Chapter 9 -- registers

```
REGISTERS
---------

An introduction to the subject of registers -- from a motivational
point of view.

This lecture is an attempt to explain a bit about why computers
are designed (currently) the way they are.  Try to remember that
speed of program execution is an important goal.  Desire for increased
speed drives the design of computer hardware.


The impediment to speed (currently):  transfering data to and from
memory.

look at a SASM instruction:
    iadd  x, y

    -x and y must all be addresses of data in memory.
    -each address is 32 bits.
    -so, this instruction requires MORE than 64 bits.

    if each read from memory delivers 32 bits of data,
    then it takes a lot of reads before this instruction can
    be completed.
       3 for instruction fetch
       1 to load x
       1 to load y
       1 to store x

       that's 6 transactions with memory for 1 instruction!


How bad is the problem?
  Assume that a 32-bit 2's complement addition takes 1 time unit.
  A read/write from/to memory takes about 10 time units.

  So we get
     fetch instruction:  30 time units
     decode               1 time unit
     load x              10 time units
     load y              10 time units
     add                  1 time unit
     store x             10 time units
     ---------------------------------
       total time:       62 time units

     60/62 = 96.7 % of the time is spent doing memory operations.



what do we do to reduce this number?
  1. transfer more data at one time
     if we transfer 2 words at one time, then it only takes 2 reads
     to get the instruction.  There is no savings in loading/storing
```

the operands. And, an extra word worth of data is transferred
for each load, a waste of resources.
So, this idea would give a saving of 1 memory transaction.

2. modify instructions such that they are smaller.
The Pentium ALREADY has done this! It only has 2 operands
for each instruction.

Most modern machines allow 3 operands, to give instructions
like:

```
add  x, y, z    ; x <- (y) + (z)
```

Note that this instruction makes the problem worse!
Add up the memory accesses for this one!
They call a machine like this a 3-address machine. Or, it has
a 3-address instruction set.

The differences between 2-address and 3-address instruction sets:
1. the 2-address instruction set can require more instructions
to do the same operation as the 3-address instruction set.

example:

```
add  x, y, z  ; 3-address instruction set

move  x, y    ; 2-address instruction set
add   x, z
```

memory accesses for this:

```
3-address instruction set: 4 instruction fetch
                           1 to load y
                           1 to load z
                           1 to store x
                   Total =  7

2-address instruction set: 3 instruction fetch (move)
                           1 to load y (move)
                           1 to store x (move)
                           3 instruction fetch (add)
                           1 to load z (add)
                           1 to load x (add)
                           1 to store x (add)
                   Total = 11
```

So, allow only 1 operand -- called a 1-address format.

now, the instruction add x, y, z will be accomplished
by something like

```
load  z
add   y
store x
```

to facilitate this, there is an implied integer of storage
associated with the ALU. All results of instructions
are placed into this integer -- called an ACCUMULATOR.

the operation of the sequence:
load z -- place the contents at address z into the accumulator
(sort of like if you did move accumulator, z in SASM)
add y -- implied operation is to add the contents of the

```
                    accumulator with the operand, and place the result
                    back into the accumulator.
          store x-- place the contents of the accumulator into the location
                    specified by the operand.
```

Notice that this 1-address instruction format implies the use of a variable (the accumulator).

How many memory transactions does it take?

```
   3 -- (load) 2 for instruction fetch, 1 for read of z
   3 -- (add) 2 for instruction fetch, 1 for read of y
   3 -- (store) 2 for instruction fetch, 1 for write of x
  ---
   9   Not better than the 3 address machine.
```

BUT, what if the operation following the add was something like

```
div x, x, 3
```

then, the value for x is already in the accumulator, and the code on the 1 address machine could be

```
load z
add  y
div  3
store x
```

there is only 1 extra instruction (3 memory transactions) for this whole sequence!

On the 3-address machine: 13 transactions
On the 1-address machine: 11 transactions

REMEMBER this: the 1 address machine uses an extra word of storage that is located in the CPU.

the example shows a savings in memory transactions when a value is re-used.

3. shorten addresses. This restricts where variables can be placed. First, make each address be 16 bits (instead of 32). Then

```
add  x, y, z
```

requires 2 32-bit words for instruction fetch.

Shorten addresses even more . . . make them each 5 bits long. Problem: that leaves only 32 words of data for operand storage. So, use extra move instructions that allow moving data from a 32 bit address to one of these special 32 words.

Then, the add can fit into 1 instruction.

NOW, put a couple of these ideas together.

Use of storage in CPU (accumulator) allowed re-use of data. Its easy to design -- put a bunch of storage in the CPU -- call them REGISTERS. How about 32 of them? Then, restrict arithmetic instructions to only use registers as operands.

```
add  x, y, z
```

becomes something more like

```
   load  reg10, y
   load  reg11, z
   add   reg12, reg11, reg10
   store x, reg12
```

presuming that the values for x, y, and z can/will be used again, the load operations take relatively less time.

A set up like this where arith/logical instr. use only registers for operands is called a LOAD/STORE architecture.

A computer that allows operands to come from main memory is often called a MEMORY TO MEMORY architecture, although that term is not universal.

Load/store architectures are common today. They have the advantages

1. instructions can be fixed length (and short)
2. their design allows (easily permits) pipelining, making load/store architectures faster
   (More about pipelining at the end of the semester)

IMPORTANT NOTE: The Pentium architecture (and also SASM) is NOT a load/store architecture! It was designed (and propagated through time) with different goals.

a discussion of addressing modes:

Once a computer has registers (and they ALL do!), then there can be lots of interesting uses of these registers.

Many computers (including the Pentium) offer more ways of getting at operands. These methods come under the classification of addressing modes.

load/store architectures usually have a VERY limited set of addressing modes available

memory to memory architectures (like Pentium) often offer LOTS of modes. This flexibility often forces these machines to have variable length instructions (like Pentium). Variable length instructions can make for all sorts of difficulties in making a processor go fast!

How to give an addressing mode? It requires extra bits for each operand to specify which addressing mode is used.

We would likely see an instruction something like:

```
   opcode   addr.mode1  extra.stuff.for.operand1
            addr.mode2  extra.stuff.for.operand2
```

Here are some addressing modes.
An addressing mode really gives the information of where an operand is (its address). An instruction decides how to use the address. This address is better termed an EFFECTIVE ADDRESS.

The processor generates an effective address for each operand. Depending on the instruction, that effective address may be used directly, OR it may be used to get the operand.

Register. The operand is in the register. The term effective address is not really appropriate here, since there is no address, just the designation for a register.

Imagine a computer that implemented SASM, but had 3 registers, called reg1, reg2, and reg3.
An addition instruction example that used a register addressing mode for one of its operands could be

```
   iadd   reg2, 1
```

The contents of reg2 is added to the value 1, and the result is placed back into reg2. The difference between this imaginary instruction and a real one is in the number of required bits for instruction encoding, and in the number of memory accesses required.

Immediate. The operand is contained within the instruction itself. So the effective address generated will be within the instruction.

```
example:  iadd  count, 3  ; a SASM example
```

Often, no effective address is generated at all. When the instruction is fetched, it contains an encoding of the immediate operand. Decoding the addressing mode for the operand leads to taking the operand from the instruction.

Direct. The effective address for an operand is in the instruction. Note that this is what SASM implies for most operands.

```
example:  iadd  count, 3  ; a SASM example
```

Register Direct. The effective address for an operand is in a register.

```
example:  add  [reg3], 3
```

The contents of reg3 is the effective address. For the add instruction, the contents at that address are loaded and then added to the immediate value 3. The result

goes back to that same effective address.

Base Displacement. Also called indexed or relative.
The effective address is the sum of the contents of a register plus a small constant.

Indirect. Adds a level of indirection to direct mode. An address is specified within the instruction. The contents at that address is the effective address.

A variation might be Register Indirect. The initial address is located in a register (instead of in the instruction).

PC Relative. The effective address is calculated relative to the current value of the program counter.

As a real life example of this, virtually every architecture has conditional control instructions that work this way.

An unnamed addressing mode for thought. The addressing mode specifies 2 registers. The effective address is calculated by adding the contents of the 2 registers together.