CS-2011 — Machine Organization and Assembly Language — Recap

Professor Hugh C. Lauer
CS-2011, Machine Organization and Assembly Language

Traditional Course in Machine Organization and Assembly Language

- **Bits, bytes, gates, logic**
  - How the computer works inside
- **Von Neumann cycle**
  - Instruction fetch and execution
- **Machine code and Assembly language**
  - Writing out those instructions
- **Machine data types**
  - Integers, short, long
- **A few primitive algorithms**
  - Bubblesort in Assembly
- **…**

Traditional Course never gets to …

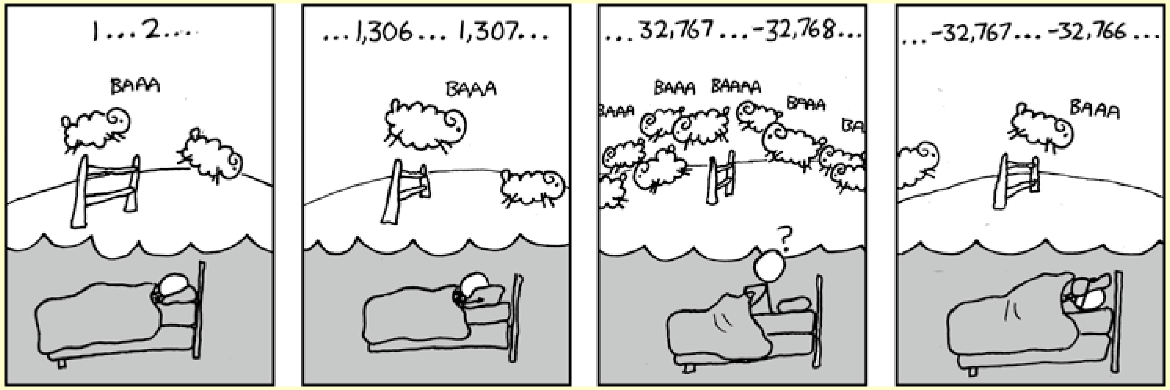
- Bits, bytes, gates, logic
  - How the computer works inside
- Von Neumann cycle
  - Instruction fetch and execution
- Machine code and Assembly lan
  - Writing out those instructions
- Machine data types
  - Integers, short, long
- A few primitive algorithms
  - Bubblesort in Assembly
- …

- **Floating point, other data types**
  - **How computer arithmetic works**
- **Representation of real programs**
  - **For-loops, if-else, switches, etc.**
  - **Functions, stack discipline**
  - **Parameters and arguments**
- **Things that matter at runtime …**
  - **Memory hierarchy**
  - **Cache performance**
- **… when the abstraction breaks down**
  - **Buffer overflow**

Great Reality #1:

Ints are not integers, floats are not reals

- **Example 1: Is $x^2 \geq 0$?**
  - Float's: Yes!
  - Int's:
    - 40000 * 40000 → 1,600,000,000
    - 50000 * 50000 → ??



-352,516,352

- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> ??

Source: xkcd.com/571

Memory referencing bug example

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  ➙    3.14
fun(1)  ➙    3.14
fun(2)  ➙    3.1399998664856
fun(3)  ➙    2.00000061035156
fun(4)  ➙    3.14, then segmentation fault
```

- **Result is architecture specific**

Memory system performance example

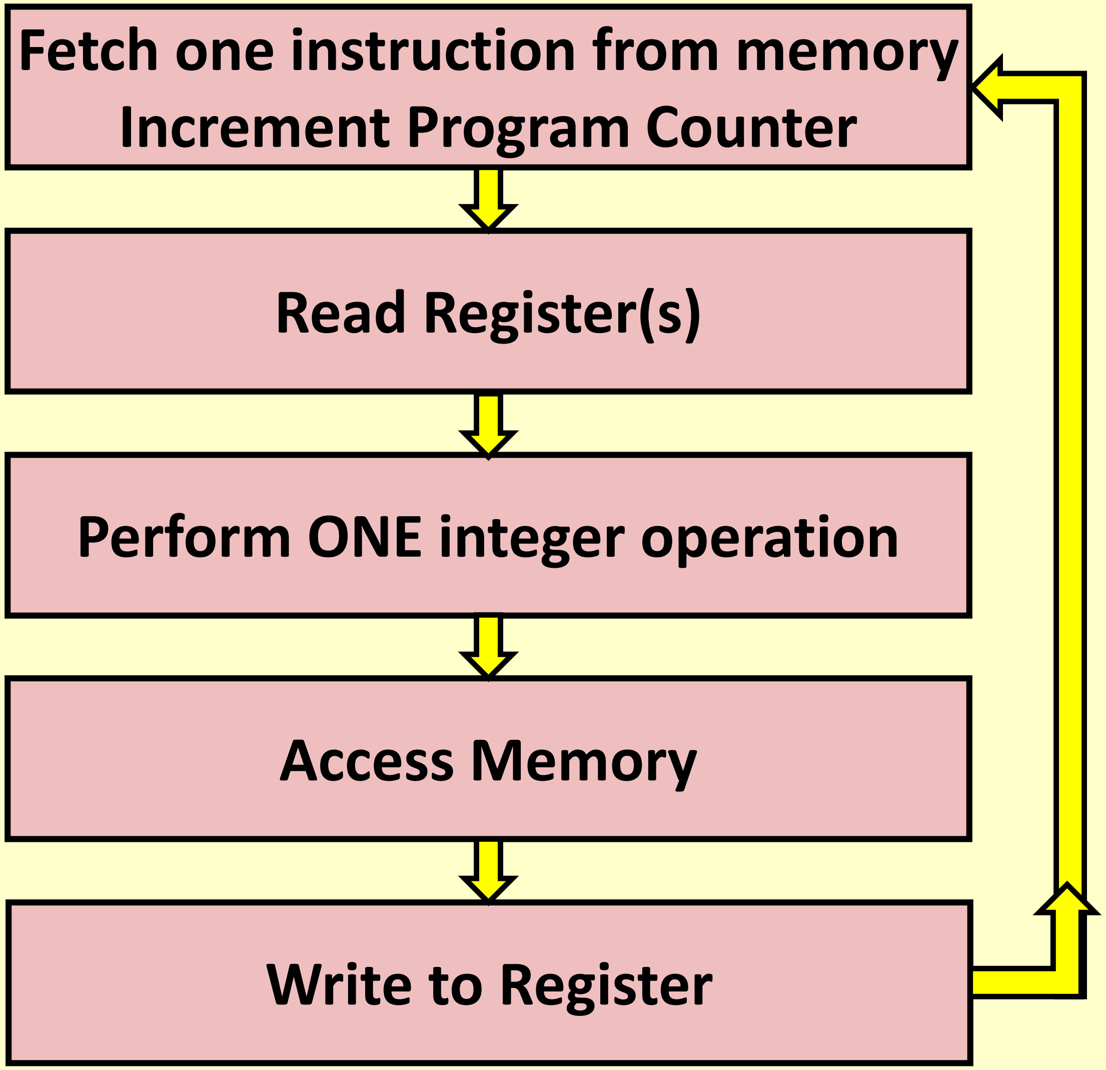
```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
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];
}
```

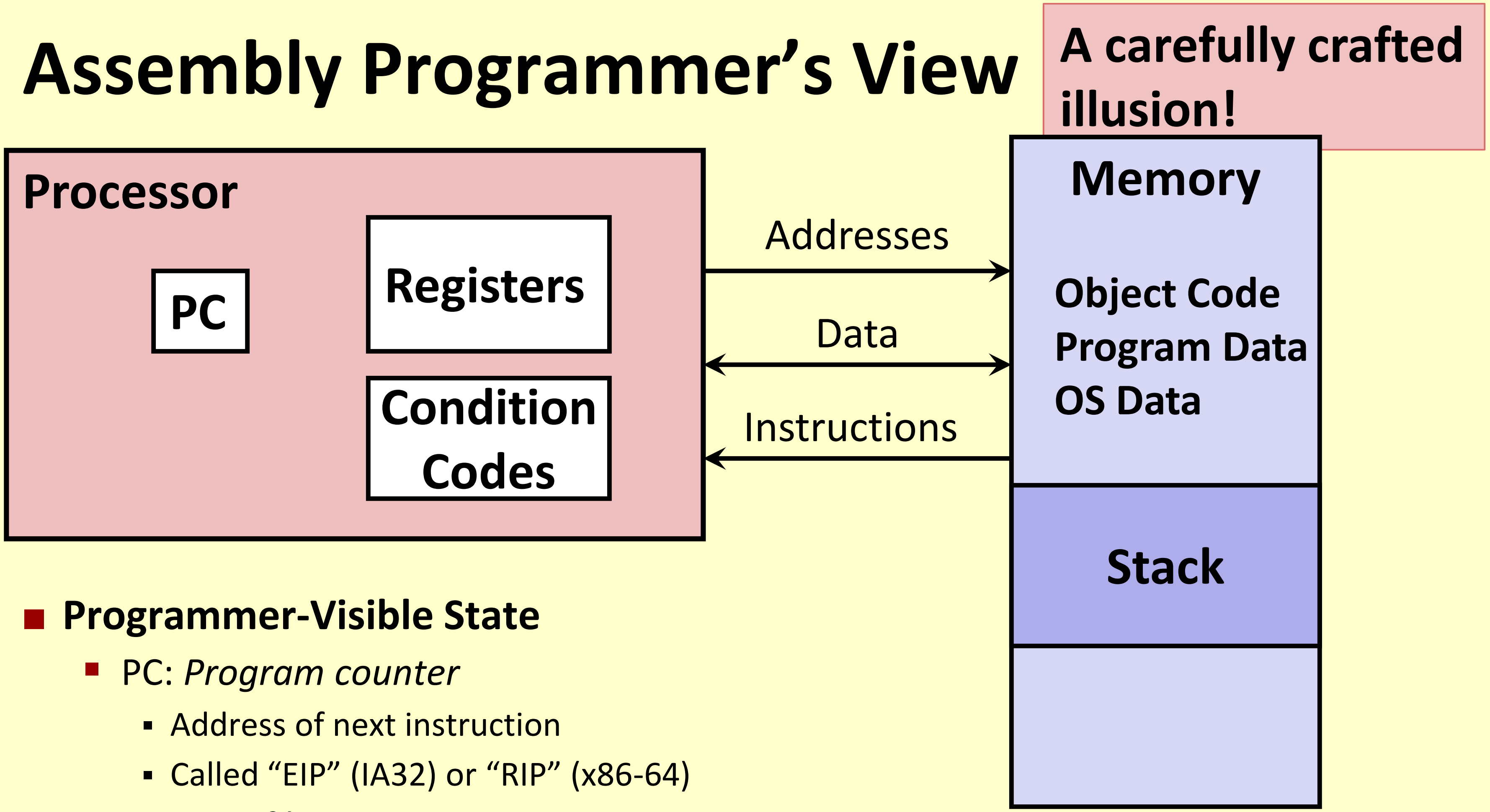
21 times slower (Pentium 4)

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

Execution Model for Modern Computers


Fetch one instruction from memory
Increment Program Counter
Read Register(s)
Perform ONE integer operation
Access Memory
Write to Register

Assembly Programmer's View



- **Programmer-Visible State**
  - PC: *Program counter*
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

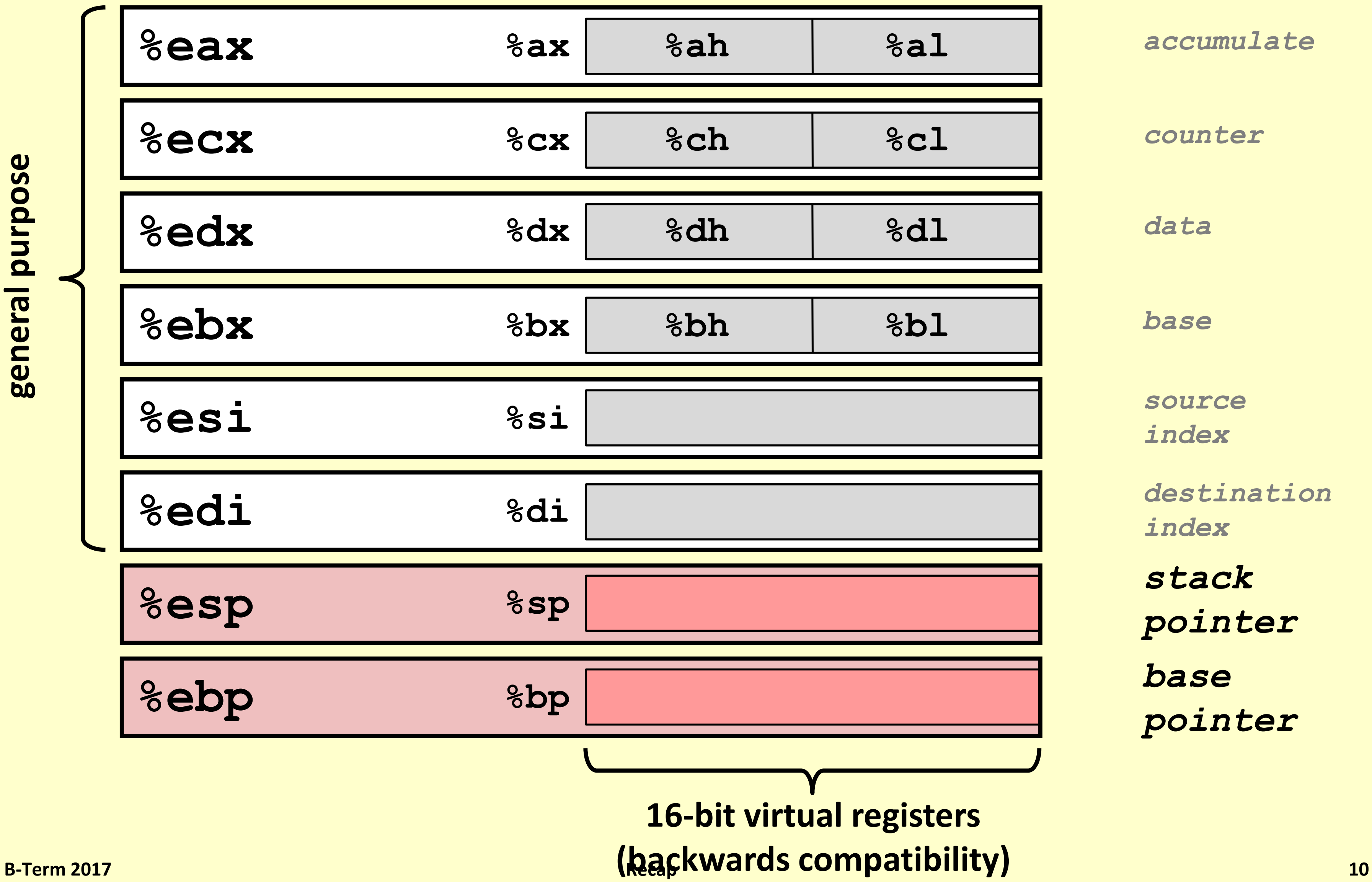
- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support functions

x86-64 Integer Registers

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

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

Integer Registers (IA32)



32-bit code for swap

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

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

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

   popl   %ebx                     }
   popl   %ebp                     } Finish
   ret                             }
```

64-bit code for swap

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

```
swap:
                                   } Set Up
      movl   (%rdi), %edx
      movl   (%rsi), %eax
      movl   %eax, (%rdi)          } Body
      movl   %edx, (%rsi)

      ret                          } Finish
```

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

"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

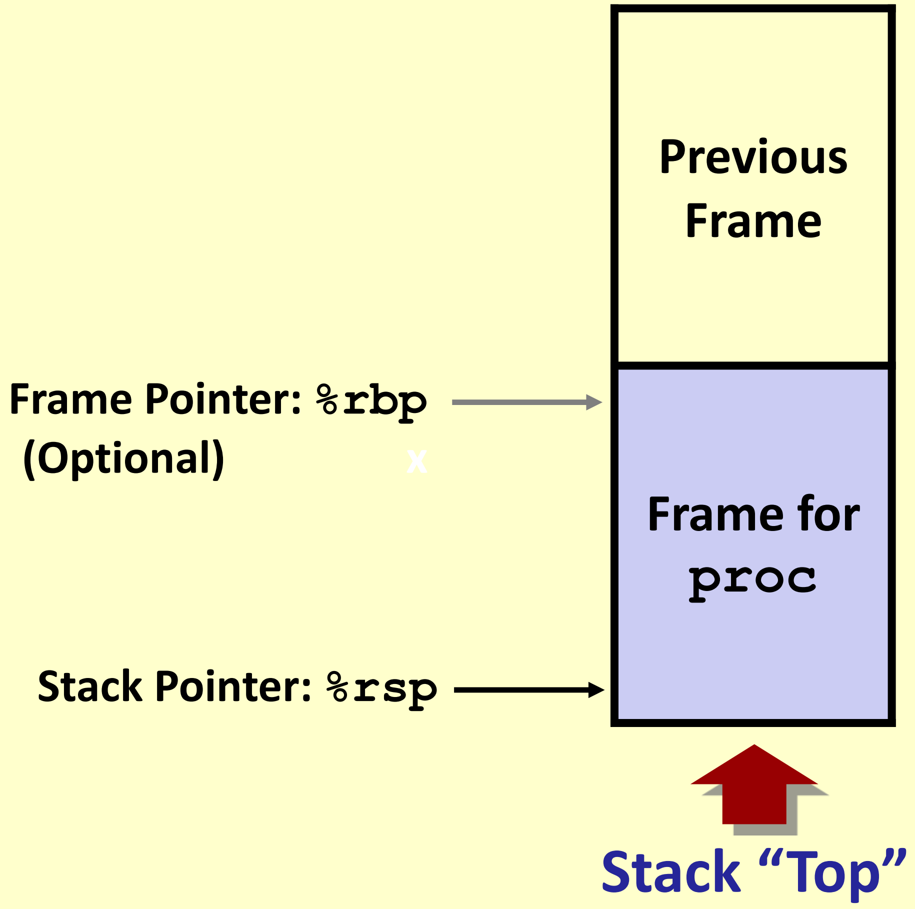
```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

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

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

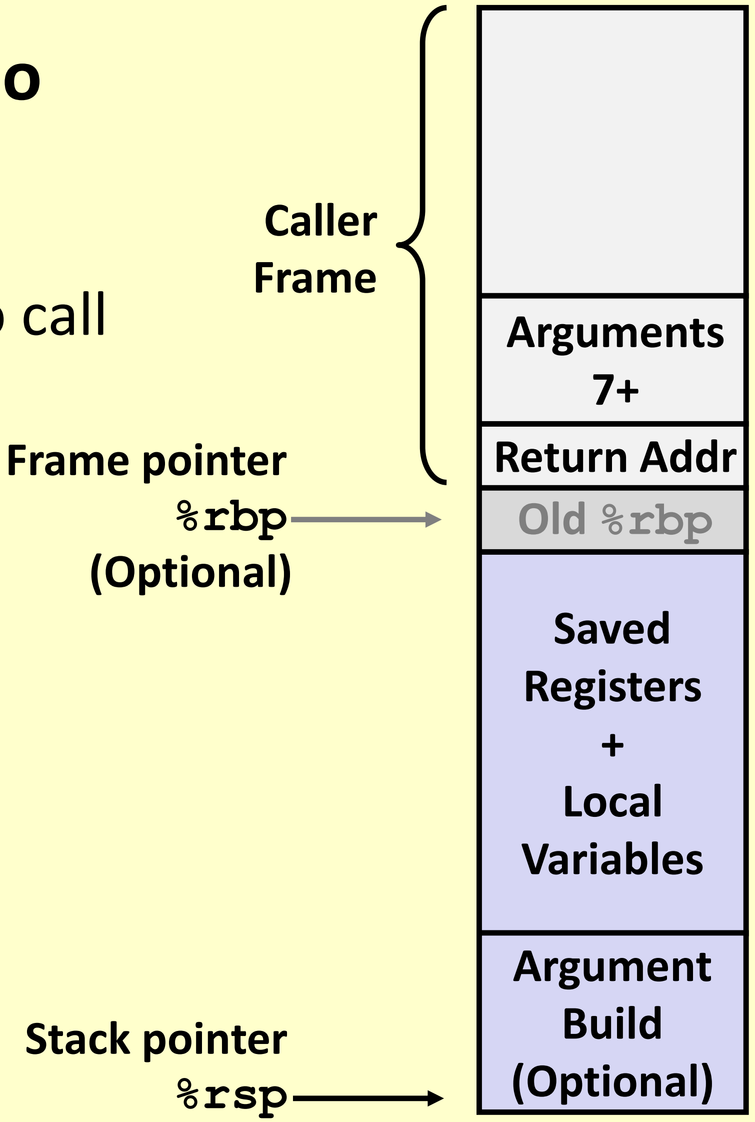


See also: Fig 3.25

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)

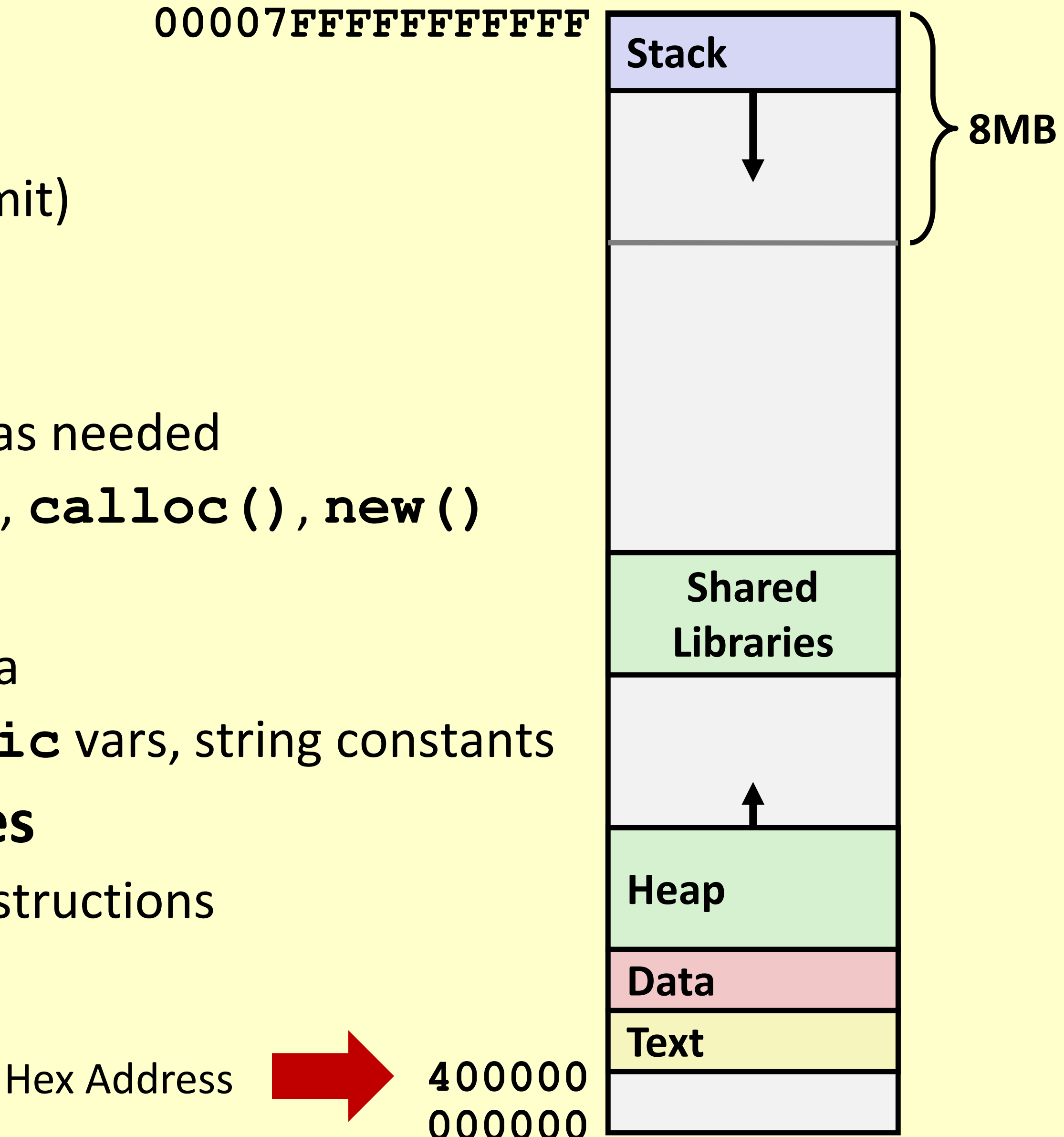
- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call


Caller Frame
Arguments 7+
Return Addr
Frame pointer %rbp (Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer %rsp

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

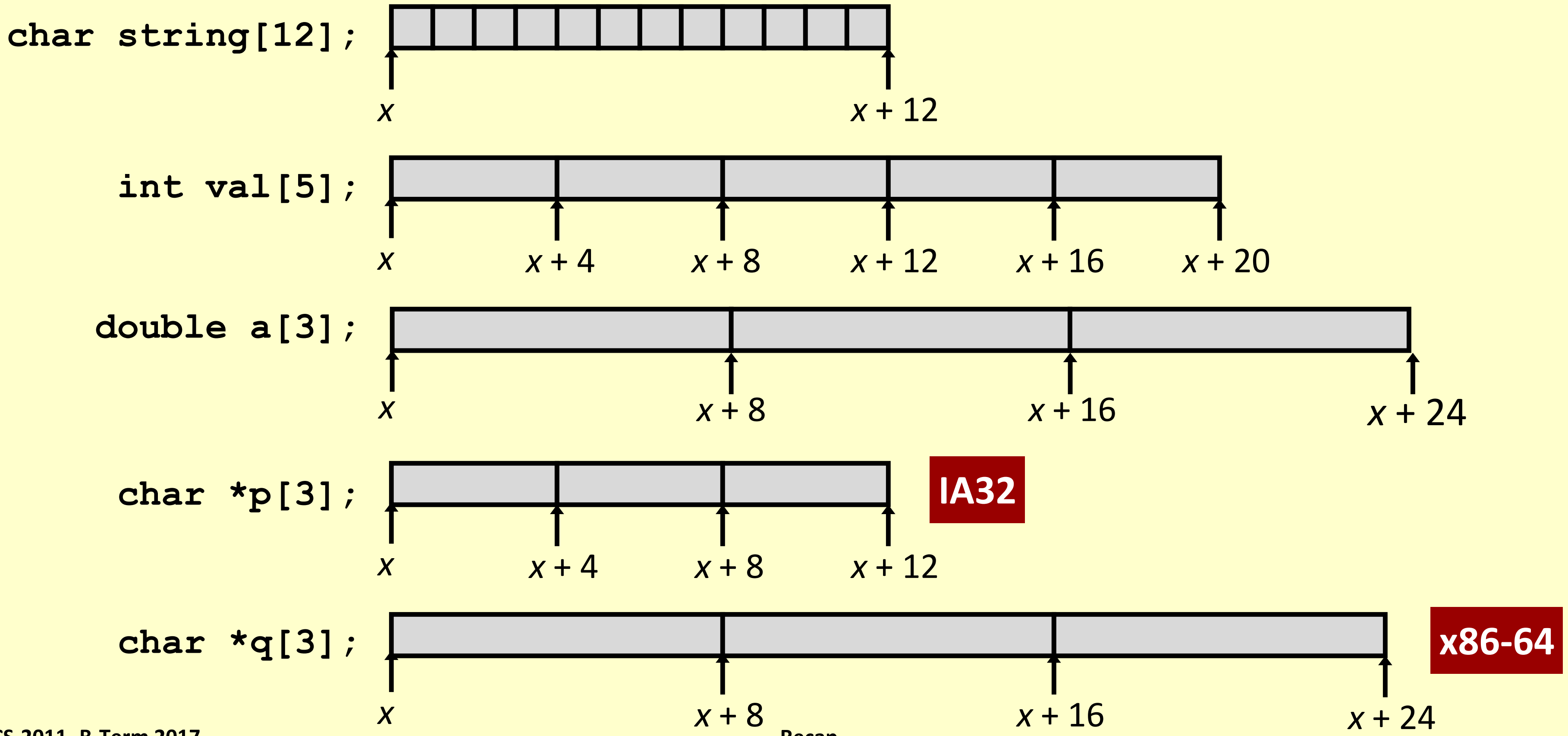


Array Allocation

- **Basic Principle**

  T **A[*L*];**

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes



Reduction in Strength

- **Replace costly operation with simpler one**
- **Shift, add instead of multiply or divide**

  ```
  16*x -->  x << 4
  ```

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- **Recognize sequence of products**

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

$\longrightarrow$

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

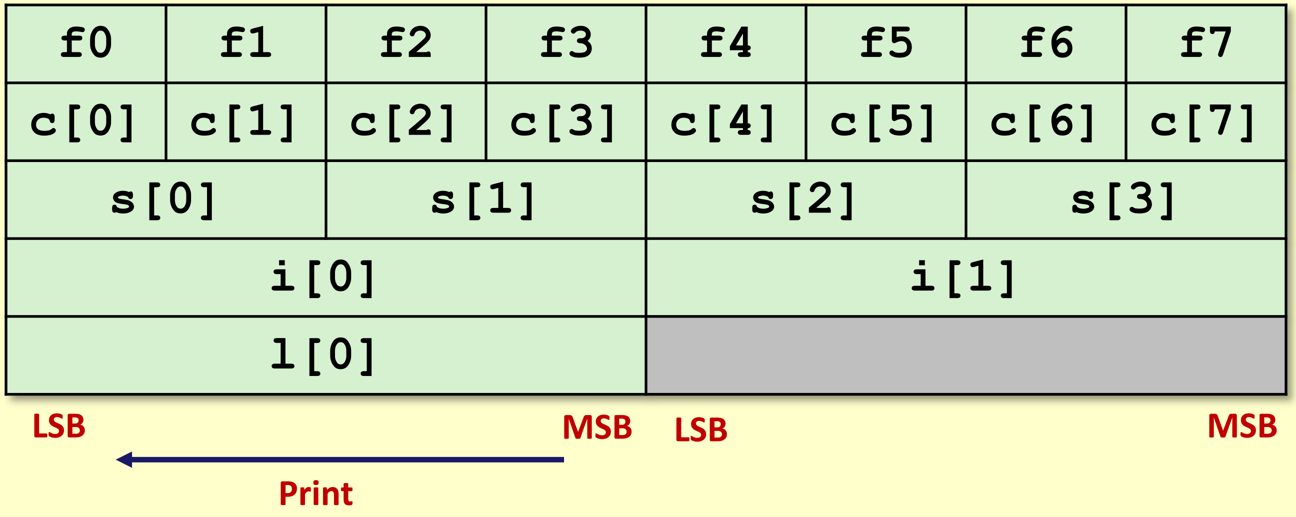
1 multiplication: i*n

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Byte Ordering on IA32

Little Endian

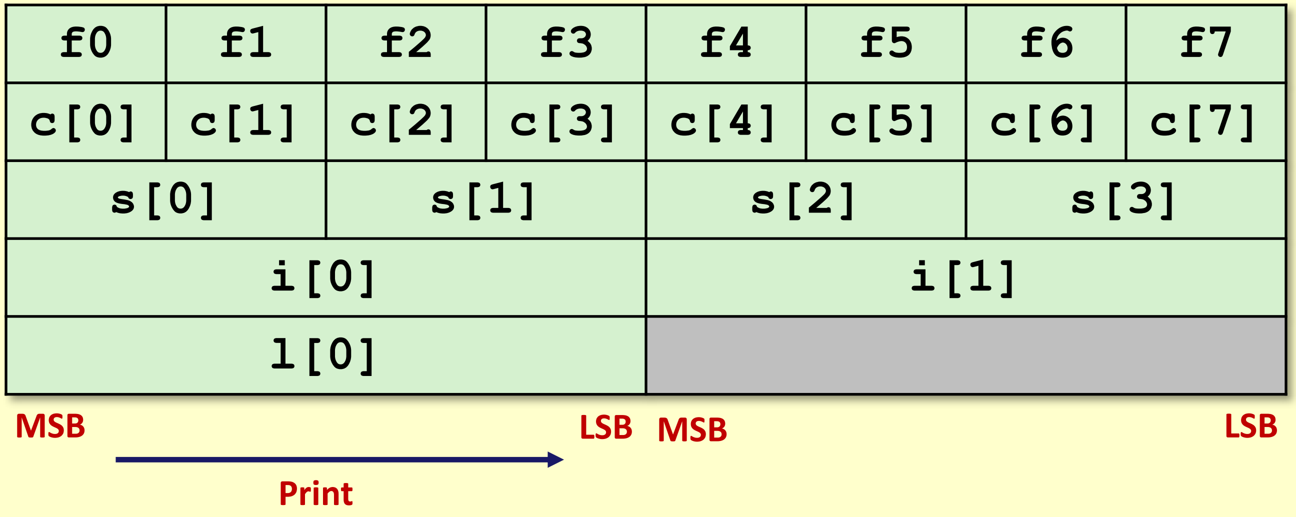


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian



Output on SPARC/IBM, etc.:

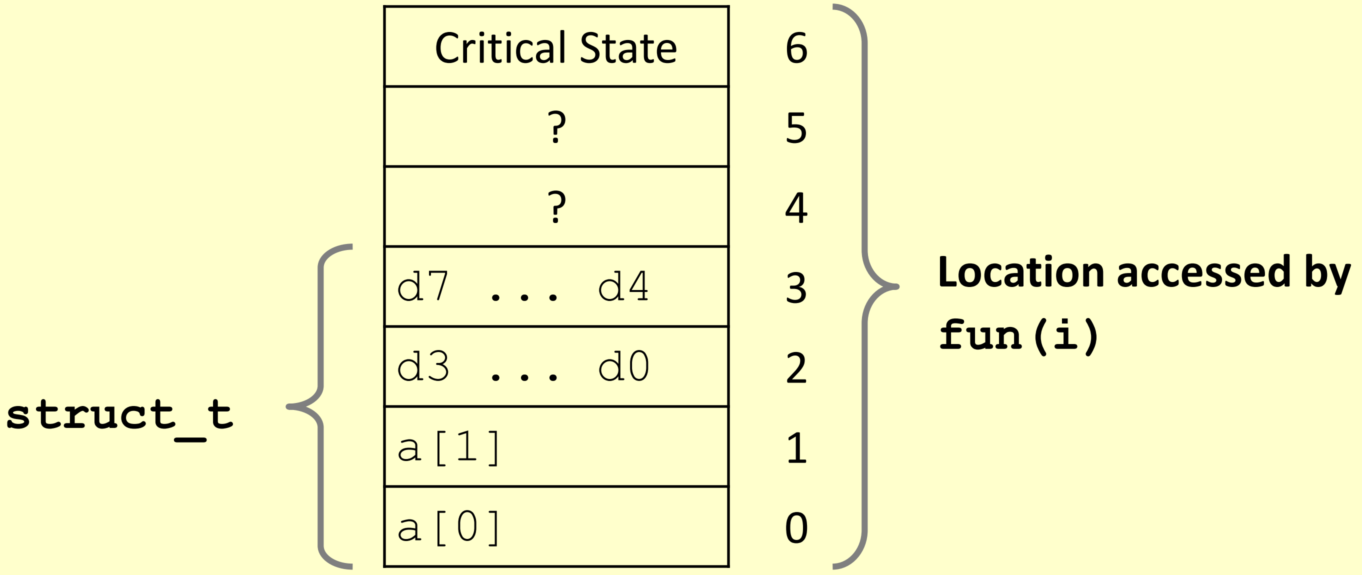
```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

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
```

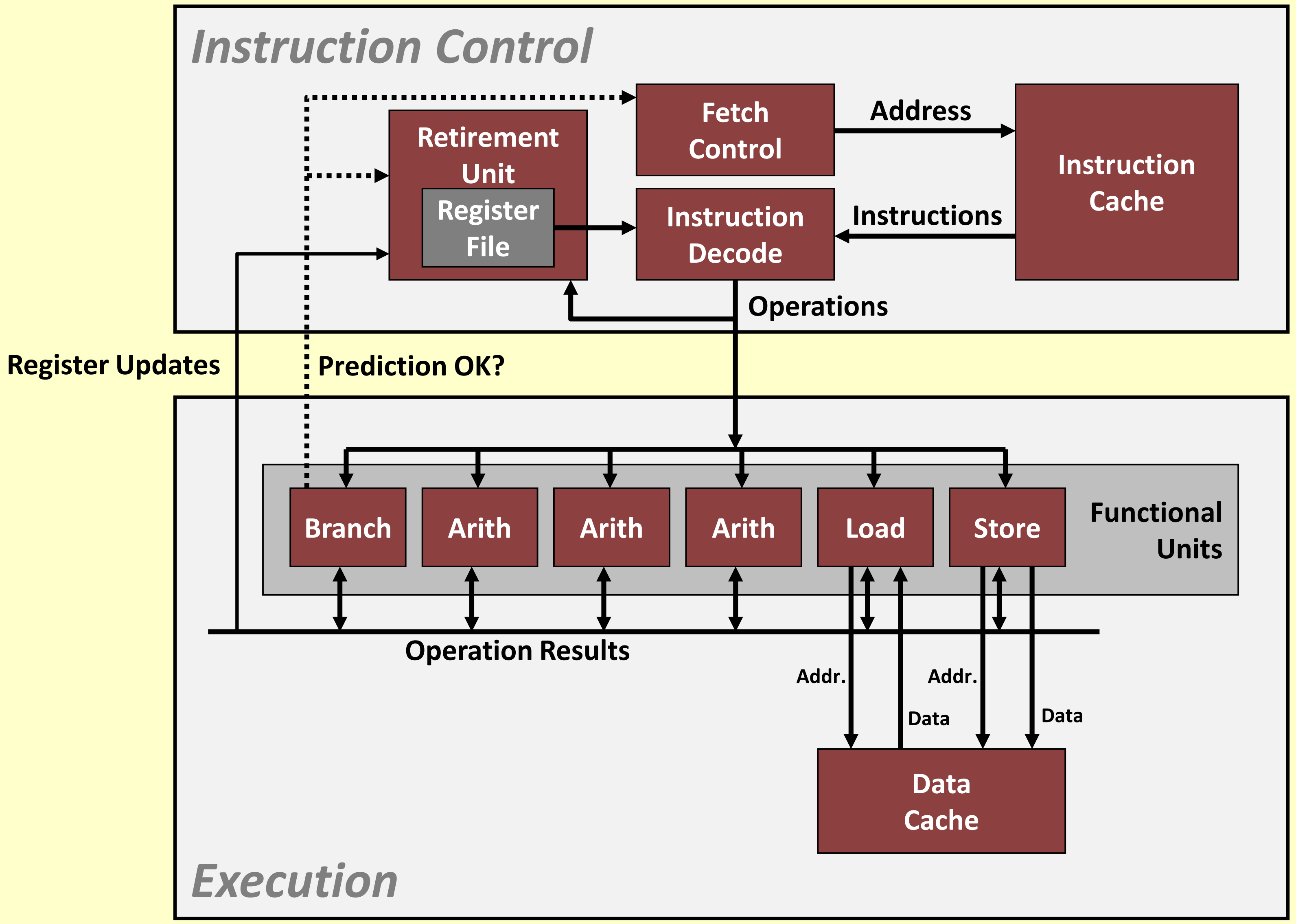
Explanation:



Such problems are a BIG deal

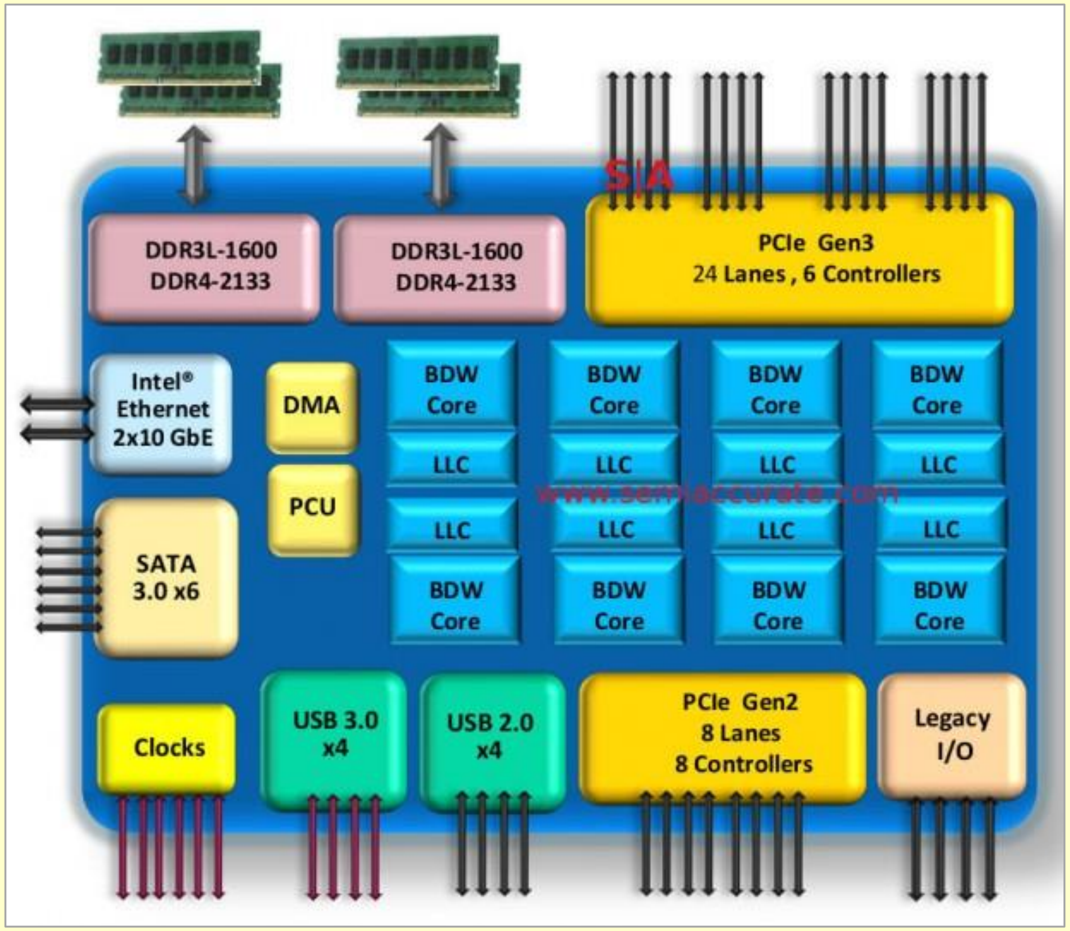
- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs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

Modern CPU Design



2015 State of the Art

- Core i7 Broadwell 2015



- **Desktop Model**
  - 4 cores
  - Integrated graphics
  - 3.3-3.8 GHz
  - 65W
- **Server Model**
  - 8 cores
  - Integrated I/O
  - 2-2.6 GHz
  - 45W

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

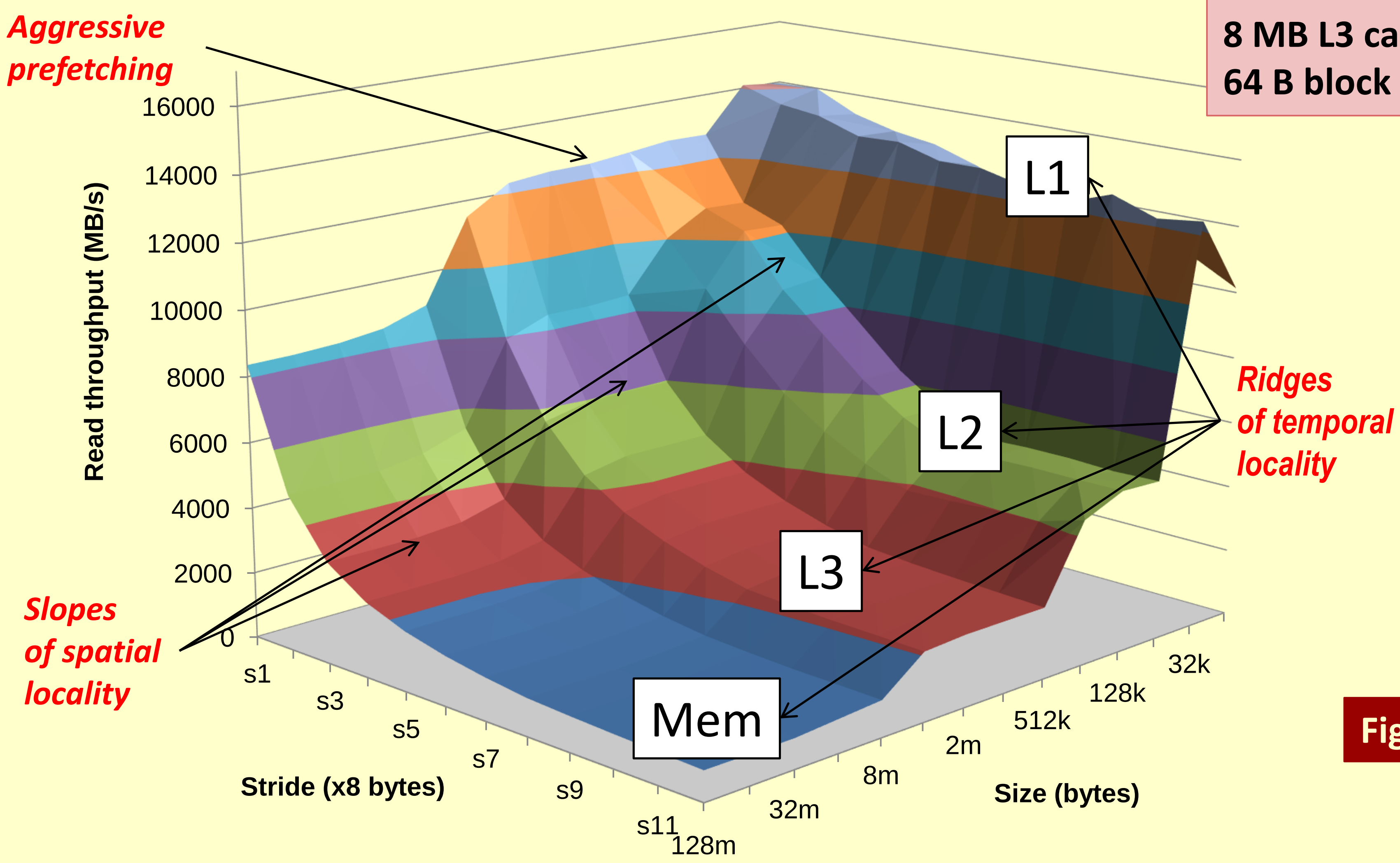


Fig. §6.41

Much more to computers …

- … than you ever expected
- Can make an entire career out of them …
- … or simply buy them and use them!

Thank you for your interest and attention

Questions?