6.004: Computation Structures

Lecturers: Professor Daniel Sanchez and Dr. Silvina Hanono Wachman

Notes by: Andrew Lin

Fall 2020

1 Introduction

This class is being taught virtually, so some improvements and changes to the class have been made over the summer.

Professor Sanchez and Dr. Wachman will be teaching the lectures (as returning lecturers), and Professor Emer and Professor Han will be teaching recitations. But Professor Emer is the most senior of the staff – some of the ideas in this class were pioneered by him! There are also nine Teaching Assistants, who will be helping with labs, recitation sessions, and Piazza questions. But the most important component of this class is us – even though the class is very large, hopefully we'll all still be able to interact with the instructors and TAs. As a result, the lecture will be kept as close as possible to an in-person lecture: interactivity is encouraged.

In short, 6.004 is an introduction to the world of **digital systems**. Digital systems are actually quite new – the first computers, like the ENIAC in 1943, was installed in Philadelphia, weighed 30 tons, consumed 200 kilowatts (the lights dimmed when it was turned on), and it could only do about 1000 operations per second. Things have advanced in the last few decades dramatically: most laptops today weigh 1 kilogram, consume 10 watts, and can do about 10 billion operations per second.

Fact 1

This kind of improvement is much faster than the rate for technology like cars, and computing devices are everywhere! Large and small systems are embedded everywhere in the world today, but even 15 years ago, many of these systems barely existed.

Our goal is to look at **how to physically implement computation**, with a bottom-up approach: we'll start with devices, materials, and atoms, building them up into combinational and sequential circuits, and then we'll use those to build a processor that can execute arbitrarily language in assembly. We'll think about how to make these fast, and then we'll move into computer systems (hardware and low-level software like virtual memory and operating systems) so that these systems we're constructing can be broadly useful.

These systems are very complex – each one has billions of devices, and they evolve very quickly. And the reason for this rapid evolution is that between each of these layers, there are **clean interfaces** of abstraction, like virtual machines and digital circuits. So this gives us a way to reason about the implementation without knowing the details:

Example 2

The ENIAC and the laptop we're using today are both built using bits and logic gates, but these logic gates use completely different implementations.

This means that we'll have a **hierarchical design** built into this class: we'll make each level of the design simple to understand, and the class will be built on abstract "building blocks."

The first module of this class will be centered around **assembly language**: we'll understand how to go from high-level programming languages, like those in 6.0001, to the basic language of computers. But then we'll go ahead and cover the topics of digital design, computer architecture, and computer systems.

The class will focus on **programmable general-purpose processors**, because microprocessors are the basic building block of computer systems – understanding them is important even if we're not planning on being hardware designers. Plus, they're the most sophisticated systems we have today – they have all of the tricks that make computations run efficiently, so they are a good vehicle for understanding design principles in hardware. Even if we think that "we will never design hardware in our own lives," it's important to note that more and more companies are designing specialized processors and accelerators today, so it's a great area to go into. And we'll see that designing hardware is actually **similar in principle** to designing software. By the end of the class, we'll be able to **design a simple processor from scratch**.

We'll do this by relying on modern design tools: **RISC-V** is a modern, open-source instruction set that's been gaining a lot of traction. And on the hardware side, hardware is going to be designed using **Minispec**, which is a hardware description language that was built specifically for 6.004. (It's basically a simplified version of the open-source **Bluespec**.)

This class will have two synchronous lectures per week – all slides, videos, and related materials will be posted on the website, https://6004.mit.edu. We'll also meet twice per week in recitations, which are designed to help us work through tutorial problems (which are similar to the quiz problems). Our assigned recitation sections are listed on the website, and we can let the instructors know via Piazza if we need to change sections.

Fact 3

Recitations will follow a **flipped model**, so students will work on tutorial problems together in small groups. In order for this to be successful, **we should watch the previous lecture and attempt some post-lecture problems before each corresponding recitation**.

Attendance at recitations is worth 5 percent of our grade, but we have four excused absences. Alternatively, we can opt out of recitation participation by October 2, in which case that percentage of the grade goes to quizzes instead. But the instructors encourage us to attend recitations, because they're an effective way for most students to learn the material.

In terms of assigned work, there will be seven mandatory lab exercises (with online submissions and virtual check-offs to go over the labs) throughout the semester. To accommodate illnesses, unforeseen circumstances, and so on, there will be seven free late days. The class ends with an open-ended design project, and there are also three quizzes throughout the term (on October 1, November 10, and December 3, from 7:30-9:30).

Throughout the class, we're encouraged to ask questions by unmuting ourselves (if the question is specific to lecture) or by Zoom chat (if the questions are less direct but still related). And we should keep our video on if possible, because that helps us stay engaged and gives the professor nonverbal feedback. (Recordings won't include our video, and those recordings aren't publicly available anyway.)

About 50 percent of our total grade come from labs and the design project (80 and 20 points respectively), and the rest come from quizzes and participation (90 and 10 points). The fixed grade cutoffs will be at $165, 145, 125$ points for an A, B, and C, respectively. An F grade is assigned if we receive less than 125 points or don't complete all of the labs.

The fastest way to get answers to questions will be the Piazza forum. Other than that, regular virtual office hours will be held to help us with any questions that we have – the schedule and lab queue are listed on the website.

2 September 1, 2020

The rest of this first lecture will dive into the first technical topic of this course, which is **binary number encoding and arithmetic**. The focus here is how computers store information and perform operations on numbers.

Fact 4

Digital systems are called "digital," because they process information using discrete symbols called "digits." (This is in contrast to "analog" systems, which can have a continuous range of values.)

We often manipulate using the **binary digits** 0 or 1, though the implementation is often done by some analog system (for example, encoding 1 if some continuous voltage is above some value). In principle, we could use digits of any type, but the most common **encoding of numbers** is using binary or base 2, because certain operations like addition, comparisons, and logical operations (AND, OR) are very efficient when we only have two possible values per digit.

Definition 5

A positive integer is encoded by assigning a weight (a power of 2) to each bit. Specifically, the value of a string of bits $b_{N-1} \cdots b_0$ of length N is

$$v = \sum_{i=0}^{N-1} 2^i b_i.$$

For example, the binary string 01111010000 encodes the number

$$v = 0 \cdot 2^{11} + 1 \cdot 2^{10} + \cdots = 1024 + 512 + 256 + 128 + 64 + 16 = 2000.$$

Notice that the smallest number we can represent in an N-bit string is 0, and the largest number is $2^N - 1$. These correspond to the all-0s and all-1s strings, respectively.

Writing long strings of 0s and 1s often gets tedious, though, so these numbers are often **transcribed** into a **higher-radix notation**. One popular choice is **hexadecimal**, where we use base $2^4 = 16$ instead of base 2. To translate between hexadecimal and binary, we encode each group of 4 (binary) bits in a single hexadecimal digit: the digits are 0 through 9, followed by A, B, C, D, E, F. (For example, 1011 corresponds to B, and 0011 correspond to 3.)

Example 6

The binary string 011111010000 becomes the hexadecimal number $7D0$. When we're mixing different bases, we often prepend the **base identifier**:

```
0b0111 1101 0000 = 0x7D0.
```

Here, the x stands for "hex."

Addition and subtraction in base 2 look similar to that in base 10: for example,

$$1110_2 + 111_2 = 10101_2,$$

keeping the "carries" into the next digit in mind. (This is the equation $14+7=21$ in base 10.) Similarly, the equation $14-7=7$ becomes

$$1110_2 - 111_2 = 111_2,$$

where this time we need to remember to "borrow" accordingly from larger digits. But something more interesting happens if we try to do a subtraction like $3-5=-2$ in base 2: the issue is that **we currently have no way to encode negative integers**.

This issue is connected to another fundamental concept, which is that of **binary modular arithmetic**. If we force all of our strings to be of a fixed number of bits, addition or other operations might take our numbers outside of the allowed range of numbers! This is called an **overflow**, and often computers just ignore extra bits that go outside of the allowed range. In other words, we do all operations **modulo** 2^N, meaning that numbers "wrap back around" every 2^N in a clock-type structure. For example, if we constrain our subtraction $3-5=-2$ to just three binary digits, we find that

$$011_2 - 101_2 \equiv 110_2 \bmod 2^3.$$

But we want to address the negative integer encoding more directly. One convention is to **add an extra bit** (the leftmost one), and let that one denote the sign. For example, we could say that

$$111111010000_2 = -2000$$

(only the first digit has been changed to a 1 from the first example). But this is not actually a great idea: there are two different encodings for 0 (either 0 or -0). And then addition and subtraction will need to be more complex, which is why this representation is never used.

Instead, we're going to go back to the modular arithmetic idea: we want addition and subtraction to be operations around our "wheel modulo 2^N," so we'll relabel some of the digits. For example, instead of letting the strings $(000, 001, 010, 011, 100, 101, 110, 111)$ represent the numbers 0 through 7, we can let them represent the numbers $0, 1, 2, 3, -4, -3, -2, -1$ (in general, making **exactly half** of the numbers negative, so that the other half stay non-negative). This is called the **two's complement encoding**, and what it basically means is that the **most significant bit contributes a negative weight**:

$$v = -2^{N-1} b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i.$$

This way, all of the negative numbers will still have a 1 in the highest-order bit, but this is still different from the sign-magnitude encoding. For example, the most negative number is encoded by the string $10\cdots0$, which is -2^{N-1}, and the most positive number is encoded by the string $01\cdots1$, which is $2^{N-1}-1$. (Meanwhile, the string of all bits now represents the number -1 – this is a good exercise to check, and it will come up frequently.)

Two's complement arithmetic is now much easier: for example, **negating a number** A can be done by inverting all bits (from 0 to 1 and vice versa), and then adding 1. We can check this with some arithmetic:

$$-A + A = 0 = -1 + 1,$$

which means that

$$-A = (-1 - A) + 1.$$

But now the number $-1-A$ is the binary number $1\cdots1$, minus the binary number $A_{n-1}\cdots A_0$. And now this subtraction has no "borrowing:" subtracting a digit A_i from 1 always flips the value, resulting in the **inverted** value $\overline{A_i}$. So $-1-A$ is indeed the inverted number of A, and then we add 1 to get $-A$ as above. And because $A-B$ is the addition

$A + (-B)$, we can now **use the same hardware circuit for addition and subtraction**. For example,

$$3 - 6 = 0011_2 - 0110_2 = 0011_2 + 1010_2$$

(where the last step is turning $3 - 6$ into $3 + (-6)$), and then this addition gives us

$$= 1101_2 = -3.$$

In summary, our digital systems encode information using binary digits because this is the most efficient and reliable way to represent numbers. Using a two's complement encoding helps us keep arithmetic simple, and it allows us to represent both positive and negative numbers. And there are some post-lecture questions we should answer on the website before 10am tomorrow!

3 September 3, 2020

Today, we'll be introducing **assembly language and RISC-V**. (Before we begin, some logistics: the first lab for this class is being released today in a few hours, and lab hours will start tomorrow. Instructions on how to set up a GitHub repository are also listed on the website.)

> **Fact 7**
> There will be an open Zoom room for lab hours, so we can feel like we are collaborating in a shared space.

The point of today's lecture is to help us begin to understand how microprocessors actually work. As discussed last time, it would be great if our hardware could execute programs in Python or Java or other high-level languages, and our goal is to build such a **general-purpose** processor. But executing the features in such languages automatically is difficult, so we'll have our first level of abstraction today: **machine (or assembly) language**.

We can think of assembly language as an interface between hardware and software: we take our high-level languages, and we do some **software translation** (also known as **compiling**) to turn this into machine language that the microprocessor can understand. Those instructions can then be directly executed by our hardware.

Every microprocessor has three main components:

- A **register file** is a grouping of **registers**, which are typically fixed size binary strings (about 32 bits). These register files are generally fast to access, and there are usually only a small number of registers (about 32).
- An **ALU**, or **arithmetic logic unit**, is where the main computations take place. This ALU works directly on the register file: it takes in sources from the register file, and outputs values to certain destinations in that register file.
- **Main memory** can be thought of as a very large (almost endless) storage which can handle all of our data storage needs. This holds both the contents of our program and the data needed during execution.

The reason for having both a register file and main memory is that we should store only a small amount of immediately needed data in our register file, so that it's easy to access and work with during computations. So we will also need a way to transfer information between those two components, and we need to design our machine language in a way that works well with these components.

Definition 8

A **assembly language program** is a sequence of very basic operations which directly map to operations that our processor can perform.

There are basically three kinds of operations that such programs are allowed to perform:

- ALU operations, which are certain primitive arithmetic operations,
- Loads and stores, which are used to transfer information between the register file and memory, and
- Control transfer, which tells us to jump between locations (lines) in our program so we don't have to execute everything sequentially.

Example 9

Let's start with a program and understand what our assembly language needs to express: consider the simple program `sum = a[0] + a[1] + ··· + a[n-1]`.

Suppose that our array is larger than the register space we have, so it needs to live in our main memory. Then our array elements `a[0]`, `a[1]`, and so on, live at certain **addresses** in memory, and we also need to store other variables for the sake of computation: in this case, we'll label three variables as `base`, `n`, and `sum`, which tell us the starting address (in memory) of our array, the number of elements in the array, and the result of our computation.

Then our **register file** will need to store some selective information for our program to run: one way for this to work is if the first register stores the address of our current array element, the second one stores n, and the third stores our sum. Then our program might take the following steps:

`x1 ← load(base)`, which loads in the first address of our array,
`x2 ← load(n)`, which loads in the total number of elements to be adding up,
`x3 ← 0`, which initializes our sum to 0.

Now `loop` the following operations:
`x4 ← load(Mem[x1])`, which is a shorthand way of saying to get the element `a[0]` and put it into `x4`,
`add x3, x3, x4` (add to our rolling sum),
`addi x1, x1, 4` (increment the address – we'll explain why we increment 4 later),
`addi x2, x2, -1` (decrement the number of remaining addends),
`bnez x2, loop` (branch if `x2` is not equal to zero, meaning that we return to `loop`).

`store(sum) ← x3` (put the value back into our main memory to free up our register).

Here, "loads" and "stores" are the steps where we are accessing memory: **load** takes us from main memory into our register, and **store** takes us from our register into main memory.

We can compare the differences between a high level language and an assembly language:

- In a high level language, we can do complex arithmetic and use complex logical operations, while in an assembly language, we're limited to primitive versions of these.
- High level languages have arrays and dictionaries and similar data types and structures, but assembly only knows about bits and integers and similarly primitive data structures.

- We can use conditional statements and loops and so on in high level languages, but we can only use a few basic **control transfer instructions** in assembly.
- But high level languages cannot be directly implemented in hardware, while assembly language is designed to be implementable. Of course, it would be very tedious to program in assembly all the time, so we need a mechanism of doing this software translation.

Definition 10
An **instruction set architecture** (or **ISA**) specifies exactly what kind of storage we have available, the full list of processor operations allowed, and what format information needs to be sent in.

In 6.004, we'll use the RISC-V ISA, which is an open-source new architecture from Berkeley which is gaining a lot of traction. There are many variants – different data widths, certain extensions which support multiplication and division or floating point arithmetic – but we'll **stick to the basic RV32I** 32-bit integer processor in this class.

Fact 11
RISC stands for "reduced instruction set computer" – the concept is to support the minimum set of operations possible that can still accomplish the necessary goals. Ideally, this means we can build hardware that execute operations very quickly (we can avoid special cases and complicated instructions).

We'll now jump in to the full set of instructions for a RISC-V processor. In our specific case, everything is of size 32: the width of each register is 32 bits, and we'll call these 32-bit strings **words**. We also have 32 registers in our register file, labeled `x0` through `x31`. Out of these, `x0` is a special register which is hardwired to 0 – having a known value turns out to be very useful.

On the other hand, our main memory also stores in chunks of 32-bit words, but the number of locations is determined by the number of address bits that we have, and our **addresses will be 32 bits wide**. (In other words, the "location" of our memory is tracked by a binary string of length 32.)

Remark 12. *For historical reasons, addresses are written and tracked in* **bytes** *(groups of 8 bits), meaning that 4 bytes form each word. So going from one 32-bit word to the next must increment us by* 4 *bytes, and that's why our addresses will often be* $0, 4, 8, \cdots$. *Notice that because our addresses are stored as* 32*-bit strings, we can have* $\frac{1}{4} \cdot 2^{32} = 2^{30}$ *words of main memory in our processor.*

As mentioned above, the three types of operations are **computational** (or ALU), **loads and stores** (moving data), and **control flow** (telling the processor not to always go in order). We'll spend the rest of this lecure going through some specific operations.

Computational instructions come in four types, **arithmetic** (such as `add` and `sub`), **comparison** (`slt`, meaning "set less than," and `sltu`, meaning "set less than unsigned,"), **logical** (`and`, `or`, and `xor`), and **shift** (`sll` (shift left logical), `srl` (shift right logical), and `sra` (shift right arithmetic)). We'll explain all of the computational instructions later on, but it's good to mention that "and" returns 1 if and only if both input bits are 1, "or" returns 0 if and only if both input bits are 1, and "xor" returns 1 if and only if the two input bits are the same. The important common thread between all of these is that the ALU **takes in two registers** as **source operands** and **outputs to one** as a **destination register**.

Fact 13

If we want to perform an operation `oper` on `src1` and `src2`, and we want to store that value in `dest`, then the syntax in our language is `oper dest, src1, src2`.

For example, `add x3, x1, x2` tells us to put the contents of `x1` and `x2` together, and store the result in x3. Meanwhile, `slt x3, x1, x2` tells us to compare `x1` and `x2`: if `x1` $<$ `x2`, then we set `x3` to 1, and otherwise we set `x3` to 0. And `sll x3, x1, x2` say to shift `x1`'s contents left by the number of bits specified in `x2`, and then put the value in `x3`.

Example 14

Suppose that our register currently has values `x1` $=$ 00101 and `x2` $=$ 00011. (Remember that the words of our register are supposed to be 32 bits wide, but we'll pretend they're 5 bits for now.)

If we want to run the operation `add x3, x1, x2`, then the binary integer 01000 will be stored in `x3` (we can check the binary addition for ourselves). A more complicated example: if we want to run `sll x3, x1, x2`, we need to shift 00101 to the left by 00011 bits, and shift in zeros in the right. So 00101 becomes 01010, which then becomes 10100, which then becomes 01000 (note that we lose the leading 1 in this process). Then this string gets stored in our register location `x3`.

Remark 15. *We always read before modifying, so something like `add x1, x1, x2` doesn't cause any problems for us.*

It turns out it's common to have one operand be from our register, but the other to come from a small constant encoded in the instruction itself (for example, incrementing a counter or adding 4 to the address). So even though we try to minimize the total number of operations, we still implement certain **register-immediate instructions**, which follow the format `oper dest, src1, const` (with `const` a constant replacing `src2`). For example, `addi x3, x1, 3` uses the "add immediate" operation we store the value (`x1` $+ 3$) in the register location `x3`. And now we have the **full list of computational instructions** for the RISC-V processor: aside from the ones listed above, we also have `addi` (we don't need subi beacuse we can add a negative value), `slti`, `sltiu`, `andi`, `ori`, `xori`, `slli`, `srli`, and `srai`.

These instruction formats that we've described need to be written in a specific way so the processor can parse them, so there is a specific **register-register instruction format**. This instruction format comes in a 32-bit binary string, broken down as follows:

- Bits 0-6: Type of instruction format described. For example, 0110011 corresponds to this "reg-reg" format.
- Bits 7-11: Location of `dest` (the destination).
- Bits 15-19: Location of `src1` (the first source).
- Bits 20-24: Location of `src2`.
- Bits 12-14 and 25-31: Encoding of the function itself.

On the other hand, a register-immediate instruction format (instruction format type 0010011 for "reg-imm") uses bits 7-11 to encode our destination, bits 12-14 to encode the function, bits 15-19 for the source 1 register, and bits 20-31 for our "const" or immediate value. **Notice that this means our constant can only be up to 12 bits large** (in two's complement binary, this means the number must fall in the range $[-2^{11}, 2^{11} - 1]$).

Example 16

Suppose we wanted to execute the complex set of operations $a = ((b + 3) \gg c) - 1$.

Even though all operations here are indeed logical operations that we've described already, we have to break this up into some basic computations, because each instruction can only specify two source operands and one destination. This is known as **three-address instruction**, and the breakdown looks something like

$$t0 = b + 3, \quad t1 = t0 \gg c, \quad a = t1 - 1.$$

If we wish to convert this to assembly, let's assume that a, b, c are stored in registers `x1`, `x2`, and `x3`, and we will store our temporary variables $t0$ and $t1$ in `x4` and `x5`. Then the code becomes

```
addi x4, x2, 3
srl x5, x4, x3
addi x1, x5, -1.
```

We'll now discuss some instructions that are slightly different from the basic logical operations described above. We'll briefly touch on `lui` first. Remember that our 32-bit instructions can't set all 32 bits of a register at once – in fact, even the register-immediate instructions can only deal with constants that are up to 12 bits long. So `lui` (**load upper immediate**) takes in an immediate value, appends 12 zeros to the end of it, and puts that value in the register. For example, something like `lui x2, 0x3` would store the value 0x3000 (remember that in hexadecimal, each digit is four binary digits) in the spot `x2`. Basically, we often use `lui` to set the 20 upper bits of our register, while using the immediate instructions to set the other 12.

And the final type of instructions we'll discuss today are **control flow**, which is best illustrated with an example: if we want to run branching code like "if($a < b$), then set $c = a + 1$. Else, set $c = b + 2$," we can use **conditional branch instructions**.

The format here looks like `comp src1, src2, label`, and this basically says to compare the values of `src1` and `src2`. If the comparison returns "true," then we take the branch located at label `label`, and otherwise we continue executing our program.

Looking at what methods of comparison are allowed, RISC-V supports `beq` (branch equal), `bne` (branch not equal), and `blt`, `bge`, `bltu`, and `bgeu` (corresponding to $<, \geq, <$ unsigned, and $\geq$ unsigned). This concept of "signed versus unsigned" is that we typically consider binary strings to be two's complement numbers, but we can also treat them as positive binary numbers if we'd like. In summary, translating this code into assembly looks like the code below if we let `x1`, `x2`, `x3` be the spots that a, b, c are stored:

`bge x1, x2, else` (this returns true if $a < b$ is false),
`addi x3, x1, 1` (in the case where $a < b$ is true, increment by 1)
`beq x0, x0, end` (end the program for sure)
`else:  addi x3, x2, 2` (in the case where $a < b$ is false, increment by 2).

Here, we're taking a branch when we have $a \geq b$, in which case we go straight to our **else** statement. The third line always returns true, so we'll always take that branch to skip the "else" line – we should be careful when working with assembly language to watch out for this kind of thing!

We'll continue next time with unconditional branches, loads, and stores.

4 September 8, 2020

Our first lab will be due next Thursday, and checkoffs for it start this Friday (we can find the signup form on Piazza). As a reminder, there is a Zoom room that we can join to emulate the "6.004 lab room" while we're waiting on the queue to get help.

The main topic of this lecture is **compiling code and implementing procedures** in RISC-V assembly, but we'll start by explaining a few more commands that weren't covered last class.

Recall that RISC-V has three kinds of allowed instructions:

1. Computational instructions executed by the ALU (which are either **register-register**, taking the form `oper dest, src1, scr2`, or **register-immediate**, taking the form `oper dest, src1, const`),
2. Control flow instructions (which are either **conditional branch instructions** of the form `branch-comparison src1, src2, label`, telling us whether or not to jump to the location with a certain label, or **unconditional branch instructions**, which we'll explore today), and
3. Loads and stores of the form `lw dest, offset(base)` or `sw dest, offset(base)`, which help us read and write information between main memory and our register.

We'll also clarify the purpose and usage of **pseudoinstructions**, which are shorthand notation for other instructions or sets of instructions.

Fact 17

Recall that one way we can achieve an "unconditional branch" to skip over certain lines of code (for instance, to skip past a few lines of code in control flow) is to run a command like `beq x0, x0, end`.

Since `x0 == x0` is always true, this will always jump to the location labeled with `end`. But RISC-V also has explicit unconditional branching, and we'll see soon why this is necessary.

Definition 18

The operation `jal` serves as an **unconditional jump and link**.

For example, the code `jal x3, label` will jump to the label (encoded as an offset from our current instruction), and then the **link** (which we'll discuss later in the lecture) will be stored in `x3`. Note that in the actual 32-bit string that encodes this instruction, there are **20 bits** where we can encode an immediate.

We can also use the operation `jalr`, which is an **unconditional jump via register and link**. This can be written for instance as `jalr x3, 4(x1)`: this jumps to the address which is four after the memory address stored in `x1`. With these two commands together, we'll be able to jump to any 32-bit address we want.

So now we can talk about how to access memory values and do computations on that memory. There isn't an instruction that can take two addresses in memory, add the values of those two words together, and put the sum in a different, so something like `a = b + c` doesn't work as a single instruction in RISC-V. Instead, we need to run the following pseudocode on the left:

```
x1 ← load(Mem[0x4])
x12← load(Mem[0x8])
x3 ← x1 + x2
store(Mem[0x10]) ← x3.
```

```
lw x1, 0x4(x0)
lw x2, 0x8(x0)
add x3, x1, x2
sw x3, 0x10(x0).
```

But memory addresses are specified in RISC-V as a pair (base address, offset). The base address needs to be stored in a register, and the offset is some 12 bit constant that we include in the instruction. In assembly, this ends up looking like the code on the right: because we're taking all addresses as offsets from 0, we're using $x0$, hard-coded to 0, as the base address.

Example 19

Now we're ready to take another look at how to implement our program for summing array elements. Recall the setup: our array is stored in consecutive addresses of our main memory, and we also have three variables `base` (the location of our array's start), `n` (the number of array elements), and `sum` (the place where the sum should end up going in main memory)

For reference, the half-pseudocode from last week is shown on the left. We assume that `x10` already contains the address of `base`, which is (in turn) the starting address of our array in main memory.

```
x1 ← load(base)
x2 ← load(n)
x3 ← 0

loop:
x4 ← load(Mem[x1])
add x3, x3, x4
addi x1, x1, 4
addi x2, x2, -1
bnez x2, loop

store(sum) ← x3.
```

```
lw x1, 0x0(x10)
lw x2, 0x4(x10)
add x3, x0, x0

loop:
lw x4, 0x0(x1)
add x3, x3, x4
addi x1, x1, 4
addi x2, x2, -1
bnez x2, loop

sw x3, 0x8(x10).
```

All that's really left to do is to convert the beginning and end of the program to "loads" and "stores," and we've done that on the right. The most important thing to note is that loads and stores always look at addresses relative to some register value (in this case, either `x1` or `x10`).

The final set of "instructions" we'll consider are **pseudoinstructions**, which aren't actual RISC-V instructions but serve as aliases to them for easier coding.

Example 20

The command `mv x2, x1` (here `mv` stands for "move") is equivalent to `addi x2, x1, 0` – both of these mean that we copy the value of `x1` to `x2`. Meanwhile, `ble x1, x2, label` is equivalent to `bge x2, x1, label`.

Example 21

For a more complicated example, `li x2,3` (standing for "load immediate") will translate directly into `addi x2, x0, 3` since the constant is small enough, but we can also load larger numbers like `li x3, 0x4321`: this will be split into the two instructions `lui x3, 0x4` and `addi x3, x3, 0x321`, because `addi` is only able to encode up to 12-bit constants.

Remark 22. *There's another component of the machine here, called an* **assembler**, *which translates these instructions into binary and does the hard work of putting everything into 32-bit binary strings. (For example, it can tell which of the two use cases of* `li` *to assemble.) But we won't look at that too much here.*

Example 23

Suppose our machine is in a state where the values `0x35, 0x3, 0x9, 0x1, 0x22, 0x23, 0x21, 0x16, 0x18` are stored in the first addresses `0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, 0x1C, and 0x20`, respectively. Also, suppose our register file has initialized the values of `0x8` and `0x14` in `x2` and `x3`.

We'll try performing a series of instructions:

- The instruction `add x1, x2, x3` will put the value `0x1C` in `x1`, and `mv x4, x3` will put `0x14` in `x4`.
- If we run the instruction `lw x5, 0(x3)`, we'll fetch the value at main memory with address `0x14` (which is the value in x3), and therefore register `x5` will have value `0x23`.
- Similarly, the instruction `lw x6, 8(x3)` will skip two words past 0x14, so we'll write `0x16` into `x6`.
- And the operation `sw x6, 0xC(x3)` will put the value 0x16 (which is currently in `x6`) into the address `0xC + 0x14 = 0x20`.

For the rest of this class, we'll talk about **compiling simple expressions into RISC-V code** and some relevant complications that come up. In other words, if we have some C or Python code, how can we turn it into assembly? Here are some concepts to keep in mind:

- Assign variables to registers – that is, decide which variables and temporary variables correspond to which registers.
- Translate operators into computational instructions (like `add` or `or`),
- Use register-immediate instructions (like `addi`) when we're dealing with operations with small constants and `li` for larger constants.

For example, if we want to run the two commands `y = (x+3) | (y + 123456)` and `z = (x * 4) ^ y`, we'll need to break down the complex instructions into simpler ones. First, we'll assign x, y, z to the registers `x10`, `x11`, and `x12`, and we'll use `x13` and `x14` for temporary variables. Then the code looks as follows:

`addi x13, x10, 3` (to create $(x+3)$)
`li x14, 123456`
`add x14, x11, x14` (to create $y + 123456$)
`or x11, x13, x14` (to finish evaluating y)
`slli x13, x10, 2` (to multiply x by 4)

`xor x12, x13, x11` (complete the final operation).

Conditional statements are a bit more difficult, but we can do those systematically as well. For example, an **if statement** of the form "if (expression), then execute (if-body)" can be written schematically as

compile `(expression)` **into a register** `xN`
`beqz xN, endif`
compile `(if-body)` **here**
`endif:`

Example 24
Suppose we have two integers x, y, and we want to execute the instruction "$y = y - x$" if $x < y$.

We can put the variable x in `x10` and y in `x11`. Then following the schematic above, our RISC-V code is

```
slt x12, x10, x11
beqz x12, endif
sub x11, x11, x10
endif:
```

Of course, we can actually combine the "expression" and the "branch" at times: in this case, the first two operations can be combined into the single command `bge x10, x11, endif`, because we're doing an ordinary comparison.

If-else statements work similarly, but are a little bit more complicated. The schematic for "if (expression), then execute (if-body), otherwise execute (else-body)" is below:

compile `(expression)` **into a register** `xN`
`beqz xN, else`
compile `(if-body)` **here**
`j endif` (that is, jump straight to endif)
`else:`
compile `(else-body)` **here**
`endif`.

And **while loops** are also similar: if we want to execute a statement "do (while-body) while (expression) is true," then we can run this list of commands:

compile `(expression)` **into a register** `xN`
`beqz xN, endwhile`
compile `(while-body)` **here**
`j while`
`endwhile`.

But we can have less branch/jump instructions by putting the comparison at the end instead, as below:

`j compare` (ensuring that we still check the (expression) condition first)
`loop`:
compile `(while-body)` **here**
`compare:`
compile `(expression)` **into a register** `xN`
`bnez xN, loop`.

This is nice because there is only one control-flow instruction, which is the `bnez` one at the end.

And what we've learned is enough to implement the **Euclidean algorithm** to find the GCD of two positive integers (which subtracts the smaller of the two numbers from the larger until they are equal). Such a sequence of code can be executed with many different initial values, so it's valuable to try to "call" this snippet multiple times:

Definition 25

A **procedure** or **function** or **subroutine** is a reusable code fragment which performs some specific task. Such a procedure has a named entry point (the name of the function), some number of formal arguments (possibly zero), and local storage, and it returns to the caller when it completes.

Such a **caller** must be able to pass arguments to our procedure and get values back when the procedure finishes, and the convention is to **use registers for both of these purposes**. To make sure procedures are implemented correctly, there are a specific set of rules specified by a **calling convention**, so that all developers can safely call procedures without breaking code.

The RISC-V calling convention gives the registers symbolic names:

Example 26

For example, the registers `x10` through `x17` have the symbolic names `a0` through `a7`, and they're known as "argument registers." We use the first two of these, `x10` and `x11` for returning values from our function.

To illustrate this idea of "calling" a little more, suppose that we've written a code fragment that sums up the elements of an array, given the necessary variables above. If we wanted to execute this code fragment in a larger program, we could put a label that led us straight to the "sum" fragment, but the problem is that **we need to know how to go back to whatever location we were at in our main code, even if we call the sum function multiple times**.

So we need to remember our return address (that is, when our procedure was called from), and that's done through the **RA** (return address) register. Every time a procedure is called, our current address is stored, and we do a **procedure call** by using the command `jal ra, label`. This puts the address (4+the procedure call address) in our register `ra`, so that the procedure will return to the correct next command by running `jr ra` at conclusion. (By convention, `x1` is often used for storing this return address.)

One issue, though, is that we shouldn't have to worry about saving registers for **nested procedures**: each procedure should be able to use all registers, regardless of how any of its procedures are acting! So when a procedure is running, the values of the relevant registers must be stored and restored at the end of the procedure; it is the **callee**'s responsibility to save registers in spots `x8-9, x18-27` (these have the symbolic name `s0` through `s11`). In contrast, the caller has some registers that it needs to save as well (such as `x1`, the return address, and `x10` through `x17`, the function arguments). So any procedure call needs to have a local storage component, known as an **activation record**, which

stores (in a **stack** in main memory) any memory allocations that don't fit in the registers. Basically, the most current procedure is always at the top of the stack, and it is deallocated when the procedure exits.

This distinction between caller-saved and callee-saved registers is important to keep in mind – the former is not preserved across function calls, but the latter is, so certain arguments must be saved on the stack to preserve values. We'll discuss this more later on in the class.

5 September 10, 2020

We'll conclude our discussion of assembly language today, looking closer at procedures, stacks, and MMIO.

As a reminder, we discussed the **three components of a microprocessor** in an earlier lecture: basically, we have a register file (to store certain information), a main memory (which holds our program and data), and an arithmetic logic unit. But there's one more special register, which we'll discuss now.

Definition 27

The **program counter** is a special register (not part of the main set of 32) which keeps track of the address of our current instruction.

If we're running loads or stores or computation, the program counter is always incremented by 4 (bytes) to get to the next instruction word in memory. But there are some special cases – if we have a control flow instruction, for example, we may need to update the program counter differently.

Example 28

Suppose we have an instruction `blt x1, x2, label`. Then the program counter needs to jump to some address if `x1` $<$ `x2`, and otherwise it gets incremented by 4 (as usual).

Remember that all instructions, including these branch instructions, are stored as 32-bit binary values in memory. So we want to understand how these instructions actually look, and that requires a more schematic understanding of our main memory.

We'll look at the RISC-V calling convention more carefully for this. Remember that we have various symbolic names for our registers: for example, the `a0` through `a7` registers are used for function arguments (out of which `a0` and sometimes `a1` are used for return values), and the `ra` register stores the address that we need to return to after completing a procedure. Other than this, we have the `t0` through `t6` **temporary registers**, the `s0` through `s11` **saved registers**, as well as the stack point `sp`, global pointer `gp`, thread pointer `tp`, and zero register `zero`.

Definition 29

A **caller-saved register** is one that is not preserved across function calls, meaning it must be saved by the caller before a procedure if its value needs to be preserved. Meanwhile, a **callee-saved register** is one that must be preserved, meaning it must be saved by the callee and restored before completing the procedure and returning control.

Examples of caller-saved registers include the argument, temporary, and return address registers, and examples of callee-saved registers include the saved and stack pointer registers.

We also discussed the necessity of **storing certain information** that our code may need to use later, such as local variables, registers that need to be saved, and data that doesn't fit in our 32 registers. The idea is that we allocate

space for **activation records** of each procedure: if procedure A needs to call procedure B, then the activation record for procedure B gets put in memory "above" procedure A's, and once procedure B is done, its record is deallocated. This **stack** (last-in-first-out) structure means that **whatever is at the top of our stack is going to be our current procedure**, and therefore keeping track of memory allocation is easiest.

This RISC-V stack is stored in a specific spot in memory: we have the stack grow from **higher to lower addresses**, and we make sure the stack pointer `sp` always points to the top of the stack. So if we want to add a register's, say `a1`'s, value to the top of the stack, we can run the following "push sequence" commands:

```
addi sp, sp, -4
a1, 0(sp).
```

Then when we want to restore a value or complete a procedure, we can run the following "pop sequence" commands:

```
lw a1, 0(sp)
addi sp, sp, 4.
```

The key point is that **the memory is always available to us, but we need to return it the way that we found it**. So a procedure can add an arbitrarily large activation record, but it must take everything off the stack at the end.

Example 30

The sequence of commands (at the beginning of the procedure) `addi sp, sp, -8`, `sw ra, 0(sp)`, `sw a1, 4(sp)` can be reversed with the exit sequence (at the end of the procedure) `lw ra, 0(sp)`, `lw a1, 4(sp)`, `addi sp, sp, 8`, `jr ra`.

Example 31 (Callee-saved registers)

Suppose we want to implement the function $f(x, y) = (x+3) \,|\, (y+123456)$ by storing intermediate values in the `s0` and `s1` registers.

Then our function will run the following arithmetic commands:

```
addi s0, a0, 3
li s1, 123456
add s1, a1, s1
or a0, s0, s1.
```

However, before we do any of this, we need to make sure to **save the values of the `s` registers**, because they're callee-saved registers! So at the beginning of our function, we should allocate two words of space on the stack and store the `s` registers by running the command `addi sp, sp, -8`, followed by `sw s0, 0(sp)`, and `sw s1, 4(sp)`. Then at the end, we reverse with the exit sequence `lw s0, 0(sp)`, `lw s1, 4(sp)`, `addi sp, sp, 8`, and finally `jr ra`, so that we exit with the stack status as we first saw it.

Remark 32. *At the end of this process, the values of saved `s0` and `s1` still exist in memory, but the stack pointer is moved back to a higher address, so it's effectively like we've cleared those words from memory.*

Example 33 (Caller-saved registers)
Suppose that a caller is storing the values $1, 2$ for x, y, and then we want to call a function **sum** to find $z =$ **sum**(x, y) and finally store $w =$ **sum**(z, y).

There are a few relevant concepts here. First of all, because we want to call another procedure, namely **sum**, we want to make sure to save the relevant registers that might be overridden. So instead of directly running the below code:

```
li a0, 1
a1, 2
jal ra, sum,
```

we want to save the registers `ra` and `a1`. Here, the former needs to be saved so that we can jump back from the **sum** subroutine properly, and the latter needs to be saved by convention because **we are reusing the value of** y **a second time later in the code**. (We don't need to save `a0`, because we never need the value of x again.) That means that we need to run the commands `addi sp, sp, -8`, `sw ra, 0(sp)`, and `sw a1, 4(sp)`.

Remark 34. *An important principle is that* **callers don't see the implementations of the callee's procedures**. *So even if it seems like the most natural way to implement* **sum** *is just to run* `add a0, a0, a1`*, which doesn't modify* `a1`*, convention still tells us that we should save* `a1` *if we need it later.*

After running that `jal ra, sum` command above, we have the value of z in `a0`, and we need to restore the value of y again via `lw a1, r(sp)`. Then we can call `jal ra, sum` again, and that gets the value of w in register `a0`, as desired. We can then finish by just running `lw ra, 0(sp)` and `addi sp, sp, 8` to return the stack and stack pointer to their original state.

In summary, callers are responsible for saving `a`, `t`, and `ra` registers, while callees are responsible for saving `s` registers. But both need to make sure the stack pointer is in the right spot at the end of the procedure.

Fact 35
Two relevant pseudoinstructions are `call func` (for the caller to jump-and-link) and `ret` (for the callee to return from the subroutine).

In the remaining time, we'll discuss some more complicated cases, such as the case of **nested procedure**. The return address `ra` is always supposed to be caller-saved, so we just need to be careful to store and load the value of `ra` before and after calling any procedures. **As long as each procedure maintains and properly restores the relevant data**, the convention will hold up in this case.

For another complex situation, we can also consider data structures that are too large to store in our registers. For example, if we want to find the maximum value in a large array, we can just keep track of the two variables **(base address)** and **(size)**, rather than passing the whole array in. Our "maximum" subroutine would then do the following:

- Start off with the base address and the size of the array as the values of `a0`, `a1`.
- Initialize some local variables `t0`, `t1` to zero, corresponding to "i," our current array index, and "max," our rolling maximum.
- In our loop, check whether `t0` $<$ `a1` (meaning that we still have something in the array to compare against).
- Load the word from the location `a0` + 4·`t0` (which is the address of the ith element of our array).

- Compare the loaded word to our maximum value stored in `t1`, and move the value if necessary.
- Increment the index `t0` by one and check our loop condition again.

This finishes our discussion of the RISC-V instruction architecture, and now we have the necessary language to be able to translate any high-level program into RISC-V! It's just good for us to keep in mind that whenever we have data structures like dictionaries, they're always going to be implemented with blocks of memory with words that refer to various addresses.

We'll finish by understanding how this all fits together into the memory layout that we're using. Most languages actually use **several distinct memory regions** for data:

- Stack (for procedure calls),
- Static (for global variables that exist throughout the program),
- Heap (for dynamically-allocated data). In languages like C, this heap needs to be manually managed with the commands `malloc()` and `free()`, but in languages like Python or Java, the system will automatically make new space when we declare an object like a dictionary.
- The text of our code.

In RISC-V, we put the **text, static, and heap regions** in memory consecutively, starting from the smallest address `0x0`. (In other words, the heap grows towards higher addresses as we allocate more memory for it.) This is why, in contrast, the stack starts from the highest address `0xFF..F` and grows towards lower addresses – we can allow flexible space for both the heap and the stack.

Finally, we have some specific pointers: the **stack pointer** `sp` points to the top of our stack, the **global pointer** `gp` points to the beginning of our static region (which we won't use in 6.004), and the **program counter** `pc` looks at the line of code that we're currently executing.

In our next lab, we'll also get to think about how this all relates to **inputs and outputs**. Recall that our high-level programs are translated using RISC-V (or other ISA) into instructions that our processor knows how to use. But there are other pieces of hardware that we may interact with, such as a display (to print to) or a keyboard (to get inputs from), or other peripherals. So **special dedicated addresses** are used to represent these input/output (I/O) devices: we can then just use `lw` or `sw` instructions to read or write accordingly. This is known as **memory mapped IO** (or **MMIO** for short), and this means that we can only use those specific addresses for input/output, not regular storage. And there are special I/O devices that respond to memory requests, rather than main memory.

Fact 36

Each time we perform a `lw` instruction from the address `0x4000 4000`, we will read one signed word from a keyboard. Meanwhile, if we perform a `sw` instruction to **0x4000 0000**, we print an ASCII character, and if we perform a `sw` instruction to `0x4000 0004` or `0x4000 0008`, respectively, we print a decimal or hexadecimal number.

Example 37

Here is an example of a program that reads in two inputs from our keyboard, adds them together, and then displays the result on an output monitor.

As always, we need to remember that loads and stores must be done with addresses relative to the value stored in some register.

`li t0, 0x40004000` (loads the address of the "read port")
`lw a0, 0(t0)` (reads in the first input word)
`lw a1, 0(t0)` (reads in the second word)
`add a0, a0, a1`
`li t0, 0x40000004` (loads the address of the "write port")
`sw a0, 0(t0)`.

And we can use MMIO for certain **performance measures** as well: we can get the **instruction count** (loading from 0x4000 5000) to learn how many instructions have been executed since the program started, or get the **performance counter** (loading from 0x4000 6000) to count the number of instructions since we have turned the counter on (which is toggled off/on by storing 0 or 1 in 0x4000 6004). So this is a way for us to track how long a piece of code takes to run!

With all of this knowledge about assembly language, we now understand what we need to build, and we're going to try to understand how to build the hardware to implement those assembly instructions (from the bottom up) in the next few lectures.

6 September 15, 2020

Today, we're going to switch gears and turn away from assembly language (which was the first module of our course). We'll start discussing **digital design** now, looking at combinational and sequential circuits, and understanding why the ISA structures are set up in the way they are. Specifically, one goal is to understand (from the hardware point of view) what constraints exist in design and what makes our systems most efficient.

To reiterate a point from earlier on in the class, even if we don't build hardware in our professional lives in the future, we'll probably still be designing some cutting-edge systems, so we'll need good knowledge of hardware. More importantly, in the modern world, systems are becoming more specialized. Performance improvements from general-purpose processors are reaching their limits – if we look at the performance of processors since the 80s, we've had a dramatic (50000-fold) improvement, mostly first following Moore's law (telling us we can have an exponential number of transistors on the chip) and then following Dennard scaling. But in the present, those kinds of performance improvements have basically stopped, and companies are now starting to build custom hardware to meet their needs. So we will likely need to **use and design specialized hardware**, rather than just sticking to general-purpose processors.

Thus, the first topic of this module is **digital abstraction** – building digital conventions for storing and processing information.

Definition 38
Analog systems represent and process information by keeping track of continuous signals, while **digital** systems do so by recording in discrete symbols.

This means that an analog system keeping track of voltage, or current, or temperature, or pressure, can record down the exact value of any of these quantities and use that information in its processing and its circuits. But a digital system encodes this kind of information by keeping track only of **ranges of physical quantities**, and we often do this in binary (recording down a "0" or "1").

So, for instance, if our digital device measures the voltage V through some component of a circuit to be larger than some value, we encode that as a 1, and if we measure it to be below that value, it is a 0. But the key advantage to keep in mind for these digital systems is that **they should tolerate noise**.

Example 39

Consider an **analog audio equalizer**, which takes in some voltage signal which represents sound pressure (in decibels) over time. This signal goes into our system, and we use the audio equalizer to adjust the amplitude of various frequencies of the signal.

(For example, maybe we want to increase our bass or our treble sound in a music track.) The way this works in practice is that we have some **band filters** to get low, middle, or high frequencies that filter different components of our sound. Then we amplify each of those components with some gain. Finally, we mix these together and get some output.

But in reality, there are going to be some differences between the output that we have and what we expect. There are many reasons why such a system may not work: various sources of noise will make their way into the system, manufacturing variations for resistors or capacitors may change the gain, and degradation of the different components also alters the sound over the time. So this is not a good way to build reliable systems, because we want to be able to process the same information over and over again, without having the output dominated by noise.

Fact 40

By switching to digital systems, we're making an abstraction that allows us to tolerate some amount of noise. So even though we want our bits (0 and 1) to behave perfectly, there is still going to be some real-world noise, and we're just trying to engineer the world to behave in a digital way.

So let's think about how we can take a continuous quantity like **voltage** and turn it into a discrete (digital) quantity (like **0** or **1**). Whatever convention we use should be uniformly adapted across all circuit components in our hardware.

Example 41

Let's try defining a threshold voltage V_T, and we can say that every component and wire in our system interprets $V < V_T$ as a "0" and $V \geq V_T$ as a "1."

This isn't a great system, because a slight amount of noise can flip a 0 to a 1 or vice versa, if the voltage is right at the edge of V_T. So we'll add some elbow room:

Example 42

Now, let's try having two thresholds $V_L < V_H$ (low and high). If the voltage satisfies $V \leq V_L$, we interpret that as a 0, and if the voltage is $V \geq V_H$, we interpret that as a 1. If we're in between the two values, we say that the bit is undefined.

Building a device that distinguishes between these voltages is not hard to do, but this isn't a great system either, because of one specific problem: **differences betewen inputs and outputs**. There are still going to be values right around the valid-invalid regions! If we have an input device which sends a valid 0, but it's transmitting a signal at voltage $V_L - \varepsilon$, it's possible that noise will turn it into $V_L + \varepsilon$ by the time it hits another (output) device in the circuit. So the 0 still turns into an undefined bit, and we run into issues with degradation again.

To address this, we make a distinction between input and output bits:

Fact 43

Any convention for our digital systems should use **narrower ranges for output voltages than input voltages**, so signals will still be valid with noise.

We'll use four different thresholds now, which we denote V_{OL}, V_{OH} (for "output low" and "output high") and V_{IL}, V_{IH} (for "input low" and "input high"). But now, a digital output device outputs a 0 if the voltage V is below V_{OL} and a 1 if it's above V_{OH}, and similarly a digital input device reads a 0 if the voltage is below V_{IL} and a 1 is above V_{IH}. **To avoid the previous problem**, we need to make sure that

$$V_{OL} < V_{IL} < V_{IH} < V_{OH}.$$

In words, digital devices are supposed to accept "marginal inputs" and "unquestionable outputs," so that an output of a 0 or 1 from some device won't turn into an undefined or opposite bit when it is read into another input device.

Fact 44

The key point is that digital systems are **restorative**: noise does not accumulate like it does in analog systems.

For example, if we have a digital circuit which takes in some voltage V and then applies some functions f and g, we might get composite noise that looks like

$$g(f(V+\varepsilon_1)+\varepsilon_2)$$

by the time we're done running through. But in a digital system, $f(V+\varepsilon_1)$ and $f(V)$ are the same as long as our thresholds are properly calibrated, and similarly $g(f(V)+\varepsilon_2)$ and $g(f(V))$ will be the same, too. So **the noise will be canceled at each stage**, and this is important so that (for example) index counters won't randomly decrease by 1 because of accumulated noise.

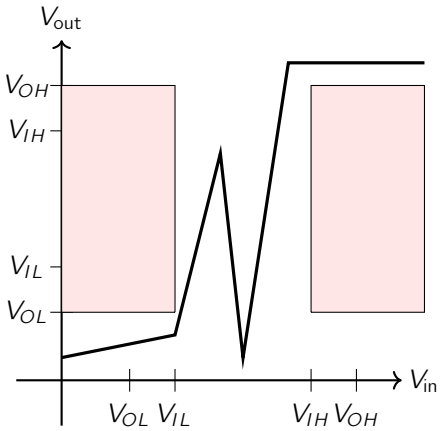
Remark 45. *Note that any noise cancellation (that is, any system that is "fighting noise") requires* **active components**, *which are feeding energy into the system from an external source.*

We can formalize this whole idea further by analyzing simple digital devices. Let's consider a **buffer**, which copies the input value into the output (so feeding in a 0 gives us a 0, and feeding in a 1 gives us a 1).

Definition 46

A **voltage transfer characteristic (VTC)** plots the output voltage as a function of the input voltage.

We find this by inputting in various values of V_{in} and then waiting for transient behavior to go away, and recording the subsequent value as V_{out}. Here's an example of a VTC:



Even though the buffer is just supposed to copy over the input value, it's not good enough to just copy the voltage from V_{in} to V_{out} because of noise. And note that our VTC needs to avoid the **forbidden zones** which are shaded above: those are the regions where we feed in a valid input, but we end up with an invalid output.

Because V_{OH} and V_{OL} have a difference larger than V_{IL} and V_{IH}, our VTC needs to "rise very quickly" in the middle of the diagram: the value of $\left|\frac{dV_{out}}{dV_{in}}\right|$, also called the **gain**, must be larger than 1 at some point in the middle, so that means **we need to have an active device** which feeds in energy (even though, again, we're just copying a bit!). But also, this VTC can do anything in the range of inputs between V_{IL} and V_{IH}: it's perfectly fine to have valid digital outputs for invalid digital inputs, for example. (Really, all that matters here is that whenever our input voltage is less than V_{IL} or larger than V_{IH}, we get the right answer.)

So we can start looking at **digital circuits** now, and in the next 7 lectures or so, we'll study different types of digital circuits. There are basically two kinds:

- **Combinational** circuits do not have memory, so each output of the circuit is only a function of the input values. For example, the **buffer** and **inverter** each take in a bit, and they output a bit based on the input. We can also have gates like **AND**, which take in multiple inputs.
- **Sequential** circuits have some internal memory or state, so the output of the device depends on both the internal state and on the inputs that are fed in.

We'll start looking at combinational circuits right now and continue this over the next few lectures.

Definition 47

A **combinational device** is a circuit element with digital inputs and outputs (working according to the above specification), as well as:

- A **functional specification** which exhaustively specifies output behavior for all possible input values, and
- A **timing specification**, which tells us (at least) how much time we need before the output has a valid digital value. This time t_{PD} is called the **propagation delay**.

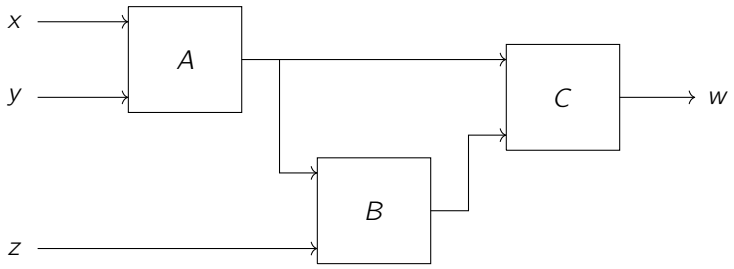
For example, a combinational device's specification can consist of the following information: "The device will output a 1 if at least 2 of the 3 inputs are 1s, and otherwise it will output 0. A valid output will be generated within 2 nanoseconds." The above components and rules in the definition are called the **static discipline**, and the reason we need them is that combinational devices can be put together and connected in large, predictable circuits.

Definition 48

A set of connected elements is called a **combinational device** if:

- each circuit element is a combinational device,
- each input is connected either to one output or to a fixed constant 0 or 1, and
- there are no directed cycles in the circuit.

Here is an example:



This device is indeed a combinational device as long as A, B, C are combinational devices, because we have digital inputs and outputs, and we can derive a functional description by writing out the function f in terms of the functions of the devices f_A, f_B, f_C: it will be

$$f(x, y, z) = f_C(f_A(x, y), f_B(f_A(x, y), z)).$$

The point is that we do indeed have a functional specification, and now we can find the propagation delay by thinking about the longest path through the circuit: in this case it's the one that goes through all three combinational devices, and we have

$$t_{PD} = t_{PD,A} + t_{PD,B} + t_{PD,C}.$$

Let's think about what goes wrong if we violate any of the conditions for a set of circuit elements being a combinational device. It's important that every input is connected to exactly one output – otherwise, our combinational device may stop working if some input isn't connected properly to the rest of the system, or we might overwrite important data or have different outputs conflicting with each other. (But this is a sufficient condition, not a necessary one – sometimes it's okay for us to have combinational circuits that run in parallel, as long as we don't get conflicting data.) And it's pretty easy to come up with examples where having cycles in our combinational circuits is bad: for example, if an output feeds back into an earlier combinational device, we may get oscillating behavior.

We'll start talking about functional specifications next lecture, but the main point is that we can specify functions in various ways. There are two main systematic approaches we'll take: we can use **truth tables**, where we enumerate outputs for all possible inputs, and **Boolean expressions**, which use AND, OR, and NOT operations. (We'll discuss / review Boolean algebra next time.) The point is that **any combinational circuit can be expressed with Boolean expressions and truth tables**, and the former is easier to optimize.

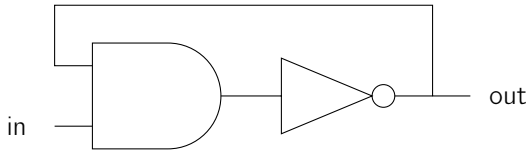
In summary, digital systems tolerate noise much better than analog systems, and we can encode voltages by using certain margins and voltage levels. Next time, we'll start studying combinational devices in more depth.

7 September 17, 2020

We'll continue our journey into digital design today, looking at **Boolean algebra and logic synthesis**. Last time, we came up with a digital signaling **discipline** that was robust against noise and helped interface different input and

output devices together, and today we'll get into the **functional** aspect of combinational circuits.

Recall that combinational devices are circuits with a **static discipline** containing digital inputs and outputs, a functional specification, and a timing specification (which contains at least the propagation delay for the device). We then composed such devices together, with the rules that all components must be combinational, all inputs must be connected to one output or a constant, and there must not be directed cycles. Feedback loops like the one below explain why certain circuits can't always reach a known or stable value if they contain directed cycles:



This circuit contains an AND gate and an inverter (we can tell the difference between an inverter and a buffer by the circle at the tip). The point is that if our input bit is 1, our circuit will not settle at a particular output value (or voltage), meaning there is no well-defined propagation delay.

So now that we can build large combinational circuits, we're going to focus today on how to specify the function of such a device. As mentioned last time, we can use **truth tables** (which enumerate all output values for all possible input values).

Example 49

Suppose we have a combinational device which takes in three bits A, B, C. If $C = 0$, then the output Y is taken to be A, and otherwise it's taken to be B.

Then the truth table can be written out as follows: we try every possible combination (A, B, C) and mark down the value of Y. (Usually, they are written vertically instead of horizontally – the rotation is just to save space.)

C	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1
A	0	1	0	1	0	1	0	1
Y	0	1	0	1	0	0	1	1

While this is a valid approach, and we can write down a truth table for any such function (with 2^n possible inputs if we have n binary inputs), we're going to talk about a second form, **Boolean expressions**, today. Boolean expressions are equations that contain binary variables (0 or 1), as well as three operations AND (denoted $\cdot$), OR, (denoted $+$), and NOT (denoted with an overbar). For example, the above function can also be written as

$$Y = \overline{C} \cdot A + C \cdot B.$$

It turns out that **all combinational functxions can be specified with Boolean expressions**, and this is nice because writing out expressions takes less space, and manipulating them is relatively simple.

Before we discuss this more, we want to return to the question of **how fast circuits are**. We said before that we need a **propagation delay**, an upper bound on how long it takes to get from valid inputs to valid outputs. If we plot the output voltage of a device V_{out} as a function of the input voltage V_{in}, there will be some lag or transit time between the time when the input value becomes a valid input bit, and the time when the output value is a valid output bit. (Remember that this means we need to measure how long it takes for the voltage to switch from V_{OH} to V_{OL}, or vice versa.)

This propagation delay t_{PD} should be minimized when possible, and the reason for this is that many processors run at **certain clock frequencies**. For example, if the frequency f_{CLK} is about 4 GHz, then $t_{CLK} = \frac{1}{f_{CLK}} = 250$ picoseconds is the time between **clock periods** of the processor. In other words, every t_{CLK}, our processor will change the inputs into our system and read the outputs. So we need an upper bound to ensure that we can run in the necessary time in the worst-case. The idea is that we have a kind of **combinational contract**, which tells us that there are no promises in some short amount of time after we change our inputs, but that we will have the correct, valid outputs after a while.

There's also another quantity we will soon care about:

Definition 50

The **contamination delay** t_{CD} is a **lower** bound on how long it takes to go from invalid inputs to invalid outputs.

This will be more important when we have state elements in sequential circuits – it's okay for us to ignore this t_{CD} for now.

So now we'll return to **Boolean algebra**, which is a way to describe and work with Boolean expressions.

Definition 51

The **AND**, **OR**, and **NOT** operators are defined as follows:

- AND is a binary operator that outputs 1 if both input bits are 1, and 0 otherwise.
- OR is a binary operator that outputs 1 if either input bit is a 1, and 0 otherwise.
- NOT is a unary operator which outputs the opposite bit that is input.

Remark 52. *We may also see "NOT a" written as $\neg a$ or $!a$ or $\sim a$ rather than $\overline{a}$, and we may see different notation for AND and OR from logicians ($\wedge, \vee$) and programmers (& or &&, and | or ||).*

It turns out that all Boolean algebra can be derived from AND, OR, and NOT, and we can specify that in a rigorous mathematical way. Specifically, we use the following set of axioms:

- **Identity**: $a \cdot 1 = a$ and $a + 0 = a$ for all a,
- **Null**: $a \cdot 0 = 0$ and $a + 1 = 1$ for all a, where we remember that $+$ is OR,
- **Negation**: $\overline{0} = 1$ and $\overline{1} = 0$.

It turns out that there's a curious symmetry between the first and second axiom in each bullet point: if we switch ORs with ANDs, and we replace 0s with 1s, that gives us another true statement. This is called a **duality principle**, and because it holds for the axioms (which derive all other expressions), **this duality principle holds for all expressions**!

And either using truth tables or axioms, we can derive a few useful properties: **commutativity, associativity, and distributivity** appear much like they do in "normal" addition and multiplication, except that the distributivity laws now look like

$$a \cdot (b + c) = a \cdot b + a \cdot c, \quad a + (b \cdot c) = (a + b) \cdot (a + c).$$

(So OR and AND each distribute over each other!) We also have the following properties:

- **Complements**: $a \cdot \overline{a} = 0$ and $a + \overline{a} = 1$,
- **Absorption**: $a \cdot (a + b) + a$ and $a + a \cdot b = a$,
- **Reduction**: $a \cdot b + a \cdot \overline{b} = a$ and $(a + b) \cdot (a + \overline{b}) = a$,

- **De Morgan's Law**: $\overline{a \cdot b} = \overline{a} + \overline{b}$, and $\overline{a + b} = \overline{a} \cdot \overline{b}$.

We should try to familiarize ourselves with these statements by proving them – this can be done either by working algebraically, from the axioms, or we can write out truth tables and verify that everything matches up.

Remark 53. *Just like in ordinary arithmetic, multiplication (AND) precedes addition (OR) in the absence of parentheses.*

Example 54

Suppose we want to show that $a \cdot b + a \cdot \overline{b} = a$ for all bits a, b, using some of the other results.

Then the relevant calculation is that

$$a \cdot b + a \cdot \overline{b} = a \cdot (b + \overline{b}) = a \cdot 1 = a,$$

by using distributivity, complements, and identity respectively.

Now we're ready to think about the statement we made earlier, which is that **truth tables and Boolean expressions are equivalent**. Going from a Boolean expression to a truth table can be done by evaluating every possible value, but the converse is not as clear.

Fact 55

Given a truth table, we can produce an equivalent Boolean expression by writing a **sum of product terms**, where each term covers a single 1 in the truth table.

For example, suppose we have Y as a function of the bits C, B, A, like in the above example. Then if $Y = 1$ at $(C, B, A) = (0, 0, 1)$, then the expression

$$\overline{C} \cdot \overline{B} \cdot A$$

is a **product term** (which only consists of variables or their negations ANDed together) which takes on the value 1 at $(0, 0, 1)$ and 0 otherwise. So if we want to write down an expression for Y, we can **repeat this process for all rows where the product is** 1, and that gives us an expression that looks like

$$Y = \overline{C}\overline{B}A + \overline{C}B\overline{A} + CB\overline{A} + CBA.$$

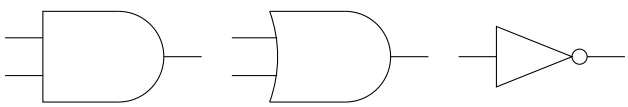
(The product is implicit here, just like in integer algebra.) This kind of expression is called the function's (disjunctive) **normal form**: it's unique, but it might not be the simplest or shortest expression for the function.

And now we can use Boolean expressions to think about building good circuits: the process of building logic circuits from expressions is called **logic synthesis**.

Definition 56

A **logic gate** is a combinational device that represents a simple function (such as AND, OR, or the inverter NOT).
A **logic diagram** is a circuit representation of a Boolean expression with logic gates and wires.

The AND, OR, and inverter (or NOT) gates are shown below, in order:



We may also represent AND and OR gates with more than two inputs (by drawing more lines). And the point of using these basic gates is that we can implement any combinational function with them – we don't need any other ones, because this set of gates is **universal**. Basically, we take any **sum-of-products** expression like the one we've constructed in the normal form above by using three levels of gates:

- Inverters, to invert some individual variables in our expression,
- ANDs, to make products of our individual terms,
- OR, to add all of these products together.

But this isn't a great way to implement all combinational functions, especially when we have a lot of product terms, and often we want to simplify our Boolean expression to avoid using too many excess gates.

Definition 57
A **minimal sum-of-products** is an expression in sum-of-products form with the smallest possible number of AND and OR operators.

A single function can have multiple minimal sum-of-product forms, and there are algorithms to compute them, but we'll be dealing with small expressions (of 3 or 4 variables) in this class, in which we should be able to use algebraic manipulation (reduction, complements, and so on) to do the minimization manually.

Remark 58. *If we compare different inputs in our truth table, we can sometimes find "don't care" values, in which a certain input does not change the output as long as other bits are fixed. This helps us combine sum-of-product terms together: for example, if* $(C, B, A) = (0, 0, 1)$ *and* $(0, 1, 1)$ *both give us the same output of* 1*, then that corresponds to a term of* $\overline{C}A$ *instead of* $\overline{C}\,\overline{B}A + \overline{C}BA$.

And it turns out that using more logic levels is often a way to reduce the number of gates we're dealing with than working with sum-of-product expressions directly:

Example 59
Suppose we have a Boolean expression like

$$F = AC + BC + AD + BD.$$

We can check that F is in minimal sum-of-products form, and this requires us to do 7 ANDs and ORs in direct implementation. But we can also rearrange this expression to

$$= (A + B) \cdot C + (A + B) \cdot D = (A + B) \cdot (C + D),$$

which cut the number of expressions down to 5 and then 3 gates. So there's a **tradeoff** here: reducing the number of gates may be good in terms of area, but it may increase delay because of the multiple levels.

Remark 60. *We won't look at the multi-level optimization described here in this class, because there are* **synthesis tools** *which do the job. Such tools take in a high-level circuit specification (in practice we'll use Minispec in this class), the cell library (basically what gates we are allowed to use), and the optimization goals (some given tradeoff between area, delay, and power), and it'll output an optimized circuit implementation.*

There are other common gates in the **cell library** that we may like to use as well: for example, the XOR of two Boolean variables, denoted $Z = A \oplus B$, is 1 if exactly one of A and B is 1, and we can rewrite this as

$$Z = A\overline{B} + \overline{A}B.$$

We also have "inverting logic" gates, called NAND and NOR, which output $\overline{AB}$ and $\overline{A+B}$, respectively. and even XNOR. ("Inverted" gates have an extra circle at the output in schematic diagrams.) But it turns out that NANDs and NORs are universal on their own, too: **any logic function can be implemented using only NAND gates, and it can also be implemented using only NOR gates**. It's a good exercise for us to think about how to do this: for example, if we connect the two inputs of a NAND, we get a NOT, and then we can attach a NOT to a NAND to get an AND, and so on.

Fact 61

A **standard cell library** often includes a set of input gates (each allowed to have 1, 2, 4, or some other number of inputs), their area, and their time delay.

It turns out that current technology can produce **faster inverting gates** than their non-inverting counterparts, and gates with more inputs take longer and consume more area. So there are now lots of design tradeoffs we can make, but we'll often rely on tools for that. The point is that **synthesizing optimized circuits is a very hard problem**.

Next lecture, we'll see how to implement these gates using transistors in what's called CMOS (complementary metal-oxide semiconductor) technology.

8 September 22, 2020

So far in these lectures, we've discussed (at a high level) how combinational devices work and how to construct them with combinations of gates and wires. Today, we'll go one step deeper and understand **how those gates are built** – in particular, we're looking at **CMOS technology**, which is the fabrication technology used in basically all digital devices today. In particular, we'll examine field effect transistors, which are the underpinning device of all of these gates.

This is the last lecture whose material is included in next Thursday's quiz, but only the first part of the lecture (functional behavior of transistors for building logic gates) will be included on that quiz. The second part of the lecture (understanding area, power consumption, and delay for gates) is a demystification section going "one level deeper" into the physics, so it will not be quizzed. (Even though transistors are very complicated, we'll stick to a simplified description.)

Fact 62

If we haven't taken 6.002 or don't remember some of the physical properties of resistors or capacitors, parts of this lecture may be confusing, so it would be good for us to refresh our understanding of circuits.

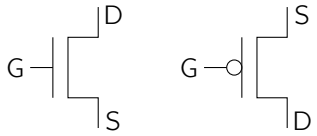
If we disassemble a laptop computer or phone, we'll see that there are lots of **packaged chips** inside that are connected together on a circuit board. But even an individual chip has many different components: there's a metal plate on one side for thermal dissipation, lots of pins attached the other side (to connect to other devices, like main memory), and inside, there is a small **silicon die** with area around 100 to 400 square millimeters and thickness around 100 microns. This is a thin slab of silicon, and if we take a cross-section of it, we see that there are about 6 to 15 different **metal layers**, corresponding to structures built at different levels. In particular, at the bottom level, we have our **transistors**, which are the smallest devices of the chip (on average, about 32 nanometers wide). In fact, a standard chip today has about 1.5 billion FETs (field effect transistors)!

These transistors are difficult to analyze in full detail, but at a high level, we can think of them as **voltage-controlled switches**. In other words, we essentially have a circuit component which can be turned on or off.

Definition 63

A **field effect transistor** (FET) has three terminals known as a **gate** (G), a **source** (S), and a **drain** (D). There are two varieties of FETs, called **nFETs** and **pFETs**. In nFETs, high voltages create conducting paths (wires) between the source and the drain, while low voltages leave an open circuit. On the other hand, in pFETs, low voltages create conducting paths, while high voltages leave the circuit open.

The schematic diagrams for the nFET and pFET, respectively, are shown below.



In other words, feeding a 1 into the nFET on the left gives us a path from D to S, while a 0 leaves the circuit closed. But feeding a 0 into the pFET gives us a path, while a 1 does not.

These circuit diagrams are interesting in that the source and drain are drawn the same way except for the labels S and D. And indeed, the terminals are basically identical, because the device itself is symmetric! The source and drain terminals are known as the **diffusion terminals**, and all that matters here is which one is at a higher voltage. To be more precise, **by convention, the source (S) terminal is defined to be the diffusion terminal at lower voltage for nFETs and the terminal at higher voltage for pFETs**. The point of this is that we'll draw our circuits so that the voltage source will be at the top of our circuit diagrams, and the ground will be at the bottom, so the **terminals closer to the bottom of the page or screen will always be at lower voltage** – this removes any source of ambiguity. And the reason we go to trouble of labeling transistor terminals with "source" and "drain" is that we can just look at the voltage between **gate and source** terminals to define the behavior of our FETs.

Fact 64

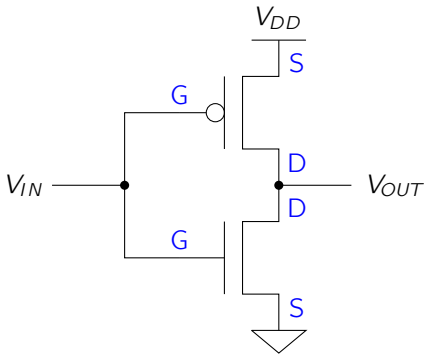
All FETs have a **threshold voltage** V_{TH}, defined as follows. If an nFET is in a state where the voltage $V_{GS} = V_G - V_S$ (between the gate and source) exceeds this threshold V_{TH}, then it is **on** (connected), and otherwise it is **off** (disconnected). Similarly, a pFET is **on** if the voltage $V_{SG} = V_S - V_G$ is more than V_{TH} and **off** otherwise.

(We use V_{GS} in one case and V_{SG} in the other, corresponding to whichever terminal is at the higher voltage.) This model is very simplified, but it's good enough for us to build gates with it (we'll see a more sophisticated explanation with resistors and capacitors soon).

Example 65

Let's try to understand how circuits can compute using transistors.

Consider the following diagram, where we assume the source (highest voltage) is at the top of the gate.



If we want to understand what this circuit computes, we start by labeling the diffusion terminals, which we've done in blue. This is easiest to do if we put the source voltage at the top of the diagram and the ground voltage at the bottom, as we've done here.

We'll assume that the transistors have the same threshold voltage, and suppose that $V_{TH} < \frac{1}{2}V_{DD}$. Then we can try "sweeping" the value of the input voltage V_{IN} from low to high and seeing what the network behavior looks like:

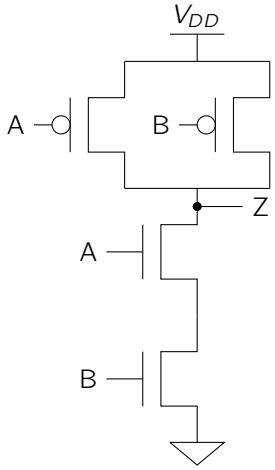
- If our input voltage is lower than V_{TH}, the pFET should be on (the voltage difference between the source and input is large enough), and then nFET should be off (because the voltage difference is too small). So the output is connected to the source voltage but not ground, meaning that V_{OUT} is basically V_{DD}.
- In the other extreme, when $V_{IN} > V_{DD} - V_{TH}$, then the pFET should be off and the nFET should be on, so V_{OUT} is 0 (because the output is connected to the ground but not the the source voltage).

In other words, if we plot V_{OUT} as a function of V_{IN}, we get a high voltage when $V_{IN} < V_{TH}$, a low voltage when $V_{IN} > V_{DD} - V_{TH}$, with some gradual change in voltage in the middle. So if we want this circuit to be a **good inverter**, we want V_{TH} to be exactly $\frac{1}{2}V_{DD}$, because that gives us a very sharp transition between high and low voltages, which gives us great margins for setting input ranges. All in all, this is a circuit which (set up properly) serves as a **(CMOS) inverter**.

Fact 66

Metal-oxide-semiconductor field-effect transistors (abbreviated **MOSFETs**) are the most common transistors used today, and thus "nFET" and "pFET"are often written as "nMOS" and "pMOS." In particular, **CMOS** stands for "complementary metal-oxide semiconductor, and we'll see soon what "complementary" means.

For now, though, let's look at another circuit and see what the relevant computation looks like:



Here, there are two input bits, A and B, and one output bit, Z. We can check the following properties:

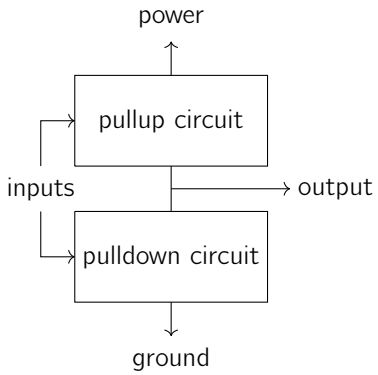
- If A and B are both low (0) inputs, then the bottom two gates are open (off), while the top two gates are closed (on). That means that the voltage at Z will be basically the voltage at V_{DD}, so Z will output 1.
- Looking at the other possibilities, if **either** of the top two gates are closed, Z will be connected to V_{DD}. But if the pFETs are open and the nFETS are are closed (which occurs if both inputs are high voltages), then Z is connected to the ground. So Z is 0 when A, B are both 1 and 1 otherwise.

This is the **CMOS NAND gate**! Remember that this is a universal gate, so we can theoretically build any combinational device from it.

Now that we have some practice, we can learn what the general recipe looks like for **building CMOS gates**, instead of continuing to do lots of examples.

- In a **CMOS gate**, we must have **pullup** and **pulldown** networks, so that the pullup is on when the pulldown is office, and vice versa. (Here is where the word "complementary" comes into play: the two networks need to be related in a particular way.)
- The pullup network connects the output(s) to the power supply, and the pulldown network connects the output(s) to ground. The **same inputs** are fed into both circuits.
- If the pullup is on and the pulldown is off, then we will drive the output to 1 (a high voltage). Similarly, if the pullup is off and the pulldown is on, then we will drive the output to 0 (a low voltage).
- But there are two problem cases. If it's possible for both the pulldown and the pullup networks to be conducting, then the power supply and ground are both connected to the output, so we'll burn the circuit and consume a lot of power. (This is called a "driven X" – we don't know what the actual output voltage will be without knowing the specific characteristics of our transistors.) And if both networks are off, this is bad too, because the output will be **disconnected**. So we need to avoid those by designing the pullup and pulldown in a (logically) complementary way.
- **In CMOS, we always use pFETs to implement the pullup network and nFETs to implement the pulldown network**.

Schematically, we should imagine that all CMOS gates look something like this:



Example 67

Let's consider a few examples of bad CMOS gates to understand what could go wrong.

1. Suppose we tried to build a NAND gate, but instead of putting the pFETs in parallel and the nFETs in series, we put both transistors in series. Now if we set both A and B to the same input bit, the circuit works, but if we set A and B to different bits, we will get a connection from V_{DD} to the output Z (through one of the pFETs), and we'll also get one from Z to ground (through the corresponding other nFET). So this circuit shorts out – somehow our pullup and pulldown networks are not complementary.

2. On the other hand, consider a circuit similar to the inverter above, but where the pullup contains an nFET and the pulldown contains a pFET. Here is where the **threshold voltage** comes in. Notice that the source (S) terminals are now closer to the V_{OUT} voltage than the gate (G) terminals, so we'll see that something goes wrong when we try plugging in a sample value of our input voltage. If we let V_{IN} run from 0 to 1, then the nFET in the pullup network turns on (makes a connection) initially, but then when the V_{OUT} voltage reaches $V_{DD} - V_{TH}$, the nFET will close! In fact, we find that we can only "pull up" the voltage to $V_{DD} - V_{TH}$, and we only "pull down" the voltage to V_{TH}. So this inverter goes into the "forbidden" zones when we plot V_{OUT} as a function of V_{IN}: it will lose a lot of range because it's not amplifying voltage differences, and therefore it's not noise-resistant.

So to fill in the last piece of the puzzle, we can understand how to construct gates by looking at CMOS complements. The rules are as follows:

- Going from A to $\overline{A}$ is switching between an nFET and a pFET (one is on when A is high, while the other is on when A is low).
- Putting nFETs together in series means we create an AND function $A \cdot B$ (since both nFETs need to be on), while putting pFETs together in parallel creates $\overline{A} + \overline{B} = \overline{A \cdot B}$ (since either pFET can be on for successful connection).
- Much in the same way, putting nFETs in parallel is complementary to putting pFETs in series, corresponding to $A + B$ and $\overline{AB} = \overline{A + B}$, respectively.

So the **recipe for creating a general CMOS gate** is to first figure out a pullup network that implements the desired behavior. (For example, if we need to implement $F = \overline{A} + \overline{BC}$, we use pFETs to generate a high output by putting B and C pFETs in series, and then connecting that with the A pFET in parallel.) And then, we replace pFETs with nFETs, replace series network connections with parallel ones and vice versa, to create the corresponding pulldown network with nFETs. (In this example, we end up with B and C nFETs in parallel, connected in series with A.) Combining them together as the schematic diagram tells us will give us the desired CMOS gate.

Fact 68

Note that individual CMOS gates **cannot implement arbitrary Boolean functions**! This is because CMOS gates are **inverting**: if we have rising inputs (from 0 to 1), that can only give us falling outputs (from 1 to 0).

To understand this, notice that nFETs can only go from "off" to "on," and pFETs can only go from "on" to "off," when an input rises in voltage. So pulldowns may connect us to the ground, or the pullup may disconnect us from V_{DD}, but **in any case the output does not rise**. So we cannot build an AND gate, for example, without putting multiple CMOS gates together.

This concludes the examinable material, and we'll finish this lecture with a quick electrical description of how CMOS gates work at a more physical level.

Fact 69

Looking at a physical design, a MOSFET consists of a slab of silicon (the semiconductor, **doped** with certain materials to give it free electrons (n-type) or positive charge carriers (p-type)) with two terminals (source and drain) connected to that silicon. The gate is a thin piece of metal, and there are "gaps" between the source, gate, and drain. In addition, there is also an oxide (serving as a dielectric) on the bottom side of the gate that insulates the gate from the semiconductor.

Two important dimensions here are the **length** of the channel (the separation between the source and drain) and the **width** (how "deep" these sheets of metal go).

Because of the doping of the silicon, the n-type source and drain will each form **p-n junctions** with the p-type slab of silicon. Translating our earlier language from the first part of this lecture, when $V_{GS} < V_{TH}$, no current will flow, because there is no conduction between source and drain. But as the voltage rises to a large enough value, a channel will form between the source and the drain, because electrons can flow across the bottom side of the gate. (The shape of the channel and other physical properties come from advanced solid-state physics, but the point is that we'll have a **low-resistance channel**.)

So if we look at this model on a first-order electricity level, we can think of having a **capacitor** from the gate to the ground, as well as a resistor between the drain and source with resistance R_{OFF} when the channel is closed (for nFETs, when $V_{GS} < V_{TH}$) or R_{ON} (for nFETs, when $V_{GS} \geq V_{TH}$) when the channel is open. (Ideally, we have $R_{ON} \ll R_{OFF}$ so the two situations match what we want.)

This isn't a super accurate model, but the point is that we can turn this into an RC circuit to analyze **gate delay**. For instance, if we have a CMOS inverter, and the input voltage $V_{IN}(t)$ is basically a step function from 0 to 1 at time $t = 0$, then the output voltage can be described using an RC circuit, which has solution

$$V_{OUT}(t) = V_{OUT}(0)e^{-t/(RC)}.$$

So now we can understand how to talk about propagation delay for gates: the time t_{PD} is proportional to the constant RC for a CMOS transistor. So to reduce the propagation delay, we need to make sure that our transistors are easy to drive, and that R and C are small.

But if we go back to our transistor model, we can look at how the capacitance of the gate and the resistance of the channel change with our L and W. We know that the capacitance is proportional to the area of the plate LW, while the resistance of the channel (flowing across) is proportional to $\frac{L}{W}$ (because a larger width makes it easier for charge to flow). So the point is that the width doesn't affect the propagation time: CMOS gates will pick the smallest possible length L (lots of ongoing research goes into minimizing this quantity even today), and the width W is chosen for performance. (Wider FETs will drive more current and give us a lower resistance, but then the gates become harder to drive.)

And a final thing to keep in mind is **power dissipation** for our CMOS gates:

Fact 70

Dissipation comes in two parts: **dynamic power**, caused by transitions in the circuit and the charging and discharging of capacitors, and **static power**, coming from the small current due to a finite resistance $R_{OFF} < \infty$, as well as an electron tunneling between the gate and channel.

This is a very difficult analysis to do, but the thing to keep in mind for CMOS gates is that **they are extremely good at having low power consumption**, which is why they're used today.

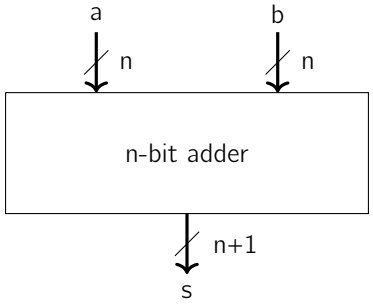
9 September 24, 2020

We'll continue looking at digital design today – in the past few lectures, we've been understanding how to build combinational circuits (which form the basic building blocks of digital logic). But things like truth tables and Boolean expressions help us specify only small combinational circuits – they don't really make it feasible to build arithmetic circuits like 32-bit adders, for example. So we'll go into the design aspect of combinational logic today, using the three examples of adders, multiplexers, and shifters. And we'll start looking at the **Minispec hardware description language**, which is specialized for describing hardware. We can then design combinational circuits using functions, and we'll see that the same principles we've been learning from programming will still apply.

Example 71

Let's start by trying to build a combinational adder – that is, given two n-bit inputs a and b, we want to produce an $(n+1)$-bit output $s = a + b$.

This circuit feeds in the input bits $a_0, \cdots, a_{n-1}$ from a, as well as the input bits $b_0, \cdots, b_{n-1}$ from b, and it outputs the bits $s_0, \cdots, s_n$. In order to avoid needing to draw lots and lots of wires in and out of the combinational device, we can use a shorthand notation, as shown below:



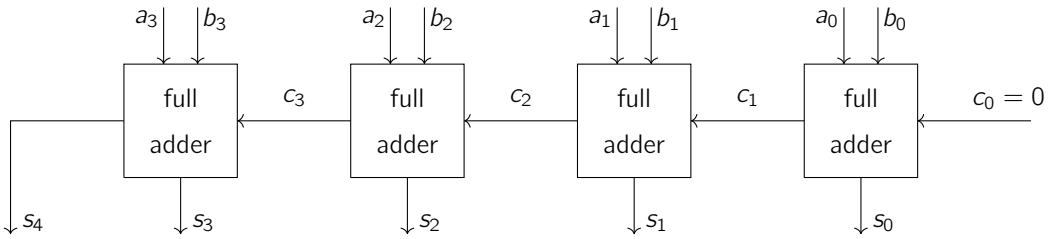
(Remember that n must be fixed in any particular circuit, though – it's not an input.) The way we'll implement this is by using the **standard addition algorithm** we use normally: start adding from the least significant bits and carry a 1 to the next column if necessary. This is better than starting from the truth table, since such a table for a 32-bit adder would give us 2^{64} different rows (which is definitely impossible in practice).

Instead of illustrating the addition algorithm by example, we'll take a more symbolic approach this time. Say that a has binary representation $a_3a_2a_1a_0$ and b has binary representation $b_3b_2b_1b_0$, and label our output s with representation $s_4s_3s_2s_1s_0$. We'll **also keep track of our carries** as $c_4c_3c_2c_10$ (the last bit is always 0, since the ones digit never has any carries).

$$\begin{array}{cccccc} & c_4 & c_3 & c_2 & c_1 & 0 \\ & & a_3 & a_2 & a_1 & a_0 \\ + & & b_3 & b_2 & b_1 & b_0 \\ \hline & s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

On the ith step of our addition (that is, the ith column from the right, where we're zero-indexing), we look at the three bits a_i, b_i, c_i in the ith column. We then then output **two 1-bit outputs**: s_i and c_{i+1}, such that $c_{i+1}s_i$ gives us the 2-bit binary sum of our three inputs. (c_i is called the **carry-in** bit, and c_{i+1} is the **carry-out** bit.) Conveniently, the possible values for the sum of the three inputs a_i, b_i, c_i is always between 0 and 3, which is also the range of possible 2-bit binary values.

So now if we want to build a circuit that performs **one step** of this adding algorithm, we can derive the truth table, write out the Boolean expressions, and build a gate. This is much more manageable than the n-bit adder: we can build this small circuit, which we call a **full adder**, and then we can combine those full adders together sequentially, cascading them to perform a full binary addition of larger numbers:



This is called a **ripple-carry adder** – it's simple and it's the cheapest adder in terms of area, but it's **slow** because of the series connections, and thus the propagation delay adds up across the n full adders. We'll see how to optimize this (to scale as $\log n$ instead of n) soon, but for now let's go back to the (one-bit) full adder and understand it more clearly.

If we use a truth table to build our circuit, there are 8 possible inputs for a, b, c_{in}, and we can write out the value of c_{out} and s in each of those cases. And then we can use the techniques we've seen before to get a minimal sum-of-products, but this circuit turns out to have some nice properties, so we'll look at this more carefully.

Notice that the digit s depends on the number of 1s in the input: it's 1 if the number of input 1s is odd, and it's 0 if the number of input 1s is even. In other words,

$$s = \text{parity}(a, b, c_{in}) = a \oplus b \oplus c_{in},$$

where $\oplus$ is the XOR function (indeed flipping the parity of the output bit whenever we have a 1). On the other hand, c_{out} is 1 when at least two of the inputs are 1, and this is also a famous function:

$$c_{out} = \text{majority}(a, b, c_{in}) = ab + ac_{in} + bc_{in}.$$

We now theoretically have everything we need to build an adder, but describing the circuit with the existing set of tools we have isn't going to work so well. For instance could write 32 different sets of Boolean equations, where each set of equations corresponds to a single full adder, and relate the whole system together with recurrences, but that's not very clean. Truth tables and circuit diagrams are also not optimal, because they are complex and it's easy to make mistakes. So industry uses something different:

Definition 72

A **hardware description language (HDL)** is a specialized programming language used to describe hardware.

Because the goals are different in software and hardware, the features of this language may not do what we want for software, but certain features will be nice: we can specify structure and behavior of circuits easily, simulate designs or synthesize them to hardware automatically, and we still care about principles like composition of building blocks and reusable code. And often, HDLs use a familiar syntax (with functions and variables and control-flow statements), which makes it feel more familiar. But the actual implementations of things like control-flow are **completely different from software**, and it's important to be aware of the relevant differences.

And this is where **Minispec** comes in: we're using this simpler HDL, instead of industry-standard HDLs, so that it's less complicated and can be learned during this semester. (For instance, SystemVerilog has about 1300 pages of reference guides, while Minispec has about 20 pages.)

Fact 73

Minispec actually internally compiles to Bluespec, which was a language specially developed for 6.004, but we won't see the internal compiling – that's been abstracted out of this class.

So we'll get started now! Minispec computes combinational circuits using functions. For example, the following function inverts a single bit:

```
function Bool inv(Bool x);
   Bool result = !x;
   return result;
endfunction
```

We can log in to https://6004hub.csail.mit.edu to test out code ourselves – it's basically like a Jupyter notebook. We'll notice that there are some noticeable differences between Python and Minispec – we have to specify the types of our input variables (because Python is **dynamically-typed**, while Minispec is **statically-typed**), and we have some different keywords and other syntax differences, like required semicolons.

Fact 74

Running the command `synth inv -V` actually **synthesizes a circuit of the function `inv` for us**. (The `-V` flag will even give us a picture of the circuit!)

One key concept to keep in mind is that **type names start with an uppercase letter**, while functions and variables start with a lowercase one. We can introduce a few simple types now:

Definition 75

An object of the **Bool** type takes on a value of either True or False, and Minispec supports Boolean and comparison operators between Bools. In other words, given two Bools `a` and `b`, we can output `!a` (negation), `a && b` (and), `a || b` (or), `a != b` (XOR), and `a == b` (XNOR).

Working with single-bit values can be very tedious, so we also define a more sophisticated type:

Definition 76

Objects of the **Bit#(n)** type take on n-bit binary values. This type supports the bitwise logical operators $\sim$ (negation), `&` (AND), `|` (OR), and `^` (XOR). We can also take in a Bit#(n) object, `a`, and slice it to obtain a Bit#(m) object `a[x:y]`. Finally, we can concatenate two Bit#(n) objects with the command `{a, b}`.

In Minispec, we can write a 4-bit binary value like `a = 4'b0011`, and we index the least significant bits to be smallest. (For example, `a[3:1] = 3'b001`.) And now we're ready to write out our (single-bit) full adder in Minispec:

```
function Bit#(2) fullAdder(Bit#(1)a, Bit#(1)b, Bit#(1) cin);
   Bit#(1) s = a ^ b ^ cin;
   Bit#(1) cout = (a & b) | (a & cin) | (b & cin);
   return {cout, s};
```

```
endfunction
```

To construct a ripple-carry adder, we want to do a cascading call of full adders. For example, here is the code of a 2-bit ripple-carry adder:

```
function Bit#(3) rca2(Bit#(2)a, Bit#(2)b, Bit#(2) cin);
   Bit#(2) lower = fullAdder(a[0], b[0], cin);
   Bit#(2) upper = fullAdder(a[1], b[1], lower[1];
   return {upper, lower[0]};
endfunction
```

One important consideration to note is how circuits are synthesized: calling a function, like fullAdder, creates **new instances of the called circuits**! This means that we create additional copies of our existing circuit, so that the fullAdder circuits for the two different bits can be composed together. (In other words, functions are **inlined**.)

And then we can build a 4-bit ripple carry adder recursively, by calling the `rca2` function twice. We can then create 8-bit and 16-bit and 32−bit adders as we wish, but we may notice that this recursive definition is very tedious. So we'll learn in the next lecture how to write **parametric functions**, which will allow us to write n-bit adders for any n.

Example 77

We'll now jump into **multiplexers**. A 2-bit multiplexer (also called a 2-bit **mux**) allows us to select between two inputs a and b based on an input bit s (called the **select input**).

The boolean expression for the output of a multiplexer is $a\overline{s} + bs$ when a and b are each only a single bit, and we already know how to do a gate-level implementation of that. And if we have n-bit inputs a and b, we can select between them by replicating this structure n different times, using the same value of s each time.

If we wish to build a 4-way multiplexer, which selects between four inputs based on a 2-bit input s, we can build everything in a **tree structure**. Specifically, if we are choosing between inputs a, b, c, d depending on the select input being one of $s = 00, 01, 10, 11$, then we can first use a 2-way mux for a and b (feeding in s_0 as the select bit), and do the same for c and d, and then finish by using a 2-way mux with select bit s_1 to get our final output. In general, an n-way multiplexer can be implemented with a tree of $(n-1)$ 2-way multiplexers, in the manner just described.

A two-way multiplexer is written in Minispec as the **conditional operator** `s? b: a`, where `s` is a Bool object – this evaluates s and returns b if True and a otherwise. (This is the equivalent of the Python statement `b if s else a`.) And if we have an N-way mux, we can use a case expression as written below:

```
case (s);
   0:  a;
   1:  b;
   2:  c;
   3:  d;
endcase
```

Notice that s is no longer required to be a Bool with the `case` syntax – we're just selecting from possible values for it.

Fact 78

Even though we have conditional statements, it's important to remember that **this doesn't mean our hardware is doing conditional execution**. In particular, a statement like `s? foo(x) : bar(y)` in software would have us evaluate `s` first before either evaluating `foo(x)` or `bar(y)` (to avoid running expensive functions), but that's not an option in hardware. Instead, we'll have to instantiate and evaluate `foo(x)` and `bar(y)` in parallel.

When we start looking at sequential logic, we'll see that we can do conditional execution, but it's not possible in combinational logic. So we're starting to see how there are some limitations that don't exist in general-purpose programming.

But muxes have some other uses for implementing Bluespec commands as well. For example, suppose that we feed in a 4-bit binary string `x`, and we want to select some bit from it. If we're trying to find something like `x[2]`, this is a **constant selector**: we can implement it by just connecting the corresponding wire to the output. But we may want to do a **dynamic selector** instead, where we feed in a binary value `i` and output `x[i]`, and that's where we can use a mux to implement the hardware.

We'll finish by introducing shifts and how to efficiently implement them. A logical right-shift or left-shift by a fixed number of bits is cheap to do in hardware, because we just need to wire our circuit appropriately (for instance, connect the 0th bit of the input to the 1st bit of the output, and so on). And arithmetic right shifts only require us to replicate the topmost bit, so that gives us something similar.

But suppose we want to do a logical shift of an N-bit input by a **variable** number of bits s (where $N = 32$ and s is between 0 and 31). One solution is that we could create 32 different fixed-size shifters and use a mux to switch between them, but that's expensive to implement. So next time, we'll see how to do better!

10 September 29, 2020

Today will be the last lecture on combinational logic: we'll cover some techniques for writing efficient code, and we'll look more deeply at how to analyze design tradeoffs in combinational circuits. In particular, we'll learn about some advanced features of Minispec, like parametric functions.

Logistically, we have a quiz on Thursday (which won't test today's material), so there will also be no lecture on that day. There will also be a **quiz review session** tonight – the Zoom link is posted on the website.

First of all, we'll continue the discussion of shifts from last week. Recall that if we want to feed in an input x and shift it to the left or right by a fixed number of bits, that's cheap to do in hardware (because the wires just need to be connected in the correct way). On the other hand, suppose we want to build a **shifter circuit**, which shifts an N-bit input by s bits (for instance, where $N = 32$, and s can be any variable amount between 0 and 31). Then, as we discussed last time, it's not very efficient to build 32 different shifters (one for each value of s) and then use a 32-way mux. That's because this implementation requires a tree of $31 \cdot 32 = 992$ different one-bit muxes (we make a tree of 31 muxes to choose between the different shifters, and each of those muxes is really 32 bitwise muxes). This requires $N(N-1)$ shifts for an N-bit input, and we want to do better.

Proposition 79

A **barrel shifter** is an efficient circuit for variable-size shifts. Implementation-wise, we only shift using powers of 2: for example, if we want to shift our number by 5, we can shift by $4+1$, and if we want to shift by 12, we can shift by $8+4$.

This is easy to implement because we can put less shifters in our circuit. To figure out which ones to use, **we just look at the bit encoding** of s: if its ith bit is a 1, then we need to shift N by 2^i. And this kind of implementation only requires $\log_2 N$ large muxes: for each of the bits in our binary number N, we choose between shifting and not shifting our input. So overall, this circuit uses $(N \log_2 N)$ one-way muxes instead of $(N^2 - N)$ muxes, which is indeed an improvement.

Example 80

Let's try writing a Minispec implementation for the barrel shifter for $N = 4$.

Because we can shift anywhere from 0 to 3 spots, we only need two bits for s, corresponding to a shift by 2 and a shift by 1, and we use a conditional operator for each one. And bit selection and concatenation help us do shifts easily in Minispec, as shown below:

```
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);
   Bit#(4) r1 = (s[1] == 0) ?  x:  {2'b00, x[3:2]}; (either keep x or shift it to the right)
   Bit#(4) r0 = (s[0] == 0) ?  x:  {1'b0, r1[3:1]};
   return r0;
endfunction
```

We've already seen that differences in implementation have a large impact on the area and characteristics of our circuits, and what we'll do now is to show a few more advanced features of Minispec so that we can improve our implementations.

So far, if we want to build large circuits, we've often been composing them from smaller ones by calling other functions. But remember that it was tedious to build an n-bit adder by doing this repeated calling: to define an adder `rca8`, we had to first define a full adder, then define `rca2` by taking two full adders and concatenating them together, then define `rca4`, and then finally define the actual function we care about. And we can notice that the definitions follow the same structure, just with different bit indices, so we want to **cut down the tedious work** we have to do. The idea is that we'll make the compiler do the hard work of expanding out our circuits by using **parametric functions**.

Fact 81

The `Bit#(n)` type is actually a parametric type: `n` is the **parameter**, an integer that we feed in, so in order to actually use a `Bit#(n)` type, we need to specify `n`.

Minispec also has other parametric types, and we can define our own as well. Here are the main features:

- Parametric types are really a family of concrete types, which makes them **generic**.
- They take in one or more parameters, whose values must be **known at compile time**. Parameters can be integers or other types – for example, `Vector#(n, T)` is an `n`-element vector of `T`s, so something like `Vector#(4, Bit#(8))` is a 4-element vector of 8-bit values.
- Specifying those parameters yields a concrete type.

But what we really care about are our **parametric functions**, which help us get around the issues of having to use fixed argument and return types. The goal is to write something like `rca#(n)`, which works as an `n`-bit ripple-carry adder, as long as we feed in some integer `n`.

Remark 82. *Parametric functions are present in basically every statically-typed language (they are also called "templates" or "generics"). The fundamental idea is that we're distributing some of the work to compile time instead of runtime.*

Example 83

Consider the n-bit parametric parity function below, which tells us if the input has an even or odd number of 1s.

```
function Bit#(1) parity#(Integer n)(Bit#(n) x);
   return(n == 1)?  x :  x[n-1] ^ parity#(n-1)(x[n-2:0]);
endfunction
```

Basically, in the case where `n` is 1, we want to return `x` itself; otherwise, we take the most significant bit of `x` and `XOR` it with the parity of the remaining bits of `x`. So there is a recursion on the **parameter** n (not on the argument x): in order to define `parity#(n)`, we need to define `parity#(n-1)`.

This means that we have some "compile-time recursion:" if another function calls `parity#(3)`, the compiler will substitute in 3 for the value of `n` in our circuit above, and now `n` no longer explicitly appears in the code. But now we have `parity#(2)` being called, so the compiler must instantiate another function by substituting in 2 for `n`. It will keep doing this until we're done – it's important to remember that all of this gets synthesized into gates and wires at the end of the day, so we can think of having a `parity#(1)` box, inside of a `parity#(2)` box, inside of a `parity#(3)` box.

Fact 84

To make sure things are done at compile time correctly, we use an **Integer** type, which allows for numbers with an **unbounded number of bits**. These Integers cannot be synthesized directly, so the compiler must evaluate all expressions with Integers (or else throw an error).

Integers do support the same operations (like arithmetic, logical, and comparison operations). But everything is being **evaluated by our compiler**, so these operations never actually generate any hardware.

And now we're ready to look at something a bit more complicated: if we want to look at an n-bit ripple-carry adder `rca#(n)`, we can attach a full adder to an `rca#(n-1)` circuit, as shown below.

```
function Bit#(n+1) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) cin);
   Bit#(n) lower = rca#(n-1)(a[n-2:0], b[n-2:0], cin);
   Bit#(2) upper = fullAdder(a[n-1], b[n-1], lower[n-1]);
   return {upper, lower[n-2:0]};
endfunction
```

(And to avoid infinite recursion, we need to define the base case `rca#(1)` as well, which is just a full adder.) Such recursions can be implemented as chains or as trees, and we will see examples of both in the rest of our work.

It's important to mention a few more features of Minispec that are tricky as well:

- We don't need to specify the type of every variable we define explicitly – compilers can infer variable types if we use the `let` keyword. For example, if `x` is a 4-bit value, then `let y = x;` will instantiate `y` as a 4-bit value, and `let n = 42;` will make `n` an Integer.

- We can declare **type synonyms**, which mean that we can give different names for the same type. For example, `typedef Bit#(8) Byte;` allows us to type `Byte` instead of `Bit#(8)` every time.
- **Structs** represent a group of values, possibly of different types. For example, we can define a Pixel to have a "red," "green," and "blue" component by typing the following code:

```
typedef struct{
   Byte red;
   Byte green;
   Byte blue;
} Pixel;
```

  And from here, commands like `Pixel p;` allow us to define Pixel objects, and commands like `p.red = 255;` allow us to specify specific values.

- **Enums** allow us to represent a set of symbolic constants (that is, choosing between N different values) instead of having to assign a number to each choice. (The compiler then generates the logic by treating these as n-bit values.) Here's an example:

```
typedef enum{
   Ready, Busy, Error
} State;
```

  From here, commands like `State state = Ready;` allow us to instantiate objects.

- Loops and conditional statements allow us to compactly express sequences of similar statements, but **they turn out to be very different from how they work in software** even if the syntax looks a lot like C or some other programming language. That's because **for loops have a fixed number of iterations** – a compiler will actually unroll out for loops at compile time, so we can accidentally run into very inefficient designs if we're not careful. The same thing is true for conditional statements – any if statement becomes a mux.

So the takeaway is that Minispec lets us build circuits with constructs that are similar to that of software programming, but implementation is very different – parametric functions and types are instantiated at compile time, functions are inlined, conditionals and loops are unrolled and written out, and we end up with an acyclic graph of gates at the end of the day. So we really just have combinational loops, and we must never forget that we're designing hardware.

We'll finish with a few thoughts about hardware design and optimizations of implementation (between delay, area, and power). **Ultimately, the main factor that determines the quality of our designs is how we choose between different algorithms**, because tools can't compensate if we give them inefficient algorithms.

Example 85

We've been discussing the ripple-carry adder so far, which is a simple but slow circuit. The issue is that the worst-case path carries us through a full chain of all full adders in a row.

In other words, the propagation delay for an n-bit ripple-carry adder is n times the propagation delay of a full adder, which is asymptotically $\Theta(n)$ (that is, the delay grows linearly with the number of bits).

Definition 86

Let $f(n), g(n)$ be two functions. We say that $\boldsymbol{g(n) = \Theta(f(n))}$ if and only if there exist $C_2 \geq C_1 > 0$ such that

$$C_2 f(n) \geq g(n) \geq C_1 f(n)$$

for all but finitely many integers $n \geq 0$ (equivalently, for n large enough).

Another way to say this is that we care about the asymptotic bounds, so it's fine if our bounds don't hold for small values of n. (If we've seen big-O notation before, that's the first inequality in the definition above.) The point is that we usually make f a simple function to understand large-n behavior: for example, $g(n) = n^2 + 2n + 3 = \Theta(n^2)$ (in words, $g(n)$ is "of order n^2").

So what we can do instead is to use a **carry-select adder**: the idea is to take our long chain and precompute part of our result, independent of how the carries are propagated. For example, instead of implementing a chain of 32 full adders, we compute the value of a 16-bit addition for the upper 16 bits in **two different ways**: both assuming a carry-in of 0 and a carry-in of 1. Then we do a 16-bit addition of the lower 16 bits, and we use the carry bit of that to **implement a mux between the two possibilities above**. This is better because our propagation delay is only

$$t_{PD,32} = t_{PD,16} + t_{PD,MUX},$$

and applying this recursively actually gives us $\boxed{t_{PD,n} = \Theta(\log n)}$. The only issue with this design is that there is a lot of **wasted area and power**: we need to compute multiple versions of the upper half, even if it's done in parallel. And this mux is wide and adds a nontrivial amount of delay as well, so there's actually another adder, called a **carry-lookahead adder**, which we won't discuss in detail here. The idea is to use a separate circuit that **computes all of the carries from the input bits in logarithmic time**, and then we feed this into n individual full-adders. This **transforms our chain of computations into a tree**, but carries aren't associative, so we need to do something more complicated. We can explore this on our own if it's interesting to us!

11 October 6, 2020

Solutions to quiz 1 will be posted later tonight, and our graded exams will also be available on Gradescope later tonight. We should be able to review them at that point, and if there is anything problematic that we find, we can submit a regrade request. Requests will be accepted until next Wednesday at 5pm.

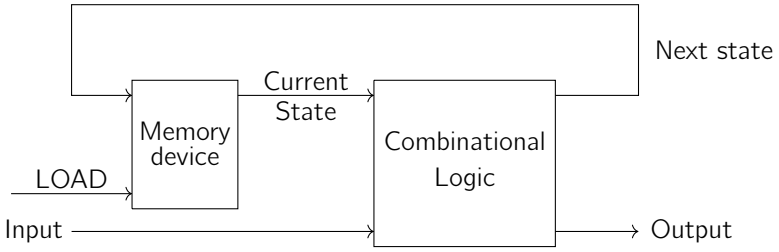
Today, we're switching gears – we've finished our study of combinational circuits, and we'll move on to **sequential circuits** – circuits with states.

Example 87

Suppose we have a design specification, which looks similar to that of our combinational circuits: we have an input button and an output light, and when the button is pushed, we should **toggle** the light on or off from its current state, within a second of the button press.

Structurally, this circuit is similar to the combinational device specifications we've had so far, but this device has a **state** which keeps track of some memory. In particular, **the outputs depend on more than just the inputs** in this circuit. Furthermore, this output is changed by an input event – pushing a button – rather than the level or value of the input. So we need to keep both of those ideas in mind as we consider what sequential circuits are.

The idea is to build a sequential circuit like in the following diagram:

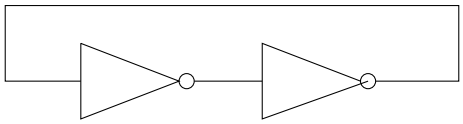


Here, the memory device stores our current state, and the combinational logic device computes two things:

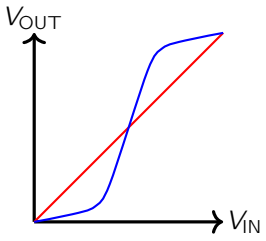
- The next state that we store in the memory device (from both the input and the current state), creating a stream or sequence of memory values,
- and the output bits of the device (also a function of the input and current state).

In addition, we should be able to periodically trigger a "LOAD control input," which edits our current state. So the first thing we need to do is build this loadable memory using the techniques we've already developed.

To do this, we will take an idea from combinational circuits. Remember that we wanted to avoid **feedback**, where inputs fed back into themselves. But consider a circuit of two inverters, where each one feeds into the next:



If we feed in a 0 into the first inverter, we'll maintain a constant value of 1 on the bottom wire between the inverters and 0 on the top wire. Furthermore, logic gates will **restore marginal signal levels**, so our circuit will continuously reset noise and keep feeding in a low voltage into the first inverter. An analogous thing happens when we feed in a 1 into the first inverter – either way, we maintain a constant value in this device, and this gives us what's called a **bistable storage element**, because it can hold a stable 0 or a stable 1. We'll now take a closer look at the voltage transfer characteristic graph: what we end up with looks something like a buffer (blue graph below). There are three places on this graph we can be, since we know that V_{IN} and V_{OUT} need to be the same value:

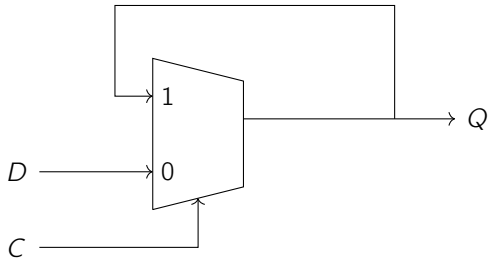


The two endpoints are **stable** here (representing the behavior when we feed in a 0 or a 1), while the middle point is **metastable** – that last point is something we'll return to later in the lecture.

Example 88

The **D Latch** is a famous circuit that can hold a state.

A schematic diagram of the D latch is below – we'll denote this device as a **DL box** in subsequent circuits:



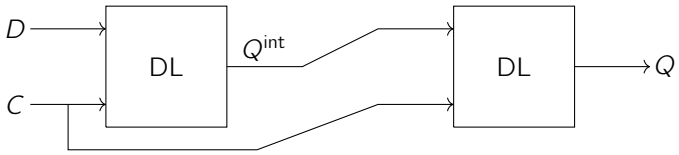
In this circuit, we still have a feedback path that maintains the current state, but we also have a new way to load a value. The multiplexer has an output tied to one of its inputs – if the control signal is 1, then we're using the feedback path where we maintain the old value, so Q holds at its current value. On the other hand, if the control signal C is 0, then we select the input D, and we can propagate the new value.

We can use the following truth table to express the behavior of the D latch, where Q^{t-1} is the previous state held in DL and Q^t is the current value, and X can be either value:

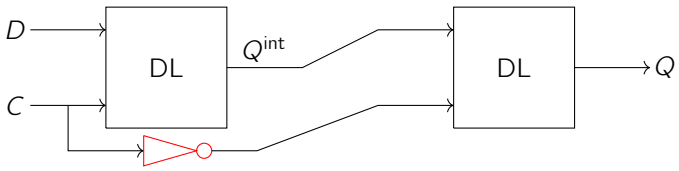
C	D	Q^{t-1}	Q^t
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

Basically, whenever $C = 0$, Q^t will mirror D (this is called a **pass**, but whenever $C = 1$, $Q^t = Q^{t-1}$ (this is called a **hold**). And we're assuming here some condition on how our inputs change, but we'll get to that in a second.

This device then allows us to hold state and store new values into it, and we'll use this in a larger circuit now. Suppose that we tie two DL latches back-to-back as shown:



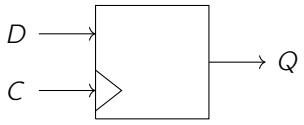
Here, we're using the same control signal C for both latches, so they'll both pass at the same time and also hold at the same time. So having two of these in series doesn't do anything for us. But we can also combine these two D latches in a different way: **for one of the D latches, we feed in the negative of the control signal**, and we can do this by **adding an inverter to the bottom C path.**



And now something interesting does happen – for whatever value of C we have, one of the D latches will pass and the other will hold. When $C = 0$, the intermediate value Q^{int} will follow our input D, but the output Q isn't changing. On the other hand, when $C = 1$, Q will follow Q^{int} but Q^{int} doesn't change.

So Q **only changes when our control signal transitions from 0 to 1** – this is called a **rising-edge** for the input C. (When C goes from 1 to 0, called a **falling-edge**, we'll accept a new value of Q^{int}, but it won't affect Q until we get C to go back to 1.)

Now that we have the two D latches working together in an interesting way, we'll call this an **edge-triggered D flip flop**, and we'll draw it like this in all diagrams from here on out:



The way to think about this is that our fed-in signal C will change periodically – it's called a **clock signal** – and half of the time, C will be a high (1) bit, and the other half of the time, it will be a low (0) bit. Now we need to think carefully about how our D signal is changing – we sample Q at the rising edge of the clock, so we look at the value of D every time when C is going from 0 to 1 and sample the value of the data D. But there's a problematic case – if our data is not stable at the rising edge of the clock, then we have a problem, which is what is called **metastability**. In order for sequential circuits to work correctly, **we need to avoid this possibility** – we must make sure input values are stable and valid for some time around that rising edge.

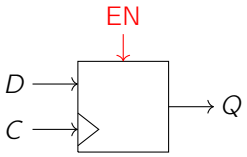
Definition 89

Let t_{SETUP} be the time required before the rising edge of the clock, and t_{HOLD} be the time required after that rising edge, that we require our flip-flop input D to be stable.

Then we measure the propagation delay t_{PD} of our circuit by looking at how much time after the rising edge it takes for us to settle into a valid output Q.

Example 90

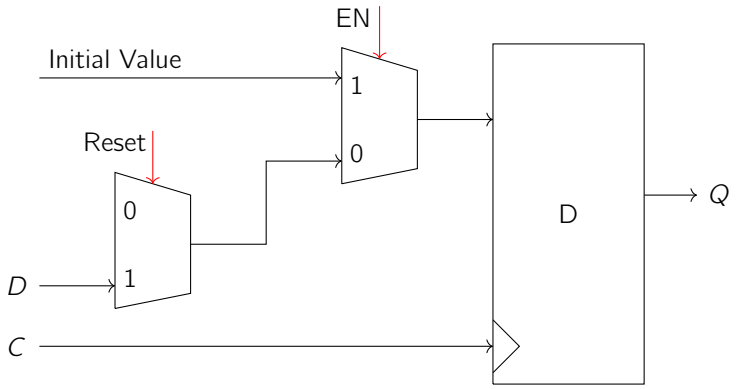
Let's now construct an alternate version of this D flip flop to give us more control, as in the schematic below. This is called a **D flip flop with enable.**



We've added one more input here, called an **enable signal**, and the idea is that we only capture new data D if the enable bit EN is equal to 1. We implement this by **putting a multiplexer in front of our D flip flop** – the circuit says that if the EN bit is 0, we keep our old state, and otherwise, we accept a new value of D to propagate over to Q. So when the enable bit is 0, we hold, and when the enable bit is 1, we copy the input.

Remark 91. *Moving forward, we won't show our C clock input explicitly as part of the truth input, because we know that transitions of state only happen at the rising edge of the clock $0 \to 1$.*

And we need to be able to do one more thing – initialize this device with some starting value. So we'll now create a D flip flop which has both an enable and a reset signal, and we can do that with another multiplexer. The entire schematic is shown below:



We now have a loadable D flip flop which is capable of remembering its state, loaded with new values, and initalized with certain values. The idea is that whenever Reset is 1, we grab the initial value regardless of the value of EN. But when Reset is 0, we behave like a D flip flop with enable.

This device can now be used to think about sequential circuits in general – in 6.004, we'll be looking at **synchronous sequential circuits**, which are circuits where all flip-flops (also called **registers**) **share the same periodic clock signal**, so all registers are updated simultaneously. And now this allows us to abstract our sequential circuits into by discretizing time into repeated cycles:

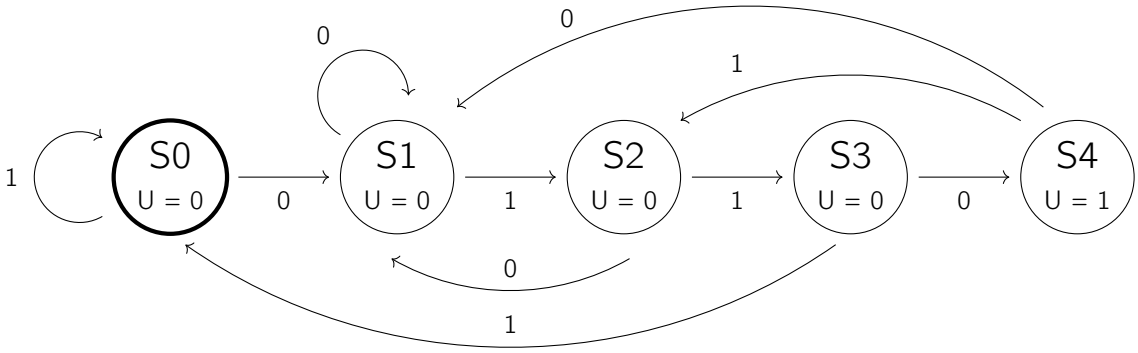
Definition 92

A **finite state machine (FSM)** stores one of finitely many possible states and switches between those states based on an input string.

Example 93

Suppose we want to create a **combination lock**, which is a sequential device with one input bit and one output signal called UNLOCK. Our device should have UNLOCK = 1 if and only if the last four inputs form the combination 0110.

The first thing we'll do is represent the behavior of this circuit using a **finite-state machine diagram**. We need five different allowed states here: we have **S0**, the initial state of our lock, and then we need a state for each of the situations **S1, S2, S3, S4** where we've entered 0, 01, 011, or 0110.



We give each state a name (**S0** through **S4**), and each state has an associated output. (In this case, only being in **S4** will allow us to unlock the lock.) We put a heavy circle around the initial state to indicate that it is where we

start, and then we have transitions that tell us how we move when we read in a 0 or a 1. The U label at each state tells us what we output if we end up at this state (in this case, either a 0 or a 1).

This formalism now lets us properly describe the combinational lock: we can confirm that the FSM keeps track of "how close" we are to getting to the output, and we need to go backwards when we get an incorrect input (and break the pattern). And when we're in a particular state, the input tells us which state to go to next, and this lets us formulate the following two conditions:

- All arcs leaving a state must be **mutually exclusive** (so we can't have two choices for a given input).
- Arcs must be **collectively exhaustive** – we must know what to do for every possible input, and if we aren't supposed to do anything, we draw an arc back to the state itself.

Example 94

Let's try to design a sequential circuit at the hardware level for a **modulo-4 counter**. This circuit takes in an input bit, which is either an increment of 0 or 1, and updates by adding (mod 4) the increment to our current value.

This time, we have four states: each of our states represents a value mod 4, representing the current mod 4 value. If we were to draw a state transition diagram, we'd have four states: **00, 01, 10, 11**, corresponding to the values 0, 1, 2, 3 mod 4. Then increment bits of 0 correspond to arcs from each state back to itself, while increment bits of 1 give us arrows from 00 to 01, from 01 to 10, from 10 to 11, and from 11 to 00.

This data can be represented in truth table form as well: given our current state `q1q0` and our increment bit `inc`, we can describe what the new value of `q1q0` is. Either way, now we can write a Boolean expression that relates these variables: for the least significant bit of our counter, we can say that

$$\mathtt{q0}^{t+1} = \sim\mathtt{inc}\cdot\mathtt{q0}^t + \mathtt{inc}\cdot\sim\mathtt{q0}^t = \mathtt{inc}\oplus\mathtt{q0}^t.$$

Similarly, we can write that

$$\mathtt{q1}^{t+1} = \sim\mathtt{inc}\cdot\mathtt{q1}^t + \mathtt{inc}\cdot\sim\mathtt{q1}^t\cdot\mathtt{q0}^t + \mathtt{inc}\cdot\mathtt{q1}^t\cdot\sim\mathtt{q0}^t = \sim\mathtt{inc}\cdot\mathtt{q1}^t + \mathtt{inc}\cdot(\mathtt{q1}^t\oplus\mathtt{q0}^t).$$

So now let's try to build a circuit. Because we need to store two bits in storage, we're going to want to use two D flip flops, one for `q0` and one for `q1`, and we also want to equip each of these two D flip flops with reset and enable bits. And we can simplify this situation by only having things run through the circuit when `inc` is 1, so we can connect `inc` to the enable bit! That simplifies our actual Boolean expressions to

$$\boxed{\mathtt{q0}^{t+1} = \sim\mathtt{q0}^t, \quad \mathtt{q1}^{t+1} = \mathtt{q1}^t\oplus\mathtt{q0}^t}.$$

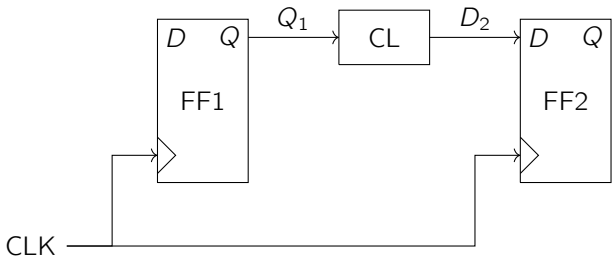
So each time we want to increment this machine's state by 1, the EN bit will turn to 1, and our machine will invert the `q0` bit, while simultaneously XORing the current two stored bits to get the `q1` bit. Setting the initial state to **00** (using the **reset** signal) finishes the construction.

Fact 95

We'll see soon that Minispec allows us to describe sequential circuits in a simpler way next lecture, so we won't need to worry about this manual method going forward. But it's still good for us to understand how to create sequential circuits!

To close, we'll talk more about the timing of sequential circuits – we mentioned earlier that we need to avoid having unstable inputs for at least t_{SETUP} after the rising edge of the clock, as well as t_{HOLD} afterward. To ensure that these conditions are satisfied, we need to think about the (single) clock that all of our registers are controlled by. Any circuit we have has a bunch of combinational logic between registers, so we need to consider **all possible register-to-register paths** in our sequential circuit, making sure that any register input satisfies the setup and hold time requirements.

What we're saying is to consider a diagram like the one below, where two registers (denoted with FF1 and FF2) are connected by the combinational device CL:



Then we need to make sure that the input D_2 is stable at least t_{SETUP} before the next rising edge of the clock, but before that can happen, we need our bit D to propagate through our first flip flop FF1, which then goes through the combinational logic. So we need at least $t_{\text{PD, FF1}}$ from the first flip flop, plus some time $t_{\text{PD, CL}}$ to turn a stable Q1 into a stable D2, to all happen early enough. In other words, we need the inequality

$$t_{\text{CLK}} \geq t_{\text{PD, FF1}} + t_{\text{PD, CL}} + t_{\text{SETUP, FF2}}$$

for **every single** register-to-register path.

And finally, we have a **hold time** constraint as well: we need to make sure that D2 doesn't change too quickly after the rising edge of the clock, so we need some constraint on the previous value of D2 being contaminated by the next propagation of logic in our circuit. We know that t_{PD} is an upper bound on how long our t_{HOLD} time is, and now we finally introduce the idea from a few classes ago:

Definition 96

The **contamination delay** t_{CD} is a lower bound on how long it takes from an input transition to an output transition in a device.

And the second constraint we have is that

$$t_{\text{CD, FF1}} + t_{\text{CD, CL}} \geq t_{\text{HOLD, FF2}} :$$

it must take enough time for a new D to propagate through to D2, so that we don't get a contaminated input.

12 October 8, 2020

We'll continue our exploration of sequential circuits today – last lecture, we explained the basics of sequential logic, as well as some methods for implementing small sequential circuits. There's a parallel we can draw here with combinational logic – back then, we looked at the basics first with things like truth tables and Boolean expressions, which work well for specifying basic circuits, but implementation with those basic methods is much more difficult at scale.

So we're going to look at the flip side of sequential logic, learning how to design more interesting circuits using Minispec. We'll explore the concept of **modules**, which solve a problem of finite state machines (namely, that they don't compose cleanly). We'll also be able to explore some advantages of sequential logic's power over combinational logic – essentially, we can perform computation over multiple cycles, and we can do computations with variable amounts of input, output, and steps.

First, we'll do a quick summary of last lecture's material. Recall that sequential circuits use a **D flip flop (DFF) state element**, which sample the data input D at the rising edge of the clock and propagate it to the output Q. Flip flops often need to be enhanced to be useful, though – we use a **reset circuit** which allows us to set the initial value reliably, and we also use a **write-enable** circuit so that we can retain our current value unless that enable bit is explicitly toggled to 1. This allows us to set and update our circuit's initial state as we wish!

Definition 97

A **register** is a group of DFFs which stores multi-bit values.

These components can then be wired up with combinational logic in various ways, but it's easy to make the circuits difficult to analyze. We want to avoid the difficulties of **asynchronous logic**, so we're sticking to **synchronous sequential circuits** in this class instead: that means that all of our registers sample values of inputs at the same physical time, driven by the same clock, so everything is discretized into cycles. And the reasoning of the circuits becomes easier too, because we can abstract everything as a finite state machine – we update our combined state only at the rising edge of the clock. Such FSMs can be described using truth tables or state-transition diagrams, but when we want to build large circuits, we need to be more careful.

In particular, back when we only had combinational logic, combining circuits was easy – that was just function composition. But **wiring up two FSMs can introduce combinational cycles**.

Example 98

It's possible that when we put a sub-FSM inside of another FSM, we end up with a loop without going through any registers.

Here's an example of some bad code:

```
fsm Inner;
   Reg r;
   out = in ^ r.q;
   ...
```

```
fsm Outer;
   Inner s;
   s.in = !s.out;
   ...
```

Then we can notice the combinational cycle formed by the XOR and NOT gates, possibly shorting out the whole system. This might seem very artificial, but larger blobs of combinational logic are hard to check. And it turns out that most hardware description languages allow us to wire FSMs however we want, and often that means we can end up with undefined behavior.

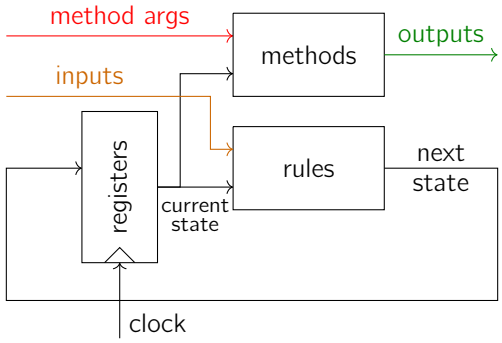
Remark 99. *We can read "Verilog is weird" (Dan Luu, 2013) for insights on how confusing this can be.*

Bluespec is nice for this particular reason – there's a lot of magic going on that allows us to combine FSMs together. But trying to explain the details of Bluespec in 6.004 takes a lot of time, so we're using Minispec instead: we're going to introduce the **minimum amount of added structure to FSMs to make them composable**.

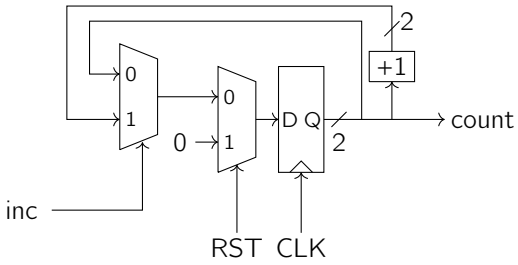
Definition 100

Minispec **modules** separate our combinational logic into **methods** and **rules**. Rules compute our next state, while methods give us outputs, but the two do not share any inputs.

We can compare this diagram with the schematic from last lecture: notice that the "methods" and "rules" boxes take the place of the "combinational circuit."



It may not be clear yet why this leads to composability, but we'll focus on the basic division of logic and look into how the syntax works for now. To do that, let's return to the **two-bit counter** from last class, which stores an incrementing value mod 4. Physically, the implementation looks something like this:



First, we use the reset RST component to initialize our state to 0. Then every time we want to increment our counter, setting inc to 1 allows the circuit to propagate forward that cycle using the wire with the "+1" instead of the wire without it. Here's what this implementation looks like in Minispec:

```
module TwoBitCounter;
   Reg#(Bit#(2)) count(0);
   method Bit#(2) getCount = count;
   input Bool inc;

   rule increment;
      if (inc)
         count <= count + 1;
   endrule
endmodule
```

Let's break down the code here a bit. The `Reg` type is a parametric module that stores (in this case) Bit#(2) values, and we're saying that we call it `count` and initialize its value to 0. We also have a `getCount` method, which

returns the current value of `count`. And the input bit `inc` is a `Bool` here that tell us whether or not we want to increment during a cycle.

Rules like `increment` fire every cycle, and they derive the next state with combinational logic. And we're using similar syntax to functions here: the increment function updates our count to `count + 1` if the `inc` input is `True`.

Definition 101
The **Reg#(T)** module is a register of values of type T.

So we can have a `Reg#(Bool)` or a `Reg#(Bit#(16))`, but we should be careful not to say something like `Reg#(16)`. Notice that we used the special register assignment counter `count <= count + 1` above instead of `count = count + 1`, because the latter is a **combinational construct**, while the former tells us to put the result into the D input. The rule is that **registers can only be written to at most once per cycle**, and we only update our registers with `<=` at the end of each cycle.

Fact 102
One nice feature is that we can swap the values in two registers `x` and `y` using the command `x <= y; y <= x;`, because neither register's value gets updated immediately. We don't need to use any temporary variables!

And now we can compose modules together: suppose that we wanted to build a FourBitCounter using two TwoBitCounters. It makes sense to keep track of both the lower and upper bits, and this means we now have **submodules** within a given moudle. But unlike classes or objects from object-oriented programming, these modules have rules of their own. Our code may look something like this:

```
module FourBitCounter;
   TwoBitCounter lower;
   TwoBitCounter upper;

   method Bit#(4) getCount = {upper.getCount, lower.getCount};

   input Bool inc; (tracks the increment for both TwoBitCounters)

   rule increment;
      lower.inc = inc;
      upper.inc = inc && (lower.getCount == 3); (because the lower counter may wrap around)
   endrule
endmodule
```

One thing we should notice is that the `increment` rule **sets the inputs** of our lower and upper submodules. And now we have a more general picture of how a module interacts with its submodules: our module's methods and rules can pass arguments and inputs to its submodules, and the submodules can pass their outputs to our module's rules and methods. And in general, we can describe the four components of our modules as follows:

1. **Submodules** (registers or other modules that we define) allow us to compose modules together (by linking inputs and outputs).

2. **Methods** take in arguments and our module's current state, and they produce outputs.
3. **Rules** take our current state and some external inputs, and they produce our module's next state, as well as inputs into the submodules.
4. **Inputs** are external values from the enclosing ("parent") module.

Fact 103

We'll use **strict hierarchical composition** in 6.004, which means that **(1)** modules only interact with their own submodules, and **(2)** methods do not read inputs (only rules do).

This guarantees that we won't have combinational cycles, and it also gives us some simple semantics to work with. Although our circuits may look like they have a lot of parallel logic, the whole system really behaves in an "outside-in" manner, because the rules can **fire sequentially** from the outermost module into their submodules. (Minispec also supports non-hierarchical composition, but this is outside the scope of 6.004.)

And now we can get into design practices, starting with how to **test or simulate a module.** In hardware design, we often test individual modules independently, and we can systematically test using **testbenches**. Testbenches are basically other modules, which have our (to-be-tested) module as a submodule, and they check that our outputs are correct through a sequence of test inputs. Here's an example of a test for our FourBitCounter above:

```
moduleFourBitCounterTest;
   FourBitCounter counter;
   Reg#(Bit#(6)) cycle(0);
   rule test;
      counter.inc = (cycle[0] == 1); (we only increment on odd cycles)
      $display("[cycle %d] getCount= %d", cycle, counter.getCount);
      cycle <= cycle + 1;
      if(cycle >= 32) $finish; (terminate after 32 cycles)
   endrule
endmodule
```

Commands with a $ at the front are called **system functions**, and they are ignored when we synthesize hardware. Their purpose is basically to output results and control the simulation process during testing.

We'll finish this lecture by explaining the advantages of sequential logic over combinational logic – one important concept is that we can do computation **over multiple cycles.** We say that "time is more flexible than space" – for example, we can build a circuit that adds two numbers of arbitrary length by adding one digit per cycle (we couldn't have done this with combinational circuits, which can only implement n-bit adders for a fixed n). All we need to do is **connect a flip flop to a single full adder**, feeding in the output carry bit `cout` back in as the input carry bit `cin`. And if we want to sum longer numbers, that just means we need to wait for more cycles to finish!

Remark 104. *Notice that this means we only need one full adder, though it will take more time than the combinational circuit we described last week. So there's more of an area versus time tradeoff going on here as well.*

As a final example, suppose we want to compute the greatest common divisor of two numbers using the Euclidean algorithm. The way this works is to repeatedly subtract the smaller number from the larger number until one of them reaches 0, at which point the nonzero number will be the GCD of our original two numbers. This algorithm can take

a variable **number of cycles** to complete, so one thing we can do is keep track of an `isDone` Boolean value (letting us know whether the process has finished). Then we feed in the next cycle's circuit inputs, based on the logic of our previous iteration. The point is that **conditional statements correspond to muxes**, and the algorithm is simple enough that we can output results easily, too. Here's the full code written out (first read down the left, then the right):

```
typedef Bit#(32) Word;
module GCD;
   Reg#(Word) x(1);
   Reg#(Word) y(0);
   input Bool start;
   input Word a;
   input Word b;
   rule gcd;
      if(start) begin
         x <= a; y <= b;
      end else if (x != 0) begin
         if(x >= y) begin (in this case subtract)
            x <= x - y;
         end else begin (in this case swap)
            x <= y; y <= x;
         end
      end
   endrule

   method Word result = y;
   method Bool isDone= (x == 0);
endmodule;
```

This module can start a GCD computation whenever we set the start input to `True` and pass in arguments through `a` and `b`. Then after some number of cycles, the module will finish the `isDone` method, and we can see the result at that point by looking at the result method.

But this actually has a poor interface, and that's because it's easy to misuse – for example, we might set a and b and not start, or we might forget that there's an isDone check we need to do. And this is tedious, because we need to specify all the inputs every cycle even if start is False. So we'll **group inputs and outputs that are related** if we want to design things well. It would be better in this case if we only have one output (either an invalid or valid result), and we may want to condense into a single input that either gives us all the arguments or none of them. So we'll need a way to encode **optional types**, and this comes through the `Maybe` type.

Definition 105

An object of the **Maybe#(T)** type is either Invalid (with no value) or Valid with some value. For example, we can define `Maybe#(Word) x = Invalid;` or check the `Bool` value of `isValid(y)`.

In summary, the whole point is to implement FSMs in a composable way. Now we can do more interesting computation than with our combinational circuits, and we can start learning to build simple, easy-to-use module interfaces.

13 October 15, 2020

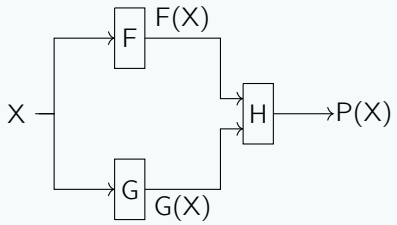
Today, we're going to move into a new topic called **pipelining**, which helps us improve the performance of our circuits. There are two main metrics we're interested in for a system:

- **Latency**: **how long** does it take from an input entered into the system to its associated output being produced?
- **Throughput**: **at what rate** can we produce valid inputs or outputs?

We'll consider each of these metrics in turn, and it's good to think about what applications should prioritize each metric. For example, an airbag deployment system should optimize for latency (because it's a one-time emergency system), but a general-purpose processor should optimize mostly for throughput, because the goal is to do lots of calculations per second.

Example 106

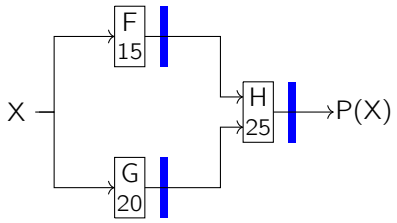
Consider a simple combinational circuit as shown below:



This circuit has three submodules F, G, H, and the latency of this circuit, also known as the propagation delay, is the sum of the times along the slowest path in the circuit. So if we have a latency of t_{PD}, then the combinational throughput will be $\frac{1}{t_{PD}}$ (that's how many outputs per second we can produce).

But notice that we're not effectively using our hardware at all times. If our input X is available at the beginning of a cycle of length t_{PD}, then F and G do their work near the beginning of the cycle, while H works near the end of the cycle. So while H performs its computation, F and G are completely idle – that isn't the best use of our hardware.

Instead, the solution is to **use registers to hold stable inputs**! If we save our values of F and G, then while H works on the current input, F and G can start working on a new input. This gives us a **2-stage pipeline** (called as such because an input X at clock cycle j gives us a valid output at clock cycle $j+2$). For example, suppose we have the following propagation delays, in nanoseconds:



We can find that the latency of the unpipelined circuit is 45 ns, so the throughput is 1/45. But if we have a 2-stage pipeline, storing our outputs in the blue marked registers above, the limiting factor for the clock is the slowest circuit component, which is H's job. Therefore, we can use a clock period of 25 ns to give a latency of 50 ns (slightly worse than before), but a throughput of 1/25 (better than before)! To summarize, we can't make our latency any better by adding registers, but we can improve our throughput.

One way that we can analyze the behavior of our pipeline circuits is to use a **pipeline diagram**:

	i	$i+1$	$i+2$	$i+3$
F & G	$F(X_i), G(X_i)$	$F(X_{i+1}), G(X_{i+1})$	$F(X_{i+2}), G(X_{i+2})$	
H		$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

The rows represent the different pipeline stages we're at, and the columns represent the different clock cycles. Basically, each set of input data moves **diagonally down** through our diagram, and we're producing a new output every cycle even though it takes two clock cycles to go through.

Let's formalize everything a bit more:

Definition 107

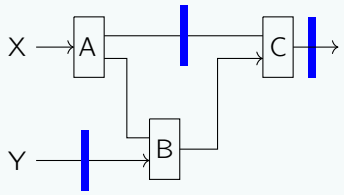
A ***k*-stage pipeline** is an acyclic circuit which has k registers **for every path** from an input to an output. (For example, a combinational circuit is a 0-stage pipeline.)

By composition, we're going to assume that every pipeline stage has a register on its output, rather than the input (we already saw this in the circuit above). If the registers are always in a consistent spot, this allows us to easily compose pipeline circuits together while using the same clock cycle patterns. And as we alluded to, the clock period t_{CLK} needs to cover the longest register-to-register path (this is a lot like how sequential circuits had a limit based on the register-register paths). Then we write down the equations for latency and throughput

$$L = K \cdot t_{\text{CLK}}, \quad T = \frac{1}{t_{\text{CLK}}}.$$

Example 108

It's possible to have ill-formed pipelines as well. For example, consider this badly pipelined circuit:

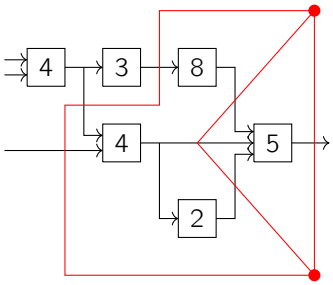


The issue with this circuit is that the path through A and C starting from X, as well as the path through B and C starting from Y, each go through 2 registers. But the A-B-C path only has a single register, so that's going to **mix our calculations together between cycles**! Specifically, we're looking at $A(X_{i+1})$ and Y_i being fed into B at the same time; that's the kind of thing that we need to be careful about.

So now let's talk about a good recipe for successful pipelining:

- Draw a line (a contour) that represents the edge of a pipeline stage, which crosses all of our outputs. The endpoints of this contour are called **terminal points**.
- Draw more lines that connect those two terminal points, making sure that **all arrows cross the lines in the same direction**.

Here is an example of this in action, with numbers representing propagation delay in nanoseconds:



And now we know where to add our registers: we put a pipeline register everywhere where a **red line crosses a wire connection**. (Because everything connects with the terminal points, we can guarantee that we will have a valid pipeline this way.) We can ask why we didn't add another line between the 4 and 3 circuit elements in the top left

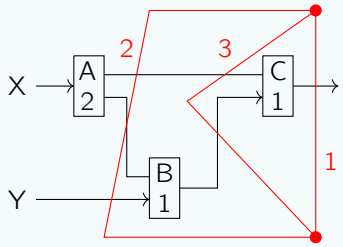
corner, and the idea is that t_{CLK} will always be at least 8 nanoseconds because of component C anyway (if we have **ideal registers** with setup time and propagation delay zero). So we already get the maximum throughput with our three pipeline stages! This means we have

$$L = 3 \cdot 8 = 24\text{ns}, \quad T = \frac{1}{8\text{ ns}}.$$

(Adding another line between A and B would make our latency worse without changing the throughput. So we shouldn't just arbitrarily add pipeline stages – we should identify the **bottlenecks**, or slowest components, of our circuit.)

Example 109

Let's go back to an earlier example with different delays and try to pipeline it in different ways. (The red lines labeled 1, 2, 3 are the first, second, and third lines drawn.)



If we have a single pipeline stage, we need to put a register at the output, so a 1-pipeline doesn't actually improve either L or T. (We've just **clocked** our circuit and turned it into a sequential one, without getting changes in performance.) So now if we add a second pipeline stage, we can put component A in its own stage and the other circuit elements in their own stage, giving us a clock time of 2 ns. Adding a third pipeline stage by dividing B and C doesn't help performance, because the bottleneck of performance is still A. Here's a table of the calculations:

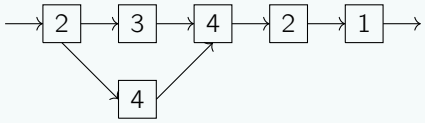
	Latency	Throughput
0-pipe	4	1/4
1-pipe	4	1/4
2-pipe	4	1/2
3-pipe	6	1/2

In other words, just adding pipeline stages willy-nilly is bad – we just isolate our slowest component. And sometimes, as we can see, we may need back-to-back registers to keep our pipeline well-formed, and this is valid.

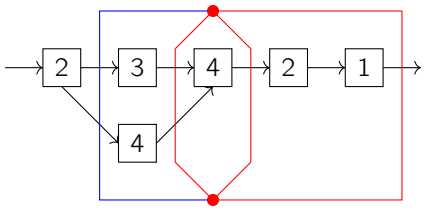
Analyzing this circuit more carefully, we can make an additional observation. If we replace A with a new circuit component which requires 2 clock cycles itself, each taking up 1 nanosecond of time, we are building **hierarchical pipelines** out of smaller ones. So now we can actually use a 4-stage pipeline with $t_{\text{CLK}} = 1$ (by drawing a red line 4 that goes through the middle of A), and our throughput improves from 1/2 to 1 (while still giving us a latency of 4)!

Example 110

Let's try to pipeline one more circuit for maximum throughput while generally minimizing latency.



We should draw contours that isolate the bottleneck component in the middle, as shown:



But the red contours don't exactly finish the job for us – the clock time is still greater than 4 ns, unless we add the blue pipeline stage to split up the $2+3=5$ ns delay from the top two circuit components. This gives us $t_{\text{CLK}}=4$, so $L=16$ and $T=\frac{1}{4}$.

For the rest of today's lecture, we'll switch gears a bit and start thinking about design tradeoffs, particularly how different goals can lead to different implementations of sequential circuits.

Example 111

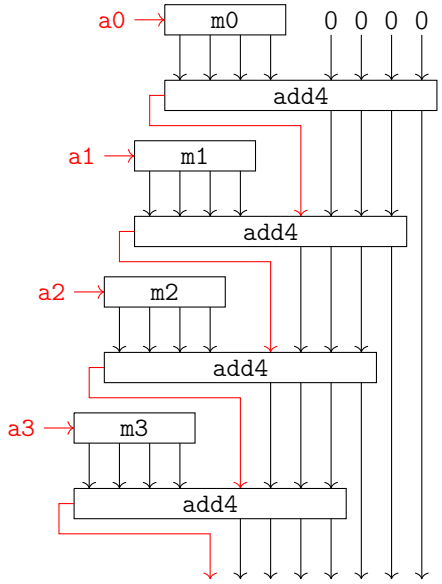
Let's consider **binary multiplication**, which is done very similarly to how we do multiplication in grade school.

For example, if we're multiplying a multiplicand $b=1101$ by a multiplier $a=1011$, then we take each bit of a and multiply it by b. In the case of binary multiplication, we always either add the value of b (shifted by some number of bits) or 0 to our result, based on whether a's bit is 1 or 0, respectively. So the equations that are relevant are (for each i)

```
mi = (a[i] == 0)?  0 :  b,
```

and then we add `m0` to `m1 « 1` to `m2 « 2` and so on. (And we're considering unsigned binary numbers here – we'll consider signed multiplication later on on our own.)

Thinking about the actual circuit, if b and a are only 4 bits long, we only need to use a 4-bit adder when adding each `mi`. Here's what the four-bit multiplication full circuit looks like schematically:



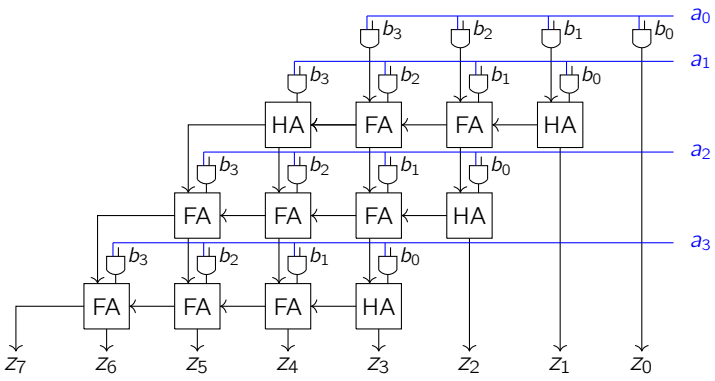
(The left and right inputs into the `add4` functions are the two four-digit numbers we add together to keep track of our running total. Notice that after the first addition, our least significant bit is already fixed, and so on.) And now we want to take this further, replacing the `add4` functions with actual circuits. We can think about what is actually happening within each box – remember that binary 1-bit multiplication is very easy, because the only time we get a

nonzero output is if both inputs are 1, and that's exactly an AND gate. That means we can write out our product in the following way, digit by digit, if we're multiplying the numbers $B_3B_2B_1B_0$ and $A_3A_2A_1A_0$:

			B_3	B_2	B_1	B_0
		$\times$	A_3	A_2	A_1	A_0
			B_3A_0	B_2A_0	B_1A_0	B_0A_0
		B_3A_1	B_2A_1	B_1A_1	B_0A_1	
	B_3A_2	B_2A_2	B_1A_2	B_0A_2		
B_3A_3	B_2A_3	B_1A_3	B_0A_3			

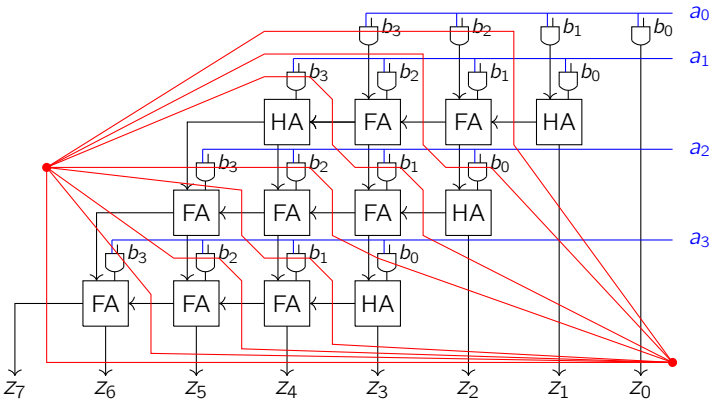
Each BA_j row here is called a **partial product** – the nice thing about viewing multiplication this way is that each bitwise multiplication B_iA_j can be implemented using an AND gate. So if we're multiplying two N-bit numbers, the hard (time-consuming) part is adding together these N different N-bit partial products, not the AND operations themselves.

We can draw out a combinational multiplier at the circuit level now. (We don't need to do an adder for `m0` itself, but at each subsequent level we're just using a 4-bit ripple-carry adder.) Here, FA means "full adder," and HA means "half adder" (when we don't actually need a carry-in, we only need to feed in two inputs):



Right now, our latency is of order N for an N-bit adder, because the longest path in the circuit goes across a full ripple-carry adder and also goes down through the various levels, giving us about $2N$ gates. So using a combinational N-bit multiplier gives us a latency of $\Theta(N)$ and a throughput of $\Theta(1/N)$. (And we can also check that our area is $\Theta(N^2)$.)

If we pipeline this circuit, there are two different methods we can try. We can separate out the different rows of the diagram as different pipeline stages, but unfortunately, the t_{CLK} is still of order N, because still need to go through each of the ripple-carry adders! So then this means $t_{CLK} = \Theta(N)$ still, but we have $\Theta(N)$ stages in our pipeline. This means our latency has worsened to $\Theta(N^2)$, and the throughput is still $\Theta(1/N)$. Instead, we need to break the "carry chain" by drawing diagonal contour lines as shown:



In each pipeline stage, we only have one full adder worth of delay now, meaning that our t_{CLK} is $\Theta(1)$ (it's just the amount of time through a single full adder), and we have $\Theta(N)$ different pipeline stages. So the latency remains $\Theta(N)$, but we've **improved our throughput** to $\Theta(1)$, which is significantly better. (In words, once we've filled our pipeline, we're able to produce a new valid output once per cycle!)

There are a few other decisions that we can make – if we don't have unlimited area available, we may be able to take advantage of the fact that we have **repetitive logic**. Since we have $(N-1)$ adders in our multiplier, we can make an improvement by feeding inputs back into a single adder. This will now require N cycles per output, but our area is now only $\Theta(N)$ – this is called a **folded circuit**. Essentially, we take our initial values of A and B, and we keep adding our values of `mi` to the previous value of our total sum, remembering to shift to the right by 1 bit. And this gives us reduced area, at the cost of having a lower throughput.

In summary, there are basically three different models to consider in pipeline design:

1. Several combinational modules in a single pipeline stage, optimizing for latency,
2. One module per pipeline stage, optimizing for throughput,
3. A folded reusable block, optimizing for area.

If we care about clock time, we prefer them in the order $2 < 1 < 3$. But $3 < 1 < 2$ for area, and $3 < 1 < 2$ for throughput. So depending on what our goals are, we'll implement circuits in very different ways, and we'll continue this discussion next time.

14 October 20, 2020

Today, we'll finish the second module of this class (digital design and logic) by looking at some different tradeoffs in designing sequential circuits.

Now that we've seen some different styles of design for digital logic, it's worth emphasizing a key difference between design of software and hardware programs (which also explains why we use a hardware description language instead of a software programming language): **timing**. When we write a software program, we specify what happens, but not when the program executes those instructions. (For example, we can make statements about various operations, and converting that to assembly gives us a basic set of instructions, but at no point do we specify how long each instruction takes.) And in contrast, for hardware design, we need to specify what happens on every single clock cycle, which in turn actually determines the length of our clock cycle and thus the overall performance.

Another difference between software and hardware is how restricted our interface can be: there aren't very many ways to optimize a particular software program, other than just generally aiming for fewer instructions. But in hardware,

we have a lot more control: we can implement the same functionality in many ways, looking at **various area-time-power tradeoffs**. Remember that we can use **throughput** and **latency** as optimization metrics, but we also need to balance this with costs like **area** on a chip, **power** consumption, and how much **energy** it takes to execute a task. And there are other metrics, too, like the cost of design and operation – how many engineers does it take to design and verify a functionality? So abstractly, we have a multi-dimensional space of possible optimizations, and we need to adjust performance in certain axes at the cost of performance in the others.

Fact 112

There isn't always a standard good answer for how to optimize: what matters is the end design goal and what tasks are being performed by our given processor.

So this lecture will start by looking at throughput, latency, and area – power and energy are outside the scope of 6.004, so we'll skip over that for now. (Power has to do with chip area and minimizing transitions, but the techniques are more specialized than we'll discuss here.) Then we'll revisit the pipeline circuit idea from last time – we're still missing a few key elements that help integrate them properly into broader circuits, which are the ideas of valid bits, stall logic, and queues. And finally, we'll study how to **generalize** everything, so that a single piece of hardware can solve multiple problems at the same time, rather than just designing specialized circuits.

Recall that in the past few lectures, we've seen a few interesting benefits that sequential logic gives us. Sequential circuits can implement more computations than combinational circuits, because we can consume a variable amount of input or produce a variable amount of output, and the whole process can take a variable number of steps (like with loops in the factorial and GCD circuits). And in general, sequential circuits also allow us more design flexibility even when combinational circuits also suffice – specifically, we can improve throughput with **pipelined circuits**, or reduce area using **folded circuits**.

So let's look more carefully at the pipelining implementation. Remember that pipelining **breaks a combinational circuit into multiple stages** by storing intermediate values in registers, and that means that each computation takes multiple cycles. That means that different stages are computing inputs from different cycles, which helps us **improve throughput** (by lowering the clock time), even if it may hurt latency.

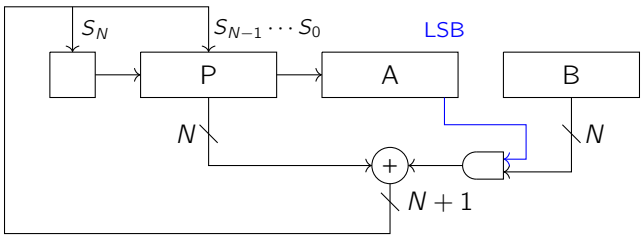
The way we analyze pipeline circuits is using pipeline diagrams: on the horizontal axis, we keep track of clock cycles, and on the vertical, we keep track of which pipeline stage we're looking at. Indeed, we can check that at each vertical time slice in the diagram below, we're processing an input at each stage:

Cycle	i	$i+1$	$i+2$	$i+3$
Stage 1	$F(X_i), G(X_i)$	$F(X_{i+1}), G(X_{i+1})$	$F(X_{i+2}), G(X_{i+2})$	
Stage 2		$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

At the end of last class, we analyzed the pipelined multiplier. Without pipelining, just taking a combinational approach, an N-bit adder has area $\Theta(N^2)$ (because that's how many gates we have) and propagation delay $\Theta(N)$ (we need to go "across" through a ripple-carry adder, and also "down" through the different levels of addition) leading us to a latency $\Theta(N)$ and throughput $\Theta(1/N)$ (because we just have a single pipeline stage or cycle). But if we do the "diagonal pipelining" from last class, breaking up our circuit into $\Theta(N)$ different levels, we still have an area of $\Theta(N^2)$, and the latency is still $\Theta(N)$. But the clock period is now a constant $\Theta(1)$, and thus our throughput is much better: it's $\Theta(1)$ as well.

We also talked last time about having a **folded circuit**, which comes up when we have a combinational circuit with repetitive logic. The main point is that if we use a single adder or single component multiple times in the same calculation, we're reducing the area but increasing the throughput. Here's an illustrative diagram of a multiplication

by repeated adding, which takes N cycles. Basically, we initialize P to 0 and load in our initial A and B inputs. Then we add B to P if the **least significant bit** of A is 1 (represented in blue), and in any case we shift S_N, P, A all to the right by one bit. This way, the input A has been shifted away after N cycles, and our final $2N$-bit result is stored in P and A.



This leads us to the same three design alternatives as last class: using multiple combinational blocks in a single pipeline stage (A), multiple pipeline stages with one block per stage (B), and a folded reused block which does multicycle computations (C). It turns out the clock times satisfy $B \approx C < A$, which leads to $C < A < B$ for throughput performance (though the $C < A$ part is less noticeable), while $C < A < B$ for area performance (with $A < B$ less noticeable).

So far, we've assumed that the clock time only depends on our circuit, so that a lower propagation delay lowers the clock, and that helps us minimize latency and maximize throughput. But there might be a propagation delay that is already larger from other circuits, or we might be in a situation where operating the whole chip at a high frequency is limited by power consumption. Then the throughput and latency tradeoffs become more nuanced:

Example 113

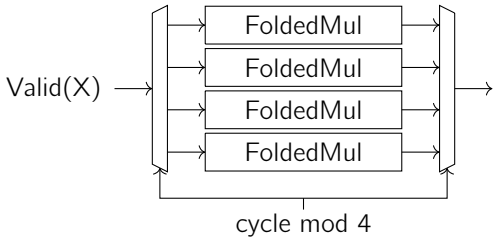
Suppose we have two options for a pipeline: 4-stage vs 2-stage. If $t_{\text{CLK, 4}} = t_{\text{CLK, 2}}/2$, then the throughput doubles while the latency stays the same (which is good). But if this circuit isn't the limiting factor, and $t_{\text{CLK, 4}} = t_{\text{CLK, 2}}$, then the latency doubles while the throughput stays the same (which is bad).

Another idea for increasing throughput is to do some more parallel computation by **replicating** a circuit. For example, if we have two pipelined circuits in parallel, then we can process two values each cycle. Then the clock period does not change (the two circuits behave identically), but our throughput doubles, and our area is also doubled. This may seem like a naive method of optimization, but modern chips have a lot of space, so we do this a lot in practice! (This has to do with **vector processing**, which will come up later in the class. Basically, modern designs have processors with lots of replication.)

Example 114

Suppose we have a 4-stage pipelined multiplier PipedMul with throughput $1/t_{\text{CLK}}$, and we also have a folded multiplier FoldedMul that takes 4 cycles per output. Assume the clock cycle stays the same, so that the throughput decreases by a factor of 4 but we have lower area.

The idea is to put four FoldedMul circuits in parallel and use a **mux** and a **demux** to select inputs and outputs between circuit numbers $0, 1, 2, 3$ sequentially (which we can see on the left and right of this diagram):



At each clock cycle, we input a valid input Valid(X), which gets fed into the circuit indexed with the value of the clock cycle mod 4. (The other three parallel circuits get fed in Invalid input.) During this cycle, we also take the output from that circuit (which comes from the input from 4 cycles ago). This way, we can ensure that each folded multiplier is taking in an input and propagating an output once per 4 cycles. This circuit actually has about the same area and clock time as the original PipedMul circuit, and we can try synthesizing each one to how they compare in practice.

So now we can talk a little bit about some **pipeline extensions** to improve implementation in the real world. In practice, pipelines need to take in inputs from a **producer** and produce outputs to a **consumer**. This gives us a few complications:

Problem 115

Producers may not always have an input available at every cycle.

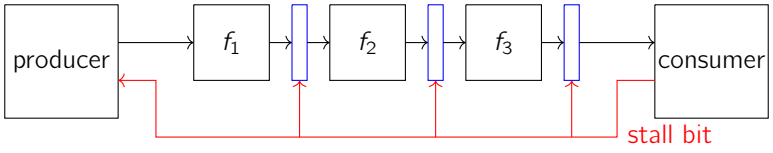
We deal with this by tracking whether the current input our output is Valid or Invalid, and this is implemented using the Minispec `Maybe` types. And then we need to make sure to propagate that information forward at each pipeline stage, so we **tag each stage** with what is called a **valid bit**. This basically gives us a pipeline diagram that propagates both valid and invalid bits forward, and we call the invalid values **pipeline bubbles** (because they're cycles where no work is actually done). Here's a picture of possible propagation of valid and invalid bits for a three-stage pipeline:

Cycle	1	2	3	4	5	6	7	8
Stage 1	V1	V2	**Inv**	V3	**Inv**	**Inv**	V4	V5
Stage 2	Inv	V1	V2	**Inv**	V3	**Inv**	**Inv**	V4
Stage 3	Inv	Inv	V1	V2	**Inv**	V3	**Inv**	**Inv**

Problem 116

Consumers may not be able to accept an output every cycle, either because they have lower throughput or take a variable amount of time to process its inputs (our outputs).

In this case, we need to **stall or freeze the pipeline** (and by extension, the producer too, if we're backlogged). We implement this with a **stall signal** that behaves similar to an "enable:" if the stall bit is 1 on some given cycle, then we hold our pipeline registers instead of proceeding with the calculation, as shown below. But otherwise, the pipeline proceeds as normal.



And now we can **put our two ideas together**: if we have certain bits that are invalid in our pipeline, and we have a stall signal, we can **fill in the pipeline bubbles** and avoid needing to stall all the way to the beginning, and we do this by feeding in the **AND of our stall bit and our valid bit** as an "enable." That is, **we stall a pipeline stage only if we get a stall signal from upstream, AND also the corresponding register is currently Valid** (meaning it is not available to take in new inputs). So now our pipeline diagram will look like this:

Cycle	1	2	3	4	5	6	7	8
Stage 1	V1	V2	**Inv**	V3	V4	**Inv**	V5	V5
Stage 2	Inv	V1	V2	**Inv**	V3	V4	V4	V4
Stage 3	Inv	Inv	V1	V2	V2	V3	V3	V3
Output	Inv	Inv	Inv	V1	V1	V2	V2	V2
Stall	False	False	False	True	False	True	True	False

Some good points to notice are that the red bold **Inv**s come from the producer having nothing to do for us, and then those start to propagate as before. But during cycle 4, the consumer stalls (it says that it can't take any outputs), so in the transition from cycle 4 to 5, V1 and V2 both freeze, while V3 moves to fill the invalid spot (and thus we can still take in a new valid bit from the producer).

Problem 117

And now we just have one more problem, which we'll need to think about during our design project too: right now the stall signal is driving a lot of register enable bits at the same time, and this fan-out causes a lot of extra delay when we have lots of registers.

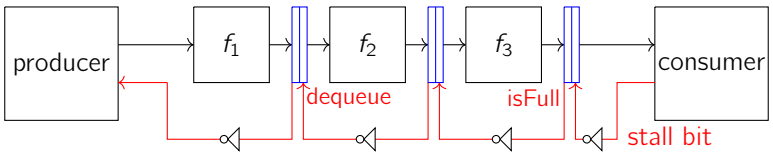
Eventually, this stall propagation delay may start influencing our clock cycle time, and that's bad! The workaround turns out to be to use **queues** instead of registers – saving some extra space at each register means that pipeline stages don't stall unless our entire queue is full, and this means we can do optimizations locally. (The real-life analogy is that if we have a lot of cars on a highway, we leave some extra space before the car in front of us, so that we only need to keep track of that one car instead of all the cars before it.)

Example 118

For a concrete implementation, let's consider a two-element first-in, first-out queue.

Such a queue holds up to two values – it can output the first enqueued value, and we can feed in a **dequeue** input to tell us whether we want to advance the queue, or get an **isFull** output which tells us whether we still have space. Basically, we have two registers **e0** and **e1** with valid bits, and we always read from e0. If both bits are valid, then we can't enqueue anymore, and then we have some logic that helps us decide which values are introduced into the two registers when we do queue.

And the point is that a queue allows us to decouple our long combinational circuit into shorter ones, as shown (each of the registers has been replaced with a two-element queue):



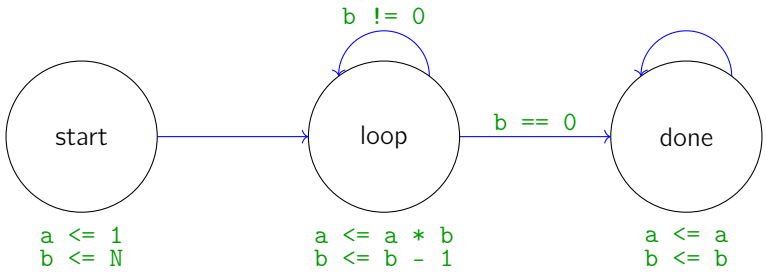
Basically, if one pipeline stage's queue is not full, then it's okay to feed an input into that stage, and we do this by calling dequeue on the previous queue. So now we don't rely on the stall signal to know whether or not to stall at a particular stage, and the tradeoff here is that **in order to avoid long combinational paths**, we can't enqueue to a queue that's full (even if we'd dequeue on that cycle anyway). These circuits are called **skip buffers** – returning to the highway analogy, it's as if we have up to two car lengths worth of space in front of the car that we are driving.

So in the remaining time, we'll close our chapter on FSMs and start looking at material of our next lectures, which is about how to go from our special-purpose circuits to general-purpose processors that are **programmable**. So far, we have learned methods to take a given problem we need to solve, design a procedure to solve it, and then design a finite state machine to implement that procedure. But by Thursday's lecture, we'll learn how to design a general-purpose computer, and we're going to try to start this discussion by generalizing our FSMs.

Example 119

Consider a factorial FSM, which we've already spent some time thinking about in this class. Recall that in order to compute factorial(N), we initialize two variables a and b to 1 and N, respectively. Then we keep multiplying a by b and decrementing b until it hits 0, at which point we're done.

We can encode the algorithm here as a **high-level FSM**, which describes **cycle-by-cycle behavior** rather than our state space. The key point is that our states are the **different logic steps** that we can take during a given clock cycle:

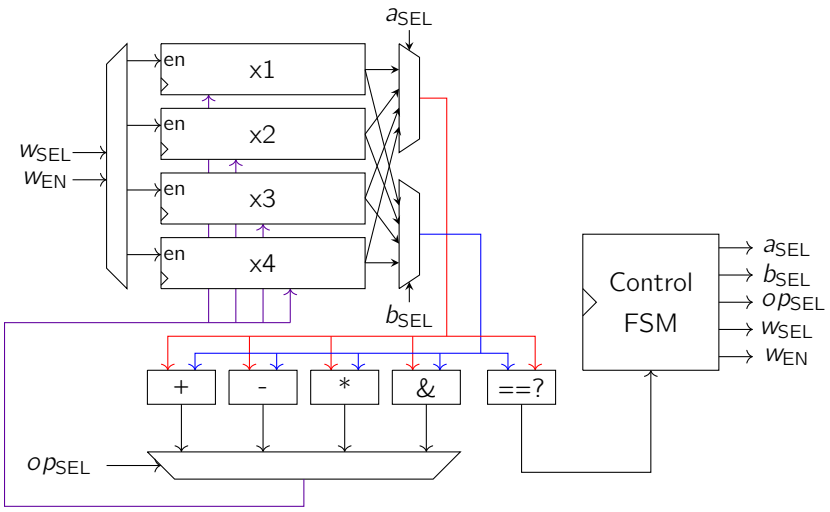


So at each cycle, we're in one of these three states, and we transition between states and do operations based on the values stored in our registers.

In order to design the hardware for this job, which is called a **datapath**, we first need to implement all of the logic for the assignments (listed in green below the states): we put `a` and `b` in registers, we implement the code that we want to perform in each of the three cycle states, and then we use muxes to switch between the three different states so that we can manipulate the registers however we want. And then we need to add an additional small piece of circuitry, called the **control FSM**, which implements our three-state finite state machine above! This control FSM is itself a register – it stores the state we're currently in, and it outputs the "select" bits for the two muxes mentioned above.

But we've gone through a very specific problem, and we want to reuse this datapath for other problems. In fact, **changing the control FSM** allows us to solve other problems that aren't just the factorial function, like squaring or multiplication. But this is still very limited, because we don't have a lot of registers (only two), and we only have so many operations and inputs that are possible to implement.

So we'll generalize the above datapath a bit, and the resulting diagram is starting to look like what we'll see next lecture:



The idea is that we've now gained a lot of flexibility: every register can be used as a source or destination for each of the operations. At each cycle, the datapath reads two operands from two of the registers (red and blue), performing the operation on those two operands, and then storing the result in one of the registers (purple). In this case, op_{SEL} tells us which operation to perform, w_{SEL} tells us which register to select, w_{EN} tells us whether or not we store an input, and a_{SEL} tells us which register the first operand pulls from.

So this allows us to do computations using any of the operations `+`, `-`, `*`, and `&`, and it also allows us to store values in four different registers. And we can then write a control FSM to generate the mux control signals, but then we can implement a high-level FSM with states that start to look like our **assembly instructions** from the beginning of 6.004.

But we still have an issue – every time we want to work on a new problem using these operators, we need to design a new control FSM and implement it with hardware. This is called **programming the datapath**, and the first digital computers did actually work this way. But the next advancement is to make this even more flexible by storing **programs in memory** as a sequence of instructions, which will allow us to use only a single control FSM for all computations. We'll see how to do this next time!

15 October 22, 2020

Today, we'll start a new section of 6.004, the **computer architecture** module. At this point, we know how to build combinational or sequential logic, so we can build hardware that implements a given finite state machine. But now, instead of trying to build hardware that's specific to a particular task, we want to build a general-purpose processor that can implement any of our programs.

Fact 120

We'll be following the **von Neumann model**, which was introduced by John von Neumann in 1945 and is what almost all modern computers are based on.

Such a model contains a **main memory** (storing the program and the data associated with the program), a **central processing unit (CPU)** (accessing and processing said memory), and a **input-output device** (allowing us to talk to the outside world). The main idea is to have our memory store both data and instructionsn and to have our CPU fetch, interpret, and execute instructions of our program. We saw how some of this worked when we studied RISC-V

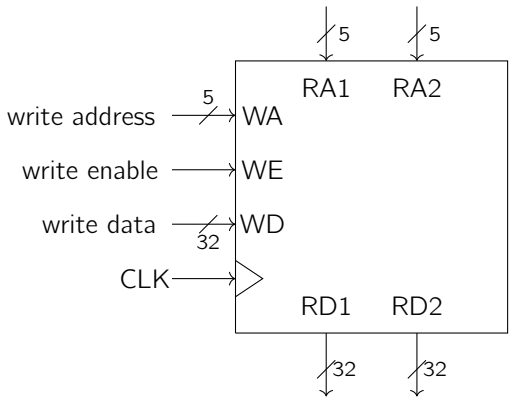
instruction set architecture earlier on: we take each instruction from our program and encode the information into various **bit fields** of our instruction, so instructions just look like certain strings of binary bits. But data also looks like a string of binary bits, and the way that our CPU can distinguish between instructions and data (and also the stack and heap that we've previously discussed) is **where they live** in main memory.

First, let's take a closer look into what this CPU looks like. Since main memory stores both data and instructions, so our CPU has two pieces, called the **datapath** (the "muscle") and **control unit** (the "brain"), respectively. As we've started exploring last lecture, the datapath contains a bunch of registers, as well as some way of accessing registers and combining the source operands with ALU operations (possibly writing a value back into the regsiter file when the computation is complete). And the control unit has a **program counter** keeping track of the instruction at the address that needs to be executed next, and then it uses certain logic to decode instructions into certain **control signals** so that the datapath can properly do the execution.

So instructions are the basic unit of work for us here – we know that each instruction has an **opcode**, as well as source **operands** and **destination**. A von Neumann machine basically keeps looping a certain process: it first fetches and decodes our instruction, then reads the source operands, executes the instruction, writes to the destination operand if necessary, and then computes the next program counter so we know where to get the next instruction.

Let's start building this processor – we'll implement datapaths for each instruction class separately and then gradually merge them with muxes. (This is called an **incremental featurism** approach.) We'll start with an implementation of our ALU, followed by load and store instructions, and finally talk about branch and jump instructions – this is the same order as we discussed RISC-V assembly at the beginning of class! And the hardware tools that we can work with are registers, muxes, a black-box ALU (which we've already constructed in previous labs), two memories (a read-only instruction memory and a read-and-write data memory), and a **3-port register file** which we'll discuss in more detail soon.

In our specific case, we'll have a processor with 32 registers. Our register file can read two source operands and write to one destination per cycle, so we'll need to have two read address ports, **RA1** and **RA2**, which each select one of the 32 registers to read on each cycle. In order to have write functionality, we also need to have a write address **WA**, which sets one of the register enables to 1 so that we can write to it. Each register here is a **load-enabled register**, which is a D flip flop with a multiplexer in front of it, so that the value stored is constant until the enable bit gets toggled. So we can use the block diagram below to describe our register file:



As we described above, each of **RA1** and **RA2** tell us which register to read from, so each of them require 5 bits of input. The 32-bit data from those registers is then available at the ports **RD1** and **RD2**, respectively. For writing, we need an extra signal **WE** (write enable) that tells us whether we're writing to a register on that given cycle, **WA** (write address) telling us which register to write to, and WD (the specific 32-bit write data that we want to write). One key to understanding this register file is that **there are two combinational read ports and one clocked write**

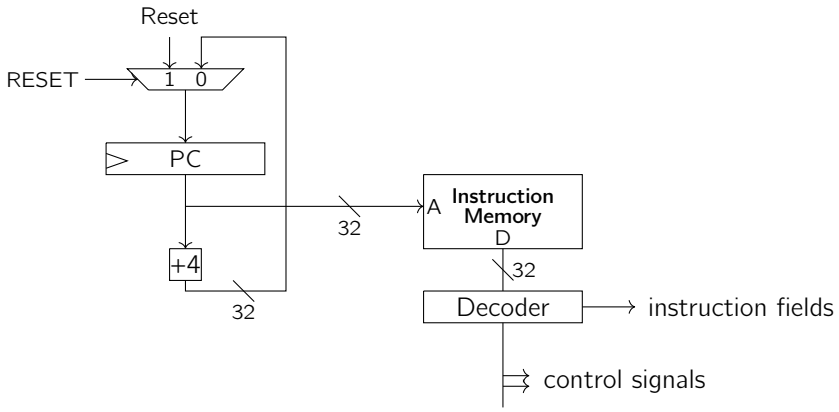
port, so it's important to take a look at how timing works for this register file.

If we're provided a read address **RA** at any point, then after some propagation delay, the register file will make the read data available in the corresponding **RD**. But writes are different – they can only happen at the rising edge of the clock, and that means we need stable and valid data for some setup time t_s before the rising edge, as well as some hold time t_h afterward. So the write enable signal must be 1 during that whole time, and the 5-bit write address **WA** and 32-bit write data **WD** must both be stable during that interval. And once some propagation time has passed after the rising edge of the clock, this new value is available to the read ports (the data has been stored into the box). An interesting case is where **WA** is the same as **RA1** (that is, we read and write to the same register): we will always read the old value of the register until the next clock rising edge occurs, so things work properly in this case.

Fact 121

For now, we're going to deal with a simplified memory (for our implementations in lab 6). Specifically, we'll assume that memory behaves like the register file, meaning that loads from our memory are combinational (data can be returned in the same clock cycle as the load request), but stores are clocked. This is called a "magic memory" module, and it's not a realistic model, but we'll discuss memory technologies more realistically next week.

So now we're ready to start building the processor bit by bit. First of all, we need to be able to fetch and decode our instructions, and the diagram for that is shown below:

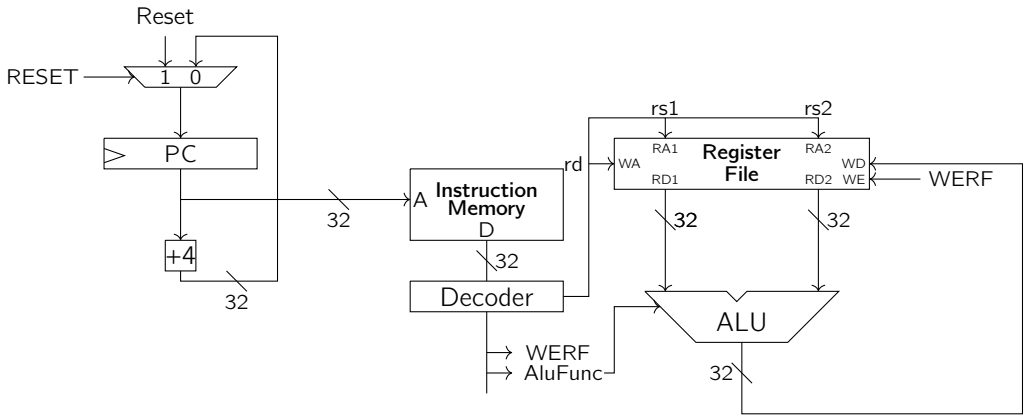


Our program counter **PC** is initially reset to 0 (so we can start the instructions at the beginning), and the value of **PC** is passed into the address **A** field of the instruction memory. Next, we do a load from the instruction memory, and it provides (in the data port **D**) the 32-bit instruction that we just read from the memory. (As mentioned before, our instruction memory is read-only, so there are only these two ports: address and data.) Next, we have to decode that instruction, and that's what we'll be doing primarily in lab 6 – writing the decoder. This decoder needs to look at the binary bits of our instruction and figure out what the different fields mean. Some bit fields are used immediately, like the source registers and immediate values, and these are called **instruction (word) fields**, but other times multiple fields (**opcode**, **funct3**, **funct7**) need to be considered simultaneously to generate **control signals**. For now, we'll assume that our pc is just incremented by 4 each time (this will be augmented by control-flow implementation later).

Example 122

The **register-register ALU operations** are ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, and AND. They all use **rs1** and **rs2**, the two source registers, perform some ALU operation, and then write the result to **rd**, the destination register.

We start every instruction by taking the value of the program counter and feeding that into the instruction memory and then the decoder. In these register-register instructions, we always take bits [19:15], [24:20], and [11:7] to determine **rs1**, **rs2**, **rd** respectively; those are fed into the address ports of our register file. Then we incorporate in the ALU that we've built – it receives two source operands out of the register file, but then it also needs to figure out the function **AluFunc** to perform. In order to do that, we need to decode the 32-bit instruction: first of all, the opcode should be of type 0110011, which tells the decoder we have a reg-reg ALU instruction. So then our decoder will produce an **AluFunc**, and we use **funct3** and **funct7** (in fact only bit 30 from funct7) to extract that final function. And there's an additional control signal, **WERF** (write-enable register file), which is always set to 1 in these operations.



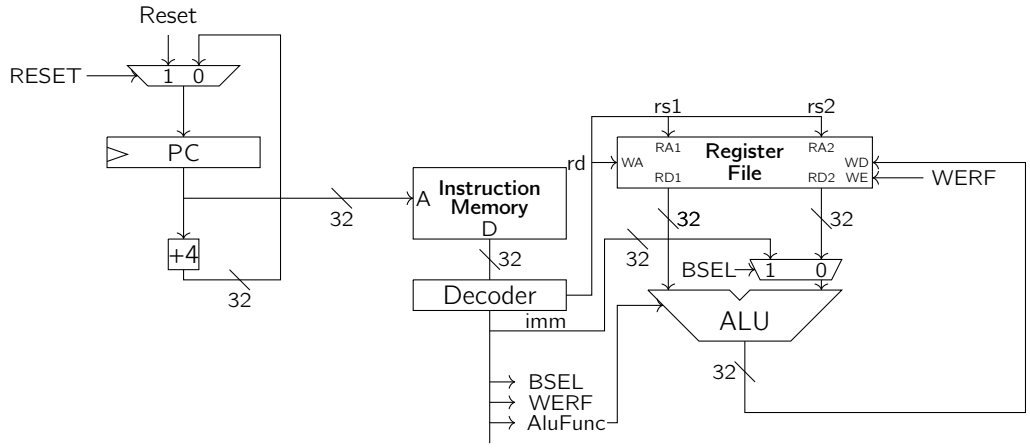
So let's walk through the data flow – once we've fetched and decoded the instruction, we fetch our two source operands, feed them to our ALU along with the **AluFunc**, and feed the information back into the **WD** port (while setting **WE** to 1 so writing is allowed). And this is all the hardware that we need to support register-register ALU operations!

Remark 123. *There's one small caveat – it's possible that we have* **invalid operations**. *For example, when the decoder looks at the instruction and evaluates the opcode, if it can't interpret the rest of the bit fields in a valid way (for example, because* **funct3** *and* **funct7** *don't give us any valid* **AluFunc***), we'll have an unsupported instruction, and that means we need to make sure we don't update any states or open the write-enable port.*

Example 124

The **register-immediate ALU operations**, which include ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, and SRAI, all use one source operand **rs1** and a constant **imm**, but are otherwise the same as reg-reg ALU operations.

We can use the same datapath as before, but this time our opcode is 0010011 (register-immediate ALU). So our ALU needs to operate on the value of the **rs1** register like before, but the second operand now comes from the immediate value encoded directly in our instruction. To allow for both functionalities, we add a multiplexer which allows us to select either the **rd2** port from the register file or the immediate value **imm**, which is just bits [31:20] of our instruction (sign-extended to a 32 bit value). And we still have to figure out the **AluFunc**, by looking at the **funct3** field (and also bit 30 to distinguish between arithmetic and logical shifts, but that's just a detail). So now that we've decoded our instruction, the datapath again feeds in the two inputs into the ALU (now possibly taking from **rs1** and **imm** instead of **rs1** and **rs2**), toggling using our **BSEL** signal. And the result can be passed along to the write dataport of the register file, at which point we set the write-enable port **WE** to 1, just like before.

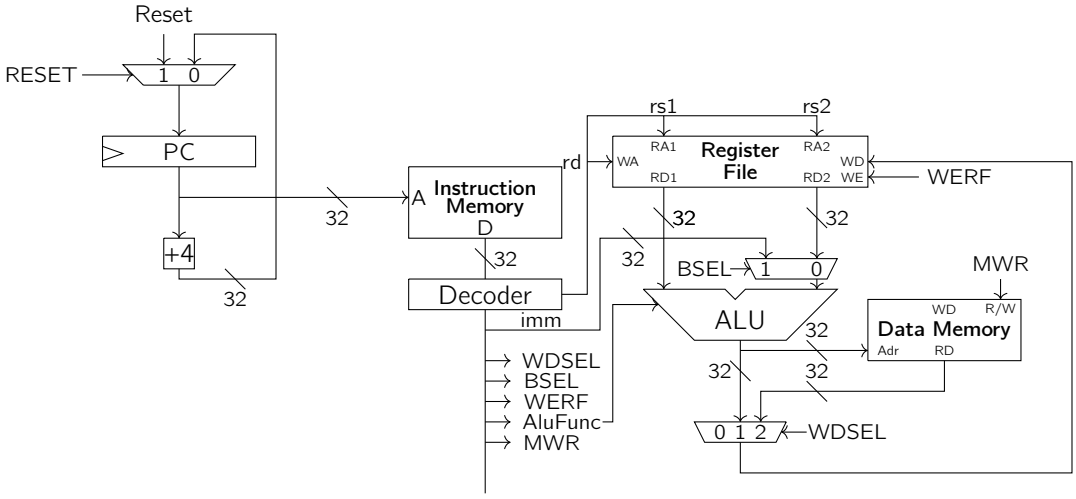


Now we have hardware that can execute all of our ALU instructions, and we're most of the way there. Let's now look at loads and stores – the key difference is that we need to access our data memory, so we'll need to add that to our hardware.

Example 125

Load instructions basically store a value to one of our registers from data memory.

We will need to compute the memory address from our instruction – we do this by looking at the contents of **rs1** and adding an offset **imm** to it, as we saw near the beginning of class. Load corresponds to the opcode 0000011, and that tells us to produce an address that will be sent to the memory (for loading new values into our register file). Our hardware already the capability of adding **rs1** to the immediate **imm** – in our existing datapath, we just set **BSEL** to 1 and also set **AluFunc** to ADD. Then the output of our ALU is not what we want to store – it's just the address that we want to feed into our data memory, and we feed it into the **Adr** port in the diagram below.



And now we need another mux, associated with the control signal **WDSEL** (write data selector), which takes on the value 2 instead of 1 if we want the data at a given address instead of just the address when it comes out of the ALU. Also, we add a **MWR** (memory write) signal, which tells us whether we're reading or writing from data memory, and in this case we want to set it to R (read). Remember that our data memory is assumed to be combinational, so we can immediately get the memory contents at the requested address (in the same clock cycle), and then we can feed this back into our register file as usual.

Example 126

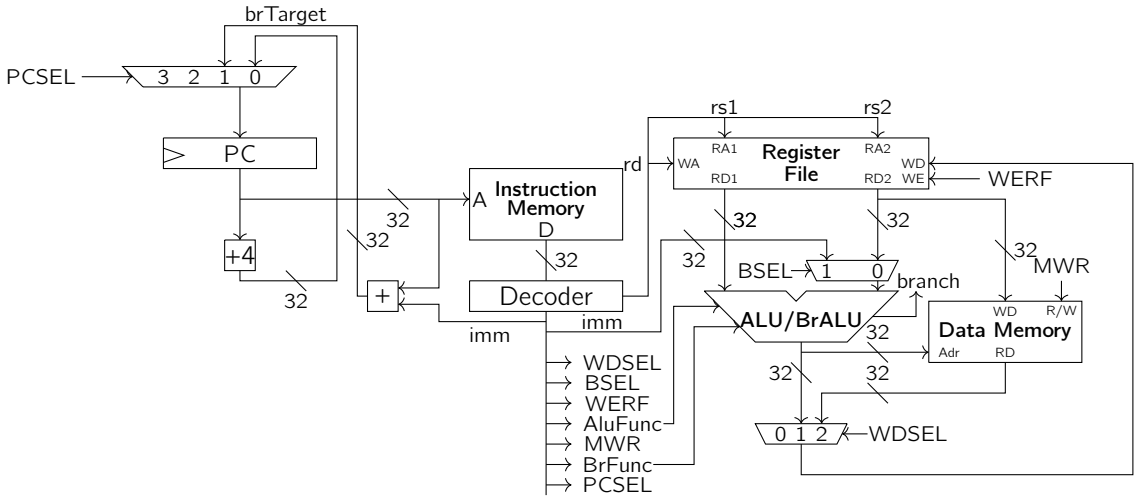
Similarly, store instructions take a value from our register file and put it into data memory.

Store instructions have an opcode 0100011, and this informs the decoder that we have an immediate, just like in load instructions. Figuring out the location in data memory to store to works exactly like the address computation for loads above, so the idea is to set **ALUFunc** to ADD and **BSEL** to 1 again. And the only other thing is that this time we **provide data to memory** rather than requesting it, and the data comes from the **rs2** register! So we need to add another path from the **rs2** register into the **WD** (write data) port, and we set the control signals **BSEL** to 1 (so that we can use the immediate), **ALUFunc** to Add, and **MWR** to W (write). But this time we don't need to write anything to our register file, so we **set our write-enable WERF** to 0 this time, so that we don't accidentally specify a register **rd** to write to. (And **WDSEL** can be anything, because the register file ignores that input anyway.) The illustration for this will be included in the next diagram, since it isn't much more complicated than the previous one.

Example 127

The last set of operations left are the control flow operations. First of all, the **branch instructions** BEQ, BNE, BLT, BGE, BLTU, BGEU are all very similar – we need to extract two registers **rs1** and **rs2** and compare the corresponding data values, and then we update the program counter accordingly using an immediate **imm**.

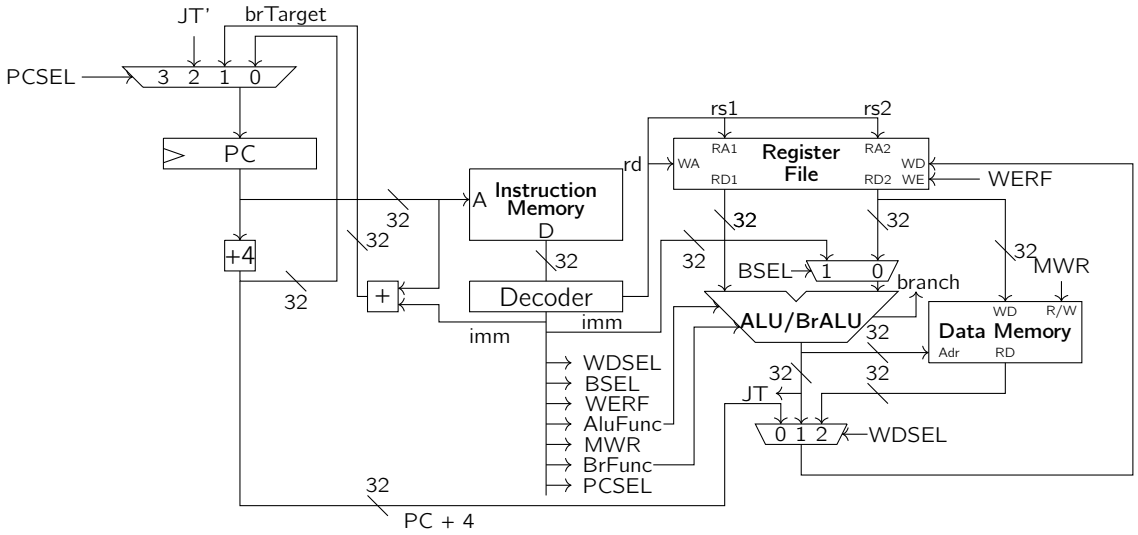
The **branch functions** that we're implementing now require something a little different from the **AluFunc**s we had before – we'll need to add a new **BrFunc** as a control signal. To do this, we add a component to the ALU beyond the one we built in lab 4, which we call a branch ALU. This branch ALU also takes in two operands, but it uses the **BrFunc** and returns a Bool (single bit) instead of a number, specifying whether we take the branch or not. We've put that into our datapath (seen below), and now we need to handle branch instructions by decoding the instruction. The opcode (1100011) tells us the rules here: we need to create a branch output signal by using the **BrFunc** function (determined by the **funct3** bits) on the values of **rs1** and **rs2**, and we set **BSE**L to 0 (because we're comparing two registers, rather than using an immediate). Then our branch ALU gives us a Bool telling us whether to take the branch or not, and now if we want to take the branch, this is the first time we want to make our next program counter not just (**PC** + 4). So we add a multiplexer at the input to our program counter with a control signal **PCSEL** (pc select) – in the regular instructions, **PCSEL** is set to 0, and it's also set to 0 if branch is false. But if we want to take the given branch, we set **PCSEL** to 1, and then the value of the next PC is **brTarget**, which we calculate using our new adder (taking the current **PC** and adding it to some immediate **imm** that the decoder tells us).



Example 128

Each of our remaining instructions is in its own category: JAL, JALR, and LUI. We'll just talk about JALR (jump-and-link register) right now, because it's the only one that requires additional hardware.

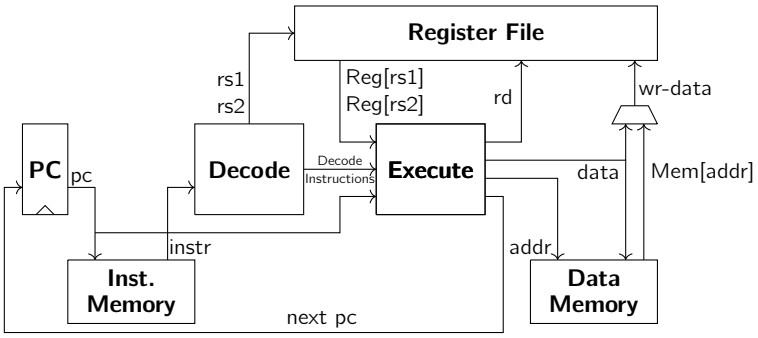
A key point is that both JAL and JALR update **both** the **rd** register and the **PC** register. JALR has opcode 1100111, and the resulting decoding that needs to be done has several aspects. We want the ability to write the value (**PC** + 4) into our register file, and we do this by taking the (**PC** + 4) adder and feed that into the 0 of our **WDSEL** mux. In addition, we'll produce a jump target **JT** telling us where we want to end up: this computation is done by using the pre-existing ALU hardware to take the contents of **rs1**, add an immediate value **imm**, and using that to update **PC**. The diagram is shown below.



So the data flow works as follows: **PC** + 4 is fed into the **WDSEL**, and then we set **WE** to 1 and **WDSEL** to 0 to complete the register update. Then we still need to update our **PC** register, which we do by having our ALU take in register **rs1** and an immediate **imm**. Thus, **BSEL** is set to 1, **AluFunc** is set to Add, and the result of that addition is our jump target **JT**. This is then related to the new **PC** value, fed into input 2 of our **PCSEL** mux – since our jumps are unconditional, we always set **PCSEL** to 2 in a JALR instruction. (A small detail is that we always replace the least significant bit of **JT** with a 0, and that's written as **JT'** in the diagram.)

With that, we've basically created our entire RISC-V processor! In lab 6, we'll build a **single-cycle RISC-V processor**, where four components are already written for us: a register file, a PC register, our instruction memory, and our data memory. We are told to write two functions: the **decode** function, which takes the contents of the program counter, feeds that address into the instruction memory to get our data field, and then produces a bunch of outputs. Specifically, **rs1** and **rs2** need to be fed to the register file so that it can provide the data from those registers. The decode function will also need to communicate with the **execute** function, which primarily consists of our ALU but also includes some additional logic. This execute function gets our two register data values, and it also takes in the **decoded instruction information** from our decode function. Finally, the value of the **PC** also needs to be passed into the execute function, and now the execute function is responsible for producing a bunch of outputs too – when doing regular ALU operations, it produces the output data that is written to the registers, as well as the destination register **rd**. In addition, this is also where we initiate loads and stores, so the output of **execute** is also used as an address field for our data memory. Depending on whether we load or store, we need to pass information of **rs2** to the data memory (for a store) or read value from the memory into our destination register (for a load). Finally,

the execute function needs to figure out the next **PC** value so that we can begin this whole cycle again. Here's a schematic diagram:



To conclude, let's break down the different components for our two functions one last time. Our decoder takes in the 32-bit instruction and extracts the various fields from it, and we've identified all of the different possible things it can identify: instruction type, the **AluFunc** and **brFunc**, the register fields **rd**, **rs1**, and **rs2**, and the 32-bit immediate **imm** from different bits based on the opcode. (Note that the decoder needs to get the correct bits based on which type of instruction it is and convert that into a 32-bit immediate with proper sign extension.) **No instruction will make use of all of these fields**, but that's what our control signals are for: they tell us which values we actually need to use for each instruction. (We can set some fields to default values if it doesn't matter.) And our execute function gets a few different inputs – the values from the register file, the decoded instruction fields, and the value of program counter **PC**. It needs to do some logic (ALU, branch ALU, and computing the new **PC**), which allows us to output the destination register **rd**, data to write (to either the register file or memory), the address for our load and store, and the next **PC** value.

In a normal semester, this is a particularly fun lab, because we get a "RISC-V inside" laptop sticker when we building the processor. When we're all back on campus, we can get our sticker if we want!

16 October 27, 2020

Today, we're going to start exploring how actual memory works, so that our processor model can become more realistic. We'll start by talking about some different memory technologies, looking into the details of how each one is physically built, and then we'll begin to talk about how to make use of these memory technologies to improve performance.

Here's a helpful reference table:

Memory type	Capacity	Latency	Cost / GB
Register	100 B	20 ps	High
SRAM	10 KB - 10 MB	1-10 ns	$1000
DRAM	10 GB	80 ns	$10
Flash	100 GB	100 μs	$1
Hard disk	1 TB	10 ms	$0.10

As we go down the table, the cost per unit of memory gets lower, and the supported capacity goes up, but the latency also goes up significantly as a result. These different tradeoffs mean that we'll often use different technologies for different applications based on how we want to access our memory! The memory type we're probably most familiar with here is the **register**. Registers are very fast, but we can only have a small number of them, or else we wouldn't be able to support the fast latency (on the order of picoseconds). And registers are actually very expensive – even though

they're part of our processor datapath for accessing current ALU data, we can't use them for more than that. So we'll need other means of storing data, and so far we've just been calling this "main memory." Looking down the rows of our table, **SRAM** and **DRAM** (static/dynamic random access memory) start to take us down the **hierarchy path**, which we'll hear a bit more about later. SRAM has a latency around 1-10 nanoseconds, but the supported capacity is much higher than that of registers. But the cost is still significant enough for us to primarily use DRAM for most of our memory (a few gigabytes). Looking past our SRAM and DRAM, we have the **secondary storage** – flash and hard disk – which are **non-volatile**, meaning that they can retain their data even when they're powered off (unlike SRAM and DRAM). We use things like a hard disk when we need huge amounts of capacity – they're good for use in I/O (input/output) subsystems as long as we don't need to frequently access the data, or for keeping track of information that we don't want to lose.

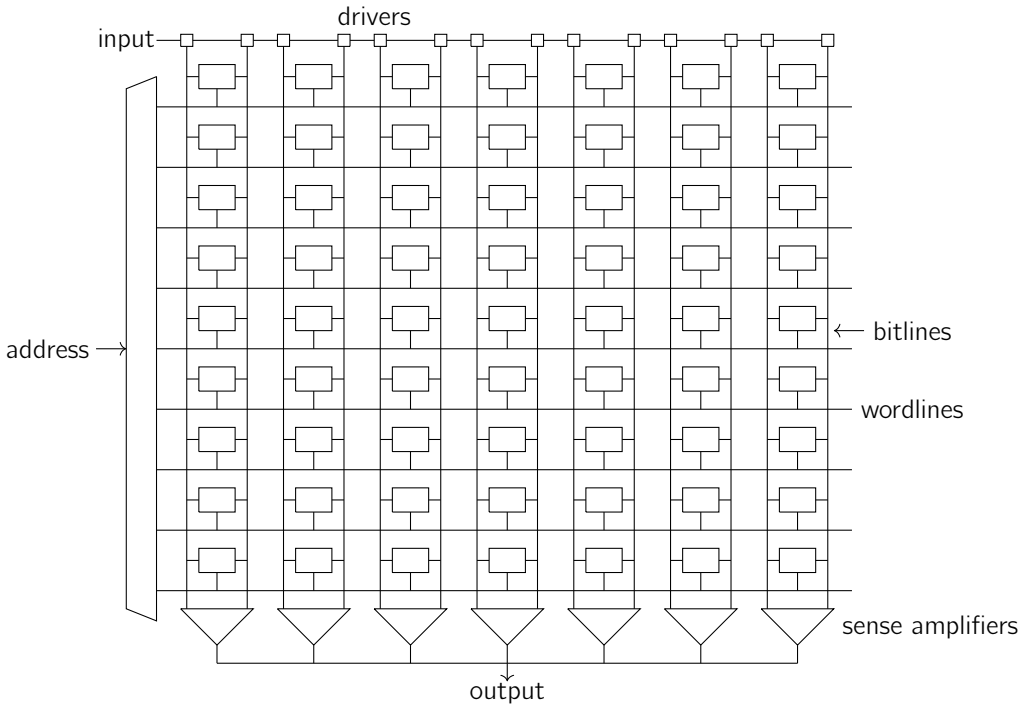
The whole point is that we determine which type of memory we use based on how the data it stores is being used, but first let's get into some of the physical details so that we can understand the features of the tradeoff table.

Fact 129

This next part of the lecture is not quizzed, but it's helpful for understanding how everything comes together.

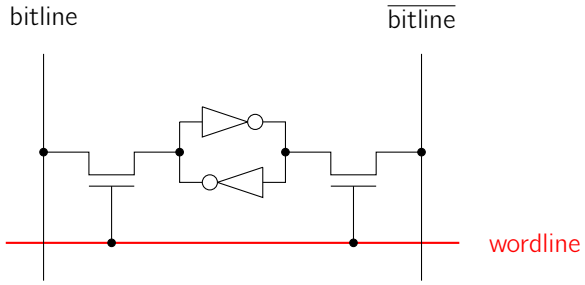
We'll just go down the rows of our table, seeing what each type of memory looks like:

1. SRAM is basically a two-dimensional array of cells, where each SRAM cell stores one bit of data. SRAM memory can be accessed by specifying the address (corresponding to one of the horizontal **wordlines** of the grid), which enables a given row of memory (but not any of the other ons). We also have vertical **bitlines**, which allow us to access the data in each cell. Here's an example of an array of eight 6-bit words:



Data comes out of one end of the SRAM. At the output end of each vertical bitline, we have **sense amplifiers**, helping to detect small changes and amplifying them so that outputs are provided more quickly. Finally, data is fed in through inputs from the **drivers** (on the opposite side of the sense amplifiers), and we can write to our SRAM by setting bitlines to appropriate voltages.

To understand this more fundamentally, let's zoom in on a single SRAM cell so we can understand how it works:



The first thing we may notice is that there are two CMOS inverters connected back-to-back, forming a **bistable element** which can retain its value. (At any time, either the left side is at a voltage of V_{DD} and the right side is at ground voltage, corresponding to a 1 bit, or vice versa, corresponding to a 0 bit.). And in order to access those data values, we use two nFETs as access transistors, which connect the bitlines to the actual SRAM data. (Adjusting these bitlines can then force the correct value into the SRAM cell.)

First of all, if we want to perform a read on our SRAM, first we **precharge all bitlines** (both the bitline bar and the $\overline{\text{bitline}}$ bar) to V_{DD}, corresponding to a 1, and then we leave those drivers floating. (There's no guarantee that they'll stay at V_{DD} once they're floating, but at least they've started off at that value.) Remembering that reads take in an input, the next step is for our decoder to select one of the wordlines, **setting the wordline (red) wire** to 1. This then completes the connection of our two access FETs, and now we should think about what happens to our voltages. For example, if the left side of the bistable element stored a 1, nothing happens when the left access FET connection is completed. But then the right side of the bistable element stores a 0, meaning that the voltage on the right side will slowly discharge.

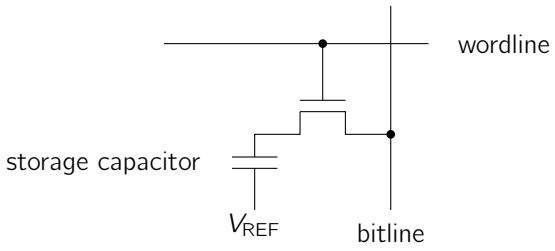
Either way, during an SRAM read, one of the bitlines holds the value 1, while the other slowly goes down to the ground voltage. And we **use the sense amplifiers** here to quickly detect small changes in the bitline (so the bitline doesn't need to go all the way to 0 before we can output the value of the read, which is determined by **which of the two bitlines discharges**). The reason we need to do this is that the discharging process is very slow, since there is a **high capacitance** for the bitlines (compared to the SRAM cell).

On the other hand, if we want to do an **SRAM write**, we start by driving our bitlines again, but this time we **hold the bitlines** in a specific way: we drive the left and right bitlines to V_{DD} and ground, respectively, if we're trying to write a 1, and vice versa if we're trying to write a 0. Once we activate the wordline, again this will turn on the access FETs, and now the SRAM cells will get overpowered by the drivers because the latter is far stronger.

Remark 130. *In making all of this work, we have to be very careful with the design – driving the bitline to ground voltage should overpower the V_{DD} in the cell (so that writing works), but driving the bitline to V_{DD} should not overpower the ground voltage in the cell (so that reading works).*

Recall that our register files have multiple ports – in a single clock cycle, we want to be able to read values from two registers and write to another one. If we wanted to do the same for our SRAMs, we could try to add an additional port like the one on the left side of the diagram, which would then need to connect to its own set of bitlines, sense amplifiers, and drivers. So the circuit becomes even more complicated (basically duplicating the whole thing). The point is that when we have N total ports, we have to use N wordlines, $2N$ bitlines, and $2N$ access FETs. And it turns out that the wires cause the biggest problem for us, because we end up having area $O(N^2)$ (lots of space!), meaning we need to be careful if we try to support multiple ports.

2. SRAMs require **six total transistors** (one for each inverter, plus the two access FETs). We want to know if we can do better, and the answer is yes:



A DRAM (dynamic RAM) cell only needs to use a **single transistor** as an access FET and a single bitline. We store the value of 1 or 0 using a storage capacitor – discharged represents a 0, while charged represents a 1. Capacitors can store and maintain charge better if the capacitance is higher, so often we use **trench capacitors** that go deep into the substrate of the chip. This way, we get a large area (so large capacitance) for storing charge, but we do not sacrifice surface area on the chip.

At a high level, we should understand that **we save lots of space** by using less transistors and building deep into the substrate – in fact, the area is about 20 times smaller than for an SRAM cell, and production is cheaper as well. But **charge leaks** from a capacitor over time, so every few milliseconds, we have to refresh the cell.

DRAM **writes** are pretty straightforward – we drive our bitline to the value (high or low) that we're trying to write into the cell, and then we turn on the access FET by activating our wordline. Our capacitor then either gets charged or discharged (based on whether the input was a 1 or a 0). On the other hand, **reads** are done by first precharging our bitline to $\frac{1}{2}V_{DD}$ (halfway between ground and V_{DD}), and then we activate our wordline. The point is that we've then turned our transistor into a wire – charge between the bitline and capacitor must be shared, and as we mentioned, the bitlines are designed to have a much higher capacitance than that of a single cell. During the sharing, this means that our bitline will either gain or lose a little bit of voltage (based on whether the capacitor was charged or discharged), and then our sense amplifiers can detect that change and give us an output.

But now we need to notice a key difference: whenever we connect the bitline with the access FET whenever we do a read, **we discharge our capacitor, meaning that reads are destructive**! Because the capacitance on our bitline is so much larger, it'll naturally suck up all of the charge, so we must remember to **write the value that we just read** each time.

3. Our next two types of storage are the **non-volatile file storage** options, which can maintain values even when powered off. First of all, **flash storage** is even denser than a DRAM – we can actually store multiple bits inside of a single transistor with this method. The idea is that we add some **floating gates** in the middle of our transistor. Such gates are very well-insulated conductors that hold charge for years, so if we're just storing a single bit, we can say that the value is 0 if there is no charge and 1 if there is some charge. In order to tell which case we're in, we **apply a voltage** on the gates – we know the **normal** amount of voltage that is needed to connect a channel between the source and drain, and if that channel isn't formed when we try to apply this voltage, we must have some charge stored inside the floating gate that's preventing the channel from forming, and we have a 1. (And of course, if the channel forms as normal, then there is a 0.)

We then expand on this idea to represent multiple bits by setting **various threshold voltages**: for example, needing only 1 volt for the channel means there's nothing in the gate, which could represent a 0. Needing 2 volts

means there's a little bit of charge in the gate, which could represent a 1, and needing 3 volts could represent a 2. And this sort of idea allows us to store multiple bits inside of a single transistor!

Of course, there are always tradeoffs, and in this case the issue is that reading and writing is extremely slow. We're dealing with things on the order of microseconds, especially because writes mean we need to supply a huge amount of voltage just to get through the insulating gate. In fact, these high voltages often damage the transistor, so there's only a limited number of times that we can write to a single flash cell.

4. Finally, a **hard disk** uses a bunch of rotating magnetic platters, where we store values based on how the platter is **magnetized**. Hard disks have a head, which can read the direction of magnetization at any given point, so we can detect what value that point is supposed to represent. These storage methods have enormous capacities, but the mechanical movement is slow (we have to move the head in or out to the desired radius, and then we need to rotate the platter until we line up at the exact location!), so we're talking about latency on the order of milliseconds.

   One thing to note for hard disks because of this is that **accessing locations close to each other in memory is much better than random locations**. This is because we only need to read in a single sweep rather than jumping everywhere, and the difference is extremely noticeable: about 100 MB/s for a sequential read/write, compared to about 100 KB/s for a random read/write.

With that, we should have a good understanding of what makes some types of technology faster or denser than others, and therefore why they're used differently.

Fact 131

The rest of this lecture is examinable, and it's about using the memory hierarchy for optimal performance of our memory system.

If we go back to the table from the beginning of class, we'll notice that we go up in capacity, up in latency, and down in cost as we read down the table. But there's no best choice for everything – nothing's both fast and large, so we should use these memory technologies together to make our memory **seem like** it's getting the best of both worlds.

Proposition 132

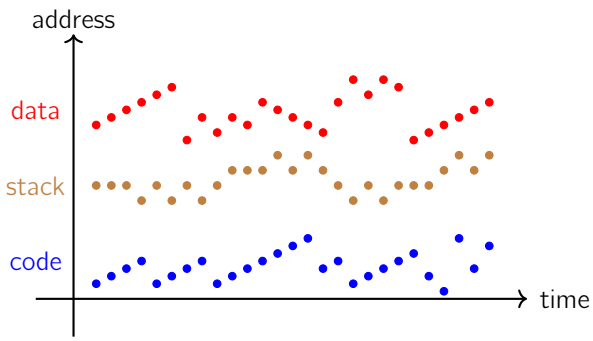
The **memory hierarchy** is the idea of using faster technologies for memories closer to the CPU, and using slower but larger-capacity technologies as we get farther away. Designed properly, this allows our CPU to look like it has a single fast memory!

There are two main choices for how to do this. The first is to **expose the hierarchy** to the processor, so that we have a small 10 KB SRAM, larger 10 MB SRAM, a 10 GB DRAM, and a 1 TB SSD (secondary storage device), each of which are available to be used as different storage alternatives at any time. So we can tell the programmer to use these memories cleverly, putting the most commonly used memory in the small ones so that retrieval is faster.

But that's a lot of work for the programmer, and what we do instead is a second method: **hide the hierarchy** by thinking of reads and writes as being done to a single memory. Then in hardware, we handle the hierarchy and determine where things should go to maximize performance by **caching**. The idea is to take an address X that we want to look up, and we first check if it's in the first-level cache. If it is, then we're done, and otherwise, it'll look at the next level of memory. If our data can be found in the second-level cache, we're done again; otherwise, it goes to the next level where we can store even more data. So the amount of time it'll take for the CPU to get the value

at the address X can be different depending on where the value actually is stored, but the requests look the same no matter what.

We've mentioned that we store data, our stack, and our code in memory, so it's useful to see what a typical access pattern looks like for each of these:



Let's go through each of these. We often access adjacent memory instructions because we run through instructions in sequence, but there is also some predictability in address access, because we often repeatedly access a set of instructions in loops. Meanwhile, for reads and writes to the stack, we see that we often do local variable accesses, and we do some shifting of the stack pointer during procedure calls that give us the diagonal lines. Finally, data is often accessed in a linear loop behavior, too (e.g. when storing arrays), so we should try to take advantage of that. All three types of memory queries have a predictable pattern, and we can summarize what we've observed above in the following two principles:

- **Temporal locality.** If a location has been accessed recently, it's likely we'll use it again soon.
- **Spatial locality.** If a location is accessed, it's likely we'll use nearby locations soon.

We should try to use both of these to our benefit, so that we keep the correct pieces of data in our fast memory and move things we don't need to the slow memory. This requires us to add a small component between our CPU pand main memory, as we've been alluding to:

Definition 133

A **cache** is a small storage component that transparently retains data from recently accessed locations.

Basically, all of our data is always in main memory, but we take some subset of that data and put a copy in our cache too. Whenever the CPU asks for a particular address, the processor will send those requests to the cache. A **cache hit** means that the address is indeed found in the cache – then, because we're using a fast technology for this small storage unit, we'll get our data returned quickly. Otherwise, we get a **cache miss**, meaning we need to fetch data from memory and send it back to the processor, and then put a copy of this data we've just grabbed into the cache.

And it's possible and typical to do multiple levels of caching, forming a **hierarchy of memories**: registers are the first level, managed by the compiler directly. After that, our hardware often has different SRAM caches living on the chip, and then if we need more storage, we can use other technologies (DRAM, flash drives, and hard disks). These technologies are often managed by the software / operating system (because they're slow enough that there's enough time to have software do computations for us). Tomorrow, we'll focus some more on these caches, and next week we'll examine how we handle our main and secondary storage memory.

For now, we'll evaluate this strategy numerically and see what kind of performance we need to get for caching makes sense.

Definition 134

Suppose a cache has h hits and m misses. Define the **hit ratio** HR and **miss ratio** MR via

$$\text{HR} = \frac{h}{h+m}, \quad \text{MR} = \frac{m}{h+m}.$$

Notice that $\text{HR} + \text{MR} = 1$.

To actually measure how well our cache is doing, we look at the **average memory access time** given by the following equation:

$$\text{AMAT} = (\text{Hit time}) + (\text{Miss ratio}) \cdot (\text{Miss penalty}),$$

where the **miss penalty** is how long it takes to fetch data from our main memory. And we can look at this recursively if we have different levels of caching, too: we just replace the miss penalty with the AMAT of the next level. Regardless, the point of caching is generally to improve this AMAT value.

Example 135

Suppose it takes 100 cycles on average to grab data from main memory, but it takes only 4 cycles to grab from our cache.

To break even, we only need

$$100 = 4 + (1 - \text{HR}) \cdot 100 \implies \text{HR} = 4\%.$$

Achieving this ratio is very easy – in fact, what we'll see is that we can usually achieve hit ratios in the 90% range. So that means that our AMAT can often be close to the latency of our cache – for example, we can have an AMAT of 5 cycles if we have a hit ratio of 99%, which is far better than 100 cycles (which is what we get if we just use the main memory). Fundamentally, the point is that we get the impression of a "very large, but fast memory."

17 October 29, 2020

We'll continue our exploration of the memory hierarchy today, getting into more discussion of how caches work and how to use them to improve performance of our memories. As a reminder, Quiz 2 will be held on Tuesday, November 10, covering material from lecture 8 (combinational circuits) up until lecture 16 (this one), and the review session will be the Monday night before it. (Practice quizzes should be posted by this Saturday.)

First of all, a quick summary of last lecture: ideally, we want a large, fast, and cheap memory for computation, but the various memory technologies each have their own benefits. So the idea is to use a hierarchy of memories (going from small to large as we move away from the CPU), so that we can fake having a large, fast memory. The hardware's job is then to store data in the fast or slow memory, based on how often it is being used, and a typical setup might have a 100 KB SRAM cache, a 10 GB DRAM main memory, and a 1 TB hard disk "swap space." We'll focus on everything within our main memory for today and save the secondary storage for later on.

The point of our **cache** is that it sits between our CPU and main memory, saving data that we recently used so we can access it again. Our processor needs to deal with **variable memory access times** when sending accesses to the cache (that is, grabbing data). This is because sometimes we'll get a cache hit, since the data is already in the cache, but sometimes we'll get a cache miss, meaning that we need to fetch from main memory and put the data in the cache as well. And the processor doesn't know how this works behind the scenes, so we need to adjust processing accordingly. The principles of **temporal and spatial locality** are what make this whole system work at the end of the

day: knowing that we often access sequential locations in memory, or loop back to a previous request, mean that we can design our caching accordingly to minimize the average memory access time (AMAT).

So let's get into the cache algorithms and design.

Proposition 136

Caches store both the **data** we're looking up and a **tag** associated with that data.

At the most basic level, whenever we're asking for some data `Mem[X]` from our memory, we look up the tag `X`, and if that's among the cache tags, our tag will be equal to `Tag(i)` for some cache line i. Then we can return the data stored in our cache at line `i`, and we're done. But otherwise, we'll have a cache miss, and we'll need to return `Mem[X]` from our main memory instead. After that, we'll select a line `k` in our cache to hold `Mem[X]`, and then we'll set `Tag(k)` to `X` and `Data(k)` to `Mem[X]` (so that it's cached for the future).

But there's a lot of design considerations that can influence the performance of our cache, so we'll think about some different tradeoffs to improve efficiency.

Problem 137

When we're looking for a particular address, how can we search the cache efficiently (faster than just going through sequentially)?

There are a variety of techniques we can use to optimize this, but the simplest cache is called a **direct-mapped cache**. Direct-mapped caches are easily to look up, because **each address can only correspond to a specific cache line**. Basically, if we have 2^W lines in our cache, we index an address into the cache using W **index bits**. Once we've selected that line, we can check an attached **valid bit** (to see if data has indeed been cached there) and see if the tag matches the tag we're looking for.

Let's be a bit more concrete: we know that we request data with a **32-bit address**. So for example, suppose we have an **8-line direct-mapped cache**. We always call the **bottom 2 bits** of the address our **byte offset bits** – we don't use them, and they're always set to 0 because we're requesting byte addresses but always grabbing words. Since there are $8 = 2^3$ possible lines in our cache, the **next 3 bits** serve as our index bits. Then the **remaining** $32 - 3 - 2 = 27$ **bits** can serve as a tag.

So to check whether we have a cache hit, the valid bit must be 1 (we add this because at the beginning of our process, the cache holds garbage that we don't want to accept). Then we compare the first 27 bits of our string to the 27 bits of the tag – if those also match, we have a cache hit, and then we can return the data in that cached line.

Example 138

Suppose we have a 64-line direct-mapped cache as shown below. The V column represents the valid bit.

	V	Tag (24 bits)	Data (32 bits)
line 0	1	0x58	0xDEADBEEF
line 1	0	0x58	0x00000000
line 2	1	0x58	0x00000007
line 3	1	0x40	0x42424242
⋮	⋮	⋮	⋮

If we're supposed to read `Mem[0x400C]` from our cache in this setup, we first convert into binary, and then we should reserve the last two bits for the byte offset, the next six for our index (since $2^6 = 64$), and the rest for our tag.

In binary, the number looks like

$$\texttt{0x400C} = \texttt{0b1000 0000 0000 1100}.$$

This corresponds to a tag of 0x40 from the first eight bits, an index of 0x3 from the next six, and a byte offset of 0x0 from the last two (notice that because our memory addresses are always a multiple of 4, this will always happen). And now, line 0x3 of the cache indeed has a valid bit of 1, and there's a matching tag in our cache. So this gives us a **cache hit**, and we return 0x42424242.

Remark 139. *Notice that 0x400C only has 16 bits instead of the full 32, which is why our tag only seems to have 8 bits instead of 24. This is just for notational simplicity.*

On the other hand, if we're supposed to read `Mem[0x4008]` from our cache, we have

$$\texttt{0x4008} = \texttt{0b1000 0000 0000 1000}.$$

Repeating the previous calculation now says that the index is 0x2. But the tag is still 0x40, while the tag listed in line 2 of the cache is 0x58. So this would be an example of a **cache miss**, and we'd have to do more work.

Fact 140

Notice that if we only store the tag, rather than the entire address, we can deduce the rest of the address bits from the current index of the cache! (For example, accessing line 1 means that our address ends in 0x0000 0100.) So in this case, we only need to store the first 24 bits of the address in our cache, which is better because it saves space and also time.

Remark 141. *We choose the* **lowest order bits** *for indexing into the cache, because we often access data in consecutive memory locations. So if we used the highest order bits, everything stored adjacently would clash, and we would have many more conflicts. So we're taking advantage of locality to make a useful optimization here!*

Our next consideration for cache design expands the cache in the "horizontal direction," fully taking advantage of spatial locality. When we get a cache miss and we need to bring something in from main memory, it makes sense to bring in a few words at a time, so that we can use the other words later on and preemptively turn them into cache hits. So the idea is to use a larger **block size**: we grab multiple words from memory at a time, and this means that we only need one tag for multiple words, further saving (tag) memory. Unfortunately, this does mean that we have fewer indices in the cache if we have a fixed total amount of space, and this could (with a large enough block size) create even more collisions.

Example 142

Suppose we want to have a 4-block, 16-word direct-mapped cache. This cache will have 4 rows, each with a valid bit, a tag, and 4 words, as shown below:

	V	Tag (24 bits)	Data (4 words = 16 bytes)			
line 0						
line 1						
line 2						
line 3						

So now if we are fed a 32-bit address, we still make the bottom two offset bits 00, and then the next two from the bottom are used as **block offset bits** (to distinguish between one of the four words in a given line). Since we have

four different cache lines, we'll need another 2 bits to choose the line index, and the remaining top $32-2-2-2=26$ bits can be used for the tag.

A natural question is to ask **how large to make these block sizes** in general. We know that any size larger than 1 helps with spatial locality, but needing to fetch multiple blocks from main memory and bring them into cache makes the **miss penalty** a little larger (because transferring large data blocks takes time). And because we have less total lines, eventually a large block size will increase the average hit time and miss ratio. In summary, as the block size increases, the miss penalty increases by a bit, though the cost of accessing memory in the first place is rather significant compared to the additional cost of accessing a larger block. On the other hand, if we try plotting the miss ratio as a function of block size, the value decreases at first when we increase the block size (taking advantage of spatial locality), but going too far means we get too many misses due to **cache conflicts**. If we measure AMAT as a function of block size, empirically the best choice turns out to be about **64 bytes** (16 word blocks) for typical caches.

Example 143

Let's talk a bit more about this cache conflict idea with some concrete examples. Suppose that we have a loop of code consisting of three instructions, and say that they live at **word addresses** 1024, 1025, and 1026. Furthermore, they're trying to access word addresses 37, 38, and 39 in our memory, respectively.

Then a 1024-line direct-mapped cache with one word per line will have words 1024, 1025, 1026 mapping to cache lines 0, 1, 2, and our data will map into cache lines 37, 38, and 39. So if we're repeating this loop over and over, after the first full loop, we'll have all instructions and data in our cache, and we'll get a hit every single time.

But if we instead run this same piece of code, but with data living at word addresses 2048, 2049, and 2050, notice that our data also maps into cache lines 0, 1, and 2! So we'll actually **keep getting cache misses all the time**: we overwrite things in our cache during every instruction and every data access. This is called a **conflict miss**, and we need a way to deal with it. (The point is that the direct-mapped cache is too restrictive if every address can only fit in a single location.)

So let's improve this by adding some more flexibility to our address locations. The opposite extreme is a **fully-associative cache**, where any address can go in any location. That's great because it gets rid of all conflict misses, but it's also very expensive because we need to search every single line in our cache to see if we have a hit or not, meaning we have lots of comparators working in parallel (we need to check every single cache line to see if the tags match the tag coming in from our address), which can have a high cost. Here's a middle ground instead:

Definition 144

An **N-way set-associative cache** is basically N direct-mapped caches working in parallel. Each one of these caches is called a **way**, and what this means is that any given address has N different locations – those locations form what is called a **set**.

So the index that we extract from our address still selects a line of our table, but that row corresponds to a cache line in way 1, as well as a cache line in way 2, and so on. So a 4-way set-associative cache can store four different memory locations that were originally supposed to map to the same index, and this dramatically decreases our conflict misses without needing a comparator for every single cache line (we only need four of them)! So we can think of a fully-associative cache as an extreme case of this N-way set-associative cache, where we have only one set and as many ways as cache lines.)

So now we can think about some additional tradeoffs between these three types of caches: associativity gives us some choices about **which location in the cache we want to store a given address to** (because there are N choices

for the N-way set-associative cache, and in fact all spots are valid choices for the fully associative cache). In other words, we'll need to make choices about **which value to remove when we take something out of the cache**, and this is called a **replacement policy**.

Ideally, if we knew the future, we would remove the data that is accessed **furthest in the future** from now. But we don't know the future, so instead, we'll make another locality argument: if a line hasn't been recently used, it's less likely to be used in the near future than a line that was recently used. This is called an **LRU** (Least Recently Used) policy, and it works well and is commonly used in practice. It's easy to implement this for a 2-way associative cache (just keep track of which of the two has been more recently used), but there are some complications – in order to implement LRU for an N-way associative cache, we need an ordered list of all past accesses to know what has been most recently used, which means we have $N!$ possible orderings. This requires $O(\log_2(N!)) = O(N \log N)$ LRU bits, as well as some complex logic to actually implement. So a lot of caches for modern processors use cheaper approximations of LRU, not fully keeping track of all of the ordering of the N items.

Remark 145. *Other replacement policies include a first-in, first-out (least recently replaced) or random (good for avoiding adversarial access patterns) strategy, but LRU has worked well in general and is still found in most caches.*

Until now, we've been talking only about reads from our cache, but now let's focus on **writes to the cache** (like "store word" operations). Notice that when we write, we also have choices: first of all, we can keep our cache in sync with the main memory, which is called a **write-through** policy. This means that we always stall our CPU until we write the value from cache back into main memory. But this is slow and may waste bandwidth (because we're writing values that we might overwrite later anyway), so we usually use a **write-back** approach instead. The idea is that **we only update a line in main memory when we have to replace that line in the cache** (because as long as it's in the cache, we'll be grabbing from there rather than from main memory anyway).

Example 146

Suppose we have a 16-line **direct-mapped** cache with block size 4, and we want to write `0x09` to `0x4818`.

With a write-back policy, we have to add an additional bit to our cache to tell us whether or not we need to write back to main memory whenever we do a replacement. The point is that if we only read from a line in the cache, there's no reason to write it back, either (since nothing was supposed to be modified):

	V	D	Tag (24 bits)	Data (4 words = 16 bytes)			
line 0	1	0					
line 1	1	0	0x48	0x01	0x02	0x03	0x04
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
line 15	0	0					

The D bit here is called a **dirty bit**, and it's a state bit set to 1 if we've written to that cache line since bringing it in. (So we only need to write if we are replacing a line that already has 1 set for the dirty bit.) We have 2 offset bits (as usual), 4 index bits (for the 16 lines), and 2 block offset bits (for the block size of 4), which explains the above tag size of $32 - 4 - 2 - 2 = 24$ bits. First,

$$\texttt{0x4818 = 0b0100 1000 0001 1000},$$

meaning the tag is `0x48`, the index is `0x1`, the block offset tells us to look at (zero-indexed) way 2, and the byte offset is 0 as always. Checking at index 1, we have a hit, because the valid bit is 1 and the tags match. So now we can

update the correct word, and we want to write to word 2 (meaning we modify the 0x03 to a 0x09). And now we set our dirty bit to 1, so that whenever we next replace something in this cache line, we need to copy the line to main memory first.

But suppose that we have a **cache miss** instead: say that index 1 actually has the incorrect tag 0x280 instead. That means that we need to replace this line in the cache with the new one we're trying to write, and we do this by **first checking the dirty bit**. If it is 1, we have to write that line (a tag of 0x280 means addresses 0x28010 through 0x2810C, by looking at the index bits and block offset bits) back to main memory, and then we can write to the cache. But if it is 0, we can just replace it without a problem (there's no deviation issues from main memory). So once we deal with the old data, we always need to load the whole line (because of our tag 0x48 from our address, we need to load 0x4810 through 0x481C) into the cache, and then we finally set the dirty bit to 0 and write our new value in.

In the last few minutes of lecture, we'll take a look at a few examples to show that no configuration is always the best one (so it's worth keeping all three in mind):

Example 147

Consider three caches: a direct-mapped (DM), 2-way associative (2-way), and fully associative (FA) caches, each with 8 words and block size of 1, using the LRU replacement policy. First of all, suppose we are repeatedly accessing the addresses 0x0, 0x10, 0x4, and 0x24.

First, we figure out which cache line each address maps to for each cache. The third through fifth most significant bits tell us the index for the DM cache, while the third and fourth tell us the index for the 2-way cache.

- First, 0x0 = 0b0000 0000 has DM index 000 and 2-way index 00. (It can go anywhere in the FA cache, so there is no set index.)
- Next, 0x10 = 0b0001 0000 has DM index 100 and 2-way index 00. (Remember that the 2-way cache can store two values in the 0 index, so this is still fine.)
- Fetching address 0x4 = 0b0000 0100 yields a DM index of 001 and a 2-way index of 01.
- Finally, 0x24 = 0b0010 0100 gives us a DM index of 001 and a 2-way index of 01.

And now we can try running through the addresses in cycle. Repeated access tells us that location 0x4 and 0x24 conflict and therefore always overwrite each other in the DM cache, so we get a 50 percent hit rate. But the other two caches give us a 100 percent hit rate.

Example 148

Next, let's say we want to repeatedly access 0x0, 0x4, 0x8, 0xC, 0x10, 0x14, 0x18, 0x1C, and 0x20 in a loop.

This time, we'll split up the analysis by cache type instead of address:

- In the DM cache, we find that locations 0x0 and 0x20 map to each other, but nothing else conflicts. So those two locations will give us cache misses, and thus our hit ratio is 7/9.
- The same analysis on our 2-way cache makes 0x0, 0x20, and 0x10 conflict with each other, but we only have 2 possible places for them to go. So those three will always be overwritten, and our performance in this case will be a hit rate of 6/9.
- Finally, the fully associative cache does a really bad job, because the cache only has room for 8 values but keeps trying to access 9. So this is called a **capacity miss** – each value kicks out something we will soon need, and our hit rate is 0.

Example 149

The access pattern 0x0, 0x4, 0x8, 0xC, 0x20, 0x24, 0x28, 0x2C, 0x10 gives us hit rates of 1/9, 6/9, and 0 for the three types of caches, respectively (we can check this for ourselves).

The key takeaway here is that there are different levers that we play with in designing caches, and depending on the application, different cache configurations will work best for us. Lab 7 will give us more practice with this!

18 November 3, 2020

Because today is Election Day, attendance for this lecture and recitation are not required – in fact, tomorrow's recitation will take on a different format where we have an hour-long demo and see the moving pieces of an operating system. (And the Lab 5 checkoff and lab 6 are also now due on Friday to give us more time.)

Today, we'll have another big change of topic, discussing **operating systems** (another key part of computer systems), focusing today on virtual machines and exceptions. There are entire courses devoted to this topic, but this is just an introduction where we focus on interactions between hardware and software, understanding how to use the concepts to make hardware run efficiently.

So far in 6.004, we've seen **single-user** or **single-program** machines, where our hardware runs a single program with complete control over and access to **all resources** in the machine (like the processor and memory, as well as disk, network card, display, keyboard, and other **I/O devices**). Then the only existing "contract" we have between hardware and software is our instruction set architecture (ISA), such as RISC-V, and we code our program **directly** to the ISA. But most systems don't work like this: only simple, single-purpose systems do, like a chip that controls a dishwasher. In fact, we know from our everyday use of computers that we have multiple executing programs that don't have access to resources directly, and they work with higher-level abstractions like files instead of directly manipulating at the hardware level.

Instead, the idea is to have an **operating system (OS)** that controls our programs and how they can share hardware resources. So now we'll need two different interfaces: the ISA still controls interactions between the hardware and the OS, while the OS interfaces with other programs with an **application binary interface (ABI)**. Let's make sure we're being precise enough with our language:

Definition 150

A **program** is a collection of instructions (in other words, the code that we write), while a **process** is an instance of a program being run (that is, it includes the code, but also the state of our registers, memory, and so on).

This is the distinction between an "instance" of an object and the object itself, similar to the difference between a function and a running instance of that function, or between a class and an object of that class. Also, there's a **particular part** of the operating system which has the privilege to interact with our hardware, because we usually mean Windows or MacOS or Linux – the set of software installed in a machine to make it useful – when we say "operating system:"

Definition 151

The **OS Kernel** is a process with special privileges (compared to other ordinary programs that also come with a typical operating system, like text editors).

There are **three main reasons** that we have this special process. First of all, we need **protection and privacy** from malicious or buggy processes: we do not want processes to access or read each other's data and memory, especially if there are crashes or other issues. (It's better if each process believes it's working on its own machine.) Second, we get the benefit of **abstraction**: the operating system can hide details of our underlying hardware, so processes can work with the various devices (for example, a file system to open and access files) abstractly, rather than needing to know how exactly to access a hard drive or other devices directly (especially since this direct access can be dangerous)! Finally, our OS can help us with **resource management**, assigning computing power to important processes, and making sure nothing takes up too much CPU usage or memory.

Let's talk in more detail about the various mechanisms that are at play here:

Proposition 152

The OS kernel provides a **private address space** (next lecture, we'll see that this is a **virtual memory**) for each process, only allowing each one can only access its own allocated space.

Basically, in different regions of our physical memory, we assign space for each of our running processes. For example, some memory will be dedicated to the OS kernel, and some other memory will be dedicated to every other running process. But it's key that from the point of view of any program, **its assigned memory is the only memory that exists**: we translate addresses, so that address 0 refers to the beginning of the process's chunk of memory rather than the absolute address 0. (We'll learn more of the details next time.)

Proposition 153

The OS kernel schedules processes into the CPU, giving each one a time slice before moving on to the next, so that each process can only use a certain amount of CPU time.

When we do this, we don't rely on programs to tell us that they're done – instead, we step in and give the processing power to another process, and this is called **preemptive multitasking**.

Example 154

Modern computers have multiple CPU cores and can run multiple programs at a time, but at any time, there are many more processes than available CPUs. If we look at the state of our system (e.g. by running the Linux `top` command), we can see that there's sometimes thousands of tasks – some are running, but almost all of them are sleeping. We need to make sure to keep this in mind to assign CPU time appropriately.

The scheduling process works as follows: one process runs for some fixed amount of time, then the OS interrupts it and schedules some other process. That new process that gets interrupted after some amount of time, and so on. So we can think of this as running a full CPU, but slower (because it takes more time for calculations to get done).

Proposition 155

For everything that is not memory or CPU, the OS exposes an interface for processes to invoke system services using **system calls** (function calls into the operating system). In other words, certain functions can help us open a file, or open a network connection into another machine.

If we think about the three propositions above, we need to respectively virtualize memory, the CPU, and all the external devices. It therefore makes sense to think of these mechanisms collectively as a **virtual machine** (the whole

interface between operating system and user-level applications). Basically, each process believes that it's running on its own machine, even though this machine doesn't exist in physical hardware! Instead, what's happening is that the OS kernel sets up VM interfaces (one per process), which allows the process to work with its own virtual processor and memory, as well as methods of dealing with files, events, memory, sockets, and so on. The key is that **each process currently running has its own VM**.

The VM idea can be thought of as an **emulation** of a computer system, but there are Python or Java virtual machines (language-level VMs), and there's software like VMware or VirtualBox (virtualization tools) which uses the same ideas. Basically, this idea of abstraction generalizes far past operating systems!

Example 156

Recall that in lab 2, we coded a program (quicksort), which becomes a RISC-V process when run.

This process is coded against a RISC-V ISA, so it believes that it's running on a RISC-V machine. But what it's actually running on is a **RISC-V emulator** (sim.py), which is a process that implements the RISC-V ISA. This emulator is itself a Python program, so it's coded against the Python language, which is interpreted by a **Python interpreter** (typically CPython). This Python interpreter implements a Python VM, but that's not running on real hardware, either: this is a user-level process coded against the Linux ABI, so we're running it on a **Linux OS kernel** (which implements a Linux-x86 VM).

But we have to keep going. This Linux OS kernel (the one running in our Athena machines) is coded against an x86 ISA, but if we run the Linux `lshw` command, we can see a description of the computer. We'll find some things that are suspicious (31 Gigabytes of memory, which is not a power of 2, Virtual Machine Communication Interface with vendor VMware). VMware is a company that produces virtualization software, and the point is that we have a **VMware program** that fakes a whole x86 ISA! And this is nice for flexibility because it lets us migrate or have multiple virtual machines, so we don't actually know (for example) whether the Athena servers run on the same physical machine. And now this VMware software implements an x86 system VM, emulating an entire system, but it's an application that runs against some other ABI (we have to ask IS&T for the details). So that means we have another **OS kernel** (implementing an OS-x86 VM), and then this x86 ISA lets us finally communicate with the physical hardware (Athena server). So in total, scanning through our analysis, there are **five levels of VMs** of different types (some are system level machines, but others are process or language level machines)!

So it sounds like virtual machines give us a lot of control and abstraction and protection, but we need to think about how we can actually implement them. There are two options – the first is to do them in software (like the Python interpreter implementing a Python VM with all of its abstractions), but this is expensive because we need to interpret every instruction and check all of the accesses manually, so for example Python programs are 10 to 100 times slower than native Linux programs, which is too expensive. Instead, we'll add some hardware to our processor so that we can support operating systems with minimal overhead. So **most of the time, the application should use the hardware directly**, but we should also have the right set of capabilities so that when we need to do something more or let the OS step in, there are ways to do that safely.

So there are four basic extensions we make to our ISA to help support OS capabilities:

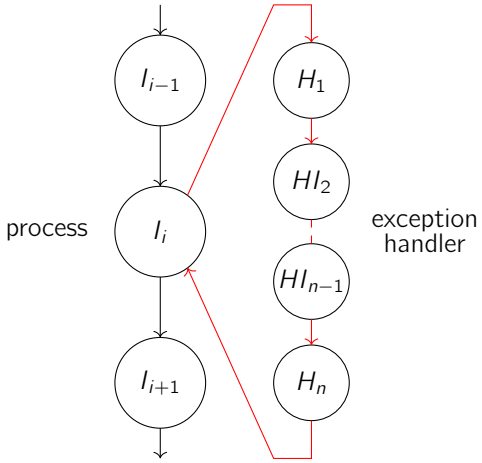
- There are two modes of execution: **user** and **supervisor** mode. The OS kernel runs in supervisor mode, while everything else runs in user mode.
- Certain states (registers) and instructions are privileged, and only supervisor mode is able to use them.
- Exceptions and interrupts (because programs misbehave or an event happens) **safely** let the OS kernel step in and transition from user mode to supervisor mode.

- Virtual memory will provide private address spaces and abstract our machine's storage resources.

On Thursday, we'll look at the last point, but for now we'll focus on the first three. And here's where a tight integration between the hardware and OS software is extremely important – the conventions need to be aligned for all of this to work.

Definition 157

An **exception** is an event that needs to be processed by an OS kernel because the application cannot handle it. Usually, this happens because the event is either **rare or unexpected**.



If a program gets an exception at some step I_i, we can think of it as an implicit call into the OS: the processor jumps to the **exception handler** code in the OS kernel to handle what needs to be done (for example, if we have a keystroke, it sends the key to the right application). Sometimes, the operating system will then terminate the process (when we have a segmentation fault or more esoteric issues), but otherwise it'll return control to the process again back at I_i.

Definition 158

The words "exception" and "interrupts" are mostly used interchangeably, but **exceptions** are usually synchronous events generated by the instructions of our program, like an illegal instruction, a system call, or a divide-by-0. Meanwhile, **interrupts** are usually asynchronous events ("doorbells") coming from an I/O device,like a keystroke, an expired timer, or a completed disk transfer.

(We'll use "exception" for both types, and we'll call synchronous events "synchronous exceptions.")

So at the processor level, we need to make some modifications to handle these exceptions. If an exception happens at instruction I_i, we must complete all instructions up to I_{i-1} but not complete I_i (not finishing the current one). (This is easy for now, but when we think about processor pipelining, we'll see why this can give us some issues when there can be multiple instructions in play at once.) We then save some information about our current state, like the program counter and the reason for the exception (and other associated data), and all of this goes in some of the special (privileged) registers that only the OS kernel can access. Finally, we enable supervisor mode, transferring control to a **pre-specified** exception handler PC. Because this location is already determined, the program cannot jump to some random place.

Remark 159. *This is a lot like the "branch" instructions we've run into earlier in the class – we're saying that every instruction now has the ability to jump somewhere else.*

Once this is all done, the process will regain control at instruction I_i again (so the exception is **transparent** to the process), unless there is an illegal operation, meaning that we'll just abort the whole process.

Let's now see three use cases for exceptions, letting us build up the idea of a virtual machine:

Example 160

CPU scheduling, limiting the CPU time of each individual process, is enabled by **timer interrupts**.

To make this work, we have a hardware device which acts like a programmable timer (for example, through MMIO). This timer is a simple device that generates an interrupt a specified amount of time (maybe 20 milliseconds) after the process has started running. So once the OS kernel hands control over to some Process 1, it sets the timer to fire (generate an interrupt) in some time, and it loads the states of Process 1 (registers, program counter, region of memory and so on) and jumps into user mode. When the timer interrupt happens, the CPU will jump to the exception handler, which will save the state of Process 1, perhaps schedule a different Process 2, and set the timer to fire in some (possibly different amount of) time. We then load the state of the new program and giving it control, and this procedure continues on. Notice that by controlling the length of the time slices, we can control the speed of our CPU for each individual process, based on priority.

Example 161

Instruction emulation can be performed by using **illegal instruction exceptions**.

In other words, we can fake instructions that don't explicitly exist in the hardware and run the functionality in software instead. For example, `mul x1, x2, x3` is an instruction in the RISC-V M **extension**, which allows us to multiply two numbers together. If we're not using the M extension, this is an illegal instruction, so the instruction's opcode will throw us an illegal instruction. It makes sense then that the code designed to run on an RV32IM machine will crash the program on an RV32I machine.

But instead, we can emulate the instruction `mul` by letting our OS kernel will get involved. In particular, once the illegal instruction exception is thrown, the kernel exception handler can realize that it can **emulate a multiply instruction in software** (by repeatedly adding, for example), and it can perform the instruction for our process. In this case, we return control to the process **after** the illegal instruction (so that we don't loop back and get another illegal instruction exception). So this way, we can make the program believe that it's executing in an RV32IM processor, even though it's running in an RV32I processor!

The main drawback is that this sequence of events is much slower than doing a simple hardware multiplication, because we're taking a lot of time transitioning control to the kernel, saving states, and going back.

Example 162

Processes can access files and invoke other system services by using **system calls**.

Remember that system calls are similar to functions, but they go into the operating system. We don't want to let programs jump into random places freely, so instead we execute special instructions that causes an exception – in the RISC-V ISA, we use `ecall` ("environment call"). There are many possible system calls – they can be used for opening and closing files or reading and writing to them, using network connections (abstracted through sockets), managing memory, getting information about the system or a given process, waiting for a notification to happen (for instance, yielding or sleeping), creating and interrupting other processes (creating trees of processes), and much more.

Note that programs usually don't invoke system calls directly. Instead, we use certain functions in low-level libraries – for example, in C, if we want to print something, we use `printf`, which internally calls `sys_write`. But the OS

manages all of that for us, so we don't need to know about it! It's important to remember that some of these system calls may block our process (for example, if it's sleeping for some time, or if we need to wait some time for a file to be read), and we unblock the process once those services are completed.

Putting everything together into a "life cycle," a process is created, gets to a ready state, and then the operating system schedules and deschedules it in and out of an executing state. Once the process is completed, it will indicate that using a system call, and it will be terminated. But system calls might block a process, so processes can also be in a "waiting" state until it is woken-up and goes back to the "ready" state. And at any time, the OS always maintains a list of all processes and what their current status looks like (ready, executing, and waiting).

In RISC-V, there's always some special privileged registers, called **control and status registers (CSRs)**, which keep track of the things we've been talking about – for example, `mepc` tracks the exception pc, `mcause` tells us the cause of the exception, and so on. We also have special instructions, `csrr` and `csrw`, that allow us to read and write to these CSRs, and an instruction to transfer control back to user level. And system calls have conventions similar to function calls, but we'll see the details in tomorrow's recitation where we'll see a tiny OS in action.

19 November 5, 2020

If we're still anxious about the results of the election, hopefully the uncertainty is resolving as answers become more clear.

Today, we'll continue studying operating systems, looking at a big piece of the hardware support needed to make operating systems fast: **virtual memory**. Quickly recapping last lecture, the point of having an operating system is to interpose between multiple processes running on a machine and the machine's hardware. This has three main goals: **(1) protection and privacy** (so that processes can't see each other's data or do something bad with the hardware resources), **(2) abstraction** (hiding away the details of the underlying hardware, so we don't need to access blocks of hard drives directly), and **(3) resource management** (controlling how different processes share the hardware). And there are a few key enabling technologies that made this possible: last time, we talked about using a combination of user and supervisor mode, along with exceptions and special permissions, to help us "share the CPU." And today, we'll get into virtual memory, which is basically how to "share the memory" between different processes.

Fact 163

The idea of virtual memory is centered around each program having the illusion of a **large, private, uniform storage**.

In terms of protection and privacy, this idea lets each process has a private address that can't be accessed by other running processes. (So the tens of gigabytes of memory can be partitioned into a few megabytes per process.) But there's also a goal of **demand paging**: at a high level, this means we **use main memory as a cache for the disk**. Basically, we have a secondary storage (flash drive or HDD) that is much larger than our main memory (usually consisting of DRAM). This type of caching allows us to run programs that are much larger than main memory, and it also helps hide differences in machine configuration (aside from differences in performance). And the main price of having these new features is needing to **translate memory references into the hardware memory addresses**, so it's good to start with some terminology for thinking about address translation.

Definition 164

Virtual addresses are addresses generated by particular processes which are specific to the private address space of that processor. Meanwhile, **physical addresses** are the addresses used to actually access hardware memory.

It's importantly the **operating system's job** to specify how to map virtual addresses into physical addresses. There are two techniques that we often use for address translation, which are **segmentation** and **paging**. Let's discuss this now:

- In **segmentation**, we give each process a specific **contiguous segment** of physical memory. We could expose the physical addresses to the program directly, but the compilation process would then become very tedious (it's easier to have the process think about the starting address as address 0 and have the operating system flexibly move our contiguous block, rather than having to add the starting address explicitly). So instead, we assign to each process a **base register** (which is **privileged**, so only the OS has control over it), and the physical address we actually use is always the **virtual address plus the segment's base register value**.

  This is called **base-and-bound** address translation, and the "bound" part has to do with not going past the allowed memory allocation. So we also need a **bound register** (which is also privileged) for each process: we check that the virtual address is in an acceptable range, and if not, we trigger a bound violation exception because the program is trying to access memory that it shouldn't.

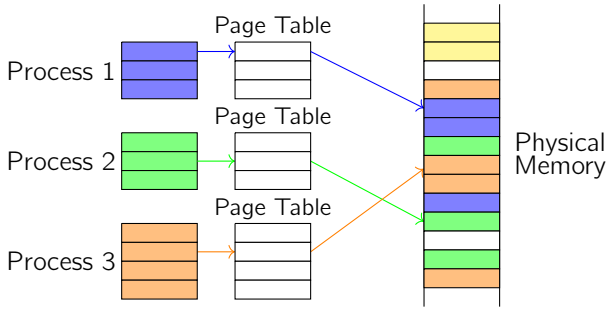
  Making this a bit more sophisticated, we can use this mechanism to make the programs a bit more robust. Basically, for each program, we'll **have separate allocations for data and code**. So for every program, we'll then have a **data segment** in main memory (along with a data bound register, a data base register, and comparing everything against a virtual address), as well as a **code segment** in main memory (along with a code bound register, a code base register, and comparing everything against the program counter). This separation is good because it's **impossible to accidentally modify our code** during an instruction (which we might have done in labs): we can only load and store words to the data segment of our program, rather than the code segment. And multiple processes can share a single code segment; if there are multiple copies of the same program running, this will save us some memory.

  This implementation is simple, but is rarely used, because there are a few problems – one of the main ones is called **memory fragmentation**. Over time, different processes will be started and stopped, meaning that we'll keep allocating and removing spaces in our memory. Over time, especially if we're allocating different sizes of memory for different processes, our free space will get fragmented into smaller pieces, and we may not be able to allocate a single contiguous chunk of space even if there is technically enough free memory at some given time! So at some point, we'll need to move storage segments around so that we can reorganize the free space, and if we want programs to be able to grow and shrink their memory over time, that can make this even more complicated.

- So let's introduce the other method, which is more complex: in **paging**, we divide our physical memory into fixed-size blocks which we call **pages**. So our main memory is divided into equally-sized (typically about 4 KB) pages, and we'll **map pages of a process into different locations of physical memory**. We do this in a physical way to solve the issues from before: specifically, we introduce the concept of a **page table**, which uses lower-order bits as an offset within a page and the higher-order bits to determine which page we're looking at. Each virtual address is now interpreted as an ordered pair **(page number, offset)**, and the page table tells us what physical page number (and therefore what starting physical address) corresponds to each virtual page number!

In particular, using a page table with 2^p possible addresses means we need p bits for the offset and the other $(32 - p)$ for the page number.

Below is a diagram, demonstrating that different processes have different page tables. The yellow pages near the top of physical memory are pages for the OS, which we'll talk about later on:



Here, we have three different processes, each with its own associated page table (boxes denote pages). Each of these pages encodes a starting address in physical memory, and if we look at the pages for the first process (for example), we'll notice that they don't need to be contiguous. If we let **VPN** refer to the **virtual page number**, this means that we can label the pages of our page table with $0, 1, 2$, and label their corresponding physical memory addresses as (1, VPN 0), (1, VPN 1), and (1, VPN 2). And even if different processes have the same VPN, that won't map to the same spot in physical memory.

We'll talk about paging in more detail soon, but we should note that introducing this more sophisticated map (of paging) **avoids fragmentation issues**. But interestingly, we can use this mechanism of mapping pages to physical memory to have main memory serve as a cache for the disk – this is the **demand paging** idea from earlier, since the paging map can just tell us "it's not in main memory, look somewhere else."

However, one big drawback is that instead of having a single base-and-bound register (64 bits of data to store locations), we need to store all of the associations in page tables, and these take up much more space (typically a few megabytes)!

Example 165

These page tables need to be stored somewhere, and let's first consider what happens if we store them in main memory.

Then we have the page tables in process 1 and 2 in some different regions of physical memory, and then we have pages for each of our processes elsewhere. (We can put them above the OS kernel page tables marked in yellow in the diagram above.)

So if we want to perform a memory access, let's think about how that can be done. We have a base register for the page table, which tells us where the page table starts for the process we're currently running. Once we do the translation and look up in our page table, we'll discover the physical page number for each virtual page number, and only then can we grab our data. So we start with a VPN (virtual page number) and offset concatenated together, and we actually need to do a two-step process: first, the **physical page number (PPN)** is located at `Mem[PT base + VPN]`, where the value of `PT base` is found by doing a look up from the kernel page table. Now that we have the PPN, we can find the physical address by computing `PPN + offset`. Then we need to do a second DRAM memory access to actually get the data word, which means we've **doubled our memory access latency** (which is bad). And every time we switch processes, we need to adjust the value of `PT base`, too.

Because of this, let's come up with a better system. In the rest of the lecture, we'll see how to use this mechanism to use main memory as a cache, and then we'll describe our page table more precisely.

Proposition 166

In **demand paging**, our page table tells us whether our data is in main memory or secondary storage, and we organize our page table in a particular way to make this happen.

(The area on the disk used to back up our DRAM is called **swap space**.) Our page table will still look like an array of entries, indexed by the virtual page number, but each entry of that array is called a **page table entry**.

Definition 167

A **page table entry (PTE)** contains the following information: it has (1) a **resident bit**, telling us whether the page exists in main memory (as opposed to disk), (2a) **PPN** (physical page number) bits telling us how to locate data that is indeed resident in main memory, OR (2b) **DPN** (disk page number) bits for a page that is not resident in main memory, and (3) **protection and usage bits** that tell us whether we can write to that row of the table, and whether it is dirty.

Sometimes pages of our processes won't completely fit in memory, but even if they do, we can bring **only what we need from the disk**. And this is also good for starting up programs quickly (instead of needing to load everything). In modern operating systems, we only load a single page of code at first, and over time, we load more and more pages.

Example 168

Suppose we have a setup with $256 = 2^8$ bytes per page, $16 = 2^4$ virtual pages, and $8 = 2^3$ physical pages. This means that our virtual addresses have $4 + 8 = 12$ bits, and our physical addresses have $3 + 8 = 11$ bits.

If we try to run a "load word" command like `lw 0x2C8(x0)`, we're being given a virtual address of `0x2C8`, and we need to find the physical address. Because we have 16 virtual pages (that is, 16 possible references to contiguous blocks in main memory), and there is one entry in our page table per virtual page, we have a 16-entry page table. Each row of this page table has a **D** (dirty), **W** (writable), and **R** (resident) bit, as well as the **PPN**. For example, if one of our rows is (D, W, R, PPN) = (0, 0, 1, 2), that means we do have a valid physical address (because R is 1) that is read-only and not dirty. Furthermore, because PPN is 2, we look in the second page of physical memory (recall that the pages of physical memory run from 0 to 7), and thus **virtual page 0 is stored in physical page 2**. On the other hand, if the resident bit were 0 instead, then we would know that virtual page 0 is not stored in main memory anywhere.

So now let's look at the specific command described above. First of all, we divide the address into VPN and offset: our virtual page number is the first four bits, `0x2`, and our offset is the last eight bits, `0xC8`. Remembering to zero-index, if looking up row 2 of our page table gives us a resident bit, then we can write down the physical page number – for sake of demonstration, say that this is page 4. That means the starting address of our physical page is `0x400` (remember that we append 8 bits for the offset), so the virtual address ends up being $\boxed{\texttt{0x4C8}}$.

So if we have data living in main memory, we know how to do the translation. But if there's a miss, we have to think harder, and the subsequent steps we take are **very similar to that of caching**. Here's some of the main differences though:

Caching	Demand paging
Cache entry	Page frame
Cache block (32 bytes)	Page (4 KB)
Miss rate 1%–20%	Miss rate <0.001%
Hit: 1 cycle	Page hit: 100 cycles
Miss: 100 cycles	Page miss: 5 million cycles

Basically, because our main memory is larger than a usual cache, we use 4 KB pages instead of 32 byte cache blocks, and the miss rates are much smaller for demand paging than for caching. This happens because the **page miss cost is extremely slow** – it takes millions of cycles to recover – which means the design of paging-based systems looks much different from the design of caches. Notice that the page table is designed such that any virtual page can be mapped to **any** location in physical memory – this is like having a fully associatives cache, and it's worth it because we want to reduce the miss rate no matter what. Also, we handle page misses mostly with software, because it'll takes many cycles to complete everything anyway, and adding a thousand cycles for software to do the job for us is not too costly.

Definition 169

Trying to access a page without a valid translation (because the resident bit is 0) gives us a **page fault exception**.

These exceptions are handled very similarly to how a cache handles a miss (except on software instead of hardware), and remember that our OS handles exceptions for our processors. We do this by first choosing a page of physical memory to replace, and if that page has dirty bit 1, then we write it back to the disk. Then, we mark the corresponding resident bit in our page table as 0, because the page is not resident in main memory anymore. Next, we read the page from our disk into the newly available physical page, and we set the resident bit of our new virtual page to 1. (So this process can edit two rows of our page table: the one originally mapping to a spot in physical memory, and the one that we're now mapping to the available spot.) Once this is completed, we return control to the program again, and this time executing the memory access will not result in an exception (because we'll get a memory access hit).

This demand paging strategy has a few performance problems, though, because we still need to perform two memory accesses. (Address translations require accessing the page table!) So the solution is that we'll **add a cache for translations**, called a **translation lookaside buffer (TLB)**. This TLB is basically a translation cache: we have status bits, a tag, and data, except that **the tag is the virtual page number, and the data is our physical page number**. And if the translation that we want is in the TLB, we get a TLB cache hit; otherwise we need to go to main memory to access the page number.

Example 170

Let's look at a concrete page table and TLB, where we have 2^{32} bytes of virtual memory, 2^{24} bytes of physical memory, a page size of 2^{10} bytes, and a 4-entry fully associative TLB shown below.

Here is the page table:

VPN	0	1	2	3	4	5	6	⋯
R	0	1	1	0	1	0	1	⋯
D	0	1	0	0	0	0	1	⋯
PPN	7	9	0	5	5	3	2	⋯

Here is the TLB (remember that the VPN is the tag, and everything else is the data):

VPN	V	R	D	PPN
0	1	0	0	7
6	1	1	1	2
1	1	1	1	9
3	1	0	0	5

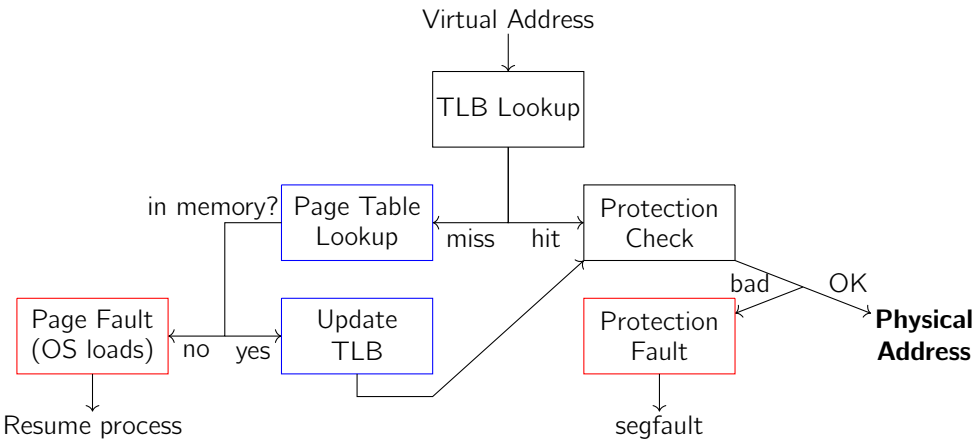
We can check a few facts for ourselves: we can store 2^{14} pages in physical memory, and there are 2^{22} entries in the page table (so there are far more VPNs than there are PPNs in physical memory). Let's find the physical address for the virtual address `0x1804`. Because the page size is 2^{10} bytes, we have to set aside 10 bits for the offset, and we write the address as `0b'000110 0000000100`, which tells us that our VPN is `0x6`. We do have an entry in the TLB with this VPN, and the corresponding data has a valid bit of 1 and also a resident bit of 1, so we get a **TLB hit**. So we need to go to the physical page number `0x2`, and thus the physical address is `0'b000010 0000000100`, which is `0x0804`. And now we know where to look in main memory, without needing to look at the page table at all!

Fact 171

TLBs usually contain a few tens of entries – if our caches are usually a few kilobytes and we get good locality at the level of cache lines, we must get good locality at the level of pages because each virtual page number references about 4 kb of data! And most modern processors have a hierarchy of different levels of TLBs to further improve performance.

There's still some discussion to have about how this TLB is implemented, though. One issue is that the TLB basically stores a part of the page table, which is supposed to be private to the processes (because it tells us about main memory directly). So every time we switch processes, we need to **flush the cache** to avoid illegal accesses. That can be expensive, so sometimes we alternatively include the process ID as a tag in each TLB entry (so translations for multiple processes can be stored in the TLB at once). Also, if we have a TLB miss, we have to look up a page table, and we can either do this in software or hardware. Basically, if the page is in memory, we need to traverse the page table (this is called a **page walk**) looking for the VPN to PPN translation to put into our TLB. If we don't find anything, we cause a page fault (using **software**), even though the page walk is done using **hardware** (with something called a "memory management unit").

Overall, here's a diagram that tells us about the entire address translation process. A black border indicates something done by hardware, a red border indicates something done by software, and a blue one indicates something that can be performed by either one.



Each time we need to do a memory reference, we start with a virtual address, extract a virtual page number, and do a TLB lookup. If it's a hit, we do a protection check (is this page actually resident, writable, etc). If we pass, then we obtain our physical address and go to memory. Otherwise, we get a protection fault and throw an exception (because the program is misbehaving). On the other hand, if we have a TLB miss, we check the page table and update the TLB (if it's in memory) or get a page fault (meaning that the operating system handles the situation by fetching the data).

We'll close with one more optimization. So far, we've looked at how to use main memory as a cache for our disk, and also how to use caches to cache main memory. But we can combine the two together, and the order we do this matters. One idea is to put the cache before the TLB, giving us a **virtually-addressed cache** (because virtual addresses come out of our CPU), and only on cache misses do we need to do a TLB lookup. Because we're caching virtual addresses, we don't need to do TLB lookups most of the time. But we do need to flush the cache whenever we switch processes, which is very expensive. So instead we can flip them, and **translate every single reference** from the processor before indexing into the cache. This is called a **physically-addressed cache**, and it allows us to context switch (between processes) without needing to flush the cache in the middle. Unfortunately, this has a larger latency because of the repeated translations.

Well, it turns out that there's a nice design that combines these two together, and it's what's broadly used today: a **virtually-indexed, physically-tagged cache**. Remember that a physical address consists of a PPN and an offset, while the fields of the cache that we care about are a cache tag, index, and line offset. One convenient situation is the one where **the index bits are a subset of the physical address offset bits**: these are nice, because we can start accessing the cache before finishing the virtual-to-physical translation, and thus we can do **lookups into the cache and TLB in parallel**. Our cache will return some physical address, the TLB will return some PPN, and we can compare the two results to see if we do indeed have a hit.

Fact 172

This requirement of having the index bits fit inside the offset bits gives us certain constraints on the size of cache – at a certain point, we can only increase the size of the cache by increasing associativity. And looking at modern designs, that's indeed the main reason that we have N-way set-associative caches – we want to be able to do these lookups in parallel.

20 November 12, 2020

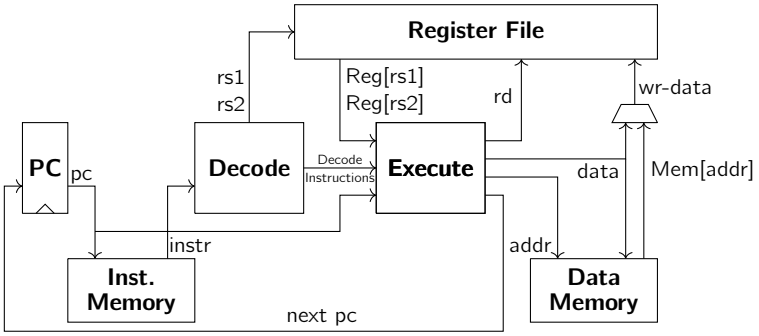
Quiz 2 is being graded, and the results should be released back to us by tomorrow evening.

Today, we'll discuss how to improve the performance of our processors through **pipelining**. Let's start by looking at the parameters that measure performance: we care about how much time it takes to finish executing a given program, and we can write this as

$$\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycle}}{\text{instruction}} \cdot \frac{\text{time}}{\text{cycle}}.$$

So we want to minimize time, and we can do that by reducing one of the three terms here: (1) reduce how many executed instructions we run, (2) decrease the number of cycles per instruction, or (3) reduce cycle time. But (1) requires us to do more work per instruction, which means we need to support more complex instructions and change our ISA, and that's out of the question. Also, (2) isn't possible either, because we're assuming that the **CPI** (cycles per instruction) is 1 for our instructions already (we can't do any better than that unless we're thinking about parallel processing). So we'll talk about method (3), and we can **improve our cycle time by pipelining**.

Recall that we have the following diagram for our single-cycle processor:



Currently, the amount of time it takes to execute one cycle is the clock period t_{CLK}, which is determined by the **longest path for any instruction** (since we want the CPI – cycles per instruction – to still be 1). And such paths can be pretty long: for example, an ALU instruction looks up the PC, fetches the instruction from our instruction memory, decodes that instruction, fetches the source operands from our register file, executes the ALU function, and then either go through our data memory or write to the register file. Right now, the clock period must allow for all of these different steps to happen, one after the next:

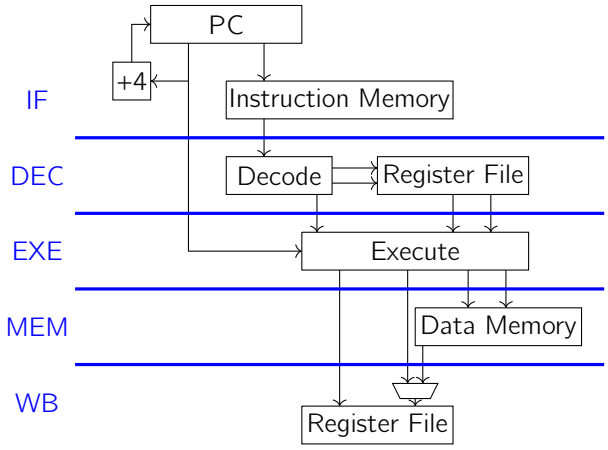
$$t_{\text{CLK}} = t_{\text{IMEM}} + t_{\text{DEC}} + t_{\text{RF}} + t_{\text{EXE}} + t_{\text{DMEM}} + t_{\text{WB}}.$$

This is very slow, so we'll take an idea from one of our previous lectures: we'll **divide our datapath into multiple pipeline stages**, meaning that each instruction takes multiple cycles to execute, but having overlapping instructions allows us to keep our throughput to about 1 instruction per cycle, and we can decrease t_{CLK} significantly. Today, we're going to look at the classic **5-stage pipeline**, with the following five stages:

1. **Instruction Fetch (IF)**: keep track of the PC and fetch the instruction, passing this to:
2. **Decode and Read Registers (DEC)**: decode the instruction, read the source operands from the register file, passing these to:
3. **Execute (EXE)**: perform the ALU (and other) operations, and give the result to:
4. **Memory (MEM)**: read from our main memory if we have a load (or just directly pass the execute unit's result through this stage) into:
5. **Writeback (WB)**: write our results back into the register file.

Just like pipelining in the past, if we put a register between each of these stages, we significantly reduce t_{CLK} and keep throughput basically the same. This might all sound simple, and recall that we already know how to pipeline combinational circuits from a previous lecture. **But this process is not quite as simple as it might initially seem**! First of all, our processor has several **states** (because it is sequential rather than combinatorial), like our memories, PC, and register files. But so far, we've only learned how to pipeline combinational circuits, and that means we need to think a bit more about how our existing states interact with register files. Second, **there are loops in our circuit**, like the nextPC calculation and writing back into the register file, that we can't break yet.

So let's simplify our single-cycle datapath, redrawing it so that we can better indicate the different stages that we plan to have in our pipeline. The diagram below redraws our single-cycle processor, indicating the different stages more clearly:



For simplification today, notice that we're assuming we have no branches or jumps, so the next PC is always PC + 4 (we'll deal with branches and jumps in Tuesday's lecture). This removes the cycle from the Execute function back to the PC, which removes a complicated loop. Also, we've replicated the register file twice in the diagram – this way, we can distinguish when it's being used for reads versus writes. In particular, we only do reads in the DEC stage, and we only do writes in the WB stage. We're also assuming here that we have magic memories, so that loads are combinational and return data in the same clock cycle as their request (to avoid overlaps between different stages). This constraint will be removed by the end of this lecture, though.

So now let's add pipeline registers at the **blue lines** marked in our diagram. It now takes five cycles for each given instruction to finish, but as soon as an instruction leaves the IF stage, we can fetch the next instruction, and this means we're overlapping the execution of instructions (this is the whole point of pipelining!). Therefore,

$$t_{\text{CLK}} = \max(t_{\text{IF}}, T_{\text{DEC}}, T_{\text{EXE}}, t_{\text{MEM}}, t_{\text{WEB}}),$$

which is significantly faster than the sum – if the delays of each stage are about equal, our clock cycle is about a fifth of what it was before.

Example 173

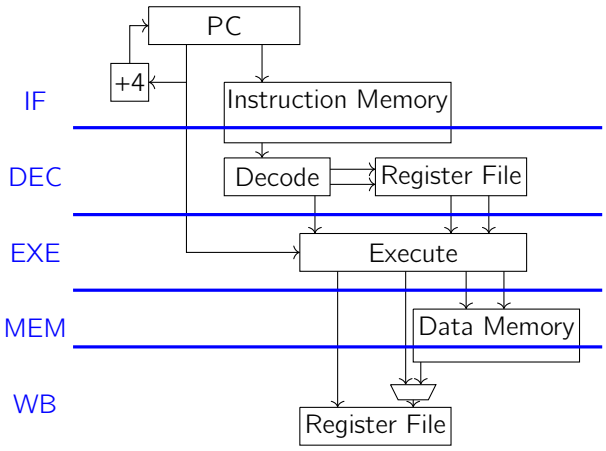
Suppose we have a set of assembly instructions, like the ones shown here:

```
addi x11, x10, 2
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, 21
add x22, x23, x24
addi x25, x26, 1
```

Let's try to construct a pipeline diagram for this execution – recall that the rows are pipeline stages, and the columns are our cycles. Then our instructions will propagate diagonally downward in the table: again, even though our latency is now 5 clock cycles, the length of each clock cycle is much shorter, so we expect the latency to actually be only a little worse than before. (And of course, the throughput is now much better.)

	1	2	3	4	5	6
IF	addi	lw	sub	xor	add	addi
DEC		addi	lw	sub	xor	add
EXE			addi	lw	sub	xor
MEM				addi	lw	sub
WB					addi	lw

Let's look at when **register reads and writes** happen in our processor now: remember that reads happen in Decode, while reads happen in the Writeback stage. For example, the addi instruction read from register x10 in cycle 2, and it does its computation, but it doesn't write to register x11 until the end of cycle 5, which is three cycles later. But now it's worth being more realistic with our memories: we don't actually use magic memories in real life, and even for load operations, we have clocked reads. So we'll redraw our diagram a little, with the only change being that our memories now extend past our registers:



As a clarification, this means that we provide the data value associated with an address **during the next cycle** (if we have a cache hit), rather than in the same cycle. Clocked reads actually **simplify some pipelining issues**, which is why we're using them going forward.

Now that our memories are realistic, let's return to the pipeline diagram for the instructions above: reads and writes still happen in the same pipeline stages, and we still have three cycles between a read and write for a given operation. But now we can point out an explicit problem our pipelining.

Example 174

Suppose we have a set of instructions like the ones below, where we are actually writing and then reading to the same register during the process:

```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

This time, the addi writes to register x11, and then the xor wants to read it in the next cycle. But right now, the value that the xor reads is an old (stale) value, because we don't write until three cycles later! That's what the rest of today's lecture is about: we'll figure out how to have our pipeline processor deal with overlaps like this.

Definition 175

A **data hazard** arises when there is bad dependency for data values, while a **control hazard** arises when there is bad dependency for the program counter.

Again, we'll handle data hazards first (today), and then we'll handle control hazards next time (on Tuesday).

Proposition 176

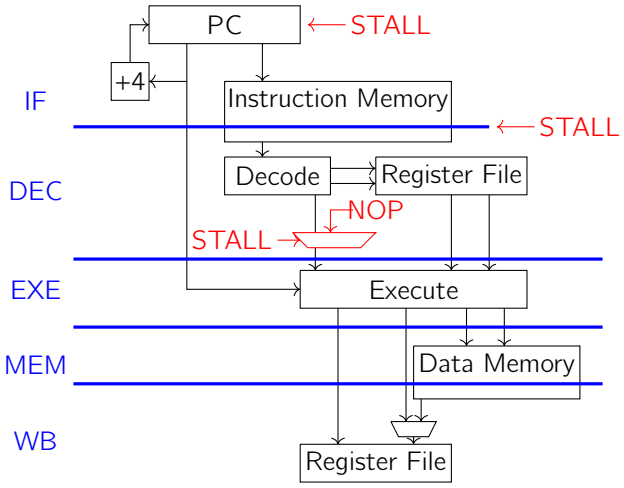
There are three main strategies for resolving hazards that we'll employ. We can **stall**, meaning that we wait for the necessary result to be written back before continuing executions, freezing the earlier pipeline stages. But to waste less cycles, we often **bypass** instead, which means we route data to earlier pipeline stages as soon as it is computed, rather than waiting for the writeback to occur. Finally, we can **speculate**, which means we try to guess the value and keep executing the following instructions. If we guess correctly, we continue; otherwise, we kill the process and restart with the actual value.

Let's first look at **stalling**, which is strategy 1. Our pipeline diagram for Example 174 will now look like this:

	1	2	3	4	5	6	7	8
IF	addi	xor	sub	sub	sub	sub	xori	
DEC		addi	xor	xor	xor	xor	sub	xori
EXE			addi	**NOP**	**NOP**	**NOP**	xor	sub
MEM				addi	**NOP**	**NOP**	**NOP**	xor
WB					addi	**NOP**	**NOP**	**NOP**

Basically, while the xor instruction is in the Decode stage, since we know that we need to read the value of the addi before we can execute the xor, we don't allow anything to happen in the earlier pipeline stages until addi completes. (Here, **NOP** stands for "no operation," and in practice it's just a command like `addi x0, x0, 0` which doesn't actually modify any state.) This handles the logic correctly, but of course it also increases our CPI (because of all of the NOPs hindering the completion of our instructions).

Implementation-wise, we add a new control signal to our pipelined processor called STALL, set to 1 whenever we detect a data dependency of some kind.



When STALL is 1, the PC doesn't accept a new value, and the IF stage will also keep operating on the same instruction it's been working on (meaning the Decode stage will also still see the same inputs from the previous cycle).

But the Decode register will receive a **NOP** instruction when STALL is 1, which will propagate forward like any other instruction through the remaining stages. To figure out when to set STALL to 1, we look at the instructions in our current pipeline, seeing whether the instruction currently at the Decode stage has a source register (input) matching a destination register (output) of either the Execute, Memory, or Writeback stage (and of course, excluding the case where the source register is `x0`).

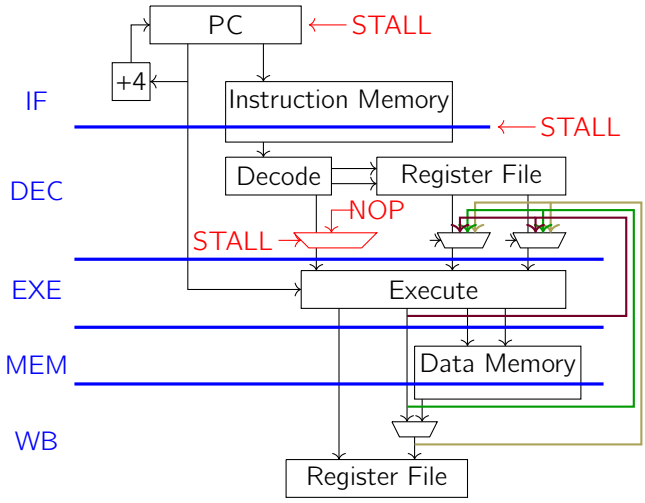
But there are other possible data hazards as well. Consider the following two instructions:

```
sw x11, 8(x10)
lw x13, 4(x12)
```

None of the registers show up twice in the code, but it's possible that the two addresses (`x10 + 8` and `x12 + 4`) actually coincide! Because this is possible, we need to make sure the store operation finishes before we do the load operation. It turns out that **because our stores write to the register file in a single cycle, there are no hazards of this type in our case**. But in general, we do need to worry about this kind of hazard, and we can do so by either having our memory system detect loads and stores to the same location, or by adding extra logic in our pipeline to check whether the locations match. If such a data hazard does happen, that gives us another reason for us to stall the pipeline.

Bringing everything back to our starting discussion, remember that even though stalling is easy to implement, it still makes our performance worse. So now we can look at **bypassing**, which is strategy 2 from above. The idea here is that even if we haven't written a result into our register, it doesn't mean the value we need isn't available to us somewhere else in our processor.

For instance, going back to Example 174, even though `addi` writes to register `x11` three cycles after it is decoded, we already know the result of the calculation in the Execute stage, just one cycle after the decoding. So `xor` should be able to use that value in the next cycle without any stalling! That type of routing is implemented in the diagram below:



When we add the ability to bypass, we have to add quite a bit of hardware: there are 32-bit wires coming out of the Execute, Memory, and Writeback stages, and those wires feed into muxes at the end of our decode stage. And the mux selection is done as follows: we feed the data from our register file into the Execute stage as a source operand if there isn't a data hazard, and we read from downstream if there is one. And if more than one of the later stages have instructions that are both writing to the same register that the instruction in Decode is trying to use, we always use the **most recent value** (so EXE takes precedence over MEM, which takes precedence over WB).

However, because of the large amount of additional wiring required in this process, we often use **partial bypassing** instead of the full bypassing described above. If we run some benchmarks on our processor, we might find that the bypass path from Memory to Decode is very rarely used, so it's better to get rid of that one. This means that we only bypass from Execute or Writeback, and **we still have the stall logic** in the rare situation where we actually do need to have a problematic data dependency between Memory and Decode. Basically, we're making tradeoffs between how much extra hardware we add to the processor, versus how much of a hit we take by not being able to do that particular bypass.

Returning even to the fully bypassed pipeline, we know that the whole bypassing setup works without stalling if the value we're trying to access always becomes available in the execute stage. But **sometimes we will still need to stall**, specifically if we have data hazards associated with load operations. For example, consider the following sequence of operations:

```
lw x11, 0(x10)
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

Even with full bypassing, we can't resolve the data hazard coming from register `x11`, because we can't get the data value from memory immediately. Bypassing still lets us save one stall cycle, though: we've only inserted 2 **NOP**s instead of 3 in the pipeline diagram below, since the source operands for xor can be obtained in cycle 5 instead of 6.

	1	2	3	4	5	6	7	8
IF	lw	xor	sub	sub	sub	xori		
DEC		lw	xor	xor	xor	sub	xori	
EXE			addi	**NOP**	**NOP**	xor	sub	xori
MEM				addi	**NOP**	**NOP**	xor	sub
WB					addi	**NOP**	**NOP**	xor

And there are other things we can do to resolve data hazards, too: we can use compilers to identify data hazards, changing the order of instructions that are independent of each other, so that we avoid the hazard altogether. But in reality, it's usually pretty hard for the compiler to find such independent instructions, so this is not that effective.

In summary, we've discussed **stalling** and **bypassing** today, which are two different methods to avoid issues with data dependencies in a pipelined processor. (We've been limiting our discussion to a five-stage pipeline, but issues with data hazards become even worse if we try to add more pipeline stages!) Next time, we'll talk about **speculating**, which will help with control hazards.

21 November 17, 2020

We'll continue our discussion of pipelined processors today, looking at **control hazards**, but first we'll review and then clarify some issues regarding **data hazards**. Recall that the goal of this whole discussion is to improve processor performance, either by decreasing the number of instructions per program, cycles per instruction, or time per clock cycle. Because we've been building a single-cycle processor, and we don't want to change our ISA, the only real change we can make is the clock cycle time. And we're doing this by pipelining our processor into multiple stages, while still trying to keep our CPI as close to 1 as possible.

Last time, we discussed two strategies for dealing with data hazards (which occur when instructions in a pipeline depend on results of other instructions, but these results haven't been written back to the register file yet). In order to get the correct, most up-to-date data, we can **stall** and freeze our earlier pipeline stages until the correct values are available. This is simple to implement, but it increases our CPI, so we sometimes **bypass** instead, which means we make our data available in our datapath by routing it back to an earlier pipeline stage when we need it (for example, from the Execute stage back to the Decode stage). Although bypassing can lower our CPI, it can be expensive (in terms of space), and sometimes we make a tradeoff between bypassing and stalling by only doing partial bypassing. (After all, data hazards coming from loads cannot be resolved just by bypassing, so we're going to need to stall sometimes anyway.)

To visually represent what's going on with each of these methods, we often use a pipeline diagram. When data hazards come up, stalling introduces **NOP**s (wasted cycles), while bypassing does not always require us to do so (because we often send computed results directly from the Execute stage to the Decode stage, and this doesn't require an extra cycle unless we're doing something like a load).

So now we'll talk a bit more about a specific concept we skipped over from last lecture – recall that we don't need to worry about data hazards caused by memory, because we're using the assumption that memory responds in a single cycle. This is a realistic model if our memory is an instruction cache and we always have a cache hit: remember that we have clocked reads, so a cache hit returns our data only in the next clock cycle. But we sometimes do have **cache misses**, so we need a way of dealing with the case where we need to go into main memory (and thus it takes longer to get the data back). So **our cache sets the stall signal to 1 if we have a cache miss**, and our pipeline stalls execution until we get our memory back. (And now we'll label the corresponding spots in our pipeline diagram as "Instruction Cache" and "Data Cache," rather than "Instruction Memory" and "Data Memory.")

With that, let's return to our pipeline diagrams and see what happens during **instruction cache misses** and **data cache misses**.

Example 177

Suppose we have the following instructions (which will not result in any data hazards due to repeated registers):

```
addi x9, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

Suppose that our instruction cache doesn't respond to the fetch for the xor instruction immediately. Then our Decode stage can't do anything yet (because we haven't gotten anything from our instruction memory), so we have to inject some more **NOP**s until we get back the actual instruction that we're waiting for.

	1	2	3	4	5	6	7	8
IF	addi	xor	sub	sub	sub	sub	xori	
DEC		addi	xor	xor	xor	xor	sub	xori
EXE			addi	**NOP**	**NOP**	**NOP**	xor	sub
MEM				addi	**NOP**	**NOP**	**NOP**	xor
WB					addi	**NOP**	**NOP**	**NOP**

In this example, the instruction cache only gives us back the valid instruction during cycle 6, so only then can we begin decoding the xor. When this occurs, we start fetching sub as well. The end result of this pipeline diagram looks

a lot like the one for data hazard stalling: **execution is stopped in the Decode** (and thus also Instruction Fetch) stage, and it's just a different root cause, because we're waiting for memory rather than other instructions.

To make instruction cache misses work in our hardware, notice that we already have support for stalling in our hardware: when the stall signal is set to 1, we insert a **NOP** into the Execute stage, and the PC and the Instruction Fetch register are disabled (stopped by the STALL signal). The **only thing that changes** now is that if our stall comes from an instruction cache miss, **the instruction cache will keep working** while everything else in that pipeline stage is stalled. So no hardware needs to be added – we just need to insert some extra control logic, where STALL is 1 **either if** there's a data hazard or an instruction cache miss.

Example 178

On the other hand, let's say we have a sequence of instructions as shown below:

```
addi x9, x10, 2
lw x13, 0(x11)
sub x17, x15, x16
xori x19, x18, 0xF
ori x2, x1, 0x3
```

This time, suppose that the lw instruction doesn't immediately finish getting the value because we have a data cache miss. Our pipeline diagram then looks like this, where nextI refers to the next instruction:

	1	2	3	4	5	6	7	8
IF	addi	lw	sub	xori	ori	nextI	nextI	nextI
DEC		addi	lw	sub	xori	ori	ori	ori
EXE			addi	lw	sub	xori	xori	xori
MEM				addi	lw	sub	sub	sub
WB					addi	lw	lw	lw

Basically, we make a lw request to the data memory in cycle 5, but in cycle 6 we find that we don't have the value yet because of the miss. This time, all five pipeline stages get stalled, but we don't have any **NOP**s, because our stall signal is caused by the lw that sits in the final stage of the pipeline! So we already have valid instructions in the processor and we just need to make sure nothing moves forward.

We're now done discussing data hazards, and we can move on to **control hazards now**. Here's the basic idea: so far, we've assumed that our execution is always sequential. So the next value of PC is always set to PC + 4, but we know that that's not necessarily true when we have branch and jump instructions.

Let's think about what information we need to compute the next value of PC in general. Clearly, we need the **opcode**, because that tells us what kind of instruction we're executing (for example, ALU operations will make the next PC just PC + 4, while jump instructions require us to do some additional calculations to determine the next PC). Recall that the nextPC is `pc + immJ` for a JAL instruction (jump by some constant), `(reg[rs1] + immI)[31:1], 1'b0` for a JALR instruction (take the value at some register, add a constant, and clear the bottom bit), and `brFunc(reg[rs1], reg[rs2])? pc+immB : pc+4` for a branch instruction (jump by the immediate if the branch comparison tells us to jump, and otherwise we proceed as normal).

Overall, this means that the value of nextPC will be **available at different stages**, based on what kind of pipeline we're using and also the kind of instruction. Looking at our five-stage RISC-V processor, the PC is available in the Instruction Fetch stage, and the opcode and immediate become available in the Decode stage. But even after the

Decode stage, we don't always have all of the information – operations (primarily addition) on `pc`, `imm`, `reg[rs1]`, and `reg[rs2]` only become available in the Execute stage. So **we don't actually know the next PC until the Execute stage for JAL, JALR, and branch functions**, while in other cases we can know it in the Decode stage (because it's just PC + 4).

Remark 179. *In the JAL case, where we just add an immediate value to the PC, we could potentially know things in the Decode stage if we were to implement an extra adder in the Decode stage. But our processor has the ALU doing that addition, so we do have to wait until Execute.*

Now, let's think about the implications of this new complication. Stalling can also help us deal with control hazards, and let's study that first:

Example 180

Suppose we have a series of instructions as shown here:

```
loop: addi x12, x11, -1
      sub x14, x15, x16
      bne x13, x0, loop
```

Assume we take the bne branch each time, and we stall whenever necessary to avoid control hazards. Then even if we do something as simple as an addi instruction, we can't assume the nextPC is going to be PC + 4 until the Decode stage! So we'll need to **stall after each instruction** so that it can make it through the Decode stage, and only then do we know that the next PC is PC + 4. So that adds **NOP**s at least every other cycle. Furthermore, we have an extra **NOP** after the bne, because we only know the PC there once we get to the Execute stage. So our pipeline diagram looks something like this:

	1	2	3	4	5	6	7	8	9
IF	addi	**NOP**	sub	**NOP**	bne	**NOP**	**NOP**	addi	**NOP**
DEC		addi	**NOP**	sub	**NOP**	bne	**NOP**	**NOP**	addi
EXE			addi	**NOP**	sub	**NOP**	bne	**NOP**	**NOP**
MEM				addi	**NOP**	sub	**NOP**	bne	**NOP**
WB					addi	**NOP**	sub	**NOP**	bne

Notice that the CPI here is 7 cycles for 3 instructions, and at that point it's not even clear whether pipelining is worth it! So we need something better, even if we don't necessarily know what our next PC is.

And this is where **speculation** comes in: the basic idea is to guess what the next PC will be, and we execute assuming that our guess is correct. If we guess correctly, we keep going, saving us some cycles. But if we guess incorrectly, we started to execute some instructions we shouldn't have, so we kill them and start over again with the correct instructions.

Proposition 181

The best guess for nextPC is PC + 4, because almost all of our instructions (and also non-taken branches) give us PC + 4.

Example 182

Let's now consider the following instructions, and let's try to solve control hazards using speculation:

```
loop: addi x12, x11, -1
      sub x14, x15, x16
      bne x13, x0, loop
      and x16, x17, x18
      xor x19, x20, x21
```

If we **don't take the bne branch**, the correct next instruction is indeed always found at PC + 4, so our pipeline diagram looks like this:

	1	2	3	4	5	6	7	8	9
IF	addi	sub	bne	and	xor				
DEC		addi	sub	bne	and	xor			
EXE			addi	sub	bne	and	xor		
MEM				addi	sub	bne	and	xor	
WB					addi	sub	bne	and	xor

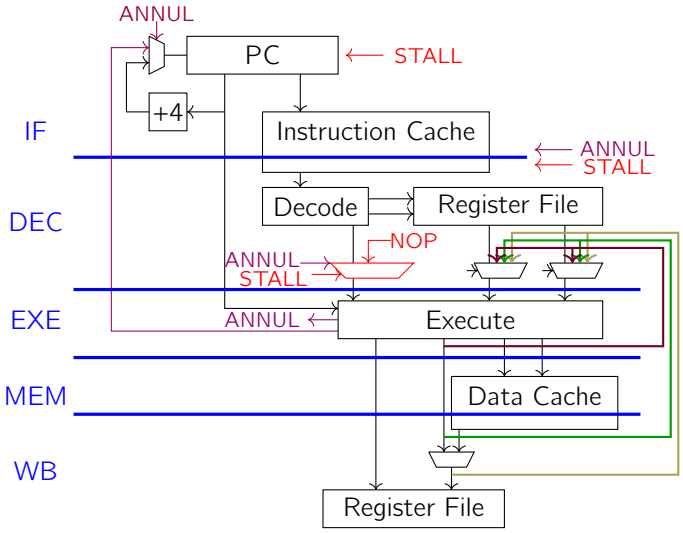
In this case, we find out in stage 5 that the bne instruction does not branch, so nothing needs to be changed, and we've avoided wasting cycles.

On the other hand, **if the bne branch is taken**, we also find this out during the Execute stage in cycle 5. So in stage 6, we are supposed to start running addi instead of and, which means our pipeline diagram looks like this:

	1	2	3	4	5	6	7	8	9
IF	addi	sub	bne	and	xor	addi	sub	bne	and
DEC		addi	sub	bne	and	**NOP**	addi	sub	bne
EXE			addi	sub	bne	**NOP**	**NOP**	addi	sub
MEM				addi	sub	bne	**NOP**	**NOP**	addi
WB					addi	sub	bne	**NOP**	**NOP**

What we've done is replace the and and xor with **NOP** operations – this is called **annulment**, and then the **NOP**s proceed through our pipeline instead of the incorrect instructions that we started. And it's okay for us to fetch and start running these instructions, because going through the Instruction Fetch and Decode stages **does not change the state** of our processor! (Nothing makes it to the Writeback stage, and thus everything we've done is safe.) Furthermore, this time our CPI is 5 cycles per 3 instructions, which is much better than the 7 cycles per 3 instructions from stalling.

This time, we need to add some extra hardware to make speculation logic possible. See the diagram below:



Now, whenever the Execute unit is computing the next PC value, it sometimes finds that we have a jump or a branch that is taken. Then we set the ANNUL control signal to 1, and we supply the mux at the top with the value of nextPC. Furthermore, when we have an ANNUL signal, we have to do **branch annulment**, which replaces instructions in the top two registers (IF-DEC and DEC-EXE) with **NOP**s, and we load the PC register with the correct value of nextPC.

We're almost done – we just need to think about some corner cases, making sure that we can handle everything that might come up.

Problem 183

Suppose that in the same cycle, we have two different hazards: the Execute stage wants to annul because of a control hazard, and the Decode stage wants to stall because of a data hazard.

Here's an example where this might happen – assume that we take the branch in the following set of instructions.

```
addi x12, x11, -1
 lw x14, 0(x15)
bne x13, x0 loop
and x16, x14, x18
xor x19, x20, x21
```

Now the`and` wants to stall because we don't have the value of `x14` yet, and the `bne` wants to annul because it turns out we're supposed to go back to the beginning of loop. And in such a case, **ANNUL comes from an earlier instruction**, so we want to have that behavior take precedence. And this kind of question needs to be addressed in implementation – we should think about all of the special cases.

To finish, let's see an example where everything comes together, and we have stalling, bypassing, and misspeculation. Take the set of five instructions above, and suppose that `bne` is taken once but then not taken the next time. Then our pipeline diagram will look like this:

	1	2	3	4	5	6	7	8	9	10	11	12
IF	addi	lw	bne	and	xor	addi	lw	bne	and	xor	xor	
DEC		addi	lw	bne	and	**NOP**	addi	lw	bne	and	and	xor
EXE			addi	lw	bne	**NOP**	**NOP**	addi	lw	bne	**NOP**	and
MEM				addi	lw	bne	**NOP**	**NOP**	addi	lw	bne	**NOP**
WB					addi	lw	bne	**NOP**	**NOP**	addi	lw	bne

We'll go through this step by step. We start by executing our addi, our lw, and our bne instructions, and then because we predicted that nextPC should be PC + 4, we fetched the and and xor instructions. But in cycle 5, we find out that our speculation is wrong, so we need to annul and start running addi instead, adding in two **NOP**s. Then in cycle 10, our fetch of and and xor instructions are correct (because we speculate correctly). But then we find a data hazard in cycle 10: the and requires the value of `x14`, so we stall for one cycle so that the lw can reach the Writeback stage and our operands can be correctly determined. (And remember that if we didn't have bypassing here, we'd need to stall for two cycles instead of one at cycle 11.)

In summary, stalling can address all pipeline hazards, but it can hurt CPI because it is much slower than the other options. In reality, there's a small piece of our pipelined processor that's missing from the diagram above – it's the data miss stalling coming from a cache miss in our Data memory. (In our design project, we will figure out how to deal with that logic ourselves.) Bypassing helps with data hazards, and speculation helps with control hazards (we don't know what the data values will be, so we shouldn't use speculation for data hazards). Modern processors actually do more work to improve the chance of correct speculation – in the last lecture, we'll see a little bit of this, using additional hardware to make branch prediction more likely to be correct.

22 November 19, 2020

This is the last lecture whose content will be quizzed – our final quiz will be Thursday, December 3, and it will cover lectures 17 through 21 (not as much material as the previous ones). Practice quizzes will be released in a few days, and the review session will take place on Tuesday, December 1. Also, the due date for lab 7 has been extended to the last day of Thanksgiving break (Sunday, November 29), which should help those of us who are moving off campus around now.

Today's topic will be **synchronization**. Last time, we were discussing ways to improve the performance of our processor by pipelining a single **thread of execution** into multiple stages, reducing the clock cycle significantly. And today we're going to think about another way of improving performance by adding parallelism, dividing our computation among different threads of execution. (Of course, this complication will introduce more things to worry about.)

These multiple threads can either be completely independent of each other, meaning we have **independent sequential threads** that are just competing for shared resources (like memory), or they can communicate with each other, meaning there are **cooperating sequential threads**. In the latter model, different threads can do different portions of calculation, and we pass results between them to make the whole process go faster.

In a simple model, we might have a shared memory model (which is what today's lecture will focus on), where there are multiple parallel threads that share a single memory and thus implicitly **communicate by doing loads and stores** to the same address space. (But an alternative model is to do **message passing**, where we have separate address spaces for each thread, and the threads explicitly send and receive messages to each other through a network.)

Proposition 184

Synchronization between threads is necessary for a few reasons. If there are **forks and joins** (meaning one of the parallel processes needs to wait for other events to occur before proceeding), a **producer-consumer model** (meaning one thread produces an output which another thread needs as input), or **mutual exclusion** (meaning our operating system needs to prevent multiple processes from using a shared resource at the same time), we need a mechanism to make sure things happen in the right order.

Basically, we need to think harder about what happens when multiple processes run in parallel, and what our expected behavior should be. The easiest way to think about a **multithreaded program** is by running it on a uniprocessor (only one CPU), using **timesharing**. This is a concept we learned about when we first discussed operating systems: we let one process run for a while, and while it runs, it believes it's the only thing running on the machine. After some time, the OS switches us to another thread, and we keep doing this until completion. But there can be a more complicated system as well: if we have a **thread-safe** multithreaded program, that means that the behavior is the same regardless of whether we run it on one or multiple processors. (But again, in this lecture, every thread will have its own dedicated processor, so the operating system doesn't need to do any switching.)

Example 185

Suppose we have two processes, specifically a producer and a consumer. The producer repeatedly loops, producing some character $\texttt{c}_\texttt{i}$, which it then sends to the consumer. Meanwhile, the consumer also repeatedly loops, taking in the value of $\texttt{c}_\texttt{i}$ and does something with it.

The producer will run a $\texttt{send}_\texttt{i}$ command on its ith cycle, which sends the value of the output, while the consumer will run a $\texttt{rcv}_\texttt{i}$ command, which receives that value. And there's an important set of **precedence constraints** that we need to impose on this system so that the sending and receiving can be done successfully: we'll write $\texttt{a} \preceq \texttt{b}$ if the execution of $\texttt{a}$ must be done before the execution of $\texttt{b}$. In this case, we need to guarantee that

$$\texttt{send}_\texttt{i} \preceq \texttt{rcv}_\texttt{i}$$

for all i, so that the value that needs to be consumed is already produced, and we also need

$$\texttt{rcv}_\texttt{i} \preceq \texttt{send}_{\texttt{i+1}},$$

so that the producer doesn't overwrite the communicated message before it's used by the consumer. (So if only one memory in location is used to pass values, we can't send more than one at once.)

If we want to relax these constraints, so that we don't necessarily need to run $\texttt{send}_1$, then $\texttt{rcv}_1$, then $\texttt{send}_2$, and so on, we can add some extra **FIFO (first-in-first-out) buffers** between our producer and consumer. If this buffer can store up to N characters, then our producer can now run up to N values ahead of the consumer, and we have

$$\texttt{rcv}_\texttt{i} \preceq \texttt{send}_{\texttt{i+N}}$$

as our constraint instead (it's okay for us to have up to N values that are queued for the consumer to use). This is good, because **the less synchronization that is required, the more parallel our processing can be**.

In hardware, this FIFO buffer is usually implemented (in shared memory) as a **ring buffer**, whose indices wrap back on themselves. Within each buffer, there is a **write pointer** and a **read pointer** – the former tells the producer where to put things into the buffer, and latter tells the consumer where to read values out. So if our producer sends the

value of c_0 and c_1 to the buffer, our read pointer will stay in place, while the write pointer now points two spots later in the ring buffer. If at a later time, the consumer runs its first cycle, it will read the value of c_0 stored in the first spot, and then the read pointer will increment by 1.

So let's think about how we might try to implement this FIFO buffer in code. Our producer and consumer communicate via shared memory here, so we'll want to put our buffer there: the `in` and `out` tell us where to store and load values in and out of the buffer.

```
char buf[N];
int in = 0, out = 0;
```

And now our producer and consumer each need functions that help us send and receive characters from the buffer: they're shown on the left and right below, respectively. (We have the modulo-N to make sure that we always wrap back around to the beginning.)

```
void send(char c){
   buf[in] = c;
   in = (in+1) & N;
}
```

```
char rcv(){
   char c;
   c = buf[out];
   out = (out+1) & N;
   return c;
}
```

This is a relatively simple implementation, but **it does not actually work** – there is no synchronization, so there's no reason why the precedence constraints will be satisfied! This means we're going to need to add a new construct to our languages, which was introduced in 1962 by Dijkstra:

Definition 186

A **semaphore** is a data type which takes on nonnegative integer values, with two operations acting on it. The first is `wait(s)`, which makes our code wait until the semaphore value `s` is positive, and then it decrements `s` by one. The other one is `signal(s)`, which increments `s` by one (so that a single waiting thread can proceed).

Essentially, what's going on here is that if we have a semaphore `s` initialized to some number K, then

$$\texttt{signal(s)}_i \preceq \texttt{wait(s)}_{i+K}$$

(we can't have more than K new loads succeed before we need to stop and wait for more signals). Let's think about how we can use this in some real precedence constraints now:

Example 187

Suppose we have two parallel threads `A` and `B`, each executing its own set of instructions sequentially. We want to make sure instruction `A2` completes before instruction `B4`.

To use semaphores to enforce this precedence constraint, we can follow a pretty simple recipe. We start by declaring a semaphore `s` initialized to 0, and we make sure a `signal(s)` statement is added right after `A2`, while a `wait(s)` signal is added right before `B4`. So both `A` and `B` can run, but we can't let `B` run `B4` until the value of the semaphore `s` reaches 1 (which happens after `A2` completes).

Semaphores can also help us with more abstract problems of resource allocation. Suppose, for example, that we have a limited pool of resources, and there are many processes that want to use them. To ensure that at any point

in time, we only use at most K resources, we can initialize a semaphore `s` with value K. Then whenever we use a resource, we must wrap the commands with a `wait(s)` and a `signal(s)` at the beginning and end, so that `s` is decremented by 1 while the resource is being used! So the semaphore value will always track how many resources are available in the pool, and we don't need to do any additional work.

So now let's apply this to our producer-consumer example, augmenting our code a bit. This time, we'll keep track of how many characters we've already written into our buffer by adding a semaphore:

```
char buf[N];
int in = 0, out = 0;
semaphore chars = 0;
```

Initializing the semaphore to 0 basically means our buffer starts off empty. And now we `signal` and `wait` accordingly to ensure that there are always enough characters for us to receive, so that we always have $\texttt{send}_i \preceq \texttt{rcv}_i$:

```
void send(char c){
   buf[in] = c;
   in = (in+1) & N;
   signal(chars);
}
```

```
char rcv(){
   char c;
   wait(chars);
   c = buf[out];
   out = (out+1) & N;
   return c;
}
```

Notice that we're both enforcing a precedence constraint and managing a resource with this single semaphore! And to get this to fully work, we need to make sure that we don't get more than N characters ahead in our buffer (or we start overwriting data that we still need to use). So we'll do that with another semaphore `spaces`, only allowing us to produce new results if there's space to do so, and this time we get the result shown below.

```
char buf[N];
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
```

```
void send(char c){
   wait(spaces);
   buf[in] = c;
   in = (in+1) & N;
   signal(chars);
}
```

```
char rcv(){
   char c;
   wait(chars);
   c = buf[out];
   out = (out+1) & N;
   signal(spaces);
   return c;
}
```

And now we keep track of both the number of characters in our FIFO buffer, as well as the number of empty spaces that we still have.

This strategy works if we have a single producer and consumer, but it still doesn't work if we have **multiple producers and consumers**. We can use an analogy to explain this:

Example 188

Suppose two people visit an ATM at the same time, and both remove $50 from the same account at the same time through the following function:

```
void debit(int account, int amount){
   t = balance[account];
   balance[account] = t - amount;
}
```

The two ATM machines that are being used are running on different threads, and if `t0` stores the address of `balance[account]`, our assembly code might look something like `lw t1, 0(t0)`, `sub t1, t1, a1`, `sw t1, 0(t0)`, running on two different threads.

Now, suppose the thread 1 code runs before the thread 2 code. Since we properly decrement the bank balance each time, we withdraw $100, and the account amount goes down by $100. Everything is fine in this case! **But if the two calls interleave**, so both `lw t1, 0(t0)` commands are called before the `sub t1, t1, a1` and `sw t1, 0(t0)` commands on either thread, then the values of `t1` in the two register files will both be the original balance, and thus both withdrawals of $50 will be deducted from the original bank balance! Then we'll have withdrawn $100, and our bank balance has only gone down by $50 – we need to make sure this kind of situation doesn't happen.

The issue is that certain parts of our code should not interleave, especially if we modify shared data in them. These code segments are called **critical sections**, and we need to make sure that in any such sequence of operations, we aren't doing any interleaving. (This is exactly the **mutual exclusion** point from earlier on.)

But semaphores save the day again here – the mutual exclusion requirement can be satisfied if we wrap critical sections with a semaphore to ensure only one thread is executing them at a time. Let's use the notation $a \prec\!\succ b$ to mean that the instructions `a` and `b` don't overlap (so either `a` precedes `b` or vice versa). Then for our debit function above, we want to make sure we put a **binary semaphore** initialized to 1, tracking whether we're locking access to the critical section:

```
semaphore lock = 1;

void debit(int account, int amount){
   wait(lock);    t = balance[account];
   balance[account] = t - amount;
   signal(lock);
}
```

This may look very similar to the resource allocation idea from earlier on. Whenever we start running through the critical section, we set `lock` to 0 so that other threads can't run them, and then we set `lock` back to 1 once we're done. And now there are some additional questions we have to answer – for example, how granular should our locks be for the bank? (Should there be one lock for all accounts, per account, and so on?) What turns out to be practical is something in between: for example, all bank accounts that end in 004 use the same lock. Then we might need to do some waiting if someone else whose bank account number has the same last three digits as us is trying to withdraw money, but it's much more manageable for the bank to keep track of all of these different semaphores.

With this, we're ready to return to the producer/consumer model, now assuming that we have **two producers**. Re-

call that both producers need to run the commands `buf[in] = c;` and `in = (in+1) & N;`, which store the value of an output and then increment our write pointer. The issue we run into is when we interleave these operations between the two threads, and **both producers write to `buf[[in]` before incrementing**. To avoid this, we will want to do the exact "wrapping" that we've just mentioned: we wrap those two lines of code in a binary semaphore `lock`, and we do the same for the analogous lines in the consumer code. Now we finally have a **full multi-producer, multi-consumer implementation**, as shown below:

```
char buf[N];
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

```
void send(char c){
   wait(spaces);
   wait(lock);
   buf[in] = c;
   in = (in+1) & N;
   signal(lock);
   signal(chars);
}
```

```
char rcv(){
   char c;
   wait(chars);
   wait(lock);
   c = buf[out];
   out = (out+1) & N;
   signal(lock);
   signal(spaces);
   return c;
}
```

Now only one of our producers or one of our consumers can be modifying our grabbing from the buffer at a time, and this avoids writing two characters to the same location or reading the same character twice!

We now see that these semaphores are very powerful – a single primitive is able to enforce both precedence relationships and mutual-exclusion relationships. But there are still some issues to discuss, and the first comes when we think about how to implement semaphores. Semaphores themselves share data, and to edit their state, we need to be able to execute read/modify/write operations and treat certain sequences as critical sections.

So we need a way to guarantee mutual exclusion for semaphores, without using semaphores in the usual way. The first (and most common) approach is to add a special instruction into our ISA, exactly for this purpose. These are called **test-and-set** instructions, and they do an **atomic read-modify-write** as a single instruction (which therefore cannot be interrupted by other instructions). But a second approach, when we have a uniprocessor, is to implement semaphores with system calls from our operating system. (System calls mean that we enter our supervisor mode, and being in the OS kernel means our commands are uninterruptible!) So then we can execute instructions relevant to semaphores without interruption.

And there's one last thing to consider, which is an issue that can come up with synchronization and semaphores.

Example 189

Suppose we're trying to access multiple semaphores at the same time – for example, say we want to **transfer** money from one bank account to another with the following code:

```
void transfer(int account1, int account2, int amount) {
   wait(lock[account1]);
   wait(lock[account2]);
   balance[account1] = balance[account1] - amount;
   balance[account2] = balance[account2] + amount;
   signal(lock[account2]);
   signal(lock[account1]);
}
```

The issue comes when two different accounts each try to transfer money to each other at the same time. Since we're wrapping the portion of code where we modify account balances with `wait`s and `signal`s, what could happen is that both `account1` and `account2` get locked by the first lines of execution from the two threads. But then the next lines of execution require our locks to be released, and thus we end up in a **deadlock** situation: neither thread is able to get the locks it needs until the other thread gives an appropriate signal!

Fact 190

The **dining philosophers** problem is another one that leads to a similar kind of deadlock. Let's describe the setup for that. Suppose that we have N philosophers sitting around a table, and there are N individual chopsticks interspersed between them (so each person has a chopstick to their left and one to their right). Suppose that the philosophers follow a polite algorithm: they take (or wait) for their left stick, then take (or wait) for their right stick, then eat, and finally they replace both sticks.

If everyone tries to eat at the beginning, then everyone will grab the chopstick on their left. But then no one will be able to eat without another chopstick, so the chopsticks will never be put back, and thus no one will be able to do anything – we've indeed hit another deadlock! There are a few conditions that are leading us to this particular issue:

1. There is mutual exclusion, since only one thread (philosopher) can hold a resource (chopstick) at a time,
2. We have a hold-and-wait situation, because a thread can hold some allocated resources (the left chopstick) when waiting for others (the right chopstick),
3. There is no idea of preemption, meaning that we cannot remove a resource (chopstick) from any thread (philosopher),
4. This leads to a circular waiting situation where each thread is hanging.

We have two main solutions for avoiding a deadlock situation like this: we can come up with mechanisms for avoiding it, or we can set up ways to detect and then recover from deadlock. It turns out that there are indeed ways to do **avoidance** – for example, in the dining philosopher problem, we can assign different numbers to the chopsticks and require each philosopher to first grab the **lower** stick and then the **higher** stick. If we do this, then there can't be a circular wait, because it's not possible for the philosopher holding the highest-numbered chopstick to be waiting for another philosopher (they must have grabbed their other chopstick earlier on!). This allows the philosophers to continue dining without getting stuck in a deadlock.

So we'll apply this exact idea to our transfer function, making sure we have some ordering on the requested resources:

```
void transfer(int account1, int account2, int amount) {
   int a = min(account1, account2);
   int b = max(account1, account2);
   wait(lock[a]);
   wait(lock[b]);
   balance[account1] = balance[account1] - amount;
   balance[account2] = balance[account2] + amount;
   signal(lock[b]);
   signal(lock[a]);
}
```

Now if we have two users trying to transfer money to each other, only one of them will be able to grab the lower lock, and thus they will also be able to grab the higher lock (since the other person is still stuck waiting for the lower lock)! With that, we've figured out how to implement semaphores and therefore deal with synchronization in general.

Our last two lectures, covering cache coherence and modern processor architecture, are interesting and helpful for our design project, but they won't be tested on our exam. We should still attend those classes if we are able to do so!

23 December 1, 2020

We'll discuss **cache coherence** today: basically, when we have multiple processors running together and each is using its own cache, we need to think about what issues might come up. In a modern microprocessor, there are multiple **cores** (usually 2 to 8), and each one has a private cache (to improve load/store performance). But the main memory is shared between all of the cores, so we often have the cores communicate by interacting with that main memory. So we need to make it seem like operations are being performed on a single shared memory, even when each core has its own private cache.

Example 191

Suppose core 0 loads from address `0xA` (and finds the value 2), and afterward, core 2 stores the value 3 to address `0xA`. Then if core 0 loads from the address again, we'll get a **stale value** because core (process) 2 has updated the value, but core 0 still has 2 in the cache.

Definition 192

A **cache coherence protocol** is a system that ensures that all processors can see all loads and stores in a sequential order, even when we have multiple processes running in parallel.

With such a protocol in place, we don't return any stale values for a particular address, and we can ensure this as long as two properties are satisfied:

- At any point in time, only one cache has write permission,

- No cache has a stale copy of the data if a write is performed to a given address by another cache.

To do this, we ask for two rules to be satisfied: **write propagation**, meaning that writes will become visible eventually to all processors, and **write serialization**, meaning that those writes are seen in the same order for all processors.

- Write propagation can be guaranteed either through **write-invalidate protocols**, meaning that all other cached copies of data become invalid before the write is actually performed, or through **write-update protocols**, meaning that the write lets all other caches know to update their copies.
- Write serialization can be done either through **snooping-based protocols**, meaning that all caches observe each other's actions on a **shared bus** and make changes to their private caches accordingly, or through **directory-based protocols**, which won't be discussed much in 6.004 (but will be briefly mentioned at the end of this lecture).

Write serialization is what we'll focus on first:

Proposition 193

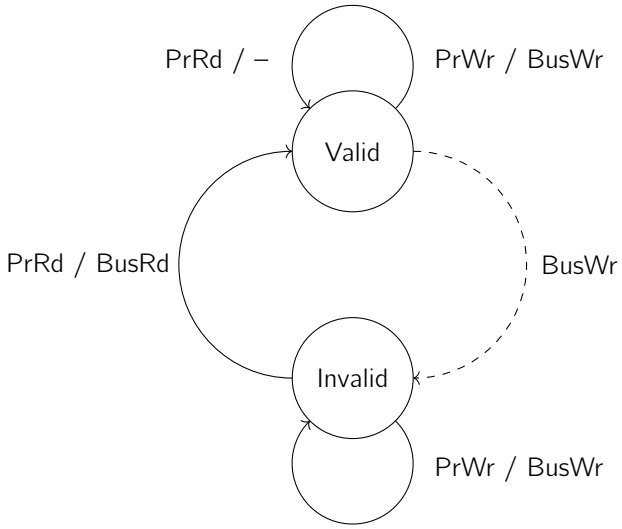
In a **snooping-based coherence protocol**, each process (core) has a "snoopy" cache, and there is a **shared bus** between each of the caches and main memory, where messages can be broadcasted (in an ordered way). All caches watch (snoop on) this bus to keep the state of memory coherent across all processor views.

We can make sure serialization occurs if all caches watch (snoop on) this bus to keep the state of memory coherent across all processor views. The way the actual protocol is defined is through state transition diagrams and actions that our caches must take in each case, and this is best illustrated with a finite-state machine. And when we think about potential transitions between the cache state, every cache cares both about its own load/store requests, but also any bus transactions that occur.

Definition 194

The simplest cache coherence protocol is known as the **valid/invalid protocol**: here, all lines can be in either the Valid or the Invalid state.

The finite-state machine is shown below, with all cache lines starting out in the Invalid state. The states correspond to each cache line's current state, and the arrows corresponding to the actions that are being initiated by various processes ("Pr" stands for "Processor," and "Rd" and "Wr" are "Read" and "Write"). In particular, **dashed arrows correspond to broadcasted messages from other caches.**



Whenever a cache wants to do a read (**PrRd**), it broadcasts a **BusRd** (Bus Read) command to the shared bus, which the main memory and other caches will see. Then main memory knows about the request, so it can respond with the correct data, and thus our cache can move from the Invalid to the Valid state. And now if this cache wants to do another read, it can respond right away, which is what the top left "PrRd / –" in the transition diagram tells us (we can answer from the cache directly, so nothing needs to be broadcasted). But now if our cache wants to write to an address, it needs to notify all of the other caches, so it does so by broadcasting a **BusWr** (Bus Write) message, and that will change the states of the other cache lines.

For example, suppose that core 0 is writing to address `0xA`, and core 2 already has a copy of the data at `0xA` in its cache. This means that core 2 will see the **BusWr** broadcast message from core 0, and it must invalidate its copy of address `0xA` in the core 2 cache, according to this protocol. But the main thing to keep in mind in this simple protocol is that we assume that we have **write-through caches**: in other words, these are caches where every time we do a write to the cache, we also update main memory. (So there is no notion of a "local write" allowed here.)

Example 195

Suppose we have two caches, and we'll work with a single cache line for all instructions. Consider the sequence of requests below.

1. If core 0 wants to load from `0xA`, it issues a **BusRd** request, and then it learns (from main memory) that tag `0xA` should be stored with a state of V (Valid) and some data, for instance 2. The cache of core 0 is then updated accordingly.
2. Next, if core 1 also wants to load from address `0xA`, it also issues a **BusRd** request, and in this case it either gets a response from the cache or main memory, and both will say that the value is 2. So core 1 also stores the (tag, state, data) information (`0xA`, V, 2), and any additional loads from either core mean we can just look at the valid value stored in each cache.
3. But next, if core 0 does a write (store) to `0xA` (for instance, if we want to change the value to 3), it issues a **BusWr** request to tell main memory and the other caches that the value is being changed. This then allows main memory to find out that the value is 3 (and updates main memory acccordingly), but it also tells core 1 to **invalidate** its copy of the cache line at tag `0xA`. So now core 0 can update its value to 3, and a read from `0xA` can be immediately done for core 0 but not core 1 anymore.

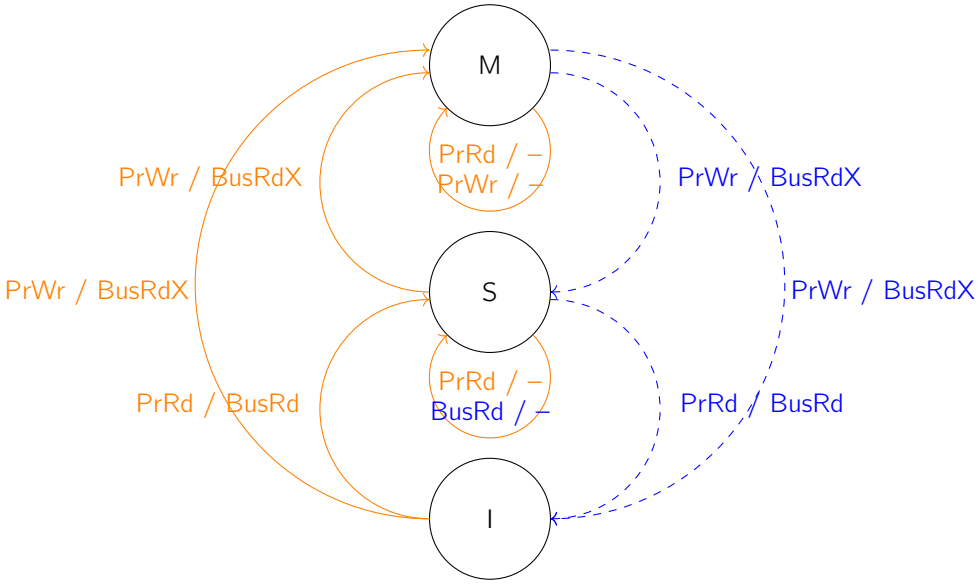
4. In particular, the next time a load from core 1 is requested, this requires a **BusRd** request, and either core 0 or main memory will tell us the value at that address.

There are problems with this valid/invalid protocol, though. Notice that every time we do a write, we need to let main memory know about the updated value, and doing this gives us worse performance than a write-back cache (where we only write back to main memory if we have to evict a line)! So we'll look at other protocols now, where not every write requires a broadcast and snoop.

Definition 196

A **MSI (modified/shared/invalid) protocol** is a different protocol with three states, where Invalid (I) means the cache doesn't have the address, Shared (S) means that multiple caches have an address (so the value can only be read), and Modified (M) means that only one cache has access, so we can read or write to that address (and all other copies are invalid).

The drawbacks of the VI protocol now go away – we can allow for writeback caches, and thus we are also able to do local writes. Let's go through this next state transition diagram more thoroughly – orange, solid arrows correspond to transitions caused by our own processor, and blue, dashed arrows correspond to transitions because of responses to snooped actions broadcasted by other caches. And the messages after the "/"s are bus requests that are broadcasted by our cache line for the transition to successfully happen.



Like before, each process can issue a process read or write request, and additionally, each process needs to look at what's happening on the bus during all of these reads and writes. In the MSI protocol, there are three types of broadcast messages: a bus read (**BusRd**), a bus read exclusive (**BusRdX)**, and a bus writeback (**BusWB**). Everyone starts off the in the Invalid (I) state, and if we want to read from an address, we issue a **BusRd** request, which allows the cache line to move into the Shared state. And if a core is in the Shared state, other processes are also allowed to request that same address (that is, other processes who issue a **PrRd** will also move from Invalid to Shared). And other processors doing reads doesn't affect any cache in the Shared state, because no one is trying to modify the value of that location.

But the complexity arises when we try to do a write: whether we're in the Invalid or Shared state, wanting to do a write means we need to initiate a **PrWr** request, and that means we also broadcast a **BusRdX** message. This

message means that we are requesting to be the only cache that gets access to this address, so the other caches need to invalidate themselves from Shared back to Invalid as soon as they snoop the message. So the bottom of our diagram corresponds to our cache line the least amount of access to the data, and as we move up the diagram, we get more privileges.

And finally, notice that when we exit the Modified state, we must issue a **BusWB** command because the value has been updated in our cache, but not yet to main memory. There are many arrows to track in this diagram, so we should go through and make sure each of them makes sense.

Example 197

As before, we work with two cores and a single cache line, and let's see what happens when we run through a sequence of requests, now with the MSI protocol.

1. Suppose core 0 wants to do a load to `0xA`. Then it broadcasts a **BusRd** request so main memory gives it the relevant value, which puts the cache line into the Shared state with some given data value, say 2.
2. Next, core 1 wants to do a load, so its **BusRd** request also puts this cache into a Shared state (and the value, 2, will be received either from main memory or core 0). And now, just like in the VI protocol, any additional loads to this address from either cache don't require broadcast requests – the private caches can respond appropriately.
3. But now if core 0 wants to write the value 3 to address A, we go from Shared to Modified, and to do this we issue a **BusRdX** command. Then core 1 needs to invalidate its copy of data at `0xA` (so core 1 goes to the Invalid state), and cache 0 is now in Modified with the value 3. Note that main memory has not yet updated the value from 2 to 3, but this setup is nice because whenever core 0 gets additional reads or writes, it can still do everything locally because it's the only cache line in the Modified state.
4. And now if core 1 wants to write the value 10 to address `0xA`, it issues a **BusRdX** request, telling core 0 that it needs to invalidate its copy. And this time, core 0 needs to writeback the latest data to main memory, and it does this with a **BusWB** request. This tells main memory, and also core 1, the current value of 3 at `0xA`, and that enables core 0 to invalidate itself (as it's passed along its new, updated value). Finally, core 1 can go into its Modified state and overwrite the value with 10.
5. Finally, if core 0 wants to load the value at `0xA` (remember that it's currently in the Invalid state), we issue a **BusRd** request, meaning that core 1 goes back down to the Shared state (using a **BusWB** request in the meantime). Then main memory gets the new value (10) and so does core 0's cache, and now both cores are in the Shared state with value 10.

The main advantage here is that once a cache line is in the Modified state, it can respond to its processor directly from the cache, which is much faster than having to go to main memory and issuing bus requests every time. **But we just need to be careful with timing** – if main memory had responded with 3 in the last step instead of core 1 responding with 10, that would have been incorrect, so we need to ensure that **caches respond more quickly than main memory** (which is what does generally happen).

And there's one final optimization we'll make today: read-modify-write sequences are pretty common in our code, but even the MSI protocol requires 2 bus transactions every time we do a read-modify-write.

Proposition 198

We can add a fourth state, **Exclusive (E)**, to the MSI protocol. Being in this state tells us that we're not sharing the data with anyone else (like Modified), but we also haven't actually modified the data yet (so it's clean). And once we overwrite data with a write, we then move to the Modified state (this is like toggling the dirty bit to 1).

Adding this fourth state to get the **MESI protocol** means that the state transition diagram starts to look even more complicated, but it's basically the same as the MSI with an extra E state. This exclusive state allows us to say that processes in the invalid state which issue a **PrRd** (and thus broadcast a **BusRd** request) can move into the Exclusive state **as long as there aren't other caches who want a copy of this data**. (If others do want it, we need to be in the Shared state as usual.) And when we're in the Exclusive state, the benefit (versus being in Shared) is that we don't need to make extra bus requests, like **BusRdX**, during a **PrWr** request to get into the Modified state, because we're already the only one with control.

To summarize, cache coherence is a topic that has lots of ongoing research – we're actively interested in how to make the best use of private caches when there are multiple processes running in parallel. Because all of these processes share the same main memory, we need to follow these protocols carefully to avoid having situations where different processes see different values the same memory address. And we'll mention briefly that **directory-based cache coherence**, at a high level, has all of the coherence transactions go through a directory (instead of requiring snooping). Basically, main memory is divided into chunks, and each processor is responsible for a specific chunk of memory. Any time that chunk wants to be accessed by a cache, that cache will communicate over a network, and the processor responsible for that chunk makes appropriate updates and keeps track of what all other caches are doing. So we don't need to do any general broadcasts, and the interconnection (1-to-1 communication) is good enough.

There's a few other issues we've skipped over in this lecture – in today's lecture, we really just looked at a situation where a cache line has a single word. But we mentioned in the cache lecture that there are actually multiple words per line, so there's a bit of a problem where we invalidate cache lines even when there isn't any sharing going on (if two cores modify different words in the same line address, for example). So this might give us a "ping-pong" set of invalidations, which is important to keep in mind.

24 December 3, 2020

We're wrapping up the course today, first looking at the main techniques that **modern processors** use to achieve good performance (similar to the concepts of pipelining and caching, which puts us at how computer architecture development looked in the 1980s). Basically, learning more techniques will help us build better processors if we decide to go down that path, and it'll also help us build faster programs because we know what's going on under the covers. And we'll also emphasize certain techniques that help us on the design project, such as simple branch prediction.

As a reminder, processor performance can be measured as a product of three quantities: the instructions per program, the average CPI during execution, and the clock cycle time. This concept is called the "iron law" of processor performance, and when expressing the performance in this way, each factor is actually impacted by different design choices (the ISA and compiler help reduce the first factor, the implementation style helps with the second, and implementation technology, like having faster transistors, helps with the third). So we can distinguish software architecture from circuit-level tradeoffs, and we can improve those two aspects separately.

We've already seen that pipelining helps us lower clock cycle time, though there is always a tradeoff: the CPI increases above the base "ideal" CPI when we pipeline our processor, because we gain an extra penalty due to data

and control hazards and dependencies (cache misses, branches, jumps, exceptions, and other long operations). We can thus think of our CPI as $\text{CPI}_{\text{ideal}} + \text{CPI}_{\text{hazard}}$, where the hazard term comes from the data and control hazards mentioned above. (And in modern processors, the tens or hundreds of cycles it takes to look in main memory or run other programs are the main contributors to the hazard term.)

Example 199

Earlier in the class, we described a standard **five-stage pipeline**: assuming full bypassing from the Execute, Writeback, and Memory stages back to the Decode stage, the ideal CPI is just 1.0 (in the case where we have no stalls, no mispredictions, and no annulments).

But there are penalties because of the pipelining – if we have a load-to-use data hazard (where we need a value very soon after requesting data from our memory), then even with full bypassing, we can lose up to 2 cycles (the worst case is if we have a load instruction like `lw x2, 0(x1)` and then immediately afterwards run something like `add x3, x2, x1`) while we wait for the data to be given to us. And we also lose up to 2 cycles if we have a taken branch that we did not predict – we don't know the next PC until we're in the Execute stage (for instance when we have a branch), and then the redirect introduces **NOP**s for the two instructions at addresses PC + 4 and PC + 8. So in the end, we're wasting cycles due to hazards, and depending on how frequent those operations are, we'll have some amount of hazard contribution added to the CPI.

Proposition 200

In the design project, it's recommended that we implement a **four-stage pipeline** without a dedicated Memory stage. In addition, we are recommended to redirect the target address of the jump or branch into the instruction cache when we annul, rather than writing to the PC (this is called **PC bypassing**), to avoid losing another cycle.

So in this four-stage pipeline, if we have a load-to-use hazard, we'll only lose 1 cycle (in the worst case, the instruction following the load only needs to wait one cycle for the load instruction to go from Execute to Writeback). And similarly, if we have a control hazard, we again only lose 1 cycle – the PC isn't incorrectly again set during the cycle that we mispredict, but instead fetched correctly because the new PC was passed into the instruction cache. So if we try implemenmting this, we'll find that there's indeed a smaller contribution coming from CPI hazards, and this will indeed get us a higher CPI and better performance.

We should note that **depending on the number of stages of the pipeline, different events have different penalties** – we should try to derive delays by thinking about the principles, rather than memorizing the specific numbers. We'll see today four techniques that improve techniques, and all four are used in modern processors:

1. Use **deeper pipelines** so that more instructions can overlap and we can shrink the clock cycle further,
2. Build **wider pipelines** by processing multiple instructions per pipeline stage in the same cycle,
3. Reduce the impact of data hazards by doing **out-of-order execution** (so we run instructions as long as the operands have become available, to better tolerate latency), and
4. Do **branch prediction** to more accurately predict the target and direction for our jumps.

We'll go through these four ideas in turn. First of all, let's think about **creating deeper pipelines**. The idea is to break up our datapath into N pipeline stages (instead of just 4 or 5) for some larger number N, perhaps breaking up the memory or ALU into multiple stages as well. Ideally, this cuts our clock cycle time by a factor of N, and this is what allows modern processors to go from running at about 1.5 GHz to about 3-4 GHz (using about 15 to 20 pipeline

stages). But using a deeper pipeline causes **more dependencies** among faraway instructions, which cause many more data and control hazards. And having lots of registers will cause their own delay and add area and power too, but this strategy still works in practice up to about 20 or 30 stages.

Next, having wider (also called **superscalar**) pipelines means that we increase our pipeline with (having multiple registers where we can store instructions and data at the end of each pipeline stage, rather than just one). And if we let the processor operate on W instructions, then this lowers the ideal CPI to $\frac{1}{W}$ – modern processors sometimes use 6-wide or even 8-wide pipelines. But we can't go much further than that, because this gives us a lot of quadratic overhead: remember that the cost of implementing a register file grows quadratically with the number of words, so we would then need to add many more ports. And bypasses also get more complicated, because there are many more special cases that we need to check!

Fact 201

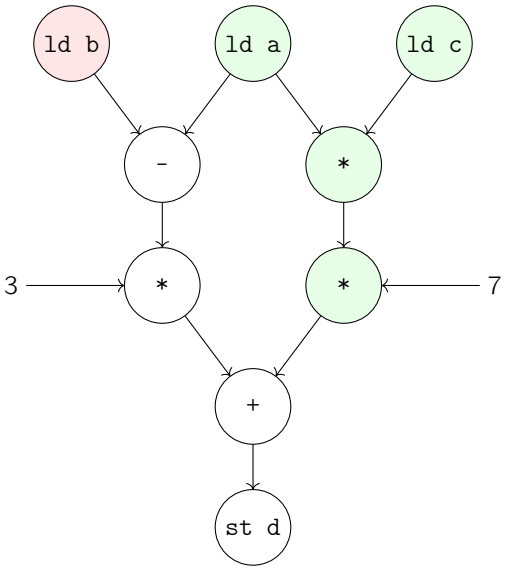
Taking a look at a sample 4-wide x86 processor and code snippet (x86 assembly is a bit different from what we've been studying, but we should still be able to understand it), we can run the `perf` command to use performance counters and tell us the number of cycles and instructions that are being run. (And indeed, we'll find that there are more instructions than cycles, so we've made the CPI less than 1 using superscalar pipelines.)

So the two techniques we described above work effectively, but **only because of the solutions for hazards** that we'll talk about in the rest of the class. To deal with data hazards (which become expensive at this scale), we're going to use a fourth strategy that's different from stalling, bypassing, or speculating, which is to **find something else to execute**.

Example 202

Suppose that we want to compute a value `d = 3(a-b) + 7ac`. In code, this can then be written as a sequential set of instructions.

But these instructions don't need to strictly be run in the order that they're listed: for example, we can build a **data-flow graph** that explains what instructions are dependent on other ones, as seen below.



The arrows represent outputs and inputs for each of the operations, and indeed we can see that there are parts of the dataflow that don't depend on others. So modern processors use this kind of internal representation to see which

operations have available inputs and run those as soon as they can! For example, if the `ld b` request gets a cache miss, we can still execute some other operations on the right (the ones colored in green), even though this is being done completely out of program order. And **modern processors can do this automatically** – a modern out-of-order superscalar processor first reconstructs the dataflow graph, then executes instructions accordingly, and finally uses a **reorder buffer** to writeback results in the correct program order. And we need this final reordering because of issues due to **synchronization and exceptions** – we need to give off the impression that everything is still running in the correct order. (This is an advanced concept called **memory consistency**, which we won't go through in this class.)

And the other problem we need to address, which we'll also focus more on for the design project, is the issue with **control hazards in a deep pipeline**. In modern processors, there are often going to be more than 10 pipeline stages between the nextPC calculation and the point of actual branch resolution, so we waste lots of cycles whenever we predict incorrectly. Doing a back-of-the-envelope calculation, we're losing more than 10 stages worth of instructions (the length of this **loose loop** between PC and Execute), multiplied by the width of the pipeline. For example, if we have a branch every 10 instructions (which is pretty typical), we take the branch half the time, and there are 15 stages and 6 instructions per stage, we turn 90 instructions into **NOP**s every 20 instructions, which is really bad (we're hurting our performance by at least a factor of 5).

So the solution for actually getting anything done with modern processors is to **make better guesses**, so that we only have mispredictions every few thousand cycles instead. First of all, let's think about the **earliest point** at which we can figure out the next PC. This depends on the pipeline, but the **two questions we need to answer** are "does instruction modify the next PC?" and "what is the next PC?". For example, the JAL instruction tells us that we need to jump once the Decode stage is completed, and we can know the target value after Decode too, because we have the immediate and the current PC already decoded. JALR is similar in that we know that we must jump after Decode, but we read the target address from a register (and then subsequently offset it), so we need to wait until after Execute to get the final answer. Finally, branches are the opposite situation – we need to wait until after Execute to know whether the condition is satisfied for taking a branch, but we already know where we would need to go after Decode is complete.

Proposition 203

This tells us one optimization to make in the design project: we can improve our CPI if we have a lot of JAL instructions by resolving those in Decode instead of Execute, saving a cycle.

And now let's think about how to make better guesses: we've been predicting that the next value is always PC + 4, and we'll do a better job now. There are multiple ways of dealing with this, and one is to do **static branch prediction**. Typically, branches are taken about 60 to 70 percent of the time, but one key characteristic is that **backwards** branches (going to a PC smaller than the instruction itself) are taken more than 90 percent of the time, while branches that are taken forward are only taken about 50 percent of the time. This is because **backwards branches are almost always loops**, and thus it's very common to run through those loops many times, repeatedly returning to the beginning. And some ISAs actually let us say whether a branch instruction is preferred, but we don't actually need that functionality for improving CPI for us. Remembering that JAL instructions have us add an immediate to the current PC, we can look at the **highest bit** of the immediate to see whether we're going forward or backwards, because that tells us the sign in two's complement encoding! And in practice, this will end up being about 80 percent accurate, and it gives us a pretty significant improvement.

But we can even do better – modern processors use **dynamic branch prediction**, in which we have a predictor make predictions for the next PC, and then we feed back the actual outcome to improve the predictions in the future. This works for the same reason as caching – there are correlations in the PC behavior, much like the temporal and

spatial locality that we learned about for data and instruction access. **Temporal correlation** comes up because the way a branch resolves once tells us about how it may resolve next time, and **spatial correlation** comes up because comparisons in nearby jumps may be related in our code.

Example 204

Here, we'll focus on the simplest possible branch technique, called a **branch target buffer (BTB)**, which we can think of as a tiny cache for branch targets.

BTBs sit inside of the Instruction Fetch stage, and the line index comes from (certain bits of) the PC. BTB lines contain a **tag** and **valid bit**, just like a cache, but what we store is the **predicted target PC**. If there is a hit during an instruction fetch, then we use the value stored in the BTB as the predicted next PC, and otherwise we use PC + 4. So whenever we encounter a jump or a taken branch, we initialize an entry in the BTB (telling us about the target). Because the BTB sits in an early stage in our pipelined processor, it can tell us the next PC immediately, and later on in the Execute stage, we figure out whether the BTB is right or wrong (and update the prediction accordingly). And the performance impact of this is much lower, as long as most of our predictions are correct, because we're continually referencing the BTB instead of waiting for calculations in the Decode or Execute stages.

Looking at the BTB implementation details, there are a few ways keys in which BTBs are different from caches – for example, **it's functionally okay if we get an incorrect next PC**, because we're just making a prediction. So there are ways to simplify the design of a BTB to make it cheaper and faster without affecting our processor's clock cycle: we can make tags small (2 or 3 bits) or not even use tags at all, store only a subset of the target PC bits (filling in the rest from the current PC, because most branches and jumps go to nearby locations anyway), or not even store valid bits at all. And even BTBs with about 8 entries or so are very effective in practice!

Looking at the implementation details, each update to the BTB requires the current and next PC, as well as whether the jump or branch was actually taken. To actually implement a BTB, we need a predict method, which gives us a predicted next PC given some input current PC. This will help us on the last part of our design project, and we can even do better with a simple trick:

Example 205

Consider a loop which runs for many iterations (but not infinitely many), such as the following code:

```
loop:
addi a1, a1, -1
bnez a1, loop
```

Let's think about how our current BTB would react to a loop like this. We make one misprediction whenever we perform a loop exit (we'll expect that we should still take the branch), and we also make a misprediction at the beginning of the next iteration of the loop (we'll expect not to take the branch, because we exited the loop last time). So when loops are short and don't take many iterations, incurring two mispredictions can be costly. So there's a way to fix this, which is to **make our predictions a bit more stubborn**: instead of using a single valid bit (telling us whether or not to take the branch), we use two bits in a **saturating counter** that encode whether we have a **strong or weak preference**, and we only change the prediction direction after we make two wrong predictions in a row. (Specifically, 11 means we strongly take a branch, 10 means we weakly take it, 01 means we weakly don't take the branch, and 00 means we strongly don't take it. And we add or subtract 1 based on the feedback that we get in execution.) With this improvement, we only make one misprediction per loop, during each loop exit.

Fact 206

And modern processors have trade secrets on how they deal with branches – predictors are specialized to predict lots of different information, like the return address or loop length. And there's a "tournament predictor," which decides which predictor is most accurate in any given situation!

To conclude, let's see how things fit into a recent Skylake processor (the Intel Core i7). We haven't talked very much about the actual hardware, but high-performance processors often have a metal heat sink (for dissipation), and underneath there's a **processor die** (a reflective surface, etched with transistors). Pictures of a chip are often colored to reflect the different components – the Intel Core i7 chip processor has 4 cores, with a 6-wide superscalar out-of-order execution, multi-level branch predictors, three levels of caches, and 19 pipeline stages running at 4 GHz. (Lots of space on the chip is dedicated to the **GPU** and input/output devices.) This chip has 1.7 billion transistors, and we can analyze how the pipeline is laid out by looking at the regularity of the physical picture of the chip! We will find that the execution units (where work is actually done) is only taking up about a sixth of the total area. (But we should note that the area being used here is much larger than for a simple core like RISC-V.) Because each core has such a complicated design, there can be a large overhead for computation, so an increasing trend is that we **build more specialized systems** instead of a general-purpose processor. So an alternate idea if we want to get all the points in the design project, and we're sick of pipelining, is to specialize our processor for sorting.