Operating Systems

CSE 231 (Section B)

Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 1: Aug 20 – Aug 27

Introduction

- What is a Computer ?
- What is an Operating System?
- Why does a computer need an Operating System ?

Class Objective: What CSE 231 Would Teach You

- Introductory course to operating systems
  - The basics – blah blah blah.
  - Processes and their siblings.
  - Process synchronization.
    - Threads and thread synchronization.
  - Issues with processes – deadlocks and deadlock detection
  - Memory Management.
  - I/O Management.
  - Storage Management

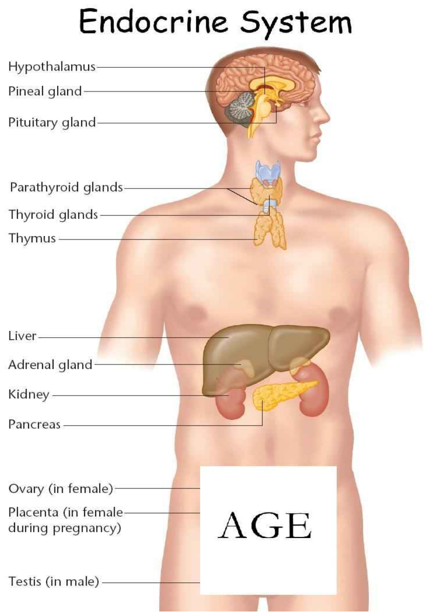
What you should expect at the end of this class

- Pretty conversant in C programming. NO fear of pointers, debuggers, assemblers (A C ninja – as per the TA who taught the refresher module)
- Pretty conversant with Linux internals.
- Compiling kernel.
- Mucking around the kernel
- Strong understanding of processes, threads and the likes.
- Good understanding of memory management as it happens inside the OS.
- Basics of storage management – aka filesystems and how data is stored on disks.
- How a program (aka binary) runs!
- How a computer boots up! – (ps: get ready for an extra credit assignment)

Grade Distribution and Administrativia

- **Grading:**
    - 10% Mid-term exam
    - 10% Final exam
    - 70% Assignments
    - 10% Quiz (In-class)
- **Textbook:**
    - 1. Operating Systems Concepts: A. Silberschatz, P. B. Galvin and G. Gagne, Ninth Edn., 2014, Wiley India Pvt Ltd., ISBN: 973-1-118-06333-0
    - Linux Systems Programming, R. Love, Second Edn, 2013, O'Rielly Media, ISBN-13: 978-1449339531
    - Linux Kernel Development, R. Love, Third Edn., ISBN-13: 978-0672329463
- **Contact email:** sambuddho@iiitd.ac.in
- **Office Hours:** Fri. 1200hrs – 1300hrs (Room No. B503, New Academic Block)
- **TA office hours:** TBA

Exactly What is a Computer



Most Basic Computer aka Turing Machine

- Proposed by Alan Turing (circa. 1936)
- Model consists of infinite tape divided into cells which contain input symbols.
- Head reads the top of the symbol and changes the internal state of the machine when the symbol is something that is not unexpected.
- The machine thus moves to the next state until it reaches a terminal state which makes that the input is invalid…

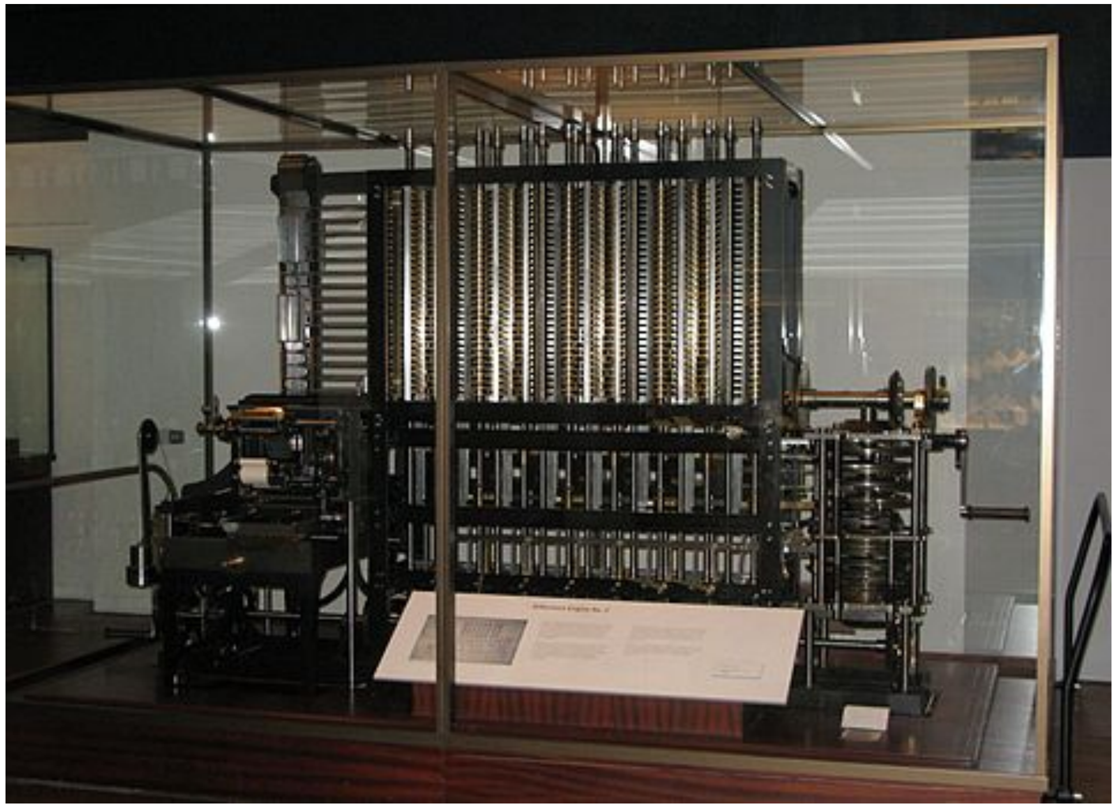
Evolution of Computers

- Abacus – Chinese, Roman and Russian – ancient form of calculator.
  - Relies on a series of beads and their movements to perform calculations.

  - Homework : Read and describe how the Abacus works (would be graded)

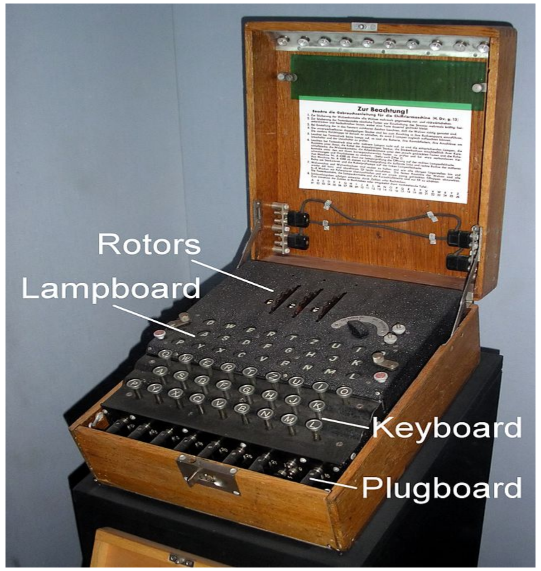
Babbage Analytical Machine

- Mechanical form of the Abacus used for calculating polynomial functions.
- Separation of storage and processing…



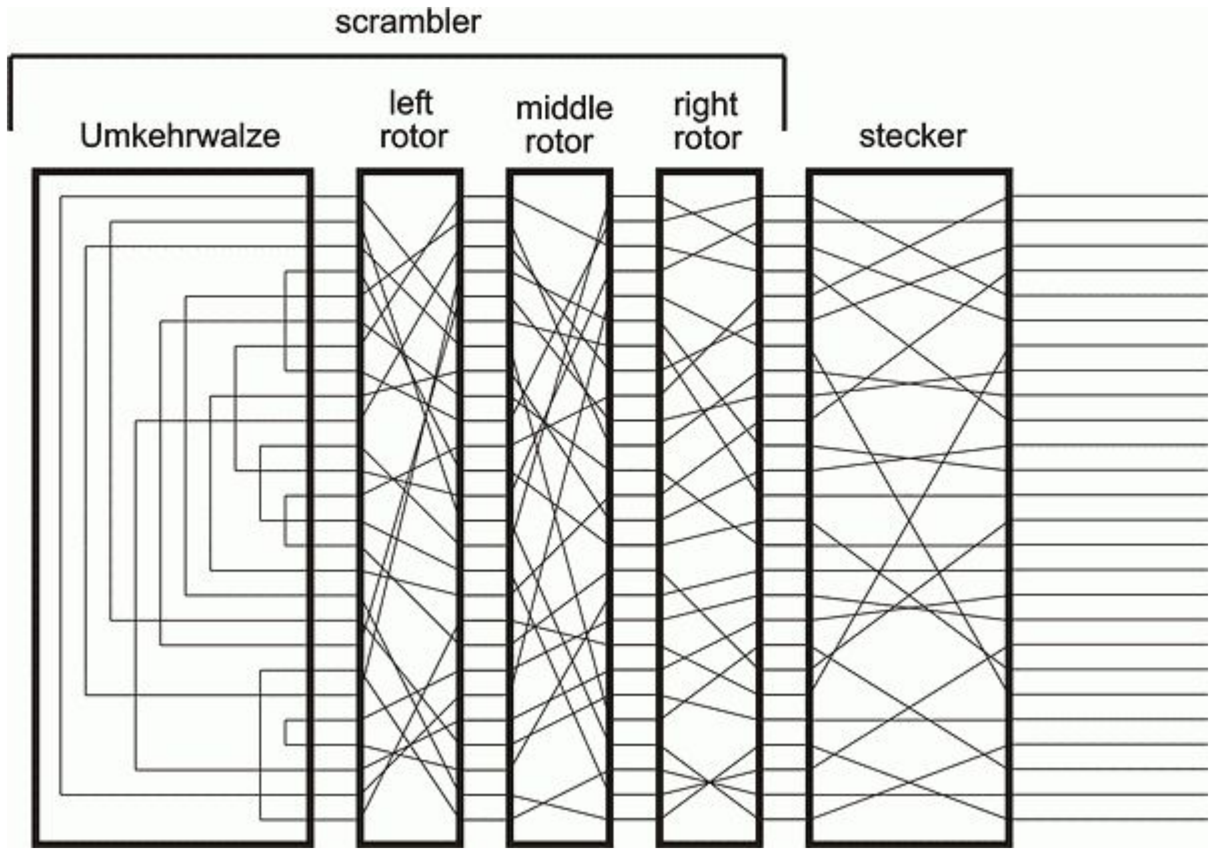
20th Century – Enigma Machine

- **Designed by Arthur Scherbius (Axis Germany) towards end of WW1 to send encrypted messages**



Used during WW2

Inside the Enigma

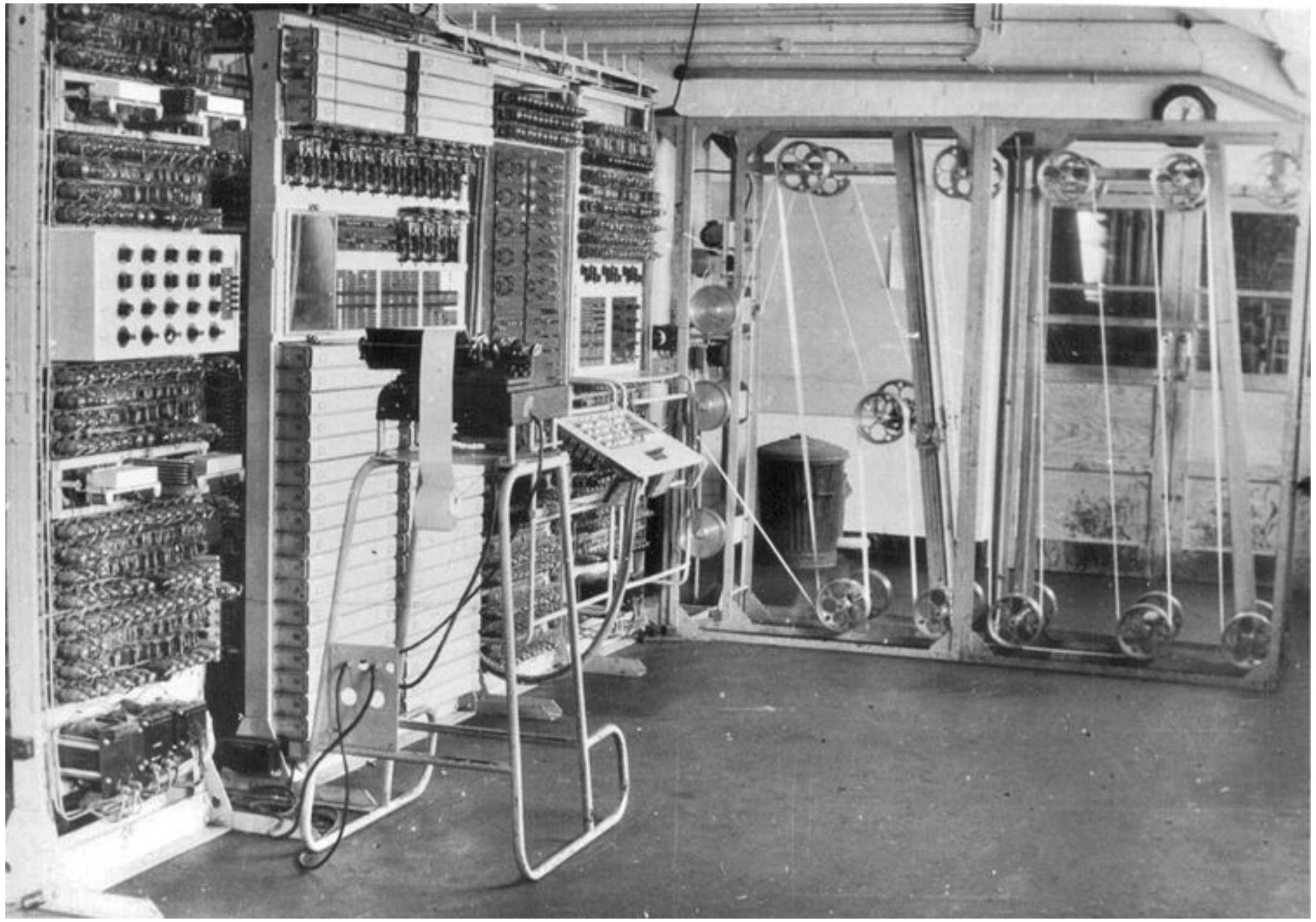


Input

Output (via Glowlamps)

Collosus Machine

- **Designed by Britain (Allied) to break encrypted messages of enigma...**



Post WW2 – **E**lectronic **N**umerical **I**ntegrator and **C**omputer (ENIAC)

-Designed by Univ of Pennsylvania, in Feb 15, 1946.
-General purpose computer designed based on Turing Machine principle

Operation time samples: 10-digit num x 10-digit num took 2800 µsec,

https://www.youtube.com/watch?v=rO2SScF6rrM

Post WW2 – **General Purpose Computers Post 1960s**

-Greatest invention for computing world – Transistor – improvement in size and speed of operation – lower power consumption

- Majority of general purpose mainframes evolved in this era – the PDP (Programmed Data Processors) – PDP 1 .. PDP 16
- Different processing capabilities

- **UNIVAC** (**UNIV**ersal **A**utomatic **C**omputer) – Mid 1960s. Used as general purpose mainframes – quite ahead of their times 36-bit word size (modern Intel x86_64 – 64bit word size).

- Big challenge in increasing word size in computers…

Era of IC based design 1970s

- Jack Kilby (Noble laureate), father of IC semi-conductors.
- Integrated circuits – semi-conductors were ``grown’’ over silicon substrates. Faster than ordinary BJTs.

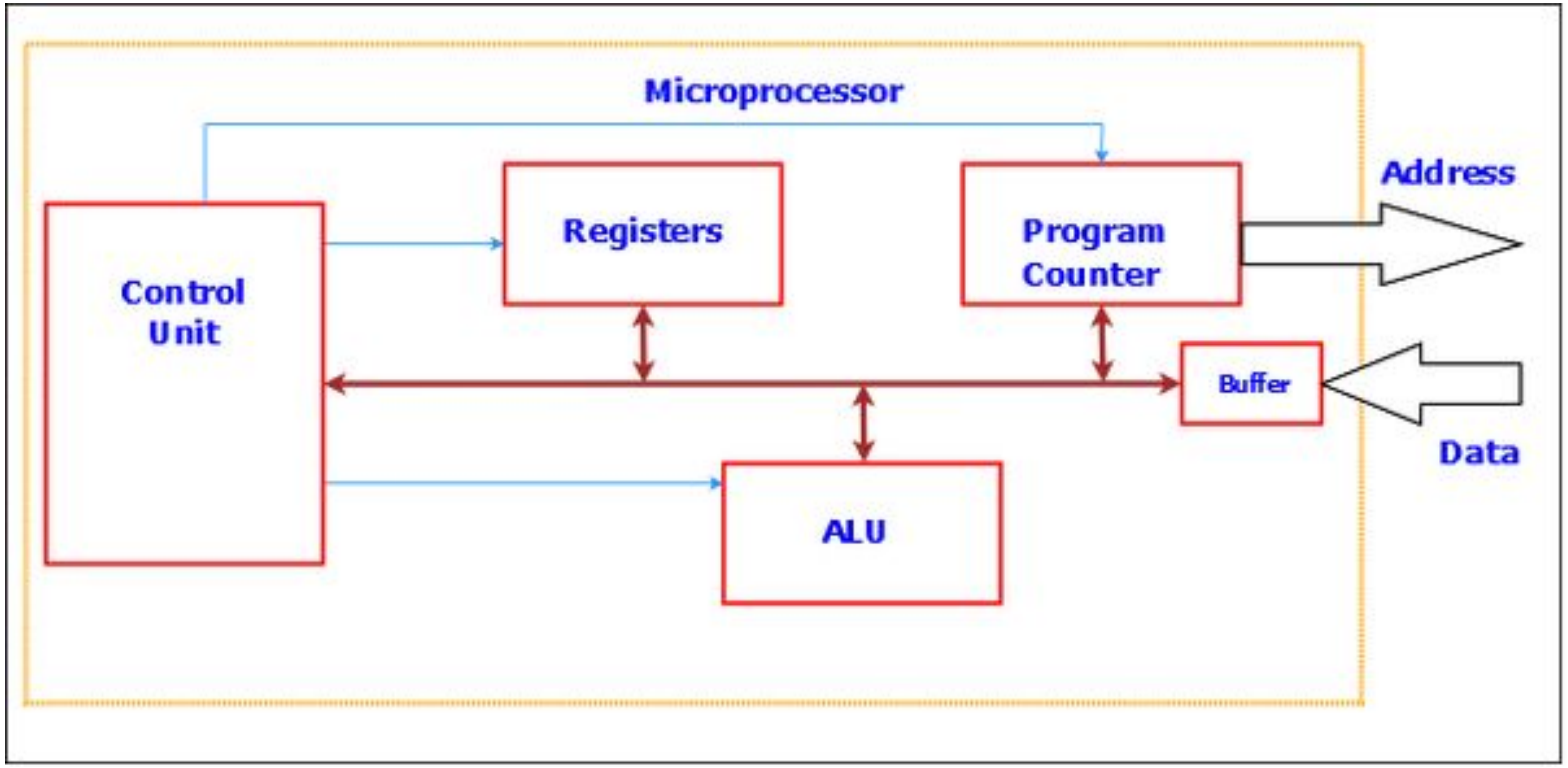
- Many ``fast’’ transistors could be integrated into a single physical ``chip’’
- Beginning of VLSI age..
- Computer chips become even smaller, faster and consume lesser power (overall faster computation speeds).
- ICs had combinations of transistors arranged to form series of logic gates – the building blocks of modern computers – can be used for all kinds of operations but the classical ones are for storage (aka memory).
- More advanced – microcontrollers and microprocessors.

Intel x86 Processors

- Totally dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

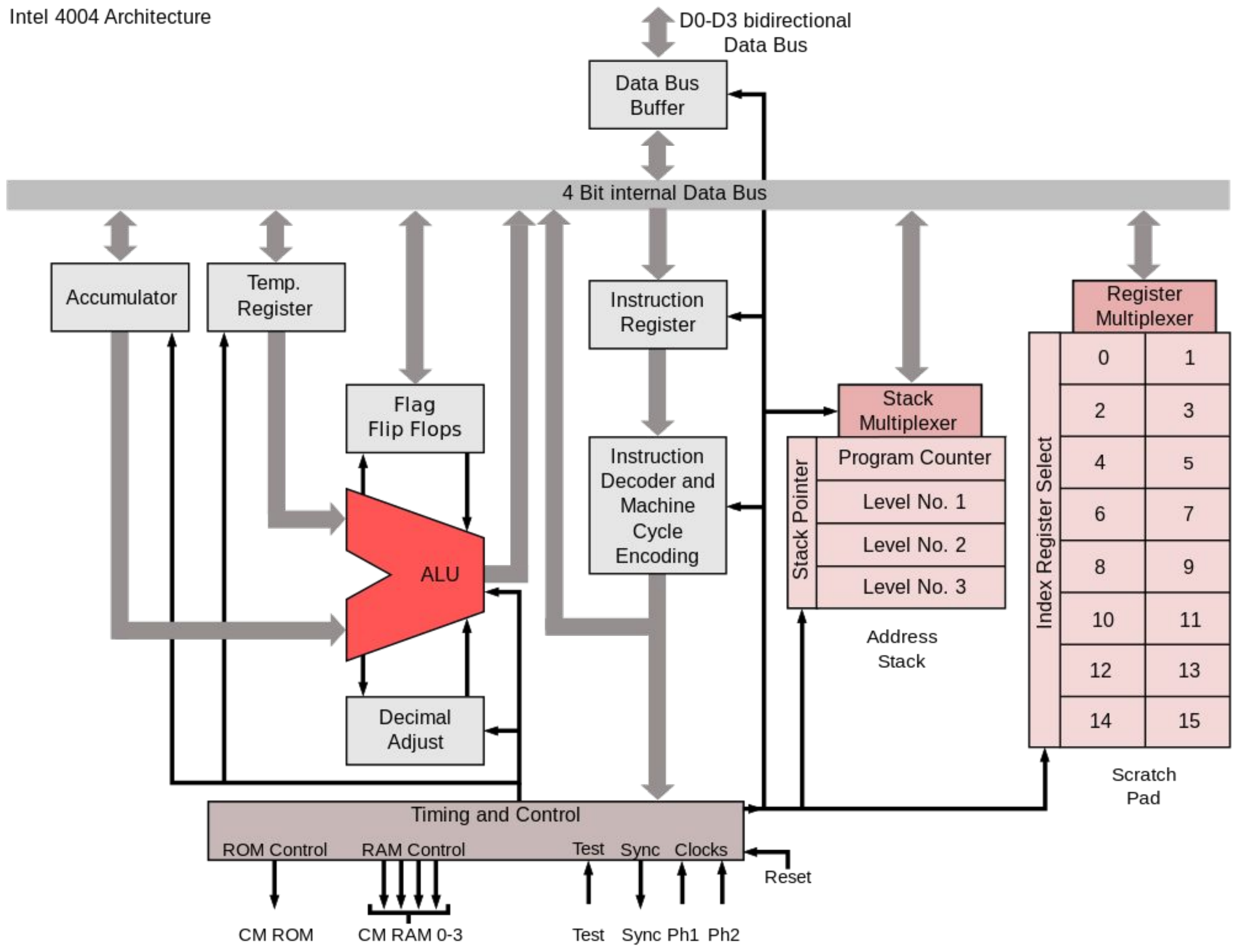
Era of Digital Computers

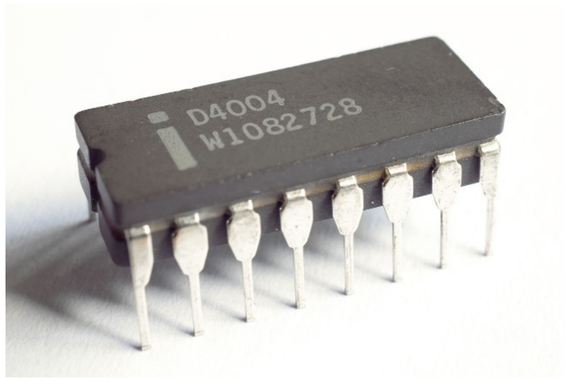
- ICs used to design a microcontrollers and microprocessors.
- Main parts – registers, ALU, control unit, PC, I/O buffers, address bus and data bus



Intel – 1970s and Beyond

-First microprocessor by Intel – 4044 (4-bit microprocessor) – release date: Nov 15, 1971
-The first commercial product to use a microprocessor was the Busicom calculator 141-PF.
- The 4004 was also used in the first microprocessor-controlled pinball game, a prototype produced by Dave Nutting Associates for Bally in 1974





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 4F 2004 125M 2800-3800
  - First 64-bit Intel processor, referred to as x86-64
- Core 2 2006 291M 1060-3500
  - First multi-core Intel processor
- Core i7 2008 731M 1700-3900
  - Four cores

Rise of the Desktops

- Xerox Alto (by Xerox PARC) developed in 1973, the first PC ever designed – used, mouse keyboard etc.
- Closely correlated to the rise of microprocessors.
- Series of systems by IBM using IBM PALM processor, running and interpretive language (BASIC).
- Altair 8800 – spawned several computer manufacturers to start to produce personal computers – Processor : Zilog Z80 or Intel 8085, running CP/M-80 operating systems.

Home computing:

- IBM PC – Intel 8088 microprocessor
- 8 bit microprocessor, booted using floppy disks…
- 5MB HDDs with 640KB of RAM
- PC-DOS operating system.
- Apple Lisa and Macintosh – GUI based interface.
- Motorola 68000 microprocessor with about 128 KB of RAM (upgraded to 512 KB of RAM)
- Newer faster processors evolved post 1990s – Intel 80386/486/Pentium processor – 32-bit
- RAM sizes increased to 16MB ⍰ 32 MB ⍰ 64 MB

Cellular Communication Married to Computers – Smart Phones

-A different revolution was brewing since the 1970s… (??)

-ARPANET – 1969 – connection established

- First message transmitted – “Login”

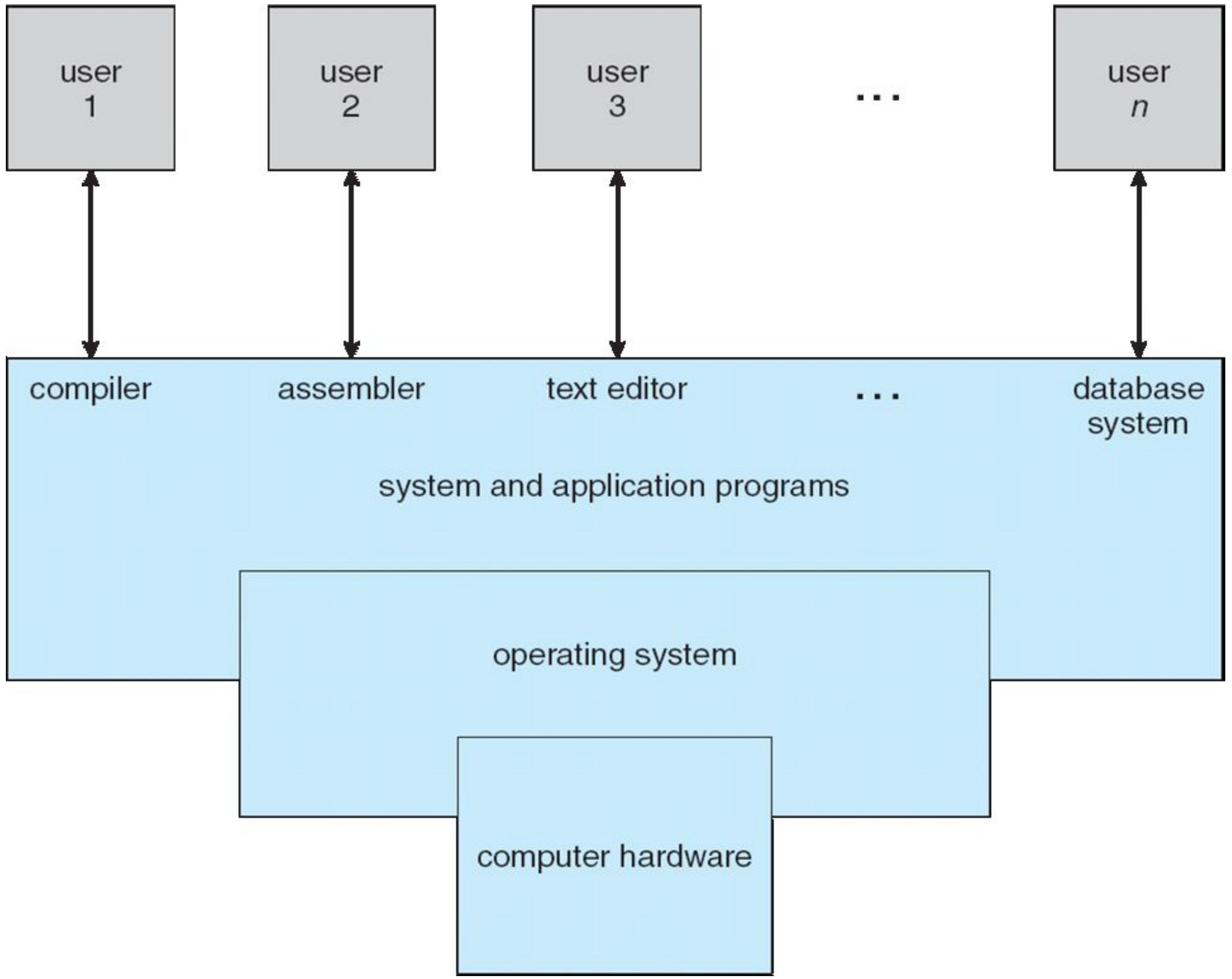
- Wireless communication has been there since 19th century.
- Telephony (1876, Bell)
- Wireless telephony – Cellular communication – 1970s.

- Computers

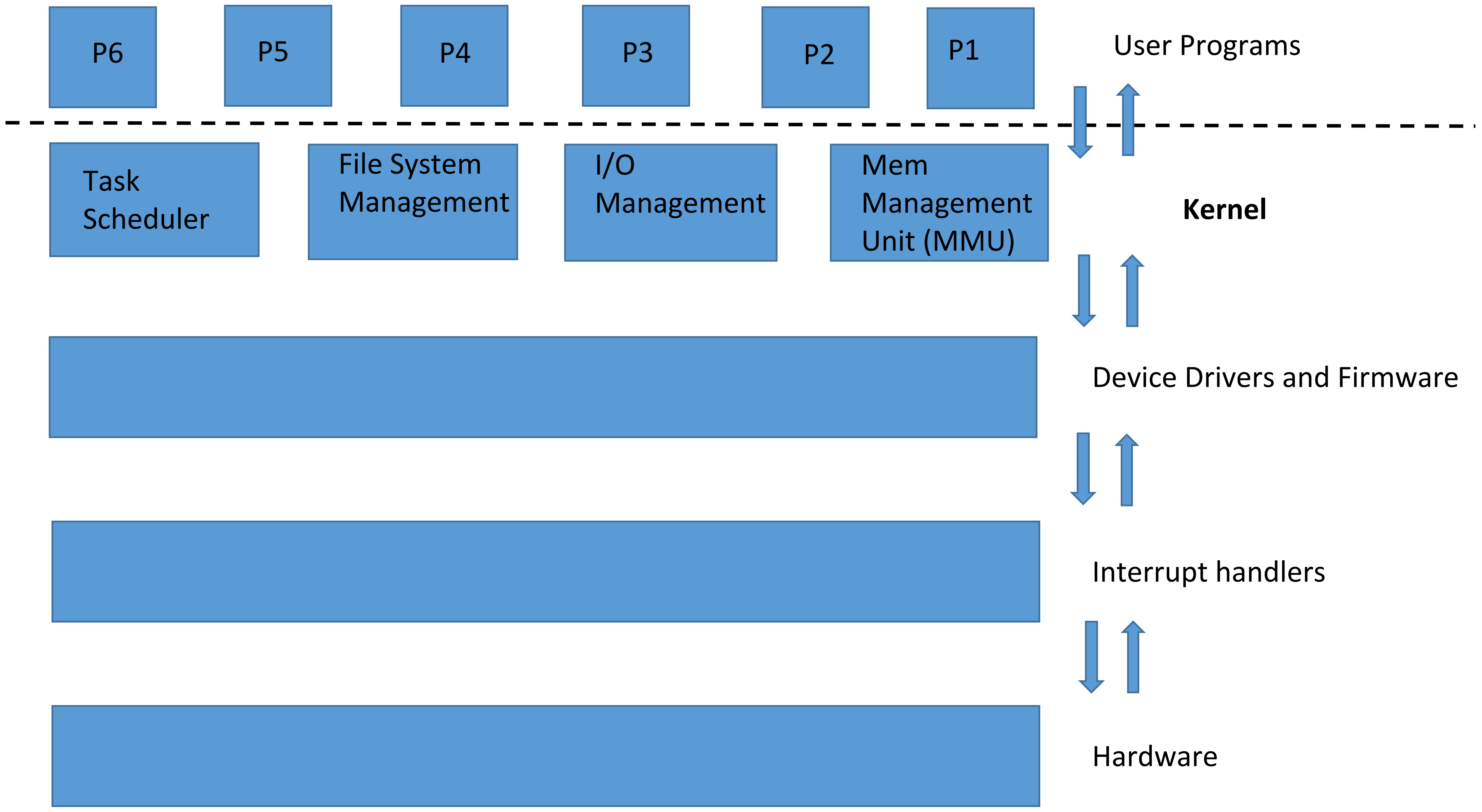
Age of wearables and implantables…

- Microprocessors are now fitted into wearable devices (e.g. watches)
- Watches + smartphones = smart-watches
- Temperature and vital statistics monitoring
- Implantables….
- Often specific purpose (not so generic…)

Components of Computer System



Layered Architecture



Interrupts and Interrupt Handling (Heart of the OS)

- Interrupts:
  - ALL devices send a signal to the CPU whenever they require to ``communicate" with the CPU.
  - CPU is switched to Interrupt Handler program (part of the OS/supplied externally).
  - The Interrupt Handler ``saves'' various information with respect to the running program and it is jumped to a fixed location in memory – the IH routine.
  - The IH routing – binaries (programs) that involve instructions and commands to communicate with the device.
  - The IH routine may either handle the device interrupt and jump back to executing the saved task
  - In case of task scheduler timer clock interrupt the CPU scheduler handles the tasks/process and handles it.

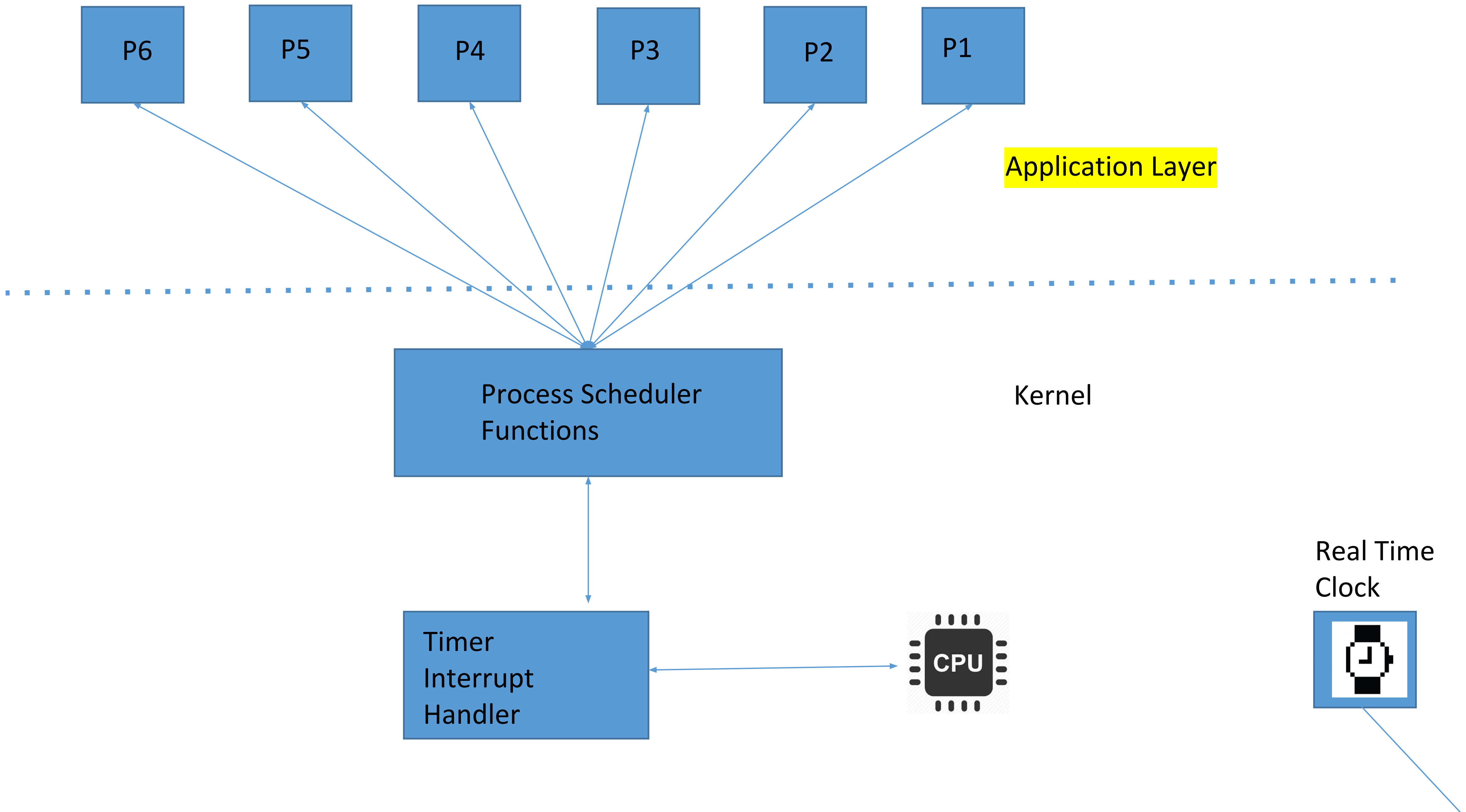
Processes/Tasks/Threads

- Running program binary
- The OS loads up the binary
  - Runtime linker supplies the necessary address bindings – the address of memory where local variables are stored – MMU Used.
  - Saves ``state'' of the running program in Process Control Block (PCB) for each process.
  - Loads the ``state'' of the new program and starts executing it.
- Every so often a timer interrupt ``interrupts'' the CPU to switch the process/task.
- Timer interrupt handler ⍰ invokes ⍰ process scheduler.

Multi-Processing (Why you exactly need an OS)

- Multiple processes running in tandem giving the illusion that multiple programs are running.
- Process executions synchronized with the help of the process/task scheduler.
  - Scheduler is invoked periodically by the timer interrupt handler which saves the state of a task and loads up another.
  - Gives illusion of multi-processing.
- Task state –
  - Every running program has a task / process state which includes:
    - Process ID (PID)
    - CPU registers
    - Memory addresses for code, stack, data, heap and BSS
    - I/O operations in progress.
    - Interrupts that are unhandled.
    - Previous process running and next process that needs to be run (in the process queue).

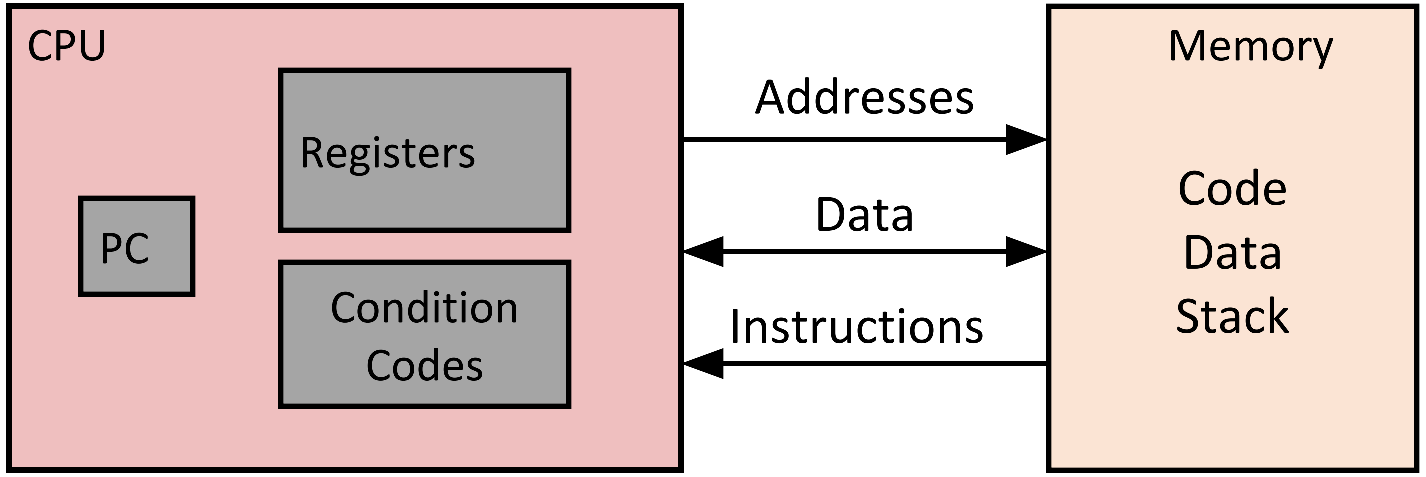
Processes


P6
P5
P4
P3
P2
P1
Application Layer
Process Scheduler Functions
Kernel
Timer Interrupt Handler
CPU
Real Time Clock

Intel’s 64-Bit

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- AMD Stepped 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!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

Assembly Programmer’s View



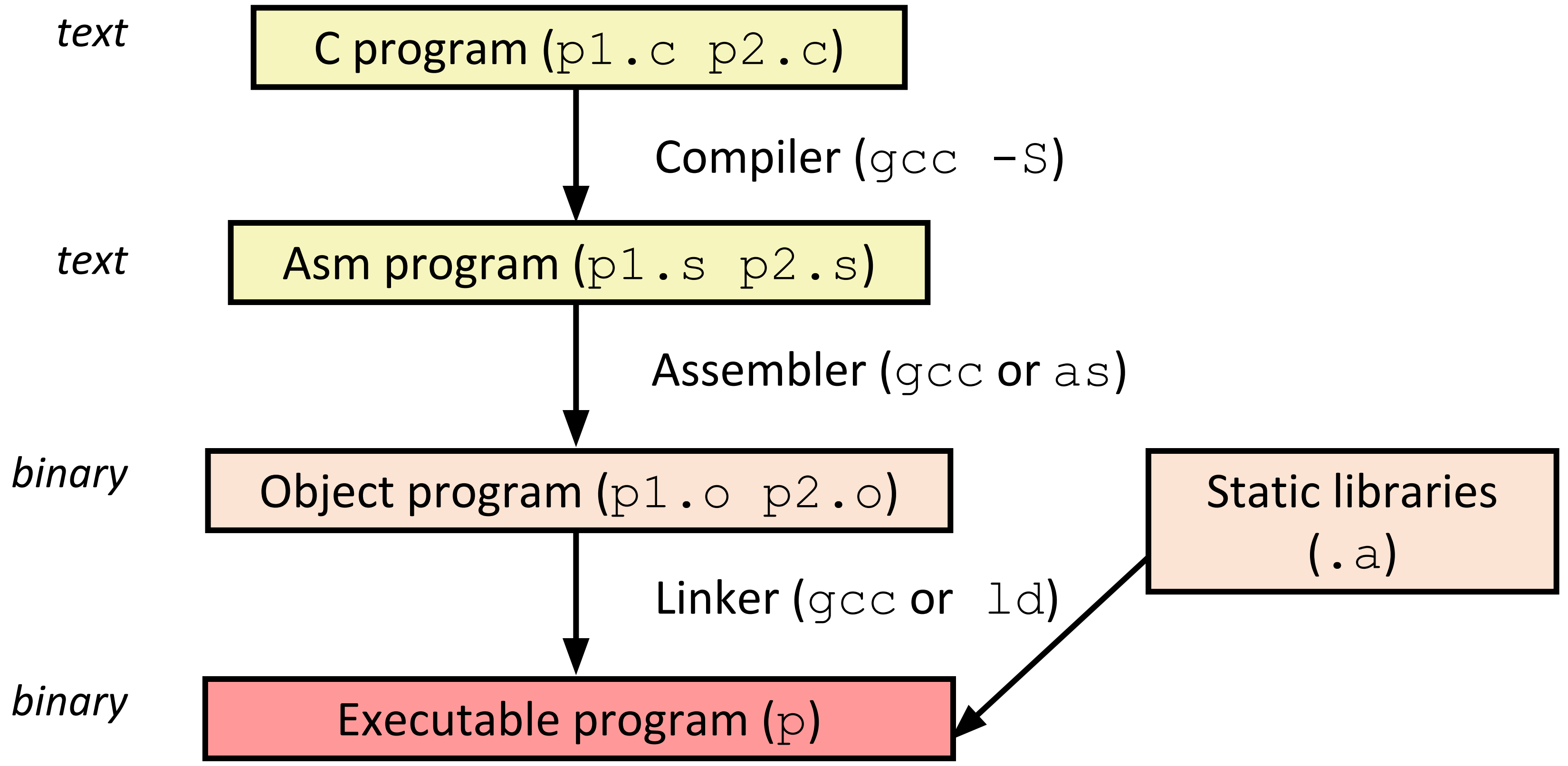
Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “EIP” (IA32) or “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 and user data
  - Stack to support procedures

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc -O1 p1.c p2.c -o p`**
  - Use basic optimizations (**`-O1`**)
  - Put resulting binary in file **`p`**



Compiling Into Assembly

C Code

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

Generated IA32 Assembly

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   popl %ebp
   ret
```

Obtain with command

```
/usr/local/bin/gcc –O1 -S code.c
```

Produces file `code.s`

Assembly Characteristics: Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

Code for `sum`

```
0x401040 <sum>:
   0x55
   0x89
   0xe5
   0x8b
   0x45
   0x0c
   0x03
   0x45
   0x08
   0x5d
   0xc3
```

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x80483ca:  03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:

    `x`:Register **`%eax`**

    `y`:Memory **`M[%ebp+8]`**

    `t`:Register **`%eax`**

    - Return function value in **`%eax`**
- Object Code
  - 3-byte instruction
  - Stored at address **`0x80483ca`**

Disassembling Object Code

Disassembled

```
080483c4 <sum>:
 80483c4:  55          push   %ebp
 80483c5:  89 e5       mov    %esp,%ebp
 80483c7:  8b 45 0c    mov    0xc(%ebp),%eax
 80483ca:  03 45 08    add    0x8(%ebp),%eax
 80483cd:  5d          pop    %ebp
 80483ce:  c3          ret
```

- Disassembler

  `objdump -d p`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

Disassembled

```
Dump of assembler code for function sum:
0x080483c4 <sum+0>:     push   %ebp
0x080483c5 <sum+1>:     mov    %esp,%ebp
0x080483c7 <sum+3>:     mov    0xc(%ebp),%eax
0x080483ca <sum+6>:     add    0x8(%ebp),%eax
0x080483cd <sum+9>:     pop    %ebp
0x080483ce <sum+10>:    ret
```

- Within gdb Debugger

  `gdb p`

  `disassemble sum`
  - Disassemble procedure

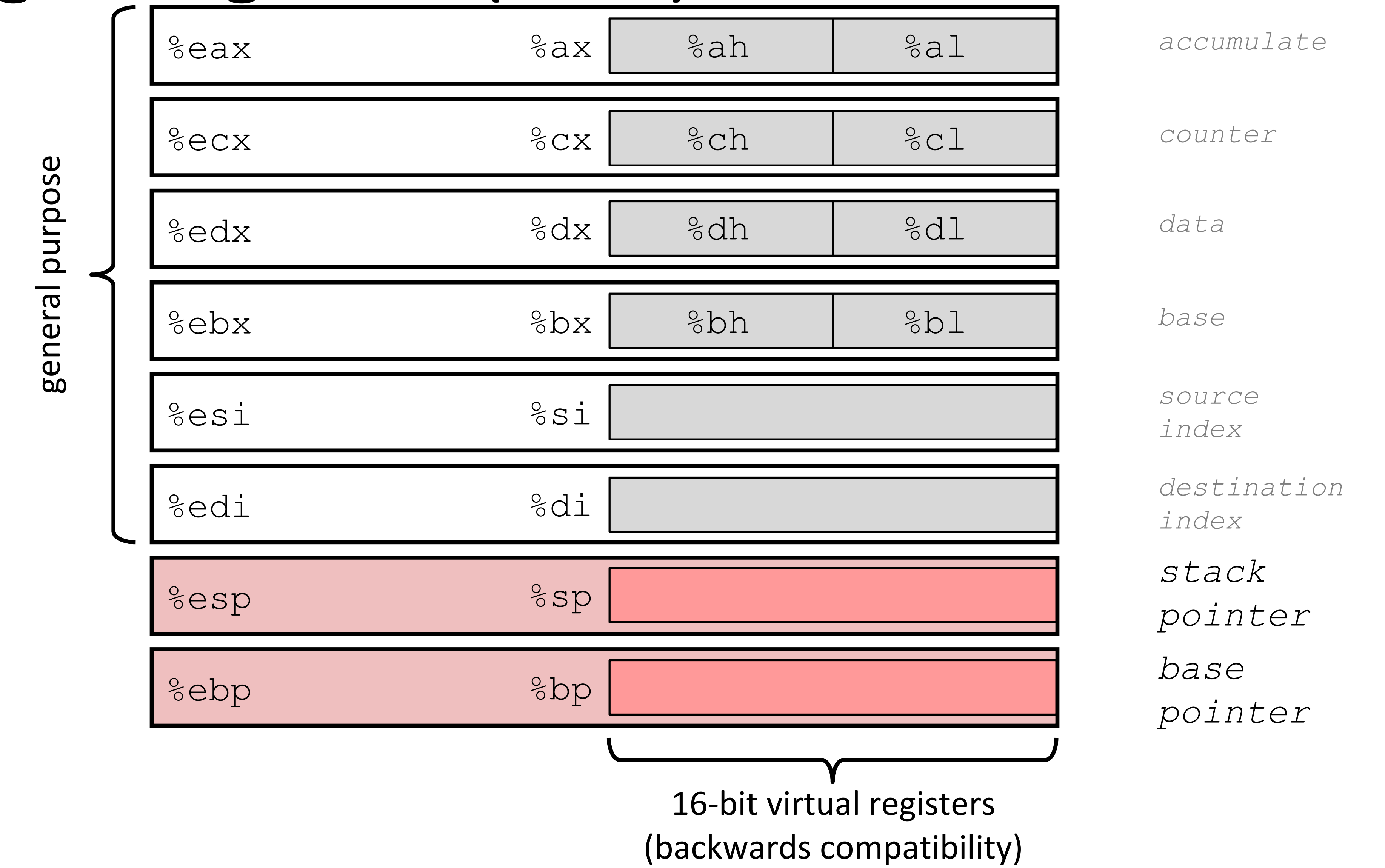
  `x/11xb sum`
  - Examine the 11 bytes starting at `sum`

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55             push   %ebp
30001001:  8b ec          mov    %esp,%ebp
30001003:  6a ff          push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Integer Registers (IA32)

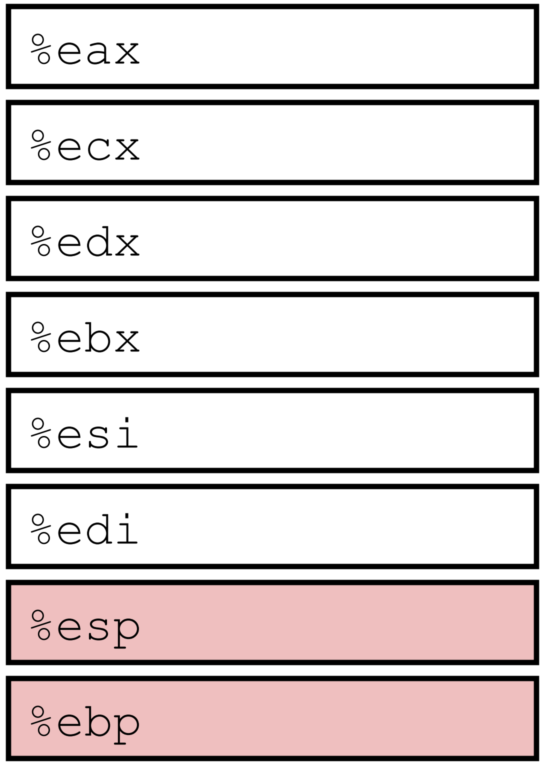


Moving Data: IA32

- Moving Data

  `movl` ***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 8 integer registers
    - Example: **`%eax`**, **`%edx`**
    - But **`%esp`** and **`%ebp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example: **`(%eax)`**
    - Various other "address modes"



`movl` Operand Combinations

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

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

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

```
movl (%ecx),%eax
```

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

```
movl 8(%ebp),%edx
```

Using Simple Addressing Modes

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

```
swap:
  pushl %ebp
  movl  %esp,%ebp
  pushl %ebx

  movl  8(%ebp), %edx
  movl  12(%ebp), %ecx
  movl  (%edx), %ebx
  movl  (%ecx), %eax
  movl  %eax, (%edx)
  movl  %ebx, (%ecx)

  popl  %ebx
  popl  %ebp
  ret
```

$\left.\begin{array}{l}\texttt{pushl \%ebp}\\ \texttt{movl \%esp,\%ebp}\\ \texttt{pushl \%ebx}\end{array}\right\}$ Set Up

$\left.\begin{array}{l}\texttt{movl 8(\%ebp), \%edx}\\ \dots\\ \texttt{movl \%ebx, (\%ecx)}\end{array}\right\}$ Body

$\left.\begin{array}{l}\texttt{popl \%ebx}\\ \texttt{popl \%ebp}\\ \texttt{ret}\end{array}\right\}$ Finish

Using Simple Addressing Modes

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

```
swap:
   pushl %ebp
   movl  %esp,%ebp
   pushl %ebx

   movl  8(%ebp), %edx
   movl  12(%ebp),
%ecx
   movl  (%edx), %ebx
   movl  (%ecx), %eax
   movl  %eax, (%edx)
   movl  %ebx, (%ecx)

   popl  %ebx
   popl  %ebp
   ret
```

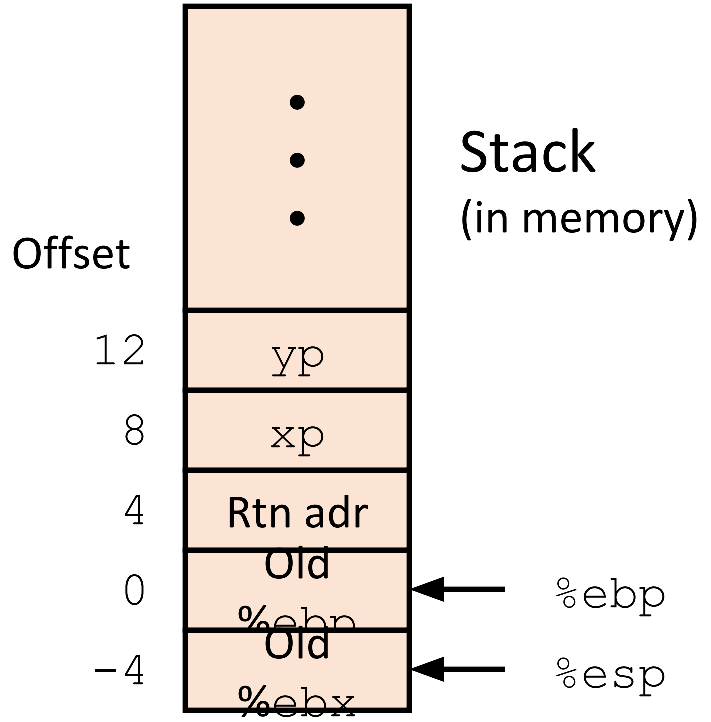
Set Up

Body

Finish

Understanding Swap

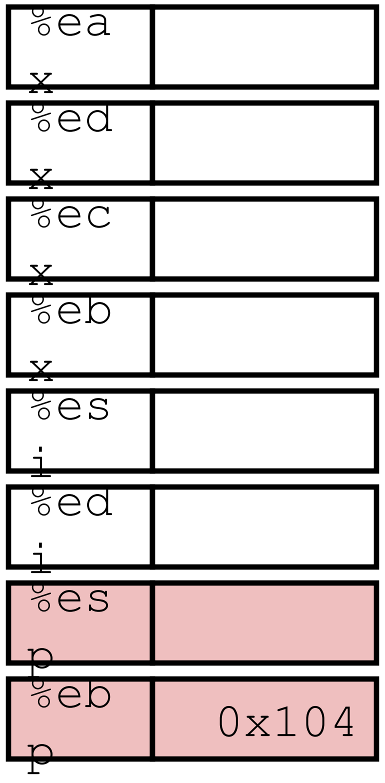
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

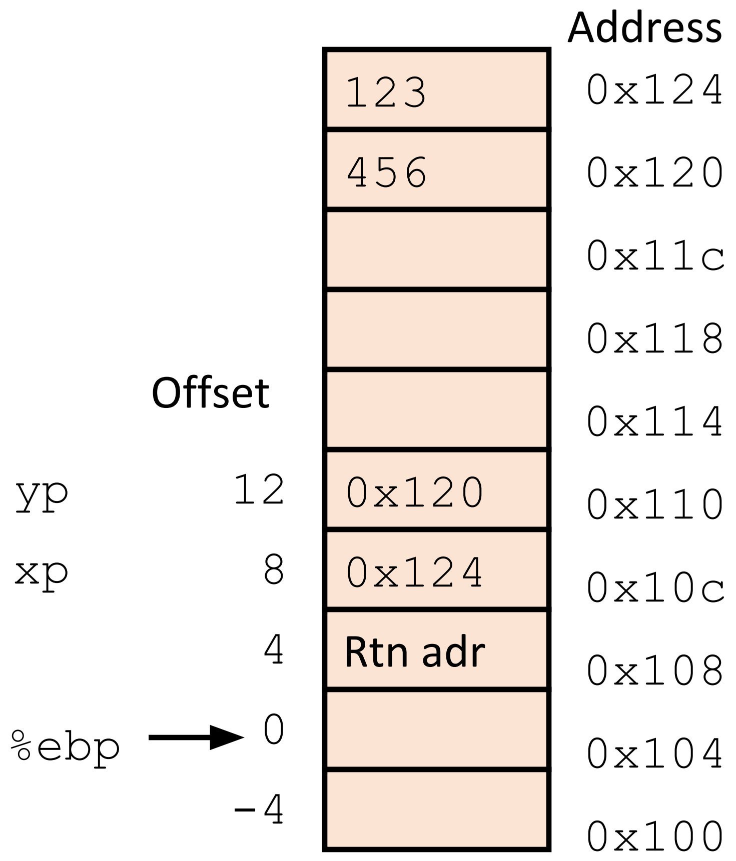


Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
movl   8(%ebp), %edx # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx  # ebx = *xp (t0)
movl   (%ecx), %eax  # eax = *yp (t1)
movl   %eax, (%edx)  # *xp = t1
movl   %ebx, (%ecx)  # *yp = t0
```

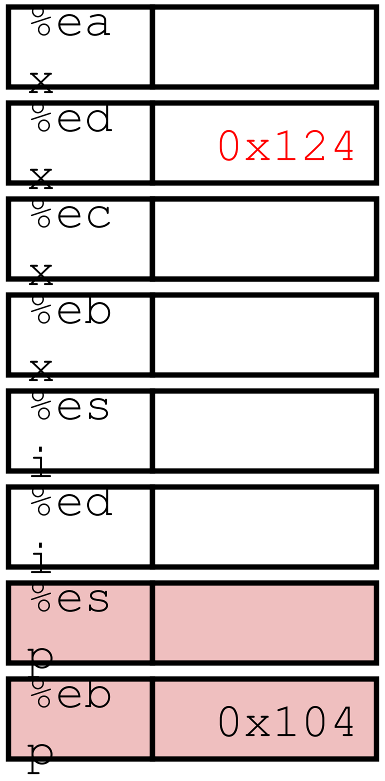
Understanding Swap

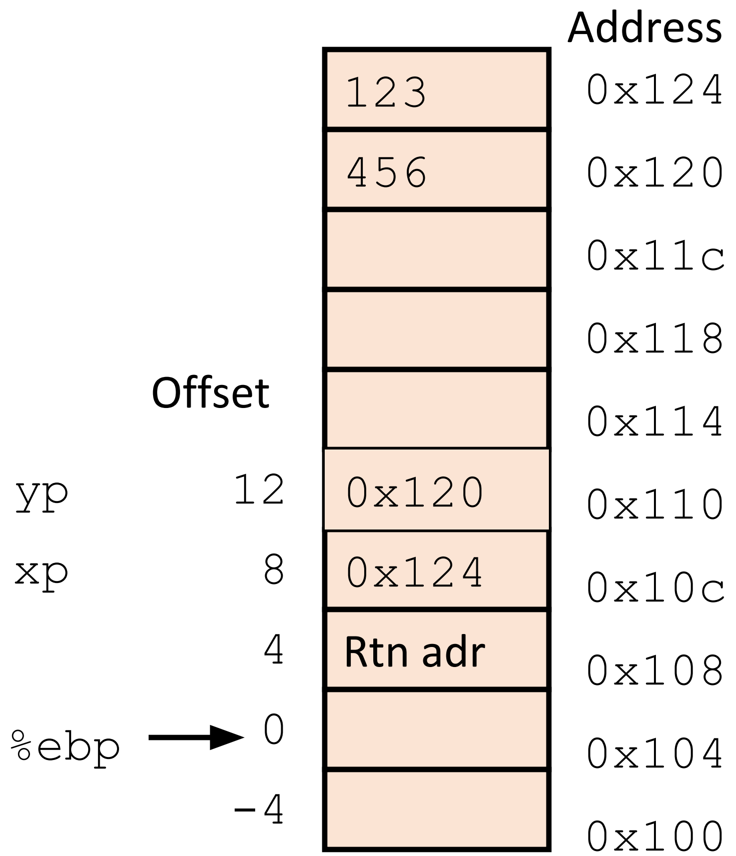




```
movl   8(%ebp), %edx # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx # ebx = *xp (t0)
movl   (%ecx), %eax # eax = *yp (t1)
movl   %eax, (%edx) # *xp = t1
movl   %ebx, (%ecx) # *yp = t0
```

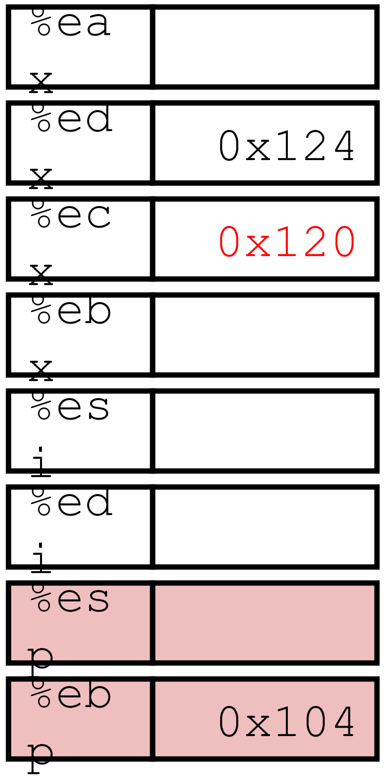
Understanding Swap

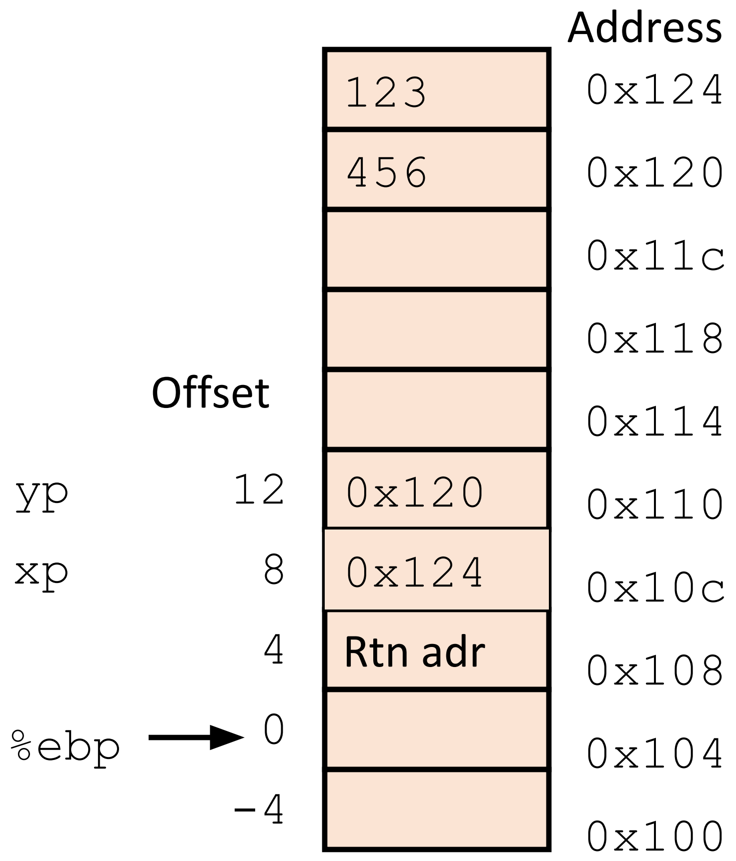




```
movl    8(%ebp), %edx # edx = xp
movl    12(%ebp), %ecx   # ecx = yp
movl    (%edx), %ebx  # ebx = *xp (t0)
movl    (%ecx), %eax  # eax = *yp (t1)
movl    %eax, (%edx)  # *xp = t1
movl    %ebx, (%ecx)  # *yp = t0
```

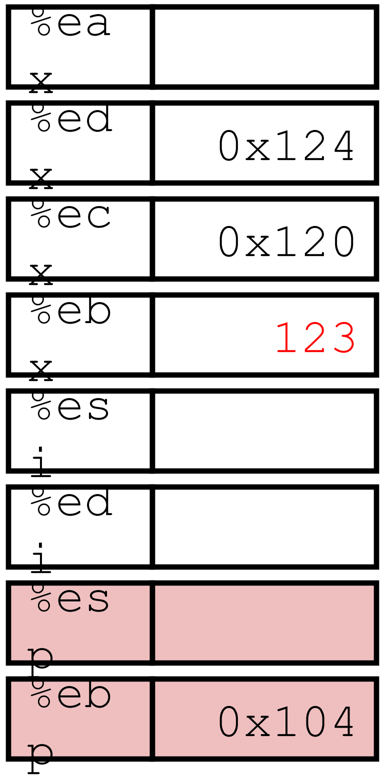
Understanding Swap

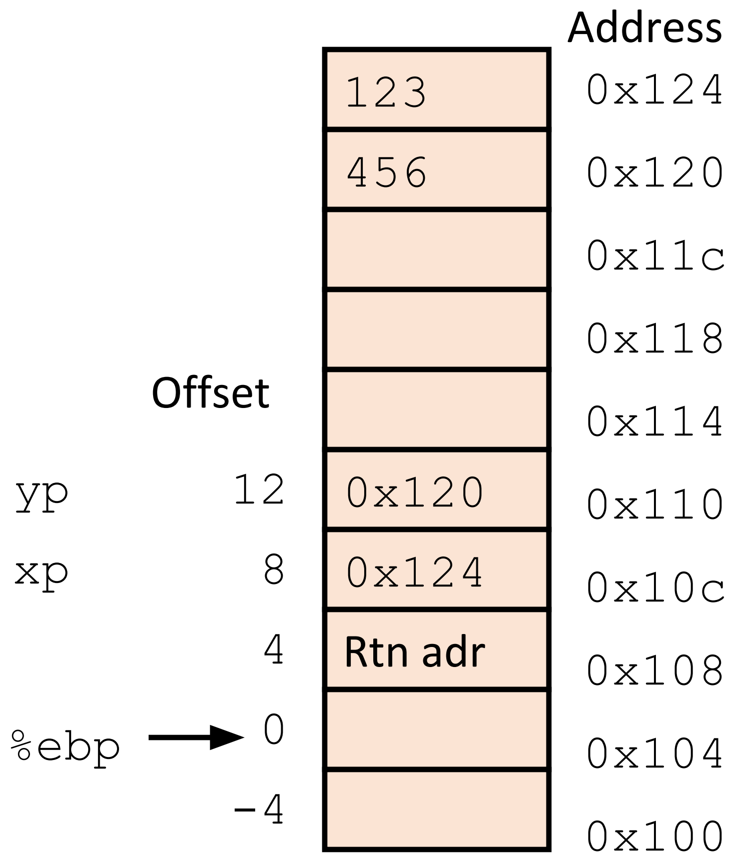




```
movl    8(%ebp), %edx # edx = xp
movl    12(%ebp), %ecx   # ecx = yp
movl    (%edx), %ebx  # ebx = *xp (t0)
movl    (%ecx), %eax  # eax = *yp (t1)
movl    %eax, (%edx)  # *xp = t1
movl    %ebx, (%ecx)  # *yp = t0
```

Understanding Swap

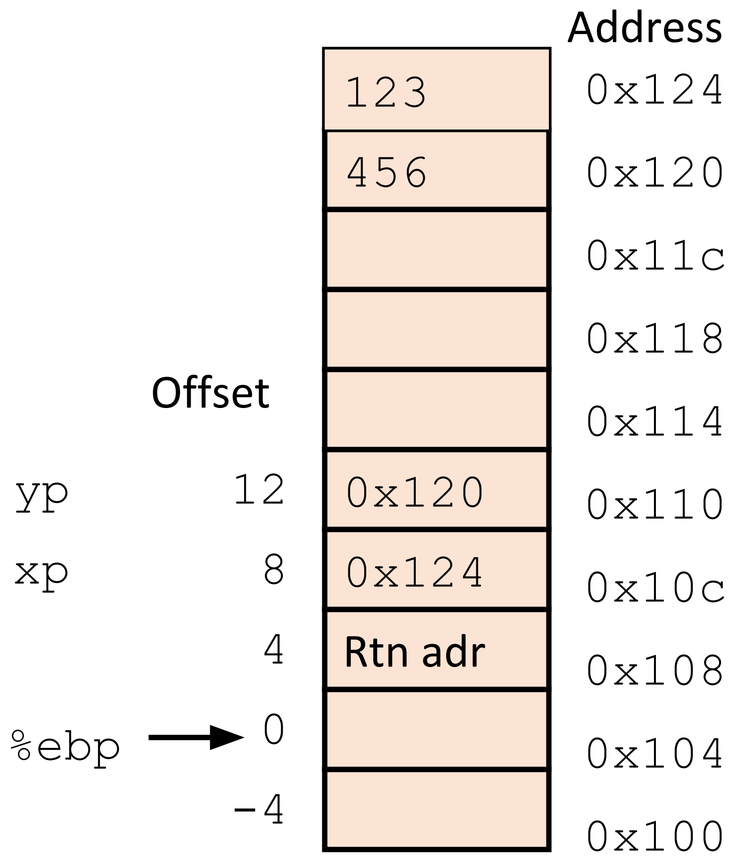




```
movl   8(%ebp), %edx # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx  # ebx = *xp (t0)
movl   (%ecx), %eax  # eax = *yp (t1)
movl   %eax, (%edx)  # *xp = t1
movl   %ebx, (%ecx)  # *yp = t0
```

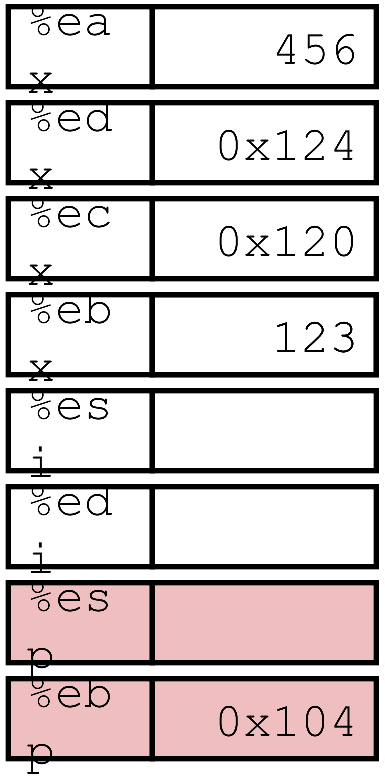
Understanding Swap

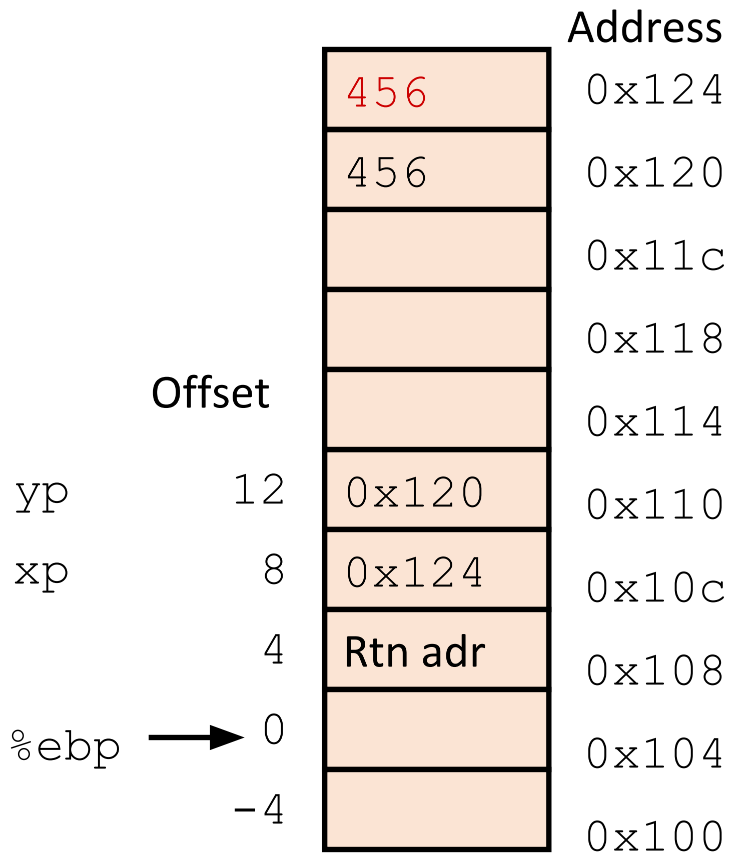
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104



```
movl    8(%ebp), %edx # edx = xp
movl    12(%ebp), %ecx   # ecx = yp
movl    (%edx), %ebx  # ebx = *xp (t0)
movl    (%ecx), %eax  # eax = *yp (t1)
movl    %eax, (%edx)  # *xp = t1
movl    %ebx, (%ecx)  # *yp = t0
```

Understanding Swap





```
movl    8(%ebp), %edx # edx = xp
movl    12(%ebp), %ecx   # ecx = yp
movl    (%edx), %ebx  # ebx = *xp (t0)
movl    (%ecx), %eax  # eax = *yp (t1)
movl    %eax, (%edx)  # *xp = t1
movl    %ebx, (%ecx)  # *yp = t0
```

Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		456	0x124
		123	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl    8(%ebp), %edx # edx = xp
movl    12(%ebp), %ecx   # ecx = yp
movl    (%edx), %ebx  # ebx = *xp (t0)
movl    (%ecx), %eax  # eax = *yp (t1)
movl    %eax, (%edx)  # *xp = t1
movl    %ebx, (%ecx)  # *yp = t0
```

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 8 integer registers
  - Ri: Index register: Any, except for **`%esp`**
    - Unlikely you’d use **`%ebp`**, 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]]

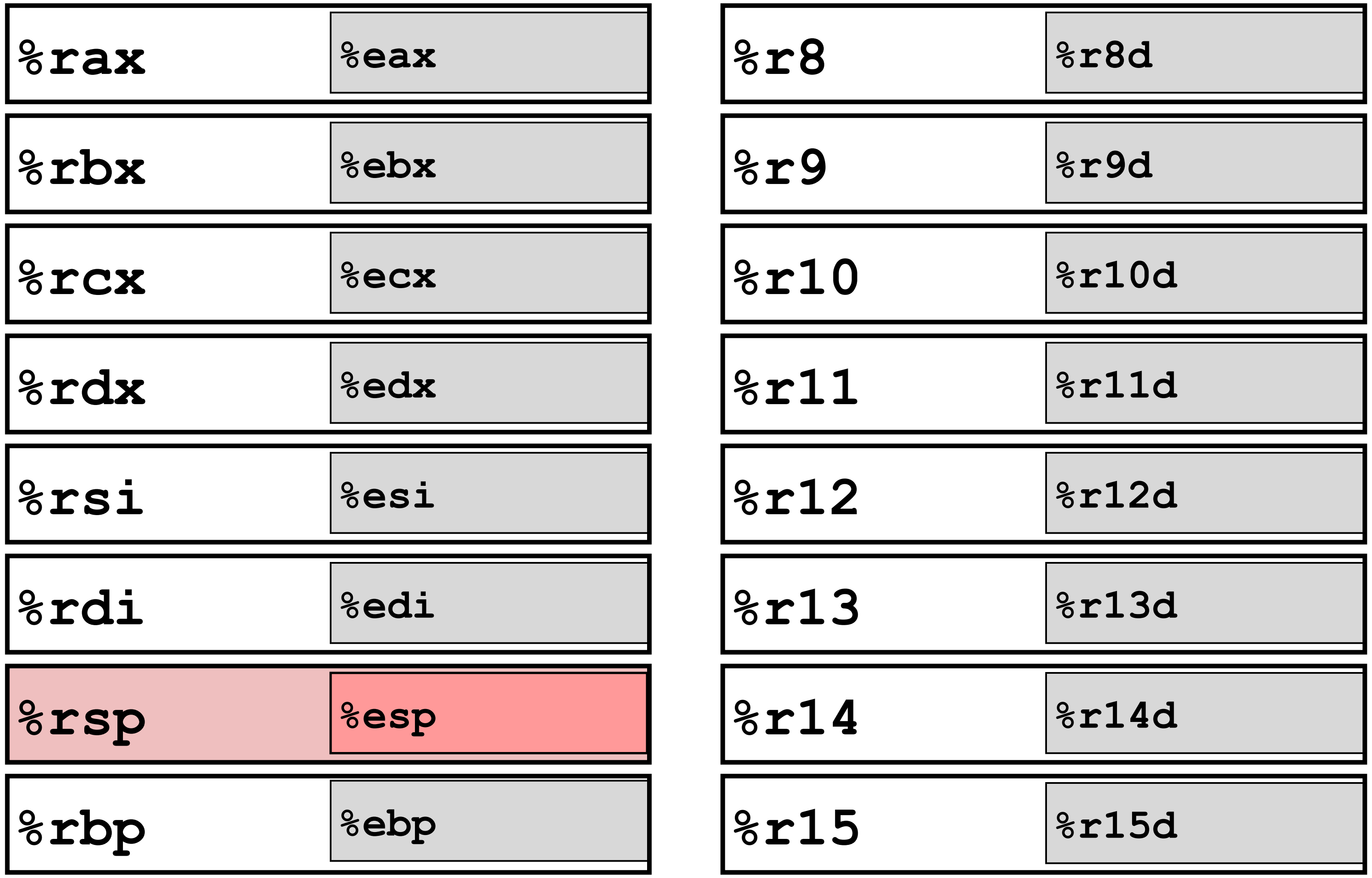
Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)

C Data Type	*Generic 32-bit*	*Intel IA32*	*x86-64*
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
char *	4	4	8

  - *Or any other pointer*

x86-64 Integer Registers



- Extend existing registers. Add 8 new ones.
- Make **`%ebp/%rbp`** general purpose

Instructions

- Long word **`l`** (4 Bytes) ↔ Quad word **`q`** (8 Bytes)

- New instructions:
  - **`movl`** → **`movq`**
  - **`addl`** → **`addq`**
  - **`sall`** → **`salq`**
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: **`addl`**

32-bit code for swap

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

```
swap:
   pushl %ebp                    
   movl  %esp,%ebp                 Set
   pushl %ebx                      Up

   movl  8(%ebp), %edx
   movl  12(%ebp), %ecx
   movl  (%edx), %ebx              Body
   movl  (%ecx), %eax
   movl  %eax, (%edx)
   movl  %ebx, (%ecx)

   popl  %ebx
   popl  %ebp                      Finish
   ret
```

64-bit code for swap

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

```
swap:
                                    } Set Up

    movl   (%rdi), %edx             }
    movl   (%rsi), %eax             } Body
    movl   %eax, (%rdi)             }
    movl   %edx, (%rsi)             }

                                    } Finish
    ret
```

- Operands passed in registers (why useful?)
  - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers **%eax** and **%edx**
  - **movl** operation

64-bit code for long int swap

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

```
swap_l:
                                  } Set Up

    movq    (%rdi), %rdx         }
    movq    (%rsi), %rax         } Body
    movq    %rax, (%rdi)         }
    movq    %rdx, (%rsi)         }

    ret                           } Finish
```

- 64-bit data
  - Data held in registers **`%rax`** and **`%rdx`**
  - ***`movq`*** operation
    - “q” stands for quad-word

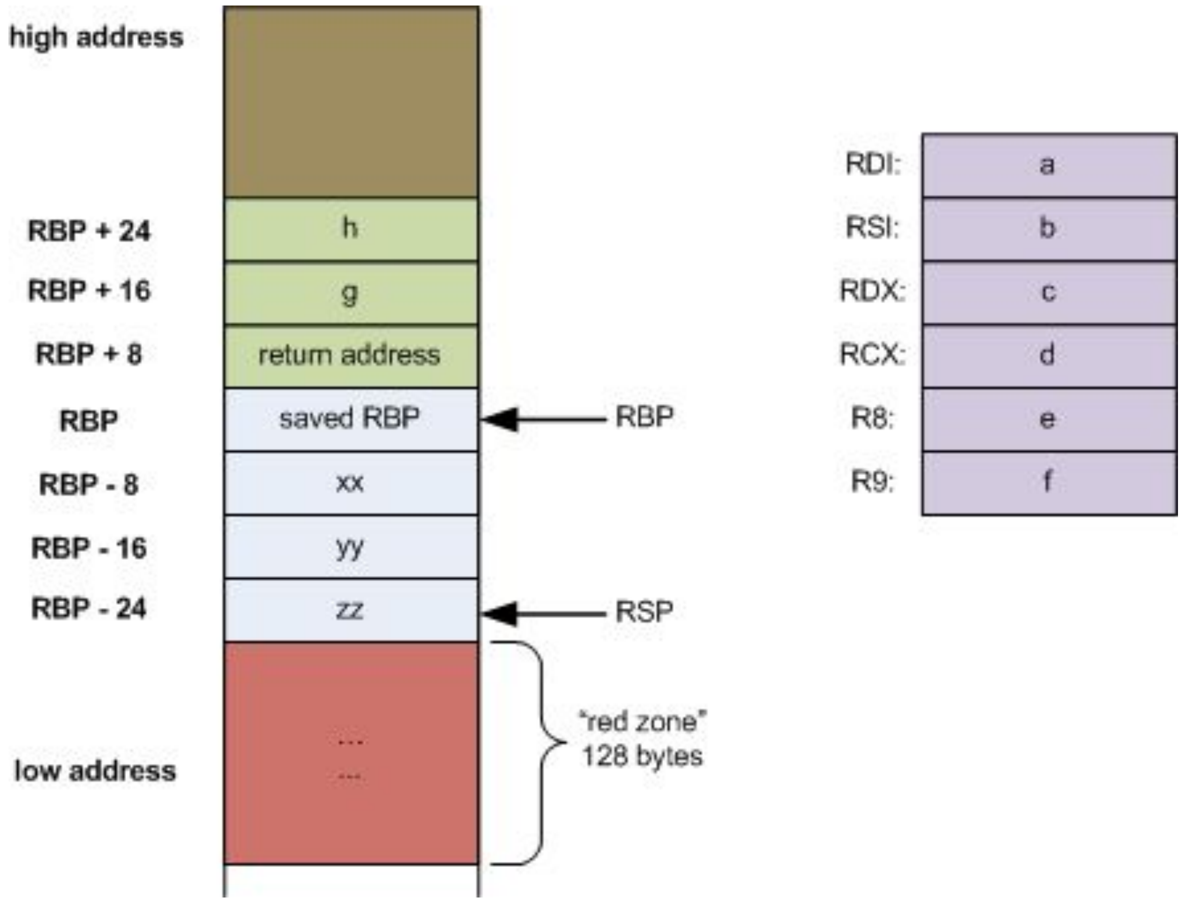
Calling conventions : X86

```
int callee(int, int, int);


int caller(void)
{
    return callee(1, 2, 3) + 5;
}
```

```
caller:
   push    ebp       ; save old call frame
    mov     ebp, esp  ; initialize new call frame
     push    3
    push    2
    push    1
    call    callee    ; call subroutine 'callee'
    add     esp, 12   ; remove call arguments from frame
    add     eax, 5    ; modify subroutine result
     mov     esp, ebp  ; most calling conventions dictate ebp be
callee-saved,
   pop     ebp       ; restore old call frame
    ret               ; return
```

Calling conventions : X86_64

```
long myfunc(long a, long b, long c, long d,
          long e, long f, long g, long h)
{
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```


high address
RBP + 24
h
RBP + 16
g
RBP + 8
return address
RBP
saved RBP
RBP
RBP - 8
xx
RBP - 16
yy
RBP - 24
zz
RSP
...
...
low address
"red zone" 128 bytes
RDI: a
RSI: b
RDX: c
RCX: d
R8: e
R9: f