

Lecture 2

Machine and Assembly Language
SIC, SIC/XE and x86-64, seen through C

Md Mahbub E Noor

Assistant Professor, Department of CSE

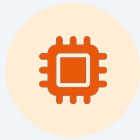
University of Barishal

```
● ● ● one line of C, three ways  
/* C */  
beta = alpha + incr - 1;  
; SIC assembly  
LDA ALPHA    ADD INCR  
SUB ONE      STA BETA  
; SIC machine code (hex)  
00100F 181012 1C1015 0C1018
```

What you will learn today



Read a short assembly program by matching each part to the C statement it came from.



Describe the SIC machine and encode a SIC instruction into hex by hand.



Name the SIC/XE formats and addressing modes, and compute a target address.



Read gcc's x86-64 output and write, build and run a small nasm program.

Today's plan (60 min)

0-5

Recap of Lecture 1

5-15

C and assembly side by side

15-35

The SIC machine and programs

35-45

SIC/XE formats, addressing

45-55

x86-64 and nasm

55-60

Checkpoint, homework

Writing nasm programs (slides 28-30) continues in the lab period.

Quick recap of Lecture 1

Q1

Which program turns `hello.s` into `hello.o`?

The assembler (`as`). Today we look at what goes into it: assembly language.

Q2

What does the CPU do, again and again?

Fetch the instruction PC points to, decode it, execute it. Today we see how an instruction is laid out in bits, so decode makes sense.

Q3

Why did `hello.o` contain `00 00 00 00`?

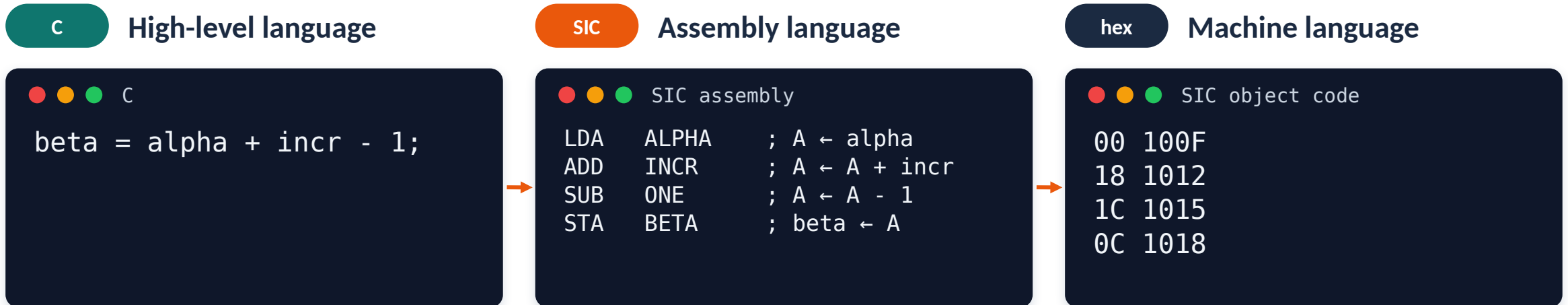
The assembler did not yet know some addresses. Today we see how an address sits inside an instruction.

PART 1

C and assembly, side by side

You already know the ideas. Assembly just spells them out one tiny step at a time.

One C statement, three levels of language



One line can hide many steps. The compiler works out the order and where the values live.

One line = **one machine instruction**. Names (mnemonics, labels) instead of numbers.

What is really stored in memory. The first byte is the **opcode**, the rest is the **address**.

Compiler = C → assembly (many-to-one, clever). **Assembler** = assembly → machine code (one-to-one, mechanical). The assembler is what we build in Lecture 3.

Book: Beck §1.3.1, Appendix A (opcodes)

You already know more assembly than you think

C you wrote in 1st semester	SIC assembly	The idea they share
<code>int alpha = 20;</code>	<code>ALPHA WORD 20</code>	a named memory word with a starting value
<code>int beta;</code>	<code>BETA RESW 1</code>	a named memory word, no starting value
<code>beta = alpha;</code>	<code>LDA ALPHA / STA BETA</code>	a copy, but always through register A
<code>beta = alpha + incr - 1;</code>	<code>LDA · ADD · SUB · STA</code>	one operation per instruction
<code>if (a > b) ...</code>	<code>COMP B then JGT label</code>	compare first, then a conditional jump
<code>for / while</code>	<code>label ... TIX ... JLT label</code>	the loop is a jump back to the top
<code>str[i]</code>	<code>LDCH STR,X</code>	register X plays the role of i
<code>y = square(x);</code>	<code>JSUB SQUARE ... RSUB</code>	call a routine and come back
<code>goto again;</code>	<code>J AGAIN</code>	exactly the same idea

Where the C picture stops working

DIFFERENT FROM C

No types

In C, the compiler checks `int` and `char`. In assembly, memory is only bytes. `LDA STR1` on the string "HELLO" loads the three bytes 48 45 4C as the number **4,736,332**. No error, no warning.

DIFFERENT FROM C

One accumulator, not many variables

C gives you as many local variables as you like. SIC has one accumulator **A**. Every calculation passes through it, so `a = b + c * d` needs an extra memory word to hold a partial result.

DIFFERENT FROM C

No if, for, while: only jumps

Structured statements do not exist. The assembler only knows **labels** and **jumps**, like C's `goto`. Keeping the structure clear is now your job, with comments and good label names.

DIFFERENT FROM C

No automatic call stack

C keeps return addresses on a stack for you, so a function can call another, or itself. SIC's `JSUB` keeps the return address in **one register, L**. A second `JSUB` overwrites it unless you save `L` first.

PART 2

The SIC machine

Beck's Simplified Instructional Computer: small enough to assemble by hand, real enough to teach every idea.

SIC at a glance: memory, registers and data



Memory

32,768 bytes (2^{15}), byte-addressed. A **word** is **3 bytes**, and the address of a word is the address of its lowest byte.



Registers

Five, each one word (24 bits): **A** accumulator, **X** index, **L** linkage, **PC** program counter, **SW** status word (holds the condition code).



Data

Integers: 24-bit two's complement. **Characters:** 8-bit ASCII. **No floating point** in plain SIC.

C int vs SIC word

	C int (lab PC)	SIC word
Size	32 bits	24 bits
Smallest	-2,147,483,648	-8,388,608
Largest	2,147,483,647	8,388,607
-9 is stored as	FF FF FF F7	FF FF F7

Same idea, two's complement; only the width differs.

Like C: an `int` is a fixed number of bytes, and negative numbers use two's complement. **Unlike C:** there is no `char` type. A character is just one byte, and only the instructions `LDCH` and `STCH` treat it as one.

The SIC instruction format: 24 bits, three fields



Every SIC instruction is exactly **3 bytes**. The **opcode** says which operation. The **x bit** says whether to use the index register. The **address** says where the operand is in memory: 15 bits, which is exactly enough for 32,768 bytes.

Direct addressing ($x = 0$)

Target address: $TA = \text{address}$

SIC: LDA ALPHA C idea: alpha

The operand is at the address written in the instruction.

Indexed addressing ($x = 1$)

Target address: $TA = \text{address} + (X)$

SIC: LDCH STR1,X C idea: str1[i]

Add the contents of X first. X plays the role of the array index i.

Book: Beck §1.3.1 (instruction format, addressing modes)

Worked example: encode STCH BUFFER,X by hand

Given: BUFFER is at address **1039** (hex), and the opcode of STCH is **54** (Beck, Appendix A).

1	Opcode in binary	54 hex → 0101 0100	8 bits
2	Indexed, so x = 1	the operand ends in ,X → x = 1	1 bit
3	Address in 15 bits	1039 hex → 001 0000 0011 1001	15 bits
4	Join the 24 bits	0101 0100 1 001 0000 0011 1001	24 bits
5	Regroup in fours	0101 0100 1001 0000 0011 1001	
6	Read as hex	5 4 9 0 3 9 → 549039	

Shortcut

Because x is the top bit of the address field, setting x = 1 just **adds 8000 hex** to the address:

$$1039 + 8000 = 9039$$

So the answer is opcode 54 followed by 9039.

Try it: LDA ALPHA with ALPHA at 1033 and LDA = 00. → x = 0, so the answer is simply 001033.

Book: Beck §1.3.1; the same instruction appears in Beck's Fig. 2.2

The SIC instruction set, grouped by job

Group	Instructions	Opcodes (hex)	What they do	C idea
Load / store	LDA <i>m</i> · STA <i>m</i>	00 · 0C	$A \leftarrow (m) \cdot m \leftarrow (A)$	$x = y$
	LDX · STX · LDL · STL	04 · 10 · 08 · 14	same for X and L	
	LDCH <i>m</i> · STCH <i>m</i>	50 · 54	one byte $\leftrightarrow$ rightmost byte of A	$c = s[i]$
Arithmetic	ADD · SUB · MUL · DIV	18 · 1C · 20 · 24	$A \leftarrow A \text{ op } (m)$	+ - * /
Compare	COMP <i>m</i>	28	$CC \leftarrow$ compare A with (m)	$a > b ?$
	TIX <i>m</i>	2C	$X \leftarrow X + 1$, then compare X with (m)	$i++; i < n ?$
Jump	J · JEQ · JGT · JLT	3C · 30 · 34 · 38	$PC \leftarrow m$ (always / if CC is = > <)	goto, if
Subroutine	JSUB <i>m</i> · RSUB	48 · 4C	$L \leftarrow PC, PC \leftarrow m \cdot PC \leftarrow L$	call, return
Input/output	TD · RD · WD	E0 · D8 · DC	test device, read byte, write byte	getchar, putchar

(*m*) means "the contents of memory at *m*". Students do not memorise opcodes: the assembler looks them up in its table (OPTAB, Lecture 3).

Book: Beck §1.3.1 and Appendix A

Assembler directives are your C declarations

A **directive** is an order to the assembler, not to the CPU. It produces **no instruction**: it only names memory and sets starting values, exactly like a declaration in C.

C declaration	SIC directive	Memory it reserves
<code>int five = 5;</code>	<code>FIVE WORD 5</code>	one word, starting value 5
<code>int count;</code>	<code>COUNT RESW 1</code>	one word, no starting value
<code>int table[100];</code>	<code>TABLE RESW 100</code>	100 words = 300 bytes
<code>char buf[4096];</code>	<code>BUFFER RESB 4096</code>	4096 bytes
<code>char eof[] = "EOF";</code>	<code>EOF BYTE C 'EOF '</code>	3 bytes: 45 4F 46 (no \0 at the end!)
<code>char dev = 0xF1;</code>	<code>INPUT BYTE X 'F1 '</code>	1 byte given in hex
<code>(program begins / ends)</code>	<code>COPY START 1000 ... END</code>	where the program is loaded, where it starts

Watch the orange row: a C string ends with an invisible `\0`. SIC's `BYTE C 'EOF '` is exactly 3 bytes. Nothing marks the end, so the program must know the length.

Book: Beck §1.3.1 (directives used in §1.3.3 examples)

Program 1: if-else becomes compare and jump

```
● ● ● C
if (numa > numb)
    max = numa;
else
    max = numb;
```

SAME AS C

Two paths meet

Both paths end at one place (ABIG), just like the code after an if-else in C.

```
● ● ● SIC (our assembler's listing)
addr  label  instr  operand  code
1000              LDA   NUMA   001012
1003              COMP  NUMB   281015
1006          JGT   ABIG   34100C
1009              LDA   NUMB   001015
100C  ABIG      STA   MAX    0C1018
. . .
1012  NUMA      WORD  17
1015  NUMB      WORD  42
1018  MAX       RESW  1
```

LDA NUMA

A ← numa, as if "then" were already chosen.

COMP NUMB

Compare A with numb: 17 < 42, so CC = '<'.

JGT ABIG

Jump only if CC is '>'. It is not, so carry on.

LDA NUMB

The else part: replace A with numb (42).

STA MAX

Both paths meet here. max = 42.

Different from C: the > of C is split in two. **COMP** only remembers the result in the condition code; the **jump** reads it later. Notice the trick too: A already holds numa, so the "then" part needs no instruction at all.

Book: Beck §1.3.3 (SIC programming examples)

Program 2: a for-loop with the index register

```
c
for (i = 0; i < 5; i++)
    str2[i] = str1[i];
```

```
SIC
MOVECH  LDX  ZERO
        LDCH STR1,X
        STCH STR2,X
        TIX  FIVE
        JLT  MOVECH

...
STR1    BYTE  C'HELLO'
STR2    RESB  5
ZERO    WORD  0
FIVE    WORD  5
```

c `i = 0` **SIC** `LDX ZERO`
X starts at 0. X is our i.

c `str2[i] = str1[i]` **SIC** `LDCH STR1,X · STCH STR2,X`
Copy one byte through A. Indexed: TA = STR1 + X, then STR2 + X.

c `i++ and i < 5` **SIC** `TIX FIVE`
Two jobs in one: X = X + 1, then compare X with 5 and set CC.

c `back to the top` **SIC** `JLT MOVECH`
If CC is '<', jump to MOVECH. When X reaches 5, CC is '=' and we fall through.

Run it: **21 instructions** (1 + 5 rounds × 4) copy HELLO. The next slide shows every one.

Watch the loop run, instruction by instruction

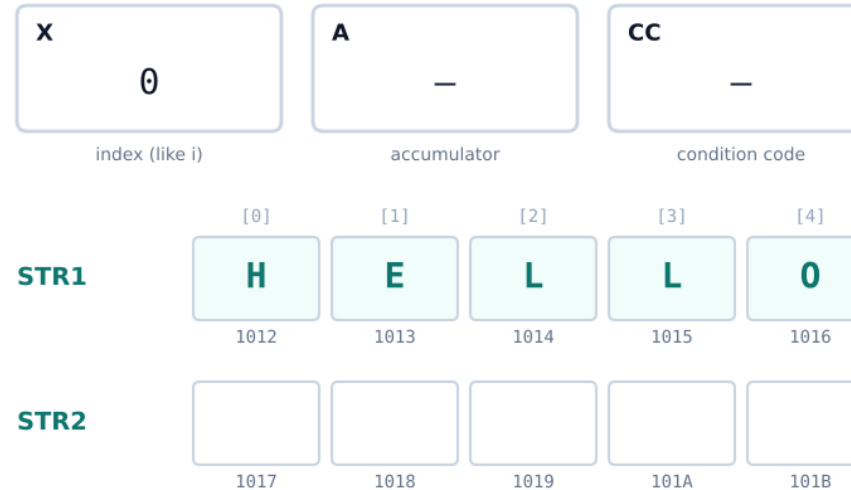
A C loop, running as SIC instructions

C (what you wrote in 1st semester)

```
for (i = 0; i < 5; i++)  
    str2[i] = str1[i];
```

```
1000      LDX  ZERO  
1003  MOVECH LDCH  STR1,X  
1006      STCH STR2,X  
1009      TIX  FIVE  
100C      JLT  MOVECH
```

Registers



The same loop twice: C on top, the SIC instructions the assembler would produce below. Watch X: it plays the role of i.

Click the video to play or pause. About 1 minute, no sound.

Watch for

1. The **orange C part** at the top-left changes as each SIC line runs.
2. **X** grows by 1 only at TIX.
3. **A** holds one character at a time: the byte travels through it.
4. The **target address** in the caption is always $STR + X$.
5. In round 5, CC becomes '=' and JLT stops jumping.

Program 3: a function call with JSUB and RSUB

```
int square(int x)
{
    return x * x;
}

res1 = square(7);
res2 = square(12);
```

SAME AS C

Call and return

Like C: jump into the function, run it, come back to the line after the call.

```
SIC
1000      LDA    NUM1
1003      JSUB   SQUARE
1006      STA    RES1
1009      LDA    NUM2
100C      JSUB   SQUARE
100F      STA    RES2
...
1015 SQUARE STA    TEMP
1018      MUL    TEMP
101B      RSUB
```

JSUB at 1003

$L \leftarrow 1006$ (the next instruction), $PC \leftarrow 1015$

RSUB

$PC \leftarrow L = 1006$. $A = 49$, stored in RES1

JSUB at 100C

$L \leftarrow 100F$, $PC \leftarrow 1015$

RSUB

$PC \leftarrow L = 100F$. $A = 144$, stored in RES2

Different from C: the argument and the result both travel in **A**, by agreement only. And the return address lives in **L**, not on a stack: if SQUARE called another routine, it would first have to save L with STL.

Book: Beck §1.3.1 (JSUB, RSUB, register L)

PART 3

SIC/XE: the extended machine

More memory, more registers, four instruction formats, and the addressing modes real CPUs use.

What SIC/XE adds to SIC

	SIC	SIC/XE
Memory	32 KB (2^{15} bytes)	1 MB (2^{20} bytes)
Registers	A, X, L, PC, SW	+ B (base), S, T (general), F (48-bit float)
Instruction formats	one: 3 bytes	four: 1, 2, 3 or 4 bytes
Addressing modes	direct, indexed	+ immediate, indirect, PC-relative, base-relative
Floating point	none	48-bit (ADDF, MULF ...)
Register-to-register	none	ADDR S,A · COMPR A,S · CLEAR X ...

Why the changes?

1 MB of memory needs 20-bit addresses, which do not fit in the old 15-bit field. So XE adds a 4-byte format.

More registers mean fewer trips to memory. In C terms: more values can stay "in your hands" instead of in variables.

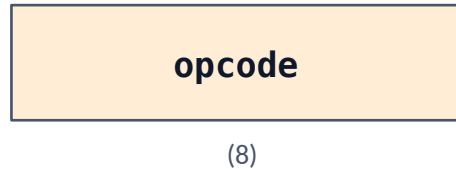
XE is **upward compatible**: an old SIC program still runs.

Book: Beck §1.3.2

The four instruction formats of SIC/XE

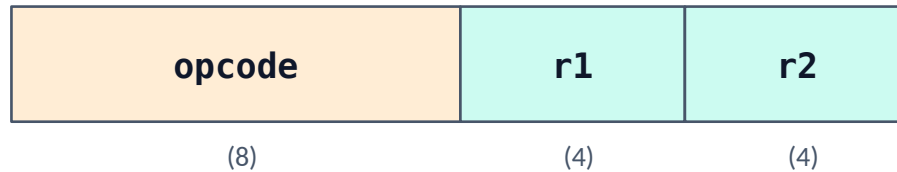
Format 1 • 1 byte

no operand, e.g. FIX, HIO



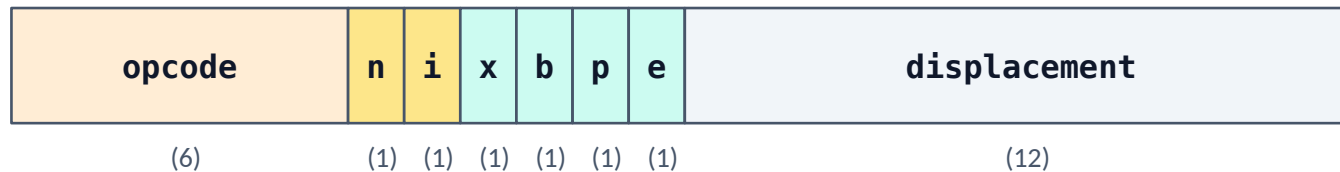
Format 2 • 2 bytes

two registers: ADDR S,A = 9040



Format 3 • 3 bytes

the common case: LDA ALPHA



Format 4 • 4 bytes

e = 1, written with a +: +JSUB RDREC



n, i say how to **use** the target address (simple, immediate, indirect). **x, b, p** say how to **compute** it (indexed, base, PC). **e** says format 3 or 4.

Book: Beck §1.3.2 (instruction formats)

Addressing modes, explained with C

Mode	Flag bits	SIC/XE	Rule	C analogy	In words
Simple	n=1, i=1	LDA ALPHA	operand = (TA)	a = alpha;	the usual case
Immediate	n=0, i=1	LDA #3	operand = TA itself	a = 3;	a constant inside the instruction
Indirect	n=1, i=0	LDA @P	operand = ((TA))	a = *p;	TA holds the address of the operand
Indexed	x=1	LDA TABLE,X	TA = disp + (X)	a = table[i];	X is the index
Base-relative	b=1	LDA FIELD	TA = (B) + disp	a = s->field;	base pointer + fixed offset
PC-relative	p=1	LDA ALPHA	TA = (PC) + disp	(no C syntax)	"600 bytes after this instruction"

SAME AS C

Immediate, indirect, indexed

are C's literal, *p and a[i]. You already use them every day.

DIFFERENT FROM C

PC-relative

has no C syntax. The assembler picks it; the program then works at any load address.

Book: Beck §1.3.2 (addressing modes)

Worked examples: six LDA instructions, six modes

Given: **(B) = 006000**, **(PC) = 003000** (already moved past the instruction), **(X) = 000090**.

Cover the orange columns, decode each row, then click to check.

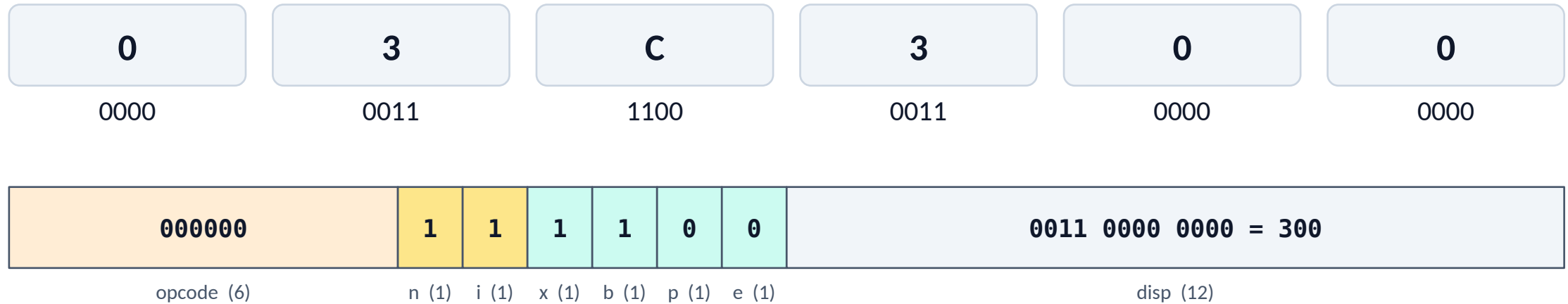
All LDA (opcode 00). Values checked with our SIC/XE decoder.

Address	Contents
3030	003600
3600	103000
6390	00C303
C303	003030

Machine code	n i x b p e	Mode	Target address	A after LDA
032600	1 1 0 0 1 0	PC-relative	$3000 + 600 = 3600$	103000
03C300	1 1 1 1 0 0	base + indexed	$6000 + 300 + 90 = 6390$	00C303
022030	1 0 0 0 1 0	indirect, PC-rel.	$3000 + 30 = 3030 \rightarrow 3600$	103000
010030	0 1 0 0 0 0	immediate	30	000030
003600	0 0 (SIC)	old SIC format	3600	103000
0310C303	1 1 0 0 0 1	format 4	C303	003030

Book: after the addressing-mode example in Beck §1.3.2

Decode 03C300 step by step



Opcode 000000 + two zero bits = 00

→ LDA

n = 1, i = 1

→ simple: the operand is (TA)

x = 1, b = 1, p = 0, e = 0

→ format 3, base-relative, indexed

TA = (B) + disp + (X)

→ 6000 + 300 + 90 = 6390

A ← (6390)

→ A = 00C303

PART 4

Real assembly: x86-64 on your lab PC

The same ideas on a real CPU: read what gcc writes, then write a program of your own with nasm.

SIC/XE and x86-64: same ideas, bigger machine

	SIC/XE (on paper)	x86-64 (lab PC)
Word size	3 bytes	8 bytes (registers), int = 4 bytes
Registers	A, X, L, B, S, T, F, PC, SW	16 general (rax ... r15), rip, rflags
Instruction length	1 to 4 bytes	1 to 15 bytes
Condition code	in SW: <, =, >	flags in rflags: ZF, SF, CF, OF
Function call	JSUB puts the return address in L	call pushes it on the stack; ret pops it
Addressing	simple, #, @, ,X, base, PC-relative	the same ideas: \$5, (%rax), 8(%rbp), (%rip)

Two spellings of the same CPU

AT&T (gcc output): `movl $0, %eax`

source first, then destination; % on registers, \$ on constants.

Intel (nasm, our labs): `mov eax, 0`

destination first, like C's `eax = 0;`

Good news for C programmers

x86-64 has a real **stack** with push, pop, call and ret. So functions, local variables and recursion work in hardware the way you know from C, unlike SIC's single L register.

Book: CS:APP §3.2–3.4, §3.7 (procedures and the stack)

Your C, compiled: if-else in x86-64

```
maxab.c

int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Same pattern as Program 1 on SIC: compare, jump, two paths meeting at one label.

```
gcc -O0 -S maxab.c (AT&T)

max:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    cmpl    -8(%rbp), %eax
    jle     .L2
    movl    -4(%rbp), %eax
    jmp     .L3
.L2:
    movl    -8(%rbp), %eax
.L3:
    popq    %rbp
    ret
```

Parameters arrive in registers

a in **edi**, b in **esi**: the calling convention. gcc -O0 first copies them into the stack frame at -4 and -8 from rbp, like C local variables.

cmpl, then jle

Compare, then jump: exactly SIC's COMP and JGT. But gcc jumps on the **opposite** test, $a \leq b$, straight to the else part (.L2).

Labels .L2, .L3

The if-else became **two labels and two jumps**. The result goes in **eax** for the return.

Book: CS:APP §3.6 (control)

LAB 2

Write it yourself: nasm on the lab PC

No C, no library: just your instructions and two system calls.

Program 4: hello world without C

```
hello.asm (Intel syntax, nasm)

    global _start
    section .data
msg:  db      "Hello from assembly!", 10
len:  equ     $ - msg
    section .text
_start:
    mov     rax, 1      ; write
    mov     rdi, 1      ; to stdout
    mov     rsi, msg    ; this text
    mov     rdx, len    ; this many bytes
    syscall
    mov     rax, 60     ; exit
    mov     rdi, 0      ; status 0
    syscall
```

```
bash

$ nasm -f elf64 hello.asm -o hello.o
$ ld hello.o -o hello
$ ./hello
Hello from assembly!
$ strace ./hello
execve(... write(1, ...) = 21 exit(0)
```

C idea: `write(1, msg, len); exit(0);` The four registers are the system call's arguments, in order.

db, equ = directives: Like SIC's BYTE and EQU. `$ - msg` = "here minus the start": the assembler computes the length, 21.

3 system calls, not 34: No C library is loaded. Compare Lecture 1: `printf`'s hello needed 34.

Program 5: the for-loop, written by hand

```
C
int s = 0;
for (int i = 1;
     i <= 10; i++)
    s = s + i;
return s; /* 55 */
```

SAME AS C

Same loop shape

Body, increment, compare, jump back.
The same shape as SIC's TIX + JLT loop.

```
sum.asm
_start:
    mov    ecx, 1      ; i = 1
    mov    eax, 0      ; s = 0
loop:
    add    eax, ecx     ; s = s + i
    inc    ecx         ; i++
    cmp    ecx, 10     ; i vs 10
    jle    loop        ; if i <= 10
    mov    edi, eax     ; status = s
    mov    eax, 60      ; exit
    syscall
```

DIFFERENT FROM C

Variables are registers

s and i never touch memory: eax and ecx are the variables. Only comments say which is which.

```
bash + gdb
$ ./sum ; echo $?
55
# gdb: break at loop
round 1: eax=0  ecx=1
round 2: eax=1  ecx=2
round 3: eax=3  ecx=3
# objdump: jle = 7e f7
# f7 = -9: PC-relative!
```

When things go wrong in assembly

AT&T syntax in nasm

```
bash
$ nasm -f elf64 e2.asm -o e2.o
e2.asm:4: error: parser: instruction expected
```

Line 4 was `movl $1, %eax` (gcc style). nasm wants Intel: `mov eax, 1`.

Forgot global _start

```
bash
$ ld e1.o -o e1
ld: warning: cannot find entry symbol _start; defaulting to 401000
```

It ran only by luck: `_start` happened to be first. Always write `global _start`.

No exit system call

```
bash
$ ./e3
Segmentation fault
$ echo $?
139
```

There is no return from `_start`. The CPU runs on into garbage bytes. End with the `exit` syscall.

Result too big

```
bash
$ ./sum30 ; echo $?
209
```

$1 + \dots + 30 = 465$, but an exit status is only **8 bits**: $465 - 256 = 209$. Like storing 465 in an unsigned char.

C link: assembly gives you C's freedom with none of its safety nets. No compiler warns you about a missing return or a value that overflows.

Lab 2 tasks, step by step

1

Hand-assemble

Encode `LDA ALPHA, STCH BUFFER, X` and `TIX MAXLEN` (ALPHA = 1033, BUFFER = 1039, MAXLEN = 1036). Show all six steps.

2

Trace on paper

Trace Program 2 (string copy) with the string "CAT" and FIVE replaced by THREE. Write X, A and CC after every instruction.

3

Read gcc output

`gcc -O0 -S sumto.c`. Mark which lines are `s = 0, i = 1, i <= n, s += i, i++`. Repeat with `-masm=intel`.

4

Build hello.asm

Type it, build with `nasm` and `ld`, run it. Change the message. Run `strace` and count the calls.

5

Build sum.asm

Change it to add 1 ... 20. Check the answer with `echo $?`. Step through two rounds in `gdb`.

6

Break it on purpose

Remove the `exit` syscall, then `global _start`. Record each message and explain it in one sentence.

Submit: your lab notebook with commands, outputs, the trace table from task 2, and one sentence per task.

Checkpoint: test yourself

- Q1 Which C statement maps one-to-one onto a single SIC instruction? → goto → J
- Q2 Encode STA BETA, with STA = 0C and BETA at 1018. → 0C1018
- Q3 What two jobs does TIX do? → $X \leftarrow X + 1$, then compare X with the operand
- Q4 In SIC/XE, $n = 1$, $i = 0$. Which C expression is this like? → *p (indirect: the word at TA is an address)
- Q5 In x86-64 Linux, which registers carry a function's first argument and its result? → first argument in rdi (edi), result in rax (eax)

Key ideas from today



One assembly line = one machine instruction. One C line can be many.



C's ideas survive: variables become labelled words, if and for become compares and jumps.



C's safety nets do not: no types, few registers, only jumps, no automatic stack in SIC.



A SIC instruction is opcode 8 + x 1 + address 15 bits. Encode it in six steps.



SIC/XE: four formats; n i x b p e decide how the target address is found and used.



x86-64 uses the same ideas, plus a real stack. gcc writes AT&T, nasm reads Intel.

Homework 2 (due before Lecture 3)

1. C $\rightarrow$ SIC

Translate into SIC with directives: `if (num < 0) res = 0 - num; else res = num;` Trace it for `num = -9` and for `num = 4`.

2. Encode

Encode `LDX ZERO, JLT MOVECH` and `STCH STR2, X` from Program 2 using the addresses on slide 15's listing (`ZERO = 101C`, `MOVECH = 1003`, `STR2 = 1017`).

3. Decode

With `(B) = 006000`, `(PC) = 003000`, `(X) = 000090`, `(2FF6) = 000777`, `(60A0) = 000ABC`, decode `032FF6`, `07C010`, `01003C`, `4B101036`. Give mode, TA and result.

4. Read gcc

Compile `int abs_val(int n) { if (n < 0) return -n; return n; }` with `gcc -O0 -S`. Which compare and which jump does gcc use? Why that jump?

5. Think

`sum.asm` with `i  $\leq$  30` exits with 209, not 465. Explain why, and say what C type behaves the same way.

Next lecture and reading

NEXT: LECTURE 3.1

Basic assembler functions

We build the program that did today's hand encoding for us.

Pass 1: give every label an address (the location counter and SYMTAB).

Pass 2: encode every line (OPTAB) and write the object program.

Main text

L. L. Beck, **System Software**, 3rd ed., 1997. Today: §1.3.1–1.3.3 and Appendix A. Read §2.1 before Lecture 3.1.

x86-64

R. E. Bryant & D. R. O'Hallaron, **Computer Systems: A Programmer's Perspective**, 3rd ed., 2016. Today: §3.1–3.6.

nasm

The **NASM manual** (nasm.us), ch. 2–3: running nasm and the language. Offline on the lab PC: `man nasm`.

System calls

M. Kerrisk, **The Linux Programming Interface**, 2010, ch. 3 (system calls).