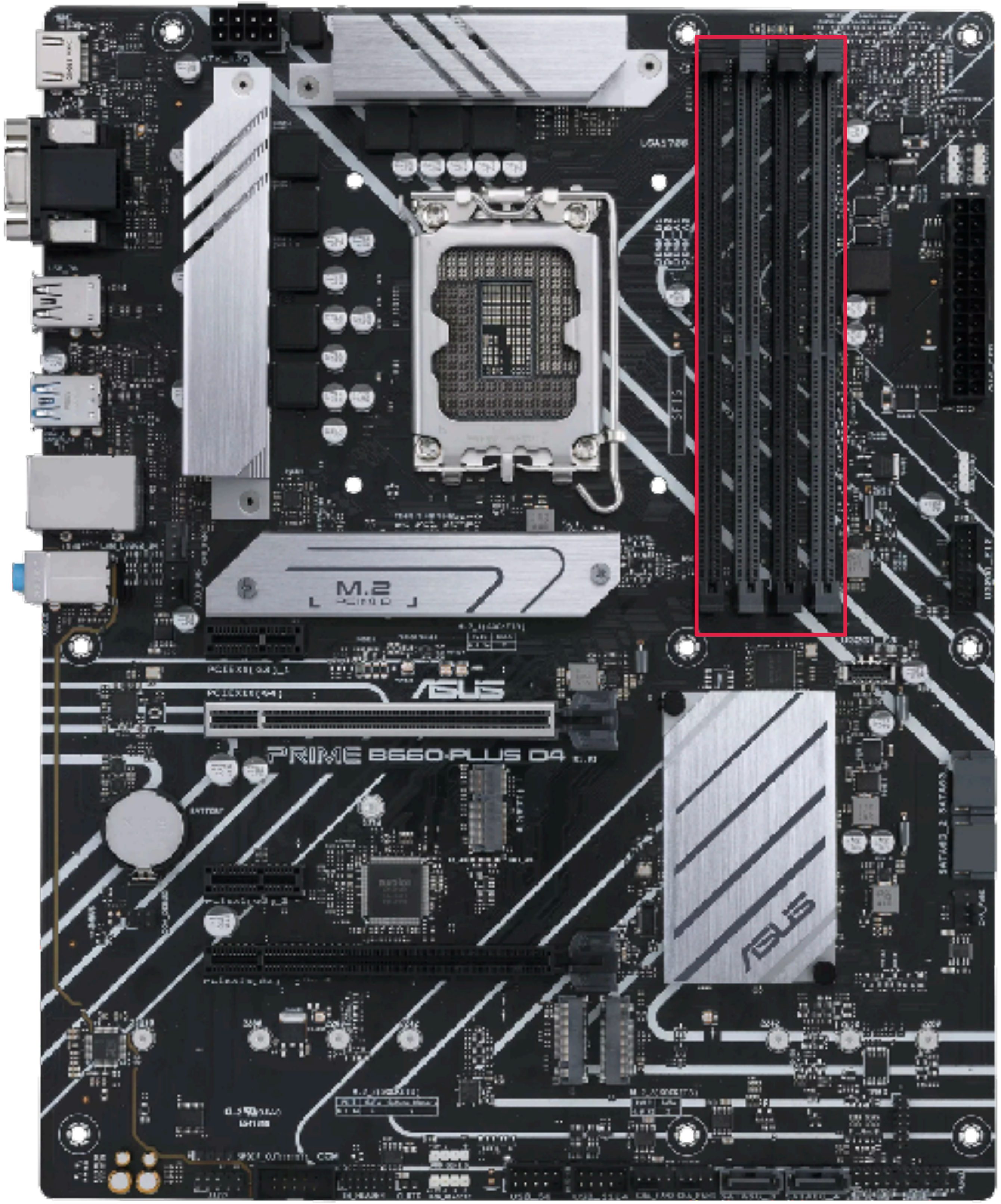


https://maji91.com/product/intel-core-i7-12700-3-6ghz-cpu-25mb-cache-lga1700-tray/

https://www.ocinside.de/review/intel_core_i7_12700k/3/

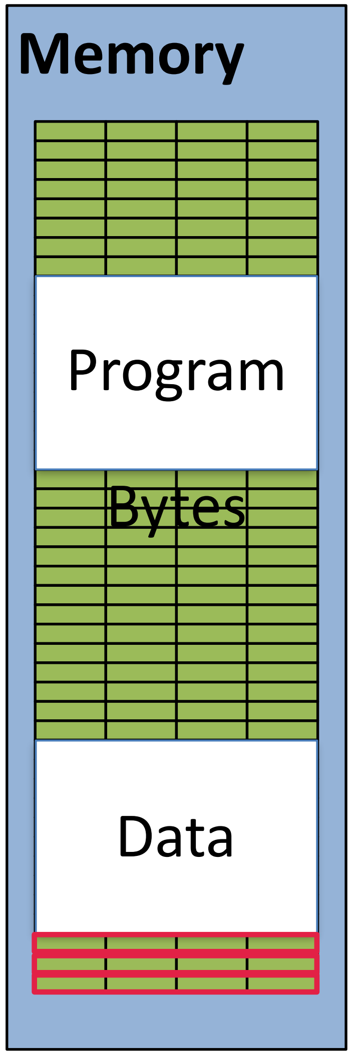


DIMM   DDR4 5066 MHz

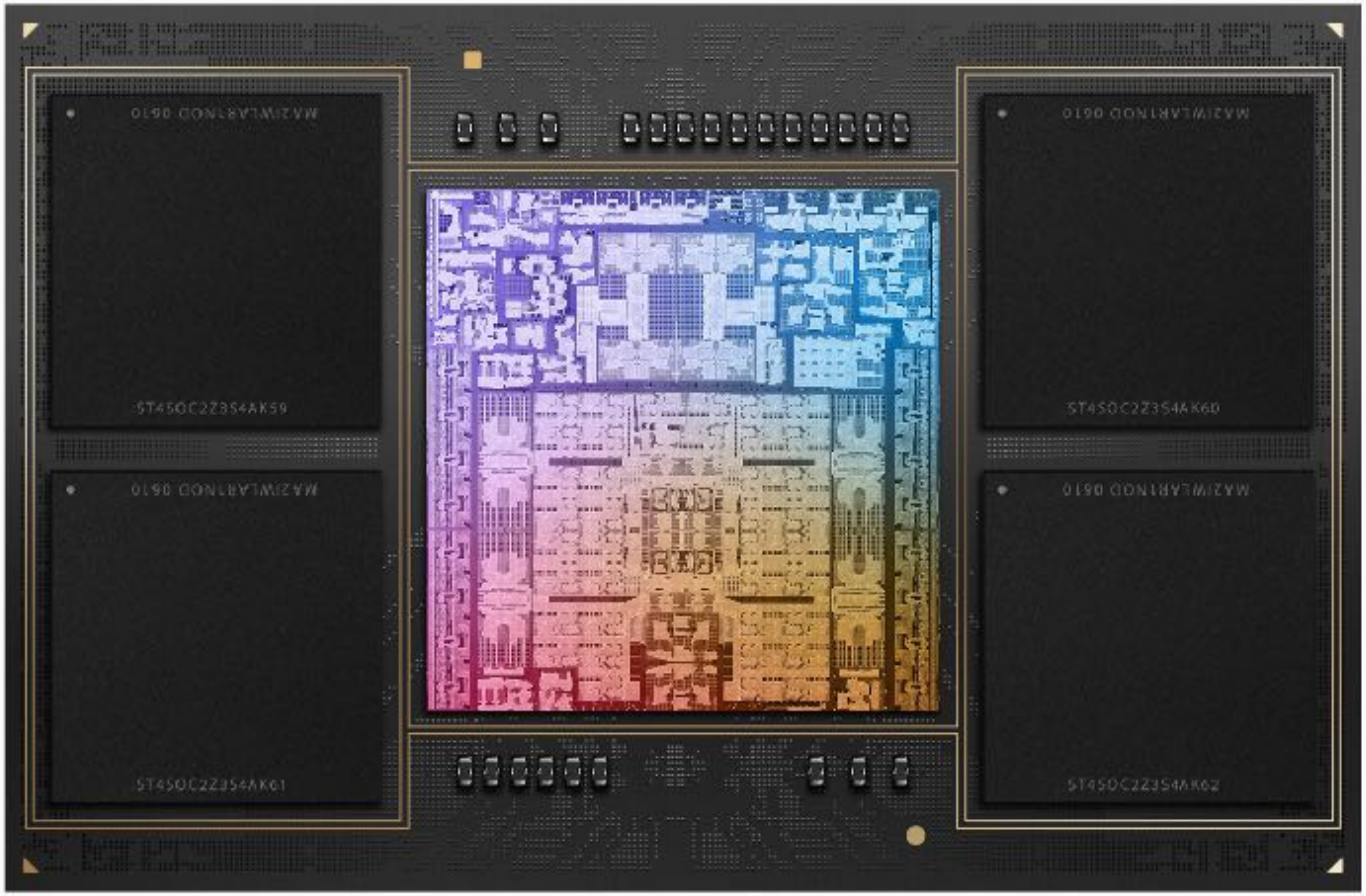


https://www.asus.com/motherboards-components/motherboards/prime/prime-b660-plus-d4/

Kingston
8/16/32G
SDRAM

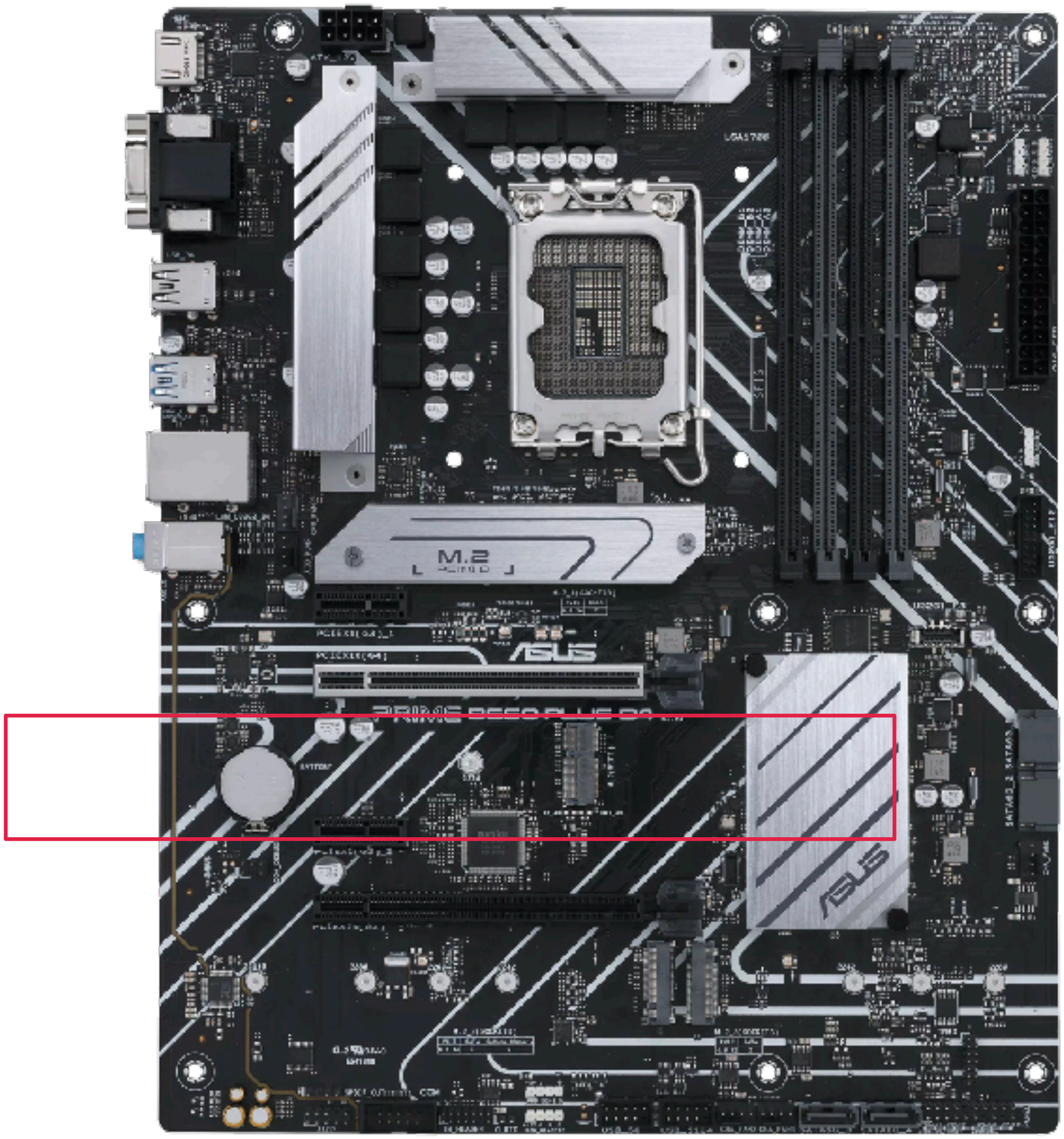


Real Stuff



Credit to Apple. Apple M2 Max, with 96GB unified RAM

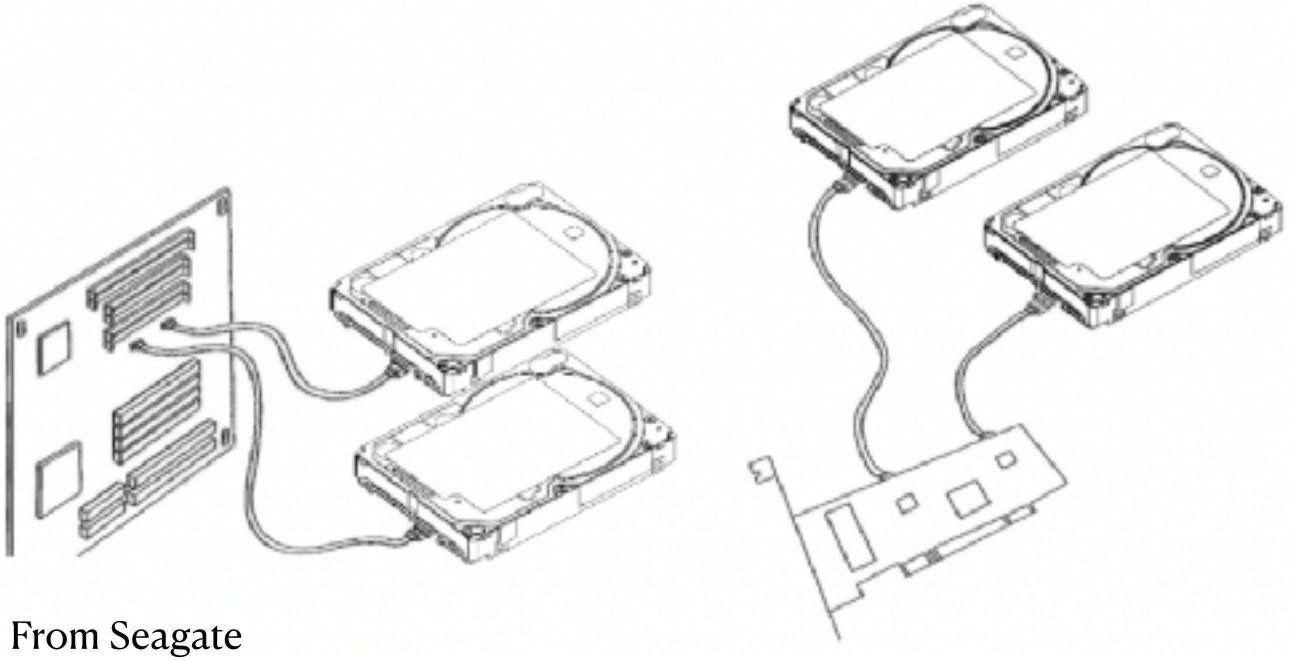
Real Stuff



https://www.asus.com/motherboards-components/motherboards/prime/prime-b660-plus-d4/

Real Stuff

SSD vs. HDD

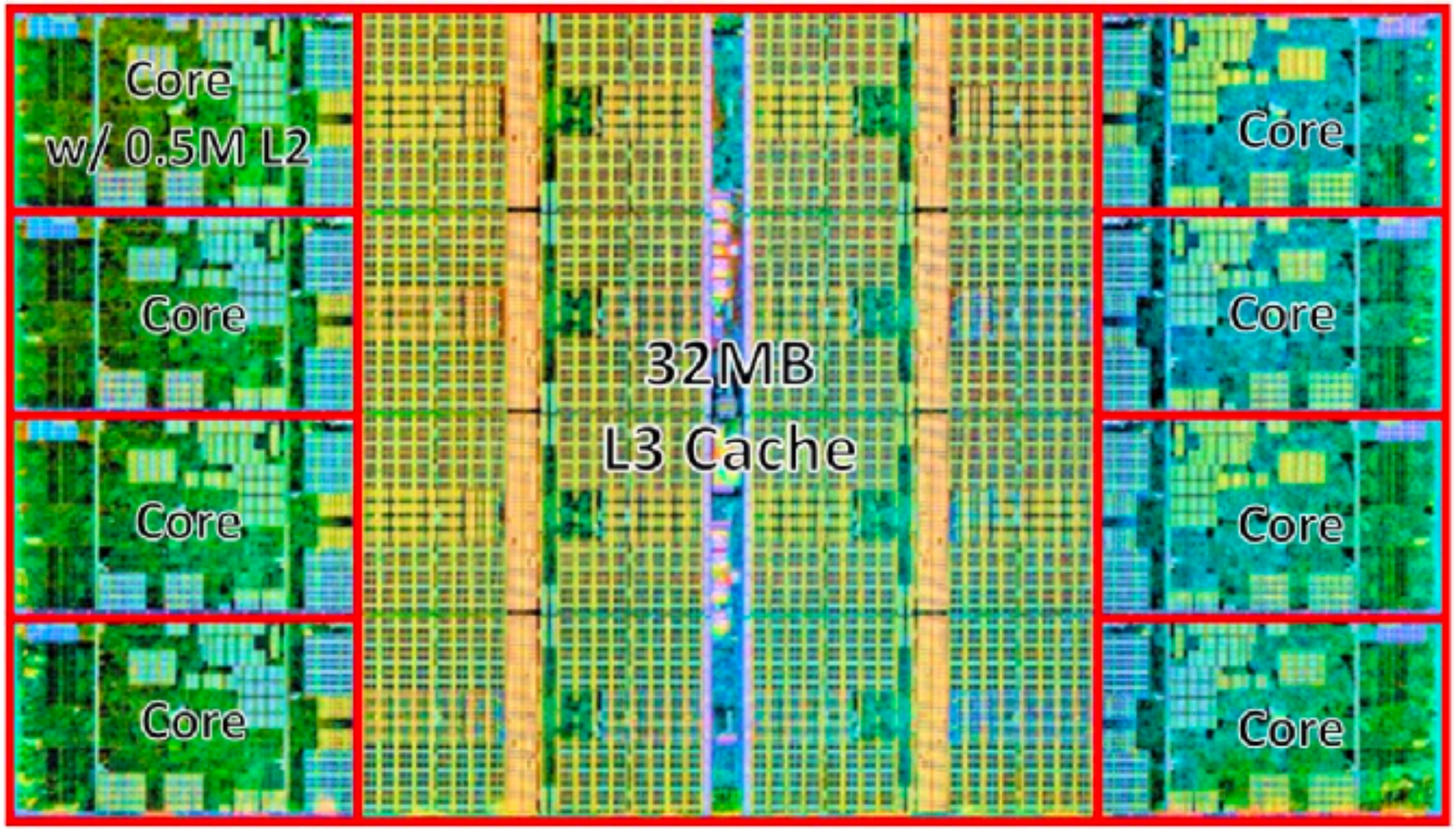


From Seagate

https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-pro-nvme-m2-512gb-mz-v7p512bw/
https://www.seagate.com/in/en/products/hard-drives/barracuda-hard-drive/

Real Stuff— Inside a CPU

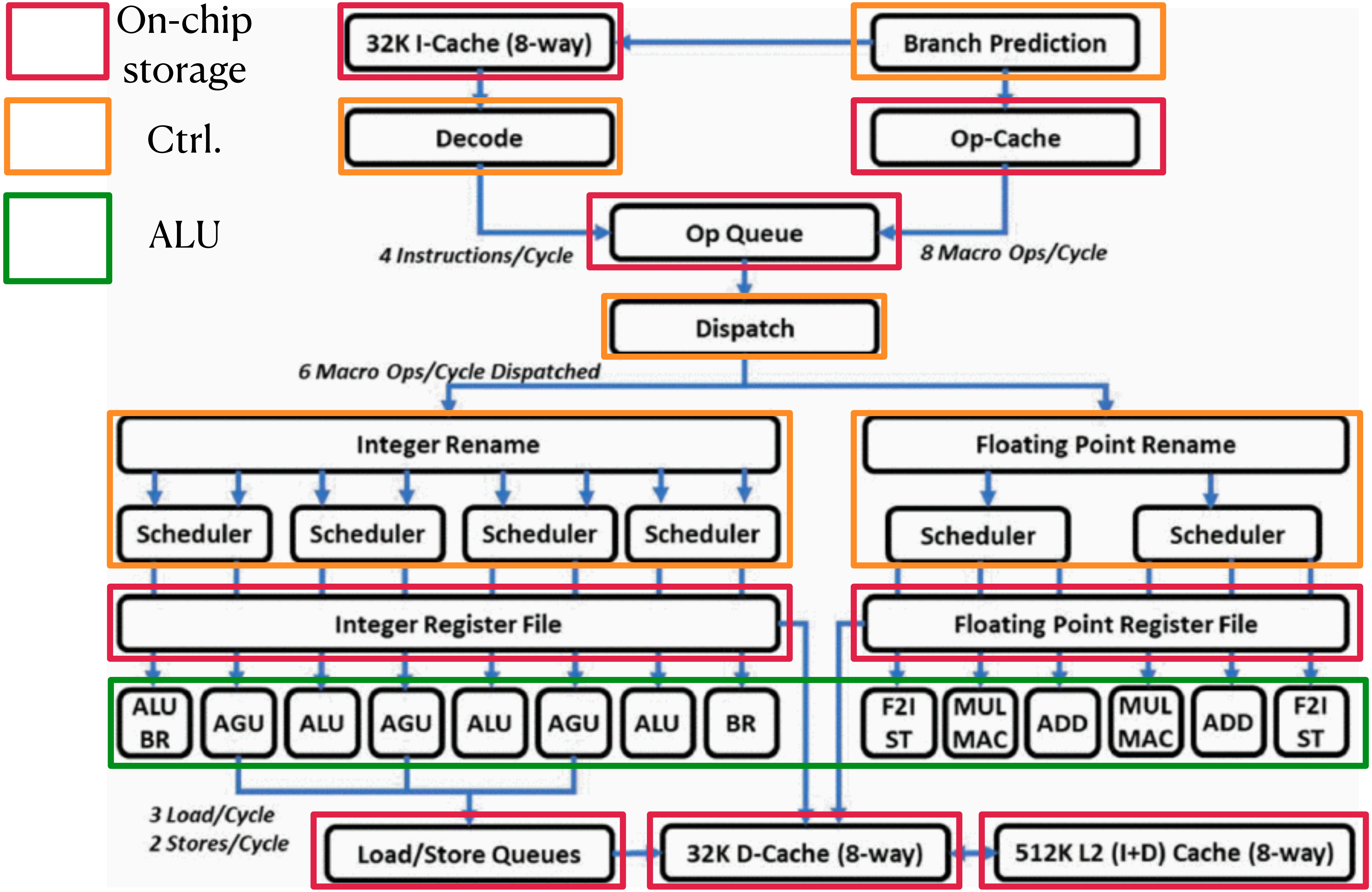
AMD Zen 3 8-core CPU, 7 nm process, 4.08B transistors in 68 mm²



T. Burd *et al.*, "Zen3: The AMD 2nd-Generation 7nm x86-64 Microprocessor Core," *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, San Francisco, CA, USA, 2022, pp. 1-3.

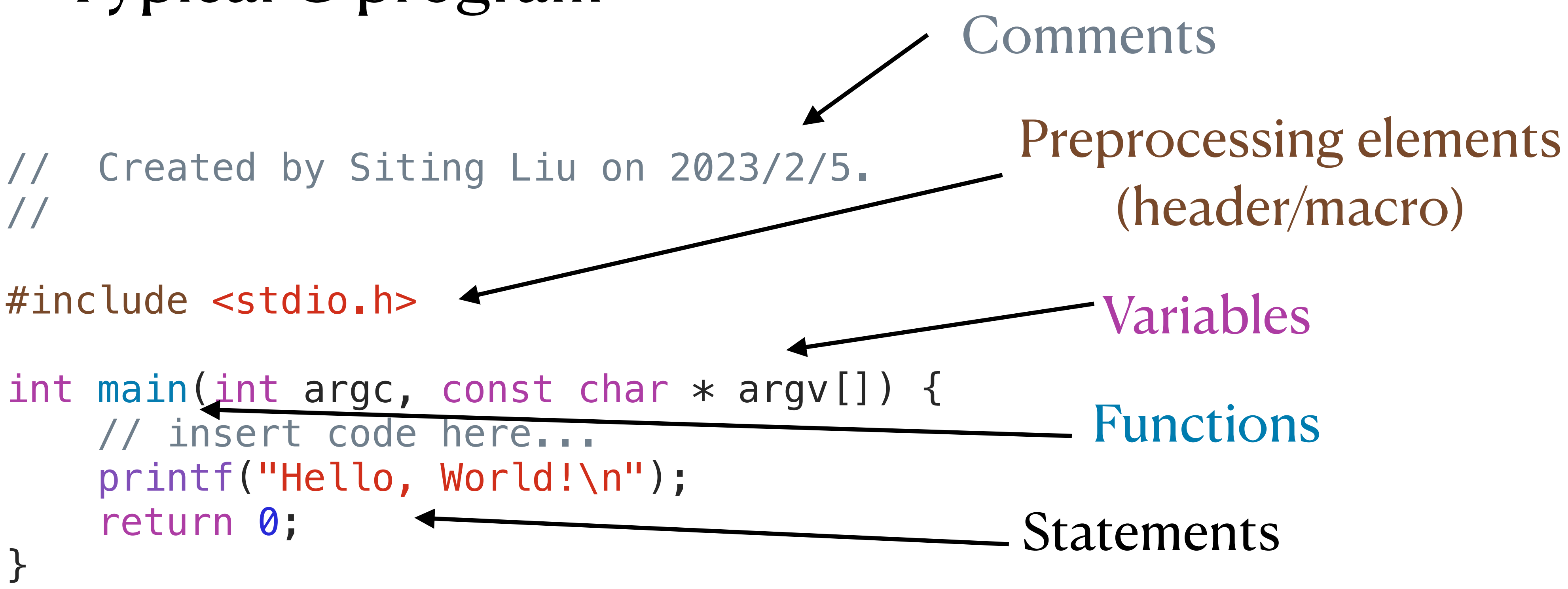
Real Stuff— Inside a CPU

AMD Zen 3, 7 nm process, a single core



Back to C

- Typical C program



```c
//  Created by Siting Liu on 2023/2/5.
//

#include <stdio.h>

int main(int argc, const char * argv[]) {
    // insert code here...
    printf("Hello, World!\n");
    return 0;
}
```

Comments

Preprocessing elements (header/macro)

Variables

Functions

Statements

- Must C program start with main()? (RTFM)

Variables

- Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

Must declare the type of data a variable will hold;

Initialize, otherwise it holds garbage

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
long	larger signed integer	-6,000,000,000
(un)signed char	single text character or symbol	'a', 'D', '?’
float	floating point decimal numbers	0.0, 1.618, -1.4
double	greater precision/big FP number	10E100

C89 standard defines a lot of “Undefined Behavior”s. It means the code may produce unpredictable behavior. It may

- Produce different results on different computers/OS;
- Produce different results among multiple runs;
- Very difficult to re-produce and debug

Integers

- Typed Variables in C

Language	sizeof(int)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64 bits

- C: int should be integer type that target processor works with most efficiently
- Generally: sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - Also, short >= 16 bits, long >= 32 bits
  - All could be 64 bits

Integer Constants

```c
#include <stdio.h>
int main() {
   printf((6-2147483648)>(6)?"T\n":"F\n");
   printf((6-0x80000000)>(6)?"T\n":"F\n");
   return 0;
}
```

Semantics: The value of a decimal constant is computed base 10; that of an octal constant base 8; that of a hexadecimal constant base 16. The lexically first digit is the most significant.
The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffixed decimal **int, long int, unsigned long int;** unsuffixed octal or hexadecimal: **int, unsigned int, long int, unsigned long int;** suffixed by the letter u or U: **unsigned int, unsigned long int**; suffixed by the letter l or L: **long int, unsigned long int**; suffixed by both the letters u or U and 1 or L: **unsigned long int**.

Range of each type defined in <`limits.h`**> (INT_MAX, INT_MIN)**

Consts. and Enums. in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

- You can have a constant version of any of the standard C variable types

- Enums: a group of related integer constants. Ex:

```
    enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
    enum color {RED, GREEN, BLUE};
```

C Syntax: Variable Declarations

- All variable declarations must appear before they are used (e.g., at the beginning of the block)
- A variable may be initialized in its declaration; if not, it holds garbage!
- Examples of declarations:
  - Correct: {
    ```
    int a = 0, b = 10;
     ...
    ```
  - Incorrect: `for (int i = 0; i < 10; i++)`

    }

Newer C standards are more flexible about this…

C Syntax: True or False

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of pointer: more on this later)
- No explicit Boolean type
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Python: only 0 or empty sequences are false, anything else is true!

C operators

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: ( )
- order relations: <, <=, >, >=
- increment and decrement: ++ and --
- member selection: ., ->
- conditional evaluation: ? :

Make sure you understand each operator!

Typed C Functions

- You need to declare the return type of a function when you declare it (plus the types of any arguments)
- You also need to declare functions before they are used
  - Usually in a separate header file, e.g.

```
int number_of_people();
float dollars_and_cents();
int sum(int x, int y);
```

- `void` type means "returns nothing"

```
int number_of_people()
{  return 3;}

float dollars_and_cents ()
{ return 10.33; }

int sum (int x, int y)
{ return x + y;}
```

Summary

- C preprocessing
- How does C work
- Basic C elements
- Variables and functions