Immediate Values (Constants)

- So far, we have seen that instruction operands can be either:
  - Addresses LOAD R1,23
  - Registers ADD R1,R2,R4
  - (or some combination of the above, in most formats except LOAD/STORE)
- We may also need constants
  - Operands can also be *immediate* values (constants) stored within the instruction
- Immediate values can be distinguished from addresses by:
  - Prefixing each immediate operand by a special symbol (e.g., "#300" for the constant 300)
  - Using special versions of each instruction (e.g., LOADI (load immediate) versus LOAD) (not a common technique…)

Immediate Values (Constants) (cont.)

- A LOAD/STORE architecture supports:
  - Arithmetic operations — third operand can be either address or immediate value
    - ADD R1,R2,R3
    - ADD R1,R1,#1
    - Instruction format has room for opcode, dest register, src1 reg, and either immediate value (large) or src2 reg (small)
  - LOAD/STORE operations — non-register operand must be address (not immediate)
    - LOAD R1,300
    - Instruction format has room for opcode, dest register, and address
  - MOV operation — third operand can be either address or immediate value
    - MOVE R2,R3 copy R3 into R2
    - MOVE R2,#1 copy "1" into R2
    - Instruction format has room for opcode, dest register, and either immediate value (large) or src2 reg (small)

Operand Sizes

- Most memory systems allow the data to be accessed in a variety of sizes
  - Word (16 bit, 32 bit, etc.)
  - Byte
  - Half-word, double-word, etc.
- The data width can be specified by:
  - Using special versions of each instruction (e.g., ADDB (add byte) versus ADDW (add word))
  - Suffixing each instruction by a special symbol (e.g., ADD.b (add byte) versus ADD.w (add word))
- RISC machines typically
  - Allow variable widths on LOAD and STORE instructions
  - Use full width for data manipulation

Control Flow Constructs in C

- *if…then…else* constructs:

```
if ( a<max )
   b = c;
else
   b = c + 1;
```

- *for* loops & *while* loops:

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

- What if all C had was a very simple *if* statement…

  if (*condition*) *statement*;

- …and a statement that can "jump" to an arbitrary line in the program?

  goto *label*;

Building an *if…then…else* Construct

```
if ( a<max )
   b = c;
else
   b = c + 1;
```

- Two ways it might be built:

```
            if ( a<max ) goto then;
            goto else;
then:       b = c;
            goto end;
else:       b = c + 1;
end:

            if ( a>=max ) goto else;
then:       b = c;
            goto end;
else:       b = c + 1;
end:
```


JUMP = "goto" in Assembly Language

- Most instruction sets include a JUMP (or BRANCH) instruction, which unconditionally jumps to the instruction at the specified address

  JUMP *label*

- The JUMP instruction works by storing the specified address in the PC (Program Counter)

- Some common conventions:
  - A label must be the first item on a line, and is followed by a colon (":")
  - A label refers to the next instruction (which may or may not be on the same line as the label)
  - It is acceptable to refer to a label in a JUMP instruction before the label is defined

BRANCH = "if" in Assembly Language

- Most instruction sets include a *conditional* BRANCH (or JUMP) instruction, which conditionally jumps to the instruction at the specified address

```
BRLT    R1, R2, label   ; PC = label
                            if R1 < R2
BRLE    R1, R2, label   ; … if <=
BRGT    R1, R2, label   ; … if >
BRGE    R1, R2, label   ; … if >=
BREQ    R1, R2, label   ; … if equal
BRNE    R1, R2, label   ; … if not equal
```

- If the relationship between first two operands is true, the instruction jumps to the specified address (the 3rd operand)
  - If true, the specified address is stored in the PC (Program Counter)
  - Otherwise, the PC is left untouched

First (?) Description of Branching

Burks, Goldstine, and von Neuman, 1947

> The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent on the results of the computation. When the iteration is completed a different sequence of [instructions] is to be followed, so we must, in most cases, give two parallel trains of [instructions] preceded by an instruction as to which routine is to be followed. This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Building an *if…then…else* Construct

- The *if…the…else* in simplified C:

```
        if ( a<max ) goto then;
        goto else;
then:   b = c;
        goto end;
else:   b = c + 1;
end:
```

- The *if…the…else* construct in assembly language (LOAD / STORE format):

```
        LOAD  R0,100      ; hold a in R0
        LOAD  R1,101      ; hold max in R1
        LOAD  R2,102      ; hold b in R2
        LOAD  R3,103      ; hold c in R3
        BRLT  R0,R1,then  ; if (a<max)…
        JUMP  else
then:   MOVE R2,R3        ; b = c
        JUMP   end
else:   ADD    R2,R3,#1   ; b = c + 1
end:    STORE 101,R2      ; store b (?)
```