

# CENG331 - Computer Organization

*Introduction*

Fall 2020

**Instructor:**

Murat Manguoglu (Sections 1-2)

Unless otherwise noted adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Introduction

- Course theme
- Five realities

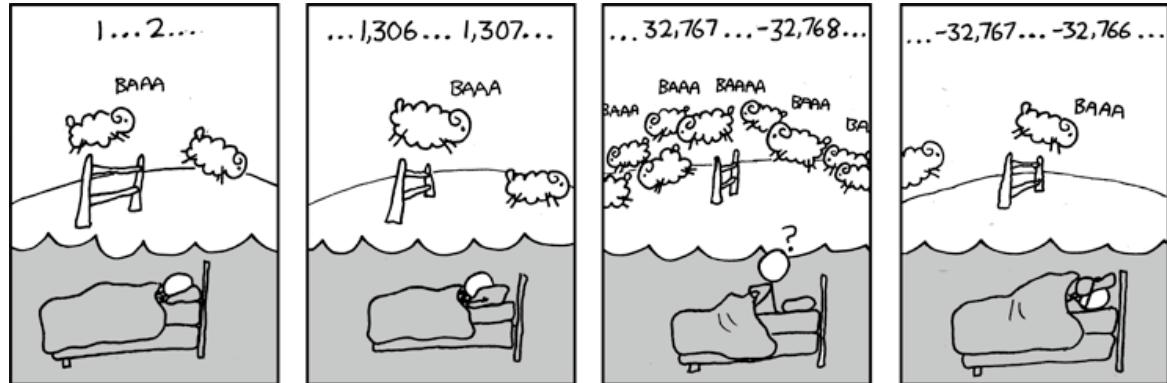
# Course Theme: Abstraction Is Good But Reality Matters

- Most CENG courses emphasize abstractions
  - Abstract data types
  - Asymptotic analysis
- These abstractions have limits
  - Especially in the presence of bugs, performance constraints
  - Need to understand details of underlying implementations
- Useful outcomes of CENG331
  - Become more effective programmers and computer scientists
    - *Able to find and eliminate bugs efficiently*
    - *Able to understand and tune your code for performance*
  - Prepare for later “systems” classes in CS/CENG
    - *Intro. to Parallel Computing, Compilers, Operating Systems, Networks, Embedded Systems, etc.*

# Reality #1: Ints are not Integers, Floats are not Reals

## ■ Example 1: Is $x^2 \geq 0$ ?

- Float's: Yes!



- Int's:

- $40000 * 40000 \rightarrow 1600000000$
- $50000 * 50000 \rightarrow ??$

## ■ Example 2: Is $(x + y) + z = x + (y + z)$ ?

- Unsigned & Signed Ints: Yes!

- Floats:

- $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
- $1e20 + (-1e20 + 3.14) \rightarrow ??$

# Computer Arithmetic

## ■ ~~Does not generate random values~~ (“most of the time”)

- Arithmetic operations have important mathematical properties
- Random errors, bit flips, etc. are possible on large scale computers or quantum computers!

## ■ Cannot assume all “usual” mathematical properties

- Due to finiteness of representations
- Integer operations satisfy “ring” properties
  - Commutativity, associativity, distributivity
- Floating point operations satisfy “ordering” properties
  - Monotonicity, values of signs

## ■ Observation

- Need to understand which abstractions apply in which contexts
- Important issues for compiler writers and serious application programmers

# Reality #2: You've Got to Know Assembly (and the processor)

- Chances are, you'll never write programs in assembly
  - Compilers are much better & more patient than you
- But: Understanding assembly is key to machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance: you can never only rely on compiler optimizations, usually you can do much better!
    - Understand optimizations done / not done by the compiler (*or can not be done*)
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

# Reality #3: Memory Matters

## Random Access Memory Is an Unphysical Abstraction

### ■ Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated

### ■ Memory referencing bugs especially pernicious

- Effects are distant in both time and space

### ■ Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

# Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

fun(0)	→	3.14
fun(1)	→	3.14
<b>fun(2)</b>	→	<b>3.1399998664856</b>
<b>fun(3)</b>	→	<b>2.00000061035156</b>
<b>fun(4)</b>	→	<b>3.14</b>
<b>fun(6)</b>	→	<b>Segmentation fault</b>

- Result is system specific

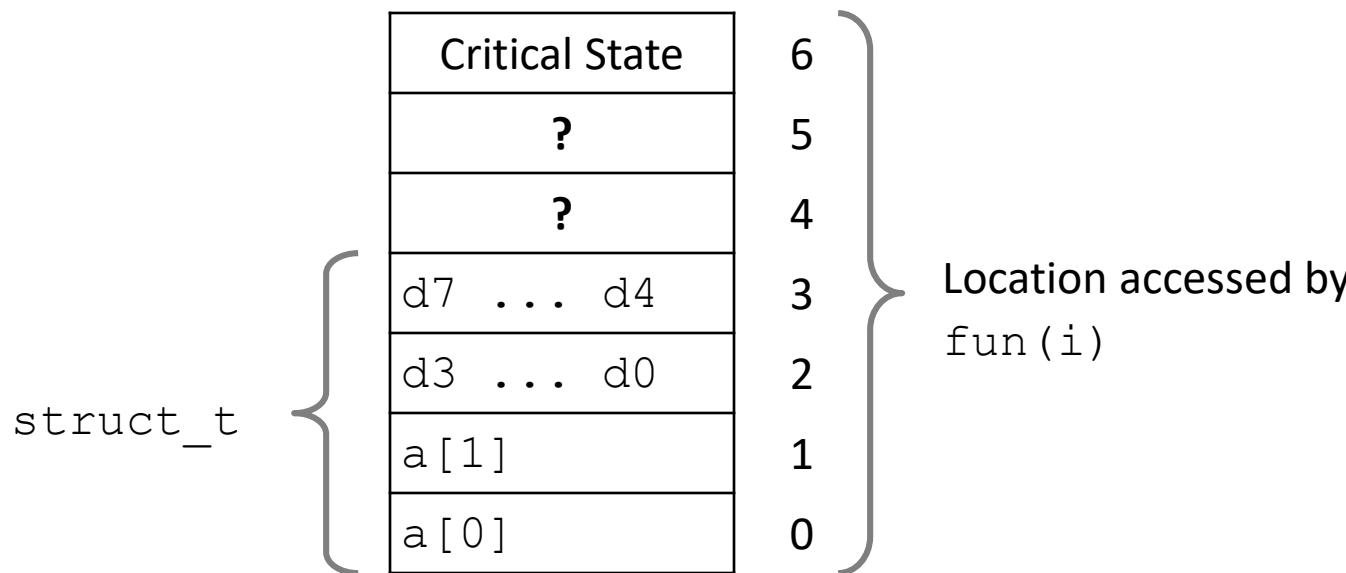
**volatile keyword** indicates that a [value](#) may change between different accesses, even if it does not appear to be modified. This keyword prevents an [optimizing compiler](#) from optimizing away subsequent reads or writes and thus incorrectly reusing a stale value or omitting writes

# Memory Referencing Bug Example

```
typedef struct {  
    int a[2];  
    double d;  
} struct_t;
```

fun(0) →	3.14
fun(1) →	3.14
fun(2) →	3.1399998664856
fun(3) →	2.00000061035156
fun(4) →	3.14
fun(6) →	Segmentation fault

## Explanation:



# Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in Java, Ruby, Python, ML, ...
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. [Valgrind](#))

# Reality #4: There's more to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

# Memory System Performance Example

```
void copyij(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (i = 0; i < 2048; i++)  
        for (j = 0; j < 2048; j++)  
            dst[i][j] = src[i][j];  
}
```

```
void copyji(int src[2048][2048],  
           int dst[2048][2048])  
{  
    int i,j;  
    for (j = 0; j < 2048; j++)  
        for (i = 0; i < 2048; i++)  
            dst[i][j] = src[i][j];  
}
```

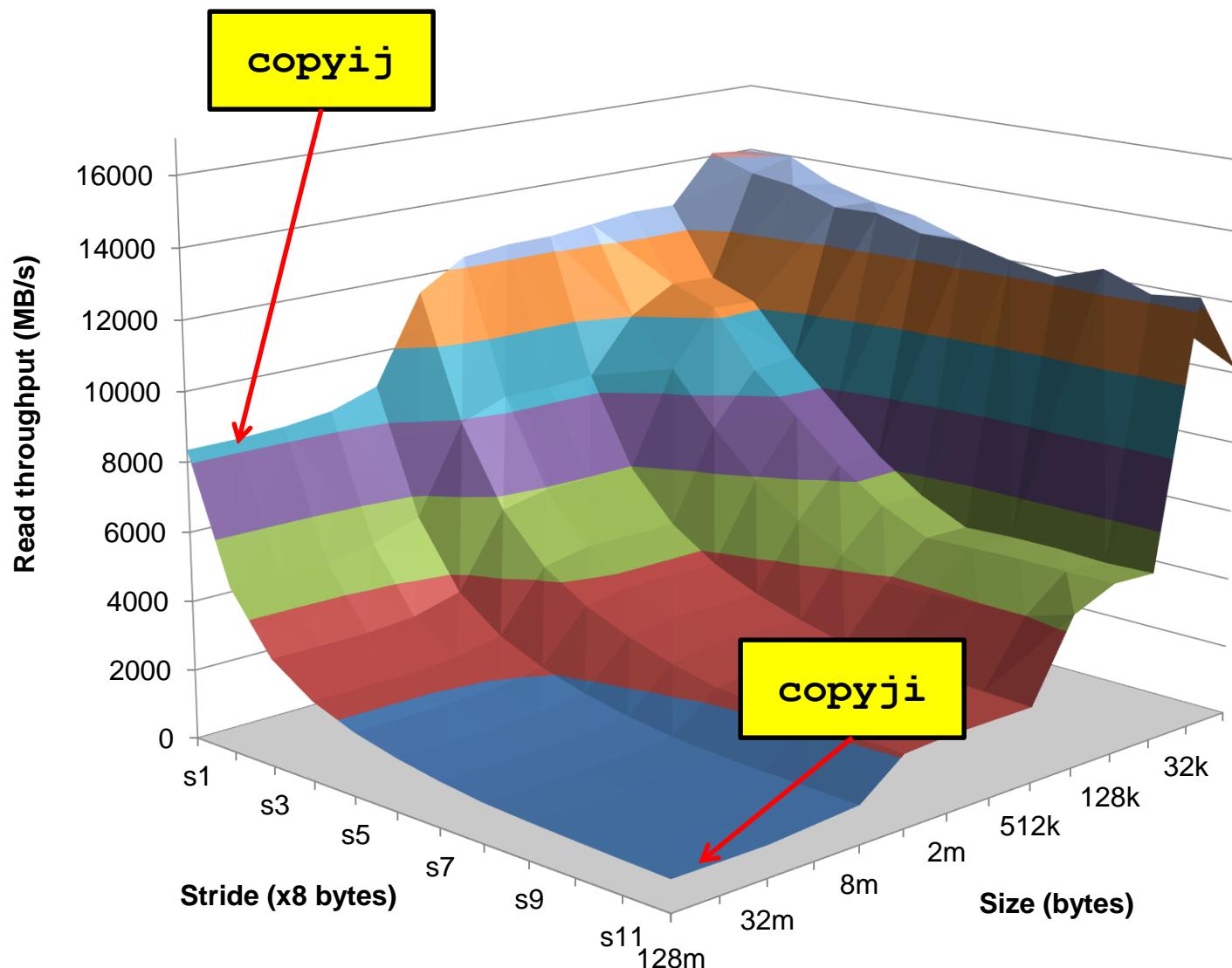
4.3ms

2.0 GHz Intel Core i7 Haswell

81.8ms

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

# Why The Performance Differs



# Another performance figure (matrix mult.)

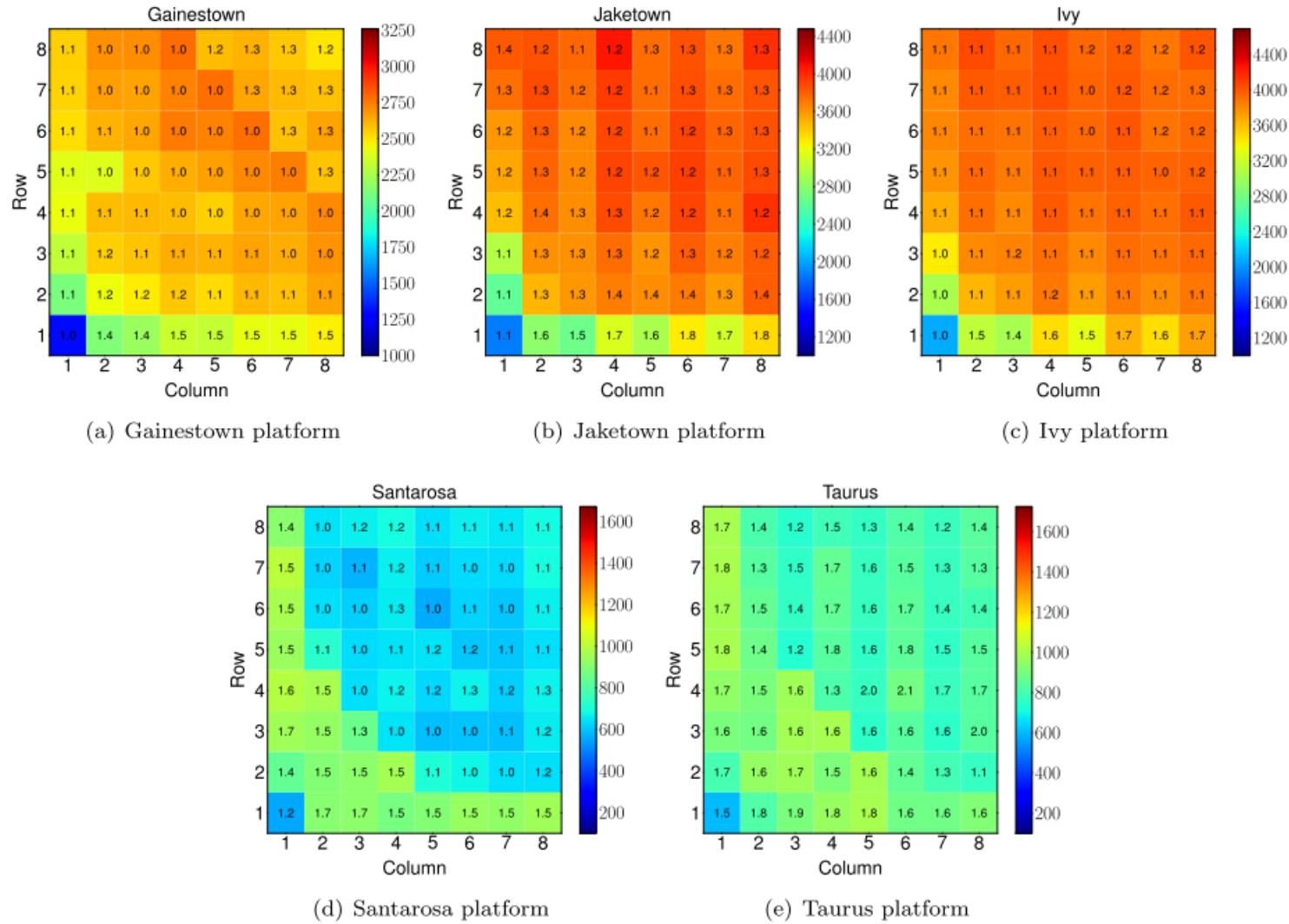


Figure 8: SpMV Performance Profiles for optimized-BCSR on a dense matrix on a single core. On each platform, each square is an  $r \times c$  implementation,  $1 \leq r, c \leq 8$ , colored by its performance in MFlops/sec, and labeled by its speedup over OSKI. The top of the speed range for each platform is the performance bound from the Roofline model, in the last column of Table 5.

Source: Byun, J. H., Lin, R., Yelick, K. A., & Demmel, J. (2012). Autotuning sparse matrix-vector multiplication for multicore. *EECS, UC Berkeley, Tech. Rep*

# Reality #5:

## Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance
- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

CENG435 Data Communications and Networking  
CENG478 Introduction to Parallel Computing

*Thank you !*

# CENG331 - Computer Organization

*Course overview and logistics*

Fall 2020

**Instructor:**

Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# **Overview and logistics**

- How the course fits into the CENG curriculum
- Syllabus
- Academic integrity

# Course Perspective

## ■ Most Systems Courses are Builder-Centric

- Operating Systems
  - Implement sample portions of operating system
- Compilers
  - Write compiler for simple language
- Networking
  - Implement and simulate network protocols

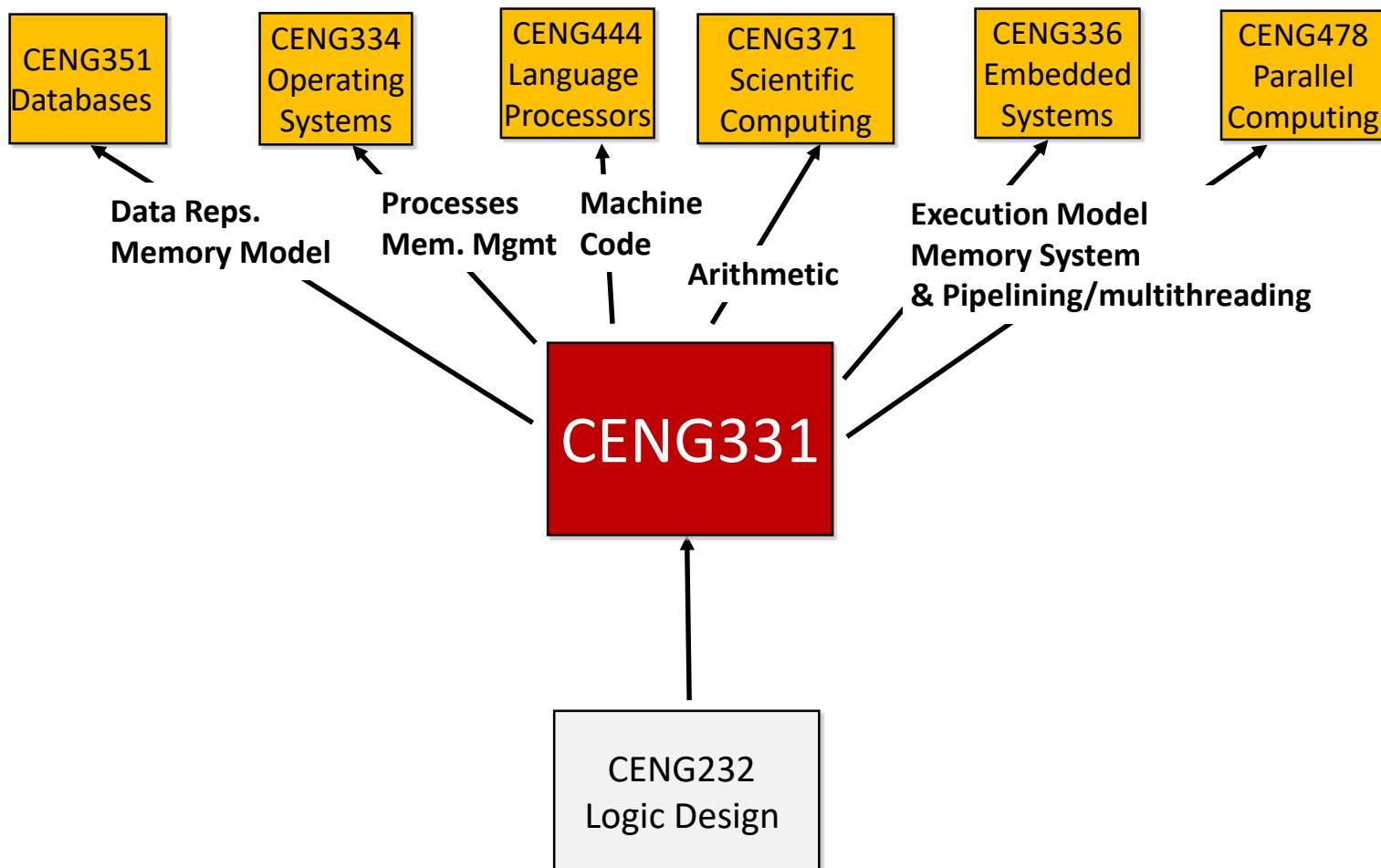
# Course Perspective (Cont.)

## ■ Our Course is Programmer-Centric

- Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
- Enable you to
  - Write programs that are more reliable and efficient
  - Incorporate features that require hooks into OS
    - E.g., concurrency, signal handlers
- Cover material in this course that you won't see elsewhere
- Not just a course for dedicated hackers
  - We bring out the hidden hacker in everyone!

# Role within CENG Curriculum

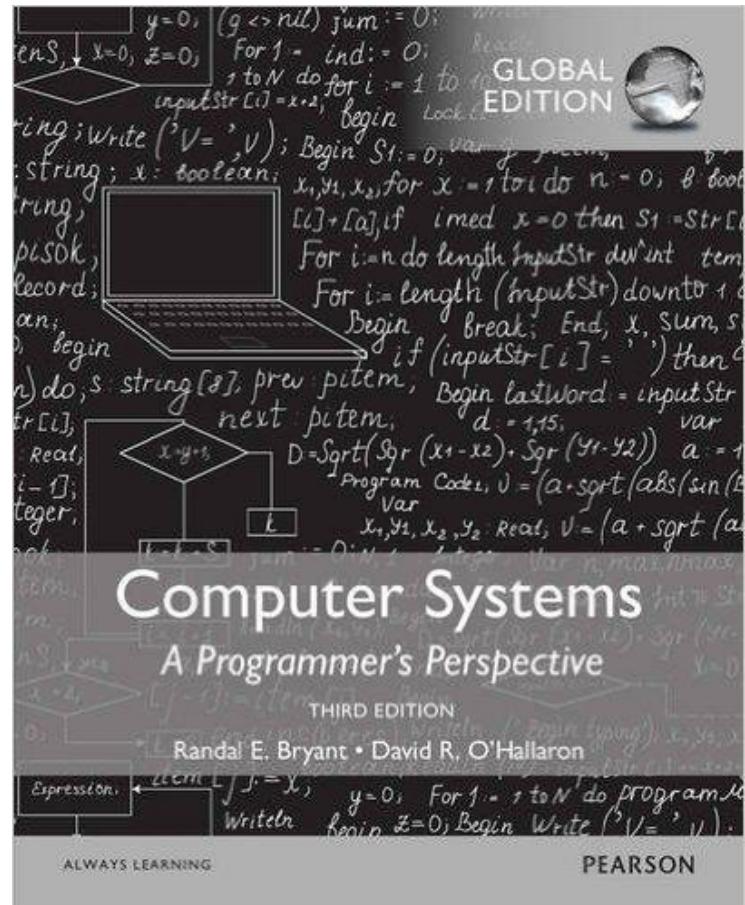
Spring 2021



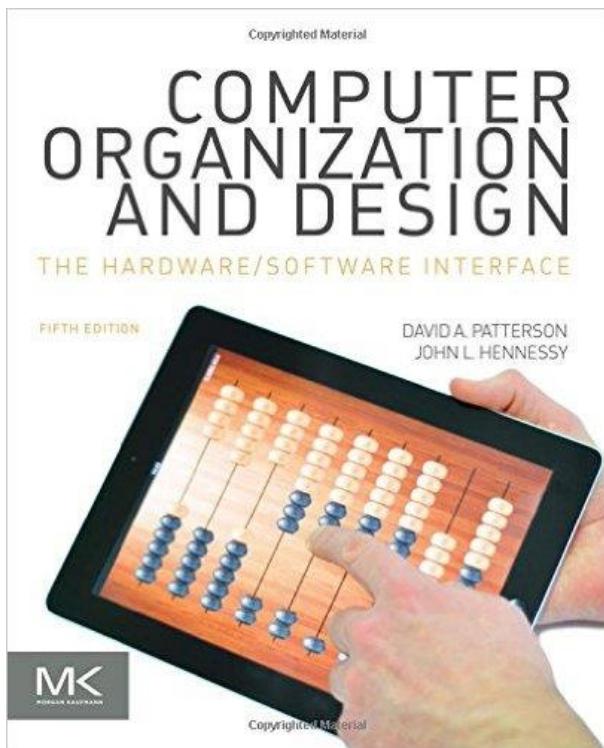
# Textbook

## ■ Randal E. Bryant and David R. O'Hallaron,

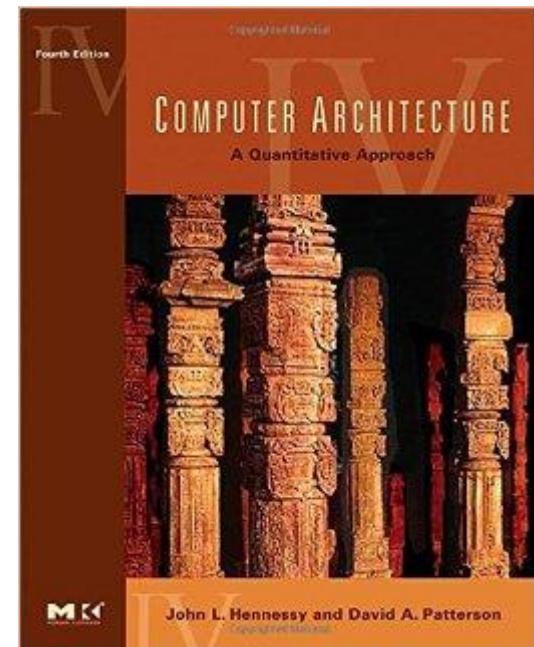
- *Computer Systems: A Programmer's Perspective*, Third Edition  
(CS:APP3e), Pearson, 2016
- <http://csapp.cs.cmu.edu>
- This book really matters for the course!
  - How to solve labs
  - Practice problems typical of exam problems



# Reference Texts



- **Computer Organization and Design: Hardware/Software Interface by Patterson and Hennessy**
  
- **Computer Architecture: A Quantitative Approach by Patterson and Hennessy**



# Course Components

## ■ Lectures

- Video Lectures : Posted weekly
- Online Lectures: During regular lecture hours ( I will post the link on ODTUClass)
  - These are discussion hours where you can ask and answer others' questions.I will act as a moderator mostly.

## ■ Take-home labs (4)

- Provide in-depth understanding and hands on experience on an aspect of computing systems; programming and performance measurement

## ■ Exams (midterm + final)

- Test your understanding of concepts & mathematical principles

## ■ Participation

- Participation in discussions and/or random quizzes during online lectures

Technique	Utility
Elaborative interrogation	Moderate
Self-explanation	Moderate
Summarization	Low
Highlighting	Low
The keyword mnemonic	Low
Imagery use for text learning	Low
Rereading	Low
Practice testing	High
Distributed practice	High
Interleaved practice	Moderate

*Improving Students' Learning With Effective Learning Techniques Promising Directions From Cognitive and Educational Psychology,*  
John Dunlosky, Katherine A. Rawson, Elizabeth J. Marsh, Mitchell J. Nathan and Daniel T. Willingham, **Physiological Science in the Public Interest**

# Tools and Getting Help

## ■ Odtuclass : <http://odtuclass.metu.edu.tr>

- Course discussion forums, announcements, labs and their grades
- Links to lecture videos

## ■ Gradescope: <http://www.gradescope.com>

- Midterm and finals will be conducted through gradescope

## ■ Email communication:

If you have a specific question that is **not beneficial to others**, you can send an e-mail to the instructor or to your teaching assistants. However make sure that the subject line starts with CENG331-Section#, state your first, last name and ID # to get faster reply.

# Email communication:

## ■ TAs:

Çağrı Utku Akpak	( <a href="mailto:capaki@ceng.metu.edu.tr">capaki@ceng.metu.edu.tr</a> )
Deniz Sayın	( <a href="mailto:sayin@ceng.metu.edu.tr">sayin@ceng.metu.edu.tr</a> )
Merve Taplı	( <a href="mailto:tapli@ceng.metu.edu.tr">tapli@ceng.metu.edu.tr</a> )
Hakan Bostan	( <a href="mailto:hbostan@ceng.metu.edu.tr">hbostan@ceng.metu.edu.tr</a> )

## Myself:

Murat Manguoglu	( <a href="mailto:manguoglu@ceng.metu.edu.tr">manguoglu@ceng.metu.edu.tr</a> )
-----------------	--

# Policies: Take-home lab assignments, exams and quizzes

## ■ Study groups

- You are encouraged to study in groups

## ■ Work groups

- You must work alone on 1<sup>st</sup>, 2<sup>nd</sup>, and 3<sup>rd</sup> take-home labs
- You can work in groups (of at most 3) for the 4<sup>th</sup> take home lab

## ■ Midterm and Final Exams

- Written exams , will be partly online (synchronous) and offline (take home)
- Open book, notes, internet but individual work

# Makeups

**There are no makeups unless:**

- Major illness, death in family, ...
- Inform and if needed submit an official report to the instructor as soon as possible

# Policies: Grading

## ■ Written Exams (66%):

- Midterm (30%)
  - Online (5%)
  - Take-home (25%)
- Final (36%)
  - Online (6%)
  - Take-home (30%)

## ■ Lab Assignments (Take-home) (24%)

- 4 x 6% for each lab
- lower threshold for passing is 10% from the labs

## ■ Participation (10%)

# Programs and Data

## ■ Topics

- Bits operations, arithmetic, assembly language programs
- Representation of C control and data structures
- Includes aspects of architecture and compilers

## ■ Take-home Lab Assignments

- Bomblab: Defusing a binary bomb (Individual work)
- Attacklab: The basics of code injection attacks (individual work)

# Processor Architecture

## ■ Topics

- Y86-64 architecture
  - Pipelining and hazards
  - Control structures

## ■ Take-home Lab Assignments

- Architecture (Individual Work)

# **Code optimization and Memory Hierarchy**

## **■ Topics**

- Code optimization
- Memory technology, memory hierarchy, caches, disks, locality
- Includes aspects of architecture and OS

## **■ Take-home assignments**

- Performance: Improve the performance of a kernel which is a bottleneck in an application (Group work up to 3 people)

# Virtual Memory

## ■ Topics

- Virtual memory, address translation
- Includes aspects of architecture and OS

# Other topics (if time permits)

## ■ Topics

- Tensor Processing Units (TPUs)
- Graphical Processing Units (GPUs)
- Quantum Processors
- Multicore Architectures
- Multithreading
- Very Large Instruction Word Machines

# Lab exam Rationale

- Each assignment has a well-defined goal such as solving a puzzle or winning a contest
- Doing the lab should result in new skills and concepts
- We try to use competition in a fun and healthy way
  - Set a reasonable threshold for full credit
  - Post intermediate results (anonymized)

# Cheating: Description

## ■ What is cheating?

- Sharing take-home exam solutions
- Sharing code: by copying, retyping, **looking at**, or supplying a file
- Describing: verbal description of code from one person to another.
- Coaching: helping your friend to write a lab, line by line
- Searching the Web for solutions
- Copying code from a previous course or online solution
  - You are only allowed to use code we supply, or from the CS:APP website

## ■ What is NOT cheating?

- Explaining how to use systems or tools
- Helping others with high-level design issues

# Cheating: Consequences

## ■ Penalty for cheating:

- Disciplinary action

## ■ Detection of cheating:

- We have sophisticated tools for detecting code plagiarism
- And other forms of cheating

## ■ Don't do it!

- Start early
- Ask the staff for help when you get stuck

# METU Honor Code

Every member of METU community adopts the following honor code as one of the core principles of academic life and strives to develop an academic environment where continuous adherence to this code is promoted.

"The members of the METU community are reliable, responsible and honourable people who embrace only the success and recognition they deserve, and act with integrity in their use, evaluation and presentation of facts, data and documents."

METU Academic Integrity Guide for Students:

*Welcome and Enjoy!*

# Bits and Bytes

CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu      (Sections 1-2)

Unless otherwise noted adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Today: Bits and Bytes

- Compiling, linking and executing: Hello World
- Representing information as bits
- Bit-level manipulations

# Hello World!

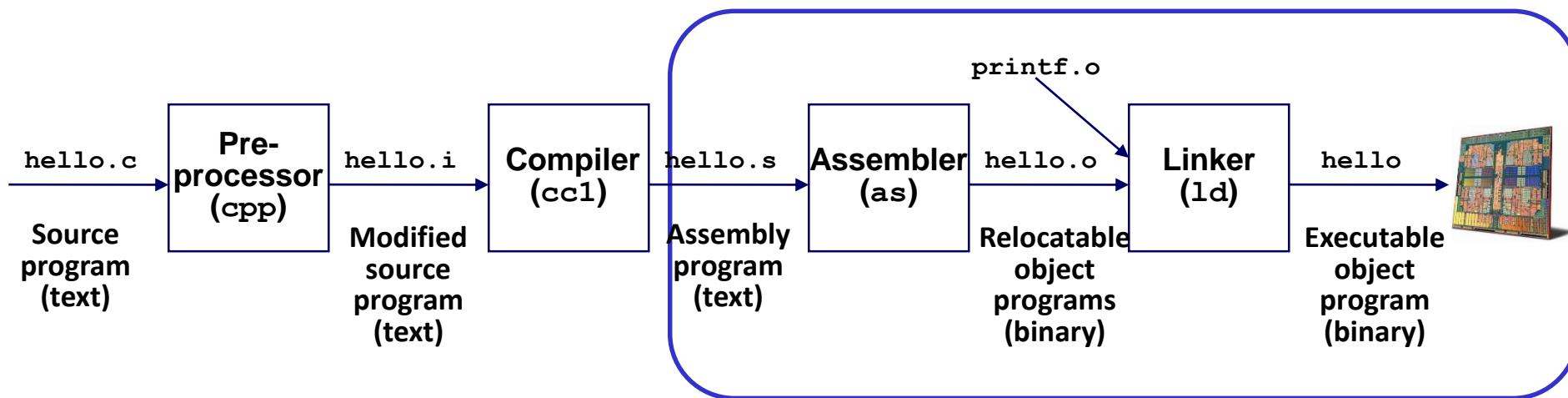
- What happens under the hood?

# hello.c

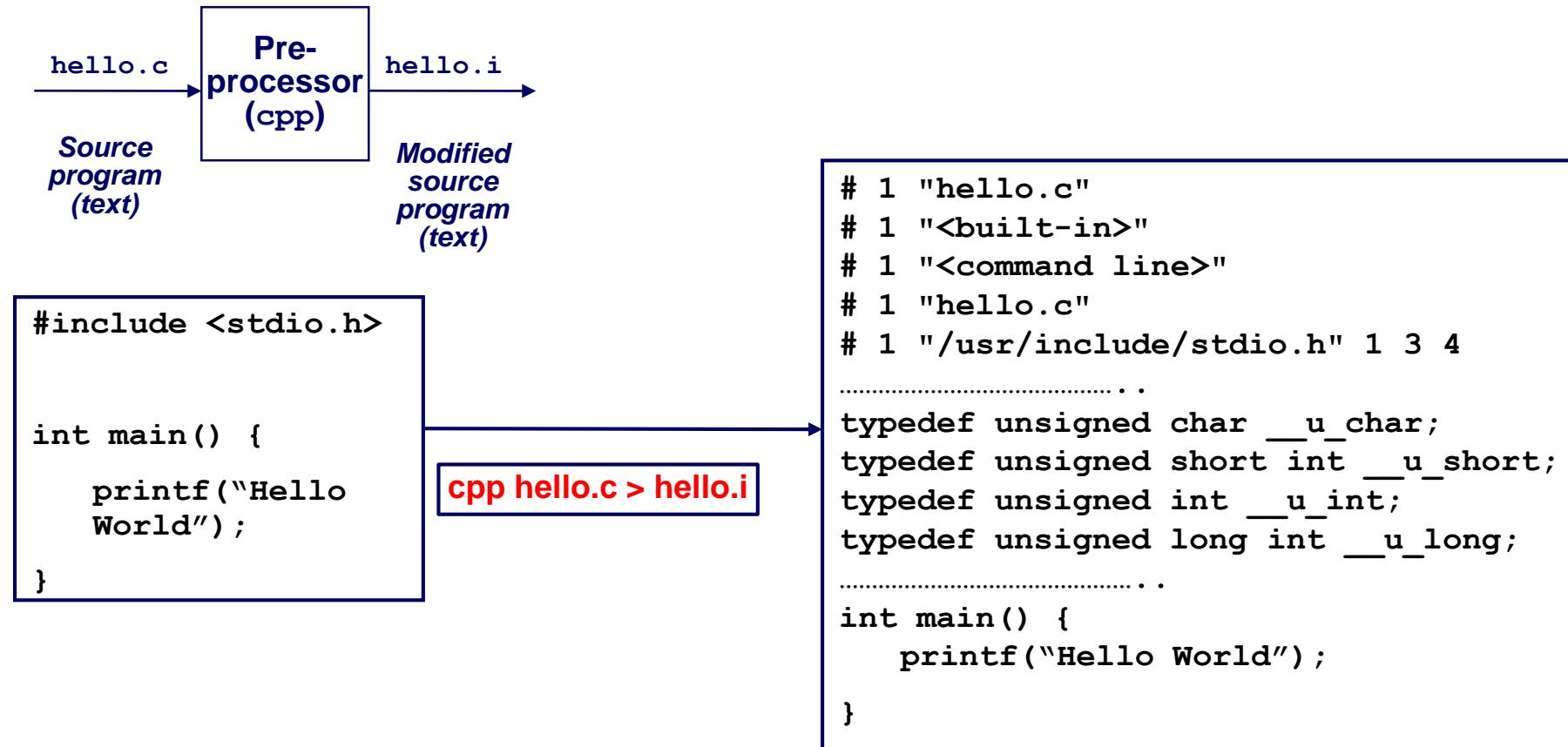
```
#include <stdio.h>

int main() {
    printf("Hello World");
}
```

# Compilation of hello.c



# Preprocessing



# Compiler

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
.....
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
.....
int main() {
    printf("Hello World");
}
```

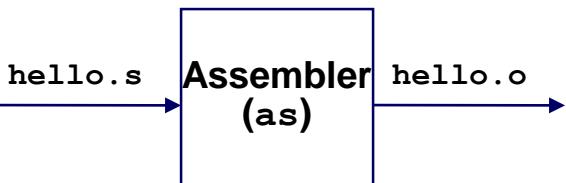


gcc -Wall -S hello.i > hello.s

```
.file  "hello.c"
      .section
      .rodata
.LC0:
      .string "Hello
World"
      .text
.globl main
      .type   main,
@function
main:
      pushl  %ebp
      movl   %esp, %ebp
      subl   $8, %esp
      andl   $-16, %esp
      movl   $0, %eax
      addl   $15, %eax
      addl   $15, %eax
      shrll  $4, %eax
      sall   $4, %eax
      subl   %eax, %esp
      subl   $12, %esp
      pushl  $.LC0
      call   printf
      addl   $16, %esp
      leave
      ret
      .size   main, .-
main
      .section
.note.GNU-
stack,"",@progbits
      .ident  "GCC:
(GNU) 3.4.1"
```

# Assembler

```
.file "hello.c"
      .section
.rodata
.LC0:
      .string "Hello
World"
      .text
.globl main
      .type main,
@function
main:
      pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      andl $-16, %esp
      movl $0, %eax
      addl $15, %eax
      addl $15, %eax
      shrcl $4, %eax
      sall $4, %eax
      subl %eax, %esp
      subl $12, %esp
      pushl $.LC0
      call printf
      addl $16, %esp
      leave
      ret
      .size main, .-
main
      .section
.note.GNU-
stack,"",@progbits
      .ident "GCC:
(GNU) 3.4.1"
```



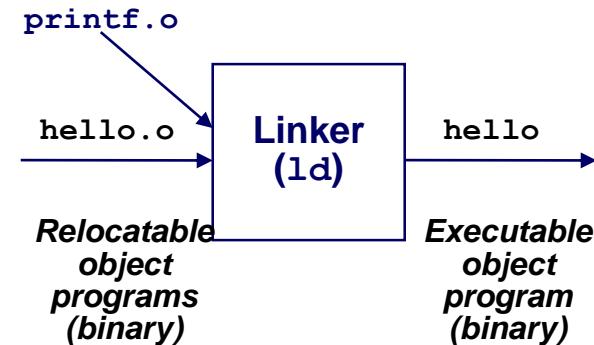
as hello.s -o hello.o

```
0000500 nul nul nul nul esc nul nul nul ht nul nul nul nul nul
0000520 nul nul nul nul d etx nul nul dle nul nul nul ht nul nul nul
0000540 soh nul nul nul eot nul nul nul bs nul nul nul % nul nul nul
0000560 soh nul nul nul etx nul nul nul nul nul nul d nul nul nul
0000600 nul eot nul nul nul
0000620 nul nul nul nul + nul nul nul bs nul nul nul etx nul nul nul
0000640 nul nul nul d nul nul
0000660 nul nul nul eot nul nul nul nul nul nul nul 0 nul nul nul
0000700 soh nul nul nul stx nul nul nul nul nul nul d nul nul nul
0000720 ff nul nul nul nul nul nul nul nul nul soh nul nul nul
0000740 nul nul nul 8 nul nul nul soh nul nul nul nul nul nul nul
0000760 nul nul nul p nul nul
0001000 nul nul nul soh nul nul nul nul nul nul nul H nul nul nul
0001020 soh nul nul nul nul nul nul nul nul nul p nul nul nul
0001040 2 nul nul nul nul nul nul nul nul nul soh nul nul nul
0001060 nul nul nul dc1 nul nul nul etx nul nul nul nul nul nul
0001100 nul nul nul " nul nul nul Q nul nul nul nul nul nul
0001120 nul nul nul soh nul nul nul nul nul nul nul soh nul nul
0001140 stx nul nul nul nul nul nul nul nul , stx nul nul
0001160 sp nul nul nul nl nul nul nul bs nul nul nul eot nul nul nul
0001200 dle nul nul ht nul nul nul etx nul nul nul nul nul nul
0001220 nul nul nul L etx nul nul nak nul nul nul nul nul nul
0001240 nul nul nul soh nul nul nul nul nul nul nul nul nul nul
0001260 nul nul nul nul nul nul nul nul soh nul nul nul
0001300 nul nul nul nul nul nul nul eot nul q del nul nul nul nul
0001320 nul nul nul nul nul nul etx nul soh nul nul nul nul nul
0001340 nul nul nul nul nul nul etx nul etx nul nul nul nul nul
0001360 nul nul nul nul nul nul etx nul etx nul eot nul nul nul
0001400 nul nul nul nul nul nul etx nul enq nul nul nul nul nul
0001420 nul nul nul nul nul nul etx nul ack nul nul nul nul nul
0001440 nul nul nul nul nul nul etx nul bel nul ht nul nul nul
0001460 nul nul nul . nul nul nul dc2 nul soh nul so nul nul nul
0001500 nul nul nul nul nul nul die nul nul nul nul h e l
0001520 l o . c nul m a i n nul p r i n t f
0001540 nul nul nul nul sp nul nul soh enq nul nul % nul nul nul
0001560 stx ht nul nul
0001564
```

od -a hello.o

# Linker

```
0000500 nul nul nul nul esc nul nul ht nul nul nul nul nul nul  
0000520 nul nul nul nul d etx nul nul dle nul nul nul ht nul nul nul  
0000540 soh nul nul nul eot nul nul nul bs nul nul nul % nul nul nul  
0000560 soh nul nul nul etx nul nul nul nul nul nul d nul nul nul  
0000600 nul eot nul nul nul  
0000620 nul nul nul nul + nul nul nul bs nul nul nul etx nul nul nul  
0000640 nul nul nul nul d nul  
0000660 nul nul nul nul eot nul nul nul nul nul nul nul 0 nul nul nul  
0000700 soh nul nul stx nul nul nul nul nul nul d nul nul nul  
0000720 ff nul nul nul nul nul nul nul nul soh nul nul nul  
0000740 nul nul nul nul 8 nul nul nul soh nul nul nul nul nul nul  
0000760 nul nul nul p nul  
0001000 nul nul nul soh nul nul nul nul nul nul H nul nul nul  
0001020 soh nul nul nul nul nul nul nul nul p nul nul nul  
0001040 2 nul nul nul nul nul nul nul nul soh nul nul nul  
0001060 nul nul nul nul dc1 nul nul nul etx nul nul nul nul nul  
0001100 nul nul nul nul " nul nul nul Q nul nul nul nul nul nul  
0001120 nul nul nul nul soh nul nul nul nul nul nul soh nul nul nul  
0001140 stx nul nul nul nul nul nul nul nul , stx nul nul  
0001160 sp nul nul nul nl nul nul nul bs nul nul nul eot nul nul nul  
0001200 dle nul nul nul ht nul nul nul etx nul nul nul nul nul nul  
0001220 nul nul nul nul L etx nul nul nak nul nul nul nul nul nul  
0001240 nul nul nul nul soh nul  
0001260 nul soh nul nul nul  
0001300 nul nul nul nul nul nul nul eot nul q del nul nul nul nul  
0001320 nul nul nul nul nul nul nul etx nul soh nul nul nul nul nul  
0001340 nul nul nul nul nul nul nul etx nul etx nul nul nul nul nul  
0001360 nul nul nul nul nul nul nul etx nul eot nul nul nul nul nul  
0001400 nul nul nul nul nul nul etx nul eng nul nul nul nul nul  
0001420 nul nul nul nul nul nul etx nul ack nul nul nul nul nul  
0001440 nul nul nul nul nul nul nul etx nul bel nul ht nul nul nul  
0001460 nul nul nul nul . nul nul nul dc2 nul soh nul so nul nul nul  
0001500 nul nul nul nul nul nul dle nul nul nul nul h e l  
0001520 l o . c nul m a i n nul p r i n t f  
0001540 nul nul nul sp nul nul nul soh enq nul nul % nul nul nul  
0001560 stx ht nul nul  
0001564
```



gcc hello.o -o hello

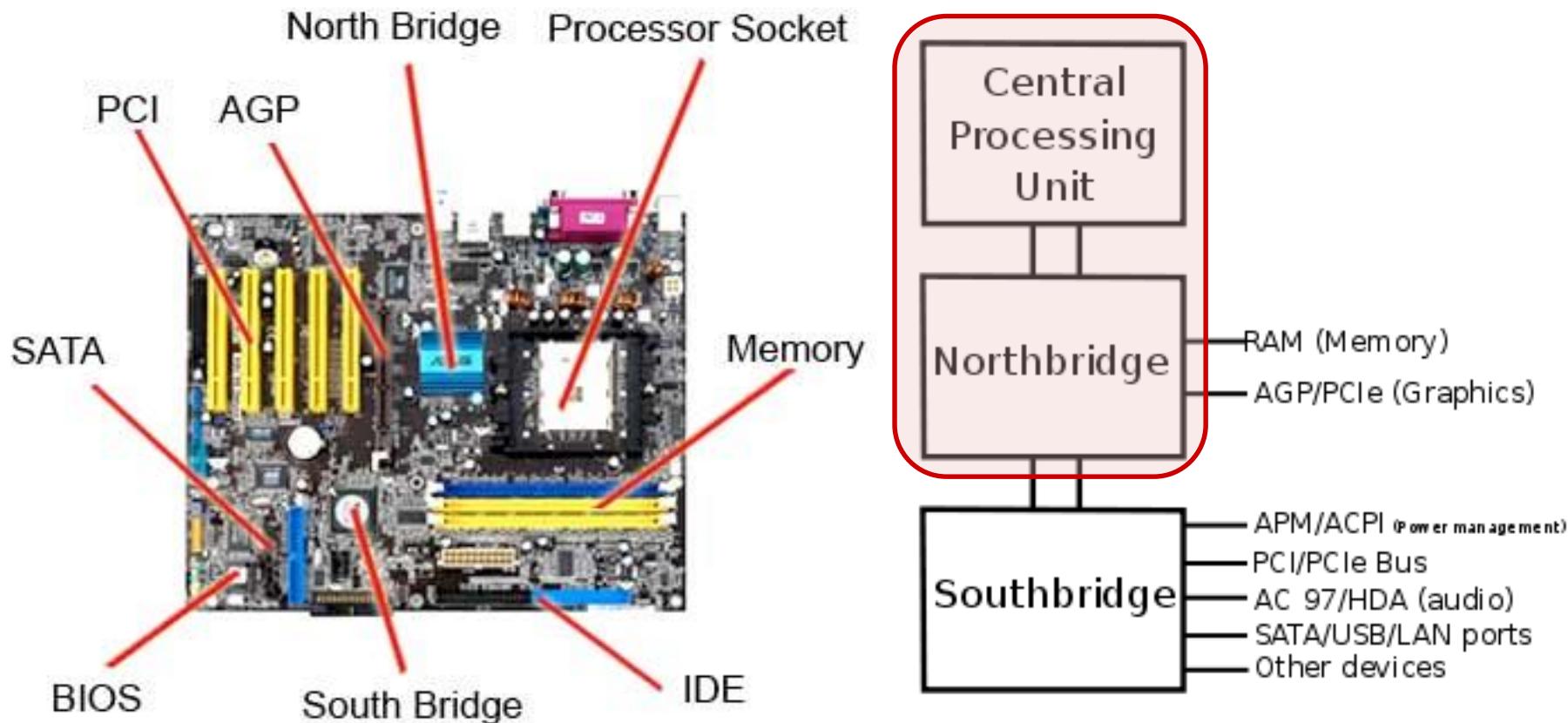
```
0000000 del E L F soh soh soh nul nul nul nul nul nul nul  
0000020 stx nul etx nul soh nul nul @ stx eot bs 4 nul nul nul  
0000040 X cr nul nul nul nul nul 4 nul sp nul bel nul ( nul  
0000060 ! nul rs nul ack nul nul nul 4 nul nul nul 4 nul eot bs  
0000100 4 nul eot bs ` nul nul nul ` nul nul nul enq nul nul nul  
0000120 eot nul nul nul etx nul nul nul dc4 soh nul nul dc4 soh eot bs  
0000140 dc4 soh eot bs dc3 nul nul nul dc3 nul nul nul eot nul nul nul  
0000160 soh nul nul nul soh nul nul nul nul nul nul nul eot bs  
0000200 nul nul eot bs bs eot nul nul enq nul nul nul  
0000220 nul dle nul nul soh nul nul nul bs eot nul nul bs dc4 eot bs  
0000240 bs dc4 eot bs nul soh nul nul eot soh nul nul ack nul nul nul  
0000260 nul dle nul nul stx nul nul nul dc4 eot nul nul dc4 dc4 eot bs  
0000300 dc4 dc4 eot bs H nul nul nul H nul nul nul ack nul nul  
0000320 eot nul nul nul eot nul nul nul ( soh nul nul ( soh eot bs  
0000340 ( soh eot bs sp nul nul nul sp nul nul nul eot nul nul nul  
0000360 eot nul nul nul Q e t d nul nul nul nul nul nul nul  
0000400 nul  
0000420 eot nul nul nul / l i b / l d - l i n u  
0000440 x . s o . 2 nul nul eot nul nul nul die nul nul nul  
0000460 soh nul nul nul G N U nul nul nul nul stx nul nul nul  
0000500 stx nul nul nul enq nul nul nul etx nul nul nul ack nul nul nul  
0000520 enq nul nul nul soh nul nul nul etx nul nul nul nul nul nul  
0000540 nul nul nul nul nul nul nul stx nul nul nul nul nul nul nul  
.....
```

od -a hello

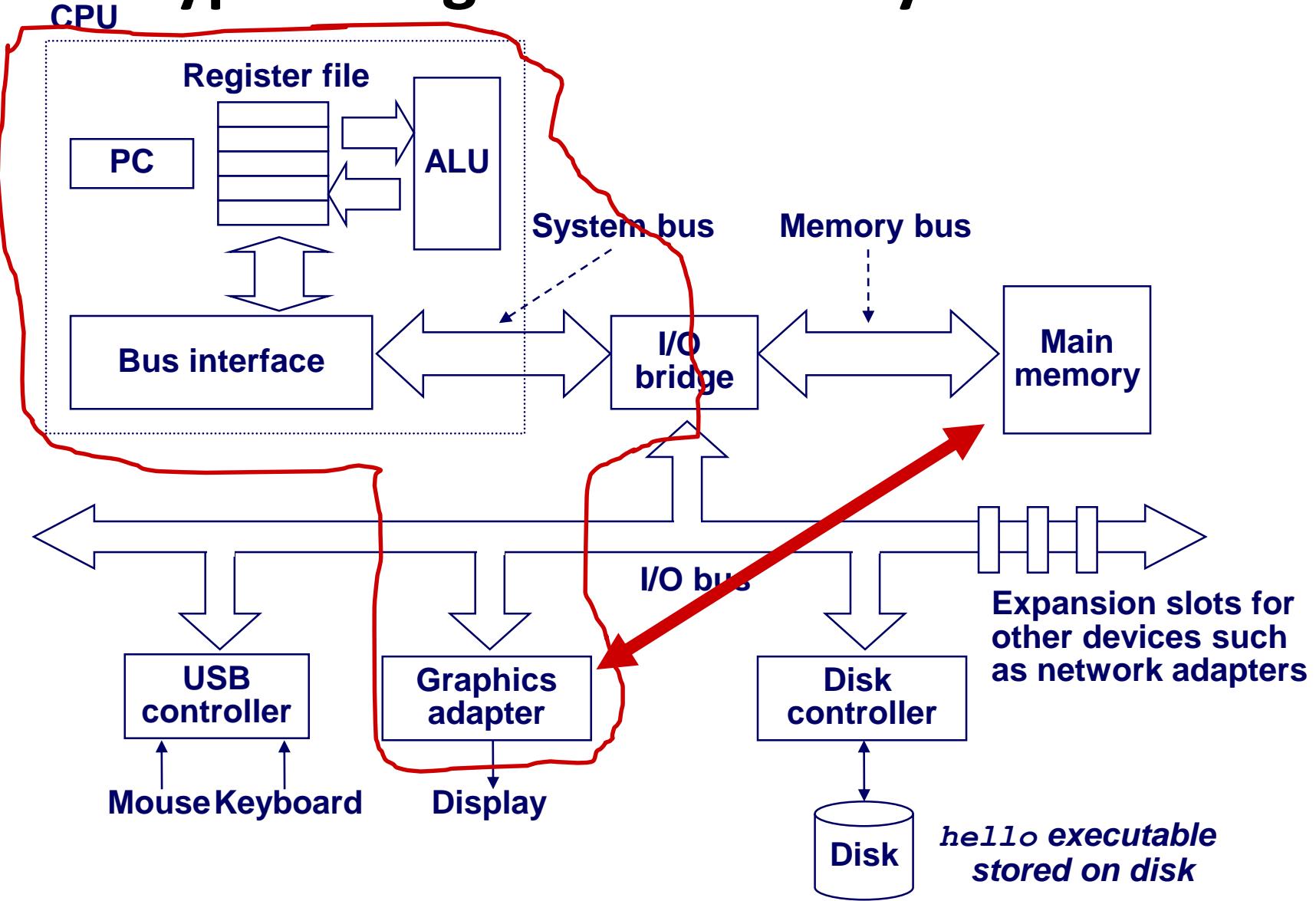
# Finally...

```
$ gcc hello.o -o hello  
$ ./hello  
Hello World$
```

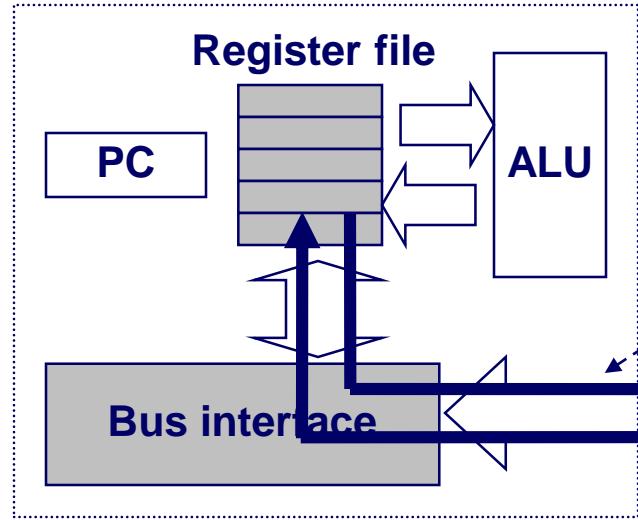
# How do you say “Hello World”?



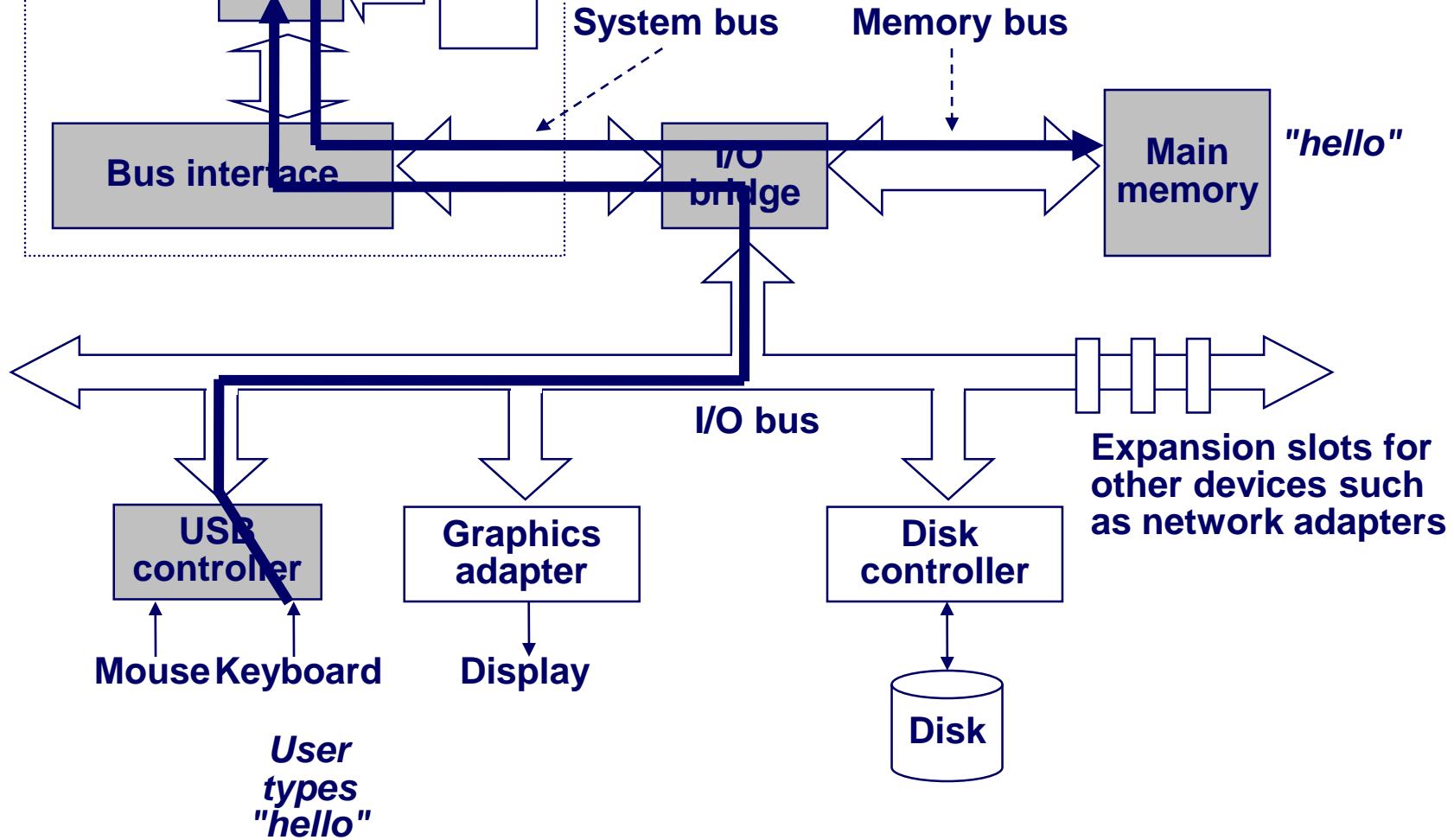
# Typical Organization of System



CPU



Reading hello command from keyboard



# Today: Bits and Bytes

- Compiling, linking and executing: Hello World
- Representing information as bits
- Bit-level manipulations

# Logical Systems in Computers

## ■ Binary ( 0 and 1)

- Example: computers we are using today

## ■ Ternary ( -1, 0 , +1)

- Example:

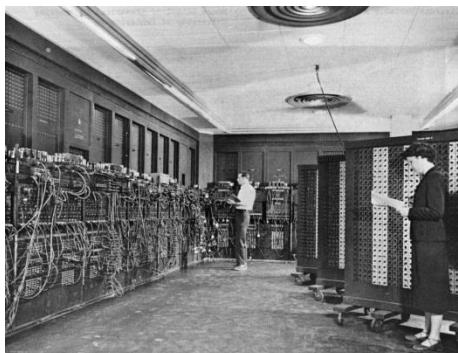


Source: <https://en.wikipedia.org/wiki/Setun>

Setun ternary computer designed by Nikolay Brusentsov in the Soviet Union (1958 Moscow State University)

## ■ Decimal

- Example:

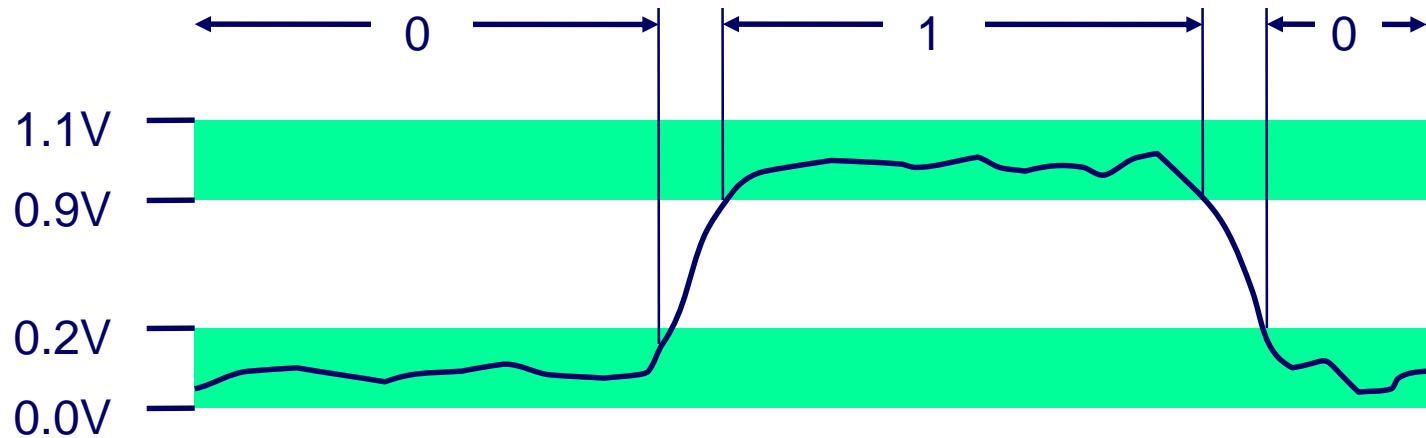


Source: <https://en.wikipedia.org/wiki/ENIAC>

ENIAC - Designed by John Mauchly and J. Presper Eckert at the University of Pennsylvania, U.S in ~1943.

# Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



# For example, can count in binary

## ■ Base 2 Number Representation

- Represent  $15213_{10}$  as  $11101101101101_2$
- Represent  $1.20_{10}$  as  $1.0011001100110011[0011]\dots_2$
- Represent  $1.5213 \times 10^4$  as  $1.1101101101101_2 \times 2^{13}$

# Encoding Byte Values

## ■ Byte = 8 bits

- Binary 0000000<sub>2</sub> to 1111111<sub>2</sub>
- Decimal: 0<sub>10</sub> to 255<sub>10</sub>
- Hexadecimal 00<sub>16</sub> to FF<sub>16</sub>
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B<sub>16</sub> in C as
    - 0xFA1D37B
    - 0xfa1d37b

**Q: Why a byte is 8-bits ?**

**A: Due to IBM 360 (~1964)**

**John von Neumann:** “Young man, in mathematics you don't understand things. You just get used to them.”

Reply, according to Dr. Felix T. Smith of Stanford Research Institute, to a physicist friend who had said "I'm afraid I don't understand the method of characteristics," as quoted in The Dancing Wu Li Masters: An Overview of the New Physics (1979) by Gary Zukav, Bantam Books, p. 208, footnote.

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<b>char</b>	1	1	1
<b>short</b>	2	2	2
<b>int</b>	4	4	4
<b>long</b>	4	8	8
<b>float</b>	4	4	4
<b>double</b>	8	8	8
<b>long double</b>	-	-	10/16
<b>pointer</b>	4	8	8

# Today: Bits and Bytes

- Compiling, linking and executing: Hello World
- Representing information as bits
- Bit-level manipulations

# Boolean Algebra

## ■ Developed by George Boole in 19th Century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And

- $A \& B = 1$  when both  $A=1$  and  $B=1$

&	0	1
0	0	0
1	0	1

Or

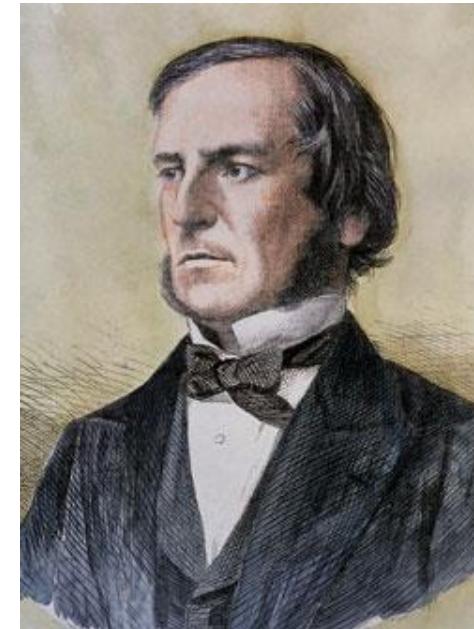
- $A | B = 1$  when either  $A=1$  or  $B=1$

	0	1
0	0	1
1	1	1

Not

- $\sim A = 1$  when  $A=0$

$\sim$	
0	1
1	0



Exclusive-Or (Xor)

- $A ^ B = 1$  when either  $A=1$  or  $B=1$ , but not both

$^$	0	1
0	0	1
1	1	0

# General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

$$\begin{array}{rcl} \begin{array}{c} 01101001 \\ \& 01010101 \end{array} & \begin{array}{c} 01101001 \\ | \quad 01010101 \end{array} & \begin{array}{c} 01101001 \\ ^ \quad 01010101 \end{array} \\ \hline \begin{array}{c} 01000001 \\ 0111101 \end{array} & \begin{array}{c} 0111101 \\ 00111100 \end{array} & \begin{array}{c} 01010101 \\ 10101010 \end{array} \end{array}$$

- All of the Properties of Boolean Algebra Apply

# Example: Representing & Manipulating Sets

## ■ Representation

- Width w bit vector represents subsets of  $\{0, \dots, w-1\}$
- $a_j = 1$  if  $j \in A$

- 01101001       $\{0, 3, 5, 6\}$

- ~~76543210~~

- 01010101       $\{0, 2, 4, 6\}$

- ~~76543210~~

## ■ Operations

▪ & Intersection	01000001	$\{0, 6\}$
▪   Union	01111101	$\{0, 2, 3, 4, 5, 6\}$
▪ ^ Symmetric difference	00111100	$\{2, 3, 4, 5\}$
▪ ~ Complement	10101010	$\{1, 3, 5, 7\}$

# Bit-Level Operations in C

## ■ Operations &, |, ~, ^ Available in C

- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

## ■ Examples (Char data type)

- $\sim 0x41 \rightarrow 0xBE$ 
  - $\sim 01000001_2 \rightarrow 10111110_2$
- $\sim 0x00 \rightarrow 0xFF$ 
  - $\sim 00000000_2 \rightarrow 11111111_2$
- $0x69 \& 0x55 \rightarrow 0x41$ 
  - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
- $0x69 | 0x55 \rightarrow 0x7D$ 
  - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

# Contrast: Logic Operations in C

## ■ Contrast to Logical Operators

- `&&`, `||`, `!`
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

Watch out for `&&` vs. `&` (and `||` vs. `|`)...  
one of the more common oopsies in  
C programming

## ■ Examples (char data type)

- `!0x41 → 0x00`
- `!0x00 → 0x01`
- `!!0x41 → 0x01`
  
- `0x69 && 0x55 → 0x01`
- `0x69 || 0x55 → 0x01`
- `p && *p` (avoids null pointer access)

# Shift Operations

## ■ Left Shift: $x \ll y$

- Shift bit-vector  $x$  left  $y$  positions
  - Throw away extra bits on left
  - Fill with 0's on right

## ■ Right Shift: $x \gg y$

- Shift bit-vector  $x$  right  $y$  positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left

## ■ Undefined Behavior

- Shift amount  $< 0$  or  $\geq$  word size

Argument $x$	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument $x$	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

*Thank you !*

# Integers

CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu      (Sections 1-2)

Unless otherwise noted adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Today: Integers

## ■ Integers

- **Representation: unsigned and signed**
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary

## ■ Representations in memory, pointers, strings

# Encoding Integers

## Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

## Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;  
short int y = -15213;
```

Sign  
Bit

### ■ C short 2 bytes long

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

### ■ Sign Bit

- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

# Two-complement Encoding Example (Cont.)

x =	15213:	00111011	01101101
y =	-15213:	11000100	10010011

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
	Sum		15213	
			-15213	

# Numeric Ranges

## ■ Unsigned Values

- $UMin = 0$   
000...0
- $UMax = 2^w - 1$   
111...1

## ■ Two's Complement Values

- $TMin = -2^{w-1}$   
100...0
- $TMax = 2^{w-1} - 1$   
011...1

## ■ Other Values

- Minus 1  
111...1

## Values for $W = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

# Values for Different Word Sizes

	W			
	8	16	32	64
<b>UMax</b>	255	65,535	4,294,967,295	18,446,744,073,709,551,615
<b>TMax</b>	127	32,767	2,147,483,647	9,223,372,036,854,775,807
<b>TMin</b>	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

## ■ Observations

- $|TMin| = TMax + 1$ 
  - Asymmetric range
- $UMax = 2 * TMax + 1$

## ■ C Programming

- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific

# Unsigned & Signed Numeric Values

$X$	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

## ■ Equivalence

- Same encodings for nonnegative values

## ■ Uniqueness

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

## ■ $\Rightarrow$ Can Invert Mappings

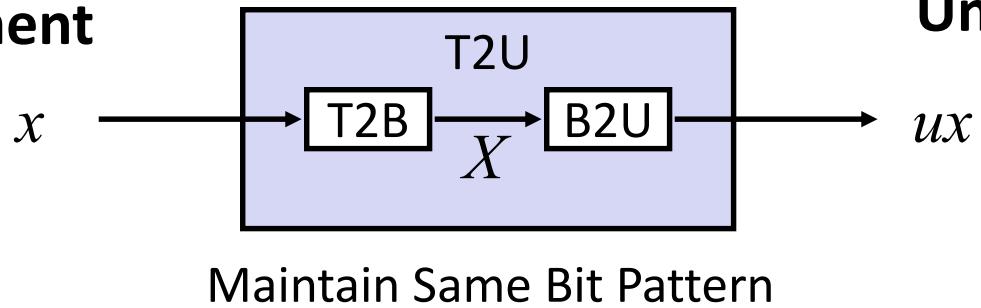
- $U2B(x) = B2U^{-1}(x)$ 
  - Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$ 
  - Bit pattern for two's comp integer

# Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

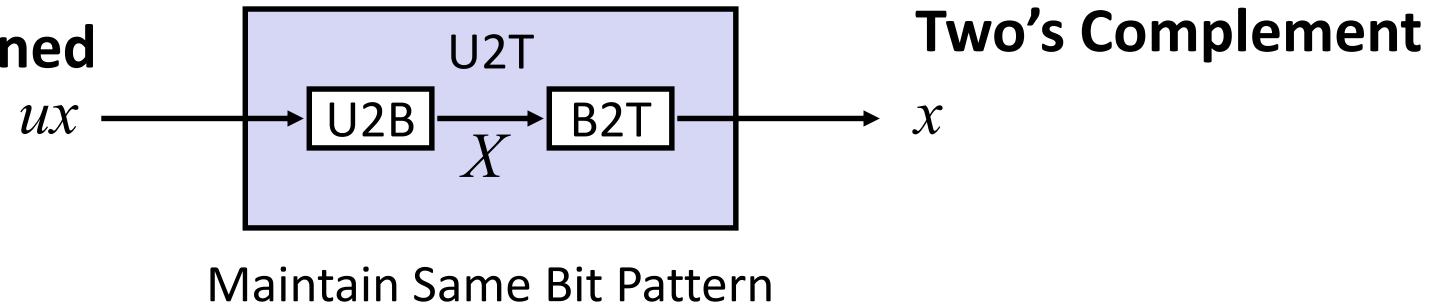
# Mapping Between Signed & Unsigned

Two's Complement



Unsigned

Unsigned



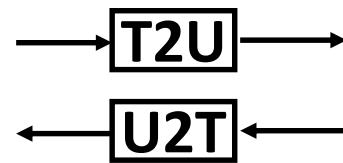
Two's Complement

- Mappings between unsigned and two's complement numbers:  
**Keep bit representations and reinterpret**

# Mapping Signed ↔ Unsigned

Bits
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Signed
0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1



Unsigned
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

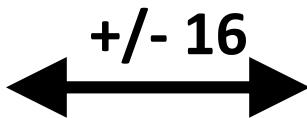
# Mapping Signed $\leftrightarrow$ Unsigned

Bits
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

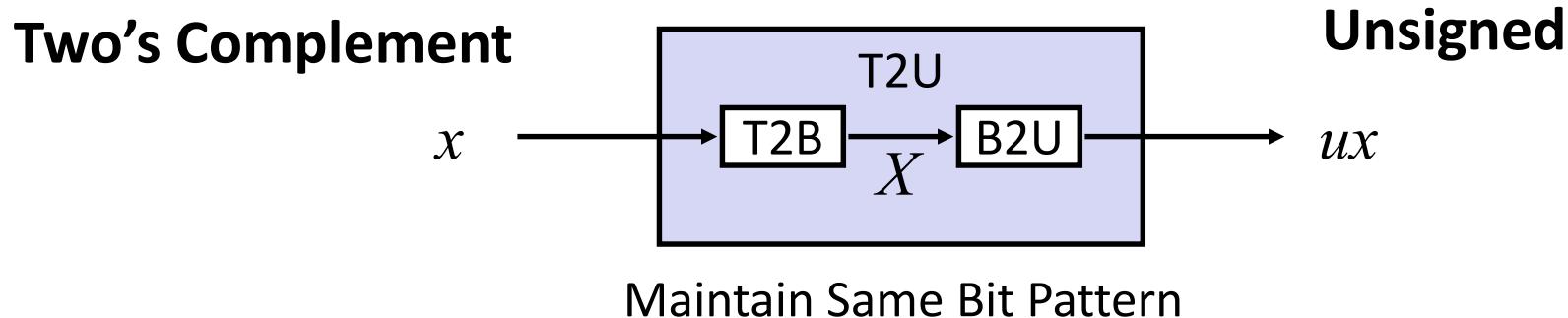
Signed
0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1



Unsigned
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15



# Relation between Signed & Unsigned



$w-1$                                     0

$ux$ 

+	+	+	•••	+	+	+
---	---	---	-----	---	---	---

$x$ 

-	+	+	•••	+	+	+
---	---	---	-----	---	---	---

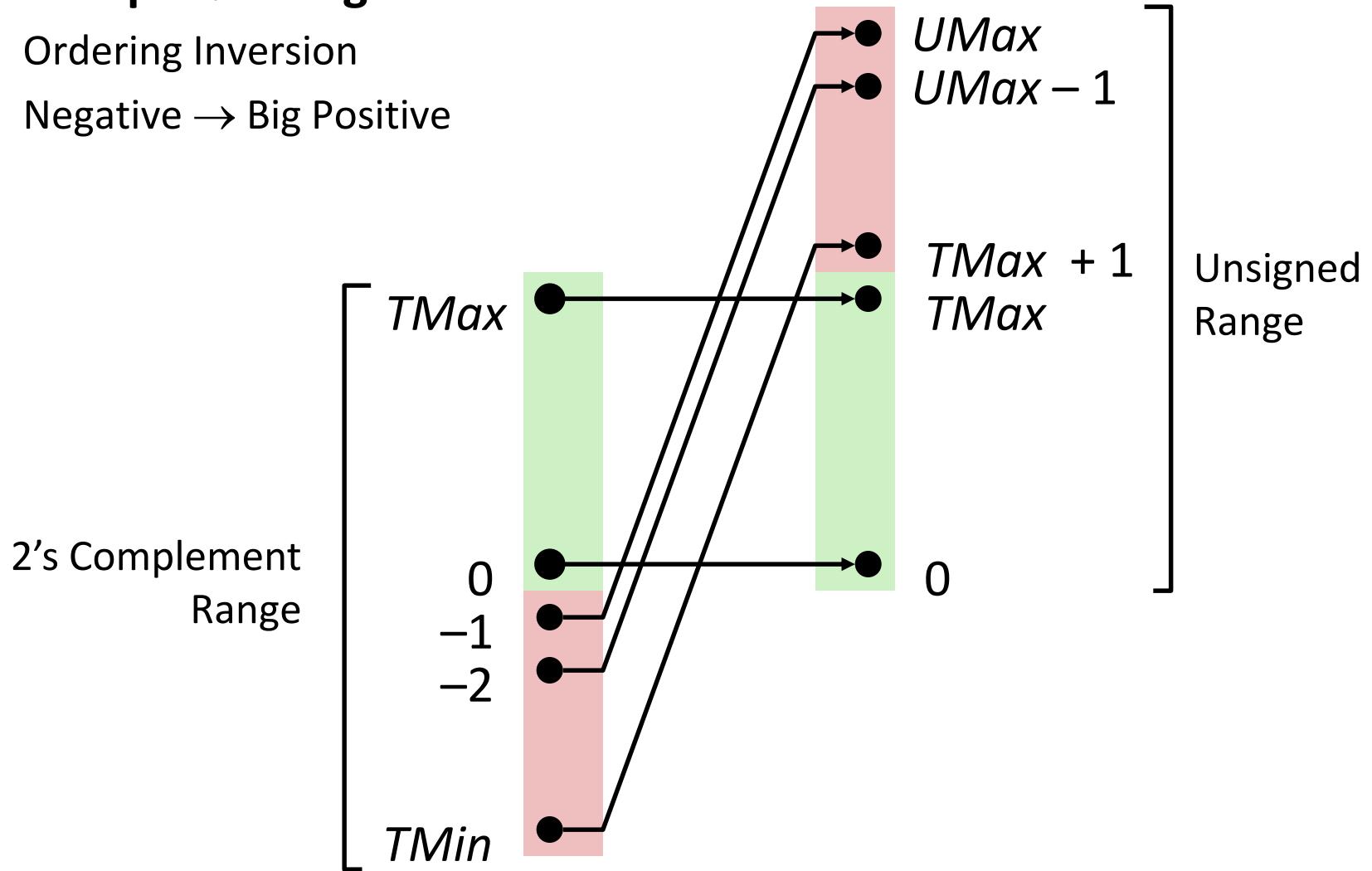


Large negative weight  
becomes  
Large positive weight

# Conversion Visualized

## ■ 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



# Signed vs. Unsigned in C

## ■ Constants

- By default are considered to be signed integers
- Unsigned if have “U” as suffix

**0U, 4294967259U**

## ■ Casting

- Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux;  
uy = (unsigned) ty;
```

- Implicit casting also occurs via assignments and procedure calls

```
tx = ux;  
uy = ty;
```

# Casting Surprises

## ■ Expression Evaluation

- If there is a mix of unsigned and signed in single expression,  
*signed values implicitly cast to unsigned*
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for  $W = 32$ : **TMIN = -2,147,483,648**, **TMAX = 2,147,483,647**

■ Constant <sub>1</sub>	Constant <sub>2</sub>	Relation	Evaluation
0	0U	<code>==</code>	unsigned
-1	0	<code>&lt;</code>	signed
-1	0U	<code>&gt;</code>	unsigned
2147483647	-2147483647-1	<code>&gt;</code>	signed
2147483647U	-2147483647-1	<code>&lt;</code>	unsigned
-1	-2	<code>&gt;</code>	signed
(unsigned)-1	-2	<code>&gt;</code>	unsigned
2147483647	2147483648U	<code>&lt;</code>	unsigned
2147483647	(int) 2147483648U	<code>&gt;</code>	signed

# Summary

## Casting Signed $\leftrightarrow$ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting  $2^w$
  
- Expression containing signed and unsigned int
  - int is cast to unsigned!!

# Today: Integers

## ■ Integers

- Representation: unsigned and signed
- Conversion, casting
- **Expanding, truncating**
- Addition, negation, multiplication, shifting
- Summary

## ■ Representations in memory, pointers, strings

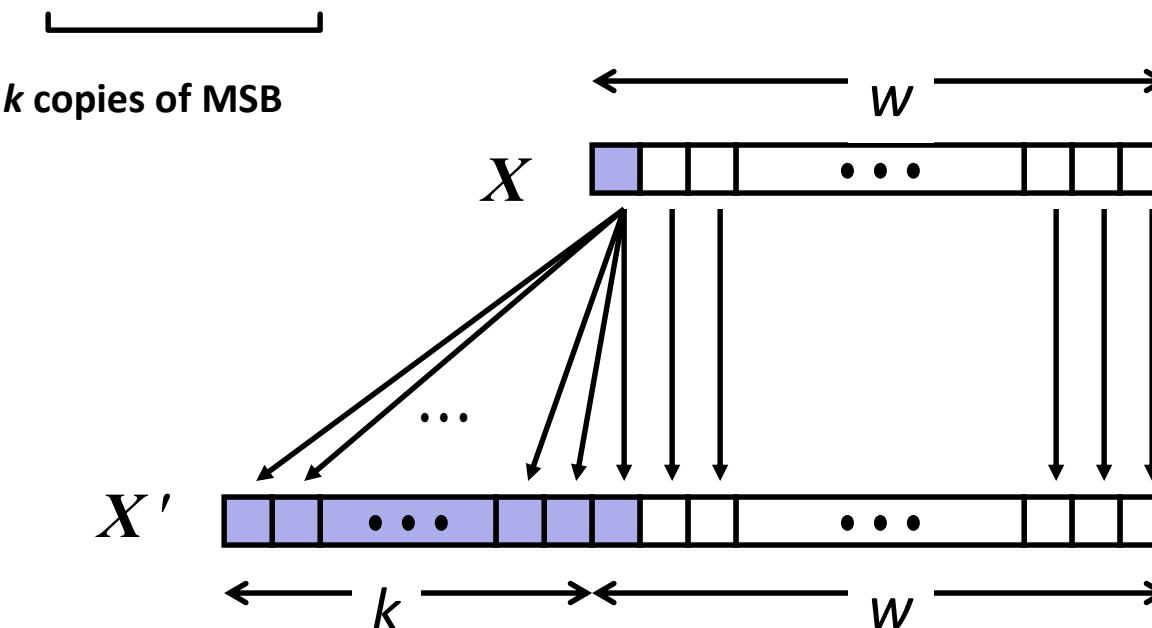
# Sign Extension

## ■ Task:

- Given  $w$ -bit signed integer  $x$
- Convert it to  $w+k$ -bit integer with same value

## ■ Rule:

- Make  $k$  copies of sign bit:
- $X' = x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0$



# Sign Extension Example

```
short int x = 15213;  
int      ix = (int) x;  
short int y = -15213;  
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

## **Summary:**

# **Expanding, Truncating: Basic Rules**

### **■ Expanding (e.g., short int to int)**

- Unsigned: zeros added
- Signed: sign extension
- Both yield expected result

### **■ Truncating (e.g., unsigned to unsigned short)**

- Unsigned/signed: bits are truncated
- Result reinterpreted
- Unsigned: mod operation
- Signed: similar to mod
  - For small numbers yields expected behavior
  - For others?

# Today: Integers

## ■ Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- **Addition, negation, multiplication, shifting**

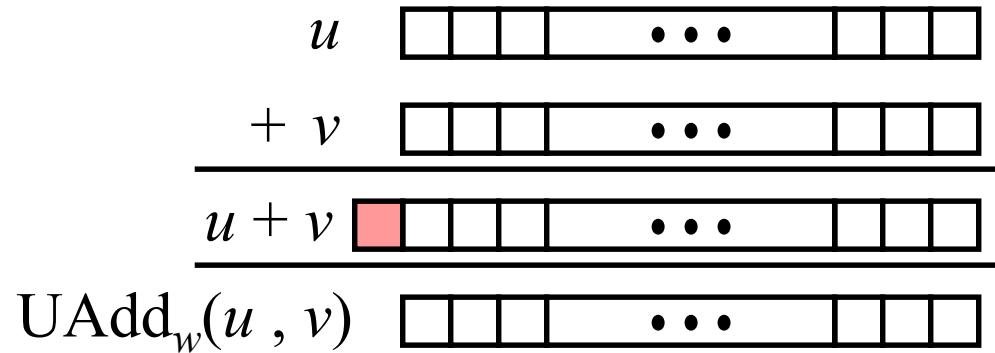
## ■ Representations in memory, pointers, strings

# Unsigned Addition

Operands:  $w$  bits

True Sum:  $w+1$  bits

Discard Carry:  $w$  bits



## ■ Standard Addition Function

- Ignores carry output

## ■ Implements Modular Arithmetic

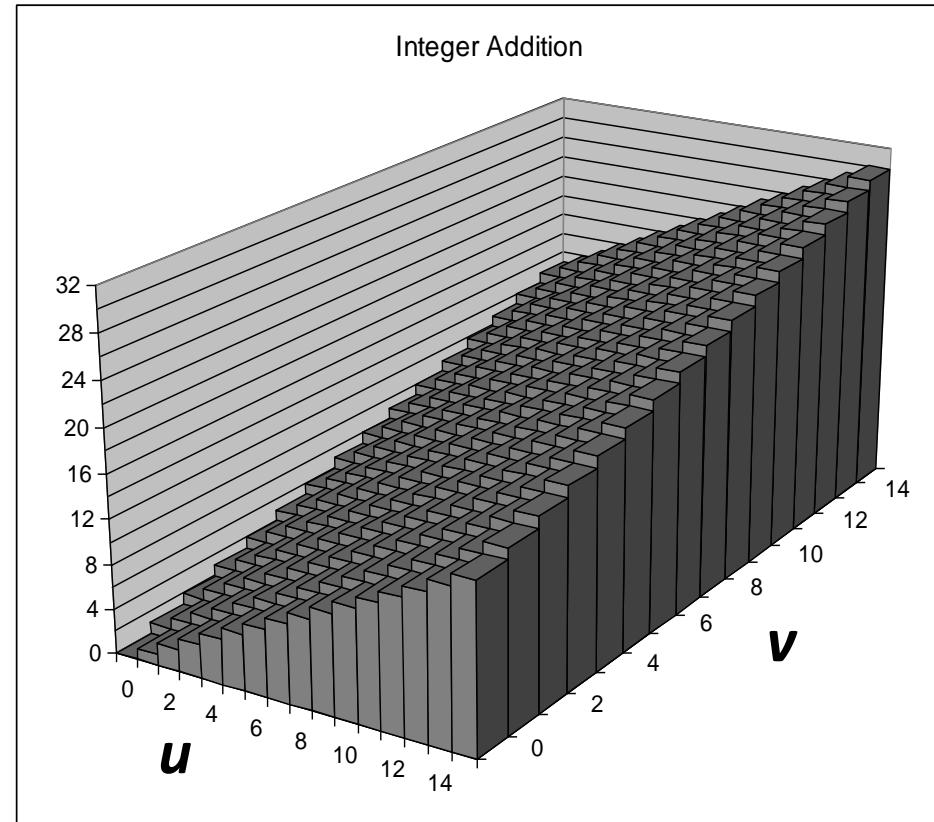
$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

# Visualizing (Mathematical) Integer Addition

## ■ Integer Addition

- 4-bit integers  $u, v$
- Compute true sum  
 $\text{Add}_4(u, v)$
- Values increase linearly  
with  $u$  and  $v$
- Forms planar surface

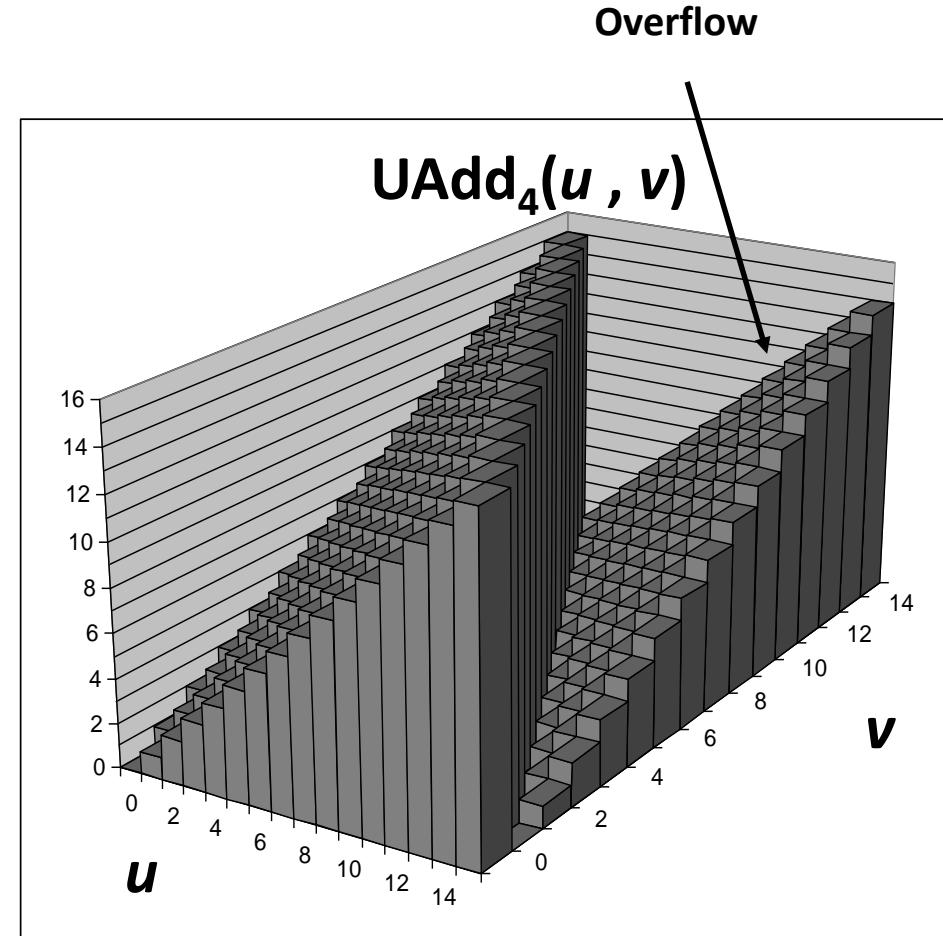
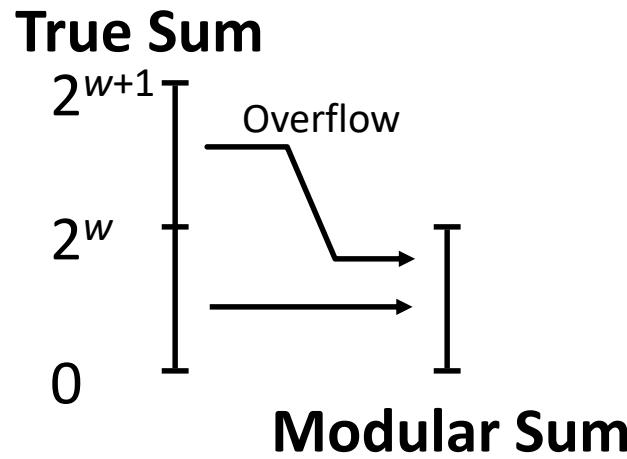
$\text{Add}_4(u, v)$



# Visualizing Unsigned Addition

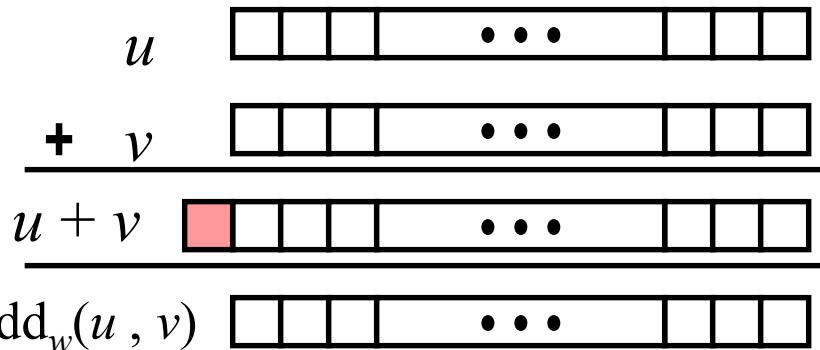
## Wraps Around

- If true sum  $\geq 2^w$
- At most once



# Two's Complement Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits

Discard Carry:  $w$  bits

$\text{TAdd}_w(u, v)$   $\boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}} \dots \boxed{\phantom{0}} \boxed{\phantom{0}}$

## ■ TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:

```
int s, t, u, v;  
s = (int) ((unsigned) u + (unsigned) v);  
t = u + v
```

- Will give  $s == t$

# TAdd Overflow

## ■ Functionality

- True sum requires  $w+1$  bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

0 111...1

0 100...0

0 000...0

1 011...1

1 000...0

## True Sum

$2^{w-1}$

$2^{w-1}-1$

0

$-2^{w-1}$

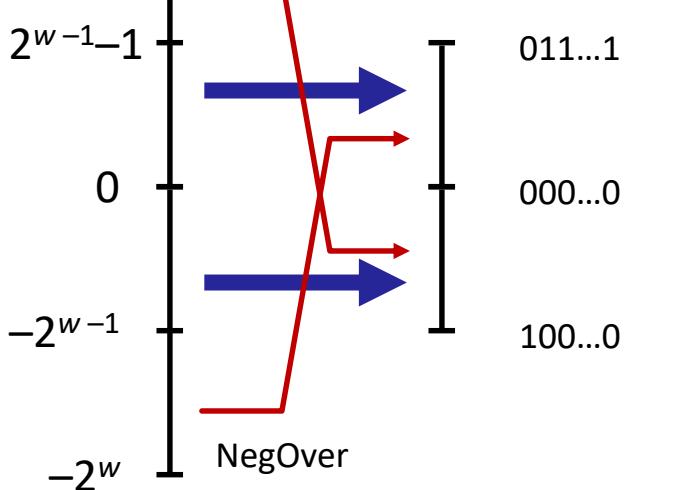
$-2^w$

## TAdd Result

011...1

000...0

100...0



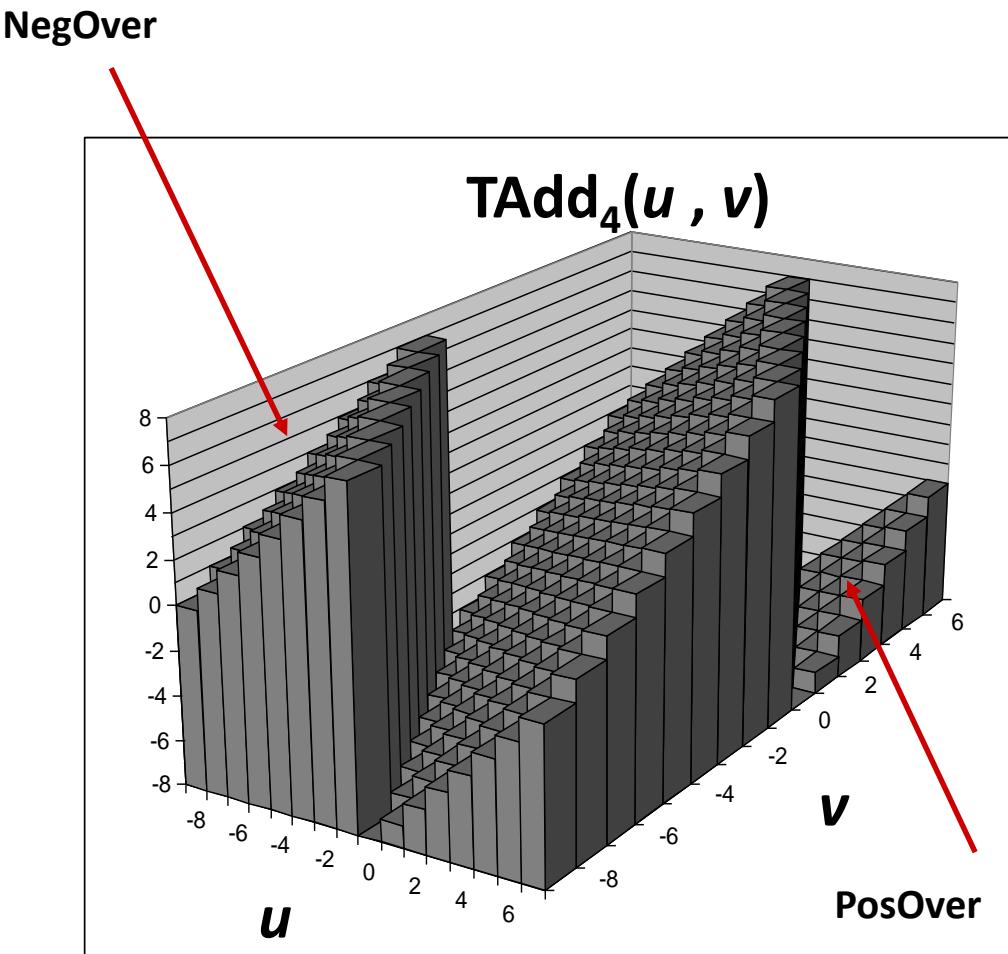
# Visualizing 2's Complement Addition

## ■ Values

- 4-bit two's comp.
- Range from -8 to +7

## ■ Wraps Around

- If  $\text{sum} \geq 2^{w-1}$ 
  - Becomes negative
  - At most once
- If  $\text{sum} < -2^{w-1}$ 
  - Becomes positive
  - At most once



# Multiplication

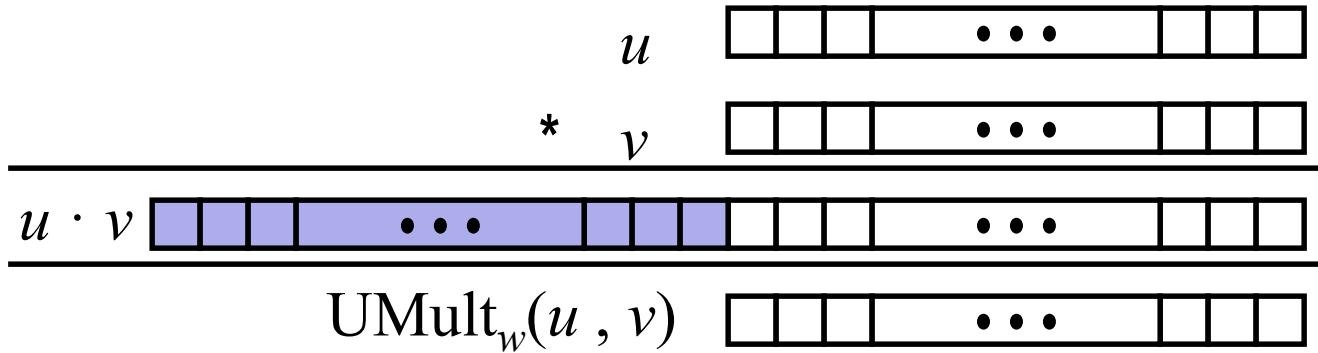
- **Goal: Computing Product of  $w$ -bit numbers  $x, y$** 
  - Either signed or unsigned
- **But, exact results can be bigger than  $w$  bits**
  - Unsigned: up to  $2w$  bits
    - Result range:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to  $2w-1$  bits
    - Result range:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to  $2w$  bits, but only for  $(TMin_w)^2$ 
    - Result range:  $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results...**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages

# Unsigned Multiplication in C

Operands:  $w$  bits

True Product:  $2^w$  bits

Discard  $w$  bits:  $w$  bits



- **Standard Multiplication Function**

- Ignores high order  $w$  bits

- **Implements Modular Arithmetic**

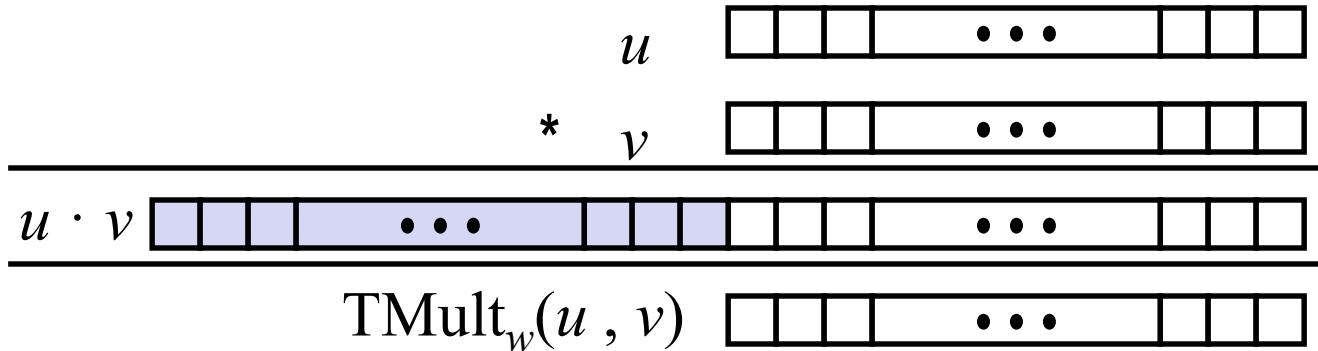
$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

# Signed Multiplication in C

Operands:  $w$  bits

True Product:  $2^w$  bits

Discard  $w$  bits:  $w$  bits



## ■ Standard Multiplication Function

- Ignores high order  $w$  bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

# Power-of-2 Multiply with Shift

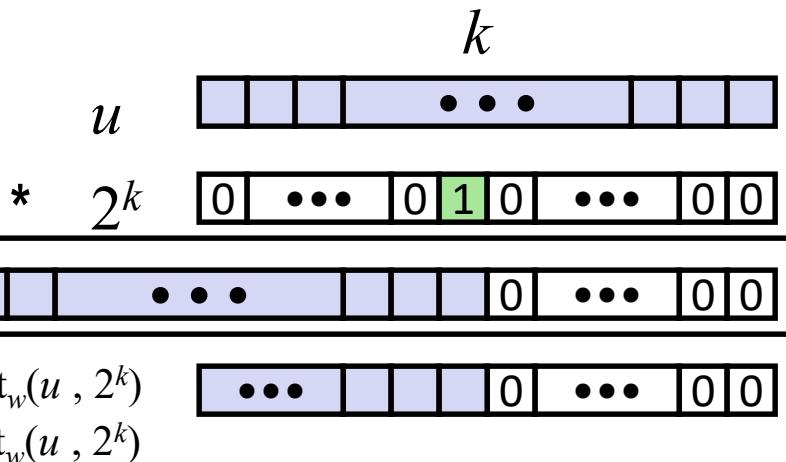
## ■ Operation

- $u \ll k$  gives  $u * 2^k$
- Both signed and unsigned

Operands:  $w$  bits

True Product:  $w+k$  bits

Discard  $k$  bits:  $w$  bits



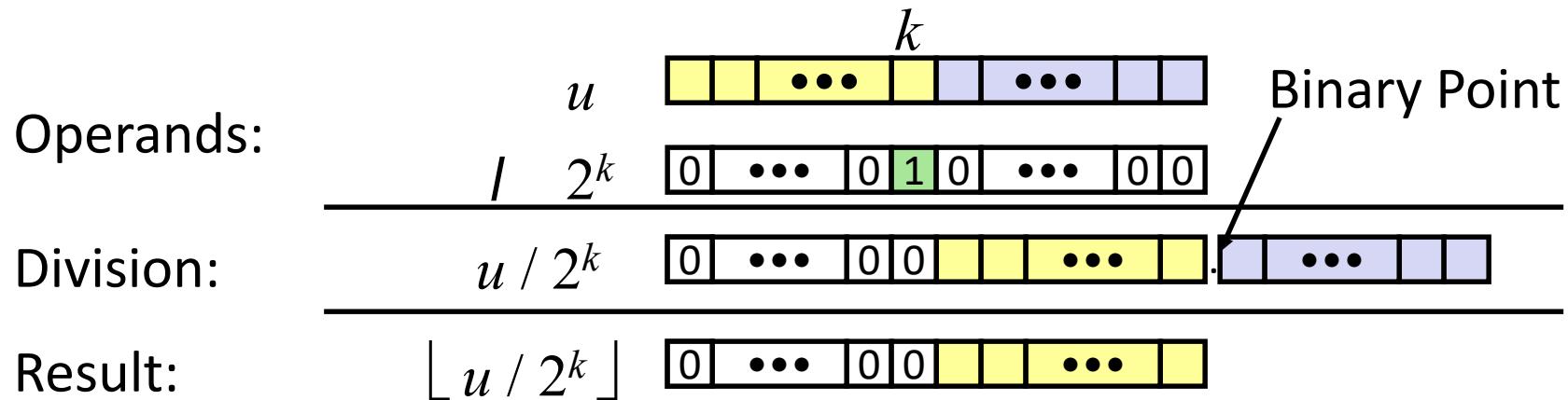
## ■ Examples

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

# Unsigned Power-of-2 Divide with Shift

## ■ Quotient of Unsigned by Power of 2

- $u \gg k$  gives  $\lfloor u / 2^k \rfloor$
- Uses logical shift



	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

# Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - **Summary**
- Representations in memory, pointers, strings

# Arithmetic: Basic Rules

## ■ Addition:

- Unsigned/signed: Normal addition followed by truncate, same operation on bit level
- Unsigned: addition mod  $2^w$ 
  - Mathematical addition + possible subtraction of  $2^w$
- Signed: modified addition mod  $2^w$  (result in proper range)
  - Mathematical addition + possible addition or subtraction of  $2^w$

## ■ Multiplication:

- Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
- Unsigned: multiplication mod  $2^w$
- Signed: modified multiplication mod  $2^w$  (result in proper range)

# Why Should I Use Unsigned? (cont.)

- ***Do Use When Performing Modular Arithmetic***
  - Multiprecision arithmetic
- ***Do Use When Using Bits to Represent Sets***
  - Logical right shift, no sign extension

Thank you!

# Representations in memory, pointers, strings

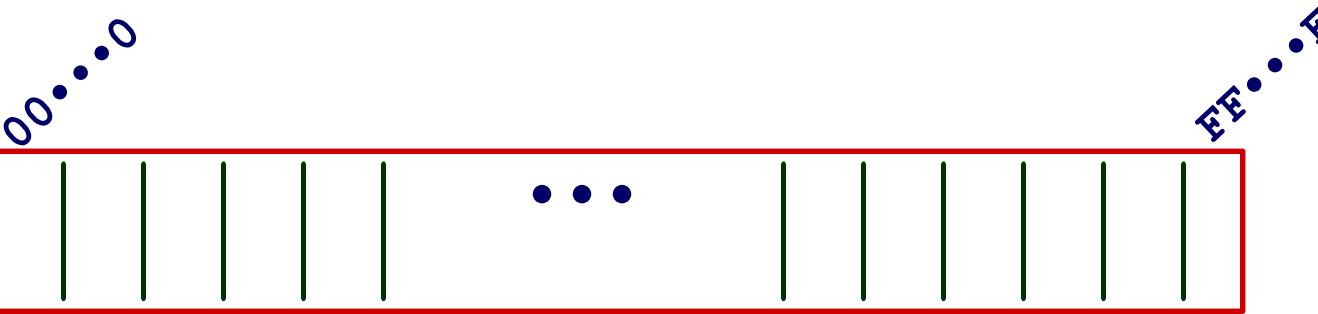
CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu      (Sections 1-2)

Unless otherwise noted adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address
  
- **Note: system provides private address spaces to each “process”**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

# Machine Words

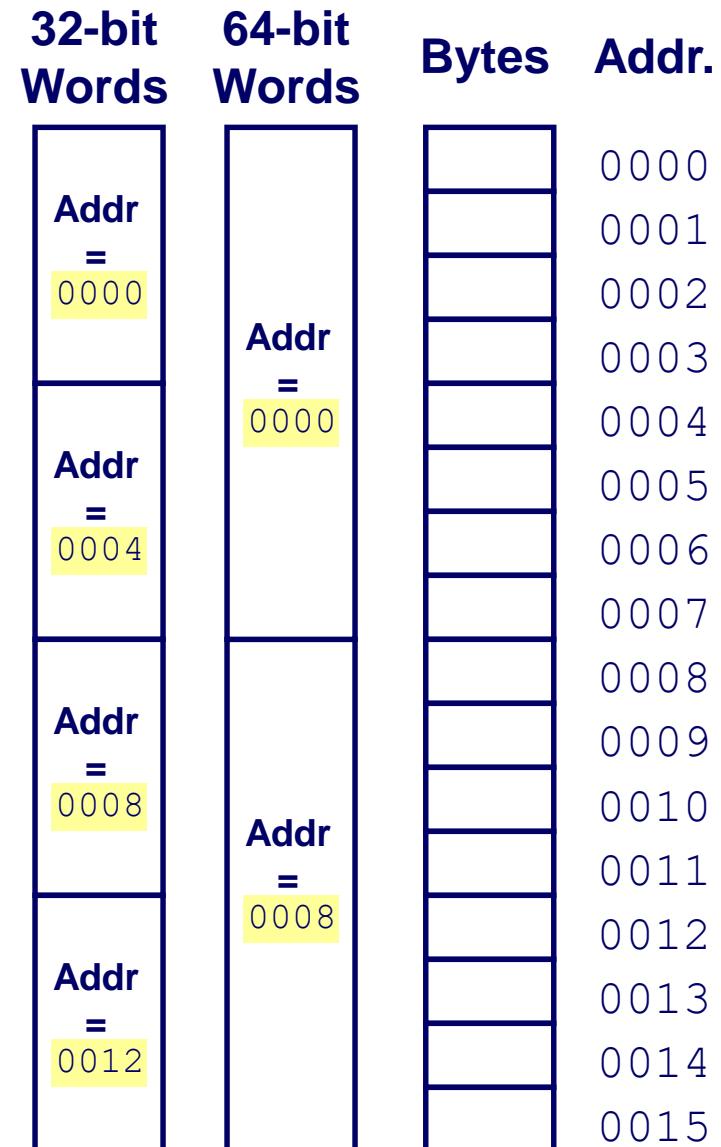
## ■ Any given computer has a “Word Size”

- Nominal size of integer-valued data
  - and of addresses (i.e. pointers)
- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB ( $2^{32}$  bytes)
- Increasingly, machines have 64-bit word size
  - Potentially, could have 18 PB (petabytes) of addressable memory
  - That's  $18.4 \times 10^{15}$
- Machines still support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

# Word-Oriented Memory Organization

## ■ Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



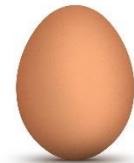
# Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<b>char</b>	1	1	1
<b>short</b>	2	2	2
<b>int</b>	4	4	4
<b>long</b>	4	8	8
<b>float</b>	4	4	4
<b>double</b>	8	8	8
<b>long double</b>	-	-	10/16
<b>pointer</b>	4	8	8

# Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

Little/Big Endian: Jonathan Swift's book "Gulliver's Travels" two kinds of religious groups (one prefer to eat their egg starting from the little end, others from the big end)

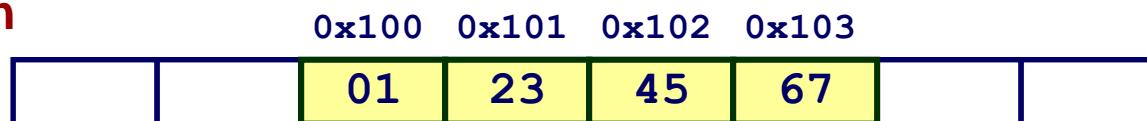


# Byte Ordering Example

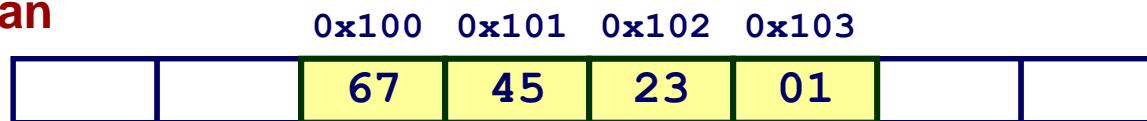
## ■ Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

### BigEndian



### LittleEndian



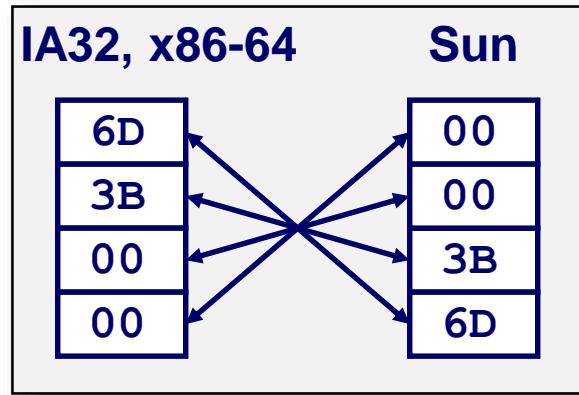
# Representing Integers

Decimal: 15213

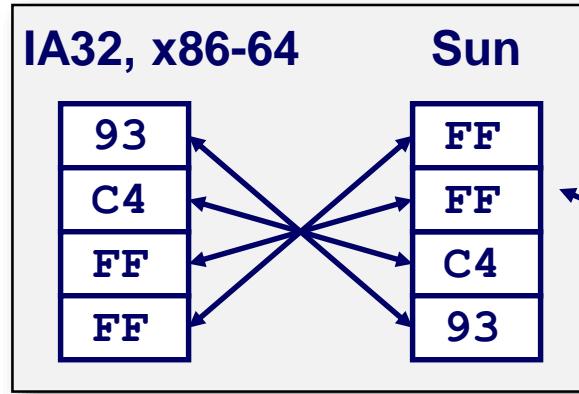
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

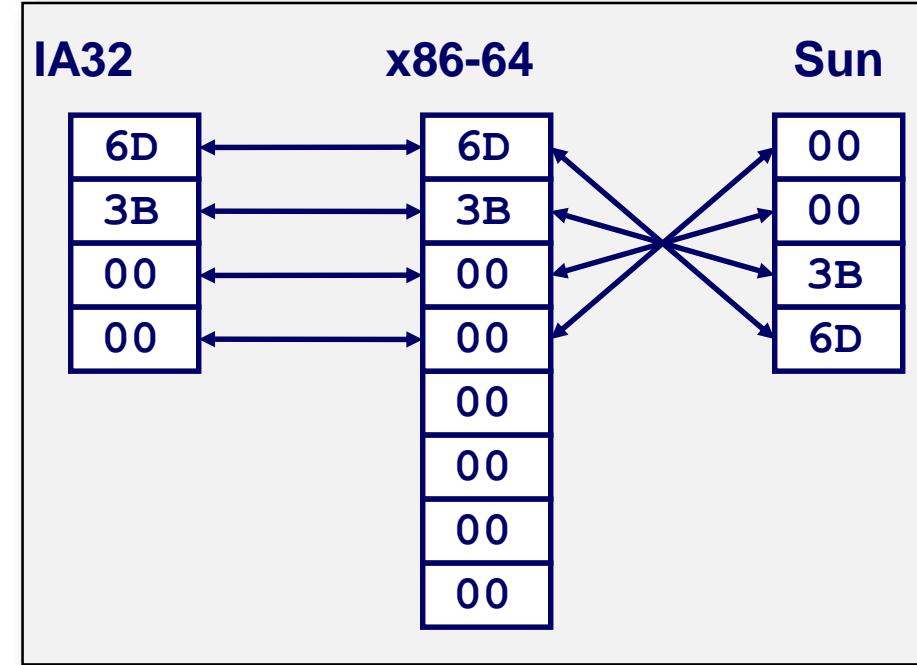
```
int A = 15213;
```



```
int B = -15213;
```



```
long int C = 15213;
```



Two's complement representation

# Examining Data Representations

## ■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char \* allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

### Printf directives:

%p: Print pointer  
%x: Print Hexadecimal

# **show\_bytes Execution Example**

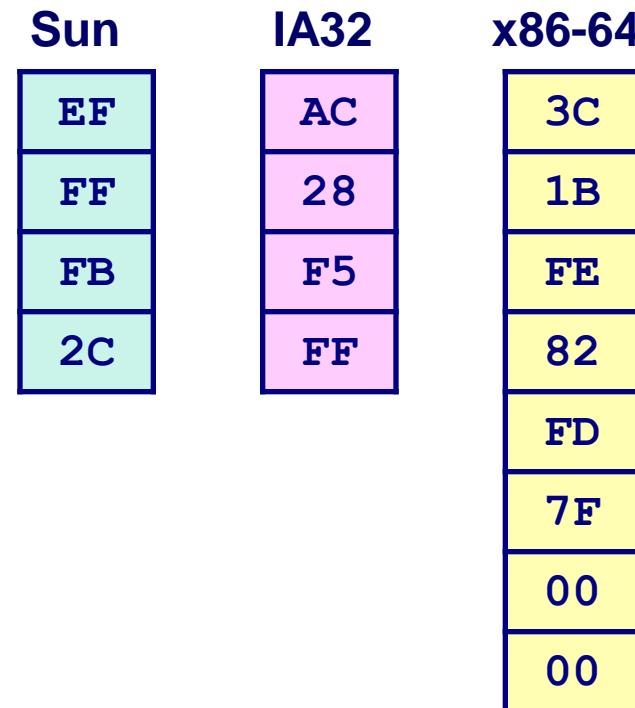
```
int a = 15213;  
printf("int a = 15213;\n");  
show_bytes((pointer) &a, sizeof(int));
```

## **Result (Linux x86-64):**

```
int a = 15213;  
0x7ffb7f71dbc      6d  
0x7ffb7f71dbd      3b  
0x7ffb7f71dbe      00  
0x7ffb7f71dbf      00
```

# Representing Pointers

```
int B = -15213;  
int *P = &B;
```



Different compilers & machines assign different locations to objects

Even get different results each time run program

# Representing Strings

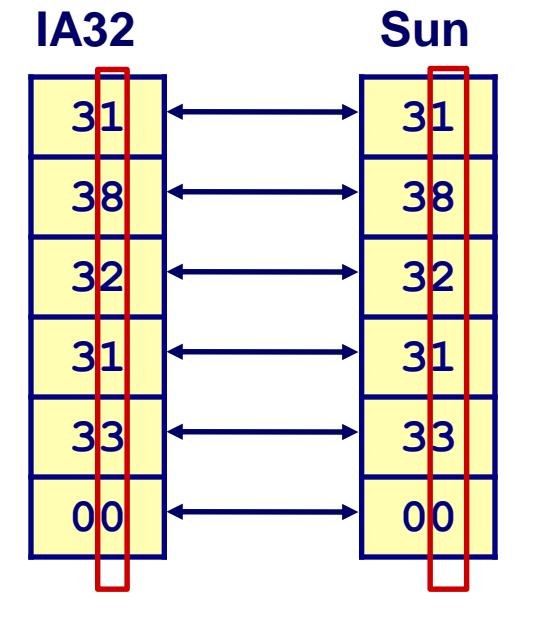
## ■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character "0" has code 0x30
    - Digit  $i$  has code  $0x30+i$
- String should be null-terminated
  - Final character = 0

## ■ Compatibility

- Byte ordering not an issue

```
char S[6] = "18213";
```



Thank you!

# Floating Point Arithmetic

Murat Manguoğlu

Sections 1-2

Patterson Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3<sup>rd</sup> Edition

# Floating-Point Addition

- Consider a 4-digit decimal example

- $9.999 \times 10^1 + 1.610 \times 10^{-1}$

- 1. Align decimal points

- Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$

- 2. Add significands

- $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$

- 3. Normalize result & check for over/underflow

- $1.0015 \times 10^2$

- 4. Round and renormalize if necessary

- $1.002 \times 10^2$

# Floating-Point Addition

## ■ Now consider a 4-digit binary example

- $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$  ( $0.5 + -0.4375$ )

## ■ 1. Align binary points

- Shift number with smaller exponent
- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

## ■ 2. Add significands

- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

## ■ 3. Normalize result & check for over/underflow

- $1.000_2 \times 2^{-4}$ , with no over/underflow

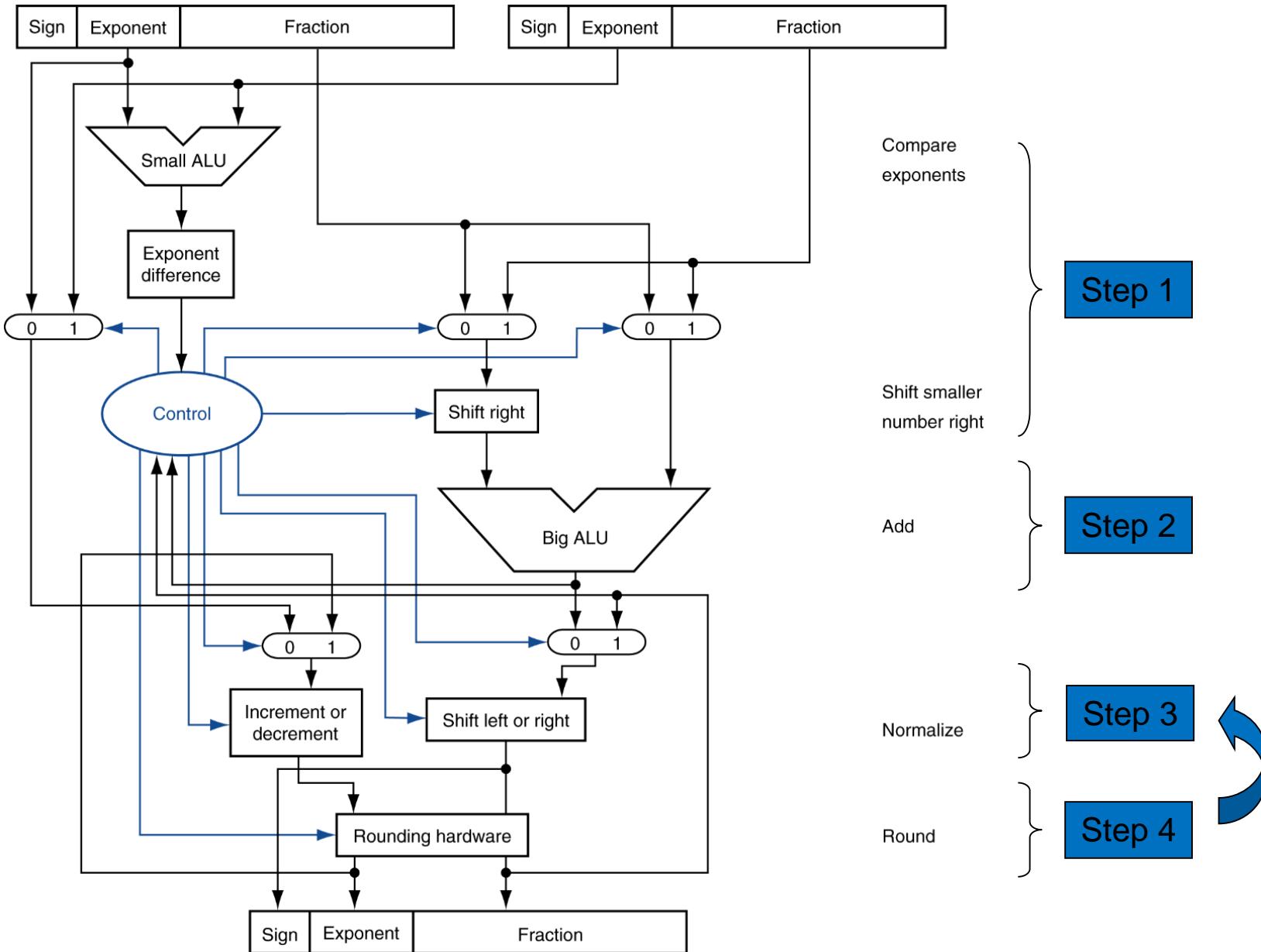
## ■ 4. Round and renormalize if necessary

- $1.000_2 \times 2^{-4}$  (no change) = 0.0625

# FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

# FP Adder Hardware



# Floating-Point Multiplication

- Consider a 4-digit decimal example

- $1.110 \times 10^{10} * 9.200 \times 10^{-5}$

- 1. Add exponents

- For biased exponents, subtract bias from sum
  - New exponent =  $10 + -5 = 5$

- 2. Multiply significands

- $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$

- 3. Normalize result & check for over/underflow

- $1.0212 \times 10^6$

- 4. Round and renormalize if necessary

- $1.021 \times 10^6$

- 5. Determine sign of result from signs of operands

- $+1.021 \times 10^6$

# Floating-Point Multiplication

## ■ Now consider a 4-digit binary example

- $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$  ( $0.5 \times -0.4375$ )

## ■ 1. Add exponents

- Unbiased:  $-1 + -2 = -3$
- Biased:  $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

## ■ 2. Multiply significands

- $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$

## ■ 3. Normalize result & check for over/underflow

- $1.110_2 \times 2^{-3}$  (no change) with no over/underflow

## ■ 4. Round and renormalize if necessary

- $1.110_2 \times 2^{-3}$  (no change)

## ■ 5. Determine sign: +ve $\times$ -ve $\Rightarrow$ -ve

- $-1.110_2 \times 2^{-3} = -0.21875$

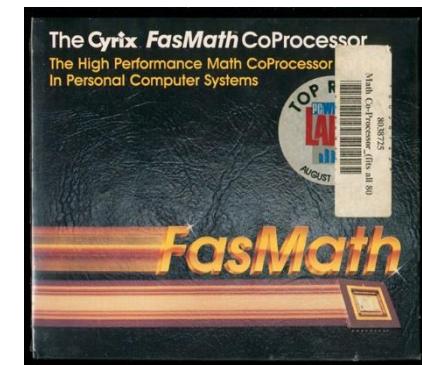
# FP Arithmetic Hardware

- **FP multiplier is of similar complexity to FP adder**
  - But uses a multiplier for significands instead of an adder
- **FP arithmetic hardware usually does**
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP  $\leftrightarrow$  integer conversion
- **Operations usually takes several cycles**
  - Can be pipelined

# x86 FP Architecture

## ■ Originally based on 8087 FP coprocessor (x87)

- 8 × 80-bit extended-precision registers
- Used as a push-down stack
- Registers indexed from TOS: ST(0), ST(1), ...



# x86 FP Architecture

- **FP values are 32-bit or 64 in memory**
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- **Not easy to generate and optimize code**
  - Result: poor FP performance
  - With XMM registers in X86-64 this is no longer the case

# Streaming SIMD Extension 2 (SSE2)

- Adds  $4 \times 128$ -bit registers

- Extended to 8 registers in AMD64/EM64T

- Can be used for multiple FP operands

- $2 \times 64$ -bit double precision
  - $4 \times 32$ -bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

CENG478 - Introduction to Parallel Computing

*Thank you!*

# **Machine-Level Programming I: Basics**

## **- History of Processors -**

**CENG331 - Computer Organization**

**Middle East Technical University**

**Instructors:**

**Murat Manguoglu      (Sections 1-2)**

**Fall 2020**

**Slides 6-33 are adapted from the slides of the textbook: D. A. Patterson and J. L. Hennessy,  
Computer Organization and Design: The Hardware/Software Interface, 3<sup>rd</sup> Edition**

**Others are adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>**

# Computer Architecture: A Little History

## Why worry about old ideas?

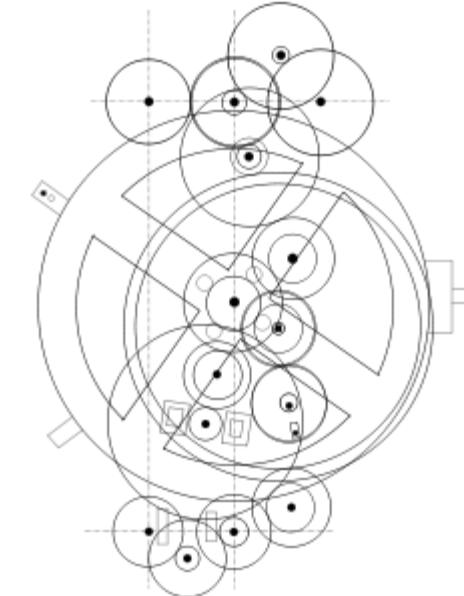
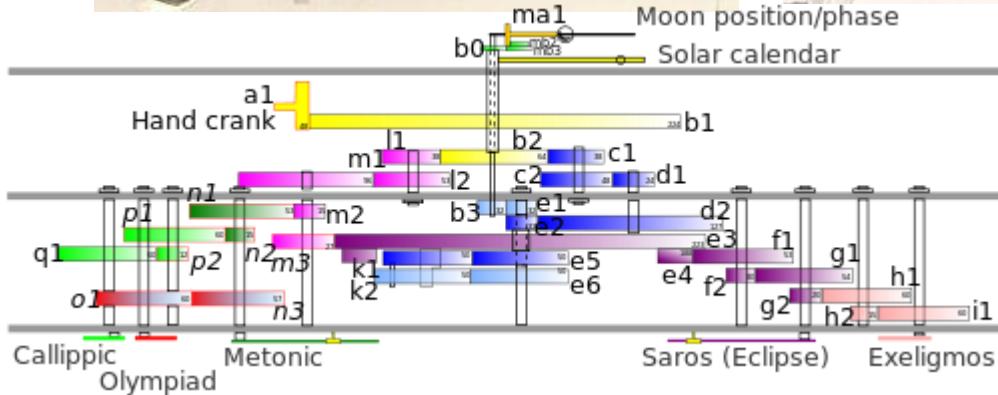
- *Die Geschichte der Wissenschaft ist die Wissenschaft selbst* (The history of science is science itself) – Johann Wolfgang von Goethe  
1749-1832
  - In fact, if you read any scientific paper/thesis/work, you will notice it starts with the review of the literature (i.e. History) of the earlier work
- Helps to illustrate the design process, and explains why certain decisions were taken
- Because future technologies might be as constrained as older ones
- **Those who ignore history are doomed to repeat it**
  - Every mistake made in mainframe design was also made in minicomputers, then microcomputers, where next?



# Antikythera Mechanism

## 150-100 BC

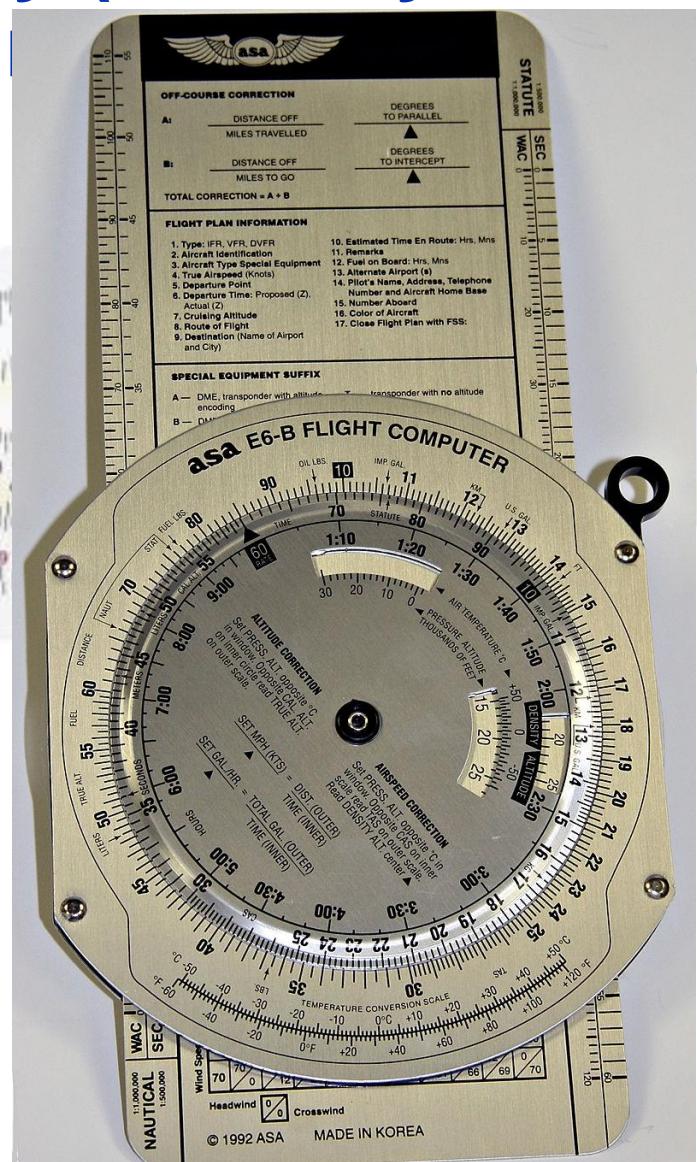
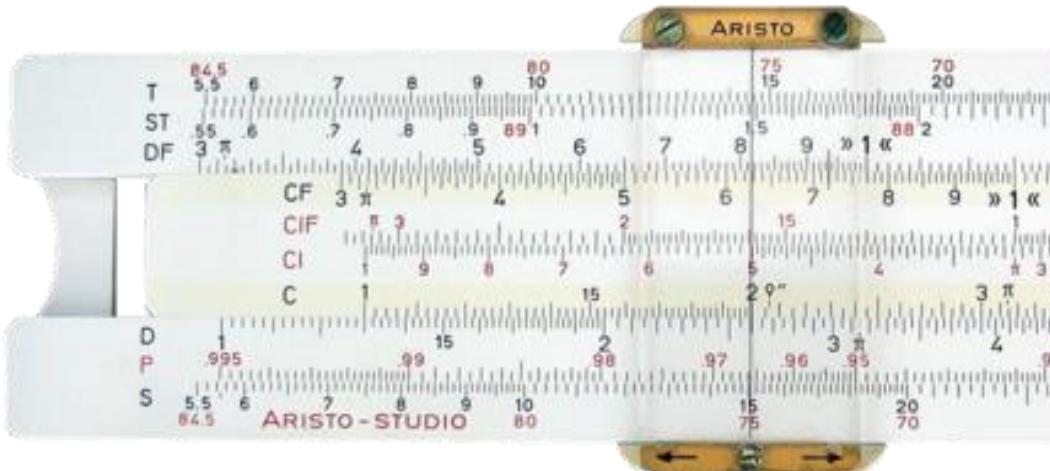
An astronomical calendar capable of tracking the position of the sun, moon, and planets; predict eclipses, and even recreate the irregular orbit of the moon



Source: [http://en.wikipedia.org/wiki/Antikythera\\_mechanism](http://en.wikipedia.org/wiki/Antikythera_mechanism)

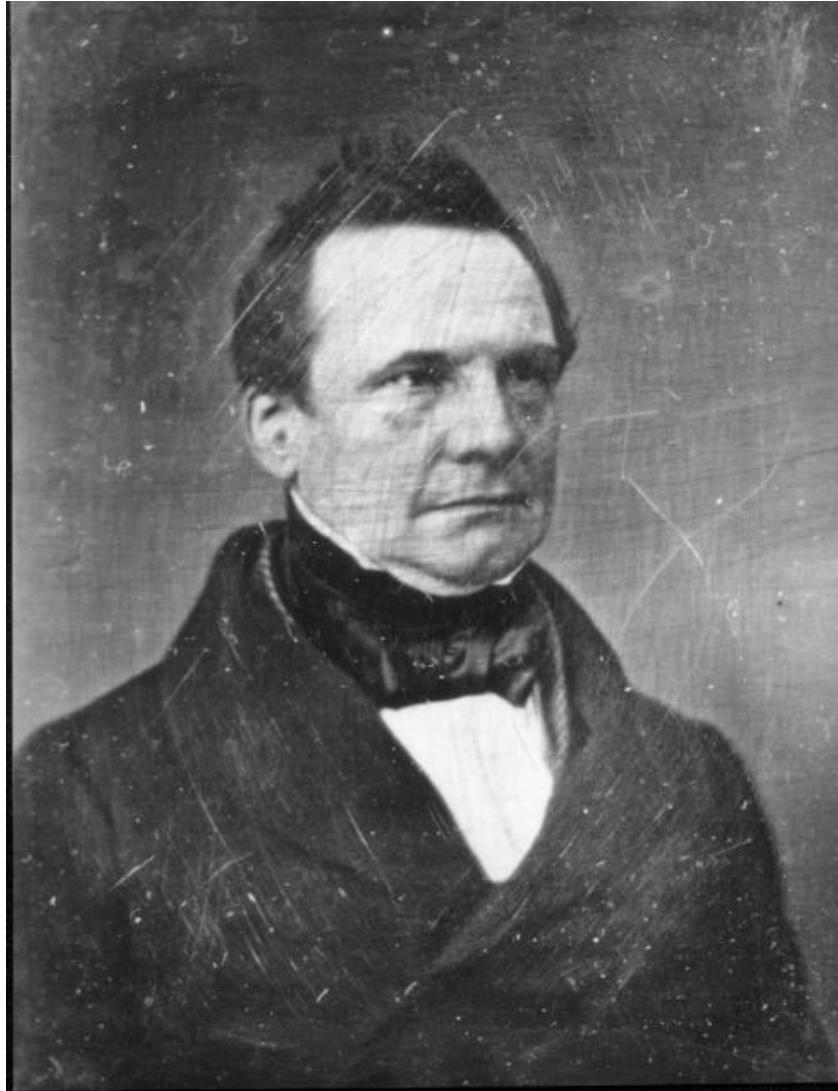
# Slide ruler

~1620 and still used today (but only as a backup computer on air)



# **Charles Babbage 1791-1871**

**Lucasian Professor of Mathematics,  
Cambridge University, 1827-1839**



# Charles Babbage

■ *Difference Engine*      1823

■ *Analytic Engine*      1833

- The forerunner of modern digital computer!

## *Application*

- Mathematical Tables – Astronomy
- Nautical Tables – Navy

## *Background*

- Any continuous function can be approximated by a polynomial --- Weierstrass

## *Technology*

- mechanical - gears, Jacquard's loom, simple calculators

# Difference Engine

A machine to compute mathematical tables

Weierstrass:

- Any continuous function can be approximated by a polynomial
- Any polynomial can be computed from *difference tables*



An example

$$f(n) = n^2 + n + 41$$

$$d_1(n) = f(n) - f(n-1) = 2n$$

$$d_2(n) = d_1(n) - d_1(n-1) = 2$$

$$f(n) = f(n-1) + d_1(n) = f(n-1) + (d_1(n-1) + 2)$$

Weierstraß

***all you need is an adder!***

n	0	1	2	3	4
d <sub>2</sub> (n)			2	2	2
d <sub>1</sub> (n)		2	4	6	8
f(n)	41	43	47	53	61

# Difference Engine

1823

- Babbage's paper is published

1834

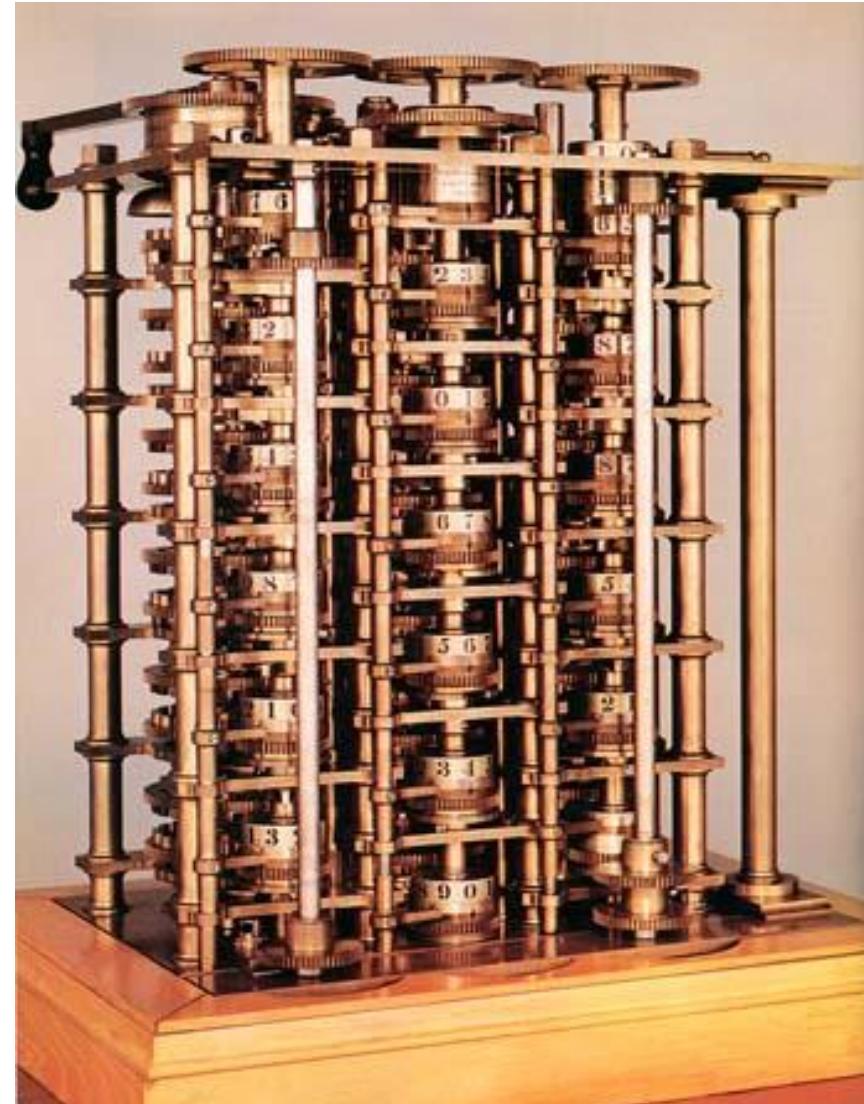
- The paper is read by Scheutz & his son in Sweden

1842

- Babbage gives up the idea of building it; he is onto Analytic Engine!

1855

- Scheutz displays his machine at the Paris World Fare
- Can compute any 6th degree polynomial
- *Speed:* 33 to 44 32-digit numbers per minute!



**Now the machine is at the Smithsonian**

# Analytic Engine

**1833: Babbage's paper was published**

- *conceived during a hiatus in the development of the difference engine*

**Inspiration: Jacquard Loom Machine**

- looms were controlled by punched cards
  - The set of cards with fixed punched holes defined the pattern of weave ⇒ *program*
  - The same set of cards could be used for different colored threads ⇒ *numbers*



**1871: Babbage dies**

- The machine remains unrealized.

***It is not clear if the analytic engine could be built using the mechanical technology of the time***

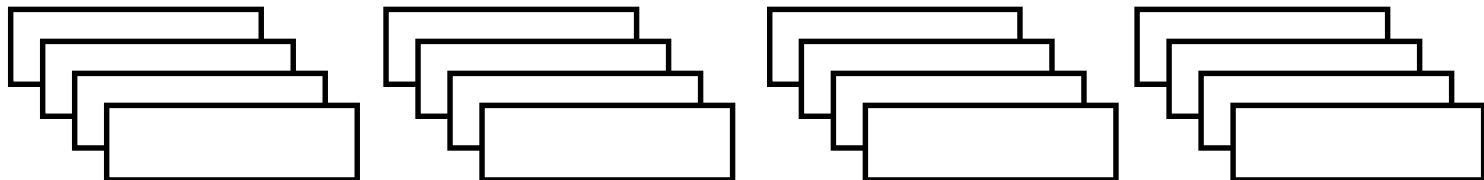
# Analytic Engine

The first conception of a general-purpose computer

1. The *store* in which all variables to be operated upon, as well as all those quantities which have arisen from the results of the operations are placed.
2. The *mill (arithmetic logic unit)* into which the quantities about to be operated upon are always brought.

*The program*  
Operation

variable1   variable2   variable3



An operation in the *mill* required feeding two punched cards and producing a new punched card for the *store*.

*An operation to alter the sequence was also provided!*

# The first programmer

Ada Byron *aka* “Lady Lovelace” 1815-52



**Ada's tutor was Babbage himself!**

# Babbage's Influence

- **Babbage's ideas had great influence later primarily because of**
  - *Luigi Menabrea*, who published notes of Babbage's lectures in Italy
  - *Lady Lovelace*, who translated Menabrea's notes in English and thoroughly expanded them.  
“... Analytic Engine weaves *algebraic patterns*....”
- **In the early twentieth century - the focus shifted to analog computers but**
  - *Harvard Mark I built in 1944 is very close in spirit to the Analytic Engine.*

# Linear Equation Solver

John Atanasoff, Iowa State University

## 1930's:

- Atanasoff built the Linear Equation Solver.
- It had 300 tubes!
- Special-purpose binary digital calculator
- Dynamic RAM (stored values on refreshed capacitors)



## *Application:*

- Linear and Integral differential equations

## *Background:*

- Vannevar Bush's Differential Analyzer  
--- *an analog computer*

## *Technology:*

- Tubes and Electromechanical relays

***Atanasoff decided that the correct mode of computation was using electronic binary digits.***

# Electronic analog computers



1960 Newmark analogue computer, made up of five units. This computer was used to solve differential equations



ELWAT , Poland, 1967



AKAT-1 , Poland, 1959

# Harvard Mark I

AIKEN - IBM AUTOMATIC SEQUENCE

## ■ Built in 1944 in IBM Endicott laboratories

- Howard Aiken – Professor of Physics at Harvard
- Essentially mechanical but had some electro-magnetically controlled relays and gears
- Weighed 5 tons and had 750,000 components
- A synchronizing clock that beat every 0.015 seconds (66Hz)

### Performance:

0.3 seconds for addition

6 seconds for multiplication

1 minute for a sine calculation

Decimal arithmetic

No Conditional Branch!

*Broke down once a week!*

# Electronic Numerical Integrator and Computer (ENIAC)

- Inspired by Atanasoff and Berry, Eckert and Mauchly designed and built ENIAC (1943-45) at the University of Pennsylvania
- The first, completely electronic, operational, general-purpose analytical calculator!
  - 30 tons, 72 square meters, 200KW
- Performance
  - Read in 120 cards per minute
  - Addition took 200  $\mu$ s, Division 6 ms
  - 1000 times faster than Mark I
- Not very reliable!

WW-2 Effort

***Application:*** Ballistic calculations

angle = f (location, tail wind, cross wind,  
air density, temperature, weight of shell,  
propellant charge, ... )



# Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC's programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions "out of order"
- Eckert, Mauchly, John von Neumann and others designed EDVAC (1944) to solve this problem
  - Solution was the *stored program computer*  
⇒ "*program can be manipulated as data*"
- *First Draft of a report on EDVAC* was published in 1945, but just had von Neumann's signature!
  - In 1973 the court of Minneapolis attributed the honor of *inventing the computer* to John Atanasoff

# Stored Program Computer

**Program = A sequence of instructions**

***How to control instruction sequencing?***

*manual control*

calculators

*automatic control*

*external (paper tape)*

Harvard Mark I , 1944

Zuse's Z1, WW2

*internal*

*plug board*

ENIAC 1946

*read-only memory*

ENIAC 1948

*read-write memory*

EDVAC 1947 (*concept*)

- The same storage can be used to store program and data

**EDSAC**

**1950**

**Maurice Wilkes**

# Technology Issues

**ENIAC**

⇒

**EDVAC**

**18,000 tubes**

**20 10-digit numbers**

**4,000 tubes**

**2000 word storage**

**mercury delay lines**

***ENIAC had many asynchronous parallel units  
but only one was active at a time***

**BINAC : Two processors that checked each other  
for reliability.**

***Didn't work well because processors never  
agreed***

```
manguoglu@desktop:~$ ./some_buggy_executable  
Segmentation fault (core dumped)
```

# Commercial Activity: 1948-52

## IBM's SSEC (follow on from Harvard Mark I)

*Selective Sequence Electronic Calculator*

- 150 word store.
- Instructions, constraints, and tables of data were read from paper tapes.
- 66 Tape reading stations!
- Tapes could be glued together to form a loop!
- Data could be output in one phase of computation and read in the next phase of computation.

# And then there was IBM 701



**IBM 701 -- 30 machines were sold in 1953-54  
used CRTs as main memory, 72 tubes of 32x32b  
each**

**IBM 650 -- a cheaper, drum based machine,  
more than 120 were sold in 1954  
and there were orders for 750 more!**

***Users stopped building their own machines.***

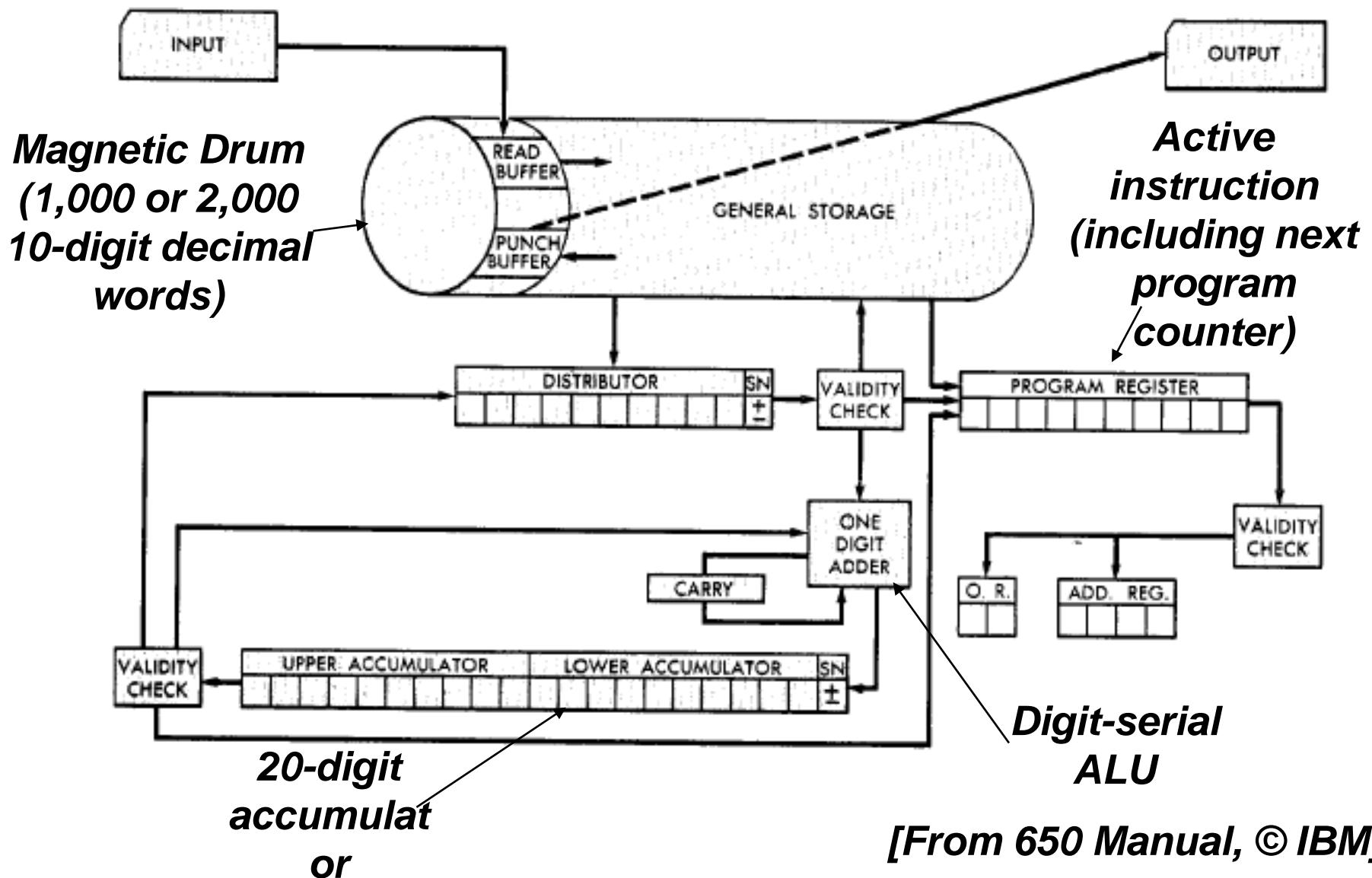
**Why was IBM late getting into  
computer technology?**

***IBM was making too much money!***  
**Even without computers, IBM  
revenues were doubling every 4 to 5  
years in 40's and 50's.**

# Computers in mid 50's

- **Hardware was expensive**
- **Stores were small (1000 words)**
  - ⇒ No resident system software!
- **Memory access time was 10 to 50 times slower than the processor cycle**
  - ⇒ Instruction execution time was totally dominated by the *memory reference time*.
- **The ability to design complex control circuits to execute an instruction was the central design concern as opposed to the speed of decoding or an ALU operation**
- **Programmer's view of the machine was inseparable from the actual hardware implementation**

# The IBM 650 (1953-4)



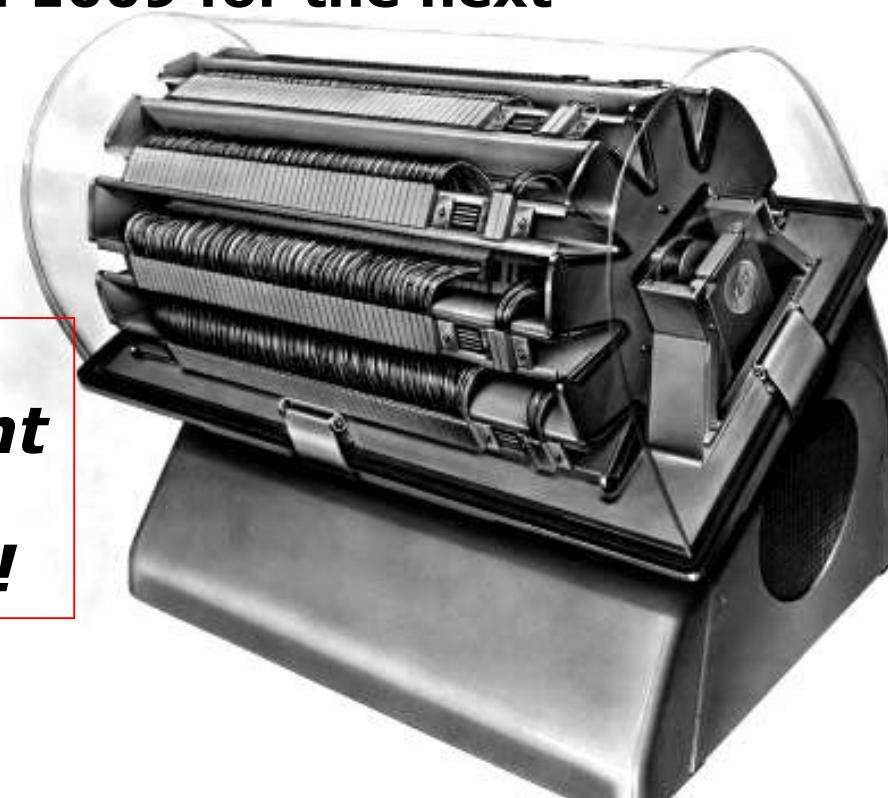
# Programmer's view of the IBM 650

## A drum machine with 44 instructions

**Instruction:** 60 1234 1009

- “Load the contents of location 1234 into the *distribution*; put it also into the *upper accumulator*; set *lower accumulator* to zero; and then go to location 1009 for the next instruction.”

***Good programmers optimized the placement of instructions on the drum to reduce latency!***



# Evolution of Addressing Modes

## 1. Single accumulator, absolute address

LOAD x

## 2. Single accumulator, index registers

LOAD x, IX

## 3. Indirection

LOAD (x)

## 4. Multiple accumulators, index registers, indirection

LOAD R, IX, x

or      LOAD R, IX, (x) the meaning?

$R \leftarrow M[M[x] + (IX)]$

or  $R \leftarrow M[M[x + (IX)]]$

## 5. Indirect through registers

LOAD R<sub>I</sub>, (R<sub>J</sub>)

## 6. The works

LOAD R<sub>I</sub>, R<sub>J</sub>, (R<sub>K</sub>)

R<sub>J</sub> = index, R<sub>K</sub> = base addr

# Variety of Instruction Formats

- ***One address formats: Accumulator machines***
  - Accumulator is always other source and destination operand
- ***Two address formats: the destination is same as one of the operand sources***

(Reg  $\times$  Reg) to Reg

$$R_I \leftarrow (R_I) + (R_J)$$

(Reg  $\times$  Mem) to Reg

$$R_I \leftarrow (R_I) + M[x]$$

- $x$  can be specified directly or via a register
- effective address calculation for  $x$  could include indexing, indirection, ...

- ***Three address formats: One destination and up to two operand sources per instruction***

(Reg  $\times$  Reg) to Reg

$$R_I \leftarrow (R_J) + (R_K)$$

(Reg  $\times$  Mem) to Reg

$$R_I \leftarrow (R_J) + M[x]$$

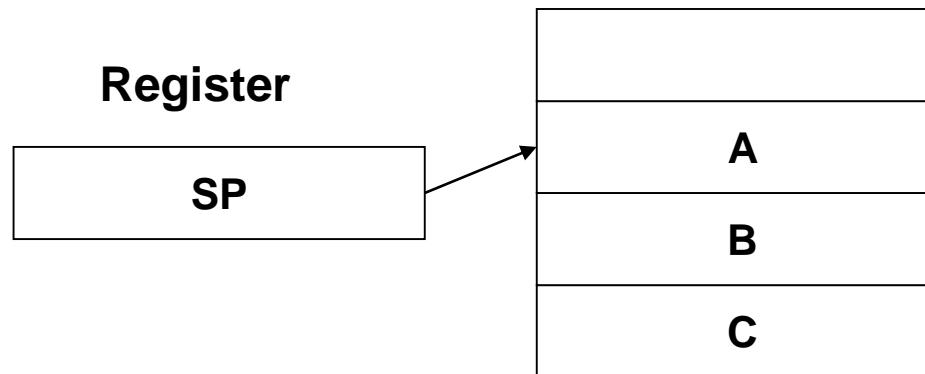
# Zero Address Formats

## ■ Operands on a stack

add       $M[sp-1] \leftarrow M[sp] + M[sp-1]$

load       $M[sp] \leftarrow M[M[sp]]$

- Stack can be in registers or in memory (usually top of stack cached in registers)



- Example: Burrough Corporation B5000

# Stack Machines (Mostly) Died by 1980

1. Stack programs are not smaller if short (Register) addresses are permitted.
2. Modern compilers can manage fast register space better than the stack discipline.

**GPR's and caches are better than stacks**

*Early language-directed architectures often did not take into account the role of compilers!*

B5000, B6700, HP 3000, ICL 2900, Symbolics 3600

*Some would claim that an echo of this mistake is visible in the SPARC architecture register windows*

# Stacks post-1980

- **Inmos Transputers (1985-2000)**
  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world's fastest microprocessor in late 80's
- **Forth machines**
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)
- **Java Virtual Machine**
  - Designed for software emulation, not direct hardware execution
  - Sun PicoJava implementation + others
- **Intel x87 floating-point unit**
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4, replaced with SSE2 FP registers

# Software Developments

## Libraries of numerical routines

- Floating point operations
- Transcendental functions
- Matrix manipulation,  
equation solvers, . . .

**Machines required experienced operators**

⇒ Most users could not be expected to understand these programs, much less write them

⇒ Machines had to be sold with a lot of resident software

1955

*High level Languages - Fortran 1956  
Operating Systems -*

- Assemblers, Loaders, Linkers, Compilers
- Accounting programs to keep track of usage and charges

# Compatibility Problem at IBM

**By early 60's, IBM had 4 incompatible lines of computers!**

<b>701</b>	$\rightarrow$	<b>7094</b>
<b>650</b>	$\rightarrow$	<b>7074</b>
<b>702</b>	$\rightarrow$	<b>7080</b>
<b>1401</b>	$\rightarrow$	<b>7010</b>

**Each system had its own**

- **Instruction set**
- **I/O system and Secondary Storage:**  
**magnetic tapes, drums and disks**
- **assemblers, compilers, libraries,...**
- **market niche**  
**business, scientific, real time, ...**

$\Rightarrow$  **IBM 360**

# IBM 360: A General-Purpose Register (GPR) Machine

## ■ Processor State

- 16 General-Purpose 32-bit Registers
  - *may be used as index and base register*
  - *Register 0 has some special properties*
- 4 Floating Point 64-bit Registers
- A Program Status Word (PSW)
  - *PC, Condition codes, Control flags*

## ■ A 32-bit machine with 24-bit addresses

- But no instruction contains a 24-bit address!

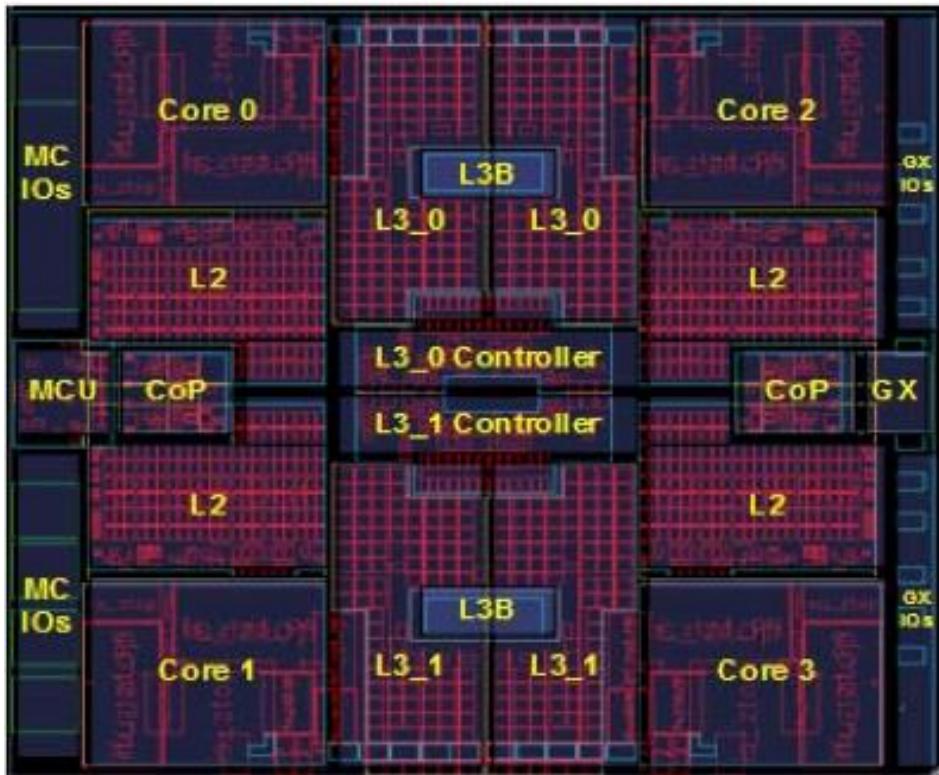
## ■ Data Formats

- 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

*The IBM 360 is why bytes are 8-bits long today!*

# IBM 360: 47 years later...

## The zSeries z11 Microprocessor



[ IBM, HotChips, 2010 ]

- 5.2 GHz in IBM 45nm PD-SOI CMOS technology
- 1.4 billion transistors in 512 mm<sup>2</sup>
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- Quad-core design
- Three-issue out-of-order superscalar pipeline
- Out-of-order memory accesses
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache on-chip
- 1.5MB private L2 unified cache per core, on-chip
- On-Chip 24MB eDRAM L3 cache
- Scales to 96-core multiprocessor with 768MB of shared L4 eDRAM

# 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
- **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 – Examples: IBM Power, Arm, ... )
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

# x86 Clones: Advanced Micro Devices (AMD)

## ■ Historically

- AMD has followed Intel, but sometimes Intel followed AMD too.
- 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
  - Leads the world in semiconductor technology
- AMD has fallen behind
  - Relies on external semiconductor manufacturer

# Intel and AMD's 64-Bit History

## ■ 2001: Intel Attempts Radical Shift from IA32 to IA64

- Totally different architecture (Itanium)
- 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, IA64 is actually not that bad, latest one was produced in 2017 (Itanium 9760 – 8 cores 2.66Ghz 32MB Cache)~~

## ■ 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

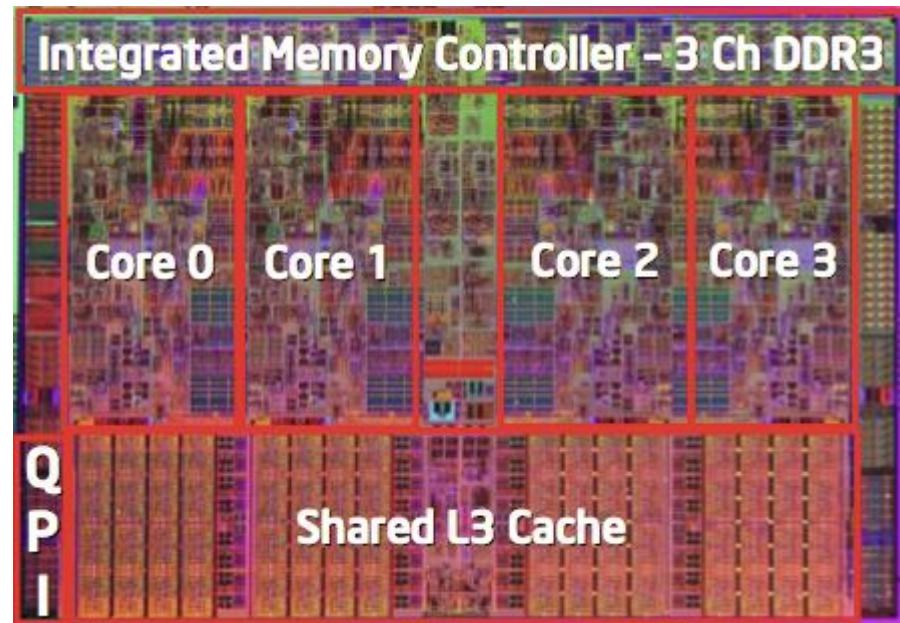
# x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ <b>8086</b>	<b>1978</b>	<b>29K</b>	<b>5-10</b>
			<ul style="list-style-type: none"><li>▪ First 16-bit Intel processor. Basis for IBM PC &amp; DOS</li><li>▪ 1MB address space</li></ul>
■ <b>386</b>	<b>1985</b>	<b>275K</b>	<b>16-33</b>
			<ul style="list-style-type: none"><li>▪ First 32 bit Intel processor , referred to as IA32</li><li>▪ Added “flat addressing”, capable of running Unix</li></ul>
■ <b>Pentium 4E</b>	<b>2004</b>	<b>125M</b>	<b>2800-3800</b>
			<ul style="list-style-type: none"><li>▪ First 64-bit Intel x86 processor, referred to as x86-64</li></ul>
■ <b>Core 2</b>	<b>2006</b>	<b>291M</b>	<b>1060-3500</b>
			<ul style="list-style-type: none"><li>▪ First multi-core Intel processor</li></ul>
■ <b>Core i7</b>	<b>2008</b>	<b>731M</b>	<b>1700-3900</b>
			<ul style="list-style-type: none"><li>▪ Four cores</li></ul>

# x86 Processors, cont.

## ■ Machine Evolution

▪ 386	1985	0.3M
▪ Pentium	1993	3.1M
▪ Pentium/MMX	1997	4.5M
▪ PentiumPro	1995	6.5M
▪ Pentium III	1999	8.2M
▪ Pentium 4	2001	42M
▪ Core 2 Duo	2006	291M
▪ Core i7	2008	731M
▪ Core i7 Skylake	2015	1.9B
▪ AMD epyc	2017	19.2B



## ■ Added Features

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

# Intel – Desktop state-of-the-art

## ■ Core i9-9980XE

- 18 cores
- 36 threads
- 3.0-4.4 Ghz
- 24.75 MB Cache

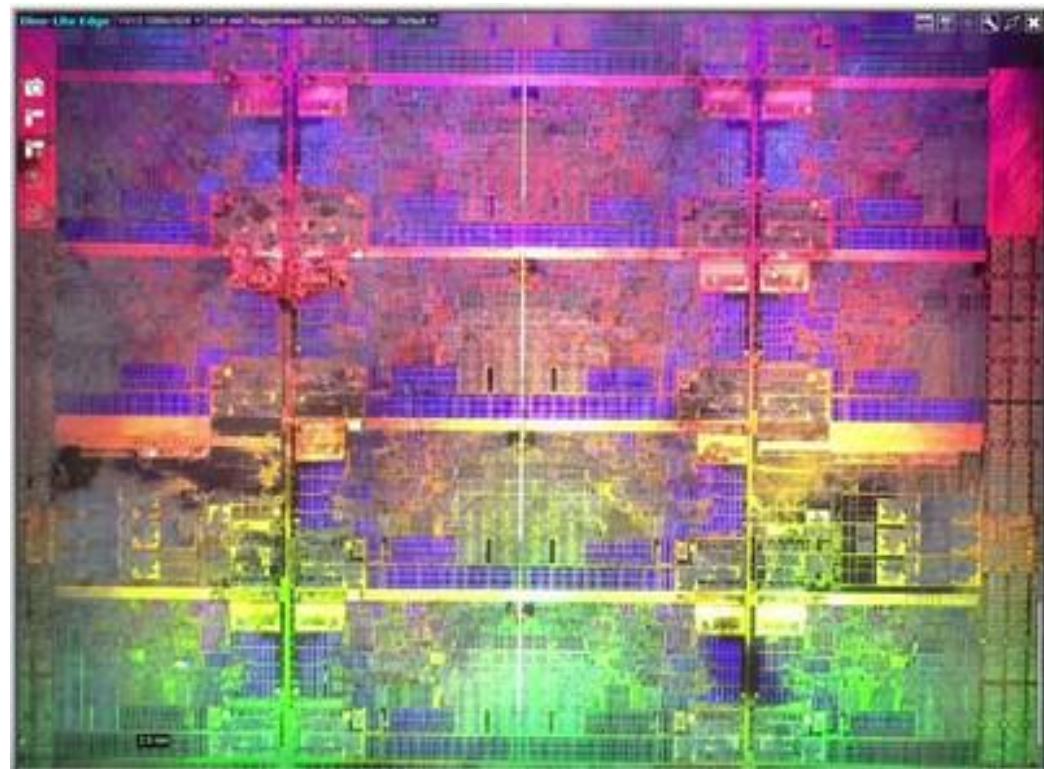


Image source: [https://hwbot.org/newsflash/5002\\_video\\_intel\\_core\\_i9\\_7980xe\\_die\\_extraction\\_with\\_de8auer\\_\(part\\_2\)](https://hwbot.org/newsflash/5002_video_intel_core_i9_7980xe_die_extraction_with_de8auer_(part_2))

# AMD –Server state-of-the-art

## ■ AMD epyc 7742

- 64 cores
- 128 threads
- 256 MB Cache
- 2.25-3.4 GHz

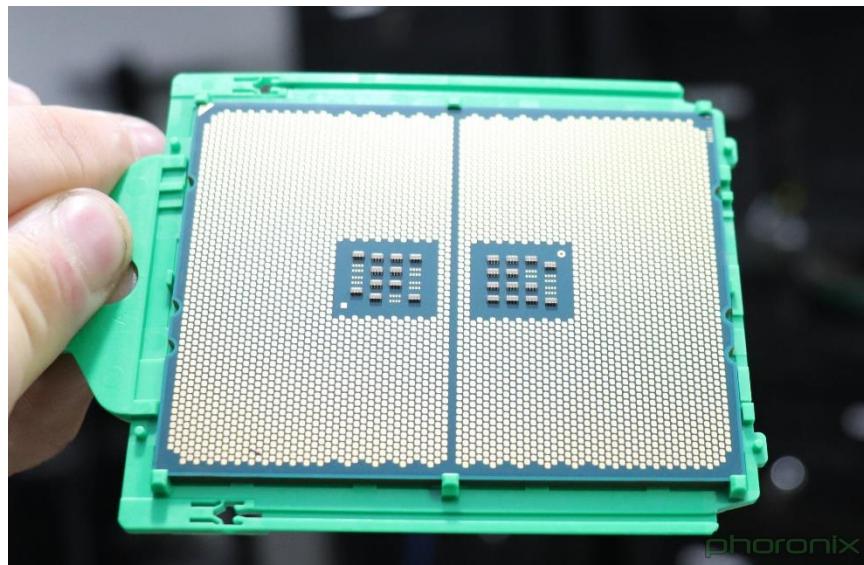


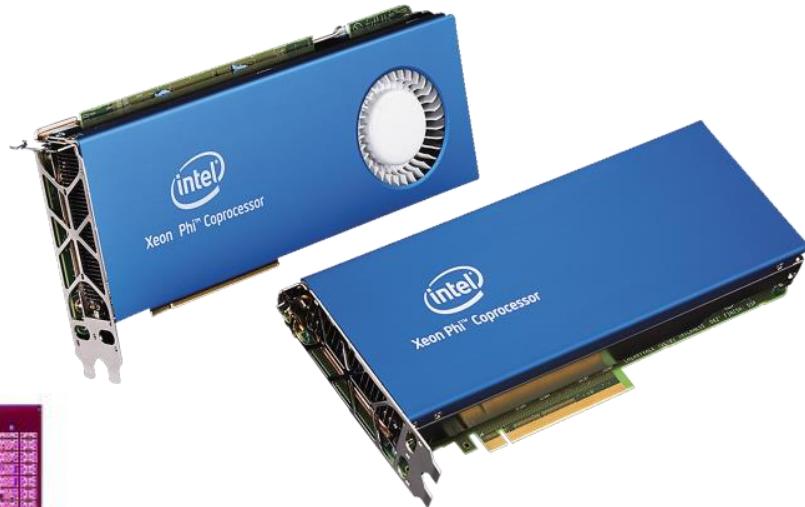
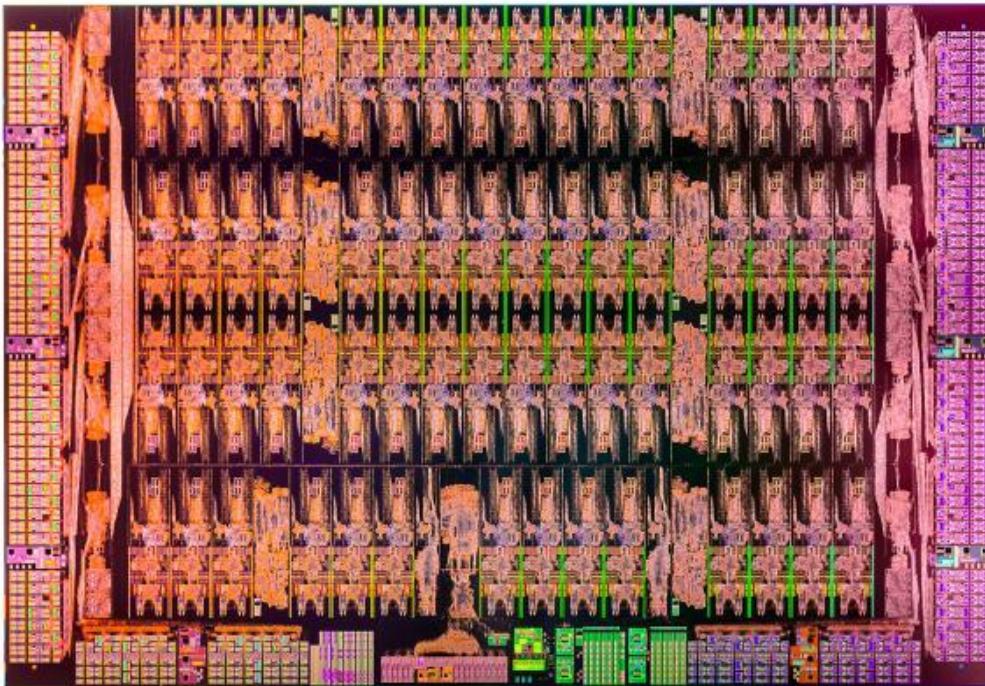
Image source: [https://www.phoronix.com/image-viewer.php?id=amd-epyc-7502-7742&image=amd\\_rome\\_4\\_lrg](https://www.phoronix.com/image-viewer.php?id=amd-epyc-7502-7742&image=amd_rome_4_lrg)

# Manycore (vs Multicore) processors

<http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>

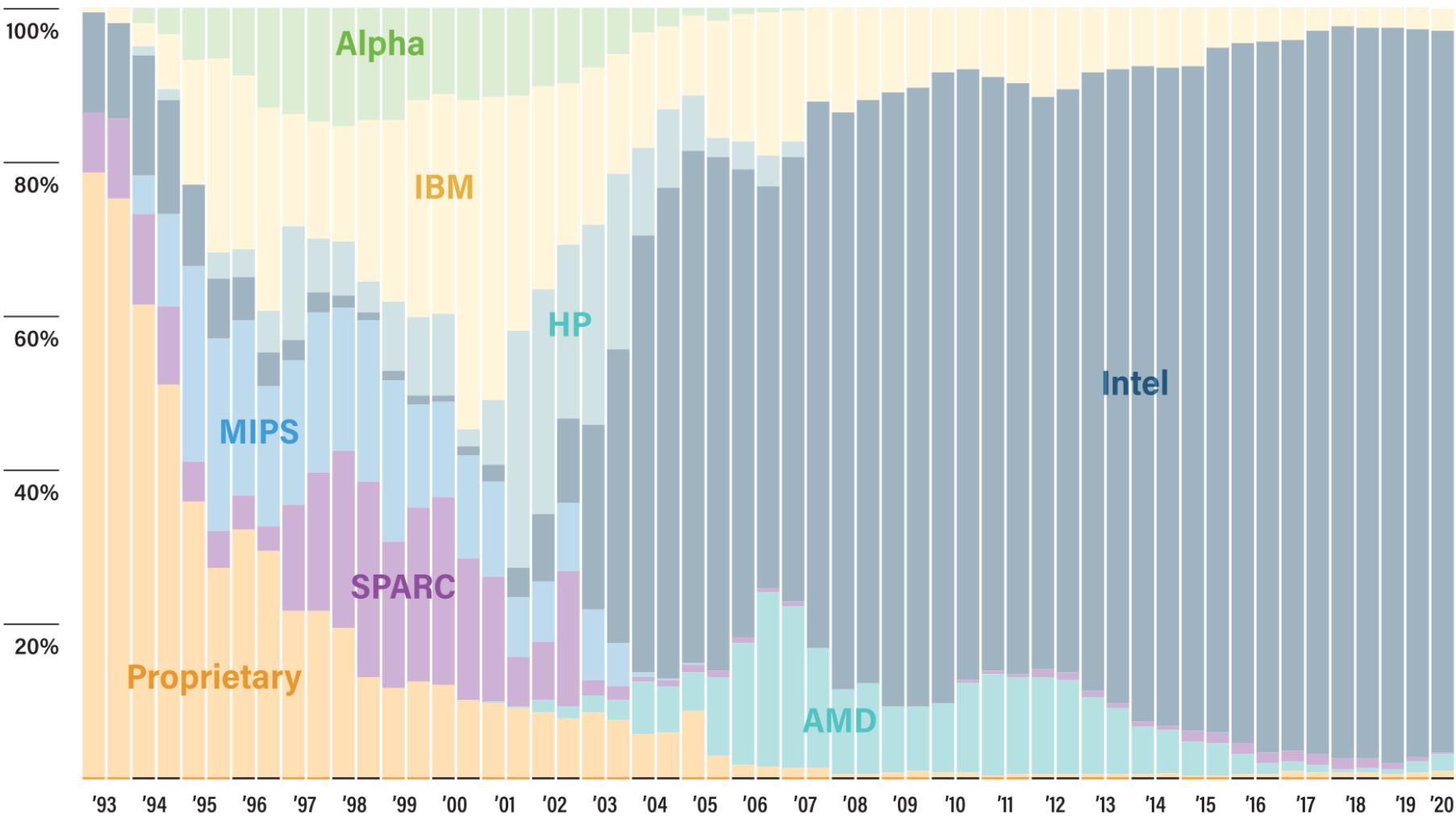
Upto 72 cores

(XeonPhi co-processors are no longer produced)



# Most Powerful Computers - which processor ?

## Chip Technology



source: [www.top500.org](http://www.top500.org)

NATIONAL BESTSELLER

# INSIDE INTEL

UPDATED  
EDITION

"A TRULY  
FASCINATING  
READ...the first  
unauthorized  
history of this  
highly secretive  
company."  
—BARRON'S

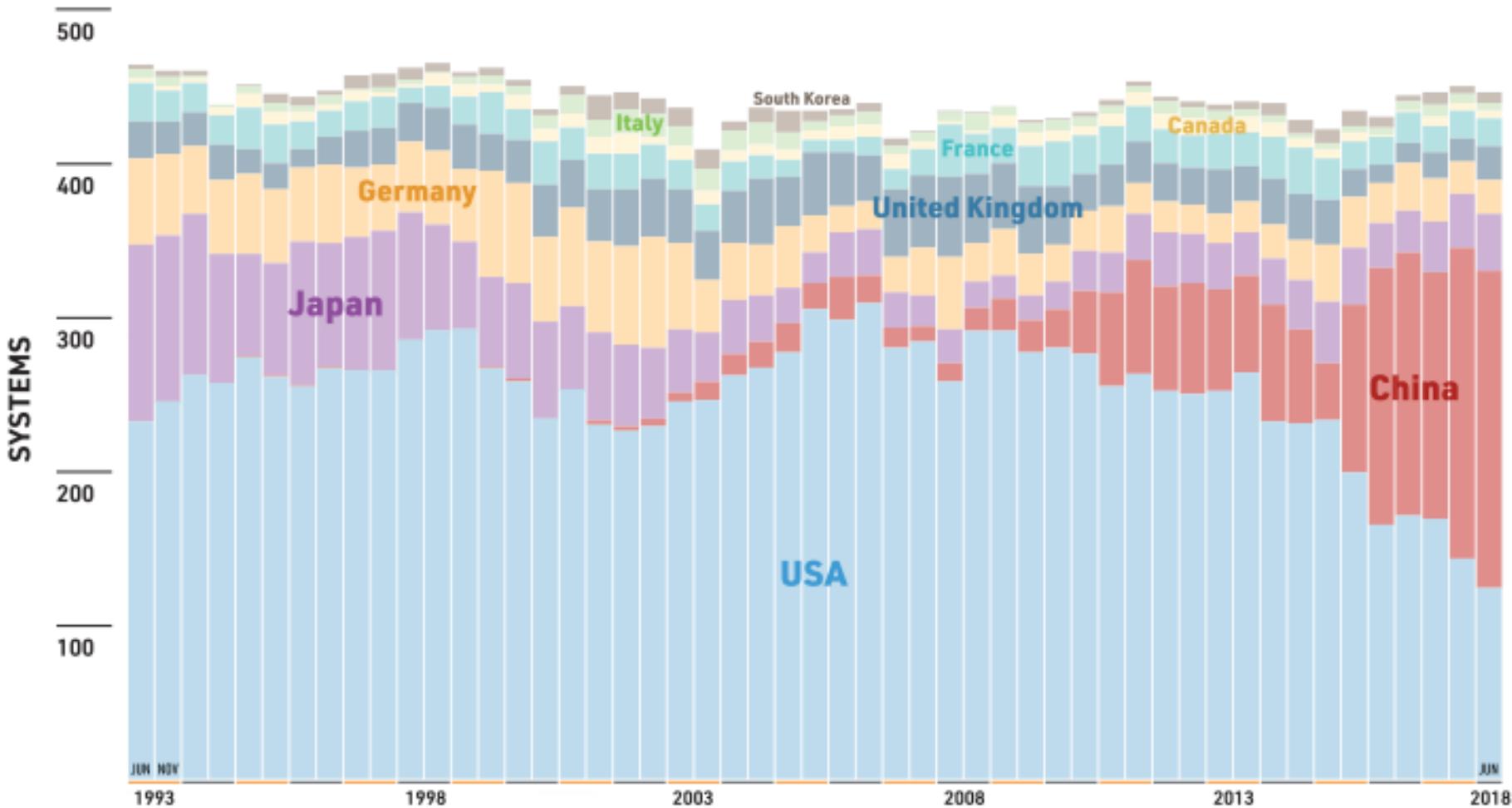


Andy Grove  
and the Rise of the World's  
Most Powerful Chip Company

TIM JACKSON

# Most Powerful Computers – where?

## COUNTRIES



source: [www.top500.org](http://www.top500.org)

# Our Coverage

- IA32
  - The traditional x86
- x86-64
  - The standard
  - \$gcc hello.c
  - \$gcc -m64 hello.c
- Presentation
  - 3<sup>rd</sup> edition covers x86-64
  - 2<sup>nd</sup> International Edition covers IA32 + x86-64
  - 2<sup>nd</sup> edition covers only IA32
  - We will only cover x86-64

*Thank you!*

# **Machine-Level Programming I: Basics**

## **- C/Assembly and Machine Code-**

**CENG331 - Computer Organization**

**Middle East Technical University**

**Instructors:**

**Murat Manguoglu      (Sections 1-2)**

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

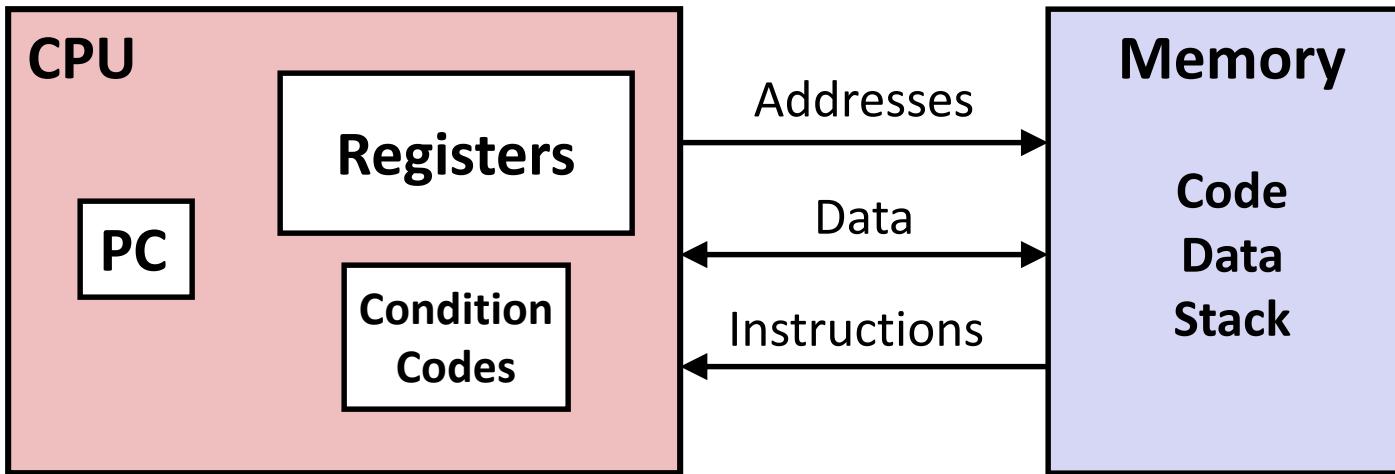
# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

# Definitions

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

# Assembly/Machine Code View

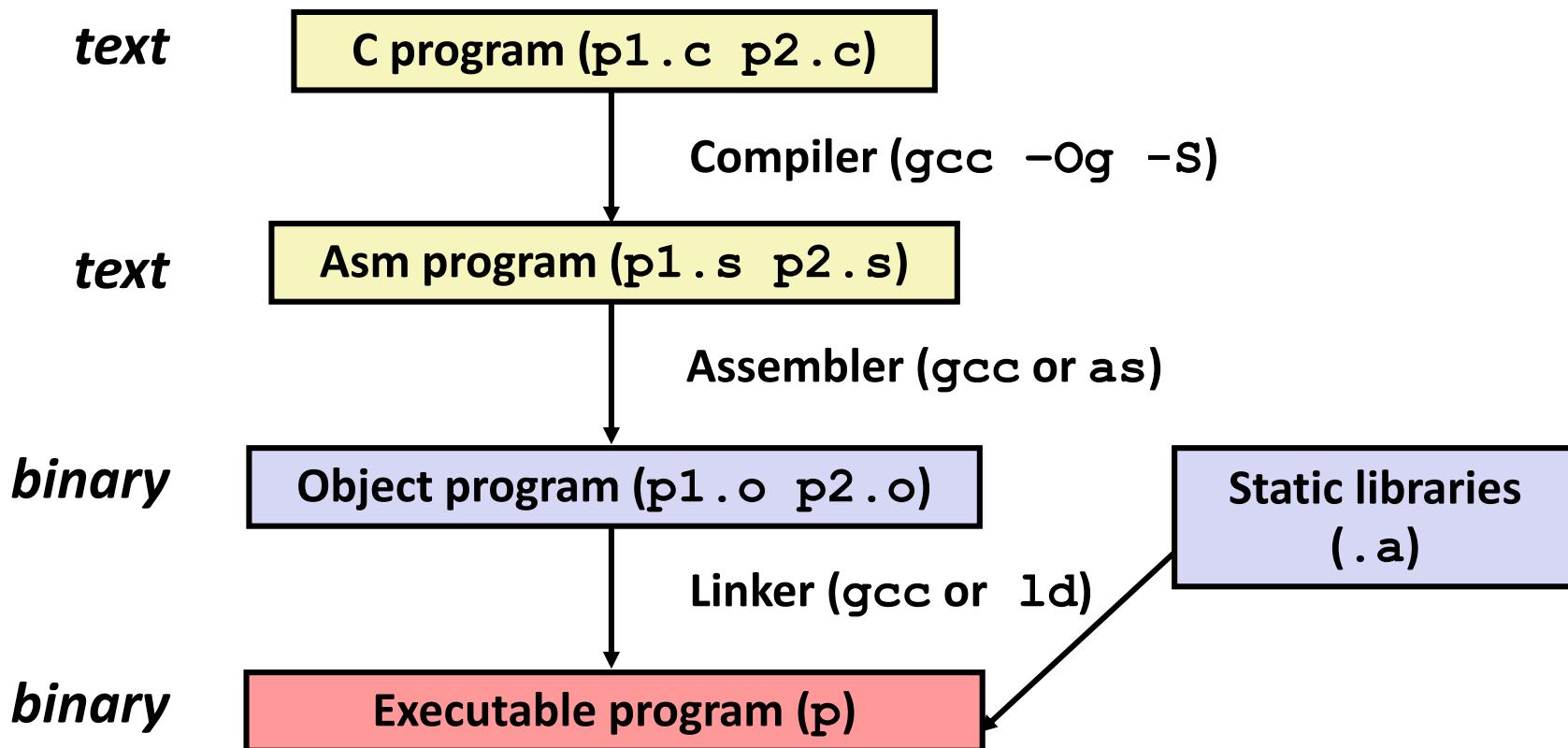


## Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called the instruction pointer register “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

# 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 (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

## Generated x86-64 Assembly

```
sumstore:
    pushq   %rbx
    movq   %rdx, %rbx
    call    plus
    movq   %rax, (%rbx)
    popq   %rbx
    ret
```

Obtain with command

```
gcc -O1 -S sum.c
```

```
gcc -O1 -Wa,-aslh -c sum.c > sum.s
```

Produces file sum.s

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

# 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
- Code: Byte sequences encoding series of instructions
- 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 sumstore

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

- Total of 14 bytes

0x48

- Each instruction  
1, 3, or 5 bytes

0x89

0x03

0x5b

- Starts at address

0xc3

0x0400595

### ■ 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

```
*dest = t;
```

## ■ C Code

- Store value **t** where designated by **dest**

```
movq %rax, (%rbx)
```

## ■ Assembly

- Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:
  - t:** Register **%rax**
  - dest:** Register **%rbx**
  - \*dest:** Memory **M[%rbx]**

```
0x40059e: 48 89 03
```

## ■ Object Code

- 3-byte instruction
- Stored at address **0x40059e**

# Disassembling Object Code

## Disassembled

```
000000000400595 <sumstore>:  
400595: 53          push    %rbx  
400596: 48 89 d3    mov     %rdx,%rbx  
400599: e8 f2 ff ff ff  callq   400590 <plus>  
40059e: 48 89 03    mov     %rax,(%rbx)  
4005a1: 5b          pop     %rbx  
4005a2: c3          retq
```

## ■ Disassembler

`objdump -d sum`

- 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

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

## Disassembled

```
Dump of assembler code for function sumstore:  
0x0000000000400595 <+0>: push    %rbx  
0x0000000000400596 <+1>: mov     %rdx,%rbx  
0x0000000000400599 <+4>: callq   0x400590 <plus>  
0x000000000040059e <+9>: mov     %rax,(%rbx)  
0x00000000004005a1 <+12>:pop    %rbx  
0x00000000004005a2 <+13>:retq
```

### ■ Within gdb Debugger

gdb sum

disassemble sumstore

- Disassemble procedure

x/14xb sumstore

- Examine the 14 bytes starting at sumstore

# 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:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by  
Microsoft End User License Agreement

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

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations

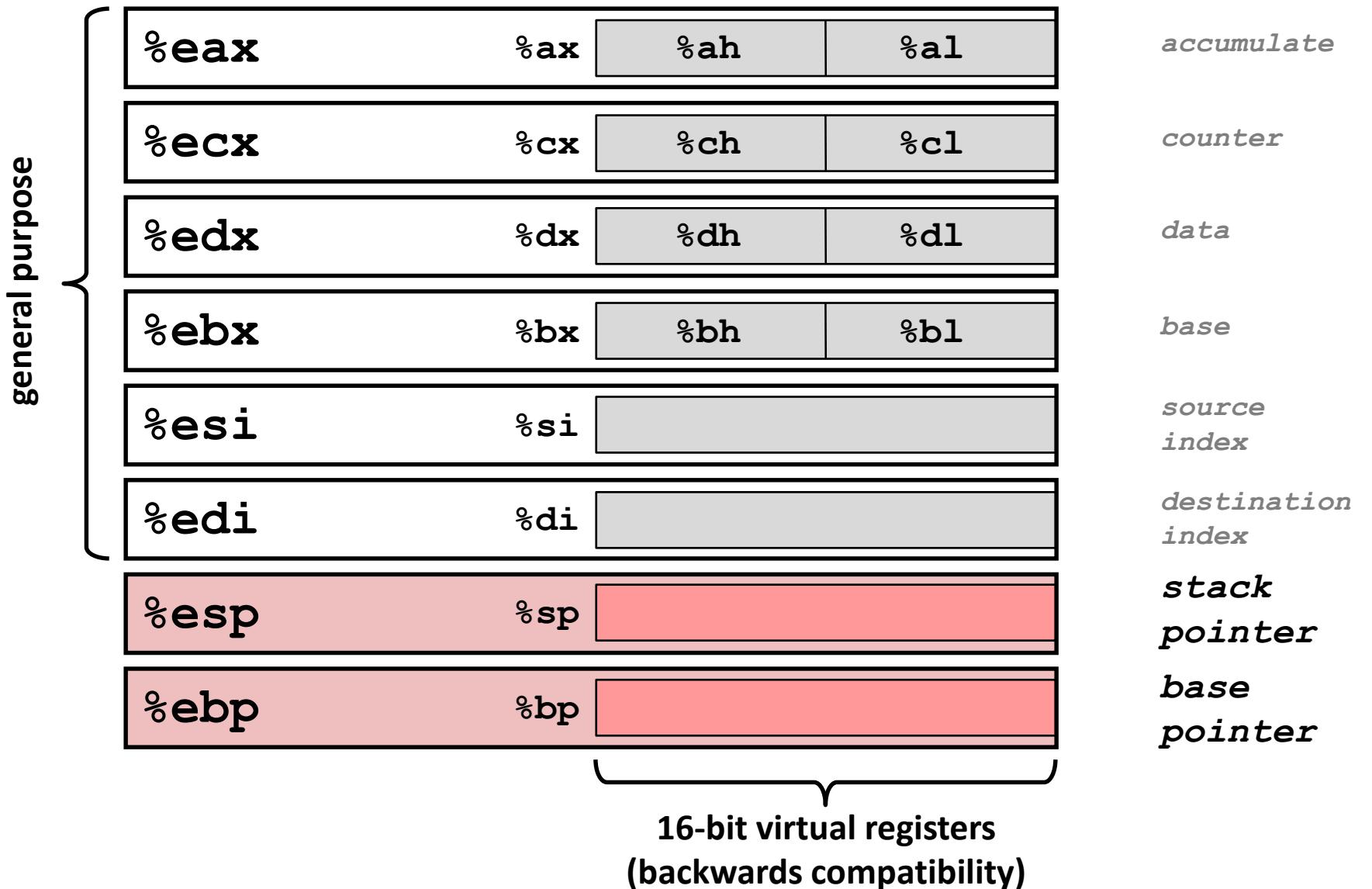
# x86-64 Integer Registers

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

# Some History: IA32 Registers



# 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 “address modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

# movl Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	<i>Imm</i>	<i>Reg</i>	movq \$0x4,%rax	temp = 0x4;
		<i>Mem</i>	movq \$-147,(%rax)	*p = -147;
	<i>Reg</i>	<i>Reg</i>	movq %rax,%rdx	temp2 = temp1;
	<i>Mem</i>	<i>Reg</i>	movq (%rax),%rdx	*p = temp;

*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

```
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
```

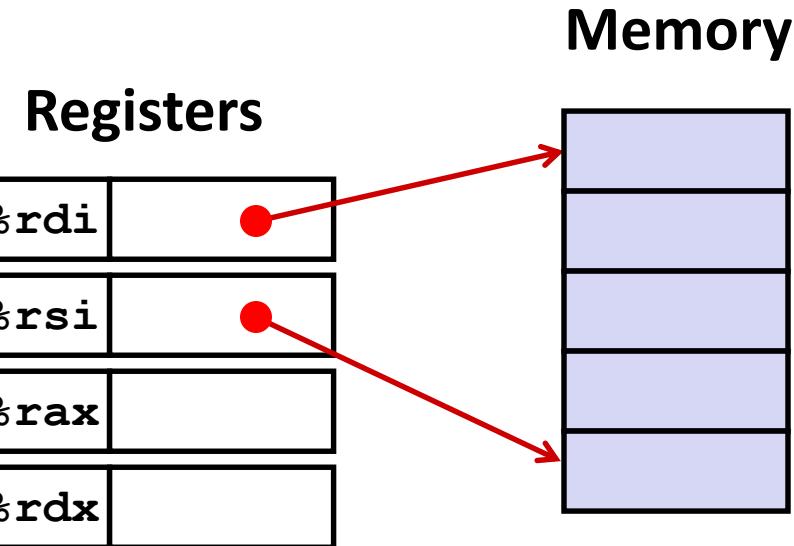
# Example of Simple Addressing Modes

```
void 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;  
}
```

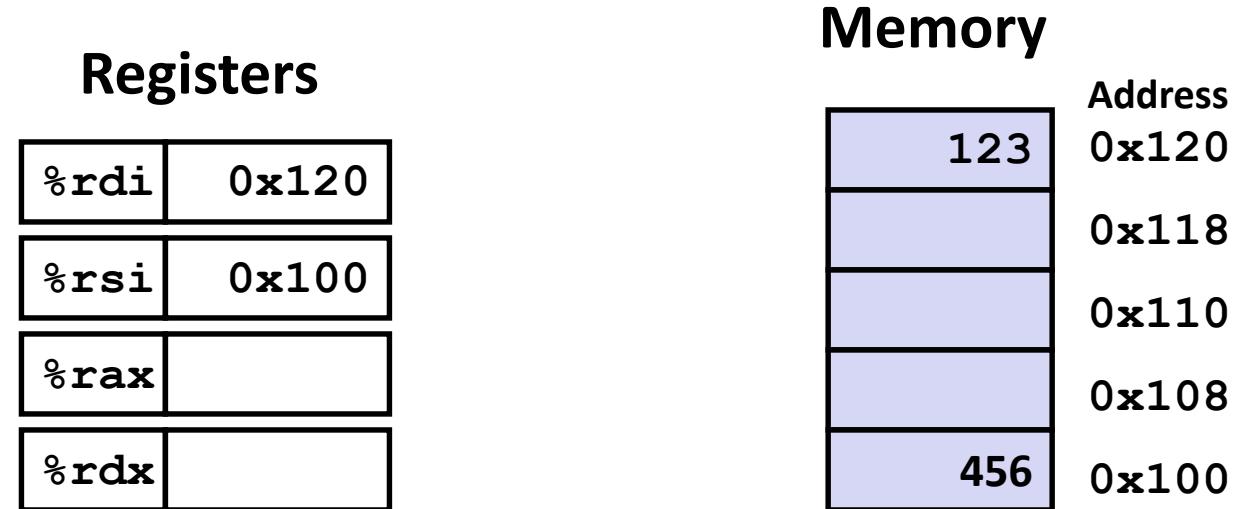


Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

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

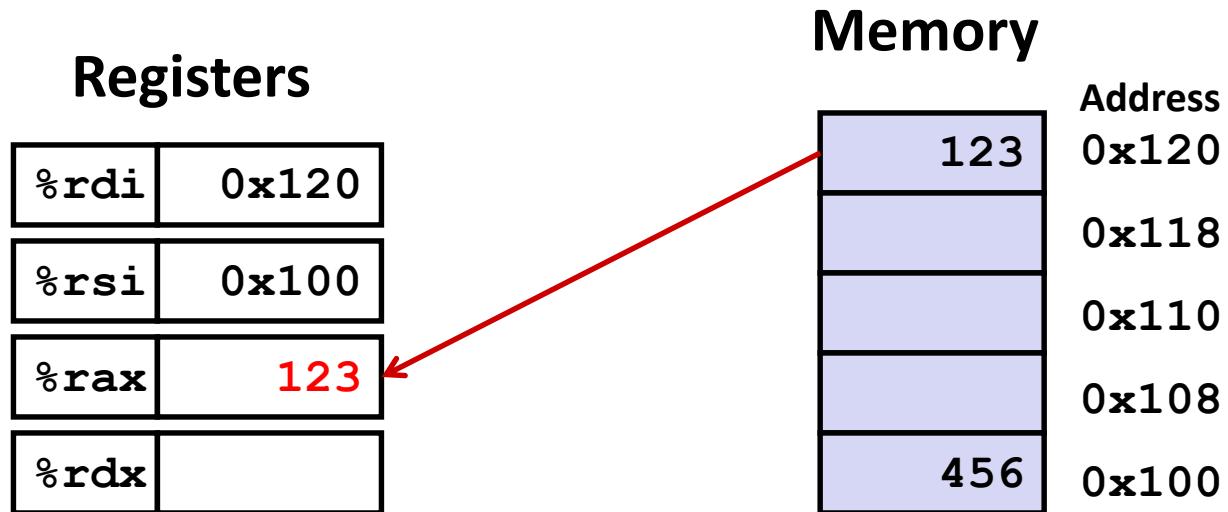
# Understanding Swap()



swap:

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

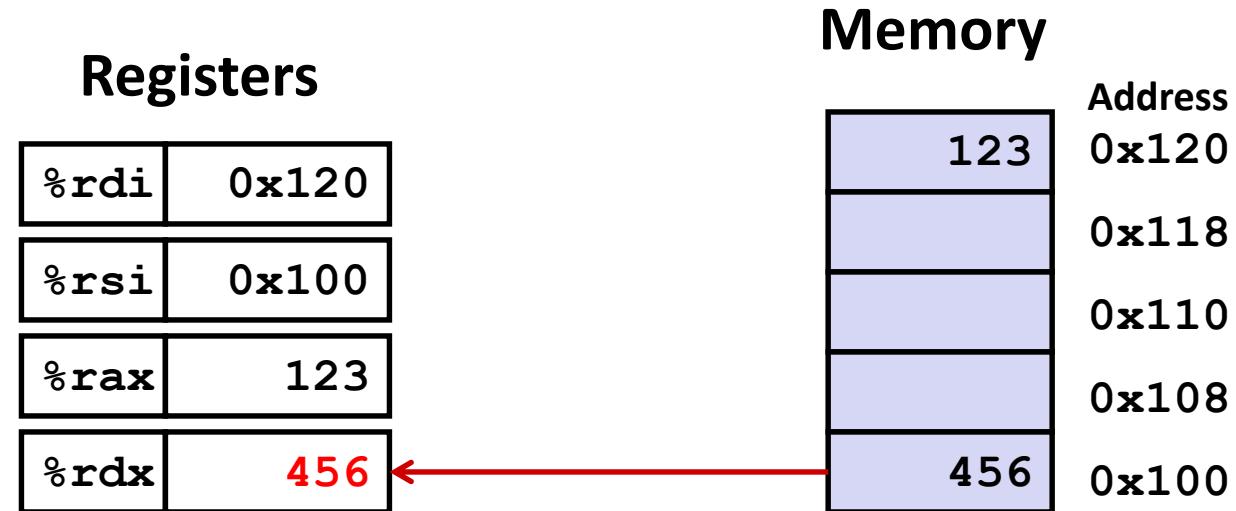
# Understanding Swap()



swap:

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

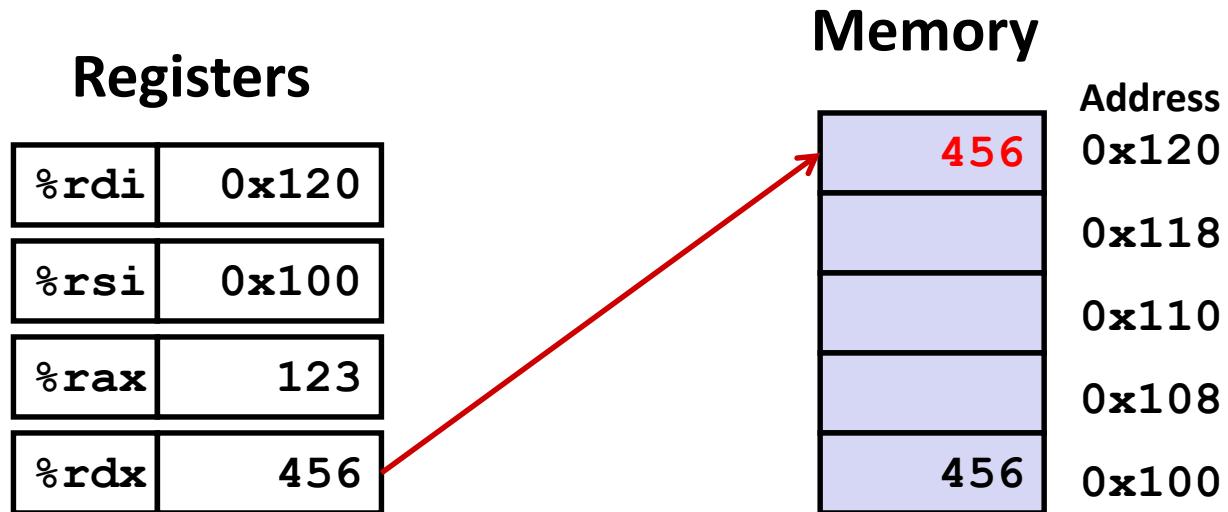
# Understanding Swap()



swap:

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

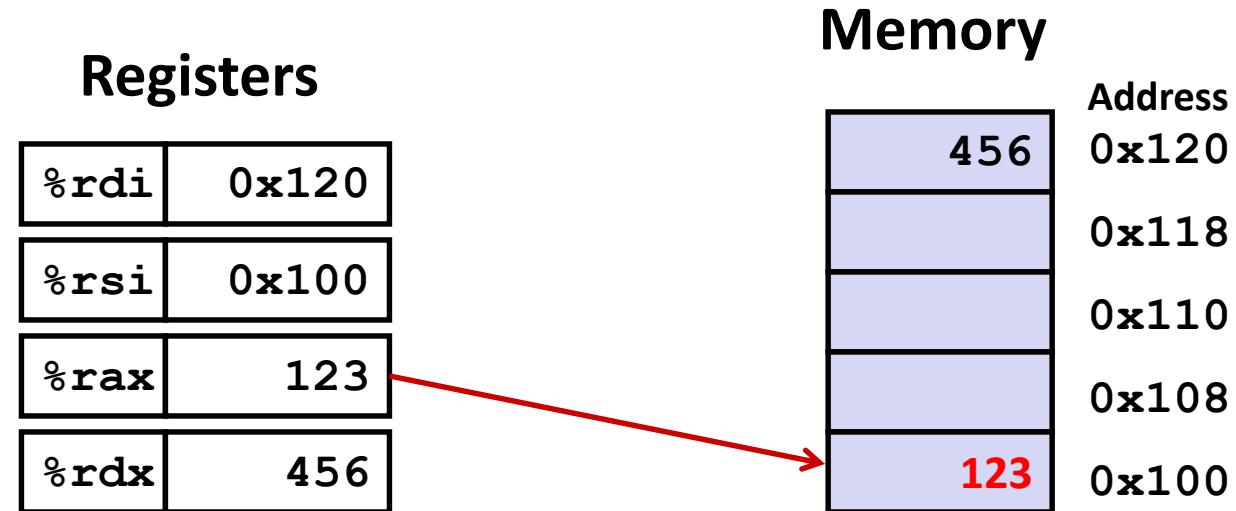
# Understanding Swap()



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
```

# 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

$D(Rb,Ri,S)$

$\text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+ D]$

- 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

$(Rb,Ri)$

$\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]$

$D(Rb,Ri)$

$\text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]$

$(Rb,Ri,S)$

$\text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]$

# Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

# 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  $x + k*y$ 
  - $k = 1, 2, 4, \text{ or } 8$

## ■ **Example**

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

Converted to ASM by compiler:

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

# Some Arithmetic Operations

## ■ Two Operand Instructions:

<i>Format</i>	<i>Computation</i>	
addq <i>Src,Dest</i>	Dest = Dest + Src	
subq <i>Src,Dest</i>	Dest = Dest – Src	
imulq <i>Src,Dest</i>	Dest = Dest * Src	
salq <i>Src,Dest</i>	Dest = Dest << Src	<i>Also called shlq</i>
sarq <i>Src,Dest</i>	Dest = Dest >> Src	<i>Arithmetic</i>
shrq <i>Src,Dest</i>	Dest = Dest >> Src	<i>Logical</i>
xorq <i>Src,Dest</i>	Dest = Dest ^ Src	
andq <i>Src,Dest</i>	Dest = Dest & Src	
orq <i>Src,Dest</i>	Dest = Dest   Src	

- Watch out for argument order!
- 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*       $Dest = \sim Dest$

## ■ See the book or Intel x86-64 manual for more instructions

# 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
  - But, only used once

# 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
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

# Machine Programming I: 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

# Machine-Level Programming II: Control

CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

# Processor State (x86-64, Partial)

## ■ Information about currently executing program

- Temporary data ( `%rax`, ... )
- Location of runtime stack ( `%rsp` )
- Location of current code control point ( `%rip`, ... )
- Status of recent tests ( `CF`, `ZF`, `SF`, `OF` )

Current stack top

Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip`      Instruction pointer

`CF`    `ZF`    `SF`    `OF`      Condition codes

# Condition Codes (Implicit Setting)

## ■ Single bit registers

- CF Carry Flag (for unsigned) SF Sign Flag (for signed)
- ZF Zero Flag OF Overflow Flag (for signed)

## ■ Implicitly set (think of it as side effect) by arithmetic operations

Example: `addq Src, Dest  $\leftrightarrow$  t = a+b`

**CF set** if carry out from most significant bit (unsigned overflow)

**ZF set** if  $t == 0$

**SF set** if  $t < 0$  (as signed)

**OF set** if two's-complement (signed) overflow

$(a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ || \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0)$

## ■ Not set by `leaq` instruction

# Condition Codes (Explicit Setting: Compare)

## ■ Explicit Setting by Compare Instruction

- **cmpq Src2, Src1**
- **cmpq b, a** like computing  $a - b$  without setting destination
- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if  $a == b$
- **SF set** if  $(a - b) < 0$  (as signed)
- **OF set** if two's-complement (signed) overflow  
$$(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \|\ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$$

# Condition Codes (Explicit Setting: Test)

## ■ Explicit Setting by Test instruction

- `testq Src2, Src1`
  - `testq 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`

# Reading Condition Codes

## ■ SetX Instructions

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

SetX	Condition	Description
<b>sete</b>	<b>ZF</b>	<b>Equal / Zero</b>
<b>setne</b>	<b><math>\sim ZF</math></b>	<b>Not Equal / Not Zero</b>
<b>sets</b>	<b>SF</b>	<b>Negative</b>
<b>setns</b>	<b><math>\sim SF</math></b>	<b>Nonnegative</b>
<b>setg</b>	<b><math>\sim (SF^OF) \ \&amp; \ \sim ZF</math></b>	<b>Greater (Signed)</b>
<b>setge</b>	<b><math>\sim (SF^OF)</math></b>	<b>Greater or Equal (Signed)</b>
<b>setl</b>	<b><math>(SF^OF)</math></b>	<b>Less (Signed)</b>
<b>setle</b>	<b><math>(SF^OF) \   \ ZF</math></b>	<b>Less or Equal (Signed)</b>
<b>seta</b>	<b><math>\sim CF \ \&amp; \ \sim ZF</math></b>	<b>Above (unsigned)</b>
<b>setb</b>	<b>CF</b>	<b>Below (unsigned)</b>

# x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

# Reading Condition Codes (Cont.)

## ■ SetX Instructions:

- Set single byte based on combination of condition codes

## ■ One of addressable byte registers

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
  - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
    return x > y;
}
```

Register	Use(s)
<code>%rdi</code>	Argument <code>x</code>
<code>%rsi</code>	Argument <code>y</code>
<code>%rax</code>	Return value

```
cmpq    %rsi, %rdi      # Compare x:y
setg    %al               # Set when >
movzbl  %al, %eax       # Zero rest of %rax
ret
```

# Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

# Jumping

## ■ jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF) & ~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)   ZF	Less or Equal (Signed)
ja	~CF & ~ZF	Above (unsigned)
jb	CF	Below (unsigned)

# Conditional Branch Example (Old Style)

## ■ Generation

```
$ gcc -Og -S -fno-if-conversion control.c
```

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

absdiff:

```
    cmpq    %rsi, %rdi  # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:      # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rax	Return value

# Expressing with Goto Code

- C allows goto statement
- Jump to position designated by label

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
    (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

# General Conditional Expression Translation (Using Branches)

## C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

## Goto Version

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

# Using Conditional Moves

## ■ Conditional Move Instructions

- Instruction supports:  
if (Test) Dest  $\leftarrow$  Src
- Supported in post-1995 x86 processors
- GCC tries to use them
  - But, only when known to be safe

## ■ Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

### C Code

```
val = Test  
? Then_Expr  
: Else_Expr;
```

### Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

# Conditional Move Example

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

absdiff:

```
    movq    %rdi, %rax    # x
    subq    %rsi, %rax    # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx    # eval = y-x
    cmpq    %rsi, %rdi    # x:y
    cmovle %rdx, %rax    # if <=, result = eval
    ret
```

# Bad Cases for Conditional Move

## Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

## Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

## Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

# Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

# “Do-While” Loop Example

## C Code

```
long pcount_do
(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

## Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- Count number of 1's in argument **x** (“popcount”)
- Use conditional branch to either continue looping or to exit loop

# “Do-While” Loop Compilation

## Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>

```
        movl    $0, %eax      # result = 0
.L2:                           # loop:
        movq    %rdi, %rdx
        andl    $1, %edx      # t = x & 0x1
        addq    %rdx, %rax    # result += t
        shrq    %rdi          # x >>= 1
        jne     .L2          # if (x) goto loop
        rep; ret
```

# General “Do-While” Translation

## C Code

```
do  
  Body  
  while ( Test );
```

## Goto Version

```
loop:  
  Body  
  if ( Test )  
    goto loop
```

■ **Body:** {  
    *Statement*<sub>1</sub>;  
    *Statement*<sub>2</sub>;  
    ...  
    *Statement*<sub>n</sub>;  
}

# General “While” Translation #1

- “Jump-to-middle” translation
- Used with -Og

## While version

```
while ( Test)  
    Body
```



## Goto Version

```
goto test;  
loop:  
    Body  
test:  
    if ( Test)  
        goto loop;  
done:
```

# While Loop Example #1

## C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

## Jump to Middle

```
long pcount_goto_jtm
(unsigned long x) {
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if(x) goto loop;
    return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

# General “While” Translation #2

## While version

```
while ( Test)  
    Body
```

- “Do-while” conversion
- Used with -O1

## Do-While Version

```
if ( ! Test)  
    goto done;  
do  
    Body  
    while( Test );  
done:
```

## Goto Version

```
if ( ! Test)  
    goto done;  
loop:  
    Body  
    if ( Test )  
        goto loop;  
done:
```

# While Loop Example #2

## C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

## Do-While Version

```
long pcount_goto_dw
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

# “For” Loop Form

## General Form

```
for (Init; Test; Update )  
    Body
```

```
#define WSIZE 8*sizeof(int)  
long pcount_for  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    for (i = 0; i < WSIZE; i++)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
    }  
    return result;  
}
```

### Init

```
i = 0
```

### Test

```
i < WSIZE
```

### Update

```
i++
```

### Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

# “For” Loop → While Loop

## For Version

```
for (Init; Test; Update)  
    Body
```



## While Version

```
Init;  
  
while (Test) {  
    Body  
    Update;  
}
```

# For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

# “For” Loop Do-While Conversion

C Code

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Goto Version

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (! (i < WSIZE)) Init
        goto done; ! Test
loop:
{
    unsigned bit =
        (x >> i) & 0x1; Body
    result += bit;
}
Update
if (i < WSIZE)
    goto loop;
done:
    return result;
}
```

- Initial test can be optimized away

# Today

- Control: Condition codes
- Conditional branches
- Loops
- **Switch Statements**

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

# Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

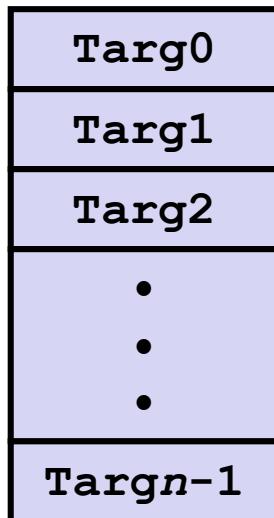
# Jump Table Structure

## Switch Form

```
switch(x) {  
    case val_0:  
        Block 0  
    case val_1:  
        Block 1  
    . . .  
    case val_{n-1}:  
        Block n-1  
}
```

## Jump Table

jtab:

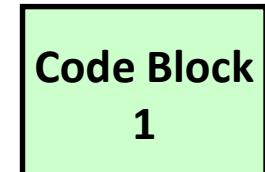


## Jump Targets

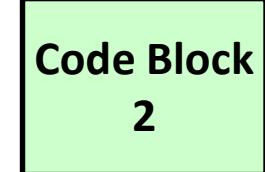
Targ0:



Targ1:

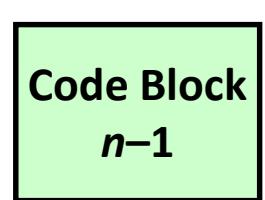


Targ2:



•  
•  
•

Targ $n-1$ :



## Translation (Extended C)

```
goto *JTab[x];
```

# Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja     .L8
    jmp    * .L4(,%rdi,8)
```

What range of values  
takes default?

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

Note that **w** not  
initialized here

# Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi      # x:6
    ja     .L8           # Use default
    Indirect jump    * .L4(,%rdi,8) # goto *JTab[x]
```

Jump table

```
.section  .rodata
.align 8
.L4:
    .quad   .L8  # x = 0
    .quad   .L3  # x = 1
    .quad   .L5  # x = 2
    .quad   .L9  # x = 3
    .quad   .L8  # x = 4
    .quad   .L7  # x = 5
    .quad   .L7  # x = 6
```

# Assembly Setup Explanation

## ■ Table Structure

- Each target requires 8 bytes
- Base address at `.L4`

## ■ Jumping

- **Direct:** `jmp .L8`
- Jump target is denoted by label `.L8`

- **Indirect:** `jmp * .L4(,%rdi,8)`
- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective Address `.L4 + x*8`
  - Only for  $0 \leq x \leq 6$

### Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8    # x = 0
.quad      .L3    # x = 1
.quad      .L5    # x = 2
.quad      .L9    # x = 3
.quad      .L8    # x = 4
.quad      .L7    # x = 5
.quad      .L7    # x = 6
```

# Jump Table

## Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8  # x = 0
.quad      .L3  # x = 1
.quad      .L5  # x = 2
.quad      .L9  # x = 3
.quad      .L8  # x = 4
.quad      .L7  # x = 5
.quad      .L7  # x = 6
```

```
switch(x) {
    case 1:          // .L3
        w = y*z;
        break;
    case 2:          // .L5
        w = y/z;
        /* Fall Through */
    case 3:          // .L9
        w += z;
        break;
    case 5:
    case 6:          // .L7
        w -= z;
        break;
    default:         // .L8
        w = 2;
}
```

# Code Blocks ( $x == 1$ )

```
switch(x) {  
    case 1:          // .L3  
        w = y*z;  
        break;  
    . . .  
}
```

```
.L3:  
    movq    %rsi, %rax  # y  
    imulq   %rdx, %rax  # y*z  
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Handling Fall-Through

```
long w = 1;  
.  
.  
switch(x) {  
.  
.case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

```
case 2:  
    w = y/z;  
    goto merge;
```

```
case 3:  
    w = 1;  
  
merge:  
    w += z;
```



# Code Blocks ( $x == 2$ , $x == 3$ )

```
long w = 1;  
.  
.  
switch(x) {  
.  
. . .  
case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

```
.L5:          # Case 2  
    movq    %rsi, %rax  
    cqto  
    idivq   %rcx      # y/z  
    jmp     .L6        # goto merge  
.L9:          # Case 3  
    movl    $1, %eax    # w = 1  
.L6:          # merge:  
    addq    %rcx, %rax # w += z  
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Code Blocks ( $x == 5$ , $x == 6$ , default)

```
switch(x) {  
    . . .  
    case 5: // .L7  
    case 6: // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

```
.L7:                      # Case 5,6  
    movl $1, %eax      # w = 1  
    subq %rdx, %rax   # w -= z  
    ret  
.L8:                      # Default:  
    movl $2, %eax      # 2  
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	Return value

# Summarizing

## ■ C Control

- if-then-else
- do-while
- while, for
- switch

## ■ Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

## ■ Standard Techniques

- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (if-elseif-elseif-else)

# Summary

## ■ Today

- Control: Condition codes
- Conditional branches & conditional moves
- Loops
- Switch statements

## ■ Next Time

- Stack
- Call / return
- Procedure call discipline

# Machine-Level Programming III: Procedures

CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Mechanisms in Procedures

## ■ Passing control

- To beginning of procedure code
- Back to return point

## ■ Passing data

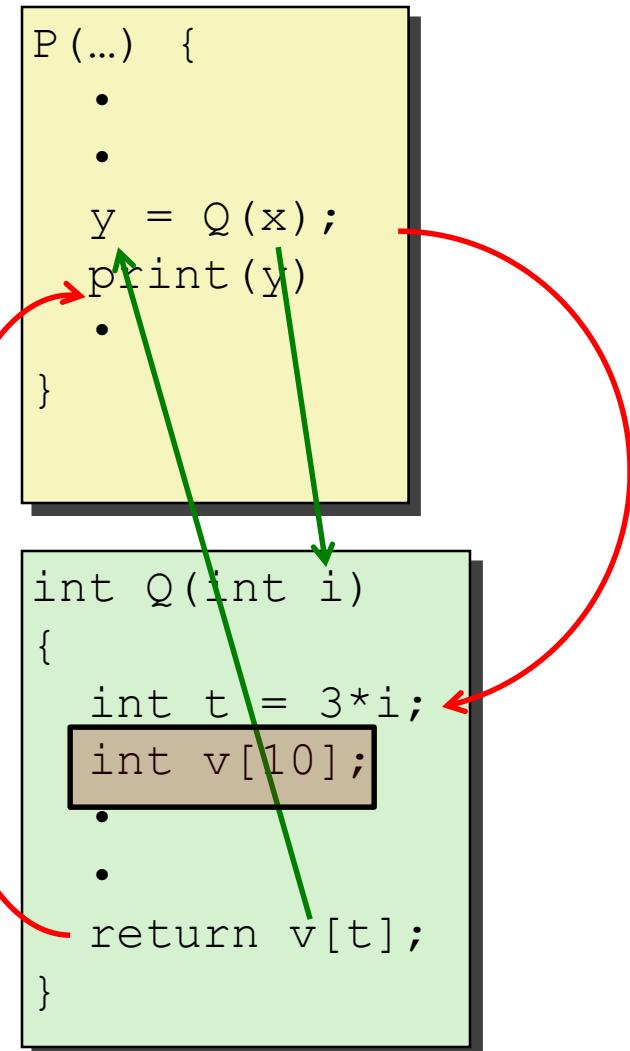
- Procedure arguments
- Return value

## ■ Memory management

- Allocate during procedure execution
- Deallocate upon return

## ■ Mechanisms all implemented with machine instructions

## ■ x86-64 implementation of a procedure uses only those mechanisms required



# Today

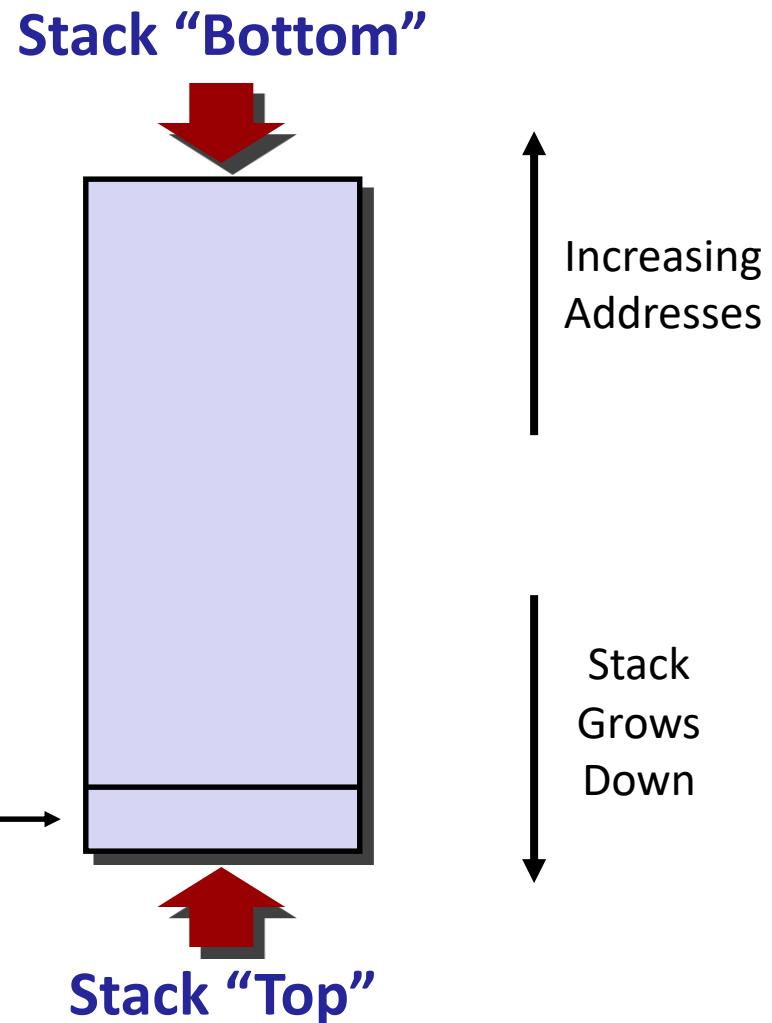
## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of “top” element

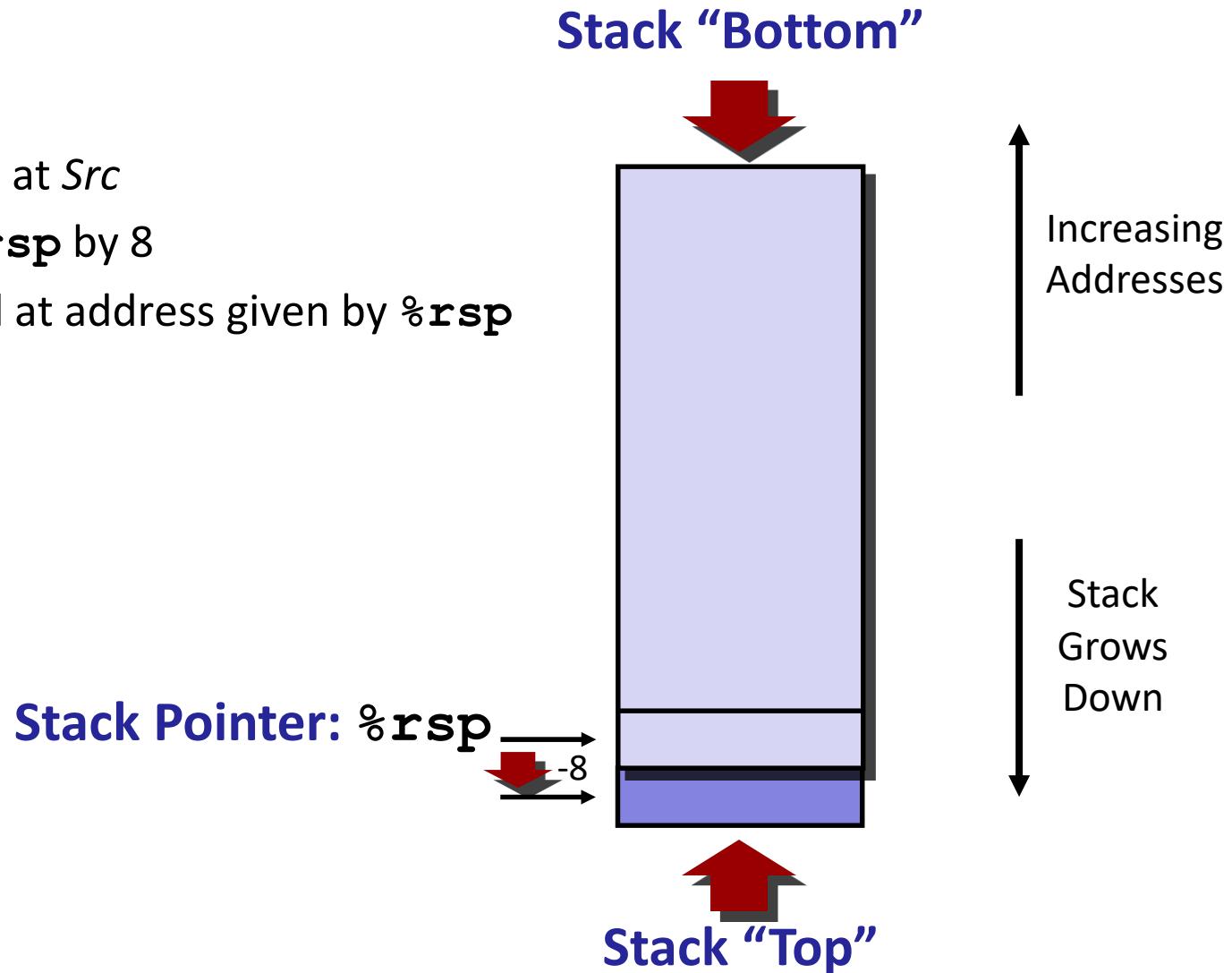
Stack Pointer: `%rsp` →



# x86-64 Stack: Push

## ■ **pushq Src**

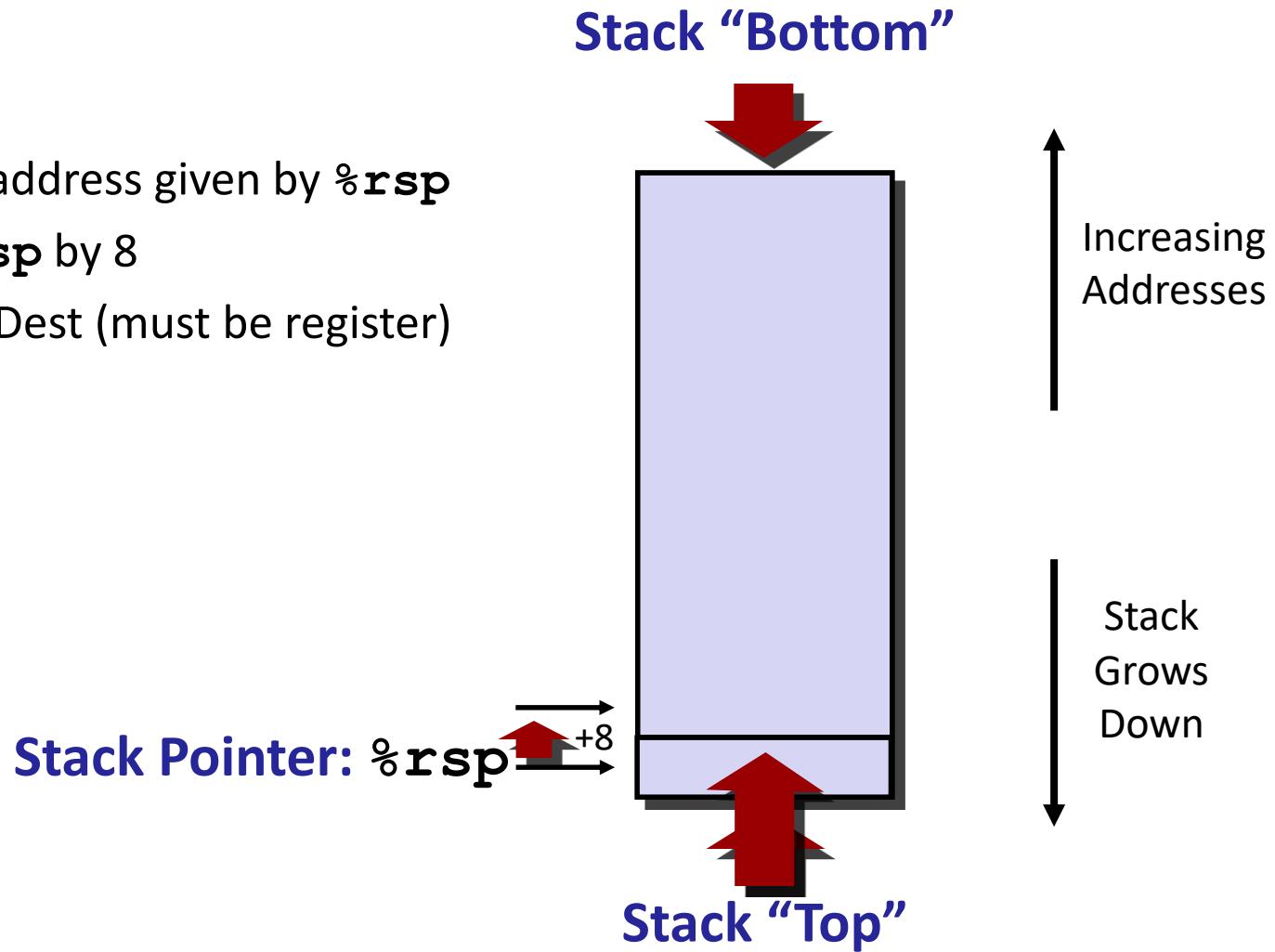
- Fetch operand at *Src*
- Decrement **%rsp** by 8
- Write operand at address given by **%rsp**



# x86-64 Stack: Pop

## ■ **popq Dest**

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (must be register)



# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Code Examples

```
void multstore  
    (long x, long y, long *dest)  
{  
    long t = mult2(x, y);  
    *dest = t;  
}
```

```
000000000400540 <multstore>:  
400540: push    %rbx          # Save %rbx  
400541: mov     %rdx,%rbx    # Save dest  
400544: callq   400550 <mult2>  # mult2(x,y)  
400549: mov     %rax,(%rbx)    # Save at dest  
40054c: pop     %rbx          # Restore %rbx  
40054d: retq               # Return
```

```
long mult2  
    (long a, long b)  
{  
    long s = a * b;  
    return s;  
}
```

```
000000000400550 <mult2>:  
400550: mov     %rdi,%rax    # a  
400553: imul   %rsi,%rax    # a * b  
400557: retq               # Return
```

# Procedure Control Flow

- Use stack to support procedure call and return

- **Procedure call: `call label`**

- Push return address on stack
  - Jump to *label*

- **Return address:**

- Address of the next instruction right after call
  - Example from disassembly

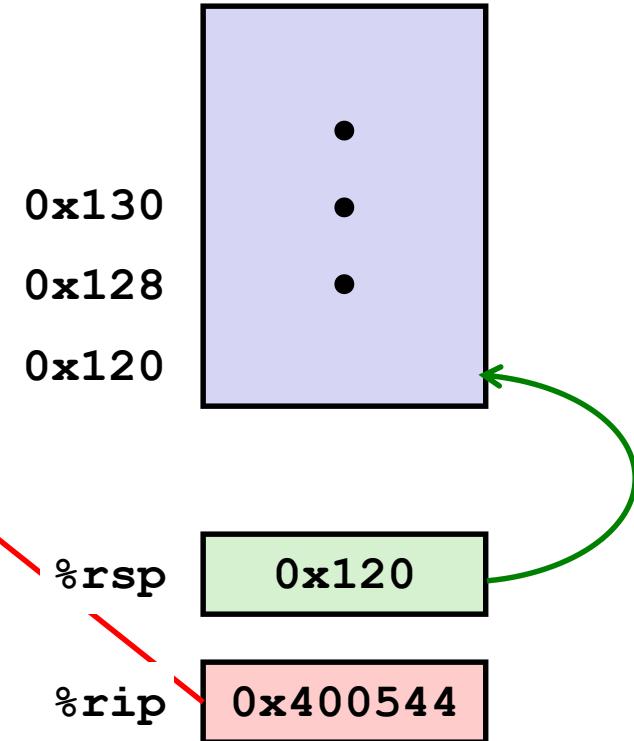
- **Procedure return: `ret`**

- Pop address from stack
  - Jump to address

# Control Flow Example #1

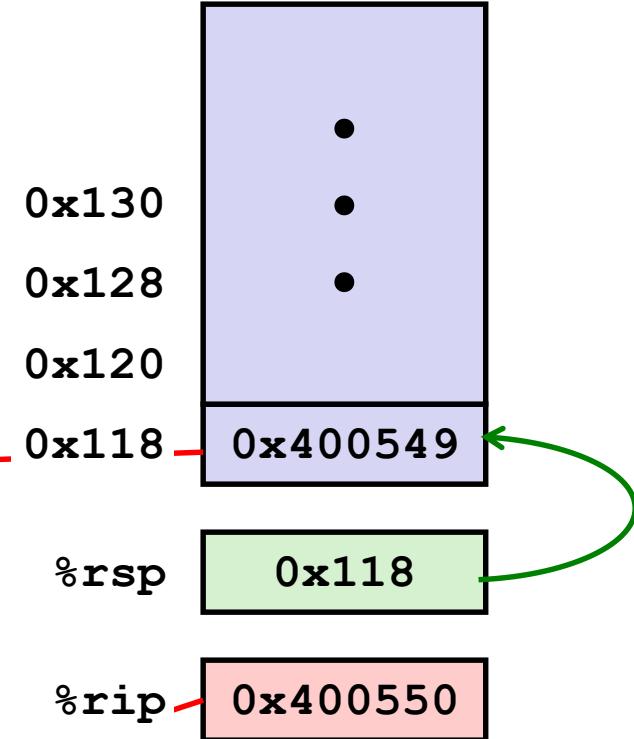
```
0000000000400540 <multstore>:  
.  
.  
400544: callq  400550 <mult2>  
400549: mov     %rax, (%rbx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: mov     %rdi,%rax  
.  
.  
400557: retq
```



# Control Flow Example #2

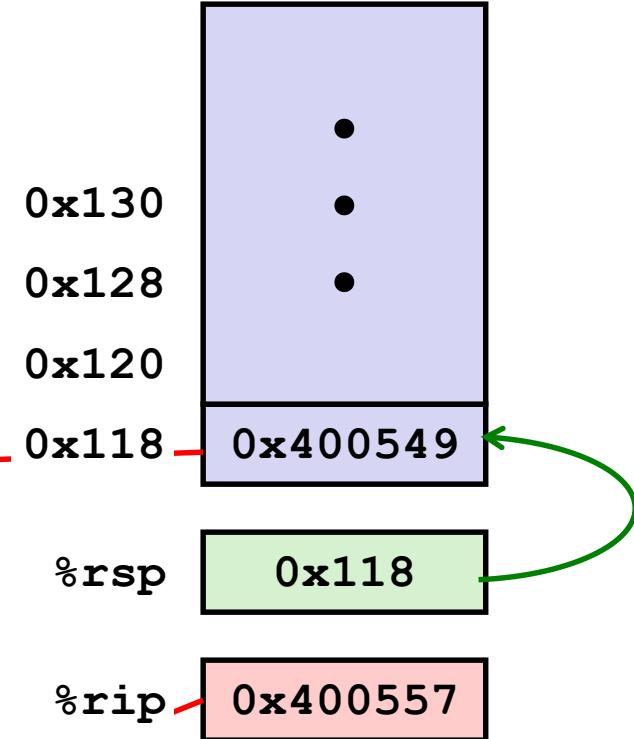
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx) ←
```



```
0000000000400550 <mult2>:  
400550: mov %rdi, %rax ←  
. .  
400557: retq
```

# Control Flow Example #3

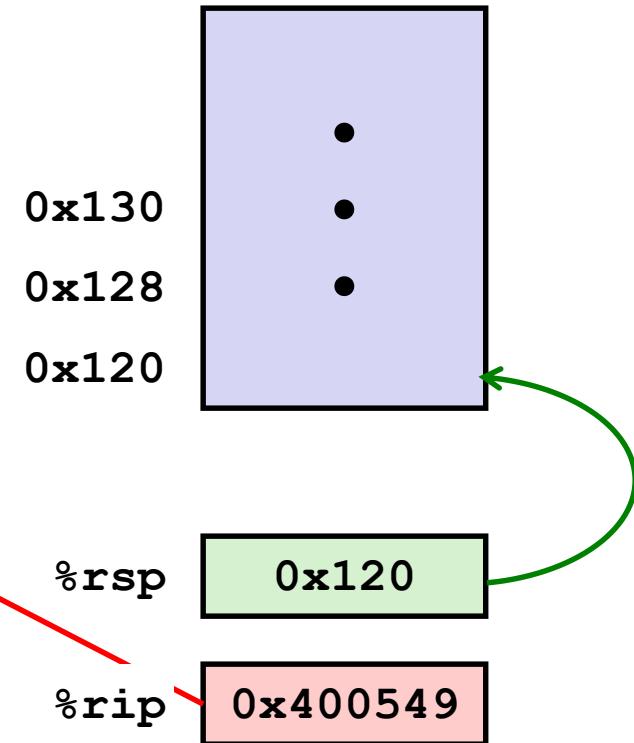
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx) ←
```



```
0000000000400550 <mult2>:  
400550: mov %rdi, %rax  
. .  
400557: retq ←
```

# Control Flow Example #4

```
0000000000400540 <multstore>:  
.  
.  
400544: callq  400550 <mult2>  
400549: mov     %rax, (%rbx) ←  
.  
.
```



```
0000000000400550 <mult2>:  
400550: mov     %rdi, %rax  
. .  
400557: retq
```

# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustrations of Recursion & Pointers

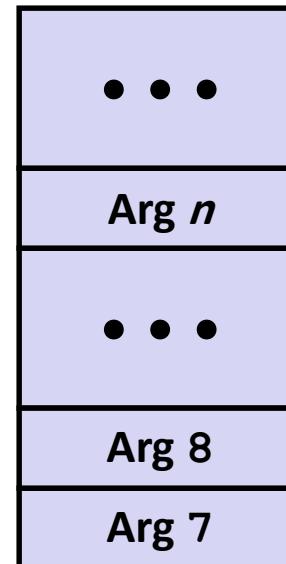
# Procedure Data Flow

## Registers

- First 6 arguments



## Stack



- Return value



- Only allocate stack space when needed

# Data Flow Examples:

```
void proc(long a1,long *a1p,int a2,int *a2p,short a3, short  
*a3p, char a4, char *a4p) {  
    *a1p += a1;  
    *a2p += a2;  
    *a3p += a3;  
    *a4p += a4;  
}
```

proc:

movq 16(%rsp), %r10	Fetch a4p (64bits)
addq %rdi, (%rsi)	*a1p += a1; (64bits)
addl %edx, (%rcx)	*a2p += a2; (32bits)
addw %r8w, (%r9)	*a3p += a3; (16bits)
movzbl 8(%rsp), %eax	Fetch a4
addb %al, (%r10)	*a4p += a4; (8bits)
ret	

# Data Flow Examples

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x in %rdi, y in %rsi, dest in %rdx
...
400541: mov    %rdx,%rbx          # Save dest
400544: callq  400550 <mult2>    # mult2(x,y)
# t in %rax
400549: mov    %rax,(%rbx)       # Save at dest
...
```

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
# a in %rdi, b in %rsi
400550: mov    %rdi,%rax        # a
400553: imul   %rsi,%rax        # a * b
# s in %rax
400557: retq   %rax             # Return
```

# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Stack-Based Languages

## ■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “*Reentrant*”
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

## ■ Stack discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

## ■ Stack allocated in *Frames*

- state for single procedure instantiation

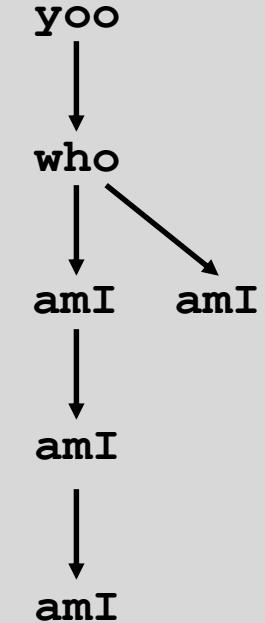
# Call Chain Example

```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```

```
who (...)  
{  
    • • •  
    amI () ;  
    • • •  
    amI () ;  
    • • •  
}
```

```
amI (...)  
{  
    •  
    •  
    amI () ;  
    •  
    •  
}
```

## Example Call Chain

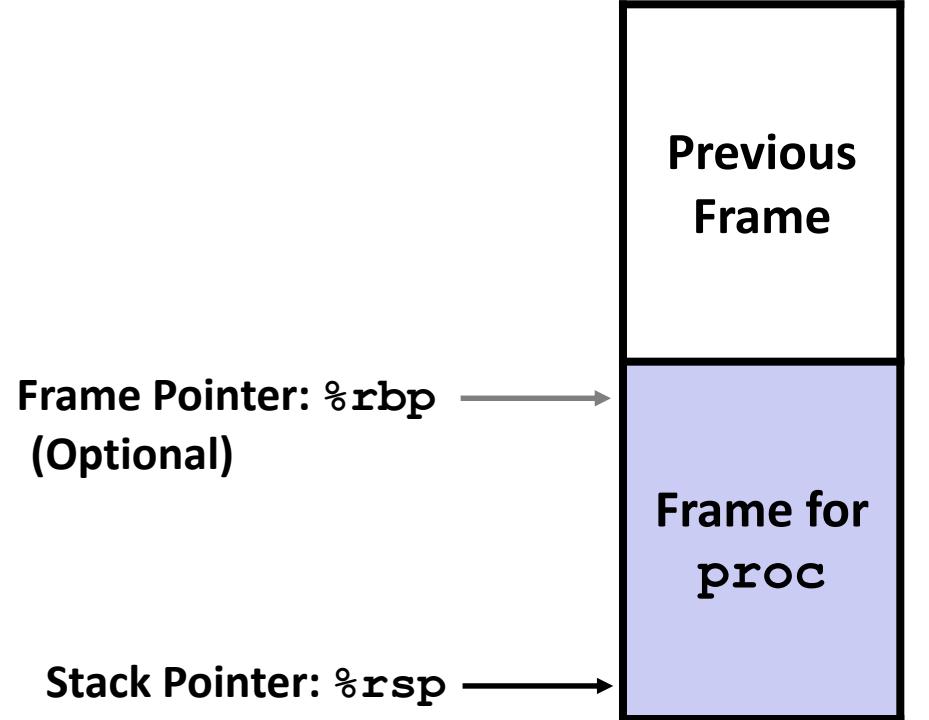


Procedure **amI ()** is recursive

# Stack Frames

## ■ Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)

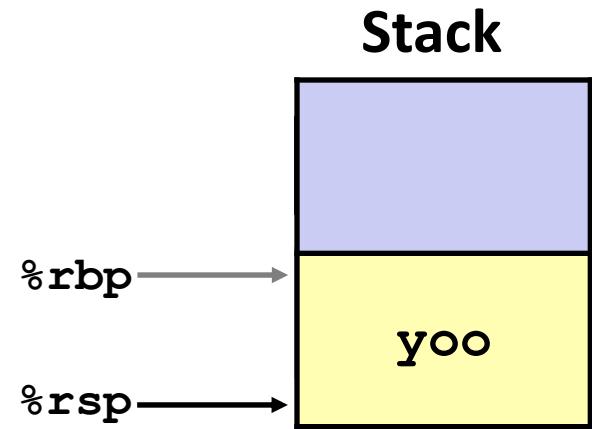
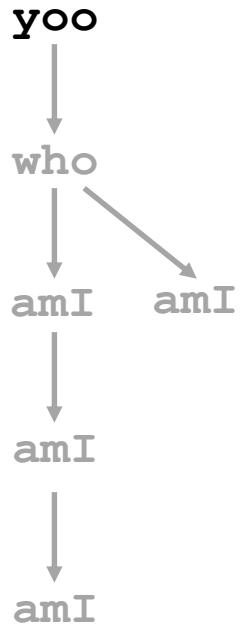


## ■ Management

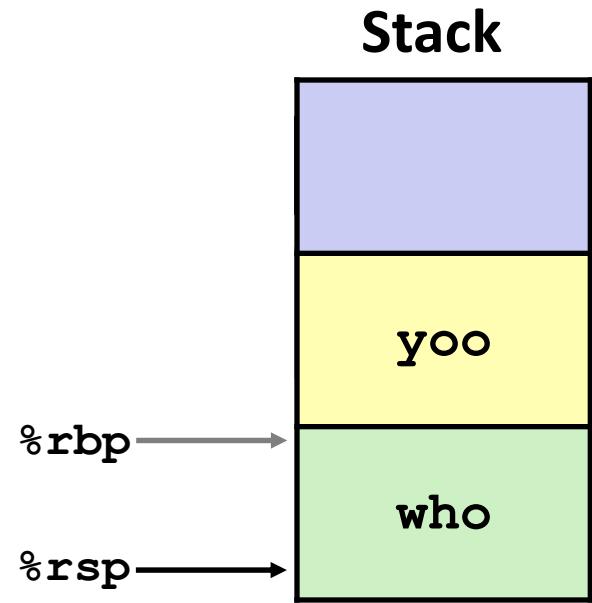
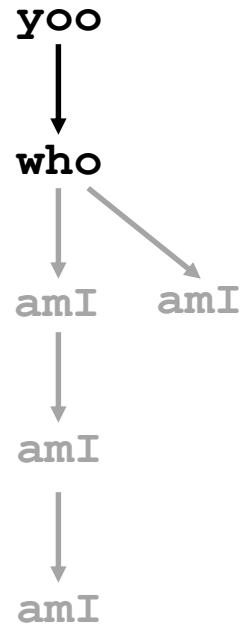
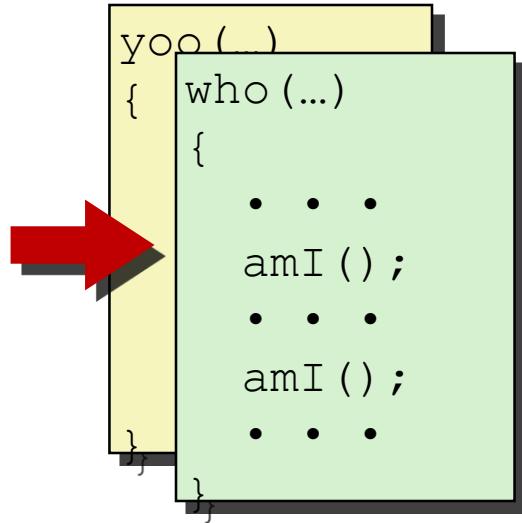
- Space allocated when enter procedure
  - “Set-up” code
  - Includes push by **call** instruction
- Deallocated when return
  - “Finish” code
  - Includes pop by **ret** instruction

# Example

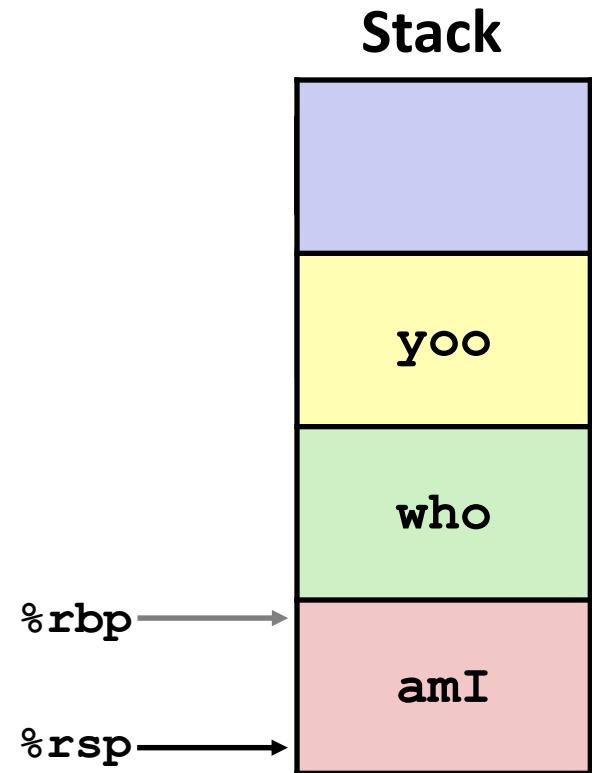
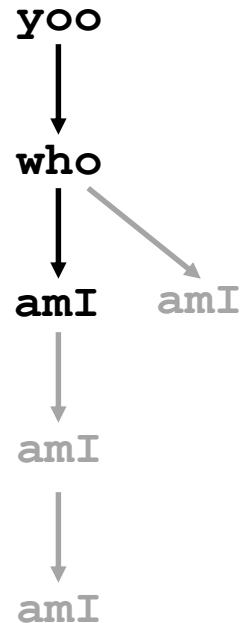
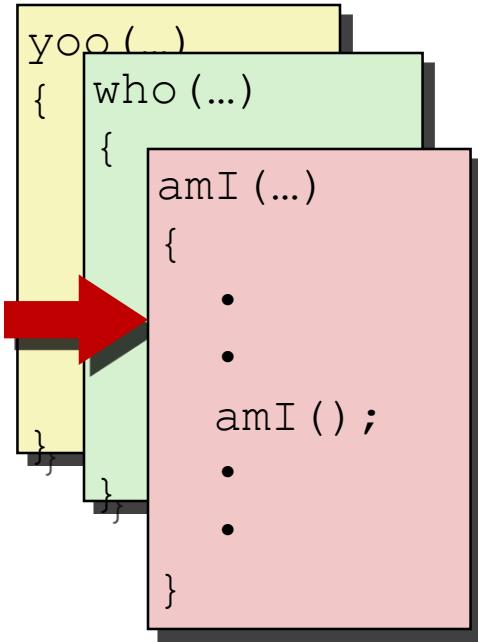
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



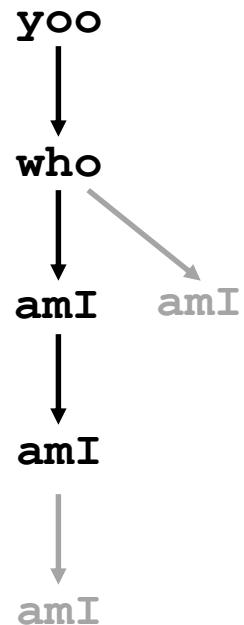
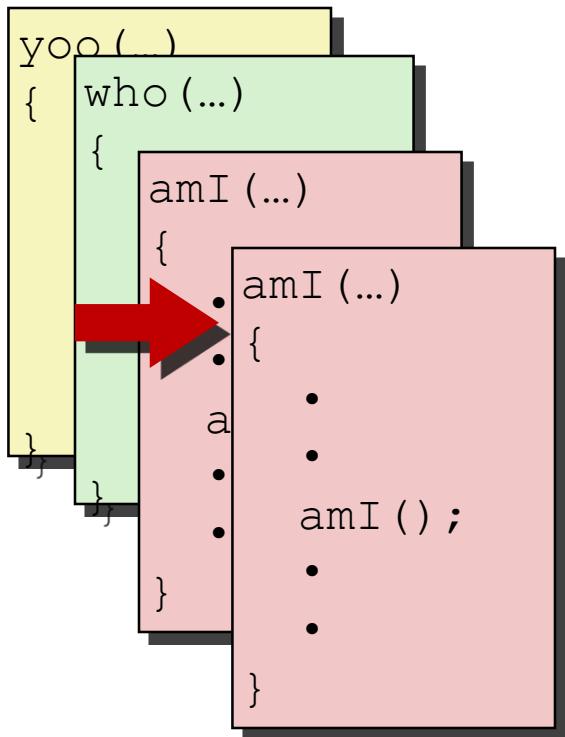
# Example



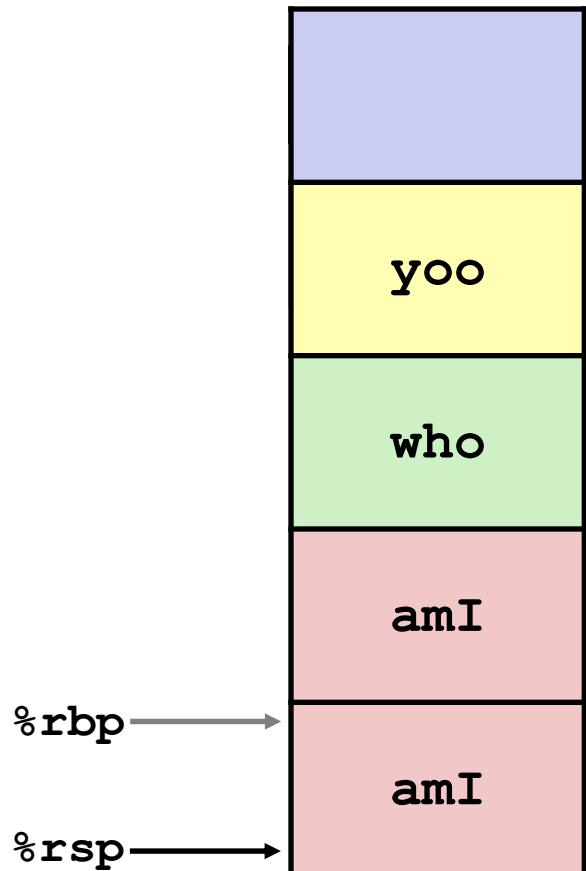
# Example



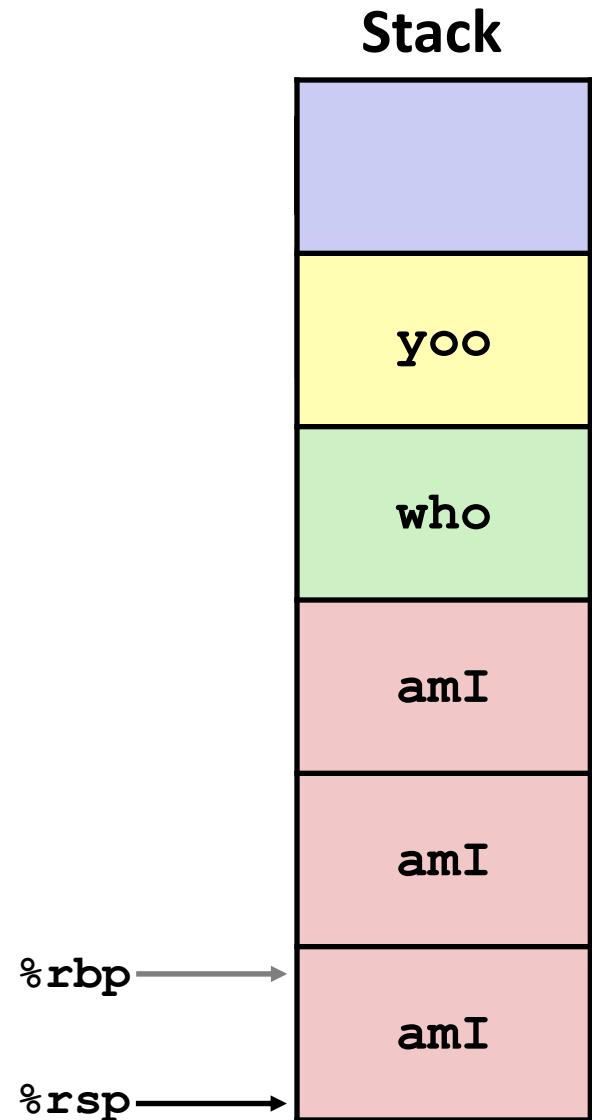
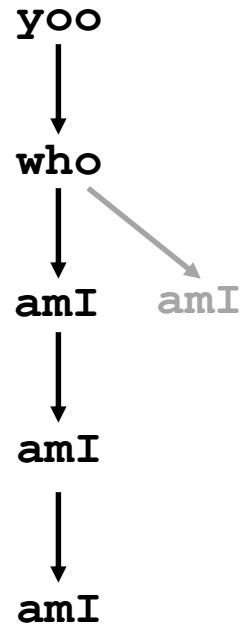
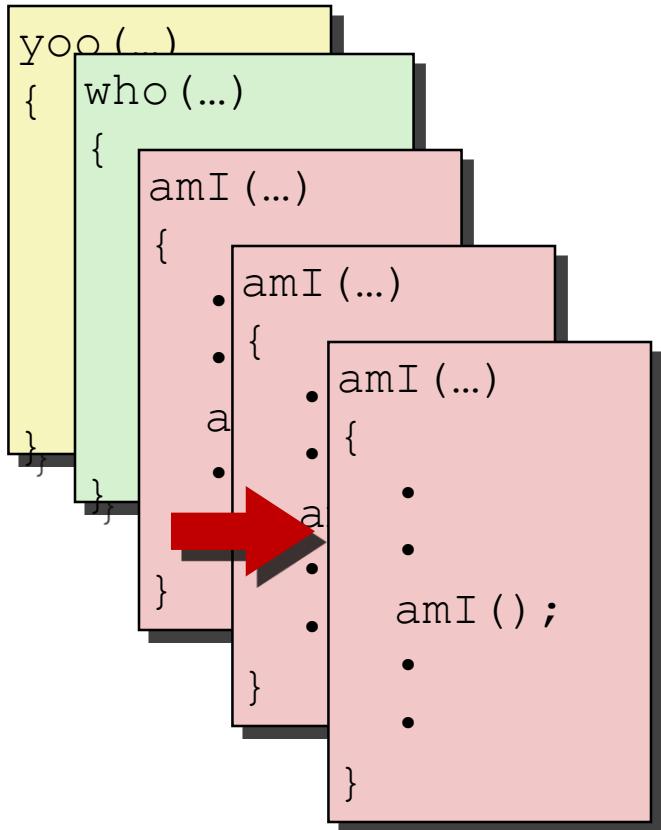
# Example



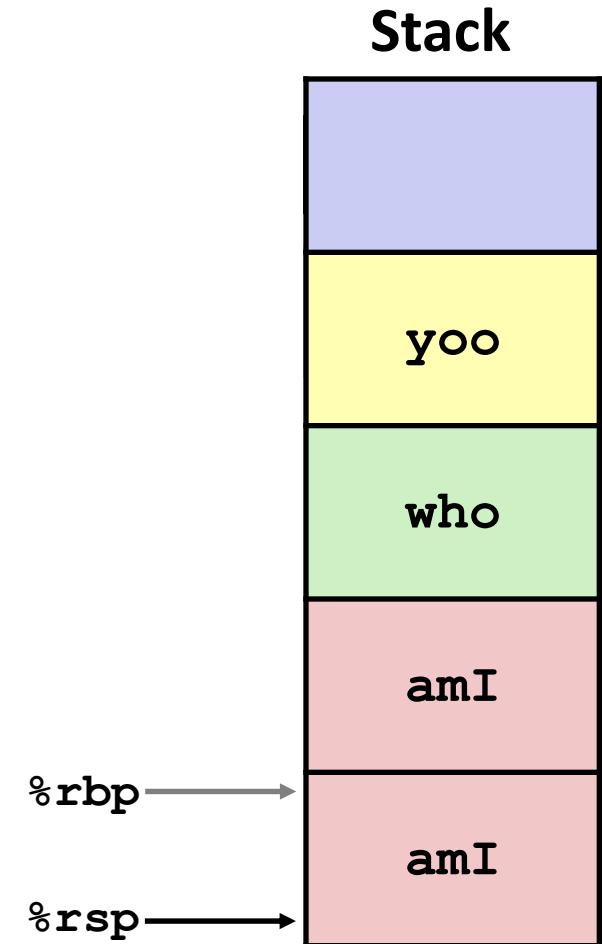
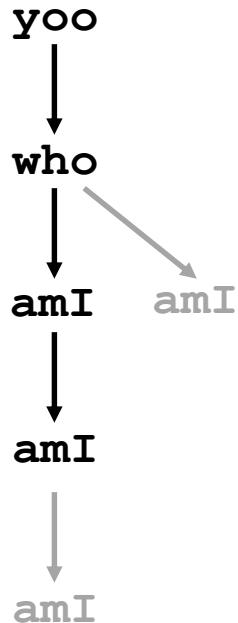
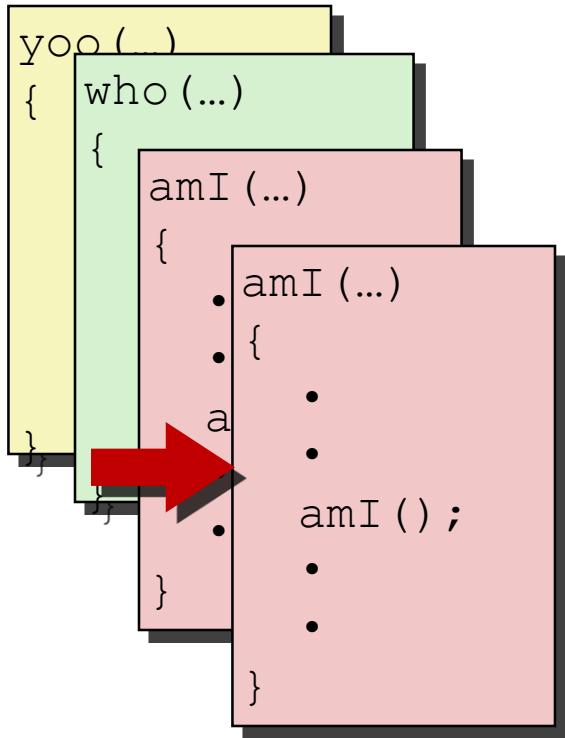
# Stack



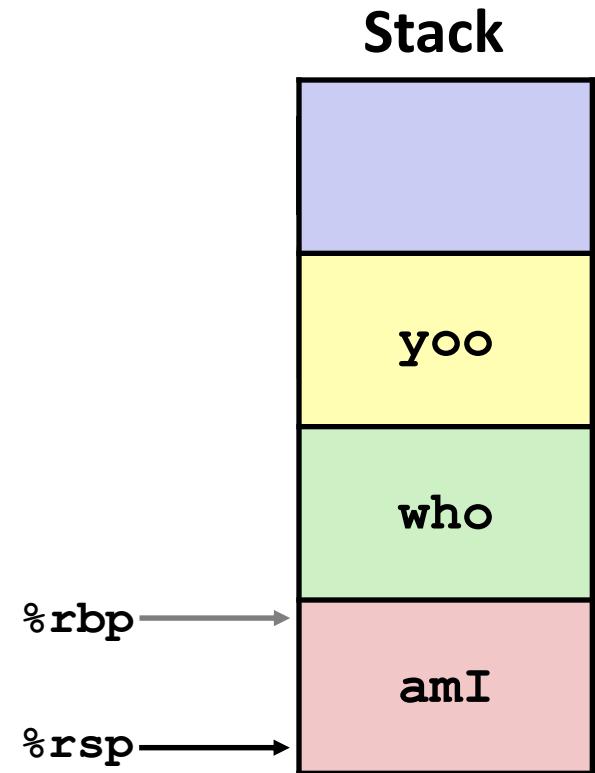
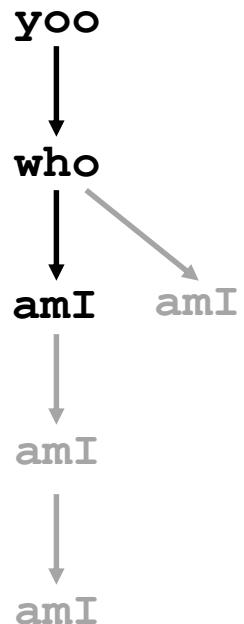
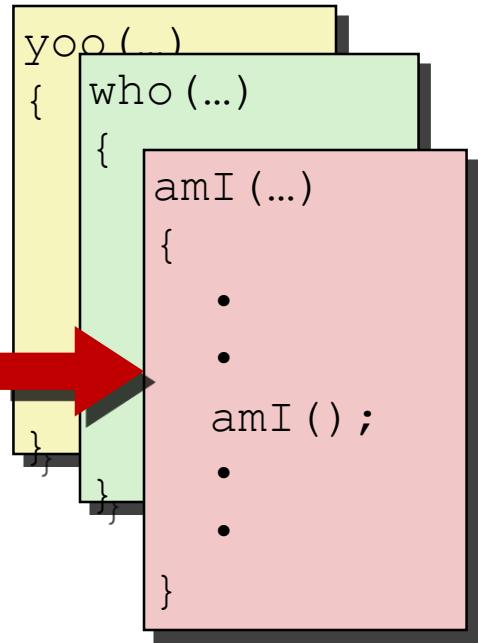
# Example



# Example

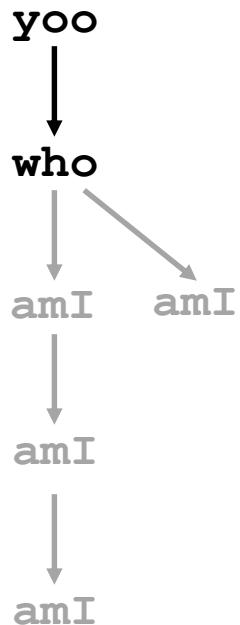


# Example

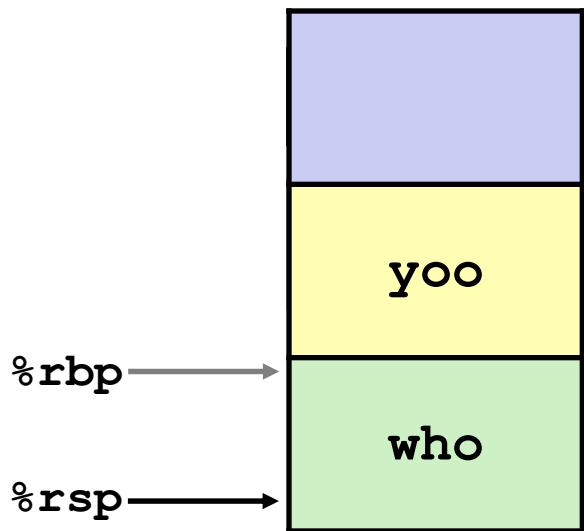


# Example

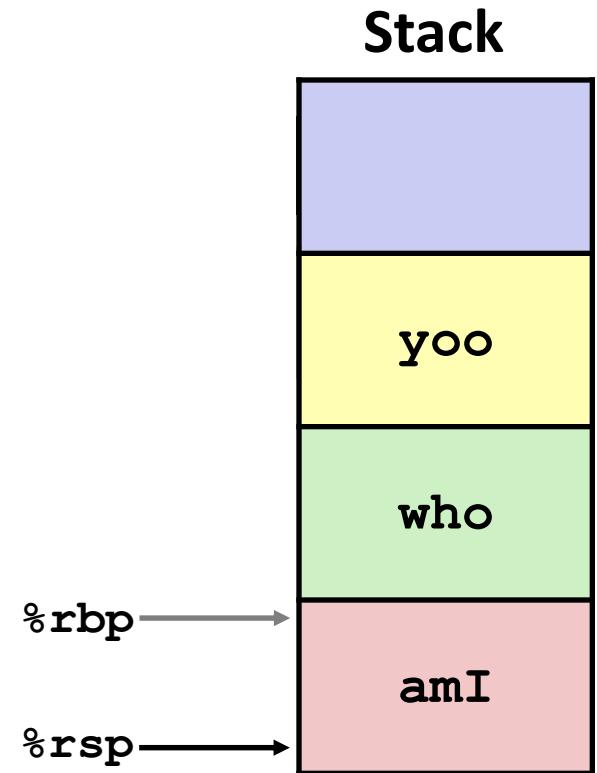
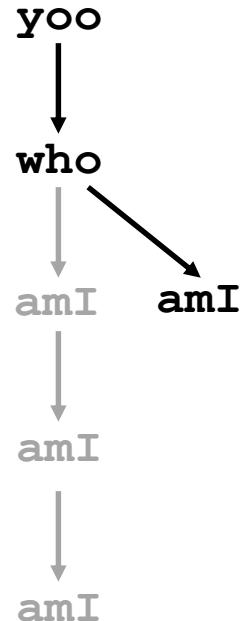
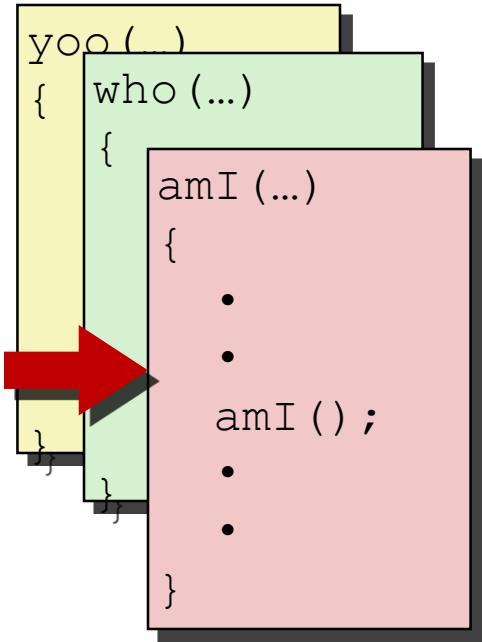
```
yoo(...)  
{   who(...)  
{  
    . . .  
    amI();  
    . . .  
    amI();  
    . . .  
}  
}
```



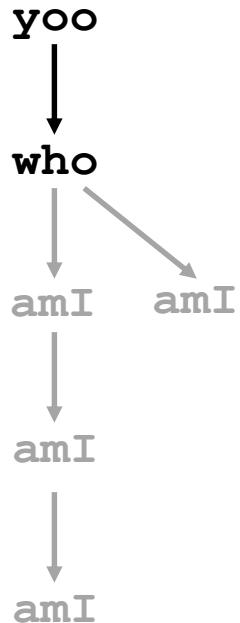
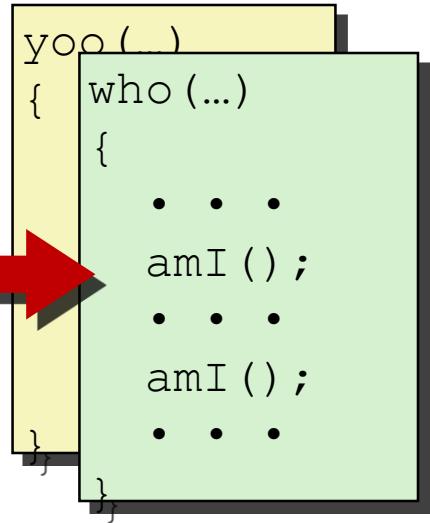
Stack



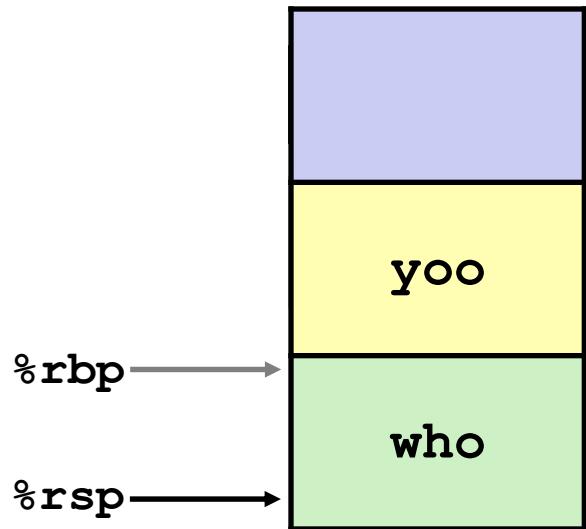
# Example



# Example

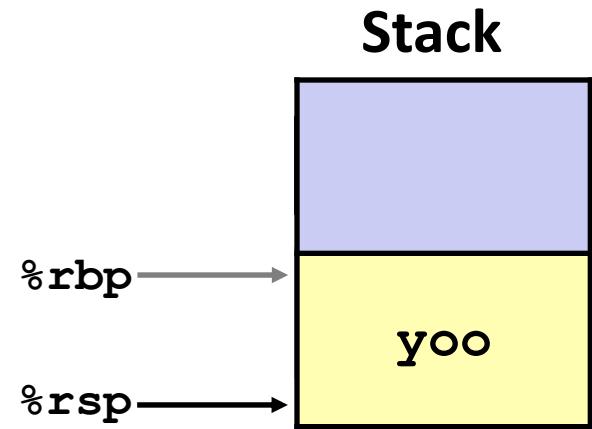
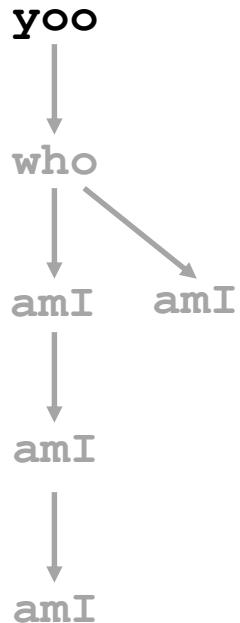


Stack



# Example

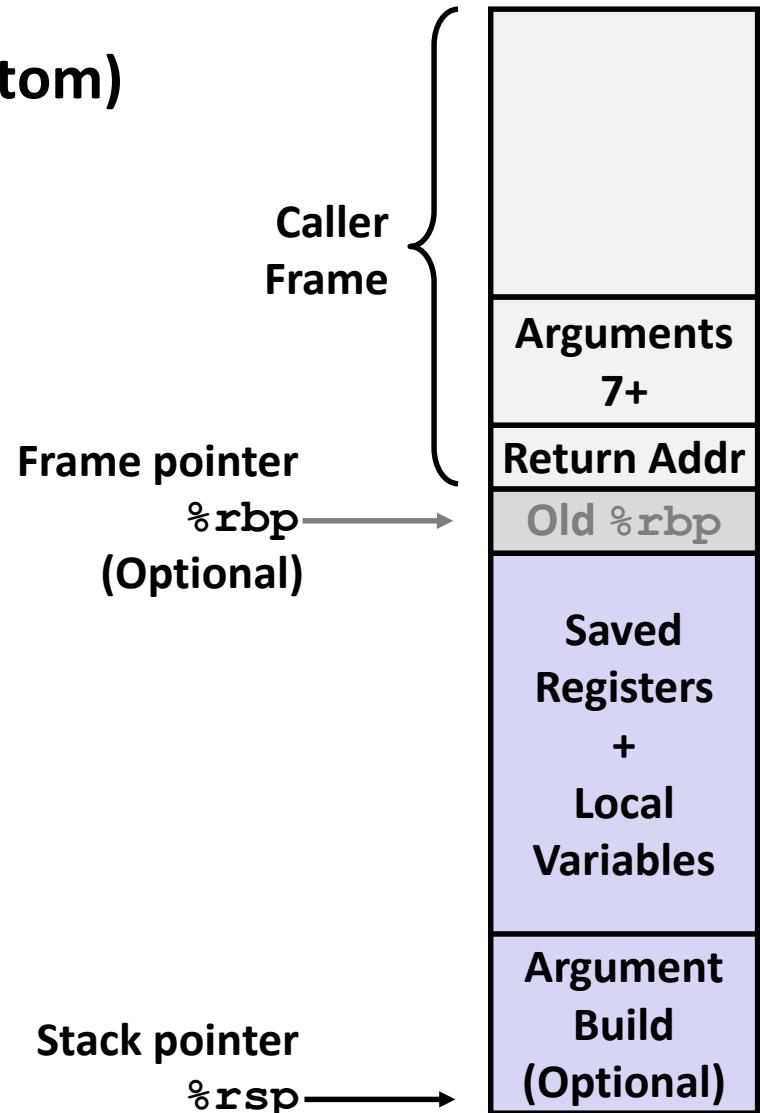
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}  
}
```



# x86-64/Linux Stack Frame

## ■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”  
Parameters for function about to call
- Local variables  
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)



## ■ Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call

# Example: incr

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

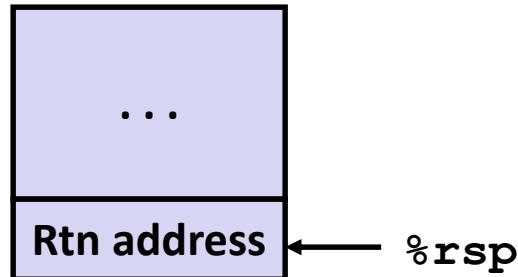
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <b>p</b>
%rsi	Argument <b>val</b> , <b>y</b>
%rax	<b>x</b> , Return value

# Example: Calling incr #1

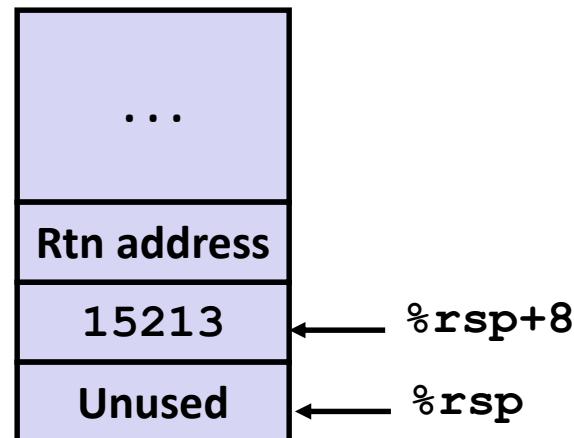
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Initial Stack Structure



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Resulting Stack Structure

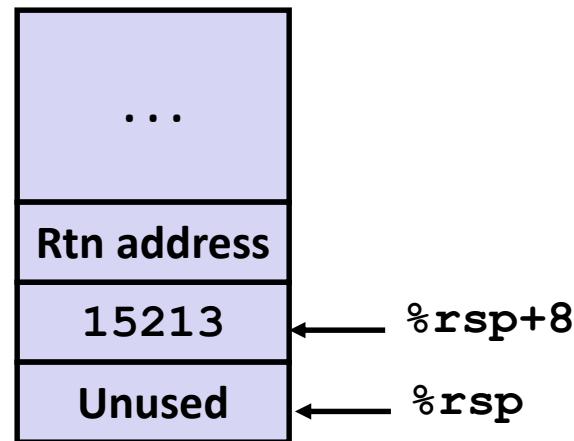


# Example: Calling incr #2

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



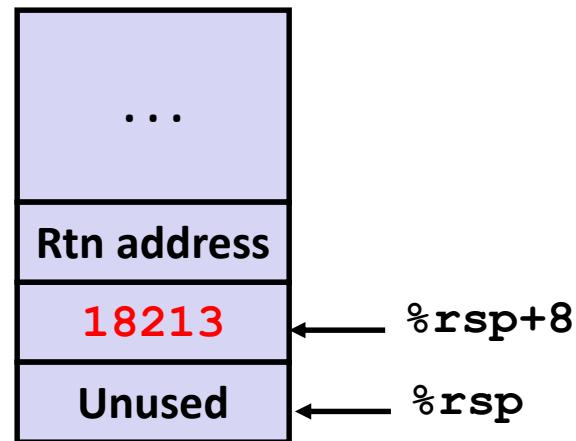
Register	Use(s)
%rdi	&v1
%rsi	3000

# Example: Calling incr #3

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure

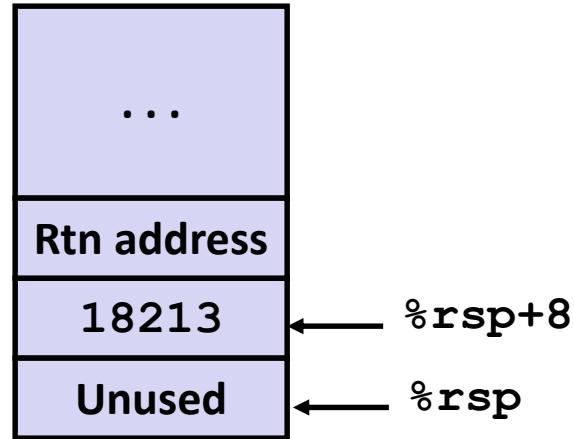


Register	Use(s)
%rdi	&v1
%rsi	3000

# Example: Calling incr #4

Stack Structure

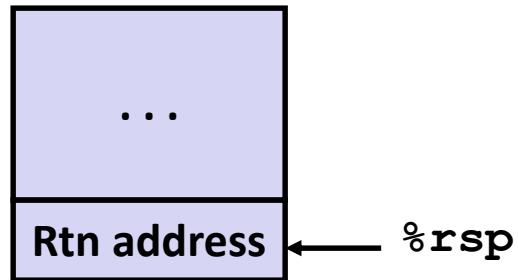
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Register	Use(s)
%rax	Return value

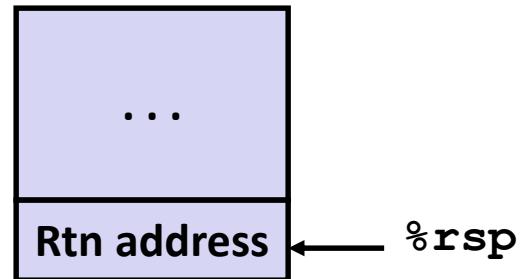
Updated Stack Structure



# Example: Calling incr #5

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

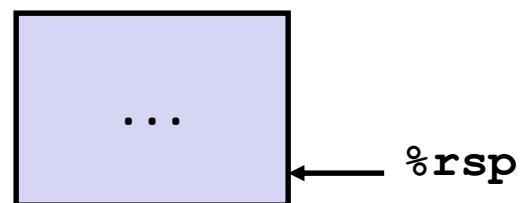
Updated Stack Structure



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Register	Use(s)
%rax	Return value

Final Stack Structure



# Register Saving Conventions

## ■ When procedure `yoo` calls `who`:

- `yoo` is the *caller*
- `who` is the *callee*

## ■ Can register be used for temporary storage?

```
yoo:
```

```
• • •  
    movq $15213, %rdx  
    call who  
    addq %rdx, %rax  
• • •  
    ret
```

```
who:
```

```
• • •  
    subq $18213, %rdx  
• • •  
    ret
```

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
  - Need some coordination

# Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the *caller*
  - **who** is the *callee*
- Can register be used for temporary storage?
- Conventions
  - “*Caller Saved*”
    - Caller saves temporary values in its frame before the call
  - “*Callee Saved*”
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

# x86-64 Linux Register Usage #1

## ■ %rax

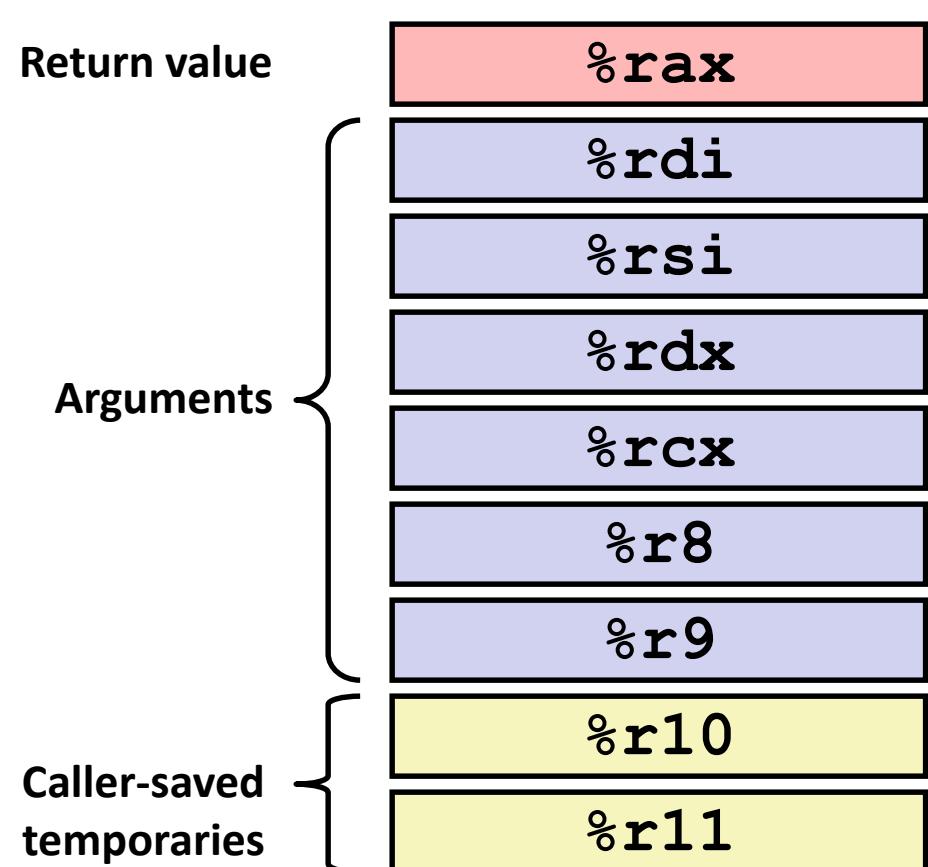
- Return value
- Also caller-saved
- Can be modified by procedure

## ■ %rdi, ..., %r9

- Arguments
- Also caller-saved
- Can be modified by procedure

## ■ %r10, %r11

- Caller-saved
- Can be modified by procedure



# x86-64 Linux Register Usage #2

## ■ **%rbx, %r12, %r13, %r14**

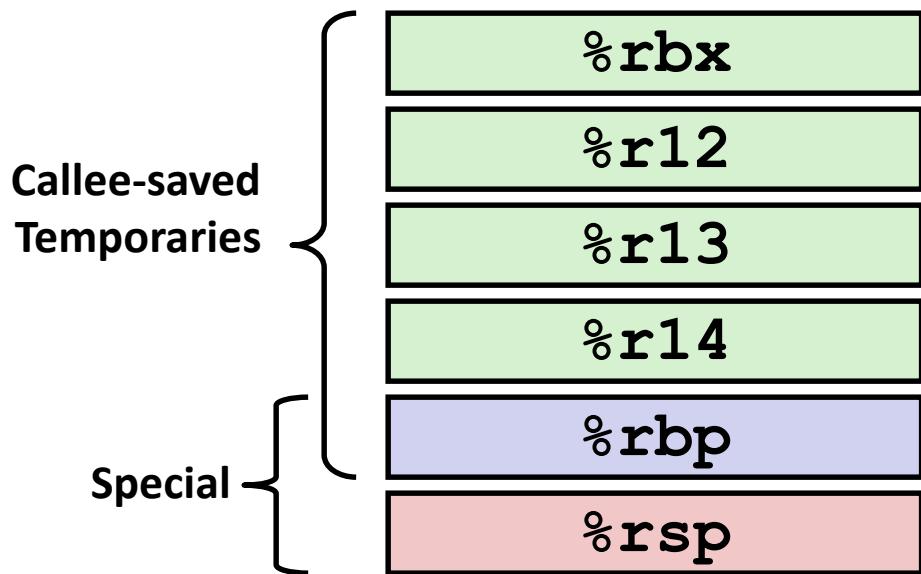
- Callee-saved
- Callee must save & restore

## ■ **%rbp**

- Callee-saved
- Callee must save & restore
- May be used as frame pointer
- Can mix & match

## ■ **%rsp**

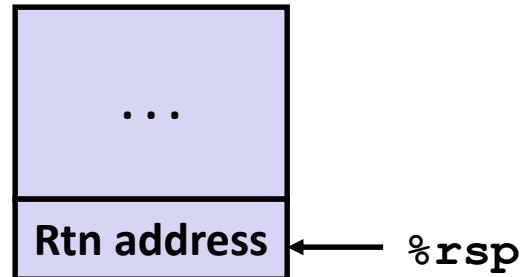
- Special form of callee save
- Restored to original value upon exit from procedure



# Callee-Saved Example #1

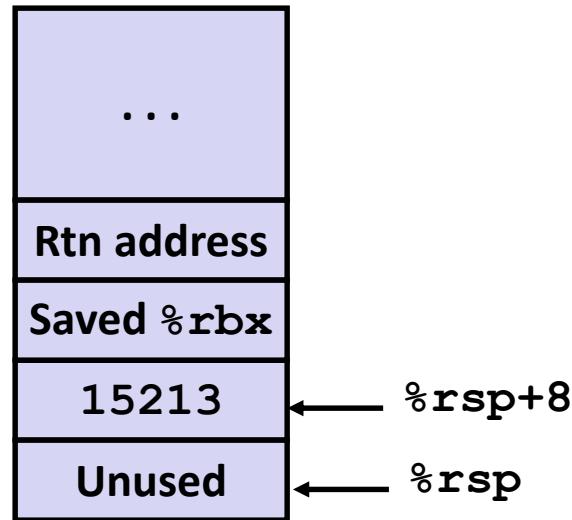
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

Initial Stack Structure



```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Resulting Stack Structure

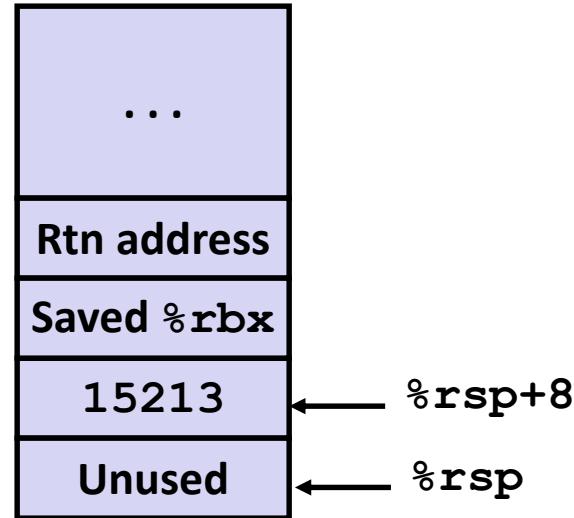


# Callee-Saved Example #2

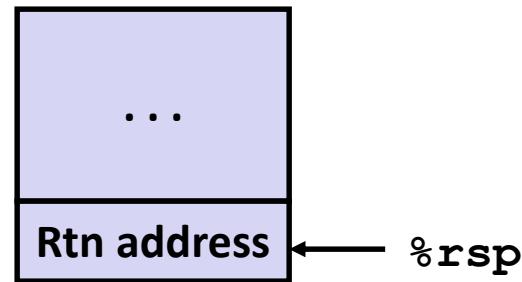
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
call_incr2:  
    pushq  %rbx  
    subq    $16, %rsp  
    movq    %rdi, %rbx  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    %rbx, %rax  
    addq    $16, %rsp  
    popq    %rbx  
    ret
```

Resulting Stack Structure



Pre-return Stack Structure



# Today

## ■ Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

# Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
.L6:
    rep; ret
```

# Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

# Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

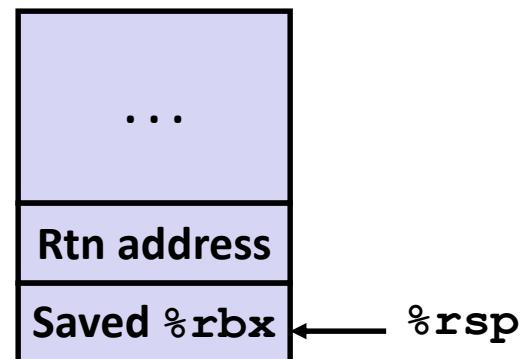
pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

```
    rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



# Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

```
    rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

# Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

rep; ret

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

# Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

```
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

# Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

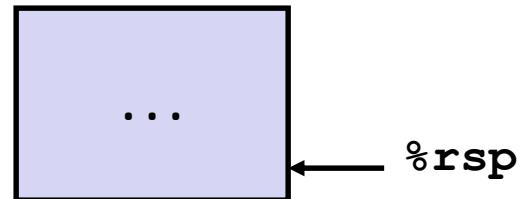
pcount\_r:

```
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call   pcount_r
    addq   %rbx, %rax
    popq   %rbx
```

.L6:

**rep; ret**

Register	Use(s)	Type
%rax	Return value	Return value



# Observations About Recursion

## ■ Handled Without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

## ■ Also works for mutual recursion

- P calls Q; Q calls P

# x86-64 Procedure Summary

## ■ Important Points

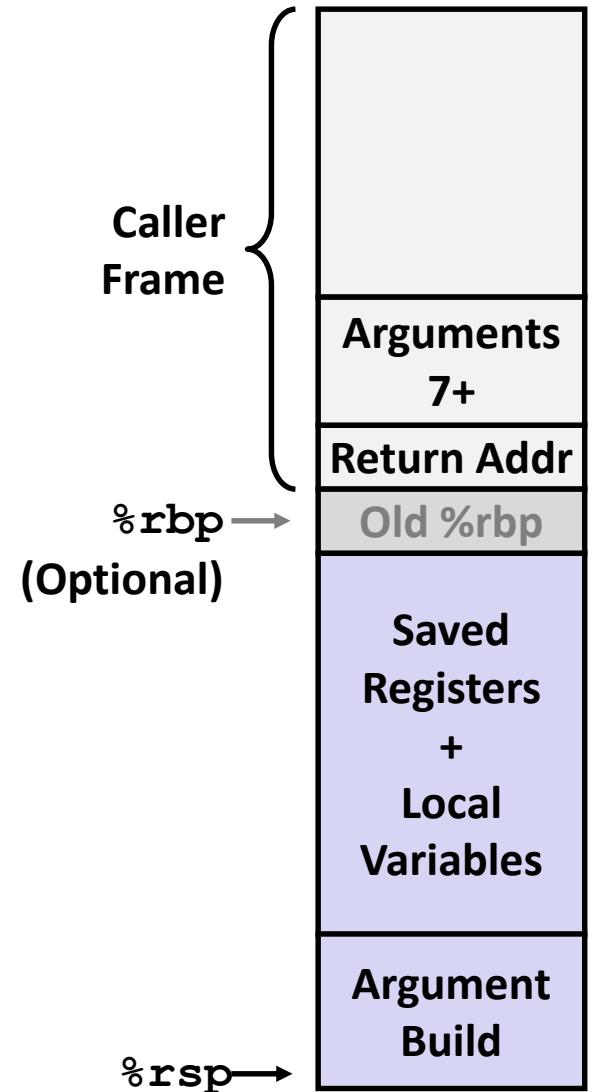
- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P

## ■ Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in **%rax**

## ■ Pointers are addresses of values

- On stack or global



# Machine-Level Programming IV: Data

CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu (Sections 1-2)

# Today

## ■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

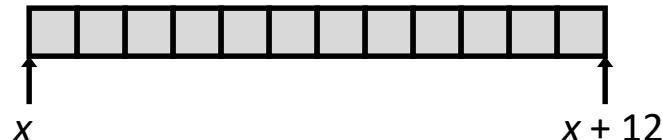
# Array Allocation

## ■ Basic Principle

$T \mathbf{A}[L];$

- Array of data type  $T$  and length  $L$
- Contiguously allocated region of  $L * \text{sizeof}(T)$  bytes in memory

```
char string[12];
```



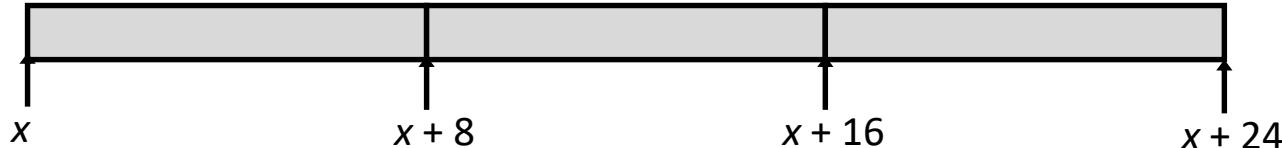
```
int val[5];
```



```
double a[3];
```



```
char *p[3];
```

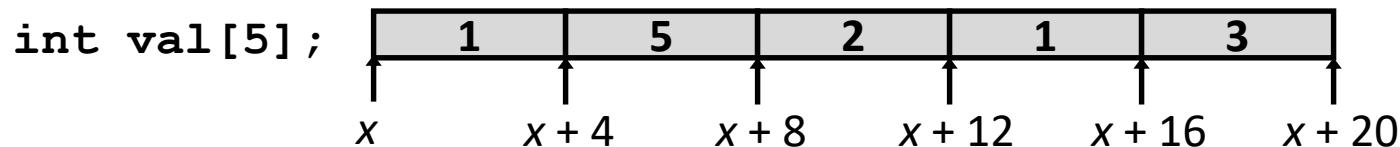


# Array Access

## ■ Basic Principle

$T \mathbf{A}[L]$  ;

- Array of data type  $T$  and length  $L$
- Identifier  $\mathbf{A}$  can be used as a pointer to array element 0: Type  $T^*$



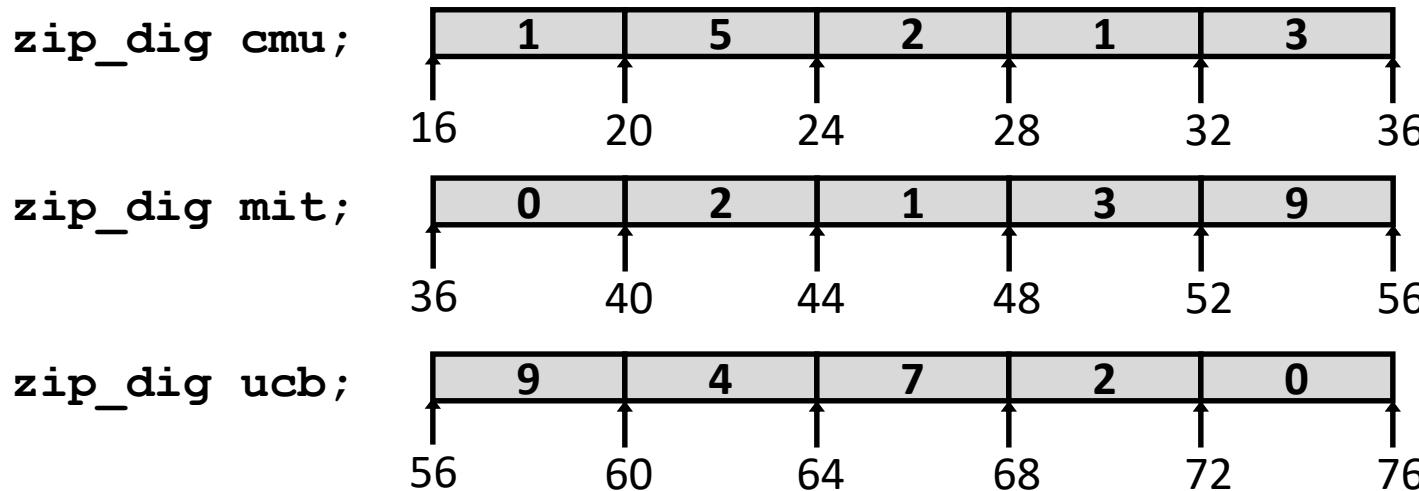
## ■ Reference      Type      Value

<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	$x$
<code>val+1</code>	<code>int *</code>	$x + 4$
<code>&amp;val[2]</code>	<code>int *</code>	$x + 8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x + 4i$

# Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

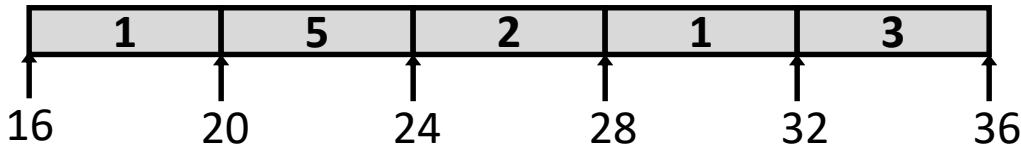
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

# Array Accessing Example

```
zip_dig cmu;
```



```
int get_digit
    (zip_dig z, int digit)
{
    return z[digit];
}
```

IA32

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at  $4 * \%rdi + \%rsi$
- Use memory reference `(%rdi,%rsi,4)`

# Array Loop Example

```
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# %rdi = z
movl    $0, %eax          # i = 0
jmp     .L3                # goto middle
.L4:               # loop:
    addl    $1, (%rdi,%rax,4) # z[i]++
    addq    $1, %rax          # i++
.L3:               # middle
    cmpq    $4, %rax          # i:4
    jbe     .L4                # if <=, goto loop
rep; ret
```

# Multidimensional (Nested) Arrays

## ■ Declaration

$T \ A[R][C];$

- 2D array of data type  $T$
- $R$  rows,  $C$  columns
- Type  $T$  element requires  $K$  bytes

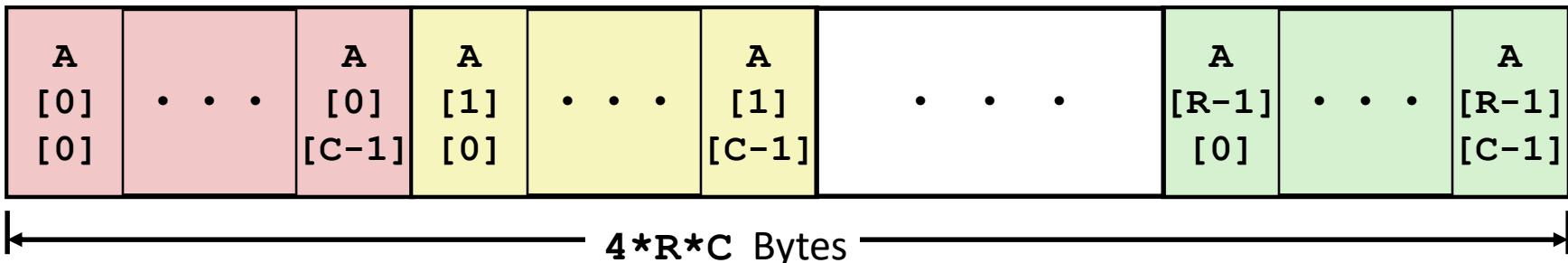
## ■ Array Size

- $R * C * K$  bytes

## ■ Arrangement

- Row-Major Ordering (in C/C++, Python, Mathematica, ....)
- Column-Major Ordering (in Fortran, Matlab, Octave, OpenGL, Julia, ....)

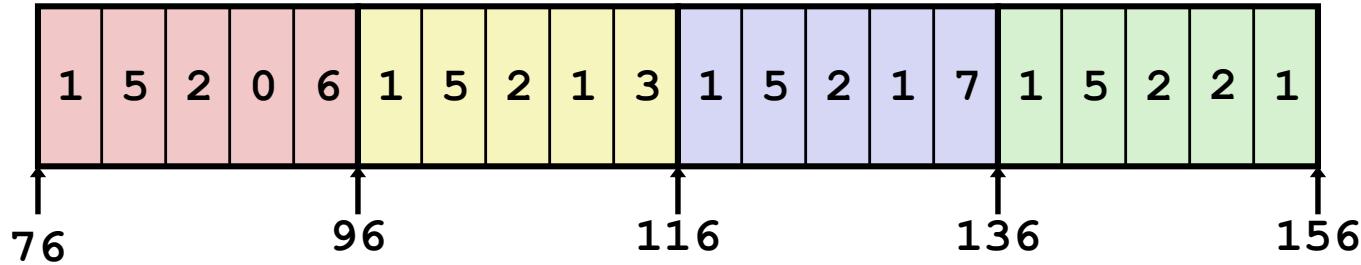
`int A[R][C];` – C: Row-Major



# Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
{{1, 5, 2, 0, 6},
 {1, 5, 2, 1, 3 },
 {1, 5, 2, 1, 7 },
 {1, 5, 2, 2, 1 }};
```

zip\_dig  
pgh[4];



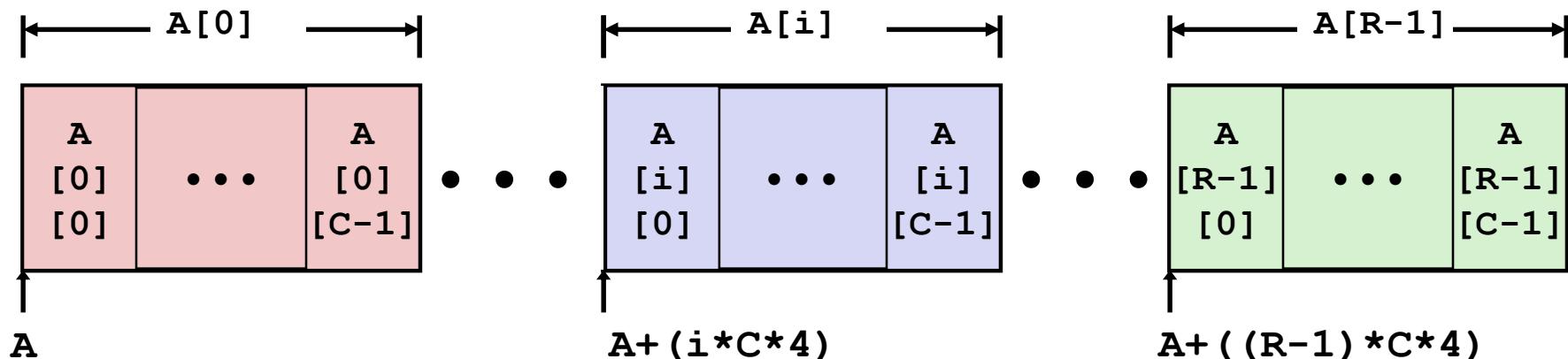
- “`zip_dig pgh [4]`” equivalent to “`int pgh [4] [5]`”
  - Variable `pgh`: array of 4 elements, allocated contiguously
  - Each element is an array of 5 `int`'s, allocated contiguously
- “Row-Major” (**C,C++**, ...) or “Column-Major” (**Fortran, Matlab, Julia** , ...) or “other” (others) ordering of all elements in memory

# Nested Array Row Access

## ■ Row Vectors

- $\mathbf{A}[i]$  is array of  $C$  elements
- Each element of type  $T$  requires  $K$  bytes
- Starting address  $\mathbf{A} + i * (C * K)$

```
int A[R][C];
```

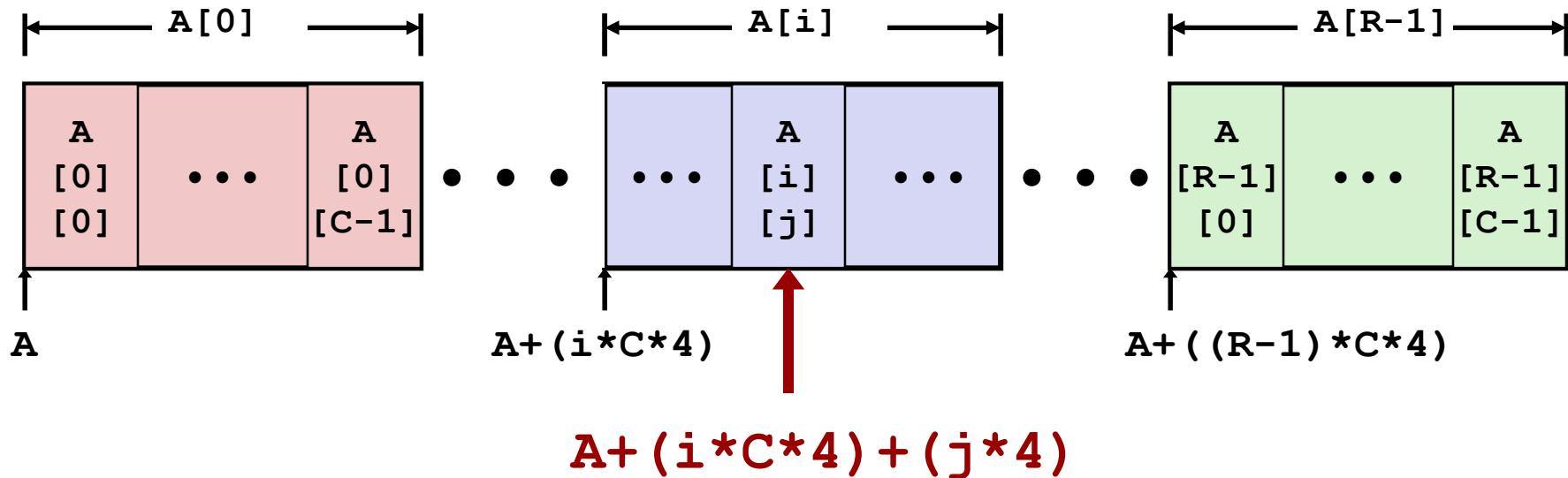


# Nested Array Element Access

## ■ Array Elements

- $A[i][j]$  is element of type  $T$ , which requires  $K$  bytes
- Address  $A + i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```

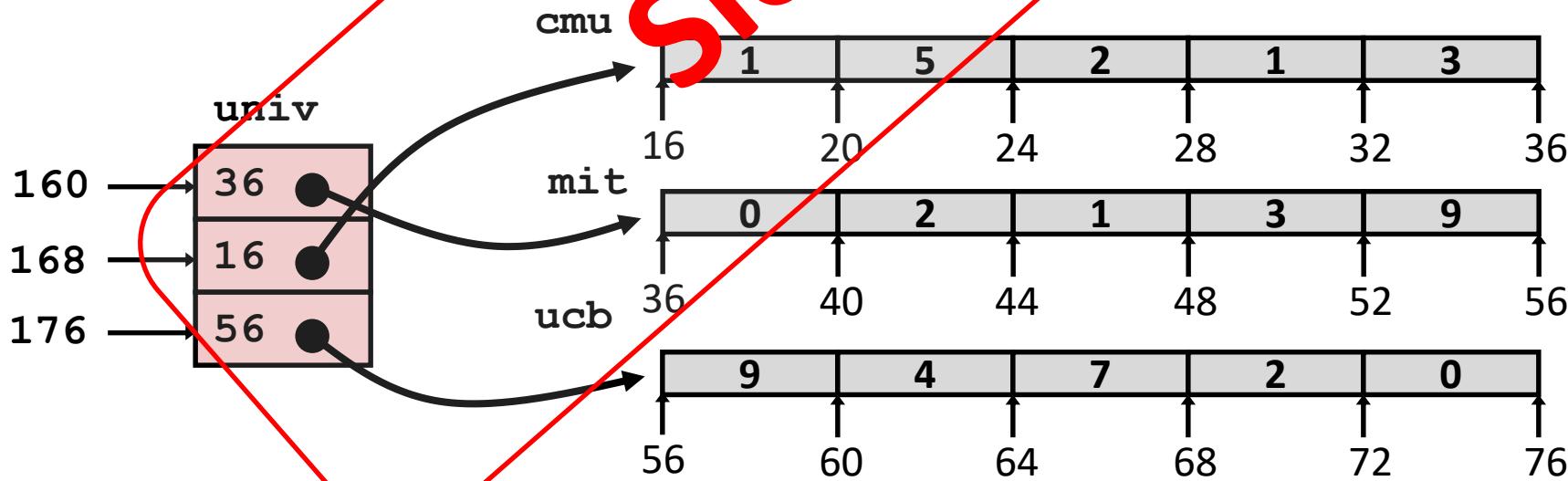


# Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig mit = { 0, 2, 1, 3, 9 };  
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3  
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s



# Element Access in Multi-Level Array

```
int get_univ_digit  
    (size_t index, size_t digit)  
{  
    return univ[index][digit];  
}
```

```
salq    $2, %rsi          # = digit  
addq    univ(%rdi,%rsi,8), %rsi # p = univ[index] + 4*digit  
movl    (%rsi), %eax        # = return *p  
ret
```

**SLOW**

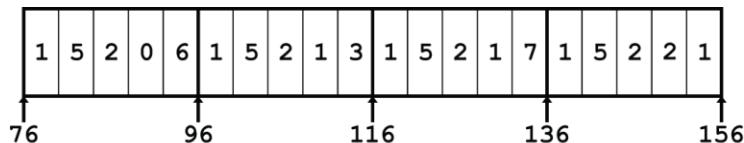
## ■ Computation

- Element access **Mem[Mem[univ+8\*index]+4\*digit]**
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

# Array Element Accesses

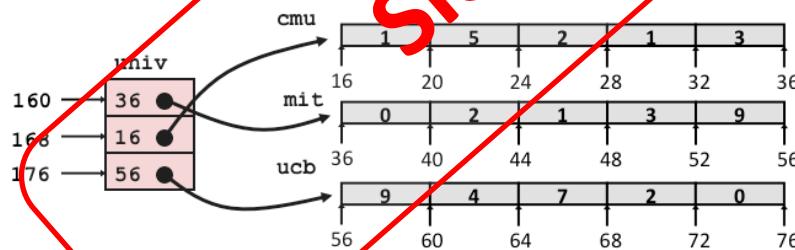
Nested array

```
int get_pgh_digit
    (size_t index, size_t digit)
{
    return pgh[index][digit];
}
```



Multi-level array

```
int get_univ_digit
    (size_t index, size_t digit)
{
    return univ[index][digit];
}
```



Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]`

`Mem[Mem[univ+8*index]+4*digit]`

# N X N Matrix

## Code

### ■ Fixed dimensions

- Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
    return a[i][j];
}
```

### ■ Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
    return a[IDX(n,i,j)];
}
```

### ■ Variable dimensions, implicit indexing

- Now supported by gcc

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
    return a[i][j];
}
```

# 16 X 16 Matrix Access

## ■ Array Elements

- Address  $\mathbf{A} + i * (C * K) + j * K$
- C = 16, K = 4

```
/* Get element a[i][j] */  
int fix_ele(fix_matrix a, size_t i, size_t j) {  
    return a[i][j];  
}
```

```
# a in %rdi, i in %rsi, j in %rdx  
salq    $6, %rsi          # 64*i  
addq    %rsi, %rdi        # a + 64*i  
movl    (%rdi,%rdx,4), %eax # M[a + 64*i + 4*j]  
ret
```

# $n \times n$ Matrix Access

## ■ Array Elements

- Address  $A + i * (C * K) + j * K$
- $C = n, K = 4$
- Must perform integer multiplication

```
/* Get element a[i][j] */  
int var_ele(size_t n, int a[n][n], size_t i, size_t j)  
{  
    return a[i][j];  
}
```

```
# n in %rdi, a in %rsi, i in %rdx, j in %rcx  
imulq    %rdx, %rdi          # n*i  
leaq     (%rsi,%rdi,4), %rax # a + 4*n*i  
movl     (%rax,%rcx,4), %eax # a + 4*n*i + 4*j  
ret
```

# Today

## ■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

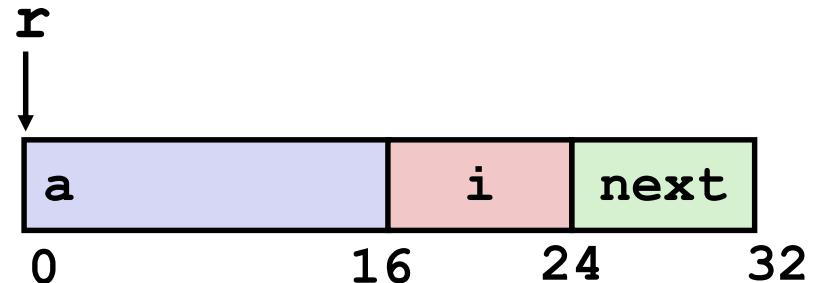
## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

# Structure Representation

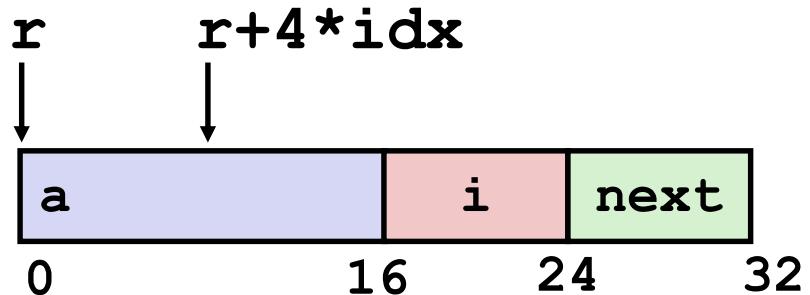
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

# Generating Pointer to Structure Member

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



## ■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as `r + 4*idx`

```
int *get_ap  
(struct rec *r, size_t idx)  
{  
    return &r->a[idx];  
}
```

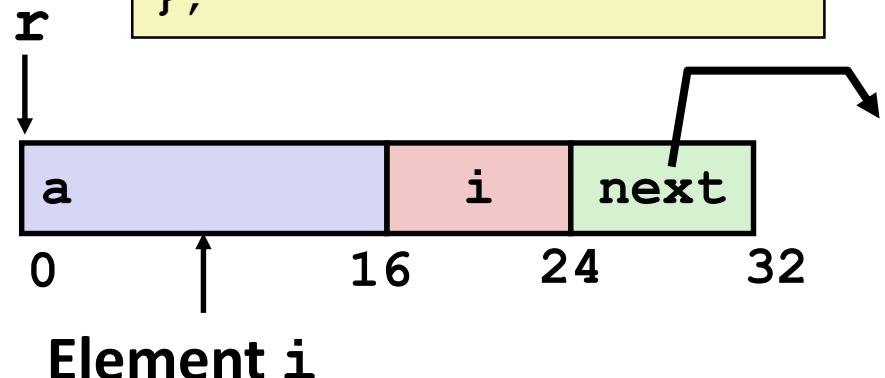
```
# r in %rdi, idx in %rsi  
leaq (%rdi,%rsi,4), %rax  
ret
```

# Following Linked List

## ■ C Code

```
void set_val
    (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

```
struct rec {
    int a[3];
    int i;
    struct rec *next;
};
```

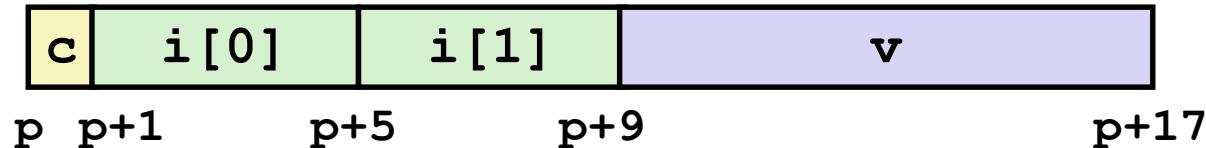


Register	Value
%rdi	<b>r</b>
%rsi	<b>val</b>

```
.L11:                      # loop:
    movslq  16(%rdi), %rax      #   i = M[r+16]
    movl    %esi, (%rdi,%rax,4) #   M[r+4*i] = val
    movq    24(%rdi), %rdi      #   r = M[r+24]
    testq   %rdi, %rdi         #   Test r
    jne     .L11                #   if !=0 goto loop
```

# Structures & Alignment

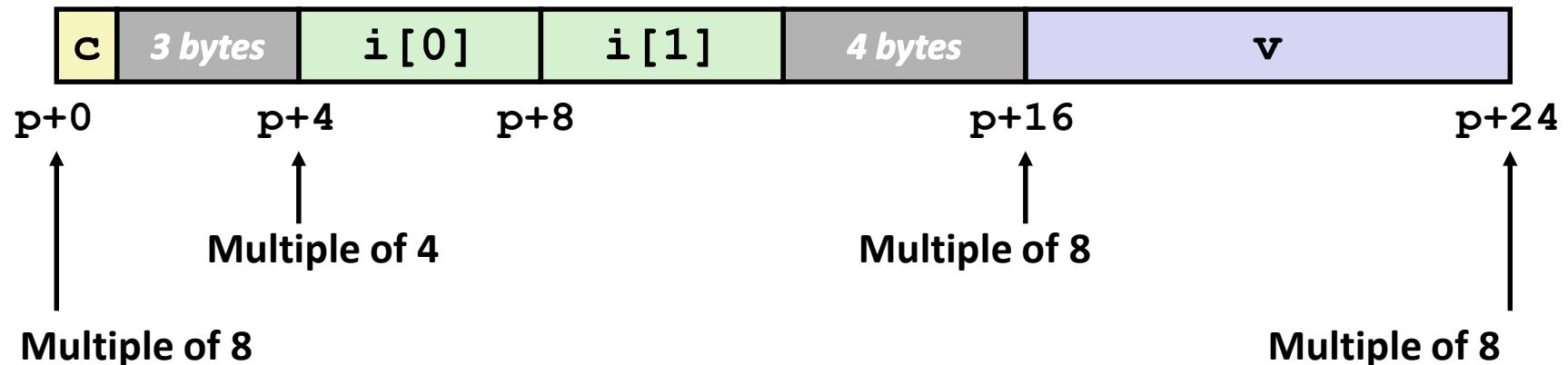
## ■ Unaligned Data



```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

## ■ Aligned Data

- Primitive data type requires  $K$  bytes
- Address must be multiple of  $K$



# Alignment Principles

## ■ Aligned Data

- Primitive data type requires  $K$  bytes
- Address must be multiple of  $K$
- Required on some machines; advised on x86-64

## ■ Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

## ■ Compiler

- Inserts gaps in structure to ensure correct alignment of fields

# Specific Cases of Alignment (x86-64)

- **1 byte: `char`, ...**
  - no restrictions on address
- **2 bytes: `short`, ...**
  - lowest 1 bit of address must be  $0_2$
- **4 bytes: `int`, `float`, ...**
  - lowest 2 bits of address must be  $00_2$
- **8 bytes: `double`, `long`, `char *`, ...**
  - lowest 3 bits of address must be  $000_2$
- **16 bytes: `long double` (GCC on Linux)**
  - lowest 4 bits of address must be  $0000_2$

# Satisfying Alignment with Structures

## ■ Within structure:

- Must satisfy each element's alignment requirement

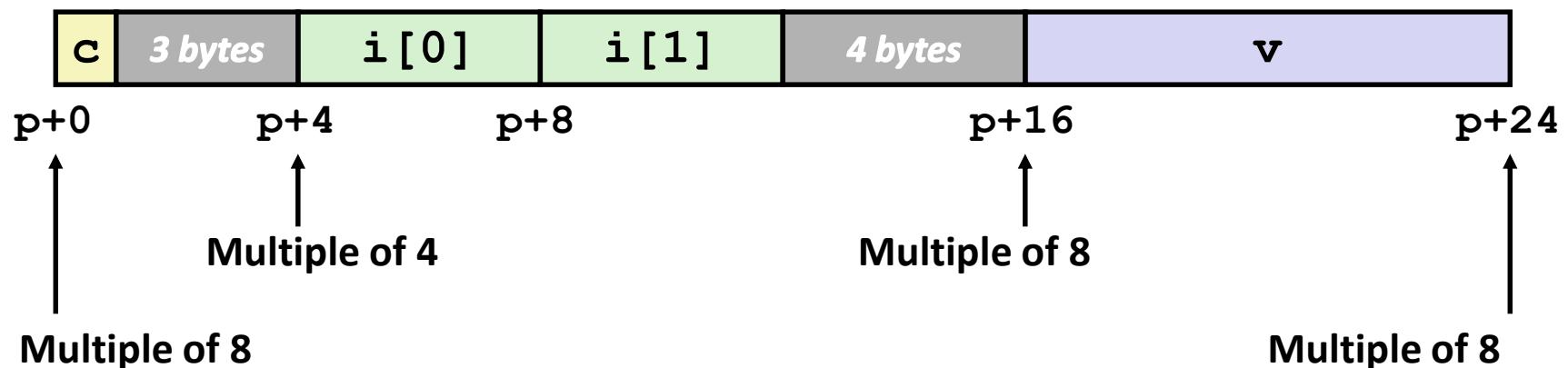
## ■ Overall structure placement

- Each structure has alignment requirement K
  - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

## ■ Example:

- K = 8, due to **double** element

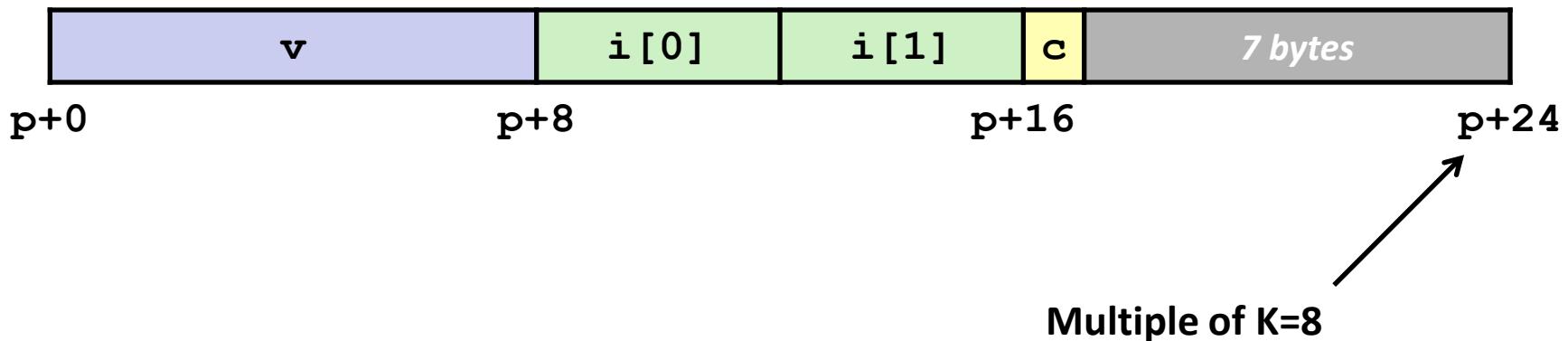
```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```



# Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

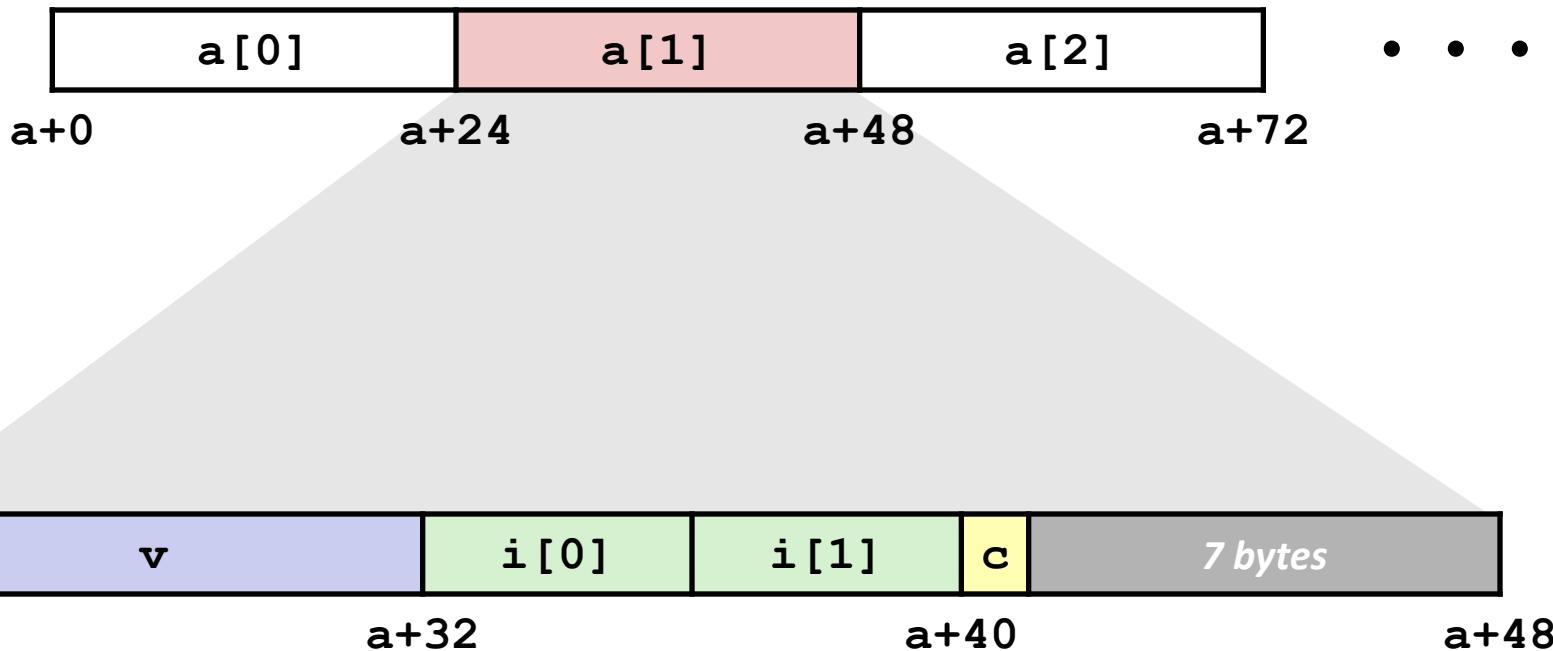
```
struct s2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



# Arrays of Structures

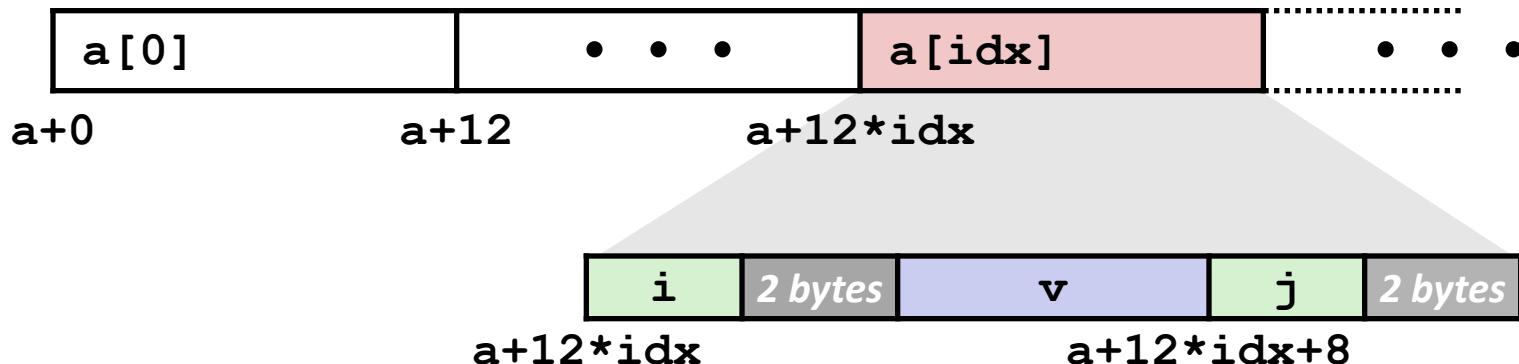
- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



# Accessing Array Elements

- Compute array offset  $12 * \text{idx}$ 
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`
  - Resolved during linking



```
short get_j(int idx)
{
    return a[idx].j;
}
```

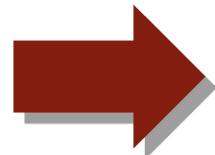
```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(%rax,4),%eax
```

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

# Saving Space

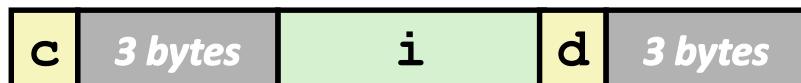
- Put large data types first

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

- Effect (K=4)



# Today

## ■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

# Background

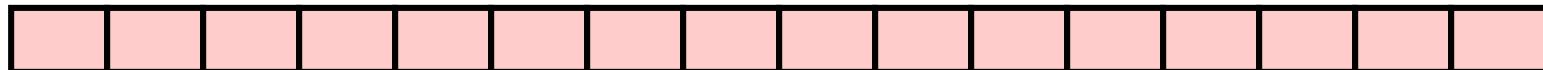
## ■ History

- x87 FP
  - Legacy, very ugly
- SSE FP
  - Special case use of vector instructions
- AVX FP
  - Newest version
  - Similar to SSE
  - Documented in book

# Programming with SSE3

## XMM Registers

- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers



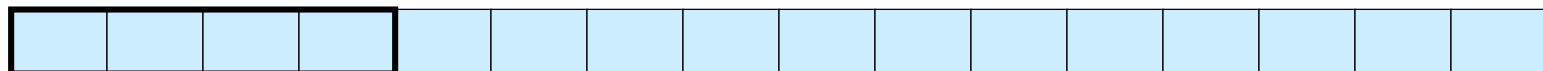
- 4 single-precision floats



- 2 double-precision floats



- 1 single-precision float



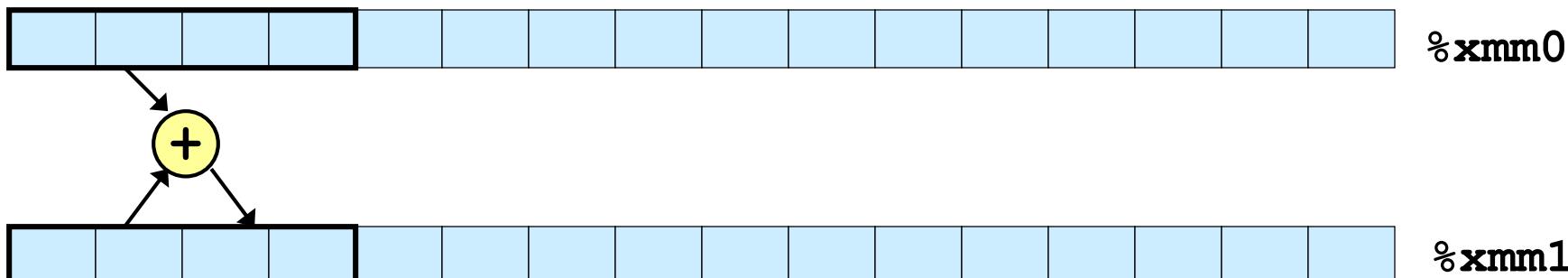
- 1 double-precision float



# Scalar & SIMD Operations

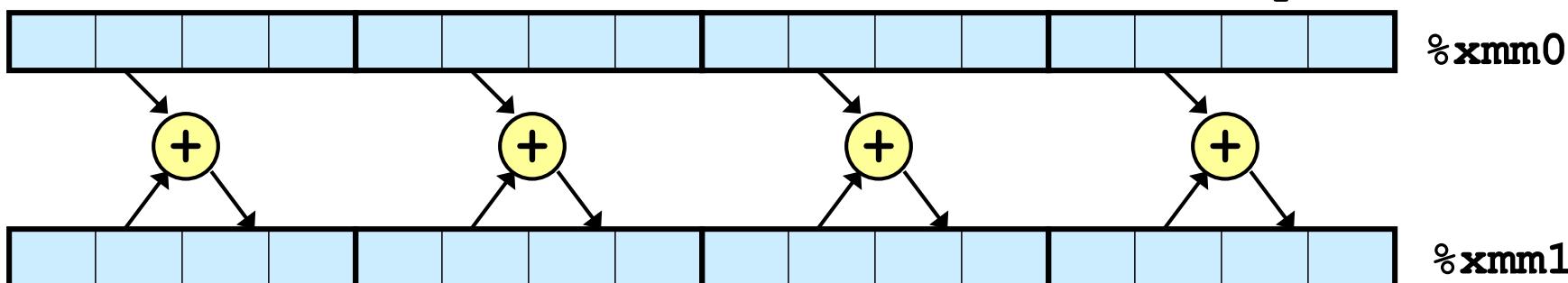
## ■ Scalar Operations: Single Precision

**addss** %xmm0 , %xmm1



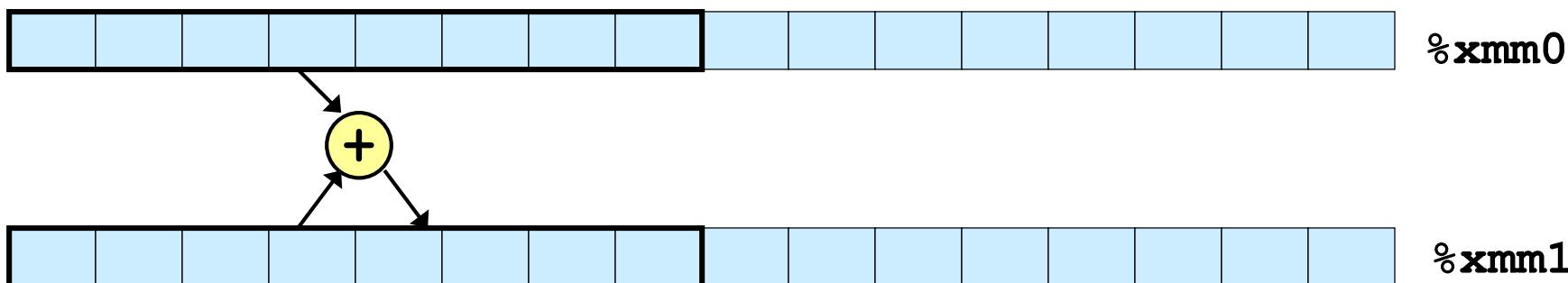
## ■ SIMD Operations: Single Precision

**addps** %xmm0 , %xmm1



## ■ Scalar Operations: Double Precision

**addsd** %xmm0 , %xmm1



# FP Basics

- Arguments passed in %xmm0, %xmm1, ...
- Result returned in %xmm0
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
addss    %xmm1, %xmm0
ret
```

```
# x in %xmm0, y in %xmm1
addsd    %xmm1, %xmm0
ret
```

# FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd  %xmm0, %xmm1    # Copy v
movsd   (%rdi), %xmm0  # x = *p
addsd   %xmm0, %xmm1    # t = x + v
movsd   %xmm1, (%rdi)  # *p = t
ret
```

# Other Aspects of FP Code

## ■ *Lots of instructions*

- Different operations, different formats, ...

## ■ Floating-point comparisons

- Instructions **ucomiss** and **ucomisd**
- Set condition codes CF, ZF, and PF

## ■ Using constant values

- Set XMM0 register to 0 with instruction **xorpd %xmm0, %xmm0**
- Others loaded from memory

# Summary

## ■ Arrays

- Elements packed into contiguous region of memory
- Use index arithmetic to locate individual elements

## ■ Structures

- Elements packed into single region of memory
- Access using offsets determined by compiler
- Possible require internal and external padding to ensure alignment

## ■ Combinations

- Can nest structure and array code arbitrarily

## ■ Floating Point

- Data held and operated on in XMM registers

# Machine-Level Programming V: Advanced Topics

CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

# x86-64 Linux Memory Layout

*not drawn to scale*

## ■ Stack

- Runtime stack (8MB limit)
- E. g., local variables

## ■ Heap

- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new()`

## ■ Data

- Statically allocated data
- E.g., global vars, `static` vars, string constants

## ■ Text / Shared Libraries

- Executable machine instructions
- Read-only

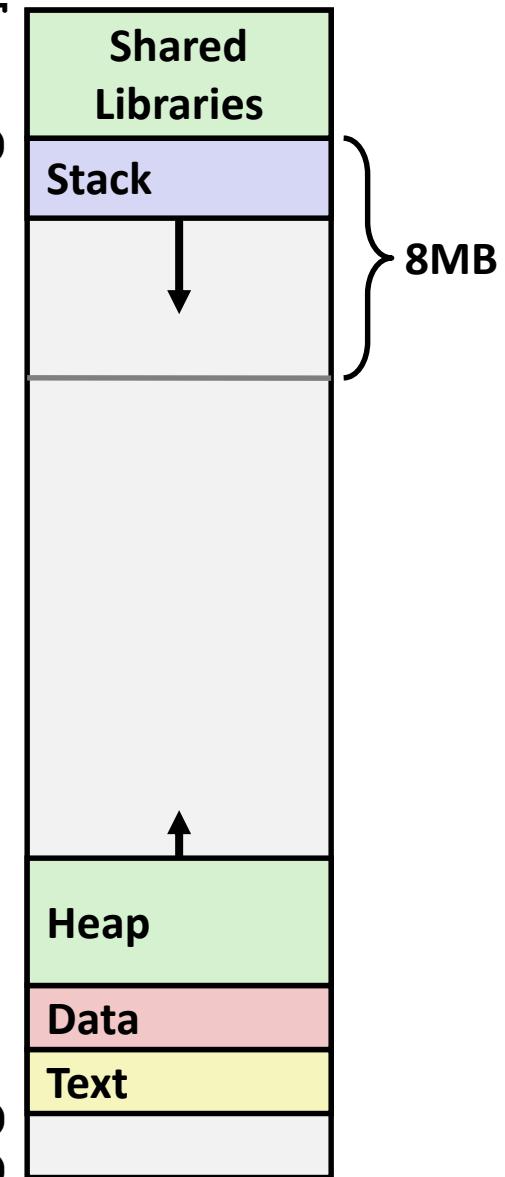
Hex Address



400000  
000000

00007FFFFFFFFF

00007FFF00000000



*not drawn to scale*

# Memory Allocation Example

00007FFFFFFFFF

```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```



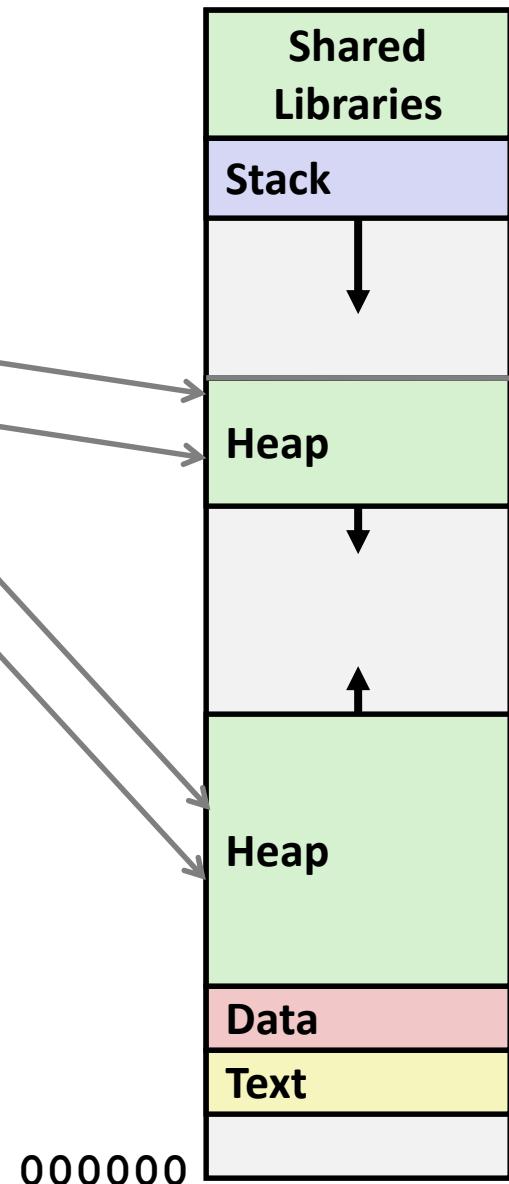
*Where does everything go?*

*not drawn to scale*

# x86-64 Example Addresses

*address range ~ $2^{47}$*

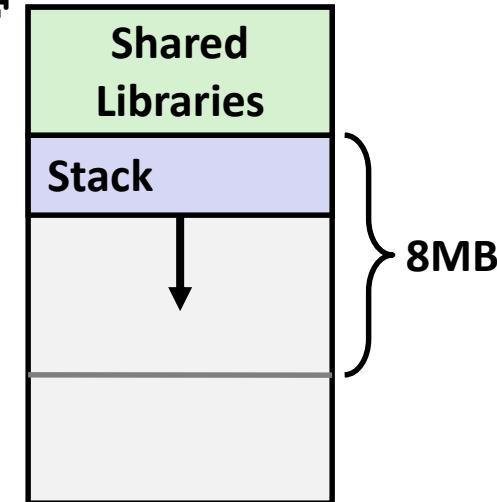
local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



# Runaway Stack Example

*not drawn to scale*

```
00007FFFFFFFFF  
int recurse(int x) {  
    int a[1<<15]; // 4*2^15 = 128 KiB  
    printf("x = %d. a at %p\n", x, a);  
    a[0] = (1<<14)-1;  
    a[a[0]] = x-1;  
    if (a[a[0]] == 0)  
        return -1;  
    return recurse(a[a[0]]) - 1;  
}
```



- Functions store local data on in stack frame
- Recursive functions cause deep nesting of frames

```
./runaway 67  
x = 67. a at 0x7ffd18aba930  
x = 66. a at 0x7ffd18a9a920  
x = 65. a at 0x7ffd18a7a910  
x = 64. a at 0x7ffd18a5a900  
. . .  
x = 4. a at 0x7ffd182da540  
x = 3. a at 0x7ffd182ba530  
x = 2. a at 0x7ffd1829a520  
Segmentation fault (core dumped)
```

# Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

# Recall: Memory Referencing Bug Example

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

```
fun(0)    -> 3.1400000000
fun(1)    -> 3.1400000000
fun(2)    -> 3.1399998665
fun(3)    -> 2.0000006104
fun(6)    -> Stack smashing detected
fun(8)    -> Segmentation fault
```

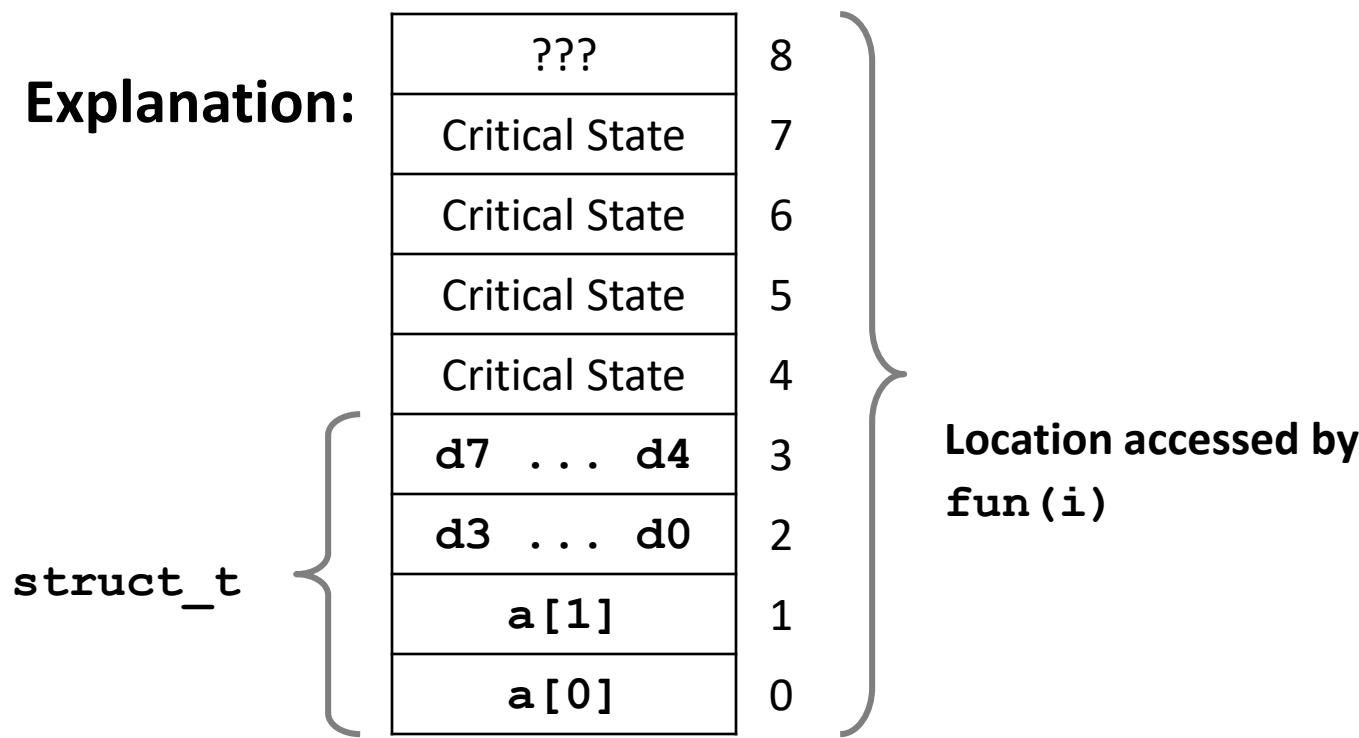
- Result is system specific

# Memory Referencing Bug Example

```
typedef struct {  
    int a[2];  
    double d;  
} struct_t;
```

fun(0)	->	3.1400000000
fun(1)	->	3.1400000000
fun(2)	->	3.1399998665
fun(3)	->	2.0000006104
fun(4)	->	Segmentation fault
fun(8)	->	3.1400000000

Explanation:



# **Such problems are a BIG deal**

- **Generally called a “buffer overflow”**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

# String Library Code

## ■ Implementation of Unix function gets ()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- No way to specify limit on number of characters to read
- **Similar problems with other library functions**
  - **strcpy, strcat**: Copy strings of arbitrary length
  - **scanf, fscanf, sscanf**, when given %s conversion specification

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

← btw, how big  
is big enough?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

```
unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Segmentation Fault
```

# Buffer Overflow Disassembly

## echo:

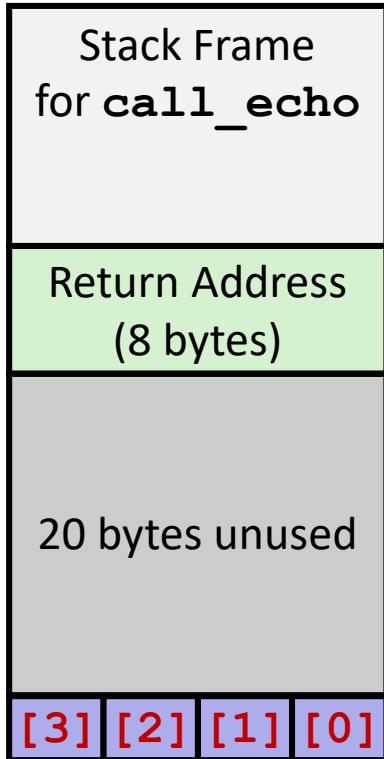
```
00000000004006cf <echo>:  
 4006cf: 48 83 ec 18          sub    $0x18,%rsp  
 4006d3: 48 89 e7          mov    %rsp,%rdi  
 4006d6: e8 a5 ff ff ff      callq  400680 <gets>  
 4006db: 48 89 e7          mov    %rsp,%rdi  
 4006de: e8 3d fe ff ff      callq  400520 <puts@plt>  
 4006e3: 48 83 c4 18          add    $0x18,%rsp  
 4006e7: c3                  retq
```

## call\_echo:

```
4006e8: 48 83 ec 08          sub    $0x8,%rsp  
4006ec: b8 00 00 00 00      mov    $0x0,%eax  
4006f1: e8 d9 ff ff ff      callq  4006cf <echo>  
4006f6: 48 83 c4 08          add    $0x8,%rsp  
4006fa: c3                  retq
```

# Buffer Overflow Stack

*Before call to gets*

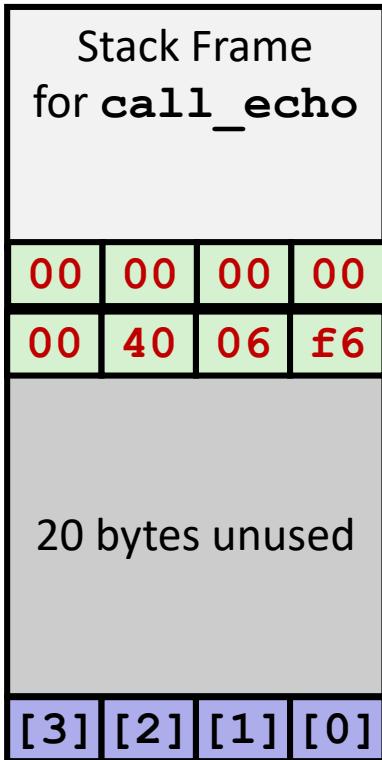


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    . . .
```

# Buffer Overflow Stack Example

*Before call to gets*



```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $x18, %rsp
    movq %rsp, %rdi
    call gets
    ...

```

`call_echo:`

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

# Buffer Overflow Stack Example #1

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    ...

```

`call_echo:`

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

`buf ← %rsp`

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

“01234567890123456789012\0”

**Overflowed buffer, but did not corrupt state**

# Buffer Overflow Stack Example #2

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

```

`call_echo:`

```
...
4006f1: callq 4006cf <echo>
4006f6: add    $0x8,%rsp
...
```

`buf ← %rsp`

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
Segmentation fault
```

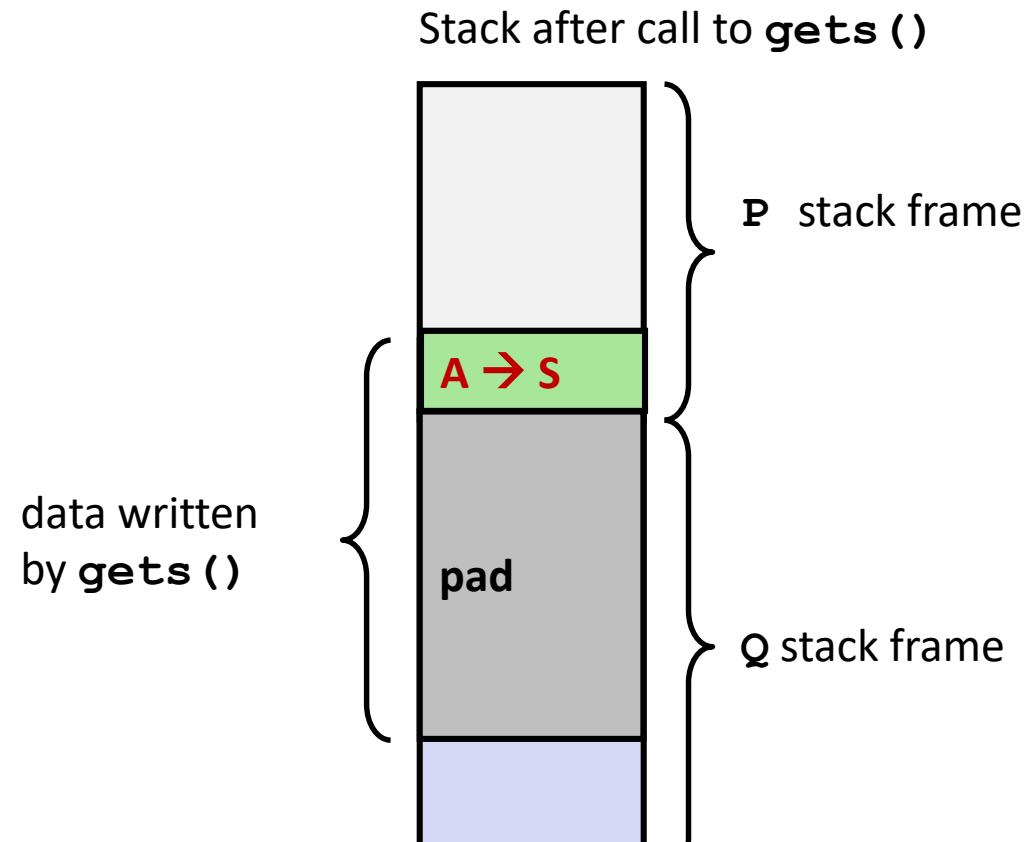
Program “returned” to 0x0400600, and then crashed.

# Stack Smashing Attacks

```
void P() {  
    Q();  
    ...  
}  
A
```

```
int Q() {  
    char buf[64];  
    gets(buf);  
    ...  
    return ...;  
}
```

```
void S() {  
    /* Something  
       unexpected */  
    ...  
}
```



- Overwrite normal return address A with address of some other code S
- When Q executes `ret`, will jump to other code

# Crafting Smashing String

Stack Frame for call echo			
00	00	00	00
00	48	83	80
00	00	00	00
00	40	06	fb

```
int echo() {  
    char buf[4];  
    gets(buf);  
    ...  
    return ...;  
}
```

← %rsp

24 bytes

*Target Code*

```
void smash() {  
    printf("I've been smashed!\n");  
    exit(0);  
}
```

00000000004006fb <smash>:

4006fb: 48 83 ec 08

*Attack String (Hex)*

30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33  
fb 06 40 00 00 00 00 00 00

# Smashing String Effect

Stack Frame for call echo			
00	00	00	00
00	48	83	80
00	00	00	00
00	40	06	fb
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

← %rsp

## Target Code

```
void smash() {  
    printf("I've been smashed!\n");  
    exit(0);  
}
```

00000000004006fb <smash>:

4006fb: 48 83 ec 08

## Attack String (Hex)

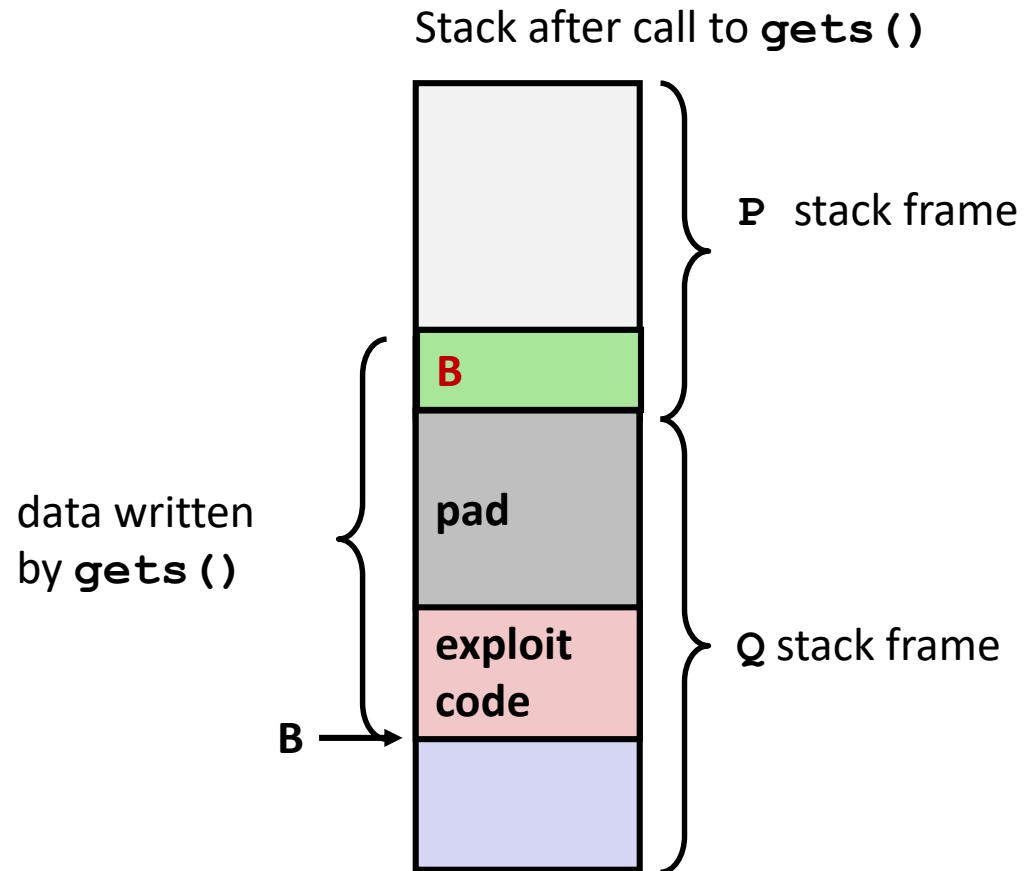
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33  
fb 06 40 00 00 00 00 00 00

# Code Injection Attacks

```
void P() {  
    Q();  
    ...  
}
```

return address  
**A**

```
int Q() {  
    char buf[64];  
    gets(buf);  
    ...  
    return ...;  
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

# How Does The Attack Code Execute?

```
void P() {  
    Q();  
    ...  
}
```

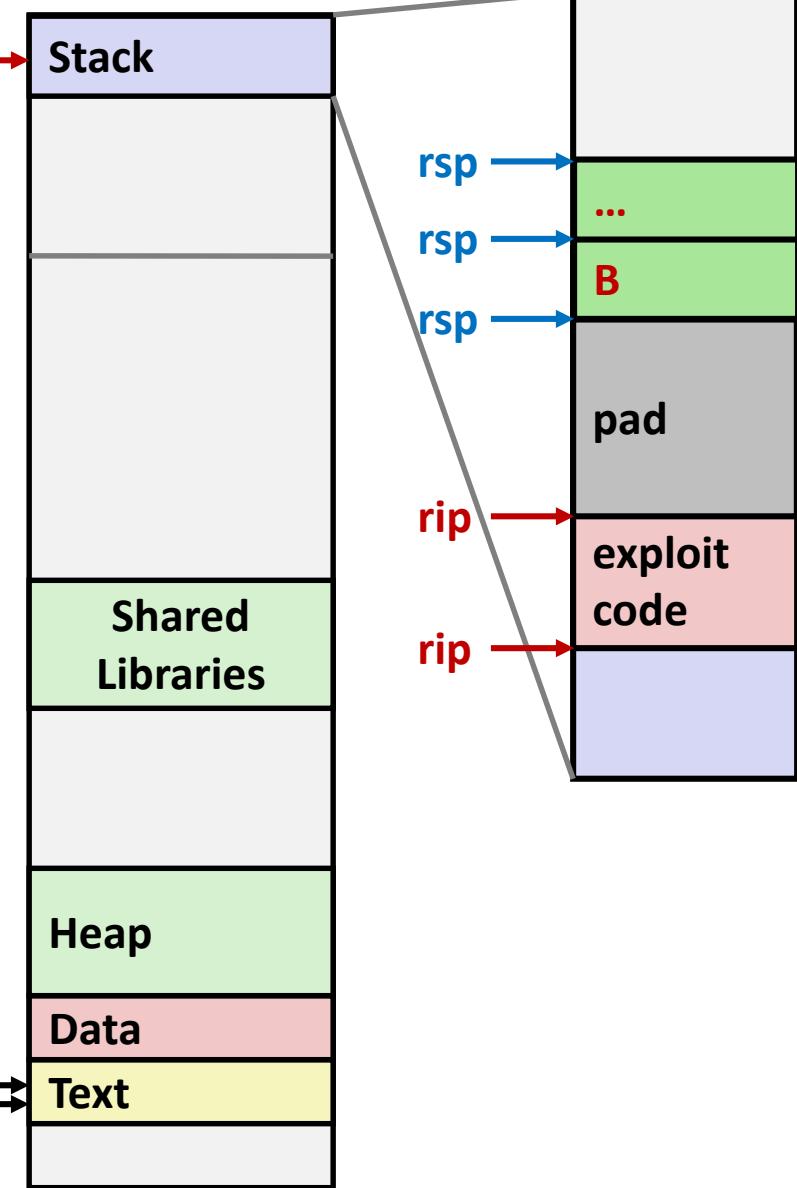
```
int Q() {  
    char buf[64];  
    gets(buf); // A->B  
    ...  
    return ...;  
}
```

ret

ret

rip

rip



# What To Do About Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each...

# 1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- For example, use library routines that limit string lengths
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use **%ns** where **n** is a suitable integer

# 2. System-Level Protections can help

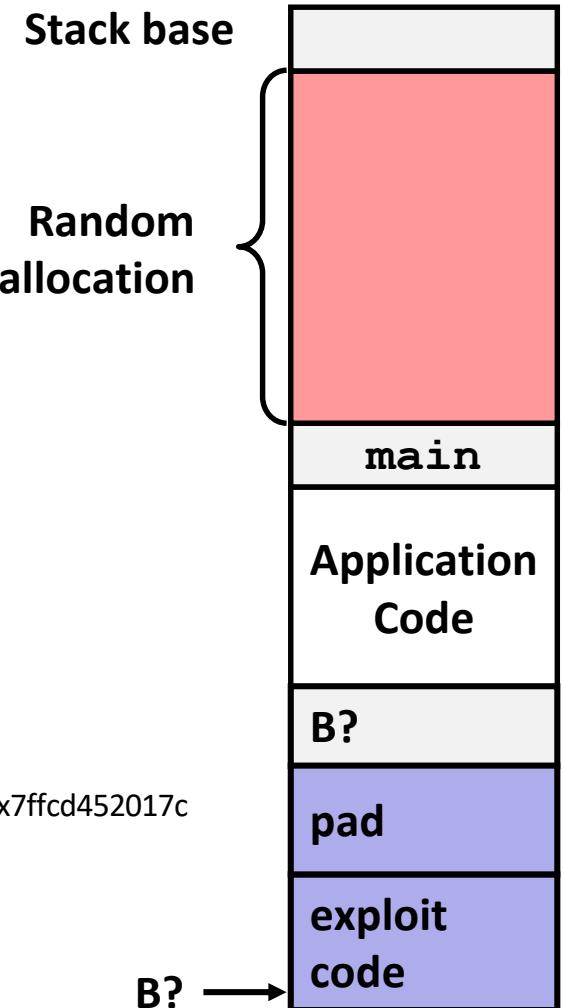
## ■ Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Shifts stack addresses for entire program
- Makes it difficult for hacker to predict beginning of inserted code
- E.g.: 5 executions of memory allocation code

local

0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

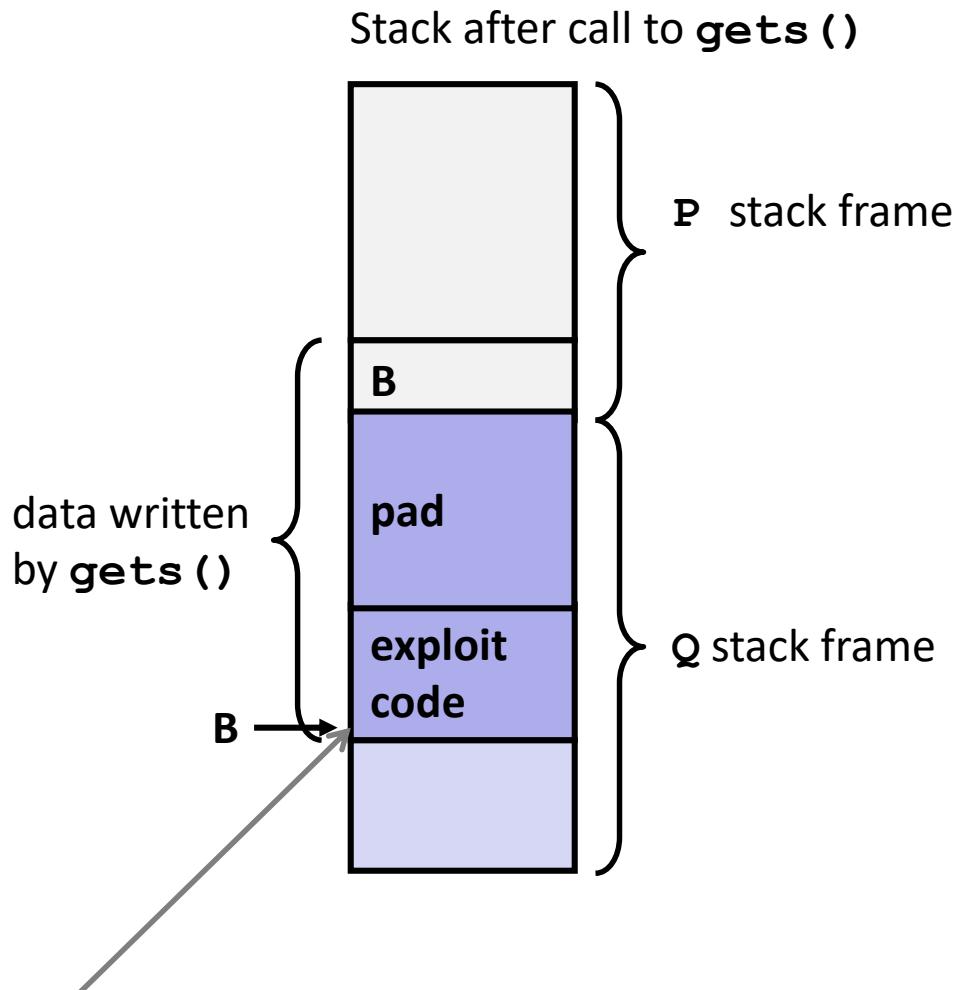
- Stack repositioned each time program executes



# 2. System-Level Protections can help

## ■ Nonexecutable code segments

- In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- x86-64 added explicit “execute” permission
- Stack marked as non-executable



Any attempt to execute this code will fail

# 3. Stack Canaries can help

## ■ Idea

- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

## ■ GCC Implementation

- **-fstack-protector**
- Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string: 0123456
0123456
```

```
unix>./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```

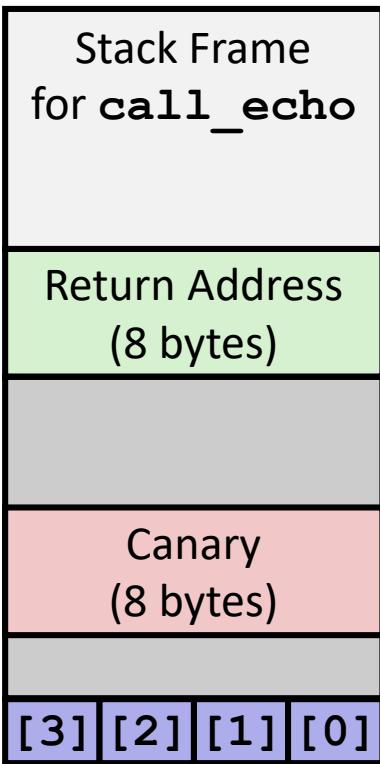
# Protected Buffer Disassembly

echo:

```
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: mov    0x8(%rsp),%rax
400758: xor    %fs:0x28,%rax
400761: je     400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq
```

# Setting Up Canary

*Before call to gets*

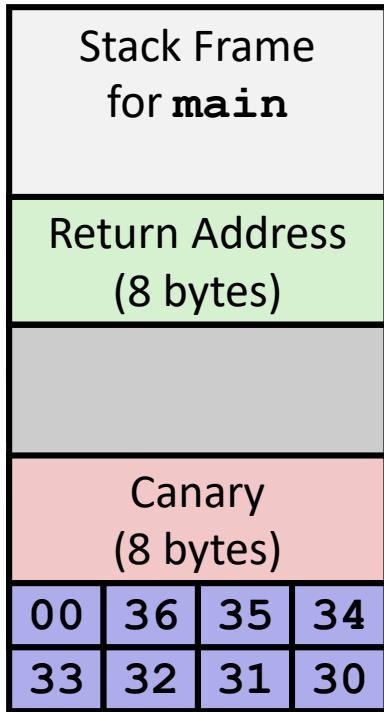


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax   # Erase canary
    . . .
```

# Checking Canary

*After call to gets*



```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

**Input: 0123456**

`buf ← %rsp`

```
echo:
. . .
movq    8(%rsp), %rax      # Retrieve from stack
xorq    %fs:40, %rax       # Compare to canary
je      .L6                  # If same, OK
call    __stack_chk_fail   # FAIL
```

# Return-Oriented Programming Attacks

## ■ Challenge (for hackers)

- Stack randomization makes it hard to predict buffer location
- Marking stack nonexecutable makes it hard to insert binary code

## ■ Alternative Strategy

- Use existing code
  - E.g., library code from stdlib
- String together fragments to achieve overall desired outcome
- *Does not overcome stack canaries*

## ■ Construct program from *gadgets*

- Sequence of instructions ending in `ret`
  - Encoded by single byte `0xc3`
- Code positions fixed from run to run
- Code is executable

# Gadget Example #1

```
long ab_plus_c  
  (long a, long b, long c)  
{  
    return a*b + c;  
}
```

```
00000000004004d0 <ab_plus_c>:  
 4004d0: 48 0f af fe  imul %rsi,%rdi  
 4004d4: 48 8d 04 17  lea  (%rdi,%rdx,1),%rax  
 4004d8: c3          retq
```

$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

# Gadget Example #2

```
void setval(unsigned *p) {  
    *p = 3347663060u;  
}
```

```
<setval>:  
4004d9: c7 07 d4 48 89 c7    movl $0xc78948d4, (%rdi)  
4004df: c3                      retq
```

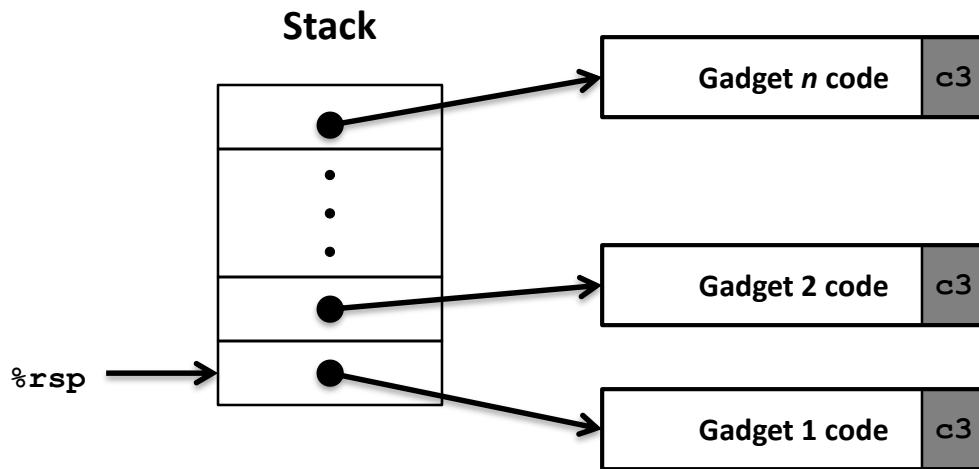
Encodes `movq %rax, %rdi`

`rdi ← rax`

Gadget address = `0x4004dc`

## ■ Repurpose byte codes

# ROP Execution



- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one

# Crafting an ROB Attack String

Stack Frame for <code>call echo</code>			
00	00	00	00
00	48	83	80
00	00	00	00
00	40	06	f6
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf

## Gadget

```
00000000004004d0 <ab_plus_c>:  
4004d0: 48 0f af fe imul %rsi,%rdi  
4004d4: 48 8d 04 17 lea (%rdi,%rdx,1),%rax  
4004d8: c3 retq
```

`rax ← rdi + rdx`

Attack: `int echo() returns rdi + rdx`

```
int echo() {  
    char buf[4];  
    gets(buf);  
    ...  
    return ...;  
}
```

## Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33  
d4 04 40 00 00 00 00 00 00 00
```

Multiple gadgets will corrupt stack upwards

# **Summary**

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Code Injection Attack
  - Return Oriented Programming

# Exploits Based on Buffer Overflows

- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more
- You will learn some of the tricks in attacklab
  - Hopefully to convince you to never leave such holes in your programs!!

# Example: the original Internet worm (1988)

## ■ Exploited a few vulnerabilities to spread

- Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
  - `finger droh@cs.cmu.edu`
- Worm attacked fingerd server by sending phony argument:
  - `finger "exploit-code padding new-return-address"`
  - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

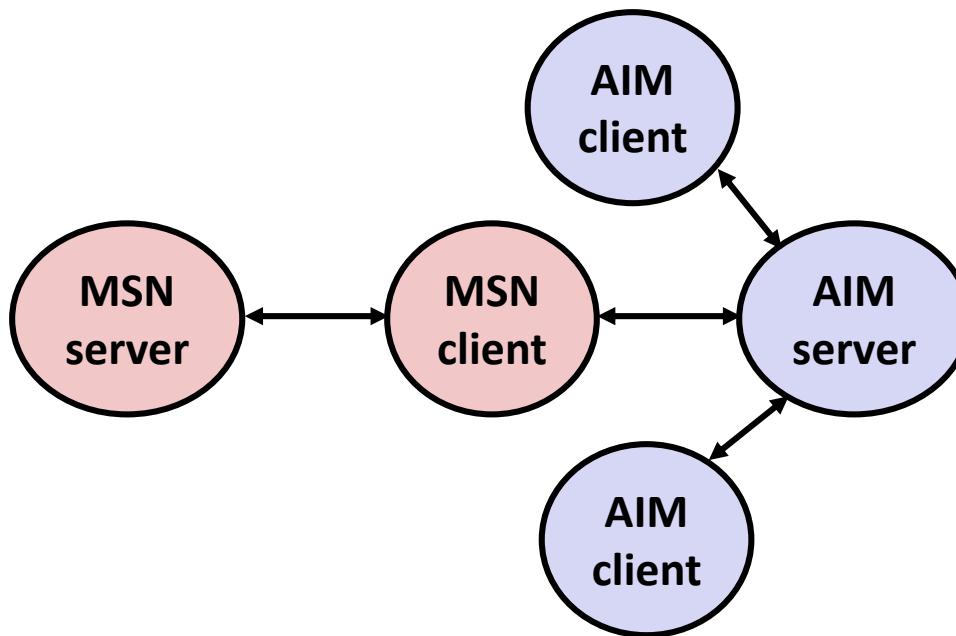
## ■ Once on a machine, scanned for other machines to attack

- invaded ~6000 computers in hours (10% of the Internet ☺ )
  - see June 1989 article in *Comm. of the ACM*
- the young author of the worm was prosecuted...
- and CERT was formed... still homed at CMU

# Example 2: IM War

## ■ July, 1999

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



# IM War (cont.)

## ■ August 1999

- Mysteriously, Messenger clients can no longer access AIM servers
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes
  - At least 13 such skirmishes
- What was really happening?
  - AOL had discovered a buffer overflow bug in their own AIM clients
  - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
  - When Microsoft changed code to match signature, AOL changed signature location

# Aside: Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- **Virus: Code that**
  - Adds itself to other programs
  - Does not run independently
- **Both are (usually) designed to spread among computers and to wreak havoc**

# **Computer Architecture I:**

## **Outline and Instruction Set Architecture**

CENG331 - Computer Organization (Sections 1-2)

Murat Manguoglu

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Coverage

- Our Approach
  - Work through designs for particular instruction set
    - Y86-64 – a simplified version of the Intel x86-64
    - If you know one, you more-or-less know them all
  - Work at “microarchitectural” level
    - Assemble basic hardware blocks into overall processor structure
      - Memories, functional units, etc.
    - Surround by control logic to make sure each instruction flows through properly
  - Use simple hardware description language to describe control logic
    - Can extend and modify
    - Test via simulation
    - Route to design using Verilog Hardware Description Language
      - See Web aside ARCH:VLOG

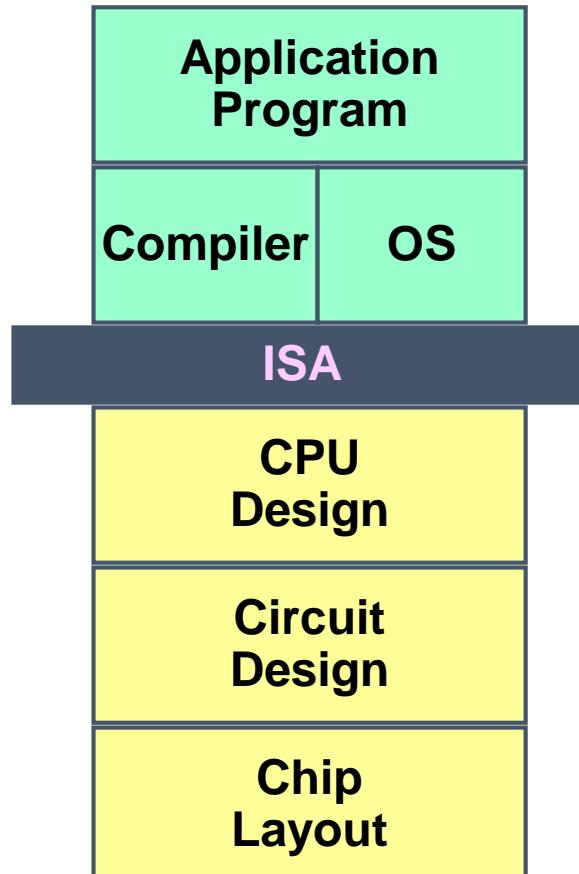
# Instruction Set Architecture

- Assembly Language View

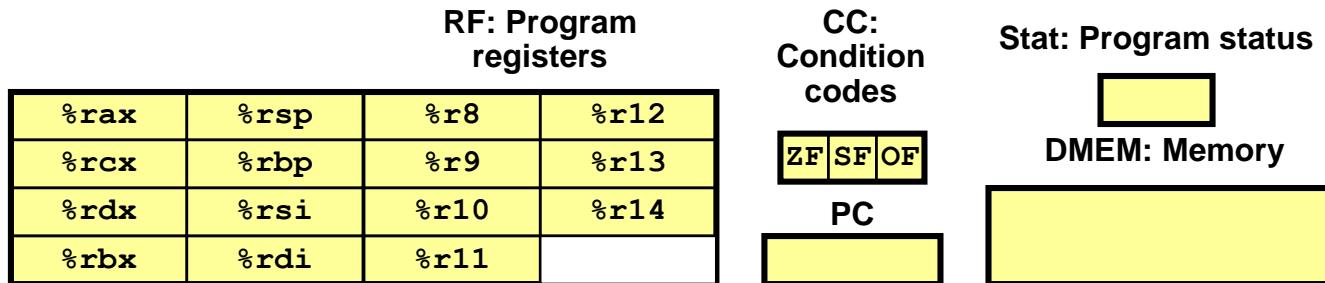
- Processor state
  - Registers, memory, ...
- Instructions
  - addq, pushq, ret, ...
  - How instructions are encoded as bytes

- Layer of Abstraction

- Above: how to program machine
  - Processor executes instructions in a sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast
  - E.g., execute multiple instructions simultaneously



# Y86-64 Processor State



- Program Registers
  - 15 registers (omit %r15). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero
    - SF:Negative
    - OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

# Y86-64 Instruction Set #1

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Y86-64 Instructions

- Format
  - 1–10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with x86-64
  - Each accesses and modifies some part(s) of the program state

# Y86-64 Instruction Set

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Y86-64 Instruction Set #2

**Byte**

halt

0	0
---	---

nop

1	0
---	---

cmoveXX rA, rB

2	fn	rA	rB
---	----	----	----

irmovq V, rB

3	0	F	rB	V
---	---	---	----	---

rrmovq rA, D(rB)

4	0	rA	rB	D
---	---	----	----	---

mrrmovq D(rB), rA

5	0	rA	rB	D
---	---	----	----	---

OPq rA, rB

6	fn	rA	rB
---	----	----	----

jXX Dest

7	fn	Dest
---	----	------

call Dest

8	0	Dest
---	---	------

ret

9	0
---	---

pushq rA

A	0	rA	F
---	---	----	---

popq rA

B	0	rA	F
---	---	----	---

rrmovq	7	0
cmovele	7	1
cmovl	7	2
cmove	7	3
cmovne	7	4
cmovge	7	5
cmovg	7	6

# Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Diagram illustrating the Y86-64 instruction set. The table shows 11 bytes for each instruction. Bytes 0-5 are fixed, while bytes 6-11 are variable. The variable bytes are color-coded: yellow for the function code (fn), pink for registers (rA, rB), and light blue for memory addresses (V, D, Dest). A bracket on the right side groups bytes 6-9, which represent various operations: addq, subq, andq, and xorq. An arrow points from the OPq rA, rB row to this group.

# Y86-64 Instruction Set #4

Byte	0	1	2	3	4	5	6	7	
halt	0	0							
nop	1	0							
cmoveXX rA, rB	2	fn	rA	rB					
irmovq V, rB	3	0	F	rB	V				
rmmovq rA, D(rB)	4	0	rA	rB	D				
mrmovq D(rB), rA	5	0	rA	rB	D				
OPq rA, rB	6	fn	rA	rB					
jXX Dest	7	fn			Dest				
call Dest	8	0			Dest				
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					
jmp	7	0							
jle	7	1							
jl	7	2							
je	7	3							
jne	7	4							
jge	7	5							
jg	7	6							

# Encoding Registers

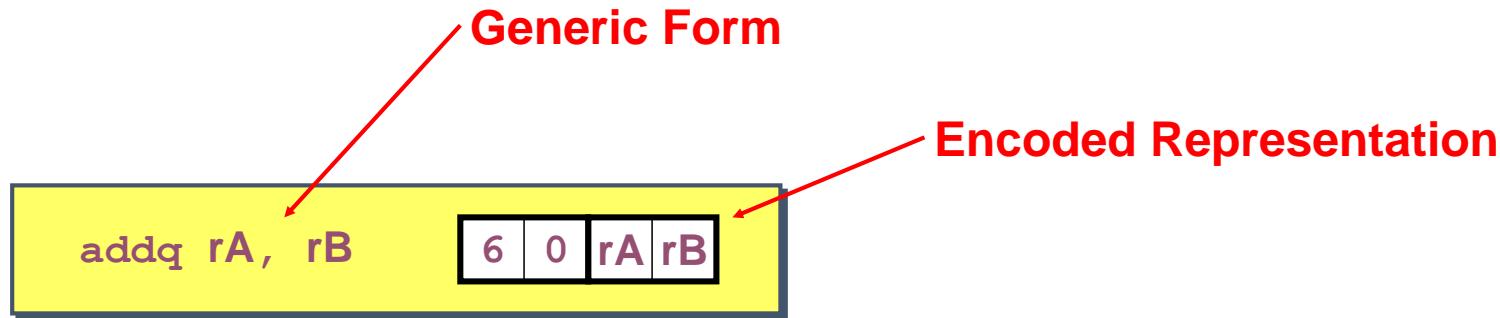
- Each register has 4-bit ID

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7
%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

- Same encoding as in x86-64
- Register ID 15 (0xF) indicates “no register”
  - Will use this in our hardware design in multiple places

# Instruction Example

- Addition Instruction



- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- e.g., `addq %rax, %rsi` Encoding: 60 06
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

# Arithmetic and Logical Operations

## Instruction Code

Add



Subtract (rA from rB)



And



Exclusive-Or



## Function Code

- Refer to generically as “OPq”
- Encodings differ only by “function code”
  - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

# Move Operations

Register → Register

rrmovq rA, rB

2	0
---	---

Immediate → Register

irmovq V, rB

3	0	F	rB
---	---	---	----

V

Register → Memory

rmmovq rA, D(rB)

4	0	rA	rB
---	---	----	----

D

Memory → Register

mrmovq D(rB), rA

5	0	rA	rB
---	---	----	----

D

- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

# Move Instruction Examples

X86-64

```
movq $0xabcd, %rdx
```

Y86-64

```
irmovq $0xabcd, %rdx
```

Encoding:

```
30 82 cd ab 00 00 00 00 00 00
```

```
movq %rsp, %rbx
```

```
rrmovq %rsp, %rbx
```

Encoding:

```
20 43
```

```
movq -12(%rbp),%rcx
```

```
mrmovq -12(%rbp),%rcx
```

Encoding:

```
50 15 f4 ff ff ff ff ff ff ff ff ff
```

```
movq %rsi,0x41c(%rsp)
```

```
rmmovq %rsi,0x41c(%rsp)
```

Encoding:

```
40 64 1c 04 00 00 00 00 00 00 00
```

# Conditional Move Instructions

## Move Unconditionally

`rrmovq rA, rB`



## Move When Less or Equal

`cmovele rA, rB`



## Move When Less

`cmovl rA, rB`



## Move When Equal

`cmove rA, rB`



## Move When Not Equal

`cmovne rA, rB`



## Move When Greater or Equal

`cmovge rA, rB`



## Move When Greater

`cmovg rA, rB`



- Refer to generically as “`cmovXX`”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of `rrmovq` instruction
  - (Conditionally) copy value from source to destination register

# Jump Instructions

## Jump (Conditionally)

jxx Dest	7	fn	Dest
----------	---	----	------

- Refer to generically as “jXX”
- Encodings differ only by “function code” fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

# Jump Instructions

## Jump Unconditionally

**jmp Dest**    7 | 0    Dest

## Jump When Less or Equal

**jle Dest**    7 | 1    Dest

## Jump When Less

**jl Dest**    7 | 2    Dest

## Jump When Equal

**je Dest**    7 | 3    Dest

## Jump When Not Equal

**jne Dest**    7 | 4    Dest

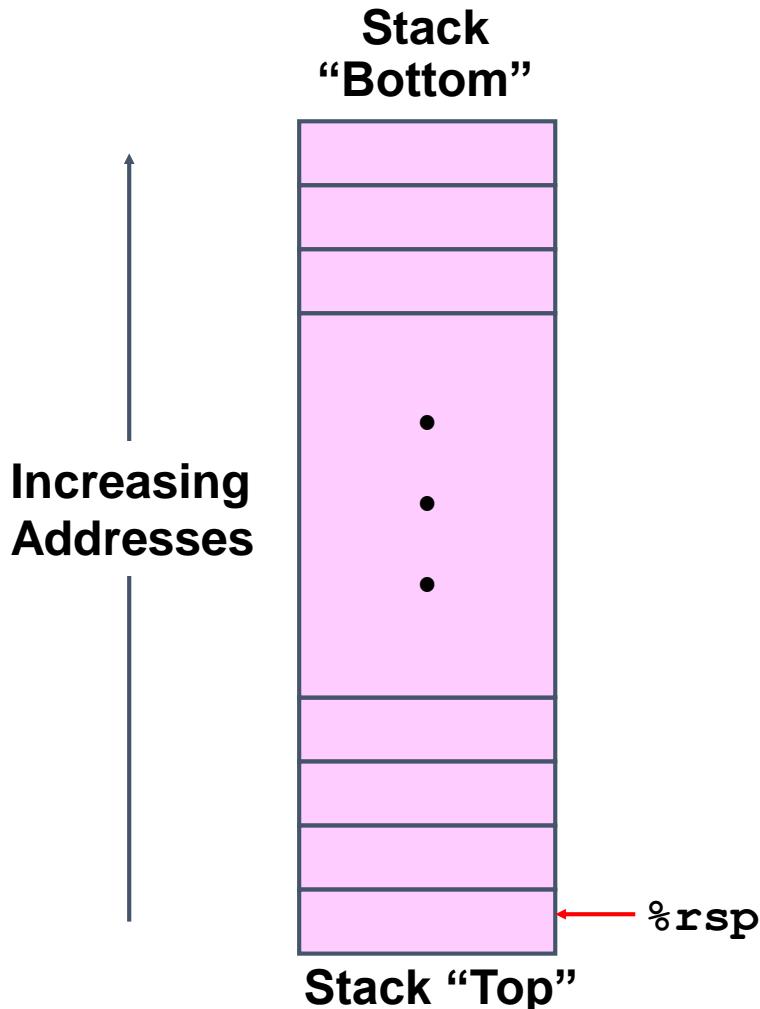
## Jump When Greater or Equal

**jge Dest**    7 | 5    Dest

## Jump When Greater

**jg Dest**    7 | 6    Dest

# Y86-64 Program Stack



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by `%rsp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

# Stack Operations

pushq rA



- Decrement `%rsp` by 8
- Store word from `rA` to memory at `%rsp`
- Like x86-64

popq rA



- Read word from memory at `%rsp`
- Save in `rA`
- Increment `%rsp` by 8
- Like x86-64

# Subroutine Call and Return

call Dest

8	0
---	---

Dest

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

ret

9	0
---	---

- Pop value from stack
- Use as address for next instruction
- Like x86-64

# Miscellaneous Instructions



- Don't do anything



- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

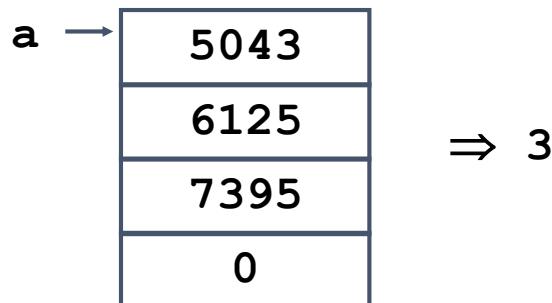
# Status Conditions

Mnemonic	Code	
AOK	1	<ul style="list-style-type: none"><li>• Normal operation</li></ul>
Mnemonic	Code	
HLT	2	<ul style="list-style-type: none"><li>• Halt instruction encountered</li></ul>
Mnemonic	Code	
ADR	3	<ul style="list-style-type: none"><li>• Bad address (either instruction or data) encountered</li></ul>
Mnemonic	Code	
INS	4	<ul style="list-style-type: none"><li>• Invalid instruction encountered</li></ul>
 <ul style="list-style-type: none"><li>• Desired Behavior<ul style="list-style-type: none"><li>• If AOK, keep going</li><li>• Otherwise, stop program execution</li></ul></li></ul>		

# Writing Y86-64 Code

- Try to Use C Compiler as Much as Possible
  - Write code in C
  - Compile for x86-64 with `gcc -Og -S`
  - Transliterate into Y86-64
  - *Modern compilers make this more difficult*
- Coding Example
  - Find number of elements in null-terminated list

```
int len1(int a[]);
```



# Y86-64 Code Generation Example

- First Try

- Write typical array code

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
    long len;
    for (len = 0; a[len]; len++)
        ;
    return len;
}
```

- Problem

- Hard to do array indexing on Y86-64
    - Since don't have scaled addressing modes

```
L3:
```

```
addq $1,%rax
cmpq $0, (%rdi,%rax,8)
jne L3
```

- Compile with gcc -Og -S

# Y86-64 Code Generation Example #2

- Second Try

- Write C code that mimics expected Y86-64 code

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```

- Result

- Compiler generates exact same code as before!
- Compiler converts both versions into same intermediate form

# Y86-64 Code Generation Example #3

```
len:  
    irmovq $1, %r8          # Constant 1  
    irmovq $8, %r9          # Constant 8  
    irmovq $0, %rax         # len = 0  
    mrmovq (%rdi), %rdx     # val = *a  
    andq %rdx, %rdx         # Test val  
    je Done                  # If zero, goto Done  
  
Loop:  
    addq %r8, %rax          # len++  
    addq %r9, %rdi          # a++  
    mrmovq (%rdi), %rdx     # val = *a  
    andq %rdx, %rdx         # Test val  
    jne Loop                 # If !0, goto Loop  
  
Done:  
    ret
```

Register	Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

# Y86-64 Sample Program Structure #1

```
init:          # Initialization
  ...
  call Main
  halt

  .align 8      # Program data
array:

Main:         # Main function
  ...
  call len    ...

len:          # Length function
  ...

  .pos 0x100   # Placement of stack
Stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

# X86-64 Program Structure #2

```
init:
```

```
    # Set up stack pointer
    irmovq Stack, %rsp
    # Execute main program
    call Main
    # Terminate
    halt
```

```
# Array of 4 elements + terminating 0
```

```
    .align 8
```

```
Array:
```

```
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
    .quad 0
```

- Program starts at address 0
- Must set up stack
- Must initialize data
- Can use symbolic names

# X86-64 Program Structure #3

Main:

```
irmovq array,%rdi  
# call len(array)  
call len  
ret
```

- Set up call to len
  - Follow X86-64 procedure conventions
  - Push array address as argument

# Assembling Y86-64 Program

```
unix> yas len.ys
```

- Generates “object code” file len.yo
  - Actually looks like disassembler output

```
0x054:          | len:  
0x054: 30f801000000000000000000 |  irmovq $1, %r8           # Constant 1  
0x05e: 30f908000000000000000000 |  irmovq $8, %r9           # Constant 8  
0x068: 30f000000000000000000000 |  irmovq $0, %rax          # len = 0  
0x072: 502700000000000000000000 |  mrmovq (%rdi), %rdx      # val = *a  
0x07c: 6222          |  andq %rdx, %rdx          # Test val  
0x07e: 73a000000000000000000000 |  je Done                 # If zero, goto Done  
0x087:          | Loop:  
0x087: 6080          |  addq %r8, %rax          # len++  
0x089: 6097          |  addq %r9, %rdi          # a++  
0x08b: 502700000000000000000000 |  mrmovq (%rdi), %rdx      # val = *a  
0x095: 6222          |  andq %rdx, %rdx          # Test val  
0x097: 748700000000000000000000 |  jne Loop                # If !0, goto Loop  
0xa0:          | Done:  
0xa0: 90             |  ret
```

# Simulating Y86-64 Program

```
unix> yis len.yo
```

- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
```

```
Changes to registers:
```

%rax:	0x0000000000000000	0x0000000000000004
%rsp:	0x0000000000000000	0x0000000000000000100
%rdi:	0x0000000000000000	0x000000000000000038
%r8:	0x0000000000000000	0x000000000000000001
%r9:	0x0000000000000000	0x000000000000000008

```
Changes to memory:
```

0x00f0:	0x0000000000000000	0x000000000000000053
0x00f8:	0x0000000000000000	0x000000000000000013

For more information: <http://csapp.cs.cmu.edu/3e/simguide.pdf>

# CISC Instruction Sets

- Complex Instruction Set Computer
- IA32 is example
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
  - `addq %rax, 12(%rbx,%rcx,8)`
    - requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions
- Philosophy
  - Add instructions to perform “typical” programming tasks

# RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- **Fewer, simpler instructions**
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- **Register-oriented instruction set**
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- **Only load and store instructions can access memory**
  - Similar to Y86-64 `mrmovq` and `rmmovq`
- **No Condition codes**
  - Test instructions return 0/1 in register

# MIPS (Microprocessor without Interlocked Pipeline Stages) Registers

\$0	\$0	Constant 0	\$16	\$s0	Callee Save Temporaries: May not be overwritten by called procedures
\$1	\$at	Reserved Temp.	\$17	\$s1	
\$2	\$v0		\$18	\$s2	
\$3	\$v1	Return Values	\$19	\$s3	
\$4	\$a0		\$20	\$s4	
\$5	\$a1	Procedure arguments	\$21	\$s5	
\$6	\$a2		\$22	\$s6	
\$7	\$a3		\$23	\$s7	
\$8	\$t0		\$24	\$t8	Caller Save Temp
\$9	\$t1		\$25	\$t9	
\$10	\$t2	Caller Save Temporaries: May be overwritten by called procedures	\$26	\$k0	Reserved for Operating Sys
\$11	\$t3		\$27	\$k1	
\$12	\$t4		\$28	\$gp	Global Pointer
\$13	\$t5		\$29	\$sp	Stack Pointer
\$14	\$t6		\$30	\$s8	Callee Save Temp
\$15	\$t7		\$31	\$ra	Return Address

# MIPS Instruction Examples

## R-R

Op	Ra	Rb	Rd	00000	Fn
----	----	----	----	-------	----

addu \$3,\$2,\$1 # Register add: \$3 = \$2+\$1

## R-I

Op	Ra	Rb	Immediate
----	----	----	-----------

addu \$3,\$2, 3145 # Immediate add: \$3 = \$2+3145

sll \$3,\$2,2 # Shift left: \$3 = \$2 << 2

## Branch

Op	Ra	Rb	Offset
----	----	----	--------

beq \$3,\$2,dest # Branch when \$3 = \$2

## Load/Store

Op	Ra	Rb	Offset
----	----	----	--------

lw \$3,16(\$2) # Load Word: \$3 = M[\$2+16]

sw \$3,16(\$2) # Store Word: M[\$2+16] = \$3

# CISC vs. RISC

- Original Debate
  - Strong opinions!
  - CISC proponents---easy for compiler, fewer code bytes
  - RISC proponents---better for optimizing compilers, can make run fast with simple chip design
- Current Status
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - x86-64 adopted many RISC features
    - More registers; use them for argument passing
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power
    - Most cell phones use ARM processor
      - Except with the addition of Apple M1 for desktops in 2020 !

# Summary

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC
- How Important is ISA Design?
  - Less now than before
    - With enough hardware, can make almost anything go fast

# CS:APP Chapter 4

# Computer Architecture

# Logic Design

CENG331 - Computer Organization (Sections 1-2)

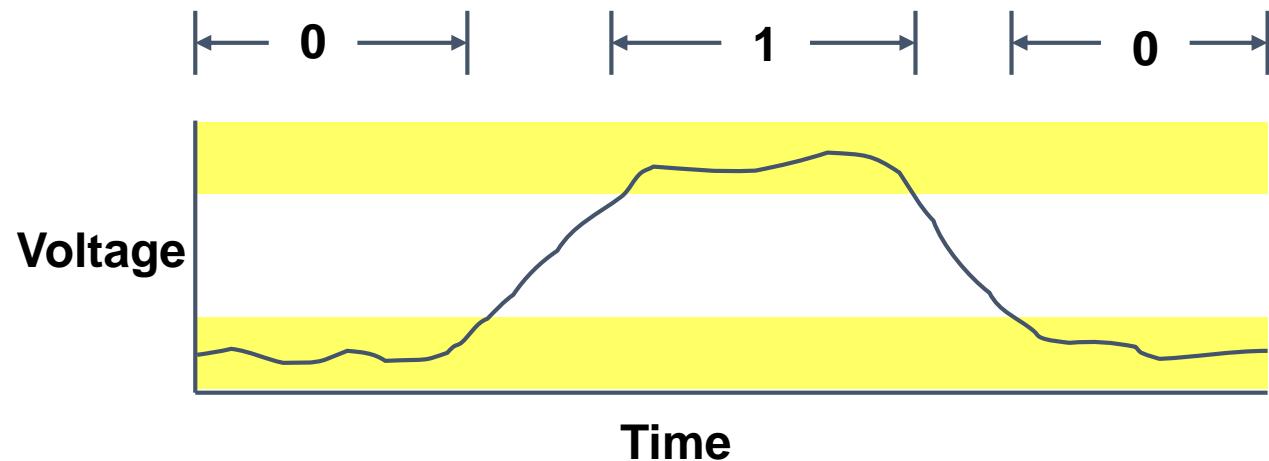
Murat Manguoglu

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

# Overview of Logic Design

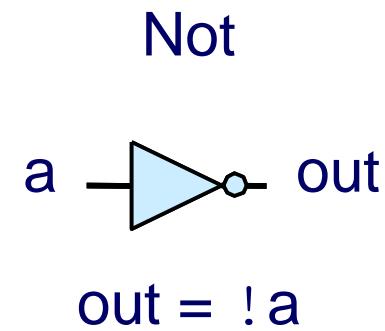
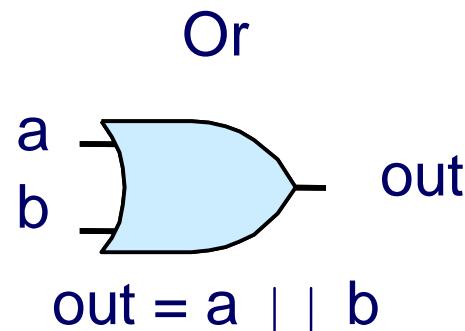
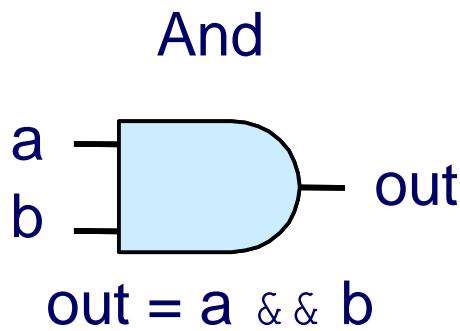
- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage
- Bits are Our Friends
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information

# Digital Signals

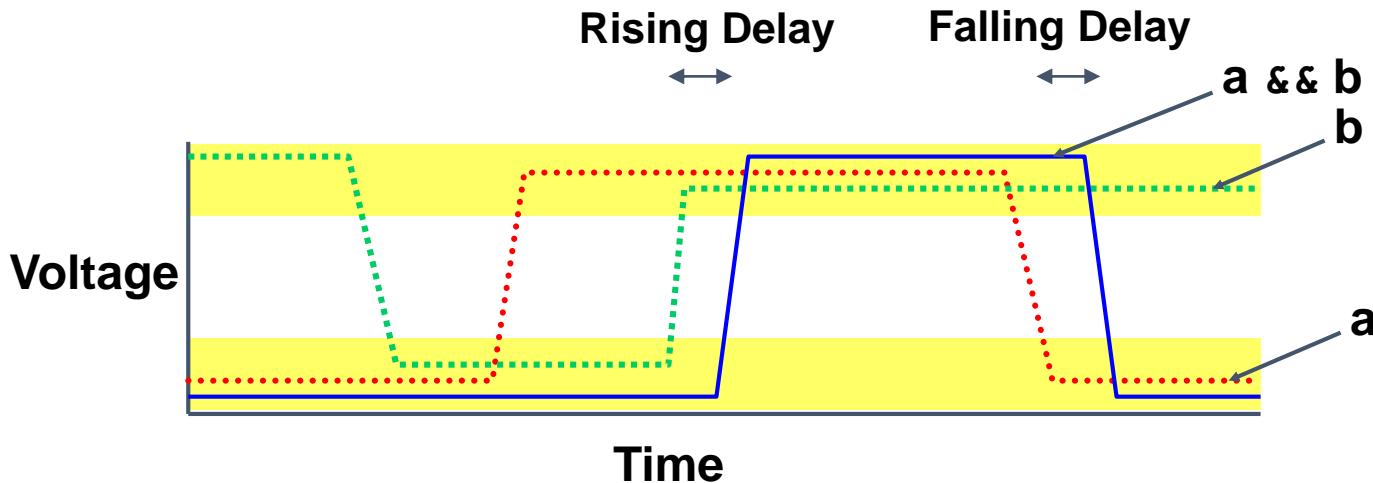


- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low quality circuit elements
  - Can make circuits simple, small, and fast

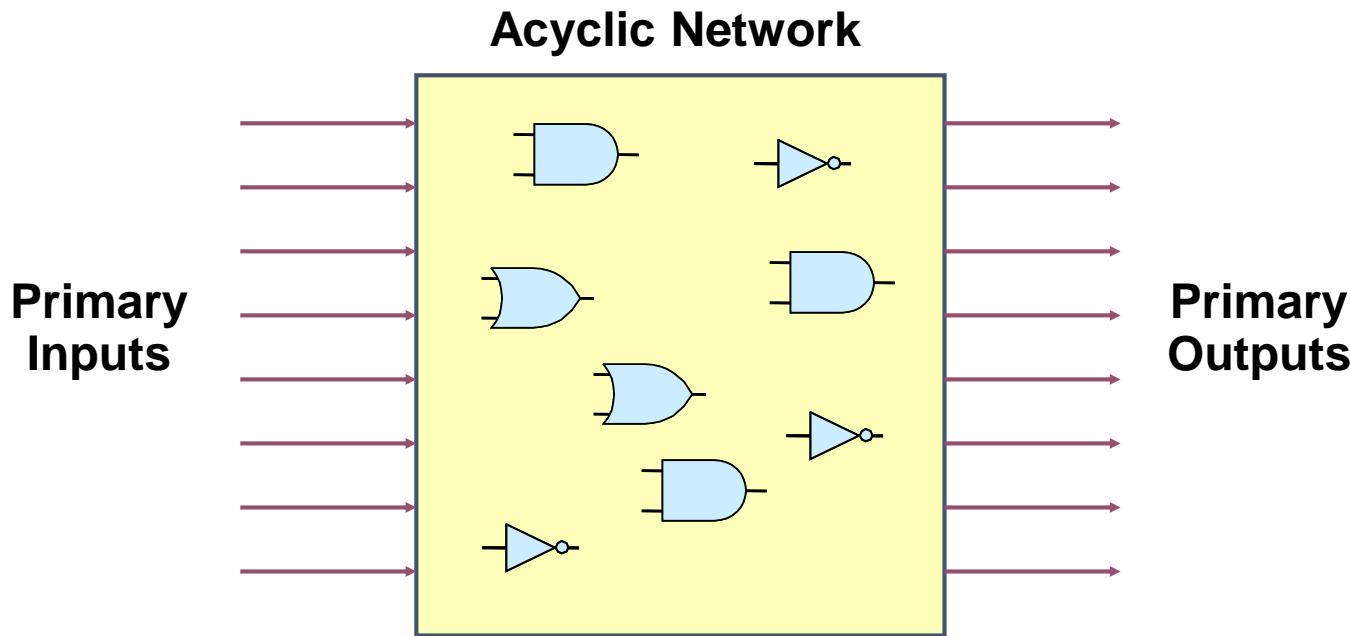
# Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some, small delay



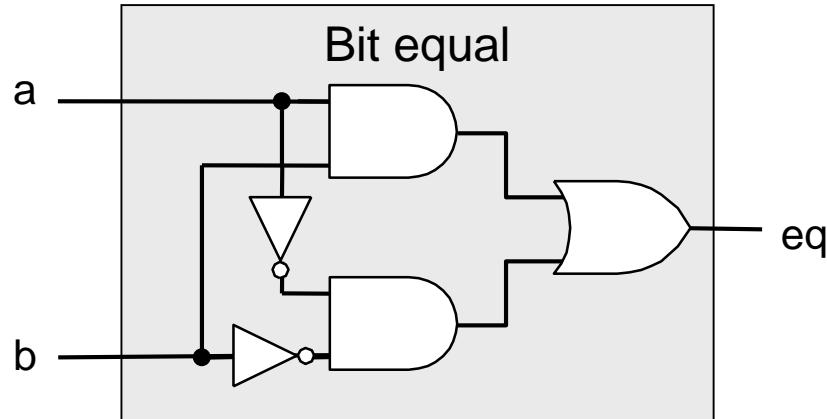
# Combinational Circuits



- **Acyclic Network of Logic Gates**

- Continuously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs

# Bit Equality

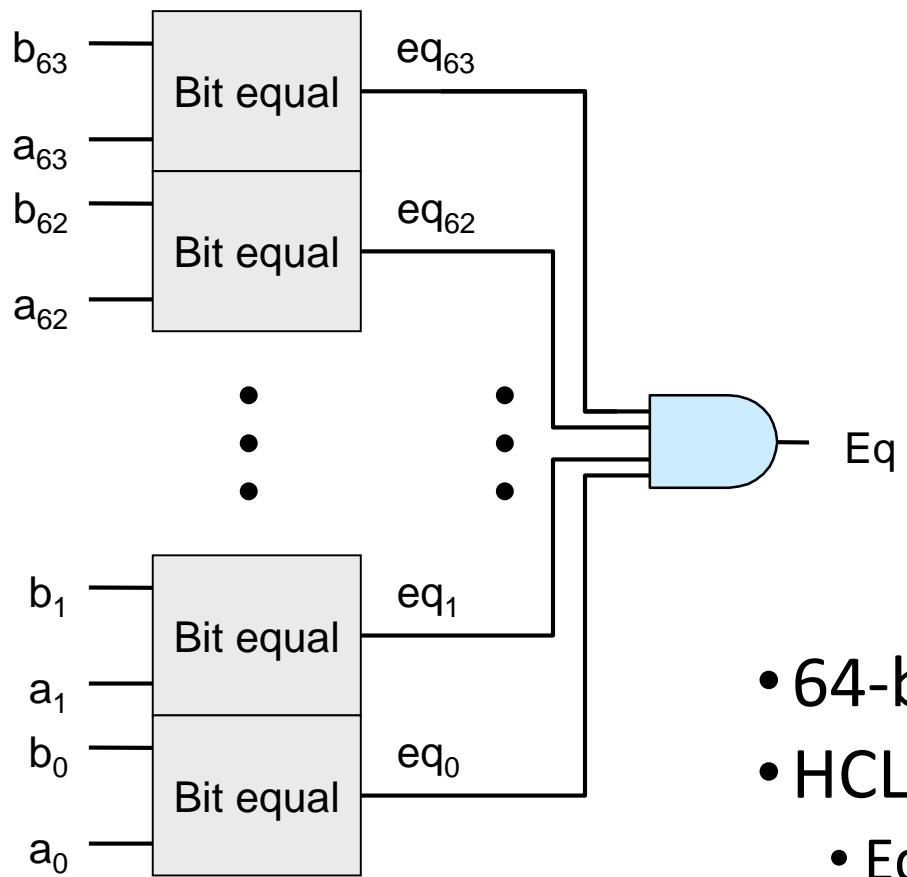


**HCL Expression**

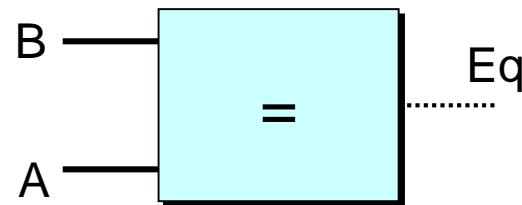
```
bool eq = (a&&b) || (!a&&!b)
```

- Generate 1 if *a* and *b* are equal
- **Hardware Control Language (HCL)**
  - Very simple hardware description language
    - Boolean operations have syntax similar to C logical operations
  - We'll use it to describe control logic for processors

# Word Equality



Word-Level Representation

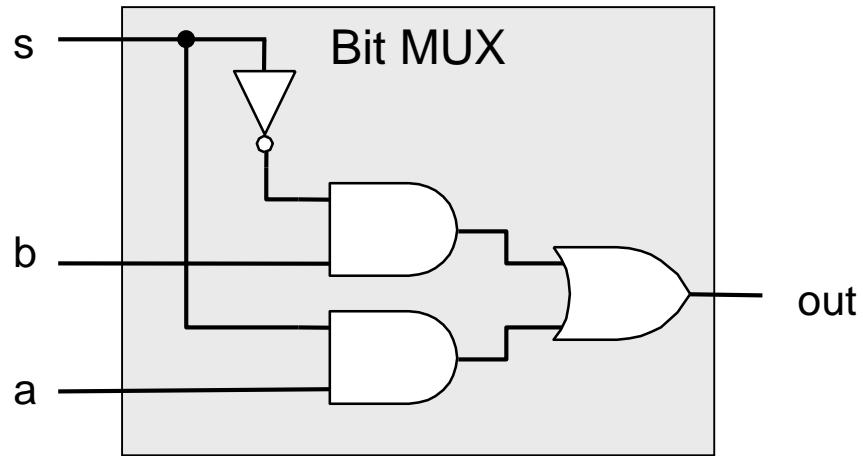


HCL Representation

bool Eq = (A == B)

- 64-bit word size
- HCL representation
  - Equality operation
  - Generates Boolean value

# Bit-Level Multiplexor

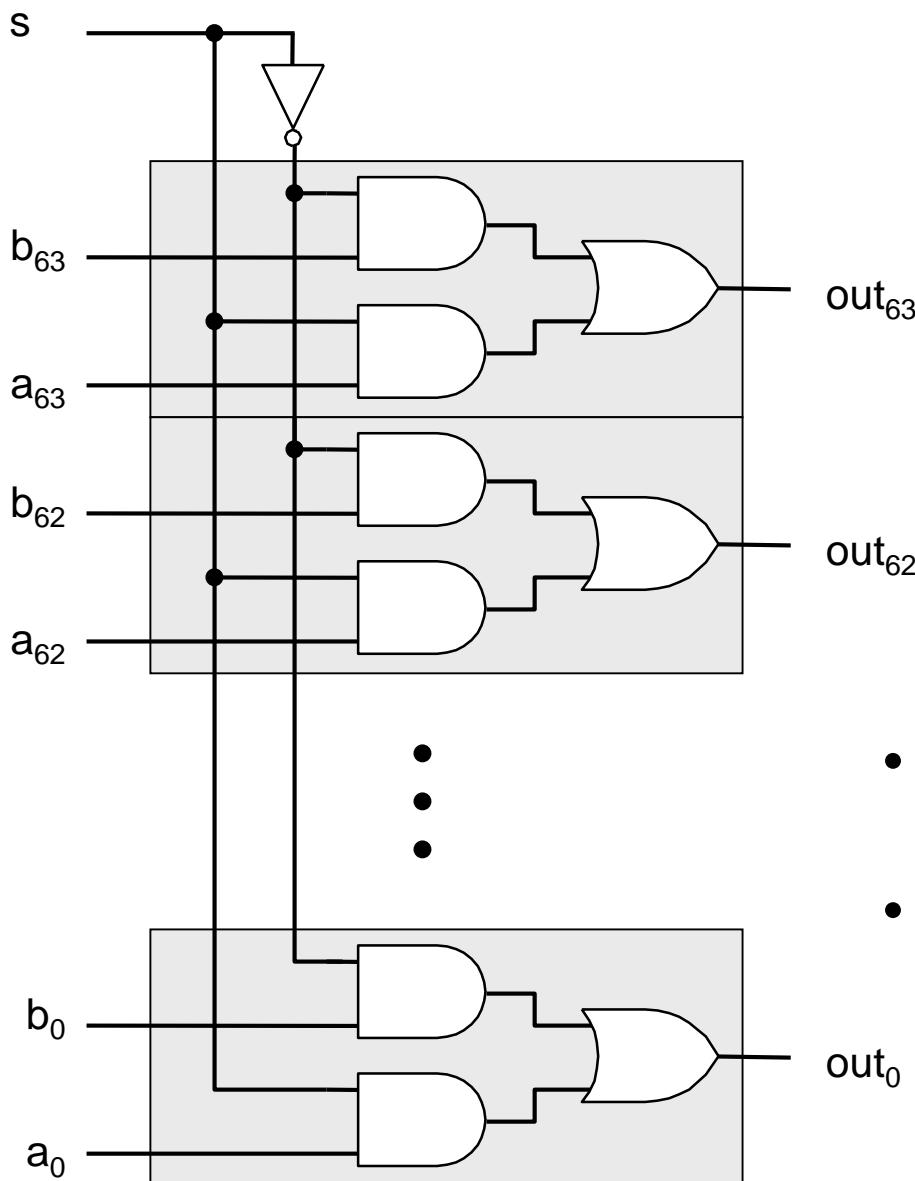


**HCL Expression**

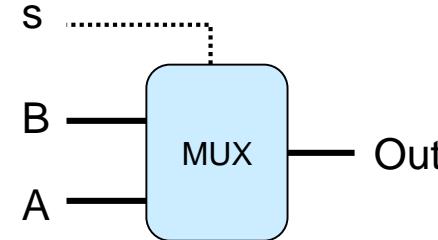
```
bool out = (s&&a) || (!s&&b)
```

- Control signal s
- Data signals a and b
- Output a when  $s=1$ , b when  $s=0$

# Word Multiplexor



## Word-Level Representation



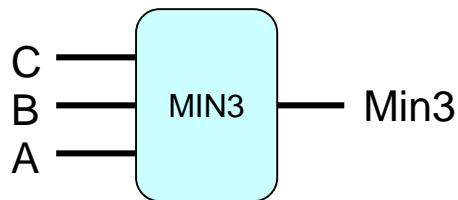
## HCL Representation

```
int Out = [  
    s : A;  
    1 : B;  
];
```

- Select input word A or B depending on control signal  $s$
- HCL representation
  - Case expression
  - Series of test : value pairs
  - Output value for first successful test

# HCL Word-Level Examples

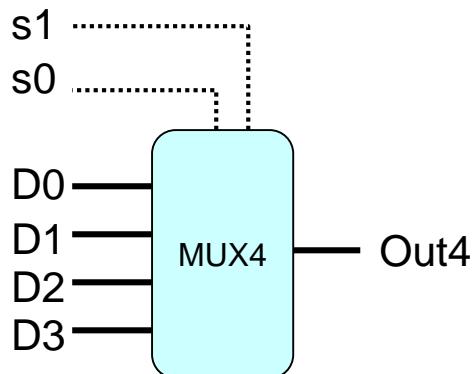
## Minimum of 3 Words



```
int Min3 = [
    A < B && A < C : A;
    B < A && B < C : B;
    1
    : C;
];
```

- Find minimum of three input words
- HCL case expression
- Final case guarantees match

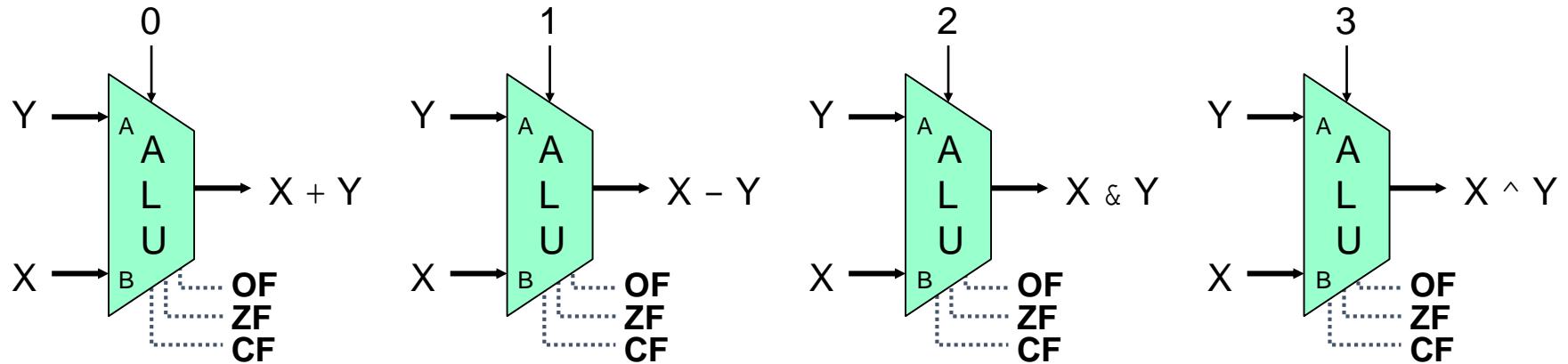
## 4-Way Multiplexor



```
int Out4 = [
    !s1&&!s0: D0;
    !s1      : D1;
    !s0      : D2;
    1        : D3;
];
```

- Select one of 4 inputs based on two control bits
- HCL case expression
- Simplify tests by assuming sequential matching

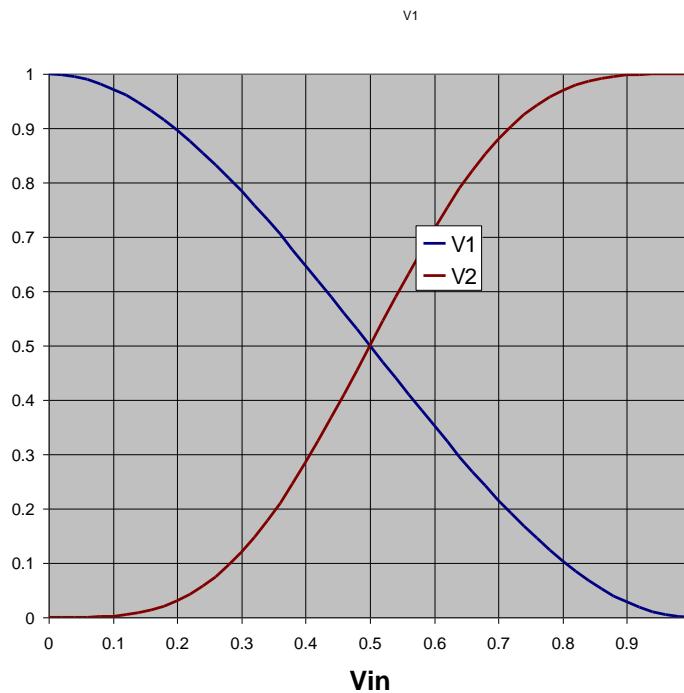
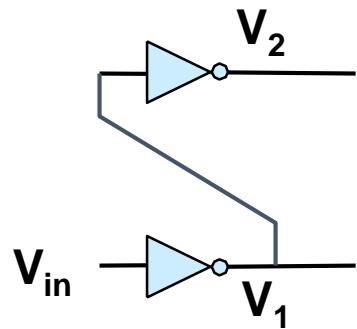
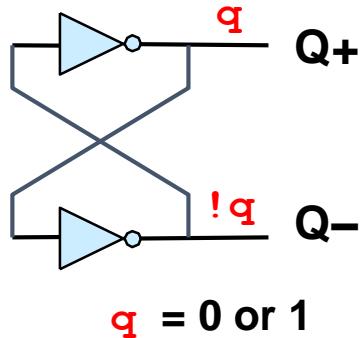
# Arithmetic Logic Unit



- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

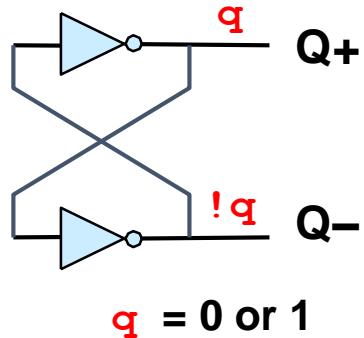
# Storing 1 Bit

## Bistable Element

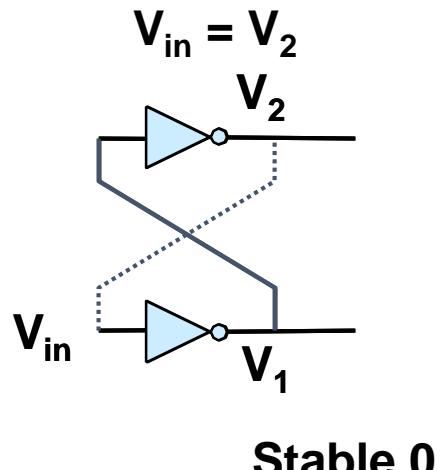


# Storing 1 Bit (cont.)

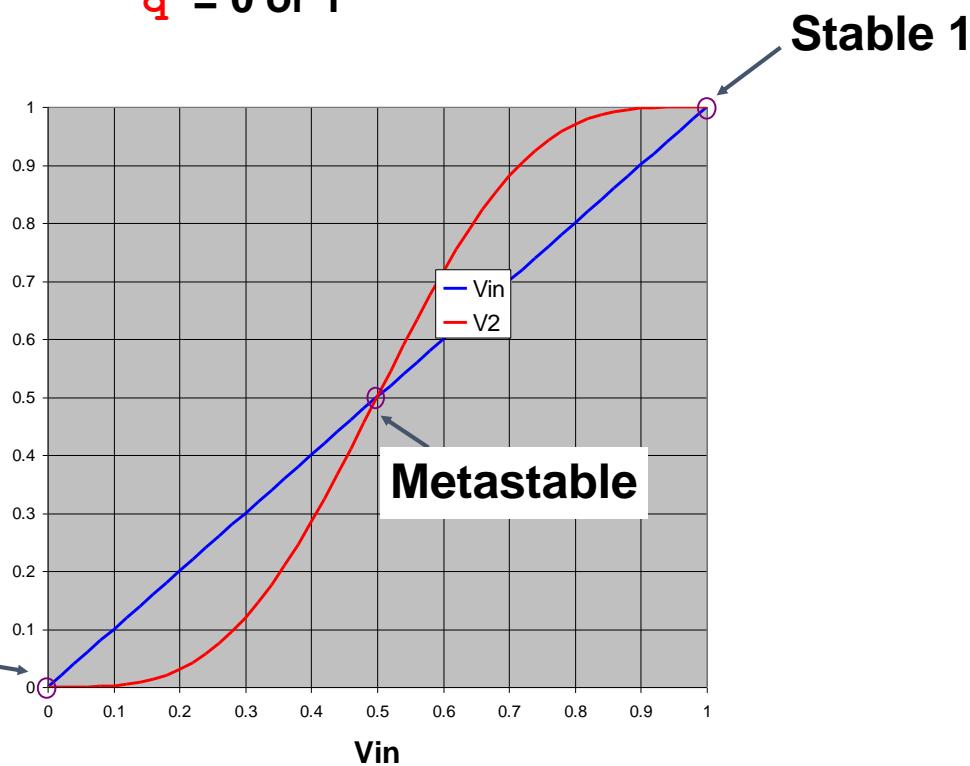
## Bistable Element



$$q = 0 \text{ or } 1$$

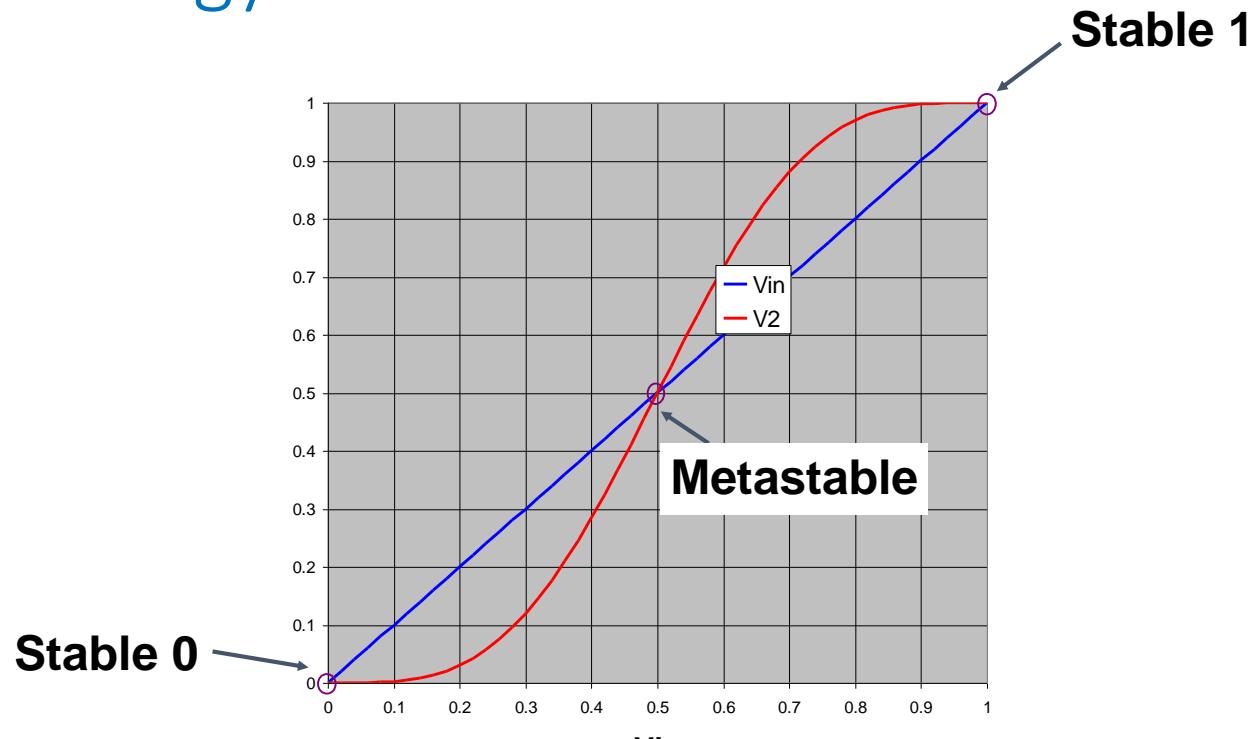


Stable 0



$V_{\text{in}}$

# Physical Analogy



Metastable

Stable 0

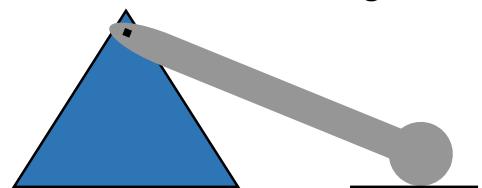
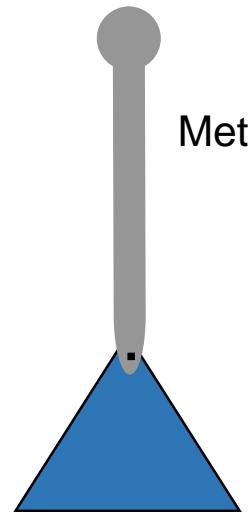
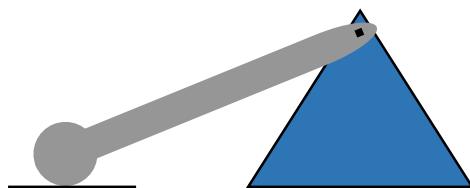
Stable 1

$V_{in}$

Metastable

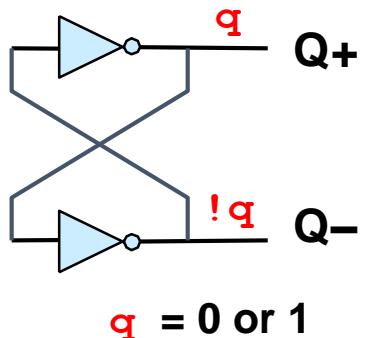
Stable left

Stable right

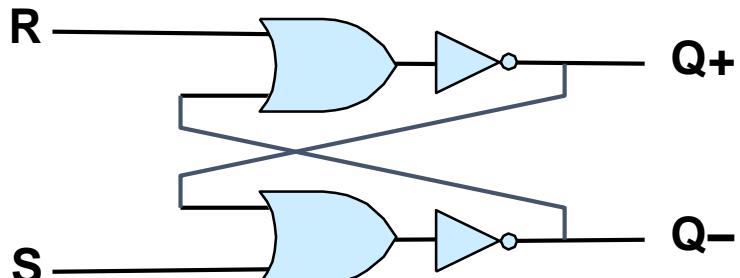


# Storing and Accessing 1 Bit

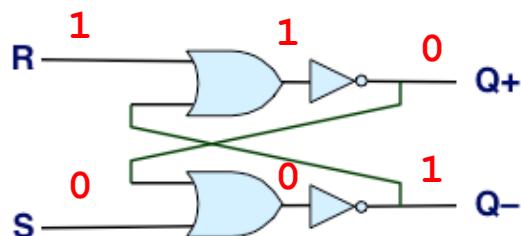
Bistable Element



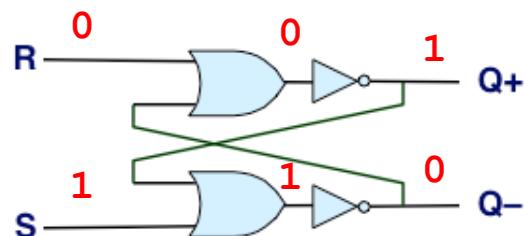
R-S Latch (Flip-Flop)



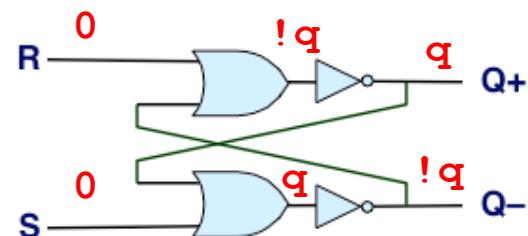
Resetting



Setting

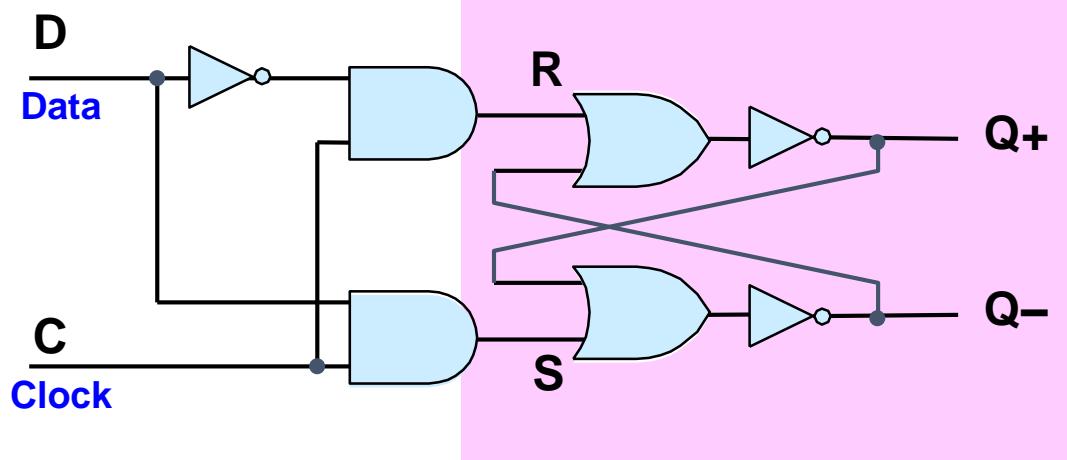


Storing

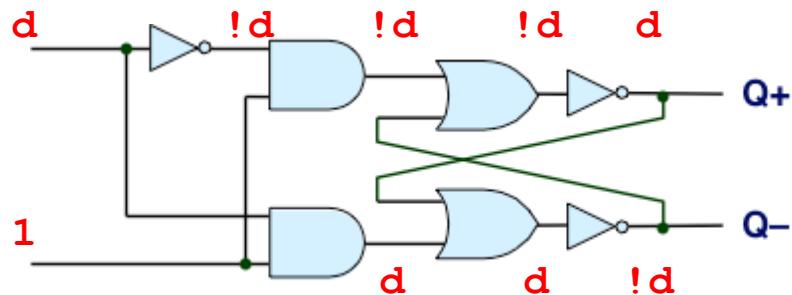


# 1-Bit Latch (Flip-Flop)

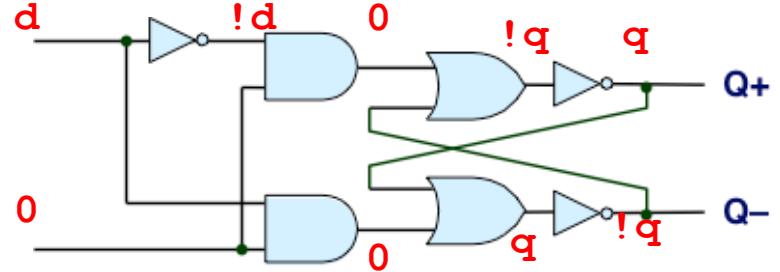
D Latch



Latching

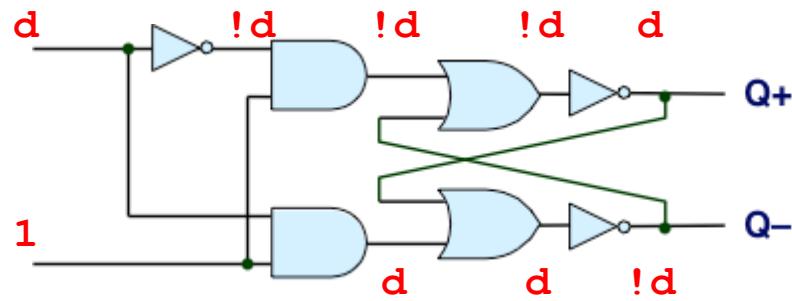


Storing

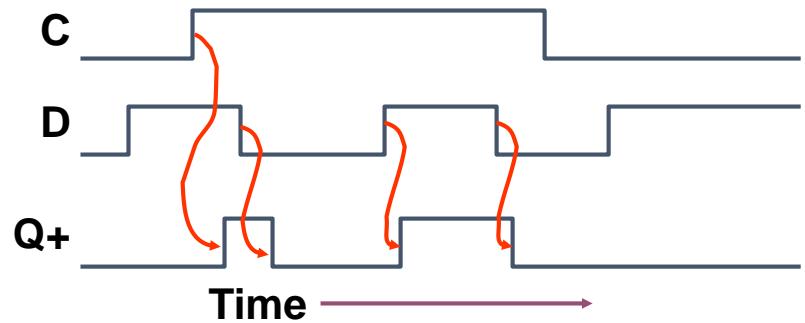


# Transparent 1-Bit Latch

## Latching

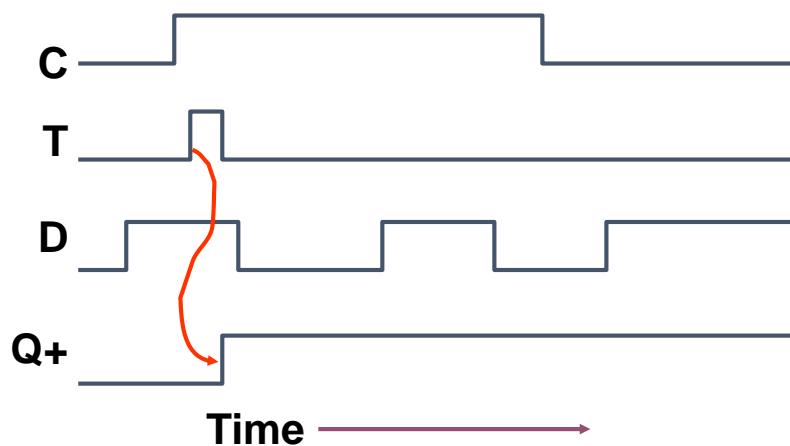
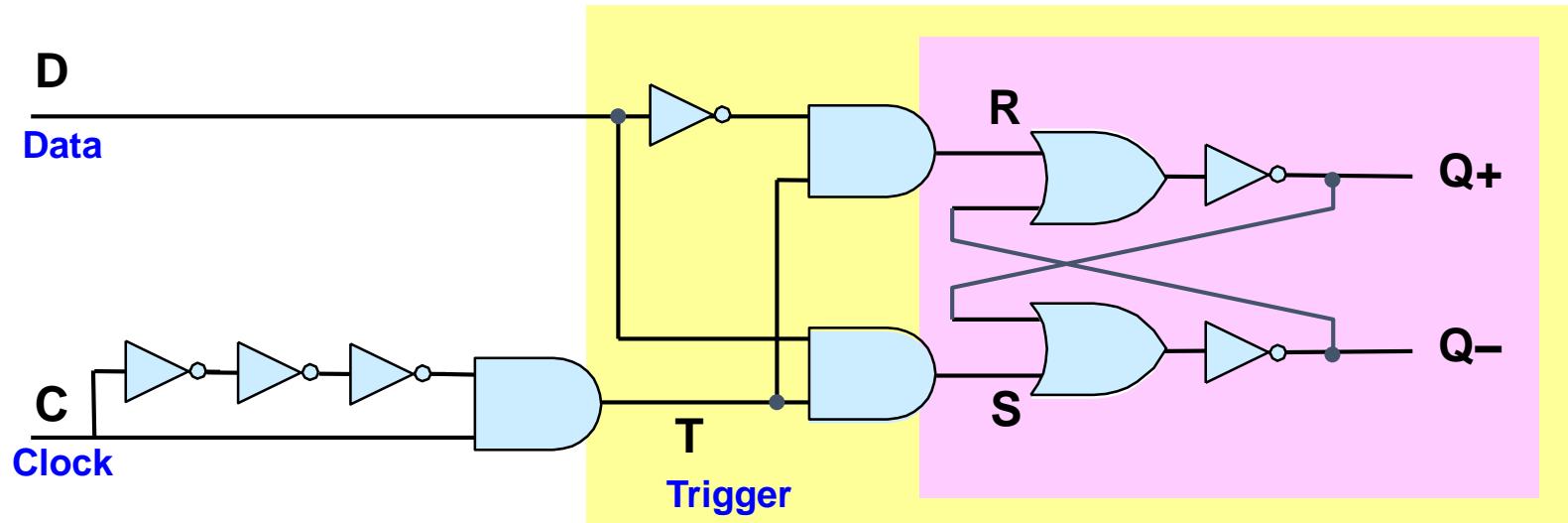


## Changing D



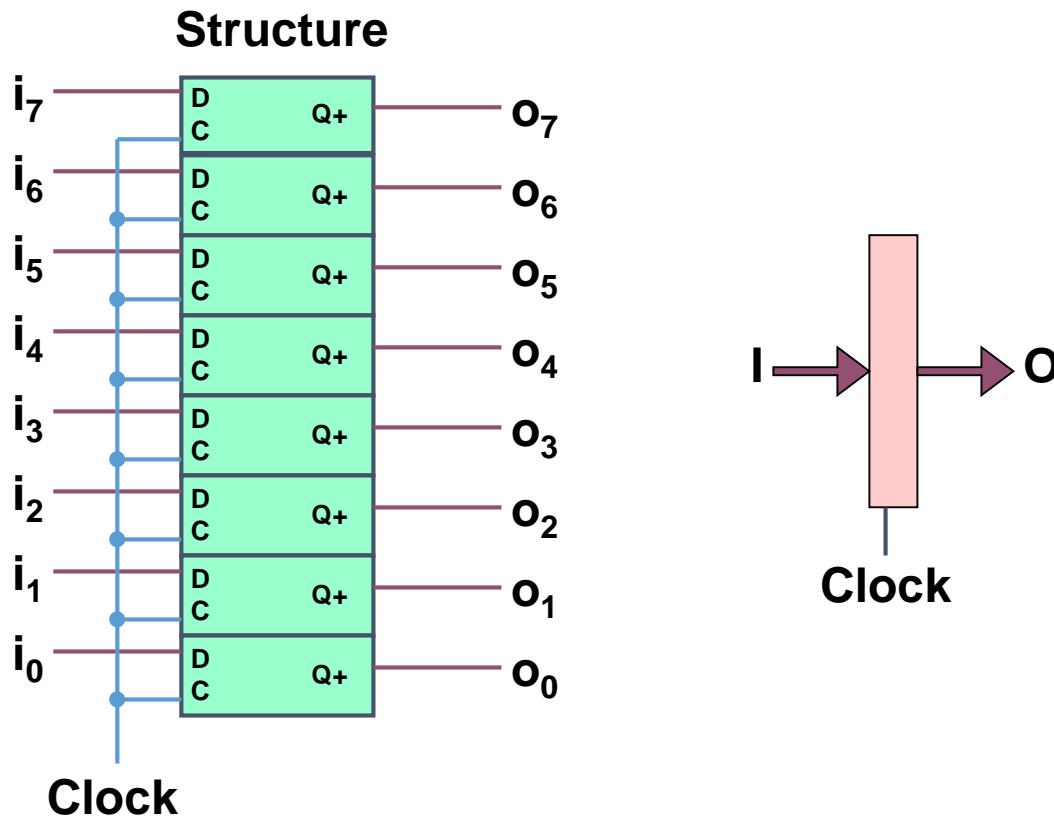
- When in latching mode, combinational propagation from  $D$  to  $Q+$  and  $Q-$
- Value latched depends on value of  $D$  as  $C$  falls

# Edge-Triggered Latch



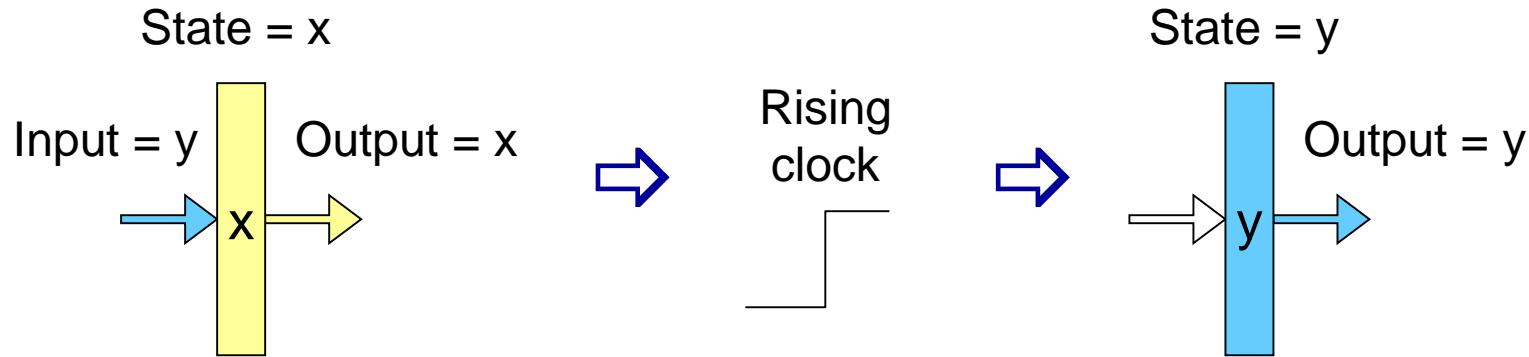
- Only in latching mode for brief period
  - Rising clock edge
- Value latched depends on data as clock rises
- Output remains stable at all other times

# Registers



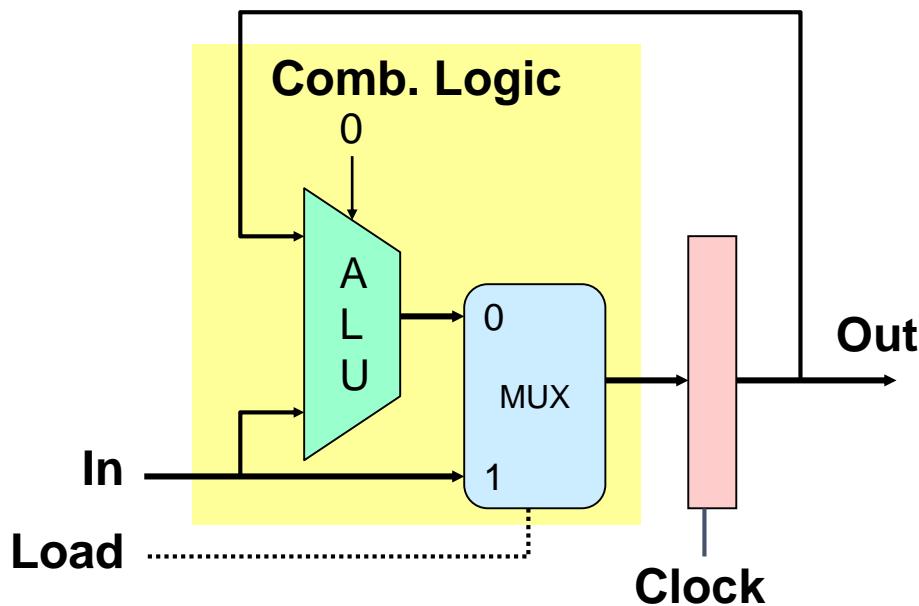
- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

# Register Operation

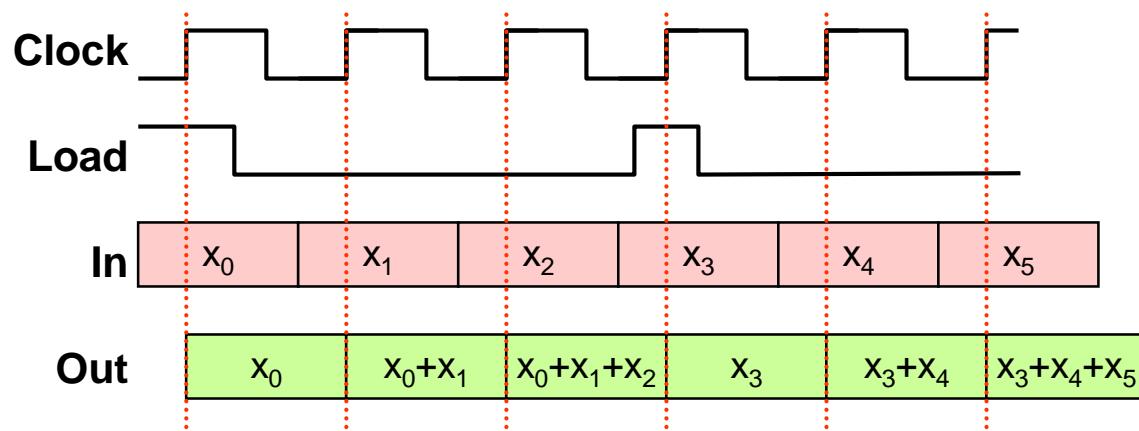


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

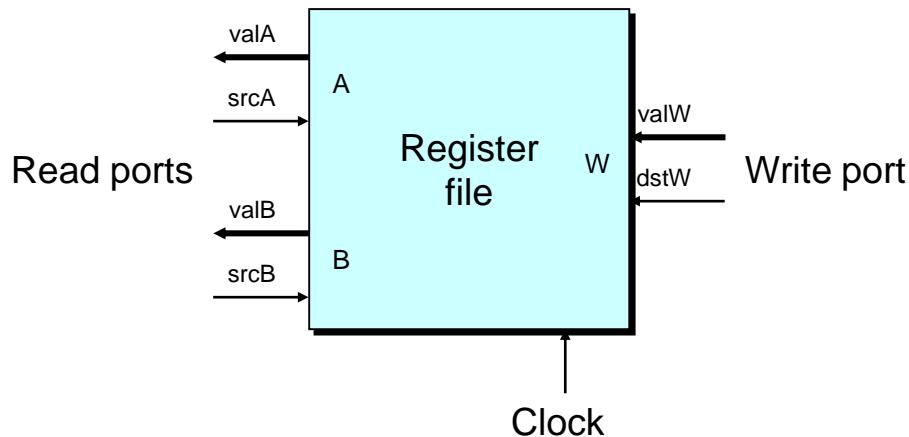
# State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle

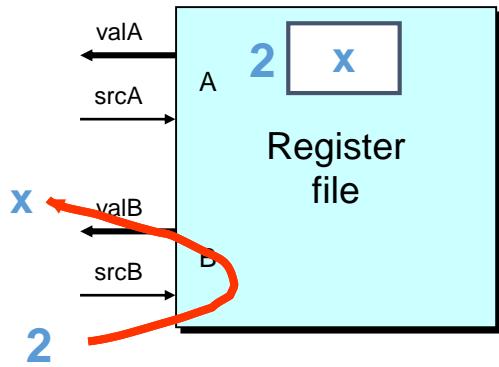


# Random-Access Memory

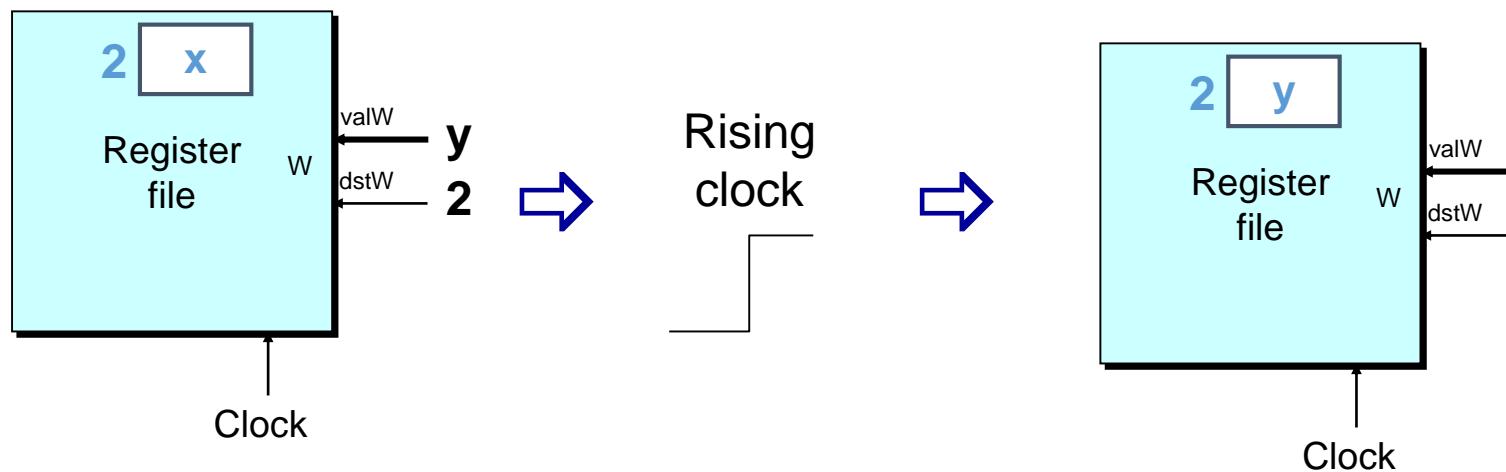


- Stores multiple words of memory
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers
  - `%rax`, `%rsp`, etc.
  - Register identifier serves as address
    - ID 15 (0xF) implies no read or write performed
- Multiple Ports
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

# Register File Timing



- **Reading**
  - Like combinational logic
  - Output data generated based on input address
    - After some delay
- **Writing**
  - Like register
  - Update only as clock rises



<u>Architecture</u>	<u>GPRs/data+address registers</u>	<u>FP registers</u>
<a href="#">z/Architecture</a>	16	16
<a href="#">Xilinx Spartan</a>	1	3
<a href="#">Xeon Phi<sup>[7]</sup></a>	16	32
<a href="#">x86-64<sup>[6][5]</sup></a>	16	16
<a href="#">W65C816S</a>	1	0
<a href="#">SPARC</a>	31	32
<a href="#">SH 16-bit</a>	1	6
<a href="#">Power Architecture</a>	32	32
<a href="#">PIC microcontroller</a>	1	0
<a href="#">Motorola 68k<sup>[9]</sup></a>	8 data (d0-d7), 8 address (a0-a7)	8 (if FP present)
<a href="#">Motorola 6800<sup>[8]</sup></a>	2 data, 1 index	0
<a href="#">MIPS</a>	31	32
<a href="#">Itanium</a>	128	128
<a href="#">IBM/360</a>	16	4 (if FP present)
<a href="#">IBM POWER</a>	32	32
<a href="#">IBM Cell SPE</a>	0	1
<a href="#">IA-32<sup>[5]</sup></a>	8	stack of 8 (if FP present), 8 (if SSE/MMX present)
<a href="#">Epiphany</a>		64 (per core)
<a href="#">Emotion Engine</a>	4	1 + 32
<a href="#">CUDA</a>	1	8/16/32/64/128
<a href="#">AVR microcontroller</a>	32	0
<a href="#">ARM 64/32-bit<sup>[12]</sup></a>	31	32
<a href="#">ARM 32-bit</a>	14	Varies (up to 32)
<a href="#">Alpha</a>	31	31
<a href="#">8080<sup>[3]</sup></a>	1 accumulator, 6 others	0
<a href="#">8008<sup>[2]</sup></a>	1 accumulator, 6 others	0
<a href="#">6502</a>	1	0
<a href="#">65002</a>	1	0
<a href="#">4004<sup>[1]</sup></a>	1 accumulator, 16 others	0
<a href="#">16-bit x86<sup>[4]</sup></a>	8	stack of 8 (if FP present)

Source: [https://en.wikipedia.org/wiki/Processor\\_register](https://en.wikipedia.org/wiki/Processor_register)

# Summary

- Computation
  - Performed by combinational logic
  - Computes Boolean functions
  - Continuously reacts to input changes
- Storage
  - Registers
    - Hold single words
    - Loaded as clock rises
  - Random-access memories
    - Hold multiple words
    - Possible multiple read or write ports
    - Read word when address input changes
    - Write word as clock rises

# CS:APP Chapter 4

## Computer Architecture

### Sequential Implementation – I

# Y86-64 Instruction Set #1

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Y86-64 Instruction Set #2

## Byte

halt

0	0
---	---

nop

1	0
---	---

cmoveXX rA, rB

2	fn	rA	rB
---	----	----	----

irmovq V, rB

3	0	F	rB	V
---	---	---	----	---

rrmovq rA, D(rB)

4	0	rA	rB	D
---	---	----	----	---

mrrmovq D(rB), rA

5	0	rA	rB	D
---	---	----	----	---

OPq rA, rB

6	fn	rA	rB
---	----	----	----

jXX Dest

7	fn	Dest
---	----	------

call Dest

8	0	Dest
---	---	------

ret

9	0
---	---

pushq rA

A	0	rA	F
---	---	----	---

popq rA

B	0	rA	F
---	---	----	---

rrmovq	7	0
cmovele	7	1
cmovl	7	2
cmove	7	3
cmovne	7	4
cmovge	7	5
cmovg	7	6

# Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Diagram illustrating the Y86-64 instruction set. The first 6 bytes (0-5) are fixed, while bytes 6-9 are variable. The variable bytes are color-coded: yellow for the function code (fn), pink for registers (rA, rB), and light blue for memory addresses (V, D, Dest).

Instructions shown:

- halt
- nop
- cmoveXX rA, rB
- irmovq V, rB
- rmmovq rA, D(rB)
- mrmovq D(rB), rA
- OPq rA, rB
- jXX Dest
- call Dest
- ret
- pushq rA
- popq rA

OPq rA, rB is expanded to show its variants:

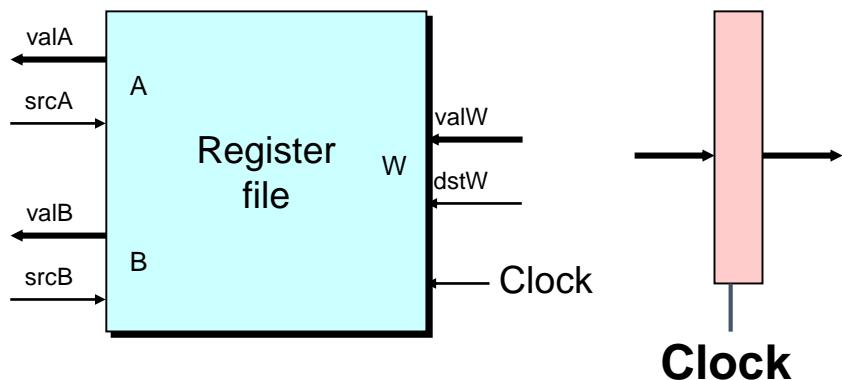
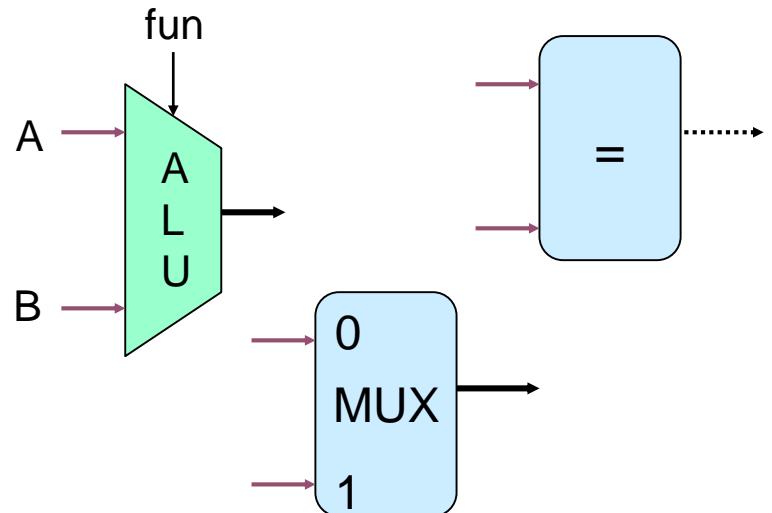
- addq (fn 6, rA 0, rB 0)
- subq (fn 6, rA 0, rB 1)
- andq (fn 6, rA 0, rB 2)
- xorq (fn 6, rA 0, rB 3)

# Y86-64 Instruction Set #4

Byte	0	1	2	3	4	5	6	7	
halt	0	0							
nop	1	0							
cmoveXX rA, rB	2	fn	rA	rB					
irmovq V, rB	3	0	F	rB	V				
rmmovq rA, D(rB)	4	0	rA	rB	D				
mrmovq D(rB), rA	5	0	rA	rB	D				
OPq rA, rB	6	fn	rA	rB					
jXX Dest	7	fn			Dest				
call Dest	8	0			Dest				
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					
jmp	7	0							
jle	7	1							
jl	7	2							
je	7	3							
jne	7	4							
jge	7	5							
jg	7	6							

# Building Blocks

- Combinational Logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control
- Storage Elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises



# Hardware Control Language

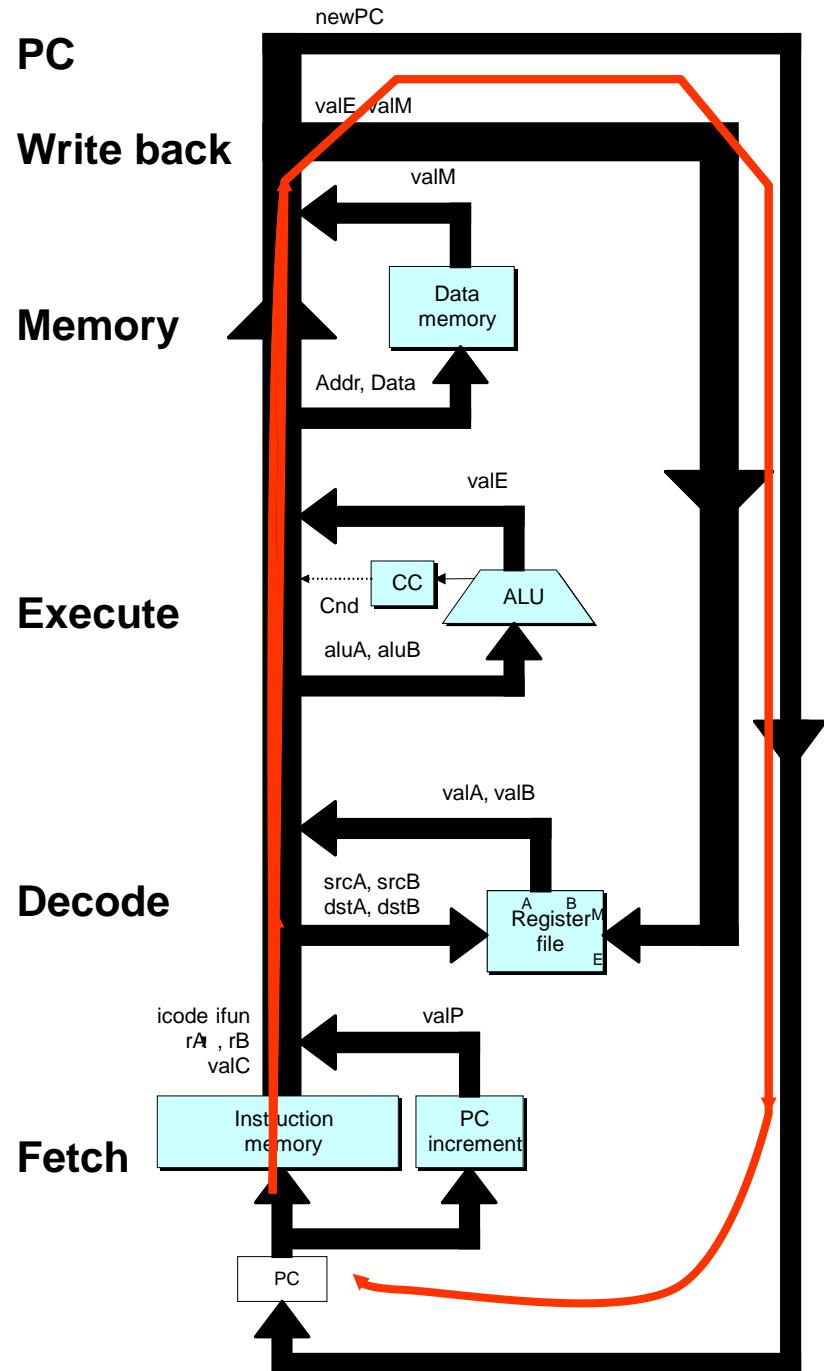
- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify
- Data Types
  - `bool`: Boolean
    - `a, b, c, ...`
  - `int`: words
    - `A, B, C, ...`
    - Does not specify word size---bytes, 32-bit words, ...
- Statements
  - `bool a = bool-expr ;`
  - `int A = int-expr ;`

# HCL Operations

- Classify by type of value returned
- Boolean Expressions
  - Logic Operations
    - `a && b, a || b, !a`
  - Word Comparisons
    - `A == B, A != B, A < B, A <= B, A >= B, A > B`
  - Set Membership
    - `A in { B, C, D }`
      - Same as `A == B || A == C || A == D`
- Word Expressions
  - Case expressions
    - `[ a : A; b : B; c : C ]`
    - Evaluate test expressions `a, b, c, ...` in sequence
    - Return word expression `A, B, C, ...` for first successful test

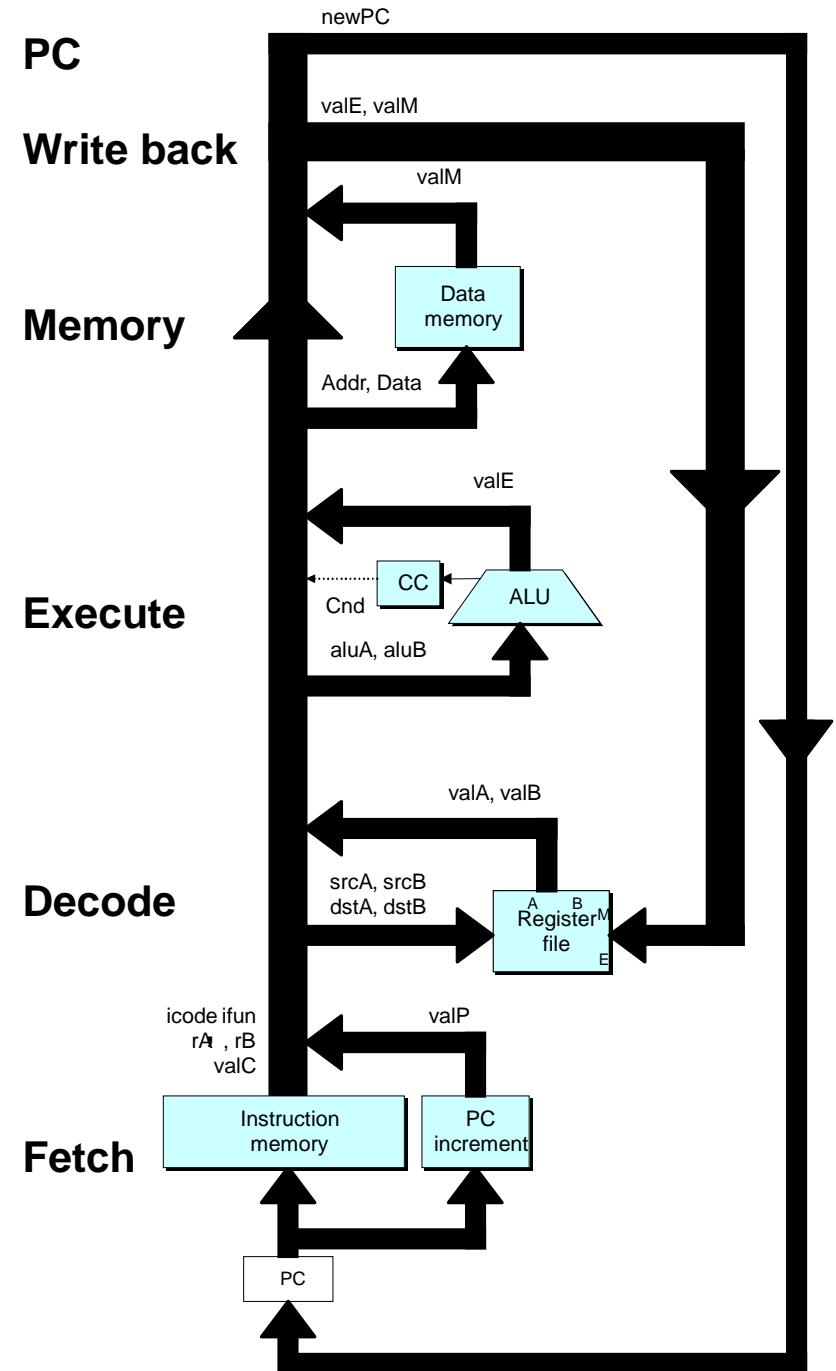
# SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

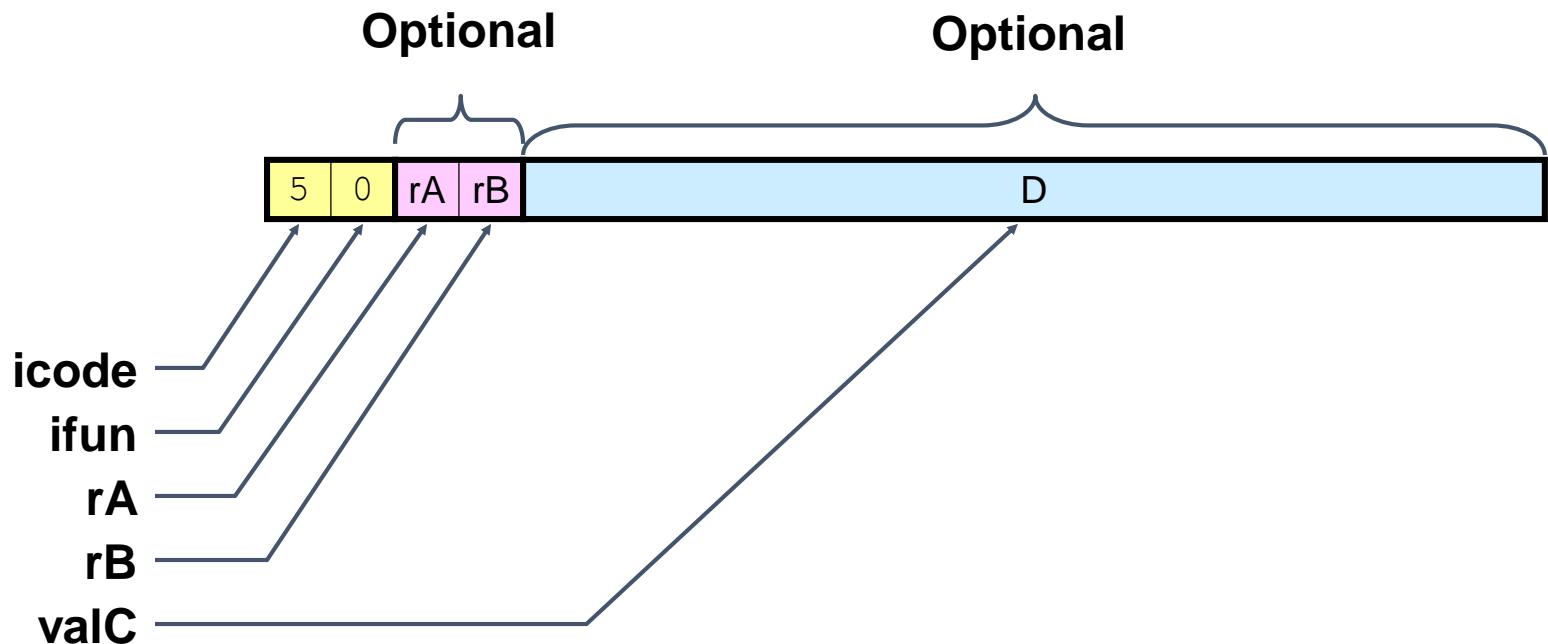


# SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



# Instruction Decoding



- **Instruction Format**

- Instruction byte
- Optional register byte
- Optional constant word

icode:ifun  
rA:rB  
valC

# Executing Arith./Logical Operation



- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

# Stage Computation: Arith/Log. Ops

	$OPq\ rA, rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$	Read instruction byte Read register byte Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
Execute	$valE \leftarrow valB\ OP\ valA$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	$R[rB] \leftarrow valE$	Write back result
PC update	$PC \leftarrow valP$	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

# Executing `rmmovq`

`rmmovq rA, D(rB)`



- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10

# Stage Computation: `rmmovq`

<code>rmmovq rA, D(rB)</code>	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC}+10$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$
Write back	
PC update	$\text{PC} \leftarrow \text{valP}$

- Use ALU for address computation

# Executing popq



- Fetch
  - Read 2 bytes
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 8
- Memory
  - Read from old stack pointer
- Write back
  - Update stack pointer
  - Write result to register
- PC Update
  - Increment PC by 2

# Stage Computation: popq

popq rA		
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $rA:rB \leftarrow M_1[\text{PC}+1]$	Read instruction byte Read register byte
	$\text{valP} \leftarrow \text{PC}+2$	Compute next PC
Decode	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	Read stack pointer Read stack pointer
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Read from stack
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[rA] \leftarrow \text{valM}$	Update stack pointer Write back result
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

# Executing Conditional Moves



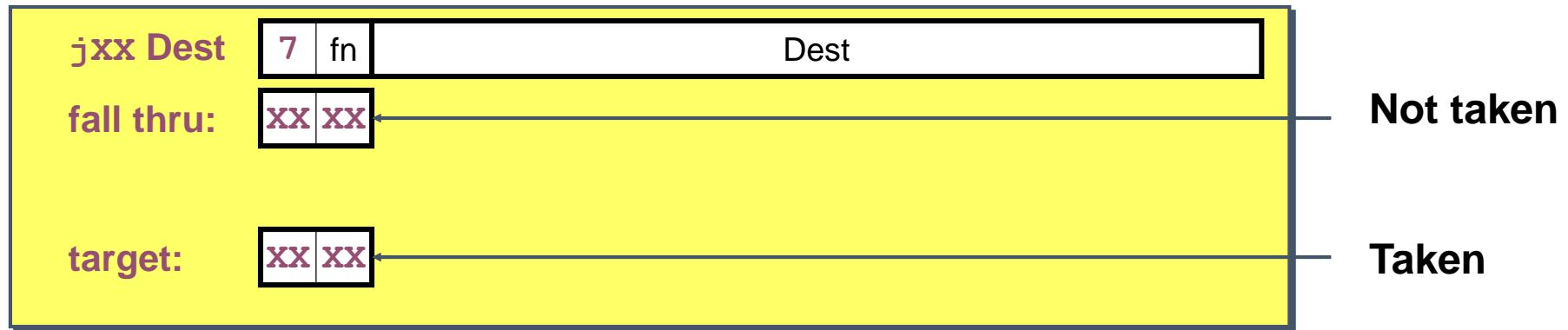
- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - If !cnd, then set destination register to 0xF
- Memory
  - Do nothing
- Write back
  - Update register (or not)
- PC Update
  - Increment PC by 2

# Stage Computation: Cond. Move

cmovXX rA, rB		
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$	Read instruction byte Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow 0$	Read operand A
Execute	$valE \leftarrow valB + valA$ If ! Cond(CC,ifun) $rB \leftarrow 0xF$	Pass valA through ALU (Disable register update)
Memory		
Write back	$R[rB] \leftarrow valE$	Write back result
PC update	$PC \leftarrow valP$	Update PC

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
  - If condition codes & move condition indicate no move

# Executing Jumps



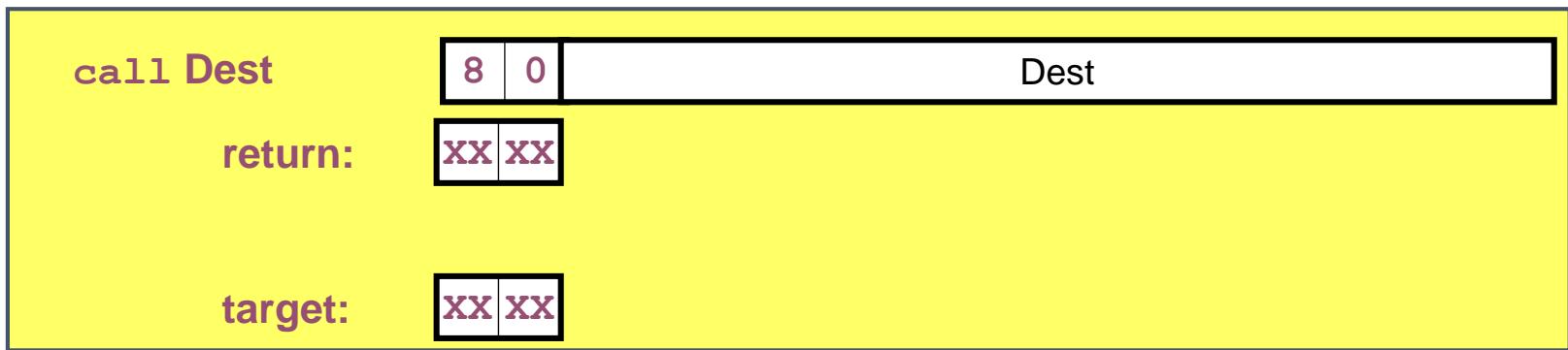
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

# Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$	Read instruction byte Read destination address Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?
Memory		
Write		
back		
PC update	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

# Executing call



- Fetch

- Read 9 bytes
- Increment PC by 9

- Decode

- Read stack pointer

- Execute

- Decrement stack pointer by 8

- Memory

- Write incremented PC to new value of stack pointer

- Write back

- Update stack pointer

- PC Update

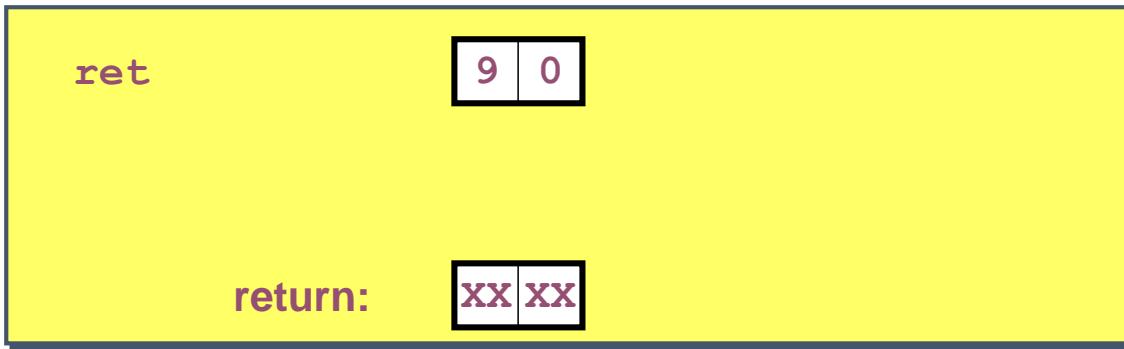
- Set PC to Dest

# Stage Computation: call

	call Dest	
Fetch	$icode:ifun \leftarrow M_1[PC]$  $valC \leftarrow M_8[PC+1]$ $valP \leftarrow PC+9$	Read instruction byte  Read destination address Compute return point
Decode	$valB \leftarrow R[%rsp]$	Read stack pointer
Execute	$valE \leftarrow valB + -8$	Decrement stack pointer
Memory	$M_8[valE] \leftarrow valP$	Write return value on stack
Write back	$R[%rsp] \leftarrow valE$	Update stack pointer
PC update	$PC \leftarrow valC$	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

# Executing `ret`



- Fetch
  - Read 1 byte
- Decode
  - Read stack pointer
- Execute
  - Increment stack pointer by 8
- Memory
  - Read return address from old stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to return address

# Stage Computation: ret

ret	
Fetch	$icode:ifun \leftarrow M_1[PC]$
Decode	$valA \leftarrow R[%rsp]$ $valB \leftarrow R[%rsp]$
Execute	$valE \leftarrow valB + 8$
Memory	$valM \leftarrow M_8[valA]$
Write back	$R[%rsp] \leftarrow valE$
PC update	$PC \leftarrow valM$

- Use ALU to increment stack pointer
- Read return address from memory

# Computation Steps

		$OPq\ rA, rB$	
Fetch	icode,ifun	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	rA,rB	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	valC		[Read constant word]
	valP	$valP \leftarrow PC+2$	Compute next PC
Decode	valA, srcA	$valA \leftarrow R[rA]$	Read operand A
	valB, srcB	$valB \leftarrow R[rB]$	Read operand B
Execute	valE	$valE \leftarrow valB\ OP\ valA$	Perform ALU operation
	Cond code	Set CC	Set/use cond. code reg
Memory	valM		[Memory read/write]
Write back	dstE	$R[rB] \leftarrow valE$	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	$PC \leftarrow valP$	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

# Computation Steps

		call Dest	
Fetch	icode,ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	rA,rB		[Read register byte]
	valC	$\text{valC} \leftarrow M_8[\text{PC+1}]$	Read constant word
	valP	$\text{valP} \leftarrow \text{PC+9}$	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	$\text{valB} \leftarrow R[\%rsp]$	Read operand B
Execute	valE	$\text{valE} \leftarrow \text{valB} + -8$	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	$M_8[\text{valE}] \leftarrow \text{valP}$	Memory read/write
Write back	dstE	$R[\%rsp] \leftarrow \text{valE}$	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	$\text{PC} \leftarrow \text{valC}$	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

# Computed Values

- Fetch

icode	Instruction code
ifun	Instruction function
rA	Instr. Register A
rB	Instr. Register B
valC	Instruction constant
valP	Incremented PC

- Decode

srcA	Register ID A
srcB	Register ID B
dstE	Destination Register E
dstM	Destination Register M
valA	Register value A
valB	Register value B

- Execute

• valE	ALU result
• Cnd	Branch/move flag

- Memory

• valM	Value from memory
--------	-------------------

# CS:APP Chapter 4

## Computer Architecture

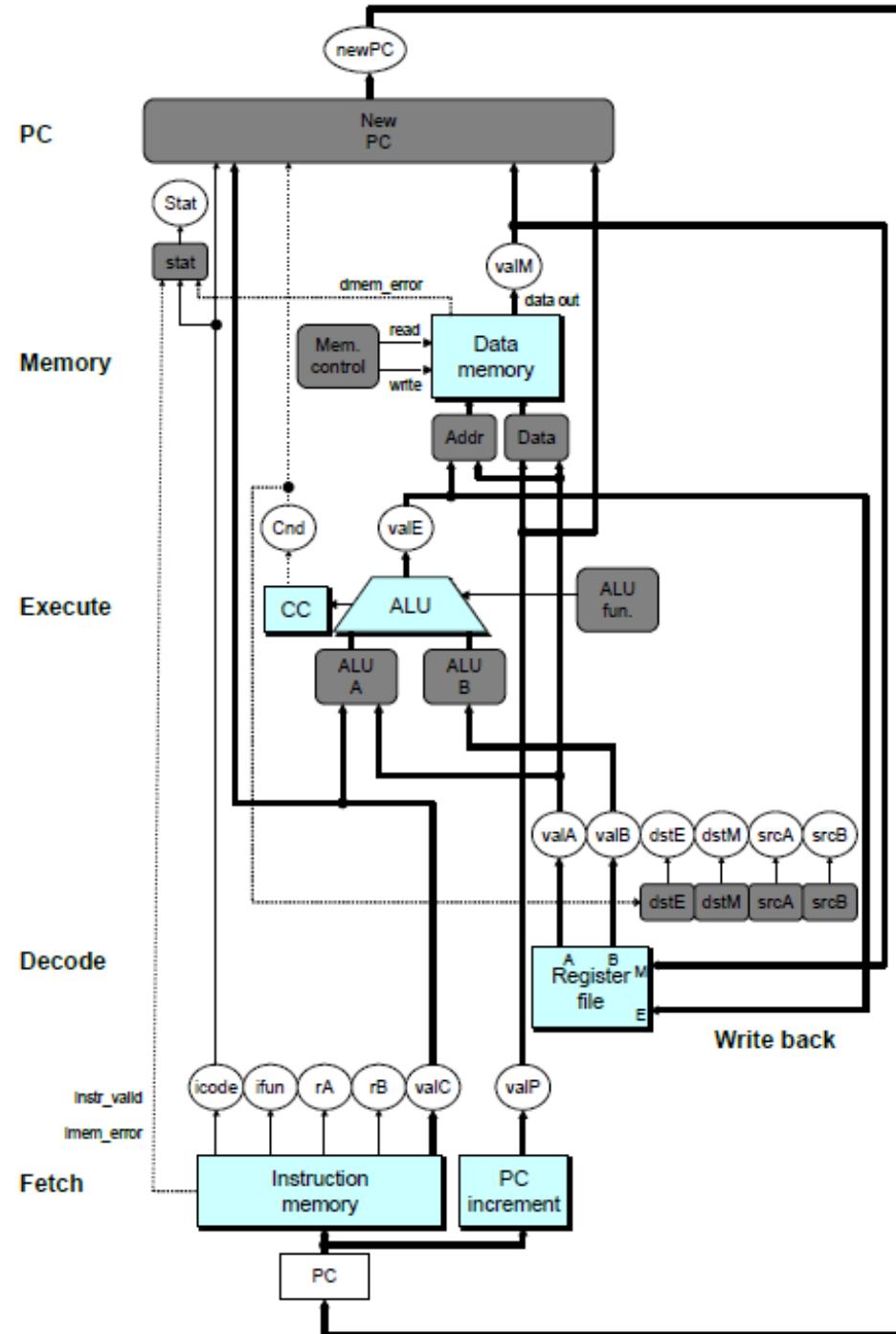
### Sequential

### Implementation - II

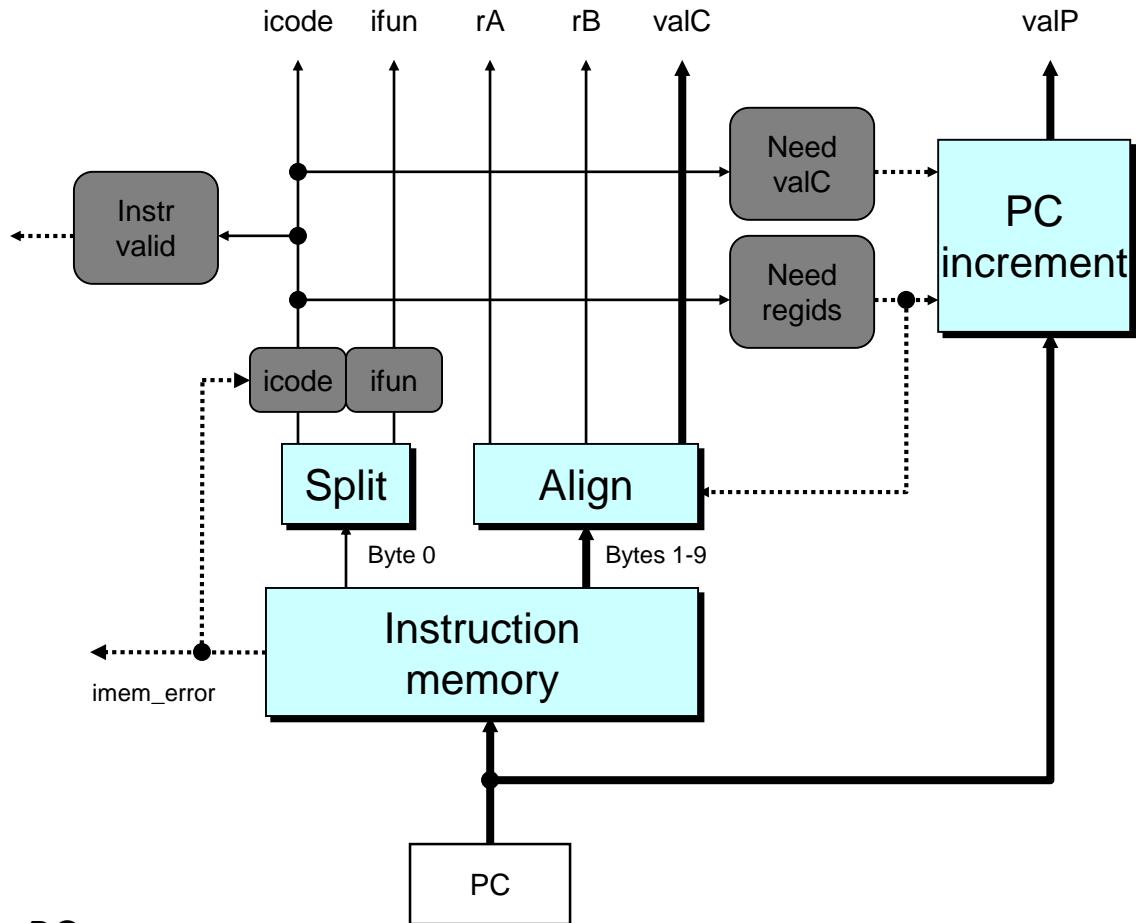
# SEQ Hardware

- Key

- Blue boxes: predefined hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
  - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



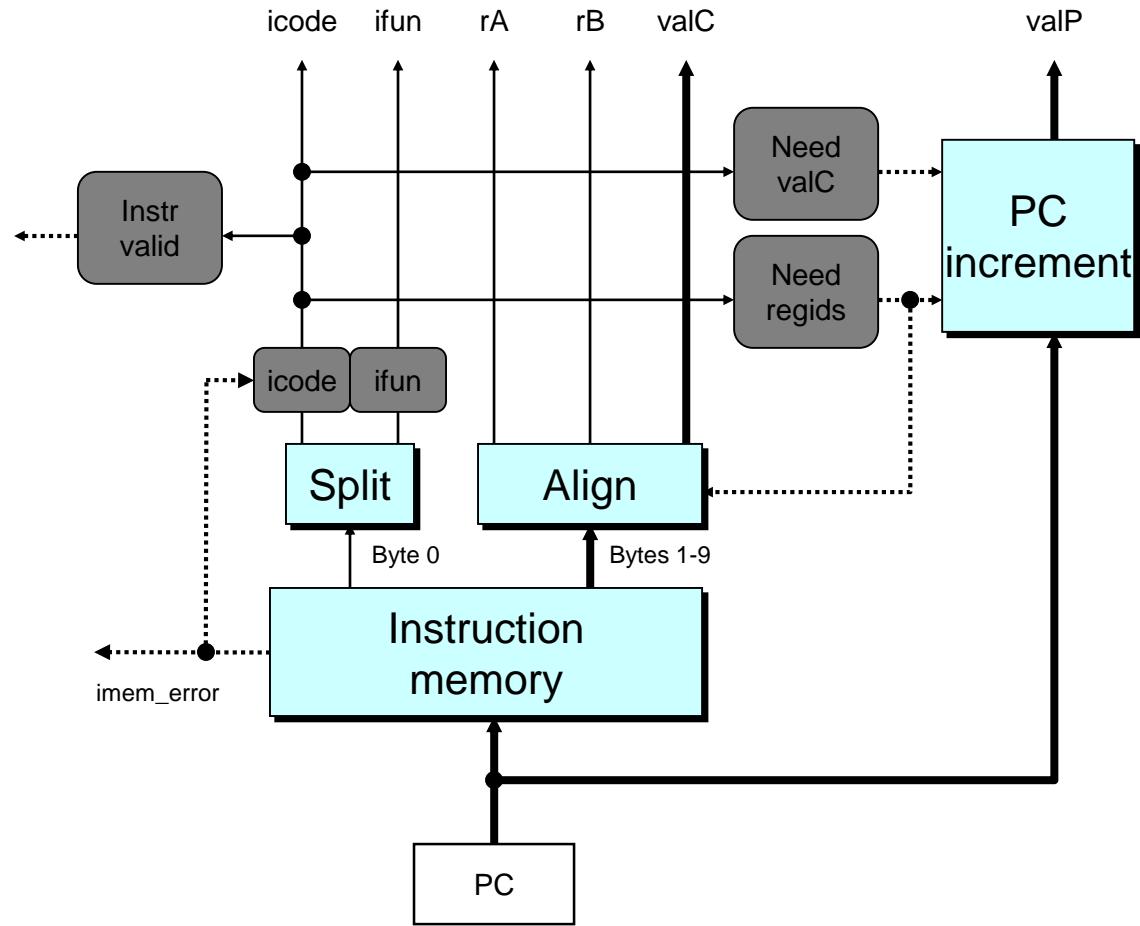
# Fetch Logic



- **Predefined Blocks**

- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
  - Signal invalid address
- Split: Divide instruction byte into iCode and iFun
- Align: Get fields for rA, rB, and valC

# Fetch Logic

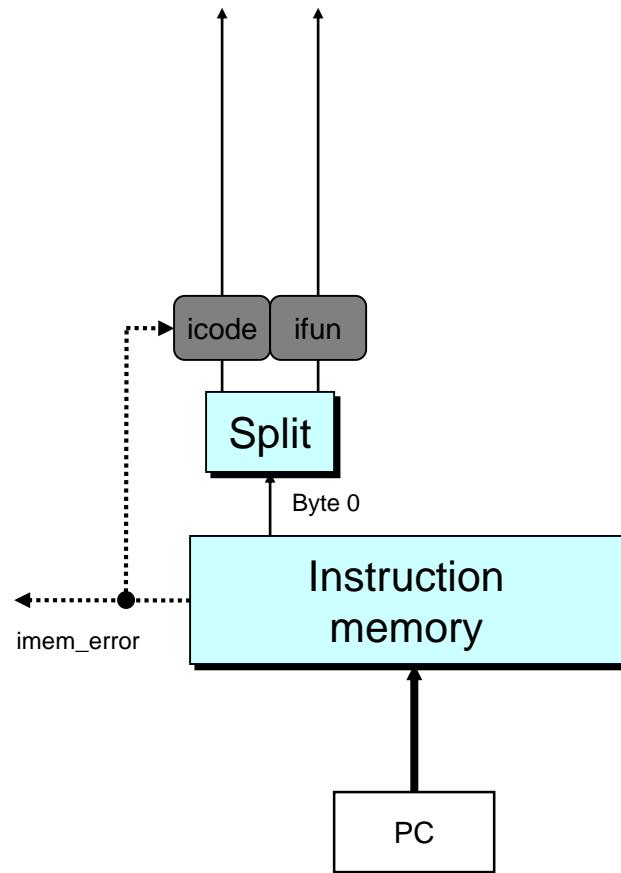


- Control Logic
  - Instr. Valid: Is this instruction valid?
  - iCode, Ifun: Generate no-op if invalid address
  - Need regids: Does this instruction have a register byte?
  - Need valC: Does this instruction have a constant word?

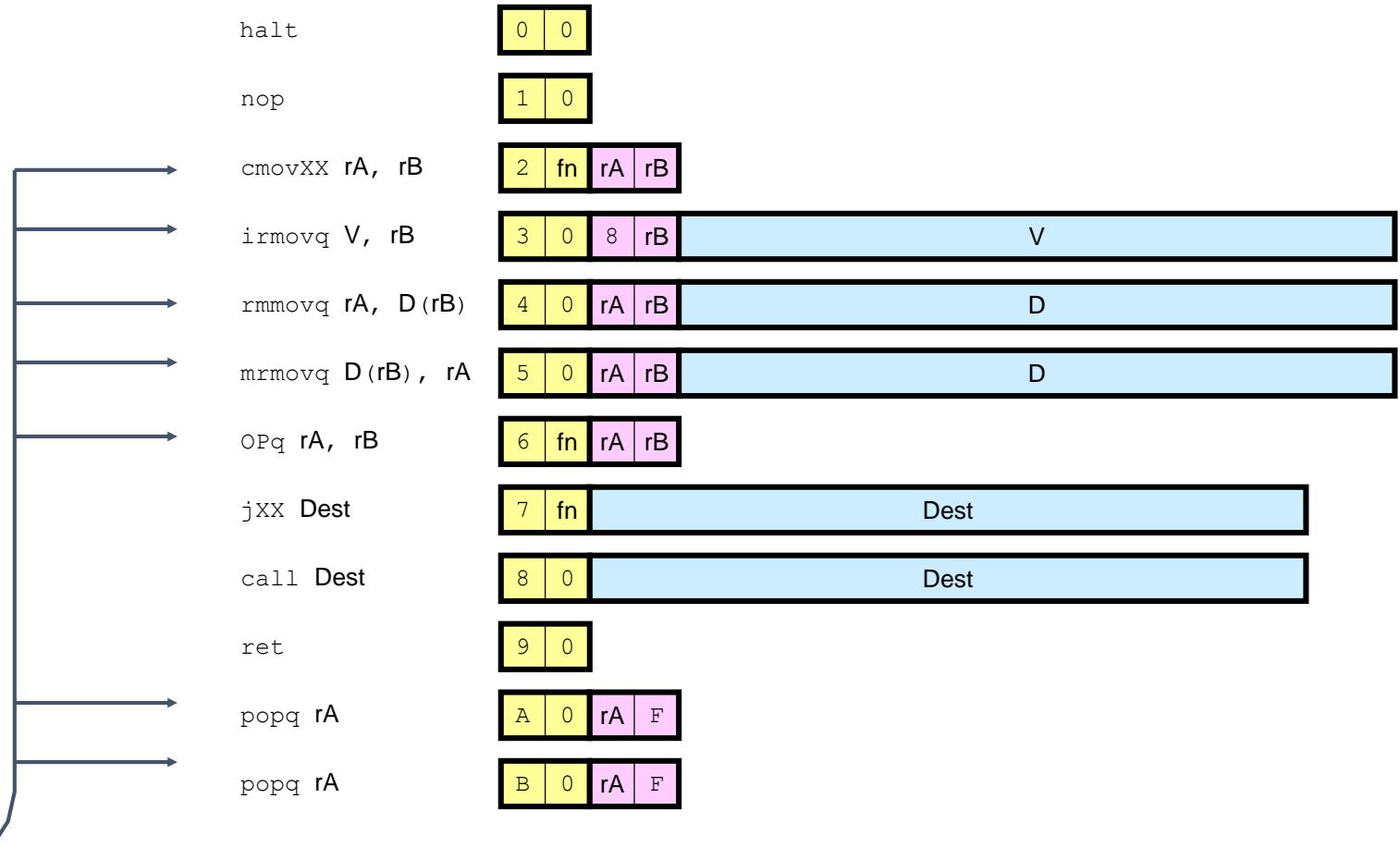
# Fetch Control Logic in HCL

```
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];

# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```



# Fetch Control Logic in HCL



```
bool need_regs =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPOQ,
               IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

```
bool instr_valid = icode in
{ INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
  IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPOQ };
```

# Decode Logic

## • Register File

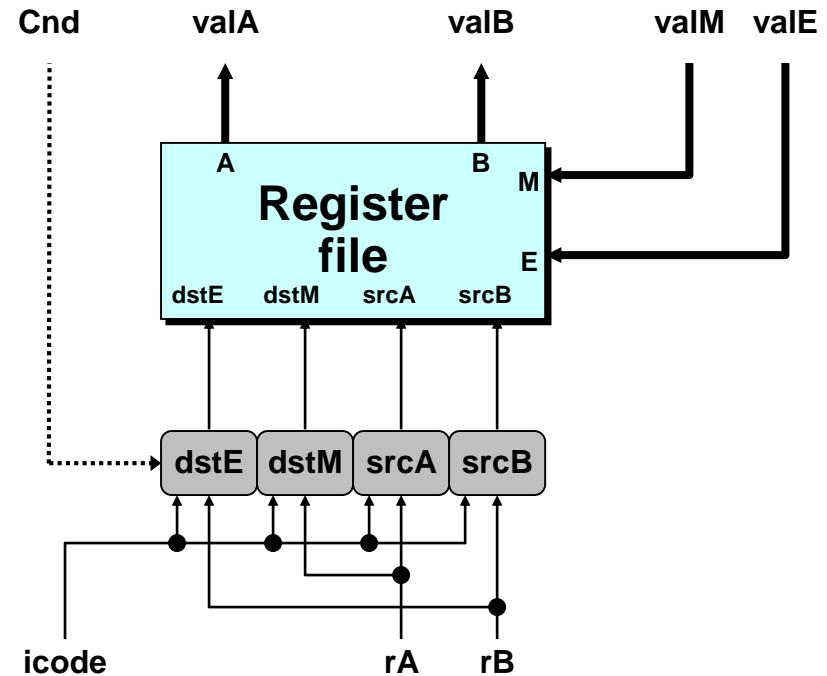
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

## Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

## Signals

- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage



# A Source

	OPq rA, rB	
Decode	valA $\leftarrow R[rA]$	Read operand A
	cmovXX rA, rB	
Decode	valA $\leftarrow R[rA]$	Read operand A
	rmmovq rA, D(rB)	
Decode	valA $\leftarrow R[rA]$	Read operand A
	popq rA	
Decode	valA $\leftarrow R[\%rsp]$	Read stack pointer
	jXX Dest	
Decode		No operand
	call Dest	
Decode		No operand
	ret	
Decode	valA $\leftarrow R[\%rsp]$	Read stack pointer

```
int srcA = [
    icode in { IRMMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # Don't need register
];
```

# E Destination

	<code>OPq rA, rB</code>	
<b>Write-back</b>	<code>R[rB] ← valE</code>	<b>Write back result</b>
	<code>cmoveXX rA, rB</code>	
<b>Write-back</b>	<code>R[rB] ← valE</code>	<b>Conditionally write back result</b>
	<code>rmmovq rA, D(rB)</code>	
<b>Write-back</b>		<b>None</b>
	<code>popq rA</code>	
<b>Write-back</b>	<code>R[%rsp] ← valE</code>	<b>Update stack pointer</b>
	<code>jXX Dest</code>	
<b>Write-back</b>		<b>None</b>
	<code>call Dest</code>	
<b>Write-back</b>	<code>R[%rsp] ← valE</code>	<b>Update stack pointer</b>
	<code>ret</code>	
<b>Write-back</b>	<code>R[%rsp] ← valE</code>	<b>Update stack pointer</b>

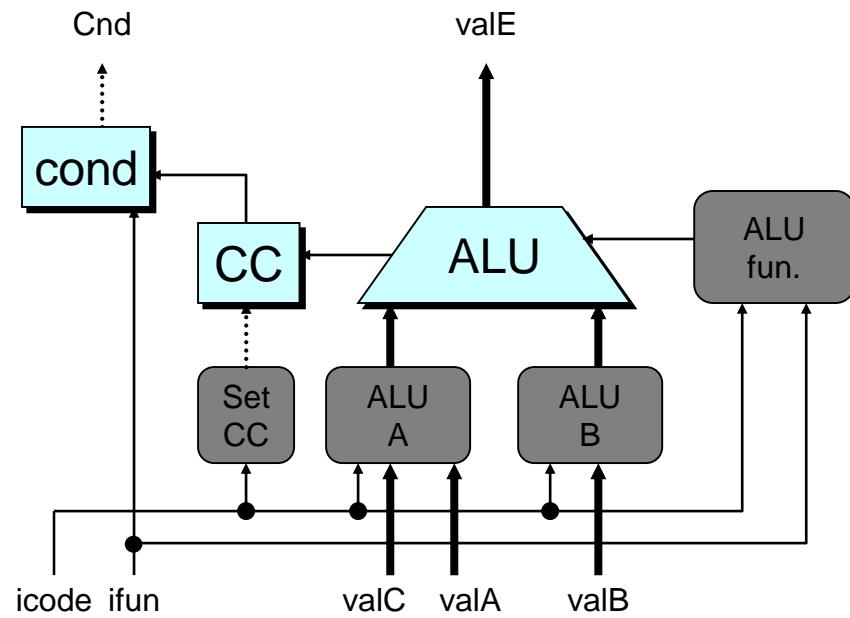
```

int dstE = [
    icode in { IRRMOVQ } && Cnd : rB;
    icode in { IIRMOVQ, IOPOQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE;  # Don't write any register
];

```

# Execute Logic

- Units
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
  - cond
    - Computes conditional jump/move flag
- Control Logic
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?



# ALU A Input

	<b>OPq rA, rB</b>	
Execute	$\text{valE} \leftarrow \text{valB OP valA}$	Perform ALU operation
	<b>cmoveXX rA, rB</b>	
Execute	$\text{valE} \leftarrow 0 + \text{valA}$	Pass valA through ALU
	<b>rmmovq rA, D(rB)</b>	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
	<b>popq rA</b>	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
	<b>jXX Dest</b>	
Execute		No operation
	<b>call Dest</b>	
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrement stack pointer
	<b>ret</b>	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer

```
int aluA = [
    icode in { IRRMOVQ, IOPQ } : valA,
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC,
    icode in { ICALL, IPUSHQ } : -8,
    icode in { IRET, IPOPQ } : 8,
    # Other instructions don't need ALU
];
```

# ALU Operation

	OPI rA, rB	
Execute	$\text{valE} \leftarrow \text{valB OP valA}$	Perform ALU operation
	cmovXX rA, rB	
Execute	$\text{valE} \leftarrow 0 + \text{valA}$	Pass valA through ALU
	rmmovl rA, D(rB)	
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
	popq rA	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer
	jXX Dest	
Execute		No operation
	call Dest	
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrement stack pointer
	ret	
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Increment stack pointer

```
int alufun = [
    icode == IOPQ : ifun;
    1 : ALUADD;
];
```

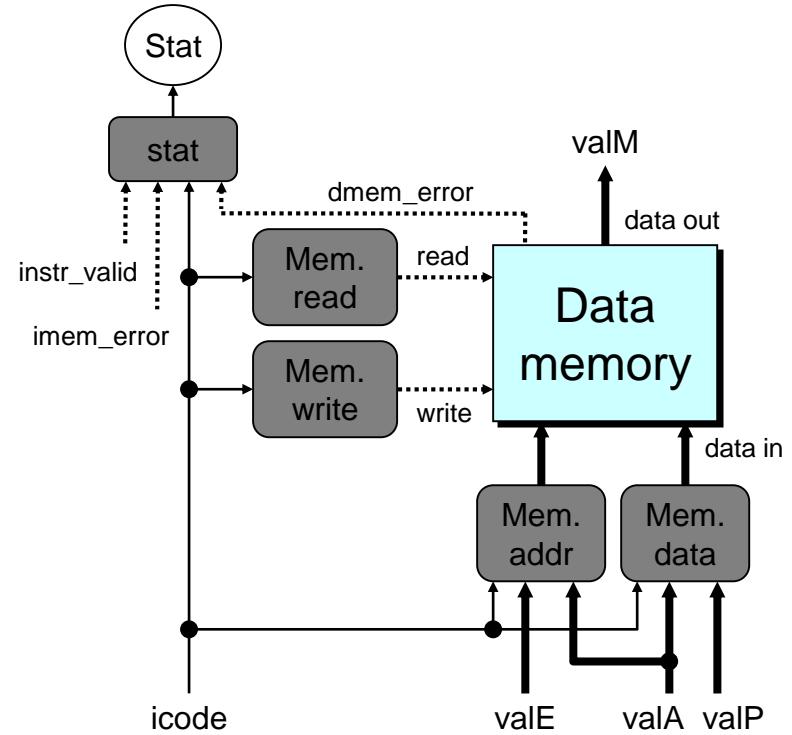
# Memory Logic

- **Memory**

- Reads or writes memory word

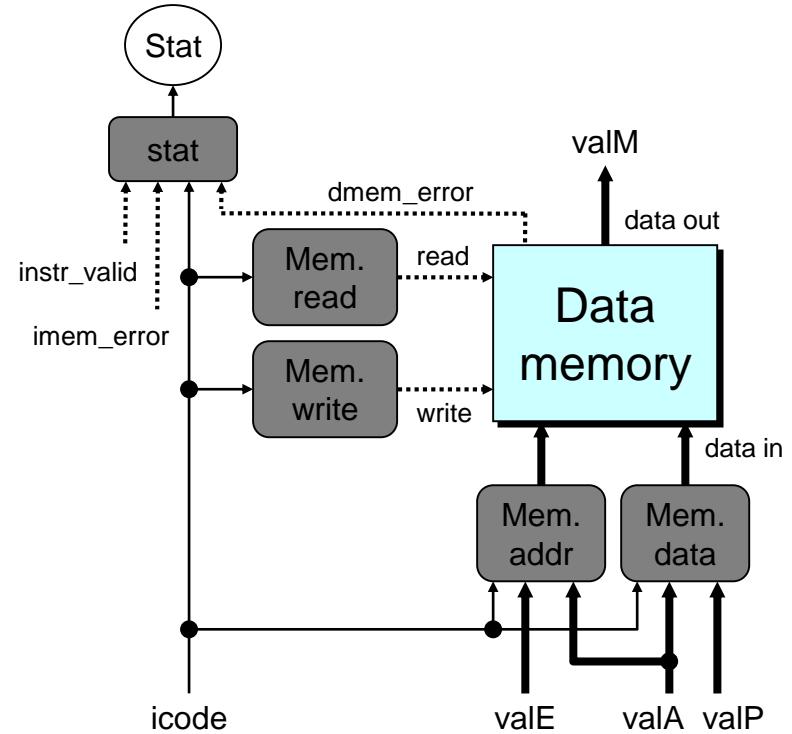
- **Control Logic**

- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



# Instruction Status

- Control Logic
  - stat: What is instruction status?



```
## Determine instruction status
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```

# Memory Address

	OPq rA, rB	
Memory		No operation
	rmmovq rA, D(rB)	
Memory	M <sub>8</sub> [valE] ← valA	Write value to memory
	popq rA	
Memory	valM ← M <sub>8</sub> [valA]	Read from stack
	jXX Dest	
Memory		No operation
	call Dest	
Memory	M <sub>8</sub> [valE] ← valP	Write return value on stack
	ret	
Memory	valM ← M <sub>8</sub> [valA]	Read return address

```
int mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
```

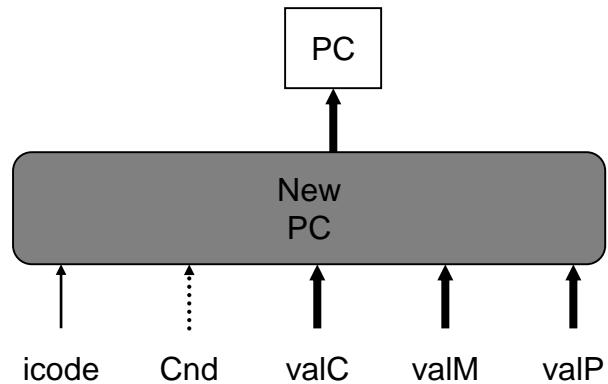
# Memory Read

OPq rA, rB	Memory	No operation
rmmovq rA, D(rB)	Memory	M <sub>8</sub> [valE] ← valA Write value to memory
popq rA	Memory	valM ← M <sub>8</sub> [valA] Read from stack
jXX Dest	Memory	No operation
call Dest	Memory	M <sub>8</sub> [valE] ← valP Write return value on stack
ret	Memory	valM ← M <sub>8</sub> [valA] Read return address

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```

# PC Update Logic

- New PC
  - Select next value of PC

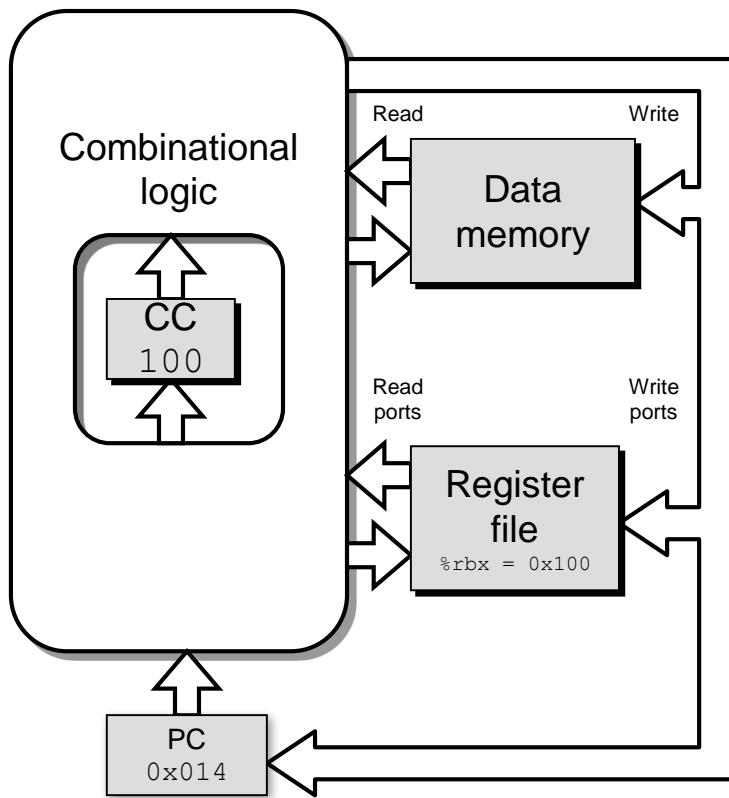


# PC Update

	OPq rA, rB	
PC update	PC $\leftarrow$ valP	Update PC
	rmmovq rA, D(rB)	
PC update	PC $\leftarrow$ valP	Update PC
	popq rA	
PC update	PC $\leftarrow$ valP	Update PC
	jXX Dest	
PC update	PC $\leftarrow$ Cnd ? valC : valP	Update PC
	call Dest	
PC update	PC $\leftarrow$ valC	Set PC to destination
	ret	
PC update	PC $\leftarrow$ valM	Set PC to return address

```
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Cnd : valC;
    icode == IRET : valM;
    1 : valP;
];
```

# SEQ Operation



- State

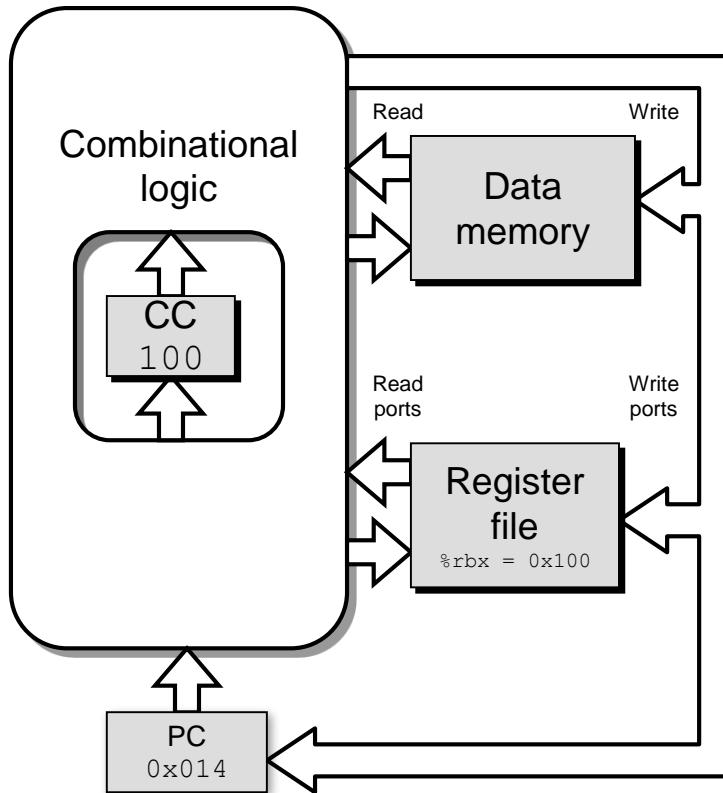
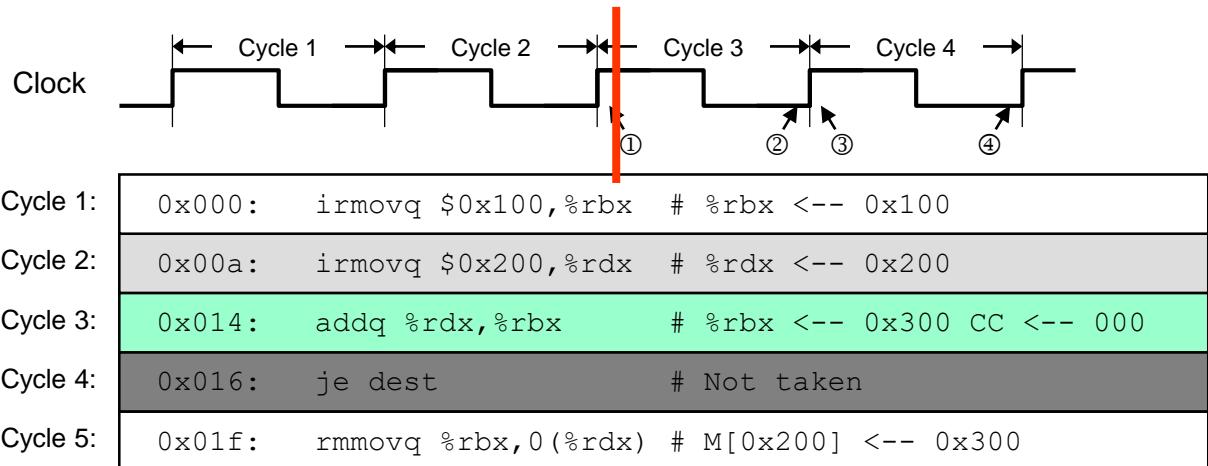
- PC register
- Cond. Code register
- Data memory
- Register file

*All updated as clock rises*

- Combinational Logic

- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

# SEQ Operation #2

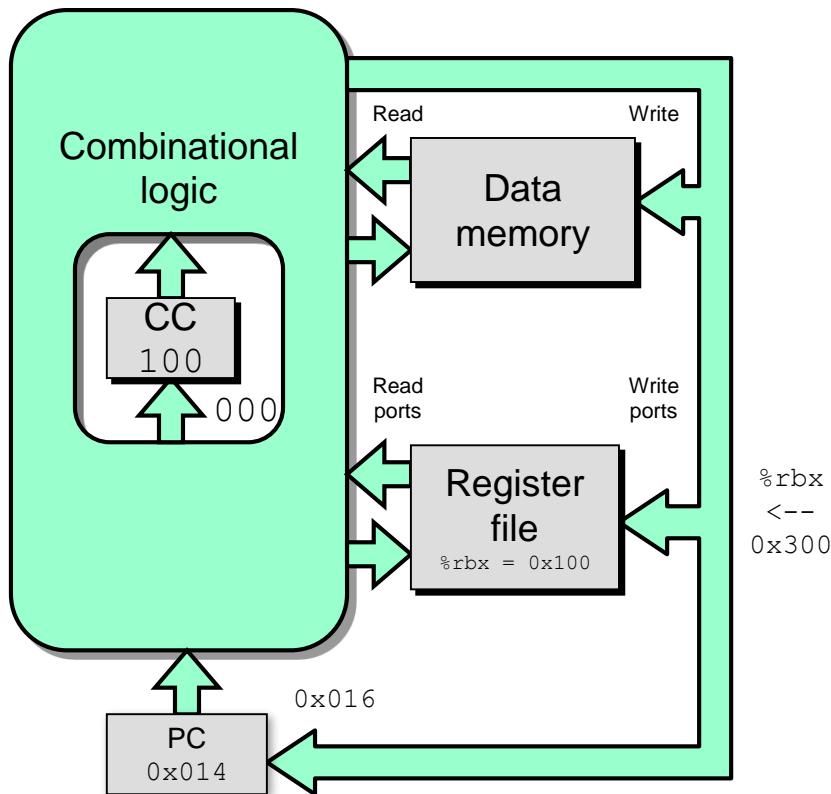


- state set according to second irmovq instruction
- combinational logic starting to react to state changes

# SEQ Operation #3

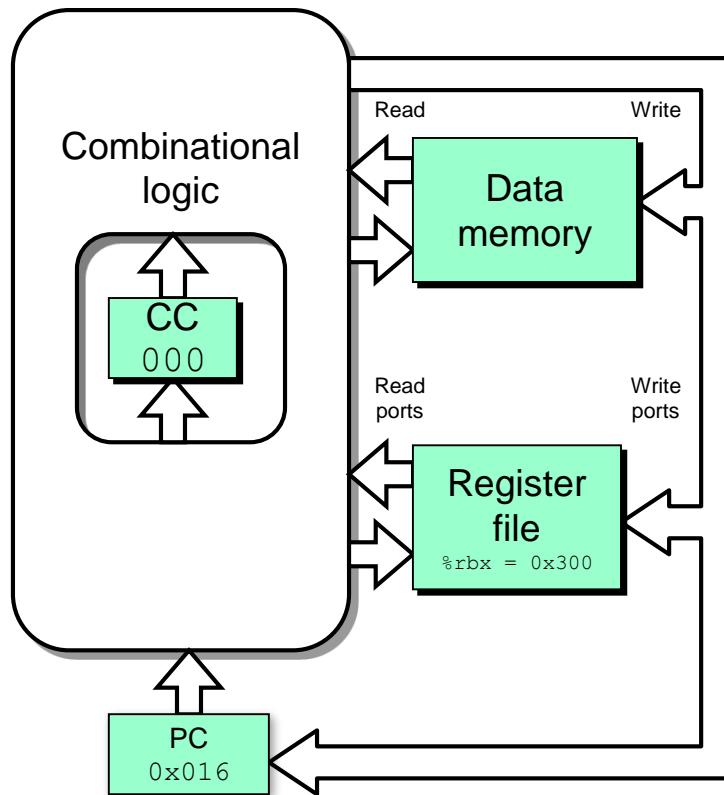
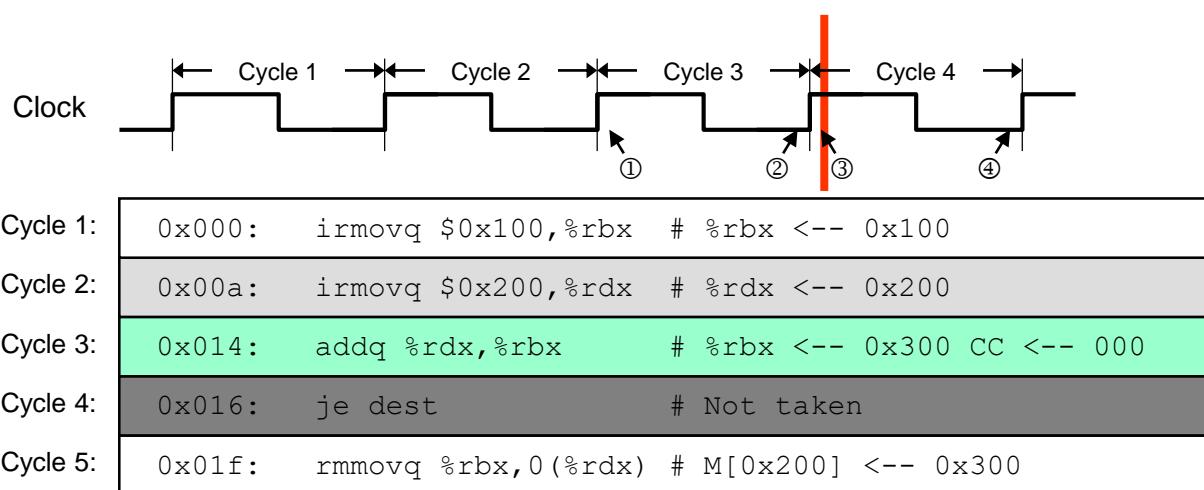
Clock

Cycle	Instruction	Description
Cycle 1:	0x000: irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a: irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014: addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016: je dest	# Not taken
Cycle 5:	0x01f: rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



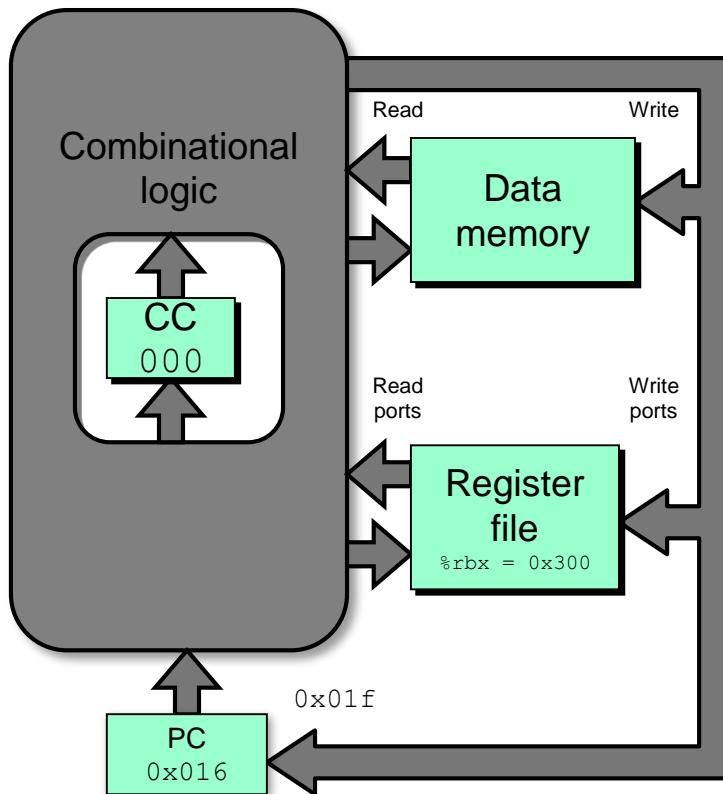
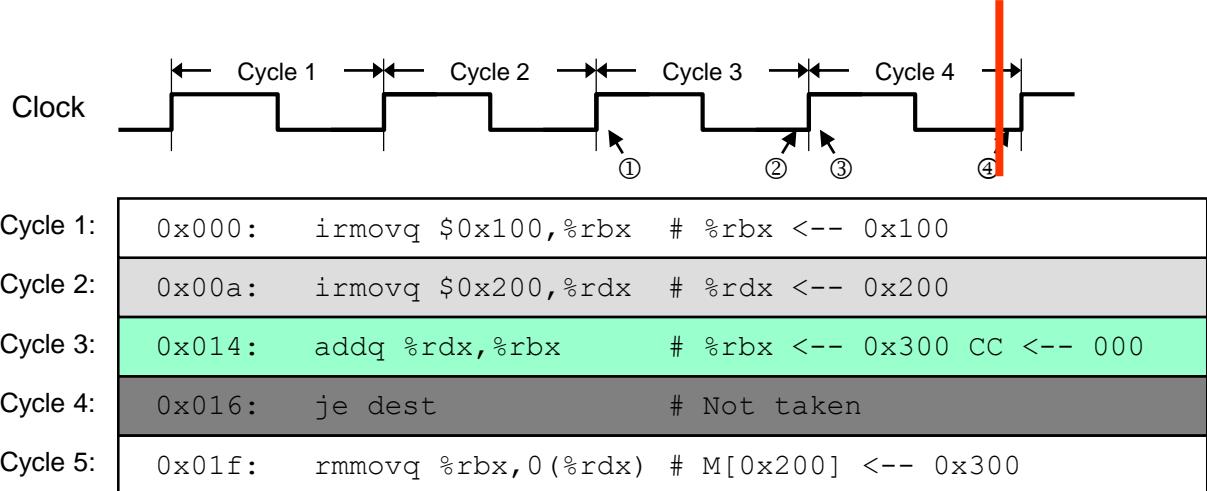
- state set according to second irmovq instruction
- combinational logic generates results for addq instruction

# SEQ Operation #4



- state set according to addq instruction
- combinational logic starting to react to state changes

# SEQ Operation #5



- state set according to addq instruction
- combinational logic generates results for je instruction

# SEQ Summary

- **Implementation**

- Express every instruction as series of simple steps
- Follow same general flow for each instruction type
- Assemble registers, memories, predesigned combinational blocks
- Connect with control logic

- **Limitations**

- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle

# Computer Architecture: Pipelined Implementation - I

CENG331 - Computer Organization

Instructor:

Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

Slides 24-28 adapted from the slides of the textbook: D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3<sup>rd</sup> Edition

# Overview

- General Principles of Pipelining
  - Goal
  - Difficulties
- Creating a Pipelined Y86-64 Processor
  - Rearranging SEQ
  - Inserting pipeline registers
  - Problems with data and control hazards

# Real-World Pipelines: Car Washes

**Sequential**



**Parallel**



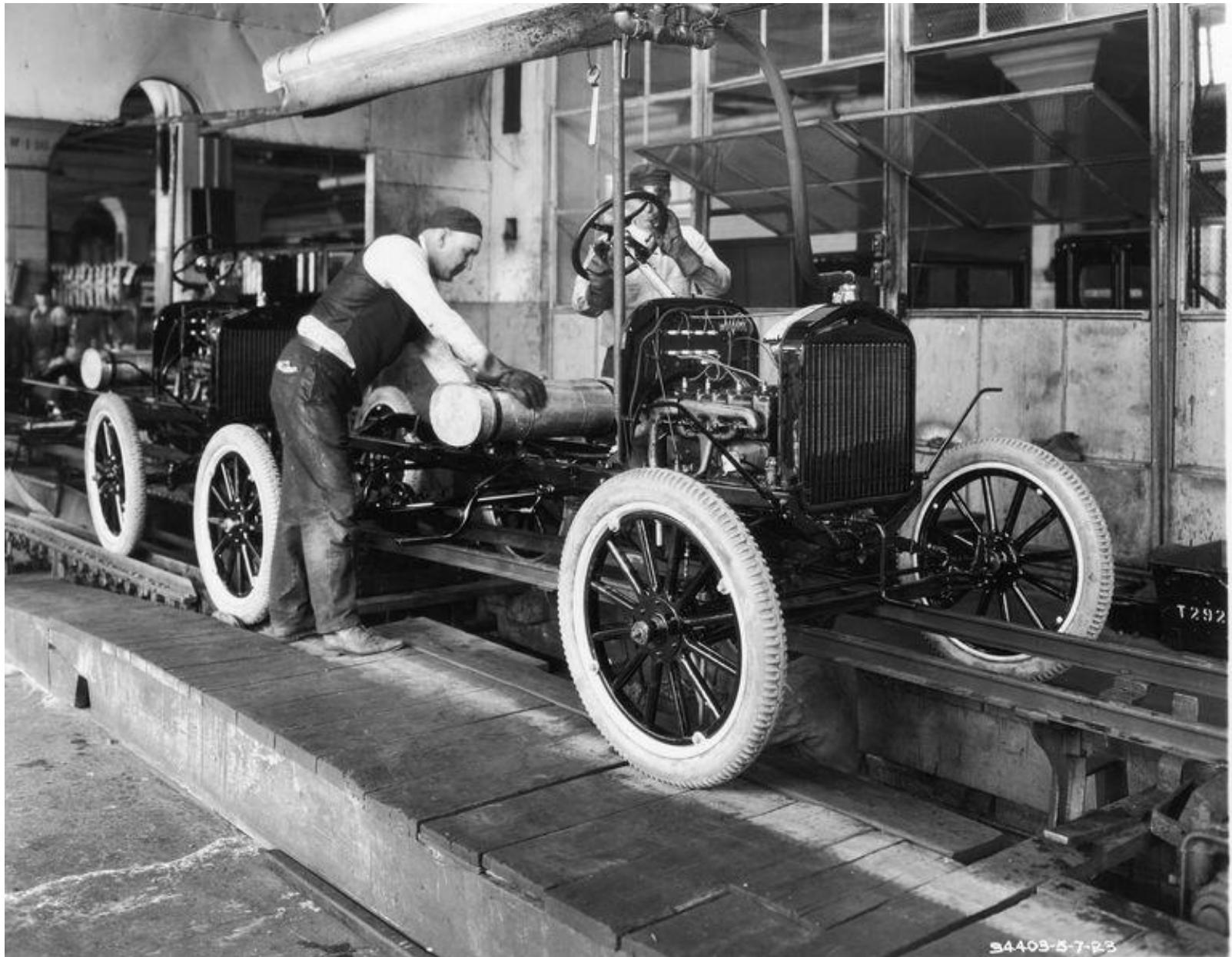
**Pipelined**



- **Idea**

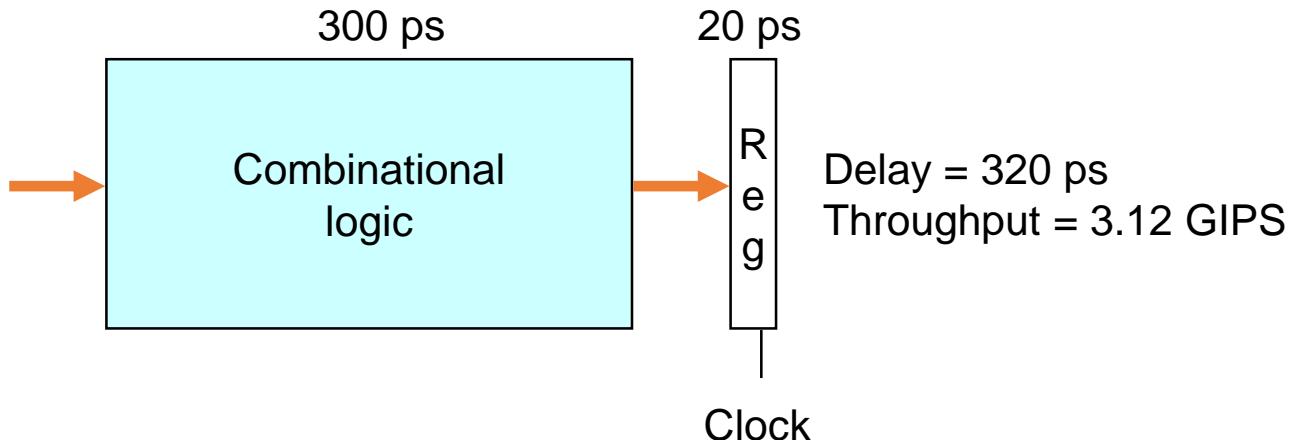
- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

# Car production: Ford assembly line



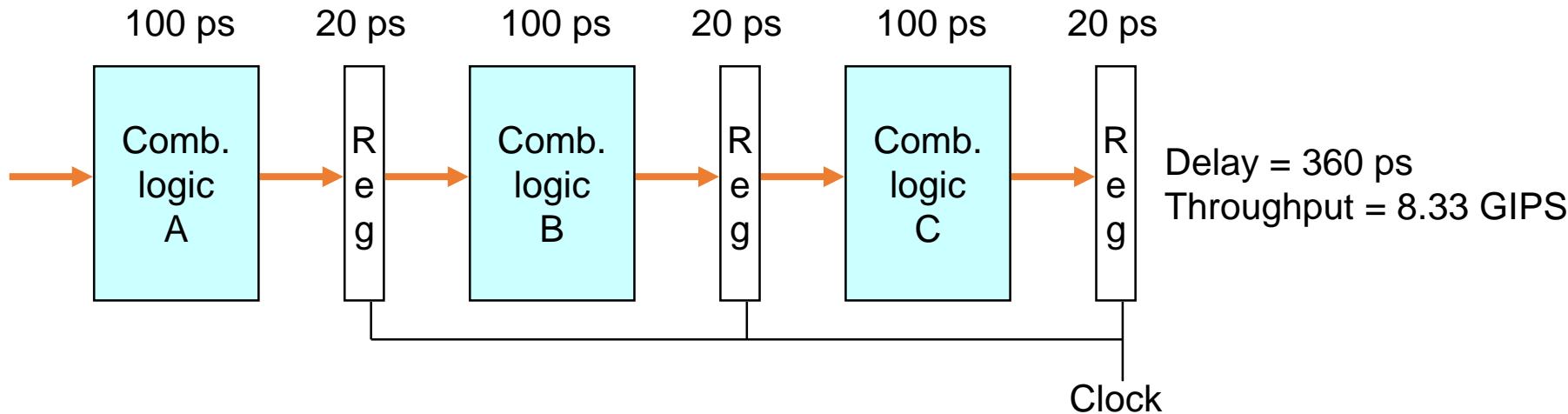
A Ford assembly line with a worker attaching a gas tank. (circa 1923). (Photo by Fotosearch/Getty Images)

# Computational Example



- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

## 3-Way Pipelined Version

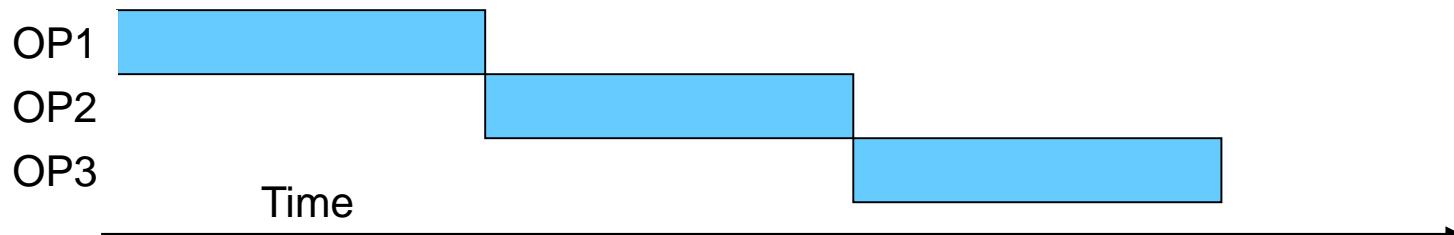


- **System**

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish

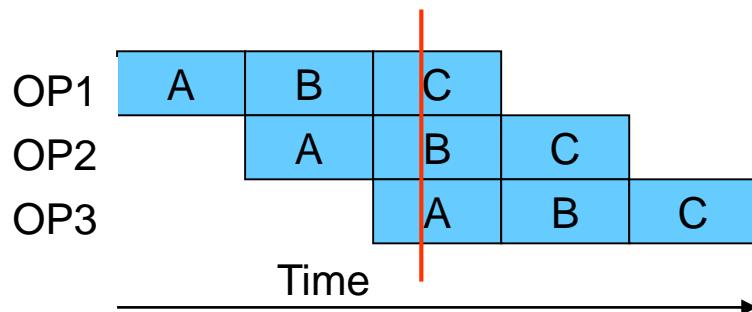
# Pipeline Diagrams

- Unpipelined



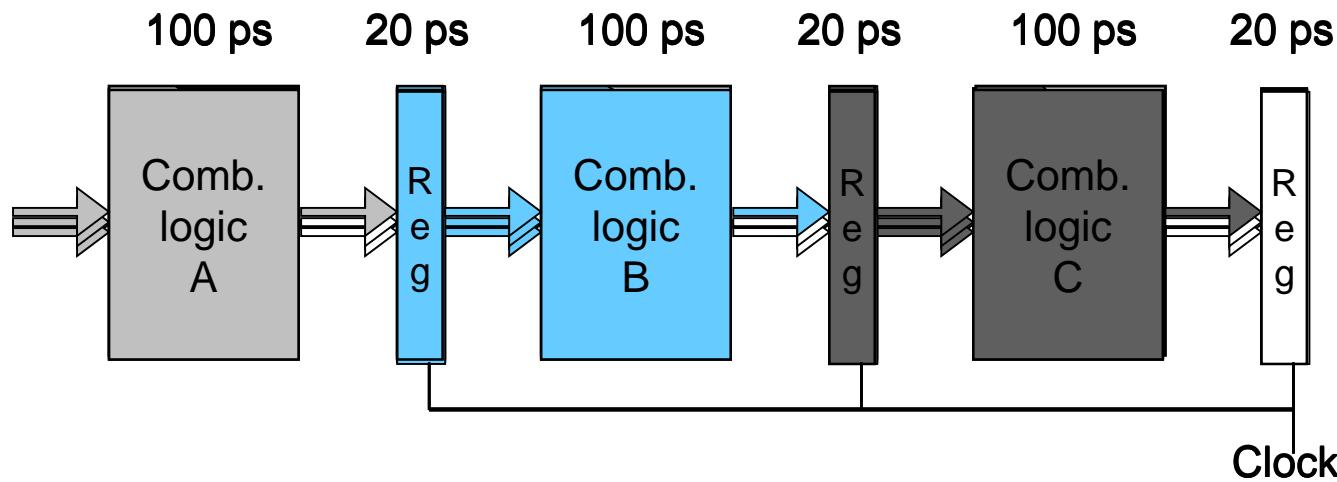
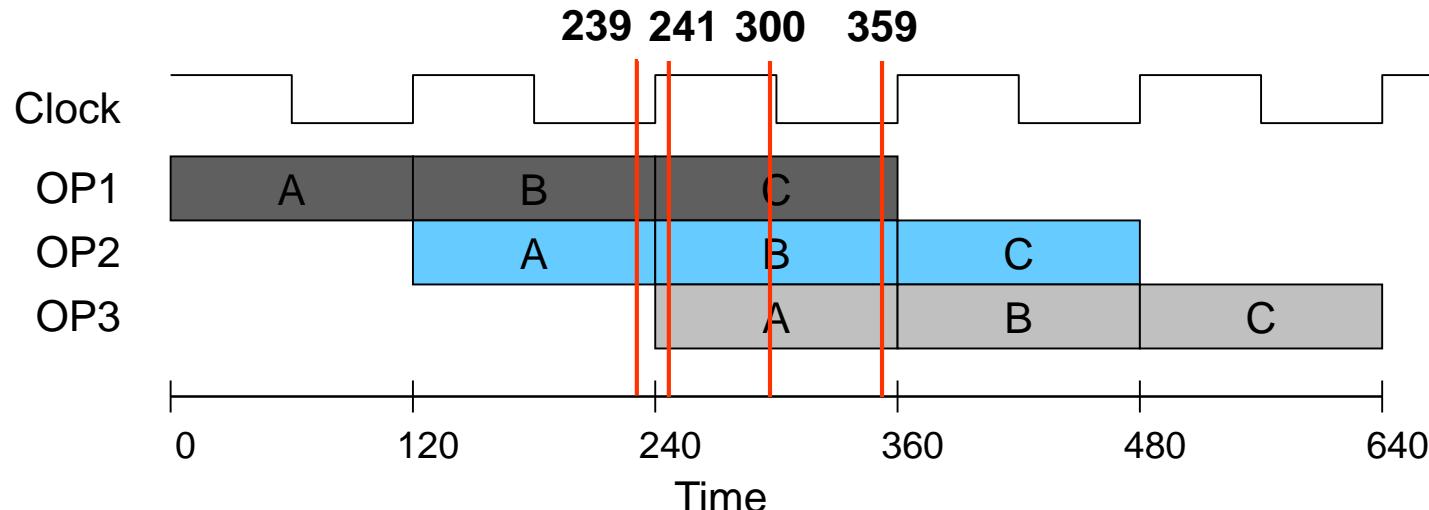
- Cannot start new operation until previous one completes

- Way Pipelined

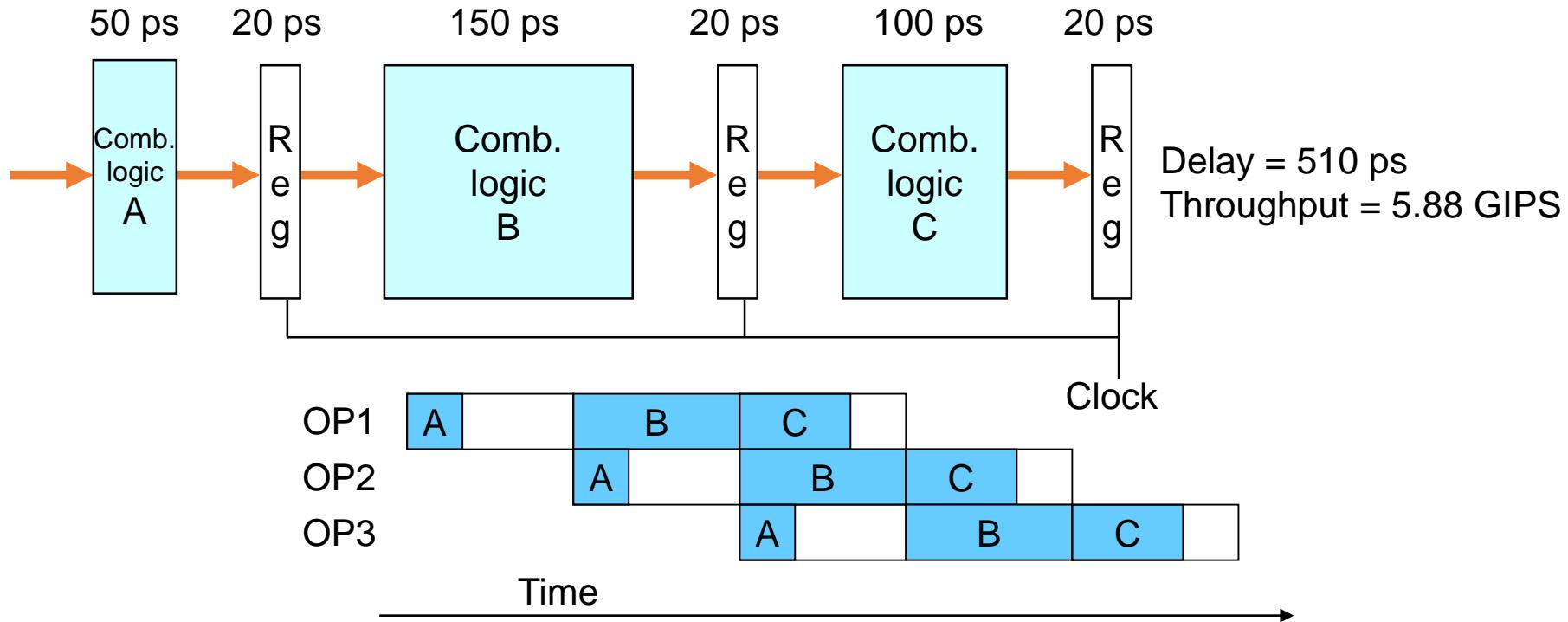


- Up to 3 operations in process simultaneously

# Operating a Pipeline

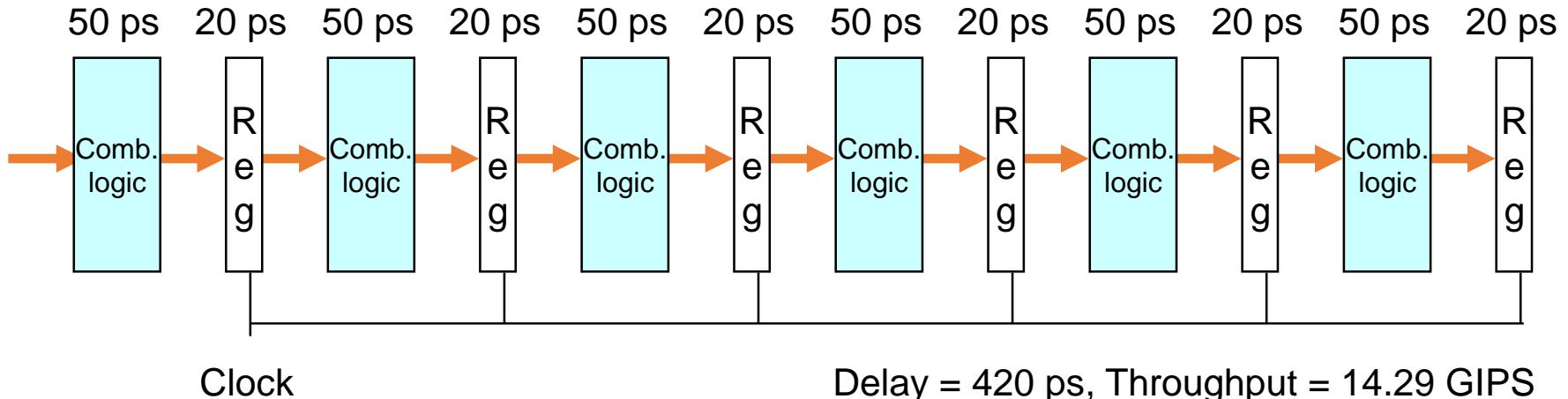


## Limitations: Nonuniform Delays



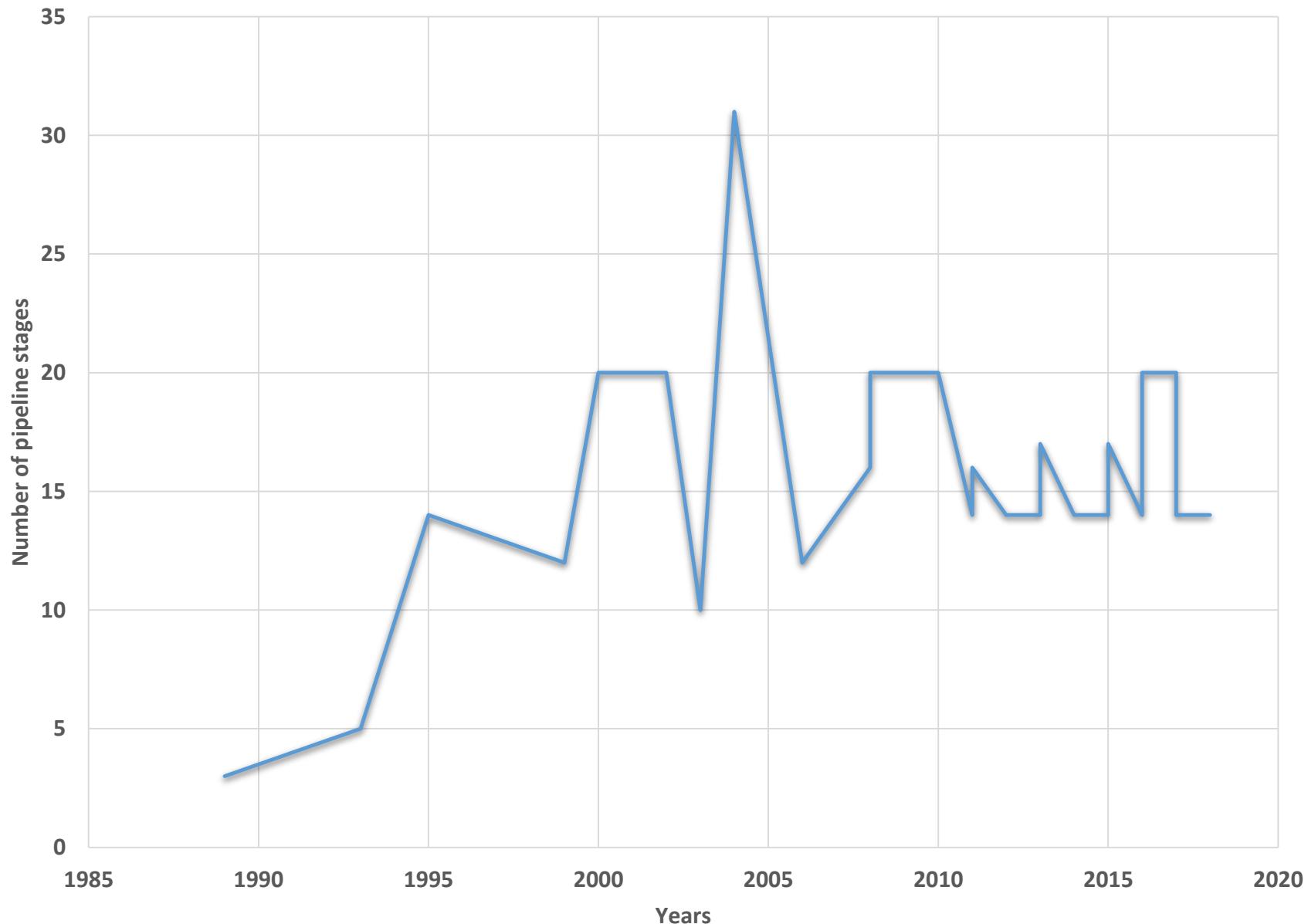
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

# Limitations: Register Overhead

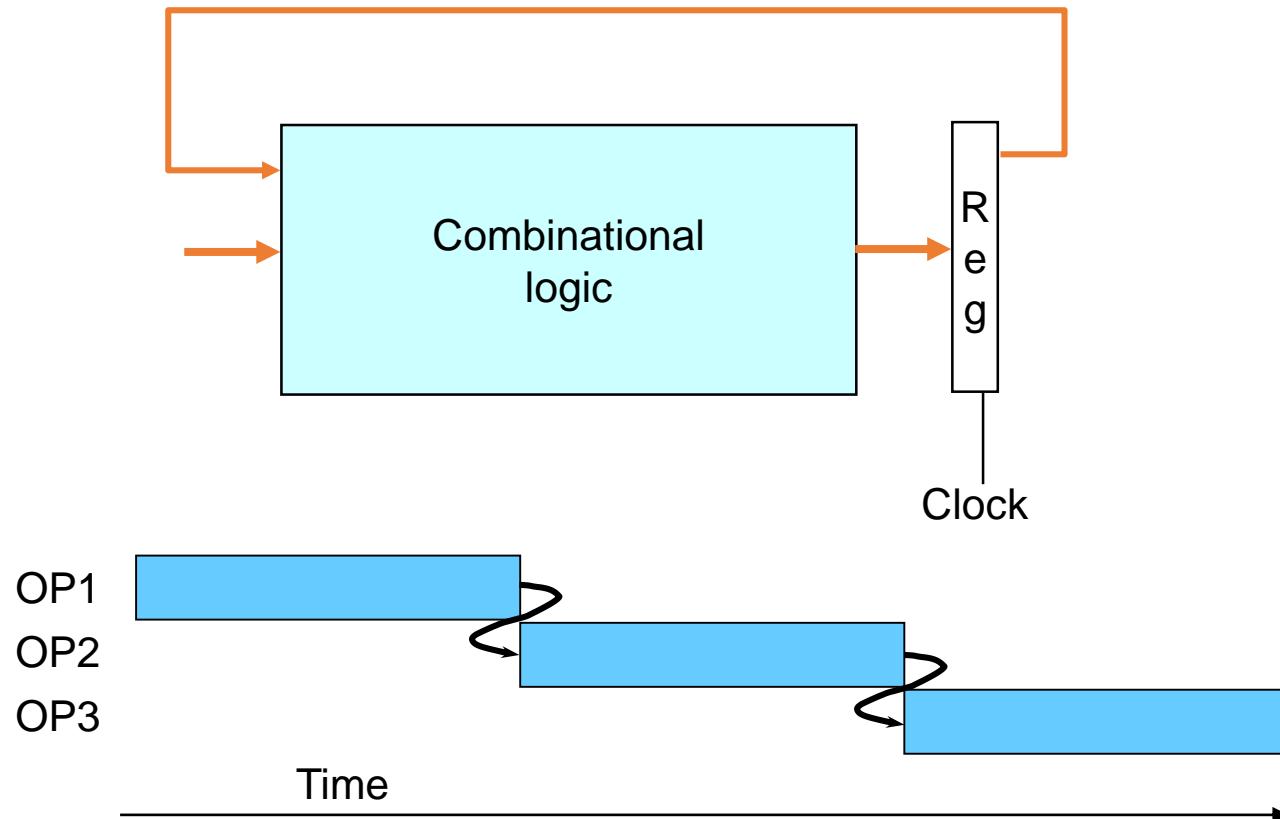


- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through ~~very deep~~ (next slide) pipelining

# Number of pipeline stages (Intel processors)

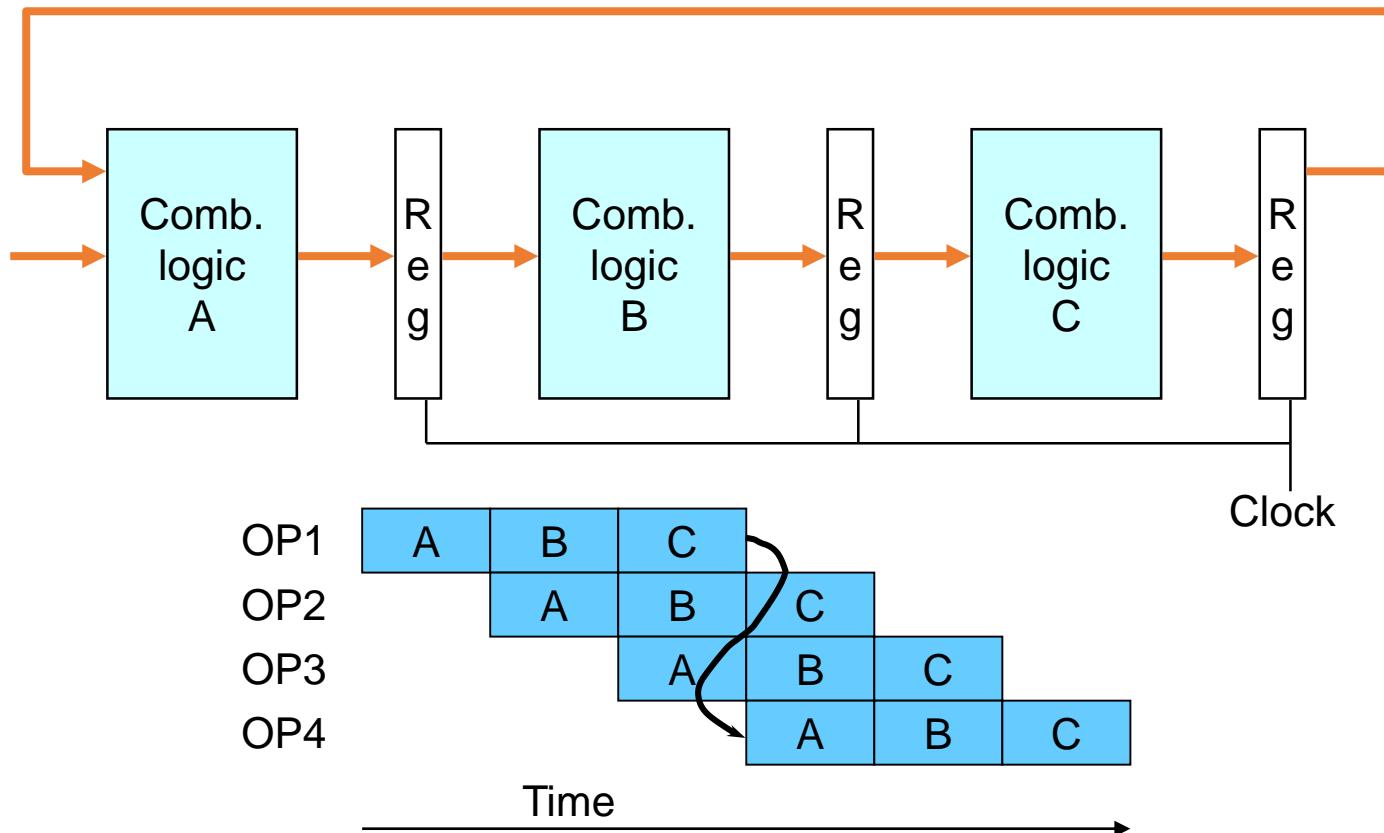


# Data Dependencies



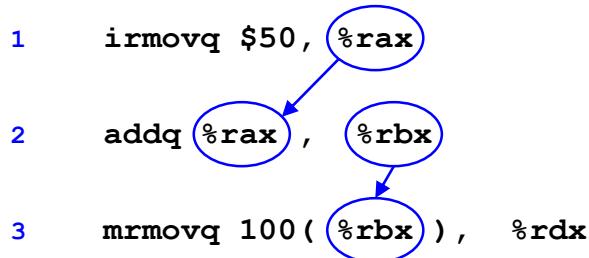
- System
  - Each operation depends on result from preceding one

# Data Hazards



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

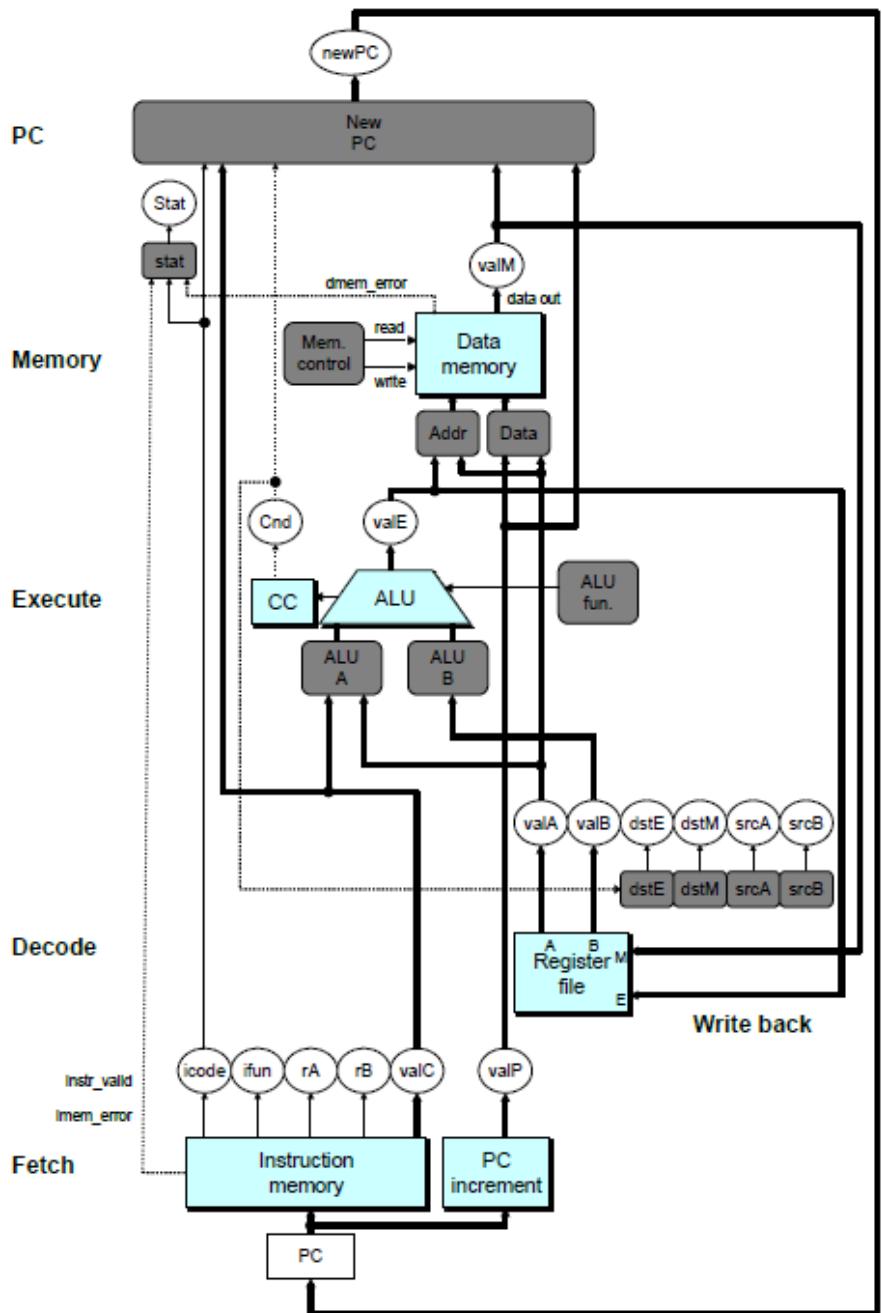
# Data Dependencies in Processors



- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
  - Write-after-read (WAR) dependency
  - Write-after-write (WAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

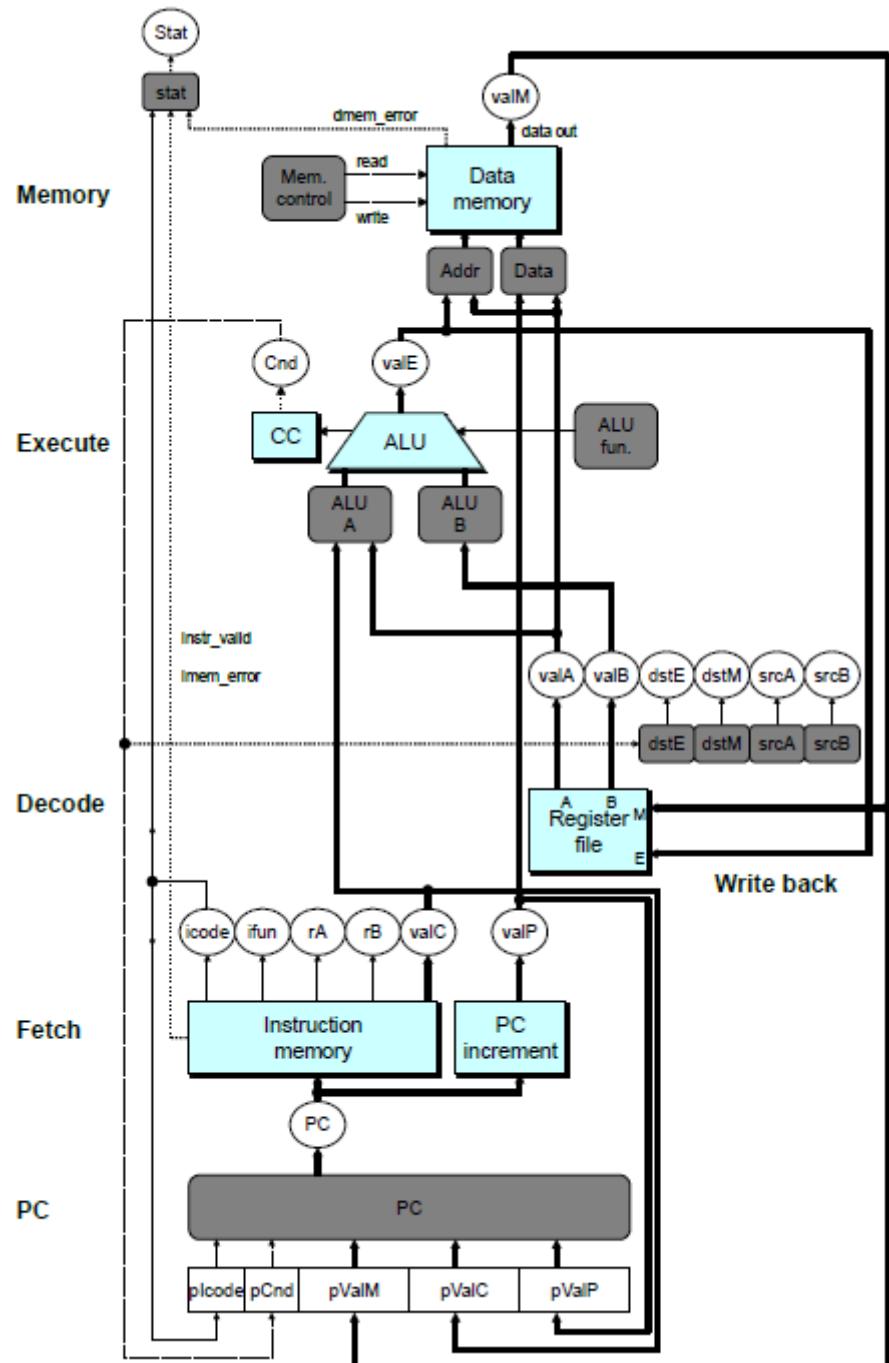
# SEQ Hardware

- Stages occur in sequence
- One operation in process at a time

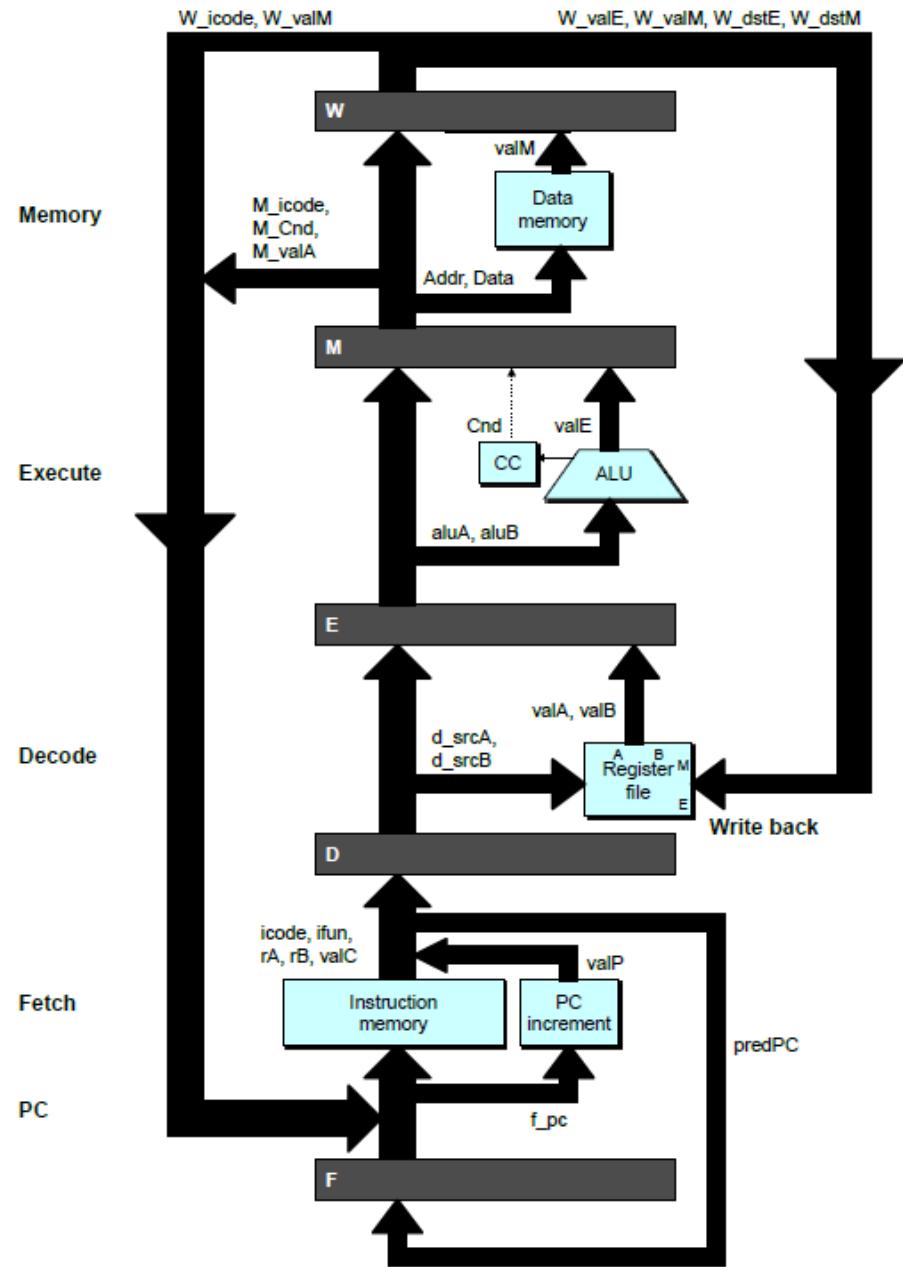
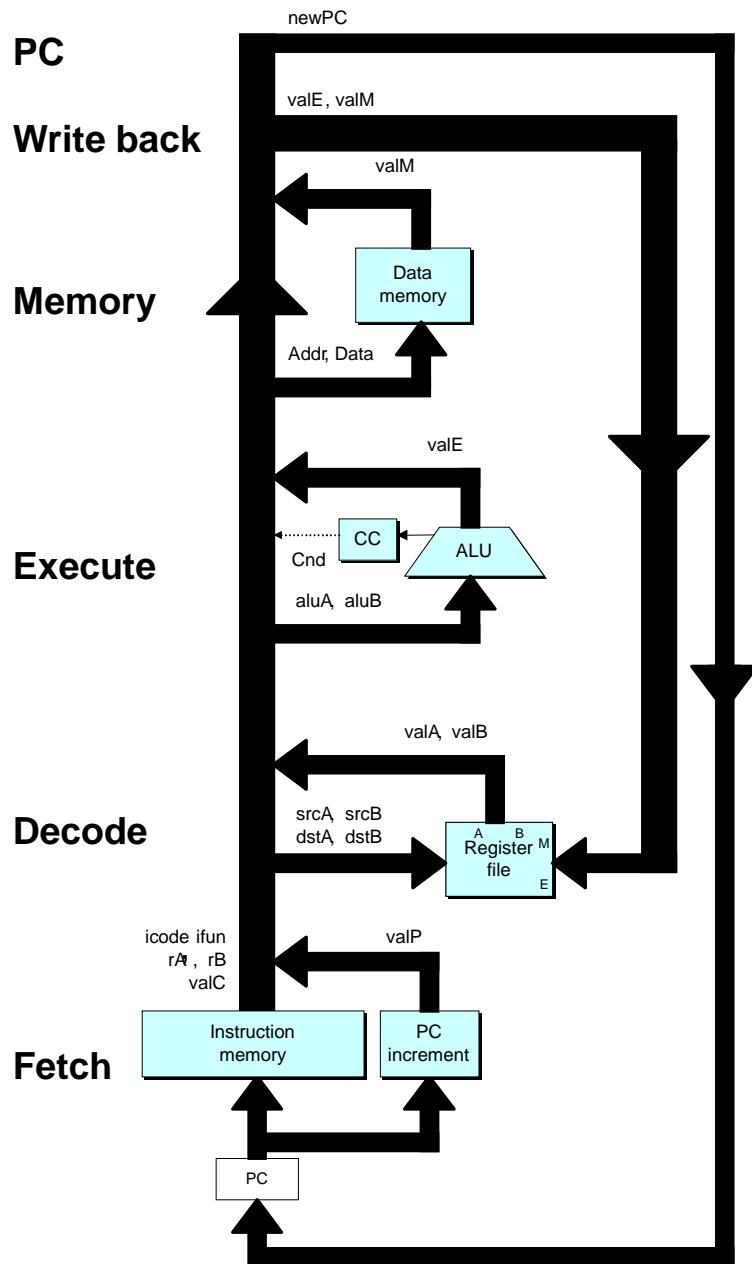


# SEQ+ Hardware

- Still sequential implementation
- Reorder PC stage to put at beginning
- PC Stage
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction
- Processor State
  - PC is no longer stored in register
  - But, can determine PC based on other stored information

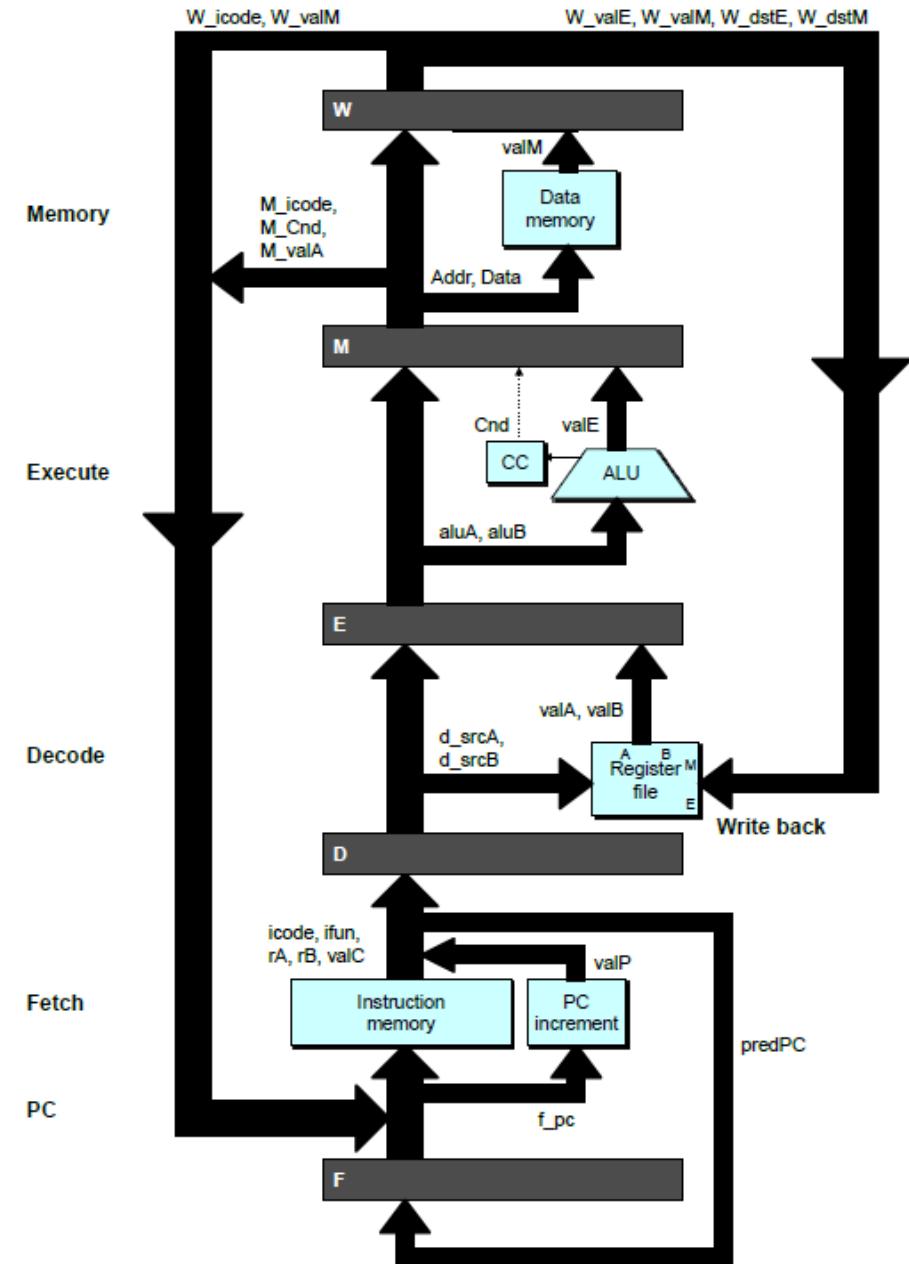


# Adding Pipeline Registers



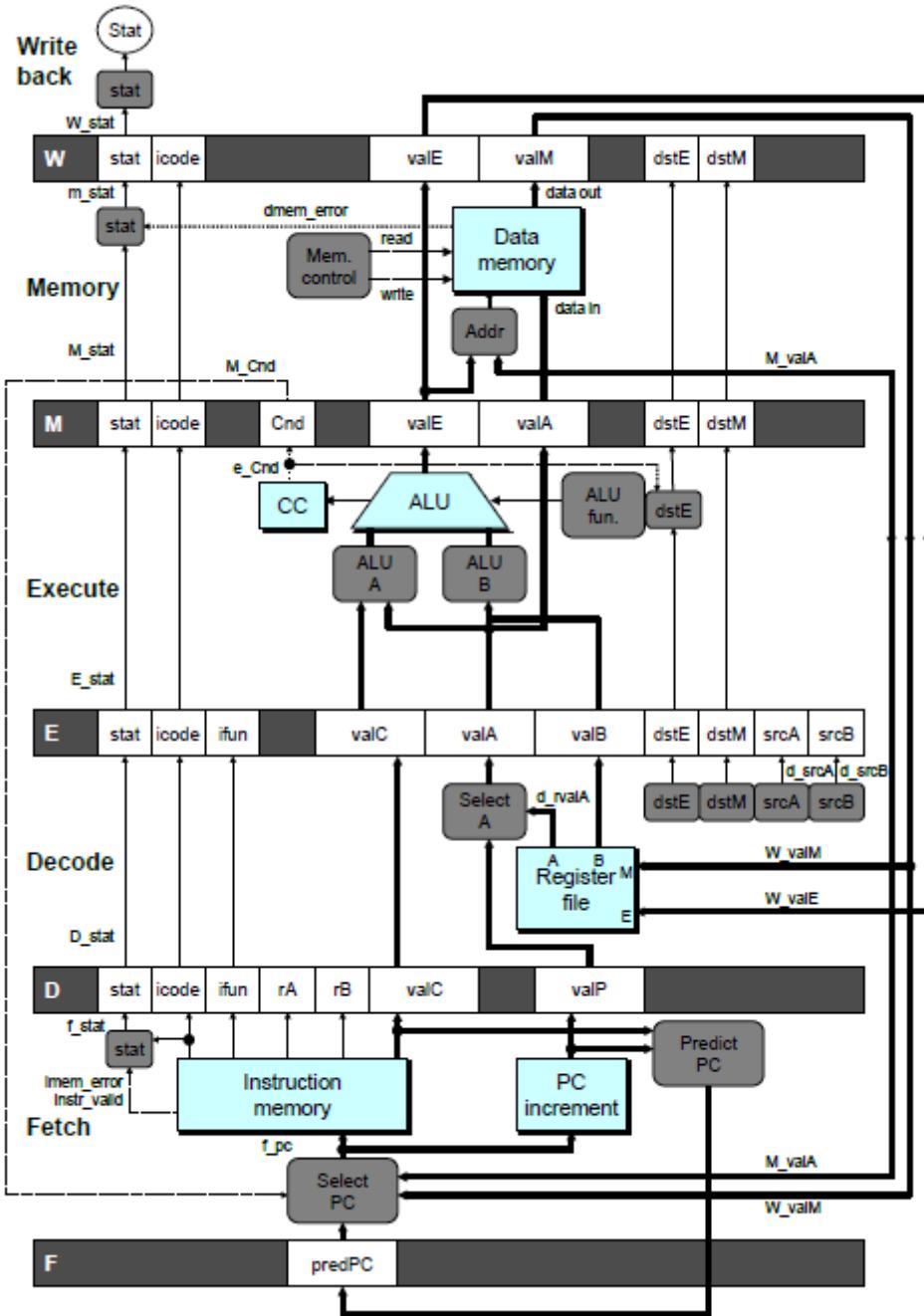
# Pipeline Stages

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file



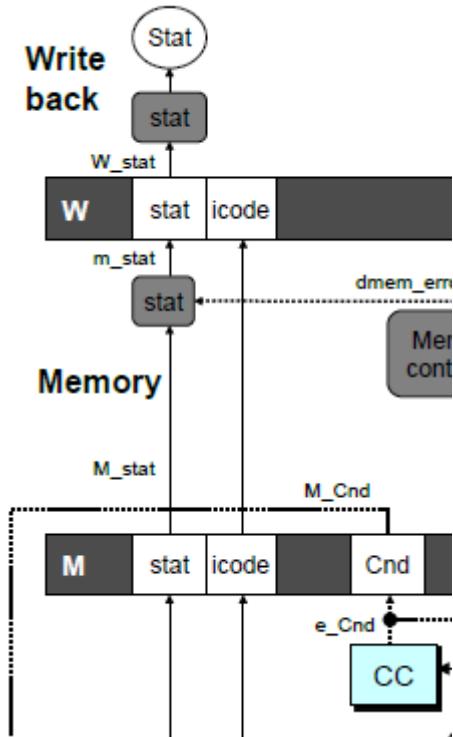
# PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode



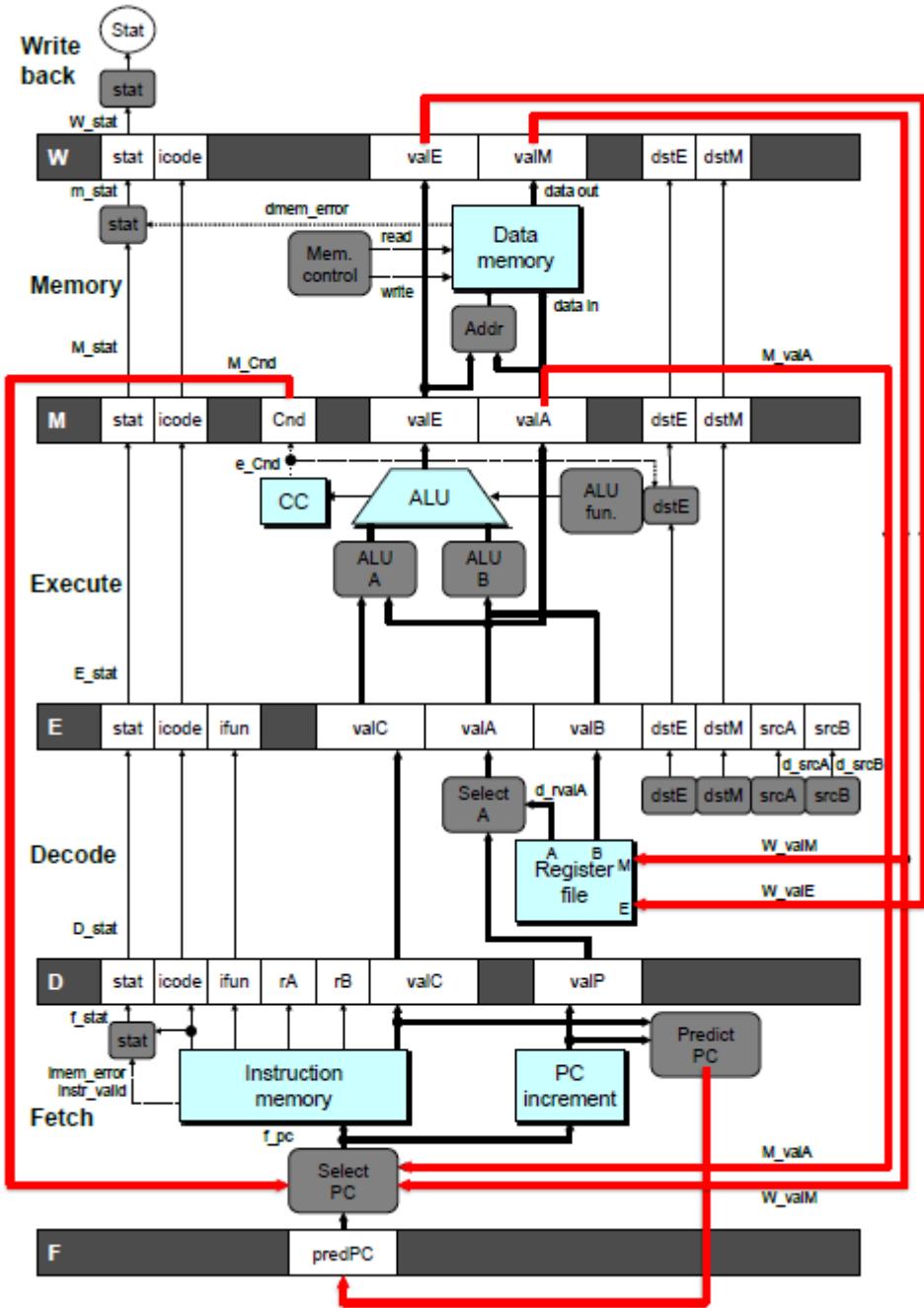
# Signal Naming Conventions

- S\_Field
  - Value of Field held in stage S pipeline register
- s\_Field
  - Value of Field computed in stage S

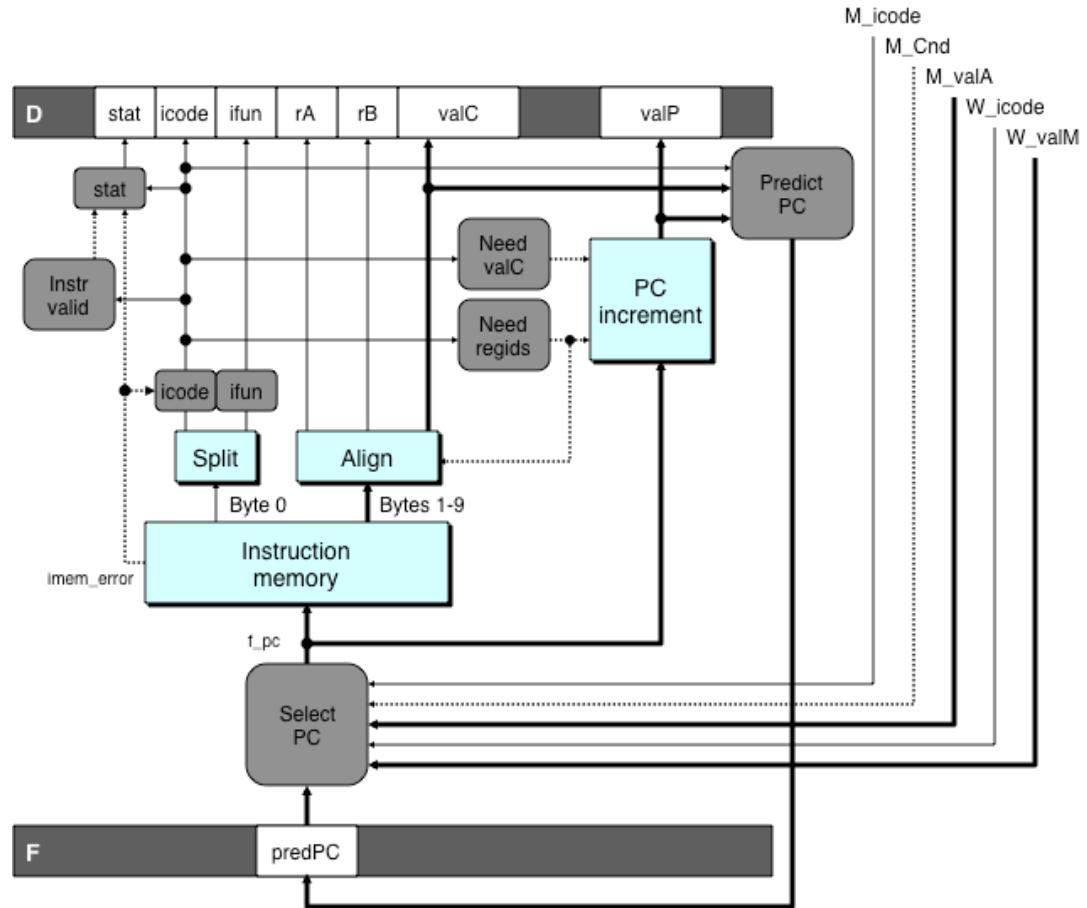


# Feedback Paths

- Predicted PC
  - Guess value of next PC
- Branch information
  - Jump taken/not-taken
  - Fall-through or target address
- Return point
  - Read from memory
- Register updates
  - To register file write ports



# Predicting the PC



- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

# Our Prediction Strategy

- Instructions that Don't Transfer Control
  - Predict next PC to be valP
  - Always reliable
- Call and Unconditional Jumps
  - Predict next PC to be valC (destination)
  - Always reliable
- Conditional Jumps
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken
    - Typically right 60% of time
- Return Instruction
  - Don't try to predict

# Branch prediction (in general)

- Static Branch Prediction

- Based on the typical behavior
- Example: loop

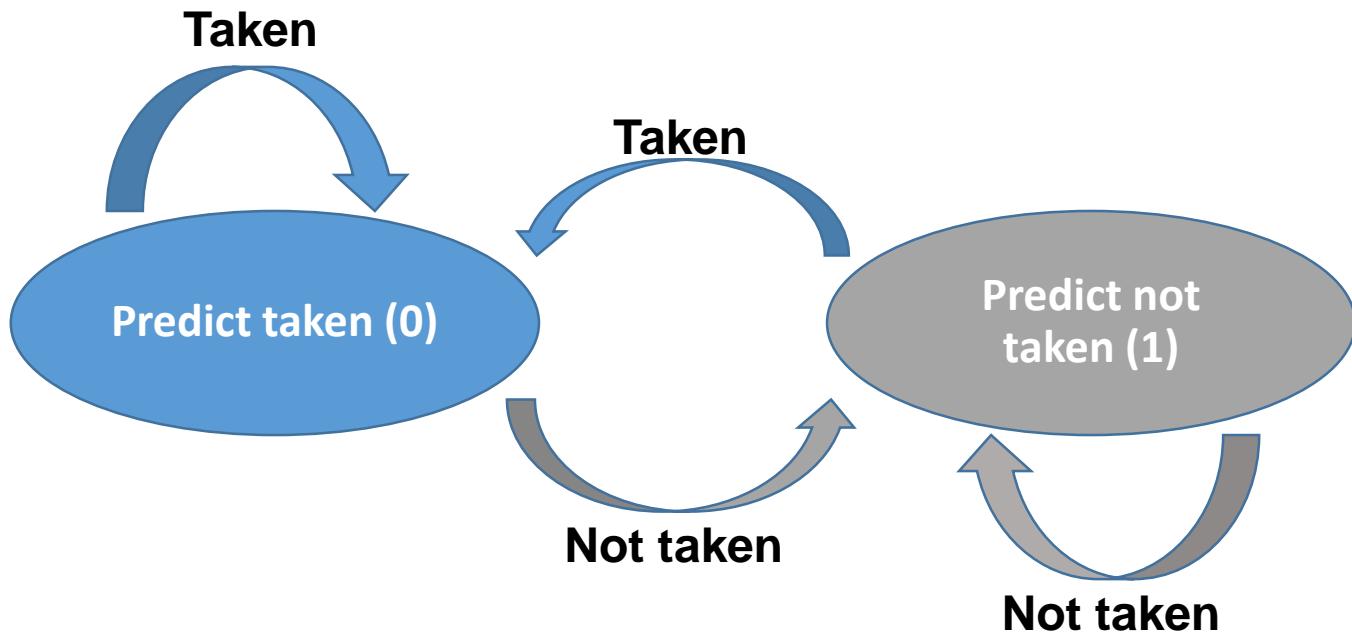
- Dynamic Branch Prediction

- Branch behavior history is measured by the hardware
- Future behavior is predicted based on the past

# Dynamic Branch Prediction

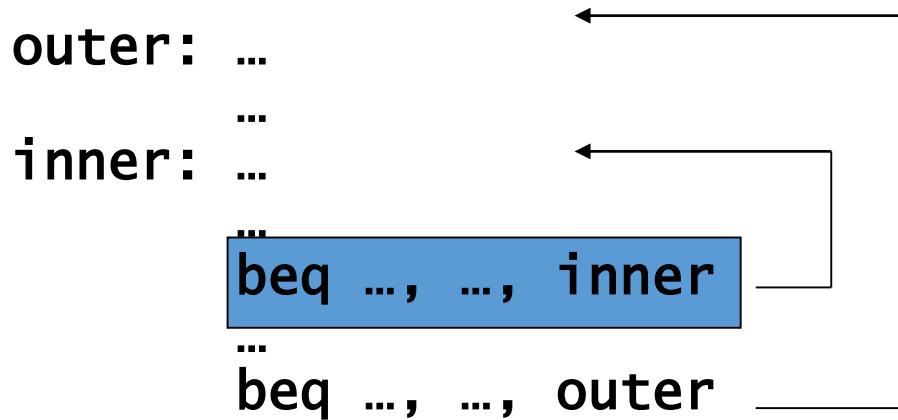
- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

# 1-Bit Predictor



# 1-Bit Predictor: Shortcoming

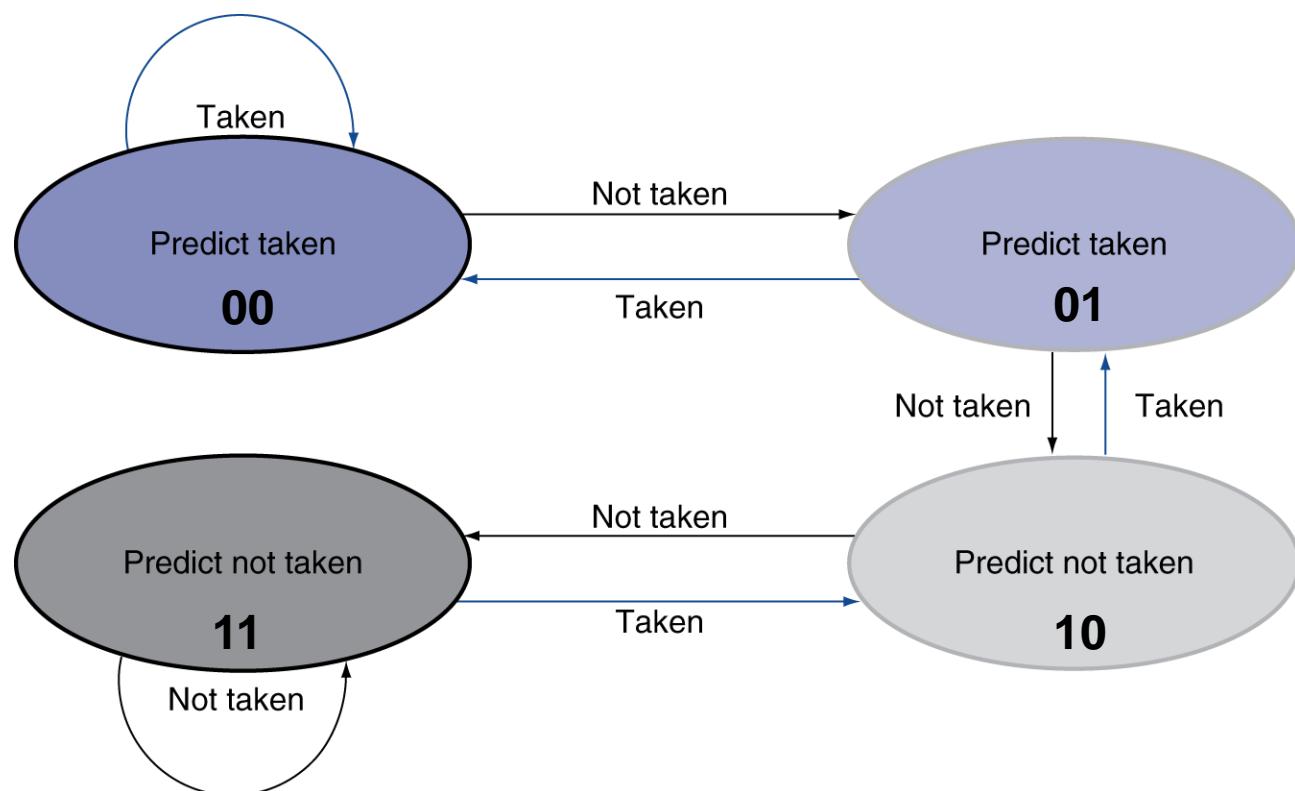
- Inner loop branches mispredicted twice!



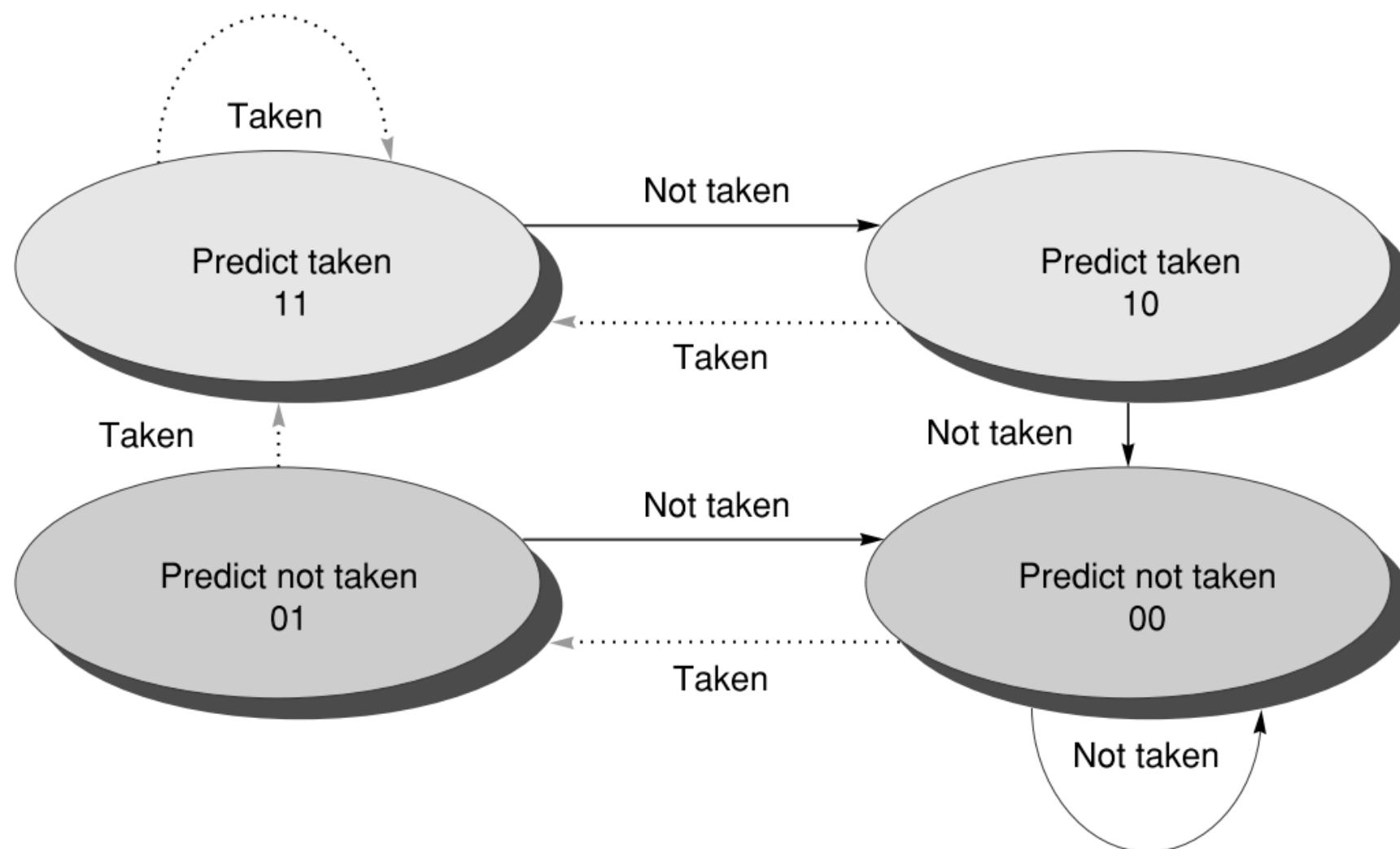
- **Mispredict as taken on last iteration of inner loop**
- **Then mispredict as not taken on first iteration of inner loop next time around**

# 2-Bit Predictor

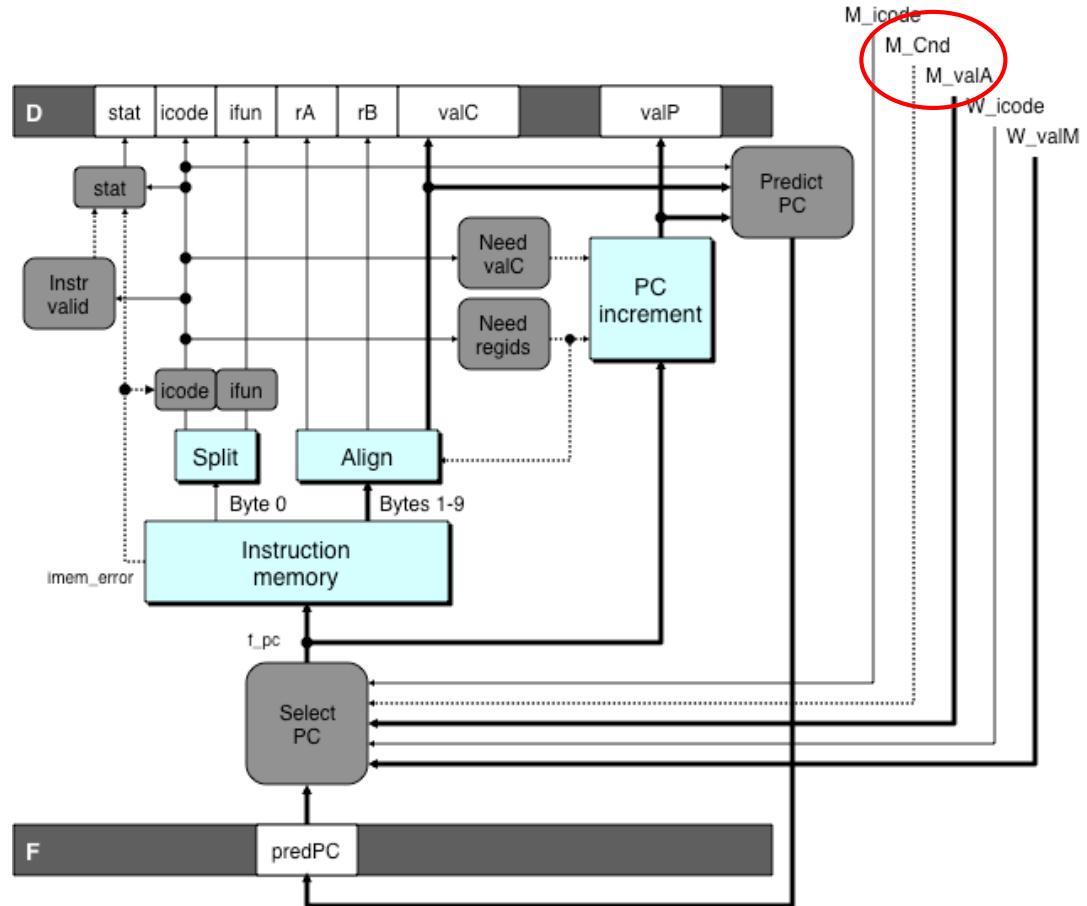
- Only change prediction on two successive mispredictions



# Another 2-Bit Predictor



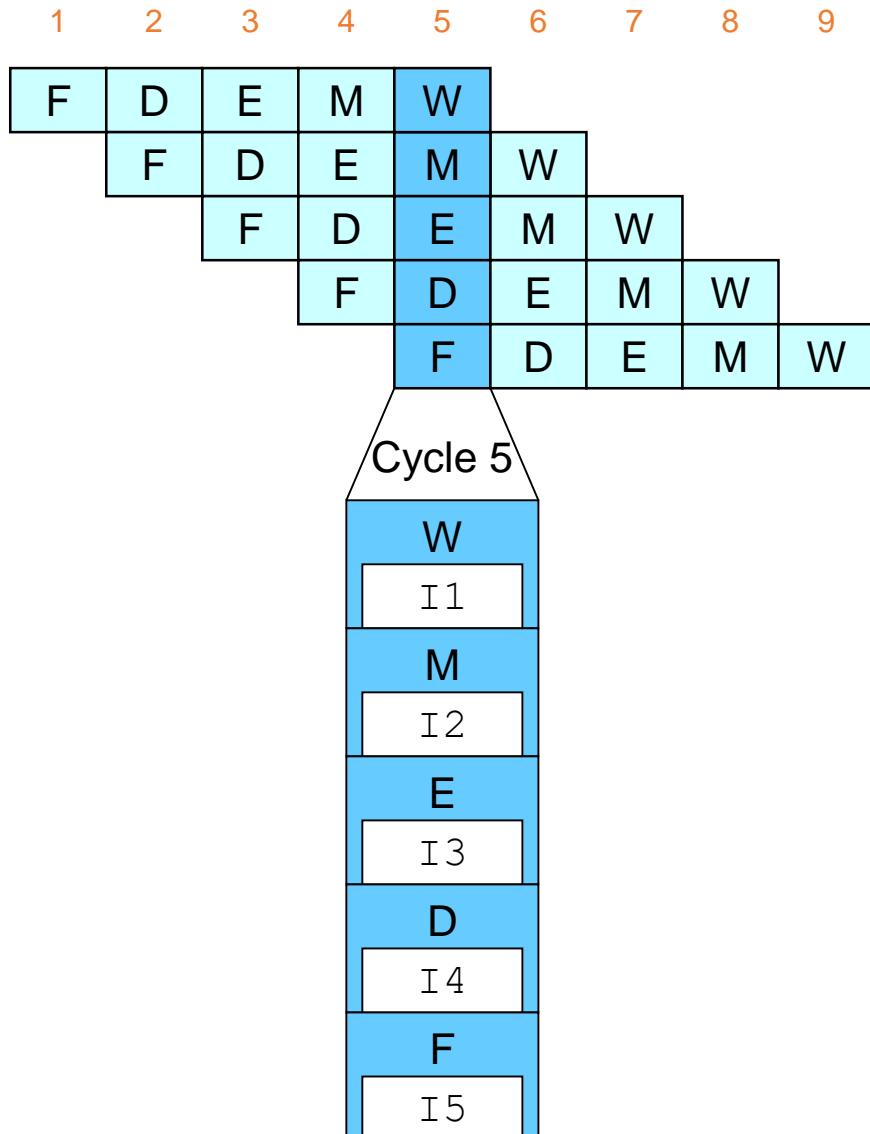
# Recovering from PC Misprediction



- Mispredicted Jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from valA (value  $M\_valA$ )
- Return Instruction
  - Will get return PC when `ret` reaches write-back stage ( $W\_valM$ )

# Pipeline Demonstration

irmovq	\$1,%rax	#I1
irmovq	\$2,%rcx	#I2
irmovq	\$3,%rdx	#I3
irmovq	\$4,%rbx	#I4
halt		#I5

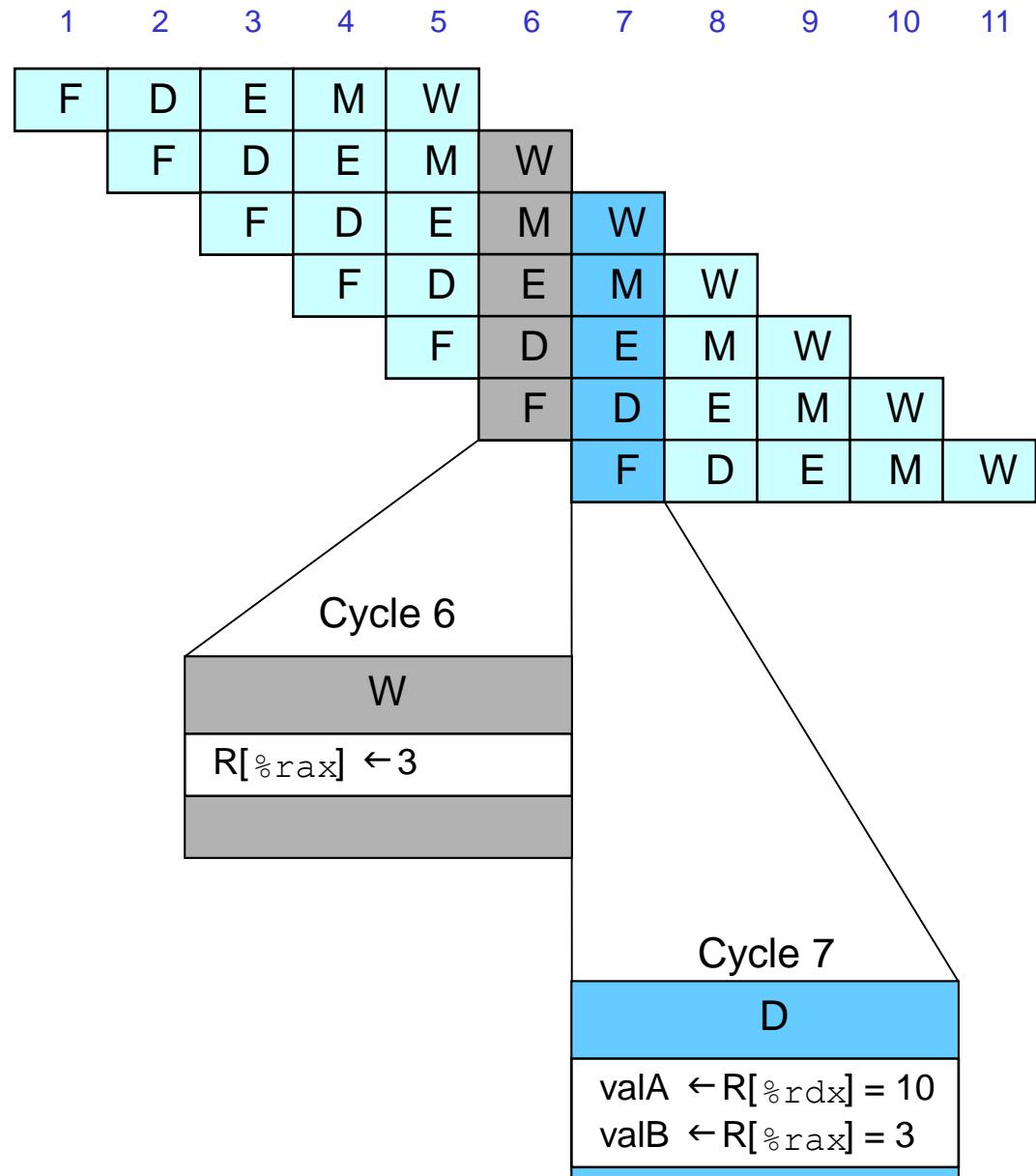


- File: demo-basic.ys

# Data Dependencies: 3 Nop's

```
# demo-h3.ys
```

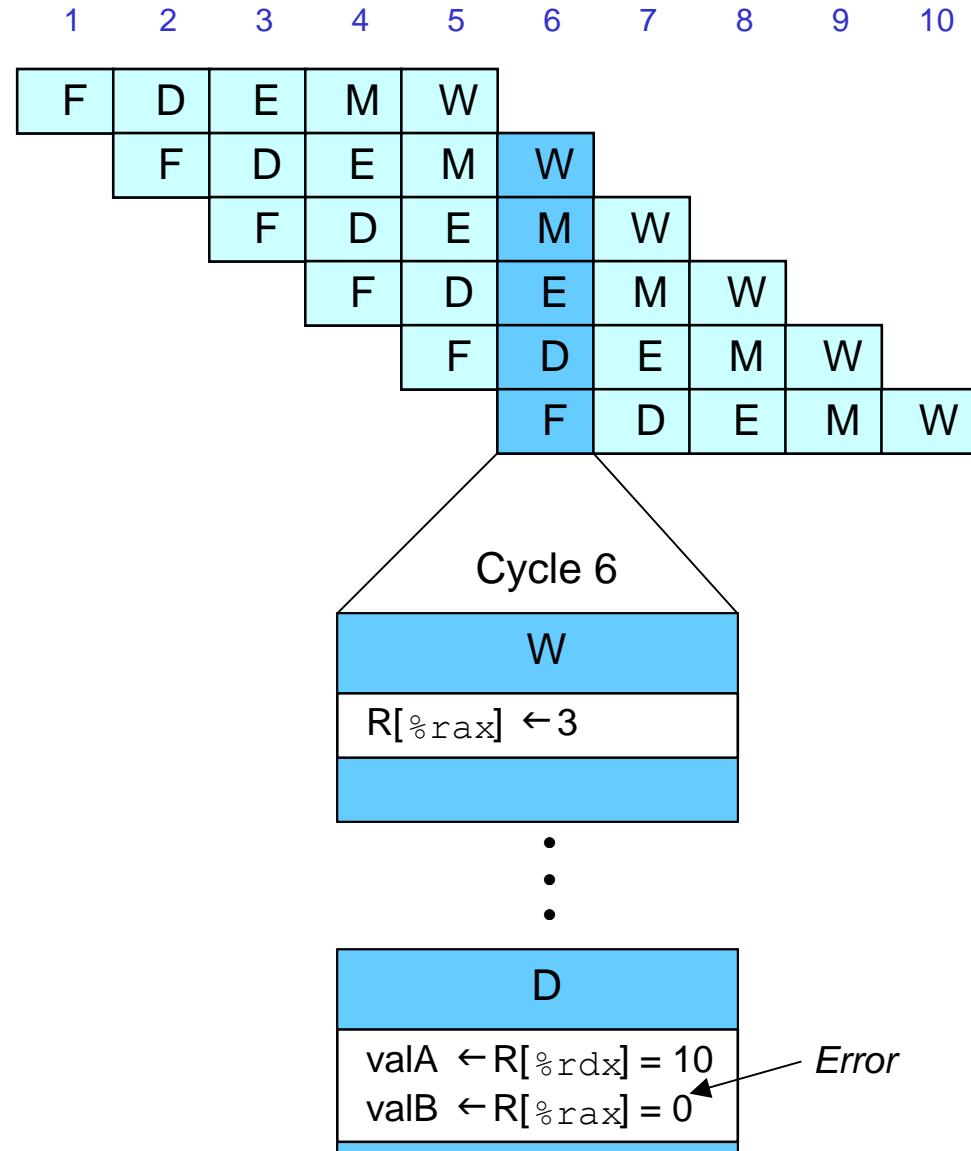
```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: nop  
0x015: nop  
0x016: nop  
0x017: addq %rdx,%rax  
0x019: halt
```



# Data Dependencies: 2 Nop's

```
# demo-h2.ys
```

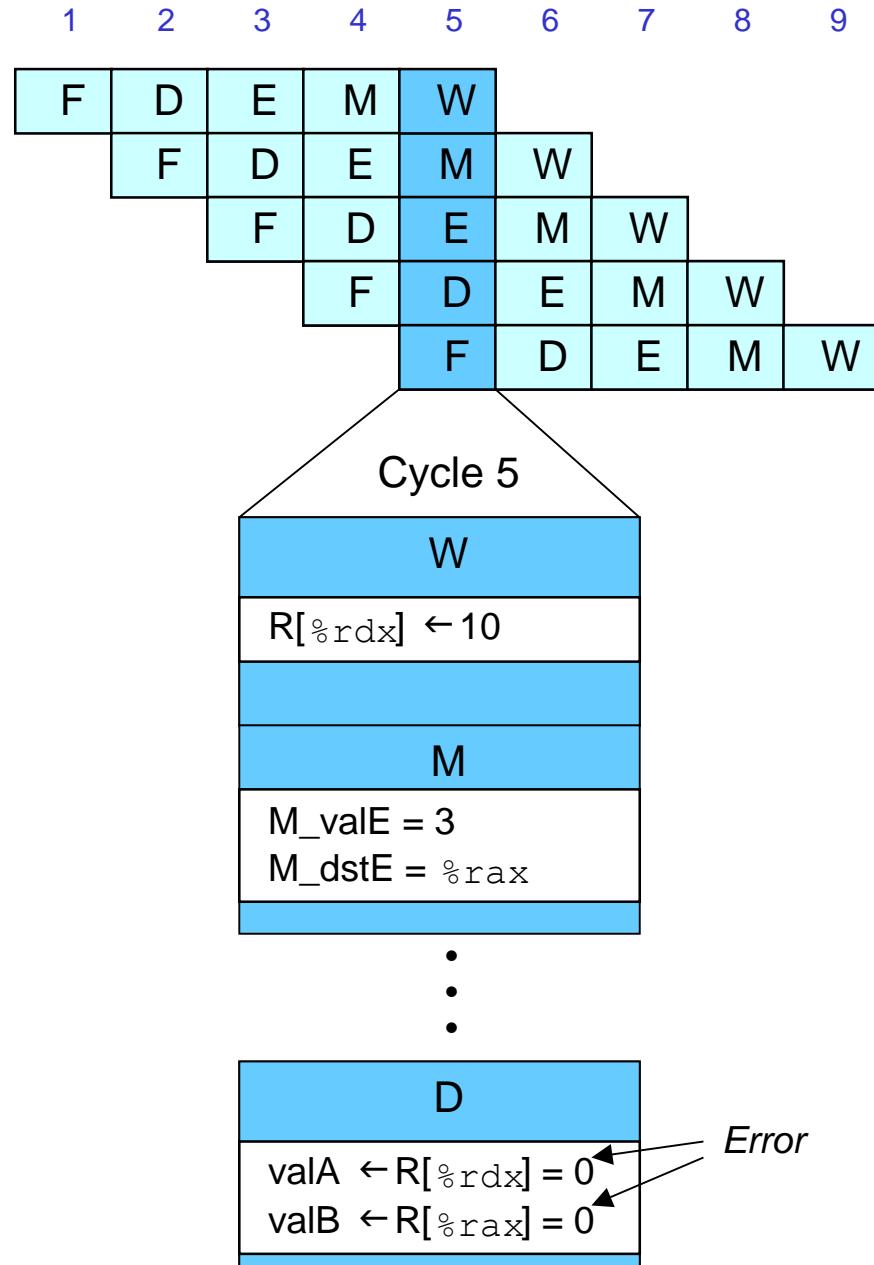
```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



# Data Dependencies: 1 Nop

# demo-h1.ys

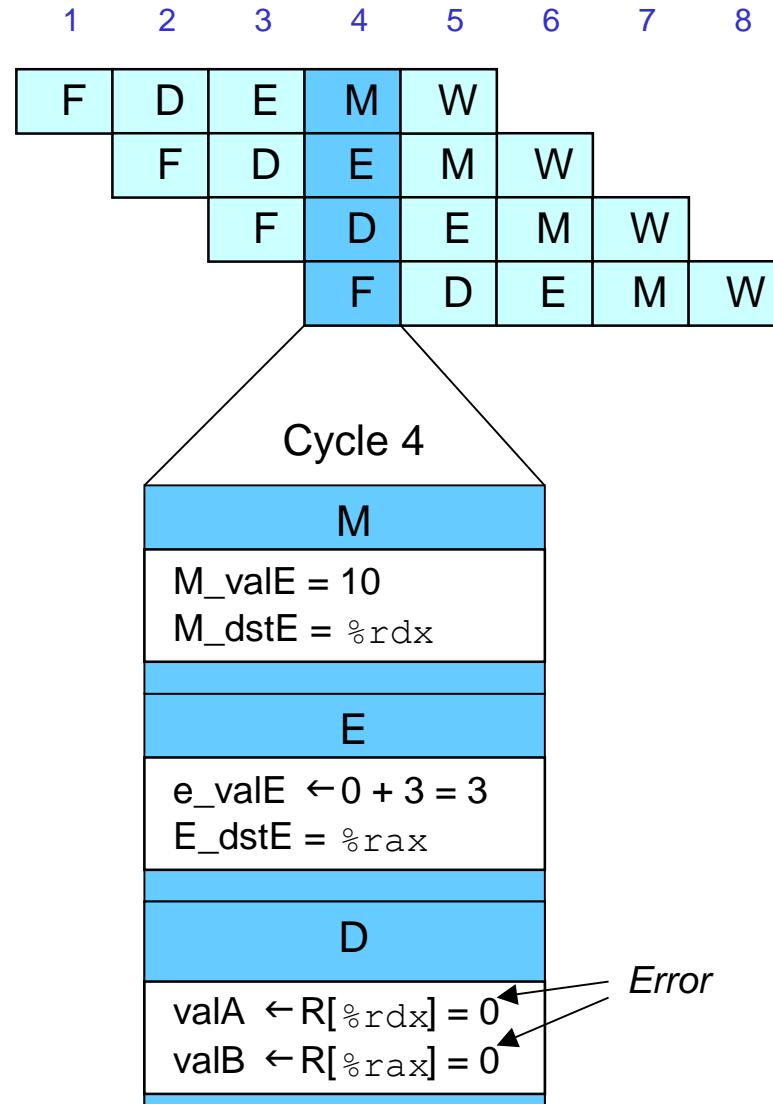
```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: nop  
0x015: addq %rdx,%rax  
0x017: halt
```



# Data Dependencies: No Nop

```
# demo-h0.ys
```

```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: addq %rdx,%rax  
0x016: halt
```



# Branch Misprediction Example

`demo-j.ys`

```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx    # Target (Should not execute)
0x023: irmovq $4, %rcx      # Should not execute
0x02d: irmovq $5, %rdx      # Should not execute
```

- Should only execute first 8 instructions

# Branch Misprediction Trace

# demo-j

0x000: xorq %rax,%rax

	1	2	3	4	5	6	7	8	9
0x000:	F	D	E	M	W				
0x002:	Jne t # Not taken	F	D	E	M	W			
0x019:	t: irmovq \$3, %rdx # Target	F	D	E	M	W			
0x023:	irmovq \$4, %rcx # Target+1	F	D	E	M	W			
0x00b:	irmovq \$1, %rax # Fall Through	F	D	E	M	W			

0x002: jne t # Not taken

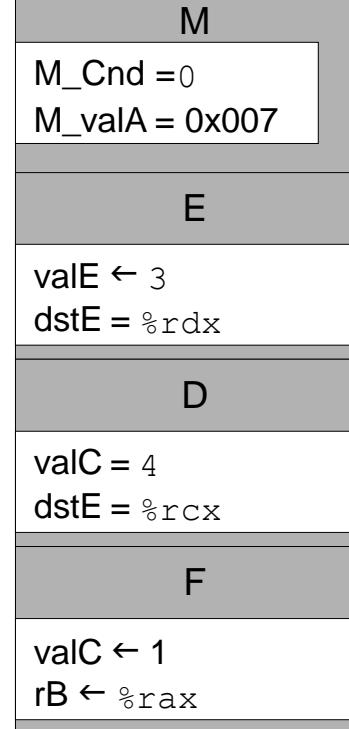
0x019: t: irmovq \$3, %rdx # Target

0x023: irmovq \$4, %rcx # Target+1

0x00b: irmovq \$1, %rax # Fall Through

- Incorrectly execute two instructions at branch target

Cycle 5



# Return Example

demo-ret.ys

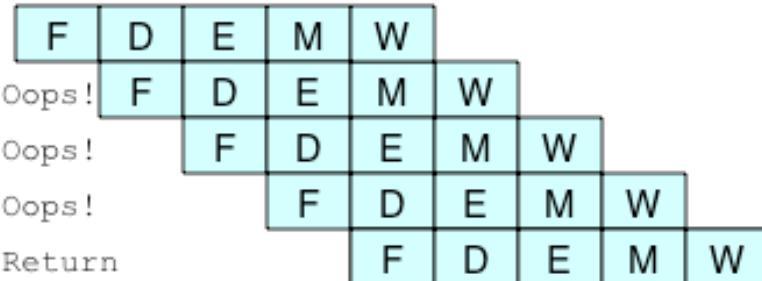
```
0x000:    irmovq Stack,%rsp  # Initialize stack pointer
0x00a:    nop                 # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p              # Procedure call
0x016:    irmovq $5,%rsi    # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p:    nop             # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax    # Should not be executed
0x02e:    irmovq $2,%rcx    # Should not be executed
0x038:    irmovq $3,%rdx    # Should not be executed
0x042:    irmovq $4,%rbx    # Should not be executed
0x100: .pos 0x100
0x100: Stack:               # Initial stack pointer
```

- Require lots of nops to avoid data hazards

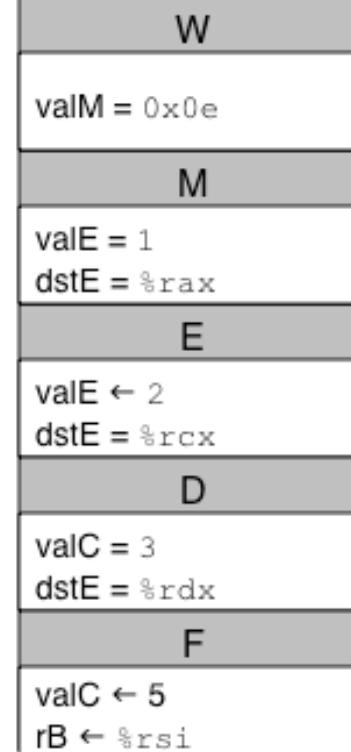
# Incorrect Return Example

```
# demo-ret
```

```
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



- Incorrectly execute 3 instructions following `ret`



# Pipeline Summary

- Concept
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode
- Limitations
  - Can't handle dependencies between instructions when instructions follow too closely
  - Data dependencies
    - One instruction writes register, later one reads it
  - Control dependency
    - Instruction sets PC in way that pipeline did not predict correctly
    - Mispredicted branch and return
- Next fixing the Pipeline

# Computer Architecture: Pipelined Implementation - II

CENG331 - Computer Organization

Instructor:

Murat Manguoglu (Sections 1-2)

Adapted from slides of the textbook: <http://csapp.cs.cmu.edu/>

Slides 24-28 adapted from the slides of the textbook: D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3<sup>rd</sup> Edition

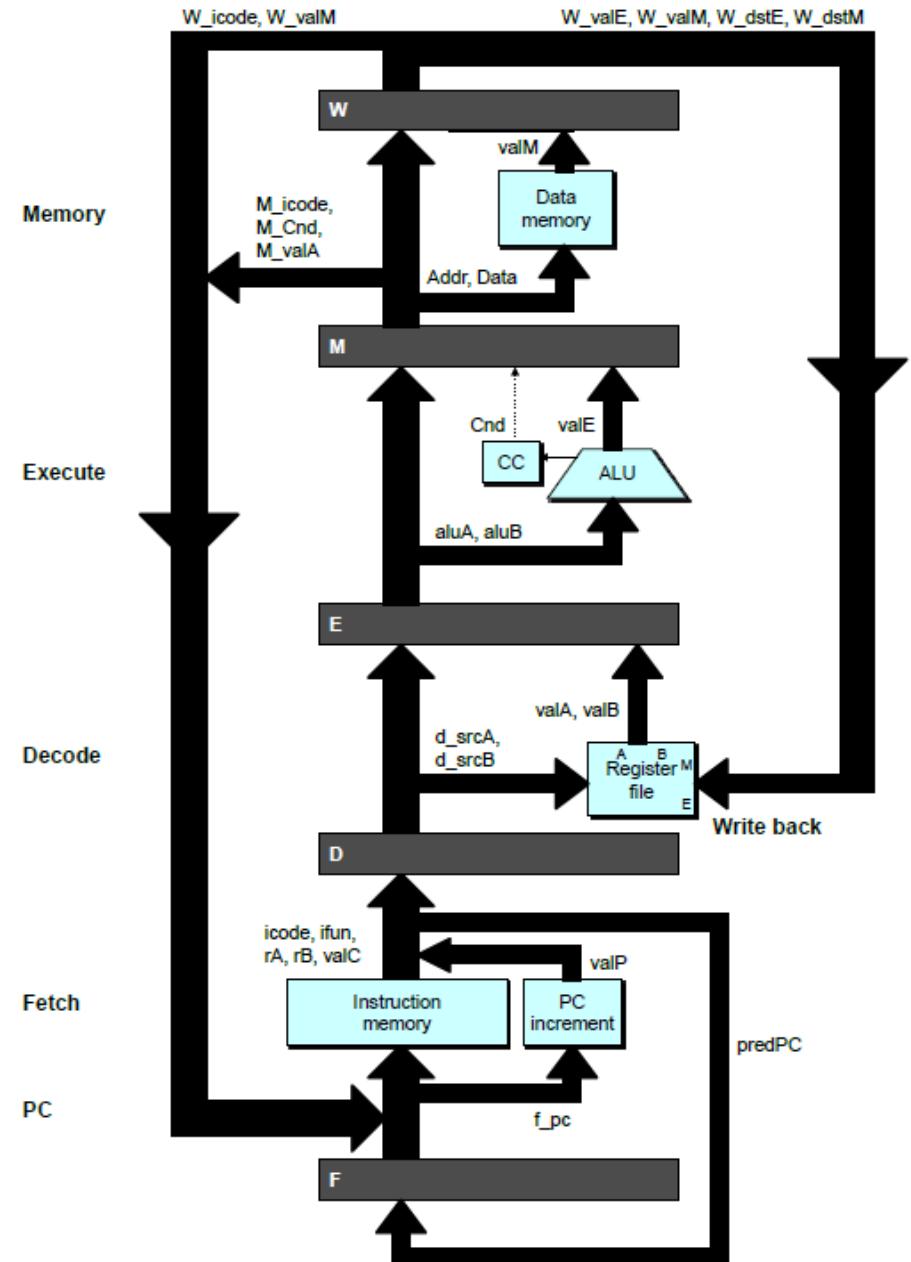
# Overview

*Make the pipelined processor work!*

- Data Hazards
  - Instruction having register R as source follows shortly after instruction having register R as destination
  - Common condition, don't want to slow down pipeline
- Control Hazards
  - Mispredict conditional branch
    - Our design predicts all branches as being taken
    - Naïve pipeline executes two extra instructions
  - Getting return address for `ret` instruction
    - Naïve pipeline executes three extra instructions
- Making Sure It Really Works
  - What if multiple special cases happen simultaneously?

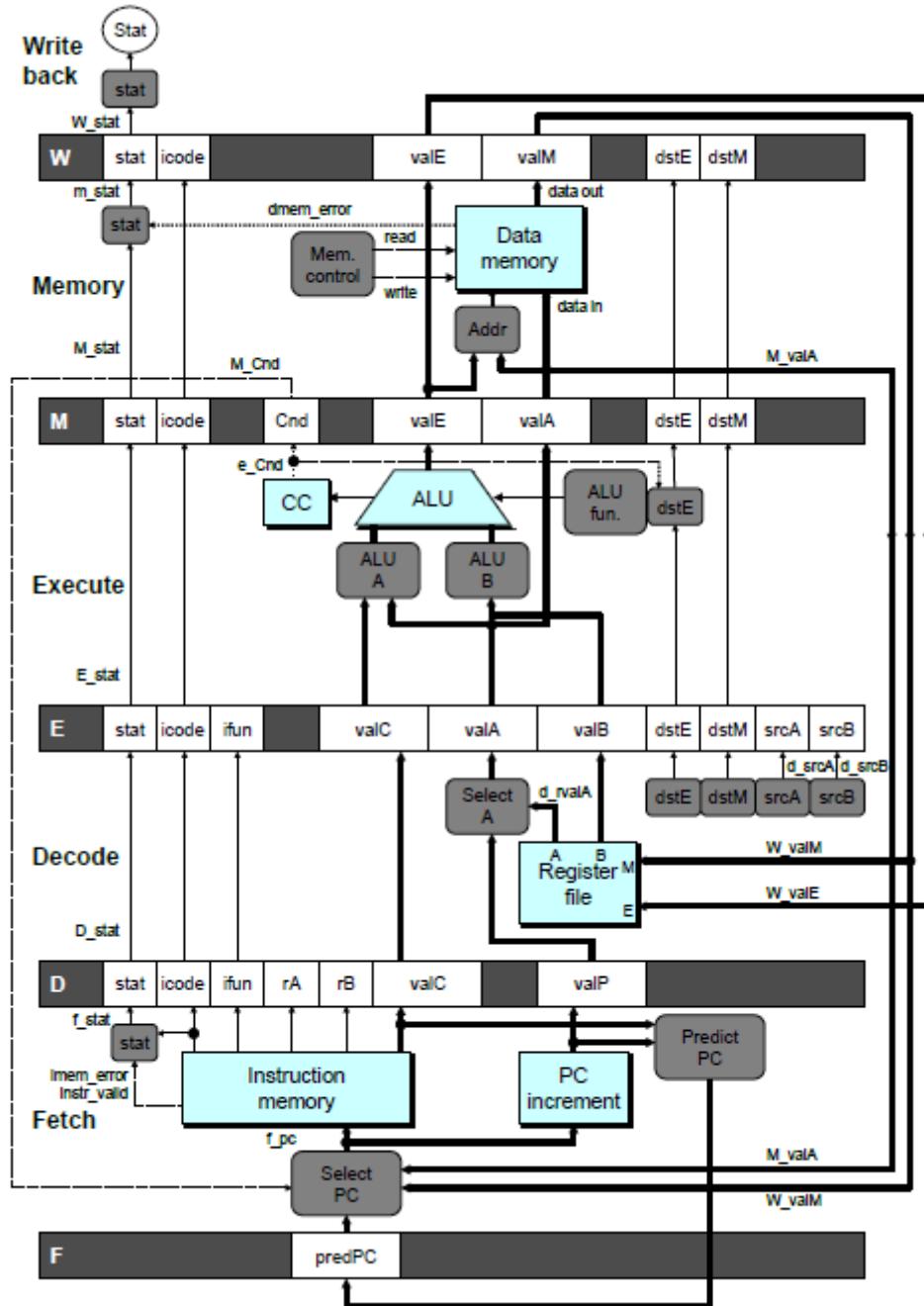
# Pipeline Stages

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file



# PIPE- Hardware

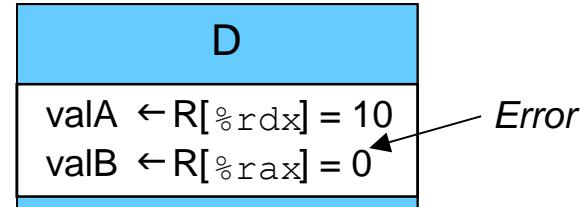
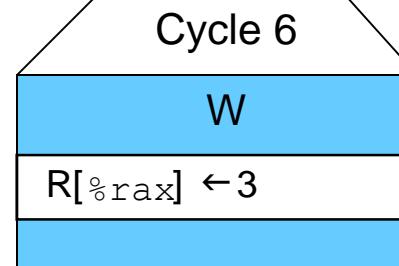
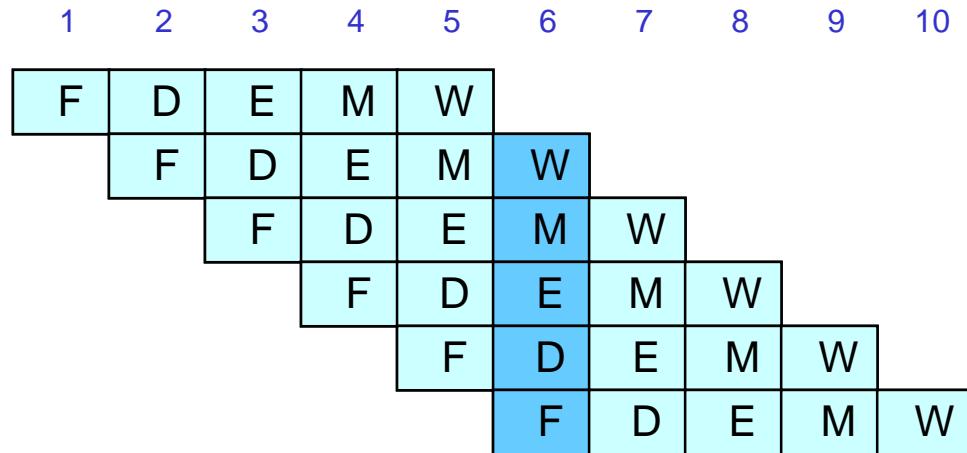
- Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode



# Data Dependencies: 2 Nop's

# demo-h2.ys

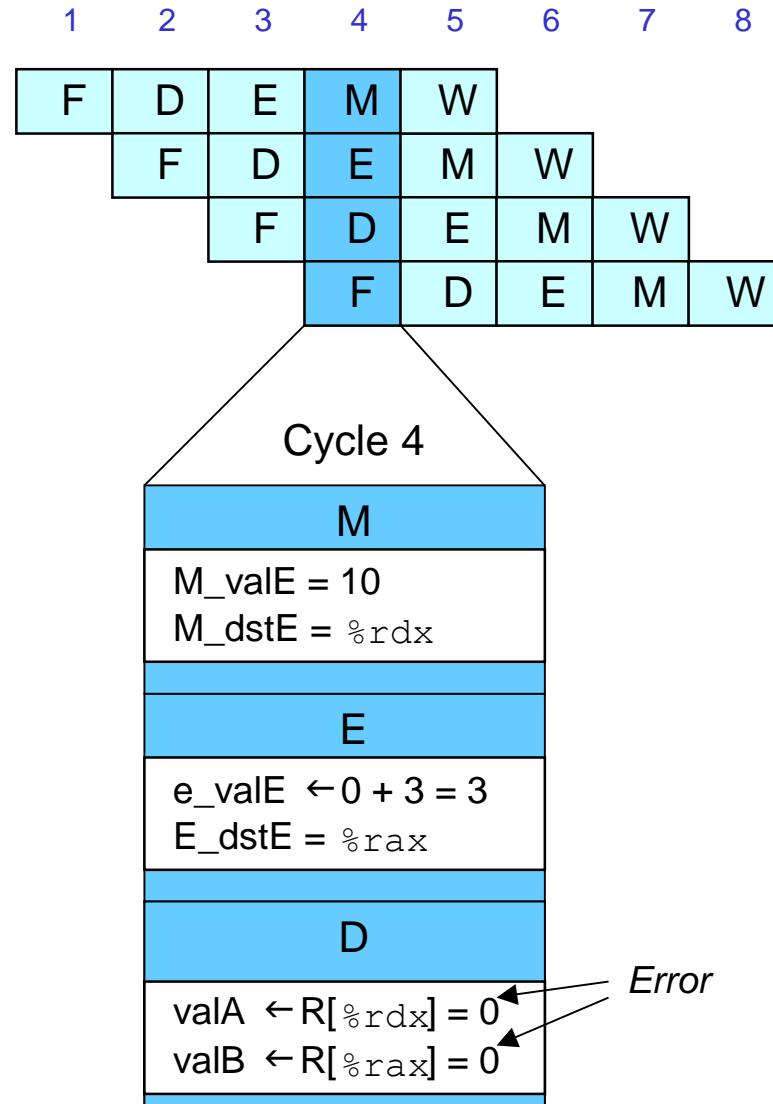
```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: nop  
0x015: nop  
0x016: addq %rdx,%rax  
0x018: halt
```



# Data Dependencies: No Nop

```
# demo-h0.ys
```

```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: addq %rdx,%rax  
0x016: halt
```



# Stalling for Data Dependencies

```
# demo-h2.ys
```

```
0x000: irmovq $10,%rdx
```

```
0x00a: irmovq $3,%rax
```

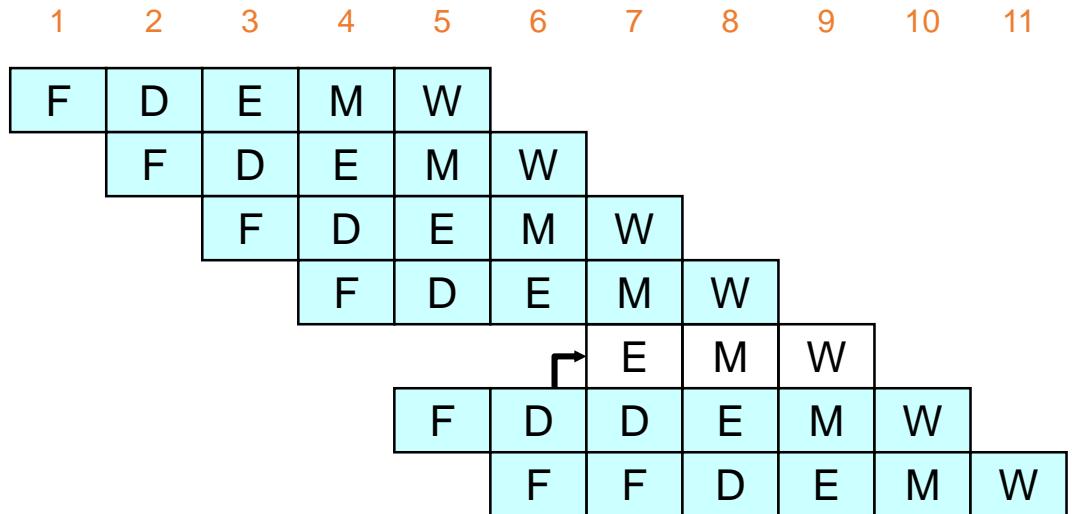
```
0x014: nop
```

```
0x015: nop
```

**bubble**

```
0x016: addq %rdx,%rax
```

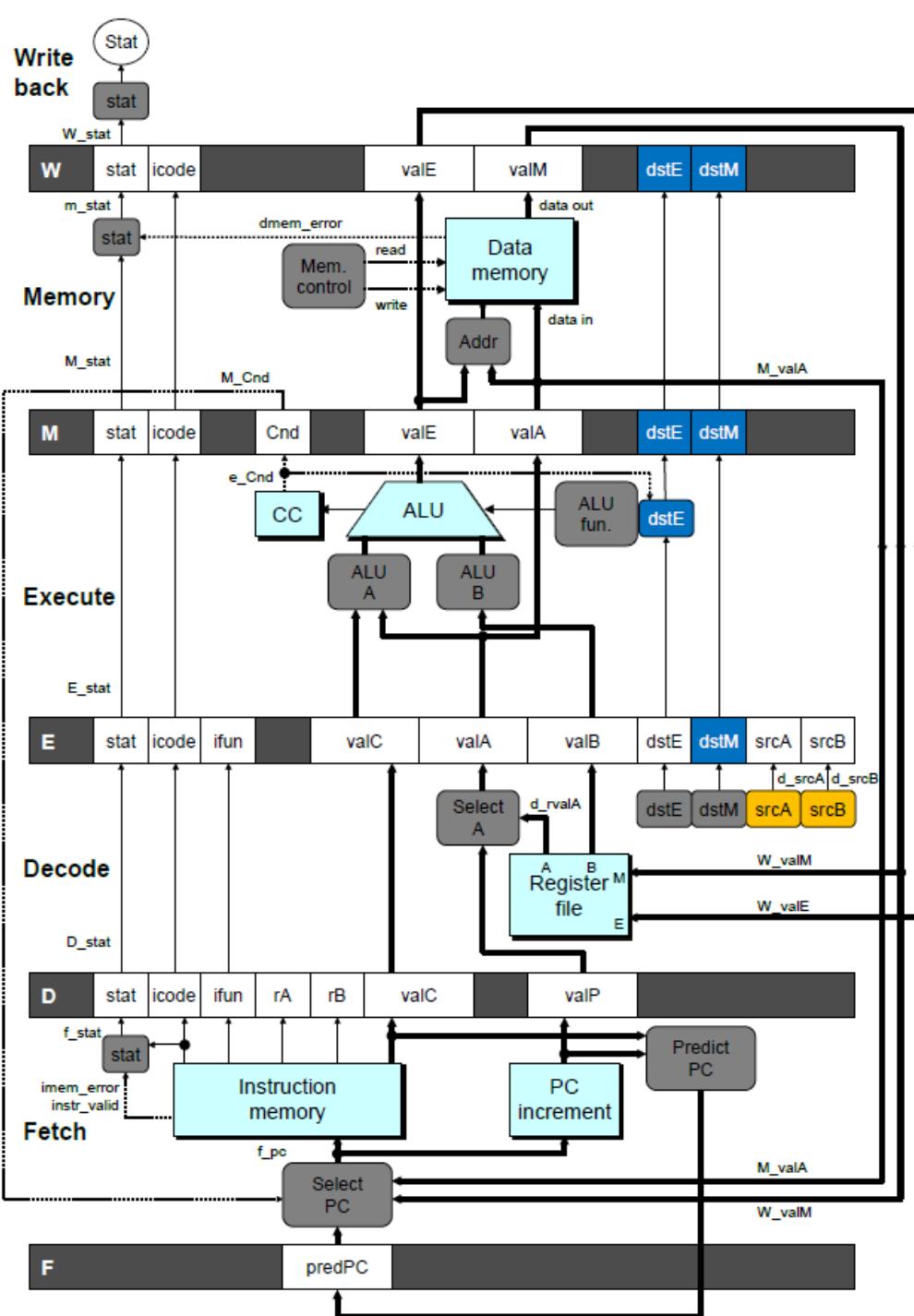
```
0x018: halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

# Stall Condition

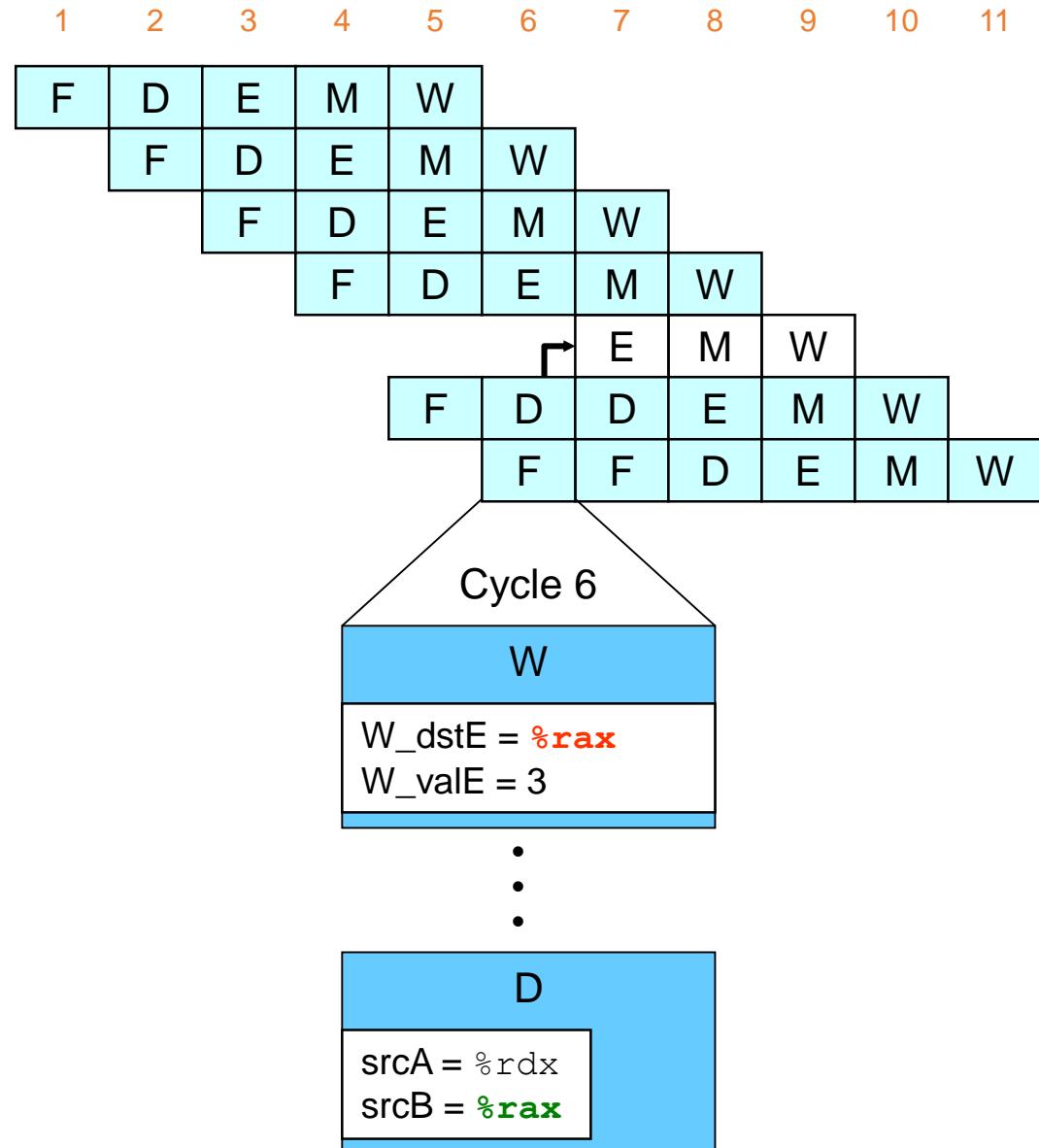
- Source Registers
  - srcA and srcB of current instruction in decode stage
- Destination Registers
  - dstE and dstM fields
  - Instructions in execute, memory, and write-back stages
- Special Case
  - Don't stall for register ID 15 (0xF)
    - Indicates absence of register operand
    - Or failed cond. move



# Detecting Stall Condition

# demo-h2.ys

```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: nop  
0x015: nop  
bubble  
0x016: addq %rdx,%rax  
0x018: halt
```



# Stalling X3

# demo-h0.ys

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

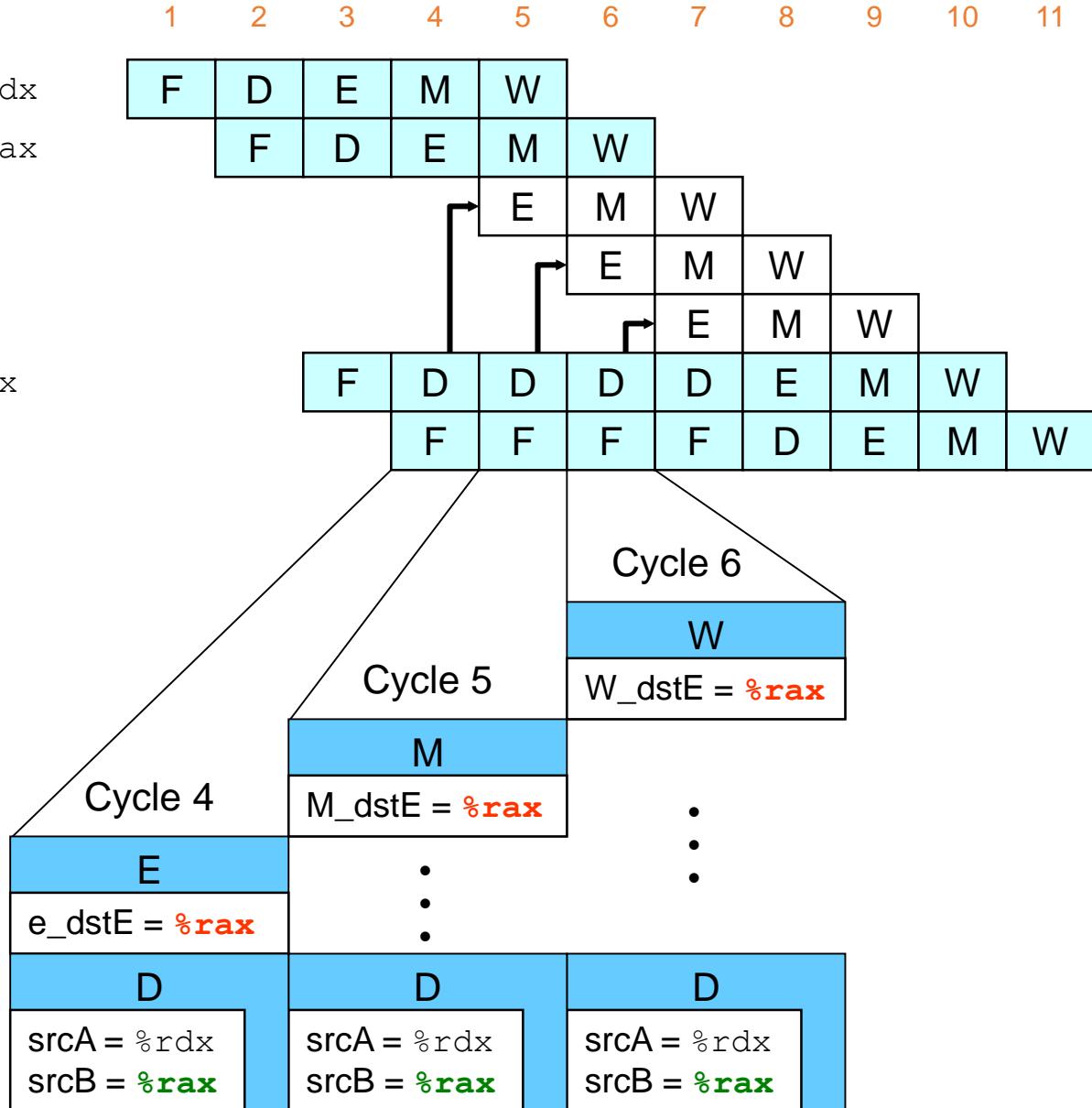
**bubble**

**bubble**

**bubble**

0x014: addq %rdx,%rax

0x016: halt



# What Happens When Stalling?

```
# demo-h0.ys
```

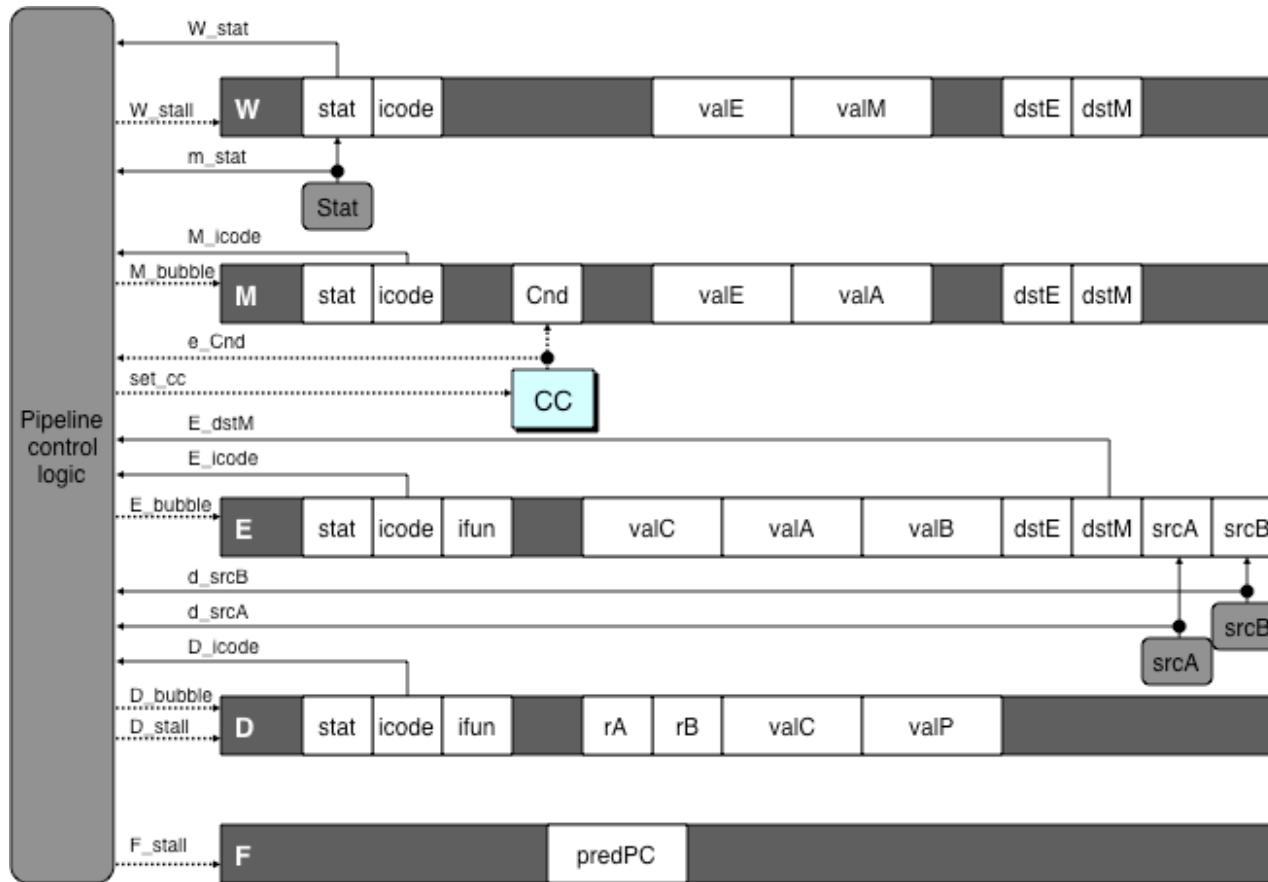
```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8

Write Back	<i>bubble</i>
Memory	<i>bubble</i>
Execute	0x014: addq %rdx,%rax
Decode	0x016: halt
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

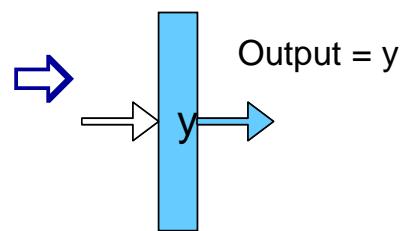
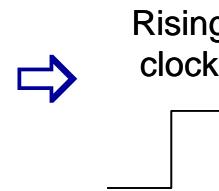
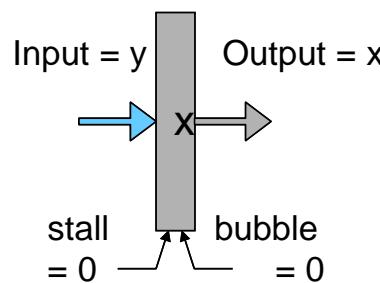
# Implementing Stalling



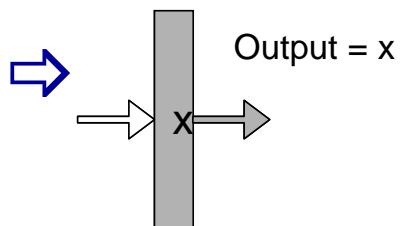
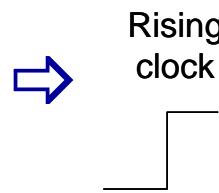
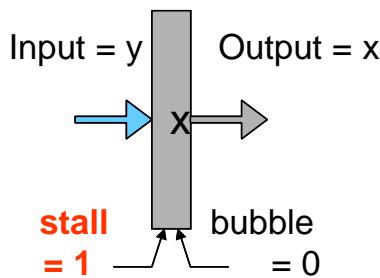
- Pipeline Control
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should update

# Pipeline Register Modes

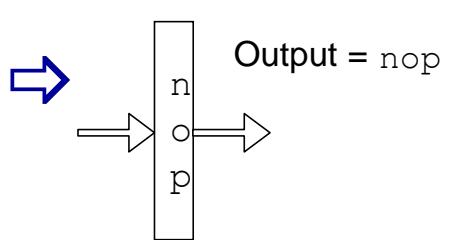
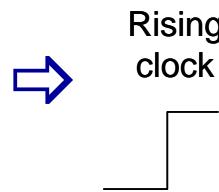
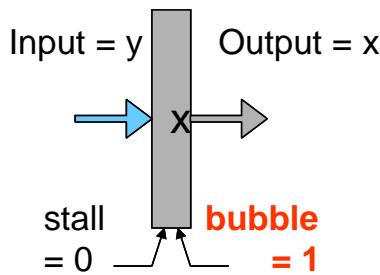
**Normal**



**Stall**



**Bubble**



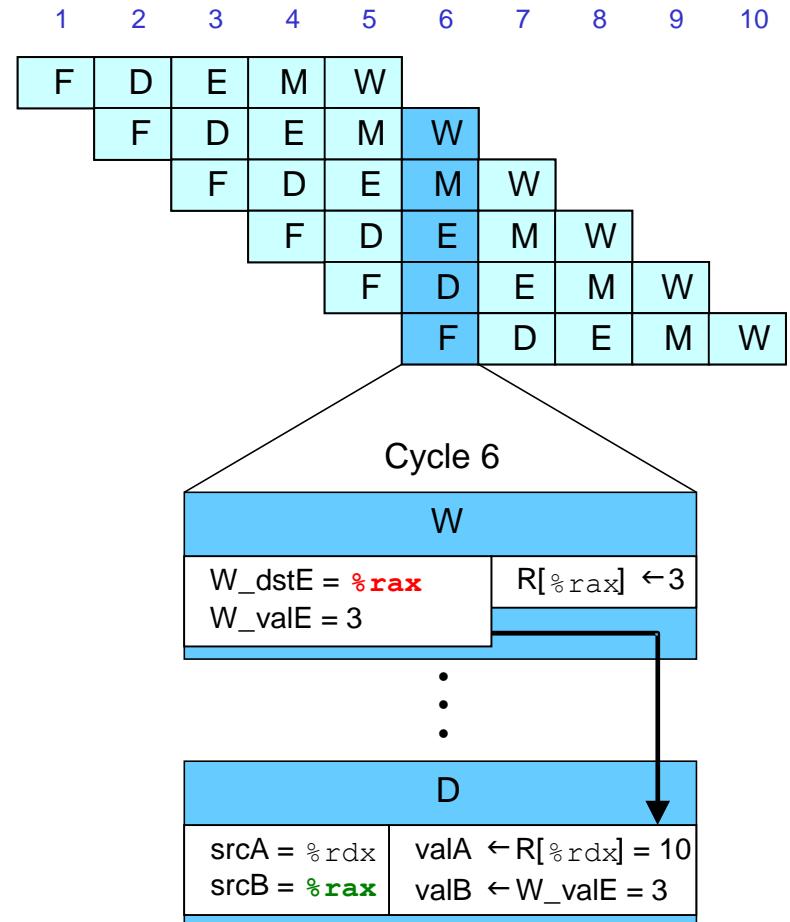
# Data Forwarding

- Naïve Pipeline
  - Register isn't written until completion of write-back stage
  - Source operands read from register file in decode stage
    - Needs to be in register file at start of stage
- Observation
  - Value generated in execute or memory stage
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

# Data Forwarding Example

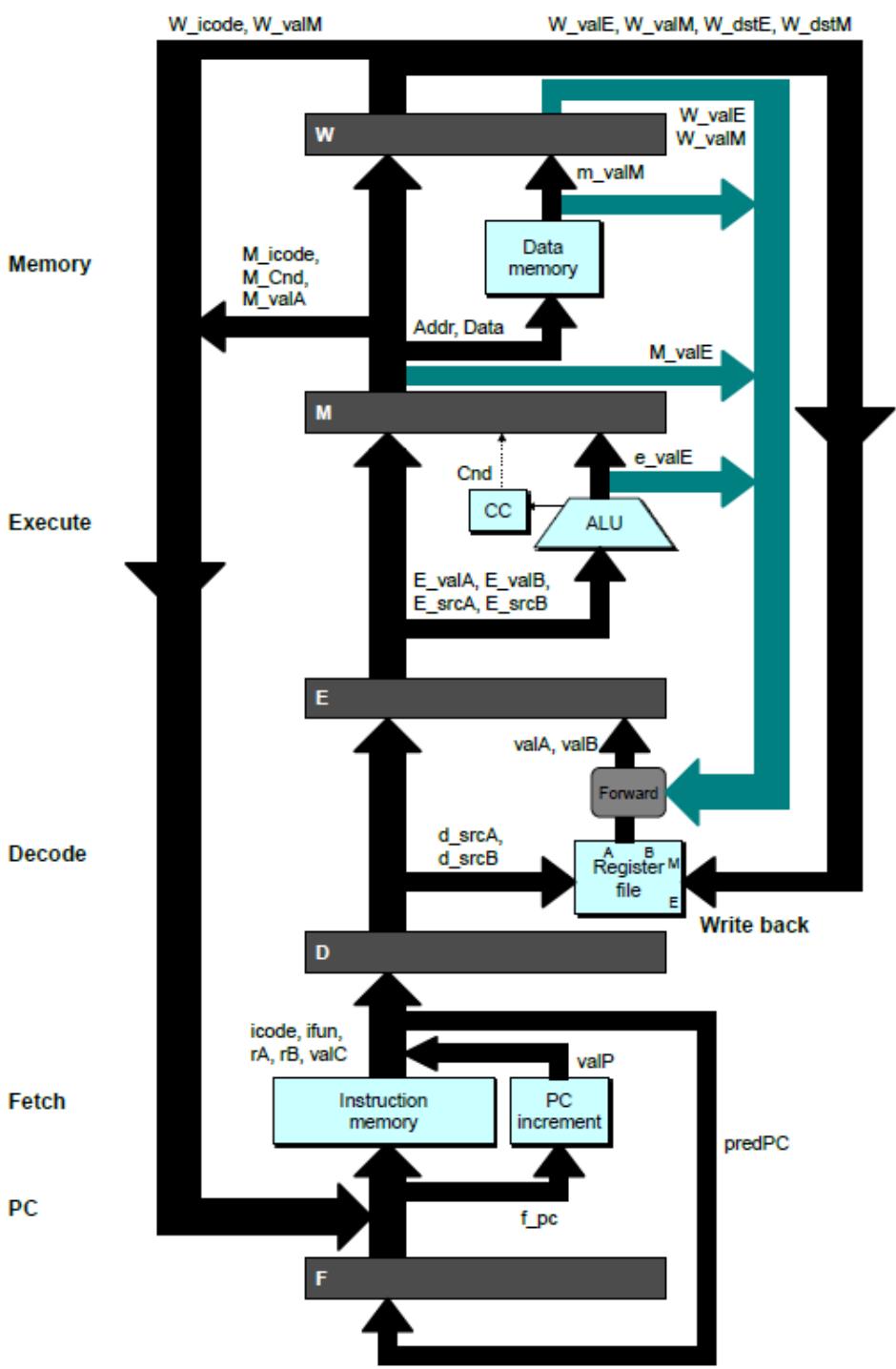
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage



# Bypass Paths

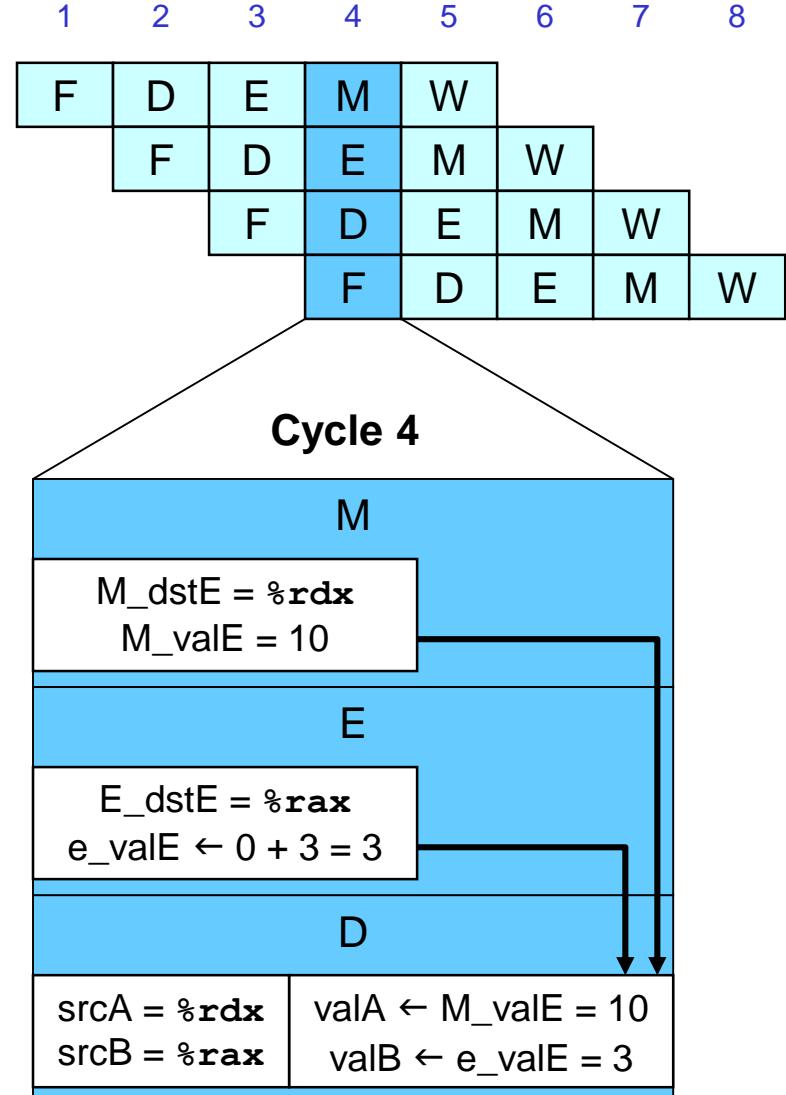
- Decode Stage
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage
- Forwarding Sources
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM



# Data Forwarding Example #2

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

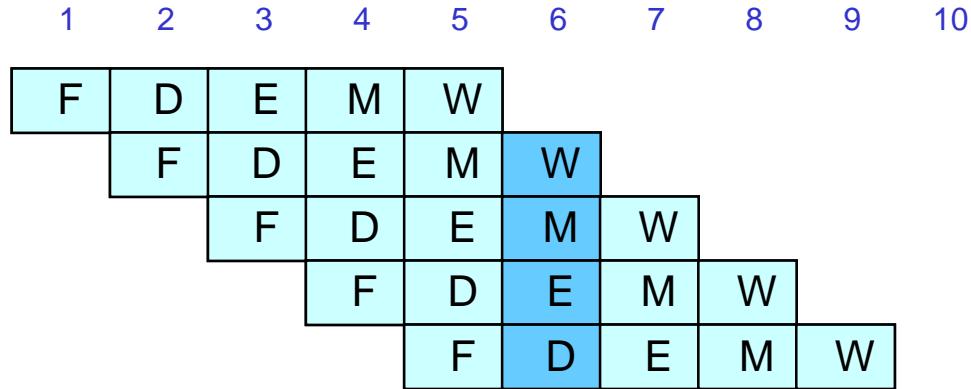
- Register `%rdx`
  - Generated by ALU during previous cycle
  - Forward from memory as `valA`
- Register `%rax`
  - Value just generated by ALU
  - Forward from execute as `valB`



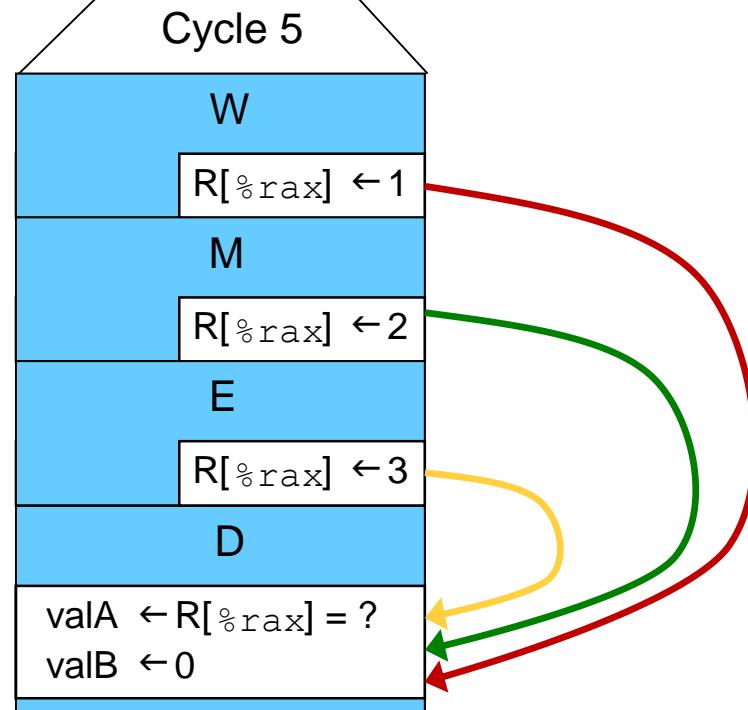
# Forwarding Priority

# demo-priority.ys

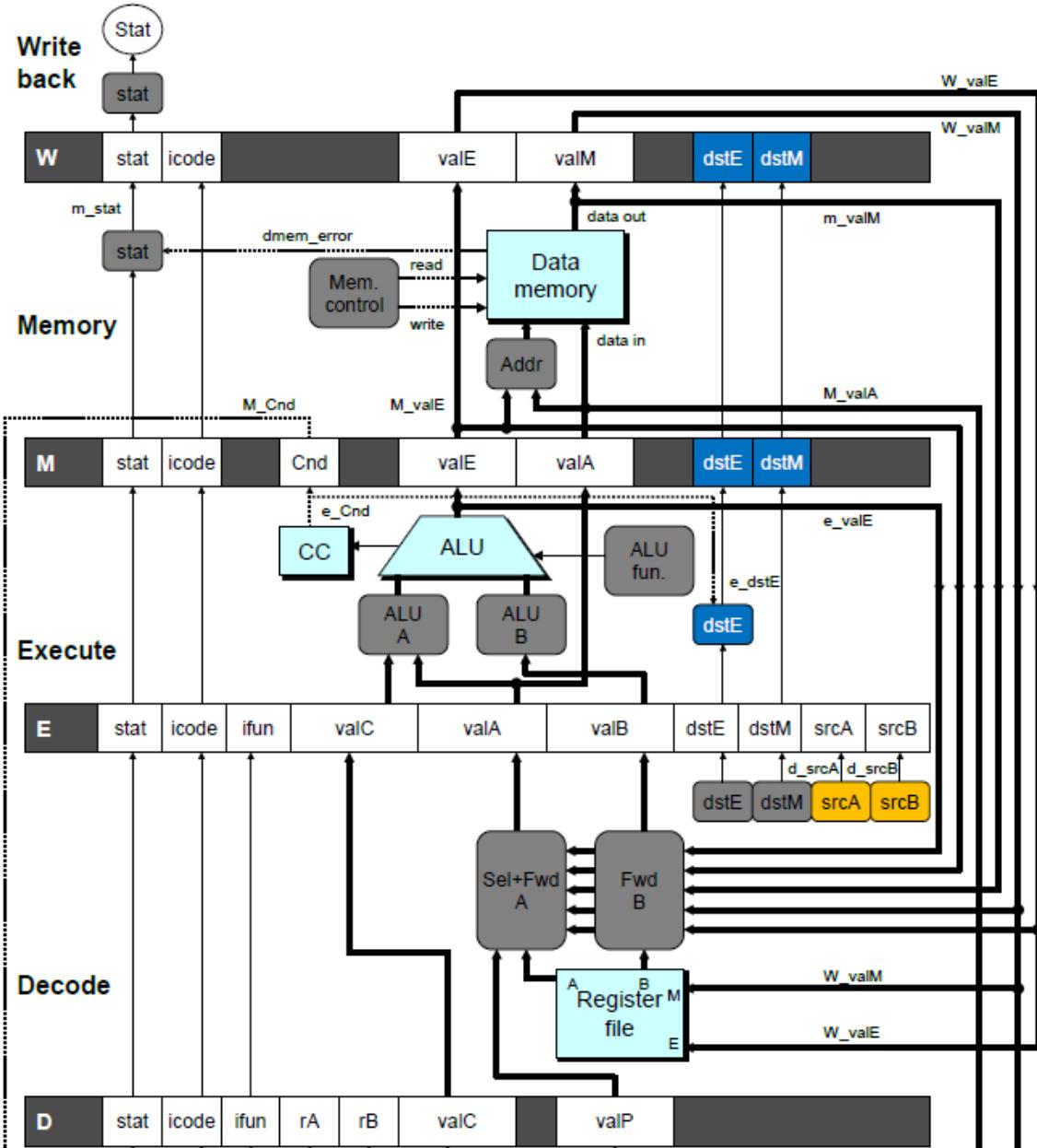
```
0x000: irmovq $1, %rax  
0x00a: irmovq $2, %rax  
0x014: irmovq $3, %rax  
0x01e: rrmovq %rax, %rdx  
0x020: halt
```



- Multiple Forwarding Choices
  - Which one should have priority
  - Match serial semantics
  - Use matching value from earliest pipeline stage



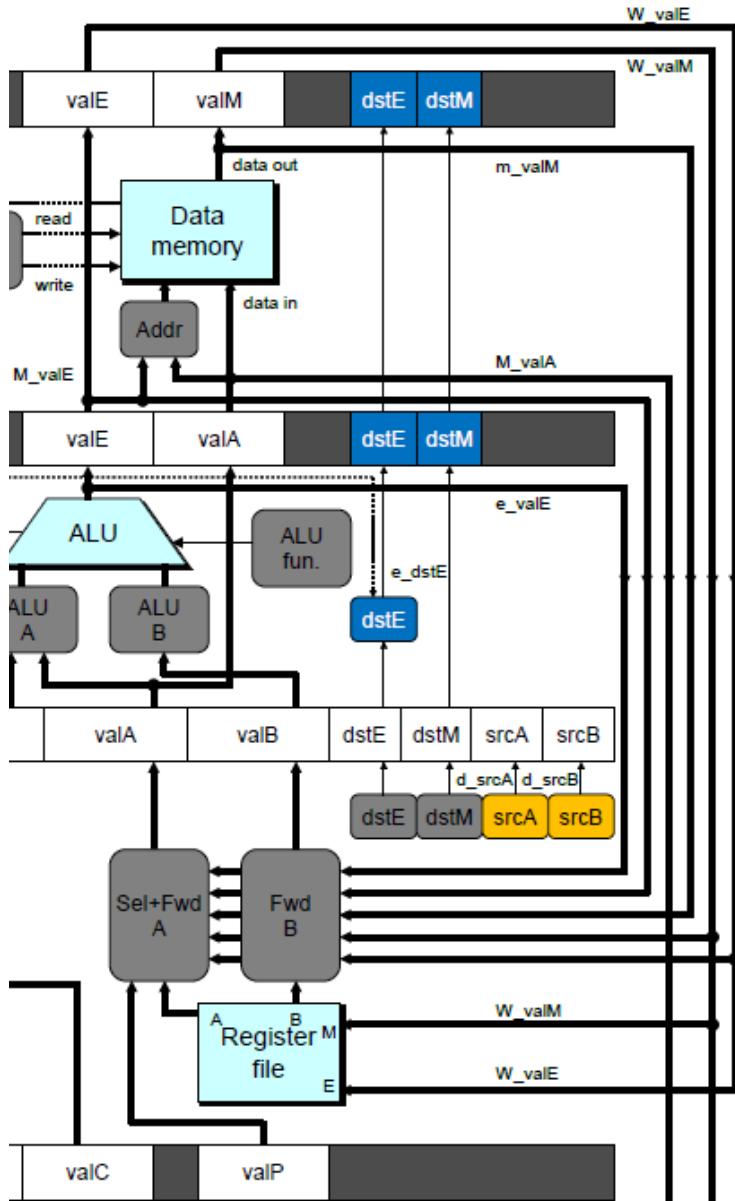
# Implementing Forwarding



Add additional feedback paths from E, M, and W pipeline registers into decode stage

Create logic blocks to select from multiple sources for valA and valB in decode stage

# Implementing Forwarding



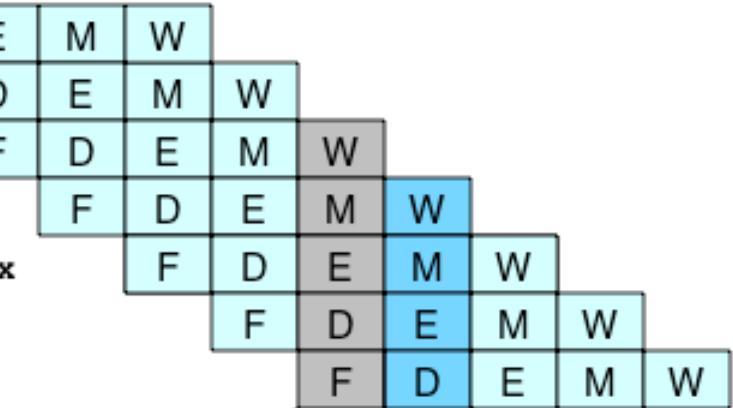
```
## What should be the A value?  
int d_valA = [  
    # Use incremented PC  
    D_icode in { ICALL, IJXX } : D_valP;  
    # Forward valE from execute  
    d_srcA == e_dstE : e_valE;  
    # Forward valM from memory  
    d_srcA == M_dstM : m_valM;  
    # Forward valE from memory  
    d_srcA == M_dstE : M_valE;  
    # Forward valM from write back  
    d_srcA == W_dstM : W_valM;  
    # Forward valE from write back  
    d_srcA == W_dstE : W_valE;  
    # Use value read from register file  
    1 : d_rvalA;  
];
```

# Limitation of Forwarding

# demo-luh.ys

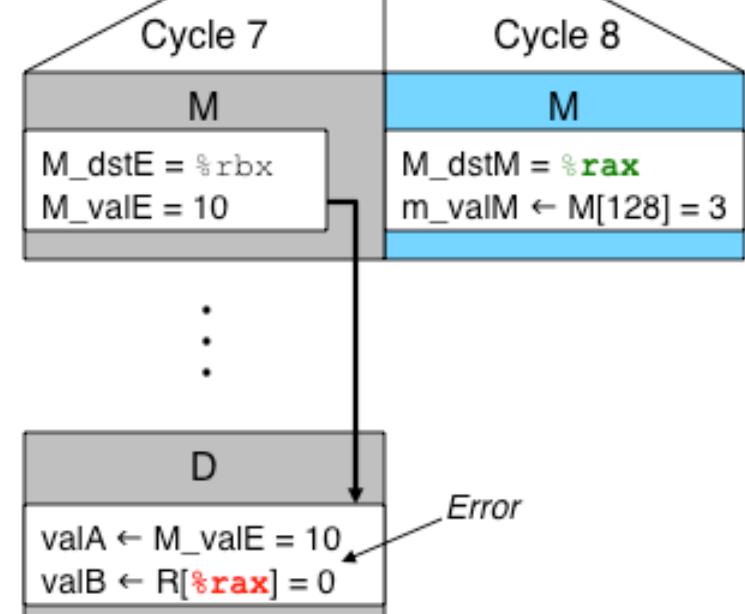
```
0x000: irmovq $128,%rdx  
0x00a: irmovq $3,%rcx  
0x014: rmmovq %rcx, 0(%rdx)  
0x01e: irmovq $10,%rbx  
0x028: mrmovq 0(%rdx),%rax # Load %rax  
0x032: addq %rbx,%rax # Use %rax  
0x034: halt
```

1 2 3 4 5 6 7 8 9 10 11



- Load-use dependency

- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8



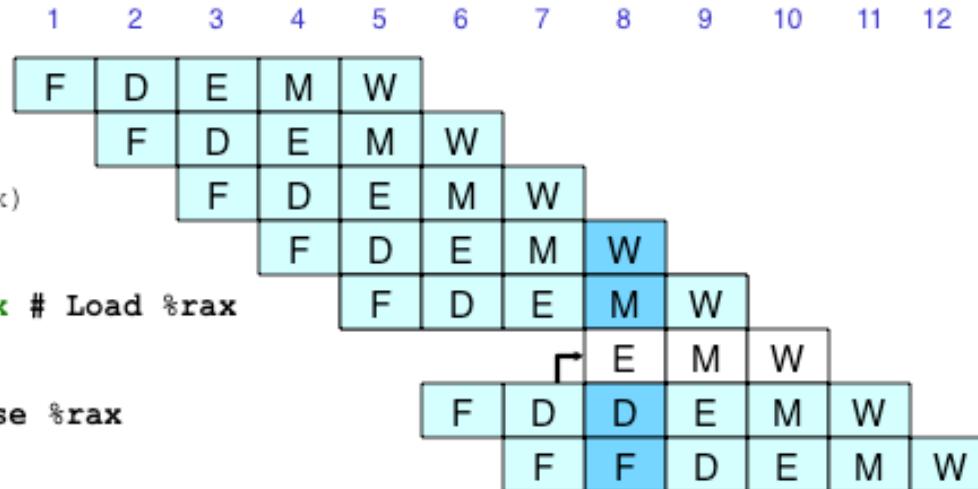
# Avoiding Load/Use Hazard

```
# demo-luh.ys
```

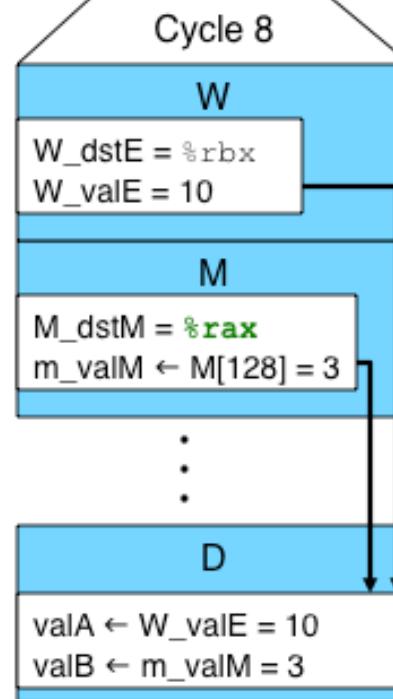
```
0x000: irmovq $128,%rdx  
0x00a: irmovq $3,%rcx  
0x014: rmmovq %rcx, 0(%rdx)  
0x01e: irmovq $10,%rbx  
0x028: mrmovq 0(%rdx),%rax # Load %rax
```

*bubble*

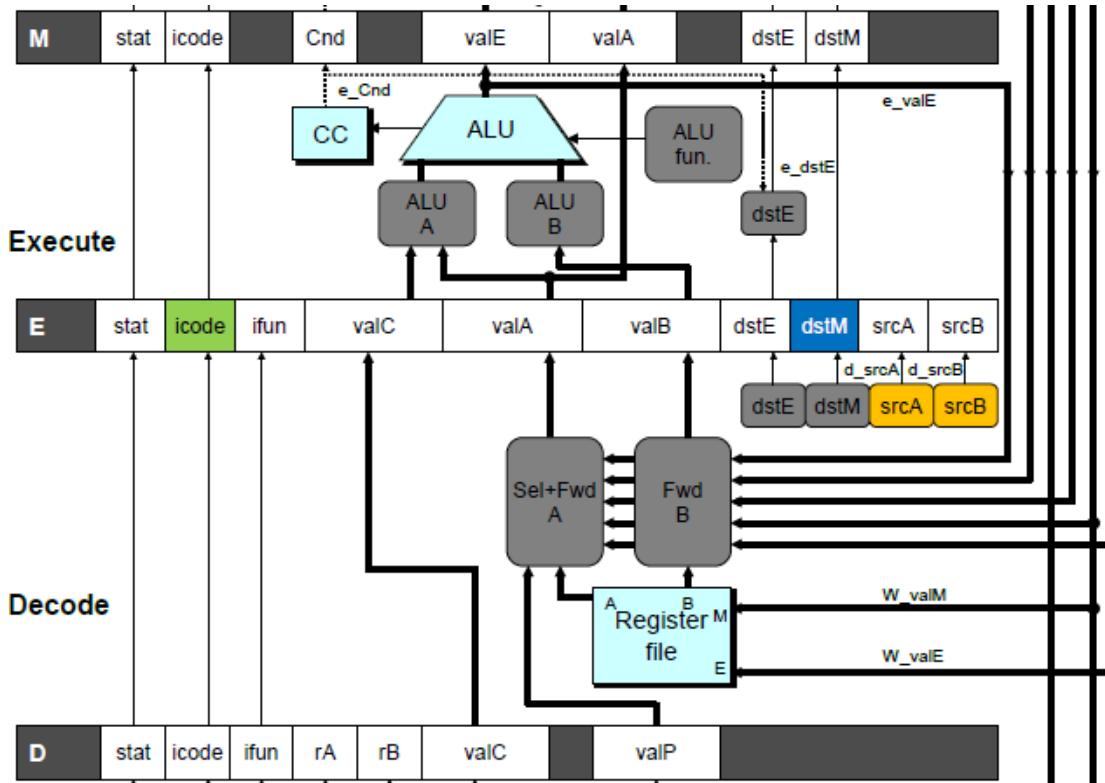
```
0x032: addq %rbx,%rax # Use %rax  
0x034: halt
```



- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage



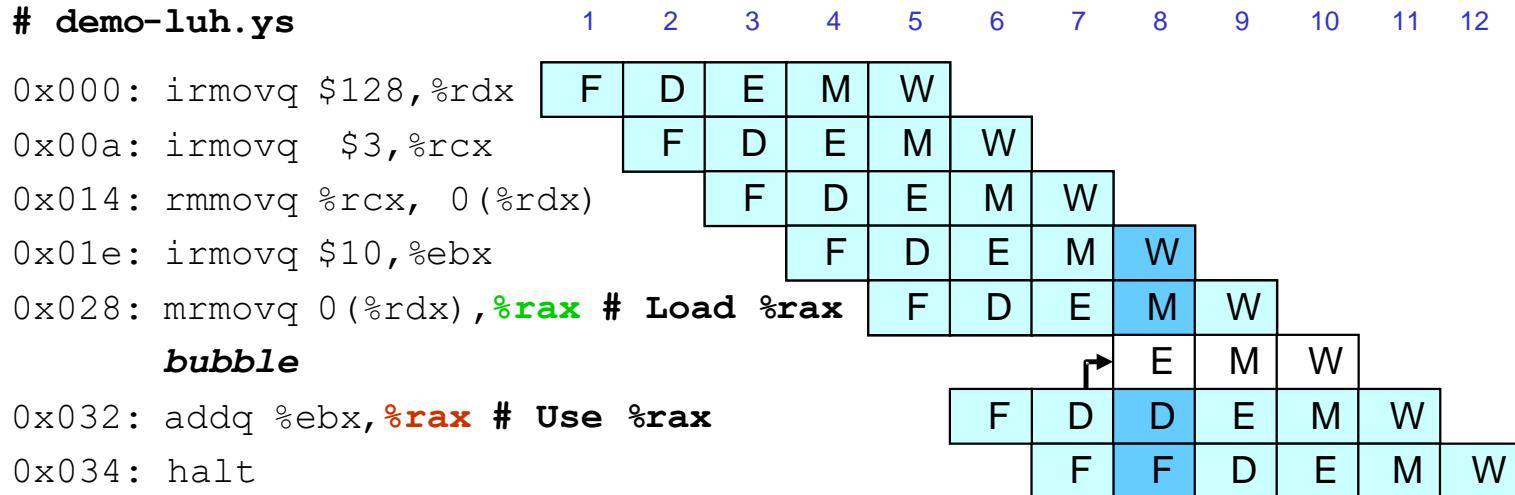
# Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	<code>E_icode in { IMRMOVQ, IPOPQ } &amp;&amp; E_dstM in { d_srcA, d_srcB }</code>

# Control for Load/Use Hazard

```
# demo-luh.ys
```



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

# Branch Misprediction Example

demo-j.ys

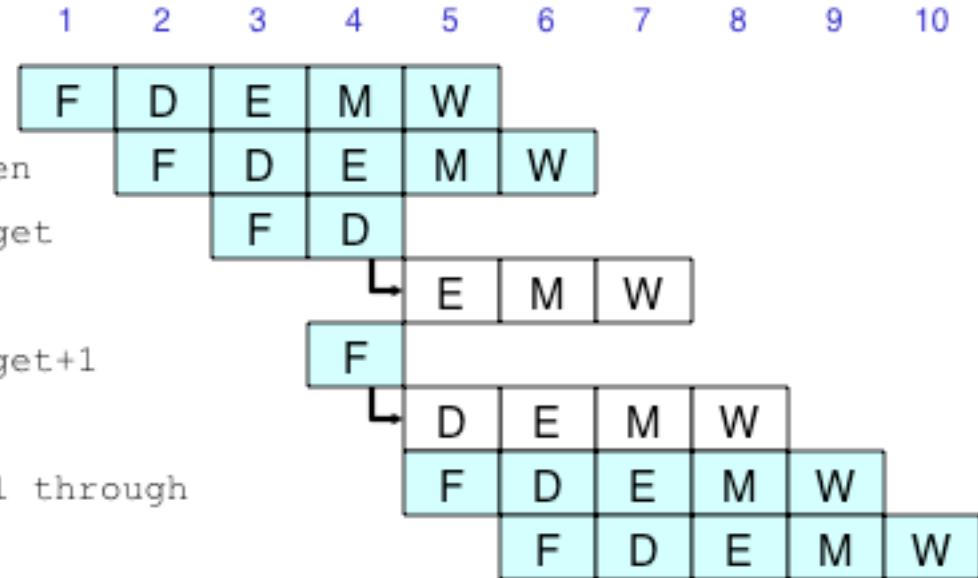
```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target
0x023: irmovq $4, %rcx # Should not execute
0x02d: irmovq $5, %rdx # Should not execute
```

- Should only execute first 8 instructions

# Handling Misprediction

```
# demo-j.ys
```

```
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
          bubble
0x020: irmovq $3,%rbx # Target+1
          bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



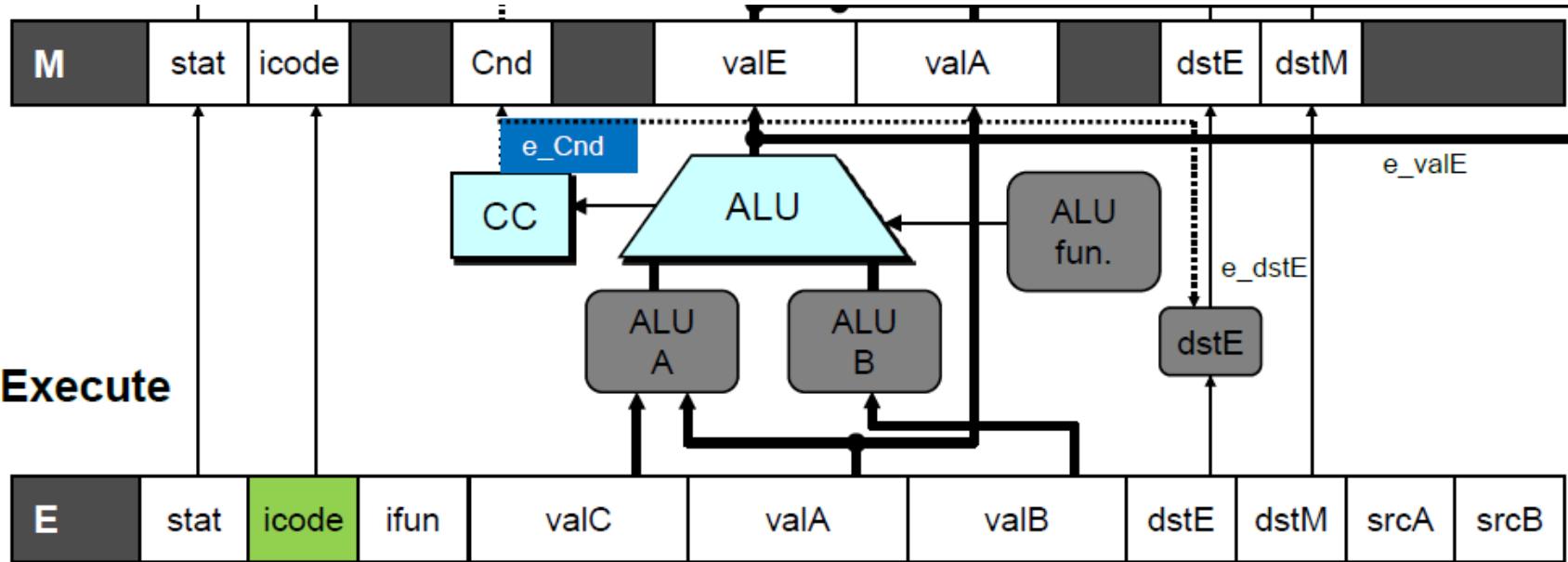
## Predict branch as taken

- Fetch 2 instructions at target

## Cancel when mispredicted

- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects have occurred yet

# Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	$E\_icode = IJXX \& !e\_Cnd$

# Control for Misprediction

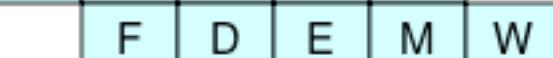
```
# demo-j.ys
```

```
0x000: xorq %rax,%rax
```

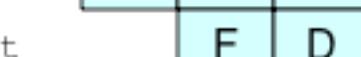
1 2 3 4 5 6 7 8 9 10



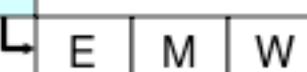
```
0x002: jne target # Not taken
```



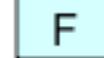
```
0x016: irmovq $2,%rdx # Target
```



*bubble*



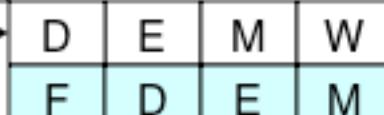
```
0x020: irmovq $3,%rbx # Target+1
```



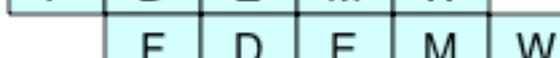
*bubble*



```
0x00b: irmovq $1,%rax # Fall through
```



```
0x015: halt
```



Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

# Return Example

demo-retb.ys

```
0x000:    irmovq Stack,%rsp    # Intialize stack pointer
0x00a:    call p              # Procedure call
0x013:    irmovq $5,%rsi      # Return point
0x01d:    halt
0x020: .pos 0x20
0x020: p: irmovq $-1,%rdi    # procedure
0x02a:    ret
0x02b:    irmovq $1,%rax      # Should not be executed
0x035:    irmovq $2,%rcx      # Should not be executed
0x03f:    irmovq $3,%rdx      # Should not be executed
0x049:    irmovq $4,%rbx      # Should not be executed
0x100: .pos 0x100
0x100: Stack:                # Stack: Stack pointer
```

- Previously executed three additional instructions

# Correct Return Example

```
# demo-retb
```

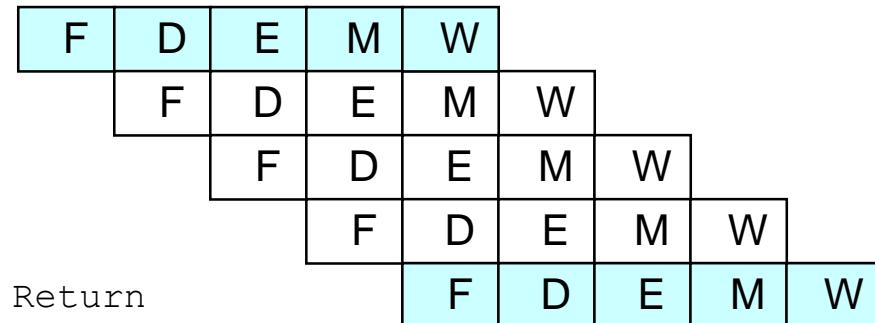
```
0x026:    ret
```

*bubble*

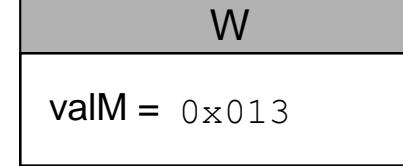
*bubble*

*bubble*

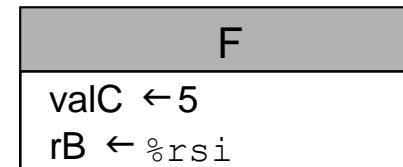
```
0x013:    irmovq $5,%rsi # Return
```



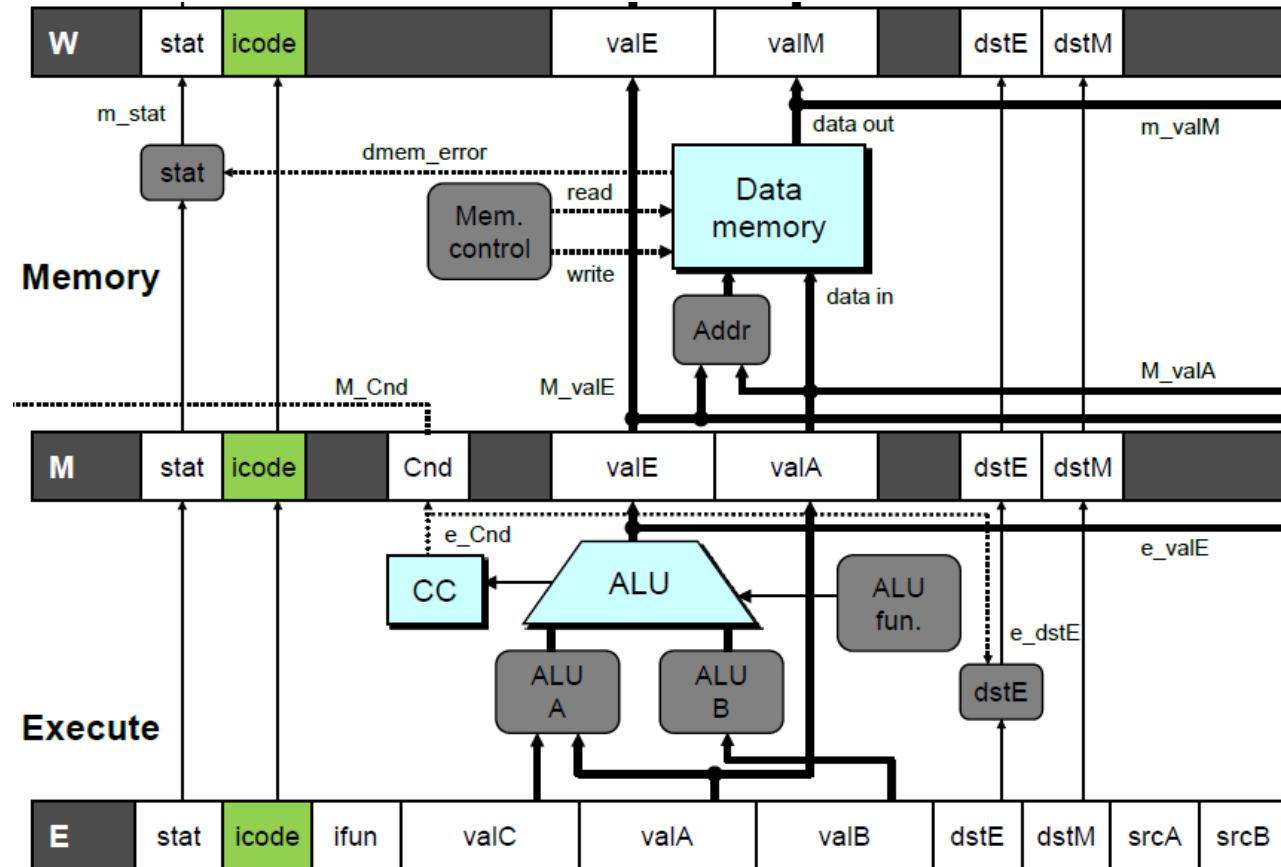
- As `ret` passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage



•  
•  
•



# Detecting Return



Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }

# Control for Return

```
# demo-retb
```

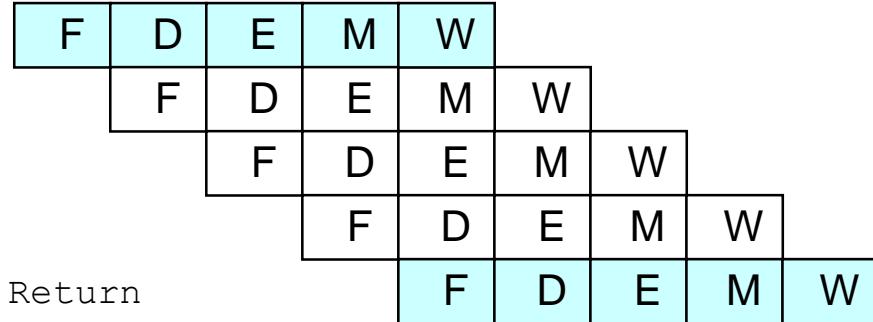
```
0x026:    ret
```

**bubble**

**bubble**

**bubble**

```
0x014:    irmovq $5,%rsi # Return
```



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal

# Special Control Cases

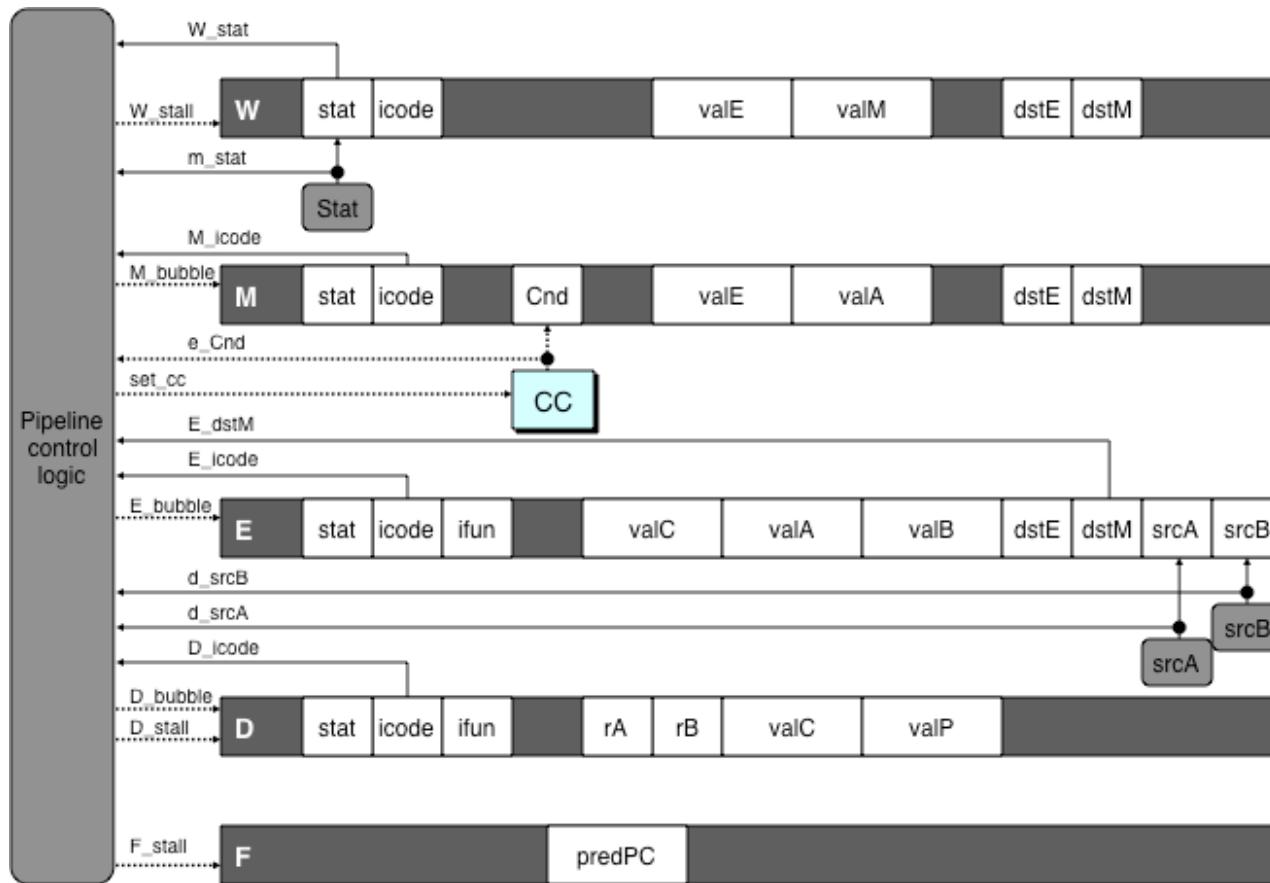
- Detection

Condition	Trigger
Processing <code>ret</code>	<code>IRET</code> in { <code>D_icode</code> , <code>E_icode</code> , <code>M_icode</code> }
Load/Use Hazard	<code>E_icode</code> in { <code>IMRMOVQ</code> , <code>IPOPQ</code> } && <code>E_dstM</code> in { <code>d_srcA</code> , <code>d_srcB</code> }
Mispredicted Branch	<code>E_icode = IJXX &amp; !e_Cnd</code>

- Action (on next cycle)

Condition	F	D	E	M	W
Processing <code>ret</code>	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

# Implementing Pipeline Control



- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

# Initial Version of Pipeline Control

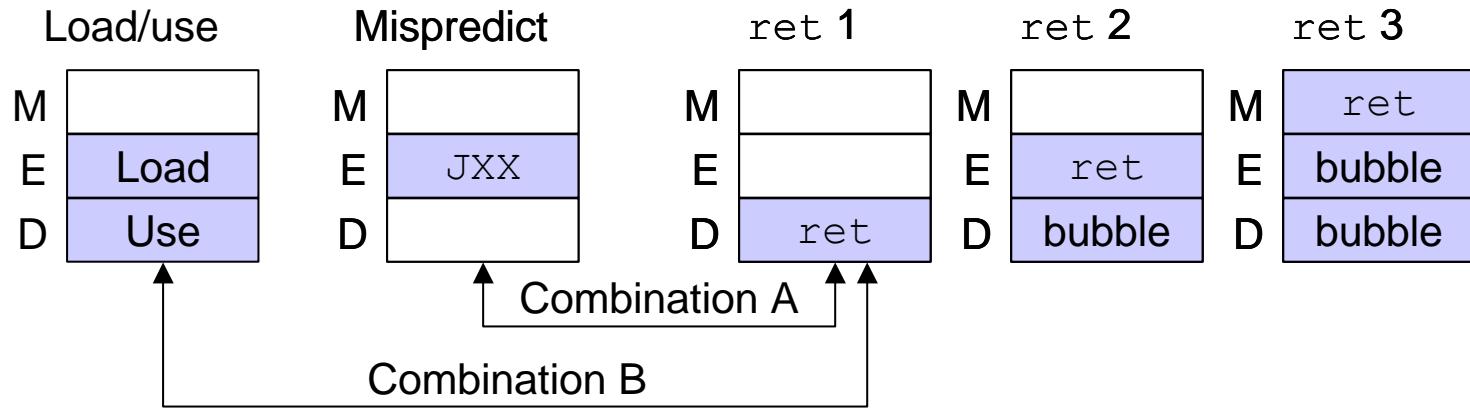
```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

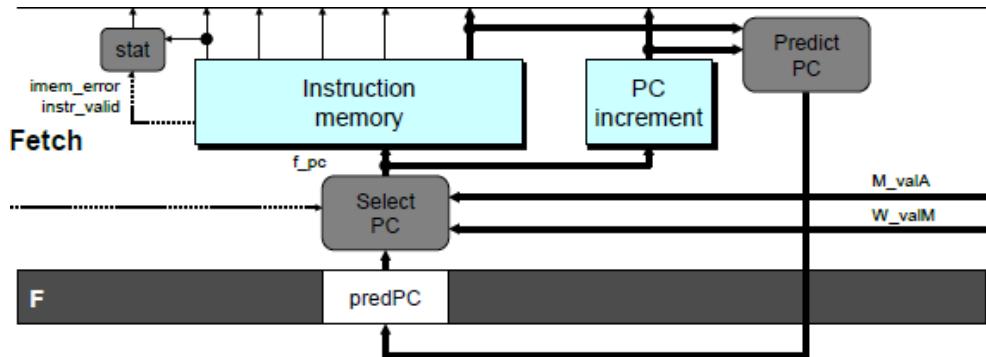
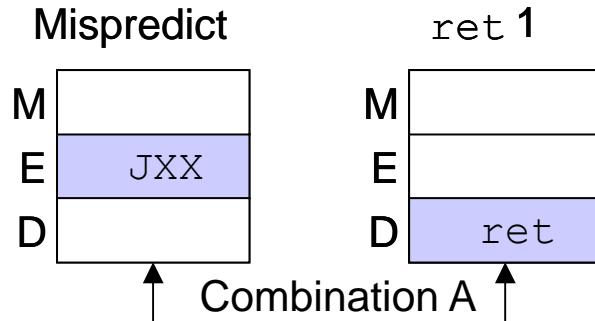
bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
```

# Control Combinations



- Special cases that can arise on same clock cycle
- Combination A
  - Not-taken branch
  - `ret` instruction at branch target
- Combination B
  - Instruction that reads from memory to `%rsp`
  - Followed by `ret` instruction

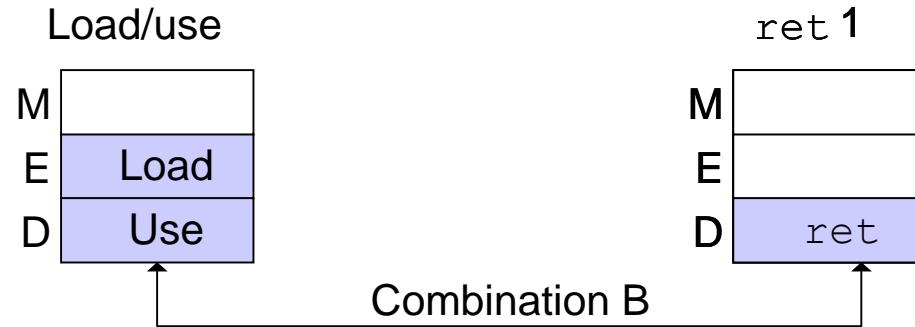
# Control Combination A



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	<i>stall</i>	<i>bubble</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M\_valM anyhow

# Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	<i>stall</i>	<i>bubble + stall</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Would attempt to bubble *and* stall pipeline register D
- Signaled by processor as pipeline error

# Handling Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	<i>stall</i>	<i>stall</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle

# Corrected Pipeline Control Logic

```
bool D_bubble =  
    # Mispredicted branch  
    (E_icode == IJXX && !e_Cnd) ||  
    # Stalling at fetch while ret passes through pipeline  
    IRET in { D_icode, E_icode, M_icode }  
    # but not condition for a load/use hazard  
    && !(E_icode in { IMRMOVQ, IPOPQ })  
    && E_dstM in { d_srcA, d_srcB } );
```

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	stall	bubble	normal	normal

- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle

# Pipeline Summary

- Data Hazards
  - Most handled by forwarding
    - No performance penalty
  - Load/use hazard requires one cycle stall
- Control Hazards
  - Cancel instructions when detect mispredicted branch
    - Two clock cycles wasted
  - Stall fetch stage while `ret` passes through pipeline
    - Three clock cycles wasted
- Control Combinations
  - Must analyze carefully
  - First version had subtle bug
    - Only arises with unusual instruction combination