Exercise Session 5

Systems Programming and Computer Architecture

Fall Semester 2024

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/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

Agenda

- DDCA ISA and Microarch. Recap
- Intro to Assembly
- Basic Assembly
- Assembly Recap & Quiz
- Assembly Hints

Basic Assembly

Assembly Control Flow

DDCA Recap

DDCA Recap

- Material is mainly from DDCA's Book Harris and Harris (see my website for full name)

Notation

Systems@ETH Zürich

- **Instruction Set Architecture (ISA)**:
- Def: Abstract model which defines how a computer understands and executes instructions: it defines **set of instructions** a processor can execute, **data types** the instructions operator on, **registers** the CPU uses, **Adressing modes** i.e. how instructions are fetched from memory

- **Michroarchitecture**:
- Def: **Implementation** of a ISA in a specific processor design, i.e. **how the hardware** internally performs instructions by the ISA

Instruction Set Architecture

- **Examples for ISAs**: x86-64, ARM (Advanced RISC Machines), RISC-V, MIPS, PowerPC, SPARC, Z/Architecture (IBM Mainframes)

x86 / x86-64	CISC	Desktops, laptops, servers (Intel, AMD)
ARM	RISC	Mobile devices, embedded systems, some laptops
RISC-V	RISC	Open-source processors, research, IoT, embedded
MIPS	RISC	Embedded systems, networking devices (less common now)
PowerPC	RISC	Older Macs, embedded, some servers
SPARC	RISC	Enterprise servers, scientific computing

Instruction Set Architecture

- **ISAs** thus define how our assembly code looks (because this is inherently what an ISA describes):
- MIPS (LHS), x86 (RHS)

```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $s0, 0($t0)     # f = A[f]
addi $t2, $t0, 4
lw   $t0, 0($t2)
add  $t0, $t0, $s0
sw   $t0, 0($t1)
```

```
Q1:   proc near
      push    sPassword
      call    _strlen
      pop     ecx
      mov     esi, eax
      mov     ebx, offset sMyPassword
      push    ebx
      call    _strlen
      pop     ecx
      cmp     esi, eax
      jz      short loc_4012B2
      xor     eax, eax
      jmp     short end_proc
loc_4012B2:
      push    esi
      push    ebx
      push    sPassword
      call    _strcmp
      add     esp, 8
      test    eax, eax
      jnz     short loc_4012CC
      mov     eax, 1
      jmp     short end_proc
loc_4012CC:
      xor     eax, eax
end_proc:
      pop     esi
      pop     ebx
      pop     ebp
      retn
endp
```

Instruction Set Architecture (ISA)

- **Computers language**: Instructions
- **Computers vocabulary**: Instruction Set (the things we can do like add, sub, shl etc.)
- **Machine language**: Computers only understand 0/1: Instructions are thus encoded in a binary format in “machine language”
- Since humans cannot easily read 0/1s in machine language, we represent instruction in this symbol format called **assembly language**
- **ISA and Assembly**: ISA defines the instructions and their binary encoding (machine code) that the processor understands, while assembly language is a human-friendly way to write those same instructions

DDCA Recap: ISA MIPS

Instruction Set Architecture (ISA): MIPS

- **Instructions Examples**: Left code is high level language (C, C++, Java), RHS in MIPS
- First Part: called **mnemonic** indicates what to perform, operation is performed on b,c the **source operands** and stored in the **destination** operand

Code Example 6.1 ADDITION

High-Level Code	MIPS Assembly Code
`a = b + c;`	`add a, b, c`

Instruction Set Architecture (ISA): MIPS

Systems@ETH Zürich

- **Highlevel code** can yield multiple assembly instructions (i.e. we have only very simple assembly instructions, there is no 1 to 1 mapping from high level concepts to assembly)

Code Example 6.3 MORE COMPLEX CODE

High-Level Code	MIPS Assembly Code

`a = b + c - d;  // single-line comment`
`/* multiple-line`
`comment */`
	`sub t, c, d  # t = c - d`
	`add a, b, t  # a = b + t`

Instruction Set Architecture (ISA): MIPS

Systems@ETH Zürich

- The machine (michroarchitecture) provides "**registers**", things were we can store stuff: we can access them with $ in MIPS, with % in x86 => we have 32 registers in mips

---

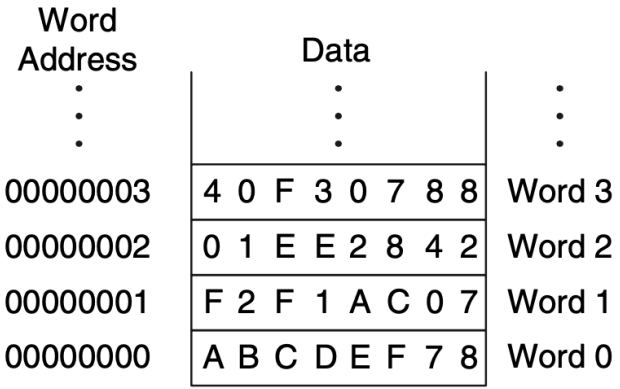
Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	MIPS Assembly Code

`a = b + c - d;`	`# $s0 = a, $s1 = b, $s2 = c, $s3 = d`
	
	`sub $t0, $s2, $s3  # t = c - d`
	`add $s0, $s1, $t0  # a = b + t`

Instruction Set Architecture (ISA): MIPS

- Since we only have limited number of registers: also have memory which we can access



Code Example 6.6 READING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

Instruction Set Architecture (ISA): MIPS

Systems@ETH Zürich

- Next to **memory** and **registers** we can work with **immediates** (i.e. constants)

Code Example 6.9 IMMEDIATE OPERANDS

High-Level Code	MIPS Assembly Code
`a = a + 4;`	`# $s0 = a, $s1 = b`
`b = a – 12;`	`addi $s0, $s0, 4  # a = a + 4`
	`addi $s1, $s0, –12  # b = a – 12`

Instruction Set Architecture (ISA): MIPS

Systems@ETH Zürich

- Assembly is for **humans** to read: but machines only understand **machine code**: need to bring assembly into **machine language**
- **Idea:** encode all instruction as words that can be stored in memory, all 32bit
- MIPS has 3 type of Instructions: R-type, I-type, J-type (Register, Immediate, Jump)

Instruction Set Architecture (ISA): MIPS

- R-type (Register type): uses 3 registers as operands, 2 as source 1 as destination
- Operation to be performed is encoded in “op” and “funct” field: all R-type instr. Have opcode=0 and funct is 32 for add and 34 for substract
- Operands encoded in rs, rt and rd (rs and rt source, rd dest)

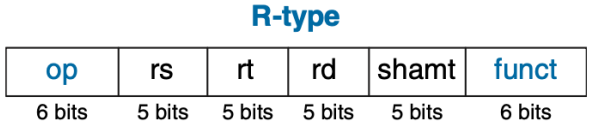


Figure 6.5 R-type machine instruction format

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`add $s0, $s1, $s2`	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
`sub $t0, $t3, $t5`	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.6 Machine code for R-type instructions

Instruction Set Architecture (ISA): MIPS

- Similar for I-type instructions: different interpretation of the bits but conceptually the same

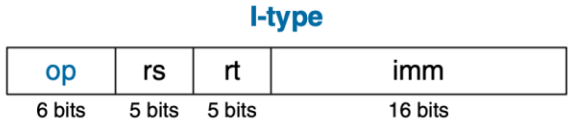


Figure 6.8 I-type instruction format

Assembly Code	op	rs	rt	imm	op	rs	rt	imm	Machine Code
addi $s0, $s1, 5	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
addi $t0, $s3, -12	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
lw $t2, 32($0)	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
sw $s1, 4($t1)	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

Field Values

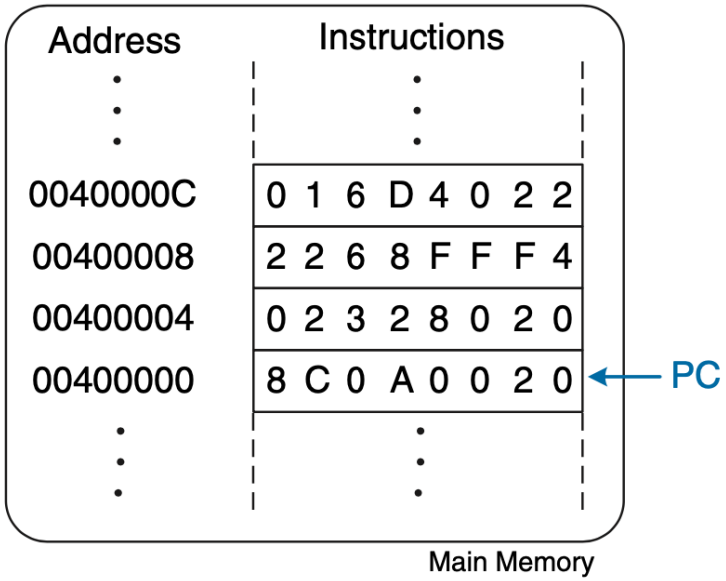
Figure 6.9 Machine code for I-type instructions

Instruction Set Architecture (ISA): MIPS

- Now we can store an entire program in memory

Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

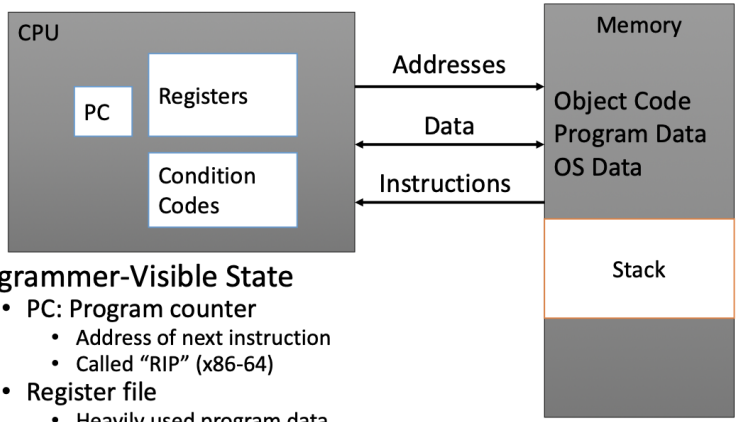
Stored Program



Instruction Set Architecture (ISA): MIPS

- This is what they meant with what Assembly programmer sees

Assembly programmer's view



Programmer-Visible State

- PC: Program counter
  - Address of next instruction
  - Called "RIP" (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code, user data, (some) OS data
- Includes stack used to support procedures

Instruction Set Architecture (ISA): MIPS

- But what is happening below, aka “under the hood”?
- Note that the following recap should **tie together** SPCA and DDCA again: you haven’t looked at this in the lecture and you will only briefly look at Computer Architecture later

DDCA Recap: Michroarchitecture

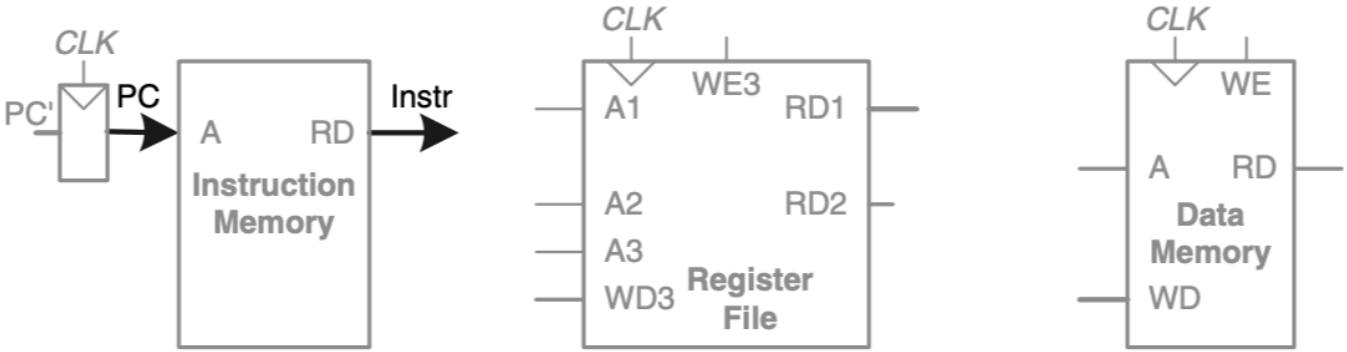
Michroarchitecture

Systems@ETH Zürich

- **Microarchitecture**: specific arrangement of ALUs, FSMs, Memories etc.
- One arch. Like MIPS can have **many different microarchitectures** with different performance, cost and complexity: they all run the same programs since all architectures share the same “language” (ISA) but they can vary in cost, performance and complexity

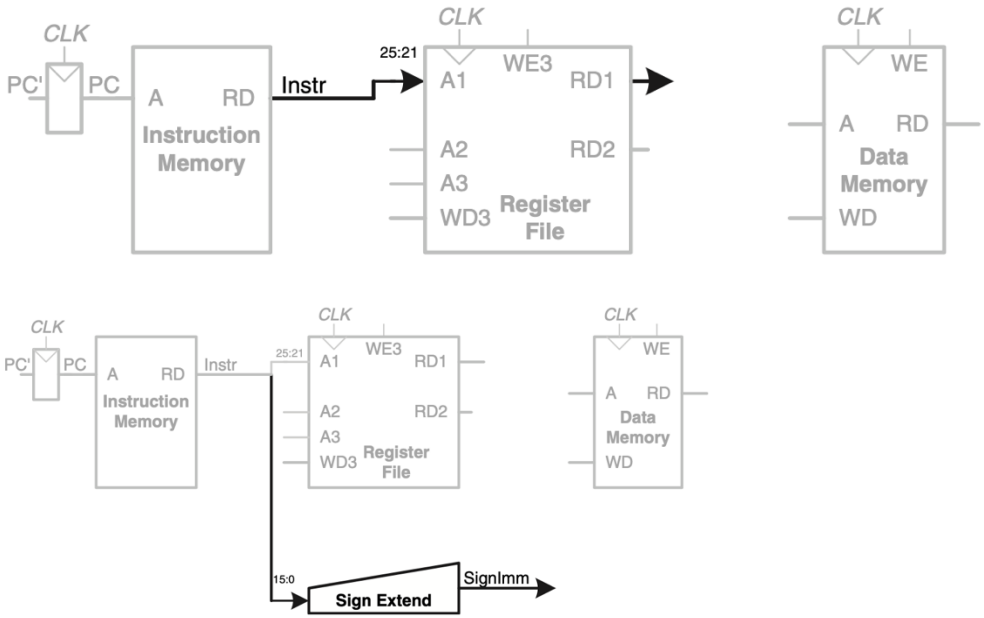
Michroarchitecture

- We fetch instructions from memory, extract the information from the 32bit instruction depending on which opcode it specifies to get the correct register etc.



Michroarchitecture

- **Load** instruction: get base reg (25:21) and sign ext. immediate



Michroarchitecture

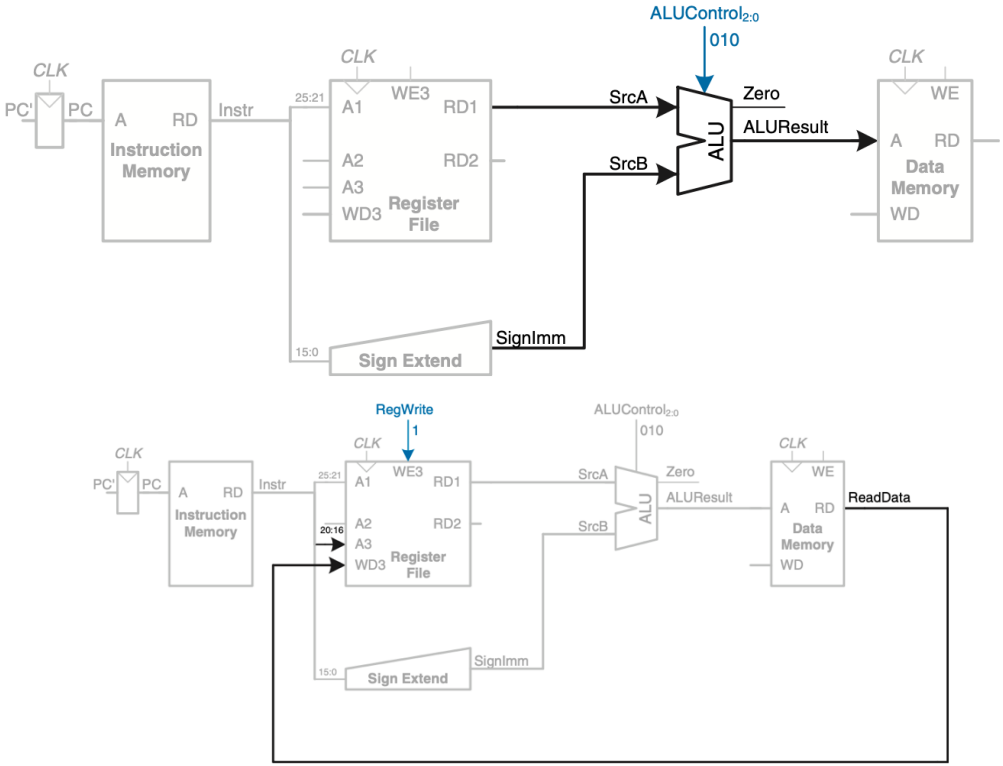


Figure 7.6 Write data back to register file

Michroarchitecture

- **Increment** Program Counter to get next instruction etc.

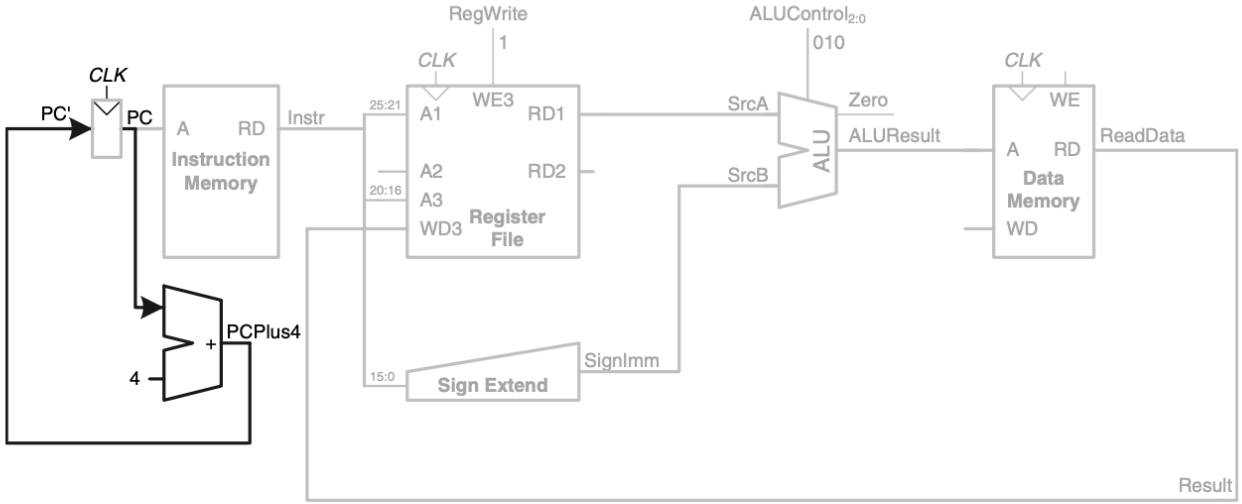


Figure 7.7 **Determine address of next instruction for PC**

Michroarchitecture

- **Full** Processor looks like this: going to briefly look at it later in SPCA in Computer Architecture lecture

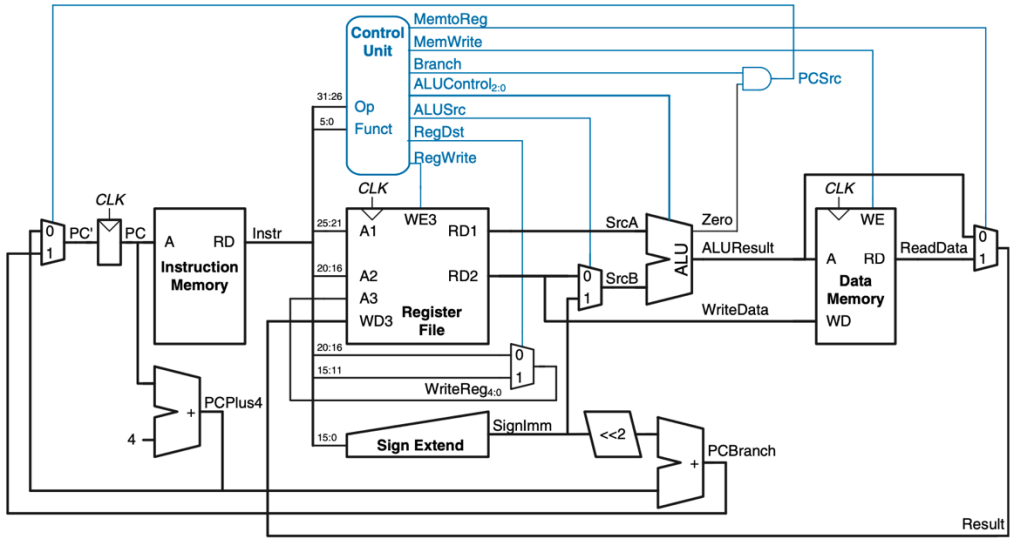


Figure 7.11 Complete single-cycle MIPS processor

Instruction Set Architecture (ISA): MIPS

Systems@ETH Zürich

- **Important takeaway**: **ISA** defines which instructions exist and how they should look like (e.g. add r1, r2, r3), **michroachitecture** HOW its implemented, i.e. how many ALUs etc. which you don’t need to know when writing assembly

Recall: DDCA Exam

- **This is** exactly what you did in DDCA
- **=>** Do I care what kind of branch prediction we have when writing assembly? No: so microarch etc.

Initials: _____ Digital Design and Computer Architecture August 11th, 2022

3 ISA vs. Microarchitecture [30 points]

Circle whether each of the following is an aspect of the ISA or the microarchitecture.

Note: we will subtract 1 point for each ***incorrect*** *answer and award 0 points for unanswered questions.*

1. [2 points] Two-level global branch prediction.
   1. ISA 2. Microarchitecture
2. [2 points] Location of the bits that identify the destination register in an ADD instruction.
   1. ISA 2. Microarchitecture
3. [2 points] Number of instructions fetched per cycle.
   1. ISA 2. Microarchitecture
4. [2 points] Ratio of the number of floating-point to integer general-purpose registers.
   1. ISA 2. Microarchitecture
5. [2 points] Number of integer arithmetic and logic units (ALUs).
   1. ISA 2. Microarchitecture

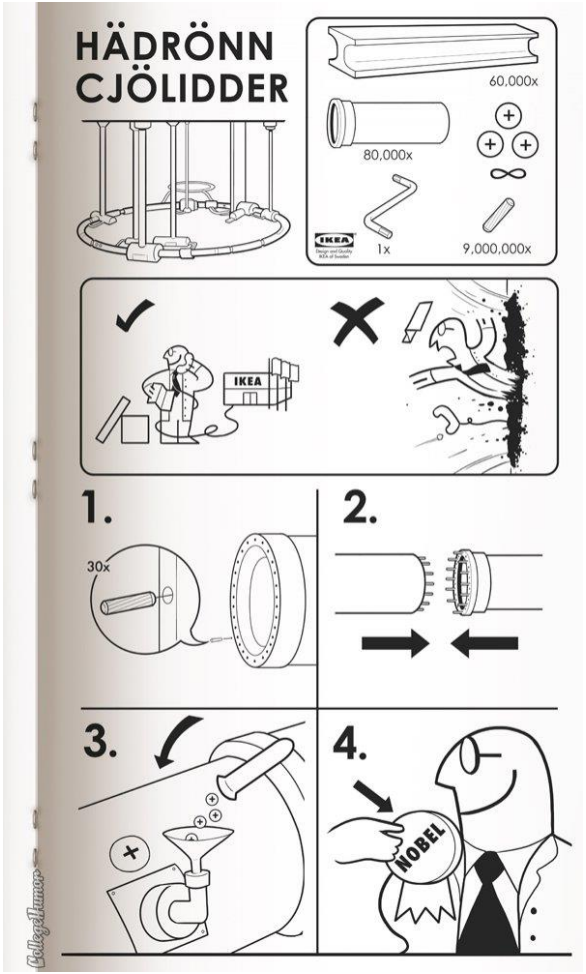
Note: MIPS vs x86

Systems@ETH Zürich

- DDCA, this Recap: we looked at **MIPS** which is one architecture
- SPCA: we look at **x86** , conceptually its again just a different architecture so very similar to MIPS
- **Differences**: as it's a different architecture assemblycode looks differently (e.g. **add r1,r2,r3** in MIPs and registers are $r1 in MIPS, but **add %rax, %rax** in x86 and **%r1** in x86 etc.)

x86 Assembly

Not this assembly…



Obtaining Assembly

- You can have GCC to output assembly code
  `gcc -S code.c`
- This will produce code.S. You can compile single C-files without main()
- Be careful with optimization flags –O
  - Optimized code is often harder to debug
  - Higher optimization levels does not always equal faster code

Example: gcc -S

- Example: gcc –S hello.c

hello.c

```c
#include <stdio.h>

int main(int argc, char *argv[]){
  printf("hello, world\n");
  return 0;
}
```

hello.s

```
        .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
        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"

.subsections_via_symbols
```

Compiling C

- Different compiler will produce different results, try it online!
  https://gcc.godbolt.org/

- Interested in compiler internals?
  http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf

-Ox: The Effects on the Code

```c
/* string.c */

char string_init(void) {
  char s[] = "Hello";
  return s[1];
}
```

```
/* -O0 */

string_init:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $1819043144, -16(%rbp)
        movw    $111, -12(%rbp)
        movzbl  -15(%rbp), %eax
        popq    %rbp
        ret
```

-Ox: The Effects on the Code

```
/* string.c */

char string_init(void) {
  char s[] = "Hello";
  return s[1];
}

/* -O */

string_init:
  movl $101, %eax
  ret
```

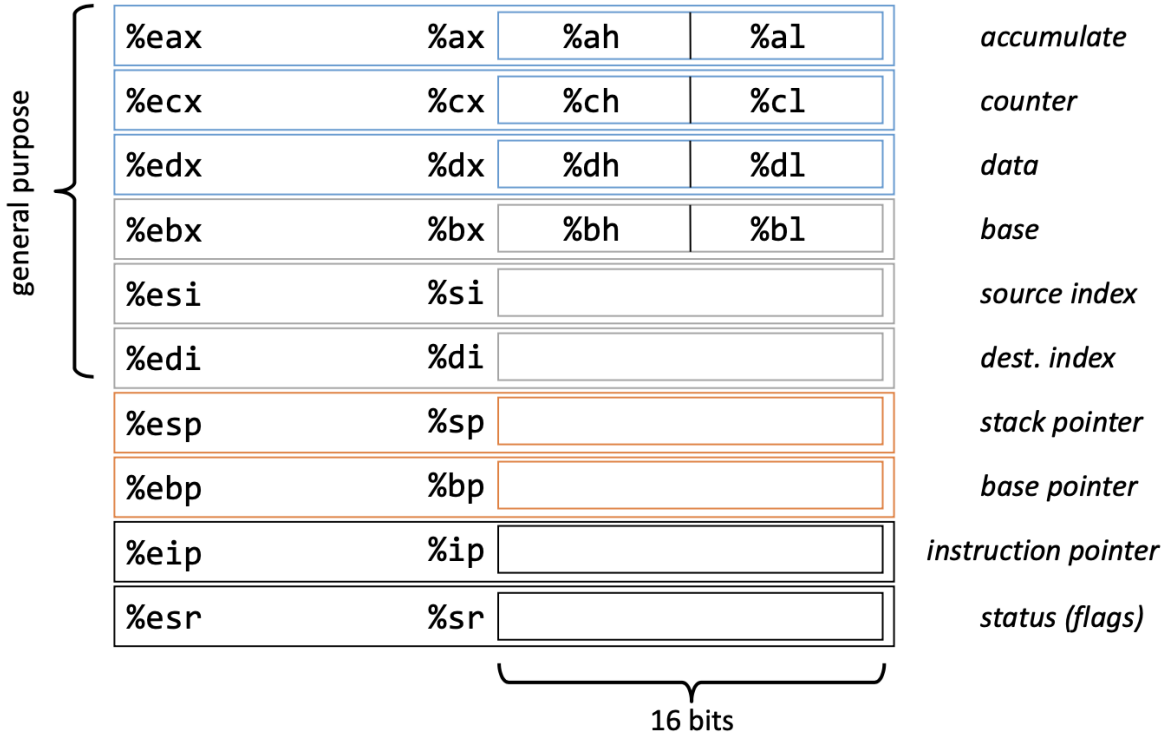


```
/* -O0 */

string_init:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $1819043144, -16(%rbp)
        movw    $111, -12(%rbp)
        movzbl  -15(%rbp), %eax
        popq    %rbp
        ret
```

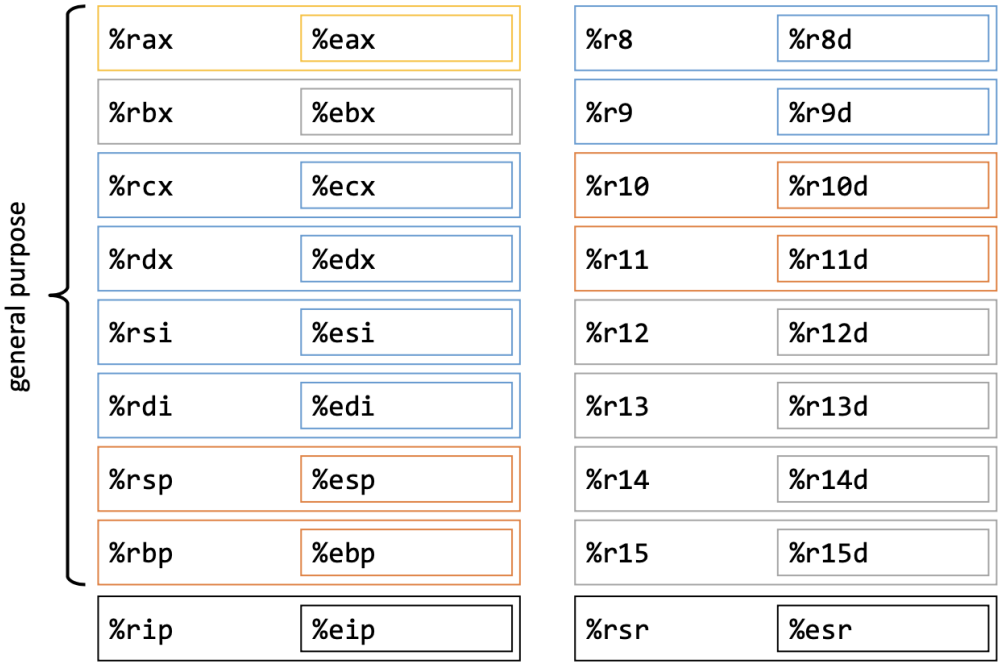
Registers in x86

80386 (ia32) registers

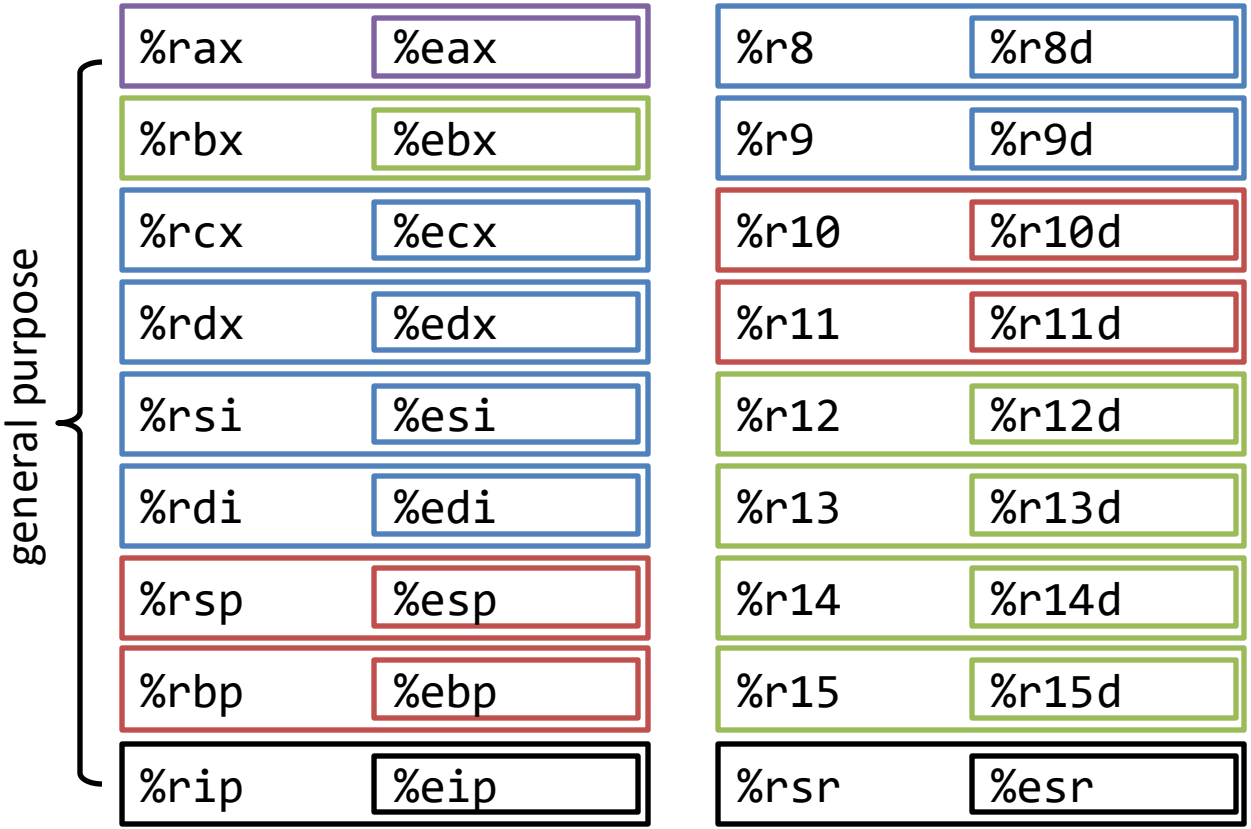


Registers in x86

x86-64 integer registers

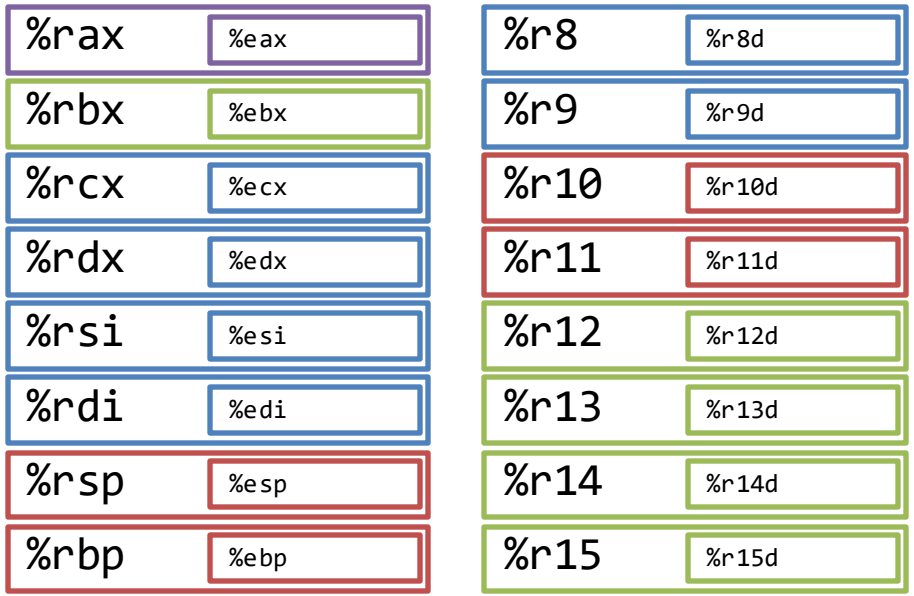


x86-64 integer registers



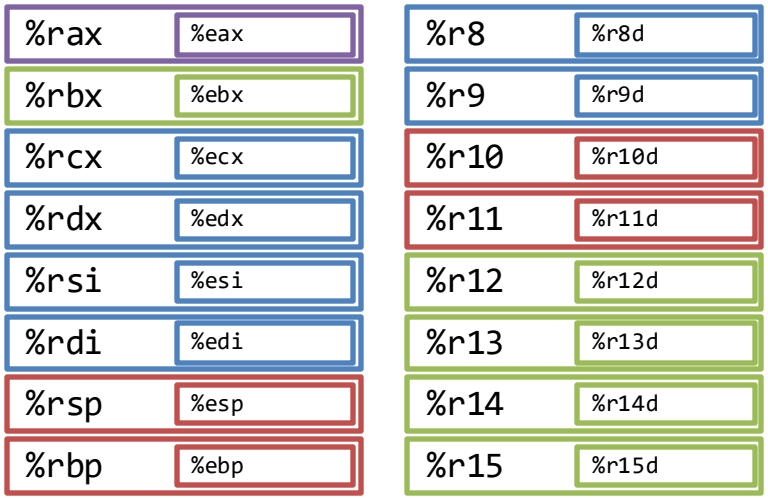
Moving Data

- `mov`***`x`*** *Source*, *Dest*
  - x in {b, w, l, q}
  - `mov`**`q`** *Source*, *Dest*:
    Move 8-byte "quad word"
  - `mov`**`l`** *Source*, *Dest*:
    Move 4-byte "long word"
  - `mov`**`w`** *Source*, *Dest*:
    Move 2-byte "word"
  - `mov`**`b`** *Source*, *Dest*:
    Move 1-byte "byte"
- Lots of these in typical code



Moving Data

- `mov`***x*** *Source*, *Dest*:
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constant, but prefixed with '`$`'
    - Encoded with 1, 2, 4, 8 bytes
  - ***Register:*** One of 16 integer registers
    - Example: `%eax`, `%r14d`
    - Note some (e.g. `%rsp`, `%rbp`) reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 1,2,4, or 8 consecutive bytes of memory at address given by register
    - Simplest example: `(%rax)`
    - Various other "address modes"



Movl combinations: no direct mem-mem

`movl` operand combinations

	Source	Dest	Src,Dest	C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movl (%rax),%edx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Complete memory addressing modes

Systems@ETH Zürich

- Most General Form:

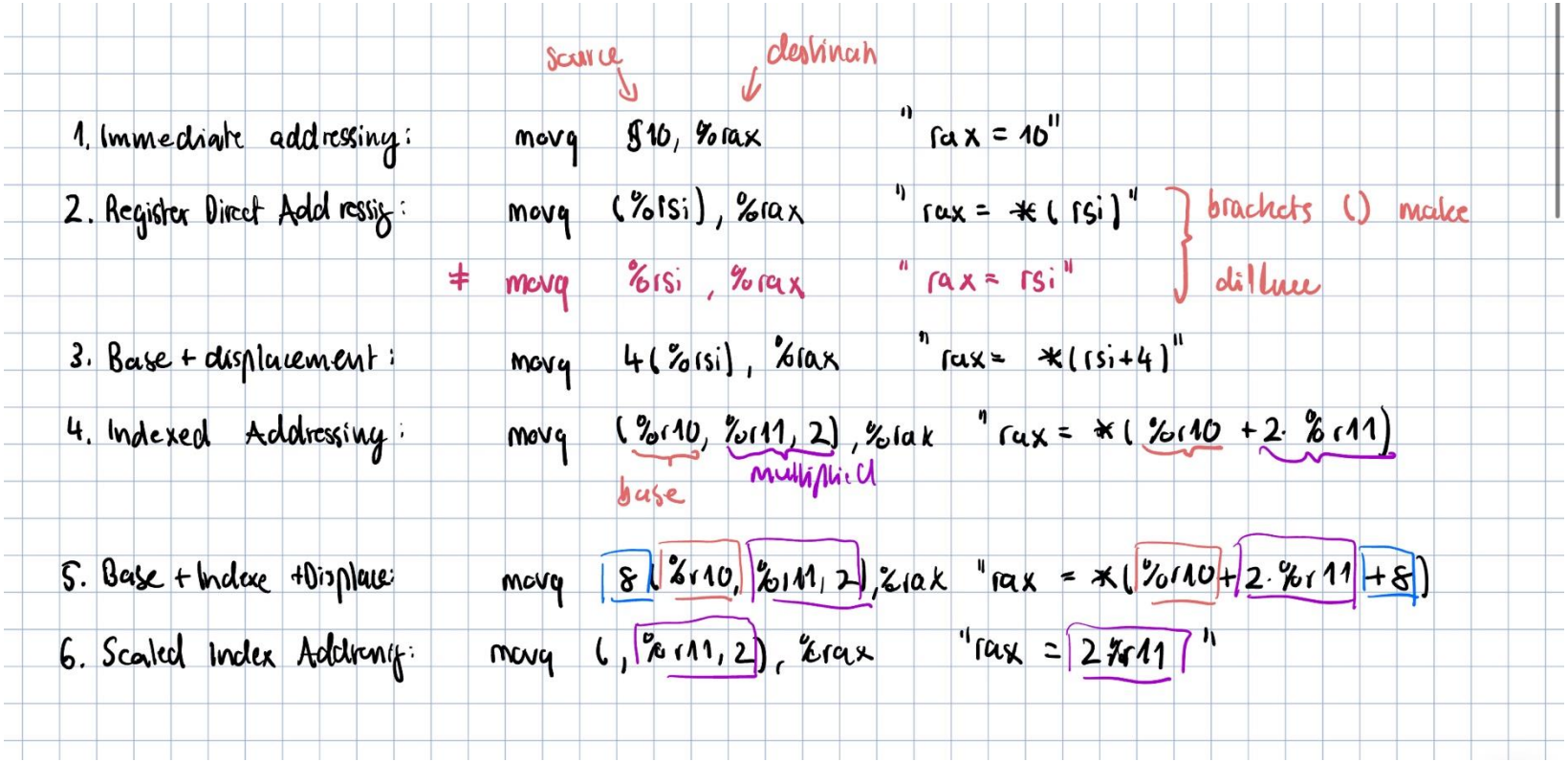
D(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" 1, 2, or 4 bytes (not 8!)
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for %rsp
    (Unlikely you'd use %rbp, either)
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- Special Cases:

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

Example: Memory addressing



Example: Memory addressing

Systems@ETH Zürich

- **Note:** discplacement in x86 is signed!
- For instance lea 0xffffffff(%eax),%esi <=> %esi = %eax-1

Example: Memory addressing

Systems@ETH Zürich

test

```
1   movl $1, 0x604892          # direct (address is constant value)
2   movl $1, (%rax)            # indirect (address is in register %rax)
3
4   movl $1, -24(%rbp)         # indirect with displacement
5                                (address = base %rbp + displacement -24)
6
7   movl $1, 8(%rsp, %rdi, 4) # indirect with displacement and scaled-index
8                                (address = base %rsp + displ 8 + index %rdi * scale 4)
9
10  movl $1, (%rax, %rcx, 8) # (special case scaled-index, displ assumed 0)
11
12  movl $1, 0x8(, %rdx, 4)  # (special case scaled-index, base assumed 0)
13
14  movl $1, 0x4(%rax, %rcx) # (special case scaled-index, scale assumed 1)
15
```

Load Effective Address

- Similar to memory addressing but only calculates address and stores it into destination

```
leax D(Rb,Ri,S),Dest
```

Dest ← Reg[Rb]+S*Reg[Ri]+ D

  - D: Constant “displacement” 1, 2, or 4 bytes (not 8!)
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
    (Unlikely you’d use `%rbp`, either)
  - S: Scale: 1, 2, 4, or 8

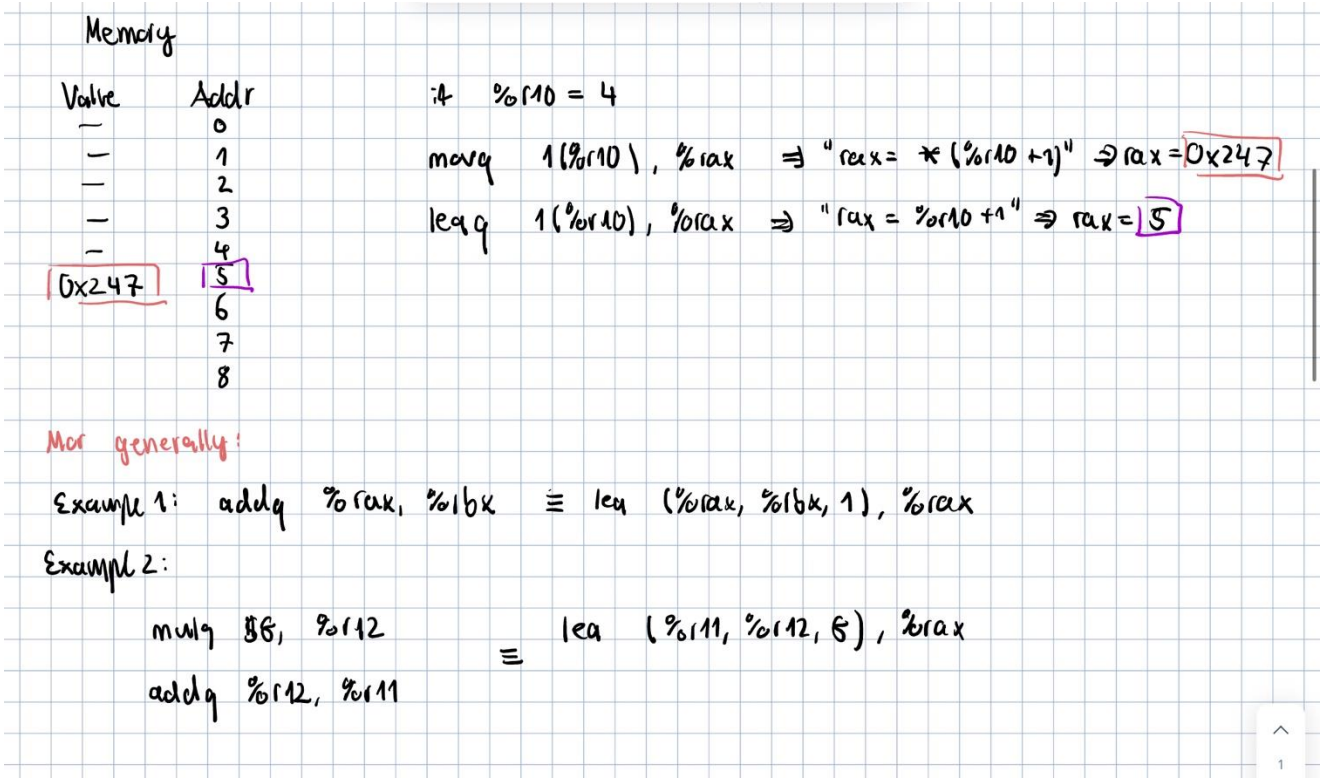
- Same special cases as in memory addressing apply
- Does not set condition codes! (Why?)

Mov and Lea

Systems@ETH Zürich

- Most frequent instruction: ”mov”, which simply copies value from source to dest (for combinations see slides before)
- **Lea** works like **mov** but does **not dereference** the result, but simply calculates the value: i.e. its nothing more than an arithmetic expression

Mov and Lea


Memory
Value
Addr
0
1
2
3
4
0x247
5
6
7
8
if %r10 = 4
movq 1(%r10), %rax ⇒ "rax = *(%r10 +1)" ⇒ rax = 0x247
leaq 1(%r10), %rax ⇒ "rax = %r10 +1" ⇒ rax = 5
Mor generally:
Example 1: addq %rax, %rbx ≡ lea (%rax, %rbx, 1), %rax
Example 2:
mulq $8, %r12
addq %r12, %r11
≡
lea (%r11, %r12, 8), %rax

Mov and Lea

Systems@ETH Zürich

test

```
1   mov src, dst              # general form of instruction dst = src
2   mov $0, %eax              # %eax = 0
3   movb %al, 0x409892        # write to address 0x409892 low-byte of %eax
4   mov 8(%rsp), %eax         # %eax = value read from address %rsp + 8
5
6   lea 0x20(%rsp), %rdi      # %rdi = %rsp + 0x20 (no dereference!)
7   lea (%rdi,%rdx,1), %rax   # %rax = %rdi + %rdx
8
```

Sign and Zeroextending: Movsbl, Movzbl

- **Mov** copies the same number of bytes from src to dst
- When we want to copy smaller bandwidth -> larger bandwidth
- **Movs**{}{}: move **s**ign extend
- **Movz**{}{}: move **z**ero extend
- {}{} is the usually, source and destination with {} in {b,w,l,q}

```
1    movsbl %al, %edx    # copy 1-byte %al, sign-extend into 4-byte %edx
2    movzbl %al, %edx    # copy 1-byte %al, zero-extend into 4-byte %edx
```

Arithmetic operations

- longword variants, others analogously:

Mnemonic	*Format*	*Computation*
`addl`	*Src,Dest*	*Dest ← Dest + Src*
`subl`	*Src,Dest*	*Dest ← Dest - Src*
`imull`	*Src,Dest*	*Dest ← Dest * Src*
`sall`	*Src,Dest*	*Dest ← Dest << Src*
`sarl`	*Src,Dest*	*Dest ← Dest >> Src*
`shrl`	*Src,Dest*	*Dest ← Dest >> Src*
`xorl`	*Src,Dest*	*Dest ← Dest ^ Src*
`andl`	*Src,Dest*	*Dest ← Dest & Src*
`orl`	*Src,Dest*	*Dest ← Dest | Src*
`Incl`	*Dest*	*Dest ← Dest + 1*
`Decl`	*Dest*	*Dest ← Dest - 1*
`Negl`	*Dest*	*Dest ← -Dest*
`Notl`	*Dest*	*Dest ← ~Dest*

Embedding Assembly into C

- **Problem:** Certain registers cannot be addressed by a variable in C directly
- **Observation:** You can access the registers via assembly instruction
- **Conclusion:** Embed assembly code into your C source file.

Inline Assembly

- Basic format to include inline assembly

```
__asm__("movb %bh (%eax)\n\t");
```

- Note: If the statement is unused, it may get deleted!

```
__asm__ volatile ("movb %bh (%eax)\n\t");
```

- **Now:** how to get the contents of the register or provide data for the register?

Volatile?

- The semantics of the volatile keyword differ from language to language

C	Java
“Do not optimize this away”	“Do read the value from the memory not from the cache.”
Important when reading device registers	Gives you some memory guarantees (Cf: Parallel Programming)

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

Inline Assembly General Structure

- **General structure**

```
1   __asm__ (
2       "assembly_code"                 // Assembly instructions as a string
3       : output_operands               // Optional: specifies output variables
4       : input_operands                // Optional: specifies input variables
5       : clobbered_registers            // Optional: specifies modified registers
6   );
7
```

- **Output operands**
- Constraints:
- “=r” operand will be written to any general purpose register
- “r” operand is read only
- “+r” both input and output

```
10  Syntax:
11  : [output_label] "constraint" (variable)
12
13  Example:
14  : [result] "=r" (result)
```

Inline Assembly General Structure

- **General structure**

```
__asm__ (
    "assembly_code"                 // Assembly instructions as a string
    : output_operands               // Optional: specifies output variables
    : input_operands                // Optional: specifies input variables
    : clobbered_registers            // Optional: specifies modified registers
);
```

- **Input operands**
- Constraints:
- “r”: operand can use any general purpose register
- “m” will be read from memroy

```
Syntax:
: [input_label] "constraint" (variable)

Example:
: [input] "r" (x)
```

Inline Assembly General Structure

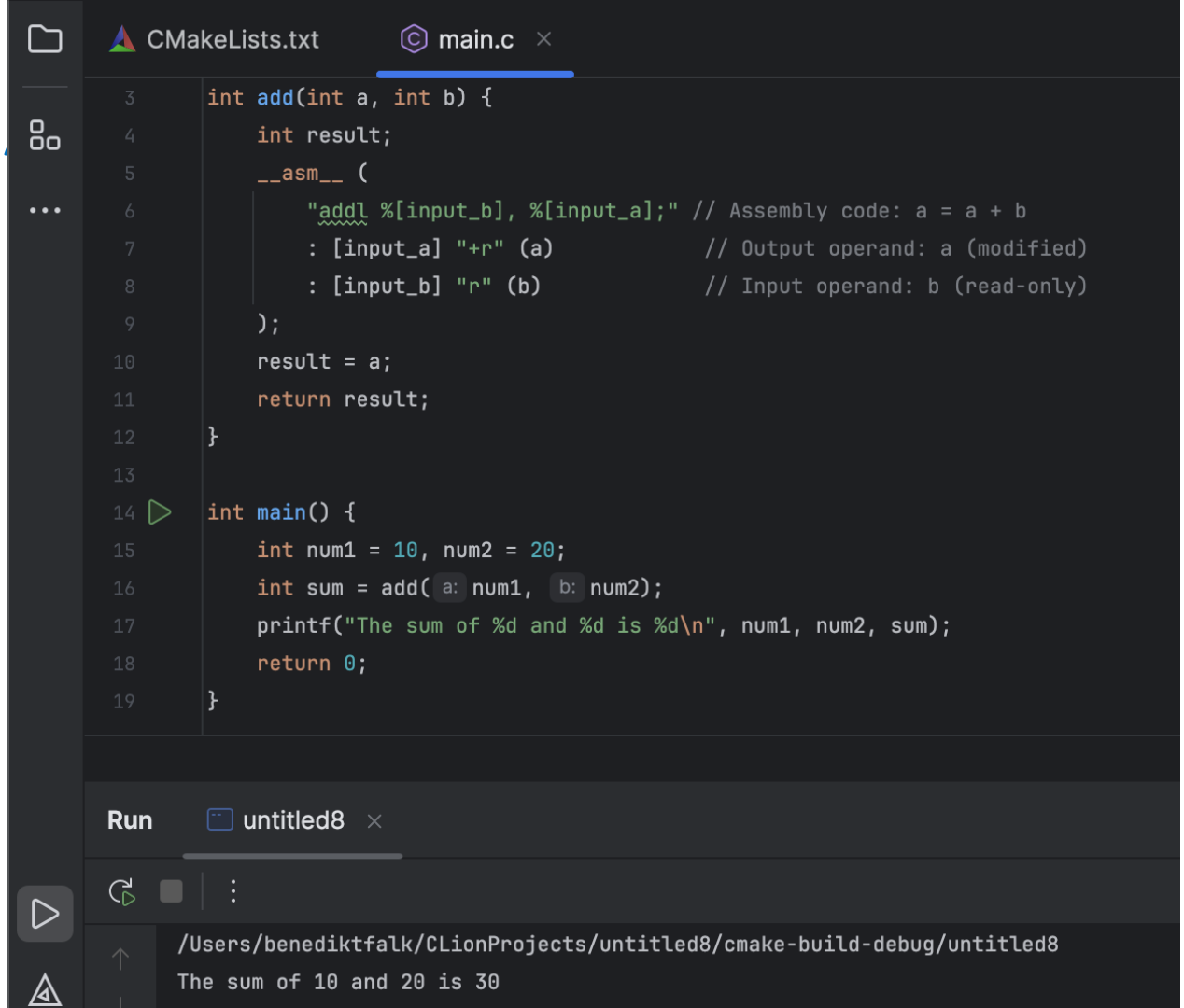
- **General structure**

```
1   __asm__ (
2       "assembly_code"                 // Assembly instructions as a string
3       : output_operands               // Optional: specifies output variables
4       : input_operands                // Optional: specifies input variables
5       : clobbered_registers            // Optional: specifies modified registers
6   );
7
```

- **Clobbered registers**
- Indicates registers which may be changed from the assembly code

```
10   Syntax:
11   : "clobbered_register_1", "clobbered_register_2", ...
12
13   Example:
14   : "memory", "eax", "ebx"
```

Inline A



Systems@ETH Zürich

Positional vs Named Operands

Systems@ETH Zürich

- **Named operands**:

```
10    int a = 10, result;
11    __asm__ (
12        "movl %[in], %%eax; movl %%eax, %[out];" // Use named operands
13        : [out] "=r" (result)    // Output operand with label [out]
14        : [in] "r" (a)           // Input operand with label [in]
15    );
```

- **Positional operands**: enumerate first all output operands, then all input operands (i.e. output0=%0, output1=%1,… outputn=%n, input0=%n+1 etc.)

```
10    int a = 10, b;
11    __asm__ (
12        "movl %1, %%eax; movl %%eax, %0;" // Use positional operands
13        : "=r" (b)    // Output operand (first operand)
14        : "r" (a)     // Input operand (first operand)
15    );
```

Extended Inline Assembly

```
__asm__ ( assembler template
          : output operands /* optional */
          : input operands /* optional */
          : list of clobbered registers /* optional */
         );
```

These registers are modified, don't store other values

```
int a=10, b;
__asm__ ("movl %1, %%eax; movl %%eax, %0;"
         :"=r"(b)
         :"r"(a)
         :"%eax"
         );
```

What's happening here?

Inline Assembly Example: Explained

The value of a (which is 10) is moved into the EAX register.

The value in EAX (now 10) is moved to b.

```
int a=10, b;
__asm__ ("movl %1, %%eax; movl %%eax, %0;"
         :"=r"(b)
         :"r"(a)
         :"%eax"
         );
```

Inline Assembly: Example

```
int bit_count_naive(int x) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result += (x >> i) & 1;
    }
    return result;
}
```

```
int bit_count_asm(int x) {
    int result;
    __asm__ ("popcnt %[in], %[out]"
            : [out] "=r" (result)
            : [in]   "r" (x)
            );
    return result;
}
```

Inline Assembly Example: Explained

```
int bit_count_naive(int x) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result += (x >> i) & 1;
    }
    return result;
}


int bit_count_asm(int x) {
    int result;
    __asm__ ("popcnt %[in], %[out]"
             : [out] "=r" (result)
             : [in]   "r" (x)
             );
    return result;
}
```

- **Popcnt**: x86 instruction which counts number of set bits (i.e. bits=1)
- **%[in]**: refers to input var x
- **%[out]**: refers to output var result

Quiz: Assembly

Questions on Handout

Assembly cheat sheet

Systems@ETH Zürich

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest ← Src
`movsbq` Src,Dest	Dest (quad) ← Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) ← Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed ≥)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed ≤)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned ≥)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned ≤)

Control transfer

`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp ← %rsp − 8, Mem[%rsp] ← Src
`popq` Dest	Dest ← Mem[%rsp], %rsp ← %rsp + 8
`call label`	push addr next instruction, `jmp label`
`ret`	%rip ← Mem[%rsp], %rsp ← %rsp + 8
`endbr64`	End branch target (no-op)
`leave`	%rsp ← %rbp ; `popq` %rbp

Arithmetic operations

`leaq` Src, Dest	Dest ← address of Src
`incq` Dest	Dest ← Dest + 1
`decq` Dest	Dest ← Dest − 1
`addq` Src, Dest	Dest ← Dest + Src
`subq` Src, Dest	Dest ← Dest − Src
`imulq` Src, Dest	Dest ← Dest ∗ Src
`xorq` Src, Dest	Dest ← Dest ^ Src
`orq` Src, Dest	Dest ← Dest | Src
`andq` Src, Dest	Dest ← Dest & Src
`negq` Dest	Dest ← − Dest
`notq` Dest	Dest ← ∼ Dest
`salq` k, Dest	Dest ← Dest ≪ k (also `shlq`)
`sarq` k, Dest	Dest ← Dest ≫ k (arithmetic)
`shrq` k, Dest	Dest ← Dest ≫ k (logical)
`cmpq` Src2, Src1	Set CCs Src1 − Src2
`testq` Src2, Src1	Set CCs Src1 & Src2

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: displacement: 1, 2, or 4 byte constant
  Rb: base register: any integer register
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Integer registers

`%rax`	Return value
`%rbx`	Callee saved
`%rcx`	4th argument
`%rdx`	3rd argument
`%rsi`	2nd argument
`%rdi`	1st argument
`%rbp`	Callee saved
`%rsp`	Stack pointer
`%r8`	5th argument
`%r9`	6th argument
`%r10`	Scratch register
`%r11`	Scratch register
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Quiz a)

Systems@ETH Zürich

a) The register rax currently has value 0. Which of the following statements are true?

(a) Executing movq (%rax), %rcx will cause a segmentation fault.

(b) Executing leaq (%rax), %rcx will cause a segmentation fault.

(c) Executing movq %rax, %rcx will cause a segmentation fault.

(d) Executing addq $8, %rsp will increase the stack allocation by 8 bytes.

Quiz a) solution

a) The register rax currently has value 0. Which of the following statements are true?

(a) Executing movq (%rax), %rcx will cause a segmentation fault.

(b) Executing leaq (%rax), %rcx will cause a segmentation fault.

(c) Executing movq %rax, %rcx will cause a segmentation fault.

(d) Executing addq $8, %rsp will increase the stack allocation by 8 bytes.

- A) Dereferencing 0 yields seg fault
- B) Only lodas 0 into rcx **w/o** dereferencing
- C) Again, moves 0 into rcx without dereferencing
- D) Stack increases when decreasing the stack pointer (Remember: stack grows downwards)

Quiz a)

a) True
   - Tries to load address 0

b) False
   - Computation on address

c) False
   - Move the value 0 to %rcx

d) False
   - Decreases the stack size

Quiz b)

b) Which of the following lines of C produce the same outcome as `lea0xffffffff(%esi),%eax`? (32-bit Machine)

(a) `*(esi-1) = eax`

(b) `esi = eax + 0xffffffff`

(c) `eax = esi - 1`

(d) `eax = *(esi -1)`

Quiz b) solution

b) Which of the following lines of C produce the same outcome as `lea0xffffffff(%esi),%eax`? (32-bit Machine)

(a) `*(esi-1) = eax`

(b) `esi = eax + 0xffffffff`

(c) `eax = esi - 1`

(d) `eax = *(esi -1)`

Solution: c)

- Recall: Immediate in front of full addressing mode is **signed**

Quiz c)

Systems@ETH Zürich

c) Which of the following statements are valid, which are not and why?

(a) `movl(, %eax, 4), %ebx`

(b) `movl 15, (%ebx)`

(c) `movl %eax, 655`

Quiz c) solution

Systems@ETH Zürich

c) Which of the following statements are valid, which are not and why?

(a) `movl(, %eax, 4), %ebx`

(b) `movl 15, (%ebx)`

(c) `movl %eax, 655`

Solution:

(a) Valid: %ebx = 4*%eax

(b) Invalid: 15 is a memory address, not intermediate! mem $\longleftrightarrow$ mem transfers are not allowed

(c) Valid: store content of %eax to memory address 655

Quiz d)

d) Which of the following values of `%eax` would cause the jump to be taken?

```
test \%eax, \%eax
jne 3d<function+0x3d>
```

(a) 1

(b) 0

(c) Any value

(d) no value

Quiz d) solution

d) Which of the following values of `%eax` would cause the jump to be taken?

```
test \%eax, \%eax
jne 3d<function+0x3d>
```

(a) 1

(b) 0

(c) Any value

(d) no value

Solution: a)

Quiz d) solution, Recall

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction

`testl`/`testq` *Src2,Src1*

`testl b,a` like computing `a & b` without setting destination

- Sets condition codes based on value of *Src1* & *Src2*
- Useful to have one of the operands be a mask

ZF set when `a & b == 0`
SF set when `a & b < 0`

Quiz d) solution, Recall

Systems@ETH Zürich

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Quiz d) solution, Recall

Systems@ETH Zürich

- Test %eax, %eax: performs bitwise AND and sets flags. Zeroflag (ZF) is set ⇔ eax is zero
- Jne **checks the zero flag**, and jumps if the zeroflag is **not set**

- For 1: 1&1=1 so it **does not set the zero flag** so here we jump (true)
- For 0: 0&0=0 so it **sets** the zero flag so here we would jump (false)
- Any value: **false** since 0 doesn't work
- No value: **false** since 1 works (in fact, any non 0 value)

Quiz e)

Systems@ETH Zürich

e) What does the `leave` instruction do? Write down an equivalent assembly.

Quiz e) solution

e) What does the `leave` instruction do? Write down an equivalent assembly.

```
mov %ebp,%esp
pop %ebp
```

- I will explain this once we get to stack frames next week, don't worry about it yet

Quiz f)

Systems@ETH Zürich

f) Translate the following C Code to Assembly

```
// input: int x (in %rdi)
// output int y (in %rax)
int func(int x) {
    int y = 0;
    if (x > 0) {
      y = 10;
    }
    y += 5;
    return y;
}
```

Quiz f) solution

```
// input: int x (in %rdi)
// output int y (in %rax)
int func(int x) {
    int y = 0;
    if (x > 0) {
      y = 10;
    }
    y += 5;
    return y;
}
```

Using gcc 4.9.4 -O0:

```
func(int):
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    $0, -4(%rbp)
        cmpl    $0, -20(%rbp)
        jle     .L2
        movl    $10, -4(%rbp)
.L2:
        addl    $5, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

Using gcc 4.9.4 -O1:

```
func(int):
        testl   %edi, %edi
        movl    $10, %edx
        movl    $0, %eax
        cmovg   %edx, %eax
        addl    $5, %eax
        ret
```

Quiz f) solution

Systems@ETH Zürich

- I would personally highly recommend doing these exercises by hand, by yourself at home: you learn a lot about assembly and if you actually understood it

Quiz g)

Systems@ETH Zürich

g) Translate the following C Code to Assembly

```
// input: int x, int y (in %rdi, %rsi)
// output int z (in %rax)

int func(int x, int y) {
    int z = 0;
    while (z <= y) {
      z +=3*(x+1);
    }
    return z;
}
```

Quiz g) solution

```
// input: int x, int y (in %rdi, %rsi)
// output int z (in %rax)

int func(int x, int y) {
    int z = 0;
    while (z <= y) {
      z +=3*(x+1);
    }
    return z;
}
```

Using gcc 4.9.4 -O0:

```
func(int, int):
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        movl    -20(%rbp), %eax
        leal    1(%rax), %edx
        movl    %edx, %eax
        addl    %eax, %eax
        addl    %edx, %eax
        addl    %eax, -4(%rbp)
.L2:
        movl    -4(%rbp), %eax
        cmpl    -24(%rbp), %eax
        jle     .L3
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

Quiz h)

h) Translate the following Assembly code to C

```
func(int, int):
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -4(%rbp), %eax
        cmpl    -8(%rbp), %eax
        jle     .L2
        movl    -4(%rbp), %eax
        jmp     .L3
.L2:
        movl    -8(%rbp), %eax
.L3:
        popq    %rbp
        ret
```

```
int func(int x, int y) {

}
```

Hint: Where are the arguments located?

Quiz h) solution

Systems@ETH Zürich

```
int func(int x, int y) {
  if (x > y) {
    return x;
  } else {
    return y;
  }
}
```

Quiz b)-e)

b)

- c) is the correct answer

c)

- a) Valid: %ebx = 4*(%eax)
- b) Invalid: 15 is a memory address not intermediate!
- c) Valid: store the content of %eax to address 655

d)

- a) is the correct answer

e) mov %ebp, %esp
pop %ebp

Quiz f)

Systems@ETH Zürich

C Code

```
1.  // input: int x (in %rdi)
2.  // output int y (in %rax)

1.  int func(int x) {
2.    int y = 0;
3.    if (x > 0) {
4.      y = 10;
5.    }
6.    y += 5;
      return y;
7.  }
```

Assembly (gcc 4.9.4 with –O0)

```
1.  pushq %rbp,
2.  movq %rsp, %rbp
3.  movl %edi, -20(%rbp)
4.  movl $0, -4(%rbp)
5.  cmpl $0, -20(%rbp)
6.  jle .L2
7.  movl $10, -4(%rbp)
.L2:
8. addl $5, -4(%rbp)
9. movl -4(%rbp), %eax
10.popq %rbp
11.ret
```

Quiz f)

Systems@ETH Zürich

C Code

```
1.  // input: int x (in %rdi)
2.  // output int y (in %rsi)

3.  int func(int x) {
4.    int y = 0;
5.    if (x > 0) {
6.      y = 10;
7.    }
8.    y += 5;
      return y;
9.  }
```

Assembly (gcc 4.9.4 with –O1)

```
1.  testl %edi, %edi
2.  movl $10, %edx
3.  movl $0, %eax
4.  cmovg %edx, %eax
5.  addl $5, %eax
6.  ret
```

Quiz g)

C Code

```
// input: int x, int y
//        (in %rdi, %rsi)
// output: int z (in %rax)

1.  int func(int x, int y) {
2.    int z = 0;
3.    while (z <= y) {
4.      z += 3*(x+1);
5.    }
6.    return z;
7.  }
```

Assembly (gcc 4.9.4 with –O0)

```
1.    pushq %rbp
2.    movq %rsp, %rbp
3.    movl %edi, -20(%rbp)
4.    movl %esi, -24(%rbp)
5.    movl $0, -4(%rbp)
6.    jmp .L2
.L3:
7.    movl -20(%rbp), %eax
8.    leal 1(%rax), %edx
9.    movl %edx, %eax
10.   addl %eax, %eax
11.   addl %edx, %eax
12.   addl %eax, -4(%rbp)
.L2:
13.   movl -4(%rbp), %eax
14.   cmpl -24(%rbp), %eax
15.   jle .L3
16.   movl -4(%rbp), %eax
17.   popq %rbp
18.   ret
```

Quiz g)

Systems@ETH Zürich

C Code

```
// input: int x, int y
//        (in %rdi, %rsi)
// output: int z (in %rax)

1.  int func(int x, int y) {
2.    int z = 0;
3.    while (z <= y) {
4.      z += 3*(x+1);
5.    }
6.    return z;
7.  }
```

Assembly (gcc 4.9.4 with –O1)

```
1.  testl %esi, %esi
2.  js .L4
3.  leal 3(%rdi,%rdi,2),
    %edx
4.  movl $0, %eax
.L3:
5.  addl %edx, %eax
6.  cmpl %eax, %esi
7.  jge .L3
8.  rep ret
.L4:
9.  movl $0, %eax
10. ret
```

Quiz g)

Systems@ETH Zürich

C Code

```
// input: int x, int y
//        (in %rdi, %rsi)
// output: int z (in %rax)

1.  int func(int x, int y) {
2.    int z = 0;
3.    while (z <= y) {
4.      z += 3*(x+1);
5.    }
6.    return z;
7.  }
```

Assembly (gcc 4.9.4 with –Os)

```
1.  incl %edi
2.  xorl %eax, %eax
3.  leal (%rdi,%rdi,2),
    %edi
.L2:
4.  cmpl %esi, %eax
5.  jg .L5
6.  addl %edi, %eax
7.  jmp .L2
.L5:
8.  ret
```

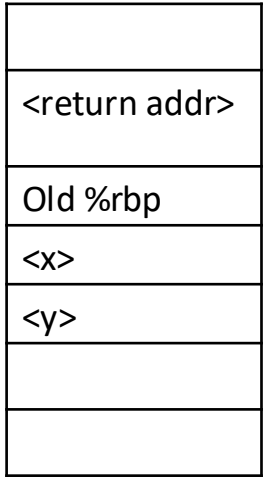
Quiz h)

Assembly

```
func(int, int):
1.  pushq %rbp
2.  movq %rsp, %rbp
3.  movl %edi, -4(%rbp)
4.  movl %esi, -8(%rbp)
5.  movl -4(%rbp), %eax
6.  cmpl -8(%rbp), %eax
7.  jle .L2
8.  movl -4(%rbp), %eax
9.  jmp .L3
.L2:
9.  movl -8(%rbp), %eax
.L3:
10. popq %rbp ret
```

C Code

```
1.  int func(int x, int y) {
2.    if (x > y) {
3.      return x;
4.    } else {
5.      return y;
6.    }
7.  }
```



lea is special...

Only lea operates on memory addresses, everything else on memory values

```
1. leaq 8(%rax, %rdx, 4), %rdi
2.  addq 8(%rax, %rdx, 4), %rdi
```

addq will load the **value at that address**

Check your registers

Make sure instruction and register size match!

```
1. movl $1337, %rcx
```

```
1. asm.s: Assembler messages:
2. asm.s:28: Error: incorrect register `%rcx' used
with `l' suffix
```

Check your registers

Make sure instruction and register size match!

```
1. movl $1337, %rcx
```


```
1. asm.s: Assembler messages:
2. asm.s:28: Error: incorrect register `%rcx' used
with `l' suffix
```

Enjoy the rest of your week!