Building a *for* Loop

```
sum = 0;
for ( i=1 ; i<=20 ; i++)
  sum = sum + i;
```

- Two ways it might be built:

```
        sum = 0;
        i=1;
test:   if (i <= 20 ) goto for;
        goto end;
for:    sum = sum + i;
        i++;
        goto test;
end:
```

```
        sum = 0;
        i=1;
for:    if (i > 20 ) goto end;
        sum = sum + i;
        i++;
        goto for;
end:
```


Building a *for* Loop

- The *for* loop in simplified C:

```
        sum = 0;
        i=1;
test:   if (i <= 20 ) goto for;
        goto end;
for:    sum = sum + i;
        i++;
        goto test;
end:
```

- The *for* loop in assembly language (LOAD / STORE format):

```
        LOAD  R0,#0        ; hold sum in R0
        LOAD  R1,#1        ; hold i in R1
test:   BRLE  R1,#20,for   ; if (i<=20)…
        JUMP  end
for:    ADD   R0,R0,R1     ; sum = sum + i
        ADD   R1,R1,#1     ; i++
        JUMP   test
end:    STORE 100,R0       ; store sum (?)
```


Variations on the BRANCH Instruction

- BRANCH with implicit second operand of zero (i.e., comparison to zero):

  BRLTZ R*1*, *label* ; PC = *label* if R*1* < 0

  BRLEZ R*1*, *label* ; … if R*1* <= 0

  …

- If the arithmetic instructions store information about their result somewhere (e.g., whether it's zero or negative), the the BRANCH can use that information as an implicit operand:

  BRZ *label* ; PC = *label* if Z

  BRN *label* ; PC = *label* if N

  BRO *label* ; PC = *label* if O

  …

  - A *condition code* register keeps track of the N (negative), Z (zero), O (overflow), etc. bits

Variations on the BRANCH Instruction (cont.)

- The familiar branch instructions can also take advantage of the condition codes:

  BRLT *label* ; PC = *label* if N

  BRLE *label* ; PC = *label* if N or Z

  BREQ *label* ; PC = *label* if Z

  …

  - Think about the above instructions when preceded by "SUB R1,R3,R6"
    - This is confusing, but when thinking about SUB the instructions above make sense
    - Thinking about ADD (or other arithmetic instructions) doesn't work with the above instructions, but does work with the more general branches on the previous slide
  - If you don't want to actually subtract, use the "compare" instruction to simply compare two operands: "CMP R3, R6"

- Read Section 4.5 in detail (except 4.5.7)

Worksheet

- A particular keyboard communicates with the CPU as follows:
  - A status register is addressed by the CPU at memory location 200. Bit 7 (little endian addressing) is used to indicate that a character is available. If so, that bit is 1; otherwise, it is 0. Other bits serve other purposes.
  - A data register, addressed by the CPU at memory location 201, stores the character typed on the keyboard.

Write an assembly language code sequence that loops forever. Inside that loop, the CPU should check the status register until a character is available, and when one is available, store it into register R9. Use the book's LOAD / STORE instruction format, and condition codes for conditional branches.

Bit Manipulation

- Bitwise operations:

```
AND     Rdest, Rsrc1, Rsrc2
                    ; Rdest = Rsrc1 & Rsrc2
OR      Rdest, Rsrc1, Rsrc2
XOR     Rdest, Rsrc1, Rsrc2
```

  - These operations perform a bitwise *and*, *or*, or *xor* on their two source operands
- Used with a *mask* to selectively manipulate bits:

1010 1100 & 1111 <u>0000</u> --> 1010 <u>0000</u>

1010 1100 | <u>1111</u> 0000 --> <u>1111</u> 1100

1010 1100 ^ <u>1111</u> 0000 --> <u>0101</u> 1100

Bit Manipulation (cont.)

- Direct bit access operations:

```
BB      Rsrc, n, label
        ; PC = label if nth bit of Rsrc is 1
BSET    Rdest, n, m
        ; sets the nth through mth bit
        ; of Rdest to 1
BCLR    Rdest, n, m
        ; sets the nth through mth bit
        ; of Rdest to 0
```

- Shift and rotate (left) operations:

```
SLZ     Rdest, Rsrc, n
        ; Rdest = Rsrc shifted left n bits,
        ; filling vacated bits with 0s
SLO     Rdest, Rsrc, n
        ; Rdest = Rsrc shifted left n bits,
        ; filling vacated bits with 1s
ROTL    Rdest, Rsrc, n
        ; Rdest = Rsrc rotated left n bits
```


Homework #4 — Due 10/26/98 (Part 1)

1. Translate into assembly language, using the book's LOAD / STORE instruction format. Assume that sum, sqsum, i, and n correspond to registers R1–R4, in that order. Store the new values of sum and sqsum back into memory locations 100 and 101, respectively.

```
sum = 0;
sqsum = 0;
for (i=1; i<=n; i++)
   sum = sum + i;
   sqsum = sqsum + i * i;
```