

Start with little programs

```

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

```

1 Add and subtract

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

Register allocation

The variables f , g , h , i and j are assigned to the registers $\$s0$, $\$s1$, $\$s2$, $\$s3$ and $\$s4$.

Code

```

1b  <code part 1b>≡
      add    $s0,$s1,$s2
      sub    $s3,$s0,$s4

```

2 A more complex expression

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

More register allocations

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

Code

```

1c  <code part 1b>+≡
      add    $t0,$s1,$s2
      add    $t1,$s3,$s4
      sub    $s0,$t0,$t1

```

3 Moving data to/from memory

```
finishValue = startValue + add0n;
```

Register allocations

\$s5, \$s6, \$s7 for finishValue, startValue, and add0n.

Code

```
2a  <data part 2a>≡
    startValue:    .word    131072
    add0n:         .word    65536
    finishValue:   .word    0

2b  <code part 1b>+≡
    lw    $s5, startValue
    lw    $s6, add0n
    add   $s7, $s5, $s6
    sw    $s7, finishValue

    or, alternatively,

2c  <code part 1b>+≡
    la    $t0, startValue
    lw    $s5, 0($t0)
    lw    $s6, 4($t0)
    add   $s7, $s5, $s6
    sw    $s7, 8($t0)
```

4 if

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

Code

```
2d  <code part 1b>+≡
    beq   $s3, $s4, L1
    add   $s0, $s1, $s2
L1:    sub   $s0, $s1, $s2
```

5 if - then - else

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

Code

Higher level languages provide good control structures for programmers, while machines just execute instructions. Because of that, the compiler must generate an extra branch and an extra label.

```
3a  <code part 1b>+≡
      beq    $s4,$0,Elsepart # ? Is $s4 == zero ?
      div    $s0,$s1,$s4
      j      AfterIf
ElsePart: add  $s0,$s1,$0
AfterIf:
```

6 Another if

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

Code

```
3b  <code part 1b>+≡
      beq    $s0,$s1,doif
      beq    $s2,$s3,doif
      j      afterif
doif:  add    $s0,$s0,$s4
      sub    $s1,$s1,$s2
afterif:
```

7 Compound conditional

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

Code

```
4 <code part 1b>+≡  
    bne    $s0,$s1,checkj  
    beq    $s2,$s3,dotEIF  
checkj: bne    $s4,$0,out  
dotEIF: add    $s0,$s0,$s1  
    move   $s2,$s4    # the move instruction  
out:
```

8 Testing if one variable is less than another

Left for the reader. For the `slt` instruction, read the explanation in P&H, pages 108 and 109.

9 A while loop

Also left to the reader.

10 Printing strings and characters

The MARS MIPS simulator provides a small set of operating system-like services through the `syscall` instruction. Explanation is in P&H, Appendix B, pages B-43 to B-45.

```
printf('%s','Hello, world! ');
printf('%c', 'Z');
// and newline
printf('%c', '\n');
```

Code

```
5a  <data part 2a>+≡
    greeting:
        .asciiz 'Hello, world!'

5b  <code part 1b>+≡
    li    $v0,4    # system call code for print_string
    la    $a0,greeting # address of string to print
    syscall      # print the string
#
    li    $v0,11   # system call code for print_char
    li    $a0,'Z'  # character to print
    syscall      # print it
#
    li    $v0,11   # system call code for print_char
    li    $a0,10   # newline
    syscall      # print it
```