Exam 1

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Instruction set architecture

- What is an instruction set? What is an ISA? What are some important features that can be used to characterize an ISA? **The instruction set is the set of instructions that the CPU can handle. ISA refers to the architecture of the instruction set for a CPU, how it is organized, how each instruction works, number of bits per instruction etc. An ISA can be stack based, register based, or accumulator based. The main part of the ISA is the set of instructions it**

contains. Instructions can be characterized by number of operands, size of operands, and types of operands.

- Explain how stack, accumulator, and register architectures work and how they differ. What are their advantages and disadvantages? Which one is most common today? **A stack machine uses zero and one operand instructions. The stack is an implicit operand. Load instructions load from memory onto the top of the stack. Store takes the top of the stack and puts it into memory. Arithmetic instructions pop and push onto the stack as needed.**

  Accumulator based machines use zero and one operand instructions. The accumulator is a register. Load loads a value from memory into the accumulator. Store takes the accumulators value and stores it in memory. Arithmetic instructions like add change the value in the accumulator. The accumulator is an implicit operand in some instructions.

  Register based machines use general purpose registers instead of a stack or a special register like the accumulator. Users are free to use these registers to hold data or results from arithmetic operations before they are stored in memory.

- A stack machine and an accumulator machine both commonly have one address. How do they differ?
- What is postfix (reverse Polish) notation? Which type of architecture requires it? Why? **Postfix is a way of writing arithmetic expressions where the operator comes after the operands. Postfix requires stack architecture because you need a stack to parse and evaluate the expression. The stack is required to hold terms before a operator is encountered.**
- What does GPR stand for? What are three types of GPR machines? How do they differ? **GPR stands for general purpose register. The three types are**

  1. **Memory-memory: Two or three operands may be in memory**
  2. **Register-memory: At least one of the operands must be a register**
  3. **Load-store: No operands may be in memory except for LOAD and STORE instructions**

  They differ in speed. Load-store is faster than register-memory, which is faster than memory-memory. The less we need to go out to memory the faster the system because of less bus traffic.

- How does the ISA affect the instruction length? Do longer instructions necessarily lead to longer programs? Why or why not? **The ISA determines the instruction length because it defines the instructions, instruction types, instruction length, type of machine, type of operands etc. Longer instructions do not necessarily lead to longer programs. Instruction length usually depends on CPU operating model, which determines the number of operands for an instruction. The length is also determined by address length.**
- What is the maximum number of fetch operand steps that instructions in each of these machines can have? Explain why for each case.

  (a) **A stack machine.**: One, instructions have at most one operand, the other operand is the stack which is implicit and not in memory.

(b) **An accumulator machine.**: One, instructions have at most one operand, the other is the accumulator register which is implicit and not in memory.

(c) **A memory-memory machine where instructions have at most 2 operands.**: Two, need to fetch from memory twice, for each operand in the instruction.

(d) **A memory-memory machine with some 3-operand instructions.**: Three,

(e) **A register-memory machine where instructions have at most 2 operands.**: One

(f) **A register-memory machine with some 3-operand instructions.**: Three

(g) **A load-store machine.**: Zero, operands are registers, no need to fetch from memory.

- A stack machine can compute any arithmetic expression without using temporary variables. An accumulator machine or a register machine cannot. Explain why. Hint: how would each of these machines calculate a formula such as (a+b)/(c+d) ? **A stack based machine holds intermediate results in the stack, whereas we would need to store these results in memory or registers (temporary space) in non-stack machines. For example, in a stack based machine $(a+b)/(c+d)$ would first be converted to postfix $ab+cd+/$, which would look like**

  PUSH a
  PUSH b
  ADD
  PUSH c
  PUSH d
  ADD
  DIV
  STORE ....

  The way the stack works with postfix requires no temporary space to hold intermediate results in the expression. In an accumulator based machine,

  load a
  add b
  store tmp1
  load c
  add d
  store tmp2
  load tmp1
  div tmp2
  store res.

  Notice that since all arithmetic happens in the accumulator, we first need to load a to the acc to add b to it, then $a+b$ is in the acc. But, we need the accumulator $c+d$. If we load c into the acc, we loose $a+b$, so we store $a+b$ in a tmp variable. Once we have $c+d$, we store, load back $a+b$, then divide by $c+d$, and store in res. This took two temporary variables

  In a GPR machine we would need one register to hold $a+b$, and another to hold $c+d$, and depending on the type of GPR machine, we may need to store these intermediate results in memory, for example if we did not have DR (register-memory).

Pipelining

- What is pipelining? Why does it speed up execution? Why can't we keep the pipeline full all the time? **Pipelining is when we divide the fetch-decode-execute cycle into smaller steps, which can be executed in parallel to increase throughput.**

  This speeds up execution by allowing multiple instructions to execute in parallel.

  The pipeline cannot be full all the time because one instruction may take more cycles than expected.

- What is a pipeline hazard? a stall? flushing the pipeline? What are three kinds of pipeline hazards? Define them. **Stalling is when the pipeline cannot be kept full because one instruction took more cycles than expected. Flushing is when no instructions can enter the pipeline until the current instruction is finished executing. Pipeline hazards are conflicts that arise that cause stalling or flushing. The three kinds of hazards are**

  1. **Resource conflicts.: When an instruction needs a resource that a previous instruction still in the pipeline is using**
  2. **Data dependencies : When an instruction needs the result of a previous instruction still in the pipeline**
  3. **Conditional branching: When an instruction causes a branch, we need to know whether subsequent instructions are to be executed or skipped, depending on the result of the instruction that causes the branch.**

- How can an optimizing compiler eliminate some types of pipeline hazards? **Can eliminate redundant instructions, or use different registers for operations we may want to execute in parallel.**

Endianness

- What is the difference between little-endian and big-endian architecture? Give an example of each type of machine. **LE machines store the least significant byte before the most significant byte. The lower order bytes are stored in addresses lower than the higher order bytes. Intel architectures are LE**

  A BE machine is the opposite. These machines store the MSB before the LSB. IBM mainframes are BE

- Why is endianness important in I/O and networking? **When sending a file from one machine to another, or in networking protocols, both machines / systems need to agree on the byte ordering to send / read data so that the machine receiving the data can know how to read it.**

ASCII encoding

- How many characters can be represented in ASCII? How many bits does it take to express those characters? Why are there multiple “extended ASCII” schemes? How many characters does each of those schemes have? How many bits does it take to express those characters? What is wrong with having multiple extended ASCII schemes? **ASCII can represent 128 characters. It is a 7-bit scheme, so** $2^7 = 128$ **possible bit patterns.**

  By extending to 8 bits (one extra bit), extended ascii can provide an additional 128 characters, so $2 \cdot 128 = 256$ **characters. There are many extended ascii schemes because different regions require a different set of characters. This is a problem because we need a standard. Otherwise, one machine can send data to a different machine, and they wont agree on what the characters mean because they are using different extended ascii schemes.**

Unicode

- What is Unicode? How does Unicode solve the "extended ASCII" problem? **Unicode is a newer model that can encode the characters of most languages in the world. This way we can have a standard so that every machine can agree on what the characters mean. One model that everybody can use.**

- What is a Unicode code point? How many of them are there? How are they written? **A unicode character is called a code point.**

  Code points are numbered from $0000 - FFFF$**, and** $10000 - 10FFFF$**, minus** $D800 - DFFF$**. These are split amongst 17 code planes. Code plane 0 handles code points 0000-FFFF, which is** 2^{16} **code points. The remaining 16 code planes (1-16) handle code points 10000-10FFFF, which is** 2^{20} **code points. Since these 16 code planes have a total of** 2^{20} **code points, each of these planes has**

  $$\frac{2^{20}}{16} = \frac{2^{20}}{2^4} = 2^{16}$$

  code points. Code points $D800 - DFFF$ **are removed, so** $2^{11} = 2048$ **are removed. Notice the last byte ranges from 00-FF** $15 - 7 = 8$ **times, so the number of code points removed is** $2^{11} = 8 \cdot 2^8 = 2048$**. Therefore, there are**

  $$2^{16} + 16 \cdot 2^{16} - 8 \cdot 2^8 = 17 \cdot 2^{16} - 8 \cdot 2^8$$

  code points. These code points are prefixed with $U+$**, so the code points range from U+0000-U+FFFF, and U+10000-U+10FFFF, minus U+D800-U+DFFF.**

- What is a Unicode code plane? How many of them are there? How many code points are there in a code plane? **The code points are split amongst 17 groupings, called code planes. There are** 2^{16} **code points per plane, as stated above.**

- Which code plane is the basic multilingual plane? Which code points does it contain? Why are some code points omitted, i.e., why is the BMP discontinuous? Why is the BMP important, i.e., why do most Unicode users never need to look outside the BMP? **The BMP is code plane 0. This plane contains code points U+0000 through U+FFFF, which is ascii, extended ascii, and some other symbols and languages like mathematical symbols and the Chinese and Japanese languages. The omitted code points are U+D800 through U+DFFF. These omissions make the BMP discontinuous. They were omitted so that they can be used as control characters. The BMP is important because it is where all of the most common languages / characters are. The other planes are supplementary and have characters that are rarely used. Many code points in these supplementary planes remain unassigned.**

- Which code points match the 7-bit ASCII characters? Which standard do the code points from 128 to 255 match? **U+0000 through U+007F match the 7-bit ascii characters. The code points from 128 to 255 (U+0080 - U+00FF) match extended ascii, specifically ISO-8859-1.**

UTF

- What is the difference between Unicode and UTF? What is UTF-8? What is UTF-16? What is UTF-32? What is self-synchronization and why is it useful? Why are there a few missing code points in the charts in the slides (hint: see slide 90b4)? (You won't have to do computations based on these charts.) **Unicode is a content standard, whereas UTF is an implementation standard. UTF defines how the computer encodes and represents the code points in memory.**

  The difference between UTF-8, 16, and 32 is the number of bits they are based on,

  - **UTF-8: 1-byte scheme**
  - **UTF-16 2-byte scheme**
  - **UTF-32 4-byte scheme**

  Self-synchronization is used to describe encodings with the property that byte boundary's can be easily located by examining the byte stream. UTF-8 is an example of this because each byte is either a leading byte, or a continuation byte. If an error occurs while reading the characters, the parser can simply back up or move forward until it finds the previous or next leading byte.

  The missing code points are used in UTF-16 to determine whether a 16-bit value is a single code point or part of a 4-byte pair.

- Why are there multiple UTF options? Which one(s) are fixed length? Which one(s) are fixed length for code points in the BMP? Why are fixed length formats useful? Why are variable length formats useful? **Some UTF options can be more convenient that others, depending on the use case. The fixed length implementations are UTF-16 and UTF-32, the variable length implementation is UTF-8. Regarding only code points in the BMP, both UTF-16 and UTF-32 are fixed length. Fixed length formats are useful because they allow for simple parsing, variable length formats are useful because they can save space, the characters only take up as much space as they need.**

- Which UTF formats have an endianness issue? Why? What is a BOM (byte order mark)? Is a BOM always required? What is the default if a file has no BOM and there is no other way to identify the encoding? **The formats with an endianness issue are UTF-16 and UTF-32. In UTF-16, bytes come in 16-bit units, so order matters, same with UTF-32, 32-bit units. In UTF-8, each byte is it's own 8-bit quantity, so order has no meaning.**

  The BOM is a byte at the start of a file that determines which ordering to use. The BOM is not needed for UTF-8 since ordering does not matter. The default is BE

- Some software manufacturers provide a "BOM" on UTF-8 files. Why is this not truly a BOM? What problems does this cause? **The "BOM" is used to indicate UTF-8, it is not truly a BOM because BOMS are used to indicate ordering, and ordering does not matter for UTF-8. This causes a problem because now UTF-8 files that match up one to one with ASCII no longer match.**

- What does Microsoft do with UTF-16 files that is different from the standard? What problems can this cause? **They say that a file with no BOM is defaulted to LE, this is a problem if you create a file on a windows machine, and want to use it on a non-windows machine that assumes default is BE.**
- Which encoding is most prevalent on the Web? What format do each of the following use internally: Linux, Windows, Java? **UTF-8 is the most prevalent on the web**
  - **Linux: UTF-8**
  - **MAC: UTF-8**
  - **Windows: UTF-16**

Floating point concepts

- What are some advantages of floating point over fixed point? **Floating point has the advantage over fixed point which is the fact that the radix point can be placed anywhere, not at a fixed place. In fixed point, the radix point is fixed at one place. Suppose the radix point is placed such that there are 4 bits before the point, and 4 bits after the point. If most of the bits before the point are zero, then we are fully taking advantage of all 8 bits. In floating point, we can place the radix point in the most optimal position, depending on the number. Floating point is also based on scientific notation, so some bits are used to scale the number, this let's us achieve both really small and really big numbers.**
- Why is floating point not generally used in accounting? **Because floating point introduces errors, and is not completely accurate. With floating point, rounding error exists if a number cannot be represented exactly. In accounting, it is better to use integers to store fractional values likes pennies.**

Floating point range

- How many bits are in each field in the IEEE 754 single precision standard? double precision standard? **For single precision,**
  - **Sign: 1 bit**
  - **Significand: 23 bits**
  - **Biased exponent: 8 bits**

  For double precision,
  - **Sign: 1 bit**
  - **Significand: 52 bits**
  - **Biased exponent: 11 bits**
- What is the purpose of the bias? How is the bias calculated? What is the smallest exponent (true value) in single precision? What is the largest? What is the biased form (binary string) of each of these values? **The bias is used to take the unsigned stored exponent, and shift its range such that negative exponents are allowed. It takes a unsigned range and shifts it so that the range matches its signed range. The bias is**

$$2^{n-1} - 1.$$

  In single precision, the smallest stored exponent is 1: 0b00000001, the largest is 254: 0b11111110.
- What is the smallest normalized number in single precision? What is the largest? How did you get that answer? **Smallest is given by**

$$(-1)^0 \times (1.000...0)_2 \times 2^{1-127} = 1 \times 2^{-126} = 2^{-126}.$$

  Largest is

$$\begin{aligned}(-1)^0 \times (1.111...1)_2 \times 2^{254-127} &= 1 + \sum_{k=1}^{23} \frac{1}{2}\left(\frac{1}{2}\right)^{k-1} \times 2^{127} \\ &= 1 + \left(1 - \frac{1}{2^{23}}\right) \times 2^{127} \approx 2 \cdot 2^{127} = 2^{129}.\end{aligned}$$

- What is the smallest denormalized number in single precision? What is the largest? How did you get that answer? **Smallest is**

$$(-1)^0 \times (0.000...1)_2 \times 2^{-126} = \frac{1}{2^{23+126}} = \frac{1}{2^{149}}.$$

  Largest is

$$(-1)^0 \times (0.111...1)_2 \times 2^{-126} = \left(1 - \frac{1}{2^{23}}\right) \times 2^{-126} \approx 2^{-126}.$$

- What is floating point normalization? What is the main benefit of normalization? What is the extra benefit we get from not representing the initial 1? **Normalization is when we take a mantissa, shift the binary point, and update the exponent such that the only digit to the left of the point is a one. This let's us have a unique representation for each number, and getting an extra digit of precision, since we do not need to store the implicit one, although it is there.**

- Make sure you can recognize the 5 special cases on slide 2j-71a. What is NaN? Why is it useful to have a specific value to represent it? Why is it useful to have a specific value to represent infinity? **The five special cases are**

  1. **Zero: When the biased exponent is zero, and the significand is zero, the value is** ± 0
  2. **Denormalized number: When the biased exponent is zero, and the significand is non-zero, denormalized number.**
  3. **Infinity: When the stored (biased)** n**-bit exponent is** 2^n**, and the significand is zero, infinity**
  4. **NaN: When the stored (biased)** n**-bit exponent is** 2^n**, and the significand is non-zero, NaN**
  5. **Normalized number: When the stored (biased)** n**-bit exponent is between** 1 **and** $2^n - 2$**, normalized number.**

  NaN is a special value that some computers use to indicate an error. It is useful to have a value represent it so that the compiler may initialize memory to NaN so that it can tell when a variable has not yet been assigned a value. Also, so that we can do calculations with NaN without the program crashing.

  Useful to have a value represent infinity so that we can do calculations with it without the program crashing.

- What are denormalized numbers? How do denormalized numbers allow for a smooth transition from the smallest normalized number to 0? Why is that a good thing? **A denormalized number is a number with the smallest exponent, and without the implicit one. This let's us represent very small numbers, and degrade slowly as numbers get smaller, instead of a hard jump from representable number to underflow.**

  They allow for a smooth transition because they are very small, and with constant (very small) spacing.

- How can you recognize a denormalized number? What value does an exponent of 0 represent? Why don't denormalized numbers have an explicit 1? **A denormalized number has a biased exponent of zero, and non-zero significand. A exponent of zero represents a denormalized number. No implicit one because we cannot change the exponent, it is fixed at** $2^{n-1} - 2$

Floating point error analysis

- Why is floating point error unavoidable? **Because some numbers cannot be represented in binary exactly. The only numbers that can be represented exactly are of the form**

  $$\frac{k}{2^m}.$$

  Where k, m are integers. Thus, the denominator must be a power of two. For example, $\frac{1}{10} = 0.1$ cannot be represented exactly because the denominator is not a power of two, so it cannot be represented exactly in binary. This means we will have roundoff error.

  Even if a number can be represented in binary exactly, we must have enough bits to store each digit. If we don't have enough bits, there will be truncation error.

- Why is it not safe to compare floating point numbers using '=='? What is a workaround for this problem? **Because of roundoff error, if a number cannot be represented exactly, it is rounded to the closest representable float. If we compare a number to a float that was rounded, they will not be equal.**

  A fix is to compare the difference against some tolerance. This way the roundoff error is accounted for.

- What is floating point overflow? What is floating point underflow? **Overflow is when a number is too large, the exponent is too large to be represented by a floating point number. Underflow is when a number is too small, and cannot be represented exactly as a denormalized number or is rounded to zero.**

- What is the range of a representation? What is accuracy? What is precision? How does normalized floating point allow us to maintain both precision and accuracy to the extent possible, i.e., why would we lose both precision and accuracy if we didn't bother with normalization?

- How does denormalized floating point allow us to maintain both precision and accuracy to the extent possible for very small numbers, i.e., what would happen if we didn't have denormalized floating point as an option?

Assemblers

- What does an assembler do? What is the difference between source code and object code? **An assembler takes assembly instructions and converts them to machine code. Source code is human readable assembly instructions not yet compiled into machine code. Object code is partially compiled source code, but external references have not yet been resolved.**

- What does the linker do? What is the output of the linker? **The linker takes one or more object files, resolves external references, and combines them into a single executable (machine code). The output is an binary executable (machine code)**

- Why do assemblers usually have at least two passes? What is a symbol table? How is the symbol table used in a typical two-pass assembler? **The assembler has at least two passes so that they can create machine code without references resolved, while building the symbol table so that during the second pass, with a completed symbol table, the references can be resolved and the machine code can be completed.**

  A symbol table is a map that maps symbol to their addresses. In a two pass assembler, the assembler builds the symbol table during the first pass. Once it first encounters a new symbol it puts the symbol in the table, and a zero for the address if the symbol is defined later in the code. Once the assembler locates the definition, it fills in the address. During the second pass, the symbol table is complete, so all unresolved references in the machine code can be resolved.

- What is a label? An assembler directive is an instruction that does not generate executable code, such as BIN and DEC. What do those instructions do? **A label is a name for an address of an instruction. It allows us to use the name of the label to reference the instructions address instead of the address. BIN tells the assembler that the defined data is in binary, and DEC tells the assembler that the defined data is in decimal.**

- What is relocatable code? What is absolute code? When is absolute code used? **Relocatable code is machine code where the address of operands is relative to where the program was loaded.**

  Absolute code is machine code that uses fixed, hard-coded memory addresses for the operands. Absolute code is used when the program is always at a fixed place in memory.

Linkers

- Why do linkers usually have at least two passes?
- What is static linking? What is dynamic linking? What is the difference between them? What is an advantage and disadvantage of each? **Static linking is when the linker combines multiple object files into a single executable, resolving references to all external symbols.**

  Dynamic linking is when any module with a function that can be called is loaded into a DLL. As the program runs, when an external function is called, the system finds the correct module in the DLL and calls the function

Compilers

- What is the difference between an assembler and a compiler? What is the difference between a compiler and an interpreter? Why is the latter difference sometimes muddy in today's world? **An assembler takes assembly instructions and converts to machine code, whereas a compiler takes code written in a higher level language, converts to assembly code first, and then into machine code. A compiler compiles code into an executable, and then the program is run. An interpreter produces and executes machine code in real time, as the program is running.**

- What are the standard six phases of the compiler? Which are the only phases that need to be revised to retarget a compiler, i.e., to produce a compiler for the same language for a different machine?

  1. **Lexical analysis:**
  2. **Syntax analysis:**
  3. **Semantic analysis:**
  4. **Intermediate code generation:**
  5. **Optimization:**
  6. **Code generation:**

  The last two

- What are two kinds of optimizers? What is the difference between them?

  1. **Pinhole: Uses a sliding window to look at a small number of consecutive instructions**
  2. **Global: Looks at the bigger picture**

Java

- What does the JRE (Java Runtime Engine) do? What is an advantage and disadvantage of using an intermediate representation vs. compiling to binary? **The JRE provides the environment necessary to run Java programs. It includes the JVM, core libraries, and supporting components that allow compiled Java bytecode to execute on any platform with a JRE installed.**

  Using an intermediate representation such as Java bytecode enables platform independence: the same compiled program can run on many different systems without recompilation.

  An intermediate representation typically introduces performance overhead because the code must be interpreted or JIT-compiled at runtime rather than executing directly as native machine instructions.

- What is Java bytecode? **An intermediate representation between java code and machine code that the JVM can interpret while the program is running.**

- What does "write once, run anywhere" mean? **Java code can be written once on one machine and run on any machine / platform that has the JRE. Platform independent**

- Can languages other than Java use the JRE? **Yes**