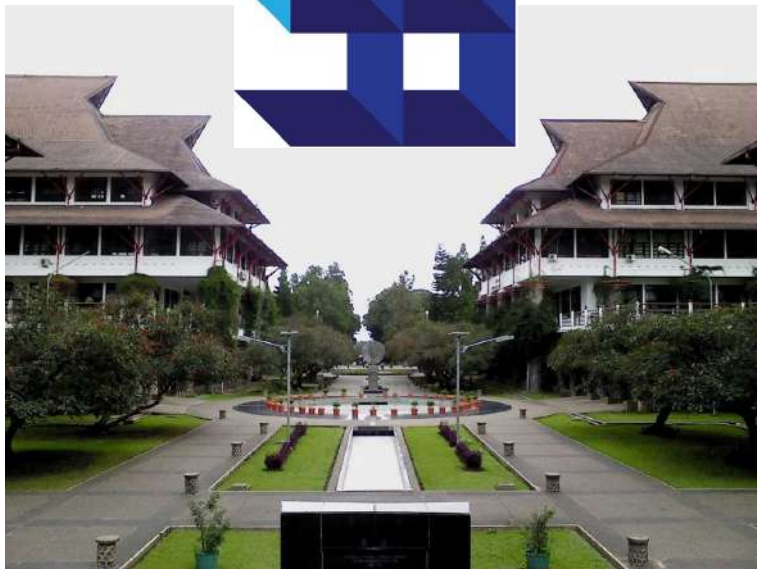


Modul 5

Intel X86-64 Bit Machine

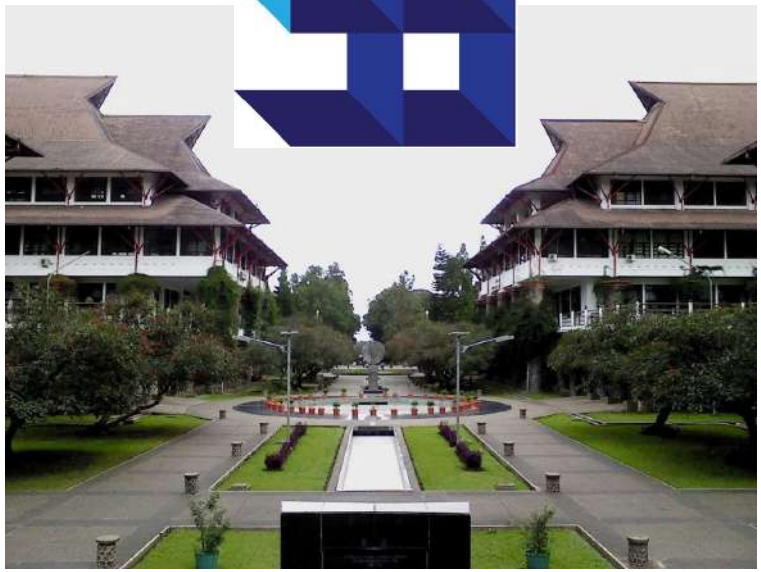
Machine-Level Programming

EL3011 Arsitektur Sistem Komputer

STEI - Institut Teknologi Bandung

Contents:

This module adopted from 15-213 Introduction to Computer Systems Lecture, Carnegie Mellon University, 2020



STEI - Institut Teknologi Bandung

Modul 5. Intel X86-64 Bit Machine

5.1. Intel X86 Evolution

EL3011 Arsitektur Sistem Komputer

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- x86 is a Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
- Compare: Reduced Instruction Set Computer (RISC)
  - RISC: *very few* instructions, with *very few* modes for each
  - RISC can be quite fast (but Intel still wins on speed!)
  - Current RISC renaissance (e.g., ARM, RISC V), especially for low-power

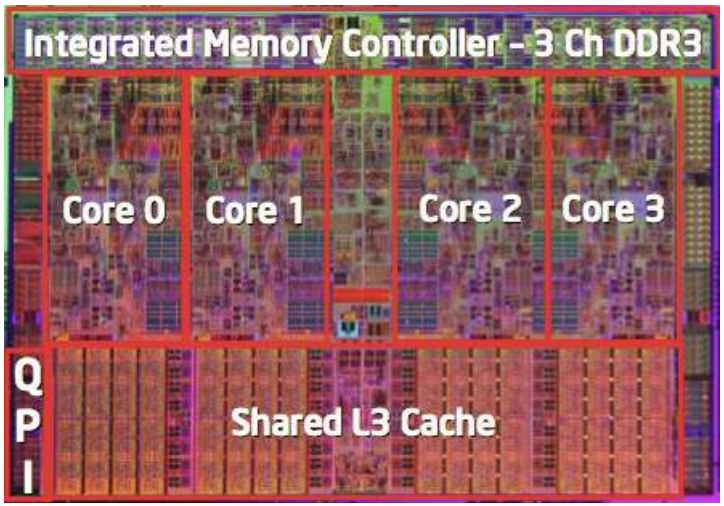
Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
• 8086	1978	29K	5-10
• First 16-bit Intel processor. Basis for IBM PC & DOS			
• 1MB address space			
• 386	1985	275K	16-33
• First 32-bit Intel processor , referred to as IA32			
• Added “flat addressing”, capable of running Unix			
• Pentium 4E	2004	125M	2800-3800
• First 64-bit Intel x86 processor, referred to as x86-64			
• Core 2	2006	291M	1060-3333
• First multi-core Intel processor			
• Core i7	2008	731M	1600-4400
• Four cores			

Intel x86 Processors

- Machine Evolution

		Transistor
• 386	1985	0.3M
• Pentium	1993	3.1M
• Pentium/MMX	1997	4.5M
• PentiumPro	1995	6.5M
• Pentium III	1999	8.2M
• Pentium 4	2000	42M
• Core 2 Duo	2006	291M
• Core i7	2008	731M
• Core i7 Skylake	2015	1.9B



- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

Intel x86 Processors, cont.

- Past Generations

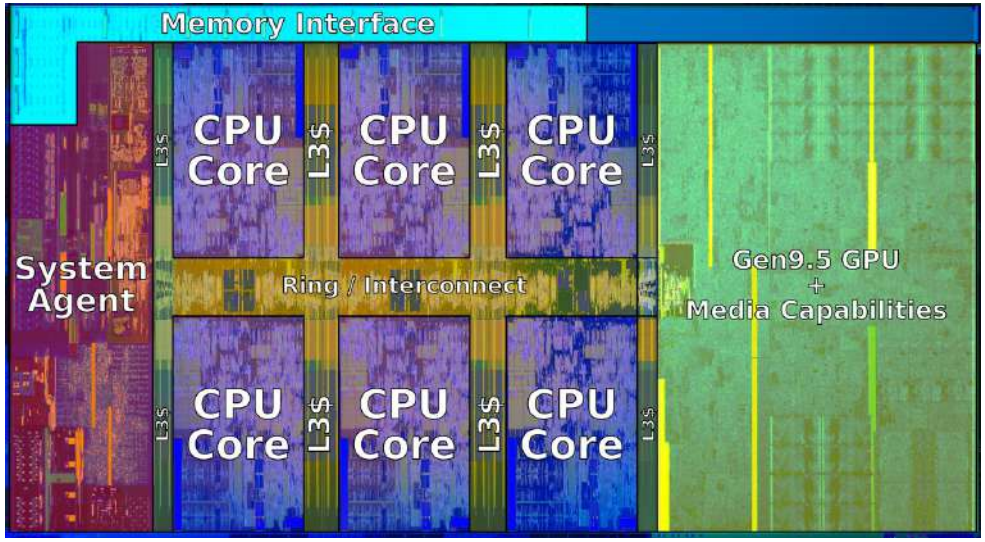
		Process technology
• 1st Pentium Pro	1995	600 nm
• 1st Pentium III	1999	250 nm
• 1st Pentium 4	2000	180 nm
• 1st Core 2 Duo	2006	65 nm

- Recent & Upcoming Generations

1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
9.	Cannon Lake	2018	10 nm
10.	Ice Lake	2019	10 nm
11.	Tiger Lake?	2020	10 nm

Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

2018 State of the Art: Coffee Lake



- Mobile Model: Core i7
  - 2.2-3.2 GHz
  - 45 W

- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W

- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W

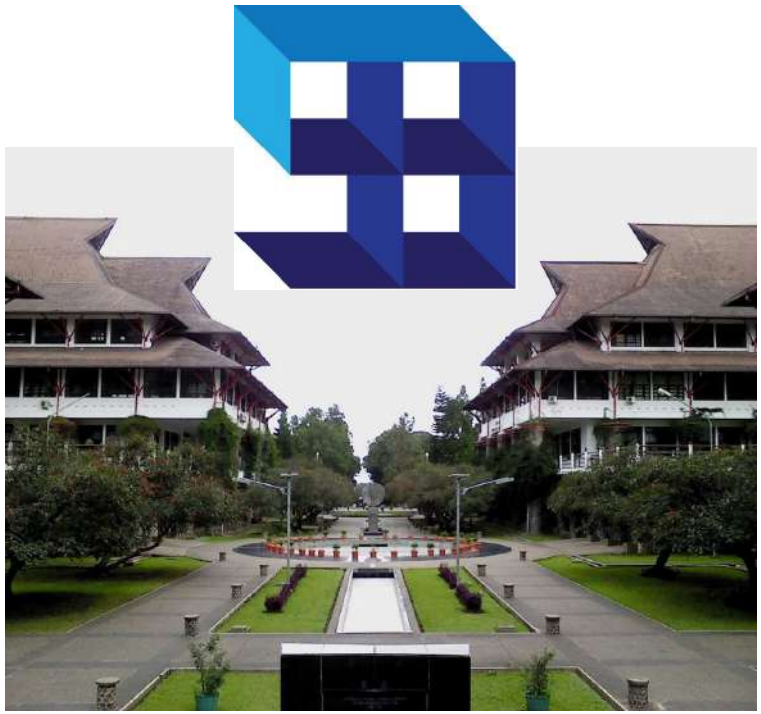
x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- Recent Years
  - Intel got its act together
    - 1995-2011: Lead semiconductor “fab” in world
    - 2018: #2 largest by $$ (#1 is Samsung)
    - 2019: reclaimed #1
  - AMD fell behind
    - Relies on external semiconductor manufacturer Global Foundaries
    - ca. 2019 CPUs (e.g., Ryzen) are competitive again

Intel's 64-Bit History

- 2001: Intel Attempts Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium, AKA "Itanic")
  - Executes IA32 code only as legacy
  - Performance disappointing
- 2003: AMD Steps in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- Virtually all modern x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

Next segment …. Data Types



Modul 5. Intel X86-64 Bit Machine

5.2. Data Types

EL3011 Arsitektur Sistem Komputer

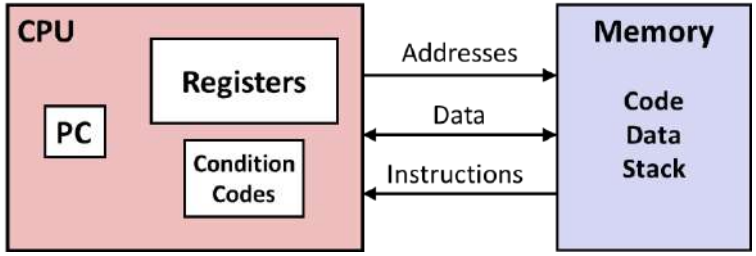
STEI - Institut Teknologi Bandung

Levels of Abstraction

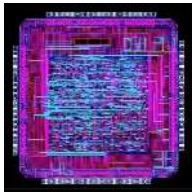
C programmer

```
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```

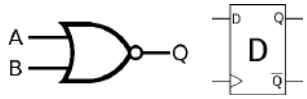
Assembly programmer



Computer designer



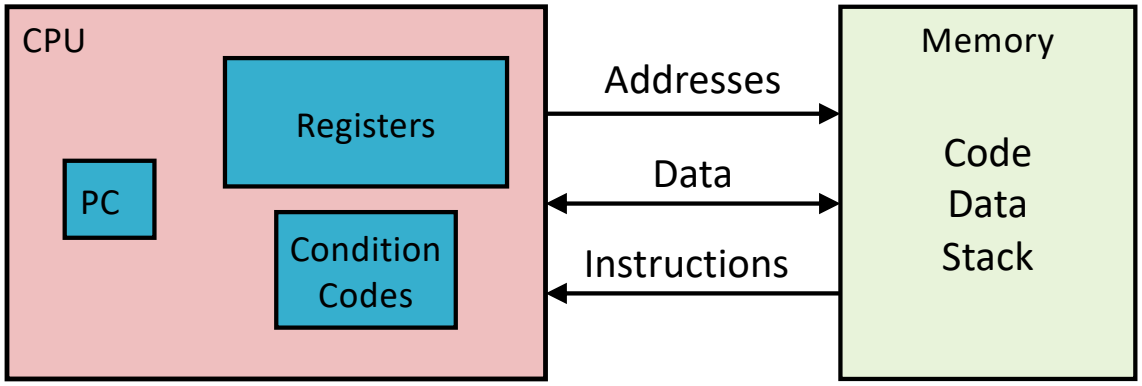
Gates, clocks, circuit layout, …



Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing correct machine/assembly code
  - Examples: instruction set specification, registers
  - **Machine Code**: The byte-level programs that a processor executes
  - **Assembly Code**: A text representation of machine code
- Microarchitecture: Implementation of the architecture
  - Examples: cache sizes and core frequency
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - RISC V: New open-source ISA

Assembly/Machine Code View



Programmer-Visible State

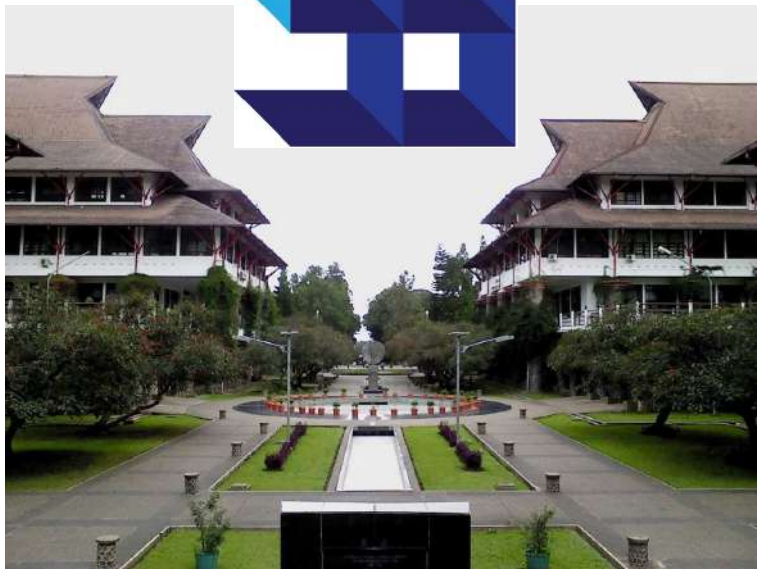
- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- (SIMD vector data types of 8, 16, 32 or 64 bytes)
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Next segment …. Processor registers



STEI - Institut Teknologi Bandung

Modul 5. Intel X86-64 Bit Machine

5.3. Registers

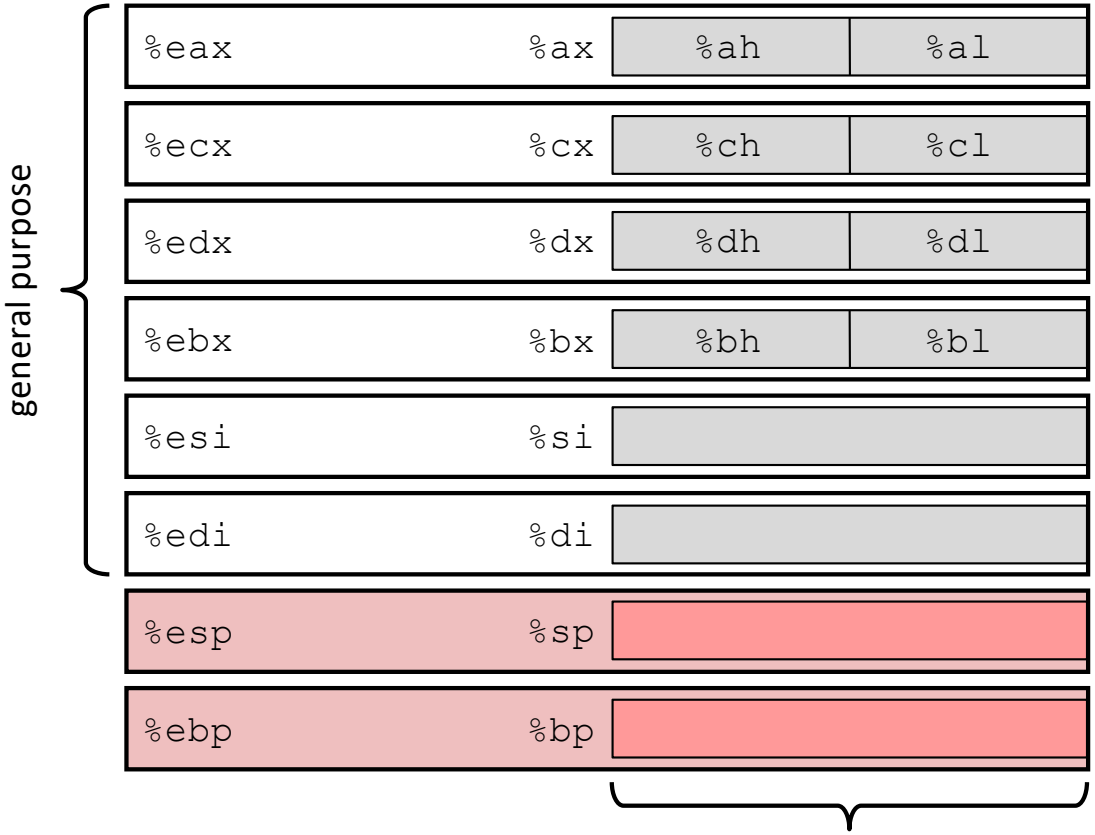
EL3011 Arsitektur Sistem Komputer

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers



Origin
(mostly obsolete)

accumulate

counter

data

base

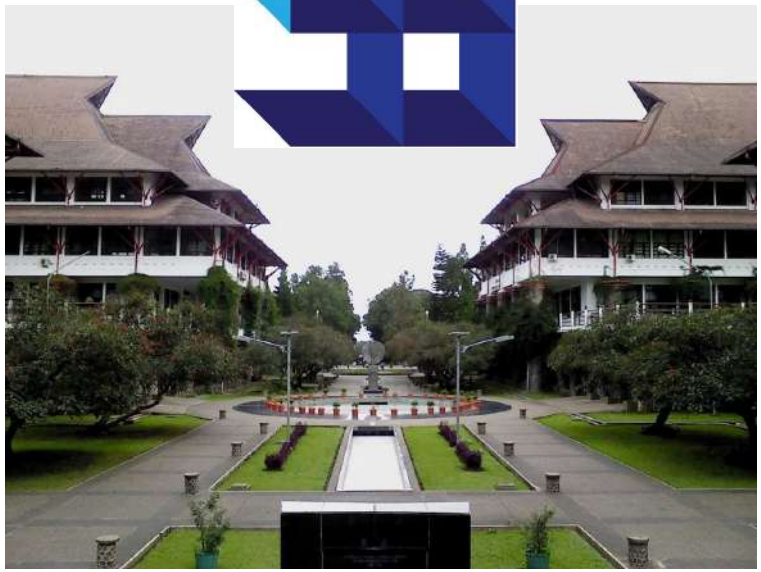
source index

destination index

stack pointer

base pointer

Next segment …. Processor operations



Modul 5. Intel X86-64 Bit Machine

5.4. Processor Operations

EL3011 Arsitektur Sistem Komputer

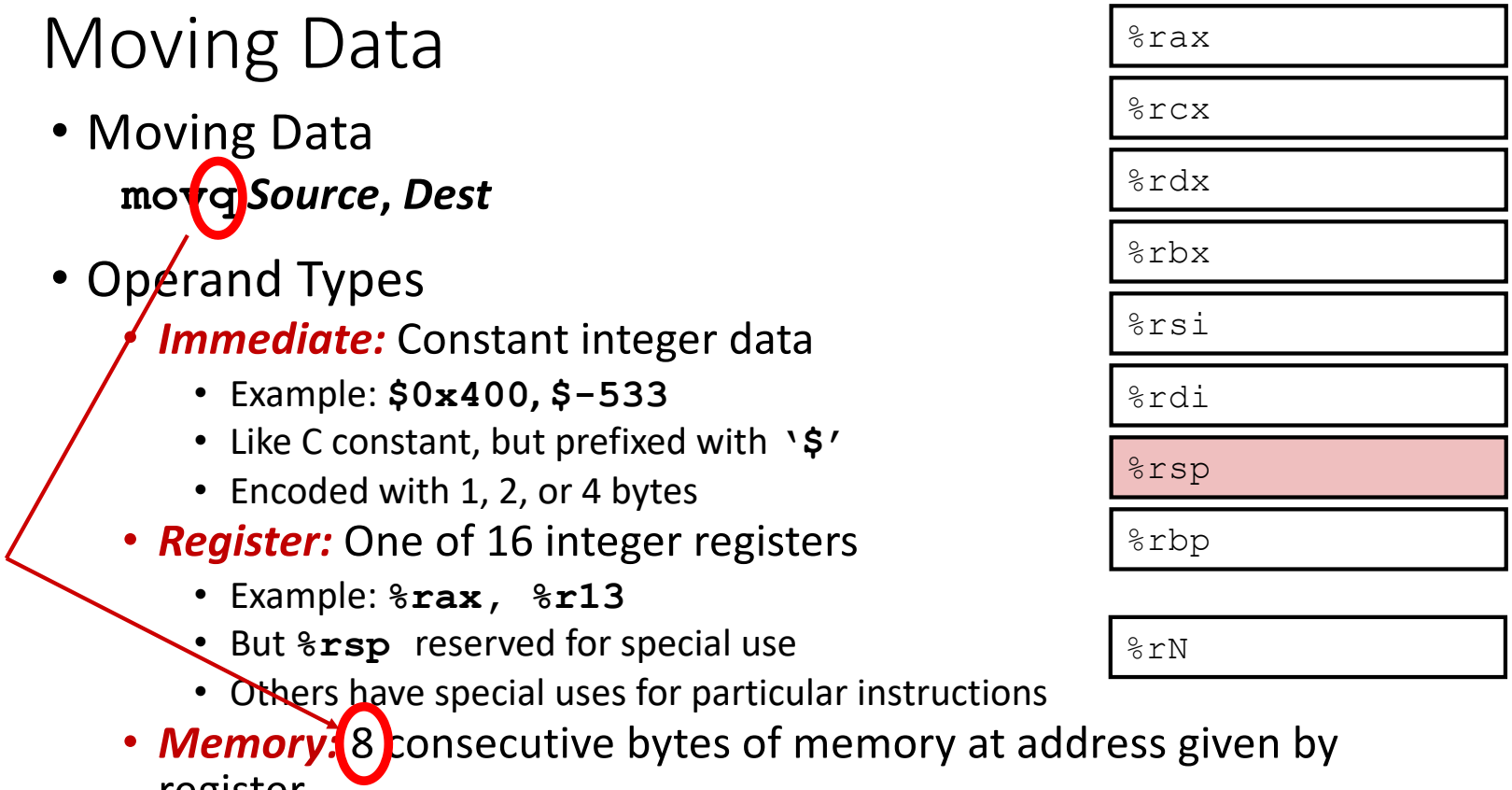
STEI - Institut Teknologi Bandung

Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

Moving Data

- Moving Data
  `movq` ***Source, Dest***
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "addressing modes"



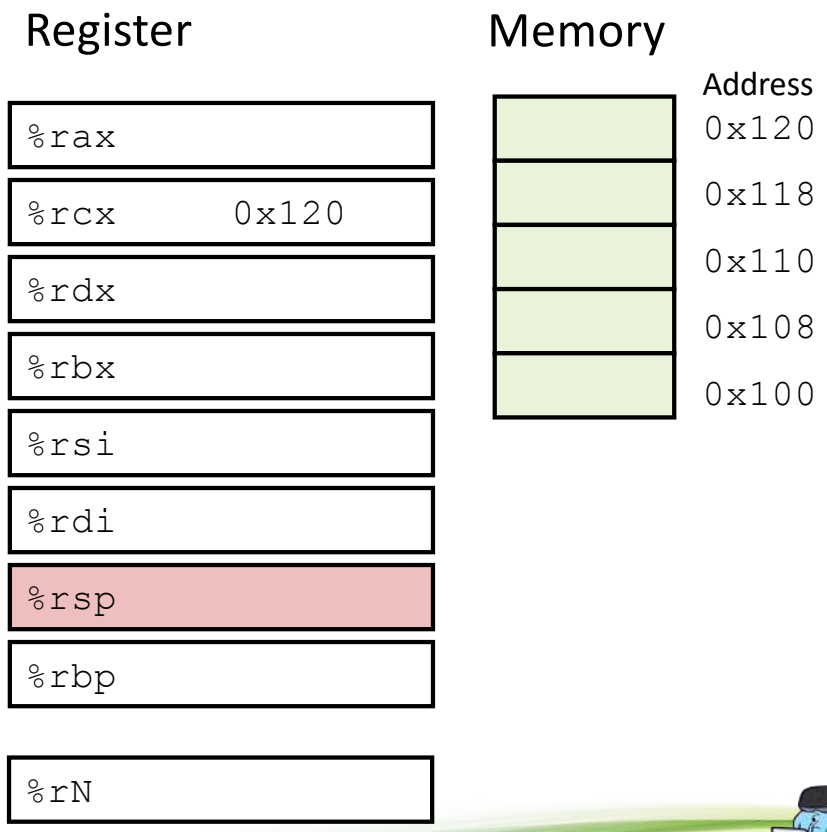
`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x108,%rax`	`temp = 0x108;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

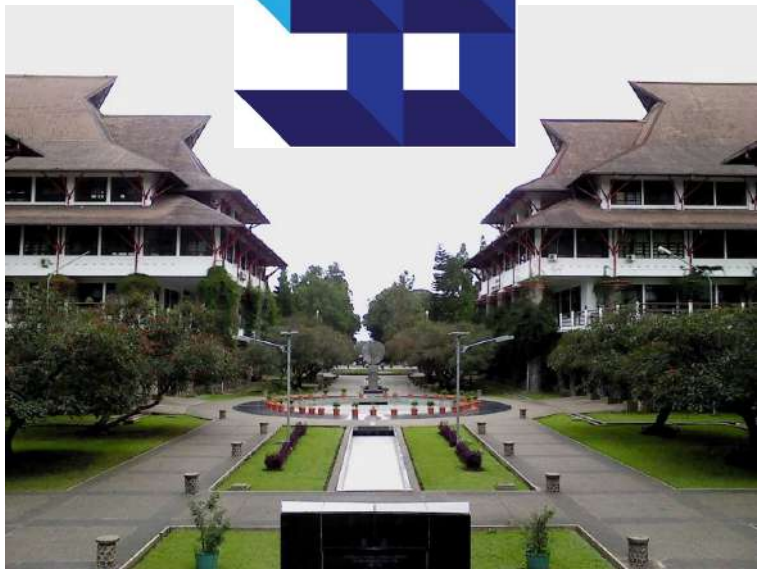
Cannot do memory-memory transfer with a single instruction

movq Operand Combinations

```
movq $0x108,%rax

movq $-147,(%rax)

movq %rax,%rdx

movq %rax,(%rcx)

movq (%rax),%rdx
```



Next segment …. more **`mov`** operations



Modul 5. Intel X86-64 Bit Machine

5.5. mov Operation

EL3011 Arsitektur Sistem Komputer

STEI - Institut Teknologi Bandung

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

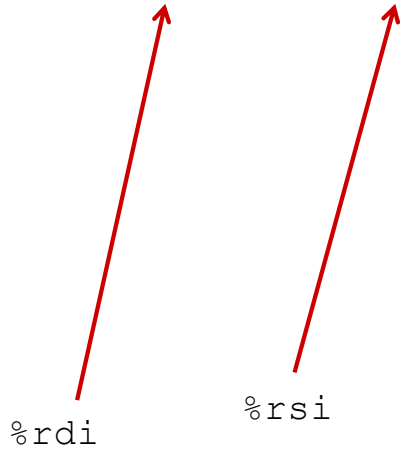
```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Example of Simple Addressing Modes

```
void whatAmI(<type> a, <type> b)
{
      ????
}
```




```
whatAmI:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

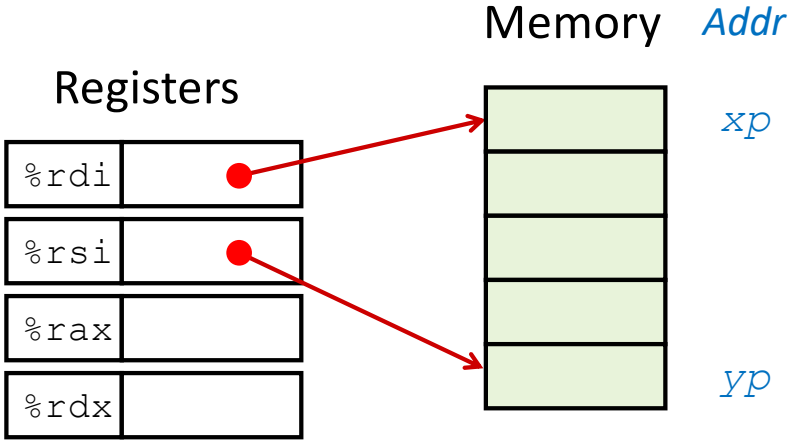
Example of Simple Addressing Modes

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Understanding `swap()`

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding `swap()`

Registers

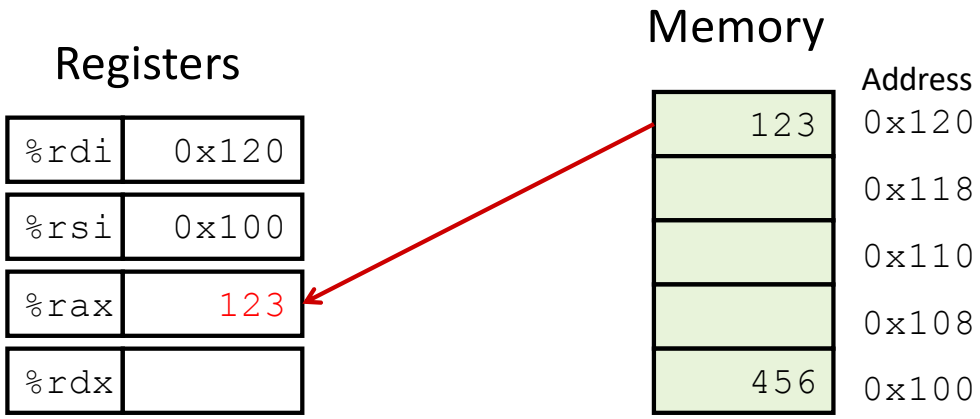
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

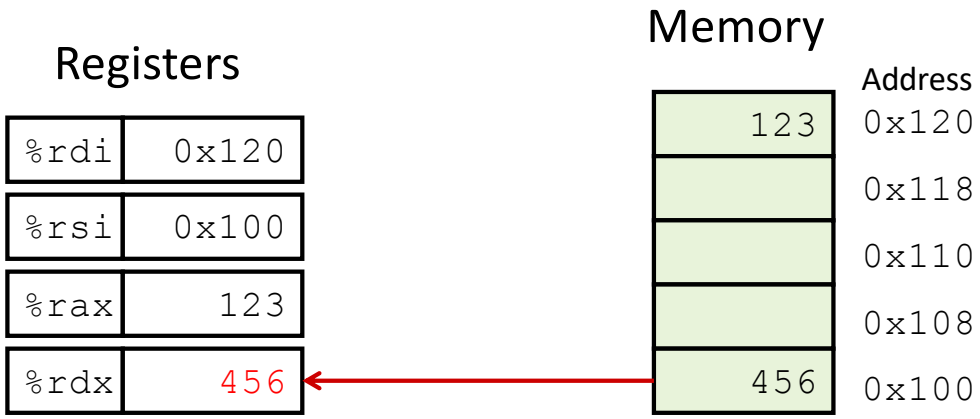
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `swap()`



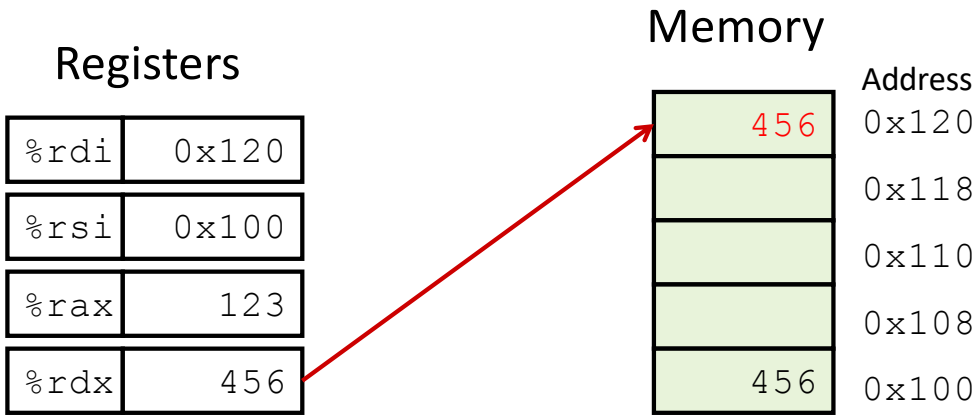

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `swap()`



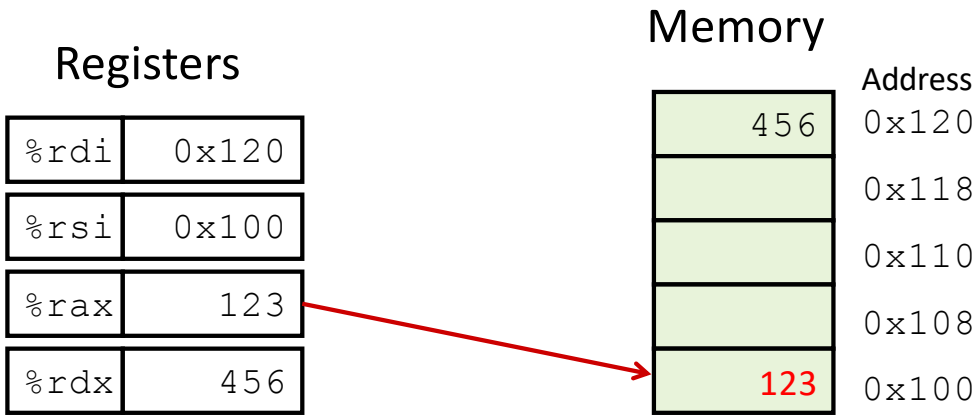

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `swap()`

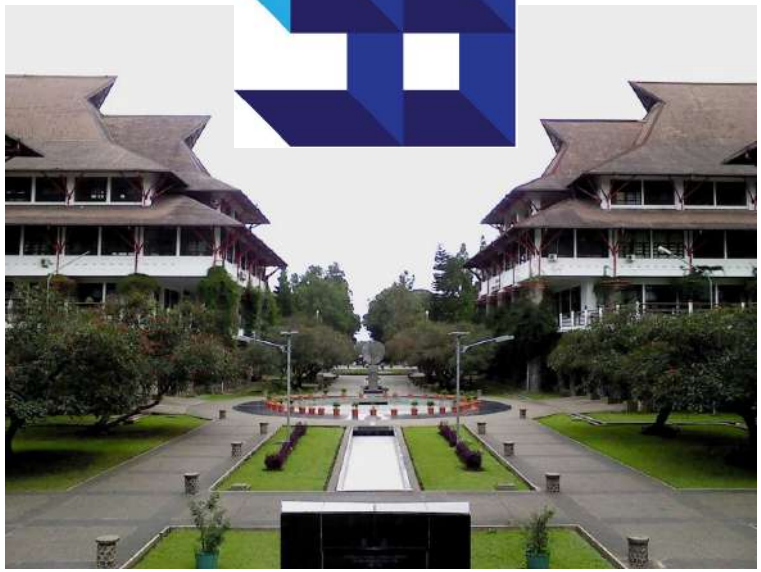



```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `swap()`




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```



STEI - Institut Teknologi Bandung

Modul 5. Intel X86-64 Bit Machine

5.6. Addressing Modes

EL3011 Arsitektur Sistem Komputer

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Complete Memory Addressing Modes

- Most General Form

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

  - D: Constant “displacement” 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8

Complete Memory Addressing Modes

- Most General Form

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

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - 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]]

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

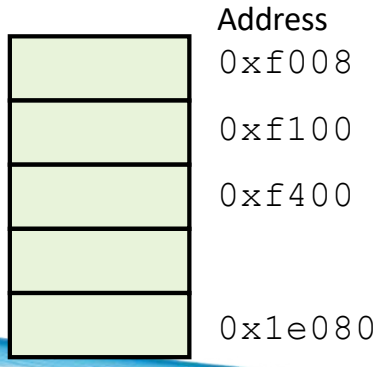
Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

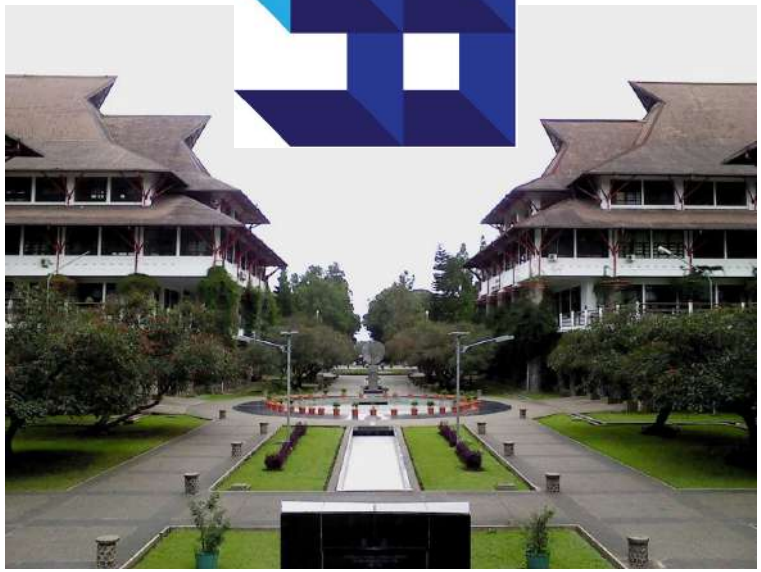
Register

%rdx	0xf000
%rcx	0x0100
%rax	
%rbx	

Memory



Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080



STEI - Institut Teknologi Bandung

Modul 5. Intel X86-64 Bit Machine

5.7. Addressing Computation

EL3011 Arsitektur Sistem Komputer

Complete Memory Addressing Modes

- Most General Form

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

  - D: Constant “displacement” 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

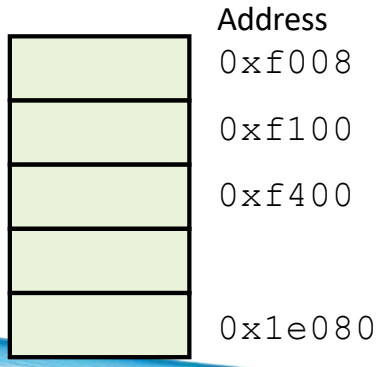
Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

Register

%rdx	0xf000
%rcx	0x0100
%rax	
%rbx	

Memory



Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Address Computation Instruction

- **`leaq`** Src, Dst
  - Src is address mode expression
  - Set Dst to address denoted by expression

- Uses
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax  # t = x+2*x
salq $2, %rax             # return t<<2
```

Address Computation Instruction

- Example: consider %rdx = x

```
movq 7(%rdx,%rdx,4),%rax
```

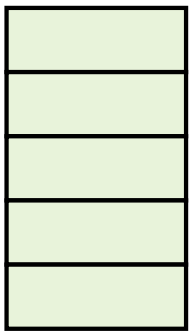
```
leaq 7(%rdx,%rdx,4),%rax
```

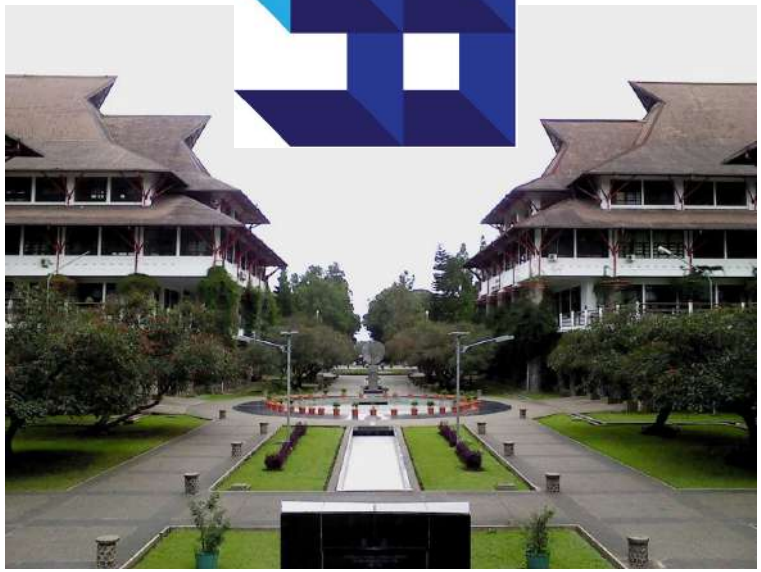
Register

%rax	
%rcx	
%rdx	
%rbx	

Memory

Address





STEI - Institut Teknologi Bandung

Modul 5. Intel X86-64 Bit Machine

5.8. Arithmetic and Logical Operations

EL3011 Arsitektur Sistem Komputer

Some Arithmetic Operations

- Two Operand Instructions:

Format		Computation	
`addq`	Src,Dest	Dest = Dest + Src	
`subq`	Src,Dest	Dest = Dest – Src	
`imulq`	Src,Dest	Dest = Dest * Src	
`shlq`	Src,Dest	Dest = Dest << Src	Synonym: salq
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`xorq`	Src,Dest	Dest = Dest ^ Src	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	

- Watch out for argument order! *Src,Dest*
  (Warning: Intel docs use "op *Dest,Src*")
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest – 1
`negq`	Dest	Dest = – Dest
`notq`	Dest	Dest = ~Dest

- See book for more instructions
  - Depending how you count, there are 2,034 total x86 instructions
  - (If you count all addr modes, op widths, flags, it's actually 3,683)

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - Curious: only used once…

```
1 arith:
2   leaq    (%rdi,%rsi), %rax
3   addq    %rdx, %rax
4   leaq    (%rsi,%rsi,2), %rdx
5   salq    $4, %rdx
6   leaq    4(%rdi,%rdx), %rcx
7   imulq   %rcx, %rax
8   ret
```

```
long arith(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

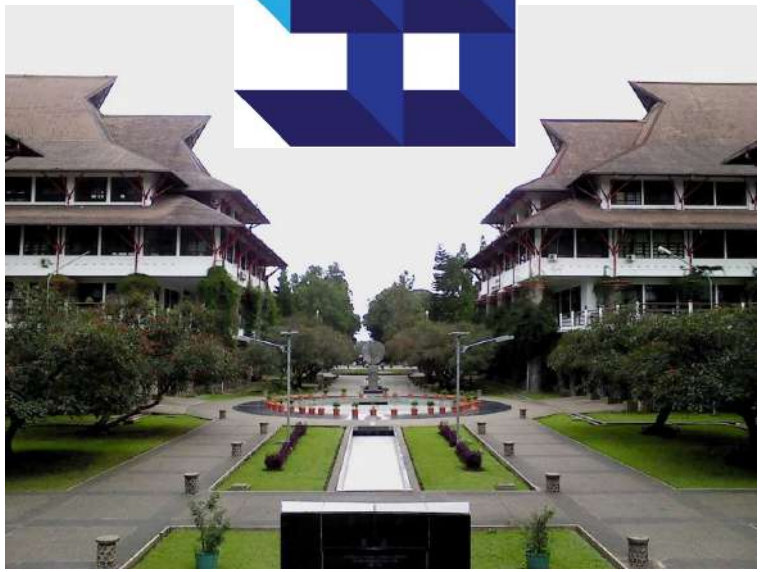
Register	
%rax	
%rcx	
%rdx	z
%rdi	x
%rsi	y

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**, **t4**
%rax	**t1**, **t2**, **rval**
%rcx	**t5**



STEI - Institut Teknologi Bandung

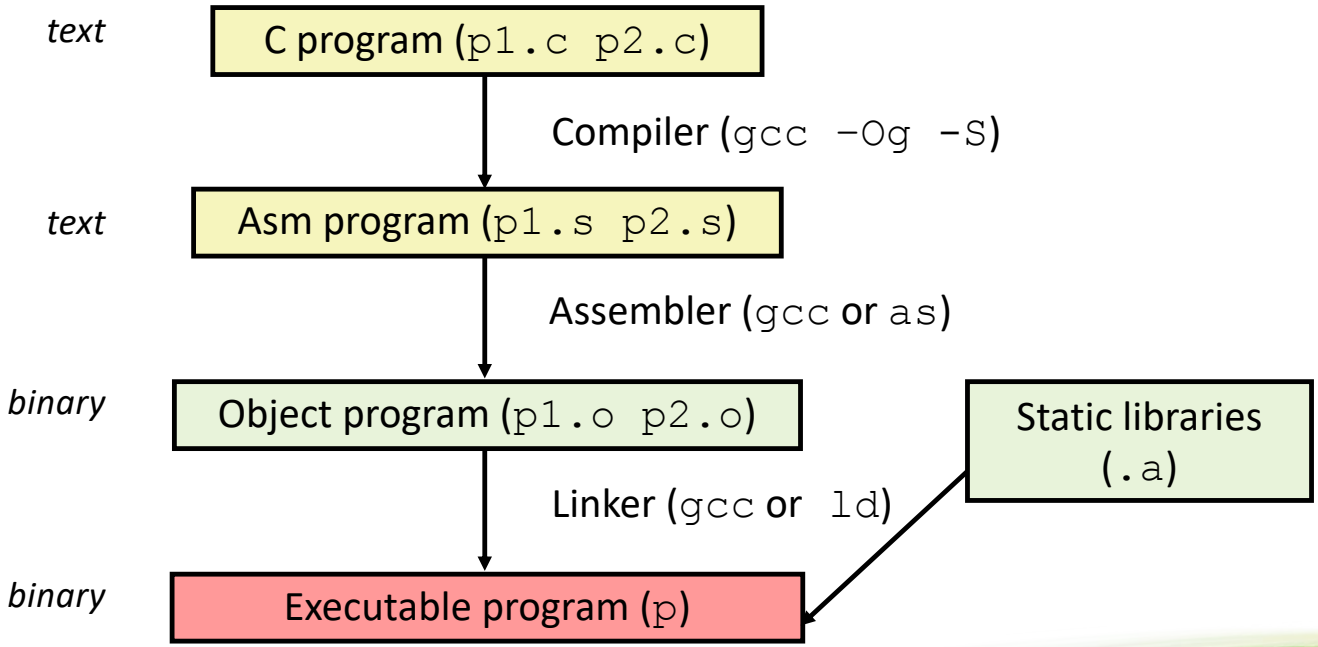
Modul 5. Intel X86-64 Bit Machine

5.9. C Programming, Assembly and Machine Code

EL3011 Arsitektur Sistem Komputer

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc –Og p1.c p2.c –o p`**
  - Use basic optimizations (**`–Og`**) [New to recent versions of GCC]
  - Put resulting binary in file **`p`**



Compiling Into Assembly

C Code (swap.c)

arith.c | swap.c

```
1  void swap (long *xp, long *yp)
2  {
3      long t0 = *xp;
4      long t1 = *yp;
5      *xp = t1;
6      *yp =  t0;
7  }
8
```

Obtain with command

```
gcc -O2 -S swap.c
```

Produces file `swap.s`

Warning: Will get very different results on other machines due to different versions of gcc and different compiler settings.

Generated x86-64 Assembly

Command Prompt

```
C:\Codes>gcc -O2 -S swap.c

C:\Codes>type swap.s
        .file   "swap.c"
        .text
        .p2align 4,,15
        .globl  swap
        .def    swap;   .scl    2;      .type   32;     .endef
        .seh_proc       swap
swap:
        .seh_endprologue
        movl    (%rcx), %eax
        movl    (%rdx), %r8d
        movl    %r8d, (%rcx)
        movl    %eax, (%rdx)
        ret
        .seh_endproc
        .ident  "GCC: (x86_64-posix-seh-rev0, Built by MinGW-W64 p

C:\Codes>
```

Compiling Into Assembly

C Code (arith.c)

arith.c

```c
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = x + t1;
    long t3 = y + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Generated x86-64 Assembly

```
arith:
        .seh_endprologue
        leal    (%rdx,%rdx,2), %eax
        sall    $4, %eax
        leal    4(%rdx,%rax), %eax
        leal    (%rdx,%rcx,2), %edx
        imull   %edx, %eax
        ret
```

Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation