A Student's Guide
to the Digital World

by Margaret Chong

Acknowledgements

Special thanks go out to my best friends in the world – Joe and Bonnie. Where would I be without your support and hugs. =)

Thank you Professor Terman and Professor Ward for giving me the opportunity to teach and realize one of my true passions.

3

Table of Contents

Binary

There are two types of binary used in 6.004 – unsigned and signed (also known as two's complement). Let's take a binary number **1100_2** and figure out its decimal equivalent.

Unsigned 2^3 2^2 2^1 2^0 $\Longrightarrow$ $2^3 + 2^2$ $= 8 + 4 = 12$

1	1	0	0

Signed 2^3 2^2 2^1 2^0 $\Longrightarrow$ $(-1)(2^3) + 2^2 = -8 + 4 = -4$

(1) The space to the left of the decimal point is for 2^0, increasing by a power of two as you go further left, decreasing by a power of two as you go right
(2) For each 1 in the binary number, add the corresponding power of two to the sum
(3) Unsigned binary - everything in the sum is positive
(4) Signed binary – the leftmost space in the number is negative

> Note: If the leftmost space is a 0, then everything in the sum is positive. For example,
> $1100_2 = -8 + 4 = -4$
> $01100_2 = 8 + 4 = 12$ ← the leftmost 0 means we don't count the -16

Octal and Hexadecimal

Octal is base 8 and is made of groups of 3 bits
Hexadecimal is base 16 and is made of groups of 4 bits

Binary	1	1	1	1	0	0	1	1	0	1	1	1
Octal	7			4			6			7		
Hexadecimal	F				3				7			

Useful Conversion Tables:

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binary	Hex	Binary	Hex
0000	0	1000	8
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

Signed Binary Addition

Binary addition is the same as regular addition, except you only have 0's and 1's.

```
                1           11          111          111          111
    3         00011       00011       00011        00011        00011
  + 5       + 00101     + 00101     + 00101      + 00101      + 00101
    8    ⇒      0    ⇒      00    ⇒    000    ⇒    1000    ⇒   01000
```

1+1 = 2 = 10 in binary, so carry the 1
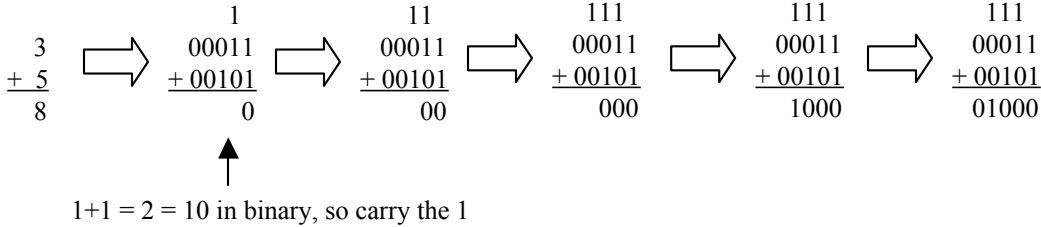
Most machines have a fixed number of bits. In the example above, the operands are both 5 bits wide, therefore the result can only be expressed in 5 bits. This introduces the possibility that the sum might be incorrect.

```
                1          11         111         111
    3         0011       0011        0011        0011
  + 5       + 0101     + 0101      + 0101      + 0101
   ??    ⇒     0    ⇒     00    ⇒    000    ⇒   1000   ⇐
```

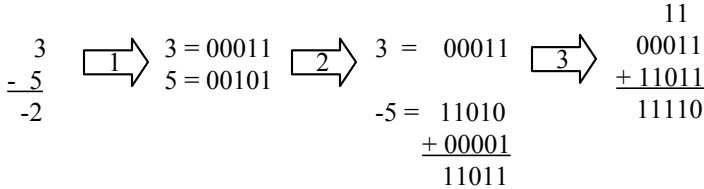**-8 is incorrect!**

We cannot express 8 in four bits

```
                           1
   -8         1000        1000
  + -8      + 1000      + 1000
   ??    ⇒           ⇒    0000   ⇐
```

0 is incorrect!

We cannot express −16 in four bits
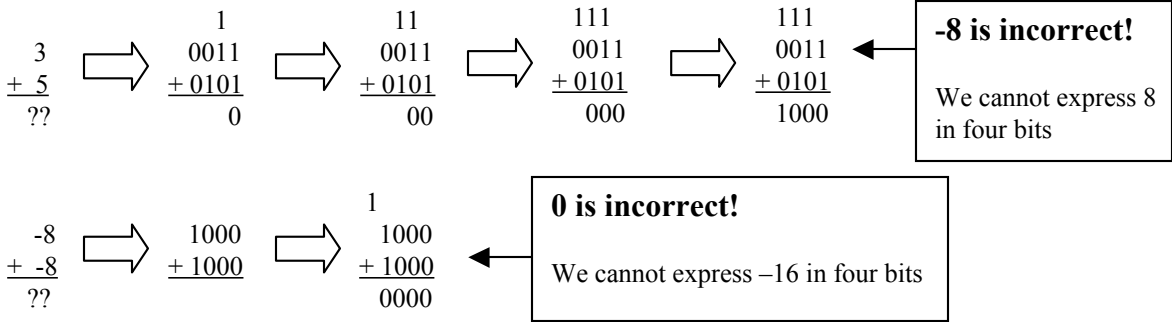
Signed Binary Subtraction

Let's do this by example, say we're subtracting 3 minus 5

(1)   Represent 3 and 5 in binary
(2)   Figure out the binary representation for *negative* 5 by flipping the bits and adding one
(3)   Compute 3 plus negative 5

```
                                                        11
    3         3 = 00011          3 =   00011           00011
  - 5    ①⇒  5 = 00101     ②⇒                   ③⇒   + 11011
   -2                             -5 = 11010           11110
                                      + 00001
                                        11011
```

Note that, like in binary addition, we are constrained to a fixed number of bits. Therefore, you might encounter cases where the "flip the bits and add one" cannot be represented in some fixed number of bits. You may also find that the result cannot be represented in some fixed number of bits.
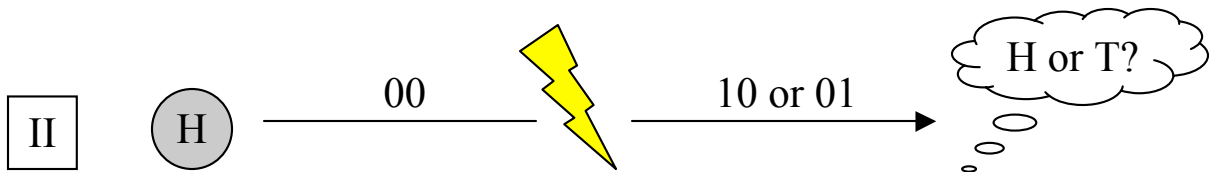
Hamming Distance

One factor to consider when transmitting a message is the possibility that the message might get corrupted in transit. For example, say we're transmitting the outcome of a coin flip over the internet. There are two possible outcomes – heads or tails.

One possible encoding uses only one bit: heads = "0" and tails = "1". We tell our friend that the coin flipped heads by transmitting a "0." However, a routing wire is struck by lightning, changing the bit to a "1." Our friend will incorrectly think tails.
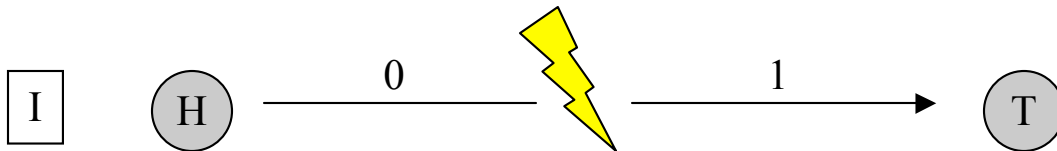
If we increase the number of bits in the encoding to two bits (heads = 00 and tails = 11), we can detect single-bit errors. However, we still cannot correct them.

With three bits, however, we *can* detect single and two-bit errors, and we can correct single bit errors (heads = 000 and tails = 111).
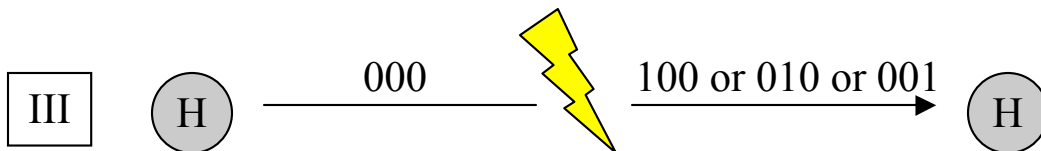
For two encodings of the *same* length, the Hamming Distance (**D**) is the number of bits you need to change to turn one into the other. We increased the Hamming Distance by one in each step of this example. (I) D = 1   (II) D = 2   (III) D = 3.

If you have a Hamming Distance of **D**, then you can:

DETECT (D − 1) bit errors   and   **CORRECT** $\frac{(D-1)}{2}$ bit errors

Information

A message carries information if it tells you something new. The higher the probability of receiving that message, the less information that message carries. There are three basic equations you need to know:

Non-equivalent probability for all choices   *Equivalent probability for all choices*

$$\text{\# bits of information} = \log_2 \left[\frac{1}{p_i} \right]$$

$$\text{\# bits of information} = \log_2 \left[\frac{\text{\# choices before}}{\text{\# choices after}} \right]$$
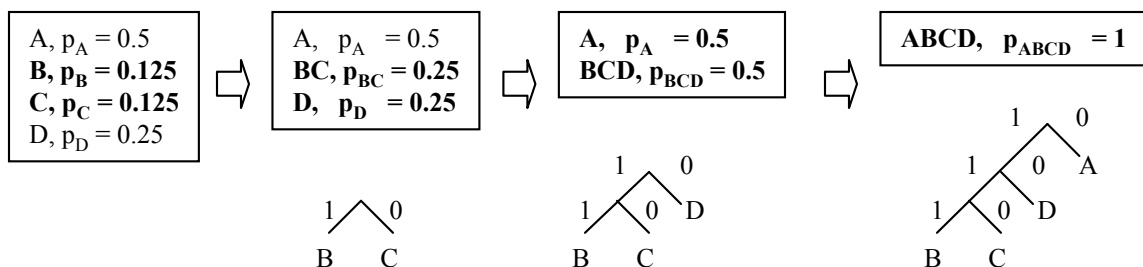
$$\text{Average \# bits of information} = \Sigma \, p_i \log_2 \left[\frac{1}{p_i} \right]$$

Huffman Encoding

We can encode symbols using a sequence of bits. Huffman encoding is merely one of the ways we can assign a symbol to a sequence of bits. The algorithm proceeds as follows:

(1) Choose the two members/subtrees with the lowest probability
(2) Label one a "0" and the other a "1"
(3) Connect them into a new subtree, and add their probabilities to produce the probability of the new subtree
(4) Repeat 1-3 until you're left with one big tree

This example illustrates Huffman encoding for the symbols A, B, C, and D with their corresponding probabilities:

A, p_A = 0.5
B, p_B = 0.125
C, p_C = 0.125
D, p_D = 0.25

$\Rightarrow$

A, p_A = 0.5
BC, p_{BC} = 0.25
D, p_D = 0.25

1 $\diagup\diagdown$ 0
B   C

$\Rightarrow$

A, p_A = 0.5
BCD, p_{BCD} = 0.5

1 $\diagup\diagdown$ 0
1 $\diagup\diagdown$ 0 D
B   C

$\Rightarrow$

ABCD, p_{ABCD} = 1

1 $\diagup\diagdown$ 0
1 $\diagup\diagdown$ 0 A
1 $\diagup\diagdown$ 0 D
B   C

Encodings: A = 0
                  B = 111
                  C = 110
                  D = 10

The Digital Abstraction
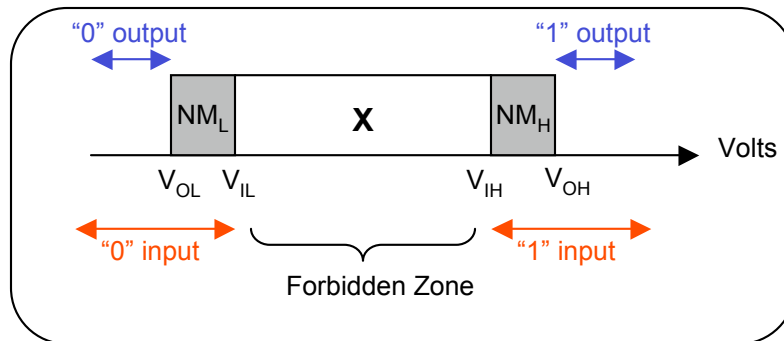
In the digital world, a voltage can represent a valid "0", a valid "1", or an invalid signal "X" (a voltage in the forbidden zone). Voltage is continuous, therefore we define boundaries to indicate whether it's a 0, 1, or X.

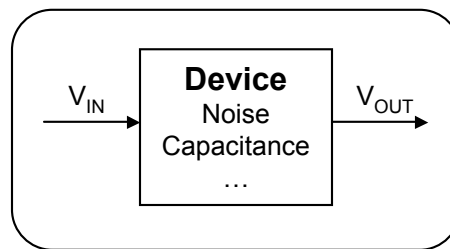The world is not ideal, however, so we must ensure that a "0" will never be mistaken for anything else. This is done by applying stricter boundaries to the outputs of combinational logic blocks (V_{OL} and V_{OH}) and more lenient boundaries on the inputs (V_{IL} and V_{IH}).

The difference between these two boundaries is called a **noise margin**. There are two noise margins: $NM_{L=LOW}$ and $NM_{H=HIGH}$. Think of NM_L as the amount of voltage it takes to turn a "0" into an "X", and NM_H as the amount of voltage it takes to turn a "1" into an "X."
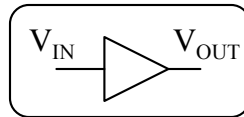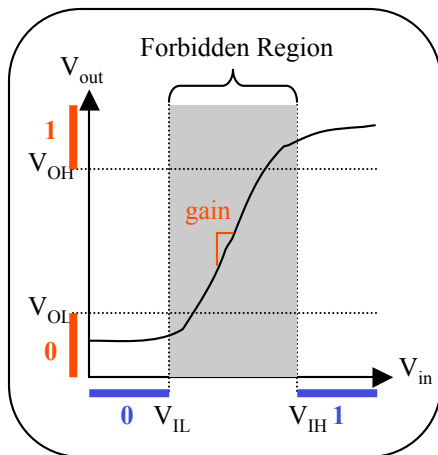
$$NM_L = V_{IL} - V_{OL}$$

$$NM_H = V_{OH} - V_{IH}$$

Noise Margin of a combinational logic block = $NM = \min(NM_L, NM_H)$

Combinational Devices and Static Discipline ⎯⎯⎯

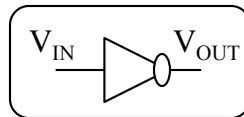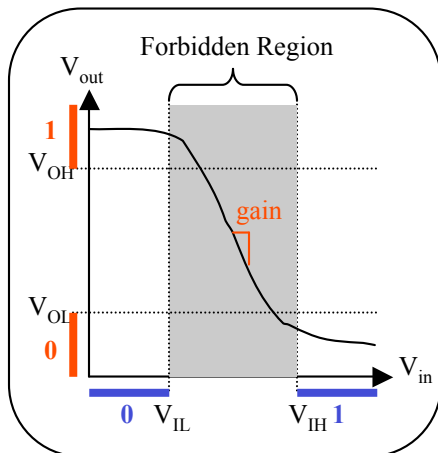## Buffer

V_{IN} ▷ V_{OUT}

Static Discipline:
If $V_{IN} \leq V_{IL}$ then $V_{OUT} \leq V_{OL}$
If $V_{IN} \geq V_{IH}$ then $V_{OUT} \geq V_{OH}$

A buffer basically takes in a "0" and outputs a "0", or takes in a "1" and outputs a "1". The buffer symbol is shown above. The graph on the left is the buffer's Voltage Transfer Curve (VTC). The valid regions are located in the lower left and upper right corners of the curve. The shaded region in the middle is the invalid Forbidden Region.

Gain is the slope in the forbidden region of the VTC. Note that the |gain| > 1 in order for the buffer to be classified as a combinational device. If it isn't, you might get negative noise margins or the signal will degrade when you cascade the device.

Inverter

V_{IN} ▷○ V_{OUT}

Static Discipline:
If $V_{IN} \leq V_{IL}$ then $V_{OUT} \geq V_{OH}$
If $V_{IN} \geq V_{IH}$ then $V_{OUT} \leq V_{OL}$

An inverter takes in a "0" and outputs a "1", or takes in a "0" and outputs a "1". Note that the inverter must also have |gain| > 1 in order to be a combinational device.

When choosing values for V_{OL}, V_{IL}, V_{IH}, V_{OH}, there is no set algorithm. The best way is to gain intuition from practice.

Some rules of thumb might include:

(1)   Pick V_{OH} and V_{OL} around the corners where the flat region meets the steep region
(2)   If you're trying to maximize the noise margins, try to chose the highest V_{OH} and the lowest V_{OL} possible
(3)   If you're trying to maximize the noise margins, first find a V_{OL}, V_{IL}, V_{IH}, V_{OH} that adheres to the static discipline. Then try to make one of the noise margins bigger by altering one of V_{OL}, V_{IL}, V_{IH}, V_{OH} and see if the new values adheres to the static discipline.

NFET pulldowns
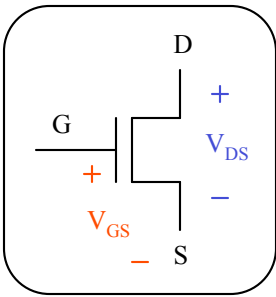
A NFET can be categorized into three "analog" regions – cutoff, linear, and saturation - and two "digital"regions – on and off.

Off/Cutoff: $V_{GS} < V_{TH}$
On/Linear: $V_{GS} \geq V_{TH}$, $V_{DS} < V_{DSsat}$
On/Saturation: $V_{GS} \geq V_{TH}$, $V_{DS} \geq V_{DSsat}$

$$\text{Where } V_{DSsat} = V_{GS} - V_{TH}$$
$$V_{TH} = \text{a set Threshold Voltage}$$

PFET pullups
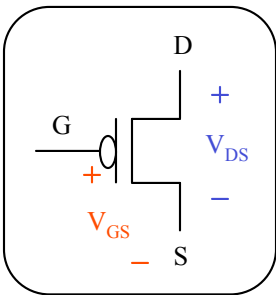
A PFET also has three "analog" regions – cutoff, linear, and saturation - and two "digital"regions – on and off.

Off/Cutoff: $V_{GS} > V_{TH}$
On/Linear: $V_{GS} \leq V_{TH}$, $V_{DS} > V_{DSsat}$
On/Saturation: $V_{GS} \leq V_{TH}$, $V_{DS} \leq V_{DSsat}$

$$\text{Where } V_{DSsat} = V_{GS} - V_{TH}$$

CMOS Inverter
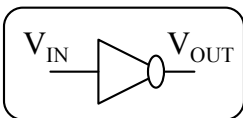
IN	OUT
0	1
1	0

A CMOS inverter has one PFET between power and the output, and one NFET between the output and ground. Think of the PFET and NFET as switches. A PFET "switch" closes when IN=0, and ties the output to V_{DD}. A NFET "switch" closes when IN=1, and ties the output to ground.

IN = 0
PFET ON
NFET OFF
OUT = 1

IN = 1
PFET OFF
NFET ON
OUT = 0
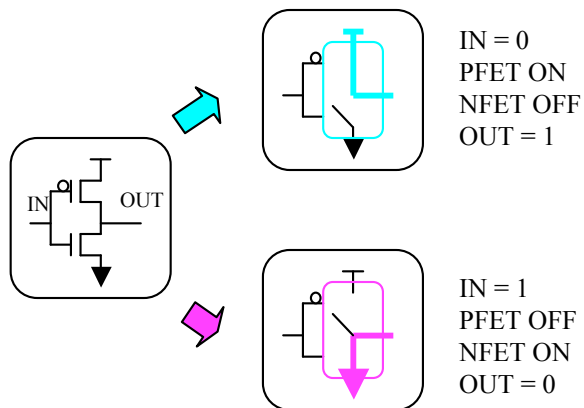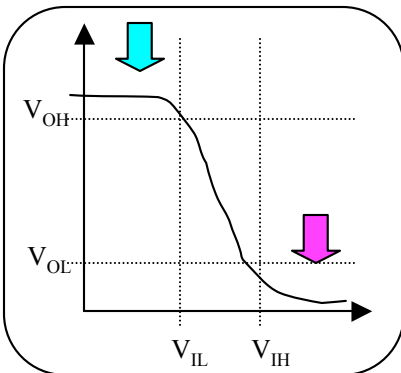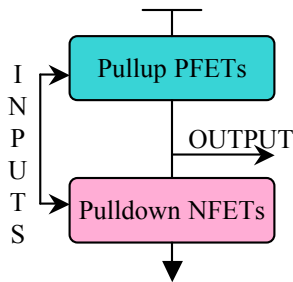
CMOS Logic

A CMOS Logic gate consists of a bunch of PFETs in the pullup (between power and the output), and a bunch of NFETs in the pulldown (between the output and ground).

The pullup may only consist of PFETs and *the pulldown may only consist of NFETs*. Otherwise, the logic gate will not have sufficient gain in the forbidden area. Also, since the NFETs always go in the pulldown, *CMOS logic gates can only implement negative logic.*

The pullup and pulldown must complement each other. If two NFETs are in parallel in the pulldown, then the corresponding PFETs must be in series in the pullup. The opposite is also true – NFETs in series must have corresponding PFETs in parallel.
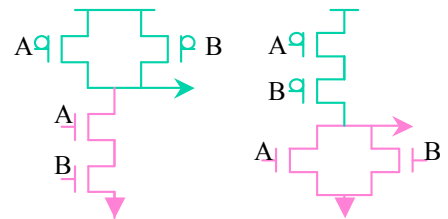
If you have a formula that you need to implement as a CMOS logic gate:

A*B means A and B are in series in the pulldown and in parallel in the pullup
A+B means A and B are in parallel in the pulldown and in series in the pullup

First draw the pulldown. Then after the pulldown is done, draw the pullup as the complement of the pulldown. In the example below, we must draw the CMOS logic gate for **OUTPUT = A * B + C**.

OUT = $\overline{A*B + C}$
PFET: A and B in parallel

OUT = $\overline{A*B + C}$
PFET: A*B in series with C

OUT = $\overline{A*B + C}$
NFET: A and B in series

OUT = $\overline{A*B + C}$
NFET: A*B in parallel with C
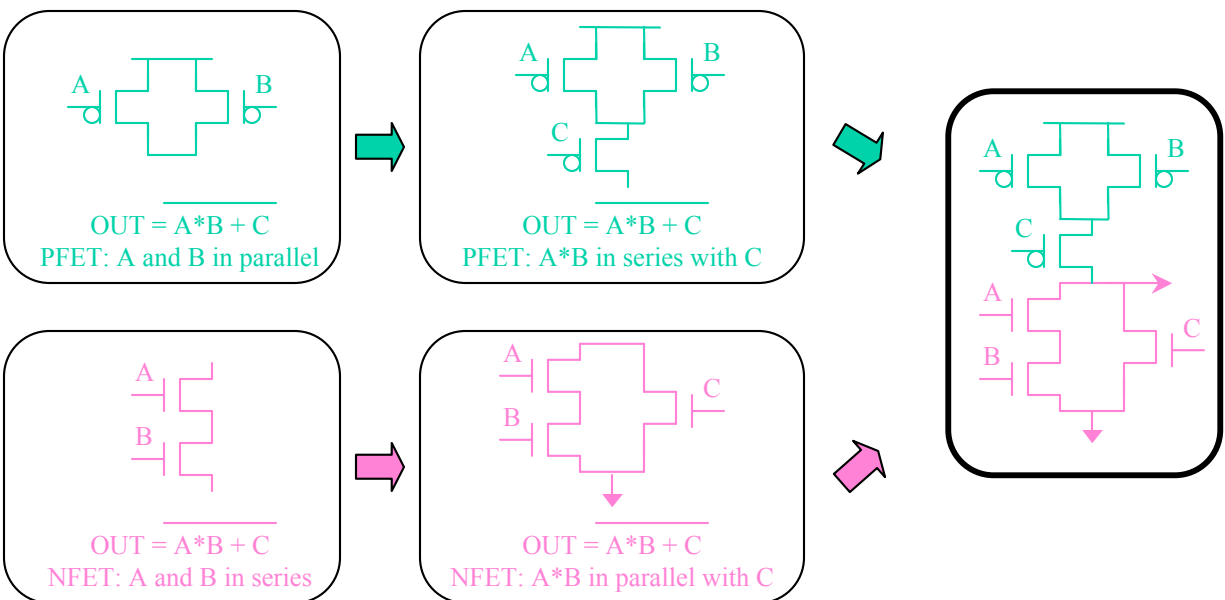
If you have a CMOS logic gate and you wish to derive the formula, follow these steps:

1) Look at the pulldown only
2) A and B in series means there is an "A*B" in the equation
3) A and B in parallel means there is an "A+B" in the equation
4) Negate the entire thing after you're done with all the pulldown NFETs.

Step 4 is necessary because you're dealing with NFETs. You will probably want to do a few similar practice problems to gain intuition.
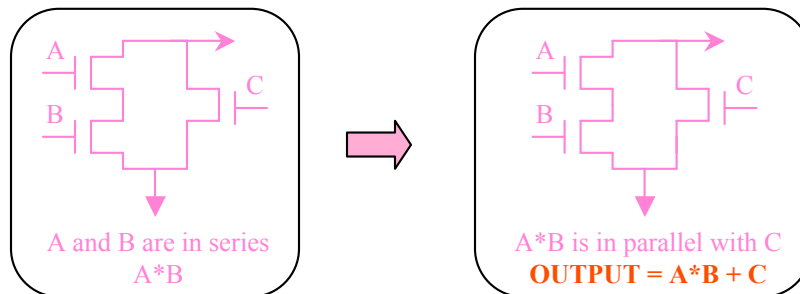
CMOS Logic Gates

INVERTER
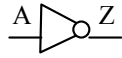
A	Z
0	1
1	0

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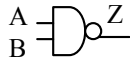
A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR

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

XNOR

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

BUFFER
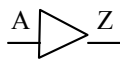
A	Z
0	0
1	1

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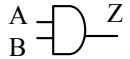
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

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

XOR

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

The figures above contain eight typical standard CMOS logic gates, their symbolic representation, and their truth tables. You can combine logic gates to implement *almost* any function. However, keep in mind that negative logic (inverter, nand, nor, xnor) is usually smaller and faster than positive logic (buffer, and, or, xor).

Boolean Algebra

A * B means "A and B"

$\overline{A * B}$ means "A nand B"

A + B means "A or B"

$\overline{A + B}$ means "A nor B"

A $\oplus$ B means "A xor B"

$\overline{A \oplus B}$ means "A xnor B"

DeMorgan's Law

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

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

Buildings Circuits with Logic Gates

If you wish to implement a function $Z = A*B + C$ using only standard logic gates such as those pictured above, you can implement it in multiple ways. The two possibilities below (all negative logic on the left and all positive logic on the right) are not necessarily faster or smaller than one another – it depends on the size and speed of the standard logic gates.

$$Z = \overline{\overline{A*B + C}}$$
$$Z = \overline{\overline{A*B} * \overline{C}}$$

← Invert it twice (two bars)
DeMorganize the bottom bar

$$Z = A*B + C$$
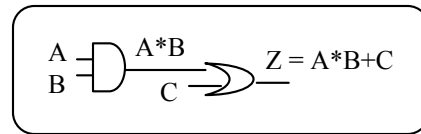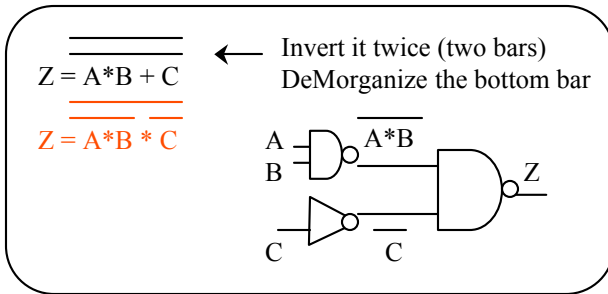
If you wish to derive a logic equation from an existing circuit, try to derive intermediate equations, starting at the inputs and working incrementally towards the output. You can use boolean algebra to manipulate the formula to whatever format you prefer.

$$Z = \overline{\overline{A} + \overline{A*B} + \overline{C}}$$
$$Z = \overline{\overline{A}} * \overline{\overline{A*B}} * \overline{\overline{C}}$$
$$Z = A * A*B * C$$
$$Z = A*B*C$$
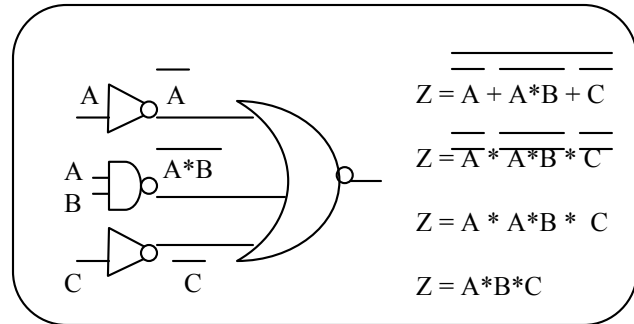
Contamination Delay

The contamination delay of a circuit, tcd, is the *minimum* amount of time it takes for an invalid input to propagate to an invalid output. The figure to the right illustrates the measurement of a buffer's contamination delay.
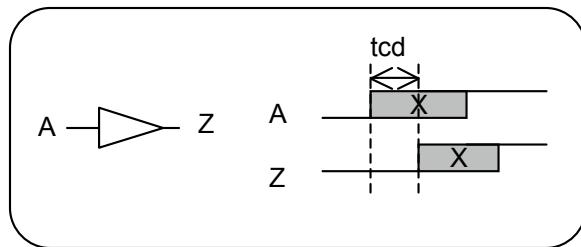
For a circuit with multiple inputs or paths to the output, the contamination delay of the entire circuit is the sum of the contamination delays in the *shortest* path. For example, the circuit to the right has three paths, with the shortest contamination delay through the bottom path for a tcd = (2.5 + 3) = 5.5
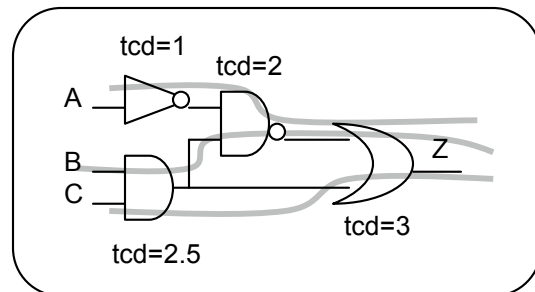
Propagation Delay

The propagation delay, tpd, is the *maximum* amount of time it takes for a valid input to propagate to a valid output. The figure to the right illustrates the measurement of a buffer's propagation delay.
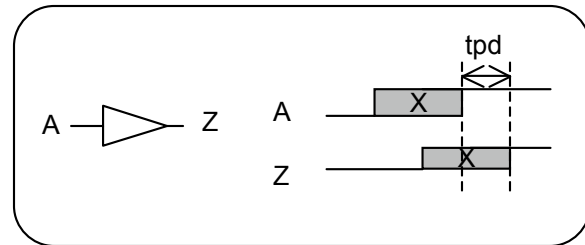
For a circuit with multiple inputs or paths to the output, the propagation delay of the entire circuit is the sum of the propagation delays in the *longest* path. For example, the circuit to the right has three paths, with the longest propagation delay through the middle path for a tpd = (3 + 2 + 5) = 10
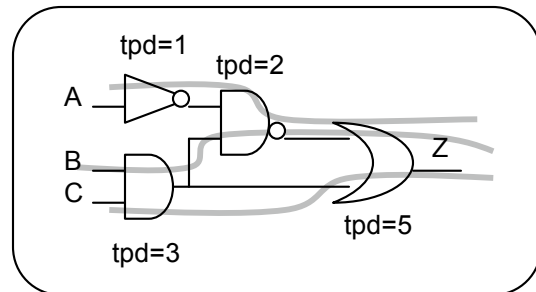
Sum of Products

Sum of Products is an easy way to derive a formula from a truth table. The formula is a sum of products. Each product represents one row where the output Z = 1. For each product/row, write down all the inputs and put individual bars over the inputs that equal zero.

If there are fewer Z = 0 than Z = 1 outputs, you can find the sum of products using the same method, but use the Z = 0 rows as the products and negate the entire sum of products.

From here, you should be able to find the *minimal* sum of products by reducing the formula using boolean arithmetic. This can be quite tedious. An easier method for finding the minimal sum of products is presented in a subsequent section on Karnaugh Maps.

A B C	Z	
0 0 0	0	
0 0 1	0	
0 1 0	1	→ $\overline{A}$ B $\overline{C}$
0 1 1	0	
1 0 0	0	
1 0 1	1	→ A $\overline{B}$ C
1 1 0	0	
1 1 1	1	→ A B C

$$Z = \overline{A}B\overline{C} + A\overline{B}C + ABC$$
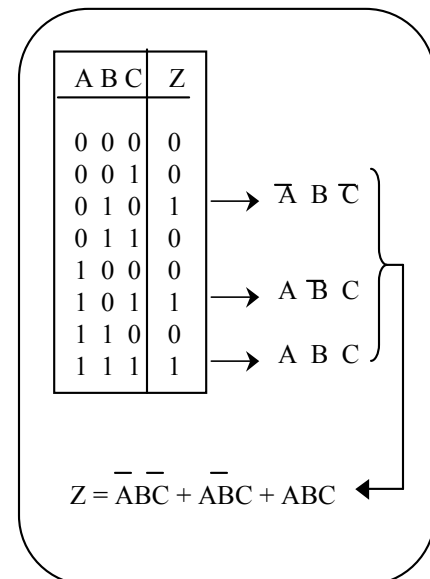
Karnaugh Maps

Karnaugh Maps are extremely useful for finding the minimal sum of products.

① Translate a truth table to a Karnaugh Map.
→ Note the order of the AB field – neighbors must differ by one bit

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	X
0	1	1	0
1	0	0	1
1	0	1	X
1	1	0	X
1	1	1	X

B ___ A ___

AB / C	00	01	11	10
0	1	X	X	1
1	0	1	X	X
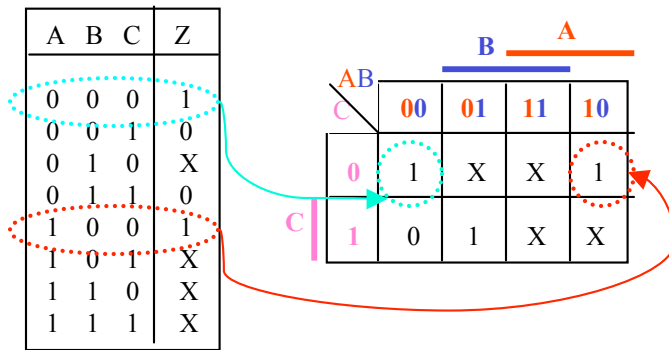
② Find and circle all 1's
- Don't Cares 'X' can be circled, if needed
- Circles must have dimensions that are a power of 2 (1, 2, 4, 8, etc.)
- You can overlap circles
- The Karnaugh Map folds over, so circles can go off the sides
- Look for the largest groups of 1's
- Try to get the smallest number of circles

B ___ A ___

AB / C	00	01	11	10
0	1	X	X	1
1	0	1	X	X

B ___ A ___

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1
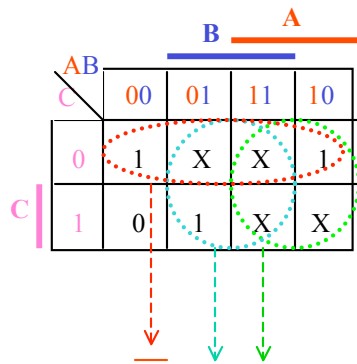
③ OUT = $\overline{C}$ + B + A
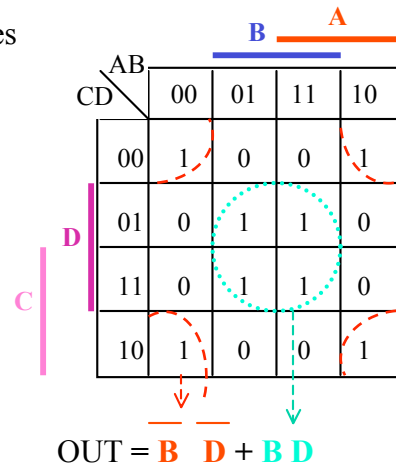
OUT = $\overline{B}\ \overline{D}$ + B D

Lenient Combinational Logic

Take a karnaugh map that is directly derived from the circuit. Each gap between neighboring circles is an opportunity for a glitch. The circuit is lenient if there are no gaps.

AB / C	00	01	11	10
0	0	1	1	0
1	0	1	1	1
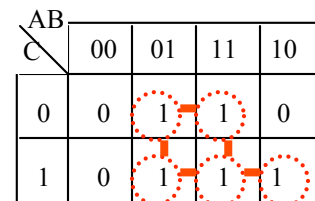
Mux

A d-input mux has a selector input, *d* data inputs (*d* is a power of two), and one output.
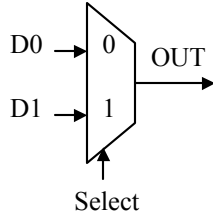
D0	D1	Select	OUT
0	X	0	0
1	X	0	1
X	0	1	0
X	1	1	1

If Select = 0, then OUT = D0
If Select = 1, then OUT = D1

ROM / PLA
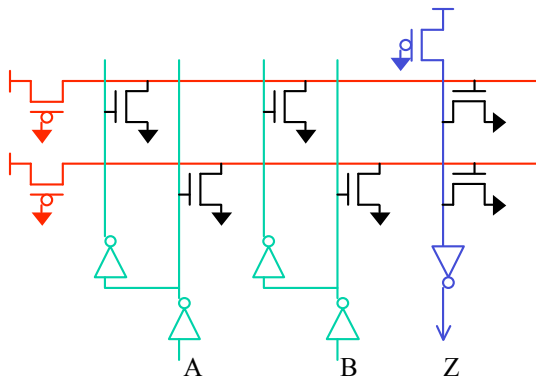
A Read-Only Memory and a Programmable Logic Array consist of address input lines, word lines, and bit lines. You can add NFETs to configure the ROM or PLA to perform many functions. Each word line and bit line is driven by a weak PFET whose input is tied to ground. The weak PFET serves to drive the wire high when there is no path to ground, but is easily overpowered by a NFET.

A ROM specifies all combinations of inputs. A PLA looks like a small ROM because unnecessary FETs and wires are eliminated. If you are given the PLA pictured above, follow these steps to figure out the formula for the output:

1. Write the logic equivalent of each word line at the top
2. Each bit line driving the gate of a NFET is a *product* in the sum of products
3. For each NFET drain attached to the bit line, negate the gate input.
   Repeat to find product.
4. Z = OR of all the products

$Z = A B + \overline{A} \overline{B}$ ← Sum of products
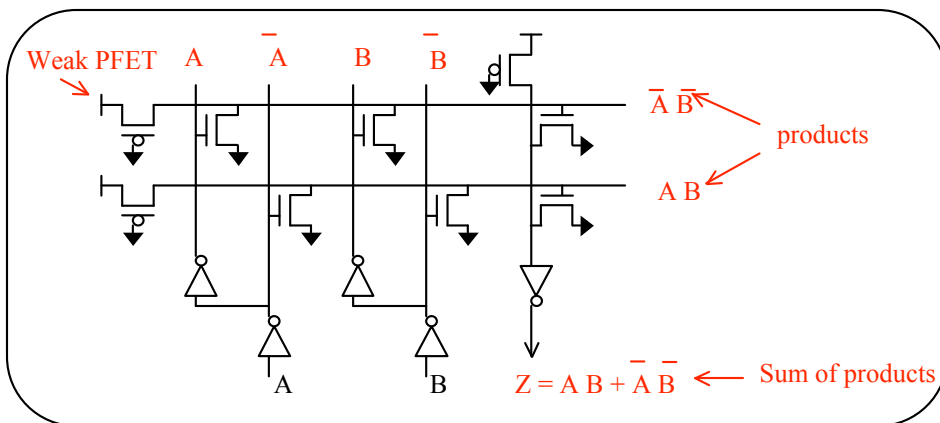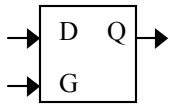
Latch

A latch has one enable input G, one data input D, and one data output Q. Think of a latch as a gate. If the latch is enabled, the gate opens and the input data propagates through to the output. If the latch is not enabled, then the gate is closed, and the output remains the same.
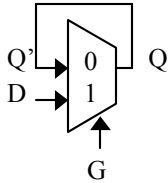
One way to implement the latch is to use a two-input mux. This introduces some timing constraints on the inputs of the latch. The data input D must be valid T_{SETUP} before G falls and D must stay valid for T_{HOLD} after G falls.

$T_{SETUP} = 2 T_{PD}$ because …

(1) One T_{PD} for D to propagate through to Q
(2) Hold Q' one T_{PD} for mux to promise a stable output after G falls
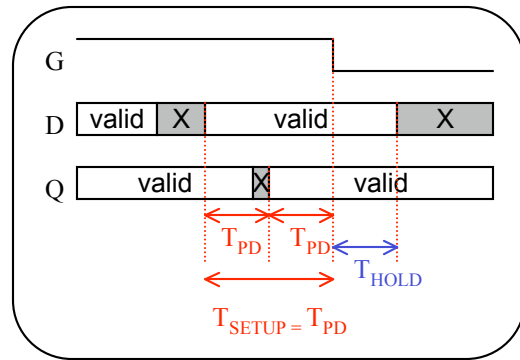
Flip Flop
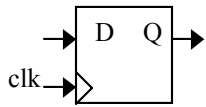
A flip flop, which is interchangeably referred to as a register, has a clock input, a data input D, and a data output Q. D propagates to Q *only at the rising 0 → 1 edge of the clock*. Otherwise, the output stays the same.

A flip flop is implemented with two latches – a master and a slave. As in the 6.004 lecture, think about this implementation as two tollbooth gates that open one at a time. The master passes the data into the middle when the clock is low. When the clock goes high, the master immediately closes to trap the data, and the slave opens to let the trapped date through to the output.

Flip Flop Implementation – Two Latches

CLK = 0

CLK = 1

Flip Flop Timing

t_{SETUP} = Minimum time D holds before rising CLK edge
t_{HOLD} = Minimum time D holds after rising CLK edge
t_{PD} = Maximum CLK→Q propagation delay
t_{CD} = Minimum CLK→Q contamination delay
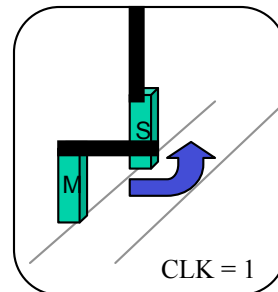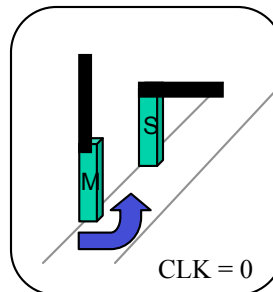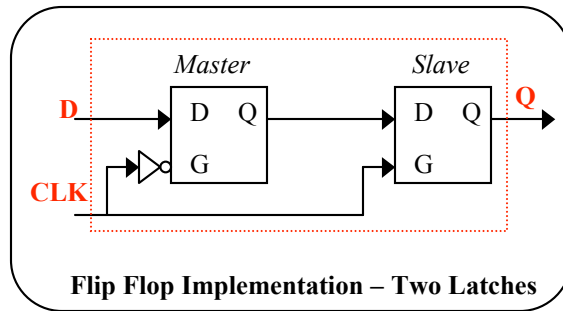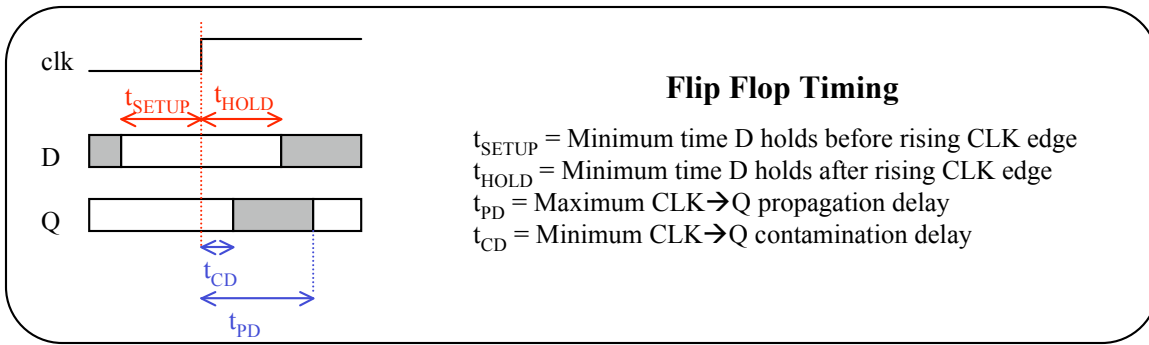
$t_{CD,MASTER} > t_{HOLD,SLAVE}$ to ensure that data will not pass through both the master and the slave while both are transitioning. If this inequality does not hold, then you can ensure that the flip flop works by inserting a buffer between the master and slave such that

$t_{CD,MASTER} + t_{CD,BUFFER} > t_{HOLD,SLAVE}$

Flip Flop Circuits
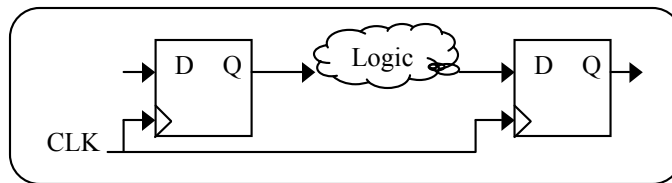
The circuit contains register 1 on the left, some random combinational logic, and register 2 on the right. There are two inequalities that must hold in order for this circuit to work.

$t_{CD,REG1} + t_{CD,LOGIC} > t_{HOLD,REG2}$
to ensure that data coming out of reg1 one does not corrupt the data going in to reg2.

$t_{CLK} \geq t_{PD,REG1} + t_{PD,LOGIC} + t_{SETUP,REG2}$ where t_{CLK} = the clock period
to ensure that reg2's input will meet reg2's setup time.

Finite State Machine

Current state			Next state		Output Based on Current State
s1	s0	IN	s1'	s0'	OUT
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

A finite state machine (FSM) contains input signals, state signals, and output signals. The example FSM pictured here contains one input, two state signals s1 and s0, and one output. The circles contain the states for s1 and s0 and tell us whether we're in state 00, 01, 10, or 11. The arcs tell us the next state based on the input.

An FSM must designate a transition for every possible input. Also, the arcs leaving a state must not allow two choices for the same input.
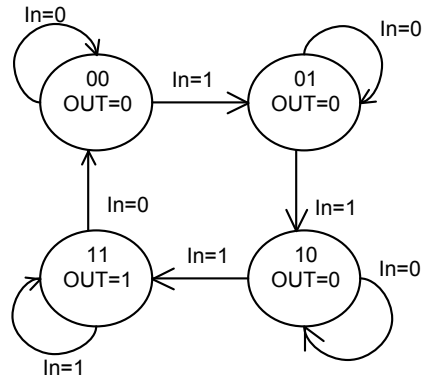
Metastability
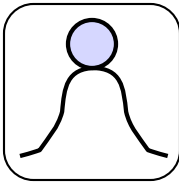
Imagine a ball at the top of a hill. If the hill is not flat, there are three possible states – a stable state at the bottom left, a stable state at the bottom right, and a metastable state at the top of the hill. That ball is in a metastable state because (1) the top of the hill is an unstable state – a small perturbation will make it roll down the hill, and (2) the ball will *eventually* settle to a stable state at the bottom of the hill.
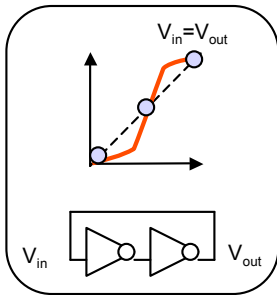
A metastable state in the digital world has the following properties:
1) Invalid
2) Unstable – a small perturbation changes it to a 0 or 1
3) It will settle in *unbounded* time
4) The longer you wait, the more likely it has stabilized

The two-inverter configuration has three possible states where $V_{in} = V_{out}$: two stable states at valid 1 and 0, and one metastable state in the forbidden region.

You cannot build certain devices with perfect reliability because they *cause* metastability problems. Some example systems are shown below:

Bounded-time Arbiter CANNOT BUILD
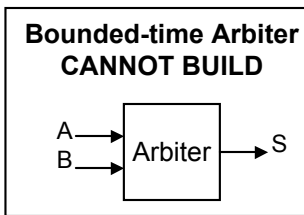
Sample S *at a specified time after A and B have risen* to find out which rising edge came first – A or B. The Arbiter may enter a metastable state if A and B rise at exactly the same time. The Arbiter will eventually come to decision in *unbounded* time but might be metastable and invalid when S is sampled.

Unbounded-time Arbiter CAN BUILD
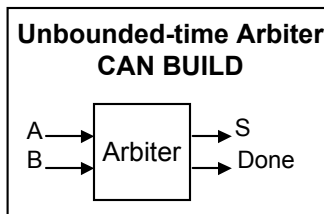
Sample S *when the Done signal is high* to find out which rising edge came first – A or B. S may enter a metastable state if A and B rise at exactly the same time. However, we are guaranteed that S will be valid when Done is high, therefore *this* arbiter will not cause other metastable states.

Bounded-time Synchronizer CANNOT BUILD
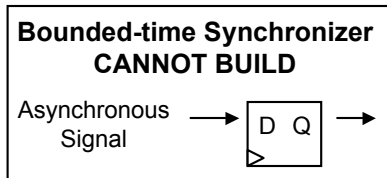
If the D input does not meet the flip flop's setup and hold time, the Q output might be metastable.

Bounded-time Combinational Logic CAN BUILD
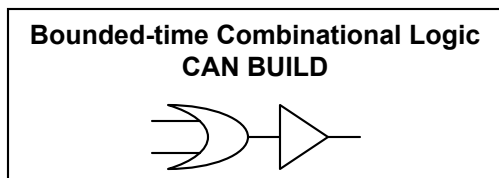
We are guaranteed a valid output after a specified propagation delay.

Unpipelined Circuits ────────────────

L = 1+6+2 = 9
T = 1 output / 9 = 1/9
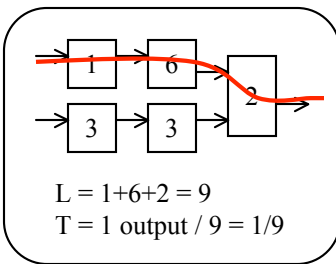
Latency = L = time an input takes to appear at the output

Throughput = T = #outputs / time

The circuit contains multiple blocks of combinational logic, each with a $t_{CD} = 0$ and a t_{PD} indicated by the number inside the box. The latency L of the circuit is the longest path from the input to the output. The entire circuit is combinational logic, so the worst-case throughput T is 1 output / L.

Pipelined Circuits ────────────────

L = 6*3 = 18
T = 1/6
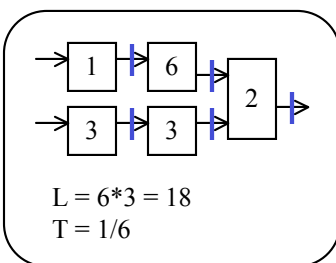
Pipelined circuits contain registers in-between the logic blocks. For simplicity, we assume that these are ideal registers with zero contamination and propagation delay, and zero setup and hold time. The register's clock cycle t_{CLK} is equal to the propagation delay of the slowest block.

Latency = L = K * $t_{PD,slowest\ stage}$ where K = # stages = # registers on each path

$$\text{Throughput} = T = \frac{1}{t_{PD,slowest\ stage}}$$

Pipelining a circuit typically adds latency and increases throughput. In the example above, the 6 ns clock cycle forces an input to spend 6 ns in every stage. Since there are three stages, the input spends 18 ns before its corresponding result appears at the output as opposed to the unpipelined circuit's 9 ns latency. However, pipelining the circuit ensures that a new result appears at the output every 6ns, thus increasing the throughput to 1/6.

Use the following rules of thumb to pipeline a circuit:
> 1) Put two dots on opposite sides of the circuit. Draw lines from one dot to another
> 2) Always put a register on the output
> 3) Isolate the slowest block with registers. Make this block the slowest stage for maximum throughput
> 4) Never cross a path twice with the same line. This helps ensure that each path has the same number of registers
> 5) Add a register every time a line crosses a wire.

1) Register on the output

2) Isolate the slowest block
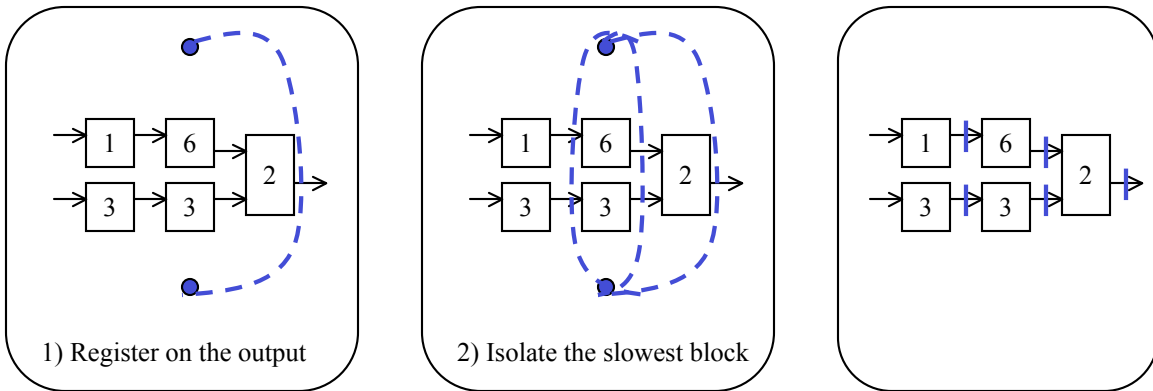
The example above is one of many ways to pipeline this circuit. Any configuration is valid as long as there are an equal number of registers on each path from input to output.

Assembly

Assembly is human-readable notation of machine language used in many computer architectures, including the simplified RISC (Reduced Instruction Set Computer) architecture discussed in this book. Some of the simple assembly operations will be discussed – arithmetic operations, load from memory, store to memory, jump, and branch.

Assembly code syntax takes many forms and this handbook discusses one possible implementation: OP(first input register, second input register, destination register). An operation OP will be performed on the two values are stored in the first and second input registers. The result of the operation will be written to the destination register. Each register is 32-bits wide and there is one register Rzero with value always zero.

Let us examine an arithmetic instruction and assume that the initial values stored in register 0, 1, and 2 are 0x23, 0x42, and 0x56 respectively. As illustrated below, after an add instruction like ADD(R0, R1, R2), the values of register 0 and 1 will be added, and the value will be written into register 2.

Register File				Register File	
R0	0x23			R0	0x23
R1	0x42	ADD(R0, R1, R2)		R1	0x42
R2	0x7			R2	0x65
...	...			...	...
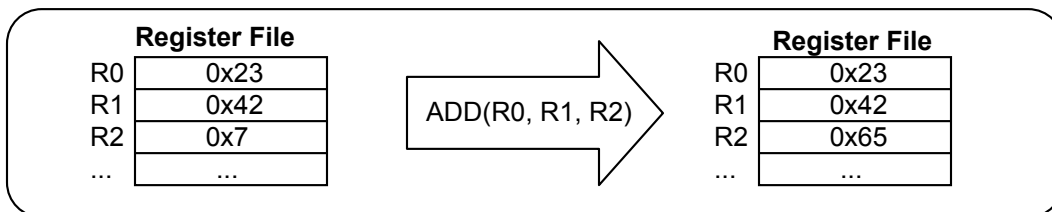
Arithmetic Operations

Arithmetic operations are often computed in the ALU (Arithmetic Logic Unit). Operations can be done with two input registers, like ADD(R0, R1, R2), or one input register and a 16-bit constant embedded in the instruction, like ADD(R0, 0x42, R2). The ALU can compute many different functions - mathematical operations like ADD (add), SUB (subtract), MUL (multiply), DIV (divide), logical operations like AND, OR, XOR, comparison operations like CMPEQ (equal), CMPLT (less than), CMPLE (less than or equal), or shift operations like SHL (shift left), SHR (shift right without sign extension), and SRA (shift right with sign extention).

Shift left SHL(R0, R1, R2) basically shifts the value in R0 left by n bits, where n is the value of register R1. The final shifted value is written into R2.

Register File				Register File	
R0	0x1			R0	0x1
R1	0x3	SHL(R0, R1, R2)		R1	0x3
R2	0x45			R2	0x8
...	...			...	...
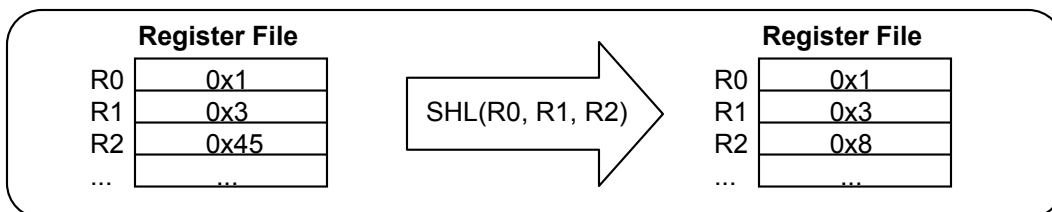
There are two types of shift right operations – one without sign extension (SHR), and one with sign extension (SRA). The sign bit is the most significant bit (MSB), or bit 31 in this instruction set. When shifting a value to the right, the most significant bits are either replaced with zero's if there is no sign extension, or is replaced with the old value of bit 31.

Register File

R0	0xF123
R1	0x4
R2	0x7
...	...

SHR(R0, R1, R2)

Register File

R0	0xF123
R1	0x4
R2	0x0F12
...	...

Register File

R0	0x8123
R1	0x4
R2	0x7
...	...

SRA(R0, R1, R2)

Register File

R0	0x8123
R1	0x4
R2	0xF812
...	...

Register File

R0	0x7123
R1	0x4
R2	0x7
...	...

SRA(R0, R1, R2)

Register File

R0	0x7123
R1	0x4
R2	0x0712
...	...
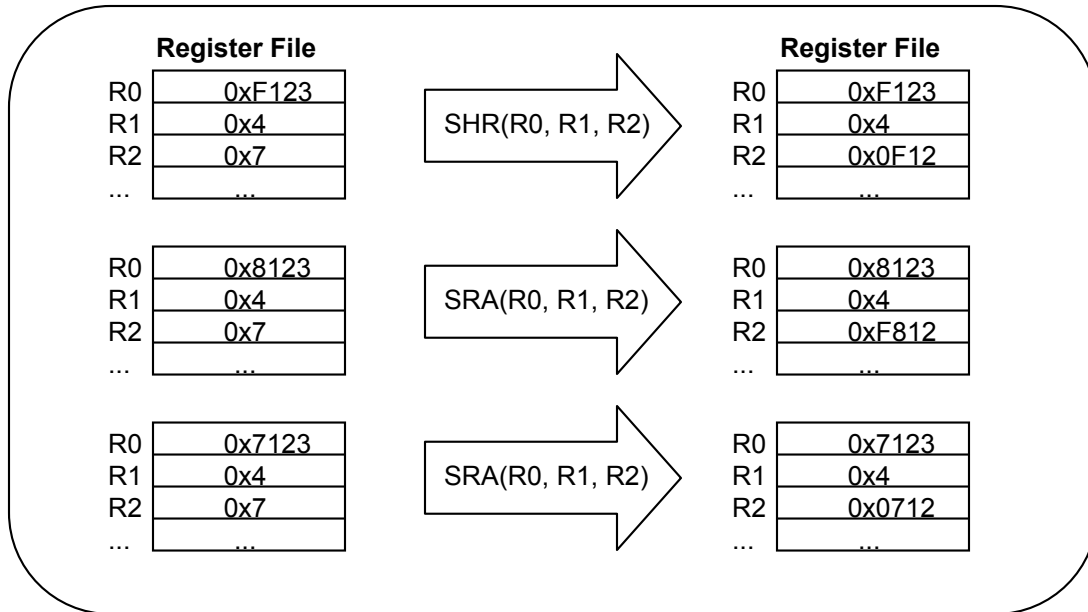
Load

A LOAD instruction loads a value from memory and stores it into a destination register: LOAD(address register, constant, destination register). The address for memory is calculated by adding the value in the address register with the constant.

Register File

R0	0x1
R1	0x3
R2	0x45
...	...

Data Memory

Addr	Value
...	
0x4	0x3
0x5	0x8
...	...

LD(R0, 0x4, R2)

Register File

R0	0x1
R1	0x3
R2	0x8
...	...

Data Memory

Addr	Value
...	
0x4	0x3
0x5	0x8
...	...
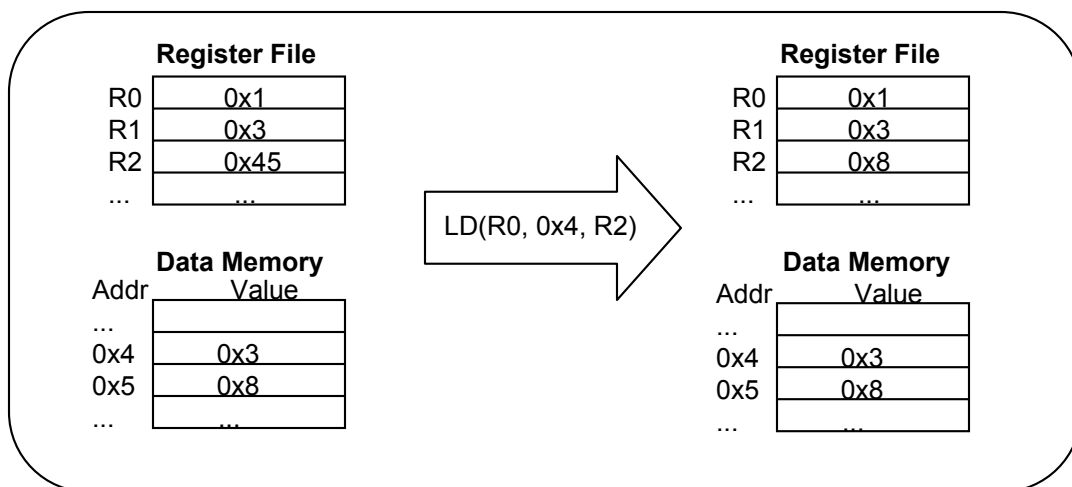
Store

A STORE instruction stores a register value to memory. STORE(value register, constant, address register). The address for memory is calculated by adding the value in the address register with the constant.
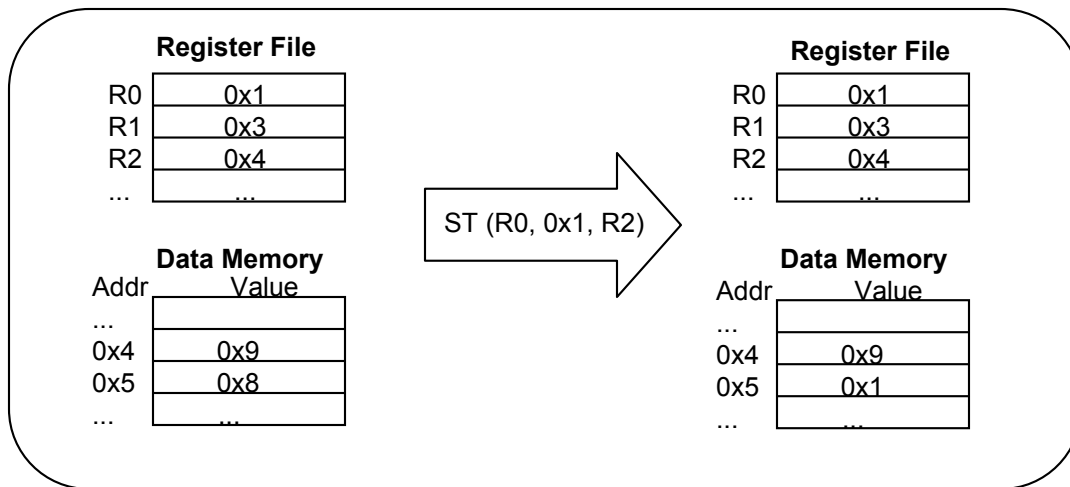
Branch

A branch instruction is an assembly version of an if-then-else construct. IF the value of the comparison register meets a certain criteria, THEN the computer will branch to the instruction indicated by the label, ELSE go to the next instruction in the sequence. If the operation does indeed branch, then a pointer will be stored in the instruction address destination register. Branch instructions include BEQ/BF (equal to zero) and BNE/BT (not equal to zero).

BEQ(comparison register, label, instruction address destination register).

For example, the following is an assembly implementation of "if (a=0) then b=2"

start: LD(Rzero, a, R0) // load value of a into register R0
        BEQ(a, btwo, R1) // if a=0, then branch to "btwo" and store instruction pointer in R1
...
btwo: ADDC(Rzero, 0x2, R0) // R0 = 2
        ST(R0, b, Rzero) // b = 2
...

RISC Processor Architecture

The simplified RISC (Reduced Instruction Set Computer) processor architecture described in this book consists of seven main parts: (1) fetch current instruction from Instruction Memory, (2) access the Register File and execute comparisons for branch instructions, (3) ALU, (4) Data Memory load and store, (5) write back to the register file, (6) next instruction address calculation, and (7) control logic. This book will explore the execution of some common operations.
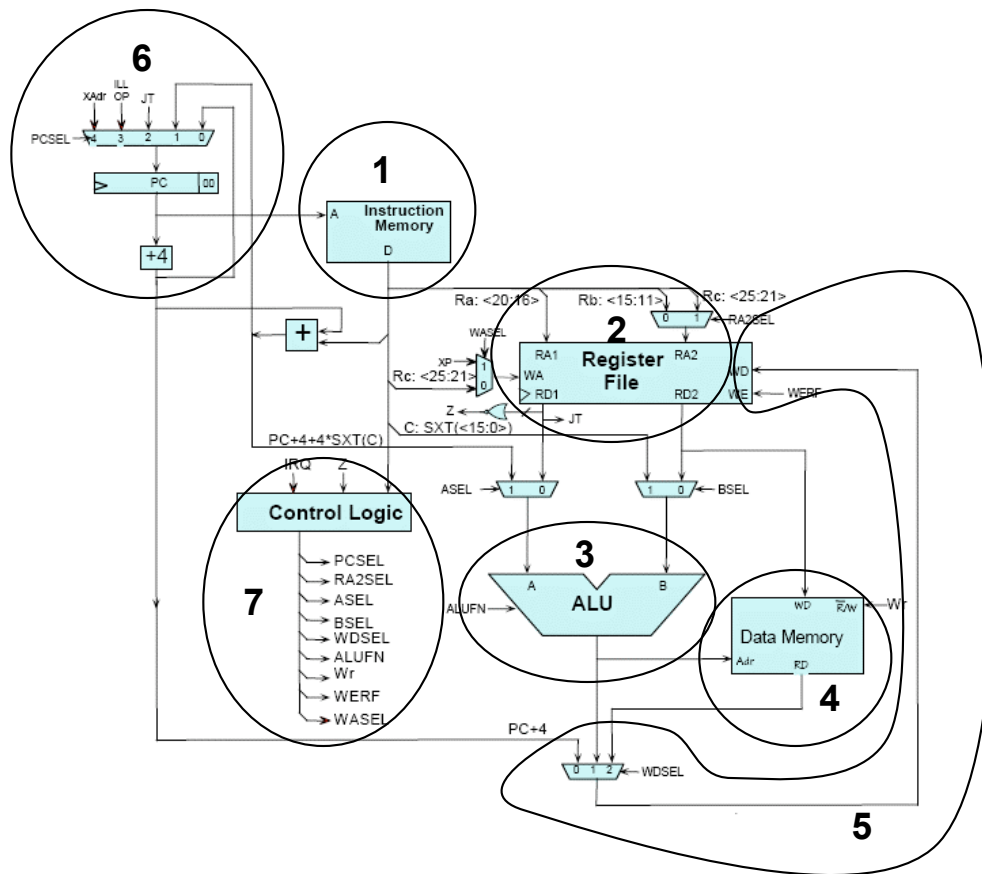
This figure is altered from an illustration taken from the MIT course 6.004 website http://6004.csail.mit.edu/

Arithmetic Operations

ADD (R0, R1, R2) would be executed in the RISC processor as follows.

1) A 32-bit instruction is fetched from the Instruction Memory

2) Register 0 and Register 1 are accessed in the register file, and the values of those registers are driven to the ALU

3) The ALU performs adds the values of Registers 0 and 1

4) Since this is not a LOAD or STORE instruction, the Data Memory is not accessed

5) The output value of the ALU is written to register 2

6) Since this is not an branch instruction, the instruction address is incremented by four.

For arithmetic operations that involve constants, like ADDC(R0, 0x1, R2), the constant 0x1 is embedded in the instruction memory output of stage 2, and the constant is bypassed to the ALU in the ASEL mux.
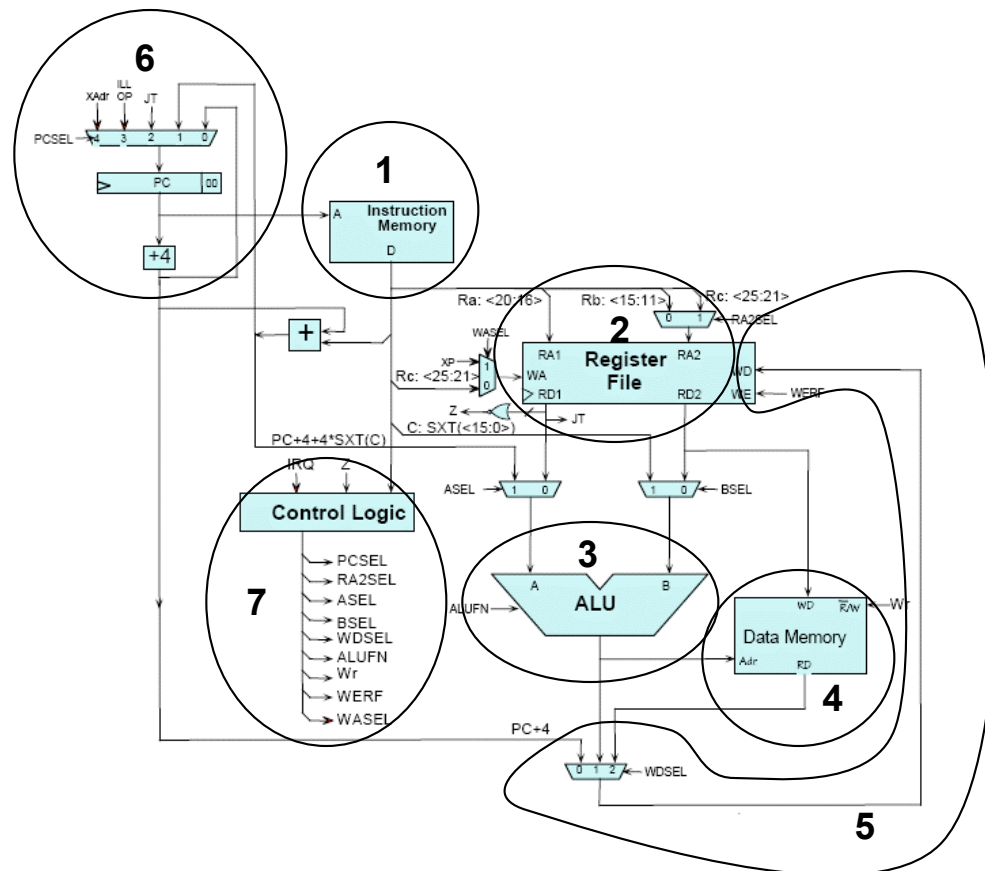
Load

LD (R0, 0x4, R2) would be executed in the RISC processor as follows.

1)   A 32-bit instruction is fetched from the Instruction Memory

2)   Register 0 is accessed in the Register File, and the value is driven to the ALU

3)   The ALU calculates the Data Memory address by adding the value of Register 0 and the constant 0x4

4)   The Data Memory is accessed and the data is output to the wdsel mux

5)   The contents of Data Memory are loaded into Register 2

6)   Since this is not an branch instruction, the instruction address is incremented by four.
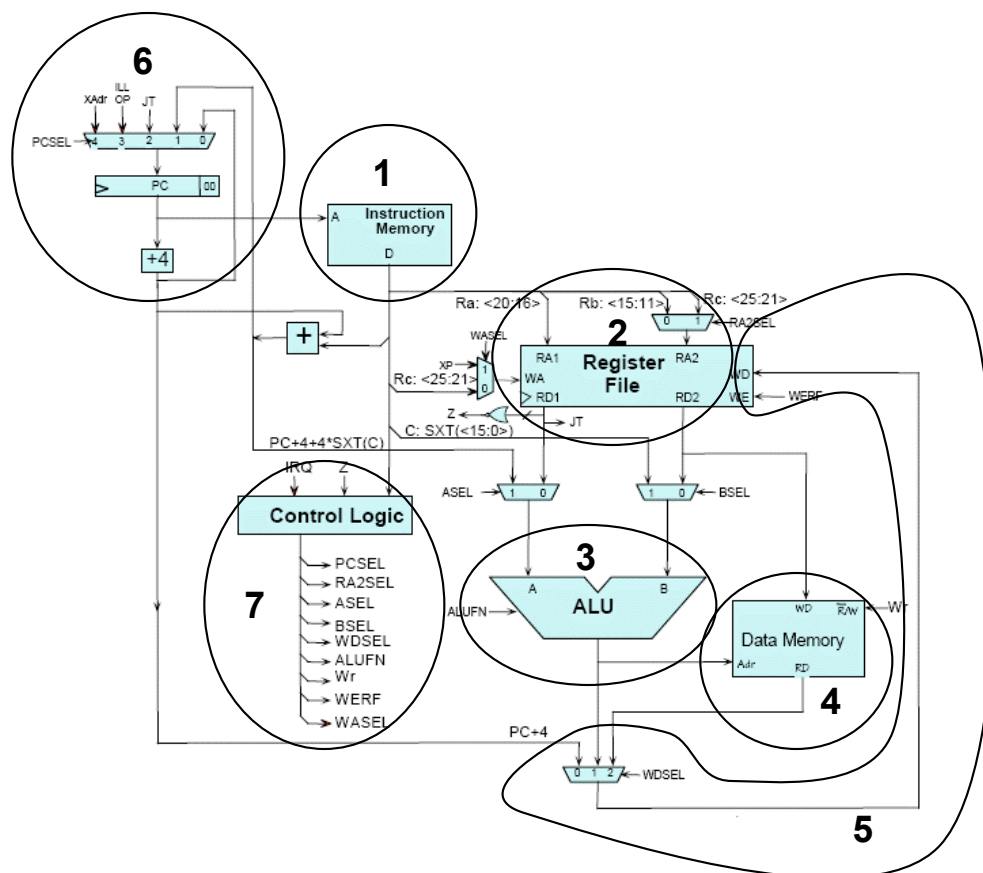
Store

ST (R0, 0x1, R2) would be executed in the RISC processor as follows.

1) A 32-bit instruction is fetched from the Instruction Memory

2) Register 0 is accessed in the register file, and the value is driven to Write Data (WD) port of Data Memory. Also, the value of Register 2 is driven to the ALU.

3) The ALU calculates the memory address by adding the value of Register 2 and the constant 0x1

4) The value of Register 0 is stored into Data Memory.

5) Nothing is written to the register file

6) Since this is not an branch instruction, the instruction address is incremented by four.
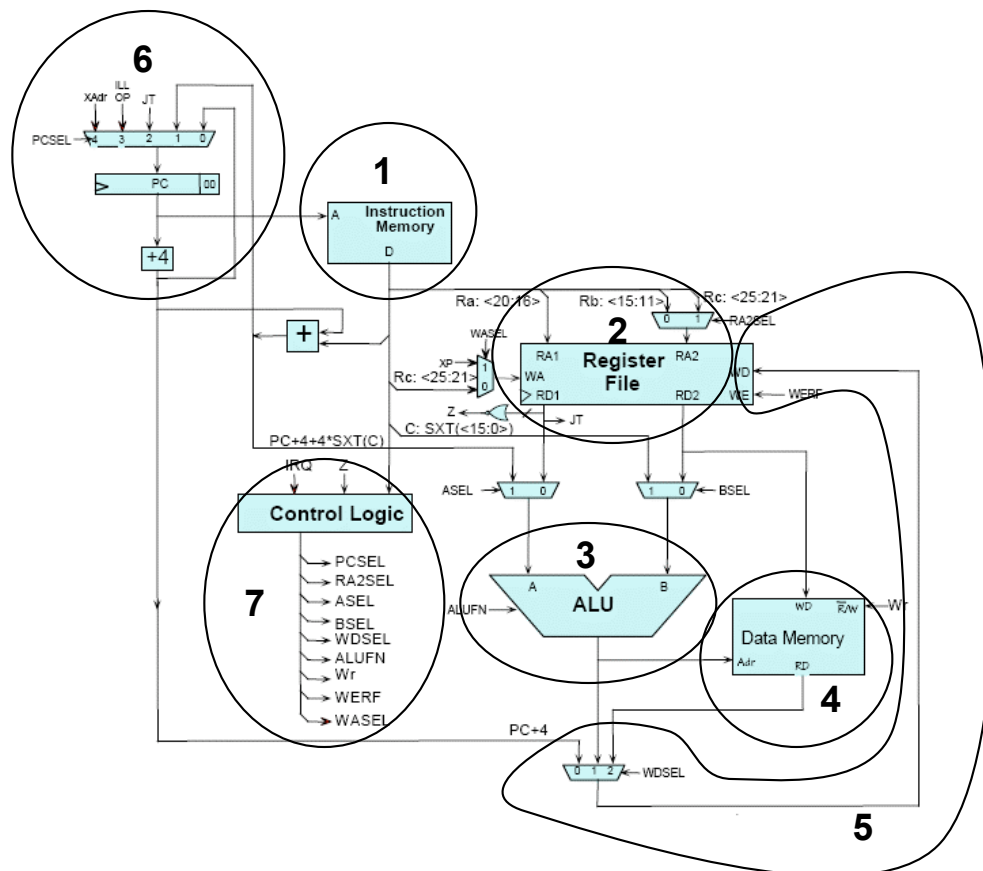
Branch

BEQ (R0, btwo, R1) would be executed in the RISC processor as follows.

1)   A 32-bit instruction is fetched from the Instruction Memory

2)   Register 0 is accessed in the register file, and a comparison is done to determine if the value of Register 0 is equal to zero $(Z = 1)$ or nonzero $(Z = 0)$

3)   If $Z = 1$, the ALU bypasses the instruction address to its output.

4)   Branch instructions do not access data memory.

5)   If $Z = 1$, then the ALU bypass is stored into Register 1.

6)   If $Z = 1$, the address "btwo" will be loaded ino the PC register, otherwise the instruction address is incremented by four
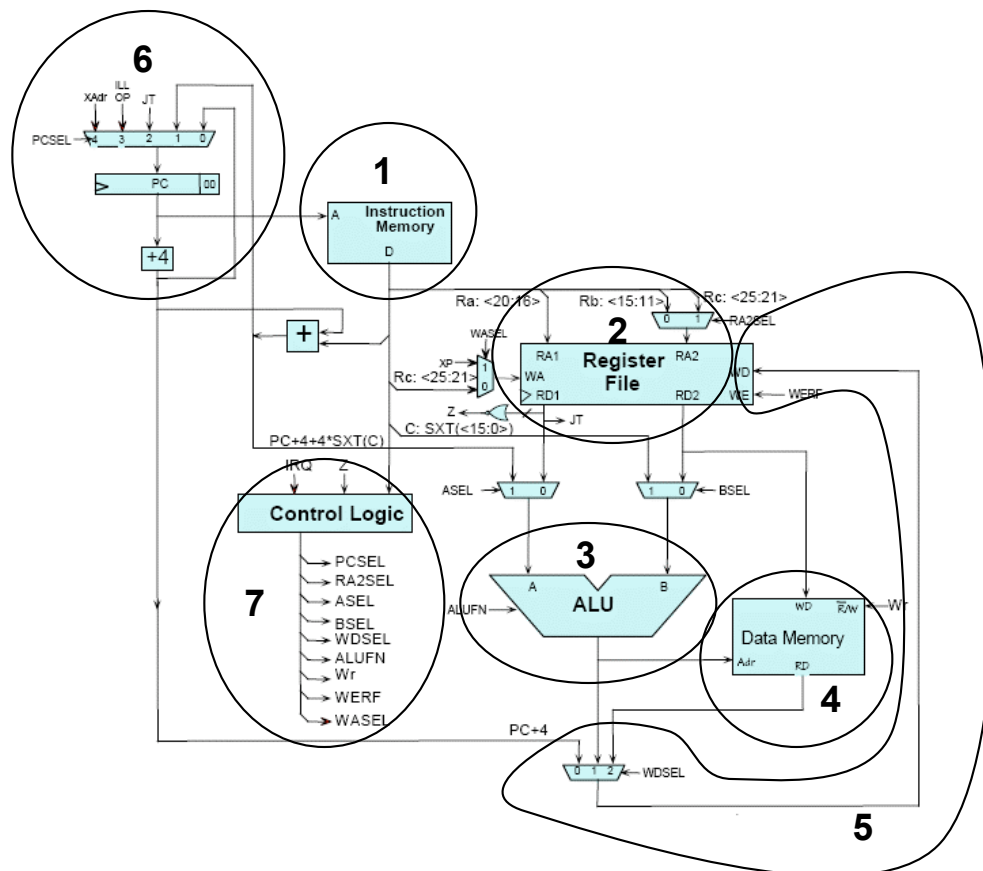
Intro to Stacks

A stack is is basically a stack of data located in memory. The bottom of the stack is located at a set address. The top of the stack is designated by the stack pointer (SP, Register 29), a register that contains the address of the next *unused* memory location.

MEMORY

0x0	Non-stack data	
0x100	data	← Bottom of stack
0x104	data	
0x108	data	← Top of stack
0x10C	unused	← SP
0x110	unused	
	⋮	
	Non-stack data	

Area of Memory Reserved For Stack

REGISTER FILE

R0	Random data
R1	Random data
	⋮
R29	0x10C
R30	Random data
R31	0

SP = Register 29

Note: The Beta is a 32-bit processor.
One byte = 8 bits
Each location in memory takes up 32 bits = 4 bytes.
Memory is byte-addressable.
Therefore, addresses increment by 4 as you go up the stack

Stack Macros

There are four stack macros you should be familiar with:

PUSH(Rx) (1) Push contents of register Rx on top of the stack
(2) SP ← SP + 4

POP(Rx) (1) Pop contents of register Rx on top of the stack
(2) SP ← SP + 4

ALLOCATE(k) (1) Reserve k lines: SP ← SP + 4*k

DEALLOCATE(k) (1) Release k lines: SP ← SP – 4*k

Note that PUSH and POP are each *two instruction* macros – one to push/pop between the register and the stack, and one to update the Stack Pointer (SP).

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	0x3
0x10C	unused ← SP
0x110	unused

PUSH(R0)

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	0x3
0x10C	**0x6**
0x110	unused ← **SP**

REGISTER FILE

R0	0x6
R1	0xF
R29	0x10C
R30	Random data
R31	0

REGISTER FILE

R0	0x6
R1	0xF
R29	**0x110**
R30	Random data
R31	0

1)   Contents of Register 0 (0x6) are pushed onto the top of the stack
2)   SP updated - 0x110 is the address of the top of the stack

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	0x3
0x10C	unused ← SP
0x110	unused

POP(R0)

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	unused ← **SP**
0x10C	unused
0x110	unused

REGISTER FILE

R0	0x6
R1	0xF
R29	0x10C
R30	Random data
R31	0

REGISTER FILE

R0	**0x3**
R1	0xF
R29	**0x108**
R30	Random data
R31	0

1)   Data at top of stack (0x3) is popped into Register 0
2)   SP updated - 0x108 is the address of the top of the stack

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	0x3
0x10C	unused ← SP
0x110	unused
	⋮

ALLOCATE(2)

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	0x3
0x10C	?
0x110	?
0x114	unused ← **SP**
	⋮

REGISTER FILE

R0	0x6
R1	0xF
	⋮
R29	0x10C
R30	Random data
R31	0

REGISTER FILE

R0	0x6
R1	0xF
	⋮
R29	**0x114**
R30	Random data
R31	0

1)   SP updated - 0x110 is the address of the top of the stack

STACK
(in memory)

0x100	0x1
0x104	0x2
0x108	0x3
0x10C	unused ← SP
0x110	unused
	⋮

DEALLOCATE(2)

STACK
(in memory)

0x100	0x1
0x104	unused ← SP
0x108	unused
0x10C	unused
0x110	unused
0x114	⋮

REGISTER FILE

R0	0x6
R1	0xF
	⋮
R29	0x10C
R30	Random data
R31	0

REGISTER FILE

R0	0x6
R1	0xF
	⋮
R29	**0x104**
R30	Random data
R31	0

1)   SP updated - 0x104 is the address of the top of the stack

Stack Frames

A portion of the STACK
(in memory)

...	
arg n	
…	
arg 0	
old old LP	Caller's Frame
old old BP	
local var. 0	← old BP
…	
local var. n	
arg n	
…	
arg 0	
old LP	Callee's Frame
old BP	
local var. 0	← BP
…	
local var. n	
unused	← SP
...	

Stack frames allow procedures to call other procedures (**caller** calls the **callee**), and prevent procedures from corrupting each other's data.

Two new registers are added to support Stack Frames. Register 27, Base Pointer **BP**, points to local variable 0 in the frame and is used to access the procedure's arguments and local variables. Register 28, Link Pointer **LP**, contains the return address of the caller. As stated in previous sections, Register 29 is the Stack Pointer **SP**.

A stack frame consists of (1) arguments pushed in reverse order, (2) the caller's link pointer, (3) the caller's base pointer, and (4) a bunch of local variables.

Two-level stack example

In this example, we have a caller **main** and a callee **f**. The initial state of the stack is pictured below. Main has been running for awhile and is about to call **f**. The program code arrow points to the line that is about to be executed. To conserve space, only relevant portions of the program code are shown in each snapshot.

Program code
(in memory)

```
Main:    ....
         ....
0xC00    PUSH(b)
0xC04    PUSH(a)
0xC08    BR(f, LP)
0xC0C    ....

f: ....
```

Stack
(in memory)

	...
0x100	Main's args
0x104	Main's LP
0x108	Main's caller's BP
0x10C	local var. 0	← BP
0x110	local var. 1
0x114	unused	← SP
....	...

Register File

R0	?
R1	?
	⋮
BP	0x10C
LP	0x004
SP	0x114
R30	?
R31	0

(1)   Initial State: Main is about to call helper procedure f
The code shown in Main is a typical calling sequence.

Program code
(in memory)

Stack
(in memory)

Register File

Main: ....
....
0xC00   PUSH(b)
0xC04   PUSH(a)
→ 0xC08   BR(f, LP)
0xC0C   ....

f: ....

	Stack	
	...	
0x100	Main's args	
0x104	Main's LP	
0x108	Main's caller's BP	
0x10C	local var. 0	← BP
0x110	local var. 1	
0x114	b	
0x118	a	
0x11C	unused	← **SP**
	...	

R0	?	
R1	?	
	⋮	
BP	0x10C	
LP	0x004	
SP	**0x11C**	
R30	?	
R31	0	

(2) Main pushed f's arguments onto the stack in reverse order

---

Program code
(in memory)

Stack
(in memory)

Register File

Main: ....
....
0xC00   PUSH(b)
0xC04   PUSH(a)
0xC08   BR(f, LP)
0xC0C   ....

→ f: ....

	Stack	
	...	
0x100	Main's args	
0x104	Main's LP	
0x108	Main's caller's BP	
0x10C	local var. 0	← BP
0x110	local var. 1	
0x114	b	
0x118	a	
0x11C	unused	← SP
	...	

R0	?	
R1	?	
	⋮	
BP	0x10C	
LP	**0xC0C**	
SP	0x11C	
R30	?	
R31	0	

(3) Main branched to f and stores return address in LP

Program code
(in memory)

Stack
(in memory)

Register File

Main: ....

f: ....
0xC00   PUSH(LP)
0xC04   PUSH(BP)
→ 0xC08   MOVE(SP,BP)
0xC0C   ALLOCATE(2)

	Stack	
	...	
0x100	Main's args	
0x104	Main's LP	
0x108	Main's caller's BP	
0x10C	local var. 0	← BP
0x110	local var. 1	
0x114	b	
0x118	a	
0x11C	**0xC0C**	
0x120	**0x10C**	
0x124	unused	← SP
	...	

R0	?	
R1	?	
	⋮	
BP	0x10C	
LP	0xC0C	
SP	**0x124**	
R30	?	
R31	0	

(4) f pushed the return address (LP) and Main's BP
The code shown in f is a typical entry sequence.

Program code
(in memory)

Main: ….

f: ….
0xC00 PUSH(LP)
0xC04 PUSH(BP)
0xC08 MOVE(SP,BP)
→ 0xC0C ALLOCATE(2)

Stack
(in memory)

	…
0x100	Main's args
0x104	Main's LP
0x108	Main's caller's BP
0x10C	local var. 0
0x110	local var. 1
0x114	b
0x118	a
0x11C	0xC0C
0x120	0x10C
0x124	unused
…	…

← BP
← SP

Register File

R0	?
R1	?
	⋮
BP	0x124
LP	0xC0C
SP	0x124
R30	?
R31	0

(5) f updated BP to reflect the current stack frame

Program code
(in memory)

Main: ….

f: ….
0xC00 PUSH(LP)
0xC04 PUSH(BP)
0xC08 MOVE(SP,BP)
0xC0C ALLOCATE(2)
→ 0xC10 …

Stack
(in memory)

	…
0x100	Main's args
0x104	Main's LP
0x108	Main's caller's BP
0x10C	local var. 0
0x110	local var. 1
0x114	b
0x118	a
0x11C	0xC0C
0x120	0x10C
0x124	reserved
0x128	reserved
0x12C	…

← BP
← SP

Register File

R0	?
R1	?
	⋮
BP	0x124
LP	0xC0C
SP	0x12C
R30	?
R31	0

(6) f allocated space for two local variables

Program code (in memory)

Main: ….

f: ….
→ 0xC30 MOVE(val, R0)
0xC34 MOVE(BP, SP)
0xC38 POP(BP)
0xC3C POP(LP)
0xC40 JMP(LP)

Stack (in memory)

Address	Value
	…
0x100	Main's args
0x104	Main's LP
0x108	Main's caller's BP
0x10C	local var. 0
0x110	local var. 1
0x114	b
0x118	a
0x11C	0xC0C
0x120	0x10C
0x124	Local var 0 ← BP
0x128	Local var 1
0x12C	… ← SP

Register File

R0	Z
R1	?
	⋮
BP	0x124
LP	?
SP	0x12C
R30	?
R31	0

(7) f did some computing and put the return value Z in R0

f loaded argument **a** into a register using: LD(BP, -12, Reg)
f loaded argument **b** into a register using: LD(BP, -16, Reg)
f loaded local variable 0 into a register using: LD(BP, 0, Reg)
f loaded local variable 1 into a register using: LD(BP, 4, Reg)

Program code (in memory)

Main: ….
0xC08 BR(f, LP)
→ 0xC0C ….

f: ….
0xC30 MOVE(val, R0)
0xC34 MOVE(BP, SP)
0xC38 POP(BP)
0xC3C POP(LP)
0xC40 JMP(LP)

Stack (in memory)

Address	Value
	…
0x100	Main's args
0x104	Main's LP
0x108	Main's caller's BP
0x10C	local var. 0 ← BP
0x110	local var. 1
0x114	b
0x118	a
0x11C	… ← SP

Register File

R0	Z
R1	?
	⋮
BP	0x10C
LP	0xC0C
SP	0x12C
R30	?
R31	0

(8) f executed a return sequence shown above:
   0xC30 - **Stored return value into memory**
   0xC34 - **Deallocated local variables by moving SP to BP**
   0xC38 - **Restored the BP**
   0xC3C - **Restored the LP**
   0xC40 - **Jumps to return address stored in LP**

Memory Hierarchy

Memory is one of the bottlenecks of a pipelined processor. We can alleviate this bottleneck with a hierarchy – a small fast cache (often notated as "$"), a larger slower main memory, and an enormous but *very* slow disk. Main memory has everything the cache has and more. Similarly, disk has everything contained in main memory and more.

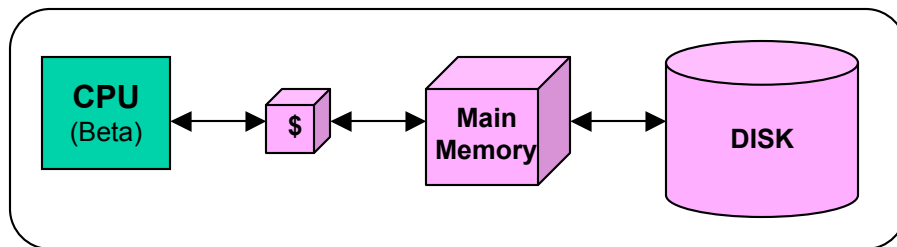When the CPU accesses a memory location A, it goes through these steps:

(1) *Check Cache* – If A is in the cache (cache hit), then return $DATA_A$, else go to step 2
(2) *Check Main Memory* – If A is in memory, then return $DATA_A$ and put it in the cache, else go to step 3.
(3) A will always be in disk. Put A in main memory and the cache, and return the value.

For simplicity, we will assume (for now) that we never have to go to disk – if there is a cache miss on A, we will always find A in main memory.

Caches are small, fast memory that contain temporary copies of selected memory locations. For 6.004, you can assume that the CPU runs at a clock speed where a cache hit on a memory access will return the cache data in one cycle, and a cache miss will cause the CPU to stall until the data is ready from memory. If the miss rate isn't too high, using a cache typically improve the average memory access time.

$$t_{ave} = t_{cache} + (\text{miss rate}) * t_{memory}$$

t_{cache} = time it takes to access the cache
t_{memory} = time it takes to access main memory

Locality

Caches are useful because they exploit locality in memory accesses. If you access memory location A, you're more likely to access A again in the near future. Every time there is a cache miss, that memory location will go in the cache and replace something that hasn't been used for awhile.

Fully Associative Cache
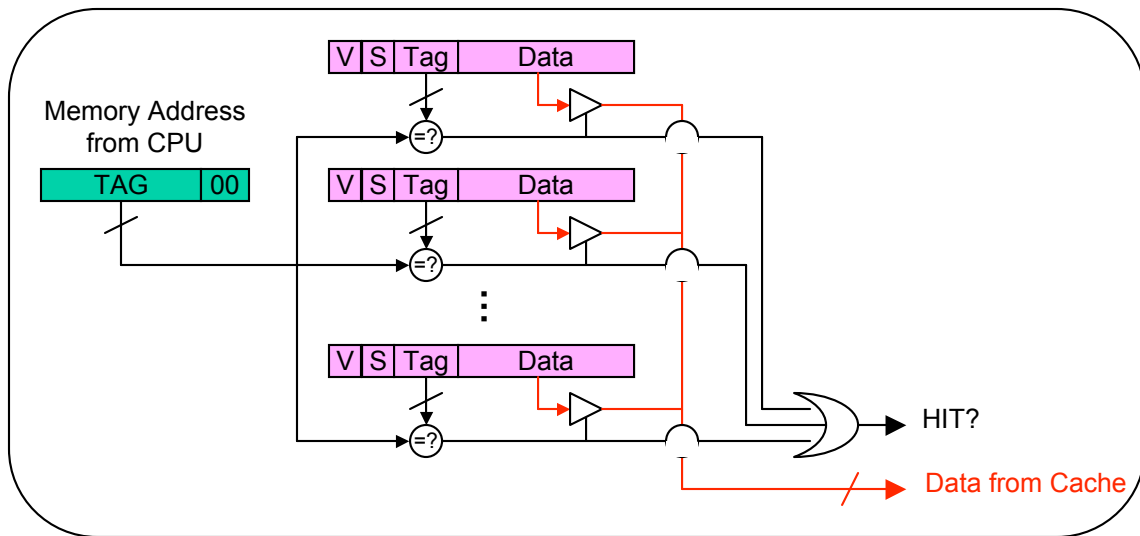
A cache contains many cache lines. A cache line contains (1) a valid bit indicating if the data in that line is valid, (2) optional status bits that indicate if the data is dirty, read-only, etc., (3) a tag formed from the memory address, and (4) the data corresponding to that address.

A fully-associative cache is one of the three cache architectures described in this handbook. It differs from other cache architectures in that it *can put data anywhere*.

When a memory address is presented to a fully associative cache, the address tag is compared to tags from all the valid lines in the cache. If there is a match, it's a cache hit and the data from that line is sent to the CPU. Otherwise it's a cache miss.

If there is a cache miss when the cache is full (all the lines are valid), we figure out which line to replace using one of the replacement policies outlined in a later section. If the line that is replaced contains valid and dirty data (V=1 and D=1), then we also have to write the data back to main memory. If it's not dirty, we just throw the line out.

Some useful equations regarding fully associative caches:

> Capacity = # lines * (1 valid bit + S status bits + T tag bits + D data bits)
>
> Typical values for the 6.004 BETA: T = 30 bits, D = 32 bits

The following example illustrates three memory accesses, starting with an empty fully associative cache. Each picture illustrates the state of the cache *after* the things in the commentary have executed.

Initial State – Empty Cache

1) **CPU tries to Load Address A from memory**
2) **Cache miss – stall the CPU while A is fetched from main memory**
3) **Put A and ADATA in the cache and return ADATA to the CPU**
4) **Cache state at this point in time shown above**

42

1) CPU tries to Load Address B from memory
2) Cache miss – stall the CPU while we get B from main memory
3) Put B and BDATA in the cache and return BDATA to the CPU

1) CPU tries to Load Address B from memory
2) Cache hit – return BDATA to the CPU
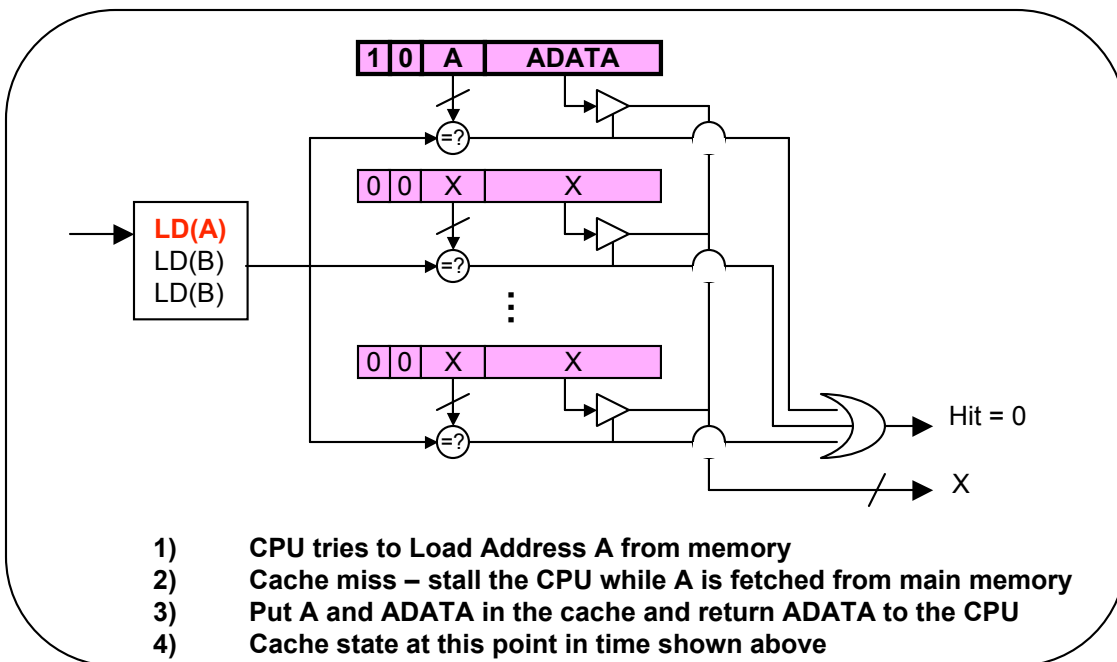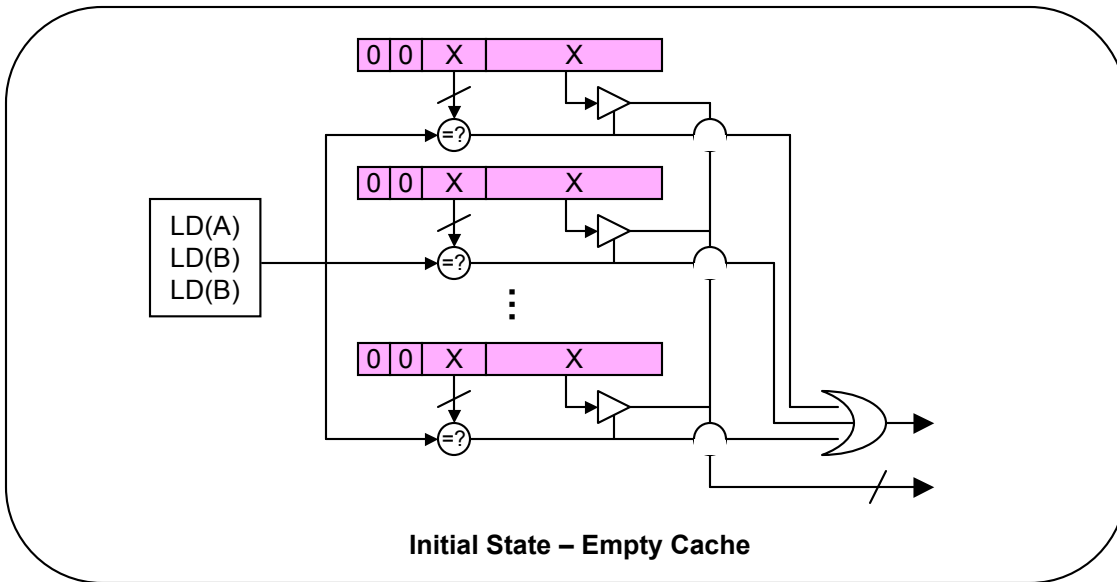
Direct-Mapped Cache

The direct-mapped cache is cheaper than the fully-associative cache because it only has one tag comparator for the entire cache, as opposed to one tag comparator for every line in a fully-associative cache. However, address collisions are possible in a direct-mapped cache, which might adversely affect performance.

When a memory address is presented, the index bits will select the cache line. If the tag from that cache line is equal to the tag from the memory address, it's a cache hit. The data from that cache line will be sent to the CPU.

If there is a cache miss and we have to go to memory, the data from memory replaces the cache line indicated by the address index. If the line being replaced is valid and dirty data, the cache must write that data to memory rather than throw it out.

Which cache should you use? It depends on the memory access pattern. Different patterns and replacement strategies are more optimal for different caches.
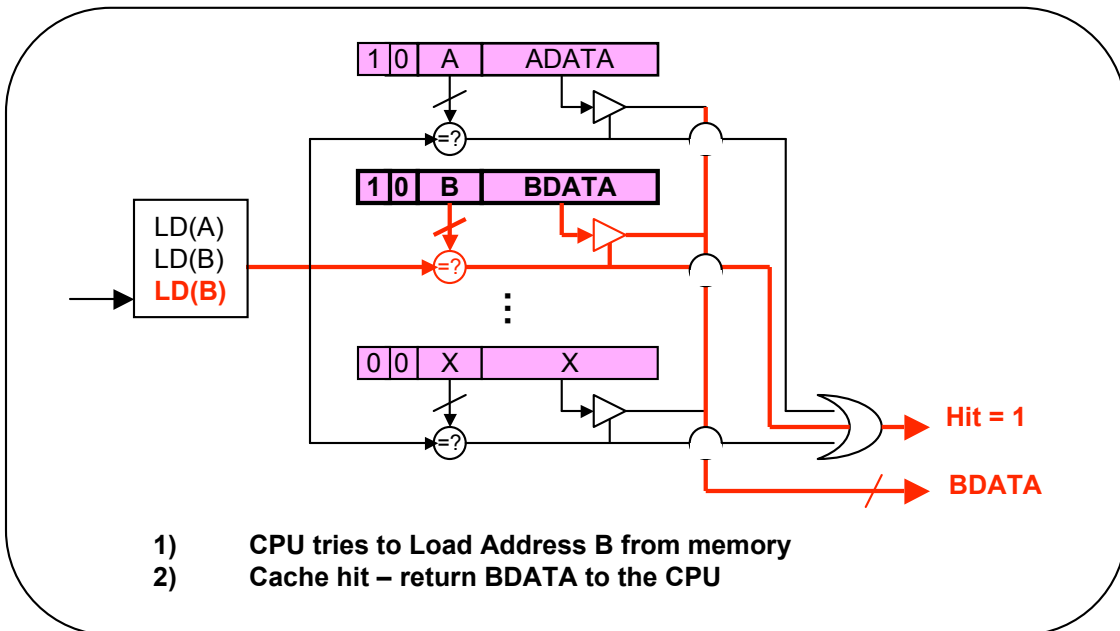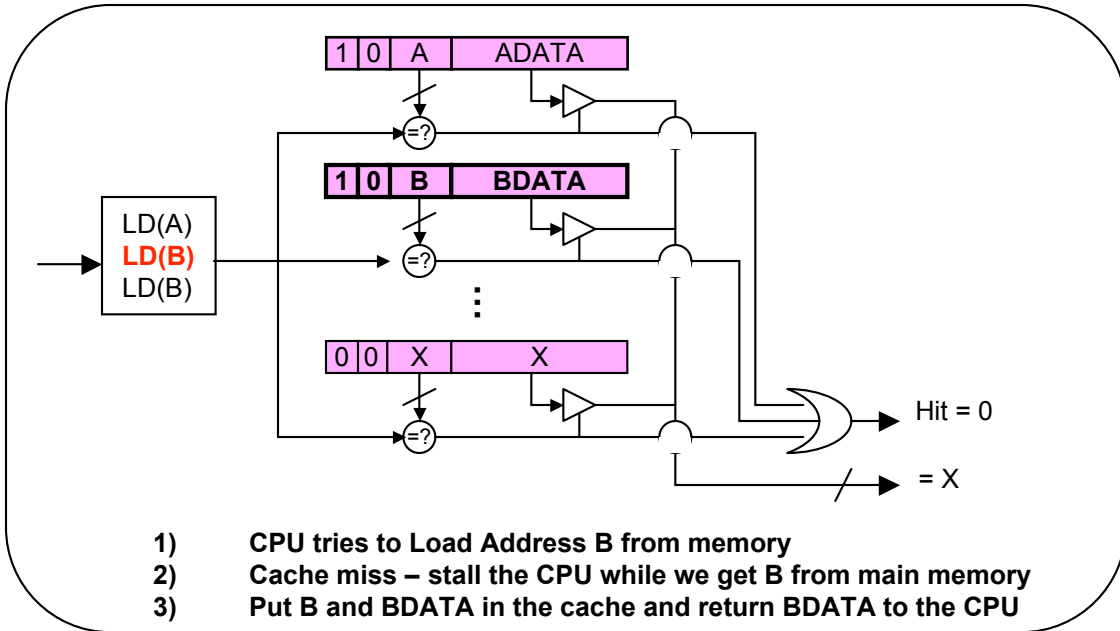
Here are some useful equations regarding direct-mapped caches:

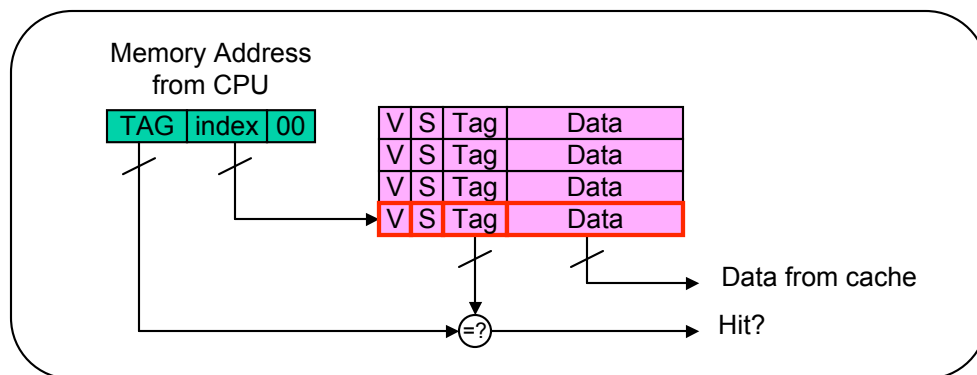Capacity = # lines * (1 valid bit + S status bit + T tag bits + D data bits)

\# lines = $2^{(\# \text{ index bits})}$

The following example illustrates four memory accesses, starting with an empty direct mapped cache. Each picture illustrates the state of the cache *after* the commentary is executed.

0	0	X	X	00
0	0	X	X	01
0	0	X	X	10
0	0	X	X	11

Data from cache

=? Hit?

Initial State – Empty direct-mapped cache

001	01	00

0	0	X	X	00
1	0	001	$DATA_{0010100}$	01
0	0	X	X	10
0	0	X	X	11

=?

1) **LD(0010100) → $tag_{address}$ = 001, index = 01**
2) **Cache line 01 is not valid, therefore this is a cache miss**
3) **Stall the CPU to find data in main memory**
4) **Store $DATA_{0010100}$ in cache line 01**

001	10	00

0	0	X	X	00
1	0	001	$DATA_{0010100}$	01
1	0	001	$DATA_{0011000}$	10
0	0	X	X	11

=?

1) **LD(0011000) → $tag_{address}$ = 001, index = 10**
2) **Cache line 10 is not valid, therefore this is a cache miss**
3) **Stall the CPU to find data in main memory**
4) **Store $DATA_{0011000}$ in cache line 00**

010	10	00

0	0	X	X	00
1	0	001	DATA $_{0010100}$	01
1	0	010	DATA $_{0101000}$	10
0	0	X	X	11

=?

1) LD(0101000) → $tag_{address}$ = 010, index = 10
2) Cache line 10 is valid
3) Address tag and cache line tag do not match → miss!
4) Stall the CPU to find data in memory
5) Dirty bit is zero, no need to write DATA$_{0011000}$ to memory
6) Store DATA$_{0101000}$ in cache line 10

N-way Set-Associative Cache

A N-way set associative cache is made up of N ways (N direct-mapped caches). Data can be stored in any one of those ways.

The set-associative cache is a tradeoff between a fully-associative and a direct-mapped cache. A set-associative cache does not have a lot of expensive tags like a fully-associative cache. A set-associative cache also cuts down on address collisions by allowing data to be stored in N cache lines rather than one cache line provided by a direct-mapped cache.

When a memory address is presented to the set-associative cache, the index bits select a cache line in all the ways. If the address tag matches any of the tags in the selected cache lines, it's a cache hit. The data from that cache line will be sent to the CPU.

If there is a cache miss and we have to go to memory, the data from memory replaces a cache line in one of the direct-mapped caches. If all the selected cache lines are valid, the cache uses a replacement policy to determine which should be replaced. Also, if the line being replaced has valid and dirty data, the cache must write that data to memory.

A set-associative cache has its advantages. However, that does not mean that the set-associative cache will always perform better than fully-associative or direct-mapped caches of the same capacity. As stated before, different patterns and replacement strategies are more optimal for different caches.

Here are some useful equations regarding set-associative caches:

Capacity = (# ways) * (# lines/way) * (1 valid bit + S status bits + T tag bits + D data bits)

lines = $2^{(\# \text{ index bits})}$

The following example illustrates three memory accesses, starting with an empty 2-way set-associative cache. Each picture illustrates the state of the cache *after* the things in the commentary have executed.

Initial State – Empty 2-way Set Associative Cache

1)   **LD(0010100) → $tag_{address}$ = 001, index = 01**
2)   **Cache line 01 is invalid in both ways. Cache miss**
3)   **Stall the CPU to find data in main memory**
4)   **Store $D_{0010100}$ in cache line 01 in a way**

First diagram (top):

Address: `010` | `01` | `00`

Way 0:
0	0	X	X
1	0	001	$D_{0010100}$
0	0	X	X
0	0	X	X

Way 1:
0	0	X	X
1	0	010	$D_{0100100}$
0	0	X	X
0	0	X	X

=? =?

1) LD(0100100) → $tag_{address}$ = 010, index = 01
2) Cache line 01 is invalid in both ways. Cache miss
3) Stall the CPU to find data in main memory
4) Store $D_{0010100}$ in cache line 01 in the other way

Second diagram (bottom):

Address: `001` | `01` | `00`

Way 0:
0	0	X	X
1	0	001	$D_{0010100}$
0	0	X	X
0	0	X	X

Way 1:
0	0	X	X
1	0	010	$D_{0100100}$
0	0	X	X
0	0	X	X

=? =?

$D_{0010100}$

1

1) LD(0010100) → $tag_{address}$ = 001, index = 01
2) Cache Hit in first way
3) Return $D_{0010100}$ to CPU

Locality and Data Block Size

If you access memory location A then you're more likely to access locations near A in the future. This applies to both data and instruction memory accesses.
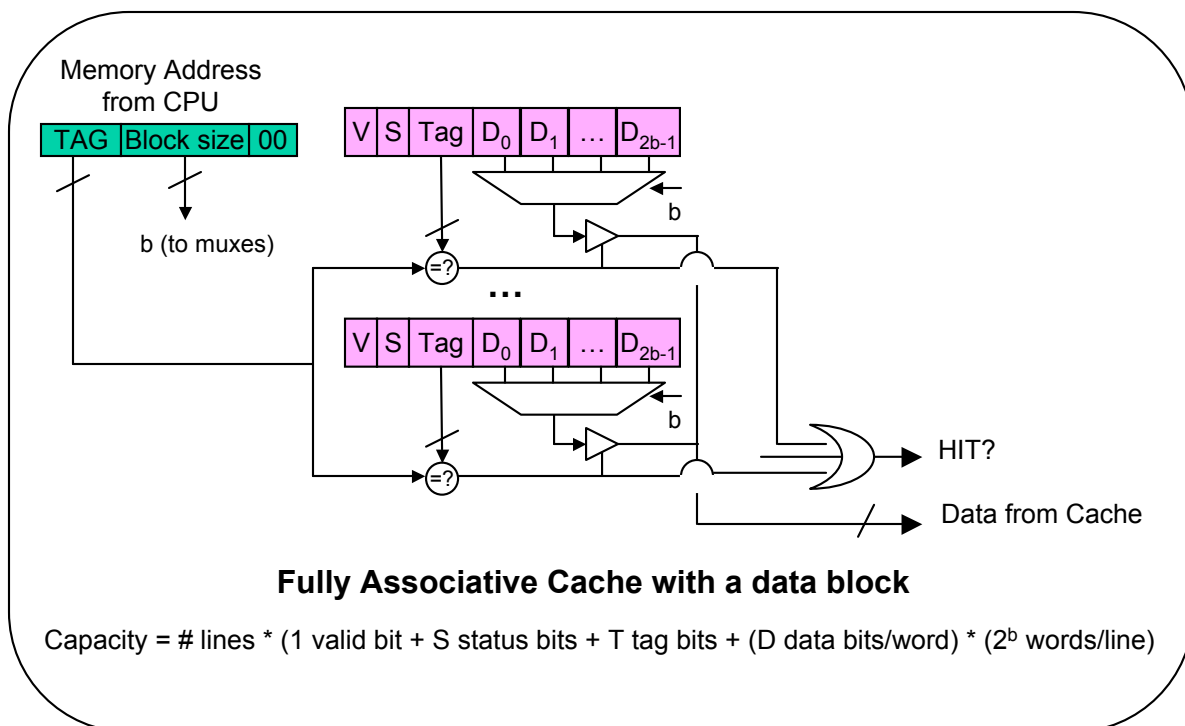
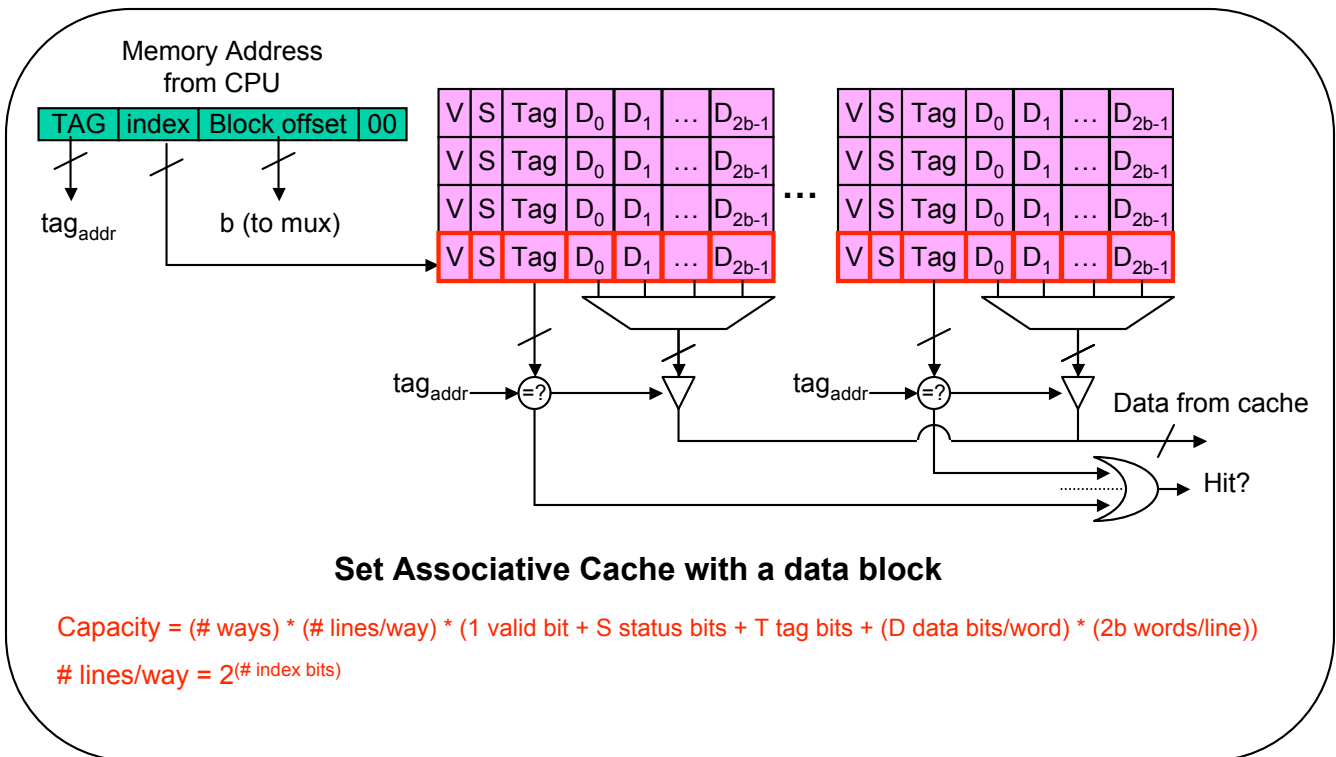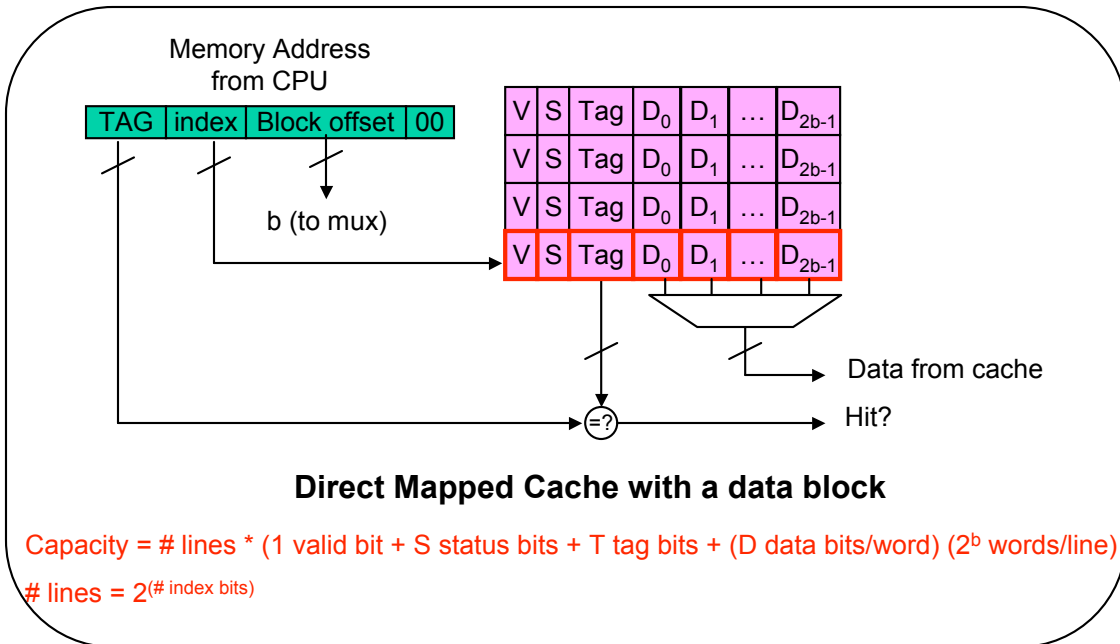***Example involving instruction accesses:***
CPU instructions are usually executed sequentially through memory (with the exceptions of jumps and branches). If you fetch an instruction from memory location A, you will likely need to fetch instructions from A+4, A+8, A+12, etc. in the near future.

Caches can take advantage of locality by increasing the data block size. Whenever there is a cache miss on memory location A, the cache will put a block of memory surrounding and including A into the cache.

There are some tradeoffs when it comes to block sizes (assuming a fixed cache capacity).

Advantages of a larger block size	Disadvantages of a larger block size
Miss rate decreases if the program exhibits locality	More address collisions → miss rate increases
	Miss penalty increases (more data is moved from memory to the cache)
	Unnecessary words fetched

Fully Associative Cache with a data block

Capacity = # lines * (1 valid bit + S status bits + T tag bits + (D data bits/word) * (2^b words/line)

Memory Address
from CPU

| TAG | index | Block offset | 00 |

b (to mux)

| V | S | Tag | D_0 | D_1 | ... | D_{2b-1} |

Data from cache

=? Hit?

Direct Mapped Cache with a data block

Capacity = # lines * (1 valid bit + S status bits + T tag bits + (D data bits/word) (2^b words/line)

lines = $2^{(\text{# index bits})}$

Memory Address
from CPU

| TAG | index | Block offset | 00 |

tag_{addr} b (to mux)

| V | S | Tag | D_0 | D_1 | ... | D_{2b-1} |

...

tag_{addr} →=? tag_{addr} →=?   Data from cache

Hit?

Set Associative Cache with a data block

Capacity = (# ways) * (# lines/way) * (1 valid bit + S status bits + T tag bits + (D data bits/word) * (2b words/line))

lines/way = $2^{(\text{# index bits})}$

Replacement Policies ━━━━━━━━━━━━━━━━━━━━━━

Fully-associative and set-associative caches need a replacement policy to decide which cache line to replace when all options are valid. Three common replacement strategies include LRU (Least recently used), FIFO (first-in, first-out), and Random.

LRU replaces the least recently used location in the cache. LRU makes sense from a locality point of view – if certain lines have been used more recently, they are more likely to be used in the future. LRU is rather costly, however, and is susceptible to memory patterns that cause worst-case cache performance.

FIFO replaces the oldest item in the cache. FIFO is much cheaper and easier to implement than LRU, but is also susceptible to worst-case scenario memory patterns that cause worst-case cache performance.

Random will randomly select a line to be replaced. Random generators that produce a uniform distribution are difficult to build, possibly less susceptible to worst-case scenarios.

Write Policies ━━━━━━━━━━━━━━━━━━━━━━━━━━

When a cache line is replaced or being written, that line needs to be written to memory at some point. Three common write policies include write through, write behind, and write back.

Write through – every time the CPU writes to memory, write to both the cache and main memory. There are no dirty bits in write through. An advantage to write through is that the contents of main memory will never be stale. A disadvantage is a sacrifice in performance from stalling the CPU on every write to memory. For example, a lot of programs contain functions that repeatedly write to the same variable and then return the final result. All we need is for the final result to be stored in main memory. Storing the intermediate results in main memory might be unnecessary.

Write behind – every time the CPU writes to memory, write to the cache and the write buffer. The CPU then proceeds to the next instruction while the write buffer writes to main memory in the background. There are no dirty bits in write behind. This usually results in better performance than write-through, but it is still possible for the write buffer to unnecessarily write things to memory.

Write-back – every time the CPU writes to memory, the cache will set the dirty bit to one. On a cache miss, if the cache line being replaced has a dirty bit of one, then that line will be written to memory. Write back performs well because the cache writes to memory only when it is absolutely necessary. However, write back sacrifices some area from storing a dirty bit in each cache line.
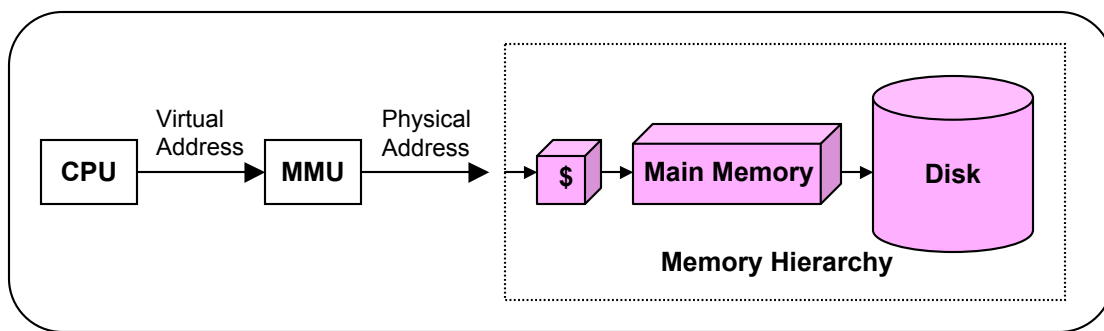
Virtual Memory

Programs that are run on the CPU commonly use *virtual* addresses rather than physical addresses when accessing memory. The CPU's virtual address is translated by the MMU (memory management unit) into a physical address that accesses the memory hierarchy.

Physical memory and virtual memory are both divided into sections called pages. Physical pages have physical page numbers (PPN) and virtual pages have virtual page numbers (VPN).

The virtual address that is inputted to the MMU contains a virtual page number and a page offset. The MMU translates the virtual page number into the appropriate physical page number via a lookup table. The MMU does not touch the page offset.

The output of the MMU is a physical address made up of a physical page number and an offset. The physical page number points to a page in memory, while the offset indicates which word to access on that page.

Using virtual memory has numerous advantages over directly addressing physical memory. Some (not all) of these advantages are discussed below.

Virtual memory separates hardware from software, thus allowing software to be written without being constrained by the size of physical memory. This allows programs to use a huge a virtual memory space that's larger than physical memory and split up the jobs between multiple computers. It also allows machines to have more physical memory than they have in virtual memory.
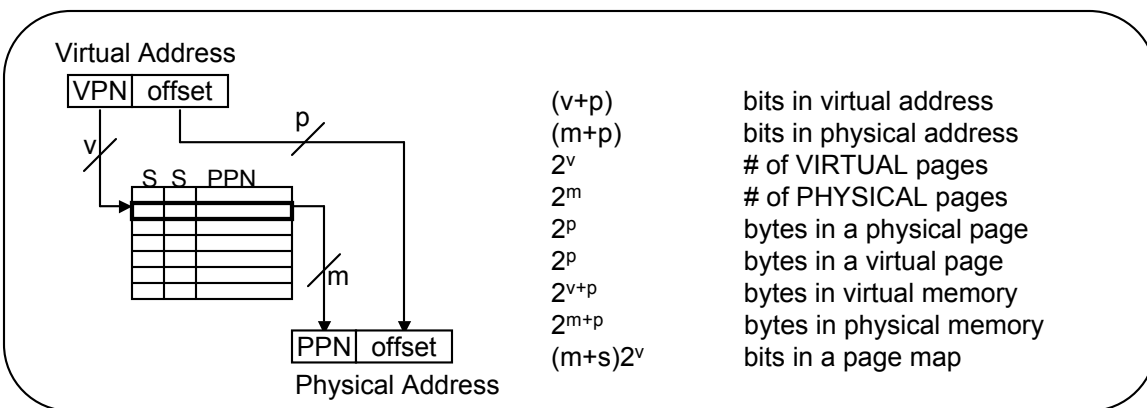
Virtual memory allows different programs to be isolated from one another, and allows the operating system to isolate itself from other software. It also allows programs and their data to be rearranged in physical memory by the operating system in a way that's transparent to the program.

Linear Page Table (one-level page map) ——————————

A page table is basically a list of PPNs somewhere in memory. The virtual address's VPN selects which PTE (page table entry) to use. The PPN in that PTE is concatenated with the offset to form a physical address.

A PTE contains optional status bits (dirty, resident, read-only, supervisor, etc.), and a PPN (physical page number).
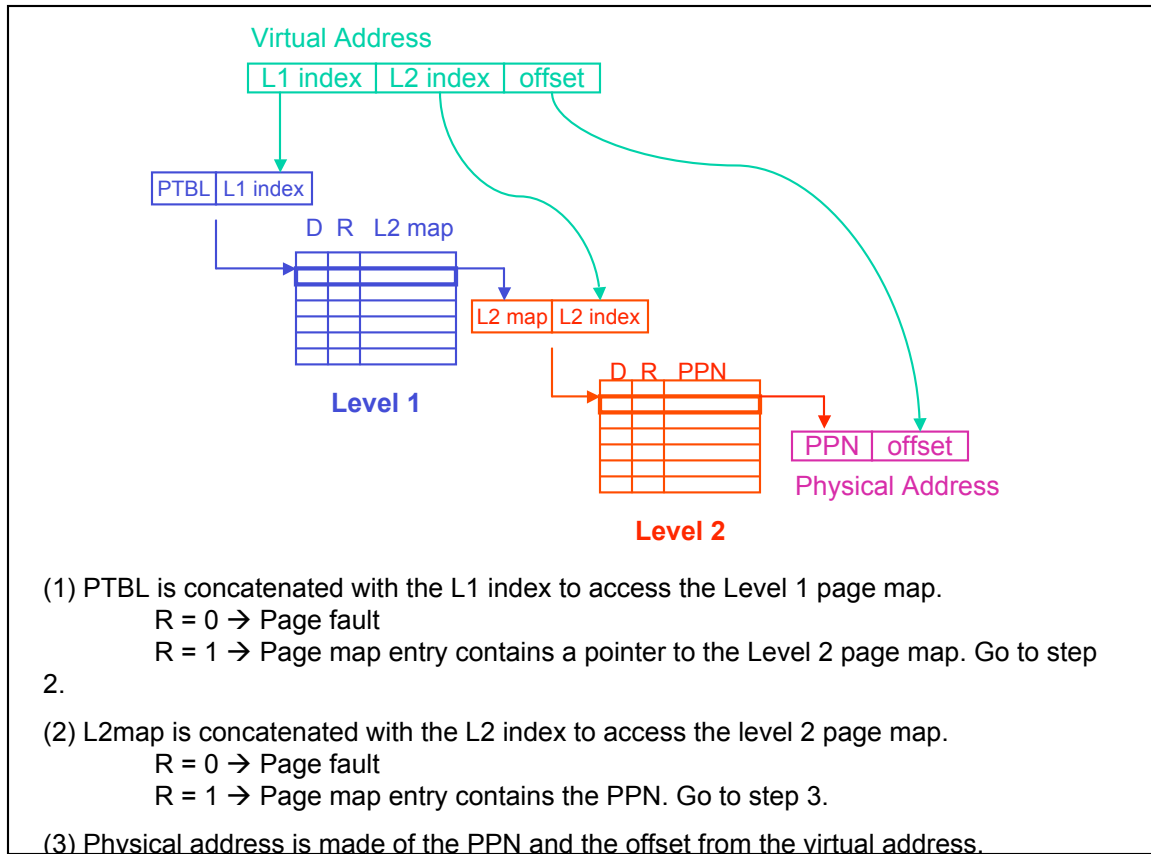
A dirty bit says the page has been changed since it was loaded from disk, and therefore needs to be written to disk when it is replaced. The resident bit is set to one for pages that are in main memory, and zero (page fault) it the page is unallocated or on disk.

Virtual Address
VPN | offset

p

v

S S PPN

m

PPN | offset
Physical Address

$(v+p)$	bits in virtual address
$(m+p)$	bits in physical address
2^v	# of VIRTUAL pages
2^m	# of PHYSICAL pages
2^p	bytes in a physical page
2^p	bytes in a virtual page
2^{v+p}	bytes in virtual memory
2^{m+p}	bytes in physical memory
$(m+s)2^v$	bits in a page map

Hierarchical Page Map ——————————

In a linear page table (one-level page table), we often have a situation where a lot of memory is unused, resulting in a lot of wasted space. One possible way to alleviate the waste of space is to use a hierarchical page map instead of a linear page table. The diagram below illustrates a two-level page map containing one Level 1 page table, and $2^{L1index}$ possible Level 2 page tables.

Multiple unused PTEs in a one-level page map can be equivalent to one unused Level 1 PTEs in a hierarchical page map. This significant savings in memory, however, comes at a cost in performance since page maps are usually stored in physical memory. A linear page table walk takes one memory access while a two-level walk takes two memory accesses.

(1) PTBL is concatenated with the L1 index to access the Level 1 page map.
  R = 0 → Page fault
  R = 1 → Page map entry contains a pointer to the Level 2 page map. Go to step 2.

(2) L2map is concatenated with the L2 index to access the level 2 page map.
  R = 0 → Page fault
  R = 1 → Page map entry contains the PPN. Go to step 3.

(3) Physical address is made of the PPN and the offset from the virtual address.

Translation Look-aside Buffer (TLB)

The TLB is a small, usually fully-associative cache that contains page map entries that translate VPNs to PPNs. A TLB miss can severely hinder performance because the page table walk can result in multiple accesses to memory or a page fault.

Caches – Virtual or Physical addresses

Caches that work with physical addresses might not be as advantageous if the we access main memory during the page table walk. Caches that work with virtual addresses might perform better, but this brings up issues that are beyond the scope of 6.004. Take 6.823 if you want to learn about these issues.

MMU Overview

The MMU goes through the following steps when it is presented with a virtual address:

1) Check the TLB (Translation Look-aside Buffer). If there is a hit, send the physical address to memory. If not, go to step 2.

2) Page Table Walk – If the page tables and final physical page reside in memory *(not in disk)*, then update the TLB and send the physical address to memory. However, if one of the page tables or the final physical page is not resident in memory, go to step 3.

3) Page Fault – a page table or final physical page resides in disk. Handled in software.
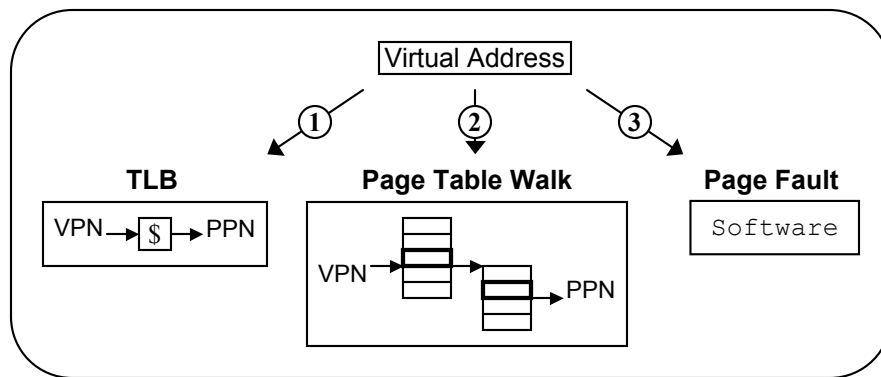
References

MIT 6.004 Lecture Notes from Fall 2000 and Spring 2003

MIT 6.823 Lecture Notes from Spring 2003