

CSE 420

Fundamentals of Computer Architecture

RISC-V ISA

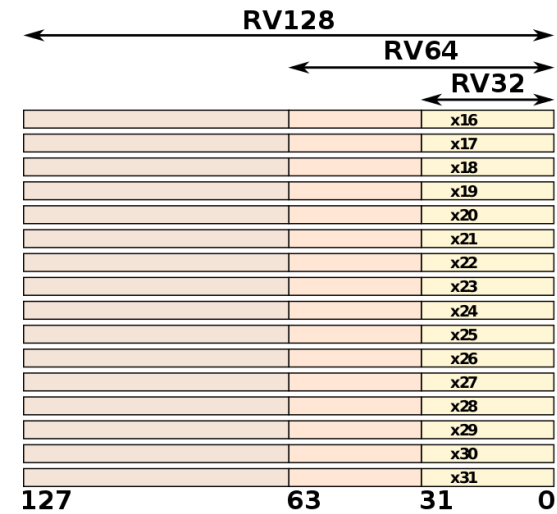
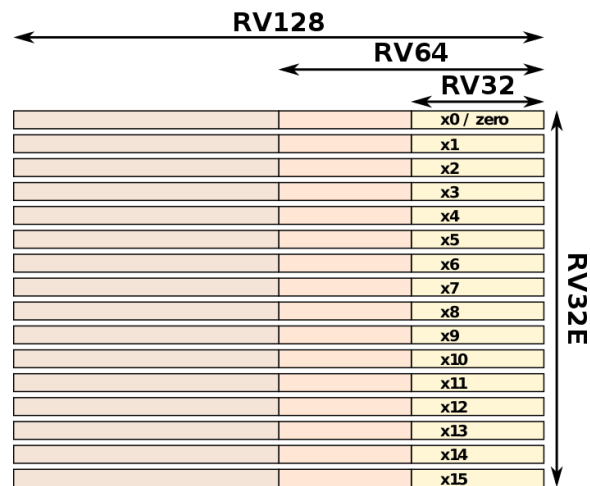
Prof. Michel A. Kinsy,
Instructors Mishel Paul & Nges Njungle

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)
 - $x0, x1, \dots, x31$
 - 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

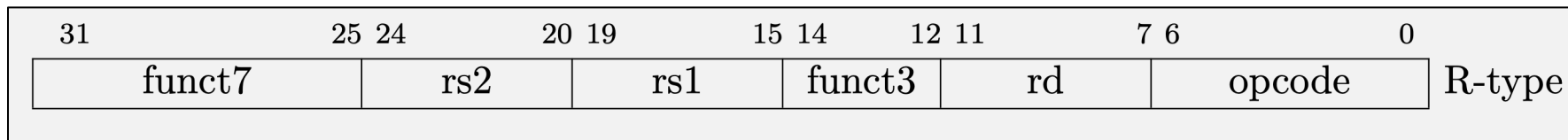
Base Instruction Formats

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

31	25 24	20 19	15 14	12 11	7 6	0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]					rd	opcode	U-type

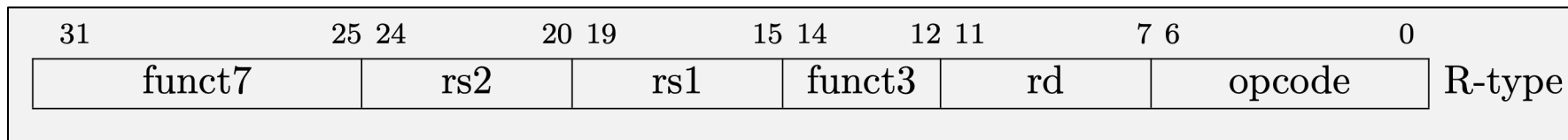
Base Instruction Formats

- 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

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
funct7				rs2			rs1		funct3		rd			opcode		R-type		
imm[11:0]						rs1		funct3		rd			opcode		I-type			
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]									rd			opcode			U-type			
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type			

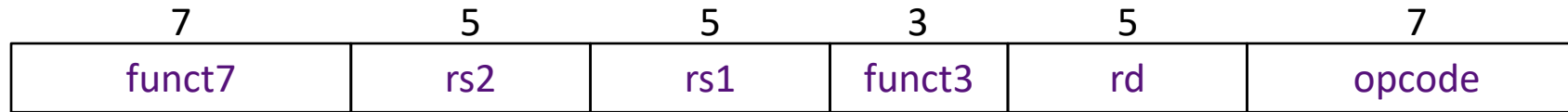
Base Instruction Formats

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

31	30	20	19	12	11	10	5	4	1	0	
— inst[31] —						inst[30:25]	inst[24:21]		inst[20]		I-immediate
— inst[31] —						inst[30:25]	inst[11:8]		inst[7]		S-immediate
— inst[31] —					inst[7]	inst[30:25]	inst[11:8]		0		B-immediate
inst[31]	inst[30:20]		inst[19:12]		— 0 —						U-immediate
— inst[31] —			inst[19:12]		inst[20]	inst[30:25]	inst[24:21]		0		J-immediate

Instruction Formats

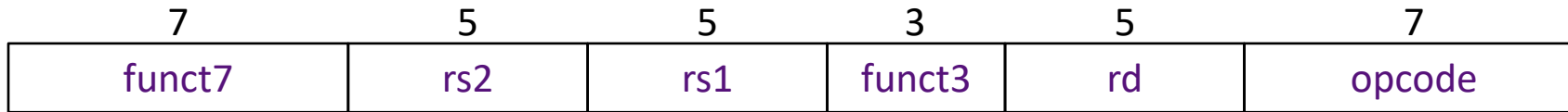
- R-type instruction



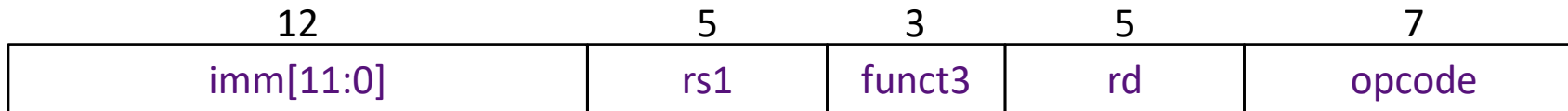
```
add x1, x2, x3           # x1 = x2 + x3
```

Instruction Formats

- R-type instruction



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



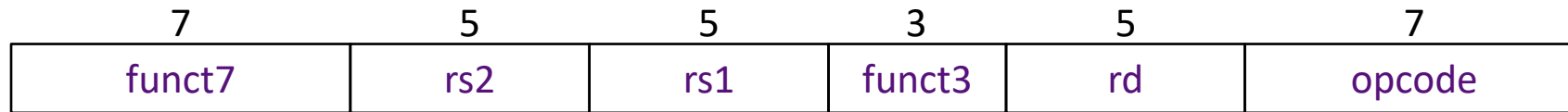
- I-imm = signExtend(inst[31:20])

```
add x1, x2, 413
```

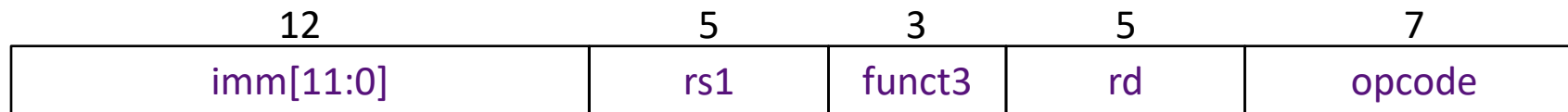
```
# x1 = x2 + 413
```

Instruction Formats

- R-type instruction

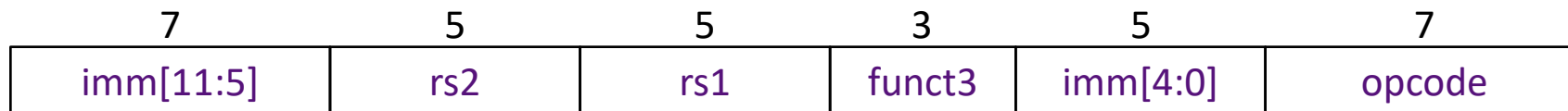


- I-type instruction & I-immediate (32 bits)



- I-imm = $\text{signExtend}(\text{inst}[31:20])$

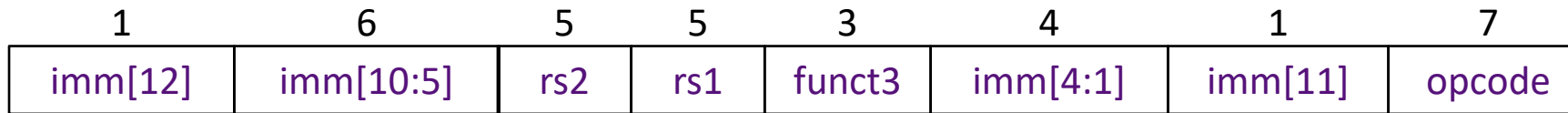
- S-type instruction & S-immediate (32 bits)



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

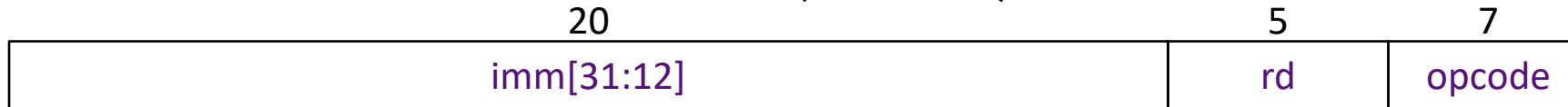
Instruction Formats

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



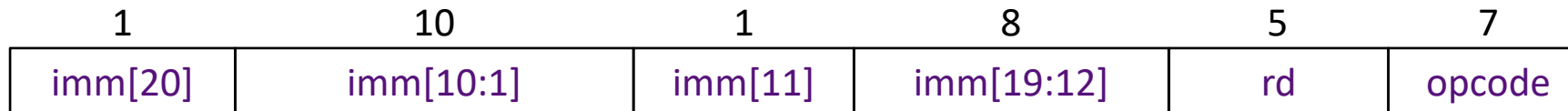
- B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})

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



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

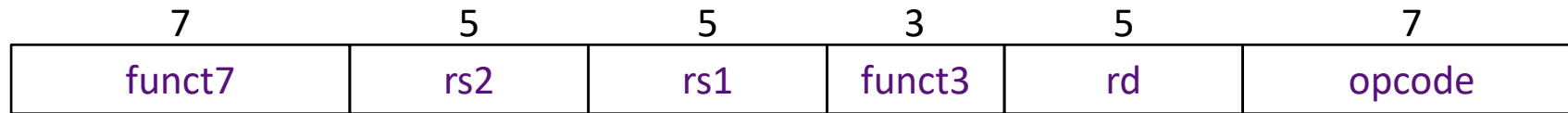
- UJ-type instruction & J-immediate (32 bits)



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

Computational Instructions

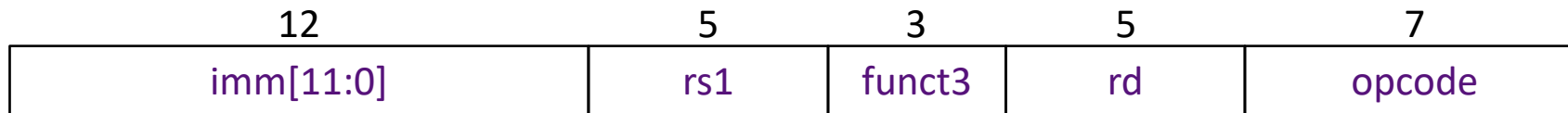
- Register-Register instructions (R-type)



- opcode=OP: $rd \leftarrow rs1 \text{ (funct3, funct7) } rs2$
- 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

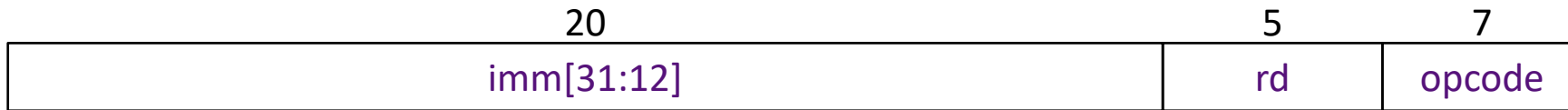
- Register-immediate instructions (I-type)



- opcode = OP-IMM: $rd \leftarrow rs1 \text{ (funct3) I-imm}$
 - I-imm = $\text{signExtend}(\text{inst}[31:20])$
 - funct3 = ADDI/SLTI/SLTIU/ANDI/ORI/XORI
- A slight variant in coding for shift instructions - SLLI / SRLI / SRAI
 - $rd \leftarrow rs1 \text{ (funct3, inst[30]) I-imm}[4:0]$

Computational Instructions

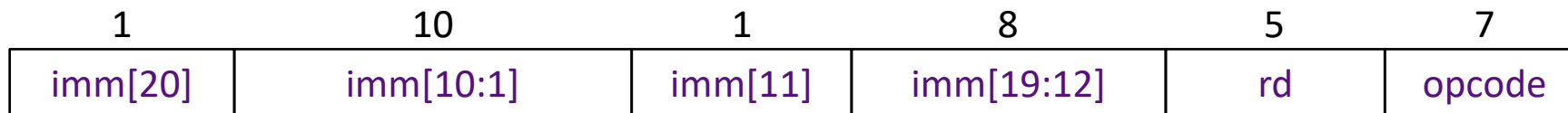
- Register-immediate instructions (U-type)



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

Control Instructions

■ Unconditional jump and link (UJ-type)



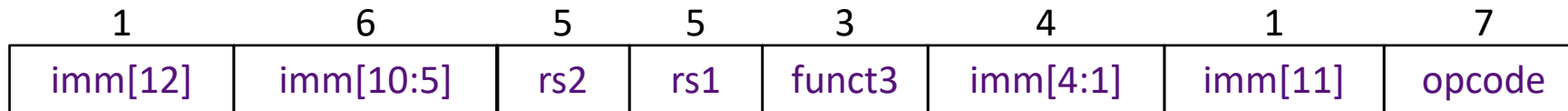
- opcode = JAL: $rd \leftarrow pc + 4; pc \leftarrow pc + J-imm$
 - J-imm = $signExtend(\{inst[31], inst[19:12], inst[20], inst[30:21], 1'b0\})$
 - Jump $\pm 1MB$ range
- ## ■ Unconditional jump via register and link (I-type)



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

Control Instructions

- Conditional branches (SB-type)



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

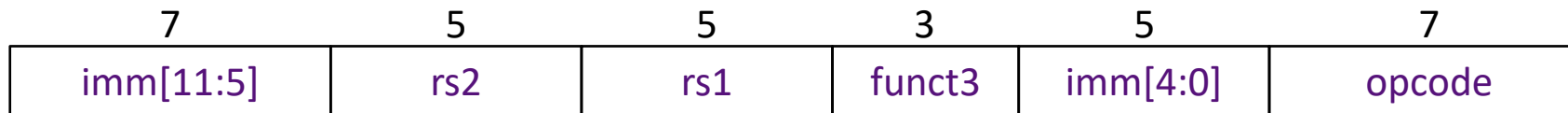
Load & Store Instructions

- Load (I-type)



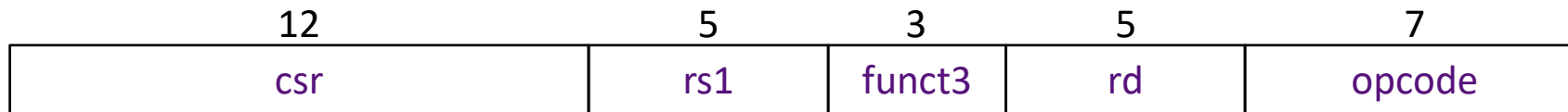
- opcode = LOAD: $rd \leftarrow \text{mem}[rs1 + \text{I-imm}]$
- I-imm = $\text{signExtend}(\text{inst}[31:20])$
- funct3 = LW/LB/LBU/LH/LHU

- Store (S-type)



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

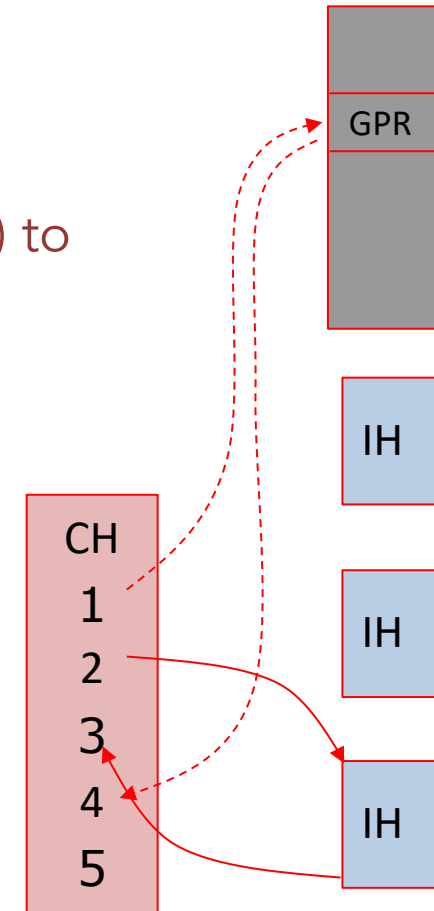
Instructions to Read and Write CSR



- opcode = SYSTEM
- CSRW rs1, csr (funct3 = CSRRW, rd = x0): $\text{csr} \leftarrow \text{rs1}$
- CSRR csr, rd (funct3 = CSRRS, rs1 = x0): $\text{rd} \leftarrow \text{csr}$

Software for interrupt handling

- Hardware transfers control to the common software interrupt handler (CH) which:
 - Saves all GPRs into the memory pointed by `mscratch`
 - Passes `mcause`, `mepc`, stack pointer to the IH (a C function) to handle the specific interrupt
 - On the return from the IH, writes the return value to `mepc`
 - Loads all GPRs from the memory
 - 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
<i>Load Byte</i>	<i>I</i>	<i>LB rd,rs1,imm</i>
<i>Load Halfword</i>	<i>I</i>	<i>LH rd,rs1,imm</i>
<i>Load Word</i>	<i>I</i>	<i>LW rd,rs1,imm</i>
<i>Load Byte Unsigned</i>	<i>I</i>	<i>LBU rd,rs1,imm</i>
<i>Load Half Unsigned</i>	<i>I</i>	<i>LHU rd,rs1,imm</i>

Instruction Functions

- Data movement

Operation	Type	Template
<i>Load Byte</i>	<i>I</i>	<i>LB rd,rs1,imm</i>
<i>Load Halfword</i>	<i>I</i>	<i>LH rd,rs1,imm</i>
<i>Load Word</i>	<i>I</i>	<i>LW rd,rs1,imm</i>
<i>Load Byte Unsigned</i>	<i>I</i>	<i>LBU rd,rs1,imm</i>
<i>Load Half Unsigned</i>	<i>I</i>	<i>LHU rd,rs1,imm</i>

Operation	Type	Template
<i>Store Byte</i>	<i>S</i>	<i>SB rs1,rs2,imm</i>
<i>Store Halfword</i>	<i>S</i>	<i>SH rs1,rs2,imm</i>
<i>Store Word</i>	<i>S</i>	<i>SW rs1,rs2,imm</i>

Instruction Functions

- Arithmetic & Logic

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

Instruction Functions

■ Arithmetic & Logic

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

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

Instruction Functions

- Arithmetic & Logic

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

Instruction Functions

- Arithmetic & Logic

Operation	Type	Template
<i>XOR</i>	<i>R</i>	<i>XOR rd,rs1,rs2</i>
<i>XOR Immediate</i>	<i>I</i>	<i>XORI rd,rs1,imm</i>
<i>OR R</i>	<i>R</i>	<i>OR rd,rs1,rs2</i>
<i>R Immediate</i>	<i>I</i>	<i>ORI rd,rs1,imm</i>
<i>AND</i>	<i>R</i>	<i>AND rd,rs1,rs2</i>
<i>AND Immediate</i>	<i>I</i>	<i>ANDI rd,rs1,imm</i>

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

Instruction Functions

- Control Flow

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

Instruction Functions

- Control Flow

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

Operation	Type	Template
<i>Jump & Link</i>	<i>UJ</i>	<i>JAL rd,imm</i>
<i>Jump & Link Register</i>	<i>UJ</i>	<i>JALR rd,rs1,imm</i>

Instruction Functions

- System call

Operation	Type	Template
<i>System CALL</i>	<i>I</i>	<i>SCALL</i>
<i>System BREAK</i>	<i>I</i>	<i>SBREAK</i>

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

BRISC-V Home
BRISC-V Simulator
Manual & Examples



Adaptive and Secure Computing Systems Lab • Boston University



ADAPTIVE & SECURE
COMPUTING SYSTEMS
LABORATORY

C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }
```

C Source Code Pane

Not interesting for this class

Let's you compile C to RISC-V assembly

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0
11
12     .file    "gcd.c"
13     .option  nopic
14     .text
15     .align   2
16     .globl   gcd
17     .type    gcd, @function
18
19 gcd:
20     addi    sp,sp,-48
21     sw      ra,44(sp)
22     sw      s0,40(sp)
23     addi    s0,sp,48
24     sw      a0,-36(s0)
25     sw      a1,-40(s0)
26     lw      a4,-36(s0)
27     lw      a5,-40(s0)
28     bne     a4,a5,.L2
29     lw      a5,-36(s0)
30     lw      a5,-20(s0)
31     j       .L3
32
33 .L2:
34     lw      a4,-36(s0)
35     lw      a5,-40(s0)
36
```

**RISC-V
Assembly
Pane**

Let's you run
assembly step-
by-step

Registers

Memory

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Register & Memory Pane

Shows the state of
registers and memory

Console

```

***** Parser Output *****
Parsing successful!
```

Console Pane

Shows messages from the compiler and the simulator

Also used for system calls – let's you input and print values

Instruction Breakdown

31	20	19	15	14	12	11	7	6	0
imm	rs1	funct3	rd	opcode					
000000000000	00000	000	00000	0010011					

Instruction Breakdown Pane

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

BRISC-V Home
BRISC-V Simulator

Documentation and example code here! Manual & Examples

ASCS LABORATORY
ADAPTIVE & SECURE COMPUTING SYSTEMS

BRISC-V Simulator
Adaptive and Secure Computing Systems Lab • Boston University

C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }
```

Click the load button to open a drop down menu for loading examples

RISC-V Assembly

Load an assembly (*.s) file

Example assembly files:

- Greatest common divisor
- Fibonacci
- Binary search

Load your code here!

Examples available here!

Registers Memory

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Instruction breakdown

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

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

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

RISC-V Assembly

Load an assembly (*.s) file

Greatest common divisor

Fibonacci

Binary search

Registers

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Console

Instruction breakdown

Load the GCD example

Kernel and User Instructions

BRISC-V Home
BRISC-V Simulator
Manual & Examples



Adaptive and Secure Computing Systems Lab • Boston University



ADAPTIVE & SECURE
COMPUTING SYSTEMS
LABORATORY

C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }
```

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0
.file   "gcd.c"
.option nopic
.text
.align 2
.globl  gcd
.type   gcd, @function
gcd:
11     addi    sp,sp,-48
12     sw      ra,44(sp)
13     sw      s0,40(sp)
14     addi    s0,sp,48
15     sw      a0,-36(s0)
16     sw      a1,-40(s0)
17     lw      a4,-36(s0)
18     lw      a5,-40(s0)
19     bne     a4,a5,.L2
20     lw      a5,-36(s0)
21     sw      a5,-20(s0)
22     j       .L3
.L2:
23     lw      a4,-36(s0)
24     lw      a5,-40(s0)
25     bne     a4,a5,.L2
26     j       .L3
.L3:
27     ret
```

Registers

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Console

```

***** Parser Output *****
Parsing successful!
```

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
000000000000		00000		000		00000		0010011	

Our code got loaded!

**The console says that it parsed the file without problems
If there were problems, they would pop up here**

Kernel and User Instructions

BRISC-V Home
BRISC-V Simulator
Manual & Examples



C source

```

1 int fib(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fib(n-1)+fib(n-2);
6     }
7 }
8
9 int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

Grey instructions are kernel instructions

They setup some registers like the stack pointer, and jump to label "main"

Console

```

***** Parser Output *****
Parsing successful!

```

RISC-V Assembly

```

1 addi zero,zero,0
2
3 kernel:
4     addi sp,zero,1536
5     call main
6     addi zero,zero,0
7     mv s1,a0
8     addi zero,zero,0
9     addi zero,zero,0
10    addi zero,zero,0
11
12 .file "gcd.c"
13 .option nopic
14 .text
15 .align 2
16 .globl gcd
17 .type gcd, @function
18
19 gcd:
20     addi sp,sp,-48
21     sw ra,44(sp)
22     sw s0,40(sp)
23     addi s0,sp,48
24     sw a0,-36(s0)
25     sw a1,-40(s0)
26     lw a4,-36(s0)
27     lw a5,-40(s0)
28     bne a4,a5,.L2
29     lw a5,-36(s0)
30     sw a5,-20(s0)
31     j .L3
32
33 .L2:
34     lw a4,-36(s0)
35     lw a5,-40(s0)
36     bne a4,a5,.L2

```

Registers

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
000000000000		00000		000		00000		0010011	

Kernel and User Instructions

BRISC-V Home
BRISC-V Simulator
Manual & Examples




C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0
11     .file    "gcd.c"
12     .option  nopie
13     .text
14     .align   2
15     .globl   gcd
16     .type    gcd, @function
gcd:
17     addi     sp,sp,-48
18     sw       ra,44(sp)
19     sw       s0,40(sp)
20     addi     s0,sp,48
21     sw       a0,-36(s0)
22     sw       a1,-40(s0)
23     lw       a4,-36(s0)
24     lw       a5,-40(s0)
25     bne      a4,a5,.L2
26     lw       a5,-36(s0)
27     sw       a5,-20(s0)
28     j        .L3
.L2:
29     lw       a4,-36(s0)
30     lw       a5,-40(s0)
31     bne      a4,a5,.L4

```

Registers

Memory

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Console

```

***** Parser Output *****
Parsing successful!

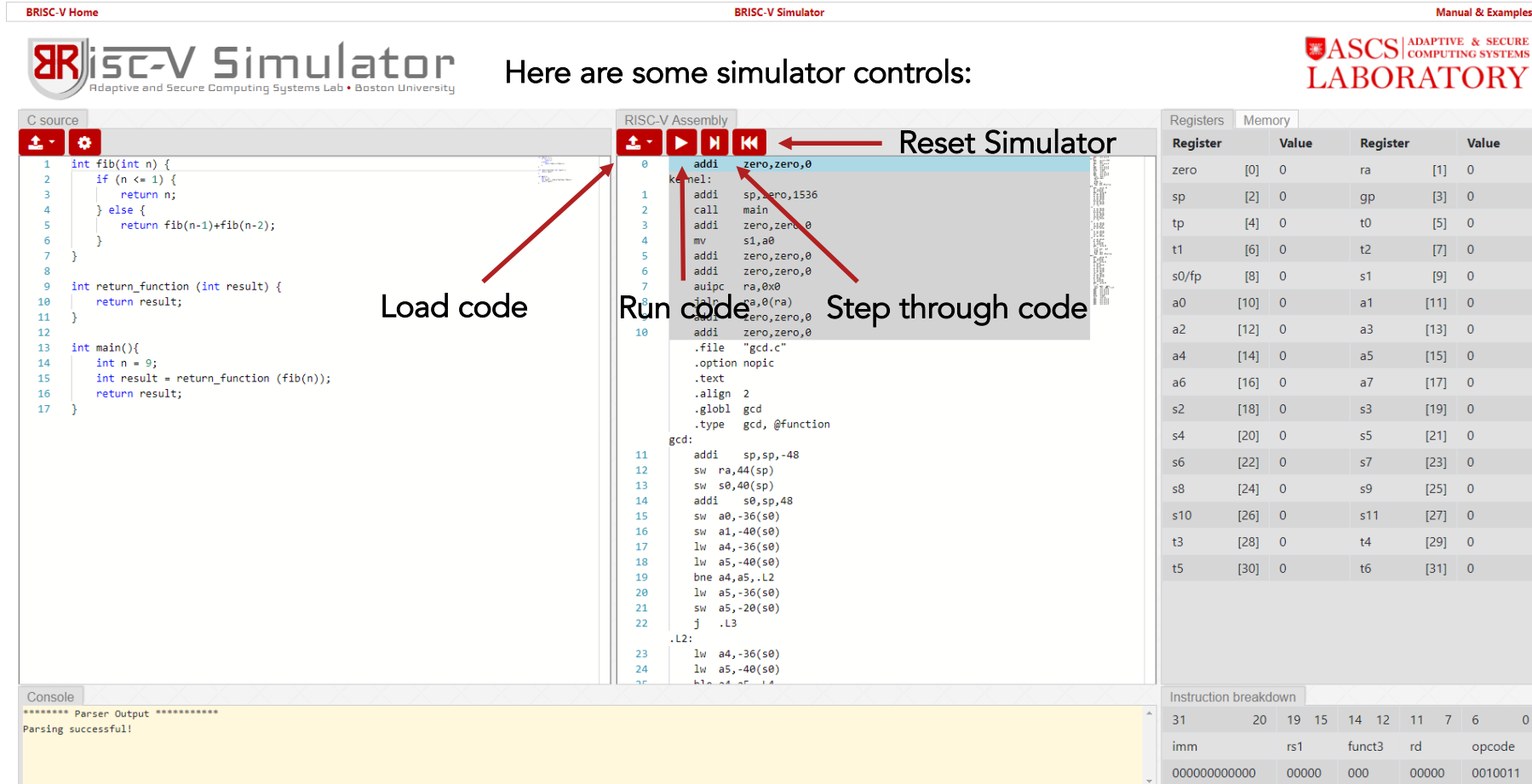
```

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
00000000000000		00000		000		00000		0010011	

White instructions are user instructions

All the assembly you write will be here



BRISC-V Home
BRISC-V Simulator
Manual & Examples



BRISC-V Simulator

Adaptive and Secure Computing Systems Lab • Boston University



ASCS LABORATORY

ADAPTIVE & SECURE
COMPUTING SYSTEMS

C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }
```

RISC-V Assembly

```

0      addi    zero,zero,0
1      .label:
2      addi    sp,zero,1536
3      call    main
4      addi    zero,zero,0
5      mv      s1,a0
6      addi    zero,zero,0
7      addi    zero,zero,0
8      auipc   ra,0x0
9      jalr    ra,0(ra)
10     addi    zero,zero,0
11
12     .file     "gcd.c"
13     .option   nopic
14     .text
15     .align    2
16     .globl   gcd
17     .type     gcd, @function
18
19 gcd:
20     addi    sp,sp,-48
21     sw      ra,44(sp)
22     sw      s0,40(sp)
23     addi    s0,sp,48
24     sw      a0,-36(s0)
25     sw      a1,-40(s0)
26     lw      a4,-36(s0)
27     lw      a5,-40(s0)
28     bne     a4,a5,.L2
29     lw      a5,-36(s0)
30     sw      a5,-20(s0)
31     j       .L3
32
33 .L2:
34     lw      a4,-36(s0)
35     lw      a5,-40(s0)
36     blt     a4,a5,.L4
37     .L4:
38     .L3:
39     ret
40
41 .L5:
42     .L6:
43     .L7:
44     .L8:
45     .L9:
46     .L10:
47     .L11:
48     .L12:
49     .L13:
50     .L14:
51     .L15:
52     .L16:
53     .L17:
54     .L18:
55     .L19:
56     .L20:
57     .L21:
58     .L22:
59     .L23:
60     .L24:
61     .L25:
62     .L26:
63     .L27:
64     .L28:
65     .L29:
66     .L30:
67     .L31:
68     .L32:
69     .L33:
70     .L34:
71     .L35:
72     .L36:
73     .L37:
74     .L38:
75     .L39:
76     .L40:
77     .L41:
78     .L42:
79     .L43:
80     .L44:
81     .L45:
82     .L46:
83     .L47:
84     .L48:
85     .L49:
86     .L50:
87     .L51:
88     .L52:
89     .L53:
90     .L54:
91     .L55:
92     .L56:
93     .L57:
94     .L58:
95     .L59:
96     .L60:
97     .L61:
98     .L62:
99     .L63:
100    .L64:
101    .L65:
102    .L66:
103    .L67:
104    .L68:
105    .L69:
106    .L70:
107    .L71:
108    .L72:
109    .L73:
110    .L74:
111    .L75:
112    .L76:
113    .L77:
114    .L78:
115    .L79:
116    .L80:
117    .L81:
118    .L82:
119    .L83:
120    .L84:
121    .L85:
122    .L86:
123    .L87:
124    .L88:
125    .L89:
126    .L90:
127    .L91:
128    .L92:
129    .L93:
130    .L94:
131    .L95:
132    .L96:
133    .L97:
134    .L98:
135    .L99:
136    .L100:
137    .L101:
138    .L102:
139    .L103:
140    .L104:
141    .L105:
142    .L106:
143    .L107:
144    .L108:
145    .L109:
146    .L110:
147    .L111:
148    .L112:
149    .L113:
150    .L114:
151    .L115:
152    .L116:
153    .L117:
154    .L118:
155    .L119:
156    .L120:
157    .L121:
158    .L122:
159    .L123:
160    .L124:
161    .L125:
162    .L126:
163    .L127:
164    .L128:
165    .L129:
166    .L130:
167    .L131:
168    .L132:
169    .L133:
170    .L134:
171    .L135:
172    .L136:
173    .L137:
174    .L138:
175    .L139:
176    .L140:
177    .L141:
178    .L142:
179    .L143:
180    .L144:
181    .L145:
182    .L146:
183    .L147:
184    .L148:
185    .L149:
186    .L150:
187    .L151:
188    .L152:
189    .L153:
190    .L154:
191    .L155:
192    .L156:
193    .L157:
194    .L158:
195    .L159:
196    .L160:
197    .L161:
198    .L162:
199    .L163:
200    .L164:
201    .L165:
202    .L166:
203    .L167:
204    .L168:
205    .L169:
206    .L170:
207    .L171:
208    .L172:
209    .L173:
210    .L174:
211    .L175:
212    .L176:
213    .L177:
214    .L178:
215    .L179:
216    .L180:
217    .L181:
218    .L182:
219    .L183:
220    .L184:
221    .L185:
222    .L186:
223    .L187:
224    .L188:
225    .L189:
226    .L190:
227    .L191:
228    .L192:
229    .L193:
230    .L194:
231    .L195:
232    .L196:
233    .L197:
234    .L198:
235    .L199:
236    .L200:
237    .L201:
238    .L202:
239    .L203:
240    .L204:
241    .L205:
242    .L206:
243    .L207:
244    .L208:
245    .L209:
246    .L210:
247    .L211:
248    .L212:
249    .L213:
250    .L214:
251    .L215:
252    .L216:
253    .L217:
254    .L218:
255    .L219:
256    .L220:
257    .L221:
258    .L222:
259    .L223:
260    .L224:
261    .L225:
262    .L226:
263    .L227:
264    .L228:
265    .L229:
266    .L230:
267    .L231:
268    .L232:
269    .L233:
270    .L234:
271    .L235:
272    .L236:
273    .L237:
274    .L238:
275    .L239:
276    .L240:
277    .L241:
278    .L242:
279    .L243:
280    .L244:
281    .L245:
282    .L246:
283    .L247:
284    .L248:
285    .L249:
286    .L250:
287    .L251:
288    .L252:
289    .L253:
290    .L254:
291    .L255:
292    .L256:
293    .L257:
294    .L258:
295    .L259:
296    .L260:
297    .L261:
298    .L262:
299    .L263:
300    .L264:
301    .L265:
302    .L266:
303    .L267:
304    .L268:
305    .L269:
306    .L270:
307    .L271:
308    .L272:
309    .L273:
310    .L274:
311    .L275:
312    .L276:
313    .L277:
314    .L278:
315    .L279:
316    .L280:
317    .L281:
318    .L282:
319    .L283:
320    .L284:
321    .L285:
322    .L286:
323    .L287:
324    .L288:
325    .L289:
326    .L290:
327    .L291:
328    .L292:
329    .L293:
330    .L294:
331    .L295:
332    .L296:
333    .L297:
334    .L298:
335    .L299:
336    .L300:
337    .L301:
338    .L302:
339    .L303:
340    .L304:
341    .L305:
342    .L306:
343    .L307:
344    .L308:
345    .L30
```

Stepping Through a Program

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

```

1 int fib(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fib(n-1)+fib(n-2);
6     }
7 }
8
9 int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0
    .file     "gcd.c"
    .option   nopie
    .text
    .align    2
    .globl    gcd
    .type     gcd, @function
gcd:
11     addi    sp,sp,-48
12     sw      ra,44(sp)
13     sw      s0,40(sp)
14     addi    s0,sp,48
15     sw      a0,-36(s0)
16     sw      a1,-40(s0)
17     lw      a4,-36(s0)
18     lw      a5,-40(s0)
19     bne     a4,a5,.L2
20     lw      a5,-36(s0)
21     sw      a5,-20(s0)
22     j       .L3
.L2:
23     lw      a4,-36(s0)
24     lw      a5,-40(s0)

```

Registers

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 0	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
011000000000		00000		000		00010		0010011	

Console

```

***** Parser Output *****
Parsing successful!

```

Annotations:

- Click the step button again
- These instructions did not do anything, but the next one will
- The blue line moved two lines down
- The instruction breakdown pane shows how the instruction is stored in memory
- The top row represents the bit ranges, the middle row shows range names, the bottom row shows binary values for the specific instruction

addi Changed the Register File

BRISC-V Home
BRISC-V Simulator
Manual & Examples




C source

```

1 int fib(int n) {
2   if (n <= 1) {
3     return n;
4   } else {
5     return fib(n-1)+fib(n-2);
6   }
7 }
8
9 int return_function (int result) {
10  return result;
11 }
12
13 int main(){
14   int n = 9;
15   int result = return_function (fib(n));
16   return result;
17 }

```

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0
11
12     .file    "gcd.c"
13     .option  nopie
14     .text
15     .align  2
16     .globl  gcd
17     .type   gcd, @function
gcd:
18     addi    sp,sp,-48
19     sw      ra,44(sp)
20     sw      s0,40(sp)
21     addi    s0,sp,48
22     sw      a0,-36(s0)
23     sw      a1,-40(s0)
24     lw      a4,-36(s0)
25     lw      a5,-40(s0)
26     lw      a5,-40(s0)
27     bne     a4,a5,.L2
28     lw      a5,-36(s0)
29     sw      a5,-20(s0)
30     j       .L3
.L2:
31     lw      a4,-36(s0)
32     lw      a5,-40(s0)
33     bne     a4,a5,.L2

```

Registers

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 1536	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/sf	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Console

```

***** Parser Output *****
Parsing successful!

```

Instruction breakdown

Pseudo-Instructions don't have a breakdown

addi sp, zero, 1536
got executed

It summed 0 and 1536,
and put it in register **sp**

Register **sp** is
highlighted!

sp stands for stack
pointer

Setting Breakpoints

BRISC-V Home

BRISC-V Simulator

Manual & Examples





C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0

```

Registers

Memory

Register	Value	Register	Value
zero	[0] 0	ra	[1] 0
sp	[2] 1536	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/fp	[8] 0	s1	[9] 0
a0	[10] 0	a1	[11] 0
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 0
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Console

```

***** Parser Output *****
Parsing successful!

```

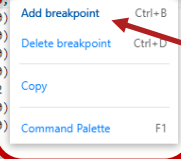
Instruction breakdown

```

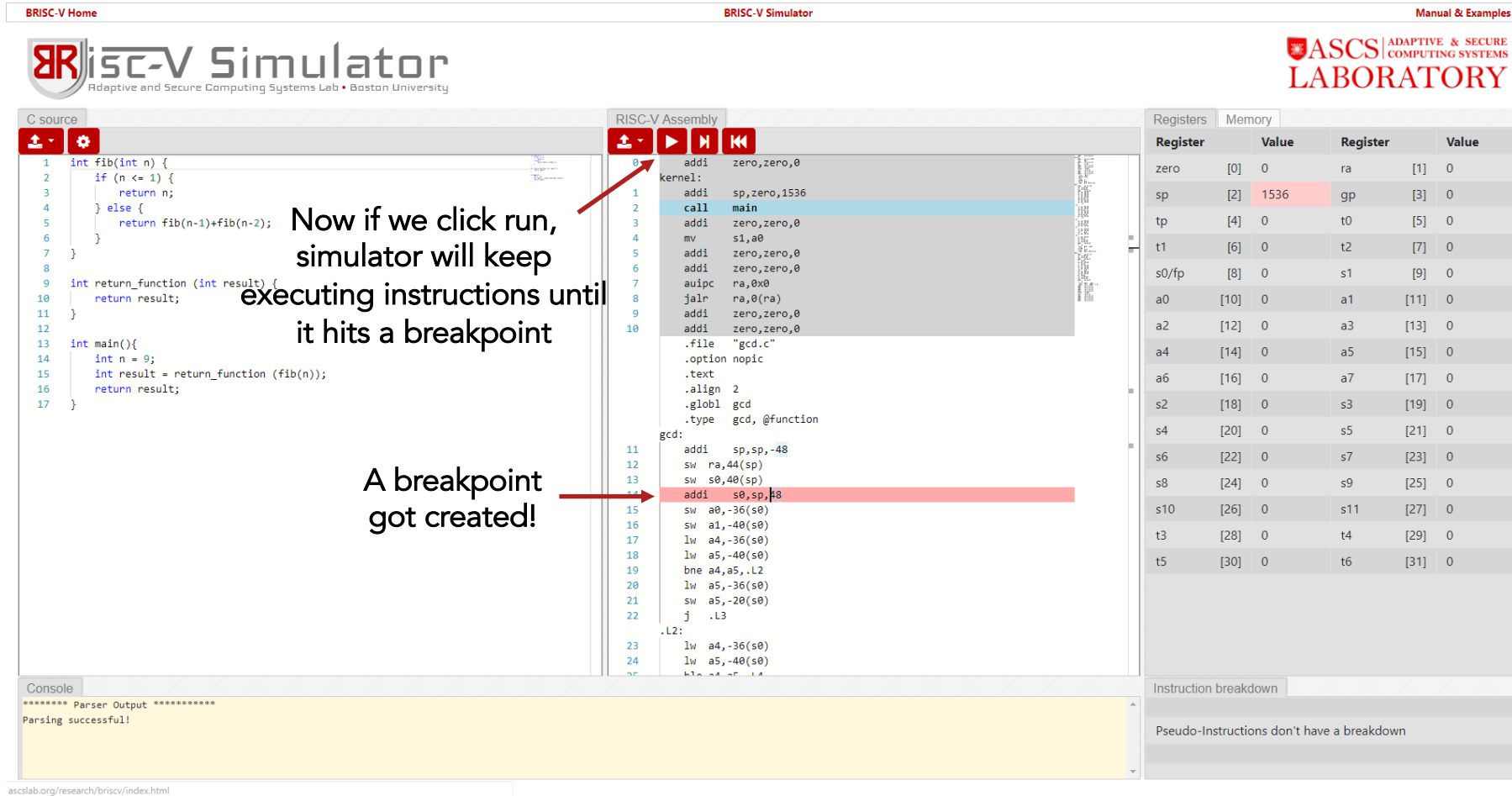
Pseudo-Instructions don't have a breakdown

```

Right-clicking on an instruction opens a menu



Let's click on the first item – Add breakpoint



Running Code until a Breakpoint

BRISC-V Home

BRISC-V Simulator

Manual & Examples





C source

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

RISC-V Assembly

```

0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0
11
12     .file    "gcd.c"
13     .option  nopie
14     .text
15     .align   2
16     .globl   gcd
17     .type    gcd, @function
18
19 gcd:
20     addi     sp,sp,-48
21     sw       ra,44(sp)
22     sw       s0,40(sp)
23     addi     s0,sp,48
24     sw       a0,-36(s0)
25     sw       a1,-40(s0)
26     lw       a4,-36(s0)
27     lw       a5,-40(s0)
28     bne      a4,a5,.L2
29     lw       a5,-36(s0)
30     sw       a5,-20(s0)
31     j        .L3
32
33 .L2:
34     lw       a4,-36(s0)
35     lw       a5,-40(s0)
36     bne      a4,a5,.L2

```

The instruction pointer moved to the breakpoint!

A lot of registers changed values!

Register	Value	Register	Value
zero	[0] 0	ra	[1] 73
sp	[2] 1456	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
s0/tp	[8] 1536	s1	[9] 0
a0	[10] 64	a1	[11] 48
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 48
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Console

```

***** Parser Output *****
Parsing successful!

```

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
000000110000		00010		000		01000		0010011	

Text and Data Sections

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

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8  .data and .rodata sections allow
9  us to set some memory before
10 the program is run
11
12 int result = fib(10);
13
14 int main(){
15     int n = 9;
16     int result = return_function(fib(n));
17     return result;
18 }

```

RISC-V Assembly

```

22  ecall
    # let's statically allocate a string "HELLO!"
    # we start this by creating a read-only data section
    .rodata
    .HELLO:
    # strings should end with the null terminator \0
    # the null terminator's binary value is 0!
    # we split HELLO!\0 into two 32bit words:
    # HELL and 0!00 - note that thats an "0!" and 2 zeros
    # we write HELL in ascii:
    # H - 0x48
    # E - 0x45
    # L - 0x4C
    # since this is a little-endian architecture, we
    # write HELL in reverse - LEHH
    .word 0x4C4C4548
    # now we write the second part 0!00
    .word 0x0000214F
    # this section is parsed when you load the program -
    # not when the instruction pointer runs over it.
    # as soon as you loaded the program, you should see
    # this string in the memory pane's data section,
    # somewhere close to the bottom of it.
    #
    # now we can go back to a text section that has code
    .text
    # print the string "HELLO!\n"
    addi t0, zero, 3      # this is the string printing syscall
    lui a0, %hi(.HELLO)   # this loads the top 20 bits
                          # of .HELLO address into a0
    addi a0, a0, %lo(.HELLO) # this loads the bottom 12 bits
    addi a1, zero, 7      # length of the string
    ecall
    # print characters '!', '\n', '-'
    addi t0, zero, 0
    ecall

```

Registers

HEX	DEC	BINARY
0x00000100	00 00 00 00	
0x00000134	00 00 00 00	
0x00000130	00 00 00 00	
0x0000012c	00 00 00 00	
0x00000128	00 00 00 00	
0x00000124	00 00 00 00	
0x00000120	00 00 00 00	
0x0000011c	00 00 00 00	
0x00000118	00 00 00 00	
0x00000114	00 00 00 00	
0x00000110	00 00 00 00	
0x0000010c	00 00 00 00	
HEAP SEGMENT		
DATA SEGMENT		
0x00000108	00 00 21 4f	
0x00000104	4c 4c 45 48	// <- .hello
TEXT SEGMENT		
0x00000100	00 00 00 13	// addi zero, zero
0x000000fc	00 00 00 13	// addi zero, zero
0x000000f8	00 00 00 13	// addi zero, zero
0x000000f4	00 00 00 13	// addi zero, zero
0x000000f0	00 00 80 e7	// jalr ra, 0(ra)
0x000000ec	00 00 00 97	// auipc ra, 0x0
0x000000e8	00 00 00 13	// addi zero, zero
0x000000e4	00 00 00 13	// addi zero, zero
0x000000e0	00 00 00 13	// addi zero, zero
0x000000dc	00 00 00 13	// addi zero, zero
0x000000d8	00 00 80 67	// jr ra
0x000000d4	00 00 00 73	// ecall
0x000000d0	71 00 05 13	// addi a0, zero,
0x000000cc	00 70 02 93	// addi t0, zero,
0x000000c8	00 00 00 73	// ecall
0x000000c4	ff 00 05 13	// addi a0, zero,
0x000000c0	00 70 02 93	// addi t0, zero,
0x000000bc	00 00 00 73	// ecall

Console

```

***** Parser Output *****
Parsing successful!

```

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
0000000000000	00000	000		00000		0010011			

.data and .rodata sections allow us to set some memory before the program is run

.text sections are used for code. Programs are in the text section by default

Text and Data Sections

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

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function(int n) {
10     return fib(n);
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

Labels in data sections point to memory statically allocated after them

Here we allocate the string "HELLO!" by into two 32bit words – one with "HELL" and one with "O!00"

RISC-V Assembly

```

22  ecall
    # let's statically allocate a string "HELLO!"
    # we start this by creating a read-only data section
    .rodata
    .HELLO:
    # strings should end with the null terminator \0
    # the null terminator's binary value is 0!
    # we split HELLO!\0 into two 32bit words:
    # HELL and O!00 - note that thats an "O!" and 2 zeros
    # we write HELL in ascii:
    # H - 0x48
    # E - 0x45
    # L - 0x4C
    # since this is a little-endian architecture, we
    # write HELL in reverse - LEHH
    .word 0x4C4C4548
    # now we write the second part O!00
    .word 0x0000214F
    # this section is parsed when you load the program -
    # not when the instruction pointer runs over it.
    # as soon as you loaded the program, you should see
    # this string in the memory pane's data section,
    # somewhere close to the bottom of it.
    #
    # now we can go back to a text section that has code
    .text
    # print the string "HELLO!\n"
    addi t0, zero, 3      # this is the string printing syscall
    lui a0, %hi(.HELLO)   # this loads the top 20 bits
                          # of .HELLO address into a0
    addi a0, a0, %lo(.HELLO) # this loads the bottom 12 bits
    addi a1, zero, 7      # length of the string
    ecall
    # print characters '!', '\n', '-'
    addi t0, zero, 0

```

Registers **Memory**

HEX DEC BINARY

0x00000100	00 00 00 00	
0x00000134	00 00 00 00	
0x00000130	00 00 00 00	
0x0000012c	00 00 00 00	
0x00000128	00 00 00 00	
0x00000124	00 00 00 00	
0x00000120	00 00 00 00	
0x0000011c	00 00 00 00	
0x00000118	00 00 00 00	
0x00000114	00 00 00 00	
0x00000110	00 00 00 00	
0x0000010c	00 00 00 00	
HEAP SEGMENT		
DATA SEGMENT		
0x00000108	00 00 21 4f	
0x00000104	4c 4c 45 48	// <- .hello
TEXT SEGMENT		
0x00000100	00 00 00 13	// addi zero, zero
0x000000fc	00 00 00 13	// addi zero, zero
0x000000f8	00 00 00 13	// addi zero, zero
0x000000f4	00 00 00 13	// addi zero, zero
0x000000f0	00 00 80 e7	// jalr ra, 0(ra)
0x000000ec	00 00 00 97	// auipc ra, 0x0
0x000000e8	00 00 00 13	// addi zero, zero
0x000000e4	00 00 00 13	// addi zero, zero
0x000000e0	00 00 00 13	// addi zero, zero
0x000000dc	00 00 00 13	// addi zero, zero
0x000000d8	00 00 80 67	// jr ra
0x000000d4	00 00 00 73	// ecall
0x000000d0	71 00 05 13	// addi a0, zero,
0x000000cc	00 70 02 93	// addi t0, zero,
0x000000c8	00 00 00 73	// ecall
0x000000c4	ff 00 05 13	// addi a0, zero,
0x000000c0	00 70 02 93	// addi t0, zero,
0x000000bc	00 00 00 73	// ecall

Console

```

***** Parser Output *****
Parsing successful!

```

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
0000000000000	00000	000		00000		0010011			

Text and Data Sections

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

```

1  int fib(int n) {
2      if (n <= 1) {
3          return n;
4      } else {
5          return fib(n-1)+fib(n-2);
6      }
7  }
8
9  int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }

```

RISC-V Assembly

```

22  ecall
    # let's statically allocate a string "HELLO!"
    # we start this by creating a read-only data section
    .rodata
    .HELLO:
    # strings should end with the null terminator \0
    # the null terminator's binary value is 0!
    # we split HELLO!\0 into two 32bit words:
    # HELL and 0!00 - note that thats an "0!" and 2 zeros
    # we write HELL in ascii:
    # H - 0x48
    # E - 0x45
    # L - 0x4C
    # since this is a little-endian architecture, we
    # write HELL in reverse - LEHH
    .word 0x4C4C4548
    # now we write the second part 0!00
    .word 0x0000214F
    # this section is parsed when you load the program -
    # not when the instruction pointer runs over it.
    # as soon as you loaded the program, you should see
    # this string in the memory pane's data section,
    # somewhere close to the bottom of it.
    #
    # now we can go back to a text section that has code
    .text
    # print the string "HELLO!\n"
    addi t0, zero, 3      # this is the string printing syscall
    lui a0, %hi(.HELLO)   # this loads the top 20 bits
                          # of .HELLO address into a0
    addi a0, a0, %lo(.HELLO) # this loads the bottom 12 bits
    addi a1, zero, 7      # length of the string
    ecall
    # print characters '!', '\n', '-'

```

Registers **Memory**

HEX DEC BINARY

0x00000100:	00 00 00 00	
0x00000134:	00 00 00 00	
0x00000130:	00 00 00 00	
0x0000012c:	00 00 00 00	
0x00000128:	00 00 00 00	
0x00000124:	00 00 00 00	
0x00000120:	00 00 00 00	
0x0000011c:	00 00 00 00	
0x00000118:	00 00 00 00	
0x00000114:	00 00 00 00	
0x00000110:	00 00 00 00	
0x0000010c:	00 00 00 00	
HEAP SEGMENT		
DATA SEGMENT		
0x00000108:	00 00 21 4f	
0x00000104:	4c 4c 45 48	// <- .hello
TEXT SEGMENT		
0x00000100:	00 00 00 13	// addi zero, zero
0x000000fc:	00 00 00 13	// addi zero, zero
0x000000f8:	00 00 00 13	// addi zero, zero
0x000000f4:	00 00 00 13	// addi zero, zero
0x000000f0:	00 00 80 e7	// jalr ra, 0(ra)
0x000000ec:	00 00 00 97	// auipc ra, 0x0
0x000000e8:	00 00 00 13	// addi zero, zero
0x000000e4:	00 00 00 13	// addi zero, zero
0x000000e0:	00 00 00 13	// addi zero, zero
0x000000dc:	00 00 00 13	// addi zero, zero
0x000000d8:	00 00 80 67	// jr ra
0x000000d4:	00 00 00 73	// ecall
0x000000d0:	71 00 05 13	// addi a0, zero,
0x000000cc:	00 70 02 93	// addi t0, zero,
0x000000c8:	00 00 00 73	// ecall
0x000000c4:	ff 00 05 13	// addi a0, zero,
0x000000c0:	00 70 02 93	// addi t0, zero,
0x000000bc:	00 00 00 73	// ecall

You can see the allocated memory in the data segment in the memory pane! You can also see the .HELLO pointer!

Console

```

***** Parser Output *****
Parsing successful!

```

Instruction breakdown

31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
0000000000000	00000	000		00000		0010011			

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!

The screenshot displays the BRISC-V Simulator interface. On the left, the 'C source' tab shows a Fibonacci function and a main function. The 'RISC-V Assembly' tab shows the corresponding assembly code, with the instruction 'addi s0,sp,48' highlighted. On the right, the 'Registers' table shows the state of various registers, with 'ra' (73) and 's0' (1536) highlighted. The 'Console' tab at the bottom shows the output: '***** Parser Output *****' and 'Parsing successful!'. Overlaid on the center of the image is the text: 'That's All Folks! Now open a text editor, write some assembly, and try to run it!'.

Register	Value	Register	Value
zero	[0] 0	ra	[1] 73
sp	[2] 1456	gp	[3] 0
tp	[4] 0	t0	[5] 0
t1	[6] 0	t2	[7] 0
fp	[8] 1536	s1	[9] 0
a0	[10] 4	a1	[11] 48
a2	[12] 0	a3	[13] 0
a4	[14] 0	a5	[15] 48
a6	[16] 0	a7	[17] 0
s2	[18] 0	s3	[19] 0
s4	[20] 0	s5	[21] 0
s6	[22] 0	s7	[23] 0
s8	[24] 0	s9	[25] 0
s10	[26] 0	s11	[27] 0
t3	[28] 0	t4	[29] 0
t5	[30] 0	t6	[31] 0

Bugs and Features

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