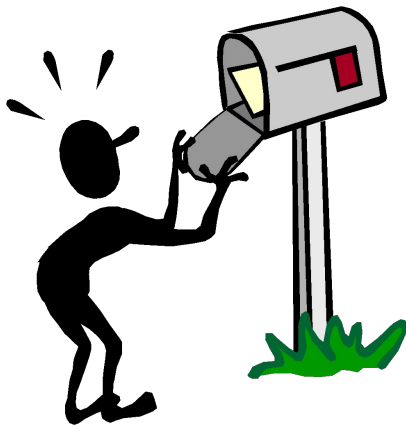


ADDRESSING MODES AND BRANCHES

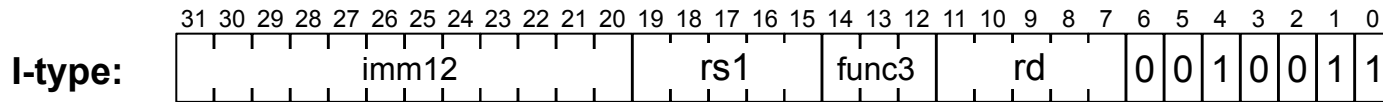


- More on Immediates
- Reading and Writing Memory
- Registers holding addresses
- Pointers
- Changing the PC
 - Loops
 - Labels
 - Calling Functions



WHY BUILT-IN CONSTANT OPERANDS?

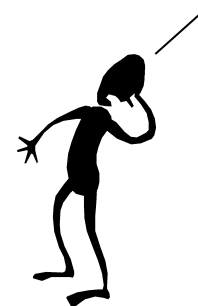
(IMMEDIATES)



- Alternatives? Why not? Do we have a choice?
 - put constants in memory (was common in older ISAs)
- SMALL constants are used frequently (50% of operands)
 - In a C compiler (gcc) 52% of ALU operations involve a constant
 - In a circuit simulator (spice) 69% involve constants
 - e.g, $B = B + 1$; $C = W \& 0xff$; $A = B - 1$;

- ISA Design Principle:
 - Make the common case easy*
 - Make the common case fast*

How large of constants should we allow for? If they are too big, we won't have enough bits leftover for the instructions or operands.





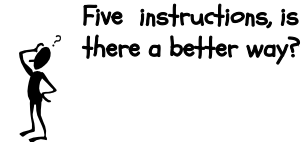
BIGGER CONSTANTS

We can load any 32-bit constant using a series of instructions

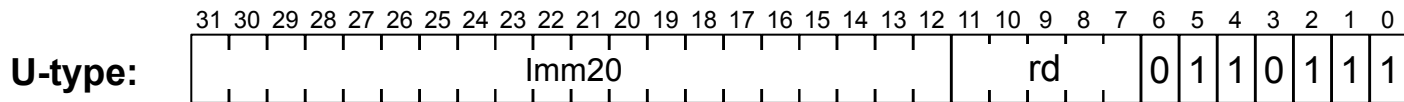
```

addi    x10, x0, 0x123
slli    x10, x10, 12
addi    x10, x10, 0x456
slli    x10, x10, 12
addi    x10, x10, 0x78

```



But there there is a special instruction for constructing large constants., called "Load Upper Immediate" or **lui**. lui uses a new instruction format call the U-type. lui can be used as part of a two-instruction sequence to construct any 32-bit constant



```

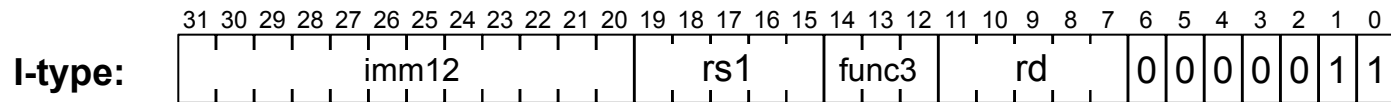
luix10, 0x12345 # encoded as 0x12345537

```



LOAD AND STORE INSTRUCTIONS

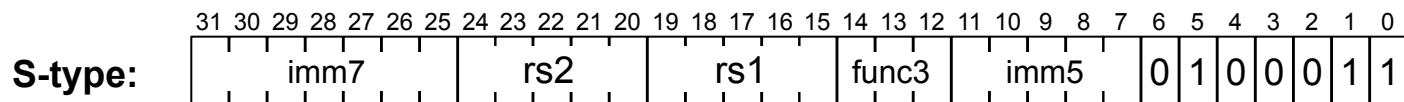
RISC-V is a "Load/Store architecture". That means that only a specific class of instructions are used to reference data in memory. As a rule, data is loaded into registers first, then processed, and the results are written back using stores. Load and Store instructions have their own format:



- 000 - LB
- 001 - LH
- 010 - LW
- 100 - LBU
- 101 - LHU



There are five types of loads from memory. Load a word, a half-word, or a byte. Bytes and half-words can be either signed or unsigned.



- 000 - SB
- 001 - SH
- 010 - SW







There are three types of stores to memory. Store a word, a half-word, or a byte. There is no need to distinguish signed and unsigned.



LOAD AND STORE OPTIONS

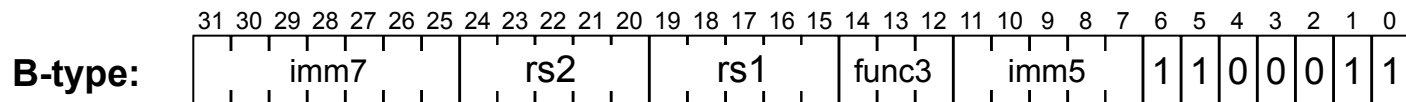
RISC-V's load and store instructions are versatile. They provide a wide range of addressing modes. Only a subset is shown here.

- `lw rd, imm12(rs)`  $Rd \leftarrow \text{Memory}[Rs + \text{imm12}]$
 Rd is loaded with the contents of memory at the address found by adding the contents of the base register, Rs , to the supplied constant
- `lb rd, -4(rs)`  $Rd \leftarrow \text{sign_extend}(\text{Memory}[Rs - 4])$
Offsets can be either added or subtracted, as indicated by a negative sign. The byte is signed-extend to fill the 32 bits of
- `lw rd, (rs)`  If no offset is specified it is assumed to be zero
- `sw rd, 12(rs)`  The contents of a register hold the address of the either the data value or a base address for a composite type (structure, object, array, or a stack)
- `sh rd, (rs)`



CHANGING THE PC

The Program Counter, or PC, is a special register that points to the address of the next instruction to be fetched. There are special instructions for changing the PC. One type is Branching Instructions. Branches are to nearby place within +/- 512 instructions away.




- 000 - BEQ
- 001 - BNE
- 100 - BLT
- 101 - BGE
- 110 - BLTU
- 111 - BGEU




There are six types of branching instructions. All compare the the contents of two registers. If the comparison is true then the next instruction will be from PC + imm12, if false PC+4

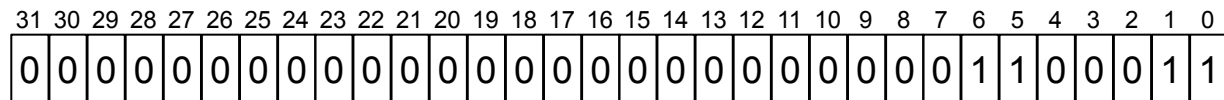


BRANCH EXAMPLES

bne x10, x0, else  If the contents of x10 is not equal to 0 then branch to the nearby address with the label "else"

blt x10, x0, neg  If the contents of x10 is less than 0 then branch to the nearby address with the label "neg"

loop: beq x0, x0, loop  An infinite loop.



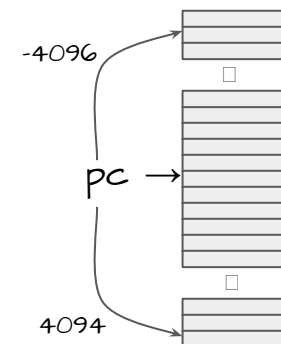
 beq x0,x0's encoding.



PC RELATIVE OFFSETS

All branch offsets in RISC-V are added to the PC to determine the target address of the next instruction in the case of a valid condition. Earlier ISAs would instead specify the "absolute" target address. What are the advantages of "relative" vs "absolute"?

- Does not implicitly limit the address space
- Requires fewer instruction bits, supports the most common cases
- Allows for "relocatable" code





A SIMPLE PROGRAM

```
# Assembly code for
# sum = 0;
# for (i = 0; i <= 10; i++)
#     sum = sum + i;
        addi    x7,x0,11      # x7 is 10 + 1
        addi    x5,x0,0      # x5 is i
        addi    x6,x0,0      # x6 is sum
loop:   add     x6,x6,x5      # sum = sum + i
        addi    x5,x5,1      # i++
        blt     x5,x7,loop
halt:   beq     x0,x0,halt
```

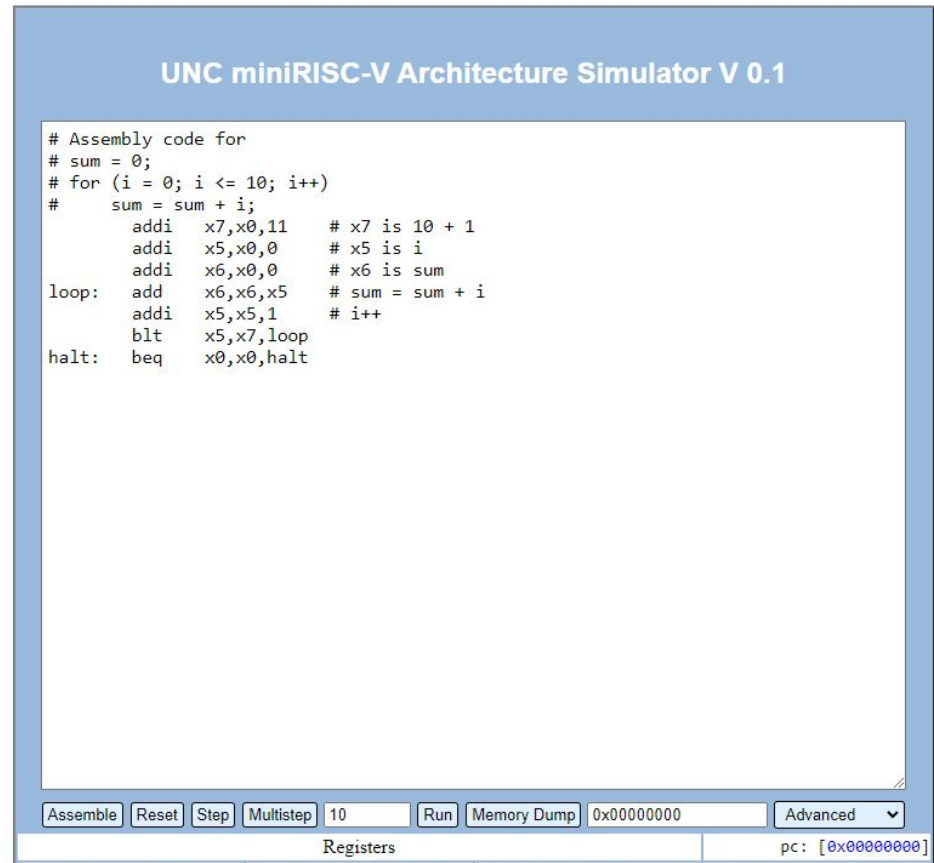
MEET THE MINIRISCV SIMULATOR



Located at:

<https://csbio.unc.edu/mcmillan/miniRISCV.html>

Or click the link in the "RESOURCES" section of the course website



```
# Assembly code for
# sum = 0;
# for (i = 0; i <= 10; i++)
#   sum = sum + i;
    addi x7,x0,11 # x7 is 10 + 1
    addi x5,x0,0  # x5 is i
    addi x6,x0,0  # x6 is sum
loop: add x6,x6,x5 # sum = sum + i
      addi x5,x5,1 # i++
      blt x5,x7,loop
halt: beq x0,x0,halt
```

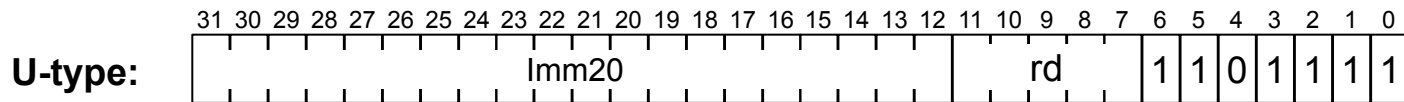
Assemble Reset Step Multistep 10 Run Memory Dump 0x00000000 Advanced ▾

Registers pc: [0x00000000]



JUMPING LONG DISTANCES

There are two more instructions for jumping long distances. Both are "unconditional", meaning that the branch is always taken, but they have another interesting feature



JAL: jump and link

Syntax: **jal rd,imm20**

Description:



jalr can be used to jump to an absolute location by first loading the address into a register

$\text{Reg}[d] \leftarrow \text{PC} + 4$

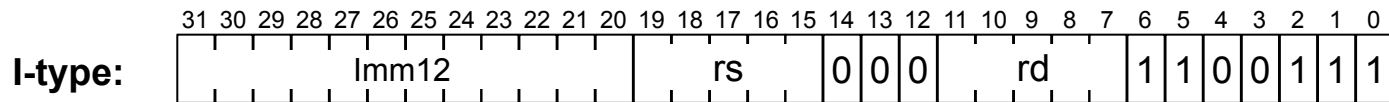
$\text{PC} \leftarrow \text{PC} + \text{sign_extended}(\text{imm20})$

Write the address of the following instruction, $\text{PC} + 4$, into $\text{Reg}[d]$, then jump to the instruction that is found by adding the current PC to the signed immediate offset. In practice this is usually specified by a label.

JUMPING LONG DISTANCES, ABSOLUTELY



There are two more instructions for jumping long distances. Both are "unconditional", meaning that the branch is always taken, but they have another interesting feature



JALR: jump and link register

Syntax: `jalr rd,imm12(rs)`
`jalr rd,(rs)`

Description:

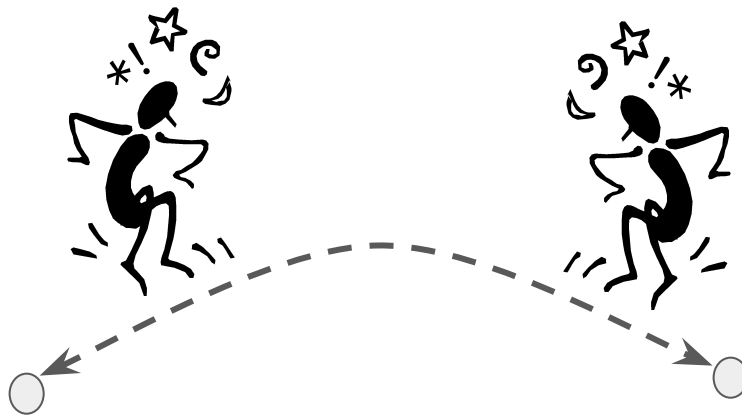
$\text{Reg}[d] \leftarrow \text{PC} + 4$
 $\text{PC} \leftarrow \text{Reg}[s] + \text{sign_extended}(\text{imm12})$

Jump to the instruction given by the contents of *Reg[s]*, and save the location of the next instruction in *Reg[s]*.



WHY STORE PC + 4?

These long-distance "jump" instructions also save the address of the following instruction in a register specified by Rd . Often, this register will be $x0$, and therefore is ignored. But in other cases it is useful to get back to where we jumped from. More about this next lecture.





NEXT TIME

We'll write more Assembly programs

Still some loose ends

- Multiplication? Division? Floating point? Relocatable data

