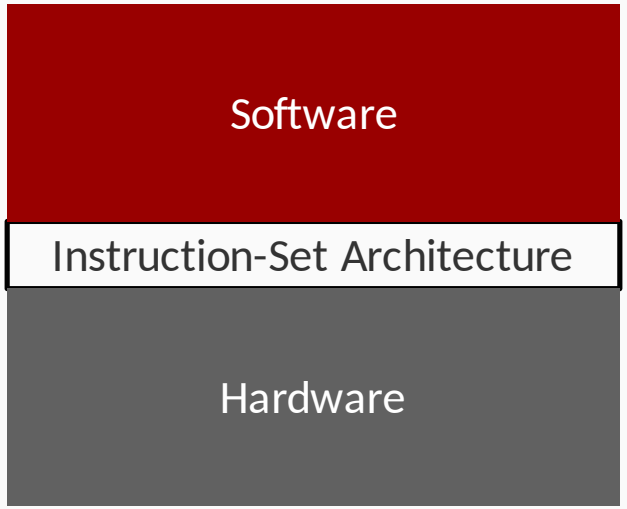
RISC-V Instruction Set Architecture and Simulators

October 15, 2019

Institute of Embedded Systems (E-13)
Hamburg University of Technology

Instruction Set Architecture



- The Interface between Software and Hardware
- The Programmers view of the processor
- Hides the details of processor implementation
- Enables a wide variety of microarchitecture implementations without compiler customization

What is RISC-V?

- Fifth generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification (here)
- > 25 RISC-V CPU Designs
- Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from microcontrollers to supercomputers
- Standard maintained by non-profit RISC-V Foundation (Foundation Members (60+))

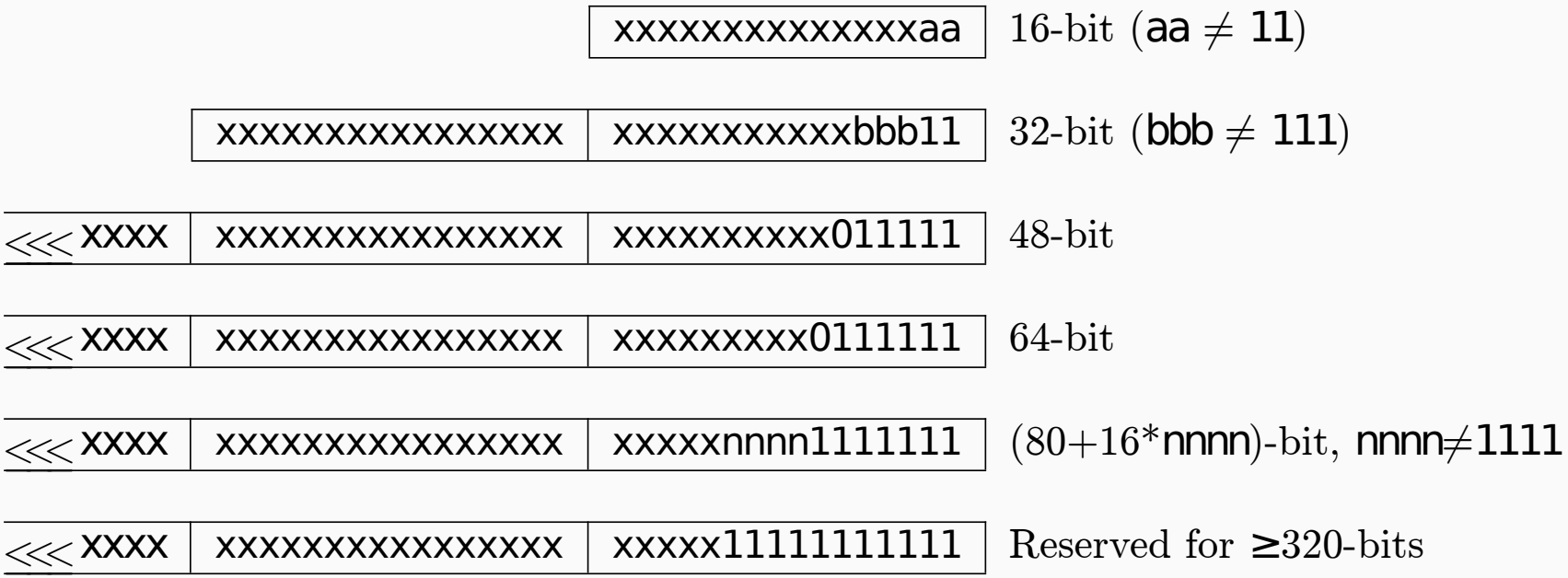
Note: The RISC-V ISA manual include useful information about the made design decisions

RISC-V instruction length encoding

32-Bit RISC-V: RV32I (Base Integer Instructions)

64-Bit RISC-V: RV64I (Base Integer Instructions)

128-Bit RISC-V: RV128I (Base Integer Instructions)



RISC-V ISA Extensions

M: Integer Multiply and Division *

F: Single-Precision Floating-Point (32-bit) *

D: Double-Precision Floating-Point (64-bit) *

Q: Quad-Precision Floating-Point (128-bit) *

L: Decimal-Precision Floating-Point

C: Compressed Instructions (16-bit encoding)

expands to 32-bit instruction (2-address format + 32 Regs | 3-address format + 8 Regs), smallest Instruction Format on SPECint2006

B: Bit Manipulation Instructions

J: Dynamically Translated Languages

T: Transactional Memory

P: Packed SIMD Instructions *

V: Vector Operations *

N: User-Level Interrupts *

G: General-Purpose (I+M+A+F+D)

TinyRV ISA

The TinyRV ISA is:

- A subset of the 32-bit RISC-V ISA suitable for teaching
- More specifically, the TinyRV ISA is a subset of the RV32IM ISA
- suitable for running simple C programs
- suitable for a FPGA implementation
- From a assembler programmer's perspective, the RV32IM ISA is very similar to the MIPS32 ISA: Many assembler programs are executable after register renaming.

Data Formats

- Tiny RISC-V supports only
  - Words / int: 32-bit
  - Half-words / short int: 16-bit
  - Bytes / char: 8-bit
  - No floating-point
- General Purpose Registers:
  - 31 general-purpose registers x1-x31, each register is 32 bits wide
  - Register x0 is hardwired to the constant zero
  - Registers have no type (Operation determines how register contents are interpreted)
  - TinyRV uses the same calling convention and symbolic register names as RISC-V

TinyRV Registers and its usage by the compiler

Mnemonic	Reg Nr.	Description	Saver
zero	x0	constant value 0	
ra	x1	return address	Caller
sp	x2	stack pointer	Callee
gp	x3	global pointer	
tp	x4	thread pointer	
t0 - t2	x5 - x7	temporary registers	Caller
s0 / fp	x8	saved register or frame pointer	Callee
s1	x9	saved register	Callee
a0 - a1	x10 - x11	function arguments / return values	Caller
a2 - a7	x12 - x17	function arguments	Caller
s2 - s11	x18 - x27	saved registers	Callee
t3 - t6	x28 - x31	temporary registers	Caller

TinyRV Memory

- TinyRV supports only a 1MB virtual memory address space from `0x00000000` to `0x000fffff`
- Memory addresses are in bytes, not words, so word addresses are 4 bytes apart (PC = PC + 4)
- The memory system is of little endianess type:
  Endianness specifies the order of bytes in the target register when we load a word from memory:
  - Assume:

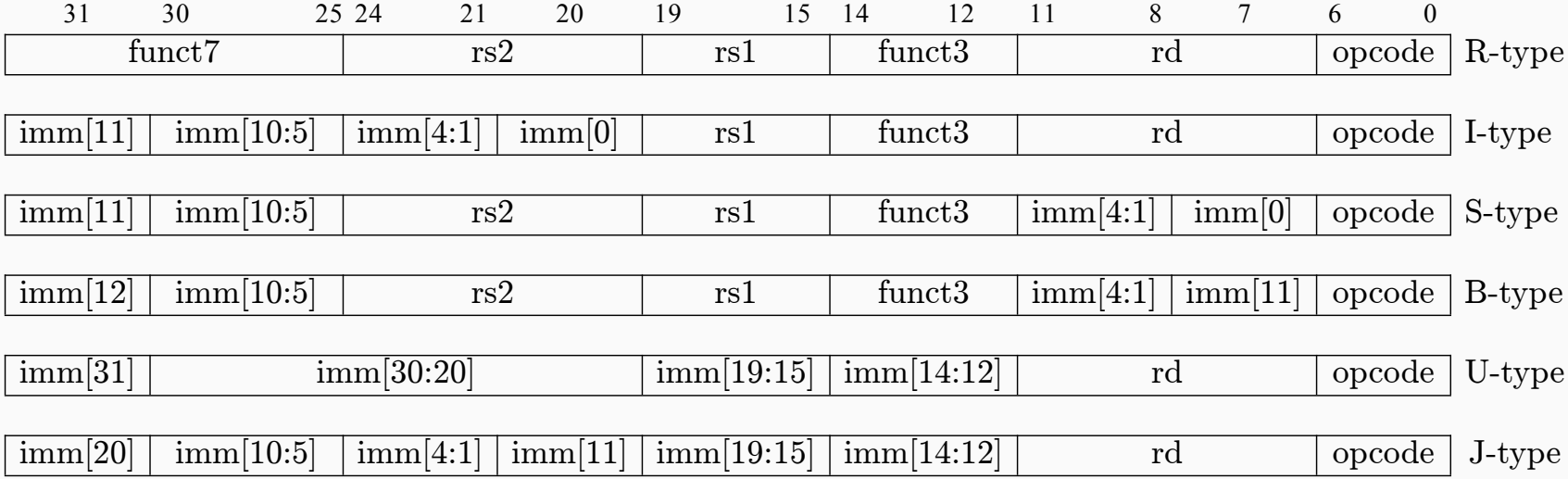
Data	A	B	C	D
Address	0x0	0x1	0x2	0x3

  - If we load a four-byte word from address 0x0, there are two options for the destination register:
    - 0xABCD (big endian)
    - 0xDCBA (little endian).

  There is no significant benefit of one system over the other

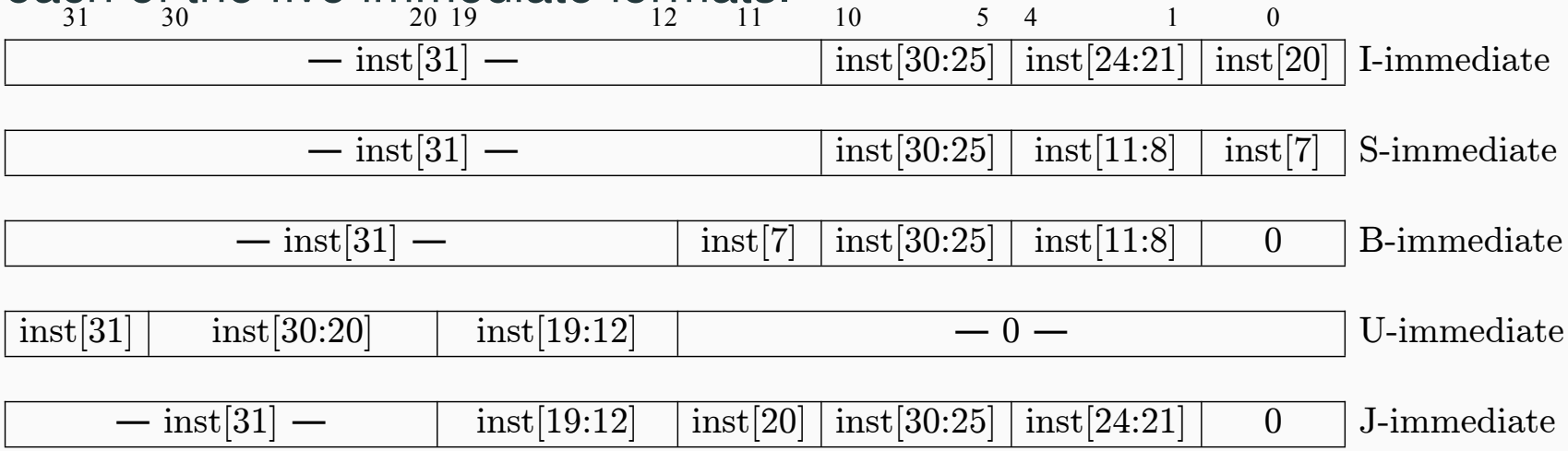
TinyRV Instruction Encoding

- The TinyRV ISA uses the same instruction encoding as RISC-V
- Each instruction has a specific instruction type



TinyRV Immediate Formats

- Note that in RISC-V all immediates are always sign extended
- Sign-bit for the immediate is always in bit 31 of the instruction.
- The following diagrams illustrate how to create a 32-bit immediate from each of the five immediate formats.



TinyRV Integer Register-Register Operations

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
add	ADD	R	0x3	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R		0x0	0x20	rd = rs1 - rs2	
xor	XOR	R		0x4	0x00	rd = rs1 ^ rs2	
or	OR	R		0x6	0x00	rd = rs1 | rs2	
and	AND	R		0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R		0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R		0x2	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith	R		0x3	0x20	rd = rs1 >> rs2	msb-ext.
slt	Set Less Than	R	0x33	0x2		rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R		0x3		rd = (rs1 < rs2)?1:0	zero-ext.
mul	multiply	R		0x0	0x01	rd = (rs1 * rs2)[31:0]	

TinyRV Integer Register-Immediate Operations

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
addi	ADD Immediate	I	0x13	0x0	0x00	rd = rs1 + imm	
xori	XOR Immediate	I		0x4	0x00	rd = rs1 ^ imm	
ori	OR Immediate	I		0x6	0x00	rd = rs1 | imm	
andi	AND Immediate	I		0x7	0x00	rd = rs1 & imm	
slli	Shift Left Logical Imm	I		0x1	0x00	rd = rs1 << imm	
srli	Shift Right Logical Imm	I		0x1	0x00	rd = rs1 >> imm	
srai	Shift Right Arith Imm	I		0x3	0x20	rd = rs1 >> imm	msb-ext.
slti	Set Less Than Imm	I		0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I		0x3		rd = (rs1 < imm)?1:0	zero-ext.
lui	Load Upper Imm	U	0x37			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0x17			rd = PC + (imm << 12)	

TinyRV Load and Store Operations

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
lb	Load Byte	I	0x3	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I		0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I		0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I		0x4		rd = M[rs1+imm][0:7]	zero-ext.
lhu	Load Half (U)	I		0x5		rd = M[rs1+imm][0:15]	zero-ext.
sb	Store Byte	S	0x23	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S		0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S		0x2		M[rs1+imm][0:31] = rs2[0:31]	
csrr	Atomic Read CSR	I	0x73	0x2		rd = CSR[csr]	csr=imm
csrw	Atomic Write CSR			0x1		CSR[csr] = rs1	csr=imm

Control Transfer Operations

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
beq	Branch ==	B	0x63	0x0		if (rs1 == rs2) PC += imm	
bne	Branch !=	B		0x1		if (rs1 != rs2) PC += imm	
blt	Branch <	B		0x4		if (rs1 < rs2) PC += imm	
bge	Branch <=	B		0x5		if (rs1 >= rs2) PC += imm	
bltu	Branch < (Unsigned)	B		0x6		if (rs1 < rs2) PC += imm	
bgeu	Branch >= (Unsigned)	B		0x7		if (rs1 >= rs2) PC += imm	
jal	Jump And Link	J	0xCF			rd = PC+4; PC += imm	zero-ext.
jalr	Jump And Link Reg	I	0x37	0x0		rd = PC+4; PC = rs1	zero-ext.
ecall	Environment Call	I	0x73	0x0	0x00	Transfer control to OS a0 = 1: print value of a1	imm: 0x000
ebreak	Environment Break	I	0x73	0x0	0x00	Transfer control to debug	imm: 0x001

TinyRV Privileged ISA?

- No distinction between user and privileged mode
  - (Using the terminology in the RISC-V vol 2 ISA manual, TinyRV only supports M-mode)
- Reset Vector
  - RISC-V specifies two potential reset vectors: one at a low address, and one at a high address.
  - TinyRV uses the low address reset vector at 0x00000200.
    - This is where assembly tests should reside as well as user code in TinyRV.

Control/Status Registers (CSRs)

TinyRV includes five non-standard CSRs:

mngr2proc: 0xFC0

Used to communicate data from the manager to the processor.
This register has register-mapped FIFO-dequeue semantics meaning reading the register essentially dequeues the data from the head of a FIFO.
Reading the register will stall if the FIFO has no valid data.
Writing the register is undefined.

proc2mngr: 0x7c0

Used to communicate data from the processor to the manager.
This register has register-mapped FIFO-enqueue semantics meaning writing the register essentially enqueues the data on the tail of a FIFO.
Writing the register will stall if the FIFO is not ready.
Reading the register is undefined.

Control/Status Registers (CSRs)

stats_en: 0x7c1
Used to enable or disable the statistics tracking feature of the processor (i.e., counting cycles and instructions)

numcores: 0xfc1
Used to store the number of cores present in a multi-core system
Writing the register is undefined.

coreid: 0xf14
Used to communicate the core id in a multi-core system
Writing the register is undefined

Address Translation

- TinyRV only supports the most basic form of address translation:
  - logical address = physical address
- In the RISC-V vol 2 ISA manual this is called a Mbare addressing environment.

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address
l{b|h|w|d} rd, symbol	auipc rd, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rd)	Load global
s{b|h|w|d} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w|d} rd, symbol[11:0](rt)	Store global
nop	addi x0, x0, 0	No operation
li rd, immediate	Myriad sequences	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if != zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero

Code examples: Transfers between Memory and Register

- Word data transfers

```
lw x10,12(x13)  -- load  word
sw x10,40(x13)  -- store word
```

  - `x13` – base register;
  - `12` – offset in bytes; Offset must be a constant known at assembly time

Code examples: Long immediates

How to set `0xDEADBEEF`?

```
lui  x10, 0xDEADB      # x10 = 0xDEADB000
addi x10, x10, 0xEEF   # x10 = 0xDEADAEEF
```

Immediates are always sign-extended, if top bit is set, will subtract -1 from upper 20 bits

Solution:

```
lui  x10, 0xDEADC      # x10 = 0xDEADC000
addi x10, x10, 0xEEF   # x10 = 0xDEADBEEF
```

Assembler pseudo-op handles all of this:

```
li  x10, 0xDEADBEEF    # Creates two instructions
```

Code examples: Translation of `if` statement

```
f: x10, g: x11, h: x12, i: x13, j: x14

if ( i == j)                          bne x13,x14,Exit
  f = g + h;                          add x10,x11,x12
                                Exit:

if ( i == j )                         bne x13,x14,Exit
  f = g + h;                          add x10,x11,x12
else                                  j Exit
  f = g – h;                  Else:   sub x10,x11,x12
                                Exit:
```

Code examples: Translation of `switch` statement

```
switch (amount) {                                          # x2:amount, x3:fee
  case  20:          case20:  addi x1, x0, 20              # x1 = 20
      fee = 2;                bne  x2, x1, case50        # amount == 20? if not
                                                         # skip to case50
                              addi x3, x0, 2             # if so, fee = 2
      break;                  j done                     # and break out of case

  case  50:          case50:  addi x1, x0, 50              # x1 = 50
      fee = 3;                bne x2, x1, case100        # amount == 50? if not,
                                                         # skip to case100
                              addi x3, x0, 3             # if so, fee = 3
      break;                  j done                     # and break out of case

  case 100:          case100: addi x1, x0, 100             # x1 = 100
      fee = 5;                bne x2, x1, default        # amount == 100? if not,
                                                         # skip to default
                              addi x3, x0, 5             # if so, fee = 5
      break;                  j done                     # and break out of case

  default:           default: add x3,x0,x0                # fee = 0
      fee = 0;                done:
}
```

Code examples: Translation of a `while` loop

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

```
        addi x1, x0, 1    # x1 = pow,  pow = 1
        addi x2, x0, 0    # x = 0, x2 = x,
        addi x3, x0, 128  # x3 = 128 for comparison
while:  beq  x1, x3, done # if pow == 128, goto done
        slli x1, x1, 1    # pow = pow * 2 by shiftlef
        addi x2, x2, 1    # x = x + 1
        j while
done:   mv a1, x2         # exit while loop
        li a0, 1
        ecall             # print x2
```

Code examples: Translation of a `for` loop

```c
int A[5] = {20,30,10,40,50};

int sum = 0;
for (int i=0;  i<5;  i++)
  sum +=  A[i];
```

```
        .data                   # data segment

len:    .word 5                 # Length of A[]
A:      .word 20, 30, 10, 40, 50
        .text
        la   x9, A              # x9   = &A[0]
        add  x10, x0, x0        # sum  = 0
        add  x11, x0, x0        # i    = 0
Loop:
        lw   x12, 0(x9)         # x12  = A[i]
        add  x10,x10,x12        # sum +=
        addi  x9, x9,4          # &A[i++]
        addi x11,x11,1          # i++
        la x13, len             # x13  = 5
        lw x13 0(x13)
        blt  x11,x13,Loop
```

Code examples: Terminal Output

```
.data
msg:    .asciiz "\nhello world"
pi:     .float 3.1415927

.text
start: li a0, 1          # ecall code: print integer
       li a1, 0xa        # integer to print
       ecall
       li a0, 4          # ecall code: print string
       la a1, msg        # null-terminated string address
       ecall
       li a0, 11         # ecall code: print character
       li a1, 'a'        # character to print
       ecall
       li a0, 10         # ecall code: stops running
       ecall
```

- RISC-V doesn't have a concept of a screen, but you can use the ecall instruction, a system call, to output characters to the console (which might be a serial port or a pseudo tty).
- https://riscvsim.com/ecalls/

Code examples: Translation of array accesses

```
            .data                       # data segment

len:        .word 5                     # Length of arrays
val:        .word 20, 30, 10, 40, 50

.text
            la   x5, val                # x5 = &val[0]
            lw   x4, 0(x5)              # x4 = val[0]
            slli x4, x4, 3              # x4 = x4 << 3 = x4 * 8
            sw   x4, 0(x5)              # val[0] = x4
            lw   x4, 4(x5)              # x4 = val[1]
            slli x4, x4, 3              # x4 = x4 << 3 = x4 * 8
            sw   x4, 4(x5)              # val[1] = x4
```

Function call

Seven Fundamental Steps in Calling a Function

1. Put parameters in a place where the function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it
6. Restore any registers you used
7. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call

- Registers faster than memory, so use them
- a0 – a7 (x10 - x17) : eight argument registers to pass parameters and two return values (a0 - a1)
- ra : one return address register to return to the point of origin (x1)

```
1000         mv a0,s0                  #  x = a
1004         mv a1,s1                  #  y = b
1008         addi ra,zero,1016         # ra =1016
1012         j sum # jump to sum
1016         ...                       # next instruction ...
2000  sum :  add a 0,a0,a1
2004         jr ra                     #  jump register instruction
```

- Assembler shorthand: `ret = jr ra`
- `sum` might be called by many places, so we can't return to a fixed place

```
1008         jal sum                   #  ra =1012,goto sum
```

(really should be `laj` "link and jump")

Stack: Memory for register values before function call

Where are old register values saved to restore them after function call?

- Need a place to save old values before call function, restore them when return, and delete
- Ideal is a stack, a last-in-first-out (LIFO) queue with its two operations:
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack is located in data memory, so need register to point to it
  - `sp` is the stack pointer in RISC-V
- Convention is grow stack down from high to low addresses
  - Push decrements `sp`
  - Pop increments `sp`

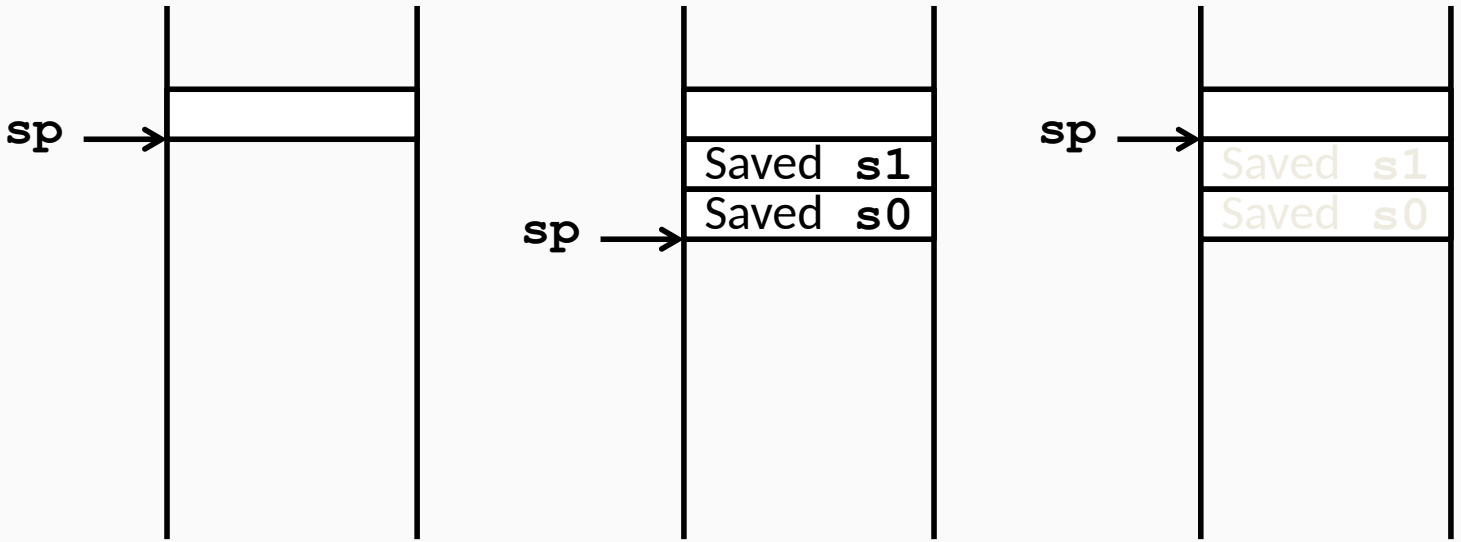
Function example

```
int Leaf (int g, int h, int i, int j) {
  int f;
  f = (g+h)−(i+j);
  return f;
}
```

- Parameter variables `g`, `h`, `i`, and `j` in argument registers `a0`, `a1`, `a2`, and `a3`, and `f` in `s0`
- Assume need one temporary register `s1`

-
```
Leaf:  addi sp, sp, −8     # adjust stack for 2 items
       sw   s1, 4 ( sp )   # save s1 for use afterwards
       sw   s0, 0 ( sp )   # save s0 for use afterwards
       add  s0, a0, a1     # s0 = g + h
       add  s1, a2, a3     # s1 = i + j
       sub  s0, s0, s1     # return value (g + h) − (i + j)
       lw   s0, 0 ( sp )   # restore register s0 for caller
       lw   s1, 4 ( sp )   # restore register s1 for caller
       addi sp,sp, 8       # adjust stack to delete 2 items
       jr ra               # jump back to calling routine
```

Stack before, during, after Function



Before call During call After call

Nested Procedures: Need to save the return address of the caller

```
int sumSquare(int x, int y) {
  return  mult (x,x) + y ;
}
```

- Something called `sumSquare` , now `sumSquare` is calling `mult`
- So there's a value in ra that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`
- Need to save `sumSquare` return address before call to `mult`
- In general, may need to save some other info in addition to `ra`.

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function - calling convention divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - `sp, gp, tp` , "saved registers" `s0-s11` (`s0` is also `fp`)
2. Not preserved across function call
   - Caller cannot rely on values being unchanged
   - Argument/return registers `a0-a7, ra` , "temporary registers" `t0-t6,` callee can use temporary registers without saving and restoring them

Allocating Space on Stack

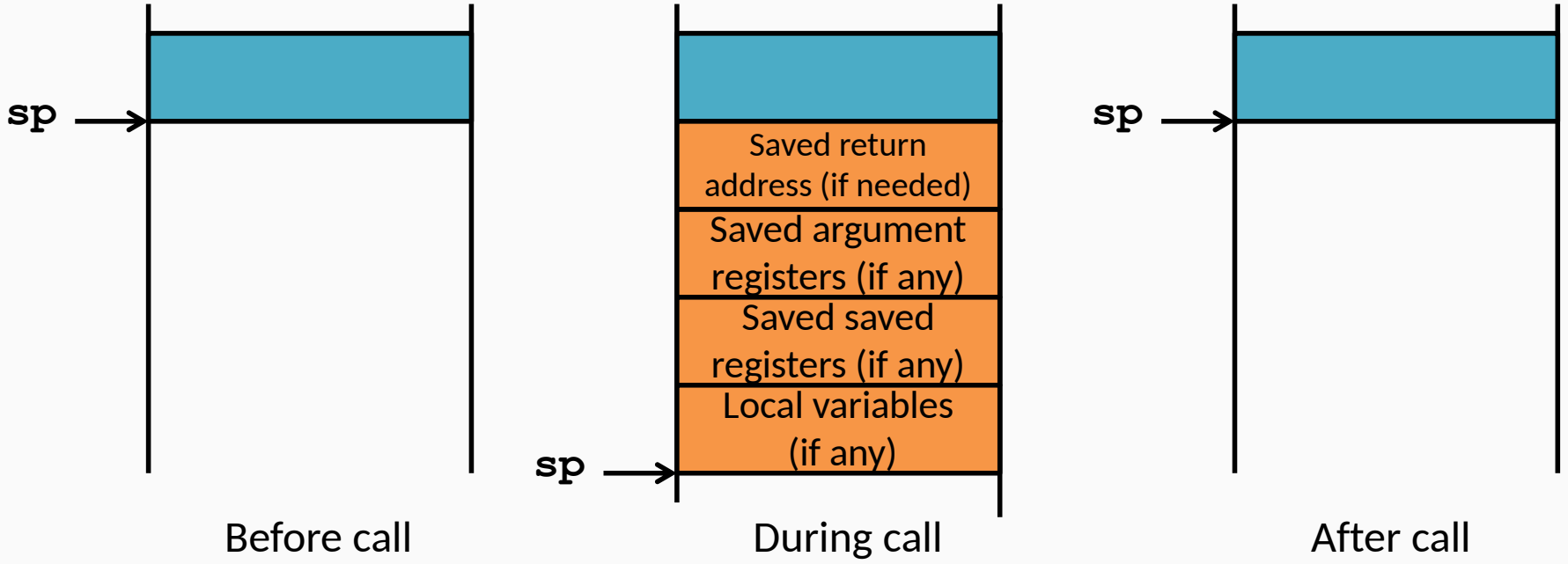
C has two storage classes:

Automatic variables are local to function and discarded when function exits

Static variables exist across exits from and entries to procedures

- Use stack for automatic (local) variables that don't fit in registers

Procedure frame / activation record : segment of stack with saved registers and local variables

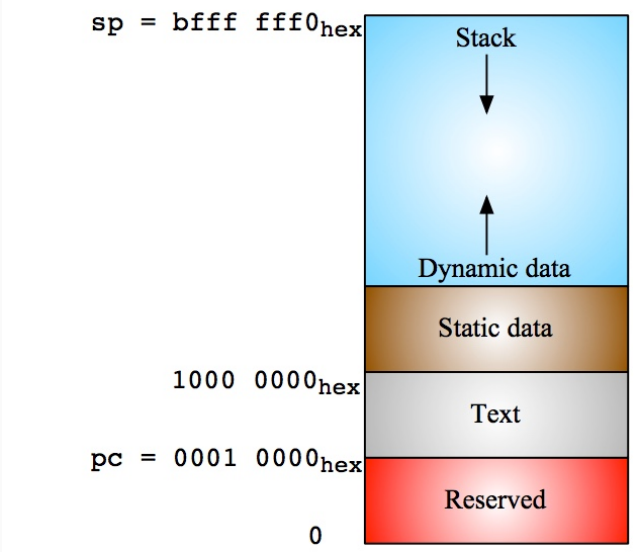


Memory Layout

When a C program is run, there are three important memory areas allocated:

Static : Variables declared once per program, e.g. C globals
Heap : Variables declared dynamically via `malloc`
Stack : Space to be used by procedure during execution; this is where we can save register values

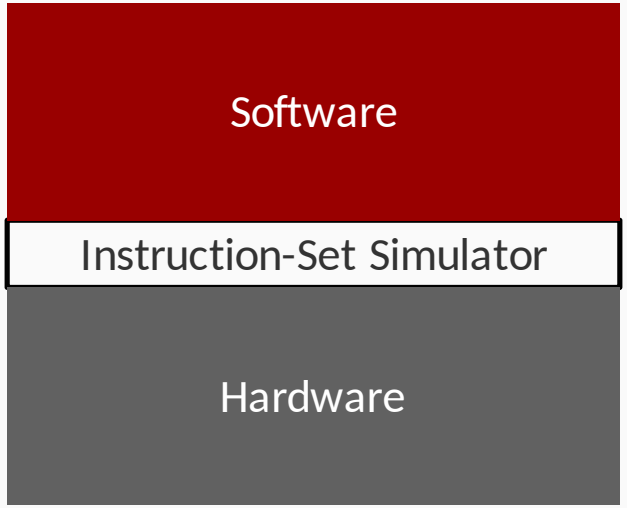
- Stack must be aligned on 16-byte boundary (not true in examples above)
- Global pointer (gp) points to static, RV32 `gp = 1000_0000` hex
- Heap above static for data structures that grow and shrink



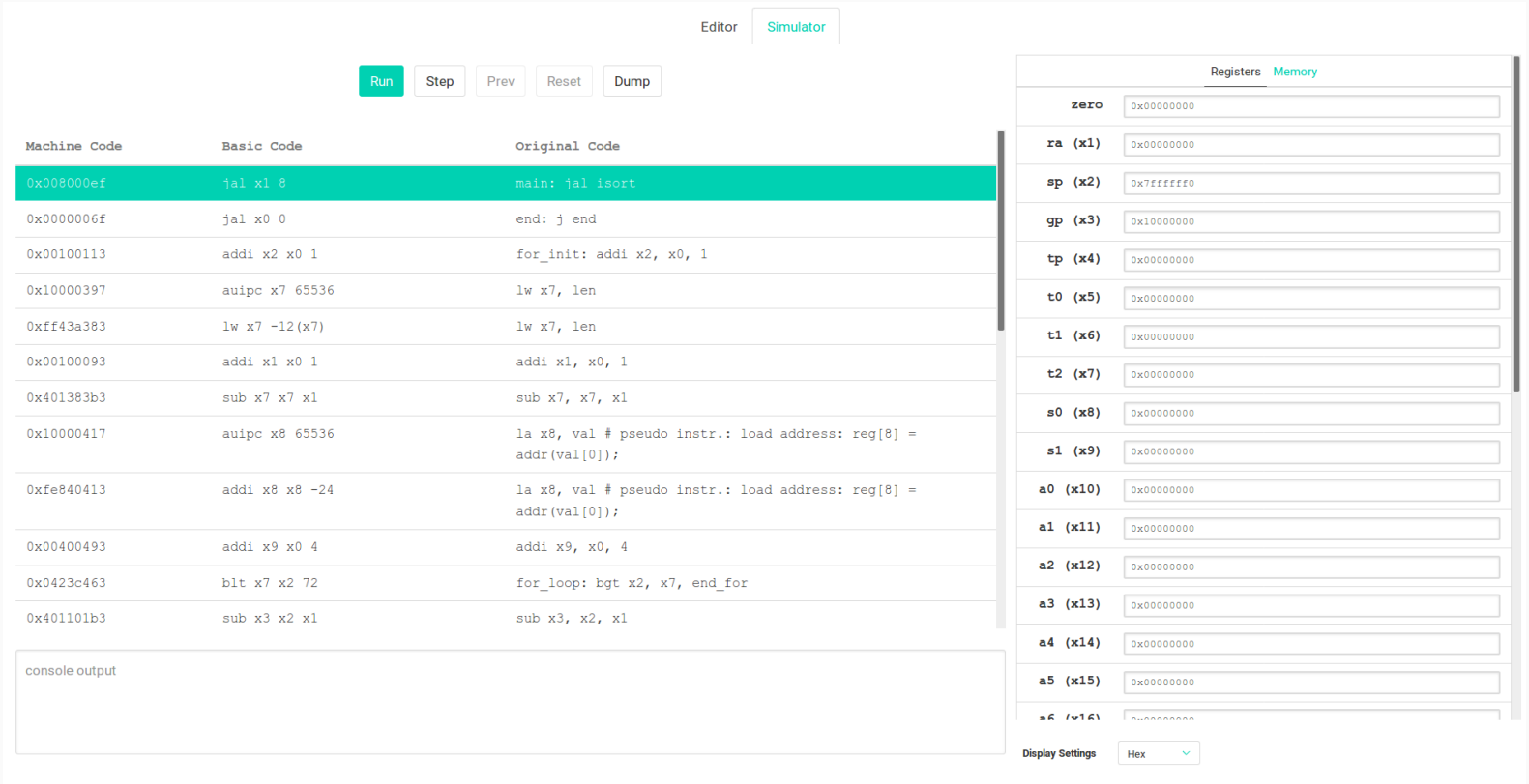
Instruction Set Simulator

Should be

- Accurate (cycle/timing or at least only functional)
- Fast to simulate entire systems (>100MIPS)
- expandable for new interfaces to the outer world
- customizable to new instructions
- Allows Multicore simulations



Instruction-Set Simulators (Venus)



Venus (http://www.kvakil.me/venus/): Simple online simulator from Berkeley to learn RISC-V assembly language

Venus: Features

- RV32IM ISA
- Single-step debugging with undo feature
- Breakpoint debugging
- View machine code and original instructions side-by-side
- Several ecalls: including `print` and `sbrk`
- Memory visualization
- Integrated CodeMirror editor with syntax highlighting
- In order to assemble code, simply click on the "Simulator" tab

Venus: (Pseudo) Instructions

- Supported Instructions
  At the moment, venus supports most of the RV32IM standard. `ebreak`, `fence`, `fence.i` and cycle counter instructions are not currently supported
- Pseudo Instructions
  Venus currently supports all standard RISC-V pseudoinstructions as long as they translate into a series of supported instructions

Venus: Assembler directives

The following assembler directives are supported:

Directive	Effect
`.data`	Store subsequent items in the [static segment]
`.text`	Store subsequent instructions in the [text segment]
`.byte`	Store listed values as 8-bit bytes
`.asciiz`	Store subsequent string in the data segment and add null-terminator
`.word`	Store listed values as unaligned 32-bit words
`.globl`	Makes the given label global

Venus: Environmental Calls

- To use an environmental call, load the ID into register a0, and load any arguments into a1 - a7
- Any return values will be stored in argument registers

The following environmental calls are currently supported:

ID (a0)	Name	Description
1	print_int	prints integer in a1
4	print_string	prints the null-terminated string whose address is in a1
9	sbrk	allocates a1 bytes on the heap, returns pointer to start in a0
10	exit	ends the program
11	print_character	prints ASCII character in a1
17	exit2	ends the program with return code in a1

As an example, the following code prints the integer 42 to the console:

```
addi a0 x0 1  # print_int ecall
addi a1 x0 42 # integer 42
ecall
```

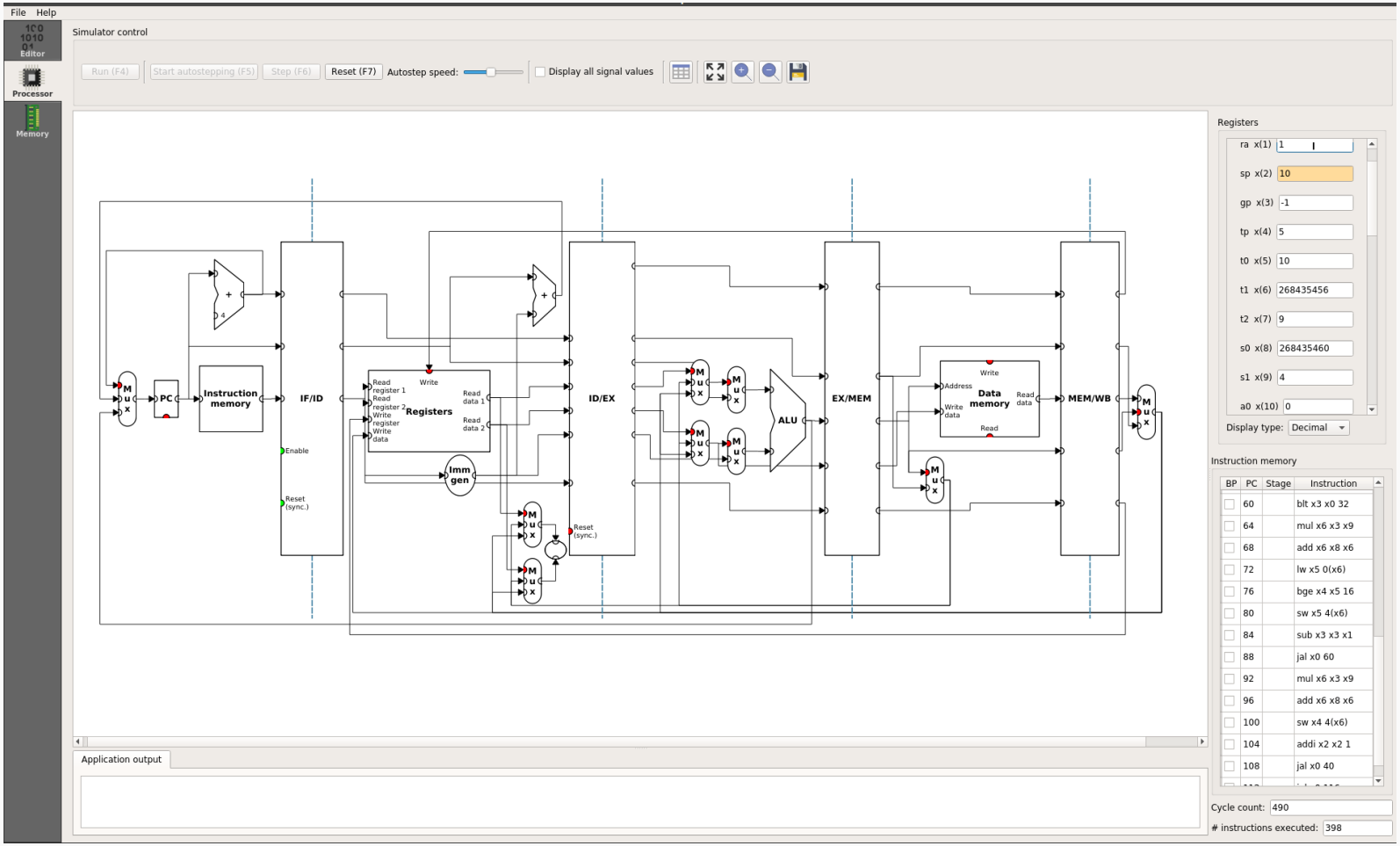
Memory Segments

Memory is divided into four different segments: text, static, heap and stack

Segment	Starting Address
text	0x0000_0000
static	0x1000_0000
heap	0x1000_8000
stack*	0x7fff_fff0

* Unlike other memory segments, the stack grows towards smaller addresses.

Instruction-Set Simulators (Ripes)



eebcRipes (https://github.com/mortbopet/Ripes): 5-stage pipelined RISC-V simulator