RISC (Reduced Instruction Set Computer)

- A single instruction can only perform one operation.
- Keep isa small as possible, makes it easier to build fast hard ware.

RISC-V ISA

Registers

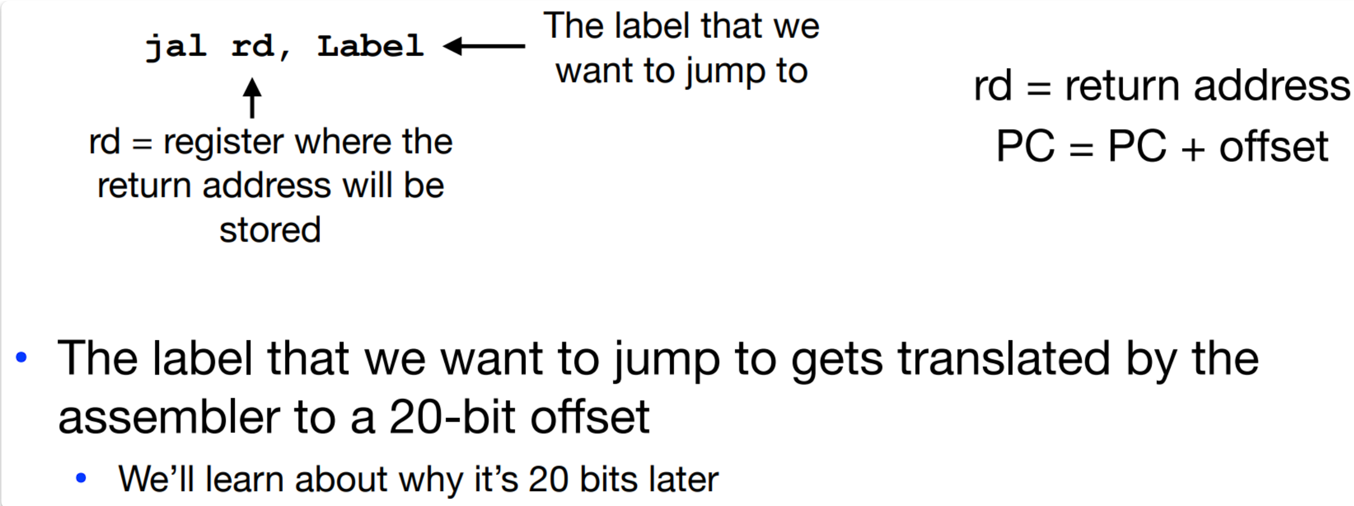
SUMMARY

32 general propose registers in rv
Referred to as x0-x31.
Associated with the word's length 32-64-128 bits.
x0 is always set to be zero.
PC is a register that holds the memory address of the instruction being executed.

WHAT IF WE WANT PC TO EXECUTE A FUNCTION AT A DIFFERENT LOCATION?

- Store the return address
- Update the value of the PC.
- Store values in registers.

JAL (jump and link)



We can choose **any** register to hold the return address.

- Usually utilize x1 to hold **return address** , so it has an alternate name **ra** .
- When we jump because of a loop or branch, we don't need a return address. For example, if a if-statement followed by else, when the instructions belong to if is finished, then jump over else without return address.
- To avoid saving the return address, we can specify x0 as the destination register.

```
jal x0,L1
```

and pseudo instruction for that is

```
j L1
```

JALR (jump and link register)

```
![[Pasted image 20220828112933.png]]
```

When we want to return from a function, the **only** thing we need to do is modifying PC's value.

```
jalr x0,rs,0
```

and pseudo instruction for that is

```
jr rs
```

If the register `ra` which contains return address, then the pseudo instruction can be simplified as

```
ret
```

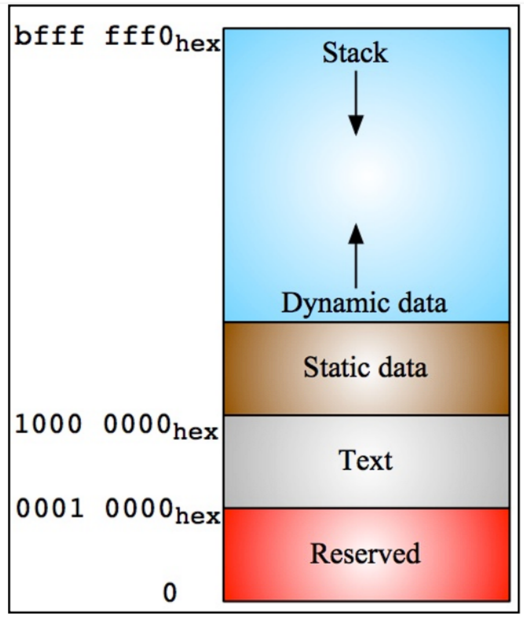
When we call another function, what happens to the value that are stored in the registers?

- We use the stack to store the info.

Stack

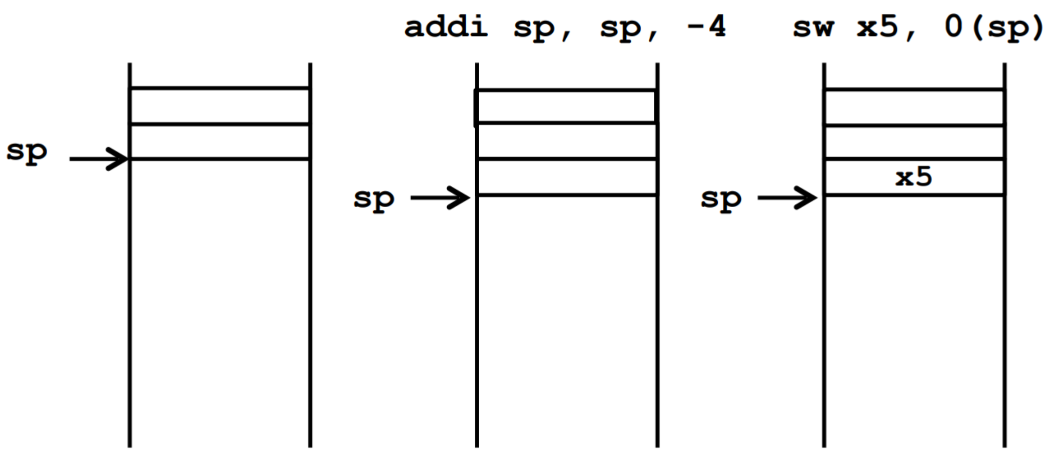
Stack Pointer(SP):A register that holds the memory address of the location of the last item placed on the stack (x2).

- When you place an item on the stack, you decrement the stack pointer
  - PUSH
- When you take an item off the stack, you increment the stack pointer
  - POP



How to Store a Value on the Stack

- If register x5 contains the data that we want to store on the stack



Saving registers

- We can save all of our registers before we call a function
  - All registers would be saved by the caller
- Another thing we can do is save all the registers before we use them
  - All registers would be saved by the callee
- Need to standardize how we do this
  - Meet somewhere in the middle, I'll save some and you save some
  - The registers that are saved by the caller and callee are specified by the ***calling convention***

Calling Convention

- Temoporary registers
  - Saved by caller.
- Saved registers

  * Saved by callee.

Register	Name	Description	Saved by
x0	**zero**	Always Zero	N/A
x1	**ra**	Return Address	Caller
x2	**sp**	Stack Pointer	Callee
x5-7	**t0-2**	Temporaries	Caller
x8-x9	**s0-s1**	Saved Registers	Callee
x18-27	**s2-11**	Saved Registers	Callee
x28-31	**t3-6**	Temporaries	Caller

Arguments and return registers

- Our functions need to have a place where they can expect the arguments and return values to be
- We will set aside registers **`x10-x17`** to be argument registers
  - New names => **`a0-a7`**
  - **`a0`** and **`a1`** will also serve as return value registers
- If the caller has some temporary values in the registers that it wants to use after making a function call, it must save those values

Register	Name	Description	Saved by
x0	zero	Always Zero	N/A
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	N/A
x4	tp	Thread Pointer	N/A
x5-7	t0-2	Temporary	Caller
x8-x9	s0-s1	Saved Registers	Callee
x10-x17	a0-7	Function Arguments/Return Values	Caller
x18-27	s2-11	Saved Registers	Callee
x28-31	t3-6	Temporaries	Caller

Calling a Function

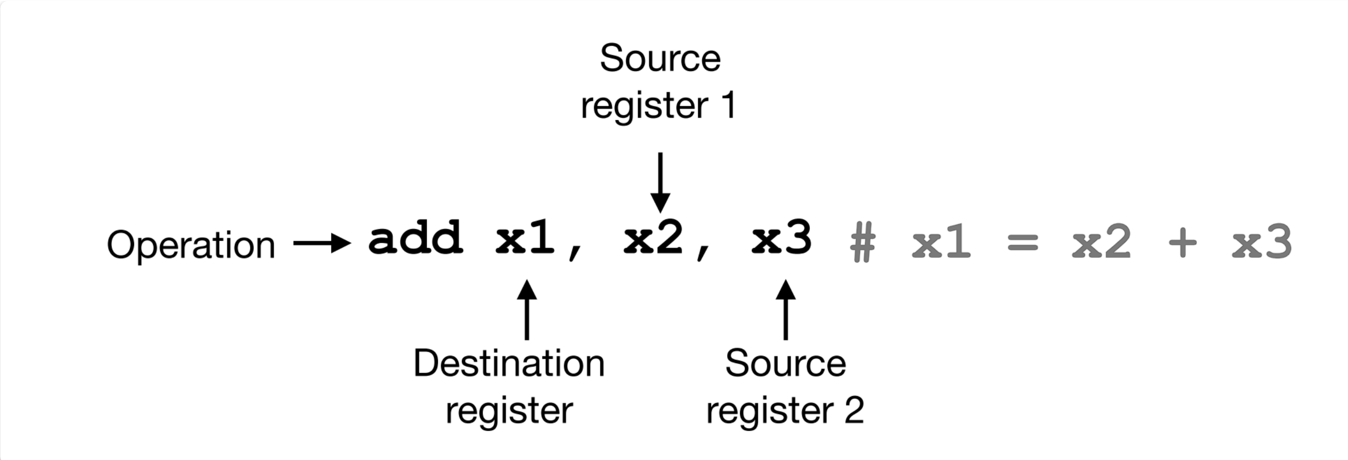
1. Put parameters in a place where function can access them
   - Put parameters in argument registers
2. Transfer control to function
   - With a jump instruction
3. Acquire (local) storage resources needed for function
   - Make room for local variables on stack
4. Perform desired task of the function
5. Put result value in a place where calling code can access it
   - `a0-a1` register
6. Return control to point of origin
   - `ret`

Comments in Assembly

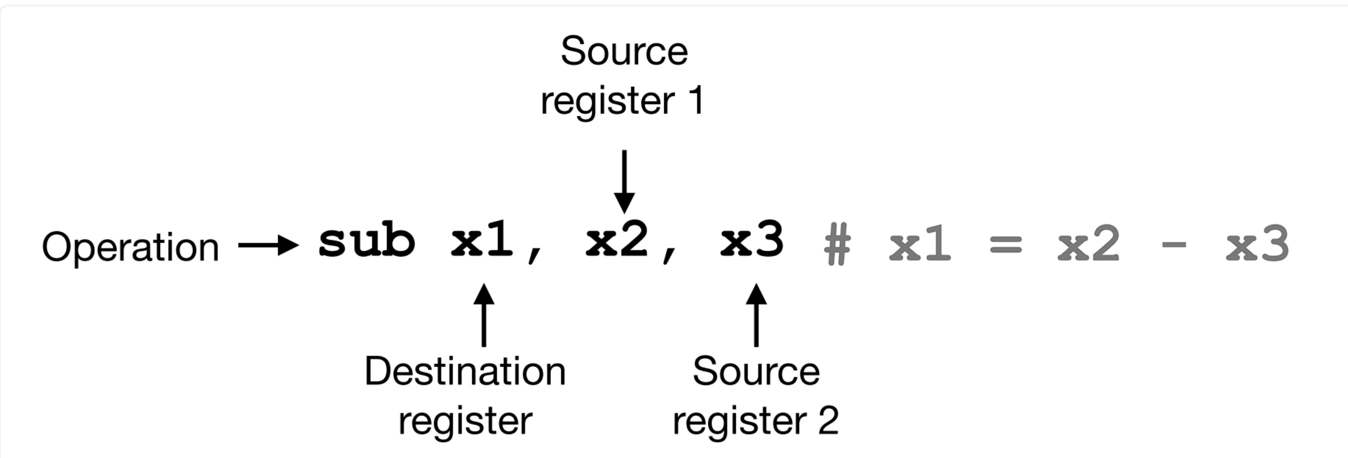
```
add x1,x2,x3 # x1=x2+x3
```

Instructions in RV

ADDITON



SUBTRACTION



IMMEDIATES

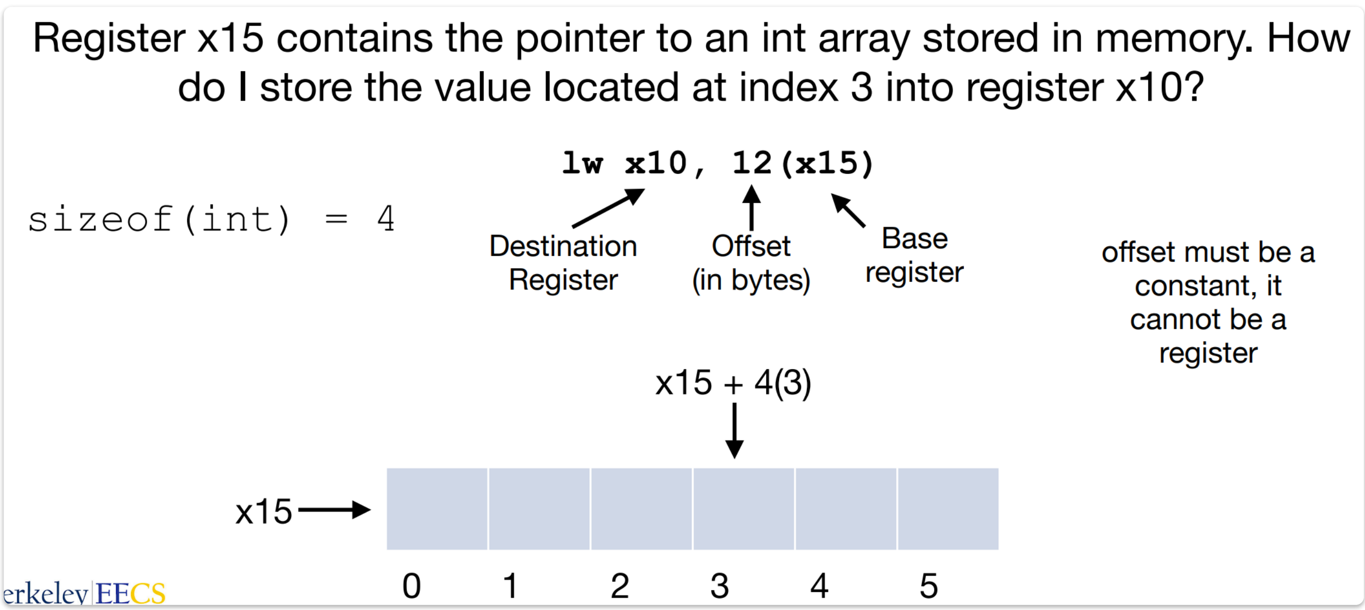
- Immediates are used to provide numerical constants
- Constants appear often in code, so there are special instructions for them:
- Ex: Add Immediate:

```
f = g - 10           (in C)
addi x3,x4,-10       (in RISC-V)
       ↑  ↑
       f  g
```

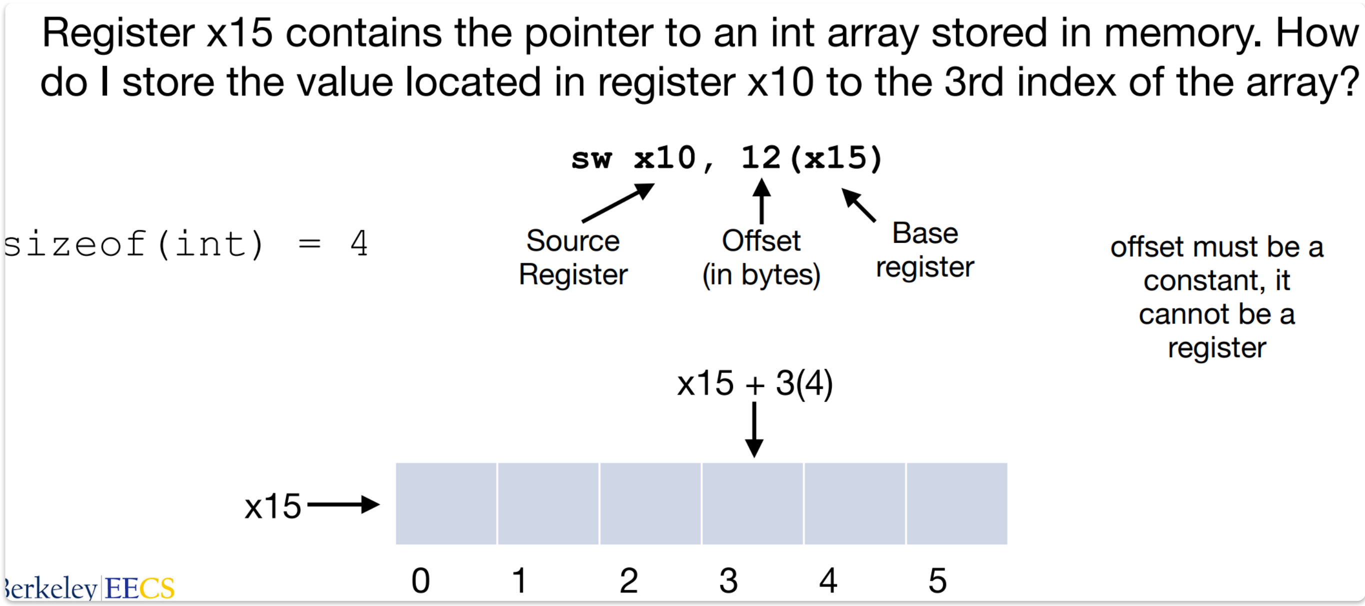
- There is no substract immediate in RV cause we can use addi to replace it.
- Addi immediates are limited to **12** bits.
- When you perform an operation with an immediate, it is **sign extended** to 32-bits.

MEMORY OPERATION

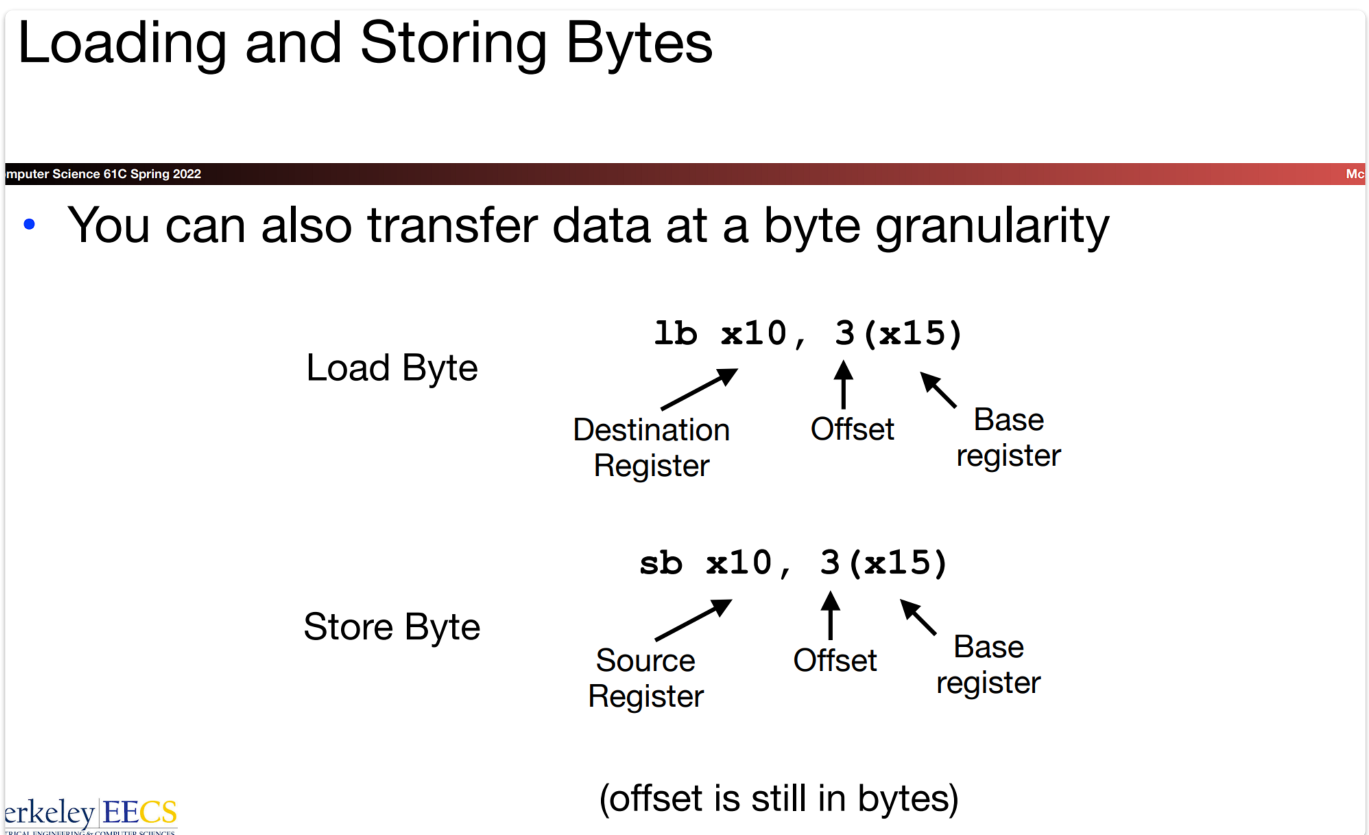
Load word(lw)



Store word



Loading and Storing bytes



- When you load a byte from mem, it is placed into the **lowest** byte of the destination register and **sign extended**.

- If you don't want the number to be sign extended , you can use `lbu` which will zero extend to fill the register.
- When you store a byte, only the lower 8 bits of the register is copied into mem, so there is no sign extension.

LOGICAL INSTRUCTIONS

Logical operations	C operators	Java operators	RISC-V instructions
Bitwise AND	&	&	**and**
Bitwise OR	|	|	**or**
Bitwise XOR	^	^	**xor**
Shift left logical	<<	<<	**sll**
Shift right	>>	>>	**srl/sra**

shifting

Computer Science 61C Spring 2022

- Shift by the contents of a register

```
sll x10, x11, x12 # x10 = x11 << x12
```

- Shift by a constant value

```
slli x10, x11, 2 # x10 = x11 << 2
```

If x10 contains 40	x10 =0b 0000 0000 0000 0000 0000 0000 0010 1000 = 40
`srli x11,x10,3`	x11 = 0b 0000 0000 0000 0000 0000 0000 0000 0101 = 5
If x10 contains 41	x10 = 0b 0000 0000 0000 0000 0000 0000 0010 1001 = 41
`srai x11,x10,3`	x11 = 0b 0000 0000 0000 0000 0000 0000 0000 0101 = 5
If x10 contains -32	x10 = 0b 1111 1111 1111 1111 1111 1111 1110 0000 = -32
`srai x12,x10,4`	x12 = 0b 1111 1111 1111 1111 1111 1111 1111 1110 = -2
If x10 contains -25	x10 = 0b 1111 1111 1111 1111 1111 1111 1110 0111 = -25
`srai x12,x10,4`	x12 = 0b 1111 1111 1111 1111 1111 1111 1111 1110 = -2

- Right shifting positive numbers and even numbers is equivalent to dividing by 2^n with the fractional part of the result being truncated
- Right shifting negative odd numbers is equivalent to dividing by 2^n and rounding the result towards negative infinity
  - This is not the behavior that we want
  - C arithmetic semantics is that division should round towards 0

DECISION MAKING INSTRUCTIONS

Labels

A label tells where the program to go.

Conditional Branches

- Branch if equal
  - **`beq reg1, reg2, L1`**
  - If reg1 == reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```
if (a != b)
    e = c + d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
        beq x10,x11,Exit
        add x14,x13,x12
Exit:
```

Branch if not equal

- **`bne reg1, reg2, L1`**
- If reg1 != reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```
if (a == b)
    e = c + d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
        bne x10,x11,Exit
        add x14,x13,x12
Exit:
```

Branch on less than

- **`blt reg1, reg2, L1`**
- If reg1 < reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```
if (a >= b)
    e = c + d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
        blt x10,x11,Exit
        add x14,x13,x12
Exit:
```

Branch on greater than or equal

- **`bge reg1, reg2, L1`**
- If reg1 >= reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```
if (a < b)
    e = c + d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
      bge x10,x11,Exit
      add x14,x13,x12
Exit:
```

nputer Science 61C Spring 2022 McMahon a

- blt and bge perform signed comparisons of the numbers
- To perform unsigned comparisons, use bltu and bgeu
- RISC-V doesn't have "branch if greater than" or "branch if less than or equal". Instead you can reverse the arguments:
  - A > B is equivalent to B < A
  - A <= B is equivalent to B >= A

Unconditional bracnces

Computer Science 61C Spring 2022

- Jump
  - **`j label`**
  - Always jump to the code located at label

IF-ELSE

```
if (a == b)
     e = c + d;
else
    e = c - d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
       bne x10,x11,else
       add x14,x12,x13
       j done
else:  sub x14,x12,x13
done:
```

Loop

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];
```

Assume x8 holds the address of the array

```
       add x9,x8,x0    # x9=&A[0]
       add x10,x0,x0   # sum=0
       add x11,x0,x0   # i=0
       addi x13,x0,20  # x13=20
Loop:  bge x11,x13,Done
       lw x12,0(x9)     # x12=A[i]
       add x10,x10,x12  # sum+=A[i]
       addi x9,x9,4     # x9=&A[i+1]
       addi x11,x11,1   # i++
       j Loop
Done:
```