Computer Architecture: Honors

Neo Wang
neow@utexas.edu

April 4, 2024

Abstract

These are notes for UT's CS 429H Computer Architecture, an upper-division class taught by Ahmed Gheith in Spring 2023. The class covers how the different parts interact, representations of programs and data, programming devices, systems programming, computer micro-architecture, and performance optimizations. The class covers C, C++, Assembly/Machine-Language, and Verilog. A link to the textbook can be found here. A good reference sheet can be found here. The teaching assistants for this course are Julia Benginow, Matthew Giordano, Neil Allavarpu, and Nikita Sharma.

Contents

1 Week 1

1.1 Lecture: January 12th, 2023

Gheith mentions the software is the program that animates hardware and makes it do exciting things. Without hardware, software is a dream. Without software, hardware is a heater. We will look at how software/hardware interactions help facilitate the development each other. Gheith then states our number one objective is to learn the interaction between software and hardware.

1.1.1 Computing Units

There are several different names for compute: processor, CPU, Core, hardware thread. Historically, it was called the CPU because it was the only processor.

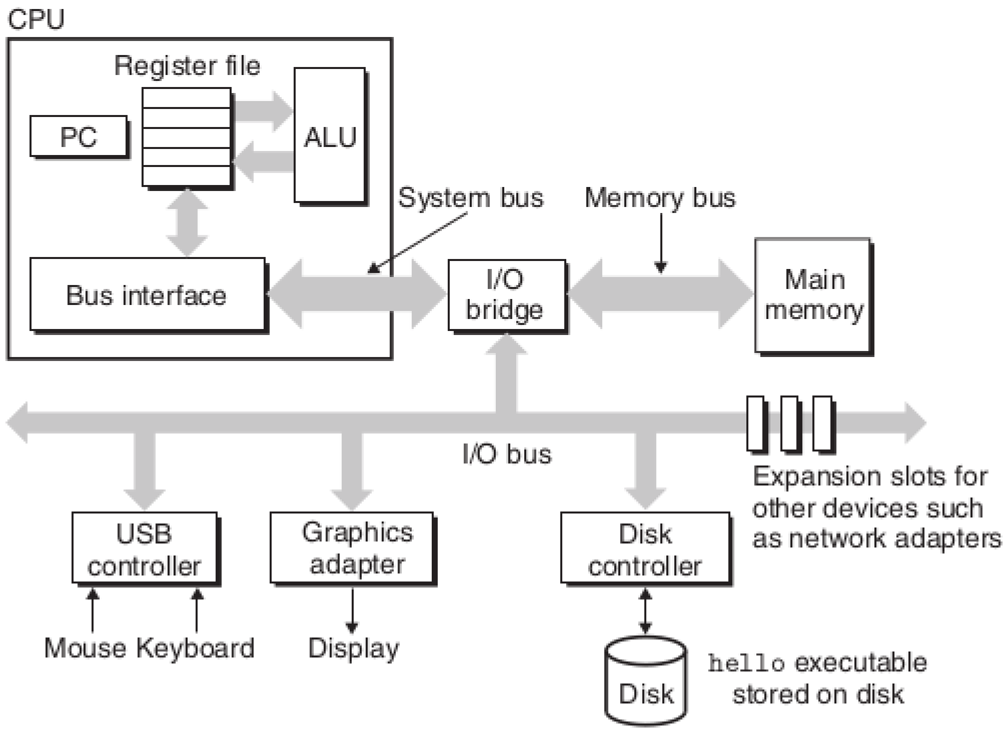
The **Arithmetic Logic Unit (ALU)** is a piece of hardware. To draw this, we have three wires a, b and a control that go into it. The output of this is an output c. To denote the number of wires going in/out, we can write a number. For example, for a 64-bit number, we can write 64 over the arrow going into the block. The control manages the operation, for example $001 = +, 010 = -, 011, *, 100 = /$. We refer to the transport from one unit to another as a **bus**. These can be drawn as bidirectional arrows. At this level, we refer to this as a **system bus**. This allows the CPU to communicate with the rest of the computer.

An analogy we can use for memory is a large storage facility for bytes. In each of the bytes, you can either read or write. Think of memory as a large array.

Programs that want to read and write from memory go through the **Load Store Unit (LSU)**. Load is another term for read and store is another term for write.

The **Input-Output (IO) Bridge** will send any address that does not fit in the memory range to the **IO Bus**. The IO bus communicates with the outside world such as a keyboard or a mouse. Each device has a range of numbers on the bus. Devices have side-effects. Example: speaker manufacturer says if a signal is sent to a certain channel then it will play that frequency.

The **Control Unit** decides what we "do next" after we finish the current instruction. We can think of the architecture as a construct of bridges and the bus as a highway.



Instructions are a language that the hardware speaks. It is a method of encoding actions. CPUs speak a set of instructions called the **instruction set**.

Example: We have eight bit wide instructions. Opcode is the instruction type or desired operation and is embedded within the instruction itself. CPU's job is to find where the next instruction is to run. The CPU has no memory, so it first **fetches** the instruction from memory. If there is a storage element sitting inside the core, we call it a register. As a general rule, if memory is big, then it is slow. When we want to process the instruction, we **first fetch the instruction, decode it, then execute it.** Then, we go to the core, and then update the **program counter (PC).**

1.1.2 Overview of Computer Architecture and Terms

Jump instructions are special in that instead of $PC = PC + 1$ the jump counter makes it do $PC = PC + n$. There are also conditional jumps. The optimizations for the architect is to design this set such that the hardware is both fast and easy to write software for. The set of all possible instructions is the **instruction set architecture**. Computer Architecture is a lot about designing those instruction sets. Machine Language is the binary encoding of a program that you need to put in memory such that when the CPU starts executing instructions, it is able to do what you want it to do. How do we point the core towards the entry point? The loader is what puts the program in memory. We find the main and put its address into the program counter. Different instructions sets like x86 will load the instructions before the CPU starts. Two wires in parallel will affect each other because of electromagnetic interference. Computers can take variables from memory and bring them to the CPU, perform the computations and send them back.

A major breakthrough was translating the opcodes into English and then translating them back to machine language. This is what is now known as **assembly language**. Assembly is a human readable form of machine language. Write characters instead of bits. We rely on a program called **assembler** to translate assembly to machine language.

A **compiler** (in machine language) takes a programming language and converts it to machine language. Modern compilers are extremely good and can write code comparable to assembly if not more efficient. To create new iterations of compilers, we feed it through old compilers–the first one still has to be written in a "lower" language. "These days we have the AI things to spit out the best programs ever (hint: it doesn't ever spit out a good program)." –Gheith

1.2 Discussion: January 13th, 2023

By default, **C does not make assumptions on your integer size.** In order to make sure your code runs on all computers, define the size. In C, you can cast any type to any type. With C you can do almost anything.

Memory is an array of bytes. Typically memory is referred to in base-16. Memory has an upper bound and a lower bound. The ampersand operator will give the address of a variable. For example, a function to modify a number will have a dereference operator in order to get the values and overwrite the original. String literals in C have an implicit trailing zero. However, if you remove the double quotes there is no sentinel value.

In C, we call malloc in order to get new memory. In Java, the counterpart is new. The memory could be anything until you set it.

2 Week 2

2.1 Lecture: January 17th, 2023

An interpreter does what the program wants to do instead of using the machine language. **Just-in-time compiling** is able to run machine language by generating it on the fly for certain high-frequency functions. Today, many languages rely on just-in-time compiling. To describe hardware, we can use a Hardware Design Language (HDL). In this class we will learn **Verilog**. In a sense, C only has static methods.

Suppose we want to take a function that takes two longs x, y and return their difference $x - y$ to machine language. If we compile gcc with the `-S` flag we can see the function in machine language. If we use the command `objdump -d x.o` we are able to see the assembly and machine language representation. For example, we have a instruction called `retq` which is `c3`. retq is a return statement. When we compile with O2 flags we can take a function from 19 instructions to 7. mov copies data between registers. There are two popular assemblers for x86. There is AT&T and Intel. Most x86 instructions that is on the internet uses the Intel syntax. **We will use the AT&T syntax** because it is the default for Unix. The percent signs in assembly syntax means register. For example, `%rdx`. There are several registers: `%rax, %rdx, %rcx, %rbx, %rsx, %rdi, %rbp, %rsp, %r8, %r9, …, %r15`. There are general purpose registers which are referred to as **GPRs**. In x86 we will have 16 of them. SP stands for stack pointer. The stack is where local storage is. For every architecture, ask yourself where the stack pointer is–this is very important! Every architecture calls the PC something different. x86 calls it `%rip` which stands for **instruction pointer register.** This style of assembly is called a **two operand instruction set** where we always have **one source, one destination.** General purpose registers are generally used to store pointers, integers. There are dedicated registers for floating point numbers.

The next instruction we discusses is `sub`. Here, we perform sub %rsi, %rax. The source is %rsi and the destination is %rax. This expands to first performing %rax = %rdi and then %rax = %rax - %rsi. By convention %rax is a return value. x86 does not have a three-operand instruction so it cannot generate %rax = %rdi - %rsa.

2.1.1 Linkage Conventions

There are linkage conventions which are the set of rules we have to use to create functions and have dependent code work together. At around 6, it begins hard to perform functions. Standardized tests like spec would measure the performance of an assembly language. Lots of architectures would use optimizations like O5 in order to attempt to get higher scores on standardized benchmarks. A **cross-compiler** is a compiler that produces binaries for a different architecture. For example, there is a fast x86 machine and compile to slow ARM machine. ARM has a three operand architecture.

For ARM, we have xpc as the program counter, and registered are numbered from x0 to x31. The linkage convention is also "very nice." When you enter the function, x0 is the first argument, x1 is the second argument, and so on... and has the same rule as x86. At some point we have to stop pushing arguments in and move them to the stack. In ARM, every instruction requires four bytes. In x86, there are variable length instructions. x86 is one of the few architectures with variable length. In order for C++ to support overloading, it has to change the name. Therefore, the header contains _Z2f1ll.

2.2 Lecture: January 19th, 2023

Companies like Microsoft will have multiple linkage conventions. The reason is so we can share binaries. Shared prepackaged libraries are only binaries. Another reason is so multiple languages can work together–if all the compilers agree on convention then they can work together. Libraries such as linear algebra libraries can experience great performance gains from handwritten assembly. **Function inlining** is where the compiler sees a function is "pasted" instead of using returns since return has some overhead.

2.2.1 Information Types and Sizes

Today, we will write a function that returns a big value. We have primitive types in C: **char, int, long, short, long long**. Characters in C are represented by a number which maps to a character. Some popular mappings are American Standard Code for Information Interchange (ASCII) and Extended Binary Coded Decimal Interchange Code (EBCDIC) were popular. Both were American and used eight bits per character. This is fine for the American alphabet. However, for countries like China, this might not work. In C, the **default character is 8 bits.** In C, it's up to the compiler to pick the size of an int–this means the types are not portable. Ints in C are $\geq$ 16 bits. A short is $\geq$ char width and $\leq$ int width. C intends for the long to be the size of a register and $\geq$ int width. The popular defaults are 32 bit integer, 16 bit short, and 32 or 64 bit longs. In Java, integers are always 32 bits, long is 64 bits. In a 32 bit architecture a long is 32 bits and in a 64 bit architecture a long has 64 bits. A long long is $\geq$ long. Generally, this is 64 bits in C. Java attempted to make it 128 bits. These inconsistencies make it inconvenient for portability. Java is popular because it has unicode built in and is relatively portable.

A new set of type names have evolved from these. These come from stdint. The types are `int16_t = 16; int32_t = 32; int64_t = 64; int128_t = 128.`

A **struct** is a collection of members that **move together**; their memory is continuous. This implies a starts at 0 and b starts at 8. Structs only have data and do not contain methods. In Java, this is abstracted away.

Now, we wonder what happens to the following code since our return value does not fit in one register:

```c
#include <stdint.h>
struct Big {
    int64_t a;
    int64_t b;
}

// if we do int Big = 100000; this is fine in C
// (there is a separation between Big and struct Big)
// Gheith claims this is a bad design decision

// we now introduce a struct (short for structure)
// our goal is to return some big value
struct Big happy(int64_t x) {
    // in java: Big big = new Big(); // objects live in the heap
    struct Big big; // you get what you asked for--instance of Big
    big.a = 10;
    big.b = 20;
    return big; // this returns a copy
}
```

Objects always come from the heap in Java. We also do a compare and contrast from C++ to Java. The below is the C code. **An activation record (AR) is a private block of memory associated with an invocation of a procedure (Google).** Activation record and stack frame are the same in C.

```c
Big happy() {
    Big b; // b is the actual object (no distinction between b and Big())
    // once the function returns, b disappears.
    big.a = 10;
    big.b = 11;
    return big;
}
```

Below is the Java code.

```java
Big happy() {
    Big b = new Big(); // b is a reference, Big() is the object
    // and doesn't die until garbage collected; the object lives on the heap
}
```

Stack is a natural representation. This model breaks for concurrent program and other parallel compute. The **stack pointer** is going to pointing to the top of the activation record. We can find a local variable by computing an offset to the stack pointer. Sometimes we have a **frame pointer** pointing at the other end of the frame. We usually do not need this. Move stack pointer to previous thing to return. To keep track of the return value, the address is stored at the end of the frame pointer.

The following is the example code with the O2 flag enabled.

```
mov $0xa, %eax // move the value 10 into eax
mov $0x14, %edx // move the value 20 into edx
retq // returns the object by value and throws the object away
```

%rax is 64 bits wide and is a return register. The lower 32 bits of %rax is %eax

There are two says to store numbers. The LSB representation stores the smallest byte first. For example, 0x12345678 goes like 78 56 34 12 and the other version is MSB and is 12 34 56 78. **Big Endian** is MSB and **Little Endian** is LSB. This comes from Galliver's Travels–TL;DR two nations called Big Endian and Little Endian fight over which side to crack an egg. Intel x86, ARM and many architectures use Little Endian. Internet uses Big Endian. There are multiple sized registers inside of rax(64), eax(32), ax(16), ah(8), al (8). Gheith then claims MongoDB is a horrible piece of software (you should use Meta products only (Gheith didn't say this)).

We then add the following lines and comment out happy and replace it with the function definition `extern struct Big happy();`:

```
int64_t sad() { // this function is a definition
    struct Big  big = happy();
    return big.b - big.a;
}
```

The external function **declaration** states that the function exists but hides implementation details. It is up to the linker to match functions to their declarations.

```
sub $0x8, %rsp
xor %eax, %eax // zeroes eax
callq f <sad+0xf> // save return address on stack
add $0x8, %rsp
sub %rax, %rdx
mov %rdx, %rax // moves to return
retq
```

We now begin with another example:

```
#include <stdint.h>
// this function does nothing because x is passed in by value
void add1(int64_t x) {
    x = x + 1;
}
```

"Oh, it *is* too late for a break." -Gheith, the time is 8:53 PM.

Some further readings: AT&T vs Intel Syntax, Guide to x86 Assembly, x86 vs ARM.

3 Week 3

3.1 Reference for Compiling / Cross-Compiling

```
# x86
gcc -c -O3 x.c // compile (c -> binary)
objdump -d x.o // disassemble (binary -> assembly)

# arm
// add the cross-compiler to the PATH
PATH=~/public/gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu/bin:$PATH
# cross-compile: runs on one architecture (x86 in our case)
# and produces binaries for another (AArch64 in our case)
# AArch64: the 64 bit ARM architecture
# PATH: a list of directories (folders) to look for programs in
aarch64-none-linux-gnu-gcc -c -O3 x.c    # cross-compile (C -> AArch64 binary)
aarch64-none-linux-gnu-objdump -d x.o    # cross-disassemble (binary -> assembly)
```

3.2 Lecture: January 24st, 2023

3.2.1 Functions

We begin by referencing the following function:

```
#include <stdint.h>

// only way to pass arguments by reference
void add(int64_t *x) {
    *x = *x + 1;
}

// x86
// addq (dollar)0x1, (%rdi)
// retq

// ARM produces
// ldr x1 [x0]
// add x1 x1 #0x1
// str x1, [x0]
// ret
```

x86, (src) immediate addressing is $ is to use the value as is. These values can be destinations only. (src, dest) Register addressing is using one of the registers like %rax, %rcx. Now we discuss things interested in memory: there is also indirect addressing (src, dest) where the operand is an array mem[reg[reg[#]]]. Any particular instruction can have a maximum of one memory reference. For example, you cannot write add (%r12), (%rax). ARM is an RR architecture. x86 is an Register Memory (RM) architecture. In ARM, we do immediate using # 1, registers using x0, x1, w0, w1, and indirect using [x3].

x86 has to do the exact same thing as ARM. Therefore x86 has to go through the exact same steps it just has a shorthand for it.

For example, if [x0] has the value 2000, it points to the memory address 2000 and the actual value x0 would be the memory addresses from 2000 to 2007. Then, x1 will get loaded with the value 100. Then we add x1 and we put that in x1, and therefore the answer is 101. Then, with store (str) instruction, we put x1, [x0] which sets the pointer of [x0] to x1.

We now change our program to

```
void add1(int8_t* x) {
    *x = *x + 1;
}

// cannot put dollar signs in minted, use 'd' instead

/* x86
endbr64
addq d0x1, (%rdi)
retq
*/

/* ARM
endbr64
addb d0x1, (%rdi)
retq
*/

// addb is a byte (8 bits), addw is a word (16 bits)
// addl is a long (always 32 bits), addq is a quad (64 bit)
// an example invalid is addb d1, \%rax
// we know rax is not an 8 bit operand
// all registers that begin with r are 64 bits
```

Now we augment the code to include a struct Data.

```
#include <stdint.h>

struct Data {
    int32_t x; // 0
    int32_t y; // 4
};

void nice_function(struct Data* p) {
    p->y += 1; // (*p).y = (*p).y + 1; is equivalent
}

/* this generates the following assembly:
addq d4, (%rdi) // pointers are 64 bits since 64 bit machine
addl d1, (%rdi)
retq // ret is normally assumed to be ret
*/

```

```
/* the actual assembly
endbr64
addl d0x1, 0x4(\%rdi) // different addressing mode called base + displacement
retq
*/
```

The function gets a single argument which is a pointer. **%rdi has the first argument**. The actual assembly generated is different, and uses a different addressing mode called base + displacement. This refers to mem[# + regs[reg #]] and this allows negative values.

Now we will see how ARM does it.

```
/*
ldr w1, [x0, #4]
add w1, w1, #0x1
w1, [x0, #4]
ret
*/
```

```
#include <stdint.h>

void nice_function(int32_t a[]) { // equivalent if we used int32_t *a
    a[4] += 1; // addl d0x1,0x10(%rdi) -- 0x10=16=4*sizeof(int)
    // ARM: x0, x0, #0x10 (adds 16 by pointer arithmetic)
    // retq
}
```

3.2.2 Subscript Operator

Gheith states that in C, arrays and pointers are the same. In C, you're adding to the pointer the integer times the size of the thing that you're pointing at. From StackOverflow: The definition of the subscript operator [] is that E1[E2] is identical to (*((E1)+(E2))). We can't use base + displacement since it has to be an immediate value.

```
void nice_function(int32_t *a, int64_t i) {
    a[i] += 1; // addl d0x1, (%rdi, %rsi, 4)
    // scaled index + base SIPB
    // in ARM:
    // w2, [x0, x1, lsl #2]
    // add w2, w2, #0x1
    // str w2, [x0, x1, lsl #2]
    // ret
}
```

The scaled index and base is (base, index, scale). It is mem[regs[index] + regs[index] * scale]. Scale is an immediate value and is in {1, 2, 4, 8}.

Suppose we have

```
struct Thing {
    uint64_t x;
    uint32_t a[1000];
};

void nice_function(struct Thing *p, int64_t i) {
    p->a[i] += 1;
    // addl d0x1,0x8(%rdi,%rsi,4)
    // retq

    // ARM:
    // add x0, x0, x1, lsl #2
    // ldr w1, [x0, #8]
    // w1, w1, #0x1
    // str w1, [x0, #8]
    // ret
}
```

We have now covered all the addressing modes of x86 and ARM. This has little bearing on performance.

Now, I'll write about some additional practice that was not part of lecture. Namely, this comes from this article and this video. There is this excellent guide to x86. Firstly, recall the sizes of the operation suffixes, for example, for add:

- addb is a byte (8 bits),
- addw is a word (16 bits)
- addl is a long (32 bits)
- addq is a quad (64 bit)

Condition codes stores status information aobut most recent arithmetic or logical operation and are used for conditional branching. Words are stored in little-endian byte order (x86). Therefore, the 'little-end' of the word comes first. The LSB comes first.

3.3 Lecture: January 26th, 2023

Locality of Reference: things accessed more recently are more likely to be retrieved and computed faster.

Since we know that ints, shorts, longs are popular we decide to make memory 8 bytes wide. For example, for a 64 bit machine we might have

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ & & & & \vdots & & & \end{bmatrix}$$

3.3.1 Memory Alignment

In hardware, bit-shifts are extremely fast; there is almost no cost. To get the row number, divide by 8. To get the offset, we get the byte number modulo 8. For example, if we want an int, find the row number and then shift to get the first half or last half. Misaligned memory references are more expensive. Generally puts integers where the address is divisible by 4. Example:

```
struct Thing { // must be divisible by 8
    int32_t x; // 0 (takes up first 4 bytes)
    int64_t y; // 8 (has to be aligned)
};
```

In summary:

- A **char** (one byte) will be 1-byte aligned.
- A **short** (two bytes) will be 2-byte aligned.
- An **int** (four bytes) will be 4-byte aligned.
- A **long** (four bytes) will be 4-byte aligned.
- A **float** (four bytes) will be 4-byte aligned.
- A **double** (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux (8-byte with -malign-double compile time option).
- A **long** long (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux (8-byte with -malign-double compile time option).
- A **long** double (ten bytes with C++Builder and DMC, eight bytes with Visual C++, twelve bytes with GCC) will be 8-byte aligned with C++Builder, 2-byte aligned with DMC, 8-byte aligned with Visual C++, and 4-byte aligned with GCC.
- Any **pointer** (four bytes) will be 4-byte aligned. (e.g.: char*, int*)

3.3.2 Alignment Requirements

Structs are put in areas that are divisible by eight for efficiency. Then the rest can be laid out without knowing the exact address. **Why must structs be divisible by 8?**. Alignment requirement. In order to ensure that the largest struct field is properly aligned in memory, that field has to be properly aligned inside the struct and the whole struct object itself has to be properly aligned in memory. This is typically achieved by giving the whole struct object the same alignment requirement as the largest alignment requirement among its fields. Array size guarantee. Taking the previous point into account, array elements [of struct type] typically have to reside at memory addresses divisible by the size of the largest field of the struct. This means that extra "spacing" might be necessary between array elements in order to ensure their proper alignment. At the same time the language guarantees that sizeof(T[N]) == sizeof(T) * N. This relationship immediately implies that the aforementioned "spacing" has to be counted as integral part of the array element. It cannot be seen as something introduced by the array itself, it has to be seen as internal padding inherent to the struct. (Source).

We now expand this example to have:

```
struct Thing { // must be divisible by 8
    uint8_t a; // div 1 -> 0
    int32_t x; // div 4 -> 4
    uint8_t b; // div 8 -> 8
    int64_t y; // div 1 -> 16
};
```

We want to load things from memory efficiently. What is the largest alignment required for this struct? Therefore, the tightest requirement is to be divisible by eight. Therefore, the whole struct has to be divisible by 8. The compiler makes the promise that the memory will be laid out in order. This algorithm is done greedily, find the first location that it works. Usually if you sort by size alignment is almost 'free.' Variables with more than 8 bytes will have split alignments for example 10 would be split into $8 + 2$. How does free know how much memory to free? There is a header that tells you the size of the block. Most of the time, we don't run into issues.

3.3.3 Struct Memory Alignment

```
extern uint64_t f2(uint64_t);

uint64_t f1(uint64_t v) {
    return f2(v) - 1;
}

// before the function:  %rsp % 16 == 8
// sub d0x8,%rsp // this has no purpose--maintain stack pointer alignment
// callq d <f1+0xd>
// add d0x8,%rsp // this has no purpose
// sub d0x1,%rax
// retq
```

3.3.4 The Stack Pointer

Compiler has to decide whether to store into XMM register. XMM Registers have 16 byte alignment and would like to guarantee something about the stack pointer. **Before you call a function, the stack pointer has to be divisible by 16.** Once this is done you can do safe offsets that maintain the alignment safely. It can be aligned without checking.

Call/return are used to transfer control between functions. The callq instruction takes one operand, the address of the function being called. It pushes the return address (current value of %rip, which is the next instruction after the call) onto the stack and then jumps to the address of the function being called. The retq instruction pops the return address from the stack into %rip, thus resuming at the saved return address. (Source)

When a function has more than six integral arguments, the other ones are passed on the stack. Assume that procedure P calls procedure Q with n integral arguments, such that n > 6. Then the code for P must allocate a stack frame with enough storage for arguments 7 through n, as illustrated in Figure 3.25. It copies arguments 1–6 into the appropriate registers, and it puts arguments 7 through n onto the stack, with argument 7 at the top of the stack. When passing parameters on the stack, all data sizes are rounded up to be multiples of eight. With the arguments in place, the program can then execute a call instruction to transfer control to procedure Q. Procedure Q can access its arguments via registers and possibly from the stack. If Q, in turn, calls some function that has more than six arguments, it can allocate space within its stack frame for these. (Textbook)

The most popular way to call a function is to use the **callq** function which takes a single operand which is the address of the function you want to call. For example, we can do call <entry-point>. This is the location in memory of the first instruction. For convenience, we use human readable names (machine-code converts these to numbers). retq says pc = mem[%rsp] and %rsp = %rsp + 8

The steps do are as follows:

1. RA = PC (in x86 looking at the PC is the following function).In x86, we push this return address on the stack. We do this by calling %rsp = %rsp - 8.

2. mem[%rsp] = PC

3. Save the return address: pc = <entry>

Let's discuss a stack implemented in the computer architecture. We push a value on top of the stack. Each element is 8 bytes, so we subtract 8 bytes. Pushing is subtraction. Then it stores the value at %rsp. We then rediscuss the rules:

1. Before you call rsp is divisible by 16 (the committee says so).

2. The reason for this is just to make compatibility with XMM.

A **Von-Neumann architecture** is where everything is stored in one memory. When a function in x86 starts running, it guarantees that it will be divisible by 8 and not by 16. We waste space to maintain alignment. Before you call, divisible by 16, then subtracts by 8. If you know a function doesn't use XMM registers you can have looser restrictions and not have the alignment. x86 is referred to as Complex Instruction Set Computer (CISC). ARM is referred to as Reduced Instruction Set Computer (RISC).

We use XMM to help with parallel computing and is the only way to do floating point arithmetic.

"He probably goes directly to Red-Black Trees" -Gheith.

"Stacks are such a trivial data structure. They are not worthy of discussion, but here we are" -Gheith.

4 Week 4

4.1 Lecture: February 7th, 2023

First, we will define a subset for P3.

- Upper-bound on parameters [vetoed]
- Local variable scoping [agreed]
- Print is a reserved keyword [agreed]
- Main method [agreed]
- Python-like comments, pound sign until EOL is a comment. [agreed]

Ronit banters about voting for neither global and local variables.

We re-discuss what a compiler is. It takes something from some language and converts it to some machine language (or some intermediate language).

The diagram goes: fun→machine languages→.s→gcc→machine language.

```
fun f1(p) {
    return p + 1
}

fun main() {
    x = 10
    y = f1(x)
    print(y)
}
```

Now, we want to convert this to assembly. How will that work? We want to have a function that becomes instructions. The first instruction is $x = 10$.

There are **l-values** which show in the left hand side of a statement, or **r-values** which are values generally in the right hand side of a statement. Notice how the answer itself doesn't have a name, but gets stored in something.

We can invent our own linkage convention since no one will be calling our code from the outside. There are two types of frames.

4.1.1 Caller-Callee Conventions

We draw a sandwhich with callee, return address, caller (drawn top to bottom).

The **callee** is the function that is being called. The %rsp will be the top of the callee. If the function wants to return, we will have to jump to the return address, which lives somewhere between the callee and caller frames. We need the stack pointer to be pointing towards the return address.

How do we pop the stack frame? There are generally two ways. The easy way: the compiler knows how much memory is allocated for the stack frame, therefore when it's time to return, add that quantity to the stack pointer which will go directly to the return instruction. There is a C function called alloca() which modifies the stack frame. Typically, you want to keep another pointer which is called the frame pointer %rbp. This points to the bottom of the stack frame.

Preserved registers should always be what they were before. For example, %rsp, %rbp, %rip. Another name we use for preserved registers are called **callee-saved registers**.

Some registers are not preserved, or are called **caller-saved registers**. If the caller wants to it would save them before it called the function. For example, %rax, %rdx.

Imagine that you want to call a function f. You know that it will give an answer in %rax. Suppose the compiler stored something in %rax. Right before the function, then %rax would contain x. This is the only place that contains x. After you call the function, its return value will 'clobber' %rax, and therefore overwrite the value of x. The most natural way to save the value is to push %rax, and then you can do whatever you want, and then pop %rax.

If everything is caller-saved, then the compiler has to save all the variables even though the functions might not have used it. Suppose we made everything callee-saved. Then, we might be saving things that the caller did not care about–this is why there is a split between **caller** and **callee** saved registers.

This is the standard linkage convention–you should probably follow it for functions like printf.

Now, suppose we have a function where y lives in %rcx and x lives in %rdi.

```
f2(x) {
    y = x + 10
    return 1 + f1(y)
}

// this will then copmile to
mov %rdi, %rcx
add $10,%rcx
mov %rcx, %rdi
push %rcx
call f1
// add %rcx, %rax
// in order to preserve %rcx we will do
pop %rcx
add %rcx, %rax
```

In main, we'll put a label in the assembly code. Labels by conventions are shifted all the way to the left. The label will not produce machine language itself–it's just a directive. **Stack grows down on x86 (defined by the architecture, pop increments stack pointer, push decrements.)**

```
main:
    push %rbp
    mov %rsp, %rbp
    sub $16,%rsp
    mov $10, -8(%rbp)
    mov -8(%rbp),%rdi
    call f1
    mov %rax,-16(%rbp)
    mov -16(%rbp),%rsi
    mov -16(%rbp),%rsi
    mov $0,%rax
    call printf
```

In C, strings are null terminated. Now that the value of y is in %rsi, we need to put a pointer to that string. Therefore you end up adding something along the lines of `mov format, %rdi`. This is absolute addressing mode. The address contains the memory address. Gives the contents of the memory and moves it. Then, on the way out we have to undo the stack pointer.

4.2 Lecture: February 9th, 2023

In today's lecture, we will discuss how to do conditionals, switch statements, arithmetic. The next phase will focus more on how the hardware itself works. For two weeks, we will do assignments on things that do not directly relate to what we talk about (unfortunate, states Gheith).

```
#include <stdint.h>
uint64_t f1(uint64_t a, uint64_t b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

/*
prelude
mov %rdi,-0x8(%rbp),
mov %rsi,-0x10(%rbp)

mov -0x8(%rbp),%rax
cmp -0x10(%rbp),%rax

jbe 20 <f1+0x20>
mov -0x8(%rbp),%rax
jmp 24 <f1+0x24>
```

```
mov -0x10(%rbp),%rax
*/
```

The function is compiled using `gcc -O0 -c x.c`. The prelude establishes the stack frame. The compiler decides that a will be stored 8 away from the base pointer. The next two lines are as follows: first we move a into %rax, and then we perform the cmp instruction. This compares a with b.

4.2.1 Instruction Flags

Understand that cmp src, dest is similar to temp = dest - src Instructions that do arithmetic have side effects that update things that are other than the destination. Some bits in this temporary variable have special meaning. There are the v, z, s, c bits. What do these bits mean?

- v: there is overflow if the arithmetic was performed (the number is too big for the type)
- z: the most recent operation yielded 0.
- s: sign flag. The most recent operation yielded a negative value.
- c: carry, when you add two numbers, determines if there was an extra one at the beginning–used to detect overflow.

These flags are 64 bits. For every instruction that does instructions, there will be flags set whether or not you want to use the result of the flags. It will only do this for instructions that do arithmetic. Question: does leaq set the flags? According to StackOverflow–no.

4.2.2 Jump Instructions

The next instruction we will examine is jbe. jbe takes in one argument. The 'j' in 'jbe' stands for jump. The control are call, return, and all the jumps. jmp is an unconditional jump (regardless of the operands). jbe, on the other hand, is a conditional jump. It jumps if the 'be' (below or equal) condition is true.

We can nicely structure our above code with:

```
    cmp b, a
    jbe a_is_below_b
    mov a, %rax
    jmp i_am_outa_here
a_is_below_b:
    mov b, %rax
i_am_outa_here:
    retq
```

In order to specify external functions, we say .global f1 in order for the linker to be able to find the function. In the below example, we tell the linker that there is another file that contains the symbol. It tells the assembler to assume that it exists, and the linker to resolve references to it. Global is complementary, it says that it's available to other files.

Suppose we have the files

```
// f1.s

// f2.s
.extern f1
```

```
f2:
    ...
    call f1
    ...

```

Another way to do jumps, you can do label names, and then use 1f (if the label was called 1). Jumping is expensive, and the processor screams in agony. cmovae will move only if the condition is true. ae stands for above or equal.

4.2.3 Returning Booleans

```c
int f2(int64_t a, int64_t b) {
    return a < b;
}
```

With this function, we get the following assembly output:

```
endbr64
xor %eax,%eax
cmp %rsi,%rdi
setg %al // low bits of the a register,
retq
```

4.2.4 Buffer Overflow Attacks

Hackers are able to make functions return to the wrong place. "You know how the stack frame looks like a stack frame?" Suppose we allocate an array on the stack, and suppose the stack of the array is 10. If you fool the function into modifying x[10], then you can overwrite the return address. You return to where you told it to return to. A URL would be stored in a buffer. They had an array of size 2000 sitting on the stack frame. The hacker sent an array of size 3000, and would overwrite the return address and could crash any server. Another clever hacker decided to add certain characters to the buffer and evolved into return oriented programming. The hacker could then make the web server do things that it was not intended to do by returning to the wrong place.

These attacks were so bad that the x86 added the endbr64 instruction. They would jump to places that weren't intended to be jumped to. You are only allowed to jump to endbr64 instructions–it's a security measure to mark places you can jump to into the code.

4.2.5 For Loops

```c
volatile extern int x;

void f1(int n) {
    for(int i = 0; i < n; i++) {
        x += i;
    }
}

/*
test %edi,%edi // set the condition codes to reflect what is in edi
```

```
jle 25 <f1+0x25> // jle less than or equal
xor %eax,%eax // sets i to zero
nopw 0x0(%rax,%rax,1) // no-op (a way of using space to fill up the bytes)
mov 0x0(%rip),%edx // compiler doesn't know where x is, moves it
add %eax,%edx
add d0x1,%eax // adds 1
mov %edx,0x0(%rip)
cmp %eax,%edi
jne 10 <f1+0x10> // jump back to the loop if not equal
retq
*/
```

Loads x from memory, and the 0x0 would be different if this were to be an actual program and not a single file.

5 Week 4

5.1 Lecture: February 14th, 2023

5.1.1 The Heap

We discuss the heap. There is a heap start and a heap end. We can glue two heaps together and pretend to have one contiguous heap. We have text, global variables, and the brk (pronounced break). This is a tiny bit of extra memory that the OS decides to give you. The area between end and brk is not dedicated to anything–this is where the heap usually begins its life.

MMAP lets you selectively pick chunks of memory and decide what to do. This is a topic for 439H (Operating Systems Honors).

Early on, everything is available. Let's implement the heap.

5.1.2 Malloc

How is malloc implemented? Let's look at the signature, it has a type parameter of void* and the size for the number of bytes you need. It will go into the neighborhood and mark an area of size as being used and then mark it as used. If you don't have this place, return 0 which indicates that void* doesn't exist. It is the responsibility of the program to free things that malloc'd.

We will now show the first implementation of the heap that meets the contract. You can technically return 0 always. This is technically a correct heap. We will now look at the next bad implementation:

```
// bad implementation!
void* malloc(size_t n) {
    void *p = avail
    avail += n
    return p
}
```

Every time that you ask for memory, then it would just increment and check if it was in bounds. We call this a bump allocator because we keep calling the the pointer and bumping it forward. This is technically a heap. If you want to free something in the middle it would get messed up. You would have to free things in reverse order for this to work.

Some people tried using metadata (adding a bit to determine if the memory could be freed). Now, suppose you had a LinkedList of all the freed blocks in memory. A node is made up of metadata and a data block. The data block is under the control of the caller. The metadata is under the control of the heap implementation. Then, you can keep track of how big the nodes are.

Suppose we have a negative allocate for discussion. Next one is 44, next one is -7, next one is 8, and then finally there is -10. Suppose someone says malloc 20. Then we can walk down the heap until we find something that is big enough. You lose a bit of space since you need to store metadata. If the 44 represents the size of the node, size of metadata is 1, then the actual available space is 3. Then, for the malloc(20), you would take it and split it into 20 + 1. Then, in the block of 44, you would get 44 - 21. Allocation policy: first-first policy is that the first node that fits we use. Best-fit is the smallest node that fits is the best one. This may need an auxiliary data structure. There is also worst-fit. There are risks in best-fit (i.e. the heap gets partitioned into tiny pieces and then you for something like 7, 10 you would have 3 leftover, and your heap ends up looking like swiss-cheese). Sometimes you're better off doing worst-fit.

5.2 Lecture: February 16th, 2023

If you know what a number is and how to add, multiply, and subtract them then you know the first 45 minutes of lecture.

In set theory, negative numbers are an equivalence class: $-2 = \{\{0, 2\}, \{1, 3\}, \ldots, \}$

Suppose your number contains 4 bits called d_3, d_2, d_1, d_0, any arithmetic operations on the number is modular arithmetic. The modulo is then 15 in this case since it's $2^n - 1$. The largest value we can write is given by:

$$\sum_{i=0}^{n-1} 2^i$$

If we are doing modular arithmetic, then for $y > x$ we have $x - y + N \equiv x - y \pmod{N}$. We also know that $x + (N - y) \equiv x - y \pmod{N}$. Therefore, we have "turned" subtraction into addition. The advantage here is that you can use the same circuitry to add and subtract. Let's do this with binary numbers:

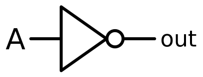
binary	unsigned	equivalence
000	0	8
001	1	9
⋮	⋮	⋮
111	7	15

5.2.1 One's and Two's Complement

This should remind you of the equivalence class we talked about. We can choose to interpret these as signed or unsigned numbers. We can choose to think of the upper bit as an indication of sign. If the front bit is 0, then it is positive, otherwise it is negative. You can see that the value of $4 = -4$. This representation is called 2's complement. The function $1 - x_i$ will flip all the bits. The most fundamental block in logic design is the inverter. The inverter does the following:

in	out
0	1
1	0

When things become popular we give them a name and a symbol. We refer to the inverter and looks like an arrow moving right with a small circle.



In mathematics we do $\neg x$, engineers do $\bar{x}$, and we (programmers) do $\sim x$.

$$-x = \bar{x} + 1 = \sum_{i=0}^{n-1} \bar{x}_i \cdot 2^i + 1$$

In computer history, we call this the 1's complement. We almost never use 1's complement. 2's complement is the 1's complement and you add one to it. If we choose the two's complement then we have a very simple rule for converting signs. For example, $2 = 010 \implies -2 = 101 + 1 = 110$

We have one more negative number. The problem is that one number is it's own negative. For example, $-4 = 011 + 1 = 100$. The issue here is that there's no way to represent whether it is positive or negative. Therefore, this implies there are two identity elements which violates the group property. ATR72 used to fall over the sky over Buffalo and Chicago in the same year simply because it was electronically controlled and the flaps would extend and then when they reach the integer limit they overflow and the computation would wrap around and the plane would fall out of the sky. Two's complement always implies that it is signed.

$$v(x) = \begin{cases} s = 0 & \sum_{i=0}^{n-2} x_i 2^i \\ s = 1 & \sum_{i=0}^{n-2} x_i 2^i - 2^{n-1} \end{cases}$$
$$v(x) = (1 - s) \sum_{i=0}^{n-2} x_i 2^i + s \left(\sum_{i=0}^{n-2} x^i 2^i - 2^{n-1} \right)$$

If the sign bit is zero, we choose to think of it as a positive number. This then becomes

$$-x_{n-1} 2^{n-1} + \sum^{n-2} x_i 2^i$$

The highest weight has a negative value. The largest value you can represent is then equal to $2^{n-1} - 1$.

$$\sum_{1}^{n-1} x_i 2^i + x_0$$

The left hand side is then guaranteed to be even. Then, we can repeat the process and continue adding the remainder into x_0. The result is then some integer value plus x_0.

5.2.2 Computer Arithmetic and Circuits

External factors that may not be part of the circuit may affect the circuit. For example, a 7 might turn into a 6 given enough noise. You could also put a Faraday shield around your computer. You can also increase the voltages in order to have more margin of error. You can never eliminate risk but you can always reduce it.

6 Week 5

6.1 Lecture: February 21, 2023

6.1.1 Shifting

We will do a **sticky-shift** to the right. The rule for sign numbers when we shift is to keep the sign bit and also shift it. We can see for the above example that -6 shifted to the right is equal to $-8 + 4 + 1 = -3$.

-8	4	2	1
1	0	1	0
1	1	0	1

We can show this with the following:

$$-a2^{n-1} + b2^{n-2} = \frac{-x2^{n-1}}{2}$$

If we set $a = x, b = x$, then the following inequality will hold.

$$-x2^{n-1} + x2^{n-2} = \frac{-x2^{n-1}}{2}$$

There are two shift instructions: shift left and shift right. The machine doesn't know if shift is signed or unsigned. Therefore, human or the compiler has to know which one to pick. **shr** generally means shift right and there is a prefix (either **ashr** or **lshr**) where a stands for arithmetic and l stands for logical, respectively. Arithmetic shift is signed and logical is unsigned.

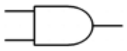
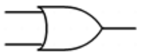
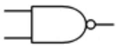
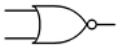
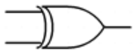
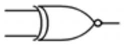
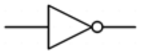
Suppose we have something of size n, and we want to extend it by some amount of bits. Let's start with one bit and figure out the recursive formula for doing it to more bits. We have an n-bit number, and we want to turn it into an $n+1$-bit number. If this is unsigned, then we can just pad it with zeros. If this is signed, we take the leftmost bit and move it all the way to the left. One of the easiest things we can do is take the value as a signed number and shift it all the way to the left and to the right (the architecture will automatically do sign extension). We keep revolving around the sign extension principle: if you want to shift it to the right do the opposite of sign extension, truncation should be all the bits you drop are equal to the new sign bit.

6.1.2 Carry Bits and Overflow

Let's say we have two four bit integers: $1111 + 0001 = 0000$ with a singular carry bit. If they were signed, then we get $-1 + 1 = 0$. Unsigned, we would get $1111 + 0001 = 0000$ would be wrong by "elementary school arithmetic" but would still be correct $\pmod 2$. Integer overflow can be detected if the carry bit is 1 after doing unsigned arithmetic. The carry bit is not necessarily overflow–that is why we also have the overflow bit. One tells us signed vs unsigned overflow. Now, we can show signed overflow with $7 + 1 = 0111 + 0001 = 1000$ and we can see that the carry bit 0. We added two positives and got a negative number. The sum of a positive number and a negative number will never overflow. If we have two numbers $a + b = x$, then this breaks down to the following casework:

$$\text{if} \quad \begin{cases} a_{n-1} \neq b_{n-1} & v = 0 \\ a_{n-1} = b_{n-1} & v = x_{n-1} \neq a_{n-1} \end{cases}$$

6.1.3 Logic Gates

Logic Gate	Symbol	Description	Boolean
AND		Output is at logic 1 when, and only when all its inputs are at logic 1,otherwise the output is at logic 0.	$X = A \cdot B$
OR		Output is at logic 1 when one or more are at logic 1.If all inputs are at logic 0,output is at logic 0.	$X = A+B$
NAND		Output is at logic 0 when,and only when all its inputs are at logic 1,otherwise the output is at logic 1	$X = \overline{A \cdot B}$
NOR		Output is at logic 0 when one or more of its inputs are at logic 1.If all the inputs are at logic 0,the output is at logic 1.	$X = \overline{A+B}$
XOR		Output is at logic 1 when one and Only one of its inputs is at logic 1. Otherwise is it logic 0.	$X = A \oplus B$
XNOR		Output is at logic 0 when one and only one of its inputs is at logic1.Otherwise it is logic 1. Similar to XOR but inverted.	$X = \overline{A \oplus B}$
NOT		Output is at logic 0 when its only input is at logic 1, and at logic 1 when its only input is at logic 0.That's why it is called and INVERTER	$X = \overline{A}$

6.1.4 The Half Adder

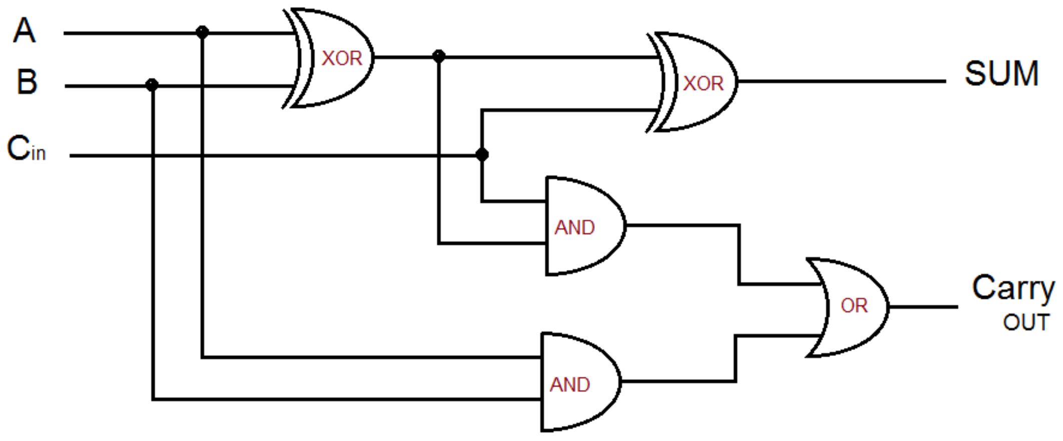
Now, we'll start talking about adders. Let's explain the below diagram. Think about each output one at a time when building circuits. Let's make a truth table of two inputs:

a	b	c	$\sum$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

So for the sum bit, we just xor the two bits. The carry bit is an and. Therefore, we have constructed a **half adder (HA)**. This is a very important and popular circuit. Now if we try to add more numbers together we can see that the half adder is not able to add them together since it can only take 2 input bits.

6.1.5 The Full Adder

Let's suppose we want to add $a + b + x$. We can compute $a + b$ using a half adder. Then, we have an additional carry. Therefore, we have $2c_1 + s_1 + x$. We can then take $s_1 + x$ using another half adder. Then, we have $2c_1 + 2c_2 + s$. You can OR both carry bits. This is easy to see: if a is carried, then we know that the other carry bit cannot possibly be on. We can also formally prove it.



If we also look at a truth table:

0	0	0
0	1	1
1	0	1
1	1	x

We can use the value x to mark it as 'don't care' which help us optimize our circuits. There is a period of uncertainty since the circuit has to go from one to another. We can pretend that these propagation are additive. This is an approximation and is quite reasonable. Therefore, if you stand at the end of the full adder circuit, looking at the output c_3 as a result of the various inputs. You'll see randomness and then eventually settles on a value.

6.2 Lecture: February 23rd, 2023

We can realize that arbitrary precision arithmetic is $\mathcal{O}(N)$ since there is no upper-bound on the size since we don't perform modular arithmetic. Now, for our adder we have an n bit quantity a and another n bit quantity b, and produces the sum s and c_{out}, the carry.

Remember we have some conditional bits such as the s bit, c bit, and the z bit. How can we compute those bits? s is easy to compute and is the most significant bit. The Z flag is set if all the bits are zero. It's maximum fan-in.

Most architectures define subtraction as the first operand plus the ones-complement of the second operand. We can think of xor as the conditional not since the xor operator will toggle bits if one of the two inputs is 'on.'

$$f(a,b) = \begin{cases} a = 0 & f(0,b) f_0(b) \\ a = 1 & f(1,b) f_1(b) \end{cases}$$

$$f(a,b) = \overline{a} f_0(b) + a f_1(b)$$

Now, let's boil this down to the function of a single variable:

$$f(b) = \begin{cases} b = 0 & f_0 \\ b = 1 & f_1 \end{cases}$$

Therefore, we can simple down $f(a,b)$ to:

$$f(a,b) = \overline{a}\overline{b}f_{00} + \overline{a}bf_{01} + a\overline{b}f_{10} + abf_{11} \tag{1}$$

The beauty of this is that the output values are predefined, which means that they are the output values in the truth table. For example:

0	0	f_{00}
0	1	f_{01}
1	0	f_{10}
1	1	f_{11}

Let's show this example for the xor function.

0	0	$f_{00} = 0$
0	1	$f_{01} = 1$
1	0	$f_{10} = 1$
1	1	$f_{11} = 0$

Therefore, xor(A,b) = $\overline{a}\overline{b}0 + \overline{a}b1 + a\overline{b}1 + ab0 = \overline{a}b + a\overline{b}$. This representation is called the **sum of products** representation of a boolean function. If given a boolean function, ignore all terms that contain zeros, and only keep the terms that have 1. Then, for those terms, look at the boolean values in the input. If you see a 0, then that represents $\overline{a}$. The product is an and. The time complexity of a circuit is typically how deep the circuit goes. With sum of products we know how deep it needs to be before hand.

7 Week 6

7.1 Lecture: February 28th, 2023

Let's have the following two functions:

```
void f1() {

}

void f2() {
    go(f1); // this is not defined by c
    channel_receive(c);
}

void f3() {
    send(c, 10);
}
```

Our intended behavior is to run f1, but not to wait for it. The idea is that we have strict concurrency not parallelism. This function will appear to return immediately. However, it will have the side effect of scheduling f1 to run. The most typical thing you do is to add it to the ready queue. The ready queue (suppose this is a LinkedList) contains all the things that are ready to run but aren't able to run right now.

Another mechanism is the channel mechanism. The channel is a communication channel. You can put values into it and you can get values out of it. The channel is a queue. The channel does not buffer, meaning that for channel receive to return a value, some other coroutine has to send to the channel. If we run the code as is, then the routine cannot make any progress. This means that you are blocking.

The send cannot proceed without receive or the channel is created. In our definition, a channel cannot hold values. It is what is called a rendezvous. They both have to exist and then they exchange and then both of them proceed. You can put a buffer but then your test case wouldn't be valid. When a coroutine is blocked they go to a different queue. Polling is inefficient because we have to keep asking.

A return is a function returning a value, a send is one coroutine sending a value to another coroutine. In our world we will never run things in parallel.

How do channels help concurrency happen? They don't help at all, they are just the primary mechanism in which you block. The primary reason is to have coroutines communicate with each other. Preemptive coroutines, run for 10 seconds, and force them to run after 10 seconds.

Everytime you have a structure that describes some abstraction, we tend to call this a **control block**. There is a thread control block, file control block, process control block. We can call this is the coroutine control block. When you try to create a coroutine, you don't want to manipulate the function. You will instead have a pointer to the function. Then the ready queue is just a ton of these blocks in a LinkedList. There is some notion of which coroutine is running and there is only one active control block running. As we create more coroutines, we add them to the queue. If the active routine is blocking, then we take ourselves, and put ourself to another place and the system goes idle. Then, we go to the front of the ready queue and make it the active coroutine.

Let's suppose we have

```
while(true) {
    i = i + 1;
    send(c, i); // this forces it to block
    // when it comes back, it needs to continue right here
    // remember the state... you want to come back to the
    // current instruction pointer and preserve all the local variables
    // this suggests every coroutine needs to have its own stack
    // we need to save the stack pointer somewhere
    // therefore each control block needs to have its own
    // area for the saved state
    // so it would become
    // next pointer (for the linkedilst)
    // saved state (what we need to save to resume execution)
    // channel
    // pointer to the function that defines the entry
}
```

Go schedules called m/n scheduling m coroutines on top of n threads. A thread is an operating system abstraction capable of operating in parallel. All threads are waiting to grab something and run it. This is how concurrency is achieved with parallelism.

7.2 Lecture: March 2nd, 2023

We are able to use promises to have non-blocking.

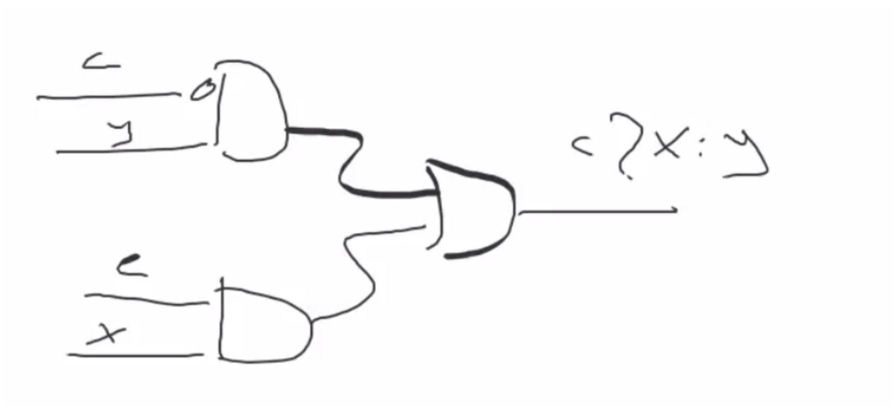
We want to construct if statements with the following truth table.

c	x	y	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

We are able to derive this using the following notation: $\overline{cx}y + \overline{c}xy + cx\overline{y} + cxy$. Now, we want to do this with a smaller number of gates. In normal algebra, we know from the distributive property that $(a+b)x = ax+bx$. In Boolean algebra, we are also able to use this as well (proof by truth table). Therefore, we can simplify the expressions above to $\overline{c}y(\overline{x} + x) + cx(\overline{y} + y)$. After further simplification, we get

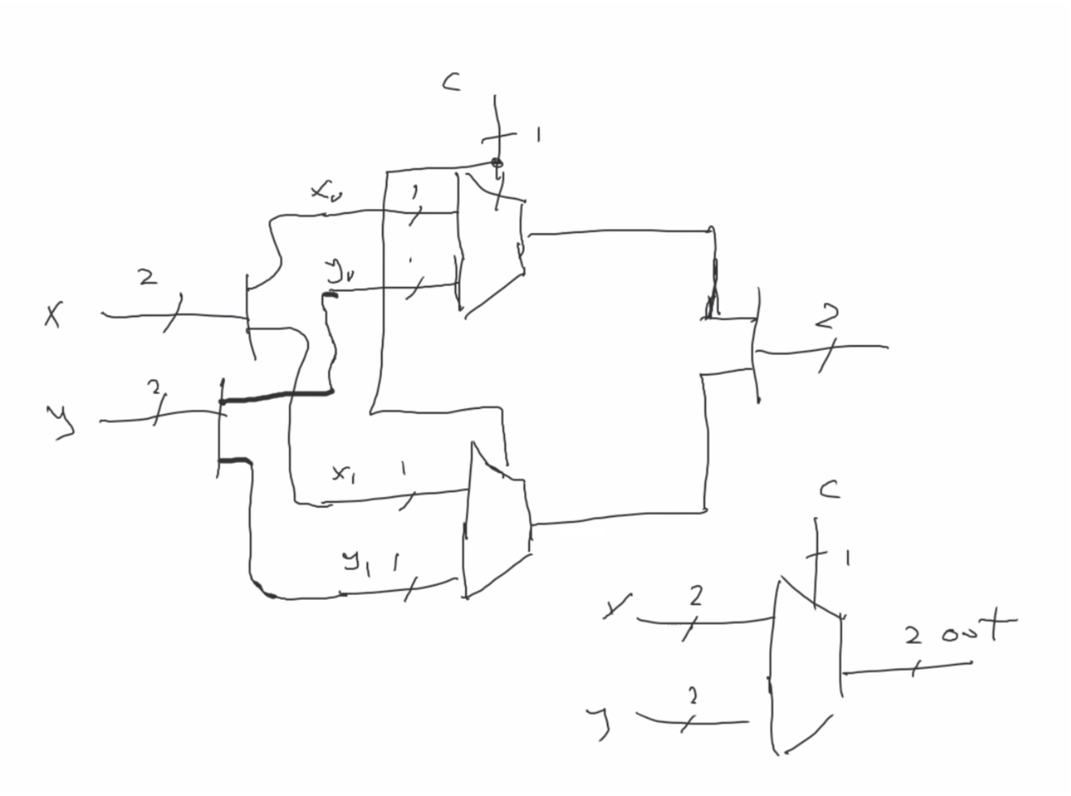
$$c?x : y \equiv \overline{c}y + cx$$

Our if statement then looks like the following:

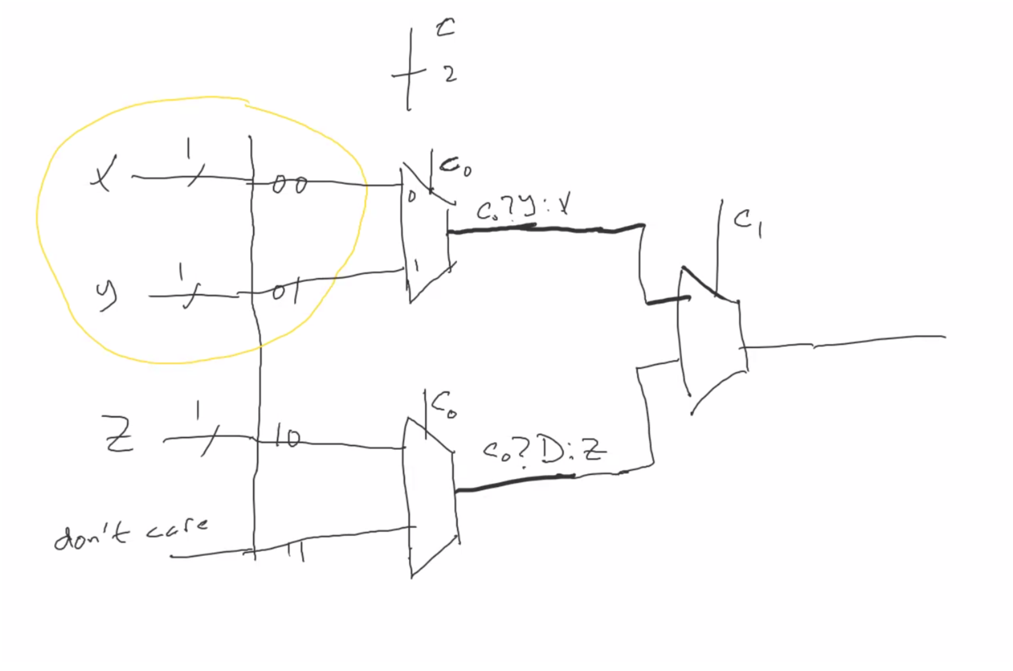


We are then able to refer to this concept as a multiplexer. We refer to this as a mux. In electronics, a multiplexer (or mux; spelled sometimes as multiplexor), also known as a data selector, is a device that selects between several analog or digital input signals and forwards the selected input to a single output line. The selection is directed by a separate set of digital inputs known as select lines. We are saying that we have a 2-way multiplexer and if the control signal is on x goes through and otherwise y goes through.

The below image is a multiplexer that produces two outputs

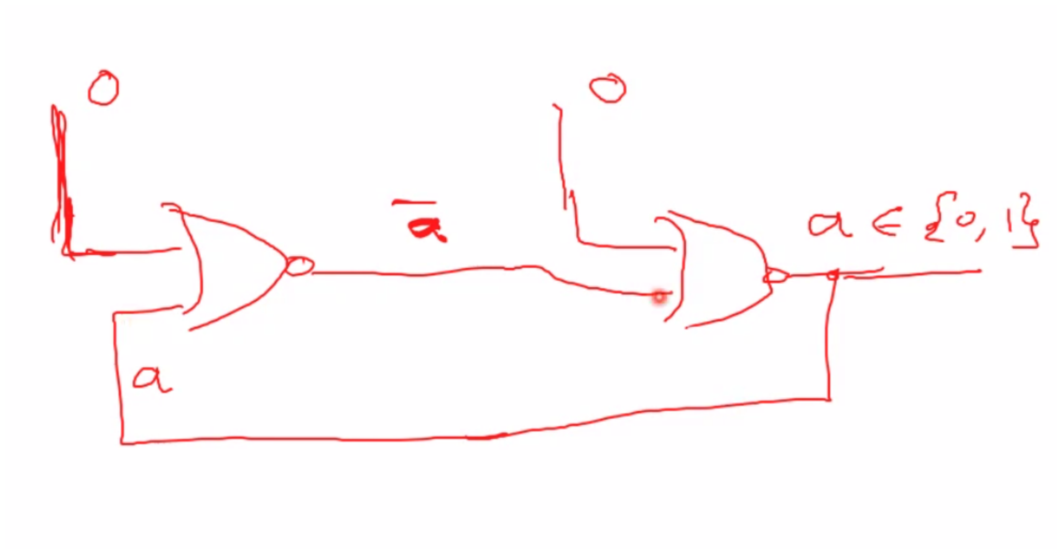


Now suppose we want to do a switch statement. Now, we want to build a multiplexer that picks on of the three values. When we want to build a three way multiplexer we have to determine how we are picking the values. So for x, y, z we can use $00, 01, 10$ respectively, and so we are wasting a bit.



Now, we have a generalized multiplexer with m control bits and up to 2^m inputs, each of which are n bits wide. We can label all those with unique binary numbers. Some of those inputs can be missing and represented as don't care. This is how we index into an array. This is why indexing into an array is somewhat expensive. The path length through the circuit is $\mathcal{O}(\log_2 s)$ which is $\mathcal{O}(M)$. The number of bits we would need is $\mathcal{O}(2^m) = \mathcal{O}(S)$. Therefore, the bigger the memory, the slower it becomes.

We then discuss how two not gates in a row can be a foundation for storing information. What if we could control the inverters by having another signal that forces their value?

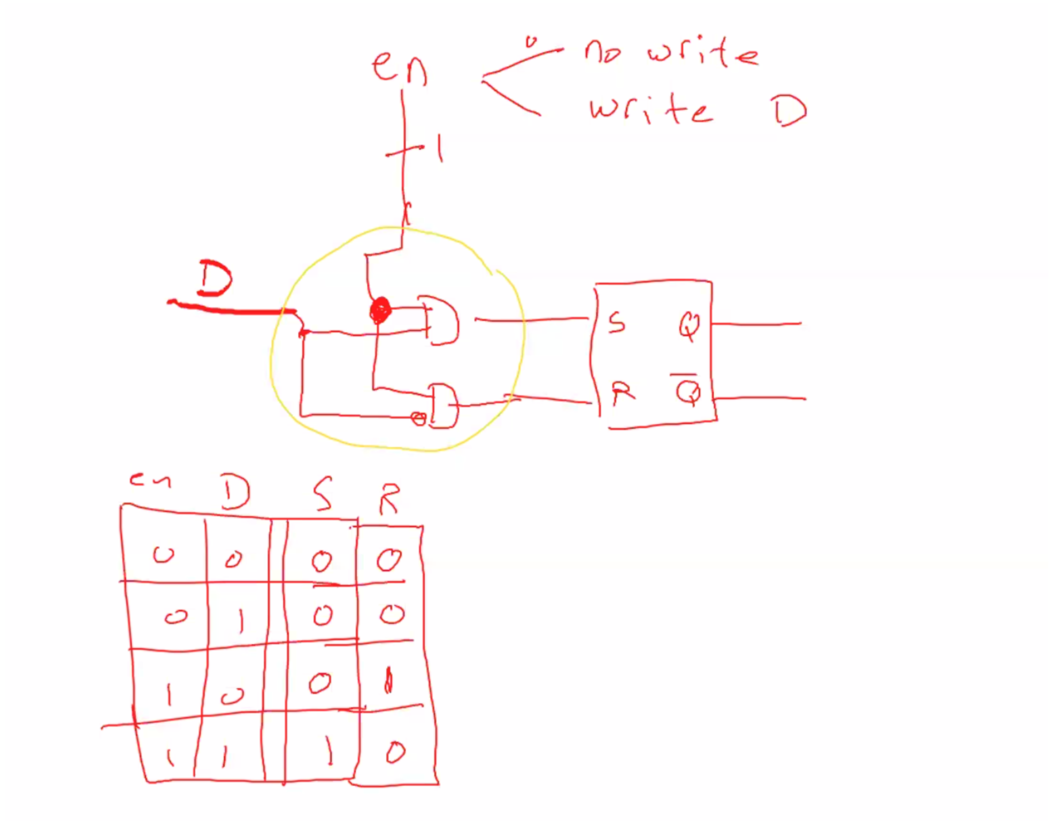


Now, we can see that the end is always 0, regardless of the initial state. We can see that the signal that we supply changes the value. We can conclude that 00 is acting liek the two inverters. We can call the right top bit the reset (R) bit. If we ever want to reset the memory cell, we can simply set the reset bit to 1, and wait a little bit and the output will turn into zero. This is why we call the other bit the set bit. We will never allow this cell to have 11 because that is forbidden. This is our first memory cell, and we can store a single bit. The truth table for the circuit is "rather interesting" because it is self-referential.

s	R	out
0	0	out
0	1	0
1	0	1

In reality, setting the forbidden state may make it oscillate. Notice that the output and the intermediate are the opposite of each other. It is always the case except for 11. We can call this a latch. This is a good building block and a good starting point but it is quite tedious to maintain. The question becomes, can we have the SR latch protect itself? The answer is yes.

We call the following circuit the D latch circuit. Once the input becomes 1, it starts capturing and stores it in the data. Otherwise, it holds on to the value. You can read more about the D-latch here. Not all instructions right in registers. We can just set the enable bit to zero and that's how we throw away values. This will come in handy for lots of our work. On Tuesday, we'll realize that D-latches are not the best option and we will discuss an improvement to the D-latch.



en	D	S	R
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

8 Week 9

So why did we skip so many weeks? Well... too many diagrams, so it's physically impossible for me to take note of all of them. But now we are back!

In Verilog, everything is ran in parallel.

Let's talk about the stage of f0.

```
// f0 stage
reg[15:0]f0_pc = 0;
reg f0_valid = 1;

// f1
reg [15:0]f1_pc =
reg f1_valid = 0;

always @(posedge clock)
    // march forward
    // f0_pc <= wb_flush ? wb_target : f0_pc + 2;
    f0_pc <= wb_flush ? wb_target :
             f1_stall ? f0_pc :
             f0_pc + 2;
end

always @(posedge clock)
```

```
    f1_pc <= f0_pc
    f0_valid = f1_valid
end

// end
reg [15:0] wb_pc = 0;
reg wb_valid = 0;
wire wb_flush = wb_valid & wb_taken_branch;
wire wb_target;

```

Gheith points out that we are making decisions locally.

Let's talk about branch prediction. You can run both branches simultaneously. Gheith's advice is to start simple and take a one cycle hit and then perform more optimizations.

If you have a flat line then we have $Cycles = L + n$ which means CPI should be roughly $\frac{L+n}{n} = \frac{L}{n} + 1 = 1 + \epsilon$

If we hit a branch, then that will cost us. Everytime you hit a taken branch you have to flush the pipeline and pay another L. Then the formula becomes expanded to

$$L + (1 - \alpha)n + \alpha Ln$$

$$CPI = \frac{L}{n}$$

If we are looping through something we will do n good and n bad. This random thing will get it right half of the time.

The idea is that instead of having a table as big as memory we can have a smaller table that is of some size. It is smaller than all of memory. How do we index it? We can use a hash function, and then we feed our PC through some hash function.

Store the actual value inside the table. The PC is modulo 16. We take the PC and then we store the actual value in the PC.

We want to make a branch predictor with a hash function that changes dynamically. We want the ones with the bits that change the quickest.

How do we deal with something that *hesitates* goes this way and then that way and then goes this way. The problem is that our predictor is too sensitive. It reacts too quickly. It then never settles on a correct answer.