CSE 420
Fundamentals of Computer Architecture

RISC-V ISA

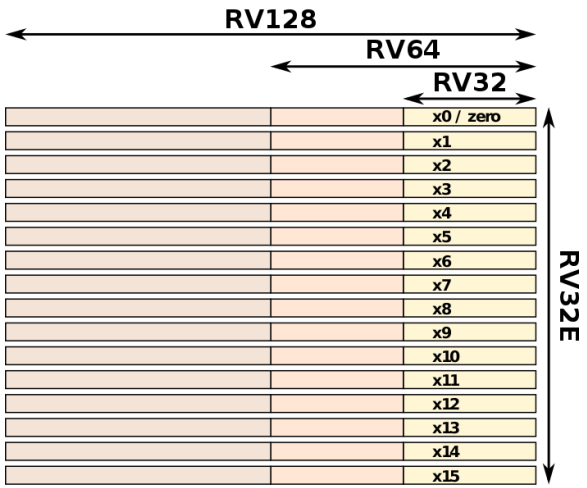
Prof. Michel A. Kinsy,
Instructors Mishel Paul & Nges Njungle

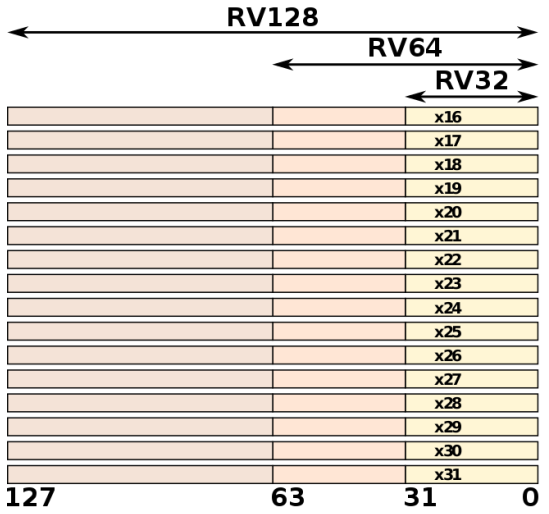
Brief Overview of the RISC-V ISA

- A new, open, free ISA from Berkeley
- Several variants
  - RV32, RV64, RV128 – Different data widths
  - 'I' – Base Integer instructions
  - 'M' – Multiply and Divide
  - 'A' – Atomic memory instructions
  - 'F' and 'D' – Single and Double precision floating point
  - 'V' – Vector extension
  - And many other modular extensions
- We will focus on the RV32I the base 32-bit variant

RV32I Register State

- 32 general purpose registers (GPR)
  - x0, x1, …, x31
  - 32-bit wide integer registers
  - x0 is hard-wired to zero





RV32I Register Conventions

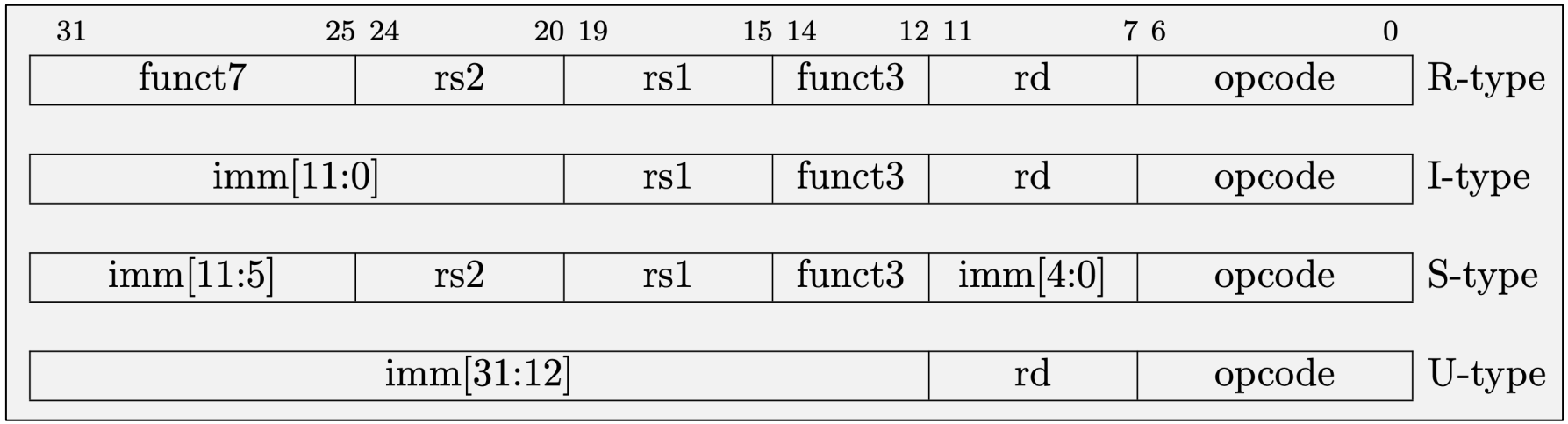
NAME	Register Number	Usage
zero	x0	Hardwired to the constant value 0
ra	x1	Return address for subroutine calls
sp	x2	Stack pointer (stack grows downwards)
gp	x3	Global pointer (e.g. to static data area)
tp	x4	Thread pointer
t0 – t2	x5 – x7	More temporary registers (caller saves)
s0/fp	x8	Frame pointer (to local variables on stack)
s1	x9	Saved register (callee saves)
a0 - a1	x10 – x11	Arguments (parameters) to subroutines / return values
a2 – a7	x12 – x17	Arguments (parameters) to subroutines
s2 - s11	x18 – x27	Saved registers (callee saves)
t3 – t6	x28 – x31	Temporary registers (caller saves)

RV32I Register State

- 32 general purpose registers (GPR)
  - x0, x1, …, x31
  - 32-bit wide integer registers
  - x0 is hard-wired to zero
- Program counter (PC)
  - 32-bit wide
- CSR (Control and Status Registers)
  - User-mode
    - cycle (clock cycles) // read only
    - instret (instruction counts) // read only
  - Machine-mode
    - hartid (hardware thread ID) // read only
    - mepc, mcause etc. used for exception handling
  - Custom
    - mtohost (output to host) // write only – custom extension

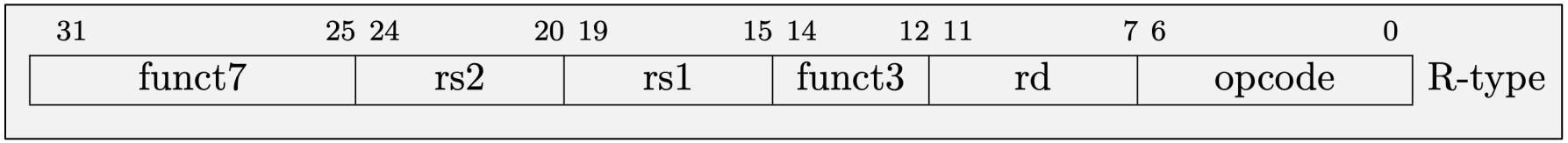
Base Instruction Formats

- The base RISC-V ISA has four main instruction formats
  - R, I, S and U types



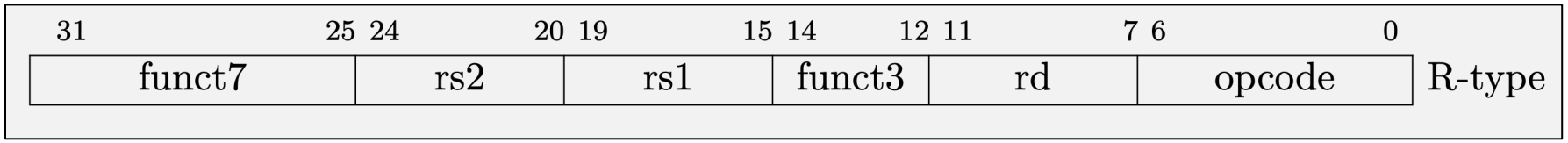
Base Instruction Formats

- opcode: Basic operation of the instruction
- rs1: The first register source operand
- rs2: The second register source operand



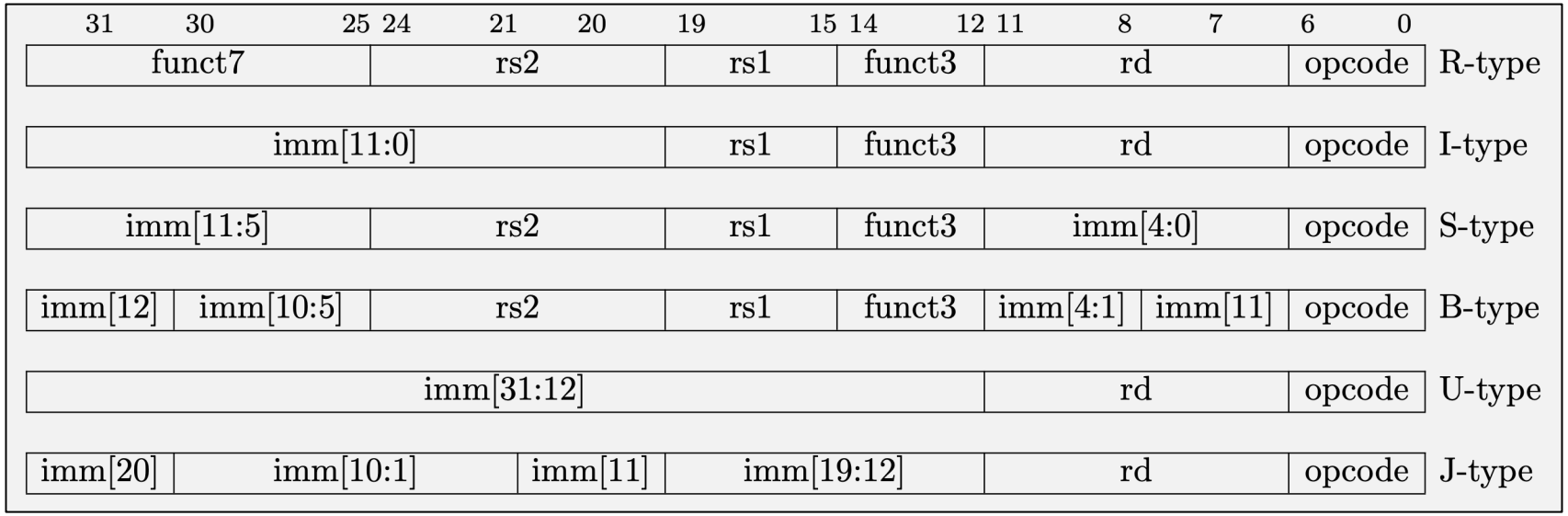
Base Instruction Formats

- opcode: Basic operation of the instruction
- rs1: The first register source operand
- rs2: The second register source operand
- rd: The register destination operand, it gets the result of the operation
- funct: Function. This field selects the specific variant of the operation in the op field, and is sometimes called the function code



Base Instruction Formats

- There are other formats dealing primarily with different versions of immediate

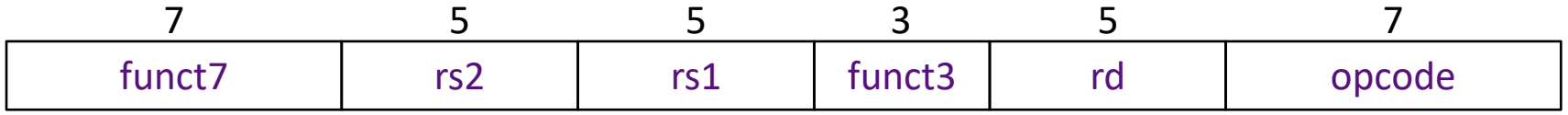


Base Instruction Formats

- There are other formats dealing primarily with different versions of immediate

Instruction Formats

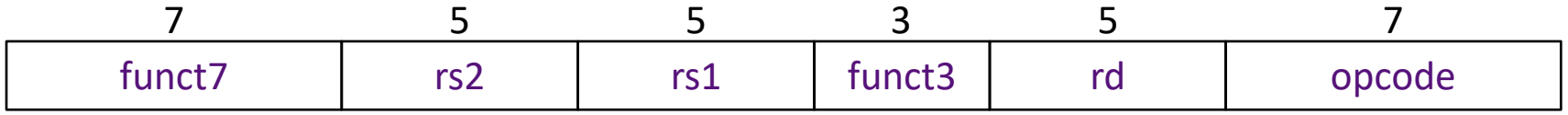
- R-type instruction



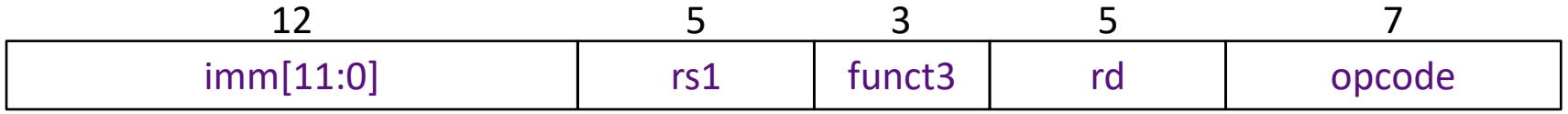

```
add x1, x2, x3                  # x1 = x2 + x3
```

Instruction Formats

- R-type instruction



- I-type instruction & I-immediate (32 bits)

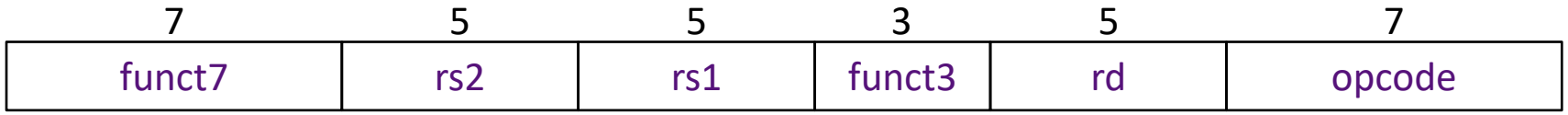


  - I-imm = signExtend(inst[31:20])

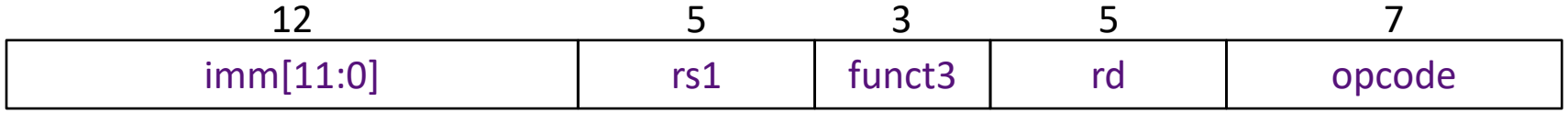
```
addx1, x2, 413                  # x1 = x2 + 413
```

Instruction Formats

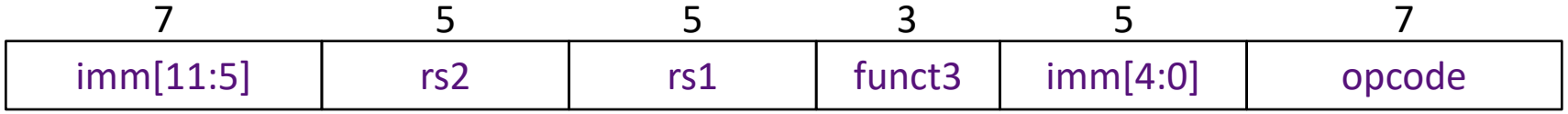
- R-type instruction



- I-type instruction & I-immediate (32 bits)



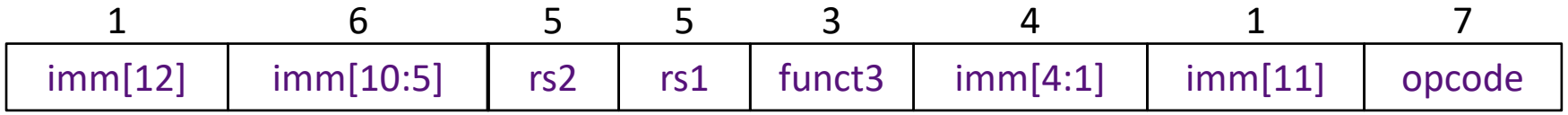
  - I-imm = signExtend(inst[31:20])
- S-type instruction & S-immediate (32 bits)



  - S-imm = signExtend({inst[31:25], inst[11:7]})

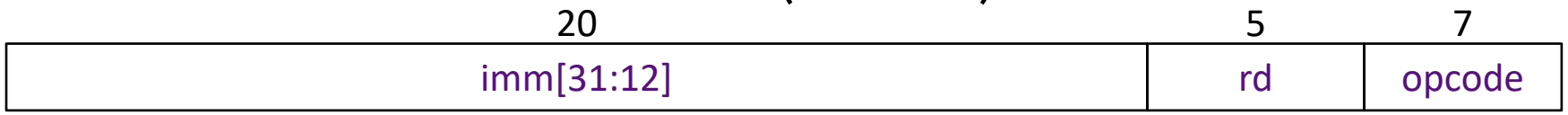
Instruction Formats

- SB-type instruction & B-immediate (32 bits)



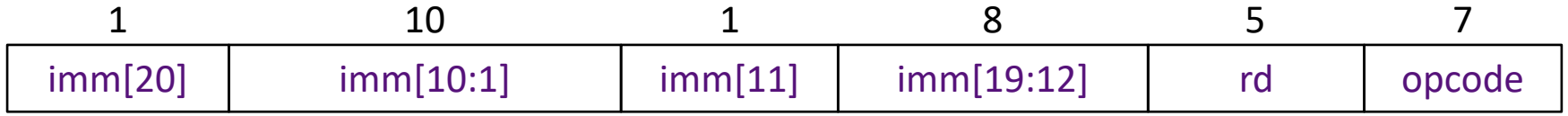
  - B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})

- U-type instruction & U-immediate (32 bits)



  - U-imm = signExtend({inst[31:12], 12'b0})

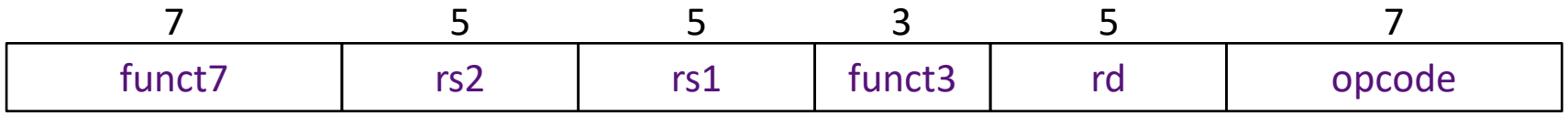
- UJ-type instruction & J-immediate (32 bits)



  - J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})

Computational Instructions

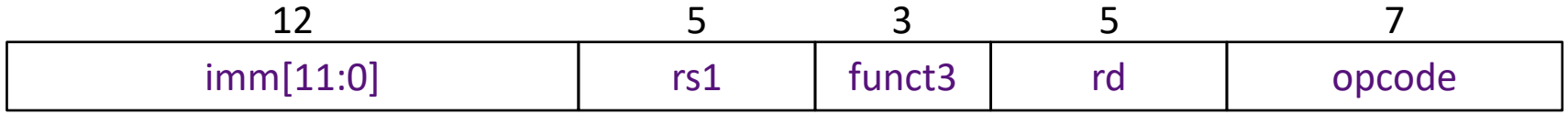
- Register-Register instructions (R-type)



  - opcode=OP: rd ← rs1 (funct3, funct7) rs2
  - funct3 = SLT/SLTU/AND/OR/XOR/SLL
  - funct3= ADD
    - funct7 = 0000000: rs1 + rs2
    - funct7 = 0100000: rs1 – rs2
  - funct3 = SRL
    - funct7 = 0000000: logical shift right
    - funct7 = 0100000: arithmetic shift right

Computational Instructions

- Register-immediate instructions (I-type)

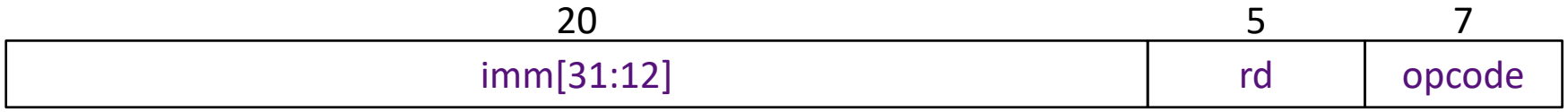


  - opcode = OP-IMM: rd ← rs1 (funct3) I-imm
  - I-imm = signExtend(inst[31:20])
  - funct3 = ADDI/SLTI/SLTIU/ANDI/ORI/XORI

- A slight variant in coding for shift instructions - SLLI / SRLI / SRAI
  - rd ← rs1 (funct3, inst[30]) I-imm[4:0]

Computational Instructions

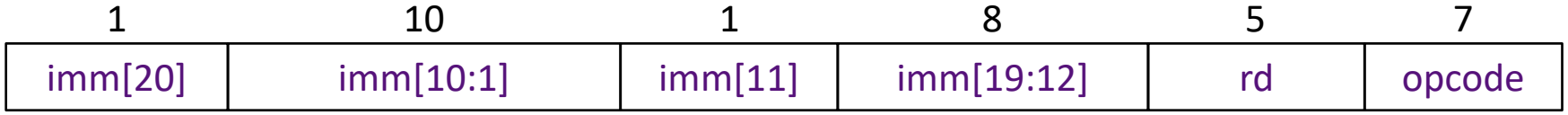
- Register-immediate instructions (U-type)



  - opcode = LUI : rd ← U-imm
  - opcode = AUIPC : rd ← pc + U-imm
  - U-imm = {inst[31:12], 12'b0}

Control Instructions

- Unconditional jump and link (UJ-type)

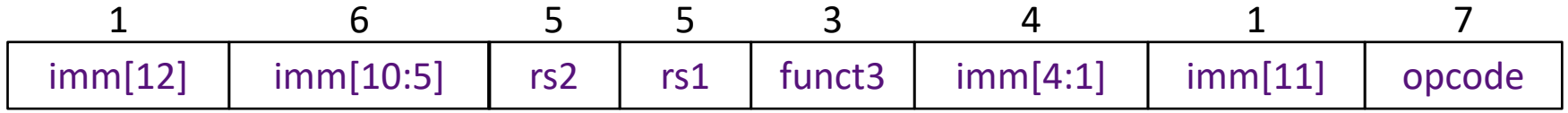


  - opcode = JAL: rd ← pc + 4; pc ← pc + J-imm
  - J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})
  - Jump ±1MB range
- Unconditional jump via register and link (I-type)

  - opcode = JALR: rd ← pc + 4; pc ← (rs1 + I-imm) & ~0x01
  - I-imm = signExtend(inst[31:20])

Control Instructions

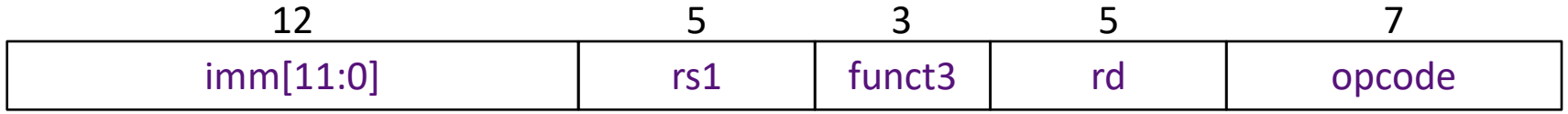
- Conditional branches (SB-type)



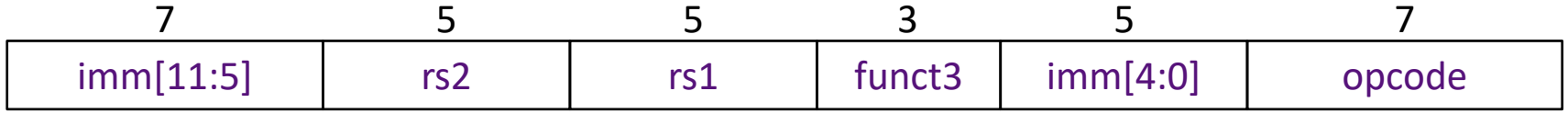
  - opcode = BRANCH: pc ← compare(funct3, rs1, rs2) ? pc + B-imm : pc + 4
  - B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})
  - Jump ±4KB range
  - funct3 = BEQ/BNE/BLT/BLTU/BGE/BGEU

Load & Store Instructions

- Load (I-type)

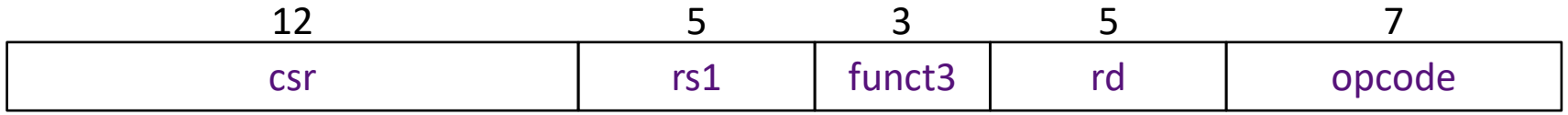


  - opcode = LOAD: rd ← mem[rs1 + I-imm]
  - I-imm = signExtend(inst[31:20])
  - funct3 = LW/LB/LBU/LH/LHU
- Store (S-type)



  - opcode = STORE: mem[rs1 + S-imm] ← rs2
  - S-imm = signExtend({inst[31:25], inst[11:7]})
  - funct3 = SW/SB/SH

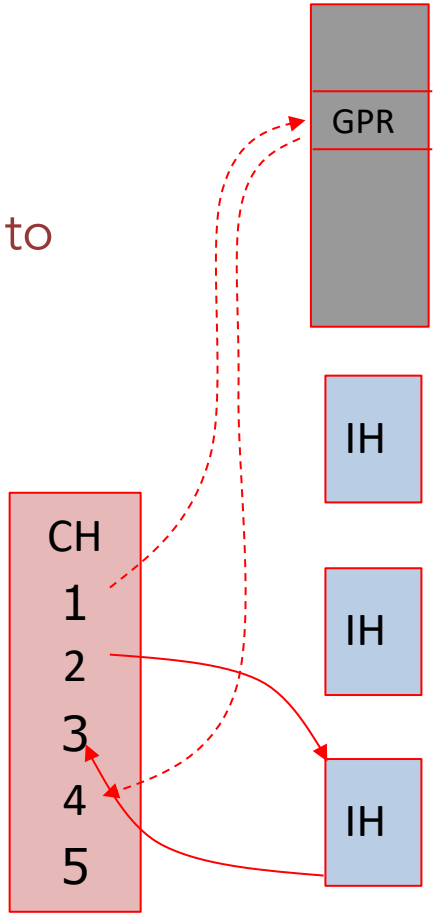
Instructions to Read and Write CSR



- opcode = SYSTEM
- CSRW rs1, csr (funct3 = CSRRW, rd = x0): csr ← rs1
- CSRR csr, rd (funct3 = CSRRS, rs1 = x0): rd ← csr

Software for interrupt handling

- Hardware transfers control to the common software interrupt handler (CH) which:
  1. Saves all GPRs into the memory pointed by `mscratch`
  2. Passes `mcause`, `mepc`, stack pointer to the IH (a C function) to handle the specific interrupt
  3. On the return from the IH, writes the return value to `mepc`
  4. Loads all GPRs from the memory
  5. Execute ERET, which does:
     - set pc to `mepc`
     - pop `mstatus` (mode, enable) stack



Instruction Types

- Register-to-Register Arithmetic and Logical operations
- Control Instructions alter the sequential control flow
- Memory Instructions move data to and from memory
- CSR Instructions move data between CSRs and GPRs; the instructions often perform read-modify-write operations on CSRs
- Privileged Instructions are needed by the operating systems, and most cannot be executed by user programs

Instruction Functions

- Data movement

Operation	Type	Template
Load Byte	*I*	*LB rd,rs1,imm*
Load Halfword	*I*	*LH rd,rs1,imm*
Load Word	*I*	*LW rd,rs1,imm*
Load Byte Unsigned	*I*	*LBU rd,rs1,imm*
Load Half Unsigned	*I*	*LHU rd,rs1,imm*

Instruction Functions

- Data movement

Operation	Type	Template
Load Byte	*I*	*LB rd,rs1,imm*
Load Halfword	*I*	*LH rd,rs1,imm*
Load Word	*I*	*LW rd,rs1,imm*
Load Byte Unsigned	*I*	*LBU rd,rs1,imm*
Load Half Unsigned	*I*	*LHU rd,rs1,imm*

Operation	Type	Template
Store Byte	*S*	*SB rs1,rs2,imm*
Store Halfword	*S*	*SH rs1,rs2,imm*
Store Word	*S*	*SW rs1,rs2,imm*

Instruction Functions

- Arithmetic & Logic

Operation	Type	Template
ADD	*R*	*ADD rd,rs1,rs2*
ADD Immediate	*I*	*ADDI rd,rs1,imm*
SUBtract	*R*	*SUB rd,rs1,rs2*
Load Upper Imm	*U*	*LUI rd,imm*
Add Upper Imm to PC	*U*	*AUIPC rd,imm*

Instruction Functions

- Arithmetic & Logic

Operation	Type	Template
ADD	R	ADD rd,rs1,rs2
ADD Immediate	I	ADDI rd,rs1,imm
SUBtract	R	SUB rd,rs1,rs2
Load Upper Imm	U	LUI rd,imm
Add Upper Imm to PC	U	AUIPC rd,imm

Operation	Type	Template
Shift Left	R	SLL rd,rs1,rs2
Shift Left Immediate	I	SLLI rd,rs1,shamt
Shift Right	R	SRL rd,rs1,rs2
Shift Right Immediate	I	SRLI rd,rs1,shamt
Shift Right Arithmetic	R	SRA rd,rs1,rs2
Shift Right Arith Imm	I	SRAI rd,rs1,shamt

Instruction Functions

- Arithmetic & Logic

Operation	Type	Template
XOR	R	XOR rd,rs1,rs2
XOR Immediate	I	XORI rd,rs1,imm
OR O	R	OR rd,rs1,rs2
R Immediate	I	ORI rd,rs1,imm
AND	R	AND rd,rs1,rs2
AND Immediate	I	ANDI rd,rs1,imm

Instruction Functions

- Arithmetic & Logic

Operation	Type	Template
XOR	R	XOR rd,rs1,rs2
XOR Immediate	I	XORI rd,rs1,imm
OR O	R	OR rd,rs1,rs2
R Immediate	I	ORI rd,rs1,imm
AND	R	AND rd,rs1,rs2
AND Immediate	I	ANDI rd,rs1,imm

Operation	Type	Template
Set <	R	SLT rd,rs1,rs2
Set < Immediate	I	SLTI rd,rs1,imm
Set < Unsigned	R	SLTU rd,rs1,rs2
Set < Imm Unsigned	I	SLTIU rd,rs1,imm

Instruction Functions

- Control Flow

Operation	Type	Template
Branch =	SB	BEQ rs1,rs2,imm
Branch ≠	SB	BNE rs1,rs2,imm
Branch <	SB	BLT rs1,rs2,imm
Branch ≥	SB	BGE rs1,rs2,imm
Branch < Unsigned	SB	BLTU rs1,rs2,imm
Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm

Instruction Functions

- Control Flow

Operation	Type	Template
Branch =	*SB*	*BEQ rs1,rs2,imm*
Branch ≠	*SB*	*BNE rs1,rs2,imm*
Branch <	*SB*	*BLT rs1,rs2,imm*
Branch ≥	*SB*	*BGE rs1,rs2,imm*
Branch < Unsigned	*SB*	*BLTU rs1,rs2,imm*
Branch ≥ Unsigned	*SB*	*BGEU rs1,rs2,imm*

Operation	Type	Template
Jump & Link	*UJ*	*JAL rd,imm*
Jump & Link Register	*UJ*	*JALR rd,rs1,imm*

Instruction Functions

- System call

Operation	Type	Template
System CALL	*I*	*SCALL*
System BREAK	*I*	*SBREAK*

Assembly Programming

- Function call
  1. Caller places parameters in a place where the procedure can access them
  2. Transfer control to the procedure
  3. Acquire storage resources required by the procedure
  4. Execute the statements in the procedure
  5. Called function places the result in a place where the caller can access it
  6. Return control to the statement next to the procedure call

Assembly Programming

- Function call
  - Argument Passing
    - Arguments to a function passed through a0-a7
    - Functions with more than 8 arguments
      - First eight arguments are put in a0-a7
      - Remaining arguments are put on stack by the caller
  - Return Values
    - Return values from a function passed through a0-a1
    - Functions with more than 2 return values

Assembly Programming

- Function call
  - Argument Passing
    - Arguments to a function passed through a0-a7
    - Functions with more than 8 arguments
  - Return Values
    - Return values from a function passed through a0-a1
    - Functions with more than 2 return values
      - First two return values put in a0-a1
      - Remaining return values put on stack by the function
      - The remaining return values are popped from the stack by the caller

If then Else Assembly

```
if (a0 < 0) then
{
        t0 = 0 - a0;
        t1 = t1 +1;
}
else
{
        t0 = a0;
        t2 = t2 + 1;
}
```

```
    bgez  a0, else
    # if (a0 is > or = zero) branch to
else
    sub   t0, zero, a0
    # t0 gets the negative of a0
    addi  t1, t1, 1
    # increment t1 by 1
    j   next
    # branch around the else code
else:
    ori   t0, a0, 0
    # t0 gets a copy of a0
    addi  t2, t2, 1
    # increment t2 by 1
next:
```

While Do Assembly

```
t0 = 1
While (a1 < a2)
    do
    {
        t1 = mem[a1];
        t2 = mem[a2];
        if (t1 != t2)
            go to break;
        a1 = a1 +1;
        a2 = a2 -1;
    }
break: t0 = 0
```

```
li   t0, 1          # Load t0 with the value 1
loop:
    bgeu a1, a2, done
    # if( a1 >= a2) Branch to done
    lw   t1, 0(a1)
    # Load a Byte: t1 = mem[a1 + 0]
    lw   t2, 0(a2)
    # Load a Byte: t2 = mem[a2 + 0]
    bne  t1, t2, break
    # if (t1 != t2) Branch to break
    addi a1, a1, 1    # a1 = a1 + 1
    addi a2, a2, -1   # a2 = a2 - 1
    b    loop      # Branch to loop
break:
    li   t0, 0    # Load t0 with the value 0
done:
```

For loop Assembly

```
a0 = 0;
For ( t0 =10;
          t0 > 0;
          t0 = t0 -1)
      do {
         a0 = a0 + t0
      }
```

```
    li   a0, 0     #  a0 = 0
    li   t0, 10
     # Initialize loop counter to 10
loop:
    add   a0, a0, t0
    addi t0, t0, -1
   # Decrement loop counter
    bgtz   t0, loop
   # if (t0 >0) Branch to loop
```

Assembly

- Case study
  - Assume that A is an array of 64 words and the compiler has associated registers a1 and a2 with the variables x and y. Also assume that the starting address, or base address is contained in register a0. Determine the RISC-V instructions associated with the following C statement:
    - *x = y + A[4]; // adds 4th element in array A to y and stores result in x*

Assembly

- Case study
  - Assume that A is an array of 64 words and the compiler has associated registers a1 and a2 with the variables x and y. Also assume that the starting address, or base address is contained in register a0.
    - *x = y + A[4];* *// adds 4th element in array A to y and stores result in x*
  - Solution:
    - *lw t0, 16(a0)* *# a0 contains the base address of array and*
      # 16 is the offset address of the 4th element
    - *add a1, a2, t0* *# performs addition*

Next Lecture Module

- Assembly Simulation Environment

A Browser-based RISC-V Simulator

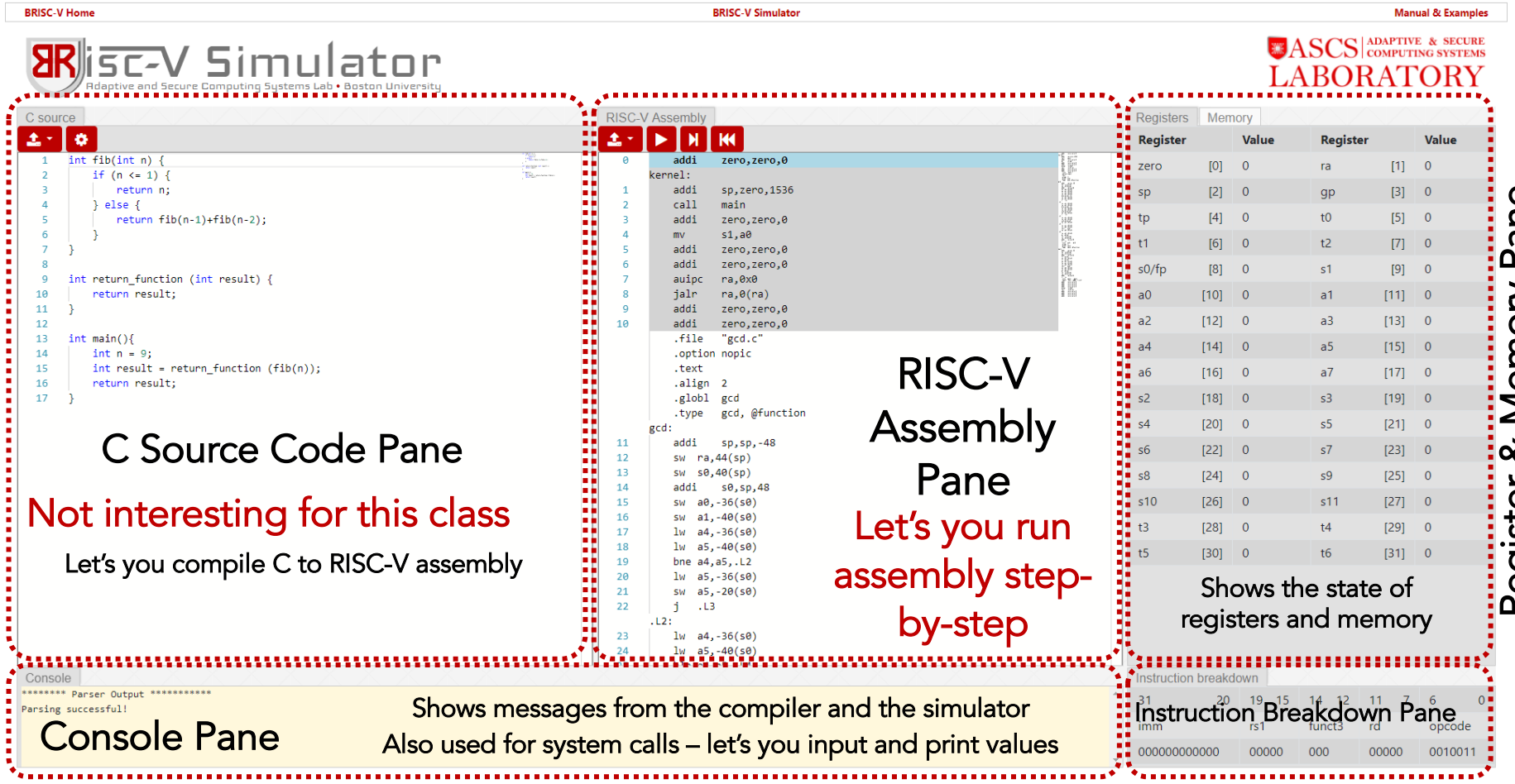
Adaptive and Secure Computing Systems (ASCS) Laboratory

BRISC-V Simulator

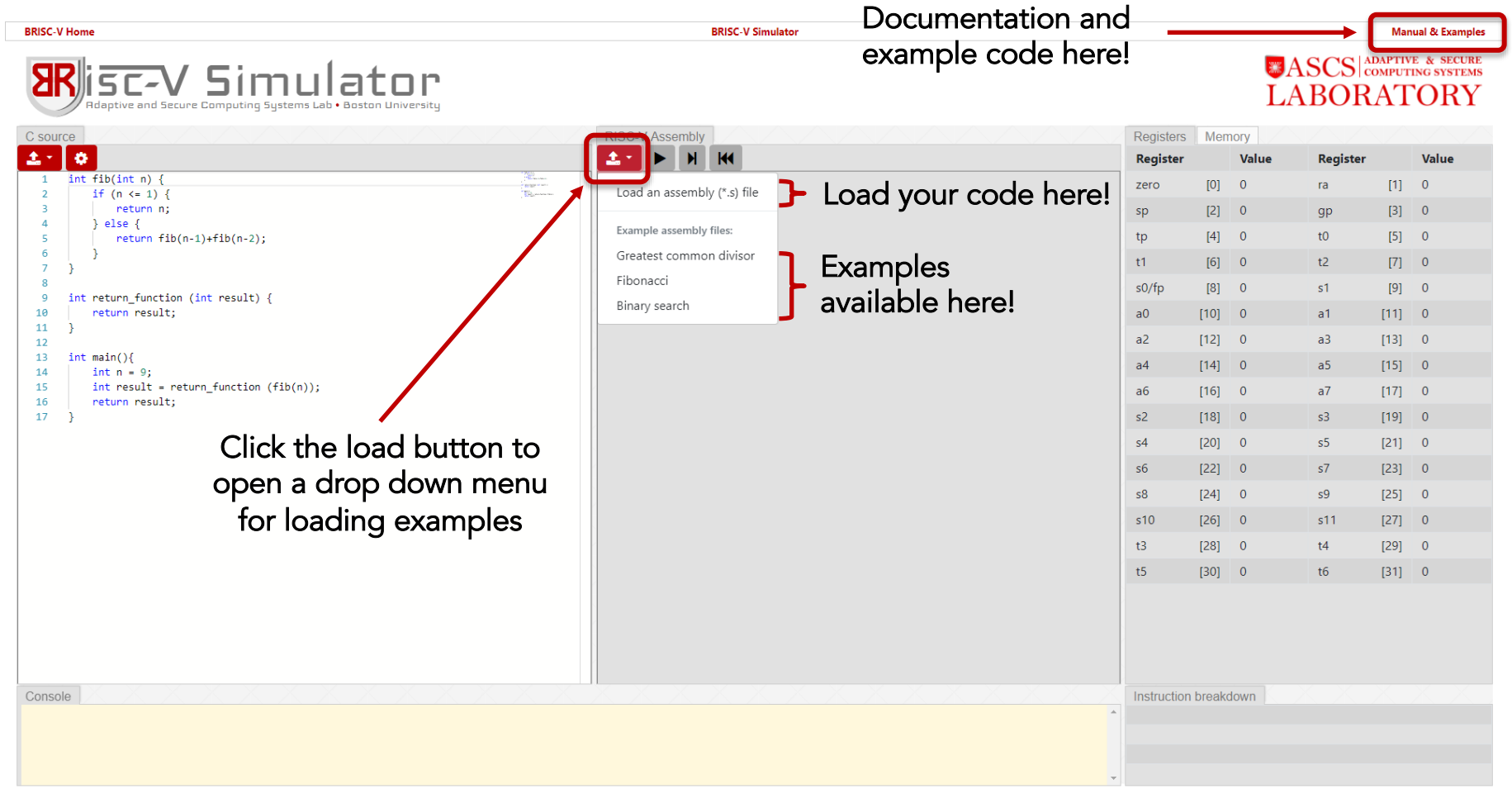
- BRISC-V Simulator let's you:
  - Run RISC-V assembly code in the browser
  - Debug hand-written assembly
  - Run code until completion or until it hits a breakpoint
  - Step through execution, instruction-by-instruction
  - View the state of memory and registers at each step
  - View how each instruction is constructed – opcodes, registers, immediate values etc.

https://ascslab.org/research/briscv/simulator/simulator.html

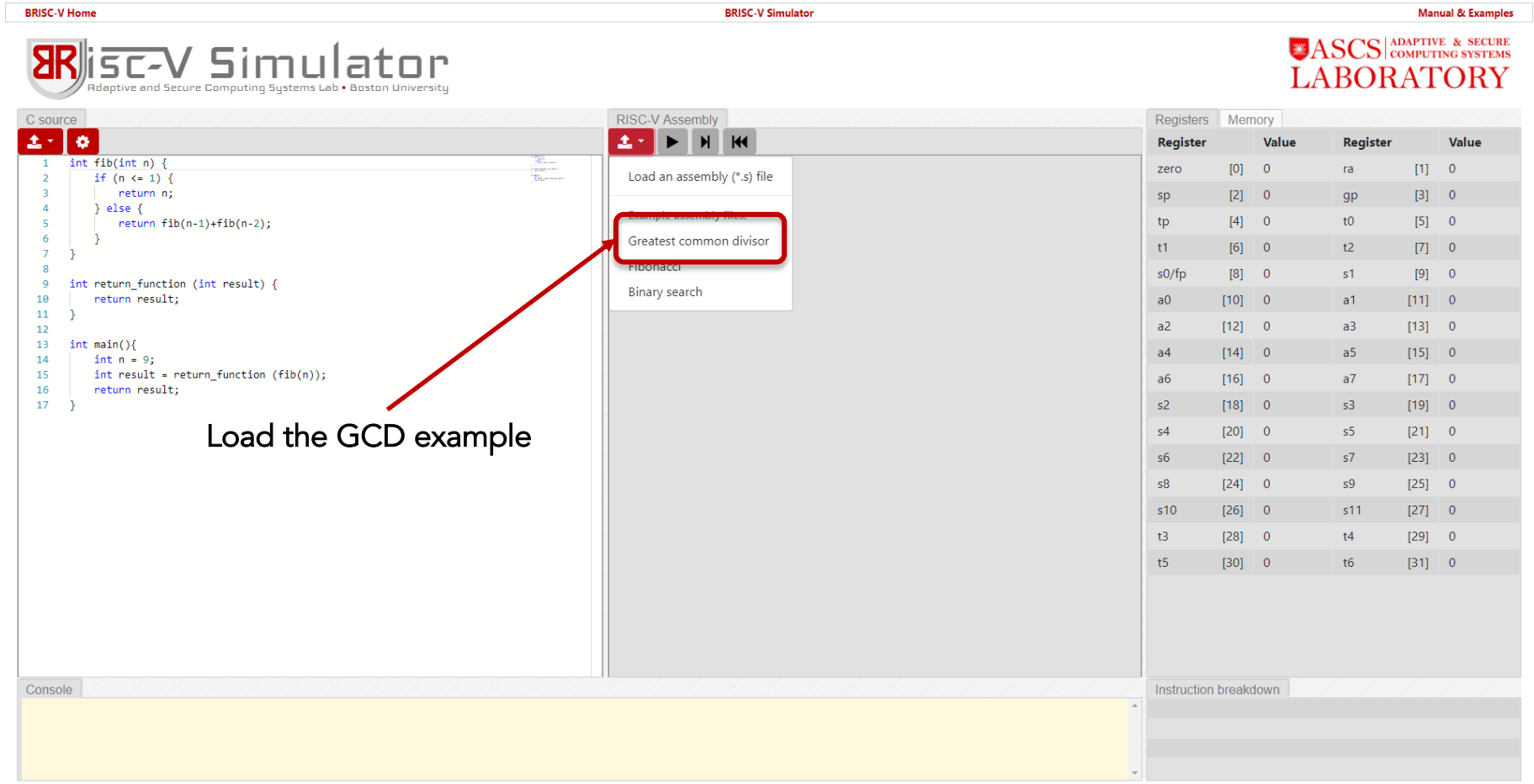
Let's get Familiar with the GUI



Don't have valid RISC-V assembly code to start with?

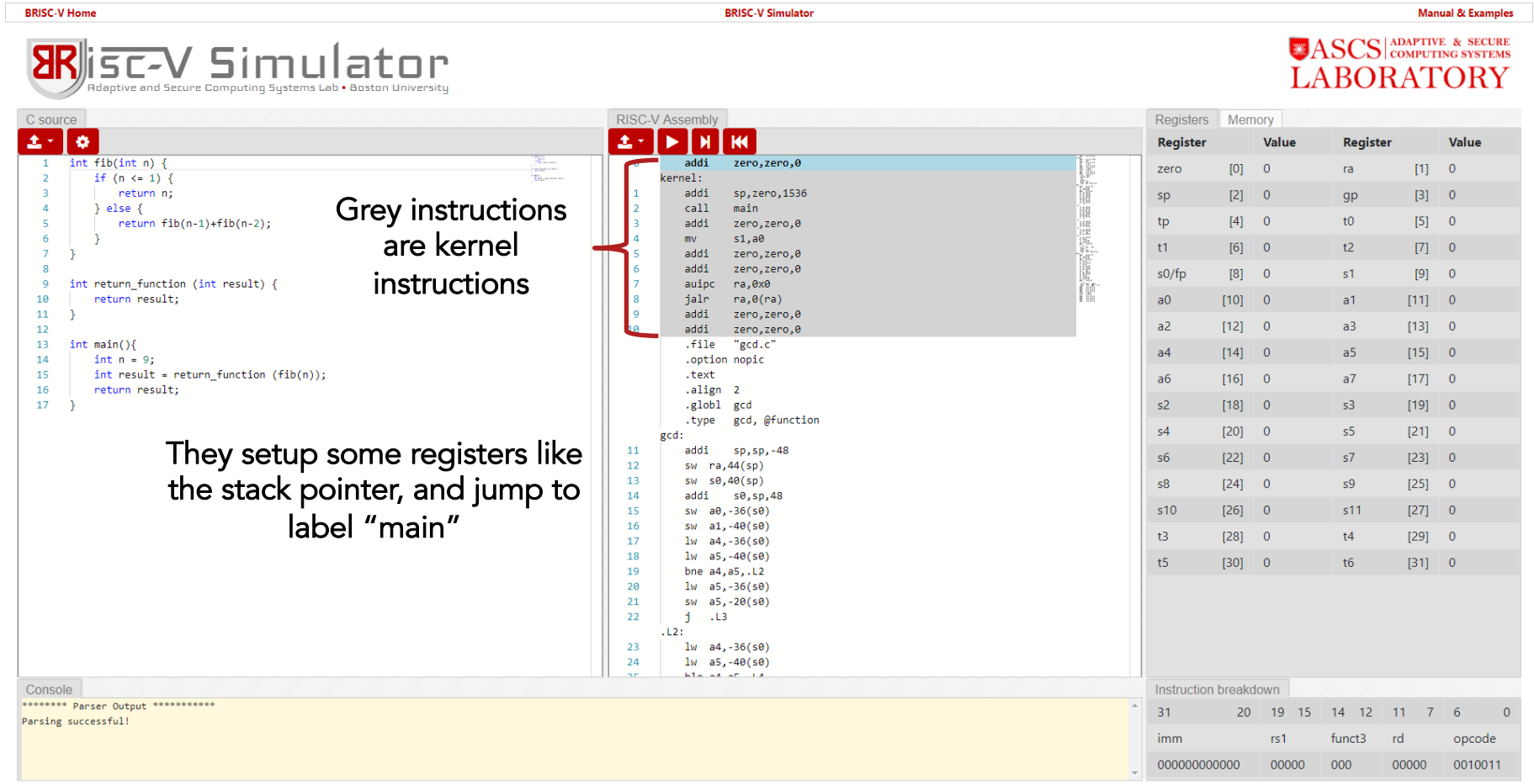


Don’t have valid RISC-V assembly code to start with?

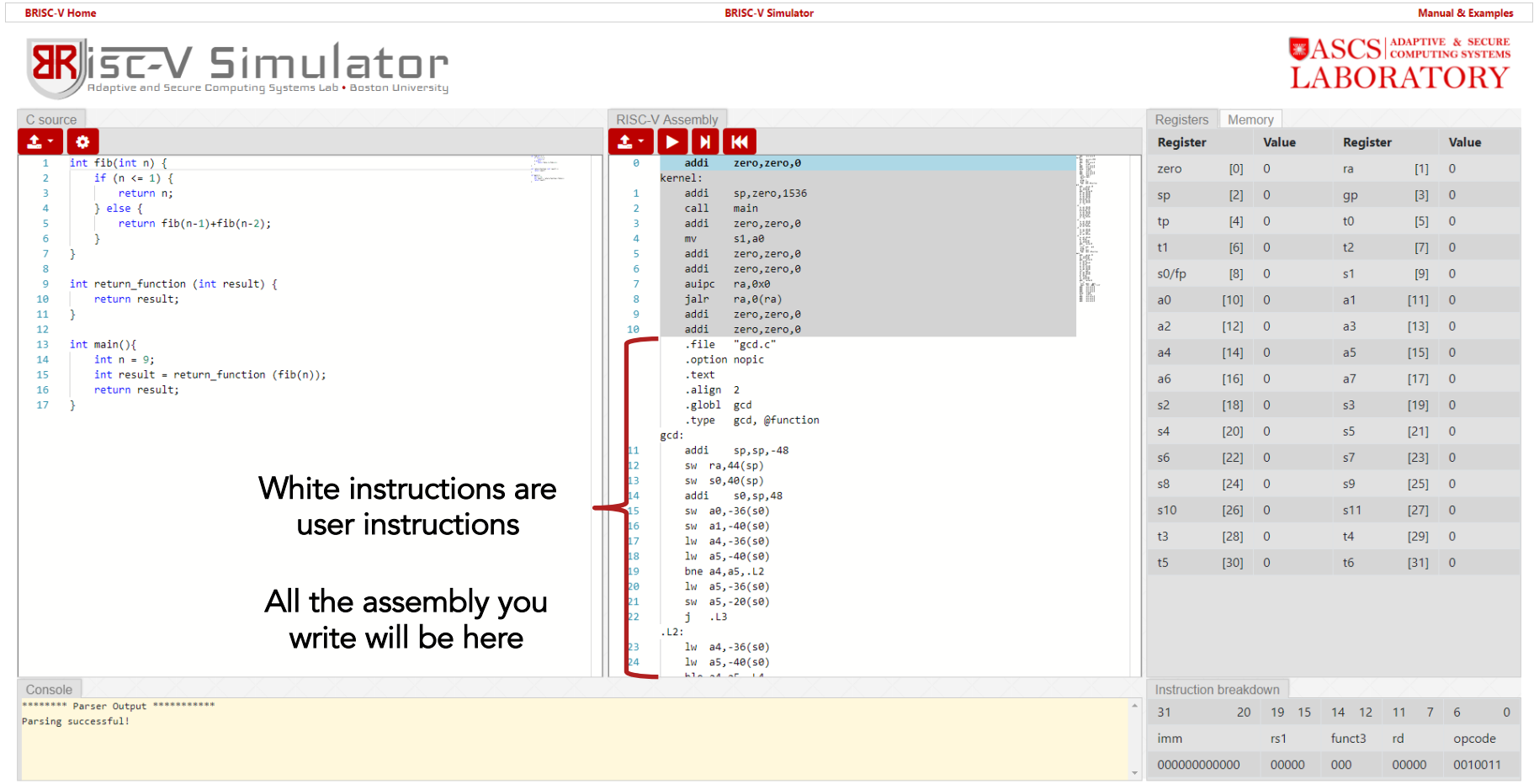


Kernel and User Instructions

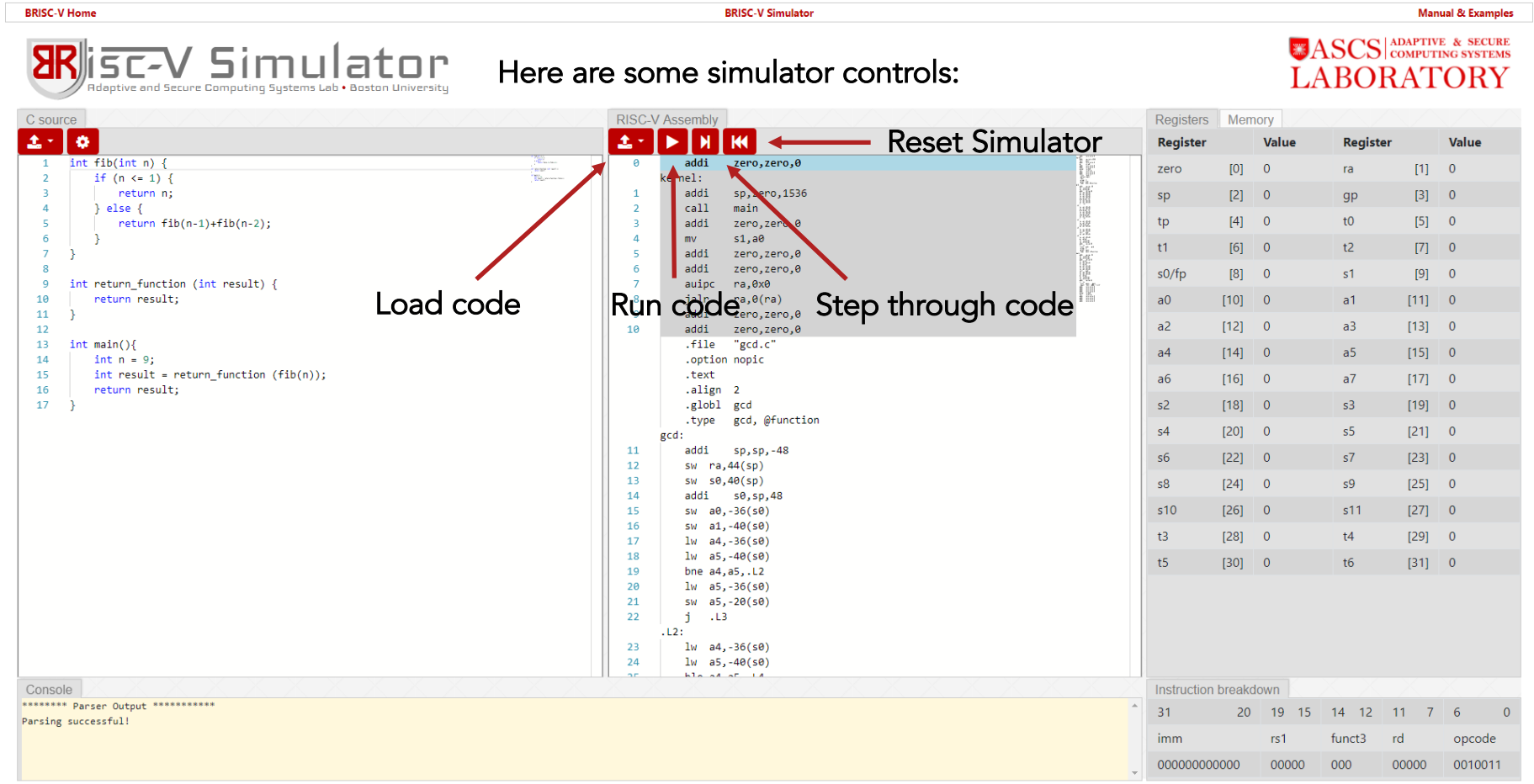
Kernel and User Instructions



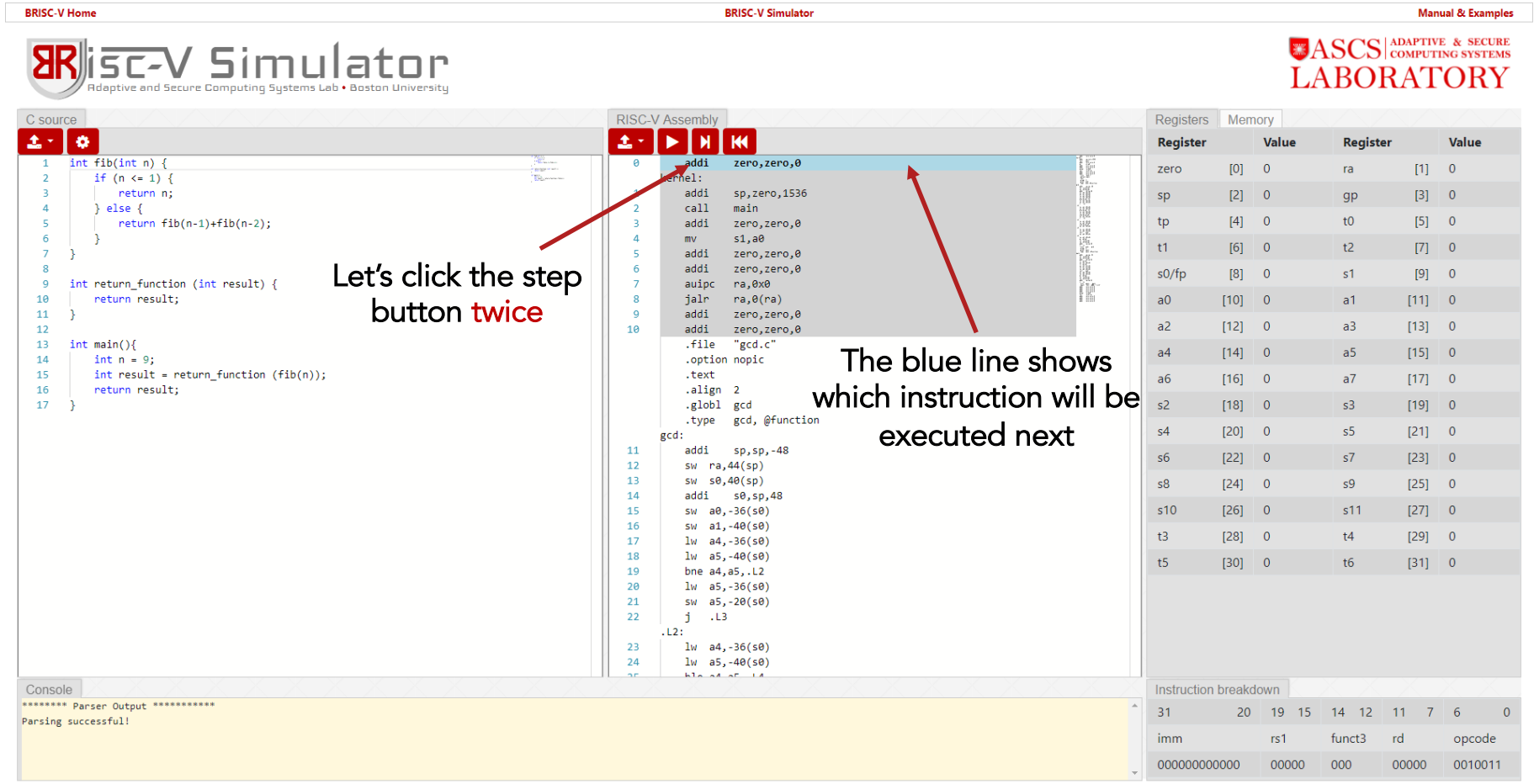
Kernel and User Instructions



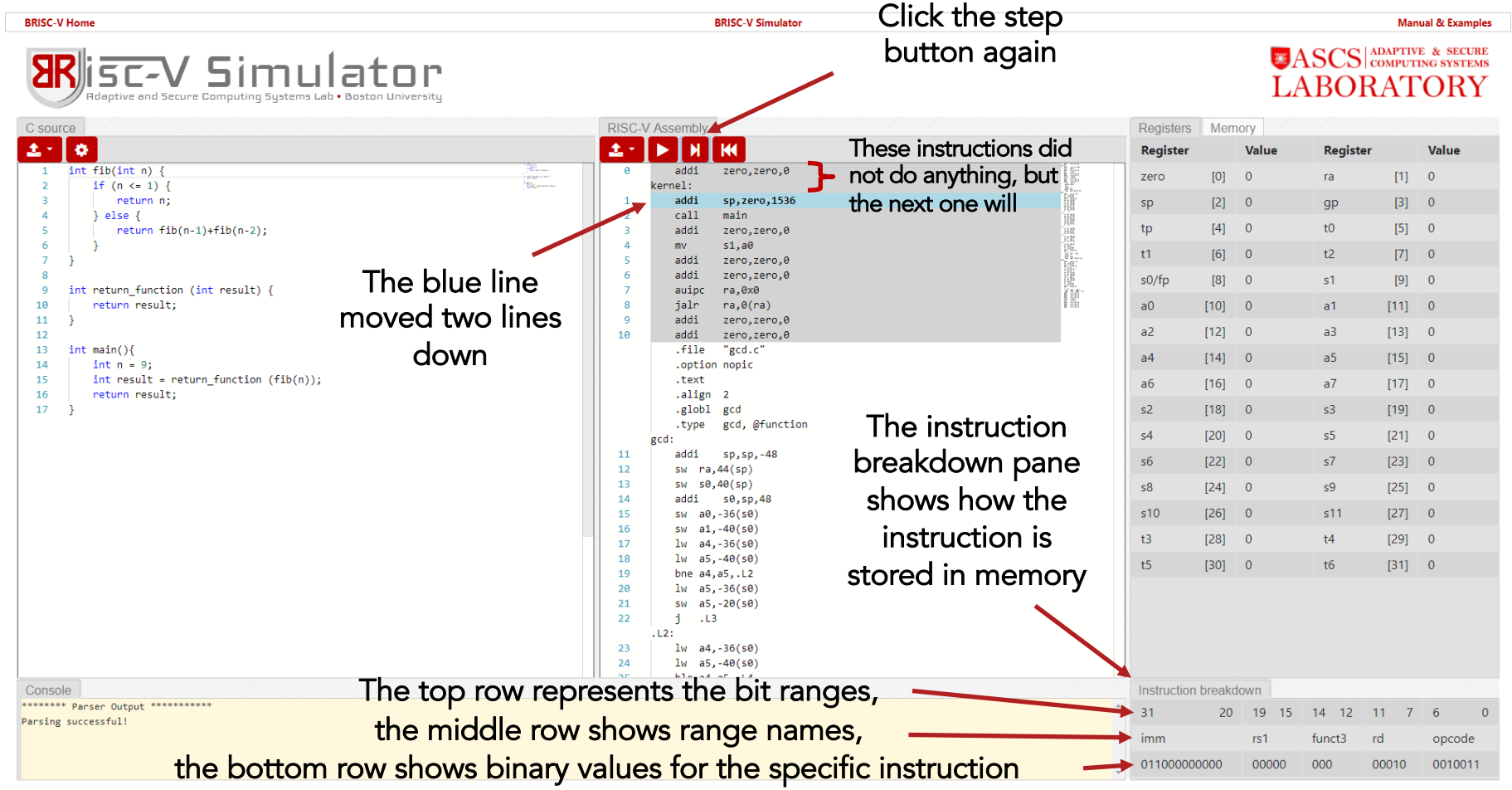
Simulator Controls



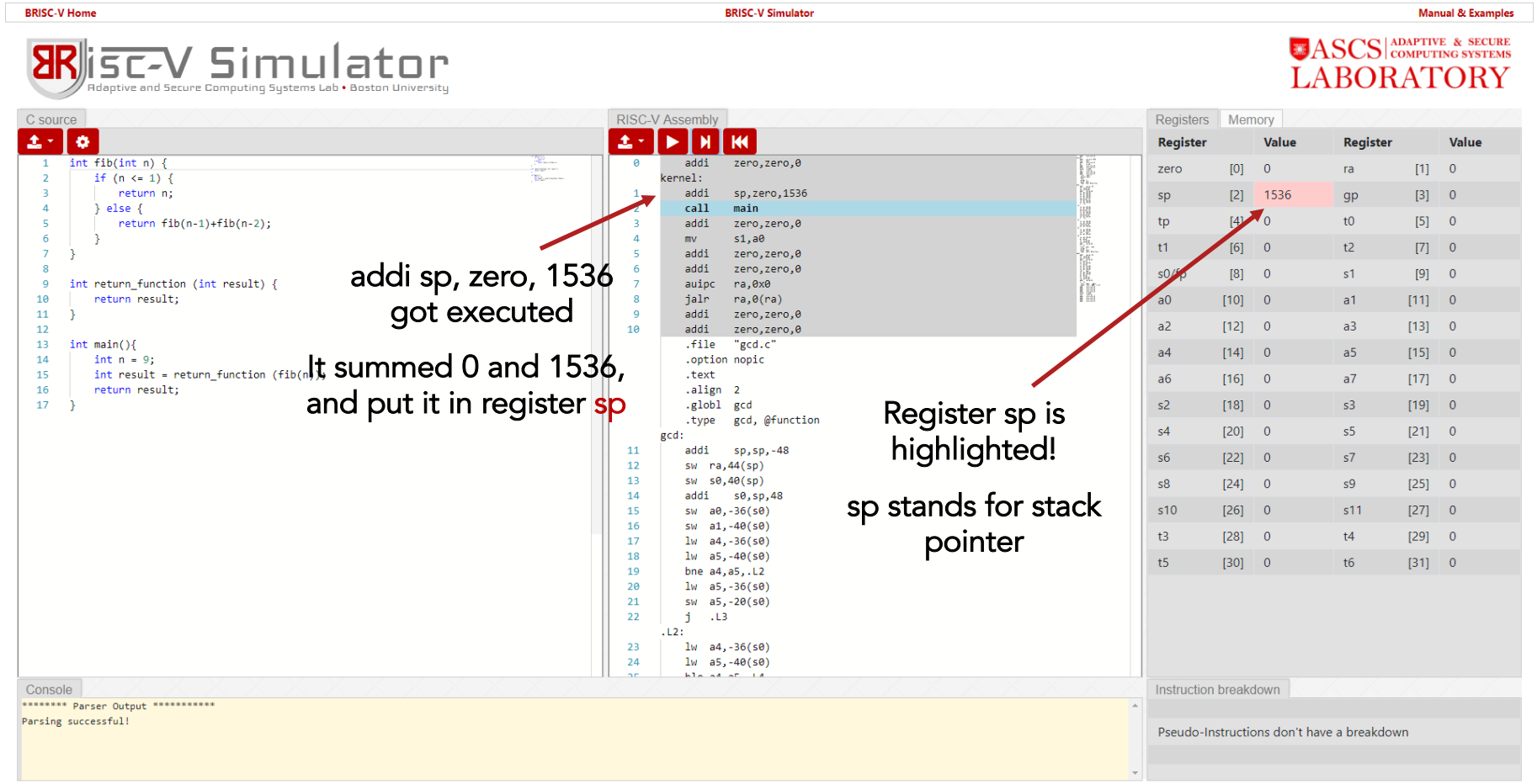
Stepping Through a Program



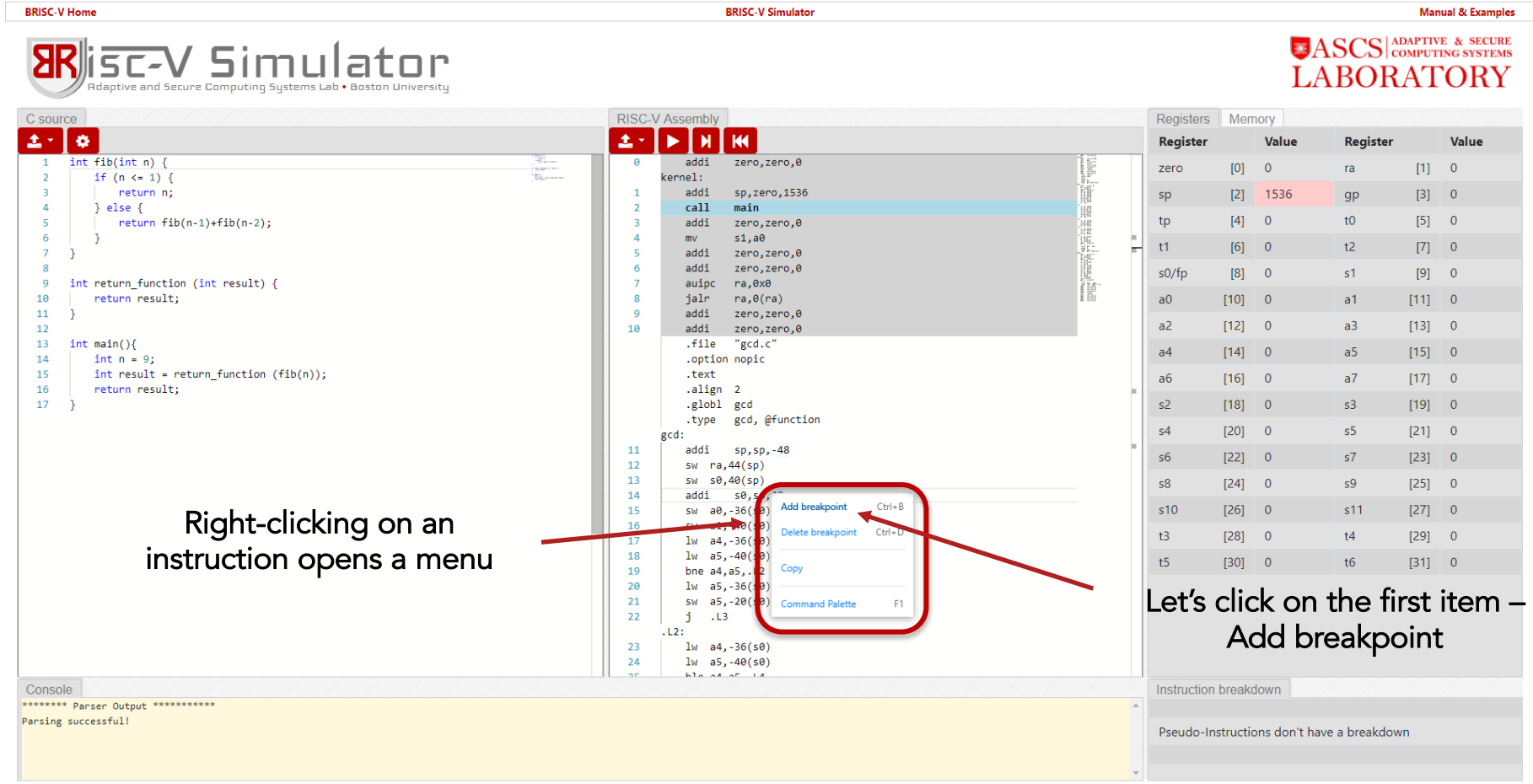
Stepping Through a Program



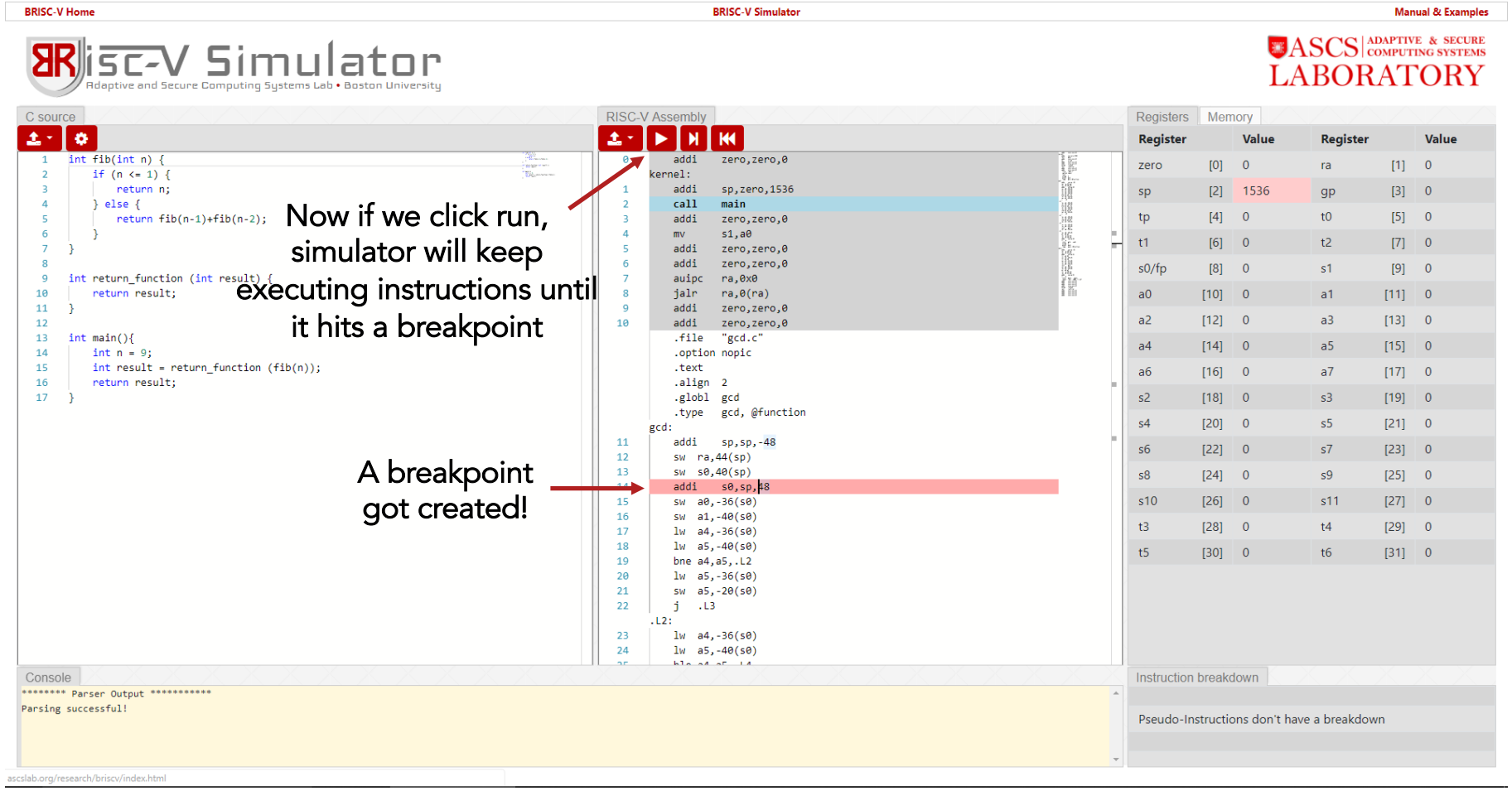
addi Changed the Register File



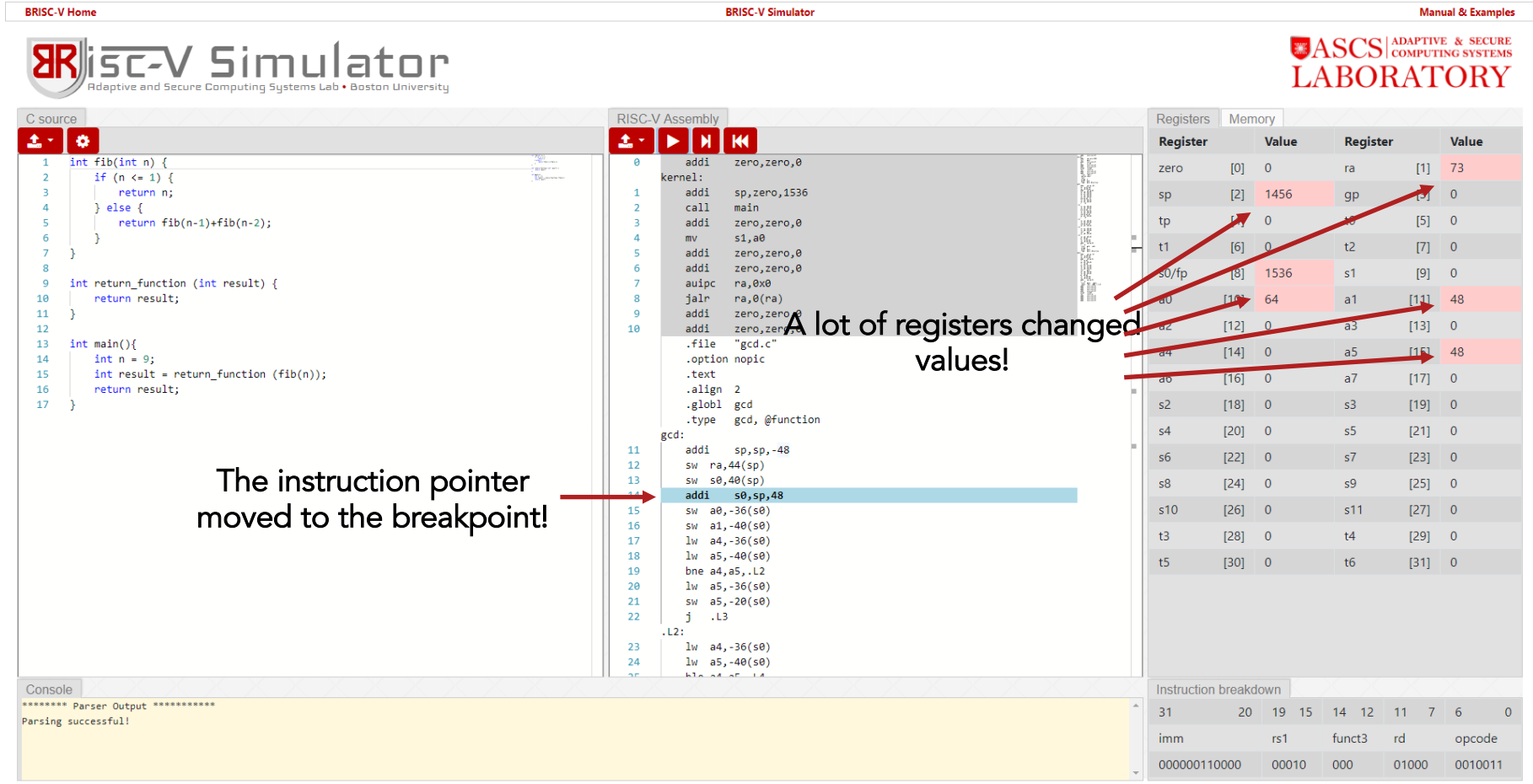
Setting Breakpoints



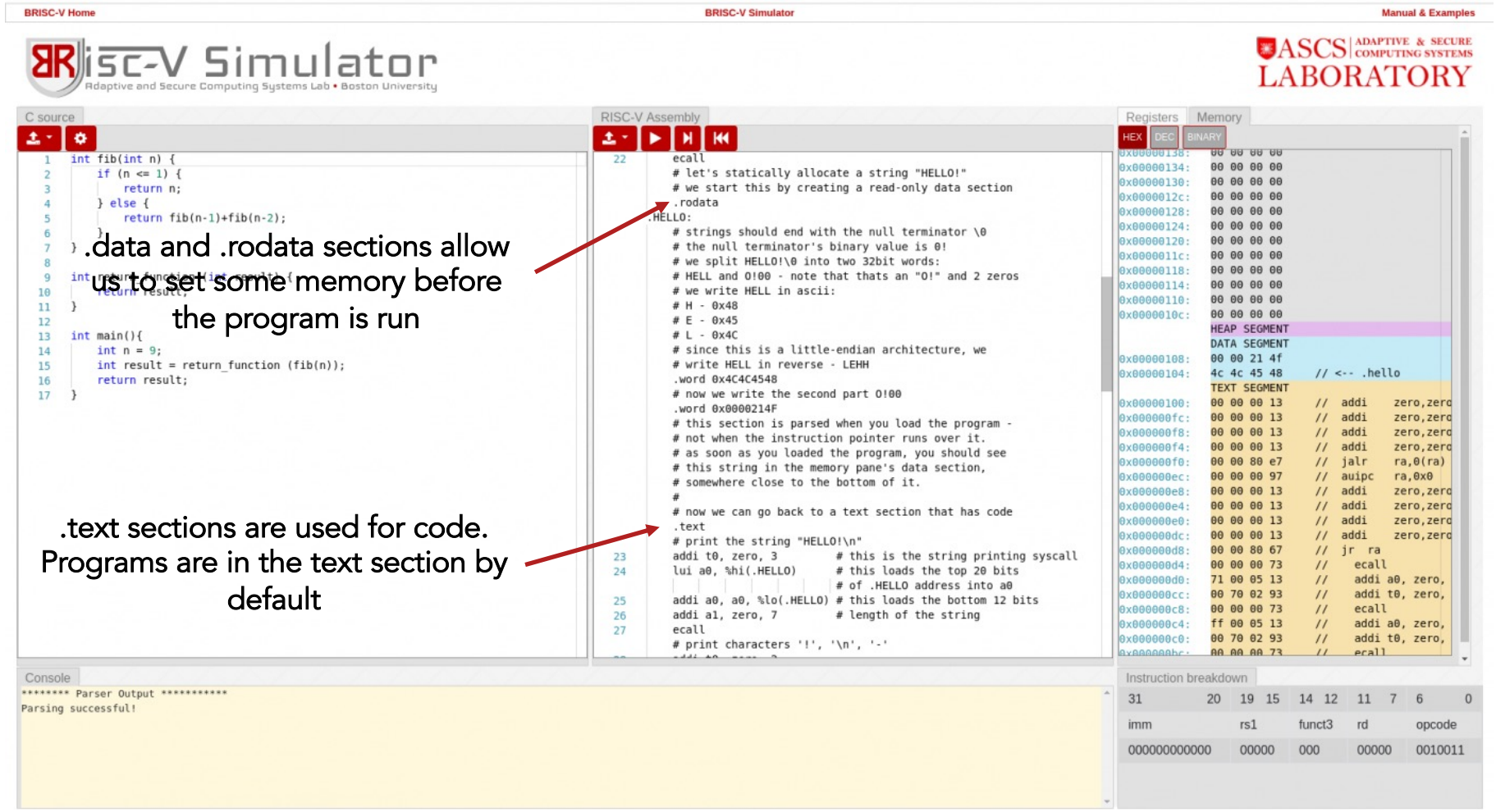
Setting Breakpoints



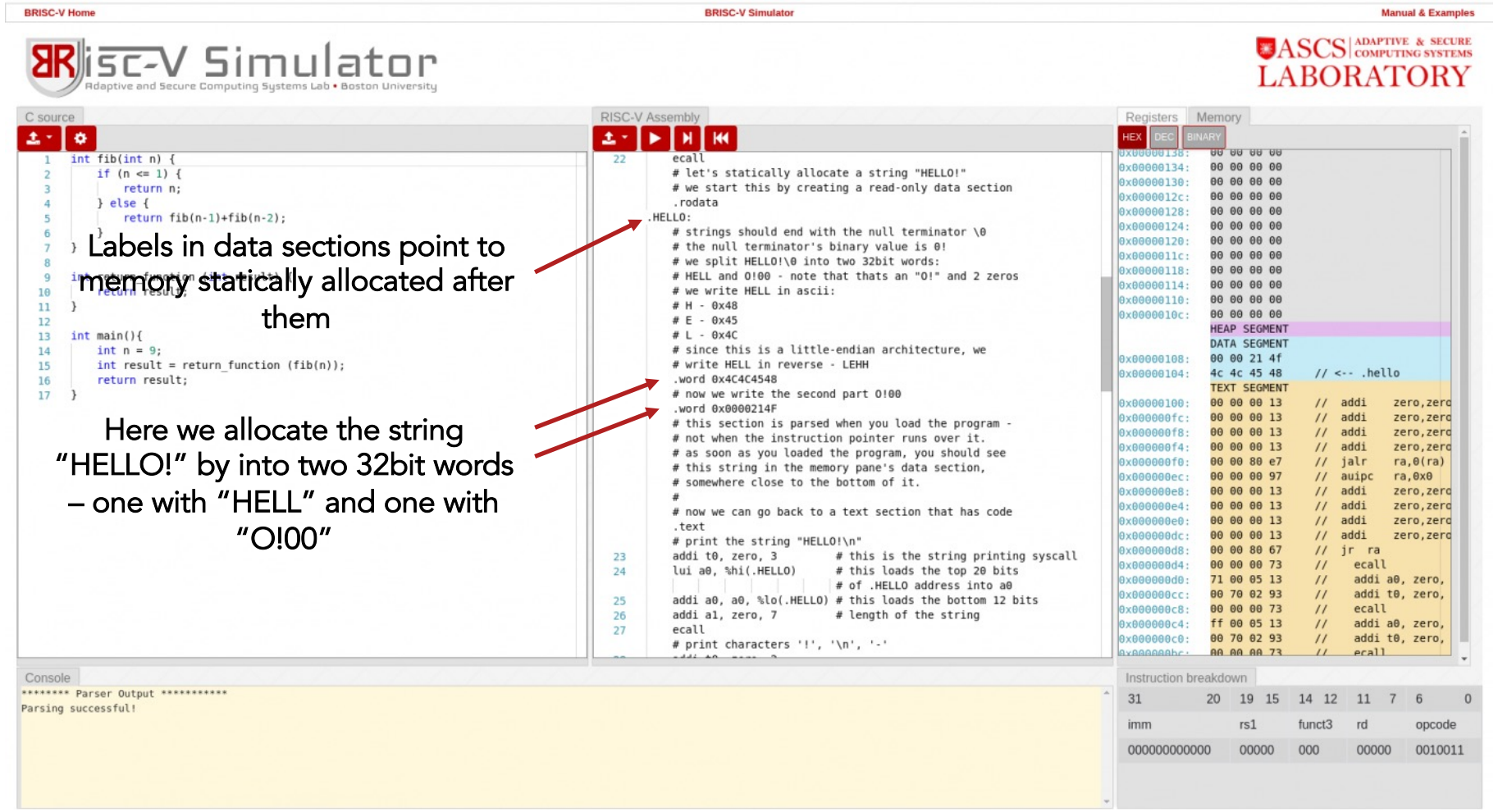
Running Code until a Breakpoint



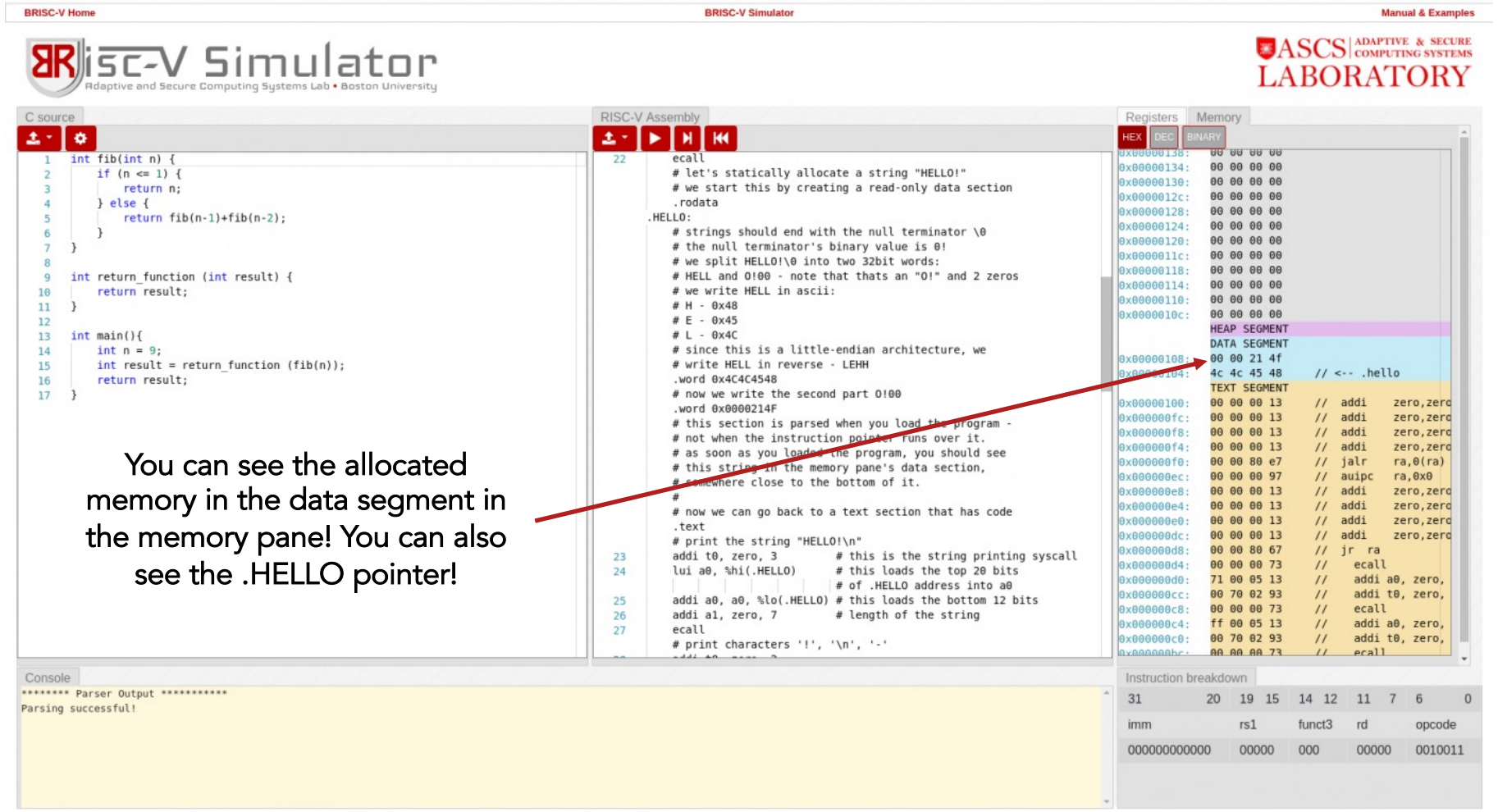
Text and Data Sections



Text and Data Sections



Text and Data Sections

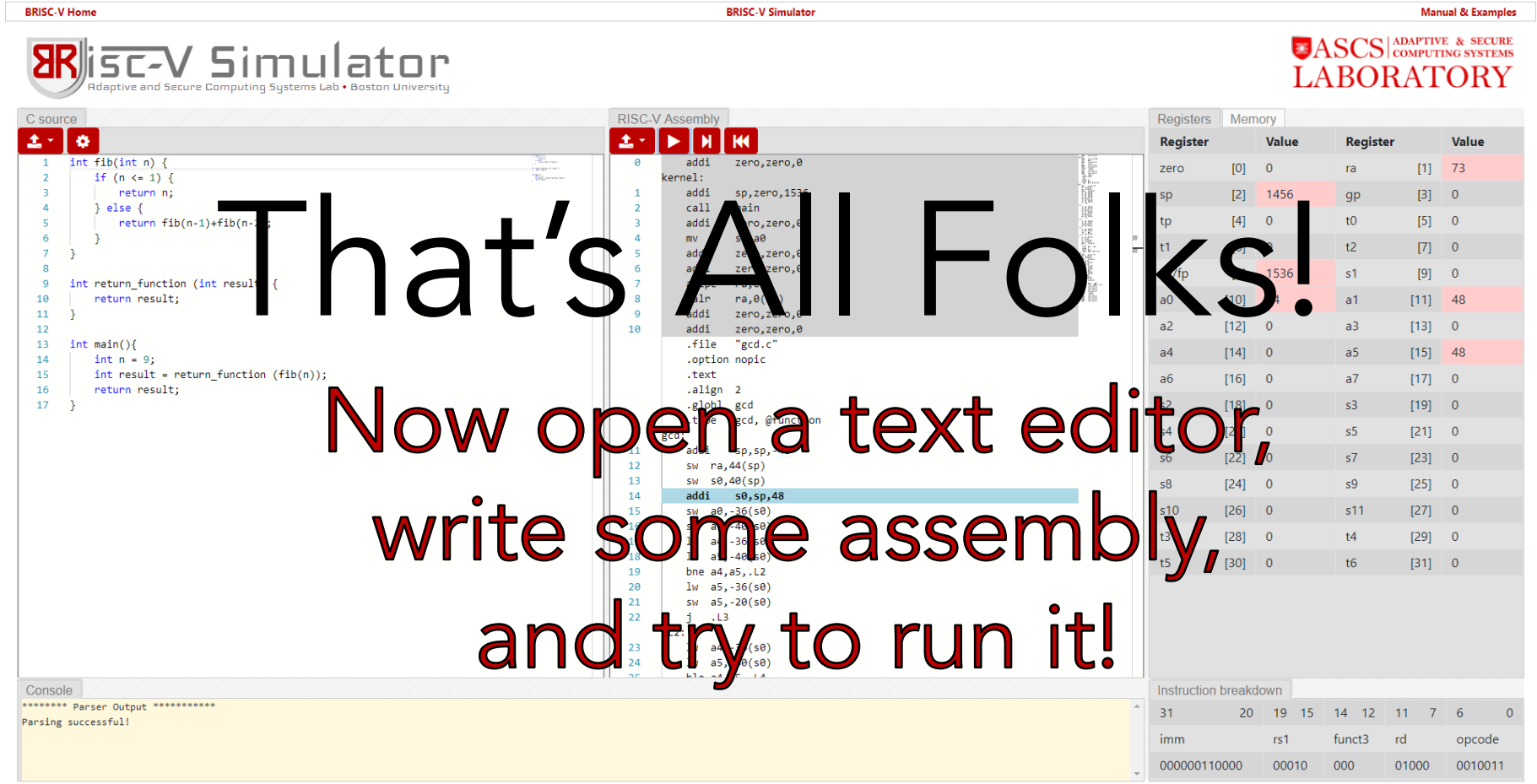


System Calls

- We also provide some simple system calls
- System calls are used for functionalities provided by the operating system
  - Think file systems, IO, etc.
- In RISC-V, system calls look something like:
  - Put the type of system call you want in register t0
    - More about that on the next slide
  - Put any arguments you may have in a0 and a1
  - Call instruction ECALL
  - If the system call has return values, they will be in a0
- To really get familiar with syscalls, try running the example syscall file in the simulator

Supported Syscalls

Syscall	Syscall ID (put this in t0)	Description
Print integer	1	Print integer value in a0 to console
Print char	2	Print ascii value in a0 to console
Print string	3	Print string with address in a0 and length in a1 to console
Read integer	4	Read integer from console into a0
Read char	5	Read character from console into a0 as an ascii value
Read string	6	Read string of length given in a1 from console and store it at address in a0
SBRK	7	Dynamically allocate the amount of bytes specified in a0. The pointer to the beginning of the newly allocated memory will be stored in a0. The value in a0 can be negative, if you want to deallocate some memory!


BRISC-V Home
BRISC-V Simulator
Manual & Examples
BRISC-V Simulator
Adaptive and Secure Computing Systems Lab • Boston University
ASCS ADAPTIVE & SECURE COMPUTING SYSTEMS LABORATORY
C source
RISC-V Assembly
Registers
Memory
Console
Instruction breakdown
That's All Folks!
Now open a text editor,
write some assembly,
and try to run it!

Bugs and Features

- Found a bug? Have an issue? Want a feature/RISC-V extension?
  - Send an email to briscv.feedback@gmail.com
- Thanks!