Instruction Set Architecture (ISA)

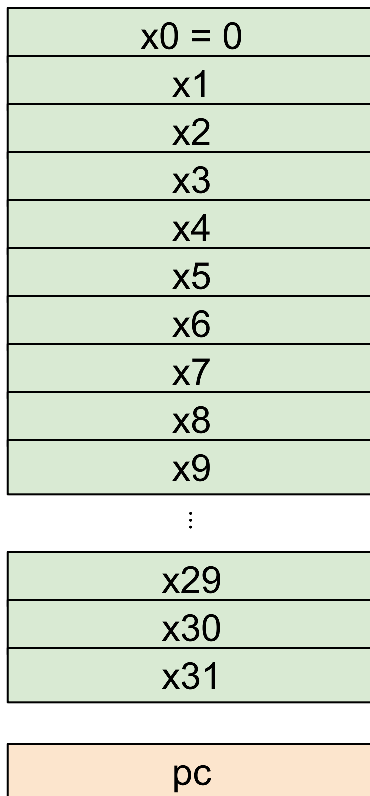
Encoding of instructions raises some interesting choices...

- Trade Offs: performance, compactness, programmability
- Uniformity. Should different instructions
  - Be the same size (number of bits)?
  - Take the same amount of time to execute?
  - Trend: Uniformity. Affords simplicity, speed, pipelining.
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
  - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed
- Mix of Engineering & Art...

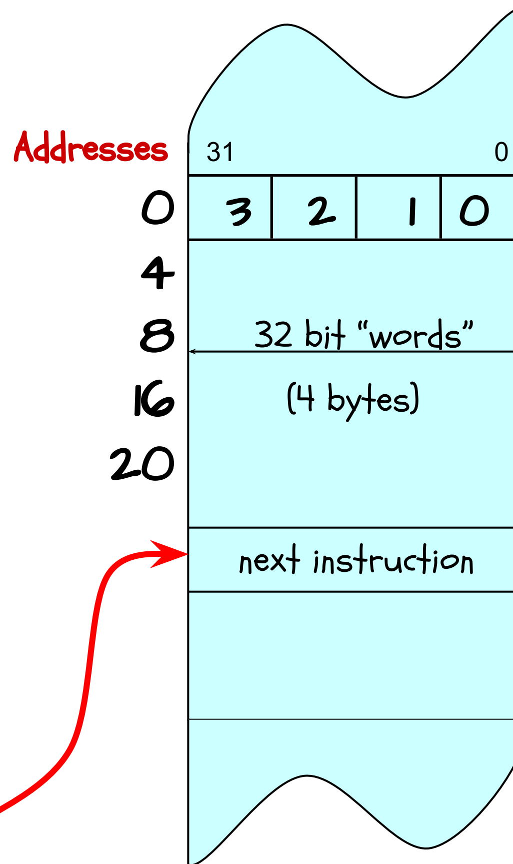
RISC-V Programming Model

A representative RISC machine

Processor State (inside the CPU)



Main Memory



In Comp 311 we'll use a subset of the RISC-V core Instruction set as an example ISA (RV32I).

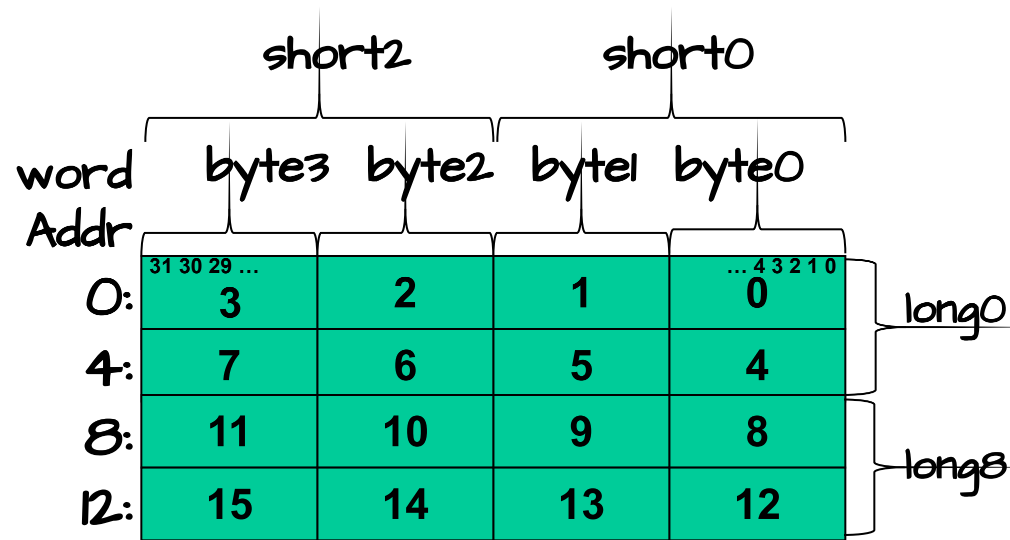
Fetch/Execute loop:

- fetch Mem[PC]
- execute fetched instruction (may change PC!)
- PC = PC + 4
- repeat!

RISC-V uses BYTE memory addresses. However, each instruction is 32-bits wide.. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

RISC-V Memory Nits

- Memory locations are addressable in different sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/doubles)
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (RISC-V is a byte-addressable machine)
- Most instructions are one word



Concocting an Instruction Set

Nerd Chef at work.

```
move    flour,bowl
add     milk,bowl
add     egg,bowl
move    bowl,mixer
rotate  mixer
...
```

First Problem Set is Posted

RISC-V Register Nits

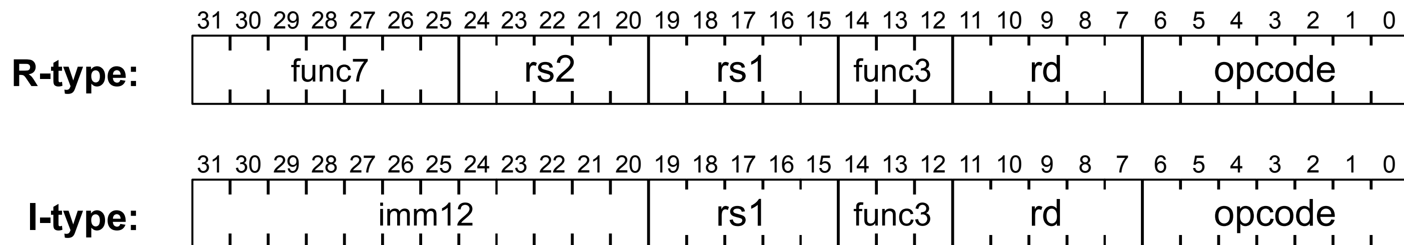
- There are 32 named registers [x0, x1, ... x31]
- x0 is special. It always contains "0" and, when used as a destination, the result is ignored
- The operands of most instructions are registers
- This means to operate on a variables in memory you must:
  - Load the value/values from memory into a register
  - Perform the instruction
  - Store the result back into memory
- Going to and from memory can be expensive (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- By convention most registers are dedicated to specific tasks

A.K.A a "Load-Store Architecture"

Basic RISC-V Instructions

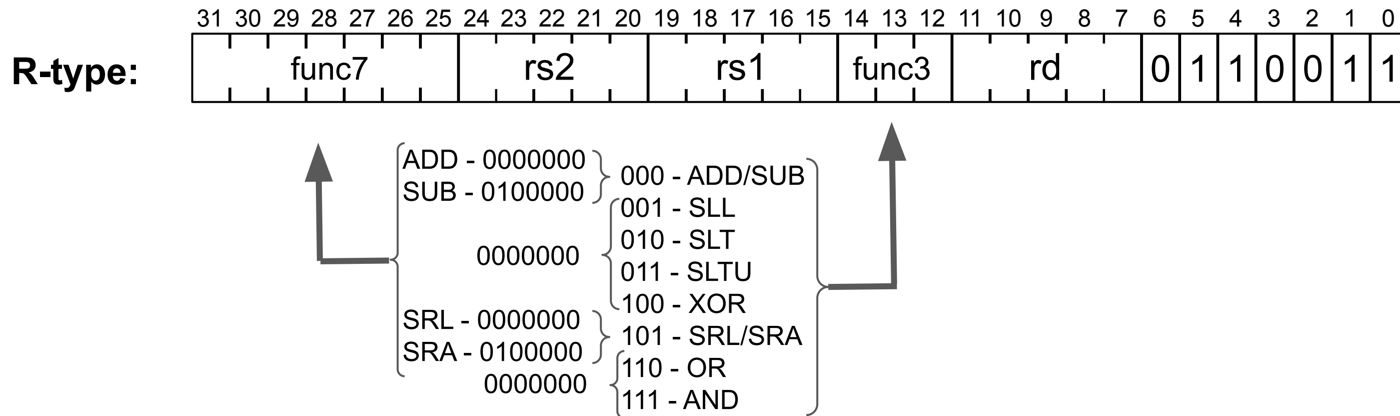
- Instructions include various "fields" that encode combinations of OPCODES and arguments
- special fields enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants ("immediate" values) of various sizes,

The basic data-processing instruction formats:



R-type Data Processing

Instructions that process three-register arguments:



add x1, x2, x3

R-type instructions have the following template:

```
OPfunc3   rd,rs1,rs2
```

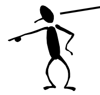
Is encoded as:

0000 0000 0011 0001 0000 0000 1011 0011

0x00310b3

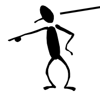
Arithmetic Instructions

```
add x5,x6,x7
```



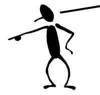
x5 ← x6 + x7
Registers can contain either 32-bit unsigned values or 32-bit 2's-complement signed values.

```
sub x6,x7,x28
```



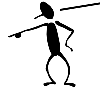
x6 ← x7 - x28
Once more, either 32-bit unsigned values or 32-bit 2's-complement signed values.

```
mul x28,x5,x6
```



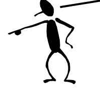
x28 ← x5 * x6
Register contents are treated as signed-values and multiplied together. The lower 32-bits of the result are saved in the destination.

```
div x7,x28,x6
```



x7 ← x28 / x6
The first source register is divided by the second. The result is saved in the destination. A divisor of 0 sets the destination to all 1's

```
rem x6,x28,x6
```



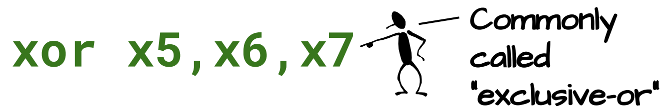
x6 ← x28 % x6
The remainder left after dividing by the first operand by the second is stored in the destination. A divisor of 0 sets the result to the dividend.

Recall that the results of arithmetic operations can overflow, or in some cases aren't even possible, such as dividing by 0. These RISC-V instructions act exactly like the C-language operators. A user must write code that detects the overflow condition. Just as they need to do in C.

Logic Instructions

Logical operations on words operate "bitwise", that is they are applied to corresponding bits of both source operands.

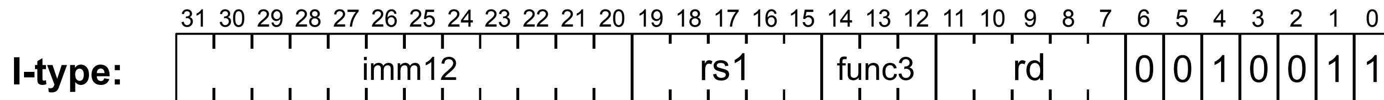
```
and x5,x6,x7

or  x5,x6,x7
```



x6:	0000 0000 0000 0000 1111 1111 0000 0000
x7:	0000 0000 0000 0000 1111 0000 1111 0000
x5:	0000 0000 0000 0000 1111 0000 0000 0000
x5:	0000 0000 0000 0000 1111 1111 1111 0000
x5:	0000 0000 0000 0000 0000 1111 1111 0000

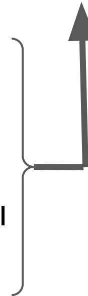
I-type Data Processing

Instructions that process one register and a constant:



Notice there is no SUBI instruction. Why?

000 - ADDI
001 - SLLI
010 - SLTI
011 - SLTUI
100 - XORI
101 - SRLI/SRAI
110 - ORI
111 - ANDI



I-type instructions have the following template:

```
OPfunc3   rd,rs1,imm12
```

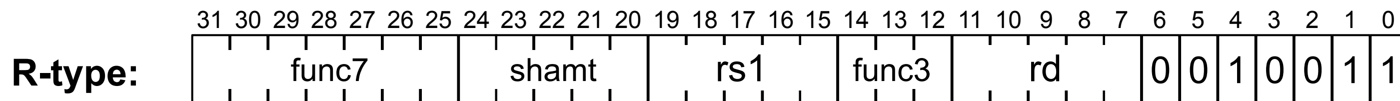
```
andi  x5,x10,255
```

Is encoded as:

```
0x0ff57293
```

Shifty Shift Immediates

RISC-V provides only 12-bits for specifying an immediate constant value. The value is consistently treated as a signed 2s-complement number, thus providing an immediate range of [-2048, 2047]. Shifts, are an ***exception*** to this rule. Shifts (slli, srli, srai) are limited to the range [0,31], and this limited range is used to encode the difference between srli and srai.



001 - SLLI

SRLI - 0000000
SRAI - 0100000
101

What is the difference between SRLI and SRAI?
Hint: x = a >>> 16 vs x = a >> 16

Why no SLAI?

```
srai  x6,x5,16
```

Is encoded as:

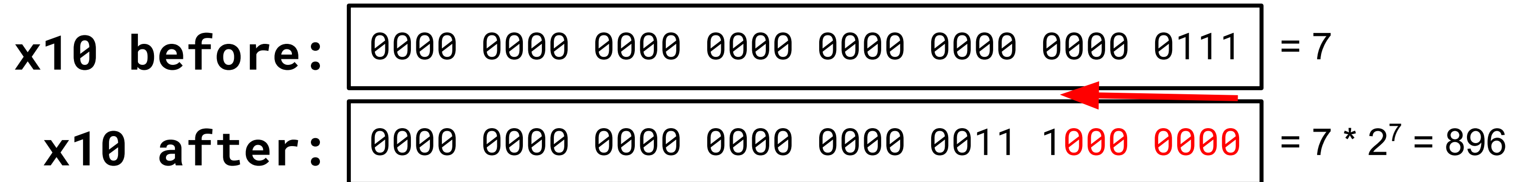
0100 0001 0000 0010 1101 0011 0001 0011

```
0x4102d313
```

Left Shifts

Left shifts effectively multiply the contents of a register by 2^s where s is the shift amount.

```
slli  x10,x10,7   # 0x00751513
```



Right Shifts

Right Shifts behave like ***dividing*** the contents of a register by 2^s where s is the shift amount, ***if*** you assume the contents of the register are ***unsigned***.

```
srli x11,x11,2
```

x11 before:	0000 0000 0000 0000 0000 0100 0000 0000	= 1024
x11 after:	0000 0000 0000 0000 0000 0001 0000 0000	= 1024 / 2^2 = 256

Arithmetic Right Shifts

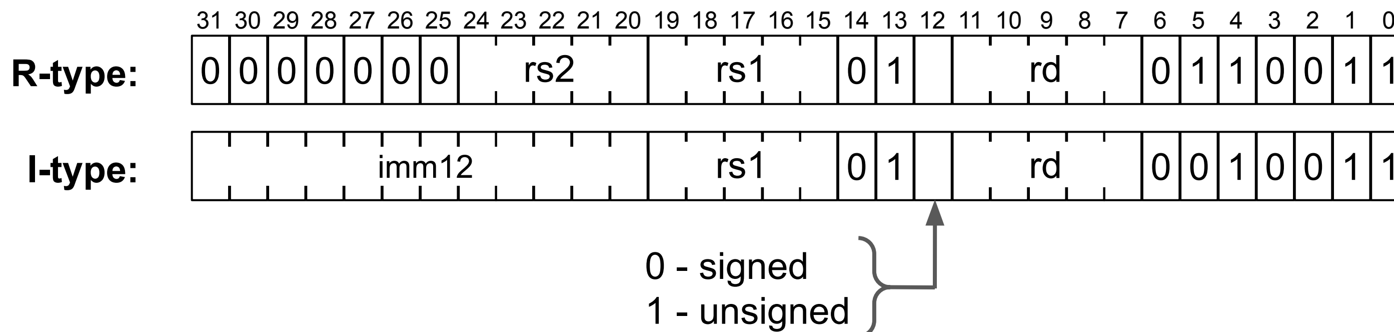
Arithmetic right Shifts behave like ***dividing*** the contents of a register by 2^s where s is the shift amount, ***if*** you assume the contents of the register are ***signed***.

```
srai  x10,x10,2
```

x10 before:	1111 1111 1111 1111 1111 1100 0000 0000	= -1024
x10 after:	1111 1111 1111 1111 1111 1111 0000 0000	= $-1024 / 2^2 = -256$

Comparison Instructions

- RISC-V has one basic comparison instruction:
  Set If Less Than that comes in 4 variations
  - SLT set if less than; R-type
  - SLTU set if less than "unsigned"; R-type
  - SLTI set if less than immediate; I-type
  - SLTUI set if less than "unsigned immediate; I-type
- Sets rd to '1' if the contents of rs1 is less than the contents of the second operand and to '0' otherwise.



Missing Comparisons

Using SLT and SLU you can create many other comparisons, such as the examples below:

Comparison	Instruction
Set rd if rs < 0	slt rd,rs,x0
Set rd if rs > 0	slt rd,x0,rs
Set rd if rs <> 0	sltu rd,x0,rs
Set rd if rs >= rt	slt rd,rt,rs

Comparisons are used to evaluate "conditional expressions" such as the test of an if statement or a while loop.

Next Time

- We will examine more instruction types and capabilities
  - Branching
  - Jump and Link
  - Loading from and storing to memory
  - Special instructions