0.1 Start with little programs

```
        .data
<<data part>>
        .text
main:   li      $s1,1   # put numbers in some registers
        li      $s2,2
        li      $s3,3
        li      $s4,4
        li      $s5,5
<<code part>>
```

0.2 Add and subtract

0.2.1 C code

```
f = g + h; i = f - j;
```

0.2.2 Register assignments

The variables f, g, h, i and j are assigned to the registers $s1, $s2, $s3, $s4, and $s5.

0.2.3 Code

```
        add     $s1,$s2,$s3
        sub     $s4,$s1,$s5
```

0.3 A more complex expression

0.3.1 C code

```
f = (g + h) - (i + j);
```

0.3.2 More register assignments

The compiler creates two new variables, say in $t0 and $t1, as temporaries holding partial results.

0.3.3 Code

```
        add     $t0,$s2,$s3
        add     $t1,$s4,$s5
        sub     $s1,$t0,$t1
```

0.4 Moving data to/from memory

0.4.1 C code

```
FinishValue = StartValue + AddOn;
```

0.4.2 Register assignments

$s0, $s1, $s2 for FinishValue, StartValue, and AddOn

0.4.3 Code

```
<<data part>>=
StartValue:      .word   131072
AddOn:           .word   65536
FinishValue:     .word   0
<<code part>>=
        lw       $s1,StartValue
        lw       $s2,AddOn
        add      $s0,$s1,$s2
        sw       $s0,FinishValue
```

0.5 if: testing for equality and inequality

0.5.1 C code

```
    if(i == j) goto L1;
    f = g + h;
L1: f = f - i;
```

0.5.2 Code

```
<<code part>>=
         beq     $s4,$s5,L1
```

```
        add     $s1,$s2,$s3
L1:     sub     $s1,$s1,$s4
```

0.6 if-then-else

0.6.1 C code

```
if(j != 0) {
    f = g / j;
} else {
    f = g + h;
}
```

0.6.2 Code

The compiler has to generate an extra branch and an extra label, which were not in the C code. This frequently happens, because hogher level languages are designed to provide good control structures for programmers, while machines just execute instructions.

What goes wrong if the extra branch is taken out? What statement in C does this remind you of?

Does every if-then-else need an extra branch?

```
<<code part>>=
# ? $s5 == zero ?
        beq     $s5,$0,ElsePart
        div     $s1,$s2,$s5
        j       AfterIf   #  j for jump
ElsePart: add   $s1,$s2,$s3
AfterIf:
```

0.7 Another if

0.7.1 C code

```
if( (f == g) || (h == i) ) {
    f = f + j;

<<code part>>=
```

```
        beq     $s1,$s2,doif
        beq     $s3,$s4,doif
        j       afterif
doif:   add     $s1,$s1,$s5
        sub     $s2,$s2,$s4
afterif:
```

0.8 Compound conditional

0.8.1 C code

```
if( ( (f == g) && (h == i) ) || (j == 0) ) {
    f = f + g;
    h = j;
}
```

0.8.2 Code

About the statement $h = j$;: If we had an instruction that copied one register to another, say $move\$s2, \$s4$, we'd use it here. Actually, there is such an instruction in the MIPS assembly language.

Thought question: How do you think the *move* instruction is implemented?

0.8.3 Code

```
<<code part>>=
        bne     $s1,$s2,checkj
        beq     $s3,$s4,dotheif
checkj: bne     $s5,$0,out
dotheif: add    $s1,$s1,$s2
        move    $s3,$s5
out:
```

0.9 Testing if one variable is less than another

The MIPS machine language has only one instruction to compare the values in registers. The instruction is called *set less than* or *slt*. Read the (very short) explanation in P&H, page 78.

0.9.1 C code

```
if(f >= g)
    h = f + g;
```

0.9.2 Register assignments

The code uses $t0 for the temporary variable needed by *slt*

0.9.3 Code

```
<<code part>>=
        slt     $t0,$s1,$s2
        bne     $t0,$0,beyond
        add     $s3,$s1,$s2
beyond:
```

0.10 Implementing a while loop

0.10.1 C code

Calculate f to the power g.

```
answer = 1;
count = g;
while(count >= 0) {
    answer = answer * f;
    count = count - 1;
}
```

Programs commonly use small constants, like the 1 that occurs at two places in this C code. So architects include instructions that make using those small constants fast.

Examples include the *load immediate* instruction or *li*.

li $t0, 1000$ puts the value 1000 in register $t0.

Another example is *add immediate* or *addi*. The result of *addi* $\$t0, -100$ is to subtract 100 from register $t0.

0.10.2 Register assignments

Answer will be $t0, and count will be $t1. The temporary for *slt* is $t2.

0.10.3 Code

```
<<code part>>=
        li      $t0,1   \# answer = 1
        move    $t1,$s2 \# count = g
while:  slt     $t2,$0,$t1  \# is 0 < count ?
        beq     $t2,$0,onward  \# if false, get out
# body of the loop
#   answer = answer * f
        mul     $t0,$t0,$s1
#   count = count - 1
        addi    $t1,$t1,-1
#   back to top of the loop
        j       while
#
# from here on is outside the loop
onward:
# We can store the answer
        sw      $t0,FinishValue
```