Why Learn Assembly Languages

Zonghua Gu

Fall 2025

IEEE Spectrum's Top Programming Languages 2022
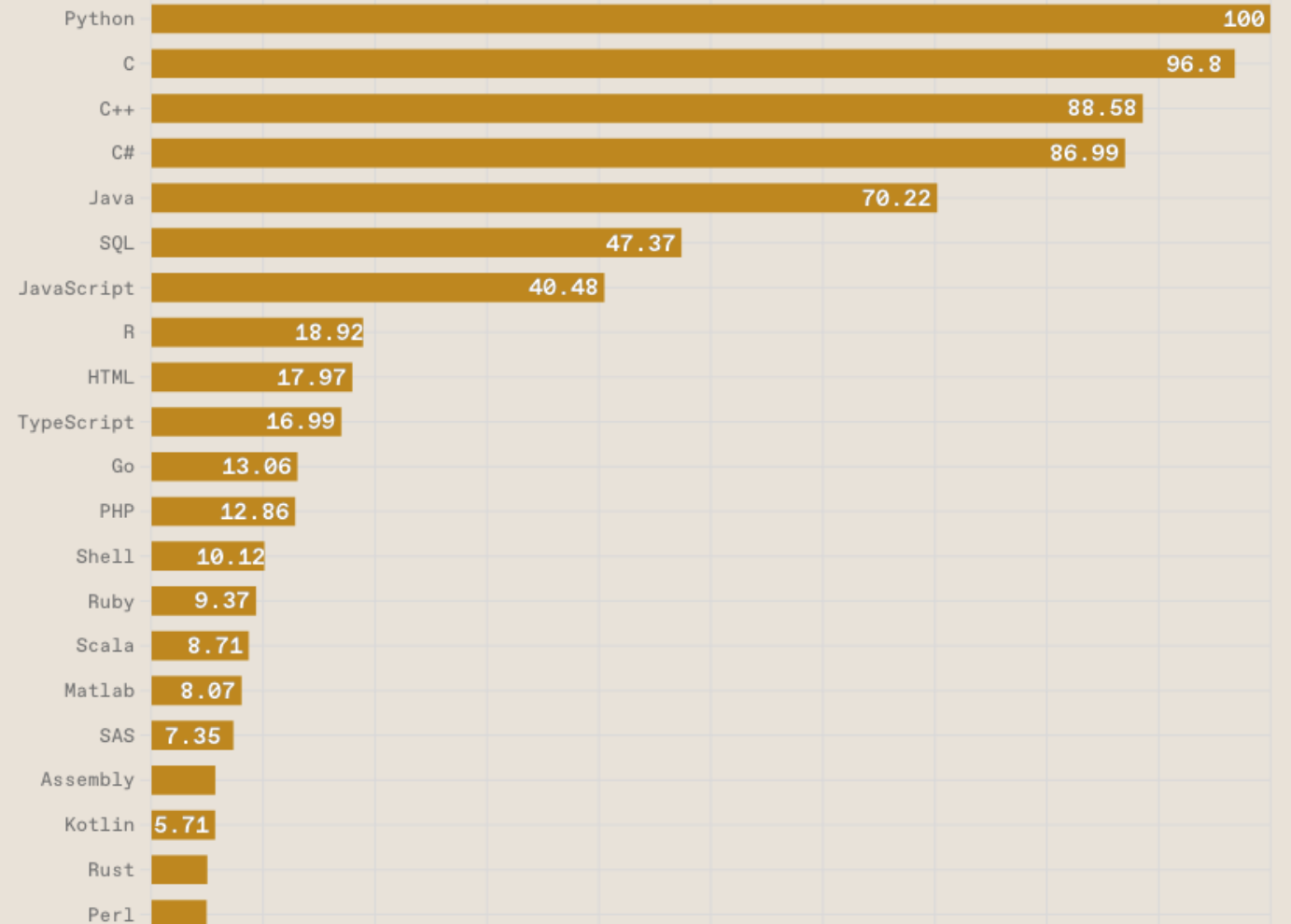
https://spectrum.ieee.org/top-programming-languages-2022

Top Programming Languages 2022

Click a button to see a differently weighted ranking

Spectrum | Jobs | Trending

| Language	Value
Python	100
C	96.8
C++	88.58
C#	86.99
Java	70.22
SQL	47.37
JavaScript	40.48
R	18.92
HTML	17.97
TypeScript	16.99
Go	13.06
PHP	12.86
Shell	10.12
Ruby	9.37
Scala	8.71
Matlab	8.07
SAS	7.35
Assembly	
Kotlin	5.71
Rust	
Perl	

Assembly: Not just another language

- Assembly vs high level languages (HLLs).
  - Most embedded systems are programmed in HLLs
  - Assembly disadvantages
    - difficult to develop, read, and maintain
    - bug prone
    - not portable

- However, assembly isn't "just another language".
 - Interface between hardware and software
  - Implements high-level languages

High-Level Language

Assembly Language

Machine Language

Hardware
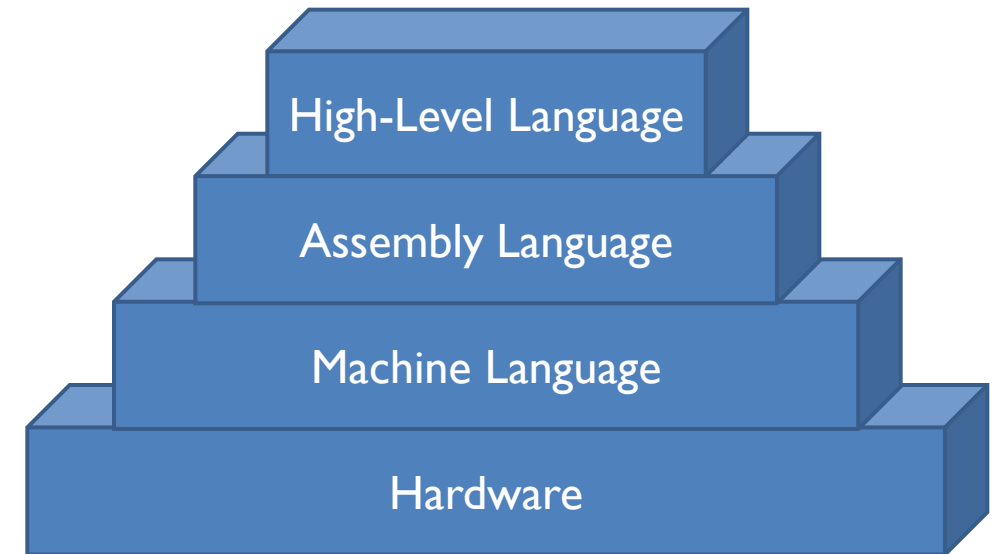
Assembly: Learn how processors work

‣ Learn about the inner workings of a processor

  ‣ Data representation
  ‣ Registers
  ‣ Computer arithmetic
  ‣ Memory addressing
  ‣ Instruction set
  ‣ I/O

‣ Provide background knowledge for later courses

  ‣ computer architecture,
  ‣ operating systems,
  ‣ compiler
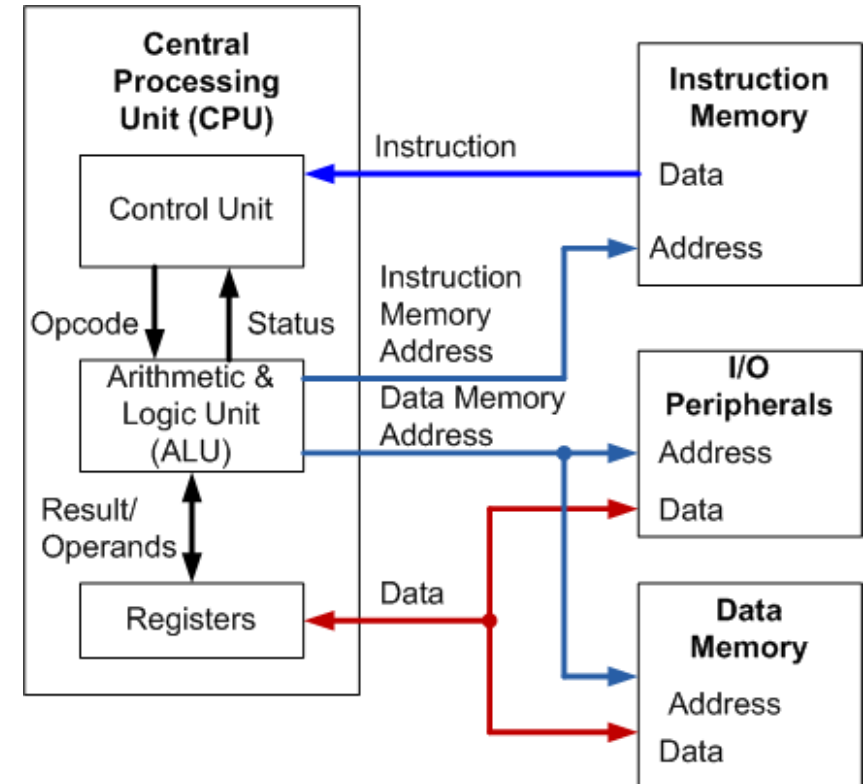
Assembly: Faster and smaller

- Assembly program runs faster than HLLs.
  - Performance critical codes must be written in assembly.
  - Use profiling tools to find the performance bottleneck and rewrite that code section in assembly
  - Latency-sensitive applications, such as aircraft controller
  - Some C compilers do not use some special Thumb instructions, such as ROR (Rotate Right) and RRX (Rotate Right Extended).

brilliant.org

- Cost-sensitive applications
  - Assembly consumes little memory
  - Embedded devices, where the size of code is limited, wash machine controller, automobile controllers
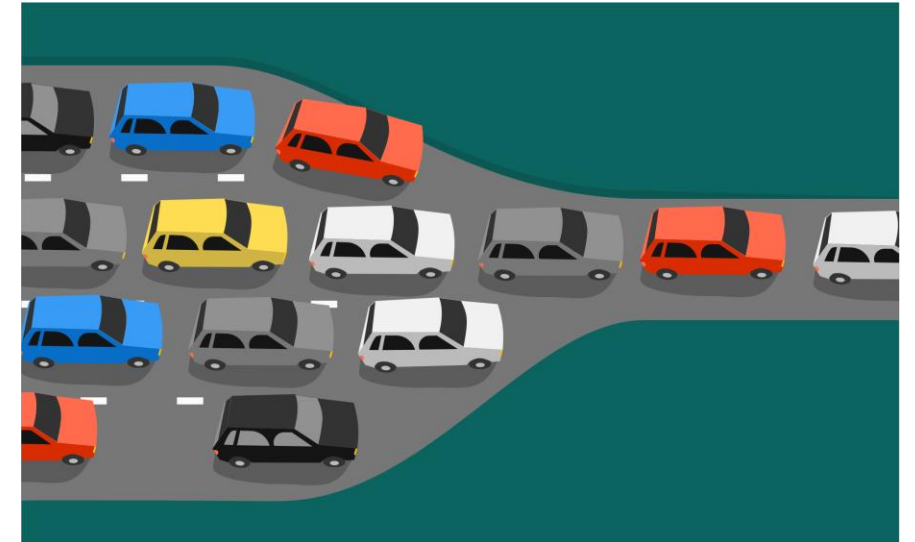
Assembly: The only choice sometimes

▸ Hardware/processor specific code,

 ▸ Special instructions not supported by a compiler

  ▸ **CPSID I** *; Disable IRQ by setting PRIMASK*
  ▸ **CPSIE I** *; Enable IRQ by clearing PRIMASK*
  ▸ **MSR/MSR** *; Read/write to special registers*
  ▸ **WFI** *; Enter low-power & wait for interrupt*

```
// Enable Interrupts
__attribute__( ( always_inline ) )
static inline void __enable_irq(void)
{
    __asm("cpsie i");
}


// Disable Interrupts
__attribute__( ( always_inline ) )
static inline void __disable_irq(void)
{
    __asm("cpsid i");
}
```

cmsis_armcc.h

Assembly: The only choice sometimes

▶ **Hardware/processor specific code,**

  ▸ Special instructions not supported by a compiler

    ▸ **CPSID I** ; *Disable IRQ by setting PRIMASK*

    ▸ **CPSIE I** ; *Enable IRQ by clearing PRIMASK*

    ▸ **MSR/MSR** ; *Read/write to special registers*

    ▸ **WFI** ; *Enter low-power & wait for interrupt*

  ▸ Startup Code

    ▸ the stack and heap areas

    ▸ interrupt vector table

    ▸ default implementation of ISRs

    ▸ written in assembly, and possible in C with inline assembly

    ▸ C version is toolchain dependent

```
...
Stack_Size EQU 0x400;
           AREA STACK,NOINIT,READWRITE,ALIGN=3
Stack_Mem  SPACE Stack_Size
__initial_sp


Heap_Size EQU 0x200;
           AREA HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem SPACE Heap_Size
__heap_limit


__Vectors
   DCD __initial_sp          ; Top of Stack
   DCD Reset_Handler         ; Reset Handler
   DCD NMI_Handler           ; NMI Handler
   DCD HardFault_Handler     ; Hard Fault Handler
   DCD MemManage_Handler     ; MPU Fault Handler
   DCD BusFault_Handler      ; Bus Fault Handler
   DCD UsageFault_Handler    ; Usage Fault Handler
   ...
```

startup_stm32l476xx.s

Assembly: The only choice sometimes

▸ **Hardware/processor specific code,**

  ▸ Special instructions not supported by a compiler
    ▸ **CPSID I** ; *Disable IRQ by setting PRIMASK*
    ▸ **CPSIE I** ; *Enable IRQ by clearing PRIMASK*
    ▸ **MSR/MRS** ; *Read/write to special registers*
    ▸ **WFI** ; *Enter Low-power & wait for interrupt*
  ▸ Startup code
    ▸ the stack and heap areas
    ▸ interrupt vector table
    ▸ default implementation of ISRs
    ▸ written in assembly, and possible in C with inline assembly
    ▸ C version is toolchain dependent
  ▸ **Device driver**
    ▸ access machine-dependent registers and I/O
    ▸ control exact code behavior in critical sections

```c
static inline u8 __raw_readb(const volatile
void __iomem *addr) {
  u8 val;
  asm volatile("ldrb %0, %1"
    : "=r" (val)
    : "Qo" (*(volatile u8 __force *)addr));
  return val;
}


static inline void __raw_writeb(u8 val,
volatile void __iomem *addr) {
  asm volatile("strb %1, %0"
    : : "Qo" (*(volatile u8 __force *)addr),
      "r" (val));
}
```

Linux 5.6, /arch/arm/include/asm/io.h

Assembly: Help you write better HLLs

▶ Help you understand and write HLLs better
  ▶ Low-level data representation
  ▶ C Pointers
    ▶ Reference & dereference
  ▶ Passing parameters to functions:
    ▶ Pass by reference
    ▶ Pass by value
  ▶ Variables declared as **volatile** or **static**
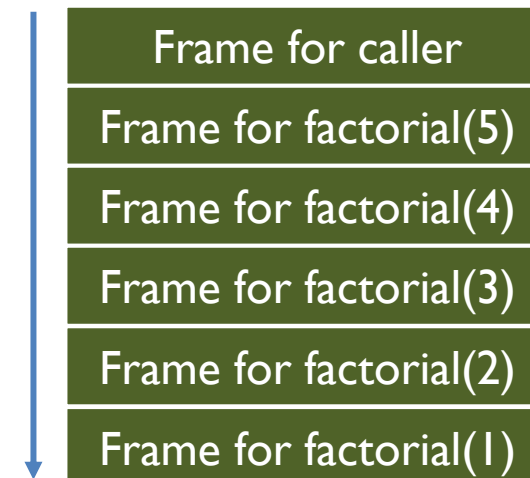 ▶ Inefficiency of recursive function: stack operations

```
*p += 100
```

```
LDR r0, =p
LDR r1, [r0]
ADD r1, r1, #100
STR r1, [r0]
```

```
int* pi;
int volatile* pvi;
int* volatile vpi;
```

| Frame for caller |
| Frame for factorial(5) |
| Frame for factorial(4) |
| Frame for factorial(3) |
| Frame for factorial(2) |
| Frame for factorial(1) |

Stack grows in recursive phase

Stack shrinks in regression phase

Assembly: Help you write better HLLs

```c
uint32_t x = 1;
int32_t  y = -1;

if (x > y)
    printf("Of course.")
else
    printf("Something is wrong!");
```

Output:
Something is wrong!

r0 = 0x00000001 ; x = 1
r1 = 0xFFFFFFFF ; y = -1

```
CMP r0, r1
BLS else   ; Branch on Unsigned Lower than or Same
```

```
compiling main.c...
linking...
Program Size: Code=640 RO-data=424 RW-data=8 ZI-data=5472
FromELF: creating hex file...
".\Objects\project.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

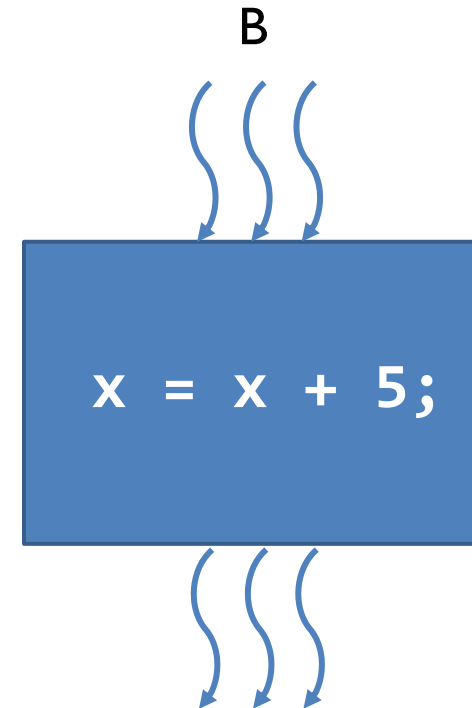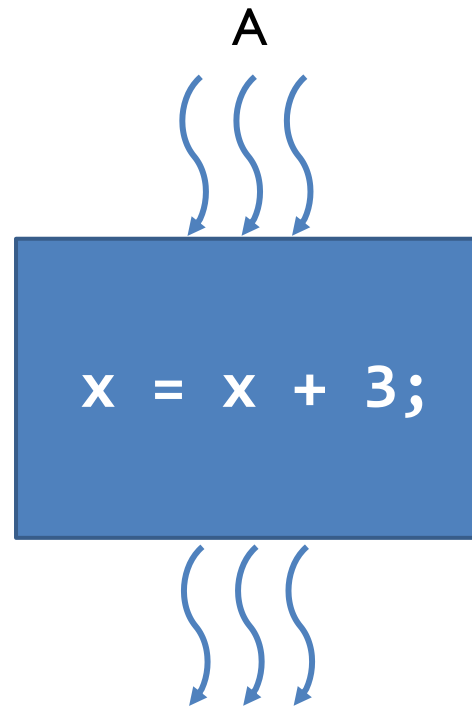# Assembly: Help you write better HLLs

```
uint32_t x = 1;
int32_t  y = -1;

if ((int32_t) x > y)
    printf("Of course.")
else
    printf("Something is wrong!");
```
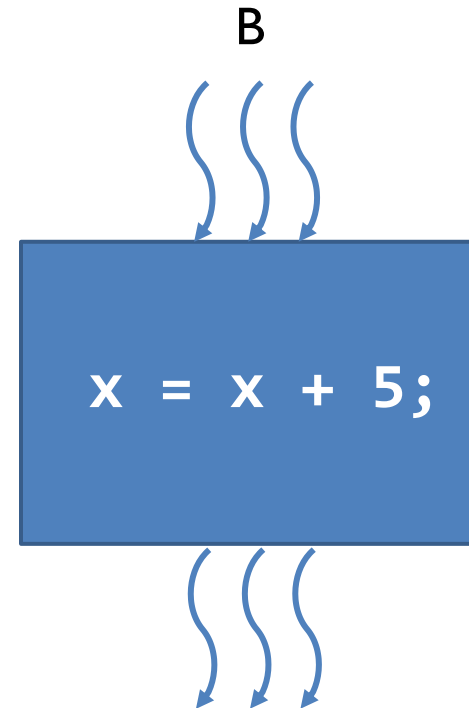
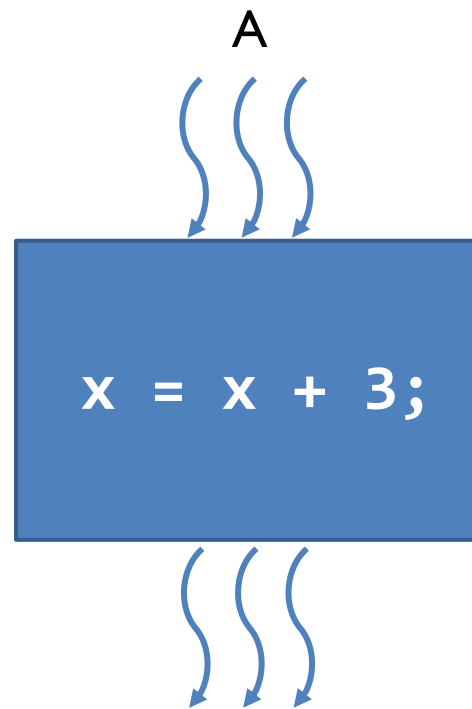# Assembly: Help you write better HLLs

`x = 1;`

A

`x = x + 3;`

B

`x = x + 5;`

What is the final value of x?
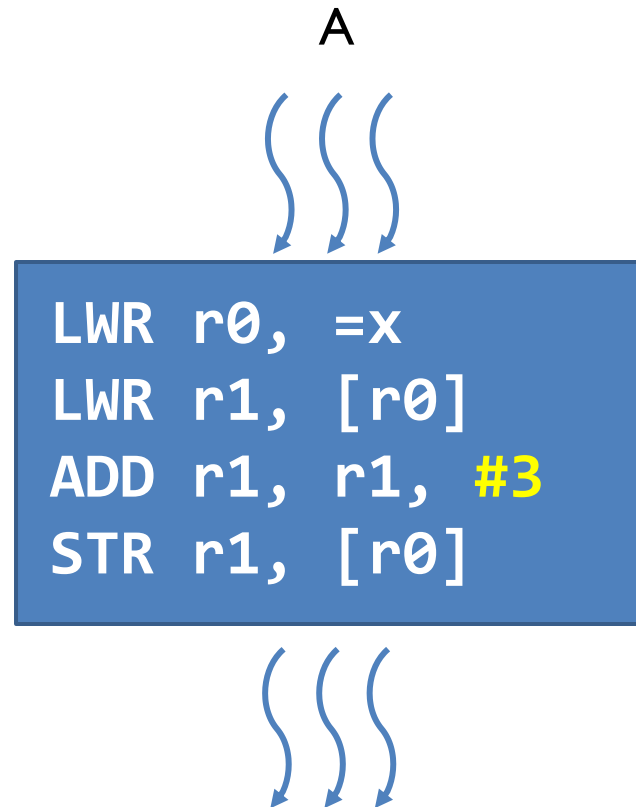
Assembly: Help you write better HLLs

x = 1;

A

x = x + 3;

B

x = x + 5;
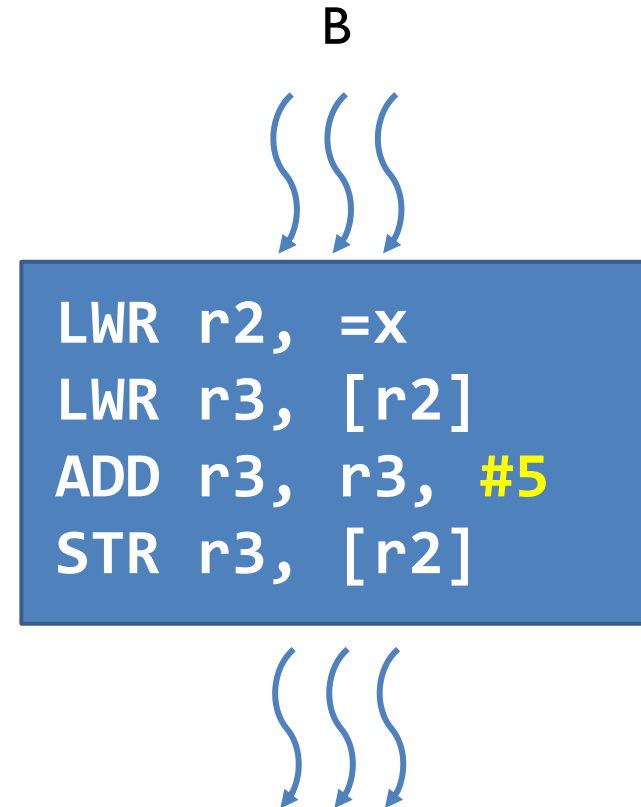
x = 4, 6, 9;

Assembly: Help you write better HLLs

x = 1;

A

```
LWR r0, =x
LWR r1, [r0]
ADD r1, r1, #3
STR r1, [r0]
```

B

```
LWR r2, =x
LWR r3, [r2]
ADD r3, r3, #5
STR r3, [r2]
```

x = 4, 6, 9;

Assembly: Help you write better HLLs

x = 1;

A

B

```
LWR r0, =x
LWR r1, [r0]
ADD r1, r1, #3
STR r1, [r0]
```

x = 4

```
LWR r2, =x
LWR r3, [r2]
ADD r3, r3, #5
STR r3, [r2]
```
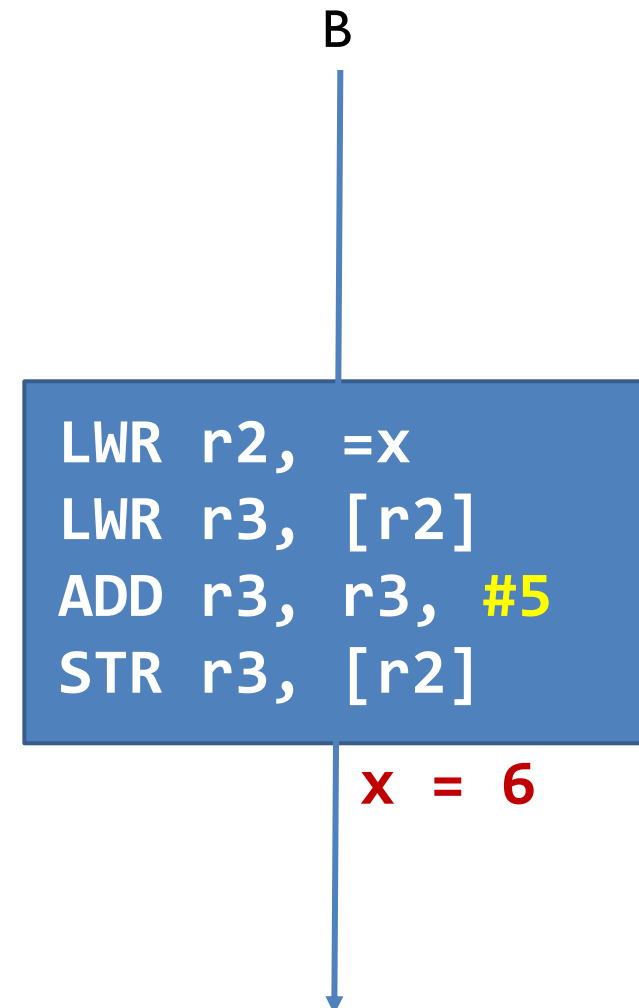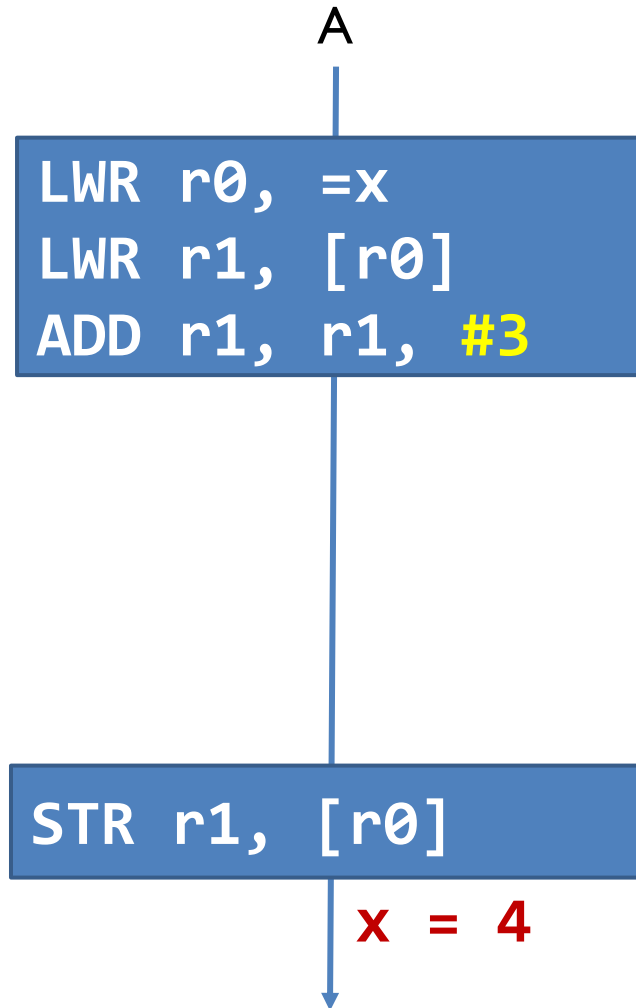
x = 9

x = 9

Assembly: Help you write better HLLs

x = 1;

A   B

```
LWR r0, =x
LWR r1, [r0]
ADD r1, r1, #3
```

```
LWR r2, =x
LWR r3, [r2]
ADD r3, r3, #5
STR r3, [r2]
```

x = 6

```
STR r1, [r0]
```

x = 4

x = 4

Assembly: Help you write better HLLs

x = 1;

A

B

```
LWR r2, =x
LWR r3, [r2]
ADD r3, r3, #5
```

```
LWR r0, =x
LWR r1, [r0]
ADD r1, r1, #3
STR r1, [r0]
```

x = 4

```
STR r3, [r2]
```

x = 6
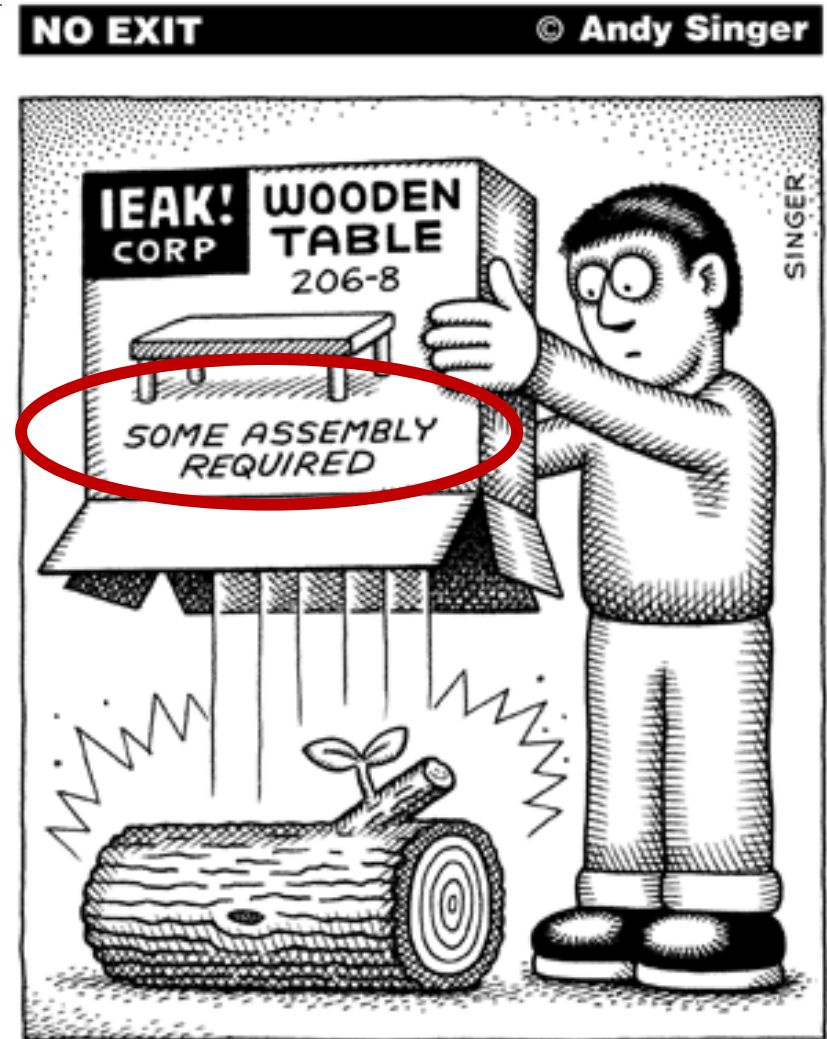
x = 6

Why should we learn Assembly?

▶ Assembly isn't "just another language".

  ▶ Help you understand how does the processor work

▶ Assembly program runs faster than high-level language.

  ▶ Performance critical codes may need to be written in assembly.

  ▶ Use the profiling tools to find the performance bottle and rewrite that code section in assembly

  ▶ Latency-sensitive applications, such as aircraft controller

  ▶ Standard C compilers do not use some operations available on ARM processors, such ROR (Rotate Right) and RRX (Rotate Right Extended).

▶ Hardware/processor specific code,

  ▶ Processor booting code

  ▶ Device drivers

  ▶ Compiler, assembler, linker

  ▶ A test-and-set atomic assembly instruction can be used to implement locks and semaphores.

▶ Cost-sensitive applications

  ▶ Embedded devices, where the size of code is limited, wash machine controller, automobile controllers

▶ Better understand high-level programming languages

Recap: Why Learn Assembly?

- Gain insights about what is under the hood of a processor
- Assembly should be used for performance critical sessions
- Assembly must be used for processor-dependent instructions that are not supported by compilers
- Understanding assembly helps us write better HLLs

NO EXIT © Andy Singer

IEAK! CORP WOODEN TABLE 206-8

SOME ASSEMBLY REQUIRED

SINGER

www.andysinger.com