YOUR CODE IS YOUR RESPONSIBILITY

DEBUGGING IS PART OF YOUR DUTY

NO PSEUDOCODE/CODE ON PIAZZA
e.g., my while loop does not work etc

https://www.cs.sfu.ca/~ashriram/Courses/CS295//policy.html#assignment-ta-help-policy

https://www.cs.sfu.ca/~ashriram/Courses/CS295//assignments.html#obtaining-help-from-tainstructor

Abstraction (Levels of Representation/Interpretation)

Python Program

```
def square(num):
   return num * num
```

C Program

```
int square(int num):
   return num * num
```

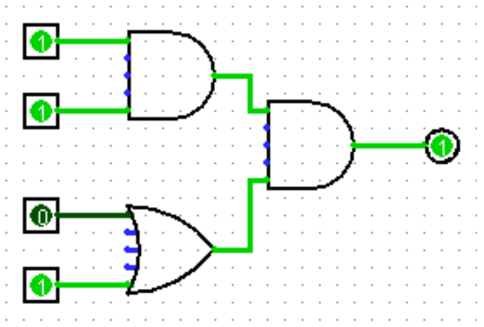
Binary

```
0x00000317
0x00830067
0xff010113
0x00112623
0x00812423
0x01010413
0xfea42a23
.......
```

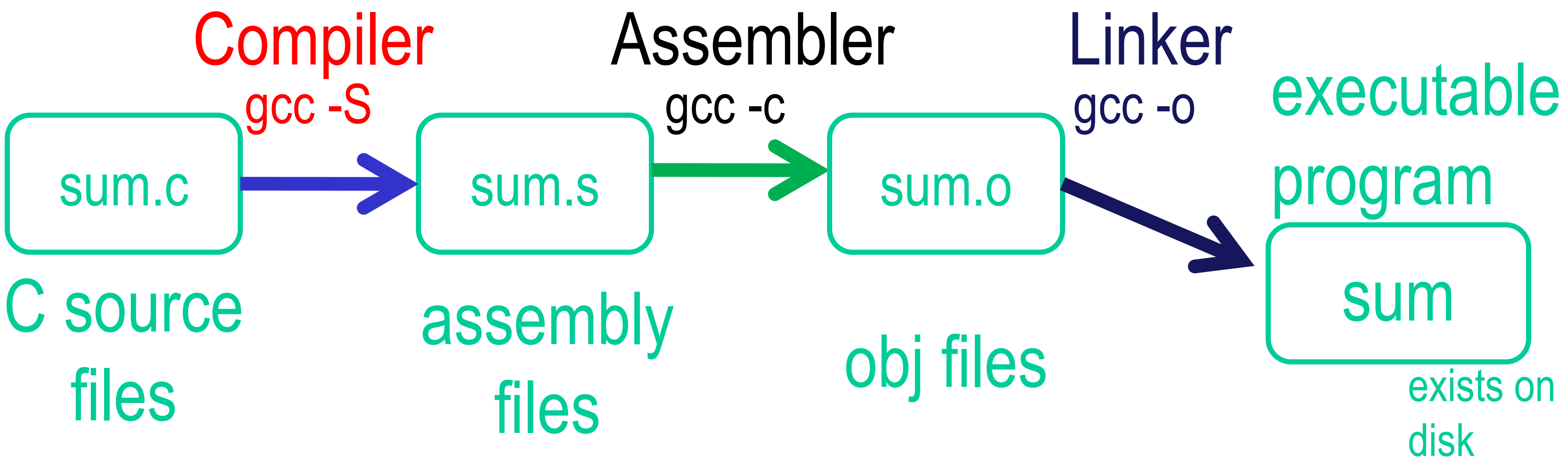
Assembly

```
square:
addi   sp,sp,-16
sw     ra,12(sp)
sw     s0,8(sp)
addi   s0,sp, 16
sw     a0,-12(s0)
lw     a0,-12(s0)
addi   a1,a0,2
mul    a0,a1,a0
lw     ra,12(sp)
lw     s0,8(sp)
addi   sp,sp,16
ret
```

Logic



From Writing to Running



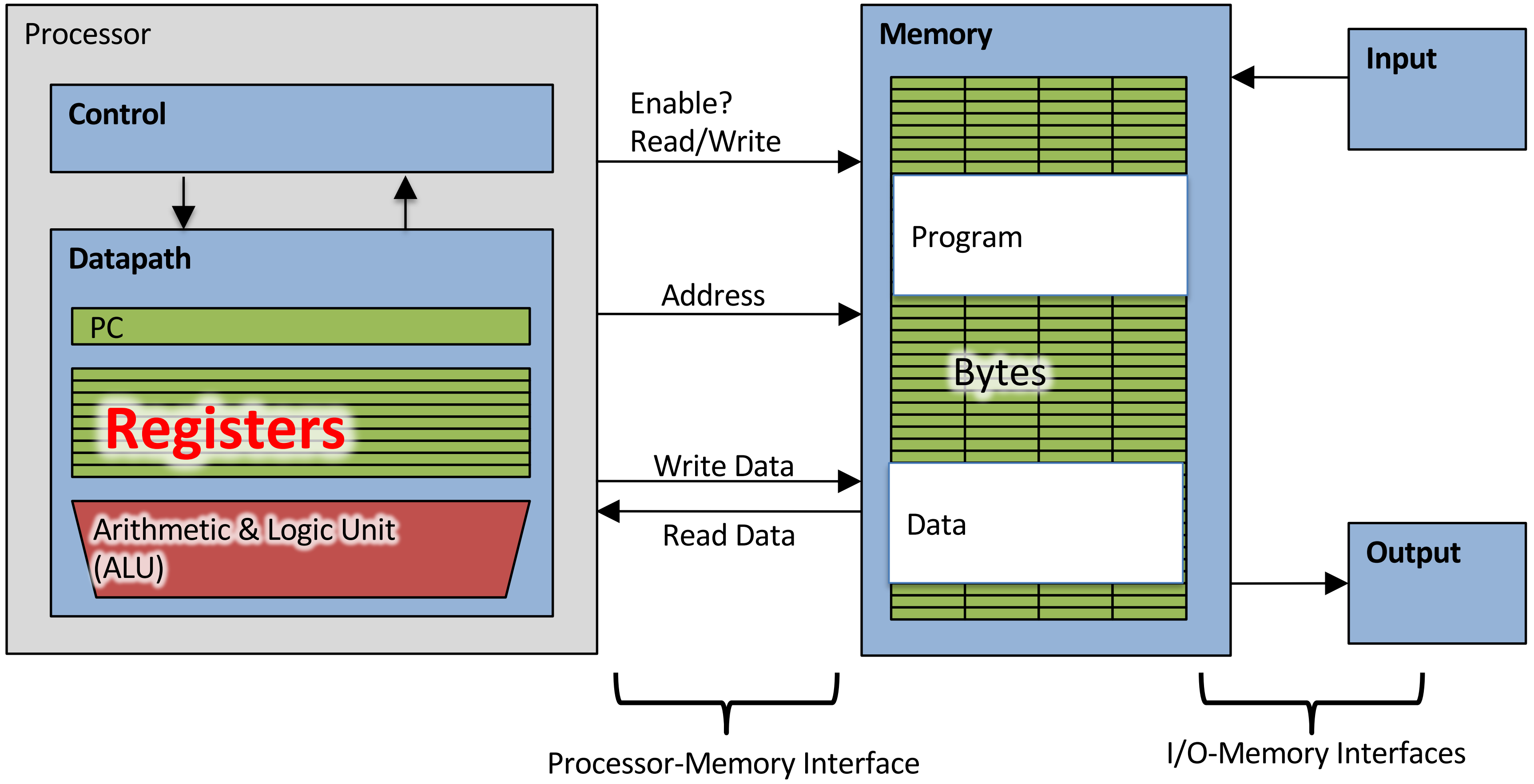
When most people say "compile" they mean the entire process:
compile + assemble + link

"It's alive!"

Executing in Memory

process

Aside: Registers are <u>Inside</u> the Processor



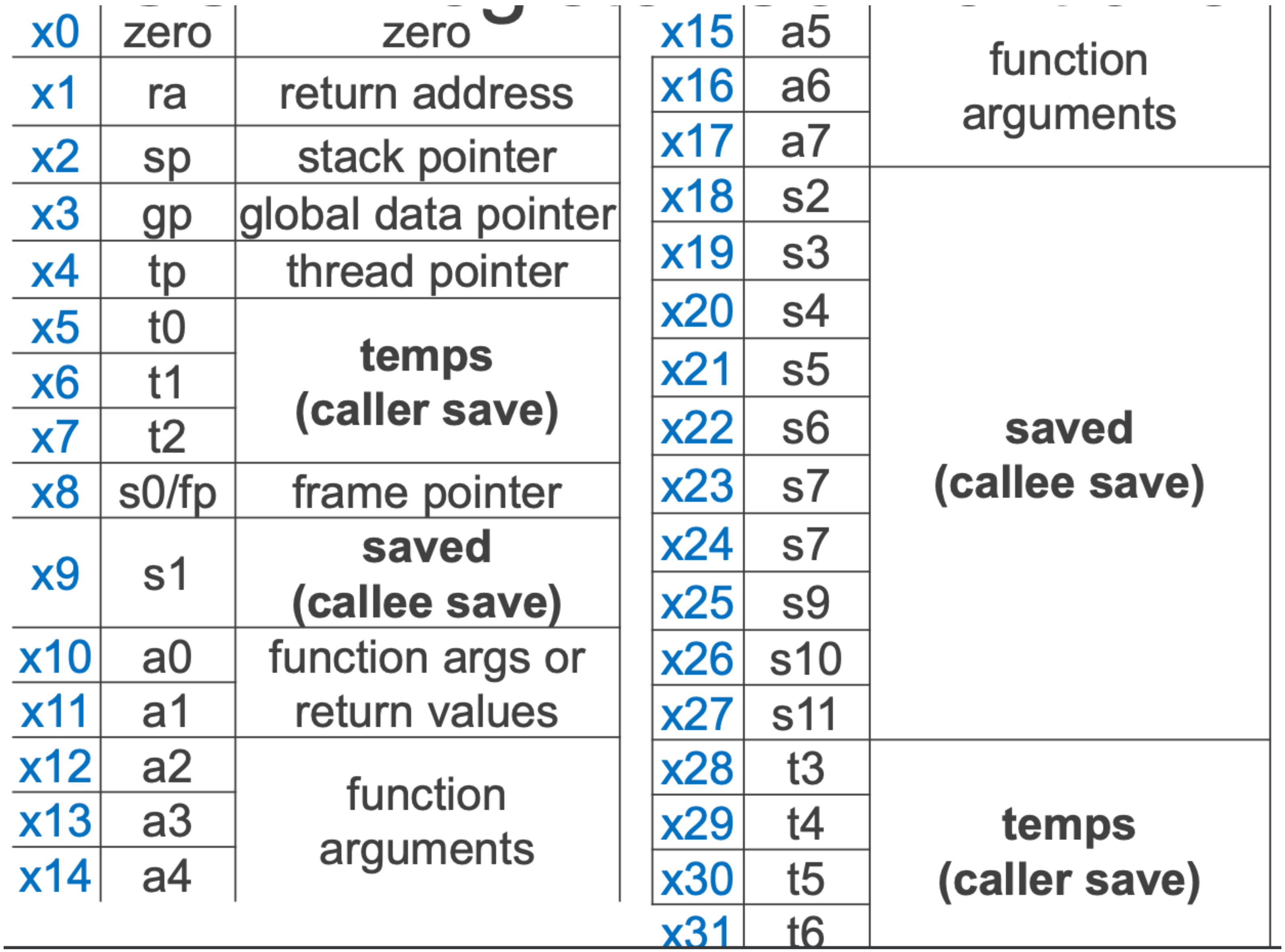
Registers -- Summary

- In high-level languages, number of variables limited only by available memory
- ISAs have a fixed, small number of operands called registers
  - Special locations built directly into hardware
  - **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  - **Drawback:** Operations can only be performed on these predetermined number of registers

Memory vs. Registers

Memory	vs.	Registers
Addresses	vs.	Names
`0x7FFFD024C3DC`		`%x0`
Big	vs.	Small
~ 8 GiB		(16 x 8 B) = 128 B
Slow	vs.	Fast
~50-100 ns		sub-nanosecond timescale
Dynamic	vs.	Static
Can "grow" as needed while program runs		fixed number in hardware

RISC V Integer Registers – 32 bits wide



x0	zero	zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global data pointer
x4	tp	thread pointer
x5	t0	**temps (caller save)**
x6	t1	
x7	t2	
x8	s0/fp	frame pointer
x9	s1	**saved (callee save)**
x10	a0	function args or return values
x11	a1	
x12	a2	function arguments
x13	a3	
x14	a4	
x15	a5	function arguments
x16	a6	
x17	a7	
x18	s2	**saved (callee save)**
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s7	
x25	s9	
x26	s10	
x27	s11	
x28	t3	**temps (caller save)**
x29	t4	
x30	t5	
x31	t6	

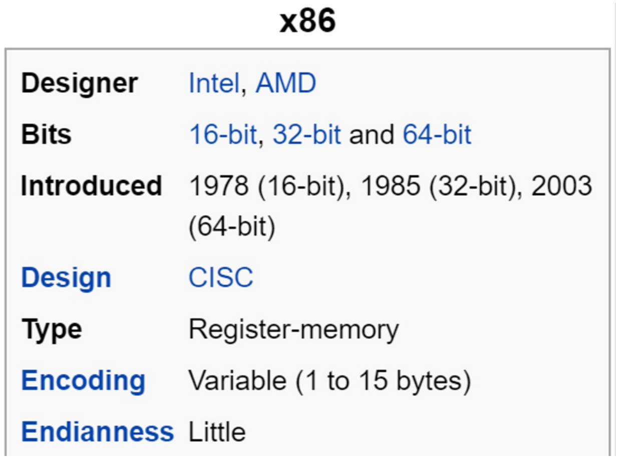
RISCV Instructions (1/2)

- Instruction Syntax is rigid:

```
op dst, src1, src2
```

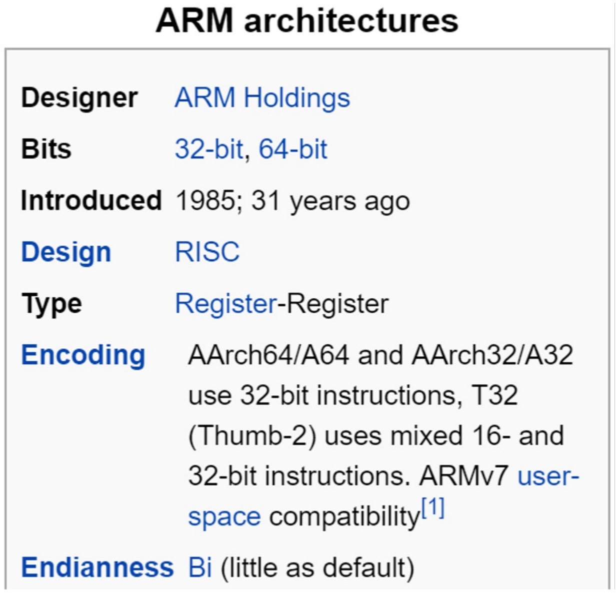
  - 1 operator, 3 operands
    - `op` = operation name ("operator")
    - `dst` = register getting result ("destination")
    - `src1` = first register for operation ("source 1")
    - `src2` = second register for operation ("source 2")
- Keep hardware simple via regularity

Mainstream ISAs



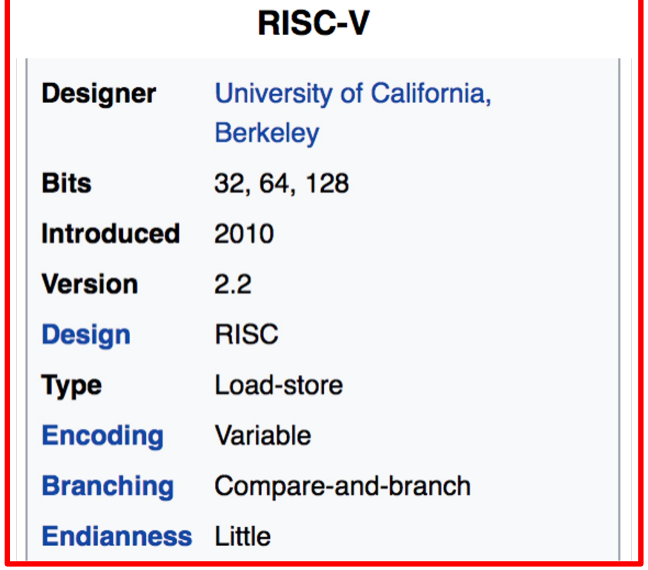
x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little



ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)



RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

PCs
(Core i3, i5, i7, M)
x86 Instruction Set

Smartphones, Laptop, Tablet
(iPhone, Macbook, iPad)
ARM Instruction Set

Versatile and open-source
Relatively new, designed for cloud computing, high-end phones, small embedded sys.
RISCV Instruction Set

Intel x86 ISA	RISC-V ISA
Complex instructions (~800 instructions) *ptr++; (e.g., string operations, REP MOVS, etc.)- inc DWORD PTR [EAX]	Small set (~48 instructions) lw t0, 0(a0) # Read memory addi t0, t0, 1 # Increment sw t0, 0(a0) #
Multiple addressing modes (e.g., base + displacement + scaled index)- Can combine memory operands with arithmetic (e.g., ADD	**Fewer addressing modes**. Separates pointer access from operations on the value.
Variable length (1 to 15 bytes)- Complex encodings	**Mostly fixed length** (32 bits in the base ISA, with optional 16-bit compressed)

RISCV Instructions (1/2)

- Instruction Syntax is rigid:

```
op dst, src1, src2
```

  - 1 operator, 3 operands
    - `op` = operation name ("operator")
    - `dst` = register getting result ("destination")
    - `src1` = first register for operation ("source 1")
    - `src2` = second register for operation ("source 2")
- Keep hardware simple via regularity

RISCV Instructions Example

- Your very first instructions!
  (assume here that the variables `a`, `b`, and `c` are assigned to registers `s1`, `s2`, and `s3`, respectively)
- Integer Addition (`add`)
  - C: `a = b + c`
  - RISCV: `add s1, s2, s3`
- Integer Subtraction (`sub`)
  - C: `a = b - c`
  - RISCV: `sub s1, s2, s3`

C example 1

- Suppose `a` → `s0`, `b` → `s1`, `c` → `s2`, `d` → `s3` and `e` → `s4`. Convert the following C statement to RISCV:

```
a = (b + c) - (d + e);
```

```
add t1, s3, s4
add t2, s1, s2
sub s0, t2, t1
```

Ordering of instructions matters (must follow order of operations)

Utilize temporary registers

Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - *Load* data from memory into register
     - `%reg` = Mem[address]
   - *Store* register data into memory
     - Mem[address] = `%reg`

Remember: Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - `c = a + b;     z = x << y;     i = h & g;`

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions

```
add x10, x1, x2  # a_temp = b + c
add x10, x10, x3 # a_temp = a_temp + d
sub x10, x10, x4 # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments). Check Apollo-11 comments!

C example 2

- Suppose `a` → `s0`, `e` → `s4`. Convert the following C statement to RISCV:

```
e = (a + 10);
```

```
addi s4, s0, 10
```

Since second operand is a literal value, no need to use separate register.

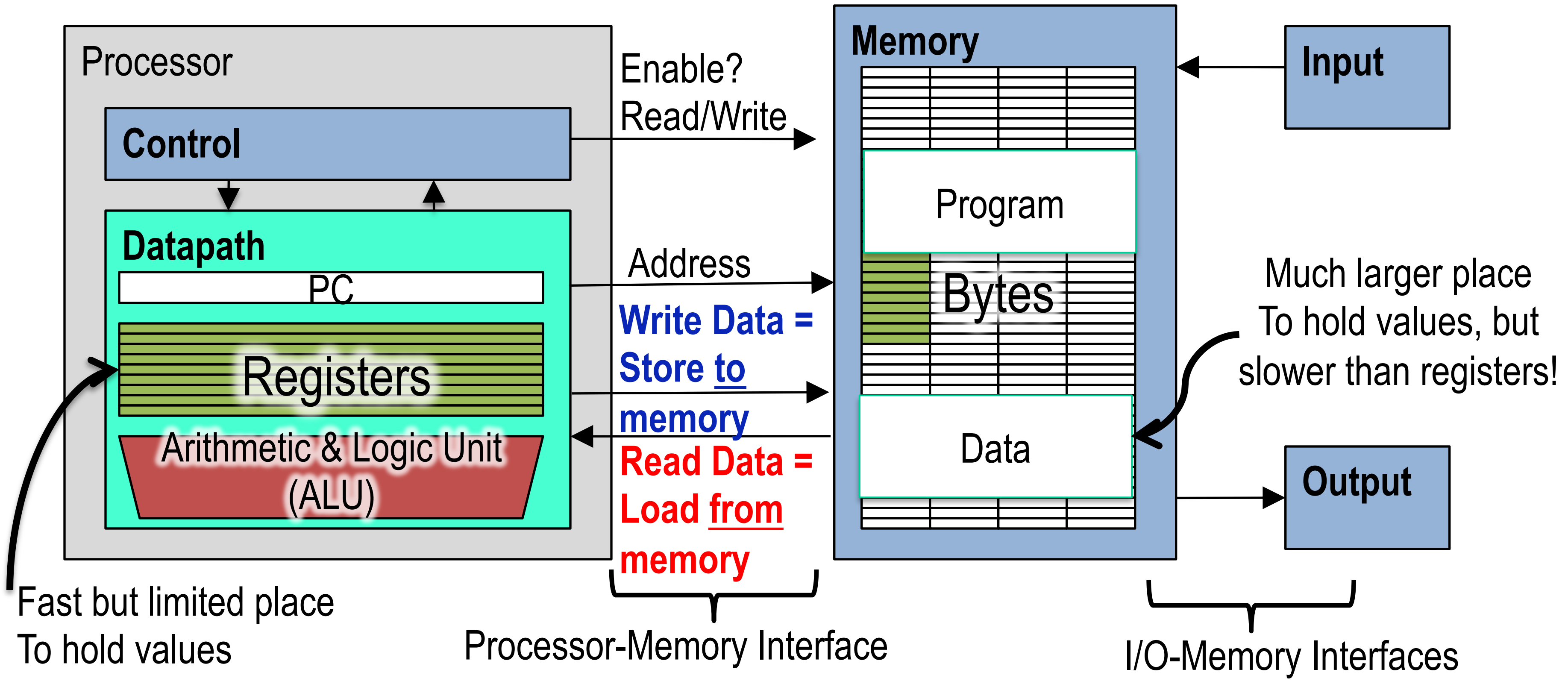
Immediates

- Numerical constants are called immediates
- Separate instruction syntax for immediates:

  `opi dst, src, imm`

  - Operation names end with ‘`i`’, replace 2nd source register with an immediate
- Example Uses:
  - `addi s1, s2, 5   # a=b+5`
  - `addi s3, s3, 1   # c++`
- Why no `subi` instruction?

Data Transfer: Load from and Store to memory


Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data = Store to memory
Read Data = Load from memory
Memory
Program
Bytes
Data
Input
Output
Much larger place To hold values, but slower than registers!
Fast but limited place To hold values
Processor-Memory Interface
I/O-Memory Interfaces

Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop)
- and physics dictates…
  - Smaller is faster
- How much faster are registers than DRAM??
  - About 100-500 times faster! (in terms of *latency* of one access)

C Example 3: Load Memory to Register

- C code

```
int  A[100];
g = h + A[3];
```

Data flow

- Using Load Word (`lw`) in RISC-V:

```
lw   x10,12(x15)  # Reg x10 gets A[3]
add x11,x12,x10  # g = h + A[3]
```

Note: `x15` – base register (pointer to A[0])
`12` – offset in bytes

Offset must be a constant known at assembly time

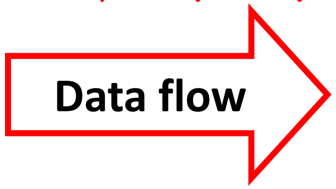
C Example 4: Store Register to Memory

- C code

```
     int  A[100];
     A[10] = h + A[3];
```

- Using Store Word (`sw`) in RISC-V:

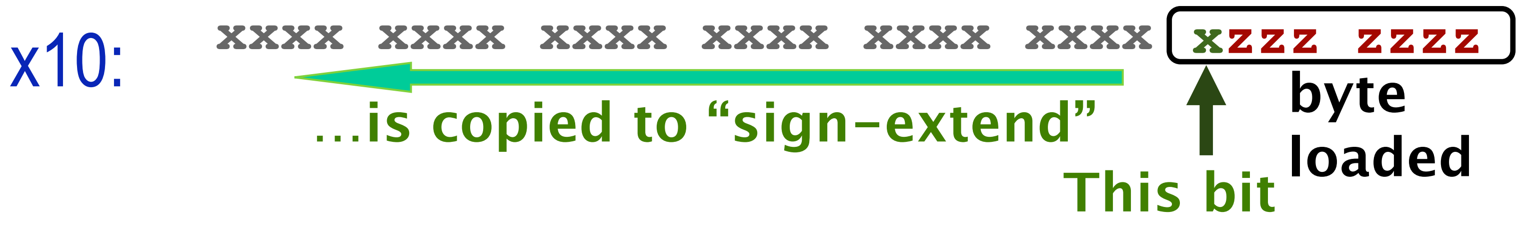
```
lw   x10,12(x15)    # Temp reg x10 gets A[3]
add  x10,x12,x10    # Temp reg x10 gets h + A[3]
sw    x10,40(x15)    # A[10] = h + A[3]
```



Note: `x15` – base register (pointer)
`12,40` – offsets in bytes
x15+12 and x15+40 must be multiples of 4

Loading and Storing Bytes

- In addition to word data transfers (**`lw`**, **`sw`**), RISC-V has byte data transfers:
  - load byte: **`lb`**
  - store byte: **`sb`**
- Same format as **`lw`**, **`sw`**
- E.g., **`lb x10,3(x11)`**
  - contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.



x10:
xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz
…is copied to "sign-extend"
This bit
byte loaded

RISC-V also has "unsigned byte" loads (**`lbu`**) which zero extends to fill register. Why no unsigned store byte **`sbu`**?

RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions

Decision Making Instructions

- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
- Jump (`j`)
  - `j label`
  - Unconditional jump to `label`

C Example 5: If Else

C Code:

```
if(i==j) {
   a = b   /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the <u>THEN</u> block
- If FALSE, execute the <u>ELSE</u> block

RISCV (beq):

```
# i→s0, j→s1
# a→s2, b→s3

beq s0,s1,???
???
sub s2, x0, s3
j   end
then:
add s2, s3, x0
end:
```

This label unnecessary

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b  /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the <u>THEN</u> block
- If FALSE, execute the <u>ELSE</u> block

RISCV (bne):

```
# i→s0, j→s1
# a→s2, b→s3
bne s0,s1,???
???
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

Branching on Conditions other than (Not) Equal

- Set Less Than (slt)
  - `slt dst, reg1,reg2`
  - If value in `reg1` < value in `reg2`, `dst` = 1, else 0
- Set Less Than Immediate (slti)
  - `slti dst, reg1,imm`
  - If value in `reg1` < `imm`, `dst` = 1, else 0

Breaking Down the If Else

C Code:

```
if(i<j) {
   a = b   /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (???):

```
# i→s0, j→s1
# a→s2, b→s3

slt t0 s0 s1
??? t0,??? else
then:
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum +=  A[i];
```

```
   add x9, x8, x0   # x9=&A[0]
   add x10, x0, x0  # sum=0
   add x11, x0, x0  # i=0
   addi x13,x0, 20  # x13=20
Loop:
   bge x11,x13,Done
   lw x12, 0(x9)    # x12=A[i]
   add x10,x10,x12  # sum+=
   addi x9, x9,4    # &A[i+1]
   addi x11,x11,1   # i++
   j Loop
Done:
```

C to **RISCV** Practice

- Let’s put our all of our new RISCV knowledge to use in an example: “Fast String Copy”
- C code is as follows:

```
/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != '\0') ;
```

- What do we know about its structure?
  - Single `while` loop
  - Exit condition is an equality test

C to RISCV Practice

- Start with code skeleton:

```
# copy String p to q
# p→s0, q→s1 (pointers)
Loop:                       # t0 = *p
                            # *q = t0
                            # p = p + 1
                            # q = q + 1
                            # if *p==0, go to Exit
      j Loop                # go to Loop
Exit:
```

C to **RISCV** Practice

- Fill in lines:

```
# copy String p to q
# p→s0, q→s1 (pointers)
Loop: lb   t0,0(s0)      # t0 = *p
      sb   t0,0(s1)      # *q = t0
      addi s0,s0,1    # p = p + 1
      addi s1,s1,1       # q = q + 1
      beq  t0,0,Exit     # if *p==0, go to Exit
      j Loop             # go to Loop
Exit:
```

C to **RISCV** Practice

- Finished code:

```
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop: lb   t0,0(s0)     # t0 = *p
      sb   t0,0(s1)     # *q = t0
      addi s0,s0,1      # p = p + 1
      addi s1,s1,1      # q = q + 1
      beq  t0,x0,Exit # if *p==0, go to Exit
      j Loop            # go to Loop
Exit: # N chars in p => N*6 instructions
```

C to **RISCV** Practice

- Alternate code using bne:

```
# copy String p to q
# p→s0, q→s1 (pointers)
Loop: lb   t0,0(s0)    # t0 = *p
      sb   t0,0(s1)    # *q = t0
      addi s0,s0,1     # p = p + 1
      addi s1,s1,1     # q = q + 1
      bne  t0,x0,Loop # if *p!=0, go to Loop
# N chars in p => N*5 instructions
```

Other Common Assembly Patterns

- A common loop in RISC-V includes initialization, a condition check, a loop body, and a branch back to the condition check.

```
   li      t0, 0         # Initialize counter t0 to 0
   li      t1, 10         # Upper limit 10
loop:
   bge     t0, t1, end    # i < 10.
   # Loop body (do something)

  addi    t0, t0, 1      #  increment loop index
   j       loop           # Jump back to the start of the loop
end:
   # Loop end
```

Other Common Assembly Patterns

- A common loop in RISC-V includes initialization, a condition check, a loop body, and a branch back to the condition check.

```
   la      t0, array     # Load base address of array into t0
   li      t1, 5         # Number of elements in the array
   li      t3, 0         # i = 0
loop:
   bge     t3, t1, end    # If i >= number of elements, exit loop
   slli    t4, t3, 2      # Calculate offset (index * 4, assuming 4-byte elements)
   add     t5, t0, t4     # Calculate address of array[index]
   lw      t6, 0(t5)      # Load array[index] into t6
   addi    t3, t3, 1      # i++
   j       loop
end:
```

Other Common Assembly Patterns

- For a 2D array with rows and cols, the address of array[i][j] is calculated as:
- Address = base_address + (i*#col+j)*sizeof(type)

```
t0= &array, t1 = # of cols  t2=i, t3=j  . X = a[i][j]

   mul     t5, t2, t1     # Row offset (row * cols)
   add     t5, t5, t3     # Add column index
   slli    t5, t5, 2      # Multiply by sizeof(type). Assume int here.
   add     t6, t0, t5     # Base_address + absolute offset.
   lw     t7, 0(t6)      # Load array[row][col] into t7
```

84056