Assembling the last few bits

- More Assembly Practice
- Multiplication
- Division
- Calling procedures
- Usage conventions

Problem set #1 is due one week from today

Meet in SN014 next Thursday

Load and Stores in action

An example of how loads and stores are used to access arrays.

C:

```
int sum = 0;
int values[10] = {1,3,5,7,9,11,
                  13,15,17,19};

int i;

for (i = 0; i < 10; i++)
    sum += value[i];
```

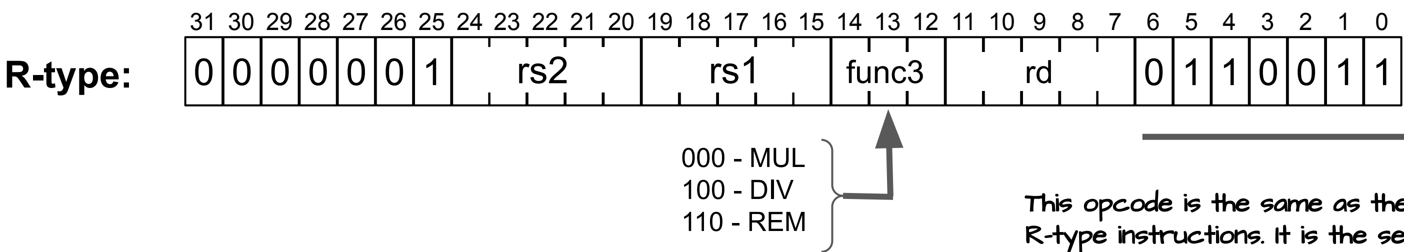
Assembly:

```
        addi    x31,x0,10
        addi    x5,x0,0         # x5 is i
loop:   slli    x6,x5,2
        lw      x6,values(x6)   # value[i]
        lw      x7,sum(x0)      # x5 is sum
        add     x7,x7,x6        # sum +=
value[i];
        sw      x7,sum(x0)
        addi    x5,x5,1
        blt     x5,x31,loop
*halt:  jal     x0,halt

sum:    .word   0
values: .word   1,3,5,7,9,11,13,15,17,19
```

Missing Math instructions

The RISC-V ISA includes integer multiplication and division as an extension to the RV32I minimal ISA called RV32M. This is because multiply and divide require significant additional H/W. These instructions can always be emulated, and cost is a consideration for embedded systems. Our miniRISCV simulator includes a subset of RV32M, the subset necessary to implement C.



This opcode is the same as the all other R-type instructions. It is the setting of the "func7" field that indicates the optional multiply/divide instructions

Multiply

MUL: multiply registers

Syntax: **mul *rd,rs,rt***

Encoding: **0000 001*t tttt ssss s*000 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] * Reg[t]

Multiply the contents of *Reg[s]* and *Reg[t]*, and place the lower 32-bits of the product in *Reg[d]*. ***Overflows are ignored.*** MUL is binary and semantically compliant with the RV32M mul instruction.

Example: `mul x5,x6,x7          # Encoded as: 0x027302B3`

Divide

DIV: divide registers

Syntax: **div *rd,rs,rt***

Encoding: **0000 001*t tttt ssss s*100 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] / Reg[t]

Divide the contents of *Reg[s]* by *Reg[t]*, and place the quotient in *Reg[d]*. ***A divisor of 0 does not generate an overflow***, and the destination register *rd* is set to all ones. DIV is binary and semantically compliant with the RV32M div instruction.

Example: `div x7,x5,x6    # Encoded as: 0x0262C3B3`

Remainder

REM: remainder of a register quotient

Syntax: **rem *rd,rs,rt***

Encoding: **0000 001*t tttt ssss s*110 *dddd d*011 0011**

Description:

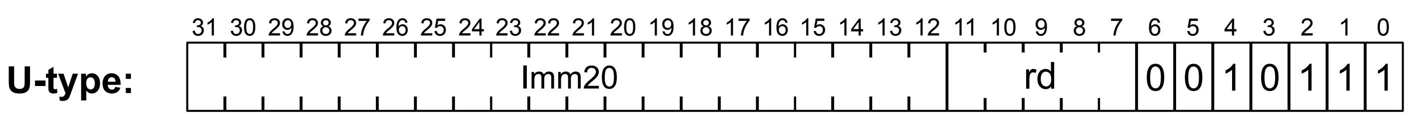
Reg[d] ← Reg[s] % Reg[t]

Divide the contents of *Reg[s]* by *Reg[t]*, and place the remainder in *Reg[d]*. A divisor of 0 does not generate an overflow and the contents of the destination register *rd* is set to the dividend, *Reg[s]*. REM is binary and semantically compliant with the RV32M rem instruction.

Example: `rem x7,x5,x6          # Encoded as: 0x0262E3B3`

Another instruction

Recall that last lecture we discussed the virtues of relocatable code resulting from having ***PC-relative*** branch and jump instructions. However, there is still a problem with relocatable "data". For instance you might want to place a data structure in a code section and be able to relocate it without the need for keeping track of the absolute addresses. Earlier RISC architectures, like MIPS, provided **lui** to construct addresses, but these were absolute addresses, so, in practice, almost all compilers would locate variables using a pointer scheme so they could be relocated. RISCV fixes this problem with a pc-relative **lui-like** instruction called ***auipc***




```
        auipc x10,next  # encoded as 0x00000517
next:
```

AUIPC DETAILS

AUIPC: Add Upper Immediate to PC

Syntax: **auipc *rd,imm20***

Encoding: ***iiii iiii iiii iiii iiii dddd d*011 0011**

Description:

Reg[d] ← PC + sign_extend(imm20 << 12)

Adds the sign-extended 20-bit immediate value, left-shifted by 12 bits, to the program counter, and writes the result to Reg[d].

Example: `auipc x31,1024          # Encoded as: 0x00400F97`

Also, the combination of an **auipc** instruction and a **jalr** can transfer control (jump) to any memory location. Both branch and jump instructions have limited ranges.

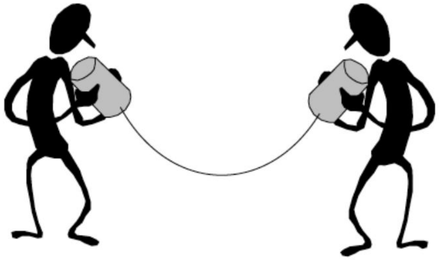
Pseudo-instructions

Assembler Directives

Functions and procedure Calls

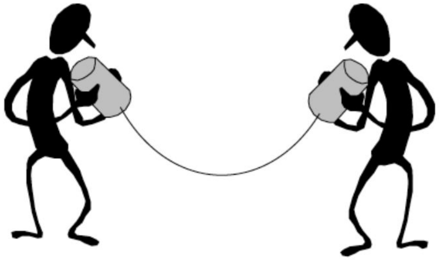
Functions and procedures are essential components of code reuse. The also allow code to be organized into modules. A key components of procedures are they:

- can be called from anywhere by a caller, and, when finished, they return back to where they were called from
- can have their only local variables
- clean up behind themselves, they avoid creating unintended side-effects
- can call themselves to implement Recursive methods/functions



Supporting procedure Calls

Reusable code also requires agreed upon conventions, such as where a caller's arguments can be found by the callee. These are actually not part of the ISA, they are part of a standard called the processor's "Application Binary Interface" or ABI.



Basics of procedure calling:

1. Put parameters where the called procedure can find them
2. Transfer control to the procedure
3. Acquire the needed storage for procedure variables
4. Perform the expected calculation
5. Put the result where the caller can find them
6. Return control to the point just after where it was called

Register use conventions

By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages. We'll use these names increasingly. Why have such conventions?

x0/zero (always zero)	x16/a6 (argument)
x1/ra (return address)	x17/a7 (argument)
x2/sp (stack pointer)	x18/s2 (saved)
x3/gp (global pointer)	x19/s3 (saved)
x4/tp (thread pointer)	x20/s4 (saved)
x5/t0 (temporary)	x21/s5 (saved)
x6/t1 (temporary)	x22 (saved)
x7/t2 (temporary)	x23 (saved)
x8/fp (frame pointer)	x24 (saved)
x9/s1 (saved)	x25 (saved)
x10/a0 (argument/return value 1)	x26 (saved)
x11/a1 (argument/return value 2)	x27 (saved)
x12/a2 (argument)	x28 (temporary)
x13/a3 (argument)	x29 (temporary)
x14/a4 (argument)	X30 (temporary)
x15/a5 (argument)	X31 (temporary)

Basics of Procedure Calling

```
main:    lw   a0,x
         lw   a1,y
         jal  ra,gcd
         sw   a0,z

*halt:   j    halt

x:       .word   35
y:       .word   55
z:       .word   0
```

```
int x = 35;
int y = 55;
int z;

void main() {
    z = gcd(x, y);
}

int gcd(a,b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

```
gcd:     beq  a0,a1,return
         blt  a0,a1,else
         sub  a0,a0,a1
         beq  x0,x0,gcd
else:    sub  a1,a1,a0
         beq  x0,x0,gcd
return:  jalr zero,(ra)
```

That was a little too easy

```
main:   lw   a0,x
        jal  ra,fact
        sw   a0,y
*halt:  j    halt

x:      .word   2
y:      .word   0
```

```
fact:    addi  t0,x0,1
         bge   t0,a0,return
         addi  t0,x0,a0
         addi  a0,a0,-1
         jal   ra,fact
         mul   a0,a0,t0
return:  jalr  x0,ra
```

```
int x = 5;
int y;

void main() {
    y = fact(x);
}

int fact(x) {
    if (x <= 1)
        return x;
    else
        return x*fact(x-1);
}
```

This time, things are really messed up.

The recursive call to fact( ) overwrites the daved value of x in t0.

To make a bad thing worse, the ra is also overwritten.

I knew there was a reason that I avoid recursion.

Next Time

- Stacks
- Contracts
- Writing serious code