CS 250 Midterm 1 - Fall 2022

schari

September 2022

1 Sections of textbook to review

- Chapter 3
- Chapter 4
- Chapter 2.17
- Chapter 5.1-5.10

2 What is Computer Architecture?

3 Information Representation

- Computers use a representation defined by a pair of symbols to represent all kinds of information
  - This is known as a bit, where a bit is defined as either a 0 or a 1
  - A bit string is an ordered sequence of bits
  - A byte is an 8-bit bit string
  - Ex: 01101001 is a byte and a bit string
- The reason computers use a 2-symbol representation is because it is easy to do so by controlling voltage; on is a 1 and off is a 0
- How do you represent a bit string electrically?
  - To represent a k-bit string electrically, you'll want k wires to each hold one bit of the string
  - A bunch of k wires carrying k bits for a k-bit string is a k-bit **bus**.
  - On a diagram, you typically see a bus represented as a single line.
  - A k-bit string can represent 2^k unique sequences
- Hexadecimal Notation

Table 1: Table between hexadecimal, binary, and decimal values

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1000	10
B	1011	11
C	1100	12
D	1101	13
E	1100	14
F	1111	15

  - To shorten the binary data that computers read, we usually use hexadecimal notation to view binary data
  - 4 bits map to one hexadecimal digit

- Prefixes for 2^k
  - Kibi is $2^{10} \approx 10^3$ (which is kilo)
  - Mebi is $2^{20} \approx 10^6$ (which is mega)
  - Gibi is $2^{30} \approx 10^9$ (which is giga)
  - Tebi is $2^{40} \approx 10^{12}$ (which is tera)
  - Note: you drop the last two letters of the 10^k prefix and add bi (for binary) to approximate the 2^k prefix
  - $2^{10} = 1024$ and $10^3 = 1000$

4 Computer Memory

- **Memory** is computer hardware that functions can write data to and read data from
- Memory contains locations where data is stored, as well as unique addresses that point to those locations
  - How do we define 2^k unique "points" in physical memory with k bits?

  - We delegate $2^{k/2}$ wires of the bit string as "horizontal" wires and the other $2^{k/2}$ wires of the bit string as "vertical" wires
  - This creates a grid with 2^k individual locations defined by k bits
- With this, we can define a pointer-mapping circuit that takes a k-bit pointer that maps to 2^k locations in memory
- What actually goes at each of these "locations"?
  - You would use a piece of circuitry called a **register**.
  - The register is made up of 4 parts: k 1-bit latches (1 latch for each bit), an enable line, an input bus, and an output bus
  - The output bus returns the contents of the latches (which each store one bit in the bit string)
  - The enable line tells the latch when to accept new values for each bit through the input bus (the latches won't change in value until we tell it to change)
- Pointing
  - To actually receive data from memory, we use a circuit called a **decoder**
  - The decoder has k wires as an input and 2^k wires as an output
  - Based on the input wires, the decoder will have one of the output wires carrying voltage, while the rest of them have no voltage
  - One of the applications of a decoder is to use the decoder as a pointer-mapping circuit that points a k-bit address to one of 2^k locations in memory
- The multiplexer (mux)
  - The multiplexer is a circuit that takes in an address and returns the contents of the location in memory that address points to
  - It is used to read data from memory
  - It has 2 inputs: an n-bit bus that represents the address of the register you want the data of, and 2^n k-bit buses that represent the wires connecting from the memory to the mux
  - The mux is given the address as input, then the mux retrieves the data from the corresponding bus
  - The mux then outputs the data through its k-bit bus
- The demultiplexer (demux)
  - The demultiplexer is a circuit that takes in an address and some data and writes that data to the location in memory that address points to

  - It has 2 inputs: an n-bit bus that represents the address of the register you want to write to, and a k-bit bus that represents the new data you want to store the value of
  - The demux points to one of the corresponding 2^n k-bit output buses and outputs the k-bit string to write the string to the corresponding location in memory
  - Essentially the inverse of the mux function; mux function reads information, while the demux function writes information

- When a bit string is transported from memory to the processor, it's called a fetch.

5 Processors

- The Harvard and Von Neumann architectures
  - The major difference between the two architectures is that the Harvard architecture has separate memory areas for holding the instructions and the data, whereas the Von Neumann architecture contains it all in one memory
  - Pros vs Cons of each:
    * Harvard
      · Pros: Simultaneous access of instructions and data, can optimize the memory design specifically for instructions and data
      · Cons: Two potential memory bottlenecks; may run out of memory for instructions or data
    * Von Neumann
      · Pros: Requires less memory; both of them occupy the same memory
      · Cons: Less secure because a given address could either point to an address or a piece of data
  - Nowadays, most processors use the Von Neumann architecture

- What are processors?
  - Many people use processor the same way as they use CPU, but they are not the same thing; processors are just a chip that can perform a multistep computation

- A general-purpose processor
  - Why would you want a processor that can do a lot of things vs one that does a few things very efficiently?

  - More cost-effective to manufacture a lot of a few processors than lots of different processors
  - The blueprint of a general-purpose processor contains a few things: a store of memory that stores data, a circuit that conducts computation (called an **ALU**, or Arithmetic and Logic Unit), and MUXes and DeMUXes to read data to the ALU and write the results of the computation back to memory
  - The ALU contains MUXes that point the operands passed in as the input data to the corresponding engines to conduct the computation (ex, an arithmetic engine, a graphics engine, etc)

- Representing machine code execution with a general-purpose processor
  - To execute a computation, you need an address pointing to the instruction that you wish to execute
  - This means that you have to store computation in memory
  - Some may choose to store their instructions separately from the data (which is Harvard architecture)

- Executing instructions
  - A simplistic model for how computers execute machine instructions is the fetch-execute model
  - Essentially, the computer fetches the next instruction, then executes it
  - What happens when there is nothing else to execute?
  - The software must figure out what the processor needs to execute next
  - In a dedicated processor, the main application will execute endlessly
  - In a general-purpose processor, the operating system will have an idle loop for the processor to run while waiting for the next process to be run

- Clock Rate and Processor Speed
  - Many circuits have a clock that paces the circuit's execution (how often should the processor execute an instruction?)
  - The registers and muxes/demuxes have a fixed delay between operations
  - The different functions in an ALU can have differing amounts of delay
  - Benchmarking a processor's execution
    * Benchmarking a processor's performance when executing a program requires 3 values:

- $\frac{Instructions}{Program}$ depends on the algorithm being executed (Software)
- $\frac{Clock\ cycles}{Instruction}$ depends on the compiler and circuit design (SW + Hardware)
- $\frac{Seconds}{Clock\ cycle}$ depends the worst-case delay (HW)
- The final fraction is $\frac{Seconds}{Program}$

6 Machine Instructions

- What operations should a processor be able to do?
  - What people want their processors to be able to do
  - It is a design challenge
  - People want their processors to do a lot
  - What they want the processors to do can change over time
  - The influence of different people can shift over time
  - Other considerations: technology, cost, adoption
- The set of operations a processor can do is the **instruction set architecture** (ISA)
- How do we go from HLLs to machine instructions?
  - HLLs might have instructions that doesn't correspond with a machine instruction
  - There might exist machine instructions that doesn't correspond to a HLL instruction
  - What matters is that every HLL instruction can be executed with a combination of machine instructions
- ISA considerations
  - The ability for an ISA to execute every possible program, Turing completeness takes little of the ISA's capabilities
  - What really matters
    - Programmer convenience: the ability for a few lines of code to do a lot of computation
    - The cost of a processor to execute so many instructions
    - Considerations like heat and speed of the circuit
- An ISA exists to control the general-purpose processor circuit

- Therefore, you pair a bit string with an instruction and configure the circuit to execute the corresponding instruction when it gets its bit string
- What goes in a machine instruction?
  - Composed of 3 parts: the opcode, the operands, and the result
  - All of these individual parts are represented by bit strings
  - Opcode
    - Required since it tells the ALU what to do
    - The encoding is up to the ISA designer, not portable between ISAs
  - Operands
    - Inputs of the computation
    - Number of operands defined by opcode
    - Pointers to the location containing the data
    - Sometimes contains extra information indicating the encoding of the data (e.g. 2's complement)
  - Results
    - Number of results defined by opcode
    - Pointer to location that will store the result of the computation
    - Not uncommon for a zero-bit pointer to be supplied
- Instruction size; two ways to design a length of a machine instruction
  - Variable length
    - Pros: can create as many instructions as you want; just add bytes, can choose short encoding for common instructions for quick execution, Easier to market as a feature
    - Cons: variable length means you have to expend extra computation time to get what the instruction actually is before you even compute it, compiler writers might worry about ISA compatibility, no one else really writes in machine code
  - Fixed length
    - Pros: Easier to fetch instructions from memory, decryption of instructions is simpler, criteria for instruction inclusion is largely technical since you can't add new bytes in the future, compiler writers like fixed length because then compiling to machine code is easier
    - Cons: instructions using only a few fields might not use all bits of fixed length string, cannot market "expanded ISA" because fixed length means you can't expand it
  - Both are measured in terms of bytes

- Instruction length and its growth
  - To represent each location of 4 GB of memory, you need a 32 bit pointer
  - For 16 GB of memory, you'd need 34 bit pointer
  - The number of bits you need to use to represent your memory increases as more memory becomes prevalent as a result of Moore's Law
  - $\therefore$ general-purpose processors that use 32 bit pointers or less have become obsolete
  - Variable length ISAs can adapt by adding an "address extension"
  - Fixed length ISAs have more limited options
- Constant pointer sizes?
  - Idea is to have a small number of registers that the ALU can directly access instead of accessing from memory
  - This means pointers can remain small while also increasing amount of memory
  - You'd typically see two kinds of registers: general registers for bit strings and integers, and floating point registers for floating point values (IEEE 754)
- Fixed length memory access
  - There are two instructions for getting memory into the registers: LOAD and STORE
  - You would LOAD the information from memory into the registers, tell the ALU to use the registers for the computation, then STORE the result in the memory
  - Using these register banks makes a more complicated circuit than one without register banks
- Examples of ISA used
  - Historic: PDP-1, IBM 360, DEC VAX, x86
  - SPARC (what `lore` uses)
  - x86-64 (aka amd64, x86_64, x64, Intel 64)
  - PowerPC
  - ARM
- Early ISAs
  - Memory is expensive; locations store one byte

  - Chips have limited pins, so limited I/O
  - Wanted to keep instruction bit strings short
  - 8-bit pointers, have specialized 0-bit registers
  - Programmers would write programs in machine language, so they would want some sophisticated instructions (could also lead to feature creep)
  - Wanted to have many different instructions
  - They would give instructions encodings in increments of 8 bits
  - More used instructions would be given shorter encodings
- x86
  - Became very successful early on thanks to inclusion in IBM PC
  - Would receive several expansions to modernize it
  - Various extensions patented by both Intel and AMD
  - x86-64
    * 64-bit extension of x86 architecture
    * x86-64 is owned by AMD, but x86 owned by Intel, so both are linked to ISA
    * Variable length from 1 to 15 bytes
    * Circuit has large propagation delay and large power consumption that doesn't improve computation performance
    * However, it's so widely used that it's still used nearly 50 years later

7 Why Assembly?

- Downsides of Assembly vs HLLs
  - Assembly is tightly coupled to processor; not portable
  - Each line of assembly does very little so you need many lines
  - You have to manually work with memory addresses and registers that HLLs abstract away from you
  - $\therefore$ HLLs seem more convenient and powerful to use
- Why you would want to program in Assembly
  - Writing more efficient and smaller code
  - Writing the core of an operating system
  - Writing a compiler for a new HLL
  - Speeding up computations for AI, ML, GPUs
  - Searching for security vulnerabilities
  - Interfacing with I/O devices