

Lecture Notes for

Computation Structures

The staff of M.I.T. 6.004

Computation Structures is an introductory course about the design and implementation of digital systems, emphasizing structural principles common to a wide range of technologies.

Starting at the device level, the course develops a hierarchical set of building blocks — logic gates, combinational and sequential circuits, finite-state machines, processors and finally complete systems. Both hardware and software mechanisms are explored through a series of design examples. The problem sets and lab exercises are intended to give students hands-on experience in designing digital systems; students complete the gate-level design for a RISC processor during the lab exercises.

For additional material (textbook-style notes, lecture videos, worksheets, hands-on labs) please visit

<http://computationstructures.org>

Table of Contents

Digital Circuits

L01. Basics of Information	3
L02. The Digital Abstraction	31
L03. CMOS Technology	53
L04. Combinational Logic	76
L05. Sequential Logic	110
L06. Finite State Machines	134
L07. Performance Measures	167
L08. Design Tradeoffs	194

Programmable Architectures

L09. Instruction Set Architectures	220
L10a. Assembly Language	257
L10b. Models of Computation	276
L11. Compilers	290
L12. Procedures & Stacks	324
L13. Building the Beta	349
L14. The Memory Hierarchy	381
L15. Pipelining the Beta	427

Computer Organization

L16. Virtual Memory	479
L17. Virtualizing the Processor	512
L18. Devices & Interrupts	541
L19. Concurrency & Synchronization	570
L20. System-level Communication	598
L21. Parallel Processing	626
L22. Wrap-up	657

L01: Basics of Information

1. What is Information?
2. Quantifying Information
3. Information Conveyed by Data
4. Example: Information Content
5. Probability and Information Content
6. Entropy
7. Meaning of Entropy
8. Encodings
9. Encodings as Binary Trees
10. Fixed-length Encodings
11. Encoding Positive Integers
12. Hexadecimal Notation
13. Encoding Signed Integers
14. Two's Complement Encoding
15. More Two's Complement
16. Variable-length Encodings
17. Example: Variable-length Encoding
18. Huffman's Algorithm
19. Can We Do Better?
20. Error Detection
21. Hamming Distance
22. Hamming Distance and Bit Errors
23. Single-bit Error Detection
24. Parity Check
25. Detecting Multi-bit Errors
26. Error Correction
27. Summary

In order to build circuits that manipulate, transmit or store information, we are going to need some engineering tools to help us determine if we're choosing a good representation for the information — that's the subject of this chapter. We'll study different ways of encoding information as bits and learn the mathematics that help us determine if our encoding is a good one. We'll also look into what we can do if our representation gets corrupted by errors — it would be nice to detect that something bad has happened and possibly even correct the problem.

What is “Information”?

Information, *n.* Data communicated or received that resolves uncertainty about a particular fact or circumstance.

Example: you receive some data about a card drawn at random from a 52-card deck. Which of the following data conveys the most information? The least?

↙ # of possibilities remaining

- 13** A. The card is a heart
- 51** B. The card is not the Ace of spades
- 12** C. The card is a face card (J, Q, K)
- 1** D. The card is the “suicide king”



Let’s start by asking “what is information?” From our engineering perspective, we’ll define information as data communicated or received that resolves uncertainty about a particular fact or circumstance. In other words, after receiving the data we’ll know more about that particular fact or circumstance. The greater the uncertainty resolved by the data, the more information the data has conveyed.

Let’s look at an example: a card has been chosen at random from a normal deck of 52 playing cards. Without any data about the chosen card, there are 52 possibilities for the type of the card. Now suppose you receive one of the following pieces of data about the choice.

- You learn the suit of the card is Heart. This narrows the choice to down to one of 13 cards.
- You learn instead the card is *not* the Ace of Spades. This still leaves 51 cards that it might be.
- You learn instead that the card is a face card, that is, a Jack, Queen or King. So the choice is one of 12 cards.
- You learn instead that the card is the *suicide king*. This is actually a particular card: the King of Hearts where the king is sticking the sword through his head. No uncertainty here! We know exactly what the choice was.

Which of the possible pieces of data conveys the most information? In other words, which data resolves the most uncertainty about the chosen card? Similarly, which data conveys the least amount of information? We’ll answer these questions in the next section.

Quantifying Information

(Claude Shannon, 1948)

Given discrete random variable X

- N possible values: x_1, x_2, \dots, x_N
- Associated probabilities: p_1, p_2, \dots, p_N

Information received when learning that choice was x_i :

$$I(x_i) = \log_2 \left(\frac{1}{p_i} \right)$$

1/p_i is proportional to the uncertainty of choice x_i.

Information is measured in bits (binary digits) = number of 0/1's required to encode choice(s)



Mathematicians like to model uncertainty about a particular circumstance by introducing the concept of a random variable. For our application, we'll always be dealing with circumstances where there are a finite number N of distinct choices, so we'll be using a discrete random variable X that can take on one of the N possible values from the set $\{x_1, x_2, \dots, x_N\}$. The probability that X will take on the value x_1 is given by the probability p_1 , the value x_2 by probability p_2 , and so on. The smaller the probability, the more uncertain it is that X will take on that particular value.

Claude Shannon, in his seminal work on the theory of information, defined the information received when learning that X had taken on the value x_i as

$$I(x_i) = \log_2 \frac{1}{p_i} \text{ bits.} \quad (1)$$

Note that the uncertainty of a choice is inversely proportional its probability, so the term inside of the log is basically the uncertainty of that particular choice. We use the \log_2 to measure the magnitude of the uncertainty in bits where a bit is a quantity that can take on the value 0 or 1. Think of the information content as the number of bits we would require to encode this choice.

Information Conveyed by Data

Even when data doesn't resolve all the uncertainty

$$I(\text{data}) = \log_2 \left(\frac{1}{p_{\text{data}}} \right) \quad \text{e.g., } I(\text{heart}) = \log_2 \left(\frac{1}{13/52} \right) = 2 \text{ bits}$$

Common case: Suppose you're faced with N equally probable choices, and you receive data that narrows it down to M choices. The probability that data would be sent is $M \cdot (1/N)$ so the amount of information you have received is

$$I(\text{data}) = \log_2 \left(\frac{1}{M \cdot (1/N)} \right) = \log_2 \left(\frac{N}{M} \right) \text{ bits}$$

Suppose the data we receive doesn't resolve all the uncertainty. For example, when earlier we received the data that the card was a Heart: some of uncertainty has been resolved since we know more about the card than we did before the receiving the data, but we don't yet know the exact card, so some uncertainty still remains. We can slightly modify Equation (1) as follows

$$I(\text{data}) = \log_2 \frac{1}{p_{\text{data}}} \text{ bits.} \quad (2)$$

In our example, the probability of learning that a card chosen randomly from a 52-card deck is a Heart is $13/52 = 0.25$, the number of Hearts over the total number of choices. So the information content is computed as

$$I(\text{heart}) = \log_2 \frac{1}{p_{\text{heart}}} = \log_2 \frac{1}{0.25} = 2 \text{ bits.}$$

This example is one we encounter often: we receive partial information about N equally-probable choices (each choice has probability $1/N$) that narrows the number of choices down to M . The probability of receiving such information is $M(1/N)$, so the information content is

$$I(N \text{ choices} \rightarrow M \text{ choices}) = \log_2 \frac{1}{M(1/N)} = \log_2 \frac{N}{M} \text{ bits.} \quad (3)$$

Example: Information Content

Examples:

- information in one coin flip:

$$N = 2 \quad M = 1 \quad \text{Info content} = \log_2(2/1) = 1 \text{ bit}$$

- card drawn from fresh deck is a heart:

$$N = 52 \quad M = 13 \quad \text{Info content} = \log_2(52/13) = 2 \text{ bits}$$

- roll of 2 dice:

$$N = 36 \quad M = 1 \quad \text{Info content} = \log_2(36/1) = 5.17$$

.17 bits ???



Let's look at some examples.

- If we learn the result (heads or tails) of a flip of a fair coin, we go from 2 choices to a single choice. So, using our equation, the information received is $\log_2(2/1) = 1$ bit. This makes sense: it would take us one bit to encode which of the two possibilities actually happened, say, "1" for heads and "0" for tails.
- Reviewing the example from earlier, learning that a card drawn from a fresh deck is a Heart gives us $\log_2(52/13) = 2$ bits of information. Again this makes sense: it would take us two bits to encode which of the four possible card suits had turned up.
- Finally consider what information we get from rolling two dice, one red and one green. Each die has six faces, so there are 36 possible combinations. Once we learn the exact outcome of the roll, we've received $\log_2(36/1) = 5.17$ bits of information.

Hmm. What do those fractional bits mean? Our digital system only deals in whole bits! So to encode a single outcome, we'd need to use 6 bits. But suppose we wanted to record the outcome of 10 successive rolls. At 6 bits/roll we would need a total of 60 bits. What this formula is telling us is that we would need not 60 bits, but only 52 bits to unambiguously encode the results. Whether we can come up with an encoding that achieves this lower bound is an interesting question that we'll take up later in this chapter.

Probability & Information Content



data	p_{data}	$\log_2(1/p_{\text{data}})$
a heart	13/52	2 bits
not the Ace of spades	51/52	0.028 bits
a face card (J, Q, K)	12/52	2.115 bits
the “suicide king”	1/52	5.7 bits



Shannon's definition for information content lines up nicely with my intuition: I get more information when the data resolves more uncertainty about the randomly selected card.

To wrap up, let's return to our initial example. Here's a table showing the different choices for the data received, along with the probability of that event and the computed information content.

The results line up nicely with our intuition: the more uncertainty is resolved by the data, the more information we have received. We can use Equation (2) to provide an exact answer to the questions at the end of the first slide. We get the most information when we learn that the card is the suicide King and the least information when we learn that the card is not the Ace of Spades.

Entropy

In information theory, the **entropy** $H(X)$ is the average amount of information contained in each piece of data received about the value of X :

$$H(X) = E(I(X)) = \sum_{i=1}^N p_i \cdot \log_2 \left(\frac{1}{p_i} \right)$$

Example: $X = \{A, B, C, D\}$

<i>choice_i</i>	<i>p_i</i>	<i>log₂(1/p_i)</i>
“A”	1/3	1.58 bits
“B”	1/2	1 bit
“C”	1/12	3.58 bits
“D”	1/12	3.58 bits

$$\begin{aligned} H(X) &= (1/3)(1.58) + \\ &\quad (1/2)(1) + \\ &\quad 2(1/12)(3.58) \\ &= 1.626 \text{ bits} \end{aligned}$$

In the next section we’re going to start our discussion on how to actually engineer the bit encodings we’ll use to encode information, but first we’ll need a way to evaluate the efficacy of an encoding. The *entropy*, $H(X)$, of a discrete random variable X is average amount of information received when learning the value of X :

$$H(X) = E(I(X)) = \sum_i p_i \log_2 \frac{1}{p_i} \tag{4}$$

Shannon followed Boltzmann’s lead in using H , the upper-case variant of the Greek letter η (eta), for “entropy” since E was already used for “expected value,” the mathematicians’ name for “average.” We compute the expected value in the usual way: we take the weighted sum, where the amount of information received when learning of a particular choice i , $\log_2(1/p_i)$ is weighted by the probability of that choice actually happening.

Here’s an example. We have a random variable that can take on one of four values $\{A, B, C, D\}$. The probabilities of each choice are shown in the table, along with the associated information content.

Now we’ll compute the entropy using Equation (4):

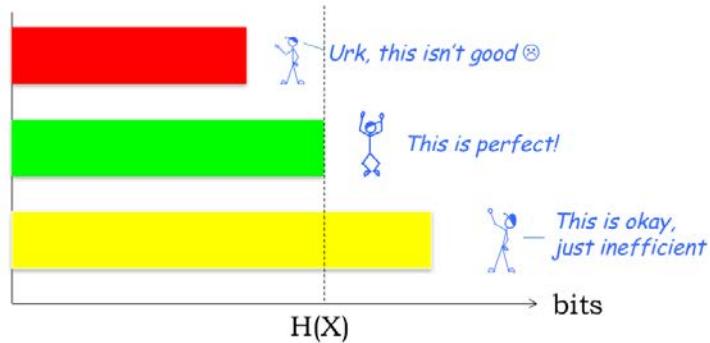
$$\begin{aligned} H(X) &= (1/3)(1.58) + (1/2)(1) + (1/12)(3.58) + (1/12)(3.58) \\ &= 1.626 \text{ bits.} \end{aligned}$$

This is telling us that a clever encoding scheme should, on the average, be able to do better than simply encoding each symbol using 2 bits to represent which of the four possible values is next. Food for thought! We’ll discuss this further in our discussion of variable-length encodings.

Meaning of Entropy

Suppose we have a data sequence describing the values of the random variable X .

Average number of bits used to transmit choice



So, what is the entropy telling us? Suppose we have a sequence of data describing a sequence of values of the random variable X .

If, on the average, we use less than $H(X)$ bits transmit each piece of data in the sequence, we will not be sending enough information to resolve the uncertainty about the values. In other words, the entropy is a lower bound on the number of bits we need to transmit. Getting less than this number of bits wouldn't be good if the goal was to unambiguously describe the sequence of values — we'd have failed at our job!

On the other hand, if we send, on the average, more than $H(X)$ bits to describe the sequence of values, we will not be making the most effective use of our resources, since the same information might have been able to be represented with fewer bits. This okay, but perhaps with some insights we could do better.

Finally, if we send on the average exactly $H(X)$ bits then we'd have the perfect encoding. Alas, perfection is, as always, a tough goal, so most of the time we'll have to settle for getting close.

Encodings

An **encoding** is an *unambiguous* mapping between bit strings and the set of possible data.

Encoding for each symbol				Encoding for "ABBA"
A	B	C	D	
00	01	10	11	00 01 01 00
01	1	000	001	01 1 1 01
X	0	10	11	0 1 1 0  ABBA? ABC? ADA?

Next we turn our attention to encoding data as sequences of 0's and 1's, *i.e.*, a string of bits. An **encoding** is an unambiguous mapping between bit strings and the members of the set of data to be encoded.

For example, suppose we have a set of four symbols $\{A, B, C, D\}$ and we want to use bit strings to encode messages constructed of these symbols, *e.g.*, "ABBA." If we choose to encode the message one character at a time, our encoding would assign a unique bit string to each symbol. The figure above shows some trial encodings.

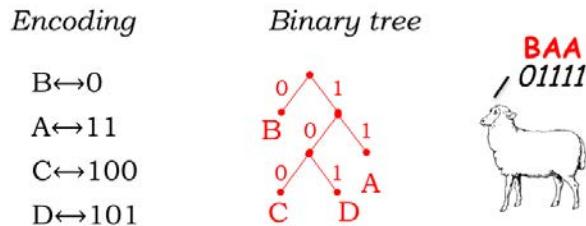
Since we have four symbols, we might choose a unique two-bit string for each: "A" could be "00," B = "01," C = "10," and D = "11," as shown in the first encoding in the figure above. This is called a *fixed-length encoding* since the bit strings used to represent the symbols all have the same length. The encoding for the message "ABBA" would be "00 01 01 00". And we can run the process backwards: given a bit string and the encoding key, we can look up the next bits in the bit string, using the key to determine the symbol they represent. "00" would be decoded as "A," "01" as B and so on.

As shown in the second encoding in the table, we can use a *variable-length encoding*, where the symbols are encoded using bit strings of different lengths. here "A" is encoded as "01," "B" as "1", "C" as "000" and "D" = "001." "ABBA" would be encoded as "01 1 1 01." We'll see that carefully constructed variable-length encodings are useful for the efficient encoding of messages where the symbols occur with different probabilities.

Finally consider the third encoding in the table. We have to be careful that the encoding is unambiguous! Using this encoding, "ABBA" would be encoded as "0 1 1 0." Looking good since that encoding is shorter than either of the previous two encodings. Now let's try to decode this bit string — oops. Using the encoding key, we can unfortunately arrive at several decodings: "ABBA" of course, but also "ADA" or "ABC" depending on how we group the bits. This attempt at specifying an encoding has failed since the message cannot be interpreted unambiguously.

Encodings as Binary Trees

It's helpful to represent an unambiguous encoding as a binary tree with the symbols to be encoded as the leaves. The labels on the path from the root to the leaf give the encoding for that leaf.



Graphically we can represent an unambiguous encoding as a binary tree, labeling the branches from each tree node with “0” and “1,” placing the symbols to be encoded as the leaves of the tree. If you build a binary tree for a proposed encoding and find that there are no symbols labeling interior nodes and exactly one symbol at each leaf, then your encoding is good to go!

For example, consider the encoding shown on the left of the figure. It just takes a second to draw the corresponding binary tree. The symbol “B” is distance 1 from the root of the tree, along an arc labeled “0”. “A” is distance two, and “C” and “D” are distance 3.

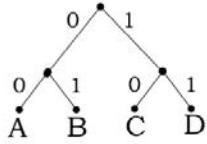
If we receive an encoded message, *e.g.*, “01111,” we can decode it by using successive bits of the encoding to identify a path from the root of tree, descending step-by-step until we come to leaf, then repeating the process starting at the root again, until all the bits in the encoded message have been consumed. So the message from the sheep is decoded as follows:

- “0” takes us from the root to the leaf “B”, which is our first decoded symbol.
- Then “1-1” takes us to “Ak” and
- the next “1-1” results in a second “A.”

The final decoded message, “BAA,” is not totally unexpected, at least from an American sheep.

Fixed-length Encodings

If all choices are **equally likely** (or we have no reason to expect otherwise), then a fixed-length code is often used. Such a code will use at least enough bits to represent the information content.



All leaves have the same depth!
Note that the entropy for N equally-probable symbols is
$$\sum_{i=1}^N \left(\frac{1}{N}\right) \log_2\left(\frac{1}{N}\right) = \log_2(N)$$

Examples:

- 4-bit binary-coded decimal (BCD) digits $\log_2(10)=3.322$
- 7-bit ASCII for printing characters $\log_2(94)=6.555$

If the symbols we are trying to encode occur with equal probability (or if we have no *a priori* reason to believe otherwise), then we'll use a fixed-length encoding, where all leaves in the encoding's binary tree are the same distance from the root. Fixed-length encodings have the advantage of supporting random access, where we can figure out the Nth symbol of the message by simply skipping over the required number of bits. For example, in a message encoded using the fixed-length code shown here, if we wanted to determine the third symbol in the encoded message, we would skip the 4 bits used to encode the first two symbols and start decoding with the 5th bit of message.

Mr. Blue is telling us about the entropy for random variables that have N equally-probable outcomes. In this case, each element of the sum in the entropy formula is simply $(1/N) * \log_2(N)$, and, since there are N elements in the sequence, the resulting entropy is just $\log_2(N)$.

Let's look at some simple examples. In binary-coded decimal, each digit of a decimal number is encoded separately. Since there are 10 different decimal digits, we'll need to use a 4-bit code to represent the 10 possible choices. The associated entropy is $\log_2(10)$, which is 3.322 bits. We can see that our chosen encoding is inefficient in the sense that we'd use more than the minimum number of bits necessary to encode, say, a number with 1000 decimal digits: our encoding would use 4000 bits, although the entropy suggests we *might* be able to find a shorter encoding, say, 3400 bits, for messages of length 1000.

Another common encoding is ASCII, the code used to represent English text in computing and communication. ASCII has 94 printing characters, so the associated entropy is $\log_2(94)$ or 6.555 bits, so we would use 7 bits in our fixed-length encoding for each character.

Encoding Positive Integers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an N-bit number encoded in this fashion is given by the following formula:

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

	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	0	1	1	1	1	1	0	1	0	0	0	0

$$\begin{aligned} V &= 0*2^{11} + 1*2^{10} + 1*2^9 + \dots \\ &= 1024 + 512 + 256 + 128 + 64 + 16 \\ &= 2000 \end{aligned}$$

Smallest number: 0

Largest number: $2^N - 1$

One of the most important encodings is the one we use to represent numbers. Let's start by thinking about a representation for unsigned integers, numbers starting at 0 and counting up from there. Drawing on our experience with representing decimal numbers, *i.e.*, representing numbers in *base 10* using the 10 decimal digits, our binary representation of numbers will use a *base 2* representation using the two binary digits.

The formula for converting an N-bit binary representation of a numeric value into the corresponding integer is shown below — just multiply each binary digit by its corresponding weight in the base-2 representation. For example, here's a 12-bit binary number, with the weight of each binary digit shown above. We can compute its value as $0 * 2^{11}$ plus $1 * 2^{10}$ plus $1 * 2^9$, and so on. Keeping only the non-zero terms and expanding the powers-of-two gives us the sum

$$1024 + 512 + 256 + 128 + 64 + 16$$

which, expressed in base-10, sums to the number 2000.

With this N-bit representation, the smallest number that can be represented is 0 (when all the binary digits are 0) and the largest number is $2^N - 1$ (when all the binary digits are 1). Many digital systems are designed to support operations on binary-encoded numbers of some fixed size, *e.g.*, choosing a 32-bit or a 64-bit representation, which means that they would need multiple operations when dealing with numbers too large to be represented as a single 32-bit or 64-bit binary string.

Hexadecimal Notation

Long strings of binary digits are tedious and error-prone to transcribe, so we usually use a higher-radix notation, choosing the radix so that it's simple to recover the original bits string.

A popular choice is transcribe numbers in base-16, called hexadecimal, where each group of 4 adjacent bits are represented as a single hexadecimal digit.

Hexadecimal - base 16		2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0000	- 0	1000	- 8	0	1	1	1	1	0	1	0	0	0
0001	- 1	1001	- 9										
0010	- 2	1010	- A										
0011	- 3	1011	- B										
0100	- 4	1100	- C										
0101	- 5	1101	- D										
0110	- 6	1110	- E										
0111	- 7	1111	- F										

0b011111010000 = 0x7D0

Long strings of binary digits are tedious and error-prone to transcribe, so let's find a more convenient notation, ideally one where it will be easy to recover the original bit string without too many calculations. A good choice is to use a representation based on a radix that's some higher power of 2, so each digit in our representation corresponds to some short contiguous string of binary bits. A popular choice these days is a radix-16 representation, called hexadecimal or "hex" for short, where each group of 4 binary digits is represented using a single hex digit.

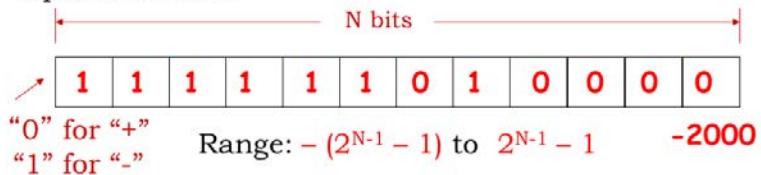
Since there are 16 possible combinations of 4 binary bits, we'll need 16 hexadecimal *digits*: we'll borrow the ten digits "0" through "9" from the decimal representation, and then simply use the first six letters of the alphabet, "A" through "F" for the remaining digits. The translation between 4-bit binary and hexadecimal is shown in the table to the left below.

To convert a binary number to hex, group the binary digits into sets of 4, starting with the least-significant bit (that's the bit with weight 2^0). Then use the table to convert each 4-bit pattern into the corresponding hex digit: "0000" is the hex digit "0", "1101" is the hex digit "D," and "0111" is the hex digit "7". The resulting hex representation is "7D0." To prevent any confusion, we'll use a special prefix "0x" to indicate when a number is being shown in hex, so we'd write "0x7D0" as the hex representation for the binary number "0111 1101 0000." This notation convention is used by many programming languages for entering binary bit strings.

Encoding Signed Integers

We use a signed magnitude representation for decimal numbers, encoding the sign of the number (using “+” and “-”) separately from its magnitude (using decimal digits).

We could adopt that approach for binary representations:



But: two representations for 0 (+0, -0) and we'd need different circuitry for addition and subtraction

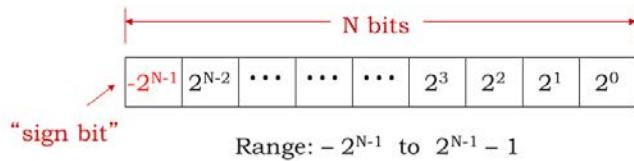
Our final challenge is figuring out how to represent signed integers, *e.g.*, what should be our representation for the number -2000?

In decimal notation, the convention is to precede the number with a “+” or “-” to indicate whether it's positive or negative, usually omitting the “+” to simplify the notation for positive numbers. We could adopt a similar notation — called *signed magnitude* — in binary, by allocating a separate bit at the front of the binary string to indicate the sign, say “0” for positive numbers and “1” for negative numbers. So the signed-magnitude representation for -2000 would be an initial “1” to indicate a negative number, followed by the representation for 2000 (as described on the previous two slides).

However there are some complications in using a signed-magnitude representation. There are two possible binary representations for zero: “+0” and “-0”. This makes the encoding slightly inefficient but, more importantly, the circuitry for doing addition of signed-magnitude numbers is different than the circuitry for doing subtraction. Of course, we're used to that — in elementary school we learned one technique for addition and another for subtraction.

Two's Complement Encoding

In a two's complement encoding, the high-order bit of the N-bit representation has negative weight:



- Negative numbers have “1” in the high-order bit
- Most negative number: $10\dots0000$ $\underline{-2^{N-1}}$
- Most positive number: $01\dots1111$ $\underline{+2^{N-1} - 1}$
- If all bits are 1: $11\dots1111$ $\underline{-1}$
- If all bits are 0: $00\dots0000$ $\underline{0}$

To keep the circuitry simple, most modern digital systems use the two's complement binary representation for signed integers. In this representation, the high-order bit of an N-bit two's complement number has a negative weight, as shown in the figure. Thus all negative numbers have a 1 in the high-order bit and, in that sense, the high-order bit is serving as the *sign bit* — if it's 1, the represented number is negative.

The most negative N-bit number has a 1-bit in the high-order position, representing the value -2^{N-1} . The most positive N-bit number has a 0 in the negative-weight high-order bit and 1's for all the positive-weight bits, representing the value $2^{N-1} - 1$. This gives us the range of possible values, *e.g.*, in an 8-bit two's complement representation, the most negative number is $-2^7 = -128$ and the most positive number is $2^7 - 1 = 127$.

If all N bits are 1, think of that as the sum of the most negative number with the most positive number, *i.e.*, $-2^{N-1} + 2^{N-1} - 1$, which equals -1. And, of course, if all N bits are 0, that's the unique representation of 0.

More Two's Complement

- Let's see what happens when we add the N-bit values for -1 and 1, keeping an N-bit answer:

$$\begin{array}{r} 11\dots1111 \\ +00\dots0001 \\ \hline 0000000 \end{array}$$



Just use ordinary binary addition, even when one or both of the operands are negative. 2's complement is perfect for N-bit arithmetic!

- To compute B-A, we'll just use addition and compute B+(-A). But how do we figure out the representation for -A?

$$A+(-A) = 0 = 1 + -1$$

$$\begin{aligned} -A &= (-1 - A) + 1 \\ &= \sim A + 1 \end{aligned}$$

$$\begin{array}{r} 1 \\ -A_i \\ \hline \sim A_i \end{array}$$



To negate a two's complement value: bitwise complement and add 1.

Let's see what happens when we add the N-bit values for -1 and 1, keeping an N-bit answer. In the rightmost column, 1 plus 1 is 0, carry the 1. In the second column, the carry of 1 plus 1 plus 0 is 0, carry the 1. And so on — the result is all zero's, the representation for 0... perfect! Notice that we just used ordinary binary addition, even when one or both of the operands are negative. Two's complement is perfect for N-bit arithmetic!

To compute B - A, we can just use addition and compute B + (-A). So now we just need to figure out the two's complement representation for -A, given the two's complement representation for A. Well, we know that $A + (-A) = 0$ and using the example above, we can rewrite 0 as $1 + (-1)$. Reorganizing terms, we see that $-A$ equals 1 plus the quantity $(-1) - A$. As we saw above, the two's complement representation for -1 is all 1-bits, so we can write that subtraction as all 1's minus the individual bits of A: A_0, A_1, \dots up to A_{N-1} . If a particular bit A_i is 0, then $1 - A_i = 1$ and if A_i is 1, then $1 - A_i = 0$. So in each column, the result is the bitwise complement of A_i , which we'll write using the C-language bitwise complement operator tilde. So we see that $-A$ equals the bitwise complement of A plus 1. Ta-dah!

To practice your skill with two's complement, try your hand at the following exercises. All you need to remember is how to do binary addition and two's complement negation (which is *bitwise complement and add 1*).

Variable-length Encodings

We'd like our encodings to use bits efficiently:

GOAL: When encoding data we'd like to match the length of the encoding to the information content of the data.

On a practical level this means:

- Higher probability → shorter encodings
- Lower probability → longer encodings

Such encodings are termed **variable-length encodings**.

Fixed-length encodings work well when all the possible choices have the same information content, *i.e.*, all the choices have an equal probability of occurring. If those choices don't have the same information content, we can do better. To see how, consider the expected length of an encoding, computed by considering each x_i to be encoded, and weighting the length of its encoding by p_i , the probability of its occurrence. By "doing better" we mean that we can find encodings that have a shorter expected length than a fixed-length encoding. Ideally we'd like the expected length of the encoding for the x_i to match the entropy $H(X)$, which is the expected information content.

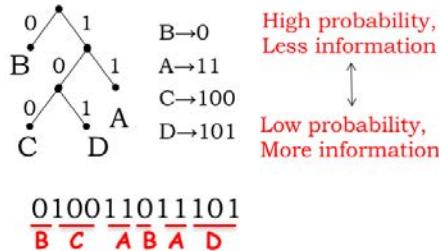
We know that if x_i has a higher probability (*i.e.*, a larger p_i), that is has a smaller information content, so we'd like to use shorter encodings. If x_i has a lower probability, then we'd use a longer encoding.

So we'll be constructing encodings where the x_i may have different length codes — we call these variable-length encodings.

Example

$choice_i$	p_i	encoding
“A”	1/3	11
“B”	1/2	0
“C”	1/12	100
“D”	1/12	101

Entropy: $H(X) = 1.626 \text{ bits}$



Expected length of this encoding:

$$(2)(1/3) + (1)(1/2) + (3)(1/12)(2) = 1.667 \text{ bits}$$

Expected length for 1000 symbols:

- With fixed-length, 2 bits/symbol = 2000 bits
- With variable-length code = 1667 bits
- Lower bound (entropy) = 1626 bits

Here’s an example we’ve seen before. There are four possible choices to encode (A, B, C, and D), each with the specified probability. The table shows a suggested encoding where we’ve followed the advice from the previous slide: high-probability choices that convey little information (*e.g.*, B) are given shorter encodings, while low-probability choices that convey more information (*e.g.*, C or D) are given longer encodings.

Let’s diagram this encoding as a binary tree. Since the symbols all appear as the leaves of the tree, we can see that the encoding is unambiguous. Let’s try decoding the following encoded data. We’ll use the tree as follows: start at the root of the tree and use bits from the encoded data to traverse the tree as directed, stopping when we reach a leaf.

Starting at the root, the first encoded bit is 0, which takes us down the left branch to the leaf B. So B is the first symbol of the decoded data. Starting at the root again, 1 takes us down the right branch, 0 the left branch from there, and 0 the left branch below that, arriving at the leaf C, the second symbol of the decoded data. Continuing on: 11 gives us A, 0 decodes as B, 11 gives us A again, and, finally, 101 gives us D. The entire decoded message is “BCABAD.”

The expected length of this encoding is easy to compute: the length of A’s encoding (2 bits) times its probability (1/3), plus the length of B’s encoding (1 bit) times 1/2, plus the contributions for C and D, each 3 times 1/12. This adds up to 1 and 2/3 bits.

How did we do? If we had used a fixed-length encoding for our four possible symbols, we’d have needed 2 bits each, so we’d need 2000 bits to encode 1000 symbols. Using our variable-length encoding, the expected length for 1000 symbols would be 1667. The lower bound on the number of bits needed to encode 1000 symbols is 1000 times the entropy $H(X)$, which is 1626 bits, so the variable-length code got us closer to our goal, but not quite all the way there.

Could another variable-length encoding have done better? In general, it would be nice to have a systematic way to generate the best-possible variable-length code, and that’s the subject of the next video.

Huffman's Algorithm

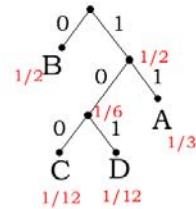
Given a set of symbols and their probabilities, constructs an optimal variable-length encoding.

Huffman's Algorithm:

- Build subtree using 2 symbols with lowest p_i
- At each step choose two symbols/subtrees with lowest p_i , combine to form new subtree
- Result: optimal tree built from the bottom-up

Example:

A = 1/3, B = 1/2, C = 1/12, D = 1/12



Given a set of symbols and their probabilities, Huffman's Algorithm tells us how to construct an optimal variable-length encoding. By “optimal” we mean that, assuming we’re encoding each symbol one-at-a-time, no other variable-length code will have a shorter expected length.

The algorithm builds the binary tree for the encoding from the bottom up. Start by choosing the two symbols with the smallest probability (which means they have highest information content and should have the longest encoding). If anywhere along the way, two symbols have the same probability, simply choose one arbitrarily. In our running example, the two symbols with the lowest probability are C and D.

Combine the symbols as a binary subtree, with one branch labeled “0” and the other “1.” It doesn’t matter which labels go with which branch. Remove C and D from our list of symbols, and replace them with the newly constructed subtree, whose root has the associated probability of 1/6, the sum of the probabilities of its two branches.

Now continue, at each step choosing the two symbols and/or subtrees with the lowest probabilities, combining the choices into a new subtree. At this point in our example, the symbol A has the probability 1/3, the symbol B the probability 1/2 and the C/D subtree probability 1/6. So we’ll combine A with the C/D subtree.

On the final step we only have two choices left: B and the A/C/D subtree, which we combine in a new subtree, whose root then becomes the root of the tree representing the optimal variable-length code. Happily, this is the code we’ve been using all along!

As mentioned above, we can produce a number of different variable-length codes by swapping the “0” and “1” labels on any of the subtree branches. But all those encodings would have the same expected length, which is determined by the distance of each symbol from the root of the tree, not the labels along the path from root to leaf. So all these different encodings are equivalent in terms of their efficiency.

Can We Do Better?

Huffman's Algorithm constructed an optimal encoding... does that mean we can't do better?

To get a more efficient encoding (closer to information content) we need to encode **sequences of choices**, not just each choice individually. This is the approach taken by most file compression algorithms...

AA=1/9, AB=1/6, AC=1/36, AD=1/36
BA=1/6, BB=1/4, BC=1/24, BD=1/24
CA=1/36, CB=1/24, CC=1/144, CD=1/144
DA=1/36, DB=1/24, DC=1/144, DD=1/144

Lookup "LZW" on Wikipedia



Using Huffman's Algorithm on pairs:
Average bits/symbol = 1.646 bits

“Optimal” sounds pretty good! Does that mean we can’t do any better? Well, not by encoding symbols one-at-a-time. But if we want to encode long sequences of symbols, we can reduce the expected length of the encoding by working with, say, pairs of symbols instead of only single symbols. The table below shows the probability of pairs of symbols from our example. If we use Huffman’s Algorithm to build the optimal variable-length code using these probabilities, it turns out the expected length when encoding pairs is 1.646 bits/symbol. This is a small improvement on the 1.667 bits/symbols when encoding each symbol individually. And we’d do even better if we encoded sequences of length 3, and so on.

Modern file compression algorithms use an adaptive algorithm to determine on-the-fly which sequences occur frequently and hence should have short encodings. They work quite well when the data has many repeating sequences, e.g., natural language data where some letter combinations or even whole words occur again and again. Compression can achieve dramatic reductions from the original file size. If you’d like to learn more, look up “LZW” on Wikipedia to read the Lempel-Ziv-Welch data compression algorithm.

Error Detection and Correction

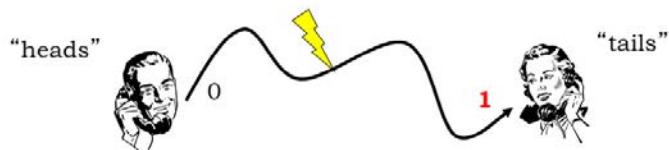
Suppose we wanted to reliably transmit the result of a single coin flip:

Heads: "0"

Tails: "1"



Further suppose that during processing a **single-bit error** occurs, i.e., a single "0" is turned into a "1" or a "1" is turned into a "0".



Now let's think a bit about what happens if there's an error and one or more of the bits in our encoded data gets corrupted. We'll focus on single-bit errors, but much of what we discuss can be generalized to multi-bit errors.

For example, consider encoding the results from some unpredictable event, *e.g.*, flipping a fair coin. There are two outcomes: *heads*, encoded as, say, 0, and *tails* encoded as 1. Now suppose some error occurs during processing, *e.g.*, the data is corrupted while being transmitted from Bob to Alice: Bob intended to send the message *heads*, but the 0 was corrupted and become a 1 during transmission, so Alice receives 1, which she interprets as *tails*. Note that Alice can't distinguish between receiving a message of *heads* that has an error and an uncorrupted message of *tails* — she cannot detect that an error has occurred. So this simple encoding doesn't work very well if there's the possibility of single-bit errors.

Hamming Distance

HAMMING DISTANCE: The number of positions in which the corresponding digits differ in two encodings of the same length.

01~~1~~0010
↓
01~~0~~0110

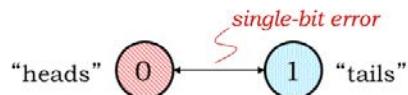


To help with our discussion, we'll introduce the notion of *Hamming distance*, defined as the number of positions in which the corresponding digits differ in two encodings of the same length. For example, here are two 7-bit encodings, which differ in their third and fifth positions, so the Hamming distance between the encodings is 2. If someone tells us the Hamming distance of two encodings is 0, then the two encodings are identical. Hamming distance is a handy tool for measuring how two encodings differ.

Hamming Distance & Bit Errors

The Hamming distance between a valid binary code word and the same code word with a single-bit error is 1.

The problem with our simple encoding is that the two valid code words (“0” and “1”) also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word...



How does this help us think about single-bit errors? A single-bit error changes exactly one of the bits of an encoding, so the Hamming distance between a valid binary code word and the same code word with a single-bit error is 1.

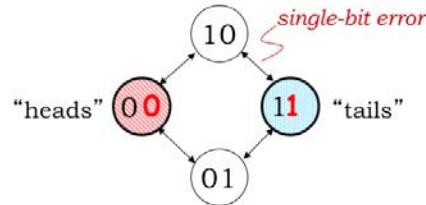
The difficulty with our simple encoding is that the two valid code words (“0” and “1”) also have a Hamming distance of 1. So a single-bit error changes one valid code word into another valid code word. We’ll show this graphically, using an arrow to indicate that two encodings differ by a single bit, *i.e.*, that the Hamming distance between the encodings is 1.

The real issue here is that when Alice receives a 1, she can’t distinguish between an uncorrupted encoding of tails and a corrupted encoding of heads — she can’t detect that an error occurred. Let’s figure how to solve her problem!



Single-bit Error Detection

What we need is an encoding where a single-bit error does *not* produce another valid code word.



A parity bit can be added to any length message and is chosen to make the total number of “1” bits even (aka “even parity”). If $\min \text{HD}(\text{code words}) = 1$, then $\min \text{HD}(\text{code words} + \text{parity}) = 2$.

The insight is to come up with a set of valid code words such that a single-bit error does NOT produce another valid code word. What we need are code words that differ by at least two bits, *i.e.*, we want the minimum Hamming distance between any two code words to be at least 2.

If we have a set of code words where the minimum Hamming distance is 1, we can generate the set we want by adding a parity bit to each of the original code words. There’s *even parity* and *odd parity* — using even parity, the additional parity bit is chosen so that the total number of 1 bits in the new code word are even.

For example, our original encoding for *heads* was 0, adding an even parity bit gives us 00. Adding an even parity bit to our original encoding for *tails* gives us 11. The minimum Hamming distance between code words has increased from 1 to 2.

Parity check = Detect Single-bit errors

- To check for a single-bit error (actually any odd number of errors), count the number of 1s in the received message and if it's odd, there's been an error.

0 1 1 0 0 1 0 1 0 0 1 1 → original word with parity
0 1 1 0 0 **0** 0 1 0 0 1 1 → single-bit error (detected)
0 1 1 0 0 **0** **1** 1 0 0 1 1 → 2-bit error (not detected)

- One can “count” by summing the bits in the word modulo 2 (which is equivalent to XOR’ing the bits together).

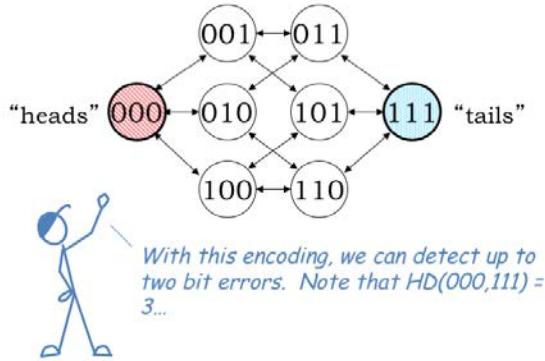
How does this help? Consider what happens when there’s a single-bit error: 00 would be corrupted to 01 or 10, neither of which is a valid code word — aha! we can detect that a single-bit error has occurred. Similarly single-bit errors for 11 would also be detected. Note that the valid code words 00 and 11 both have an even number of 1-bits, but that the corrupted code words 01 or 10 have an odd number of 1-bits. We say that corrupted code words have a *parity error*.

It’s easy to perform a parity check: simply count the number of 1s in the code word. If it’s even, a single-bit error has NOT occurred; if it’s odd, a single-bit error HAS occurred. We’ll see in a couple of chapters that the Boolean function exclusive-or can be used to perform parity checks.

Note that parity won’t help us if there’s an even number of bit errors, where a corrupted code word would have an even number of 1-bits and hence appear to be okay. Parity is useful for detecting single-bit errors; we’ll need a more sophisticated encoding to detect more errors.

Detecting Multi-bit Errors

To detect E errors, we need a minimum Hamming distance of $E+1$ between code words.

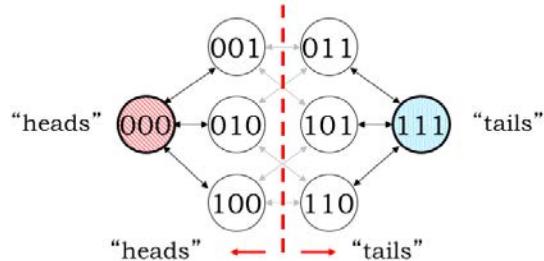


In general, to detect some number E of errors, we need a minimum Hamming distance of $E + 1$ between code words. We can see this graphically below which shows how errors can corrupt the valid code words 000 and 111, which have a Hamming distance of 3. In theory this means we should be able to detect up to 2-bit errors.

Each arrow represents a single-bit error and we can see from the diagram that following any path of length 2 from either 000 or 111 doesn't get us to the other valid code word. In other words, assuming we start with either 000 or 111, we can detect the occurrence of up to 2 errors.

Basically our error detection scheme relies on choosing code words far enough apart, as measured by Hamming distance, so that E errors can't corrupt one valid code word so that it looks like another valid code word.

Single-bit Error Correction



By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of words produced by single-bit errors don't overlap. So assuming at most one error, we can perform **error correction** since we can tell what the valid code was before the error happened.

To **correct** E errors, we need a minimum Hamming distance of $2E+1$ between code words.

Is there any chance we can not only detect a single-bit error but also correct the error to recover the original data? Sure! Here's how.

By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of code words produced by single-bit errors don't overlap. The set of code words produced by corrupting 000 (100, 010, or 001) has no code words in common with the set of code words produced by corrupting 111 (110, 101, or 011). Assuming that at most one error occurred, we can deduce the original code word from whatever code word we receive. For example if we receive 001, we deduce that the original code word was 000 and there has been a single-bit error.

Again we can generalize this insight: if we want to correct up to E errors, the minimum Hamming distance between valid code words must be at least $2E + 1$. For example, to correct single-bit errors we need valid code words with a minimum Hamming distance of 3.

Coding theory is a research area devoted to developing algorithms to generate code words that have the necessary error detection and correction properties. You can take entire courses on this topic! But we'll stop here with our basic insights: by choosing code words far enough apart (as measured by Hamming distance) we can ensure that we can detect and even correct errors that have corrupted our encoded data. Pretty neat!

Summary

- Information resolves uncertainty
- Choices equally probable:
 - N choices down to M $\Rightarrow \log_2(N/M)$ bits of information
 - use fixed-length encodings
 - encoding numbers: 2's complement signed integers
- Choices not equally probable:
 - choice_i with probability p_i $\Rightarrow \log_2(1/p_i)$ bits of information
 - average amount of information = H(X) = $\sum p_i \log_2(1/p_i)$
 - use variable-length encodings, Huffman's algorithm
- To detect E-bit errors: Hamming distance > E
- To correct E-bit errors: Hamming distance > 2E

Next time:

- encoding information electrically
- the digital abstraction
- combinational devices

L02: The Digital Abstraction

1. Encoding Information
2. Electricity to the Rescue
3. Representing Information with Voltage
4. Using Voltages to Encode a Picture
5. Information Processing = Computation
6. Let's Build a System!
7. Why Did Our System Fail?
8. The Digital Abstraction
9. Using Voltages Digitally
10. Combinational Devices
11. A Combinational Digital System
12. Is This a Combination Device?
13. Dealing With Noise
14. Where Does Noise Come From?
15. Noise Margins
16. Voltage Transfer Characteristic
17. VTC Deductions
18. VTC Example
19. Summary

In the previous lecture, we discussed how to encode information as sequences of bits. In this lecture, we turn our attention to finding a useful physical representation for bits, our first step in building devices that can process information.

Concrete Encoding of Information

To this point we've discussed encoding information using bits. But where do bits come from?

If we're going to design a machine that manipulates information, how should that information be physically encoded?

What makes a good bit?

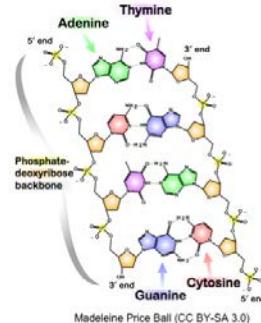
- small, inexpensive (we want a lot of them)
- stable (reliable, repeatable)
- ease and speed of manipulation
(access, transform, combine, transmit, store)

Rosetta Stone



Captmondo (CC BY-SA 3.0)

DNA



Madeleine Price Ball (CC BY-SA 3.0)

So, what makes a good bit, *i.e.*, what properties do we want our physical representation of bits to have?

Well, we'll want a lot of them. We expect to carry billions of bits around with us, *e.g.*, music files. And we expect to have access to trillions of additional bits on the web for news, entertainment, social interactions, commerce — the list goes on and on. So we want bits to be small and inexpensive.

Mother Nature has a suggestion: the chemical encoding embodied in DNA, where sequences of the nucleotides Adenine, Thymine, Guanine and Cytosine form codons that encode genetic information that serve as the blueprint for living organisms. The molecular scale meets our size requirements and there's active research underway on how to use the chemistry of life to perform interesting computations on a massive scale.

We'd certainly like our bits to be stable over long periods of time — once a 0, always a 0! The Rosetta Stone, shown here as part of its original tablet containing a decree from the Egyptian King Ptolemy V, was created in 196 BC and encoded the information needed for archeologists to start reliably deciphering Egyptian hieroglyphics almost 2000 years later. But, the very property that makes stone engravings a stable representation of information makes it difficult to manipulate the information.

Which brings us to the final item on our shopping list: we'd like our representation of bits to make it easy to quickly access, transform, combine, transmit and store the information they encode.

Let's Use Electrical Phenomenon...

Consider using phenomenon associated with charged particles:

voltages	phase
currents	frequency

In this course we'll use **voltages** to encode information. But the best choice depends on the intended application...

Voltage pros:

- easy generation, detection
- lots of engineering knowledge
- potentially ~~low~~ power in steady state
- zero

Voltage cons:

- easily affected by environment
- DC connectivity required?
- R & C effects slow things down



Assuming we don't want to carry around buckets of gooey DNA or stone chisels, how should we represent bits?

With some engineering we can represent information using the electrical phenomenon associated with charged particles. The presence of charged particles creates differences in electrical potential energy we can measure as voltages, and the flow of charged particles can be measured as currents. We can also encode information using the phase and frequency of electromagnetic fields associated with charged particles — these latter two choices form the basis for wireless communication. Which electrical phenomenon is the best choice depends on the intended application.

In this course, we'll use voltages to represent bits. For example, we might choose 0V to represent a 0-bit and 1V to represent a 1-bit. To represent sequences of bits we can use multiple voltage measurements, either from many different wires, or as a sequence of voltages over time on a single wire.

A representation using voltages has many advantages: electrical outlets provide an inexpensive and mostly reliable source of electricity and, for mobile applications, we can use batteries to supply what we need. For more than a century, we've been accumulating considerable engineering knowledge about voltages and currents. We now know how to build very small circuits to store, detect and manipulate voltages. And we can make those circuits run on a very small amount of electrical power. In fact, we can design circuits that require close to zero power dissipation in a steady state if none of the encoded information is changing.

However, a voltage-based representation does have some challenges: voltages are easily affected by changing electromagnetic fields in the surrounding environment. If I want to transmit voltage-encoded information to you, we need to be connected by a wire. And changing the voltage on a wire takes some time, since the timing of the necessary flow of charged particles is determined by the resistance and capacitance of the wire. In modern integrated circuits, these RC time constants are small, but sadly not zero.

We have good engineering solutions for these challenges, so let's get started!

Representing Information with Voltage

Representation of each (x,y) point on a B&W image:

0 volts: BLACK

1 volt: WHITE

0.37 volts: 37% Gray



John Phelan (CC BY 3.0)

How much information at each point?

Suppose we can reliably distinguish voltages that differ by $1/2^N$ volts. Then we can represent N bits of information by voltages in the range 0V to 1V. What are realistic values for N?

Consider the problem of using voltages to represent the information in a black-and-white image. Each (x,y) point in the image has an associated intensity: black is the weakest intensity, white the strongest. An obvious voltage-based representation would be to encode the intensity as a voltage, say 0V for black, 1V for white, and some intermediate voltage for intensities in-between.

First question: how much information is there at each point in the image? The answer depends on how well we can distinguish intensities or, in our case, voltages. If we can distinguish arbitrarily small differences, then there's potentially an infinite amount of information in each point of the image. But, as engineers, we suspect there's a lower-bound on the size of differences we can detect.

To represent the same amount of information that can be represented with N bits, we need to be able to distinguish a total 2^N voltages in the range of 0V to 1V. For example, for N = 2, we'd need to be able to distinguish between four possible voltages. That doesn't seem too hard — an inexpensive volt-meter would let us easily distinguish between 0V, 1/3V, 2/3V and 1V.

In theory, N can be arbitrarily large. In practice, we know it would be quite challenging to make measurements with, say, a precision of 1-millionth of a volt and probably next to impossible if we wanted a precision of 1-billionth of a volt. Not only would the equipment start to get very expensive and the measurements very time consuming, but we'd discover that phenomenon like thermal noise would confuse what we mean by the instantaneous voltage at a particular time.

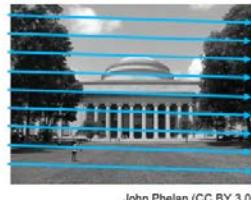
So our ability to encode information using voltages will clearly be constrained by our ability to reliably and quickly distinguish the voltage at particular time.

Using Voltages to Encode a Picture

Representation of a picture:

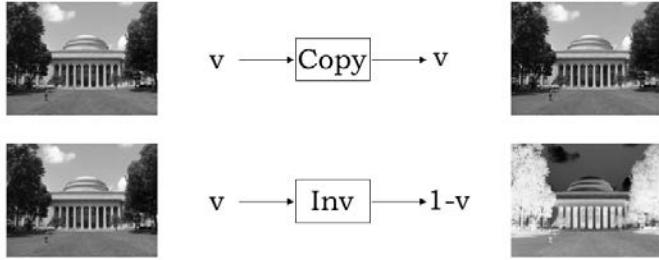
Scan points in some prescribed raster order...

Generate voltage waveform:



To complete our project of representing a complete image, we'll scan the image in some prescribed raster order — left-to-right, top-to-bottom — converting intensities to voltages as we go. In this way, we can convert the image into a time-varying sequence of voltages. This is how the original televisions worked: the picture was encoded as a voltage waveform that varied between the representation for black and that for white. Actually the range of voltages was expanded to allow the signal to specify the end of the horizontal scan and the end of an image, the so-called sync signals. We call this a *continuous waveform* to indicate that it can take on any value in the specified range at a particular point in time.

Information Processing = Computation



Why have processing blocks?

- Pre-packaged functionality: rely on behavior without having to be an analog engineer
- Predictable **composition** of functions
→ Tinker-toy assembly
- Guaranteed behavior:
if components work, system will work!



Wow, rules simple enough for a programmer to follow!

Now let's see what happens when we try to build a system to process this signal.

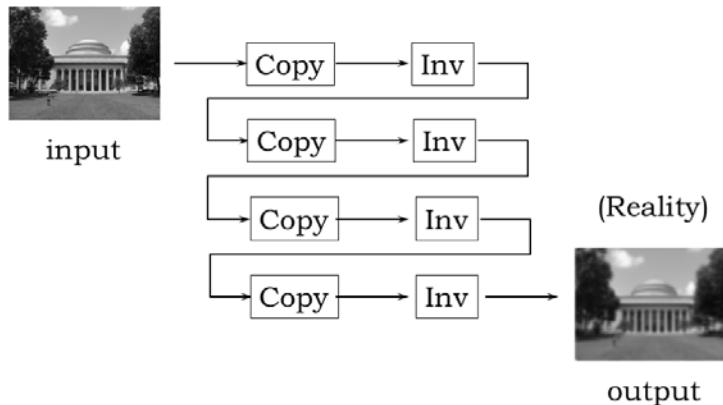
We'll create a system using two simple processing blocks. The COPY block reproduces on its output whatever voltage appears on its input. The output of a COPY block looks the same as the original image. The INVERTING block produces a voltage of $1-V$ when the input voltage is V , i.e., white is converted to black and vice-versa. We get the negative of the input image after passing it through an INVERTING block.

Why have processing blocks? Using pre-packaged blocks is a common way of building large circuits — we can assemble a system by connecting the blocks one to another and reason about the behavior of the resulting system without having to understand the internal details of each block. The pre-packaged functionality offered by the blocks makes them easy to use without having to be an expert analog engineer!

Moreover, we would expect to be able to wire up the blocks in different configurations when building different systems and be able to predict the behavior of each system based on the behavior of each block. This would allow us to build systems like tinker toys, simply by hooking one block to another. Even a programmer who doesn't understand the electrical details could expect to build systems that perform some particular processing task.

The whole idea is that there's a guarantee of predictable behavior: if the components work and we hook them up obeying whatever the rules are for connecting blocks, we would expect the system to work as intended.

Let's Build a System!



So, let's build a system with our COPY and INVERTING blocks. Here's an image processing system using a few instances each block. What do we expect the output image to look like?

Well, the COPY blocks don't change the image and there are an even number of INVERTING blocks, so, in theory, the output image should be identical to the input image.

But in reality, the output image isn't a perfect copy of the input, it's slightly fuzzy — the intensities are slightly off and it looks like sharp changes in intensity have been smoothed out, creating a blurry reproduction of the original. What went wrong?

Why Did Our System Fail?

Why doesn't theory match reality?

1. COPY block doesn't work right
2. INV block doesn't work right
3. Theory is imperfect
4. Reality is imperfect
5. Our system architecture stinks

ANSWER: all of the above!

Noise and inaccuracy are inevitable; we can't reliably reproduce infinite information – we must **design our system to tolerate some amount of error** if it is to process information reliably.

Why doesn't theory match reality?

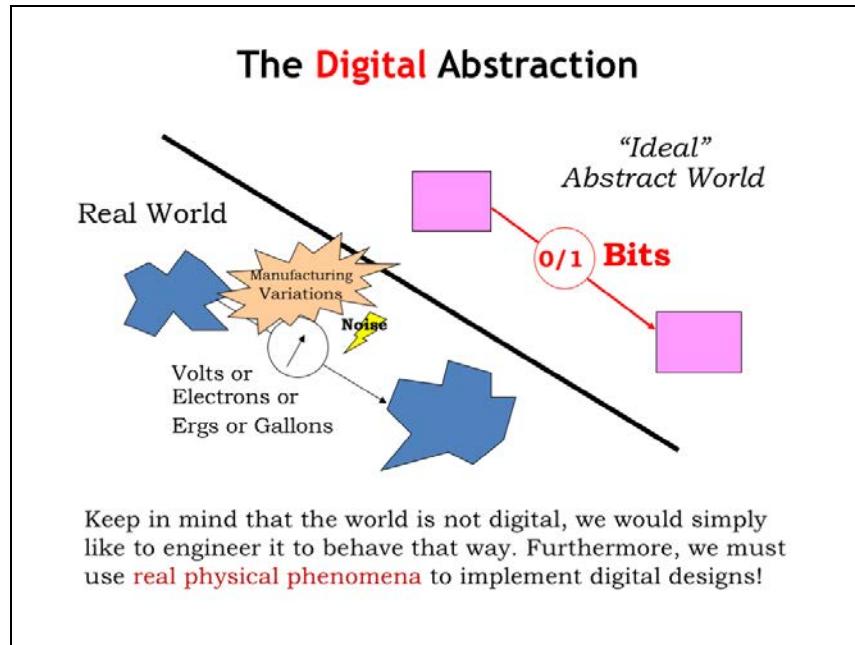
Perhaps the COPY and INVERTING blocks don't work correctly? That's almost certainly true, in the sense that they don't precisely obey the mathematical description of their behavior. Small manufacturing variations and differing environmental conditions will cause each instance of the COPY block to produce not V volts for a V -volt input, but $V+\epsilon$ volts, where ϵ represents the amount of error introduced during processing. Ditto for the INVERTING block.

The difficulty is that in our continuous-value representation of intensity, $V+\epsilon$ is a perfectly correct output value, just not for a V -volt input! In other words, we can't tell the difference between a slightly corrupted signal and a perfectly valid signal for a slightly different image.

More importantly — and this is the real killer — the errors accumulate as the encoded image passes through the system of COPY and INVERTING blocks. The larger the system, the larger the amount of accumulated processing error. This doesn't seem so good: it would be awkward, to say the least, if we had to have rules about how many computations could be performed on encoded information before the results became too corrupted to be usable.

You would be correct if you thought this meant that the theory we used to describe the operation of our system was imperfect. We'd need a very complicated theory indeed to capture all the possible ways in which the output signal could differ from its expected value. Those of us who are mathematically minded might complain that "reality is imperfect." That's going a bit far though. Reality is what it is and, as engineers, we need to build our systems to operate reliably in the real world. So perhaps the real problem lies in how we chose to engineer the system.

In fact, all of the above are true! Noise and inaccuracy are inevitable. We can't reliably reproduce infinite information. We must design our system to tolerate some amount of error if it is to process information reliably. Basically, we need to find a way to notice that errors have been introduced by a processing step and restore the correct values before the errors have a chance to accumulate. How to do that is our next topic.

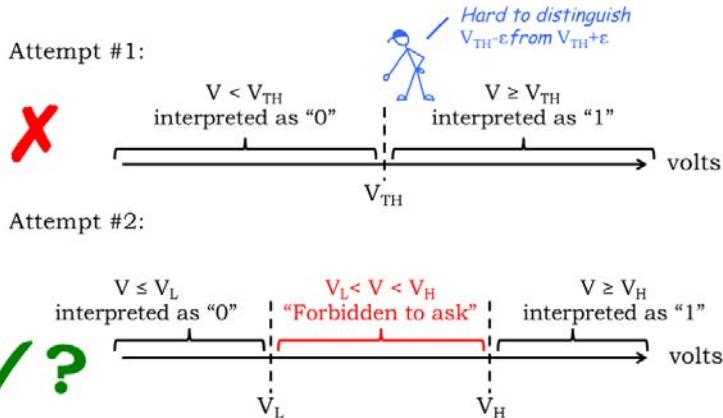


To solve our engineering problem, we will introduce what we'll call the *digital abstraction*. The key insight is to use the continuous world of voltages to represent some small, finite set of values, in our case, the two binary values, 0 and 1. Keep in mind that the world is not inherently digital, we would simply like to engineer it to behave that way, using continuous physical phenomenon to implement digital designs.

As a quick aside, let me mention that there are physical phenomenon that are naturally digital, *i.e.*, that are observed to have one of several quantized values, *e.g.*, the spin of an electron. This came as a surprise to classical physicists who thought measurements of physical values were continuous. The development of quantum theory to describe the finite number of degrees of freedom experienced by subatomic particles completely changed the world of classical physics. We're just now starting to research how to apply quantum physics to computation and there's interesting progress to report on building quantum computers. But for this course, we'll focus on how to use classical continuous phenomenon to create digital systems.

Using Voltages “Digitally”

- Key idea: encode only one bit of information: 2 values “0”, “1”
- Use the same uniform representation convention for *every* component and wire in our digital system



The key idea in using voltages digitally is to have a signaling convention that encodes only one bit of information at time, *i.e.*, one of two values, 0 or 1. We'll use the same uniform representation for every component and wire in our digital system.

It'll take us three attempts to arrive at a voltage representation that solves all the problems. Our first cut is the obvious one: simply divide the range of voltages into two sub-ranges, one range to represent 0, the other range to represent 1. Pick some threshold voltage, V_{th} , to divide the range in two. When a voltage V is less than the threshold voltage, we'll take it to represent a bit value of 0. When a voltage V is greater than or equal to the threshold voltage, it will represent a bit value of 1. This representation assigns a digital value to all possible voltages.

The problematic part of this definition is the difficulty in interpreting voltages near the threshold. Given the numeric value for a particular voltage, it's easy to apply the rules and come up with the corresponding digital value. But determining the correct numeric value accurately gets more time consuming and expensive as the voltage gets closer and closer to the threshold. The circuits involved would have to be made of precision components and run in precisely-controlled physical environments — hard to accomplish when we consider the multitude of environments and the modest cost expectations for the systems we want to build.

So although this definition has an appealing mathematical simplicity, it's not workable on practical grounds. This one gets a big red X.

In our second attempt, we'll introduce two threshold voltages: V_L and V_H . Voltages less than or equal to V_L will be interpreted as 0, and voltages greater than or equal to V_H will be interpreted as 1. The range of voltages between V_L and V_H is called the *forbidden zone*, where we are forbidden to ask for any particular behavior of our digital system. A particular system can interpret a voltage in the forbidden as either a 0 or a 1, and is not even required to be consistent in its interpretation. In fact the system is not required to produce any interpretation at all for voltages in this range.

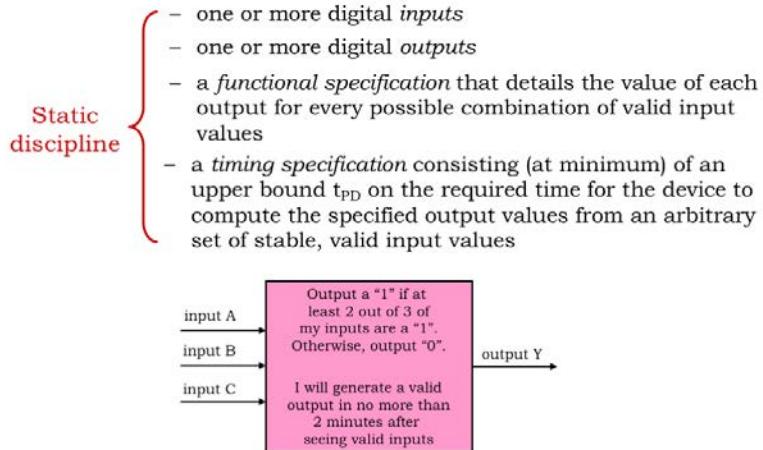
How does this help? Now we can build a quick-and-dirty voltage-to-bit converter, say by using a high-gain op-amp and reference voltage somewhere in the forbidden zone to decide if a given voltage is

above or below the threshold voltage. This reference voltage doesn't have to be super-accurate, so it could be generated with, say, a voltage divider built from low-cost 10%-accurate resistors. The reference could change slightly as the operating temperature varied or the power supply voltage changed, and so on. We only need to guarantee the correct behavior of our converter for voltages below V_L or above V_H .

This representation is pretty promising and we'll tentatively give it a green checkmark for now. After a bit more discussion, we'll need to make one more small tweak before we get to where we want to go.

A Digital Processing Element

A combinational device is a circuit element that has



We're now in a position to define what it means to be a digital processing element. We say a device is a *combinational device* if it meets the following four criteria:

First, it should have digital inputs, by which we mean the device uses our signaling convention, interpreting input voltages below V_L as the digital value 0, and voltages above V_H as the digital value 1.

Second, the device's outputs should also be digital, producing outputs of 0 by generating voltages less than or equal to V_L and outputs of 1 by generating voltages greater than or equal to V_H .

With these two criteria, we should be able to hook the output of one combinational device to the input of another and expect the signals passing between them to be interpreted correctly as 0's and 1's.

Next, a combinational device is required to have a functional specification that details the value of each output for every possible combination of digital values on the inputs.

In the example, the device has three digital inputs, and since each input can take on one of two digital values, there are $2^3 = 8$ possible input configurations. So the functional specification simply has to tell us the value of the output Y when the inputs are 000, and the output when the inputs are 001, and so on, for all 8 input patterns. A simple table with eight rows would do the trick.

Finally, a combinational device has a timing specification that tells us how long it takes for the output of the device to reflect changes in its input values. At a minimum, there must be a specification of the propagation delay, called t_{PD} , that is an upper bound on the time from when the inputs reach stable and valid digital values, to when the output is guaranteed to have a stable and valid output value.

Collectively, we call these four criteria the *static discipline*, which must be satisfied by all combinational devices.

A Combinational Digital System

A set of interconnected elements is a combinational device if

- each circuit element is combinational
- every input is connected to exactly one output or to some vast supply of constant 0's and 1's
- the circuit contains no directed cycles



In order to build larger combinational systems from combinational components, we'll follow the composition rules set forth below.

First, each component of the system must itself be a combinational device.

Second, each input of each component must be connected a system input, or to exactly one output of another device, or to a constant voltage representing the value 0 or the value 1.

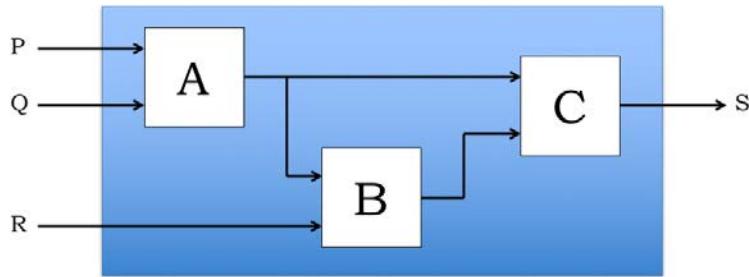
Finally, the interconnected components cannot contain any directed cycles, *i.e.*, paths through the system from its inputs to its outputs will only visit a particular component at most once.

Our claim is that systems built using these composition rules will themselves be combinational devices. In other words, we can build big combinational devices out of combinational components. Unlike our flaky analog system from the start of the chapter, the system can be of any size and still be expected to obey the static discipline.

Why is this true?

Is This a Combinational Device?

A, B and C are combinational devices. Is the following circuit a combinational device?



- Does it have digital inputs?
- Does it have digital outputs?
- Can you derive a functional description?
- Can you derive a t_{PD} ?

To see why the claim is true, consider the following system built from the combinational devices A, B and C. Let's see if we can show that the overall system, as indicated by the containing blue box, will itself be combinational. We'll do this by showing that the overall system does, in fact, obey the static discipline.

First, does the overall system have digital inputs? Yes! The system's inputs are inputs to some of the component devices. Since the components are combinational, and hence have digital inputs, the overall system has digital inputs. In this case, the system is inheriting its properties from the properties of its components.

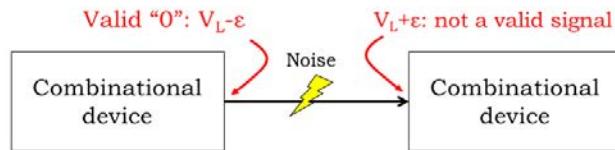
Second, does the overall system have digital outputs? Yes, by the same reasoning: all the system's outputs are connected to one of the components and since the components are combinational, the outputs are digital.

Third, can we derive a functional specification for the overall system, *i.e.*, can we specify the expected output values for each combination of digital input values? Yes, we can by incrementally propagating information about the current input values through the component modules. In the example shown, since A is combinational, we can determine the value on its output given the value on its inputs by using A's functional specification. Now we know the values on all of B's inputs and can use its functional specification to determine its output value. Finally, since we've now determined the values on all of C's inputs, we can compute its output value using C's functional specification. In general, since there are no cycles in the circuit, we can determine the value of every internal signal by evaluating the behavior of the combinational components in an order that's determined by the circuit topology.

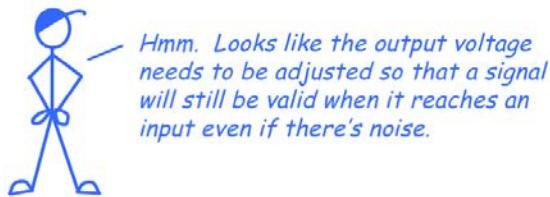
Finally, can we derive the system's propagation delay, t_{PD} , using the propagation delays of the components? Again, since there are no cycles, we can enumerate the finite-length paths from system inputs to system outputs. Then, we can compute the t_{PD} along a particular path by summing the t_{PDs} of the components along the path. The t_{PD} of the overall system will be the maximum of the path t_{PDs} considering all the possible paths from inputs to outputs, *i.e.*, the t_{PD} of the longest such path.

So the overall system does in fact obey the static discipline and so it is indeed a combinational device. Pretty neat — we can use our composition rules to build combinational devices of arbitrary complexity.

Will This System Work?



Upstream device transmits a signal at $V_L - \epsilon$, a valid "0". Noise on the connecting wire causes the downstream device to receive $V_L + \epsilon$, a signal in the forbidden zone.



There's one more issue we need to deal with before finalizing our signaling specification. Consider the following combinational system where the upstream combinational device on the left is trying to send a digital 0 to the downstream combinational device on right. The upstream device is generating an output voltage just slightly below V_L , which, according to our proposed signaling specification, qualifies as the representation for a digital 0.

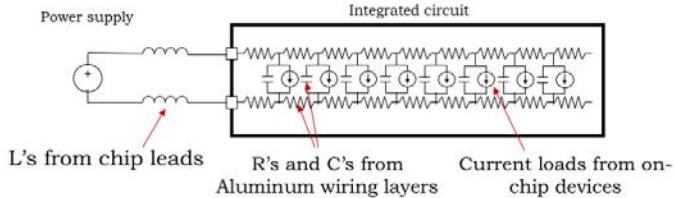
Now suppose some electrical noise slightly changes the voltage on the wire so that the voltage detected on the input of the downstream device is slightly above V_L , i.e., the received signal no longer qualifies as a valid digital input and the combinational behavior of the downstream device is no longer guaranteed.

Oops, our system is behaving incorrectly because of some small amount of electrical noise. Just the sort of flaky behavior we are hoping to avoid by adopting a digital systems architecture.

One way to address the problem is to adjust the signaling specification so that outputs have to obey tighter bounds than the inputs, the idea being to ensure that valid output signals can be affected by noise without becoming invalid input signals.

Where Does Noise Come From?

- Parasitic resistance, inductance, capacitance
 - IR drop, $L(dI/dt)$ drop, LC ringing from current steps



- Imprecision of component values
 - Manufacturing variations, allowable tolerances
- Environmental effects
 - External EM fields, temperature variations, etc.
- ...

Can we avoid the problem altogether by somehow avoiding noise? A nice thought, but not a goal that we can achieve if we're planning to use electrical components. Voltage noise, which we'll define as variations away from nominal voltage values, comes from a variety of sources.

- Noise can be caused by electrical effects such as IR drops in conductors due to Ohm's law, capacitive coupling between conductors, and $L(dI/dt)$ effects caused by inductance in the component's leads and changing currents.
- Voltage deviations can be caused manufacturing variations in component parameters from their nominal values that lead to small differences in electrical behavior device-to-device.
- Voltages can be effected by environmental factors such as thermal noise or voltage effects from external electromagnetic fields.

The list goes on! Note that in many cases, noise is caused by normal operation of the circuit or is an inherent property of the materials and processes used to make the circuits, and so is unavoidable. However, we can predict the magnitude of the noise and adjust our signaling specification appropriately — let's see how this would work.

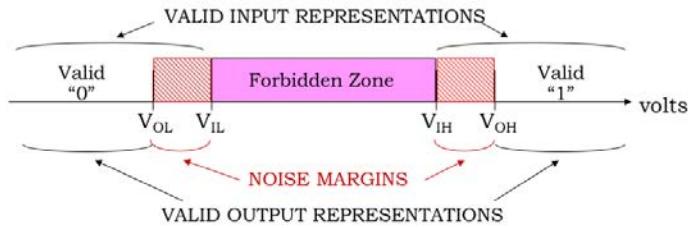
Needed: Noise Margins!

Proposed fix: separate specifications for inputs and outputs

•digital output: “0” $\leq V_{OL}$, “1” $\geq V_{OH}$

•digital input: “0” $\leq V_{IL}$, “1” $\geq V_{IH}$

• $V_{OL} < V_{IL} < V_{IH} < V_{OH}$



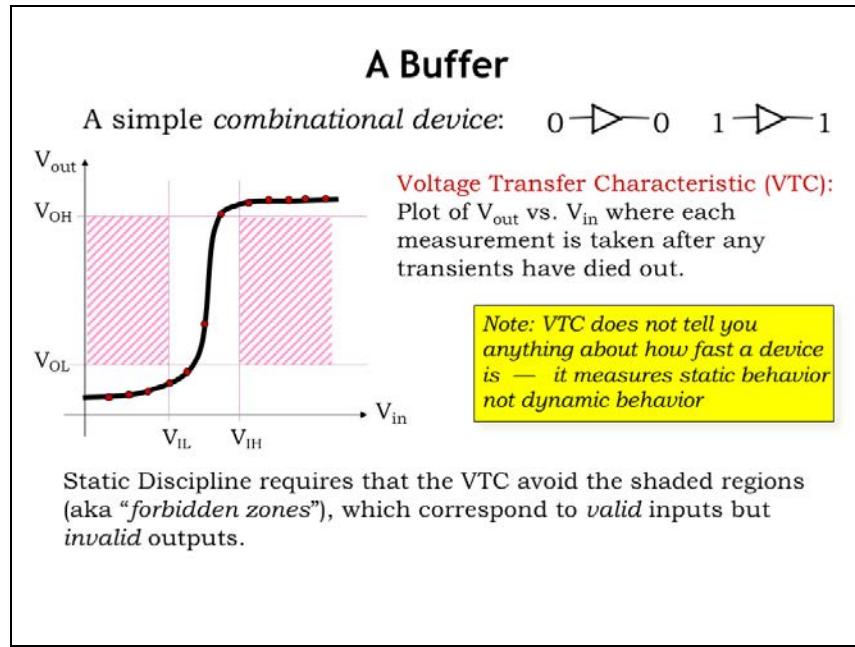
A combinational device accepts marginal inputs and provides unquestionable outputs (to leave room for noise).

Our proposed fix to the noise problem is to provide separate signaling specifications for digital inputs and digital outputs. To send a 0, digital outputs must produce a voltage less than or equal to V_{OL} and to send a 1, produce a voltage greater than or equal to V_{OH} . So far this doesn't seem very different than our previous signaling specification...

The difference is that digital inputs must obey a different signaling specification. Input voltages less than or equal to V_{IL} must be interpreted as a digital 0 and input voltages greater than or equal to V_{IH} must be interpreted as a digital 1.

The values of these four signaling thresholds are chosen to satisfy the constraints shown here. Note that V_{IL} is strictly greater than V_{OL} and V_{IH} is strictly less than V_{OH} . The gaps between the input and output voltage thresholds are called the *noise margins*. The noise margins tell us how much noise can be added to a valid 0 or a valid 1 output signal and still have the result interpreted correctly at the inputs to which it is connected. The smaller of the two noise margins is called the *noise immunity* of the signaling specification. Our goal as digital engineers is to design our signaling specifications to provide as much noise immunity as possible.

Combinational devices that obey this signaling specification work to remove the noise on their inputs before it has a chance to accumulate and eventually cause signaling errors. The bottom line: digital signaling doesn't suffer from the problems we saw in our earlier analog signaling example!



Let's make some measurements using one of the simplest combinational devices: a buffer. A buffer has a single input and single output, where the output will be driven with the same digital value as the input after some small propagation delay. This buffer obeys the static discipline — that's what it means to be combinational — and uses our revised signaling specification that includes both low and high noise margins.

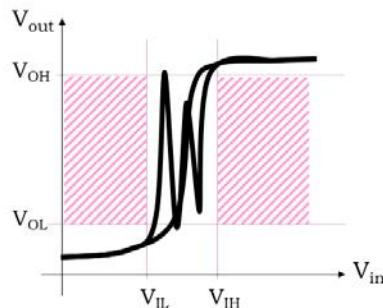
The measurements will be made by setting the input voltage to a sequence of values ranging from 0V up to the power supply voltage. After setting the input voltage to a particular value, we'll wait for the output voltage to become stable, *i.e.*, we'll wait for the propagation delay of the buffer. We'll plot the result on a graph with the input voltage on the horizontal axis and the measured output voltage on the vertical axis. The resulting curve is called the voltage transfer characteristic of the buffer. For convenience, we've marked our signal thresholds on the two axes.

Before we start plotting points, note that the static discipline constrains what the voltage transfer characteristic must look like for any combinational device. If we wait for the propagation delay of the device, the measured output voltage must be a valid digital value if the input voltage is a valid digital value — “*valid in, valid out*.” We can show this graphically as shaded forbidden regions on our graph. Points in these regions correspond to valid digital input voltages but invalid digital output voltages. So if we're measuring a legal combinational device, none of the points in its voltage transfer characteristic will fall within these regions.

Okay, back to our buffer: setting the input voltage to a value less than the low input threshold V_{IL} , produces an output voltage less than V_{OL} , as expected — a digital 0 input yields a digital 0 output. Trying a slightly higher but still valid 0 input gives a similar result. Note that these measurements don't tell us anything about the speed of the buffer, they are just measuring the static behavior of the device, not its dynamic behavior.

If we proceed to make all the additional measurements, we get the voltage transfer characteristic of the buffer, shown as the black curve on the graph. Notice that the curve does not pass through the shaded regions, meeting the expectations we set out above for the behavior of a legal combinational device.

Voltage Transfer Characteristic



- 1) Note the VTC can do anything when $V_{IL} < V_{IN} < V_{IH}$.
- 2) Note that the center white region is taller than it is wide ($V_{OH} - V_{OL} > V_{IH} - V_{IL}$). Net result: combinational devices must have **GAIN > 1** and be **NONLINEAR**.

There are two interesting observations to be made about voltage transfer characteristics.

Let's look more carefully at the white region in the center of the graph, corresponding to input voltages in the range V_{IL} to V_{IH} . First note that these input voltages are in the forbidden zone of our signaling specification and so a combinational device can produce any output voltage it likes and still obey the static discipline, which only constrains the device's behavior for *valid* inputs.

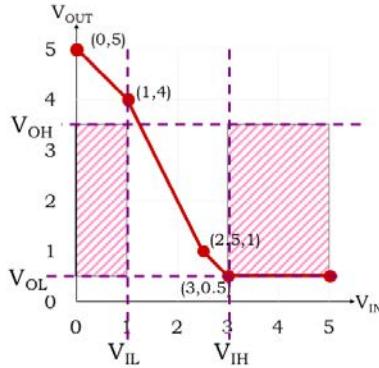
Second, note that the center white region bounded by the four voltage thresholds is taller than it is wide. This is true because our signaling specification has positive noise margins, so $V_{OH} - V_{OL}$ is strictly greater than $V_{IH} - V_{IL}$. Any curve passing through this region — as the VTC must — has to have some portion where the magnitude of the slope of the curve is greater than 1. At the point where the magnitude of the slope of the VTC is greater than one, note that a small change in the input voltage produces a larger change in the output voltage. That's what it means when the magnitude of the slope is greater than 1. In electrical terms, we would say the device as a gain greater than 1 or less than -1, where we define gain as the change in output voltage for a given change in input voltage.

If we're considering building larger circuits out of our combinational components, any output can potentially be wired to some other input. This means the range on the horizontal axis (V_{IN}) has to be the same as the range on the vertical axis (V_{OUT}), *i.e.*, the graph of VTC must be square and the VTC curve fits inside the square. In order to fit within the square bounds, the VTC must change slope at some point since we know from above there must be regions where the magnitude of the slope is greater than 1 and it can't be greater than 1 across the whole input range. Devices that exhibit a change in gain across their operating range are called nonlinear devices.

Together these observations tell us that we cannot use only linear devices such as resistors, capacitors and inductors, to build combinational devices. We'll need nonlinear devices with gain > 1 . Finding such devices is the subject of the next chapter.

Can This Be a Combinational Inverter?

Suppose that you measured the voltage transfer curve of the device shown below. Can we find a signaling specification that would allow this device to be a combinational inverter?



The device must be able to actually produce the desired output level. Thus, V_{OL} can be no lower than 0.5 V.

Try $V_{OL} = 0.5$ V

V_{IH} must be high enough to produce V_{OL}

Try $V_{IH} = 3$ V

Now, find noise margin N and compute

$$V_{OH} = V_{IH} + N$$

$$V_{IL} = V_{OL} + N$$

Then verify that $V_{OUT} \geq V_{OH}$ when $V_{IN} \leq V_{IL}$.

Try $N = 0.5$ V

Device is a combinational inverter when $V_{OL}=0.5$, $V_{IL}=1$, $V_{IH}=3$, $V_{OH}=3.5$

Let's look at a concrete example. This graph shows the voltage transfer characteristic for a particular device and we're wondering if we can use this device as a combinational inverter. In other words, can we pick values for the voltage thresholds V_{OL} , V_{IL} , V_{IH} and V_{OH} so that the shown VTC meets the constraints imposed on a combinational device?

An inverter outputs a digital 1 when its input is a digital 0 and vice versa. In fact this device does produce a high output voltage when its input voltage is low, so there's a chance that this will work out.

The lowest output voltage produced by the device is 0.5V, so if the device is to produce a legal digital output of 0, we have to choose V_{OL} to be at least 0.5V.

We want the inverter to produce a valid digital 0 whenever its input is valid digital 1. Looking at the VTC, we see that if the input is higher than 3V, the output will be less than or equal to V_{OL} , so let's set V_{IH} to 3V. We could set it to a higher value than 3V, but we'll make it as low as possible to leave room for a generous high noise margin.

That takes care of two of the four signal thresholds, V_{OL} and V_{IH} . The other two thresholds are related to these two by the noise margin N as shown by these two equations. Can we find a value for N such that $V_{OUT} \geq V_{OH}$ when $V_{IN} \leq V_{IL}$? If we chose $N = 0.5$ V, then the formulas tell us that $V_{IL} = 1$ V and $V_{OH} = 3.5$ V. Plotting these thresholds on the graph and adding the forbidden regions, we see that happily the VTC is, in fact, legal!

So we can use this device as a combinational inverter if we use the signaling specification with $V_{OL} = 0.5$ V, $V_{IL} = 1$ V, $V_{IH} = 3$ V and $V_{OH} = 3.5$ V. We're good to go!

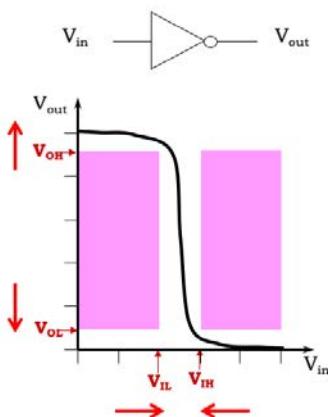
Summary

- Use voltages to encode information
- “Digital” encoding
 - valid voltage levels for representing “0” and “1”
 - forbidden zone avoids mistaking “0” for “1” and vice versa
 - gives rise to notion of signal VALIDITY.
- Noise
 - Want to tolerate real-world conditions: NOISE.
 - Key: tougher standards for output than for input
 - devices must have gain and have a non-linear VTC
- Combinational devices
 - Each logic family has Tinkertoy-set simplicity, modularity
 - predictable composition: “parts work → whole thing works”
 - static discipline
 - digital inputs, outputs; restore marginal input voltages
 - complete functional spec
 - valid inputs lead to valid outputs in bounded time

L03: CMOS Technology

1. Combinational Device Wish List
2. N-Channel MOSFET: Physical View
3. N-Channel MOSFET: Electrical View
4. N-Channel MOSFET I_{DS} vs. V_{DS}
5. FETs Come in Two Flavors
6. CMOS Recipe
7. CMOS Inverter VTC
8. Beyond Inverters
9. CMOS Complements
10. A Pop Quiz!
11. General CMOS Gate Recipe
12. CMOS Gates Are Naturally Inverting
13. CMOS Timing Specifications
14. Propagation Delay
15. Contamination Delay
16. The Combinational Contract
17. Acyclic Combinational Circuits
18. One Last Timing Issue
19. Lenient Gates
20. Summary

Combinational Device Wish List



- ✓ Design our system to tolerate some amount of error
⇒ Add positive noise margins
⇒ VTC: gain > 1 & nonlinearity
- ✓ Lots of gain ⇒ big noise margin
- ✓ Cheap, small
- ✓ Changing voltages will require us to dissipate power, but if no voltages are changing, we'd like zero power dissipation
- ✓ Want to build devices with useful functionality (what sort of operations do we want to perform?)

Let's review our wish list for the characteristics of a combinational device. In the previous lecture we worked hard to develop a voltage-based representation for information that could tolerate some amount of error as the information flowed through a system of processing elements.

We specified four signaling thresholds: V_{OL} and V_{OH} set the upper and lower bounds on voltages used to represent 0 and 1 respectively at the outputs of a combinational device. V_{IL} and V_{IH} served a similar role for interpreting the voltages at the inputs of a combinational device. We also specified that V_{OL} be strictly less than V_{IL} , and termed the difference between these two low thresholds as the low noise margin, the amount of noise that could be added to an output signal and still have the signal interpreted correctly at any connected inputs. For the same reasons we specified that V_{IH} be strictly less than V_{OH} .

We saw the implications of including noise margins when we looked at the voltage transfer characteristic — a plot of V_{OUT} vs. V_{IN} — for a combinational device. Since a combinational device must, in the steady state, produce a valid output voltage given a valid input voltage, we can identify forbidden regions in the VTC, which for valid input voltages identify regions of invalid output voltages. The VTC for a legal combinational device could not have any points that fall within these regions. The center region bounded by the four threshold voltages is narrower than it is high and so any legal VTC has to have a region where its gain is greater than 1 and the overall VTC has to be nonlinear. The VTC shown here is that for a combinational device that serves as an inverter.

If we're fortunate to be using a circuit technology that provides high gain and has output voltages close to the ground and the power supply voltage, we can push V_{OL} and V_{OH} outward towards the power supply rails, and push V_{IL} and V_{IH} inward, with the happy consequence of increasing the noise margins — always a good thing!

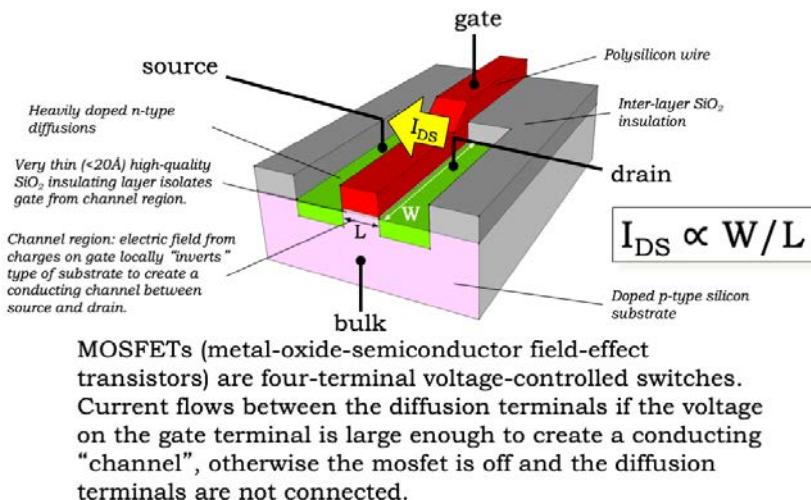
Remembering back to the beginning of Lecture 2, we'll be wanting billions of devices in our digital systems, so each device will have to be quite inexpensive and small.

In today's mobile world, the ability to run our systems on battery power for long periods of time means that we'll want to have our systems dissipate as little power as possible. Of course, manipulating information will necessitate changing voltages within the system and that will cost us some amount of power. But if our system is idle and no internal voltages are changing, we'd like for our system to have zero power dissipation.

Finally, we'll want to be able to implement systems with useful functionality and so need to develop a catalog of the logic computations we want to perform.

Quite remarkably, there is a circuit technology that will make our wishes come true! That technology is the subject of this lecture.

N-Channel MOSFET: Physical View



The star of our show is the metal-oxide-semiconductor field-effect transistor, or MOSFET for short.

Here's a 3D drawing showing a cross-section of a MOSFET, which is constructed from a complicated sandwich of electrical materials as part of an integrated circuit, so called because the individual devices in an integrated circuit are manufactured en-masse during a series of manufacturing steps.

In modern technologies the dimensions of the block shown here are a few 10's of nanometers on a side — that's 1/1000 of the thickness of a thin human hair. This dimension is so small that MOSFETs can't be viewed using an ordinary optical microscope, whose resolution is limited by the wavelength of visible light, which is 400 to 750nm. For many years, engineers have been able to shrink the device dimensions by a factor of 2 every 24 months or so, an observation known as "Moore's Law" after Gordon Moore, one of the founders of Intel, who first remarked on this trend in 1965. Each 50% shrink in dimensions enables integrated circuit (IC) manufacturers to build four times as many devices in the same area as before, and, as we'll see, the devices themselves get faster too! An integrated circuit in 1975 might have had 2500 devices; today we're able to build ICs with two to three billion devices.

Here's a quick tour of what we see in the diagram.

The substrate upon which the IC is built is a thin wafer of silicon crystal which has had impurities added to make it conductive. In this case the impurity was an acceptor atom like Boron, and we characterize the doped silicon as a p-type semiconductor. The IC will include an electrical contact to the p-type substrate, called the *bulk* terminal, so we can control its voltage.

When want to provide electrical insulation between conducting materials, we'll use a layer of silicon dioxide (SiO_2). Normally the thickness of the insulator isn't terribly important, except for when it's used to isolate the gate of the transistor (shown here in red) from the substrate. The insulating layer in that region is very thin so that the electrical field from charges on the gate conductor can easily affect the substrate.

The gate terminal of the transistor is a conductor, in this case, polycrystalline silicon. The gate, the thin oxide insulating layer, and the p-type substrate form a capacitor, where changing the voltage on the gate will cause electrical changes in the p-type substrate directly under the gate. In early manufacturing processes the gate terminal was made of metal, and the term *metal-oxide-semiconductor* (MOS) is referring to this particular structure.

After the gate terminal is in place, donor atoms such as Phosphorous are implanted into the p-type substrate in two rectangular regions on either side of the gate. This changes those regions to an n-type semiconductor, which become the final two terminals of the MOSFET, called the source and the drain. Note that source and drain are usually physically identical and are distinguished by the role they play during the operation of the device, our next topic.

As we'll see in the next slide, the MOSFET functions as a voltage-controlled switch connecting the source and drain terminals of the device. When the switch is conducting, current will flow from the

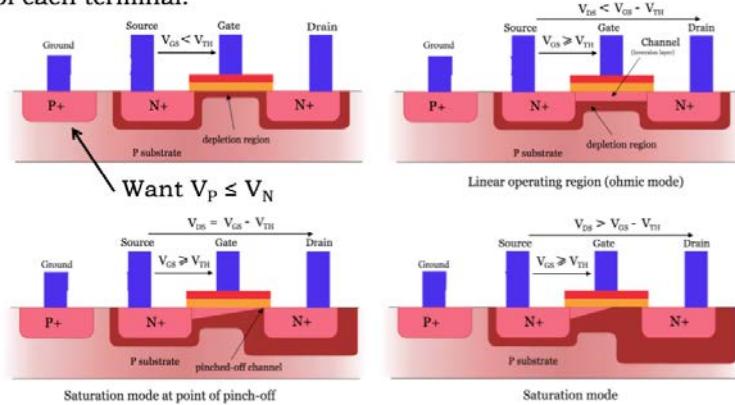
drain to the source through the conducting channel formed as the second plate of the gate capacitor. The MOSFET has two critical dimensions: its length L , which measures the distance the current must cross as it flows from drain to source, and its width W , which determines how much channel is available to conduct current. The current, termed I_{DS} , flowing across the switch is proportional to the ratio of the channel's width to its length.

Typically, IC designers make the length as short as possible — when a news article refers to a 14nm process, the 14nm refers to the smallest allowable value for the channel length. And designers choose the channel width to set the desired amount of current flow. If I_{DS} is large, voltage transitions on the source and drain nodes will be quick, at the cost of a physically larger device.

To summarize: the MOSFET has four electrical terminals: bulk, gate, source, and drain. Two of the device dimensions are under the control of the designer: the channel length, usually chosen to be as small as possible, and the channel width chosen to set the current flow to the desired value. It's a solid-state switch — there are no moving parts and the switch operation is controlled by electrical fields determined by the relative voltages of the four terminals.

N-Channel MOSFET: Electrical View

The four terminals of a Field Effect Transistor (gate, source, drain and bulk) connect to conductors that generate a set of electric fields in the channel region which depend on the relative voltages of each terminal.



Olivier Deleage and Peter Scott (CC BY-SA 3.0)

Now let's look at the electrical view of the MOSFET. Its operation is determined by the voltages of its four terminals.

First we'll label the two diffusion terminals on either side of the gate terminal: our convention is to call the diffusion terminal with the highest voltage potential the *drain* and the other lower-potential terminal the *source*. With this labeling if any current is flowing through the MOSFET switch, it will flow from drain to source.

When the MOSFET is manufactured, it's designed to have a particular threshold voltage, V_{TH} , which tells us when the switch goes from non-conducting/OFF/OPEN to conducting/ON/CLOSED. For the n-channel MOSFET shown here, we'd expect V_{TH} to be around 0.5V in a modern process.

The P+ terminal on the left of the diagram is the connection to the p-type substrate. For the MOSFET to operate correctly, the substrate must always have a voltage less than or equal to the voltage of the source and drain. We'll have specific rules about how to connect up this terminal.

The MOSFET is controlled by the difference between the voltage of the gate, V_G , and the voltage of the source, V_S , which, following the usual terminology for voltages we call V_{GS} , a shortcut for saying $V_G - V_S$.

The first picture shows the configuration of the MOSFET when V_{GS} is less than the MOSFET's threshold voltage. In this configuration, the switch is open or non-conducting, *i.e.*, there is no electrical connection between the source and drain.

When n-type and p-type materials come in physical contact, a depletion region (shown in dark red in the diagram) forms at their junction. This is a region of substrate where the current-carrying electrical particles have migrated away from the junction. The depletion zone serves as an insulating layer between the substrate and source/drain. The width of this insulating layer grows as the voltage of source/drain gets larger relative to the voltage of the substrate. And, as you can see in the diagram, that insulating layer fills the region of the substrate between the source and drain terminals, keeping them electrically isolated.

Now, as V_{GS} gets larger, positive charges accumulate on the gate conductor and generate an electrical field which attracts the electrons in the atoms in the substrate. For a while that attractive force gets larger without much happening, but when it reaches the MOSFET's threshold voltage, the field is strong enough to pull the substrate electrons from the valence band into the conduction band, and the newly mobile electrons will move towards the gate conductor, collecting just under the thin oxide that serves the gate capacitor's insulator.

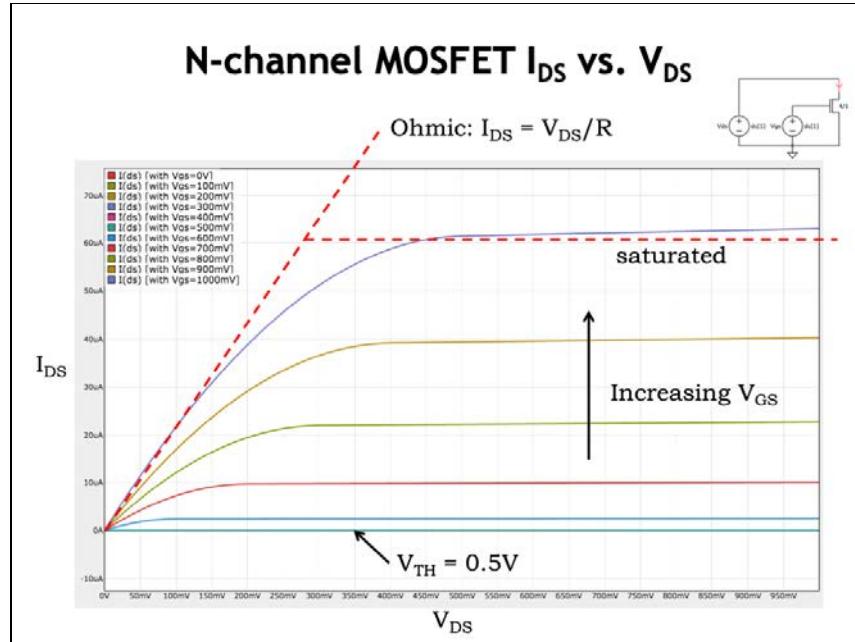
When enough electrons accumulate, the type of the semiconductor changes from p-type to n-type and there's now a channel of n-type material forming a conducting path between the source and drain terminals. This layer of n-type material is called an inversion layer, since its type has been inverted from the original p-type material. The MOSFET switch is now closed or conducting. Current will flow

from drain to source in proportion to V_{DS} , the difference in voltage between the drain and source terminals.

At this point the conducting inversion layer is acting like a resistor governed by Ohm's Law so $I_{DS} = V_{DS}/R$ where R is the effective resistance of the channel. This process is reversible: if V_{GS} falls below the threshold voltage, the substrate electrons drop back into the valence band, the inversion layer disappears, and the switch no longer conducts.

The story gets a bit more complicated when V_{DS} is larger than V_{GS} , as shown in the bottom figures. A large V_{DS} changes the geometry of the electrical fields in the channel and the inversion layer pinches off at the end of the channel near the drain. But with a large V_{DS} , the electrons will tunnel across the pinch-off point to reach the conducting inversion layer still present next to the source terminal.

How does pinch-off affect I_{DS} , the current flowing from drain to source? To see, let's look at some plots of I_{DS} on the next slide.



Okay, this plot has a lot of information, so let's see what we can learn. Each curve is a plot of I_{DS} as a function of V_{DS} , for a particular value of V_{GS} . First, notice that I_{DS} is 0 when V_{GS} is less than or equal to the threshold voltage. The first six curves are all plotted on top of each other along the x-axis.

Once V_{GS} exceeds the threshold voltage I_{DS} becomes non-zero, and increases as V_{GS} increases. This makes sense: the larger V_{GS} becomes, the more substrate electrons are attracted to the bottom plate of the gate capacitor and the thicker the inversion layer becomes, allowing it to conduct more current. When V_{DS} is smaller than V_{GS} , we said the MOSFET behaves like a resistor obeying Ohm's Law. This is shown in the linear portions of the I_{DS} curves at the left side of the plots. The slope of the linear part of the curve is essentially inversely proportional to the resistance of the conducting MOSFET channel. As the channel gets thicker with increasing V_{GS} , more current flows and the slope of the line gets steeper, indicating a smaller channel resistance.

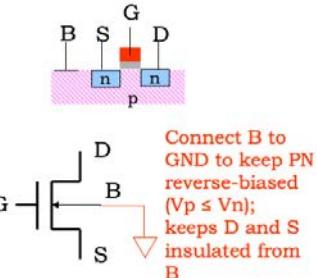
But when V_{DS} gets larger than V_{GS} , the channel pinches off at the drain end and, as we see in on the right side of the I_{DS} plots, the current flow no longer increases with increasing V_{DS} . Instead I_{DS} is approximately constant and the curve becomes a horizontal line. We say that the MOSFET has reached *saturation* where I_{DS} has reached some maximum value.

Notice that the saturated part of the I_{DS} curve isn't quite flat and I_{DS} continues to increase slightly as V_{DS} gets larger. This effect is called channel-length modulation and reflects the fact that the increase in channel pinch-off isn't exactly matched by the increase current induced by the larger V_{DS} .

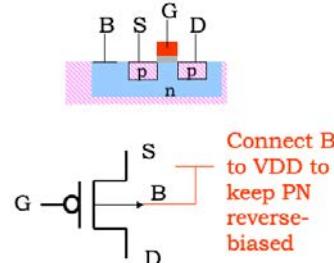
Whew! MOSFET operation is complicated! Fortunately, as designers, we'll be able to use the much simpler mental model of a switch if we obey some simple rules when designing our MOSFET circuits.

FETs Come in Two Flavors

NFET: n-type source/drain diffusions in a p-type substrate. Positive threshold voltage; inversion forms n-type channel



PFET: p-type source/drain diffusions in a n-type substrate. Negative threshold voltage; inversion forms p-type channel.



The use of both NFETs and PFETs – complimentary transistor types – is a key to CMOS (complementary MOS) logic families.

Up to now, we've been talking about MOSFETs built as shown in the diagram on the left: with n-type source/drain diffusions in a p-type substrate. These are called n-channel MOSFETs since the inversion layer, when formed, is an n-type semiconductor. The schematic symbol for an n-channel MOSFET is shown here, with the four terminals arranged as shown. In our MOSFET circuits, we'll connect the bulk terminal of the MOSFET to ground, which will ensure that the voltage of the p-type substrate is always less than or equal to the voltage of the source and drain diffusions.

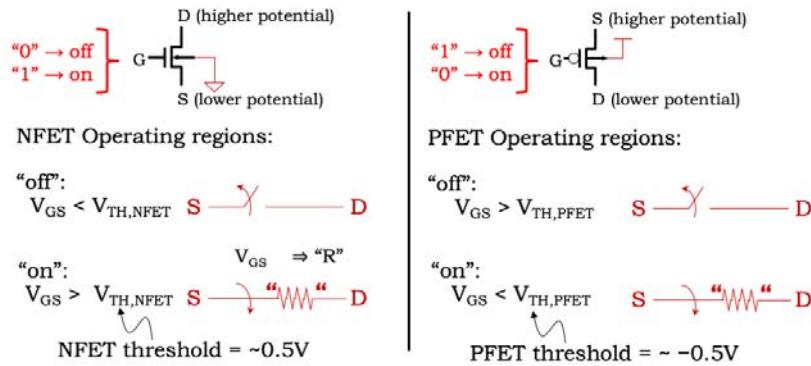
We can also build a MOSFET by flipping all the material types, creating p-type source/drain diffusions in a n-type substrate. This is called a p-channel MOSFET, which also behaves as voltage-controlled switch, except that all the voltage potentials are reversed! As we'll see, control voltages that cause an n-channel switch to be ON will cause a p-channel switch to be OFF and vice-versa.

Using both types of MOSFETs will give us switches that behave in a complementary fashion. Hence the name *complementary MOS*, CMOS for short, for circuits that use both types of MOSFETs. Now that we have our two types of voltage-controlled switches, our next task is to figure out how to use them to build circuits useful for manipulating information encoded as voltages.

CMOS Recipe

If we follow two rules when constructing CMOS circuits, we can model the behavior of the mosfets as simple *voltage-controlled switches*:

- Rule #1: **only use NFETs in pulldown circuits**
- Rule #2: **only use PFETs in pullup circuits**



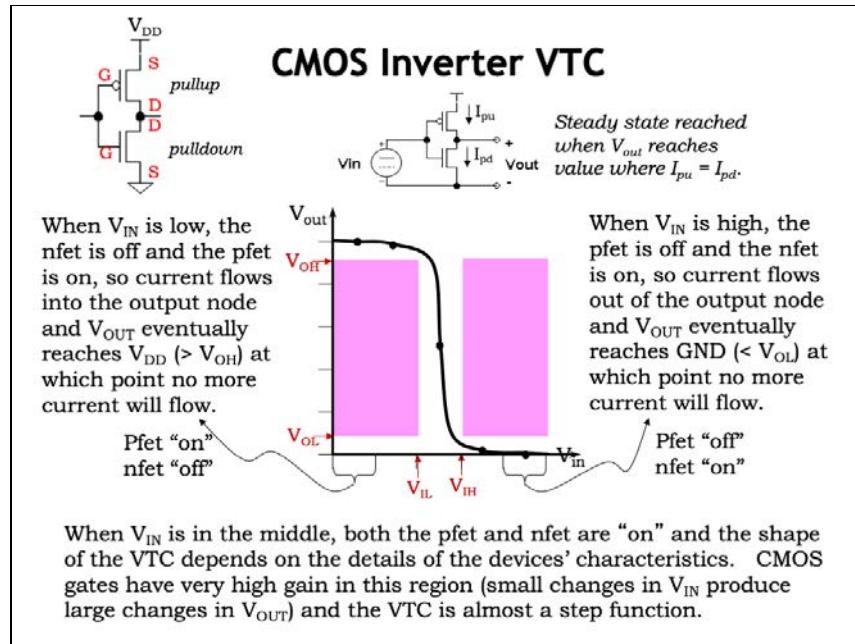
Now that we have some sense about how MOSFETs function, let's use them to build circuits to process our digitally encoded information. We have two simple rules we'll use when building the circuits, which, if they're followed, will allow us to abstract the behavior of the MOSFET as a simple voltage-controlled switch.

The first rule is that we'll only use n-channel MOSFETs, which we'll call NFETs for short, when building pulldown circuits that connect a signaling node to the GND rail of the power supply. When the pulldown circuit is conducting, the signaling node will be at 0V and qualify as a digital 0.

If we obey this rule, NFETs will act switches controlled by V_{GS} , the difference between the voltage of the gate terminal and the voltage of the source terminal. When V_{GS} is lower than the MOSFET's threshold voltage, the switch is OPEN or not conducting and there is no connection between the MOSFET's source and drain terminals. If V_{GS} is greater than the threshold voltage, the switch is ON or conducting and there is a connection between the source and drain terminals. That path has a resistance determined by the magnitude of V_{GS} . The larger V_{GS} , the lower the effective resistance of the switch and the more current that will flow from drain to source. When designing pulldown circuits of NFET switches, we can use the following simple mental model for each NFET switch: if the gate voltage is a digital 0, the switch will be off; if the gate voltage is a digital 1, the switch will be on.

The situation with PFET switches is analogous, except that the potentials are reversed. Our rule is that PFETs can only be used in pullup circuits, used to connect a signaling node to the power supply voltage, which we'll call V_{DD} . When the pullup circuit is conducting, the signaling node will be at V_{DD} volts and qualify as a digital 1. PFETs have a negative threshold voltage and V_{GS} has to be less than the threshold voltage in order for the PFET switch to be conducting. All these negatives can be a bit confusing, but, happily there's a simple mental model we can use for each PFET switch in the pullup circuit: if the gate voltage is a digital 0, the switch will be on; if the gate voltage is a digital 1, the switch will be off — basically the opposite behavior of the NFET switch.

You may be wondering why we can't use NFETs in pullup circuits or PFETs in pulldown circuits. You'll get to explore the answer to this question in one of the lab assignments. Meanwhile, the short answer is that the signaling node will experience degraded signaling levels and we'll lose the noise margins we've worked so hard to create!



Now consider the CMOS implementation of a combinational inverter. If the inverter's input is a digital 0, its output is a digital 1, and vice versa. The inverter circuit consists of a single NFET switch for the pulldown circuit, connecting the output node to GND and a single PFET switch for the pullup circuit, connecting the output to V_{DD} . The gate terminals of both switches are connected to the inverter's input node. The inverter's voltage transfer characteristic is shown in the figure.

When V_{IN} is a digital 0 input, we see that V_{OUT} is greater than or equal to V_{OH} , representing a digital 1 output. Let's look at the state of the pullup and pulldown switches when the input is a digital 0. Recalling the simple mental model for the NFET and PFET switches, a 0-input means the NFET switch is off, so there's no connection between the output node and ground, and the PFET switch is on, making a connection between the output node and V_{DD} . Current will flow through the pullup switch, charging the output node until its voltage reaches V_{DD} . Once both the source and drain terminals are at V_{DD} , there's no voltage difference across the switch and hence no more current will flow through the switch.

Similarly, when V_{IN} is a digital 1, the NFET switch is on and PFET switch is off, so the output is connected to ground and eventually reaches a voltage of 0V. Again, current flow through pulldown switch will cease once the output node reaches 0V.

When the input voltage is in the middle of its range, it's possible, depending on the particular power supply voltage used and the threshold voltage of the MOSFETs, that both the pullup and pulldown circuits will be conducting for a short period of time. That's okay. In fact, with both MOSFET switches on, small changes in the input voltage will produce large changes in the output voltage, leading to the very high gain exhibited by CMOS devices. This in turn will mean we can pick signaling thresholds that incorporate generous noise margins, allowing CMOS devices to work reliably in many different operating environments.

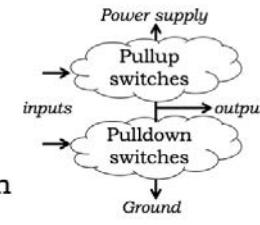
This is our first CMOS combinational logic gate. In the next slide, we'll explore how to build other, more interesting logic functions.

Beyond Inverters: Complementary pullups and pulldowns

Now you know what the “C” in CMOS stands for!

We want **complementary** pullup and pulldown logic, i.e., the pulldown should be “on” when the pullup is “off” and vice versa.

pullup	pulldown	F(inputs)
on	off	driven “1”
off	on	driven “0”
on	on	driven “X”
off	off	no connection



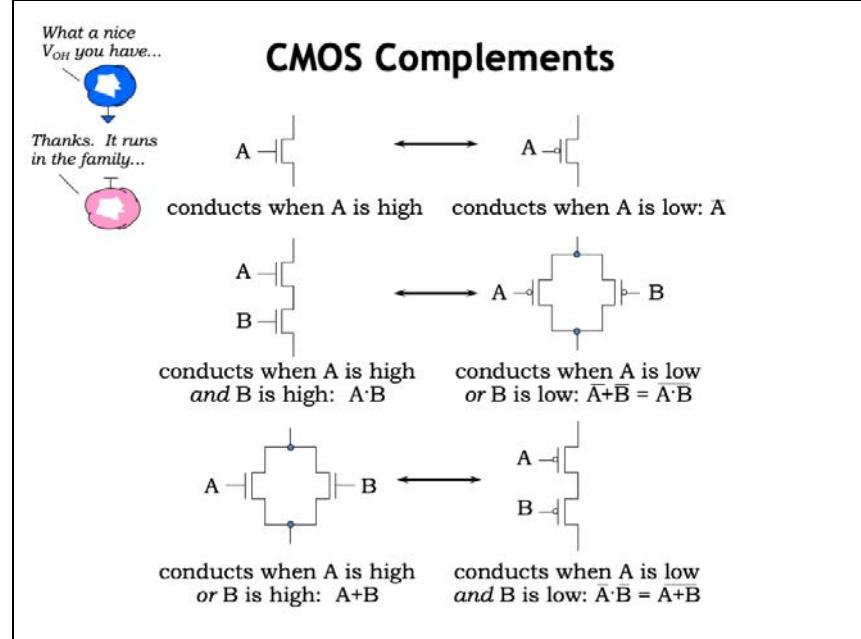
Since there's plenty of capacitance on the output node, when the output becomes disconnected it “remembers” its previous voltage -- at least for a while. The “memory” is the load capacitor's charge. Leakage currents will cause eventual decay of the charge (that's why DRAMs need to be refreshed!).

Now we get to the fun part! To build other logic gates, we'll design complementary pullup and pulldown circuits, hooked up as shown in the diagram on the right, to control the voltage of the output node. *Complementary* refers to property that when one of the circuits is conducting, the other is not. When the pullup circuit is conducting and the pulldown circuit is not, the output node has a connection to V_{DD} and its output voltage will quickly rise to become a valid digital 1 output. Similarly, when the pulldown circuit is conducting and the pullup is not, the output node has a connection to GND and its output voltage will quickly fall to become a valid digital 0 output.

If the circuits are incorrectly designed so that they are not complementary and could both be conducting for an extended period of time, there's a path between V_{DD} and GND and large amounts of short circuit current will flow, a very bad idea. Since our simple switch model won't let us determine the output voltage in this case, we'll call this output value X or unknown.

Another possibility with a non-complementary pullup and pulldown is that neither is conducting and the output node has no connection to either power supply voltage. At this point, the output node is electrically floating and whatever charge is stored by the nodal capacitance will stay there, at least for a while. This is a form of memory and we'll come back to this in a couple of lectures.

For now, we'll concentrate on the behavior of devices with complementary pullups and pulldowns.



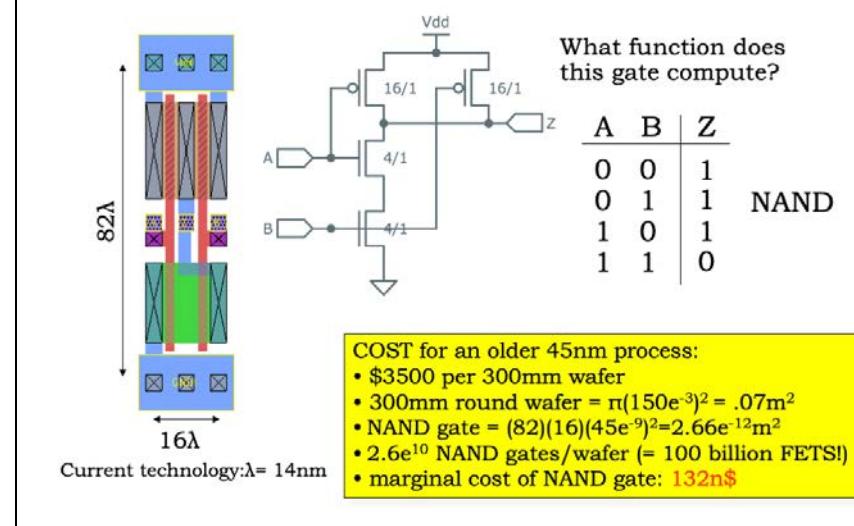
Since the pullup and pulldown circuits are complementary, we'll see there's a nice symmetry in their design. We've already seen the simplest complementary circuit: a single NFET pulldown and a single PFET pullup. If the same signal controls both switches, it's easy to see that when one switch is on, the other switch is off.

Now consider a pulldown circuit consisting of two NFET switches in series. There's a connection through both switches when A is 1 and B is 1. For any other combination of A and B values, one or the other of the switches (or both!) will be off. The complementary circuit to NFET switches in series is PFET switches in parallel. There's a connection between the top and bottom circuit nodes when either of the PFET switches is on, *i.e.*, when A is 0 or B is 0. As a thought experiment consider all possible pairs of values for A and B: 00, 01, 10, and 11. When one or both of the inputs is 0, the series NFET circuit is not conducting and parallel PFET circuit is. And when both inputs are 1, the series NFET circuit is conducting but the parallel PFET circuit is not.

Finally consider the case where we have parallel NFETs and series PFETs. Conduct the same thought experiment as above to convince yourself that when one of the circuits is conducting the other isn't.

Let's put these observations to work when building our next CMOS combinational device.

A Pop Quiz!



In this device, we're using series NFETs in the pulldown and parallel PFETs in the pullup, circuits that we convinced ourselves were complementary in the previous slide. We can build a tabular representation, called a *truth table*, that describes the value of Z for all possible combinations of the input values for A and B.

When A and B are 0, the PFETs are on and the NFETs are off, so Z is connected to V_{DD} and the output of the device is a digital 1. In fact, if either A or B is 0 that continues to be the case, and the value of Z is still 1. Only when both A and B are 1 will both NFETs be on and the value of Z become 0. This particular device is called a NAND gate, short for NOT-AND, a function that is the inverse of the AND function.

Returning to a physical view for a moment, the figure on the left is a bird's eye view, looking down on the surface of the integrated circuit, showing how the MOSFETs are laid out in two dimensions. The blue material represents metal wires with large top and bottom metal runs connecting to V_{DD} and GND. The red material forms the polysilicon gate nodes, the green material the n-type source/drain diffusions for the NFETs and the tan material the p-type source/drain diffusions for the PFETs.

Can you see that the NFETs are connected in series and the PFETs in parallel?

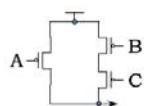
Just to give you a sense of the costs of making a single NAND gate, the yellow box is a back-of-the-envelope calculation showing that we can manufacture approximately 26 billion NAND gates on a single 300mm (that's 12 inches for us non-metric folks) silicon wafer. For the older IC manufacturing process shown here, it costs about \$3500 to buy the materials and perform the manufacturing steps needed to form the circuitry for all those NAND gates. So the final cost is a bit more than 100 nano-dollars per NAND gate. I think this qualifies as both cheap and small!

General CMOS Gate Recipe

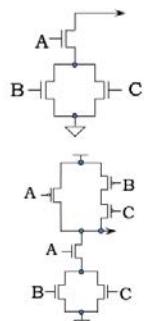
Step 1. Figure out the pullup network that does what you want, e.g.,

$$F = \bar{A} + \bar{B} \cdot \bar{C}$$

(Determine what combination of inputs generates a high output)



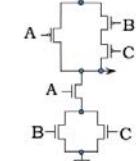
Step 2. Walk the hierarchy replacing nfets with pfets, series subnets with parallel subnets, and parallel subnets with series subnets



Does this recipe work for all logic functions?



Step 3. Combine pfet pullup network from Step 1 with nfet pulldown network from Step 2 to form a fully-complementary CMOS gate.



Using more complicated series/parallel networks of switches, we can build devices that implement more complex logic functions.

To design a more complex logic gate, first figure the series and parallel connections of PFET switches that will connect the gate's output to V_{DD} for the right combination of inputs. In this example, the output F will be 1 when A is 0 OR when both B is 0 AND C is 0. The OR translates into a parallel connection, and the AND translates into a series connection, giving the pullup circuit you see to the right.

To build the complementary pulldown circuit, systematically walk the hierarchy of pullup connections, replacing PFETs with NFETs, series subcircuits with parallel subcircuits, and parallel subcircuits with series subcircuits. In the example shown, the pullup circuit had a switch controlled by A in parallel with a series subcircuit consisting of switches controlled by B and C. The complementary pulldown circuit uses NFETs, with the switch controlled by A in series with a parallel subcircuit consisting of switches controlled by B and C.

Finally combine the pullup and pulldown circuits to form a fully-complementary CMOS implementation. This probably went by a bit quickly, but with practice you'll get comfortable with the CMOS design process.

Mr. Blue is asking a good question: will this recipe work for any and all logic functions? The answer is "no," let's see why.

CMOS Gates Are Naturally Inverting

In a CMOS gate, rising inputs ($0 \rightarrow 1$) lead to falling outputs

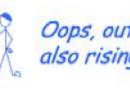
- NFETs go from “off” to “on”
 - pulldown paths connected
 - output may be connected to ground
- PFETs go from “on” to “off”
 - pullup paths disconnected
 - output may be disconnected from V_{DD}

For CMOS gate:

All inputs 0	→ nfets off, pfets on
→ output must be 1	
All inputs 1	→ nfets on, pfets off
→ output must be 0	

A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Corollary: you can't build positive logic, e.g., AND, with one CMOS gate

A=1, B rising...  *Oops, output is also rising!* 

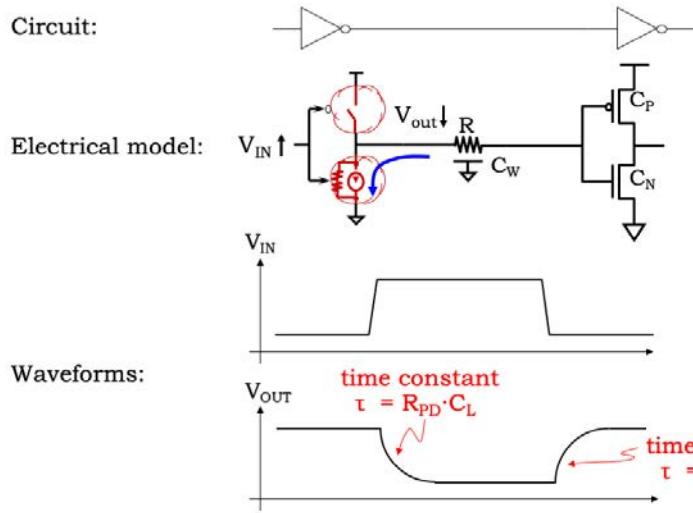
Using CMOS, a single gate (a circuit with one pullup network and one pulldown network) can only implement the so-called inverting functions where rising inputs lead to falling outputs and vice versa. To see why, consider what happens with one of the gate's inputs goes from 0 to 1.

Any NFET switches controlled by the rising input will go from OFF to ON. This may enable one or more paths between the gate's output and GND. And PFET switches controlled by the rising input will go from ON to OFF. This may disable one or more paths between the gate's output and V_{DD} . So if the gate's output changes as the result of the rising input, it must be because some pulldown path was enabled and some pullup path was disabled. In other words, any change in the output voltage due to a rising input must be a falling transition from 1 to 0.

Similar reasoning tells us that falling inputs must lead to rising outputs. In fact, for any non-constant CMOS gate, we know that its output must be 1 when all inputs are 0 (since all the NFETs are off and all the PFETs are on). And vice-versa: if all the inputs are 1, the gate's output must be 0. This means that so-called positive logic can't be implemented with a single CMOS gate.

Look at this truth table for the AND function. It's value when both inputs are 0 or both inputs are 1 is inconsistent with our deductions about the output of a CMOS gate for these combinations of inputs. Furthermore, we can see that when A is 1 and B rises from 0 to 1, the output rises instead of falls. Moral of the story: when you're a CMOS designer, you'll get very good at implementing functionality with inverting logic!

CMOS Timing Specifications



Okay, now that we understand how to build combinational logic gates using CMOS, let's turn our attention to the timing specifications for the gates.

Here's a simple circuit consisting of two CMOS inverters connected in series, which we'll use to understand how to characterize the timing of the inverter on the left. It will be helpful to build an electrical model of what happens when we change V_{IN} , the voltage on the input to the left inverter. If V_{IN} makes a transition from a digital 0 to a digital 1, the PFET switch in the pullup turns off and the NFET switch in pulldown turns on, connecting the output node of the left inverter to GND.

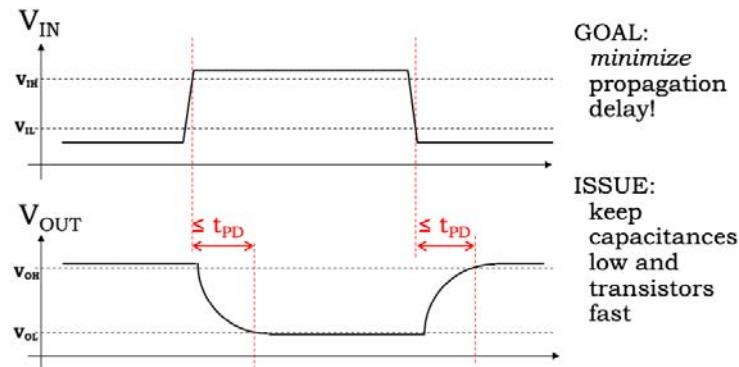
The electrical model for this node includes the distributed resistance and capacitance of the physical wire connecting the output of the left inverter to the input of the right inverter. And there is also capacitance associated with the gate terminals of the MOSFETs in the right inverter. When the output node is connected to GND, the charge on this capacitance will flow towards the GND connection through the resistance of the wire and the resistance of the conducting channel of the NFET pulldown switch. Eventually the voltage on the wire will reach the potential of the GND connection, 0V. The process is much the same for falling transitions on V_{IN} , which cause the output node to charge up to V_{DD} .

Now let's look at the voltage waveforms as a function of time.

The top plot shows both a rising and, later, a falling transition for V_{IN} . We see that the output waveform has the characteristic exponential shape for the voltage of a capacitor being discharged or charged through a resistor. The exponential is characterized by its associated R-C time constant, where, in this case, the R is the net resistance of the wire and MOSFET channel, and the C is the net capacitance of the wire and MOSFET gate terminals. Since neither the input nor output transition is instantaneous, we have some choices to make about how to measure the inverter's propagation delay. Happily, we have just the guidance we need from our signaling thresholds!

Propagation Delay

Propagation delay (t_{PD}): An UPPER BOUND on the delay from **valid inputs** to **valid outputs**.



The propagation delay of a combinational logic gate is defined to be an upper bound on the delay from valid inputs to valid outputs. Valid input voltages are defined by the V_{IL} and V_{IH} signaling thresholds, and valid output voltages are defined by the V_{OL} and V_{OH} signaling thresholds. We've shown these thresholds on the waveform plots.

To measure the delay associated with the rising transition on V_{IN} , first identify the time when the input becomes a valid digital 1, *i.e.*, the time at which V_{IN} crosses the V_{IH} threshold. Next identify the time when the output becomes a valid digital 0, *i.e.*, the time at which V_{OUT} crosses the V_{OL} threshold. The interval between these two time points is the delay for this particular set of input and output transitions.

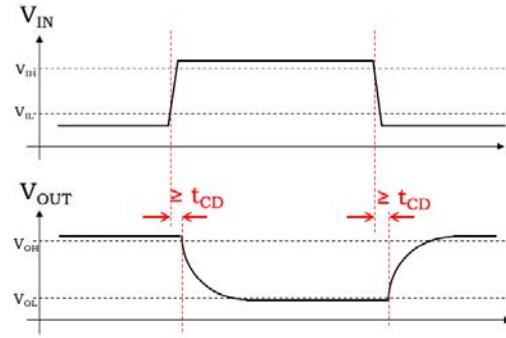
We can go through the same process to measure the delay associated with a falling input transition. First, identify the time at which V_{IN} cross the V_{IL} threshold. Then find the time at which V_{OUT} crosses the V_{OH} threshold. The resulting interval is the delay we wanted to measure.

Since the propagation delay, t_{PD} , is an upper bound on the delay associated with *any* input transition, we'll choose a value for t_{PD} that's greater than or equal to the measurements we just made. When a manufacturer selects the t_{PD} specification for a gate, it must take into account manufacturing variations, the effects of different environmental conditions such as temperature and power-supply voltage, and so on. It should choose a t_{PD} that will be an upper bound on any delay measurements their customers might make on actual devices.

From the designer's point of view, we can rely on this upper bound for each component of a larger digital system and use it to calculate the system's t_{PD} without having to repeat all the manufacturer's measurements. If our goal is to minimize the propagation delay of our system, then we'll want to keep the capacitances and resistances as small as possible. There's an interesting tension here: to make the effective resistance of a MOSFET switch smaller, we would increase its width. But that would add additional capacitance to the switch's gate terminal, slowing down transitions on the input node that connects to the gate! It's a fun optimization problem to figure out transistor sizing that minimizes the overall propagation delay.

Contamination Delay

Contamination delay (t_{CD}): A LOWER BOUND on the delay from any **invalid input** to an **invalid output**



Do we really need t_{CD} ?

Usually not... it'll be important when we design circuits with registers (coming soon!)

If t_{CD} is not specified, safe to assume it's 0.

Although not strictly required by the static discipline, it will be useful to define another timing specification, called the *contamination delay*. It measures how long a gate's previous output value remains valid after the gate's inputs start to change and become invalid. Technically, the contamination delay will be a lower bound on the delay from an invalid input to an invalid output. We'll make the delay measurements much as we did for the propagation delay.

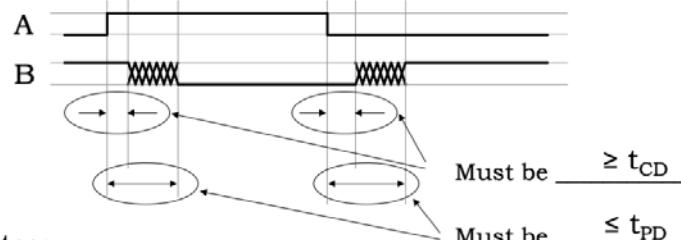
On a rising input transition, the delay starts when the input is no longer a valid digital 0, *i.e.*, when V_{IN} crosses the V_{IL} threshold. And the delay ends when the output becomes invalid, *i.e.*, when V_{OUT} crosses the V_{OL} threshold. We can make a similar delay measurement for the falling input transition. Since the contamination delay, t_{CD} , is a lower bound on the delay associated with *any* input transition, we'll choose a value for t_{CD} that's less than or equal to the measurements we just made.

Do we really need the contamination delay specification? Usually not. And if not's specified, designers should assume that the t_{CD} for a combinational device is 0. In other words a conservative assumption is that the outputs go invalid as soon as the inputs go invalid.

By the way, manufacturers often use the term "minimum propagation delay" to refer to a device's contamination delay. That terminology is a bit confusing, but now you know what it is they're trying to tell you.

The Combinational Contract

$A \rightarrowtail B$	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td style="padding: 2px;">A</td><td style="padding: 2px;">B</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td></tr> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td></tr> </table>	A	B	0	1	1	0	t_{PD} propagation delay t_{CD} contamination delay
A	B							
0	1							
1	0							



Notes:

1. No Promises during ~~xxxx~~
2. Default (conservative) spec: $t_{CD} = 0$

So here's a quick summary of the timing specifications for combinational logic. These specifications tell us how the timing of changes in the output waveform (labeled B in this example) are related to the timing of changes in the input waveform (labeled A).

A combinational device may retain its previous output value for some interval of time after an input transition. The contamination delay of the device is a guarantee on the minimum size of that interval, *i.e.*, t_{CD} is a lower bound on how long the old output value stays valid. As stated in Note 2, a conservative assumption is that the contamination delay of a device is 0, meaning the device's output may change immediately after an input transition. So t_{CD} gives us information on when B will start to change.

Similarly, it would be good to know when B is guaranteed to be done changing after an input transition. In other words, how long do we have to wait for a change in the inputs to reflect in an updated value on the outputs? This is what t_{PD} tells us since it is an upper bound on the time it takes for B to become valid and stable after an input transition.

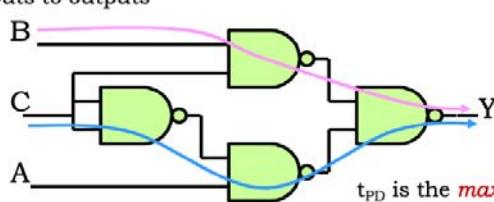
As Note 1 points out, in general there are no guarantees on the behavior of the output in the interval after t_{CD} and before t_{PD} , as measured from the input transition. It would be legal for the B output to change several times in that interval, or even have a non-digital voltage for any part of the interval. As we'll see in the last part of this lecture, we'll be able to offer more insights into B's behavior in this interval for a subclass of combinational devices. But in general, a designer should make no assumptions about B's value in the interval between t_{CD} and t_{PD} .

Acyclic Combinational Circuits

If NAND gates have a $t_{PD} = 4\text{ nS}$ and $t_{CD} = 1\text{ nS}$

$$t_{CD} \text{ is the } \textcolor{red}{minimum} \text{ cumulative contamination delay over all paths from inputs to outputs} \quad t_{PD} = \underline{12} \text{ nS}$$

$$t_{CD} = \underline{2} \text{ nS}$$



t_{PD} is the *maximum* cumulative propagation delay over all paths from inputs to outputs

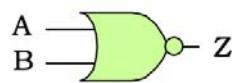
How do we calculate the propagation and contamination delays of a larger combinational circuit from the timing specifications of its components?

Our example is a circuit of four NAND gates where each NAND has a t_{PD} of 4 ns and t_{CD} of 1 ns. To find the propagation delay for the larger circuit, we need to find the maximum delay from an input transition on nodes A, B, or C to a valid and stable value on the output Y. To do this, consider each possible path from one of the inputs to Y and compute the path delay by summing the t_{PD} s of the components along the path. Choose the largest such path delay as the t_{PD} of the overall circuit. In our example, the largest delay is a path that includes three NAND gates, with a cumulative propagation delay of 12 ns. In other words, the output Y is guaranteed to be stable and valid within 12 ns of a transition on A, B, or C.

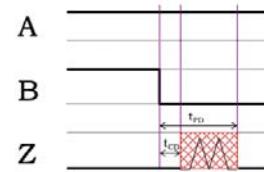
To find the contamination delay for the larger circuit, we again investigate all paths from inputs to outputs, but this time we're looking for the shortest path from an invalid input to an invalid output. So we sum the t_{CD} s of the components along each path and choose the smallest such path delay as the t_{CD} of the overall circuit. In our example, the smallest delay is a path that includes two NAND gates with a cumulative contamination delay of 2 ns. In other words, the output Y will retain its previous value for at least 2 ns after one of the inputs goes invalid.

One Last Timing Issue...

NOR:



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0



Recall the rules for *combinational devices*:

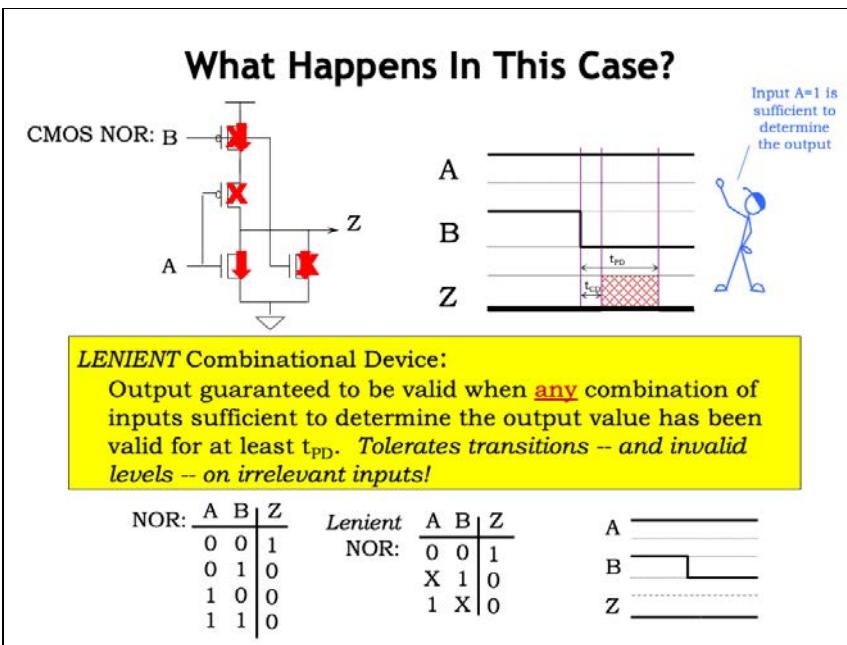
Output guaranteed to be valid when *all* inputs have been valid for at least t_{PD} , and, outputs may become invalid no earlier than t_{CD} after an input changes!

Many gate implementations—e.g., CMOS—adhere to even tighter restrictions.

It turns out we can say a bit more about the timing of output transitions for CMOS logic gates. Let's start by considering the behavior of a non-CMOS combinational device that implements the NOR function. Looking at the waveform diagram, we see that initially the A and B inputs are both 0, and the output Z is 1, just as specified by the truth table. Now B makes a 0-to-1 transition and the Z output will eventually reflect that change by making a 1-to-0 transition. As we learned in the previous video, the timing of the Z transition is determined by the contamination and propagation delays of the NOR gate. Note that we can't say anything about the value of the Z output in the interval of t_{CD} to t_{PD} after the input transition, which we indicate with a red shaded region on the waveform diagram.

Now, let's consider a different set up, where initially both A and B are 1, and, appropriately, the output Z is 0. Examining the truth table we see that if A is 1, the output Z will be 0 regardless of the value of B. So what happens when B makes a 1-to-0 transition? Before the transition, Z was 0 and we expect it to be 0 again, t_{PD} after the B transition. But, in general, we can't assume anything about the value of Z in the interval between t_{CD} and t_{PD} . Z could have any behavior it wants in that interval and the device would still be a legitimate combinational device.

Many gate technologies — e.g., CMOS — adhere to even tighter restrictions.



Let's look in detail at the switch configuration in a CMOS implementation of a NOR gate when both inputs are a digital 1. A high gate voltage will turn on NFET switches (as indicated by the red arrows) and turn off PFET switches (as indicated by the red X's). Since the pullup circuit is not conducting and the pulldown circuit is conducting, the output Z is connected to GND, the voltage for a digital 0 output.

Now, what happens when the B input transitions from 1 to 0? The switches controlled by B change their configuration: the PFET switch is now on and the NFET switch is now off. But overall the pullup circuit is still not conducting and there is still a pulldown path from Z to GND. So while there used to be two paths from Z to GND and there is now only one path, Z has been connected to GND the whole time and its value has remained valid and stable throughout B's transition. In the case of a CMOS NOR gate, when one input is a digital 1, the output will be unaffected by transitions on the other input.

A *lenient combinational device* is one that exhibits this behavior, namely that the output is guaranteed to be valid when any combination of inputs sufficient to determine the output value has been valid for at least t_{PD} . When some of the inputs are in a configuration that triggers this lenient behavior, transitions on the other inputs will have no effect on the validity of the output value. Happily most CMOS implementations of logic gates are naturally lenient.

We can extend our truth-table notation to indicate lenient behavior by using X for the input values on certain rows to indicate that input value is irrelevant when determining the correct output value. The truth table for a lenient NOR gate calls out two such situations: when A is 1, the value of B is irrelevant, and when B is 1, the value of A is irrelevant. Transitions on the irrelevant inputs don't trigger the t_{CD} and t_{PD} output timing normally associated with an input transition.

When does lenience matter? We'll need lenient components when building memory components, a topic we'll get to in a couple of lectures.

You're ready to try building some CMOS gates of your own!

Summary

- CMOS
 - Only use NFETs in pulldowns, PFETs in pullups → mosfets behave as voltage-controlled switches
 - Series/parallel Pullup and pulldown switch circuits are complementary
 - CMOS gates are naturally inverting (rising input transition can only cause falling output transition, and vice versa).
 - “Perfect” VTC (high gain, $V_{OH} = V_{DD}$, $V_{OL} = GND$) means large noise margins and no static power dissipation.
- Timing specs
 - t_{PD} : upper bound on time from valid inputs to valid outputs
 - t_{CD} : lower bound on time from invalid inputs to invalid outputs
 - If not specified, assume $t_{CD} = 0$
 - Lenient gates: output unaffected by some input transitions
- Next time: logic simplification, other canonical forms

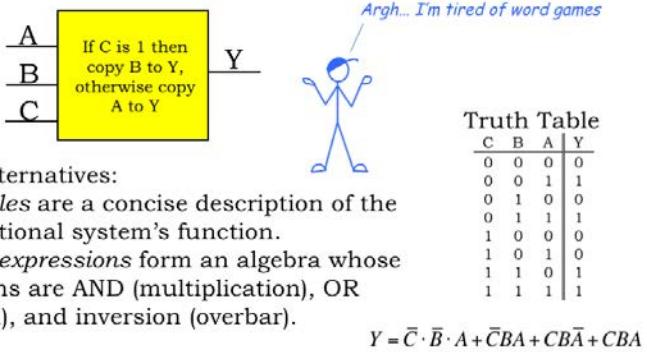
4: Combinational Logic

1. Functional Specifications
2. Here's a Design Approach
3. Sum-of-products Building Blocks
4. Straightforward Synthesis
5. ANDs and ORs with > 2 Inputs
6. More Building Blocks
7. Universal Building Blocks
8. CMOS Loves Inverting Logic
9. Wide NANDs and NORs
10. CMOS Sum-of-products Implementation
11. Logic Simplification
12. Boolean Minimization
13. Truth Tables with Don't Cares
14. The Case for a Non-minimal SOP
15. Karnuagh Maps: A Geometric Approach
16. Extending K-maps to 4-variable Tables
17. Finding Implicants
18. Finding Prime Implicants
19. Write Down Equations
20. Prime Implicants, Glitches & Leniency
21. We've Been Designing a MUX
22. Systematic Implementation Strategies
23. Synthesis By Table Lookup
24. Read-only Memory (ROM)
25. ROM Example
26. ROM Example continued
27. Faster ROMs
28. Logic According to ROMs
29. Summary

In this lecture, you'll learn various techniques for creating combinational logic circuits that implement a particular functional specification.

Functional Specifications

There are many ways of specifying the function of a combinational device, for example:



Any combinational (Boolean) function can be specified as a truth table or an equivalent sum-of-products Boolean expression!

A functional specification is part of the static discipline we use to build the combinational logic abstraction of a circuit. One approach is to use natural language to describe the operation of a device. This approach has its pros and cons. In its favor, natural language can convey complicated concepts in surprisingly compact form and it is a notation that most of us know how to read and understand. But, unless the words are very carefully crafted, there may be ambiguities introduced by words with multiple interpretations or by lack of completeness since it's not always obvious whether all eventualities have been dealt with.

There are good alternatives that address the shortcomings mentioned above. Truth tables are a straightforward tabular representation that specifies the values of the outputs for each possible combination of the digital inputs. If a device has N digital inputs, its truth table will have 2^N rows. In the example shown here, the device has 3 inputs, each of which can have the value 0 or the value 1. There are $2 \cdot 2 \cdot 2 = 2^3 = 8$ combinations of the three input values, so there are 8 rows in the truth table. It's straightforward to systematically enumerate the 8 combinations, which makes it easy to ensure that no combination is omitted when building the specification. And since the output values are specified explicitly, there isn't much room for misinterpreting the desired functionality!

Truth tables are an excellent choice for devices with small numbers of inputs and outputs. Sadly, they aren't really practical when the devices have many inputs. If, for example, we were describing the functionality of a circuit to add two 32-bit numbers, there would be 64 inputs altogether and the truth table would need 2^{64} rows. Hmm, not sure how practical that is! If we entered the correct output value for a row once per second, it would take 584 billion years to fill in the table!

Another alternative specification is to use Boolean equations to describe how to compute the output values from the input values using Boolean algebra. The operations we use are the logical operations AND, OR, and XOR, each of which takes two Boolean operands, and NOT which takes a single Boolean operand. Using the truth tables that describe these logical operations, it's straightforward to compute an output value from a particular combination of input values using the sequence of operations laid out in the equation.

Let me say a quick word about the notation used for Boolean equations. Input values are represented by the name of the input, in this example one of A, B, or C. The digital input value 0 is equivalent to the Boolean value FALSE and the digital input value 1 is equivalent to the Boolean value TRUE.

The Boolean operation NOT is indicated by a horizontal line drawn above a Boolean expression. In this example, the first symbol following the equal sign is a C with line above it, indicating that the value of C should be inverted before it's used in evaluating the rest of the expression.

The Boolean operation AND is represented by the multiplication operation using standard mathematical notation. Sometimes we'll use an explicit multiplication operator — usually written as a dot between two Boolean expressions — as shown in the first term of the example equation. Sometimes the AND operator is implicit as shown in the remaining three terms of the example equation.

The Boolean operation OR is represented by the addition operation, always shown as a “+” sign.

Boolean equations are useful when the device has many inputs. And, as we'll see, it's easy to convert a Boolean equation into a circuit schematic.

Truth tables and Boolean equations are interchangeable. If we have a Boolean equation for each output, we can fill in the output columns for a row of the truth table by evaluating the Boolean equations using the particular combination of input values for that row. For example, to determine the value for Y in the first row of the truth table, we'd substitute the Boolean value FALSE for the symbols A, B, and C in the equation and then use Boolean algebra to compute the result.

We can go the other way too. We can always convert a truth table into a particular form of Boolean equation called a sum-of-products. Let's see how...

Here's a Design Approach

Truth Table

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

1. Write out our functional spec as a truth table

2. Write down a Boolean expression with terms covering each '1' in the output:

$$Y = \bar{C}\bar{B}A + \bar{C}BA + C\bar{B}A + CBA$$

3. We'll show how to build a circuit using this equation in the next two slides.

This approach will always give us Boolean expressions in a particular form: SUM-OF-PRODUCTS

Start by looking at the truth table and answering the question “When does Y have the value 1?” Or in the language of Boolean algebra: “When is Y TRUE?” Well, Y is TRUE when the inputs correspond to row 2 of the truth table, OR to row 4, OR to rows 7 OR 8. Altogether there are 4 combinations of inputs for which Y is TRUE. The corresponding Boolean equation thus is the OR for four terms, where each term is a Boolean expression which evaluates to TRUE for a particular combination of inputs.

Row 2 of the truth table corresponds to C=0, B=0, and A=1. The corresponding Boolean expression is $\bar{C} \cdot \bar{B} \cdot A$, an expression that evaluates to TRUE if and only if C is 0, B is 0, and A is 1.

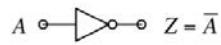
The Boolean expression corresponding to row 4 is $\bar{C} \cdot B \cdot A$. And so on for rows 7 and 8.

This approach will always give us an expression in the form of a sum-of-products. “Sum” refers to the OR operations and “products” refers to the groups of AND operations. In this example, we have the sum of four product terms.

Our next step is to use the Boolean expression as a recipe for constructing a circuit implementation using combinational logic gates.

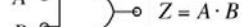
Sum-of-products Building Blocks

INVERTER:



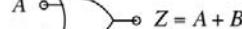
A	Z
0	1
1	0

AND:



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR:



A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

As circuit designers, we'll be working with a library of combinational logic gates, which either is given to us by the integrated circuit manufacturer, or which we've designed ourselves as CMOS gates using NFET and PFET switches.

One of the simplest gates is the inverter, which has the schematic symbol shown here. The small circle on the output wire indicates an inversion, a common convention used in schematics. We can see from its truth table that the inverter implements the Boolean NOT function.

The AND gate outputs 1 if and only if the A input is 1 *and* the B input is 1, hence the name AND. The library will usually include AND gates with 3 inputs, 4 inputs, etc., which produce a 1 output if and only if all of their inputs are 1.

The OR gate outputs 1 if the A input is 1 **or** if the B input is 1, hence the name OR. Again, the library will usually include OR gates with 3 inputs, 4 inputs, etc., which produce a 1 output when at least one of their inputs is 1.

These are the standard schematic symbols for AND and OR gates. Note that the AND symbol is straight on the input side, while the OR symbol is curved. With a little practice, you'll find it easy to remember which schematic symbols are which.

Now let's use these building blocks to build a circuit that implements a sum-of-products Boolean equation.

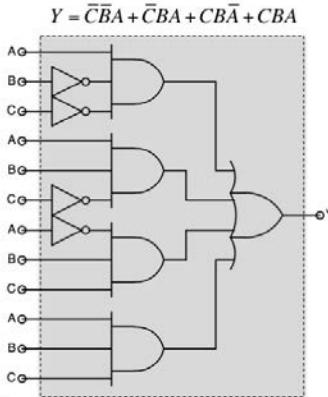
Straightforward Synthesis

We can implement
SUM-OF-PRODUCTS
with just three levels of
logic:

1. Inverters
2. ANDs
3. OR



Propagation delay --
No more than 3 gate delays?*



*assuming gates with an arbitrary number of inputs,
which, as we'll see, isn't a good assumption!

The structure of the circuit exactly follows the structure of the Boolean equation. We use inverters to perform the necessary Boolean NOT operations. In a sum-of-products equation the inverters are operating on particular input values, in this case A, B and C. To keep the schematic easy to read we've used a separate inverter for each of the four NOT operations in the Boolean equation, but in real life we might invert the C input once to produce a NOT-C signal, then use that signal whenever a NOT-C value is needed.

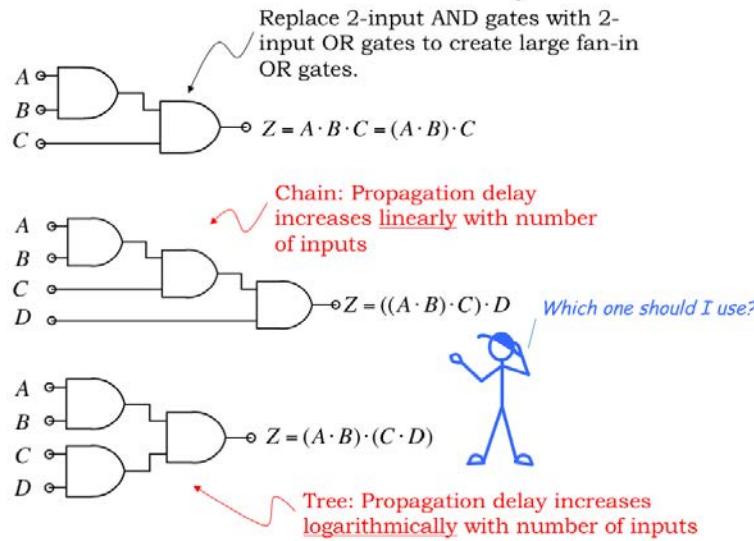
Each of the four product terms is built using a 3-input AND gate. And the product terms are ORed together using a 4-input OR gate. The final circuit has a layer of inverters, a layer of AND gates and final OR gate. In the next section, we'll talk about how to build AND or OR gates with many inputs from library components with fewer inputs.

The propagation delay for a sum-of-products circuit looks pretty short: the longest path from inputs to outputs includes an inverter, an AND gate and an OR gate. Can we really implement any Boolean equation in a circuit with a t_{PD} of three gate delays?

Actually not, since building ANDs and ORs with many inputs will require additional layers of components, which will increase the propagation delay. We'll learn about this in the next section.

The good news is that we now have straightforward techniques for converting a truth table to its corresponding sum-of-products Boolean equation, and for building a circuit that implements that equation.

ANDs and ORs with > 2 Inputs



On our to-do list from the previous section is figuring out how to build AND and OR gates with many inputs. These will be needed when creating circuit implementations using a sum-of-products equation as our template. Let's assume our gate library only has 2-input gates and figure how to build wider gates using the 2-input gates as building blocks. We'll work on creating 3- and 4-input gates, but the approach we use can be generalized to create AND and OR gates of any desired width.

The approach shown here relies on the associative property of the AND operator. This means we can perform an N-way AND by doing pair-wise ANDs in any convenient order. The OR and XOR operations are also associative, so the same approach will work for designing wide OR and XOR circuits from the corresponding 2-input gate. Simply substitute 2-input OR gates or 2-input XOR gates for the 2-input AND gates shown below and you're good to go!

Let's start by designing a circuit that computes the AND of three inputs A, B, and C. In the circuit shown here, we first compute (A AND B), then AND that result with C.

Using the same strategy, we can build a 4-input AND gate from three 2-input AND gates. Essentially we're building a chain of AND gates, which implement an N-way AND using N-1 2-input AND gates.

We can also associate the four inputs a different way: computing (A AND B) in parallel with (C AND D), then combining those two results using a third AND gate. Using this approach, we're building a tree of AND gates.

Which approach is best: chains or trees? First we have to decide what we mean by "best." When designing circuits, we're interested in cost, which depends on the number of components, and performance, which we characterize by the propagation delay of the circuit.

Both strategies require the same number of components since the total number of pair-wise ANDs is the same in both cases. So it's a tie when considering costs. Now consider propagation delay.

The chain circuit in the middle has a t_{PD} of 3 gate delays, and we can see that the t_{PD} for an N-input chain will be N-1 gate delays. The propagation delay of chains grows linearly with the number of

inputs.

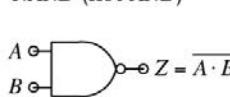
The tree circuit on the bottom has a t_{PD} of 2 gates, smaller than the chain. The propagation delay of trees grows logarithmically with the number of inputs. Specifically, the propagation delay of tree circuits built using 2-input gates grows as $\log_2(N)$. When N is large, tree circuits can have dramatically better propagation delay than chain circuits.

The propagation delay is an upper bound on the worst-case delay from inputs to outputs and is a good measure of performance assuming that all inputs arrive at the same time. But in large circuits, A, B, C and D might arrive at different times depending on the t_{PD} of the circuit generating each one. Suppose input D arrives considerably after the other inputs. If we used the tree circuit to compute the AND of all four inputs, the additional delay in computing Z is two gate delays after the arrival of D. However, if we use the chain circuit, the additional delay in computing Z might be as little as one gate delay.

The moral of this story: it's hard to know which implementation of a subcircuit, like the 4-input AND shown here, will yield the smallest overall t_{PD} unless we know the t_{PD} of the circuits that compute the values for the input signals.

More Building Blocks

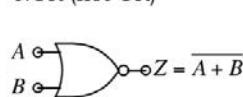
NAND (not AND)



A B Z

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR (not OR)

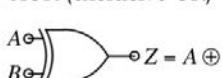


A B Z

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

In a CMOS gate, rising inputs lead to falling outputs and vice-versa, so CMOS gates are naturally inverting. Want to use NANDs and NORs in CMOS designs... But **NAND and NOR operations are not associative**, so wide NAND and NOR gate can't use a chain or tree strategy. Stay tuned for more on this!

XOR (exclusive OR)



A B Z

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

XOR is very useful when implementing parity and arithmetic logic. Also used as a "programmable inverter": if A=0, Z=B; if A=1, Z=~B

Wide fan-in XORs can be created with chains or trees of 2-input XORs.

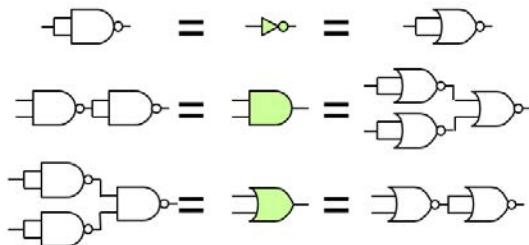
In designing CMOS circuits, the individual gates are naturally inverting, so instead of using AND and OR gates, for the best performance we want to the use the NAND and NOR gates shown here. NAND and NOR gates can be implemented as a single CMOS gate involving one pullup circuit and one pulldown circuit. AND and OR gates require two CMOS gates in their implementation, e.g., a NAND gate followed by an INVERTER. We'll talk about how to build sum-of-products circuitry using NANDs and NORs in the next section.

Note that NAND and NOR operations are not associative: $\text{NAND}(A,B,C)$ is not equal to $\text{NAND}(\text{NAND}(A,B),C)$. So we can't build a NAND gate with many inputs by building a tree of 2-input NANDs. We'll talk about this in the next section too!

We've mentioned the exclusive-or operation, sometimes called XOR, several times. This logic function is very useful when building circuitry for arithmetic or parity calculations. As you'll see in Lab 2, implementing a 2-input XOR gate will take many more NFETs and PFETs than required for a 2-input NAND or NOR.

Universal Building Blocks

NANDs and NORs are universal:



Any logic function can be implemented using only NANDs (or, equivalently, NORs). Good news for CMOS technologies!

We know we can come up with a sum-of-products expression for any truth table and hence build a circuit implementation using INVERTERS, AND gates, and OR gates. It turns out we can build circuits with the same functionality using only 2-INPUT NAND gates — we say the 2-INPUT NAND is a universal gate.

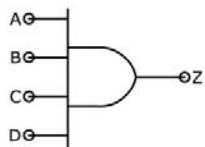
Here we show how to implement the sum-of-products building blocks using just 2-input NAND gates. In a minute we'll show a more direct implementation for sum-of-products using only NANDs, but these little schematics are a proof-of-concept showing that NAND-only equivalent circuits exist.

2-INPUT NOR gates are also universal, as shown by these little schematics.

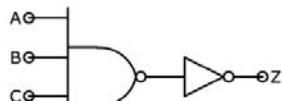
Inverting logic takes a little getting used to, but it's the key to designing low-cost high-performance circuits in CMOS.

CMOS ❤️ Inverting Logic

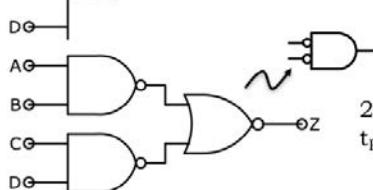
See "The Standard Cell Library" handout in *Updates & Handouts*



AND4:
 $t_{PD} = 160 \text{ ps}$, size = $20\mu^2$



NAND4 + INV:
 $t_{PD} = 90 \text{ ps}$, size = $27\mu^2$



Demorgan's Laws:
 $\overline{A \cdot B} = \overline{A} + \overline{B}$
 $\overline{A + B} = \overline{A} \cdot \overline{B}$

2*NAND2 + NOR2:
 $t_{PD} = 80 \text{ ps}$, size = $30\mu^2$

Now would be a good time to take a moment to look at the documentation for the library of logic gates we'll use for our designs — look for *The Standard Cell Library* handout. The information on this slide is taken from there.

The library has both inverting gates (such as inverters, NANDs and NORs) and non-inverting gates (such as buffers, ANDs and ORs). Why bother to include both types of gates? Didn't we just learn we can build any circuit using only NAND or NOR?

Good questions! We get some insight into the answers if we look at these three implementations for a 4-input AND function.

The upper circuit is a direct implementation using the 4-input AND gate available in the library. The t_{PD} of the gate is 160 picoseconds and its size is 20 square microns. Don't worry too much about the actual numbers, what matters on this slide is how the numbers compare between designs.

The middle circuit implements the same function, this time using a 4-INPUT NAND gate hooked to an inverter to produce the AND functionality we want. The t_{PD} of this circuit is 90 picoseconds, considerably faster than the single gate above. The tradeoff is that the size is somewhat larger.

How can this be? Especially since we know the AND gate implementation is the NAND/INVERTER pair shown in the middle circuit. The answer is that the creators of the library decided to make the non-inverting gates small but slow by using MOSFETs with much smaller widths than used in the inverting logic gates, which were designed to be fast.

Why would we ever want to use a slow gate? Remember that the propagation delay of a circuit is set by the longest path in terms of delay from inputs to outputs. In a complex circuit, there are many input/output paths, but it's only the components on the longest path that need to be fast in order to achieve the best possible overall t_{PD} . The components on the other, shorter paths, can potentially be a bit slower. And the components on short input/output paths can be very slow indeed. So for the portions of the circuit that aren't speed sensitive, it's a good tradeoff to use slower but smaller gates. The overall performance isn't affected, but the total size is improved.

So for faster performance we'll design with inverting gates, and for smallest size we'll design with non-inverting gates. The creators of the gate library designed the available gates with this tradeoff in mind.

The 4-input inverting gates are also designed with this tradeoff in mind. For the ultimate in performance, we want to use a tree circuit of 2-input gates, as shown in the lower circuit. This implementation shaves 10 picoseconds off the t_{PD} , while costing us a bit more in size.

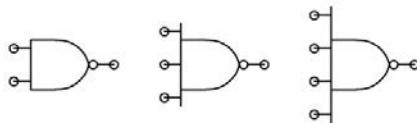
Take a closer look at the lower circuit. This tree circuit uses two NAND gates whose outputs are combined with a NOR gate. Does this really compute the AND of A, B, C, and D? Yup, as you can verify by building the truth table for this combinational system using the truth tables for NAND and NOR.

This circuit is a good example of the application of a particular Boolean identity known as Demorgan's Law. There are two forms of Demorgan's law, both of which are shown here. The top form is the one we're interested in for analyzing the lower circuit. It tells us that the NOR of A with B is equivalent to the AND of (NOT A) with (NOT B). So the 2-input NOR gate can be thought of as a 2-input AND gate with inverting inputs. How does this help? We can now see that the lower circuit is actually a tree of AND gates, where the inverting outputs of the first layer match up with the inverting inputs of the second layer.

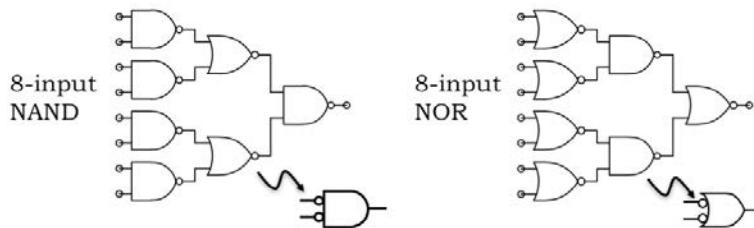
It's a little confusing the first time you see it, but with practice you'll be comfortable using Demorgan's law when building trees or chains of inverting logic.

Wide NANDs and NORs

Most logic libraries include 2-, 3- and 4-input devices:



But for a large number of inputs, the series connections of too many MOSFETs can lead to very large effective R.
Design note: use trees of smaller devices...



Using Demorgan's Law we can answer the question of how to build NANDs and NORs with large numbers of inputs. Our gate library includes inverting gates with up to 4 inputs. Why stop there? Well, the pulldown chain of a 4-input NAND gate has 4 NFETs in series and the resistance of the conducting channels is starting to add up. We could make the NFETs wider to compensate, but then the gate gets much larger and the wider NFETs impose a higher capacitive load on the input signals. The number of possible tradeoffs between size and speed grows rapidly with the number of inputs, so it's usually just best for the library designer to stop at 4-input gates and let the circuit designer take it from there.

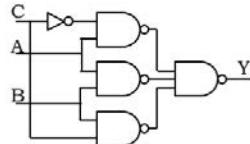
Happily, Demorgan's law shows us how build trees of alternating NANDs and NORs to build inverting logic with a large number of inputs. Here we see schematics for an 8-input NAND and an 8-input NOR gate.

Think of the middle layer of NOR gates in the left circuit as AND gates with inverting inputs and then it's easy to see that the circuit is a tree of ANDs with an inverting output.

Similarly, think of the middle layer of NAND gates in the right circuit as OR gates with inverting inputs and see that we really have a tree of OR gates with an inverting output.

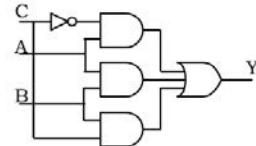
CMOS Sum-of-products Implementation

NAND-NAND

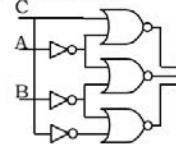


$$\overline{AB} = \overline{A} + \overline{B}$$

"Pushing Bubbles"



NOR-NOR



$$\overline{AB} = \overline{A} + \overline{B}$$

$$A\bar{C} + AB + BC$$

You might think all these extra inverters would make this structure less attractive. However, quite the opposite is true.



$$A\bar{C} + AB + BC$$

Now let's see how to build sum-of-products circuits using inverting logic. The two circuits shown here implement the same sum-of-products logic function. The one on the top uses two layers of NAND gates, the one on the bottom, two layers of NOR gates.

Let's visualize Demorgan's Law in action on the top circuit. The NAND gate with Y on its output can be transformed by Demorgan's Law into an OR gate with inverting inputs. So we can redraw the circuit on the top left as the circuit shown on the top right. Now, notice that the inverting outputs of the first layer are cancelled by the inverting inputs of the second layer [CLICK], a step we can show visually by removing matching inversions. And, voila, we see the NAND/NAND circuit in sum-of-products form: a layer of inverters, a layer of AND gates, and an OR gate to combine the product terms.

We can use a similar visualization to transform the output gate of the bottom circuit, giving us the circuit on the bottom right. Match up the bubbles and we see that we have the same logic function as above.

Looking at the NOR/NOR circuit on the bottom left, we see it has 4 inverters, whereas the NAND/NAND circuit only has one. Why would we ever use the NOR/NOR implementation? It has to do with the loading on the inputs. In the top circuit, the input A connects to a total of four MOSFET switches. In the bottom circuit, it connects to only the two MOSFET switches in the inverter. So, the bottom circuit imposes half the capacitive load on the A signal. This might be significant if the signal A connected to many such circuits.

The bottom line: when you find yourself needing a fast implementation for the AND/OR circuitry for a sum-of-products expression, try using the NAND/NAND implementation. It'll be noticeably faster than using AND/OR.

Logic Simplification

Can we implement the same function with fewer gates? Before trying we'll add a few more tricks in our bag.

BOOLEAN ALGEBRA:

OR rules: $a + 1 = 1, a + 0 = a, a + a = a$

AND rules: $a1 = a, a0 = 0, aa = a$

Commutative: $a + b = b + a, ab = ba$

Associative: $(a + b) + c = a + (b + c), (ab)c = a(bc)$

Distributive: $a(b+c) = ab + ac, a + bc = (a+b)(a+c)$

Complements: $a + \bar{a} = 1, a\bar{a} = 0$

Absorption: $a + ab = a, a + \bar{a}b = a + b, a(a+b) = a, a(\bar{a} + b) = ab$

Reduction: $ab + \bar{a}b = b, (a+b)(\bar{a}+b) = b$

DeMorgan's Law: $\bar{a} + \bar{b} = \bar{ab}, \bar{ab} = \bar{a} + \bar{b}$

The previous sections showed us how to build a circuit that computes a given sum-of-products expression. An interesting question to ask is if we can implement the same functionality using fewer gates or smaller gates? In other words is there an equivalent Boolean expression that involves fewer operations? Boolean algebra has many identities that can be used to transform an expression into an equivalent, and hopefully smaller, expression.

The reduction identity in particular offers a transformation that simplifies an expression involving two variables and four operations into a single variable and no operations. Let's see how we might use that identity to simplify a sum-of-products expression.

Boolean Minimization

Let's (again!) simplify

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

Using the identity

$$\alpha A + \alpha \overline{A} = \alpha(A + \overline{A}) = \alpha \cdot 1 = \alpha$$

For any expression α and variable A:

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

$$Y = \overline{C}BA + CB + \overline{C}BA$$

$$Y = \overline{C}A + CB$$

*Can't he come up
with a new example???*



*Hey... I could write
a program to do
that*



Here's the equation from the start of this chapter, involving 4 product terms. We'll use a variant of the reduction identity involving a Boolean expression alpha and a single variable A. Looking at the product terms, the middle two offer an opportunity to apply the reduction identity if we let alpha be the expression (C AND B). So we simplify the middle two product terms to just alpha, i.e., (C AND B), eliminating the variable A from this part of the expression.

Considering the now three product terms, we see that the first and last terms can also be reduced, this time letting alpha be the expression (NOT C and A). Wow, this equivalent equation is much smaller! Counting inversions and pair-wise operations, the original equation has 14 operations, while the simplified equation has 4 operations. The simplified circuit would be much cheaper to build and have a smaller t_{PD} in the bargain!

Doing this sort of Boolean simplification by hand is tedious and error-prone. Just the sort of task a computer program could help with. Such programs are in common use, but the computation needed to discover the smallest possible form for an expression grows faster than exponentially as the number of inputs increases. So for larger equations, the programs use various heuristics to choose which simplifications to apply. The results are quite good, but not necessarily optimal. But it sure beats doing the simplification by hand!

Truth Tables with “Don’t Cares”

One way to reveal the opportunities for a more compact implementation is to rewrite the truth table using “don’t cares” (- or X) to indicate when the value of a particular input is irrelevant in determining the value of the output.

C	B	A	Y		C	B	A	Y	
0	0	0	0		0	X	0	0	
0	0	1	1		0	X	1	1	→ $\bar{C}A$
0	1	0	0		1	0	X	0	
0	1	1	1		1	1	X	1	→ CB
1	0	0	0		X	0	0	0	
1	0	1	0		X	1	1	1	→ BA
1	1	0	1						
1	1	1	1						



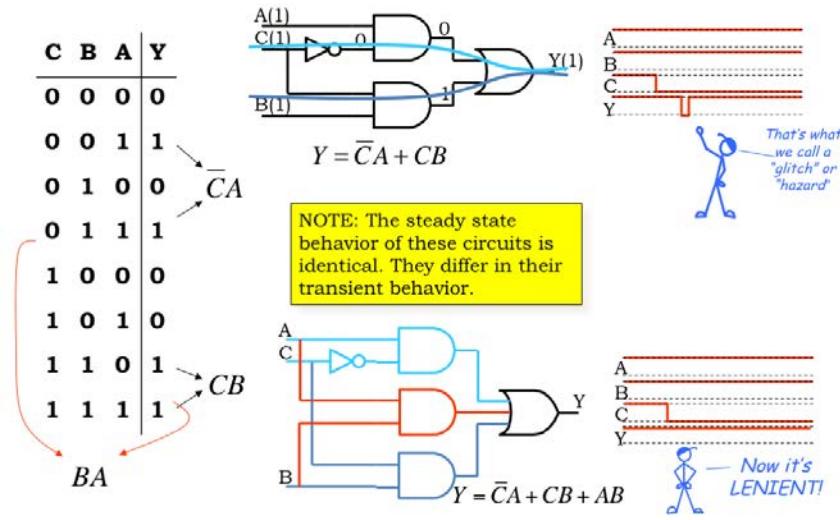
Note: Some input combinations (e.g., 000) are matched by more than one row in the “don’t care” table. It would be a bug if all the matching rows didn’t specify the same output value!

Another way to think about simplification is by searching the truth table for don’t-care situations. For example, look at the first and third rows of the original truth table on the left. In both cases A is 0, C is 0, and the output Y is 0. The only difference is the value of B, which we can then tell is irrelevant when both A and C are 0. This gives us the first row of the truth table on the right, where we use X to indicate that the value of B doesn’t matter when A and C are both 0. By comparing rows with the same value for Y, we can find other don’t-care situations.

The truth table with don’t-cares has only three rows where the output is 1. And, in fact, the last row is redundant in the sense that the input combinations it matches (011 and 111) are covered by the second and fourth rows.

The product terms derived from rows two and four are exactly the product terms we found by applying the reduction identity.

The Case for a Non-minimal SOP



Do we always want to use the simplest possible equation as the template for our circuits? Seems like that would minimize the circuit cost and maximize performance, a good thing.

The simplified circuit is shown here. Let's look at how it performs when A is 1, B is 1, and C makes a transition from 1 to 0. Before the transition, C is 1 and we can see from the annotated node values that it's the bottom AND gate that's causing the Y output to be 1.

When C transitions to 0, the bottom AND gate turns off and the top AND gate turns on, and, eventually the Y output becomes 1 again. But the turning on of the top AND is delayed by the t_{PD} of the inverter, so there's a brief period of time where neither AND gate is on, and the output momentarily becomes 0. This short blip in Y's value is called a glitch and it may result in short-lived changes on many node values as it propagates through other parts of the circuit. All those changes consume power, so it would be good to avoid these sorts of glitches if we can.

If we include the third product term BA in our implementation, the circuit still computes the same long-term answer as before. But now when A and B are both high, the output Y will be 1 independently of the value of the C input. So the 1-to-0 transition on the C input doesn't cause a glitch on the Y output. If you recall the last section of the previous chapter, the phrase we used to describe such circuits is *lenient*.

Karnaugh Maps: A Geometric Approach

K-Map: a truth table arranged so that terms which differ by exactly one variable are adjacent to one another so we can see potential reductions easily.

Truth Table

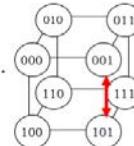
C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Here's the layout of a 3-variable K-map filled in with the values from our truth table:

C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0

Why did he shade that row Gray?

It's cyclic. The left edge is adjacent to the right edge.
(It's really just a flattened out cube).



When trying to minimize a sum-of-products expression using the reduction identity, our goal is to find two product terms that can be written as one smaller product term, eliminating the don't-care variable. This is easy to do when two the product terms come from adjacent rows in the truth table. For example, look at the bottom two rows in this truth table. Since the Y output is 1 in both cases, both rows will be represented in the sum-of-products expression for this function. It's easy to spot the don't care variable: when C and B are both 1, the value of A isn't needed to determine the value of Y. Thus, the last two rows of the truth table can be represented by the single product term (B AND C).

Finding these opportunities would be easier if we reorganized the truth table so that the appropriate product terms were on adjacent rows. That's what we've done in the Karnaugh map, K-map for short, shown on the right. The K-map organizes the truth table as a two-dimensional table with its rows and columns labeled with the possible values for the inputs. In this K-map, the first row contains entries for when C is 0 and the second row contains entries for when C is 1. Similarly, the first column contains entries for when A is 0 and B is 0. And so on. The entries in the K-map are exactly the same as the entries in the truth table, they're just formatted differently.

Note that the columns have been listed in a special sequence that's different from the usual binary counting sequence. In this sequence, called a Gray Code, adjacent labels differ in exactly one of their bits. In other words, for any two adjacent columns, either the value of the A label changed, or the value of the B label changed.

In this sense, the leftmost and rightmost columns are also adjacent. We write the table as a two-dimensional matrix, but you should think of it as cylinder with its left and right edges touching. If it helps you visualize which entries are adjacent, the edges of the cube shows which 3-bit input values differ by only one bit. As shown by the red arrows, if two entries are adjacent in the cube, they are also adjacent in the table.

Extending K-maps to 4-variable Tables

4-variable K-map $F(A,B,C,D)$:

\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

Again it's cyclic. The left edge is adjacent to the right edge, and the top is adjacent to the bottom.

For functions of 5 or 6 variables, we'd need to use the 3rd dimension to build a 4x4x4 K-map. But then we're out of dimensions...

It's easy to extend the K-map notation to truth tables for functions with 4 inputs, as shown here. We've used a Gray code sequencing for the rows as well as the columns. As before, the leftmost and rightmost columns are adjacent, as are the top and bottom rows. Again, as we move to an adjacent column or an adjacent row, only one of the four input labels will have changed.

To build a K-map for functions of 6 variables we'd need a 4x4x4 matrix of values. That's hard to draw on the 2D page and it would be a challenge to tell which cells in the 3D matrix were adjacent. For more than 6 variables we'd need additional dimensions. Something we can handle with computers, but hard those of us who live in only a three-dimensional space!

As a practical matter, K-maps work well for up to 4 variables, and we'll stick with that. But keep in mind that you can generalize the K-map technique to higher dimensions.

Finding Implicants

An implicant

- is a rectangular region of the K-map where the function has the value 1 (i.e., a region that will need to be described by one or more product terms in the sum-of-products)
- has a width and length that must be a power of 2: 1, 2, 4
- can overlap other implicants
- is a prime implicant if it is not completely contained in any other implicant.

C\AB	00	01	11	10
0	0	0	1	1
1	1	0	0	0

$\bar{A}C$

$\bar{A}BC$

C\AB	00	01	11	10
0	1	0	0	1
1	1	1	0	1

$\bar{A}C$

\bar{B}

- can be uniquely identified by a single product term. The larger the implicant, the smaller the product term.

So why talk about K-maps? Because patterns of adjacent K-map entries that contain 1's will reveal opportunities for using simpler product terms in our sum-of-products expression.

Let's introduce the notion of an implicant, a fancy name for a rectangular region of the K-map where the entries are all 1's. Remember when an entry is a 1, we'll want the sum-of-products expression to evaluate to TRUE for that particular combination of input values.

We require the width and length of the implicant to be a power of 2, *i.e.*, the region should have 1, 2, or 4 rows, and 1, 2, or 4 columns.

It's okay for implicants to overlap. We say that an implicant is a *prime implicant* if it is not completely contained in any other implicant. Each product term in our final minimized sum-of-products expression will be related to some prime implicant in the K-map.

Let's see how these rules work in practice using these two example K-maps. As we identify prime implicants, we'll circle them in red. Starting with the K-map on the left, the first implicant contains the singleton 1-cell that's not adjacent to any other cell containing 1's.

The second prime implicant is the pair of adjacent 1's in the upper right hand corner of the K-map. This implicant is has one row and two columns, meeting our constraints on an implicant's dimensions.

Finding the prime implicants in the right-hand K-map is a bit trickier. Recalling that the left and right columns are adjacent, we can spot a 2x2 prime implicant. Note that this prime implicant contains many smaller 1x2, 2x1 and 1x1 implicants, but none of those would be prime implicants since they are completely contained in the 2x2 implicant.

It's tempting draw a 1x1 implicant around the remaining 1, but actually we want to find the largest implicant that contains this particular cell. In this case, that's the 1x2 prime implicant shown here. Why do we want to find the largest possible prime implicants? We'll answer that question in a minute...

Each implicant can be uniquely identified by a product term, a Boolean expression that evaluates to TRUE for every cell contained within the implicant and FALSE for all other cells. Just as we did for the truth table rows at the beginning of this chapter, we can use the row and column labels to help us build the correct product term.

The first implicant we circled corresponds to the product term $\overline{A} \cdot \overline{B} \cdot C$, an expression that evaluates to TRUE when A is 0, B is 0, and C is 1.

How about the 1x2 implicant in the upper-right hand corner? We don't want to include the input variables that change as we move around in the implicant. In this case the two input values that remain constant are C (which has the value 0) and A (which has the value 1), so the corresponding product term is $A \cdot \overline{C}$.

Here are the two product terms for the two prime implicants in the right-hand K-map. Notice that the larger the prime implicant, the smaller the product term! That makes sense: as we move around inside a large implicant, the number of inputs that remain constant across the entire implicant is smaller. Now we see why we want to find the largest possible prime implicants: they give us the smallest product terms!

Finding Prime Implicants

We want to find all the prime implicants. The right strategy is a greedy one.

- Find the uncircled prime implicant with the greatest area
 - Order: $4 \times 4 \Rightarrow 2 \times 4$ or $4 \times 2 \Rightarrow 4 \times 1$ or 1×4 or $2 \times 2 \Rightarrow 2 \times 1$ or $1 \times 2 \Rightarrow 1 \times 1$
 - Overlap is okay
- Circle it
- Repeat until all prime implicants are circled

\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

Let's try another example. Remember that we're looking for the largest possible prime implicants. A good way to proceed is to find some un-circled 1, and then identify the largest implicant we can find that incorporates that cell.

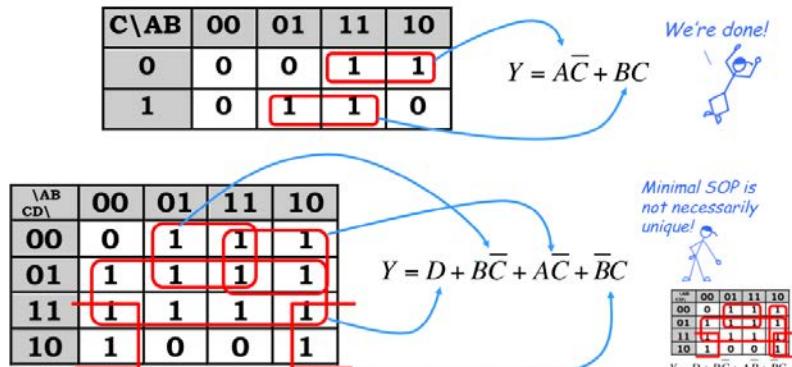
There's a 2×4 implicant that covers the middle two rows of the table. Looking at the 1's in the top row, we can identify two 2×2 implicants that include those cells.

There's a 4×1 implicant that covers the right column, leaving the lonely 1 in the lower left-hand corner of the table. Looking for adjacent 1's and remembering the table is cyclic, we can find a 2×2 implicant that incorporates this last un-circled 1.

Notice that we're always looking for the largest possible implicant, subject to constraint that each dimension has to be either 1, 2 or 4. It's these largest implicants that will turn out to be prime implicants.

Write Down Equations

Picking just enough prime implicants to cover all the 1's in the KMap, combine equations to form minimal sum-of-products.



Now that we've identified the prime implicants, we're ready to build the minimal sum-of-products expression.

Here are two example K-maps where we've shown only the prime implicants needed to cover all the 1's in the map. This means, for example, that in the 4-variable map, we didn't include the 4x1 implicant covering the right column. That implicant was a prime implicant since it wasn't completely contained by any other implicant, but it wasn't needed to provide a cover for all the ones in the table.

Looking at the top table, we'll assemble the minimal sum-of-products expression by including the product terms for each of the shown implicants. The top implicant has the product term A AND (not C), and the bottom implicant has the product term (B AND C). And we're done! Why is the resulting equation minimal? If there was some further reduction that could be applied, to produce a yet smaller product term, that would mean there was a larger prime implicant that could have been circled in the K-map.

Looking at the bottom table, we can assemble the sum-of-products expression term-by-term. There were 4 prime implicants, so there are 4 product terms in the expression.

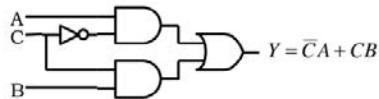
And we're done. Finding prime implicants in a K-map is faster and less error-prone than fooling around with Boolean algebra identities.

Note that the minimal sum-of-products expression isn't necessarily unique. If we had used a different mix of the prime implicants when building our cover, we would have come up with different sum-of-products expression. Of course, the two expressions are equivalent in the sense that they produce the same value of Y for any particular combination of input values — they were built from the same truth table after all. And the two expressions will have the same number of operations.

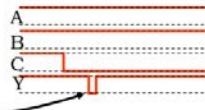
So when you need to come up with a minimal sum-of-products expression for functions of up to 4 variables, K-maps are the way to go!

Prime Implicants, Glitches & Leniency

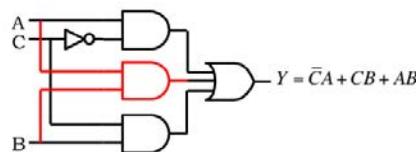
This circuit produces a glitch on Y when A=1, B=1, C: 1→0



C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0



To make the circuit lenient, include product terms for ALL prime implicants.



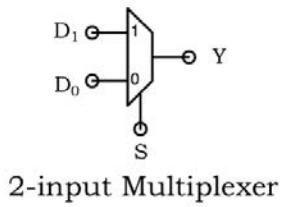
We can also use K-maps to help us remove glitches from output signals. Earlier in the chapter we saw this circuit and observed that when A was 1 and B was 1, then a 1-to-0 transition on C might produce a glitch on the Y output as the bottom product term turned off and the top product term turned on.

That particular situation is shown by the yellow arrow on the K-map, where we're transitioning from the cell on the bottom row of the 1-1 column to the cell on the top row. It's easy to see that we're leaving one implicant and moving to another. It's the gap between the two implicants that leads to the potential glitch on Y.

It turns out there's a prime implicant that covers the cells involved in this transition shown here with a dotted red outline. We didn't include it when building the original sum-of-products implementation since the other two product terms provided the necessary functionality. But if we do include that implicant as a third product term in the sum-of-products, no glitch can occur on the Y output.

To make an implementation lenient, simply include all the prime implicants in the sum-of-products expression. That will bridge the gaps between product terms that lead to potential output glitches.

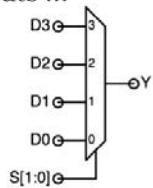
We've Been Designing a Mux



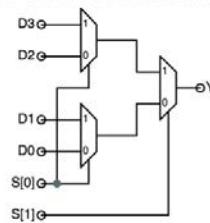
Truth Table

S	D ₁	D ₀	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

MUXes can be generalized to 2^k data inputs and k select inputs ...



... and implemented as a tree of smaller MUXes:



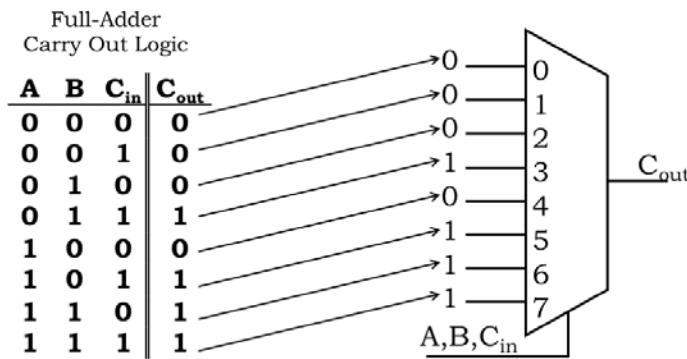
The truth table we've been using as an example describes a very useful combinational device called a 2-to-1 multiplexer. A multiplexer, or MUX for short, selects one of its two input values as the output value. When the select input, marked with an S in the diagram, is 0, the value on data input D0 becomes the value of the Y output. When S is 1, the value of data input D1 is selected as the Y output value.

MUXes come in many sizes, depending on the number of select inputs. A MUX with K select inputs will choose between the values of 2^K data inputs. For example, here's a 4-to-1 multiplexer with 4 data inputs and 2 select inputs.

Larger MUXes can be built from a tree of 2-to-1 MUXes, as shown here.

Systematic Implementation Strategies

Consider implementing some arbitrary Boolean function, $F(A,B,C) \dots$ using a MULTIPLEXER as the only circuit element:

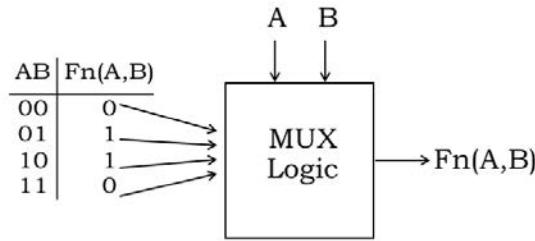


Why are MUXes interesting? One answer is that they provide a very elegant and general way of implementing a logic function. Consider the 8-to-1 MUX shown on the right. The 3 inputs — A, B, and CIN — are used as the three select signals for the MUX. Think of the three inputs as forming a 3-bit binary number. For example, when they're all 0, the MUX will select data input 0, and when they're all 1, the MUX will select data input 7, and so on.

How does make it easy to implement the logic function shown in the truth table? Well, we'll wire up the data inputs of the MUX to the constant values shown in the output column in the truth table. The values on the A, B and CIN inputs will cause the MUX to select the appropriate constant on the data inputs as the value for the COUT output.

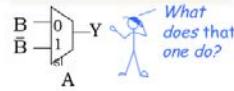
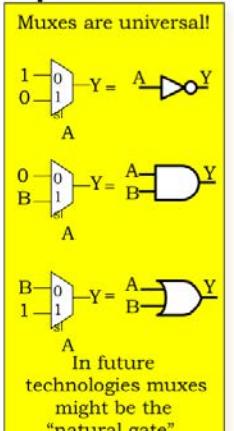
If later on we change the truth table, we don't have to redesign some complicated sum-of-products circuit, we simply have to change the constants on the data inputs. Think of the MUX as a table-lookup device that can be reprogrammed to implement, in this case, any three-input equation. This sort of circuit can be used to create various forms of programmable logic, where the functionality of the integrated circuit isn't determined at the time of manufacture, but is set during a programming step performed by the user at some later time. Modern programmable logic circuits can be programmed to replace millions of logic gates. Very handy for prototyping digital systems before committing to the expense of a custom integrated circuit implementation.

Synthesis By Table Lookup



Generalizing:
 In theory, we can build any 1-output combinational logic block with multiplexers.
 For an N -input function we need a 2^N input mux.

Is this practical for BIG truth tables?
 How about 10-input function? 20-input?

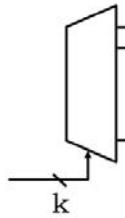


So MUXes with N select lines are effectively stand-ins for N -input logic circuits. Such a MUX would have 2^N data inputs. They're useful for N up to 5 or 6, but for functions with more inputs, the exponential growth in circuit size makes them impractical.

Not surprisingly, MUXes are universal as shown by these MUX-based implementations for the sum-of-products building blocks. There is some speculation that in molecular-scale logic technologies, MUXes may be the natural gate, so it's good to know they can be used to implement any logic function.

Even XOR is simple to implement with a single 2-to-1 MUX!

A New Combinational Device



DECODER:

- k SELECT inputs,
 - $N = 2^k$ DATA OUTPUTS.
- Select inputs choose one of the D_i to assert HIGH, all others will be LOW.

Have I mentioned that HIGH is a synonym for '1' and LOW means the same as '0'

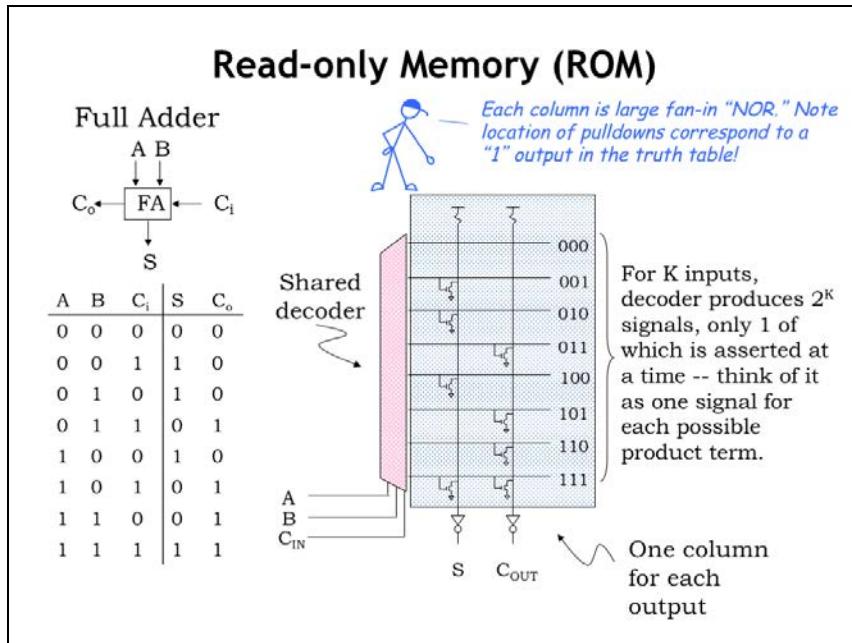


NOW, we are well on our way to building a general purpose table-lookup device.

We can build a 2-dimensional ARRAY of decoders and selectors as follows ...

Here's a final logic implementation strategy using read-only memories. This strategy is useful when you need to generate many different outputs from the same set of inputs, a situation we'll see a lot when we get to finite state machines later on in the course. Where MUXes are good for implementing truth tables with one output column, read-only memories are good for implementing truth tables with many output columns.

One of the key components in a read-only memory is the decoder which has K select inputs and 2^K data outputs. Only one of the data outputs will be 1 (or HIGH) at any given time, which one is determined by the value on the select inputs. The Jth output will be 1 when the select lines are set to the binary representation of J.



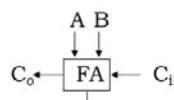
Here's a read-only memory implementation for the 2-output truth table shown on the left. This particular 2-output device is a full adder, which is used as a building block in addition circuits.

The three inputs to the function (A, B, and CI) are connected to the select lines of a 3-to-8 decoder. The 8 outputs of the decoder run horizontally in the schematic diagram and each is labeled with the input values for which that output will be HIGH. So when the inputs are 000, the top decoder output will be HIGH and all the other decoder outputs LOW. When the inputs are 001 — *i.e.*, when A and B are 0 and CI is 1 — the second decoder output will be HIGH. And so on.

The decoder outputs control a matrix of NFET pulldown switches. The matrix has one vertical column for each output of the truth table. Each switch connects a particular vertical column to ground, forcing it to a LOW value when the switch is on. The column circuitry is designed so that if no pulldown switches force its value to 0, its value will be a 1. The value on each of the vertical columns is inverted to produce the final output values.

Read-only Memory (ROM)

Full Adder



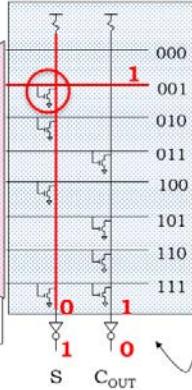
A	B	C _i	S	C _o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Each column is large fan-in "NOR." Note location of pulldowns correspond to a "1" output in the truth table!

Shared decoder

A
B
C_{IN}



For K inputs, decoder produces 2^K signals, only 1 of which is asserted at a time -- think of it as one signal for each possible product term.

One column for each output

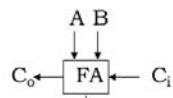
So how do we use all this circuitry to implement the function described by the truth table? For any particular combination of input values, exactly one of the decoder outputs will be HIGH, all the others will be low. Think of the decoder outputs as indicating which row of the truth table has been selected by the input values. All of the pulldown switches controlled by the HIGH decoder output will be turned ON, forcing the vertical column to which they connect LOW.

For example, if the inputs are 001, the decoder output labeled 001 will be HIGH. This will turn on the circled pulldown switch, forcing the S vertical column LOW. The COUT vertical column is not pulled down, so it will be HIGH. After the output inverters, S will be 1 and COUT will be 0, the desired output values.

By changing the locations of the pulldown switches, this read-only memory can be programmed to implement any 3-input, 2-output function.

Read-only Memory (ROM)

Full Adder

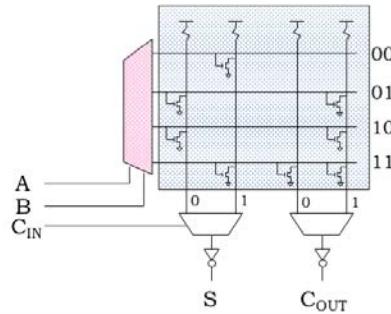


A	B	C _i	S	C _o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



LONG LINES slow down propagation times...

The best way to improve this is to build square arrays, using some inputs to drive output selectors (MUXes):



2D Addressing: Standard for ROMs, RAMs, logic arrays...

For read-only memories with many inputs, the decoders have many outputs and the vertical columns in the switch matrix can become quite long and slow. We can reconfigure the circuit slightly so that some of the inputs control the decoder and the other inputs are used to select among multiple shorter and faster vertical columns. This combination of smaller decoders and output MUXes is quite common in these sorts of memory circuits.

Logic According to ROMs

ROMs *ignore* the structure of combinational functions ...

- Size, layout, and design are independent of function
- Any Truth table can be “programmed” by minor reconfiguration:

- Metal layer (masked ROMs)
- Fuses (Field-programmable PROMs)
- Charge on floating gates (EPROMs)
- ... etc.

ROMs tend to generate “glitchy” outputs. WHY?

Model: LOOK UP value of function in truth table...

Inputs: “ADDRESS” of a T.T. entry

ROM SIZE = # TT entries...

... for an N-input boolean function, size $\equiv 2^N \times \#outputs$

Read-only memories, ROMs for short, are an implementation strategy that ignores the structure of the particular Boolean expression to be implemented. The ROM’s size and overall layout are determined only by the number of inputs and outputs. Typically the switch matrix is fully populated, with all possible switch locations filled with an NFET pulldown. A separate physical or electrical programming operation determines which switches are actually controlled by the decoder lines. The other switches are configured to be in the permanently off state.

If the ROM has N inputs and M outputs, then the switch matrix will have 2^N rows and M output columns, corresponding exactly to the size of the truth table.

As the inputs to the ROM change, various decoder outputs will turn off and on, but at slightly different times. As the decoder lines cycle, the output values may change several times until the final configuration of the pulldown switches is stable. So ROMs are not lenient and the outputs may show the glitchy behavior discussed earlier.

Summary

- Sum of products
 - Any function that can be specified by a truth table or, equivalently, in terms of AND/OR/NOT (Boolean expression)
 - “3-level” implementation of any logic function
 - Limitations on number of inputs (fan-in) increases depth
 - SOP implementation methods
 - NAND-NAND, NOR-NOR
- Muxes used to build table-lookup implementations
 - Easy to change implemented function -- just change constants
- ROMs
 - Decoder logic generates all possible product terms
 - Selector logic determines which terms are ORed together

Whew! This has been a whirlwind tour of various circuits we can use to implement logic functions. The sum-of-products approach lends itself nicely to implementation with inverting logic. Each circuit is custom-designed to implement a particular function and as such can be made both fast and small. The design and manufacturing expense of creating such circuits is worthwhile when you need high-end performance or are producing millions of devices.

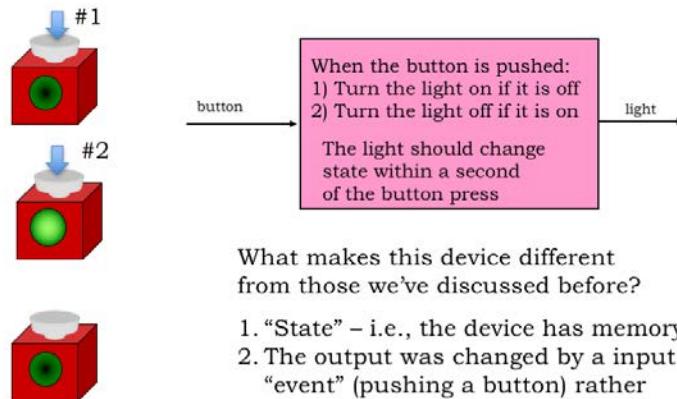
MUX and ROM circuit implementations are mostly independent of the specific function to be implemented. That’s determined by a separate programming step, which may be completed after the manufacture of the devices. They are particularly suited for prototyping, low-volume production, or devices where the functionality may need to be updated after the device is out in the field.

L5: Sequential Logic

1. Something We Can't Build (Yet)
2. Digital State: What We'd Like to Build
3. Memory: Using Capacitors
4. Memory: Using Feedback
5. Settable Memory Element
6. New Device: D Latch
7. A Plea for Lenience
8. ... With a Little Discipline
9. Let's Try it Out!
10. Flaky Control Systems
11. Solution: Escapement Strategy (2 Gates)
12. Edge-triggered D Register
13. D-Register Waveforms
14. Um, About That Hold Time...
15. D-Register Timing
16. Single-clock Synchronous Circuits
17. Timing in a Single-clock System
18. Model: Discrete Time
19. Sequential Circuit Timing
20. Summary

Something We Can't Build (Yet)

What if you were given the following design specification:



In the last lecture we learned how to build combinational logic circuits given a functional specification that told us how output values were related to the current values of the inputs.

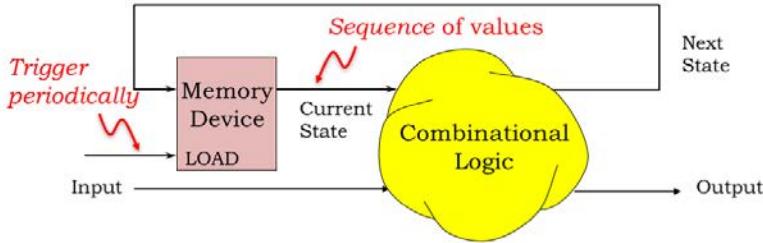
But here's a simple device we can't build with combinational logic. The device has a light that serves as the output and push button that serves as the input. If the light is off and we push the button, the light turns on. If the light is on and we push the button, the light turns off.

What makes this circuit different from the combinational circuits we've discussed so far? The biggest difference is that the device's output is not function of the device's **current** input value. The behavior when the button is pushed depends on what has happened in the past: odd numbered pushes turn the light on, even numbered pushes turn the light off. The device is "remembering" whether the last push was an odd push or an even push so it will behave according to the specification when the next button push comes along. Devices that remember something about the history of their inputs are said to have state.

The second difference is more subtle. The push of the button marks an event in time: we speak of the state before the push ("the light is on") and state after the push ("the light is off"). It's the transition of the button from un-pushed to pushed that we're interested in, not the whether the button is currently pushed or not.

The device's internal state is what allows it to produce different outputs even though it receives the same input. A combinational device can't exhibit this behavior since its outputs depends only on the current values of the input. Let's see how we'll incorporate the notion of device state into our circuitry.

Digital State: What We'd Like to Build



Plan: Build a Sequential Circuit with stored digital STATE –

- Memory stores CURRENT state, produced at output
- Combinational Logic computes
 - NEXT state (from input, current state)
 - OUTPUT bits (from input, current state)
- State changes on LOAD control input



We'll introduce a new abstraction of a memory component that will store the current state of the digital system we want to build. The memory component stores one or more bits that encode the current state of the system. These bits are available as digital values on the memory component's outputs, shown here as the wire marked "Current State".

The current state, along with the current input values, are the inputs to a block of combinational logic that produces two sets of outputs. One set of outputs is the next state of the device, encoded using the same number of bits as the current state. The other set of outputs are the signals that serve as the outputs of the digital system. The functional specification for the combinational logic (perhaps a truth table, or maybe a set of Boolean equations) specifies how the next state and system outputs are related to the current state and current inputs.

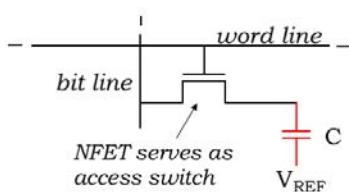
The memory component has two inputs: a LOAD control signal that indicates when to replace the current state with the next state, and a data input that specifies what the next state should be. Our plan is to periodically trigger the LOAD control, which will produce a sequence of values for the current state. Each state in the sequence is determined from the previous state and the inputs at the time the LOAD was triggered.

Circuits that include both combinational logic and memory components are called sequential logic. The memory component has a specific capacity measured in bits. If the memory component stores K bits, that puts an upper bound of 2^K on the number of possible states since the state of the device is encoded using the K bits of memory.

So, we'll need to figure out how to build a memory component that can loaded with new values now and then. That's the subject of this chapter. We'll also need a systematic way of designing sequential logic to achieve the desired sequence of actions. That's the subject of the next chapter.

Memory: Using Capacitors

We've chosen to encode information using voltages and we know from physics that we can "store" a voltage as charge on a capacitor:



To write:

Drive bit line, turn on access fet,
force storage cap to new voltage

To read:

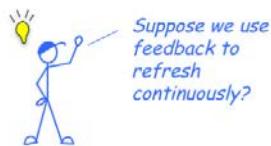
precharge bit line, turn on access fet,
detect (small) change in bit line voltage

Pros:

- compact – low cost/bit
(on BIG memories)

Cons:

- complex interface
- stable? (noise, ...)
- it leaks! \Rightarrow refresh



We've been representing bits as voltages, so we might consider using a capacitor to store a particular voltage. The capacitor is passive two-terminal device. The terminals are connected to parallel conducting plates separated by insulator. Adding charge Q to one plate of the capacitor generates a voltage difference V between the two plate terminals. Q and V are related by the capacitance C of the capacitor: $Q = CV$.

When we add charge to a capacitor by hooking a plate terminal to higher voltage, that's called "charging the capacitor". And when we take away charge by connecting the plate terminal to a lower voltage, that's called "discharging the capacitor".

So here's how a capacitor-based memory device might work. One terminal of the capacitor is hooked to some stable reference voltage. We'll use an NFET switch to connect the other plate of the capacitor to a wire called the bit line. The gate of the NFET switch is connected to a wire called the word line.

To write a bit of information into our memory device, drive the bit line to the desired voltage (*i.e.*, a digital 0 or a digital 1). Then set the word line HIGH, turning on the NFET switch. The capacitor will then charge or discharge until it has the same voltage as the bit line. At this point, set the word line LOW, turning off the NFET switch and isolating the capacitor's charge on the internal plate. In a perfect world, the charge would remain on the capacitor's plate indefinitely.

At some later time, to access the stored information, we first charge the bit line to some intermediate voltage. Then set the word line HIGH, turning on the NFET switch, which connects the charge on the bit line to the charge on the capacitor. The charge sharing between the bit line and capacitor will have some small effect on the charge on the bit line and hence its voltage. If the capacitor was storing a digital 1 and hence was at a higher voltage, charge will flow from the capacitor into the bit line, raising the voltage of the bit line. If the capacitor was storing a digital 0 and was at lower voltage, charge will flow from the bit line into the capacitor, lowering the voltage of the bit line. The change in the bit line's voltage depends on the ratio of the bit line capacitance to C , the storage capacitor's capacitance, but is usually quite small. A very sensitive amplifier, called a sense amp, is used to detect that small change and produce a legal digital voltage as the value read from the memory cell.

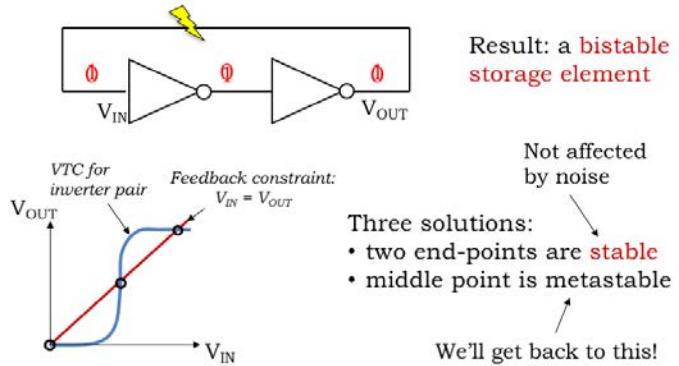
Whew! Reading and writing require a whole sequence of operations, along with carefully designed analog electronics. The good news is that the individual storage capacitors are quite small — in modern integrated circuits we can fit billions of bits of storage on relatively inexpensive chips called dynamic random-access memories, or DRAMs for short. DRAMs have a very low cost per bit of storage.

The bad news is that the complex sequence of operations required for reading and writing takes a while, so access times are relatively slow. And we have to worry about carefully maintaining the charge on the storage capacitor in the face of external electrical noise. The really bad news is that the NFET switch isn't perfect and there's a tiny amount leakage current across the switch even when it's officially off. Over time that leakage current can have a noticeable impact on the stored charge, so we have to periodically refresh the memory by reading and re-writing the stored value before the leakage has corrupted the stored information. In current technologies, this has to be done every 10ms or so.

Hmm. Maybe we can get around the drawbacks of capacitive storage by designing a circuit that uses feedback to provide a continual refresh of the stored information...

Memory: Using Feedback

IDEA: use **positive feedback** to maintain storage indefinitely.
Our logic gates are built to restore marginal signal levels, so noise shouldn't be a problem!



Here's a circuit using combinational inverters hooked in a positive feedback loop. If we set the input of one of the inverters to a digital 0, it will produce a digital 1 on its output. The second inverter will then produce a digital 0 on its output, which is connected back around to the original input. This is a stable system and these digital values will be maintained, even in the presence of noise, as long as this circuitry is connected to power and ground. And, of course, it's also stable if we flip the digital values on the two wires. The result is a system that has two stable configurations, called a bi-stable storage element.

Here's the voltage transfer characteristic showing how V_{OUT} and V_{IN} of the two-inverter system are related. The effect of connecting the system's output to its input is shown by the added constraint that V_{IN} equal V_{OUT} . We can then graphically solve for values of V_{IN} and V_{OUT} that satisfy both constraints. There are three possible solutions where the two curves intersect.

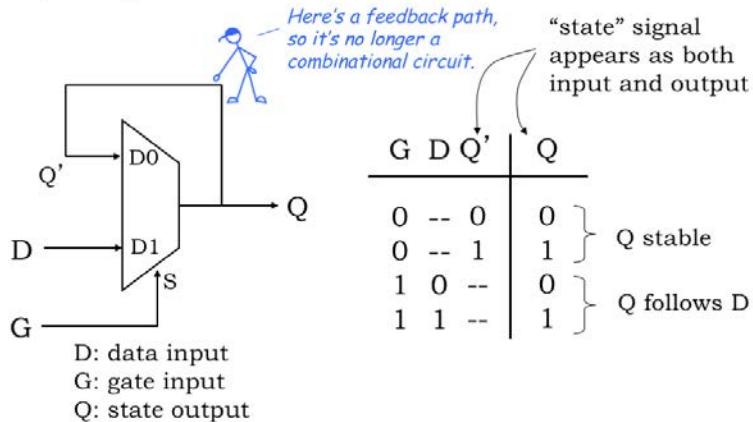
The two points of intersection at either end of the VTC are stable in the sense that small changes in V_{IN} (due, say, to electrical noise), have no effect on V_{OUT} . So the system will return to its stable state despite small perturbations.

The middle point of intersection is what we call metastable. In theory the system could “balance” at this particular V_{IN}/V_{OUT} voltage forever, but the smallest perturbation will cause the voltages to quickly transition to one of the stable solutions. Since we're planning to use this bi-stable storage element as our memory component, we'll need to figure out how to avoid getting the system into this metastable state. More on this in the next chapter.

Now let's figure out how to load new values into our bi-stable storage element.

Settable Memory Element

It's easy to build a settable storage element (called a **latch**) using a *lenient* MUX:



We can use a 2-to-1 multiplexer to build a settable storage element. Recall that a MUX selects as its output value the value of one of its two data inputs. The output of the MUX serves as the state output of the memory component. Internally to the memory component we'll also connect the output of the MUX to its D0 data input. The MUX{}'s D1 data input will become the data input of the memory component. And the select line of the MUX will become the memory component's load signal, here called the gate.

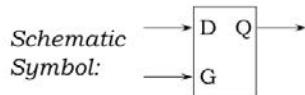
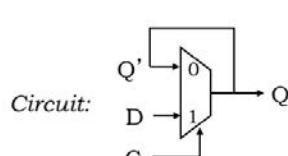
When the gate input is LOW, the MUX{}'s output is looped back through MUX through the D0 data input, forming the bi-stable positive feedback loop discussed in the last section. Note our circuit now has a cycle, so it no longer qualifies as a combinational circuit.

When the gate input is HIGH, the MUX{}'s output is determined by the value of the D1 input, *i.e.*, the data input of the memory component.

To load new data into the memory component, we set the gate input HIGH for long enough for the Q output to become valid and stable. Looking at the truth table, we see that when G is 1, the Q output follows the D input. While the G input is HIGH, any changes in the D input will be reflected as changes in the Q output, the timing being determined by the tPD of the MUX.

Then we can set the gate input LOW to switch the memory component into memory mode, where the stable Q value is maintained indefinitely by the positive feedback loop as shown in the first two rows of the truth table.

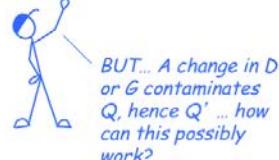
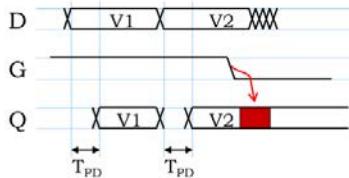
New Device: D Latch



$G=1$: Q Follows D, independently of Q'

$G=0$: Q Holds stable Q' , independently of D

$G=1$:
Q follows D
 $G=0$:
Q holds



Our memory device is called a D latch, or just a latch for short, with the schematic symbol shown here.

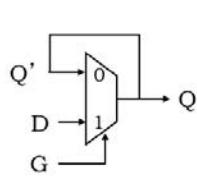
When the latch's gate is HIGH, the latch is open and information flows from the D input to the Q output. When the latch's gate is LOW, the latch is closed and in "memory mode", remembering whatever value was on the D input when the gate transitioned from HIGH to LOW.

This is shown in the timing diagrams on the right. The waveforms show when a signal is stable, i.e., a constant signal that's either LOW or HIGH, and when a signal is changing, shown as one or more transitions between LOW and HIGH.

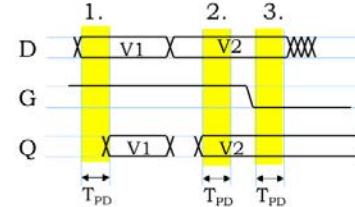
When G is HIGH, we can see Q changing to a new stable output value no later than tPD after D reaches a new stable value.

Our theory is that after G transitions to a LOW value, Q will stay stable at whatever value was on D when G made the HIGH to LOW transition. But, we know that in general, we can't assume anything about the output of a combinational device until tPD after the input transition — the device is allowed to do whatever it wants in the interval between tCD and tPD after the input transition. But how will our memory work if the 1-to-0 transition on G causes the Q output to become invalid for a brief interval? After all it's the value on the Q output we're trying to remember! We're going to have to ensure that a 1-to-0 transition on G doesn't affect the Q output.

A Plea for Lenience



G	D	Q'	Q
1	0	x	0
1	1	x	1
x	0	0	0
x	1	1	1
0	x	0	0
0	x	1	1



Assume LENIENT Mux,
propagation delay of T_{PD}

Then output valid when

1. $G=1$, D stable for T_{PD} ,
independently of Q' ; or
2. $Q=D$ stable for T_{PD} ,
independently of G ; or
3. $G=0$, Q stable for T_{PD} ,
independently of D

Does lenience *guarantee* a
working latch?



*What if D and G
change at about
the same time...*

That's why we specified a lenient MUX for our memory component. The truth table for a lenient MUX is shown here. The output of a lenient MUX remains valid and stable even after an input transition under any of the following three conditions.

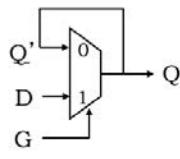
(1) When we're loading the latch by setting G HIGH, once the D input has been valid and stable for t_{PD} , we are guaranteed that the Q output will be stable and valid with the same value as the D input, independently of Q's initial value.

Or (2) If both Q and D are valid and stable for t_{PD} , the Q output will be unaffected by subsequent transitions on the G input. This is the situation that will allow us to have a 1-to-0 transition on G without contaminating the Q output.

Or, finally, (3) if G is LOW and Q has been stable for at least t_{PD} , the output will be unaffected by subsequent transitions on the D input.

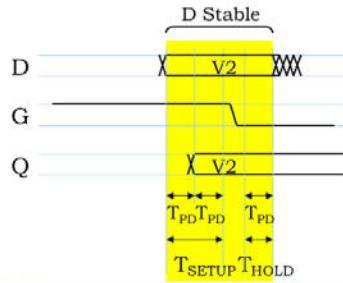
Does lenience guarantee a working latch? Well, only if we're careful about ensuring that signals are stable at the right times so we can leverage the lenient behavior of the MUX.

...With a Little Discipline



To reliably latch V2:

- Apply V2 to D, holding G=1
- After T_{PD} , V2 appears at $Q=Q'$
- After another T_{PD} , Q' & D both valid for T_{PD} ; *will hold $Q=V2$ independently of G*
- Set G=0, while Q' & D hold $Q=D$
- After another T_{PD} , G=0 and Q' are sufficient to hold $Q=V2$ *independently of D*



Dynamic Discipline for our latch:

$T_{SETUP} = 2T_{PD}$: interval *prior to G transition* for which D must be stable & valid

$T_{HOLD} = T_{PD}$: interval *following G transition* for which D must be stable & valid

Here are the steps we need to follow in order to ensure the latch will work as we want.

First, while the G input is HIGH, set the D input to the value we wish store in the latch. Then, after t_{PD} , we're guaranteed that value will be stable and valid on the Q output. This is condition (1) from the previous slide.

Now we wait another t_{PD} so that the information about the new value on the Q' input propagates through the internal circuitry of the latch. Now, both D **and** Q' have been stable for at least t_{PD} , giving us condition (2) from the previous slide.

So if D is stable for $2*t_{PD}$, transitions on G will not affect the Q output. This requirement on D is called the setup time of the latch: it's how long D must be stable and valid before the HIGH-to-LOW transition of G.

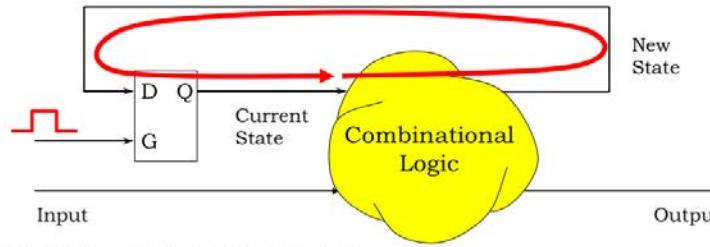
Now we can set G to LOW, still holding D stable and valid. After another t_{PD} to allow the new G value to propagate through the internal circuitry of the latch, we've satisfied condition (3) from the previous slide, and the Q output will be unaffected by subsequent transitions on D.

This further requirement on D's stability is called the hold time of the latch: it's how long after the transition on G that D must stay stable and valid.

Together the setup and hold time requirements are called the dynamic discipline, which must be followed if the latch is to operate correctly.

In summary, the dynamic discipline requires that the D input be stable and valid both before and after a transition on G. If our circuit is designed to obey the dynamic discipline, we can guarantee that this memory component will reliably store the information on D when the gate makes a HIGH-to-LOW transition.

Let's Try It Out!



When $G=1$, latch is *Transparent*...
 ... provides a combinational path from D to Q.
 Can't work without tricky timing constraints on $G=1$ pulse:

- Must fit within contamination delay of logic
- Must accommodate latch setup, hold times

 Want to signal an *INSTANT*, not an *INTERVAL*...

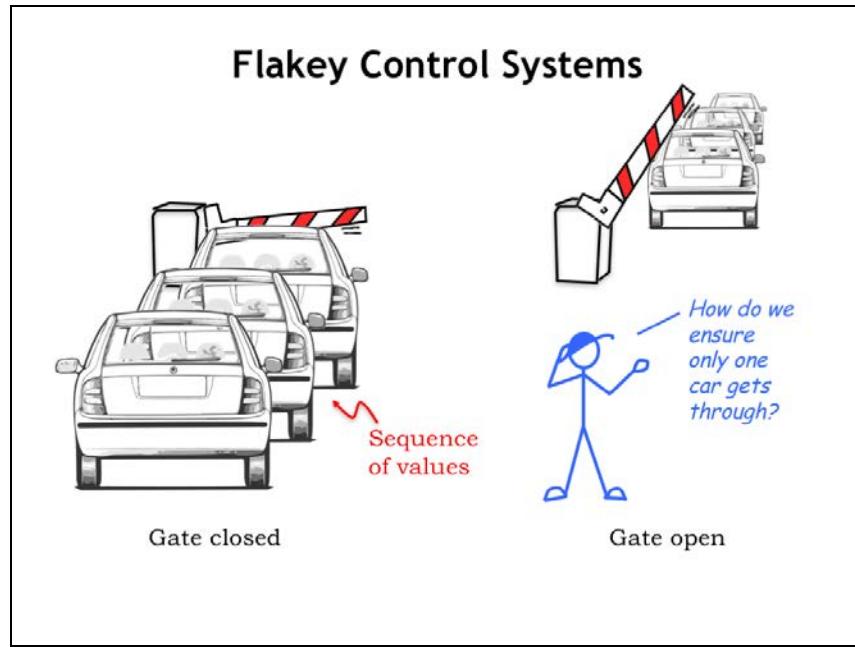


Let's try using the latch as the memory component in our sequential logic system.

To load the encoding of the new state into the latch, we open the latch by setting the latch's gate input HIGH, letting the new value propagate to the latch's Q output, which represents the current state. This updated value propagates through the combinational logic, updating the new state information. Oops, if the gate stays HIGH too long, we've created a loop in our system and our plan to load the latch with new state goes awry as the new state value starts to change rapidly as information propagates around and around the loop.

So to make this work, we need to carefully time the interval when G is HIGH. It has to be long enough to satisfy the constraints of the dynamic discipline, but it has to be short enough that the latch closes again before the new state information has a chance to propagate all the way around the loop.

Hmm. I think Mr. Blue is right: this sort of tricky system timing would likely be error-prone since the exact timing of signals is almost impossible to guarantee. We have upper and lower bounds on the timing of signal transitions but no guarantees of exact intervals. To make this work, we want to a load signal that marks an instant in time, not an interval.

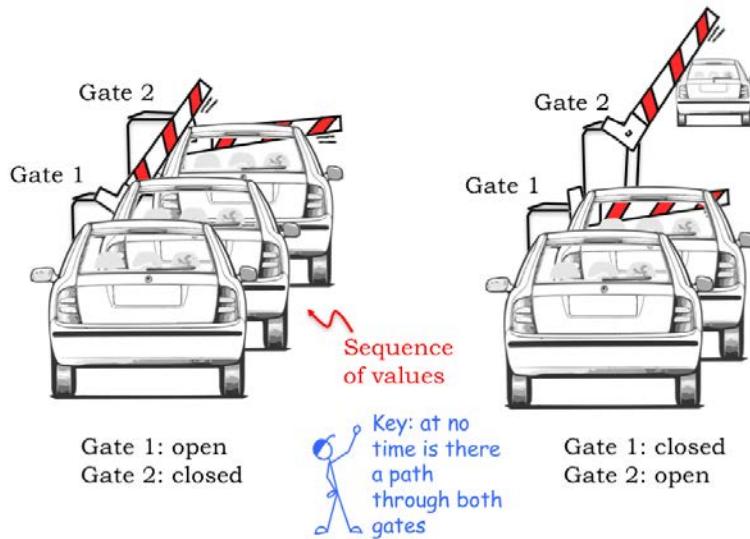


Here's an analogy that will help us understand what's happening and what we can do about it. Imagine a line cars waiting at a toll booth gate. The sequence of cars represents the sequence of states in our sequential logic and the gated toll both represents the latch.

Initially the gate is closed and the cars are waiting patiently to go through the toll booth. When the gate opens, the first car proceeds out of the toll booth. But you can see that the timing of when to close the gate is going to be tricky. It has to be open long enough for the first car to make it through, but not too long lest the other cars also make it through. This is exactly the issue we faced with using the latch as our memory component in our sequential logic.

So how do we ensure only one car makes it through the open gate?

Solution: Escapement Strategy (2 gates)



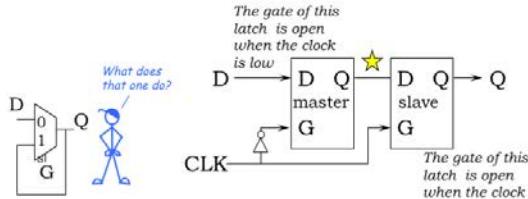
One solution is to use *two gates*! Here's the plan: Initially Gate 1 is open allowing exactly one car to enter the toll booth and Gate 2 is closed. Then at a particular point in time, we close Gate 1 while opening Gate 2. This lets the car in the toll booth proceed on, but prevents any other car from passing through. We can repeat this two-step process to deal with each car one-at-time. The key is that at no time is there a path through both gates.

This is the same arrangement as the escapement mechanism in a mechanical clock. The escapement ensures that the gear attached to the clock's spring only advances one tooth at a time, preventing the spring from spinning the gear wildly causing a whole day to pass at once!

If we observed the toll booth's output, we would see a car emerge shortly after the instant in time when Gate 2 opens. The next car would emerge shortly after the next time Gate 2 opens, and so on. Cars would proceed through the toll booth at a rate set by the interval between Gate 2 openings.

Let's apply this solution to design a memory component for our sequential logic.

(Edge-Triggered) D Register



Observations:

- only one latch “transparent” at any time:
 - master closed when slave is open
 - slave closed when master is open
- ⇒ no combinational path through register
(the feedback path in one of the master or slave latches is always active)

Taking our cue from the 2-gate toll both, we'll design a new component, called a D register, using two back-to-back latches. The load signal for a D register is typically called the register's “clock”, but the register's D input and Q output play the same roles as they did for the latch.

First we'll describe the internal structure of the D register, then we'll describe what it does and look in detail at how it does it.

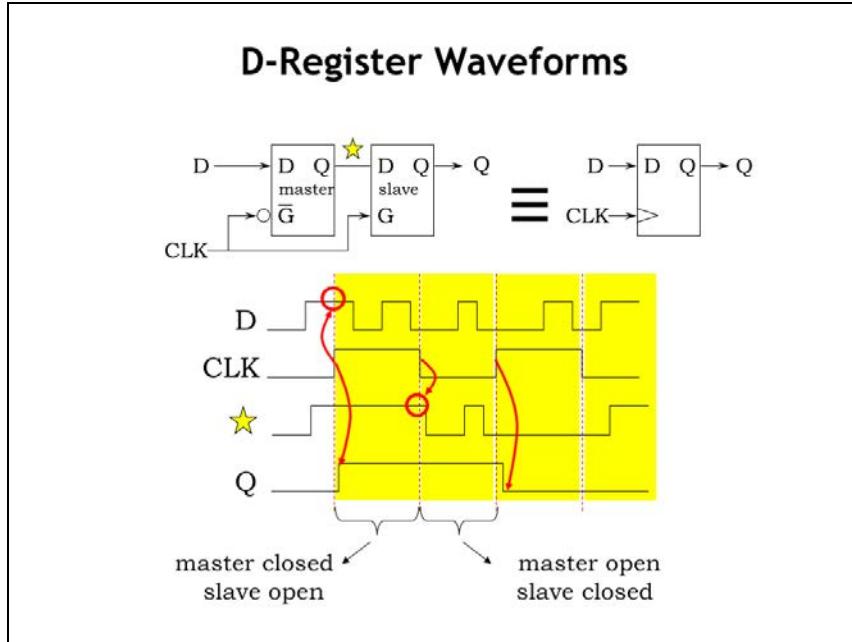
The D input is connected to what we call the master latch and the Q output is connected to the slave latch.

Note that the clock signal is inverted before it's connected to the gate input of the master latch. So when the master latch is open, the slave is closed, and vice versa. This achieves the escapement behavior we saw on the previous slide: at no time is there active path from the register's D input to the register's Q output.

The delay introduced by the inverter on the clock signal might give us cause for concern. When there's a rising 0-to-1 transition on the clock signal, might there be a brief interval when the gate signal is HIGH for both latches since there will be a small delay before the inverter's output transitions from 1 to 0? Actually the inverter isn't necessary: Mr Blue is looking at a slightly different latch schematic where the latch is open when G is LOW and closed when G is high. Just what we need for the master latch!

By the way, you'll sometimes hear a register called a flip-flop because of the bistable nature of the positive feedback loops in the latches.

That's the internal structure of the D register. In the next section we'll take a step-by-step tour of the register in operation.



We'll get a good understanding of how the register operates as we follow the signals through the circuit.

The overall operation of the register is simple: At the rising 0-to-1 transition of the clock input, the register samples the value of the D input and stores that value until the next rising clock edge. The Q output is simply the value stored in the register. Let's see how the register implements this functionality.

The clock signal is connected to the gate inputs of the master and slave latches. Since all the action happens when the clock makes a transition, it's those events we'll focus on. The clock transition from LOW to HIGH is called the rising edge of the clock. And its transition from HIGH to LOW is called the falling edge. Let's start by looking at the operation of the master latch and its output signal, which is labeled STAR in the diagram.

On the rising edge of the clock, the master latch goes from open to closed, sampling the value on its input and entering memory mode. The sampled value thus becomes the output of the latch as long as the latch stays closed. You can see that the STAR signal remains stable whenever the clock signal is high.

On the falling edge of the clock the master latch opens and its output will then reflect any changes in the D input, delayed by the tPD of the latch.

Now let's figure out what the slave is doing. Its output signal, which also serves as the output of D register, is shown as the bottom waveform. On the rising edge of the clock the slave latch opens and its output will follow the value of the STAR signal. Remember though that the STAR signal is stable while the clock is HIGH since the master latch is closed, so the Q signal is also stable after an initial transition if value saved in the slave latch is changing.

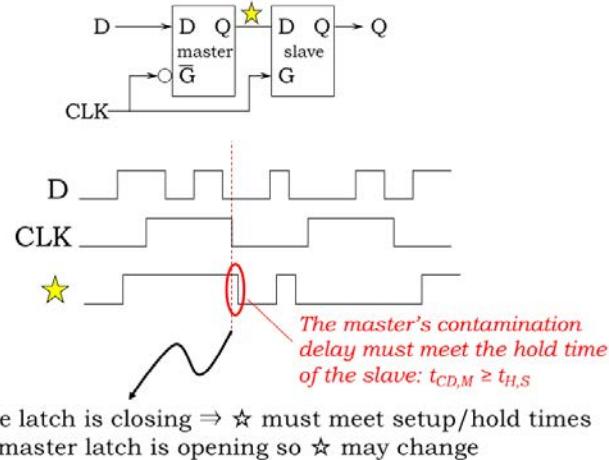
At the falling clock edge [CLICK], the slave goes from open to closed, sampling the value on its input and entering memory mode. The sampled value then becomes the output of the slave latch as long as

the latch stays closed. You can see that that the Q output remains stable whenever the clock signal is LOW.

Now let's just look at the Q signal by itself for a moment. It only changes when the slave latch opens at the rising edge of the clock. The rest of the time either the input to slave latch is stable or the slave latch is closed. The change in the Q output is triggered by the rising edge of the clock, hence the name "positive-edge-triggered D register".

The convention for labeling the clock input in the schematic icon for an edge-triggered device is to use a little triangle. You can see that here in the schematic symbol for the D register.

Um, about that hold time...

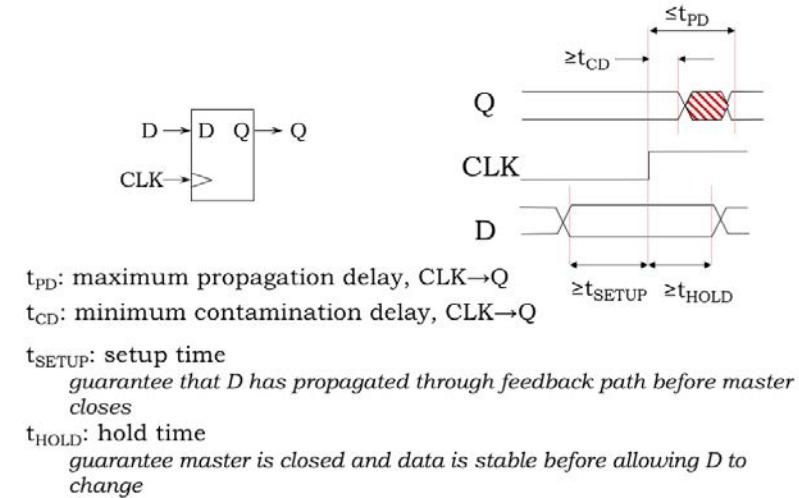


There is one tricky problem we have to solve when designing the circuitry for the register. On the falling clock edge, the slave latch transitions from open to closed and so its input (the STAR signal) must meet the setup and hold times of the slave latch in order to ensure correct operation.

The complication is that the master latch opens at the same time, so the STAR signal may change shortly after the clock edge. The contamination delay of the master latch tells us how long the old value will be stable after the falling clock edge. And the hold time on the slave latch tells us how long it has to remain stable after the falling clock edge.

So to ensure correct operation of the slave latch, the contamination delay of the master latch has to be greater than or equal to the hold time of the slave latch. Doing the necessary analysis can be a bit tricky since we have to consider manufacturing variations as well as environmental factors such as temperature and power supply voltage. If necessary, extra gate delays (e.g., pairs of inverters) can be added between the master and slave latches to increase the contamination delay on the slave's input relative to the falling clock edge. Note that we can only solve slave latch hold time issues by changing the design of the circuit.

D-Register Timing 1



Here's a summary of the timing specifications for a D register.

Changes in the Q signal are triggered by a rising edge on the clock input. The propagation delay t_{PD} of the register is an upper bound on the time it takes for the Q output to become valid and stable after the rising clock edge.

The contamination delay of the register is a lower bound on the time the previous value of Q remains valid after the rising clock edge.

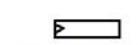
Note that both t_{CD} and t_{PD} are measured relative to the rising edge of the clock. Registers are designed to be lenient in the sense that if the previous value of Q and the new value of Q are the same, the stability of the Q signal is guaranteed during the rising clock edge. In other words, the t_{CD} and t_{PD} specifications only apply when the Q output actually changes.

In order to ensure correct operation of the master latch, the register's D input must meet the setup and hold time constraints for the master latch. So the following two specifications are determined by the timing of the master latch.

t_{SETUP} is the amount of time that the D input must be valid and stable before the rising clock edge and t_{HOLD} is the amount of time that D must be valid and stable after the rising clock. This region of stability surrounding the clock edge ensures that we're obeying the dynamic discipline for the master latch.

So when you use a D register component from a manufacturer's gate library, you'll need to look up these four timing specifications in the register's data sheet in order to analyze the timing of your overall circuit. We'll see how this analysis is done in the next section.

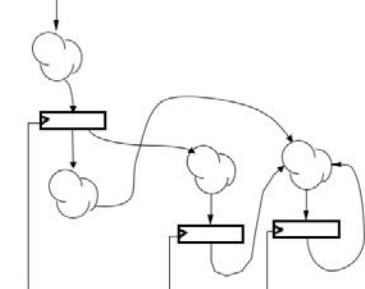
Single-clock Synchronous Circuits



We'll use registers in a highly constrained way to build digital systems:



Does that symbol register?



Single-clock Synchronous Discipline

- No combinational cycles
- Single periodic clock signal shared among all clocked devices
- Only care about value of register data inputs just before rising edge of clock
- Period greater than every combinational delay + setup time
- Change saved state after noise-inducing logic transitions have stopped!

In 6.004, we have a specific plan on how we'll use registers in our designs, which we call the single-clock synchronous discipline.

Looking at the sketch of a circuit on the left, we see that it consists of registers — the rectangular icons with the edge-triggered symbol — and combinational logic circuits, shown here as little clouds with inputs and outputs.

Remembering that there is no combinational path between a register's input and output, the overall circuit has no combinational cycles. In other words, paths from system inputs and register outputs to the inputs of registers never visit the same combinational block twice.

A single periodic clock signal is shared among all the clocked devices. Using multiple clock signals is possible, but analyzing the timing for signals that cross between clock domains is quite tricky, so life is much simpler when all registers use the same clock.

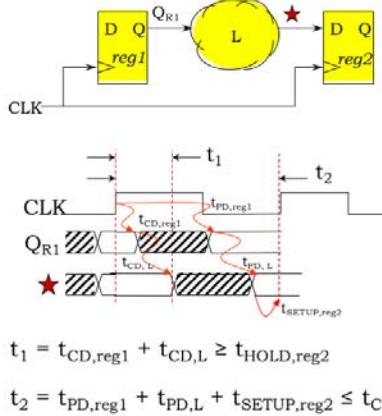
The details of which data signals change when are largely unimportant. All that matters is that signals hooked to register inputs are stable and valid for long enough to meet the registers' setup time. And, of course, stay stable long enough to meet the registers' hold time.

We can guarantee that the dynamic discipline is obeyed by choosing the clock period to be greater than the t_{PD} of every path from register outputs to register inputs, plus, of course, the registers' setup time.

A happy consequence of choosing the clock period in this way is that at the moment of the rising clock edge, there are no other noise-inducing logic transitions happening anywhere in the circuit. Which means there should be no noise problems when we update the stored state of each register.

Our next task is to learn how to analyze the timing of a single-clock synchronous system.

Timing in a Single-clock System



Questions for register-based designs:

- how much time for useful work (i.e. for combinational logic delay)?
- what happens if there's no combinational logic between two registers?
- what happens if CLK signal doesn't arrive at the two registers at exactly the same time (a phenomenon known as "clock skew")?

Here's a model of a particular path in our synchronous system. A large digital system will have many such paths and we have to do the analysis below for each one in order to find the path that will determine the smallest workable clock period. As you might suspect, there are computed-aided design programs that will do these calculations for us.

There's an upstream register, whose output is connected to a combinational logic circuit which generates the input signal, labeled STAR, to the downstream register.

Let's build a carefully-drawn timing diagram showing when each signal in the system changes and when it is stable.

The rising edge of the clock triggers the upstream register, whose output (labeled Q_{REG1}) changes as specified by the contamination and propagation delays of the register. Q_{REG1} maintains its old value for at least the contamination delay of REG1, and then reaches its final stable value by the propagation delay of REG1. At this point Q_{REG1} will remain stable until the next rising clock edge.

Now let's figure out the waveforms for the output of the combinational logic circuit, marked with a red star in the diagram. The contamination delay of the logic determines the earliest time STAR will go invalid measured from when Q_{REG1} went invalid. The propagation delay of the logic determines the latest time STAR will be stable measured from when Q_{REG1} became stable.

Now that we know the timing for STAR, we can determine whether STAR will meet the setup and hold times for the downstream register REG2. Time t_1 measures how long STAR will stay valid after the rising clock edge. t_1 is the sum of REG1's contamination delay and the logic's contamination delay. The HOLD time for REG2 measures how long STAR has to stay valid after the rising clock edge in order to ensure correct operation. So t_1 has to be greater than or equal to the HOLD time for REG2.

Time t_2 is the sum of the propagation delays for REG1 and the logic, plus the SETUP time for REG2. This tells us the earliest time at which the next rising clock edge can happen and still ensure that the SETUP time for REG2 is met. So t_2 has to be less than or equal to the time between rising clock edges,

called the clock period or t_{CLK}. If the next rising clock happens before t₂, we'll be violating the dynamic discipline for REG2.

So we have two inequalities that must be satisfied for every register-to-register path in our digital system. If either inequality is violated, we won't be obeying the dynamic discipline for REG2 and our circuit will not be guaranteed to work correctly.

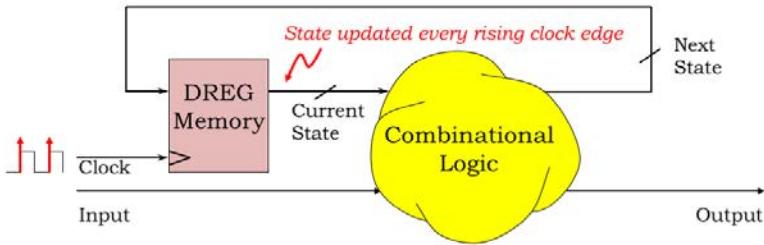
Looking at the inequality involving t_{CLK}, we see that the propagation delay of the upstream register and setup time for the downstream register take away from the time available useful work performed by the combinational logic. Not surprisingly, designers try to use registers that minimize these two times.

What happens if there's no combinational logic between the upstream and downstream registers? This happens when designing shift registers, digital delay lines, etc. Well, then the first inequality tells us that the contamination delay of the upstream register had better be greater than or equal to the hold time of the downstream register. In practice, contamination delays are smaller than hold times, so in general this wouldn't be the case. So designers are often required to insert dummy logic, *e.g.*, two inverters in series, in order to create the necessary contamination delay.

Finally we have to worry about the phenomenon called clock skew, where the clock signal arrives at one register before it arrives at the other. We won't go into the analysis here, but the net effect is to increase the apparent setup and hold times of the downstream register, assuming we can't predict the sign of the skew.

The clock period, t_{CLK}, characterizes the performance of our system. You may have noticed that Intel is willing to sell you processor chips that run at different clock frequencies, *e.g.*, a 1.7 GHz processor vs. a 2 GHz processor. Did you ever wonder how those chips are different? As it turns out they're not! What's going on is that variations in the manufacturing process mean that some chips have better tPDs than others. On fast chips, a smaller tPD for the logic means that they can have a smaller t_{CLK} and hence a higher clock frequency. So Intel manufactures many copies of the same chip, measures their tPDs and selects the fast ones to sell as higher-performance parts. That's what it takes to make money in the chip biz!

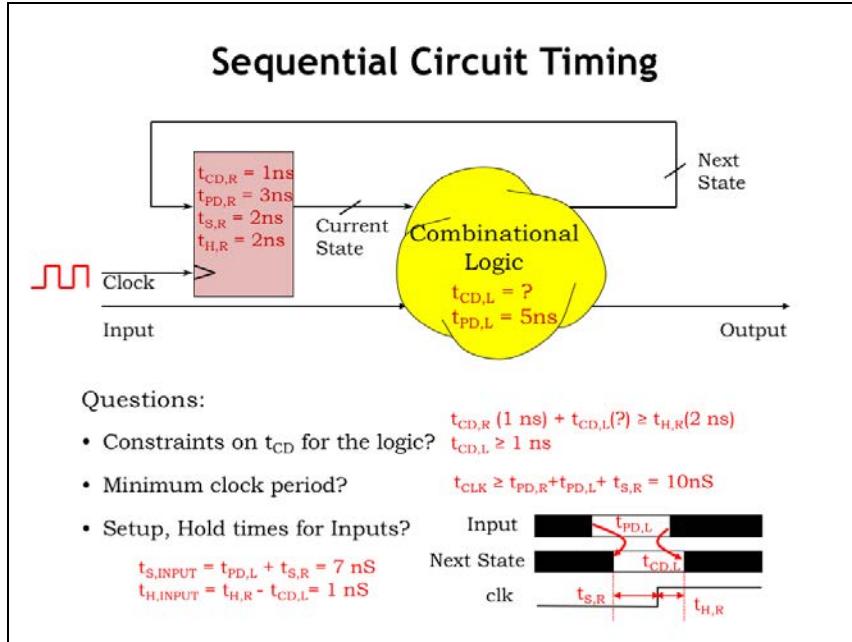
Model: Discrete Time



Active Clock Edges punctuate time ---

- Discrete Clock periods
- Sequences of states
- Simple rules – eg truth tables – relating outputs to inputs and the current state)
- ABSTRACTION: Finite State Machines (next lecture!)

Using a D register as the memory component in our sequential logic system works great! At each rising edge of the clock, the register loads the new state, which then appears at the register's output as the current state for the rest of the clock period. The combinational logic uses the current state and the value of the inputs to calculate the next state and the values for the outputs. A sequence of rising clock edges and inputs will produce a sequence of states, which leads to a sequence of outputs. In the next chapter we'll introduce a new abstraction, finite state machines, that will make it easy to design sequential logic systems.



Let's use the timing analysis techniques we've learned on the sequential logic system shown here. The timing specifications for the register and combinational logic are as shown. Here are the questions we need to answer.

The contamination delay of the combinational logic isn't specified. What does it have to be in order for the system to work correctly? Well, we know that the sum of register and logic contamination delays has to be greater than or equal to the hold time of the register. Using the timing parameters we do know along with a little arithmetic tells us that the contamination delay of the logic has to be at least 1 ns.

What is the minimum value for the clock period t_{CLK} ? The second timing inequality from the previous section tells us that t_{CLK} has to be greater than the sum of the register and logic propagation delays plus the setup time of the register. Using the known values for these parameters gives us a minimum clock period of 10ns.

What are the timing constraints for the Input signal relative to the rising edge of the clock? For this we'll need a diagram! The Next State signal is the input to the register so it has to meet the setup and hold times as shown here. Next we show the Input signal and how the timing of its transitions affect the timing of the Next State signal. Now it's pretty easy to figure out when Input has to be stable before the rising edge of the clock, *i.e.*, the setup time for Input. The setup time for Input is the sum of propagation delay of the logic plus the setup time for the register, which we calculate as 7ns. In other words, if the Input signal is stable at least 7ns before the rising clock edge, then Next State will be stable at least 2ns before the rising clock edge and hence meet the register's specified setup time.

Similarly, the hold time of Input has to be the hold time of the register minus the contamination delay of the logic, which we calculate as 1 ns. In other words, if Input is stable at least 1 ns after the rising clock edge, then Next State will be stable for another 1 ns, *i.e.*, a total of 2 ns after the rising clock edge. This meets the specified hold time of the register.

This completes our introduction to sequential logic. Pretty much every digital system out there is a sequential logic system and hence is obeying the timing constraints imposed by the dynamic discipline. So next time you see an ad for a 1.7 GHz processor chip, you'll know where the "1.7" came from!

Summary

Basic memory elements:

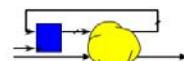
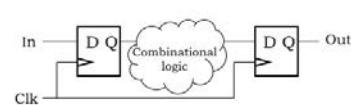
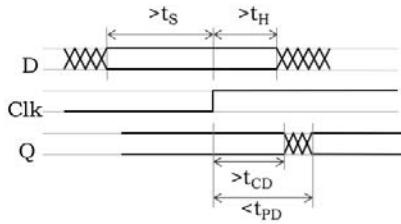
- Feedback, detailed analysis
=> basic level-sensitive devices (eg, latch)
- 2 Latches => Register
- Dynamic Discipline: constraints on input timing

Synchronous 1-clock logic:

- Simple rules for sequential circuits
- Yields clocked circuit with T_S , T_H constraints on input timing

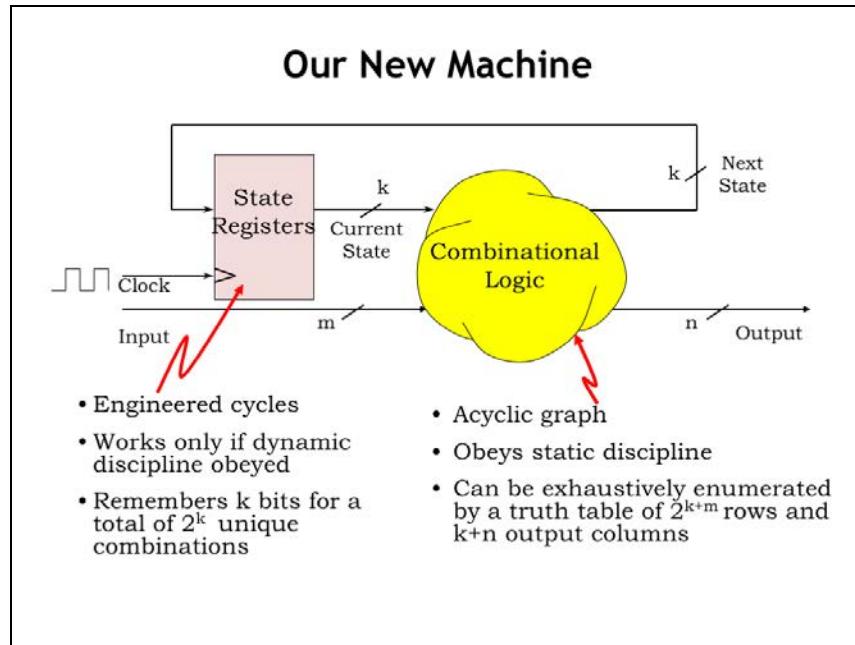
Finite State Machines

Next Lecture Topic!



6: Finite State Machines

1. Our New Machine
2. A Simple Sequential Circuit
3. Abstraction du jour: Finite State Machines
4. State Transition Diagram
5. Valid State Diagrams
6. State Transition Diagram as a Truth Table
7. Now Put It in Hardware
8. Discrete Time, Discrete State
9. Housekeeping Issues...
10. FSM States
11. What's My Transition Diagram?
12. FSM Equivalence
13. Let's Build a RoboAnt
14. Lost in Space
15. Bonk!
16. A Little to the Right...
17. Then a Little to the Left...
18. Dealing With Outside Corners
19. Equivalent State Reduction
20. An Evolutionary Step
21. Building the Transition Table
22. Implementation Details
23. Ant Schematic
24. FSMs All the Way Down?
25. The World Doesn't Run on Our Clock!
26. The Bounded-tie Synchronizer
27. Unsolvable? That Can't Be True...
28. The Mysterious Metastable State
29. Metastable State: Properties
30. Solution: Delay Increases Reliability



In the last chapter we developed sequential logic, which contains both combinational logic and memory components.

The combinational logic cloud is an acyclic graph of components that obeys the static discipline. The static discipline guarantees if we supply valid and stable digital inputs, then we will get valid and stable digital outputs by some specified interval after the last input transition. There's also a functional specification that tells us the output values for every possible combination of input values. In this diagram, there are $k+m$ inputs and $k+n$ outputs, so the truth table for the combinational logic will have 2^{k+m} rows and $k+n$ output columns.

The job of the state registers is to remember the current state of the sequential logic. The state is encoded as some number k of bits, which will allow us to represent 2^k unique states. Recall that the state is used to capture, in some appropriate way, the relevant history of the input sequence. To the extent that previous input values influence the operation of the sequential logic, that happens through the stored state bits. Typically the LOAD input of the state registers is triggered by the rising edge of a periodic clock signal, which updates the stored state with the new state calculated by the combinational logic.

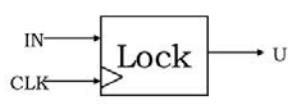
As designers we have several tasks: first we must decide what output sequences need to be generated in response to the expected input sequences. A particular input may, in fact, generate a long sequence of output values. Or the output may remain unchanged while the input sequence is processed, step-by-step, where the FSM is remembering the relevant information by updating its internal state. Then we have to develop the functional specification for the logic so it calculates the correct output and next state values. Finally, we need to come up with an actual circuit diagram for sequential logic system.

All the tasks are pretty interesting, so let's get started!

A Simple Sequential Circuit...

Lets make a digital binary *Combination Lock*:

Specification:



How many state bits do I need?



- One input ("0" or "1")
- One output ("Unlock" signal)
- UNLOCK is 1 if and only if:

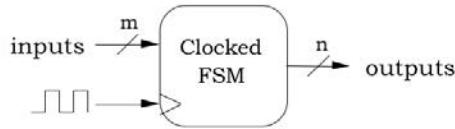
Last 4 inputs were the "combination": 0110

As an example sequential system, let's make a combination lock. The lock has a 1-bit input signal, where the user enters the combination as a sequence of bits. There's one output signal, UNLOCK, which is 1 if and only if the correct combination has been entered. In this example, we want to assert UNLOCK, *i.e.*, set UNLOCK to 1, when the last four input values are the sequence 0-1-1-0.

Mr. Blue is asking a good question: how many state bits do we need? Do we have to remember the last four input bits? In which case, we'd need four state bits. Or can we remember less information and still do our job? Aha! We don't need the complete history of the last four inputs, we only need to know if the most recent entries represent some part of a partially-entered correct combination. In other words if the input sequence doesn't represent a correct combination, we don't need to keep track of exactly how it's incorrect, we only need to know that is incorrect.

With that observation in mind, let's figure out how to represent the desired behavior of our digital system.

Abstraction du jour: Finite State Machines



A FINITE STATE MACHINE has

- k STATES: $S_1 \dots S_k$ (one is “initial” state)
- m INPUTS: $I_1 \dots I_m$
- n OUTPUTS: $O_1 \dots O_n$
- Transition Rules: $s'(s, I)$ for each state s and input I
- Output Rules: $Out(s)$ or $Out(s, I)$ for each state s and input I

We can characterize the behavior of a sequential system using a new abstraction called a finite state machine, or FSM for short. The goal of the FSM abstraction is to describe the input/output behavior of the sequential logic, independent of its actual implementation.

A finite state machine has a periodic CLOCK input. A rising clock edge will trigger the transition from the current state to the next state. The FSM has a some fixed number of states, with a particular state designated as the initial or starting state when the FSM is first turned on. One of the interesting challenges in designing an FSM is to determine the required number of states since there's often a tradeoff between the number of state bits and the complexity of the internal combinational logic required to compute the next state and outputs.

There are some number of inputs, used to convey all the external information necessary for the FSM to do its job. Again, there are interesting design tradeoffs. Suppose the FSM required 100 bits of input. Should we have 100 inputs and deliver the information all at once? Or should we have a single input and deliver the information as a 100-cycle sequence? In many real world situations where the sequential logic is *much* faster than whatever physical process we’re trying to control, we’ll often see the use of bit-serial inputs where the information arrives as a sequence, one bit at a time. That will allow us to use much less signaling hardware, at the cost of the time required to transmit the information sequentially.

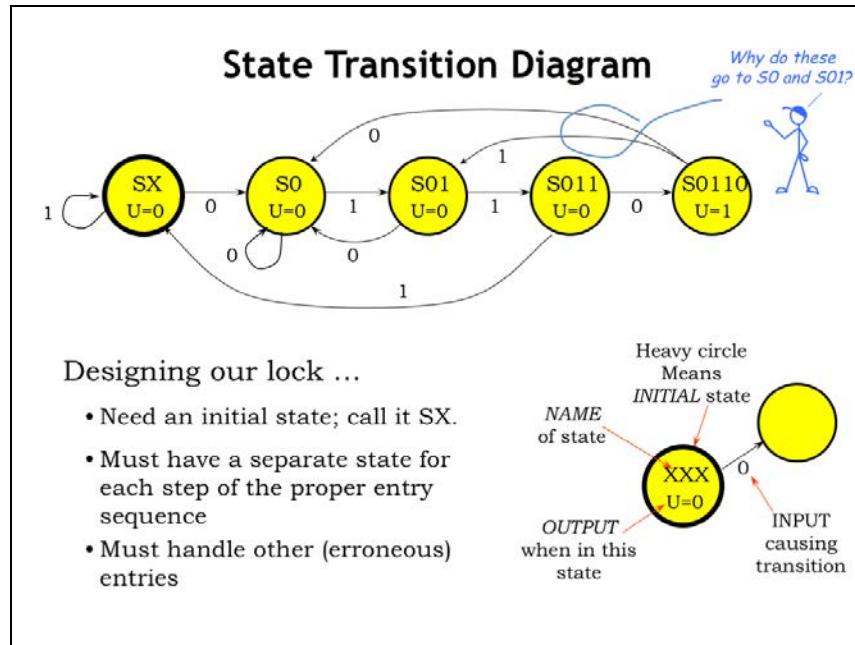
The FSM has some number outputs to convey the results of the sequential logic’s computations. The comments before about serial vs. parallel inputs applies equally to choosing how information should be encoded on the outputs.

There are a set of transition rules, specifying how the next state S' is determined from the current state S and the inputs I . The specification must be complete, enumerating S' for every possible combination of S and I .

And, finally, there’s the specification for how the output values should be determined. The FSM design is often a bit simpler if the outputs are strictly a function of the current state S , but, in general, the outputs can be a function of both S and the current inputs.

Now that we have our abstraction in place, let's see how to use it to design our combinational lock.

\section{State Transition Diagrams}



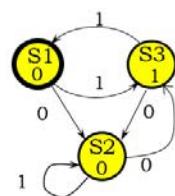
We'll describe the operation of the FSM for our combination lock using a state transition diagram. Initially, the FSM has received no bits of the combination, a state we'll call SX. In the state transition diagram, states are represented as circles, each labeled for now with a symbolic name chosen to remind us of what history it represents. For this FSM, the unlock output U will be a function of the current state, so we'll indicate the value of U inside the circle. Since in state SX we know nothing about past input bits, the lock should stay locked and so $U = 0$. We'll indicate the initial state with a wide border on the circle.

We'll use the successive states to remember what we've seen so far of the input combination. So if the FSM is in state SX and it receives a 0 input, it should transition to state S0 to remind us that we've seen the first bit of the combination of 0-1-1-0. We use arrows to indicate transitions between states and each arrow has a label telling us when that transition should occur. So this particular arrow is telling us that when the FSM is in state SX and the next input is a 0, the FSM should transition to state S0. Transitions are triggered by the rising edge of the FSM's clock input.

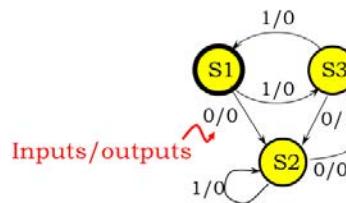
Let's add the states for the remainder of the specified combination. The rightmost state, S0110, represents the point at which the FSM has detected the specified sequence of inputs, so the unlock signal is 1 in this state. Looking at the state transition diagram, we see that if the FSM starts in state SX, the input sequence 0-1-1-0 will leave the FSM in state S0110.

So far, so good. What should the FSM do if an input bit is not the next bit in the combination? For example, if the FSM is in state SX and the input bit is a 1, it still has not received any correct combination bits, so the next state is SX again. Here are the appropriate non-combination transitions for the other states. Note that an incorrect combination entry doesn't necessarily take the FSM to state SX. For example, if the FSM is in state S0110, the last four input bits have been 0-1-1-0. If the next input is a 1, then the last four inputs bits are now 1-1-0-1, which won't lead to an open lock. But the last two bits might be the first two bits of a valid combination sequence and so the FSM transitions to S01, indicating that a sequence of 0-1 has been entered over the last two bits.

Valid State Diagrams



MOORE Machine:
Outputs on States



MEALY Machine:
Outputs on Transitions

Arcs leaving a state must be:

- (1) **mutually exclusive**
 - can't have two choices for a given input value
- (2) **collectively exhaustive**
 - every state must specify what happens for each possible input combination. "Nothing happens" means arc back to itself.

We've been working with an FSM where the outputs are function of the current state, called a Moore machine. Here the outputs are written inside the state circle.

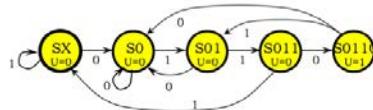
If the outputs are a function of both the current state and the current inputs, it's called a Mealy machine. Since the transitions are also a function of the current state and current inputs, we'll label each transition with appropriate output values using a slash to separate input values from output values. So, looking at the state transition diagram on the right, suppose the FSM is in state S3. If the input is a 0, look for the arrow leaving S3 labeled "0/0". The value after the slash tells us the output value, in this case 1. If the input had been a 1, the output value would be 0.

There are some simple rules we can use to check that a state transition diagram is well formed. The transitions from a particular state must be mutually exclusive, *i.e.*, for a each state, there can't be more than one transition with the same input label. This makes sense: if the FSM is to operate consistently there can't be any ambiguity about the next state for a given current state and input. By "consistently" we mean that the FSM should make the same transitions if it's restarted at the same starting state and given the same input sequences.

Moreover, the transitions leaving each state should be collectively exhaustive, *i.e.*, there should a transition specified for each possible input value. If we wish the FSM to stay in its current state for that particular input value, we need to show a transition from the current state back to itself.

With these rules there will be exactly one transition selected for every combination of current state and input value.

State Transition Diagram as a Truth Table



IN	Current State		Next State	Unlock
0	000	SX	S0	001 0
1	000	SX	SX	000 0
0	001	S0	S0	001 0
1	001	S0	S01	011 0
0	011	S01	S0	001 0
1	011	S01	S011	010 0
0	010	S011	S0110	100 0
1	010	S011	SX	000 0
0	100	S0110	S0	001 1
1	100	S0110	S01	011 1



The assignment of codes to states can be arbitrary, however, if you choose them carefully you can greatly reduce your logic requirements.

All state transition diagrams can be described by truth tables...



Binary encodings are assigned to each state (a bit of an art)

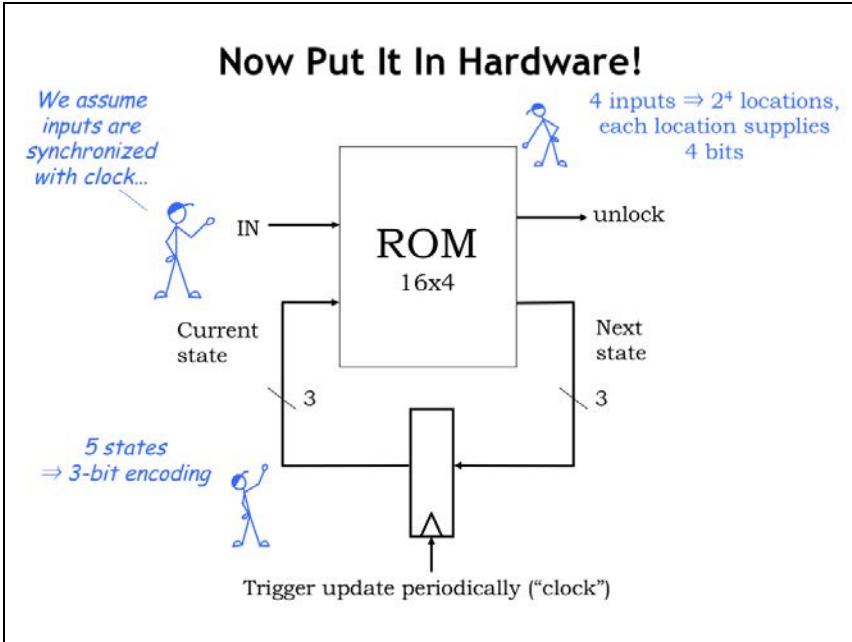
The truth table can then be simplified using the reduction techniques we learned for combinational logic

All the information in a state transition diagram can be represented in tabular form as a truth table. The rows of the truth table list all the possible combinations of current state and inputs. And the output columns of the truth table tell us the next state and output value associated with each row.

If we substitute binary values for the symbolic state names, we end up with a truth table just like the ones we saw in Chapter 4. If we have K states in our state transition diagram we'll need $\log_2 K$ state bits, rounded up to the next integer since we don't have fractional bits! In our example, we have a 5-state FSM, so we'll need 3 state bits.

We can assign the state encodings in any convenient way, e.g., 000 for the first state, 001 for the second state, and so on. But the choice of state encodings can have a big effect on the logic needed to implement the truth table. It's actually fun to figure out the state encoding that produces the simplest possible logic.

With a truth table in hand, we can use the techniques from Chapter 4 to design logic circuits that implement the combinational logic for the FSM. Of course, we can take the easy way out and simply use a read-only memory to do the job!



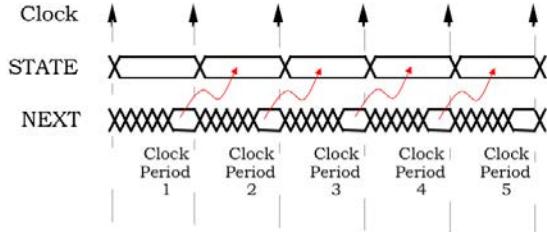
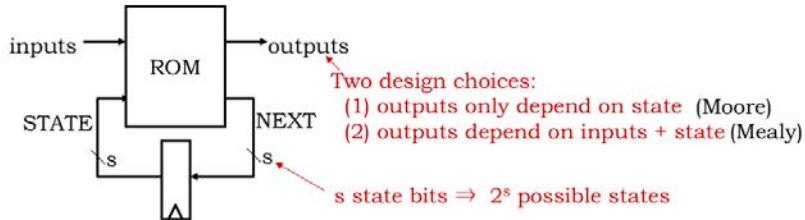
w

In this circuit, a read-only memory is used to compute the next state and outputs from the current state and inputs. We're encoding the 5 states of the FSM using a 3-bit binary value, so we have a 3-bit state register. The rectangle with the edge-triggered input is schematic shorthand for a multi-bit register. If a wire in the diagram represents a multi-bit signal, we use a little slash across the wire with a number to indicate how many bits are in the signal. In this example, both `current_state` and `next_state` are 3-bit signals.

The read-only memory has a total of 4 input signals — 3 for the current state and 1 for the input value — so the read-only memory has $2^4 = 16$ locations, which correspond to the 16 rows in the truth table. Each location in the ROM supplies the output values for a particular row of the truth table. Since we have 4 output signals — 3 for the next state and 1 for the output value — each location supplies 4 bits of information. Memories are often annotated with their number of locations and the number of bits in each location. So our memory is a 16-by-4 ROM: 16 locations of 4-bits each.

Of course, in order for the state registers to work correctly, we need to ensure that the dynamic discipline is obeyed. We can use the timing analysis techniques described at the end of Chapter 5 to check that this is so. For now, we'll assume that the timing of transitions on the inputs are properly synchronized with the rising edges of the clock.

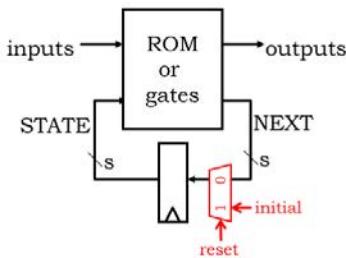
Discrete State, Discrete Time



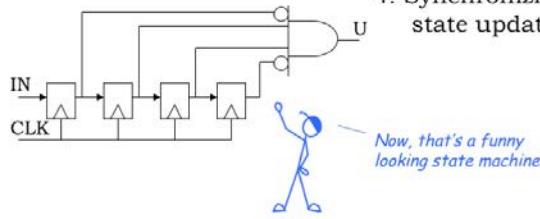
So now we have the FSM abstraction to use when designing the functionality of a sequential logic system, and a general-purpose circuit implementation of the FSM using a ROM and a multi-bit state register. Recapping our design choices: the output bits can be strictly a function of the current state (the FSM would then be called a Moore machine), or they can be a function of both the current state and current inputs, in which case the FSM is called a Mealy machine. We can choose the number of state bits — S state bits will give us the ability to encode 2^S possible states. Note that each extra state bit **DOUBLES** the number of locations in the ROM! So when using ROMs to implement the necessary logic, we're very interested in minimizing the number of state bits.

The waveforms for our circuitry are pretty straightforward. The rising edge of the clock triggers a transition in the state register outputs. The ROM then does its thing, calculating the next state, which becomes valid at some point in the clock cycle. This is the value that gets loaded into the state registers at the next rising clock edge. This process repeats over-and-over as the FSM follows the state transitions dictated by the state transition diagram.

Housekeeping Issues...



1. Initialization? Clear the memory?
2. Unused state encodings?
 - waste ROM (use gates)
 - what does it mean?
 - can the FSM recover?
3. Choosing encoding for state?
4. Synchronizing input changes with state update?



There are a few housekeeping details that need our attention.

On start-up we need some way to set the initial contents of the state register to the correct encoding for the initial state. Many designs use a RESET signal that's set to 1 to force some initial state and then set to 0 to start execution. We could adopt that approach here, using the RESET signal to select an initial value to be loaded into the state register.

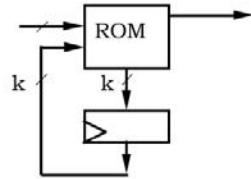
In our example, we used a 3-bit state encoding which would allow us to implement an FSM with up to $2^3 = 8$ states. We're only using 5 of these encodings, which means there are locations in the ROM we'll never access. If that's a concern, we can always use logic gates to implement the necessary combinational logic instead of ROMs. Suppose the state register somehow got loaded with one of the unused encodings? Well, that would be like being in a state that's not listed in our state transition diagram. One way to defend against this problem is design the ROM contents so that unused states always point to the initial state. In theory the problem should never arise, but with this fix at least it won't lead to unknown behavior.

We mentioned earlier the interesting problem of finding a state encoding that minimized the combinational logic. There are computer-aided design tools to help do this as part of the larger problem of finding minimal logic implementations for Boolean functions. Mr. Blue is showing us another approach to building the state register for the combination lock: use a shift register to capture the last four input bits, then simply look at the recorded history to determine if it matches the combinations. No fancy next state logic here!

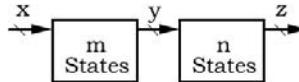
Finally, we still have to address the problem of ensuring that input transitions don't violate the dynamic discipline for the state register. We'll get to this in the last section of this chapter.

FSM States

1. What can you say about the number of states?



2. Same question:



3. Here's an FSM. Can you discover its rules?



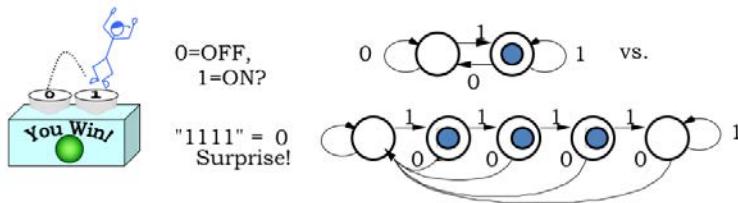
Let's think a bit more about the FSM abstraction.

If we see an FSM that uses K state bits, what can we say about the number of states in its state transition diagram? Well, we know the FSM can have at most 2^K states, since that's the number of unique combinations of K bits.

Suppose we connect two FSMs in series, with the outputs of the first FSM serving as the inputs to the second. This larger system is also an FSM — how many states does it have? Well, if we don't know the details of either of the component FSMs, the upper bound on the number of states for the larger system is $M \times N$. This is because it may be possible for the first FSM to be in any of its M states while the second FSM is any of its N states. Note that the answer doesn't depend on X or Y , the number of input signals to each of the component FSMs. Wider inputs just mean that there are longer labels on the transitions in the state transition diagram, but doesn't tell us anything about the number of internal states.

Finally, here's a question that's a bit trickier than it seems. I give you an FSM with two inputs labeled 0 and 1, and one output implemented as a light. Then I ask you to discover its state transition diagram. Can you do it? Just to be a bit more concrete, you experiment for an hour pushing the buttons in a variety of sequences. Each time you push the 0 button the light turns off if it was on. And when you push the 1 button the light turns on if it was off, otherwise nothing seems to happen. What state transition diagram could we draw based on our experiments?

What's My Transition Diagram?



- If you know **NOTHING** about the FSM, you're never sure!
 - If you have a **BOUND** on the number of states, you can discover its behavior:
 - K-state FSM: Every (reachable) state can be reached in $< k$ steps.
- BUT ... FSMs may be **equivalent!**

Consider the following two state transition diagrams. The top diagram describes the behavior we observed in our experiments: pushing 0 turns the light off, pushing 1 turns the light on.

The second diagram appears to do the same thing unless you happened to push the 1 button 4 times in a row!

If we don't have an upper bound on the number of states in the FSM, we can never be sure that we've explored all of its possible behaviors.

But if we do have an upper bound, say, K, on the number of states and we reset the FSM to its initial state, we can discover its behavior. This is because in a K-state FSM every reachable state can be reached in less than K transitions, starting from the initial state. So if we try all the possible K-step input sequences one after the other starting each trial at the initial state, we'll be guaranteed to have visited every state in the machine.

Our answer is also complicated by the observation that FSMs with different number of states may be equivalent.

FSM Equivalence



ARE THEY DIFFERENT?

NOT in any practical sense! They are EXTERNALLY INDISTINGUISHABLE, hence interchangeable.

FSMs are *EQUIVALENT* iff every input sequence yields identical output sequences.

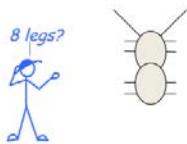
ENGINEERING GOAL:

- HAVE an FSM which *works...*
- WANT simplest (ergo cheapest) equivalent FSM.

Here are two FSMs, one with 2 states, one with 5 states. Are they different? Well, not in any practical sense. Since the FSMs are externally indistinguishable, we can use them interchangeably. We say that two FSMs are equivalent if and only if every input sequence yields identical output sequences from both FSMs.

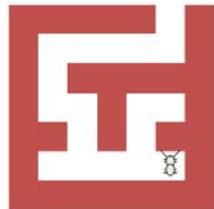
So as engineers, if we have an FSM we would like to find the the simplest (and hence the least expensive) equivalent FSM. We'll talk about how to find smaller equivalent FSMs in the context of our next example.

Let's Build a RoboAnt



- SENSORS: antennae L and R, each 1 if in contact with something.
- ACTUATORS: Forward Step F, ten-degree turns TL and TR (left, right).

GOAL: Make our ant smart enough to get out of a maze like:



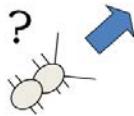
STRATEGY: "Right antenna to the wall"

Surprise! We've just been given a robotic ant that has an FSM for its brain. The inputs to the FSM come from the ant's two antennae, labeled L and R. An antenna input is 1 if the antenna is touching something, otherwise its 0. The outputs of the FSM control the ant's motion. We can make it step forward by setting the F output to 1, and turn left or right by asserting the TL or TR outputs respectively. If the ant tries to both turn and step forward, the turn happens first. Note that the ant can turn when its antennae are touching something, but it can't move forward. We've been challenged to design an ant brain that will let it find its way out of a simple maze like the one shown here.

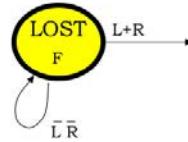
We remember reading that if the maze doesn't have any unconnected walls (*i.e.*, no islands), we can escape using the “right-hand rule” where we put our right hand on the wall and walk so that our hand stays on the wall.

Let's try to implement this strategy.

Lost In Space

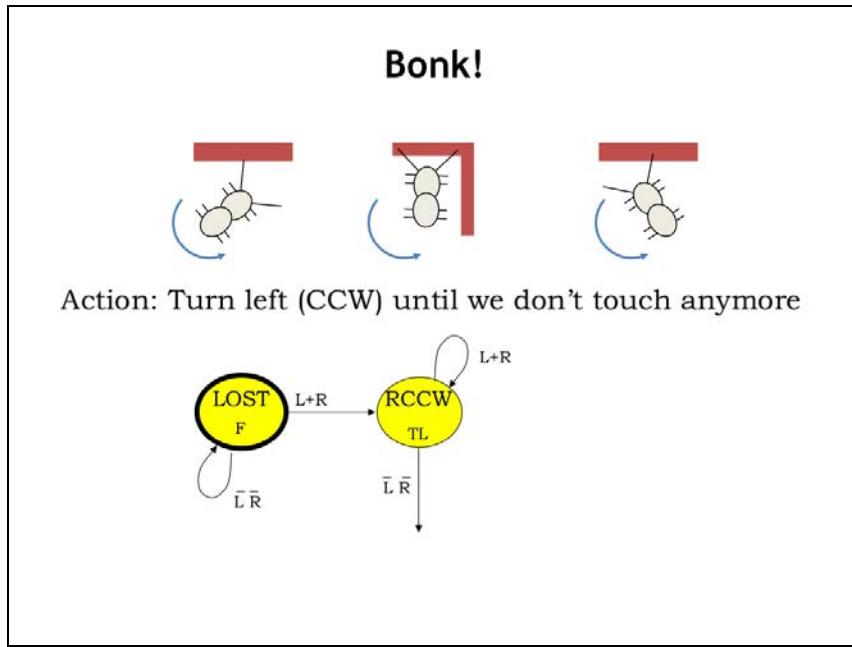


Action: Go forward until we hit something.



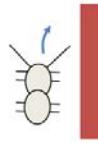
"lost" is the
initial state

We'll assume that initially our ant is lost in space. The only sensible strategy to walk forward until we find a maze wall. So our initial state, labeled LOST, asserts the F output, causing the ant to move forward until at least one of the antennae touches something, *i.e.*, at least one of the L or R inputs is a 1.

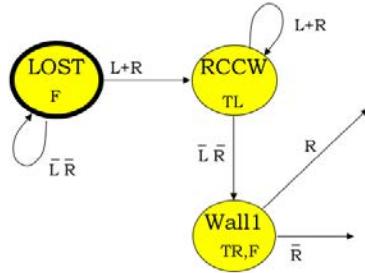


So now the ant finds itself in one of these three situations. To implement the right-hand rule, the ant should turn left (counterclockwise) until its antennae have just cleared the wall. To do this, we'll add a rotate-counterclockwise state, which asserts the turn-left output until both L and R are 0.

A Little to the Right...

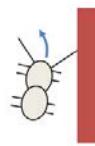


Action: Step and turn right a little, look for wall

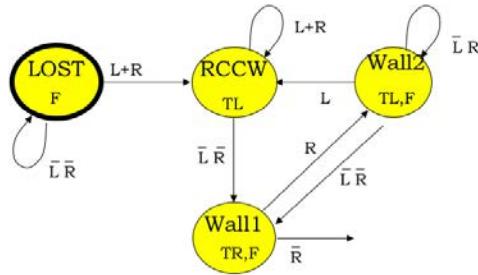


Now the ant is standing with a wall to its right and we can start the process of following the wall with its right antenna. So we have the ant step forward and right, assuming that it will immediately touch the wall again. The WALL1 state asserts both the turn-right and forward outputs, then checks the right antenna to see what to do next.

Then a Little to the Left



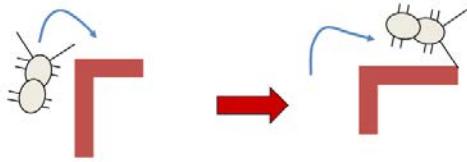
Action: Step and turn left a little, till not touching (again)



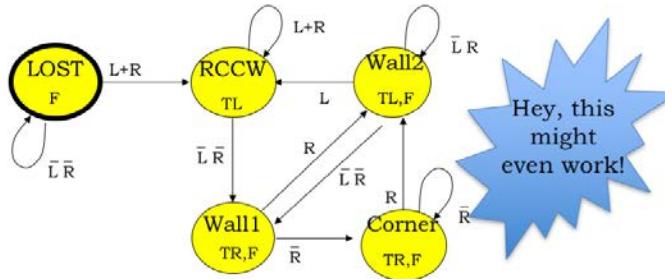
If the right antenna does touch, as expected, the ant turns left to free the antenna and then steps forward. The WALL2 state asserts both the turn-left and forward outputs, then checks the antennae. If the right antenna is still touching, it needs to continue turning. If the left antenna touches, it's run into a corner and needs to reorient itself so the new wall is on its right, the situation we dealt with the rotate-counterclockwise state. Finally, if both antennae are free, the ant should be in the state of the previous slide: standing parallel to the wall, so we return the WALL1 state.

Our expectation is that the FSM will alternate between the WALL1 and WALL2 states as the ant moves along the wall. If it reaches an inside corner, it rotates to put the new wall on its right and keeps going. What happens when it reaches an outside corner?

Dealing With Outside Corners



Action: Step and turn right until we hit perpendicular wall



When the ant is in the WALL1 state, it moves forward and turns right, then checks its right antenna, expecting to find the wall it's traveling along. But if it's an outside corner, there's no wall to touch! The correct strategy in this case is to keep turning right and stepping forward until the right antenna touches the wall that's around the corner. The CORNER state implements this strategy, transitioning to the WALL2 state when the ant reaches the wall again.

Hey, this might even work!

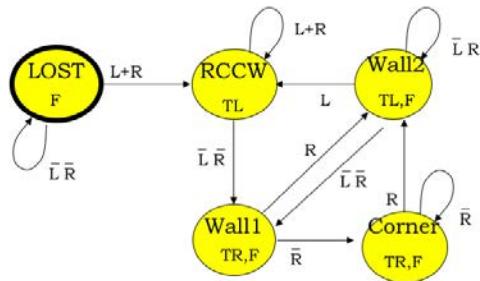
Equivalent State Reduction

Observation: two states are equivalent if

1. Both states have identical outputs; AND
2. Every input \Rightarrow equivalent states.

Reduction Strategy:

Find pairs of equivalent states, MERGE them.



Earlier we talked about finding equivalent FSMs with fewer states. Now we'll develop an approach for finding such FSMs by looking for two states that can be merged into a single state without changing the behavior of the FSM in any externally distinguishable manner.

Two states are equivalent if they meet the following two criteria. First, the states must have identical outputs. This makes sense: the outputs are visible to the outside, so if their values differed between the two states, that difference would clearly be externally distinguishable!

Second, for each combination of input values, the two states transition to equivalent states.

Our strategy for deriving an equivalent machine with fewer states will be to start with our original FSM, find pairs of equivalent states and merge those states. We'll keep repeating the process until we can't find any more equivalent states.

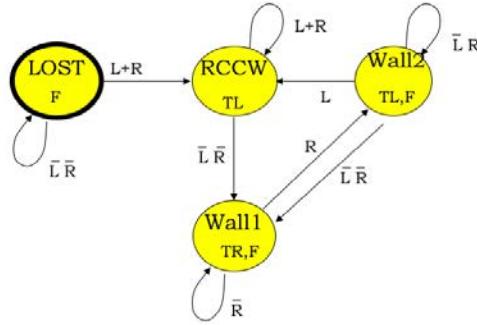
Let's try this on our ant FSM. First we need to find a pair of states that have the same outputs. As it turns out, there's only one such pair: WALL1 and CORNER, both of which assert the turn-right and forward outputs.

Okay, so let's assume that WALL1 and CORNER are equivalent and ask if they transition to equivalent states for each applicable combination of input values. For these two states, all the transitions depend only on the value of the R input, so we just have to check two cases. If R is 0, both states transition to CORNER. If R is 1, both states transition to WALL2.

So both equivalence criteria are satisfied and we can conclude that the WALL1 and CORNER states are equivalent and can be merged.

An Evolutionary Step

Merge equivalent states Wall1 and Corner into a single new, combined state.



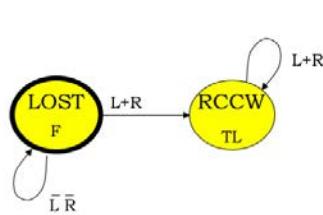
Behaves exactly as previous (5-state) FSM, but requires half the ROM in its implementation!

This gives us the four-state FSM shown here, where we've called the single merged state WALL1. This smaller, equivalent FSM behaves exactly as the previous 5-state FSM. The implementation of the 5-state machine requires 3 state bits; the implementation of the 4-state machine only requires 2 state bits. Reducing the number of state bits by 1 is huge since it reduces the size of the required ROM by half!

Just as we were able to achieve considerable hardware savings by minimizing Boolean equations, we can do the same in sequential logic by merging equivalent states.

Roboant customers are looking forward to the price cut!

Building The Transition Table



S	L	R		S'	TR	TL	F
00	0	0		00	0	0	1
00	0	1		01	0	0	1
00	1	0		01	0	0	1
00	1	1		01	0	0	1
01	0	1		01	0	1	0
01	1	0		01	0	1	0
01	1	1		01	0	1	0

Let's look at what we'd need to do if we wanted to implement the FSM using logic gates instead a ROM for the combinational logic. First we have to build the truth table, entering all the transitions in the state transition diagram. We'll start with the LOST state. So if the FSM is in this state, the F output should be 1. If both antenna inputs are 0, the next state is also LOST. Assigning the LOST state the encoding 00, we've captured this information in the first row of the table.

If either antenna is touching, the FSM should transition from LOST to the rotate-counterclockwise state. We've given this an encoding of 01. There are three combinations of L and R values that match this transition, so we've added three rows to the truth table. This takes care of all the transitions from the LOST state.

Now we can tackle the transitions from the rotate-counterclockwise state. If either antenna is touching, the next state is again rotate-counterclockwise. So we've identified the matching values for the inputs and added the appropriate three rows to the transition table.

We can continue in a similar manner to encode the transitions one-by-one.

Implementation Details

S	L	R		S'	TR	TL	F
00	00	0		00	0	0	1
00	1	-		01	0	0	1
00	-	1		01	0	0	1
01	1	-		01	0	1	0
01	-	1		01	0	1	0
01	0	0		10	0	1	0
10	-	0		10	1	0	1
10	-	1		11	1	0	1
11	1	-		01	0	1	1
11	0	0		10	0	1	1
11	0	1		11	0	1	1

Complete Transition table

		S ₁ '			
		S ₁ S ₀			
		00	01	11	10
LR	00	0	1	1	1
LR	01	0	0	1	1
LR	11	0	0	0	1
LR	10	0	0	0	1

$$S_1' = S_1 \overline{S_0} + \overline{LS}_1 + \overline{LRS}_0$$

		S ₀ '			
		S ₁ S ₀			
		00	01	11	10
LR	00	0	0	0	0
LR	01	1	1	1	1
LR	11	1	1	1	1
LR	10	1	1	1	0

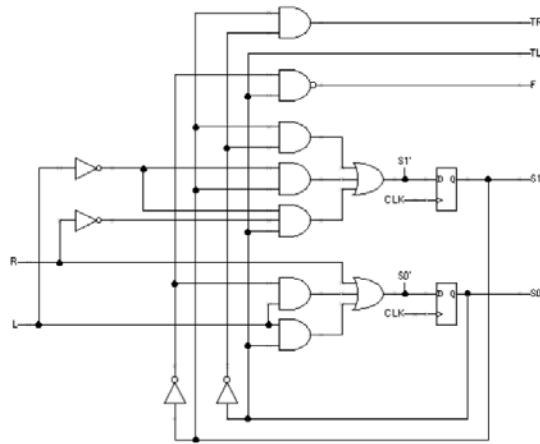
$$S_0' = R + L\overline{S}_1 + LS_0$$

Here's the final table, where we've used don't cares to reduce the number of rows for presentation. Next we want to come up with Boolean equations for each of the outputs of the combinational logic, *i.e.*, the two next-state bits and the three motion-control outputs.

Here are the Karnaugh maps for the two next-state bits. Using our K-map skills from Chapter 4, we'll find a cover of the prime implicants for S₁-prime and write down the corresponding product terms in a minimal sum-of-products equation. And then do the same for the other next-state bit.

We can follow a similar process to derive minimal sum-of-products expressions for the motion-control outputs.

Ant Schematic



Implementing each sum-of-products in a straight-forward fashion with AND and OR gates, we get the following schematic for the ant brain. Pretty neat! Who knew that maze following behavior could be implemented with a couple of D registers and a handful of logic gates?

FSMs All the Way Down?

- More than ants:
Swarming, flocking, and schooling can result from collections of very simple FSMs
- Perhaps most physics:
Cellular automata, arrays of simple FSMs, can more accurately model fluids than numerical solutions to PDEs
- What if:
We replaced the ROM with a RAM and have outputs that modify the RAM?

... You'll see FSMs for the rest of your life!



There are many complex behaviors that can be created with surprisingly simple FSMs. Early on, the computer graphics folks learned that group behaviors like swarming, flocking and schooling can be modeled by equipping each participant with a simple FSM. So next time you see the massive battle scene from the Lord of the Rings movie, think of many FSMs running in parallel!

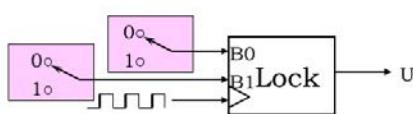
Physical behaviors that arise from simple interactions between component molecules can sometimes be more easily modeled using cellular automata — arrays of communicating FSMS — than by trying to solve the partial differential equations that model the constraints on the molecules' behavior.

And here's an idea: what if we allowed the FSM to modify its own transition table? Hmm. Maybe that's a plausible model for evolution!

FSMs are everywhere! You'll see FSMs for the rest of your life...

The World Doesn't Run on Our Clock!

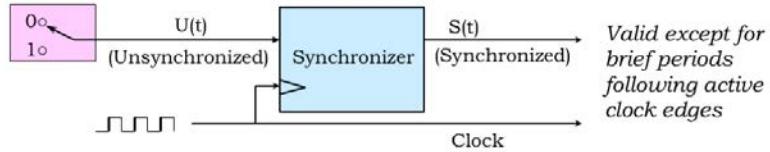
What if each button input is an asynchronous 0/1 level?



But what about the dynamic discipline?

To build a system with asynchronous inputs, we have to break the rules: *we cannot guarantee that setup and hold time requirements are met at the inputs!*

So, we need a “synchronizer” at each input:



Okay, it's finally time to investigate issues caused by asynchronous inputs to a sequential logic circuit. By “asynchronous” we mean that the timing of transitions on the input is completely independent of the timing of the sequential logic clock. This situation arises when the inputs arrive from the outside world where the timing of events is not under our control.

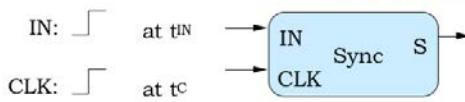
As we saw at the end of Lecture 5, to ensure reliable operation of the state registers, inputs to a sequential logic system have to obey setup and hold time constraints relative to the rising edge of the system clock. Clearly if the input can change at anytime, it can change at time that would violate the setup and hold times.

Maybe we can come up with a synchronizer circuit that takes an unsynchronized input signal and produces a synchronized signal that only changes shortly after the rising edge of the clock. We'd use a synchronizer on each asynchronous input and solve our timing problems that way.

The Bounded-time Synchronizer

A classic ~~problem~~

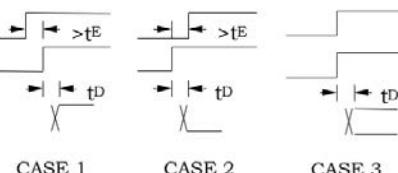
UNRESOLVABLE



For NO finite values of t_E and t_D is this spec realizable, even with reliable components!

Synchronizer specifications:

- finite t_D (decision time)
 - finite t_E (allowable error)
 - value of S at time $t_C + t_D$:
- | | | |
|---|------|--------|
| $1 \quad \text{if } t_{IN} < t_C - t_E$ | IN: | CASE 1 |
| $0 \quad \text{if } t_{IN} > t_C + t_E$ | CLK: | CASE 2 |
| $0, 1 \quad \text{otherwise}$ | S: | CASE 3 |



Here's a detailed specification for our synchronizer.

The synchronizer has two inputs, IN and CLK, which have transitions at time t_{IN} and t_C respectively.

If IN's transition happens sufficiently before C's transition, we want the synchronizer to output a 1 within some bounded time t_D after CLK's transition.

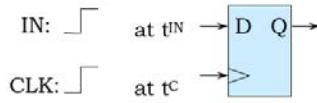
And if CLK's transition happens sufficient before IN's transition, we want the synchronizer to output a 0 within time t_D after CLK's transition.

Finally, if the two transitions are closer together than some specified interval t_E , the synchronizer can output either a 0 or a 1 within time t_D of CLK's transition. Either answer is fine so long as it's a stable digital 0 or digital 1 by the specified deadline.

This turns out to be an unsolvable problem! For no finite values of t_E and t_D can we build a synchronizer that's guaranteed to meet this specification even when using components that are 100% reliable.

Unsolvable? That can't be true...

Let's just use a D Register:



We're lulled by the digital abstraction into assuming that Q must be either 1 or 0. But let's look at the input latch in the flip flop when IN and CLK change at about the same time...

DECISION TIME is T_{PD} of register.

ALLOWABLE ERROR is $\max(t_{SETUP}, t_{HOLD})$

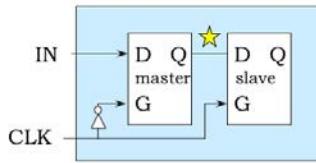
Our logic:

T_{PD} after T_C , we'll have

$$Q=1 \text{ iff } t_{IN} + t_{SETUP} < t_C$$

$$Q=0 \text{ iff } t_C + t_{HOLD} < t_{IN}$$

$Q=0$ or 1 otherwise.

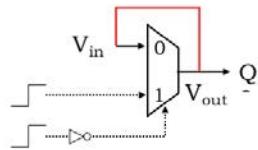


But can't we just use a D register to solve the problem? We'll connect IN to the register's data input and connect CLK to the register's clock input. We'll set the decision time t_D to the propagation delay of the register and the allowable error interval to the larger of the register's setup and hold times.

Our theory is that if the rising edge of IN occurs at least t_{SETUP} before the rising edge of CLK, the register is guaranteed to output a 1. And if IN transitions more than t_{HOLD} after the rising edge of CLK, the register is guaranteed to output a 0. So far, so good. If IN transitions during the setup and hold times with respect to the rising edge on CLK, we know we've violated the dynamic discipline and we can't tell whether the register will store a 0 or a 1. But in this case, our specification lets us produce either answer, so we're good to go, right?

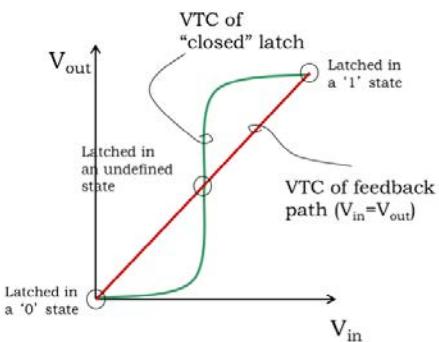
Sadly, we're not good to go. We're lulled by the digital abstraction into assuming that even if we violate the dynamic discipline that Q must be either 1 or 0 after the propagation delay. But that isn't a valid assumption as we'll see when we look more carefully at the operation of the register's master latch when B and C change at about the same time.

The Mysterious Metastable State



Recall that the latch output is the solution to two simultaneous constraints:

1. The VTC of path thru MUX; and
2. $V_{in} = V_{out}$



In addition to our expected stable solutions, we find an unstable equilibrium in the forbidden zone called the "Metastable State"

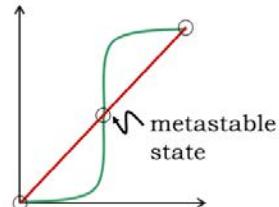
Recall that the master latch is really just a lenient MUX that can be configured as a bi-stable storage element using a positive feedback loop. When the latch is in memory mode, it's essentially a two-gate cyclic circuit whose behavior has two constraints: the voltage transfer characteristic of the two-gate circuit, shown in green on the graph, and that V_{IN} equals t_{OUT} , a constraint that's shown in red on the graph.

These two curves intersect at three points. Our concern is the middle point of intersection. If IN and CLK change at the same time, the voltage on Q may be in transition at the time the MUX closes and enables the positive feedback loop. So the initial voltage in the feedback loop may happen to be at or very near the voltage of the middle intersection point.

When Q is at the metastable voltage, the storage loop is in an unstable equilibrium called the metastable state. In theory the system could balance at this point forever, but a small change in the voltages in the loop will move the system away from the metastable equilibrium point and set it irrevocably in motion towards the stable equilibrium points. Here's the issue we face: we can't bound the amount of time the system will spend in the metastable state.

Metastable State: Properties

1. It corresponds to an *invalid* logic level.
2. It's an *unstable* equilibrium; a small perturbation will cause it to move toward a stable 0 or 1.
3. It will settle to a valid 0 or 1... eventually.
4. BUT – depending on how close it is to the $V_{in}=V_{out}$ “fixed point” of the device – it may take arbitrarily long to settle out.
5. EVERY bistable system exhibits at least one metastable state!



If metastable at t_0 :

- $p(\text{metastable at } t_0+T) > 0$ for finite T
- $p(\text{metastable at } t_0+T)$ decreases exponentially with increasing T

Here's what we know about the metastable state.

It's in the forbidden zone of the digital signaling specifications and so corresponds to an invalid logic level. Violating the dynamic discipline means that our register is no longer guaranteed to produce a digital output in bounded time.

A persistent invalid logic level can wreak both logical and electrical havoc in our sequential logic circuit. Since combinational logic gates with invalid inputs have unpredictable outputs, an invalid signal may corrupt the state and output values in our sequential system.

At the electrical level, if an input to a CMOS gate is at the metastable voltage, both PFET and NFET switches controlled by that input would be conducting, so we'd have a path between V_{DD} and GND, causing a spike in the system's power dissipation.

It's an unstable equilibrium and will eventually be resolved by a transition to one of the two stable equilibrium points. You can see from the graph that the metastable voltage is in the high-gain region of the VTC, so a small change in V_{IN} results in a large change in t_{OUT} , and once away from the metastable point the loop voltage will move towards 0 or V_{DD} .

The time it takes for the system to evolve to a stable equilibrium is related to how close Q's voltage was to the metastable point when the positive feedback loop was enabled. The closer Q's initial voltage is to the metastable voltage, the longer it will take for the system to resolve the metastability. But since there's no lower bound on how close Q is to the metastable voltage, there's no upper bound on the time it will take for resolution. In other words, if you specify a bound, e.g., t_D , on the time available for resolution, there's a range of initial Q voltages that won't be resolved within that time.

If the system goes metastable at some point in time, then there's a non-zero probability that the system will still be metastable after some interval T , for any finite choice of T .

The good news is that the probability of being metastable at the end of the interval decreases exponentially with increasing T .

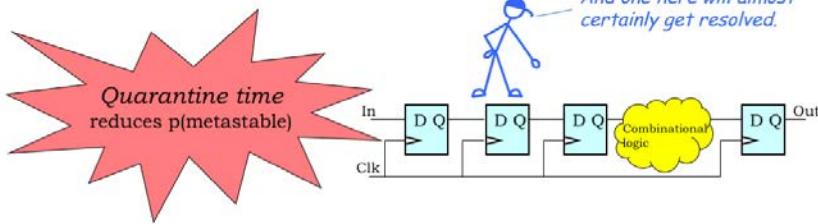
Note that every bistable system has at least one metastable state. So metastability is the price we pay for building storage elements from positive feedback loops.

If you'd like to read a more thorough discussion of synchronizers and related problems and learn about the mathematics behind the exponential probabilities, please see Chapter 10 of the Course Notes.

Solution: Delay Increases Reliability

Extra registers between the asynchronous input and your logic are the best insurance against metastable states.

For higher clock rates, consider adding additional registers.



Our approach to dealing with asynchronous inputs is to put the potentially metastable value coming out of our D-register synchronizer into *quarantine* by adding a second register hooked to the output of the first register.

If a transition on the input violates the dynamic discipline and causes the first register to go metastable, it's not immediately an issue since the metastable value is stopped from entering the system by the second register. In fact, during the first half of the clock cycle, the master latch in the second register is closed, so the metastable value is being completely ignored.

It's only at the next clock edge, an entire clock period later, that the second D register will need a valid and stable input. There's still some probability that the first register will be metastable after an entire clock period, but we can make that probability as low as we wish by choosing a sufficiently long clock period. In other words, the output of the second register, which provides the signal used by the internal combinational logic, will be stable and valid with a probability of our choosing. Validity is not 100% guaranteed, but the failure times are measured in years or decades, so it's not an issue in practice. Without the second register, the system might see a metastability failure every handful of hours — the exact failure rate depends on the transition frequencies and gains in the circuit.

What happens if our clock period is short but we want a long quarantine time? We can use multiple quarantine registers in series — it's the total delay between when the first register goes metastable and when the synchronized input is used by the internal logic that determines the failure probability.

The bottom line: we can use synchronizing registers to quarantine potentially metastable signals for some period of time. Since the probability of still being metastable decreases exponentially with the quarantine time, we can reduce the failure probability to any desired level. Not a 100% guaranteed, but close enough that metastability is not a practical issue if we use our quarantine strategy.

7: Performance Measures

1. Forget Circuits... Let's solve a Real Problem
2. One Load At A Time
3. Doing N Loads of Laundry
4. Doing N Loads... The 6.004 Way
5. Performance Measures
6. Okay, Back to Circuits...
7. Pipelined Circuits
8. Pipeline Diagrams
9. Pipeline Conventions
10. Ill-formed Pipeline
11. A Pipelining Methodology
12. Pipeline Example
13. Pipelined Components
14. How Do 6.004 Students Do Laundry?
15. Back to Our Bottleneck...
16. Circuit Interleaving I
17. Circuit Interleaving II
18. Circuit Interleaving III
19. Combine Techniques
20. And Add A Little Parallelism...
21. Control Structure Alternatives
22. Self-timed Example
23. Self-timed Example
24. Control Structure Taxonomy
25. Summary

In this lecture our goal is to introduce some metrics for measuring the performance of a circuit and then investigate ways to improve that performance. We'll start by putting aside circuits for a moment and look at an everyday example that will help us understand the proposed performance metrics.

Forget circuits... Let's Solve a Real Problem

INPUT:
dirty laundry



OUTPUT:
6 more weeks



Device: Washer
Function: Fill, Agitate, Spin
 $\text{Washer}_{\text{PD}} = 30 \text{ mins}$



Device: Dryer
Function: Heat, Spin
 $\text{Dryer}_{\text{PD}} = 60 \text{ mins}$

Laundry is a processing task we all have to face at some point! The input to our laundry “system” is some number of loads of dirty laundry and the output is the same loads, but washed, dried, and folded. There two system components: a washer that washes a load of laundry in 30 minutes, and a dryer that dries a load in 60 minutes. You may be used to laundry system components with different propagation delays, but let’s go with these delays for our example.

Our laundry follows a simple path through the system: each load is first washed in the washer and afterwards moved to the dryer for drying. There can, of course, be delays between the steps of loading the washer, or moving wet, washed loads to the dryer, or in taking dried loads out of the dryer. Let’s assume we move the laundry through the system as fast as possible, moving loads to the next processing step as soon as we can.

One Load At a Time

Everyone knows that the real reason that we put off doing laundry so long is not because we procrastinate, are lazy, or even have better things to do.

The fact is, doing one load at a time is not smart.

Step 1:



Step 2:



$$\text{Total}_{\text{PD}} = \text{Washer}_{\text{PD}} + \text{Dryer}_{\text{PD}}$$


$$= \underline{\hspace{2cm} \textbf{90} \hspace{2cm}} \text{ mins}$$

Most of us wait to do laundry until we've accumulated several loads. That turns out to be a good strategy! Let's see why...

To process a single load of laundry, we first run it through the washer, which takes 30 minutes. Then we run it through the dryer, which takes 60 minutes. So the total amount of time from system input to system output is 90 minutes. If this were a combinational logic circuit, we'd say the circuit's propagation delay is 90 minutes from valid inputs to valid outputs.

Okay, that's the performance analysis for a single load of laundry. Now let's think about doing N loads of laundry.

Doing N Loads of Laundry

Here's how they do laundry at Harvard, the "combinational" way.

Of course, this is just an urban legend. No one at Harvard actually *does* laundry. The butlers all arrive on Wednesday morning, pick up the dirty laundry and return it all pressed and starched in time for afternoon tea.

Step 1:



Step 2:



Step 3:



Step 4:

...

$$\begin{aligned} \text{Total}_{\text{PD}} &= N * (\text{Washer}_{\text{PD}} + \text{Dryer}_{\text{PD}}) \\ &= \underline{\underline{N * 90}} \text{ mins} \end{aligned}$$



Here at MIT we like to make gentle fun of our colleagues at the prestigious institution just up the river from us. So here's how we imagine they do N loads of laundry at Harvard. They follow the combinational recipe of supplying new system inputs after the system generates the correct output from the previous set of inputs. So in step 1 the first load is washed and in step 2, the first load is dried, taking a total of 90 minutes. Once those steps complete, Harvard students move on to step 3, starting the processing of the second load of laundry. And so on...

The total time for the system to process N laundry loads is just N times the time it takes to process a single load. So the total time is N*90 minutes.

Of course, we're being silly here! Harvard students don't actually do laundry. Mummy sends the family butler over on Wednesday mornings to collect the dirty loads and return them starched and pressed in time for afternoon tea.

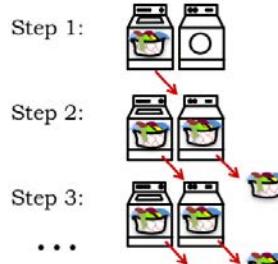
But I hope you're seeing the analogy we're making between the Harvard approach to laundry and combinational circuits. We can all see that the washer is sitting idle while the dryer is running and that inefficiency has a cost in terms of the rate at which N load of laundry can move through the system.

Doing N Loads... The 6.004 Way

6.004 students
“pipeline” the laundry
process.

That's why we wait!

Actually, it's more like $N*60 + 30$ if we account for the startup transient correctly. When doing pipeline analysis, we're mostly interested in the “steady state” where we assume we have an infinite supply of inputs.



$$\begin{aligned} \text{Total}_{\text{PD}} &= N * \text{Max}(\text{Washer}_{\text{PD}}, \text{Dryer}_{\text{PD}}) \\ &= \underline{\text{N*60}} \text{ mins} \end{aligned}$$

As engineering students here in 6.004, we see that it makes sense to overlap washing and drying. So in step 1 we wash the first load. And in step 2, we dry the first load as before, but, in addition, we start washing the second load of laundry. We have to allocate 60 minutes for step 2 in order to give the dryer time to finish. There's a slight inefficiency in that the washer finishes its work early, but with only one dryer, it's the dryer that determines how quickly laundry moves through the system.

Systems that overlap the processing of a sequence of inputs are called pipelined systems and each of the processing steps is called a stage of the pipeline. The rate at which inputs move through the pipeline is determined by the slowest pipeline stage. Our laundry system is a 2-stage pipeline with a 60-minute processing time for each stage.

We repeat the overlapped wash/dry step until all N loads of laundry have been processed [CLICK]. We're starting a new washer load every 60 minutes and getting a new load of dried laundry from the dryer every 60 minutes. In other words, the effective processing rate of our overlapped laundry system is one load every 60 minutes. So once the process is underway N loads of laundry takes $N*60$ minutes. And a particular load of laundry, which requires two stages of processing time, takes 120 minutes.

The timing for the first load of laundry is a little different since the timing of Step 1 can be shorter with no dryer to wait for. But in the performance analysis of pipelined systems, we're interested in the steady state where we're assuming that we have an infinite supply of inputs.

Performance Measures

Latency:

The delay from when an input is established until the output associated with that input becomes valid.

$$\begin{array}{l} \text{Harvard Laundry} = \frac{90}{\text{mins}} \\ \text{6.004 Laundry} = \frac{120}{\text{mins}} \end{array}$$

Assuming that the wash is started as soon as possible and waits (wet) in the washer until dryer is available.

Throughput:

The *rate* at which inputs or outputs are processed.

$$\begin{array}{l} \text{Harvard Laundry} = \frac{1/90}{\text{outputs/min}} \\ \text{6.004 Laundry} = \frac{1/60}{\text{outputs/min}} \end{array}$$

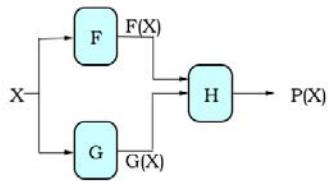
We see that there are two interesting performance metrics. The first is the latency of the system, the time it takes for the system to process a particular input. In the Harvard laundry system, it takes 90 minutes to wash and dry a load. In the 6.004 laundry, it takes 120 minutes to wash and dry a load, assuming that it's not the first load.

The second performance measure is throughput, the rate at which the system produces outputs. In many systems, we get one set of outputs for each set of inputs, and in such systems, the throughput also tells us the rate at which inputs are consumed. In the Harvard laundry system, the throughput is 1 load of laundry every 90 minutes. In the 6.004 laundry, the throughput is 1 load of laundry every 60 minutes.

The Harvard laundry has lower latency, the 6.004 laundry has better throughput. Which is the better system? That depends on your goals! If you need to wash 100 loads of laundry, you'd prefer to use the system with higher throughput. If, on the other hand, you want clean underwear for your date in 90 minutes, you're much more concerned about the latency.

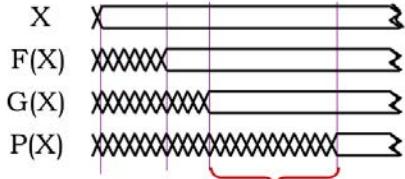
The laundry example also illustrates a common tradeoff between latency and throughput. If we increase throughput by using pipelined processing, the latency usually increases since all pipeline stages must operate in lock-step and the rate of processing is thus determined by the slowest stage.

Okay, Back To Circuits...



For combinational logic:
 $\text{latency} = t_{\text{PD}}$,
 $\text{throughput} = 1/t_{\text{PD}}$.

We can't get the answer faster,
 but are we making effective use
 of our hardware at all times?



F & G are “idle”, just holding their outputs stable while H performs its computation

Okay, now let's apply all this analysis to improving the performance of our circuits. The latency of a combinational logic circuit is simply its propagation delay t_{PD} . And the throughput is just $1/t_{\text{PD}}$ since we start processing the next input only after finishing the computation on the current input.

Consider a combinational system with three components: F, G, and H, where F and G work in parallel to produce the inputs to H. Using this timing diagram we can follow the processing of a particular input value X. Sometime after X is valid and stable, the F and G modules produce their outputs F(X) and G(X).

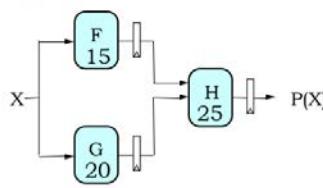
Now that the inputs to H are valid and stable, the H module will produce the system output P(X) after a delay set by the propagation delay of H. The total elapsed time from valid input to valid output is determined by the propagation delays of the component modules. Assuming we use those modules as-is, we can't make any improvements on this latency.

But what about the system's throughput? Observe that after producing their outputs, the F and G modules are sitting idle, just holding their outputs stable while H performs its computation. Can we figure out a way for F and G to get started processing the next input while still letting H do its job on the first input? In other words, can we divide the processing of the combinational circuit into two stages where the first stage computes F(X) and G(X), and the second stage computes H(X)? If we can, then we can increase the throughput of the system.



Pipelined Circuits

use registers to hold H's input stable!



Now F & G can be working on input X_{i+1} while H is performing its computation on X_i . We've created a 2-stage **pipeline**: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal zero-delay registers:

latency throughput

unpipelined	45	1/45
2-stage pipeline	50	1/25

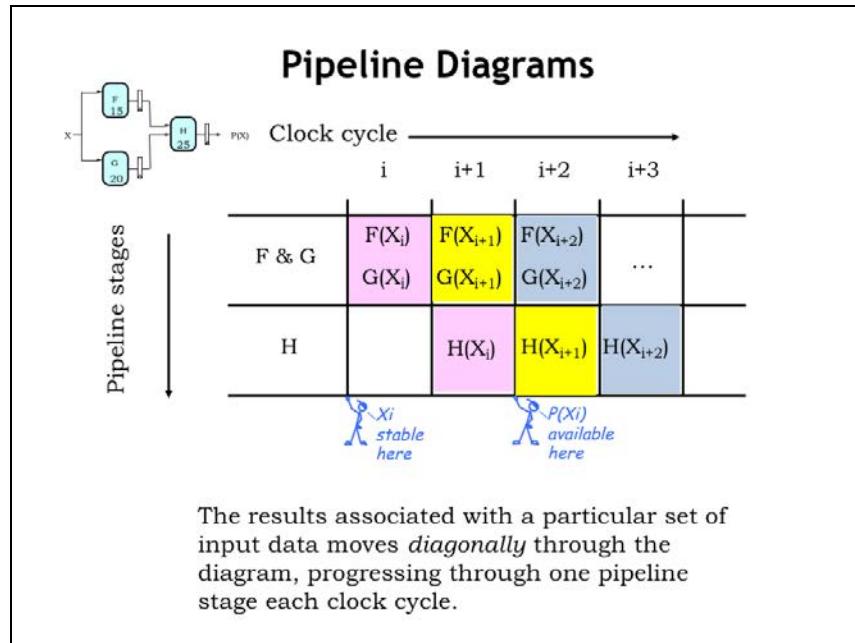
worse *better*

Mr. Blue's inspiration is to use registers to hold the values $F(X)$ and $G(X)$ for use by H, while the F and G modules start working on the next input value. To make our timing analysis a little easier, we'll assume that our pipelining registers have a zero propagation delay and setup time.

The appropriate clock period for this sequential circuit is determined by the propagation delay of the slowest processing stage. In this example, the stage with F and G needs a clock period of at least 20 ns to work correctly. And the stage with H needs a clock period of 25 ns to work correctly. So the second stage is the slowest and sets the system clock period at 25 ns.

This will be our general plan for increasing the throughput of combinational logic: we'll use registers to divide the processing into a sequence of stages, where the registers capture the outputs from one processing stage and hold them as inputs for the next processing stage. A particular input will progress through the system at the rate of one stage per clock cycle.

In this example, there are two stages in the processing pipeline and the clock period is 25 ns, so the latency of the pipelined system is 50 ns, *i.e.*, the number of stages times the system's clock period. The latency of the pipeline system is a little longer than the latency of the unpipelined system. However, the pipeline system produces 1 output every clock period, or 25 ns. The pipeline system has considerably better throughput at the cost of a small increase in latency.



Pipeline diagrams help us visualize the operation of a pipelined system. The rows of the pipeline diagram represent the pipeline stages and the columns are successive clock cycles.

At the beginning of clock cycle i the input X_i becomes stable and valid. Then during clock cycle i the F and G modules process that input and at the end of the cycle the results $F(X_i)$ and $G(X_i)$ are captured by the pipeline registers between the first and second stages.

Then in cycle $i + 1$, H uses the captured values do its share of the processing of X_i . And, meanwhile, the F and G modules are working on X_{i+1} . You can see that the processing for a particular input value moves diagonally through the diagram, one pipeline stage per clock cycle.

At the end of cycle $i+1$, the output of H is captured by the final pipeline register and is available for use during cycle $i+2$. The total time elapsed between the arrival of an input and the availability of the output is two cycles.

The processing continues cycle after cycle, producing a new output every clock cycle.

Using the pipeline diagram we can track how a particular input progresses through the system or see what all the stages are doing in any particular cycle.

Pipeline Conventions

DEFINITION:

A well-formed **K-Stage Pipeline** ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.

COMPOSITION CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUT* (not on its input).

ALWAYS:

The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP} .

The LATENCY of a K-pipeline is K times the period of the system's clock.

The THROUHPUT of a K-pipeline is the frequency of the clock.

We'll define a K-stage pipeline (or K-pipeline for short) as an acyclic circuit having exactly K registers on every path from input to output. An unpipelined combinational circuit is thus a 0-stage pipeline.

To make it easy to build larger pipelined systems out of pipelined components, we'll adopt the convention that every pipeline stage, and hence every K-stage pipeline, has a register on its OUTPUT.

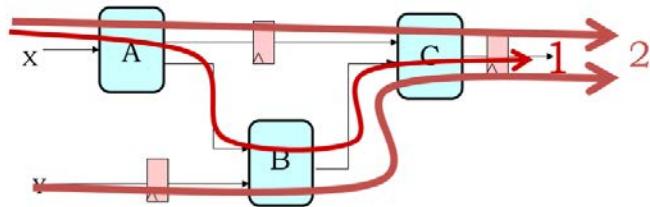
We'll use the techniques we learned for analyzing the timing of sequential circuits to ensure the clock signal common to all the pipeline registers has a period sufficient to ensure correct operation of each stage. So for every register-to-register and input-to-register path, we need to compute the sum of the propagation delay of the input register, plus the propagation delay of the combinational logic, plus the setup time of the output register. Then we'll choose the system's clock period to be greater than or equal to the largest such sum.

With the correct clock period and exactly K-registers along each path from system input to system output, we are guaranteed that the K-pipeline will compute the same outputs as the original unpipelined combinational circuit.

The latency of a K-pipeline is K times the period of the system's clock. And the throughput of a K-pipeline is the frequency of the system's clock, *i.e.*, 1 over the clock period.

III-formed Pipelines

Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

ANS: none

Problem:

Successive inputs get mixed: e.g., B(A(X_{i+1}), Y_j). This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

Here's a failed attempt at pipelining a circuit. For what value of K is the circuit a K-pipeline? Well, let's count the number of registers along each path from system inputs to system outputs.

The top path through the A and C components has 2 registers. As does the bottom path through the B and C components. But the middle path through all three components has only 1 register. Oops, this not a well-formed K-pipeline.

Why do we care? We care because this pipelined circuit does not compute the same answer as the original unpipelined circuit. The problem is that successive generations of inputs get mixed together during processing. For example, during cycle i+1, the B module is computing with the current value of the X input but the previous value of the Y input.

This can't happen with a well-formed K-pipeline. So we need to develop a technique for pipelining a circuit that guarantees the result will be well-formed.

A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

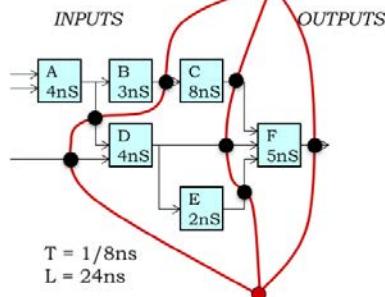
Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



Here's our strategy that will ensure if we add a pipeline register along one path from system inputs to system outputs, we will add pipeline registers along every path.

Step 1 is to draw a contour line that crosses every output in the circuit and mark its endpoints as the terminal points for all the other contours we'll add.

During Step 2 continue to draw contour lines between the two terminal points across the signal connections between modules. Make sure that every signal connection crosses the new contour line in the same direction. This means that system inputs will be one side of the contour and system outputs will be on the other side. These contours demarcate pipeline stages.

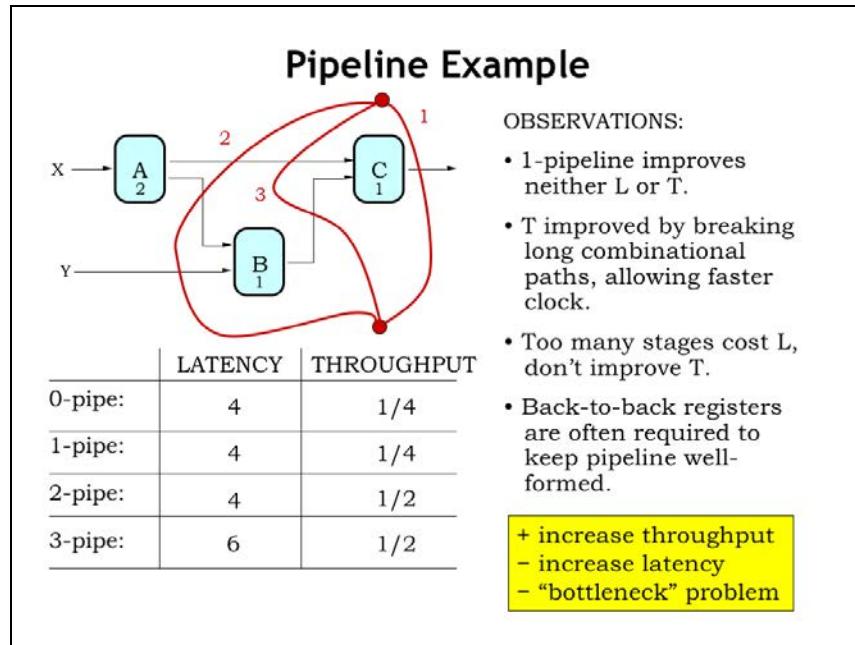
Place a pipeline register wherever a signal connection intersects the pipelining contours. Here we've marked the location of pipeline registers with large black dots.

By drawing the contours from terminal point to terminal point we guarantee that we cross every input-output path, thus ensuring our pipeline will be well-formed.

Now we can compute the system's clock period by looking for the pipeline stage with the longest register-to-register or input-to-register propagation delay. With these contours and assuming ideal zero-delay pipeline registers, the system clock must have a period of 8 ns to accommodate the operation of the C module. This gives a system throughput of 1 output every 8 ns. Since we drew 3 contours, this is a 3-pipeline and the system latency is 3 times 8 ns or 24 ns total.

Our usual goal in pipelining a circuit is to achieve maximum throughput using the fewest possible registers. So our strategy is to find the slowest system component (in our example, the C component) and place pipeline registers on its inputs and outputs. So we drew contours that pass on either side of the C module. This sets the clock period at 8 ns, so we position the contours so that longest path between any two pipeline registers is at most 8.

There are often several choices for how to draw a contour while maintaining the same throughput and latency. For example, we could have included the E module in the same pipeline stage as the F module.



Okay, let's review our pipelining strategy.

First we draw a contour across all the outputs. This creates a 1-pipeline, which, as you can see, will always have the same throughput and latency as the original combinational circuit.

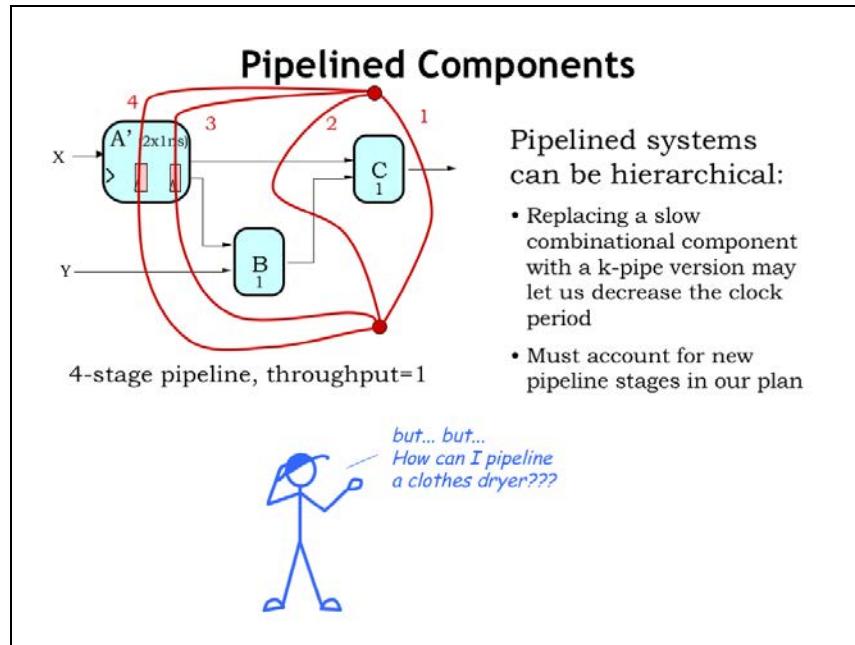
Then we draw our next contour, trying to isolate the slowest component in the system. This creates a 2-pipeline with a clock period of 2 and hence a throughput of $1/2$, or double that of the 1-pipeline.

We can add additional contours, but note that the 2-pipeline had the smallest possible clock period, so after that additional contours add stages and hence increase the system's latency without increasing its throughput. Not illegal, just not a worthwhile investment in hardware.

Note that the signal connection between the A and C module now has two back-to-back pipelining registers. Nothing wrong with that; it often happens when we pipeline a circuit where the input-output paths are of different lengths.

So our pipelining strategy will be to pipeline implementations with increased throughput, usually at the cost of increased latency. Sometimes we get lucky and the delays of each pipeline stage are perfectly balanced, in which case the latency will not increase. Note that a pipelined circuit will NEVER have a smaller latency than the unpipelined circuit.

Notice that once we've isolated the slowest component, we can't increase the throughput any further. How do we continue to improve the performance of circuits in light of these performance bottlenecks?



One solution is to use pipelined components if they're available! Suppose we're able to replace the original A component with a 2-stage pipelined version A-prime.

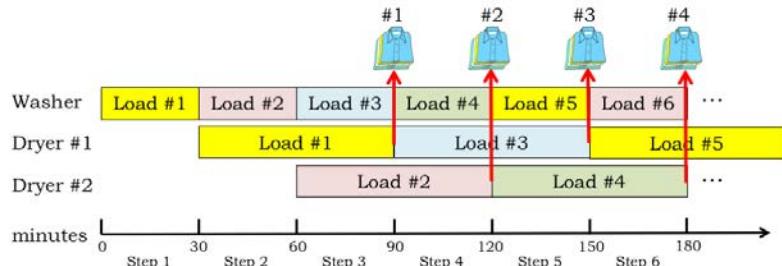
We can redraw our pipelining contours, making sure we account for the internal pipeline registers in the A-prime component. This means that 2 of our contours have to pass through the A-prime component, guaranteeing that we'll add pipeline registers elsewhere in the system that will account for the two-cycle delay introduced by A-prime.

Now the maximum propagation delay in any stage is 1 ns, doubling the throughput from 1/2 to 1/1. This is a 4-pipeline so the latency will be 4 ns.

This is great! But what can we do if our bottleneck component doesn't have a pipelined substitute. We'll tackle that question in the next section.

How Do 6.004 Students Do Laundry?

They work around the bottleneck. First, they find a laundromat with two dryers for every washer. Then they use dryer #1 for odd-numbered wash loads and dryer #2 for even-numbered wash loads.



$$\text{Throughput} = \underline{1/30} \text{ loads/min}, \text{Latency} = \underline{90} \text{ mins/load}$$

6.004 students work around the dryer bottleneck by finding a laundromat that has two dryers for every washer. Looking at the timeline you can see the plan, which is divided into 30-minute steps. The washer is in use every step, producing a newly-washed load every 30 minutes. Dryer usage is interleaved, where Dryer #1 is used to dry the odd-numbered loads and Dryer #2 is used to dry the even-numbered loads. Once started, a dryer runs for a duration of two steps, a total of 60 minutes. Since the dryers run on a staggered schedule, the system as a whole produces a load of clean, dry laundry every 30 minutes.

The steady-state throughput is 1 load of laundry every 30 minutes and the latency for a particular load of laundry is 90 minutes.

And now here's the take-home message from this example. Consider the operation of the two-dryer system. Even though the component dryers themselves aren't pipelined, the two-dryer interleaving system is acting like a 2-stage pipeline with a clock period of 30 minutes and a latency of 60 minutes. In other words, by interleaving the operation of 2 unpipelined components we can achieve the effect of a 2-stage pipeline.

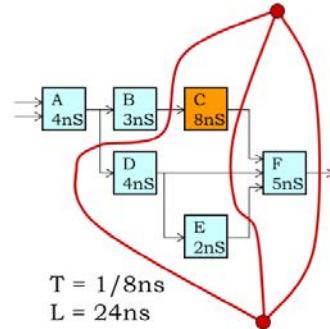
Back To Our Bottleneck...

Recall our earlier example...

- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to $1/8$ ns.

We could improve throughput by

- Finding a pipelined version of C;
- OR ...
- *interleaving* multiple copies of C!



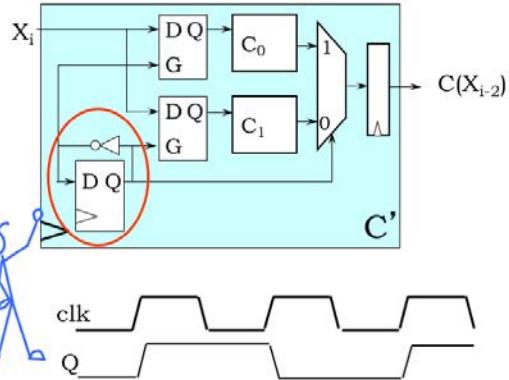
Returning to the example of the previous section, we couldn't improve the throughput of our pipelined system past $1/8$ ns since the minimum clock period was set by the 8 ns latency of the C module.

To improve the throughput further we either need to find a pipelined version of the C component or use an interleaving strategy to achieve the effect of a 2-stage pipeline using two instances of the unpipelined C component. Let's try that...

Circuit Interleaving

We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.

This is a simple 2-state FSM that alternates between 0 and 1 on each clock

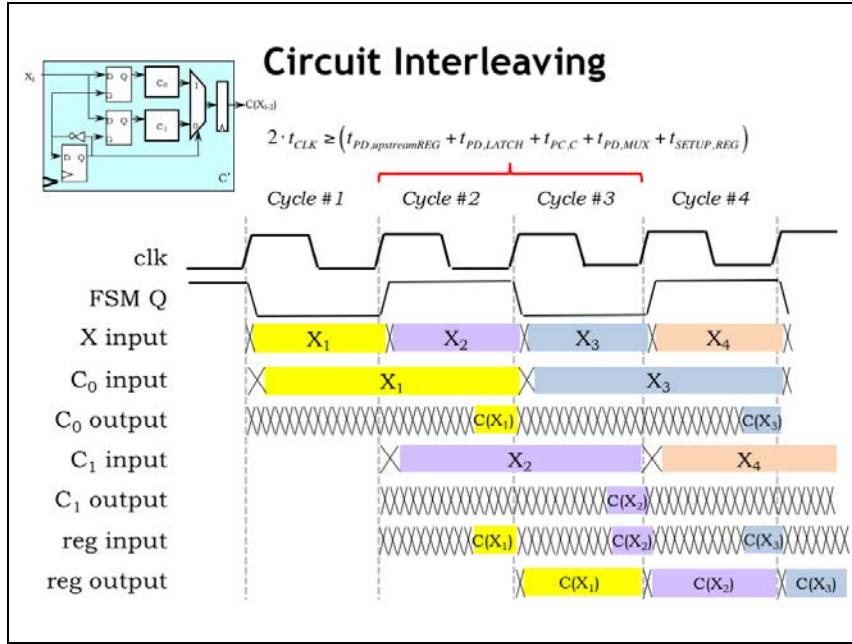


Here's a circuit for a general-purpose two-way interleaver, using, in this case, two copies of the unpipelined C component, C_0 and C_1 .

The input for each C component comes from a D-latch, which has the job of capturing and holding the input value. There's also a multiplexer to select which C-component output will be captured by the output register.

In the lower left-hand corner of the circuit is a very simple 2-state FSM with one state bit. The next-state logic is a single inverter, which causes the state to alternate between 0 and 1 on successive clock cycles. This timing diagram shows how the state bit changes right after each rising clock edge.

To help us understand the circuit, we'll look at some signal waveforms to illustrate its operation.



To start, here are the waveforms for the CLK signal and our FSM state bit from the previous slide.

A new X input arrives from the previous stage just after the rising edge of the clock.

Next, let's follow the operation of the C_0 component. Its input latch is open when $FSM\ Q$ is low, so the newly arriving X_1 input passes through the latch and C_0 can begin its computation, producing its result at the end of clock cycle #2. Note that the C_0 input latch closes at the beginning of the second clock cycle, holding the X_1 input value stable even though the X input is starting to change. The effect is that C_0 has a valid and stable input for the better part of 2 clock cycles giving it enough time to compute its result.

The C_1 waveforms are similar, just shifted by one clock cycle. C_1 's input latch is open when $FSM\ Q$ is high, so the newly arriving X_2 input passes through the latch and C_1 can begin its computation, producing its result at the end of clock cycle #3.

Now let's check the output of the multiplexer. When $FSM\ Q$ is high, it selects the value from C_0 and when $FSM\ Q$ is low, it selects the value from C_1 . We can see that happening in the waveform shown.

Finally, at the rising edge of the clock, the output register captures the value on its input and holds it stable for the remainder of the clock cycle.

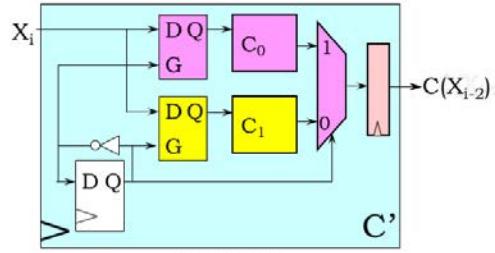
The behavior of the interleaving circuit is like a 2-stage pipeline: the input value arriving in cycle i is processed over two clock cycles and the result output becomes available on cycle i+2.

What about the clock period for the interleaving system? Well, there is some time lost to the propagation delays of the upstream pipeline register that supplies the X input, the internal latches and multiplexer, and the setup time of the output register. So the clock cycle has to be just a little bit longer than half the propagation delay of the C module.

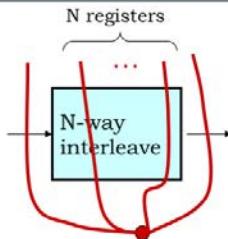
Circuit Interleaving

2-Clock Martinizing
“In by t_i , out by t_{i+2} ”

Throughput = 1/clock
Latency = 2 clocks



N-way interleaving
is equivalent to
N pipeline Stages...



We can treat the interleaving circuit as a 2-stage pipeline, consuming an input value every clock cycle and producing a result two cycles later.

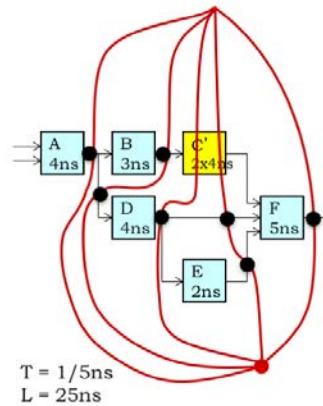
When incorporating an N-way interleaved component in our pipeline diagrams, we treat it just like a N-stage pipeline. So N of our pipelining contours have to pass through the component.

Combine Techniques

We can combine interleaving and pipelining. Here, C' interleaves two C elements and has an effective t_{CLK} of 4 ns and a latency of 8 ns.

Since C' behaves as a 2-stage pipeline, two of our pipelining contours must pass through the C' component.

By combining interleaving with pipelining we move the bottleneck from the C element to the F element.



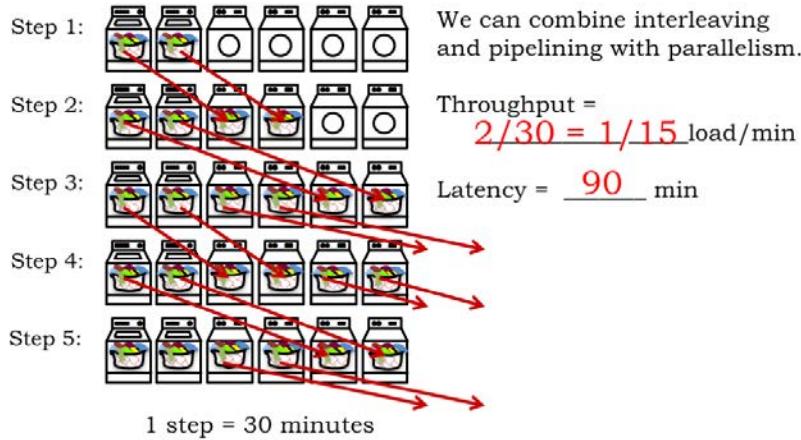
Here we've replaced the slow unpipelined C component with a 2-way interleaved C-prime component.

We can follow our process for drawing pipelining contours. First we draw a contour across all the outputs. Then we add contours, ensuring that two of them pass through the C-prime component. Then we add pipeline registers at the intersections of the contours with the signal connections. We see that the contours passing through C-prime have caused extra pipeline registers to be added on the other inputs to the F module, accommodating the 2-cycle delay through C-prime.

Somewhat optimistically we've specified the C-prime minimum t_{CLK} to be 4 ns, so that means that the slow component which determines the system's clock period is now the F module, with a propagation delay of 5 ns.

So the throughput of our new pipelined circuit is 1 output every 5 ns, and with 5 contours, it's a 5-pipeline so the latency is 5 times the clock period or 25 ns.

And Add A Little Parallelism...



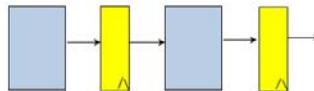
By running pipelined systems in parallel we can continue to increase the throughput. Here we show a laundry with 2 washers and 4 dryers, essentially just two copies of the 1-washer, 2-dryer system shown earlier. The operation is as described before, except that at each step the system produces and consumes two loads of laundry.

So the throughput is 2 loads every 30 minutes for an effective rate of 1 load every 15 minutes. The latency for a load hasn't changed; it's still 90 minutes per load.

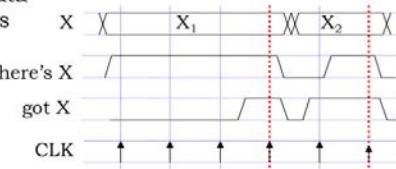
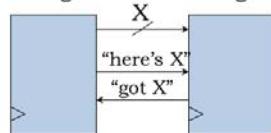
We've seen that even with slow components we can use interleaving and parallelism to continue to increase throughput. Is there an upper bound on the throughput we can achieve? Yes! The timing overhead of the pipeline registers and interleaving components will set a lower bound on the achievable clock period, thus setting an upper bound on the achievable throughput. Sorry, no infinite speed-up is possible in the real world.

Control Structure Alternatives

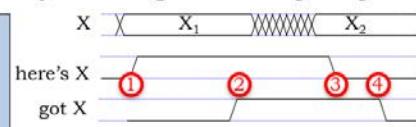
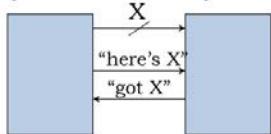
Synchronous, globally-timed:



Synchronous, locally-timed:
Local FSMs control flow of data
using "handshake" signals



Asynchronous, locally-timed system using *transition signaling*:

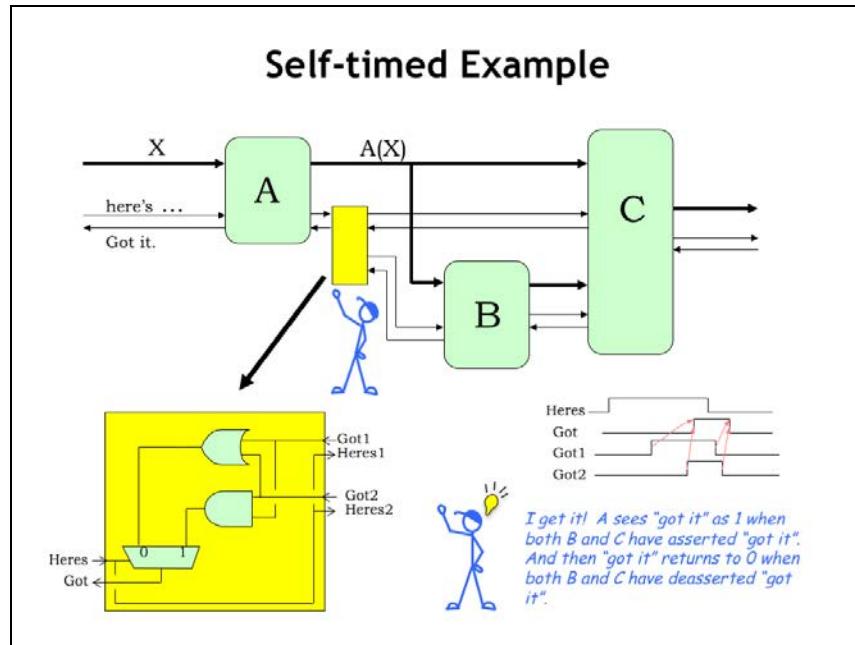


We've been designing our processing pipelines to have all the stages operate in lock step, choosing the clock period to accommodate the worst-case processing time over all the stages. This is what we'd call a synchronous, globally timed system.

But what if there are data dependencies in the processing time, *i.e.*, if for some data inputs a particular processing stage might be able to produce its output in a shorter time? Can we design a system that could take advantage of that opportunity to increase throughput?

One alternative is to continue to use a single system clock, but for each stage to signal when it's ready for a new input and when it has a new output ready for the next stage. It's fun to design a simple 2-signal handshake protocol to reliably transfer data from one stage to the next. The upstream stage produces a signal called HERE-IS-X to indicate that it has new data for the downstream stage. And the downstream stage produces a signal called GOT-X to indicate when it is willing to consume data. It's a synchronous system so the signal values are only examined on the rising edge of the clock.

The handshake protocol works as follows: the upstream stage asserts HERE-IS-X if it will have a new output value available at the next rising edge of the clock. The downstream stage asserts GOT-X if it will grab the next output at the rising edge of the clock. Both stages look at the signals on the rising edge of the clock to decide what to do next. If both stages see that HERE-IS-X and GOT-X are asserted at the same clock edge, the handshake is complete and the data transfer happens at that clock edge. Either stage can delay a transfer if they are still working on producing the next output or consuming the previous input.



It's possible, although considerably more difficult, to build a clock-free asynchronous self-timed system that uses a similar handshake protocol. The handshake involves four phases. In phase 1, when the upstream stage has a new output and GOT-X is deasserted, it asserts its HERE-IS-X signal and then waits to see the downstream stage's reply on the GOT-X signal. In phase 2, the downstream stage, seeing that HERE-IS-X is asserted, asserts GOT-X when it has consumed the available input. In phase 3, the downstream stage waits to see the HERE-IS-X go low, indicating that the upstream stage has successfully received the GOT-X signal. In phase 4, once HERE-IS-X is deasserted, the downstream stage deasserts GOT-X and the transfer handshake is ready to begin again. Note that the upstream stage waits until it sees the GOT-X deasserted before starting the next handshake.

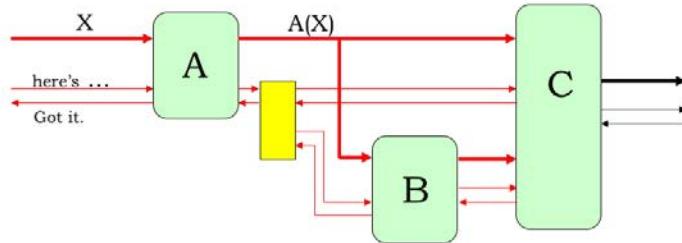
The timing of the system is based on the transitions of the handshake signals, which can happen at any time the conditions required by the protocol are satisfied. No need for a global clock here!

It's fun to think about how this self-timed protocol might work when there are multiple downstream modules, each with their own internal timing.

In this example, A's output is consumed by both the B and C stages. We need a special circuit, shown as a yellow box in the diagram, to combine the GOT-X signals from the B and C stages and produce a summary signal for the A stage.

Let's take a quick look at the timing diagram shown here. After A has asserted HERE-IS-X, the circuit in the yellow box waits until both the B AND the C stage have asserted their GOT-X signals before asserting GOT-X to the A stage. At this point the A stage deasserts HERE-IS-X, then the yellow box waits until both the B and C stages have deasserted their GOT-X signals, before deasserting GOT-X to the A stage.

Self-timed Example



Elegant, timing-independent design:

- Each component specifies its own time constraints
- Local adaptation to special cases (eg, multiplication by 0)
- Module performance improvements automatically exploited
- Can be made asynchronous (no clock at all!) or synchronous

Let's watch the system in action! When a signal is asserted we'll show it in red, otherwise it's shown in black.

A new value for the A stage arrives on A's data input and the module supplying the value then asserts its HERE-IS-X signal to let A know it has a new input.

At some point later, A signals GOT-X back upstream to indicate that it has consumed the value, then the upstream stage deasserts HERE-IS-X, followed by A deasserting its GOT-X output. This completes the transfer of the data to the A stage.

When A is ready to send a new output to the B and C stages, it checks that its GOT-X input is deasserted (which it is), so it asserts the new output value and signals HERE-IS-X to the yellow box which forwards the signal to the downstream stages.

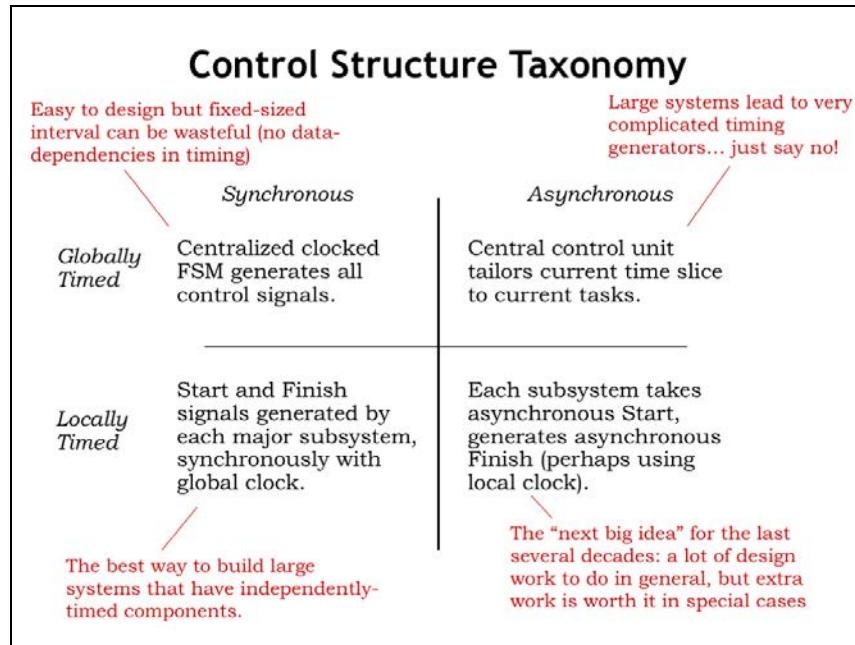
B is ready to consume the new input and so asserts its GOT-X output. Note that C is still waiting for its second input and has yet to assert its GOT-X output.

After B finishes its computation, it supplies a new value to C and asserts its HERE-IS-X output to let C know its second input is ready.

Now C is happy and signals both upstream stages that it has consumed its two inputs. Now that both GOT-X inputs are asserted, the yellow box asserts A's GOT-X input to let it know that the data has been transferred. Meanwhile B completes its part of the handshake, and C completes its transaction with B and A deasserts HERE-IS-X to indicate that it has seen its GOT-X input. When the B and C stages see their HERE-IS-X inputs go low, they finish their handshakes by deasserting their GOT-X outputs, and when they're both low, the yellow box lets A know the handshake is complete by deasserting A's GOT-X input.

Whew! The system has returned to the initial state where A is now ready to accept some future input value.

This is an elegant design based entirely on transition signaling. Each module is in complete control of when it consumes inputs and produces outputs, and so the system can process data at the fastest possible speed, rather than waiting for the worst-case processing delay.



Let's summarize what we've learned about controlling pipelined systems. The most straightforward approach is to use a pipeline with the system clock chosen to accommodate the worst-case processing time. These systems are easy to design but can't produce higher throughputs if the processing stages might run more quickly for some data values.

We saw that we could use a simple handshake protocol to move data through the system. All communication still happens on the rising edge of the system clock, but the specific clock edge used to transfer data is determined by the stages themselves.

It's tempting to wonder if we can might adjust the global clock period to take advantage of data-dependent processing speedups. But the necessary timing generators can be very complicated in large systems. It's usually much easier to use local communication between modules to determine system timing than trying to figure out all the constraints at the system level. So this approach isn't usually a good one.

But what about locally-timed asynchronous systems like the example we just saw? Each generation of engineers has heard the siren call of asynchronous logic. Sadly, it usually proves too hard to produce a provably reliable design for a large system, say, a modern computer. But there are special cases, such as the logic for integer division, where the data-dependent speed-ups make the extra work worthwhile.

Summary

Latency (L) = time it takes for given input to arrive at output

Throughput (T) = rate at which new outputs appear

For combinational circuits: $L = t_{PD}$ of circuit, $T = 1/L$

For K-pipelines ($K > 0$):

- always have register on output(s)
- K registers on every path from input to output
- Inputs available shortly after clock i , outputs available shortly after clock $(i+K)$
- $t_{CLK} = t_{PD,REG} + t_{PD}$ of slowest pipeline stage + t_{SETUP}
- $T = 1/t_{CLK}$
 - more throughput \Rightarrow split slowest pipeline stage(s)
 - use replication/interleaving if no further splits possible
- $L = K*t_{CLK} = K / T$
 - pipelined latency \geq combinational latency

We characterized the performance of our systems by measuring their latency and throughput. For combinational circuits, the latency is simply the propagation delay of the circuit and its throughput is just 1/latency.

We introduced a systematic strategy for designing K-pipelines, where there's a register on the outputs of each stage, and there are exactly K registers on every path from input to output. The period of the system clock t_{CLK} is determined by the propagation delay of the slowest pipeline stage. The throughput of a pipelined system is $1/t_{CLK}$ and its latency is Kt_{CLK} .

Pipelining is the key to increasing the throughput of most high-performance digital systems.

8: Design Tradeoffs

1. Optimizing Your Design
2. CMOS Static Power Dissipation
3. CMOS Dynamic Power Dissipation I
4. CMOS Dynamic Power Dissipation II
5. How Can We Reduce Power?
6. Fewer Transitions → Lower Power
7. Improving Speed: Adder Example
8. Performance/Cost Analysis
9. Carry Select Adders
10. 32-bit Carry Select Adder
11. Wanted: Faster Carry Logic!
12. Carry Look-ahead Adders (CLA)
13. 8-bit CLA (generate G & P)
14. 8-bit CLA (carry generation)
15. 8-bit CLA (complete)
16. Binary Multiplication
17. Combinational Multiplier
18. 2's Complement Multiplication
19. 2's Complement Multiplier
20. Increase Throughput With Pipelining
21. Carry-save Pipelined Multiplier
22. Reduce Area With Sequential Logic
23. Summary

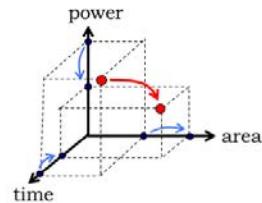
In this lecture, we're going to look into optimizing digital systems to make them smaller, faster, higher performance, more energy efficient, and so on. It would be wonderful if we could achieve all these goals at the same time and for some circuits we can. But, in general, optimizing in one dimension usually means doing less well in another. In other words, there are design tradeoffs to be made.

Optimizing Your Design

There are a large number of implementations of the same functionality -- each represents a different point in the area-time-power space

Optimization metrics:

1. Area of the design
2. Throughput
3. Latency
4. Power consumption
5. Energy of executing a task
6. ...



©Advanced Micro Devices (with permission)



Justin14 (CC BY-SA 4.0)

Making tradeoffs correctly requires that we have a clear understanding of our design goals for the system. Consider two different design teams: one is charged with building a high-end graphics card for gaming, the other with building the Apple watch.

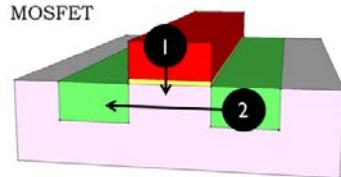
The team building the graphics card is mostly concerned with performance and, within limits, is willing to trade-off cost and power consumption to achieve their performance goals. Graphics cards have a set size, so there's a high priority in making the system small enough to meet the required size, but there's little to be gained in making it smaller than that.

The team building the watch has very different goals. Size and power consumption are critical since it has fit on a wrist and run all day without leaving scorch marks on the wearer's wrist!

Suppose both teams are thinking about pipelining part of their logic for increased performance. Pipelining registers are an obvious additional cost. The overlapped execution and higher t_{CLK} made possible by pipelining would increase the power consumption and the need to dissipate that power somehow. You can imagine the two teams might come to very different conclusions about the correct course of action!

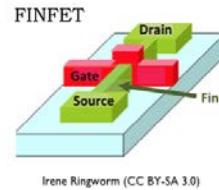
This chapter takes a look at some of the possible tradeoffs. But as designers you'll have to pick and choose which tradeoffs are right for your design. This is the sort of design challenge on which good engineers thrive! Nothing is more satisfying than delivering more than anyone thought possible within the specified constraints.

CMOS Static Power Dissipation



1 Tunneling current through gate oxide: SiO_2 is a very good insulator, but when very thin ($< 20\text{\AA}$) electrons can tunnel across.

- 2 Current leakage from drain to source even though MOSFET is “off” (aka sub-threshold conduction)
- Leakage gets larger as difference between V_{TH} and “off” gate voltage (eg, V_{OL} in an nFET) gets smaller. Significant as V_{TH} has become smaller.
 - Fix: 3D FINFET wraps gate around inversion region



Our first optimization topic is power dissipation, where the usual goal is to either meet a certain power budget, or to minimize power consumption while meeting all the other design targets.

In CMOS circuits, there are several sources of power dissipation, some under our control, some not.

Static power dissipation is power that is consumed even when the circuit is idle, i.e., no nodes are changing value. Using our simple switch model for the operation of MOSFETs, we’d expect CMOS circuits to have zero static power dissipation. And in the early days of CMOS, we came pretty close to meeting that ideal. But as the physical dimensions of the MOSFET have shrunk and the operating voltages have been lowered, there are two sources of static power dissipation in MOSFETs that have begun to loom large.

We’ll discuss the effects as they appear in n-channel MOSFETs, but keep in mind that they appear in p-channel MOSFETs too.

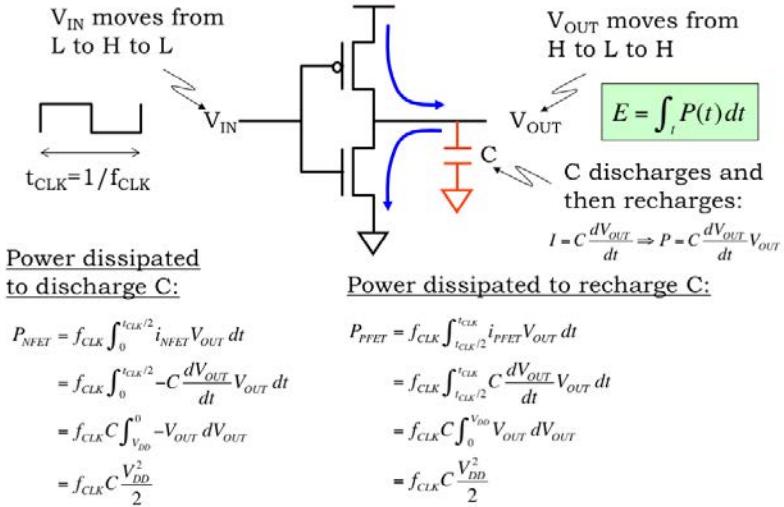
The first effect depends on the thickness of the MOSFET’s gate oxide, shown as the thin yellow layer in the MOSFET diagram on the left. In each new generation of integrated circuit technology, the thickness of this layer has shrunk, as part of the general reduction in all the physical dimensions. The thinner insulating layer means stronger electrical fields that cause a deeper inversion layer that leads to NFETs that carry more current, producing faster gate speeds. Unfortunately the layers are now thin enough that electrons can tunnel through the insulator, creating a small flow of current from the gate to the substrate. With billions of NFETs in a single circuit, even tiny currents can add up to non-negligible power drain.

The second effect is caused by current flowing between the drain and source of a NFET that is, in theory, not conducting because V_{GS} is less than the threshold voltage. Appropriately this effect is called sub-threshold conduction and is exponentially related to $V_{GS} - V_{TH}$ (a negative value when the NFET is off). So as V_{TH} has been reduced in each new generation of technology, $V_{GS} - V_{TH}$ is less negative and the sub-threshold conduction has increased.

One fix has been to change the geometry of the NFET so the conducting channel is a tall, narrow fin with the gate terminal wrapped around 3 sides, sometimes referred to as a tri-gate configuration. This has reduced the sub-threshold conduction by an order-of-magnitude or more, solving this particular problem for now.

Neither of these effects is under the control of the system designer, except of course, if they're free to choose an older manufacturing process! We mention them here so that you're aware that newer technologies often bring additional costs that then become part of the trade-off process.

CMOS Dynamic Power Dissipation



A designer does have some control over the dynamic power dissipation of the circuit, the amount of power spent causing nodes to change value during a sequence of computations. Each time a node changes from 0-to-1 or 1-to-0, currents flow through the MOSFET pullup and pulldown networks, charging and discharging the output node's capacitance and thus changing its voltage.

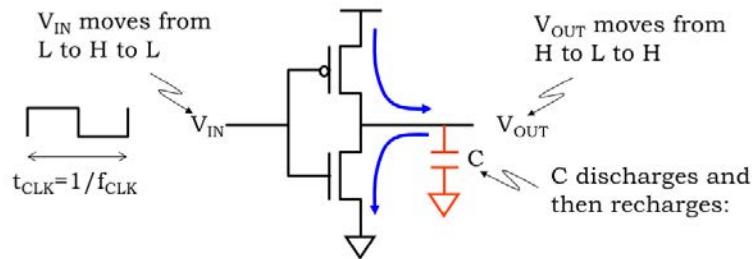
Consider the operation of an inverter. As the voltage of the input changes, the pullup and pulldown networks turn on and off, connecting the inverter's output node to VDD or ground. This charges or discharges the capacitance of the output node changing its voltage. We can compute the energy dissipated by integrating the instantaneous power associated with the current flow through the pullups and pulldowns over the time taken by the output transition.

The power dissipated across the resistance of the MOSFET channel is simply I_{DS} times V_{DS} . Here's the energy integral for the 1-to-0 transition of the output node, where we're measuring I_{DS} using the equation for the current flowing out of the output node's capacitor: $I = CdV/dt$. Assuming that the input signal is a clock signal of period t_{CLK} and that each transition is taking half a clock cycle, we can work through the math to determine that energy dissipated through the pulldown network is $0.5fCV_{DD}^2$, where the frequency f tells us the number of such transitions per second, C is the nodal capacitance, and V_{DD} (the power supply voltage) is the starting voltage of the nodal capacitor.

There's a similar integral for the current dissipated by pullup network when charging the capacitor and it yields the same result.

So one complete cycle of charging then discharging dissipates fCV^2 joules. Note the all this energy has come from the power supply — the first half is dissipated when the output node is charged and the other half stored as energy in the capacitor. Then the capacitor's energy is dissipated as it discharges.

CMOS Dynamic Power Dissipation



Power dissipated

$$= f C V_{DD}^2 \text{ per node}$$

$$= f N C V_{DD}^2 \text{ per chip}$$

where

$$f = \text{frequency of charge/discharge}$$

$$N = \text{number of changing nodes/chip}$$

"Back of the envelope": trends

$f \sim 1\text{GHz} = 1\text{e}9 \text{ cycles/sec}$	↑
$N \sim 1\text{e}8 \text{ changing nodes/cycle}$	↑
$C \sim 1\text{fF} = 1\text{e}-15 \text{ farads/node}$	↓
$V \sim 1\text{V}$	↓
$\Rightarrow 100 \text{ Watts}$	

These results are summarized in the lower left. We've added the calculation for the energy dissipation of an entire circuit assuming N of the circuit's nodes change each clock cycle.

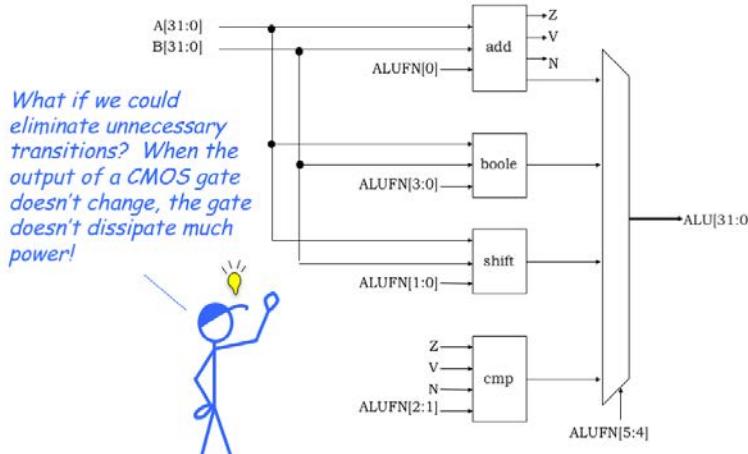
How much energy could be consumed by a modern integrated circuit? Here's a quick back-of-the-envelope estimate for a current generation CPU chip. It's operating at, say, 1 GHz and will have 100 million internal nodes that could change each clock cycle. Each nodal capacitance is around 1 femto Farad and the power supply is about 1V. With these numbers, the estimated power consumption is 100 watts. We all know how hot a 100W light bulb gets! You can see it would be hard to keep the CPU from overheating.

This is way too much energy to be dissipated in many applications, and modern CPUs intended, say, for laptops only dissipate a fraction of this energy. So the CPU designers must have some tricks up their sleeve, some of which we'll see in a minute.

But first notice how important it's been to be able to reduce the power supply voltage in modern integrated circuits. If we're able to reduce the power supply voltage from 3.3V to 1V, that alone accounts for more than a factor of 10 in power dissipation. So the newer circuit can be say, 5 times larger and 2 times faster with the same power budget!

Newer technologies trends are shown here. The net effect is that newer chips would naturally dissipate more power if we could afford to have them do so. We have to be very clever in how we use more and faster MOSFETs in order not to run up against the power dissipation constraints we face.

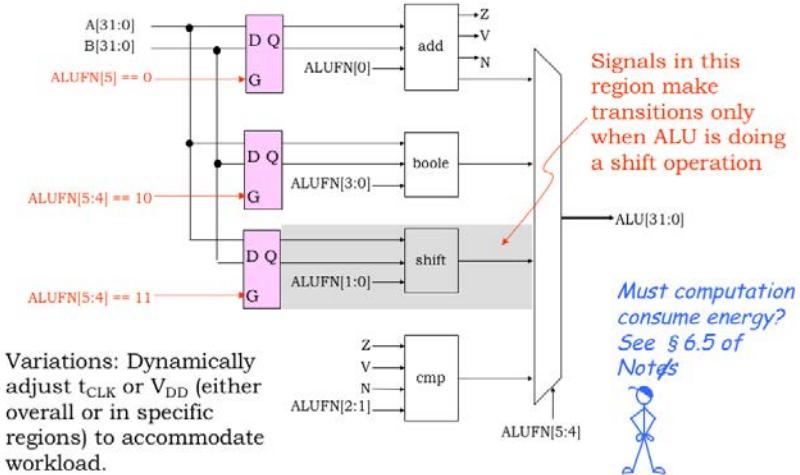
How Can We Reduce Power?



To see what we can do to reduce power consumption, consider the following diagram of an arithmetic and logic unit (ALU) like the one you'll design in the final lab in this part of the course. There are four independent component modules, performing the separate arithmetic, boolean, shifting and comparison operations typically found in an ALU. Some of the ALU control signals are used to select the desired result in a particular clock cycle, basically ignoring the answers produced by the other modules.

Of course, just because the other answers aren't selected doesn't mean we didn't dissipate energy in computing them. This suggests an opportunity for saving power! Suppose we could somehow "turn off" modules whose outputs we didn't need? One way to prevent them from dissipating power is to prevent the module's inputs from changing, thus ensuring that no internal nodes would change and hence reducing the dynamic power dissipation of the "off" module to zero.

Fewer Transitions → Lower Power



One idea is to put latches on the inputs to each module, only opening a module's input latch if an answer was required from that module in the current cycle. If a module's latch stayed closed, its internal nodes would remain unchanged, eliminating the module's dynamic power dissipation. This could save a substantial amount of power. For example, the shifter circuitry has many internal nodes and so has a large dynamic power dissipation. But there are comparatively few shift operations in most programs, so with our proposed fix, most of the time those energy costs wouldn't be incurred.

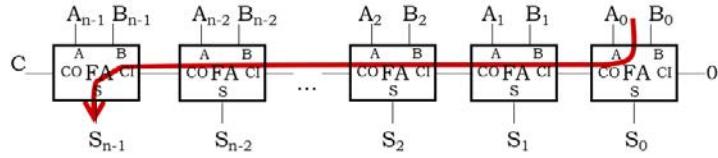
A more draconian approach to power conservation is to literally turn off unused portions of the circuit by switching off their power supply. This is more complicated to achieve, so this technique is usually reserved for special power-saving modes of operation, where we can afford the time it takes to reliably power the circuitry back up.

Another idea is to slow the clock (reducing the frequency of nodal transitions) when there's nothing for the circuit to do. This is particularly effective for devices that interact with the real world, where the time scales for significant external events are measured in milliseconds. The device can run slowly until an external event needs attention, then speed up the clock while it deals with the event.

All of these techniques and more are used in modern mobile devices to conserve battery power without limiting the ability to deliver bursts of performance. There is much more innovation waiting to be done in this area, something you may be asked to tackle as designers!

One last question is whether computation has to consume energy? There have been some interesting theoretical speculations about this question — see section 6.5 of the course notes to read more.

Improving Speed: Adder Example



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001.

$$t_{PD} = (N-1) * (\underbrace{t_{PD,NAND3} + t_{PD,NAND2}}_{CI \text{ to } CO} + \underbrace{t_{PD,XOR}}_{CI_{n-1} \text{ to } S_{n-1}}) \approx \Theta(N)$$

$\Theta(N)$ is read “order N” and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

The most straightforward way to improve performance is to reduce the propagation delay of a circuit. Let’s look at a perennial performance bottleneck: the ripple-carry adder.

To fix it, we first have to figure out the path from inputs to outputs that has the largest propagation delay, i.e., the path that’s determining the overall t_{PD} . In this case that path is the long carry chain following the carry-in to carry-out path through each full adder module. To trigger the path add -1 and 1 by setting the A inputs to all 1’s and the B input to all 0’s except for the low-order bit which is 1. The final answer is 0, but notice that each full adder has to wait for the carry-in from the previous stage before it produces 0 on its sum output and generates a carry-out for the next full adder. The carry really does ripple through the circuit as each full adder in turn does its thing.

To total propagation delay along this path is $N-1$ times the carry-in to carry-out delay of each full adder, plus the delay to produce the final bit of the sum.

How would the overall latency change if we, say, doubled the size of the operands, i.e., made N twice as large? It’s useful to summarize the dependency of the latency on N using the “order-of” notation to give us the big picture. Clearly as N gets larger the delay of the XOR gate at the end becomes less significant, so the order-of notation ignores terms that are relatively less important as N grows.

In this example, the latency is $\Theta(N)$, which tells us that the latency would be expected to essentially double if we made N twice as large.

Performance/Cost Analysis

"Order Of" notation:

" $g(n)$ is of order $f(n)$ " $g(n) = \Theta(f(n))$

$g(n)=\Theta(f(n))$ if there exist $C_2 \geq C_1 > 0$
such that for all but finitely many
integral $n \geq 0$

$$C_1 \cdot f(n) \leq g(n) \leq C_2 \cdot f(n)$$

$$\Theta(\dots) \text{ implies both}$$

inequalities;

$O(\dots)$ implies only
the second.



Example:

$$n^2 + 2n + 3 = \Theta(n^2)$$

since

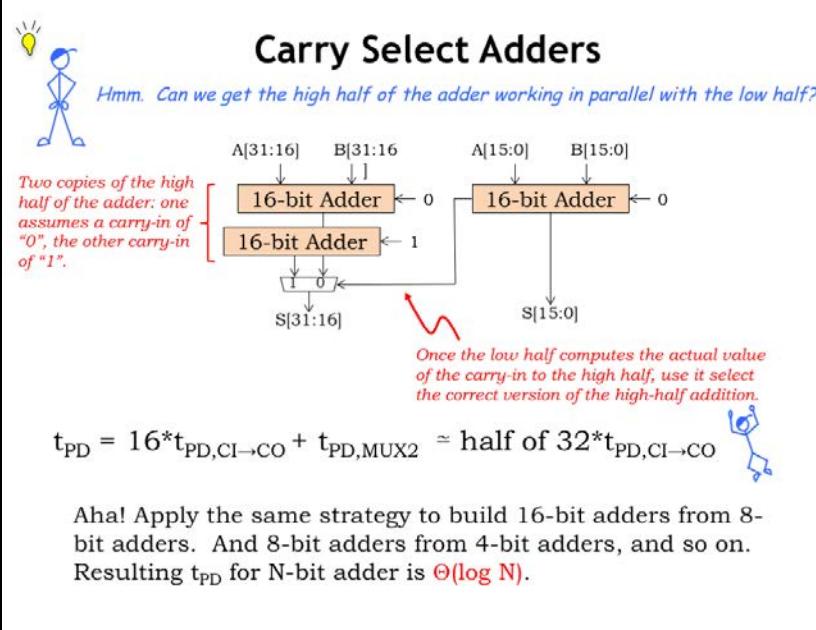
$$n^2 < n^2 + 2n + 3 < 2n^2$$

"almost always"

The order-of notation, which theoreticians call asymptotic analysis, tells us the term that would dominate the result as N grows. The yellow box contains the official definition, but an example might make it easier to understand what's happening.

Suppose we want to characterize the growth in the value of the equation $n^2 + 2n + 3$ as n gets larger. The dominant term is clearly n^2 and the value of our equation is bounded above and below by simple multiples of n^2 , except for finitely many values of n . The lower bound is always true for n greater than or equal to 0. And in this case, the upper bound doesn't hold only for n equal to 0, 1, 2, or 3. For all other positive values of n the upper inequality is true. So we'd say that this equation was $\Theta(n^2)$.

There are actually two variants for the order-of notation. We use the $\Theta()$ notation to indicate that $g(n)$ is bounded above AND below by multiples of $f(n)$. The $O()$ notation is used when $g(n)$ is only bounded above by a multiple of $f(n)$.



Here's a first attempt at improving the latency of our addition circuit. The trouble with the ripple-carry adder is that the high-order bits have to wait for the carry-in from the low-order bits. Is there a way in which we can get high half the adder working in parallel with the low half?

Suppose we wanted to build a 32-bit adder. Let's make two copies of the high 16 bits of the adder, one assuming the carry-in from the low 16 bits is 0, and the other assuming the carry-in is 1. So now we have three 16-bit adders, all of which can operate in parallel on newly arriving A and B inputs. Once the 16-bit additions are complete, we can use the actual carry-out from the low-half to select the answer from the particular high-half adder that used the matching carry-in value. This type of adder is appropriately named the carry-select adder.

The latency of this carry-select adder is just a little more than the latency of a 16-bit ripple-carry addition. This is approximately half the latency of the original 32-bit ripple-carry adder. So at a cost of about 50% more circuitry, we've halved the latency!

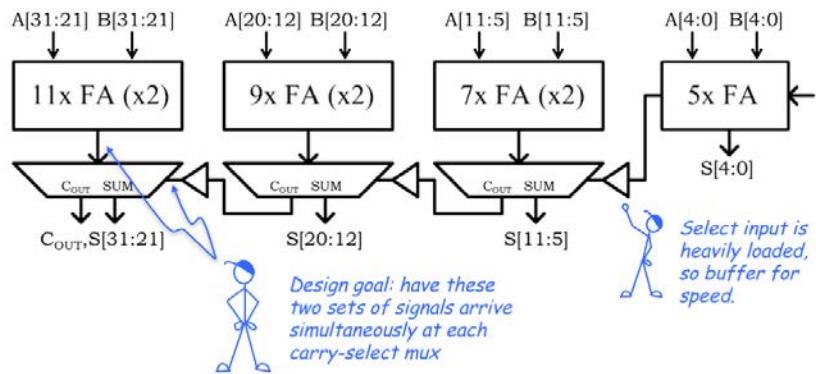
As a next step, we could apply the same strategy to halve the latency of the 16-bit adders. And then again to halve the latency of the 8-bit adders used in the previous step. At each step we halve the adder latency and add a MUX delay. After $\log_2(N)$ steps, N will be 1 and we're done.

At this point the latency would be some constant cost to do a 1-bit addition, plus $\log_2(N)$ times the MUX latency to select the right answers. So the overall latency of the carry-select adder is $\Theta(\log N)$. Note that $\log_2 N$ and $\log N$ only differ by a constant factor, so we ignore the base of the log in order-of notation.

The carry-select adder shows a clear performance-size tradeoff available to the designer.

32-bit Carry Select Adder

Practical Carry-select addition: choose block sizes so that trial sums and carry-in from previous stage arrive simultaneously at MUX.



Since adders play a big role in many digital systems, here's a more carefully engineered version of a 32-bit carry-select adder. You could try this in your ALU design!

The size of the adder blocks has been chosen so that the trial sums and the carry-in from the previous stage arrive at the carry-select MUX at approximately the same time. Note that since the select signal for the MUXes is heavily loaded we've included a buffer to make the select signal transitions faster.

This carry-select adder is about two-and-a-half times faster than a 32-bit ripple-carry adder at the cost of about twice as much circuitry. A great design to remember when you're looking to double the speed of your ALU!

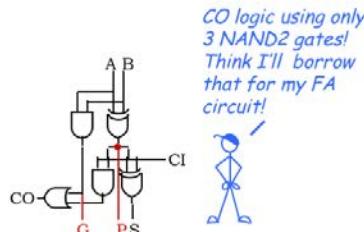
Wanted: Faster Carry Logic!

Let's see if we can improve the speed by rewriting the equations for C_{OUT} :

$$\begin{aligned}
 C_{OUT} &= AB + AC_{IN} + BC_{IN} \\
 &= AB + (A + B)C_{IN} \\
 &= \underset{\substack{\text{generate} \\ \text{propagate}}}{G + P} C_{IN} \quad \text{where } G = AB \text{ and } P = A + B
 \end{aligned}$$

Actually, P is usually defined as $P = A \oplus B$ which won't change C_{OUT} but will allow us to express S as a simple function of P and C_{IN} :

$$S = P \oplus C_{IN}$$



Here's another approach to improving the latency of our adder, this time focusing just on the carry logic. Early on in the course, we learned that by going from a chain of logic gates to a tree of logic gates, we could go from a linear latency to a logarithmic latency. Let's try to do that here.

We'll start by rewriting the equations for the carry-out from the full adder module. The final form of the rewritten equation has two terms. The G , or generate, term is true when the inputs will cause the module to generate a carry-out right away, without having to wait for the carry-in to arrive. The P , or propagate, term is true if the module will generate a carry-out only if there's a carry-in.

So there only two ways to get a carry-out from the module: it's either generated by the current module or the carry-in is propagated from the previous module.

Actually, it's usual to change the logic for the P term from "A OR B" to "A XOR B". This doesn't change the truth table for the carry-out but will allow us to express the sum output as " P XOR carry-in". Here's the schematic for the reorganized full adder module. The little sum-of-products circuit for the carry-out can be implemented using 3 2-input NAND gates, which is a bit more compact than the implementation for the three product terms we suggested in Lab 2. Time to update your full adder circuit!

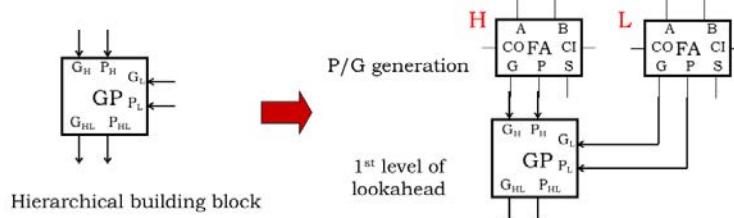
Carry Look-ahead Adders (CLA)

We can build a hierarchical carry chain by generalizing our definition of the Carry Generate/Propagate (GP) Logic. We start by dividing our addend into two parts, a higher part, H, and a lower part, L. The GP function can be expressed as follows:

$$G_{HL} = G_H + P_H G_L$$

$$P_{HL} = P_H P_L$$

Generate a carry out if the high part generates one, or if the low part generates one and the high part propagates it.
Propagate a carry if both the high and low parts propagate theirs.



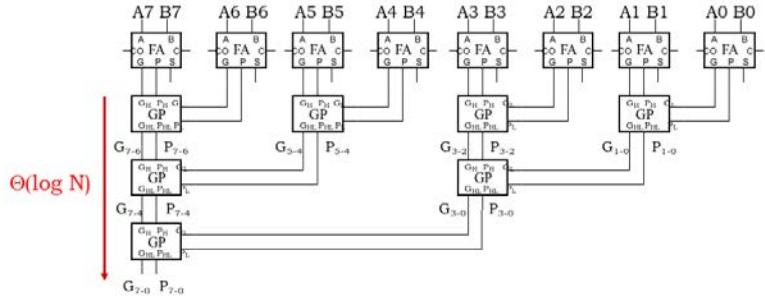
Now consider two adjacent adder modules in a larger adder circuit: we'll use the label H to refer to the high-order module and the label L to refer to the low-order module.

We can use the generate and propagate information from each of the modules to develop equations for the carry-out from the pair of modules treated as a single block.

We'll generate a carry-out from the block when a carry-out is generated by the H module, or when a carry-out is generated by the L module and propagated by the H module. And we'll propagate the carry-in through the block only if the L module propagates its carry-in to the intermediate carry-out and H module propagates that to the final carry-out. So we have two simple equations requiring only a couple of logic gates to implement.

Let's use these equations to build a generate-propagate (GP) module and hook it to the H and L modules as shown. The G and P outputs of the GP module tell us under what conditions we'll get a carry-out from the two individual modules treated as a single, larger block.

8-bit CLA (generate G & P)



We can build a tree of GP units to compute the generate and propagate logic for any sized adder. Assuming N is a power of 2, we'll need $N-1$ GP units.

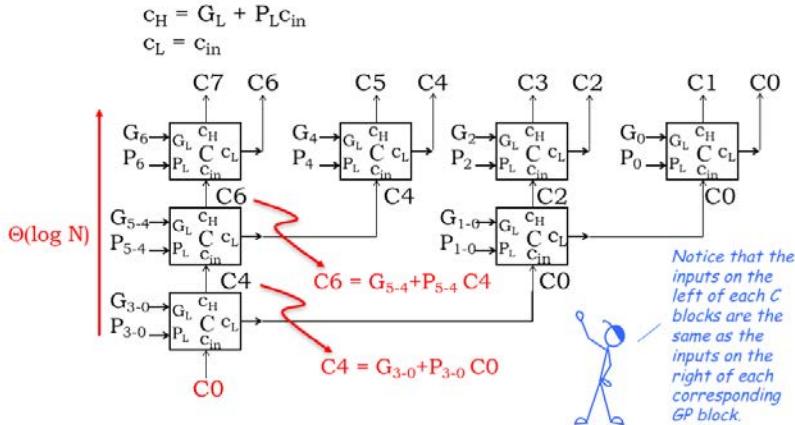
This will let us to quickly compute the carry-ins for each FA!

We can use additional layers of GP modules to build a tree of logic that computes the generate and propagate logic for adders with any number of inputs. For an adder with N inputs, the tree will contain a total of $N-1$ GP modules and have a latency that's $\Theta(\log N)$.

In the next step, we'll see how to use the generate and propagate information to quickly compute the carry-in for each of the original full adder modules.

8-bit CLA (carry generation)

Now, given a the value of the carry-in of the least-significant bit, we can generate the carries for every adder.



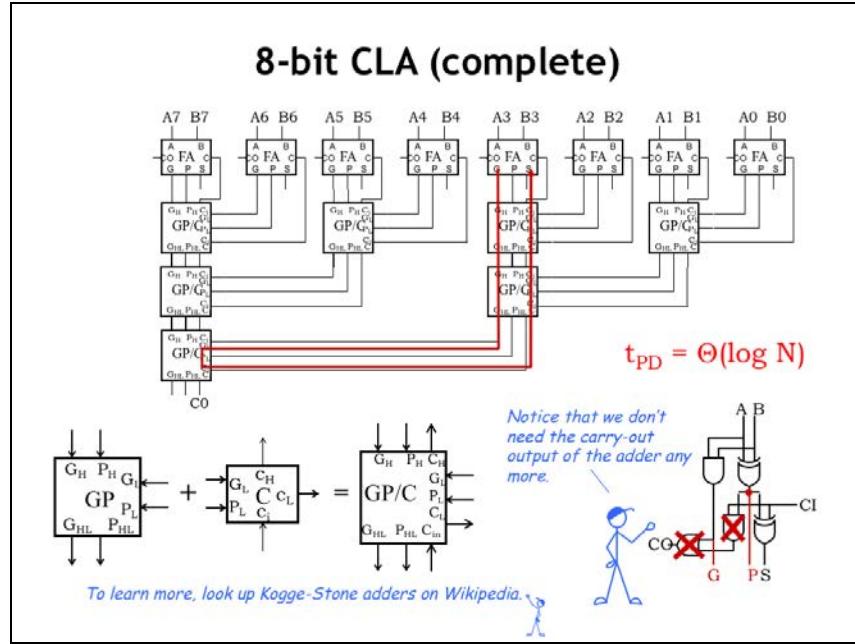
Once we're given the carry-in C_0 for the low-order bit, we can hierarchically compute the carry-in for each full adder module.

Given the carry-in to a block of adders, we simply pass it along as the carry-in to the low-half of the block. The carry-in for the high-half of the block is computed the using the generate and propagate information from the low-half of the block.

We can use these equations to build a C module and arrange the C modules in a tree as shown to use the C_0 carry-in to hierarchically compute the carry-in to each layer of successively smaller blocks, until we finally reach the full adder modules. For example, these equations show how C_4 is computed from C_0 , and C_6 is computed from C_4 .

Again the total propagation delay from the arrival of the C_0 input to the carry-ins for each full adder is $\Theta(\log N)$.

Notice that the G_L and P_L inputs to a particular C module are the same as two of the inputs to the GP module in the same position in the GP tree.



We can combine the GP module and C module to form a single carry-lookahead module that passes generate and propagate information up the tree and carry-in information down the tree. The schematic at the top shows how to wire up the tree of carry-lookahead modules.

And now we get to the payoff for all this hard work! The combined propagation delay to hierarchically compute the generate and propagate information on the way up and the carry-in information on the way down is $\Theta(\log N)$, which is then the latency for the entire adder since computing the sum outputs only takes one additional XOR delay. This is a considerable improvement over the $\Theta(N)$ latency of the ripple-carry adder.

A final design note: we no longer need the carry-out circuitry in the full adder module, so it can be removed.

Variations on this generate-propagate strategy form the basis for the fastest-known adder circuits. If you'd like to learn more, look up "Kogge-Stone adders" on Wikipedia.

Binary Multiplication*

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

* Actually unsigned binary multiplication

Hey, that looks like an AND gate

$$\begin{array}{r}
 & A_3 & A_2 & A_1 & A_0 \\
 \times & B_3 & B_2 & B_1 & B_0 \\
 \hline
 A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
 A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
 + & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
 \hline
 \end{array}$$

Multiplying N-digit number by M-digit number gives (N+M)-digit result

Easy part: forming partial products (just an AND gate since B_i is either 0 or 1)
 Hard part: adding M N-bit partial products

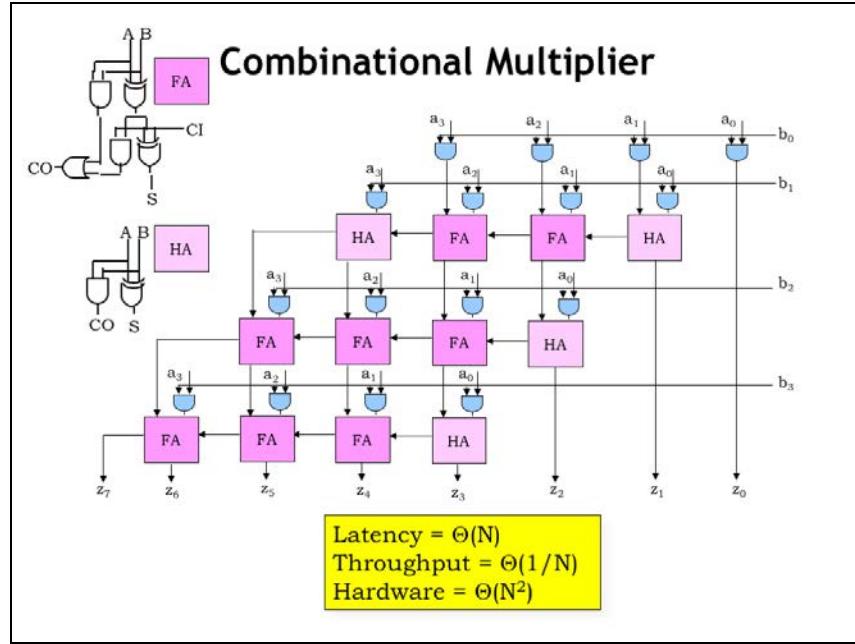
One of the biggest and slowest circuits in an arithmetic and logic unit is the multiplier. We'll start by developing a straightforward implementation and then, in the next section, look into tradeoffs to make it either smaller or faster.

Here's the multiplication operation for two unsigned 4-bit operands broken down into its component operations. This is exactly how we learned to do it in primary school. We take each digit of the multiplier (the B operand) and use our memorized multiplication tables to multiply it with each digit of the multiplicand (the A operand), dealing with any carries into the next column as we process the multiplicand right-to-left. The output from this step is called a partial product, and then we repeat the step for the remaining bits of the multiplier. Each partial product is shifted one digit to the left, reflecting the increasing weight of the multiplier digits.

In our case the digits are just single bits, i.e., they're 0 or 1 and the multiplication table is pretty simple! In fact, the 1-bit-by-1-bit binary multiplication circuit is just a 2-input AND gate. And look Mom, no carries!

The partial products are N bits wide since there are no carries. If the multiplier has M bits, there will be M partial products. And when we add the partial products together, we'll get an N+M bit result if we account for the possible carry-out from the high-order bit.

The easy part of the multiplication is forming the partial products — it just requires some AND gates. The more expensive operation is adding together the M N-bit partial products.



Here's the schematic for the combinational logic needed to implement the 4×4 multiplication, which would be easy to extend for larger multipliers (we'd need more rows) or larger multiplicands (we'd need more columns).

The $M \times N$ 2-input AND gates compute the bits of the M partial products. The adder modules add the current row's partial product with the sum of the partial products from the earlier rows. Actually there are two types of adder modules. The full adder is used when the modules needs three inputs. The simpler half adder is used when only two inputs are needed.

The longest path through this circuit takes a moment to figure out. Information is always moving either down a row or left to the adjacent column. Since there are M rows and, in any particular row, N columns, there are at most $N+M$ modules along any path from input to output. So the latency is $\Theta(N)$, since M and N differ by just some constant factor.

Since this is a combinational circuit, the throughput is just $1/\text{latency}$. And the total amount of hardware is $\Theta(N^2)$.

In the next section, we'll investigate how to reduce the hardware costs, or, separately, how to increase the throughput.

But before we do that, let's take a moment to see how the circuit would change if the operands were two's complement integers instead of unsigned integers.

2's Complement Multiplication

Step 1: two's complement operands so high order bit is -2^{N-1} . Must sign extend partial products and subtract the last one

$$\begin{array}{r}
 * \quad \begin{matrix} X_3 & X_2 & X_1 & X_0 \\ Y_3 & Y_2 & Y_1 & Y_0 \end{matrix} \\
 \hline
 \begin{matrix} X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\ + X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\ + X_3Y_2 & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\ - X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 \end{matrix} \\
 \hline
 \begin{matrix} Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0 \end{matrix}
 \end{array}$$

Step 2: don't want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction into add of (complement + 1).

$$\begin{array}{r}
 \begin{matrix} X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\ + 1 & + X_3Y_1 & X_3Y_1 & X_3Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 \\ + X_3Y_2 & X_3Y_2 & X_3Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 \\ + X_3Y_3 & X_3Y_3 & X_3Y_3 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 \end{matrix} \\
 \hline
 \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}
 \end{array}$$

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

$$\begin{array}{r}
 \begin{matrix} \overline{X_3Y_0} & X_2Y_0 & X_1Y_0 & X_0Y_0 \\ + \overline{X_3Y_1} & X_2Y_1 & X_1Y_1 & X_0Y_1 \\ + \overline{X_3Y_2} & X_2Y_2 & X_1Y_2 & X_0Y_2 \\ + \overline{X_3Y_3} & X_2Y_3 & X_1Y_3 & X_0Y_3 \\ + 1 & & & \end{matrix} \\
 \hline
 \begin{matrix} 1 & 1 & 1 & 1 \end{matrix}
 \end{array}$$

Step 4: finish computing the constants...

$$\begin{array}{r}
 \begin{matrix} \overline{X_3Y_0} & X_2Y_0 & X_1Y_0 & X_0Y_0 \\ + \overline{X_3Y_1} & X_2Y_1 & X_1Y_1 & X_0Y_1 \\ + \overline{X_3Y_2} & X_2Y_2 & X_1Y_2 & X_0Y_2 \\ + \overline{X_3Y_3} & X_2Y_3 & X_1Y_3 & X_0Y_3 \\ + 1 & & & 1 \end{matrix} \\
 \hline
 \begin{matrix} 1 & 1 & 1 & 1 \end{matrix}
 \end{array}$$

Result: multiplying 2's complement operands takes just about same amount of hardware as multiplying unsigned operands!

With a two's complement multiplier and multiplicand, the high-order bit of each has negative weight. So when adding together the partial products, we'll need to sign-extend each of the N-bit partial products to the full N+M-bit width of the addition. This will ensure that a negative partial product is properly treated when doing the addition. And, of course, since the high-order bit of the multiplier has a negative weight, we'd subtract instead of add the last partial product.

Now for the clever bit. We'll add 1's to various of the columns and then subtract them later, with the goal of eliminating all the extra additions caused by the sign-extension. We'll also rewrite the subtraction of the last partial product as first complementing the partial product and then adding 1. This is all a bit mysterious but...

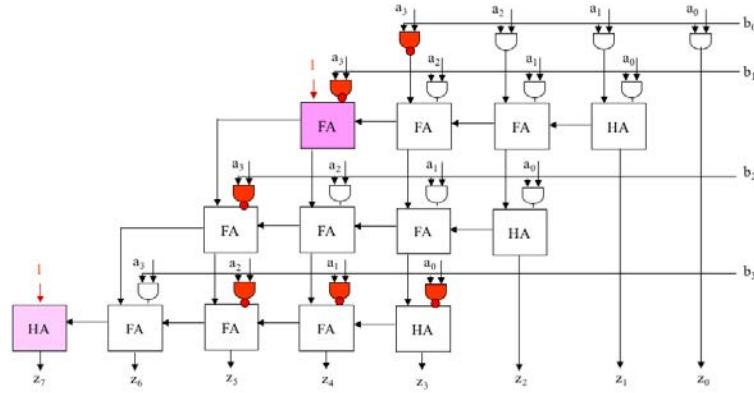
Here in step 3 we see the effect of all the step 2 machinations. Let's look at the high order bit of the first partial product X3Y0. If that partial product is non-negative, X3Y0 is a 0, so all the sign-extension bits are 0 and can be removed. The effect of adding a 1 in that position is to simply complement X3Y0.

On the other hand, if that partial product is negative, X3Y0 is 1, and all the sign-extension bits are 1. Now when we add a 1 in that position, we complement the X3Y0 bit back to 0, but we also get a carry-out. When that's added to the first sign-extension bit (which is itself a 1), we get zero with another carry-out. And so on, with all the sign-extension bits eventually getting flipped to 0 as the carry ripples to the end. Again the net effect of adding a 1 in that position is to simply complement X3Y0.

We do the same for all the other sign-extended partial products, leaving us with the results shown here.

In the final step we do a bit of arithmetic on the remaining constants to end up with this table of work to do. Somewhat to our surprise, this isn't much different than the original table for the unsigned multiplication. There are a few partial product bits that need to be complemented, and two 1-bits that need to be added to particular columns.

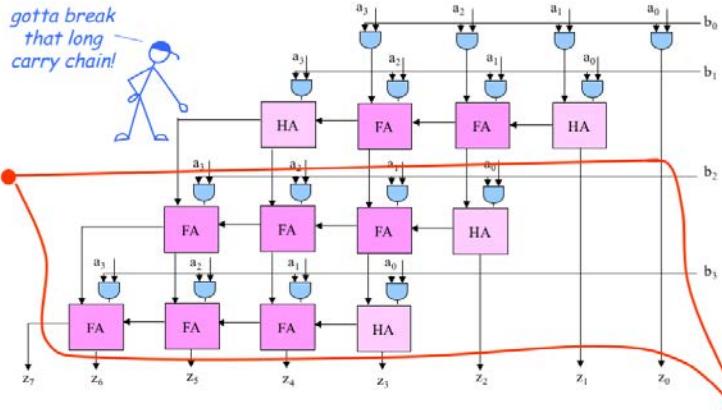
2's Complement Multiplier



The resulting circuitry is shown here. We've changed some of the AND gates to NAND gates to perform the necessary complements. And we've changed the logic necessary to deal with the two 1-bit that needed to be added in.

The colored elements show the changes made from the original unsigned multiplier circuitry. Basically, the circuit for multiplying two's complement operands has the same latency, throughput and hardware costs as the original circuitry.

Increase Throughput With Pipelining



Before pipelining: Throughput = $\sim 1/(2N) = \Theta(1/N)$
 After pipelining: Throughput = $\sim 1/N = \Theta(1/N)$

Let's see if we can improve the throughput of the original combinational multiplier design. We'll use our patented pipelining process to divide the processing into stages with the expectation of achieving a smaller clock period and higher throughput. The number to beat is approximately 1 output every $2N$, where N is the number of bits in each of the operands.

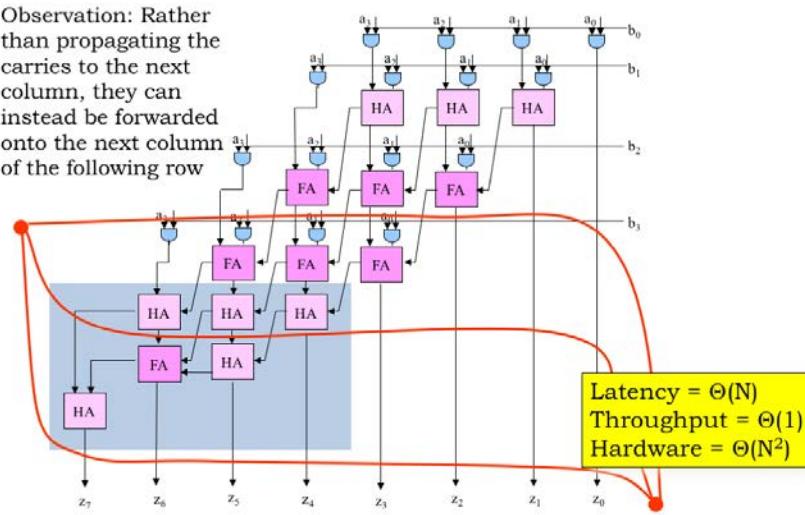
Our first step is to draw a contour across all the outputs. This creates a 1-pipeline, which gets us started but doesn't improve the throughput.

Let's add another contour, dividing the computations about in half. If we're on the right track, we hope to see some improvement in the throughput. And indeed we do: the throughput has doubled. Yet both the before and after throughputs are $\Theta(1/N)$. Is there any hope of a dramatically better throughput?

The necessary insight is that as long as an entire row is inside a single pipeline stage, the latency of the stage will be $\Theta(N)$ since we have to leave time for the N -bit ripple-carry add to complete.

“Carry-save” Pipelined Multiplier

Observation: Rather than propagating the carries to the next column, they can instead be forwarded onto the next column of the following row



There are several ways to tackle this problem. The technique illustrated here will be useful in our next task. In this schematic we've redrawn the carry chains. Carry-outs are still connected to a module one column to the left, but, in this case, a module that's down a row. So all the additions that need to happen in a specific column still happen in that column, we've just reorganized which row does the adding.

Let's pipeline this revised diagram, creating stages with approximately two module's worth of propagation delay.

The horizontal contours now break the long carry chains and the latency of each stage is now constant, independent of N.

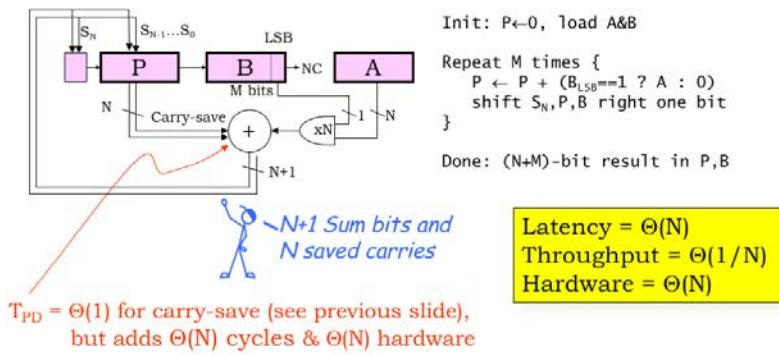
Note that we had to add $\Theta(N)$ extra rows to take care of propagating the carries all the way to the end — the extra circuitry is shown in the grey box.

To achieve a latency that's independent of N in each stage, we'll need $\Theta(N)$ contours. This means the latency is constant, which in order-of notation we write as $\Theta(1)$. But this means the clock period is now independent of N, as is the throughput — they are both $\Theta(1)$. With $\Theta(N)$ contours, there are $\Theta(N)$ pipeline stages, so the system latency is $\Theta(N)$. The hardware cost is still $\Theta(N^2)$. So the pipelined carry-save multiplier has dramatically better throughput than the original circuit, another design tradeoff we can remember for future use.

We'll use the carry-save technique in our next optimization, which is to implement the multiplier using only $\Theta(N)$ hardware.

Reduce Area With Sequential Logic

Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that *processes a single partial product at a time* and then cycle the circuit M times:



This sequential multiplier design computes a single partial product in each step and adds it to the accumulating sum. It will take $\Theta(N)$ steps to perform the complete multiplication.

In each step, the next bit of the multiplier, found in the low-order bit of the B register, is ANDed with the multiplicand to form the next partial product. This is sent to the N-bit carry-save adder to be added to the accumulating sum in the P register. The value in the P register and the output of the adder are in “carry-save format”. This means there are 32 data bits, but, in addition, 31 saved carries, to be added to the appropriate column in the next cycle. The output of the carry-save adder is saved in the P register, then in preparation for the next step both P and B are shifted right by 1 bit. So each cycle one bit of the accumulated sum is retired to the B register since it can no longer be affected by the remaining partial products. Think of it this way: instead of shifting the partial products left to account for the weight of the current multiplier bit, we’re shifting the accumulated sum right!

The clock period needed for the sequential logic is quite small, and, more importantly is independent of N. Since there’s no carry propagation, the latency of the carry-save adder is very small, i.e., only enough time for the operation of a single full adder module.

After $\Theta(N)$ steps, we’ve generated the necessary partial products, but will need to continue for another $\Theta(N)$ steps to finish propagating the carries through the carry-save adder.

But even at $2N$ steps, the overall latency of the multiplier is still $\Theta(N)$. And at the end of the $2N$ steps, we produce the answer in the P and B registers combined, so the throughput is $\Theta(1/N)$. The big change is in the hardware cost at $\Theta(N)$, a dramatic improvement over the $\Theta(N^2)$ hardware cost of the original combinational multiplier.

This completes our little foray into multiplier designs. We’ve seen that with a little cleverness we can create designs with $\Theta(1)$ throughput, or designs with only $\Theta(N)$ hardware. The technique of carry-save addition is useful in many situations and its use can improve throughput at constant hardware cost, or save hardware at a constant throughput.

Summary

- Power dissipation can be controlled by dynamically varying T_{CLK} , V_{DD} or by selectively eliminating unnecessary transitions.
- Functions with N inputs have minimum latency of $O(\log N)$ if output depends on all the inputs. But it can take some doing to find an implementation that achieves this bound.
- Performing operations in “slices” is a good way to reduce hardware costs (but latency increases)
- Pipelining can increase throughput (but latency increases)
- Asymptotic analysis only gets you so far – factors of 10 matter in real life and typically N isn’t a parameter that’s changing within a given design.

This discussion of design tradeoffs completes Part 1 of the course. We’ve covered a lot of ground in the last eight lectures.

We started by looking at the mathematics underlying information theory and used it to help evaluate various alternative ways of effectively using sequences of bits to encode information content. Then we turned our attention to adding carefully-chosen redundancies to our encoding to ensure that we could detect and even correct errors that corrupted our bit-level encodings.

Next we learned how analog signaling accumulates errors as we added processing elements to our system. We solved the problem by using voltages “digitally” choosing two ranges of voltages to encode the bit values 0 and 1. We had different signaling specifications for outputs and inputs, adding noise margins to make our signaling more robust. Then we developed the static discipline for combinational devices and were led to the conclusion that our devices had to be non-linear and exhibit gains > 1 .

In our study of combinational logic, we first learned about the MOSFET, a voltage-controlled switch. We developed a technique for using MOSFETs to build CMOS combinational logic gates, which met all the criteria of the static discipline. Then we discussed systematic ways of synthesizing larger combinational circuits that could implement any functionality we could express in the form a truth table.

To be able to perform sequences of operations, we first developed a reliable bistable storage element based on a positive feedback loop. To ensure the storage elements worked correctly we imposed the dynamic discipline which required inputs to the storage elements to be stable just before and after the time the storage element was transitioned to “memory mode”. We introduced finite-state machines as a useful abstraction for designing sequential logic. And then we figured out how to deal with asynchronous inputs in way that minimized the chance of incorrect operation due to metastability.

In the last two lectures we developed latency and throughput as performance measures for digital systems and discussed ways of achieving maximum throughput under various constraints. We discussed how it’s possible to make tradeoffs to achieve goals of minimizing power dissipation and increasing performance through decreased latency or increased throughput.

Whew! That's a lot of information in a short amount of time.

9: Instruction Set Architectures

1. Example: Factorial I
2. Example: Factorial II
3. Datapath for Factorial
4. Control FSM for Factorial
5. Control FSM Hardware
6. So Far: Single-purpose Hardware
7. A Simple Programmable Datapath
8. A Control FSM for Factorial
9. New Problem → New Control FSM
10. The Eniac Computer
11. Programming The Eniac
12. The von Neumann Model
13. Key Idea: Stored-program Computer
14. Anatomy of a von Neumann Computer
15. Instructions
16. Instruction Set Architecture (ISA)
17. ISA Design
18. Beta ISA: Storage
19. Storage Conventions
20. Beta ISA: Instructions
21. Beta ALU Instructions
22. Implementation Sketch #1
23. Should We Support Constant Operands?
24. Beta ALU Instructions with Constant
25. Implementation Sketch #2
26. Beta Load and Store Instructions
27. Using LD and ST
28. Can We Solve Factorial with ALU Instructions?
29. Beta Branch Instructions
30. Can We Solve Factorial Now?
31. Beta JMP Instruction
32. Beta ISA Summary

Welcome to Part 2 of 6.004! In this part of the course, we turn our attention to the design and implementation of digital systems that can perform useful computations on different types of binary data. We'll come up with a general-purpose design for these systems, which we call "computers", so that they can serve as useful tools in many diverse application areas. Computers were first used to perform numeric calculations in science and engineering, but today they are used as the central control element in any system where complex behavior is required.

Example: Factorial

```
factorial(N) = N! = N*(N-1)*...*1
```

C:

```
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)

initially: a = 1, b = 5
after iter 1: a = 5, b = 4
after iter 2: a = 20, b = 3
after iter 3: a = 60, b = 2
after iter 4: a = 120, b = 1
after iter 5: a = 120, b = 0
Done!
```

We have a lot to do in this lecture, so let's get started! Suppose we want to design a system to compute the factorial function on some numeric argument N . $N!$ is defined as the product of N times $N-1$ times $N-2$, and so on down to 1.

We can use a programming language like C to describe the sequence of operations necessary to perform the factorial computation. In this program there are two variables, “ a ” and “ b ”. “ a ” is used to accumulate the answer as we compute it step-by-step. “ b ” is used to hold the next value we need to multiply. “ b ” starts with the value of the numeric argument N . The DO loop is where the work gets done: on each loop iteration we perform one of the multiplies from the factorial formula, updating the value of the accumulator “ a ” with the result, then decrementing “ b ” in preparation for the next loop iteration.

Example: Factorial

factorial(N) = N! = N*(N-1)*...*1

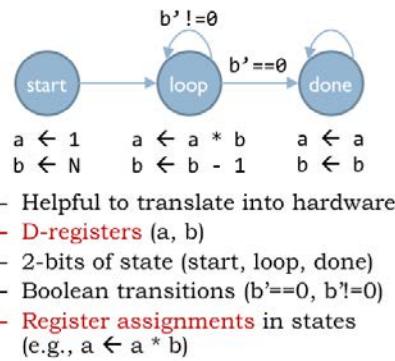
C:

```

int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)

start: a ← 1, b ← 5
loop: a ← 5, b ← 4
loop: a ← 20, b ← 3
loop: a ← 60, b ← 2
loop: a ← 120, b ← 1
loop: a ← 120, b ← 0
done:
  
```

High-level FSM:

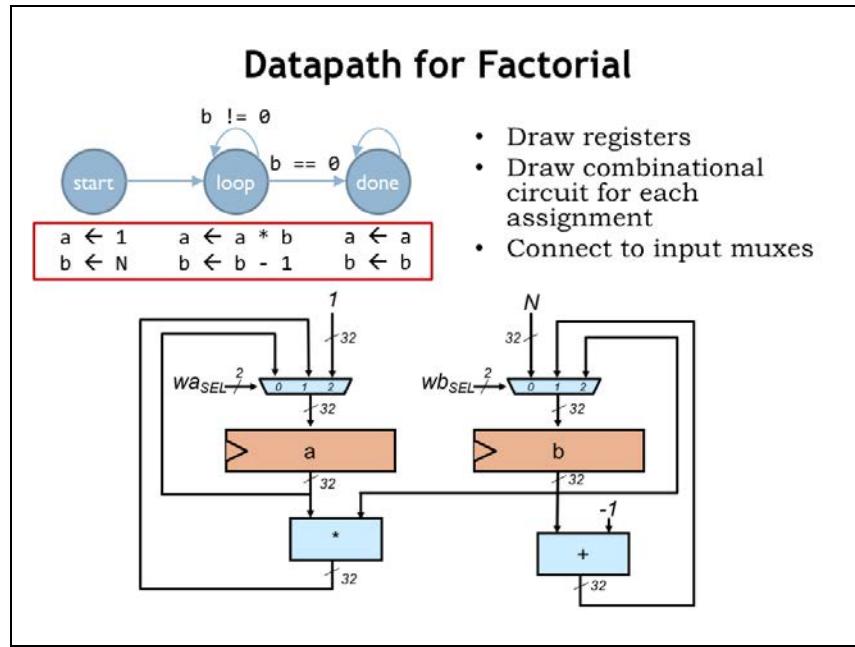


- Helpful to translate into hardware
- D-registers (a, b)
- 2-bits of state (start, loop, done)
- Boolean transitions ($b'==0$, $b'!=0$)
- Register assignments in states
(e.g., $a \leftarrow a * b$)

If we want to implement a digital system that performs this sequence of operations, it makes sense to use sequential logic! Here's the state transition diagram for a high-level finite-state machine designed to perform the necessary computations in the desired order. We call this a high-level FSM since the “outputs” of each state are more than simple logic levels. They are formulas indicating operations to be performed on source variables, storing the result in a destination variable.

The sequence of states visited while the FSM is running mirrors the steps performed by the execution of the C program. The FSM repeats the LOOP state until the new value to be stored in “b” is equal to 0, at which point the FSM transitions into the final DONE state.

The high-level FSM is useful when designing the circuitry necessary to implement the desired computation using our digital logic building blocks. We'll use 32-bit D-registers to hold the “a” and “b” values. And we'll need a 2-bit D-register to hold the 2-bit encoding of the current state, *i.e.*, the encoding for either START, LOOP or DONE. We'll include logic to compute the inputs required to implement the correct state transitions. In this case, we need to know if the new value for “b” is zero or not. And, finally, we'll need logic to perform multiply and decrement, and to select which value should be loaded into the “a” and “b” registers the end of each FSM cycle.

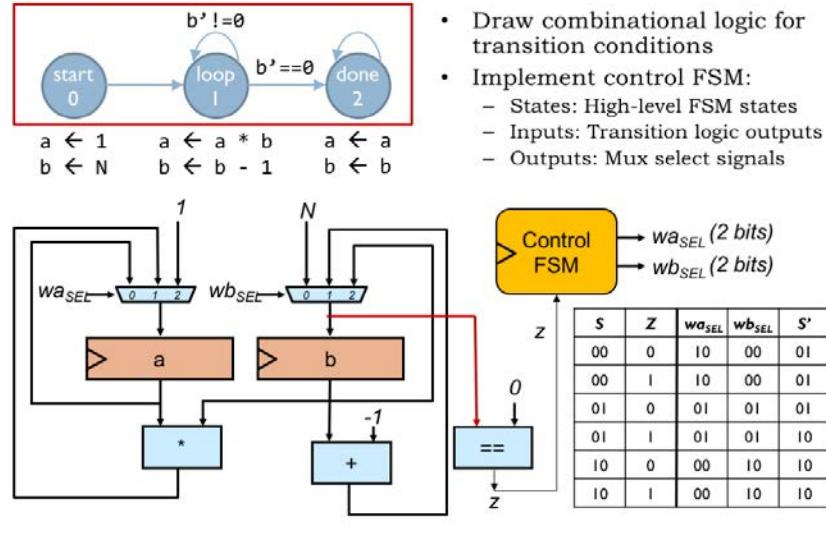


Let's start by designing the logic that implements the desired computations — we call this part of the logic the “datapath”.

First we'll need two 32-bit D-registers to hold the “a” and “b” values. Then we'll draw the combinational logic blocks needed to compute the values to be stored in those registers. In the START state , we need the constant 1 to load into the “a” register and the constant N to load into the “b” register. In the LOOP state, we need to compute $a \times b$ for the “a” register and $b - 1$ for the “b” register. Finally, in the DONE state , we need to be able to reload each register with its current value.

We'll use multiplexers to select the appropriate value to load into each of the data registers. These multiplexers are controlled by 2-bit select signals that choose which of the three 32-bit input values will be the 32-bit value to be loaded into the register. So by choosing the appropriate values for WASEL and WBSEL, we can make the datapath compute the desired values at each step in the FSM's operation.

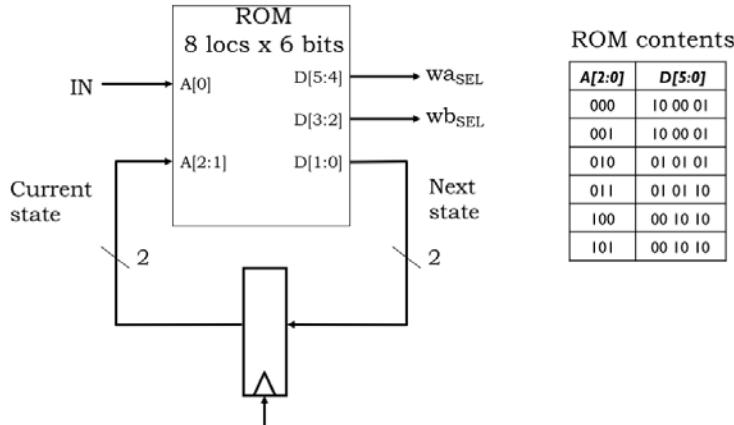
Control FSM for Factorial



Next we'll add the combinational logic needed to control the FSM's state transitions. In this case, we need to test if the new value to be loaded into the "b" register is zero. The Z signal from the datapath will be 1 if that's the case and 0 otherwise.

Now we're all set to add the hardware for the control FSM, which has one input (Z) from the datapath and generates two 2-bit outputs (WASEL and WBSEL) to control the datapath. Here's the truth table for the FSM's combinational logic. S is the current state, encoded as a 2-bit value, and S' is the next state.

Control FSM Hardware



Using our skills from Part 1 of the course, we're ready to draw a schematic for the system! We know how to design the appropriate multiplier and decrement circuitry. And we can use our standard register-and-ROM implementation for the control FSM. The Z signal from the datapath is combined with the 2 bits of current state to form the 3 inputs to the combinational logic, in this case realized by a read-only memory with $2^3 = 8$ locations. Each ROM location has the appropriate values for the 6 output bits: 2 bits each for WASEL, WBSEL, and next state. The table on the right shows the ROM contents, which are easily determined from the table on the previous slide.

So Far: Single-Purpose Hardware

- Problem → Procedure (High-level FSM) → Implementation
- **Systematic way** to implement high-level FSM as a datapath + control FSM
 - Is this implementation an FSM itself?
 - If so, can you draw the truth table?
- How should we generalize our approach so we can solve many problems with one set of hardware?
 - More storage for operands and results
 - A larger repertoire of operations
 - General-purpose datapath

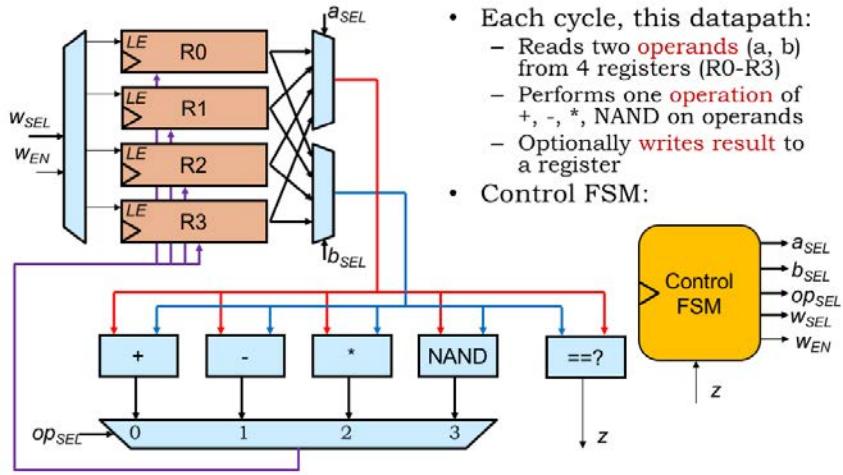
Okay, we've figured out a way to design hardware to perform a particular computation: Draw the state transition diagram for an FSM that describes the sequence of operations needed to complete the computation. Then construct the appropriate datapath, using registers to store values and combinational logic to implement the needed operations. Finally build an FSM to generate the control signals required by the datapath.

Is the datapath plus control logic itself an FSM? Well, it has registers and some combinational logic, so, yes, it is an FSM. Can we draw the truth table? In theory, yes. In practice, there are 66 bits of registers and hence 66 bits of state, so our truth table would need 2^{66} rows! Hmm, not very likely that we'd be able to draw the truth table! The difficulty comes from thinking of the registers in the datapath as part of the state of our super-FSM. That's why we think about the datapath as being separate from the control FSM.

So how do we generalize this approach so we can use one computer circuit to solve many different problems. Well, most problems would probably require more storage for operands and results. And a larger list of allowable operations would be handy. This is actually a bit tricky: what's the minimum set of operations we can get away with? As we'll see later, surprisingly simple hardware is sufficient to perform any realizable computation. At the other extreme, many complex operations (*e.g.*, fast fourier transform) are best implemented as sequences of simpler operations (*e.g.*, add and multiply) rather than as a single massive combinational circuit. These sorts of design tradeoffs are what makes computer architecture fun!

We'd then combine our larger storage with logic for our chosen set of operations into a general purpose datapath that could be reused to solve many different problems. Let's see how that would work...

A Simple Programmable Datapath



Here's a datapath with 4 data registers to hold results. The ASEL and BSEL multiplexers allow any of the data registers to be selected as either operand for our repertoire of arithmetic and boolean operations. The result is selected by the OPSEL MUX and can be written back into any of the data registers by setting the WEN control signal to 1 and using the 2-bit WSEL signal to select which data register will be loaded at the next rising clock edge. Note that the data registers have a load-enable control input: when this signal is 1, the register will load a new value from its D input, otherwise it ignores the D input and simply reloads its previous value.

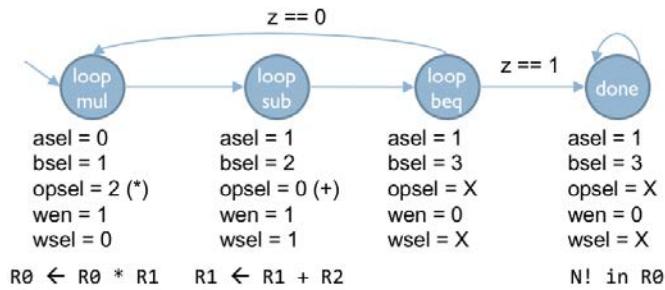
And, of course, we'll add a control FSM to generate the appropriate sequence of control signals for the datapath. The Z input from the datapath allows the system to perform data-dependent operations, where the sequence of operations can be influenced by the actual values in the data registers.

A Control FSM for Factorial

- Assume initial register contents:

R_0 value = 1
 R_1 value = N
 R_2 value = -1
 R_3 value = 0

- Control FSM:



Here's the state transition diagram for the control FSM we'd use if we wanted to use this datapath to compute factorial assuming the initial contents of the data registers are as shown. We need a few more states than in our initial implementation since this datapath can only perform one operation at each step. So we need three steps for each iteration: one for the multiply, one for the decrement, and one for the test to see if we're done.

As seen here, it's often the case that general-purpose computer hardware will need more cycles and perhaps involve more hardware than an optimized single-purpose circuit.

New Problem → New Control FSM

- You can solve many more problems with this datapath!
 - Exponentiation, division, square root, ...
 - But nothing that requires more than four registers
- By designing a control FSM, we are **programming the datapath**
- Early digital computers were programmed this way!
 - ENIAC (1943):
 - First general-purpose digital computer
 - Programmed by setting huge array of dials and switches
 - Reprogramming it took about 3 weeks

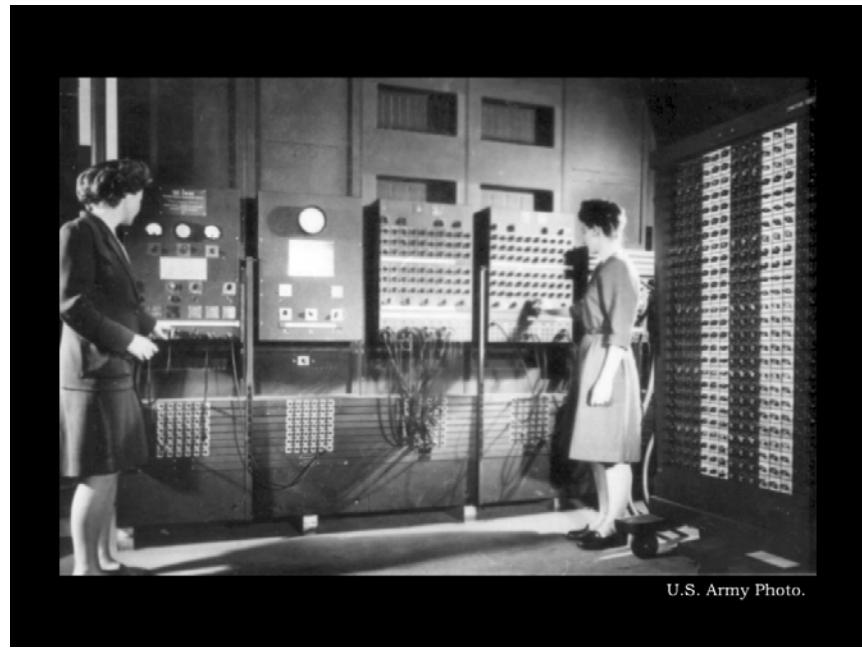
You can solve many different problems with this system: exponentiation, division, square root, and so on, so long as you don't need more than four data registers to hold input data, intermediate results, or the final answer.

By designing a control FSM, we are in effect “programming” our digital system, specifying the sequence of operations it will perform.



"Eniac" by Unknown - U.S. Army Photo.

This is exactly how the early digital computers worked! Here's a picture of the ENIAC computer built in 1943 at the University of Pennsylvania.

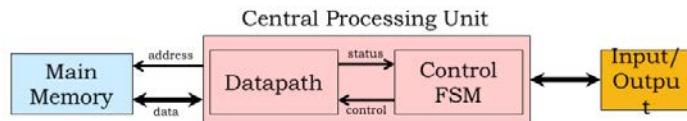


The Wikipedia article on the ENIAC tells us that “ENIAC could be programmed to perform complex sequences of operations, including loops, branches, and subroutines. The task of taking a problem and mapping it onto the machine was complex, and usually took weeks. After the program was figured out on paper, the process of getting the program into ENIAC by manipulating its switches and cables could take days. This was followed by a period of verification and debugging, aided by the ability to execute the program step by step.”

It's clear that we need a less cumbersome way to program our computer!

The von Neumann Model

- Many approaches to build a general-purpose computer. Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



- Central processing unit:
Performs operations on values in registers & memory
- Main memory:
Array of W words of N bits each
- Input/output devices to communicate with the outside world

There are many approaches to building a general-purpose computer that can be easily re-programmed for new problems. Almost all modern computers are based on the “stored program” computer architecture developed by John von Neumann in 1945, which is now commonly referred to as the “von Neumann model”.

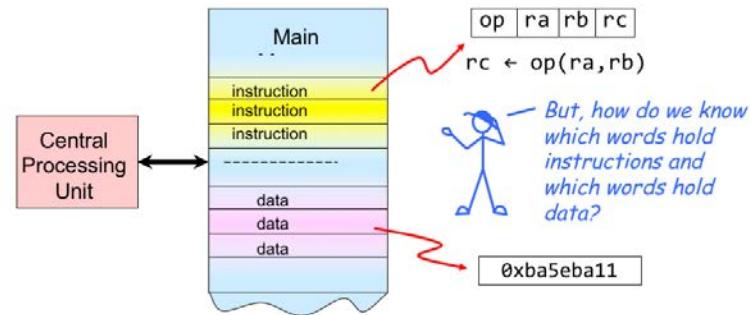
The von Neumann model has three components. There’s a central processing unit (aka the CPU) that contains a datapath and control FSM as described previously.

The CPU is connected to a read/write memory that holds some number W of words, each with N bits. Nowadays, even small memories have a billion words and the width of each location is at least 32 bits (usually more). This memory is often referred to as “main memory” to distinguish it from other memories in the system. You can think of it as an array: when the CPU wishes to operate on values in memory, it sends the memory an array index, which we call the address, and, after a short delay (currently 10’s of nanoseconds) the memory will return the N-bit value stored at that address. Writes to main memory follow the same protocol except, of course, the data flows in the opposite direction. We’ll talk about memory technologies a couple of lectures from now.

And, finally, there are input/output devices that enable the computer system to communicate with the outside world or to access data storage that, unlike main memory, will remember values even when turned off.

Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program



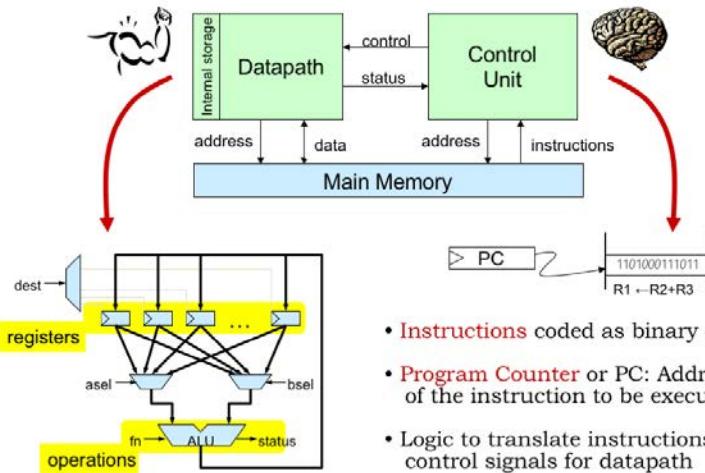
The key idea is to use main memory to hold the instructions for the CPU as well as data. Both instructions and data are, of course, just binary data stored in main memory.

Interpreted as an instruction, a value in memory can be thought of as a set of fields containing one or bits encoding information about the actions to be performed by the CPU. The opcode field indicates the operation to be performed (e.g., ADD, XOR, COMPARE). Subsequent fields specify which registers supply the source operands and the destination register where the result is stored. The CPU interprets the information in the instruction fields and performs the requested operation. It would then move on to the next instruction in memory, executing the stored program step-by-step. The goal of this chapter is to discuss the details of what operations we want the CPU to perform, how many registers we should have, and so on.

Of course, some values in memory are not instructions! They might be binary data representing numeric values, strings of characters, and so on. The CPU will read these values into its temporary registers when it needs to operate on them and write newly computed values back into memory.

Mr. Blue is asking a good question: how do we know which words in memory are instructions and which are data? After all, they're both binary values! The answer is that we can't tell by looking at the values — it's how they are used by the CPU that distinguishes instructions from data. If a value is loaded into the datapath, it's being used as data. If a value is loaded by the control logic, it's being used as an instruction.

Anatomy of a von Neumann Computer



So this is the digital system we'll build to perform computations. We'll start with a datapath that contains some number of registers to hold data values. We'll be able to select which registers will supply operands for the arithmetic and logic unit that will perform an operation. The ALU produces a result and other status signals. The ALU result can be written back to one of the registers for later use. We'll provide the datapath with means to move data to and from main memory.

There will be a control unit that provides the necessary control signals to the datapath. In the example datapath shown here, the control unit would provide ASEL and BSEL to select two register values as operands and DEST to select the register where the ALU result will be written. If the datapath had, say, 32 internal registers, ASEL, BSEL and DEST would be 5-bit values, each specifying a particular register number in the range 0 to 31. The control unit also provides the FN function code that controls the operation performed by the ALU. The ALU we designed in Part 1 of the course requires a 6-bit function code to select between a variety of arithmetic, boolean and shift operations.

The control unit would load values from main memory to be interpreted as instructions. The control unit contains a register, called the “program counter”, that keeps track of the address in main memory of the next instruction to be executed. The control unit also contains a (hopefully small) amount of logic to translate the instruction fields into the necessary control signals. Note the control unit receives status signals from the datapath that will enable programs to execute different sequences of instructions if, for example, a particular data value was zero.

The datapath serves as the brawn of our digital system and is responsible for storing and manipulating data values. The control unit serves as the brain of our system, interpreting the program stored in main memory and generating the necessary sequence of control signals for the datapath.

Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
 - An operation or **opcode** to be performed
 - Source **operands** and **destination** for the result
- In a von Neumann machine, instructions are executed sequentially
 - CPU logically implements this loop:
 - By default, the next PC is current PC + size of current instruction unless the instruction says otherwise



Instructions are the fundamental unit of work. They're fetched by the control unit and executed one after another in the order they are fetched. Each instruction specifies the operation to be performed along with the registers to supply the source operands and destination register where the result will be stored.

In a von Neumann machine, instruction execution involves the steps shown here: the instruction is loaded from the memory location whose address is specified by the program counter. When the requested data is returned by the memory, the instruction fields are converted to the appropriate control signals for the datapath, selecting the source operands from the specified registers, directing the ALU to perform the specified operation, and storing the result in the specified destination register. The final step in executing an instruction is updating the value of the program counter to be the address of the next instruction.

This execution loop is performed again and again. Modern machines can execute up more than a billion instructions per second!

Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
 - Functional definition of **operations** and **storage locations**
 - **Precise description** of how software can invoke and access them
- The ISA is a new layer of abstraction:
 - ISA specifies **what** the hardware provides, **not how** it's implemented
 - Hides the complexity of CPU implementation
 - Enables fast innovation in hardware (no need to change software!)
 - 8086 (1978): 29 thousand transistors, 5 MHz, 0.33 MIPS
 - Pentium 4 (2003): 44 million transistors, 4 GHz, ~5000 MIPS
 - Both implement x86 ISA
 - Dark side: Commercially successful ISAs last for decades
 - Today's x86 CPUs carry baggage of design decisions from the 70's

The discussion so far has been a bit abstract. Now it's time to roll up our sleeves and figure out what instructions we want our system to support. The specification of instruction fields and their meaning along with the details of the datapath design are collectively called the instruction set architecture (ISA) of the system. The ISA is a detailed functional specification of the operations and storage mechanisms and serves as a contract between the designers of the digital hardware and the programmers who will write the programs. Since the programs are stored in main memory and can hence be changed, we'll call them software, to distinguish them from the digital logic which, once implemented, doesn't change. It's the combination of hardware and software that determine the behavior of our system.

The ISA is a new layer of abstraction: we can write programs for the system without knowing the implementation details of the hardware. As hardware technology improves we can build faster systems without having to change the software. You can see here that over a fifteen year timespan, the hardware for executing the Intel x86 instruction set went from executing 300,000 instructions per second to executing 5 billion instructions per second. Same software as before, we've just taken advantage of smaller and faster MOSFETs to build more complex circuits and faster execution engines.

But a word of caution is in order! It's tempting to make choices in the ISA that reflect the constraints of current technologies, *e.g.*, the number of internal registers, the width of the operands, or the maximum size of main memory. But it will be hard to change the ISA when technology improves since there's a powerful economic incentive to ensure that old software can run on new machines, which means that a particular ISA can live for decades and span many generations of technology. If your ISA is successful, you'll have to live with any bad choices you made for a very long time.

Instruction Set Architecture Design

- Designing an ISA is hard:
 - How many operations?
 - What types of storage, how much?
 - How to encode instructions?
 - How to future-proof?
- How to decide? Take a **quantitative approach**
 - Take a set of representative benchmark programs
 - Evaluate versions of your ISA and implementation with and without feature
 - Pick what works best overall (performance, energy, area...)
- Corollary: **Optimize the common case**

Let's design our own instruction set: the Beta!

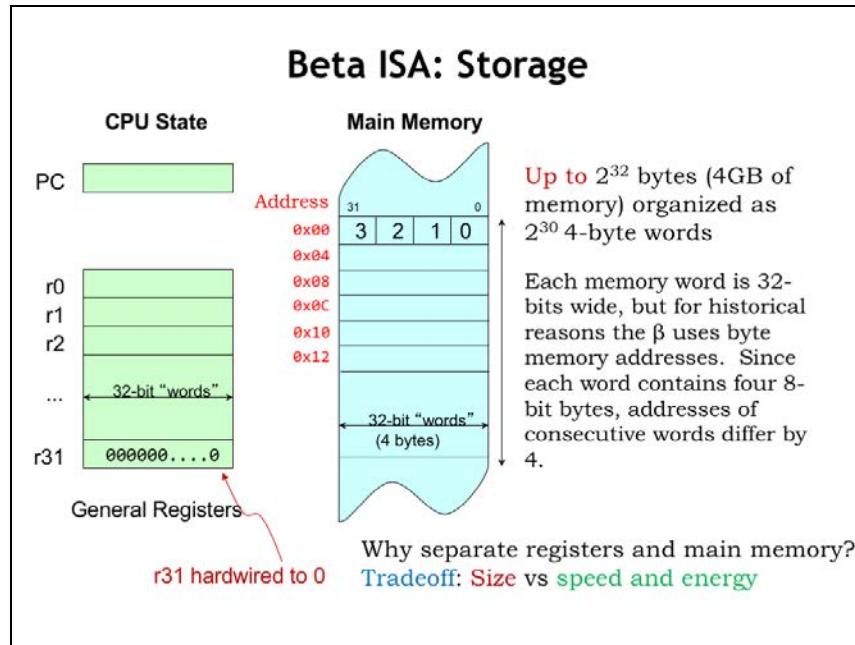
Designing an ISA is hard! What are the operations that should be supported? How many internal registers? How much main memory? Should we design the instruction encoding to minimize program size or to keep the logic in the control unit as simple as possible? Looking into our crystal ball, what can we say about the computation and storage capabilities of future technologies?

We'll answer these questions by taking a quantitative approach. First we'll choose a set of benchmark programs, chosen as representative of the many types of programs we expect to run on our system. So some of benchmark programs will perform scientific and engineering computations, some will manipulate large data sets or perform database operations, some will require specialized computations for graphics or communications, and so on. Happily, after many decades of computer use, several standardized benchmark suites are available for us to use.

We'll then implement the benchmark programs using our instruction set and simulate their execution on our proposed datapath. We'll evaluate the results to measure how well the system performs. But what do we mean by "well"? That's where it gets interesting: "well" could refer to execution speed, energy consumption, circuit size, system cost, etc. If you're designing a smart watch, you'll make different choices than if you're designing a high-performance graphics card or a data-center server.

Whatever metric you choose to evaluate your proposed system, there's an important design principle we can follow: identify the common operations and focus on them as you optimize your design. For example, in general-purpose computing, almost all programs spend a lot of their time on simple arithmetic operations and accessing values in main memory. So those operations should be made as fast and energy efficient as possible.

Now, let's get to work designing our own instruction set and execution engine, a system we'll call the Beta.



The Beta is an example of a reduced-instruction-set computer (RISC) architecture. “Reduced” refers to the fact that in the Beta ISA, most instructions only access the internal registers for their operands and destination. Memory values are loaded and stored using separate memory-access instructions, which implement only a simple address calculation. These reductions lead to smaller, higher-performance hardware implementations and simpler compilers on the software side. The ARM and MIPS ISAs are other examples of RISC architectures. Intel’s x86 ISA is more complex.

There is a limited amount of storage inside of the CPU — using the language of sequential logic, we’ll refer to this as the CPU state. There’s a 32-bit program counter (PC for short) that holds the address of the current instruction in main memory. And there are thirty-two registers, numbered 0 through 31. Each register holds a 32-bit value. We’ll use use 5-bit fields in the instruction to specify the number of the register to be used an operand or destination. As shorthand, we’ll refer to a register using the prefix “R” followed by its number, *e.g.*, “R0” refers to the register selected by the 5-bit field 0b00000.

Register 31 (R31) is special — its value always reads as 0 and writes to R31 have no affect on its value.

The number of bits in each register and hence the number of bits supported by ALU operations is a fundamental parameter of the ISA. The Beta is a 32-bit architecture. Many modern computers are 64-bit architectures, meaning they have 64-bit registers and a 64-bit datapath.

Main memory is an array of 32-bit words. Each word contains four 8-bit bytes. The bytes are numbered 0 through 3, with byte 0 corresponding to the low-order 7 bits of the 32-bit value, and so on. The Beta ISA only supports word accesses, either loading or storing full 32-bit words. Most “real” computers also support accesses to bytes and half-words.

Even though the Beta only accesses full words, following a convention used by many ISAs it uses byte addresses. Since there are 4 bytes in each word, consecutive words in memory have addresses that differ by 4. So the first word in memory has address 0, the second word address 4, and so on. You can see the addresses to left of each memory location in the diagram shown here. Note that we’ll usually use hexadecimal notation when specifying addresses and other binary values — the “0x” prefix

indicates when a number is in hex. When drawing a memory diagram, we'll follow the convention that addresses increase as you read from top to bottom.

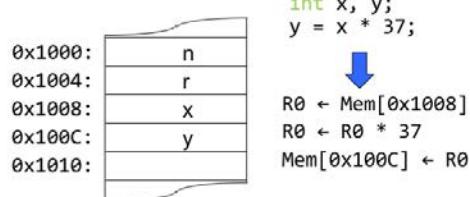
The Beta ISA supports 32-bit byte addressing, so an address fits exactly into one 32-bit register or memory location. The maximum memory size is 2^{32} bytes or 2^{30} words — that's 4 gigabytes (4 GB) or one billion words of main memory. Some Beta implementations might actually have a smaller main memory, *i.e.*, one with fewer than 1 billion locations.

Why have separate registers and main memory? Well, modern programs and datasets are very large, so we'll want to have a large main memory to hold everything. But large memories are slow and usually only support access to one location at a time, so they don't make good storage for use in each instruction which needs to access several operands and store a result. If we used only one large storage array, then an instruction would need to have three 32-bit addresses to specify the two source operands and destination — each instruction encoding would be huge! And the required memory accesses would have to be one-after-the-other, really slowing down instruction execution.

On the other hand, if we use registers to hold the operands and serve as the destination, we can design the register hardware for parallel access and make it very fast. To keep the speed up we won't be able to have very many registers — a classic size-vs-speed performance tradeoff we see in digital systems all the time. In the end, the tradeoff leading to the best performance is to have a small number of very fast registers used by most instructions and a large but slow main memory. So that's what the BETA ISA does.

Storage Conventions

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
 - Load them
 - Compute on them
 - Store the results



In general, all program data will reside in main memory. Each variable used by the program “lives” in a specific main memory location and so has a specific memory address. For example, in the diagram below, the value of variable “x” is stored in memory location 0x1008, and the value of “y” is stored in memory location 0x100C, and so on.

To perform a computation, *e.g.*, to compute $x * 37$ and store the result in y , we would have to first load the value of x into a register, say, $R0$. Then we would have the datapath multiply the value in $R0$ by 37, storing the result back into $R0$. Here we’ve assumed that the constant 37 is somehow available to the datapath and doesn’t itself need to be loaded from memory. Finally, we would write the updated value in $R0$ back into memory at the location for y .

Whew! A lot of steps... Of course, we could avoid all the loading and storing if we chose to keep the values for x and y in registers. Since there are only 32 registers, we can’t do this for all of our variables, but maybe we could arrange to load x and y into registers, do all the required computations involving x and y by referring to those registers, and then, when we’re done, store changes to x and y back into memory for later use. Optimizing performance by keeping often-used values in registers is a favorite trick of programmers and compiler writers.

So the basic program template is some loads to bring values into the registers, followed by computation, followed by any necessary stores. ISAs that use this template are usually referred to as “load-store architectures”.

Beta ISA: Instructions

- Three types of instructions:
 - Arithmetic and logical: Perform operations on general registers
 - Loads and stores: Move data between general registers and main memory
 - Branches: Conditionally change the program counter
- All instructions have a fixed length: 32 bits (4 bytes)
 - Tradeoff (vs variable-length instructions):
 - Simpler decoding logic, next PC is easy to compute
 - Larger code size

Having talked about the storage resources provided by the Beta ISA, let's design the Beta instructions themselves. This might be a good time to print a copy of the handout called the "Summary of Beta Instruction Formats" so you'll have it for handy reference.

The Beta has three types of instructions: compute instructions that perform arithmetic and logic operations on register values, load and store instructions that access values in main memory, and branch instructions that change the value of the program counter.

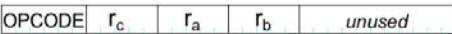
We'll discuss each class of instructions in turn.

In the Beta ISA, all the instruction encodings are the same size: each instruction is encoded in 32 bits and hence occupies exactly one 32-bit word in main memory. This instruction encoding leads to simpler control-unit logic for decoding instructions. And computing the next value of the program counter is very simple: for most instructions, the next instruction can be found in the following memory location. We just need to add 4 to the current value of program counter to advance to the next instruction.

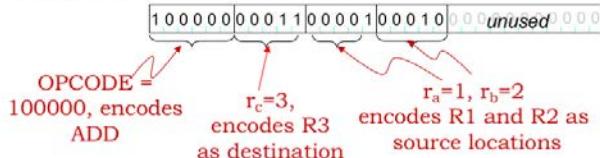
As we saw in Part 1 of the course, fixed-length encodings are often inefficient in the sense that the same information content (in this case, the encoded program) can be encoded using fewer bits. To do better we would need a variable-length encoding for instructions, where frequently-occurring instructions would use a shorter encoding. But hardware to decode variable-length instructions is complex since there may be several instructions packed into one memory word, while other instructions might require loading several memory words. The details can be worked out, but there's a performance and energy cost associated with the more efficient encoding.

Nowadays, advances in memory technology have made memory size less of an issue and the focus is on the higher-performance needed by today's applications. Our choice of a fixed-length encoding leads to larger code size, but keeps the hardware execution engine small and fast.

Beta ALU Instructions

Format: 

Example coded instruction: ADD



32-bit hex: 0x80611000

We prefer to write a **symbolic representation**: ADD(r_1, r_2, r_3)

ADD(r_a, r_b, r_c):

$$\text{Reg}[r_c] \leftarrow \text{Reg}[r_a] + \text{Reg}[r_b]$$

"Add the contents of r_a to the contents of r_b ; store the result in r_c "

Similar instructions for other ALU operations:

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR, XNOR
shift: SHL, SHR, SRA

The computation performed by the Beta datapath happens in the arithmetic-and-logic unit (ALU). We'll be using the ALU designed in Part 1 of the course.

The Beta ALU instructions have 4 instruction fields. There's a 6-bit field specifying the ALU operation to be performed — this field is called the opcode. The two source operands come from registers whose numbers are specified by the 5-bit "ra" and "rb" fields. So we can specify any register from R0 to R31 as a source operand. The destination register is specified by the 5-bit "rc" field.

This instruction format uses 21 bits of the 32-bit word, the remaining bits are unused and should be set to 0. The diagram shows how the fields are positioned in the 32-bit word. The choice of position for each field is somewhat arbitrary, but to keep the hardware simple, when we can we'll want to use the same field positions for similar fields in the other instruction encodings. For example, the opcode will always be found in bits [31:26] of the instruction.

Here's the binary encoding of an ADD instruction. The opcode for ADD is the 6-bit binary value 0b100000 — you can find the binary for each opcode in the Opcode Table in the handout mentioned before. The "rc" field specifies that the result of the ADD will be written into R3. And the "ra" and "rb" fields specify that the first and second source operands are R1 and R2 respectively. So this instruction adds the 32-bit values found in R1 and R2, writing the 32-bit sum into R3.

Note that it's permissible to refer to a particular register several times in the same instruction. So, for example, we could specify R1 as the register for both source operands AND also as the destination register. If we did, we'd be adding R1 to R1 and writing the result back into R1, which would effectively multiply the value in R1 by 2.

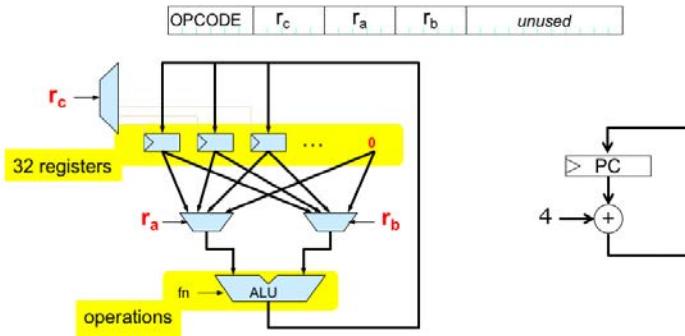
Since it's tedious and error-prone to transcribe 32-bit binary values, we'll often use hexadecimal notation for the binary representation of an instruction. In this example, the hexadecimal notation for the encoded instruction is 0x80611000. However, it's **much** easier if we describe the instructions using a functional notation, *e.g.*, "ADD(r_1, r_2, r_3)". Here we use a symbolic name for each operation, called a mnemonic. For this instruction the mnemonic is "ADD", followed by a parenthesized list of operands, in this case the two source operands (r_1 and r_2), then the destination (r_3). So we'll

understand that ADD(ra,rb,rc) is shorthand for asking the Beta to compute the sum of the values in registers ra and rb, writing the result as the new value of register rc.

Here's the list of the mnemonics for all the operations supported by the Beta. There is a detailed description of what each instruction does in the Beta Documentation handout. Note that all these instructions use same 4-field template, differing only in the value of the opcode field. This first step was pretty straightforward — we simply provided instruction encodings for the basic operations provided by the ALU.

Implementation Sketch #1

Now that we have our first set of instructions, we can create a more concrete implementation sketch:



Now that we have our first group of instructions, we can create a more concrete implementation sketch.

Here we see our proposed datapath. The 5-bit “ra” and “rb” fields from the instruction are used to select which of the 32 registers will be used for the two operands. Note that register 31 isn’t actually a read/write register, it’s just the 32-bit constant 0, so that selecting R31 as an operand results in using the value 0. The 5-bit “rc” field from the instruction selects which register will be written with the result from the ALU. Not shown is the hardware needed to translate the instruction opcode to the appropriate ALU function code — perhaps a 64-location ROM could be used to perform the translation by table lookup.

The program counter logic supports simple sequential execution of instructions. It’s a 32-bit register whose value is updated at the end of each instruction by adding 4 to its current value. This means the next instruction will come from the memory location following the one that holds the current instruction.

In this diagram we see one of the benefits of a RISC architecture: there’s not much logic needed to decode the instruction to produce the signals needed to control the datapath. In fact, many of the instruction fields are used as-is!

Should We Support Constant Operands?

Many programs use small constants frequently

e.g., our factorial example: 0, 1, -1

Tradeoff:

When used, they save registers and instructions

More opcodes → more complex control logic and datapath

Analyzing operands when running SPEC CPU benchmarks, we find that constant operands appear in

- >50% of executed arithmetic instructions
 - Loop increments, scaling indices
- >80% of executed compare instructions
 - Loop termination condition
- >25% of executed load instructions
 - Offsets into data structures

ISA designers receive many requests for what are affectionately known as “features” — additional instructions that, in theory, will make the ISA better in some way. Dealing with such requests is the moment to apply our quantitative approach in order to be able to judge the tradeoffs between cost and benefits.

Our first “feature request” is to allow small constants as the second operand in ALU instructions. So if we replaced the 5-bit “rb” field, we would have room in the instruction to include a 16-bit constant as bits [15:0] of the instruction. The argument in favor of this request is that small constants appear frequently in many programs and it would make programs shorter if we didn’t have to use load operations to read constant values from main memory. The argument against the request is that we would need additional control and datapath logic to implement the feature, increasing the hardware cost and probably decreasing the performance.

So our strategy is to modify our benchmark programs to use the ISA augmented with this feature and measure the impact on a simulated execution. Looking at the results, we find that there is compelling evidence that small constants are indeed very common as the second operands to many operations. Note that we’re not so much interested in simply looking at the code. Instead we want to look at what instructions actually get executed while running the benchmark programs. This will take into account that instructions executed during each iteration of a loop might get executed 1000’s of times even though they only appear in the program once.

Looking at the results, we see that over half of the arithmetic instructions have a small constant as their second operand.

Comparisons involve small constants 80% of the time. This probably reflects the fact that during execution comparisons are used in determining whether we’ve reached the end of a loop.

And small constants are often found in address calculations done by load and store operations.

Operations involving constant operands are clearly a common case, one well worth optimizing. Adding support for small constant operands to the ISA resulted in programs that were measurably smaller and

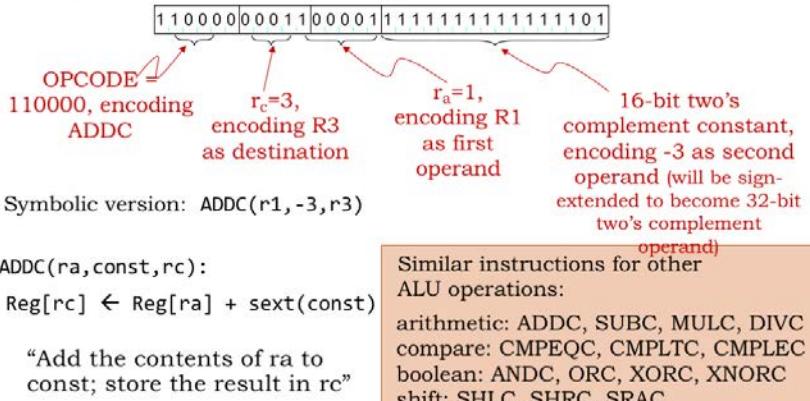
faster. So: feature request approved!

Beta ALU Instructions with Constant

Format:

OPCODE	r_c	r_a	16-bit signed constant
--------	-------	-------	------------------------

Example instruction: ADDC adds register contents and constant:



Here we see the second of the two Beta instruction formats. It's a modification of the first format where we've replaced the 5-bit "rb" field with a 16-bit field holding a constant in two's complement format. This will allow us to represent constant operands in the range of 0x8000 (decimal -32768) to 0x7FFF (decimal 32767).

Here's an example of the add-constant (ADDC) instruction which adds the contents of R1 and the constant -3, writing the result into R3. We can see that the second operand in the symbolic representation is now a constant (or, more generally, an expression that can evaluated to get a constant value).

One technical detail needs discussion: the instruction contains a 16-bit constant, but the datapath requires a 32-bit operand. How does the datapath hardware go about converting from, say, the 16-bit representation of -3 to the 32-bit representation of -3?

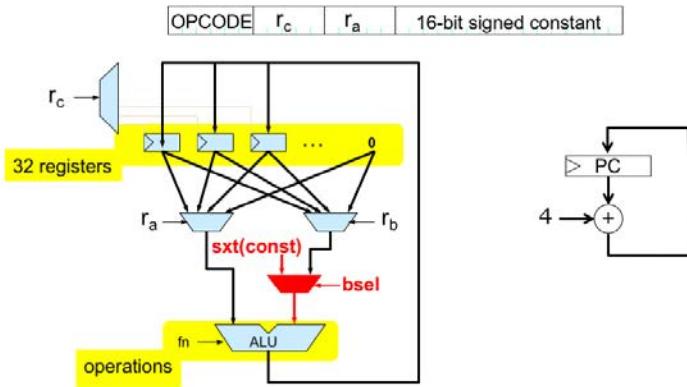
Comparing the 16-bit and 32-bit representations for various constants, we see that if the 16-bit two's complement constant is negative (*i.e.*, its high-order bit is 1), the high sixteen bits of the equivalent 32-bit constant are all 1's. And if the 16-bit constant is non-negative (*i.e.*, its high-order bit is 0), the high sixteen bits of the 32-bit constant are all 0's. Thus the operation the hardware needs to perform is "sign extension" where the sign-bit of the 16-bit constant is replicated sixteen times to form the high half of the 32-constant. The low half of the 32-bit constant is simply the 16-bit constant from the instruction. No additional logic gates will be needed to implement sign extension — we can do it all with wiring.

Here are the fourteen ALU instructions in their "with constant" form, showing the same instruction mnemonics but with a "C" suffix indicate the second operand is a constant. Since these are additional instructions, these have different opcodes than the original ALU instructions.

Finally, note that if we need a constant operand whose representation does NOT fit into 16 bits, then we have to store the constant as a 32-bit value in a main memory location and load it into a register for use just like we would any variable value.

Implementation Sketch #2

Next we add the datapath hardware to support small constants as the second ALU operand:



To give some sense for the additional datapath hardware that will be needed, let's update our implementation sketch to add support for constants as the second ALU operand. We don't have to add much hardware: just a multiplexer which selects either the "rb" register value or the sign-extended constant from the 16-bit field in the instruction. The BSEL control signal that controls the multiplexer is 1 for the ALU-with-constant instructions and 0 for the regular ALU instructions.

We'll put the hardware implementation details aside for now and revisit them in a few lectures.

Beta Load and Store Instructions

Loads and stores move data between the internal registers and main memory

OPCODE	r _c	r _a	16-bit signed constant
--------	----------------	----------------	------------------------

address

Address calculation
is just like ADDC
instruction!

LD(ra, const, rc) Reg[rc] \leftarrow Mem[Reg[ra] + sext(const)]

Load rc with the contents of the memory location

ST(rc, const, ra) Mem[Reg[ra] + sext(const)] \leftarrow Reg[rc]

Store the contents of rc into the memory location

To access memory the CPU has to generate an address. LD and ST compute the address by adding the sign-extended constant to the contents of register ra.

- To access a constant address, specify R31 as ra.
- To use only a register value as the address, specify a constant of 0.

Now let's turn our attention to the second class of instructions: load (LD) and store (ST), which allow the CPU to access values in memory. Note that since the Beta is a load-store architecture these instructions are the **only** mechanism for accessing memory values.

The LD and ST instructions use the same instruction template as the ALU-with-constant instructions. To access memory, we'll need a memory address, which is computed by adding the value of the "ra" register to the sign-extended 16-bit constant from the low-order 16 bits of the instruction. This computation is exactly the one performed by the ADDC instruction — so we'll reuse that hardware — and the sum is sent to main memory as the byte address of the location to be accessed. For the LD instruction, the data returned by main memory is written to the "rc" register.

The store instruction (ST) performs the same address calculation as LD, then reads the data value from the "rc" register and sends both to main memory. The ST instruction is special in several ways: it's the only instruction that needs to read the value of the "rc" register, so we'll need to adjust the datapath hardware slightly to accommodate that need. And since "rc" is serving as a source operand, it appears as the first operand in the symbolic form of the instruction, followed by "const" and "ra" which are specifying the destination address. ST is the only instruction that does **not** write a result into the register file at end of the instruction.

Using LD and ST

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
 - Load them
 - Compute on them
 - Store the results

0x1000:	n
0x1004:	r
0x1008:	x
0x100C:	y
0x1010:	

```
int x, y;  
y = x * 37;  
  
R0 ← Mem[0x1008]  
R0 ← R0 * 37  
Mem[0x100C] ← R0  
  
LD(R31,0x1008,R0)  
MULC(R0,37,R0)  
ST(R0,0x100C,R31)
```

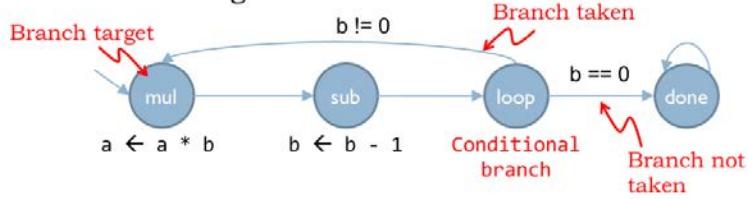
Here's the example we saw earlier, where we needed to load the value of the variable x from memory, multiply it by 37 and write the result back to the memory location that holds the value of the variable y.

Now that we have actual Beta instructions, we've expressed the computation as a sequence of three instructions. To access the value of variable x, the LD instruction adds the contents of R31 to the constant 0x1008, which sums to 0x1008, the address we need to access. The ST instruction specifies a similar address calculation to write into the location for the variable y.

The address calculation performed by LD and ST works well when the locations we need to access have addresses that fit into the 16-bit constant field. What happens when we need to access locations at addresses higher than 0xFFFF? Then we need to treat those addresses as we would any large constant, and store those large addresses in main memory so they can be loaded into a register to be used by LD and ST. Okay, but what if the number of large constants we need to store is greater than will fit in low memory, *i.e.*, the addresses we can access directly? To solve this problem, the Beta includes a “load relative” (LDR) instruction, which we'll see in the lecture on the Beta implementation.

Can We Solve Factorial With ALU Instructions?

- No! Recall high-level FSM:



- Factorial needs to **loop**
- So far we can only encode sequences of operations on registers
- Need a way to change the PC based on data values!
 - Called “branching”. If the branch is taken, the PC is changed. If the branch is not taken, keep executing sequentially.

Finally, let's discuss the third class of instructions that let us change the program counter. Up until now, the program counter has simply been incremented by 4 at the end of each instruction, so that the next instruction comes from the memory location that immediately follows the location that held the current instruction, *i.e.*, the Beta has been executing instructions sequentially from memory.

But in many programs, such as in factorial, we need to disrupt sequential execution, either to loop back to repeat some earlier instruction, or to skip over instructions because of some data dependency. We need a way to change the program counter based on data values generated by the program's execution. In the factorial example, as long as b is not equal to 0, we need to keep executing the instructions that calculate $a*b$ and decrement b . So we need instructions to test the value of b after it's been decremented and if it's non-zero, change the PC to repeat the loop one more time.

Changing the PC depending on some condition is implemented by a branch instruction, and the operation is referred to as a “conditional branch”. When the branch is taken, the PC is changed and execution is restarted at the new location, which is called the branch target. If the branch is not taken, the PC is incremented by 4 and execution continues with the instruction following the branch.

As the name implies, a branch instruction represents a potential fork in the execution sequence. We'll use branches to implement many different types of control structures: loops, conditionals, procedure calls, etc.

Beta Branch Instructions

The Beta's *branch instructions* provide a way to conditionally change the PC to point to a nearby location...

... and, optionally, remembering (in Rc) where we came from (useful for procedure calls).

BEQ or BNE r_c r_a 16-bit signed constant

“offset” is a SIGNED CONSTANT encoded as part of the instruction!

BEQ(ra,offset,rc): Branch if equal	BNE(ra,offset,rc): Branch if not equal
NPC \leftarrow PC + 4	NPC \leftarrow PC + 4
Reg[rc] \leftarrow NPC	Reg[rc] \leftarrow NPC
if (Reg[ra] == 0)	if (Reg[ra] != 0)
PC \leftarrow NPC + 4*offset	PC \leftarrow NPC + 4*offset
else	else
PC \leftarrow NPC	PC \leftarrow NPC

offset = distance in words to branch target, counting from the instruction following the BEQ/BNE. Range: -32768 to +32767.

Branch instructions also use the instruction format with the 16-bit signed constant. The operation of the branch instructions are a bit complicated, so let's walk through their operation step-by-step.

Let's start by looking at the operation of the BEQ instruction. First the usual PC+4 calculation is performed, giving us the address of the instruction following the BEQ. This value is written to the “rc” register whether or not the branch is taken. This feature of branches is pretty handy and we'll use it to implement procedure calls a couple of lectures from now. Note that if we don't need to remember the PC+4 value, we can specify R31 as the “rc” register.

Next, BEQ tests the value of the “ra” register to see if it's equal to 0. If it is equal to 0, the branch is taken and the PC is incremented by the amount specified in the constant field of the instruction. Actually the constant, called an offset since we're using it to offset the PC, is treated as a word offset and is multiplied by 4 to convert it a byte offset since the PC uses byte addressing. If the contents of the “ra” register is not equal to 0, the PC is incremented by 4 and execution continues with the instruction following the BEQ.

Let me say a few more words about the offset. The branches are using what's referred to as “pc-relative addressing”. That means the address of the branch target is specified relative to the address of the branch, or, actually, relative to the address of the instruction following the branch. So an offset of 0 would refer to the instruction following the branch and an offset of -1 would refer to the branch itself. Negative offsets are called “backwards branches” and are usually seen at branches used at the end of loops, where the looping condition is tested and we branch backwards to the beginning of the loop if another iteration is called for. Positive offsets are called “forward branches” and are usually seen in code for “if statements”, where we might skip over some part of the program if a condition is not true.

We can use BEQ to implement a so-called unconditional branch, *i.e.*, a branch that is always taken. If we test R31 to see if it's 0, that's always true, so BEQ(R31,...) would always branch to the specified target.

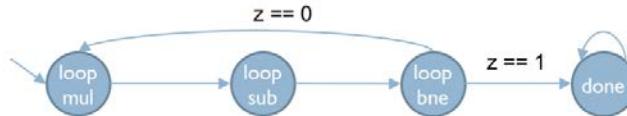
There's also a BNE instruction, identical to BEQ in its operation except the sense of the condition is reversed: the branch is taken if the value of register “ra” is non-zero.

It might seem that only testing for zero/non-zero doesn't let us do everything we might want to do. For example, how would we branch if "a < b"? That's where the compare instructions come in — they do more complicated comparisons, producing a non-zero value if the comparison is true and a zero value if the comparison is false. Then we can use BEQ and BNE to test the result of the comparison and branch appropriately.

Can We Solve Factorial Now?

```
int a = 1;                                // Assume r1 = N
int b = N;                                 ADDC(r31, 1, r0) // r0 = 1
do {                                         L:MUL(r0, r1, r0) // r0 = r0 * r1
    a = a * b;                            SUBC(r1, 1, r1) // r1 = r1 - 1
    b = b - 1;                           BNE(r1, L, r31) // if r1 != 0, run MUL next
} while (b != 0)                           // at this point, r0 = N!
```

- Remember control FSM for our simple programmable datapath?



- Control FSM states → instructions!
 - Not the case in general
 - Happens here because datapath is similar to basic von Neumann datapath

At long last we're finally in a position to write Beta code to compute factorial using the iterative algorithm shown in C code on the left. In the Beta code, the loop starts at the second instruction and is marked with the "L:" label. The body of the loop consists of the required multiplication and the decrement of b. Then, in the fourth instruction, b is tested and, if it's non-zero, the BNE will branch back to the instruction with the label L.

Note that in our symbolic notation for BEQ and BNE instructions we don't write the offset directly since that would be a pain to calculate and would change if we added or removed instructions from the loop. Instead we reference the instruction to which we want to branch, and the program that translates the symbolic code into the binary instruction fields will do the offset calculation for us.

There's a satisfying similarity between the Beta code and the operations specified by the high-level FSM we created for computing factorial in the simple programmable datapath discussed earlier in this lecture. In this example, each state in the high-level FSM matches up nicely with a particular Beta instruction. We wouldn't expect that high degree of correspondence in general, but since our Beta datapath and the example datapath were very similar, the states and instructions match up pretty well.

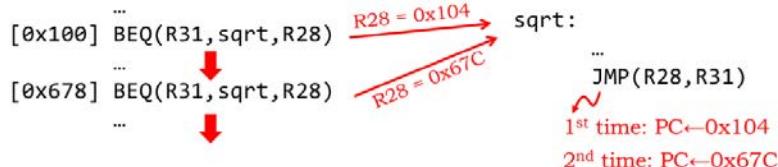
Beta JMP Instruction

Branches transfer control to some predetermined destination specified by a constant in the instruction. It will be useful to be able to transfer control to a computed address.



JMP(Ra,Rc): $\text{Reg}[Rc] \leftarrow PC + 4$
 $PC \leftarrow \text{Reg}[Ra]$

Useful for procedure call return...



Finally, our last instruction! Branches conditionally transfer control to a specific target instruction. But we'll also need the ability to compute the address of the target instruction — that ability is provided by the JMP instruction which simply sets the program counter to value from register "ra". Like branches, JMP will write the PC+4 value into to the specified destination register.

This capability is very useful for implementing procedures in Beta code. Suppose we have a procedure "sqrt" that computes the square root of its argument, which is passed in, say, R0. We don't show the code for sqrt on the right, except for the last instruction, which is a JMP.

On the left we see that the programmer wants to call the sqrt procedure from two different places in his program. Let's watch what happens...

The first call to the sqrt procedure is implemented by the unconditional branch at location 0x100 in main memory. The branch target is the first instruction of the sqrt procedure, so execution continues there. The BEQ also writes the address of the following instruction (0x104) into its destination register, R28. When we reach the end of first procedure call, the JMP instruction loads the value in R28, which is 0x104, into the PC, so execution continues with the instruction following the first BEQ. So we've managed to return from the procedure and continue execution where we left off in the main program.

When we get to the second call to the sqrt procedure, the sequence of events is the same as before except that this time R28 contains 0x67C, the address of the instruction following the second BEQ. So the second time we reach the end of the sqrt procedure, the JMP sets the PC to 0x67C and execution resumes with the instruction following the second procedure call.

Neat! The BEQs and JMP have worked together to implement procedure call and return. We'll discuss the implementation of procedures in detail in an upcoming lecture.

Beta ISA Summary

- Storage:
 - Processor: 32 registers (r31 hardwired to 0) and PC
 - Main memory: 32-bit byte addresses; each memory access involves a 32-bit word. Since there are 4 bytes/word, all addresses will be a multiple of 4.



- Instruction formats:



- Instruction types:

- ALU: Two input registers, or register and constant
- Loads and stores
- Branches, Jumps

That wraps up the design of the Beta instruction set architecture. In summary, the Beta has 32 registers to hold values that can be used as operands for the ALU. All other values, along with the binary representation of the program itself, are stored in main memory. The Beta supports 32-bit memory addresses and can access values in $2^{32} = 4$ gigabytes of main memory. All Beta memory access refers to 32-bit words, so all addresses will be a multiple of 4 since there are 4 bytes/word.

There are two instruction formats. The first specifies an opcode, two source registers and a destination register. The second replaces the second source register with a 32-bit constant, derived by sign-extending a 16-bit constant stored in the instruction itself.

There are three classes of instructions: ALU operations, LD and ST for accessing main memory, and branches and JMPs that change the order of execution.

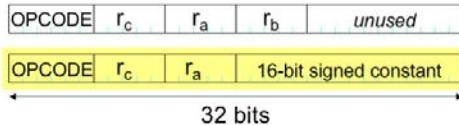
And that's it! As we'll see in the next lecture, we'll be able to parlay this relatively simple repertoire of operations into a system that can execute any computation we can specify.

10a: Assembly Language

1. Beta ISA Summary
2. Programming Languages
3. Assembly Language
4. Example UASM Source File
5. How Does It Get Assembled?
6. Registers are Predefined Symbols
7. Labels and Offsets
8. Mighty Macroinstructions
9. Assembly of Instructions
10. Example Assembly
11. UASM Macros for Beta Instructions
12. Pseudoinstructions
13. Factorial with Pseudoinstructions
14. Raw Data
15. UASM Expressions and Layout
16. Summary: Assembly Language

Beta ISA Summary

- Storage:
 - Processor: 32 registers (r31 hardwired to 0) and PC
 - Main memory: Up to 4 GB, 32-bit words, 32-bit byte addresses, 4-byte-aligned accesses



- Instruction formats:
- Instruction classes:
 - ALU: Two input registers, or register and constant
 - Loads and stores: access memory
 - Branches, Jumps: change program counter

In the previous lecture we developed the instruction set architecture for the Beta, the computer system we'll be building throughout this part of the course. The Beta incorporates two types of storage or memory. In the CPU datapath there are 32 general-purpose registers, which can be read to supply source operands for the ALU or written with the ALU result. In the CPU's control logic there is a special-purpose register called the program counter, which contains the address of the memory location holding the next instruction to be executed.

The datapath and control logic are connected to a large main memory with a maximum capacity of 2^{32} bytes, organized as 2^{30} 32-bit words. This memory holds both data and instructions.

Beta instructions are 32-bit values comprised of various fields. The 6-bit OPCODE field specifies the operation to be performed. The 5-bit Ra, Rb, and Rc fields contain register numbers, specifying one of the 32 general-purpose registers. There are two instruction formats: one specifying an opcode and three registers, the other specifying an opcode, two registers, and a 16-bit signed constant.

There are three classes of instructions. The ALU instructions perform an arithmetic or logic operation on two operands, producing a result that is stored in the destination register. The operands are either two values from the general-purpose registers, or one register value and a constant. The yellow highlighting indicates instructions that use the second instruction format.

The Load/Store instructions access main memory, either loading a value from main memory into a register, or storing a register value to main memory.

And, finally, there are branches and jumps whose execution may change the program counter and hence the address of the next instruction to be executed.

Programming Languages

32-bit (4-byte) ADD instruction:

The diagram shows a 32-bit instruction word divided into five fields: opcode, rc, ra, rb, and (unused). The opcode field is 5 bits long, rc is 2 bits, ra is 5 bits, rb is 5 bits, and the remaining 11 bits are labeled as (unused).

Means, to the BETA, Reg[4] \leftarrow Reg[2] + Reg[3]

We'd rather write in *assembly language*:

ADD(R2, R3, R4)

or better yet a *high-level language*:

a = b + c;

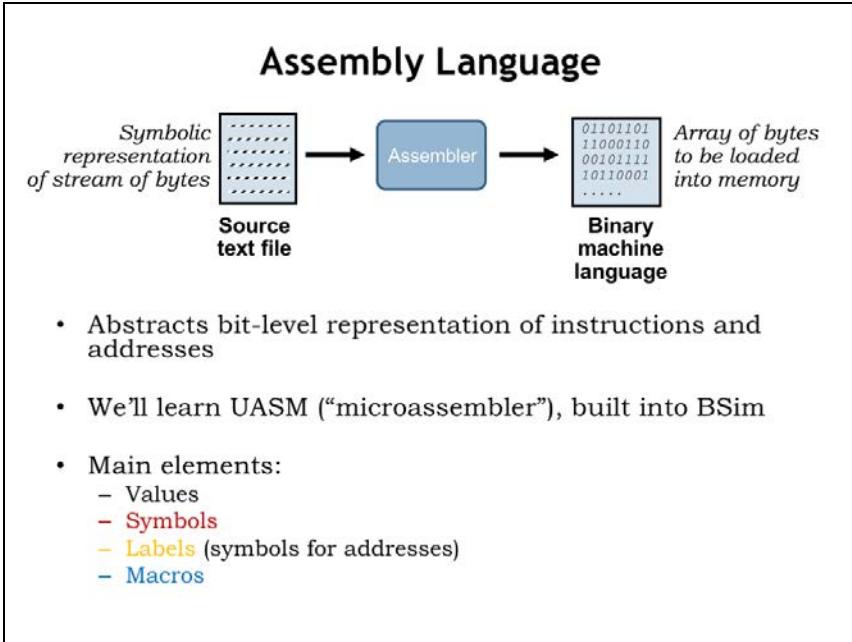
Coming up

To program the Beta we'll need to load main memory with binary-encoded instructions. Figuring out each encoding is clearly the job for a computer, so we'll create a simple programming language that will let us specify the opcode and operands for each instruction. So instead of writing the binary at the top of slide, we'll write assembly language statements to specify instructions in symbolic form. Of course we still have think about which registers to use for which values and write sequences of instructions for more complex operations.

By using a high-level language we can move up one more level abstraction and describe the computation we want in terms of variables and mathematical operations rather than registers and ALU functions.

In this lecture we'll describe the assembly language we'll use for programming the Beta. And in the next lecture we'll figure out how to translate high-level languages, such as C, into assembly language.

The layer cake of abstractions gets taller yet: we could write an interpreter for say, Python, in C and then write our application programs in Python. Nowadays, programmers often choose the programming language that's most suitable for expressing their computations, then, after perhaps many layers of translation, come up with a sequence of instructions that the Beta can actually execute.



Okay, back to assembly language, which we'll use to shield ourselves from the bit-level representations of instructions and from having to know the exact location of variables and instructions in memory. A program called the "assembler" reads a text file containing the assembly language program and produces an array of 32-bit words that can be used to initialize main memory.

We'll learn the UASM assembly language, which is built into BSim, our simulator for the Beta ISA. UASM is really just a fancy calculator! It reads arithmetic expressions and evaluates them to produce 8-bit values, which it then adds sequentially to the array of bytes which will eventually be loaded into the Beta's memory. UASM supports several useful language features that make it easier to write assembly language programs. Symbols and labels let us give names to particular values and addresses. And macros let us create shorthand notations for sequences of expressions that, when evaluated, will generate the binary representations for instructions and data.

Example UASM Source File

```
N = 12          // loop index initial value
ADDC(r31, N, r1) // r1 = loop index
ADDC(r31, 1, r0) // r0 = accumulated product
loop: MUL(r0, r1, r0) // r0 = r0 * r1
      SUBC(r1, 1, r1) /* r1 = r1 - 1 */
      BNE(r1, loop, r31) // if r1 != 0, NextPC=loop
```

- **Comments** after //, ignored by assembler (also /*...*/)
- **Symbols** are symbolic representations of a constant value (they are NOT variables!)
- **Labels** are symbols for addresses
- **Macros** expand into sequences of bytes
 - Most frequently, macros are instructions
 - We can use them for other purposes

Here's an example UASM source file. Typically we write one UASM statement on each line and can use spaces, tabs and newlines to make the source as readable as possible. We've added some color coding to help in our explanation.

Comments (shown in green) allow us to add text annotations to the program. Good comments will help remind you how your program works. You really don't want to have figure out from scratch what a section of code does each time you need to modify or debug it! There are two ways to add comments to the code. “//” starts a comment, which then occupies the rest of the source line. Any characters after “//” are ignored by the assembler, which will start processing statements again at the start of the next line in the source file. You can also enclose comment text using the delimiters “/*” and “*/” and the assembler will ignore everything in-between. Using this second type of comment, you can “comment-out” many lines of code by placing “/*” at the start and, many lines later, end the comment section with “*/”.

Symbols (shown in red) are symbolic names for constant values. Symbols make the code easier to understand, *e.g.*, we can use N as the name for an initial value for some computation, in this case the value 12. Subsequent statements can refer to this value using the symbol N instead of entering the value 12 directly. When reading the program, we'll know that N means this particular initial value. So if later we want to change the initial value, we only have to change the definition of the symbol N rather than find all the 12's in our program and change them. In fact some of the other appearances of 12 might not refer to this initial value and so to be sure we only changed the ones that did, we'd have to read and understand the whole program to make sure we only edited the right 12's. You can imagine how error-prone that might be! So using symbols is a practice you want to follow!

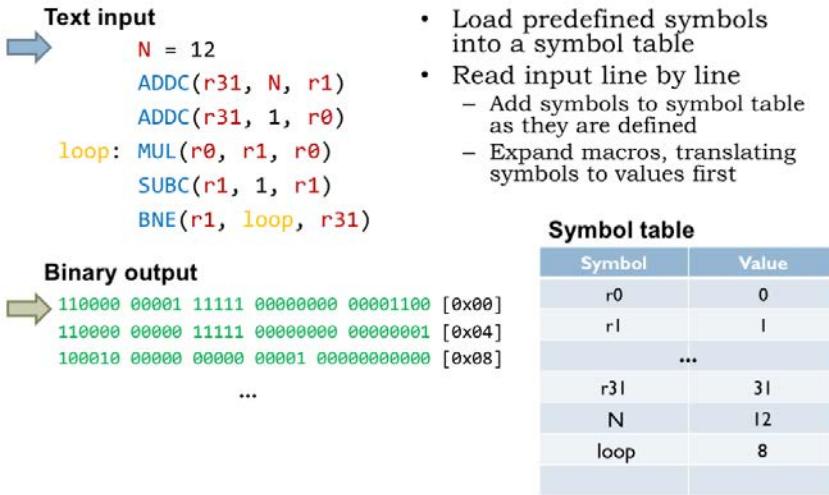
Note that all the register names are shown in red. We'll define the symbols R0 through R31 to have the values 0 through 31. Then we'll use those symbols to help us understand which instruction operands are intended to be registers, *e.g.*, by writing R1, and which operands are numeric values, *e.g.*, by writing the number 1. We could just use numbers everywhere, but the code would be much harder to read and understand.

Labels (shown in yellow) are symbols whose value are the address of a particular location in the program. Here, the label “loop” will be our name for the location of the MUL instruction in memory. In the BNE at the end of the code, we use the label “loop” to specify the MUL instruction as the branch target. So if R1 is non-zero, we want to branch back to the MUL instruction and start another iteration.

We’ll use indentation for most UASM statements to make it easy to spot the labels defined by the program. Indentation isn’t required, it’s just another habit assembly language programmers use to keep their programs readable.

We use macro invocations (shown in blue) when we want to write Beta instructions. When the assembler encounters a macro, it “expands” the macro, replacing it with a string of text provided by in the macro’s definition. During expansion, the provided arguments are textually inserted into the expanded text at locations specified in the macro definition. Think of a macro as shorthand for a longer text string we could have typed in. We’ll see how all this works in the next video segment.

How Does It Get Assembled?



Let's follow along as the assembler processes our source file. The assembler maintains a symbol table that maps symbols names to their numeric values. Initially the symbol table is loaded with mappings for all the register symbols.

The assembler reads the source file line-by-line, defining symbols and labels, expanding macros, or evaluating expressions to generate bytes for the output array. Whenever the assembler encounters a use of a symbol or label, it's replaced by the corresponding numeric value found in the symbol table.

The first line, $N = 12$, defines the value of the symbol N to be 12, so the appropriate entry is made in the symbol table.

Advancing to the next line, the assembler encounters an invocation of the `ADDC` macro with the arguments “`r31`”, “`N`”, and “`r1`”. As we'll see in a couple of slides, this triggers a series of nested macro expansions that eventually lead to generating a 32-bit binary value to be placed in memory location 0. The 32-bit value is formatted here to show the instruction fields and the destination address is shown in brackets.

The next instruction is processed in the same way, generating a second 32-bit word.

On the fourth line, the label `loop` is defined to have the value of the location in memory that's about to be filled (in this case, location 8). So the appropriate entry is made in the symbol table and the `MUL` macro is expanded into the 32-bit word to be placed in location 8.

The assembler processes the file line-by-line until it reaches the end of the file. Actually the assembler makes two passes through the file. On the first pass it loads the symbol table with the values from all the symbol and label definitions. Then, on the second pass, it generates the binary output. The two-pass approach allows a statement to refer to symbol or label that is defined later in the file, *e.g.*, a forward branch instruction could refer to the label for an instruction later in the program.

Registers are Predefined Symbols

- $r0 = 0, \dots, r31 = 31$
- Treated like normal symbols:
 $\text{ADDC}(r31, N, r1)$
↓
Substitute symbols with their values
 $\text{ADDC}(31, 12, 1)$
↓
Expand macro
 $110000\ 00001\ 11111\ 00000000\ 00001100$
- No “type checking” if you use the wrong opcode...
 $\text{ADDC}(r31, r12, r1)$
↓
 $\text{ADDC}(31, 12, 1)$
 $\text{Reg}[1] \leftarrow \text{Reg}[31] + 12$
 $\text{ADD}(r31, N, r1)$
↓
 $\text{ADD}(31, 12, 1)$
 $\text{Reg}[1] \leftarrow \text{Reg}[31] + \text{Reg}[12]$

As we saw in the previous slide, there's nothing magic about the register symbols — they are just symbolic names for the values 0 through 31. So when processing ADDC(r31,N,r1), UASM replaces the symbols with their values and actually expands ADDC(31,12,1).

UASM is very simple. It simply replaces symbols with their values, expands macros and evaluates expressions. So if you use a register symbol where a numeric value is expected, the value of the symbol is used as the numeric constant. Probably not what the programmer intended.

Similarly, if you use a symbol or expression where a register number is expected, the low-order 5 bits of the value is used as the register number, in this example, as the Rb register number. Again probably not what the programmer intended.

The moral of the story is that when writing UASM assembly language programs, you have to keep your wits about you and recognize that the interpretation of an operand is determined by the opcode macro, not by the way you wrote the operand.

Labels and Offsets

Input file

```
N = 12
ADDc(r31, N, r1)
ADDc(r31, 1, r0)
loop: MUL(r0, r1, r0)
      SUBc(r1, 1, r1)
      BNE(r1, loop, r31)
```

Output file

```
110000 00001 11111 00000000 00001100 [0x00]
110000 00000 11111 00000000 00000001 [0x04]
100010 00000 00001 00000 0000000000 [0x08]
110001 00001 00001 00000000 00000001 [0x0C]
loop: 011101 11111 00001 11111111 11111101 [0x10]
```

offset = (label - <addr of BNE/BEQ>)/4 - 1
 $= (8 - 16)/4 - 1 = -3$

- Label value is the address of a memory location
- BEQ/BNE macros compute offset automatically
- Labels hide addresses!

Symbol table

Symbol	Value
r0	0
r1	1
...	...
r31	31
N	12
loop	8

Recall from Lecture 9 that branch instructions use the 16-bit constant field of the instruction to encode the address of the branch target as a word offset from the location of the branch instruction. Well, actually the offset is calculated from the instruction immediately following the branch, so an offset of -1 would refer to the branch itself.

The calculation of the offset is a bit tedious to do by hand and would, of course, change if we added or removed instructions between the branch instruction and branch target. Happily macros for the branch instructions incorporate the necessary formula to compute the offset from the address of the branch and the address of the branch target. So we just specify the address of the branch target, usually with a label, and let UASM do the heavy lifting.

Here we see that BNE branches backwards by three instructions (remember to count from the instruction following the branch) so the offset is -3. The 16-bit two's complement representation of -3 is the value placed in the constant field of the BNE instruction.

Mighty Macroinstructions

Macros are parameterized abbreviations, or shorthand

```
// Macro to generate 4 consecutive bytes:  
.macro consec(n) n n+1 n+2 n+3  
  
// Invocation of above macro:  
consec(37)
```

Is expanded to

$\Rightarrow 37\ 37+1\ 37+2\ 37+3 \Rightarrow 37\ 38\ 39\ 40$

Here are macros for breaking multi-byte data types into byte-sized chunks

```
// Assemble into bytes, little-endian:  
.macro WORD(x) x%256 (x/256)%256  
.macro LONG(x) WORD(x) WORD(x >> 16)  
  
. = 0x100  
LONG(0xdeadbeef)
```

Has same effect as:

Mem: 0xef 0xbe 0xad 0xde

*Boy, that's hard to read.
Maybe, those big-endian
types do have a point.*



Let's take a closer look at how macros work in UASM. Here we see the definition of the macro "consec" which has a single parameter "n". The body of the macro is a sequence of four expressions. When there's an invocation of the "consec" macro, in this example with the argument 37, the body of the macro is expanded replacing all occurrences of "n" with the argument 37. The resulting text is then processed as if it had appeared in place of the macro invocation. In this example, the four expressions are evaluated to give a sequence of four values that will be placed in the next four bytes of the output array.

Macro expansions may contain other macro invocations, which themselves will be expanded, continuing until all that's left are expressions to be evaluated. Here we see the macro definition for WORD, which assembles its argument into two consecutive bytes. And for the macro LONG, which assembles its argument into four consecutive bytes, using the WORD macro to process the low 16 bits of the value, then the high 16 bits of the value.

These two UASM statements cause the constant 0xDEADBEEF to be converted to 4 bytes, which are deposited in the output array starting at index 0x100.

Note that the Beta expects the least-significant byte of a multi-byte value to be stored at the lowest byte address. So the least-significant byte 0xEF is placed at address 0x100 and the most-significant byte 0xDE is placed at address 0x103. This is the "little-endian" convention for multi-byte values: the least-significant byte comes first. Intel's x86 architecture is also little-endian.

There is a symmetrical "big-endian" convention where the most-significant byte comes first. Both conventions are in active use and, in fact, some ISAs can be configured to use either convention! There's no "right answer" for which convention to use, but the fact that there are two conventions means that we have to be alert for the need to convert the representation of multi-byte values when moving values between one ISA and another, e.g., when we send a data file to another user.

As you can imagine there are strong advocates for both schemes who are happy to defend their point of view at great length. Given the heat of the discussion, it's appropriate that the names for the

conventions were drawn from Jonathan Swift's "Gulliver's Travels" in which a civil war is fought over whether to open a soft-boiled egg at its big end or its little end.

Assembly of Instructions

OPCODE	RC	RA	RB	UNUSED	
110000	00000	01111	1000000000000000	00000	

```

// Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+(RC%32)<<21)+((RA%32)<<16)+(RB%32)<<11)
}

// Assemble Beta opc instructions
.macro betaoopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+(RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000)
}

// Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL) betaoopc(OP,RA,((LABEL-(.+4))>>2),RC)

For example:
.macro ADDC(RA,C,RC) betaoopc(0x30,RA,C,RC)

ADDC(R15, -32768, R0) --> betaoopc(0x30,15,-32768,0)

```

Let's look at the macros used to assemble Beta instructions. The BETAOP helper macro supports the 3-register instruction format, taking as arguments the values to be placed in the OPCODE, Ra, Rb, and Rc fields. The ".align 4" directive is a bit of administrative bookkeeping to ensure that instructions will have a byte address that's a multiple of 4, *i.e.*, that they span exactly one 32-bit word in memory. That's followed by an invocation of the LONG macro to generate the 4 bytes of binary data representing the value of the expression shown here. The expression is where the actual assembly of the fields takes place. Each field is limited to requisite number of bits using the modulo operator (%), then shifted left (<<) to the correct position in the 32-bit word.

And here are the helper macros for the instructions that use a 16-bit constant as the second operand.

Let's follow the assembly of an ADDC instruction to see how this works. The ADDC macro expands into an invocation of the BETAOPC helper macro, passing along the correct value for the ADDC opcode, along with the three operands.

The BETAOPC macro does the following arithmetic: The OP argument, in this case the value 0x30, is shifted left to occupy the high-order 6 bits of the instruction. Then the RA argument, in this case 15, is placed in its proper location. The 16-bit constant -32768 is positioned in the low 16 bits of the instruction. And, finally, the Rc argument, in this case 0, is positioned in the Rc field of the instruction.

You can see why we call this processing "assembling an instruction". The binary representation of an instruction is assembled from the binary values for each of the instruction fields. It's not a complicated process, but it requires a lot of shifting and masking, tasks that we're happy to let a computer handle.

Example Assembly

```
ADDC (R3 , 1234 , R17)
↓ expand ADDC macro with RA=R3, C=1234, RC=R17
betaopc (0x30 , R3 , 1234 , R17)
↓ expand betaopc macro with OP=0x30, RA=R3, CC=1234, RC=R17
.align 4
LONG ( (0x30<<26) + ( (R17%32)<<21) + ( (R3%32)<<16) + (1234 % 0x10000) )
↓ expand LONG macro with X=0xC22304D2
WORD (0xC22304D2) WORD (0xC22304D2 >> 16)
↓ expand first WORD macro with X=0xC22304D2
0xC22304D2%256 (0xC22304D2/256)%256 WORD (0xC223)
↓ evaluate expressions, expand second WORD macro with X=0xC223
0xD2 0x04 0xC223%256 (0xC223/256)%256
↓ evaluate expressions
0xD2 0x04 0x23 0xC2
```

Here's the entire sequence of macro expansions that assemble this ADDC instruction into an appropriate 32-bit binary value in main memory.

You can see that the knowledge of Beta instruction formats and opcode values is built into the bodies of the macro definitions. The UASM processing is actually quite general — with a different set of macro definitions it could process assembly language programs for almost any ISA!

UASM Macros for Beta Instructions

(defined in beta.uasm)

```
| BETA Instructions:  
.macro ADD(RA,RB,RC)    betaop(0x20,RA,RB,RC)  
.macro ADDC(RA,C,RC)    betaopc(0x30,RA,C,RC)  
.macro AND(RA,RB,RC)    betaop(0x28,RA,RB,RC)  
.macro ANDC(RA,C,RC)    betaopc(0x38,RA,C,RC)  
.macro MUL(RA,RB,RC)    betaop(0x22,RA,RB,RC)  
.macro MULC(RA,C,RC)    betaopc(0x32,RA,C,RC)  
  
.  
.macro LD(RA,CC,RC)    betaopc(0x18,RA,CC,RC)  
.macro LD(CC,RC)    betaopc(0x18,R31,CC,RC)  
.macro ST(RC,CC,RA)    betaopc(0x19,RA,CC,RC)  
.macro ST(RC,CC)    betaopc(0x19,R31,CC,RC)  
  
.  
.macro BEQ(RA,LABEL,RC) betabr(0x1C,RA,RC,LABEL)  
.macro BEQ(RA,LABEL)    betabr(0x1C,RA,r31,LABEL)  
.macro BNE(RA,LABEL,RC) betabr(0x1D,RA,RC,LABEL)  
.macro BNE(RA,LABEL)    betabr(0x1D,RA,r31,LABEL)
```

Convenience
macros so we
don't have to
specify R31...

All the macro definitions for the Beta ISA are provided in the beta.uasm file, which is included in each of the assembly language lab assignments. Note that we include some convenience macros to define shorthand representations that provide common default values for certain operands. For example, except for procedure calls, we don't care about the PC+4 value saved in the destination register by branch instructions, so almost always would specify R31 as the Rc register, effectively discarding the PC+4 value saved by branches. So we define two-argument branch macros that automatically provide R31 as the destination register. Saves some typing, and, more importantly, it makes it easier to understand the assembly language program.

Pseudoinstructions

- Convenience macros that expand to one or more real instructions
- Extend set of operations without adding instructions to the ISA

```
// Convenience macros so we don't have to use R31
.macro LD(CC,RC)      LD(R31,CC,RC)
.macro ST(RA,CC)       ST(RA,CC,R31)
.macro BEQ(RA,LABEL)   BEQ(RA,LABEL,R31)
.macro BNE(RA,LABEL)   BNE(RA,LABEL,R31)

.macro MOVE(RA,RC)     ADD(RA,R31,RC)    // Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)   ADDC(R31,C,RC)   // Reg[RC] <- C
.macro COM(RA,RC)      XORC(RA,-1,RC)  // Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)      SUB(R31,RB,RC)   // Reg[RC] <- -Reg[RB]
.macro NOP()           ADD(R31,R31,R31) // do nothing

.macro BR(LABEL)        BEQ(R31,LABEL)  // always branch
.macro BR(LABEL,RC)    BEQ(R31,LABEL,RC) // always branch
.macro CALL(LABEL)      BEQ(R31,LABEL,LP) // call subroutine
.macro BF(RA,LABEL,RC)  BEQ(RA,LABEL,RC) // 0 is false
.macro BF(RA,LABEL)    BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)  BNE(RA,LABEL,RC) // 1 is true
.macro BT(RA,LABEL)    BNE(RA,LABEL)

// Multi-instruction sequences
.macro PUSH(RA)         ADDC(SP,4,SP)  ST(RA,-4,SP)
.macro POP(RA)           LD(SP,-4,RA)   ADDC(SP,-4,SP)
```

Here are a whole set of convenience macros intended to make programs more readable. For example, unconditional branches can be written using the BR() macro rather than the more cumbersome BEQ(R31,...). And it's more readable to use branch-false (BF) and branch-true (BT) macros when testing the results of a compare instruction.

And note the PUSH and POP macros at the bottom of page. These expand into multi-instruction sequences, in this case to add and remove values from a stack data structure pointed to by the SP register.

We call these macros “pseudo instructions” since they let us provide the programmer with what appears a larger instruction set, although underneath the covers we’ve just using the same small instruction repertoire developed in Lecture 9.

Factorial with Pseudoinstructions

Before	After
<pre>N = 12 ADDC(r31, N, r1) ADDC(r31, 1, r0) loop: MUL(r0, r1, r0) SUBC(r1, 1, r1) BNE(r1, loop, r31)</pre>	<pre>N = 12 CMOVE(N, r1) CMOVE(1, r0) loop: MUL(r0, r1, r0) SUBC(r1, 1, r1) BNE(r1, loop)</pre>

In this example we've rewritten the original code we had for the factorial computation using pseudo instructions. For example, CMOVE is a pseudo instruction for moving small constants into a register. It's easier for us to read and understand the intent of a "constant move" operation than an "add a value to 0" operation provided by the ADDC expansion of CMOVE. Anything we can do to remove the cognitive clutter will be very beneficial in the long run.

Raw Data

- LONG assembles a 32-bit value

- Variables

- Constants > 16 bits

N: LONG(12)
factN: LONG(0xdeadbeef)

...

Start:

LD(N, r1) CMOVE(1, r0) loop: MUL(r0, r1, r0) SUBC(r1, 1, r1) BT(r1, loop) ST(r0, factN)	LD(r31, N, r1) LD(31, 0, 1) Reg[1] ← Mem[Reg[31] + 0] ← Mem[0] ← 12
--	---

Symbol table

Symbol	Value
...	
N	0
factN	4

So far we've talked about assembling instructions. What about data? How do we allocate and initialize data storage and how do we get those values into registers so that they can be used as operands?

Here we see a program that allocates and initializes two memory locations using the LONG macro. We've used labels to remember the addresses of these locations for later reference.

When the program is assembled the values of the label N and factN are 0 and 4 respectively, the addresses of the memory locations holding the two data values.

To access the first data value, the program uses a LD instruction, in this case one of convenience macros that supplies R31 as the default value of the Ra field. The assembler replaces the reference to the label N with its value 0 from the symbol table. When the LD is executed, it computes the memory address by adding the constant (0) to the value of the Ra register (which is R31 and hence the value is 0) to get the address (0) of the memory location from which to fetch the value to be placed in R1.

UASM Expressions and Layout

- Values can be written as expressions
 - Assembler evaluates expressions, they are *not* translated to instructions to compute the value!
- The “.” (period) symbol means the next byte address to be filled
 - Can read or write to it
 - Useful to control data layout or leave empty space (e.g., for arrays)

```
. = 0x100          // Assemble into 0x100
LONG(0xdeadbeef)
k = .
LONG(0x000dec0de)
. = .+16          // Skip 16 bytes
LONG(0xc0ffeeee)
```

The constants needed as values for data words and instruction fields can be written as expressions. These expressions are evaluated by the assembler as it assembles the program and the resulting value is used as needed. Note that the expressions are evaluated at the time the assembler runs. By the time the program runs on the Beta, the resulting value is used. The assembler does NOT generate ADD and MUL instructions to compute the value during program execution. If a value is needed for an instruction field or initial data value, the assembler has to be able to perform the arithmetic itself. If you need the program to compute a value during execution, you have to write the necessary instructions as part of your program.

One last UASM feature: there's a special symbol “.”, called “dot”, whose value is the address of the next main memory location to be filled by the assembler when it generates binary data. Initially “.” is 0 and it's incremented each time a new byte value is generated.

We can set the value of “.” to tell the assembler where in memory we wish to place a value. In this example, the constant 0xDEADBEEF is placed into location 0x100 of main memory. And we can use “.” in expressions to compute the values for other symbols, as shown here when defining the value for the symbol “k”. In fact, the label definition “k.” is exactly equivalent to the UASM statement “k = .”

We can even increment the value of “.” to skip over locations, e.g., if we wanted to leave space for an un initialized array.

Summary: Assembly Language

- Low-level language, symbolic representation of sequence of bytes. Abstracts:
 - Bit-level representation of instructions
 - Addresses
- Elements: Values, **symbols**, **labels**, **macros**
- Values can be constants or expressions
- **Symbols** are symbolic representations of values
- **Labels** are symbols for addresses
- **Macros** are expanded to byte sequences:
 - Instructions
 - Pseudoinstructions (translate to 1+ real instructions)
 - Raw data
- Can control where to assemble with “.” symbol

And that's assembly language! We use assembly language as a convenient notation for generating the binary encoding for instructions and data. We let the assembler build the bit-level representations we need and to keep track of the addresses where these values are stored in main memory.

UASM itself provides support for values, symbols, labels and macros.

Values can be written as constants or expressions involving constants.

We use symbols to give meaningful names to values so that our programs will be more readable and more easily modified. Similarly, we use labels to give meaningful names to addresses in main memory and then use the labels in referring to data locations in LD or ST instructions, or to instruction locations in branch instructions.

Macros hide the details of how instructions are assembled from their component fields.

And we can use “.” to control where the assembler places values in main memory.

The assembler is itself a program that runs on our computer. That raises an interesting “chicken and egg problem”: how did the first assembler program get assembled into binary so it could run on a computer? Well, it was hand-assembled into binary. I suspect it processed a very simple language indeed, with the bells and whistles of symbols, labels, macros, expression evaluation, etc. added only after basic instructions could be assembled by the program. And I'm sure they were very careful not to lose the binary so they wouldn't have to do the hand-assembly a second time!

10b: Models of Computation

1. Universality?
2. Models of Computation
3. FSM Limitations
4. Turing Machines
5. Other Models of Computation
6. Other Models of Computation
7. Turing Machines Galore!
8. The Universal Function
9. Universality
10. Turing Universality
11. Coded Algorithms: Key to CS
12. Uncomputability!
13. Why f_{H} is Uncomputable

Universality?

- Recall: We say a set of Boolean gates is universal if we can implement any Boolean function using only gates from that set.
- What problems can we solve with a von Neumann computer? (e.g., the Beta)
 - Everything that FSMs can solve?
 - Every problem?
 - Does it depend on the ISA?
- Needed: a mathematical model of computation
 - Prove what can be computed, what can't

An interesting question for computer architects is what capabilities must be included in the ISA? When we studied Boolean gates in Part 1 of the course, we were able to prove that NAND were universal, *i.e.*, that we could implement any Boolean function using only circuits constructed from NAND gates.

We can ask the corresponding question of our ISA: is it universal, *i.e.*, can it be used to perform any computation? what problems can we solve with a von Neumann computer? Can the Beta solve any problem FSMs can solve? Are there problems FSMs can't solve? If so, can the Beta solve those problems? Do the answers to these questions depend on the particular ISA?

To provide some answers, we need a mathematical model of computation. Reasoning about the model, we should be able to prove what can be computed and what can't. And hopefully we can ensure that the Beta ISA has the functionality needed to perform any computation.

Models of Computation

The roots of computer science stem from the evaluation of many alternative mathematical “models” of computation to determine the classes of computations each could represent.

An elusive goal was to find a universal model, capable of representing *all* practical computations...

- switches
- gates
- combinational logic
- memories
- FSMs

*We've got FSMs...
what else do we need?*



Are FSMs the ultimate digital computing device?

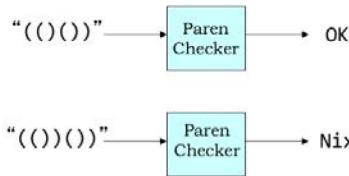
The roots of computer science stem from the evaluation of many alternative mathematical models of computation to determine the classes of computation each could represent. An elusive goal was to find a universal model, capable of representing **all** realizable computations. In other words if a computation could be described using some other well-formed model, we should also be able to describe the same computation using the universal model.

One candidate model might be finite state machines (FSMs), which can be built using sequential logic. Using Boolean logic and state transition diagrams we can reason about how an FSM will operate on any given input, predicting the output with 100% certainty.

Are FSMs the universal digital computing device? In other words, can we come up with FSM implementations that implement all computations that can be solved by any digital device?

FSM Limitations

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM. For instance:



Well-formed Parentheses Checker:

Given any string of coded left & right parens, outputs 1 if it is balanced, else 0.

Simple, easy to describe.

Can this problem be solved using an FSM??? **NO!**

PROBLEM: Requires *arbitrarily* many states, depending on input. Must "COUNT" unmatched left parens. An FSM can only keep track of a finite number of unmatched parens: for every FSM, we can find a string it can't check.



*I know how
to fix that!*

Alan Turing

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM. For example, can we build an FSM to determine if a string of parentheses (properly encoded into a binary sequence) is well-formed? A parenthesis string is well-formed if the parentheses balance, *i.e.*, for every open parenthesis there is a matching close parenthesis later in the string. In the example shown here, the input string on the top is well-formed, but the input string on the bottom is not. After processing the input string, the FSM would output a 1 if the string is well-formed, 0 otherwise.

Can this problem be solved using an FSM? No, it can't. The difficulty is that the FSM uses its internal state to encode what it knows about the history of the inputs. In the paren checker, the FSM would need to count the number of unbalanced open parens seen so far, so it can determine if future input contains the required number of close parens. But in a finite state machine there are only a fixed number of states, so a particular FSM has a maximum count it can reach. If we feed the FSM an input with more open parens than it has the states to count, it won't be able to check if the input string is well-formed.

The "finite-ness" of FSMs limits their ability to solve problems that require unbounded counting. Hmm, what other models of computation might we consider? Mathematics to the rescue, in this case in the form of a British mathematician named Alan Turing.

Turing Machines

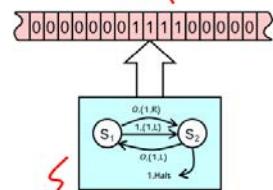
Alan Turing was one of a group of researchers studying alternative models of computation.

He proposed a conceptual model consisting of an FSM combined with an infinite digital tape that could be read and written at each step.

- encode input as symbols on tape
- FSM reads tape/writes symbols/ changes state until it halts
- Answer encoded on tape

Turing's model (like others of the time) solves the "FINITE" problem of FSMs.

Bounded tape configuration can be expressed as a (large!) integer



*We can talk about TM 347 running on input 51, producing an answer of 42.
TMs as integer functions:
 $y = TM_1[x]$*

In the early 1930's Alan Turing was one of many mathematicians studying the limits of proof and computation. He proposed a conceptual model consisting of an FSM combined with a infinite digital tape that could read and written under the control of the FSM. The inputs to some computation would be encoded as symbols on the tape, then the FSM would read the tape, changing its state as it performed the computation, then write the answer onto the tape and finally halting. Nowadays, this model is called a Turing Machine (TM). Turing Machines, like other models of the time, solved the "finite" problem of FSMs.

So how does all this relate to computation? Assuming the non-blank input on the tape occupies a finite number of adjacent cells, it can be expressed as a large integer. Just construct a binary number using the bit encoding of the symbols from the tape, alternating between symbols to the left of the tape head and symbols to the right of the tape head. Eventually all the symbols will be incorporated into the (very large) integer representation.

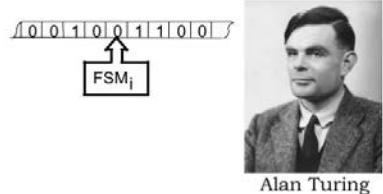
So both the input and output of the TM can be thought of as large integers, and the TM itself as implementing an integer function that maps input integers to output integers.

The FSM brain of the Turing Machine can be characterized by its truth table. And we can systematically enumerate all the possible FSM truth tables, assigning an index to each truth table as it appears in the enumeration. Note that indices get very large very quickly since they essentially incorporate all the information in the truth table. Fortunately we have a very large supply of integers!

We'll use the index for a TM's FSM to identify the TM as well. So we can talk about TM 347 running on input 51, producing the answer 42.

Other Models of Computation...

Turing Machines [Turing]



Lambda calculus [Church, Curry, Rosser...]



$$\lambda x. \lambda y. xxy$$

`(lambda (x) (lambda (y) (x (x y))))`

Alonzo
Church

Recursive Functions [Kleene]

$$F(0, x) = x$$

$$F(1+y, x) = 1+F(y, x)$$

```
(define (fact n)
  (... (fact (- n 1)))
```



Stephen
Kleene

Production Systems [Post, Markov]



$$\alpha \rightarrow \beta$$

IF pulse=0 THEN
patient=dead

Emile Post

There are many other models of computation, each of which describes a class of integer functions where a computation is performed on an integer input to produce an integer answer. Kleene, Post and Turing were all students of Alonzo Church at Princeton University in the mid-1930's. They explored many other formulations for modeling computation: recursive functions, rule-based systems for string rewriting, and the lambda calculus. They were all particularly intrigued with proving the existence of problems unsolvable by realizable machines. Which, of course, meant characterizing the problems that could be solved by realizable machines.

Computability

FACT: Each model studied is capable of computing exactly the same set of integer functions!

Proof Technique:
Constructions that translate between models

BIG IDEA:
Computability, independent of computation scheme chosen



Church's Thesis:

Every discrete function computable by ANY realizable machine is computable by some Turing machine.

$$f(x) \text{ computable} \Leftrightarrow \text{for some } k, \text{ all } x \\ f(x) = T_k[x]$$



*unproved, but
universally
accepted...*

It turned out that each model was capable of computing *exactly* the same set of integer functions! This was proved by coming up with constructions that translated the steps in a computation between the various models. It was possible to show that if a computation could be described by one model, an equivalent description exists in the other model. This lead to a notion of computability that was independent of the computation scheme chosen. This notion is formalized by Church's Thesis, which says that every discrete function computable by any realizable machine is computable by some Turing Machine. So if we say the function $f(x)$ is computable, that's equivalent to saying that there's a TM that given x as an input on its tape will write $f(x)$ as an output on the tape and halt.

As yet there's no proof of Church's Thesis, but it's universally accepted that it's true. In general "computable" is taken to mean "computable by some TM".

If you're curious about the existence of uncomputable functions, please see the optional video at the end of this lecture.

meanwhile...

Turing machines Galore!

"special-purpose"
Turing Machines....

$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$

Multiplication

$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$

Sorting

$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$

Factorization

$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$

Primality Test



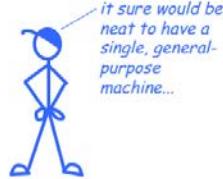
*Is there an alternative to
infinitely many ad-hoc Turing
Machines?*

Okay, we've decided that Turing Machines can model any realizable computation. In other words for every computation we want to perform, there's a (different) Turing Machine that will do the job. But how does this help us design a general-purpose computer? Or are there some computations that will require a special-purpose machine no matter what?

The Universal Function

Here's an interesting function to explore: the Universal function, U , defined by

$$U(k, j) = T_k[j]$$



Could this be computable???

SURPRISE! U is computable by a Turing Machine:

$$\begin{array}{ccc} k \rightarrow & T_U & \rightarrow T_k[j] \\ j \rightarrow & & \end{array}$$

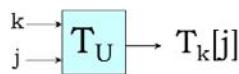
In fact, there are infinitely many such machines. Each is capable of performing *any* computation that can be performed by *any* TM!

What we'd like to find is a universal function U : it would take two arguments, k and j , and then compute the result of running T_k on input j . Is U computable, *i.e.*, is there a universal Turing Machine T_U ? If so, then instead of many ad-hoc TMs, we could just use T_U to compute the results for any computable function.

Surprise! U is computable and T_U exists. In fact there are infinitely many universal TMs, some quite simple - the smallest known universal TM has 4 states uses 6 tape symbols. A universal machine is capable of performing any computation that can be performed by any TM!

Universality

What's going on here?



k encodes a “program” – a description of some arbitrary machine.

j encodes the input data to be used.

T_U *interprets* the program, emulating its processing of the data!

KEY IDEA: Interpretation.

Manipulate *coded representations* of computing machines, rather than the machines themselves.

What's going on here? k encodes a “program” - a description of some arbitrary TM that performs a particular computation. j encodes the input data on which to perform that computation. T_U “interprets” the program, emulating the steps T_k will take to process the input and write out the answer. The notion of interpreting a coded representation of a computation is a key idea and forms the basis for our stored program computer.

Turing Universality

The *Universal Turing Machine* is the paradigm for modern general-purpose computers!

Basic threshold test: Is your computer *Turing Universal*?

- If so, it can emulate every other Turing machine!
- Thus, your computer can compute any computable function

To show your computer is Universal: demonstrate that it can emulate some known UTM.

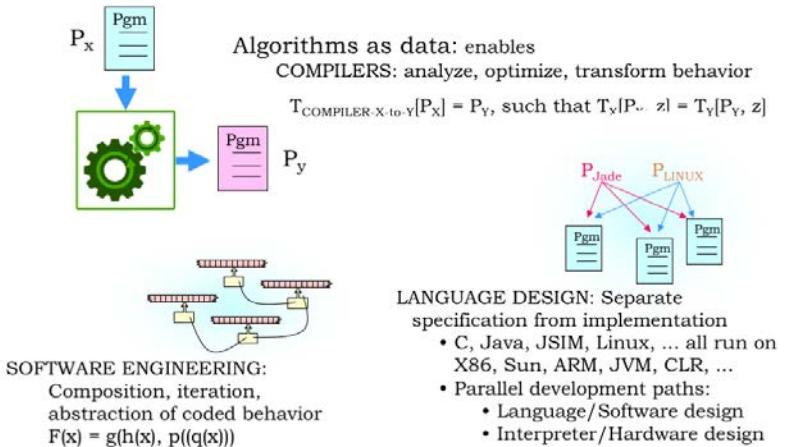
- Actually given finite memory, can only emulate UTMs + inputs up to a certain size
- This is not a high bar: conditional branches (BEQ) and some simple arithmetic (SUB) are enough.

The Universal Turing Machine is the paradigm for modern general-purpose computers. Given an ISA we want to know if it's equivalent to a universal Turing Machine. If so, it can emulate every other TM and hence compute any computable function.

How do we show our computer is Turing Universal? Simply demonstrate that it can emulate some known Universal Turing Machine. The finite memory on actual computers will mean we can only emulate UTM operations on inputs up to a certain size, but within this limitation we can show our computer can perform any computation that fits into memory.

As it turns out this is not a high bar: so long as the ISA has conditional branches and some simple arithmetic, it will be Turing Universal.

Coded Algorithms: Key to CS data vs hardware



This notion of encoding a program in a way that allows it to be data to some other program is a key idea in computer science.

We often translate a program P_x written to run on some abstract high-level machine (eg, a program in C or Java) into, say, an assembly language program P_y that can be interpreted by our CPU. This translation is called compilation.

Much of software engineering is based on the idea of taking a program and using it as a component in some larger program.

Given a strategy for compiling programs, that opens the door to designing new programming languages that let us express our desired computation using data structures and operations particularly suited to the task at hand.

So what have learned from the mathematicians' work on models of computation? Well, it's nice to know that the computing engine we're planning to build will be able to perform any computation that can be performed on any realizable machine. And the development of the universal Turing Machine model paved the way for modern stored-program computers. The bottom line: we're good to go with the Beta ISA!

Uncomputability (!)

Uncomputable functions: There are well-defined discrete functions that a Turing machine cannot compute

- No algorithm can compute $f(x)$ for arbitrary x in finite number of steps
- Not that we don't know algorithm - can prove no algorithm exists
- Corollary: Finite memory is not the only limiting factor on whether we can solve a problem

The most famous uncomputable function is the so-called Halting function, $f_H(k, j)$, defined by:

$$\begin{aligned} f_H(k, j) = 1 & \text{ if } T_k[j] \text{ halts;} \\ & 0 \text{ otherwise.} \end{aligned}$$

$f_H(k, j)$ determines whether the k^{th} TM halts when given a tape containing j .

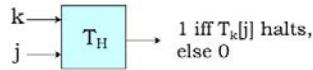
We've discussed computable functions. Are there uncomputable functions?

Yes, there are well-defined discrete functions that cannot be computed by any TM, *i.e.*, no algorithm can compute $f(x)$ for arbitrary finite x in a finite number of steps. It's not that we don't know the algorithm, we can actually prove that no algorithm exists. So the finite memory limitations of FSMs wasn't the only barrier as to whether we can solve a problem.

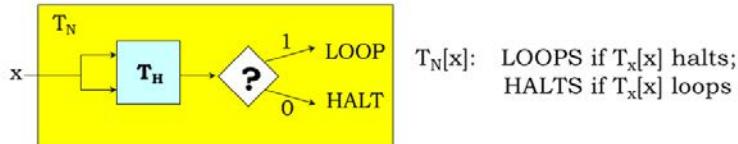
The most famous uncomputable function is the so-called Halting function. When TMs undertake a computation there two possible outcomes. Either the TM writes an answer onto the tape and halts, or the TM loops forever. The Halting function tells which outcome we'll get: given two integer arguments k and j , the Halting function determines if the k^{th} TM halts when given a tape containing j as the input.

Why f_H is Uncomputable

If f_H is computable, it is equivalent to some TM (say, T_H):



Then T_N (N for “Nasty”), which must be computable if T_H is:



Finally, consider giving N as an argument to T_N :

$T_N[N]$: LOOPS if $T_N[N]$ halts;
HALTS if $T_N[N]$ loops

Contradiction!

T_N can't be
computable, hence
 T_H can't either!

Let's quickly sketch an argument as to why the Halting function is not computable. Well, suppose it was computable, then it would be equivalent to some TM, say T_H .

So we can use T_H to build another TM, T_N (the “ N ” stands for nasty!) that processes its single argument and either LOOPS or HALTs. $T_N[X]$ is designed to loop if TM X given input X halts. And vice versa: $T_N[X]$ halts if TM X given input X loops. The idea is that $T_N[X]$ does the opposite of whatever $T_X[X]$ does. T_N is easy to implement assuming that we have T_H to answer the “halts or loops” question.

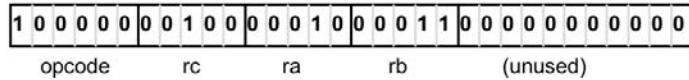
Now consider what happens if we give N as the argument to T_N . From the definition of T_N , $T_N[N]$ will LOOP if the halting function tells us that $T_N[N]$ halts. And $T_N[N]$ will HALT if the halting function tells us that $T_N[N]$ loops. Obviously $T_N[N]$ can't both LOOP and HALT at the same time! So if the Halting function is computable and T_H exists, we arrive at this impossible behavior for $T_N[N]$. This tells us that T_H cannot exist and hence that the Halting function is not computable.

11: Compilers

1. Programming Languages
2. High-level Languages
3. Interpretation
4. Compilation
5. Interpretation vs. Compilation
6. Compilers
7. A Simple Compilation Strategy
8. `compile_expr(expr) → Rx`
9. Compiling Expressions
10. `compile_statement`
11. `compile_statement`: Conditional
12. `compile_statement`: Iteration
13. Putting It All Together: Factorial
14. Optimization: keep values in regs
15. Anatomy of a Modern Compiler
16. Frontend Stages: Lexical Analysis
17. Frontend Stages: Syntactic Analysis
18. Frontend Stages: Semantic Analysis
19. Intermediate Representation (IR)
20. Common IR: Control Flow Graph
21. Control Flow Graph for GCD
22. IR Optimization
23. Example IR Optimizations I
24. Example IR Optimizations II
25. Example IR Optimizations II
26. Example IR Optimizations III
27. Example IR Optimizations IV
28. Example IR Optimizations IV
29. Code Generation
30. Putting It All Together I
31. Putting It All Together II
32. Summary

Programming Languages

32-bit (4-byte) ADD instruction:



Means, to BETA, $\text{Reg}[4] \leftarrow \text{Reg}[2] + \text{Reg}[3]$

We'd rather write

ADD(R2, R3, R4) *(Assembly)*

or better yet

a = b + c; *(High-Level Language)*

Today we're going to talk about how to translate high-level languages into code that computers can execute.

So far we've seen the Beta ISA, which includes instructions that control the datapath operations performed on 32-bit data stored in the registers. There are also instructions for accessing main memory and changing the program counter. The instructions are formatted as opcode, source, and destination fields that form 32-bit values in main memory.

To make our lives easier, we developed assembly language as a way of specifying sequences of instructions. Each assembly language statement corresponds to a single instruction. As assembly language programmers, we're responsible for managing which values are in registers and which are in main memory, and we need to figure out how to break down complicated operations, *e.g.*, accessing an element of an array, into the right sequence of Beta operations.

We can go one step further and use high-level languages to describe the computations we want to perform. These languages use variables and other data structures to abstract away the details of storage allocation and the movement of data to and from main memory. We can just refer to a data object by name and let the language processor handle the details. Similarly, we'll write expressions and other operators such as assignment (=) to efficiently describe what would require many statements in assembly language.

Today we're going to dive into how to translate high-level language programs into code that will run on the Beta.

High-Level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm:

```
/* Compute greatest common divisor
 * using Euclid's method
 */
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```

- 6.004 uses C, a common systems programming language. Modern popular alternatives include C++, Java, Python, and many others
- Advantages over assembly
 - Productivity (concise, readable, maintainable)
 - Correctness (type checking, etc)
 - Portability (run same program on different hardware)
- Disadvantages over assembly?
 - Efficiency?

Implementations: Interpretation vs compilation

Here we see Euclid's algorithm for determining the greatest common divisor of two numbers, in this case the algorithm is written in the C programming language. We'll be using a simple subset of C as our example high-level language. Please see the brief overview of C in the Handouts section if you'd like an introduction to C syntax and semantics. C was developed by Dennis Ritchie at AT&T Bell Labs in the late 60's and early 70's to use when implementing the Unix operating system. Since that time many new high-level languages have been introduced providing modern abstractions like object-oriented programming along with useful new data and control structures.

Using C allows us describe a computation without referring to any of the details of the Beta ISA like registers, specific Beta instructions, and so on. The absence of such details means there is less work required to create the program and makes it easier for others to read and understand the algorithm implemented by the program.

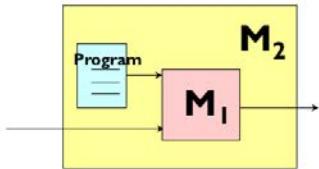
There are many advantages to using a high-level language. They enable programmers to be very productive since the programs are concise and readable. These attributes also make it easy to maintain the code. Often it is harder to make certain types of mistakes since the language allows us to check for silly errors like storing a string value into a numeric variable. And more complicated tasks like dynamically allocating and deallocating storage can be completely automated. The result is that it can take much less time to create a correct program in a high-level language than it would when writing in assembly language.

Since the high-level language has abstracted away the details of a particular ISA, the programs are portable in the sense that we can expect to run the same code on different ISAs without having to rewrite the code.

What do we lose by using a high-level language? Should we worry that we'll pay a price in terms of the efficiency and performance we might get by crafting each instruction by hand? The answer depends on how we choose to run high-level language programs. The two basic execution strategies are "interpretation" and "compilation".

Interpretation

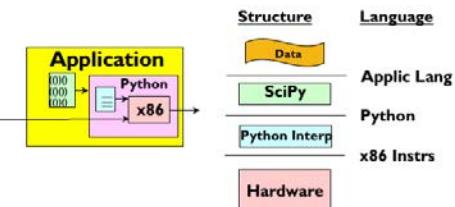
Model of Interpretation:



- Start with some hard-to-program machine, say M₁
- Write a single program for M₁ that mimics the behavior of some easier machine, M₂
- Result: a “virtual” M₂

Layers of interpretation:

- Often we use several layers of interpretation to achieve desired behavior, e.g.:
 - x86 CPU, running
 - Python, running
 - SciPy application, performing
 - Numerical analysis



To interpret a high-level language program, we'll write a special program called an “interpreter” that runs on the actual computer, M₁. The interpreter mimics the behavior of some abstract easy-to-program machine M₂ and for each M₂ operation executes sequences of M₁ instructions to achieve the desired result. We can think of the interpreter along with M₁ as an implementation of M₂, *i.e.*, given a program written for M₂, the interpreter will, step-by-step, emulate the effect of M₂ instructions.

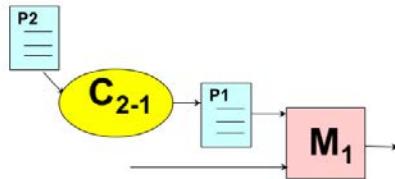
We often use several layers of interpretation when tackling computation tasks. For example, an engineer may use her laptop with an Intel CPU to run the Python interpreter. In Python, she loads the SciPy toolkit, which provides a calculator-like interface for numerical analysis for matrices of data. For each SciPy command, *e.g.*, “find the maximum value of a dataset”, the SciPy tool kit executes many Python statements, *e.g.*, to loop over each element of the array, remembering the largest value. For each Python statement, the Python interpreter executes many x86 instructions, *e.g.*, to increment the loop index and check for loop termination. Executing a single SciPy command may require executing of tens of Python statements, which in turn each may require executing hundreds of x86 instructions. The engineer is very happy she didn't have to write each of those instructions herself!

Interpretation is an effective implementation strategy when performing a computation once, or when exploring which computational approach is most effective before making a more substantial investment in creating a more efficient implementation.

Compilation

Model of Compilation:

- Given some hard-to-program machine, say M_1 ...
- Find some easier-to-program language L_2 (perhaps for a more complicated machine, M_2); write programs in that language
- Build a translator (compiler) that translates programs from M_2 's language to M_1 's language. May run on M_1 , M_2 , or some other machine.



Interpretation and compilation: two ways to execute high-level languages

- Both allow changes in the source program
- Both afford programming applications in platform (e.g., processor) independent languages
- Both are widely used in modern computer systems!

We'll use a compilation implementation strategy when we have computational tasks that we need to execute repeatedly and hence we are willing to invest more time up-front for more efficiency in the long-term.

In compilation, we also start with our actual computer M_1 . Then we'll take our high-level language program P_2 and translate it statement-by-statement into a program for M_1 . Note that we're not actually running the P_2 program. Instead we're using it as a template to create an equivalent P_1 program that can execute directly on M_1 . The translation process is called "compilation" and the program that does the translation is called a "compiler".

We compile the P_2 program once to get the translation P_1 , and then we'll run P_1 on M_1 whenever we want to execute P_2 . Running P_1 avoids the overhead of having to process the P_2 source and the costs of executing any intervening layers of interpretation. Instead of dynamically figuring out the necessary machine instructions for each P_2 statement as it's encountered, in effect we've arranged to capture that stream of machine instructions and save them as a P_1 program for later execution. If we're willing to pay the up-front costs of compilation, we'll get more efficient execution.

And, with different compilers, we can arrange to run P_2 on many different machines — M_2 , M_3 , etc. — without having rewrite P_2 .

So we now have two ways to execute a high-level language program: interpretation and compilation. Both allow us to change the original source program. Both allow us to abstract away the details of the actual computer we'll use to run the program. And both strategies are widely used in modern computer systems!

Interpretation vs Compilation

- Characteristic differences:

	Interpretation	Compilation
How it treats input “x+2”	Computes $x+2$	Generates a program that computes $x+2$
When it happens	During execution	Before execution
What it complicates/slow	Program execution	Program development
Decisions made at	Run time	Compile time

- Major choice we'll see repeatedly: do it at compile time or at run time?
 - Which is faster?
 - Which is more general?

Let's summarize the differences between interpretation and compilation.

Suppose the statement “ $x+2$ ” appears in the high-level program. When the interpreter processes this statement it immediately fetches the value of the variable x and adds 2 to it. On the other hand, the compiler would generate Beta instructions that would LD the variable x into a register and then ADD 2 to that value.

The interpreter is executing each statement as it's processed and, in fact, may process and execute the same statement many times if, *e.g.*, it was in a loop. The compiler is just generating instructions to be executed at some later time.

Interpreters have the overhead of processing the high-level source code during execution and that overhead may be incurred many times in loops. Compilers incur the processing overhead once, making the eventual execution more efficient. But during development, the programmer may have to compile and run the program many times, often incurring the cost of compilation for only a single execution of the program. So the compile-run-debug loop can take more time.

The interpreter is making decisions about the data type of x and the type of operations necessary at run time, *i.e.*, while the program is running. The compiler is making those decisions during the compilation process.

Which is the better approach? In general, executing compiled code is much faster than running the code interpretively. But since the interpreter is making decisions at run time, it can change its behavior depending, say, on the type of the data in the variable X , offering considerable flexibility in handling different types of data with the same algorithm. Compilers take away that flexibility in exchange for fast execution.

Compilers

- Bare minimum for a functional compiler:
- 

- Good compilers:
 - Produce meaningful errors on incorrect programs
 - Even better: meaningful warnings
 - Produce fast, optimized code

- This lecture:
 - Simple techniques to compile a C programs into assembly
 - Overview of how modern compilers work

A compiler is a program that translates a high-level language program into a functionally equivalent sequence of machine instructions, *i.e.*, an assembly language program.

A compiler first checks that the high-level program is correct, *i.e.*, that the statements are well formed, the programmer isn't asking for nonsensical computations — *e.g.*, adding a string value and an integer — or attempting to use the value of a variable before it has been properly initialized. The compiler may also provide warnings when operations may not produce the expected results, *e.g.*, when converting from a floating-point number to an integer, where the floating-point value may be too large to fit in the number of bits provided by the integer.

If the program passes scrutiny, the compiler then proceeds to generate efficient sequences of instructions, often finding ways to rearrange the computation so that the resulting sequences are shorter and faster. It's hard to beat a modern optimizing compiler at producing assembly language, since the compiler will patiently explore alternatives and deduce properties of the program that may not be apparent to even diligent assembly language programmers.

In this section, we'll look at a simple technique for compiling C programs into assembly. Then, in the next section, we'll dive more deeply into how a modern compiler works.

A Simple Compilation Strategy

- Programs are sequences of statements, so repeatedly call `compile_statement(statement)`:
 - Unconditional: `expr;`
 - Compound: `{ statement1; statement2; ... }`
 - Conditional: `if (expr) statement1; else statement2;`
 - Iteration: `while (expr) statement;`
- Also need `compile_expr(expr)` to generate code to compute value of `expr` into a register
 - Constants: `1234`
 - Variables: `a, b[expr]`
 - Assignment: `a = expr`
 - Operations: `expr1 + expr2, ...`
 - Procedure calls: `proc(expr, ...)`



There are two main routines in our simple compiler: `compile_statement` and `compile_expr`. The job of `compile_statement` is to compile a single statement from the source program. Since the source program is a sequence of statements, we'll be calling `compile_statement` repeatedly.

We'll focus on the compilation technique for four types of statements. An unconditional statement is simply an expression that's evaluated once. A compound statement is simply a sequence of statements to be executed in turn. Conditional statements, sometimes called "if statements", compute the value of a test expression, e.g., a comparison such as "A < B". If the test is true then `statement1` is executed, otherwise `statement2` is executed. Iteration statements also contain a test expression. In each iteration, if the test true, then the statement is executed, and the process repeats. If the test is false, the iteration is terminated.

The other main routine is `compile_expr` whose job it is to generate code to compute the value of an expression, leaving the result in some register. Expressions take many forms:

simple constant values

values from scalar or array variables,

assignment expressions that compute a value and then store the result in some variable,

unary or binary operations that combine the values of their operands with the specified operator. Complex arithmetic expressions can be decomposed into sequences of unary and binary operations.

And, finally, procedure calls, where a named sequence of statements will be executed with the values of the supplied arguments assigned as the values for the formal parameters of the procedure. Compiling procedures and procedure calls is a topic that we'll tackle next lecture since there are some complications to understand and deal with.

Happily, compiling the other types of expressions and statements is straightforward, so let's get started.

compile_expr(expr) \Rightarrow Rx

- Constants: $1234 \Rightarrow Rx$
 - CMOVE(1234,Rx)
 - LD(c1,Rx)
 - ...
 - c1: LONG(123456)

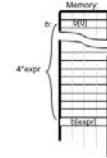
- Variables: $a \Rightarrow Rx$
 - LD(a,Rx)
 - ...
 - a: LONG(0)

- Assignment: $a=expr \Rightarrow Rx$
 - compile_expr(expr) $\Rightarrow Rx$
 - ST(Rx,a)

- Variables: $b[expr] \Rightarrow Rx$
 - compile_expr(expr) $\Rightarrow Rx$
 - MULC(Rx,bsize,Rx)
 - LD(Rx,b,Rx)
 - ...
 - // reserve array space
 - b: . = . + bsize*bLen

- Operations:

$$expr_1 + expr_2 \Rightarrow Rx$$
 - compile_expr(expr₁) $\Rightarrow Rx$
 - compile_expr(expr₂) $\Rightarrow Ry$
 - ADD(Rx,Ry,Rx)



What code do we need to put the value of a constant into a register? If the constant will fit into the 16-bit constant field of an instruction, we can use CMOVE to load the sign-extended constant into a register. This approach works for constants between -32768 and +32767. If the constant is too large, it's stored in a main memory location and we use a LD instruction to get the value into a register.

Loading the value of a variable is much the same as loading the value of a large constant: we use a LD instruction to access the memory location that holds the value of the variable.

Performing an array access is slightly more complicated: arrays are stored as consecutive locations in main memory, starting with index 0. Each array element occupies some fixed number bytes. So we need code to convert the array index into the actual main memory address for the specified array element.

We first invoke compile_expr to generate code that evaluates the index expression and leaves the result in Rx. That will be a value between 0 and the size of the array minus 1. We'll use a LD instruction to access the appropriate array entry, but that means we need to convert the index into a byte offset, which we do by multiplying the index by bsize, the number of bytes in one element. If b was an array of integers, bsize would be 4. Now that we have the byte offset in a register, we can use LD to add the offset to the base address of the array computing the address of the desired array element, then load the memory value at that address into a register.

Assignment expressions are easy: invoke compile_expr to generate code that loads the value of the expression into a register, then generate a ST instruction to store the value into the specified variable.

Arithmetic operations are pretty easy too: use compile_expr to generate code for each of the operand expressions, leaving the results in registers. Then generate the appropriate ALU instruction to combine the operands and leave the answer in a register.

Compiling Expressions

C code:

```
int x, y;  
y = (x-3)*(y+123456)
```

```
compile_expr(y = (x-3)*(y+123456))  
compile_expr((x-3)*(y+123456))  
compile_expr(x-3)  
compile_expr(x)
```

Beta assembly code:

```
x: LONG(0)  
y: LONG(0)  
c1: LONG(123456)  
...
```

```
LD(x, r1)  
CMOVE(3, r2)  
SUB(r1, r2, r1) => SUBC(r1, 3, r1)  
LD(y, r2)  
LD(c1, r3)  
ADD(r2, r3, r2)  
MUL(r2, r1, r1)  
ST(r1, y)
```

```
compile_expr(3)  
CMOVE(3, r2)  
SUB(r1, r2, r1)  
compile_expr(y+123456)  
compile_expr(y)  
LD(y, r2)  
compile_expr(123456)  
LD(c1, r3)  
ADD(r2, r3, r2)  
MUL(r1, r2, r1)  
ST(r1, y)
```

Let's look at example to see how all this works. Here have an assignment expression that requires a subtract, a multiply, and an addition to compute the required value.

Let's follow the compilation process from start to finish as we invoke `compile_expr` to generate the necessary code.

Following the template for assignment expressions from the previous page, we recursively call `compile_expr` to compute value of the right-hand-side of the assignment.

That's a multiply operation, so, following the Operations template, we need to compile the left-hand operand of the multiply.

That's a subtract operation, so, we call `compile_expr` again to compile the left-hand operand of the subtract.

Aha, we know how to get the value of a variable into a register. So we generate a LD instruction to load the value of x into r1.

The process we're following is called “recursive descent”. We've used recursive calls to `compile_expr` to process each level of the expression tree. At each recursive call the expressions get simpler, until we reach a variable or constant, where we can generate the appropriate instruction without descending further. At this point we've reached a leaf of the expression tree and we're done with this branch of the recursion.

We now need to get the value of the right-hand operand of the subtract into a register. In case it's a small constant, so we generate a CMOVE instruction.

Now that both operand values are in registers, we return to the subtract template and generate a SUB instruction to do the subtraction. We now have the value for the left-hand operand of the multiply in r1.

We follow the same process for the right-hand operand of the multiply, recursively calling `compile_expr` to process each level of the expression until we reach a variable or constant. Then we

return up the expression tree, generating the appropriate instructions as we go, following the dictates of the appropriate template from the previous slide.

The generated code is shown on the left of the slide. The recursive-descent technique makes short work of generating code for even the most complicated of expressions.

There's even opportunity to find some simple optimizations by looking at adjacent instructions. For example, a CMOVE followed by an arithmetic operation can often be shorted to a single arithmetic instruction with the constant as its second operand. These local transformations are called "peephole optimizations" since we're only considering just one or two instructions at a time.

compile_statement

- Unconditional: *expr*;

Beta assembly:

```
compile_expr(expr)
```

- Compound: { *statement₁*; *statement₂*; ... }

Beta assembly:

```
compile_statement(statement1)
```

```
compile_statement(statement2)
```

```
...
```

Now let's turn our attention compile _ statement.

The first two statement types are pretty easy to handle. Unconditional statements are usually assignment expressions or procedure calls. We'll simply ask compile_expr to generate the appropriate code.

Compound statements are equally easy. We'll recursively call compile_statement to generate code for each statement in turn. The code for statement_2 will immediately follow the code generated for statement_1. Execution will proceed sequentially through the code for each statement.

compile_statement: Conditional

C code:

```
if (expr)
    statement;
```

Beta assembly:

```
compile_expr(expr)⇒Rx
BF(rx, Lendif)
compile_statement(statement)
```

Lendif:

C code:

```
if (expr)
    statement1;
else
    statement2;
```

Beta assembly:

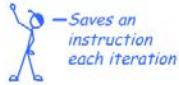
```
compile_expr(expr)⇒Rx
BF(rx, Lelse)
compile_statement(statement1)
BR(Lendif)
Lelse:
compile_statement(statement2)
Lendif:
```

Here we see the simplest form the conditional statement, where we need to generate code to evaluate the test expression and then, if the value in the register is FALSE, skip over the code that executes the statement in the THEN clause. The simple assembly-language template uses recursive calls to compile_expr and compile_statement to generate code for the various parts of the IF statement.

The full-blown conditional statement includes an ELSE clause, which should be executed if the value of the test expression is FALSE. The template uses some branches and labels to ensure the course of execution is as intended.

You can see that the compilation process is really just the application of many small templates that break the code generation task down step-by-step into smaller and smaller tasks, generating the necessary code to glue all the pieces together in the appropriate fashion.

compile_statement: Iteration

C code:	Beta assembly:	Better Beta assembly:
<pre>while (expr) statement;</pre>	<pre>Lwhile: compile_expr(expr)⇒Rx BF(rx, Lendwhile) compile_statement(statement) BR(Lwhile) Lendwhile:</pre>	<pre>BR(Ltest) Lwhile: compile_statement(statement) Ltest: compile_expr(expr)⇒Rx BT(rx, Lwhile)</pre>
		 —Saves an instruction each iteration
C code:	is equivalent to:	
<pre>for (init; test; increment) statement;</pre>	<pre>init; while (test) { statement; increment; }</pre>	
Example:		
<pre>for (i=0; i < 10; i = i + 1) sum = sum + b[i];</pre>		

And here's the template for the WHILE statement, which looks a lot like the template for the IF statement with a branch at the end that causes the generated code to be re-executed until the value of the test expression is FALSE.

With a bit of thought, we can improve on this template slightly. We've reorganized the code so that only a single branch instruction (BT) is executed each iteration, instead of the two branches (BF, BR) per iteration in the original template. Not a big deal, but little optimizations to code inside a loop can add up to big savings in a long-running program.

Just a quick comment about another common iteration statement, the FOR loop. The FOR loop is a shorthand way of expressing iterations where the loop index ("i" in the example shown) is run through a sequence of values and the body of the FOR loop is executed once for each value of the loop index.

The FOR loop can be transformed into the WHILE statement shown here, which can then be compiled using the templates shown above.

Putting It All Together: Factorial

```
int n = 20;      {   n: LONG(20)
int r = 0;       {   r: LONG(0)
start:
r = 1;          {     CMOVE(1, r0)
                  ST(r0, r)
while (n > 0) { {   BR(test)           Easy translation
loop:
                  LD(r, r3)
                  LD(n,r1)
                  MUL(r1, r3, r3)      Slow code
                  ST(r3, r)           (10 instructions
                                         in the loop)
                  LD(n,r1)
                  SUBC(r1, 1, r1)
                  ST(r1, n)
}
test:
                  LD(n, r1)
                  CMPLT(r31, r1, r2)
                  BT(r2, loop)
done:
```

In this example, we've applied our templates to generate code for the iterative implementation of the factorial function that we've seen before. Look through the generated code and you'll be able to match code fragments with the templates from last couple of slides. It's not the most efficient code, but not bad given the simplicity of the recursive-descent approach for compiling high-level programs.

Optimization: keep values in regs

```
int n = 20,      n: LONG(20)
int r;          r: LONG(0)

r = 1;          start:
                CMOVE(1, r0)
                ST(r0, r)
                LD(n,r1)    | keep n in r1
                LD(r,r3)    | keep r in r3

while (n > 0)  BR(test)
{
    r = r*n;    loop:
                MUL(r1, r3, r3)
                SUBC(r1, 1, r1)
    n = n-1;    test:
                CMPLT(r31, r1, r2)
                BT(r2, loop)

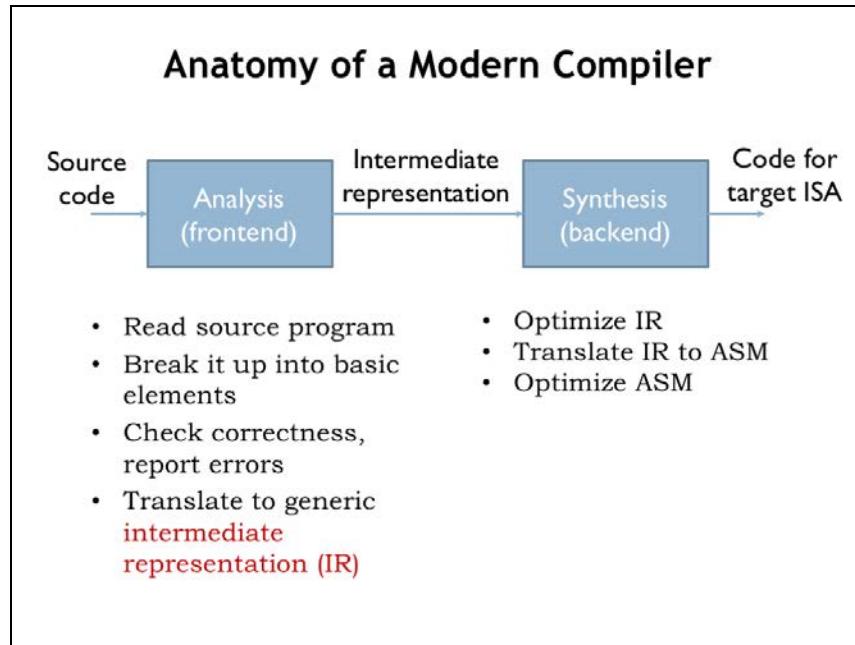
done:
                ST(r1,n)    | save final n
                ST(r3,r)    | save final r
```

Optimization:
Keep n, r in registers
⇒ move LDs/STs
out of loop!

4 instructions in the loop

It's a simple matter to modify the recursive-descent process to accommodate variable values that are stored in dedicated registers rather than in main memory. Optimizing compilers are quite good at identifying opportunities to keep values in registers and hence avoid the LD and ST operations needed to access values in main memory. Using this simple optimization, the number of instructions in the loop has gone from 10 down to 4. Now the generated code is looking pretty good!

But rather than keep tweaking the recursive-descent approach, let's stop here. In the next segment, we'll see how modern compilers take a more general approach to generating code. Still though, the first time I learned about recursive descent, I ran home to write a simple implementation and marveled at having authored my own compiler in an afternoon!



A modern compiler starts by analyzing the source program text to produce an equivalent sequence of operations expressed in a language — and machine-independent intermediate representation (IR). The analysis, or frontend, phase checks that program is well-formed, *i.e.*, that the syntax of each high-level language statement is correct. It understands the meaning (semantics) of each statement. Many high-level languages include declarations of the type — *e.g.*, integer, floating point, string, etc. — of each variable, and the frontend verifies that all operations are correctly applied, ensuring that numeric operations have numeric-type operands, string operations have string-type operands, and so on. Basically the analysis phase converts the text of the source program into an internal data structure that specifies the sequence and type of operations to be performed.

Often there are families of frontend programs that translate a variety of high-level languages (*e.g.*, C, C++, Java) into a common IR.

The synthesis, or backend, phase then optimizes the IR to reduce the number of operations that will be executed when the final code is run. For example, it might find operations inside of a loop that are independent of the loop index and can move outside the loop, where they are performed once instead of repeatedly inside the loop. Once the IR is in its final optimized form, the backend generates code sequences for the target ISA and looks for further optimizations that take advantage of particular features of the ISA. For example, for the Beta ISA we saw how a CMOVE followed by an arithmetic operation can be shorted to a single operation with a constant operand.

Frontend Stages

- Lexical analysis (scanning): Source → List of tokens

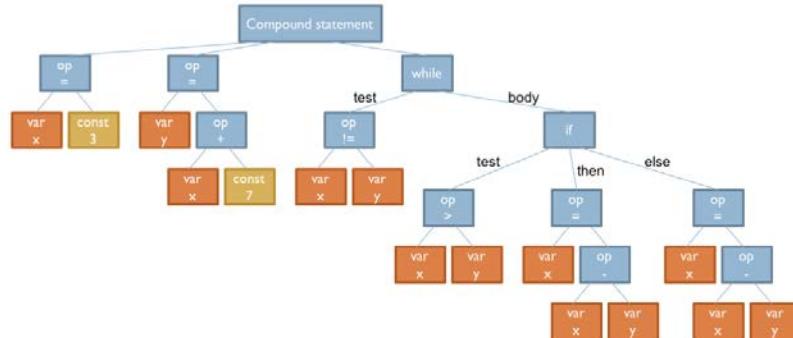
```
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
("int", KEYWORD)
("x", IDENTIFIER)
("=", OPERATOR)
("3", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("int", KEYWORD)
("y", IDENTIFIER)
("=", OPERATOR)
("x", IDENTIFIER)
("+", OPERATOR)
("7", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("while", KEYWORD)
("(" , SPECIAL_SYMBOL)
...

```

The analysis phase starts by scanning the source text and generating a sequence of token objects that identify the type of each piece of the source text. While spaces, tabs, newlines, and so on were needed to separate tokens in the source text, they've all been removed during the scanning process. To enable useful error reporting, token objects also include information about where in the source text each token was found, *e.g.*, the file name, line number, and column number. The scanning phase reports illegal tokens, *e.g.*, the token “3x” would cause an error since in C it would not be a legal number or a legal variable name.

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree



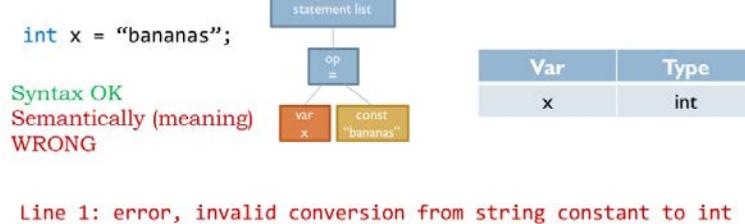
The parsing phase processes the sequence of tokens to build the syntax tree, which captures the structure of the original program in a convenient data structure. The operands have been organized for each unary and binary operation. The components of each statement have been found and labeled. The role of each source token has been determined and the information captured in the syntax tree.

Compare the labels of the nodes in the tree to the templates we discussed in the previous segment. We can see that it would be easy to write a program that did a depth-first tree walk, using the label of each tree node to select the appropriate code generation template. We won't do that quite yet since there's still some work to be done analyzing and transforming the tree.

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree
- Semantic analysis (mainly, type checking)

Consider:



The syntax tree makes it easy to verify that the program is semantically correct, *e.g.*, to check that the types of the operands are compatible with the requested operation.

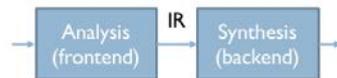
For example, consider the statement `x = "bananas"`. The syntax of the assignment operation is correct: there's a variable on the left-hand side and an expression on the right-hand side. But the semantics is not correct, at least in the C language! By looking in its symbol table to check the declared type for the variable `x` (int) and comparing it to the type of the expression (string), the semantic checker for the "op =" tree node will detect that the types are not compatible, *i.e.*, that we can't store a string value into an integer variable.

When the semantic analysis is complete, we know that the syntax tree represents a syntactically correct program with valid semantics, and we've finished converting the source program into an equivalent, language-independent sequence of operations.

Intermediate Representation (IR)

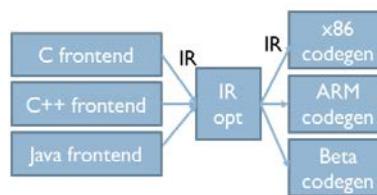
- Internal compiler language that is:

- Language-independent
 - Machine-independent
 - Easy to optimize



- Why yet another language?

- Assembly does not have enough info to optimize it well
 - Enables modularity and reuse



The syntax tree is a useful intermediate representation (IR) that is independent of both the source language and the target ISA. It contains information about the sequencing and grouping of operations that isn't apparent in individual machine language instructions. And it allows frontends for different source languages to share a common backend targeting a specific ISA. As we'll see, the backend processing can be split into two sub-phases. The first performs machine-independent optimizations on the IR. The optimized IR is then translated by the code generation phase into sequences of instructions for the target ISA.

Common IR: Control Flow Graph

- Assignments:



- Basic block: Sequence of assignments with an optional branch at the end

```
x = 3  
y = x + 7  
if (x != y)
```

- Control flow graph:

- Nodes: Basic blocks
- Edges: branches between basic blocks

A common IR is to reorganize the syntax tree into what's called a control flow graph (CFG). Each node in the graph is a sequence of assignment and expression evaluations that ends with a branch. The nodes are called "basic blocks" and represent sequences of operations that are executed as a unit: once the first operation in a basic block is performed, the remaining operations will also be performed without any other intervening operations. This knowledge lets us consider many optimizations, *e.g.*, temporarily storing variable values in registers, that would be complicated if there was the possibility that other operations outside the block might also need to access the variable values while we were in the middle of this block.

The edges of the graph indicate the branches that take us to another basic block.

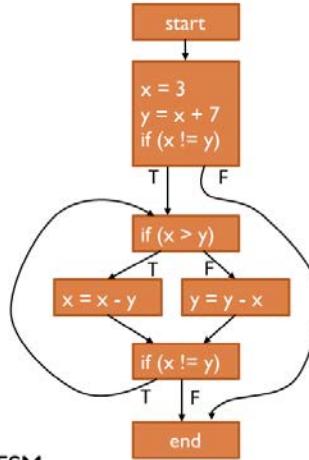
Control Flow Graph for GCD

```

int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}

```

Looks like a high-level FSM...



For example, here's the CFG for GCD.

If a basic block ends with a conditional branch, there are two edges, labeled “T” and “F” leaving the block that indicate the next block to execute depending on the outcome of the test. Other blocks have only a single departing arrow, indicating that the block always transfers control to the block indicated by the arrow.

Note that if we can arrive at a block from only a single predecessor block, then any knowledge we have about operations and variables from the predecessor block can be carried over to the destination block. For example, if the “if ($x > y$)” block has generated code to load the values of x and y into registers, both destination blocks can use that information and use the appropriate registers without having to generate their own LDs.

But if a block has multiple predecessors, such optimizations are more constrained: we can only use knowledge that is common to *all* the predecessor blocks.

The CFG looks a lot like the state transition diagram for a high-level FSM!

IR Optimization

- Perform a set of passes over the CFG
 - Each pass does a specific, simple task over the CFG
 - By repeating multiple simple passes on the CFG over and over, compilers achieve very complex optimizations
- Example optimizations:
 - Dead code elimination: Eliminate assignments to variables that are never used, or basic blocks that are never reached
 - Constant propagation: Identify variables that are constant, substitute the constant elsewhere
 - Constant folding: Compute and substitute constant expressions

We'll optimize the IR by performing multiple passes over the CFG. Each pass performs a specific, simple optimization. We'll repeatedly apply the simple optimizations in multiple passes, until we can't find any further optimizations to perform. Collectively, the simple optimizations can combine to achieve very complex optimizations.

Here are some example optimizations:

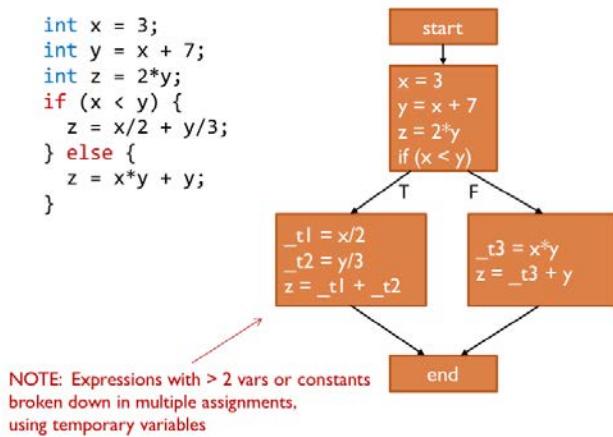
We can eliminate assignments to variables that are never used and basic blocks that are never reached. This is called “dead code elimination”.

In constant propagation, we identify variables that have a constant value and substitute that constant in place of references to the variable.

We can compute the value of expressions that have constant operands. This is called “constant folding”.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

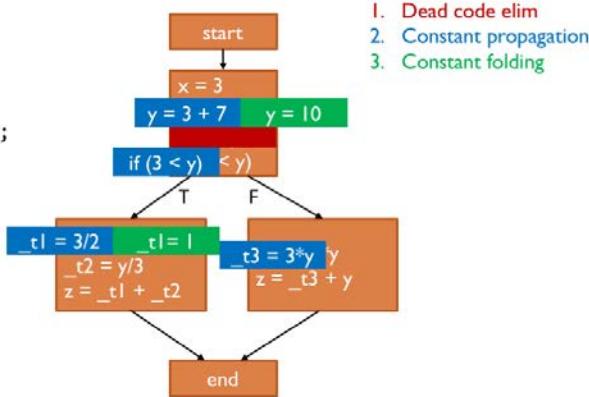


NOTE: Expressions with > 2 vars or constants broken down in multiple assignments, using temporary variables

To illustrate how these optimizations work, consider this slightly silly source program and its CFG. Note that we've broken down complicated expressions into simple binary operations, using temporary variable names (e.g., “ $_t1$ ”) to name the intermediate results.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



Let's get started!

The dead code elimination pass can remove the assignment to Z in the first basic block since Z is reassigned in subsequent blocks and the intervening code makes no reference to Z.

Next we look for variables with constant values. Here we find that X is assigned the value of 3 and is never re-assigned, so we can replace all references to X with the constant 3.

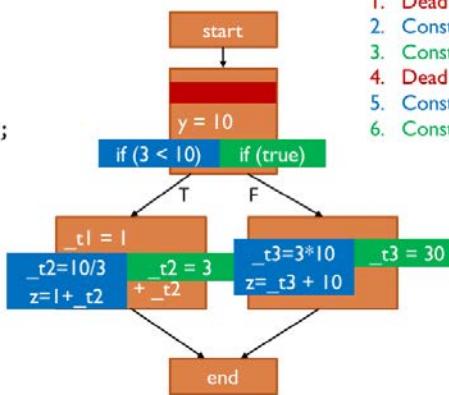
Now perform constant folding, evaluating any constant expressions.

Example IR Optimizations

```

int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}

```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding

Here's the updated CFG, ready for another round of optimizations.

First dead code elimination.

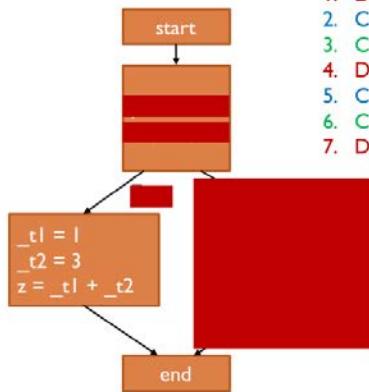
Then constant propagation.

And, finally, constant folding.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim

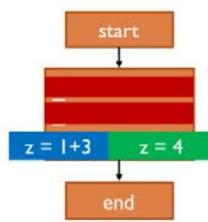


So after two rounds of these simple operations, we've thinned out a number of assignments. Onto round three!

Dead code elimination. And here we can determine the outcome of a conditional branch, eliminating entire basic blocks from the IR, either because they're now empty or because they can no longer be reached.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim

Wow, the IR is now considerably smaller.

Next is another application of constant propagation.

And then constant folding.

Followed by more dead code elimination.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



Dumb repetition of simple transformations on CFGs

Extremely powerful optimizations

More optimizations by adding passes: Common subexpression elimination, loop-invariant code motion, loop unrolling...

1. Dead code elim
 2. Constant propagation
 3. Constant folding
 4. Dead code elim
 5. Constant propagation
 6. Constant folding
 7. Dead code elim
 8. Constant propagation
 9. Constant folding
 10. Dead code elim
 11. Constant propagation
 12. Constant folding
 13. Dead code elim
 14. Constant propagation
 15. Constant folding
- No changes in 13,14,15 → DONE

The passes continue until we discover there are no further optimizations to perform, so we're done!

Repeated applications of these simple transformations have transformed the original program into an equivalent program that computes the same final value for Z.

We can do more optimizations by adding passes: eliminating redundant computation of common subexpressions, moving loop-independent calculations out of loops, unrolling short loops to perform the effect of, say, two iterations in a single loop execution, saving some of the cost of increment and test instructions. Optimizing compilers have a sophisticated set of optimizations they employ to make smaller and more efficient code.

Code Generation

- Translate generated IR to assembly
- Register allocation: Map variables to registers
 - If variables > registers, map some to memory, and load/store them when needed
- Translate each assignment to instructions
 - Some assignments may require > 1 instr if our ISA doesn't have op
- Emit each basic block: label, assignments, and branches
- Lay out basic blocks, removing superfluous jumps
- ISA and CPU-specific optimizations
 - e.g., if possible, reorder instructions to improve performance

Okay, we're done with optimizations. Now it's time to generate instructions for the target ISA.

First the code generator assigns each variable a dedicated register. If we have more variables than registers, some variables are stored in memory and we'll use LD and ST to access them as needed. But frequently-used variables will almost certainly live as much as possible in registers.

Use our templates from before to translate each assignment and operation into one or more instructions.

Then emit the code for each block, adding the appropriate labels and branches.

Reorder the basic block code to eliminate unconditional branches wherever possible.

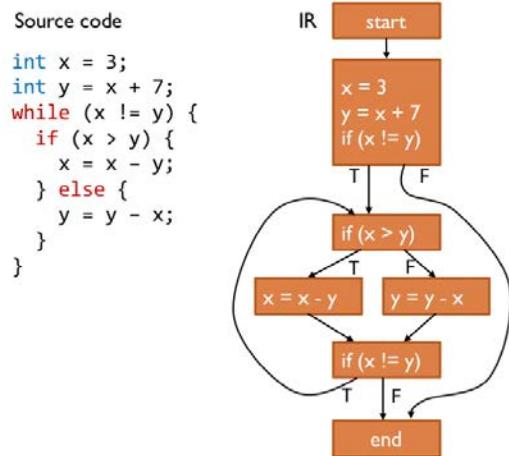
And finally perform any target-specific peephole optimizations.

Putting It All Together: GCD

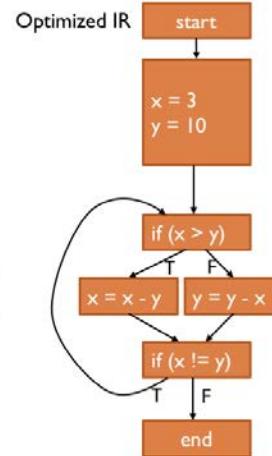
Source code

```
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

IR



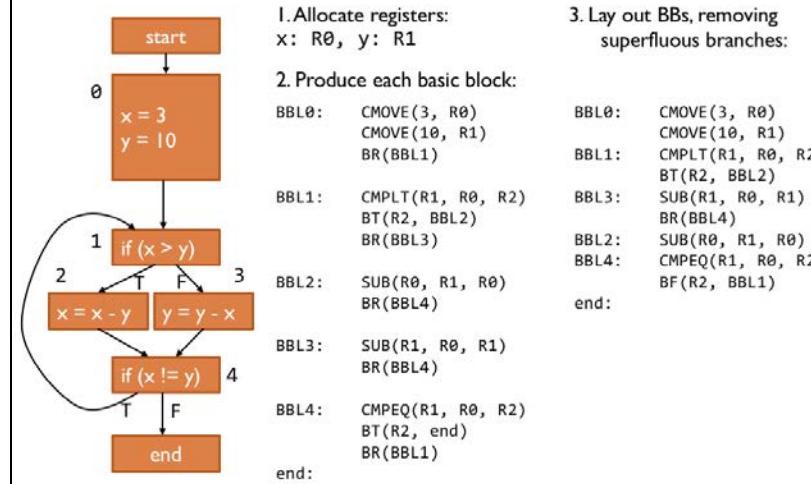
Optimized IR



Here's the original CFG for the GCD code, along with the slightly optimized CFG. GCD isn't as trivial as the previous example, so we've only been able to do a bit of constant propagation and constant folding.

Note that we can't propagate knowledge about variable values from the top basic block to the following "if" block since the "if" block has multiple predecessors.

Putting It All Together: GCD



Here's how the code generator will process the optimized CFG.

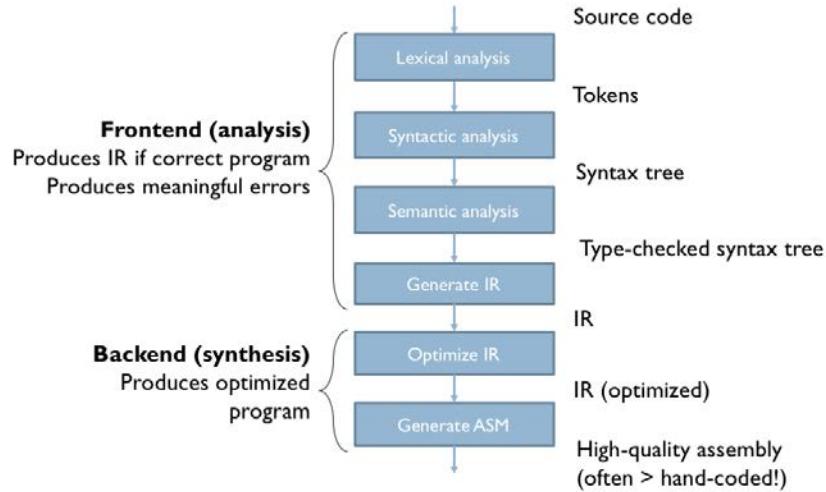
First, it dedicates registers to hold the values for x and y .

Then, it emits the code for each of the basic blocks.

Next, reorganize the order of the basic blocks to eliminate unconditional branches wherever possible.

The resulting code is pretty good. There are no obvious changes that a human programmer might make to make the code faster or smaller. Good job, compiler!

Summary: Modern Compilers



Here are all the compilation steps shown in order, along with their input and output data structures. Collectively they transform the original source code into high-quality assembly code. The patient application of optimization passes often produces code that's more efficient than writing assembly language by hand.

Nowadays, programmers are able to focus on getting the source code to achieve the desired functionality and leave the details of translation to instructions in the hands of the compiler.

12: Procedures and Stacks

1. Procedures: A Software Abstraction
2. Implementing Procedures
3. Procedure Calling Convention
4. Procedure Linkage: First Try
5. Procedure Storage Needs
6. Activation Records
7. Insight: We Need a Stack!
8. Stack Implementation
9. Stack Management Macros
10. Fun With Stacks
11. Solving Procedure Linkage Problems
12. Stack Frames as Activation Records
13. Stack Frame Details
14. Argument Order & BP Usage
15. Procedure Linkage: The Contract
16. Procedure Linkage Templates
17. Putting It All Together: Factorial
18. Recursion?
19. Stack Detective
20. Summary of Dedicated Registers
21. Summary

Procedures: A Software Abstraction

- Procedure: Reusable code fragment that performs a specific task
 - Single named entry point
 - Zero or more formal parameters
 - Local storage
 - Returns control to the caller when finished
- Using multiple procedures enables **abstraction** and **reuse**
 - Compose large programs from collections of simple procedures

```
int gcd(int a, int b) {  
    int x = a;  
    int y = b;  
    while (x != y) {  
        if (x > y) {  
            x = x - y;  
        } else {  
            y = y - x;  
        }  
    }  
    return x;  
}  
  
bool coprimes(int a, int b) {  
    return gcd(a, b) == 1;  
}  
  
coprimes(5, 10); // false  
coprimes(9, 10); // true
```

One of the most useful abstractions provided by high-level languages is the notion of a procedure or subroutine, which is a sequence of instructions that perform a specific task.

A procedure has a single named entry point, which can be used to refer to the procedure in other parts of the program. In the example here, this code is defining the GCD procedure, which is declared to return an integer value.

Procedures have zero or more formal parameters, which are the names the code inside the procedure will use to refer the values supplied when the procedure is invoked by a “procedure call”. A procedure call is an expression that has the name of the procedure followed by parenthesized list of values called “arguments” that will be matched up with the formal parameters. For example, the value of the first argument will become the value of the first formal parameter while the procedure is executing.

The body of the procedure may define additional variables, called “local variables”, since they can only be accessed by statements in the procedure body. Conceptually, the storage for local variables only exists while the procedure is executing. They are allocated when the procedure is invoked and deallocated when the procedure returns.

The procedure may return a value that’s the result of the procedure’s computation. It’s legal to have procedures that do not return a value, in which case the procedures would only be executed for their “side effects”, *e.g.*, changes they make to shared data.

Here we see another procedure, COPRIMES, that invokes the GCD procedure to compute the greatest common divisor of two numbers. To use GCD, the programmer of COPRIMES only needed to know the input/output behavior of GCD, *i.e.*, the number and types of the arguments and what type of value is returned as a result. The procedural abstraction has hidden the implementation of GCD, while still making its functionality available as a “black box”.

This is a very powerful idea: encapsulating a complex computation so that it can be used by others. Every high-level language comes with a collection of pre-built procedures, called “libraries”, which can be used to perform arithmetic functions (*e.g.*, square root or cosine), manipulate collections of data

(e.g., lists or dictionaries), read data from files, and so on — the list is nearly endless! Much of the expressive power and ease-of-use provided by high-level languages comes from their libraries of “black boxes”.

The procedural abstraction is at the heart of object-oriented languages, which encapsulate data and procedures as black boxes called objects that support specific operations on their internal data. For example, a LIST object has procedures (called “methods” in this context) for indexing into the list to read or change a value, adding new elements to the list, inquiring about the length of the list, and so on. The internal representation of the data and the algorithms used to implement the methods are hidden by the object abstraction. Indeed, there may be several different LIST implementations to choose from depending on which operations you need to be particularly efficient.

Okay, enough about the virtues of the procedural abstraction! Let’s turn our attention to how to implement procedures using the Beta ISA.

Implementing Procedures

- Option 1: Inlining
 - Compiler substitutes procedure call with body
 - Problems?
 - Code size
 - Recursion
- Option 2: Linking
 - Produce separate code for each procedure
 - Caller evaluates input arguments, stores them and transfers control to the callee's entry point
 - Callee runs, stores result, transfers control to caller

```
int fact(int n) {
    if (n > 0) {
        return n*fact(n - 1);
    } else {
        return 1;
    }
}
```

A possible implementation is to “inline” the procedure, where we replace the procedure call with a copy of the statements in the procedure’s body, substituting argument values for references to the formal parameters. In this approach we’re treating procedures very much like UASM macros, *i.e.*, a simple notational shorthand for making a copy of the procedure’s body.

Are there any problems with this approach? One obvious issue is the potential increase in the code size. For example, if we had a lengthy procedure that was called many times, the final expanded code would be huge! Enough so that inlining isn’t a practical solution except in the case of short procedures where optimizing compilers do sometimes decide to inline the code.

A bigger difficulty is apparent when we consider a recursive procedure where there’s a nested call to the procedure itself. During execution the recursion will terminate for some values of the arguments and the recursive procedure will eventually return answer. But at compile time, the inlining process would not terminate and so the inlining scheme fails if the language allows recursion.

The second option is to “link” to the procedure. In this approach there is a single copy of the procedure code which we arrange to be run for each procedure call — all the procedure calls are said to link to the procedure code.

Here the body of the procedure is translated once into Beta instructions and the first instruction is identified as the procedure’s entry point. The procedure call is compiled into a set of instructions that evaluate the argument expressions and save the values in an agreed-upon location. Then we’ll use a BR instruction to transfer control to the entry point of the procedure. Recall that the BR instruction not only changes the PC but saves the address of the instruction following the branch in a specified register. This saved address is the “return address” where we want execution to resume when procedure execution is complete.

After branching to the entry point, the procedure code runs, stores the result in an agreed-upon location and then resumes execution of the calling program by jumping to the supplied return address.

Procedure Calling Convention

```
int fact(int n) {
    if (n > 0) {
        return n*fact(n - 1);
    } else {
        return 1;
    }
}

fact(3);
```

- Need **calling convention**: Uniform way to transfer data and control between procedures
- Proposed convention:
 - Pass argument (value of n) in R1
 - Pass return address in R28
 - use BR(fact,r28) to call and JMP(r28) to return
 - Return result in R0

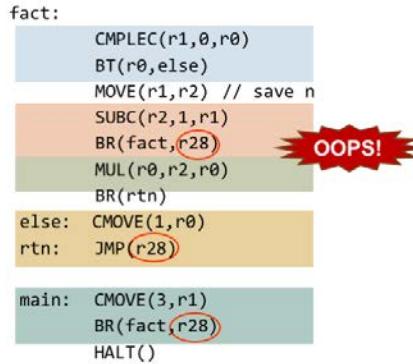
To complete this implementation plan we need a “calling convention” that specifies where to store the argument values during procedure calls and where the procedure should store the return value. It’s tempting to simply allocate specific memory locations for the job: how about using registers? We could pass the argument value in registers starting, say, with R1. The return address could be stored in another register, say R28. As we can see, with this convention the BR and JMP instructions are just what we need to implement procedure call and return. It’s usual to call the register holding the return address the “linkage pointer”. And finally the procedure can use, say, R0 to hold the return value.

Let’s see how this would work when executing the procedure call `fact(3)`. As shown on the right, `fact(3)` requires a recursive call to compute `fact(2)`, and so on. Our goal is to have a uniform calling convention where all procedure calls and procedure bodies use the same convention for storing arguments, return addresses and return values. In particular, we’ll use the same convention when compiling the recursive call `fact(n-1)` as we did for the initial call to `fact(3)`.

Procedure Linkage: First Try

```
int fact(int n) {
    if (n > 0) {
        return n*fact(n - 1);
    } else {
        return 1;
    }
}

fact(3);
```



- Proposed convention:
 - Pass argument (value of n) in R1
 - Pass return address in R28
 - Return result in R0

Okay. In the code shown on the right we've used our proposed convention when compiling the Beta code for fact(). Let's take a quick tour.

To compile the initial call fact(3) the compiler generated a CMOVE instruction to put the argument value in R1 and then a BR instruction to transfer control to fact's entry point while remembering the return address in R28.

The first statement in the body of fact tests the value of the argument using CMPLIC and BT instructions.

When n is greater than 0, the code performs a recursive call to fact, saving the value of the recursive argument n-1 in R1 as our convention requires. Note that we had to first save the value of the original argument n because we'll need it for the multiplication after the recursive call returns its value in R0.

If n is not greater than 0, the value 1 is placed in R0. Then the two possible execution paths merge, each having generated the appropriate return value in R0, and finally there's a JMP to return control to the caller. The JMP instruction knows to find the return address in R28, just where the BR put it as part of the original procedure call.

Some of you may have noticed that there are some difficulties with this particular implementation. The code is correct in the sense that it faithfully implements procedure call and return using our proposed convention. The problem is that during recursive calls we'll be overwriting register values we need later.

For example, note that following our calling convention, the recursive call also uses R28 to store the return address. When executed, the code for the original call stored the address of the HALT instruction in R28. Inside the procedure, the recursive call will store the address of the MUL instruction in R28. Unfortunately that overwrites the original return address.

Even the attempt to save the value of the argument N in R2 is doomed to fail since during the execution of the recursive call R2 will be overwritten.

The crux of the problem is that each recursive call needs to remember the value of its argument and return address, *i.e.*, we need two storage locations for each active call to fact(). And while executing fact(3), when we finally get to calling fact(0) there are four nested active calls, so we'll need $4*2 = 8$ storage locations. In fact, the amount of storage needed varies with the depth of the recursion. Obviously we can't use just two registers (R2 and R28) to hold all the values we need to save.

One fix is to disallow recursion! And, in fact, some of the early programming languages such as FORTRAN did just that. But let's see if we can solve the problem another way.

Procedure Storage Needs

- Basic requirements for procedure calls:
 - Input arguments
 - Return address
 - Results
- Local storage:
 - Variables that compiler can't fit in registers
 - Space to save caller's register values for registers that we overwrite

Each of these is specific to a particular *activation* of a procedure. We call them the procedure's *activation record*

The problem we need to solve is where to store the values needed by procedure: its arguments, its return address, its return value. The procedure may also need storage for its local variables and space to save the values of the caller's registers before they get overwritten by the procedure. We'd like to avoid any limitations on the number of arguments, the number of local variables, etc.

So we'll need a block of storage for each active procedure call, what we'll call the "activation record". As we saw in the factorial example, we can't statically allocate a single block of storage for a particular procedure since recursive calls mean we'll have many active calls to that procedure at points during the execution.

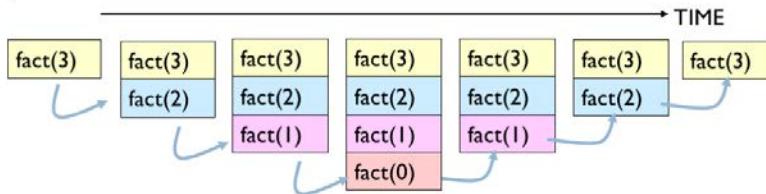
What we need is a way to dynamically allocate storage for an activation record when the procedure is called, which can then be reclaimed when the procedure returns.

Activation Records

```

int fact(int n) {
    if (n > 0) return n*fact(n - 1);
    else return 1;
}

```



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when callee finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

Let's see how activation records come and go as execution proceeds.

The first activation record is for the call `fact(3)`. It's created at the beginning of the procedure and holds, among other things, the value of the argument `n` and the return address where execution should resume after the `fact(3)` computation is complete.

During the execution of `fact(3)`, we need to make a recursive call to compute `fact(2)`. So that procedure call also gets an activation record with the appropriate values for the argument and return address. Note that the original activation record is kept since it contains information needed to complete the computation of `fact(3)` after the call to `fact(2)` returns. So now we have two active procedure calls and hence two activation records.

`fact(2)` requires computing `fact(1)`, which, in turn, requires computing `fact(0)`. At this point there are four active procedure calls and hence four activation records.

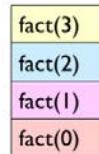
The recursion terminates with `fact(0)`, which returns the value 1 to its caller. At this point we've finished execution of `fact(0)` and so its activation record is no longer needed and can be discarded.

`fact(1)` now finishes its computation returning 1 to its caller. We no longer need its activation record. Then `fact(2)` completes, returning 2 to its caller and its activation can be discarded. And so on...

Note that the activation record of a nested procedure call is always discarded before the activation record of the caller. That makes sense: the execution of the caller can't complete until the nested procedure call returns. What we need is a storage scheme that efficiently supports the allocation and deallocation of activation records as shown here.

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records
- Activation records are allocated and deallocated in last-in-first-out (LIFO) order
- Stack: push, pop, access to top element
- For C, we only need to access to the activation record of the currently executing procedure



Early compiler writers recognized that activation records are allocated and deallocated in last-in first-out (LIFO) order. So they invented the “stack”, a data structure that implements a PUSH operation to add a record to the top of the stack and a POP operation to remove the top element. New activation records are PUSHed onto the stack during procedure calls and the POPed from the stack when the procedure call returns. Note that stack operations affect the top (*i.e.*, most recent) record on the stack.

C procedures only need to access the top activation record on the stack. Other programming languages, *e.g.* Java, support accesses to other active activation records. The stack supports both modes of operation.

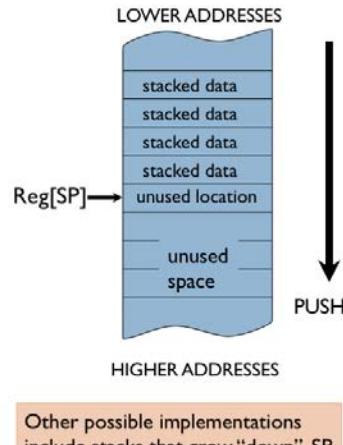
One final technical note: some programming languages support closures (*e.g.*, Javascript) or continuations (*e.g.*, Python’s yield statement), where the activation records need to be preserved even after the procedure returns. In these cases, the simple LIFO behavior of the stack is no longer sufficient and we’ll need another scheme for allocating and deallocating activation records. But that’s a topic for another course!

Stack Implementation

CONVENTIONS:

- Dedicate a register for the Stack Pointer (**SP = R29**).
- Builds *up* (towards higher addresses) on PUSH
- SP points to first **UNUSED** location; locations with addresses lower than SP have been previously allocated.
- Discipline: can use stack *at any time*; but leave it as you found it!
- Reserve a large block of memory well away from our program and its data

We use only *software conventions* to implement our stack (many architectures dedicate hardware)



Other possible implementations include stacks that grow "down", SP points to top of stack, etc.

Here's how we'll implement the stack on the Beta:

We'll dedicate one of the Beta registers, R29, to be the "stack pointer" that will be used to manage stack operations.

When we PUSH a word onto the stack, we'll increment the stack pointer. So the stack grows to successively higher addresses as words are PUSHed onto the stack.

We'll adopt the convention that SP points to (*i.e.*, its value is the address of) the first unused stack location, the location that will be filled by next PUSH. So locations with addresses lower than the value in SP correspond to words that have been previously allocated.

Words can be PUSHed to or POPped from the stack at any point in execution, but we'll impose the rule that code sequences that PUSH words onto the stack must POP those words at the end of execution. So when a code sequence finishes execution, SP will have the same value as it had before the sequence started. This is called the "stack discipline" and ensures that intervening uses of the stack don't affect later stack references.

We'll allocate a large region of memory to hold the stack located so that the stack can grow without overwriting other program storage. Most systems require that you specify a maximum stack size when running a program and will signal an execution error if the program attempts to PUSH too many items onto the stack.

For our Beta stack implementation, we'll use existing instructions to implement stack operations, so for us the stack is strictly a set of software conventions. Other ISAs provide instructions specifically for stack operations.

There are many other sensible stack conventions, so you'll need to read up on the conventions adopted by the particular ISA or programming language you'll be using.

Stack Management Macros

PUSH (RX): Push Reg[x] onto stack

Reg[SP] \leftarrow Reg[SP] + 4;
Mem[Reg[SP]-4] \leftarrow Reg[x]

ADDC(R29, 4, R29)
ST(RX, -4, R29)

POP (RX): Pop value on top of the stack into Reg[x]

Reg[x] \leftarrow Mem[Reg[SP]-4]
Reg[SP] \leftarrow Reg[SP] - 4;

LD(R29, -4, RX)
SUBC(R29, 4, R29)

ALLOCATE (k): Reserve k WORDS of stack

Reg[SP] \leftarrow Reg[SP] + 4*k

ADDC(R29, 4*k, R29)

DEALLOCATE (k): Release k WORDS of stack

Reg[SP] \leftarrow Reg[SP] - 4*k

SUBC(R29, 4*k, R29)

We've added some convenience macros to UASM to support stacks.

The PUSH macro expands into two instructions. The ADDC increments the stack pointer, allocating a new word at the top of stack, and then initializes the new top-of-stack from a specified register value with a ST instruction.

The POP macro LDs the value at the top of the stack into the specified register, then uses a SUBC instruction to decrement the stack pointer, deallocating that word from the stack.

Note that the order of the instructions in the PUSH and POP macro is very important. As we'll see in the next lecture, interrupts can cause the Beta hardware to stop executing the current program between any two instructions, so we have to be careful about the order of operations. So for PUSH, we first allocate the word on the stack, then initialize it. If we did it the other way around and execution was interrupted between the initialization and allocation, code run during the interrupt which uses the stack might unintentionally overwrite the initialized value. But, assuming all code follows stack discipline, allocation followed by initialization is always safe.

The same reasoning applies to the order of the POP instructions. We first access the top-of-stack one last time to retrieve its value, then we deallocate that location.

We can use the ALLOCATE macro to reserve a number of stack locations for later use. Sort of like PUSH but without the initialization.

DEALLOCATE performs the opposite operation, removing N words from the stack.

In general, if we see a PUSH or ALLOCATE in an assembly language program, we should be able to find the corresponding POP or DEALLOCATE, which would indicate that stack discipline is maintained.

Fun with Stacks

We can use stacks to save values we'll need later. For instance, the following code fragment can be inserted anywhere within a program.

```
// Argh!!! I'm out of registers Scotty!!
//
PUSH(R0)          // Frees up R0
PUSH(R1)          // Frees up R1
LD(dilithium_xtals, R0)
LD(seconds_til_explosion, R1)
suspense: SUBC(R1, 1, R1)
BNE(R1, suspense)
ST(R0, warp_engines)
POP(R1)           // Restores R1
POP(R0)           // Restores R0
```

Data is popped off the stack in the opposite order that it is pushed on

Next, we'll use show how to use stacks for activation records..

We'll use stacks to save values we'll need later. For example, if we need to use some registers for a computation but don't know if the register's current values are needed later in the program, we can PUSH their current values onto the stack and then are free to use the registers in our code. After we're done, we can use POP to restore the saved values.

Note that we POP data off the stack in the opposite order that the data was PUSHed, *i.e.*, we need to follow the last-in first-out discipline imposed by the stack operations.

Now that we have the stack data structure, we'll use it to solve our problems with allocating and deallocating activation records during procedure calls.

Solving Procedure Linkage Problems

Reminder: Procedure storage needs

- 1) We need a way to *pass arguments* to the procedure
- 2) Procedures need their own *LOCAL storage*
- 3) Procedures need to *call other procedures*; special case: recursive procedures that *call themselves*

Plan for caller:

- Push argument values onto stack *in reverse order* for use by callee
- Branch to callee, save return address in dedicated register (**LP = R28**)
- Clean up stack after callee return

C code:

proc(expr₁, ..., expr_n)

Beta assembly:

```
compile_expr(exprn)⇒Rx  
PUSH(rx)  
...  
compile_expr(expr1)⇒Rx  
PUSH(rx)  
BR(proc, LP)  
DEALLOCATE(n)
```

We'll use the stack to hold a procedure's activation record. That includes the values of the arguments to the procedure call. We'll allocate words on the stack to hold the values of the procedure's local variables, assuming we don't keep them in registers. And we'll use the stack to save the return address (passed in LP) so the procedure can make nested procedure calls without overwriting its return address.

The responsibility for allocating and deallocating the activation record will be shared between the calling procedure (the "caller") and the called procedure (the "callee").

The caller is responsible for evaluating the argument expressions and saving their values in the activation record being built on the stack. We'll adopt the convention that argument values are pushed in reverse order, *i.e.*, the first argument will be the last to be pushed on the stack. We'll explain why we made this choice in a couple of slides...

The code compiled for a procedure involves a sequence of expression evaluations, each followed by a PUSH to save the calculated value on the stack. So when the callee starts execution, the top of the stack contains the value of the first argument, the next word down the value of the second argument, and so on.

After the argument values, if any, have been pushed on the stack, there's a BR to transfer control to the procedure's entry point, saving the address of the instruction following the BR in the linkage pointer, R28, a register that we'll dedicate to that task.

When the callee returns and execution resumes in the caller, a DEALLOCATE is used to remove all the argument values from the stack, preserving stack discipline.

So that's the code the compiler generates for the procedure. The rest of the work happens in the called procedure.

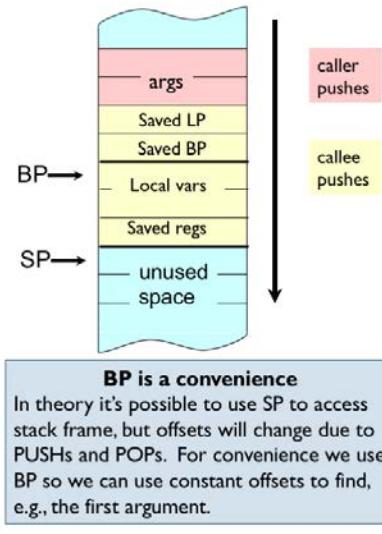
Stack Frames as Activation Records

CALLEE uses stack for all of the its storage needs:

1. Saving return address back to the caller (it's in LP)
2. Saving BP of caller (pointer to caller's activation record)
3. Allocating stack locations to hold local variables
4. Save any registers callee uses: "callee saves" convention

Dedicate another register (**BP = R27**) to hold address of the activation record. Use when accessing

- Arguments
- Other local storage



BP is a convenience
In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHes and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

The code at the start of the called procedure completes the allocation of the activation record. Since when we're done the activation record will occupy a bunch of consecutive words on the stack, we'll sometimes refer to the activation record as a "stack frame" to remind us of where it lives.

The first action is to save the return address found in the LP register. This frees up LP to be used by any nested procedure calls in the body of the callee.

In order to make it easy to access values stored in the activation record, we'll dedicate another register called the "base pointer" (BP = R27) which will point to the stack frame we're building. So as we enter the procedure, the code saves the pointer to the caller's stack frame, and then uses the current value of the stack pointer to make BP point to the current stack frame. We'll see how we use BP in just a moment.

Now the code will allocate words in the stack frame to hold the values for the callee's local variables, if any.

Finally, the callee needs to save the values of any registers it will use when executing the rest of its code. These saved values can be used to restore the register values just before returning to the caller. This is called the "callee saves" convention where the callee guarantees that all register values will be preserved across the procedure call. With this convention, the code in the caller can assume any values it placed in registers before a nested procedure call will still be there when the nested call returns.

Note that dedicating a register as the base pointer isn't strictly necessary. All accesses to the values on the stack can be made relative to the stack pointer, but the offsets from SP will change as values are PUSHed and POPed from the stack, e.g., during procedure calls. It will be easier to understand the generated code if we use BP for all stack frame references.

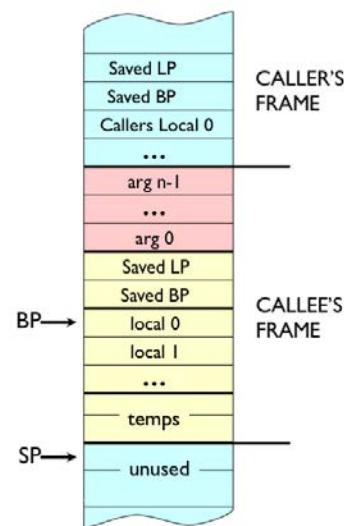
Stack Frame Details

CALLER passes arguments to CALLEE on the stack in *reverse* order

F(1,2,3,4) translates to:

```
CMOVE(4,R0)  
PUSH(R0)  
CMOVE(3,R0)  
PUSH(R0)  
CMOVE(2,R0)  
PUSH(R0)  
CMOVE(1,R0)  
PUSH(R0)  
BR(F, LP)
```

Why push args in REVERSE order?



Let's return to the question about the order of argument values in the stack frame. We adopted the convention of PUSHing the values in reverse order, *i.e.*, where the value of the first argument is the last one to be PUSHED.

So, why PUSH argument values in reverse order?

Argument Order & BP sage

Why push args in reverse order? It allows the BP to serve double duties when accessing the local frame

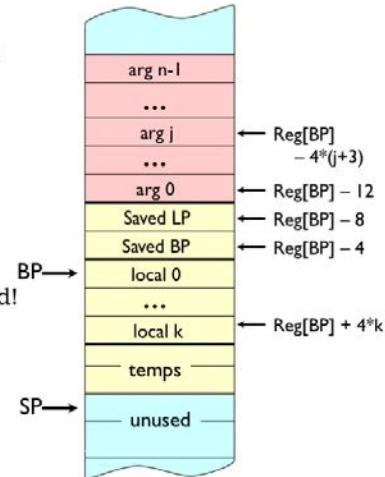
1) To access j^{th} argument ($j \geq 0$):

`LD(BP, -4*(j+3), rx)`
or
`ST(rx, -4*(j+3), BP)`

CALLEE can access the first few arguments without knowing how many arguments have been passed!

2) To access k^{th} local variable ($k \geq 0$)

`LD(BP, k*4, rx)`
or
`ST(rx, k*4, BP)`



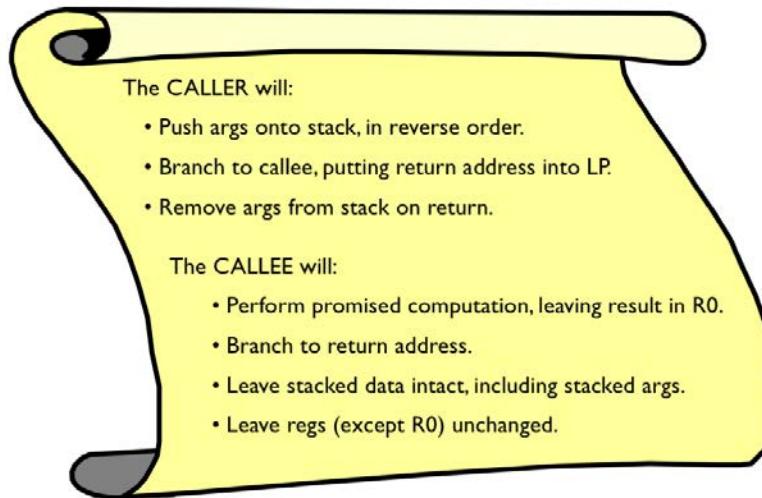
With the arguments PUSHed in reverse order, the first argument (labeled “arg 0”) will be at a fixed offset from the base pointer regardless of the number of argument values pushed on the stack. The compiler can use a simple formula to determine the correct BP offset value for any particular argument. So the first argument is at offset -12, the second at -16, and so on.

Why is this important? Some languages, such as C, support procedure calls with a variable number of arguments. Usually the procedure can determine from, say, the first argument, how many additional arguments to expect. The canonical example is the C printf function where the first argument is a format string that specifies how a sequence of values should be printed. So a call to printf includes the format string argument plus a varying number of additional arguments. With our calling convention the format string will always be in the same location relative to BP, so the printf code can find it without knowing the number of additional arguments in the current call.

The local variables are also at fixed offsets from BP. The first local variable is at offset 0, the second at offset 4, and so on.

So we see that having a base pointer makes it easy to access the values of the arguments and local variables using fixed offsets that can be determined at compile time. The stack above the local variables is available for other uses, e.g., building the activation record for a nested procedure call!

Procedure Linkage: The Contract



Okay, here's our final contract for how procedure calls will work:

The calling procedure (“caller”) will

PUSH the argument values onto the stack in reverse order

Branch to the entry point of the callee, putting the return address into the linkage pointer.

When the callee returns, remove the argument values from the stack.

The called procedure (“callee”) will

Perform the promised computation, leaving the result in R0.

Jump to the return address when the computation has finished

Remove any items it has placed on the stack, leaving the stack as it was when the procedure was entered. Note that the arguments were PUSHed on the stack by the caller, so it will be up to the caller to remove them.

Preserve the values in all registers except R0, which holds the return value. So the caller can assume any values placed in registers before a nested call will be there when the nested call returns.

Procedure Linkage Templates

Calling Sequence	PUSH(arg _n) ... PUSH(arg ₁) BR(f, LP) DEALLOCATE(n) ...	// push args, last arg first // Call f. // Clean up! // (f's return value in R0)
Entry Sequence	f: PUSH(LP) PUSH(BP) MOVE(SP,BP) ALLOCATE(nlocals) (push other regs)	// Save LP and BP // in case we make new calls. // set BP=frame base // allocate locals // preserve any regs used
Exit Sequence	// return value in R0... (pop other regs) MOVE(BP,SP) Why no POP(BP) DEALLOCATE? POP(LP) JMP(LP)	// restore regs // strip locals, etc // restore CALLER's linkage // (the return address) // return.

We saw the code template for procedure calls on an earlier slide.

Here's the template for the entry point to a procedure F. The code saves the caller's LP and BP values, initializes BP for the current stack frame and allocates words on the stack to hold any local variable values. The final step is to PUSH the value of any registers (besides R0) that will be used by the remainder of the procedure's code.

The template for the exit sequence mirrors the actions of the entry sequence, restoring all the values saved by the entry sequence, performing the POP operations in the reverse of the order of the PUSH operations in the entry sequence. Note that in moving the BP value into SP we've reset the stack to its state at the point of the MOVE(SP,BP) in the entry sequence. This implicitly undoes the effect of the ALLOCATE statement in the entry sequence, so we don't need a matching DEALLOCATE in the exit sequence.

The last instruction in the exit sequence transfers control back to the calling procedure.

With practice you'll become familiar with these code templates. Meanwhile, you can refer back to this slide whenever you need to generate code for a procedure call.

Putting It All Together: Factorial

```
fact:    PUSH(LP)           // save linkages
         PUSH(BP)
         MOVE(SP,BP)        // new frame base
         PUSH(r1)            // preserve regs
         LD(BP,-12,r1)       // r1 ← n
         CMPLEC(r1,0,r0)     // if (n > 0)
         BT(r0,else)

int fact(int n) {
    if (n > 0) {
        return n*fact(n-1);
    } else {
        return 1;
    }
}

         SUBC(r1,1,r1)      // r1 ← (n-1)
         PUSH(r1)            // push arg1
         BR(fact,LP)          // fact(n-1)
         DEALLOCATE(1)        // pop arg1
         LD(BP,-12,r1)       // r1 ← n
         MUL(r1,r0,r0)        // r0 ← n*fact(n-1)
         BR(rtn)

         else:   CMOVE(1,r0)   // return 1

rtn:    POP(r1)             // restore regs
        MOVE(BP,SP)          // Why?
        POP(BP)               // restore links
        POP(LP)
        JMP(LP)               // return
```

Here's the code our compiler would generate for the C implementation of factorial shown on the left.

The entry sequence saves the caller's LP and BP, then initializes BP for the current stack frame. The value of R1 is saved so we can use R1 in code that follows.

The exit sequence restores all the saved values, including that for R1. The code for the body of the procedure has arranged for R0 to contain the return value by the time execution reaches the exit sequence.

The nested procedure call passes the argument value on the stack and removes it after the nested call returns.

The remainder of the code is generated using the templates we saw in the previous lecture. Aside from computing and pushing the values of the arguments, there are approximately 10 instructions needed to implement the linking approach to a procedure call. That's not much for a procedure of any size, but might be significant for a trivial procedure. As mentioned earlier, some optimizing compilers can make the tradeoff of inlining small non-recursive procedures saving this small amount of overhead.

Recursion?

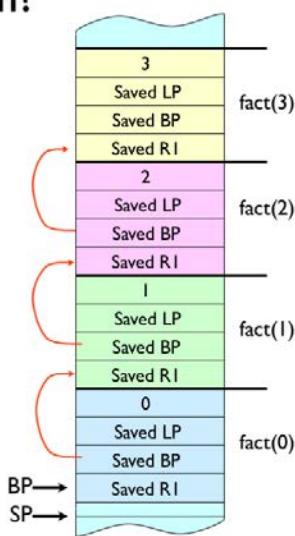
Of course!

- Frames allocated for each recursive call...
- Deallocated (in inverse order) as recursive calls return

Debugging skill:
“stack crawling”

- Given code, stack snapshot – figure out what, where, how, who...
- Follow old <BP> links to parse frames
- Decode args, locals, return locations, etc etc etc

Particularly useful on 6.004 quizzes!

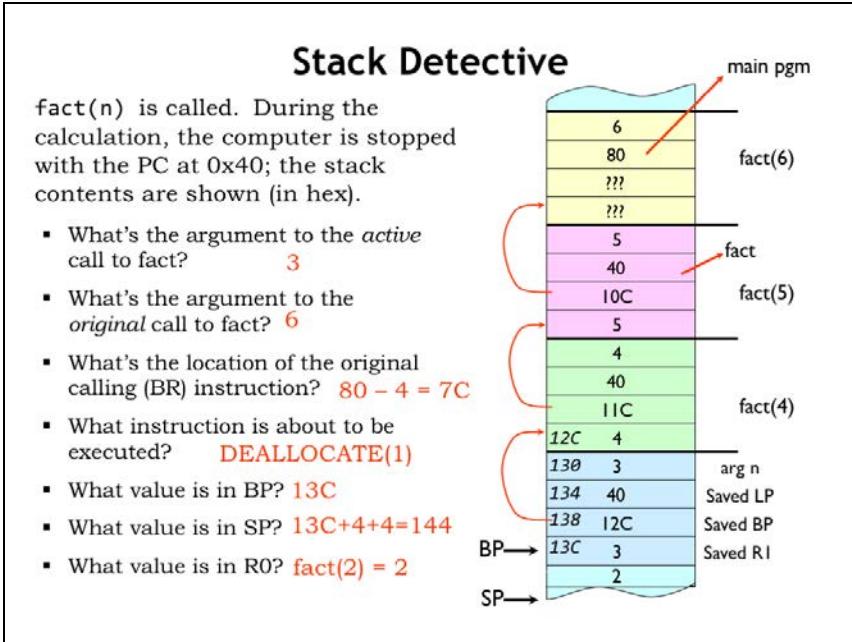


So have we solved the activation record storage issue for recursive procedures?

Yes! A new stack frame is allocated for each procedure call. In each frame we see the storage for the argument and return address [CLICK]. And as the nested calls return the stack frames will be deallocated in inverse order.

Interestingly we can learn a lot about the current state of execution by looking at the active stack frames. The current value of BP, along the older values saved in the activation records, allow us to identify the active procedure calls and determine their arguments, the values of any local variables for active calls, and so on. If we print out all this information at any given time we would have a “stack trace” showing the progress of the computation. In fact, many language runtimes will print out the stack trace to help the programmer determine what happened.

And, of course, if you can interpret the information in the stack frames, you can show you understand our conventions for procedure call and return. Don’t be surprised to find such a question on a quiz :)



Let's practice our newfound skill and see what we can determine about a running program which we've stopped somewhere in the middle of its execution. We're told that a computation of `fact()` is in progress and that the PC of the next instruction to be executed is `0x40`. We're also given the stack dump shown on right.

Since we're in the middle of a `fact` computation, we know that current stack frame (and possibly others) is an activation record for the `fact` function. Using the code on the previous slide we can determine the layout of the stack frame and generate the annotations shown on the right of the stack dump. With the annotations, it's easy to see that the argument to current active call is the value 3.

Now we want to know the argument to original call to `fact`. We'll have to label the other stack frames using the saved BP values. Looking at the saved LP values for each frame (always found at an offset of -8 from the frame's BP), we see that many of the saved values are `0x40`, which must be the return address for the recursive `fact` calls.

Looking through the stack frames we find the first return address that's **not** `0x40`, which must be a return address to code that's not part of the `fact` procedure. This means we've found the stack frame created by the original call to `fact` and can see that argument to the original call is 6.

What's the location of the BR that made the original call? Well the saved LP in the stack frame of the original call to `fact` is `0x80`. That's the address of the instruction following the original call, so the BR that made the original call is one instruction earlier, at location `0x7C`. To answer these questions you have to be good at hex arithmetic!

What instruction is about to be executed? We were told its address is `0x40`, which we notice is the saved LP value for all the recursive `fact` calls. So `0x40` must be the address of the instruction following the BR(`fact`,`LP`) instruction in the `fact` code. Looking back a few slides at the `fact` code, we see that's a `DEALLOCATE(1)` instruction.

What value is in BP? Hmm. We know BP is the address of the stack location containing the saved RI value in the current stack frame. So the saved BP value in the current stack frame is the address of the

saved R1 value in the *previous* stack frame. So the saved BP value gives us the address of a particular stack location, from which we can derive the address of all the other locations! Counting forward, we find that the value in BP must be 0x13C.

What value is in SP? Since we're about to execute the DEALLOCATE to remove the argument of the nested call from the stack, that argument must still be on the stack right after the saved R1 value. Since the SP points to first unused stack location, it points to the location after that word, so it has the value 0x144.

Finally, what value is in R0? Since we've just returned from a call to fact(2) the value in R0 must be the result from that recursive call, which is 2.

Wow! You can learn a lot from the stacked activation records and a little deduction! Since the state of the computation is represented by the values of the PC, the registers, and main memory, once we're given that information we can tell exactly what the program has been up to. Pretty neat...

Summary of Dedicated Registers

The Beta ISA has 32 registers. But we've dedicated several of them to serve a specific purpose:

- R31 is always zero [ISA]
- R30 ... reserved for future use... [next lecture]
- R29 = SP, stack pointer [software convention]
- R28 = LP, linkage pointer [software convention]
- R27 = BP, base pointer [software convention]

Wrapping up, we've been dedicating some registers to help with our various software conventions. To summarize:

R31 is always zero, as defined by the ISA.

We'll also dedicate R30 to a particular function in the ISA when we discuss the implementation of the Beta in the next lecture. Meanwhile, don't use R30 in your code!

The remaining dedicated registers are connected with our software conventions:

R29 (SP) is used as the stack pointer,

R28 (LP) is used as the linkage pointer, and

R27 (BP) is used as the base pointer.

As you practice reading and writing code, you'll grow familiar with these dedicated registers.

Summary

- Each procedure invocation has an activation record
 - Created during procedure call/entry sequence
 - Discarded when procedure returns
 - Holds:
 - Argument values (in reverse order)
 - Saved LP, BP from caller (callee reuses those regs)
 - Storage for local variables (if any)
 - Other saved regs from caller (callee needs regs to use)
 - BP points to activation record of active call
 - Access arguments at offsets of -12, -16, -20, ...
 - Access local variables at offsets of 0, 4, 8, ...
- “Callee saves” convention: all reg values preserved
- Except for R0, which holds return value

In thinking about how to implement procedures, we discovered the need for an activation record to store the information needed by any active procedure call.

An activation record is created by the caller and callee at the start of a procedure call. And the record can be discarded when the procedure is complete.

The activation records hold argument values, saved LP and BP values along with the caller’s values in any other of the registers. Storage for the procedure’s local variables is also allocated in the activation record.

We use BP to point to the current activation record, giving easy access the values of the arguments and local variables.

We adopted a “callee saves” convention where the called procedure is obligated to preserve the values in all registers except for R0.

Taken together, these conventions allow us to have procedures with arbitrary numbers of arguments and local variables, with nested and recursive procedure calls. We’re now ready to compile and execute any C program!

13: Building the Beta

1. CPU Design Tradeoffs
2. Processor Performance
3. Reminder: Beta ISA
4. Approach: Incremental Featurism
5. Multi-ported Register File
6. Register File Timing
7. ALU Instructions
8. Instruction Fetch/Decode
9. ALU Op Datapath I
10. ALU Op Datapath II
11. ALU Operations (with constant) I
12. ALU Operations (with constant) II
13. Load Instruction I
14. Load Instruction II
15. Store Instruction I
16. Store Instruction II
17. JMP Instruction I
18. JMP Instruction II
19. BEQ/BNE Instructions I
20. BEQ/BNE Instructions II
21. Load Relative Instruction
22. LDR Instruction I
23. LDR Instruction II
24. Exceptions
25. Exception Processing
26. Exception Implementation
27. Exceptions I
28. Exceptions II
29. Beta: Our “Final Answer”
30. Control Logic
31. Beta Inside!

CPU Design Tradeoffs



Maximum Performance: measured by the numbers of instructions executed per second



Minimum Cost : measured by the size of the circuit.



Best Performance/Price: measured by the ratio of MIPS to size. In power-sensitive applications MIPS/Watt is important too.

Today we're going to describe the datapath and control logic needed to execute Beta instructions. In an upcoming lab assignment, we'll ask you to build a working implementation using our standard cell library. When you're done, you'll have designed and debugged a 32-bit reduced-instruction set computer! Not bad...

Before tackling a design task, it's useful to understand the goals for the design. Functionality, of course; in our case the correct execution of instructions from the Beta ISA. But there are other goals we should think about.

An obvious goal is to maximize performance, as measured by the number of instructions executed per second. This is usually expressed in MIPS, an acronym for "Millions of Instructions Per Second". When the Intel 8080 was introduced in 1974, it executed instructions at 0.29 MIPS or 290,000 instructions per second as measured by the Dhrystone benchmark. Modern multi-core processors are rated between 10,000 and 100,000 MIPS.

Another goal might be to minimize the manufacturing cost, which in integrated circuit manufacturing is proportional to the size of the circuit.

Or we might want have the best performance for a given price. In our increasingly mobile world, the best performance per watt might be an important goal.

One of the interesting challenges in computer engineering is deciding exactly how to balance performance against cost and power efficiency. Clearly the designers of the Apple Watch have a different set of design goals than the designers of high-end desktop computers.

Processor Performance

- “Iron Law” of performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}} \quad \text{Perf} = \frac{1}{\text{Time}}$$

- Options to reduce execution time:
 - Executed instructions ↓ (work/instruction ↑)
 - Cycles per instruction (CPI) ↓
 - Cycle time ↓ (frequency ↑)
- Today: Simple, CPI=1 but low-frequency Beta
 - Later: Pipelining to increase frequency

The performance of a processor is inversely proportional to the length of time it takes to run a program. The shorter the execution time, the higher the performance. The execution time is determined by three factors.

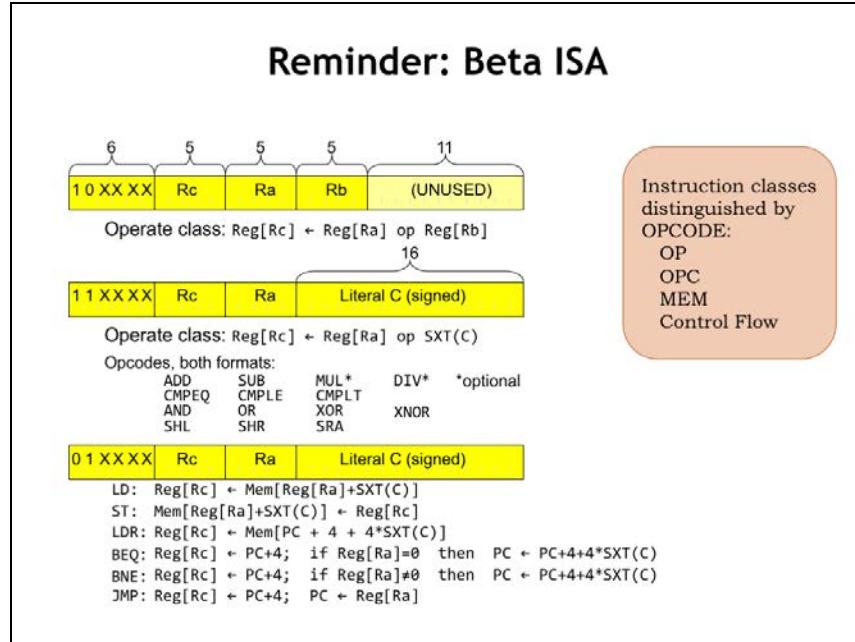
First, the number of instructions in the program.

Second, the number of clock cycles our sequential circuit requires to execute a particular instruction. Complex instructions, *e.g.*, adding two values from main memory, may make a program shorter, but may also require many clock cycles to perform the necessary memory and datapath operations.

Third, the amount of time needed for each clock cycle, as determined by the propagation delay of the digital logic in the datapath.

So to increase the performance we could reduce the number of instructions to be executed. Or we can try to minimize the number of clock cycles needed on the average to execute our instructions. There's obviously a bit of a tradeoff between these first two options: more computation per instruction usually means it will take more time to execute the instruction. Or we can try to keep our logic simple, minimizing its propagation delay in the hopes of having a short clock period.

Today we'll focus on an implementation for the Beta ISA that executes one instruction every clock cycle. The combinational paths in our circuit will be fairly long, but, as we learned in Part 1 of the course, this gives us the opportunity to use pipelining to increase our implementation's throughput. We'll talk about the implementation of a pipelined processor in some upcoming lectures.



Here's a quick refresher on the Beta ISA. The Beta has thirty-two 32-bit registers that hold values for use by the datapath. The first class of ALU instructions, which have 0b10 as the top 2 bits of the opcode field, perform an operation on two register operands (Ra and Rb), storing the result back into a specified destination register (Rc). There's a 6-bit opcode field to specify the operation and three 5-bit register fields to specify the registers to use as source and destination. The second class of ALU instructions, which have 0b11 in the top 2 bits of the opcode, perform the same set of operations where the second operand is constant in the range -32768 to +32767.

The operations include arithmetic operations, comparisons, boolean operations, and shifts. In assembly language, we use a "C" suffix added to the mnemonics shown here to indicate that the second operand is a constant.

This second instruction format is also used by the instructions that access memory and change the normally sequential execution order.

The use of just two instruction formats will make it very easy to build the logic responsible for translating the encoded instructions into the signals needed to control the operation of the datapath. In fact, we'll be able to use many of the instruction bits as-is!

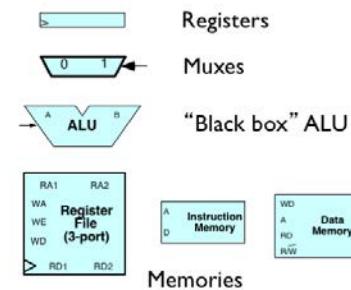
Approach: Incremental Featurism

We'll implement datapaths for each instruction class individually, and merge them (using MUXes, etc)

Steps:

1. ALU instructions
2. Load & store instructions
3. Jump & branch instructions
4. Exceptions

Component Repertoire:



We'll build our datapath incrementally, starting with the logic needed to perform the ALU instructions, then add additional logic to execute the memory and branch instructions. Finally, we'll need to add logic to handle what happens when an exception occurs and execution has to be suspended because the current instruction cannot be executed correctly.

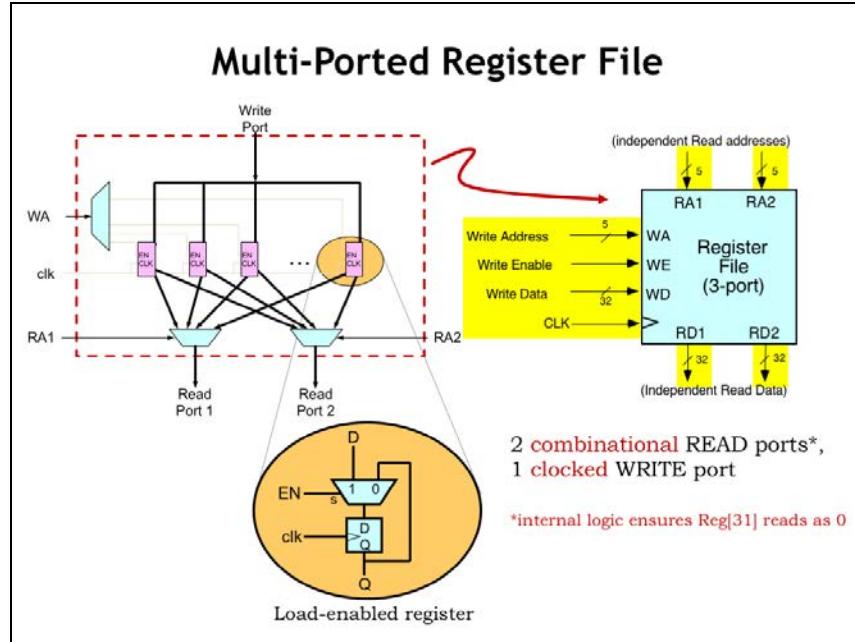
We'll be using the digital logic gates we learned about in Part 1 of the course. In particular, we'll need multi-bit registers to hold state information from one instruction to the next. Recall that these memory elements load new values at the rising edge of the clock signal, then store that value until the next rising clock edge.

We'll use a lot of multiplexers in our design to select between alternative values in the datapath.

The actual computations will be performed by the arithmetic and logic unit (ALU) that we designed at the end of Part 1. It has logic to perform the arithmetic, comparison, boolean and shift operations listed on the previous slide. It takes in two 32-bit operands and produces a 32-bit result.

And, finally, we'll use several different memory components to implement register storage in the datapath and also for main memory, where instructions and data are stored.

You might find it useful to review the chapters on combinational and sequential logic in Part 1 of the course.



The Beta ISA specifies thirty-two 32-bit registers as part of the datapath. These are shown as the magenta rectangles in the diagram below. These are implemented as load-enabled registers, which have an EN signal that controls when the register is loaded with a new value. If EN is 1, the register will be loaded from the D input at the next rising clock edge. If EN is 0, the register is reloaded with its current value and hence its value is unchanged. It might seem easier to add enabling logic to the clock signal, but this is almost never a good idea since any glitches in that logic might generate false edges that would cause the register to load a new value at the wrong time. Always remember the mantra: NO GATED CLOCKS!

There are multiplexers (shown underneath the registers) that let us select a value from any of the 32 registers. Since we need two operands for the datapath logic, there are two such multiplexers. Their select inputs (RA1 and RA2) function as addresses, determining which register values will be selected as operands. And, finally, there's a decoder that determines which of the 32 register load enables will be 1 based on the 5-bit WA input.

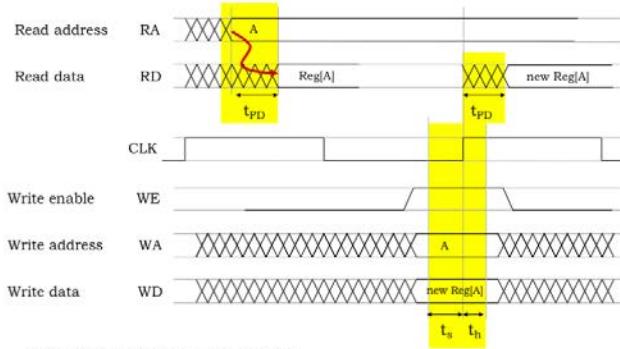
For convenience, we'll package all this functionality up into a single component called a "register file". The register file has two read ports, which given a 5-bit address input, deliver the selected register value on the read-data ports. The two read ports operate independently. They can read from different registers or, if the addresses are the same, read from the same register.

The signals on the left of the register file include a 5-bit value (WA) that selects a register to be written with the specified 32-bit write data (WD). If the write enable signal (WE) is 1 at the rising edge of the clock (CLK) signal, the selected register will be loaded with the supplied write data. [CLICK]

Note that in the BETA ISA, reading from register address 31 should always produce a zero value. The register file has internal logic to ensure that happens.

Register File Timing

2 combinational READ ports, 1 clocked WRITE port



Here's a timing diagram that shows the register file in operation. To read a value from the register file, supply a stable address input (RA) on one of read ports. After the register file's propagation delay, the value of the selected register will appear on the corresponding read data port (RD). [CLICK]

Not surprisingly, the register file write operation is very similar to writing an ordinary D-register. The write address (WA), write data (WD) and write enable (WE) signals must all be valid and stable for some specified setup time before the rising edge of the clock. And must remain stable and valid for the specified hold time after the rising clock edge. If those timing constraints are met, the register file will reliably update the value of the selected register. [CLICK]

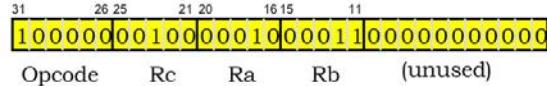
When a register value is written at the rising clock edge, if that value is selected by a read address, the new data will appear after the propagation delay on the corresponding data port. In other words, the read data value changes if either the read address changes or the value of the selected register changes. [CLICK]

Can we read and write the same register in a single clock cycle? Yes! If the read address becomes valid at the beginning of the cycle, the old value of the register will appear on the data port for the rest of the cycle. Then, the write occurs at the *end* of the cycle and the new register value will be available in the next clock cycle.

Okay, that's a brief run-through of the components we'll be using. Let's get started on the design!

ALU Instructions

32-bit (4-byte) ADD instruction:



Means, to Beta, $Reg[R4] \leftarrow Reg[R2] + Reg[R3]$

Need hardware to:

- **FETCH** (read) 32-bit instruction for the current cycle
- **DECODE** instruction: ADD, SUB, XOR, etc
- **READ** operands (Ra, Rb) from Register File
- **EXECUTE** operation
- **WRITE-BACK** result into Register File (Rc)

Our first task is to work on the datapath logic needed to execute ALU instructions with two register operands. Each instruction requires the same processing steps:

Fetch, where the 32-bit encoded instruction is read from main memory from the location specified by the program counter (PC).

Decode, where the opcode field (instruction bits [31:26]) is used to determine the values for the datapath control signals.

Read, where the contents of the registers specified by the RA and RB fields (instruction bits [20:16] and [15:11]) are read from the register file.

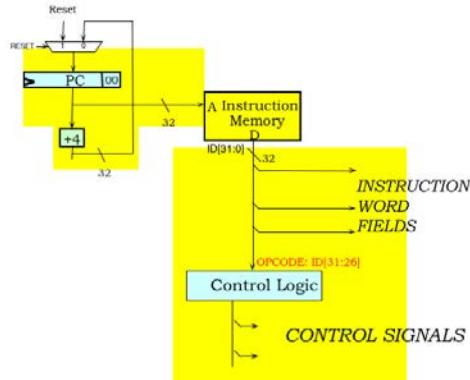
Execute, where the requested operation is performed on the two operand values. We'll also need to compute the next value for the PC.

And Write-back, where the result of the operation is written to the register file in the register specified by the RC field (instruction bits [25:21]).

The system's clock signal is connected to the register file and the PC register. At the rising edge of the clock, the new values computed during the Execute phase are written to these registers. The rising clock edge thus marks the end of execution for the current instruction and the beginning of execution for the next instruction. The period of the clock, *i.e.*, the time between rising clock edges, needs to be long enough to accommodate the cumulative propagation delay of the logic that implements the 5 steps described here. Since one instruction is executed each clock cycle, the frequency of the clock tells us the rate at which instructions are executed. If the clock period was 10ns, the clock frequency would be 100 MHz and our Beta would be executing instructions at 100 MIPS!

Instruction Fetch/Decode

Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)



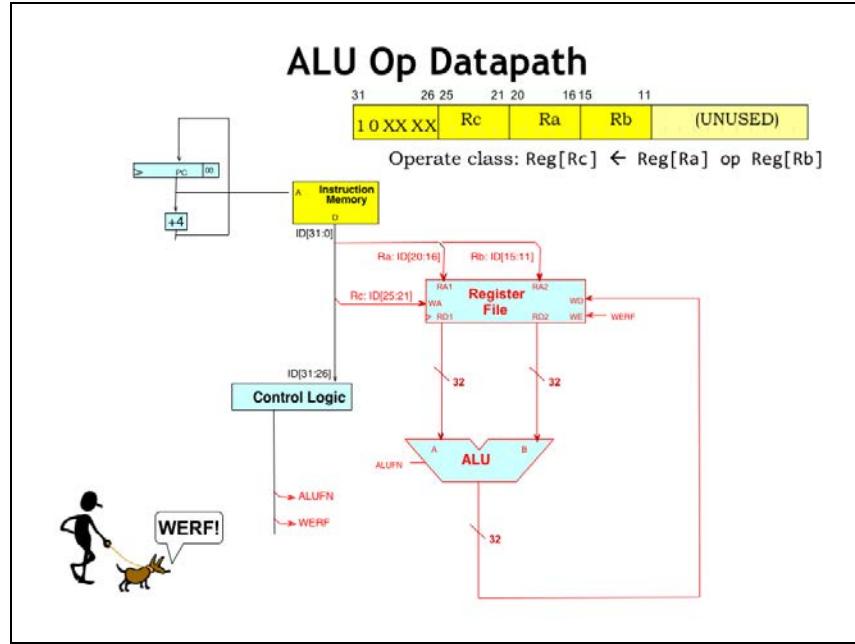
- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
 - Use some instruction fields directly (register numbers, 16-bit constant)
 - Use bits [31:26] to generate control signals

Here's a sketch showing the hardware needed for the Fetch and Decode steps.

The current value of the PC register is routed to main memory as the address of the instruction to be fetched.

For ALU instructions, the address of the next instruction is simply the address of the current instruction plus 4. There's an adder dedicated to performing the "PC+4" computation and that value is routed back to be used as the next value of the PC. We've also included a MUX used to select the initial value for the PC when the RESET signal is 1.

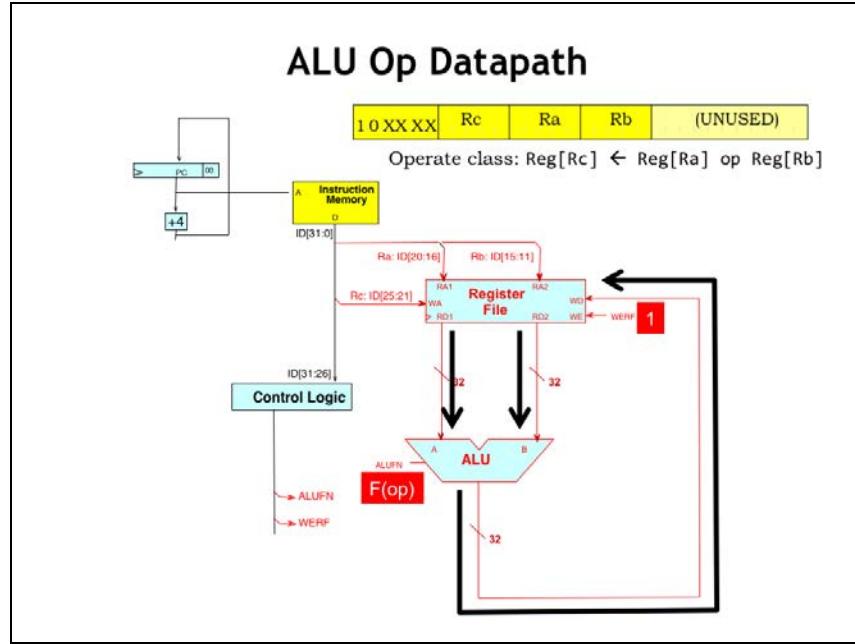
After the memory propagation delay, the instruction bits (ID[31:0]) are available and the processing steps can begin. Some of the instruction fields can be used directly as-is. To determine the values for other control signals, we'll need some logic that computes their values from the bits of the opcode field. [CLICK]



Now let's fill in the datapath logic needed to execute ALU instructions with two register operands. The instruction bits for the 5-bit RA, RB and RC fields can be connected directly to the appropriate address inputs of the register file. The RA and RB fields supply the addresses for the two read ports and the RC field supplies the address for the write port.

The outputs of the read data ports are routed to the inputs of the ALU to serve as the two operands. The ALUFN control signals tell the ALU what operation to perform. These control signals are determined by control logic from the 6-bit opcode field. For specificity, let's assume that the control logic is implemented using a read-only memory (ROM), where the opcode bits are used as the ROM's address and the ROM's outputs are the control signals. Since there are 6 opcode bits, we'll need $2^6 = 64$ locations in the ROM. We'll program the contents of the ROM to supply the correct control signal values for each of the 64 possible opcodes.

The output of the ALU is routed back to the write data port of the register file, to be written into the RC register at the end of the cycle. We'll need another control signal, WERF ("write-enable register file"), that should have the value 1 when we want to write into the RC register. Let me introduce you to Werf, the 6.004 mascot, who, of course, is named after her favorite control signal, which she's constantly mentioning.

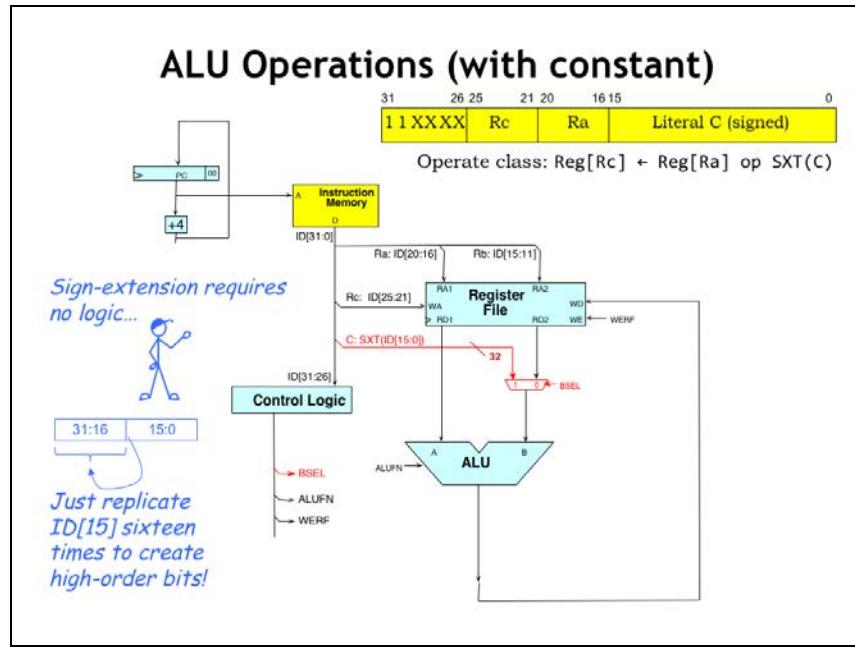


Let's follow the flow of data as we execute the ALU instruction. After the instruction has been fetched, supplying the RA and RB instruction fields, the RA and RB register values appear on the read data ports of the register file.

The control logic has decoded the opcode bits and supplied the appropriate ALU function code. You can find a listing of the possible function codes in the upper right-hand corner of the Beta Diagram handout.

The result of the ALU's computation is sent back to the register file to be written into the RC register. Of course, we'll need to set WERF to 1 to enable the write.

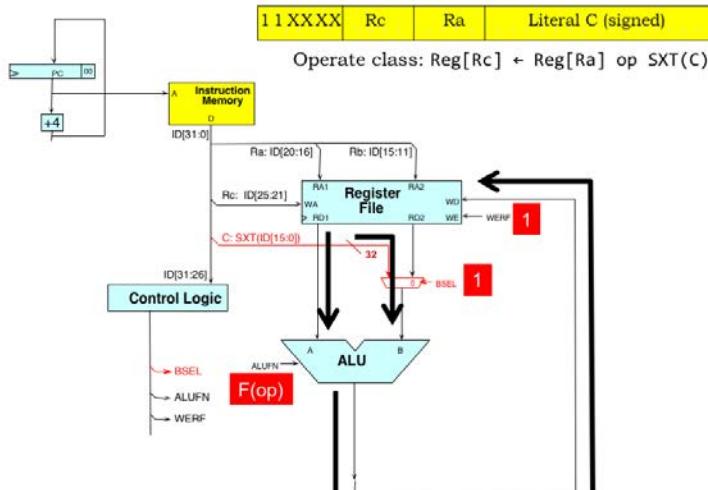
Here we see one of the major advantages of a reduced-instruction set computer architecture: the datapath logic required for execution is very straightforward!



The other form of ALU instructions uses a constant as the second ALU operand. The 32-bit operand is formed by sign-extending the 16-bit two's complement constant stored in the literal field (bits [15:0]) of the instruction. In order to select the sign-extended constant as the second operand, we added a MUX to the datapath. When its BSEL control signal is 0, the output of the register file is selected as the operand. When BSEL is 1, the sign-extended constant is selected as the operand. The rest of the datapath logic is the same as before.

Note that no logic gates are needed to perform sign-extension — it's all done with wiring! To sign-extend a two's complement number, we just need to replicate the high-order, or sign, bit as many times as necessary. You might find it useful to review the discussion of two's complement in Lecture 1 of Part 1 of the course. So to form a 32-bit operand from a 16-bit constant, we just replicate its high-order bit (ID[15]) sixteen times as we make the connection to the BSEL MUX.

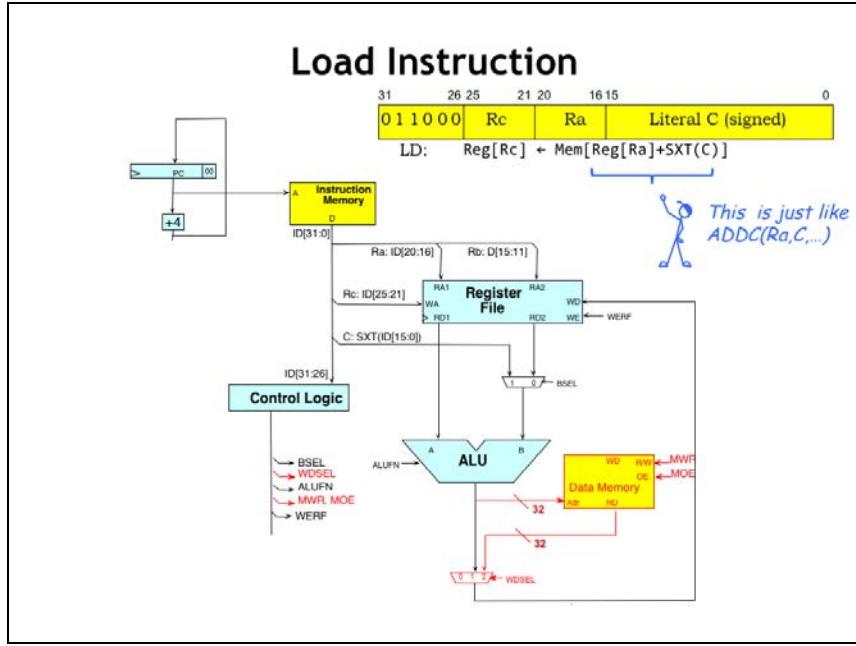
ALU Operations (with constant)



During execution of ALU-with-constant instructions, the flow of data is much as it was before. The one difference is that the control logic sets the BSEL control signal to 1, selecting the sign-extended constant as the second ALU operand.

As before, the control logic generates the appropriate ALU function code and the output of the ALU is routed to the register file to be written back to the RC register.

Amazingly, this datapath is sufficient to execute most of the instructions in the Beta ISA! We just have the memory and branch instruction left to implement. That's our next task.



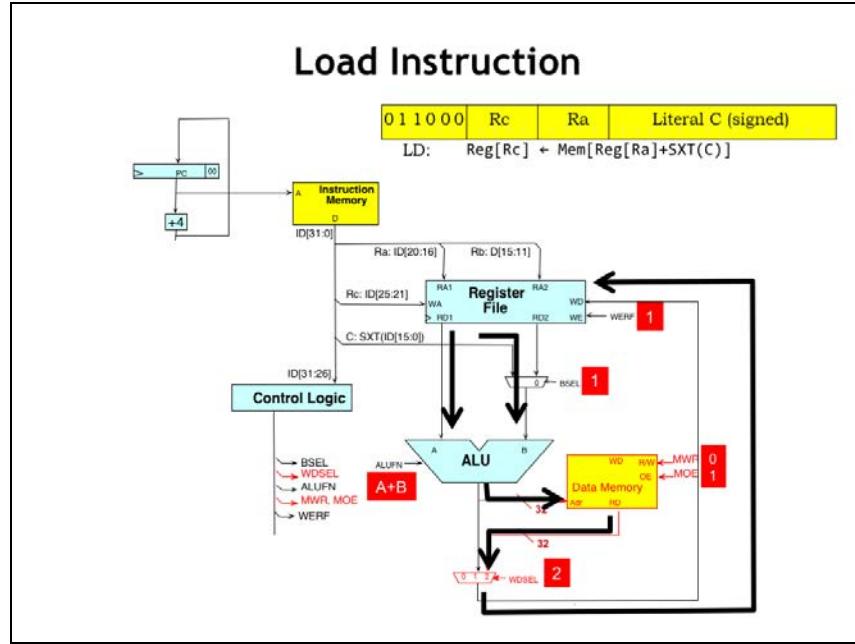
The LD and ST instructions access main memory. Note that its the same main memory that holds the instructions, even though for drafting convenience we show it has two separate boxes in our datapath diagram. In the form we show it here, main memory has three ports: two read ports for fetching instructions and reading load data, and one write port used by the ST instruction to write data into main memory.

The address calculation is exactly the same computation as performed by the ADDC instruction: the contents of the RA register are added to the sign-extended 16-bit literal from the low-order 16 bits of the instruction. So we'll simply reuse the existing datapath hardware to compute the address.

For the LD instruction the output of the ALU is routed to main memory as the address of the location we wish to access. After the memory's propagation delay, the contents of the addressed location is returned by the memory and we need to route that back to the register file to be written into the RC register.

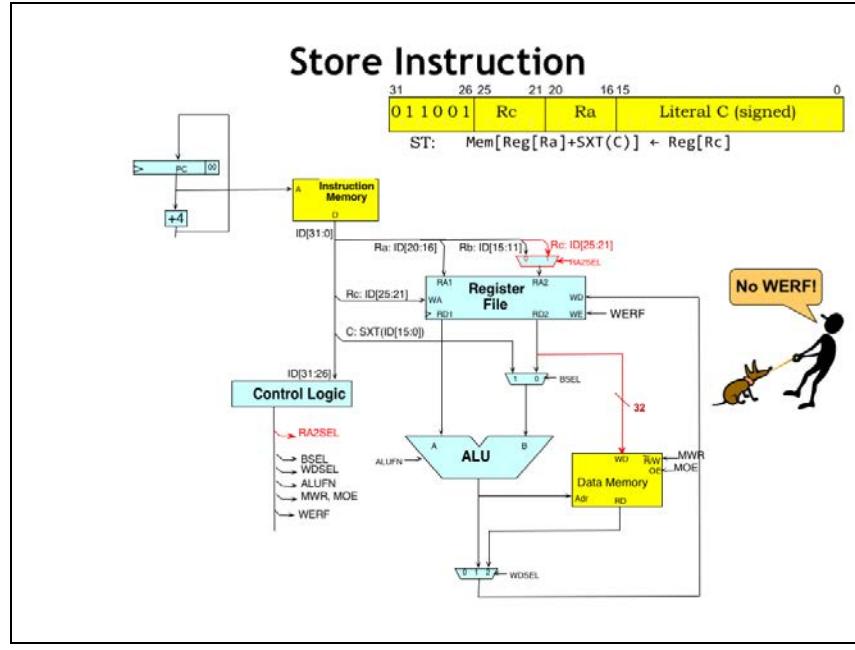
The memory has two control signals: MOE (memory output enable), which we set to 1 when we want to read a value from the memory. And MWE (memory write enable) which is set to 1 when we want main memory to store the value on its write data (WD) port into the addressed memory location.

We need to add another MUX to select which value to write back to the register file: the output of the ALU or the data returning from main memory. We've used a 3-to-1 MUX and we'll see the use for the other MUX input when we get to the implementation of branches and jumps. The two-bit WDSEL signal is used to select the source of the write-back value.



Let's follow the flow of data when executing the LD instruction. The ALU operands are chosen just as they are for the ADDC instruction and the ALU is requested to perform an ADD operation.

The ALU result is connected to the address port of main memory, who's control signals are set for a read operation. The WDSEL control signals are set to 2 to route the returning data to the register file.

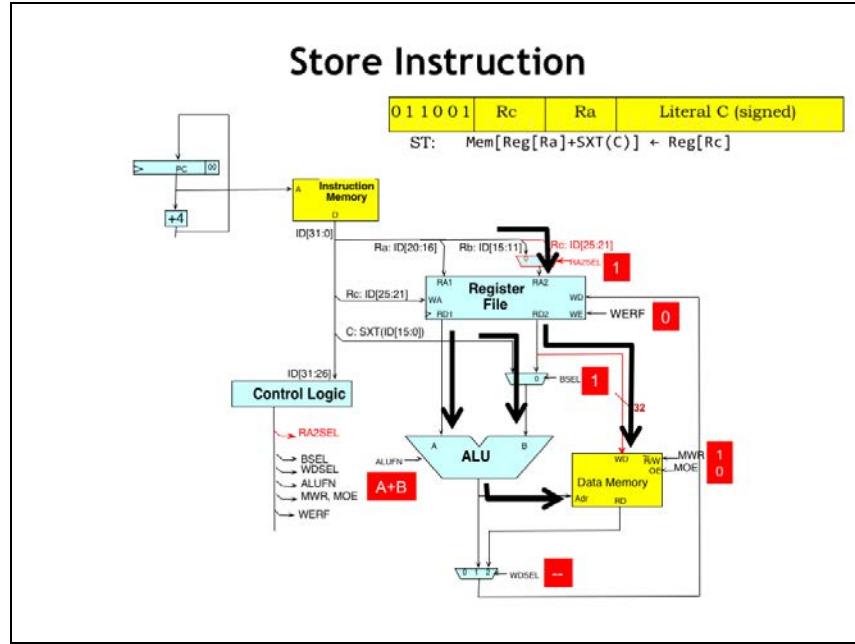


Execution of the ST instruction is very similar to the execution of the LD instruction, with one extra complication. The value to be written to memory comes from the RC register, but the RC instruction field is not connected a register file read address.

Happily, the RB register address isn't being used by the ST instruction since the second ALU operand comes from the literal field. So we'll use a MUX to enable the RC field to be selected as the address for the register file's second read port. When the RA2SEL control signal is 0, the RB field is selected as the address. When RA2SEL is 1, the RC field is selected.

The output from the second read data port is connected to the write data port of main memory.

The ST instruction is the only instruction that does not write a result into the register file. So the WERF control signal will be 0 when executing ST.

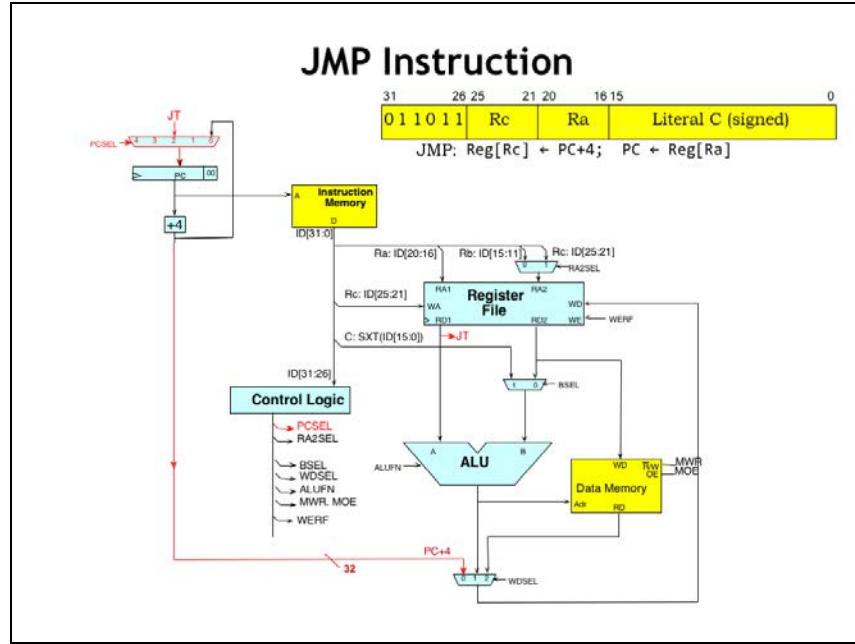


Here's the flow of data when executing ST. The operands are selected as for LD and the ALU performs the address computation with the result sent to main memory as the address.

Meanwhile the RC field is selected as the address for the second register file read port and the value from the RC register becomes the write data for main memory. By setting the MWR control signal to 1, the main memory will write the WD data into the selected memory location at the end of the cycle.

The WERF control signal is set to zero since we won't be writing a value into the register file. And, since we're not writing to the register file, we don't care about the value for the WDSEL signal.

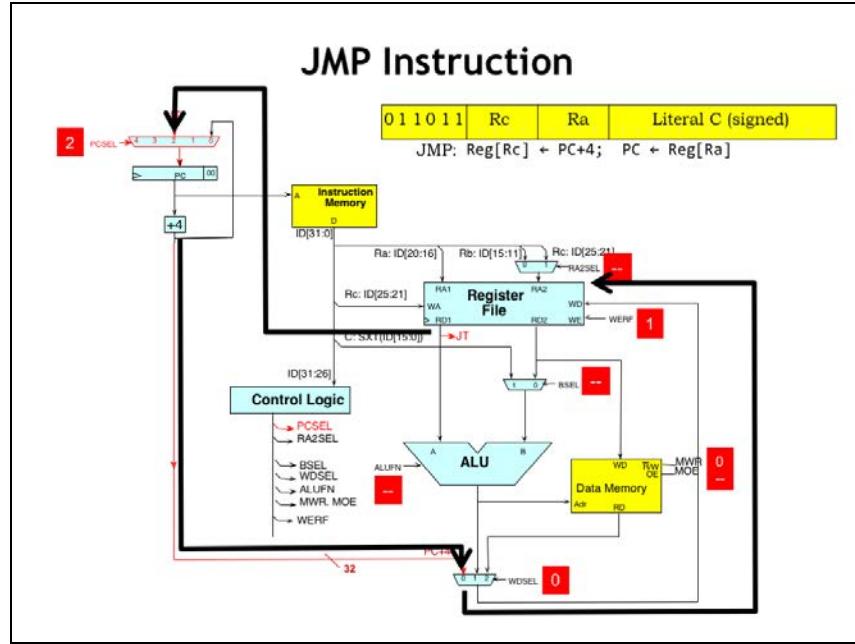
Of course, the logic will need to supply some value for WDSEL. The "don't care" annotation is telling the logic designer that she's welcome to supply whatever value is most convenient. This is particularly useful when using Karnaugh maps to optimize the control logic, where the value can be chosen as either 0 or 1, whichever results in the best minimization of the logic equations.



We're on the home stretch now. For all the instructions up until now, the next instruction has come from the location following the current instruction — hence the “PC+4” logic. Branches and JMPs change that by altering the value in the PC.

The JMP instruction simply takes the value in the RA register and makes it the next PC value. The PCSEL MUX in the upper left-hand corner lets the control logic select the source of the next PC value. When PCSEL is 0, the incremented PC value is chosen. When PCSEL is 2, the value of the RA register is chosen. We'll see how the other inputs to the PCSEL MUX are used in just a moment.

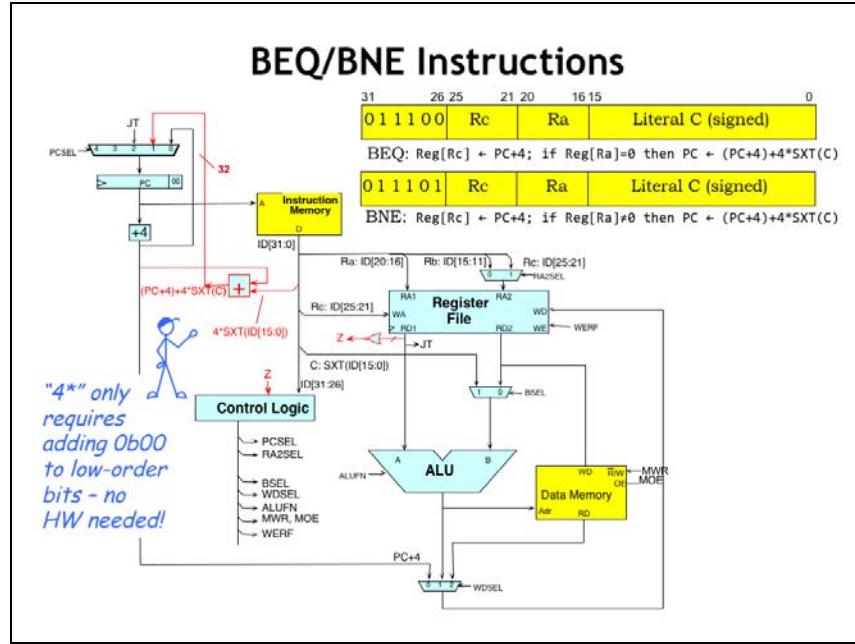
The JMP and branch instructions also cause the address of the following instruction, *i.e.*, the PC+4 value, to be written to the RC register. When WDSEL is 0, the “0” input of the WDSEL MUX is used to select the PC+4 value as the write-back data.



Here's how the data flow works. The output of the PC+4 adder is is routed to the register file and WERF is set to 1 to enable that value to be written at the end of the cycle.

Meanwhile, the value of RA register coming out of the register file is connected to the "2" input of the PCSEL MUX. So setting PCSEL to 2 will select the value in the RA register as the next value for the PC.

The rest of the control signals are "don't cares", except, of course for the memory write enable (MWR), which can never be "don't care" lest we cause an accidental write to some memory location.

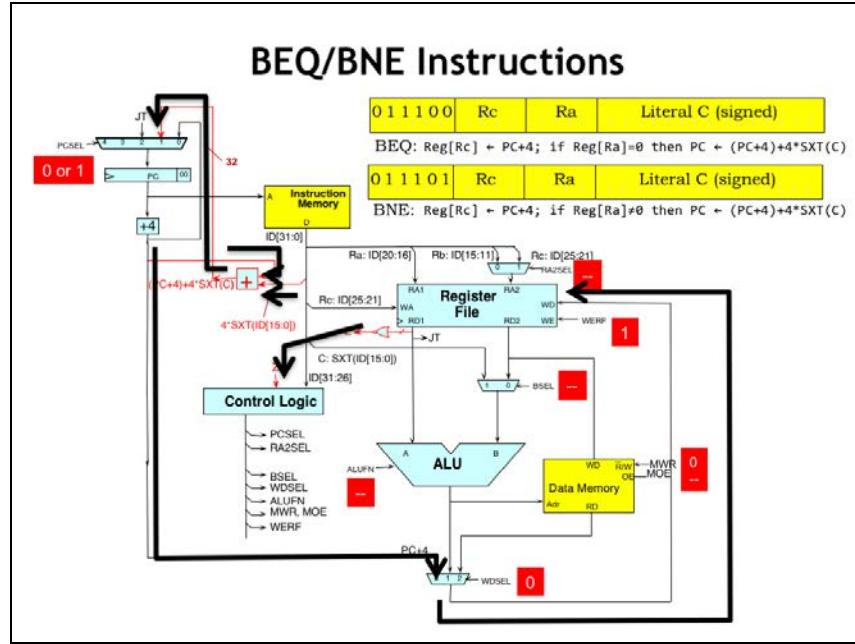


The branch instruction requires an additional adder to compute the target address by adding the scaled offset from the instruction's literal field to the current PC+4 value. Remember that we scale the offset by a factor of 4 to convert it from the word offset stored in the literal to the byte offset required for the PC. The output of the offset adder becomes the “1” input to the PCSEL MUX, where, if the branch is taken, it will become the next value of the PC.

Note that multiplying by 4 is easily accomplished by shifting the literal two bits to the left, which inserts two 0-bits at the low-order end of the value. And, like before, the sign-extension just requires replicating bit ID[15], in this case fourteen times. So implementing this complicated-looking expression requires care in wiring up the input to the offset adder, but no additional logic!

We do need some logic to determine if we should branch or not. The 32-bit NOR gate connected to the first read port of the register file tests the value of the RA register. The NOR’s output Z will be 1 if all the bits of the RA register value are 0, and 0 otherwise.

The Z value can be used by the control logic to determine the correct value for PCSEL. If Z indicates the branch is taken, PCSEL will be 1 and the output of the offset adder becomes the next value of the PC. If the branch is not taken, PCSEL will be 0 and execution will continue with the next instruction at PC+4.



As in the JMP instruction, the PC+4 value is routed to the register file to be written into the RC register at end of the cycle.

Meanwhile, the value of Z is computed from the value of the RA register while the branch offset adder computes the address of the branch target.

The output of the offset adder is routed to the PCSEL MUX where the value of the 3-bit PCSEL control signal, computed by the control logic based on Z, determines whether the next PC value is the branch target or the PC+4 value.

The remaining control signals are unused and set to their default “don’t care” values.

Load Relative Instruction

0 1 1 1 1 1	Rc	Ra	Literal C (signed)
LDR: Reg[Rc] \leftarrow Mem[PC + 4 + 4*SXT(C)]			

What's Load Relative good for anyway??? I thought

- Code is “PURE”, i.e. READ-ONLY; and stored in a “PROGRAM” region of memory;
- Data is READ-WRITE, and stored either
 - On the STACK (local); or
 - In some GLOBAL VARIABLE region; or
 - In a global storage HEAP.

So why have an instruction designed to load data that's “near” the instruction???

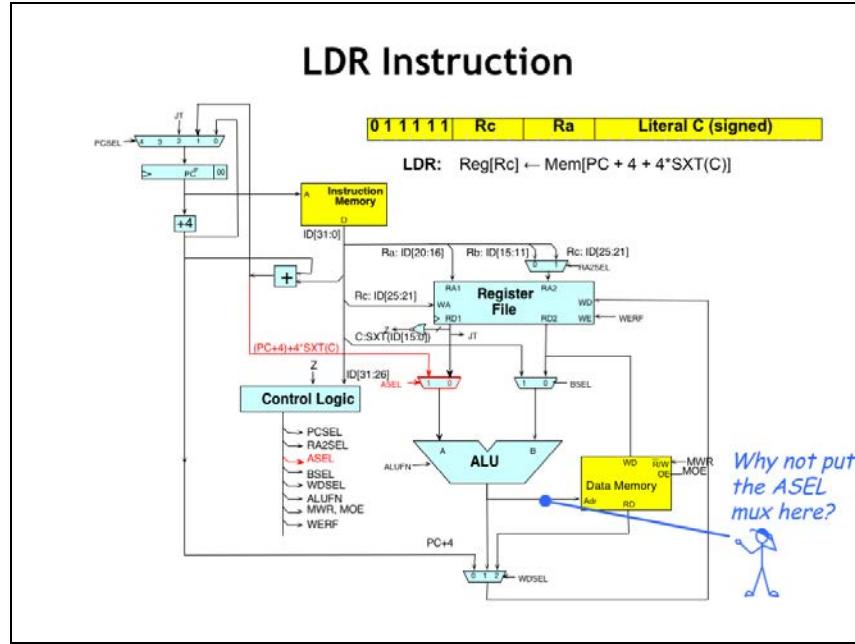
Addresses & other large constants

C: X = X * 123456;
BETA:
LD(X, r0)
LDR(c1, r1)
MUL(r0, r1, r0)
ST(r0, X)
...
c1: LONG(123456)

We have one last instruction to introduce: the LDR or load-relative instruction. LDR behaves like a normal LD instruction except that the memory address is taken from the branch offset adder.

Why would it be useful to load a value from a location near the LDR instruction? Normally such addresses would refer to the neighboring instructions, so why would we want to load the binary encoding of an instruction into a register to be used as data?

The use case for LDR is accessing large constants that have to be stored in main memory because they are too large to fit into the 16-bit literal field of an instruction. In the example shown here, the compiled code needs to load the constant 123456. So it uses an LDR instruction that refers to a nearby location C1: that has been initialized with the required value. Since this read-only constant is part of the program, it makes sense to store it with the instructions for the program, usually just after the code for a procedure. Note that we have to be careful to place the storage location so that it won't be executed as an instruction!



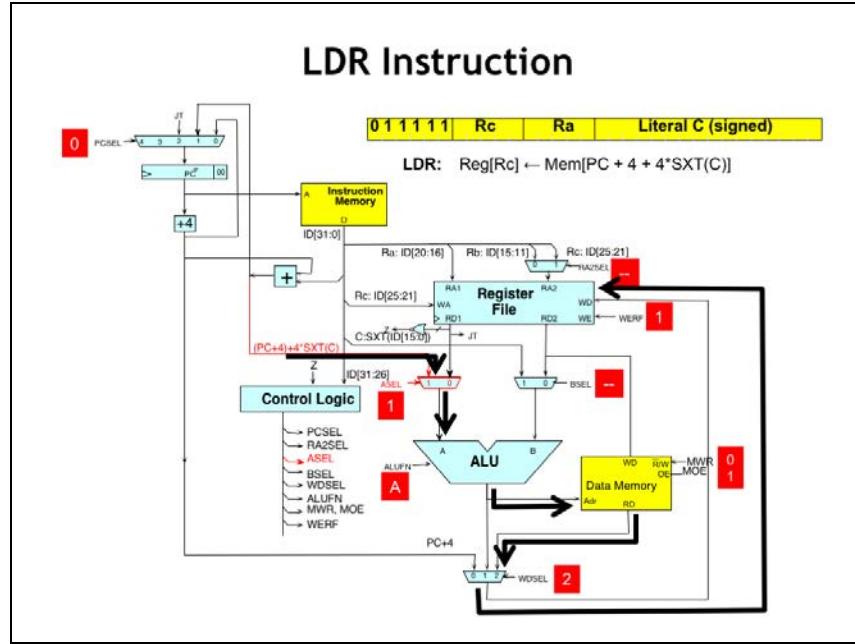
To route the output of the offset adder to the main memory address port, we'll add ASEL MUX so we can select either the RA register value (when ASEL=0) or the output of the offset adder (when ASEL=1) as the first ALU operand.

For LDR, ASEL will be 1, and we'll then ask the ALU compute the Boolean operation “A”, *i.e.*, the boolean function whose output is just the value of the first operand. This value then appears on the ALU output, which is connected to the main memory address port and the remainder of the execution proceeds just like it did for LD.

This seems a bit complicated! Mr. Blue has a good question: why not just put the ASEL MUX on the wire leading to the main memory address port and bypass the ALU altogether?

The answer has to do with the amount of time needed to compute the memory address. If we moved the ASEL MUX here, the data flow for LD and ST addresses would then pass through two MUXes: the BSEL MUX and the ASEL MUX, slowing down the arrival of the address by a small amount. This may not seem like a big deal, but the additional time would have to be added the clock period, thus slowing down every instruction by a little bit. When executing billions of instructions, a little extra time on each instruction really impacts the overall performance of the processor.

By placing the ASEL MUX where we did, its propagation delay overlaps that of the BSEL MUX, so the increased functionality it provides comes with no cost in performance.



Here's the data flow for the LDR instruction. The output of the offset adder is routed through the ASEL MUX to the ALU. The ALU performs the Boolean computation "A" and the result becomes the address for main memory.

The returning data is routed through the WDSEL MUX so it can be written into the RC register at the end of the cycle.

The remaining control signals are given their usual default values.

Exceptions

- What if something bad happens?
 - Execution of an illegal opcode
 - Reference to non-existent memory
 - Divide by zero
- Or maybe just something unanticipated
 - User hits a key
 - A packet comes in via the network
- Exceptions let us handle these cases in software:
 - Treat each case as an (implicit) procedure call
 - Procedure handles problem, returns to interrupted program
 - Transparent to interrupted program!
 - Important added capability: handlers for certain errors (illegal opcodes), can extend ISA using software

One last bit of housekeeping, then we're done! What should our hardware do if for some reason an instruction can't be executed? For example, if a programming error has led to trying to execute some piece of data as an instruction and the opcode field doesn't correspond to a Beta instruction (a so-called "illop" or illegal operation). Or maybe the memory address is larger than the actual amount main memory. Or maybe one of the operand values is not acceptable, *e.g.*, if the B operand for a DIV instruction is 0.

In modern computers, the accepted strategy is cease execution of the running program and transfer control to some error handler code. The error handler might store the program state onto disk for later debugging. Or, for an unimplemented but legal opcode, it might emulate the missing instruction in software and resume execution as if the instruction had been implemented in hardware!

There's also the need to deal with external events, like those associated with input and output. Here we'd like to interrupt the execution of the current program, run some code to deal with the external event, then resume execution as if the interrupt had never happened.

To deal with these cases, we'll add hardware to treat exceptions like forced procedure calls to special code to handle the situation, arranging to save the PC+4 value of the interrupted program so that the handler can resume execution if it wishes.

This is a very powerful feature since it allows us to transfer control to software to handle most any circumstance beyond the capability of our modest hardware. As we'll see in Part 3 of the course, the exception hardware will be our key to interfacing running programs to the operating system (OS) and to allow the OS to deal with external events without any awareness on the part of the running program.

Exception Processing

- Plan:
 - Interrupt running program
 - Invoke exception handler (like a procedure call)
 - Return to continue execution
- Exception and interrupt terms often used interchangeably, with minor distinctions:
 - Exceptions usually refer to **synchronous events**, generated by program (e.g., illegal instruction, divide-by-0, illegal address)
 - Interrupts usually refer to **asynchronous events**, generated by I/O devices (e.g., keystroke, packet received, disk transfer complete)

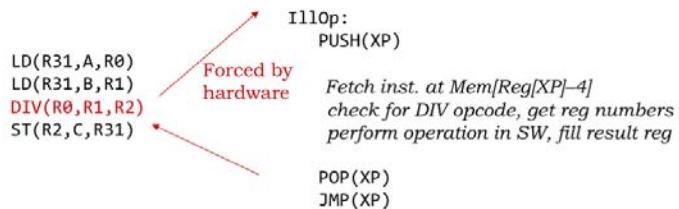
So our plan is to interrupt the running program, acting like the current instruction was actually a procedure call to the handler code. When it finishes execution, the handler can, if appropriate, use the normal procedure return sequence to resume execution of the user program.

We'll use the term "exception" to refer to exceptions caused by executing the current program. Such exceptions are "synchronous" in the sense that they are triggered by executing a particular instruction. In other words, if the program was re-run with the same data, the same exception would occur.

We'll use the term "interrupt" to refer to asynchronous exceptions resulting from external events whose timing is unrelated to the currently running program.

Exception Implementation

- Instead of executing instruction, fake a procedure call
 - Save current PC+4 (as branches do)
 - Load PC with exception vector: 0x4 for synchronous events, 0x8 for asynchronous events
- We save PC+4 in register R30 (which we call XP)
 - ... and prohibit programs from using XP (why?)
- Example: DIV unimplemented



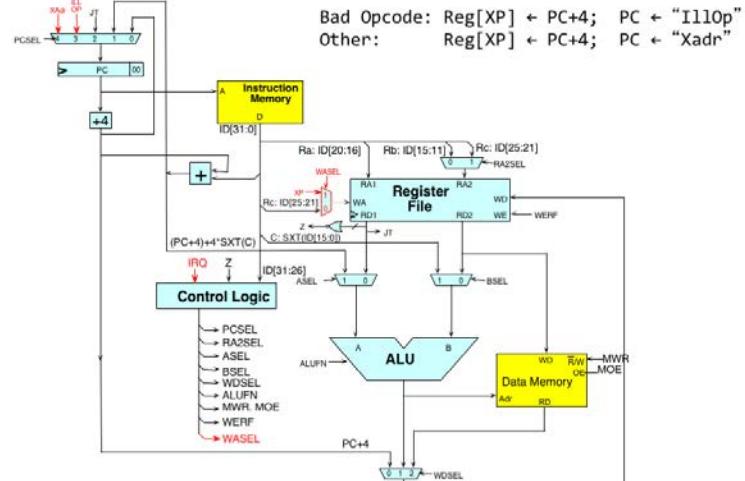
The implementation for both types of exceptions is the same. When an exception is detected, the Beta hardware will behave as if the current instruction was a taken BR to either location 0x4 (for synchronous exceptions) or location 0x8 (for asynchronous interrupts). Presumably the instructions in those locations will jump to the entry points of the appropriate handler routines.

We'll save the PC+4 value of the interrupted program into R30, a register dedicated to that purpose. We'll call that register XP ("exception pointer") to remind ourselves of how we're using it. Since interrupts in particular can happen at any point during a program's execution, thus overwriting the contents of XP at any time, user programs can't use the XP register to hold values since those values might disappear at any moment!

Here's how this scheme works: suppose we don't include hardware to implement the DIV instruction, so it's treated as an illegal opcode. The exception hardware forces a procedure call to location 0x4, which then branches to the Illop handler shown here. The PC+4 value of the DIV instruction has been saved in the XP register, so the handler can fetch the illegal instruction and, if it can, emulate its operation in software. When handler is complete, it can resume execution of the original program at the instruction following DIV by performing a JMP(XP).

Pretty neat!

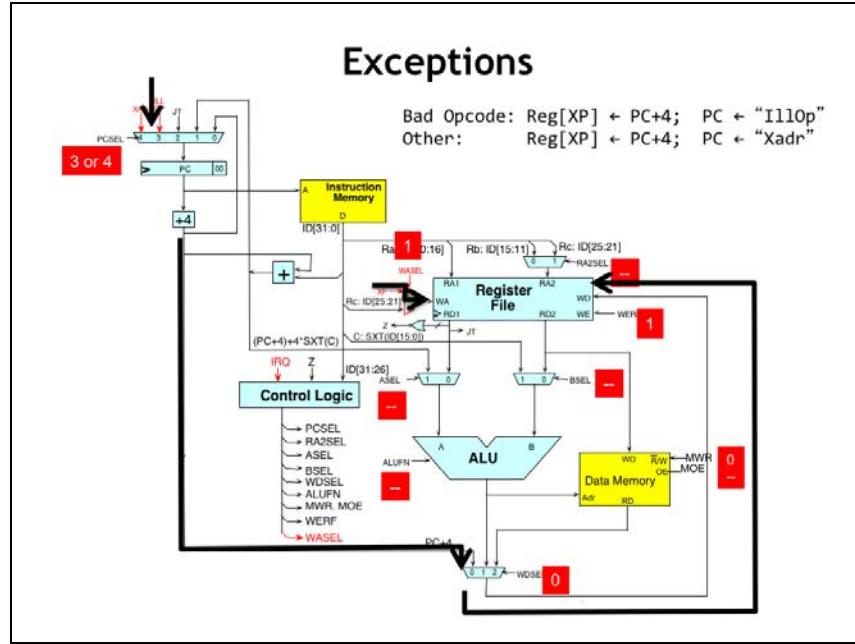
Exceptions



To handle exceptions, we only need a few simple changes to the datapath.

We've added a MUX controlled by the WASEL signal to choose the write-back address for the register file. When WASEL is 1, write-back will occur to the XP register, *i.e.*, register 30. When WASEL is 0, write-back will occur normally, *i.e.*, to the register specified by the RC field of the current instruction.

The remaining two inputs of the PCSEL MUX are set to the constant addresses for the exception handlers. In our case, 0x4 for illegal operations, and 0x8 for interrupts.

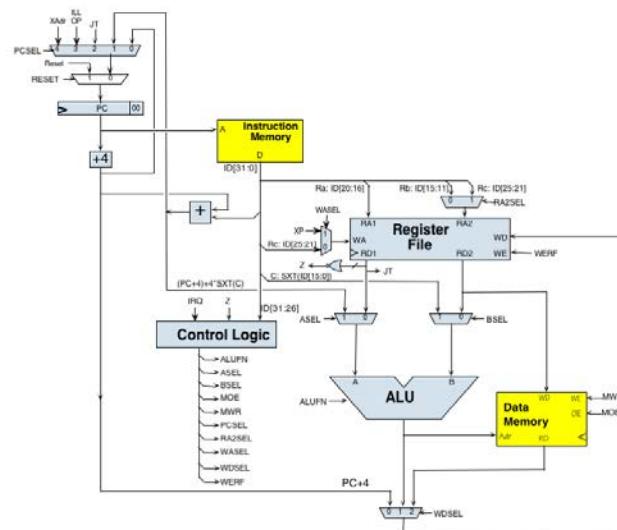


Here's the flow of control during an exception. The PC+4 value for the interrupted instruction is routed through the WDSEL MUX to be written into the XP register. Meanwhile the control logic chooses either 3 or 4 as the value of PCSEL to select the appropriate next instruction that will initiate the handling the exception.

The remaining control signals are forced to their “don’t care” values, since we no longer care about completing execution of the instruction we had fetched from main memory at the beginning of the cycle.

Note that the interrupted instruction has not been executed. So if the exception handler wishes to execute the interrupted instruction, it will have to subtract 4 from the value in the XP register before performing a JMP(XP) to resume execution of the interrupted program.

Beta: Our “Final Answer”



Okay, we’re done! Here’s the final datapath for executing instructions and handling exceptions. Please take a moment to remind yourself of what each datapath component does, *i.e.*, why it was added to the datapath. Similarly, you should understand how the control signals affect the operation of the datapath.

At least to my eye, this seems like a very modest amount of hardware to achieve all this functionality! It’s so modest in fact, that will ask you to actually complete the logic design for the Beta in an upcoming lab assignment :)

How does our design compare to the processor you’re using to view this course online? Modern processors have many additional complexities to increase performance: pipelined execution, the ability to execute more than instruction per cycle, fancier memory systems to reduce average memory access time, etc. We’ll cover some of these enhancements in upcoming lectures. The bottom line: the Beta hardware might occupy 1 or 2 sq mm on a modern integrated circuit, while a modern Intel processor occupies 300 to 600 sq mm. Clearly all that extra circuitry is there for a reason! If you’re curious, I’d recommend taking a course on advanced processor architecture.

Control Logic

	RESET	IRQ	OP	OPC	LD	LDR	ST	JMP	BEQ	BNE	ILLOP
ALUFN	--	--	F(op)	F(op)	"+"	"A"	"+"	--	--	--	--
ASEL	--	--	0	0	0	1	0	--	--	--	--
BSEL	--	--	0	1	1	--	1	--	--	--	--
MOE	--	--	--	--	1	1	0	--	--	--	--
MWR	0	0	0	0	0	0	1	0	0	0	0
PCSEL	--	4	0	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	3
RA2SEL	--	--	0	--	--	--	1	--	--	--	--
WASEL	--	1	0	0	0	0	--	0	0	0	1
WDSEL	--	0	1	1	2	2	--	0	0	0	0
WERF	--	1	1	1	1	1	0	1	1	1	1

Implementation choices:

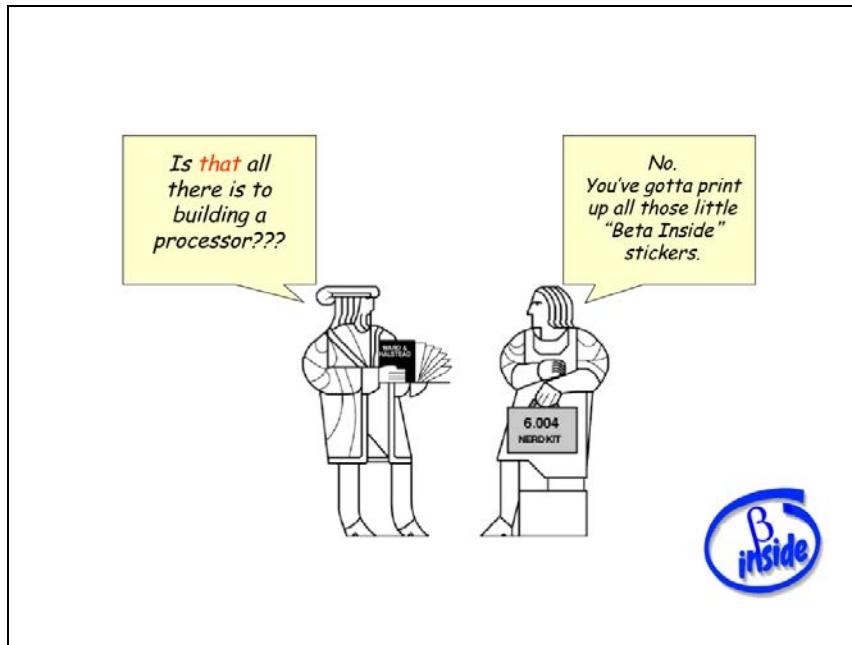
- 64-location ROM indexed by opcode with external logic to handle changes due to Z and IRQ inputs
- Entirely combinational logic (faster, but much more work!)

Here we've gathered up all the control signal settings for each class of instructions, including the settings needed for exceptions and during reset. Wherever possible, we've specified "don't care" for control signals whose value does not affect the actions of the datapath needed for a particular instruction.

Note that the memory write enable signal always has a defined value, ensuring that we only write to the memory during ST instructions. Similarly, the write enable for the register file is well-defined, except during RESET when presumably we're restarting the processor and don't care about preserving any register values.

As mentioned previously, a read-only memory (ROM) indexed by the 6-bit opcode field is the easiest way to generate the appropriate control signals for the current instruction. The Z and IRQ inputs to the control logic will affect the control signals and this can be accomplished with a small amount of logic to process the ROM outputs.

One can always have fun with Karnugh maps to generate a minimal implementation using ordinary logic gates. The result will be much smaller, both in terms of size and propagation delay, but requires a lot more design work! My recommendation: start with the ROM implementation and get everything else working. Then come back later when you feel like hacking logic gates :)



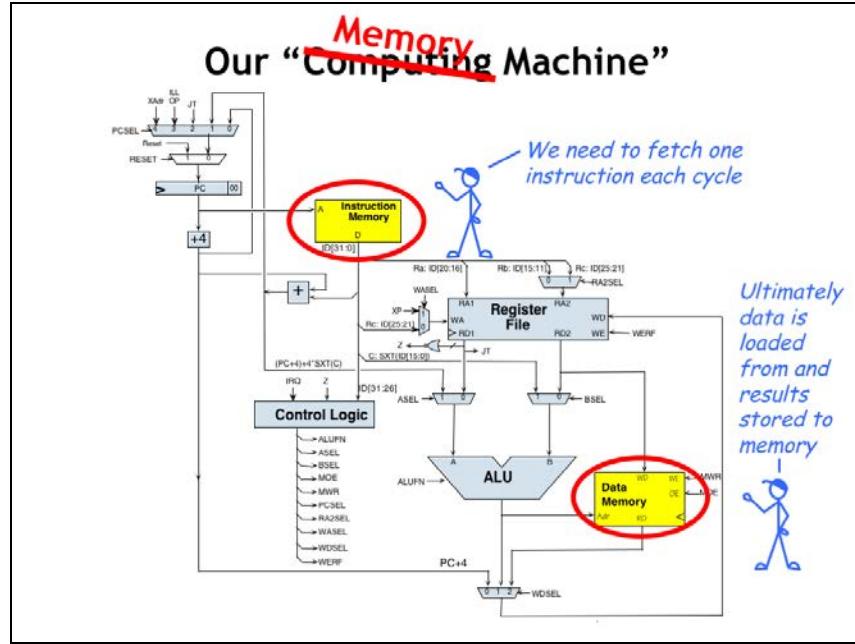
So that's what it takes to design the hardware for a simple 32-bit computer. Of course, we made the job easy for ourselves by choosing a simple binary encoding for our instructions and limiting the hardware functionality to efficiently executing the most common operations. Less common and more complex functionality can be left to software. The exception mechanism gave us a powerful tool for transferring control to software when the hardware couldn't handle the task.

Have fun completing the hardware design of your Beta. Thousands of MIT students have enjoyed that "Yes!" moment when their design works for the first time. For their efforts we reward them with the "Beta Inside" sticker you see here, which you can see on laptops as you walk around the Institute.

Good luck!

14: The Memory Hierarchy

1. Our Memory Machine
2. Memory Technologies
3. Static RAM (SRAM)
4. SRAM Cell
5. SRAM Read
6. SRAM Write
7. Multiported SRAMs
8. Summary: SRAM
9. 1T Dynamic RAM (DRAM) Cell
10. 1T DRAM Writes and Reads
11. Summary: DRAM
12. Non-Volatile Storage: Flash
13. Non-Volatile Storage: Hard Disk
14. Summary: Memory Technologies
15. Memory Hierarchy Interface
16. Memory Hierarchy Interface
17. The Locality Principle
18. Memory Reference Patterns
19. Caches
20. A Typical Memory Hierarchy
21. Cache Access
22. Cache Metrics
23. Example: How High of a Hit Ratio?
24. Basic Cache Algorithm
25. Direct-Mapped Caches
26. Example: Direct-Mapped Caches
27. Block Size
28. Block Size Tradeoffs
29. Direct-Mapped Cache Problem: Conflict Misses
30. Fully-Associative Cache
31. N-way Set-Associative Cache I
32. N-way Set-Associative Cache II
33. “Let me count the ways.”
34. Associativity Tradeoffs
35. Associativity Implies Choices
36. Replacement Policies
37. Write Policy
38. Write-back
39. Write-back with “Dirty” Bits
40. Summary: Cache Tradeoffs



In the last lecture we completed the design of the Beta, our reduced-instruction-set computer. The simple organization of the Beta ISA meant that there was a lot commonality in the circuitry needed to implement the instructions. The final design has a few main building blocks with MUX steering logic to select input values as appropriate.

If we were to count MOSFETs and think about propagation delays, we'd quickly determine that our 3-port main memory (shown here as the two yellow components) was the most costly component both in terms of space and percentage of the cycle time required by the memory accesses. So in many ways, we really have a "memory machine" instead of a "computing machine".

The execution of every instruction starts by fetching the instruction from main memory. And ultimately all the data processed by the CPU is loaded from or stored to main memory. A very few frequently-used variable values can be kept in the CPU's register file, but most interesting programs manipulate **much** more data than can be accommodated by the storage available as part of the CPU datapath.

In fact, the performance of most modern computers is limited by the bandwidth, *i.e.*, bytes/second, of the connection between the CPU and main memory, the so-called *memory bottleneck*. The goal of this lecture is to understand the nature of the bottleneck and to see if there architectural improvements we might make to minimize the problem as much as possible.

Memory Technologies

Technologies have vastly different tradeoffs between capacity, access latency, bandwidth, energy, and cost

- ... and logically, different applications

	Capacity	Latency	Cost/GB
Register	1000s of bits	20 ps	\$\$\$\$
SRAM	~10 KB-10 MB	1-10 ns	~\$1000
DRAM	~10 GB	80 ns	~\$10
Flash*	~100 GB	100 us	~\$1
Hard disk*	~1 TB	10 ms	~\$0.10

* non-volatile (retains contents when powered off)

Processor
Datapath
Memory
Hierarchy
I/O
subsystem

We have a number of memory technologies at our disposal, varying widely in their capacity, latency, bandwidth, energy efficiency and their cost. Not surprisingly, we find that each is useful for different applications in our overall system architecture.

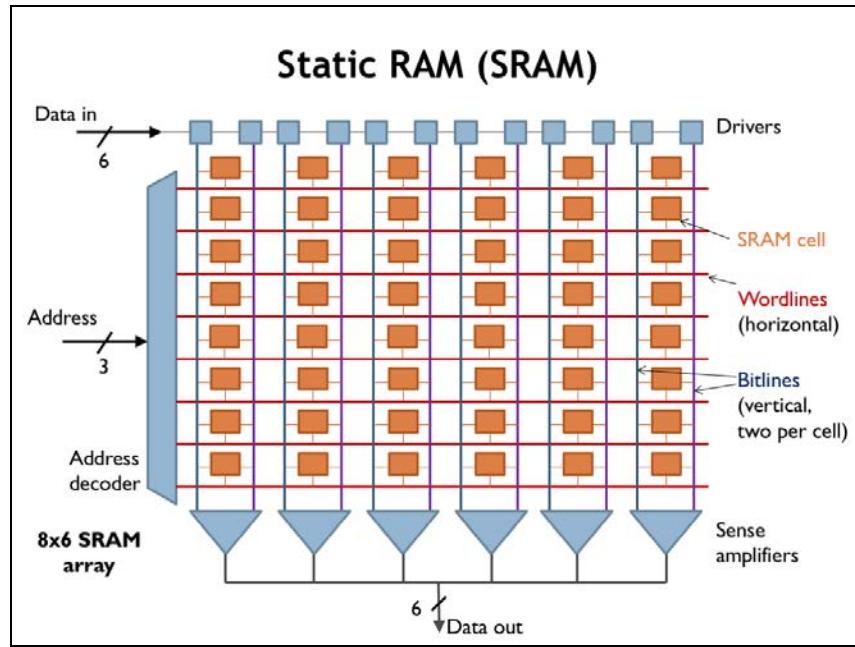
Our registers are built from sequential logic and provide very low latency access (20ps or so) to at most a few thousands of bits of data. Static and dynamic memories, which we'll discuss further in the coming slides, offer larger capacities at the cost of longer access latencies. Static random-access memories (SRAMs) are designed to provide low latencies (a few nanoseconds at most) to many thousands of locations. Already we see that more locations means longer access latencies — this is a fundamental size vs. performance tradeoff of our current memory architectures. The tradeoff comes about because increasing the number of bits will increase the area needed for the memory circuitry, which will in turn lead to longer signal lines and slower circuit performance due to increased capacitive loads.

Dynamic random-access memories (DRAMs) are optimized for capacity and low cost, sacrificing access latency. As we'll see in this lecture, we'll use both SRAMs and DRAMs to build a hybrid memory hierarchy that provides low average latency and high capacity — an attempt to get the best of both worlds!

Notice that the word “average” has snuck into the performance claims. This means that we'll be relying on statistical properties of memory accesses to achieve our goals of low latency and high capacity. In the worst case, we'll still be stuck with the capacity limitations of SRAMs and the long latencies of DRAMs, but we'll work hard to ensure that the worst case occurs infrequently!

Flash memory and hard-disk drives provide non-volatile storage. *Non-volatile* means that the memory contents are preserved even when the power is turned off. Hard disks are at the bottom of the memory hierarchy, providing massive amounts of long-term storage for very little cost. Flash memories, with a 100-fold improvement in access latency, are often used in concert with hard-disk drives in the same way that SRAMs are used in concert with DRAMs, *i.e.*, to provide a hybrid system for non-volatile storage that has improved latency **and** high capacity.

Let's learn a bit more about each of these four memory technologies, then we'll return to the job of building our memory system.



SRAMs are organized as an array of memory locations, where a memory access is either reading or writing all the bits in a single location. Here we see the component layout for a 8-location SRAM array where each location hold 6 bits of data. You can see that the individual bit cells are organized as 8 rows (one row per location) by 6 columns (one column per bit in each memory word). The circuitry around the periphery is used to decode addresses and support read and write operations.

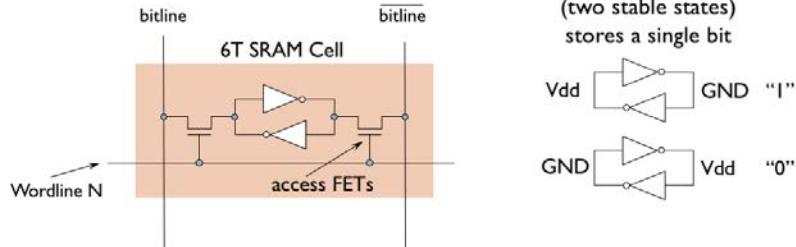
To access the SRAM, we need provide enough address bits to uniquely specify the location. In this case we need 3 address bits to select one of the 8 memory locations. The address decoder logic sets one of the 8 wordlines (the horizontal wires in the array) high to enable a particular row (location) for the upcoming access. The remaining wordlines are set low, disabling the cells they control. The active wordline enables each of the SRAM bit cells on the selected row, connecting each cell to a pair of bit lines (the vertical wires in the array). During read operations the bit lines carry the analog signals from the enabled bit cells to the sense amplifiers, which convert the analog signals to digital data. During write operations incoming data is driven onto the bit lines to be stored into the enabled bit cells.

Larger SRAMs will have a more complex organization in order to minimize the length, and hence the capacitance, of the bit lines.

SRAM Cell

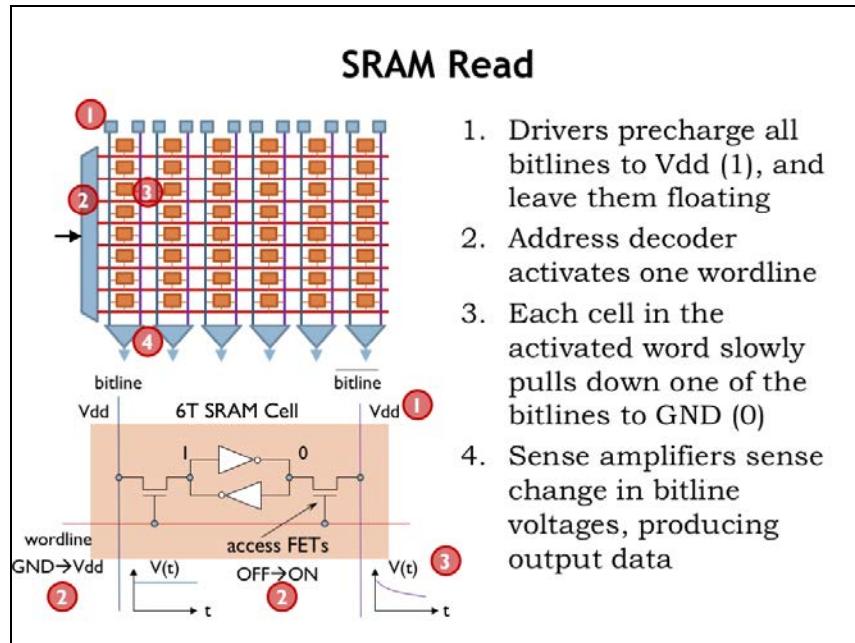
6-MOSFET (6T) cell:

- Two CMOS inverters (4 MOSFETs) forming a **bistable element**
- Two **access transistors**



The heart of the SRAM are the bit cells. The typical cell has two CMOS inverters wired in a positive feedback loop to create a bistable storage element. The diagram on the right shows the two stable configurations. In the top configuration, the cell is storing a 1 bit. In the bottom configuration, it's storing a 0 bit. The cell provides stable storage in the sense that as long as there's power, the noise immunity of the inverters will ensure that the logic values will be maintained even if there's electrical noise on either inverter input.

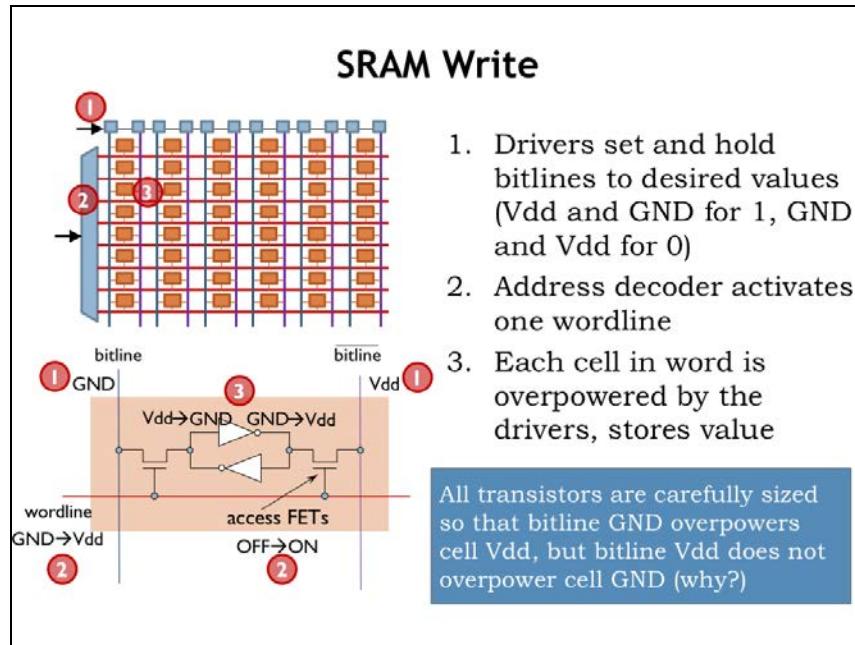
Both sides of the feedback loop are connected via access FETs to the two vertical bit lines. When the wordline connected to the gates of the access FETs is high, the FETs are on, *i.e.*, they will make an electrical connection between the cell's internal circuitry and the bitlines. When the wordline is low, the access FETs are off and the bistable feedback loop is isolated from the bitlines and will happily maintain the stored value as long as there's power.



During a read operation, the drivers first recharge all the bitlines to Vdd (*i.e.*, a logical 1 value) and then disconnect, leaving the bitlines floating at 1. Then the address decoder sets one of the wordlines high, connecting a row of bit cells to their bitlines. Each cell in the selected row then pulls one of its two bitlines to GND. In this example, it's the right bitline that's pulled low. Transitions on the bitlines are slow since the bitline has a large total capacitance and the MOSFETs in the two inverters are small to keep the cell has small as possible. The large capacitance comes partly from the bitline's length and partly from the diffusion capacitance of the access FETs in other cells in the same column.

Rather than wait for the bitline to reach a valid logic level, sense amplifiers are used to quickly detect the small voltage difference developing between the two bitlines and generate the appropriate digital output. Since detecting small changes in a voltage is very sensitive to electrical noise, the SRAM uses a pair of bitlines for each bit and a differential sense amplifier to provide greater noise immunity.

As you can see, designing a low-latency SRAM involves a lot of expertise with the analog behavior of MOSFETs and some cleverness to ensure electrical noise will not interfere with the correct operation of the circuitry.



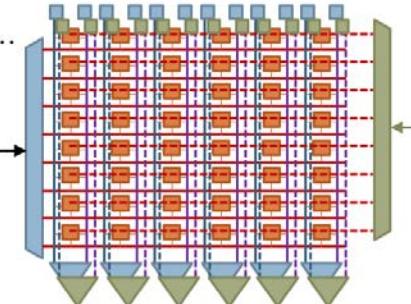
Write operations start by driving the bitlines to the appropriate values. In the example shown here, we want to write a 0-bit into the cell, so the left bitline is set to GND and the right bitline is set to VDD. As before, the address decoder then sets one of the wordlines high, selecting all the cells in a particular row for the write operation.

The drivers have much larger MOSFETs than those in the cell's inverters, so the internal signals in the enabled cells are forced to the values on the bitlines and the bistable circuits “flip” into the new stable configuration. We’re basically shorting together the outputs of the driver and the internal inverter, so this is another analog operation! This would be a no-no in a strictly digital circuit.

Since n-fets usually carry much higher source-drain currents than p-fets of the same width and given the threshold-drop of the n-fet access transistor, almost all the work of the write is performed by the large n-fet pulldown transistor connected to the bitline with the 0 value, which easily overpowers the small p-fet pullup of the inverters in the cell. Again, SRAM designers need a lot of expertise to correctly balance the sizes of MOSFETs to ensure fast and reliable write operations.

Multiported SRAMs

- SRAM so far can do either one read or one write/cycle
- We can do multiple reads and writes with multiple ports by adding one set of wordlines and bitlines per port
- Cost/bit? For N ports...
 - Wordlines: $\frac{N}{2^*N}$
 - Bitlines: $\frac{2^*N}{2^*N}$
 - Access FETs: $\frac{2^*N}{2^*N}$
- Wires often dominate area $\rightarrow O(N^2)$ area!



It's not hard to augment the SRAM to support multiple read/write ports, a handy addition for register file circuits. We'll do this by adding additional sets of wordlines, bitlines, drivers, and sense amps. This will give us multiple paths to independently access the bistable storage elements in the various rows of the memory array.

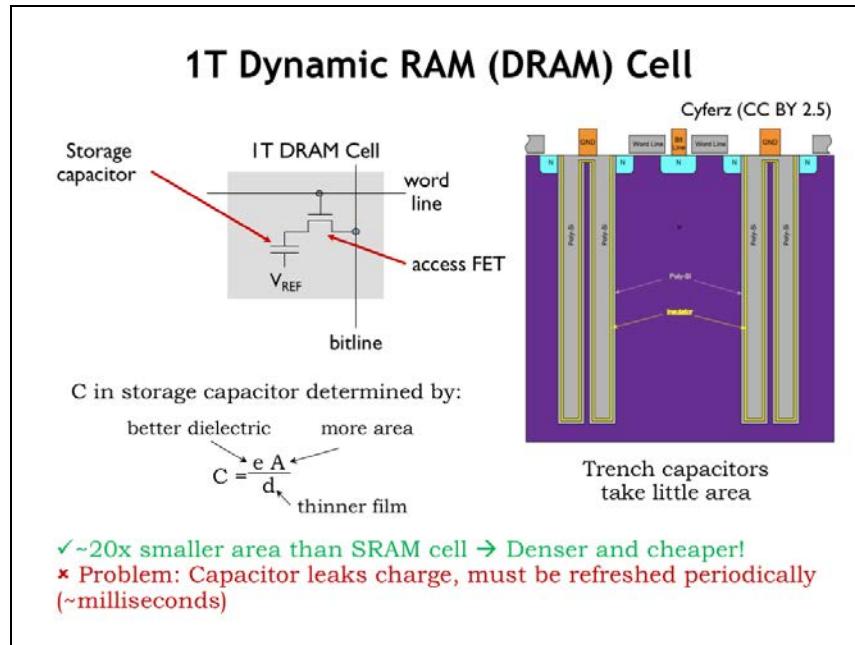
With an N-port SRAM, for each bit we'll need N wordlines, $2N$ bitlines and $2N$ access FETs. The additional wordlines increase the effective height of the cell and the additional bitlines increase the effective width of the cell and so the area required by all these wires quickly dominates the size of the SRAM. Since both the height and width of a cell increase when adding ports, the overall area grows as the square of the number of read/write ports. So one has to take care not to gratuitously add ports lest the cost of the SRAM get out of hand.

Summary: SRAMs

- Array of $k \times b$ cells (k words, b cells per word)
- Cell is a bistable element + access transistors
 - Analog circuit with carefully sized transistors to allow reads and writes
- Read: Precharge bitlines, activate wordline, sense
- Write: Drive bitlines, activate wordline, overpower cells
- 6 MOSFETs/cell... can we do better?
 - What's the minimum number of MOSFETs needed to store a single bit?

In summary, the circuitry for the SRAM is organized as an array of bit cells, with one row for each memory location and one column for each bit in a location. Each bit is stored by two inverters connected to form a bistable storage element. Reads and writes are essentially analog operations performed via the bitlines and access FETs.

The SRAM uses 6 MOSFETs for each bit cell. Can we do better? What's the minimum number of MOSFETs needed to store a single bit of information?



Well, we'll need at least one MOSFET to serve as the access FET so we can select which bits will be affected by read and write operations. We can use a simple capacitor for storage, where the value of a stored bit is represented by voltage across the plates of the capacitor. The resulting circuit is termed a dynamic random-access memory (DRAM) cell.

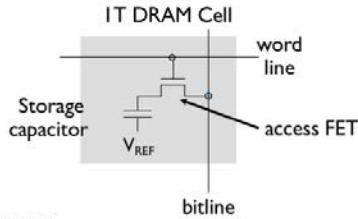
If the capacitor voltage exceeds a certain threshold, we're storing a 1 bit, otherwise we're storing a 0. The amount of charge on the capacitor, which determines the speed and reliability of reading the stored value, is proportional to the capacitance. We can increase the capacitance by increasing the dielectric constant of the insulating layer between the two plates of the capacitor, increasing the area of the plates, or by decreasing the the distance between the plates. All of these are constantly being improved.

A cross section of a modern DRAM cell is shown here. The capacitor is formed in a large trench dug into the substrate material of the integrated circuit. Increasing the depth of the trench will increase the area of the capacitor plates without increasing the cell's area. The wordline forms the gate of the N-FET access transistor connecting the outer plate of the capacitor to the bitline. A very thin insulating layer separates the outer plate from the inner plate, which is connected to some reference voltage (shown as GND in this diagram). You can Google *trench capacitor* to get the latest information on the dimensions and materials used in the construction of the capacitor.

The resulting circuit is quite compact: about 20-times less area/bit than an SRAM bit cell. There are some challenges however. There's no circuitry to main the static charge on the capacitor, so stored charge will leak from the outer plate of the capacitor, hence the name *dynamic memory*. The leakage is caused by small picoamp currents through the PN junction with the surrounding substrate, or subthreshold conduction of the access FET even when it's turned off. This limits the amount of time we can leave the capacitor unattended and still expect to read the stored value. This means we'll have to arrange to read then re-write each bit cell (called a *refresh cycle*) every 10ms or so, adding to the complexity of the DRAM interface circuitry.

DRAM Writes and Reads

- Writes: Drive bitline to Vdd or GND, activate wordline, charge or discharge capacitor
- Reads:
 1. Precharge bitline to $V_{DD}/2$
 2. Activate wordline
 3. Capacitor and bitline share charge
 - If capacitor was discharged, bitline voltage decreases slightly
 - If capacitor was charged, bitline voltage increases slightly
 4. Sense bitline to determine if 0 or 1
 - Issue: **Reads are destructive!** (charge is gone!)
 - So, data must be rewritten to cell at end of read



DRAM write operations are straightforward: simply turn on the access FET with the wordline and charge or discharge the storage capacitor through the bitline.

Reads are bit more complicated. First the bitline is precharged to some intermediate voltage, *e.g.*, $V_{DD}/2$, and then the precharge circuitry is disconnected. The wordline is activated, connecting the storage capacitor of the selected cell to the bitline causing the charge on the capacitor to be shared with the charge stored by the capacitance of the bitline. If the value stored by the cell capacitor is a 1, the bitline voltage will increase very slightly (*e.g.*, a few tens of millivolts). If the stored value is a 0, the bitline voltage will decrease slightly. Sense amplifiers are used to detect this small voltage change to produce a digital output value.

This means that read operations wipe out the information stored in the bit cell, which must then be rewritten with the detected value at the end of the read operation.

DRAM circuitry is usually organized to have wide rows, *i.e.*, multiple consecutive locations are read in a single access. This particular block of locations is selected by the DRAM row address. Then the DRAM column address is used to select a particular location from the block to be returned. If we want to read multiple locations in a single row, then we only need to send a new column address and the DRAM will respond with that location without having to access the bit cells again. The first access to a row has a long latency, but subsequent accesses to the same row have very low latency. As we'll see, we'll be able to use fast column accesses to our advantage.

Summary: DRAM

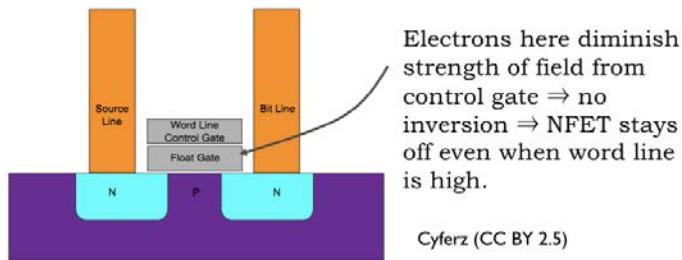
- 1T DRAM cell: transistor + capacitor
- Smaller than SRAM cell, but destructive reads and capacitors leak charge
- DRAM arrays include circuitry to:
 - Write word again after every read (to avoid losing data)
 - Refresh (read+write) every word periodically
- DRAM vs SRAM:
 - ~20x denser than SRAM
 - ~2-10x slower than SRAM

In summary, DRAM bit cells consist of a single access FET connected to a storage capacitor that's cleverly constructed to take up as little area as possible. DRAMs must rewrite the contents of bit cells after they are read and every cell must be read and written periodically to ensure that the stored charge is refreshed before it's corrupted by leakage currents.

DRAMs have much higher capacities than SRAMs because of the small size of the DRAM bit cells, but the complexity of the DRAM interface circuitry means that the initial access to a row of locations is quite a bit slower than an SRAM access. However subsequent accesses to the same row happen at speeds close to that of SRAM accesses.

Both SRAMs and DRAMs will store values as long as their circuitry has power. But if the circuitry is powered down, the stored bits will be lost. For long-term storage we will need to use non-volatile memory technologies, the topic of the next lecture segment.

Non-Volatile Storage: Flash



Cyferz (CC BY 2.5)

Flash Memory: Use “floating gate” transistors to store charge

- **Very dense:** Multiple bits/transistor, read and written in blocks
- **Slow** (especially on writes), 10-100 us
- **Limited number of writes:** charging/discharging the floating gate (writes) requires large voltages that damage transistor

Non-volatile memories are used to maintain system state even when the system is powered down. In flash memories, long-term storage is achieved by storing charge on a well-insulated conductor called a floating gate, where it will remain stable for years. The floating gate is incorporated in a standard MOSFET, placed between the MOSFET’s gate and the MOSFET’s channel. If there is no charge stored on the floating gate, the MOSFET can be turned on, *i.e.*, be made to conduct, by placing a voltage V_1 on the gate terminal, creating an inversion layer that connects the MOSFET’s source and drain terminals. If there is a charge stored on the floating gate, a higher voltage V_2 is required to turn on the MOSFET. By setting the gate terminal to a voltage between V_1 and V_2 , we can determine if the floating gate is charged by testing to see if the MOSFET is conducting.

In fact, if we can measure the current flowing through the MOSFET, we can determine how much charge is stored on the floating gate, making it possible to store multiple bits of information in one flash cell by varying the amount of charge on its floating gate. Flash cells can be connected in parallel or series to form circuits resembling CMOS NOR or NAND gates, allowing for a variety of access architectures suitable for either random or sequential access.

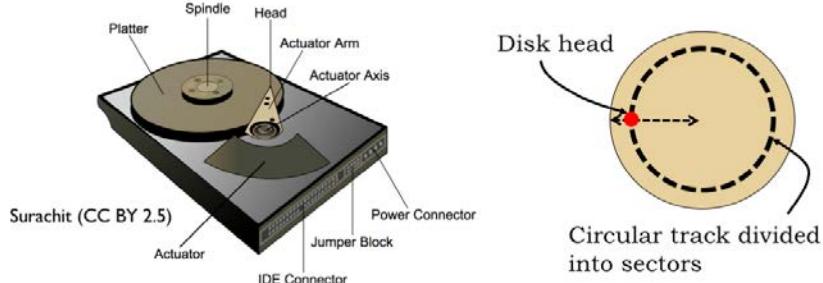
Flash memories are very dense, approaching the areal density of DRAMs, particularly when each cell holds multiple bits of information.

Read access times for NOR flash memories are similar to that of DRAMs, several tens of nanoseconds. Read times for NAND flash memories are much longer, on the order of 10 microseconds. Write times for all types of flash memories are quite long since high voltages have to be used to force electrons to cross the insulating barrier surrounding the floating gate.

Flash memories can only be written some number of times before the insulating layer is damaged to the point that the floating gate will no longer reliably store charge. Currently the number of guaranteed writes varies between 100,000 and 1,000,000. To work around this limitation, flash chips contain clever address mapping algorithms so that writes to the same address actually are mapped to different flash cells on each successive write.

The bottom line is that flash memories are a higher-performance but higher-cost replacement for the hard-disk drive, the long-time technology of choice for non-volatile storage.

Non-Volatile Storage: Hard Disk



Hard Disk: Rotating magnetic platters + read/write head

- **Extremely slow** (~10ms): Mechanically move head to position, wait for data to pass underneath head
- ~100MB/s for sequential read/writes
- ~100KB/s for random read/writes
- **Cheap**

A hard-disk drive (HDD) contains one or more rotating platters coated with a magnetic material. The platters rotate at speeds ranging from 5400 to 15000 RPM. A read/write head positioned above the surface of a platter can detect or change the orientation of the magnetization of the magnetic material below. The read/write head is mounted an actuator that allows it to be positioned over different circular tracks.

To read a particular sector of data, the head must be positioned radially over the correct track, then wait for the platter to rotate until it's over the desired sector. The average total time required to correctly position the head is on the order of 10 milliseconds, so hard disk access times are quite long.

However, once the read/write head is in the correct position, data can be transferred at the respectable rate of 100 megabytes/second. If the head has to be repositioned between each access, the effective transfer rate drops 1000-fold, limited by the time it takes to reposition the head.

Hard disk drives provide cost-effective non-volatile storage for terabytes of data, albeit at the cost of slow access times.

Summary: Memory Technologies

	Capacity	Latency	Cost/GB
Register	1000s of bits	20 ps	\$\$\$\$
SRAM	~10 KB-10 MB	1-10 ns	~\$1000
DRAM	~10 GB	80 ns	~\$10
Flash	~100 GB	100 us	~\$1
Hard disk	~1 TB	10 ms	~\$0.10

- Different technologies have vastly different tradeoffs
- Size is a **fundamental limit**, even setting cost aside:
 - Small + low latency, high bandwidth, low energy, **or**
 - Large + high-latency, low bandwidth, high energy
- Can we get the best of both worlds? (large, fast, cheap)

This completes our whirlwind tour of memory technologies. If you'd like to learn a bit more, Wikipedia has useful articles on each type of device. SRAM sizes and access times have kept pace with the improvements in the size and speed of integrated circuits. Interestingly, although capacities and transfer rates for DRAMs and HDDs have improved, their initial access times have not improved nearly as rapidly. Thankfully over the past decade flash memories have helped to fill the performance gap between processor speeds and HDDs. But the gap between processor cycle times and DRAM access times has continued to widen, increasing the challenge of designing low-latency high-capacity memory systems.

The capacity of the available memory technologies varies over 10 orders of magnitude, and the variation in latencies varies over 8 orders of magnitude. This creates a considerable challenge in figuring out how to navigate the speed vs size tradeoffs.

Each transition in memory hierarchy shows the same fundamental design choice: we can pick smaller-and-faster or larger-and-slower. This is a bit awkward actually — can we figure how to get the best of both worlds?

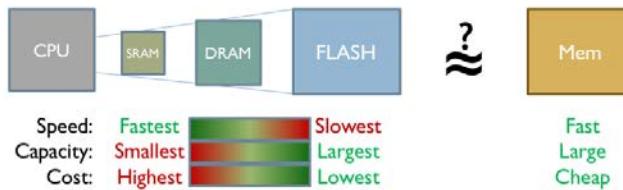
The Memory Hierarchy

Want large, fast, and cheap memory, but...

Large memories are slow (even if built with fast components)

Fast memories are expensive

Idea: Can we use a **hierarchical system** of memories with different tradeoffs to **emulate** a large, fast, cheap memory?



We want our system to behave as if it had a large, fast, and cheap main memory. Clearly we can't achieve this goal using any single memory technology.

Here's an idea: can we use a hierarchical system of memories with different tradeoffs to achieve close to the same results as a large, fast, cheap memory? Could we arrange for memory locations we're using often to be stored, say, in SRAM and have those accesses be low latency? Could the rest of the data be stored in the larger and slower memory components, moving the between the levels when necessary? Let's follow this train of thought and see where it leads us.

Memory Hierarchy Interface

Approach 1: Expose Hierarchy

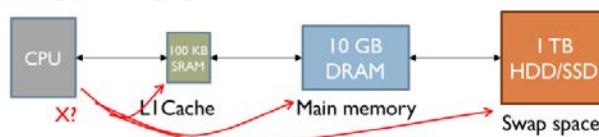
- Registers, SRAM, DRAM, Flash, Hard Disk each available as storage alternatives



- Tell programmers: "Use them cleverly"

Approach 2: Hide Hierarchy

- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns



There are two approaches we might take. The first is to expose the hierarchy, providing some amount of each type of storage and let the programmer decide how best to allocate the various memory resources for each particular computation. The programmer would write code that moved data into fast storage when appropriate, then back to the larger and slower memories when low-latency access was no longer required. There would only be a small amount of the fastest memory, so data would be constantly in motion as the focus of the computation changed.

This approach has had notable advocates. Perhaps the most influential was Seymour Cray, the “Steve Jobs” of supercomputers. Cray was the architect of the world’s fastest computers in each of three decades, inventing many of the technologies that form the foundation of high-performance computing. His insight to managing the memory hierarchy was to organize data as vectors and move vectors in and out of fast memory under program control. This was actually a good data abstraction for certain types of scientific computing and his vector machines had the top computing benchmarks for many years.

The second alternative is to hide the hierarchy and simply tell the programmer they have a large, uniform address space to use as they wish. The memory system would, behind the scenes, move data between the various levels of the memory hierarchy, depending on the usage patterns it detected. This would require circuitry to examine each memory access issued by the CPU to determine where in the hierarchy to find the requested location. And then, if a particular region of addresses was frequently accessed — say, when fetching instructions in a loop — the memory system would arrange for those accesses to be mapped to the fastest memory component and automatically move the loop instructions there. All of this machinery would be transparent to the programmer: the program would simply fetch instructions and access data and the memory system would handle the rest.

Could the memory system automatically arrange for the right data to be in the right place at the right time? Cray was deeply skeptical of this approach. He famously quipped “that you can’t fake what you haven’t got”. Wouldn’t the programmer, with her knowledge of how data was going to be used by a particular program, be able to do a better job by explicitly managing the memory hierarchy?

It turns out that when running general-purpose programs, it is possible to build an automatically managed, low-latency, high-capacity hierarchical memory system that appears as one large, uniform

memory. What's the insight that makes this possible? That's the topic of the next section.

The Locality Principle

Keep the most often-used data in a small, fast SRAM (often local to CPU chip)

Refer to Main Memory only rarely, for remaining data.

The reason this strategy works: LOCALITY

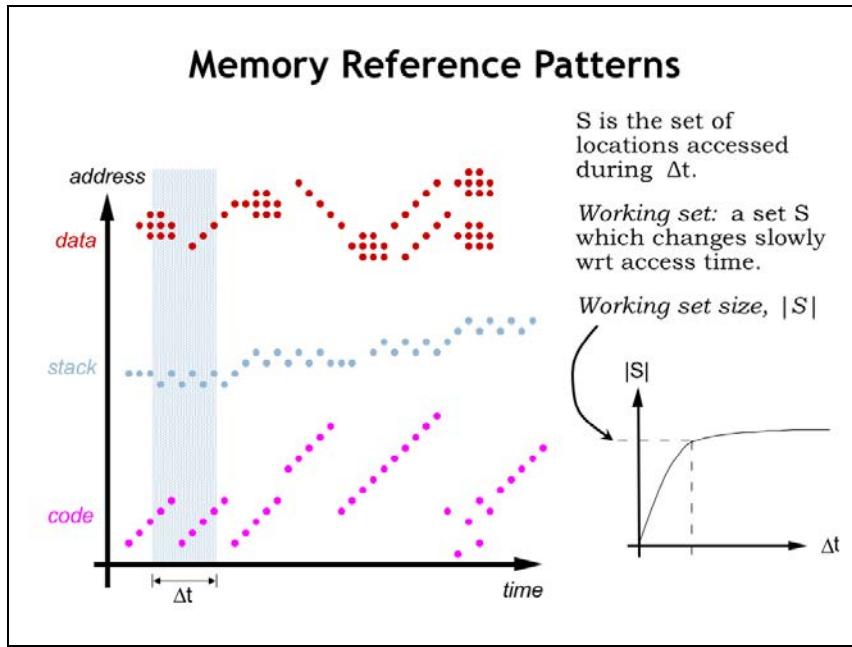
Locality of Reference:

Access to address X at time t implies that access to address $X + \Delta X$ at time $t + \Delta t$ becomes more probable as ΔX and Δt approach zero.

So, how can the memory system arrange for the right data to be in the right place at the right time? Our goal is to have the frequently-used data in some fast SRAM. That means the memory system will have to be able to predict which memory locations will be accessed. And to keep the overhead of moving data into and out of SRAM manageable, we'd like to amortize the cost of the move over many accesses. In other words we want any block of data we move into SRAM to be accessed many times.

When not in SRAM, data would live in the larger, slower DRAM that serves as main memory. If the system is working as planned, DRAM accesses would happen infrequently, e.g., only when it's time to bring another block of data into SRAM.

If we look at how programs access memory, it turns out we *can* make accurate predictions about which memory locations will be accessed. The guiding principle is *locality of reference* which tells us that if there's an access to address X at time t, it's very probable that the program will access a nearby location in the near future.



To understand why programs exhibit locality of reference, let's look at how a running program accesses memory.

Instruction fetches are quite predictable. Execution usually proceeds sequentially since most of the time the next instruction is fetched from the location after that of the current instruction. Code that loops will repeatedly fetch the same sequence of instructions, as shown here on the left of the time line. There will of course be branches and subroutine calls that interrupt sequential execution, but then we're back to fetching instructions from consecutive locations. Some programming constructs, e.g., method dispatch in object-oriented languages, can produce scattered references to very short code sequences (as shown on the right of the time line) but order is quickly restored.

This agrees with our intuition about program execution. For example, once we execute the first instruction of a procedure, we'll almost certainly execute the remaining instructions in the procedure. So if we arranged for all the code of a procedure to move to SRAM when the procedure's first instruction was fetched, we'd expect that many subsequent instruction fetches could be satisfied by the SRAM. And although fetching the first word of a block from DRAM has relatively long latency, the DRAM's fast column accesses will quickly stream the remaining words from sequential addresses. This will amortize the cost of the initial access over the whole sequence of transfers.

The story is similar for accesses by a procedure to its arguments and local variables in the current stack frame. Again there will be many accesses to a small region of memory during the span of time we're executing the procedure's code.

Data accesses generated by LD and ST instructions also exhibit locality. The program may be accessing the components of an object or struct. Or it may be stepping through the elements of an array. Sometimes information is moved from one array or data object to another, as shown by the data accesses on the right of the timeline.

Using simulations we can estimate the number of different locations that will be accessed over a particular span of time. What we discover when we do this is the notion of a *working set* of locations that are accessed repeatedly. If we plot the size of the working set as a function of the size of the time

interval, we see that the size of the working set levels off. In other words once the time interval reaches a certain size the number of locations accessed is approximately the same independent of when in time the interval occurs.

As we see in our plot to the left, the actual addresses accessed will change, but the number of *different* addresses during the time interval will, on the average, remain relatively constant and, surprisingly, not all that large!

This means that if we can arrange for our SRAM to be large enough to hold the working set of the program, most accesses will be able to be satisfied by the SRAM. We'll occasionally have to move new data into the SRAM and old data back to DRAM, but the DRAM access will occur less frequently than SRAM accesses. We'll work out the mathematics in a slide or two, but you can see that thanks to locality of reference we're on track to build a memory out of a combination of SRAM and DRAM that performs like an SRAM but has the capacity of the DRAM.

Caches

Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations

- Very fast access if data is cached, otherwise accesses slower, larger cache or memory
- Exploits the locality principle

Computer systems often use multiple levels of caches

Caching widely applied beyond hardware (e.g., web caches)

The SRAM component of our hierarchical memory system is called a *cache*. It provides low-latency access to recently-accessed blocks of data. If the requested data is in the cache, we have a *cache hit* and the data is supplied by the SRAM.

If the requested data is not in the cache, we have a *cache miss* and a block of data containing the requested location will have to be moved from DRAM into the cache. The locality principle tells us that we should expect cache hits to occur much more frequently than cache misses.

Modern computer systems often use multiple levels of SRAM caches. The levels closest to the CPU are smaller but very fast, while the levels further away from the CPU are larger and hence slower. A miss at one level of the cache generates an access to the next level, and so on until a DRAM access is needed to satisfy the initial request.

Caching is used in many applications to speed up access to frequently-accessed data. For example, your browser maintains a cache of frequently-accessed web pages and uses its local copy of the web page if it determines the data is still valid, avoiding the delay of transferring the data over the Internet.

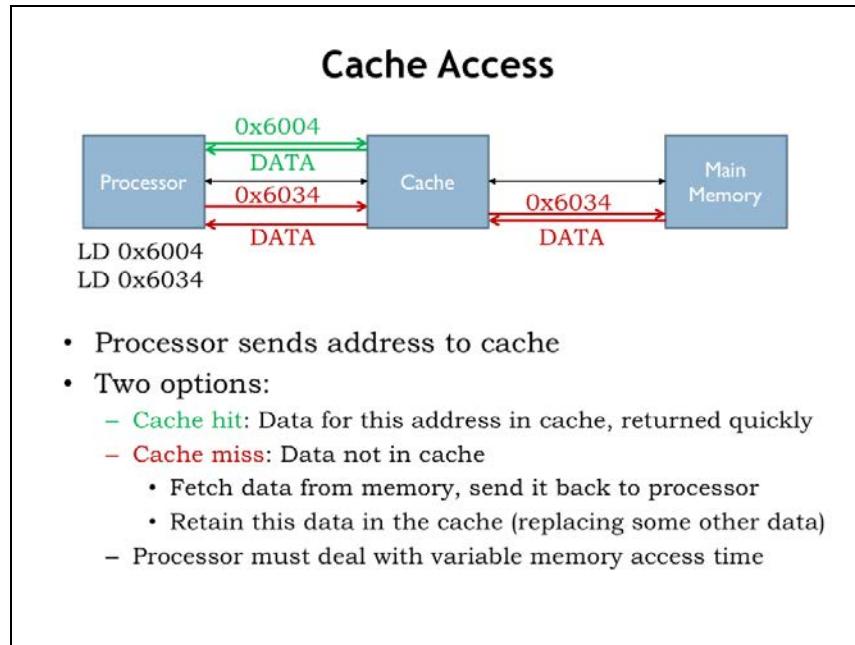
A Typical Memory Hierarchy

- Everything is a cache for something else...

	Access time	Capacity	Managed By
On the datapath	Registers	1 cycle	1 KB Software/Compiler
On chip	Level 1 Cache	2-4 cycles	32 KB Hardware
	Level 2 Cache	10 cycles	256 KB Hardware
	Level 3 Cache	40 cycles	10 MB Hardware
Other chips	Main Memory	200 cycles	10 GB Software/OS
Mechanical devices	Flash Drive	10-100us	100 GB Software/OS
	Hard Disk	10ms	1 TB Software/OS

Here's an example memory hierarchy that might be found on a modern computer. There are three levels on-chip SRAM caches, followed by DRAM main memory and a flash-memory cache for the hard disk drive. The compiler is responsible for deciding which data values are kept in the CPU registers and which values require the use of LDs and STs. The 3-level cache and accesses to DRAM are managed by circuitry in the memory system. After that the access times are long enough (many hundreds of instruction times) that the job of managing the movement of data between the lower levels of the hierarchy is turned over to software.

Today we're discussing how the on-chip caches work. In a later lecture, we'll discuss how the software manages main memory and non-volatile storage devices. Whether managed by hardware or software, each layer of the memory system is designed to provide lower-latency access to frequently-accessed locations in the next, slower layer. But, as we'll see, the implementation strategies will be quite different in the slower layers of the hierarchy.



Okay, let's review our plan. The processor starts an access by sending an address to the cache. If data for the requested address is held in the cache, it's quickly returned to the CPU.

If the data we request is not in the cache, we have a cache miss, so the cache has to make a request to main memory to get the data, which it then returns to processor. Typically the cache will remember the newly fetched data, possibly replacing some older data in the cache.

Suppose a cache access takes 4 ns and a main memory access takes 40 ns. Then an access that hits in the cache has a latency of 4 ns, but an access that misses in the cache has a latency of 44 ns. The processor has to deal with the variable memory access time, perhaps by simply waiting for the access to complete, or, in modern hyper-threaded processors, it might execute an instruction or two from another programming thread.

Cache Metrics

$$\text{Hit Ratio: } HR = \frac{\text{hits}}{\text{hits} + \text{misses}} = 1 - MR$$

$$\text{Miss Ratio: } MR = \frac{\text{misses}}{\text{hits} + \text{misses}} = 1 - HR$$

Average Memory Access Time (AMAT):

$$AMAT = HitTime + MissRatio \times MissPenalty$$

- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times AMAT_{L2} =$$

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times (HitTime_{L2} + MissRatio_{L2} \times AMAT_{L3}) = \dots$$

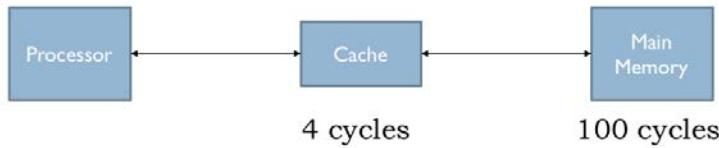
The hit and miss ratios tell us the fraction of accesses which are cache hits and the fraction of accesses which are cache misses. Of course, the ratios will sum to 1.

Using these metrics we can compute the average memory access time (AMAT). Since we always check in the cache first, every access includes the cache access time (called the hit time). If we miss in the cache, we have to take the additional time needed to access main memory (called the miss penalty). But the main memory access only happens on some fraction of the accesses: the miss ratio tells us how often that occurs.

So the AMAT can be computed using the formula shown here. The lower the miss ratio (or, equivalently, the higher the hit ratio), the smaller the average access time. Our design goal for the cache is to achieve a high hit ratio.

If we have multiple levels of cache, we can apply the formula recursively to calculate the AMAT at each level of the memory. Each successive level of the cache is slower, *i.e.*, has a longer hit time, which is offset by lower miss ratio because of its increased size.

Example: How High of a Hit Ratio?



What hit ratio do we need to break even?
(Main memory only: AMAT = 100)

$$100 = 4 + (1 - HR) \times 100 \Rightarrow HR = 4\%$$

What hit ratio do we need to achieve AMAT = 5 cycles?

$$5 = 4 + (1 - HR) \times 100 \Rightarrow HR = 99\%$$

Let's try out some numbers. Suppose the cache takes 4 processor cycles to respond, and main memory takes 100 cycles. Without the cache, each memory access would take 100 cycles. With the cache, a cache hit takes 4 cycles, and a cache miss takes 104 cycles.

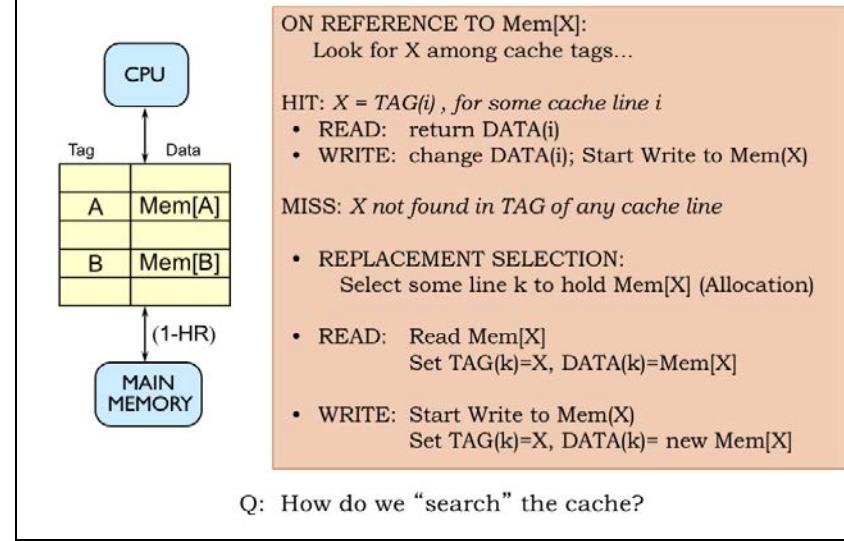
What hit ratio is needed so that the AMAT with the cache is 100 cycles, the break-even point? Using the AMAT formula from the previously slide, we see that we only need a hit ratio of 4% in order for memory system of the Cache + Main Memory to perform as well as Main Memory alone. The idea, of course, is that we'll be able to do much better than that.

Suppose we wanted an AMAT of 5 cycles. Clearly most of the accesses would have to be cache hits. We can use the AMAT formula to compute the necessary hit ratio. Working through the arithmetic we see that 99% of the accesses must be cache hits in order to achieve an average access time of 5 cycles.

Could we expect to do that well when running actual programs? Happily, we can come close. In a simulation of the Spec CPU2000 Benchmark, the hit ratio for a standard-size level 1 cache was measured to be 97.5% over some ~10 trillion accesses.

[See the “All benchmarks” arithmetic-mean table at
<http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/>]

Basic Cache Algorithm



Here's a start at building a cache. The cache will hold many different blocks of data; for now let's assume each block is an individual memory location. Each data block is *tagged* with its address. A combination of a data block and its associated address tag is called a cache line.

When an address is received from the CPU, we'll search the cache looking for a block with a matching address tag. If we find a matching address tag, we have a cache hit. On a read access, we'll return the data from the matching cache line. On a write access, we'll update the data stored in the cache line and, at some point, update the corresponding location in main memory.

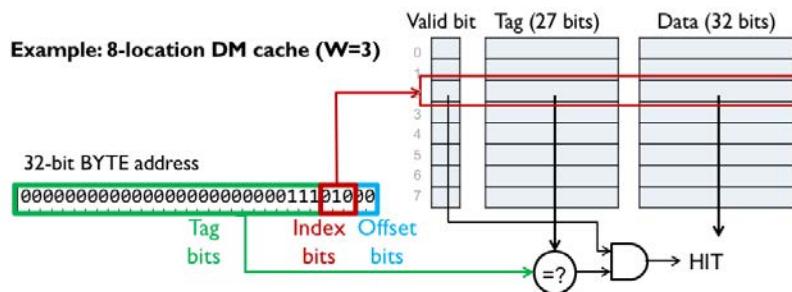
If no matching tag is found, we have a cache miss. So we'll have to choose a cache line to use to hold the requested data, which means that some previously cached location will no longer be found in the cache. For a read operation, we'll fetch the requested data from main memory, add it to the cache (updating the tag and data fields of the cache line) and, of course, return the data to the CPU. On a write, we'll update the tag and data in the selected cache line and, at some point, update the corresponding location in main memory.

So the contents of the cache is determined by the memory requests made by the CPU. If the CPU requests a recently-used address, chances are good the data will still be in the cache from the previous access to the same location. As the working set slowly changes, the cache contents will be updated as needed. If the entire working set can fit into the cache, most of the requests will be hits and the AMAT will be close to the cache access time. So far, so good!

Of course, we'll need to figure how to quickly search the cache, *i.e.*, we'll need a fast way to answer the question of whether a particular address tag can be found in some cache line. That's our next topic.

Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with 2^W lines):
 - Index into cache with W address bits (the **index bits**)
 - Read out valid bit, tag, and data
 - If valid bit == 1 and tag matches upper address bits, HIT



The simplest cache hardware consists of an SRAM with a few additional pieces of logic. The cache hardware is designed so that each memory location in the CPU's address space maps to a particular cache line, hence the name *direct-mapped (DM) cache*. There are, of course, many more memory locations than there are cache lines, so many addresses are mapped to the same cache line and the cache will only be able to hold the data for one of those addresses at a time.

The operation of a DM cache is straightforward. We'll use part of the incoming address as an index to select a single cache line to be searched. The search consists of comparing the rest of the incoming address with the address tag of the selected cache line. If the tag matches the address, there's a cache hit and we can immediately use the data in the cache to satisfy the request.

In this design, we've included an additional *valid bit* which is 1 when the tag and data fields hold valid information. The valid bit for each cache line is initialized to 0 when the cache is powered on, indicating that all cache lines are empty. As data is brought into the cache, the valid bit is set to 1 when the cache line's tag and data fields are filled. The CPU can request that the valid bit be cleared for a particular cache line — this is called *flushing the cache*. If, for example, the CPU initiates a read from disk, the disk hardware will read its data into a block of main memory, so any cached values for that block will be out-of-date. So the CPU will flush those locations from the cache by marking any matching cache lines as invalid.

Let's see how this works using a small DM cache with 8 lines where each cache line contains a single word (4 bytes) of data. Here's a CPU request for the location at byte address 0xE8. Since there are 4 bytes of data in each cache line, the bottom 2 address bits indicate the appropriate byte offset into the cached word. Since the cache deals only with word accesses, the byte offset bits aren't used.

Next, we'll need to use 3 address bits to select which of the 8 cache lines to search. We choose these cache index bits from the low-order bits of the address. Why? Well, it's because of locality. The principle of locality tells us that it's likely that the CPU will be requesting nearby addresses and for the cache to perform well, we'd like to arrange for nearby locations to be able to be held in the cache at the same time. This means that nearby locations will have to be mapped to different cache lines. The

addresses of nearby locations differ in their low-order address bits, so we'll use those bits as the cache index bits — that way nearby locations will map to different cache lines.

The data, tag and valid bits selected by the cache line index are read from the SRAM. To complete the search, we check the remaining address against the tag field of the cache. If they're equal and the valid bit is 1, we have a cache hit, and the data field can be used to satisfy the request.

How come the tag field isn't 32 bits, since we have a 32-bit address? We could have done that, but since all values stored in cache line 2 will have the same index bits (0b010), we saved a few bits of SRAM and chose not save those bits in the tag. In other words, there's no point in using SRAM to save bits we can generate from the incoming address.

So the cache hardware in this example is an 8-location by 60 bit SRAM plus a 27-bit comparator and a single AND gate. The cache access time is the access time of the SRAM plus the propagation delays of the comparator and AND gate. About as simple and fast as we could hope for.

The downside of the simplicity is that for each CPU request, we're only looking in a single cache location to see if the cache holds the desired data. Not much of search is it? But the mapping of addresses to cache lines helps us out here. Using the low-order address bit as the cache index, we've arranged for nearby locations to be mapped to different cache lines. So, for example, if the CPU were executing an 8-instruction loop, all 8 instructions can be held in the cache at the same time. A more complicated search mechanism couldn't improve on that. The bottom line: this extremely simple search is sufficient to get good cache hit ratios for the cases we care about.

Example: Direct-Mapped Caches

64-line direct-mapped cache → 64 indexes → **6 index bits**

Read Mem[0x400C]

 TAG: 0x40
 INDEX: 0x3
 OFFSET: 0x0
 HIT, DATA 0x42424242

Would 0x4008 hit?
 INDEX: 0x2 → tag mismatch → miss

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	0	0x000058	0x00000007
3	1	0x000040	0x42424242
4	1	0x000007	0x6FBA2381
⋮	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

What are the addresses of data in indexes 0, 1, and 2?

TAG: 0x58 → 0101 1000 iiiii ii00 (substitute line # for iiiii) → 0x5800, 0x5804, 0x5808

Part of the address (index bits) is **encoded in the location!**
 Tag + Index bits unambiguously identify the data's address

Let's try a few more examples, in this case using a DM cache with 64 lines.

Suppose the cache gets a read request for location 0x400C. To see how the request is processed, we first write the address in binary so we can easily divide it into the offset, index and tag fields. For this address the offset bits have the value 0, the cache line index bits have the value 3, and the tag bits have the value 0x40. So the tag field of cache line 3 is compared with the tag field of the address. Since there's a match, we have a cache hit and the value in the data field of cache line can be used to satisfy the request.

Would an access to location 0x4008 be a cache hit? This address is similar to that in our first example, except the cache line index is now 2 instead of 3. Looking in cache line 2, we see that its tag field (0x58) doesn't match the tag field in the address (0x40), so this access would be a cache miss.

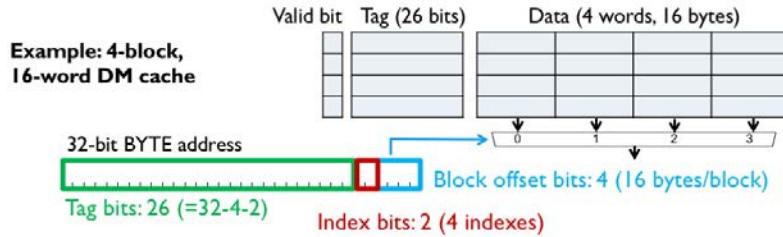
What are the addresses of the words held by cache lines 0, 1, and 2, all of which have the same tag field? Well, we can run the address matching process backwards! For an address to match these three cache lines it would look like the binary shown here, where we've used the information in the cache tag field to fill in the high-order address bits and low-order address bits will come from the index value. If we fill in the indices 0, 1, and 2, then convert the resulting binary to hex we get 0x5800, 0x5804, and 0x5808 as the addresses for the data held in cache lines 0, 1, and 2.

Note that the complete address of the cached locations is formed by combining the tag field of the cache line with the index of the cache line. We of course need to be able to recover the complete address from the information held in the cache so it can be correctly compared against address requests from the CPU.

Block Size

Take advantage of locality: increase block size

- Another advantage: Reduces size of tag memory!
- Potential disadvantage: Fewer blocks in the cache



We can tweak the design of the DM cache a little to take advantage of locality and save some of the overhead of tag fields and valid bits.

We can increase the size of the data field in a cache from 1 word to 2 words, or 4 words, etc. The number of data words in each cache line is called the *block size* and is always a power of two. Using a larger block size makes sense. If there's a high probability of accessing nearby words, why not fetch a larger block of words on a cache miss, trading the increased cost of the miss against the increased probability of future hits.

Compare the 16-word DM cache shown here with a block size of 4 with a different 16-word DM cache with a block size of 1. In this cache for every 128 bits of data there are 27 bits of tags and valid bit, so ~17% of the SRAM bits are overhead in the sense that they're not being used to store data. In the cache with block size 1, for every 32 bits of data there are 27 bits of tag and valid bit, so ~46% of the SRAM bits are overhead. So a larger block size means we'll be using the SRAM more efficiently.

Since there are 16 bytes of data in each cache line, there are now 4 offset bits. The cache uses the high-order two bits of the offset to select which of the 4 words to return to the CPU on a cache hit.

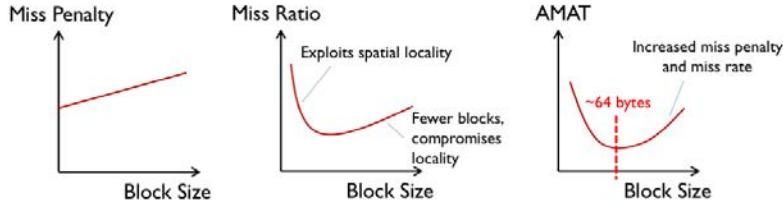
There are 4 cache lines, so we'll need two cache line index bits from the incoming address.

And, finally, the remaining 26 address bits are used as the tag field.

Note that there's only a single valid bit for each cache line, so either the entire 4-word block is present in the cache or it's not. Would it be worth the extra complication to support caching partial blocks? Probably not. Locality tells us that we'll probably want those other words in the near future, so having them in the cache will likely improve the hit ratio.

Block Size Tradeoffs

- Larger block sizes...
 - Take advantage of spatial locality
 - Incur larger miss penalty since it takes longer to transfer the block into the cache
 - Can increase the average hit time and miss rate
- Average Access Time (AMAT) = HitTime + MissPenalty*MR



What's the tradeoff between block size and performance? We've argued that increasing the block size from 1 was a good idea. Is there a limit to how large blocks should be? Let's look at the costs and benefits of an increased block size.

With a larger block size we have to fetch more words on a cache miss and the miss penalty grows linearly with increasing block size. Note that since the access time for the first word from DRAM is quite high, the increased miss penalty isn't as painful as it might be.

Increasing the block size past 1 reduces the miss ratio since we're bringing words into the cache that will then be cache hits on subsequent accesses. Assuming we don't increase the overall cache capacity, increasing the block size means we'll make a corresponding reduction in the number of cache lines. Reducing the number of lines impacts the number of separate address blocks that can be accommodated in the cache. As we saw in the discussion on the size of the working set of a running program, there are a certain number of separate regions we need to accommodate to achieve a high hit ratio: program, stack, data, etc. So we need to ensure there are a sufficient number of blocks to hold the different addresses in the working set. The bottom line is that there is an optimum block size that minimizes the miss ratio and increasing the block size past that point will be counterproductive.

Combining the information in these two graphs, we can use the formula for AMAT to choose the block size that gives us the best possible AMAT. In modern processors, a common block size is 64 bytes (16 words).

Direct-Mapped Cache Problem: Conflict Misses

Word Address	Cache Line index	Hit/ Miss
Loop A: Pgm at 1024, data at 37:	1024 0 HIT 37 37 HIT 1025 1 HIT 38 38 HIT 1026 2 HIT 39 39 HIT 1024 0 HIT 37 37 HIT ...	
Loop B: Pgm at 1024, data at 2048:	1024 0 MISS 2048 0 MISS 1025 1 MISS 2049 1 MISS 1026 2 MISS 2050 2 MISS 1024 0 MISS 2048 0 MISS ...	Assume: 1024-line DM cache Block size = 1 word Consider looping code, in steady state Assume WORD, not BYTE, addressing

Inflexible mapping (each address can only be in one cache location) → **Conflict misses!**

DM caches do have an Achilles heel. Consider running the 3-instruction LOOPA code with the instructions located starting at word address 1024 and the data starting at word address 37 where the program is making alternating accesses to instruction and data, *e.g.*, a loop of LD instructions.

Assuming a 1024-line DM cache with a block size of 1, the steady state hit ratio will be 100% once all six locations have been loaded into the cache since each location is mapped to a different cache line.

Now consider the execution of the same program, but this time the data has been relocated to start at word address 2048. Now the instructions and data are competing for use of the same cache lines. For example, the first instruction (at address 1024) and the first data word (at address 2048) both map to cache line 0, so only one them can be in the cache at a time. So fetching the first instruction fills cache line 0 with the contents of location 1024, but then the first data access misses and then refills cache line 0 with the contents of location 2048. The data address is said to *conflict* with the instruction address. The next time through the loop, the first instruction will no longer be in the cache and it's fetch will cause a cache miss, called a *conflict miss*. So in the steady state, the cache will never contain the word requested by the CPU.

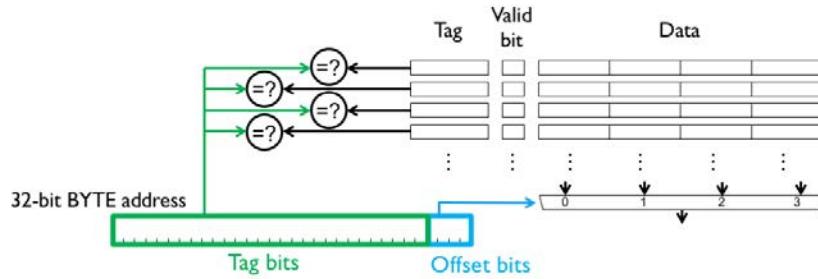
This is very unfortunate! We were hoping to design a memory system that offered the simple abstraction of a flat, uniform address space. But in this example we see that simply changing a few addresses results in the cache hit ratio dropping from 100% to 0%. The programmer will certainly notice her program running 10 times slower!

So while we like the simplicity of DM caches, we'll need to make some architectural changes to avoid the performance problems caused by conflict misses.

Fully-Associative Cache

Opposite extreme: Any address can be in any location

- No cache index!
- **Flexible** (no conflict misses)
- **Expensive**: Must compare tags of all entries in parallel to find matching one (can do this in hardware, this is called a CAM)



A fully-associative (FA) cache has a tag comparator for each cache line. So the tag field of *every* cache line in a FA cache is compared with the tag field of the incoming address. Since all cache lines are searched, a particular memory location can be held in any cache line, which eliminates the problems of address conflicts causing conflict misses. The cache shown here can hold 4 different 4-word blocks, regardless of their address. The example from the end of the previous segment required a cache that could hold two 3-word blocks, one for the instructions in the loop, and one for the data words. This FA cache would use two of its cache lines to perform that task and achieve a 100% hit ratio regardless of the addresses of the instruction and data blocks.

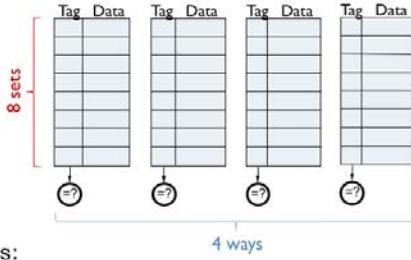
FA caches are very flexible and have high hit ratios for most applications. Their only downside is cost: the inclusion of a tag comparator for each cache line to implement the parallel search for a tag match adds substantially the amount of circuitry required when there are many cache lines. Even the use of hybrid storage/comparison circuitry, called a content-addressable memory, doesn't make a big dent in the overall cost of a FA cache.

DM caches searched only a single cache line. FA caches search all cache lines. Is there a happy middle ground where some small number of cache lines are searched in parallel?

Yes! If you look closely at the diagram of the FA cache shown here, you'll see it looks like four 1-line DM caches operating in parallel. What would happen if we designed a cache with four multi-line DM caches operating in parallel?

N-way Set-Associative Cache

- Compromise between direct-mapped and fully associative
 - Nomenclature:
 - # Rows = # Sets
 - # Columns = # Ways
 - Set size = #ways
= “set associativity”
(e.g., 4-way → 4 entries/set)
 - compare all tags from all ways in parallel
- An N-way cache can be seen as:
 - N direct-mapped caches in parallel
- Direct-mapped and fully-associative are just special cases of N-way set-associative



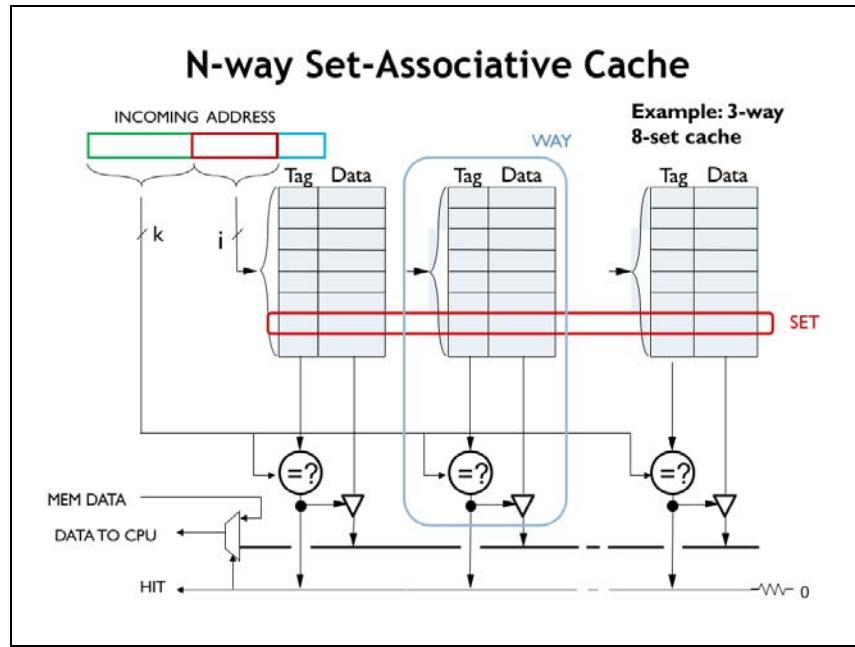
The result would be what we call an 4-way set-associative (SA) cache. An N-way SA cache is really just N DM caches (let's call them sub-caches) operating in parallel. Each of the N sub-caches compares the tag field of the incoming address with the tag field of the cache line selected by the index bits of the incoming address. The N cache lines searched on a particular request form a search *set* and the desired location might be held in any member of the set.

The 4-way SA cache shown here has 8 cache lines in each sub-cache, so each set contains 4 cache lines (one from each sub-cache) and there are a total of 8 sets (one for each line of the sub-caches).

An N-way SA cache can accommodate up to N blocks whose addresses map to the same cache index. So access to up to N blocks with conflicting addresses can still be accommodated in this cache without misses. This a big improvement over a DM cache where an address conflict will cause the current resident of a cache line to be evicted in favor of the new request.

And an N-way SA cache can have a very large number of cache lines but still only have to pay the cost of N tag comparators. This is a big improvement over a FA cache where a large number of cache lines would require a large number of comparators.

So N-way SA caches are a good compromise between a conflict-prone DM cache and the flexible but very expensive FA cache.



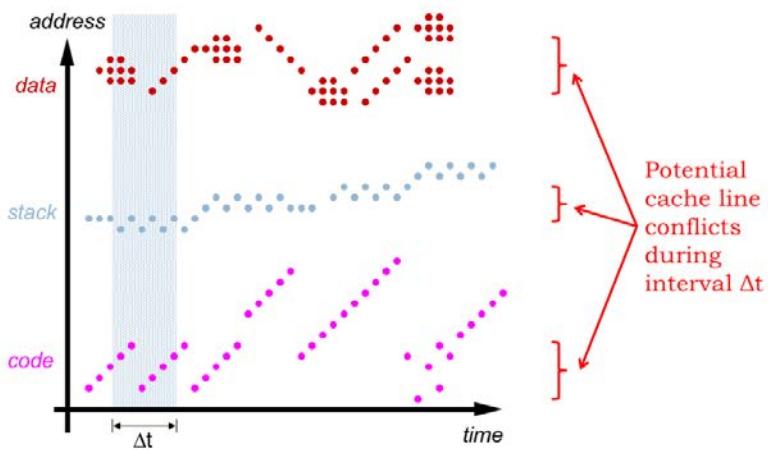
Here's a slightly more detailed diagram, in this case of a 3-way 8-set cache. Note that there's no constraint that the number of ways be a power of two since we aren't using any address bits to select a particular way. This means the cache designer can fine tune the cache capacity to fit her space budget.

Just to review the terminology: the N cache lines that will be searched for a particular cache index are called a set. And each of N sub-caches is called a way.

The hit logic in each *way* operates in parallel with the logic in other ways. Is it possible for a particular address to be matched by more than one way? That possibility isn't ruled out by the hardware, but the SA cache is managed so that doesn't happen. Assuming we write the data fetched from DRAM during a cache miss into a single sub-cache - we'll talk about how to choose that way in a minute — there's no possibility that more than one sub-cache will ever match an incoming address.

“Let me count the ways.”

Elizabeth Barrett Browning



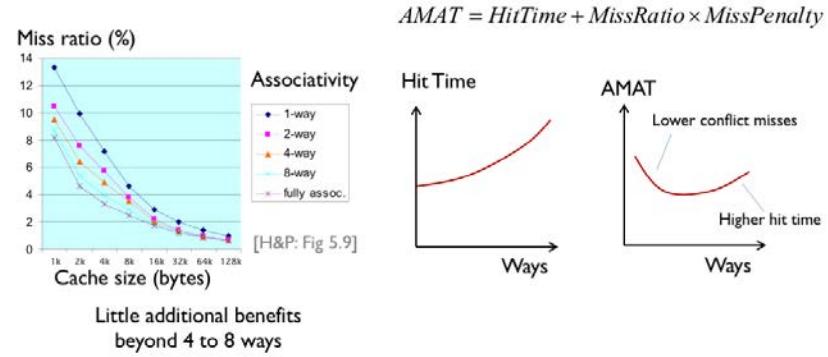
How many ways to do we need? We'd like enough ways to avoid the cache line conflicts we experienced with the DM cache. Looking at the graph we saw earlier of memory accesses vs. time, we see that in any time interval there are only so many potential address conflicts that we need to worry about.

The mapping from addresses to cache lines is designed to avoid conflicts between neighboring locations. So we only need to worry about conflicts between the different regions: code, stack and data. In the examples shown here there are three such regions, maybe 4 if you need two data regions to support copying from one data region to another. If the time interval is particularly large, we might need double that number to avoid conflicts between accesses early in the time interval and accesses late in the time interval.

The point is that a small number of ways should be sufficient to avoid most cache line conflicts in the cache.

Associativity Tradeoffs

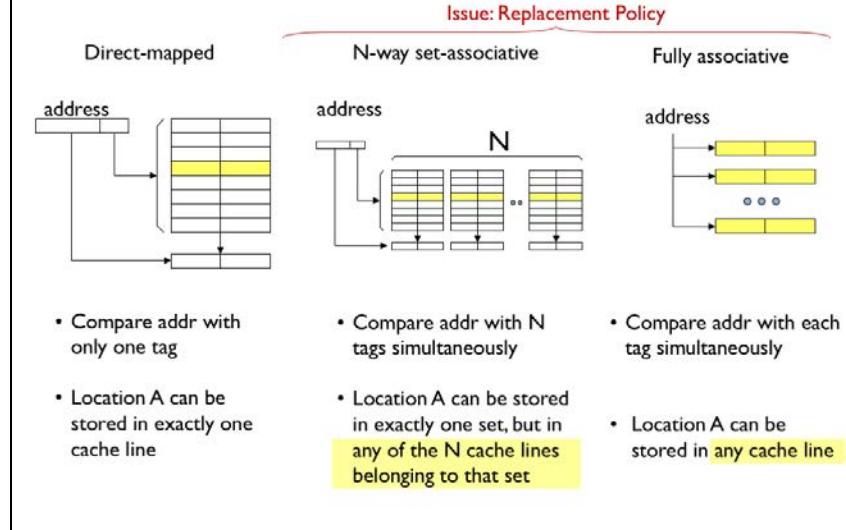
- More ways...
 - Reduce conflict misses
 - Increase hit time



As with block size, it's possible to have too much of a good thing: there's an optimum number of ways that minimizes the AMAT. Beyond that point, the additional circuitry needed to combine the hit signals from a large number of ways will start have a significant propagation delay of its own, adding directly to the cache hit time and the AMAT.

More to the point, the chart on the left shows that there's little additional impact on the miss ratio beyond 4 to 8 ways. For most programs, an 8-way set-associative cache with a large number of sets will perform on a par with the much more-expensive FA cache of equivalent capacity.

Associativity Implies Choices



There's one final issue to resolve with SA and FA caches. When there's a cache miss, which cache line should be chosen to hold the data that will be fetched from main memory? That's not an issue with DM caches, since each data block can only be held in one particular cache line, determined by its address. But in N-way SA caches, there are N possible cache lines to choose from, one in each of the ways. And in a FA cache, any of the cache lines can be chosen.

So, how to choose? Our goal is to choose to replace the contents of the cache line which will minimize the impact on the hit ratio in the future.

Replacement Policies

- Optimal policy (Belady's MIN): Replace the block that is accessed furthest in the future
 - Requires knowing the future...
- Idea: Predict the future from looking at the past
 - If a block has not been used recently, it's often less likely to be accessed in the near future (a locality argument)
- **Least Recently Used (LRU):** Replace the block that was accessed furthest in the past
 - Works well in practice
 - Need to keep ordered list of N items → N! orderings
→ $O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic
 - Caches often implement cheaper approximations of LRU
- Other policies:
 - First-In, First-Out (least recently replaced)
 - Random: Choose a candidate at random
 - Not very good, but does not have adversarial access patterns

The optimal choice is to replace the block that is accessed furthest in the future (or perhaps is never accessed again). But that requires knowing the future...

Here's an idea: let's predict future accesses by looking at recent accesses and applying the principle of locality. If a block has not been recently accessed, it's less likely to be accessed in the near future.

That suggests the least-recently-used replacement strategy, usually referred to as LRU: replace the block that was accessed furthest in the past. LRU works well in practice, but requires us to keep a list ordered by last use for each set of cache lines, which would need to be updated on each cache access. When we needed to choose which member of a set to replace, we'd choose the last cache line on this list. For an 8-way SA cache there are 8! possible orderings, so we'd need $\log_2(8!) = 16$ state bits to encode the current ordering. The logic to update these state bits on each access isn't cheap; basically you need a lookup table to map the current 16-bit value to the next 16-bit value. So most caches implement an approximation to LRU where the update function is much simpler to compute.

There are other possible replacement policies: First-in, first-out, where the oldest cache line is replaced regardless of when it was last accessed. And Random, where some sort of pseudo-random number generator is used to select the replacement.

All replacement strategies except for random can be defeated. If you know a cache's replacement strategy you can design a program that will have an abysmal hit rate by accessing addresses you know the cache just replaced. I'm not sure I care about how well a program designed to get bad performance runs on my system, but the point is that most replacement strategies will occasionally cause a particular program to execute much more slowly than expected.

When all is said and done, an LRU replacement strategy or a close approximation is a reasonable choice.

Write Policy

Write-through: CPU writes are cached, but also written to main memory immediately (stalling the CPU until write is completed). Memory always holds current contents

- Simple, slow, wastes bandwidth

Write-behind: CPU writes are cached; writes to main memory may be buffered. CPU keeps executing while writes are completed in the background

- Faster, still uses lots of bandwidth

Write-back: CPU writes are cached, but not written to main memory until we replace the block. Memory contents can be “stale”

- Fastest, low bandwidth, more complex
- Commonly implemented in current systems

Okay, one more cache design decision to make, then we’re done!

How should we handle memory writes in the cache? Ultimately we’ll need update main memory with the new data, but when should that happen?

The most obvious choice is to perform the write immediately. In other words, whenever the CPU sends a write request to the cache, the cache then performs the same write to main memory. This is called *write-through*. That way main memory always has the most up-to-date value for all locations. But this can be slow if the CPU has to wait for a DRAM write access — writes could become a real bottleneck! And what if the program is constantly writing a particular memory location, *e.g.*, updating the value of a local variable in the current stack frame? In the end we only need to write the last value to main memory. Writing all the earlier values is waste of memory bandwidth.

Suppose we let the CPU continue execution while the cache waits for the write to main memory to complete — this is called *write-behind*. This will overlap execution of the program with the slow writes to main memory. Of course, if there’s another cache miss while the write is still pending, everything will have to wait at that point until both the write and subsequent refill read finish, since the CPU can’t proceed until the cache miss is resolved.

The best strategy is called *write-back* where the contents of the cache are updated and the CPU continues execution immediately. The updated cache value is only written to main memory when the cache line is chosen as the replacement line for a cache miss. This strategy minimizes the number of accesses to main memory, preserving the memory bandwidth for other operations. This is the strategy used by most modern processors.

Write-Back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: $\text{TAG}(X) == \text{Tag}[i]$, for some cache block i

- READ: return Data[i]
- ~~WRITE: change Data[i]; Start Write to Mem[X]~~

MISS: $\text{TAG}(X)$ not found in tag of any cache block that X can map to

•REPLACEMENT SELECTION:

- Select some line k to hold Mem[X]
- ~~Write Back: Write Data[k] to Mem[Address from Tag[k]]~~

- READ: Read Mem[X]
➤ Set Tag[k] = TAG(X), Data[k] = Mem[X]

- ~~WRITE: Start Write to Mem[X]~~
➤ Set Tag[k] = TAG(X), Data[k] = new Mem[X]

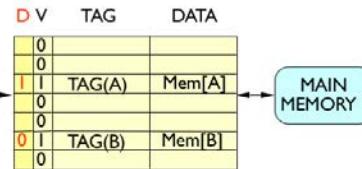
Write-back is easy to implement. Returning to our original cache recipe, we simply eliminate the start of the write to main memory when there's a write request to the cache. We just update the cache contents and leave it at that.

However, replacing a cache line becomes a more complex operation, since we can't reuse the cache line without first writing its contents back to main memory in case they had been modified by an earlier write access.

Hmm. Seems like this does a write-back of all replaced cache lines whether or not they've been written to.

Write-Back with “Dirty” Bits

Add 1 bit per block to record whether block has been written to. Only write back dirty blocks.



ON REFERENCE TO $\text{Mem}[X]$: Look for $\text{TAG}(X)$ among tags...

HIT: $\text{TAG}(X) == \text{Tag}[i]$, for some cache block i

- READ: return $\text{Data}[i]$
- WRITE: change $\text{Data}[i]$ Start Write to $\text{Mem}[X]$ $\text{D}[i]=1$

MISS: $\text{TAG}(X)$ not found in tag of any cache block that X can map to

• REPLACEMENT SELECTION:

- Select some block k to hold $\text{Mem}[X]$

▪ If $\text{D}[k] == 1$ (Writeback) Write $\text{Data}[k]$ to $\text{Mem}[\text{Address of Tag}[k]]$

- READ: Read $\text{Mem}[X]$; Set $\text{Tag}[k] = \text{TAG}(X)$, $\text{Data}[k] = \text{Mem}[X]$, $\text{D}[k]=0$

• WRITE: Start Write to $\text{Mem}[X]$ $\text{D}[k]=1$

➢ Set $\text{Tag}[k] = \text{TAG}(X)$, $\text{Data}[k] = \text{new Mem}[X]$

We can avoid unnecessary write-backs by adding another state bit to each cache line: the *dirty* bit. The dirty bit is set to 0 when a cache line is filled during a cache miss. If a subsequent write operation changes the data in a cache line, the dirty bit is set to 1, indicating that value in the cache now differs from the value in main memory.

When a cache line is selected for replacement, we only need to write its data back to main memory if its dirty bit is 1.

So a write-back strategy with a dirty bit gives an elegant solution that minimizes the number of writes to main memory and only delays the CPU on a cache miss if a dirty cache line needs to be written back to memory.

Summary: Cache Tradeoffs

$$AMAT = HitTime + MissRatio \times MissPenalty$$

- Larger **cache size**: Lower miss rate, higher hit time
- Larger **block size**: Trade off spatial for temporal locality, higher miss penalty
- More **associativity** (ways): Lower miss rate, higher hit time
- More intelligent **replacement**: Lower miss rate, higher cost
- **Write policy**: Lower bandwidth, more complexity
- How to navigate all these dimensions? Simulate different cache organizations on real programs

That concludes our discussion of caches, which was motivated by our desire to minimize the average memory access time by building a hierarchical memory system that had both low latency and high capacity.

There were a number of strategies we employed to achieve our goal.

Increasing the number of cache lines decreases AMAT by decreasing the miss ratio.

Increasing the block size of the cache let us take advantage of the fast column accesses in a DRAM to efficiently load a whole block of data on a cache miss. The expectation was that this would improve AMAT by increasing the number of hits in the future as accesses were made to nearby locations.

Increasing the number of ways in the cache reduced the possibility of cache line conflicts, lowering the miss ratio.

Choosing the least-recently used cache line for replacement minimized the impact of replacement on the hit ratio.

And, finally, we chose to handle writes using a write-back strategy with dirty bits.

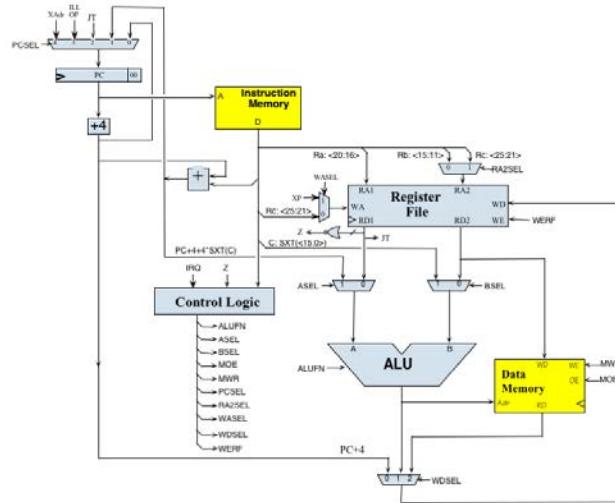
How do we make the tradeoffs among all these architectural choices? As usual, we'll simulate different cache organizations and chose the architectural mix that provides the best performance on our benchmark programs.

15: Pipelining the Beta

1. Reminder: Single-Cycle Beta
2. Single-Cycle Beta Performance
3. Pipelined Implementation
4. Why Isn't This a 20-minute Lecture?
5. Pipeline Hazards
6. Simplified Unpipelined Beta Datapath
7. 5-Stage Pipelined Datapath
8. Pipelined Control
9. Pipelined Execution Example
10. Example: Cycle 1
11. Example: Cycle 2
12. Example: Cycle 3
13. Example: Cycle 4
14. Example: Cycle 5
15. Pipeline Diagrams
16. Data Hazards
17. Resolving Hazards I
18. Resolving Data Hazards I
19. Stall Logic
20. Resolving Data Hazards II
21. Bypass Logic
22. Fully Bypassed Pipeline
23. Load-to-Use Stalls
24. Summary: Pipelining with Data Hazards
25. Compilers Can Help
26. Or take the lazy route...
27. Control Hazards I
28. Control Hazards II
29. Resolving Control Hazards
30. Resolving Control Hazards With Stalls
31. Stall Logic For Control Hazards
32. ISA Issues: Simple vs. Complex Branches
33. Resolving Hazards II
34. Resolving Hazards with Speculation I
35. Resolving Hazards with Speculation II
36. Speculation Logic For Control Hazards
37. Branch Prediction
38. Branch Delay Slots I
39. Branch Delay Slots II
40. Exceptions
41. When Can Exceptions Happen?
42. Resolving Exceptions
43. Exception Handling Logic
44. Multiple Exceptions?
45. Asynchronous Interrupts
46. Exception + Interrupt Logic
47. 5-stage Beta: Final Version

48. Reminder: Resolving Hazards

Reminder: Single-Cycle Beta



In this lecture, we're going to use the circuit pipelining techniques we learned earlier in the course to improve the performance of the 32-bit Beta CPU design we developed earlier in the course. This CPU design executes one Beta instruction per clock cycle. Hopefully you remember the design! If not, you might find it worthwhile to review "Building the Beta".

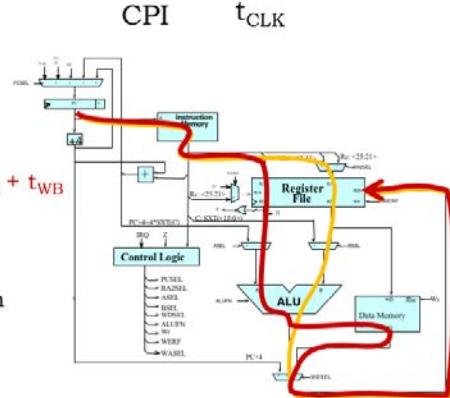
At the beginning of the clock cycle, this circuit loads a new value into the program counter, which is then sent to main memory as the address of the instruction to be executed this cycle. When the 32-bit word containing the binary encoding of the instruction is returned by the memory, the opcode field is decoded by the control logic to determine the control signals for the rest of the data path. The operands are read from the register file and routed to the ALU to perform the desired operation. For memory operations, the output of the ALU serves as the memory address and, in the case of load instructions, the main memory supplies the data to be written into the register file at the end of the cycle. PC+4 and ALU values can also be written to the register file.

The clock period of the Beta is determined by the cumulative delay through all the components involved in instruction execution. Today's question is: how can we make this faster?

Single-Cycle Beta Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Clock Cycle}}$$

- CPI = 1
 - t_{CLK} = Longest path for any instruction
- $t_{\text{CLK}} \approx t_{\text{IFETCH}} + t_{\text{RF}} + t_{\text{ALU}} + t_{\text{MEM}} + t_{\text{WB}}$
- Slow
 - Inflexible: Instructions with smaller critical path cannot execute faster



We can characterize the time spent executing a program as the product of three terms. The first term is the total number of instructions executed. Since the program usually contains loops and procedure calls, many of the encoded instructions will be executed many times. We want the total count of instructions executed, not the static size of the program as measured by the number of encoded instructions in memory. The second term is the average number of clock cycles it takes to execute a single instruction. And the third term is the duration of a single clock cycle.

As CPU designers it's the last two terms which are under our control: the cycles per instruction (CPI) and the clock period (t_{CLK}). To affect the first term, we would need to change the ISA or write a better compiler!

Our design for the Beta was able to execute every instruction in a single clock cycle, so our CPI is 1. As we discussed in the previous slide, t_{CLK} is determined by the longest path through the Beta circuitry. For example, consider the execution of an OP-class instruction, which involves two register operands and an ALU operation. The arrow shows all the components that are involved in the execution of the instruction. Aside from a few MUXes, the main memory, register file, and ALU must all have time to do their thing.

The worst-case execution time is for the LD instruction. In one clock cycle we need to fetch the instruction from main memory (t_{IFETCH}), read the operands from the register file (t_{RF}), perform the address addition in the ALU (t_{ALU}), read the requested location from main memory (t_{MEM}), and finally write the memory data to the destination register (t_{WB}).

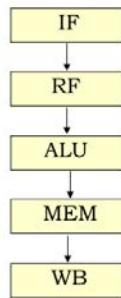
The component delays add up and the result is a fairly long clock period and hence it will take a long time to run the program. And our two example execution paths illustrate another issue: we're forced to choose the clock period to accommodate the worst-case execution time, even though we may be able to execute some instructions faster since their execution path through the circuitry is shorter. We're making all the instructions slower just because there's one instruction that has a long critical path.

So why not have simple instructions execute in one clock cycle and more complex instructions take multiple cycles instead of forcing all instructions to execute in a single, long clock cycle? As we'll see

in the next few slides, we have a good answer to this question, one that will allow us to execute *all* instructions with a short clock period.

Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce t_{CK}
 - Each instruction executes over multiple cycles
 - Consecutive instructions are overlapped to keep CPI ≈ 1.0
- We'll study the classic 5-stage pipeline:



Instruction Fetch stage: Maintains PC, fetches instruction and passes it to

Register File stage: Reads source operands from register file, passes them to

ALU stage: Performs indicated operation in ALU, passes result to

Memory stage: If it's a LD, use ALU result as an address, pass mem data (or ALU result if not LD) to

Write-Back stage: writes result back into register file.

$$t_{CLK} = \max\{t_{IFETCH}, t_{RF}, t_{ALU}, t_{MEM}, t_{WB}\}$$

We're going to use pipelining to address these issues. We're going to divide the execution of an instruction into a sequence of steps, where each step is performed in successive stages of the pipeline. So it will take multiple clock cycles to execute an instruction as it travels through the stages of the execution pipeline. But since there are only one or two components in each stage of the pipeline, the clock period can be much shorter and the throughput of the CPU can be much higher.

The increased throughput is the result of overlapping the execution of consecutive instructions. At any given time, there will be multiple instructions in the CPU, each at a different stage of its execution. The time to execute all the steps for a particular instruction (*i.e.*, the instruction latency) may be somewhat higher than in our unpipelined implementation. But we will finish the last step of executing some instruction in each clock cycle, so the instruction throughput is 1 per clock cycle. And since the clock cycle of our pipelined CPU is quite a bit shorter, the instruction throughput is quite a bit higher.

All this sounds great, but, not surprisingly, there are few issues we'll have to deal with.

There are many ways to pipeline the execution of an instruction. We're going to look at the design of the classic 5-stage instruction execution pipeline, which was widely used in the integrated circuit CPU designs of the 1980's.

The 5 pipeline stages correspond to the steps of executing an instruction in a von-Neumann stored-program architecture. The first stage (IF) is responsible for fetching the binary-encoded instruction from the main memory location indicated by the program counter.

The 32-bit instruction is passed to the register file stage (RF) where the required register operands are read from the register file.

The operand values are passed to the ALU stage (ALU), which performs the requested operation.

The memory stage (MEM) performs the second access to main memory to read or write the data for LD, LDR, or ST instructions, using the value from the ALU stage as the memory address. For load

instructions, the output of the MEM stage is the read data from main memory. For all other instructions, the output of the MEM stage is simply the value from the ALU stage.

In the final write-back stage (WB), the result from the earlier stages is written to the destination register in the register file.

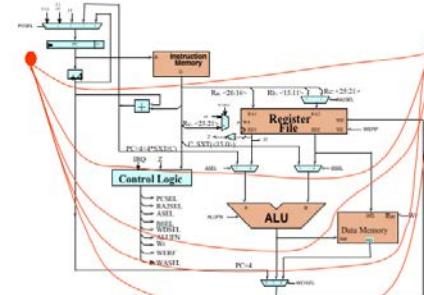
Looking at the execution path from the previous slide, we see that each of the main components of the unpipelined design is now in its own pipeline stage. So the clock period will now be determined by the slowest of these components.

Having divided instruction execution into five stages, would we expect the clock period to be one fifth of its original value? Well, that would only happen if we were able to divide the execution so that each stage performed exactly one fifth of the total work. In real life, the major components have somewhat different latencies, so the improvement in instruction throughput will be a little less than the factor of 5 a perfect 5-stage pipeline could achieve. If we have a slow component, *e.g.*, the ALU, we might choose to pipeline that component into further stages, or, interleave multiple ALUs to achieve the same effect. But for this lecture, we'll go with the 5-stage pipeline described above.

Why isn't this a 20-minute lecture?

We know how to pipeline
combinational circuits,
what's the big deal?

- Beta has state: PC, Register file, Memories
 - There are dependencies we cannot break!
 - To compute the next PC
 - To write result into the register file
 - We'll be addressing these issues as we examine the operation of our execution pipeline.



So why isn't this a 20-minute lecture? After all we know how pipeline combinational circuits: we can build a valid k -stage pipeline by drawing k contours across the circuit diagram and adding a pipeline register wherever a contour crosses a signal. What's the big deal here?

Well, is this circuit combinational? No! There's state in the registers and memories. This means that the result of executing a given instruction may depend on the results from earlier instructions. There are loops in the circuit where data from later pipeline stages affects the execution of earlier pipeline stages. For example, the write to the register file at the end of the WB stage will change values read from the register file in the RF stage. In other words, there are execution dependencies between instructions and these dependencies will need to be taken into account when we're trying to pipeline instruction execution. We'll be addressing these issues as we examine the operation of our execution pipeline.

Pipeline Hazards

- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
 - A data value → Data hazard
 - The program counter → Control hazard (branches, jumps, exceptions)
- Plan of attack:
 1. Design a 5-stage pipeline that works with sequences of independent instructions
 2. Handle data hazards
 3. Handle control hazards

Sometimes execution of a given instruction will depend on the results of executing a previous instruction. There are two types of problematic dependencies.

The first, termed a data hazard, occurs when the execution of the current instruction depends on data produced by an earlier instruction. For example, an instruction that reads R0 will depend on the execution of an earlier instruction that wrote R0.

The second, termed a control hazard, occurs when a branch, jump, or exception changes the order of execution. For example, the choice of which instruction to execute after a BNE depends on whether the branch is taken or not.

Instruction execution triggers a hazard when the instruction on which it depends is also in the pipeline, *i.e.*, the earlier instruction hasn't finished execution! We'll need to adjust execution in our pipeline to avoid these hazards.

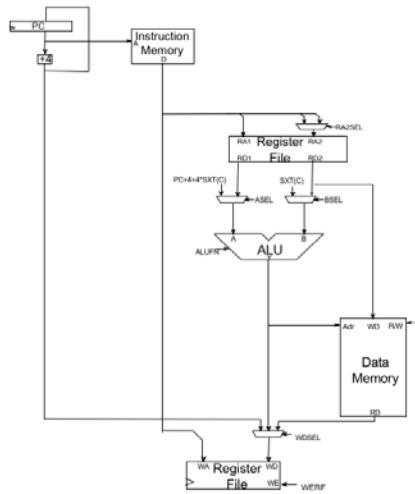
Here's our plan of attack:

We'll start by designing a 5-stage pipeline that works with sequences of instructions that don't trigger hazards, *i.e.*, where instruction execution doesn't depend on earlier instructions still in the pipeline.

Then we'll fix our pipeline to deal correctly with data hazards.

And finally, we'll address control hazards.

Simplified Unpipelined Beta Datapath

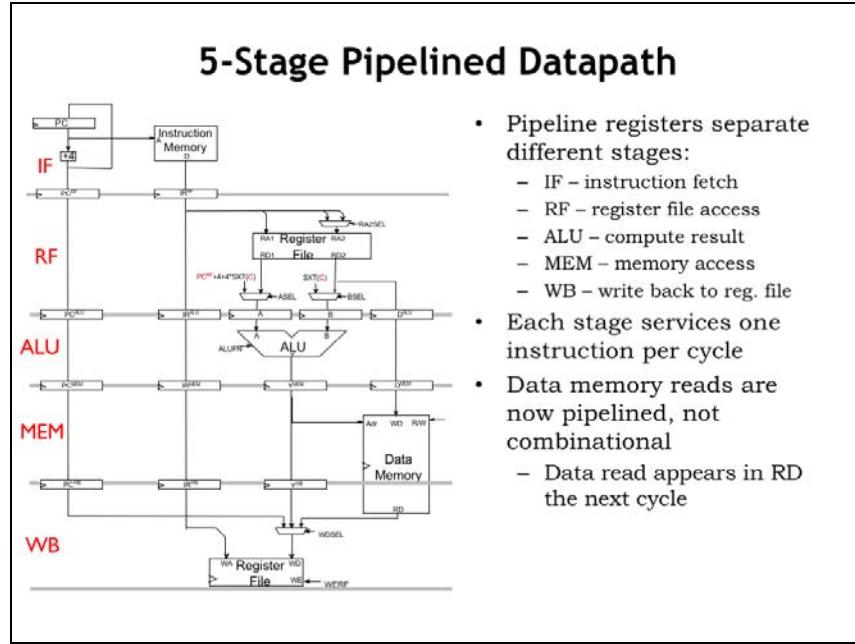


- NextPC = PC+4 (we'll worry about control hazards later)
- Same register file appears twice in the diagram
 - Top: reads
 - Bottom: writes

Let's start by redrawing and simplifying the Beta data path so that it will be easier to reason about when we add pipelining.

The first simplification is to focus on sequential execution and so leave out the branch addressing and PC MUX logic. Our simplified Beta always executes the next instruction from PC+4. We'll add back the branch and jump logic when we discuss control hazards.

The second simplification is to have the register file appear twice in the diagram so that we can tease apart the read and write operations that occur at different stages of instruction execution. The top Register File shows the combinational read ports, used to when reading the register operands in the RF stage. The bottom Register File shows the clocked write port, used to write the result into the destination register at the end of the WB stage. Physically, there's only one set of 32 registers, we've just drawn the read and write circuitry as separate components in the diagram.



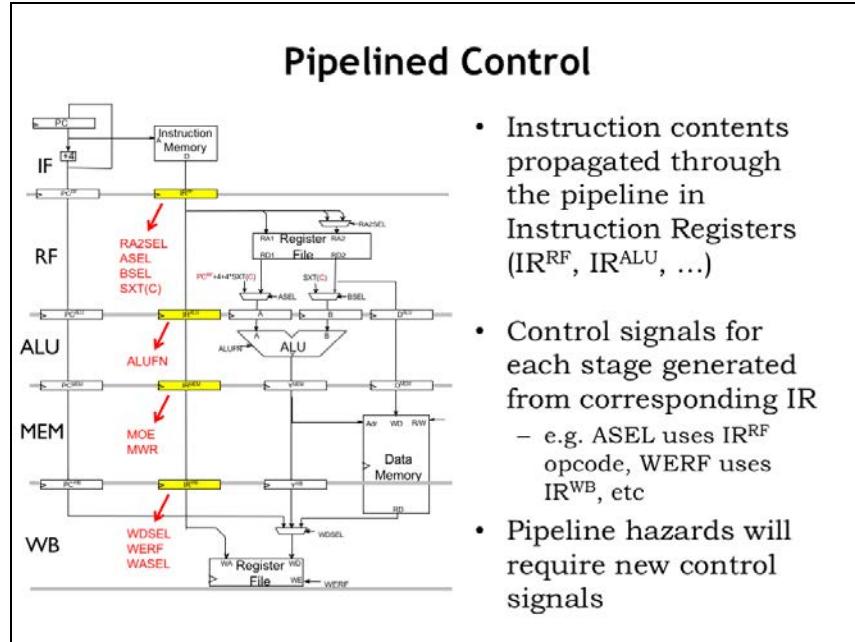
If we add pipeline registers to the simplified diagram, we see that execution proceeds through the five stages from top to bottom. If we consider execution of instruction sequences with no data hazards, information is flowing down the pipeline and the pipeline will correctly overlap the execution of all the instructions in the pipeline.

The diagram shows the components needed to implement each of the five stages. The IF stage contains the program counter and the main memory interface for fetching instructions. The RF stage has the register file and operand multiplexers. The ALU stage uses the operands and computes the result. The MEM stage handles the memory access for load and store operations. And the WB stage writes the result into the destination register.

In each clock cycle, each stage does its part in the execution of a particular instruction. In a given clock cycle, there are 5 instructions in the pipeline.

Note that data accesses to main memory span almost two clock cycles. Data accesses are initiated at the beginning of the MEM stage and returning data is only needed just before the end of the WB stage. The memory is itself pipelined and can simultaneously finish the access from an earlier instruction while starting an access for the next instruction.

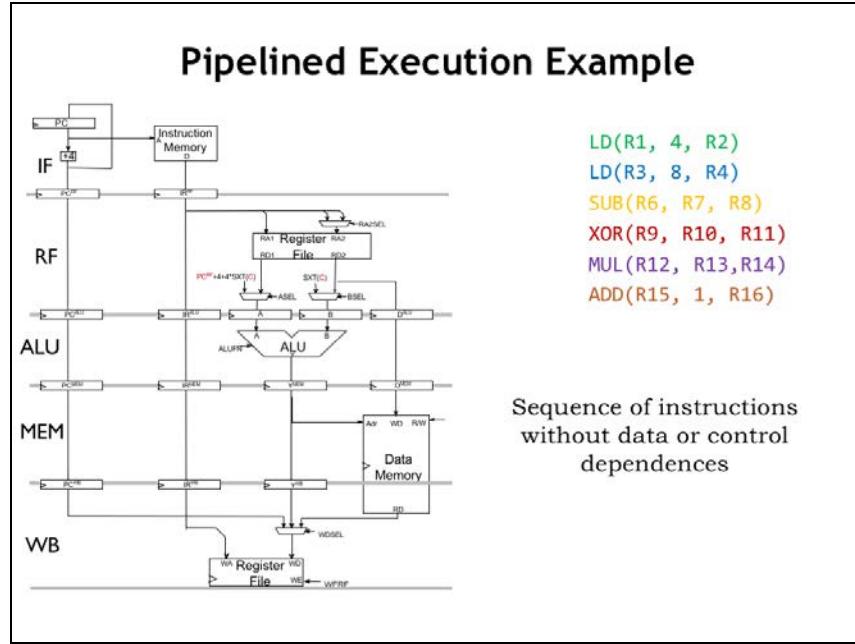
This simplified diagram isn't showing how the control logic is split across the pipeline stages. How does that work?



Note that we've included instruction registers as part of each pipeline stage, so that each stage can compute the control signals it needs from its instruction register. The encoded instruction is simply passed from one stage to the next as the instruction flows through the pipeline.

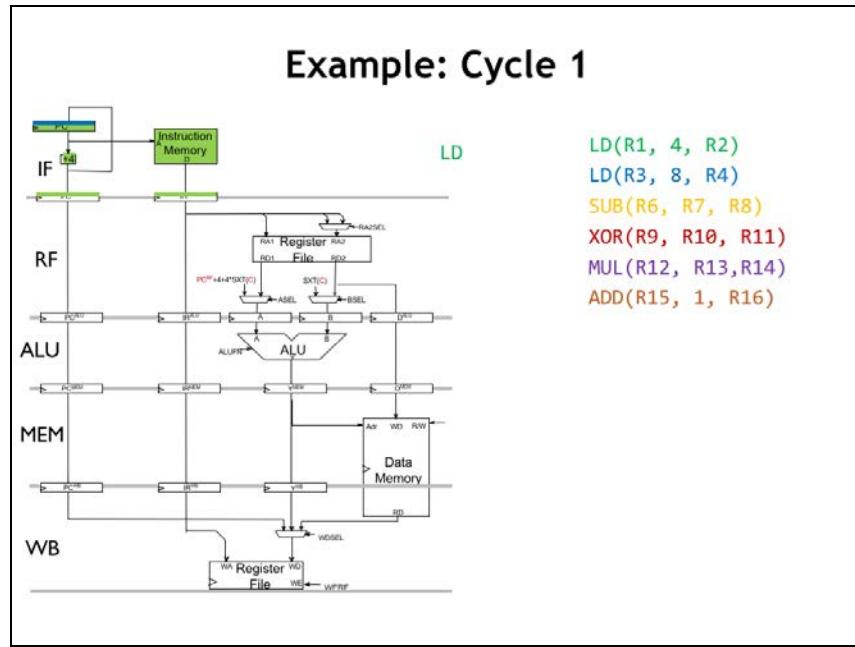
Each stage computes its control signals from the opcode field of its instruction register. The RF stage needs the RA, RB, and literal fields from its instruction register. And the WB stage needs the RC field from its instruction register. The required logic is very similar to the unpipelined implementation, it's just been split up and moved to the appropriate pipeline stage.

We'll see that we will have to add some additional control logic to deal correctly with pipeline hazards.



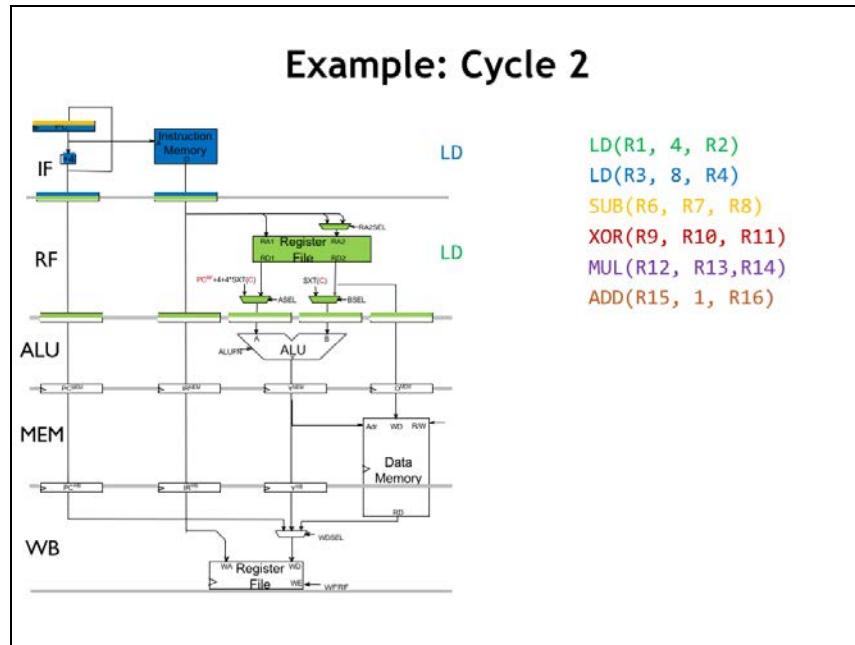
Our simplified diagram isn't so simple anymore! To see how the pipeline works, let's follow along as it executes this sequence of six instructions. Note that the instructions are reading and writing from different registers, so there are no potential data hazards. And there are no branches and jumps, so there are no potential control hazards. Since there are no potential hazards, the instruction executions can be overlapped and their overlapped execution in the pipeline will work correctly.

Okay, here we go!



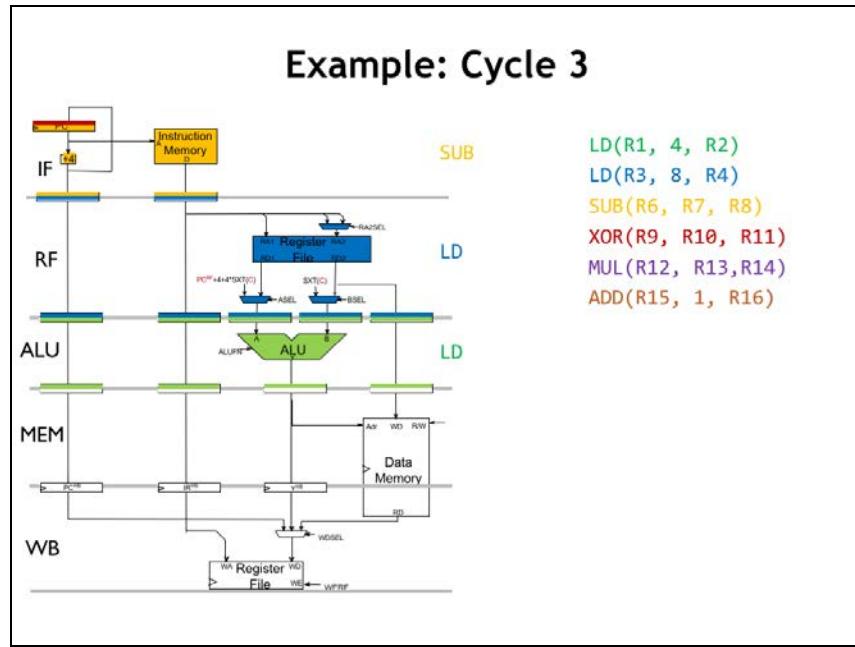
During cycle 1, the IF stage sends the value from the program counter to main memory to fetch the first instruction (the green LD instruction), which will be stored in the RF-stage instruction register at the end of the cycle. Meanwhile, it's also computing $PC + 4$, which will be the next value of the program counter. We've colored the next value blue to indicate that it's the address of the blue instruction in the sequence.

We'll add the appropriately colored label on the right of each pipeline stage to indicate which instruction the stage is processing.

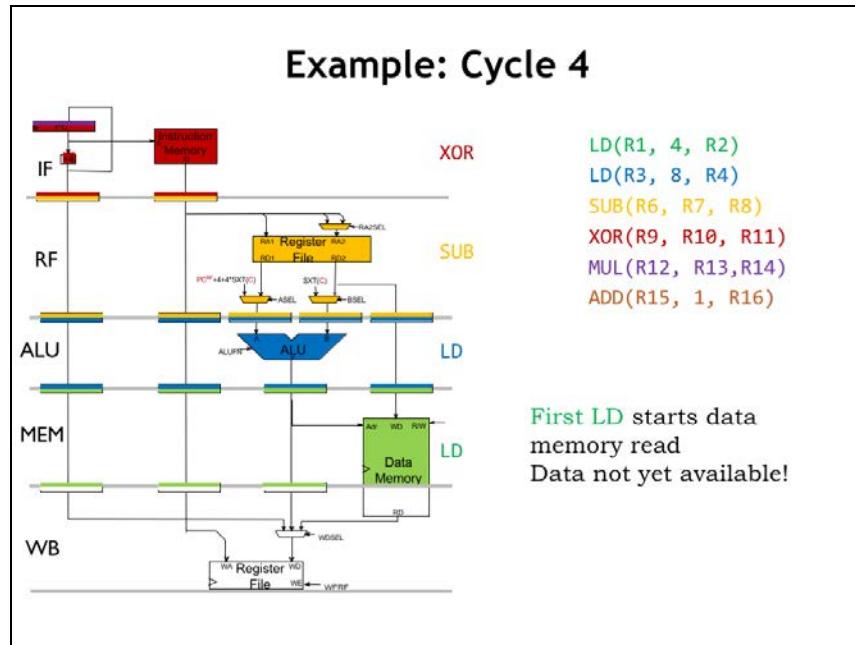


At the start of cycle 2, we see that values in the PC and instruction registers for the RF stage now correspond to the green instruction. During the cycle the register file will be reading the register operands, in this case R1, which is needed for the green instruction. Since the green instruction is a LD, ASEL is 0 and BSEL is 1, selecting the appropriate values to be written into the A and B operand registers at the end of the cycle.

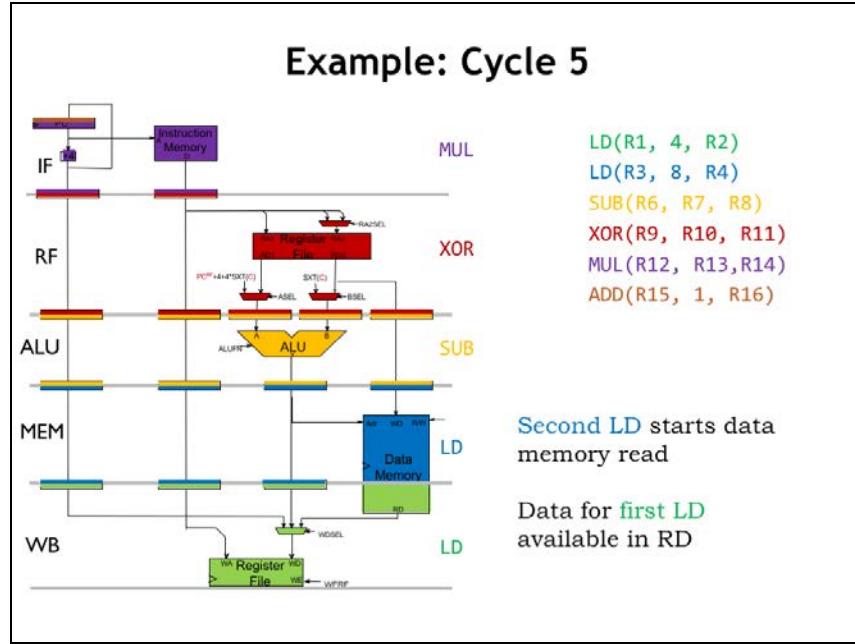
Concurrently, the IF stage is fetching the blue instruction from main memory and computing an updated PC value for the next cycle.



In cycle 3, the green instruction is now in the ALU stage, where the ALU is adding the values in its operand registers (in this case the value of R1 and the constant 4) and the result will be stored in Y_MEM register at the end of the cycle.



In cycle 4, we're overlapping the execution of four instructions. The MEM stage initiates a memory read for the green LD instruction. Note that the read data will first become available in the WB stage - it's not available to CPU in the current clock cycle.



In cycle 5, the results of the main memory read initiated in cycle 4 are available for writing to the register file in the WB stage. So execution of the green LD instruction will be complete when the memory data is written to R2 at the end of cycle 5.

Meanwhile, the MEM stage is initiating a memory read for the blue LD instruction.

The pipeline continues to complete successive instructions in successive clock cycles. The latency for a particular instruction is 5 clock cycles. The throughput of the pipelined CPU is 1 instruction/cycle. This is the same as the unpipelined implementation, except that the clock period is shorter because each pipeline stage has fewer components.

Note that the effects of the green LD, *i.e.*, filling R2 with a new value, don't happen until the rising edge of the clock at the end of cycle 5. In other words, the results of the green LD aren't available to other instructions until cycle 6. If there were instructions in the pipeline that read R2 before cycle 6, they would have gotten an old value! This is an example of a data hazard. Not a problem for us, since our instruction sequence didn't trigger this data hazard.

Tackling data hazards is our next task.

Pipeline Diagrams

- Represent pipeline utilization over time.
 - When do reads and writes happen?
 - Read REGFILE in RF stage
 - Write REGFILE at end of WB stage
- LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13, R14)
ADD(R15, 1, R16)

	Cycles →					
	1	2	3	4	5	6
IF	LD	LD	SUB	XOR	MUL	ADD
RF		LD	LD	SUB	XOR	MUL
ALU			LD	LD	SUB	XOR
MEM				LD	LD	SUB
WB					LD	LD

↓ Read R1 Write R2 ↑

The data path diagram isn't all that useful in diagramming the pipelined execution of an instruction sequence since we need a new copy of the diagram for each clock cycle. A more compact and easier-to-read diagram of pipelined execution is provided by the pipeline diagrams we met back in Part 1 of the course.

There's one row in the diagram for each pipeline stage and one column for each cycle of execution. Entries in the table show which instruction is in each pipeline stage at each cycle. In normal operation, a particular instruction moves diagonally through the diagram as it proceeds through the five pipeline stages.

To understand data hazards, let's first remind ourselves of when the register file is read and written for a particular instruction. Register reads happen when the instruction is in the RF stage, *i.e.*, when we're reading the instruction's register operands. Register writes happen at the end of the cycle when the instruction is in the WB stage. For example, for the first LD instruction, we read R1 during cycle 2 and write R2 at the end of cycle 5. Or consider the register file operations in cycle 6: we're reading R12 and R13 for the MUL instruction in the RF stage, and writing R4 at the end of the cycle for the LD instruction in the WB stage.

Data Hazards

- Consider this instruction sequence:

ADDC(R1, 1, R2)
 SUBC(R2, 4, R3)
 MUL(R6, R7, R8)
 XOR(R9, R10, R11)

	1	2	3	4	...	5	6
IF	ADDC	SUBC	MUL	XOR			
RF		ADDC	SUBC	MUL	XOR		
ALU			ADDC	SUBC	MUL	XOR	
MEM				ADDC	SUBC	MUL	
WB					ADDC	SUBC	

- SUBC reads R2 on cycle 3, but ADDC does not update it until end of cycle 5 → **R2 is stale!**
- Pipeline must maintain correct behavior...**

Okay, now let's see what happens when there are data hazards. In this instruction sequence, the ADDC instruction writes its result in R2, which is immediately read by the following SUBC instruction. Correct execution of the SUBC instruction clearly depends on the results of the ADDC instruction. This what we'd call a read-after-write dependency.

This pipeline diagram shows the cycle-by-cycle execution where we've circled the cycles during which ADDC writes R2 and SUBC reads R2.

Oops! ADDC doesn't write R2 until the end of cycle 5, but SUBC is trying to read the R2 value in cycle 3. The value in R2 in the register file in cycle 3 doesn't yet reflect the execution of the ADDC instruction. So as things stand the pipeline would **not** correctly execute this instruction sequence. This instruction sequence has triggered a data hazard.

We want the pipelined CPU to generate the same program results as the unpipelined CPU, so we'll need to figure out a fix.

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass (aka Forward). Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
 - Guess a value and continue executing anyway
 - When actual value is available, two cases
 - Guessed correctly → do nothing
 - Guessed incorrectly → kill & restart with correct value

There are three general strategies we can pursue to fix pipeline hazards. Any of the techniques will work, but as we'll see they have different tradeoffs for instruction throughput and circuit complexity.

The first strategy is to stall instructions in the RF stage until the result they need has been written to the register file. "Stall" means that we don't reload the instruction register at the end of the cycle, so we'll try to execute the same instruction in the next cycle. If we stall one pipeline stage, all earlier stages must also be stalled since they are blocked by the stalled instruction. If an instruction is stalled in the RF stage, the IF stage is also stalled. Stalling will always work, but has a negative impact on instruction throughput. Stall for too many cycles and you'll lose the performance advantages of pipelined execution!

The second strategy is to route the needed value to earlier pipeline stages as soon as its computed. This called bypassing or forwarding. As it turns out, the value we need often exists somewhere in the pipelined data path, it just hasn't been written yet to the register file. If the value exists and can be forwarded to where it's needed, we won't need to stall. We'll be able to use this strategy to avoid stalling on most types of data hazards.

The third strategy is called speculation. We'll make an intelligent guess for the needed value and continue execution. Once the actual value is determined, if we guessed correctly, we're all set. If we guessed incorrectly, we have to back up execution and restart with the correct value. Obviously speculation only makes sense if it's possible to make accurate guesses. We'll be able to use this strategy to avoid stalling on control hazards.

Let's see how the first two strategies work when dealing with our data hazard.

Resolving Data Hazards (1)

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

ADDC(R1, 1, R2)
 SUBC(R2, 4, R3)
 MUL(R6, R7, R8)
 XOR(R9, R10, R11)
 ...

	1	2	3	4	5	6	7	8
IF	ADDC	SUBC	MUL	MUL	MUL	MUL	XOR	
RF		ADDC	SUBC	SUBC	SUBC	SUBC	MUL	XOR
ALU			ADDC	NOP	NOP	NOP	SUBC	MUL
MEM				ADDC	NOP	NOP	NOP	SUBC
WB					ADDC	NOP	NOP	NOP

↑ R2 updated

Stalls increase CPI!

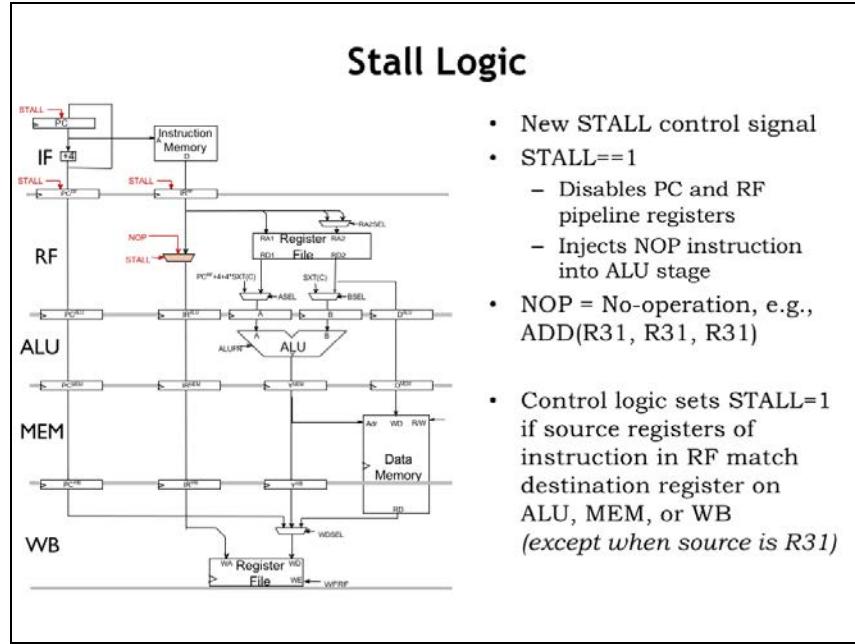
Applying the stall strategy to our data hazard, we need to stall the SUBC instruction in the RF stage until the ADDC instruction writes its result in R2. So in the pipeline diagram, SUBC is stalled three times in the RF stage until it can finally access the R2 value from the register file in cycle 6.

Whenever the RF stage is stalled, the IF stage is also stalled. You can see that in the diagram too. But when RF is stalled, what should the ALU stage do in the next cycle? The RF stage hasn't finished its job and so can't pass along its instruction! The solution is for the RF stage to make-up an innocuous instruction for the ALU stage, what's called a NOP instruction, short for "no operation". A NOP instruction has no effect on the CPU state, *i.e.*, it doesn't change the contents of the register file or main memory. For example any OP-class or OPC-class instruction that has R31 as its destination register is a NOP.

The NOPs introduced into the pipeline by the stalled RF stage are shown in red. Since the SUBC is stalled in the RF stage for three cycles, three NOPs are introduced into the pipeline. We sometimes refer to these NOPs as "bubbles" in the pipeline.

How does the pipeline know when to stall? It can compare the register numbers in the RA and RB fields of the instruction in the RF stage with the register numbers in the RC field of instructions in the ALU, MEM, and WB stage. If there's a match, there's a data hazard and the RF stage should be stalled. The stall will continue until there's no hazard detected. There are a few details to take care of: some instructions don't read both registers, the ST instruction doesn't use its RC field, and we don't want R31 to match since it's always okay to read R31 from the register file.

Stalling will ensure correct pipelined execution, but it does increase the effective CPI. This will lead to longer execution times if the increase in CPI is larger than the decrease in cycle time afforded by pipelining.



To implement stalling, we only need to make two simple changes to our pipelined data path. We generate a new control signal, STALL, which, when asserted, disables the loading of the three pipeline registers at the input of the IF and RF stages, which means they'll have the same value next cycle as they do this cycle. We also introduce a mux to choose the instruction to be sent along to the ALU stage. If STALL is 1, we choose a NOP instruction, e.g., an ADD with R31 as its destination. If STALL is 0, the RF stage is not stalled, so we pass its current instruction to the ALU.

And here we see how to compute STALL as described in the previous slide.

The additional logic needed to implement stalling is pretty modest, so the real design tradeoff is about increased CPI due to stalling vs. decreased cycle time due to pipelining.

So we have a solution, although it carries some potential performance costs.

Resolving Data Hazards (2)

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

ADDC(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
...
- ADDC writes to R2 at the end of cycle 5...
but the result is available at the end of the ALU stage!

	1	2	3	4	5
IF	ADDC	SUBC	MUL	XOR	
RF		ADDC	SUBC	MUL	XOR
ALU			ADDC	SUBC	MUL
MEM				ADDC	SUBC
WB					ADDC

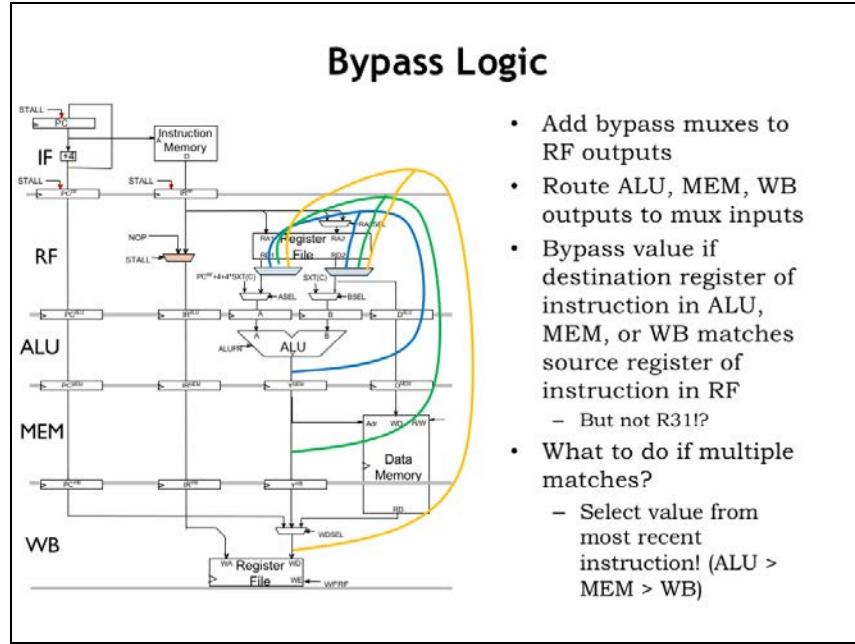
ADDC result computed ↑ R2 updated ↘

Now let's consider our second strategy: bypassing, which is applicable if the data we need in the RF stage is somewhere in the pipelined data path.

In our example, even though ADDC doesn't write R2 until the end of cycle 5, the value that will be written is computed during cycle 3 when the ADDC is in the ALU stage. In cycle 3, the output of the ALU is the value needed by the SUBC that's in the RF stage in the same cycle.

So, if we detect that the RA field of the instruction in the RF stage is the same as the RC field of the instruction in the ALU stage, we can use the output of the ALU in place of the (stale) RA value being read from the register file. No stalling necessary!

In our example, in cycle 3 we want to route the output of the ALU to the RF stage to be used as the value for R2. We show this with a red "bypass arrow" showing data being routed from the ALU stage to the RF stage.

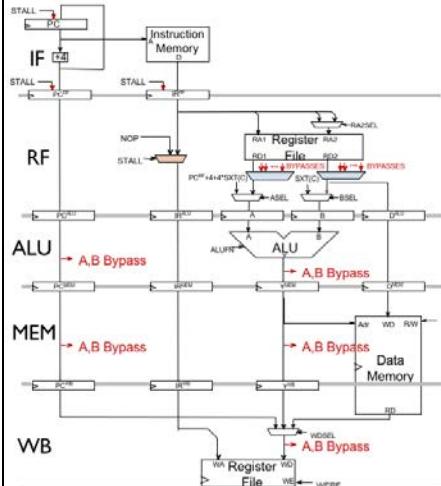


To implement bypassing, we'll add a many-input multiplexer to the read ports of the register file so we can select the appropriate value from other pipeline stages. Here we show the combinational bypass paths from the ALU, MEM, and WB stages. For the bypassing example of previous slides, we use the blue bypass path during cycle 3 to get the correct value for R2.

The bypass MUXes are controlled by logic that's matching the number of the source register to the number of the destination registers in the ALU, MEM, and WB stages, with the usual complications of dealing with R31.

What if there are multiple matches, *i.e.*, if the RF stage is trying to read a register that's the destination for, say, the instructions in both the ALU and MEM stages? No problem! We want to select the result from the most recent instruction, so we'd chose the ALU match if there is one, then the MEM match, then the WB match, then, finally, the output of the register file.

Fully Bypassed Pipeline



- Some instructions write PC+4...
- Route PC^{ALU} and PC^{MEM} as additional bypass mux inputs
- Bypasses are expensive
 - Lots of wiring & large muxes
 - May affect clock cycle time...
- But full bypassing is not needed! We can always stall
 - e.g., just bypass from ALU
- With a fully bypassed pipeline, do we still need the stall signal?

Here's a diagram showing all the bypass paths we'll need. Note that branches and jumps write their PC+4 value into the register file, so we'll need to bypass from the PC+4 values in the various stages as well as the ALU values.

Note that the bypassing is happening at the end of the cycle, *e.g.*, after the ALU has computed its answer. To accommodate the extra t_{PD} of the bypass MUX, we'll have to extend the clock period by a small amount. So once again there's a design tradeoff - the increased CPI of stalling vs the slightly increased cycle time of bypassing. And, of course, in the case of bypassing there's the extra area needed for the necessary wiring and MUXes.

We can cut back on the costs by reducing the amount of bypassing, say, to only bypassing ALU results from the ALU stage and use stalling to deal with all the other data hazards.

If we implement full bypassing, do we still need the STALL logic?

Load-To-Use Stalls

- Bypassing cannot eliminate load delays because data is not available until the WB stage!
- Bypassing from WB still saves a cycle:

		LD(R1, 1, R2)	SUBC(R2, 4, R3)	MUL(R6, R7, R8)	XOR(R9, R10, R11)	...	
IF	LD	SUBC	MUL	MUL	MUL	XOR	
RF		LD	SUBC	SUBC	SUBC	MUL	XOR
ALU			LD	NOP	NOP	SUBC	MUL
MEM				LD	NOP	NOP	SUBC
WB					LD	NOP	NOP

LD data available ↑ R2 updated ↘

As it turns out, we do! There's one data hazard that bypassing doesn't completely address. Consider trying to immediately use the result of a LD instruction. In the example shown here, the SUBC is trying to use the value the immediately preceding LD is writing to R2. This is called a load-to-use hazard.

Recalling that LD data isn't available in the data path until the cycle when LD reaches the WB stage, even with full bypassing we'll need to stall SUBC in the RF stage until cycle 5, introducing two NOPs into the pipeline. Without bypassing from the WB stage, we need to stall until cycle 6.

Summary: Pipelining with Data Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
 - Simple, wastes cycles, higher CPI
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
 - More expensive, lower CPI
 - Still needs stalls when result is produced after ALU stage
 - Can use fewer bypasses & stall more often
- More pipeline stages → More frequent data hazards
 - Lower t_{CK} , but higher CPI

In summary, we have two strategies for dealing with data hazards.

We can stall the IF and RF stages until the register values needed by the instruction in the RF stage are available in the register file. The required hardware is simple, but the NOPs introduced into the pipeline waste CPU cycles and result in an higher effective CPI.

Or we can use bypass paths to route the required values to the RF stage assuming they exist somewhere in the pipelined data path. This approach requires more hardware than stalling, but doesn't reduce the effective CPI. Even if we implement bypassing, we'll need still need stalls to deal with load-to-use hazards.

Can we keep adding pipeline stages in the hopes of further reducing the clock period? More pipeline stages mean more instructions in the pipeline at the same time, which in turn increases the chance of a data hazard and the necessity of stalling, thus increasing CPI.

Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:



- Only works well when compiler can find independent instructions to move around!

Compilers can help reduce dependencies by reorganizing the assembly language code they produce. Here's the load-to-use hazard example we saw earlier. Even with full bypassing, we'd need to stall for 2 cycles.

But if the compiler (or assembly language programmer!) notices that the MUL and XOR instructions are independent of the SUBC instruction and hence can be moved before the SUBC, the dependency is now such that the LD is naturally in the WB stage when the ST is in the RF stage, so no stalls are needed.

This optimization only works when the compiler can find independent instructions to move around. Unfortunately there are plenty of programs where such instructions are hard to find.

Or take the lazy route...

- Don't stall or bypass, just change the ISA so that registers are updated with a 3-instruction delay!
 - Compiler writers will love this!
 - Programmers will love this!
 - You will love this when you decide to release an 8-stage pipelined processor!

I'm altering the ISA.
Pray I do not alter
it further...



Roger Schultz
(CC-BY 2.0)

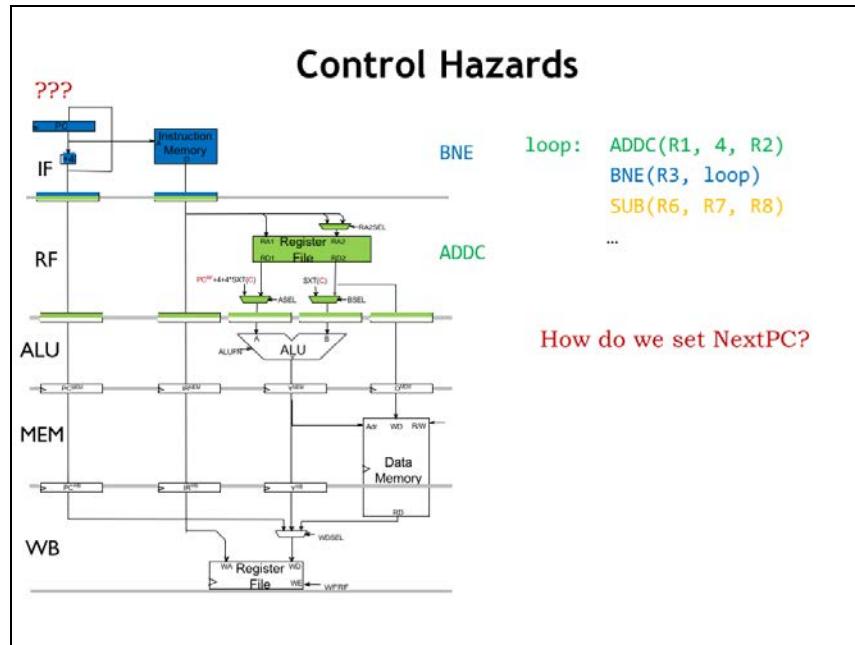
- ISAs outlive implementations, this is a bad idea

Then there's one final approach we could take - change the ISA so that data hazards are part of the ISA, *i.e.*, just explain that writes to the destination register happen with a 3-instruction delay! If NOPs are needed, make the programmer add them to the program. Simplify the hardware at the “small” cost of making the compilers work harder.

You can imagine exactly how much the compiler writers will like this suggestion. Not to mention assembly language programmers! And you can change the ISA again when you add more pipeline stages!

This is how a compiler writer views CPU architects who unilaterally change the ISA to save a few logic gates :)

The bottom line is that successful ISAs have very long lifetimes and so shouldn't include tradeoffs driven by short-term implementation considerations. Best not to go there.



Now let's turn our attention to control hazards, illustrated by the code fragment shown here. Which instruction should be executed after the BNE? If the value in R3 is non-zero, ADDC should be executed. If the value in R3 is zero, the next instruction should be SUB. If the current instruction is an explicit transfer of control (*i.e.*, JMPs or branches), the choice of the next instruction depends on the execution of the current instruction.

What are the implications of this dependency on our execution pipeline?

Control Hazards

- What do we need to compute NextPC?

```

- BEQ/BNE:   BEQ(Ra, label, Rc):
              Reg[Rc] ← PC + 4
              if (Reg[Ra] == 0)
                  PC ← PC + 4 + 4*SXT(offset)
              else
                  PC ← PC + 4
      Opcode, offset, PC+4, Reg[Ra] ←

- JMP:        JMP(Ra, Rc):
              Reg[Rc] ← PC + 4
              PC ← Reg[Ra]
      Opcode, Reg[Ra] ←

- All other instructions: Opcode, PC+4
- (Exceptions also change PC, we'll deal with them later)

```

Opcode, offset, PC+4, Reg[Ra] ←
Unknown until RF stage...

How does the unpipelined implementation determine the next instruction?

For branches (BEQ or BNE), the value to be loaded into the program counter depends on (1) the opcode, *i.e.*, whether the instruction is a BEQ or a BNE, (2) the current value of the program counter since that's used in the offset calculation, and (3) the value stored in the register specified by the RA field of the instruction since that's the value tested by the branch.

For JMP instructions, the next value of the program counter depends once again on the opcode field and the value of the RA register.

For all other instructions, the next PC value depends only the opcode of the instruction and the value PC+4.

Exceptions also change the program counter. We'll deal with them later in the lecture.

The control hazard is triggered by JMP and branches since their execution depends on the value in the RA register, *i.e.*, they need to read from the register file, which happens in the RF pipeline stage. Our bypass mechanisms ensure that we'll use the correct value for the RA register even if it's not yet been written into the register file. What we're concerned about here is that the address of the instruction following the JMP or branch will be loaded into program counter at the end of the cycle when the JMP or branch is in the RF stage. But what should the IF stage be doing while all this is going on in RF stage?

Resolving Control Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
 - Guess a value and continue executing anyway
 - When actual value is available, two cases
 - Guessed correctly → do nothing
 - Guessed incorrectly → kill & restart with correct value

The answer is that in the case of JMPs and taken branches, we don't know what the IF stage should be doing until those instructions are able to access the value of the RA register in the RF stage.

One solution is to stall the IF stage until the RF stage can compute the necessary result. This was the first of our general strategies for dealing with hazards. How would this work?

Resolving Control Hazards With Stalls

- If branch or jump in IF, stall IF for one cycle
- Assume BNE is always taken in example code

	I	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP	ADDC
RF		ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

↑ R3!=0 → Taken ↑ R3!=0

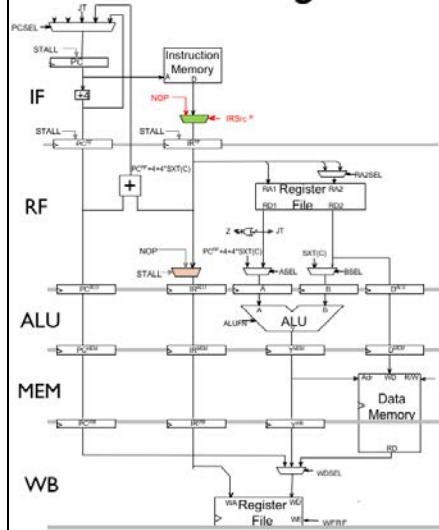
- Steady-state CPI?

If the opcode in the RF stage is JMP, BEQ, or BNE, stall the IF stage for one cycle. In the example code shown here, assume that the value in R3 is non-zero when the BNE is executed, *i.e.*, that the instruction following BNE should be the ADDC at the top of the loop.

The pipeline diagram shows the effect we're trying to achieve: a NOP is inserted into the pipeline in cycles 4 and 8. Then execution resumes in the next cycle after the RF stage determines what instruction comes next. Note, by the way, that we're relying on our bypass logic to deliver the correct value for R3 from the MEM stage since the ADDC instruction that wrote into R3 is still in the pipeline, *i.e.*, we have a data hazard to deal with too!

Looking at, say, the WB stage in the pipeline diagram, we see it takes 4 cycles to execute one iteration of our 3-instruction loop. So the effective CPI is 4/3, an increase of 33%. Using stall to deal with control hazards has had an impact on the instruction throughput of our execution pipeline.

Stall Logic For Control Hazards



- IRSrc^{IF} control signal
- If $\text{opcode}^{\text{RF}} == \text{JMP}$, BEQ , BNE
 - IRSrc^{IF}=1, inject NOP
 - Set PCSEL to load branch or jump target

We've already seen the logic needed to introduce NOPs into the pipeline. In this case, we add a mux to the instruction path in the IF stage, controlled by the IRSrc_{IF} select signal. We use the superscript on the control signals to indicate which pipeline stage holds the logic they control.

If the opcode in the RF stage is JMP, BEQ, or BNE we set IRSrc_{IF} to 1, which causes a NOP to replace the instruction that was being read from main memory. And, of course, we'll be setting the PCSEL control signals to select the correct next PC value, so the IF stage will fetch the desired follow-on instruction in the next cycle.

If we replace an instruction with NOP, we say we “annulled” the instruction.

ISA Issue: Simple vs Complex Branches

- Beta has very simple branch condition
 - Reg[RA]==0 easily computed in RF
- Other ISAs have more complex branches (e.g., branch if greater than) that are resolved in ALU
- What if branches were resolved in ALU stage?

	1	2	3	4	5	6	7	8
IF	ADDC	MUL	BNE	SUB	NOP	ADDC	MUL	BNE
RF		ADDC	MUL	BNE	NOP	NOP	ADDC	MUL
ALU			ADDC	MUL	BNE	NOP	NOP	ADDC
MEM				ADDC	MUL	BNE	NOP	NOP
WB					ADDC	MUL	BNE	NOP

More annulments (but sometimes fewer instructions)

The branch instructions in the Beta ISA make their branch decision in the RF stage since they only need the value in register RA.

But suppose the ISA had a branch where the branch decision was made in ALU stage.

When the branch decision is made in the ALU stage, we need to introduce two NOPs into the pipeline, replacing the now unwanted instructions in the RF and IF stages. This would increase the effective CPI even further. But the tradeoff is that the more complex branches may reduce the number of instructions in the program.

If we annul instructions in all the earlier pipeline stages, this is called “flushing the pipeline”. Since flushing the pipeline has a big impact on the effective CPI, we do it when it’s the only way to ensure the correct behavior of the execution pipeline.

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
 - Guess a value and continue executing anyway
 - When actual value is available, two cases
 - Guessed correctly → do nothing
 - Guessed incorrectly → annul & restart with correct value

We can be smarter about when we choose to flush the pipeline when executing branches. If the branch is not taken, it turns out that the pipeline has been doing the right thing by fetching the instruction following the branch. Starting execution of an instruction even when we're unsure whether we really want it executed is called “speculation”.

Speculative execution is okay if we're able to annul the instruction before it has an effect on the CPU state, *e.g.*, by writing into the register file or main memory. Since these state changes (called “side effects”) happen in the later pipeline stages, an instruction can progress through the IF, RF, and ALU stages before we have to make a final decision about whether it should be annulled.

Resolving Hazards with Speculation

- What's a good guess for NextPC? **PC+4**
- Assume BNE is not taken in example

loop: ADDC(R1, -1, R3)
 MUL(R4, R5, R6)
 BNE(R3, loop)
 SUB(R6, R7, R8)
 XOR(R9, R10, R11)
 ...

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	XOR				
RF		ADDC	MUL	BNE	SUB	XOR			
ALU			ADDC	MUL	BNE	SUB	XOR		
MEM				ADDC	MUL	BNE	SUB	XOR	
WB					ADDC	MUL	BNE	SUB	XOR

Start fetching at PC+4 (SUB) but ↑
 BNE not resolved yet... Guessed right, keep going

How does speculation help with control hazards? Guessing that the next value of the program counter is PC+4 is correct for all but JMPs and taken branches.

Here's our example again, but this time let's assume that the BNE is not taken, *i.e.*, that the value in R3 is zero. The SUB instruction enters the pipeline at the start of cycle 4. At the end of cycle 4, we know whether or not to annul the SUB. If the branch is not taken, we want to execute the SUB instruction, so we just let it continue down the pipeline.

In other words, instead of always annulling the instruction following branch, we only annull it if the branch was taken. If the branch is not taken, the pipeline has speculated correctly and no instructions need to be annullled.

Resolving Hazards with Speculation

- What's a good guess for NextPC? **PC+4**
- Assume BNE is **taken** in example

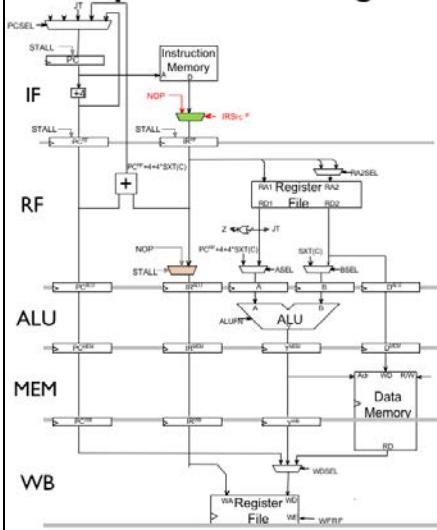
loop: ADDC(R1, -1, R3)
 MUL(R4, R5, R6)
 BNE(R3, loop)
 SUB(R6, R7, R8)
 XOR(R9, R10, R11)
 ...

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	ADDC	MUL	BNE	SUB	ADDC
RF		ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

Start fetching at PC+4 (SUB) but ↑
 BNE not resolved yet... Guessed wrong, annul SUB

However if the BNE is taken, the SUB is annulled at the end of cycle 4 and a NOP is executed in cycle 5. So we only introduce a bubble in the pipeline when there's a taken branch. Fewer bubbles will decrease the impact of annulment on the effective CPI.

Speculation Logic For Control Hazards



- This looks familiar...
- IRSrc^{IF} control signal
- If opcode^{RF} == JMP or taken BEQ/BNE
 - IRSrc^{IF}=1, inject NOP to annul fetched inst. (aka “branch annulment”)
 - Set PCSEL to load branch or jump target

We'll be using the same data path circuitry as before, we'll just be a bit more clever about when the value of the IRSrc_{IF} control signal is set to 1. Instead of setting it to 1 for all branches, we only set it to 1 when the branch is taken.

Branch Prediction

- Always guessing PC+4 wastes a cycle on taken branches and jumps
 - ~10% higher CPI
- With deeper pipelines, taken branches waste many more cycles
 - E.g., Intel Nehalem takes about 17 cycles to resolve whether a branch is taken
- Modern CPUs dynamically predict the outcome of control-flow instructions
 - Predict both the branch condition and the target
 - Works well because branches have repeated behavior
 - E.g. branches for loops are usually taken
 - E.g. termination/limit/error tests are usually not taken

Our naive strategy of always speculating that the next instruction comes from PC+4 is wrong for JMPs and taken branches. Looking at simulated execution traces, we'll see that this error in speculation leads to about 10% higher effective CPI. Can we do better?

This is an important question for CPUs with deep pipelines. For example, Intel's Nehalem processor from 2009 resolves the more complex x86 branch instructions quite late in the pipeline. Since Nehalem is capable of executing multiple instructions each cycle, flushing the pipeline in Nehalem actually annuls the execution of many instructions, resulting in a considerable hit on the CPI.

Like many modern processor implementations, Nehalem has a much more sophisticated speculation mechanism. Rather than always guessing the next instruction is at PC+4, it only does that for non-branch instructions. For branches, it predicts the behavior of each individual branch based on what the branch did last time it was executed and some knowledge of how the branch is being used. For example, backward branches at the end of loops, which are taken for all but the final iteration of the loop, can be identified by their negative branch offset values. Nehalem can even determine if there's correlation between branch instructions, using the results of an another, earlier branch to speculate on the branch decision of the current branch. With these sophisticated strategies, Nehalem's speculation is correct 95% to 99% of the time, greatly reducing the impact of branches on the effective CPI.

Branch Delay Slots

- Change the ISA so that the instruction following a jump or branch is always executed

loop: ADDC(R1, -1, R3)
BNE(R3, loop)

Delay slot instruction executes
regardless of branch outcome

```

MUL(R4, R5, R6)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
...

```

	1	2	3	4	5	6	7	8
IF	ADDC	BNE	MUL	ADDC	BNE	MUL	ADDC	BNE
RF		ADDC	BNE	MUL	ADDC	BNE	MUL	ADDC
ALU			ADDC	BNE	MUL	ADDC	BNE	MUL
MEM				ADDC	BNE	MUL	ADDC	BNE
WB					ADDC	BNE	MUL	ADDC

There's also the lazy option of changing the ISA to deal with control hazards. For example, we could change the ISA to specify that the instruction following a jump or branch is always executed. In other words the transfer of control happens **after** the next instruction. This change ensures that the guess of PC+4 as the address of the next instruction is always correct!

In the example shown here, assuming we changed the ISA, we can reorganize the execution order of the loop to place the MUL instruction after the BNE instruction, in the so-called "branch delay slot". Since the instruction in the branch delay slot is always executed, the MUL instruction will be executed during each iteration of the loop.

The resulting execution is shown in this pipeline diagram. Assuming we can find an appropriate instruction to place in the branch delay slot, the branch will have zero impact on the effective CPI.

Branch Delay Slots

- **Pro:** If compiler can fill slot with useful instruction, no branch/jump penalty
- **Cons:**
 - Can't fill slot with useful work ~50% of the time → Must insert NOP, longer code
 - Longer pipeline → More delay slots?
 - Branch prediction works better in practice

I'm altering the ISA.
Pray I do not alter
it further...



Roger Schultz
(CC-BY 2.0)

- **ISAs outlive implementations, this is a bad idea**

Are branch delay slots a good idea? Seems like they reduce the negative impact that branches might have on instruction throughput.

The downside is that only half the time can we find instructions to move to the branch delay slot. The other half of the time we have to fill it with an explicit NOP instruction, increasing the size of the code. And if we make the branch decision later in the pipeline, there are more branch delay slots, which would be even harder to fill. In practice, it turns out that branch prediction works better than delay slots in reducing the impact of branches.

So, once again we see that it's problematic to alter the ISA to improve the throughput of pipelined execution. ISAs outlive implementations, so it's best not to change the execution semantics to deal with performance issues created by a particular implementation.

Exceptions

- On an exception, need to:
 - Save current PC+4 in XP (R30)
 - Load PC with exception vector (IlliOp or XAdr)
- Exceptions cause control flow hazards!
 - They are **implicit branches**
- Want **precise exceptions**:
 - All preceding instructions must have completed
 - Instruction causing exception and future instructions must not have executed
 - No updates to register or memory
 - Simple in single-cycle machines, more complex with pipelining

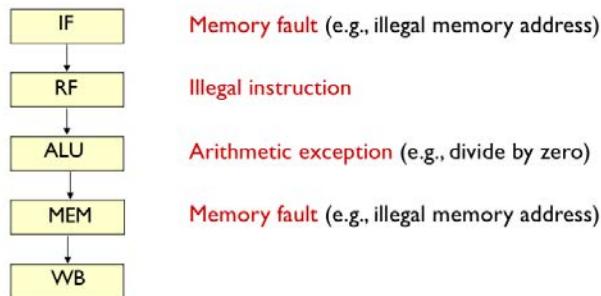
Now let's figure out how exceptions impact pipelined execution. When an exception occurs because of an illegal instruction or an external interrupt, we need to store the current PC+4 value in the XP register and load the program counter with the address of the appropriate exception handler.

Exceptions cause control flow hazards since they are effectively implicit branches.

In an unpipelined implementation, exceptions affect the execution of the current instruction. We want to achieve exactly the same effect in our pipelined implementation. So first we have to identify which one of the instructions in our pipeline is affected, then ensure that instructions that came earlier in the code complete correctly and that we annul the affected instruction and any following instructions that are in the pipeline.

Since there are multiple instructions in the pipeline, we have a bit of sorting out to do.

When Can Exceptions Happen?



- Instructions following the one that causes the exception may already be in the pipeline...
- ... but none has written registers or memory yet ☺

When, during pipelined execution, do we determine that an instruction will cause an exception? An obvious example is detecting an illegal opcode when we decode the instruction in the RF stage. But we can also generate exceptions in other pipeline stages. For example, the ALU stage can generate an exception if the second operand of a DIV instruction is 0. Or the MEM stage may detect that the instruction is attempting to access memory with an illegal address. Similarly the IF stage can generate a memory exception when fetching the next instruction.

In each case, instructions that follow the one that caused the exception may already be in the pipeline and will need to be annulled. The good news is that since register values are only updated in the WB stage, annulling an instruction only requires replacing it with a NOP. We won't have to restore any changed values in the register file or main memory.

Resolving Exceptions

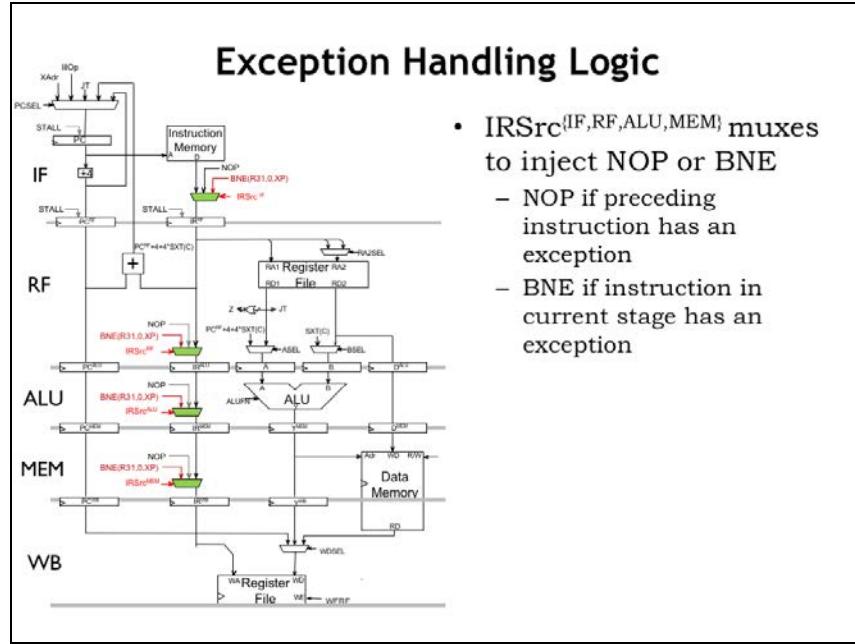
- If an instruction has an exception at stage i
 - Turn that instruction into BNE(R31, 0, XP) to save PC+4
 - Annul instructions in stages $i-1, \dots, 1$ (flush the pipeline)
 - Set PC \leftarrow IllOp or XAdr
- Example: LD has memory fault

LD(R1, 4, R2) XAdr: ADDC
 ST(R3, 0, R4) ST
 MUL(R4, R5, R6)
 SUB(R7, R8, R9)

	I	2	3	4	5	6
IF	LD	ST	MUL	SUB	ADDC	ST
RF		LD	ST	MUL	NOP	ADDC
ALU			LD	ST	NOP	NOP
MEM				LD	NOP	NOP
WB					BNE	NOP

Here's our plan. If an instruction causes an exception in stage i, replace that instruction with this BNE instruction, whose only side effect is writing the PC+4 value into the XP register. Then flush the pipeline by annulling instructions in earlier pipeline stages. And, finally, load the program counter with the address of the exception handler.

In this example, assume that LD will generate a memory exception in the MEM stage, which occurs in cycle 4. The arrows show how the instructions in the pipeline are rewritten for cycle 5, at which point the IF stage is working on fetching the first instruction in the exception handler.



Here are the changes required to the execution pipeline. We modify the muxes in the instruction path so that they can replace an actual instruction with either NOP if the instruction is to be annulled, or BNE if the instruction caused the exception.

Multiple Exceptions?

Causes memory fault

	→ LD(R1, 4, R2)	Xadr: ADDC	IllOp: XORC
	→ ???	ST	SUBC
Invalid opcode	MUL(R4, R5, R6)
	SUB(R7, R8, R9)		

	I	2	3	4	5	6
IF	LD	???	MUL	XORC	ADDC	ST
RF		LD	???	NOP	NOP	ADDC
ALU			LD	BNE	NOP	NOP
MEM				LD	NOP	NOP
WB				BNE	NOP	

↑ ↑
Invalid opcode detected Memory fault detected

Works fine even if exception from latter instruction is detected first!

Since the pipeline is executing multiple instructions at the same time, we have to worry about what happens if multiple exceptions are detected during execution. In this example assume that LD will cause a memory exception in the MEM stage and note that it is followed by an instruction with an illegal opcode.

Looking at the pipeline diagram, the invalid opcode is detected in the RF stage during cycle 3, causing the illegal instruction exception process to begin in cycle 4. But during that cycle, the MEM stage detects the illegal memory access from the LD instruction and so causes the memory exception process to begin in cycle 5. Note that the exception caused by the earlier instruction (LD) overrides the exception caused by the later illegal opcode even though the illegal opcode exception was detected first. That's the correct behavior since once the execution of LD is abandoned, the pipeline should behave as if none of the instructions that come after the LD were executed.

If multiple exceptions are detected in the *same* cycle, the exception from the instruction furthest down the pipeline should be given precedence.

Asynchronous Interrupts

Interrupts are easier:

```
// Interrupted code:
...
LD(...)
ADD(...)
SUB(...) ← HERE
...
// Interrupt handler:
XAdr: OR(...)

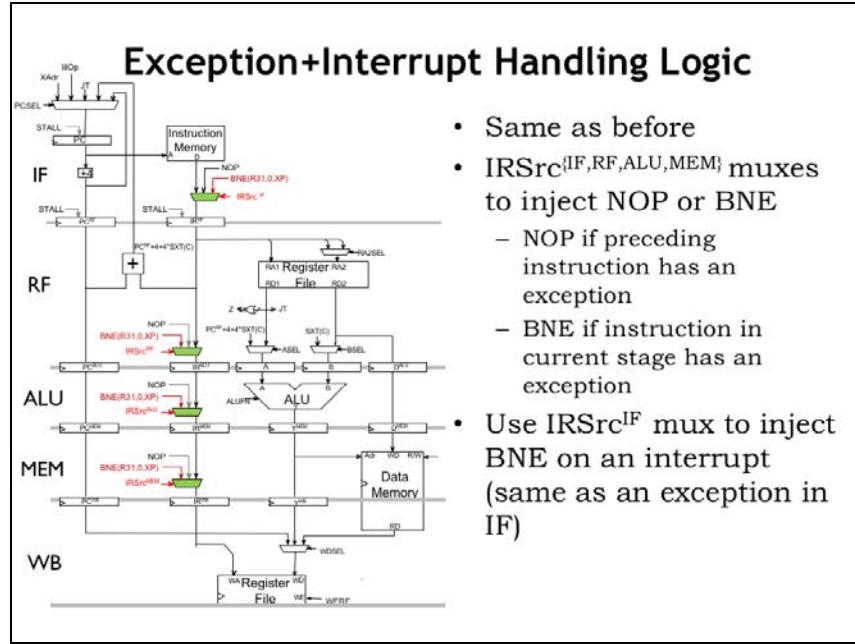
...
SUBC(xp,4,xp)
JMP(xp)
```

	I	2	3	4
IF	ADD	BNE	OR	...
RF	LD	ADD	BNE	OR
ALU		LD	ADD	BNE
MEM			LD	ADD
WB				LD

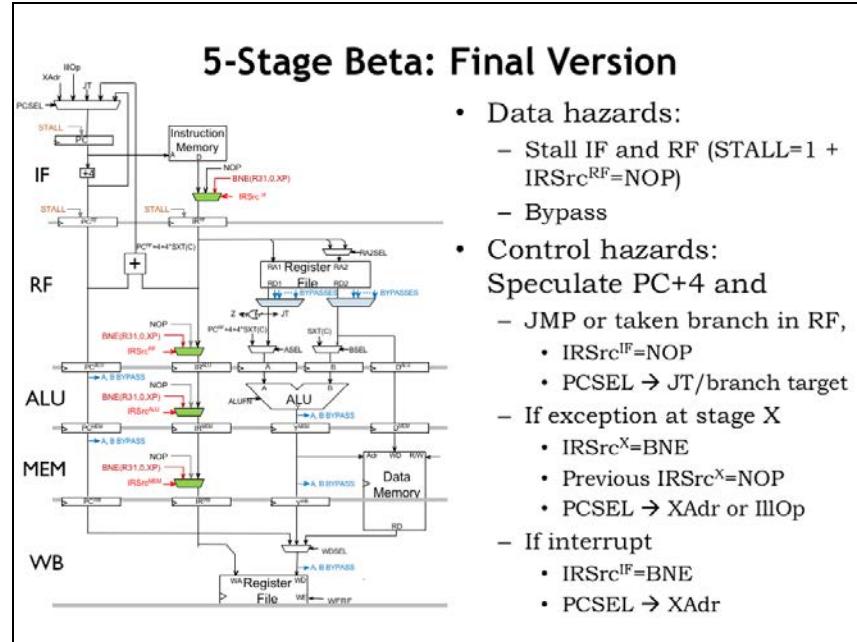
- Suppose interrupt is requested while SUB is in the IF stage (cycle 2)
- To handle:
 - Replace SUB instruction with BNE(...,XP)
 - Select Xadr as next PC
 - Code handler to return to SUB instruction
 - ADD and earlier insts. are unaffected

External interrupts also behave as implicit branches, but it turns out they are a bit easier to handle in our pipeline. We'll treat external interrupts as if they were an exception that affected the IF stage. Let's assume the external interrupt occurs in cycle 2. This means that the SUB instruction will be replaced by our magic BNE to capture the PC+4 value and we'll force the next PC to be the address of the interrupt handler. After the interrupt handler completes, we'll want to resume execution of the interrupted program at the SUB instruction, so we'll code the handler to correct the value saved in the XP register so that it points to the SUB instruction.

This is all shown in the pipeline diagram. Note that the ADD, LD, and other instructions that came before SUB in the program are unaffected by the interrupt.



We can use the existing instruction-path muxes to deal with interrupts, since we're treating them as IF-stage exceptions. We simply have to adjust the logic for IRSrc_IF to also make it 1 when an interrupt is requested.



So here's the final version of our 5-stage pipelined data path.

To deal with data hazards we've added stall logic to the IF and RF input registers. We've also added bypass muxes on the output of the register file read ports so we can route values from later in the data path if we need to access a register value that's been computed but not yet written to the register file. We also made a provision to insert NOPs into the pipeline after the RF stage if the IF and RF stages are stalled.

To deal with control hazards, we speculate that the next instruction is at PC+4. But for JMPs and taken branches, that guess is wrong so we added a provision for annulling the instruction in the IF stage.

To deal with exceptions and interrupts we added instruction muxes in all but the final pipeline stage. An instruction that causes an exception is replaced by our magic BNE instruction to capture its PC+4 value. And instructions in earlier stages are annulled.

All this extra circuitry has been added to ensure that pipelined execution gives the same result as unpipelined execution. The use of bypassing and branch prediction ensures that data and control hazards have only a small negative impact on the effective CPI. This means that the much shorter clock period translates to a large increase in instruction throughput.

Reminder: Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
 - Guess a value and continue executing anyway
 - When actual value is available, two cases
 - Guessed correctly → do nothing
 - Guessed incorrectly → kill & restart with correct value

It's worth remembering the strategies we used to deal with hazards: stalling, bypassing and speculation. Most execution issues can be dealt with using one of these strategies, so keep these in mind if you ever need to design a high-performance pipelined system.

This completes our discussion of pipelining. We'll explore other avenues to higher processor performance in a later lecture discussing parallel processing.

16: Virtual Memory

1. Reminder: A Typical Memory Hierarchy
2. Reminder: Hardware Caches
3. Reminder: A Typical Memory Hierarchy
4. Extending the Memory Hierarchy
5. Impact of Enormous Miss Penalty
6. Virtual Memory
7. Virtual Memory Implementation: Paging
8. Demand Paging
9. Simple Page Map Design
10. Example: Virtual → Physical Translation
11. Page Faults
12. Example: Page Fault
13. Virtual Memory: the CS View
14. The HW/SW Balance
15. Page Map Arithmetic
16. Example: Page Map Arithmetic
17. RAM-Resident Page Maps
18. Translation Look-aside Buffer (TLB)
19. MMU Address Translation
20. Putting it All Together: MMU with TLB
21. Contexts
22. Contexts: A Sneak Preview
23. Memory Management & Protection
24. Multi-level Page Maps
25. Rapid Context Switching
26. Using Caches with Virtual Memory
27. Best of Both Worlds: Overlapped Operation
28. Summary: Virtual Memory

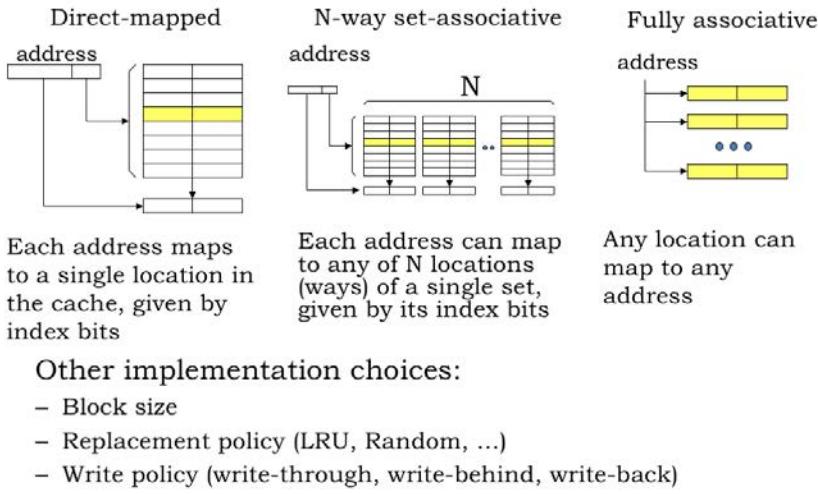
Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else

	Access time	Capacity	Managed By
On the datapath Registers	1 cycle	1 KB	Software/Compiler
Level 1 Cache	2-4 cycles	32 KB	Hardware
Level 2 Cache	10 cycles	256 KB	Hardware
On chip Level 3 Cache	40 cycles	10 MB	Hardware
Main Memory	200 cycles	10 GB	Software/OS
Other chips Flash Drive	10-100us	100 GB	Software/OS
Mechanical devices Hard Disk	10ms	1 TB	Software/OS

In this lecture we return to the memory system that we last discussed in “The Memory Hierarchy”. There we learned about the fundamental tradeoff in current memory technologies: as the memory’s capacity increases, so does its access time. It takes some architectural cleverness to build a memory system that has a large capacity and a small average access time. The cleverness is embodied in the cache, a hardware subsystem that lives between the CPU and main memory. Modern CPUs have several levels of cache, where the modest-capacity first level has an access time close to that of the CPU, and higher levels of cache have slower access times but larger capacities.

Reminder: Hardware Caches



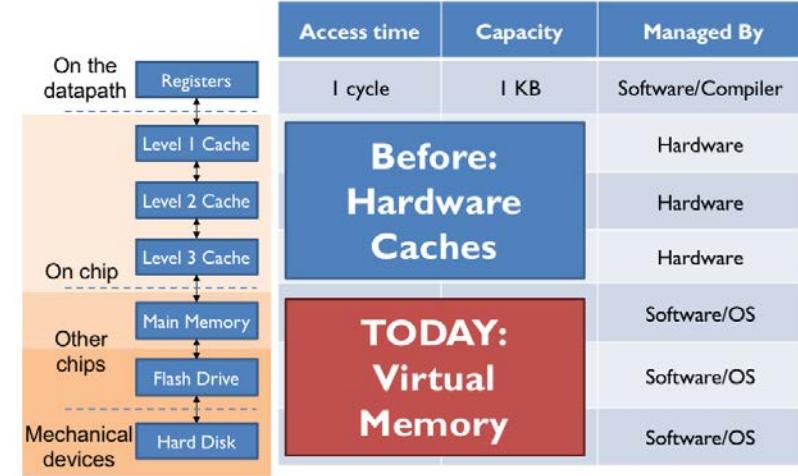
Caches give fast access to a small number of memory locations, using associative addressing so that the cache has the ability to hold the contents of the memory locations the CPU is accessing most frequently. The current contents of the cache are managed automatically by the hardware. Caches work well because of the principle of locality: if the CPU accesses location X at time T, it's likely to access nearby locations in the not-too-distant future. The cache is organized so that nearby locations can all reside in the cache simultaneously, using a simple indexing scheme to choose which cache location should be checked for a matching address. If the address requested by the CPU resides in the cache, access time is quite fast.

In order to increase the probability that requested addresses reside in the cache, we introduced the notion of “associativity”, which increased the number of cache locations checked on each access and solved the problem of having, say, instructions and data compete for the same cache locations.

We also discussed appropriate choices for block size (the number of words in a cache line), replacement policy (how to choose which cache line to reuse on a cache miss), and write policy (deciding when to write changed data back to main memory). We'll see these same choices again in this lecture as we work to expand the memory hierarchy beyond main memory.

Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else



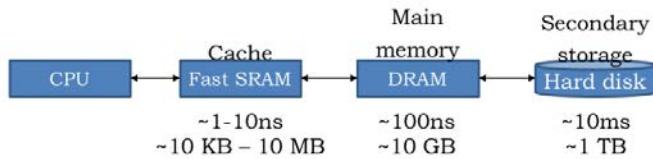
We never discussed where the data in main memory comes from and how the process of filling main memory is managed. That's the topic of today's lecture.

Flash drives and hard disks provide storage options that have more capacity than main memory, with the added benefit of being non-volatile, *i.e.*, they continue to store data even when turned off. The generic name for these new devices is “secondary storage”, where data will reside until it’s moved to “primary storage”, *i.e.*, main memory, for use. So when we first turn on a computer system, all of its data will be found in secondary storage, which we’ll think of as the final level of our memory hierarchy.

As we think about the right memory architecture, we’ll build on the ideas from our previous discussion of caches, and, indeed, think of main memory as another level of cache for the permanent, high-capacity secondary storage. We’ll be building what we call a virtual memory system, which, like caches, will automatically move data from secondary storage into main memory as needed. The virtual memory system will also let us control what data can be accessed by the program, serving as a stepping stone to building a system that can securely run many programs on a single CPU.

Let's get started!

Extending the Memory Hierarchy



- Problem: DRAM vs disk has much more extreme differences than SRAM vs DRAM
 - Access latencies:
 - DRAM ~10-100x slower than SRAM
 - Disk ~100,000x slower than DRAM
 - Importance of sequential accesses:
 - DRAM: Fetching successive words ~5x faster than first word
 - Disk: Fetching successive words ~100,000x faster than first word
- Result: Design decisions driven by **enormous cost of misses**

Here we see the cache and main memory, the two components of our memory system as developed in Lecture 14. And here's our new secondary storage layer. The good news: the capacity of secondary storage is huge! Even the most modest modern computer system will have 100's of gigabytes of secondary storage and having a terabyte or two is not uncommon on medium-size desktop computers. Secondary storage for the cloud can grow to many petabytes (a petabyte is 10^{15} bytes or a million gigabytes).

The bad news: disk access times are 100,000 times longer than those of DRAM. So the change in access time from DRAM to disk is much, much larger than the change from caches to DRAM.

When looking at DRAM timing, we discovered that the additional access time for retrieving a contiguous block of words was small compared to the access time for the first word, so fetching a block was the right plan assuming we'd eventually access the additional words. For disks, the access time difference between the first word and successive words is even more dramatic. So, not surprisingly, we'll be reading fairly large blocks of data from disk.

The consequence of the much, much larger secondary-storage access time is that it will be very time consuming to access disk if the data we need is not in main memory. So we need to design our virtual memory system to minimize misses when accessing main memory. A miss, and the subsequent disk access, will have a huge impact on the average memory access time, so the miss rate will need to be very, very, small compared to, say, the rate of executing instructions.

Impact of Enormous Miss Penalty

- If DRAM was to be organized like an SRAM cache, how should we design it?
 - Associativity: High, minimize miss ratio
 - Block size: Large, amortize cost of a miss over multiple words (locality)
 - Write policy: Write back, minimize number of writes
- Is there anything good about misses being so expensive?
 - We can handle them in software! What's 1000 extra instructions (~1us) vs 10ms?
 - Approach: Handle hits in hardware, misses in software
 - Simpler implementation, more flexibility

Given the enormous miss penalties of secondary storage, what does that tell us about how it should be used as part of our memory hierarchy?

We will need high associativity, *i.e.*, we need a great deal of flexibility on how data from disk can be located in main memory. In other words, if our working set of memory accesses fit in main memory, our virtual memory system should make that possible, avoiding unnecessary collisions between accesses to one block of data and another.

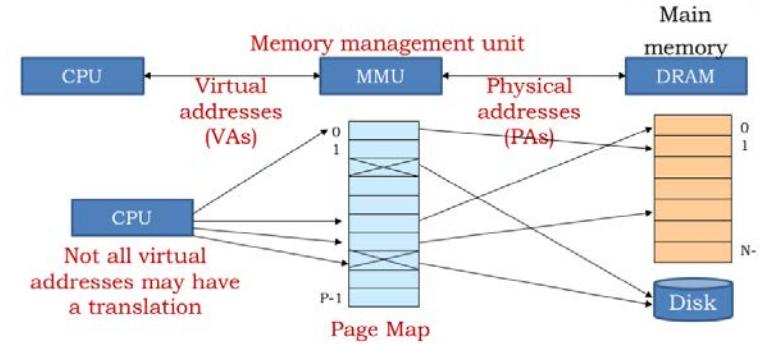
We'll want to use a large block size to take advantage of the low incremental cost of reading successive words from disk. And, given the principle of locality, we'd expect to be accessing other words of the block, thus amortizing the cost of the miss over many future hits.

Finally, we'll want to use a write-back strategy where we'll only update the contents of disk when data that's changed in main memory needs to be replaced by data from other blocks of secondary storage.

There is upside to misses having such long latencies. We can manage the organization of main memory and the accesses to secondary storage in software. Even it takes 1000's of instructions to deal with the consequences of a miss, executing those instructions is quick compared to the access time of a disk. So our strategy will be to handle hits in hardware and misses in software. This will lead to simple memory management hardware and the possibility of using very clever strategies implemented in software to figure out what to do on misses.

Virtual Memory

- Two kinds of addresses:
 - CPU uses **virtual addresses**
 - Main memory uses **physical addresses**
- Hardware translates virtual addresses to physical addresses via an operating system (OS)-managed table, the **page map**



Here's how our virtual memory system will work. The memory addresses generated by the CPU are called virtual addresses to distinguish them from the physical addresses used by main memory. In between the CPU and main memory there's a new piece of hardware called the memory management unit (MMU). The MMU's job is to translate virtual addresses to physical addresses.

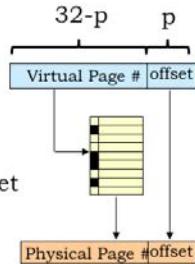
"But wait!" you say. "Doesn't the cache go between the CPU and main memory?" You're right and at the end of this lecture we'll talk about how to use both an MMU and a cache. But for now, let's assume there's only an MMU and no cache.

The MMU hardware translates virtual addresses to physical addresses using a simple table lookup. This table is called the page map or page table. Conceptually, the MMU uses the virtual address as index to select an entry in the table, which tells us the corresponding physical address. The table allows a particular virtual address to be found anywhere in main memory. In normal operation we'd want to ensure that two virtual addresses don't map to the same physical address. But it would be okay if some of the virtual addresses did not have a translation to a physical address. This would indicate that the contents of the requested virtual address haven't yet been loaded into main memory, so the MMU would signal a memory-management exception to the CPU, which could assign a location in physical memory and perform the required I/O operation to initialize that location from secondary storage.

The MMU table gives the system a lot of control over how physical memory is accessed by the program running on the CPU. For example, we could arrange to run multiple programs in quick succession (a technique called time sharing) by changing the page map when we change programs. Main memory locations accessible to one program could be made inaccessible to another program by proper management of their respective page maps. And we could use memory-management exceptions to load program contents into main memory on demand instead of having to load the entire program before execution starts. In fact, we only need to ensure the current working set of a program is actually resident in main memory. Locations not currently being used could live in secondary storage until needed. In this lecture and next, we'll see how the MMU plays a central role in the design of a modern timesharing computer system.

Virtual Memory Implementation: Paging

- Divide physical memory in fixed-size blocks, called **pages**
 - Typical page size (2^p): 4KB -16 KB
 - Virtual address: Virtual page number + offset bits
 - Physical address: Physical page number + offset bits
 - Why use lower bits as offset?
- MMU maps virtual pages to physical pages
 - Use page map to perform translation
 - Cause a **page fault** (a miss) if virtual page is not resident in physical memory.



Using main memory as a page cache = *paging or demand paging*

Of course, we'd need an impossibly large table to separately map each virtual address to a physical address. So instead we divide both the virtual and physical address spaces into fixed-sized blocks, called pages. Page sizes are always a power-of-2 bytes, say 2^p bytes, so p is the number address bits needed to select a particular location on the page. We'll use low-order p bits of the virtual or physical address as the page offset. The remaining address bits tell us which page is being accessed and are called the page number.

A typical page size is 4KB to 16KB, which correspond to $p=12$ and $p=14$ respectively. Suppose $p=12$. So if the CPU produces a 32-bit virtual address, the low-order 12 bits of the virtual address are the page offset and the high-order 20 bits are the virtual page number. Similarly, the low-order p bits of the physical address are the page offset and the remaining physical address bits are the physical page number.

The key idea is that the MMU will manage pages, not individual locations. We'll move entire pages from secondary storage into main memory. By the principle of locality, if a program access one location on a page, we expect it will soon access other nearby locations. By choosing the page offset from the low-order address bits, we'll ensure that nearby locations live on the same page (unless of course we're near one end of the page or the other). So pages naturally capture the notion of locality. And since pages are large, by dealing with pages when accessing secondary storage, we'll take advantage that reading or writing many locations is only slightly more time consuming than accessing the first location.

The MMU will map virtual page numbers to physical page numbers. It does this by using the virtual page number (VPN) as an index into the page table. Each entry in the page table indicates if the page is resident in main memory and, if it is, provides the appropriate physical page number (PPN). The PPN is combined with the page offset to form the physical address for main memory.

If the requested virtual page is NOT resident in main memory, the MMU signals a memory-management exception, called a **page fault**, to the CPU so it can load the appropriate page from secondary storage and set up the appropriate mapping in the MMU.

Our plan to use main memory as page cache is called “paging” or sometimes “demand paging” since movements of pages to and from secondary storage is determined by the demands of the program.

Demand Paging

Basic idea:

- Start with all virtual pages in secondary storage, MMU “empty”, ie, there are no pages resident in physical memory
- Begin running program... each VA “mapped” to a PA
 - Reference to RAM-resident page: RAM accessed by hardware
 - Reference to a non-resident page: page fault, which traps to software handler, which
 - Fetches missing page from DISK into RAM
 - Adjusts MMU to map newly-loaded virtual page directly in RAM
 - If RAM is full, may have to replace (“swap out”) some little-used page to free up RAM for the new page.
- Working set incrementally loaded via page faults, gradually evolves as pages are replaced...

So here's the plan. Initially all the virtual pages for a program reside in secondary storage and the MMU is empty, *i.e.*, there are no pages resident in physical memory.

The CPU starts running the program and each virtual address it generates, either for an instruction fetch or data access, is passed to the MMU to be mapped to a physical address in main memory.

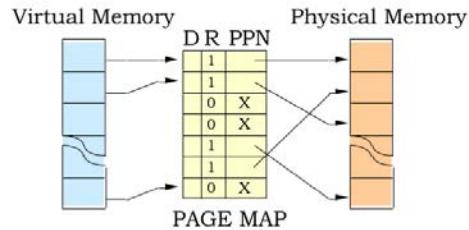
If the virtual address is resident in physical memory, the main memory hardware can complete the access.

If the virtual address is NOT resident in physical memory, the MMU signals a page fault exception, forcing the CPU to switch execution to special code called the page fault handler. The handler allocates a physical page to hold the requested virtual page and loads the virtual page from secondary storage into main memory. It then adjusts the page map entry for the requested virtual page to show that it is now resident and to indicate the physical page number for the newly allocated and initialized physical page.

When trying to allocate a physical page, the handler may discover that all physical pages are currently in use. In this case it chooses an existing page to replace, *e.g.*, a resident virtual page that hasn't been recently accessed. It swaps the contents of the chosen virtual page out to secondary storage and updates the page map entry for the replaced virtual page to indicate it is no longer resident. Now there's a free physical page to re-use to hold the contents of the virtual page that was missing.

The working set of the program, *i.e.*, the set of pages the program is currently accessing, is loaded into main memory through a series of page faults. After a flurry of page faults when the program starts running, the working set changes slowly, so the frequency of page faults drops dramatically, perhaps close to zero if the program is small and well-behaved. It is possible to write programs that constantly generate page faults, a phenomenon called thrashing. Given the long access times of secondary storage, a program that's thrashing runs **very** slowly, usually so slowly that user's give up and rewrite the program to behave more sensibly.

Simple Page Map Design



One entry per virtual page

- **Resident bit** R = 1 for pages stored in RAM, or 0 for non-resident (disk or unallocated). Page fault when R = 0
- Contains physical page number (PPN) of each resident page
- **Dirty bit** D = 1 if we've changed this page since loading it from disk (and therefore need to write it to disk when it's replaced)

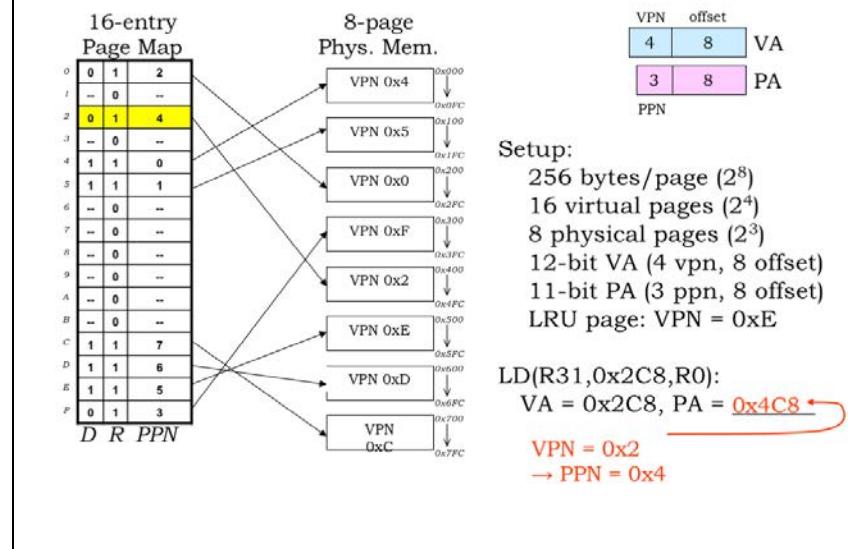
The design of the page map is straightforward. There's one entry in the page map for each virtual page. For example, if the CPU generates a 32-bit virtual address and the page size is 2^{12} bytes, the virtual page number has $32 - 12 = 20$ bits and the page table will have 2^{20} entries.

Each entry in the page table contains a “resident bit” (R) which is set to 1 when the virtual page is resident in physical memory. If R is 0, an access to that virtual page will cause a page fault. If R is 1, the entry also contains the PPN, indicating where to find the virtual page in main memory.

There's one additional state bit called the “dirty bit” (D). When a page has just been loaded from secondary storage, it's “clean”, i.e., the contents of physical memory match the contents of the page in secondary storage. So the D bit is set to 0. If subsequently the CPU stores into a location on the page, the D bit for the page is set to 1, indicating the page is “dirty”, i.e., the contents of memory now differ from the contents of secondary storage. If a dirty page is ever chosen for replacement, its contents must be written to secondary storage in order to save the changes before the page gets reused.

Some MMUs have additional state bits in each page table entry. For example, there could be a “read-only” bit which, when set, would generate an exception if the program attempts to store into the page. This would be useful for protecting code pages from accidentally being corrupted by errant data accesses, a very handy debugging feature.

Example: Virtual → Physical Translation



Here's an example of the MMU in action. To make things simple, assume that the virtual address is 12 bits, consisting of an 8-bit page offset and a 4-bit virtual page number. So there are $2^4 = 16$ virtual pages. The physical address is 11 bits, divided into the same 8-bit page offset and a 3-bit physical page number. So there are $2^3 = 8$ physical pages.

On the left we see a diagram showing the contents of the 16-entry page map, *i.e.*, an entry for each virtual page. Each page table entry includes a dirty bit (D), a resident bit (R) and a 3-bit physical page number, for a total of 5 bits. So the page map has 16 entries, each with 5-bits, for a total of $16 \times 5 = 80$ bits. The first entry in the table is for virtual page 0, the second entry for virtual page 1, and so on.

In the middle of the slide there's a diagram of physical memory showing the 8 physical pages. The annotation for each physical page shows the virtual page number of its contents. Note that there's no particular order to how virtual pages are stored in physical memory — which page holds what is determined by which pages are free at the time of a page fault. In general, after the program has run for a while, we'd expected to find the sort of jumbled ordering we see here.

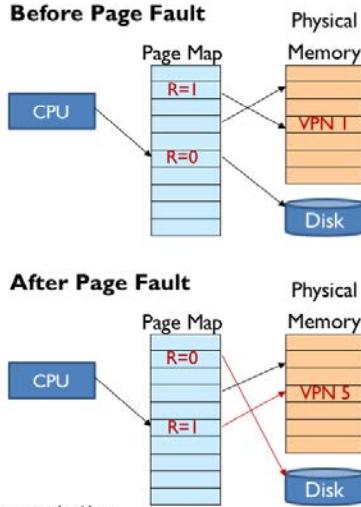
Let's follow along as the MMU handles the request for virtual address 0x2C8, generated by the execution of the LD instruction shown here. Splitting the virtual address into page number and offset, we see that the VPN is 2 and the offset is 0xC8. Looking at the page map entry with index 2, we see that the R bit is 1, indicating that virtual page 2 is resident in physical memory. The PPN field of entry tells us that virtual page 2 can be found in physical page 4.

Combining the PPN with the 8-bit offset, we find that the contents of virtual address 0x2C8 can be found in main memory location 0x4C8. Note that the offset is unchanged by the translation process — the offset into the physical page is always the same as the offset into the virtual page.

Page Faults

If a page does not have a valid translation, MMU causes a **page fault**. OS page fault handler is invoked, handles miss:

- Choose a page to replace, write it back if dirty. Mark page as no longer resident
 - Are there any restrictions on which page we can select? **
- Read page from secondary storage into available physical page
- Update page map to show new page is resident
- Return control to program, which re-executes memory access



** https://en.wikipedia.org/wiki/Page_replacement_algorithm#Page_replacement_algorithms

Let's review what happens when the CPU accesses a non-resident virtual page, *i.e.*, a page with its resident bit set to 0. In the example shown here, the CPU is trying to access virtual page 5.

In this case, the MMU signals a page fault exception, causing the CPU to suspend execution of the program and switch to the page fault handler, which is code that deals with the page fault. The handler starts by either finding an unused physical page or, if necessary, creating an unused page by selecting an in-use page and making it available. In our example, the handler has chosen virtual page 1 for reuse. If the selected page is dirty, *i.e.*, its D bit is 1 indicating that its contents have changed since being read from secondary storage, write it back to secondary storage. Finally, mark the selected virtual page as no longer resident. In the "after" figure, we see that the R bit for virtual page 1 has been set to 0. Now physical page 4 is available for re-use.

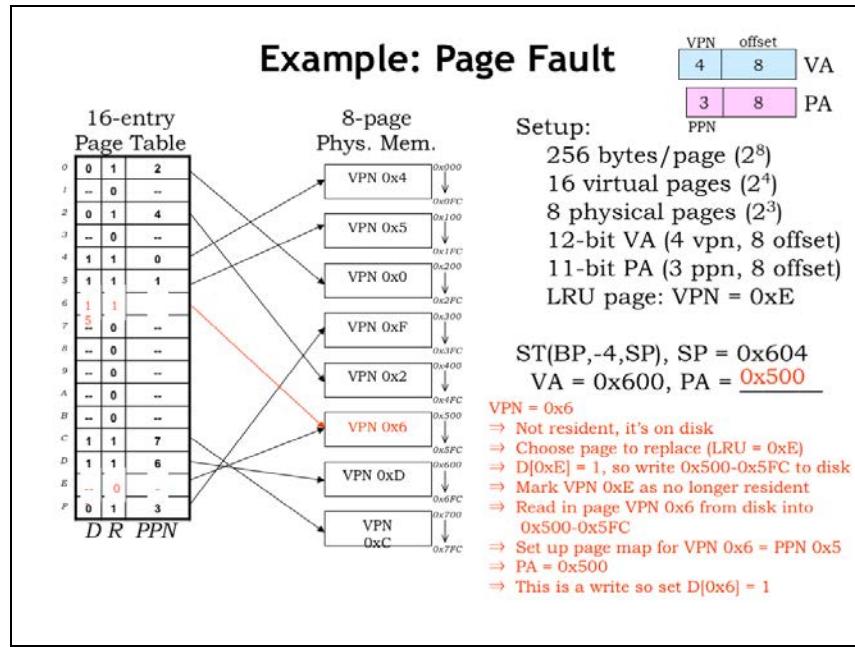
Are there any restrictions on which page we can select? Obviously, we can't select the page that holds the code for the page fault handler. Pages immune from selection are called "wired" pages. And it would be very inefficient to choose the page that holds the code that made the initial memory access, since we expect to start executing that code as soon as we finish handling the page fault.

The optimal strategy would be to choose the page whose next use will occur farthest in the future. But, of course, this involves knowledge of future execution paths and so isn't a realizable strategy. Wikipedia provides a nice description of the many strategies for choosing a replacement page, with their various tradeoffs between ease of implementation and impact on the rate of page faults — see the URL given at the bottom of the slide. The aging algorithm they describe is frequently used since it offers near optimal performance at a moderate implementation cost.

Next, the desired virtual page is read from secondary storage into the selected physical page. In our example, virtual page 5 is now loaded into physical page 4.

Then the R bit and PPN fields in the page table entry for virtual page 5 are updated to indicate that the contents of that virtual page now reside in physical page 4.

Finally the handler is finished and execution of the original program is resumed, re-executing the instruction that caused the page fault. Since the page map has been updated, this time the access succeeds and execution continues.



To double-check our understanding of page faults, let's run through an example. Here's the same setup as in our previous example, but this time consider a store instruction that's making an access to virtual address 0x600, which is located on virtual page 6.

Checking the page table entry for VPN 6, we see that its R bit 0 indicating that it is NOT resident in main memory, which causes a page fault exception.

The page fault handler selects VPN 0xE for replacement since we've been told in the setup that it's the least-recently-used page.

The page table entry for VPN 0xE has D=1 so the handler writes the contents of VPN 0xE, which is found in PPN 0x5, to secondary storage. Then it updates the page table to indicate that VPN 0xE is no longer resident.

Next, the contents of VPN 0x6 are read from secondary storage into the now available PPN 0x5.

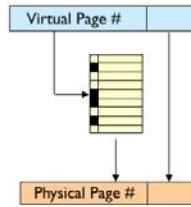
Now the handler updates the page table entry for VPN 0x6 to indicate that it's resident in PPN 0x5.

The page fault handler has completed its work, so program execution resumes and the ST instruction is re-executed. This time the MMU is able to translate virtual address 0x600 to physical address 0x500. And since the ST instruction modifies the contents of VPN 0x6, its D bit is set to 1.

Whew! We're done :)

Virtual Memory: the CS View

Problem: Translate
VIRTUAL ADDRESS
to PHYSICAL ADDRESS



```
int VtoP(int Vaddr) {
    int VPageNo = Vaddr >> p;
    int PO = Vaddr & ((1 << p) - 1);
    if (R[VPageNo] == 0)
        PageFault(VPageNo);
    return (PPN[VPageNo] << p) | PO;
} Multiply by 2p, the page size.

/* Handle a missing page... */
void PageFault(int VPageNo) {
    int i;

    i = SelectLRUPage();
    if (D[i] == 1)
        WritePage(DiskAddr[i], PPN[i]);
    R[i] = 0;

    PPN[VPageNo] = PPN[i];
    ReadPage(DiskAddr[VPageNo], PPN[i]);
    R[VPageNo] = 1;
    D[VPageNo] = 0;
}
```

We can think of the work of the MMU as being divided into two tasks, which as computer scientists, we would think of as two procedures. In this formulation the information in the page map is held in several arrays: the R array holds the resident bits, the D array holds the dirty bits, the PPN array holds the physical page numbers, and the DiskAdr array holds the location in secondary storage for each virtual page.

The VtoP procedure is invoked on each memory access to translate the virtual address into a physical address. If the requested virtual page is not resident, the PageFault procedure is invoked to make the page resident. Once the requested page is resident, the VPN is used as an index to lookup the corresponding PPN, which is then concatenated with the page offset to form the physical address.

The PageFault routine starts by selecting a virtual page to be replaced, writing out its contents if it's dirty. The selected page is then marked as not resident.

Finally the desired virtual page is read from secondary storage and the page map information updated to reflect that it's now resident in the newly filled physical page.

The HW/SW Balance

IDEA:

- devote HARDWARE to high-traffic, performance-critical path
- use (slow, cheap) SOFTWARE to handle exceptional cases

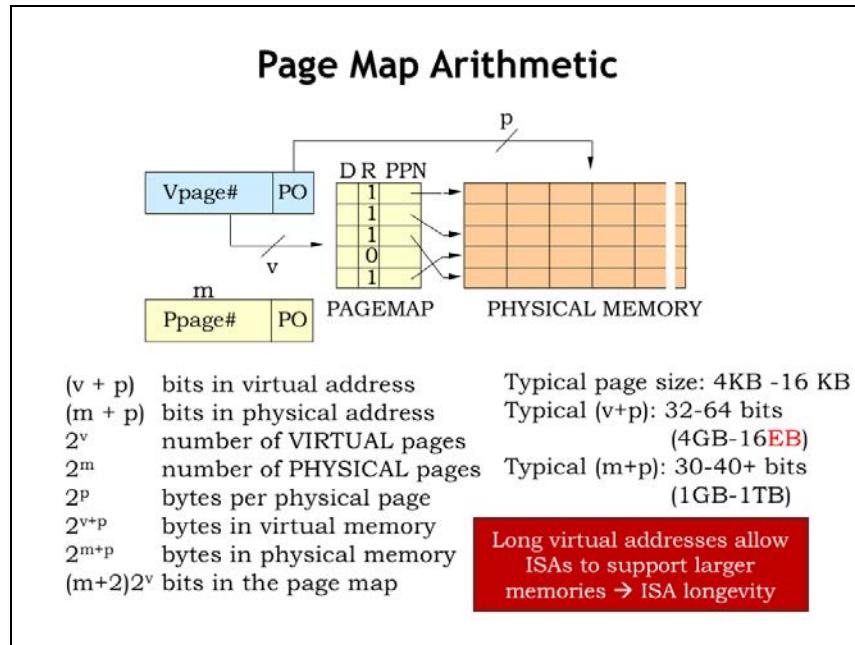
```
hardware {  
    int VtoP(int VPageNo,int PO) {  
        if (R[VPageNo] == 0) PageFault(VPageNo);  
        return (PPN[VPageNo] << p) | PO;  
    }  
  
    /* Handle a missing page... */  
    void PageFault(int VPageNo) {  
        int i = SelectLRUPage();  
        if (D[i] == 1) WritePage(DiskAddr[i],PPN[i]);  
        R[i] = 0;  
  
        PA[VPageNo] = PPN[i];  
        ReadPage(DiskAddr[VPageNo],PPN[i]);  
        R[VPageNo] = 1;  
        D[VPageNo] = 0;  
    }  
}
```

HARDWARE performs address translation, detects page faults:

- running program interrupted (“suspended”);
- PageFault(...) is forced;
- On return from PageFault; running program continues

We'll use hardware to implement the VtoP functionality since it's needed for every memory access. The call to the PageFault procedure is accomplished via a page fault exception, which directs the CPU to execute the appropriate handler software that contains the PageFault procedure.

This is a good strategy to pursue in all our implementation choices: use hardware for the operations that need to be fast, but use exceptions to handle the (hopefully infrequent) exceptional cases in software. Since the software is executed by the CPU, which is itself a piece of hardware, what we're really doing is making the tradeoff between using special-purpose hardware (*e.g.*, the MMU) or using general-purpose hardware (*e.g.*, the CPU). In general, one should be skeptical of proposals to use special-purpose hardware, reserving that choice for operations that truly are commonplace and whose performance is critical to the overall performance of the system.



There are three architectural parameters that characterize a virtual memory system and hence the architecture of the MMU.

P is the number of address bits used for the page offset in both virtual and physical addresses. V is the number of address bits used for the virtual page number. And M is the number of address bits used for the physical page number. All the other parameters, listed on the right, are derived from these three parameters.

As mentioned earlier, the typical page size is between 4KB and 16KB, the sweet spot in the tradeoff between the downside of using physical memory to hold unwanted locations and the upside of reading as much as possible from secondary storage so as to amortize the high cost of accessing the initial word over as many words as possible.

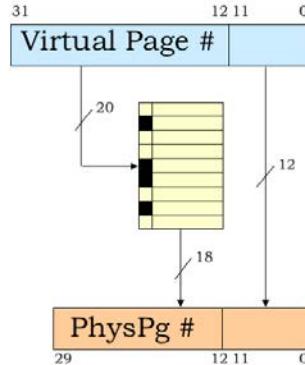
The size of the virtual address is determined by the ISA. We're now making the transition from 32-bit architectures, which support a 4 gigabyte virtual address space, to 64-bit architectures, which support a 16 exabyte virtual address space. "Exa" is the SI prefix for 10^{18} - a 64-bit address can access a lot of memory!

The limitations of a small virtual address have been the main cause for the extinction of many ISAs. Of course, each generation of engineers thinks that the transition they make will be the final one! I can remember when we all thought that 32 bits was an unimaginably large address. Back then we're buying memory by the megabyte and only in our fantasies did we think one could have a system with several thousand megabytes. Today's CPU architects are feeling pretty smug about 64 bits — we'll see how they feel in a couple of decades!

The size of physical addresses is currently between 30 bits (for embedded processors with modest memory needs) and 40+ bits (for servers that handle large data sets). Since CPU implementations are expected to change every couple of years, the choice of physical memory size can be adjusted to match current technologies. Since programmers use virtual addresses, they're insulated from this implementation choice. The MMU ensures that existing software will continue to function correctly

with different sizes of physical memory. The programmer may notice differences in performance, but not in basic functionality.

Example: Page Map Arithmetic



SUPPOSE...

32-bit Virtual address ($v+p$)

30-bit physical address ($m+p$)

4 KB page size ($p = 12$)

THEN:

$$\# \text{ Physical Pages} = \underline{2^{18} = 256\text{K}}$$

$$\# \text{ Virtual Pages} = \underline{2^{20}}$$

$$\# \text{ Page Map Entries} = \underline{2^{20} = 1\text{M}}$$

$$\# \text{ Bits In pagemap} = \underline{20 * 2^{20} \sim 20\text{M}}$$

Use fast SRAM for page map??? OUCH!

For example, suppose our system supported a 32-bit virtual address, a 30-bit physical address and a 4KB page size. So $p = 12$, $v = 32 - 12 = 20$, and $m = 30 - 12 = 18$.

There are 2^m physical pages, which is 2^{18} in our example.

There are 2^v virtual pages, which is 2^{20} in our example.

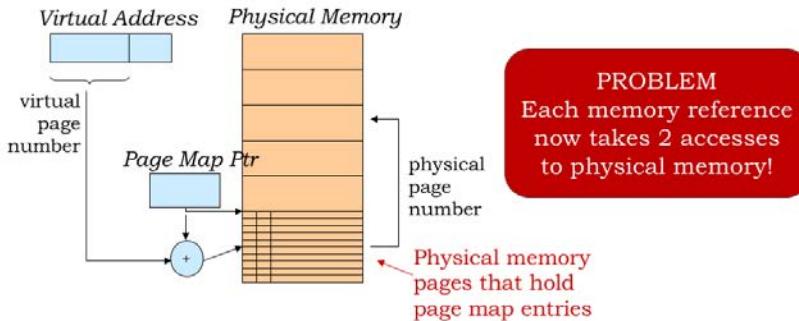
And since there is one entry in the page map for each virtual page, there are 2^{20} (approximately one million) page map entries.

Each page map entry contains a PPN, an R bit and a D bit, for a total of $m+2$ bits, which is 20 bits in our example. So there are approximately 20 million bits in the page map.

If we were thinking of using a large special-purpose static RAM to hold the page map, this would get pretty expensive!

RAM-Resident Page Maps

- **Small** page maps can use dedicated SRAM... gets expensive for big ones!
- Solution: Move page map to **main memory**:



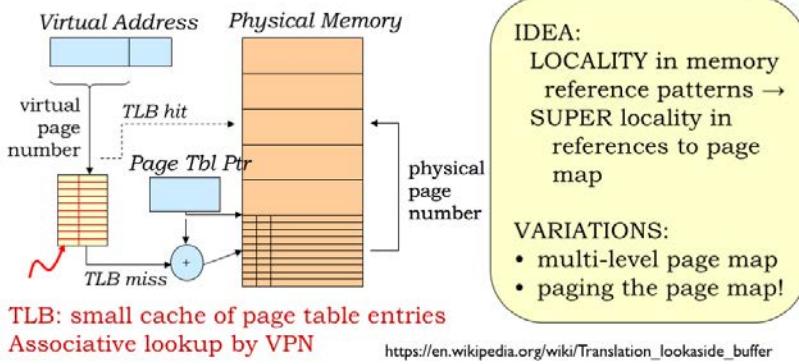
But why use a special-purpose memory for the page map? Why not use a portion of main memory, which we have a lot of and have already bought and paid for!

We could use a register, called the page map pointer, to hold the address of the page map array in main memory. In other words, the page map would occupy some number of dedicated physical pages. Using the desired virtual page number as an index, the hardware could perform the usual array access calculation to fetch the needed page map entry from main memory.

The downside of this proposed implementation is that it now takes two accesses to physical memory to perform one virtual access: the first to retrieve the page table entry needed for the virtual-to-physical address translation, and the second to actually access the requested location.

Translation Look-aside Buffer (TLB)

- Problem: 2x performance hit... each memory reference now takes 2 accesses!
- Solution: Cache the page map entries

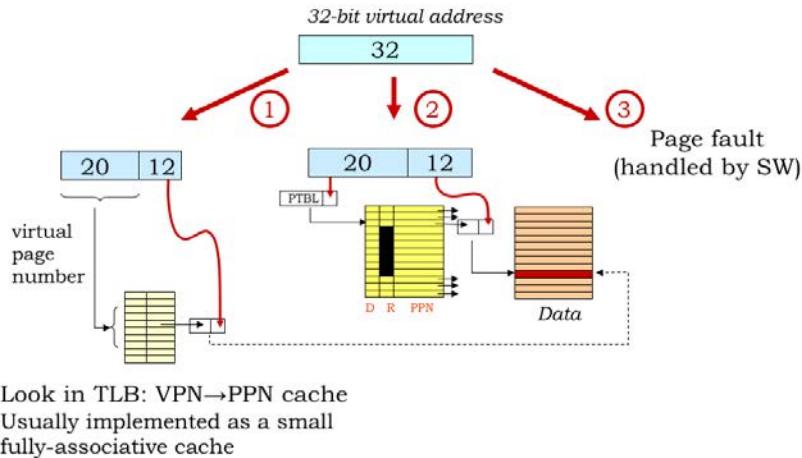


Once again, caches to the rescue. Most systems incorporate a special-purpose cache, called a translation look-aside buffer (TLB), that maps virtual page numbers to physical page numbers. The TLB is usually small and quite fast. It's usually fully-associative to ensure the best possible hit ratio by avoiding collisions. If the PPN is found by using the TLB, the access to main memory for the page table entry can be avoided, and we're back to a single physical access for each virtual access.

The hit ratio of a TLB is quite high, usually better than 99%. This isn't too surprising since locality and the notion of a working set suggest that only a small number of pages are in active use over short periods of time.

As we'll see in a few slides, there are interesting variations to this simple TLB page-map-in-main-memory architecture. But the basic strategy will remain the same.

MMU Address Translation



Putting it all together: the virtual address generated by the CPU is first processed by the TLB to see if the appropriate translation from VPN to PPN has been cached. If so, the main memory access can proceed directly.

If the desired mapping is not in the TLB, the appropriate entry in the page map is accessed in main memory. If the page is resident, the PPN field of the page map entry is used to complete the address translation. And, of course, the translation is cached in the TLB so that subsequent accesses to this page can avoid the access to the page map.

If the desired page is not resident, the MMU triggers a page fault exception and the page fault handler code will deal with the problem.

Putting it All Together: MMU with TLB

Suppose

- virtual memory of 2^{32} bytes
- physical memory of 2^{24} bytes
- page size is 2^{10} (1 K) bytes
- 4-entry fully associative TLB
- $[p = 10, v = 22, m = 14]$

TLB		VPN	Page Map		
Tag	Data		R	D	PPN
VPN R D PPN			0 0 7		
-----+-----			1 1 9		
0 0 0 3			2 0 0		
6 1 1 2			3 0 0 5		
1 1 1 9			4 1 0 5		
3 0 0 5			5 0 0 3		
			6 1 1 2		
			7 1 0 4		
			8 1 0 1		
			...		

1. How many pages can reside in physical memory at one time? 2^{14}
2. How many entries are there in the page table? 2^{22}
3. How many bits per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit) 16
4. How many pages does the page table occupy? 2^{23} bytes = 2^{13} pages
5. What fraction of virtual memory can be resident? $1/2^8$
6. What is the physical address for virtual address 0x1804? What components are involved in the translation? [VPN=6] 0x804
7. Same for 0x1080 [VPN=4] 0x1480
8. Same for 0x0FC [VPN=0] page fault

Here's a final example showing all the pieces in action. In this example, $p = 10$, $v = 22$, and $m = 14$

How many pages can reside in physical memory at one time? There are 2^m physical pages, so 2^{14} .

How many entries are there in the page table? There's one entry for each virtual page and there are 2^v virtual pages, so there are 2^{22} entries in the page table.

How many bits per entry in the page table? Assume each entry holds the PPN, the resident bit, and the dirty bit. Since the PPN is m bits, there are $m + 2$ bits in each entry, so 16 bits.

How many pages does the page table occupy? There are 2^v page table entries, each occupying $(m + 2)/8$ bytes, so the total size of the page table in this example is 2^{23} bytes. Each page holds $2^p = 2^{10}$ bytes, so the page table occupies $2^{23}/2^{10} = 2^{13}$ pages.

What fraction of virtual memory can be resident at any given time? There are 2^v virtual pages, of which 2^m can be resident. So the fraction of resident pages is $2^m/2^v = 2^{14}/2^{22} = 1/2^8$.

What is the physical address for virtual address 0x1804? Which MMU components are involved in the translation? First we have to decompose the virtual address into VPN and offset. The offset is the low-order 10 bits, so is 0x004 in this example. The VPN is the remaining address bits, so the VPN is 0x6. Looking first in the TLB, we see that the VPN-to-PPN mapping for VPN 0x6 is cached, so we can construct the physical address by concatenating the PPN (0x2) with the 10-bit offset (0x4) to get a physical address of 0x804. You're right! It's a bit of pain to do all the bit manipulations when p is not a multiple of 4.

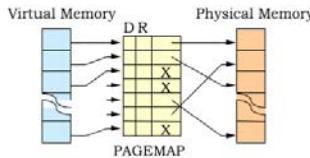
How about virtual address 0x1080? For this address the VPN is 0x4 and the offset is 0x80. The translation for VPN 0x4 is not cached in the TLB, so we have to check the page map, which tells us that the page is resident in physical page 5. Concatenating the PPN and offset, we get 0x1480 as the physical address.

Finally, how about virtual address 0x0FC? Here the VPN is 0 and the offset 0xFC. The mapping for VPN 0 is not found in the TLB and checking the page map reveals that VPN 0 is not resident in main memory, so a page fault exception is triggered.

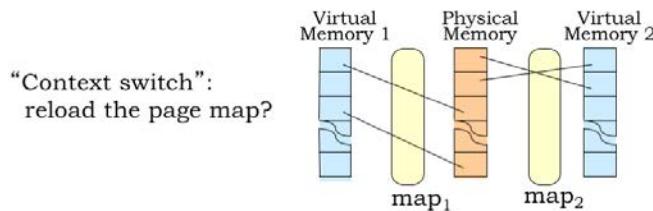
There are a few things to note about the example TLB and page map contents. Note that a TLB entry can be invalid (it's R bit is 0). This can happen when a virtual page is replaced, so when we change the R bit to 0 in the page map, we have to do the same in the TLB. And should we be concerned that PPN 0x5 appears twice in the page table? Note that the entry for VPN 0x3 doesn't matter since it's R bit is 0. Typically when marking a page not resident, we don't bother to clear out the other fields in the entry since they won't be used when R=0. So there's only one **valid** mapping to PPN 5.

Contexts

A **context** is a mapping of VIRTUAL to PHYSICAL locations, as dictated by contents of the page map:



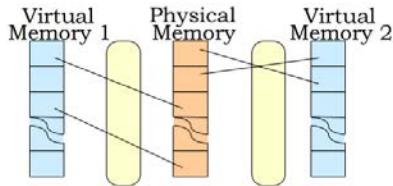
Several programs may be simultaneously loaded into main memory, each in its separate context:



The page map provides the context for interpreting virtual addresses, *i.e.*, it provides the information needed to correctly determine where to find a virtual address in main memory or secondary storage.

Several programs may be simultaneously loaded into main memory, each with its own context. Note that the separate contexts ensure that the programs don't interfere with each other. For example, the physical location for virtual address 0 in one program will be different than the physical location for virtual address 0 in another program. Each program operates independently in its own virtual address space. It's the context provided by the page map that allows them to coexist and share a common physical memory. So we need to switch contexts when switching programs. This is accomplished by reloading the page map.

Contexts: A Sneak Preview



First Glimpse of a VIRTUAL MACHINE

1. TIMESHARING among several programs
 - Separate context for each program
 - OS loads appropriate context into page map when switching among programs
2. Separate context for OS "Kernel" (e.g., interrupt handlers)...
 - "Kernel" vs "User" contexts
 - Switch to Kernel context on interrupt;
 - Switch back on interrupt return.

In a timesharing system, the CPU will periodically switch from running one program to another, giving the illusion that multiple programs are each running on their own virtual machine. This is accomplished by switching contexts when switching the CPU state to the next program.

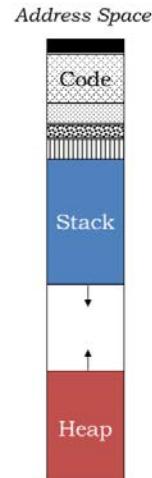
There's a privileged set of code called the operating system (OS) that manages the sharing of one physical processor and main memory amongst many programs, each with its own CPU state and virtual address space. The OS is effectively creating many virtual machines and choreographing their execution using a single set of shared physical resources.

The OS runs in a special OS context, which we call the kernel. The OS contains the necessary exception handlers and timesharing support. Since it has to manage physical memory, it's allowed to access any physical location as it deals with page faults, etc. Exceptions in running programs cause the hardware to switch to the kernel context, which we call entering "kernel mode". After the exception handling is complete, execution of the program resumes in what we call "user mode".

Since the OS runs in kernel mode it has privileged access to many hardware registers that are inaccessible in user mode. These include the MMU state, I/O devices, and so on. User-mode programs that need to access, say, the disk, need to make a request to the OS kernel to perform the operation, giving the OS the chance to vet the request for appropriate permissions, etc. We'll see how all of this works in an upcoming lecture.

Memory Management & Protection

- Applications are written as if they have access to the entire virtual address space, without considering where other applications reside
 - Enables fixed conventions (e.g., program starts at 0x1000, stack is contiguous and grows up, ...) without worrying about conflicts
- OS Kernel controls all contexts, prevents programs from reading and writing into each other's memory



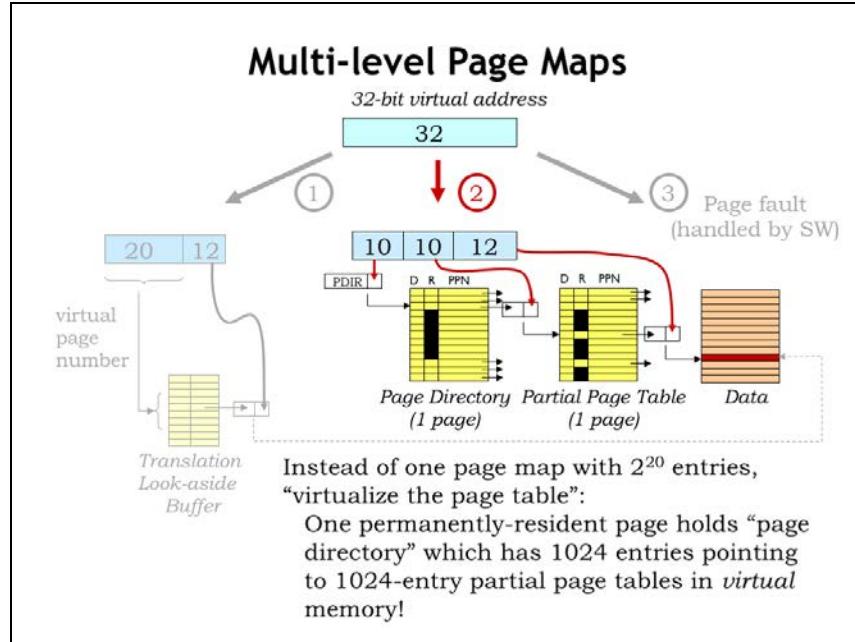
User-mode programs (aka applications) are written as if they have access to the entire virtual address space. They often obey the same conventions such as the address of the first instruction in the program, the initial value for the stack pointer, etc. Since all these virtual addresses are interpreted using the current context, by controlling the contexts the OS can ensure that the programs can coexist without conflict.

The diagram on the right shows a standard plan for organizing the virtual address space of an application. Typically the first virtual page is made inaccessible, which helps catch errors involving references to initialized (*i.e.*, zero-valued) pointers. Then come some number of read-only pages that hold the application's code and perhaps the code from any shared libraries it uses. Marking code pages as read-only avoids hard-to-find bugs where errant data accesses inadvertently change the program!

Then there are read-write pages holding the application's statically allocated data structures. The rest of the virtual address space is divided between two data regions that can grow over time. The first is the application's stack, used to hold procedure activation records. Here we show it located at the lower end of the virtual address space since our convention is that the stack grows towards higher addresses.

The other growable region is the heap, used when dynamically allocating storage for long-lived data structures. “Dynamically” means that the allocation and deallocation of objects is done by explicit procedure calls while the application is running. In other words, we don't know which objects will be created until the program actually executes. As shown here, as the heap expands it grows towards lower addresses.

The page fault handler knows to allocate new pages when these regions grow. Of course, if they ever meet somewhere in the middle and more space is needed, the application is out of luck — it's run out of virtual memory!



There are a few MMU implementation details we can tweak for more efficiency or functionality.

In our simple page-map implementation, the full page map occupies some number of physical pages. Using the numbers shown here, if each page map occupies one word of main memory, we'd need 2^{20} words (or 2^{12} pages) to hold the page table. If we have multiple contexts, we would need multiple page tables, and the demands on our physical memory resources would start to get large.

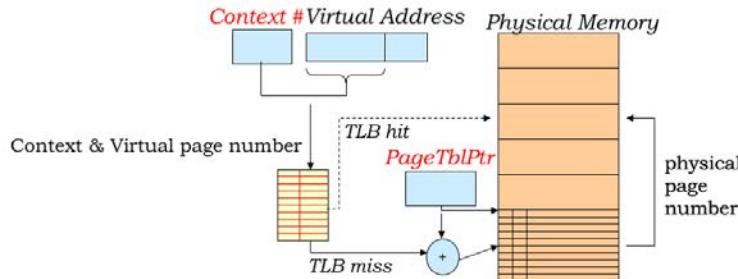
The MMU implementation shown here uses a hierarchical page map. The top 10 bits of virtual address are used to access a “page directory”, which indicates the physical page that holds the page map for that segment of the virtual address space. The key idea is that the page map segments are in virtual memory, *i.e.*, they don't all have to be resident at any given time. If the running application is only actively using a small portion of its virtual address space, we may only need a handful of pages to hold the page directory and the necessary page map segments. The resultant savings really add up when there are many applications, each with their own context.

In this example, note that the middle entries in the page directory, *i.e.*, the entries corresponding to the as-yet unallocated virtual memory between the stack and heap, are all marked as not resident. So no page map resources need be devoted to holding a zillion page map entries all marked “not resident”.

Accessing the page map now requires two access to main memory (first to the page directory, then to the appropriate segment of the page map), but the TLB makes the impact of that additional access negligible.

Rapid Context -Switching

Add a register to hold index of current context. To switch contexts: update Context # and PageTblPtr registers. Don't have to flush TLB since each entry's tag includes context # in addition to virtual page number

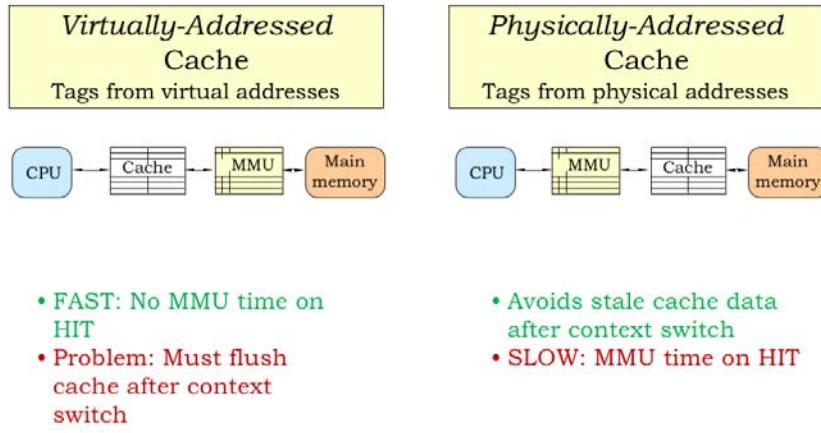


Normally when changing contexts, the OS would reload the page-table pointer to point to the appropriate page table (or page table directory if we adopt the scheme from the previous slide). Since this context switch in effect changes all the entries in the page table, the OS would also have to invalidate all the entries in the TLB cache. This naturally has a huge impact on the TLB hit ratio and the average memory access time takes a huge hit because of the all page map accesses that are now necessary until the TLB is refilled.

To reduce the impact of context switches, some MMUs include a context-number register whose contents are concatenated with the virtual page number to form the query to the TLB. Essentially this means that the tag field in the TLB cache entries will expand to include the context number provided at the time the TLB entry was filled.

To switch contexts, the OS would now reload both the context-number register and the page-table pointer. With a new context number, entries in the TLB for other contexts would no longer match, so no need to flush the TLB on a context switch. If the TLB has sufficient capacity to cache the VPN-to-PPN mappings for several contexts, context switches would no longer have a substantial impact on average memory access time.

Using Caches with Virtual Memory

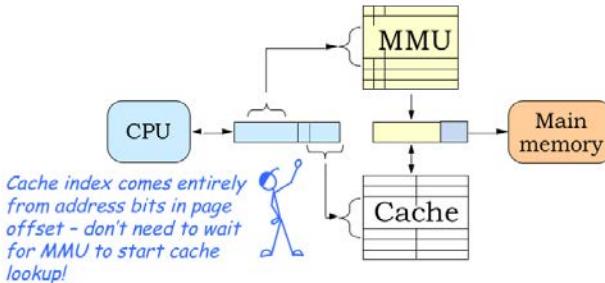


Finally, let's return to the question about how to incorporate both a cache and an MMU into our memory system.

The first choice is to place the cache between the CPU and the MMU, *i.e.*, the cache would work on virtual addresses. This seems good: the cost of the VPN-to-PPN translation is only incurred on a cache miss. The difficulty comes when there's a context switch, which changes the effective contents of virtual memory. After all that was the point of the context switch, since we want to switch execution to another program. But that means the OS would have to invalidate all the entries in the cache when performing a context switch, which makes the cache miss ratio quite large until the cache is refilled. So once again the performance impact of a context switch would be quite high.

We can solve this problem by caching physical addresses, *i.e.*, placing the cache between the MMU and main memory. Thus the contents of the cache are unaffected by context switches — the requested physical addresses will be different, but the cache handles that in due course. The downside of this approach is that we have to incur the cost of the MMU translation before we can start the cache access, slightly increasing the average memory access time.

Best of Both Worlds: Overlapped Operation



OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can *overlap* page map access. Tag from cache is compared with physical page address from MMU to determine hit/miss.

Problem: Limits # of bits of cache index → increase cache capacity by increasing associativity

But if we're clever we don't have to wait for the MMU to finish before starting the access to the cache. To get started, the cache needs the line number from the virtual address in order to fetch the appropriate cache line. If the address bits used for the line number are completely contained in the page offset of the virtual address, those bits are unaffected by the MMU translation, and so the cache lookup can happen in parallel with the MMU operation.

Once the cache lookup is complete, the tag field of the cache line can be compared with the appropriate bits of the physical address produced by the MMU. If there was a TLB hit in the MMU, the physical address should be available at about the same time as the tag field produced by the cache lookup.

By performing the MMU translation and cache lookup in parallel, there's usually no impact on the average memory access time! Voila, the best of both worlds: a physically addressed cache that incurs no time penalty for MMU translation.

One final detail: one way to increase the capacity of the cache is to increase the number of cache lines and hence the number of bits of address used as the line number. Since we want the line number to fit into the page offset field of the virtual address, we're limited in how many cache lines we can have. The same argument applies to increasing the block size. So to increase the capacity of the cache our only option is to increase the cache associativity, which adds capacity without affecting the address bits used for the line number.

Summary: Virtual Memory

- Goal 1: Exploit locality on a large scale
 - Programmers want a large, flat address space, but use a small portion!
 - Solution: Cache working set into RAM from disk
 - Basic implementation: MMU with single-level page map
 - Access loaded pages via fast hardware path
 - Load virtual memory on demand: page faults
 - Several optimizations:
 - Moving page map to RAM, for cost reasons
 - Translation Lookaside Buffer (TLB) to regain performance
 - Cache/VM interactions: Can cache physical or virtual locations
- Goals 2 & 3: Ease memory management, protect multiple contexts from each other
 - We'll see these in detail on the next lecture!

That's it for our discussion of virtual memory. We use the MMU to provide the context for mapping virtual addresses to physical addresses. By switching contexts we can create the illusion of many virtual address spaces, so many programs can share a single CPU and physical memory without interfering with each other.

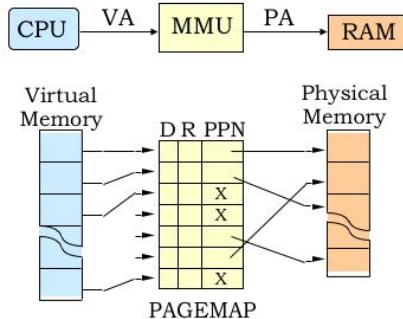
We discussed using a page map to translate virtual page numbers to physical page numbers. To save costs, we located the page map in physical memory and used a TLB to eliminate the cost of accessing the page map for most virtual memory accesses. Access to a non-resident page causes a page fault exception, allowing the OS to manage the complexities of equitably sharing physical memory across many applications.

We saw that providing contexts was the first step towards creating virtual machines, which is the topic of our next lecture.

17: Virtualizing the Processor

1. Review: Virtual Memory
2. MMU Address Translation
3. Contexts
4. Building a Virtual Machine (VM)
5. One VM For Each Process
6. Processes: Multiplexing the CPU
7. Key Technology: Timer Interrupts
8. Beta Interrupt Handling
9. Example: Timer Interrupt Handler
10. Interrupt Handler Coding
11. Simple Timesharing Scheduler
12. OS Organization: Processes
13. One Interrupt at a Time
14. Exception Hardware
15. Exception Handling
16. Useful Macros
17. Illop Handler
18. Accessing User Locations
19. Handler for Actual Illops
20. Emulated Instruction: swapreg(Ra,Rc)
21. Communicationg with the OS
22. OS Organization: Supervisor Calls
23. Handler for SVCs
24. Returning to User-mode
25. Adding New SVCs
26. New SVC Handlers

Review: Virtual Memory



Goal: create illusion of large virtual address space

- divide address into (VPN,offset), map to (PPN,offset) or page fault
- use high address bits to select page: keep related data on same page
- use cache (TLB) to speed up mapping mechanism—works well
- long disk latencies: keep working set in physical memory, use write-back

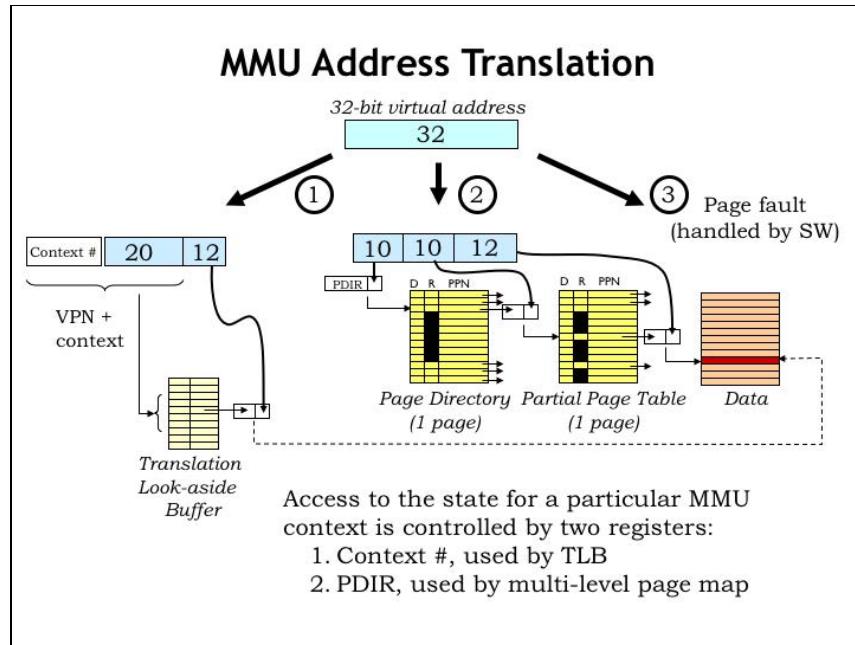
In the last lecture we introduced the notion of virtual memory and added a Memory Management Unit (MMU) to translate the virtual addresses generated by the CPU to the physical addresses sent to main memory. This gave us the ability to share physical memory between many running programs while still giving each program the illusion of having its own large address space.

Both the virtual and physical address spaces are divided into a sequence of pages, each holding some fixed number of locations. For example if each page holds 2^{12} bytes, a 32-bit address space would have $2^{32}/2^{12} = 2^{20}$ pages. In this example the 32-bit address can be thought of as having two fields: a 20-bit page number formed from the high-order address bits and a 12-bit page offset formed from the low-order address bits. This arrangement ensures that nearby data will be located on the same page.

The MMU translates virtual page numbers into physical page numbers using a page map. Conceptually the page map is an array where each entry in the array contains a physical page number along with a couple of bits indicating the page status. The translation process is simple: the virtual page number is used as an index into the array to fetch the corresponding physical page number. The physical page number is then combined with the page offset to form the complete physical address.

In the actual implementation the page map is usually organized into multiple levels, which permits us to have resident only the portion of the page map we're actively using. And to avoid the costs of accessing the page map on each address translation, we use a cache (called the translation look-aside buffer) to remember the results of recent VPN-to-PPN translations.

All allocated locations of each virtual address space can be found on secondary storage. Note that they may not necessarily be resident in main memory. If the CPU attempts to access a virtual address that's not resident in main memory, a page fault is signaled and the operating system will arrange to move the desired page from second storage into main memory. In practice, only the active pages for each program are resident in main memory at any given time.



Here's a diagram showing the translation process. First we check to see if the required VPN-to-PPN mapping is cached in the TLB. If not, we have to access the hierarchical page map to see if the page is resident and, if so, lookup its physical page number. If we discover that the page is not resident, a page fault exception is signaled to the CPU so that it can run a handler to load the page from secondary storage.

Note that access to a particular mapping context is controlled by two registers. The context-number register controls which mappings are accessible in the TLB. And the page-directory register indicates which physical page holds the top tier of the hierarchical page map. We can switch to another context by simply reloading these two registers.

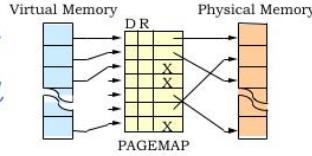
To effectively accommodate multiple contexts we'll need to have sufficient TLB capacity to simultaneously cache the most frequent mappings for all the processes. And we'll need some number of physical pages to hold the required page directories and segments of the page tables. For example, for a particular process, three pages will suffice hold the resident two-level page map for 1024 pages at each end of the virtual address space, providing access to up to 8MB of code, stack, and heap, more than enough for many simple programs.

Contexts

A **context** is an entire set of mappings from VIRTUAL to PHYSICAL page numbers as specified by the contents of the page map:



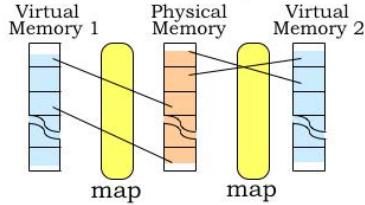
We would like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.



THE BIG IDEA: Several programs, each with their own context, may be simultaneously loaded into main memory!

"Context switch":
reload the page map!

We only have
change Context#
and PDIR

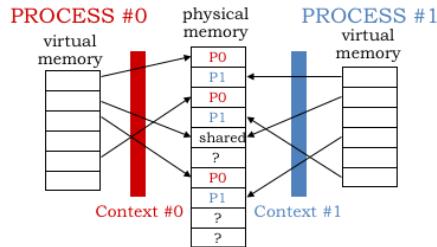


The page map creates the context needed to translate virtual addresses to physical addresses. In a computer system that's working on multiple tasks at the same time, we would like to support multiple contexts and be able to quickly switch from one context to another.

Multiple contexts would allow us to share physical memory between multiple programs. Each program would have an independent virtual address space, e.g., two programs could both access virtual address 0 as the address of their first instruction and would end up accessing different physical locations in main memory. When switching between programs, we'd perform a "context switch" to move to the appropriate MMU context.

The ability to share the CPU between many programs seems like a great idea! Let's figure out the details of how that might work...

Building a Virtual Machine (VM)



Goal: give each program its own “VIRTUAL MACHINE”; programs don’t “know” about each other...

- New abstraction: a **process** which has its own
- machine state: R0, ..., R30
 - context (virtual address space)
 - PC, stack
 - program (w/ shared code)
 - virtual I/O devices

“OS Kernel” is a special, privileged process running in its own context. It manages the execution of other processes and handles real I/O devices, emulating virtual I/O devices for each process.

Let's create a new abstraction called a “process” to capture the notion of a running program. A process encompasses all the resources that would be used when running a program including those of the CPU, the MMU, input/output devices, etc. Each process has a “state” that captures everything we know about its execution.

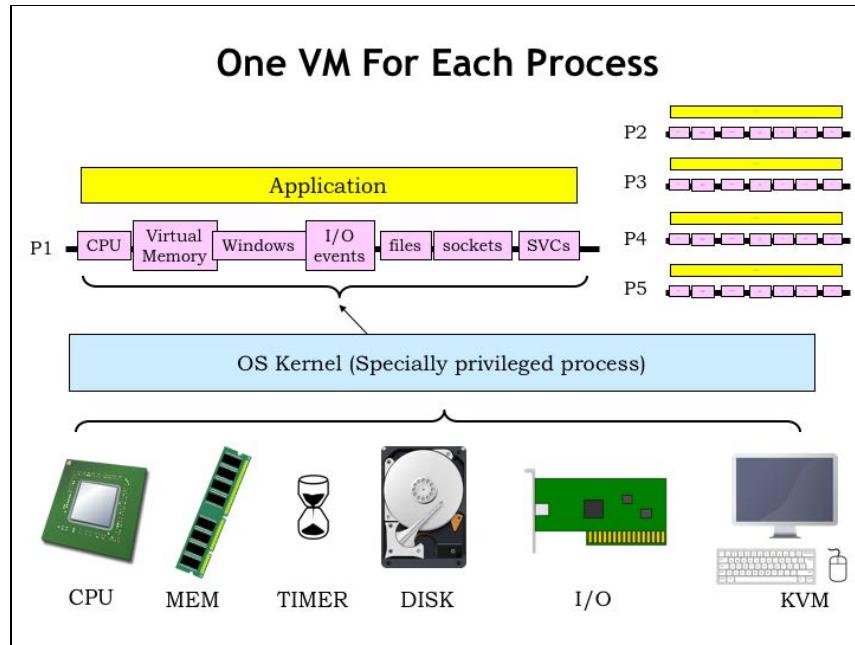
The process state includes

- the hardware state of the CPU, *i.e.*, the values in the registers and program counter.
- the contents of the process' virtual address space, including code, data values, the stack, and data objects dynamically allocated from the heap. Under the management of the MMU, this portion of the state can be resident in main memory or can reside in secondary storage.
- the hardware state of the MMU, which, as we saw earlier, depends on the context-number and page-directory registers. Also included are the pages allocated for the hierarchical page map.
- additional information about the process' input and output activities, such as where it has reached in reading or writing files in the file system, the status and buffers associated with open network connections, pending events from the user interface (*e.g.*, keyboard characters and mouse clicks), and so on.

As we'll see, there is a special, privileged process, called the operating system (OS), running in its own kernel-mode context. The OS manages all the bookkeeping for each process, arranging for the process run periodically. The OS will provide various services to the processes, such as accessing data in files, establishing network connections, managing the window system and user interface, and so on.

To switch from running one user-mode process to another, the OS will need to capture and save the *entire* state of the current user-mode process. Some of it already lives in main memory, so we're all set there. Some of it will be found in various kernel data structures. And some of it we'll need to be able to save and restore from the various hardware resources in the CPU and MMU. In order to successfully implement processes, the OS must be able to make it seem as if each process was running

on its own “virtual machine” that works independently of other virtual machines for other processes. Our goal is to efficiently share one physical machine between all the virtual machines.



Here's a sketch of the organization we're proposing. The resources provided by a physical machine are shown at the bottom of the slide. The CPU and main memory form the computation engine at heart of the system. Connected to the CPU are various peripherals, a collective noun coined from the English word "periphery" that indicates the resources surrounding the CPU.

A timer generates periodic CPU interrupts that can be used to trigger periodic actions.

Secondary storage provides high-capacity non-volatile memories for the system.

Connections to the outside world are important too. Many computers include USB connections for removable devices. And most provide wired or wireless network connections.

And finally there are usually video monitors, keyboards and mice that serve as the user interface. Cameras and microphones are becoming increasing important as the next generation of user interface.

The physical machine is managed by the OS running in the privileged kernel context. The OS handles the low-level interfaces to the peripherals, initializes and manages the MMU contexts, and so on. It's the OS that creates the virtual machine seen by each process.

User-mode programs run directly on the physical processor, but their execution can be interrupted by the timer, giving the OS the opportunity to save away the current process state and move to running the next process. Via the MMU, the OS provides each process with an independent virtual address space that's isolated from the actions of other processes.

The virtual peripherals provided by the OS isolate the process from all the details of sharing resources with other processes. The notion of a window allows the process to access a rectangular array of pixels without having to worry if some pixels in the window are hidden by other windows. Or worrying about how to ensure the mouse cursor always appears on top of whatever is being displayed, and so on. Instead of accessing I/O devices directly, each process has access to a stream of I/O events that are generated when a character is typed, the mouse is clicked, etc. For example, the OS deals with how to determine which typed characters belong to which process. In most window systems, the user clicks on

a window to indicate that the process that owns the window now has the keyboard focus and should receive any subsequent typed characters. And the position of the mouse when clicked might determine which process receives the click. All of which is to say that the details of sharing have been abstracted out of the simple interface provided by the virtual peripherals.

The same is true of accessing files on disk. The OS provides the useful abstraction of having each file appear as a contiguous, growable array of bytes that supports read and write operations. The OS knows how the file is mapped to a pool of sectors on the disk and deals with bad sectors, reducing fragmentation, and improving throughput by doing read look-aheads and write behinds.

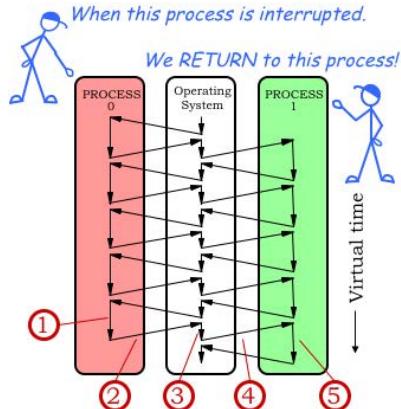
For networks, the OS provides access to an in-order stream of bytes to some remote socket. It implements the appropriate network protocols for packetizing the stream, addressing the packets, and dealing with dropped, damaged, or out-of-order packets.

To configure and control these virtual services, the process communicates with the OS using supervisor calls (SVCs), a type of controlled-access procedure call that invokes code in the OS kernel.

The details of the design and implementation of each virtual service are beyond the scope of this course. If you're interested, a course on operating systems will explore each of these topics in detail.

The OS provides an independent virtual machine for each process, periodically switching from running one process to running the next process.

Processes: Multiplexing the CPU



1. Running in process #0
2. Stop execution of process #0 either because of explicit *yield* or some sort of timer *interrupt*; trap to handler code, saving current PC+4 in XP
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just like return from other trap handlers (ie., use address in XP) but we're returning from a *different* trap than happened in step 2!
5. Running in process #1

Let's follow along as we switch from running process #0 to running process #1.

Initially, the CPU is executing user-mode code in process #0. That execution is interrupted, either by an explicit yield by the program, or, more likely, by a timer interrupt. Either ends up transferring control to OS code running in kernel mode, while saving the current PC+4 value in the XP register. We'll talk about the interrupt mechanism in more detail in just a moment.

The OS saves the state of process #0 in the appropriate table in kernel storage. Then it reloads the state from the kernel table for process #1. Note that the process #1 state was saved when process #1 was interrupted at some earlier point.

The OS then uses a JMP() to resume user-mode execution using the newly restored process #1 state. Execution resumes in process #1 just where it was when interrupted earlier.

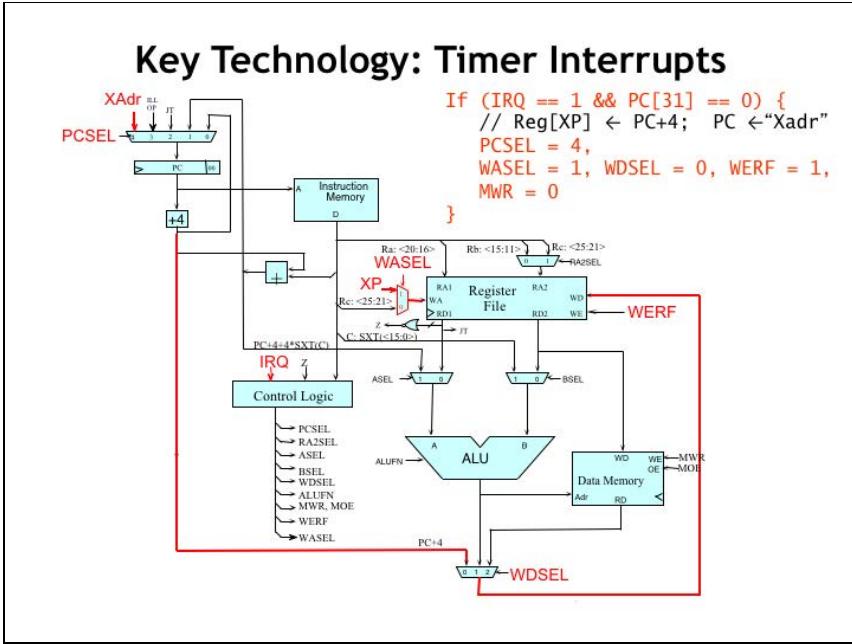
And now we're running the user-mode program in process #1.

We've interrupted one process and resumed execution of another. We'll keep doing this in a round-robin fashion, giving each process a chance to run, before starting another round of execution.

The black arrows give a sense for how time proceeds. For each process, virtual time unfolds as a sequence of executed instructions. Unless it looks at a real-time clock, a process is unaware that occasionally its execution is suspended for a while. The suspension and resumption are completely transparent to a running process.

Of course, from the outside we can see that in real time, the execution path moves from process to process, visiting the OS during switches, producing the dove-tailed execution path we see here.

Time-multiplexing of the CPU is called "timesharing" and we'll examine the implementation in more detail in the following segment.



A key technology for timesharing is the periodic interrupt from the external timer device. Let's remind ourselves how the interrupt hardware in the Beta works.

External devices request an interrupt by asserting the Beta's interrupt request (IRQ) input. If the Beta is running in user mode, *i.e.*, the supervisor bit stored in the PC is 0, asserting IRQ will trigger the following actions on the clock cycle the interrupt is recognized.

The goal is to save the current PC+4 value in the XP register and force the program counter (PC) to a particular kernel-mode instruction, which starts the execution of the interrupt handler code. The normal process of generating control signals based on the current instruction is superseded by forcing particular values for some of the control signals.

PCSEL is set to 4, which selects a specified kernel-mode address as the next value of the program counter. The address chosen depends on the type of external interrupt. In the case of the timer interrupt, the address is 0x80000008. Note that PC[31], the supervisor bit, is being set to 1 and the CPU will be in kernel-mode as it starts executing the code of the interrupt handler.

The WASEL, WDSEL, and WERF control signals are set so that PC+4 is written into the XP register (*i.e.*, R30) in the register file.

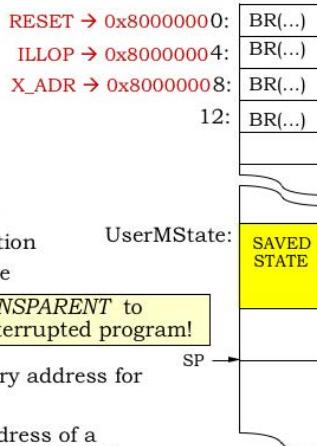
And, finally, MWR is set to 0 to ensure that if we're interrupting a ST instruction that its execution is aborted correctly.

So in the next clock cycle, execution starts with the first instruction of the kernel-mode interrupt handler, which can find the PC+4 of the interrupted instruction in the XP register of the CPU.

Beta Interrupt Handling

Minimal Hardware Implementation:

- Check for Interrupt Requests (IRQs) before each instruction fetch.
- On IRQ j:
 - copy PC+4 into Reg[XP];
 - INSTALL j*4 as new PC.



Handler Coding:

- Save state in "UserMState" structure
- Call C procedure to handle the exception
- re-install saved state from UserMState
- Return to Reg[XP]-4

WHERE to find handlers?

- BETA Scheme: WIRE IN a low-memory address for each exception handler entry point
- Common alternative: WIRE IN the address of a TABLE of handler addresses ("interrupt vectors")

As we can see the interrupt hardware is pretty minimal: it saves the PC+4 of the interrupted user-mode program in the XP register and sets the program counter to some predetermined value that depends on which external interrupt happened.

The remainder of the work to handle the interrupt request is performed in software. The state of the interrupted process, *e.g.*, the values in the CPU registers R0 through R30, is stored in main memory in an OS data structure called UserMState. Then the appropriate handler code, usually a procedure written in C, is invoked to do the heavy lifting. When that procedure returns, the process state is reloaded from UserMState. The OS subtracts 4 from the value in XP, making it point to the interrupted instruction and then resumes user-mode execution with a JMP(XP).

Note that in our simple Beta implementation the first instructions for the various interrupt handlers occupy consecutive locations in main memory. Since interrupt handlers are longer than one instruction, this first instruction is invariably a branch to the actual interrupt code. Here we see that the reset interrupt (asserted when the CPU first starts running) sets the PC to 0, the illegal instruction interrupt sets the PC to 4, the timer interrupt sets the PC to 8, and so on. In all cases, bit 31 of the new PC value is set to 1 so that handlers execute in supervisor or kernel mode, giving them access to the kernel context.

A common alternative is provide a table of new PC values at a known location and have the interrupt hardware access that table to fetch the PC for the appropriate handler routine. This provides the same functionality as our simple Beta implementation.

Since the process state is saved and restored during an interrupt, interrupts are transparent to the running user-mode program. In essence, we borrow a few CPU cycles to deal with the interrupt, then it's back to normal program execution.

Example: Timer Interrupt Handler

Example:

Operating System maintains current time of day (TOD) count.
But...this value must be updated periodically in response to
clock EVENTS, i.e. signal triggered by 60 Hz timer hardware.

Program A (Application)

- Executes instructions of the user program.
- Doesn't want to know about clock hardware, interrupts, etc!!
- Can incorporate TOD into results by "asking" OS.

Clock Handler

- GUTS: Sequence of instructions that increments TOD.
Written in C.
- Entry/Exit sequences save & restore interrupted state, call
the C handler. Written as assembler "stubs".

Here's how the timer interrupt handler would work. Our initial goal is to use the timer interrupt to update a data value in the OS that records the current time of day (TOD). Let's assume the timer interrupt is triggered every 1/60th of a second.

A user-mode program executes normally, not needing to make any special provision to deal with timer interrupts. Periodically the timer interrupts the user-mode program to run the clock interrupt handler code in the OS, then resumes execution of the user-mode program. The program continues execution just as if the interrupt had not occurred. If the program needs access to the TOD, it makes the appropriate service request to the OS.

The clock handler code in the OS starts and ends with a small amount of assembly-language code to save and restore the state. In the middle, the assembly code makes a C procedure call to actually handle the interrupt.

Interrupt Handler Coding

```
long TimeOfDay;
struct MState { int Regs[31]; } UserMState;

/* Executed 60 times/sec */
Clock_Handler(){
    TimeOfDay = TimeOfDay+1;
    if (TimeOfDay % QUANTUM == 0) Scheduler();
}

Clock_h:
    ST(r0, UserMState)           // Save state of
    ST(r1, UserMState+4)         // interrupted
    ...
    ST(r30, UserMState+30*4)
    LD(KStack, SP)               // Use KERNEL SP
    BR(Clock_Handler,lp)         // call handler
    LD(UserMState, r0)           // Restore saved
    LD(UserMState+4, r1)          // state.
    ...
    LD(UserMState+30*4, r30)
    SUBC(XP, 4, XP)              // execute interrupted inst
    JMP(XP)                      // Return to app.
```

Handler
(written in C)

“Interrupt stub”
(written in assy.)

Here's what the handler code might look like. In C, we find the declarations for the TOD data value and the structure, called UserMState, that temporarily holds the saved process state.

There's also the C procedure for incrementing the TOD value.

A timer interrupt executes the BR() instruction at location 8, which branches to the actual interrupt handler code at CLOCK_H. The code first saves the values of all the CPU registers into the UserMState data structure. Note that we don't save the value of R31 since its value is always 0.

After setting up the kernel-mode stack, the assembly-language stub calls the C procedure above to do the hard work. When the procedure returns, the CPU registers are reloaded from the saved process state and the XP register value decremented by 4 so that it will point to the interrupted instruction. Then a JMP(XP) resumes user-mode execution.

Okay, that was simple enough. But what does this all have to do with timesharing - wasn't our goal to arrange to periodically switch which process was running?

Aha! We have code that runs on every timer interrupt, so let's modify it so that every so often we arrange to call the OS' Scheduler() routine. In this example, we'd set the constant QUANTUM to 2 if we wanted to call Scheduler() every second timer interrupt.

The Scheduler() subroutine is where the time sharing magic happens!

Simple Timesharing Scheduler

```
struct MState { int Regs[31]; } UserMState;

struct PCB {           // Process Control Block
    struct MState State;          // Processor state
    struct Context PageMap;       // MMU state for proc
    int DPYNum;                  // Console number (and other I/O state)
} ProcTbl[N];           // one per process

int Cur;                // index of "Active" process

Scheduler() {
    ProcTbl[Cur].State = UserMState; // Save Cur state
    Cur = (Cur+1)%N;               // Incr mod N
    UserMState = ProcTbl[Cur].State; // Install state for next User
    LoadUserContext(ProcTbl[Cur].PageMap); // Install context
}
```

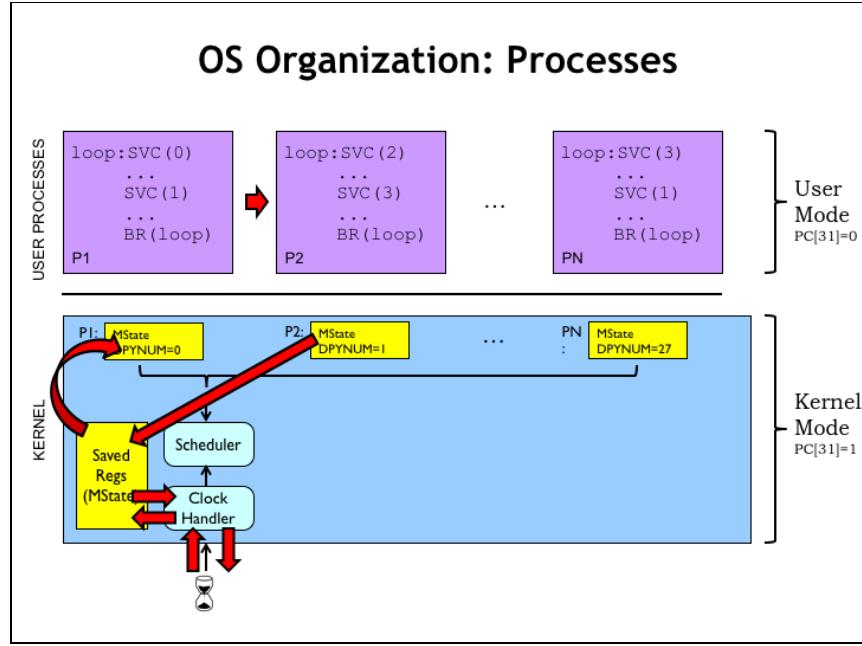
Here we see the UserMState data structure from the previous slide where the user-mode process state is stored during interrupts.

And here's an array of process control block (PCB) data structures, one for each process in the system. The PCB holds the complete state of a process when some other process is currently executing - it's the long-term storage for processor state! As you can see, it includes a copy of MState with the process' register values, the MMU state, and various state associated with the process' input/output activities, represented here by a number indicating which virtual user-interface console is attached to the process.

There are N processes altogether. The variable CUR gives the index into ProcTable for the currently running process.

And here's the surprisingly simple code for implementing timesharing. Whenever the Scheduler() routine is called, it starts by moving the temporary saved state into the PCB for the current process. It then increments CUR to move to the next process, making sure it wraps back around to 0 when we've just finished running the last of the N processes. It then loads/reloads the temporary state from the PCB of the new process and sets up the MMU appropriately.

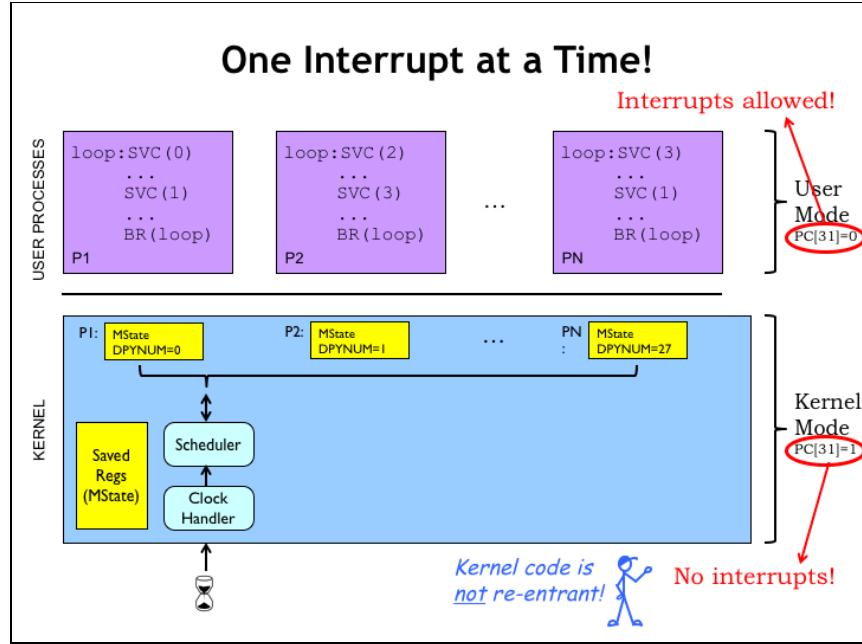
At this point Scheduler() returns and the clock interrupt handler reloads the CPU registers from the updated temporary saved state and resumes execution. Voila! We're now running a new process...



Let's use this diagram to once again walk through how time sharing works. At the top of the diagram you'll see the code for the user-mode processes, and below the OS code along with its data structures.

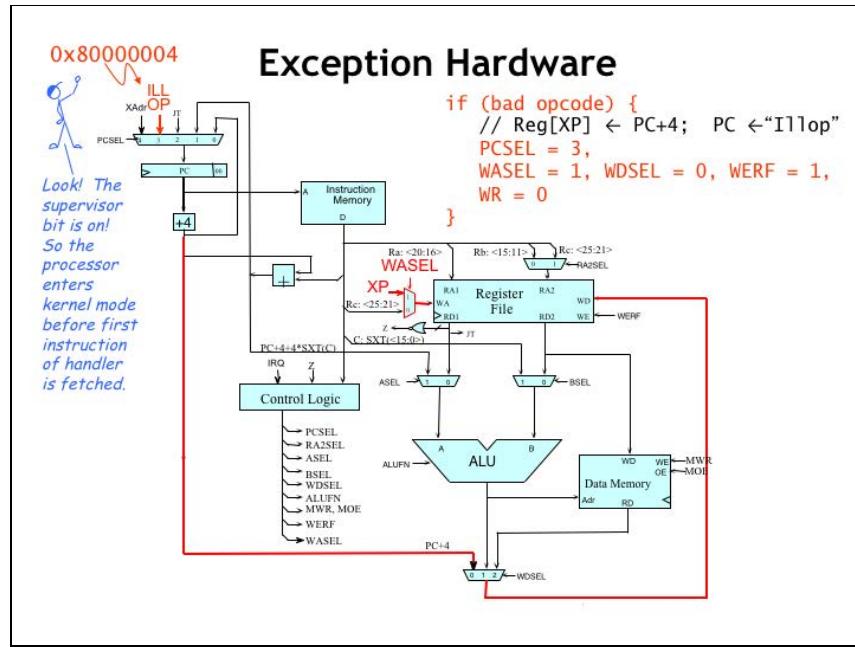
The timer interrupts the currently running user-mode program and starts execution of the OS' clock handler code. The first thing the handler does is save all the registers into the UserMState data structure.

If the Scheduler() routine is called, it moves the temporarily saved state into the PCB, which provides the long-term storage for a process' state. Next Scheduler() copies the saved state for the next process into the temporary holding area. Then the clock handler reloads the updated state into the CPU registers and resumes execution, this time running code in the new process.



While we're looking at the OS, note that since its code runs with the supervisor mode bit set to 1, interrupts are disabled while in the OS. This prevents the awkward problem of getting a second interrupt while still in the middle of handling a first interrupt, a situation that might accidentally overwrite the state in UserMState. But that means one has to be very careful when writing OS code. Any sort of infinite loop can never be interrupted. You may have experienced this when your machine appears to freeze, accepting no inputs and just sitting there like a lump. At this point, your only choice is to power-cycle the hardware (the ultimate interrupt!) and start afresh.

Interrupts are allowed during execution of user-mode programs, so if they run amok and need to be interrupted, that's always possible since the OS is still responding to, say, keyboard interrupts. Every OS has a magic combination of keystrokes that is guaranteed to suspend execution of the current process, sometimes arranging to make copy of the process state for later debugging. Very handy!



Another service provided by operating system is dealing properly with the attempt to execute instructions with “illegal” opcodes. Illegal is quotes because that just means opcodes whose operations aren’t implemented directly by the hardware. As we’ll see, it’s possible extend the functionality of the hardware via software emulation.

The action of the CPU upon encountering an illegal instruction (sometimes referred to as an unimplemented user operation or UUO) is very similar to how it processes interrupts. Think of illegal instructions as an interrupt caused directly by the CPU! As for interrupts, the execution of the current instruction is suspended and the control signals are set to values to capture PC+4 in the XP register and set the PC to, in this case, 0x80000004. Note that bit 31 of the new PC, aka the supervisor bit, is set to 1, meaning that the OS handler will have access to the kernel-mode context.

Exception Handling

```
// hardware interrupt vectors are in low memory
· = 0
    BR(I_Reset) // when Beta first starts
    BR(I_Illop) // on Illegal Instruction (eg SVC)
    BR(I_Clk) // on timer interrupt
    BR(I_Kbd) // on keyboard interrupt, use RDCHAR() to get character
    BR(I_Mouse) // on mouse interrupt, use CLICK() to get coords

// start of kernel-mode storage

KStack:
    LONG(.+4) // Pointer to ...
    STORAGE(256) // ... the kernel stack.

// Here's the SAVED STATE of the interrupted user-mode process
// filled by interrupt handlers
UserMState:
    STORAGE(32) // R0-R30... (PC is in XP/R30!)

N = 16           // max number of processes
Cur:
    LONG(0) // index (0 to N-1) into ProcTbl for current process
ProcTbl:
    STORAGE(N*PCB_Size) // PCB_Size = # bytes to hold complete state
```



This is where the HW sets the PC during an illegal opcode exception

Here's some code similar to that found in the Tiny Operating System (TinyOS), which you'll be experimenting with in the final lab assignment. Let's do a quick walk-through of the code executed when an illegal instruction is executed. Starting at location 0, we see the branches to the handlers for the various interrupts and exceptions. In the case of an illegal instruction, the BR(I_Illop) in location 4 will be executed.

Immediately following is where the OS data structures are allocated. This includes space for the OS stack, UserMState where user-mode register values are stored during interrupts, and the process table, providing long-term storage for the complete state of each process while another process is executing.

Code is from beta.uasm

Useful Macros

```
// Macro to extract and right-adjust a bit field from RA, and leave it
// in RB.  The bit field M:N, where M >= N.
.macro extract_field (RA, M, N, RB) {
    SHLC(RA, 31-M, RB)      // Shift left, to mask out high bits
    SHRC(RB, 31-(M-N), RB)  // Shift right, to mask out low bits.
}

.macro save_all_regs(WHERE) save_all_regs(WHERE, r31)
.macro save_all_regs(WHERE, base_reg) {
    ST(r0, WHERE, base_reg)
    ...
    ST(r30, WHERE+120, base_reg)
}

.macro restore_all_regs(WHERE) restore_all_regs(WHERE, r31)
.macro restore_all_regs(WHERE, base_reg) {
    LD(base_reg, WHERE, r0)
    ...
    LD(base_reg, WHERE+120, r30)
```

Macros can be used
like an in-lined
procedure call



When writing in assembly language, it's convenient to define macros for operations that are used repeatedly. We can use a macro call whenever we want to perform the action and the assembler will insert the body of the macro in place of the macro call, performing a lexical substitution of the macro's arguments.

Here's a macro for a two-instruction sequence that extracts a particular field of bits from a 32-bit value. M is the bit number of the left-most bit, N is bit number of the right-most bit. Bits are numbered 0 through 31, where bit 31 is the most-significant bit, *i.e.*, the one at the left end of the 32-bit binary value.

And here are some macros that expand into instruction sequences that save and restore the CPU registers to or from the UserMState temporary storage area.

Illegal Handler

```
/// Handler for Illegal Instructions
I_Illop:
    save_all_regs(UserMState) // Save the machine state.
    LD(KStack, SP)           // Install kernel stack pointer.

    ADDC(XP, -4, r0)         // Fetch the illegal instruction
    BR(ReadUserMem, LP)      // interpret addr in user context

    SHRC(r0, 26, r1)         // Extract the 6-bit OPCODE
    MULC(r1, 4, r1)          // Make it a WORD (4-byte) index
    LD(r1, UUOTb1, r1)       // Fetch UUOTb1[OPCODE]
    JMP(r1)                  // and dispatch to the UUO handler.

.macro UUO(ADR) LONG(ADR+0x80000000) // Auxiliary Macros
.macro BAD() UUO(UUOError)

UUOTb1: BAD()    UUO(SVC_UUO)    UUO(swapreg) BAD()
          BAD()    BAD()          BAD()        BAD()
          BAD()    BAD()          BAD()        BAD()
... more table follows ...
```

So kernel code can make subroutine calls!

supervisor bit...

This is a 64-entry dispatch table. Each entry is an address of a "handler"

With those macros in hand, let's see how illegal opcodes are handled...

Like all interrupt handlers, the first action is to save the user-mode registers in the temporary storage area and initialize the OS stack.

Next, we fetch the illegal instruction from the user-mode program. Note that the saved PC+4 value is a virtual address in the context of the interrupted program. So we'll need to use the MMU routines to compute the correct physical address - more about this on the next slide.

Then we'll use the opcode of the illegal instruction as an index into a table of subroutine addresses, one for each of the 64 possible opcodes. Once we have the address of the handler for this particular illegal opcode, we JMP there to deal with the situation.

Selecting a destination from a table of addresses is called "dispatching" and the table is called the "dispatch table". If the dispatch table contains many different entries, dispatching is much more efficient in time and space than a long series of compares and branches. In this case, the table is indicating that the handler for most illegal opcodes is the UUOError routine, so it might have smaller and faster simply to test for the two illegal opcodes the OS is going to emulate.

Illegal opcode 1 will be used to implement procedure calls from user-mode to the OS, which we call supervisor calls. More on this in the next segment.

As an example of having the OS emulate an instruction, we'll use illegal opcode 2 as the opcode for the SWAPREG instruction, which we'll discuss now.

Accessing User Locations

We'll need to use the VtoP routine from the previous lecture to translate a user-mode virtual address into the appropriate physical address. VtoP will have to be modified slightly to find the correct context now that we have multiple processes.

```
// expects user-mode virtual address in R0,  
// returns contents of that location in user's virtual memory  
ReadUserMem:  
    PUSH(LP)           // save registers we use below  
  
    PUSH(r0)          // returns physical address in R0  
    BR(VtoP,LP)  
    DEALLOCATE(1)  
    LD(r0,0,r0)       // load the contents  
  
    POP(LP)           // restore regs, return to caller  
    JMP(LP)
```

But first just a quick look at how the OS converts user-mode virtual addresses into physical addresses it can use. We'll build on the MMU VtoP procedure, described in the previous lecture. This procedure expects as its arguments the virtual page number and offset fields of the virtual address, so, following our convention for passing arguments to C procedures, these are pushed onto the stack in reverse order. The corresponding physical address is returned in R0.

We can then use the calculated physical address to read the desired location from physical memory.

Handler for Actual Illops

```
// Here's the handler for truly unused opcodes (not SVCs or swapreg):  
// Illegal instruction is in R0, it's address is Reg[XP]-4  
UUOError:  
    CALL(KWrMsg)           // Type out an error msg,  
    .text "Illegal instruction"  
  
    ADDC(XP, -4, r0)      // Fetch the illegal instruction  
    BR(ReadUserMem,LP)    // interpret addr in user context  
    CALL(KHexPrt)  
    CALL(KWrMsg)  
    .text " at location 0x"  
  
    MOVE(xp,r0)  
    CALL(KHexPrt)  
    CALL(KWrMsg)  
    .text "! ....."  
  
    HALT()                // Then crash system.
```



These kernel utility routines (Kxxx) don't follow our usual calling convention - they take their args in registers or from words immediately following the procedure call! They adjust LP to skip past any args before returning.

Okay, back to dealing with illegal opcodes. Here's the handler for opcodes that are truly illegal. In this case the OS uses various kernel routines to print out a helpful error message on the user's console, then crashes the system! You may have seen these "blue screens of death" if you run the Windows operating system, full of cryptic hex numbers.

Actually, this wouldn't be the best approach to handling an illegal opcode in a user's program. In a real operating system, it would be better to save the state of the process in a special debugging file historically referred to as a "core dump" and then terminate this particular process, perhaps printing a short message on the user's console to let them know what happened. Then later the user could start a debugging program to examine the dump file to see where their bug is.

Emulated Instruction: swapreg(Ra,Rc)

```
// swapreg(RA,RC) swaps the contents of the two named registers.  
.macro swapreg(RA,RC) betaopc(0x02,RA,0,RC)  
  
// swapreg instruction is in R0, it's address is Reg[XP]-4  
swapreg:  
    extract_field(r0, 25, 21, r1) // extract rc field  
    MULC(r1, 4, r1)           // convert to byte offset into regs array  
    extract_field(r0, 20, 16, r2) // extract ra field  
    MULC(r2, 4, r2)           // convert to byte offset into regs array  
    LD(r1, UserMState, r3) // r3 <- regs[rc]  
    LD(r2, UserMState, r4) // r4 <- regs[ra]  
    ST(r4, UserMState, r1) // regs[rc] <- old regs[ra]  
    ST(r3, UserMState, r2) // regs[ra] <- old regs[rc]  
  
// all done! Resume execution of user-mode program  
BR(I_Rtn)                  // defined in the next section!
```

Finally, here's the handler that will emulate the actions of the SWAPREG instruction, after which program execution will resume as if the instruction had been implemented in hardware. SWAPREG is an instruction that swaps the values in the two specified registers.

To define a new instruction, we'd first have to let the assembler know to convert the swapreg(ra,rc) assembly language statement into binary. In this case we'll use a binary format similar to the ADDC instruction, but setting the unused literal field to 0. The encoding for the RA and RC registers occur in their usual fields and the opcode field is set to 2.

Emulation is surprisingly simple. First we extract the RA and RC fields from the binary for the swapreg instruction and convert those values into the appropriate byte offsets for accessing the temporary array of saved register values.

Then we use RA and RC offsets to access the user-mode register values that have been saved in UserMState. We'll make the appropriate interchange, leaving the updated register values in UserMState, where they'll be loaded into the CPU registers upon returning from the illegal instruction interrupt handler.

Finally, we'll branch to the kernel code that restores the process state and resumes execution. We'll see this code in the next segment.

Communicating with the OS

User-mode programs need to communicate with OS code:

- Access virtual I/O devices
- Communicate with other processes
- ...



*But if OS Kernel is in
another context (ie, not in
user-mode address space)
how do we get to it?*

Solution:

Abstraction: a supervisor call (SVC) with args in registers –
result in R0 or maybe user-mode memory

Implementation:

use *illegal instructions* to cause an exception --
OS code will recognize these particular illegal
instructions as a user-mode SVCs

*Okay...
show me
how it
works!*

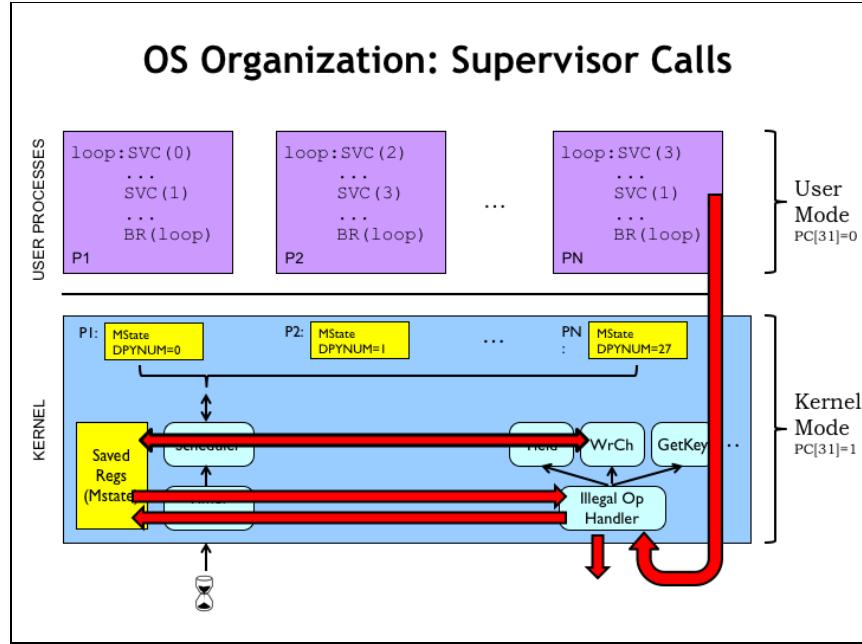


User-mode programs need to communicate with the OS to request service or get access to useful OS data like the time of day. But if they're running in a different MMU context than the OS, they don't have direct access to OS code and data. And that might be bad idea in any case: the OS is usually responsible for implementing security and access policies and other users of the system would be upset if any random user program could circumvent those protections.

What's needed is the ability for user-mode programs to call OS code at specific entry points, using registers or the user-mode virtual memory to send or receive information. We'd use these "supervisor calls" to access a well-documented and secure OS application programming interface (API). An example of such an interface is POSIX, a standard interface implemented by many Unix-like operating systems.

As it turns out, we have a way of transferring control from a user-mode program to a specific OS handler - just execute an illegal instruction! We'll adopt the convention of using illegal instructions with an opcode field of 1 to serve as supervisor calls. The low order bits of these SVC instructions will contain an index indicating which SVC service we're trying to access.

Let's see how this would work.



Here's our user-mode/kernel-mode diagram again. Note that the user-mode programs contain supervisor calls with different indices, which when executed are intended to serve as requests for different OS services.

When an SVC instruction is executed, the hardware detects the opcode field of 1 as an illegal instruction and triggers an exception that runs the OS IllOp handler, as we saw in the previous segment.

The handler saves the process state in the temporary storage area, then dispatches to the appropriate handler based on the opcode field. This handler can access the user's registers in the temporary storage area, or using the appropriate OS subroutines can access the contents of any user-mode virtual address. If information is to be returned to the user, the return values can be stored in the temporary storage area, overwriting, say, the saved contents of the user's R0 register. Then, when the handler completes, the potentially-updated saved register values are reloaded into the CPU registers and execution of the user-mode program resumes at the instruction following the supervisor call.

Handler for SVCs

SVC Instruction format



```
// Sub-handler for SVCs, called from I_Il10p on SVC opcode:  
// SVC instruction is in R0, it's address is Reg[XP]-4  
SVC_UU0:  
    ANDC(r0,0x7,r1)      // Pick out low bits,  
    SHLC(r1,2,r1)        // make a word index,  
    LD(r1,SVCTb1,r1)     // and fetch the table entry.  
  
SVCTb1: UU0(HaltH)      // SVC(0): User-mode HALT instruction  
         UU0(WrMsgH)  
         UU0(WrChH)  
         UU0(GetKeyH)  
         UU0(HexPrth)  
         UU0(WaitH)       // SVC(1): Write message  
         UU0(SignalH)     // SVC(2): Write Character  
         UU0(YieldH)      // SVC(3): Get Key  
                     // SVC(4): Hex Print  
                     // SVC(5): Wait(S), S in R3  
                     // SVC(6): Signal(S), S in R3  
                     // SVC(7): Yield()  
  
Another  
dispatch  
table!
```



Earlier we saw how the illegal instruction handler uses a dispatch table to choose the appropriate sub-handler depending on the opcode field of the illegal instruction.

In this slide we see the sub-handler for SVC instructions, *i.e.*, those with an opcode field of 1. This code uses the low-order bits of the instruction to access another dispatch table to select the appropriate code for each of the eight possible SVCs.

Our Tiny OS only has a meagre selection of simple services. A real OS would have SVCs for accessing files, dealing with network connections, managing virtual memory, spawning new processes, and so on.

Returning to User-mode

```
// Alternate return from interrupt handler which BACKS UP PC,
// and calls the scheduler prior to returning. This causes
// the trapped SVC to be re-executed when the process is
// eventually rescheduled...

HaltH:
I_Wait:
    LD(UserMState+(4*XP), r0) // Grab XP from saved MState,
    SUBC(r0, 4, r0)           // back it up to point to
    ST(r0, UserMState+(4*XP)) // SVC instruction

YieldH:
    CALL(Scheduler)          // Switch current process,
    BR(I_Rtn)

// Here's the common exit sequence from Kernel interrupt handlers:
// Restore registers, and jump back to the interrupted user-mode
// program.

I_Rtn:
    restore_all_regs(UserMState)
    JMP(XP)                  // Good place for debugging breakpoint!
```

Here's the code for resuming execution of the user-mode process when the SVC handler is done: simply restore the saved values for the registers and JMP to resume execution at the instruction following the SVC instruction.

There are times when for some reason the SVC request cannot not be completed and the request should be retried in the future. For example, the ReadCh SVC returns the next character typed by the user, but if no character has yet been typed, the OS cannot complete the request at this time. In this case, the SVC handler should branch to I_Wait, which arranges for the SVC instruction to be re-executed next time this process runs and then calls Scheduler() to run the next process. This gives all the other processes a chance to run before the SVC is tried again, hopefully this time successfully.

You can see that this code also serves as the implementation for two different SVCS! A process can give up the remainder of its current execution time slice by calling the Yield() SVC. This simply causes the OS to call Scheduler(), suspending execution of the current process until its next turn in the round-robin scheduling process.

And to stop execution, a process can call the Halt() SVC. Looking at the implementation, we can see that "halt" is a bit of misnomer. What really happens is that the system arranges to re-execute the Halt() SVC each time the process is scheduled, which then causes the OS to schedule the next process for execution. The process appears to halt since the instruction following the Halt() SVC is never executed.

Adding New SVCs

```
.macro GetTOD() SVC(8)      // return time of today in R0
.macro SetTOD() SVC(9)      // set time of day to value in R0

// Sub-handler for SVCs, called from I_ILLOp on SVC opcode:
// SVC instruction is in R0, it's address is Reg[XP]-4
SVC_UU0:
    ANDCC(r0,0xF,r1)        // Pick out low bits,
    SHLC(r1,2,r1)           // make a word index,
    LD(r1,SVCTb1,r1)         // and fetch the table entry.
    JMP(r1)

SVCTb1: UU0(HaltH)          // SVC(0): User-mode HALT instruction
UU0(WrMsgH)           // SVC(1): Write message
UU0(WrChH)            // SVC(2): Write Character
UU0(GetKeyH)          // SVC(3): Get Key
UU0(HexPrtH)          // SVC(4): Hex Print
UU0(WaitH)             // SVC(5): Wait(S), S in R3
UU0(SignalH)           // SVC(6): Signal(S), S in R3
UU0(YieldH)            // SVC(7): Yield()
UU0(GetTOD)            // SVC(8): return time of day
UU0(SetTOD)            // SVC(9): set time of day
```

Adding new SVC handlers is straightforward.

First we need to define new SVC macros for use in user-mode programs. In this example, we're defining SVCs for getting and setting the time of day.

Since these are the eighth and ninth SVCs, we need to make a small adjustment to the SVC dispatch code and then add the appropriate entries to the end of the dispatch table.

New SVC Handlers

```
// return the current time of day in R0
GetTOD:
    LD(TimeOfDay,r0)      // load OS time of day value
    ST(r0,UserMState+4*0)  // store into user's R0
    BR(I_Rtn)              // resume execution with updated R0 value

// set the current time of day from the value in user's R0
SetTOD:
    LD(UserMState+4*0,r0)  // load value in (saved) user's R0
    ST(r0,TimeOfDay)        // store to OS time of day value
    BR(I_Rtn)              // resume execution
```



SVCs provide controlled access to OS services and data values and offer "atomic" (uninterrupted) execution of instruction sequences.

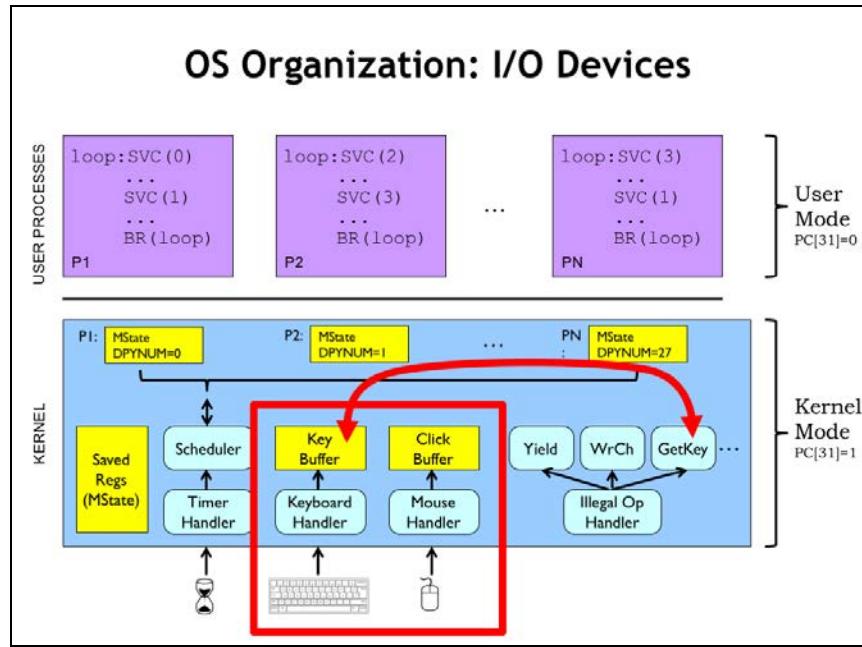
The code for the new handlers is equally straightforward. The handler can access the value of the program's R0 by looking at the correct entry in the UserMState temporary holding area. It just takes a few instructions to implement the desired operations.

The SVC mechanism provides controlled access to OS services and data. As we'll see in a few lectures, it'll be useful that SVC handlers can't be interrupted since they are running in supervisor mode where interrupts are disabled. So, for example, if we need to increment a value in main memory, using a LD/ADDC/ST sequence, but we want to ensure no other process execution intervenes between the LD and the ST, we can encapsulate the required functionality as an SVC, which is guaranteed to be uninterrupted.

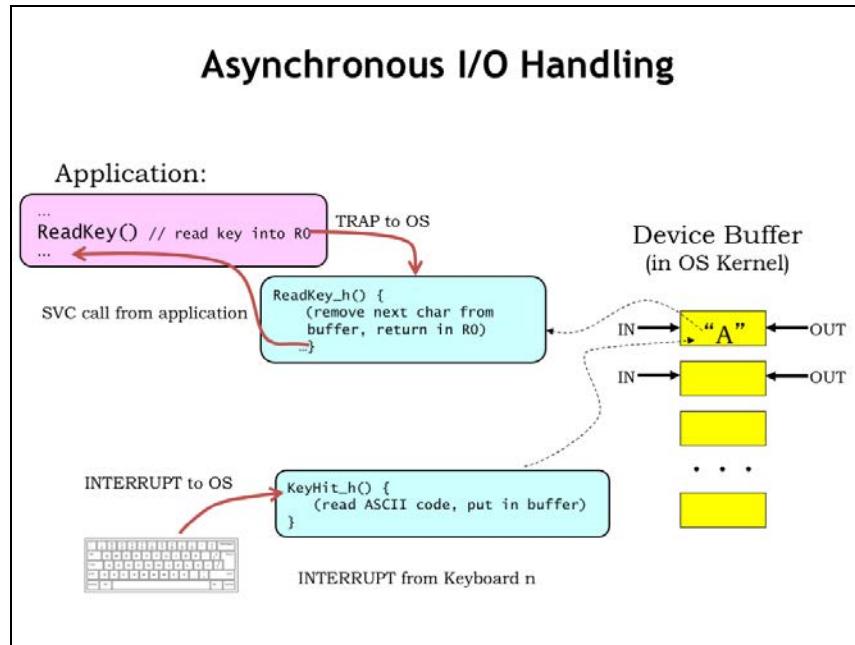
We've made an excellent start at exploring the implementation of a simple time-shared operating system. We'll continue the exploration in the next lecture when we see how the OS deals with external input/output devices.

18: Devices and Interrupts

1. OS Organization: I/O Devices
2. Asynchronous I/O Handling
3. Interrupt-based Asynch I/O
4. ReadKey SVC: Attempt #1
5. ReadKey SVC: Attempt #2
6. ReadKey SVC: Attempt #3
7. Sophisticated Scheduling
8. ReadKey SVC: Attempt #4
9. Example: Match Handler to OS
10. Which Handler and OS? #1
11. Which Handler and OS? #2
12. Which Handler and OS? #3
13. The Need for “Real Time”
14. Interrupt Latency
15. Sources of Interrupt Latency
16. Scheduling of Multiple Devices
17. Weak (Non-preemptive) Priorities
18. Setting Priorities
19. The Need for Preemption
20. Strong Priority Implementation
21. Recurring Interrupts
22. Interrupt Load
23. Mr. Blue Visits the ISS
24. Mr. Blue Visits the ISS (cont'd.)
25. Summary



Let's turn our attention to how the operating system (OS) deals with input/output devices. There are actually two parts to the discussion. First, we'll talk about how the OS interacts with the devices themselves. This will involve a combination of interrupt handlers and kernel buffers. Then we'll discuss how supervisor calls access the kernel buffers in response to requests from user-mode processes. As we'll see, this can get a bit tricky when the OS cannot complete the request at the time the SVC was executed.



Here's the plan! When the user types a key on the keyboard, the keyboard triggers an interrupt request to the CPU. The interrupt suspends execution of the currently-running process and executes the handler whose job it is to deal with this particular I/O event.

In this case, the keyboard handler reads the character from the keyboard and saves it in a kernel buffer associated with the process that has been chosen to receive incoming keystrokes. In the language of OSes, we'd say that process has the keyboard focus. This transfer takes just a handful of instructions and when the handler exits, we resume running the interrupted process.

Assuming the interrupt request is serviced promptly, the CPU can easily keep up with the arrival of typed characters. Humans are pretty slow compared to the rate of executing instructions! But the buffer in the kernel can hold only so many characters before it fills up. What happens then?

Well, there are a couple of choices. Overwriting characters received earlier doesn't make much sense: why keep later characters if the earlier ones have been discarded. Better that the CPU discard any characters received after the buffer was full, but it should give some indication that it's doing so. And, in fact, many systems beep at the user to signal that the character they've just typed is being ignored.

At some later time, a user-mode program executes a `ReadKey()` supervisor call, requesting that the OS return the next character in R0. In the OS, the `ReadKey` SVC handler grabs the next character from the buffer, places it in the user's R0, and resumes execution at the instruction following the SVC.

There are few tricky bits we need to figure out. The `ReadKey()` SVC is what we call a "blocking I/O" request, *i.e.*, the program assumes that when the SVC returns, the next character is in R0. If there isn't (yet) a character to be returned, execution should be "blocked", *i.e.*, suspended, until such time that a character is available.

Many OSes also provide for non-blocking I/O requests, which always return immediately with both a status flag and a result. The program can check the status flag to see if there was a character and do the right thing if there wasn't, *e.g.*, reissue the request at a later time.

Interrupt-based Asynch I/O

OPERATION: NO attention to Keyboard during normal operation

- on key strike: hardware asserts IRQ to request interrupt
- USER program interrupted, PC+4 of interrupted inst. saved in XP
- state of USER program saved on KERNEL stack;
- Keyboard handler invoked, runs to completion;
- state of USER program restored; program resumes.

TRANSPARENT to USER program.

Keyboard Interrupt Handler (in O.S. KERNEL):

Assume each keyboard has an associated buffer


```
struct Device {  
    char Flag, Data;  
} Keyboard;  
  
KeyHit_h() {  
    Buffer[inptr] = Keyboard.Data;  
    inptr = (inptr + 1) % BUFSIZE;  
}
```

Note that the user-mode program didn't have any direct interaction with the keyboard, *i.e.*, it's not constantly polling the device to see if there's a keystroke to be processed. Instead, we're using an "event-driven" approach, where the device signals the OS, via an interrupt, when it needs attention.

This is an elegant separation of responsibilities. Imagine how cumbersome it would be if every program had to check constantly to see if there were pending I/O operations. Our event-driven organization provides for on-demand servicing of devices, but doesn't devote CPU resources to the I/O subsystem until there's actually work to be done. The interrupt-driven OS interactions with I/O devices are completely transparent to user programs.

Here's sketch of what the OS keyboard handler code might actually look like. Depending on the hardware, the CPU might access device status and data using special I/O instructions in the ISA. For example, in the simulated Beta used for lab assignments, there's a RDCHAR() instruction for reading keyboard characters and a CLICK() instruction for reading the coordinates of a mouse click.

Another common approach is to use "memory-mapped I/O", where a portion of the kernel address space is devoted to servicing I/O devices. In this scheme, ordinary LD and ST store instructions are used to access specific addresses, which the CPU recognizes as accesses to the keyboard or mouse device interfaces. This is the scheme shown in the code here. The C data structure represents the two I/O locations devoted to the keyboard: one for status and one for the actual keyboard data.

The keyboard interrupt handler reads the keystroke data from the keyboard and places the character into the next location in the circular character buffer in the kernel.

In real life keyboard processing is usually a bit more complicated. What one actually reads from a keyboard is a key number and a flag indicating whether the event is a key press or a key release. Knowing the keyboard layout, the OS translates the key number into the appropriate ASCII character, dealing with complications like holding down the shift key or control key to indicate a capital character or a control character. And certain combination of keystrokes, *e.g.*, CTRL-ALT-DEL on a Windows system, are interpreted as special user commands to start running particular applications like the Task

Manager. Many OSes let the user specify whether they want “raw” keyboard input (*i.e.*, the key numbers and status) or “digested” input (*i.e.*, ASCII characters).

Whew! Who knew that processing keystrokes could be so complicated!

Next, we’ll figure out how to code the associated supervisor call that lets user programs read characters.

.ReadKey SVC: Attempt #1

SVC recap: SVC, encoded as illegal instruction, causes an exception. OS notices special SVC opcode, dispatches to appropriate sub-handler based on index in low-bits of SVC inst.

First draft of a ReadKey SVC handler (supporting a *virtual* keyboard): returns next keystroke on a user's keyboard in response to the SVC request:

```
.ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum)) {
        /* busy wait loop */
    }
    UserMState.Reg[0] = ReadInputBuffer(kbdnum);
}
```



Problem: Can't interrupt code running in the supervisor mode... so the buffer never gets filled.

When a user-mode program wants to read a typed character it executes a ReadKey() SVC. The binary representation of the SVC has an illegal value in the opcode field, so the CPU hardware causes an exception, which starts executing the illegal opcode handler in the OS. The OS handler recognizes the illegal opcode value as being an SVC and uses the low-order bits of the SVC instruction to determine which sub-handler to call.

Here's our first draft for the ReadKey sub-handler, this time written in C. The handler starts by looking at the process table entry for the current process to determine which keyboard buffer holds the characters for the process. Let's assume for the moment the buffer is *not* empty and skip to the last line, which reads the character from the buffer and uses it to replace the saved value for the user's R0 in the array holding the saved register values. When the handler exits, the OS will reload the saved registers and resume execution of the user-mode program with the just-read character in R0.

Now let's figure what to do when the keyboard buffer is empty. The code shown here simply loops until the buffer is no longer empty. The theory is that eventually the user will type a character, causing an interrupt, which will run the keyboard interrupt handler discussed in the previous section, which will store a new character into the buffer.

This all sounds good until we remember that the SVC handler is running with the supervisor bit (PC[31]) set to 1, disabling interrupts. Oops! Since the keyboard interrupt will never happen, the while loop shown here is actually an infinite loop. So if the user-mode program tries to read a character from an empty buffer, the system will appear to hang, not responding to any external inputs since interrupts are disabled. Time to reach for the power switch :)

.ReadKey SVC: Attempt #2

A BETTER keyboard SVC handler:

```
.ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        /* busy wait loop */
        UserMState.Reg[XP] = UserMState.Reg[XP]-4;
    } else
        UserMState.Reg[0] = ReadInputBuffer(kbdnum);
}
```

That's a funny way to write a loop



This one actually works!

Problem: The process just wastes its time-slice waiting for someone to hit a key...

We'll fix the looping problem by adding code to subtract 4 from the saved value of the XP register before returning. How does this fix the problem?

Recall that when the SVC illegal instruction exception happened, the CPU stored the PC+4 value of the illegal instruction in the user's XP register. When the handler exits, the OS will resume execution of the user-mode program by reloading the registers and then executing a JMP(XP), which would normally then execute the instruction *following* the SVC instruction. By subtracting 4 from the saved XP value, it will be the SVC itself that gets re-executed.

That, of course, means we'll go through the same set of steps again, repeating the cycle until the keyboard buffer is no longer empty. It's just a more complicated loop! But with a crucial difference: one of the instructions — the ReadKey() SVC — is executed in user-mode with PC[31] = 0. So during that cycle, if there's a pending interrupt from the keyboard, the device interrupt will supersede the execution of the ReadKey() and the keyboard buffer will be filled. When the keyboard interrupt handler finishes, the ReadKey() SVC will be executed again, this time finding that the buffer is no longer empty. Yay!

So this version of the handler actually works, with one small caveat. If the buffer is empty, the user-mode program will continually re-execute the complicated user-mode/kernel-mode loop until the timer interrupt eventually transfers control to the next process. This seems pretty inefficient. Once we've checked and found the buffer is empty, it would be better to give other processes a chance to run before we try again.

.ReadKey SVC: Attempt #3

EVEN BETTER: On I/O wait, YIELD remainder of quantum:

```
.ReadKey_h()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        UserMState.Reg[XP] = UserMState.Reg[XP]-4;
        Scheduler();
    } else
        UserMState.Reg[0] = ReadInputBuffer(kbdnum);
}
```

RESULT: Better CPU utilization!!

Does timesharing cause CPU use to be less efficient?

- COST: Scheduling, context-switching overhead; but
- GAIN: Productive use of idle time of one process by running another.

This problem is easy to fix! We'll just add a call to Scheduler() right after arranging for the ReadKey() SVC to be re-executed. The call to Scheduler() suspends execution of the current process and arranges for the next process to run when the handler exits. Eventually the round-robin scheduling will come back to the current process and the ReadKey() SVC will try again.

With this simple one-line fix the system will spend much less time wasting cycles checking the empty buffer and instead use those cycles to run other, hopefully more productive, processes. The cost is a small delay in restarting the program after a character is typed, but typically the time slices for each process are small enough that one round of process execution happens more quickly than the time between two typed characters, so the extra delay isn't noticeable.

So now we have some insights into one of the traditional arguments against timesharing. The argument goes as follows. Suppose we have 10 processes, each of which takes 1 second to complete its computation. Without timesharing, the first process would be done after 1 second, the second after 2 seconds, and so on. With timesharing using, say, a 1/10 second time slice, all the processes will complete sometime after 10 seconds since there's a little extra time needed for the hundred or so process switches that would happen before completion. So in a timesharing system the time-to-completion for **all** processes is as long the worst-case completion time without time sharing! So why bother with timesharing?

We saw one answer to this question earlier in this slide. If a process can't make productive use of its time slice, it can donate those cycles to completion of some other task. So in a system where most processes are waiting for some sort of I/O, timesharing is actually a great way of spending cycles where they'll do the most good.

If you open the Task Manager or Activity Monitor on the system you're using now, you'll see there are hundreds of processes, almost all of which are in some sort of I/O wait. So timesharing does extract a cost when running compute-intensive computations, but in an actual system where there's a mix of I/O and compute tasks, time sharing is the way to go.

Sophisticated Scheduling

To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- Processes can be in **ACTIVE** or **WAITING** (“sleeping”) states;
- Scheduler cycles among **ACTIVE PROCESSES** only;
- Active process moves to **WAITING** status when it tries to read a character and buffer is empty;
- Waiting processes each contain a code (eg, in PCB) designating what they are waiting for (eg, keyboard N);
- Device interrupts (eg, on keyboard N) move any processes waiting on that device to **ACTIVE** state.

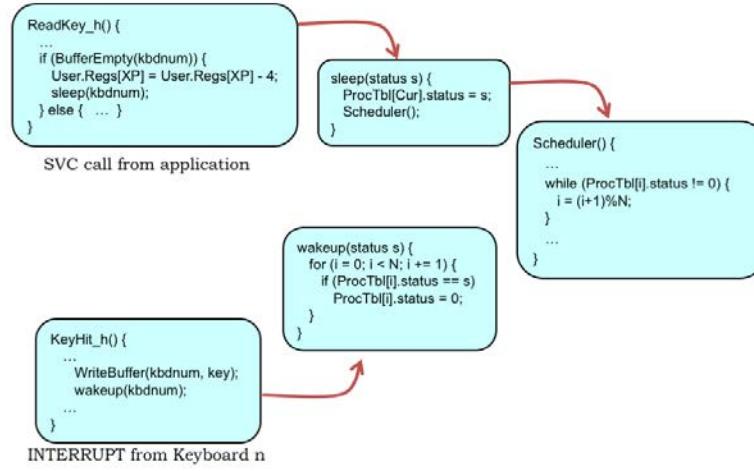
UNIX kernel utilities:

- `sleep(reason)` - Puts CurProc to sleep. “Reason” is an arbitrary binary value giving a condition for reactivation.
- `wakeup(reason)` - Makes active any process in `sleep(reason)`.

We can actually go one step further to ensure we don’t run processes waiting for an I/O event that hasn’t yet happened. We’ll add a status field to the process state indicating whether the process is ACTIVE (*e.g.*, status is 0) or WAITING (*e.g.*, status is non-zero). We’ll use different non-zero values to indicate what event the process is waiting for. Then we’ll change the Scheduler() to only run ACTIVE processes.

To see how this works, it’s easiest to use a concrete example. The UNIX OS has two kernel subroutines: `sleep()` and `wakeup()` both of which require a non-zero argument. The argument will be used as the value of the status field. Let’s see this in action.

.ReadKey SVC: Attempt #4



When the `ReadKey()` SVC detects the buffer is empty, it calls `sleep()` with an argument that uniquely identifies the I/O event it's waiting for, in this case the arrival of a character in a particular buffer. `sleep()` sets the process status to this unique identifier, then calls `Scheduler()`.

`Scheduler()` has been modified to skip over processes with a non-zero status, not giving them a chance to run. Meanwhile, a keyboard interrupt will cause the interrupt handler to add a character to the keyboard buffer and call `wakeup()` to signal any process waiting on that buffer. Watch what happens when the `kbdnum` in the interrupt handler matches the `kbdnum` in the `ReadKey()` handler.

`wakeup()` loops through all processes, looking for ones that are waiting for this particular I/O event. When it finds one, it sets the status for the process to zero, marking it as ACTIVE. The zero status will cause the process to run again next time the `Scheduler()` reaches it in its round-robin search for things to do.

The effect is that once a process goes to `sleep()` WAITING for an event, it's not considered for execution again until the event occurs and `wakeup()` marks the process as ACTIVE. Pretty neat! Another elegant fix to ensure that no CPU cycles are wasted on useless activity. I can remember how impressed I was when I first saw this many years ago in a (very) early version of the UNIX code :)

Example: Match Handler to OS

C ↤ R1

```
ReadCh_h() { // Version R1
    if (BufferEmpty(0))
        UserMState.Rregs[XP] = UserMState.Rregs[XP] - 4;
    else
        UserMState.Rregs[0] = ReadInputBuffer(0);
}
```

B ↤ R2

```
ReadCh_h() { // Version R2
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum));
    UserMState.Rregs[0] = ReadInputBuffer(kbdnum);
}
```

A ↤ R3

```
ReadCh_h() { // Version R3
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        UserMState.Rregs[XP] = UserMState.Rregs[XP] - 4;
        Scheduler();
    } else
        UserMState.Rregs[0] = ReadInputBuffer(kbdnum);
}
```

Always reads from the same buffer

Oops! Infinite loop?

Model A: A timeshared Beta system whose OS kernel is uninterruptable

Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps

Model C: A single-process (not timeshared) system which runs dedicated application code

Here's an old quiz problem we can use to test our understanding of all the factors that went into the final design of our ReadKey() SVC code. We're considering three different versions (R1, R2, and R3) of the ReadKey() SVC code, all variants of the various attempts from the previous section. And there are three types of systems (Models A, B, and C). We've been asked to match the three handlers to the appropriate system.

Looking at R1, we see it's similar to Attempt #2 from the previous section, except it always reads from the same keyboard regardless of the process making the SVC request. That wouldn't make much sense in a timesharing system since a single stream of input characters would be shared across all the processes. So this handler must be intended for the Model C system, which has only a single process.

Looking at R2, we see it's similar to Attempt #1 from the previous section, which had the fatal flaw of a potentially infinite loop if attempting to read from an empty buffer. So this code would only run successfully on the Model B system, which *does* allow device interrupts even when the CPU is running inside an SVC call. So the keyboard interrupt would interrupt the while loop in R2 and the next iteration of the loop would discover that buffer was no longer empty.

By the process of elimination that leaves the R3 handler to be paired with the Model A system. R3 is Attempt #3 from the previous section and is designed for our standard system in which the kernel is uninterruptible.

```

RI
ReadCh_h() { // Version R1
    if (BufferEmpty(0))
        UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
    else
        UserMState.Regs[0] = ReadInputBuffer(0);
}

R2
ReadCh_h() { // Version R2
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum));
    UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}

R3
ReadCh_h() { // Version R3
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        UserMState.Regs[XP] = UserMState.Regs[XP] - 4;
        Scheduler();
    } else
        UserMState.Regs[0] = ReadInputBuffer(kbdnum);
}

```

	A	B	C
R1	X	X	X
R2	X	X	X
R3	X	X	

Model A: A timeshared Beta system whose OS kernel is uninterruptable
Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps
Model C: A single-process (not timeshared) system which runs dedicated application code

Which handler & OS?
 "I get compile-time errors; Scheduler and ProcTbl are undefined!"

The problem goes on to say that a fumble-fingered summer intern has jumbled up the disks containing the handlers and sent an unknown handler version to each user running one of the three model systems. To atone for the mistake, he's been assigned the task of reading various user messages sent after the user has tried the new handler disk on their particular system. Based on the message, he's been asked to identify which handler disk and system the user is using.

The first message says "I get compile-time errors; Scheduler and ProcTbl are undefined."

On the right of the slide we've included a table enumerating all the combinations of handlers and systems, where we've X-ed the matches from the previous slide since they correspond to when the new handler would be the same as the old handler and the user wouldn't be sending a message!

The phrase "Scheduler and ProcTbl are undefined" wouldn't apply to a timesharing system, which includes both symbols. So we can eliminate the first two columns from consideration. And we can also eliminate the second row, since handler R2 doesn't include a call to Scheduler.

So this message came from a user trying to run handler R3 on a Model C system. Since Model C doesn't support timesharing, it would have neither Scheduler nor ProcTbl as part the OS code.

R1	<pre> ReadCh_h() { // Version R1 if (BufferEmpty(0)) UserMState.Reg[XP] = UserMState.Reg[XP] - 4; else UserMState.Reg[0] = ReadInputBuffer(0); } ReadCh_h() { // Version R2 int kbdnum = ProcTbl[Cur].DPYNum; while (BufferEmpty(kbdnum)); UserMState.Reg[0] = ReadInputBuffer(kbdnum); } ReadCh_h() { // Version R3 int kbdnum = ProcTbl[Cur].DPYNum; if (BufferEmpty(kbdnum)) { UserMState.Reg[XP] = UserMState.Reg[XP] - 4; Scheduler(); } else UserMState.Reg[0] = ReadInputBuffer(kbdnum); } </pre>		
R1	A	B	C
R2	X	X	X
R3	X	X	X

Model A: A timeshared Beta system whose OS kernel is uninterruptable
Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps
Model C: A single-process (not timeshared) system which runs dedicated application code

Which handler & OS?

"Hey, now the system always reads everybody's input from keyboard 0.
In addition, it seems to waste a lot more CPU cycles than it used to."

Okay, here's the next message: "Hey, now the system always reads everybody's input from keyboard 0. Besides that, it seems to waste a lot more CPU cycles than it used to."

R1 is the only handler that always reads from keyboard 0, so we can eliminate rows 2 and 3.

So how can we tell if R1 is being run on a Model A or a Model B system. The R1 handler wastes a lot of cycles looping while waiting for a character to arrive and the implication is that was a big change for the user since they're complaining that running R1 is wasting time compared to their previous handler. If the user had been running R2 on a model B system, they're already used to the performance hit of looping and so wouldn't have noticed a performance difference switching to R1, so we can eliminate Model B from consideration.

So this message came from a user running handler R1 on a model A system.

R1	<pre>ReadCh_h() { // Version R1 if (BufferEmpty(0)) UserMState.Regs[XP] = UserMState.Regs[XP] - 4; else UserMState.Regs[0] = ReadInputBuffer(0); }</pre>		
R2	<pre>ReadCh_h() { // Version R2 int kbdnum = ProcTbl[Cur].DPYNum; while (BufferEmpty(kbdnum)); UserMState.Regs[0] = ReadInputBuffer(kbdnum); }</pre>		
R3	<pre>ReadCh_h() { // Version R3 int kbdnum = ProcTbl[Cur].DPYNum; if (BufferEmpty(kbdnum)) { UserMState.Regs[XP] = UserMState.Regs[XP] - 4; Scheduler(); } else UserMState.Regs[0] = ReadInputBuffer(kbdnum); }</pre>		
	A	B	C
R1	X	X	X
R2	X	X	X
R3	X		X

Model A: A timeshared Beta system whose OS kernel is uninterruptable
Model B: A timeshared Beta system which enables device interrupts during handing of SVC traps
Model C: A single-process (not timeshared) system which runs dedicated application code

Which handler & OS?
 “Neat, the new system seems to work fine.
 It even wastes less CPU time than it used to!”

The final message reads “Neat, the new system seems to work fine. It even wastes less CPU time than it used to!”

Since the system works as expected with the new handler, we can eliminate a lot of possibilities.

Handler R1 wouldn't work fine on a timesharing system since the user could tell that the processes were now all reading from the same keyboard buffer, so we can eliminate R1 on Models A and B.

And handlers R2 and R3 wouldn't work on a Model C system since that doesn't include process tables or scheduling, eliminating the right-most column.

Finally handler R2 wouldn't work on a Model A system with its uninterruptible kernel since any attempt to read from an empty buffer would cause an infinite loop.

So, the message must have been sent by a Model B user now running R3.

Well, that was fun! Just like solving the logic puzzles you find in games magazines :)

The Need for “Real Time”

Side-effects of CPU virtualization

- + abstraction of machine resources
(memory, I/O, registers, etc.)
- + multiple “processes” executing concurrently
- + better CPU utilization
- Processing throughput is more variable

Our approach to dealing with the asynchronous world

- I/O - separate “event handling” from “event processing”

Difficult to meet “hard deadlines”

- control applications, e.g., ESC on cars
- playing videos/MP3s

Real-time as an alternative to time-sliced or fixed-priority preemptive scheduling



So far in constructing our timesharing system, we’ve worked hard to build an execution environment that gives each process the illusion of running on its own independent virtual machine. The processes appear to run concurrently although we’re really quickly switching between running processes on a single hardware system. This often leads to better overall utilization since if a particular process is waiting for an I/O event, we can devote the unneeded cycles to running other processes.

The downside of timesharing is that it can be hard to predict exactly how long a process will take to complete since the CPU time it will receive depends on how much time the other processes are using. So we’d need to know how many other processes there are, whether they’re waiting for I/O events, etc. In a timesharing system we can’t make any guarantees on completion times.

And we chose to have the OS play the intermediary between interrupt events triggered by the outside world and the user-mode programs where the event processing occurs. In other words, we’ve separated event handling (where the data is stored by the OS) and event processing (where the data is passed to user-mode programs via SVCs).

This means that using a conventional timesharing system, it’s hard to ensure that event processing will be complete by a specified event deadline, *i.e.*, before the end of a specified time period after the event was triggered.

Since modern CPU chips provide inexpensive, high-performance, general-purpose computing, they are often used as the “brains” of control systems where deadlines are a fact of life.

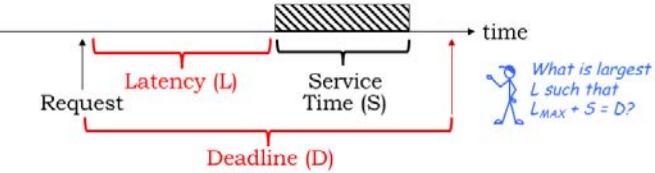
For example, consider the electronic stability control (ESC) system on modern cars. This system helps drivers maintain control of their vehicle during steering and braking maneuvers by keeping the car headed in the driver’s intended direction. The computer at the heart of the system measures the forces on the car, the direction of steering, and the rotation of the wheels to determine if there’s been a loss of control due to a loss of traction, *i.e.*, is the car “spinning out”? If so, the ESC uses rapid automatic braking of individual wheels to prevent the car’s heading from veering from the driver’s intended heading. With ESC you can slam on your brakes or swerve to avoid an obstacle and not worry that the car will suddenly fishtail out of control. You can feel the system working as a chatter in the brakes.

To be effective, the ESC system has to guarantee the correct braking action at each wheel within a certain time of receiving dangerous sensor settings. This means that it has to be able to guarantee that certain subroutines will run to completion within some predetermined time of a sensor event. To be able to make these guarantees we'll have to come up with a better way to schedule process execution — round-robin scheduling won't get the job done! Systems that can make such guarantees are called "real-time systems".

Interrupt Latency

One way to measure the real-time performance of a system is **INTERRUPT LATENCY**:

- HOW MUCH TIME can elapse between an interrupt request and the START of its handler?



Sometimes bad things happen when service is delayed beyond its "dead"-line:

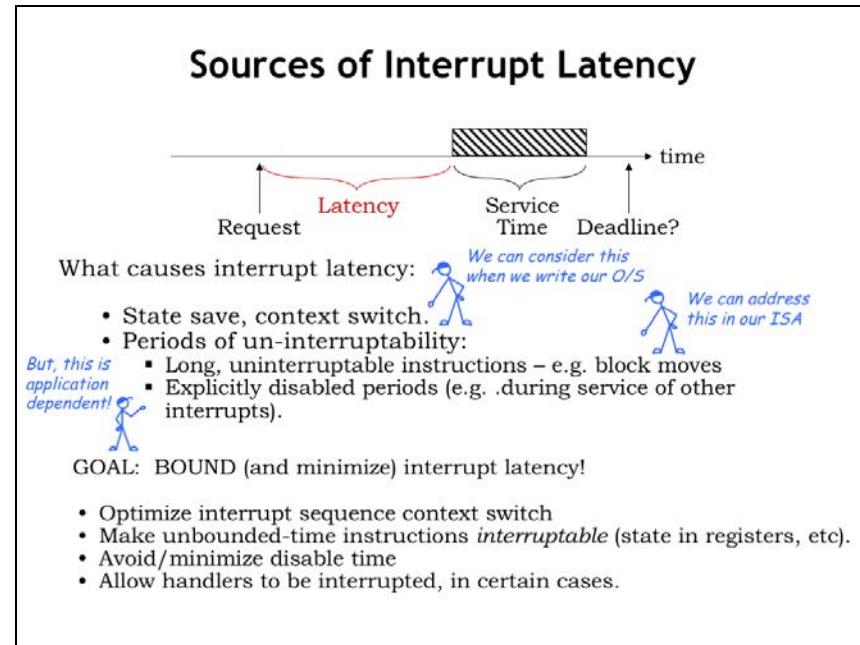
Missed characters
Automobile crashes
Nuclear meltdowns

"HARD"
Real time constraints



One measure of performance in a real-time system is the interrupt latency L , the amount of time that elapses between a request to run some code and when that code actually starts executing. If there's a deadline D associated with servicing the request, we can compute the maximum allowable latency that still permits the service routine to complete by the deadline. In other words, what's the largest L such that $L_{max} + S = D$?

Bad things can happen if we miss certain deadlines. Maybe that's why we call them "dead"-lines :) In those cases we want our real time system to guarantee that the actual latency is always less than the maximum allowable latency. These critical deadlines give rise to what we call "hard real-time constraints".



What factors contribute to interrupt latency?

Well, while handling an interrupt it takes times to save the process state, switch to the kernel context, and dispatch to the correct interrupt handler. When writing our OS, we can work to minimize the amount of code involved in the setup phase of an interrupt handler.

We also have to avoid long periods of time when the processor cannot be interrupted. Some ISAs have complex multi-cycle instructions, *e.g.*, block move instructions where a single instruction makes many memory accesses as it moves a block of data from one location to another. In designing the ISA, we need to avoid such instructions or design them so that they can be interrupted and restarted.

The biggest problem comes when we're executing another interrupt handler in kernel mode. In kernel mode, interrupts are disabled, so the actual latency will be determined by the time it takes to complete the current interrupt handler in addition to the other costs mentioned above. This latency is not under the control of the CPU designer and will depend on the particular application. Writing programs with hard real-time constraints can get complicated!

Our goal is to bound and minimize interrupt latency. We'll do this by optimizing the cost of taking an interrupt and dispatching to the correct handler code. We'll avoid instructions whose execution time is data dependent. And we'll work to minimize the time spent in kernel mode.

But even with all these measures, we'll see that in some cases we'll have to modify our system to allow interrupts even in kernel mode.

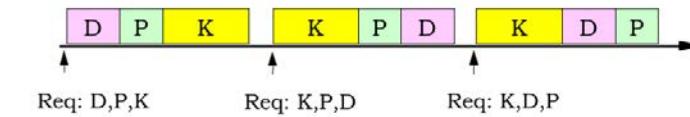
Next we'll look at some concrete examples and see what mechanisms are required to make guarantees about hard real-time constraints.

Scheduling of Multiple Devices

"TOY" System scenario:	Actual w/c Latency	DEVICE	Service Time
	$500 + 400 = 900$	Keyboard	800
	$800 + 400 = 1200$	Disk	500
	$800 + 500 = 1300$	Printer	400



What is the WORST CASE latency seen by each device?



Assumptions:

- Infrequent interrupt requests (each happens only once/scenario)
- Simultaneous requests might be served in ANY order.... Whence
- Service of EACH device might be delayed by ALL others!

Suppose we have a real-time system supporting three devices: a keyboard whose interrupt handler has a service time of 800 us, a disk with a service time of 500 us, and a printer with a service time of 400 us.

What is the worst-case latency seen by each device? For now we'll assume that requests are infrequent, *i.e.*, that each request only happens once in each scenario. Requests can arrive at any time and in any order. If we serve the requests in first-come-first-served order, each device might be delayed by the service of all other devices.

So the start of the keyboard handler might be delayed by the execution of the disk and printer handlers, a worst-case latency of 900 us.

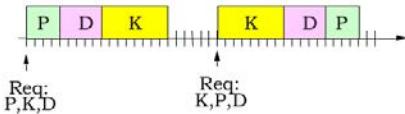
The start of the disk handler might be delayed by the keyboard and printer handlers, a worst-case latency of 1200 us.

And the printer handler might be delayed by the keyboard and disk handlers, a worst-case latency of 1300 us.

In this scenario we see that long-running handlers have a huge impact on the worst-case latency seen by the other devices. What are the possibilities for reducing the worst-case latencies? Is there a better scheduling algorithm than first-come-first-served?

Weak (Non-preemptive) Priorities

ISSUE: Processor becomes interruptible on returning to user mode, several interrupt requests are pending. Which is served first?



WEAK PRIORITY ORDERING: Check in prescribed sequence, e.g.:
DISK > PRINTER > KEYBOARD.

Latencies with WEAK PRIORITIES:
Service of each device might be delayed by:

- Service of 1 other (arbitrary) device, whose interrupt request was just honored;
- +
- Service of ALL higher-priority devices.

Actual w/c Latency	DEVICE	Service Time
900	Keyboard	800
800	Disk	500
1300	Printer	400

vs 1200 –
Now delayed by only 1 service!

One strategy is to assign priorities to the pending requests and to serve the requests in priority order. If the handlers are uninterruptible, the priorities will be used to select the *next* task to be run at the completion of the current task. Note that under this strategy, the current task always runs to completion even if a higher-priority request arrives while it's executing. This is called a "nonpreemptive" or "weak" priority system.

Using a weak priority system, the worst-case latency seen by each device is the worst-case service time of all the other devices (since that handler may have just started running when the new request arrives), plus the service time of all higher-priority devices (since they'll be run first).

In our example, suppose we assigned the highest priority to the disk, the next priority to the printer, and the lowest priority to the keyboard.

The worst-case latency of the keyboard is unchanged since it has the lowest priority and hence can be delayed by the higher-priority disk and printer handlers.

The disk handler has the highest priority and so will always be selected for execution after the current handler completes. So its worst-case latency is the worst-case service time for the currently-running handler, which in this case is the keyboard, so the worst-case latency for the disk is 800 us. This is a considerable improvement over the first-come-first-served scenario.

Finally the worst-case scenario for the printer is 1300 us since it may have to wait for the keyboard handler to finish running (which can take up to 800 us) and then for a higher-priority disk request to be serviced (which takes 500 us).

Setting Priorities

How should priorities be assigned given hard real-time constraints? We'll assume each device has a service deadline D.

If not otherwise specified, assume D is the time until the next request for the same device, e.g., the keyboard handler should be finished processing one character before the next arrives.

“Earliest Deadline” is a strategy for assigning priorities that is guaranteed to meet the deadlines if any priority assignment can meet the deadlines:

1. Sort the requests by their deadlines
2. Assign the highest priority to the earliest deadline, second priority to the next deadline, and so on.
3. Weak priority scheduling: choose the pending request with the highest priority, i.e., that has the earliest deadline.

How should priorities be assigned given hard real-time constraints? We'll assume each device has a service deadline D after the arrival of its service request.

If not otherwise specified, assume D is the time until the *next* request for the same device. This is a reasonably conservative assumption that prevents the system from falling further and further behind. For example, it makes sense that the keyboard handler should finish processing one character before the next arrives.

“Earliest Deadline” is a strategy for assigning priorities that is guaranteed to meet the deadlines if any priority assignment can meet the deadlines. It's very simple: Sort the requests by their deadlines.

Assign the highest priority to the earliest deadline, second priority to the next deadline, and so on. A weak priority system will choose the pending request with the highest priority, *i.e.*, the request that has the earliest deadline.

Earliest Deadline has an intuitive appeal. Imagine standing in a long security line at the airport. It would make sense to prioritize the processing of passengers who have the earliest flights assuming that there's enough time to process everyone before their flight leaves. Processing 10 people whose flights leave in 30 minutes before someone whose flight leaves in 5 min will cause that last person to miss their flight. But if that person is processed first, the other passengers may be slightly delayed but everyone will make their flight. This is the sort of scheduling problem that Earliest Deadline and a weak priority system can solve.

It's outside the scope of our discussion, but it's interesting to think about what should happen if some flights are going to be missed. If the system is overloaded, prioritizing by earliest deadline may mean that everyone will miss their flights! In this scenario it might be better to assign priorities to the minimize the total number of missed flights. This gets complicated in a hurry since the assignment of priorities now depends on exactly what requests are pending and how long it will take them to be serviced. An intriguing problem to think about!

The Need for Preemption

Without preemption, ANY interrupt service can delay ANY other service request... the slowest service time constrains response to fastest devices. Often, tight deadlines can't be met using this scheme alone.

EXAMPLE: 800 uSec deadline (hence 300 uSec maximum interrupt latency) on disk service, to avoid missing next sector...

Priority	Latency w/ preemption	Latency using weak priority	Device	Service Time (S)	Deadline (D)	L _{MAX}
1	[D,P] 900	900us	Keyboard	800us		
3	~0	800us	Disk	500us	800us	300us
2	[D] 500	1300us	Printer	400us		

CAN'T SATISFY the disk requirement in this system using weak priorities!

need PREEMPTION: Allow handlers for LOWER PRIORITY interrupts to be interrupted by HIGHER priority requests!

In a weak priority system the currently-running task will always run to completion before considering what to run next. This means the worst-case latency for a device always includes the worst-case service time across all the other devices, *i.e.*, the maximum time we have to wait for the currently-running task to complete.

If there's a long-running task that usually means it will be impossible to meet tight deadlines for other tasks. For example, suppose disk requests have a 800 us deadline in order to guarantee the best throughput from the disk subsystem. Since the disk handler service time is 500 us, the maximum allowable latency between a disk request and starting to execute the disk service routine is 300 us.

Oops! The weak priority scheme can only guarantee a maximum latency of 800 us, not nearly fast enough to meet the disk deadline. We can't meet the disk deadline using weak priorities.

We need to introduce a preemptive priority system that allows lower-priority handlers to be interrupted by higher-priority requests. We'll refer to this as a "strong" priority system. Suppose we gave the disk the highest priority, the printer second priority, and keyboard the lowest priority, just like we did before.

Now when a disk request arrives, it will start executing immediately without having to wait for the completion of the lower-priority printer or keyboard handlers. The worst-case latency for the disk has dropped to 0.

The printer can only be preempted by the disk, so its worst-case latency is 500 us. Since it has the lowest priority, the worst-case latency for the keyboard is unchanged at 900 us since it might still have to wait on the disk and printer.

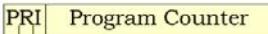
The good news: with the proper assignment of priorities, the strong priority system can guarantee that disk requests will be serviced by the 800 us deadline.

Strong Priority Implementation

STRONG PRIORITY ORDERING: Allow handlers for LOWER PRIORITY interrupts to be preempted (interrupted) by HIGHER PRIORITY requests.

SCHEME:

- Expand supervisor bit in PC to be a PRIORITY integer PRI (eg, 3 bits for 8 levels)
- ASSIGN a priority to each device.
- Prior to each instruction execution:
 - Find priority P_{DEV} of highest requesting device, say D_i
 - Take interrupt if and only if $P_{DEV} > PRI$, set $PRI = P_{DEV}$.

PC: 

Strong priorities:

KEY: Priority in Processor state

Allows interruption of (certain) handlers

Allows preemption, but not reentrance

BENEFIT: Latency seen at high priorities UNAFFECTED by service times at low priorities.

We'll need to make a small tweak to our Beta hardware to implement a strong priority system. We'll replace the single supervisor mode bit in PC[31] with, say, a three-bit field (PRI) in PC[31:29] that indicates which of the eight priority levels the processor is currently running at.

Next, we'll modify the interrupt mechanism as follows. In addition to requesting an interrupt, the requesting device also specifies the 3-bit priority it was assigned by the system architect. We'll add a priority encoder circuit to the interrupt hardware to select the highest-priority request and compare the priority of that request (P_{DEV}) to the 3-bit PRI value in the PC. The system will take the interrupt request only if $P_{DEV} > PRI$, *i.e.*, if the priority of the request is *higher* than the priority the system is running at.

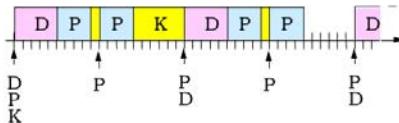
When the interrupt is taken, the old PC and PRI information is saved in XP, and the new PC is determined by the type of interrupt and the new PRI field is set to P_{DEV} . So the processor will now be running at the higher priority specified by the device.

A strong priority system allows low-priority handlers to be interrupted by higher-priority requests, so the worst-case latencies seen at high priorities is unaffected by the service times of lower-priority handlers.

Recurring Interrupts

Consider interrupts which recur at bounded rates:

Priority	Latency using strong priority	Device	Service Time (S)	Deadline (D)	L_{MAX}	Max Freq.
1	900us	Keyboard	800us			100/s
3	0	Disk	500us	800us	300us	500/s
2	500us	Printer	400us			1000/s



Note that interrupt LATENCIES don't tell the whole story—consider COMPLETION TIMES, e.g., for Keyboard in the example above.

Keyboard service not complete until 3 ms after request!

Using strong priorities allows us to assign a high priority to devices with tight deadlines and thus guarantee their deadlines are met.

Now let's consider the impact of recurring interrupts, *i.e.*, multiple interrupt requests from each device. We've added a "maximum frequency" column to our table, which gives the maximum rate at which requests will be generated by each device.

The execution diagram for a strong priority system is shown below the table. Here we see there are multiple requests from each device, in this case shown at their maximum possible rate of request. Each tick on the timeline represent 100 us of real time. Printer requests occur every 1 ms (10 ticks), disk requests every 2 ms (20 ticks), and keyboard requests every 10 ms (100 ticks).

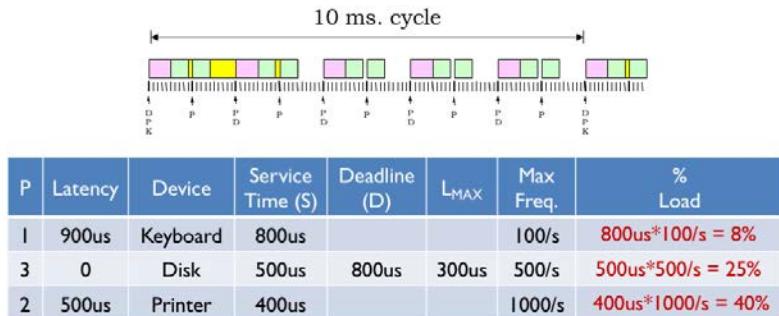
In the diagram you can see that the high-priority disk requests are serviced as soon as they're received. And that medium-priority printer requests preempt lower-priority execution of the keyboard handler. Printer requests would be preempted by disk requests, but given their request patterns, there's never a printer request in progress when a disk request arrives, so we don't see that happening here.

The maximum latency before a keyboard requests starts is indeed 900 us. But that doesn't tell the whole story! As you can see, the poor keyboard handler is continually preempted by higher-priority disk and printer requests and so the keyboard handler doesn't complete until 3 ms after its request was received! This illustrates why real-time constraints are best expressed in terms of deadlines and not latencies.

If the keyboard deadline had been less than 3 ms, even the strong priority system would have failed to meet the hard real-time constraints. The reason would be that there simply aren't enough CPU cycles to meet the recurring demands of the devices in the face of tight deadlines.

Interrupt Load

How much CPU time is consumed by interrupt service?



- User-mode share of CPU = $1 - \sum(S_{DEV} * \text{max_freq}_{DEV}) = 0.27$
- Also check to see if enough CPU time to meet all deadlines

Speaking of having enough CPU cycles, there are several calculations we need to do when thinking about recurring interrupts.

The first is to consider how much load each periodic request places on the system. There's one keyboard request every 10 ms and servicing each request takes 800 us, which consumes $800\mu s / 10ms = 8\%$ of the CPU. A similar calculation shows that servicing the disk takes 25% of the CPU and servicing the printer takes 40% of the CPU.

Collectively servicing all the devices takes 73% of the CPU cycles, leaving 27% for running user-mode programs. Obviously we'd be in trouble if takes more than 100% of the available cycles to service the devices.

Another way to get in trouble is to not have enough CPU cycles to meet each of the deadlines. We need $500/800 = 67.5\%$ of the cycles to service the disk in the time between the disk request and disk deadline.

If we assume we want to finish serving one printer request before receiving the next, the effective printer deadline is 1000 us. In 1000 us we need to be able to service one higher-priority disk request (500 us) and, obviously, the printer request (400 us). So we'll need to use 900 us of CPU in that 1000 us interval. Whew, just barely made it!

Suppose we tried setting the keyboard deadline to 2000 us. In that time interval we'd also need to service 1 disk request and 2 printer requests. So the total service time needed is $500 + 2*400 + 800 = 2100$ us. Oops, that exceeds the 2000 us window we were given, so we can't meet the 2000 us deadline with the available CPU resources.

But if the keyboard deadline is 3000 us, let's see what happens. In a 3000 us interval we need to service 2 disk requests, 3 printer requests, and, of course, 1 keyboard request, for a total service time of $2*500 + 3*400 + 800 = 3000$ us. Whew! Just made it!

Example: Mr. Blue Visits the ISS



International Space Station's on-board computer performs 3 tasks:

- guiding incoming supply ships to a safe docking
- monitoring gyros to keep solar panels properly oriented
- controlling air pressure in the crew cabin

Task	Period	Service time	Deadline	
16.67% Supply ship guidance	30ms	5ms	25ms	C,G = 10 + 10 + (5) = 25
25% Gyroscopes	40	10	20	C = 10 + (10) = 20
10% Cabin pressure	100	? 10	100	S,G = 5 + 10 + (10) = 25

Assuming a **weak priority system**:

- What is the maximum service time for "cabin pressure" that still allows all constraints to be met? $\leq 10 \text{ ms}$
- Give a weak priority ordering that meets the constraints $G > SSG > CP$
- What fraction of the time will the processor spend idle? 48.33%
- What is the worst-case completion time for each task?

Let's finish up by looking at two extended examples. The scenario for both examples is the control system for the International Space Station, which has to handle three recurring tasks: supply ship guidance (SSG), gyroscope control (G), and cabin pressure (CP). For each device, the table shows us the time between successive requests (the period), the service time for each request, and the service deadline for each request.

We'll first analyze the system assuming that it's using a weak priority system.

First question: What is the maximum service time for the cabin pressure task that still allows all constraints to be met? Well, the SSG task has a maximum allowable latency of 20 ms, *i.e.*, its service routine must start execution within 20 ms if it is to meet its 25 ms deadline. The G task has a maximum allowable latency of 10 ms if it's to meet its deadline. So no other handler can take longer than 10 ms to run or the G task will miss its deadline.

- Give a weak priority ordering that meets the constraints. Using the earliest deadline strategy discussed earlier, the priority would be G with the highest priority, SSG with the middle priority, and CP with the lowest priority.
- What fraction of time will the processor spend idle? We need to compute the fraction of CPU cycles needed to service the recurring requests for each task. SSG takes $5/30 = 16.67\%$ of the CPU cycles. G takes $10/40 = 25\%$ of the CPU cycles. And CP takes $10/100 = 10\%$ of the CPU cycles. So servicing the task requests takes 51.67% of the cycles, leaving 48.33% of the cycles unused. So the astronauts will be able to play Minecraft in their spare time :)
- What is the worst-case delay for each task until completion of its service routine? Each task might have to wait for the longest-running lower-priority handler to complete plus the service times of any other higher-priority tasks plus, of course, its own service time.

SSG might have to wait for CP and G to complete (a total of 20 ms), then add its own service time (5 ms). So its worst-case completion time is 25 ms after the request.

G might wait for CP to complete (10 ms), then add its own service time (10 ms) for a worst-case completion time of 20 ms.

CP might have to wait for SSG to finish (5 ms), then wait for G to run (10 ms), then add its own service time (10 ms) for a worst-case completion time of 25 ms.

Example: Mr. Blue Visits ISS (cont'd.)

Our Russian collaborators don't like the sound of a "weak" priority interrupt system and lobby heavily to use a "strong" priority interrupt system instead.

	Task	Period	Service time	Deadline	
16.67%	Supply ship guidance	30ms	5ms	25ms	[G] 10 + 5
25%	Gyroscopes	40	10	20	10
50%	Cabin pressure	100	? 50	100	100

Assuming a **strong priority system**, G > SSG > CP:

1. What is the maximum service time for "cabin pressure" that still allows all constraints to be met? $100 - (3*10) - (4*5) = 50$
2. What fraction of the time will the processor spend idle? 8.33%
3. What is the worst-case completion time for each task?

Let's redo the problem, this timing assuming a strong priority system where, as before, G has the highest priority, SSG the middle priority, and CP the lowest priority.

1. What is the maximum service time for CP that still allows all constraints to be met? This calculation is different in a strong priority system, since the service time of CP is no longer constrained by the maximum allowable latency of the higher-priority tasks — they'll simply preempt CP when they need to run!

Instead we need to think about how much CPU time will be used by the SSG and G tasks in the 100 ms interval between the CP request and its deadline.

In a 100 ms interval, there might be four SSG requests (at times 0, 30, 60, and 90) and three G requests (at times 0, 40, and 80). Together these require a total of 50 ms to service. So the service time for CP can be up 50 ms and still meet the 100 ms deadline.

2. What fraction of the time will the processor spend idle? Assuming a 50 ms service time for CP, it now consumes 50% of the CPU. The other request loads are as before, so 91.67% of the CPU cycles will be spent servicing requests, leaving 8.33% of idle time.

3. What is the worst-case completion time for each task?

The G task has the highest priority, so its service routine runs immediately after the request is received and its worst-case completion time is exactly its service time.

In the 25 ms interval between an SSG request and its deadline, there might be at most one G request that will preempt execution. So the worst-case completion time is one G service time (10 ms) plus the SSG service time (5 ms).

Finally, from the calculation for problem 1, we chose the service time for the CP task so that it will complete just at its deadline of 100 ms, taking into account the service time for multiple higher-priority requests.

Summary

Device interface – two parts:

- Device side: handle interrupts from device (transparent to apps)
- Application side: handle interrupts (SVCs) from application

Scheduler interaction:

- “Sleeping” (*inactive) processes waiting for device I/O
- Handler coding issues, looping thru User mode

Real Time constraints, scheduling, guarantees

- Complex, hard scheduling problems – a black art!
- Weak (non-preemptive) vs Strong (preemptive) priorities help...
- Common real-world interrupt systems:
 - Fixed number (eg, 8 or 16) of strong priority levels
 - Each strong priority level can support many devices, arranged in a weak priority chain

We covered a lot of ground in this lecture!

We saw that the computation needed for user-mode programs to interact with external devices was split into two parts. On the device-side, the OS handles device interrupts and performs the task of moving data between kernel buffers and the device. On the application side, user-mode programs access the information via SVC calls to the OS.

We worried about how to handle SVC requests that needed to wait for an I/O event before the request could be satisfied. Ultimately we came up with a sleep/wakeup mechanism that suspends execution of the process until the some interrupt routine signals that the needed information has arrived, causing the sleeping process to marked as active. Then the SVC is retried the next time the now active process is scheduled for execution.

We discussed hard real-time constraints with their latencies, service times and deadlines. Then we explored the implementation of interrupt systems using both weak and strong priorities.

Real-life computer systems usual implement strong priorities and support a modest number of priority levels, using a weak priority system to deal with multiple devices assigned to the same strong priority level. This seems to work quite well in practice, allowing the systems to meet the variety of real-time constraints imposed by their I/O devices.

L19: Concurrency and Synchronization

1. Interprocess Communication
2. Synchronous Communication
3. FIFO Buffering
4. Example: Bounded Buffer Problem
5. Semaphores (Dijkstra)
6. Semaphores for Precedence
7. Semaphores for Resource Allocation
8. Bounded Buffer Problem with Semaphores
9. Flow Control Problems
10. Bounded Buffer Problem with More Semaphores
11. Simultaneous Transactions
12. But, What If...
13. Semaphores for Mutual Exclusion
14. Producer/Consumer Atomicity Problems
15. Bounded Buffer Problem with Even More Semaphores
16. The Power of Semaphores
17. Semaphore Implementation
18. Semaphores as a Supervisor Call
19. Hardware Support for Semaphores
20. Synchronization: The Dark Side
21. Dining Philosophers
22. Deadlock!
23. One Solution
24. Dealing With Deadlocks
25. Summary

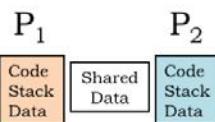
Interprocess Communication

Why have multiple processes?

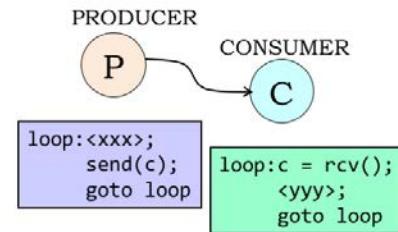
- Concurrency
- Asynchrony
- Processes as a programming primitive
- Data-/Event-driven

How to communicate?

- Shared Memory (overlapping contexts)...
- Synchronization instructions (hardware support)
- Supervisor calls



Classic Example:
“Producer-Consumer” Problem



Real-World Examples:
Compiler/Assembler,
Application Frontend/Backend,
UNIX pipeline

It's not unusual to find that an application is organized as multiple communicating processes. What's the advantage of using multiple processes instead of just a single process?

Many applications exhibit concurrency, *i.e.*, some of the required computations can be performed in parallel. For example, video compression algorithms represent each video frame as an array of 8-pixel by 8-pixel macroblocks. Each macroblock is individually compressed by converting the 64 intensity and color values from the spatial domain to the frequency domain and then quantizing and Huffman encoding the frequency coefficients. If you're using a multi-core processor to do the compression, you can perform the macroblock compressions concurrently.

Applications like video games are naturally divided into the “front-end” user interface and “back-end” simulation and rendering engines. Inputs from the user arrive asynchronously with respect to the simulation and it's easiest to organize the processing of user events separately from the backend processing.

Processes are an effective way to encapsulate the state and computation for what are logically independent components of an application, which communicate with one another when they need to share information.

These sorts of applications are often data- or event-driven, *i.e.*, the processing required is determined by the data to be processed or the arrival of external events.

How should the processes communicate with each other?

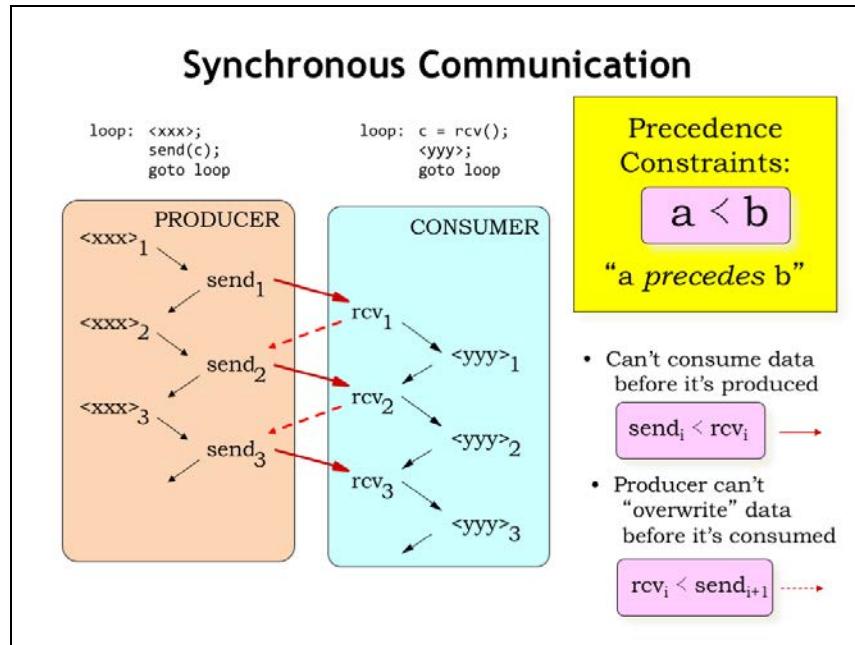
If the processes are running out of the same physical memory, it would be easy to arrange to share memory data by mapping the same physical page into the contexts for both processes. Any data written to that page by one process will be able to be read by the other process.

To make it easier to coordinate the processes' communicating via shared memory, we'll see it's convenient to provide synchronization primitives. Some ISAs include instructions that make it easy to do the required synchronization.

Another approach is to add OS supervisor calls to pass messages from one process to another. Message passing involves more overhead than shared memory, but makes the application programming independent of whether the communicating processes are running on the same physical processor.

In this lecture, we'll use the classic producer-consumer problem as our example of concurrent processes that need to communicate and synchronize. There are two processes: a producer and a consumer. The producer is running in a loop, which performs some computation to generate information, in this case, a single character C. The consumer is also running a loop, which waits for the next character to arrive from the producer, then performs some computation .

The information passing between the producer and consumer could obviously be much more complicated than a single character. For example, a compiler might produce a sequence of assembly language statements that are passed to the assembler to be converted into the appropriate binary representation. The user interface front-end for a video game might pass a sequence of player actions to the simulation and rendering back-end. In fact, the notion of hooking multiple processes together in a processing pipeline is so useful that the Unix and Linux operating systems provide a PIPE primitive in the operating system that connects the output channel of the upstream process to the input channel of the downstream process.



Let's look at a timing diagram for the actions of our simple producer/consumer example. We'll use arrows to indicate when one action happens before another. Inside a single process, *e.g.*, the producer, the order of execution implies a particular ordering in time: the first execution of `is` is followed by the sending of the first character. Then there's the second execution of `is`, followed by the sending of the second character, and so on. In later examples, we'll omit the timing arrows between successive statements in the same program.

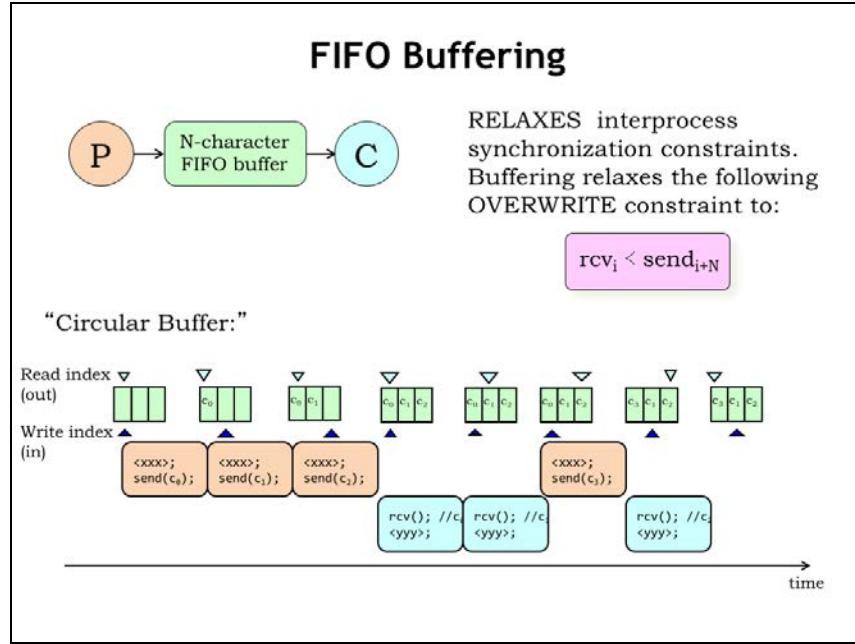
We see a similar order of execution in the consumer: the first character is received, then the computation is performed for the first time, etc. Inside of each process, the process' program counter is determining the order in which the computations are performed.

So far, so good - each process is running as expected. However, for the producer/consumer system to function correctly as a whole, we'll need to introduce some additional constraints on the order of execution. These are called “precedence constraints” and we'll use this stylized less-than sign to indicate that computation A must precede, *i.e.*, come before, computation B.

In the producer/consumer system we can't consume data before it's been produced, a constraint we can formalize as requiring that the i^{th} send operation has to precede the i^{th} receive operation. This timing constraint is shown as the solid red arrow in the timing diagram.

Assuming we're using, say, a shared memory location to hold the character being transmitted from the producer to the consumer, we need to ensure that the producer doesn't overwrite the previous character before it's been read by the consumer. In other words, we require the i^{th} receive to precede the $i+1^{\text{st}}$ send. These timing constraints are shown as the dotted red arrows in the timing diagram.

Together these precedence constraints mean that the producer and consumer are tightly coupled in the sense that a character has to be read by the consumer before the next character can be sent by the producer, which might be less than optimal if the and computations take a variable amount of time. So let's see how we can relax the constraints to allow for more independence between the producer and consumer.



We can relax the execution constraints on the producer and consumer by having them communicate via N-character first-in-first-out (FIFO) buffer. As the producer produces characters it inserts them into the buffer. The consumer reads characters from the buffer in the same order as they were produced. The buffer can hold between 0 and N characters. If the buffer holds 0 characters, it's empty; if it holds N characters, it's full. The producer should wait if the buffer is full, the consumer should wait if the buffer is empty.

Using the N-character FIFO buffer relaxes our second overwrite constraint to the requirement that the i^{th} receive must happen before $i+N^{\text{th}}$ send. In other words, the producer can get up to N characters ahead of the consumer.

FIFO buffers are implemented as an N-element character array with two indices: the read index indicates the next character to be read, the write index indicates the next character to be written. We'll also need a counter to keep track of the number of characters held by the buffer, but that's been omitted from this diagram. The indices are incremented modulo N, *i.e.*, the next element to be accessed after the N-1st element is the 0th element, hence the name “circular buffer”.

Here's how it works. The producer runs, using the write index to add the first character to the buffer. The producer can produce additional characters, but must wait once the buffer is full.

The consumer can receive a character anytime the buffer is not empty, using the read index to keep track of the next character to be read. Execution of the producer and consumer can proceed in any order so long as the producer doesn't write into a full buffer and the consumer doesn't read from any empty buffer.

Example: Bounded Buffer Problem

```
SHARED MEMORY:  
char buf[N]; /* The buffer */  
int in=0, out=0;
```

```
PRODUCER:  
send(char c){  
    buf[in] = c;  
    in = (in+1)% N;  
}
```

```
CONSUMER:  
char rcv(){  
    char c;  
    c = buf[out];  
    out = (out+1)% N;  
    return c;  
}
```

Problem: Doesn't enforce precedence constraints
(e.g. rcv() could be invoked prior to any send())

Here's what the code for the producer and consumer might look like. The array and indices for the circular buffer live in shared memory where they can be accessed by both processes. The SEND routine in the producer uses the write index IN to keep track of where to write the next character. Similarly the RCV routine in the consumer uses the read index OUT to keep track of the next character to be read. After each use, each index is incremented modulo N.

The problem with this code is that, as currently written, neither of the two precedence constraints is enforced. The consumer can read from an empty buffer and the producer can overwrite entries when the buffer is full.

We'll need to modify this code to enforce the constraints and for that we'll introduce a new programming construct that we'll use to provide the appropriate inter-process synchronization.

Semaphores (Dijkstra)



Programming construct for synchronization:

- NEW DATA TYPE: *semaphore*, an integer ≥ 0
`semaphore s = K; // initialize s to K`



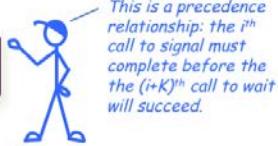
- NEW OPERATIONS (defined on semaphores):

- `wait(semaphore s)`
wait until $s > 0$, then $s = s - 1$
 - `signal(semaphore s)`
 $s = s + 1$ (one WAITing process may now be able to proceed)

- SEMANTIC GUARANTEE: A semaphore s initialized to K enforces the constraint:

Often you will see
`P(s)` used for `wait(s)`
and
`V(s)` used for `signal(s)`!
 P = "proberen" (test) or
"pakken" (grab)
 V = "verhogen" (increase)

`signal(s)i < wait(s)i+K`



What we'd like to do is to create a single abstraction that can be used to address all our synchronization needs. In the early 1960's, the Dutch computer scientist Edsger Dijkstra proposed a new abstract data type called the semaphore, which has an integer value greater than or equal to 0. A programmer can declare a semaphore as shown here, specifying its initial value. The semaphore lives in a memory location shared by all the processes that need to synchronize their operation.

The semaphore is accessed with two operations: WAIT and SIGNAL. The WAIT operation will wait until the specified semaphore has a value greater than 0, then it will decrement the semaphore value and return to the calling program. If the semaphore value is 0 when WAIT is called, conceptually execution is suspended until the semaphore value is non-zero. In a simple (inefficient) implementation, the WAIT routine loops, periodically testing the value of the semaphore, proceeding when its value is non-zero.

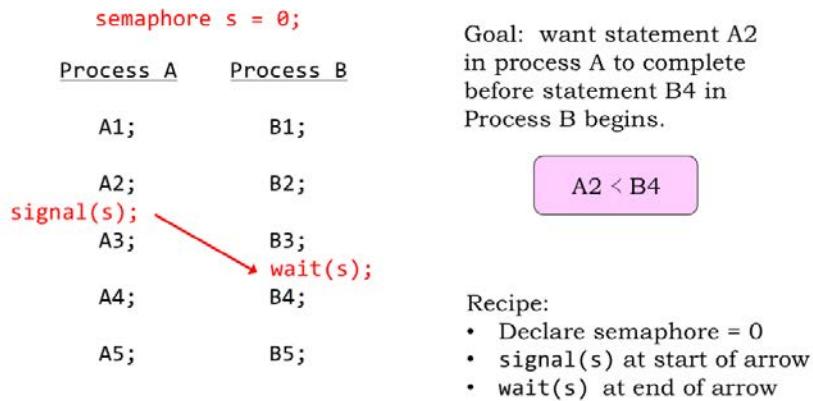
The SIGNAL operation increments the value of the specified semaphore. If there any processes WAITing on that semaphore, exactly one of them may now proceed. We'll have to be careful with the implementation of SIGNAL and WAIT to ensure the "exactly one" constraint is satisfied, *i.e.*, that two processes both WAITing on the same semaphore won't both think they can decrement it and proceed after a SIGNAL.

A semaphore initialized with the value K guarantees that the i^{th} call to SIGNAL will precede $(i+K)^{\text{th}}$ call to WAIT. In a moment, we'll see some concrete examples that will make this clear. Note that in 6.004, we're ruling out semaphores with negative values.

In the literature, you may see `P(s)` used in place of `WAIT(s)` and `V(s)` used in place of `SIGNAL(s)`. These operation names are derived from the Dutch words for "test" and "increase".

Let's see how to use semaphores to implement precedence constraints.

Semaphores for Precedence



Here are two processes, each running a program with 5 statements. Execution proceeds sequentially within each process, so A1 executes before A2, and so on. But there are no constraints on the order of execution between the processes, so statement B1 in Process B might be executed before or after any of the statements in Process A. Even if A and B are running in a timeshared environment on a single physical processor, execution may switch at any time between processes A and B.

Suppose we wish to impose the constraint that the execution of statement A2 completes before execution of statement B4 begins. The red arrow shows the constraint we want.

Here's the recipe for implementing this sort of simple precedence constraint using semaphores.

First, declare a semaphore (called "s" in this example) and initialize its value to 0.

Place a call to signal(s) at the start of the arrow. In this example, signal(s) is placed after the statement A2 in process A.

Then place a call to wait(s) at the end of the arrow. In this example, wait(s) is placed before the statement B4 in process B.

With these modifications, process A executes as before, with the signal to semaphore s happening after statement A2 is executed.

Statements B1 through B3 also execute as before, but when the wait(s) is executed, execution of process B is suspended until the signal(s) statement has finished execution. This guarantees that execution of B4 will start only after execution of A2 has completed.

By initializing the semaphore s to 0, we enforced the constraint that the first call to signal(s) had to complete before the first call to wait(s) would succeed.

Semaphores for Resource Allocation

Abstract problem:

- POOL of K resources
- Many processes, each needs resource for occasional uninterrupted period
- MUST guarantee that at most K resources are in use at any time.

Semaphore Solution:

In shared memory:
`semaphore s = K; // K resources`

Using resources:
`wait(s); // Allocate a resource
... // use it for a while
signal(s); // return it to pool`

Invariant: Semaphore value = number of resources left in pool

Another way to think about semaphores is as a management tool for a shared pool of K resources, where K is the initial value of the semaphore. You use the SIGNAL operation to add or return resources to the shared pool. And you use the WAIT operation to allocate a resource for your exclusive use.

At any given time, the value of the semaphore gives the number of unallocated resources still available in the shared pool.

Note that the WAIT and SIGNAL operations can be in the same process, or they may be in different processes, depending on when the resource is allocated and returned.

Bounded Buffer Problem w/ Semaphores

SHARED MEMORY:

```
char buf[N];      /* The buffer */  
int in=0, out=0;  
semaphore chars=0;
```

PRODUCER:

```
send(char c)  
{  
    buf[in] = c;  
    in = (in+1)%N;  
    signal(chars);  
}
```

CONSUMER:

```
char rcv()  
{  
    char c;  
    wait(chars);  
    c = buf[out];  
    out = (out+1)%N;  
    return c;  
}
```



Does
this
work?

PRECEDENCE managed by semaphore: $\text{send}_i \prec \text{rcv}_i$
RESOURCE managed by semaphore chars: # of chars in buf

We can use semaphores to manage our N-character FIFO buffer. Here we've defined a semaphore CHARS and initialized it to 0. The value of CHARS will tell us how many characters are in the buffer.

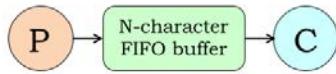
So SEND does a signal(CHARS) after it has added a character to the buffer, indicating the buffer now contains an additional character.

And RCV does a wait(CHARS) to ensure the buffer has at least one character before reading from the buffer.

Since CHARS was initialized to 0, we've enforced the constraint that the i^{th} call to signal(CHARS) precedes the completion of the i^{th} call to wait(CHARS). In other words, RCV can't consume a character until it has been placed in the buffer by SEND.

Does this mean our producer and consumer are now properly synchronized? Almost! Using the CHARS semaphore, we implemented *one* of the two precedence constraints we identified as being necessary for correct operation. Next we'll see how to implement the other precedence constraint.

Flow Control Problems



Q: What keeps PRODUCER from putting $N+1$ characters into the N -character buffer?

A: Nothing.

Result: OVERFLOW.

WHAT we've got thus far:

$\text{send}_i \prec \text{recv}_i$

WHAT we still need:

$\text{recv}_i \prec \text{send}_{i+N}$

What keeps the producer from putting more than N characters into the N -character buffer? Nothing. Oops, the producer can start to overwrite characters placed in the buffer earlier even though they haven't yet been read by the consumer. This is called buffer overflow and the sequence of characters transmitted from producer to consumer becomes hopelessly corrupted.

What we've guaranteed so far is that the consumer can read a character only after the producer has placed it in the buffer, *i.e.*, the consumer can't read from an empty buffer.

What we still need to guarantee is that the producer can't get too far ahead of the consumer. Since the buffer holds at most N characters, the producer can't send the $(i+N)^{\text{th}}$ character until the consumer has read the i^{th} character.

Bounded Buffer Problem w/ more Semaphores

SHARED MEMORY:

```
char buf[N];           /* The buffer */  
int in=0, out=0;  
semaphore chars=0, space=N;
```

PRODUCER:

```
send(char c)  
{  
    wait(space);  
    buf[in] = c;  
    in = (in+1)%N;  
    signal(chars);  
}
```

CONSUMER:

```
char rcv()  
{  
    char c;  
    wait(chars);  
    c = buf[out];  
    out = (out+1)%N;  
    signal(space);  
    return c;  
}
```

Resources managed by semaphore: characters in FIFO,
spaces in FIFO. Works with single producer, consumer. But
what about multiple producers and consumers?

Here we've added a second semaphore, SPACES, to manage the number of spaces in the buffer. Initially the buffer is empty, so it has N spaces. The producer must WAIT for a space to be available. When SPACES is non-zero, the WAIT succeeds, decrementing the number of available spaces by one and then the producer fills that space with the next character.

The consumer signals the availability of another space after it reads a character from the buffer.

There's a nice symmetry here. The producer consumes spaces and produces characters. The consumer consumes characters and produces spaces. Semaphores are used to track the availability of both resources (*i.e.*, characters and spaces), synchronizing the execution of the producer and consumer.

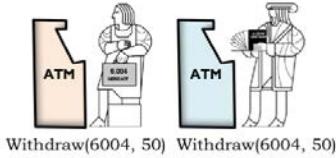
This works great when there is a single producer process and a single consumer process. Next we'll think about what will happen if we have multiple producers and multiple consumers.

Simultaneous Transactions

Suppose you and your friend visit the ATM at exactly the same time, and remove \$50 from your account. What happens?

```
Debit(int account, int amount) {  
    t = balance[account];  
    balance[account] = t - amount;  
}
```

What is *supposed* to happen?



Process # 1	Process #2
LD(R10, balance, R0)	
SUB(R0, R1, R0)	
ST(R0, balance, R10)	
...	...
	LD(R10, balance, R0)
	SUB(R0, R1, R0)
	ST(R0, balance, R10)

NET: You have \$100, and your bank balance is \$100 less.

Let's take a moment to look at a different example. Automated teller machines allow bank customers to perform a variety of transactions: deposits, withdrawals, transfers, etc. Let's consider what happens when two customers try to withdraw \$50 from the same account at the same time.

A portion of the bank's code for a withdrawal transaction is shown in the upper right. This code is responsible for adjusting the account balance to reflect the amount of the withdrawal. Presumably the check to see if there is sufficient funds has already happened.

What's supposed to happen? Let's assume that the bank is using a separate process to handle each transaction, so the two withdrawal transactions cause two different processes to be created, each of which will run the Debit code. If each of the calls to Debit run to completion without interruption, we get the desired outcome: the first transaction debits the account by \$50, then the second transaction does the same. The net result is that you and your friend have \$100 and the balance is \$100 less.

So far, so good.

But, What If...

Process # 1	Process #2
LD(R10, balance, R0)	
...	
	LD(R10, balance, R0)
	SUB(R0, R1, R0)
	ST(R0, balance, R10)
	...
	SUB(R0, R1, R0)
	ST(R0, balance, R10)
	...

NET: You have \$100 and your bank balance is \$50 less!



We need to be careful when writing concurrent programs. In particular, when modifying shared data.

For certain code segments, called **CRITICAL SECTIONS**, we would like to ensure that no two executions overlap.

This constraint is called **MUTUAL EXCLUSION**.

Solution: embed critical sections in wrappers (e.g., "transactions") that guarantee their atomicity, i.e., make them appear to be single, instantaneous operations.

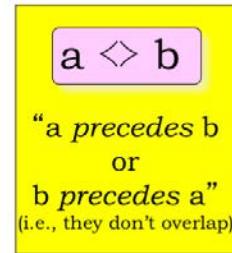
But what if the process for the first transaction is interrupted just after it's read the balance? The second process subtracts \$50 from the balance, completing that transaction. Now the first process resumes, using the now out-of-date balance it loaded just before being interrupted. The net result is that you and your friend have \$100, but the balance has only been debited by \$50.

The moral of the story is that we need to be careful when writing code that reads and writes shared data since other processes might modify the data in the middle of our execution. When, say, updating a shared memory location, we'll need to LD the current value, modify it, then ST the updated value. We would like to ensure that no other processes access the shared location between the start of the LD and the completion of the ST. The LD/modify/ST code sequence is what we call a "critical section." We need to arrange that other processes attempting to execute the same critical section are delayed until our execution is complete. This constraint is called "mutual exclusion," i.e., only one process at a time can be executing code in the same critical section.

Once we've identified critical sections, we'll use semaphores to guarantee they execute atomically, i.e., that once execution of the critical section begins, no other process will be able to enter the critical section until the execution is complete. The combination of the semaphore to enforce the mutual exclusion constraint and the critical section of code implement what's called a "transaction". A transaction can perform multiple reads and writes of shared data with the guarantee that none of the data will be read or written by other processes while the transaction is in progress.

Semaphores for Mutual Exclusion

```
semaphore lock = 1;  
  
Debit(int account, int amount) {  
    wait(lock); // Wait for exclusive access  
    t = balance[account];  
    balance[account] = t - amount;  
    signal(lock); // Finished with lock  
}
```



RESOURCE managed by “lock” semaphore:
Access to critical section

ISSUES:

Granularity of lock

- 1 lock for whole balance database?
- 1 lock per account?
- 1 lock for all accounts ending in 004



Look up “database”
on Wikipedia to
learn about systems
that support
efficient
transactions on
shared data.

Here's the original code to Debit, which we'll modify by adding a LOCK semaphore. In this case, the resource controlled by the semaphore is the right to run the code in the critical section. By initializing LOCK to 1, we're saying that at most one process can execute the critical section at a time.

A process running the Debit code WAITS on the LOCK semaphore. If the value of LOCK is 1, the WAIT will decrement value of LOCK to 0 and let the process enter the critical section. This is called acquiring the lock. If the value of LOCK is 0, some other process has acquired the lock and is executing the critical section and our execution is suspended until the LOCK value is non-zero.

When the process completes execution of the critical section, it releases the LOCK with a call to SIGNAL, which will allow other processes to enter the critical section. If there are multiple WAITing processes, only one will be able to acquire the lock, and the others will still have to wait their turn.

Used in this manner, semaphores are implementing a mutual exclusion constraint, *i.e.*, there's a guarantee that two executions of the critical section cannot overlap. Note that if multiple processes need to execute the critical section, they may run in any order and the only guarantee is that their executions will not overlap.

There are some interesting engineering issues to consider. There's the question of the granularity of the lock, *i.e.*, what shared data is controlled by the lock? In our bank example, should there be one lock controlling access to the balance for all accounts? That would mean that no one could access any balance while a transaction was in progress. That would mean that transactions accessing different accounts would have to run one after the other even though they're accessing different data. So one lock for all the balances would introduce unnecessary precedence constraints, greatly slowing the rate at which transactions could be processed.

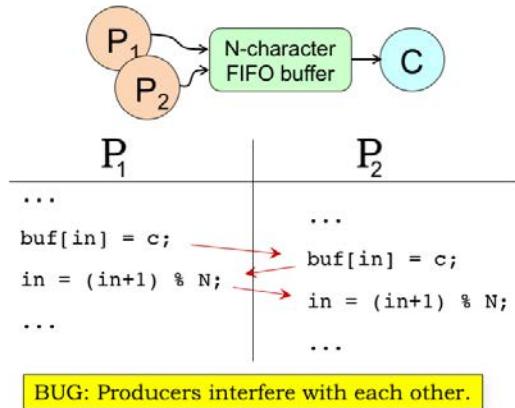
Since the guarantee we need is that we shouldn't permit multiple simultaneous transactions on the same account, it would make more sense to have a separate lock for each account, and change the Debit code to acquire the account's lock before proceeding. That will only delay transactions that truly overlap, an important efficiency consideration for a large system processing many thousands of mostly non-overlapping transactions each second.

Of course, having per-account locks would mean a lot of locks! If that's a concern, we can adopt a compromise strategy of having locks that protect groups of accounts, *e.g.*, accounts with same last three digits in the account number. That would mean we'd only need 1000 locks, which would allow up to 1000 transactions to happen simultaneously.

The notion of transactions on shared data is so useful that we often use a separate system called a database that provides the desired functionality. Database systems are engineered to provide low-latency access to shared data, providing the appropriate transactional semantics. The design and implementation of databases and transactions is pretty interesting - to follow up, I recommend reading about databases on the web.

Producer/Consumer Atomicity Problems

Consider multiple PRODUCER processes:



Returning to our producer/consumer example, we see that if multiple producers are trying to insert characters into the buffer at the same time, it's possible that their execution may overlap in a way that causes characters to be overwritten and/or the index to be improperly incremented.

We just saw this bug in the bank example: the producer code contains a critical section of code that accesses the FIFO buffer and we need to ensure that the critical section is executed atomically.

Bounded Buffer Problem w/ even more Semaphores

SHARED MEMORY:

```
char buf[N];           /* The buffer */
int in=0, out=0;
semaphore chars=0, space=N;
semaphore lock=1;
```

PRODUCER:

```
send(char c)
{
    wait(space);
    wait(lock);
    buf[in] = c;
    in = (in+1)%N;
    signal(lock);
    signal(chars);
}
```

CONSUMER:

```
char rcv()
{
    char c;
    wait(chars);
    wait(lock);
    c = buf[out];
    out = (out+1)%N;
    signal(lock);
    signal(space);
    return c;
}
```

Here we've added a third semaphore, called LOCK, to implement the necessary mutual exclusion constraint for the critical section of code that inserts characters into the FIFO buffer. With this modification, the system will now work correctly when there are multiple producer processes.

There's a similar issue with multiple consumers, so we've used the same LOCK to protect the critical section for reading from the buffer in the RCV code.

Using the same LOCK for producers and consumers will work, but does introduce unnecessary precedence constraints since producers and consumers use different indices, *i.e.*, IN for producers and OUT for consumers. To solve this problem we could use two locks: one for producers and one for consumers.

The Power of Semaphores

SHARED MEMORY:

```
char buf[N];           /* The buffer */
int in=0, out=0;
semaphore chars=0, space=N;
semaphore lock=1;
```

PRODUCER:

```
send(char c)
{
    wait(space);
    wait(lock)
    buf[in] = c;
    in = (in+1)%N;
    signal(lock);
    signal(chars);
}
```

CONSUMER:

```
char rcv()
{
    char c;
    wait(chars);
    wait(lock);
    c = buf[out];
    out = (out+1)%N;
    signal(lock);
    signal(space);
    return c;
}
```

A single synchronization primitive that enforces both:

Precedence relationships:

$$\text{send}_i \prec \text{rcv}_i$$
$$\text{rcv}_i \prec \text{send}_{i+N}$$

Mutual-exclusion relationships:

protect variables
in and out

Semaphores are a pretty handy Swiss army knife when it comes to dealing with synchronization issues. When WAIT and SIGNAL appear in different processes, the semaphore ensures the correct execution timing between processes. In our example, we used two semaphores to ensure that consumers can't read from an empty buffer and that producers can't write into a full buffer.

We also used semaphores to ensure that execution of critical sections - in our example, updates of the indices IN and OUT - were guaranteed to be atomic. In other words, that the sequence of reads and writes needed to increment a shared index would not be interrupted by another process between the initial read of the index and the final write.

Semaphore Implementation

Semaphores are themselves shared data and implementing WAIT and SIGNAL operations will require read/modify/write sequences that must be executed as critical sections. So how do we guarantee mutual exclusion in these particular critical sections without using semaphores?

Approaches:

- SVC implementation, using atomicity of kernel handlers.
Works in timeshared processor sharing a single uninterruptible kernel.
- Implementation of a simple lock using a special instruction (e.g. "test and set"), depends on atomicity of single instruction execution. Works with shared-bus multiprocessors supporting atomic read-modify-write bus transactions. Using a simple lock to implement critical sections, we can use software to implement other semaphore functionality.
- Implementation using atomicity of individual read or write operations. For example, see "Dekker's Algorithm" on Wikipedia.



Now let's figure out how to implement semaphores. They are themselves shared data and implementing the WAIT and SIGNAL operations will require read/modify/write sequences that must be executed as critical sections. Normally we'd use a lock semaphore to implement the mutual exclusion constraint for critical sections. But obviously we can't use semaphores to implement semaphores! We have what's called a bootstrapping problem: we need to implement the required functionality from scratch.

Happily, if we're running on a timeshared processor with an uninterruptible OS kernel, we can use the supervisor call (SVC) mechanism to implement the required functionality.

We can also extend the ISA to include a special test-and-set instruction that will let us implement a simple lock semaphore, which can then be used to protect critical sections that implement more complex semaphore semantics. Single instructions are inherently atomic and, in a multi-core processor, will do what we want if the shared main memory supports both reading the old value and writing a new value to a specific memory location as a single memory access.

There are other, more complex, software-only solutions that rely only on the atomicity of individual reads and writes to implement a simple lock. For example, see "Dekker's Algorithm" on Wikipedia.

We'll look in more detail at the first two approaches.

Semaphores as a Supervisor Call

```
wait_h( ) {
    int *addr;
    addr = Vtop(User.Reg[R0]); // get arg
    if (*addr <= 0) {
        User.Reg[XP] = User.Reg[XP] - 4;
        sleep(addr);
    } else
        *addr = *addr - 1;
}

signal_h( ) {
    int *addr;
    addr = Vtop(User.Reg[R0]); // get arg
    *addr = *addr + 1;
    wakeup(addr);
}
```

Calling sequence:
...
// put address of lock
// into R0
CMOVE(lock, R0)
SVC(WAIT) or SVC(SIGNAL)

SVC call is not
interruptible since it is
executed in kernel
mode.

Here are the OS handlers for the WAIT and SIGNAL supervisor calls. Since SVCs are run kernel mode, they can't be interrupted, so the handler code is naturally executed as a critical section.

Both handlers expect the address of the semaphore location to be passed as an argument in the user's R0. The WAIT handler checks the semaphore's value and if it's non-zero, the value is decremented and the handler resumes execution of the user's program at the instruction following the WAIT SVC. If the semaphore is 0, the code arranges to re-execute the WAIT SVC when the user program resumes execution and then calls SLEEP to mark the process as inactive until the corresponding WAKEUP call is made.

The SIGNAL handler is simpler: it increments the semaphore value and calls WAKEUP to mark as active any processes that were WAITing for this particular semaphore.

Eventually the round-robin scheduler will select a process that was WAITing and it will be able to decrement the semaphore and proceed. Note that the code makes no provision for fairness, *i.e.*, there's no guarantee that a WAITing process will eventually succeed in finding the semaphore non-zero. The scheduler has a specific order in which it runs processes, so the next-in-sequence WAITing process will always get the semaphore even if there are later-in-sequence processes that have been WAITing longer. If fairness is desired, WAIT could maintain a queue of waiting processes and use the queue to determine which process is next in line, independent of scheduling order.

Hardware Support for Semaphores

```
TCLR(RA, literal, RC) test and clear location  
PC ← PC + 4  
EA ← Reg[RA] + literal  
Reg[RC] ← MEM[EA]  
MEM[EA] ← 0 } Atomicity guaranteed by memory
```

Executed ATOMICALLY (cannot be interrupted)
Can easily implement mutual exclusion using binary semaphore

```
wait:  TCLR(R31, lock, R0) } wait(lock)  
      BEQ(R0,wait)  
      ... critical section ...  
      CMOVE(1,R0)  
      ST(R0, lock, R31) } signal(lock)
```

Many ISAs support an instruction like the TEST-and-CLEAR instruction shown here. The TCLR instruction reads the current value of a memory location and then sets it to zero, all as a single operation. It's like a LD except that it zeros the memory location after reading its value.

To implement TCLR, the memory needs to support read-and-clear operations, as well as normal reads and writes.

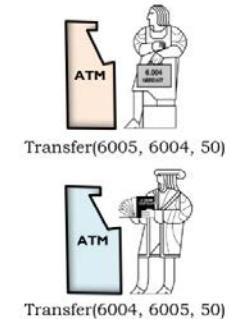
The assembly code at the bottom of the slide shows how to use TCLR to implement a simple lock. The program uses TCLR to access the value of the lock semaphore. If the returned value in RC is zero, then some other process has the lock and the program loops to try TCLR again. If the returned value is non-zero, the lock has been acquired and execution of the critical section can proceed. In this case, TCLR has also set the lock to zero, so that other processes will be prevented from entering the critical section.

When the critical section has finished executing, a ST instruction is used to set the semaphore to a non-zero value.

Synchronization: The Dark Side

The naïve use of synchronization constraints can introduce its own set of problems, particularly when a process requires access to more than one protected resource.

```
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]);  
}
```



DEADLOCK (aka “deadly embrace”)!

If the necessary synchronization requires acquiring more than one lock, there are some special considerations that need to be taken into account. For example, the code below implements the transfer of funds from one bank account to another. The code assumes there is a separate semaphore lock for each account and since it needs to adjust the balance of two accounts, it acquires the lock for each account.

Consider what happens if two customers try simultaneous transfers between their two accounts. The top customer will try to acquire the locks for accounts 6005 and 6004. The bottom customer tries to acquire the same locks, but in the opposite order. Once a customer has acquired both locks, the transfer code will complete, releasing the locks.

But what happens if the top customer acquires his first lock (for account 6005) and the bottom customer simultaneously acquires his first lock (for account 6004). So far, so good, but now each customer will be not be successful in acquiring their second lock, since those locks are already held by the other customer!

This situation is called a “deadlock” or “deadly embrace” because there is no way execution for either process will resume - both will wait indefinitely to acquire a lock that will never be available.

Obviously, synchronization involving multiple resources requires a bit more thought.

Dining Philosophers

Philosophers think deep thoughts, but have simple secular needs. When hungry, a group of N philosophers will sit around a table with N chopsticks interspersed between them. Food is served, and each philosopher enjoys a leisurely meal using the chopsticks on either side to eat.

They are exceedingly polite and patient, and each follows the following dining protocol:

PHILOSOPHER'S ALGORITHM:

- Take (wait for) LEFT stick
- Take (wait for) RIGHT stick
- EAT until sated
- Replace both sticks



The problem of deadlock is elegantly illustrated by the Dining Philosophers problem. Here there are, say, 5 philosophers waiting to eat. Each requires two chopsticks in order to proceed, and there are 5 chopsticks on the table.

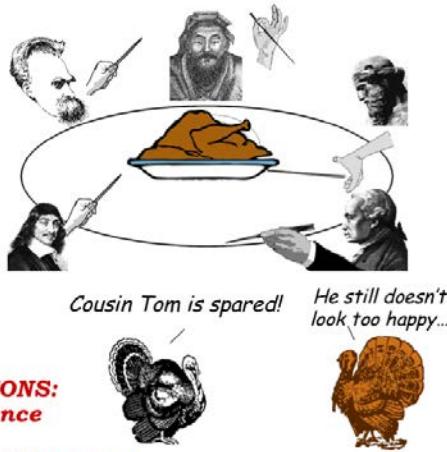
The philosophers follow a simple algorithm. First they pick up the chopstick on their left, then the chopstick on their right. When they have both chopsticks they eat until they're done, at which point they return both chopsticks to the table, perhaps enabling one of their neighbors to pick them up and begin eating. Again, we see the basic setup of needing two (or more) resources before the task can complete.

Deadlock!

No one can make progress because they are all waiting for an unavailable resource

CONDITIONS:

- 1) Mutual exclusion - only one process can hold a resource at a given time
- 2) Hold-and-wait - a process holds allocated resources while waiting for others
- 3) No preemption - a resource can not be removed from a process holding it
- 4) Circular Wait



Hopefully you can see the problem that may arise...

If all philosophers pick up the chopstick on their left, then all the chopsticks have been acquired, and none of the philosophers will be able to acquire their second chopstick and eat. Another deadlock!

Here are the conditions required for a deadlock:

1. Mutual exclusion, where a particular resource can only be acquired by one process at a time.
2. Hold-and-wait, where a process holds allocated resources while waiting to acquire the next resource.
3. No preemption, where a resource cannot be removed from the process which acquired it. Resources are only released after the process has completed its transaction.
4. Circular wait, where resources needed by one process are held by another, and vice versa.

How can we solve the problem of deadlocks when acquiring multiple resources? Either we avoid the problem to begin with, or we detect that deadlock has occurred and implement a recovery strategy. Both techniques are used in practice.

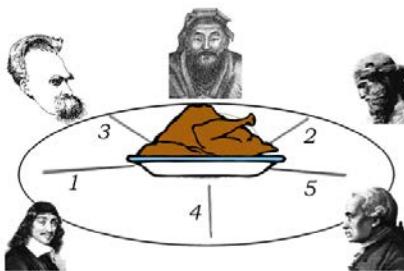
One Solution

KEY: Assign a unique number to each chopstick, request resources in globally consistent order:

New Algorithm:

- Take LOW stick
- Take HIGH stick
- EAT
- Replace both sticks.

SIMPLE PROOF:



Deadlock means that each philosopher is waiting for a resource held by some other philosopher ...

But, the philosopher holding the highest numbered chopstick can't be waiting for any other philosopher (no hold-and-wait cycle) ...

Thus, there can be no deadlock.

In the Dining Philosophers problem, deadlock can be avoided with a small modification to the algorithm. We start by assigning a unique number to each chopstick to establish a global ordering of all the resources, then rewrite the code to acquire resources using the global ordering to determine which resource to acquire first, which second, and so on.

With the chopsticks numbered, the philosophers pick up the lowest-numbered chopstick from either their left or right. Then they pick up the other, higher-numbered chopstick, eat, and then return the chopsticks to the table.

How does this avoid deadlock? Deadlock happens when all the chopsticks have been picked up but no philosopher can eat. If all the chopsticks have been picked up, that means some philosopher has picked up the highest-numbered chopstick and so must have earlier picked up the lower-numbered chopstick on his other side. So that philosopher can eat then return both chopsticks to the table, breaking the hold-and-wait cycle.

So if all the processes in the system can agree upon a global ordering for the resources they require, then acquire them in order, there will be no possibility of a deadlock caused by a hold-and-wait cycle.

Dealing With Deadlocks

Cooperating processes:

- Establish a fixed ordering to shared resources and require all requests to be made in the prescribed order

```
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]);  
}
```



Transfer(6005, 6004, 50)



Transfer(6004, 6005, 50)

Unconstrained processes:

- O/S discovers circular wait & kills waiting process
- Transaction model
- Hard problem

A global ordering is easy to arrange in our banking code for the transfer transaction. We'll modify the code to first acquire the lock for the lower-numbered account, then acquire the lock for the higher-numbered account. Now, both customers will first try to acquire the lock for the 6004 account. The customer that succeeds then can acquire the lock for the 6005 account and complete the transaction. The key to deadlock avoidance was that customers contended for the lock for the *first* resource they both needed - acquiring that lock ensured they would be able to acquire the remainder of the shared resources without fear that they would already be allocated to another process in a way that could cause a hold-and-wait cycle.

Establishing and using a global order for shared resources is possible when we can modify all processes to cooperate. Avoiding deadlock without changing the processes is a harder problem. For example, at the operating system level, it would be possible to modify the WAIT SVC to detect circular wait and terminate one of the WAITing processes, releasing its resources and breaking the deadlock.

The other strategy we mentioned was detection and recovery. Database systems detect when there's been an external access to the shared data used by a particular transaction, which causes the database to abort the transaction. When issuing a transaction to a database, the programmer specifies what should happen if the transaction is aborted, e.g., she can specify that the transaction be retried. The database remembers all the changes to shared data that happen during a transaction and only changes the master copy of the shared data when it is sure that the transaction will not be aborted, at which point the changes are committed to the database.

Summary

Communication among asynchronous processes requires synchronization....

- Precedence constraints: a partial ordering among operations
- Semaphores as a mechanism for enforcing precedence constraints
- Mutual exclusion (critical sections, atomic transactions) as a common compound precedence constraint
- Solving Mutual Exclusion via binary semaphores
- Synchronization *serializes* operations, limits parallel execution.

Many alternative synchronization mechanisms exist!

Deadlocks:

- Consequence of undisciplined use of synchronization mechanism
- Can be avoided in special cases, detected and corrected in others.

In summary, we saw that organizing an application as communicating processes is often a convenient way to go. We used semaphores to synchronize the execution of the different processes, providing guarantees that certain precedence constraints would be met, even between statements in different processes.

We also introduced the notion of critical code sections and mutual exclusion constraints that guaranteed that a code sequence would be executed without interruption by another process. We saw that semaphores could also be used to implement those mutual exclusion constraints.

Finally we discussed the problem of deadlock that can occur when multiple processes must acquire multiple shared resources, and we proposed several solutions based on a global ordering of resources or the ability to restart a transaction.

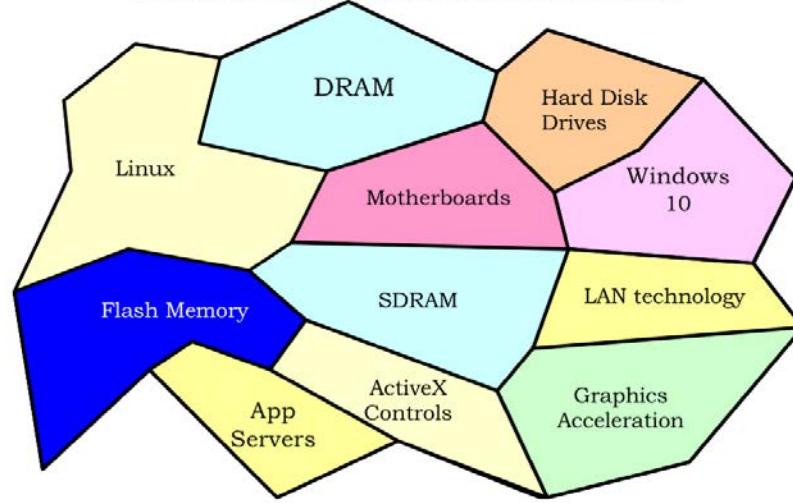
Synchronization primitives play a key role in the world of “big data” where there are vast amounts of shared data, or when trying to coordinate the execution of thousands of processes in the cloud. Understanding synchronization issues and their solutions is a key skill when writing most modern applications.

20: System-level communication

1. Computer System Technologies
2. Interfaces Last Forever
3. System Interfaces & Modularity
4. Buses, Interconnect, So...?
5. Electrical Model for Real Wires
6. Real-world Consequences
7. Space & Time Constraints
8. Gates, Wires, & Delays
9. Interface Standard: Backplane Bus
10. A Parallel Bus Transaction
11. Bus Lines as Transmission Lines
12. Meanwhile, Outside the Box...
13. Lessons learned: single driver; point-to-point
14. Lessons learned: clock recovery
15. Serial, Point-to-point Communications
16. Improving on the Bus
17. Communications in Today's Computers
18. Example serial link: PCI Express (PCIe)
19. Communication Topologies
20. Quadratic-cost Topologies
21. Mesh Topologies
22. Logarithmic-latency Networks
23. Communication Technologies: Latency
24. Communications Futures

Computer System Technologies

What is the most important part of this picture?



Computer systems bring together many technologies and harness them to provide fast execution of programs. Some of these technologies are relatively new, others have been with us for decades. Each of the system components comes with a detailed specification of their functionality and interface. The expectation is that system designers can engineer the system based on the component specifications without having to know the details of the implementations of each component. This is good since many of the underlying technologies change, often in ways that allow the components to become smaller, faster, cheaper, more energy efficient, and so on. Assuming the new components still implement same interfaces, they can be integrated into the system with very little effort.

The moral of this story is that the important part of the system architecture is the interfaces.

Technology comes & goes; interfaces last forever

- Interfaces typically deserve more engineering attention than the technologies they interface...
 - Abstraction: should outlast many technology generations
 - Often “virtualized” to extend beyond original function (e.g. memory, I/O, services, machines)
 - Represent more potential value to their proprietors than the technologies they connect.
- Interface sob stories:
 - Interface “warts”: Big/little Endian wars
 - Early IBM PC reliance on the exact signaling of 8086 chips
- ... and many success stories:
 - IBM 360 Instruction set architecture; Postscript; Compact Flash;
 - ...
 - TCP/IP-based packet networks

Our goal is to design interface specifications that can survive many generations of technological change. One approach to long-term survival is to base the specification on a useful abstraction that hides most, if not all, of the low-level implementation details. Operating systems provide many interfaces that have remained stable for many years. For example, network interfaces that reliably deliver streams of bytes to named hosts, hiding the details of packets, sockets, error detection and recovery, etc. Or windowing and graphics systems that render complex images, shielding the application from details about the underlying graphics engine. Or journaled file systems that behind-the-scenes defend against corruption in the secondary storage arrays.

Basically, we’re long since past the point where we can afford to start from scratch each time the integrated circuit gurus are able to double the number of transistors on a chip, or the communication wizards figure out how to go from 1GHz networks to 10GHz networks, or the memory mavens are able to increase the size of main memory by a factor of 4. The interfaces that insulate us from technological change are critical to ensure that improved technology isn’t a constant source of disruption.

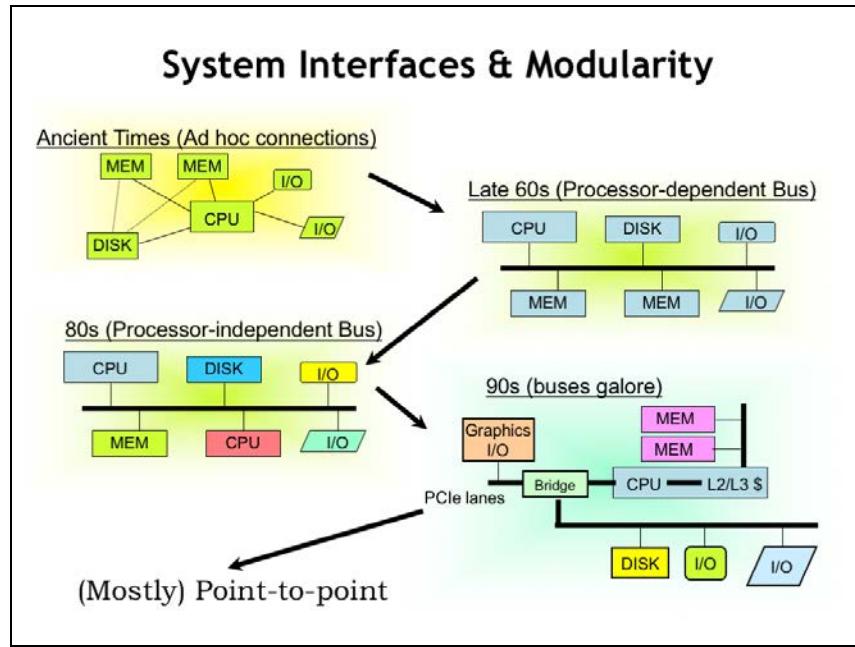
There are some famous examples of where an apparently convenient choice of interface has had embarrassing long-term consequences. For example, back in the days of stand-alone computing, different ISAs made different choices on how to store multi-byte numeric values in main memory. IBM architectures store the most-significant byte in the lowest address (so called “big endian”), while Intel’s x86 architectures store the least-significant byte first (so called “little endian”). But this leads to all sorts of complications in a networked world where numeric data is often transferred from system to system. This is a prime example of a locally-optimal choice having an unfortunate global impact. As the phrase goes: “a moment of convenience, a lifetime of regret.”

Another example is the system-level communication strategy chosen for the first IBM PC, the original personal computer based Intel CPU chips. IBM built their expansion bus for adding I/O peripherals, memory cards, etc., by simply using the interface signals provided by then-current x86 CPU. So the width of the data bus, the number of address pins, the data-transfer protocol, etc. were exactly as designed for interfacing to that particular CPU. A logical choice since it got the job done while keeping costs as low a possible. But that choice quickly proved unfortunate as newer, higher-performance CPUs were introduced, capable of addressing more memory or providing 32-bit instead of 16-bit

external data paths. So system architects were forced into offering customers the Hobson's choice of crippling system throughput for the sake of backward compatibility, or discarding the networking card they bought last year since it was now incompatible with this year's system.

But there are success stories too. The System/360 interfaces chosen by IBM in the early 1960s carried over to the System/370 in the 70's and 80's and to the Enterprise System Architecture/390 of the 90's. Customers had the expectation that software written for the earlier machines would continue to work on the newer systems and IBM was able to fulfill that expectation.

Maybe the most notable long-term interface success is the design the TCP and IP network protocols in the early 70's, which formed the basis for most packet-based network communication. A recent refresh expanded the network addresses from 32 to 128 bits, but that was largely invisible to applications using the network. It was a remarkably prescient set of engineering choices that stood the test of time for over four decades of exponential growth in network connectivity.



Today's lecture topic is figuring out the appropriate interface choices for interconnecting system components. In the earliest systems these connections were very ad hoc in the sense that the protocols and physical implementation were chosen independently for each connection that had to be made. The cable connecting the CPU box to the memory box (yes, in those days, they lived in separate 19-inch racks!) was different than the cable connecting the CPU to the disk.

Improving circuit technologies allowed system components to shrink from cabinet-size to board-size and system engineers designed a modular packaging scheme that allowed users to mix-and-match board types that plugged into a communication backplane. The protocols and signals on the backplane reflected the different choices made by each vendor — IBM boards didn't plug into Digital Equipment backplanes, and vice versa.

This evolved into some standardized communication backplanes that allowed users to do their own system integration, choosing different vendors for their CPU, memory, networking, etc. Healthy competition quickly brought prices down and drove innovation. However this promising development was overtaken by rapidly improving performance, which required communication bandwidths that simply could not be supported across a multi-board backplane.

These demands for higher performance and the ability to integrate many different communication channels into a single chip, lead to a proliferation of different channels. In many ways, the system architecture was reminiscent of the original systems — ad-hoc purpose-built communication channels specialized to a specific task.

As we'll see, engineering considerations have led to the widespread adoption of general-purpose unidirectional point-to-point communication channels. There are still several types of channels depending on the required performance, the distance travelled, etc., but asynchronous point-to-point channels have mostly replaced the synchronous multi-signal channels of earlier systems.

Most system-level communications involve signaling over wires, so next we'll look into some the engineering issues we've had to deal with as communication speeds have increased from kHz to GHz.

Buses, Interconnect, So...?

Aren't communication channels simply logic circuits with long wires?

Wires – circuit theorist's view:

Equipotential "nodes" of a circuit.

Instant propagation of v , i over entire node.

"distance" abstracted out of design model.

Time issues dictated by RLC elements; wires are timeless.



Wires – interconnect engineer's view:

Transmission lines.

Finite signal propagation velocity.

Distance matters.

Time matters.

Reality matters.



So, how hard can it be to build a communication channel? Aren't they just logic circuits with a long wire that runs from one component to another?

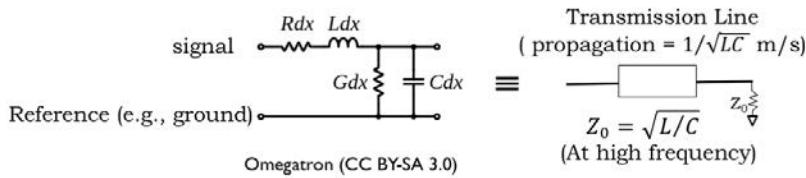
A circuit theorist would tell you that wires in a schematic diagram are intended to represent the equipotential nodes of the circuit, which are used to connect component terminals. In this simple model, a wire has the same voltage at all points and any changes in the voltage or current at one component terminal is instantly propagated to the other component terminals connected to the same wire. The notion of distance is abstracted out of our circuit models: terminals are either connected by a wire, or they're not. If there are resistances, capacitances, and inductances that need to be accounted for, the necessary components can be added to the circuit model. Wires are timeless. They are used to show how components connect, but they aren't themselves components.

In fact, thinking of wires as equipotential nodes is a very workable model when the rate of change of the voltage on the wire is slow compared to the time it takes for an electromagnetic wave to propagate down the wire. Only as circuit speeds have increased with advances in integrated circuit technologies did this rule-of-thumb start to be violated in logic circuits where the components were separated by at most 10's of inches.

In fact, it has been known since the late 1800s that changes in voltage levels take finite time to propagate down a wire. Oliver Heaviside was a self-taught English electrical engineer who, in the 1880's, developed a set of "telegrapher's equations" that described how signals propagate down wires. Using these, he was able to show how to improve the rate of transmission on then new transatlantic telegraph cable by a factor of 10.

We now know that for high-speed signaling we have to treat wires as transmission lines, which we'll say more about in the next few slides. In this domain, the distance between components and hence the lengths of wires is of critical concern if we want to correctly predict the performance of our circuits. Distance and signal propagation matter — real-world wires are, in fact, fairly complex components!

Electrical Model for Real Wires



	Description	On chip	On PCB
R	Resistance of conductor	150 kΩ/m	5 Ω/m
L	Self-inductance of conductor (due to magnetic field induced by current)	600 nH/m	300 nH/m
C	Capacitance between signal and ground	200 pF/m	100 pF/m
G	Conductance between signal and ground (through insulator)	small	small

http://cva.stanford.edu/books/dig_sys_engr/lectures/

Here's an electrical model for an infinitesimally small segment of a real-world wire. An actual wire is correctly modeled by imagining many copies of the model shown here connected end-to-end. The signal, *i.e.*, the voltage on the wire, is measured with respect to the reference node which is also shown in the model.

There are 4 parameters that characterize the behavior of the wire. R tells us the resistance of the conductor. It's usually negligible for the wiring on printed circuit boards, but it can be significant for long wires in integrated circuits. L represents the self-inductance of the conductor, which characterizes how much energy will be absorbed by the wire's magnetic fields when the current flowing through the wire changes. The conductor and reference node are separated by some sort insulator (which might be just air!) and hence form a capacitor with capacitance C. Finally, the conductance G represents the current that leaks through the insulator. Usually this is quite small.

The table shows the parameter values we might measure for wires inside an integrated circuit and on printed circuit boards.

If we analyze what happens when sending signals down the wire, we can describe the behavior of the wires using a single component called a transmission line which has a characteristic complex-valued impedance Z_0 . At high signaling frequencies and over the distances found on-chip or on circuit boards, such as one might find in a modern digital system, the transmission line is lossless, and voltage changes ("steps") propagate down the wire at the rate of $1/\sqrt{LC}$ meters per second. Using the values given here for a printed circuit board, the characteristic impedance is approximately 50 ohms and the speed of propagation is about 18 cm (7 in.) per ns.

To send digital information from one component to another, we change the voltage on the connecting wire, and that voltage step propagates from the sender to the receiver. We have to pay some attention to what happens to that energy front when it gets to the end of the wire! If we do nothing to absorb that energy, conservation laws tell us that it reflects off the end of the wire as an "echo" and soon our wire will be full of echoes from previous voltage steps!

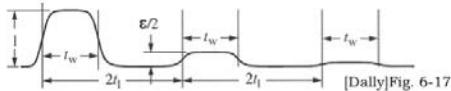
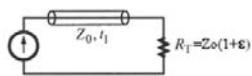
To prevent these echoes we have to terminate the wire with a resistance to ground that matches the characteristic impedance of the transmission line. If the signal can propagate in both directions, we'll need termination at both ends.

What this model is telling us is the time it takes to transmit information from one component to another and that we have to be careful to absorb the energy associated with the transmission when the information has reached its destination.

Real-World Consequences

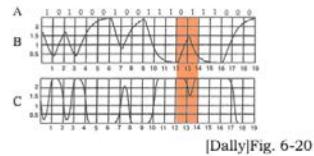
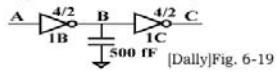
ΔV from energy storage left over from earlier signaling on the wire:

- **transmission line discontinuities**
(reflections off of impedance mismatches and terminations)



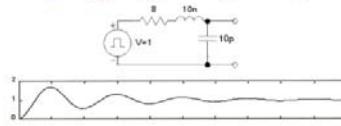
[Dally]Fig. 6-17

- **charge storage in RC circuit**
(narrow pulses are lost due to incomplete transitions)



[Dally]Fig. 6-20

- **RLC ringing** (triggered by voltage steps)



Fix: slower operation, limiting voltage swings and slew rates

Dally, W.J., Poulton, J.W., *Digital Systems Engineering*, 1998

With that little bit of theory as background, we're in a position to describe the real-world consequences of poor engineering of our signal wires. The key observation is that unless we're careful there can still be energy left over from previous transmissions that might corrupt the current transmission. The general fix to this problem is time, *i.e.*, giving the transmitted value a longer time to settle to an interference-free value. But slowing down isn't usually acceptable in high-performance systems, so we have to do our best to minimize these energy storage effects.

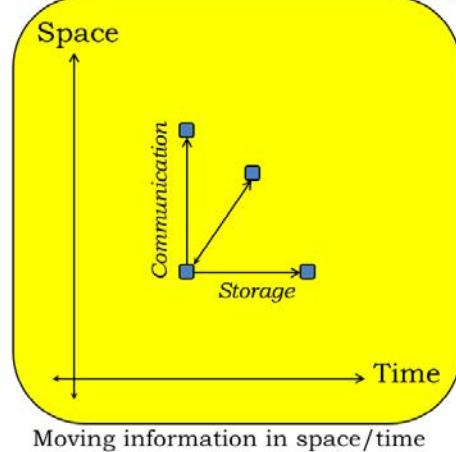
If there the termination isn't exactly right, we'll get some reflections from any voltage step reaching the end of the wire, and it will take a while for the echoes to die out. In fact, as we'll see, energy will reflect off of any impedance discontinuity, which mean we'll want to minimize the number of such discontinuities.

We need to be careful to allow sufficient time for signals to reach valid logic levels. The shaded region shows a transition of the wire A from 1 to 0 to 1. The first inverter is trying to produce a 1-output from the initial input transition to 0, but doesn't have sufficient time to complete the transition on wire B before the input changes again. This leads to a runt pulse on wire C, the output of the second inverter, and we see that the sequence of bits on A has been corrupted by the time the signal reaches C. This problem was caused by the energy storage in the capacitance of the wire between the inverters, which will limit the speed at which we can run the logic.

And here see we what happens when a large voltage step triggers oscillations in wire, called ringing, voltages due to the wire's inductance and capacitance. The graph shows it takes some time before the ringing damps to the point that we have a reliable digital signal. The ringing can be diminished by spreading the voltage step over a longer time.

The key idea here is that by paying close attention to the design of our wiring and the drivers that put information onto the wire, we can minimize the performance implications of these energy-storage effects.

Space & Time Constraints



Fundamental Physical Constraints:

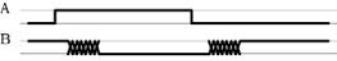
- Bounds on propagation speeds
 - Signals travel $\sim 18\text{cm/ns}$ on PCB
- Bounds on device density
 - Must be finite distances between components
- Bounds on flow of charge
 - finite currents \rightarrow finite rise/fall times
 - wire delays depend on loading

Okay, enough electrical engineering! Suppose we have some information in our system. If we preserve that information over time, we call that storage. If we send that information to another component, we call that communication. In the real world, we've seen that communication takes time and we have to budget for that time in our system timing. Our engineering has to accommodate the fundamental bounds on propagating speeds, distances between components and how fast we can change wire voltages without triggering the effects we saw on the previous slide. The upshot: our timing models will have to account for wire delays.

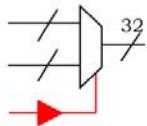
Gates, Wires, & Delays

Our t_{pd} , t_{cd} timing model

- bundles delays into device specs
- ignores loading, wire lengths



Reality check:



- long / heavily-loaded outputs will be slower
- can bundle internal wire delays into t_{pd} of a device; but external load matters!
- partial fixes: buffers, distribution trees
- optimizing performance requires attention to loading issues (You'll see this in the design project!).

Particularly problematic: system-wide interconnect!

Earlier in the course, we had a simple timing model that assigned a fixed propagation delay, t_{PD} , to the time it took for the output of a logic gate to reflect a change to the gate's input.

We'll need to change our timing model to account for delay of transmitting the output of a logic gate to the next components. The timing will be load dependent, so signals that connect to the inputs of many other gates will be slower than signals that connect to only one other gate. The Jade simulator takes the loading of a gate's output signal into account when calculating the effective propagation delay of the gate.

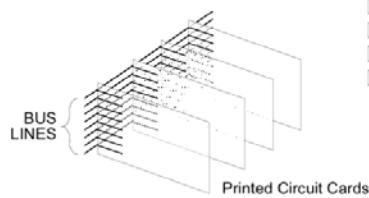
We can improve propagation delays by reducing the number of loads on output signals or by using specially-design gates called buffers (the component shown in red) to drive signals that have very large loads. A common task when optimizing the performance of a circuit is to track down heavily loaded and hence slow wires and re-engineering the circuit to make them faster.

Today our concern is wires used to connect components at the system level. So next we'll turn our attention to possible designs for system-level interconnect and the issues that might arise.

Interface Standard: Backplane Bus

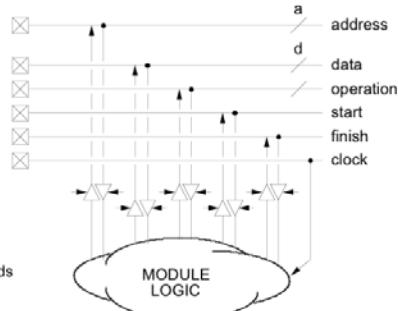
Modular cards that plug into a common backplane:

CPUs
Memories
Bulk storage
I/O devices
S/W?



The backplane provides:

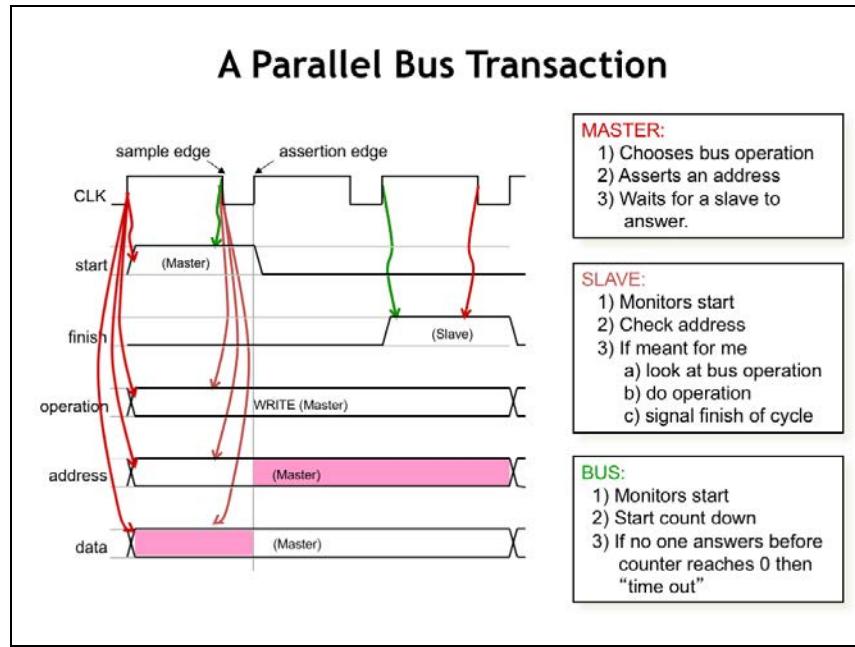
Power
Common system clock
Wires for communication



If we want our system to be modular and expandable, how should its design accommodate components that the user might add at a later time? For many years the approach was to provide a way to plug additional printed circuit boards into the main “motherboard” that holds the CPU, memory, and the initial collection of I/O components. The socket on the motherboard connects the circuitry on the add-in card to the signals on the motherboard that allow the CPU to communicate with the add-in card. These signals include power and a clock signal used to time the communication, along with the following.

- * Address wires to select different communication end points on the add-in card. The end points might include memory locations, control registers, diagnostic ports, etc.
- * Data wires for transferring data to and from the CPU. In older systems, there would many data wires to support byte- or word-width data transfers.
- * Some number of control wires that tell the add-in card when a particular transfer has started and that allow the add-in card to indicate when it has responded.

If there are multiple slots for plugging in multiple add-in cards, the same signals might be connected to all the cards and the address wires would be used to sort out which transfers were intended for which cards. Collectively these signals are referred to as the system bus. “Bus” is system architect jargon for a collection of wires used to transfer data using a pre-determined communication protocol.



Here's an example of how a bus transaction might work.

The CLK signal is used to time when signals are placed on the bus wires (at the assertion edge of CLK) and when they're read by the recipient (at the sample edge of the CLK). The timing of the clock waveform is designed to allow enough time for the signals to propagate down the bus and reach valid logic levels at all the receivers.

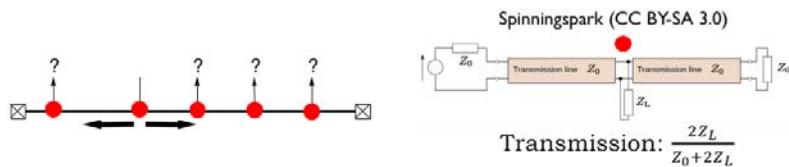
The component initiating the transaction is called the bus master who is said to "own" the bus. Most buses provide a mechanism for transferring ownership from one component to another. The master sets the bus lines to indicate the desired operation (read, write, block transfer, etc.), the address of the recipient, and, in the case of a write operation, the data to be sent to the recipient.

The intended recipient, called the slave, is watching the bus lines looking for its address at each sample edge. When it sees a transaction for itself, the slave performs the requested operation, using a bus signal to indicate when the operation is complete. On completion it may use the data wires to return information to the master.

The bus itself may include circuitry to look for transactions where the slave isn't responding and, after an appropriate interval, generate an error response so the master can take the appropriate action.

This sort of bus architecture proved to be a very workable design for accommodating add-in cards as long as the rate of transactions wasn't too fast, say less than 50 MHz.

Bus Lines as Transmission Lines



ANALOG ISSUES:

- Propagation times
 - Signals travel at ~18 cm/ns on a PCB
- Skew
 - Different points along the bus see the signals at different times
 - Bits of data propagate at slightly different rates along parallel wires
- Reflections & standing waves
 - At each interface (places where the propagation medium changes) the signal may reflect if the impedances are not matched.
 - Make a transition on a long line – may have to wait many transition times for echoes to subside.

$$\text{Reflection: } \frac{-Z_0}{Z_0 + 2Z_L}$$

https://en.wikipedia.org/wiki/Reflections_of_signals_on_conducting_lines

But as system speeds increased, transaction rates had to increase to keep system performance at acceptable levels, so the time for each transaction got smaller. With less time for signaling on the bus wires, various effects began loom large.

If the clock had too short a period, there wasn't enough time for the master to see the assertion edge, enable its drivers, have the signal propagate down a long bus to the intended receiver and be stable at each receiver for long enough before the sample edge.

Another problem was that the clock signal would arrive at different cards at different times. So a card with an early-arriving clock might decide it was its turn to start driving the bus signals, while a card with a late-arriving clock might still be driving the bus from the previous cycle. These momentary conflicts between drivers could add huge amounts of electrical noise to the system.

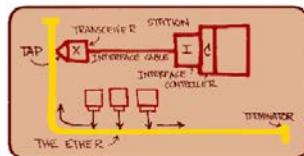
Another big issue is that energy would reflect off all the small impedance discontinuities caused by the bus connectors. If there were many connectors, there would be many small echoes which would corrupt the signal seen by various receivers. The equations in the upper right show how much of the signal energy is transmitted and how much is reflected at each discontinuity. The net effect was like trying to talk very fast while yelling into the Grand Canyon — the echoes could distort the message beyond recognition unless sufficient time was allocated between words for the echoes to die away.

Eventually buses were relegated to relatively low-speed communication tasks and a different approach had to be developed for high-speed communication.

Meanwhile, Outside the Box...

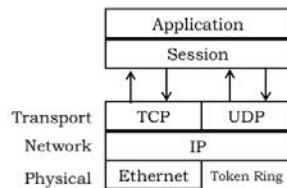
The network as an interface standard.

ETHERNET: In the mid-70's Bob Metcalf (at Xerox PARC, an MIT alum) devised a bus for networking computers together.



- Inspired by Aloha net (radio)
- COAX replaced "ether"
- Bit-serial (optimized for long wires)
- Variable-length "packets":
 - self-paced data (no clock, skew!)
 - header (dest), data bits, checksum
- Issues: sharing, contention, arbitration, "backoff"

IDEA: Protocol "layers" that isolate application-level interface from low-level physical devices:



Network technologies were developed to connect components (in this case individual computer systems) separated by larger distances, *i.e.*, distances measured in meters instead of centimeters. Communicating over these larger distances led to different design tradeoffs. In early networks, information was sent as a sequence of bits over the shared communication medium. The bits were organized into packets, each containing the address of the destination. Packets also included a checksum used to detect errors in transmission and the protocol supported the ability to request the retransmission of corrupted packets.

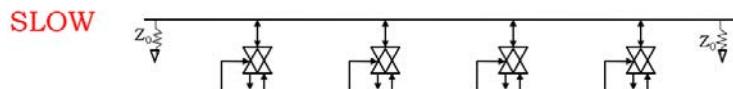
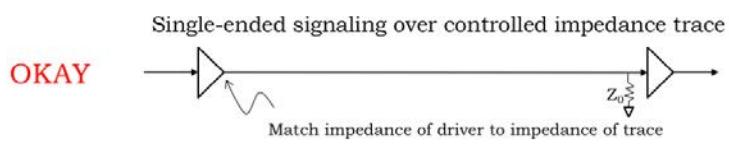
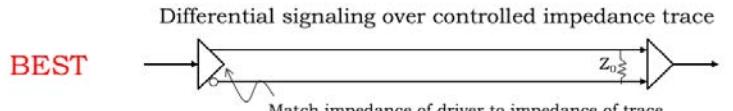
The software controlling the network is divided into a "stack" of modules, each implementing a different communication abstraction. The lowest-level physical layer is responsible for transmitting and receiving an individual packet of bits. Bit errors are detected and corrected, and packets with uncorrectable errors are discarded. There are different physical-layer modules available for the different types of physical networks.

The network layer deals with the addressing and routing of packets. Clever routing algorithms find the shortest communication path through the multi-hop network and deal with momentary or long-term outages on particular network links.

The transport layer is responsible for providing the reliable communication of a stream of data, dealing with the issues of discarded or out-of-order packets. In an effort to optimize network usage and limit packet losses due to network congestion, the transport layer deals with flow control, *i.e.*, the rate at which packets are sent.

A key idea in the networking community is the notion of building a reliable communication channel on top of a "best efforts" packet network. Higher layers of the protocol are designed so that it's possible to recover from errors in the lower layers. This has proven much more cost-effective and robust than trying to achieve 100% reliability at each layer.

Lessons learned: single driver, point-to-point



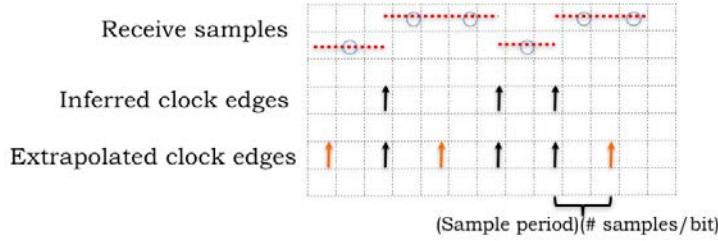
Issues:

- Impedance troubles when driving in middle
- Turn-around time when sharing a wire (wired-or glitch)

As we saw in the previous section, there are a lot of electrical issues when trying to communicate over a shared wire with multiple drivers and receivers. Slowing down the rate of communication helps to solve the problems, but “slow” isn’t in the cards for today’s high-performance systems.

Experience in the network world has shown that the fastest and least problematic communication channels have a single driver communicating with a single receiver, what’s called a point-to-point link. Using differential signaling is particularly robust. With differential signaling, the receiver measures the voltage difference across the two signaling wires. Electrical effects that might induce voltage noise on one signaling wire will affect the other in equal measure, so the voltage difference will be largely unaffected by most noise. Almost all high-performance communication links use differential signaling.

Lessons learned: clock recovery



- Receiver can infer presence of clock edge every time there's a transition in the received samples.
- Using sample period, extrapolate remaining edges
 - Now know first and last sample for each bit
 - Choose "middle" sample to determine message bit
- Can't go too long without a clock edge → 8b10b encoding

If we're sending digital data, does that mean we also have to send a separate clock signal so the receiver knows when to sample the signal to determine the next bit?

With some cleverness, it turns out that we can recover the timing information from the received signal assuming we know the nominal clock period at the transmitter. If the transmitter changes the bit it's sending at the rising edge of the transmitter's clock, then the receiver can use the transitions in the received waveform to infer the timing for some of the clock edges.

Then the receiver can use its knowledge of the transmitter's nominal clock period to infer the location of the remaining clock edges. It does this by using a phase-locked loop to generate a local facsimile of the transmitter's clock, using any received transitions to correct the phase and period of the local clock. The transmitter adds a training sequence of bits at the front of packet to ensure that the receiver's phased-lock loop is properly synchronized before the packet data itself is transmitted. A special unique bit sequence is used to separate the training signal from the packet data so the receiver can tell exactly where the packet data starts even if it missed a few training bits while the clocks were being properly synchronized.

Once the receiver knows the timing of the clock edges, it can then sample the incoming waveform towards the end of each clock period to determine the transmitted bit.

To keep the local clock in sync with the transmitter's clock, the incoming waveform needs to have reasonably frequent transitions. But if the transmitter is sending say, all zeroes, how can we guarantee frequent-enough clock edges?

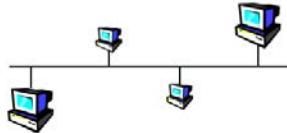
The trick, invented by IBM, is for the transmitter to take the stream of message bits and re-encode them into a bit stream that is guaranteed to have transitions no matter what the message bits are. The most commonly used encoding is 8b10b, where 8 message bits are encoded into 10 transmitted bits, where the encoding guarantees a transition at least every 6 bit times. Of course, the receiver has to reverse the 8b10b encoding to recover the actual message bits. Pretty neat!

The benefit of this trick is that we truly only need to send a single stream of bits. The receiver will be able to recover both the timing information and the data without also needing to transmit a separate clock signal.

Serial, Point-to-Point Communications

ETHERNET: Broadcast technology

- Sharing (contention) issues
- Multiple-drop-point issues...
- bit-serial (single wire)
- "Packets" for multi-bit data



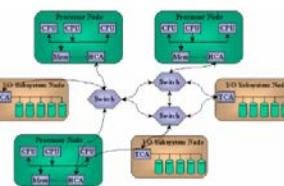
Evolution: Point-to-point

- 10BaseT, separate R & T wires
- Each link connects only 2 hosts, one sends, the other receives
- Network riddled with switches, routers



Serial point-to-point bus replacements

- Multi Gbit/sec serial links!
- PCIe, Infiniband, SATA, USB, ...
- Packets, headers
- Switches, routing
- Trend: localized, superfast, serial networks!



Using these lessons, networks have evolved from using shared communication channels to using point-to-point links. Today local-area networks use 10, 100, or 1000 BaseT wiring which includes separate differential pairs for sending and receiving, *i.e.*, each sending or receiving channel is unidirectional with a single driver and single receiver. The network uses separate switches and routers to receive packets from a sender and then forward the packets over a point-to-point link to the next switch, and so on, across multiple point-to-point links until the packet arrives at its destination.

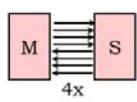
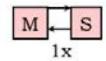
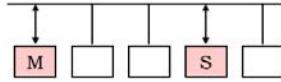
System-level connections have evolved to use the same communication strategy: point-to-point links with switches for routing packets to their intended destination. Note that communication along each link is independent, so a network with many links can actually support a lot of communication bandwidth. With a small amount of packet buffering in the switches to deal with momentary contention for a particular link, this is a very effective strategy for moving massive amounts of information from one component to the next.

In the next section, we'll look at some of the more interesting details.

Improving on the bus: *lessons learned from the network world*

Bus issues:

- shared medium → arbitrate between requesters
- clock skew → parallel bit lines, variable timings
- multiple masters → turnaround time
- impedance discontinuities, stubs → reflections



REPLACEMENT: fast unidirectional **serial point-to-point link**

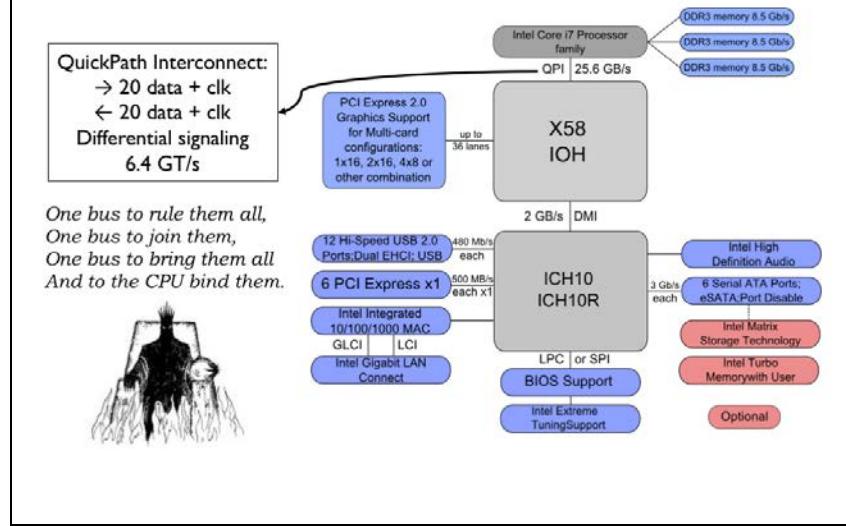
- one transmitter, one receiver → no arbitration, no turnaround
- serial packets replace parallel wire bundles
- clock recovered from data bits → no skew problems
- unidirectional, point-to-point → good signal quality
- need more throughput? → use multiple serial links in parallel...
- need many-to-many communication? → switches (like Ethernet)
- complex interface → Moore's law to the rescue!

Serial point-to-point links are the modern replacement for the parallel communications bus with all its electrical and timing issues. Each link is unidirectional and has only a single driver and the receiver recovers the clock signal from the data stream, so there are no complications from sharing the channel, clock skew, and electrical problems. The very controlled electrical environment enables very high signaling rates, well up into the gigahertz range using today's technologies.

If more throughput is needed, you can use multiple serial links in parallel. Extra logic is needed to reassemble the original data from multiple packets sent in parallel over multiple links, but the cost of the required logic gates is very modest in current technologies.

Note that the expansion strategy of modern systems still uses the notion of an add-in card that plugs into the motherboard. But instead of connecting to a parallel bus, the add-in card connects to one or more point-to-point communication links.

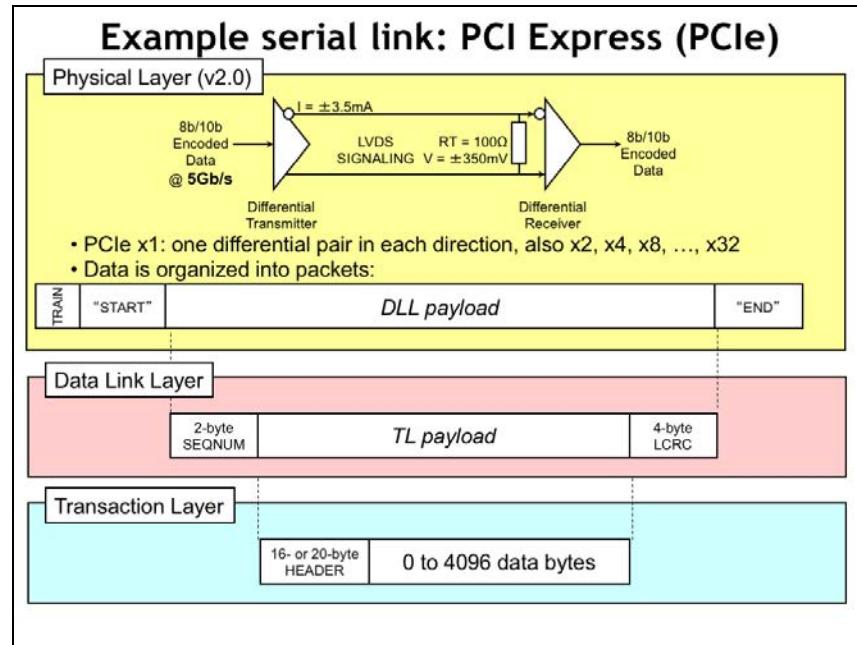
Communications in Today's Computers



Here's the system-level communications diagram for a recent system based on an Intel Core i7 CPU chip. The CPU is connected directly to the memories for the highest-possible memory bandwidth, but talks to all the other components over the QuickPath Interconnect (QPI), which has 20 differential signaling paths in each direction. QPI supports up to 6.4 billion 20-bit transfers in each direction every second.

All the other communication channels (USB, PCIe, networks, Serial ATA, Audio, etc.) are also serial links, providing various communication bandwidths depending on the application.

Reading about the QPI channel used by the CPU reminded me a lot of the one ring that could be used to control all of Middle Earth in the Tolkien trilogy *Lord of the Rings*. Why mess around with a lot of specialized communication channels when you have a single solution that's powerful enough to solve all your communication needs?



PCI Express (PCIe) is often used as the communication link between components on the system motherboard. A single PCIe version 2 “lane” transmits data at 5 Gb/sec using low-voltage differential signal (LVDS) over wires designed to have a 100-Ohm characteristic impedance.

The PCIe lane is under the control of the same sort of network stack as described earlier. The physical layer transmits packetized data through the lane. Each packet starts with a training sequence to synchronize the receiver’s clock-recovery circuitry, followed by a unique start sequence, then the packet’s data payload, and ends with a unique end sequence.

The physical layer payload is organized as a sequence number, a transaction layer payload and a cyclical redundancy check sequence that’s used to validate the data. Using the sequence number, the data link layer can tell when a packet has been dropped and request the transmitter restart transmission at the missing packet. It also deals with flow control issues.

Finally, the transaction layer reassembles the message from the transaction layer payloads from all the lanes and uses the header to identify the intended recipient at the receive end.

Altogether, a significant amount of logic is needed to send and receive messages on multiple PCIe lanes, but the cost is quite acceptable when using today’s integrated circuit technologies. Using 8 lanes, the maximum transfer rate is 4 GB/sec, capable of satisfying the needs of high-performance peripherals such as graphics cards.

So knowledge from the networking world has reshaped how components communicate on the motherboard, driving the transition from parallel buses to a handful of serial point-to-point links. As a result today’s systems are faster, more reliable, more energy-efficient and smaller than ever before.

Communication Topologies

asymptotic cost/performance tradeoffs

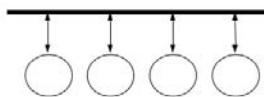
Goal: enable communications between n components

- Each point-to-point link requires one hardware unit.
- Each point-to-point communication requires one time unit.
- Each link operates independently

1-dimensional approaches:

BUS

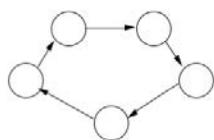
Shared communication channel allows only one message at a time



Throughput	$O(1)$
Latency	$O(1)$
Cost	$O(n)$

RING

Each component has link to next component on ring



Throughput	$O(n)$
Latency	$O(n)$
Cost	$O(n)$

Let's wrap up our discussion of system-level interconnect by considering how best to connect N components that need to send messages to one another, *e.g.*, CPUs on a multicore chip. Today such chips have a handful of cores, but soon they may have 100s or 1000s of cores.

We'll build our communications network using point-to-point links. In our analysis, each point-to-point link is counted at a cost of 1 hardware unit. Sending a message across a link requires one time unit. And we'll assume that different links can operate in parallel, so more links will mean more message traffic.

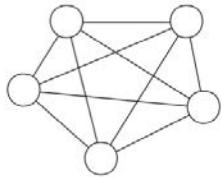
We'll do an asymptotic analysis of the throughput (total messages per unit time), latency (worst-case time to deliver a single message), and hardware cost. In other words, we'll make a rough estimate how these quantities change as N grows.

Note that in general the throughput and hardware cost are proportional to the number of point-to-point links.

Our baseline is the backplane bus discussed earlier, where all the components share a single communication channel. With only a single channel, bus throughput is 1 message per unit time and a message can travel between any two components in one time unit. Since each component has to have an interface to the shared channel, the total hardware cost is $O(n)$.

In a ring network each component sends its messages to a single neighbor and the links are arranged so that it's possible to reach all components. There are N links in total, so the throughput and cost are both $O(n)$. The worst case latency is also $O(n)$ since a message might have to travel across $N-1$ links to reach the neighbor that's immediately upstream. Ring topologies are useful when message latency isn't important or when most messages are to the component that's immediately downstream, *i.e.*, the components form a processing pipeline.

Quadratic-cost Topologies



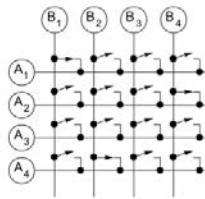
COMPLETE GRAPH

Dedicated lines connecting each pair of communicating nodes. There are $\sum_{i=1}^N (N - i) = O(n^2)$ links.

Throughput	$O(n^2)$
Latency	$O(1)$
Cost	$O(n^2)$

CROSSBAR SWITCH

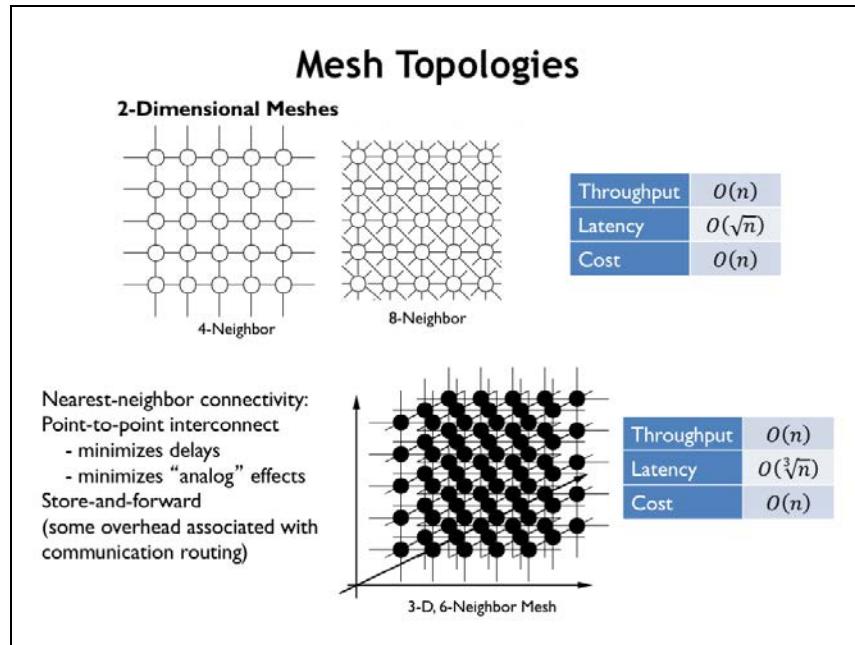
- Switch dedicated between each pair of nodes
- Each A_i can be connected to one B_j at any time
- Special cases:
 - A = processors, B = memories
 - A, B same type of node
 - A, B same nodes (complete graph)



Throughput	$O(n)$
Latency	$O(1)$
Cost	$O(n^2)$

The most general network topology is when every component has a direct link to every other component. There are $O(N^{**2})$ links so the throughput and cost are both $O(N^{**2})$. And the latency is 1 time unit since each destination is directly accessible. Although expensive, complete graphs offer very high throughput with very low latencies.

A variant of the complete graph is the crossbar switch where a particular row and column can be connected to form a link between particular A and B components with the restriction that each row and each column can only carry 1 message during each time unit. Assume that the first row and first column connect to the same component, and so on, *i.e.*, that the example crossbar switch is being used to connect 4 components. Then there are $O(n)$ messages delivered each time unit, with a latency of 1. There are N^{**2} switches in the crossbar, so the cost is $O(N^{**2})$ even though there are only $O(n)$ links.



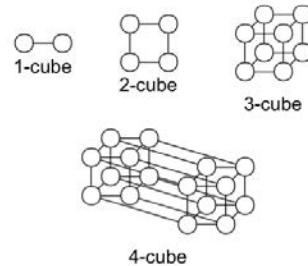
In mesh networks, components are connected to some fixed number of neighboring components, in either 2- or 3-dimensions. Hence the total number of links is proportional to the number of components, so both throughput and cost are $O(n)$. The worst-case latencies for mesh networks are proportional to length of the sides, so the latency is $O(\sqrt{n})$ for 2D meshes and $O(\sqrt[3]{n})$ for 3D meshes. The orderly layout, constant per-node hardware costs, and modest worst-case latency make 2D 4-neighbor meshes a popular choice for the current generation of experimental multi-core processors.

Logarithmic-latency Networks

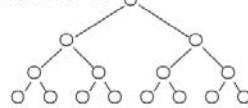
HYPERCUBE

D dimensions $\rightarrow 2^D$ nodes
Each node has D links

Throughput	$O(n \log_D n)$
Latency	$O(\log_D n)$
Cost	$O(n \log_D n)$



BINARY TREE



Throughput	$O(n)$
Latency	$O(\log_2 n)$
Cost	$O(n)$

Hypercube and tree networks offer logarithmic latencies, which for large N may be faster than mesh networks. The original CM-1 Connection Machine designed in the 80's used a hypercube network to connect up to 65,536 very simple processors, each connected to 16 neighbors. Later generations incorporated smaller numbers of more sophisticated processors, still connected by a hypercube network. In the early 90's the last generation of Connection Machines used a tree network, with the clever innovation that the links towards the root of the tree had a higher message capacity.

Communication Technologies: Latency

- Theorist's view:
 - Each point-to-point link requires one hardware unit.
 - Each point-to-point communication requires one time unit.
- | Topology | \$ | Theoretical Latency | Actual Latency |
|----------------|-----------------|---------------------|------------------|
| Complete graph | $O(n^2)$ | $O(1)$ | $O(\sqrt[3]{n})$ |
| Crossbar | $O(n^2)$ | $O(1)$ | $O(n)$ |
| 1D Bus | $O(n)$ | $O(1)$ | $O(n)$ |
| 2D Mesh | $O(n)$ | $O(\sqrt{n})$ | $O(\sqrt{n})$ |
| 3D Mesh | $O(n)$ | $O(\sqrt[3]{n})$ | $O(\sqrt[3]{n})$ |
| Tree | $O(n)$ | $O(\log_2 n)$ | $O(\sqrt[3]{n})$ |
| N-cube | $O(n \log_D n)$ | $O(\log_D n)$ | $O(\sqrt[3]{n})$ |
- Engineer's view:
 - Loading increases with number of connections (bus, crossbar)
 - Nodes have size: limits possible 2D, 3D density (other topologies)

Here's a summary of the theoretical latencies we calculated for the various topologies.

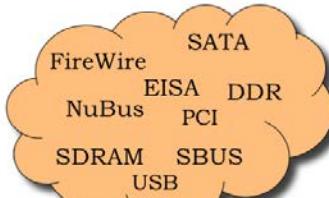
As a reality check, it's important to realize that the lower bound on the worst-case distance between components in our 3-dimensional world is $O(\text{cube root of } N)$. In the case of a 2D layout, the worst-case distance is $O(\sqrt{N})$. Since we know that the time to transmit a message is proportional to the distance traveled, we should modify our latency calculations to reflect this physical constraint.

Note that the bus and crossbar involve N connections to a single link, so here the lower-bound on the latency needs to reflect the capacitive load added by each connection.

The winner? Mesh networks avoid the need for longer wires as the number of connected components grows and appear to be an attractive alternative for high-capacity communication networks connecting 1000's of processors.

Communications Futures

- Backplane buses have evolved into point-to-point links
- + links operate independently
 - + links can be managed in groups
 - + packetized data deals with errors

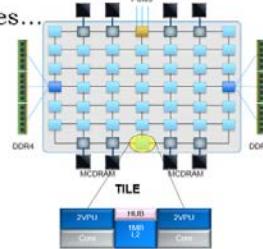


Specialized buses for memory

Networked “peripherals” for mobile devices...

New-generation communications...

- how should 100 (1000?) cores communicate?



Summarizing our discussion:

- Point-to-point links are in common use today for system-level interconnect, and as a result our systems are faster, more reliable, more energy-efficient and smaller than ever before.
- Multi-signal parallel buses are still used for very-high-bandwidth connections to memories, with a lot of very careful engineering to avoid the electrical problems observed in earlier bus implementations.
- Wireless connections are in common use to connect mobile devices to nearby components and there has been interesting work on how to allow mobile devices to discover what peripherals are nearby and enable them to connect automatically.
- The upcoming generation of multi-core chips will have 10's to 100's of processing cores. There is a lot ongoing research to determine which communication topology would offer the best combination of high communication bandwidth and low latency. The next ten years will be an interesting time for on-chip network engineers!

L21: Parallel Processing

1. Processor Performance
2. 5-Stage Pipelined Processors
3. Improving 5-Stage Pipeline Performance
4. Limits to Pipeline Depth
5. Improving 5-Stage Pipeline Performance
6. Instruction-level Parallelism (ILP)
7. Wider or Superscalar Pipelines
8. A Modern Out-of-Order Superscalar Processor
9. Limits to Single-Processor Performance
10. Data-Level Parallelism
11. Vector Code Example
12. Data-Dependent Vector Operations
13. Vector Processing Implementations
14. Multicore Processors
15. Amdahl's Law
16. Amdahl's Law and Parallelism
17. Thread-Level Parallelism
18. Multicore Caches
19. What Are the Possible Outcomes
20. Uniprocessor Outcome
21. Sequential Consistency
22. Alternatives to Sequential Consistency?
23. Fix: "Snoopy" Cache Coherence Protocol
24. Example: MESI Cache Coherence Protocol
25. The Cache Has Two Customers!
26. MESI Activity Diagram
27. Cache Coherence in Action
28. Parallel Processing Summary

Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction Cycle}} \cdot \frac{\text{Time}}{\text{CPI} \cdot t_{\text{CLK}}}$$

- Pipelining lowers t_{CLK} . What about CPI?
- $\text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{stall}}$
 - $\text{CPI}_{\text{ideal}}$: cycles per instruction if no stall
- $\text{CPI}_{\text{stall}}$ contributors
 - Data hazards
 - Control hazards: branches, exceptions
 - Memory latency: cache misses

The modern world has an insatiable appetite for computation, so system architects are always thinking about ways to make programs run faster. The running time of a program is the product of three terms:

The number of instructions in the program, multiplied by the average number of processor cycles required to execute each instruction (CPI), multiplied by the time required for each processor cycle (t_{CLK}).

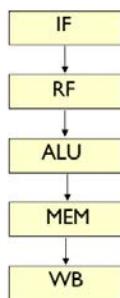
To decrease the running time we need to decrease one or more of these terms. The number of instructions per program is determined by the ISA and by the compiler that produced the sequence of assembly language instructions to be executed. Both are fair game, but for this discussion, let's work on reducing the other two terms.

As we've seen, pipelining reduces t_{CLK} by dividing instruction execution into a sequence of steps, each of which can complete its task in a shorter t_{CLK} . What about reducing CPI?

In our 5-stage pipelined implementation of the Beta, we designed the hardware to complete the execution of one instruction every clock cycle, so $\text{CPI}_{\text{ideal}}$ is 1. But sometimes the hardware has to introduce "NOP bubbles" into the pipeline to delay execution of a pipeline stage if the required operation couldn't (yet) be completed. This happens on taken branch instructions, when attempting to immediately use a value loaded from memory by the LD instruction, and when waiting for a cache miss to be satisfied from main memory. $\text{CPI}_{\text{stall}}$ accounts for the cycles lost to the NOPs introduced into the pipeline. Its value depends on the frequency of taken branches and immediate use of LD results. Typically it's some fraction of a cycle. For example, if a 6-instruction loop with a LD takes 8 cycles to complete, $\text{CPI}_{\text{stall}}$ for the loop would be 2/6, *i.e.*, 2 extra cycles for every 6 instructions.

5-Stage Pipelined Processors

- Advantages
 - CPI_{ideal} is 1 (pipelining)
 - Simple, elegant
 - Still used in ARM & MIPS processors
- Room for improvement
 - Upper performance bound is CPI=1
 - High-latency instructions not handled well
 - 1 stage for accesses to large caches or multiplier
 - Long clock cycle time
 - Unnecessary stalls due to rigid pipeline
 - If one instruction stalls, anything behind it stalls



Our classic 5-stage pipeline is an effective compromise that allows for a substantial reduction of t_{CLK} while keeping CPI_{stall} to a reasonably modest value.

There is room for improvement. Since each stage is working on one instruction at a time, CPI_{ideal} is 1.

Slow operations — *e.g.*, completing a multiply in the ALU stage, or accessing a large cache in the IF or MEM stages — force t_{CLK} to be large to accommodate all the work that has to be done in one cycle.

The order of the instructions in the pipeline is fixed. If, say, a LD instruction is delayed in the MEM stage because of a cache miss, all the instructions in earlier stages are also delayed even though their execution may not depend on the value produced by the LD. The order of instructions in the pipeline always reflects the order in which they were fetched by the IF stage.

Let's look into what it would take to relax these constraints and hopefully improve program runtimes.

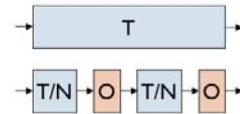
Improving 5-stage Pipeline Performance

- Lower t_{CLK} : deeper pipelines
 - Overlap more instructions

Increasing the number of pipeline stages should allow us to decrease the clock cycle time. We'd add stages to break up performance bottlenecks, *e.g.*, adding additional pipeline stages (MEM1 and MEM2) to allow a longer time for memory operations to complete. This comes at cost to CPI_stall since each additional MEM stage means that more NOP bubbles have to be introduced when there's a LD data hazard. Deeper pipelines mean that the processor will be executing more instructions in parallel.

Limits to Pipeline Depth

- Each pipeline stage introduces some overhead (O)
 - Propagation delay of pipeline registers
 - Setup and hold times
 - Clock skew
 - Inequalities in work per stage
 - Cannot break up work into stages at arbitrary points
- If original t_{CLK} was T , with N stages t_{CLK} is $T/N+O$
 - If $N \rightarrow \infty$, speedup = $T / (T/N+O) \rightarrow T/O$
 - Assuming that CPI stays constant
 - Eventually overhead dominates and deeper pipelines have diminishing returns



Let's interrupt enumerating our performance shopping list to think about limits to pipeline depth.

Each additional pipeline stage includes some additional overhead costs to the time budget. We have to account for the propagation, setup, and hold times for the pipeline registers. And we usually have to allow a bit of extra time to account for clock skew, *i.e.*, the difference in arrival time of the clock edge at each register. And, finally, since we can't always divide the work exactly evenly between the pipeline stages, there will be some wasted time in the stages that have less work. We'll capture all of these effects as an additional per-stage time overhead of O .

If the original clock period was T , then with N pipeline stages, the clock period will be $T/N + O$.

At the limit, as N becomes large, the speedup approaches T/O . In other words, the overhead starts to dominate as the time spent on work in each stage becomes smaller and smaller. At some point adding additional pipeline stages has almost no impact on the clock period.

As a data point, the Intel Core-2 x86 chips (nicknamed "Nehalem") have a 14-stage execution pipeline.

Improving 5-stage Pipeline Performance

- Lower t_{CLK} : **deeper pipelines**
 - Overlap more instructions
- Higher CPI_{ideal} : **wider pipelines**
 - Each pipeline stage processes multiple instructions
- Lower CPI_{stall} : **out-of-order execution**
 - Execute each instruction as soon as its source operands are available
- Balance conflicting goals
 - Deeper & wider pipelines \Rightarrow more control hazards
 - **Branch prediction**
- It all works because of **instruction-level parallelism (ILP)**

Okay, back to our performance shopping list...

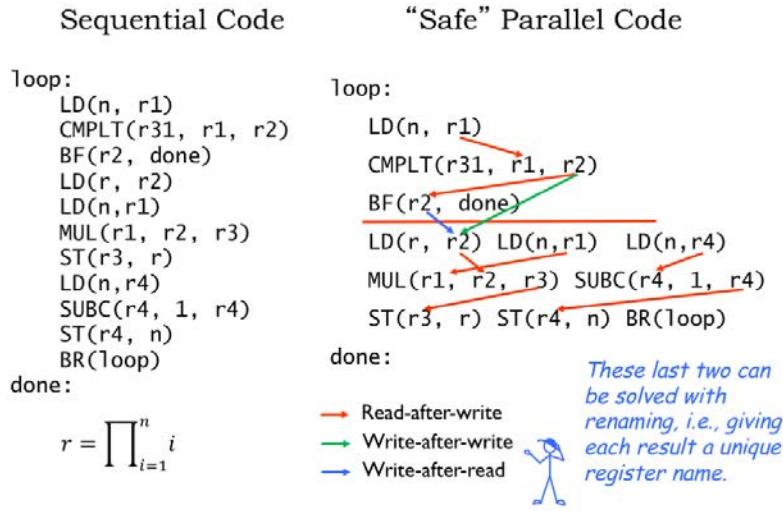
There may be times we can arrange to execute two or more instructions in parallel, assuming that their executions are independent from each other. This would increase CPI_{ideal} at the cost of increasing the complexity of each pipeline stage to deal with concurrent execution of multiple instructions.

If there's an instruction stalled in the pipeline by a data hazard, there may be following instructions whose execution could still proceed. Allowing instructions to pass each other in the pipeline is called out-of-order execution. We'd have to be careful to ensure that changing the execution order didn't affect the values produced by the program.

More pipeline stages and wider pipeline stages increase the amount of work that has to be discarded on control hazards, potentially increasing CPI_{stall} . So it's important to minimize the number of control hazards by predicting the results of a branch (*i.e.*, taken or not taken) so that we increase the chances that the instructions in the pipeline are the ones we'll want to execute.

Our ability to exploit wider pipelines and out-of-order execution depends on finding instructions that can be executed in parallel or in different orders. Collectively these properties are called "instruction-level parallelism" (ILP).

Instruction Level Parallelism (ILP)



Here's an example that will let us explore the amount of ILP that might be available. On the left is an unoptimized loop that computes the product of the first N integers. On the right, we've rewritten the code, placing instructions that could be executed concurrently on the same line.

First notice the red line following the BF instruction. Instructions below the line should only be executed if the BF is *not* taken. That doesn't mean we couldn't start executing them before the results of the branch are known, but if we executed them before the branch, we would have to be prepared to throw away their results if the branch was taken.

The possible execution order is constrained by the read-after-write (RAW) dependencies shown by the red arrows. We recognize these as the potential data hazards that occur when an operand value for one instruction depends on the result of an earlier instruction. In our 5-stage pipeline, we were able to resolve many of these hazards by bypassing values from the ALU, MEM, and WB stages back to the RF stage where operand values are determined.

Of course, bypassing will only work when the instruction has been executed so its result is available for bypassing! So in this case, the arrows are showing us the constraints on execution order that guarantee bypassing will be possible.

There are other constraints on execution order. The green arrow identifies a write-after-write (WAW) constraint between two instructions with the same destination register. In order to ensure the correct value is in R2 at the end of the loop, the LD(r,R2) instruction has to store its result into the register file after the result of the CMPLT instruction is stored into the register file.

Similarly, the blue arrow shows a write-after-read (WAR) constraint that ensures that the correct values are used when accessing a register. In this case, LD(r,R2) must store into R2 after the Ra operand for the BF has been read from R2.

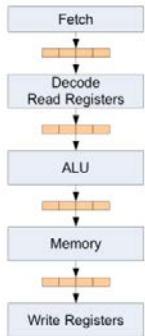
As it turns out, WAW and WAR constraints can be eliminated if we give each instruction result a unique register name. This can actually be done relatively easily by the hardware by using a generous supply of temporary registers, but we won't go into the details of renaming here. The use of temporary

registers also makes it easy to discard results of instructions executed before we know the outcomes of branches.

In this example, we discovered that the potential concurrency was actually pretty good for the instructions following the BF.

Wider or Superscalar Pipelines

- Each stage operates on up to N instructions each clock cycle
 - Known as wide or superscalar pipelines
 - $CPI_{ideal} = 1/N$
- Options (from simpler to harder)
 - One integer and one floating-point instruction
 - Any N=2 instructions
 - Any N=4 instructions
 - Any N=? Instructions
 - What are the limits?



See <http://people.ee.duke.edu/~sorin/ece252/lectures/3-superscalar.pdf>

To take advantage of this potential concurrency, we'll need to modify the pipeline to execute some number N of instructions in parallel. If we can sustain that rate of execution, CPI_ideal would then be 1/N since we'd complete the execution of N instructions in each clock cycle as they exited the final pipeline stage.

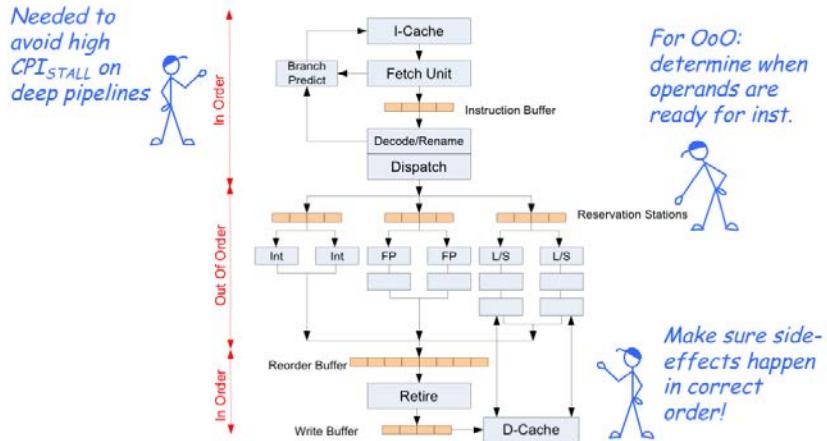
So what value should we choose for N? Instructions that are executed by different ALU hardware are easy to execute in parallel, *e.g.*, ADDs and SHIFTS, or integer and floating-point operations. Of course, if we provided multiple adders, we could execute multiple integer arithmetic instructions concurrently. Having separate hardware for address arithmetic (called LD/ST units) would support concurrent execution of LD/ST instructions and integer arithmetic instructions.

This set of lecture slides from Duke gives a nice overview of techniques used in each pipeline stage to support concurrent execution.

Basically by increasing the number of functional units in the ALU and the number of memory ports on the register file and main memory, we would have what it takes to support concurrent execution of multiple instructions. So, what's the right tradeoff between increased circuit costs and increased concurrency?

As a data point, the Intel Nehalem core can complete up to 4 micro-operations per cycle, where each micro-operation corresponds to one of our simple RISC instructions.

A Modern Out-of-Order Superscalar Processor



Here's a simplified diagram of a modern out-of-order superscalar processor.

Instruction fetch and decode handles, say, 4 instructions at a time. The ability to sustain this execution rate depends heavily on the ability to predict the outcome of branch instructions, ensuring that the wide pipeline will be mostly filled with instructions we actually want to execute. Good branch prediction requires the use of the history from previous branches and there's been a lot of cleverness devoted to getting good predictions from the least amount of hardware! If you're interested in the details, search for "branch predictor" on Wikipedia.

The register renaming happens during instruction decode, after which the instructions are ready to be dispatched to the functional units.

If an instruction needs the result of an earlier instruction as an operand, the dispatcher has identified which functional unit will be producing the result. The instruction waits in a queue until the indicated functional unit produces the result and when all the operand values are known, the instruction is finally taken from the queue and executed. Since the instructions are executed by different functional units as soon as their operands are available, the order of execution may not be the same as in the original program.

After execution, the functional units broadcast their results so that waiting instructions know when to proceed. The results are also collected in a large reorder buffer so that they can be retired (*i.e.*, write their results in the register file) in the correct order.

Whew! There's a lot of circuitry involved in keeping the functional units fed with instructions, knowing when instructions have all their operands, and organizing the execution results into the correct order. So how much speed up should we expect from all this machinery? The effective CPI is very program-specific, depending as it does on cache hit rates, successful branch prediction, available ILP, and so on. Given the architecture described here the best speed up we could hope for is a factor of 4. Googling around, it seems that the reality is an average speed-up of 2, maybe slightly less, over what would be achieved by an in-order, single-issue processor.

Limits To Single-Processor Performance

- Pipeline depth: getting close to pipelining limits
 - Clocking overheads, CPI degradation
- Branch prediction & memory latency limit the practical benefits of out-of-order execution
- Power grows superlinearly with higher frequency & more OoO logic
- Extreme design complexity
- Limited ILP → Must exploit DLP and TLP
 - Data-Level Parallelism: Vector extensions, GPUs
 - Thread-Level Parallelism: Multiple threads and cores

What can we expect for future performance improvements in out-of-order, superscalar pipelines?

Increases in pipeline depth can cause CPI_stall and timing overheads to rise. At the current pipeline depths the increase in CPI_stall is larger than the gains from decreased t_{CLK} and so further increases in depth are unlikely.

A similar tradeoff exists between using more out-of-order execution to increase ILP and the increase in CPI_stall caused by the impact of mis-predicted branches and the inability to run main memories any faster.

Power consumption increases more quickly than the performance gains from lower t_{CLK} and additional out-of-order execution logic.

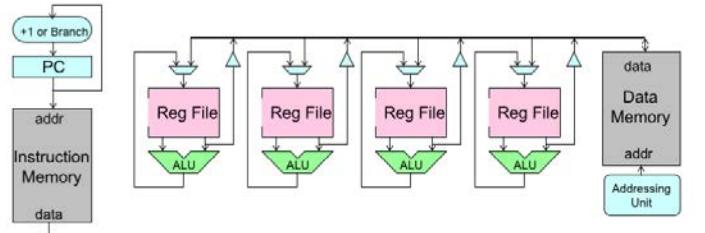
The additional complexity required to enable further improvements in branch prediction and concurrent execution seems very daunting.

All of these factors suggest that it is unlikely that we'll see substantial future improvements in the performance of out-of-order superscalar pipelined processors.

So system architects have turned their attention to exploiting data-level parallelism (DLP) and thread-level parallelism (TLP). These are our next two topics.

Data-Level Parallelism

- Same operation applied to multiple data elements
`for (int i = 0; i < 16; i++) x[i] = a[i] + b[i];`
- Exploit with **vector processors** or vector ISA extensions



- » Each datapath has its own local storage (register file)
- » All datapaths execute the same instruction
- » Memory access with vector loads and stores + wide memory port

For some applications, data naturally comes in vector or matrix form. For example, a vector of digitized samples representing an audio waveform over time, or a matrix of pixel colors in a 2D image from a camera. When processing that data, it's common to perform the same sequence of operations on each data element. The example code shown here is computing a vector sum, where each component of one vector is added to the corresponding component of another vector.

By replicating the datapath portion of our CPU, we can design special-purpose vector processors capable of performing the same operation on many data elements in parallel. Here we see that the register file and ALU have been replicated and the control signals from decoding the current instruction are shared by all the datapaths. Data is fetched from memory in big blocks (very much like fetching a cache line) and the specified register in each datapath is loaded with one of the words from the block. Similarly each datapath can contribute a word to be stored as a contiguous block in main memory. In such machines, the width of the data buses to and from main memory is many words wide, so a single memory access provides data for all the datapaths in parallel.

Executing a single instruction on a machine with N datapaths is equivalent to executing N instructions on a conventional machine with a single datapath. The result achieves a lot of parallelism without the complexities of out-of-order superscalar execution.

Vector Code Example

```
for (i = 0; i < 16; i++) x[i] = a[i] + b[i];
```

Beta assembly	Equivalent vector assembly
CMOVE(16, R0)	LD.V(R1, 0, V1)
loop: LD(R1, 0, R4)	LD.V(R2, 0, V2)
LD(R2, 0, R5)	ADD.V(V1, V2, V3)
ADDC(R1, 4, R1)	ST.V(V3, 0, R3)
ADDC(R2, 4, R2)	
ADD(R4, R5, R6)	
ST(R6, 0, R3)	
ADDC(R3, 4, R3)	
SUBC(R0, 1, R1)	
BNE(R0, loop)	
# of cycles = 1 + 10*15 + 9 = 160	# of cycles = 4

Suppose we had a vector processor with 16 datapaths. Let's compare its performance on a vector-sum operation to that of a conventional pipelined Beta processor.

Here's the Beta code, carefully organized to avoid any data hazards during execution. There are 9 instructions in the loop, taking 10 cycles to execute if we count the NOP introduced into the pipeline when the BNE at the end of the loop is taken. It takes 160 cycles to sum all 16 elements assuming no additional cycles are required due to cache misses.

And here's the corresponding code for a vector processor where we've assumed constant-sized 16-element vectors. Note that "V" registers refer to a particular location in the register file associated with each datapath, while the "R" registers are the conventional Beta registers used for address computations, etc. It would only take 4 cycles for the vector processor to complete the desired operations, a speed-up of 40.

This example shows the best-possible speed-up. The key to a good speed-up is our ability to "vectorize" the code and take advantage of all the datapaths operating in parallel. This isn't possible for every application, but for tasks like audio or video encoding and decoding, and all sorts of digital signal processing, vectorization is very doable. Memory operations enjoy a similar performance improvement since the access overhead is amortized over large blocks of data.

Data-dependent Vector Operations

```
for (i = 0; i < 16; i++)
    if (a[i] < b[i]) c[i] = c[i] + 3;
```

Equivalent vector assembly

```
LD.V(R1, 0, V1) // load a[i]
LD.V(R2, 0, V2) // load b[i]
LD.V(R3, 0, V3) // load c[i]
CMPLT.V(V1, V2) // set local predicate flags

// predicated instructions perform the
// indicated operation if the local predicate
// flag is true or is false.
ADDC.V.iftrue(V3, 3, V3)
```

You might wonder if it's possible to efficiently perform data-dependent operations on a vector processor. Data-dependent operations appear as conditional statements on conventional machines, where the body of the statement is executed if the condition is true. If testing and branching is under the control of the single instruction execution engine, how can we take advantage of the parallel datapaths?

The trick is provide each datapath with a local predicate flag. Use a vectorized compare instruction (CMPLT.V) to perform the $a[i] < b[i]$ comparisons in parallel and remember the result locally in each datapath's predicate flag. Then extend the vector ISA to include "predicated instructions" which check the local predicate to see if they should execute or do nothing. In this example, ADDC.V.iftrue only performs the ADDC on the local data if the local predicate flag is true.

Instruction predication is also used in many non-vector architectures to avoid the execution-time penalties associated with mis-predicted conditional branches. They are particularly useful for simple arithmetic and boolean operations (*i.e.*, very short instruction sequences) that should be executed only if a condition is met. The x86 ISA includes a conditional move instruction, and in the 32-bit ARM ISA almost all instructions can be conditionally executed.

Vector Processing Implementations

- Advantages of vector ISAs:
 - **Compact**: 1 instruction defines N operations
 - **Parallel**: N operations are (data) parallel and independent
 - **Expressive**: Memory operations describe regular patterns
- Modern CPUs: Vector extensions & wider registers
 - SSE: 128-bit operands (4x32-bit or 2x64-bit)
 - AVX (2011): 256-bit operands (8x32-bit or 4x64-bit)
 - AVX-512 (upcoming): 512-bit operands
 - Explicit parallelism, extracted at compile time (vectorization)
- GPUs: Designed for data parallelism from the ground up
 - 32 to 64 32-bit floating-point elements
 - Implicit parallelism, scalar binary with multiple instances executed in lockstep (and regrouped dynamically)

The power of vector processors comes from having 1 instruction initiate N parallel operations on N pairs of operands.

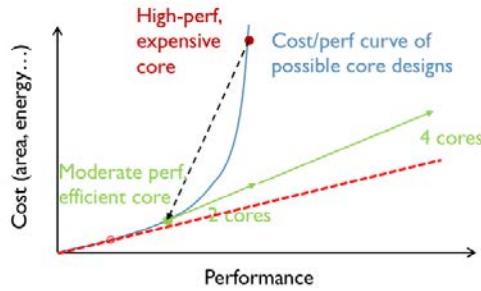
Most modern CPUs incorporate vector extensions that operate in parallel on 8-, 16-, 32- or 64-bit operands organized as blocks of 128-, 256-, or 512-bit data. Often all that's needed is some simple additional logic on an ALU designed to process full-width operands. The parallelism is baked into the vector program, not discovered on-the-fly by the instruction dispatch and execution machinery. Writing the specialized vector programs is a worthwhile investment for certain library functions which see a lot use in processing today's information streams with their heavy use of images, and A/V material.

Perhaps the best example of architectures with many datapaths operating in parallel are the graphics processing units (GPUs) found in almost all computer graphics systems. GPU datapaths are typically specialized for 32- and 64-bit floating point operations found in the algorithms needed to display in real-time a 3D scene represented as billions of triangular patches as a 2D image on the computer screen. Coordinate transformation, pixel shading and antialiasing, texture mapping, etc., are examples of "embarrassingly parallel" computations where the parallelism comes from having to perform the same computation independently on millions of different data objects. Similar problems can be found in the fields of bioinformatics, big data processing, neural net emulation used in deep machine learning, and so on. Increasingly, GPUs are used in many interesting scientific and engineering calculations and not just as graphics engines.

Data-level parallelism provides significant performance improvements in a variety of useful situations. So current and future ISAs will almost certainly include support for vector operations.

Multicore Processors

If applications have a lot of parallelism, using a larger number of simpler cores is more efficient!



What is the optimal tradeoff between core cost and number of cores?

In discussing out-of-order superscalar pipelined CPUs we commented that the costs grow very quickly relative to performance gains, leading to the cost-performance curve shown here. If we move down the curve, we can arrive at more efficient architectures that give, say, 1/2 the performance at a 1/4 of the cost.

When our applications involve independent computations that can be performed in parallel, it may be that we would be able to use two cores to provide the same performance as the original expensive core, but a fraction of the cost. If the available parallelism allows us to use additional cores, we'll see a linear relationship between increased performance vs. increased cost. The key, of course, is that desired computations can be divided into multiple tasks that can run independently, with little or no need for communication or coordination between the tasks.

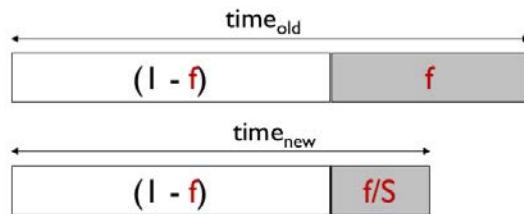
What is the optimal tradeoff between core cost and the number of cores? If our computation is arbitrarily divisible without incurring additional overhead, then we would continue to move down the curve until we found the cost-performance point that gave us the desired performance at the least cost. In reality, dividing the computation across many cores does involve some overhead, e.g., distributing the data and code, then collecting and aggregating the results, so the optimal tradeoff is harder to find. Still, the idea of using a larger number of smaller, more efficient cores seems attractive.

Amdahl's Law

- Speedup = time_{without enhancement} / time_{with enhancement}
- Suppose an enhancement speeds up a fraction f of a task by a factor of S

$$\text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot (1-f) + f/S$$

$$S_{\text{overall}} = \text{time}_{\text{old}} / \text{time}_{\text{new}} = 1 / (1-f) + f/S$$



Corollary: Make the common case fast

Many applications have some computations that can be performed in parallel, but also have computations that won't benefit from parallelism. To understand the speedup we might expect from exploiting parallelism, it's useful to perform the calculation proposed by computer scientist Gene Amdahl in 1967, now known as Amdahl's Law.

Suppose we're considering an enhancement that speeds up some fraction F of the task at hand by a factor of S . As shown in the figure, the gray portion of the task now takes F/S of the time that it used to require.

Some simple arithmetic lets us calculate the overall speedup we get from using the enhancement. One conclusion we can draw is that we'll benefit the most from enhancements that affect a large portion of the required computations, *i.e.*, we want to make F as large as possible.

Amdahl's Law and Parallelism

What is the maximum speedup you can get by running on a multicore machine?

$$S_{\text{overall}} = 1 / ((1-f) + f/S)$$

$$S_{\text{overall}} \xrightarrow[S \rightarrow \infty]{\lim} 1 / (1-f)$$

Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential

$$f = 0.9, S = \infty \rightarrow S_{\text{overall}} = 10$$

What f do you need to use a 1000-core machine well?

$$S_{\text{overall}} = 500 \rightarrow f = 0.998$$

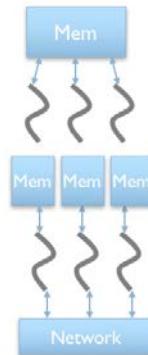
What's the best speedup we can hope for if we have many cores that can be used to speed up the parallel part of the task? Here's the speedup formula based on F and S, where in this case F is the parallel fraction of the task. If we assume that the parallel fraction of the task can be speed up arbitrarily by using more and more cores, we see that the best possible overall speed up is $1/(1-F)$.

For example, you write a program that can do 90% of its work in parallel, but the other 10% must be done sequentially. The best overall speedup that can be achieved is a factor of 10, no matter how many cores you have at your disposal.

Turning the question around, suppose you have a 1000-core machine which you hope to be able to use to achieve a speedup of 500 on your target application. You would need to be able parallelize 99.8% of the computation in order to reach your goal! Clearly multicore machines are most useful when the target task has lots of natural parallelism.

Thread-Level Parallelism

- Divide computation among multiple threads of execution
 - Each thread executes a different instruction stream
 - More flexible than vector processing, but more expensive
- Communication models:
 - Shared memory:
 - Single address space
 - Implicit communication by memory loads & stores
 - Message passing:
 - Separate address spaces
 - Explicit communication by sending and receiving messages



Using multiple independent cores to execute a parallel task is called thread-level parallelism (TLP), where each core executes a separate computation “thread”. The threads are independent programs, so the execution model is potentially more flexible than the lock-step execution provided by vector machines.

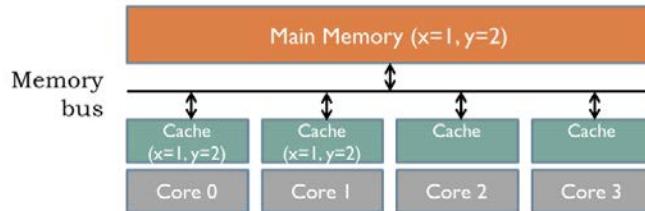
When there are a small number of threads, you often see the cores sharing a common main memory, allowing the threads to communicate and synchronize by sharing a common address space. We’ll discuss this further in the next section. This is the approach used in current multicore processors, which have between 2 and 12 cores.

Shared memory becomes a real bottleneck when there 10’s or 100’s of cores, since collectively they quickly overwhelm the available memory bandwidth. In these architectures, threads communicate using a communication network to pass messages back and forth. We discussed possible network topologies in an earlier lecture. A cost-effective on-chip approach is to use a nearest-neighbor mesh network, which supports many parallel point-to-point communications, while still allowing multi-hop communication between any two cores. Message passing is also used in computing clusters, where many ordinary CPUs collaborate on large tasks. There’s a standardized message passing interface (MPI) and specialized, very high throughput, low latency message-passing communication networks (*e.g.*, Infiniband) that make it easy to build high-performance computing clusters.

In the next couple of sections we’ll look more closely at some of the issues involved in building shared-memory multicore processors.

Multicore Caches

- Multicores have **multiple private caches** for performance
- We want the semantics of a single shared memory



Consider the following trivial threads running on C_0 and C_1 :

Thread A (C_0)

$x = 3;$
print(y);

Thread B (C_1)

$y = 4;$
print(x);

A conceptual schematic for a multicore processor is shown below. To reduce the average memory access time, each of the four cores has its own cache, which will satisfy most memory requests. If there's a cache miss, a request is sent to the shared main memory. With a modest number of cores and a good cache hit ratio, the number of memory requests that must access main memory during normal operation should be pretty small. To keep the number of memory accesses to a minimum, the caches implement a write-back strategy, where ST instructions update the cache, but main memory is only updated when a dirty cache line is replaced.

Our goal is that each core should share the contents of main memory, *i.e.*, changes made by one core should visible to all the other cores. In the example shown here, core 0 is running Thread A and core 1 is running Thread B. Both threads reference two shared memory locations holding the values for the variables X and Y.

The current values of X and Y are 1 and 2, respectively. Those values are held in main memory as well as being cached by each core.

What Are the Possible Outcomes?

Thread A

```
x = 3;  
print(y);
```

$\$_1: x = \textcolor{red}{\mathbf{X}} 3$
 $y = 2$

Thread B

```
y = 4;  
print(x);
```

$\$_2: x = 1$
 $y = \textcolor{red}{\mathbf{X}} 4$

Plausible execution sequences:

SEQUENCE	A prints	B prints	<i>Hey, we get the same answer every time... Let's go build it!</i>
x=3; print(y); y=4; print(x);	2	1	
x=3; y=4; print(y); print(x);	2	1	
x=3; y=4; print(x); print(y);	2	1	
y=4; x=3; print(x); print(y);	2	1	
y=4; x=3; print(y); print(x);	2	1	
y=4; print(x); x=3; print(y);	2	1	



What happens when the threads are executed? Each thread executes independently, updating its cache during stores to X and Y. For any possible execution order, either concurrent or sequential, the result is the same: Thread A prints “2”, Thread B prints “1”. Hardware engineers would point to the consistent outcomes and declare victory!

But closer examination of the final system state reveals some problems. After execution is complete, the two cores disagree on the values of X and Y. Threads running on core 0 will see X=3 and Y=2. Threads running on core 1 will see X=1 and Y=4. Because of the caches, the system isn’t behaving as if there’s a single shared memory. On the other hand, we can’t eliminate the caches since that would cause the average memory access time to skyrocket, ruining any hoped-for performance improvement from using multiple cores.

Uniprocessor Outcome

But, what are the possible outcomes if we ran Thread A and Thread B on a **single timed-shared processor**?

Thread A

x = 3;
print(y);

Thread B

y = 4;
print(x);

Plausible Uniprocessor execution sequences:

SEQUENCE	A prints	B prints
x=3; print(y); y=4; print(x);	2	3
x=3; y=4; print(y); print(x);	4	3
x=3; y=4; print(x); print(y);	4	3
y=4; x=3; print(x); print(y);	4	3
y=4; x=3; print(y); print(x);	4	3
y=4; print(x); x=3; print(y);	4	1

*Notice that
the outcome
2, 1 does not
appear in
this list!*



What outcome should we expect? One plausible standard of correctness is the outcome when the threads are run a single timeshared core. The argument would be that a multicore implementation should produce the same outcome but more quickly, with parallel execution replacing timesharing.

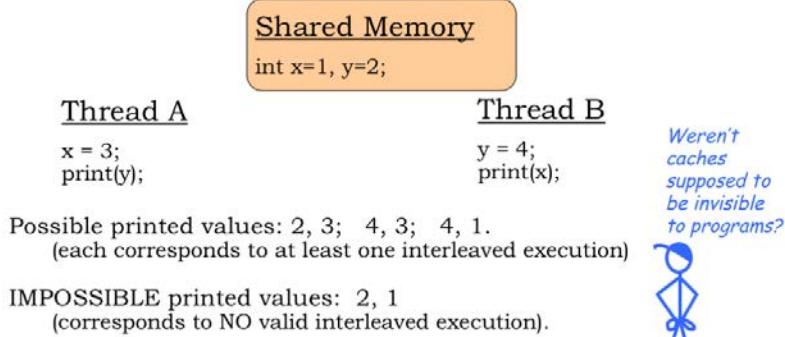
The table shows the possible results of the timesharing experiment, where the outcome depends on the order in which the statements are executed. Programmers will understand that there is more than one possible outcome and know that they would have to impose additional constraints on execution order, say, using semaphores, if they wanted a specific outcome.

Notice that the multicore outcome of 2,1 doesn't appear anywhere on the list of possible outcomes from sequential timeshared execution.

Sequential Consistency

Semantic constraint:

Result of executing N parallel threads should correspond to *some* interleaved execution on a single processor.



The notion that executing N threads in parallel should correspond to some interleaved execution of those threads on a single core is called “sequential consistency”. If multicore systems implement sequential consistency, then programmers can think of the systems as providing hardware-accelerated timesharing.

So, our simple multicore system fails on two accounts. First, it doesn’t correctly implement a shared memory since, as we’ve seen, it’s possible for the two cores to disagree about the current value of a shared variable. Second, as a consequence of the first problem, the system doesn’t implement sequential consistency.

Clearly, we’ll need to figure out a fix!

Alternatives to Sequential Consistency?

ALTERNATIVE MEMORY SEMANTICS:

"WEAK" consistency

EASIER GOAL: Memory operations from each thread appear to be performed in order issued by that thread ;

Memory operations from different threads may overlap in arbitrary ways (not necessarily consistent with any interleaving).

ALTERNATIVE APPROACH:

- Weak consistency, by default;
- MEMORY BARRIER instruction: stalls thread until all previous memory operations have completed.

See <http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf> for a very readable discussion of memory semantics in multicore systems.

One possible fix is to give up on sequential consistency. An alternative memory semantics is “weak consistency”, which only requires that the memory operations from each thread appear to be performed in the order issued by that thread. In other words in a weakly consistent system, if a particular thread writes to X and then writes to Y, the possible outcomes from reads of X and Y by any thread would be one of

(unchanged X, unchanged Y), or (changed X, unchanged Y), or (changed X, changed Y).

But no thread would see changed Y but unchanged X.

In a weakly consistent system, memory operations from other threads may overlap in arbitrary ways (not necessarily consistent with any sequential interleaving).

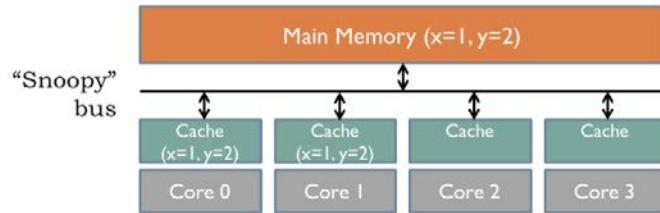
Note that our multicore cache system doesn’t itself guarantee even weak consistency. A thread that executes “write X; write Y” will update its local cache, but later cache replacements may cause the updated Y value to be written to main memory before the updated X value. To implement weak consistency, the thread should be modified to “write X; communicate changes to all other processors; write Y”. In the next section, we’ll discuss how to modify the caches to perform the required communication automatically.

Out-of-order cores have an extra complication since there’s no guarantee that successive ST instructions will complete in the order they appeared in the program. These architectures provide a BARRIER instruction that guarantees that memory operations before the BARRIER are completed before memory operation executed after the BARRIER.

There are many types of memory consistency — each commercially-available multicore system has its own particular guarantees about what happens when. So the prudent programmer needs to read the ISA manual carefully to ensure that her program will do what she wants. See the referenced PDF file for a very readable discussion about memory semantics in multicore systems.

Fix: “Snoopy” Cache Coherence Protocol

Idea: Have caches communicate over shared bus, letting other caches know when a shared cached value changes



Goal: minimize contention for snoopy bus by communicating only when necessary, i.e., when there's a shared value.

The problem with our simple multicore system is that there is no communication when the value of a shared variable is changed. The fix is to provide the necessary communications over a shared bus that's watched by all the caches. A cache can then "snoop" on what's happening in other caches and then update its local state to be consistent. The required communications protocol is called a "cache coherence protocol".

In designing the protocol, we'd like to incur the communications overhead only when there's actual sharing in progress, *i.e.*, when multiple caches have local copies of a shared variable.

Example: MESI Cache Coherence Protocol

Cache line:	State	Tag	Data
-------------	-------	-----	------

- **Modified** The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state.
- **Exclusive** The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a bus read request. Alternatively, it may be changed to the Modified state when writing to it.
- **Shared** Indicates that this cache line may be stored in other caches of the machine and is clean; it matches the main memory. The line may be discarded (changed to the Invalid state) at any time. **Writes to SHARED cache lines get special handling...**
- **Invalid** Indicates that this cache line is invalid (unused).

https://en.wikipedia.org/wiki/MESI_protocol

To implement a cache coherence protocol, we'll change the state maintained for each cache line.

The initial state for all cache lines is INVALID indicating that the tag and data fields do not contain up-to-date information. This corresponds to setting the valid bit to 0 in our original cache implementation.

When the cache line state is EXCLUSIVE, this cache has the only copy of those memory locations and indicates that the local data is the same as that in main memory. This corresponds to setting the valid bit to 1 in our original cache implementation.

If the cache line state is MODIFIED, that means the cache line data is the sole valid copy of the data. This corresponds to setting both the dirty and valid bits to 1 in our original cache implementation.

To deal with sharing issues, there's a fourth state called SHARED that indicates when other caches may also have a copy of the same unmodified memory data.

When filling a cache from main memory, other caches can snoop on the read access and participate if fulfilling the read request.

If no other cache has the requested data, the data is fetched from main memory and the requesting cache sets the state of that cache line to EXCLUSIVE.

If some other cache has the requested in line in the EXCLUSIVE or SHARED state, it supplies the data and asserts the SHARED signal on the snoopy bus to indicate that more than one cache now has a copy of the data. All caches will mark the state of the cache line as SHARED.

If another cache has a MODIFIED copy of the cache line, it supplies the changed data, providing the correct values for the requesting cache as well as updating the values in main memory. Again the SHARED signal is asserted and both the reading and responding cache will set the state for that cache line to SHARED.

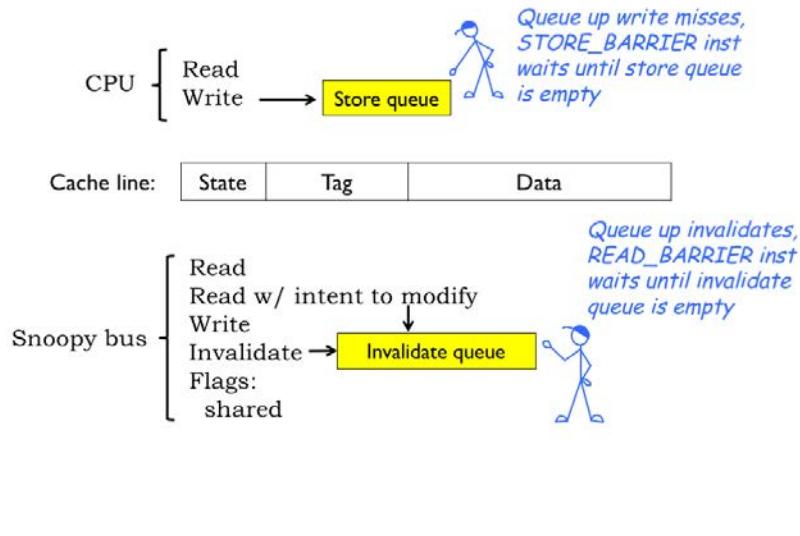
So, at the end of the read request, if there are multiple copies of the cache line, they will all be in the SHARED state. If there's only one copy of the cache line it will be in the EXCLUSIVE state.

Writing to a cache line is when the sharing magic happens. If there's a cache miss, the first cache performs a cache line read as described above. If the cache line is now in the SHARED state, a write will cause the cache to send an INVALIDATE message on the snoopy bus, telling all other caches to invalidate their copy of the cache line, guaranteeing the local cache now has EXCLUSIVE access to the cache line. If the cache line is in the EXCLUSIVE state when the write happens, no communication is necessary. Now the cache data can be changed and the cache line state set to MODIFIED, completing the write.

This protocol is called "MESI" after the first initials of the possible states. Note that the valid and dirty state bits in our original cache implementation have been repurposed to encode one of the four MESI states.

The key to success is that each cache now knows when a cache line may be shared by another cache, prompting the necessary communication when the value of a shared location is changed. No attempt is made to update shared values, they're simply invalidated and the other caches will issue read requests if they need the value of the shared variable at some future time.

The Cache Has Two Customers!

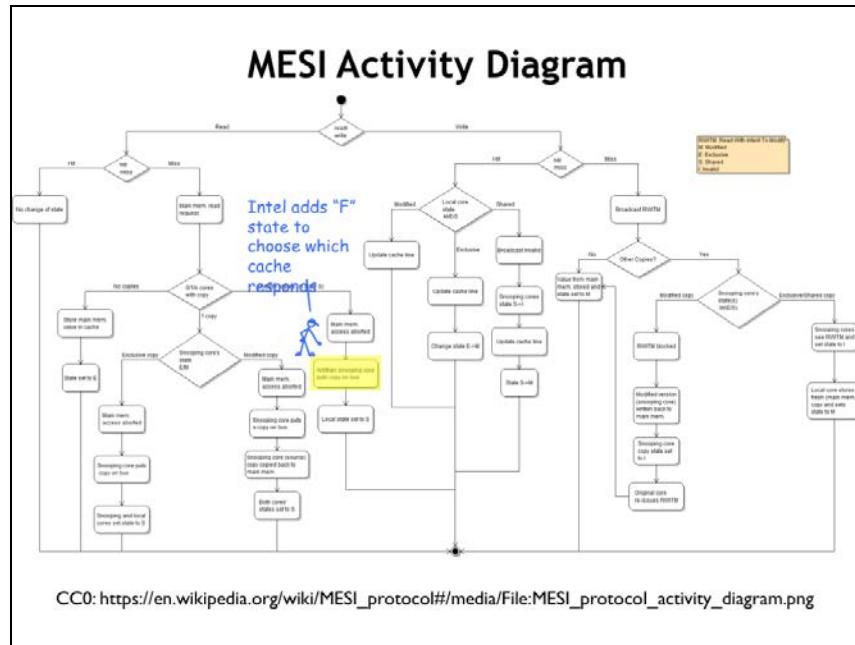


To support cache coherence, the cache hardware has to be modified to support two request streams: one from the CPU and one from the snoopy bus.

The CPU side includes a queue of store requests that were delayed by cache misses. This allows the CPU to proceed without having to wait for the cache refill operation to complete. Note that CPU read requests will need to check the store queue before they check the cache to ensure the most-recent value is supplied to the CPU. Usually there's a STORE_BARRIER instruction that stalls the CPU until the store queue is empty, guaranteeing that all processors have seen the effect of the writes before execution resumes.

On the snoopy-bus side, the cache has to snoop on the transactions from other caches, invalidating or supplying cache line data as appropriate, and then updating the local cache line state. If the cache is busy with, say, a refill operation, INVALIDATE requests may be queued until they can be processed. Usually there's a READ_BARRIER instruction that stalls the CPU until the invalidate queue is empty, guaranteeing that updates from other processors have been applied to the local cache state before execution resumes.

Note that the “read with intent to modify” transaction shown here is just protocol shorthand for a READ immediately followed by an INVALIDATE, indicating that the requester will be changing the contents of the cache line.

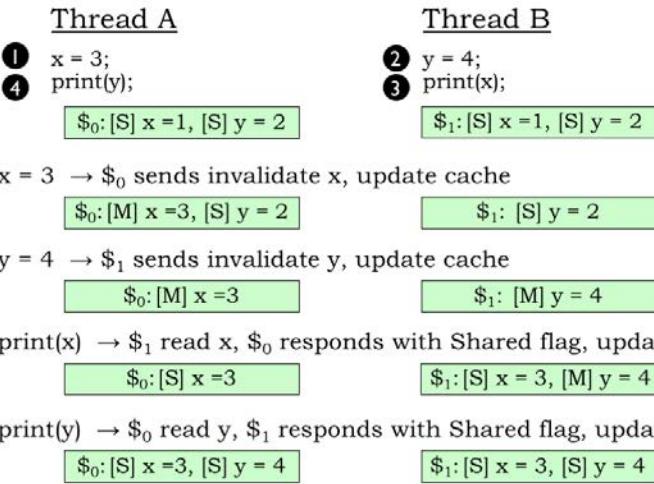


How do the CPU and snoopy-bus requests affect the cache state? Here in micro type is a flow chart showing what happens when. If you're interested, try following the actions required to complete various transactions.

Intel, in its wisdom, adds a fifth “F” state, used to determine which cache will respond to read request when the requested cache line is shared by multiple caches — basically it selects which of the SHARED cache lines gets to be the responder.

But this is a bit abstract. Let’s try the MESI cache coherence protocol on our earlier example!

Cache Coherence in Action



Here are our two threads and their local cache states indicating that values of locations X and Y are shared by both caches. Let's see what happens when the operations happen in the order (1 through 4) shown here. You can check what happens when the transactions are in a different order or happen concurrently.

First, Thread A changes X to 3. Since this location is marked as SHARED [S] in the local cache, the cache for core 0 (\$₀) issues an INVALIDATE transaction for location X to the other caches, giving it exclusive access to location X, which it changes to have the value 3. At the end of this step, the cache for core 1 (\$₁) no longer has a copy of the value for location X.

In step 2, Thread B changes Y to 4. Since this location is marked as SHARED in the local cache, cache 1 issues an INVALIDATE transaction for location Y to the other caches, giving it exclusive access to location Y, which it changes to have the value 4.

In step 3, execution continues in Thread B, which needs the value of location X. That's a cache miss, so it issues a read request on the snoopy bus, and cache 0 responds with its updated value, and both caches mark the location X as SHARED. Main memory, which is also watching the snoopy bus, also updates its copy of the X value.

Finally, in step 4, Thread A needs the value for Y, which results in a similar transaction on the snoopy bus.

Note the outcome corresponds exactly to that produced by the same execution sequence on a timeshared core since the coherence protocol guarantees that no cache has an out-of-date copy of a shared memory location. And both caches agree on the ending values for the shared variables X and Y.

If you try other execution orders, you'll see that sequential consistency and shared memory semantics are maintained in each case. The cache coherency protocol has done its job!

Parallel Processing Summary

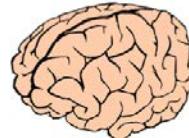
Prospects for future CPU architectures:

Pipelining - Well understood, but mined-out
Superscalar - At its practical limits
Vector/GPU - Useful for special applications

Prospects for future Computer System architectures:

Single-thread limits: forcing multicores, parallelism

Brains work well, with dismal clock rates ... parallelism?



Needed: NEW models, NEW ideas, NEW approaches

FINAL ANSWER: It's up to YOUR generation!

Let's summarize our discussion of parallel processing.

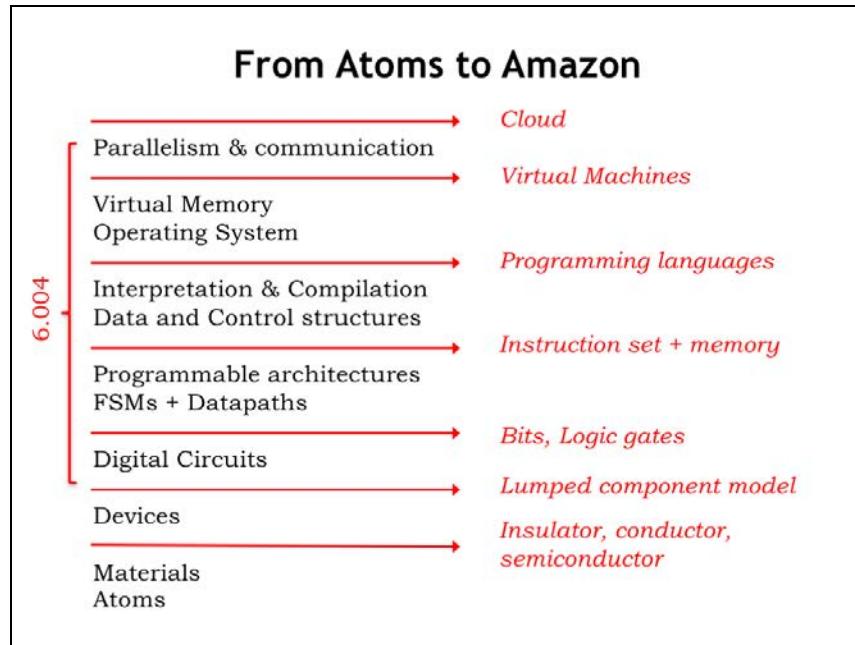
At the moment, it seems that the architecture of a single core has reached a stable point. At least with the current ISAs, pipeline depths are unlikely to increase and out-of-order, superscalar instruction execution has reached the point of diminishing performance returns. So it seems unlikely there will be dramatic performance improvements due to architectural changes inside the CPU core. GPU architectures continue to evolve as they adapt to new uses in specific application areas, but they are unlikely to impact general-purpose computing.

At the system level, the trend is toward increasing the number of cores and figuring out how to best exploit parallelism with new algorithms.

Looking further ahead, notice that the brain is able to accomplish remarkable results using fairly slow mechanisms (it takes ~.01 seconds to get a message to the brain and synapses fire somewhere between 0.3 to 1.8 times per second). Is it massive parallelism that gives the brain its “computational” power? Or is it that the brain uses a different computation model, *e.g.*, neural nets, to decide upon new actions given new inputs? At least for applications involving cognition there are new architectural and technology frontiers to explore. You have some interesting challenges ahead if you get interested in the future of parallel processing!

L22: Wrap-up

1. From Atoms to Amazon
2. The Power of Engineering Abstractions
3. 6.004: The Big Lesson
4. Things To Look Forward To...
5. Thinking Outside the Box
6. The End!



And now we've reached the end of 6.004. Looking back, there are two ways of thinking about the material we've discussed, the skills we've practiced, and the designs we've completed.

Starting at devices, we've worked our way up the design hierarchy, each level serving as building blocks for the next. Along the way we thought about design tradeoffs, choosing the alternatives that would make our systems reliable, efficient and easy to understand and hence easy to maintain.

In the other view of 6.004, we created and then used a hierarchy of engineering abstractions that are reasonably independent of the technologies they encapsulate. Even though technologies change at a rapid pace, these abstractions embody principles that are timeless. For example, the symbolic logic described by George Boole in 1847 is still used to reason about the operation of digital circuits you and I design today.

The Power of Engineering Abstractions

Good abstractions allow us to reason about behavior while shielding us from the details of the implementation.

Corollary: implementation technologies can evolve while preserving the engineering investment at higher levels.

- Leads to hierarchical design:
- Limited complexity at each level ⇒ shorten design time, easier to verify
 - Reusable building blocks

Cloud

Virtual Machines

Programming languages

Instruction set + memory

Bits, Logic gates

Lumped component model

Insulator, conductor, semiconductor

The power of engineering abstractions is that they allow us to reason about the behavior of a system based on the behavior of the components without having to understand the implementation details of each component.

The advantage of viewing components as “black boxes” implementing some specified function is that the implementation can change as long as the same specification is satisfied. In my lifetime, the size of a 2-input NAND gate has shrunk by 10 orders of magnitude, yet a 50-year-old logic design would still work as intended if implemented in today’s technologies.

Imagine trying to build a circuit that added two binary numbers if you had to reason about the electrical properties of doped silicon and conducting metals. Using abstractions lets us limit the design complexity at each level, shortening design time and making it easier to verify that the specifications have been met. And once we’ve created a useful repertoire of building blocks, we can use them again and again to assemble many different systems.

6.004: The Big Lesson

You've built, debugged, understood a complex computer from FETs to OS... what have you learned?

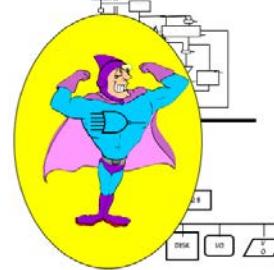
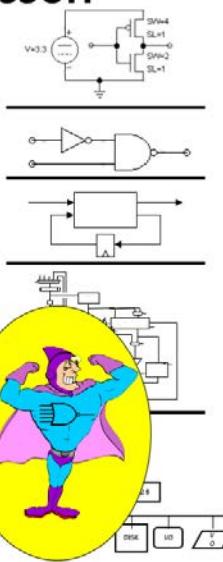
Engineering Abstractions:

- Understanding of their technical underpinnings
- Respect for their value
- Techniques for using them

But, most importantly:

The self assurance to discard them, in favor of new abstractions!

Good engineers *use* abstractions;
GREAT engineers *create* them!

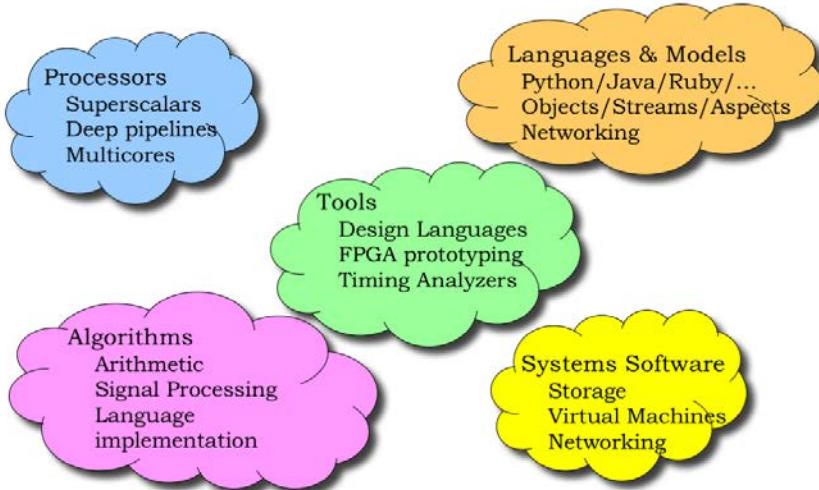


Our goal in 6.004 is to demystify how computers work, starting with MOSFETs and working our way up to operating systems. We hope you've understood the engineering abstractions we've introduced and had a chance to practice using them when completing the design problems offered in the labs.

We also hope that you'll also understand their limitations and have the confidence to create new abstractions when tackling new engineering tasks. Good engineers use abstractions, but great engineers create them.

Things to look forward to...

6.004 is only an appetizer!



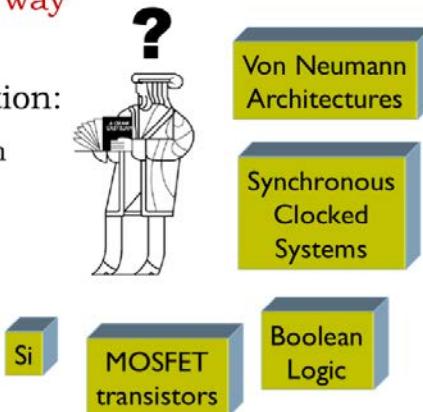
6.004 is an introductory course that only touches upon the basic principles used at each level of the design hierarchy. If a particular topic struck you as especially interesting, we hope you'll seek out a more advanced course that will let you dig deeper into that engineering discipline. Hundreds of thousands of engineers have worked to create the digital systems that are the engines of today's information society. As you can imagine, there's no end of interesting engineering to explore and master - so roll up your sleeves and come join in the fun!

Thinking Outside the Box

Will computers always
look and operate the way
computers do today?

Some things to question:

- Well-defined system “state”
- Silicon-based logic
- Logic at all
- Programming



What will be the engineering challenges of tomorrow? Here are a few thoughts about how the future of computing may be very different than the present.

The systems we build today have a well-defined notion of state: the exact digital values stored in their memories, produced by their logic components, and traveling along their interconnect. But computation based on the principles of quantum mechanics may allow us to solve what are now intractable problems using states described not as collections of 1's and 0's, but as interrelated probabilities that describe the superposition of many states.

We've built our systems using voltages to encode information and voltage-controlled switches to perform computation, using silicon-based electrical devices. But the chemistry of life has been carrying out detailed manufacturing operations for millennia using information encoded as sequences of amino acids. Some of the information encoded in our DNA has been around for millions of years, a truly long-lived information system! Today biologists are starting to build computational components from biological materials. Maybe in 50 years instead of plugging in your laptop, you'll have to feed it :)

Instead of using truth tables and logic functions, some computations are best performed neural networks that operate by forming appropriately weighted combinations of analog inputs, where the weights are learned by the system as it is trained using example inputs that should produce known outputs. Artificial neural nets are thought to model the operation of the synapses and neurons in our brains. As we learn more about how the brain operates, we may get many new insights into how to implement systems that are good at recognition and reasoning.

Again using living organisms as useful models, programming may be replaced by learning, where stimulus and feedback are used to evolve system behavior. In other words, systems will use adaptation mechanisms to evolve the desired functionality rather than have it explicitly programmed.

This all seems the stuff of science fiction, but I suspect our parents feel the same way about having conversations with Siri about tomorrow's weather.

THE END!

*Computing is slow...
The future is in your hands.
Start innovating!
-- 6.004 Staff*

*The only problem
with Haiku is that you just
get started and then
-- Roger McGough*



Thanks for joining us here in 6.004. We've enjoyed presenting the material and challenging you with design tasks to exercise your new skills and understanding. There are interesting times ahead in the world of digital systems and we can certainly use your help in inventing the future!

We'd welcome any feedback you have about the course so please feel free leave comments in the forum. Good bye for now... and good luck in your future studies...