RISC-V ISA

CS 3220 / CS 5220 Lecture 3-B

Jason Hibbeler

University of Vermont

Spring 2024

Topics

Registers

Example instructions

The stack

Function calls

Analogies

Everyone uses the “Honda Civic vs. Maserati” analogy to represent the relationship between high-level languages and assembly code.

I prefer a different analogy.

Analogies

Everyone uses the "Honda Civic vs. Maserati" analogy to represent the relationship between high-level languages and assembly code.

I prefer a different analogy.

a program written in your favorite high-level language

a program written in assembly

RISC-V

RISC-V (pronounced “risk five”) is an open-standard instructions-set architecture (ISA)

- it was designed to be simple
- it serves as a reference ISA for the study of hardware architecture
- it’s provided under open-source licenses
- some companies have produced CPUs that implement RISC-V

For CS 3220 / CS 5220

- we’ll use the base 32-bit version of RISC-V
- 32-bit instructions, and 32-bit data

RISC-V

Characteristics

- all operations on data apply to data in registers (and typically change the entire register); registers are 32 bits or 64 bits in length
- the only operations that affect memory are load and store, which move data from memory to a register, or vice versa
- the instruction formats are few and simple to parse
- and so for these reasons, RISC-V instructions are easy to pipeline*
- and RISC-V programs are easy to write and understand**

*coming soon

**you'll see this yourselves!

Glossary

imm: immediate, a numeric value

imm_{12}: a 12-bit immediate value

GPR: general-purpose register

rd: destination register

rs1: source register #1

rs2: source register #2

RISC-V Registers

Register	ABI name	Purpose
x0	zero	hard-wired value of zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5	t0	temporary
x6	t1	temporary
x7	t2	temporary
x8	s0/fp	saved register / frame pointer
x9	s1	saved register
x10	a0	function argument / return value
x11	a1	function argument / return value
x12 – x17	a2 – a7	function arguments
x18 – x27	s2 – s11	saved registers
x28 – x31	t3 – t6	temporaries

The registers in red are the ones we'll use in our RISC-V code

RISC-V doesn't enforce the behavior of any of these—it's up to the software (the compiler)

And in particular, our RISC-V interpreter doesn't make any assumptions about how registers are used—it's up to the programmer to manage them

Think of the x* names as the hardware names and the other names as the software names

ABI: application binary interface; how a user of real RISC-V would identify the registers

Examples of RISC-V Instructions

Now we'll look at examples of some specific instructions from different categories

Note: RISC-V is not case-sensitive

- do whatever you're comfortable with
- I mix cases from time to time in the examples I'll show

Special Register: `x0`

`x0` has the value zero

- for ever and ever and ever

It's a read-only register

- can be used as the destination register in statements where you don't care about what's being produced
- can also be used to "build" numbers (as the constant zero)

```
addi a0, x0, 56  # this will add 0+56 and put the result in register a0
```

Sign Extension

Sign extension describes how the arithmetic sign of a signed binary number is represented

- there's no "negative sign" in modern CPUs
- all modern systems use two's complement

Two's complement: for an N-bit number x, the sum of x and $-x$ is 2^N

So if $x = 15$

- then as an 8-bit binary number this is 00001111
- and -15 is 11110001

Conversion: if the high-order bit is set, then it's a negative number; flip all of the bits and add 1 to get the magnitude

Example: 11110001 ⇒ 00001110
and 00001110 + 1 = 00001111 = 15

Two's Complement

Example eight-bit signed values

0	00000000
1	00000001
2	00000010
126	01111110
127	01111111
-128	10000000
-127	10000001
-2	11111110
-1	11111111

Sign Extension

Two's complement: another example, with 16-bit numbers

So if $x = 15$

- then as a 16-bit binary number this is 00000000 00001111
- and -15 is 11111111 11110001

Sign Extension

Why is this important?

- not all numbers are positive—we need a way to represent negative numbers also

Implication:

- suppose I have an 8-bit two's complement signed number
- and I want to represent that number using 16 bits
- for example: I save a value stored as a single byte in a two-byte location

First, let's consider a positive number

Sign Extension

If I consider 15 as an eight-bit binary number, then it's `00001111`

If I want to store this number in a 16-bit storage location, then I have to "extend" it:

`00001111 → 00000000 00001111`

This is straightforward for a positive number: just pad out the number with zeros

But what about for a negative number?

Sign Extension

If I have -15 as a two's complement eight-bit number, then it's `11110001`

And to store this number in a 16-bit storage location, I have to "sign-extend" it:

`11110001 → 11111111 11110001`

In other words, the arithmetic sign of the result should be unchanged

- so if the number is negative, we have to pad it out with ones

RISC-V Instructions

Now we'll look in detail at some representative RISC-V instructions

- `ADDI`, `ADD`
- `BEQ`
- `JAL`, `JALR`
- `LW`
- `SW`

ADDI

ADDI: Add Immediate

ADDI rd, rs1, imm_{12}

there's also SUBI

GPR[rd] ← GPR[rs1] + sign-extend(imm_{12})

PC ← PC + 4

How to read this:

- put the sum of the contents of the general-purpose register specified by rs1 and the sign-extended immediate value in the GPR specified by rd
- increment the program counter by four (so that it points to the next instruction—all RISC-V instructions are four bytes in length)

ADDI

`ADDI`: Add Immediate

`ADDI rd, rs1,` $\texttt{imm}_{12}$

there's also `SUBI`

`GPR[rd]` $\leftarrow$ `GPR[rs1] + sign-extend(`$\texttt{imm}_{12}$`)`

`PC` $\leftarrow$ `PC + 4`

For "I" (immediate) instructions, it's the funct3 field and the opcode field that determine what the instruction is

the imm value	source register #1	funct3	destination register	opcode
imm_{12}	rs1	000	rd	0010011
12 bits	5 bits	3 bits	5 bits	7 bits

four bytes (all RISC-V instructions are four bytes)

ADD

ADD: Add

```
ADD rd, rs1, rs2
```

there's also `SUB`

```
GPR[rd] ← GPR[rs1] + GPR[rs2]
PC ← PC + 4
```

funct7	source register #2	source register #1	funct3	destination register	opcode
0000000	rs2	rs1	000	rd	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

For "R" instructions (3x registers), the funct3 and funct7 fields and the opcode field determine what the instruction is

`ADD`, `ADDI`: Examples

Suppose that `a1` contains the value 128 and `a2` contains the value 25

```
ADD a0, a1, a2   # after this instruction, a0 will contain 153

ADDI a0, a1, 67  # after this instruction, a0 will contain 195
```

BEQ

`BEQ`: Branch if Equal

`BEQ rs1, rs2,` $\texttt{imm}_{13}$

there's also `BGE`, `BNE`, `BLT`, etc.

`target = PC + sign_extend(`$\texttt{imm}_{13}$`)`

`if GPR[rs1] == GPR[rs2]`

`then PC` $\leftarrow$ `target`

`else PC` $\leftarrow$ `PC + 4`

destination	source register #2	source register #1	funct3	destination	opcode
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

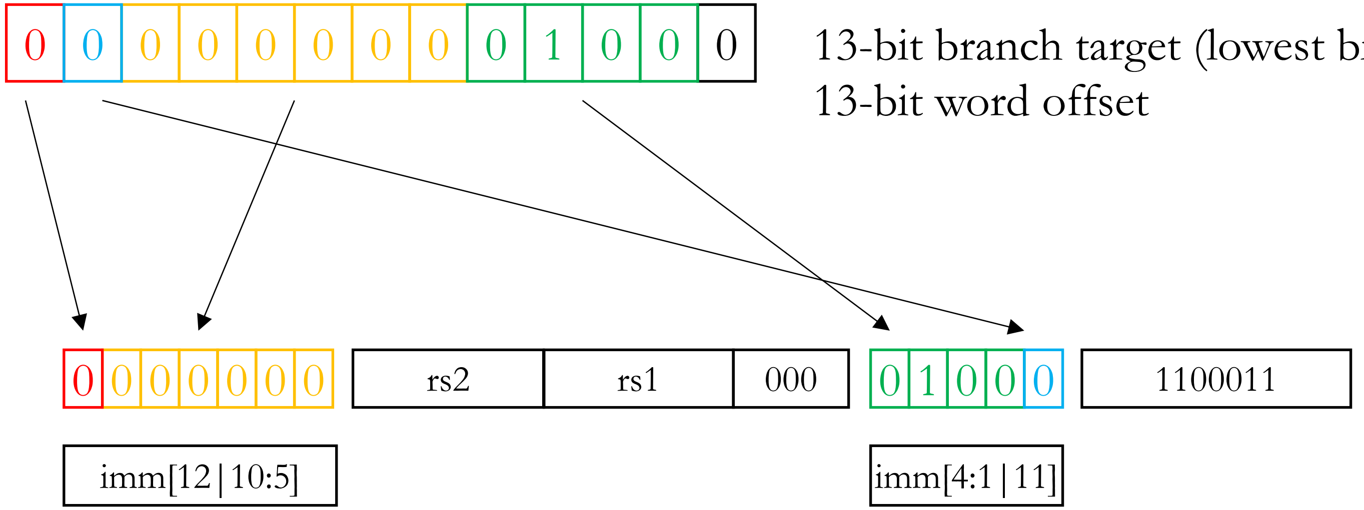
<u>destination must be four-byte aligned</u>

B-Type Addressing for Branch Instructions

The target of a branch instruction is a 13-bit immediate value

- it’s actually treated as a half-word offset (half-word: two bytes)
- and the lowest-order bit is always assumed to be zero (can’t jump an odd number of half-words)
- and the 12 bits are split between two fields in a branch instruction

Example: suppose the branch value is 16 (i.e., PC ← PC + 16): this is 8 half-words



13-bit branch target (lowest bit is discarded), representing a 13-bit word offset

B-Type Addressing for Branch Instructions

Why so complicated?

Simple reason: to speed up instruction decoding

- keeps the register specifiers in the same place in every instruction

General RISC-V Instruction Formats

This shows the six different formats:

32-bit RISC-V instruction formats

<table>
<tr><th rowspan="2">Format</th><th colspan="32">Bit</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Register/register</td><td colspan="7">funct7</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Immediate</td><td colspan="12">imm[11:0]</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Upper immediate</td><td colspan="20">imm[31:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Store</td><td colspan="7">imm[11:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">imm[4:0]</td><td colspan="7">opcode</td></tr>
<tr><td>Branch</td><td>[12]</td><td colspan="6">imm[10:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="4">imm[4:1]</td><td>[11]</td><td colspan="7">opcode</td></tr>
<tr><td>Jump</td><td>[20]</td><td colspan="10">imm[10:1]</td><td>[11]</td><td colspan="8">imm[19:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
</table>

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats*.
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

BEQ: Example

Suppose `a1` contains 43 and `a2` contains 43

```
beq  a1, a2, 8    # this will add 8 to the PC
addi a0, x0, 18
addi a2, x0, 34
```

and land here

BEQ: Example

But to make the code readable, we can use text labels

- the labels (and offsets) are converted to numeric values by the assembler

```
  beq  a1, a2, L1   # if a1==a2, then this will branch to L1
  addi a0, x0, 18
L1:
  addi a2, x0, 34
```

Technically, labels should start with a dot

- e.g., `.L1`
- this keeps them out of the symbol table (hides them from the outside world)
- but in your assembly code, you can use whatever you want for labels

JAL

`JAL`: Jump And Link

$\texttt{JAL rd, imm}_{20}$

$\texttt{target} \leftarrow \texttt{PC + sign-extend(imm}_{20}\texttt{)}$
$\texttt{GPR[rd]} \leftarrow \texttt{PC + 4}$
$\texttt{PC} \leftarrow \texttt{target}$

target	destination register	opcode
imm[20|10:1|11|19:12]	rd	1101111
20 bits	5 bits	7 bits

- destination must be four-byte aligned
- the imm value can be a label, in which case the displacement is computed automatically
- 4 + the current value of the PC is saved in the destination register
- the increment by 4 means that we will return to the instruction following the `JAL` if use the value saved in rd as a jump target

JAL: Example

```
  addi a0, x0, 0       # set a0 to zero
  addi a1, x0, 4       # set a1 to 4
LOOP:
  addi a0, a0, 1       # increment a0
  beq a0, a1, DONE     # if a0 == a1, then jump to DONE
  jal x0, LOOP         # jump to the label LOOP; save PC+4 to x0
DONE:
  # more instructions
```

as an actual offset, this would be 8

as an actual offset, this would be -8

Here, the "save PC to x0" has no effect, since x0 is a read-only register

This is the structure of a for loop (or while loop)

JALR

`JALR`: Jump Indirect

$\texttt{JALR rd, rs1, imm}_{12}$

$\texttt{target = GPR[rs1] + sign-extend(imm}_{12}\texttt{)}$

$\texttt{GPR[rd]} \leftarrow \texttt{PC + 4}$

$\texttt{PC} \leftarrow \texttt{target}$

target	source register	funct3	destination register	opcode
imm[11:0]	rs1	000	rd	1100111
12 bits	5 bits	3 bits	5 bits	7 bits

- destination must be four-byte aligned
- the imm value can be a label, in which case the displacement is computed automatically
- the increment by 4 means that we will return to the instruction following the `JALR`

JALR: Example

```
  addi a0, x0, 3   # put 3 in a0
  addi a1, x0, 4   # put 4 in a1
  addi a3, x0, 19  # put 19 in a3
  jal  a2, FCN     # jump to the label FCN; save PC+4 to a2
  addi a3, x0, 12  # put 12 in a3
  jal  x0, L2      # jump to the label L2; save PC+4 to x0 (no effect)
FCN:
  addi a3, x0, 4   # put 4 in a3
  jalr x0, a2, 0   # jump to the value in a2 (this is effect a return)
L2:
  # more instructions
```

Here, the "save PC to x0" has no effect, since x0 is a read-only register

`JAL` vs. `JALR`

`JAL`

- saves the current value of the PC
- transfers control to a specific location, specified as an immediate value (a value that's known at compile time)

`JALR`

- saves the current value of the PC
- transfers control to a location computed from an immediate and the contents of a register
- in other words, the offset does not need to be known at compile time
- this allows us to implement a return statement for a function call

Handy Rule

As an offset for a jump or a branch, 0 will cause an infinite loop

For example:

```
jal x0, 0
```

or:

```
beq a0, a1, 0
```

LW

LW: load word

LW rd, offset$_{12}$(rs1)

btye_address$_{32}$ = sign_extend(offset$_{12}$) + GPR[rs1]

GPR[rd] $\leftarrow$ mem[byte_address]

PC $\leftarrow$ PC + 4

immediate value	source register	funct3	destination register	opcode
imm[11:0]	rs1	010	rd	0000011
12 bits	5 bits	3 bits	5 bits	7 bits

SW

SW: store word

$\texttt{SW rs2, offset}_{12}\texttt{(rs1)}$

$\texttt{byte\_address}_{32} = \texttt{sign\_extend(offset}_{12}\texttt{) + GPR[rs1]}$

$\texttt{mem[byte\_address]} \leftarrow \texttt{GPR[rs2]}$

$\texttt{PC} \leftarrow \texttt{PC + 4}$

immediate value	source register #2	source register #1	funct3	immediate value	opcode
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Example: `LW, SW`

Suppose `a0` contains 17 and `a1` contains 124

```
sw a0, 8(a1)       # this puts the value 17 in memory location 132
addi a0, x0, 132  # a0 now has 0 + 132 = 132
lw a2, 0(a0)       # this loads 17 (from mem[132]) into a2
```

Observation: there is not a unique way of referring to a particular memory address

- offsets can be negative; e.g. `sw a0, -8(a2)`

Also: RISC-V is not case sensitive

- you can use uppercase or lowercase, as you prefer

SLTI

SLTI: set [register] if less than, immediate

kind of like the first part of a branch instruction

SLTI rd, rs1, imm_{12}

if GPR[rs1] $<$ sign-extend(imm_{12})
  GPR[rd] $\leftarrow$ 1
else
  GPR[rd] $\leftarrow$ 0
PC $\leftarrow$ PC + 4

immediate value	source register	funct3	destination register	opcode
imm_{12}	rs1	010	rd	0010011
12 bits	5 bits	3 bits	5 bits	7 bits

SLT

SLT: set [register] if less than

```
SLT rd, rs1, rs2
```

```
if GPR[rs1] < GPR[rs2]
  GPR[rd] ← 1
else
  GPR[rd] ← 0
PC ← PC + 4
```

funct7	source register #2	source register #1	funct3	destination register	opcode
0000000	rs2	rs1	010	rd	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Additional Instructions

Here are additional instructions you might need for the programming assignments

`sltu, sltiu`	signed and unsigned comparisons
`or, ori`	bitwise or
`and, andi`	bitwise and
`xor, xori`	bitwise xor
`srli, srli`	bitwise right shift
`sra, srai`	bitwise right shift
`sll, slli`	bitwise left shift

RISC-V Documentation

Here is the official manual

- https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
- it's a bit dense and not very user friendly
- but it is the official guide to RISC-V

Here's a more readable document

- https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html
- definitely easier to read

Example

```c
int f() {
  int i, sum;
  sum = 0;
  for (i=0; i<4; ++i)
    sum = sum + i;
  // rest of code
}
```

```
f:
  addi a0, x0, 0    # set a0 to 0 : this represents i
  addi a1, x0, 0    # set a1 to 0 : this represents sum
  addi a2, x0, 4    # set a2 to 4 : this is the constant 4
L1:
  bge a0, a2, L2    # branch if a0 >= a2
  add a1, a1, a0    # sum ← sum + i
  addi a0, a0, 1    # i ← i + 1
  jal x0, L1        # jump to L1 (and don't save PC)
L2:
  # rest of code
```

Here, we're not actually using memory to store the program variables—all of the values are kept in registers: think of this as a highly optimized kernel

The Stack

Each time a function is called, a small working area for the function is created

- by reserving a region on the stack
- this region contains storage for local variables
- and for function parameters (which, in a sense, are just local variables)

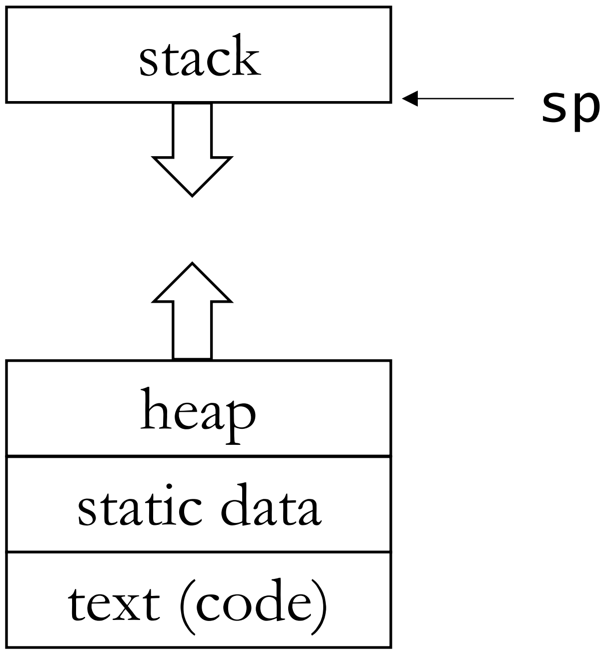
We also save two pieces of bookkeeping information:

- the return address for the function
- the current top of the of the stack (also called the frame pointer)

The Stack

Simplified view of the memory image of a Linux process

- the stack grows downward



So, to reserve space on the stack in a function

- we decrement the stack pointer (`sp`) by the number of bytes we want to reserve

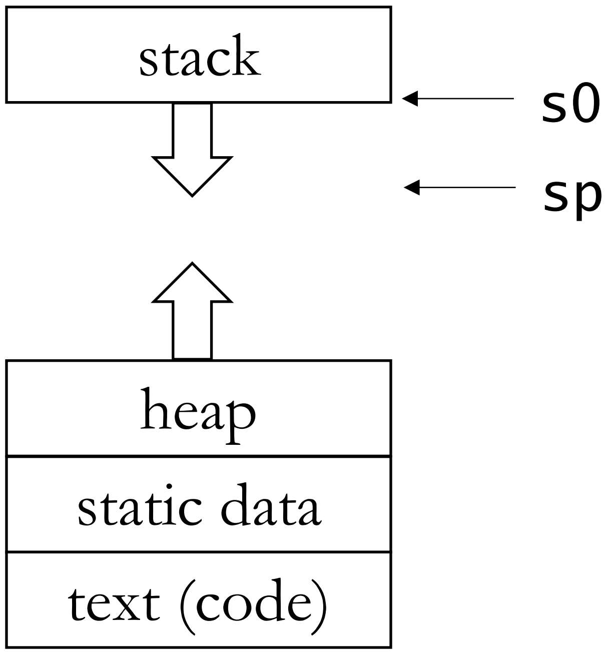
The frame pointer (`s0` / `fp`)

- as a convenience, the compiler saves the current value of `sp` in this register before decrementing `sp`
- that means my local variables will be saved in `s0-offset`

The Stack

After reserving space on the stack

- by saving the current value of `sp` to `s0`
- and then decrementing `sp`



The Stack, in Action

```
int f(int p) {
  int i = p + 4;
  int j = g(i);
  return j;
}
```

What local information do I need here?
- the parameter `p`
- the two local variables: `i` and `j`
- where I came from (the return address)
- what the top of the stack was (the frame pointer)

in RISC-V, the register `s0` is the frame pointer

```
int g(int q) {
  if (q > 0)
    return 1;
  else
    return 0;
}
```

What local information do I need here?
- the parameter `q`
- where I came from (the return address)
- what the top of the stack was (the frame pointer)

```
int h() {
  int var = 45;
  int rc = f(var)
  return 0;
}
```

What local information do I need here?
- the two local variables (`var` and `rc`)
- where I came from (the return address)
- what the top of the stack was (the frame pointer)

The Stack, in Action

```
int f(int p) {
  int i = p + 4;
  int j = g(i);
  return j;
}


int g(int q) {
  if (q > 0)
    return 1;
  else
    return 0;
}

int h() {
  int var = 45;
  int rc = f(var)
  return 0;
}
```

Each function call overwrites the value of `ra` and `sp`

Remember what a function call means:

- jump to a different place in the code (in the stream of instructions)
- and then jump back to where you came from
- this means we need to keep track of where we came from: this is `ra`
- also keep track of what the stack pointer was at the beginning of the function: this is `s0` (the frame pointer)

The Stack, in Action

After `h()` has called `f()` and `f()` has called `g()`

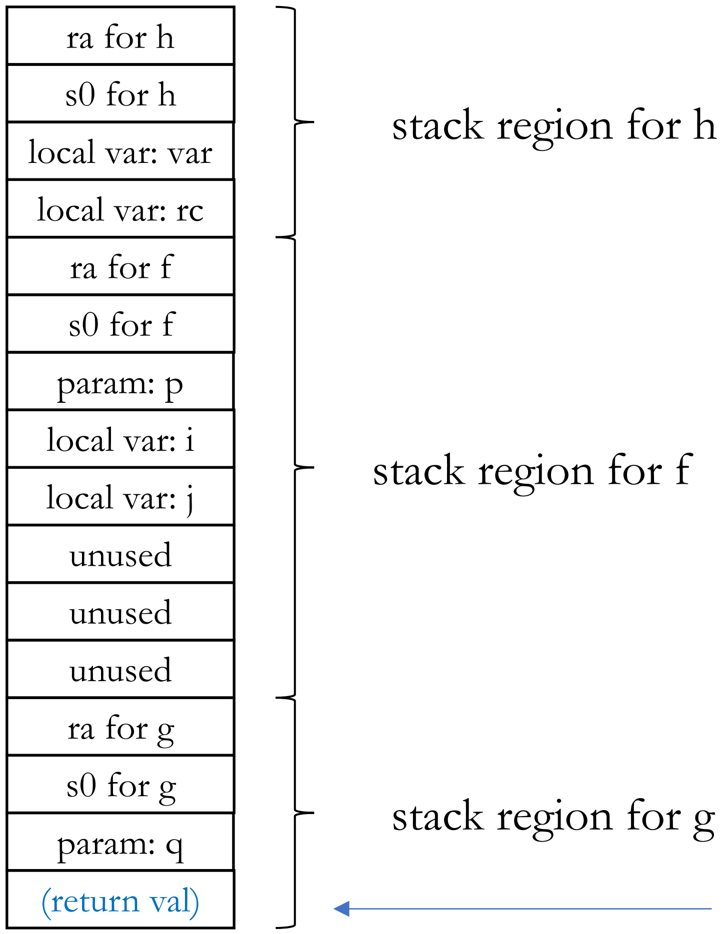
Convention:

- save the return address (`ra`)
- save the frame pointer (`s0`)
- save function parameters
- save local variables

Parameters are passed in registers

- first param in `a0`
- second param in `a1`
- etc.

the stack is allocated in multiples of four words



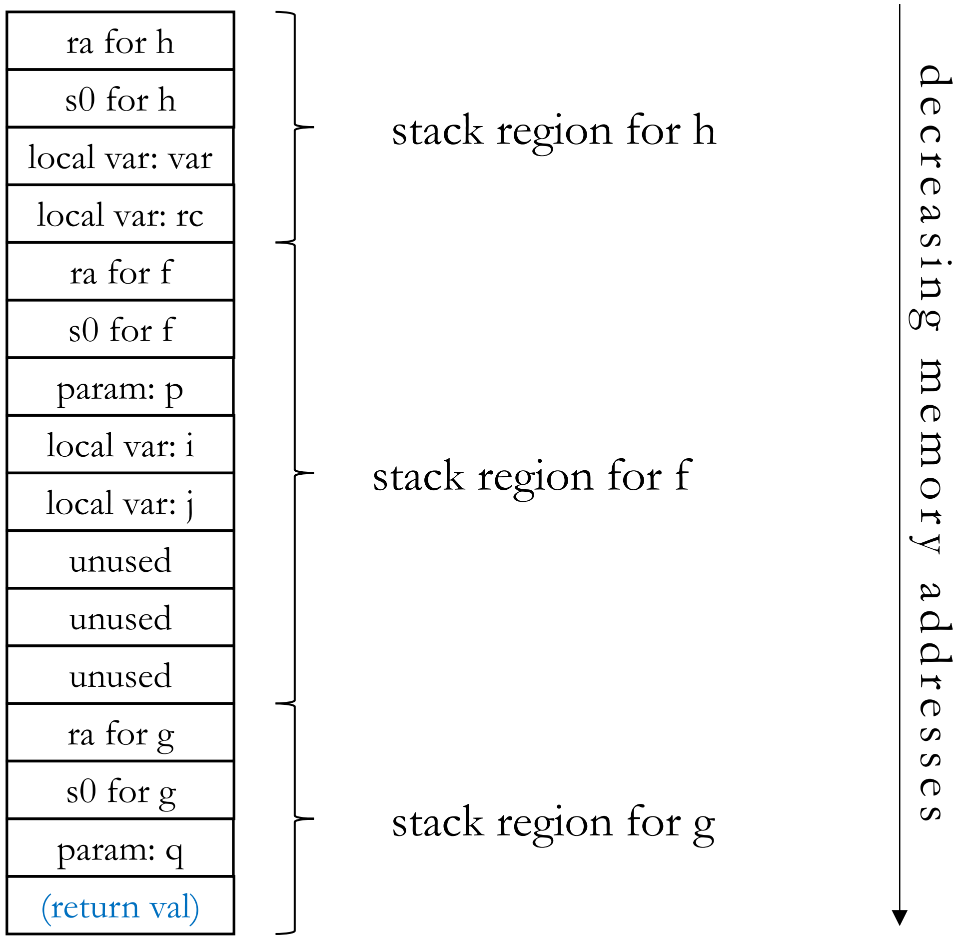
stack space is used for the function's return val, because the logic of the return val is split across a branch; this is kind of a technicality, and we can ignore this

The Stack, in Action

```
int f(int p) {
  int i = p + 4;
  int j = g(i);
  return j;
}


int g(int q) {
  if (q > 0)
    return 1;
  else
    return 0;
}

int h() {
  int var = 45;
  int rc = g(var)
  return 0;
}
```



`sp` vs `s0`

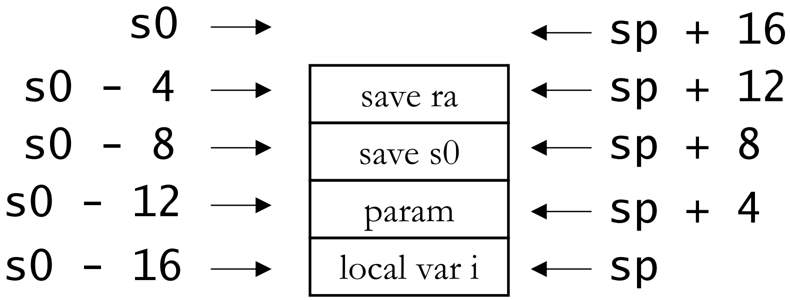
This is a little confusing

- the first instructions in a function reserve space on the stack by decrementing the stack pointer (`sp`)
- `s0` (the previous top of the stack) will then point to the top of the stack region for the function
- so an address in the stack can be referred to as $\texttt{s0} - offset_1$; or equivalently as $\texttt{sp} + offset_2$, where $offset_1 + offset_2$ is equal to the size of the stack frame for the function
- it's OK to do either one in our code

Here's the instruction to reserve 16 bytes on the stack:

```
addi sp, sp, −16
```

After this instruction:

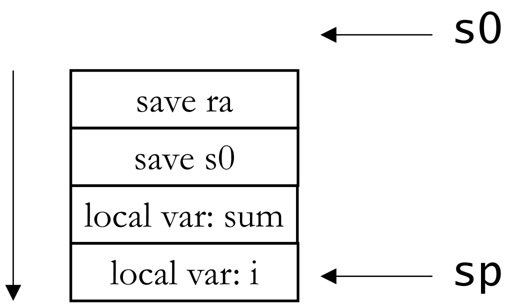


Example

```
int f() {
  int i, sum;
  sum = 0;
  for (i=0; i<4; ++i)
    sum = sum + i;
  // rest of code
}
```

```
f:
  addi a0, x0, 0     # set a0 to 0
  sw a0, -16(s0)     # store a0 in mem[s0-16] : this is i
  sw a0, -12(s0)     # store a0 in mem[s0-12] : this is sum
L1:
  lw a0, -16(s0)     # i
  addi a2, x0, 4     # put 4 in a2
  lw a1, -12(s0)     # sum
  bge a0, a2, L2     # branch if i >= 4
  add a1, a1, a0     # sum = sum + i
  sw a1, -12(s0)     # save sum to memory
  addi a0, a0, 1     # i = i + 1
  sw a0, -16(s0)     # save i to memory
  jal x0, L1         # jump to L1
L2:
  # rest of code
```

stack space is allocated in multiples of four words



This example doesn't have complete management of the stack — that's coming soon

Making a Function Call

Required steps to enable a function call:

1. put the function parameters in a place (registers) where the function can access them
2. transfer control to the function (a jump)
3. reserve local storage on the stack (for local variables) for the function
4. fetch and save local parameters from registers; also save `s0` and `ra`
5. execute the instructions of the function
6. put the result (if there is one) in a place where the caller can access it (a register)
7. restore the saved `s0` and `ra` registers
8. return control to the point of origin (another jump)

Example: Function Call

```c
f(9, 7);
// rest of code
```

```
addi a0, x0, 9   # set a0 to 9
addi a1, x0, 7   # set a1 to 7
jal ra, f        # save PC in ra and jump to label f
# rest of code
```

Conventions

- first param in `a0`, second param in `a1`
- the return address is saved in the register `ra`

The callee will manage the stack

Example: One Parameter, One Local Variable

return address will be in `ra`

```
int f1(int p) {
  int i;
  i = p + 45;
  return i;
}
```

```
f1:
  addi   sp, sp, -16   # reserve 4 words on the stack
  sw     ra, 12(sp)    # save ra in mem[sp+12]
  sw     s0, 8(sp)     # save fp (s0) in mem[sp+8]
  addi   s0, sp, 16    # s0 ← sp + 16
  sw     a0, -12(s0)   # save a0 in mem[s0-12]
  lw     a0, -12(s0)   # load a0 from mem[s0-12]
  addi   a0, a0, 45    # a0 ← a0 + 45
  sw     a0, -16(s0)   # store a0 in mem[s0-16]
  lw     a0, -16(s0)   # load a0 from mem[s0-16]
  lw     s0, 8(sp)     # load s0 from mem[sp+8]
  lw     ra, 12(sp)    # load ra from mem[sp+12]
  addi   sp, sp, 16    # restore sp: sp ← sp + 16
  jalr   x0, ra, 0     # jump to return address
```

input parameter (p) is in `a0`

save i

put return value in `a0`

restore `s0`

load return address

restore the stack

← `s0`

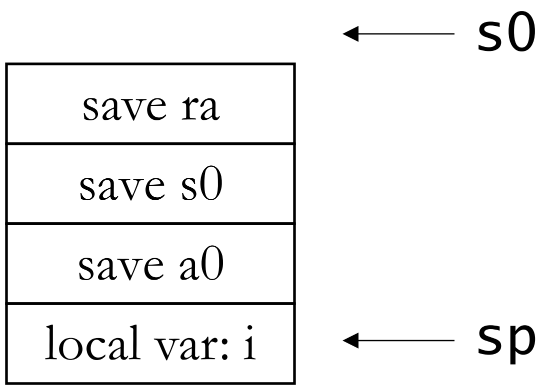
save ra
save s0
save a0
local var i

← `sp`

This <u>does</u> show complete and correct management of the stack

The Stack: During the Function Call

Here's what's stored on the stack during execution of `f`



After decrementing `sp` by 16 and setting `s0 = sp+16`:

```
mem[s0-16] = mem[sp]
mem[s0-12] = mem[sp+4]
mem[s0-8] = mem[sp+8]
```

etc.

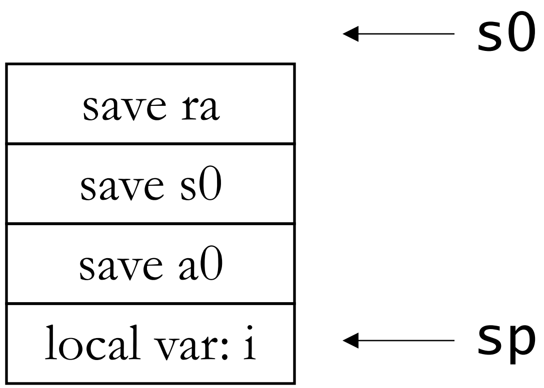
and in code, after setting `s0 = sp+16`:

`lw a0, -16(s0)` is equivalent to `lw a0, 0(sp)`
`lw a0, -12(s0)` is equivalent to `lw a0, 4(sp)`
etc.

Note: by convention, the RISC-V stack is allocated in blocks of four words

The Stack: During the Function Call

Here's what's stored on the stack during execution of f



Note: by convention, the RISC-V stack is allocated in blocks of four words

And here, only `a0` is used for parameter passing

We always save `ra` and `s0`

And we only need to save `a0`, since it's the only register used for parameter passing in this function: `int f1(int);`

Reason: if this function calls another function, then we need to restore the current values (inside this function) of `ra` and `s0` after that second function call; the parameter `a0` acts like a local variable for the function

Example: Two Parameters, Two Local Variables

```c
int f2(int p1, int p2) {
  int i, j;
  i = p1 + 3;
  j = p2 + 7;
  return i+j;
}
```

input parameters:
p1 is in a0, p2 is in a1

use a0 for computation

put return value in a0

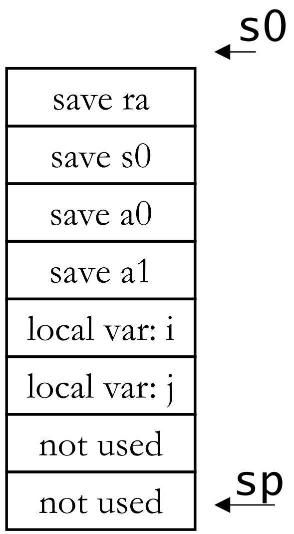
restore s0

load return address

restore the stack

compiler reserves stack space in
blocks of four words

```
f2:
  addi    sp, sp, -32    # reserve 8 words on the stack
  sw      ra, 28(sp)     # save ra in mem[sp+28]
  sw      s0, 24(sp)     # save s0 in mem[sp+24]
  addi    s0, sp, 32     # s0 ← sp + 32
  sw      a0, -12(s0)    # save a0 in mem[s0-12] (mem[sp+20])
  sw      a1, -16(s0)    # save a1 in mem[s0-16] (mem[sp+16])
  lw      a0, -12(s0)    # load a0 from mem[s0-12] : this is p1
  addi    a0, a0, 3      # a0 ← a0 + 3
  sw      a0, -20(s0)    # save a0 in mem[s0-20] : this is i
  lw      a0, -16(s0)    # load a0 from mem[s0-16] : this is p2
  addi    a0, a0, 7      # a0 ← a0 + 7
  sw      a0, -24(s0)    # store a0 in mem[s0-24] : this is j
  lw      a0, -20(s0)    # load a0 from mem[s0-20] : this is i
  lw      a1, -24(s0)    # load a1 from mem[s0-24] : this is j
  add     a0, a0, a1     # a0 ← a0 + a1 : this is now i + j
  lw      s0, 24(sp)     # load s0 from mem[sp+24]
  lw      ra, 28(sp)     # load ra from mem[sp+28]
  addi    sp, sp, 32     # sp ← sp + 32
  jalr    x0, ra, 0      # jump to return address
```



The Stack: During the Function Call

Here's what's stored on the stack during execution of f2

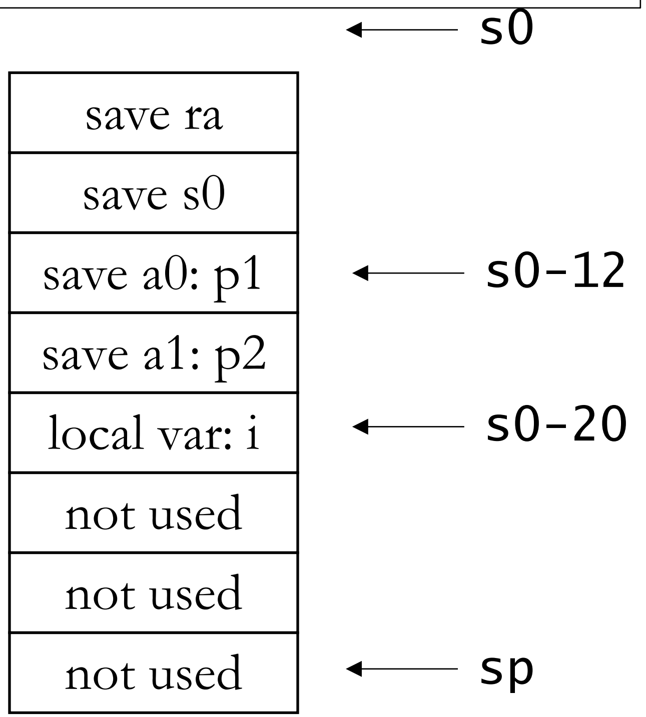
	← s0
save ra	
save s0	
save a0	this will be the param p1
save a1	this will be the param p2
local var: i	
local var: j	
not used	
not used	← sp

```
int f2(int p1, int p2) {
  int i, j;
  i = p1 + 3;
  j = p2 + 7;
  return i+j;
}
```

Again, the RISC-V stack is allocated in blocks of four words

Example: Two Parameters, One Local Variable

```c
int f3(int p1, int p2) {
  int i;
  i = p1 + p2;
  return i;
}
```

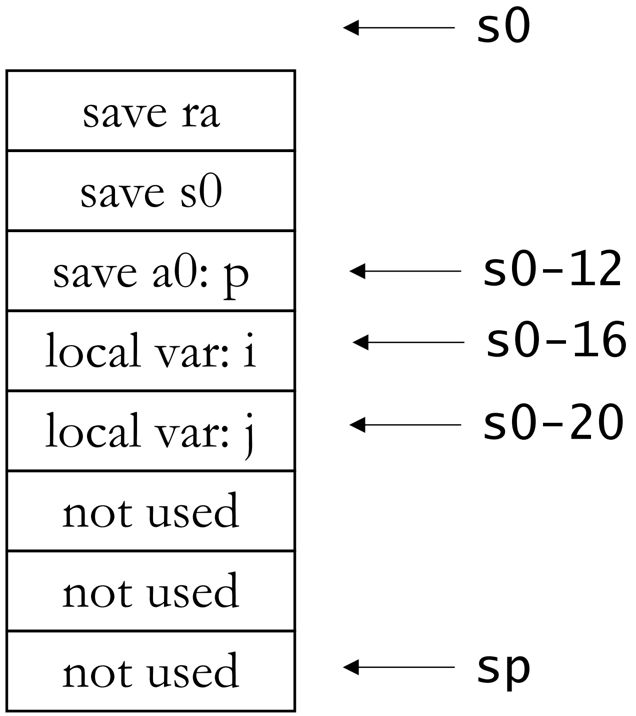



```
f3:
  addi    sp, sp, -32    # sp <- sp - 32
  sw      ra, 28(sp)     # save ra on the stack
  sw      s0, 24(sp)     # save s0 on the stack
  addi    s0, sp, 32     # s0 <- sp + 32
  sw      a0, -12(s0)    # save a0 on the stack
  sw      a1, -16(s0)    # save a1 on the stack
  lw      a0, -12(s0)    # load a0 from the stack
  lw      a1, -16(s0)    # load a1 from the stack
  add     a0, a0, a1     # a0 <- a0 + a1
  sw      a0, -20(s0)    # i <- a0 (save i)
  lw      a0, -20(s0)    # load i
  lw      s0, 24(sp)     # restore s0
  lw      ra, 28(sp)     # restore ra
  addi    sp, sp, 32     # restore sp
  jalr    x0, ra, 0      # return (jump to ra)
```

Example: One Parameter, Two Local Variables

```
int f4(int p) {
  int i, j;
  i = p >> 1;
  j = i + p;
  return i+j;
}
```

```
f4:
  addi  sp, sp, -32
  sw    ra, 28(sp)
  sw    s0, 24(sp)
  addi  s0, sp, 32
  sw    a0, -12(s0)
  lw    a0, -12(s0)
  srai  a0, a0, 1
  sw    a0, -16(s0)
  lw    a0, -16(s0)
  lw    a1, -12(s0)
  add   a0, a0, a1
  sw    a0, -20(s0)
  lw    a0, -20(s0)
  lw    s0, 24(sp)
  lw    ra, 28(sp)
  addi  sp, sp, 32
  jalr  x0, ra, 0
```



Our Interpreter

Here's a JavaScript interpreter that I found at Cornell

- Vincent Moeykens brought it over to UVM
- https://riscv.jhibbele.w3.uvm.edu

It's a quick and easy way to learn RISC-V

- just put in your instructions
- and either step through or run
- it can run at the blindingly fast frequency of 256 Hz

Our Interpreter

Features:

- you can put a breakpoint on an instruction by left-clicking next to the line
- you can view registers and memory
- in the interpreter, memory addresses start at zero and are specified in hexadecimal

Caveat:

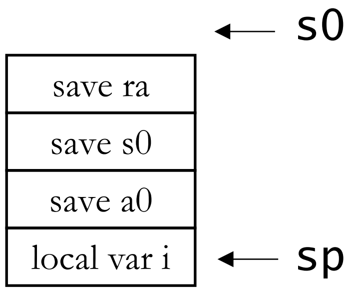
- supports a subset of RISC-V, but enough to learn and use the language
- supports integer operations only; but, this isn't a significant restriction for CS 3220 / CS 5220

Calling f1 in Our Interpreter

Suppose I want to call f1(14)

- set the stack pointer to some nonzero value, say 64
- put 14 in `a0`: this is the fcn param
- call f1(14)

```
int f1(int p) {
  int i;
  i = p + 45;
  return i;
}
```



I like to put my functions first, and my "main" after the functions

```
jal x0, main
```

this is literally the same code shown earlier

```
f1:
  addi    sp, sp, -16   # reserve 4 words on the stack
  sw      ra, 12(sp)    # save ra in mem[sp+12]
  sw      s0, 8(sp)     # save fp (s0) in mem[sp+8]
  addi    s0, sp, 16    # s0 ← sp + 16
  sw      a0, -12(s0)   # save a0 in mem[sp+4]
  lw      a0, -12(s0)   # load a0 from mem[sp+4]
  addi    a0, a0, 45    # a0 ← a0 + 45
  sw      a0, -16(s0)   # store a0 in mem[sp]; this is i
  lw      a0, -16(s0)   # load a0 from mem[sp]; this is the rtnval
  lw      s0, 8(sp)     # load s0 from mem[sp+8]
  lw      ra, 12(sp)    # load ra from mem[sp+12]
  addi    sp, sp, 16    # restore sp: sp ← sp + 16
  jalr    x0, ra, 0     # jump to return address
```

```
main:
  addi sp, x0, 64
  addi a0, x0, 14
  jal  ra, f1
  addi a2, a0, 0
```

- `addi sp, x0, 64` ← set `sp` to 64 (top of stack—here, an arbitrary nonzero value big enough for the stack needs)
- `addi a0, x0, 14` ← set `a0` to 14 (this is the function parameter)
- `jal ra, f1` ← call the function
- `addi a2, a0, 0` ← random statement just to show that the return value of the function has been placed in a0

Arrays and Vectors

In an array (vector) of integers, each element occupies four bytes*

Suppose the array is located starting at memory address 100

- then the first element is at memory address 100
- and the second element is at memory address 104
- and the third element is at memory address 108
- etc

*again, we are using the 32-bit base RISC-V

Arrays and Vectors

So, for example, if I want to load `a[2]`

- then the offset from the start of `a` is 8 (= 2 * 4)

Suppose that `a2` holds the starting address of the array

```
addi a0, x0, 0  # put 0 in a0: this is the offset of the element we want, in bytes
add a0, a0, a2  # add the starting addr: this is then the addr of the element we want
lw a1, 0(a0)    # and so this loads a[0]
```

Arrays and Vectors

Suppose that `a2` holds the starting address of the array

- and assume that `a0` is the index, currently set to zero

Now, to load `a2[1]`:

```
addi a0, a0, 1  # increment the index
slli a0, a0, 2  # multiply by four, to convert to a byte offset
add a0, a0, a2  # add the starting addr: this is then the addr of the element we want
lw a1, 0(a0)    # this loads a[1]
```

Example

Summing the elements of an array

```
int arraysum(int *p, int n) {
  int s = 0;
  for (int i=0; i<n; ++i) {
    s = s + p[i];
  return s;
}
```

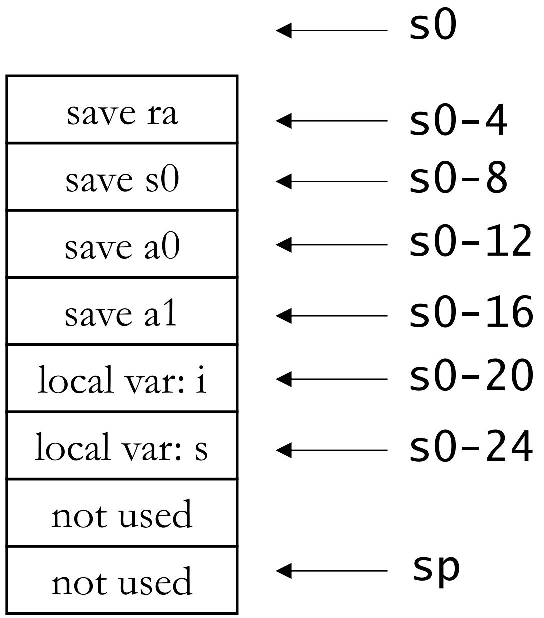
Example

Summing the elements of a vector

```
int arraysum(int *p, int n) {
  int s = 0;
  for (int i=0; i<n; ++i) {
    s = s + p[i];
  return s;
}
```

```
arraysum:
  addi    sp, sp, -32   # reserve 32 bytes on the stack
  sw      ra, 28(sp)    # save ra
  sw      s0, 24(sp)    # save s0
  addi    s0, sp, 32    # set s0, for convenience
  sw      a0, -12(s0)   # p, on the stack
  sw      a1, -16(s0)   # n, on the stack
  # should check that n <= 0 and error if so
  addi    a0, x0, 0     # put 0 in register a0
  sw      a0, -20(s0)   # i = 0, on the stack
  sw      a0, -24(s0)   # s = 0, on the stack

loop:
  # on next slide
```

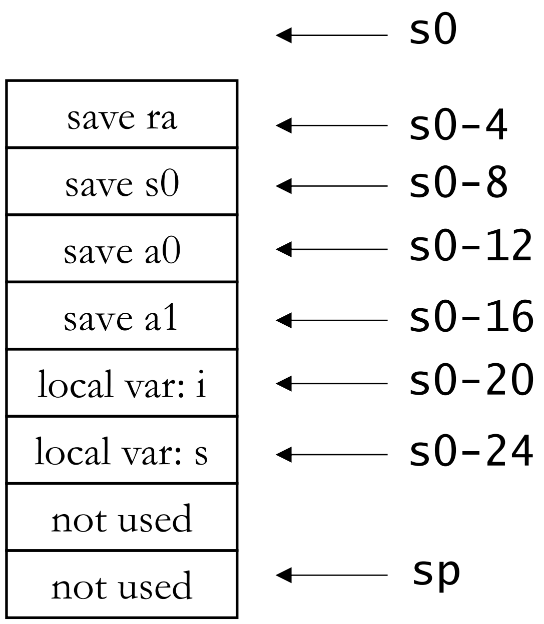


Example

Summing the elements of a vector

```
int arraysum(int *p, int n) {
  int s = 0;
  for (int i=0; i<n; ++i) {
    s = s + p[i];
  return s;
}
```

```
loop:
  lw    a0, -20(s0)     # load i
  lw    a1, -16(s0)     # load n
  beq   a0, a1, done    # branch if i == n
  lw    a1, -12(s0)     # load p
  slli  a0, a0, 2       # i = i * 4, to make it an array index
  add   a0, a0, a1      # this is &p[i]
  lw    a2, 0(a0)       # load p[i]
  lw    a0, -24(s0)     # load s
  add   a0, a0, a2      # a0 <- s + p[i]
  sw    a0, -24(s0)     # save s
  lw    a0, -20(s0)     # load i
  addi  a0, a0, 1       # increment i
  sw    a0, -20(s0)     # save i
  jal   x0, loop        # back to the top
```

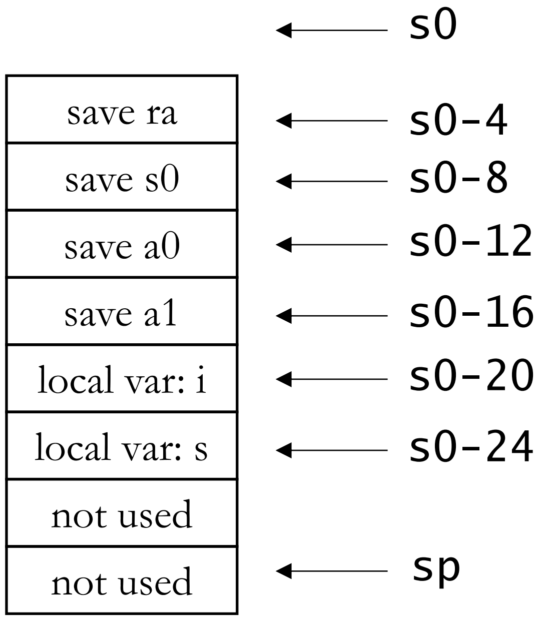


Example

```c
int arraysum(int *p, int n) {
  int s = 0;
  for (int i=0; i<n; ++i) {
    s = s + p[i];
  return s;
}
```

Summing the elements of a vector

```
done:
  lw      a0, -24(s0) # load s into a0 (the return value)
  lw      s0, 24(sp)  # restore s0
  lw      ra, 28(sp)  # restore ra
  addi    sp, sp, 32  # restore stack pointer
  jalr    x0, ra, 0   # return (jump to ra)
```



Tips for Writing Assembly Language

Here are a few tips for the RISC-V assignments

1. Have the C code in front of you for reference
2. Draw the stack, so that you'll know where params and variables are
3. Comment, comment, comment

And in our interpreter:

- use breakpoints
- write small pieces of code and test them
- test with small amounts of data

I ❤️ RISC-V