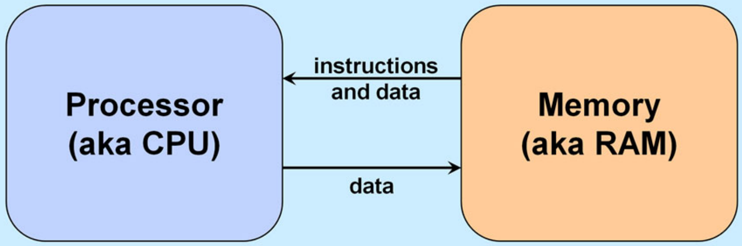
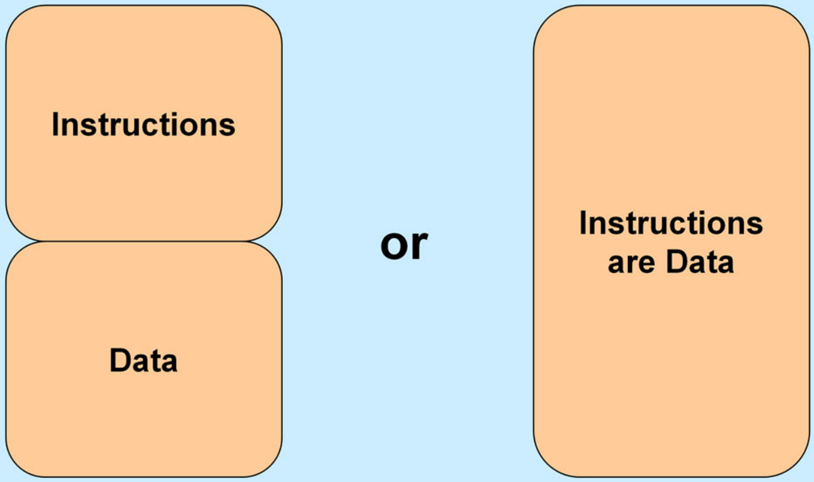
CS 33

Intro to Machine Programming

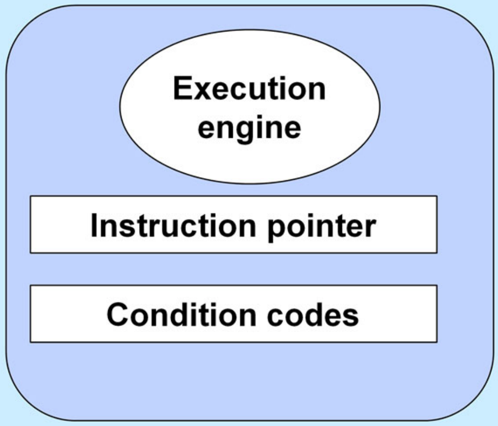
Machine Model



Memory



Processor: Some Details



Processor: Basic Operation

```
while (forever) {
  fetch instruction IP points at
  decode instruction
  fetch operands
  execute
  store results
  update IP and condition code
}
```

Instructions ...

Op code	Operand1	Operand2	...

Operands

- Form
  - immediate vs. reference
    - » value vs. address
- How many?
  - 3
    - » add a,b,c
      - c = a + b
  - 2
    - » add a,b
      - b += a

Operands (continued)

- Accumulator
  - special memory in the processor
    - known as a *register*
    - fast access
  - allows single-operand instructions
    - add a
      - acc += a
    - add b
      - acc += b

From C to Assembler ...

```
a = (b + c) * d;

    mov    b,%acc
    add    c,%acc
    mul    d,%acc
    mov    %acc,a
```

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

    cmp    a,b
    jge    .L1
    mov    $1,c
    jmp    .L2
.L1
    mov    $1,d
.L2
```

immediate operand

immediate operand

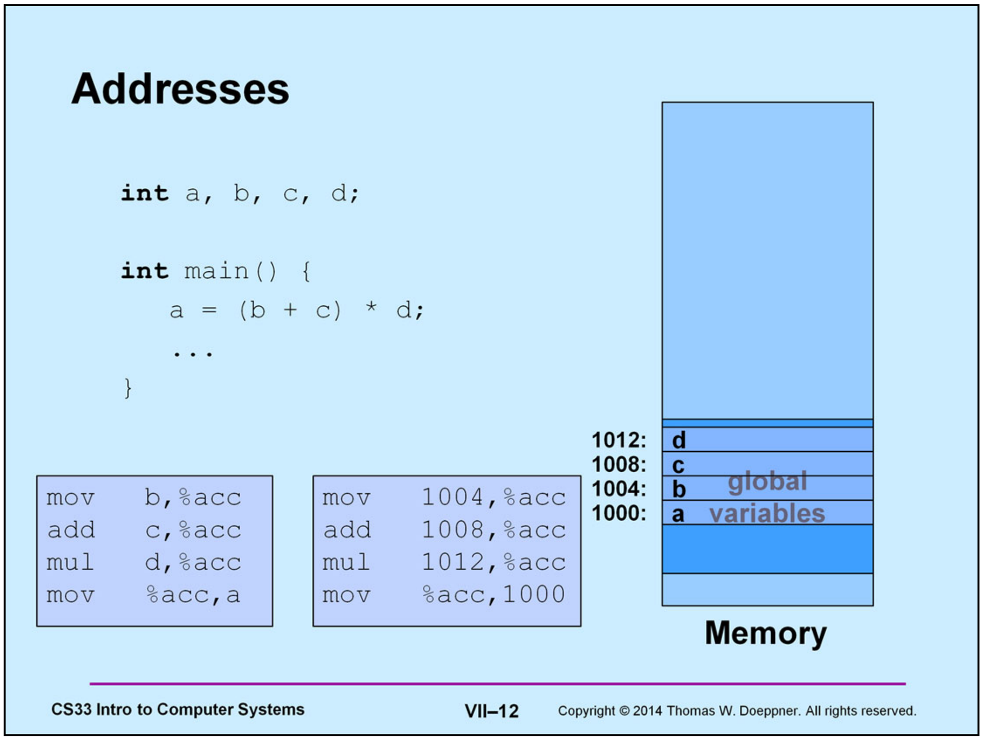
Note that we're using the accumulator in two-operand instructions. The "%" makes it clear that "acc" is a register.

Condition Codes

- Set of flags including status of most recent operation:
  - zero flag
    - » result was or was not zero
  - sign flag
    - » result was or was not negative (sign bit is or is not set)
  - overflow flag
    - » for values treated as signed
  - carry flag
    - » for values treated as unsigned
- Set implicitly by arithmetic instructions
- Set explicitly by compare instruction
  - cmp a,b
    - » sets flags based on result of b-a

Jump Instructions

- Unconditional jump
  - just do it
- Conditional jump
  - to jump or not to jump determined by condition-code flags
  - field in the op code indicates how this is computed
  - in assembler language, simply say
    - » je
      - jump on equal
    - » jne
      - jump on not equal
    - » jgt
      - jump on greater than
    - » etc.



In the C code above, the assignment to *a* might be coded in assembler as shown in the box in the lower left. But this brings up the question, where are the values represented by *a*, *b*, *c*, and *d*? Variable names are part of the C language, not assembler. Let's assume that these global variables are located at addresses 1000, 1004, 1008, and 1012, as shown on the right. Thus correct assembler language would be as in the middle box, which deals with addresses, not variable names.

Addresses

```
int b;

int func(int c, int d) {
    int a;
    a = (b + c) * d;
    ...
}

    mov    ?,%acc
    add    ?,%acc
    mul    ?,%acc
    mov    %acc,?
```

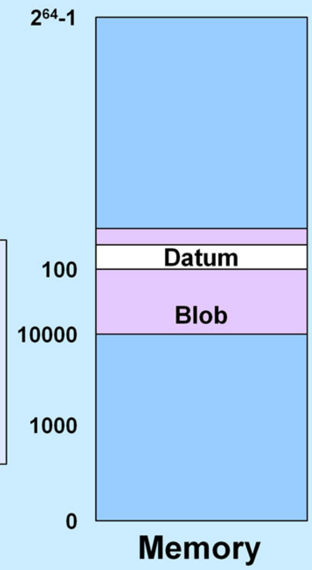
- One copy of *b* for duration of program's execution
  - *b*'s address is the same for each call to *func*
- Different copies of *a*, *c*, and *d* for each call to *func*
  - addresses are different in each call

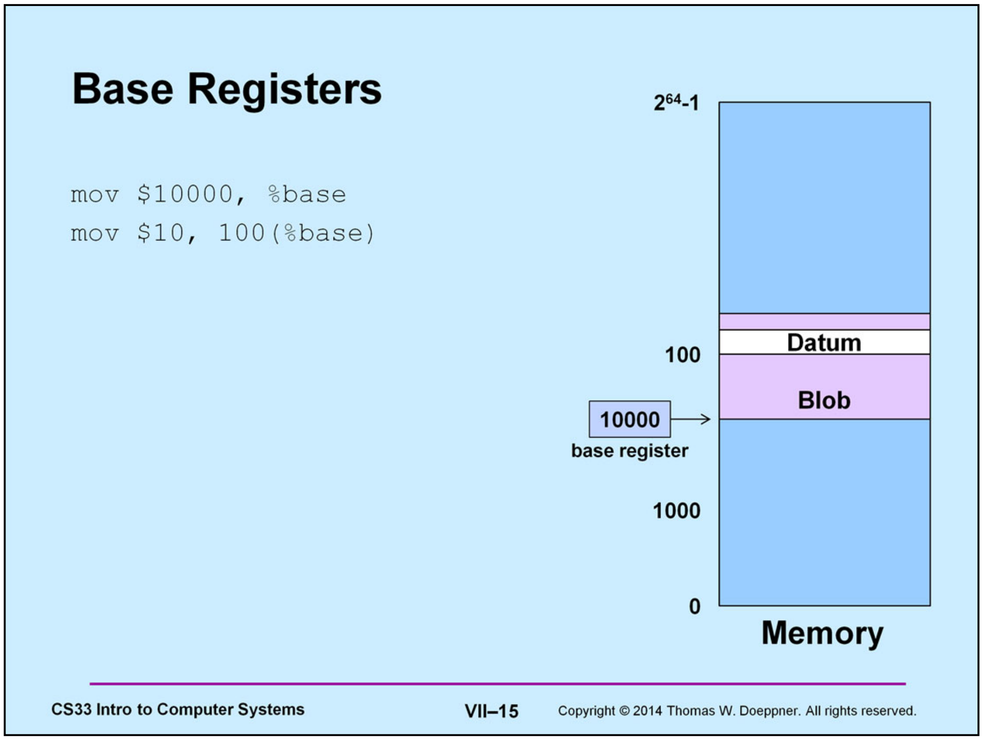
Here we rearrange things a bit. *b* is a global variable, but a is a local variable within *func*, and *c* and *d* are arguments. The issue here is that the locations associated with *a*, *c*, and *d* will, in general, be different for each call to *func*. Thus we somehow must modify the assembler code to take this into account.

Relative Addresses

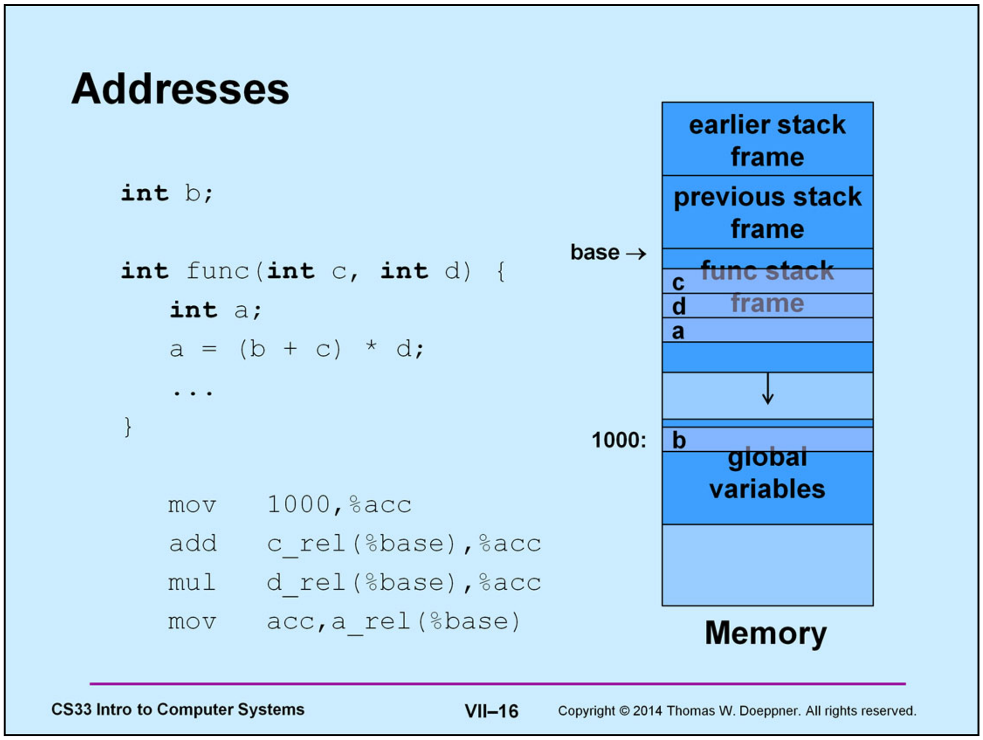
- **Absolute address**
  - actual location in memory
- **Relative address**
  - distance from some other location

- Blob's absolute address is 10000
- Datum's relative address (to Blob) is 100
  - its absolute address is 10100

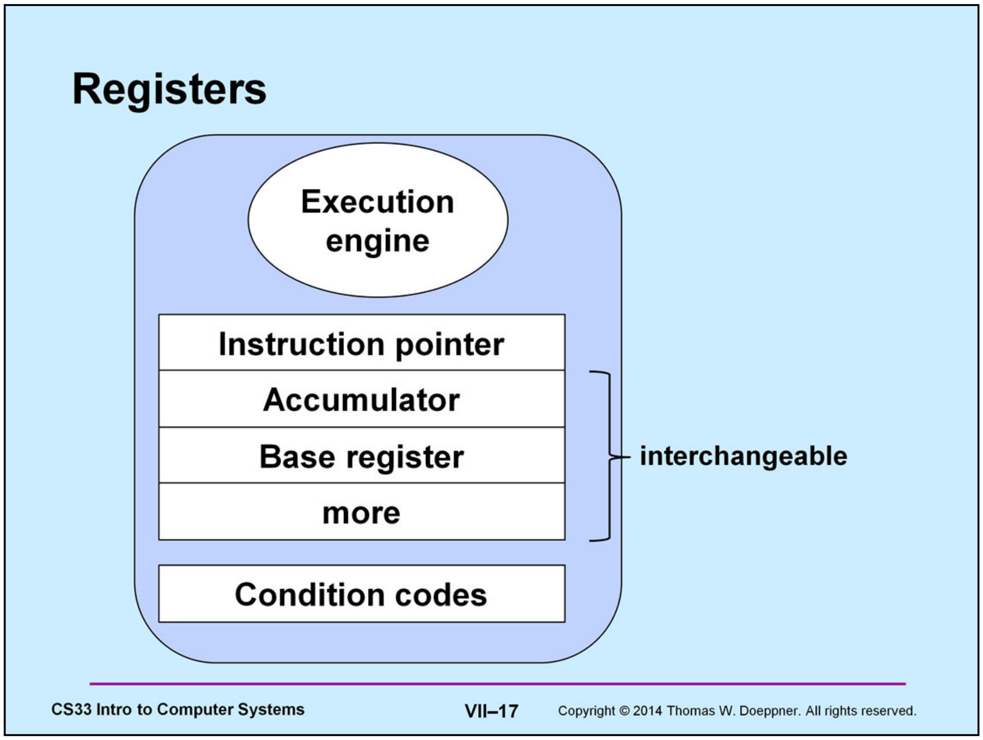




Here we load the value 10,000 into the base register (recall that the "$" means what follows is a literal value; a "%" sign means that what follows is the name of a register), then store the value 10 into the memory location 10100 (the contents of the base register plus 100): the notation *n(%base)* means the address obtained by adding *n* to the contents of the base register.



Here we return to our earlier example. We assume that, as part of the call to *func*, the base register is loaded with the address of *func*'s current stack frame, and that the local variable *a* and the parameters *c* and *d* are located within the frame. Thus we refer to them by their relative addresses within the stack frame, which are assumed to be *a_rel*, *c_rel*, and *d_rel*.



We've now seen four registers: the instruction pointer, the accumulator, the base register, and the condition codes. The accumulator is used to hold intermediate results for arithmetic; the base register is used to hold addresses for relative addressing. There's no particular reason why the accumulator can't be used as the base register and vice versa: thus they may be used interchangeably. Furthermore, it is useful to have more than two such dual-purpose registers. As we will see, the x86 architecture has eight such registers; the x86-64 architecture has 16.