

Data Representation

Announcements (lots of 'em)

- **8** handouts (#03 - #10). Most of this is support files for the first assignment. Pick up one of 03M or 03P depending on whether you have a Macintosh or a PC.
- **1** book. This book is provided by Motorola free of cost. We currently only have about 26 of these but are ordering more. You don't really need this for the first assignment, but it's nice to get. I will also refer to this book in class, so it's a good idea to bring it. If you don't get a book today and find that you need to look something up, you can use the online version at: http://www.mot.com/SPS/WIRELESS/pdf/MC68000/UM_REV9/COMPLETE.PDF
- If you want to be sure to get a device, come by my office hours with your check **today**. There are a limited number of devices and after today, I will be opening up the loaner program to people in CS 194 (senior project) as well.
- The first assignment should be up on the web page and on Catacombs. The libraries (Palm instructions) are brand new this quarter, so let us know ASAP of any problems.
- I recently got a new version of the Macintosh PalmDebugger from Palm that allows you to attach to the emulator (and also contains some bug fixes). If you downloaded the early version of the Palm SDK (X-Palm SDK for Stanford), you should download the new version of the debugger.
- For some reason, the emulator and the debugger aren't getting along really well on the PC side. If you are trying to work device-less, you will have to stick to the Macintosh for now. I will work on getting a solution, but do realize that these are prerelease (and unsupported) tools that Palm is providing to us, so help might be a little slow.
- I will be gone starting Wednesday after class and will be back sometime Monday. This has a few consequences:
 - I will not be holding office hours on Wednesday.
 - Cameron will likely teach class on Monday. I will try to make it back for my office hours on Monday if at all possible.
 - Please direct any questions to cs110-staff@cs.stanford.edu. Cameron or I (if I can get access to e-mail) will answer questions as soon as we can.

Last time

- Data representation
 - Booleans
 - Unsigned integers
 - binary
 - hexadecimal
 - Signed integers
 - signed-magnitude
 - two's complement (started)
 - ones' complement (briefly)

This time

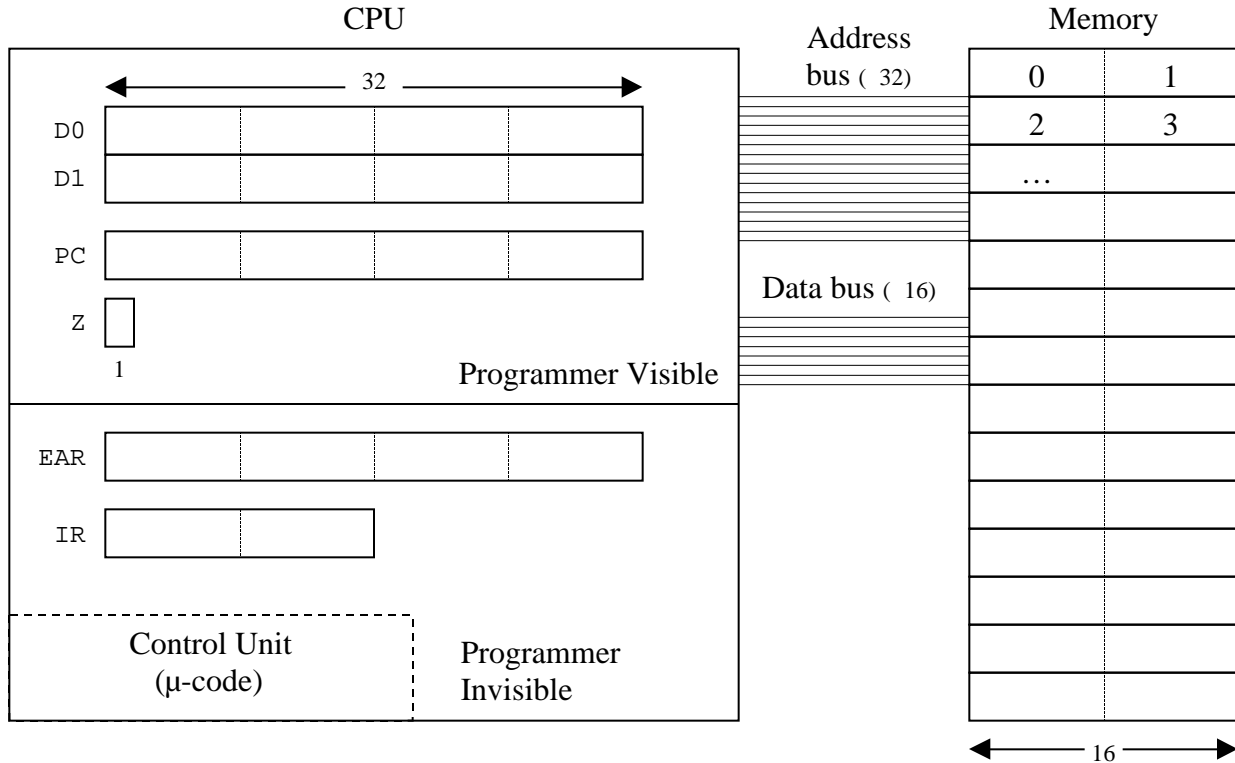
- Data representation (cont.)
 - Signed integers (cont)
 - a bit more about two's complement
 - Real numbers
 - fixed point (briefly)
 - floating point (very briefly)
 - Characters
 - ASCII
 - EBCDIC (briefly)
 - Strings
 - C-Strings
 - Pascal Strings
- Instruction representation
 - A simple computer
 - Overview
 - Programmer model
 - Memory
 - Registers
 - Other
 - Instruction representation
 - Inner workings
 - Example
 - The real 68000

A simple computer

We now know how to store numbers in a computer's memory. However, so far we have no model of how a computer really works. To get started, we will look at a simplified model of a 68000 processor and see how it works. Our CPU will have a (very) small subset of the features and instructions of a real CPU. We'll use our model to show how instructions can be represented in memory and how a CPU can process them.

Our model

Our computer model consists of a simple CPU (central processing unit) attached to memory:



Overview

The CPU is the brain of a computer. From a theoretical standpoint, its job is very simple. It does three things over and over and over again. These three things are:

1. **Fetch** - Fetch the next instruction from memory.
2. **Decode** - Figure out what that instruction is supposed to do.
3. **Execute** - Do the instruction, which will somehow change the state of the CPU or of memory.

How does the CPU accomplish its job? To some extent, that depends on the design of the chip. On a very low level, you can look at the CPU as a sequence of electronic components that is hard-wired to execute instructions stored in memory. We will see a little bit more about one way that some chips work a bit later.

What the programmer sees

Let's see what the programmer has available when writing programs for our simple CPU. From our drawing, we can see a few things:

Memory

As we recall from last class, memory is just a big sequence of bits that can be used to store data. In our model, the CPU is attached to the computer's memory by two groups of wires. The first group in our picture is known as the *address bus*. The address bus (32 bits wide in our model) is used by

the CPU to select a location in memory to read to or write from. Because the address bus is 32 bits wide, the CPU can select from 2^{32} (about 4 billion) memory locations. Our simple CPU is byte-addressable (just like the 68K). That means that the location indicated by the address bus is the location of one byte (8 bits) of memory. This gives us an addressable memory space of approximately 4 gigabytes.

The second set of lines in our model is labeled as the *data bus*. The data bus is used to transfer information between the CPU and memory. Notice that our data bus is 16 bits wide. If more than 16 bits need to be transferred to/from memory, the CPU must make two separate transfer requests to memory. The number of lines in the data bus varies greatly between different processors—it's a cost vs. speed tradeoff.

Notice the way that we've drawn the memory in our diagram. There are a few interesting points:

- Our convention in class is to draw memory as being 16 bits wide. This size is known as **1 word** of memory. Drawing memory as being 1 word wide is simply a convention—there's really nothing special about it.
- Notice the way that we've labeled memory. We've organized words of memory so that the lower address of memory makes up the more significant byte of the word and the higher address of memory makes up the less significant byte of the word. This method of constructing words out of bytes is known as the "Big Endian" method. The alternative ordering—shown here—is called the "Little Endian" method. The two methods have different advantages and disadvantages that are beyond the scope of this class. However, it is interesting to note that the only modern processor that uses the Little Endian method is the Intel x86 family. All other computers (include the standard for transmitting bytes over a network) are Big Endian.

1	0
3	2
...	

Registers

There are two locations in our model labeled `D0` and `D1`. These two locations are the registers in our simple machine. Registers are additional memory locations that the programmer can use to store data in. CPUs generally contain registers for two reasons. The first reason is that registers are much easier to address than memory. If our data is stored in memory, we always need to use 32 bits to specify its location. If our data is stored in a register, we need far fewer bits (in this case, we only need 1 bit to specify `D0` or `D1`). The second reason that CPUs have registers is that it is much quicker to access data in a register. Transferring information over the data bus is an expensive operation. If we can avoid doing it, our program will run much faster. Both of our registers are 32 bits wide.

Other features

There is one location labeled `PC`. This is the program counter. It keeps track of the memory location that holds the next instruction. The PC is 32 bits wide (because all memory addresses in our CPU are 32 bits wide).

There is a small location labeled `Z`. In our simple CPU, this is a single bit that tells whether the result of the last operation was 0.

Instructions

Thus far, we have talked about instructions being stored in memory without ever talking about why or how there are stored there. These days, it doesn't surprise most people that instructions are stored in the same memory location as data. However, early on in computers, this wasn't done. A computer's memory was used only to store data. Instructions were represented differently—either hard-wired into the machine or stored in another bank of memory. Computers that store data and instructions in the same memory are called Von Neumann machines.

Von Neumann machines work because we can translate instructions into binary and store them in memory. The instructions available and how to store them in memory differs from computer to computer. Our simple computer has four different instructions. Here are their names and their representations:

LOAD <addr> (\$2039 <addr1> <addr2>) - This instruction loads 32 bits into D0 from memory.

ADD (\$D280) - This instruction adds the contents of D0 to the contents of D1.

CLR (\$4281) - This instruction clears the contents of D1 (sets D1 to 0).

STORE <addr> - (\$23C1 <addr1> <addr2>) - This instruction stores 32 bits into memory from D1.

Obviously, our simple CPU can't do very much, but it will work well as an example.

Probably the first question that pops into your head is: "What was Doug thinking? How did he come up with these representations for instructions?" As far as our simple CPU is concerned, the representations that we have chosen here is completely arbitrary. However, in a real CPU (with more instructions), the representations are chosen so as to make the logic of the CPU simpler.

The inner workings of our CPU

As I said before, there are lots of ways that a CPU can get its job done. One way that is used for CISC chips like the 68000 is to write a program in a very simple instruction set whose job is to run the more complex instructions of the actual instruction set. This simple program is called the μ -code (micro-code), is not visible to the programmer. The programmer just knows that the CPU will execute instruction according to the definition of the instruction set. He/she does not know specifically how the CPU works (remember the wall of abstraction from CS 106?) Because the inner workings of the CPU are invisible to the programmer, chip designers are free to change how the chip works. They can add lots of fancy optimizations to make things go faster. Some CPUs (especially ones with simpler instruction sets) avoid μ -code altogether.

For the sake of example, our CPU does use micro-code. Here is what our CPU does (in C code):

```
static char MEM[4294967296];
static bool Z;
static short IR;
static long EAR, PC, D0, D1;

static void Fetch (void)
{
    IR = *((short*)(MEM + PC));
    PC += 2;
}
```

```

static void Execute (void)
{
    switch (IR) {
        case 0x2039:
            EAR = *((long *) (MEM + PC));
            PC += 4;
            D0 = MEM[EAR];
            Z = (D0 == 0);
            break;
        case 0xD280:
            D1 = D0 + D1;
            Z = (D1 == 0);
            break;
        case 0x4281:
            D1 = 0;
            Z = true;
            break;
        case 0x23C1:
            EAR = *((long *) (MEM + PC));
            PC += 4;
            MEM[EAR] = D0;
            Z = (MEM[EAR] == 0);
            break;
    }
}

main()
{
    PC = <starting address>;

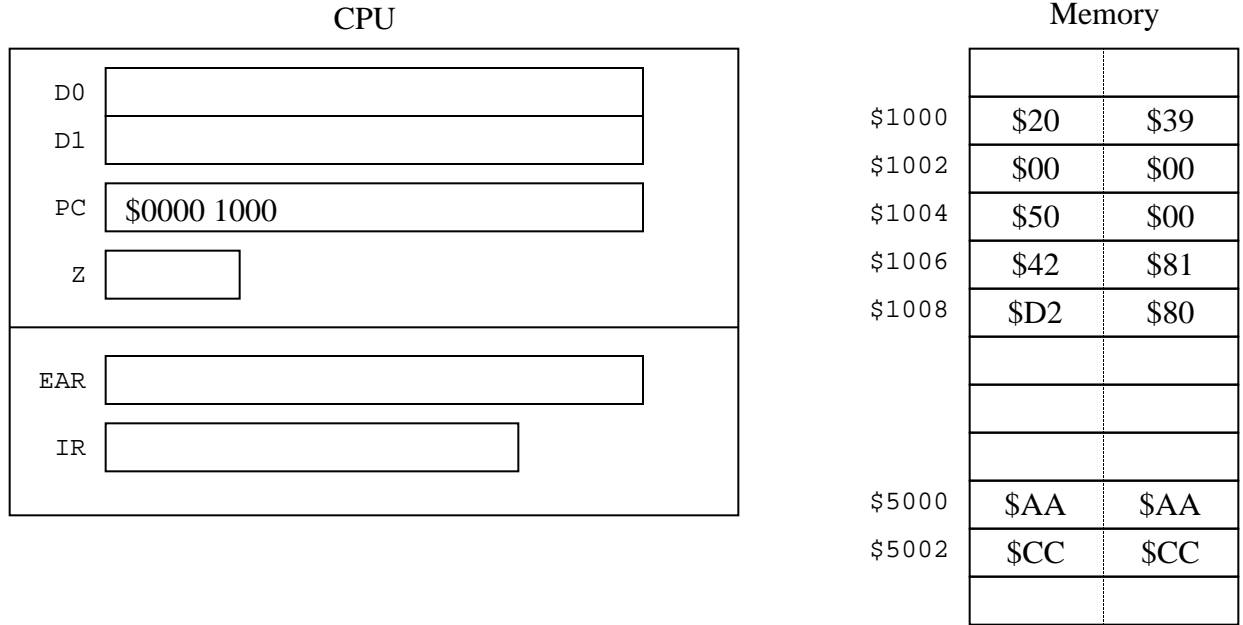
    while (true) {
        Fetch();
        Execute();
    }
}

```

Notice that all of the instructions that we used to implement our more “complex” instructions are fairly simple. The whole purpose of μ -code is that you want to simplify your logic but still be able to have complex instructions visible to the user.

An example

We now want to run through an example. I’ve drawn out a diagram of the initial state of a CPU. In class, we will run through and see what happens.



The real 68000

We will talk more about the real 68000 later. However, for now we can take a look at the 68000 model by looking at page 1-2 in the 68000 reference manual. There, you can see a diagram of the CPU that is very similar to the one given in this handout. Note the differences:

- There are more data registers available (D0 - D7).
- There are address registers available (A0 - A7). Address registers will provide a different way for the programmer to reference memory (rather than encoding the address itself in the instruction). As you can see from the diagram, A7 is special—we will learn why in a little while.
- The Z register has been replaced with the condition code register (CCR). The CCR contains the Z bit plus a whole bunch of other bits that indicate the current status of the CPU.