Computer System

Cheng JIN

jc@fudan.edu.cn

People

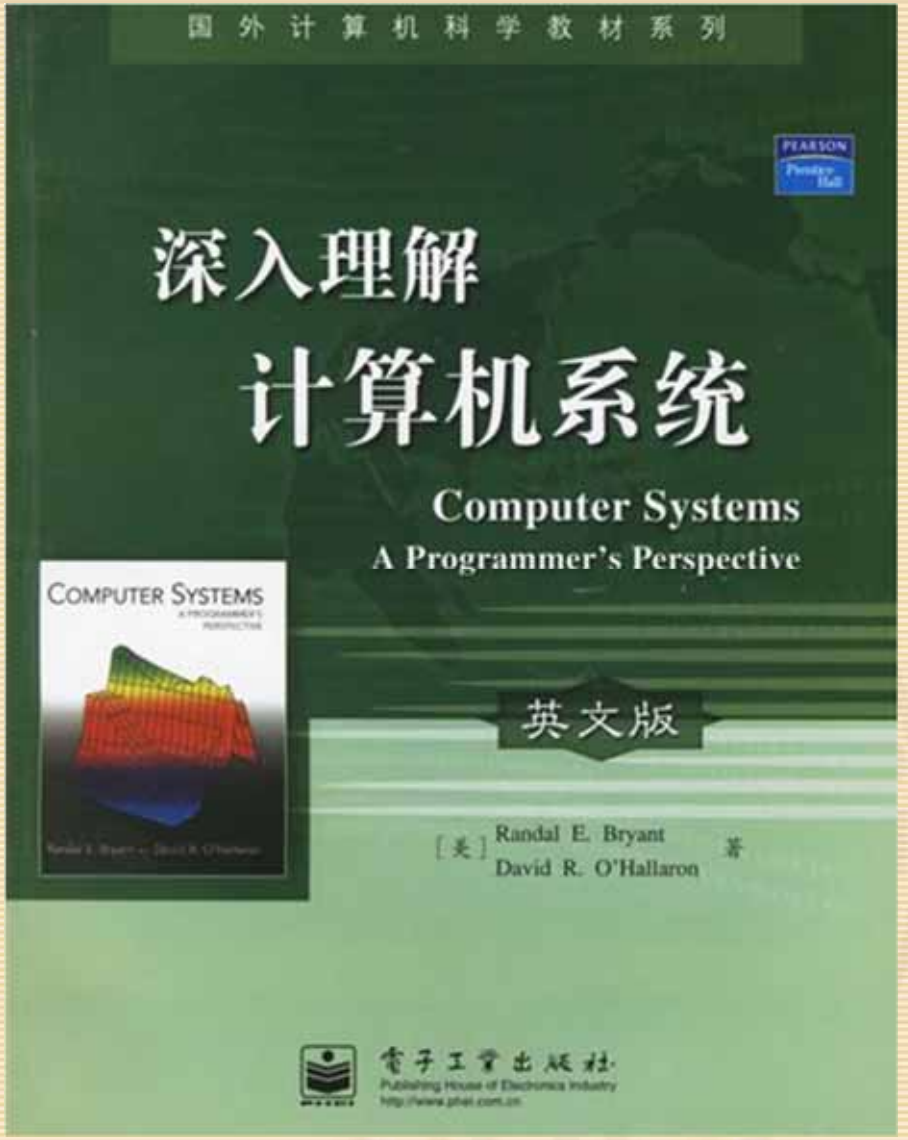
- Cheng JIN (金城)
- Email: jc@fudan.edu.cn
- Office: 306-4, Computer Building

- TA
- 俞晨光 12210240073@fudan.edu.cn
- 孙昌 12210240067@fudan.edu.cn

Textbook

- Computer Systems – A Programmer's Perspective
  - *by*
  - Randal E. Bryant & David R. O'Hallaron

- 深入理解计算机系统
  - 电子工业出版社



Grading

- Exam(s)
  - 1~2
- Lab(s)
  - Due 23:59:59 of pre-specified date
  - Losing 1/5 of the points each late day
- Homework

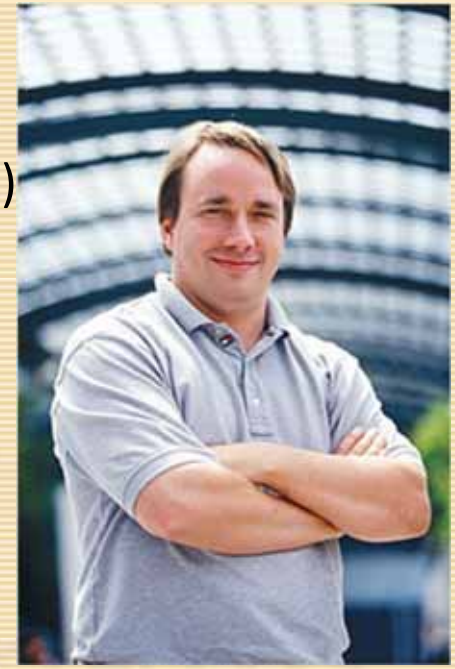
Rules

- Attendance
- Quiet vs. Talkative
- Portrait Photo
  - 240 (width) x 320 (height)
  - 学号_姓名.jpg

Why this course?

- *Page 2*
- You are going to learn:
- *How to avoid strange numerical errors*
- *How to optimize your C code*
- *How the compiler implements procedure calls*
- *How to recognize and avoid nasty errors*
- *How to write your own dynamic storage allocation package*
- *How to…*

Why this course?



From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

Newsgroups: comp.os.minix

Subject: What would you like to see most in minix?

Summary: small poll for my new operating system

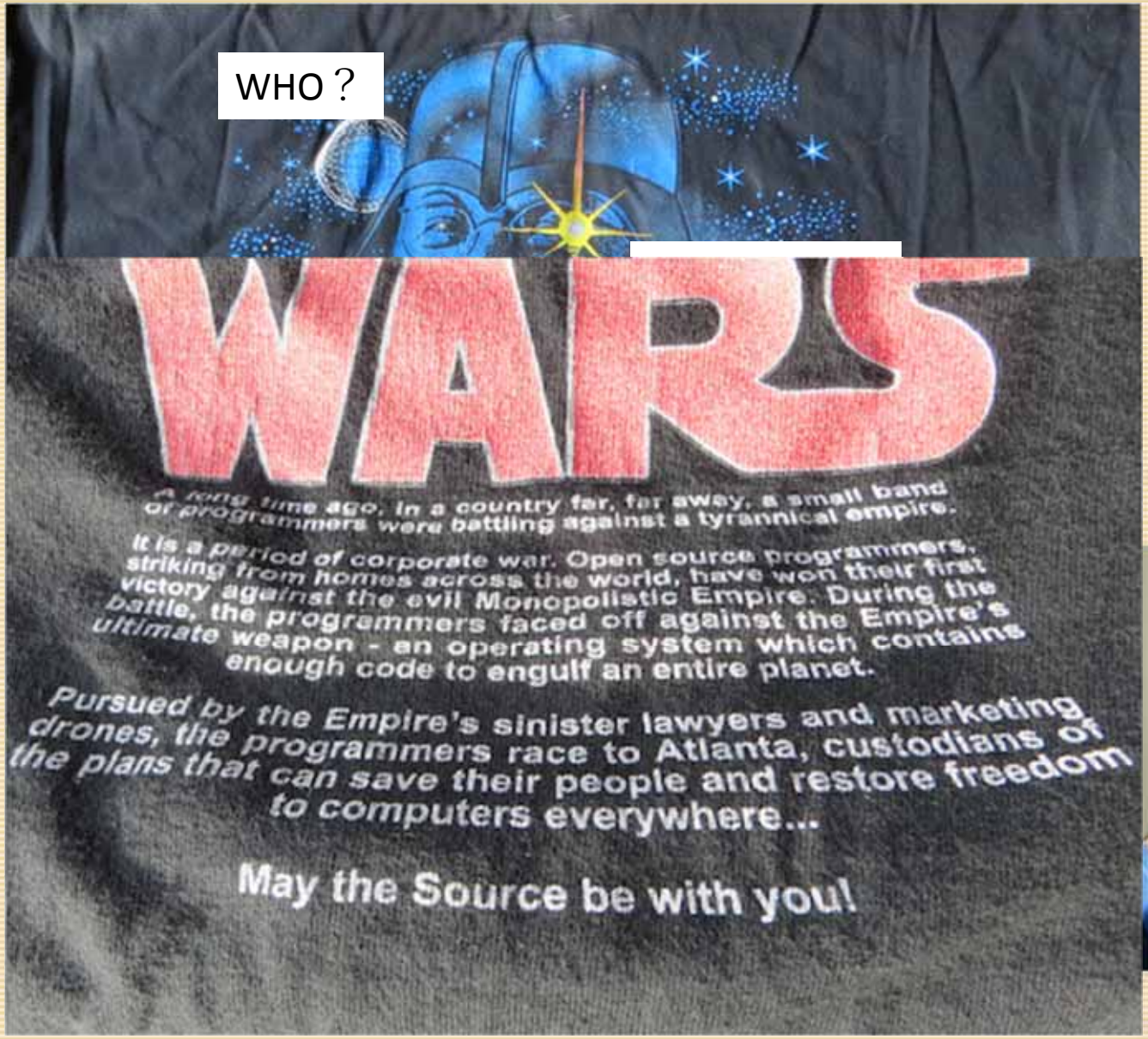
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix –

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

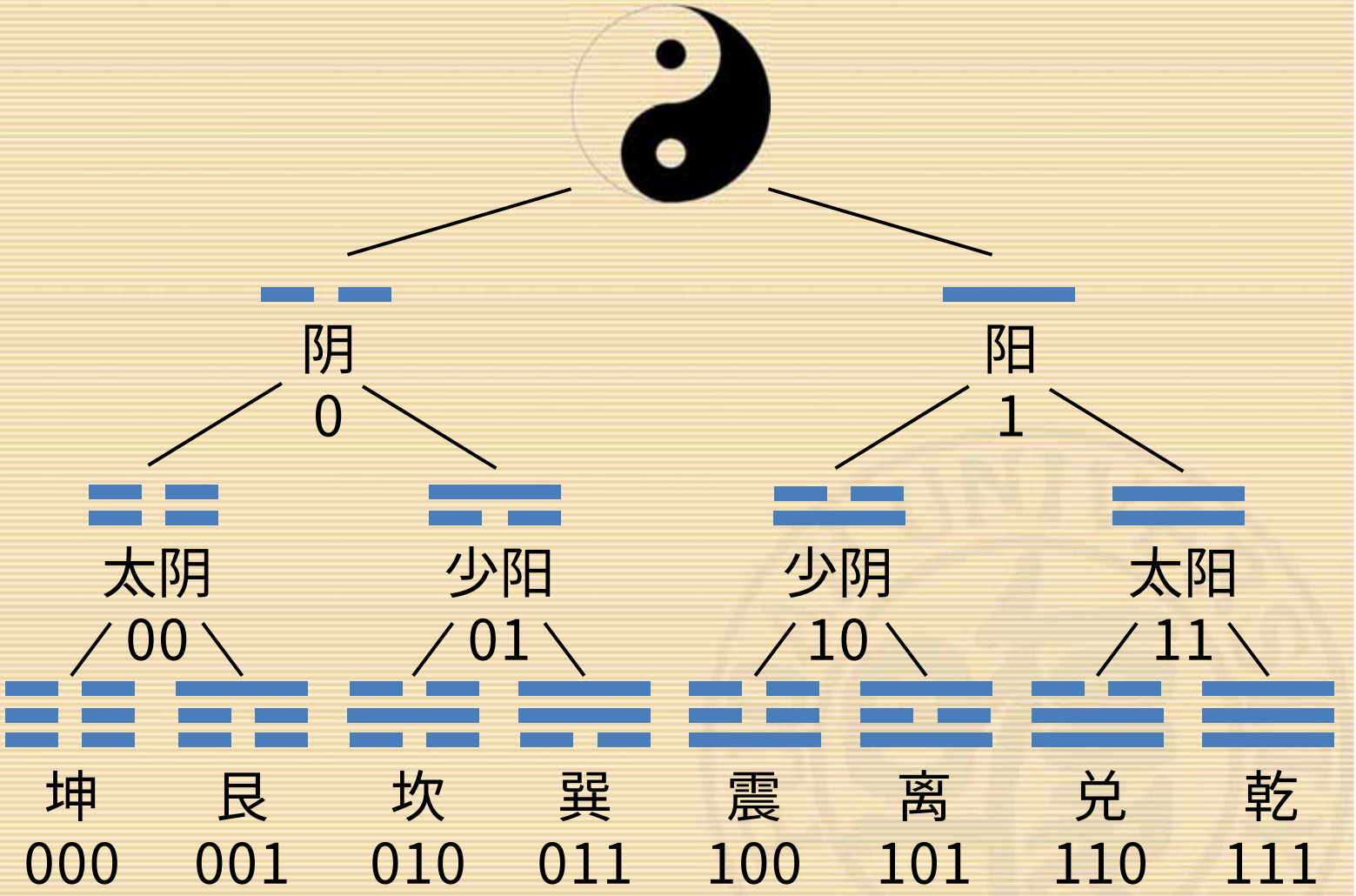
Why this course?



jc@fudan.edu.cn

Bit World

- 1
- 2
- 4
- 8



Hello World

```
#include <stdio.h>


int main()
{
        printf("hello, world\n");
}
```

Hello World

- # i n c l u d e <sp> < s t d i o .
- 35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46

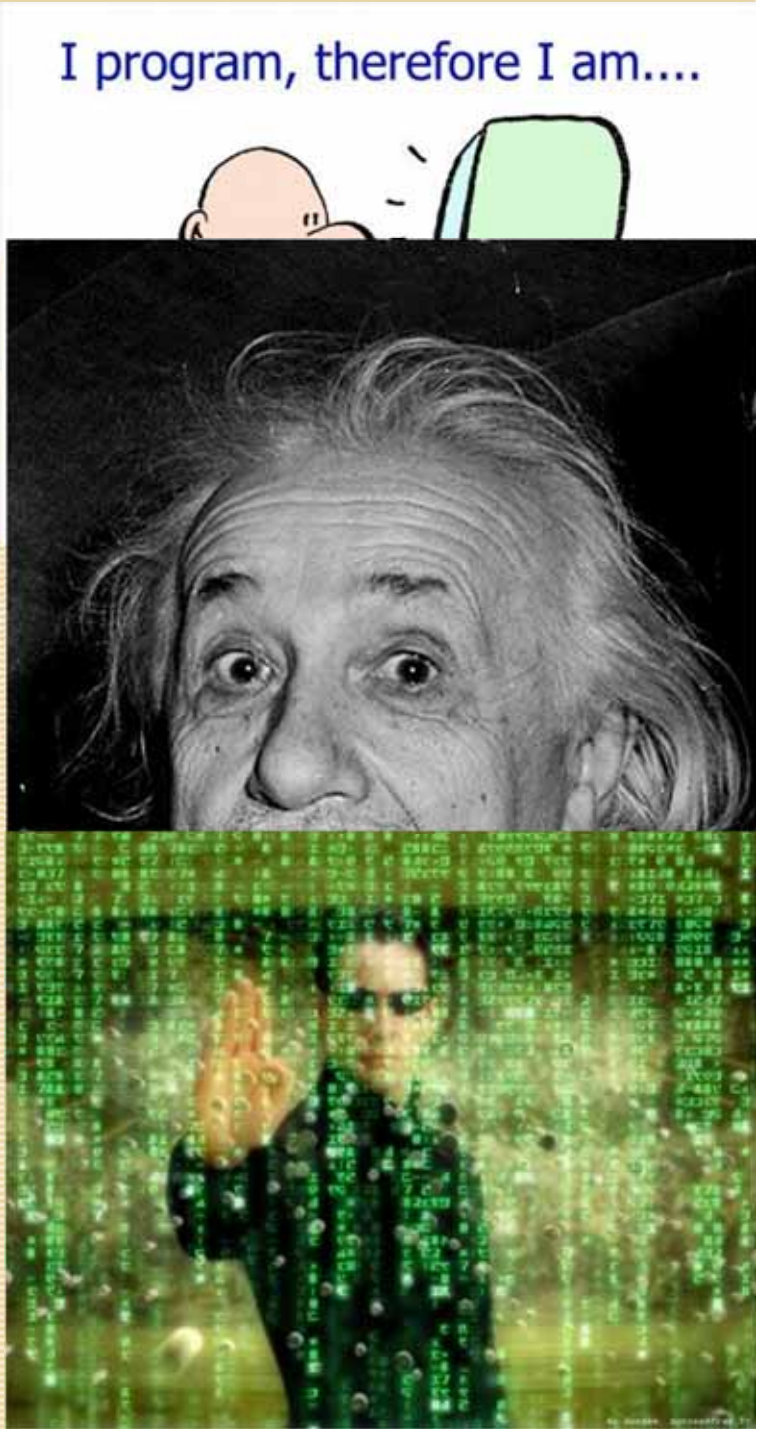
- h > \n \n i n t <sp> m a i n ( ) \n {
- 104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123

- \n <sp> <sp> <sp> <sp> p r i n t f (“ h e
- 10 32 32 32 32 112 114 105 110 116 102 40 34 104 101

- l l o , <sp> w o r l d \ n ") ; \n }
- 108 108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125

Program Life

- Source *Text*
- Modified Source *Text*
- Assembly *Text*
- Relocatable *Binary*
- Executable *Binary*



jc@fudan.edu.cn

How are they created?

- I/O system
- File

Where are they stored?

- Register ~KB
- Cache ~MB
  - On-chip, L1
  - Off-chip, L2
- Memory ~GB
- Hard Disk ~TB
- Remote/Network Disk ~PB
- Virtual Memory

Who are using them?

- Process
- Thread

Chapter 2

- Interesting problems
  - 200*300*400*500 = -884,901,888
  - 1+(1e20-1e20)
  - (1+1e20)-1e20

Number

- 20140224_{10}
- $10011001101010000011000000_2$
- 0001 0011 0011 0101 0000 1100 0000_2
- 1 3 3 5 0 C 0
- $13350C0_{16}$ 0x13350C0

10 vs 2 vs 16

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

Word Size

- Numbers are stored in addressed “memory”
- Address is also a number
- Word size, indicating the normal size of:
  - An Integer
  - A Pointer
  - An Address

Word Size

- For machine with *n-bit* word size
  - Virtual address can range from 0 to 2^n-1
- n = 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- n = 64 bits (8 bytes)
  - Potential address ≈ 1.8 X 10^{19} bytes

Data Size (P33)

C Declaration	Typical 32-bit	Compaq Alpha
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

Byte Ordering

- 0x1234567

- Big Endian
  - 01 23 45 67
  - Sun, Mac

- Little Endian
  - 67 45 23 01
  - Alpha, Intel PC

Byte Ordering (P37)

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf(“ %.2x", start[i]);
    printf("\n");
}
```

Issue caused by Byte Ordering

- Data Reading
  - Normal
  - Casting
    - Reference an object according to a different data type from which it was created
    - Most application programming ☹
    - System-level programming ☺

Example (P39)

Machine	Value	Type	Bytes (hex)
Linux	12,345	int	39 30 00 00
NT	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Alpha	12,345	int	39 30 00 00

Boolean Algebra

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

~A = 1 when A=0

~	
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

jc@fudan.edu.cn

Relations Between Operations

- De Morgan's Laws
  - Express & in terms of | and ~

    A & B = ~(~A | ~B)

    - A and B are true if and only if neither A nor B is false

  - Express | in terms of & and ~

    A | B = ~(~A & ~B)

    - A or B are true if and only if A and B are not both false

Augustus De Morgan
(27 June 1806 – 18 March 1871)

Relations Between Operations

- Exclusive-Or using Inclusive Or

  A ^ B = (~A & B) | (A & ~B)

  – Exactly one of A and B is true

  A ^ B = (A | B) & ~(A & B)

  – Either A is true, or B is true, but not both

Expand to Bit Vectors

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  01000001      01111101      00111100      10101010
```

Bit Vector Application

- Representation of Sets
  - {0, 1, 2, 3, 4, 5, 6, 7}
  - Two sub-sets
    - { 0, 3, 5, 6 } 01101001
    - { 0, 2, 4, 6 } 01010101
  - & 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 }

Two-Level Operations

- Bit Level
  - &, |, ~, ^
- Data Level
  - &&, ||, !

Xor (P47)

```
void funny(int *x, int *y)
{
   *x = *x ^ *y;     /* #1 */
   *y = *x ^ *y;     /* #2 */
   *x = *x ^ *y;     /* #3 */
}
```

Step	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A^(B^B) = A^0 = A
3	(A^B)^A = (B^A)^A = B^(A^A) = B^0 = B	A
End	B	A

Shift

- Left Shift <<
- Right Shift >>
  - Logical 0
  - Arithmetic most significant bit

Integer Representation

jc@fudan.edu.cn

How Do Decimals Work?

jc@fudan.edu.cn

How Will Binaries Work?

- Non-Negative Integers

How Will Binaries Work?

- Negative Integers

Two's Complement

- Binary
  - Bit vector $[x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0]$

- Using 2's complement to represent integer

Unsigned

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

Two's Complement

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

Sign Bit

From 2's Complement to Binary

- If nonnegative
  - Nothing changes
- If negative

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

Two's Complement Encoding

- Binary/Hexadecimal Representation for 12345

  Binary: 0011 0000 0011 1001

  Hex: 3 0 3 9

- Binary/Hexadecimal Representation for –12345

  Binary: 1100 1111 1100 0111

  Hex: C F C 7

More about Numbers

Weight		12,345		-12,345		53,191
	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4096	0	0	0	0
8,192	1	8192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32,768	0	0	1	-32,768	1	32,768
Total		12,345		-12,345		53,191

Numeric Range

- Unsigned Values
  - Umin=0
  - Umax=2^w-1
- Two's Complement Values
  - Tmin = -2^{w-1}
  - Tmax = 2^{w-1}-1

Interesting Numbers

Quantity	Word Size ω			
	8	16	32	64
$UMax_\omega$	0xFF 255	0xFF FF 65,535	0xFF FF FF FF 4,294,967,295	0xFF FF FF FF FF FF FF FF 18,446,744,073,709,551,615
$TMax_\omega$	0x7F 127	0x7F FF 32,767	0x7F FF FF FF 2,147,483,647	0x7F FF FF FF FF FF FF FF 9,223,372,036,854,775,807
$TMin_\omega$	0x80 -128	0x80 00 -32,768	0x80 00 00 00 - 2,147,483,648	0x80 00 00 00 00 00 00 00 -9,223,372,036,854,775,808
-1	0xFF	0xFF FF	0xFF FF FF FF	0xFF FF FF FF FF FF FF FF
0	0x00	0x00 00	0x00 00 00 00	0x00 00 00 00 00 00 00 00

Binary, Unsigned and 2's Complement

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

Unsigned & Signed Numeric Values

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

Unsigned & Signed Numeric Values

- $\Rightarrow$ 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

Alternative Representations

- One's Complement:
  - The most significant bit has weight $-(2^{w-1}-1)$
- Sign-Magnitude
  - The most significant bit is a sign bit
- 0?

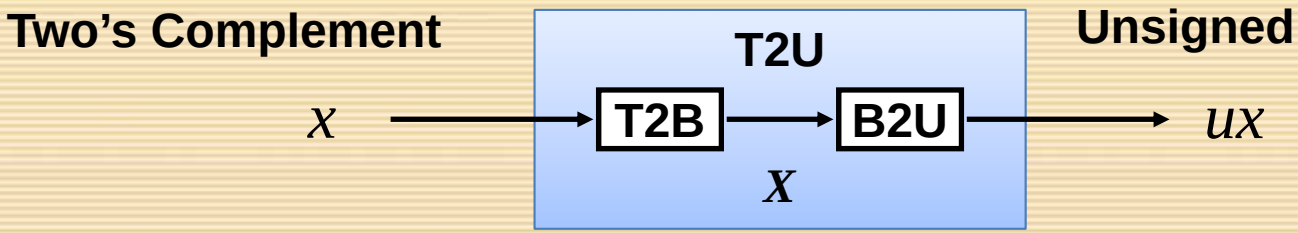
Casting Signed to Unsigned

- C Allows Conversions from Signed to Unsigned

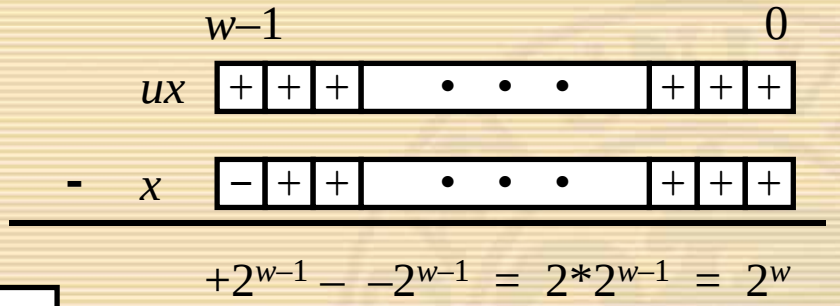
```
short int          x =  12345;
unsigned short int ux = (unsigned short) x;
short int          y  = -12345;
unsigned short int uy = (unsigned short) y;
```

- Resulting Value
  - No change in bit representation
  - Nonnegative values unchanged
    - *ux* = 12345
  - Negative values change into (large) positive values
    - *uy* = 53191

Relation Between 2's Comp. & Unsigned

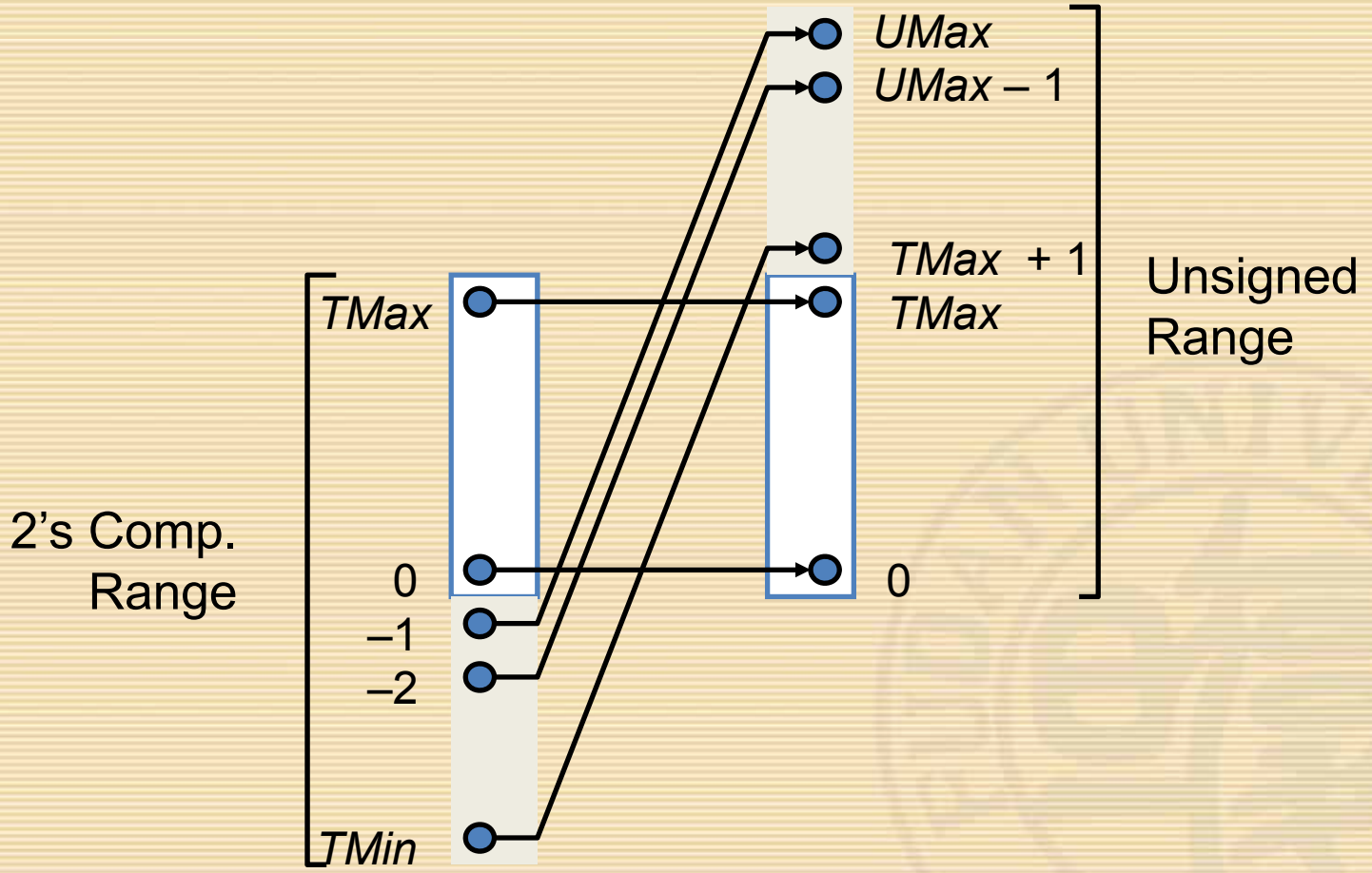


Maintain Same Bit Pattern



$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

Conversion


UMax
UMax – 1
TMax + 1
TMax
0
Unsigned
Range
2's Comp.
Range
TMax
0
–1
–2
TMin

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    - 0U, 4,294,967,295U

Signed vs. Unsigned in C

- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U
    - int tx, ty;
    - unsigned ux, uy;
    - tx = (int) ux;
    - uy = (unsigned) ty;

Signed vs. Unsigned in C

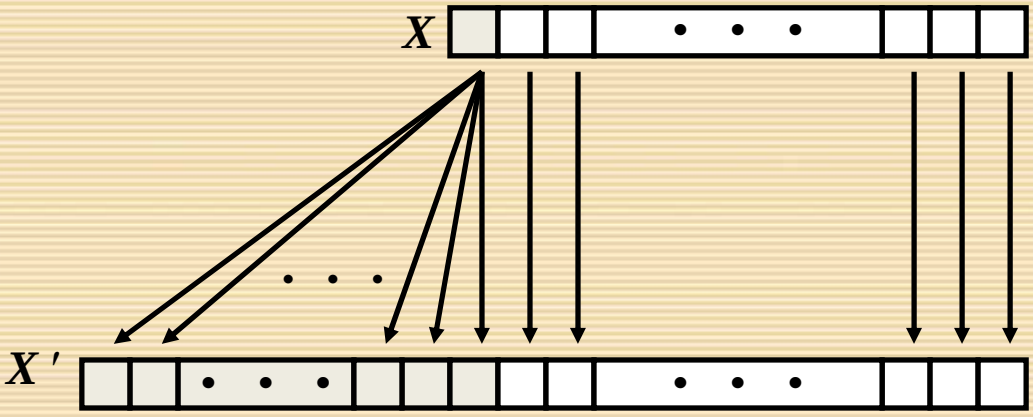
- Casting
  - Implicit casting also occurs via assignments and procedure calls
    - int tx, ty;
    - unsigned ux, uy;
    - tx = ux; /* Cast to signed */
    - uy = ty; /* Cast to unsigned */

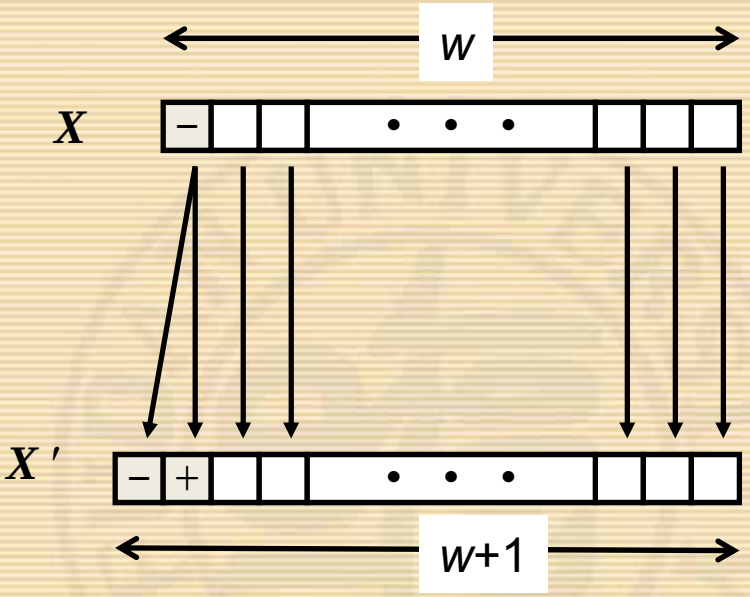
Casting Convention

Constant1	Constant2	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned

Expanding the Bit Representation

- Unsigned: Zero extension
  - Add leading 0s to the representation
- Signed: Sign extension
  - $[x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0]$





Solution 1

Sign Extension Example

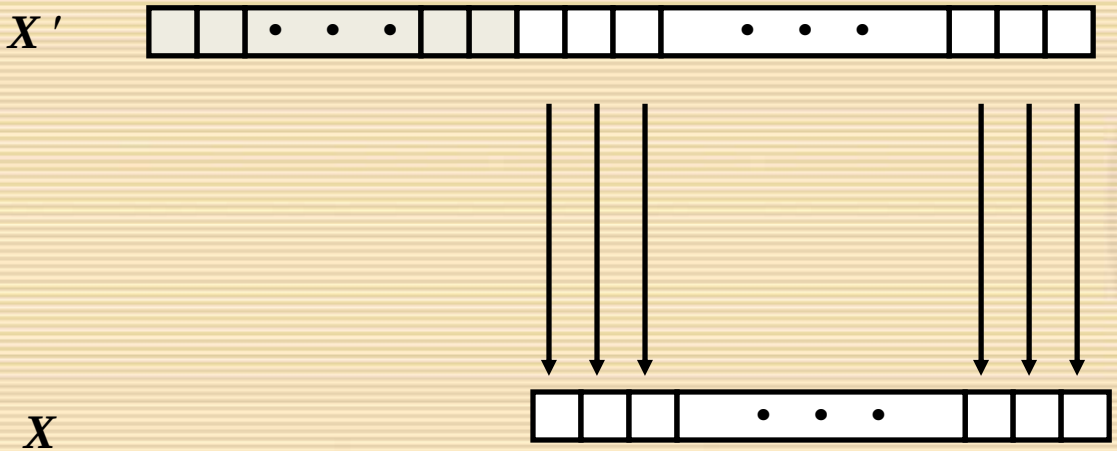
```
short int x =  12345;
int       ix = (int) x;
short int y = -12345;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

Truncating Numbers

```
int       x  =  53191;
short int sx = -12345;
int       y  = -12345;
```

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111
y	-12345	FF FF CF C7	11111111 11111111 11001111 11000111



Truncating Numbers

- Unsigned Truncating **P64 Eq. (2.7)**

$$B2U_w([x_{w-1}, x_{w-2}, \cdots, x_0]) \bmod 2^k = B2U_k([x_{k-1}, x_{k-2}, \cdots x_0])$$

- Signed Truncating **P64 Eq. (2.8)**

$$B2T_k([x_{k-1}, x_{k-2}, \cdots x_0]) = U2T_k(B2U_w([x_{w-1}, x_{w-2}, \cdots, x_0]) \bmod 2^k)$$

Advice on Signed vs. Unsigned

Nonintuitive Features

```
unsigned length ;
int i ;
…..
for ( i = 0; i <= length – 1; i++)
      result += a[i] ;
```

Integer Operations

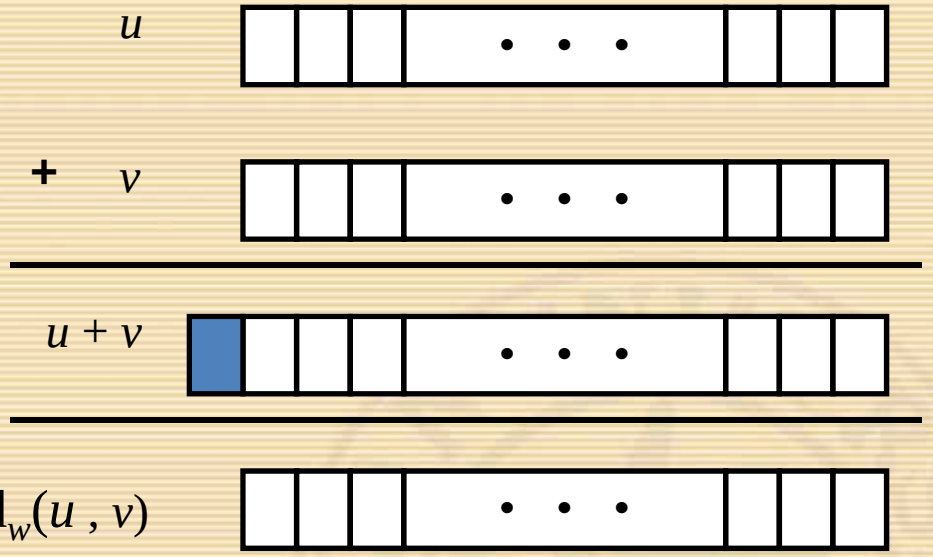
jc@fudan.edu.cn

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



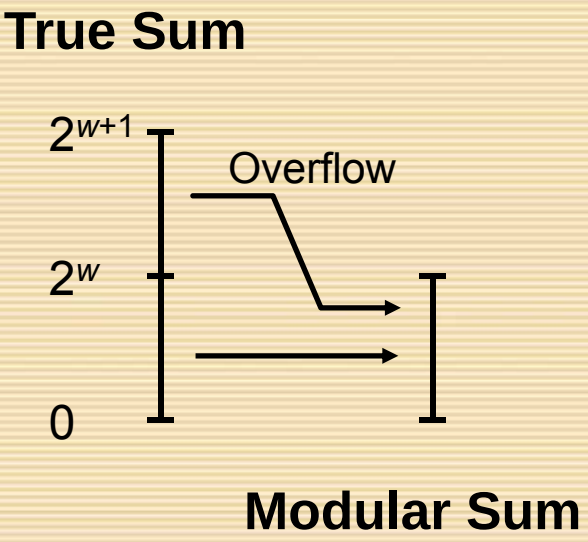
Unsigned Addition

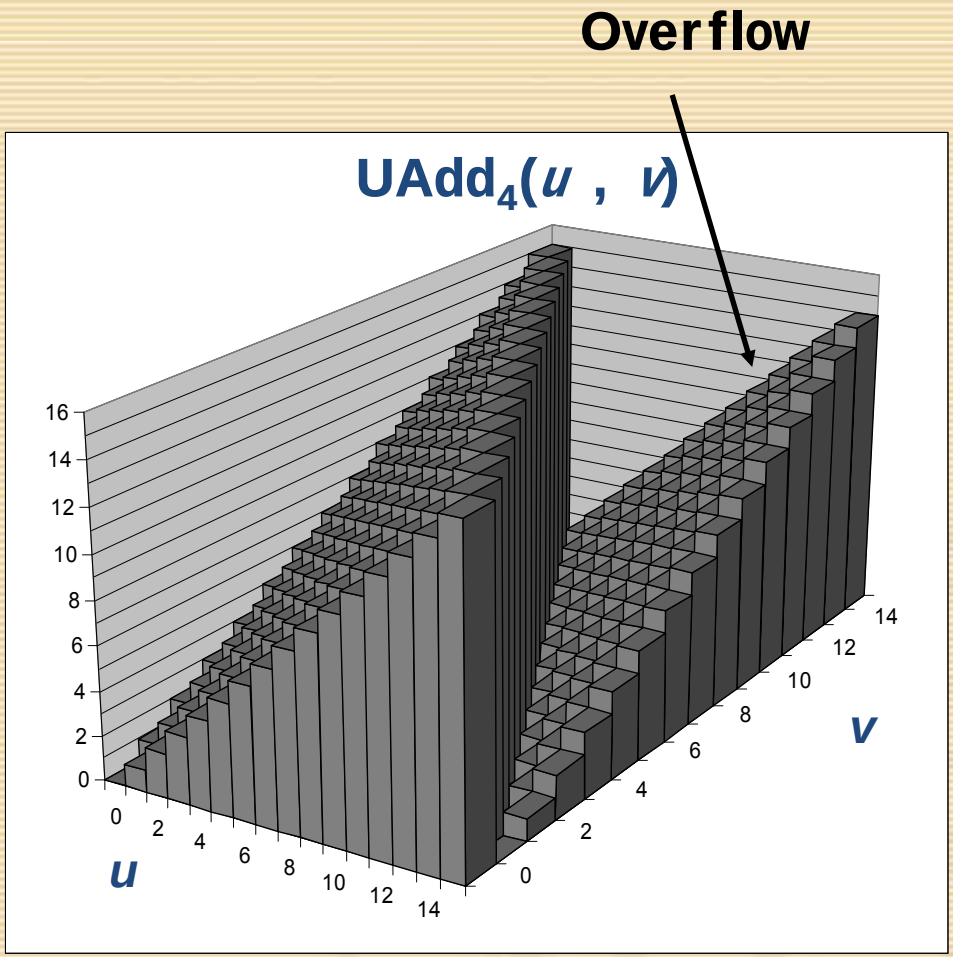
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  - $s = \text{UAdd}w(u , v) = (u + v) \mod 2^w$

$$UAdd_w(u,v) = \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Visualizing Unsigned Addition

- Wraps Around
  - If true sum ≥ 2^w
  - At most once





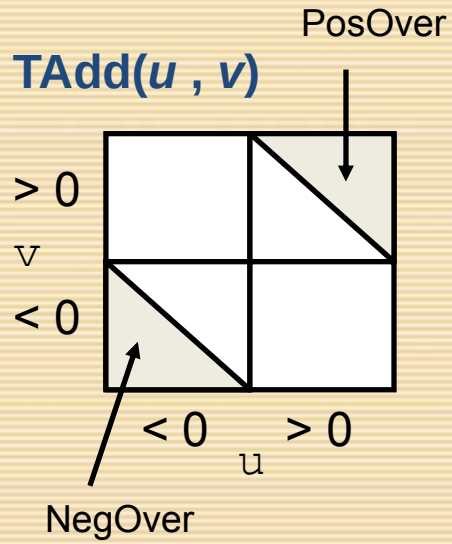
Signed Addition

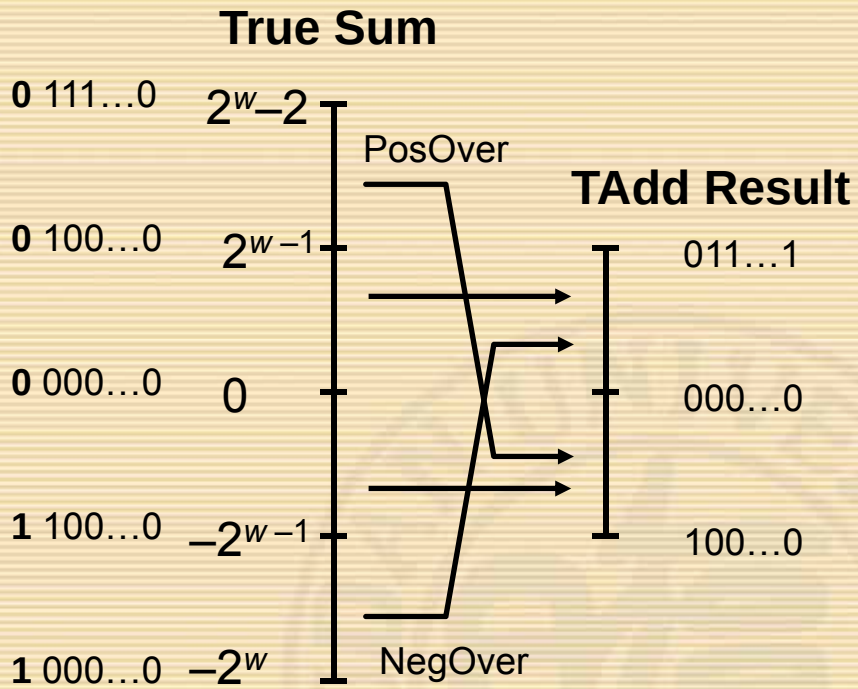
- Functionality
  - True sum requires ***w*+1** bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer

$$Tadd(u,v)=\begin{cases} u+v-2^{w} & TMax_{w} < u+v \ (PosOver) \\ u+v & TMin_{w} \le u+v \le TMax_{w} \\ u+v+2^{w} & u+v < TMin_{w} \ (NegOver) \end{cases}$$

PosOver：Positive Overflow
NegOver：Negative Overflow

Signed Addition

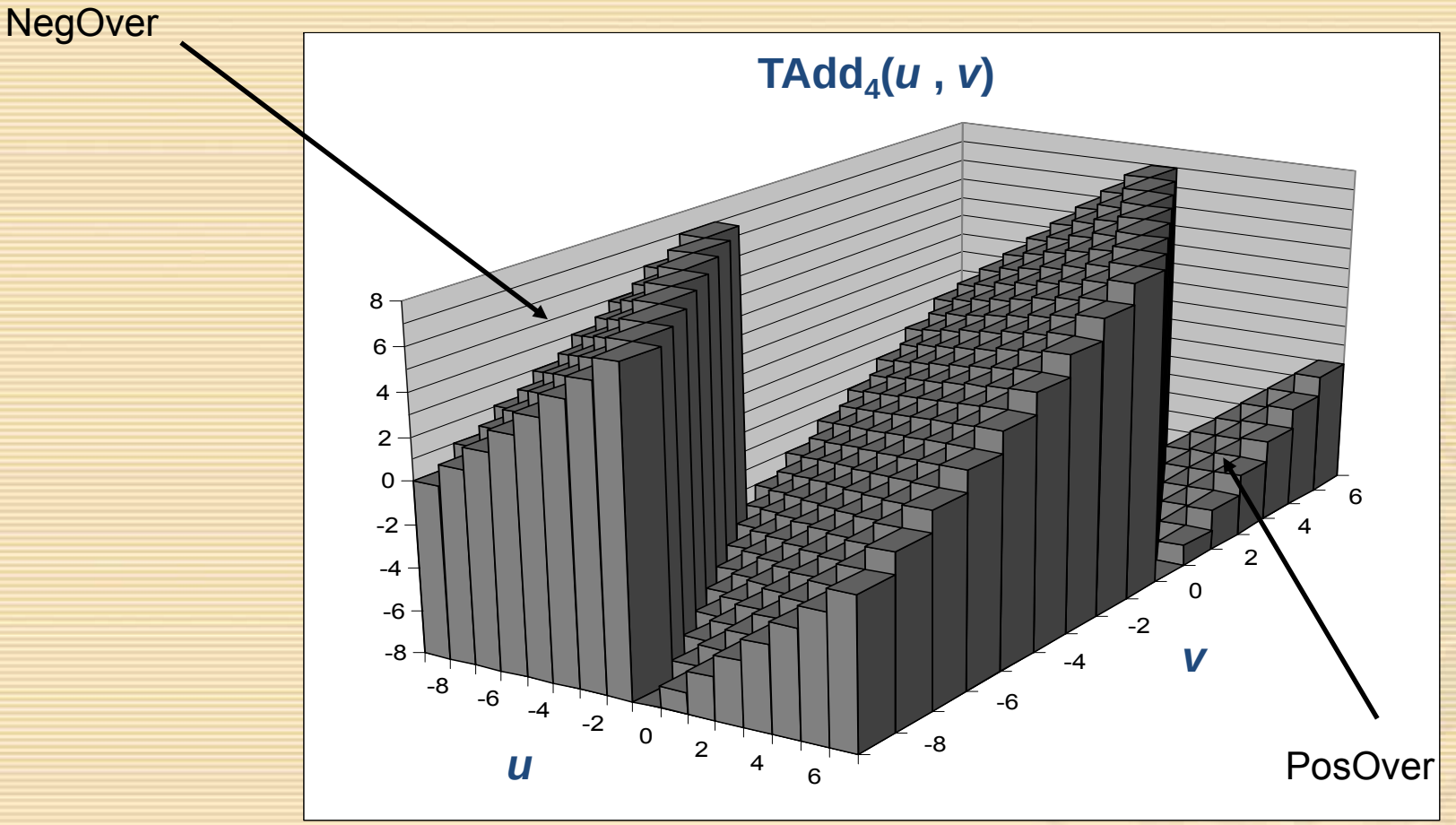




Visualizing 2's Comp. Addition

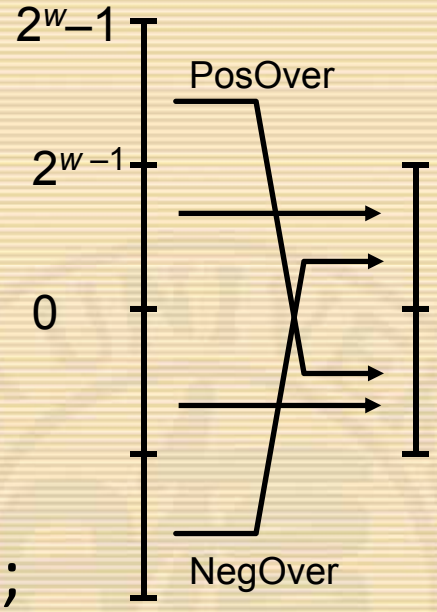
- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
  - If sum $< -2^{w-1}$
    - Becomes positive

Visualizing 2's Comp. Addition



Detecting Tadd Overflow

- Task
  - Given $s = \text{TAdd}_w(u , v)$
  - Determine if $s == \text{Add}_w(u , v)$
- Claim
  - Overflow iff either:
    - $u, v < 0, s \geq 0$ (NegOver)
    - $u, v \geq 0, s < 0$ (PosOver)
  - ovf = (u<0 == v<0) && (u<0 != s<0);



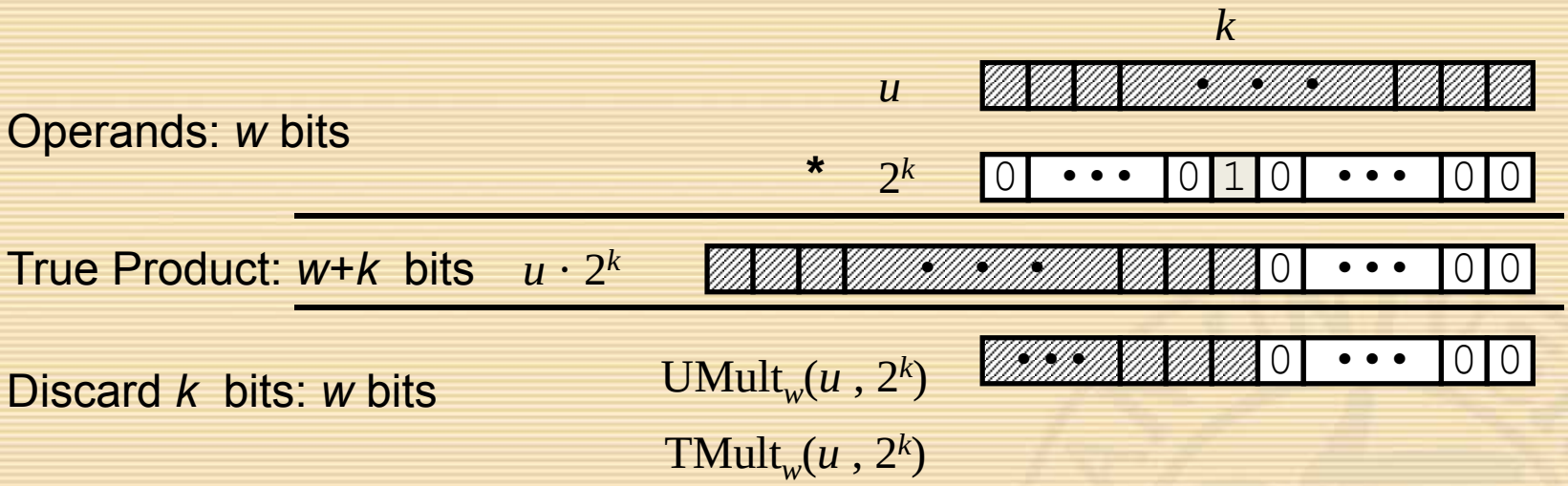
Negation

$$-_w^t x = \begin{cases} -2^{w-1} & x = -2^{w-1} \\ -x & x > -2^{w-1} \end{cases}$$

Multiplication

- Computing Exact Product of *w*-bit numbers *x*, *y*
  - Either signed or unsigned
- Ranges
  - Unsigned: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to 2*w* bits
  - Two's complement min: $x * y \ge -2^{w-1}*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to 2*w*–1 bits
  - Two's complement max: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
    - Up to *2w-1* bits, but only for $TMinw^2$
- Same bit pattern for both unsigned and signed

Power-of-2 Multiply with Shift

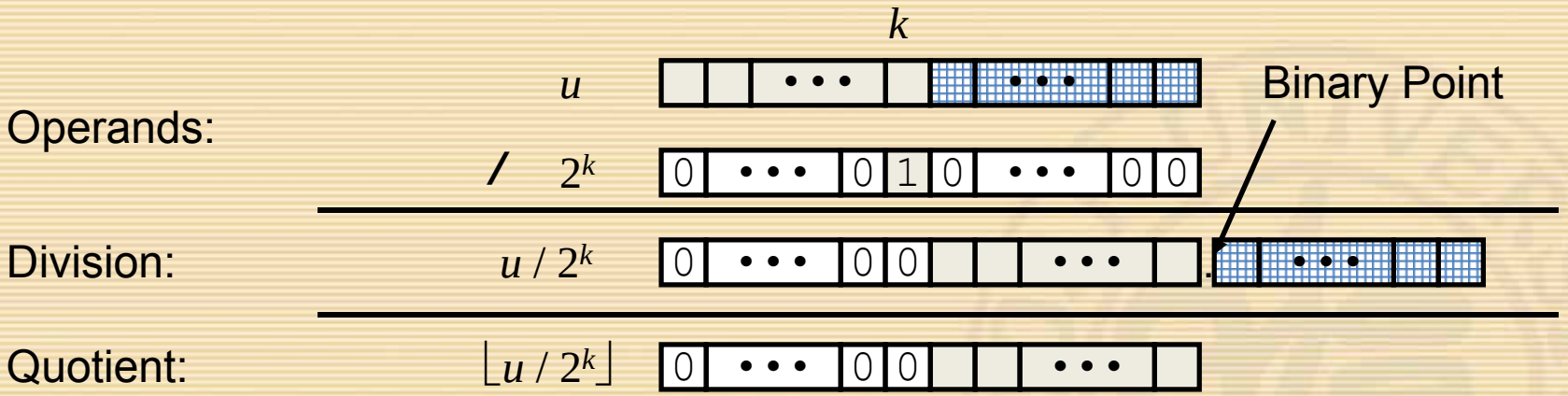


Power-of-2 Multiply with Shift

- Operation
  - u << k gives u * 2^k
  - Both signed and unsigned
- Examples
  - u << 3 == u * 8
  - (u << 5) – (u << 3) == u * 24
  - Most machines shift and add much faster than multiply
    - Compiler will generate this code automatically

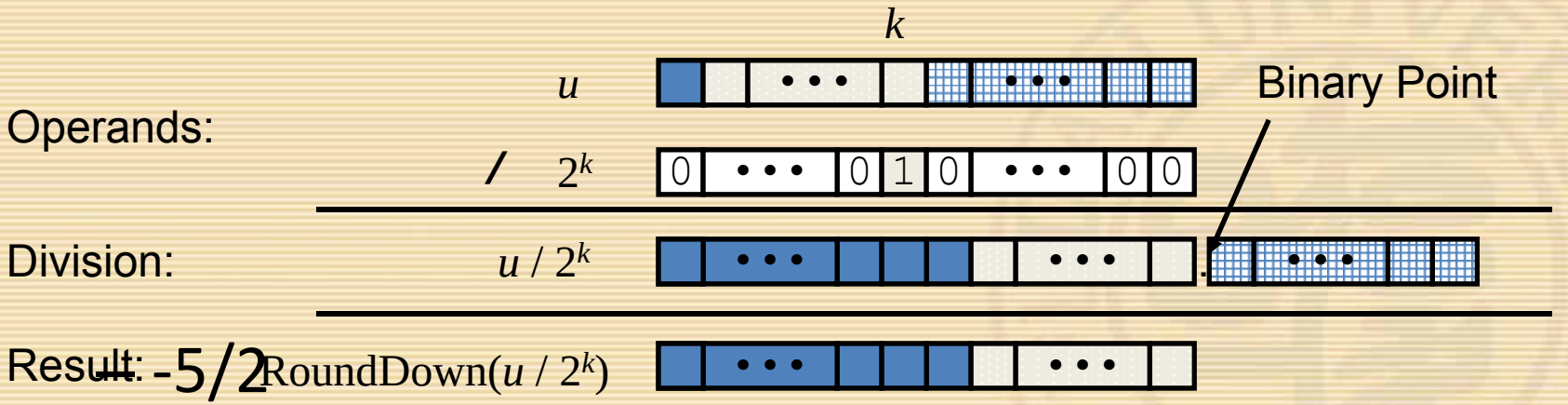
Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - u >> k gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift



2's Comp Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - u >> k gives $\lfloor u / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when u < 0



Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want $\lceil u / \mathbf{2}^k \rceil$ **(Round Toward 0)**
  - Because: $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$
  - $x = ky+b$, $0 \le b \le y-1$
  - Compute as $\lfloor (u+\mathbf{2}^k-1)/ \mathbf{2}^k \rfloor$
    - In C: **(u + (1<<k)-1) >> k**
    - Biases dividend toward 0

Floating Point

jc@fudan.edu.cn

How Do Decimals Work?

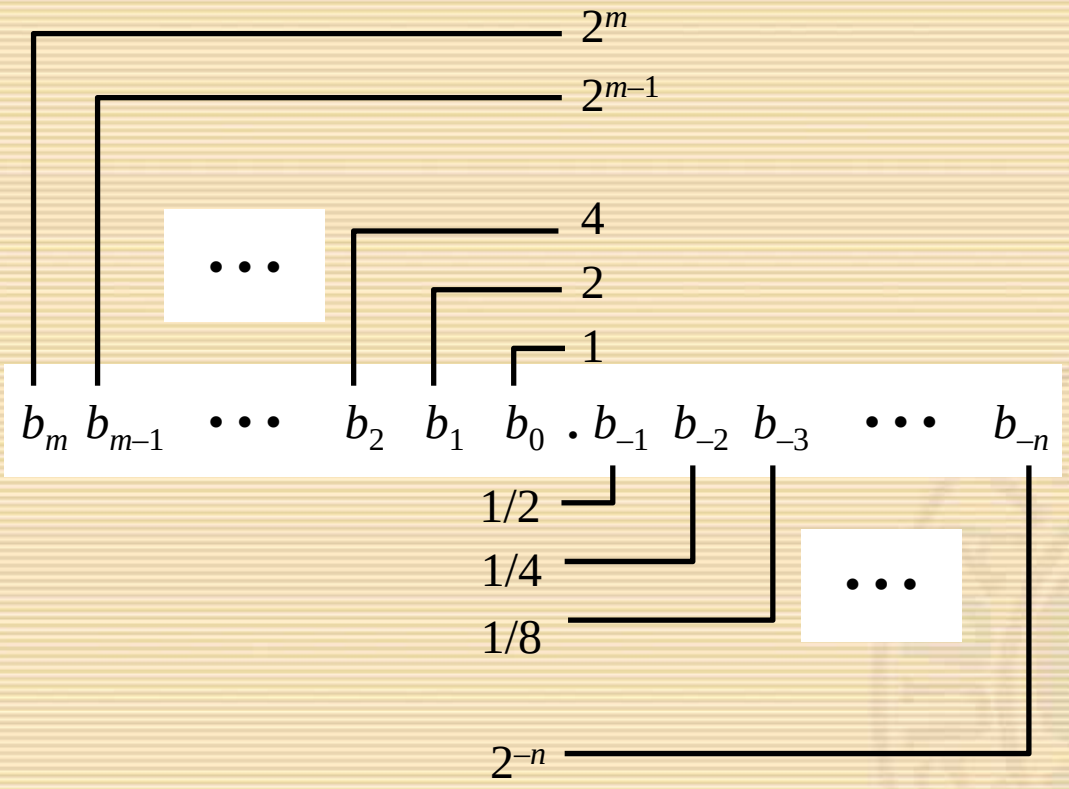
jc@fudan.edu.cn

How Will Binaries Work?

- Form V = $x \times 2^{y}$
- Very useful when $|V|$ >> 0 or $|V|$ <<1

jc@fudan.edu.cn

Fractional Binary Numbers



Fraction Binary Number Examples

Value	Binary Fraction
0.3	?

- Observations:
  - The form 0.11111…11 represent numbers just below 1.0 which is noted as $1.0-\varepsilon$
  - Binary Fractions can only exactly represent $x/2^k$
  - Others have repeated bit patterns

IEEE Floating-Point Representation

- Numeric form
  - $V=(-1)^s \times M \times 2^E$
    - Sign bit *s* determines whether number is negative or positive
    - Significand *M* normally a fractional value in range [1.0,2.0)
    - Exponent *E* weights value by power of two

IEEE Floating-Point Representation

- Encoding
  - 

  - s is sign bit
  - **exp** field encodes *E*
  - **frac** field encodes *M*
- Sizes
  - Single precision (32 bits): 8 exp bits, 23 frac bits
  - Double precision (64 bits): 11 exp bits, 52 frac bits

Normalized Values

- Condition
  - **exp ≠ 000…0 and exp ≠ 111…1**
- Exponent coded as *biased* value
  - *E = Exp – Bias*
    - *Exp* : unsigned value denoted by **exp**
    - *Bias* : Bias value
      - Single precision: **127** (*Exp*: 1…254, *E* : -126…127)
      - Double precision: **1023** (*Exp*: 1…2046, *E* : -1022 …1023)
      - In general: *Bias* = $2^{k-1} - 1$, where *k* is the number of exponent bits

Normalized Values

- Significand coded with implied leading 1
  - m = `1.xxx…x`$_2$
    - `xxx…x`: bits of `frac`
    - Minimum when `000…0` (M = 1.0)
    - Maximum when `111…1` (M = 2.0 – ε)
    - Get extra leading bit for “free”

Normalized Encoding Examples

- Value: 12345 (Hex: 0x3039)
- Binary bits: 11000000111001
- Fraction representation: $1.1000000111001*2^{13}$
- M: 10000001110010000000000
- E: 10001100 (140)
- Binary Encoding
  - 0100 0110 0100 0000 1110 0100 0000 0000
  - 4 6 4 0 E 4 0 0

Denormalized Values

- Condition
  - **exp = `000…0`**
- Values
  - Exponent Value: $E = 1 - Bias$
  - Significant Value m = `0.xxx…x`$_2$
    - `xxx…x`: bits of `frac`

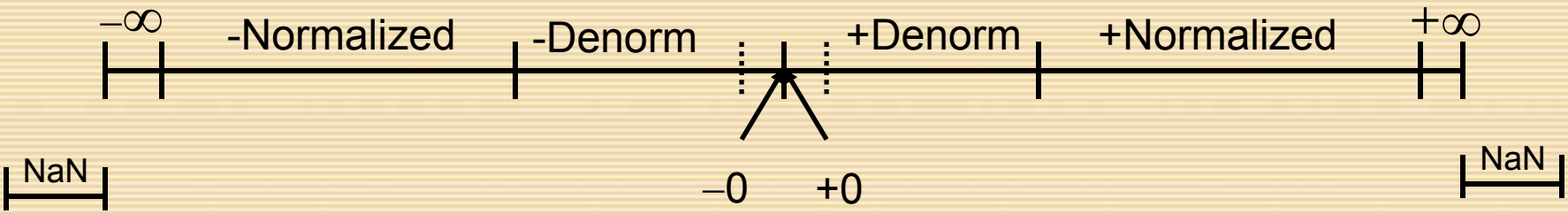
Denormalized Values

- Cases
  - **exp = 000…0, frac = 000…0**
    - Represents value 0
    - Note that have distinct values +0 and –0
  - **exp = 000…0, frac ≠ 000…0**
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - “Gradual underflow”

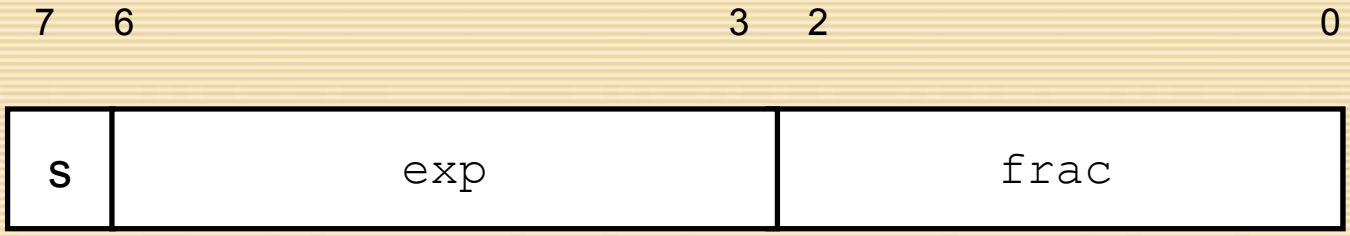
Special Values

- Condition
  - s=0, exp = 111…1, frac=000…0 +∞
  - s=1, exp = 111…1, frac=000…0 -∞
  - exp = 111…1 NaN

Summary of Real Number Encodings



8-bit Floating-Point Representations



Interesting Numbers

1 / 8 / 23 1 / 11 / 52

Description	exp	frac	Single Precision		Double Precision	
			Value	Decimal	Value	Decimal
Zero	**00…00**	00…00	0	0.0	0	0.0
Smallest denorm.	**00…00**	00…01	$2^{-23}x2^{-126}$	$1.4x10^{-45}$	$2^{-52}x2^{-1022}$	$4.9x10^{-324}$
Largest denorm.	**00…00**	11…11	$(1-\varepsilon)x2^{-126}$	$1.2x10^{-38}$	$(1-\varepsilon)x2^{-1022}$	$2.2x10^{-308}$
Smallest norm.	**00…01**	00…00	$1x2^{-126}$	$1.2x10^{-38}$	$1x2^{-1022}$	$2.2x10^{-308}$
One	**01…11**	00…00	$1x2^{0}$	1.0	$1x2^{0}$	1.0
Largest norm.	**11…10**	11…11	$(2-\varepsilon)x2^{127}$	$3.4x10^{38}$	$(2-\varepsilon)x2^{1023}$	$1.8x10^{308}$

Special Properties of Encoding

- FP Zero Same as Integer Zero
  - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider -0 = 0
  - NaNs are problematic
    - Will be greater than any other values
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Round Mode

Mode	1.40	1.60	1.50	2.50	-1.50
Round-to-even	1	2	2	2	-2
Round-toward-zero	1	1	1	2	-1
Round-down	1	1	1	2	-2
Round-up	2	2	2	3	-1

Round-to-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under- estimated

Round-to-Even

- Applying to Other Decimal Places
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

1.2349999	
1.2350001	
1.2350000	
1.2450000	

Rounding Binary Number

- "Even" when least significant bit is 0
- Half way when bits to right of rounding position = $100\ldots_2$

Value	Binary	Rounded	Action	Round Decimal
2+3/32	10.00011			
2+3/16	10.0011			
2+7/8	10.111			
2+5/8	10.101			

Floating-Point Operations

- Conceptual View
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

FP Multiplication

- Operands

  $(-1)^{s1}\ M1\ \ 2^{E1}$

  $(-1)^{s2}\ M2\ \ 2^{E2}$

- Exact Result

  $(-1)^{s}\ M\ \ 2^{E}$

  – Sign *s* : *s1 ^ s2*

  – Significand *M* : *M1 * M2*

  – Exponent *E* : *E1 + E2*

FP Multiplication

- Fixing
  - If $M \geq 2$, shift M right, increment E
  - If E is out of range, overflow
  - Round M to fit `frac` precision

FP Addition

- Operands

  $(-1)^{s1}\ M1\ \ 2^{E1}$

  $(-1)^{s2}\ M2\ \ 2^{E2}$

  – Assume $E1 > E2$

- Exact Result

  $(-1)^{s}\ M\ \ 2^{E}$

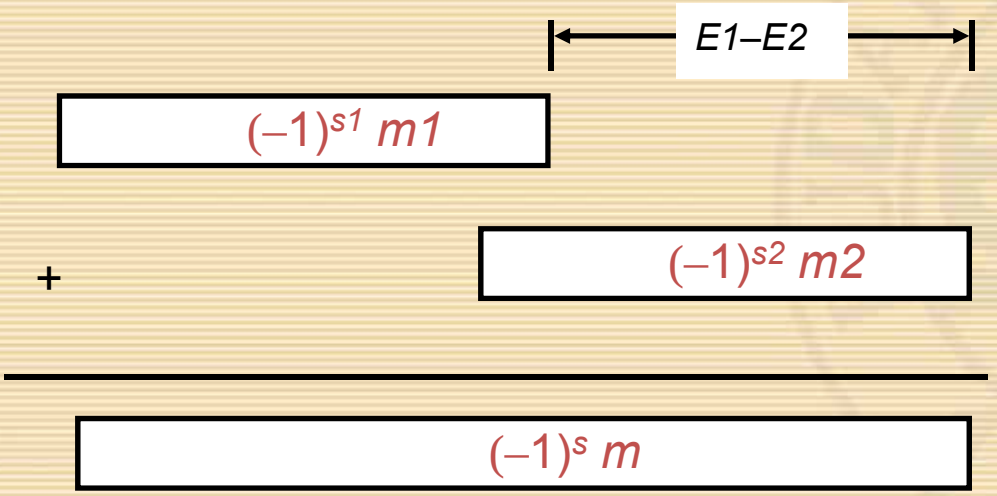
  – Sign s, significand M:

    • Result of signed align & add

  – Exponent E : $E1$

FP Addition

- Fixing
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision



Floating Point Puzzles

- int x = …;
- float f = …;
- double d = …;
- Assume neither d nor f is NAN or infinity

Floating Point in C

- x == (int)(float) x
- x == (int)(double) x
- f == (float)(double) f
- d == (float) d
- f == -(-f);
- 2/3 == 2/3.0
- d < 0.0 ⇨ ((d*2) < 0.0)
- d*d >= 0.0
- (d+f)-d == f

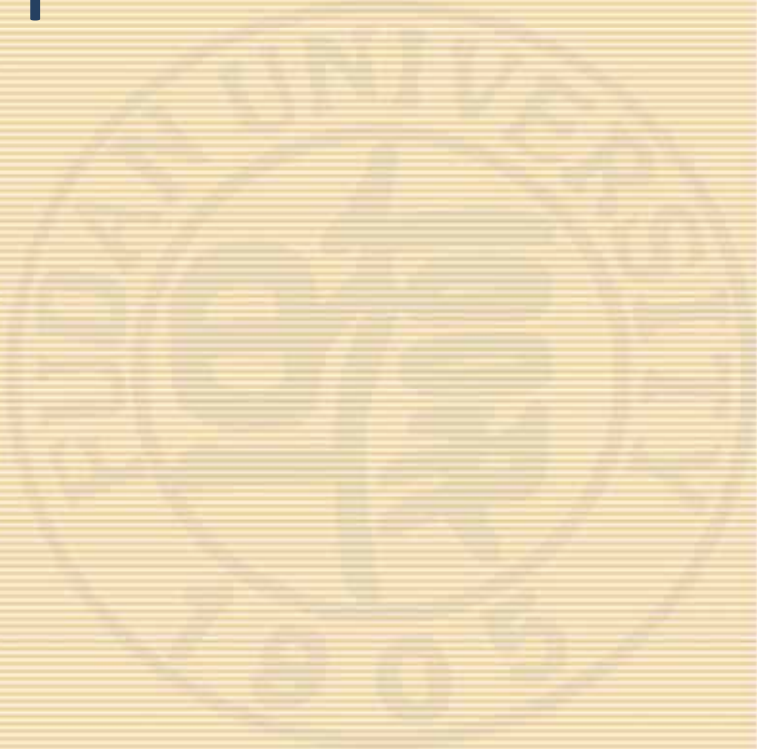
Integer Operations Outline

- Arithmetic Operations
  - overflow
  - Unsigned addition, multiplication
  - Signed addition, negation, multiplication
  - Using Shift to perform power-of-2 multiply/divide
- Suggested reading
  - Chap 2.3

Lab1

- Implement following functions
- bitAnd(x,y)
  - Duplicate the behavior of the bit operation &
  - Only use operations | and ~
  - Rating: 1 / Max Ops: 8
- bitOr(x,y)
  - Duplicate the behavior of the bit operation |
  - Only use the operations & and ~
  - Rating: 1 / Max Ops: 8
- isEqual(x,y)
  - Compares x to y, i.e. x==y. It should return 1 if the tested condition holds and 0 otherwise.
  - Rating: 2 / Max Ops: 5
- logicalShift(x,n)
  - Does a logical right shift of x by n.
  - Rating: 3 / Max Ops: 16

Machine-Level Representation of Programs I

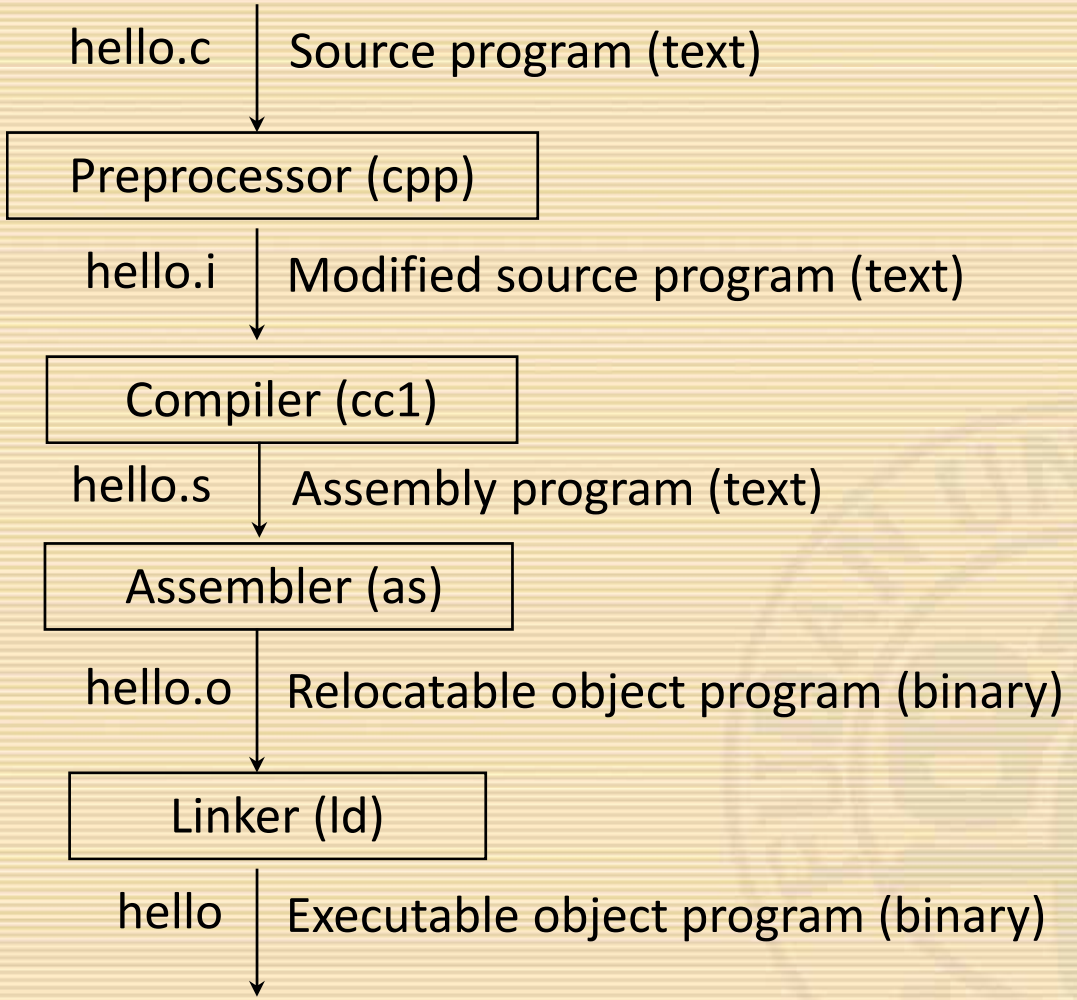


jc@fudan.edu.cn

The Hello Program

- The C programs are translated into
  - A sequence of low-level *machine-language* instructions
- These instructions are then packaged in a form
  - called an *object program*
- *Object program* are stored as a binary disk file
  - Also referred to as *executable object files*

The Context of a Compiler



Why should we understand the assembly code

- Understand the optimization capabilities of the compiler
- Analyze the underlying inefficiencies in the code
- Sometimes the run-time behavior of a program is needed

From writing assembly code to understanding assembly code

- Different set of skills
  - Transformations
  - Relation between source code and assembly code
- *Reverse engineering*
  - Trying to understand the process by which a system was created
    - By studying the system and
    - By working backward

Data layout

- Object model in C
  - Different data types can be declared

Data layout

- Object model in assembly
  - A large, byte-addressable array
  - No distinctions even between signed or unsigned integers
  - Code, user data, OS data
  - Run-time stack for managing procedure call and return
  - Blocks of memory allocated by user

Operations in C constructs

- Arithmetic expression evaluation
- Loops
- Procedure calls and returns
- Translated into sequences of instructions

Operations in Assembly Instructions

- Performs only a very elementary operation
- Normally one by one in sequential
- Operate data stored in registers
- Transfer data between memory and a register
- Conditionally branch to a new instruction address

Registers

%eax	%ax	%ah	%al
%edx	%dx	%dh	%dl
%ecx	%cx	%ch	%cl
%ebx	%bx	%bh	%bl
%esi			
%edi			
%esp			
%ebp			

What are needed?

- What are the minimum set of information a computer needs to know to ensure proper running of an assembly program?

- What about C Language?

Programmer-Visible States

- Program Counter (%eip)
  - Address of the next instruction
- Register File
  - Heavily used program data
  - Integer and floating-point
- Condition codes register
  - Hold status information about the most recently executed instruction
  - Implement conditional changes in the control flow

Types of Instructions

- What types of instructions do we have in C?

Moving Data

- Moving Data
  - movl *Source*,*Dest*:
  - Move 4-byte (“long”) word
  - Lots of these in typical code
- Operand Types
  - Immediate: Constant integer data
    - Like C constant, but prefixed with ‘$’
    - E.g., $0x400, $-533
    - Encoded with 1, 2, or 4 bytes
  - Register: One of 8 integer registers
    - But %esp and %ebp reserved for special use
    - Others have special uses for particular instructions
  - Memory: 4 consecutive bytes of memory
    - Various “address modes”

Code Examples

C code

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

Obtain with command

```
gcc –O2 -S code.c
```

Assembly file code.s

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

Code Examples

55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3	Obtain with command `gcc –O2 -c code.c` Relocatable object file `code.o`

Code Examples

Obtain with command

```
objdump -d code.o
```

Disassembly output

```
0x80483b4 <sum>:
0x80483b4   55                  push   %ebp
0x80483b5   89 e5               mov    %esp,%ebp
0x80483b7   8b 45 0c            mov    0xc(%ebp),%eax
0x80483ba   03 45 08            add    0x8(%ebp),%eax
0x80483bd   89 ec               mov    %ebp,%esp
0x80483bf   5d                  pop    %ebp
0x80483c0   c3                  ret
                                nop
```

C Code

- Add two signed integers
- int t = x+y;

Assembly Code

- Operands:
  - x: Register %eax
  - y: Memory M[%ebp+8]
  - t: Register %eax
- Instruction
  - addl 8(%ebp),%eax
  - Add 2 4-byte integers
  - Similar to expression x +=y
- Return function value in %eax

Object Code

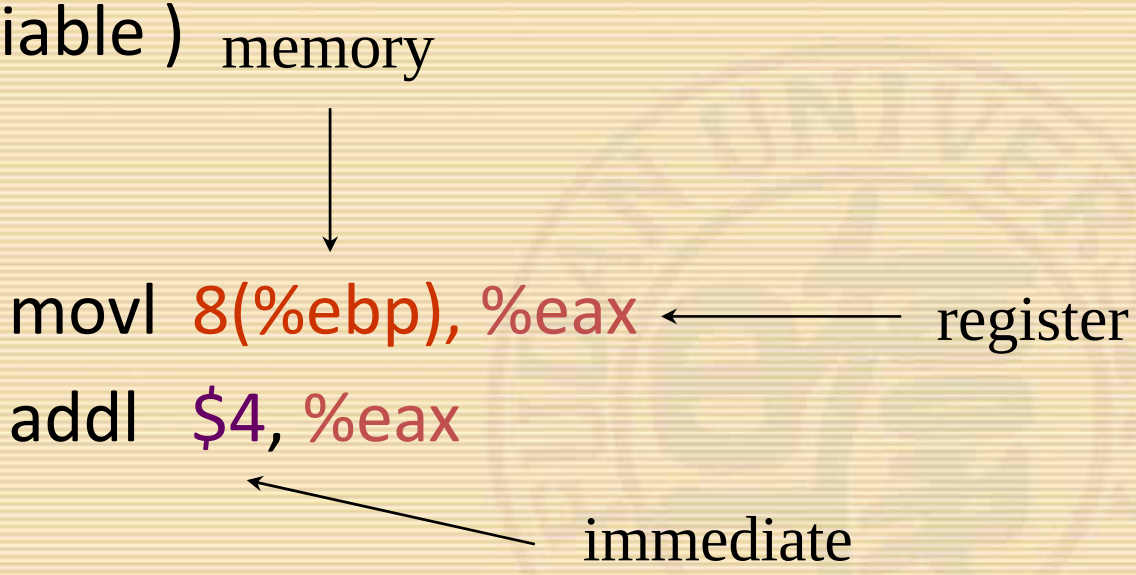
- 3-byte instruction
- Stored at address 0x80483ba
- 0x80483ba : 03 45 08

Operands

- In high level languages
  - Either constants （常数）
  - Or variable （变量）
- Example
  - A = A + 4

Operands

- Counterparts in assembly languages
  - Immediate (constant)
  - Register (variable)
  - Memory (variable)
- Example



Simple Addressing Mode

- Immediate
  - represents a constant
  - The format is $imm
    - $4, $0xffffffff
- Registers
  - The fastest storage units in computer systems
  - Typically 32-bit long
  - Register mode E_a
    - The value stored in the register
    - Noted as R[E_a]

Memory References

- The name of the array is annotated as M
- If *addr* is a memory address
- M[*addr*] is the content of the memory starting at *addr*
- *addr* is used as an array index
- How many bytes are there in M[*addr*]?
  - It depends on the context

Memory Addressing Mode

- An expression for
  - a memory address (or an array index)
- Most general form
  - imm (E_b, E_i, s)
  - s: 1, 2, 4, 8
- The address represented by the above form
  - imm + R[E_b] + R[E_i] * s
- It gives the value
  - M[imm + R[E_b] + R[E_i] * s]

Addressing Mode

Type	Form	Operand value	Name
Immediate	$Imm	Imm	Immediate
Register	E_a	R[E_a]	Register
Memory	Imm	M[Imm]	Absolute
Memory	(E_a)	M[R[E_a]]	Indirect
Memory	Imm(E_b)	M[Imm+ R[E_b]]	Base+displacement
Memory	(E_b, E_i)	M[R[E_b]+ R[E_i]*s]	Indexed
Memory	Imm(E_b, E_i)	M[Imm+ R[E_b]+ R[E_i]]	Scaled indexed
Memory	(, E_i, s)	M[R[E_i]*s]	Scaled indexed
Memory	(E_b, E_i, s)	M[R[E_b]+ R[E_i]*s]	Scaled indexed
Memory	Imm(E_b, E_i, s)	M[Imm+ R[E_b]+ R[E_i]*s]	Scaled indexed

Data Formats

C declaration	Intel data type	GAS suffix	Size (byte)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
unsigned	Double word	l	4
long int	Double word	l	4
unsigned long	Double word	l	4
char *	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

Data Formats

- Move data instruction
  - mov (general)
  - movb (move byte)
  - movw (move word)
  - movl (move double word)

Data Movement

Instruction	Effect	Description
movl S, D	D ← S	Move double word
movw S, D	D ← S	Move word
movb S, D	D ← S	Move byte
movsbl S, D	D ← SignedExtend(S)	Move sign-extended byte
movzbl S, D	D ← ZeroExtend(S)	Move zero-extended byte
pushl S	R[%esp] ← R[%esp]-4 M[R[%esp]] ← S	Push
popl D	D ← M[R[%esp]] R[%esp] ← R[%esp]+4	Pop

Move Instructions

- Format
  - movl src, dest
  - src and dest can only be one of the following
    - Immediate (except dest)
    - Register
    - Memory

Move Instructions

- Format
  - The only possible combinations of the (src, dest) are
    - (immediate, register)
    - (memory, register) load
    - (register, register)
    - (immediate, memory) store
    - (register, memory) store

Data Movement Example

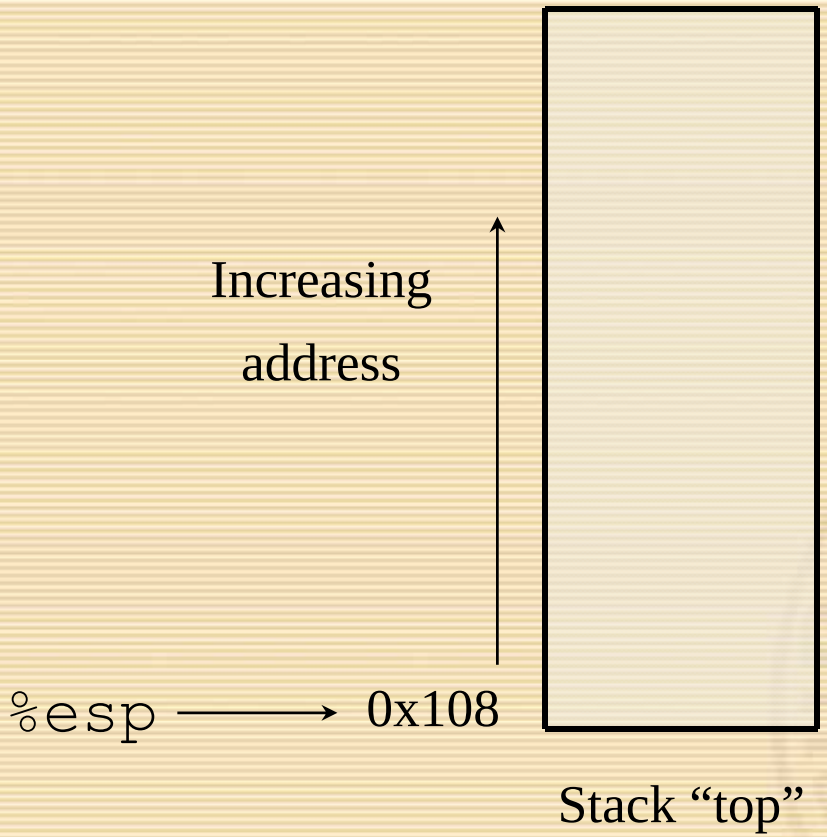
movl $0x4050, %eax	*immediate*	*register*
movl %ebp, %esp	*register*	*register*
movl (%edx, %ecx), %eax	*memory*	*register*
movl $-17, (%esp)	*immediate*	*memory*
movl %eax, -12(%ebp)	*register*	*memory*

Data Movement Example

Initial value %dh=8d %eax =0x98765432

- movb %dh, %al %eax=0x98765<u>48d</u>
- movsbl %dh, %eax %eax=0x<u>ffffff8d</u>
- movzbl %dh, %eax %eax=0x<u>0000008d</u>

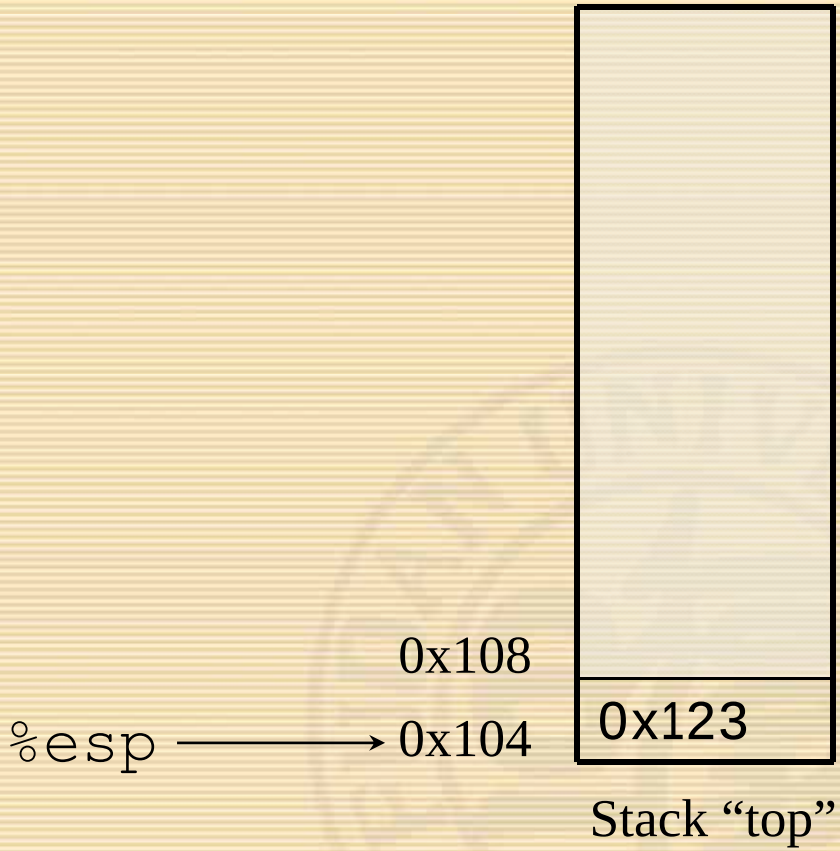
Stack Operations



Stack Operations

%eax	0x123
%edx	0
%esp	0x104

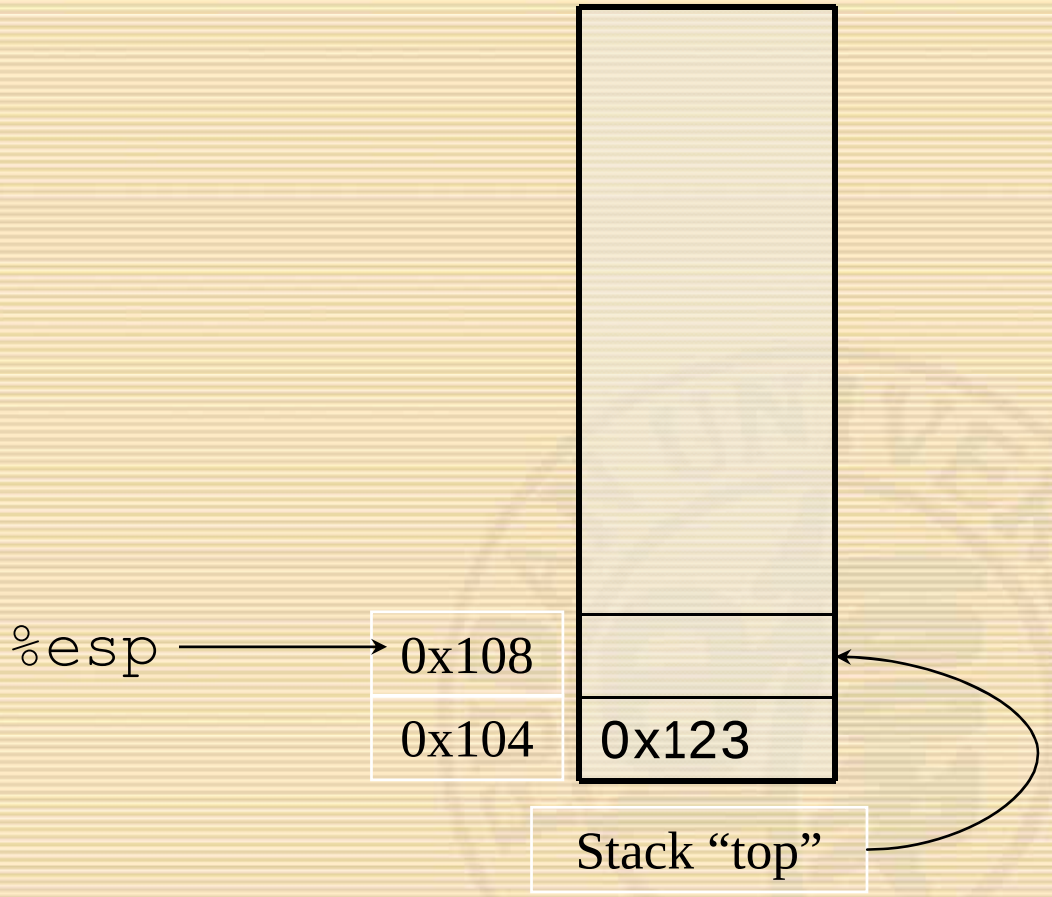
pushl %eax



Stack Operations

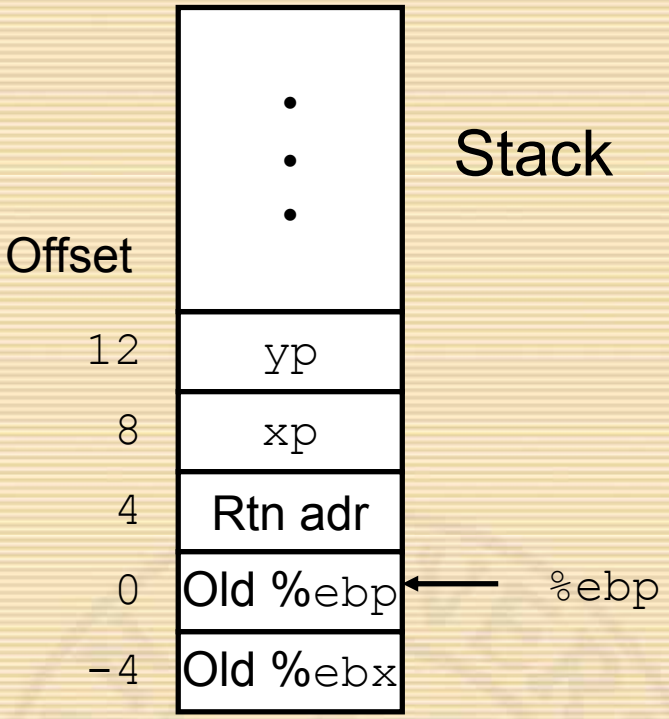
%eax	0x123
%edx	0x123
%esp	0x104

popl %edx



Understanding Swap

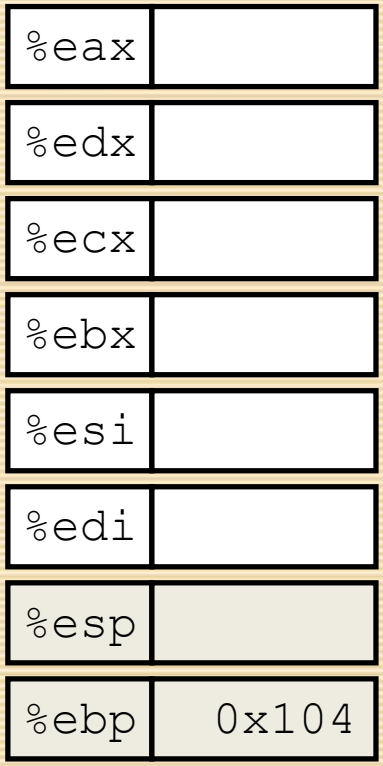
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

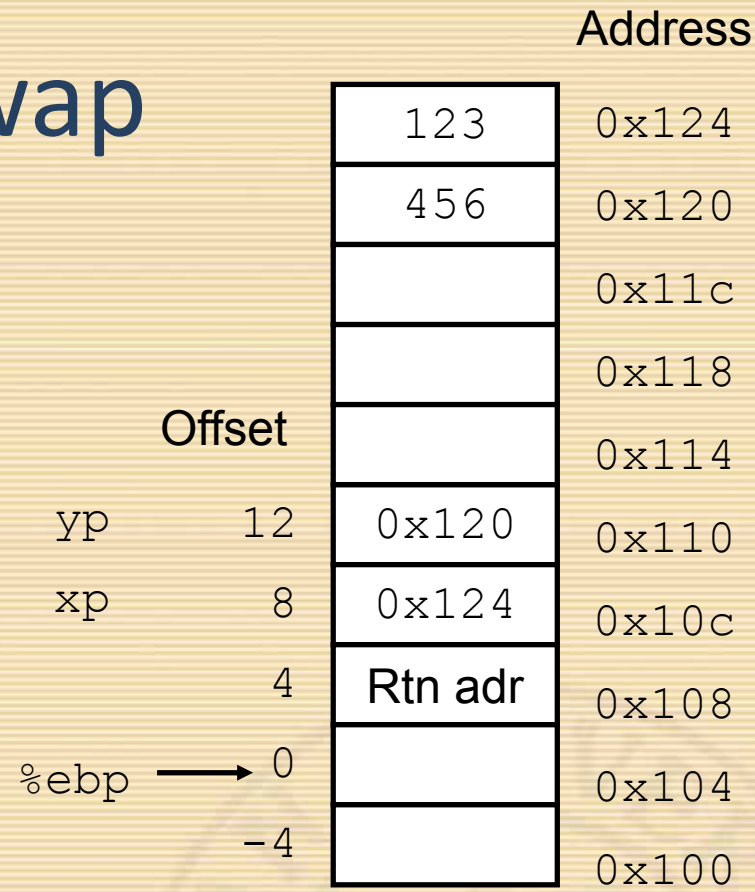


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

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

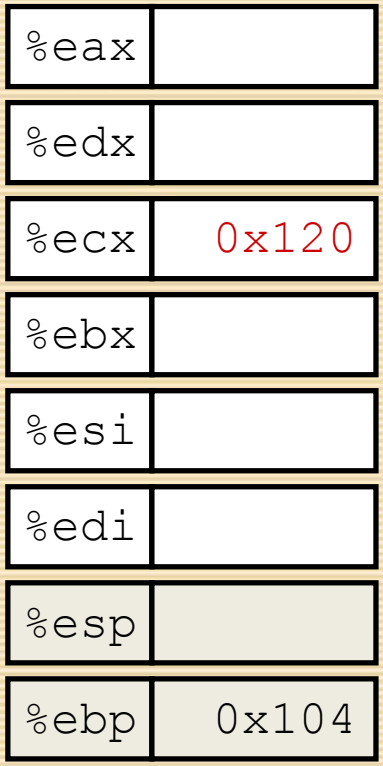
Understanding Swap

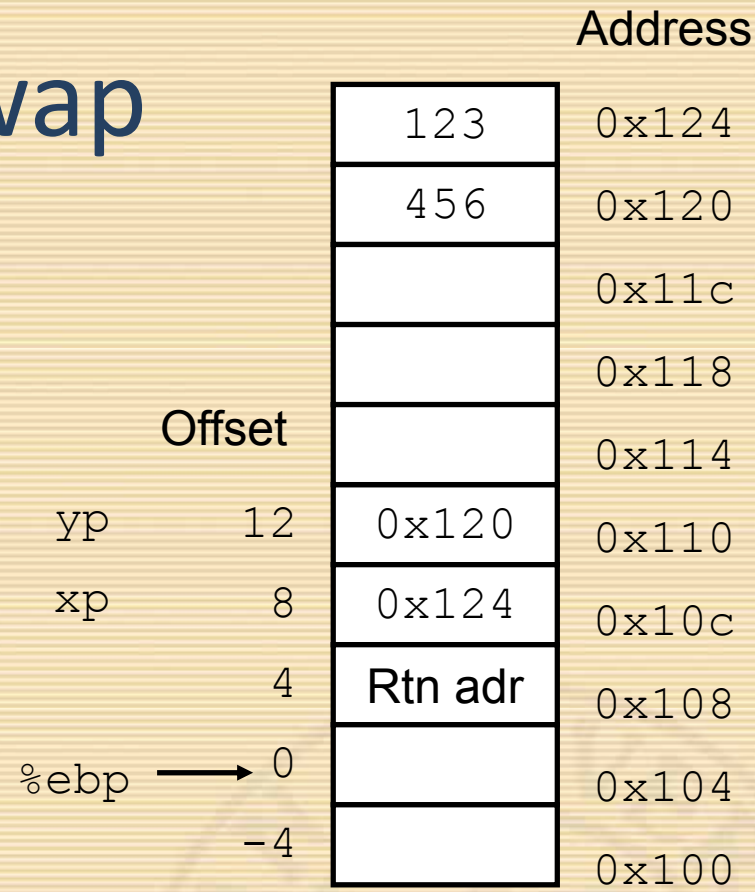





```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

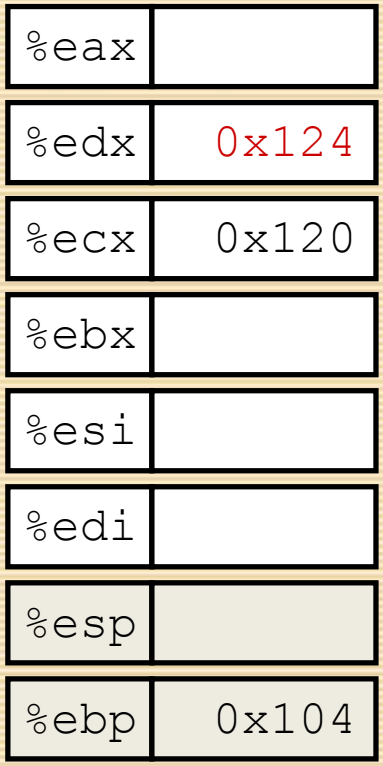
Understanding Swap

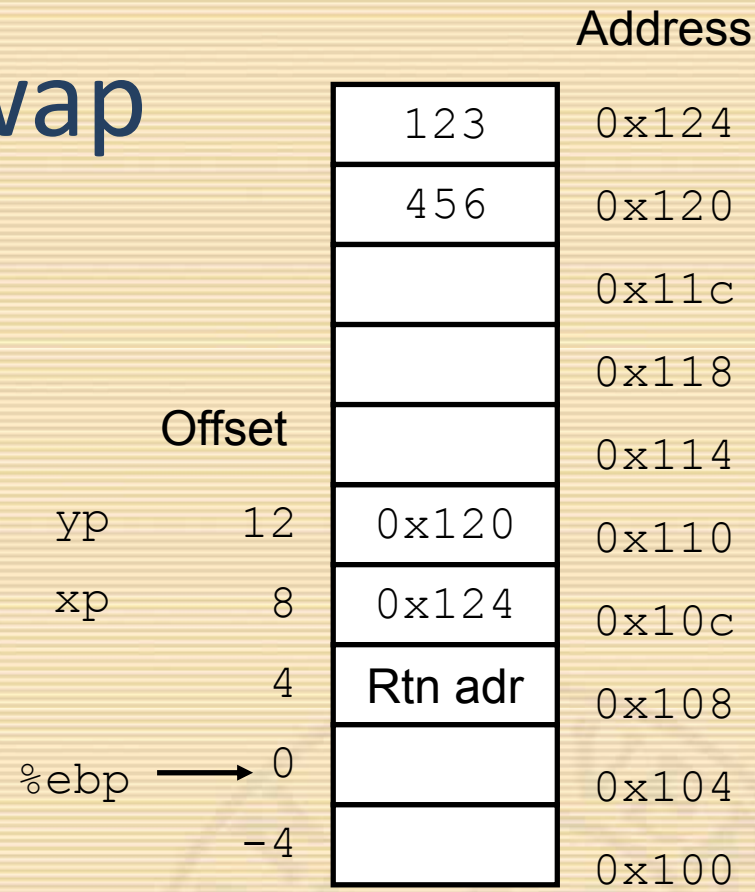





```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

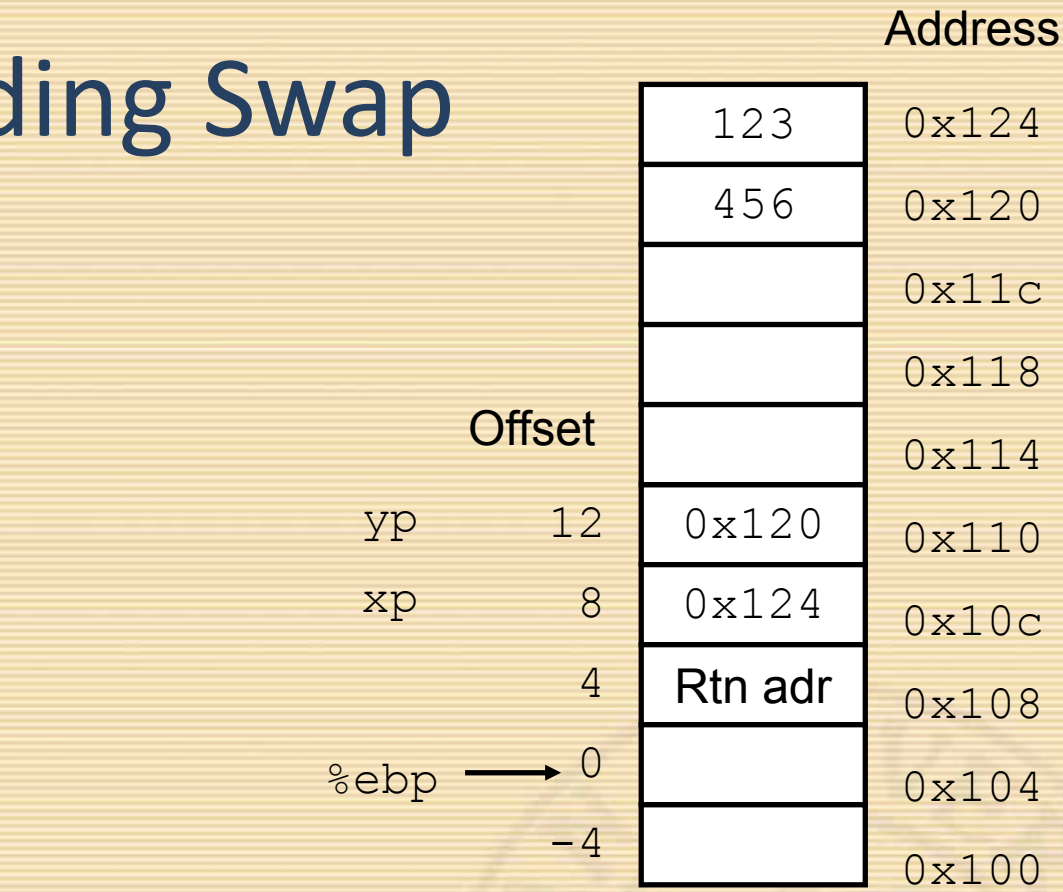





```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

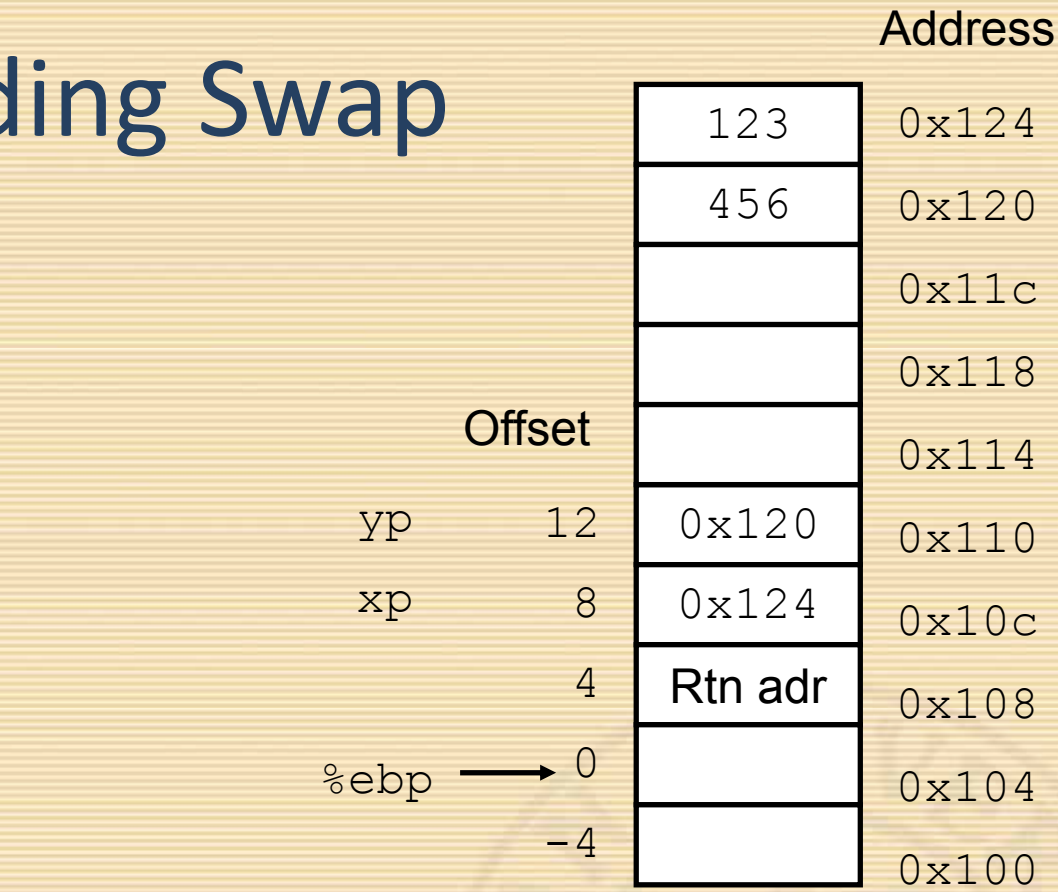
%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




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

Understanding Swap

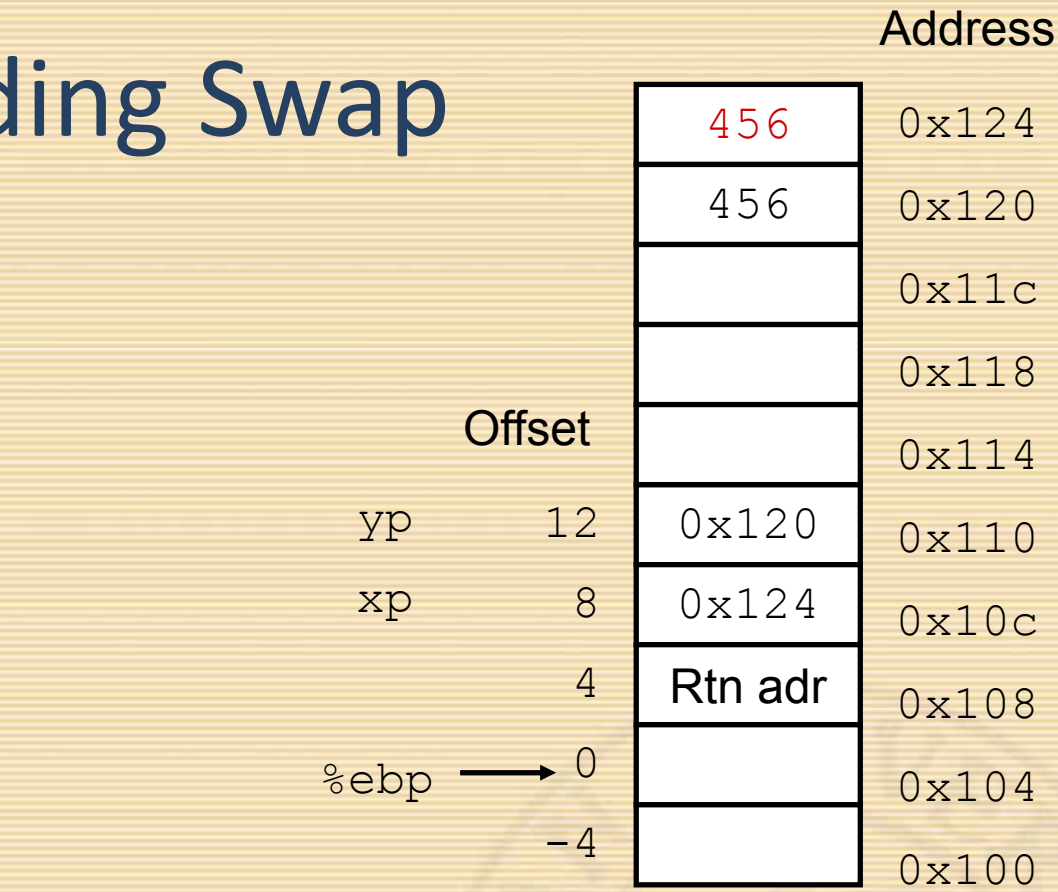
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




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

Understanding Swap

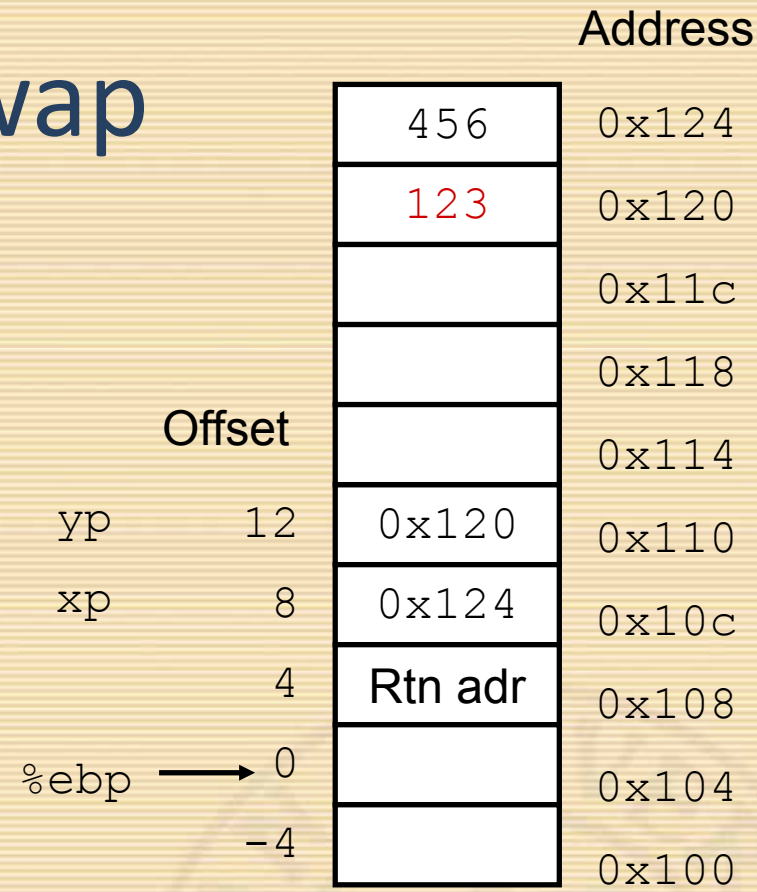
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




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

Understanding Swap

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




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

Address Computation Examples

%edx	0xf000
%ecx	0x100

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

Address Computation Instruction

- **`leal`** *Src,Dest*
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression
- Uses
  - Computing address without doing memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8.

Some Arithmetic Operations

Format	Computation	
• Two Operand Instructions		
`addl` *Src,Dest*	*Dest* = *Dest* + *Src*	
`subl` *Src,Dest*	*Dest* = *Dest* – *Src*	
`imull` *Src,Dest*	*Dest* = *Dest* * *Src*	
`sall` *Src,Dest*	*Dest* = *Dest* << *Src*	Also called `shll`
`sarl` *Src,Dest*	*Dest* = *Dest* >> *Src*	Arithmetic
`shrl` *Src,Dest*	*Dest* = *Dest* >> *Src*	Logical
`xorl` *Src,Dest*	*Dest* = *Dest* ^ *Src*	
`andl` *Src,Dest*	*Dest* = *Dest* & *Src*	
`orl` *Src,Dest*	*Dest* = *Dest* | *Src*	

Some Arithmetic Operations

Format	Computation
• One Operand Instructions	
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* – 1
`negl` *Dest*	*Dest* = – *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

Arithmetic and Logical Operations

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination	Value
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull $16, (%eax, %edx, 4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx, %eax	%eax	0xFD

Assembly Code for Arithmetic Expressions

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z*48;
  int t3 = t1&0xFFFF;
  int t4 = t2*t3;
  return t4;
}
```

```
movl 12(%ebp),%eax        Get y
movl 16(%ebp),%edx        Get z
addl 8(%ebp),%eax         Compute t1=x+y
leal (%edx,%edx,2),%edx   Compute 3*z
sall $4,%edx              Compute t2=48*z
andl $0xFFFF,%eax         Compute t3=t1&FFFF
imull %edx,%eax           Compute t4=t2*t3
```

Special Arithmetic Operations

imull S	R[%edx]:R[%eax] ←S*R[%eax]	Signed full multiply
mull S	R[%edx]:R[%eax] ←S*R[%eax]	Unsigned full multiply
Cltd	R[%edx]:R[%eax] ← SignExtend(R[%eax])	Convert to quad word
idivl S	R[%edx] ← R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] / S	Signed divide
divl S	R[%edx] ←R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] / S	Unsigned divide

Whose Assembler?

Intel/Microsoft Format	GAS/Gnu Format
`lea eax,[ecx+ecx*2]`	`leal (%ecx,%ecx,2),%eax`
`sub esp,8`	`subl $8,%esp`
`cmp dword ptr [ebp-8],0`	`cmpl $0,-8(%ebp)`
`mov eax,dword ptr [eax*4+100h]`	`movl 0x100(,%eax,4),%eax`

- Intel/Microsoft Differs from GAS
  - Operands listed in opposite order

    `mov` Dest, Src `movl` Src, Dest
  - Constants not preceded by '$', Denote hex with 'h' at end

    `100h` `$0x100`
  - Operand size indicated by operands rather than operator suffix

    `sub` `subl`
  - Addressing format shows effective address computation

    `[eax*4+100h]` `0x100(,%eax,4)`

Control

- Two of the most important parts of program execution
  - Data flow (Accessing and operating data)
  - Control flow (control the sequence of operations)

Control

- Sequential execution is default
  - The statements in C and the instructions in assembly code are executed in the order they appear in the program
- Change the control flow
  - Control constructs in C
  - Jump in assembly

Condition Codes

- Condition codes
  - A set of single-bit
  - Maintained in a condition code register
  - Describe attributes of the most recently arithmetic or logical operation

Condition Codes

- EFLAGS
  - CF: Carry Flag
    - The most recent operation generated a carry out of the most significant bit
    - Used to detect overflow for unsigned operations
  - OF: Overflow Flag
    - The most recent operation caused a two's complement overflow — either negative or positive
  - ZF: Zero Flag
    - The most recent operation yielded zero
  - SF: Sign Flag
    - The most recent operation yielded a negative value

Setting Condition Codes

- Implicit Setting By Arithmetic Operations

  `addl` *Src, Dest*

  C analog: `t = a+b`

  – CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  – ZF set if `t == 0`
  – SF set if `t < 0`
  – OF set if two's complement overflow
    - `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

Condition Code

- lea instruction
  - has no effect on condition codes
- xor instruction
  - The carry and overflow flags are set to 0
- shift instruction
  - carry flag is set to the last bit shifted out
  - Overflow flag is set to 0

Setting Condition Codes I

- Explicit Setting by Compare Instruction

  `cmpl` *Src2,Src1*

  – `cmpl 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`

  – OF set if two's complement overflow

```
(a>0 && b<0 && (a-b)<0) ||
(a<0 && b>0 && (a-b)>0)
```

Setting Condition Codes II

- Explicit Setting by Test instruction

  `testl` *Src2,Src1*

  – Sets condition codes based on value of *Src1* `&` *Src2*

    - Useful to have one of the operands be a mask

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

  – ZF set when `a&b == 0`

  – SF set when `a&b < 0`

Accessing Condition Codes

- The condition codes cannot be read directly
- One of the most common methods of accessing them is
  - setting an integer register based on some combination of condition codes
  - `Set` *Des*
- After each set command is executed
  - A single byte to 0 or to 1 is obtained
- The descriptions of the different set commands apply to the case
  - where a comparison instruction has been executed

Accessing Condition Codes

Instruction	Synonym	Effect	Set Condition
Sete	Setz	ZF	Equal/zero
Setne	Setnz	~ZF	Not equal/not zero
Sets		SF	Negative
Setns		~SF	Nonnegative
Setl	Setnge	SF^OF	Less
Setle	Setng	(SF^OF)|ZF	Less or Equal
Setg	Setnle	~(SF^OF)&~ZF	Greater
Setge	Setnl	~(SF^OF)	Greater or Equal
Seta	Setnbe	~CF&~ZF	Above
Setae	Setnb	~CF	Above or equal
Setb	Setnae	CF	Below
Setbe	Setna	CF|ZF	Below or equal

Accessing Condition Codes

- The destination operand is either
  - One of the eight single-byte register elements or
  - A memory location where the single byte is to be stored

- To generate a 32-bit result
  - We must also clear the high-order 24 bits

Accessing Condition Codes

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

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

Jumping

jX *label*	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)
`jae`	`~CF`	Above or Equal(unsigned)
`jb`	`CF`	Below (unsigned)
`jbe`	`CF|ZF`	Below or Equal(unsigned)

Conditional Branch Example

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

```
_max:
        pushl %ebp
        movl %esp,%ebp

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

Conditional Branch Example (Cont.)

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

- C allows "goto" as means of transferring control
  - Closer to machine-level programming style
- Generally considered as bad coding style

```
  movl 8(%ebp),%edx    # edx = x
  movl 12(%ebp),%eax   # eax = y
  cmpl %eax,%edx       # x : y
  jle L9               # if <= goto L9
  movl %edx,%eax       # eax = x
L9:                    # Done:
```

} Skipped when x ≤ y

Loops

jc@fudan.edu.cn

"Do-While" Loop Example

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Use backward branch to continue looping
- Only take branch when "while" condition holds

"Do-While" Loop Compilation

Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Registers
  - %edx x
  - %eax result

Assembly

```
_fact_goto:
   pushl %ebp          # Setup
   movl %esp,%ebp      # Setup
   movl $1,%eax        # eax = 1
   movl 8(%ebp),%edx   # edx = x

L11:
   imull %edx,%eax     # result *= x
   decl %edx           # x--
   cmpl $1,%edx        # Compare x : 1
   jg L11              # if > goto loop

   movl %ebp,%esp      # Finish
   popl %ebp           # Finish
   ret                 # Finish
```

General "Do-While" Translation

C Code

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

Goto Version

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

– *Body* can be any C statement

- Typically compound statement:

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

– *Test* is expression returning integer

= 0 interpreted as false ≠0 interpreted as true

"While" Loop Example #1

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

First Goto Version

```
int fact_while_goto
  (int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Actual "While" Loop Translation

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- Uses same inner loop as do-while version
- Guards loop entry with extra test

Second Goto Version

```
int fact_while_goto2
  (int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

General "While" Translation

C Code

```
while (Test)
  Body
```

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:
```

"For" Loop Example

```
/* A strange function */
int ipwr_for(int x, unsigned p) {
int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

- Algorithm
  - Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots 2^{n-1}p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}{}^2)^2)\ldots)^2}_{n-1 \text{ times}}$

    $z_i = 1$ when $p_i = 0$

    $z_i = x$ when $p_i = 1$
  - Complexity $O(\log p)$

Example

$3^{10} = 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

ipwr Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

result	x	p
1	3	10
1	9	5
9	81	2
9	6561	1
531441	43046721	0

“For” Loop Example

```
int result;
for (result = 1;
     p != 0;
     p = p>>1) {
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

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

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"For" → "Goto"

For Version

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

While Version

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

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

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

"For" Loop Compilation

Goto Version

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

```
   result = 1;
   if (p == 0)
     goto done;
loop:
   if (p & 0x1)
     result *= x;
   x = x*x;
   p = p >> 1;
   if (p != 0)
     goto loop;
done:
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Switch Statements

- Implementation Options
  - Series of conditionals
    - Good if few cases
    - Slow if many
  - Jump Table
    - Lookup branch target
    - Avoids conditionals
    - Possible when cases are small integer constants
  - GCC
    - Picks one based on case structure
  - Bugs in example code?

Jump Table Structure

Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    . . .
  case val_n-1:
    Block n–1
}
```

Jump Table

jtab:

Targ0
Targ1
Targ2
. . .
Targ*n*–1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

. . .

Targ*n*–1: Code Block *n*–1

Approx. Translation

```
target = JTab[op];
goto *target;
```

Switch Statement Example

- Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

```
ADD   0
MULT  1
MINUS 2
DIV   3
MOD   4
BAD   5
```

Setup:

```
unparse_symbol:
   pushl %ebp          # Setup
   movl %esp,%ebp      # Setup
   movl 8(%ebp),%eax   # eax = op
   cmpl $5,%eax        # Compare op : 5
   ja .L49             # If > goto done
   jmp *.L57(,%eax,4) # goto Table[op]
```

Assembly Setup Explanation

- Symbolic Labels
  - Labels of form `.L`*XX* translated into addresses by assembler
- Table Structure
  - Each target requires 4 bytes
  - Base address at `.L57`
- Jumping

  `jmp .L49`
  - Jump target is denoted by label `.L49`

  `jmp *.L57(,%eax,4)`
  - Start of jump table denoted by label .L57
  - Register `%eax` holds `op`
  - Must scale by factor of 4 to get offset into table
  - Fetch target from effective Address `.L57 + op*4`

Jump Table

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

jc@fudan.edu.cn

Switch Statement Completion

```
.L49:                           # Done:
    movl %ebp,%esp              # Finish
    popl %ebp                   # Finish
    ret                         # Finish
```

- Puzzle
  - What value returned when `op` is invalid?
- Answer
  - Register `%eax` set to `op` at beginning of procedure
  - This becomes the returned value
- Advantage of Jump Table
  - Can do *k*-way branch in *O*(1) operations

Summarizing

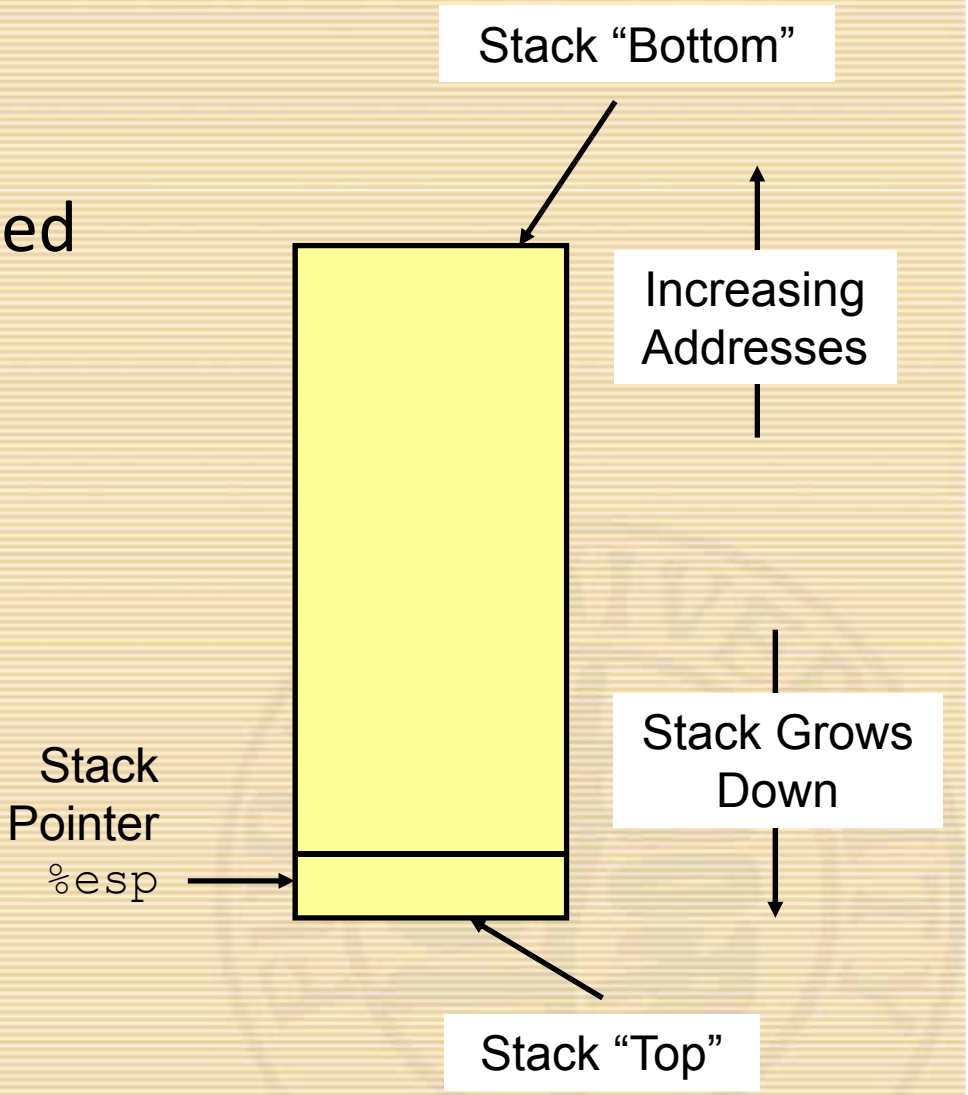
- C Control
  - if-then-else
  - do-while
  - While
  - for
  - switch
- Assembler Control
  - jump
  - Conditional jump
- Compiler
  - Must generate assembly code to implement more complex control
- Standard Techniques
  - All loops converted to do-while form
  - Large switch statements use jump tables

Machine-Level Programming III: Procedures

jc@fudan.edu.cn

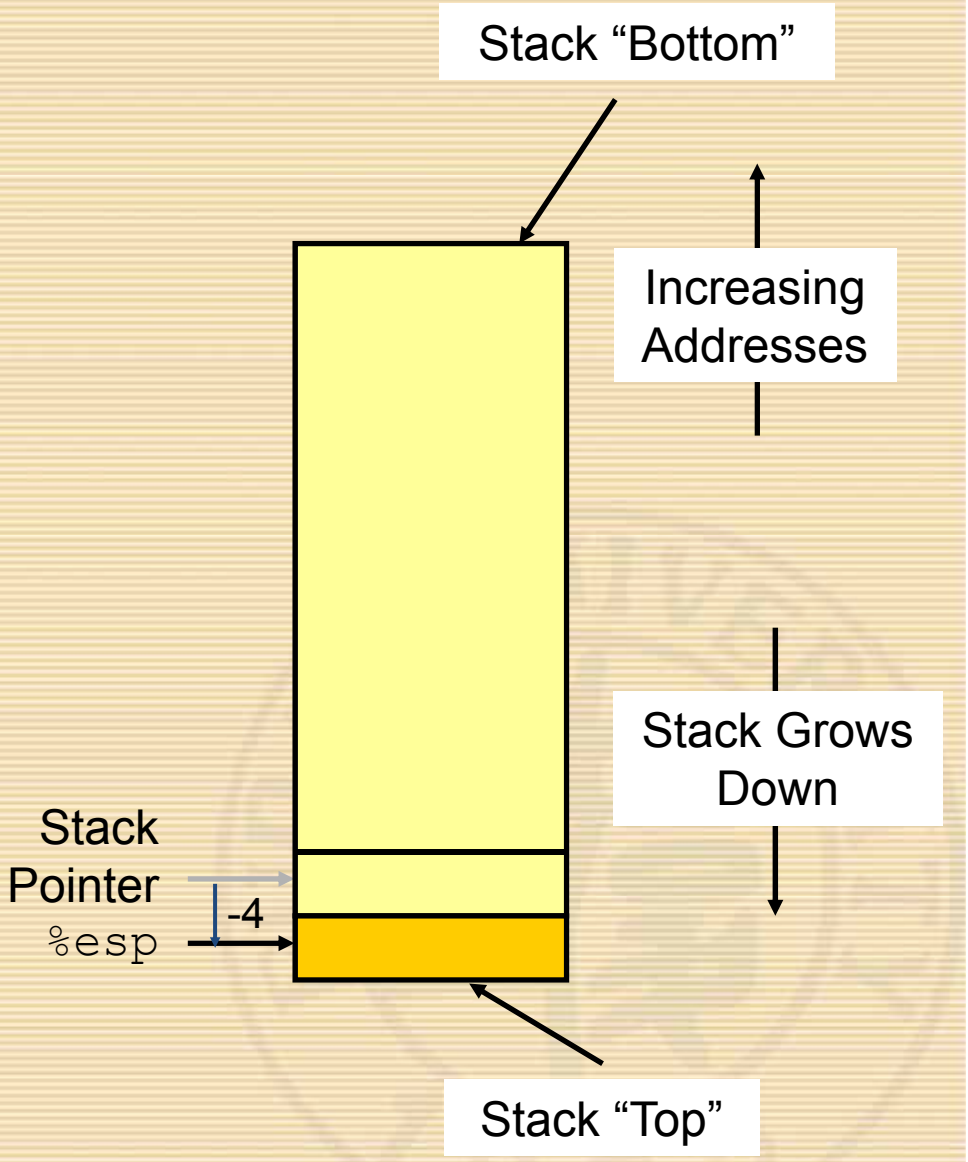
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



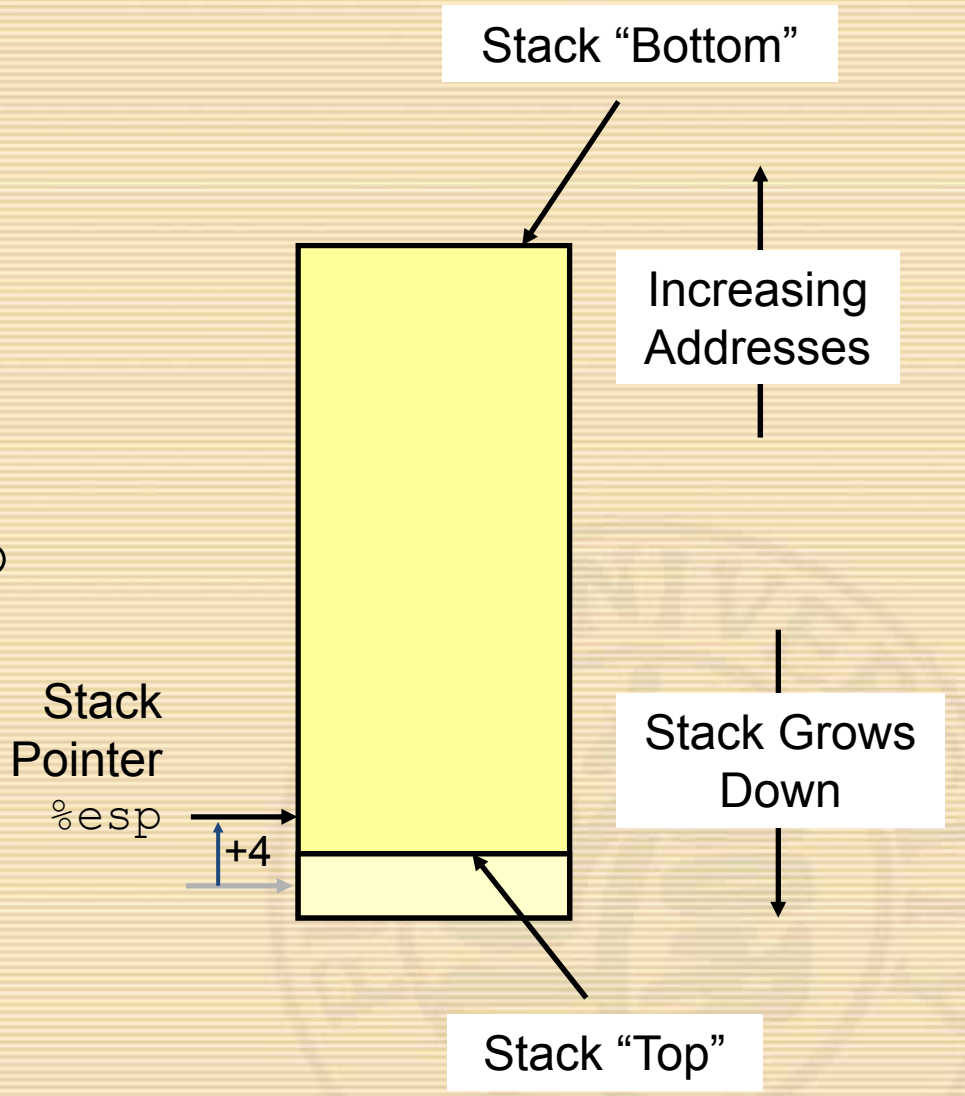
IA32 Stack Pushing

- Pushing
  - `pushl` *Src*
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`



IA32 Stack Popping

- Popping
  - `popl` *Dest*
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*



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 value
  - Address of instruction beyond `call`
  - Example from disassembly

```
804854e: e8 3d 06 00 00    call   8048b90 <main>
8048553: 50                pushl  %eax
```

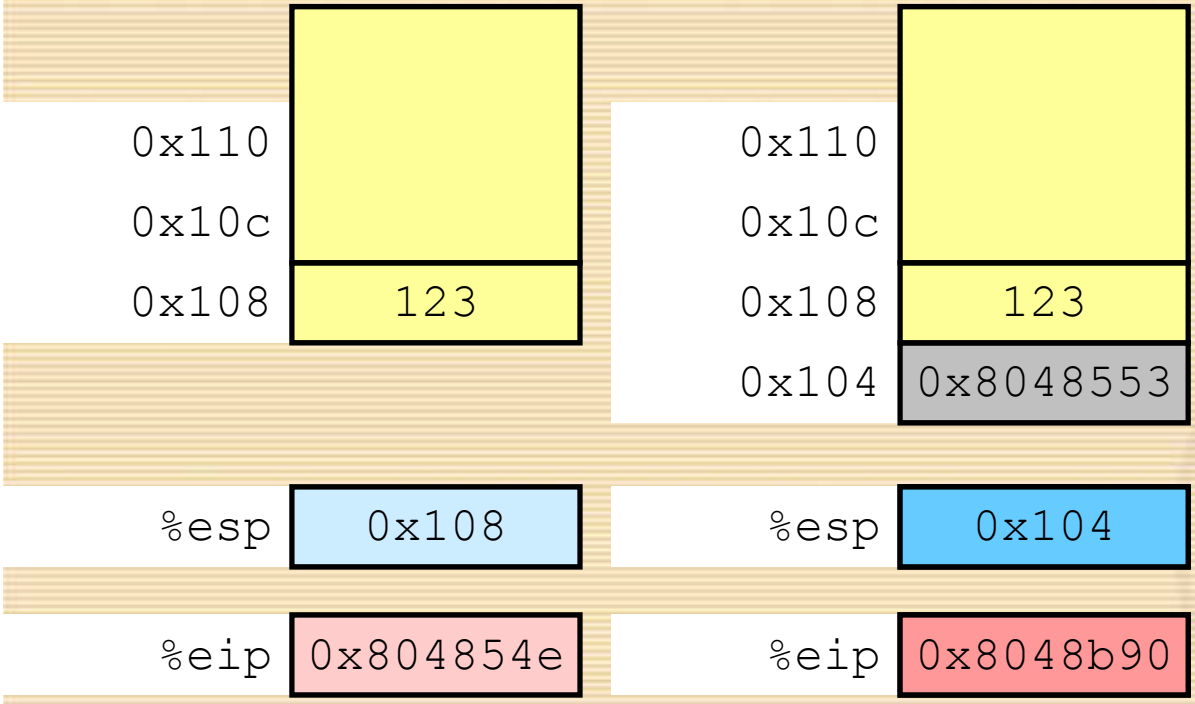
    - Return address = `0x8048553`

- Procedure return:
  - `ret` Pop address from stack; Jump to address

Procedure Call Example

```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    pushl  %eax
```

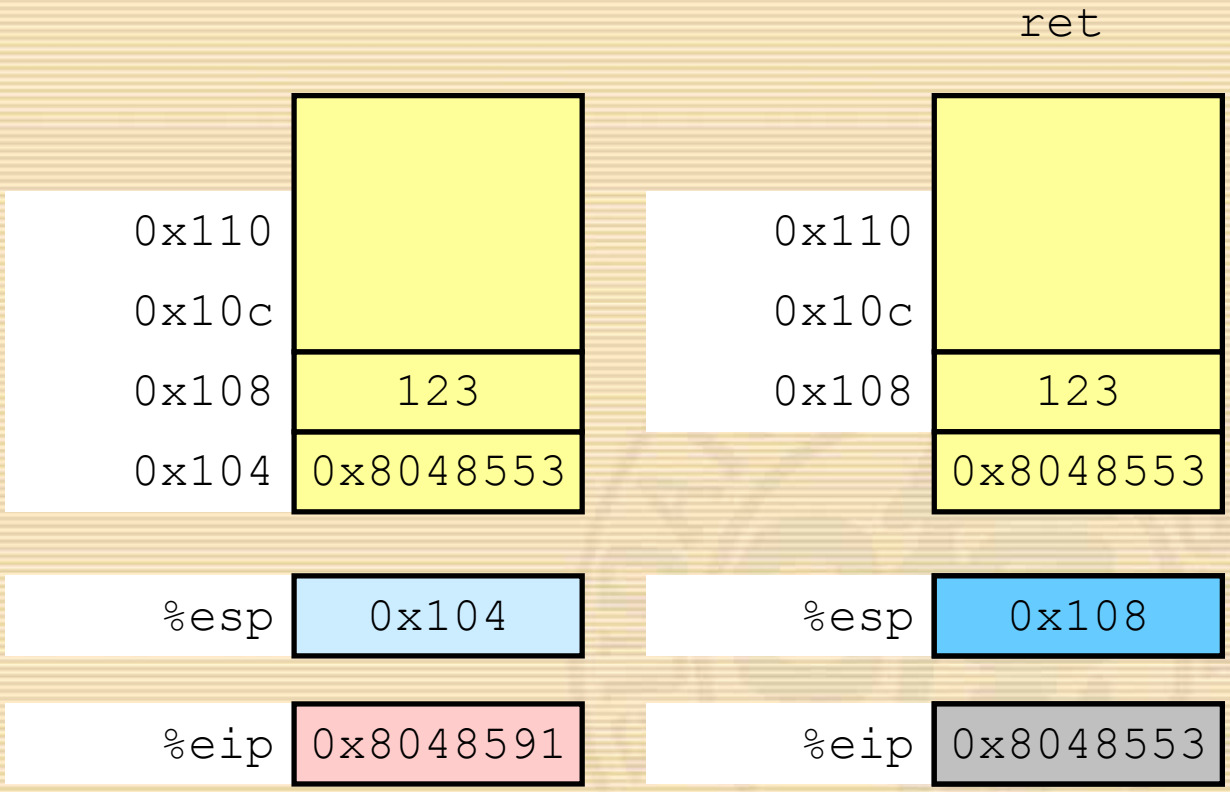
```
                call   8048b90
```



`%eip` is program counter

Procedure Call Example

```
8048591:  c3                    ret
```



`%eip` is program counter

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

- Code Structure

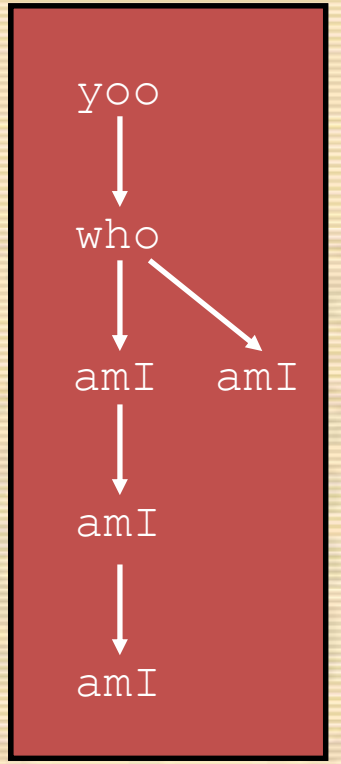
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

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

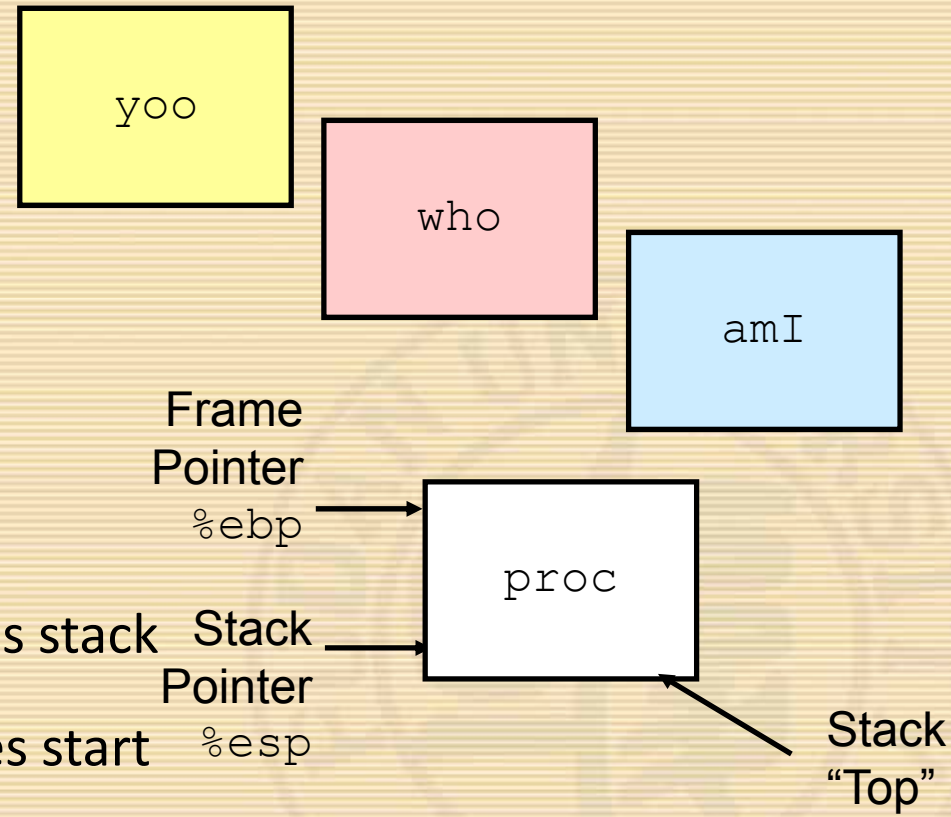
– Procedure `amI` recursive

Call Chain



Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code
- Pointers
  - Stack pointer `%esp` indicates stack top
  - Frame pointer `%ebp` indicates start of current frame

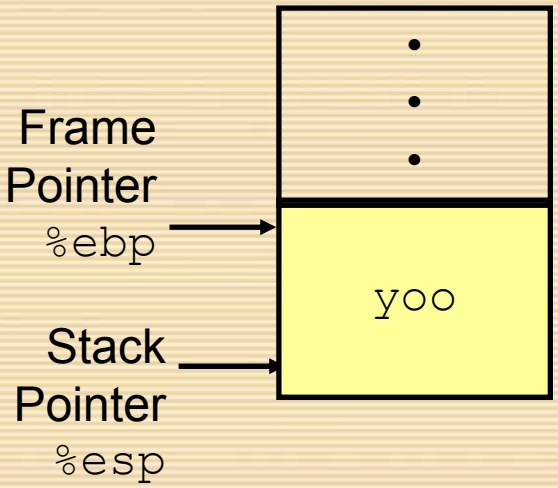


Stack Operation

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

Call Chain

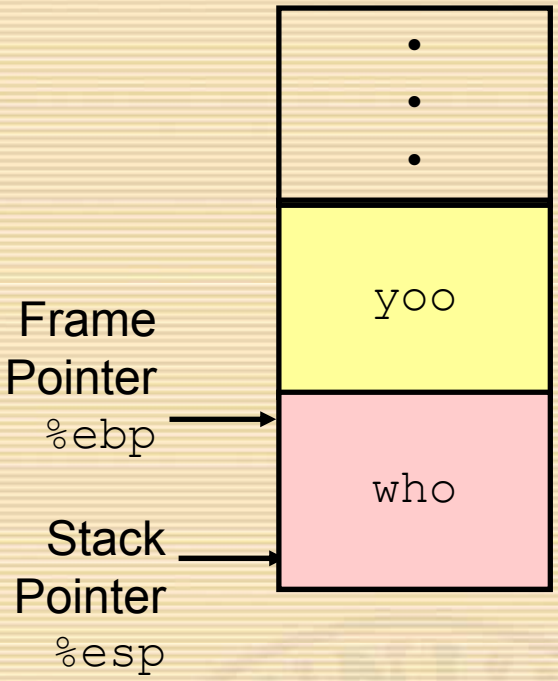
yoo



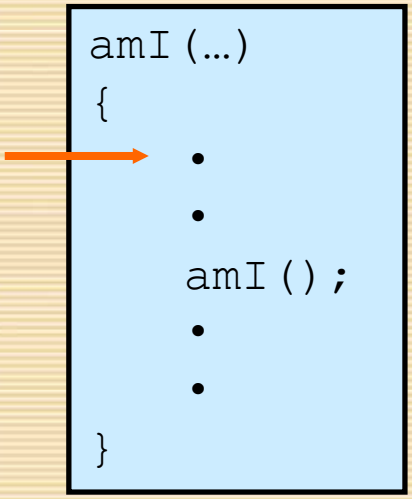
Stack Operation

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

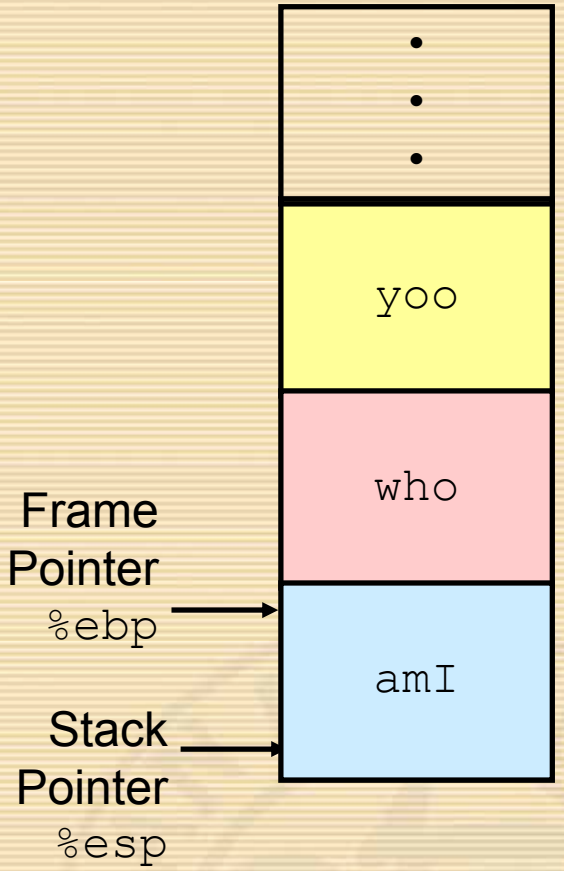
Call Chain



Stack Operation



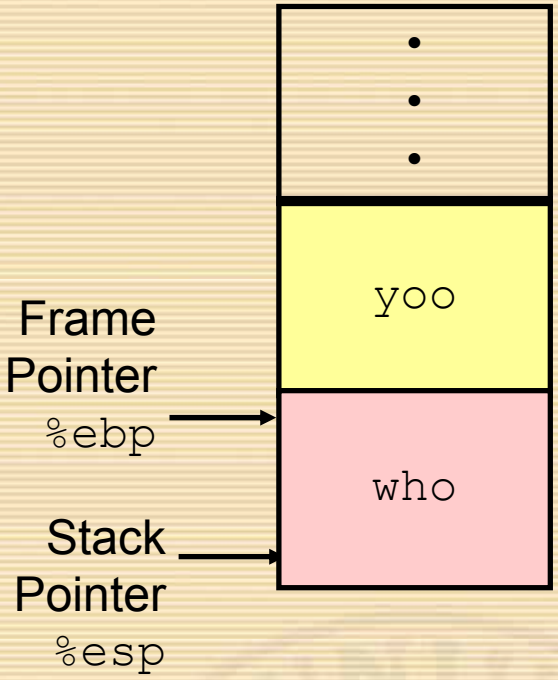
Call Chain



Stack Operation

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

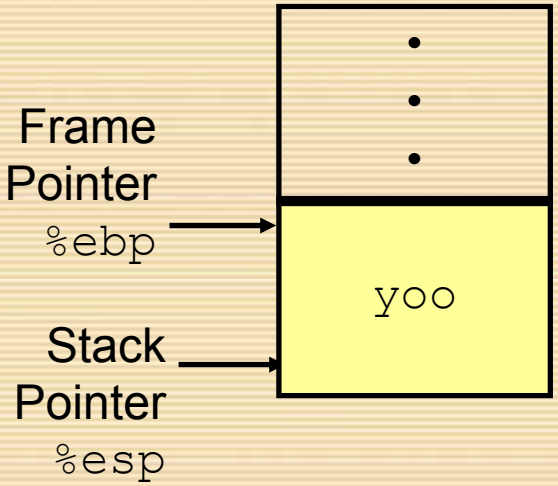
Call Chain



Stack Operation

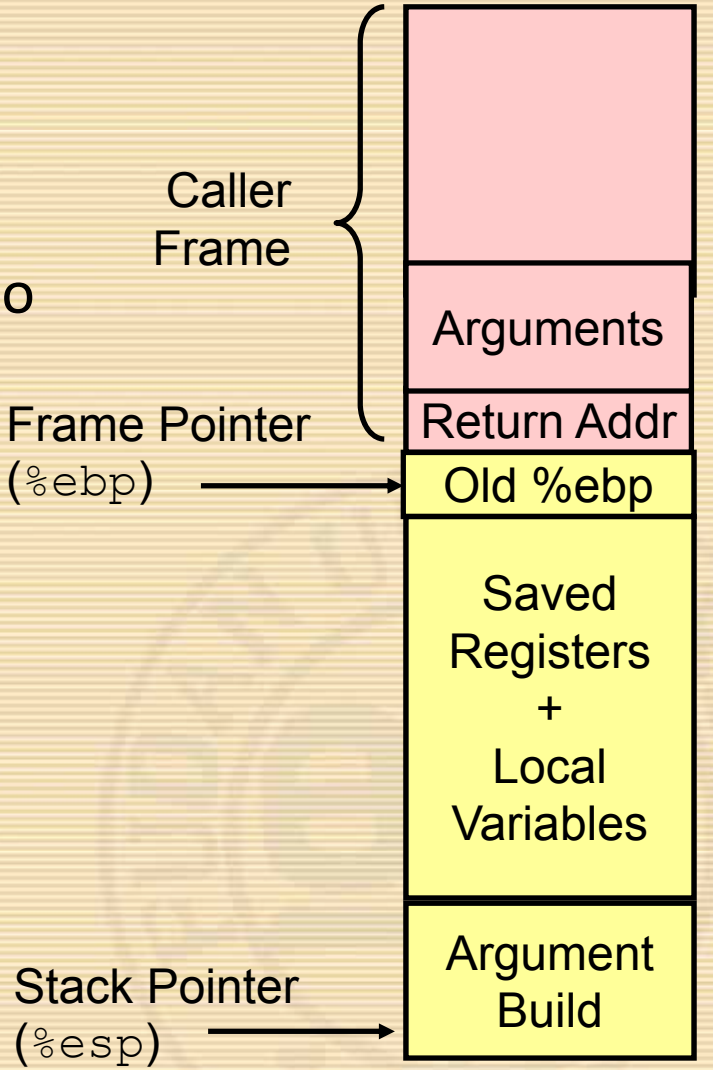
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

Call Chain



IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - Parameters for function about to call
    - "Argument build"
  - Local variables
    - If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



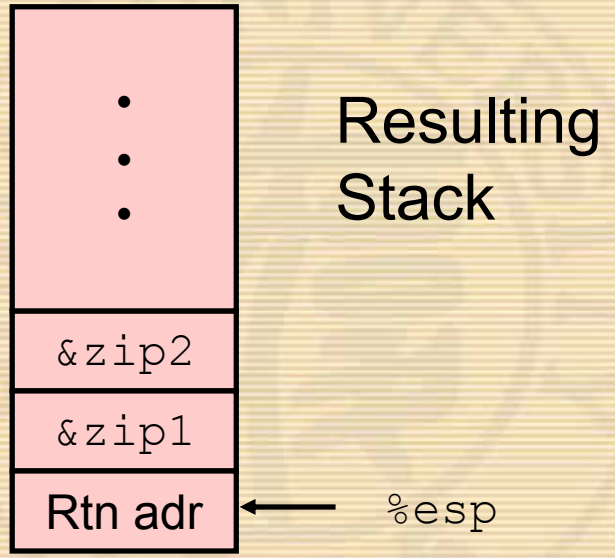
Revisiting swap

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

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

Calling swap from call_swap

```
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    . . .
```



Revisiting swap

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

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

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

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

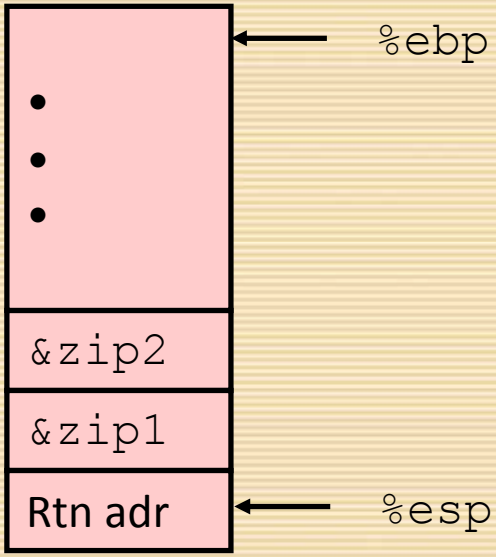
Set Up

Body

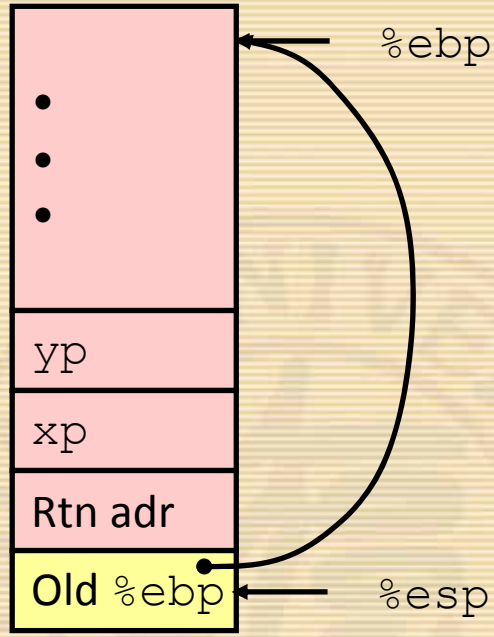
Finish

swap Setup #1

Entering Stack



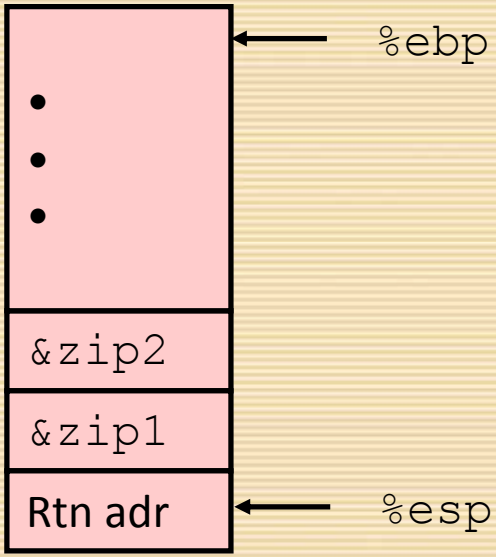
Resulting Stack



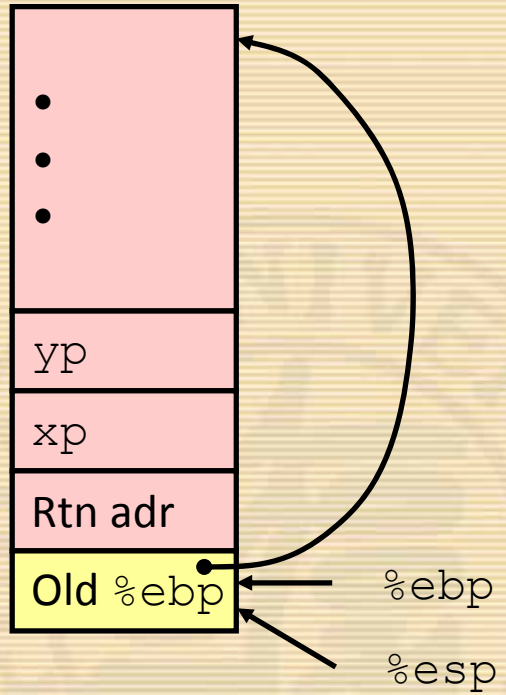

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



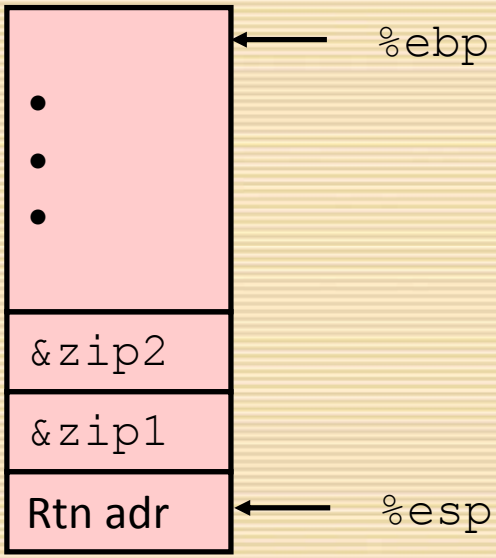
Resulting Stack



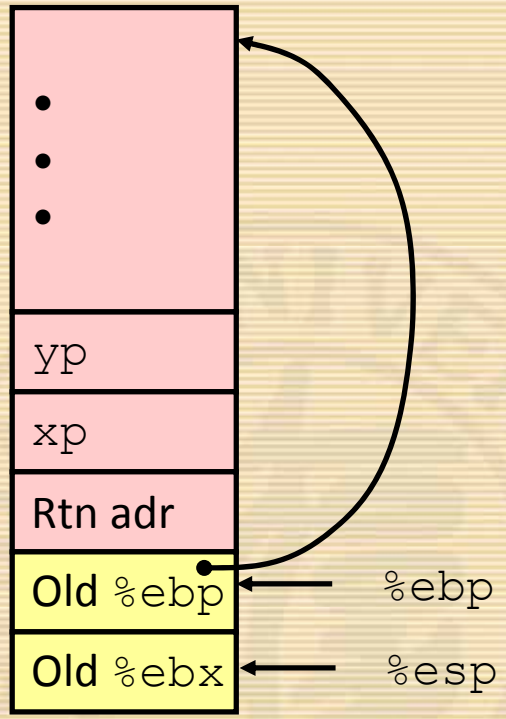

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

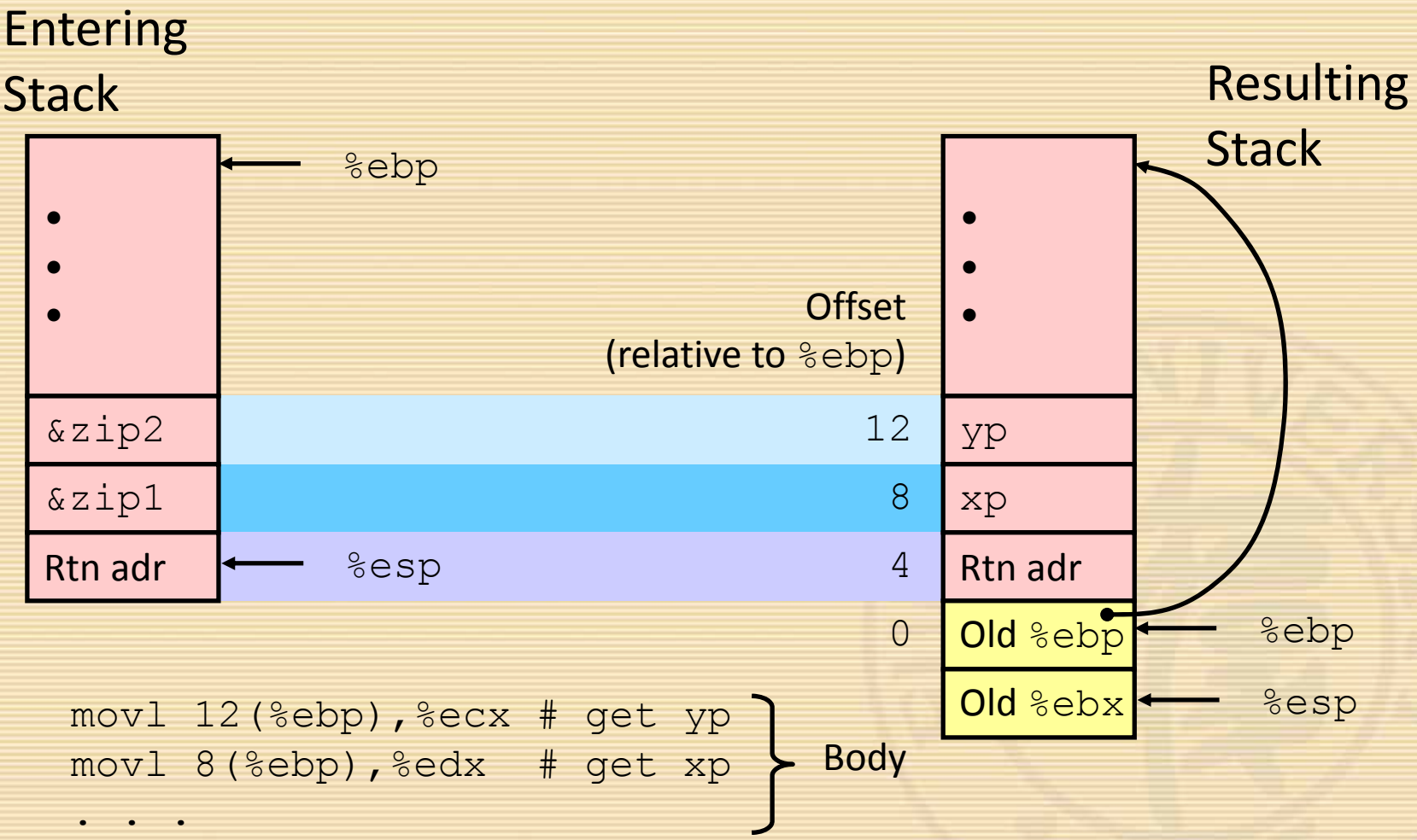


Resulting Stack

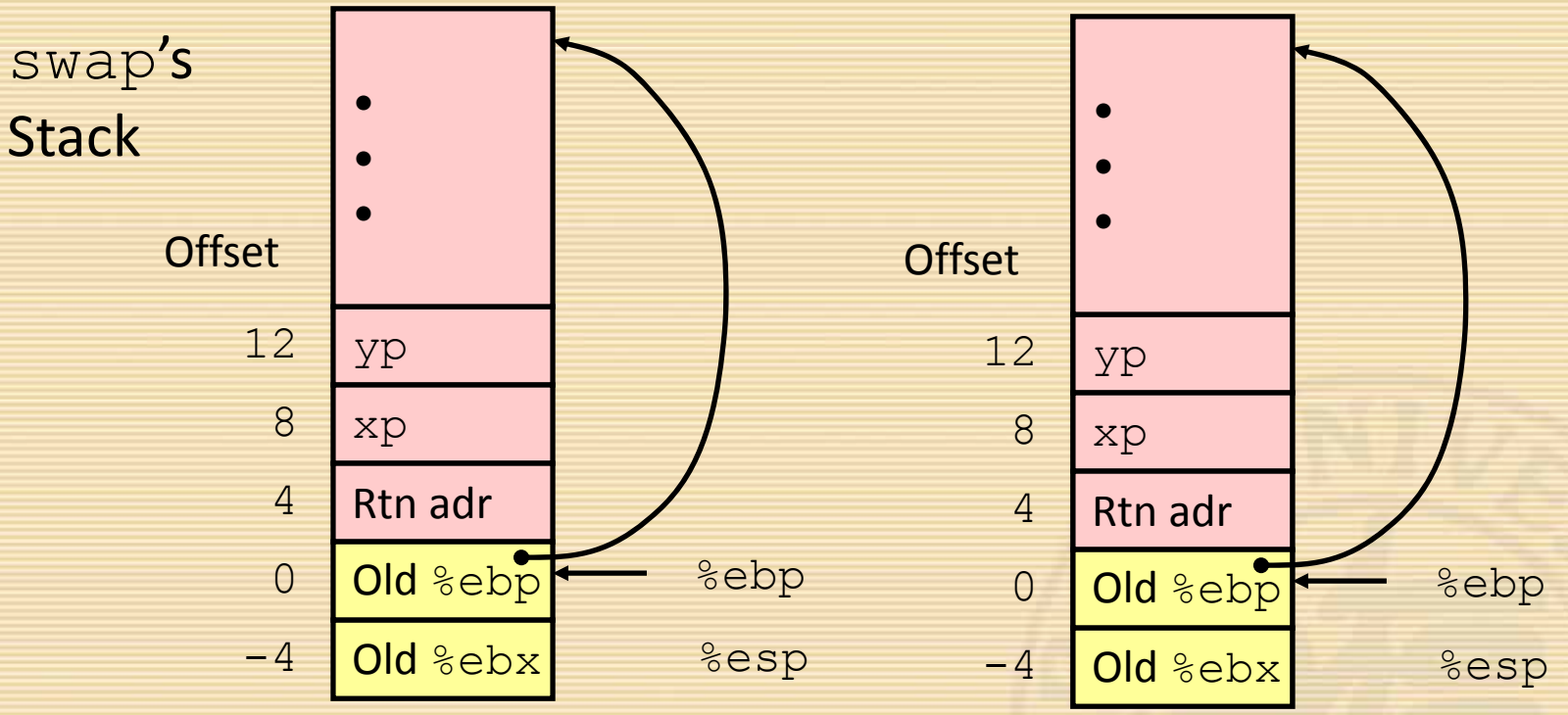



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Effect of `swap` Setup



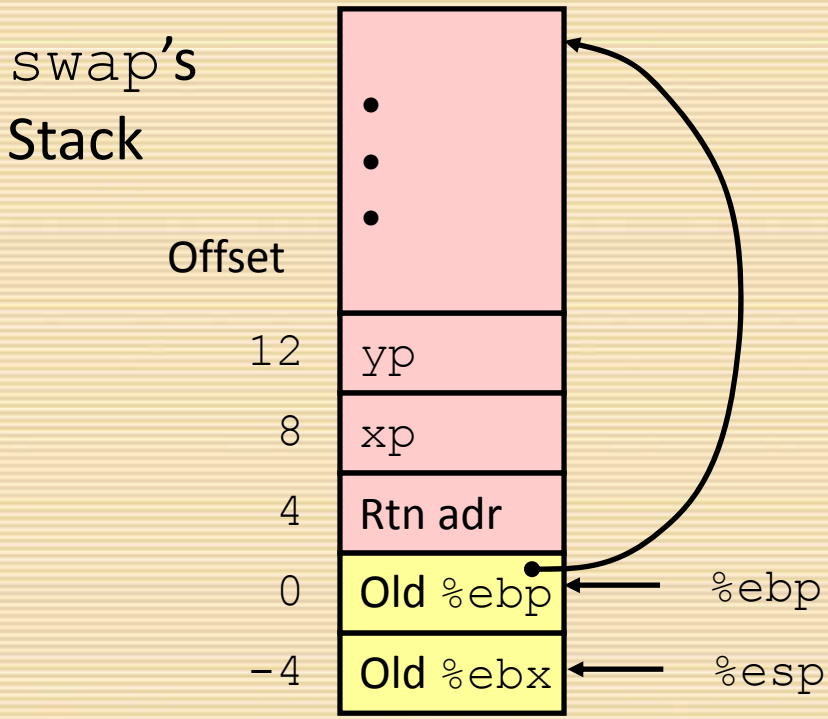
swap Finish #1

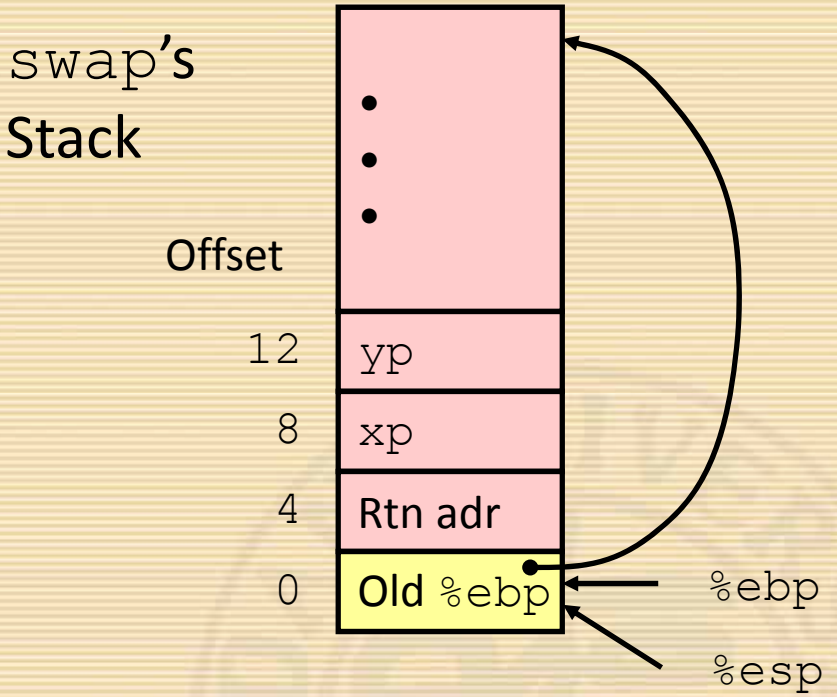



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- Observation
  - Saved & restored register %ebx

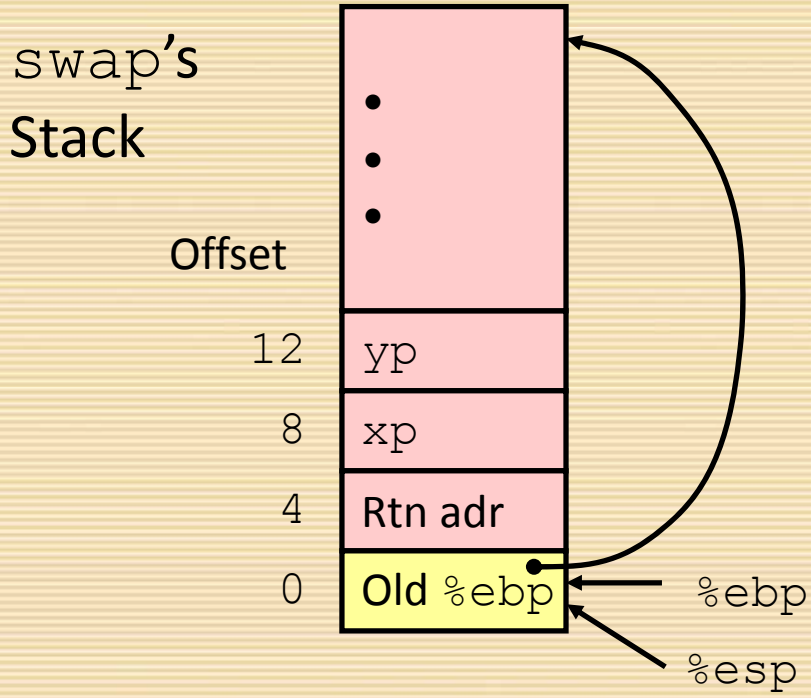
swap Finish #2

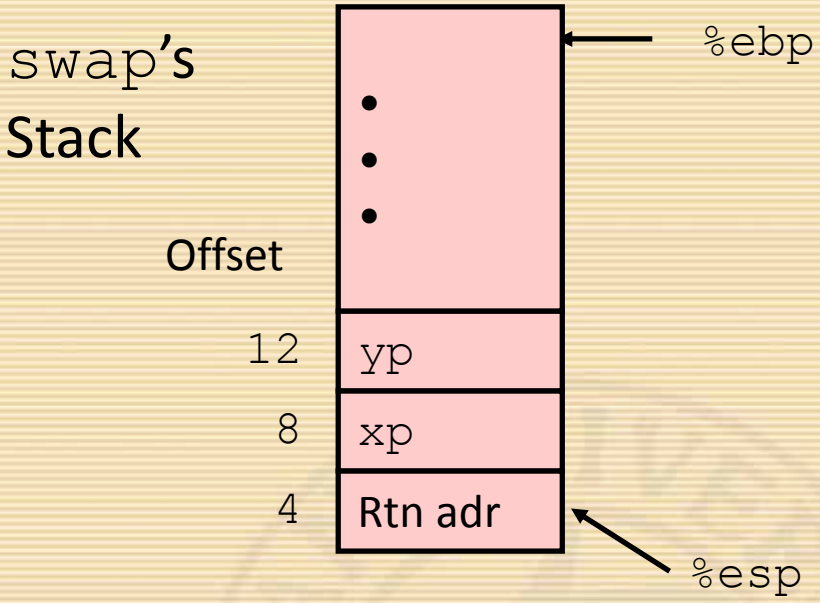





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

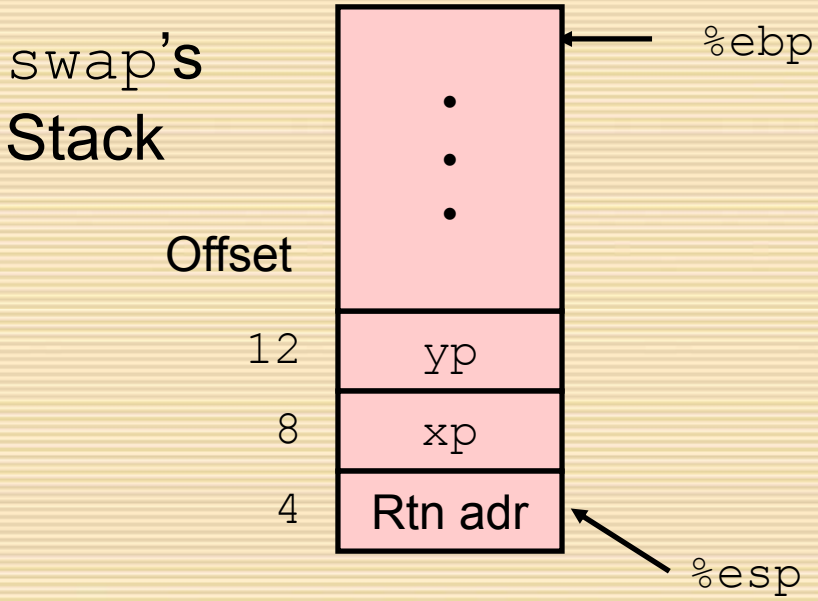
swap Finish #3

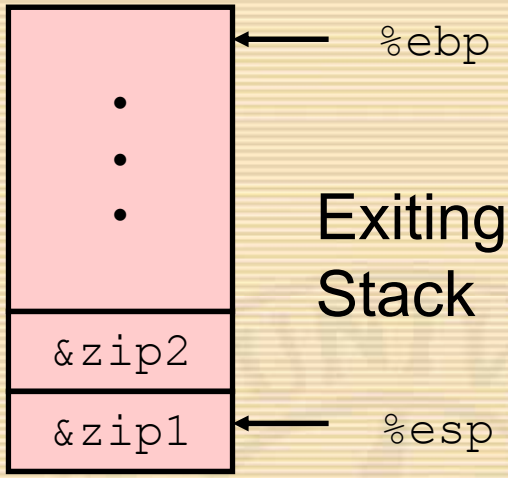





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- Observation
  - Saved & restored register `%ebx`
  - Didn't do so for `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*, `who` is the *callee*
- Can Register be Used for Temporary Storage?

```
yoo:
    . . .
    movl $15213, %edx
    call who
    addl %edx, %eax
    . . .
    ret
```

```
who:
    . . .
    movl 8(%ebp), %edx
    addl $91125, %edx
    . . .
    ret
```

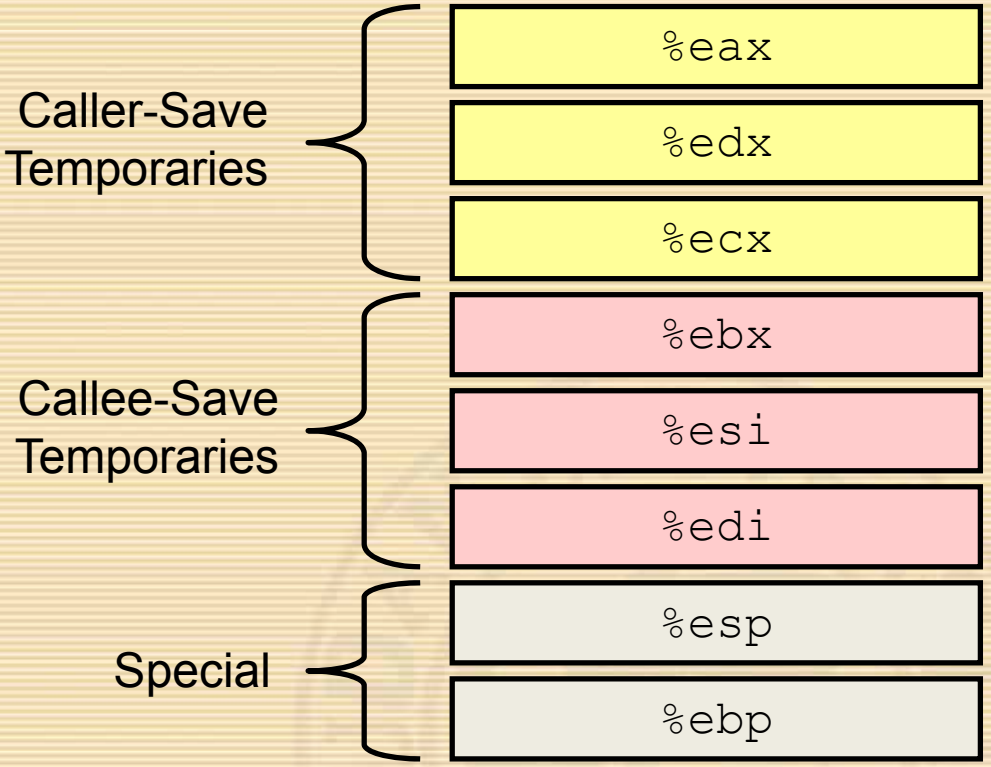
- Contents of register `%edx` overwritten by `who`

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 Save"
    - Caller saves temporary in its frame before calling
  - "Callee Save"
    - Callee saves temporary in its frame before using

IA32/Linux Register Usage

- Integer Registers
  - Two have special uses
    `%ebp`, `%esp`
  - Three managed as callee-save
    `%ebx`, `%esi`, `%edi`
    - Old values saved on stack prior to using
  - Three managed as caller-save
    `%eax`, `%edx`, `%ecx`
    - Do what you please, but expect any callee to do so, as well
  - Register `%eax` also stores returned value



Summary

- The Stack Makes Recursion Work
  - Private storage for each *instance* of procedure call
    - Instantiations don't clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Can be managed by stack discipline
    - Procedures return in inverse order of calls
- IA32 Procedures Combination of Instructions + Conventions
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - `%ebp` and `%esp`
  - Stack frame organization conventions

Machine-Level Programming IV: Structured Data

jc@fudan.edu.cn

Basic Data Types

- Integral
  - Stored & operated on in general registers
  - Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`

- Floating Point
  - Stored & operated on in floating point registers

Intel	GAS	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12	`long double`

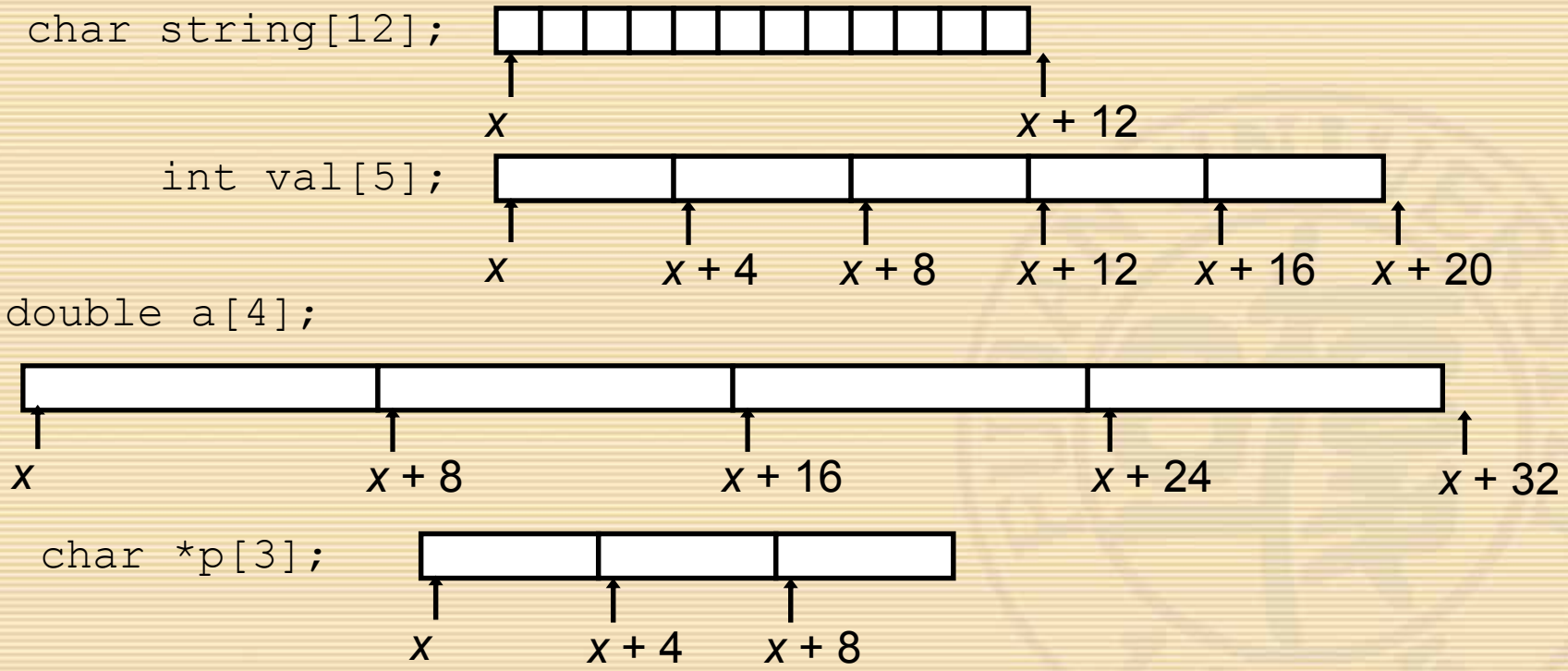
Array Allocation

- Basic Principle

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

  – Array of data type *T* and length *L*

  – Contiguously allocated region of *L* * `sizeof(`*T*`)` bytes



Array Access

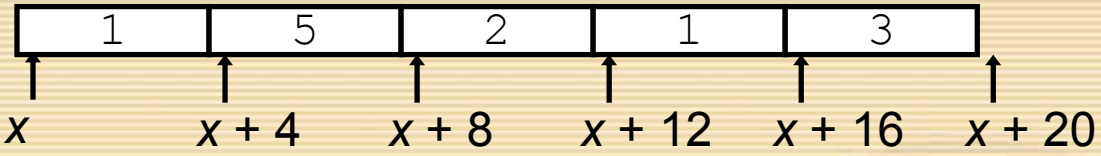
- Basic Principle

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

  – Array of data type *T* and length *L*

  – Identifier `A` can be used as a pointer to array element 0

`int val[5];`

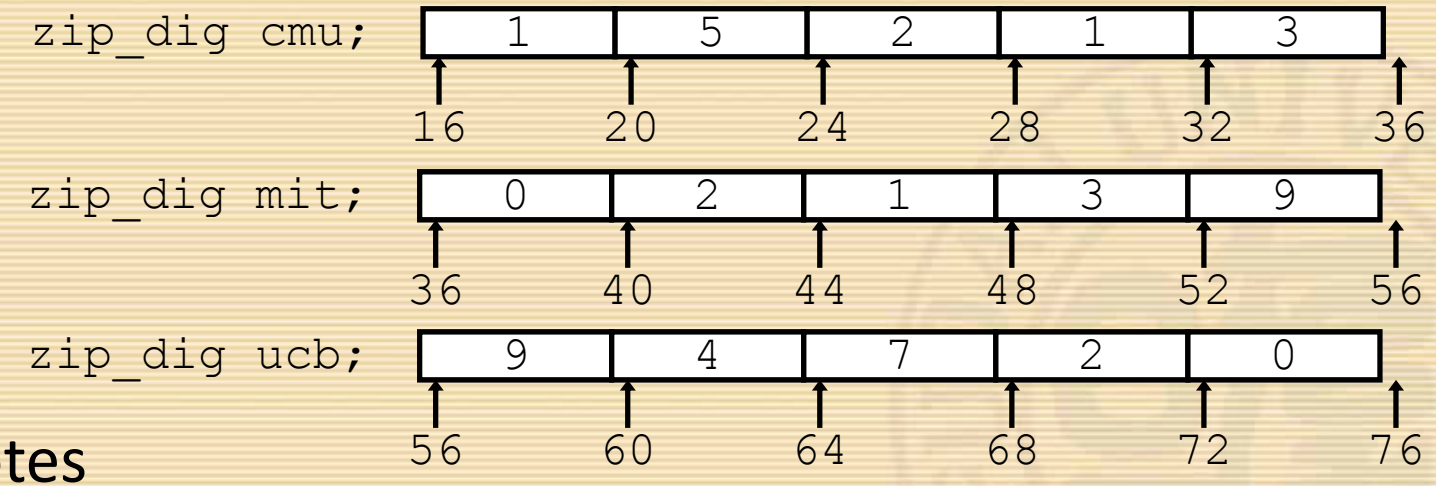


- Reference

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

Array Example

```
typedef int zip_dig[5];

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



- Notes
  - 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

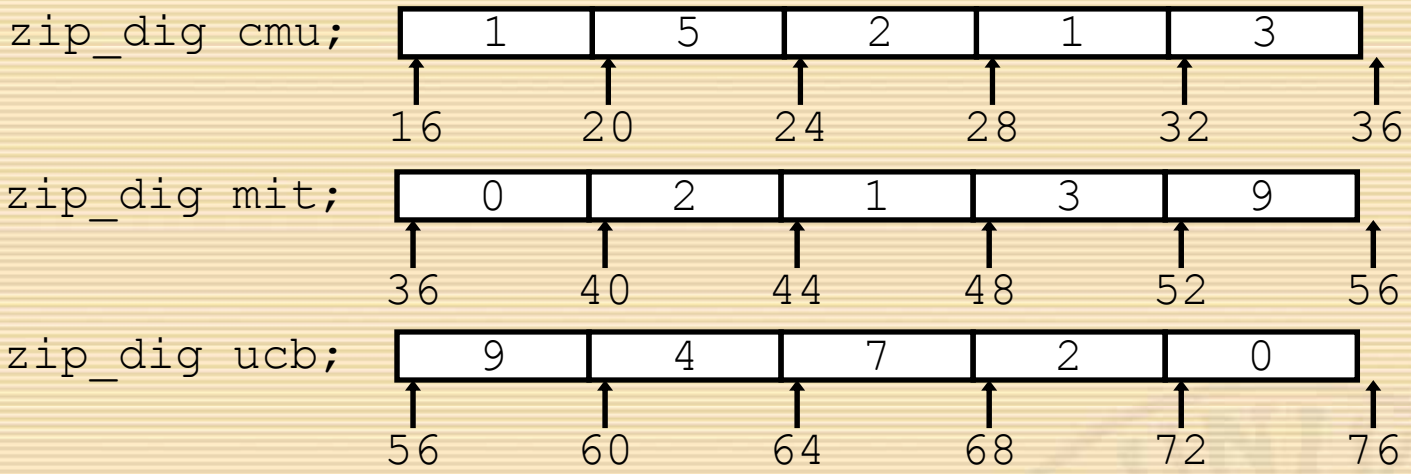
- Computation
  - Register `%edx` contains starting address of array
  - Register `%eax` contains array index
  - Desired digit at `4*%eax + %edx`
  - Use memory reference `(%edx,%eax,4)`

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

- Memory Reference Code

```
# %edx = z
# %eax = dig
 movl (%edx,%eax,4),%eax # z[dig]
```

Referencing Examples



- Code Does Not Do Any Bounds Checking!
- Reference Address Value Guaranteed?

`mit[3]`	`36 + 4* 3 = 48`	3	Yes
`mit[5]`	`36 + 4* 5 = 56`	9	No
`mit[-1]`	`36 + 4*-1 = 32`	3	No
`cmu[15]`	`16 + 4*15 = 76`	??	No

  - Out of range behavior implementation-dependent
    - No guaranteed relative allocation of different arrays

Array Loop Example

- Original Source

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- Transformed Version
  - As generated by GCC
  - Eliminate loop variable `i`
  - Convert array code to pointer code
  - Express in do-while form
    - No need to test at entrance

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

Array Loop Implementation

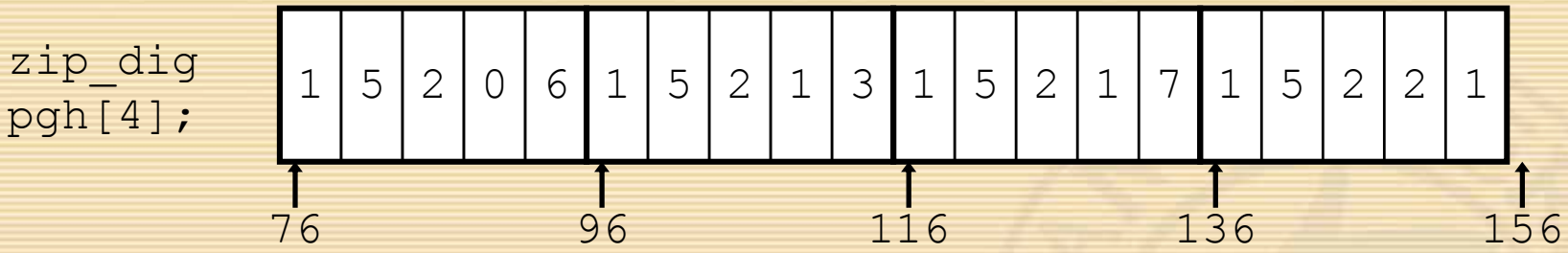
- Registers
  - %ecx z
  - %eax zi
  - %ebx zend
- Computations
  - 10*zi + *z implemented as *z + 2*(zi+4*zi)
  - z++ increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax          # zi = 0
  leal 16(%ecx),%ebx      # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # 5*zi
  movl (%ecx),%eax        # *z
  addl $4,%ecx            # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx          # z : zend
  jle .L59                # if <= goto loop
```

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 }};
```



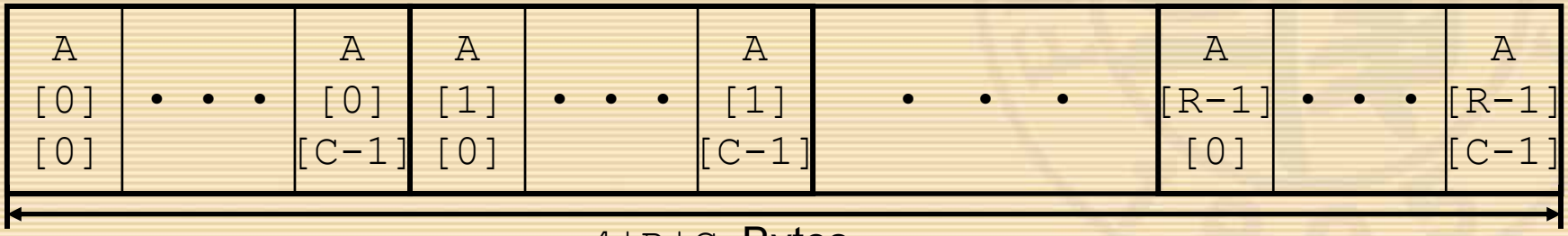
- Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable `pgh` denotes array of 4 elements
    - Allocated contiguously
  - Each element is an array of 5 `int`'s
    - Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Nested Array Allocation

- Declaration
  T `A[`*R*`][`*C*`];`
  - 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

```
A[0][0]   • • •   A[0][C-1]
   •                 •
   •                 •
   •                 •
A[R-1][0] • • •  A[R-1][C-1]
```

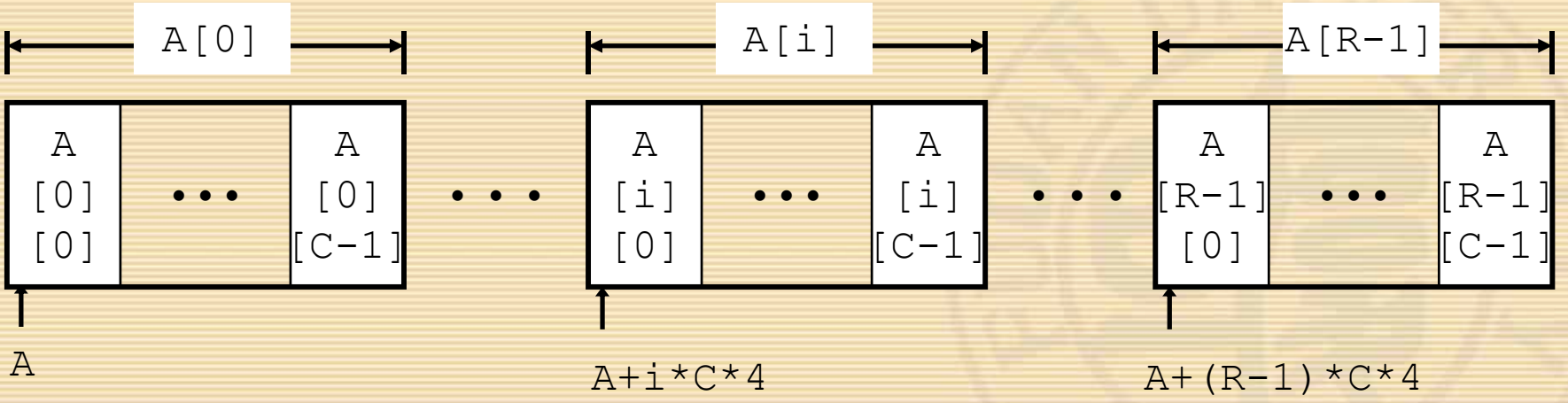
`int A[R][C];`



Nested Array Row Access

- Row Vectors
  - `A[i]` is array of *C* elements
  - Each element of type *T*
  - Starting address: `A` + *i* * *C* * *K*

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



Nested Array Row Access Code

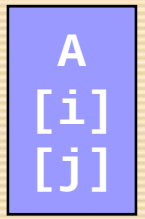
```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

- Row Vector
  - `pgh[index]` is array of 5 `int`'s
  - Starting address `pgh+20*index`
- Code
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

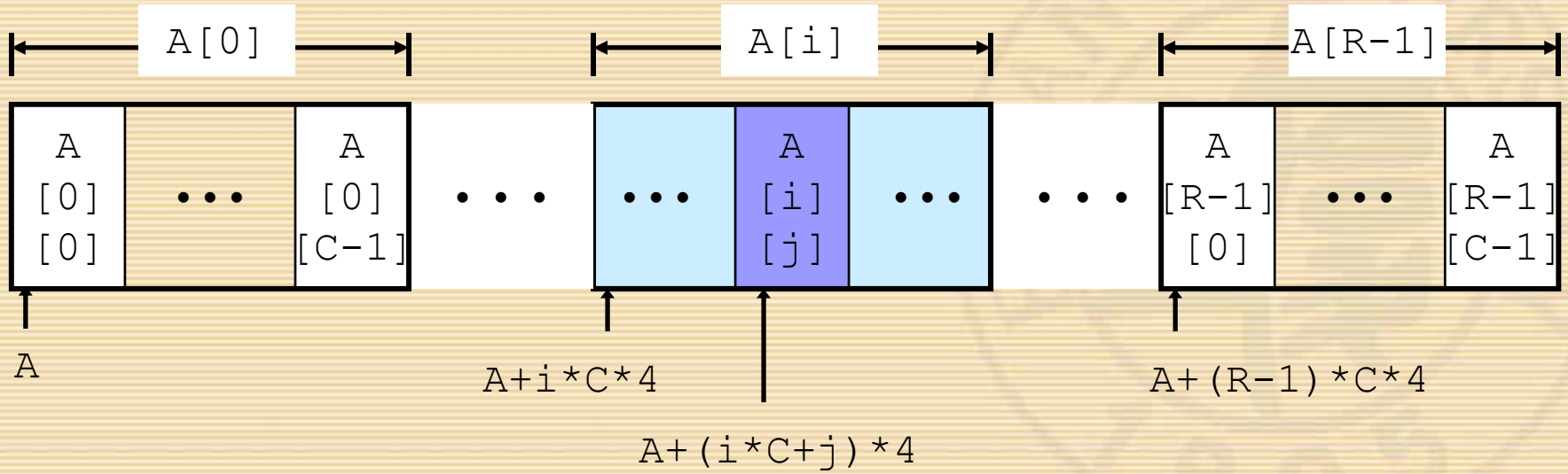
```
# %eax = index
 leal (%eax,%eax,4),%eax  # 5 * index
 leal pgh(,%eax,4),%eax   # pgh + (20 * index)
```

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type *T*
  - Address `A` + (*i* * *C* + *j*) * *K*




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



Nested Array Element Access Code

- Array Elements
  - `pgh[index][dig]` is `int`
  - Address:

    `pgh + 20*index + 4*dig`

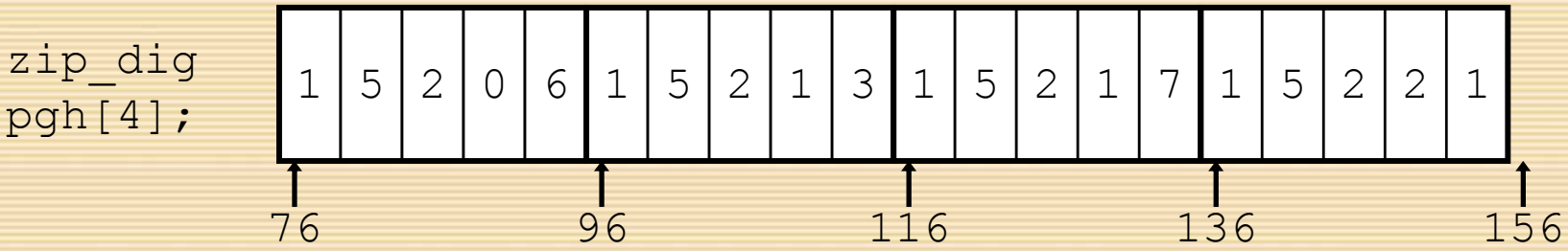
```
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```

- Code
  - Computes address

    `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (%eax,%eax,4),%eax       # 5*index
movl pgh(%edx,%eax,4),%eax    # *(pgh + 4*dig + 20*index)
```

Strange Referencing Examples



- Reference Address Value Guaranteed?

`pgh[3][3]`	`76+20*3+4*3 = 148`	2	Yes
`pgh[2][5]`	`76+20*2+4*5 = 136`	1	Yes
`pgh[2][-1]`	`76+20*2+4*-1 = 112`	3	Yes
`pgh[4][-1]`	`76+20*4+4*-1 = 152`	1	Yes
`pgh[0][19]`	`76+20*0+4*19 = 152`	1	Yes
`pgh[0][-1]`	`76+20*0+4*-1 = 72`	??	No

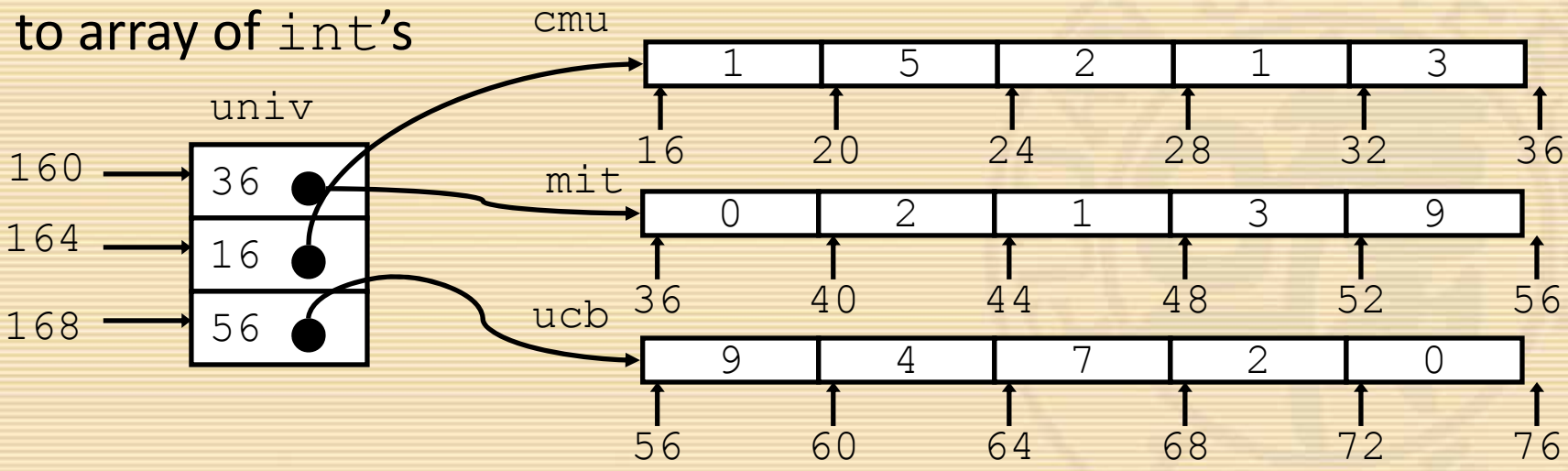
  - Code does not do any bounds checking
  - Ordering of elements within array guaranteed

Multi-Level Array Example

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

```
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};
```



Element Access in Multi-Level Array

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

- Computation
  - Element access

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

```
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx     # 4*index
movl univ(%edx),%edx     # Mem[univ+4*index]
movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

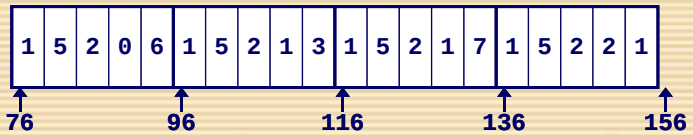
Array Element Accesses

- Nested Array
  - Similar C references

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

  - Element at

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

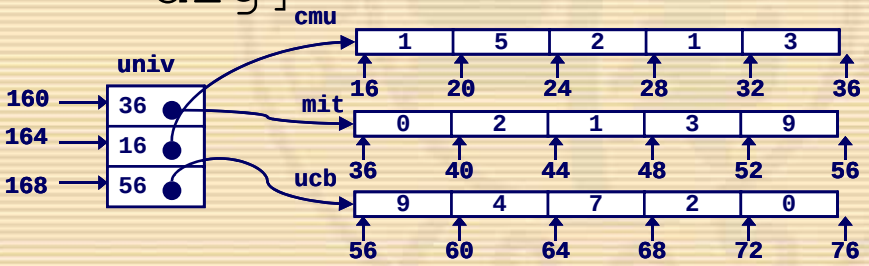


- Multi-Level Array
  - Different address computation

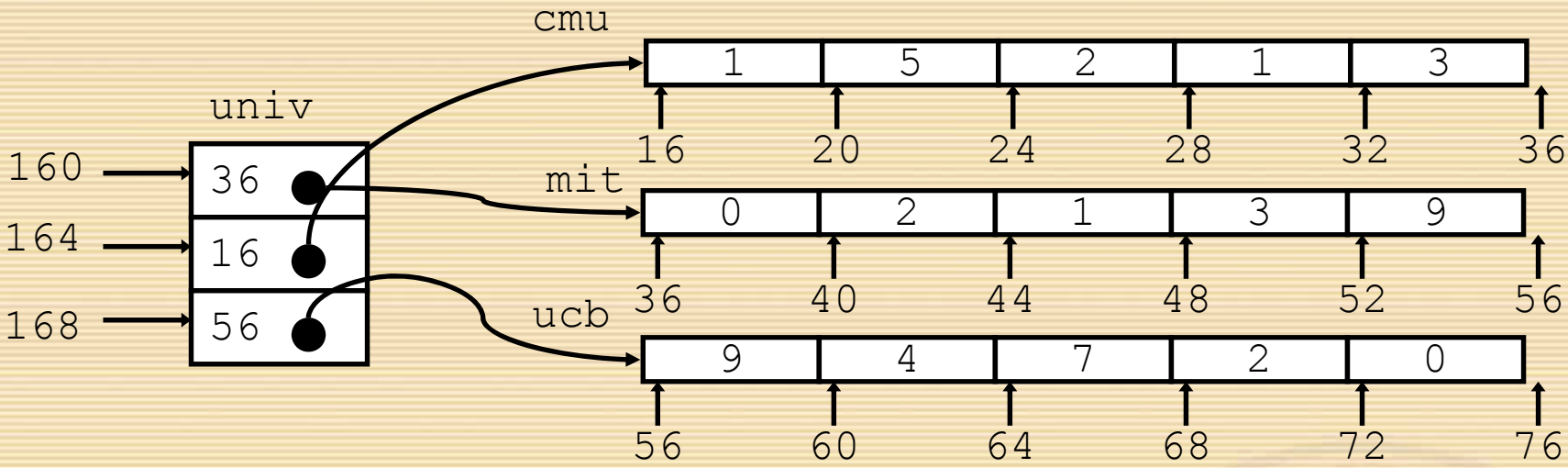
```
int get_univ_digit
   (int index, int dig)
{
   return univ[index][dig];
}
```

  - Element at

```
Mem[Mem[univ+4*index]+4
   *dig]
```



Strange Referencing Examples

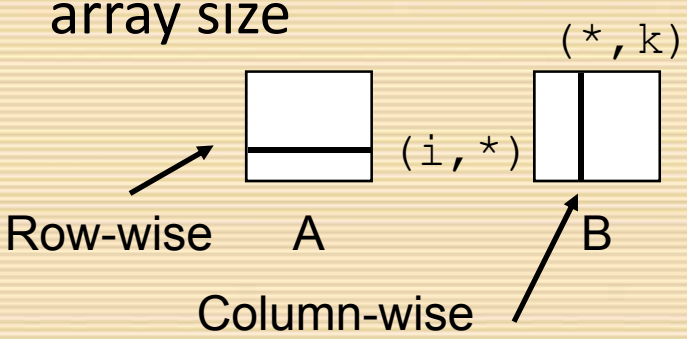


Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	2	Yes
`univ[1][5]`	`16+4*5 = 36`	0	No
`univ[2][-1]`	`56+4*-1 = 52`	9	No
`univ[3][-1]`	`??`	`??`	No
`univ[1][12]`	`16+4*12 = 64`	7	No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

Using Nested Arrays

- Strengths
  - C compiler handles doubly subscripted arrays
  - Generates very efficient code
    - Avoids multiply in index computation
- Limitation
  - Only works if have fixed array size




```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```

Dynamic Nested Arrays

- Strength
  - Can create matrix of arbitrary size
- Programming
  - Must do index computation explicitly
- Performance
  - Accessing single element costly
  - Must do multiplication

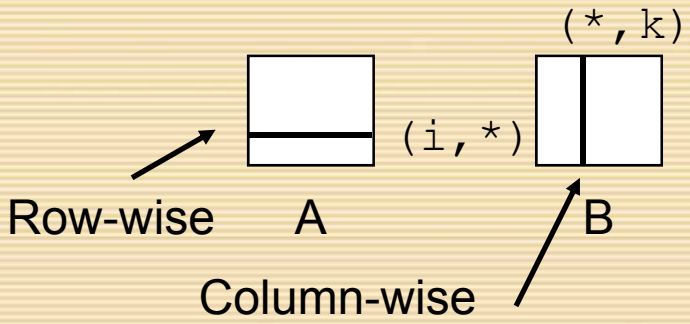
```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
int var_ele
  (int *a, int i,
   int j, int n)
{
  return a[i*n+j];
}
```

```
movl 12(%ebp),%eax      # i
movl 8(%ebp),%edx       # a
imull 20(%ebp),%eax     # n*i
addl 16(%ebp),%eax      # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

Dynamic Array Multiplication

- Without Optimizations
  - Multiplies
    - 2 for subscripts
    - 1 for data
  - Adds
    - 4 for array indexing
    - 1 for loop index
    - 1 for data




```
/* Compute element i,k of
   variable matrix product */
int var_prod_ele
  (int *a, int *b,
   int i, int k, int n)
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```

Optimizing Dynamic Array Mult.

- Optimizations
  - Performed when set optimization level to `-O2`
- Code Motion
  - Expression `i*n` can be computed outside loop
- Strength Reduction
  - Incrementing `j` has effect of incrementing `j*n+k` by `n`
- Performance
  - Compiler can optimize regular access patterns

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

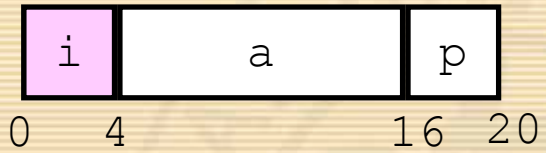
```
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

Structures

- Concept
  - Continuously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

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

Memory Layout



- Accessing Structure Member

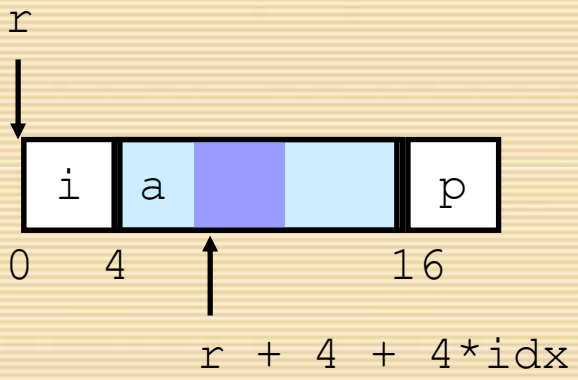
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Struct Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

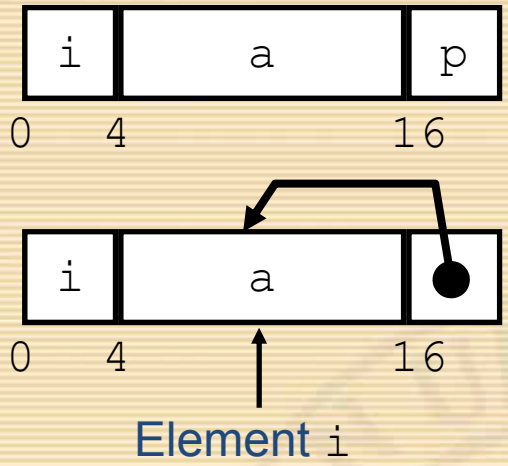
```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```

Structure Referencing (Cont.)

- C Code

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




```
void
set_p(struct rec *r)
{
  r->p =
    &r->a[r->i];
}
```

```
 # %edx = r
movl  (%edx),%ecx          # r->i
leal  0(,%ecx,4),%eax      # 4*(r->i)
leal  4(%edx,%eax),%eax    # r+4+4*(r->i)
movl  %eax,16(%edx)        # Update r->p
```

Alignment

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    - treated differently by Linux and Windows!
- Motivation for Aligning Data
  - Memory accessed by (aligned) double or quad-words
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

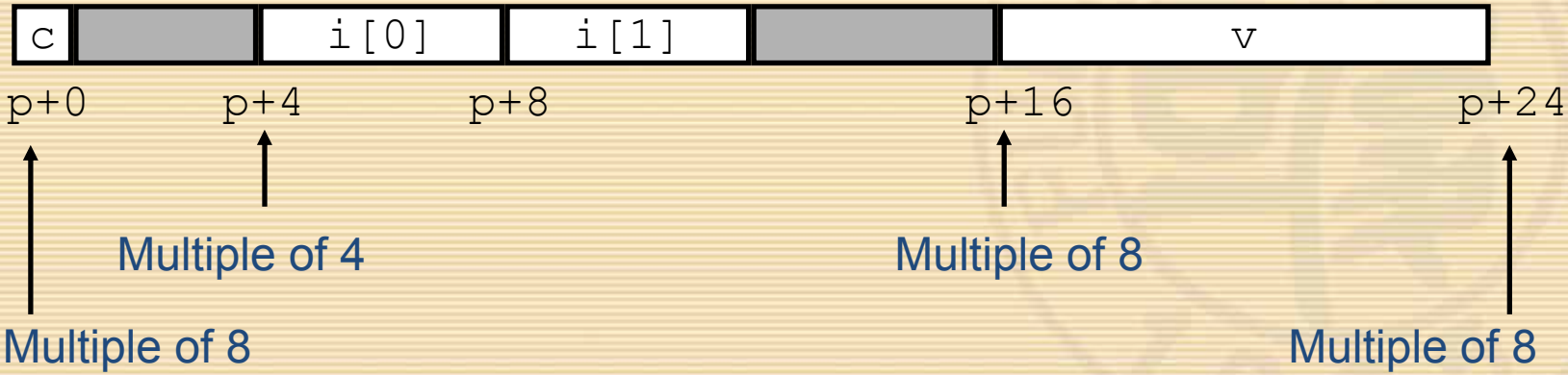
Specific Cases of Alignment

- Size of Primitive Data Type:
  - 1 byte (e.g., `char`)
    - no restrictions on address
  - 2 bytes (e.g., `short`)
    - lowest 1 bit of address must be 0_2
  - 4 bytes (e.g., `int`, `float`, `char *`, etc.)
    - lowest 2 bits of address must be 00_2
  - 8 bytes (e.g., `double`)
    - Windows (and most other OS's & instruction sets):
      - lowest 3 bits of address must be 000_2
    - Linux:
      - lowest 2 bits of address must be 00_2
      - i.e., treated the same as a 4-byte primitive data type
  - 12 bytes (`long double`)
    - Linux:
      - lowest 2 bits of address must be 00_2
      - i.e., treated the same as a 4-byte primitive data type

Satisfying Alignment with Structures

- Offsets Within Structure
  - Must satisfy element's alignment requirement
- Overall Structure Placement
  - Each structure has alignment requirement K
    - Largest alignment of any element
  - Initial address & structure length must be multiples of K
- Example (under Windows):
  - K = 8, due to `double` element

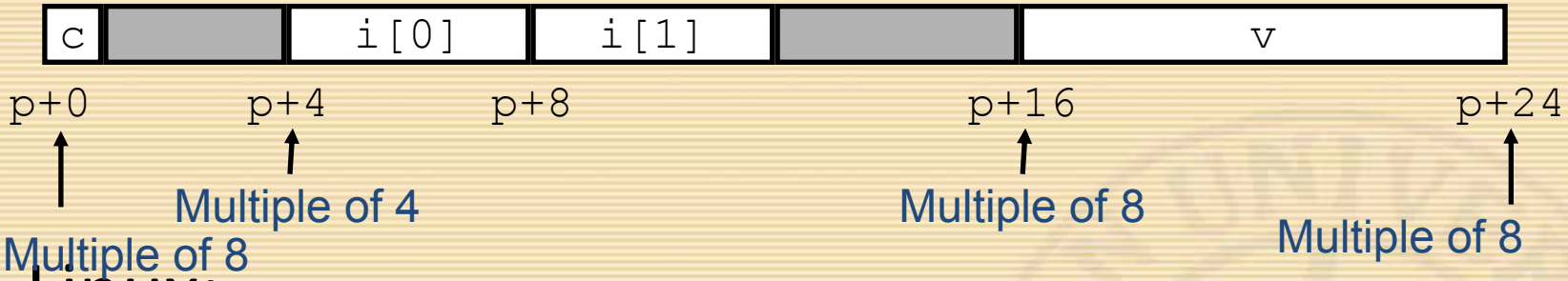
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



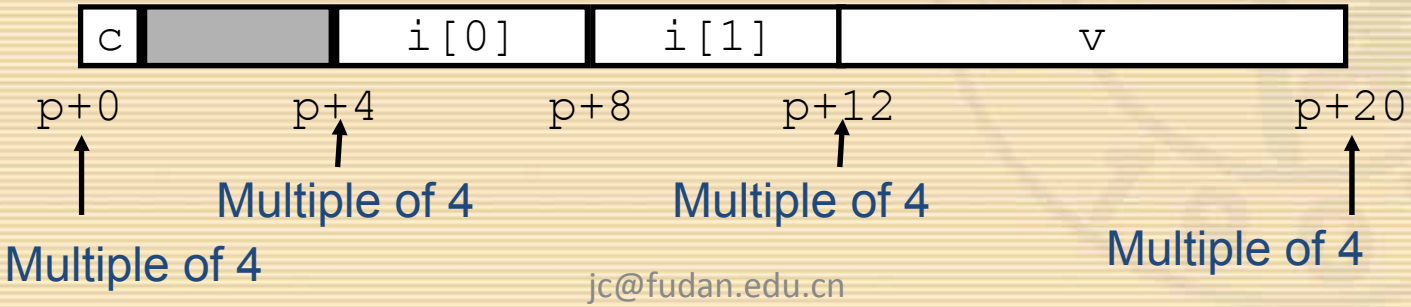
Linux vs Windows

- Windows (including Cygwin):
  - K = 8, due to `double` element

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



- Linux:
  - K = 4; `double` treated like a 4-byte data type



Overall Alignment Requirement

```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

p must be multiple of:
- 8 for Windows
- 4 for Linux

p+0 p+8 p+12 p+16 Windows: p+24 Linux: p+20

```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

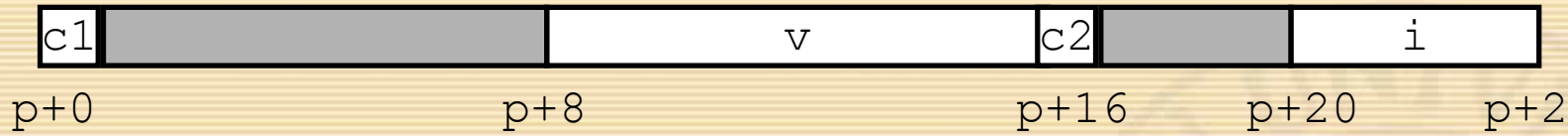
p must be multiple of 4 (in either OS)

p+0 p+4 p+8 p+12 p+16 p+20

Ordering Elements Within Structure

```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows

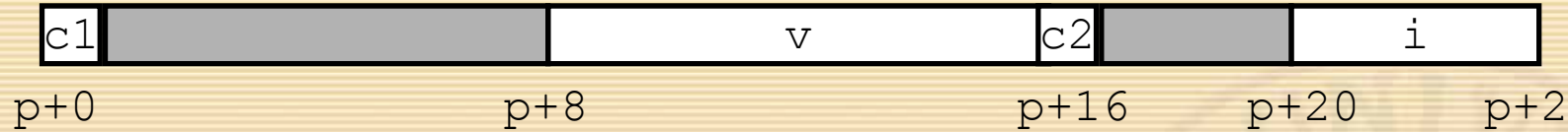


Any way to improve it?

Ordering Elements Within Structure

```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows



p+0 p+8 p+16 p+20 p+24

```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

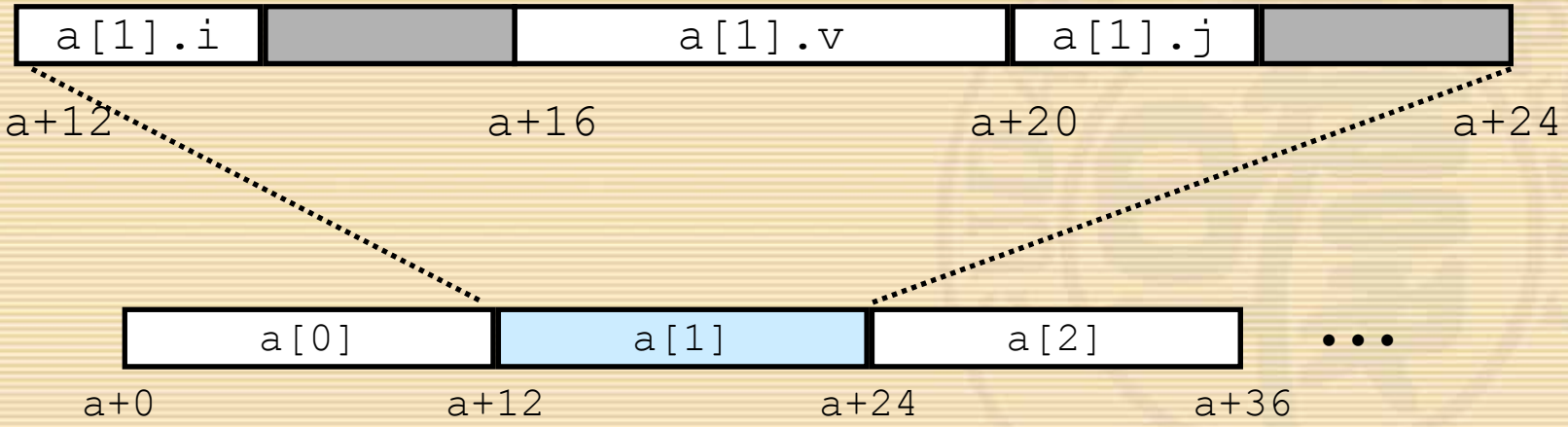
2 bytes wasted space

p+0 p+8 p+12 p+16

Arrays of Structures

- Principle
  - Allocated by repeating allocation for array type
  - In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



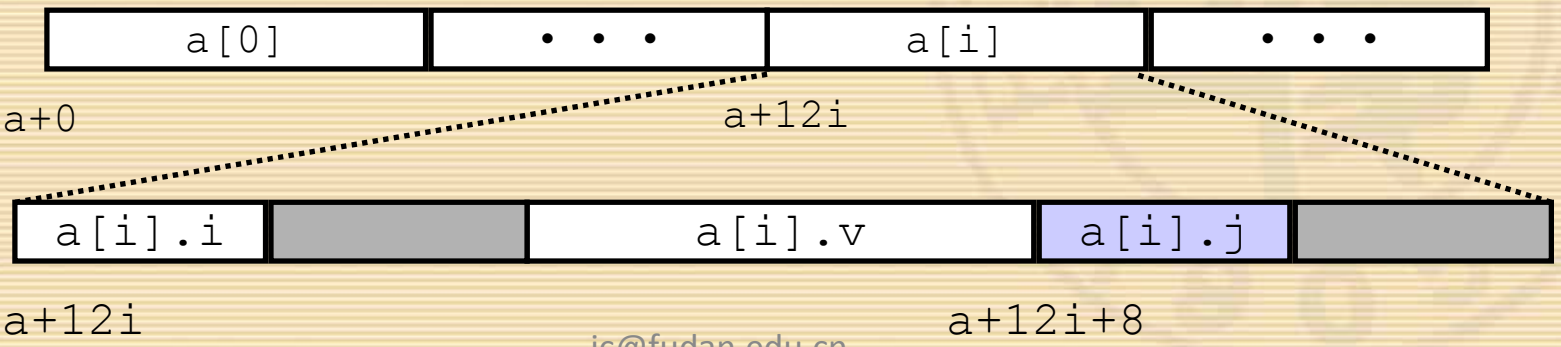
Accessing Element within Array

- Compute offset to start of structure
  - Compute 12*i as 4*(i+2i)
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as a + 8
    - Linker must set actual value

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

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

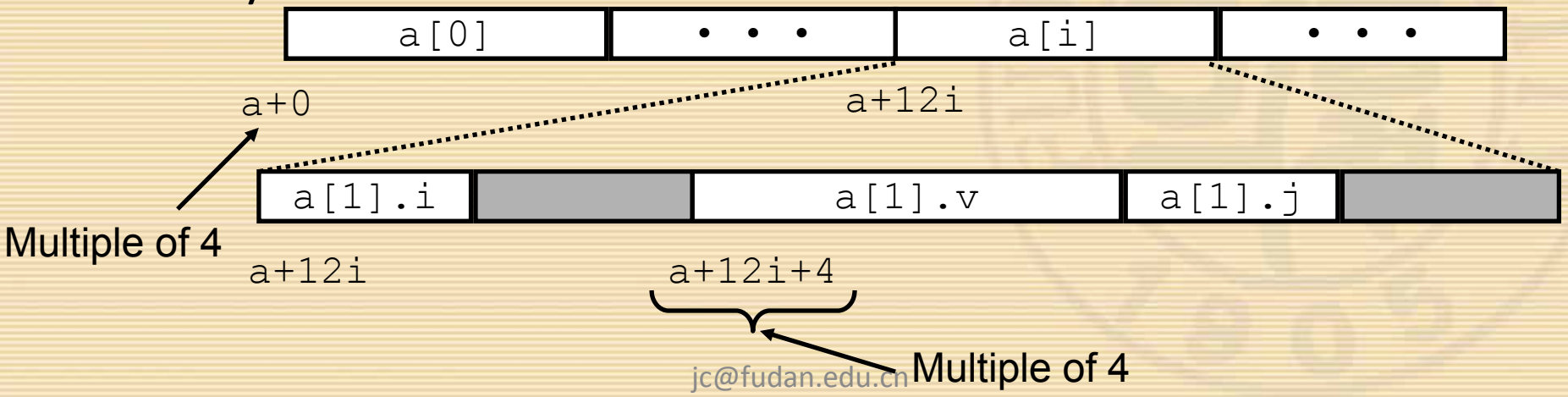
```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```



Satisfying Alignment within Structure

- Achieving Alignment
  - Starting address of structure array must be multiple of worst-case alignment for any element
    - `a` must be multiple of 4
  - Offset of element within structure must be multiple of element's alignment requirement
    - `v`'s offset of 4 is a multiple of 4
  - Overall size of structure must be multiple of worst-case alignment for any element
    - Structure padded with unused space to be 12 bytes

```
struct S6 {
   short i;
   float v;
   short j;
} a[10];
```

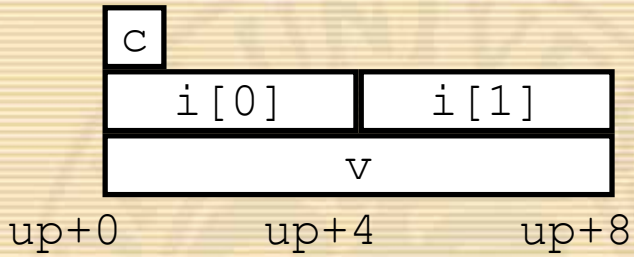


Union Allocation

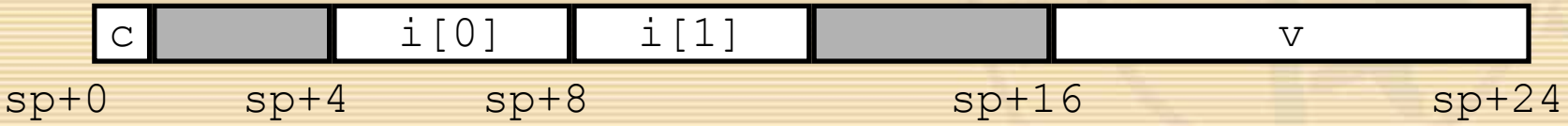
- Principles
  - Overlay union elements
  - Allocate according to largest element
  - Can only use one field at a time

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

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

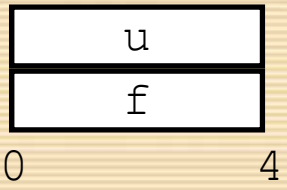


(Windows alignment)



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`

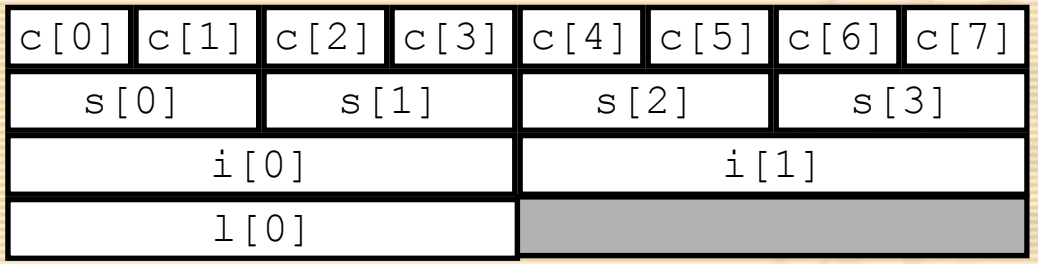
```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - PowerPC, Sparc
- Little Endian
  - Least significant byte has lowest address
  - Intel x86, Alpha

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

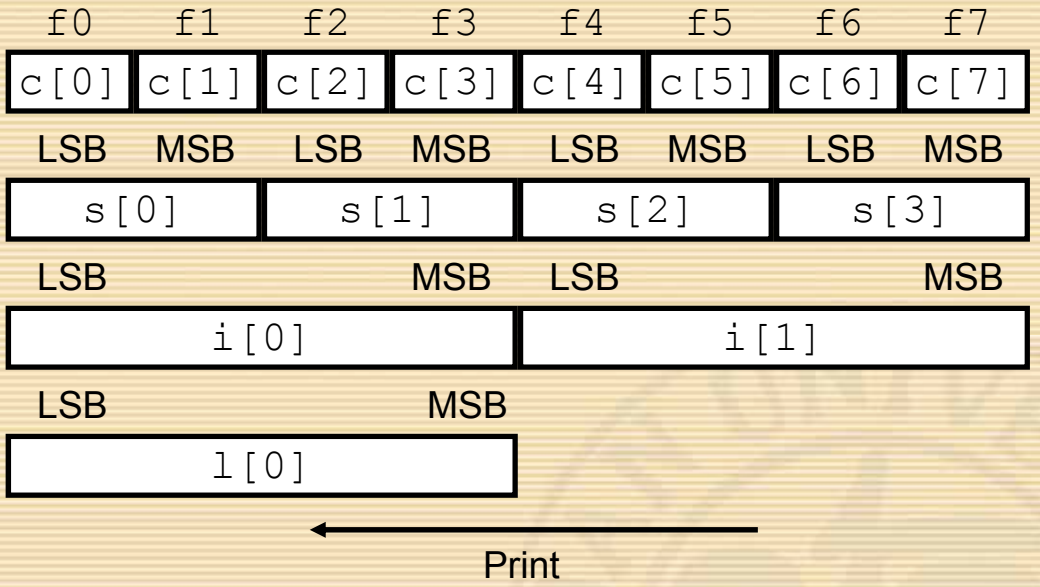


Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on x86

Little Endian

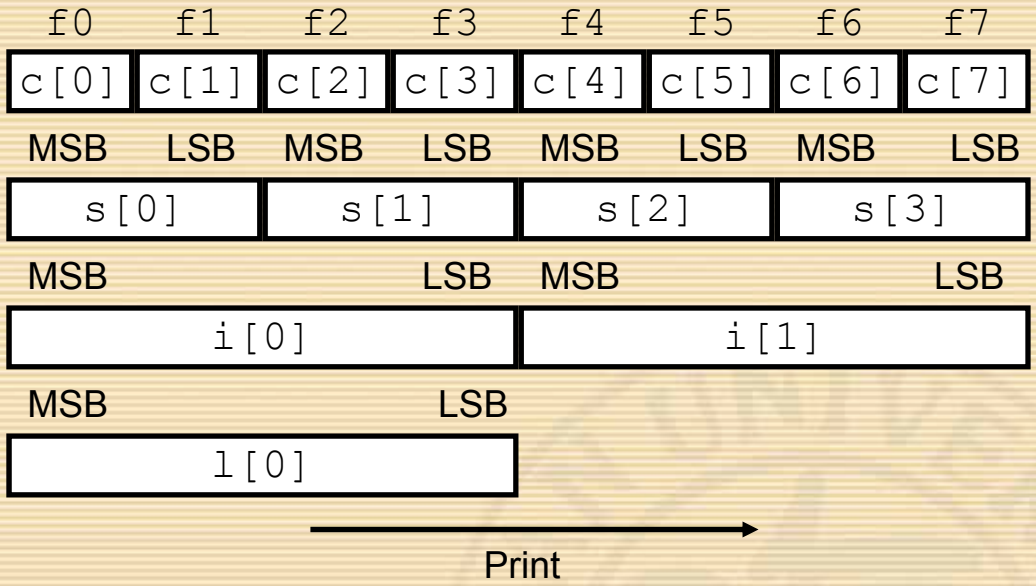


Output on Pentium:

```
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   == [f3f2f1f0]
```

Byte Ordering on Sun

Big Endian

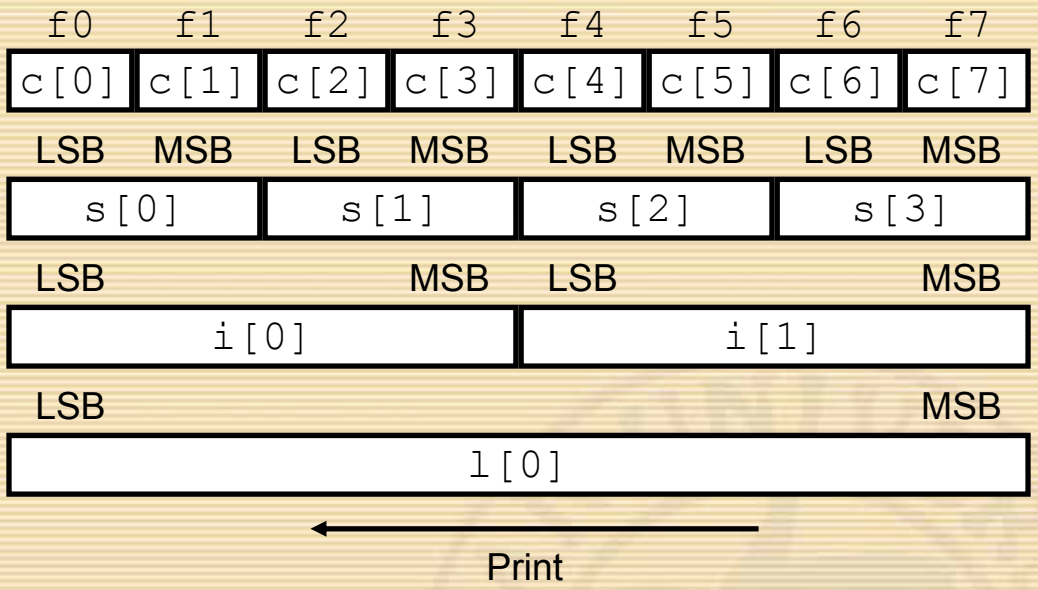


Output on Sun:

```
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]
```

Byte Ordering on Alpha

Little Endian

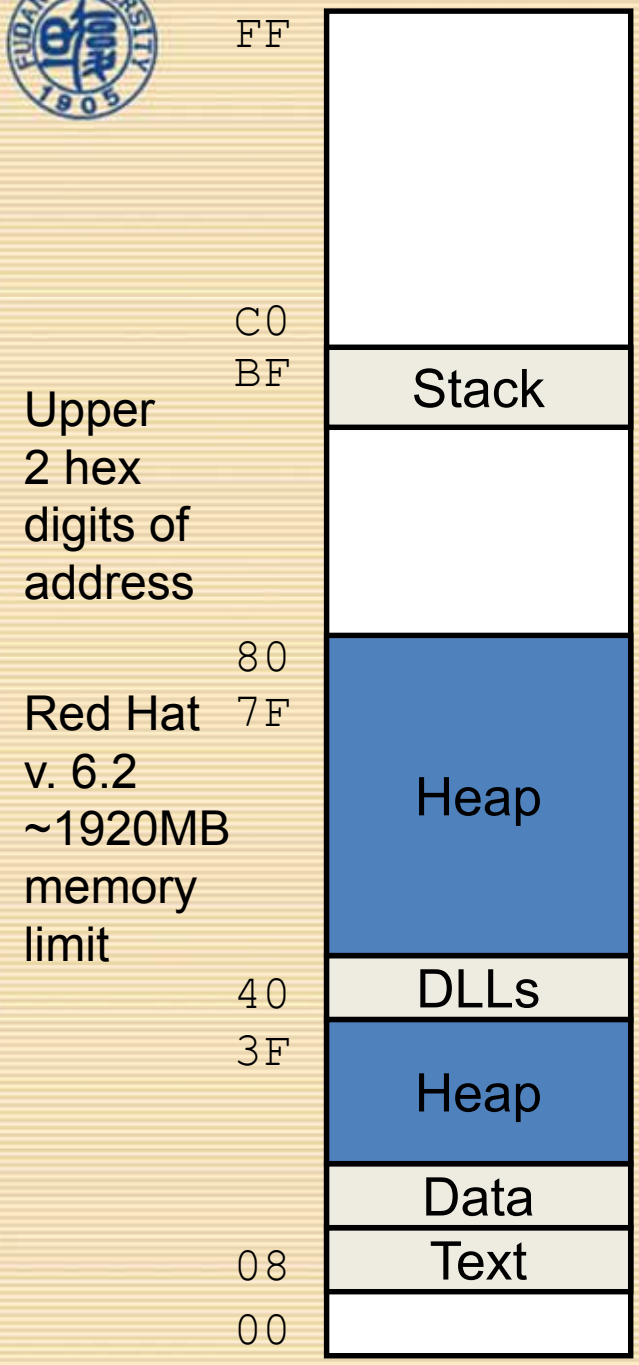


Output on Alpha:

```
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   == [0xf7f6f5f4f3f2f1f0]
```

Summary

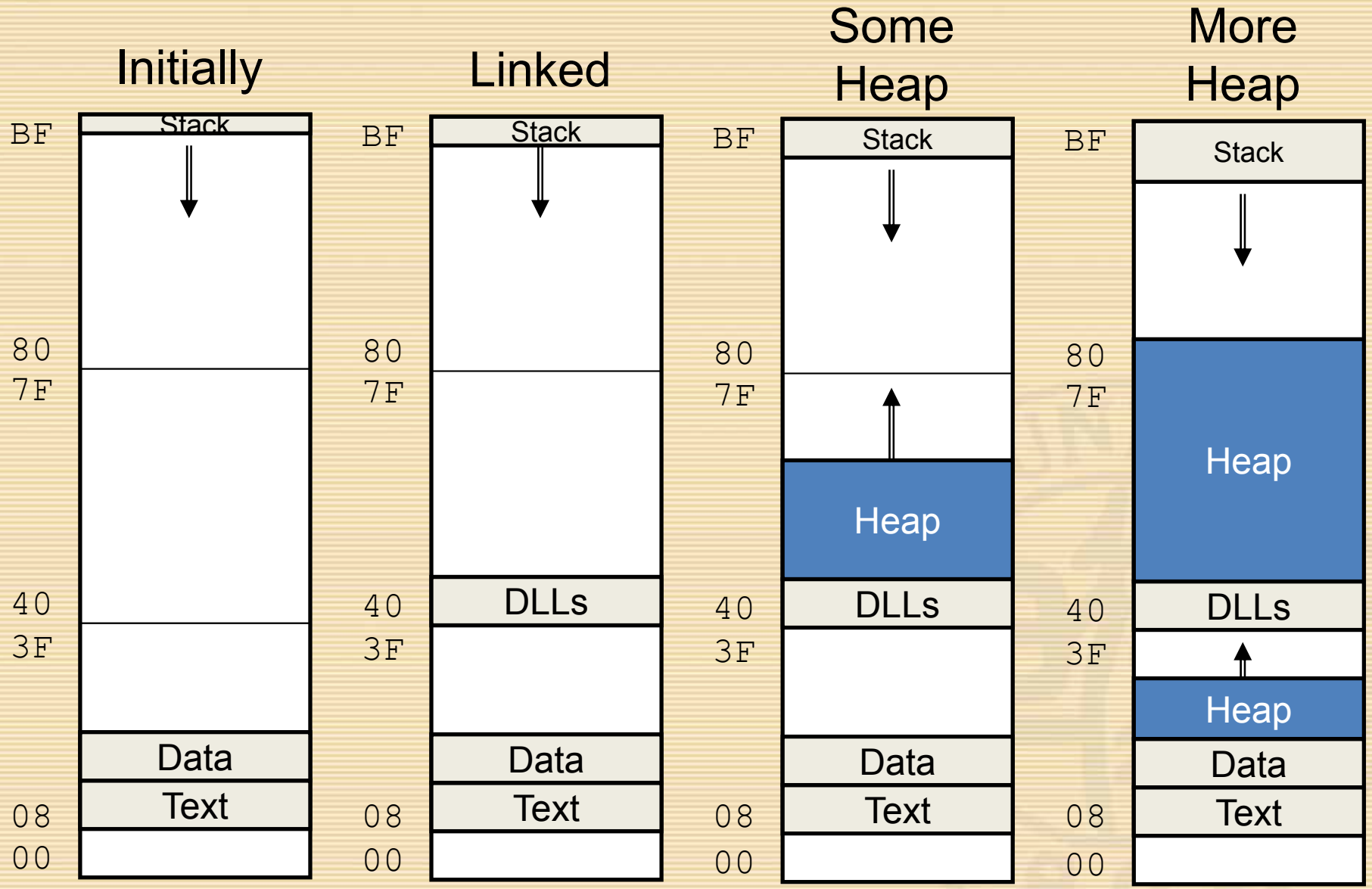
- Arrays in C
  - Contiguous allocation of memory
  - Pointer to first element
  - No bounds checking
- Compiler Optimizations
  - Compiler often turns array code into pointer code (zd2int)
  - Uses addressing modes to scale array indices
  - Lots of tricks to improve array indexing in loops
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system



Linux Memory Layout (Opt.)

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated storage
  - When call `malloc`, `calloc`, `new`
- DLLs
  - Dynamically Linked Libraries
  - Library routines (e.g., `printf`, `malloc`)
  - Linked into object code when first executed
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only

Linux Memory Allocation



String Library Code

- Implementation of Unix function `gets`
  - No way to specify limit on number of characters to read

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

- Similar problems with other Unix functions
  - `strcpy`: Copies string 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);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

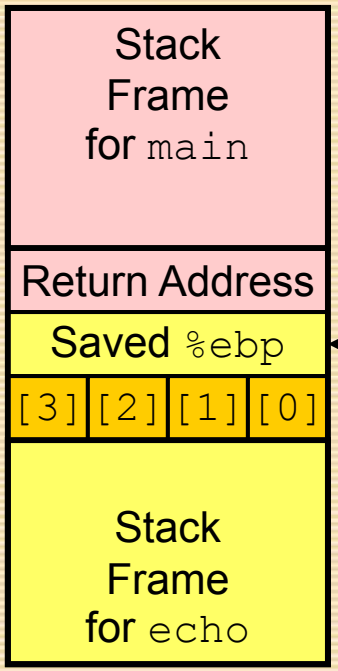
Buffer Overflow Executions

```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

Buffer Overflow Stack



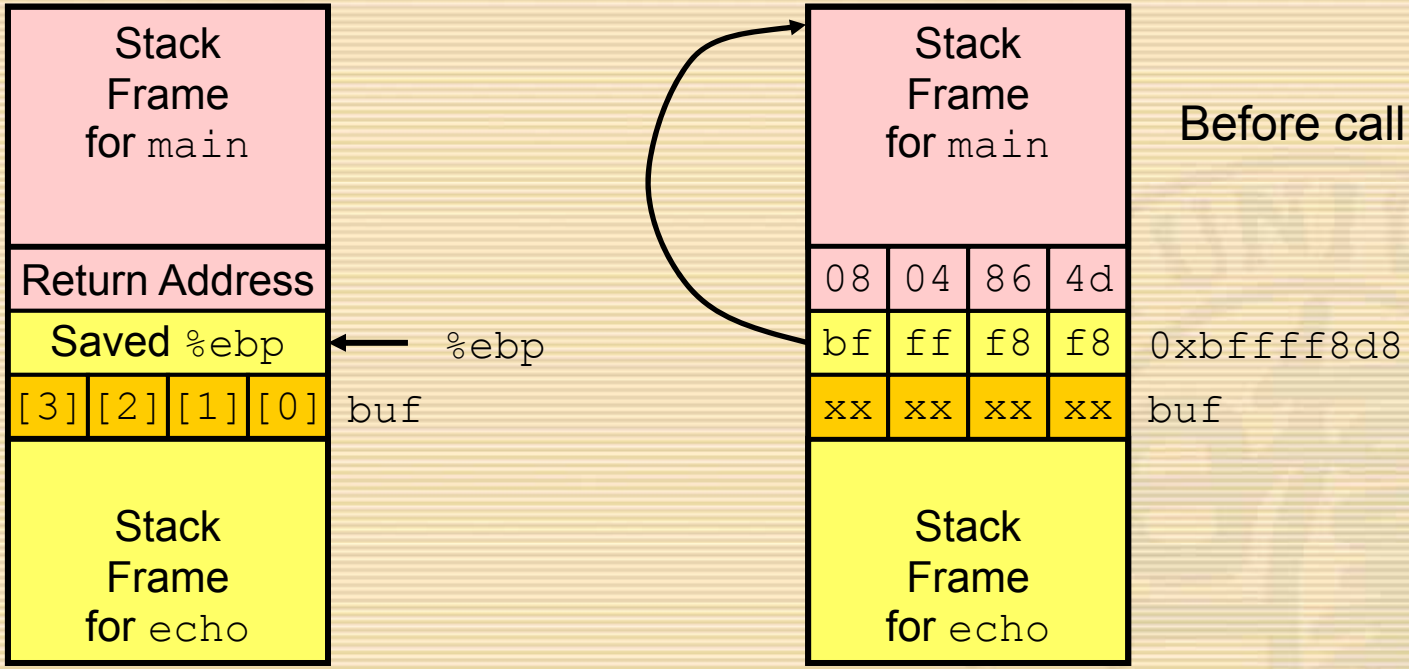
%ebp

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

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp           # Allocate space on stack
    pushl %ebx              # Save %ebx
    addl $-12,%esp          # Allocate space on stack
    leal -4(%ebp),%ebx      # Compute buf as %ebp-4
    pushl %ebx              # Push buf on stack
    call gets               # Call gets
    . . .
```

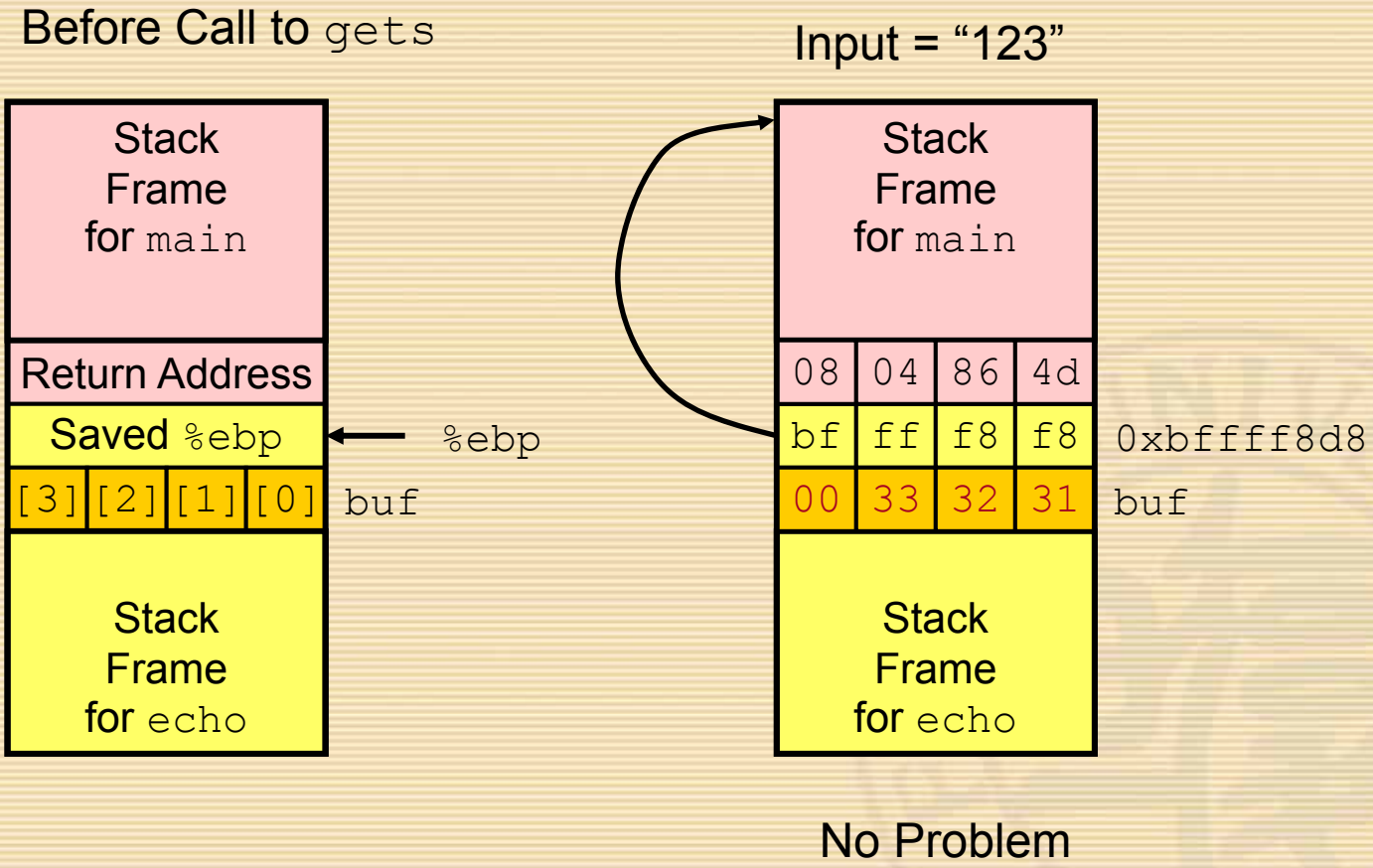
Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

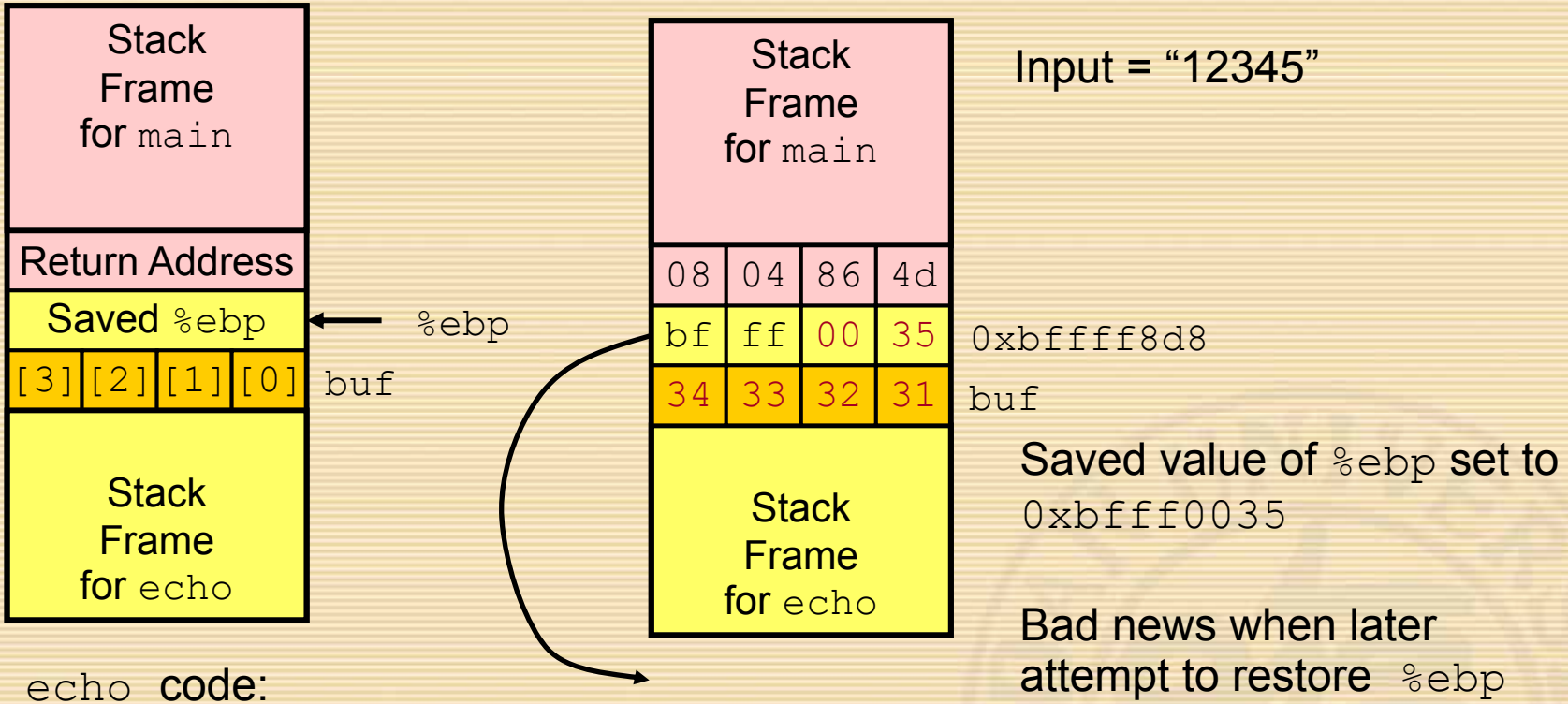



```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

Buffer Overflow Example #1



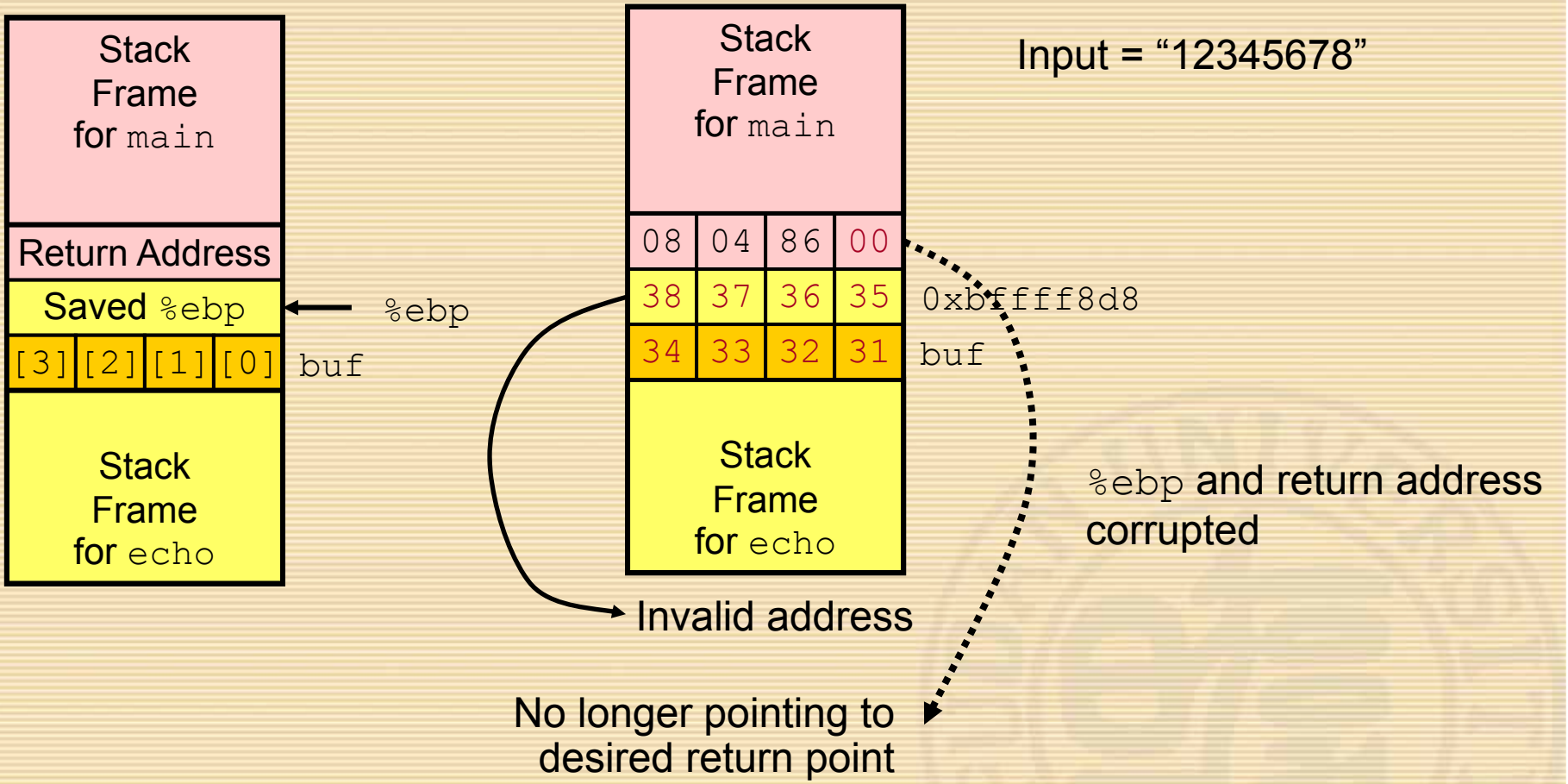
Buffer Overflow Example #2



`echo` code:

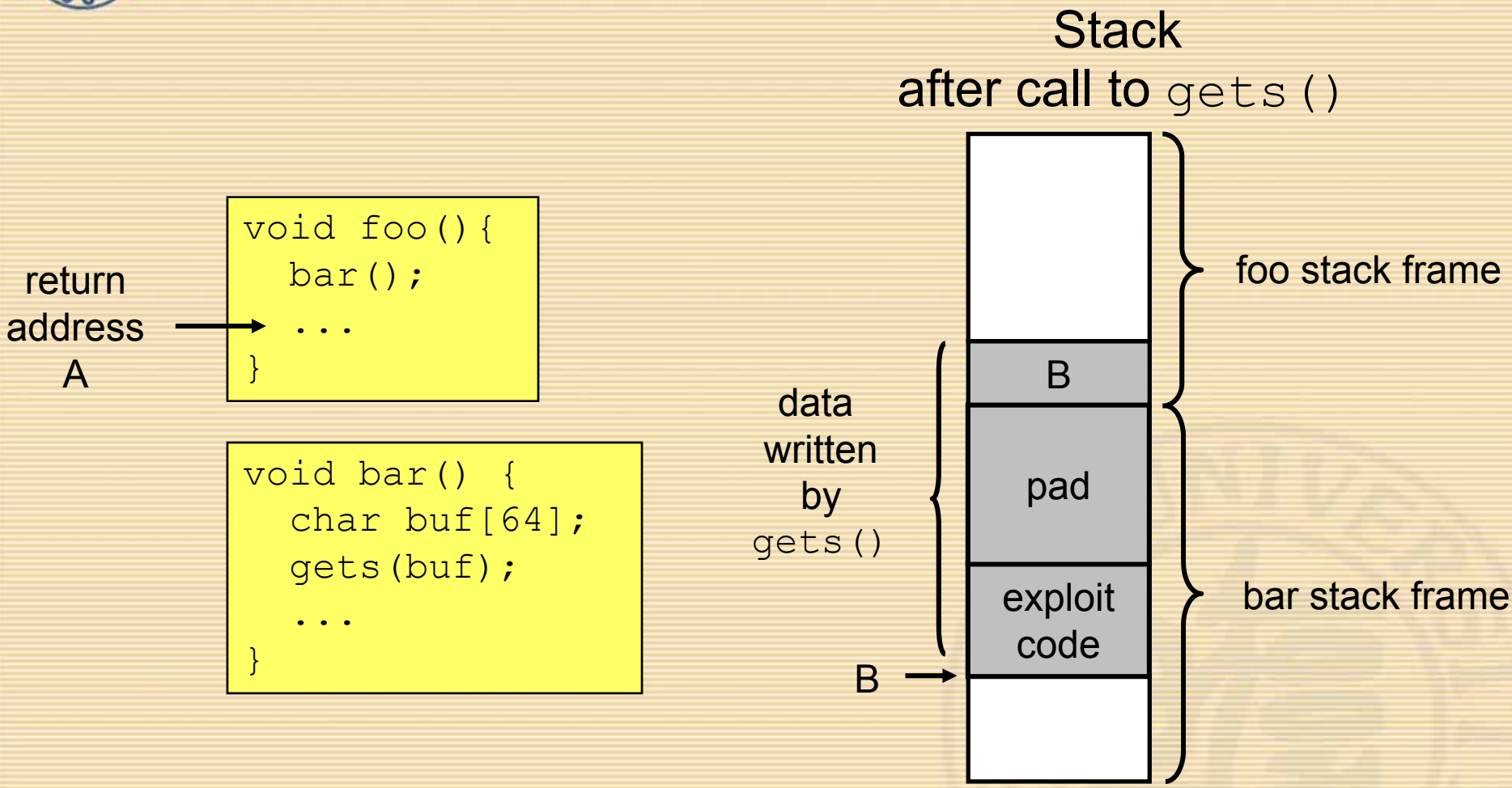
```
8048592:  push   %ebx
8048593:  call   80483e4 <_init+0x50>   # gets
8048598:  mov    0xffffffe8(%ebp),%ebx
804859b:  mov    %ebp,%esp
804859d:  pop    %ebp     # %ebp gets set to invalid value
804859e:  ret
```

Buffer Overflow Example #3




```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

Malicious Use of Buffer Overflow



- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `bar()` executes `ret`, will jump to exploit code

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.*

Avoiding Overflow Vulnerability

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

- 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

CIH (Chernobyl Virus)

- First wave of attack: 1999.4.26
- Second wave: 2000.4.26
- 60 Million Computers were infected including 15% of all Korean Computers
- CIH 2.0: 80% of the work was done when he was arrested

jc@fudan.edu.cn

Processor Architecture

jc@fudan.edu.cn

Goal

- Understand basic computer organization
  - Instruction set architecture
- Deeply explore the CPU working mechanism
  - How the instruction is executed: sequential and pipeline version
- Help you programming
  - Fully understand how computer is organized and works will help you write more stable and efficient code

CPU Design (Why?)

- It is interesting.
- Aid in understanding how the overall computer system works.
- Many design hardware systems contain processors.
- Maybe you will work on a processor design.

CPU Design

- Instruction set architecture
- Logic design
- Sequential implementation
- Pipelining and initial pipelined implementation
- Making the pipeline work
- Modern processor design

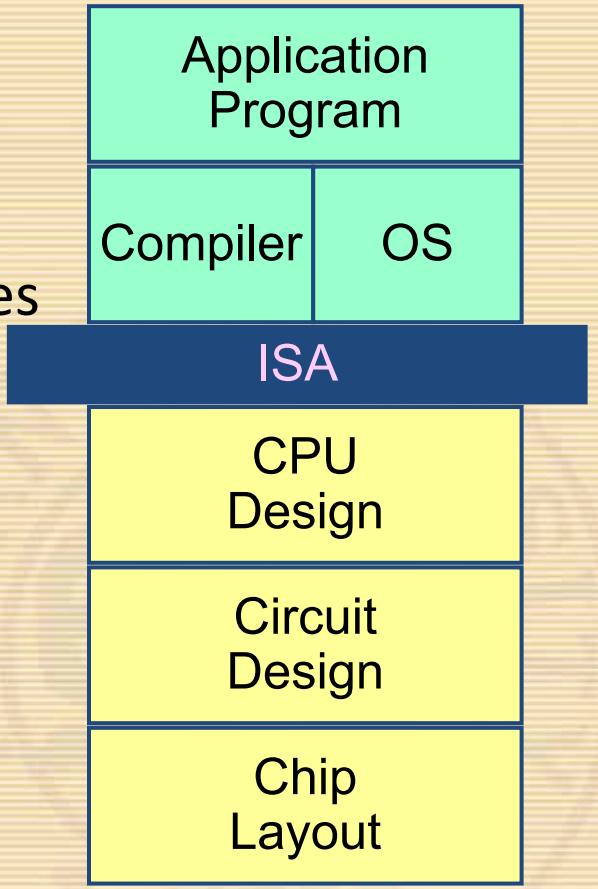
Instruction Set Architecture #1

- What is it ?
  - Assemble Language Abstraction
  - Machine Language Abstraction

  } Instruction Set Architecture (ISA)
- What does it provide?
  - An abstraction of the real computer, hide the details of implementation
    - The syntax of computer instructions
    - The semantics of instructions
    - The execution model
    - Programmer-visible computer status

Instruction Set Architecture #2

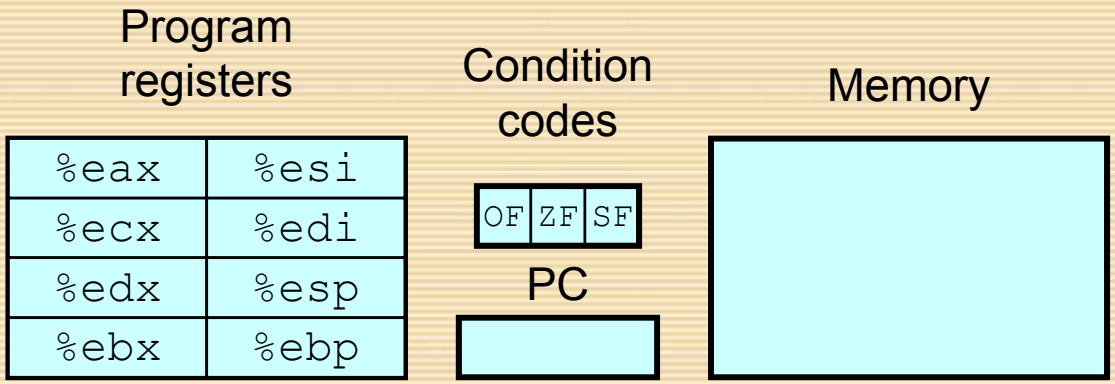
- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `movl`, `leal`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Instructions executed in a sequence
  - Below: what needs to be built
    - Use tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



Instruction Set Architecture #3

- ISA defines the processor family
  - Two main kind: RISC and CISC
    - RISC:SPARC, MIPS, PowerPC
    - CISC:X86 (or called IA32)
  - Another divide: Superscalar, VLIW and EPIC
    - Superscalar: all the above
    - VLIW: Philips TriMedia
    - EPIC: IA64
- Under same ISA, there are many different processors
  - From different manufacturers:
    - X86 from Intel and AMD and VIA
  - Different models
    - 8086, 80386, Pentium, Pentium 4…

Y86 Processor State



- Program Registers
  - Same 8 as with IA32. Each 32 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - OF: Overflow ZF: Zero SF:Negative
- Program Counter
  - Indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86 Instructions

- Format
  - 1--6 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with IA32
  - Each accesses and modifies some part(s) of the program state
- Errata: JXX and call are 5 bytes long.

Encoding Registers

- Each register has 4-bit ID

%eax	0
%ecx	1
%edx	2
%ebx	3

%esi	6
%edi	7
%esp	4
%ebp	5

- Same encoding as in IA32, but IA32 uses only 3-bit ID

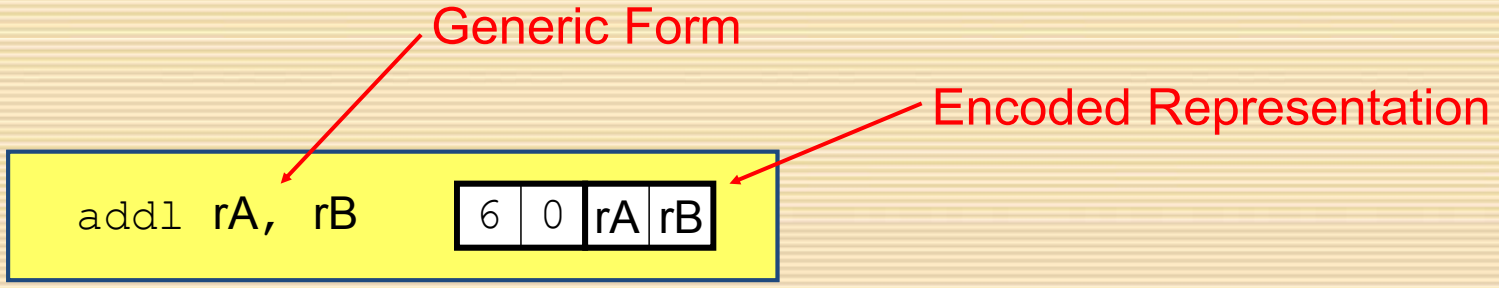
- Register ID 8 indicates “no register”
  - Will use this in our hardware design in multiple places

Supported Instructions

- mov
- +-&^
- Jxx
- cmp
- test
- call
- ret
- push
- pop

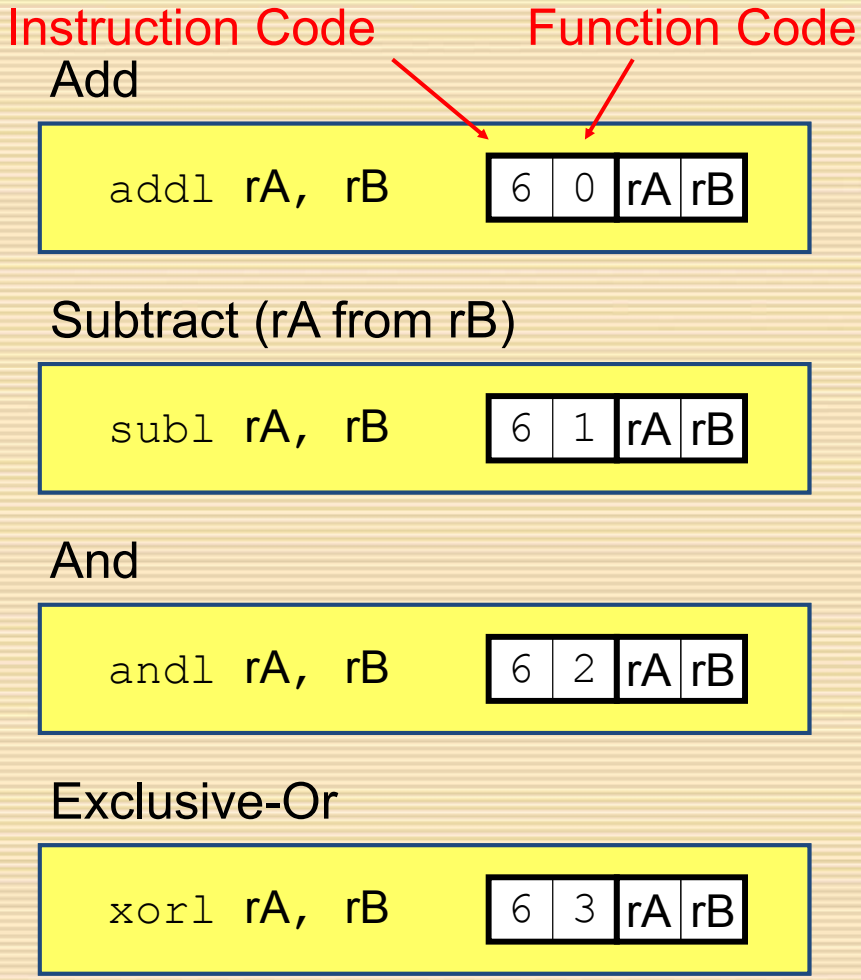
Instruction Example

- Addition Instruction



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

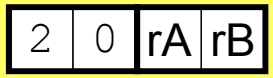
Arithmetic and Logical Operations



- Refer to generically as "`OPl`"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction byte
- Set condition codes as side effect
- Notice: no multiply or divide operation

Move Operations

`rrmovl` rA, rB



Register --> Register

`irmovl` V, rB

Immediate --> Register

`rmmovl` rA, D(rB)

Register --> Memory

`mrmovl` D(rB), rA

Memory --> Register

- Like the IA32 `movl` instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

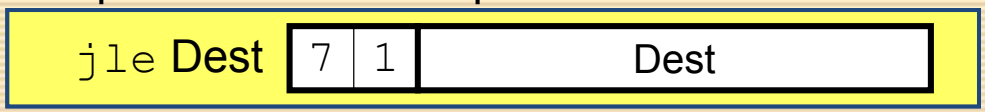
Move Instruction Examples

IA32	Y86	Encoding
`movl $0xabcd, %edx`	`irmovl $0xabcd, %edx`	`30 82 cd ab 00 00`
`movl %esp, %ebx`	`rrmovl %esp, %ebx`	`20 43`
`movl -12(%ebp),%ecx`	`mrmovl -12(%ebp),%ecx`	`50 15 f4 ff ff ff`
`movl %esi,0x41c(%esp)`	`rmmovl %esi,0x41c(%esp)`	`40 64 1c 04 00 00`

Jump Instructions

Jump Unconditionally

Jump When Less or Equal

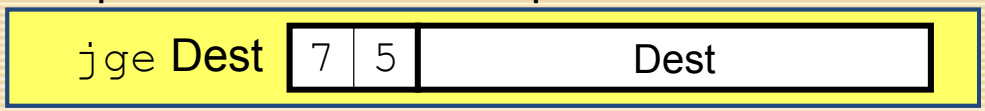


Jump When Less

Jump When Equal

Jump When Not Equal

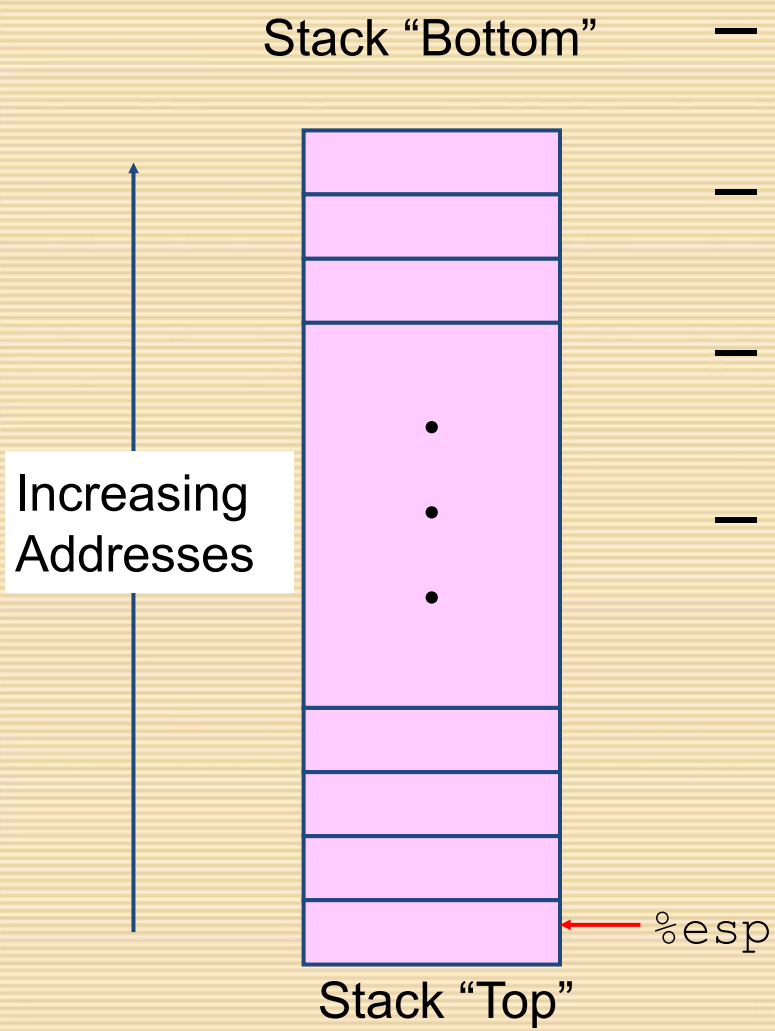
Jump When Greater or Equal



Jump When Greater

- Refer to generically as "`jXX`"
- Encodings differ only by "function code"
- Based on values of condition codes
- Same as IA32 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in IA32

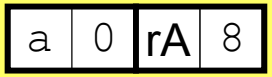
Y86 Program Stack



- Region of memory holding program data
- Used in Y86 (and IA32) for supporting procedure calls
- Stack top indicated by `%esp`
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at lowest address in the stack
  - When pushing, must first decrement stack pointer
  - When popping, increment stack pointer

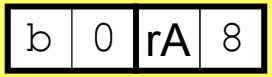
Stack Operations

`pushl` rA



- Decrement `%esp` by 4
- Store word from rA to memory at `%esp`
- Like IA32

`popl` rA



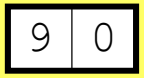
- Read word from memory at `%esp`
- Save in rA
- Increment `%esp` by 4
- Like IA32

Subroutine Call and Return

`call` Dest

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like IA32

`ret`



- Pop value from stack
- Use as address for next instruction
- Like IA32

Miscellaneous Instructions

nop	0	0

– Don’t do anything

halt	1	0

– Stop executing instructions
– IA32 has comparable instruction, but can’t execute it in user mode
– We will use it to stop the simulator

Y86 Program Structure

```
   irmovl Stack,%esp  # Set up stack
   rrmovl %esp,%ebp   # Set up frame
   irmovl List,%edx
   pushl %edx         # Push argument
   call len2          # Call Function
   halt               # Halt
.align 4
List:                 # List of elements
   .long 5043
   .long 6125
   .long 7395
   .long 0

# Function
len2:
   . . .
# Allocate space for stack
.pos 0x100
Stack:
```

- Program starts at address 0
- Must set up stack
  - Make sure don't overwrite code!
- Must initialize data
- Can use symbolic names

Page 266

Assembling Y86 Program

```
unix> yas eg.ys
```

– Generates "object code" file `eg.yo`

- Actually looks like disassembler output

```
0x000: 308400010000 | irmovl Stack,%esp    # Set up stack
0x006: 2045         | rrmovl %esp,%ebp     # Set up frame
0x008: 308218000000 | irmovl List,%edx
0x00e: a028         | pushl %edx           # Push argument
0x010: 8028000000   | call len2            # Call Function
0x015: 10           | halt                 # Halt
0x018:              | .align 4
0x018:              | List:                # List of elements
0x018: b3130000     |   .long 5043
0x01c: ed170000     |   .long 6125
0x020: e31c0000     |   .long 7395
0x024: 00000000     |   .long 0
```

Simulating Y86 Program

```
unix> yis eg.yo
```

- Instruction set simulator
  - Computes effect of each instruction on processor state
  - Prints changes in state from original

```
Stopped in 41 steps at PC = 0x16.  Exception 'HLT', CC Z=1 S=0
O=0
Changes to registers:
%eax:                   0x00000000              0x00000003
%ecx:                   0x00000000              0x00000003
%edx:                   0x00000000              0x00000028
%esp:                   0x00000000              0x000000fc
%ebp:                   0x00000000              0x00000100
%esi:                   0x00000000              0x00000004

Changes to memory:
0x00f4:                 0x00000000              0x00000100
0x00f8:                 0x00000000              0x00000015
0x00fc:                 0x00000000              0x00000018
```

Summary

- Y86 Instruction Set Architecture
  - Similar state and instructions as IA32
  - 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
  - Intel has moved away from IA32
    - Does not allow enough parallel execution
    - Introduced IA64
      - 64-bit word sizes (overcome address space limitations)
      - Radically different style of instruction set with explicit parallelism
      - Requires sophisticated compilers

Logic Design

- Digital circuit
  - What is digital circuit?
  - Know what a CPU will base on?

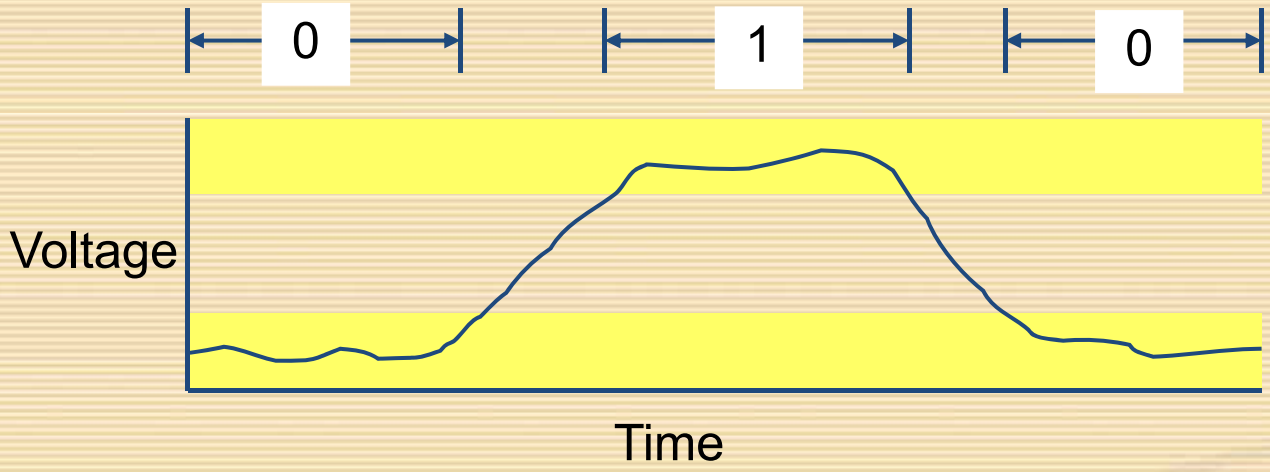
- Hardware Control Language (HCL)
  - A simple and functional language to describe our CPU implementation
  - Syntax like C

Category of Circuit

- Analog Circuit
  - Use all the range of Signal
  - Most part is amplifier
  - Hard to model and automatic design
  - Use transistor and capacitance as basis
  - We will not discuss it here
- Digital Circuit
  - Has only two values, 0 and 1
  - Easy to model and design
  - Use true table and other tools to analyze
  - Use gate as the basis
  - The voltage of 1 differs in different kind circuit.
    - E.g. TTL circuit using 5 voltage as 1

Amplifier: 放大器
Transistor：晶体管
Capacitance: 电容

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

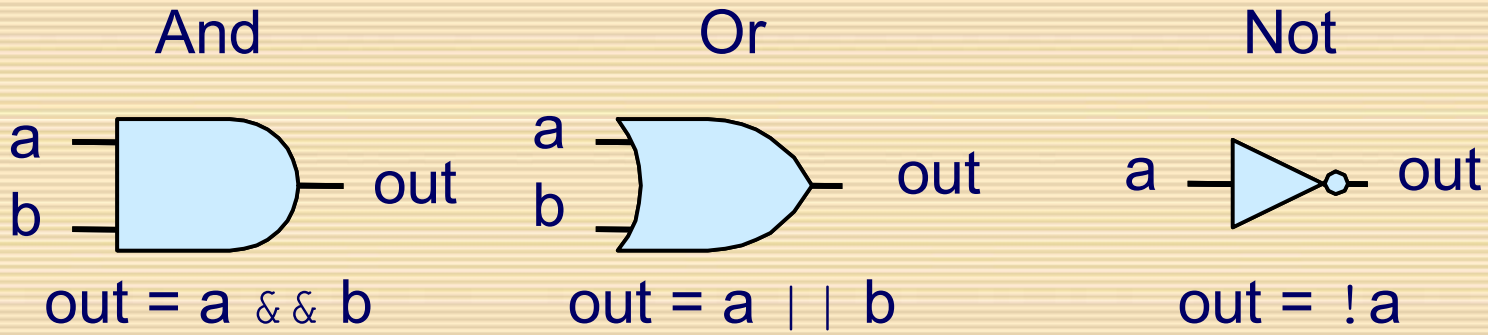
Overview of Logic Design

- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage (Memory)
  - Clock Signal
- 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
  - Clock Signal

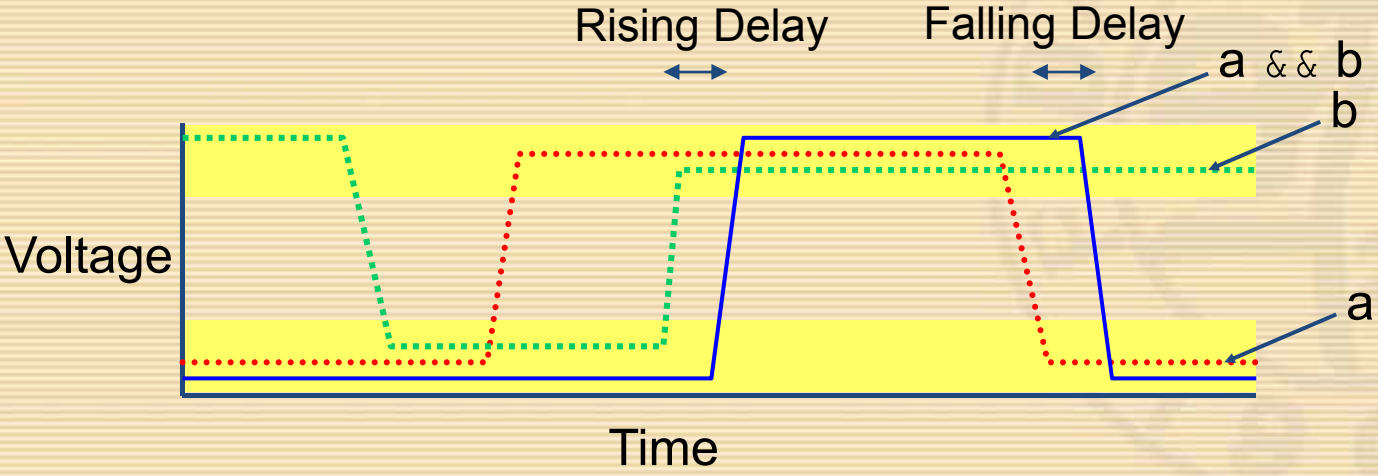
Category of Digital Circuit

- Combinational Circuit
  - Without memory. So the circuit can't have state. Any same input will get the same output at any time.
  - Needn't clock signal
  - Typical application: ALU
- Sequential Circuit
  - Equals to: Combinational circuit + memory and clock signal
  - Have state. Two same inputs may not generate the same output.
  - Use clock signal to control the run of circuit.
  - Typical application: CPU

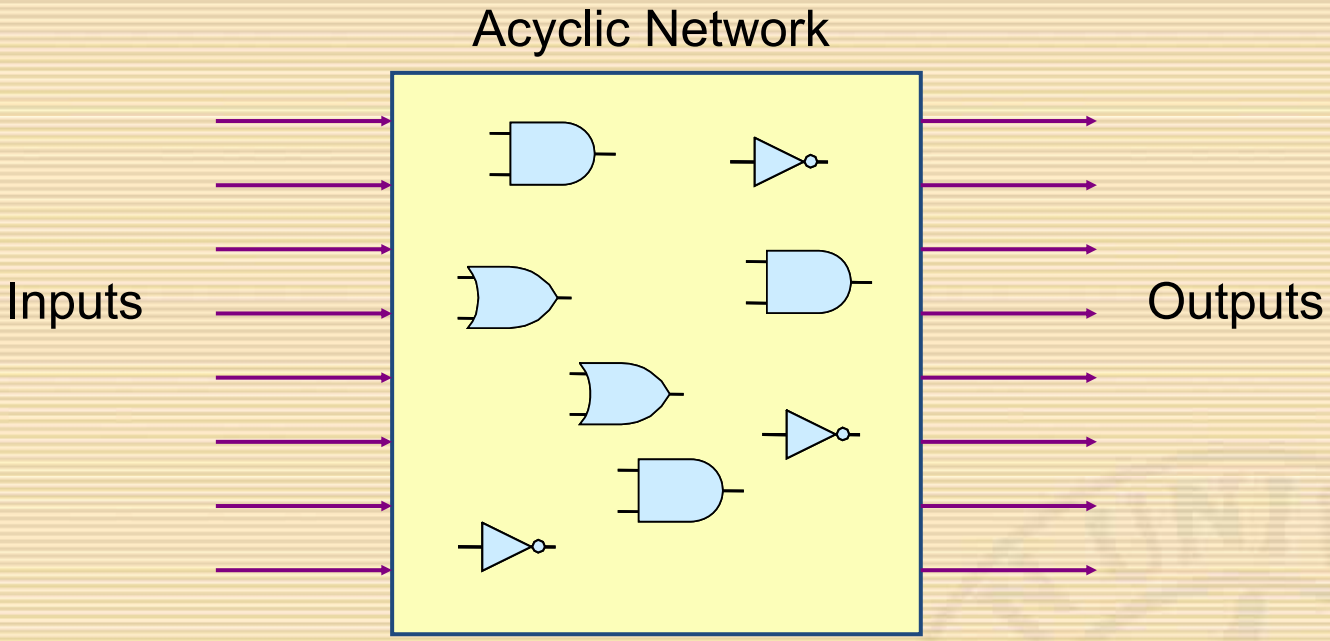
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Not an assignment operation, just give the circuit a name
- Respond continuously to changes in inputs
  - With some, small delay

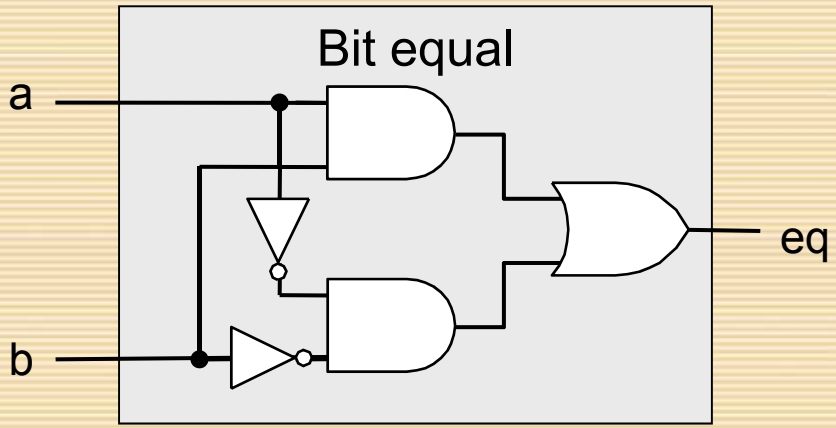


Combinational Circuits



- Acyclic Network of Logic Gates
  - Continuously responds to changes on inputs
  - Outputs become (after some delay) Boolean functions of inputs

Bit Equality

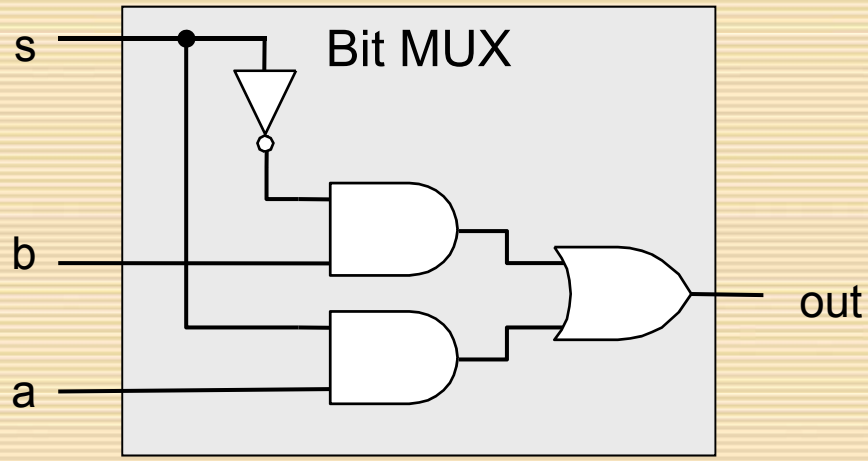


HCL Expression

```
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

Bit-Level Multiplexor

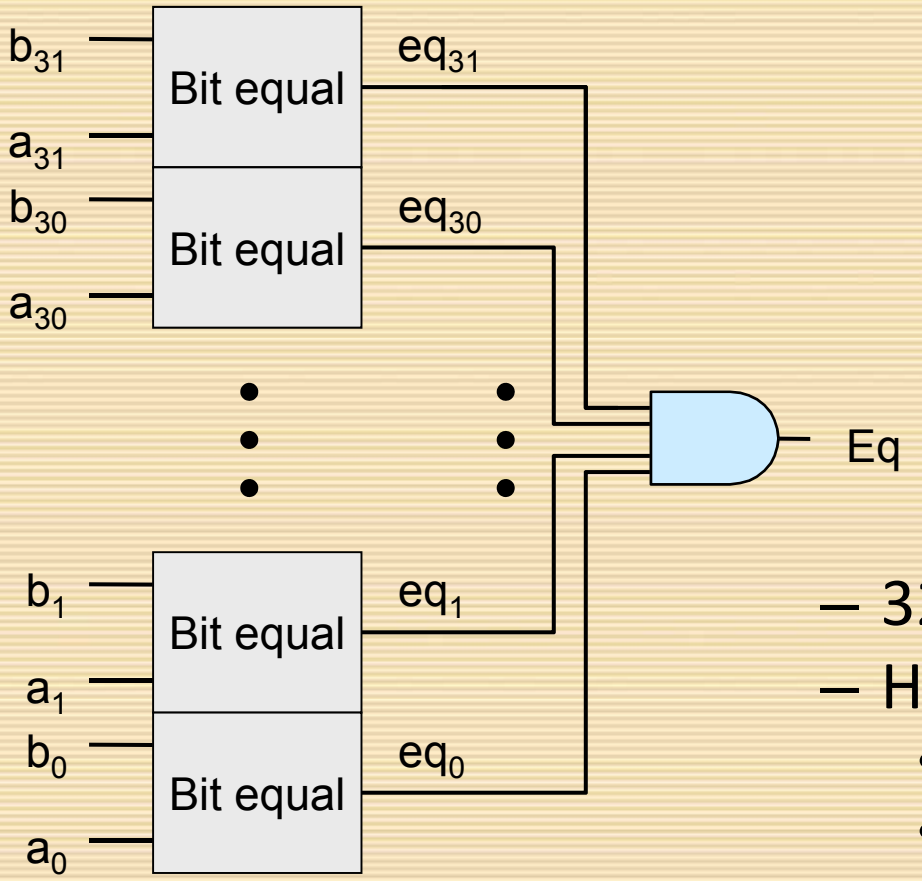


HCL Expression

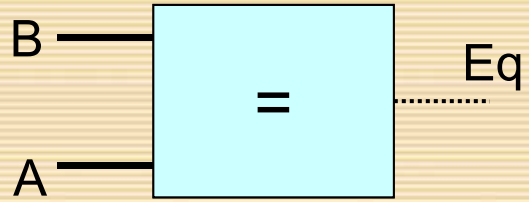
```
bool out = (s&&a)||(!s&&b)
```

- Control signal s
- Data signals a and b
- Output a when s=1, b when s=0
- Its name: MUX
- Usage: Select one signal from a couple of signals

Word Equality



Word-Level Representation

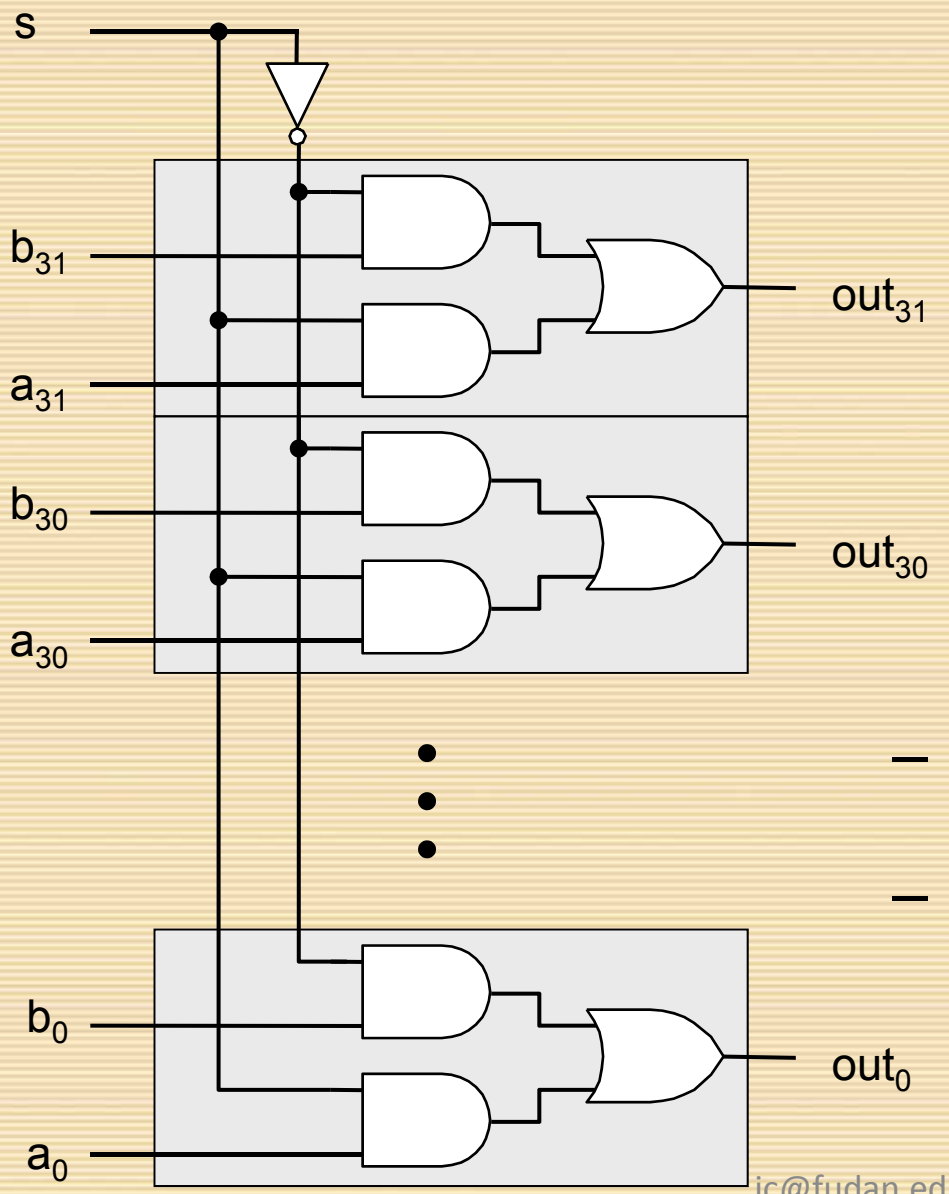


HCL Representation

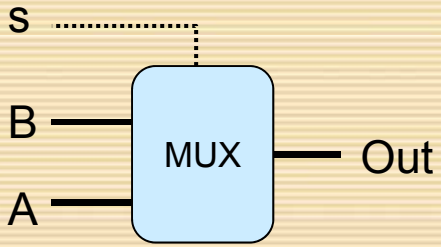
```
bool Eq = (A == B)
```

- 32-bit word size
- HCL representation
  - Equality operation
  - Generates Boolean value

Word Multiplexor



Word-Level Representation



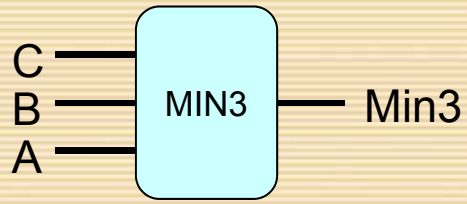
HCL Representation

```
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

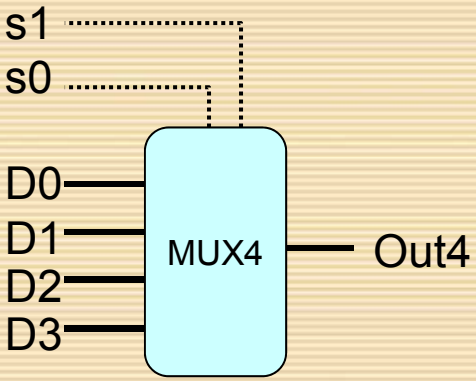
Minimum of 3 Words




```
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

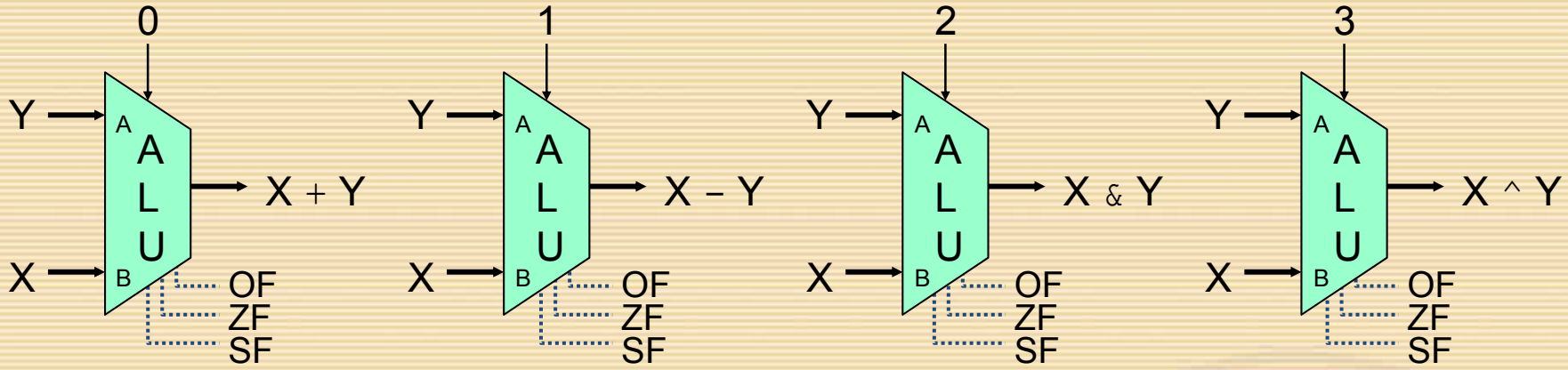
4-Way Multiplexor




```
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

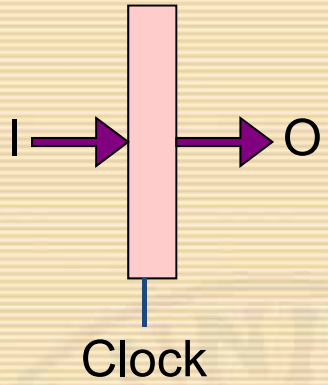
Arithmetic Logic Unit



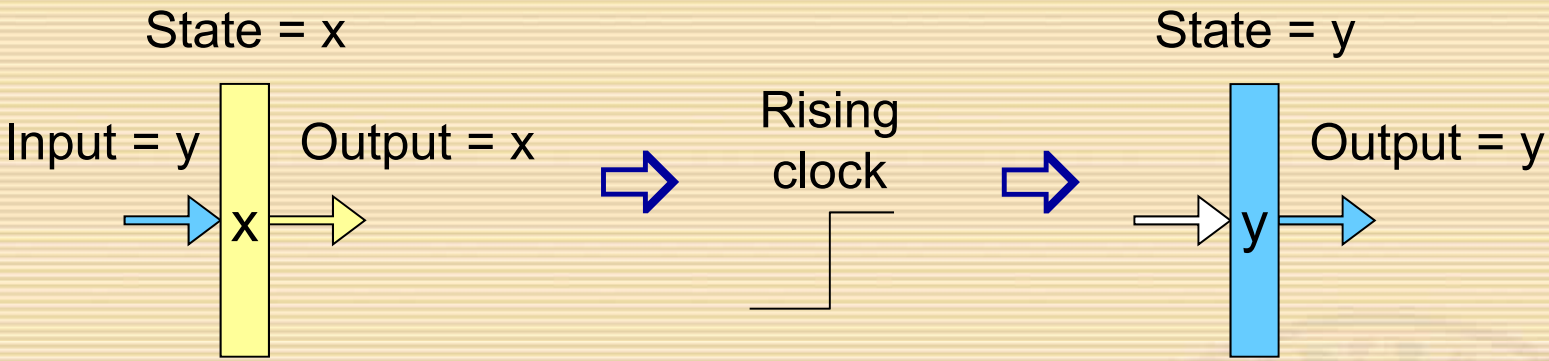
- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86
- Also computes values for condition codes
- We will use it as a basic component for our CPU

Storage

- Registers
  - Hold single words or bits
  - Loaded as clock rises
  - Not only *program registers*
- Random-access memories
  - Hold multiple words
  - E.g. register file, memory
  - Possible multiple read or write ports
  - Read word when address input changes
  - Write word as clock rises

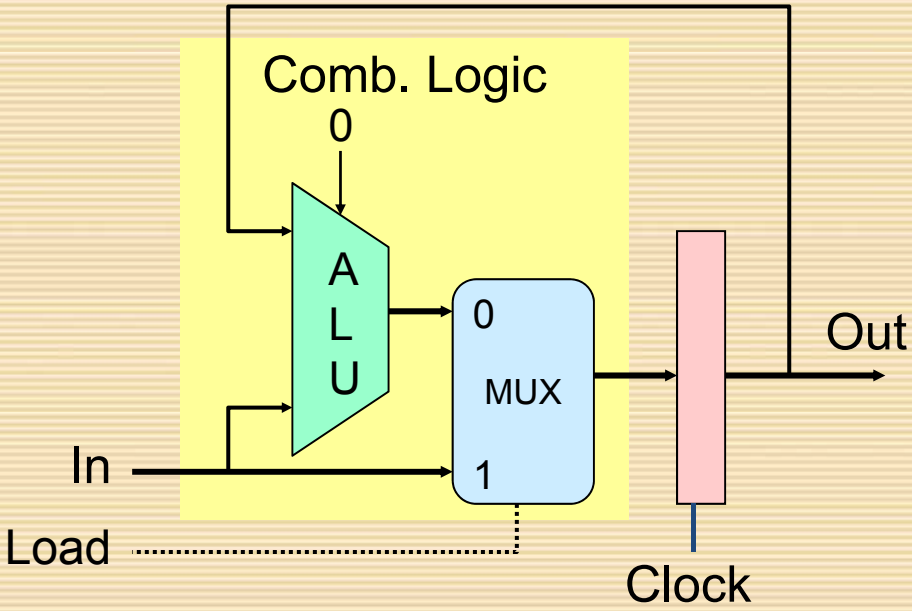


Register Operation

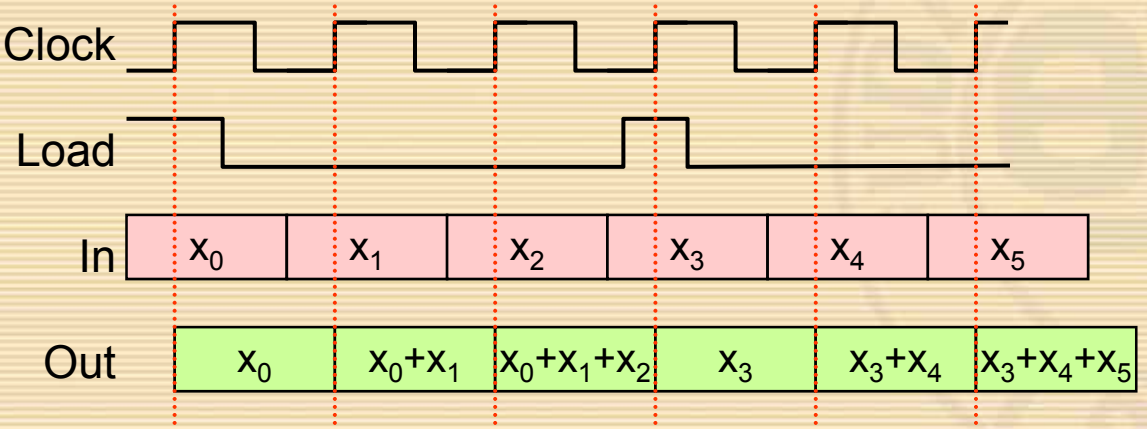


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

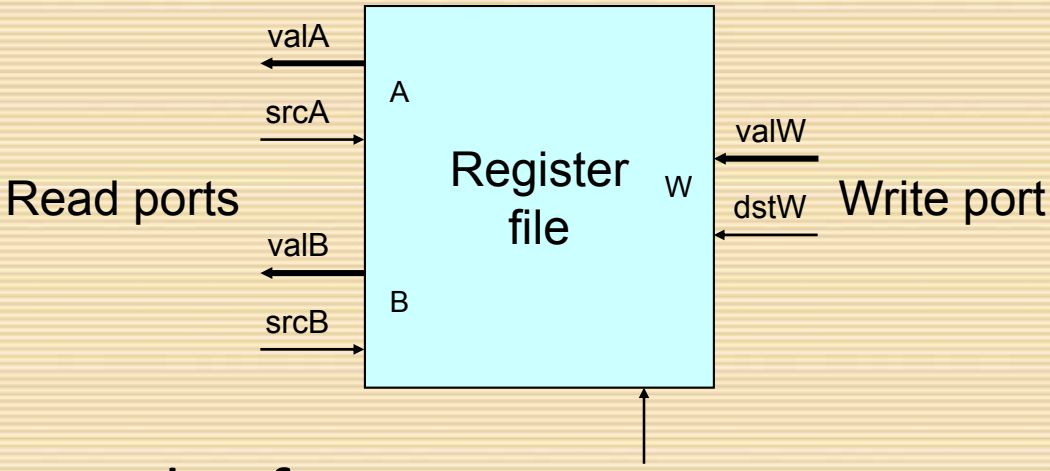
State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle



Random-Access Memory

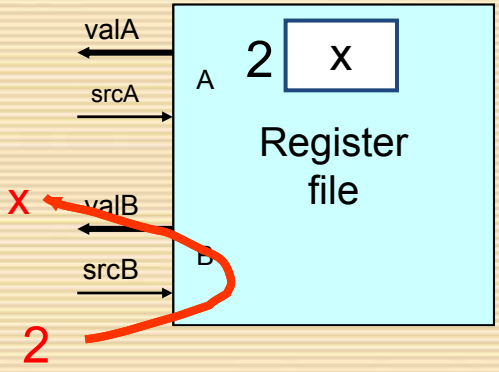


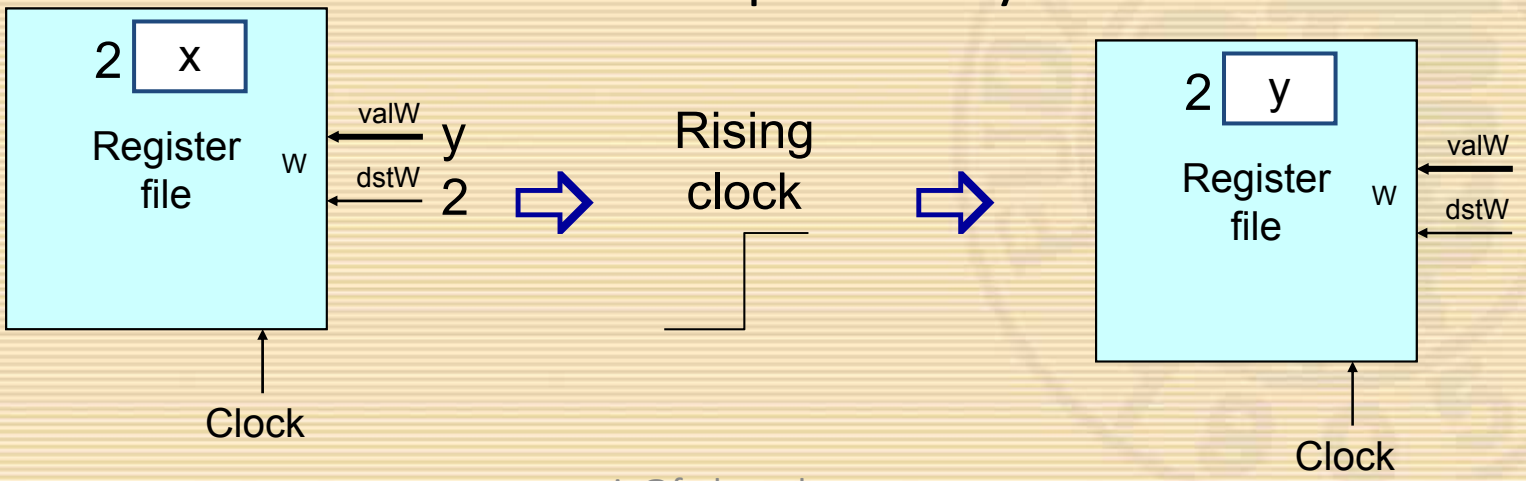
Page 280

- Stores multiple words of memory
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers
  - `%eax`, `%esp`, etc.
  - Register identifier serves as address
    - ID 8 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





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

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