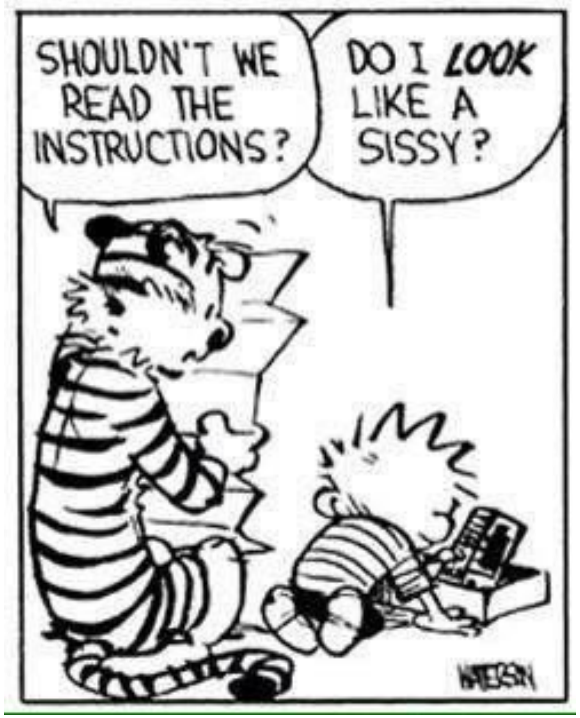
Some Assembly Required

- What's up with immediates
- Let's Compile
- Let's Optimize

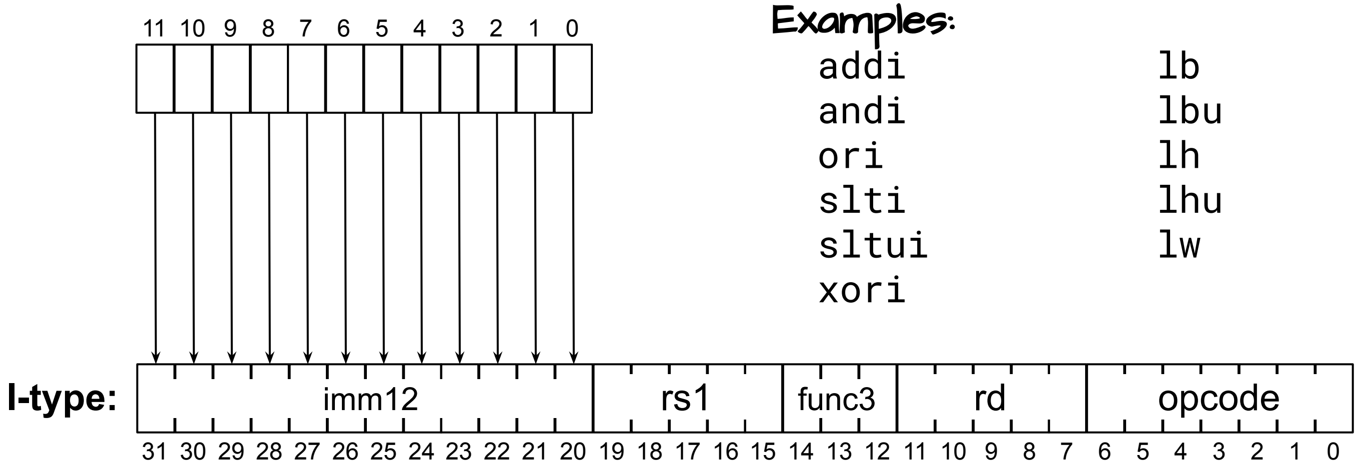


Midterm is one week from today!

Open book, open notes, open internet
(no communications however)

Immediate encodings

Many RISC-V instructions encode a constant value as part of the instruction. Unlike other ISAs, the encoding of constants is not uniform. The I-type instructions encode their immediate operand as follows:

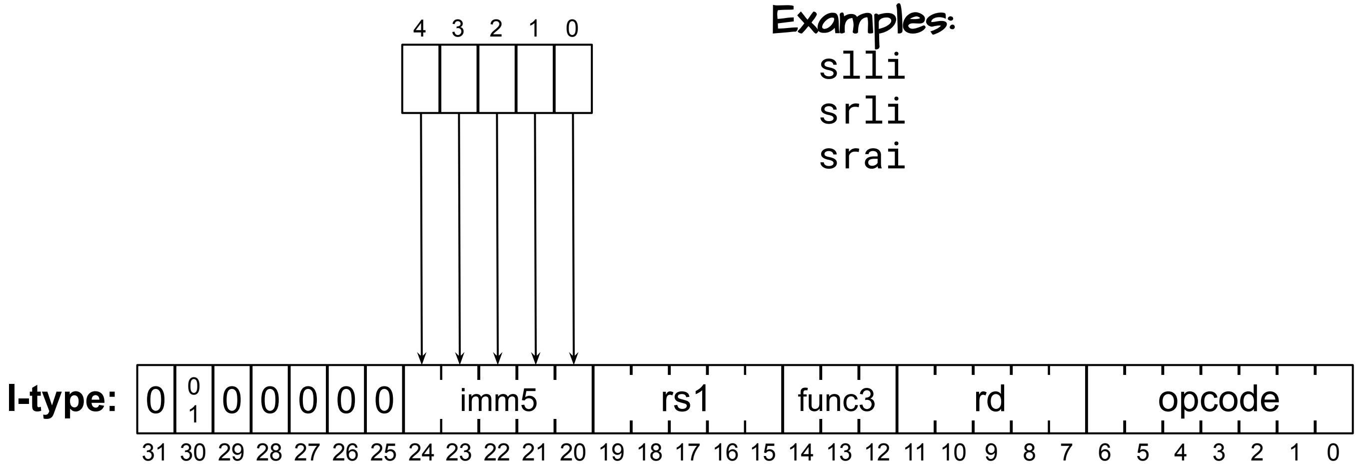


Examples:

```
addi     lb
andi     lbu
ori      lh
slti     lhu
sltui    lw
xori
```

Immediate encodings

Many RISC-V instructions encode a constant value as part of the instruction. Unlike other ISAs, the encoding of constants is not uniform. The I-type instructions encode their immediate operand as follows:



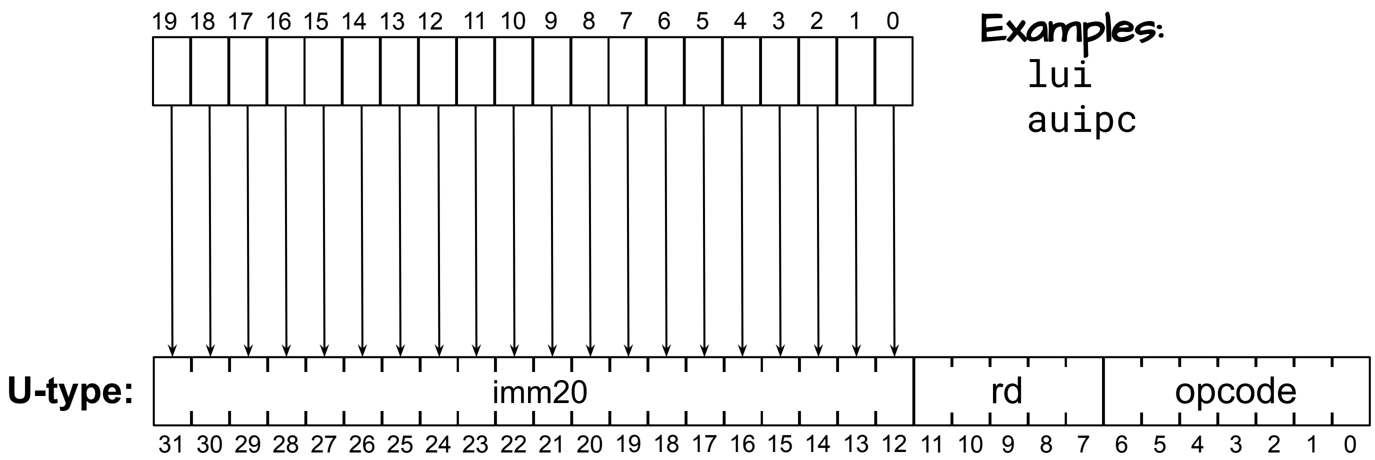
Examples:

```
slli
srli
srai
```

But there are exceptions!

LUI's Immediate value

The immediate-field encoding of `lui` and `auipc` also seems straightforward, but, as discussed last lecture, the immediate values might not be what you expect due to the sign-extension of its companion addi instruction.



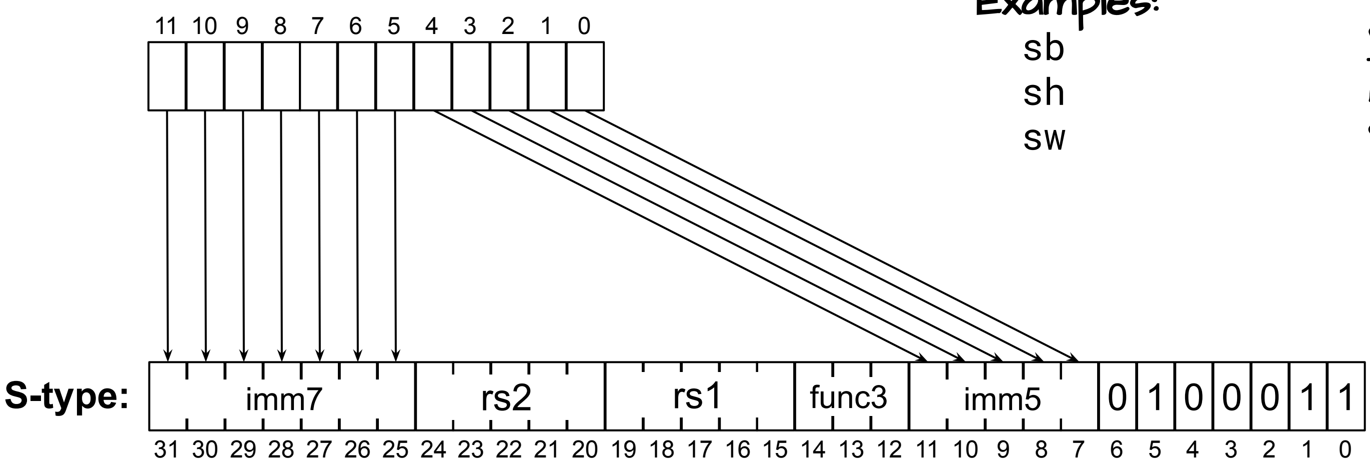
Examples:

```
lui
auipc
```

Remember if bit 11 of the large constant is set, then you add 1 to the LUI/AUIPC immediate value.

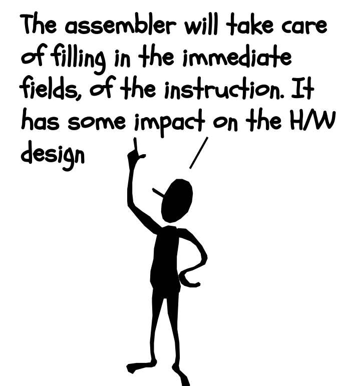
Immediate Offsets for stores

RISC-V store instructions include an offset, but it is not encoded in an obvious way. Note that there is no need for a rd register in a store, but both rs1 and rs2 are needed if the encodings are used consistently.



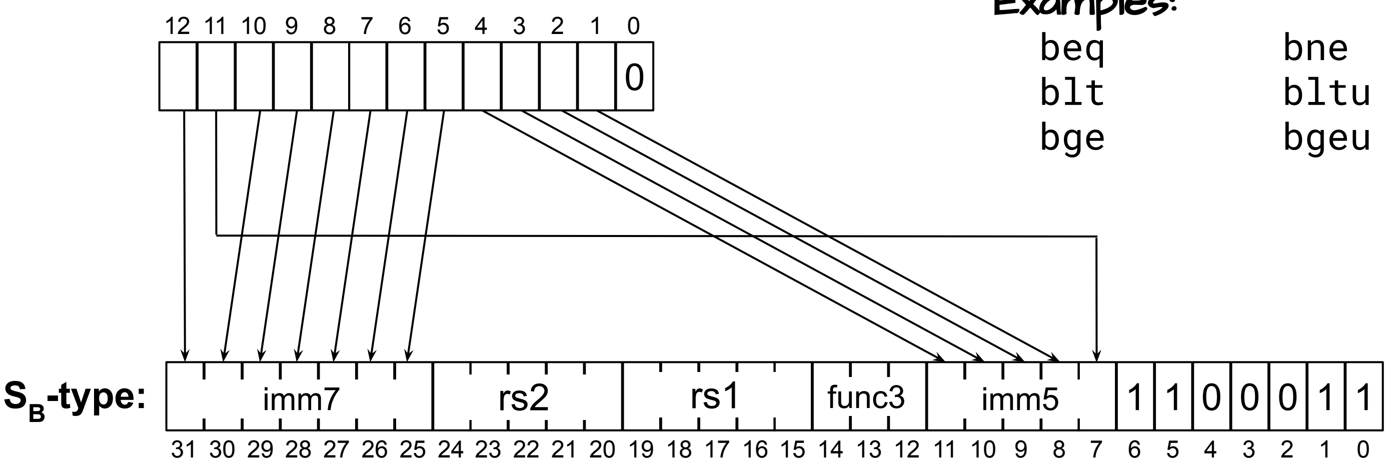
Examples:

sb
sh
sw



Immediate Offsets for Branches

The offsets of RISC-V branch instructions use the same format as a stores, but the ***offset encoding*** is non-obvious. Again, there is no need for a rd register; only rs1 and rs2 are used. Branch offsets can only be even. In fact, all instructions must be aligned to half-word boundaries.

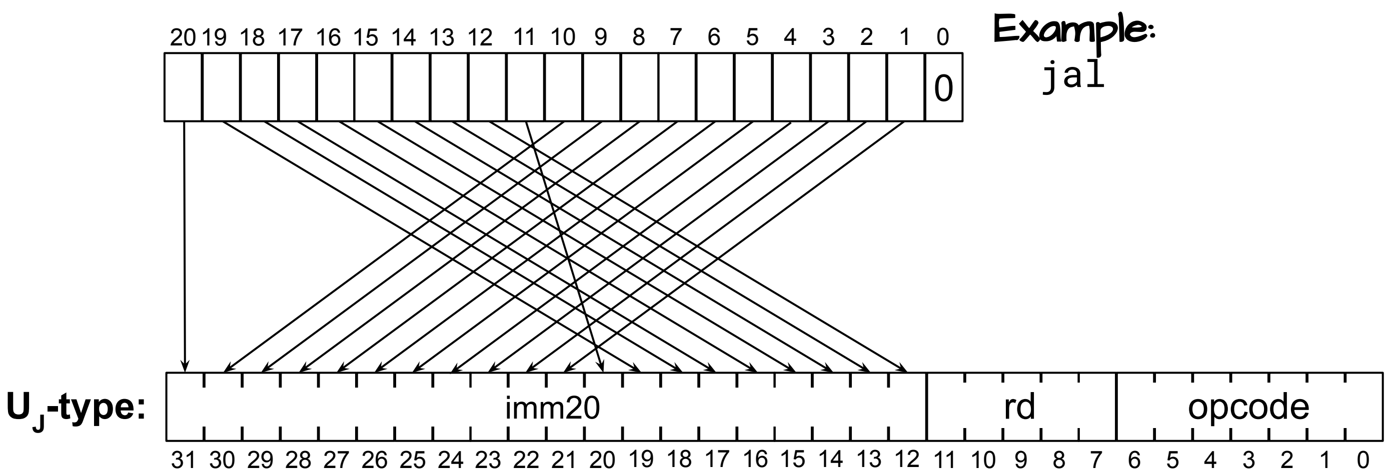


Examples:

```
beq     bne
blt     bltu
bge     bgeu
```

Crazy Jump offsets

The offsets of RISC-V jump instructions use the same U-format as `lui/auipc`, but the ***offset encoding*** is even stranger. Jump offsets, like branch offsets can only be even.

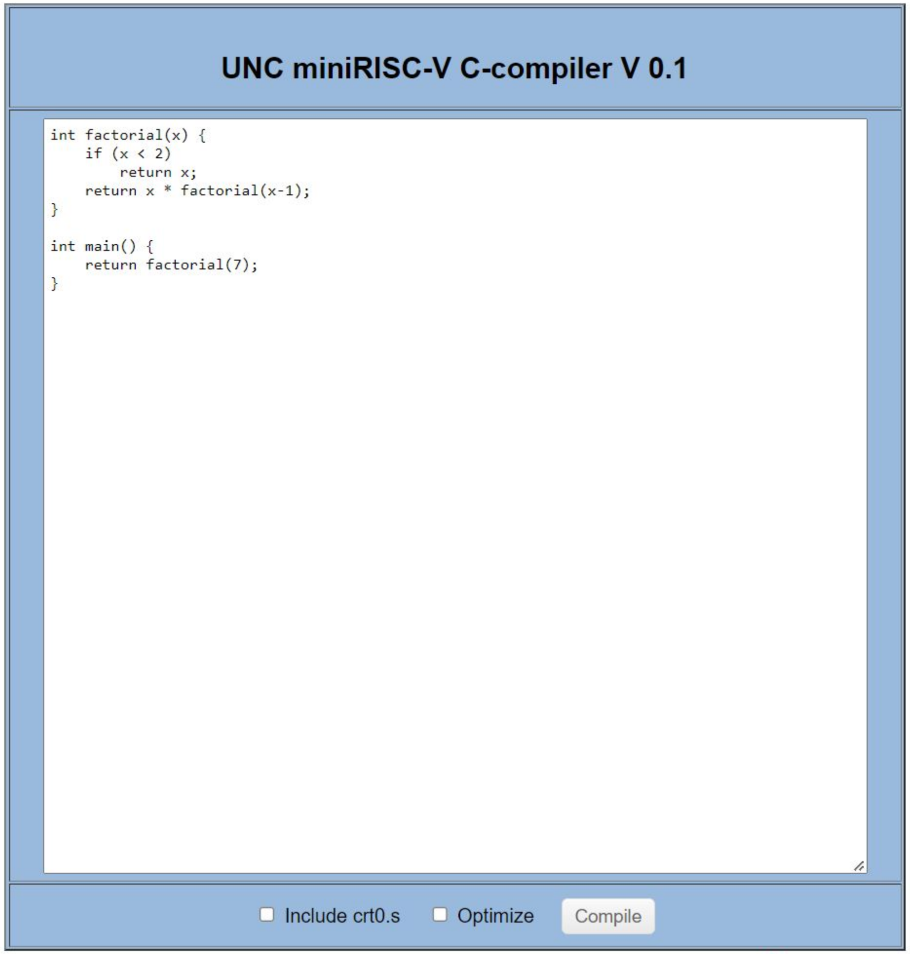


Why assemble? Compile

There are a lot of details involved when mapping an assembly language program to bits in memory. But, this mapping can be automated. Compilers provide a means for hiding the details of assembly language. Lets, look at a simple recursive program:

```
int factorial(x) {
    if (x < 2)
        return x;
    return x * factorial(x-1);
}

int main() {
    return factorial(7);
}
```

The miniRISC-V C-compiler



Goto
https://csbio.unc.edu/mcmillan/index.py?run=rv

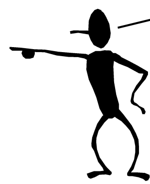
This compiler does nly code generation. It bypasses the linking stage,

The assembly code that it generates can be pasted directly into the simulator

Cut and paste your C-code and press [Compile]

The miniRISC-V C-Compiler

```
        .align 4
        .text
        .globl factorial
        .align  4
factorial:
        addi x2,x2,-64
        sw x8,60(x2)
        addi x8,x2,48
        sw x1,24(x2)
        sw x27,28(x2)
        mv x27,x12
        li x30,2
        bge x27,x30,L.2
        addi x10,x27,0
        jal x0,L.1
L.2:
        addi x12,x27,-1
        jal x1,factorial
        addi x30,x10,0
        mul x10,x27,x30
L.1:
        lw x1,24(x2)
        lw x27,28(x2)
        lw x8,60(x2)
        addi x2,x2,64
        jalr x0,x1,0
```



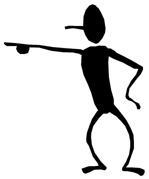
A larger than necessary stack frame with a "frame pointer".

```
        .globl main
        .align  4
main:
        addi x2,x2,-48
        sw x8,44(x2)
        addi x8,x2,32
        sw x1,24(x2)
        li x12,7
        jal x1,factorial
        addi x30,x10,0
L.4:
        lw x1,24(x2)
        lw x8,44(x2)
        addi x2,x2,48
        jalr x0,x1,0

        .align 4
```

While all the assembly code generated is valid. It does not provide any support for initializing before startup. Rerun the compiler with "include crt0.s" selected.

The C Runtime StartUP Code

```
reset:  lui     sp,0xc0000    # initialize stack pointer
        addi    sp,sp,0xff0
        jal     ra,main       # call main
*halt:  j       halt

##########################################
```

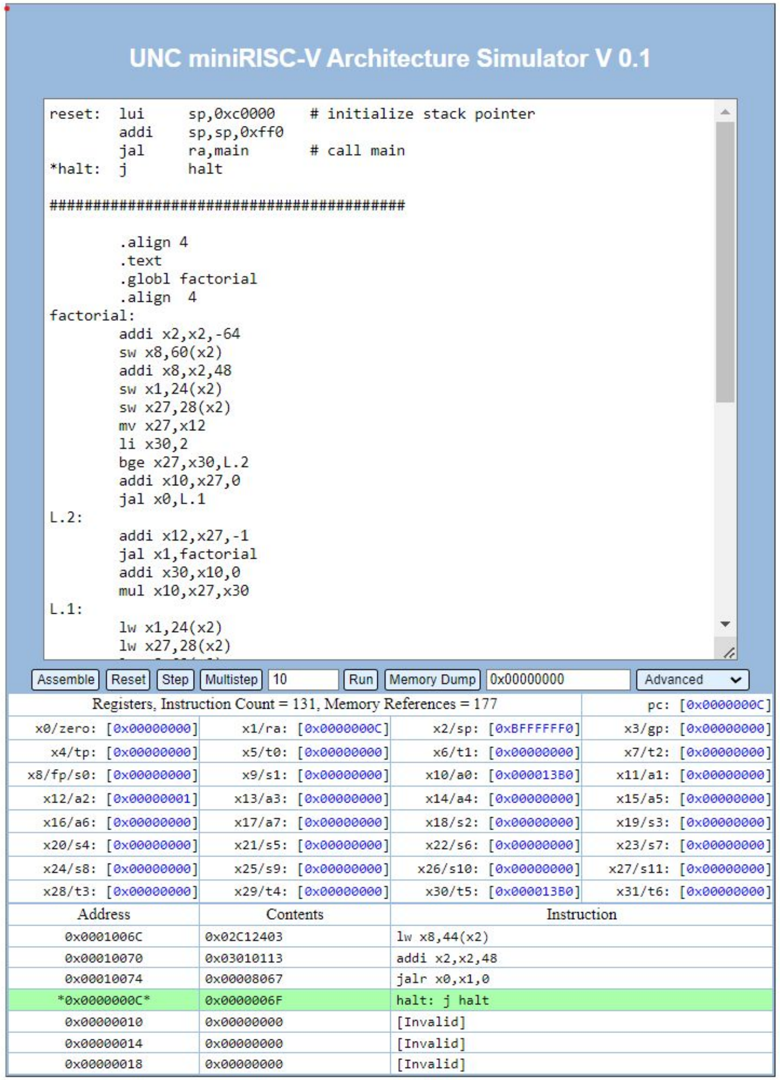


This small section of code initializes the stack pointer and calls the function "main".

This code is loaded into the .kernel memory section.

Note that the lui/addi involves a constant with bit 11 set.

Run it!



The code should assemble and run.

The simulator keeps track of the number of instructions executed. (131)

How could we count the number of calls to factorial?

Another Example

Counts ones:

```
int countOnes(unsigned int x) {
    if (x == 0)
       return x;
    return (x & 1) + countOnes(x>>1);
}

int main() {
     return countOnes(1023);
}
```

What does this function do?

Compile and test it.

How would you improve it?

- Speed?
- Size?

Next Time

We look into the hardware