Lectures 15: Von Neuman, RISC vs CISC

1. Von Neuman machine recognizes that by assigning different meanings to bit patterns *depending on the state of the machine*, the next state can be dynamically altered. (Remember, we got from state to state based on inputs and current state).

1.1. Add these parts:
   1.1.1. PC points to next bit pattern that will be put into the IR
   1.1.2. IR is register that holds bit pattern that is to be decoded (the instruction)
   1.1.3. X, Z are registers needed so multiple things don't simultaneously try to write to the bus. (Also need registers to use 3-state devices, right? Won't work at this point using only the circuits we know about.)

1.2. Sequence of actions: F, D, E
   1.2.1. PC > Mem
   1.2.2. M > IR
   1.2.3. IR selects operation to perform Operation takes place on selected date (need to get data)

1.2.4. So, for example, how do we add 2 memory locations?
   1.2.4.1. PC of ADD inst > MAR
   1.2.4.2. ADD inst > IR
   1.2.4.3. IR decoded, appropriate state selected
      1.2.4.3.1. In this case, Address of first Operand put into MAR
      1.2.4.3.2. MDR => X (temp storage)
      1.2.4.3.3. Address of 2nd operand => MAR
      1.2.4.3.4. MDR => Z
      1.2.4.3.5. ACC <= Z, X; ACC => Z

1.3. Buses
   1.3.1. Our 1-bus machine requires multiple cycles to accomplish anything because of contention for the bus.
   1.3.2. Draw 2-bus machine (source, destination). This works better, but is still not perfect. Works well for A=A+B
   1.3.3. Draw 3-bus machine. Now, we can do instructions like A=B+C, and it will only take a single cycle!

1.3.4. Dealing with Bus Inefficiency
   1.3.4.1. 3-bus computer takes fewer cycles to execute instructions than 1-bus and 2-bus, but it still takes more than 1.
      1.3.4.1.1. Incrementing the PC requires the adder.
      1.3.4.1.2. Solve by putting a little adder up in the fetch area.
   1.3.4.2. Complex addressing modes require multiple trips to memory.
      1.3.4.2.1. Solve by limiting my instruction set to Load/Store architecture. If I say all ALU instructions can only use registers, and all memory instructions must be by themselves as well, then I can get instructions to execute in a single cycle!

2. The rise of RISC as CISCs compete
   2.1. 1970s has dozens of machines and instruction sets. Designers were adding instructions to instruction sets based on what they "felt" would help the code writers. Remember, at this point in time the state of the art of compiling was very limited, and a lot of code was either written by hand in assembly language, or written in a high level language and then optimized by hand.
   2.2. If a machine had a control store, and it was writeable, then in effect you could have had one piece of hardware execute *any* instruction set.
      2.2.1. Theoretically the poly instruction would make the compiler easier to write but it doesn’t because it was hard to recognize the poly situations in C source code.

   2.2.2. Some specialized instructions for case and switch were too restrictive to use well.
   2.3. In the late 1970's Dave Patterson at Berkeley spent a sabbatical at IBM, and they looked at how the worst case path affected processor performance. Essentially, what they found was that if you reduced the complexity of the instruction set, keeping only really simple instructions that could be implemented using a faster clock, then you could actually make a program run in less time.
   2.4. In 1981, John Hennessey at Stanford studied logs of CPUs running real programs, and found:
      2.4.1. Roughly half of all hll instructions are assignment statements.
      2.4.2. The vast majority of arithmetic expressions involve a single operation.
      2.4.3. Conditional jumps form the next most common group of instructions.
      2.4.4. Calls to procedures are also quite common. Most calls involve relatively few parameters. Deep nesting of procedures with little intermediate work is rare. The majority of variable references within called procedures are to local variables.
      2.4.5. Most instructions employ the simpler addressing modes.
   2.5. With CISC we are writing programs in a high level language, which gets converted to an assembly language in the compiler, which gets converted into microprograms in the CPU.
2.6. With RISC we have the compiler write the micro code directly.
2.7. Pipelining, which means adding registers.
   2.7.1. Optimized using branch prediction and delay slots.