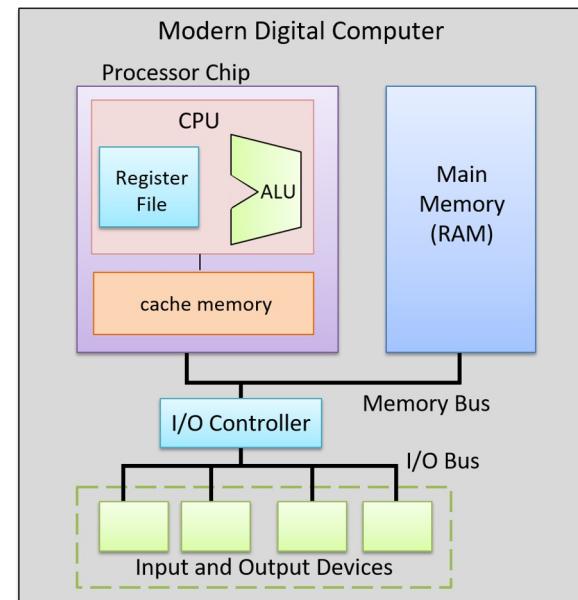
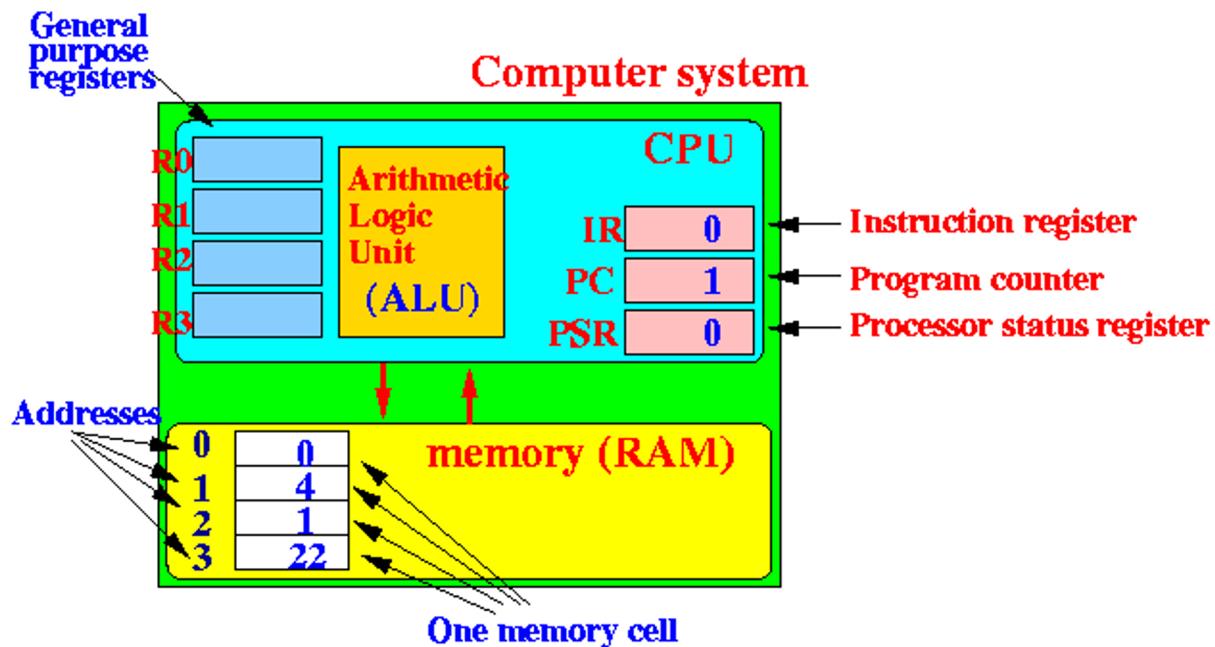
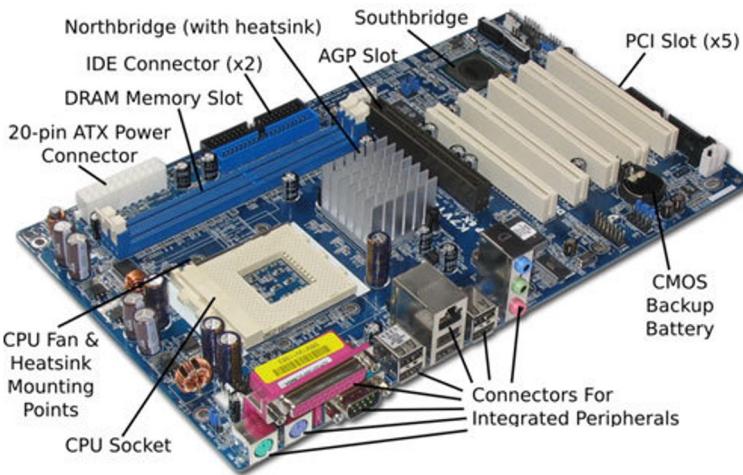

Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

Unit 1: Module Overview, Instruction Operations and Operands

ITSC 2181 - Introduction to Computer Systems
College of Computing and Informatics

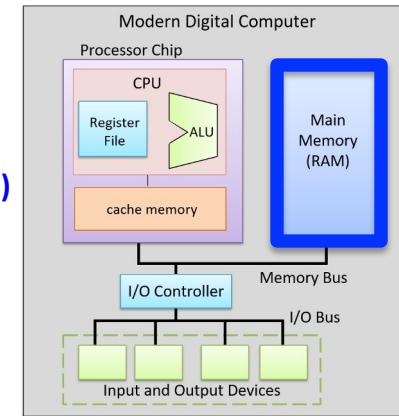
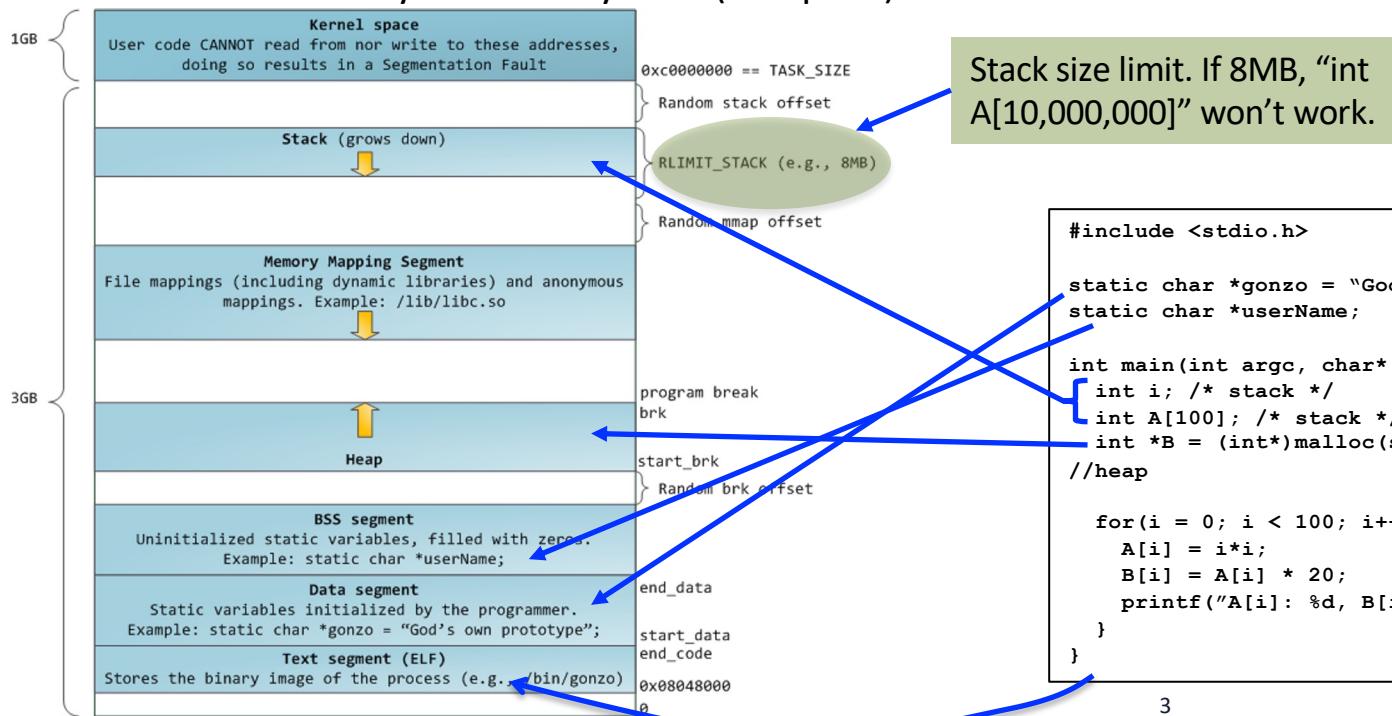
Hardware Components of a Computer



Memory Layout of a Program

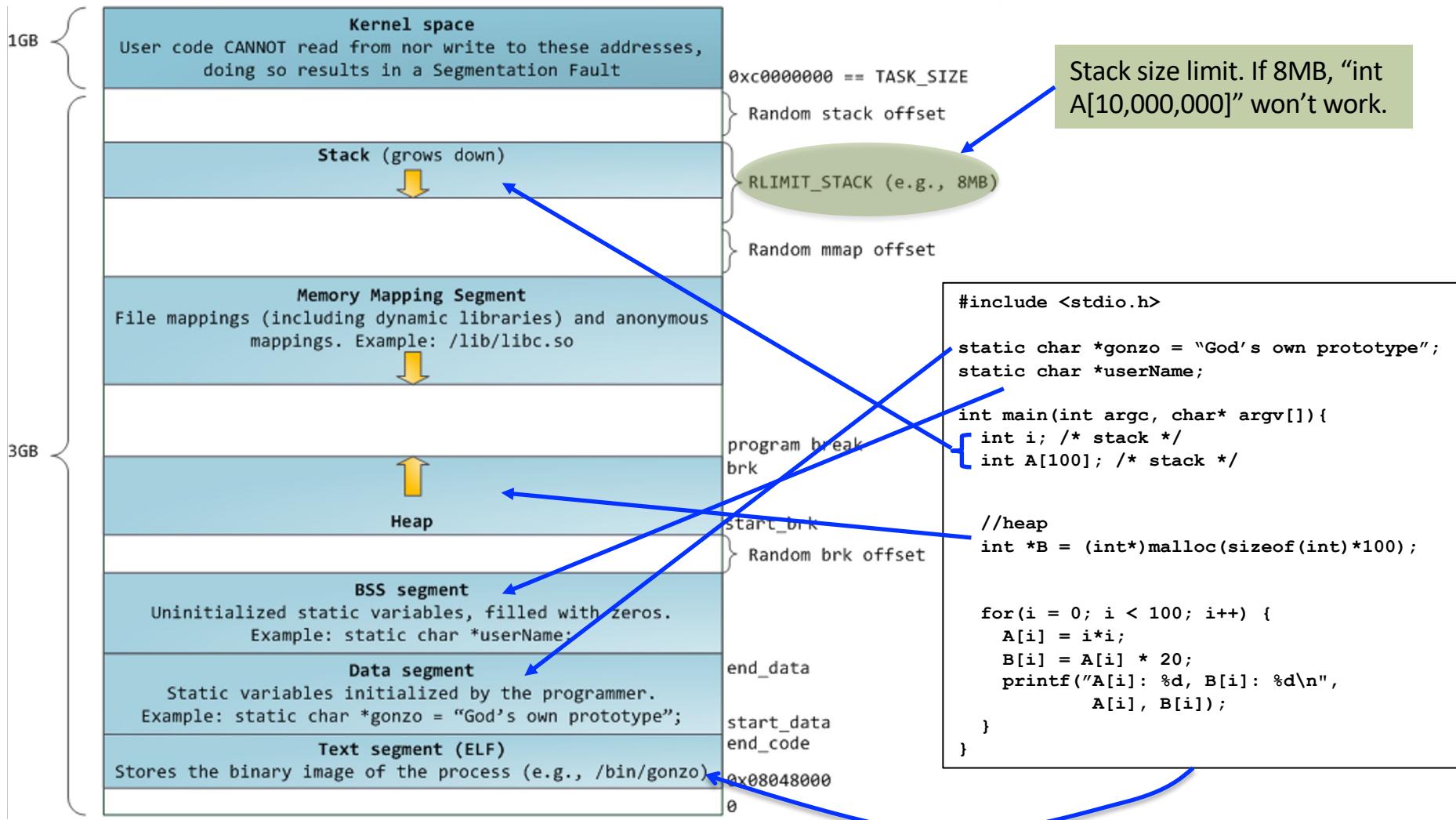
- A program includes both code and variables
 - In system: that becomes instructions and data, all are stored in memory
- Code (machine instructions) → Text segment
- Static variables → Data or BSS segment
- Function variables → stack (i, A[100] and B)
 - A is a variable that stores memory address, the memory for A's 100 int elements is in the stack
 - B is a memory address, it is stored in stack, but the memory B points to is in heap (100 int elements)
- Dynamic allocated memory using malloc or C++ “new” → heap (B[100])

Linux Process Memory in 32-bit System (4G space)



Memory Layout of a Program (cont'd)

Linux Process Memory in 32-bit System (4G space)



Levels of Program Code

- High-level language
 - Level of abstraction closer to problem domain
 - Provides for **productivity and portability**
- Machine-Level Code:
Instructions
 - **Textual representation of binary machine instructions**
 - **Interface between HW and SW**
 - **Assembly language**
- Hardware representation
 - **Binary digits (bits)**
 - **Encoded instructions and data**

High-level
language
program
(in C)

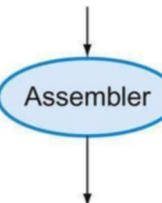
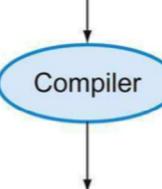
```
swap(size_t v[], size_t k)
{
    size_t temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly
language
program
(for RISC-V)

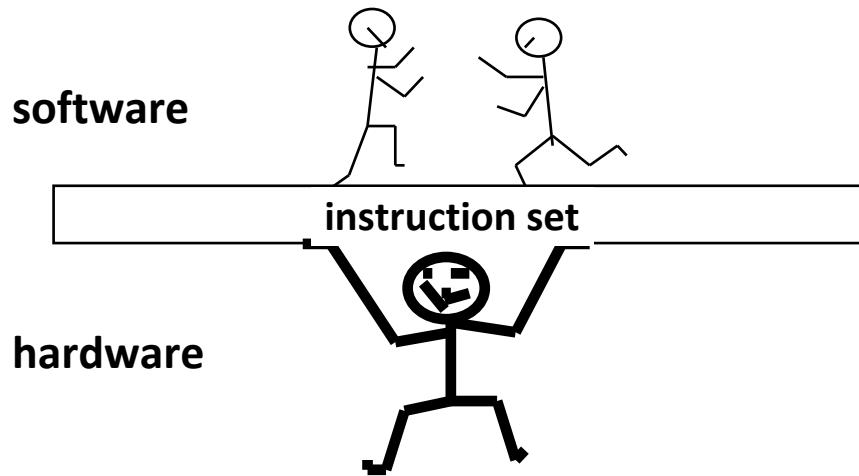
```
swap:
    slli x6, x11, 3
    add x6, x10, x6
    ld x5, 0(x6)
    ld x7, 8(x6)
    sd x7, 0(x6)
    sd x5, 8(x6)
    jalr x0, 0(x1)
```

Binary machine
language
program
(for RISC-V)

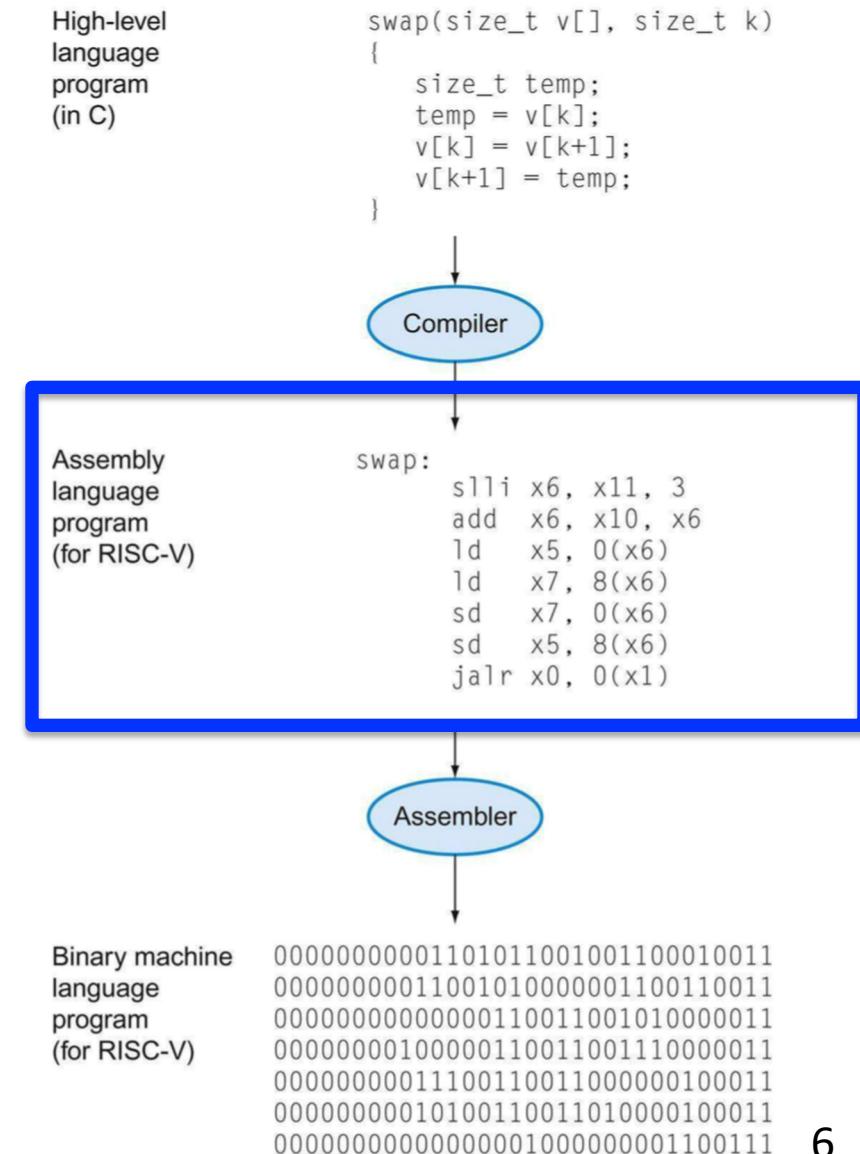
```
000000000001101011001001100010011
0000000000011001010000001100110011
0000000000000000110011001010000011
000000001000001100110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
0000000000000000100000000110001111
```



Instruction Set Architecture: The Interface Between Hardware and Software

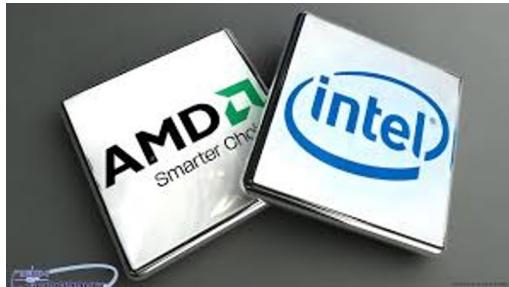


- The words of a computer language are called instructions, and its vocabulary/dictionary is called an instruction set
 - lowest software interface, assembly level, to the users or to the compiler writer
 - **Instruction Set Architecture** – A type of computer



Major Types of ISA (Computers)

- X86: Intel and AMD, Desktop, laptop, server market



- ARM: embedded, pad/phone, etc, now laptop/server



- RISC-V: fast growing one, embedded so far
- Power (mainly IBM) and SPARC (mainly Oracle and Fujitsu): server market

Levels of Program Code to Multiple Target Architectures

High-level language program (in C)

```
swap(size_t v[], size_t k)
{
    size_t temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Compiler

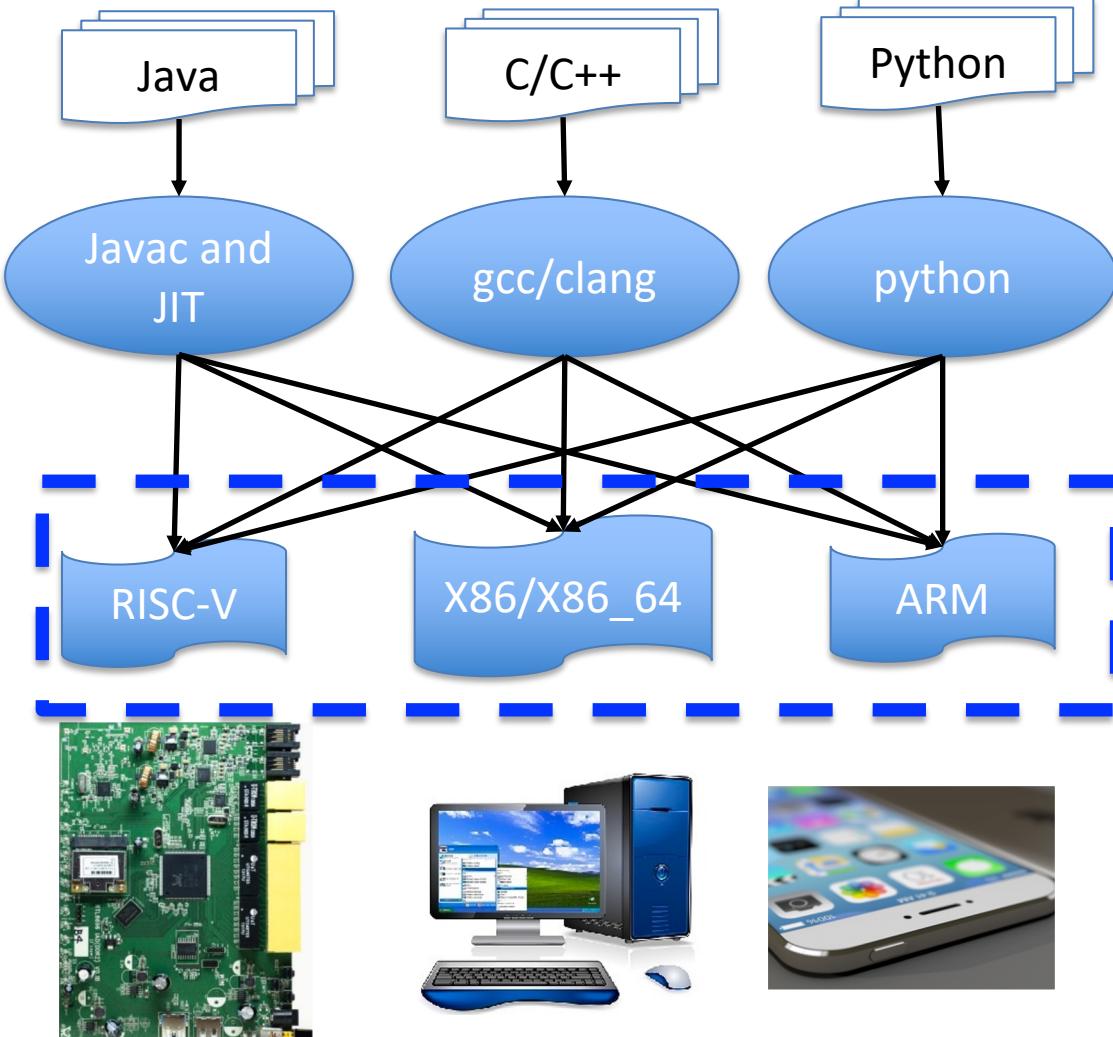
Assembly language program (for RISC-V)

```
swap:
    slli x6, x11, 3
    add x6, x10, x6
    ld x5, 0(x6)
    ld x7, 8(x6)
    sd x7, 0(x6)
    sd x5, 8(x6)
    jalr x0, 0(x1)
```

Assembler

Binary machine language program (for RISC-V)

```
0000000000001101011001001100010011
0000000000011001010000001100110011
0000000000000000110011001010000011
000000000100000110011001110000011
00000000001100110011000000100011
000000000010100110011010000100011
00000000000000001000000001100111
```



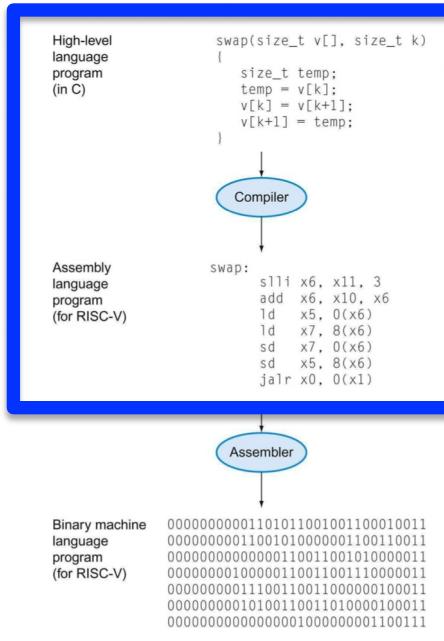
X86_64 Assembly Example

Using “-S” compiler flag to translate high-level code to assembly instructions

```
yanyh@vm:~$ uname -a
Linux vm 4.4.0-170-generic #199-Ubuntu SMP Thu Nov 14 01:45:04 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
yanyh@vm:~$ gcc -S swap.c
yanyh@vm:~$ cat swap.s
```

```
.file    "swap.c"
.text
.globl  swap
.type   swap, @function

swap:
.LFB0:
.cfi_startproc
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq   %rsp, %rbp
.cfi_def_cfa_register 6
movq   %rdi, -24(%rbp)
movl   %esi, -28(%rbp)
movl   -28(%rbp), %eax
cltq
leaq    0(%rax,4), %rdx
movq   -24(%rbp), %rax
addq   %rdx, %rax
movl   (%rax), %eax
movl   %eax, -4(%rbp)
movl   -28(%rbp), %eax
cltq
leaq    0(%rax,4), %rdx
movq   -24(%rbp), %rax
addq   %rax, %rdx
movl   -28(%rbp), %eax
cltq
addq   $1, %rax
leaq    0(%rax,4), %rcx
movq   -24(%rbp), %rax
```



- X86_64 is ISA Architecture for most Intel and AMD desktop/server CPUs
- RISC-V is one ISA
- ARM is another ISA
 - Most cellphone/smartphone are ARM CPUs

Try the highlighted command for swap.c from the terminal of
<https://repl.it/languages/c>

<https://passlab.github.io/ITSC3181/exercises/swap/>

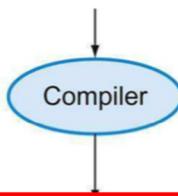
X86_64 Assembly Example

<https://repl.it/languages/c>

Disassembly a machine binary code to assembly instructions using “objdump”

High-level
language
program
(in C)

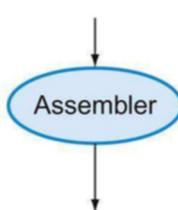
```
swap(size_t v[], size_t k)
{
    size_t temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



Assembly
language
program
(for RISC-V)

```
swap:
    slli x6, x11, 3
    add x6, x10, x6
    ld x5, 0(x6)
    ld x7, 8(x6)
    sd x7, 0(x6)
    sd x5, 8(x6)
    jalr x0, 0(x1)
```

Disassembly



Binary machine
language
program
(for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

```
yanyh@vm:~$ gcc -c swap.c
yanyh@vm:~$ objdump -D swap.o
```

swap.o: file format elf64-x86-64

Disassembly of section .text:

```
0000000000000000 <swap>:
0: 55                      push  %rbp
1: 48 89 e5                mov   %rsp,%rbp
4: 48 89 7d e8              mov   %rdi,-0x18(%rbp)
8: 89 75 e4                mov   %esi,-0x1c(%rbp)
b: 8b 45 e4                mov   -0x1c(%rbp),%eax
e: 48 98                  cltq 
10: 48 8d 14 85 00 00 00   lea   0x0(%rax,4),%rdx
17: 00
18: 48 8b 45 e8              mov   -0x18(%rbp),%rax
1c: 48 01 d0                add   %rdx,%rax
1f: 8b 00                  mov   (%rax),%eax
21: 89 45 fc                mov   %eax,-0x4(%rbp)
24: 8b 45 e4                mov   -0x1c(%rbp),%eax
27: 48 98                  cltq 
29: 48 8d 14 85 00 00 00   lea   0x0(%rax,4),%rdx
30: 00
31: 48 8b 45 e8              mov   -0x18(%rbp),%rax
35: 48 01 c2                add   %rax,%rdx
38: 8b 45 e4                mov   -0x1c(%rbp),%eax
3b: 48 98                  cltq 
3d: 48 83 c0 01              add   $0x1,%rax
41: 48 8d 0c 85 00 00 00   lea   0x0(%rax,4),%rcx
48: 00
49: 48 8b 45 e8              mov   -0x18(%rbp),%rax
4d: 48 01 c8                add   %rcx,%rax
50: 8b 00                  mov   (%rax),%rax
```

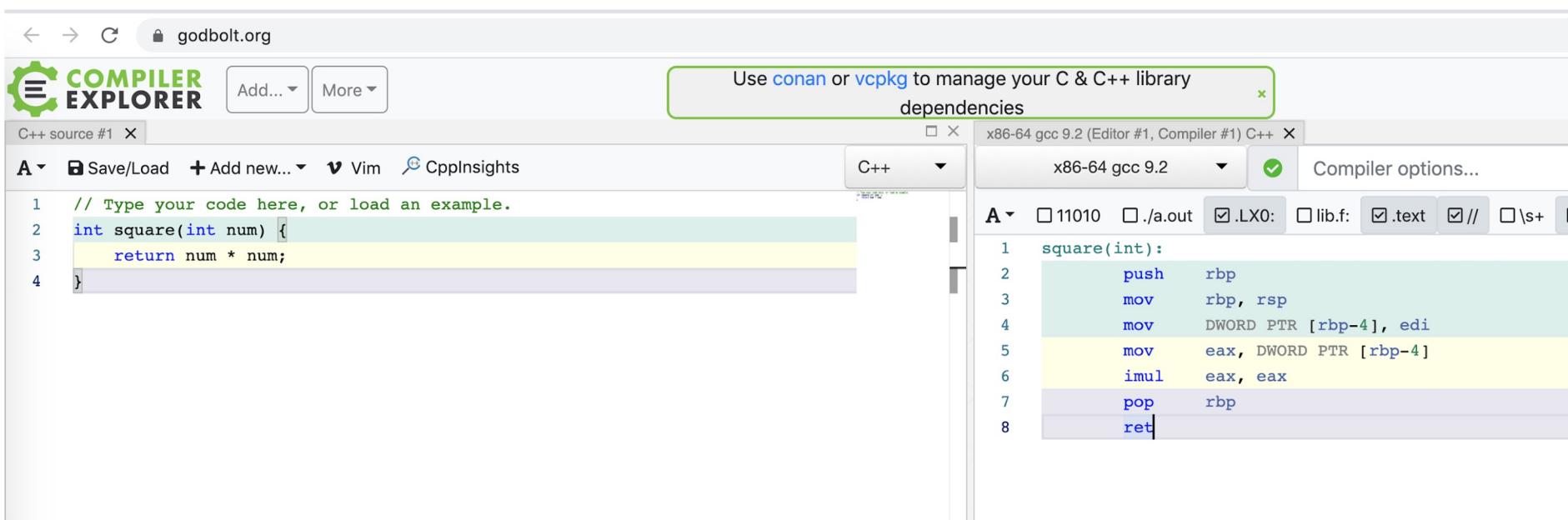
Exercise: Inspect ISA for sum

- Swap example
 - <https://passlab.github.io/ITSC3181/exercises/sum>
- Check
 - [sum_x86_64.s](#),
- Generate and execute
 - [gcc -s sum.c -o sum_x86_64.s](#)
 - [gcc -c sum.c](#)
 - [objdump -D sum.o > sum_x86_64_objdump.txt](#)
- Other system commands:
 - [cat /proc/cpuinfo](#) to show the CPU and #cores
 - [top](#) command to show system usage and memory

```
float sum(int N, float x[], float a) {  
    int i;  
    float result = 0.0;  
    for (i = 0; i < N; ++i)  
        result += a * x[i];  
    return result;  
}
```

Compiler Explorer

- Explore other ISA assembly from Compiler Explorer at
<https://godbolt.org/>



The screenshot shows the Compiler Explorer web interface. On the left, there is a code editor window titled "C++ source #1" containing the following C++ code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

On the right, there is a assembly output window titled "x86-64 gcc 9.2 (Editor #1, Compiler #1) C++". It displays the assembly code generated by the compiler:

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul   eax, eax
7     pop    rbp
8     ret
```

The assembly code is color-coded: green for labels and instructions, yellow for memory locations, and grey for comments and labels.

Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

Relevant course level outcomes:

- Explain the instruction set architecture of a computer system and how an instruction is represented at the machine level and symbolic assembler level.
- Use assembly instructions to write simple programs to understand the core concepts of ISA and assembly programming.
- Be able to map major high-level language statements into assembly/machine language notations.

At the end of this module, students should be able to:

1. explain what an instruction set architecture (ISA) is and recognize existing ISAs used in the computer industry (x86, ARM, RISC-V, etc) and their CPU vendors;
2. explain how ISA is used for creating and executing a program (e.g. program in high-level language is translated into assembly and assembly program is assembled into binary);
3. enumerate and explain the three major classes of instructions in most RISC ISA: arithmetic-logic instructions, memory access instructions, and control transfer instructions;
4. use arithmetic and logic instructions, memory access instructions, and control transfer instructions to convert high-level C language constructs to assembly instructions.

Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

- 👉 **Unit 1: Module overview, Instruction Set Architecture (ISA) and assembly programs, registers, instruction operations and operands, register and immediate operands, arithmetic and logic instructions**
- **Unit 2: Memory operands and memory access instructions**
- **Unit 3: Conditional control and instructions for making Decisions (if-else) and loops**
- **Unit 4: Supporting Functions and procedures**
- **Unit 5: Sort examples and comparison with other ISAs**

- **Materials are developed based on textbook:**
 - Computer Organization and Design RISC-V Edition: The Hardware/Software Interface, [Amazon](#)
 - RISC-V Specification: <https://riscv.org/technical/specifications/>
 - ITSC 3181: <https://passlab.github.io/ITSC3181/>

The RISC-V Instruction Set

- Used as the example throughout the book
 - We will use and study only three classes of instructions
 - Sufficient for most programs.
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
 - See RISC-V Reference Data Card
- Similar ISAs have a large share of embedded core market
 - Applications in consumer electronics, network/storage, cameras, printers, ...
- Other Instruction Set Architectures:
 - X86 and X86_32: Intel and AMD, main-stream desktop/laptop/server
 - ARM: smart phones/tablets
 - RISC-V: emerging and free ISA, closer to MIPS than other ISAs
 - Others: Power, SPARC, etc

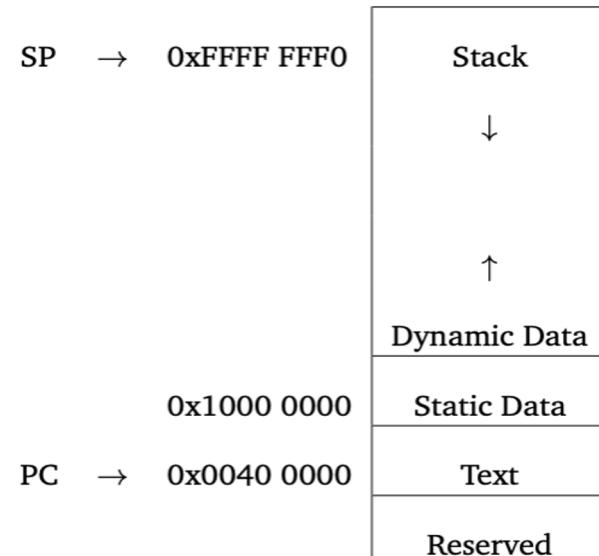
RISC-V Base Integer Instructions

Registers

Inst	Name	Description	Note
add	ADD	$R[rd] = R[rs1] + R[rs2]$	
sub	SUB	$R[rd] = R[rs1] - R[rs2]$	
xor	XOR	$R[rd] = R[rs1] \wedge R[rs2]$	
or	OR	$R[rd] = R[rs1] \vee R[rs2]$	
and	AND	$R[rd] = R[rs1] \& R[rs2]$	
sll	Shift Left Logical	$R[rd] = R[rs1] \ll R[rs2]$	
srl	Shift Right Logical	$R[rd] = R[rs1] \gg R[rs2]$	
sra	Shift Right Arith*	$R[rd] = R[rs1] \gg R[rs2]$	
slt	Set Less Than	$R[rd] = (rs1 < rs2)?1:0$	sign-extends
addi	ADD Immediate	$R[rd] = R[rs1] + SE(imm)$	
xori	XOR Immediate	$R[rd] = R[rs1] \wedge SE(imm)$	
ori	OR Immediate	$R[rd] = R[rs1] \vee SE(imm)$	
andi	AND Immediate	$R[rd] = R[rs1] \& SE(imm)$	
slli	Shift Left Logical Imm	$R[rd] = R[rs1] \ll imm[4:0]$	
srlti	Shift Right Logical Imm	$R[rd] = R[rs1] \gg imm[4:0]$	
srai	Shift Right Arith Imm	$R[rd] = R[rs1] \gg imm[4:0]$	
lw	Load Word	$R[rd] = M[R[rs1]+SE(immm)]$	
sw	Store Word	$M[R[rs1]+SE(immm)] = R[rs2]$	
beq	Branch ==	if(rs1 == rs2) PC += SE(immm) << 1	
bne	Branch !=	if(rs1 != rs2) PC += SE(immm) << 1	
blt	Branch <	if(rs1 < rs2) PC += SE(immm) <<1	
bge	Branch >=	if(rs1 >= rs2) PC += SE(immm) <<1	
jal	Jump And Link	$R[rd] = PC+4;$ $PC += SE(immm) <<1$	
jalr	Jump And Link Reg	$R[rd] = PC+4;$ $PC = R[rs1] + SE(immm)$	
lui	Load Upper Imm	$R[rd] = SE(immm) \ll 12$	
auipc	Add Upper Imm to PC	$R[rd] = PC + (SE(immm) \ll 12)$	
csrrw	CSR read & write	$R[rd] = CSRs[csr];$ $CSRs[csr] = R[rs1]$	
csrrs	CSR read & set	$R[rd] = CSRs[csr];$ $CSRs[csr] = CSRs[csr] R[rs1]$	
csrrc	CSR read & clear	$R[rd] = CSRs[csr];$ $CSRs[csr] =$ $CSRs[csr] \& \sim R[rs1]$	
ecall	Environment Call	Transfer control to OS	
ebreak	Environment Break	Transfer control to debugger	

Register	Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x30	t3-t5	Temporaries	Caller
x31	at	Assembler Temporary	Caller

Memory Allocation



Mapping High-Level Program to Instructions: Operands and Instructions

The screenshot shows the Compiler Explorer interface. On the left, the C source code for a function named `sum` is displayed:

```
1 int sum(int N, int X[], int a) {
2     int i;
3     int result = 0.0;
4     for (i = 0; i < N; ++i)
5         result += a * X[i];
6     return result;
7 }
```

The code is color-coded: `int`, `for`, and `return` are blue; variable names (`i`, `result`, `a`, `N`, `X`) are green; and the body of the loop and the multiplication expression are orange.

On the right, the generated RISC-V assembly code is shown:

```
1 sum:
2     addi    sp,sp,-48
3     sd     ra,40(sp)
4     sd     s0,32(sp)
5     addi   s0,sp,48
6     mv    a5,a0
7     sd     a1,-48(s0)
8     mv    a4,a2
9     sw     a5,-36(s0)
10    mv    a5,a4
11    sw     a5,-40(s0)
12    sw     zero,-24(s0)
13    sw     zero,-20(s0)
14    j     .L2
15 .L3:
16    lw     a5,-20(s0)
17    slli  a5,a5,2
18    ld     a4,-48(s0)
19    add   a5,a4,a5
20    lw     a5,0(a5)
21    lw     a4,-40(s0)
22    mulw a5,a4,a5
23    sext.w a5,a5
24    lw     a4,-24(s0)
25    addw a5,a4,a5
26    sw     a5,-24(s0)
```

The assembly code uses standard RISC-V mnemonics like `addi`, `sd`, `sw`, and `lw`. It also includes some floating-point operations (e.g., `slli` for multiplication by powers of 2).

- Variables: “*memory*” locations as we learned so far
 - Are mapped to **operands** of instructions
 - Three kinds of operands:
 - Register
 - Immediate (constant)
 - Memory location
- Operations: +/-/& etc, if-else, for loop
 - Are mapped to **Instructions** that use operands
 - Three kinds of instructions:
 - Compute: +, -, &, etc
 - Move data: between memory and registers
 - Control: if-else, for loop

We Will Study: Three Kinds of Operands and Three Classes of Instructions

- General form:
 - <op word> <dest operand> <src operand 1> <src operand 2>
 - E.g.: add x5, x3, x4, which performs [x5] = [x3] + [x4]

Three Kinds of Operands

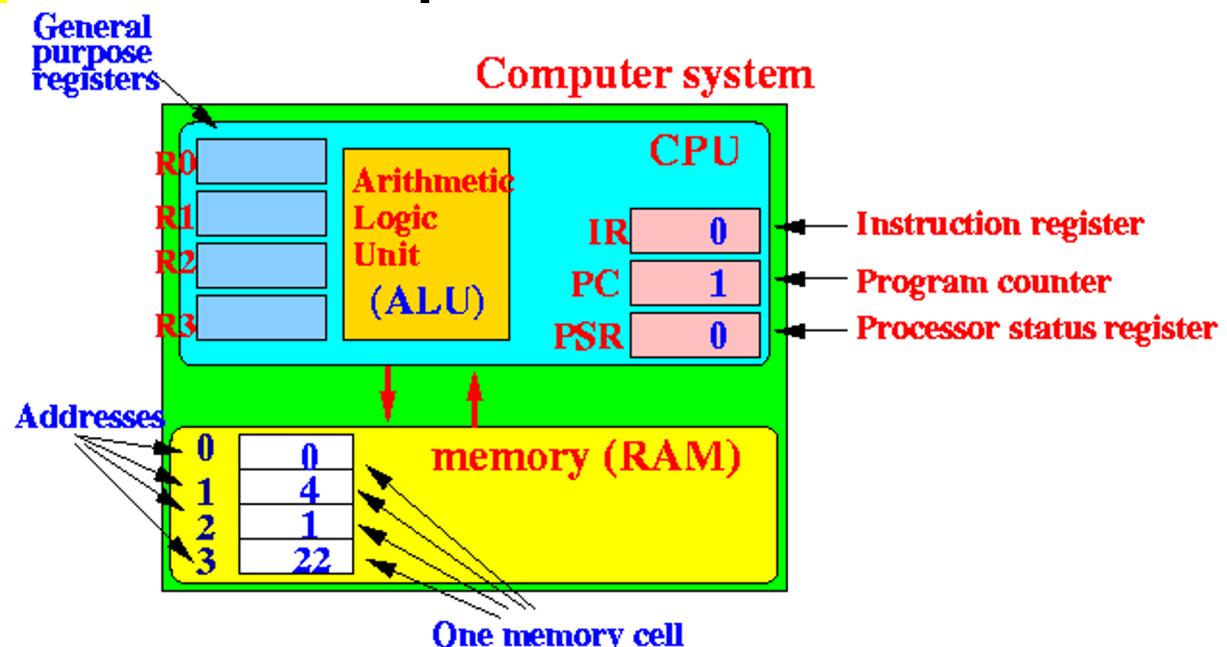
1. Register operands, e.g., x0 – x31
2. Immediate operands, e.g., 0, -10, etc
3. Memory operands, e.g. 16(x4)

Three Classes of Instructions

1. Arithmetic-logic instructions
 - add, sub, addi, and, or, shift left|right, etc
2. Memory load and store instructions
 - lw and sw: Load/store word
 - ld and sd: Load/store doubleword
3. Control transfer instructions (changing sequence of instruction execution)
 - Conditional branch: bne, beq
 - Unconditional jump: j (j offset OR jal x0 offset)
 - Procedure call and return: jal and jr

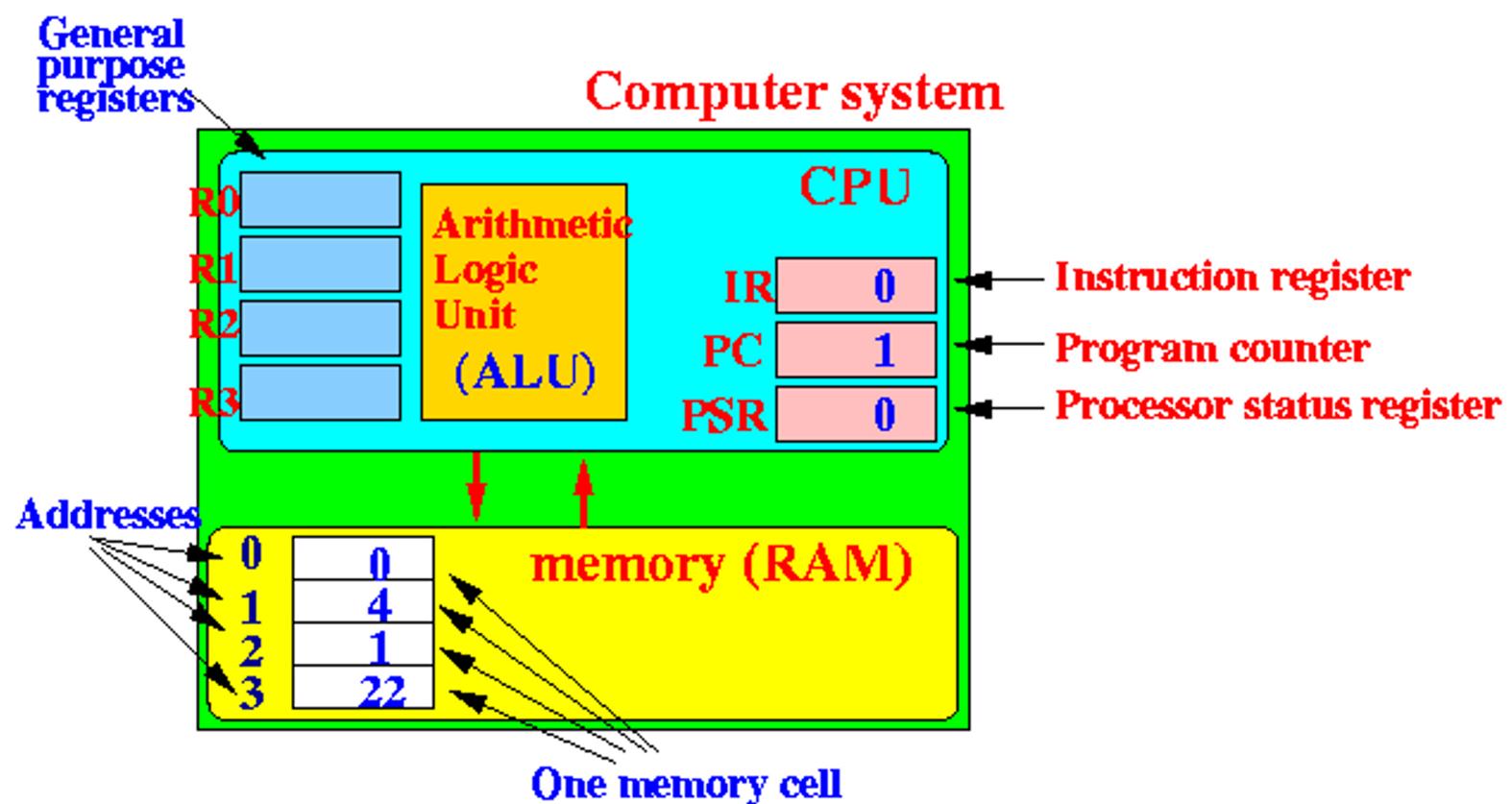
Registers in CPU and Register Operands of Instructions

- Registers are super-fast small **memory/storage** used in CPU.
 - General-purpose registers, program counter, instruction register, status register, floating-point register, etc
 - **32 GP Registers** in RISC-V CPU, 32-bit or 64-bit size for each
- Data and instructions need to be loaded to memory **and then register** in order to be processed.



Registers vs. Memory

- Registers are faster to access than memory
 - ~100x faster, ~10 more expensive, and takes more space
- **Operating on memory data requires loads and stores**
 - More instructions to be executed



Register Operands

- Arithmetic instructions use register operands
 - `add <dest>, <src1>, <src2>`
- 64-bit RISC-V has 32 64-bit general purpose *registers*
 - The storage for all GP registers is called a register file
 - It is storage, i.e., to store data
 - Use for frequently accessed data
 - Numbered `x0` to `x31`
 - the “memory address” for register
 - 64-bit data is called a “doubleword”
 - 32-bit data called a “word”
- *Design Principle:* Smaller is faster
 - c.f. main memory: millions of locations

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

RISC-V 32 64-Bit Registers, x0 to x31

- Usage **convention** for most programs:
 - **x0: the constant value 0**
 - **x1: return address of a function**
 - **x2: stack pointer of a functon**
 - **x3: global pointer**
 - **x4: thread pointer**
 - **x5 – x7, x28 – x31: temporaries**
 - **x8: frame pointer**
 - **x9, x18 – x27: saved registers**
 - **x10 – x11: function arguments/results**
 - **x12 – x17: function arguments**

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

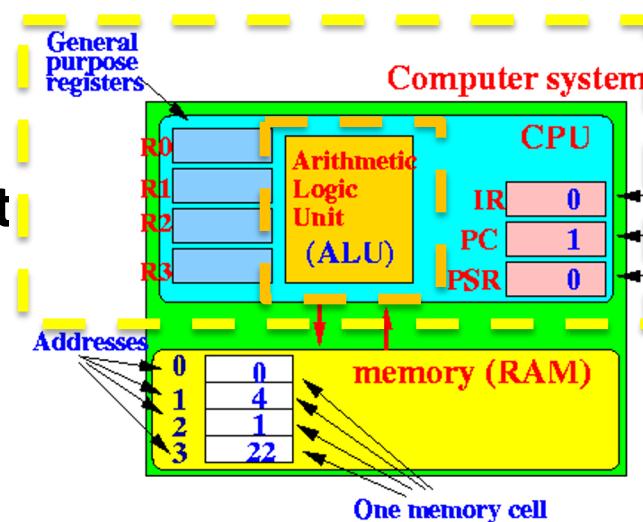
Arithmetic Operations (of the First Class Instrs)

- Add and subtract, three operands
 - Two source operands: provide input or source data
 - One destination operand: where result goes to.

`add a, b, c //sum of b and c is stored in a`

- All arithmetic operations have this form
 - Three operands, two sources and one destination
 - 3-operands instructions

- ***Design Principle: Simplicity favors regularity***
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost



Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

- f, ..., j values are already loaded in x19, x20, ..., x23

- Compiled RISC-V code, all are register operands

- **Three operands: the first operand is destination, last two are source operands**

```
add x5, x20, x21    // [x5] = [x20] + [x21]
```

```
add x6, x22, x23    // [x6] = [x22] + [x23]
```

```
sub x19, x5, x6     // [x19] = [x5] - [x6]
```

- [x5]: the actual data in register x5

Constant or Immediate Operands

`int a += 4;` if a is in register x22

- **Constant** data specified in an instruction

`addi x22, x22, 4`

Instructions that take immediate operands end with the letter **i**

- There is no subtract immediate instruction
 - Just use a negative constant
`addi x2, x1, -1`
- ***Design Principle: Make the common case fast***
 - **Small constants are common**
 - **Immediate operand avoids a load instruction**

The Constant Zero

- RISC-V register x0 is the constant 0 always
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers, e.g. a=b;
 - `add x9, x5, x0`
 - `addi x9, x5, 0`
 - E.g., initialization such as i = 0; j=-100;
 - `addi x9, x0, 0`
 - `addi x11, x0, -100`

Instructions for Logical Operations

- Instructions for **bitwise** manipulation.

Operation	C	Java	RISC-V
Shift left	<code><<</code>	<code><<</code>	<code>sll, slli</code>
Shift right	<code>>></code>	<code>>>></code>	<code>srl, srli</code>
Bit-by-bit AND	<code>&</code>	<code>&</code>	<code>and, andi</code>
Bit-by-bit OR	<code> </code>	<code> </code>	<code>or, ori</code>
Bit-by-bit XOR	<code>^</code>	<code>^</code>	<code>xor, xorri</code>
Bit-by-bit NOT	<code>~</code>	<code>~</code>	

- Useful for extracting and inserting groups of bits in a *word*.

In RISC-V architecture, a *word* refers to the size of a basic unit of data. Standard word sizes are typically 32, 64, and 128 bits.

Shift Logic Operation Examples

- Shift Left Logic: **slli** by i bits: multiplies by 2^i

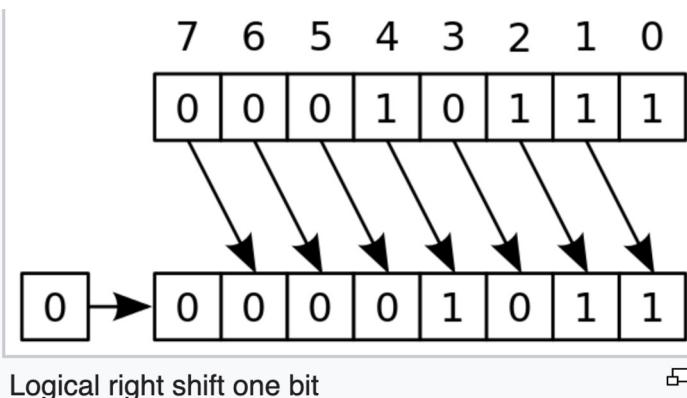
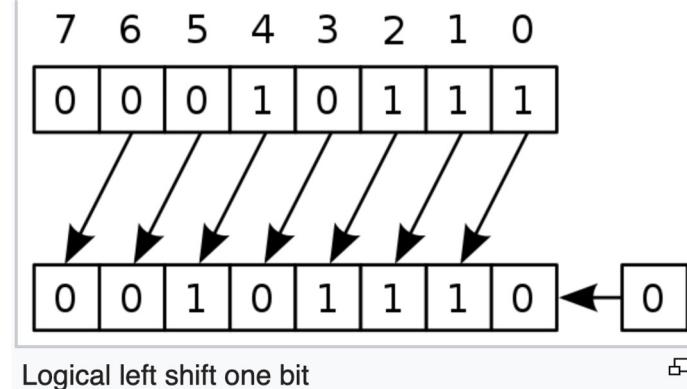
- C/java: `int i = 23; int j = i<<1; //46`
 - RISC-V: If i is in $x5$, and j is stored in $x6$:
 - **slliw x6, x5, 1**
 - **slliw: shift left logic immediate word**

- Instruction name

- Carries the operand type it operates
 - B: byte, H: half-word, W: word, D: double word

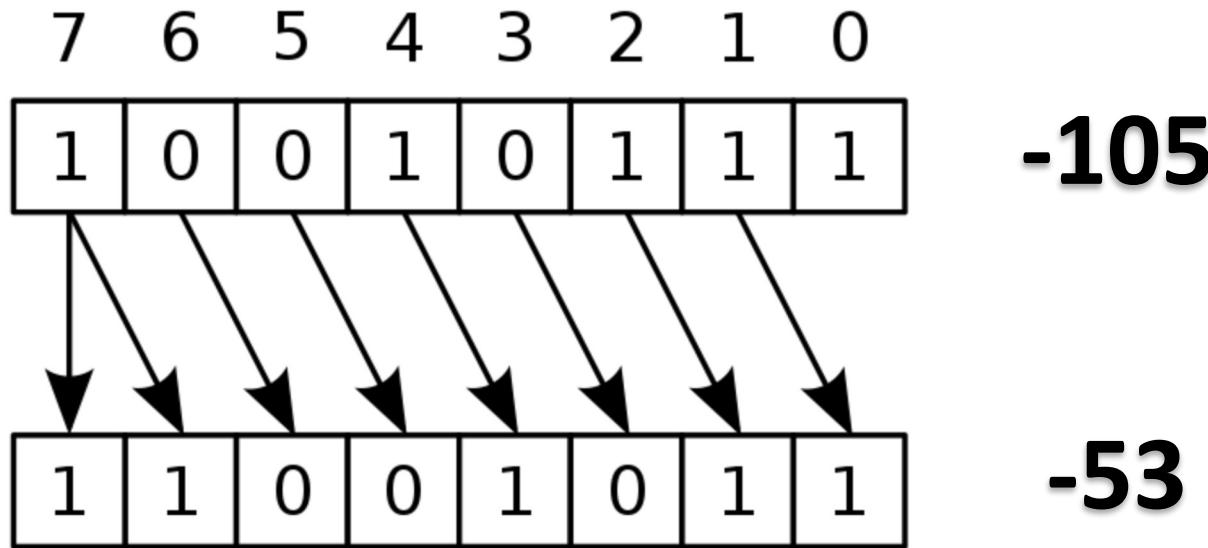
- Shift Right Logic: **srl**

- Java: `int i = 23; int j = i>>1; //j=11`
 - C: `int i = 23; int j = i >> 1; //j=11`
 - RISC-V: if i is in $x5$, j will be in $x6$:
 - **srlw x6, x5, 1**
 - Fill in 0, not much used for signed



Shift Right Arithmetic

- Shift right arithmetic (srai): Format: srai(w) rd, rs, #immediate
 - Shift right and fill with sign bit
 - srai by i bits: divides by 2^i
 - Java: `i=-105; int j=i>>1; // -53`
 - RISC-V: if i is in $x5$, j will be in $x6$:
 - `sraiw x6, x5, -1;`



Summary of Shift Operations

- immediate: how many positions to shift
- Shift left logical (sll): **Format: slli(w) rd, rs, #immediate**
 - Shift left and fill with 0 bits
 - slli by i bits: multiplies by 2^i
 - E.g., `int a = b << 2; //a = b * 4 (22)`
- Shift right logical (srl): **Format: srli(w) rd, rs, #immediate**
 - Shift right and fill **with 0 bits**
 - srli by i bits: divides by 2^i (unsigned only)
 - E.g., `int a = b >> 2; //a = b / 4 (22)`
- Shift right arithmetic (sra): **Format: srai(w) rd, rs, #immediate**
 - Shift right and fill with sign bit
 - srai by i bits: divides by 2^i

Instructions for AND, OR, XOR Operations

- Useful to mask bits in a word
 - Select only some bits, clear others to 0

and x9, x10, x11

or x9, x10, x11

xor x9, x10, x12

C Operator Precedence (recap)

(<i>type</i>)	casts	unary	14	right-to-left
* / %	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
	bitwise or	binary	6	left-to-right
&&	logical AND	binary	5	left-to-right
	logical OR	binary	4	left-to-right
? :	conditional	ternary	3	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right

A higher number in this table means higher precedence.

More Examples

Using ONLY the add, sub and slli instruction to convert the following C statement to the corresponding RISC-V assembly. Assume that the variables f, g, and j are integers assigned to registers t0, t1, and t2 respectively. You can use other temporary registers such as t3, t4, t5, t6, etc.

f = g - j + f * 16;

```
sub  t6, t1, t2      # t6 now has g - j
slli t5, t0, 4        # t5 now has f * 16
add  t0, t5, t6       # t0 (f) now has the result
```

More Examples

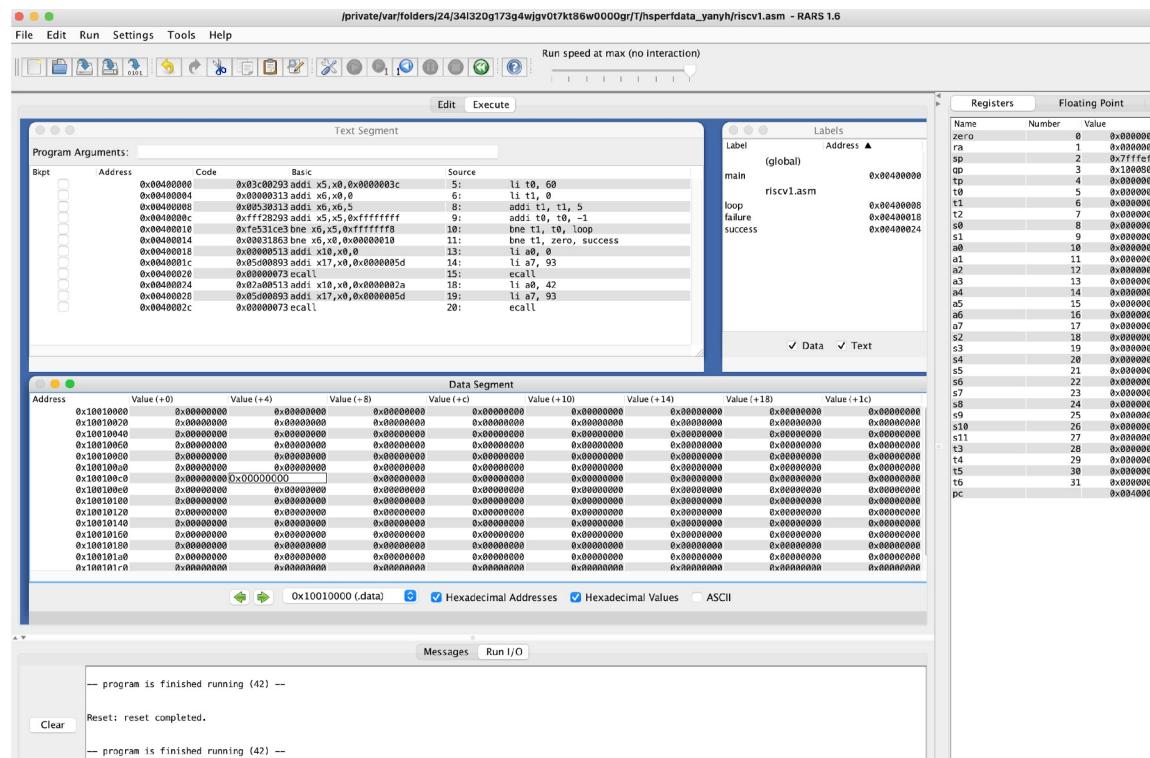
Using ONLY the add, sub and slli instruction to convert the following C statement to the corresponding RISC-V assembly. Assume that the variables `f`, `g`, and `j` are integers assigned to registers `t0`, `t1`, and `t2` respectively. You can use other temporary registers such as `t3`, `t4`, `t5`, `t6`, etc.

```
f = g - j * 65 + f * 8; //Hint: 65 = 64 + 1
```

slli t4, t2, 6	# t4 now has $j * 64$ ($2^6=64$)
add t4, t4, t2	# t4 now has $j * 65$
slli t5, t0, 3	# t5 now has $f * 8$. ($2^3=8$)
sub t6, t1, t4	# t6 now has $g - j * 65$
add t0, t5, t6	# t0 (f) now has the result.

RARS -- RISC-V Assembler and Runtime Simulator

- <https://github.com/TheThirdOne/rars>
 - We will use to write and execute simple RISC-V assembly programs
 - Download the jar file and execute it directly
https://github.com/TheThirdOne/rars/releases/download/v1.6/rars1_6.jar



Pseudo-instructions Used in RARS

- Are NOT machine instructions
- Are assembly instructions that help programmers
 - Translated to machine instructions by assembler
- For example
 - **mv x6, x7 //move/copy value from x7 to x6**
 - Machine instruction: add x6, x7, x0 //since x0 is always 0
 - Machine instruction: addi x6, x7, 0
 - **li x8, 100 //set the value of a register to be an immediate (load immediate)**
 - Machine instruction: addi x8, x0, 100
 - **la x10, label //load address of label to register**
 - Need two machine instructions
 - auipc x8, xxx
 - addi x0, x0, xxx

Module 06 - Unit 1 Exercise 2 on Canvas

RISC-V Assembly Programming with add/sub/slli